

## Course Guideline

### Previous knowledge Requirements:

- Java programming

### Course goals:

- Explain and compare the structure and properties of standard algorithms and data structures.
- Execute and visualize standard algorithms and data structures on given inputs.
- Use mathematical methods to analyze the time and space complexity of algorithms and data structures.
- Implement algorithms and data structures using the Java programming language.
- Solve programming tasks using standard algorithms and data structures.

### Course content:

1. Data Structures
  - a. data containers
    - i. vector
    - ii. list
    - iii. tree
    - iv. set
  - b. ordered data structures
    - i. stack
    - ii. queue
    - iii. priority queue
    - iv. heap
    - v. map
  - c. operations on data structures
    - i. iterative implementations
    - ii. recursive implementations
2. Sorting
  - d. selection sort
  - e. insertion sort
  - f. heap sort
  - g. merge sort
  - h. quick sort
  - i. bucket sort
  - j. radix sort
3. Searching
  - k. search structures
    - i. search trees
    - ii. AVL trees
    - iii. (2,4) trees
  - l. backtracking
4. Graphs and Graph Algorithms
  - m. graph data structures
    - i. directed graphs
    - ii. undirected graphs
    - iii. weights
    - iv. representations
  - n. graph algorithms
    - i. graph traversals
    - ii. path finding
    - iii. cycle finding
    - iv. connectivity
    - v. topological ordering
    - vi. shortest path
    - vii. minimum spanning tree

Lecture Topics Order:

- Big-Oh notation
- Complexity in recursive methods
- Arrays & LinkedLists
- Stacks & Queues
- Dynamic arrays
- Positional lists
- Iterators
- Trees
- Priority Queues
- Sorting
- Key-based sorting
- Maps & hashing
- Search Trees
- Graph properties
- Graph algorithms

## Table of Contents

Week 1. Introduction to Algorithms and Data Structures.....	1	4.4.1 By Example.....	15
Complexity analysis and big-Oh notation [Sections 4.1, 4.2, 4.3].....	6	4.4.2 The Contra Attack .....	15
4.1 Experimental Studies .....	6	4.4.3 Induction and Loop Invariantes .....	15
4.2 The seven functions .....	8	Lecture 1 & 2 & Exercises .....	16
The constant function .....	8	Counting primitives (raw vs constants) operations:.....	19
The logarithm function .....	8	Recursion [Chapter5].....	20
The linear function .....	8	5.1 Illustrative examples.....	20
The N-Log-N function.....	8	The factorial function .....	20
The quadratic function.....	8	Fractal ruler.....	21
Polynomials and Summations.....	8	Binary search .....	22
The exponential function amd geometric summations.....	9	File systems.....	24
Comparing the functions .....	9	5.2 Analyzing Recursive Algorithms.....	26
4.3 Asymptotic Analysis .....	9	Computing Factorials big-Oh time .....	26
Big-Omega.....	10	Computing Fractal (like some tree branches) big-Oh time .....	26
Big-Theta .....	10	Computing Binary Search Time .....	27
Comparissions.....	10	Disk Usage.....	27
Analysis .....	11	5.3 Properties of recursive algorithms .....	28
Constant-Time Operations: fetching an array element.....	11	5.4 Designing recursive algorithm.....	28
Finding the max of an array .....	11	Recursion Limitations .....	28
Updating the max in a random array with unique values .....	11	Infinite recursion .....	29
Composing strings.....	12	Eliminating Tail recursion .....	29
Three-Way Set Disjointness .....	12	Recursion exercises .....	30
Element uniqueness.....	13	Space complexity [Video] .....	34
Using Sorting as a Problem-Solving Tool..	13	Exercise .....	39
Prefix Averages .....	13	Week 2. Arrays, stacks and queues .....	40
Prefix average: quadatric-time algorithm	14	Arrays. Insertion and deletion .....	40
Prefix average: linear-time algorithm .....	14	Linked Lists.....	45
Complexity analysis: proof methods [Section 4.4] .....	15	Singly Linked lists .....	45
		Circulary Linked Lists.....	48
		Doubly linked lists.....	50

Arrays and lists [Chapter 3 extras].....	53	Chapter 7. List Abstractions.....	71
java.util Methods for Arrays .....	53	Dynamic Array .....	76
java.util Methods for Random .....	53	StringBuilder vs concatenation.....	77
Cryptography .....	54	Week 3. Position-based lists, iterators, trees, and priority queues .....	78
Cryptogrphay Implementation CaesarCipher available in IntelliJ workspace.....	54	Positional List and iterators .....	78
Two-Dimensional Arrays.....	54	Trees .....	80
TicTacToe Implementation available in IntelliJ workspace .....	55	Priority queues.....	90
Singly Linked List Implementation available in IntelliJ workspace.....	55	Binary heaps .....	93
Circulary Linked Lists.....	55	Heap representations .....	97
Round-Rogin scheduling .....	55	Heap construction .....	101
Circulary Linked List Implementation available in IntelliJ workspace.....	55	7.3 Position-based lists .....	107
Doubly Linked Lists.....	55	Doubly Linked List Implementation .....	108
Sentinels.....	55	DoublyLinkedList //Implementation code available in IntelliJ workspace .....	108
Equivalence testing .....	56	Array Implementation .....	109
Cloning .....	56	7.4 Iterator.....	109
Stacks .....	57	8.1-8.4 Trees .....	111
Queues .....	61	Computing Depth .....	113
Dequees .....	64	Computing Height.....	113
The Stack and (De)Queues [Chapter 6 Extras] .	65	Operations for Updating a Linked Binary Tree.....	114
Array based Stack.....	65	9.1 Priority queues.....	115
Singly Link based Stack.....	66	9.2 List-based priority queues .....	115
Adapter Patern.....	66	9.3 Tree-based priority queues: heaps.....	117
Matching Parentheses .....	67	Up-heap Bubbling After an Insertion (at the bottom).....	118
Matching HTML tags .....	68	Down-heap bubbling after a removal....	119
Queues .....	68	Array-Based Representation of a Complete Binary Tree.....	120
Implementing a Queue with a Singly Linked List.....	69	Bottom-up construction .....	121
A Circular Queue .....	69	9.5 Adaptable priority queues .....	122
Double-Ended (Deck) Queues.....	69	Week 4.1 Sort .....	123
Tail recursion example [ $O(n)$ time, $O(1)$ memory vs $O(2^n)$ time and $O(n)$ memory] ...	70	Insertion vs selection sorts both $O(n^2)$ ...	123

Heap sort $O(n \log n)$ – array $\rightarrow$ heapify $\rightarrow$ popback.....	128	Hash functions and has codes .....	157
Merge sort( $n \log n$ ) .....	129	Compression functions .....	158
Array-based implementation of Merge-Sort .....	134	Collision-Handling Schemes.....	159
Quick sort .....	136	Time complexity.....	160
Week 6. lower bound & key-based sorting, selection .....	142	Sorted Maps.....	161
Lower bound .....	142	Sorted Search Tables .....	161
Bucket sort .....	144	Sets, Multisets, and Multimaps .....	162
Radix sorts (LSD, MSD) .....	147	Multiset.....	164
Randomized (in-place) quick select .....	151	Multimap .....	164
Maps .....	155	Week 7. Search trees .....	165
Hashing.....	155	Binary Search Trees .....	165
Application: Counting Word Frequencies..	156	Balanced Search Trees.....	168
Hash Tables .....	156	(2,4) Trees.....	173
		Red-Black Trees .....	180
		Course Guideline .....	1

## Week 1. Introduction to Algorithms and Data Structures

- **data structure:** is a systematic way of organizing and accessing data.
- **algorithm:** step by step procedure for performing some task in a finite amount of time.

To classify data structures as good or bad we must have precise ways of analyzing them:

- **Running time:** the faster the computer time to complete the task the better
- **Space usage:** Another measure for data/algorithm performance, the less space the better.
- Relationship between the running time of an algorithm and the size of its input (**running time as a function of input**)  $t(x)$

## Complexity analysis and big-Oh notation [Sections 4.1, 4.2, 4.3]

### 4.1 Experimental Studies

#### Running time analysis:

A simple mechanism for collecting such running times in Java is based on use of the `currentTimeMillis` method of the `System` class. It will return the offset time from the “epoch” time (**January 1, 1970 UTC**) so we can compare the difference of an offset at the start and at the end, such difference being the **elapsed** time of an algorithm’s execution.

```
long startTime = System.currentTimeMillis(); // record the starting time
/*code*/
long endTime = System.currentTimeMillis(); // record the ending time
long elapsed = endTime - startTime; // compute the elapsed time
```

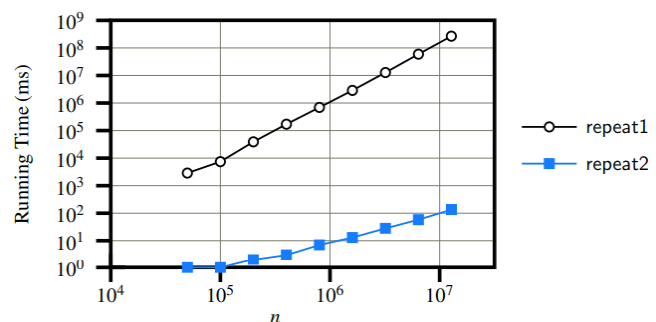
`System.nanoTime()` can be used to measure time in nanoseconds.

#### Relation between input size and running time:

Collect two values,  $x =$  input size ( $n$ ) and  $y =$  running time ( $t$ ). Have a large enough sample to allow a statistical analysis to fit the the best function  $t(n)$  that matches the data.

**Disclaimer:** Times between machines will be different, also within the same machine because the CPU is shared by many processes. But as long as 2 algorithms are compared under similar circumstances, it should be fine.

$n$	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



Running times for string `+=` and `StringBuilder.append()`. Concatenation is not only slower but it gets even slower over time.

**Experiment limitations:**

- **(Environment replication)** Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments
- **(Input bias)** Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important)
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

**Developing the ideal analytical tool:**

- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
- Is performed by studying a high-level description of the algorithm without need for implementation.
- Takes into account all possible inputs

**Counting primitive Operations:**

Instead of using “dirty” elapsing times on inconsistent machines, we “platonify it” and create a universal mathematical model that assigns certain “efficiency penalties” to specific event’s that happen within an algorithm, such “penalty” being the number of times a primitive operation is used:

- Assigning a value to a variable
- Following an object reference
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of an array by index
- Calling a method
- Returning from a method

**primitive operation:** corresponds to a low-level instruction with an execution time that is constant

The sum of the number of times that all primitives are used consitute “ $t$ ”, the platonified running time of the algorithm. **Assumption of the model:** the running time of different primitive operations is the same.

**Measuring Operations as a Function of Input Size:**

A function  $f(n)$  that characterizes the number of primitive operations that are performed as a function of the input size  $n$

**Focusing on the Worst Case Input:**

Certain algorithms respond with different times to inputs of the same size. Ideally the **average case** times would be considered as the reference point, but it is hard to know the average as it requires sophisticated statistical anlysis. Instead the **worst case** time is used as a function of the input size  $n$ . To find the worst case is easy and it amkes the standard of success for an algorithm to perform well even in the worst case.

## 4.2 The seven functions

### The constant function

$$f(n) = c$$

It does not matter what the value of  $n$  is;  $f(n)$  will always be equal to the constant value  $c$

It characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to a variable, or comparing two numbers

### The logarithm function

$$f(n) = \log_b n \text{ and } b > 1$$

The most common base for the logarithm function in computer science is 2 as computers store integers in binary. So  $\log = \log_2$  in this course unless otherwise specified.

- **ceiling**: of a real number,  $x$ , is the smallest integer greater than or equal to  $x$ , denoted with  $\lceil x \rceil$ . aka **round to the nearest integer** (and if it is already an integer, then it remains the same).
- **floor  $\lfloor x \rfloor$ : Round to the lowest integer** (and if it is already an integer, then it remains the same).
- $\lceil \log_b n \rceil$  is an easy to find approximation of  $\log_b n$ . Just divide  $n$  for  $b$  times until the first outcome smaller or equal to 1. the number of divisions is equal to  $\lceil \log_b n \rceil$

### The linear function

$$f(n) = n$$

This function arises in algorithm analysis any time we have to do a single basic operation for each of  $n$  elements.

The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of  $n$  objects that are not already in the computer's memory, because reading in the  $n$  objects already requires  $n$  operations.

### The N-Log-N function

$$f(n) = n \log n$$

It's run time is between the linear function and the quadratic function

### The quadratic function

$$f(n) = n^2$$

There are many algorithms that have **nested loops**, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs  $n \cdot n = n^2$  operations

### Polynomials and Summations

Since the constant, linear, and quadratic functions are very important these have not been lumped into the polynomial category although they actually are one. A polynomial can be summarised to summation

$$f(n) = a^d = f(n) = \sum_{i=0}^d a_i n^i$$



The exponential function and geometric summations

$$f(n) = b^n$$

b is the base and n is the exponent. The most common base in computerscience is 2.

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \dots + a^n$$

and  $a > 0, a \neq 1$

$$= \frac{a^{n+1} - 1}{a - 1}$$

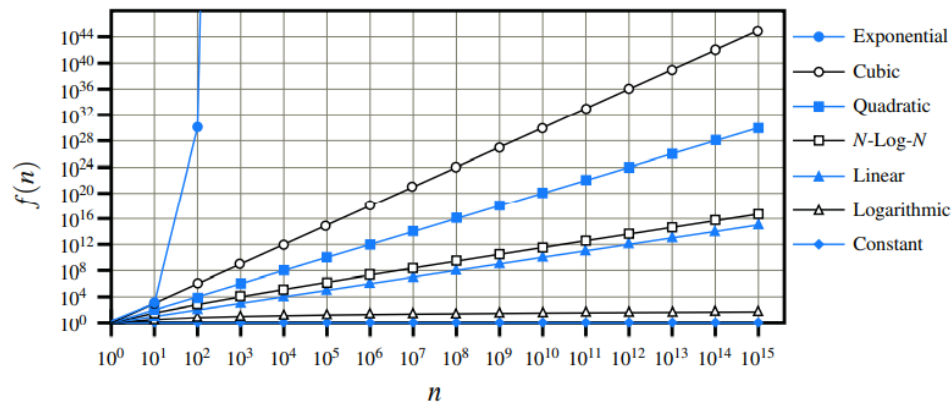
Called geometric summations, because each term is geometrically larger than the previous one if  $a > 1$

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1$$

$$0b1111111 = 0b10000000 - 0b1$$

Comparing the functions

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time.

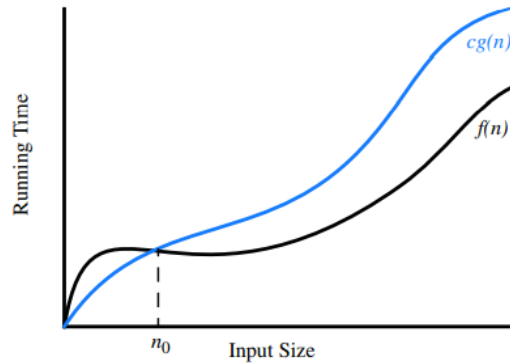


### 4.3 Asymptotic Analysis

Since we focus on the growth rate of running time over input size that is a “big picture” approach, where just giving the proportional grow of t to n is a sufficient enough metric.

#### Big-Oh notation:

The big-Oh notation allows us to say that a function  $f(n)$  is “less than or equal to” another function  $g(n)$  up to a constant factor “c” from  $n_0$  to infinity



$f(n)$  "is" (approximates)  $O(g(n))$  since  $f(n) \leq c \cdot g(n)$  when  $n \geq n_0$

The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

We can use limit properties of a polynomial so that from  $f(n) = a_0 + a_1n + \dots + a_dn^d$  and  $a_d > 0$  we have that  $f(n)$  is  $O(n^d)$ . The highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial and we should use such degree (the closest) to characterize a big-Oh notation function (even if a larger degree technically holds for becoming a big-Oh), in addition lower-order terms should be omitted so we can provide the big-Oh on its simplest terms.

So, for example, we would say that an algorithm that runs in worst-case time  $4n^2 + n \log n$  is a quadratic-time algorithm, since it runs in  $O(n^2)$  time.

### Big-Omega

If big-oh = a function is less than or equal to another function, then big-Omega = a function is greater than or equal to that of another.

We say that  $f(n)$  is  $\Omega(g(n))$ , pronounced "  $f(n)$  is big-Omega of  $g(n)$ ," if  $g(n)$  is  $O(f(n))$ .

$$f(n) \geq cg(n), \text{ for } n \geq n_0$$

### Big-Theta

Two functions grow at the same rate,  $f(n)$  is  $\Theta(g(n))$  "  $f(n)$  is big-Theta of  $g(n)$ ," if  $f(n)$  is  $O(g(n))$  and  $f(n)$  is  $\Omega(g(n))$ . It is expressed as  $c'g(n) \leq f(n) \leq c''g(n)$ , for  $n \geq n_0$  where  $c'$  and  $c''$  are real constants and  $n_0 \geq 1$ .

### Comparissions

We can use the big-Oh notation to order classes of functions by asymptotic growth rate.

$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	$1.84 \times 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 \times 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 \times 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 \times 10^{154}$

Inversely, instead of looking at the running time growth, we can also look at the input capacity, that is capability of solving an amount of input  $n$ , over time (where here bigger is better).

Running Time ( $\mu$ s)	Maximum Problem Size ( $n$ )		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
$2^n$	19	25	31

### Asymptotic limitations:

Although  $10^{100}n$  is  $O(n)$  we would prefer an algorithm that is  $10n \cdot \log n$  that is  $O(n \cdot \log n)$  because the constant factor of one googol is so large that an input will rarely be larger than that where  $10^{100}n$  would outperform  $10n \log n$ . Besides,  $O(n \cdot \log n)$  is regarded as an efficient algorithm.

### Analysis

Constant-Time Operations: fetching an array element.

Operations that run in constant-time are expressed as  $O(1)$ . Assume variable  $A$  is an array of  $n$  elements.  $A.length$  is evaluated in constant time as an explicit variable is stored that records the length of the array (so we don't have to compute it but to fetch it). Similarly,  $A[j]$  can be accessed in constant time as an array uses a consecutive block of memory. The  $j^{\text{th}}$  element is found by validating the index with the  $.length$  variable and fetching its memory address. Therefore  $A[j]$  is  $O(1)$ .

### Finding the max of an array

Is an algorithm that grows proportional to  $n$  as all entries of the array will need to be evaluated against the current max value found. That is, such algorithm runs in  $O(n)$  time. The justification is provided:

```
//Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0]; // assume first entry is biggest (for now)
    for (int j = 1; j < n; j++) // consider all other entries
        if (data[j] > currentMax) // if data[j] is biggest thus far...
            currentMax = data[j]; // record it as the current max
    return currentMax;
}
```

1. The initialization at lines 3 and 4 and the return statement at line 8 require only a constant number of primitive operations.
2. Each iteration of the loop also requires only a constant number of primitive operations, and the loop executes  $n - 1$  times

From 1 and 2 we have that the time is  $t(n) = c'' + c'(n-1)$ , which approximates to  $t(n) = c'n$  in the long term, which is  $O(n)$ .

### Updating the max in a random array with unique values

the expected number of times we update the biggest (including initialization) is  $n^{\text{th}}$  Harmonic number. It can be shown that  $H_n$  is  $O(\log n)$ . The Harmonic number is originally expressed as a summation:

$$H_n = \sum_{i=1}^n 1/i$$

Composing strings

**Using concatenation:**

```
//Uses repeated concatenation to compose a String with n copies of char c. */
public static String concatenate(char c, int n) {
    String answer = "";
    for (int j = 0; j < n; j++)
        answer += c;
    return answer;
}
```

Strings in Java are immutable objects. Once created, an instance cannot be modified (So they are essentially an especial reference type). The concatenation operator += does not append a character to the existing String, instead it produces a new String with the desired sequence of characters. Then it reassigns the variable to point to the new memory location with the new String.

The creation of a new string as a result of a concatenation, requires time that is proportional to the length of the resulting string.

Therefore, the overall time taken by this algorithm is proportional to  $1 + 2 + \dots + n$ . Which is  $O(n^2)$ . It resembles a loop inside a loop.

Three-Way Set Disjointness

The worst case scenario when checking the disjointness of 3 sets is that eventually it is disjoint. None of them share a common value present at all of them. An algorithm that evaluates for each element in setA whether it is in setB and setC may require 3 loops, which are nested.

```
//Returns true if there is no element common to all three arrays.
public static boolean disjoint1(int[] groupA, int[] groupB, int[] groupC) {
    for (int a : groupA)
        for (int b : groupB)
            for (int c : groupC)
                if ((a == b) && (b == c))
                    return false; // we found a common value
    return true; // if we reach this, sets are disjoint
}
```

This is  $O(n^3)$  as in the worst case scenario each of the sets have size n. However, if an element a does not match with any element in B it is a waste of time to check if there's a match in C.

```
//Returns true if there is no element common to all three arrays.
public static boolean disjoint2(int[] groupA, int[] groupB, int[] groupC) {
    for (int a : groupA)
        for (int b : groupB)
            if (a == b) // only check C when we find match from A and B
                for (int c : groupC)
                    if (a == c) // and thus b == c as well
                        return false; // we found a common value
    return true; // if we reach this, sets are disjoint
}
```

In the second algorithm:

1. The management of the for loop over A requires  $O(n)$  time.
2. The time of the for loop over B accounts for a total of  $O(n^2)$  (loop inside a loop)
3. There are at most  $n$  such pairs  $(a,b)$  where  $a \neq b$
4. Therefore, the management of the loop over C and the commands within the body of that loop use at most  $O(n^2)$ . This is because the commands within the body of that loop are a constant, and big-Oh ignores constants if the degree of the polynomial is bigger than 1. And for each element in C (that is, for  $n$  times) we check only against 1 remaining set, that is all the  $a$ 's that don't exist in B. Comparing 2 sets is  $O(n^2)$ . Other way of seeing it would be that after  $O(n^2)$  after  $A*B$ , there's a constant body check in loop B equal to  $n$ , so  $O(n^2+n) = O(n^2)$

Element uniqueness

```
//Returns true if there are no duplicate elements in the array.
public static boolean unique1(int[] data) {
    int n = data.length;
    for (int j = 0; j < n - 1; j++)
        for (int k = j + 1; k < n; k++)
            if (data[j] == data[k])
                return false; // found duplicate pair
    return true; // if we reach this, elements are unique
}
```

The worst-case running time of this method is proportional to  $(n-1) + (n-2) + \dots + 2 + 1$ , Observe that there are 2 nested loops. This is  $O(n^2)$ .

Using Sorting as a Problem-Solving Tool

By sorting the array of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, to determine if there are any duplicates, all we need to do is perform a single pass over the sorted array, looking for consecutive duplicates.

```
//Returns true if there are no duplicate elements in the array.
public static boolean unique2(int[] data) {
    int n = data.length;
    int[] temp = Arrays.copyOf(data, n); // make copy of data
    Arrays.sort(temp); // and sort the copy
    for (int j = 0; j < n - 1; j++)
        if (temp[j] == temp[j + 1]) // check neighboring entries
            return false; // found duplicate pair
    return true; // if we reach this, elements are unique
}
```

The best sorting algorithms (including those used by `Array.sort` in Java) guarantee a worst-case running time of  $O(n \log n)$ . Once the data is sorted, the subsequent loop runs in  $O(n)$  time.

The entire algorithm runs in  $O(n \log n + n) = O(n \log n)$  time.

Prefix Averages

Given a sequence  $x$  consisting of  $n$  numbers, we want to compute a sequence  $a$  such that  $a_j$  is the average of elements  $x_0, \dots, x_j$ , for  $j = 0, \dots, n-1$ , that is:

$$a_j = \frac{\sum_{i=0}^j x_i}{j+1}$$

Prefix average: quadratic-time algorithm

It computes each element  $a_j$  independently, using an inner loop to compute that partial sum.

```
//Returns an array a such that, for all j, a[j] equals the average of x[0],
..., x[j].
public static double[] prefixAverage1(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    for (int j = 0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i = 0; i <= j; i++)
            total += x[i];
        a[j] = total / (j + 1); // record the average
    }
    return a;
}
```

Calling this method with a sample array  $\{7.0, 5.0, 7.0, 5.0\}$  returns:

$$7.0 = 7/1$$

$$6.0 = (7+5)/2$$

$$6.333333333333333 = (7+5+7)/3$$

$$6.0 = (7+5+7+5)/4$$

1. The initialization of  $n = x.length$  at line 3 and the eventual return of a reference to array  $a$  at line 11 both execute in  $O(1)$  time.
2. Creating and initializing the new array,  $a$ , at line 4 can be done with in  $O(n)$  time, using a constant number of primitive operations per element.
3. The body of the outer loop, controlled by counter  $j$ , is executed  $n$  times. So we have  $O(n)$  time.
4. The body of the inner loop, which is controlled by counter  $i$ , is executed  $j+1$  times. The inner loop, is executed  $1 + 2 + 3 + \dots + n$  times, which is  $n(n+1)/2$  in other words,  $O(n^2)$  time.

The running time of implementation `prefixAverage1` is given by the sum of these terms. The first term is  $O(1)$ , the second and third terms are  $O(n)$ , and the fourth term is  $O(n^2)$  which is  $O(n^2)$

Prefix average: linear-time algorithm

You can achieve the same result without doing two nested loops. Which is not only more readable but also is  $O(n)$

```
public static double[] prefixAverage2(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    double total = 0; // compute prefix sum as x[0] + x[1] + ...
    for (int j = 0; j < n; j++) {
        total += x[j]; // update prefix sum to include x[j]
        a[j] = total / (j + 1); // compute average based on current sum
    }
    return a;
}
```

## Complexity analysis: proof methods [Section 4.4]

### 4.4.1 By Example

To justify claims of the generic form “There is an element  $x$  in a set  $S$  that has property  $P$ ” we only need to produce a particular  $x$  in  $S$  with such property  $P$ .

Similarly, to justify that a claim is false of the generic form “For all...” we just need to provide a particular  $x$  that does not contain such property. That is a counter example.

### 4.4.2 The Contra Attack

Proof by contrapositive. Sometimes the contrapositive, which is equivalent to the initial statement, might be easier to prove. See CSE1300 Reasoning and Logic.

Proof by contradiction. We reach a contradiction in the statement that is the exact opposite of the original statement. A contradiction in the opposite statement makes the original statement true.

### 4.4.3 Induction and Loop Invariantes

Statements that use “for all  $n \geq 1$ ...” it might be possible to use induction as a proof method. It shows that there is a sequence of implications that starts with something known to be true (base case) and leads to showing that  $k+1$  for all  $k$  is true, so continuity proof. Proving the induction step and using the property of Natural number continuity (so, by the principle of induction), the initial argument can be proven. The inductive argument is a template for building a sequence of direct justifications.

**Justification:** We will justify this equality by induction.

**Base case:**  $n = 1$ . Trivial, for  $1 = n(n+1)/2$ , if  $n = 1$ .

**Induction step:**  $n \geq 2$ . Assume the inductive hypothesis is true for any  $j < n$ .

Therefore, for  $j = n - 1$ , we have

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}.$$

Hence, we obtain

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{2n+n^2-n}{2} = \frac{n^2+n}{2} = \frac{n(n+1)}{2},$$

thereby proving the inductive hypothesis for  $n$ . ■

Loop invariant:

To prove that some statement  $L$  about a Loop is correct, define  $L$  in terms of a series of smaller statements, where:

1. The initial claim  $L_0$  is true before the loop begins.
2. If  $L_{j-1}$  is true before the iteration  $j$ , then  $L_j$  will be true after iteration  $j$
3. The final statement,  $L_k$ , implies the desired statement to be true.

[https://www.youtube.com/watch?v=3YP6NP1\\_tFO](https://www.youtube.com/watch?v=3YP6NP1_tFO)

## Lecture 1 &amp; 2 &amp; Exercises

A function  $f(n)$  is  $O(n)$  if and only if we can put a factor before  $n$  such that the result will always be greater than  $f(n)$  for values of  $n$  above some value.

Indeed for  $f(n)$  is  $O(n)$ :

$$f(n) \text{ is } O(n) \leftrightarrow \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot n)))$$

for  $f(n)$  is  $O(1)$ :

$$f(n) \text{ is } O(1) \leftrightarrow \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot 1)))$$

for  $f(n)$  is  $O(x)$ :

$$f(n) \text{ is } O(x) \leftrightarrow \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot x)))$$

change  $x$  for whichever  $O(\text{type})$  you want.

for  $f(n)$  is  $O(g(n))$ :

$$f(n) \text{ is } O(g(n)) \leftrightarrow \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))))$$

This is the one used in formal proofs of functions.

$$\$f(n) \text{ is } O(g(n)) \iff \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)))\$$$

How to prove such functions?

Choose a  $c > 0$  and an  $n_0 > 0$ , take an arbitrary  $n \geq n_0$ , and show that  $f(n) \leq c \cdot g(n)$

What  $c$  and  $n_0$  can we choose to prove that  $2n + 10$  is  $O(n)$ ?

- (a)  $c = 0, n_0 = 100$
- (b)  $c = 1, n_0 = 35$
- (c)  $c = 2, n_0 = 15$
- (d)  $c = 3, n_0 = 11$

Because:

$$f(11) \leq 3 \cdot O(11)$$

$$2 \cdot 11 + 10 \leq 3 \cdot 11$$

$$32 \leq 33$$



**Theorem:  $2n^2 + 5n + 49$  is  $O(n^2)$** 

Proof.

From the definition of  $f(n)$  is  $O(g(n))$  we have that

$$f(n) \text{ is } O(g(n)) \leftrightarrow \exists c, n_0 \left( 0 < c \wedge 0 < n_0 \wedge \left( \forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)) \right) \right)$$

" $f(n)$  is  $O(g(n))$  iff there exists a  $c$  and  $n_0$  that are bigger than 0 and that have the property that for all  $n$ , iff  $n$  is greater or equal to  $n_0$  then  $f(n)$  is less or equal to  $c \cdot g(n)$ ".

Since we part from taking the definition  $f(n)$  is  $O(g(n))$  as true, we just need to find a  $c > 0$  and an  $n_0 > 0$ , whereby if we have an arbitrary  $n \geq n_0$ , we can show that  $f(n) \leq c \cdot g(n)$ .

Take  $c = 3$ ,  $n_0 = 10$ , and an arbitrary  $n \geq n_0$ . This means that for all  $n \geq 10$ ,  $f(n) \leq 3 \cdot n^2$ .

Therefore, we have that for all all  $n \geq 10$ :

$$f(n)/n^2 \leq 3$$

$$(2n^2 + 5n + 49)/n^2 \leq 3$$

$$2 + 5/n + 49/n^2 \leq 3$$

$$5/n + 49/n^2 \leq 1, \text{ which is true for } n = 10 \text{ since } 5/10 + 49/100 \leq 1$$

and it is also true for any value of  $n > 10$  as the left side of the equation will only get smaller.

Therefore, for all  $n \geq 10$ ,  $2n^2 + 5n + 49$  is  $O(n^2)$

Q.E.D.

$O(g(n)) = g(n)$  OR FASTER

$\Omega(g(n)) = g(n)$  OR SLOWER

$\Theta(g(n)) = O(g(n))$  and  $\Omega(g(n))$ , which means tightes bound of  $O(g(n))$  which means precisely  $g(n)$

Which of these functions are  $O(n^2)$ ?

- (a)  $50n^2 + 100$
- (b)  $3n^2 + 5n^3 + 4$
- (c)  $n^4$
- (d)  $80n + 5$
- (e)  $2^n$
- (f)  $80$
- (g)  $\log_2(n)$

What is the tightest bound on  $f(n) = 80$ ?

Answer:  $O(1)$

Is  $f(n)$  is  $O(n) \Rightarrow f(n)$  is  $O(n^2)$  true for all  $f(n)$ ?

Answer: Yes!

Let  $R$  be a relation such that  $(f, g) \in R$  iff  $f(n)$  is  $O(g(n))$ . Is  $R$  an equivalence relation?

- (a) Yes
- (b) No,  $R$  is not reflexive
- (c) No,  $R$  is not symmetric
- (d) No,  $R$  is not transitive
- (e) No,  $R$  is not reflexive and not symmetric
- (f) No,  $R$  is not reflexive and not transitive
- (g) No,  $R$  is not symmetric and not transitive
- (h) No,  $R$  is not reflexive, not symmetric and not transitive

This means that  $f(n)$  is  $O(g(n))$  but  $g(n)$  is not necessarily  $O(f(n))$

A function  $f(n)$  is  $\Omega(g(n))$  if and only if we can put a factor before  $n$  such that the result will always be smaller than  $f(n)$  for values of  $n$  above some value. How can this be written in predicate logic?

$$f(n) \text{ is } O(g(n)) \leftrightarrow \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \geq c \cdot g(n))))$$

$$f(n) \text{ is } O(g(n)) \iff \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))))$$

Which of these functions are  $\Omega(n^2)$ ?

- (a)  $50n^2 + 100$
- (b)  $3n^2 + 5n^3 + 4$
- (c)  $n^4$
- (d)  $80n + 5$
- (e)  $2^n$
- (f) 80
- (g)  $\log_2(n)$

Given that a function  $f(n)$  is  $\Theta(g(n))$  if  $f(n)$  is both  $O(g(n))$  and  $\Omega(g(n))$ .

Which of these functions are  $\Theta(n^2)$ ?

- (a)  $50n^2 + 100$
- (b)  $3n^2 + 5n^3 + 4$
- (c)  $n^4$
- (d)  $80n + 5$
- (e)  $2^n$
- (f) 80
- (g)  $\log_2(n)$

For theta:

$$f(n) \text{ is } O(g(n)) \iff \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))) \wedge (n \geq n_0 \Rightarrow f(n) \geq c \cdot g(n)))$$

$$f(n) \text{ is } \Theta(g(n)) \iff \exists c, n_0 (0 < c \wedge 0 < n_0 \wedge (\forall n (n \geq n_0 \Rightarrow f(n) = c \cdot g(n))))$$

Let  $R$  be a relation such that  $(f, g) \in R$  iff  $f(n)$  is  $\Theta(g(n))$ . Is  $R$  an equivalence relation?

(a) Yes

Counting primitives (raw vs constants) operations:

```

public static int MaxSRDifference1(List<Integer> SR) {
    int currentMax = 0; // 1
    for(int i = 0; i < SR.size(); i++) // 1 + (n + 1) + 2n
        for(int j = 0; j < SR.size(); j++) // (1 + (n + 1) + 2n)n
            int SRDifference = Math.abs(SR.get(i) - SR.get(j)); // 3n2
            if(SRDifference > currentMax) // n2
                currentMax = SRDifference; // n2
    return currentMax; // 1
} // Total : 7n2 + 5n + 3

public static int MaxSRDifference2(List<Integer> SR) {
    int currentMax = 0; // 1
    for(int i = 0; i < SR.size(); i++) // 1 + (n + 1) + 2n
        for(int j = i+1; j < SR.size(); j++) // 2n + (n(n + 1)/2) + 2(n(n - 1)/2)
            int SRDifference = Math.abs(SR.get(i) - SR.get(j)); // 3(n(n - 1)/2)
            if(SRDifference > currentMax) // 1(n(n - 1)/2)
                currentMax = SRDifference; // 1(n(n - 1)/2)
    return currentMax; // 1
} // Total : 4n2 + 2n + 3

public static int MaxSRDifference3(List<Integer> SR) {
    int currentMax = 0; // 1
    for(int i = 0; i < SR.size(); i++) // 1 + (n + 1) + 2n
        int thisSR = SR.get(i); // n
        if(thisSR > currentMax) // n
            currentMax = thisSR; // n
    int currentMin = Integer.MAX_VALUE; // 1
    for(int i = 0; i < SR.size(); i++) // 1 + (n + 1) + 2n
        int thisSR = SR.get(i); // n
        if(thisSR < currentMin) // n
            currentMin = thisSR; // n
    return currentMax - currentMin; // 2
} // Total : 12n + 8

```

We can observe from here that there is not a consensus where it comes to identifying which operations count as a primitive operation. Furthermore, we already know that these might also have different real times.

In the next exercise we can see that it does not matter what constitutes as a primitive, the number of primitives per line can be reduced to just a constant. What we will see that since we use big-Oh notation, the focus is on observing the relation with  $n$ . Do we see a constant time? a linear ( $n$ ) time? quadratic ( $n^2$ )? logarithmic? etc.

## Big-Oh in practice

- What is the tightest bound on the runtime of method  $f(\text{int}[] x)$ ?

```

1  public static int f(int[] x) {
2      int sum; // c0
3      switch(x.length) { // c1
4          case 1: // c2
5              sum = 2; // c3
6              break; // c4
7          default: // c5
8              sum = 0; // c6
9      }
10     for(int i : x) { // c7 · n + c8
11         sum += Math.sqrt(i < 10 ? i : i >>> 1); // c9 · n
12     }
13     return sum; // c10
14 } // Total: (c7 + c9) · n + c0 + c1 + c2 + c3 + c4 + c5 + c6 + c8 + c10

// Total: c11 · n + c12

```

Which is  $O(n)$

## Recursion [Chapter5]

### 5.1 Illustrative examples

Recursion is a technique by which a method makes one or more calls to itself. When one invocation of the method makes a recursive call, that invocation is **interrupted** (the return address is saved and the new instruction is executed) until the recursive call completes.

The factorial function

The factorial of a positive integer  $n$ , denoted  $n!$ , is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \dots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

Application of the factorial: It is used to find the number of ways in which  $n$  distinct items can be arranged into a sequence, that is the number of permutations of  $n$  items

permutation: each of several possible ways in which a set or number of things can be ordered or arranged.

For example 3 characters a,b,c can be arranged in  $3! = 3 \cdot 2 \cdot 1 = 6$  ways: abc, acb, bac, bca, cab, and cba.

The piecewise function above can be simplified into a **recursive definition**:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

We have (one or more) **base case** at  $n! = 1$  for  $n = 0$ . We have one or more **recursive cases**, which define the function in terms of itself.

```

public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0)
        throw new IllegalArgumentException(); // argument must be nonnegative
    else if (n == 0)
        return 1; // base case
    else
        return n * factorial(n - 1); // recursive case
}

```

This method does not use explicit loops. Repetition is achieved through repeated recursive invocations of the method.

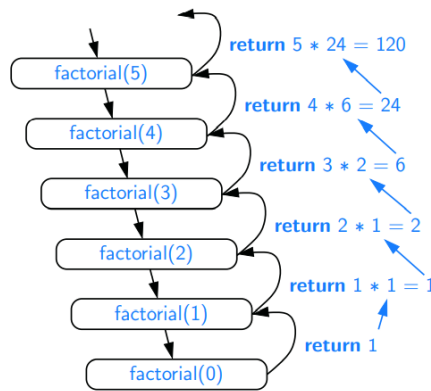
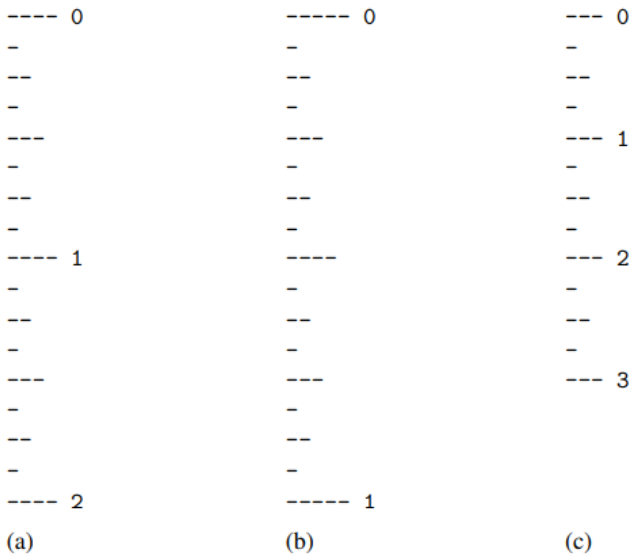


Figure 5.1: A recursion trace for the call factorial(5).

Fractal ruler



If you zoom in in a big ruler from meters to mm, going through all the in between measures, you see that a meter can be divided by 2, half a meter is 5dm, if you zoom to 1dm a dm can be split in 2 and you get 5cm, if you zoom in you get 1cm, etc. and all these zooms have the same image pattern.

## Binary search

Binary search is a recursive algorithm that efficiently locates a target value **within a sorted sequence** of  $n$  elements stored in an array. We will see that it will help us reduce the  $O$  time from  $O(n)$  to **\*\*\*instert\*\*\*** making it one of the most important computer algorithms and the reasons why we often store data in sorted order.

Take the following array as an example:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

**Figure 5.4:** Values stored in sorted order within an array. The numbers at top are the indices.

**Linear search  $O(n)$ :** When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set

**sorted and indexable** sequence: The bible has 1200 pages and you have to try to guess which page I have randomly chosen. You could progressively name a one page after the other, but that could take you 1200 tries in the worst case and on average 600 tries. Or we could play “higher” and “lower” and split the options in half. Let’s say I’ve chosen page 610.

You:  $1200/2 = 600$

Me: higher!

You:  $(1200+600)/2 = 900$

Me: lower!

You:  $(900+600)/2 = 750$

Me: lower!

You:  $(750+600)/2 = 675$

Me: lower!

You:  $(675+600)/2 = 638$  (rounded)

Me: lower!

You:  $(638+600)/2 = 619$

Me: lower!

You:  $(638+600)/2 = 610$  (rounded)

Me: Yes!

It took us just 7 tries! In the longterm, this search algorithm will work best on average. The algorithm in the next pages has a slightly different base case logic but follows the same “halving” approach.

We call an element of the sequence a **candidate** if, at the current stage of the search, we cannot rule out that this item matches the target.

The algorithm maintains two parameters, low and high, such that all the candidate elements have index at least low and at most high with the starting values of low = 0 and high = n-1

**median candidate:**  $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$  (floor)

So, in our example low candidate = all the non-ruled out bible pages. Target = page 610. In the binary search algorithm we consider:

- If the **target** (610) **equals** the **median** (of current highest and lowest possible bible pages) candidate, then we have found the item we are looking for, and the **search terminates**.
- **If the target is less than the median candidate**, then we **recur** on the first half of the sequence, that is, on the interval of indices from **low to median-1**.
- **If the target is greater than the median candidate**, then we **recur** on the second half of the sequence, that is, on the interval of indices from **median+1 to high**

If the element does not exist in the array we get an unsuccessful search where low > high is an empty.

Therefore binary search manages to run in  $O(\log n)$  time. Computer Science logs are base 2. So, an n of 1 billion takes only 30 operations. Example of binary search below:

```
/**
 * Binary Search
 * @param data = ASC SORTED int array
 * @return Returns true if the target value is found
 * @low - smallest possible candidate
 * @high - largest possible candidate
 */
public static boolean binarySearch(int[] data, int target, int low, int high)
{
    if (low > high)
        return false; // interval empty; no match
    else {
        int mid = (low + high) / 2;
        if (target == data[mid])
            return true; // found a match
        else if (target < data[mid])
            return binarySearch(data, target, low, mid - 1); // recur left of
the middle
        else
            return binarySearch(data, target, mid + 1, high); // recur right
of the middle
    }
}

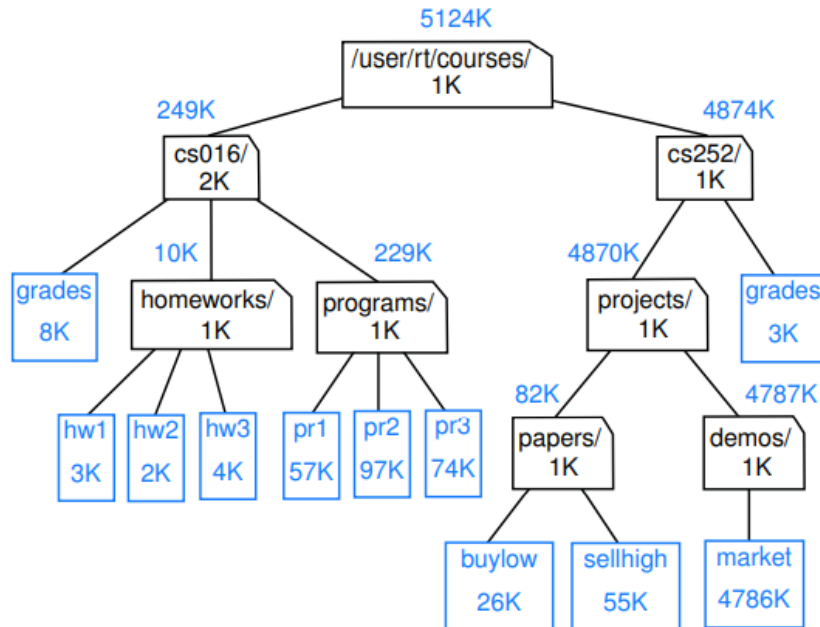
/**
 * Lazy method call that implicitly takes [0,data.length-1] as interval
 */
public static boolean binarySearch(int[] data, int target){
    return binarySearch(data, target, 0, data.length-1);
}
```

## File systems

File-system directories (also called folders) are defined by the OS in a recursive way.

You have a top-level directory, and the contents of this directory consists of files and other directories. The OS allows directories to be nested arbitrarily deep (as long as there is memory available) but eventually there will be a base directory without folders in it.

Managing directories is done with recursive algorithms, such as computing the total disk usage for all files and folders within a particular directory.



**immediate disk space:** disk space used by each entry

**cummulative disk space:** disk space used by that entry and its nested folders.

cs016 has 2k of immediate space but 249 of cummulative space.

The **cumulative disk space** for an entry can be computed with the **immediate disk space used by the entry plus the sum of the cumulative disk space** of its nested folders. In pseudocode:

**Algorithm** DiskUsage(*path*):

**Input:** A string designating a path to a file-system entry

**Output:** The cumulative disk space used by that entry and any nested entries

$total = size(path)$  {immediate disk space used by the entry}

**if** *path* represents a directory **then**

**for** each *child* entry stored within directory *path* **do**

$total = total + DiskUsage(child)$  {recursive call}

**return** *total*

**Code Fragment 5.4:** An algorithm for computing the cumulative disk space usage nested at a file-system entry. We presume that method `size` returns the immediate disk space of an entry.



```

Java implementation:
/**
 * Calculates the total disk usage (in bytes) of the portion of the file
 * system rooted
 * at the given path, while printing a summary akin to the standard 'du' Unix
 * tool.
 */
public static long diskUsage(File root) {
    long total = root.length(); // start with direct disk usage
    if (root.isDirectory()) { // and if this is a directory,
        for (String childName : root.list()) { // then for each child
            File child = new File(root, childName); // compose full path to
            child
            total += diskUsage(child); // add child's usage to total
        }
    }
    System.out.println(total + "\t" + root); // descriptive output
    return total; // return the grand total
}

```

```

8      /user/rt/courses/cs016/grades
3      /user/rt/courses/cs016/homeworks/hw1
2      /user/rt/courses/cs016/homeworks/hw2
4      /user/rt/courses/cs016/homeworks/hw3
10     /user/rt/courses/cs016/homeworks
57     /user/rt/courses/cs016/programs/pr1
97     /user/rt/courses/cs016/programs/pr2
74     /user/rt/courses/cs016/programs/pr3
229   /user/rt/courses/cs016/programs
249   /user/rt/courses/cs016
26     /user/rt/courses/cs252/projects/papers/buylow
55     /user/rt/courses/cs252/projects/papers/sellhigh
82     /user/rt/courses/cs252/projects/papers
4786  /user/rt/courses/cs252/projects/demos/market
4787  /user/rt/courses/cs252/projects/demos
4870  /user/rt/courses/cs252/projects
3      /user/rt/courses/cs252/grades
4874  /user/rt/courses/cs252
5124  /user/rt/courses/

```

**Figure 5.8:** A report of the disk usage for the file system shown in Figure 5.7, as generated by our `diskUsage` method from Code Fragment 5.5, or equivalently by the Unix/Linux command `du` with option `-a` (which lists both directories and files).

It will first go as deep as possible until hitting the directory without folders and then it will display its size, then it will return, repeat the same process and once all directories at a directory have been called, then the cumulative size of such location is displayed, and so on.

## 5.2 Analyzing Recursive Algorithms

With a recursive algorithm, we account for the number of operations within the body of a call. Then we can account for the summation of all calls. Each recursive method has its own dynamics, so they need to be carefully analysed. For each of the previous examples we have the following results.

Computing Factorials big-Oh time

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

```
public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0)
        throw new IllegalArgumentException(); // c0
    else if (n == 0)
        return 1; // c1
    else
        return n * factorial(n - 1); // c2 + c3 * #cumulative f(n-1) calls
}
```

- Let's try to find the number of cumulative calls:  
Since we only accept  $n \geq 0$ . We have that if  $n = 0$ , that's the base case which is 1 call. then we have each of the integers from 1 to  $n$ , that is, a total of  $n$  integers making a call. if we sum  $n$  calls + the base call we get a total of  $n+1$  calls.
- Since  $f(n)$  has  $c_0 + c_1 + c_2 + c_3 * \text{cumulative } f(n-1) \text{ calls}$ , we have  $f(n)$  has  $c_4 + c_3 * n$ . Therefore  $f(n)$  is  $O(n)$ .

Computing Fractal (like some tree branches) big-Oh time

Let's say that the example of the ruler has the property that calling the method once and not at  $c = 0 =$  basecase, will spawn two calls, and then each of those two calls will spawn other two. We get into a tree situation where for  $c \geq 0$ , a call to `drawInterval(c)` results in precisely  $2^c - 1$  lines of output.

Proof.

We can see that `drawInterval(0)` generates no output and therefore  $2^0 - 1 = 0$  is true, which serves as a base case for our claim.

Recursive case:

$$\begin{aligned} \text{drawInterval}(n) &= c_1 + 2 * (\text{drawInterval}(n-1)) \\ &= c_1 + 2 * (c_1 + 2 * (\text{drawInterval}(n-2))) \\ &= c_1 + 2 * (c_1 + 2 * (c_1 + 2 * (\text{drawInterval}(n-3)))) \\ &= c_1 + 2 * c_1 + 4 * (c_1 + 2 * (\text{drawInterval}(n-3))) = 3 * c_1 + 4 * c_1 + 8 * (\text{drawInterval}(n-3)) \\ &= 7 * c_1 + 8 * (\text{drawInterval}(n-3)) \end{aligned}$$

repeat  $n$  times (until reaching base case)

$$(2^n - 1) * c_1 + 2^n * (\text{drawInterval}(n-n)) = (2^n - 1) * c_1 + 2^n * (\text{drawInterval}(0)) = (2^n - 1) * c_1 + 2^n * 0 = (2^n - 1) * c_1$$

Therefore  $f(n)$  is  $O(2^n)$

### Computing Binary Search Time

As in all recursive functions. The running time is proportional to the number of recursive calls performed and such number is multiplied to the “immediate time” of the recursive function body.

In the binary search example, the “immediate time” (that is the folder space without regarding sub folders), was equal to  $O(1)$ . so  $O(1)$  will be multiplied by the number of recursive calls. Which in this case is expected to be  $\log n + 1$  in the worst case scenario.

Proposition: The binary search algorithm runs in  $O(\log n)$  time for a sorted array with  $n$  elements. Where  $n$  is bigger than 0

**Justification:** To prove this claim, a crucial fact is that with each recursive call the number of candidate elements still to be searched is given by the value

$$\text{high} - \text{low} + 1.$$

Moreover, the number of remaining candidates is reduced by at least one-half with each recursive call. Specifically, from the definition of  $\text{mid}$ , the number of remaining candidates is either

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

or

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}.$$

Initially, the number of candidates is  $n$ ; after the first call in a binary search, it is at most  $n/2$ ; after the second call, it is at most  $n/4$ ; and so on. In general, after the  $j^{\text{th}}$  call in a binary search, the number of candidate elements remaining is at most  $n/2^j$ . In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate elements. Hence, the maximum number of recursive calls performed, is the smallest integer  $r$  such that

$$\frac{n}{2^r} < 1.$$

Since we know that  $n > 0$ , we also have that  $r > 0$  so multiplying both sides of the equations with  $2^r$  wont change the sign of the inequality. Then we have that  $n < 2^r$

$$\log(n) < \log(2^r)$$

$$\log(n) < r$$

Being  $r$  the smallest integer such that  $r > \log n$ , we have  $r = \lfloor \log n \rfloor + 1$ . That means that binary search runs in  $O(\log n)$  time.

### Disk Usage

**amortization:** Counting the number of nested loops does not provide the tight upper bound, as it can be sometimes proved that sometimes  $O(n)$  is not achieved at a sepcific level, therefore making the assumption that  $f(n)$  runs at  $O(n^{\text{loops}})$  wrong.

### 5.3 Properties of recursive algorithms

- **linear recursion:** If a recursive call starts at most one other. Factorial is an example, and also binary search (despite the binary prefix) since from within the recursion body you call at most one other recursion. Other examples include: summing array elements, reversing elements of an array, power function (with int powers example)
- **binary recursion:** If a recursive call may start two others. Examples: fractal ruler, summing elements of a sequence (which has  $O(\log n)$  space but  $O(n)$  time.
- **multiple recursion:** If a recursive call may start three or more others, such as diskSpace method. Multiple recursion can be used in algorithms that solve a combinatorial puzzle by enumerating and testing all possible configurations.

### 5.4 Designing recursive algorithms

1. **Test for base cases:** These base cases should be defined so that every possible chain of recursive calls will eventually end at a base case.
2. **Recur:** If not a base case, we perform one or more recursive calls that progress towards a base case.

### Recursion Limitations

#### An Inefficient Recursion for Computing Fibonacci Numbers

In Section 2.2.3, we introduced a process for generating the progression of Fibonacci numbers, which can be defined recursively as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \text{for } n > 1. \end{aligned}$$

Ironically, a recursive implementation based directly on this definition results in the method fibonacciBad shown in Code Fragment 5.13, which computes a Fibonacci number by making two recursive calls in each non-base case.

```

1  /** Returns the nth Fibonacci number (inefficiently). */
2  public static long fibonacciBad(int n) {
3      if (n <= 1)
4          return n;
5      else
6          return fibonacciBad(n-2) + fibonacciBad(n-1);
7  }
```

**Code Fragment 5.13:** Computing the  $n^{\text{th}}$  Fibonacci number using binary recursion.

Unfortunately, such a direct implementation of the Fibonacci formula results in a terribly inefficient method. Computing the  $n^{\text{th}}$  Fibonacci number in this way requires an exponential number of calls to the method.

We can compute  $F_n$  much more efficiently using a recursion in which each invocation makes only one recursive call. Rather than having the method return a single value, which is the  $n^{\text{th}}$  Fibonacci number, we define a recursive method that returns an array with two consecutive Fibonacci numbers  $F_n, F_{n-1}$  using the convention  $F_{-1}=0$ .

Now we return twice the data but execute half the recursion calls. Since there is only one call per body and the operations within the body are constant the algorithm below runs at  $O(n)$  time.

```
//Returns array containing the pair of Fibonacci numbers, F(n) and F(n-1). */
public static long[] fibonacciGood(int n) {
    if (n <= 1) {
        long[] answer = {n, 0};
        return answer;
    } else {
        long[] temp = fibonacciGood(n - 1); // returns {Fn-1, Fn-2}
        long[] answer = {temp[0] + temp[1], temp[0]}; // we want {Fn, Fn-1}
        return answer;
    }
}
```

### Infinite recursion

To combat against infinite recursions, the designers of Java made an intentional decision to limit the overall space used to store activation frames for simultaneously active method calls. If this limit is reached, the Java Virtual Machine throws a `StackOverflowError`. A typical value might allow upward of 1000 simultaneous calls. While not a problem to `binarySearch`, could be a problem to other recursive methods. You can reconfigure the Virtual Machine to allow for greater space for nested method calls, or you could use traditional loops instead of recursion.

### Eliminating Tail recursion

We can use the stack data structure, which we will introduce in Section 6.1, to convert a recursive algorithm into a nonrecursive algorithm by managing the nesting of the recursive structure ourselves.

**tail recursion:** A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context. They can be automatically reimplemented nonrecursively by enclosing the body in a loop for repetition, and replacing a recursive call with new parameters by a reassignment of the existing parameters to those value. many programming language implementations may convert tail recursions in this way as an optimization.

```
1  /** Returns true if the target value is found in the data array. */
2  public static boolean binarySearchIterative(int[] data, int target) {
3      int low = 0;
4      int high = data.length - 1;
5      while (low <= high) {
6          int mid = (low + high) / 2;
7          if (target == data[mid])
8              return true;
9          else if (target < data[mid])
10             high = mid - 1;
11          else
12             low = mid + 1;
13     }
14     return false;
15 }
```

```
int low, int high) {
    if (low > high)
        return false;
    else {
        int mid = (low + high) / 2;
        if (target == data[mid])
            return true;
        else if (target < data[mid])
            return binarySearch(data, target, low, mid - 1);
        else
            return binarySearch(data, target, mid + 1, high);
    }
}
```

Instead of calling yourself again. You just update the fields within your method and run the while loop again (with half the interval) instead of creating a new stack frame with a recursion call.

Recursion exercises

► What is the tightest bound on the runtime of method  $f(\text{int } x)$ ?

```

1 public static int f(int x) {
2     if(x < 1) { // 1
3         return 1; // 1
4     } else {
5         return f(x-1) + f(x-1); // T(n-1) + T(n-1) + 2
6     }
7 } // Total: 2 (if n < 1)
8 // Total: 2T(n-1) + 3 (else)
    
```

$$T(n) = c_1 + 2T(n - 1) \quad \text{if } n > 0$$

$$T(0) = c_0$$

Where  $T(n)$  is a function of  $n$ . But to make a Big-Oh notation of the original algorithm we need to rephrase  $T(n)$  to a different form, non dependent on  $T$ :

Unfolding of the recurrence equation

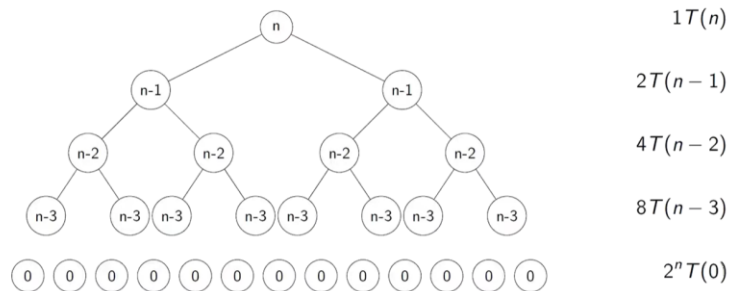
$$T(n) = c_1 + 2T(n - 1) \quad \text{if } n > 0$$

$$T(0) = c_0$$

$T(n) = c_1 + 2T(n - 1)$	by $T(n) = c_1 + 2T(n - 1)$
$= c_1 + 2(c_1 + 2T(n - 2))$	by $T(n - 1) = c_1 + 2T(n - 2)$
$= 3c_1 + 4T(n - 2)$	by arithmetic
$= 3c_1 + 4(c_1 + 2T(n - 3))$	by $T(n - 2) = c_1 + 2T(n - 3)$
$= 7c_1 + 8T(n - 3)$	by arithmetic
$= (2^k - 1)c_1 + 2^k T(n - k)$	repeat $k$ times
$= (2^n - 1)c_1 + 2^n T(n - n)$	by letting $k = n$
$= (2^n - 1)c_1 + 2^n c_0$	by $T(0) = c_0$

So we have that  $f(n)$  is  $O(2^n)$ . But there's an alternative way to do this:

Make an educated guess



$$(1 + 2 + 4 + 8 + \dots + 2^{n-1})c_1 + 2^n c_0 = (2^n - 1)c_1 + 2^n c_0$$

But this does not count as a formal proof.

And prove it is correct

Theorem  
Given that

$$\begin{aligned} T(n) &= c_1 + 2T(n-1) \quad \text{if } n > 0 \\ T(0) &= c_0 \end{aligned}$$

Then  $T(n) = (2^n - 1)c_1 + 2^n c_0$  for all  $n \geq 0$

Proof.

► Case  $k = 0$ :

$$\begin{aligned} T(0) &= c_0 && \text{by } T(0) = c_0 \\ &= (2^0 - 1)c_1 + 2^0 c_0 && \text{by arithmetic} \end{aligned}$$

► Case  $k > 0$ :

$$\text{IH: } T(k) = (2^k - 1)c_1 + 2^k c_0$$

$$\begin{aligned} T(k+1) &= c_1 + 2T(k) && \text{by } T(n) = c_1 + 2T(n-1) \\ &= c_1 + 2((2^k - 1)c_1 + 2^k c_0) && \text{by the induction hypothesis (IH)} \\ &= c_1 + 2(2^k c_1 - c_1 + 2^k c_0) && \text{by the by arithmetic} \\ &= c_1 + 2^{k+1} c_1 - 2c_1 + 2^{k+1} c_0 && \text{by the by arithmetic} \\ &= (2^{k+1} - 1)c_1 + 2^{k+1} c_0 && \text{by the by arithmetic} \end{aligned}$$

since  $k$  is an arbitrary integer, the statement holds for all  $n \geq 0$

Q.E.D

## Exercise

Let's go back to the splitting problem about the *SRList* from lecture 1. Given that *SRList* is sorted, there is an algorithm that solves that problem with a better runtime than  $O(n)$ .

Hint1: you might want to use the method `List.subList(int fromIndex, int toIndex)`, which splits the list in  $O(1)$  time.

Hint2: you probably want to use a recursive method.

- Make an algorithm that splits the *SRList* in 2 lists
- Give a tight bound on the runtime of this algorithm
- Implement the algorithm
- Extra: Do an empirical analysis to check whether your algorithm and implementation are correct

**Answer:**

```

/**
 * Gets the index of a sorted list in  $O(\log n)$  (Short signature)
 */
public static int getIndex(List<Integer> list, int target) {
    return getIndex(list, target, 0, list.size());
}

/**
 * Gets the index of a sorted list element in  $O(\log n)$  (full signature)
 *
 */
public static int getIndex(List<Integer> list, int target, int low, int high)
{
    if (low > high)
        throw new ArrayIndexOutOfBoundsException("Ranking not in the list");
    else {
        int mid = (low + high) / 2; //we start at the middle
        if (target == list.get(mid))
            return mid; // found a match, returns INDEX
        else if (target < list.get(mid)) // recur left of the middle
            return getIndex(list, target, low, mid - 1);
        else
            return getIndex(list, target, mid + 1, high); //recur right of
the middle
    }
}

/**
 * Takes an SR List and adds the split contents into a lower an upper list,
in less than  $O(n)$ 
 *
 * @param SR - List with players rankings (assume it's already sorted)
 * @param lower - lower bound list reference (assume it's already empty)
 * @param upper - upper bound list reference (assume it's already empty)
 */
public static void SplitLog(List<Integer> SR, List<Integer> lower,
List<Integer> upper, int splitValue) {
    int index = getIndex(SR, splitValue); //  $O(\log n)$  search -> c1
 * log n
    lower.addAll(SR.subList(0, index)); //interval is [0,index) -> c2
    upper.addAll(SR.subList(index, SR.size())); //interval is [0,index) ->
c3
} //  $O(\log n)$ 

```



## Binary Search

BinarySearch(data,target,low,high):

- ▶ if  $low = high$ : c
  - ▶ return  $low$  c
- ▶  $mid \leftarrow (low + high)/2$  c
- ▶ if  $data[mid] < targetvalue$ : c
  - ▶ return  $BinarySearch(data, target, mid + 1, high)$  T(n/2)
- ▶ else: T(n/2)
  - ▶ return  $BinarySearch(data, target, low, mid)$

$$T(1) = c_0$$

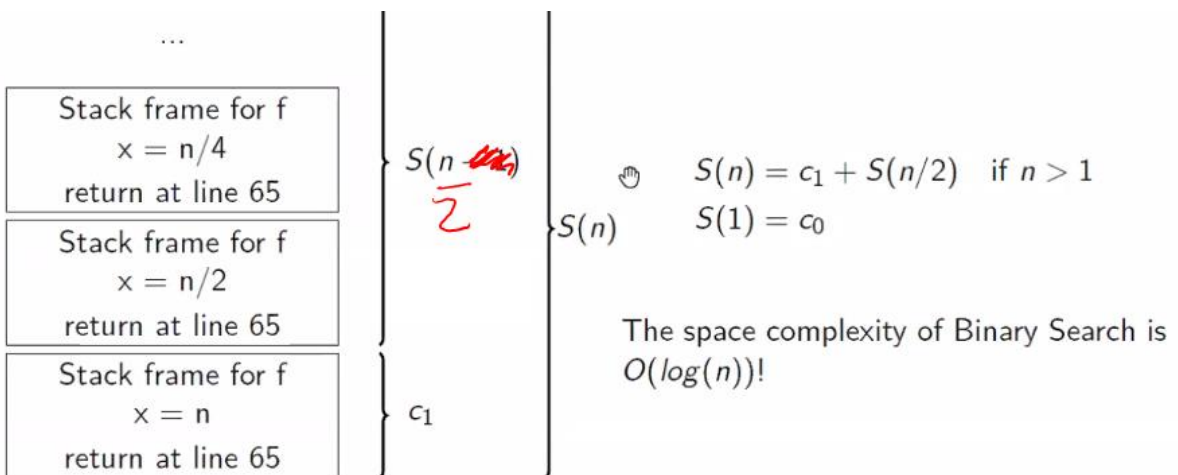
$$T(n) = c_1 + T(n/2) \quad \text{if } n > 1$$



What is the runtime complexity of this algorithm?

$  \begin{aligned}  T(n) &= c_1 + T(n/2) \\  &= c_1 + c_1 + T(n/4) \\  &= 2c_1 + T(n/4) \\  &= 2c_1 + c_1 + T(n/8) \\  &= 3c_1 + T(n/8) \\  &= k \cdot c_1 + T(n/2^k) \\  &= \log_2(n) \cdot c_1 + T(n/n) \\  &= \log_2(n) \cdot c_1 + c_0  \end{aligned}  $	<p>by <math>T(n) = c_1 + T(n/2)</math></p> <p>by <math>T(n/2) = c_1 + T(n/4)</math></p> <p>by arithmetic</p> <p>by <math>T(n/4) = c_1 + T(n/8)</math></p> <p>by arithmetic</p> <p>repeat k times</p> <p>by letting <math>k = \log_2(n)</math></p> <p>by <math>T(1) = c_0</math></p>
--	---

The runtime complexity of Binary Search is  $O(\log(n))!$



## Space complexity [Video]

We can also use  $O$  notation to compute the space complexity of an algorithm.

## Memory in Java: The call stack and heap

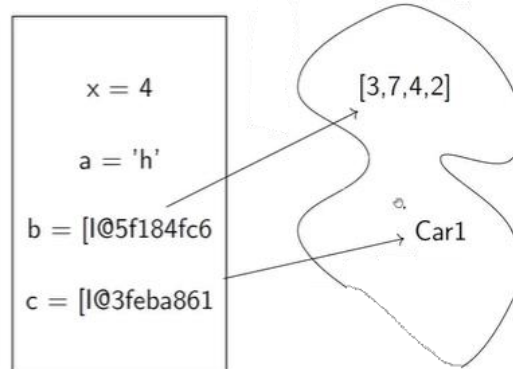
In Java (and most programming languages) the memory is organized in two parts:

**The call stack** contains:

- ▶ The method arguments (primitives and references)
- ▶ The local variables (primitives and references)
- ▶ The return address (more about that in a moment)

**The heap** contains:

- ▶ Arrays
- ▶ Objects



## The call stack

```

1  public static void main(String[] args) {
2      int a = 4;
3      char b = 'H';
4      String c = f(a);
5      System.out.println(b + c);
6  }
7
8  public static String f(int x) {
9      h(x);
10     char a = g();
11     return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

Stack frame for main
args =
a =
b =
c =

## The call stack

```

1  public static void main(String[] args) {
2      int a = 4;
3      char b = 'H';
4      String c = f(a);
5      System.out.println(b + c);
6  }
7
8  public static String f(int x) {
9      h(x);
10     char a = g();
11     return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

Stack frame for f

x =

a =

return at line 4

Stack frame for main

args =

a =

b =

c =

## The call stack

```

1  public static void main(String[] args) {
2      int a = 4;
3      char b = 'H';
4      String c = f(a);
5      System.out.println(b + c);
6  }
7
8  public static String f(int x) {
9      h(x);
10     char a = g();
11     return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

Stack frame for h

y =

return at line 9

Stack frame for f

x =

a =

return at line 4

Stack frame for main

args =

a =

b =

c =

## The call stack

```

1 public static void main(String[] args) {
2     int a = 4;
3     char b = 'H';
4     String c = f(a);
5     System.out.println(b + c);
6 }
7
8 public static String f(int x) {
9     h(x);
10    char a = g();
11    return a + "llo";
12 }
13
14 public static char g() {
15    char a = 'e';
16    h(5);
17    return a;
18 }
19
20 public static void h(int y) {
21    System.out.println(y);
22 }

```

Stack frame for f

x =

a =

return at line 4

Stack frame for main

args =

a =

b =

c =

## The call stack

```

1 public static void main(String[] args) {
2     int a = 4;
3     char b = 'H';
4     String c = f(a);
5     System.out.println(b + c);
6 }
7
8 public static String f(int x) {
9     h(x);
10    char a = g();
11    return a + "llo";
12 }
13
14 public static char g() {
15    char a = 'e';
16    h(5);
17    return a;
18 }
19
20 public static void h(int y) {
21    System.out.println(y);
22 }

```

Stack frame for g

a =

return at line 10

Stack frame for f

x =

a =

return at line 4

Stack frame for main

args =

a =

b =

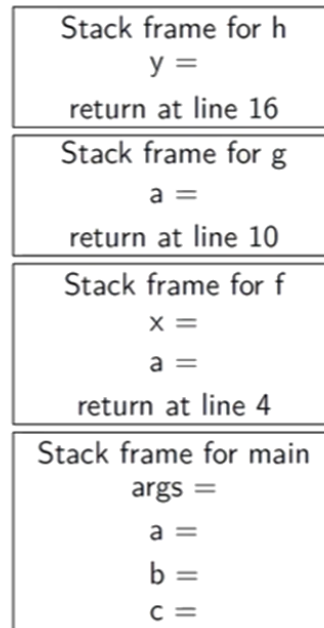
c =

## The call stack

```

1 public static void main(String[] args) {
2     int a = 4;
3     char b = 'H';
4     String c = f(a);
5     System.out.println(b + c);
6 }
7
8 public static String f(int x) {
9     h(x);
10    char a = g();
11    return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

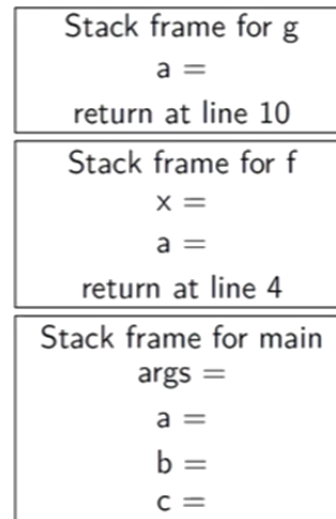


## The call stack

```

1 public static void main(String[] args) {
2     int a = 4;
3     char b = 'H';
4     String c = f(a);
5     System.out.println(b + c);
6 }
7
8 public static String f(int x) {
9     h(x);
10    char a = g();
11    return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```



## The call stack

```

1  public static void main(String[] args) {
2      int a = 4;
3      char b = 'H';
4      String c = f(a);
5      System.out.println(b + c);
6  }
7
8  public static String f(int x) {
9      h(x);
10     char a = g();
11     return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

Stack frame for f

x =

a =

return at line 4

Stack frame for main

args =

a =

b =

c =

## The call stack

```

1  public static void main(String[] args) {
2      int a = 4;
3      char b = 'H';
4      String c = f(a);
5      System.out.println(b + c);
6  }
7
8  public static String f(int x) {
9      h(x);
10     char a = g();
11     return a + "llo";
12 }
13
14 public static char g() {
15     char a = 'e';
16     h(5);
17     return a;
18 }
19
20 public static void h(int y) {
21     System.out.println(y);
22 }

```

Stack frame for main

args =

a =

b =

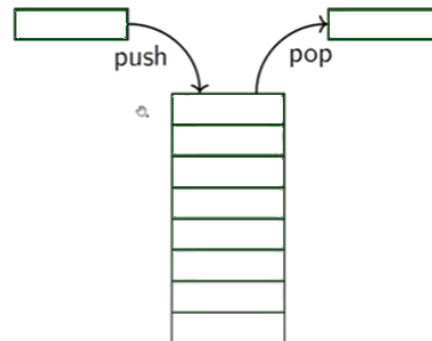
c =

## Stacks

A **stack** organizes a set of elements in a **Last In, First Out (LIFO)** manner

The two basic operations on a stack:

- ▶ pushing a new element on the stack
- ▶ popping an element from the stack



### Exercise

- ▶ What is the tightest bound on the space complexity of method  $f(\text{int } x)$ ?
- ▶ Extra: And what is the tightest bound on the space complexity of other methods you have seen?

```

1 public static int f(int x) {
2     if(x <= 1) {
3         return 1;
4     } else {
5         return f(x-1) * x;
6     }
7 }

```

If we have input  $n$ , we have that there will be  $n, n-1, n-2, \dots, 3, 2, 1$  calls.  $[1 \text{ to } n] = n$  calls

f(1) (basecase): x = 1 return to 5
f(2): x = 2 return to 5
...
f(n-1): x = n-1 return to 5
f(n): x = n return to somewhere

Within the body of a call we see that the amount of variables is constant. Therefore the space complexity of  $f(n) = c \cdot n$  which is  $O(n)$ .

## Week 2. Arrays, stacks and queues

### Arrays. Insertion and deletion

# Arrays

An array stores a **fixed-length** sequence of elements of the **same type**.  
Allows access to elements by integer **indices**, in  $O(1)$  time.

Book sections 3.1.1 and 3.1.2

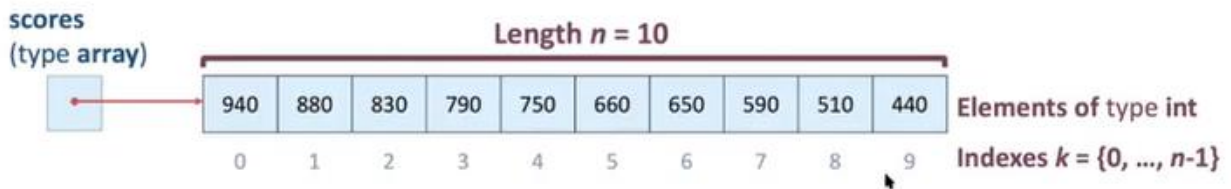
## Arrays of primitives

### In Java

Array is a **reference type**: variable of type array stores a reference to the actual array.

Example: scoreboard to keep 10 highest scores of a game, in order (non-increasing).

Array variable **scores** stores 10 highest scores as integers.



## Arrays of objects

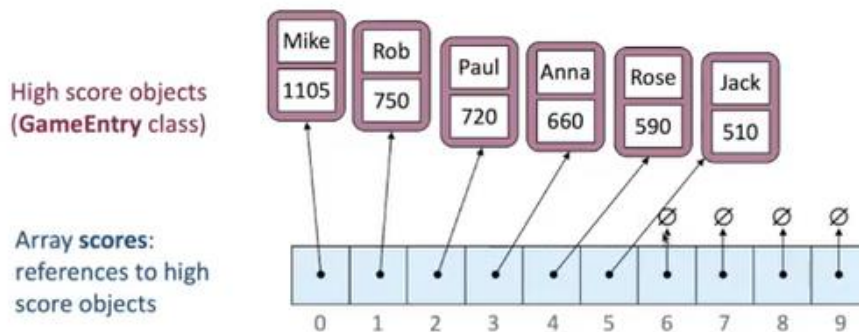
Java code 3.1,3.2: pages 94-95

```
public class GameEntry
implements Cloneable {
    protected String name;
    protected int score;
    ...
}
```

Class **GameEntry**: information on a game's high score + player name.

Array `scores`:

- information on the 10 highest scores in a game;
- the array is kept sorted.



$\emptyset = null$

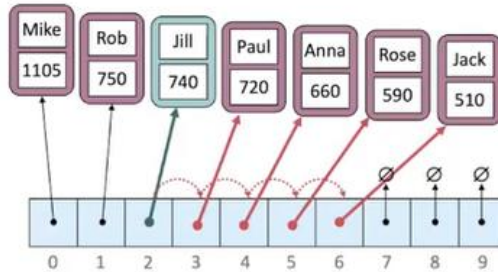


## Arrays: insert element

Java code fragment 3.3: pi

**Add new high scorer Jill with score 740 at index 2:**

Make room at index 2 by shifting subsequent elements forward (update references).



## Arrays: insert element

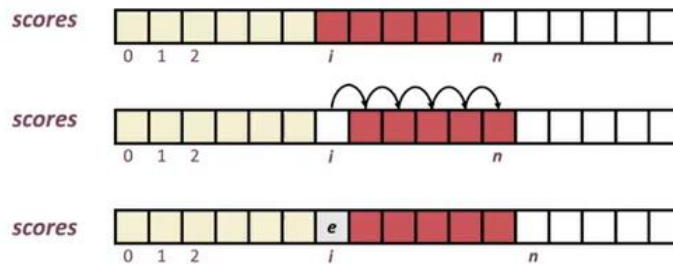
Java code fragment 3.3: page 96

Time  $O(n)$   
Space  $O(1)$

**Add element  $e$  to array  $scores$  at index  $i$ :**

Shift forward the  $n - i$  non-null elements  $scores[i]$  to  $scores[n-1]$ .

Set element at index  $i$  of array  $scores$  to  $e$ . Increment  $n$ .

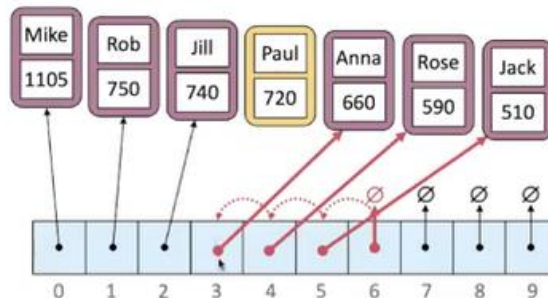


## Arrays: remove element

Java code

**Remove cheater Paul at index 3:**

Shift subsequent elements backward (update references).



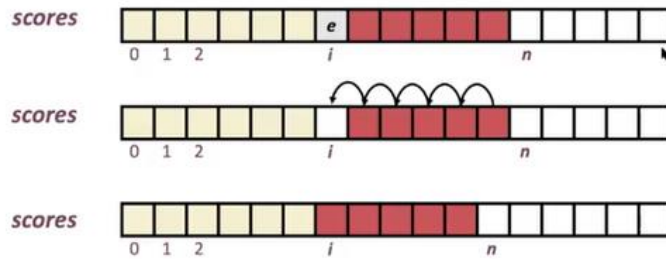
## Arrays: remove element

Java code fragment 3.4: page 9

Time  $O(n)$   
Space  $O(1)$

**Remove element  $e$  at index  $i$  of array  $scores$ :**

Shift backward the  $n - i - 1$  non-null elements  $scores[i+1]$  to  $scores[n-1]$ .  
Decrement  $n$ .



⌘

## Arrays: sort

**Algorithm sort ( $a$ )**

**Input:** An array  $a$  of  $n$  comparable elements.

**Output:** The same array  $a$  with its elements rearranged in a specific order (typically non-decreasing).



## Arrays: insertion sort

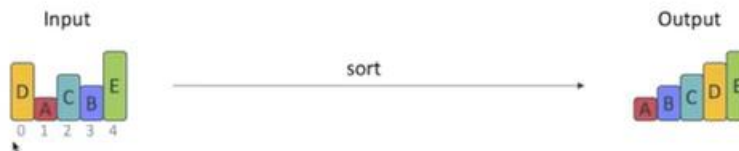
**InsertionSort( $a$ )**

for  $k$  from 1 to  $n-1$  do

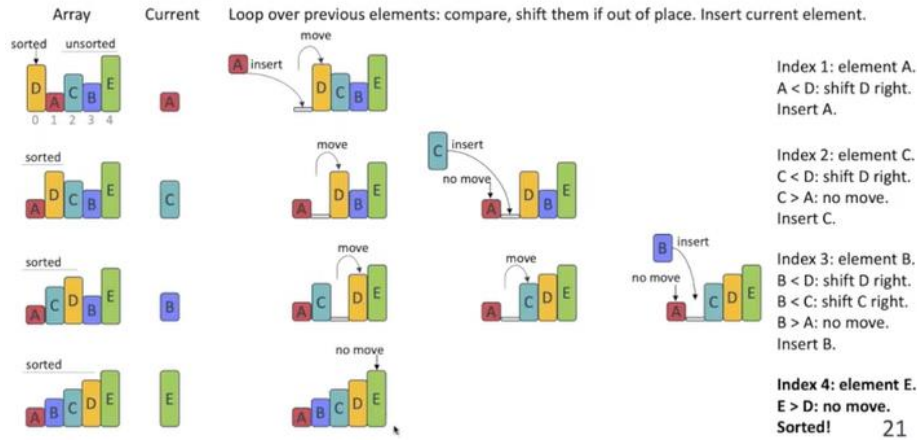
insert  $a[k]$  at its proper location within  $a[0], a[1], \dots, a[k-1]$

**In-place algorithm:** modifies the array directly (no new array to store result).

**Implicit output:** array is passed by reference, no need for explicit return.



## Arrays: insertion sort



## Arrays: insertion sort

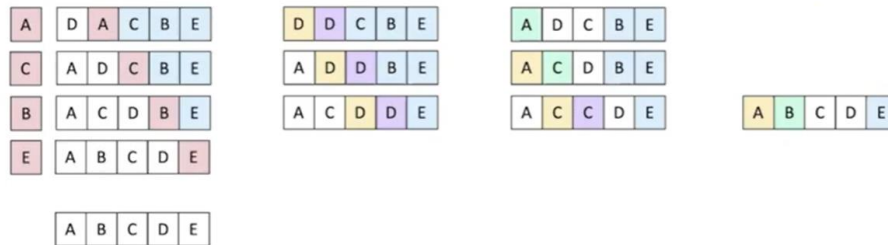
Java code fragment 3.6: page 101

Iterate over the array (index  $k$  from 1 to  $n - 1$ )

$$c \cdot (1 + 2 + \dots + (n - 1)) = c \cdot \sum_{k=1}^{n-1} k = c \cdot \frac{(n - 1)n}{2}$$

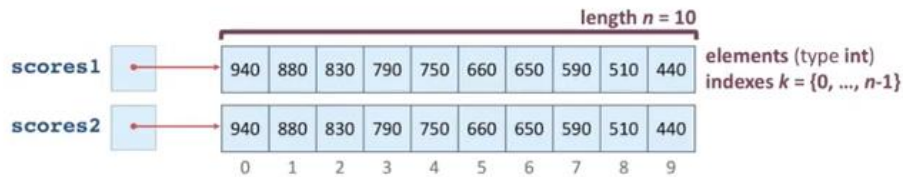
Time  $O(n^2)$   
Space  $O(1)$

At iteration  $k$ : compare with previous, shift if necessary: worst-case  $c \cdot k$  comparisons



## Arrays: testing for equality

```
int[] scores1 = {940, 880, 830, 790, 750, 660, 650, 590, 510, 440};
int[] scores2 = {940, 880, 830, 790, 750, 660, 650, 590, 510, 440};
```



### Tests for equality

```
scores1 == scores2
```

What is compared? array variable (reference) Result? false Time?  $O(1)$

```
scores1.equals(scores2)
```

(defined for Object, compares with == by default; should be overridden)

array variable (reference) false  $O(1)$

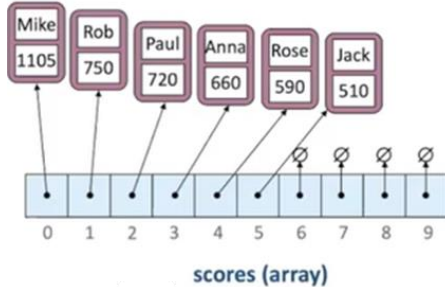
```
java.util.Arrays.equals(scores1, scores2)
```

(loops over  $k$  to compare  $a[k]$  against  $b[k]$ ; uses  $a[k] == b[k]$  for primitives, calls  $a[k].equals(b[k])$  for objects)

array's elements true  $O(n)$

## Arrays: clone

```
1 GameEntry[] backup = new GameEntry[15];
2 backup = scores.clone();
3 scores[4].setScore(600);
```



What do instructions 1 to 3 do?

- 1 Declares array **backup** for 15 GameEntry.
- 2 Clones scores and assigns to **backup**.
- 3 Sets score of element at index 4 in array **scores** to 600.

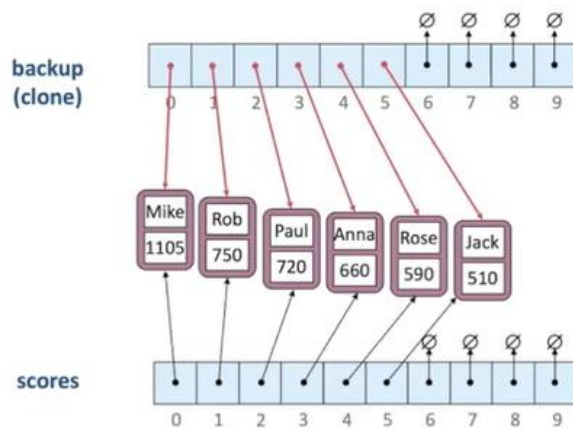
What is the value of

backup.length  
after 1? 15 after 2? 10

backup[4].getScore()  
after 1? NullPointerException after 2? 590 after 3? 600

Why is backup[4].getScore() 600 and not 590?

## Arrays: clone



Clone makes a **shallow copy**

```
backup = scores.clone();
```

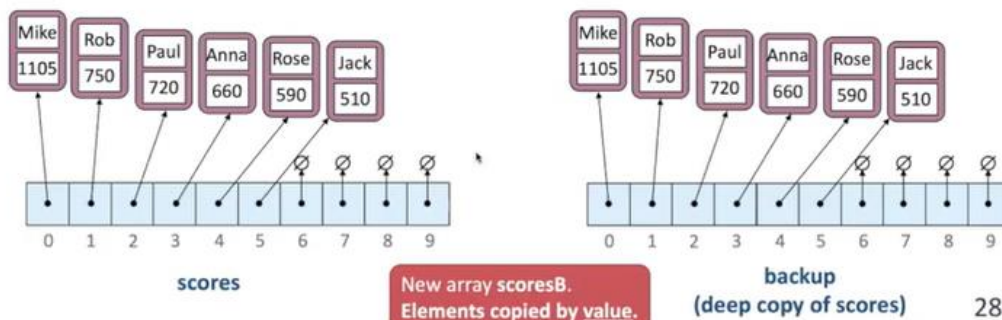
As a result, backup[4] and scores[4] point to the same object.

New array backup.  
Elements copied by **reference**.

## Arrays: deep copy

**We want to preserve the integrity of our backup!**  
It should not change when we alter the original array scores.

**We need a deep copy. How do we achieve this?**



New array scoresB.  
Elements copied by **value**.

backup  
(deep copy of scores) 28

## Arrays: deep copy

```
// clone() signature in Object class
public class Object {
    protected native Object clone()
        throws CloneNotSupportedException;
    ...
}

// Cloneable indicates Object's clone()
// can be used to copy instances of a class
public interface Cloneable { }

// Class overrides Object method clone()
public class GameEntry
    implements Cloneable interface {
    ...
    public Object clone() {
        ...
    }
}
```

### Making a deep copy

1. GameEntry class has to implement interface Cloneable, and override java.lang.Object clone().
2. Create new array and copy elements one by one into new array by invoking clone() on each of them.

```
GameEntry[] scoresB = new GameEntry[scoresA.length];
for(int k = 0; k < scoresA.length; k++)
    scoresB[k] = (GameEntry) scoresA[k].clone();
```

## Arrays: expand capacity

How can we increase the capacity of array `scores` to keep information about the 20 highest scores (instead of 10)?

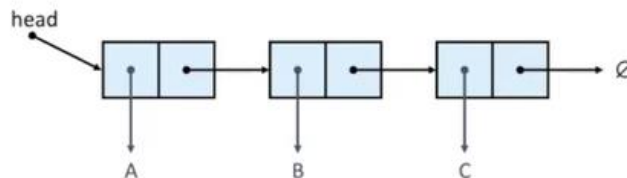
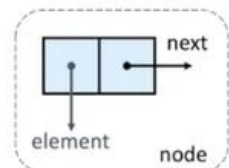
**Solution: make a (deep) copy into a larger array.**

```
GameEntry[] larger = new GameEntry[2*scores.length];
for(int k = 0; k < scores.length; k++)
    larger[k] = (GameEntry) scores[k].clone();
scores = larger;
```

Array expansion with at most  $n$  additional positions is  $O(n)$  time and  $O(n)$  space.

Linked Lists  
Singly Linked lists

## Singly linked list



A List is a sequence of nodes starting from a **head** pointer:

- **tail**: last node of the list; its **next** reference is **null**;
- **traversal**: moving through the list following nodes' **next** references.

Node stores:

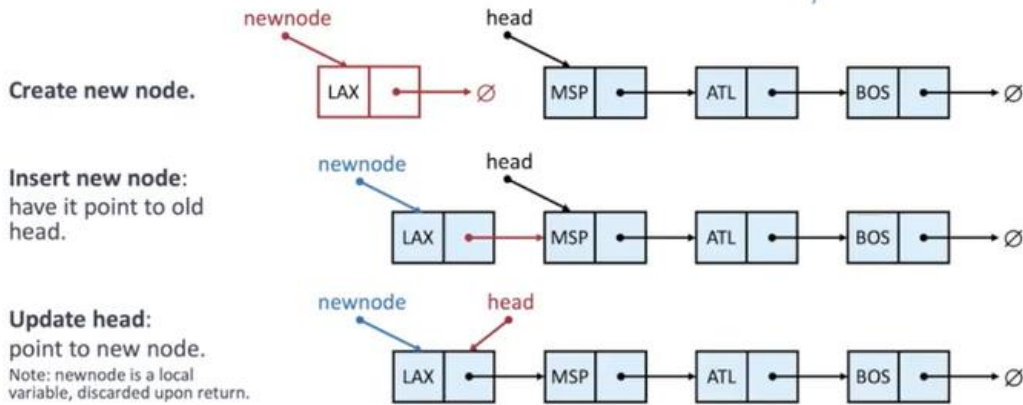
- (reference to) **element**;
- reference to **next** node in the list.

```
public class Node<E> {
    private E element;
    private Node<E> next;
    public Node(E e, Node<E> n) {
        element = e;
        next = n;
    }
    public E getElement() {
        return element;
    }
    public Node<E> getNext() {
        return next;
    }
    public void setNext(Node<E> n) {
        next = n;
    }
}
```

## SLL: insert at the head

Time  $O(1)$   
Space  $O(1)$

```
Code 3.10, 3.14
public void addFirst(E e) {
    head = new Node<>(e, head);
    if (size == 0)
        tail = head;
    size++;
}
```

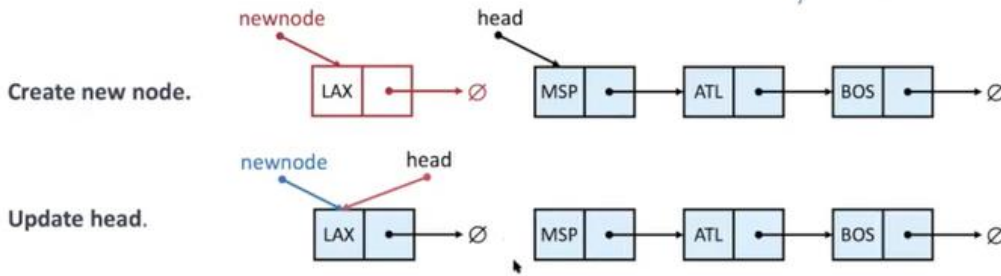


What happens if we first update the head, after creating the new node?

37

## SLL: insert at the head

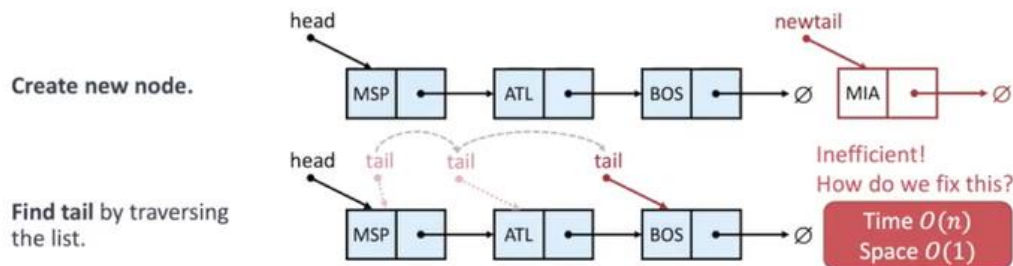
```
Code 3.10, 3.14
public void addFirst(E e) {
    head = new Node<>(e, head);
    if (size == 0)
        tail = head;
    size++;
}
```



Lost reference to the list!  
List unreachable, cannot perform insertion.

## SLL: insert at the tail

Java code 3.11



Inefficient!  
How do we fix this?

Time  $O(n)$   
Space  $O(1)$

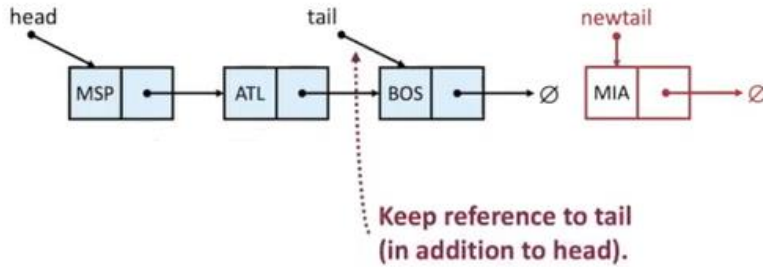
## SLL: insert at the tail

Time  $O(1)$   
Space  $O(1)$

```

Java code 3.11
public void addLast(E e) {
    Node<E> n =
        new Node<>(e, null);
    if (isEmpty()) head = n;
    else tail.setNext(n);
    tail = n;
    size++;
}
    
```

Create new node.



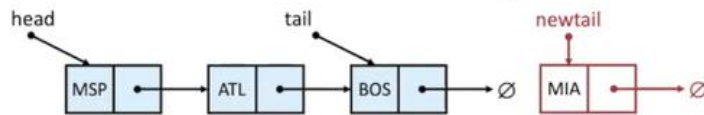
## SLL: insert at the tail

Time  $O(1)$   
Space  $O(1)$

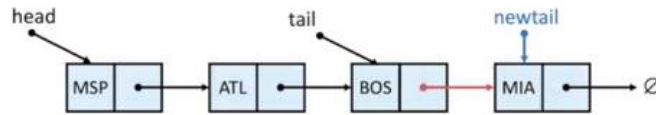
```

Java code 3.11
public void addLast(E e) {
    Node<E> n =
        new Node<>(e, null);
    if (isEmpty()) head = n;
    else tail.setNext(n);
    tail = n;
    size++;
}
    
```

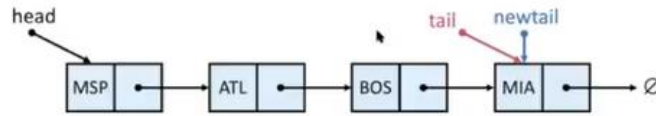
Create new node.



Update tail's next:  
point to new node.



Update tail:  
point to new node.



44

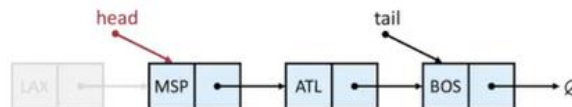
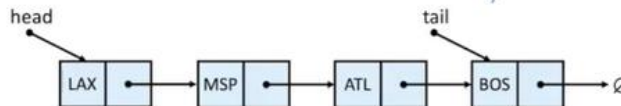
## SLL: remove head

Time  $O(1)$   
Space  $O(1)$

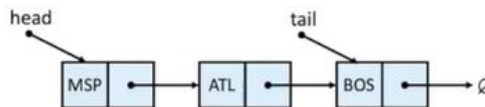
```

Java code 3.12
public void removeFirst(E e){
    if (isEmpty()) return null;
    E e = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0) tail = null;
    return e;
}
    
```

Update head: point  
to current head's  
next.



Garbage collector will  
reclaim old head  
node (if there are no  
other references to it).

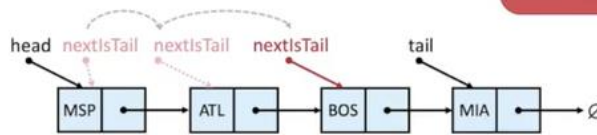


46

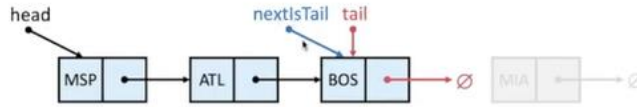
### SLL: remove tail

$O(n)$  time.  
Reference to tail  
doesn't help.

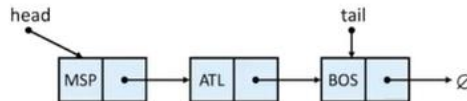
**Find last but one node:**  
traverse list while checking  
`node.next` and  
`node.next.next`.



**Update tail:**  
point to last but one node.  
**Update tail's next:**  
set to null.



Garbage collector will  
reclaim old tail node  
(if there are no references to it).

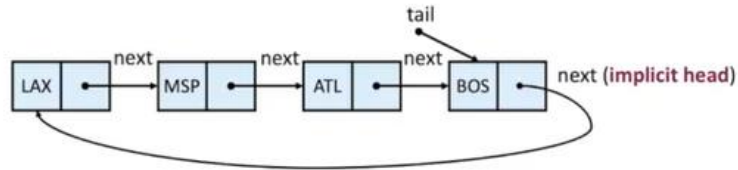


49

### Circularly Linked Lists

### Circularly linked list

```
Java code 3.15
public class CircularlyLinkedList
```



### Circularly linked list:

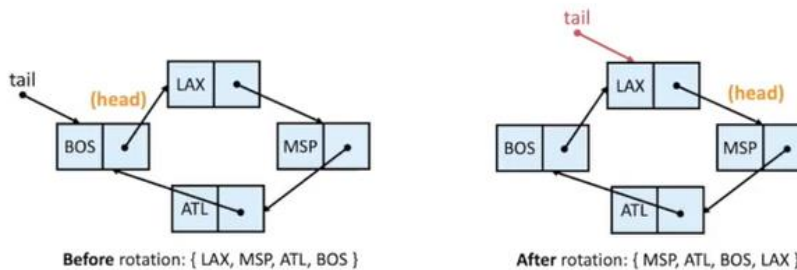
- Singly linked list (SLL) with **tail's next** pointing to head node, rather than **null** (no explicit head).
- Operations: `addFirst(e)`, `addLast(e)`, `removeFirst()` as in SLL + update method `rotate()`.

Only explicit reference to the tail. No head reference

### CLL: rotate

Time  $O(1)$   
Space  $O(1)$

```
Java code 3.15
public void rotate(){
    if (tail != null)
        tail = tail.getNext();
}
```



**Rotate method:** updates the tail by following its next reference (implicit head).

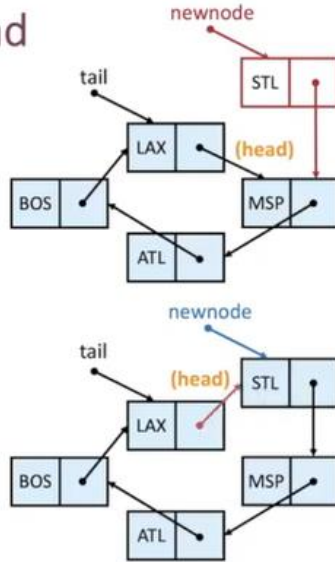


### CLL: insert at head

Create new node with:  
 - element;  
 - next reference pointing to tail's next.

Update tail's next: point to new node.

(newnode is a local variable within method addFirst(), reference is not kept)



```

Java code 3.15
public void addFirst(E e){
    if (size == 0) {
        tail = new Node<>(e, null);
        tail.setNext(tail);
    }
    else {
        Node<E> n = new Node<>(e,
            tail.getNext());
        tail.setNext(n);
    }
    size++;
}
    
```

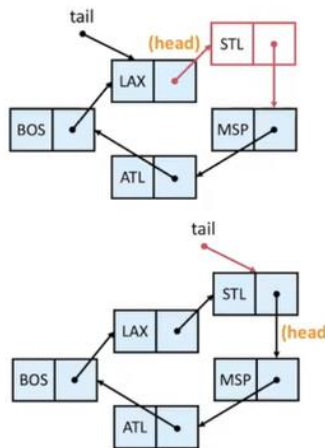
Time  $O(1)$   
 Space  $O(1)$

56

### CLL: insert at tail

Insert at the head.

Rotate.



```

Java code 3.15
public void addLast(E e){
    addFirst(e);
    tail = tail.getNext();
}
    
```

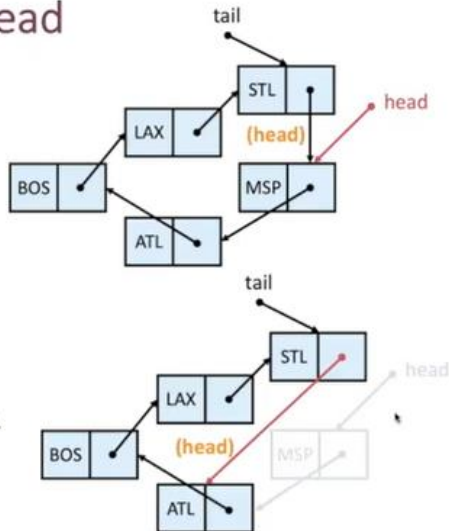
Time  $O(1)$   
 Space  $O(1)$

58

### CLL: remove head

Get explicit pointer to head.  
 If identical to tail, there's only one node: set tail to null.

Otherwise:  
 Set tail's next to head's next.



```

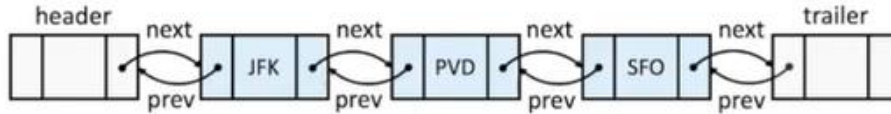
Java code 3.15
public E removeFirst(){
    if (isEmpty())
        return null;
    Node<E> h = tail.getNext();
    if (h == tail)
        tail = null;
    else
        tail.setNext(h.getNext());
    size--;
    return h.getElement();
}
    
```

Time  $O(1)$   
 Space  $O(1)$

60

Doubly linked lists

# Doubly linked list (DLL)



### Doubly linked list

Each node contains:

- **element**;
- **next**: reference to next node;
- **prev**: reference to previous node.

Sentinels **header** and **trailer**:

- dummy nodes to simplify insertions and deletions (avoid special cases).

```
public class DoublyLinkedList<E> {
    private static class Node<E> {
        private E element;
        private Node<E> prev;
        private Node<E> next;
        public Node(E e, Node<E> p, Node<E> n){
            ...
        }
        public E getElement(){ return element; }
        public Node<E> getPrev(){ return prev; }
        public Node<E> getNext(){ return next; }
        public void setPrev(E p){ prev = p; }
        public void setNext(E n){ next = n; }
    }
    private Node<E> header;
    private Node<E> trailer;
    private int size = 0;
    ...
}
```

64

## DLL: create/initialize

Create **header** node:

- **element**: null;
- **prev** reference: null;
- **next** reference: null.



Time  $O(1)$   
Space  $O(1)$

Create **trailer** node:

- **element**: null;
- **prev** reference: **header**;
- **next** reference: null.



Point **header's next** to **trailer**.

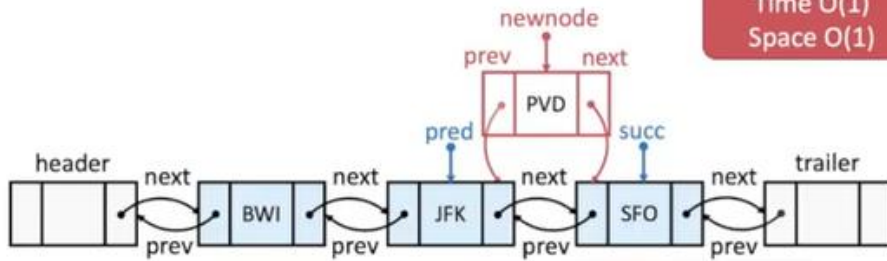


```
Java code 3.16, 3.17
public DoublyLinkedList(){
    header = new Node<>(null, null, null);
    trailer = new Node<>(null, header, null);
    header.setNext(trailer);
}
```

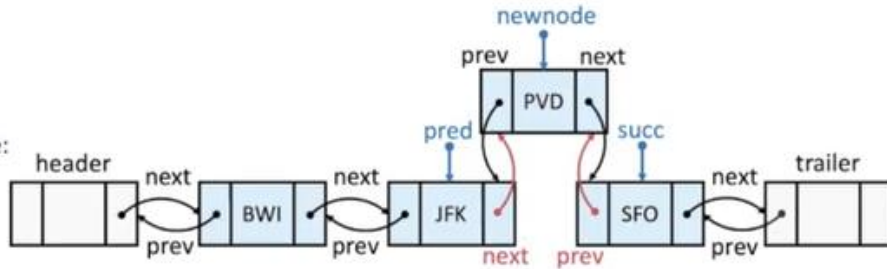
## DLL: insert

Time  $O(1)$   
Space  $O(1)$

- Create new node:
- element  $e$ ;
  - $prev$  point to  $pred$ ;
  - $next$  point to  $succ$ .



- Set references to new node:
- $pred$ 's  $next$ ;
  - $succ$ 's  $prev$ .



Java code 3.17: page 126

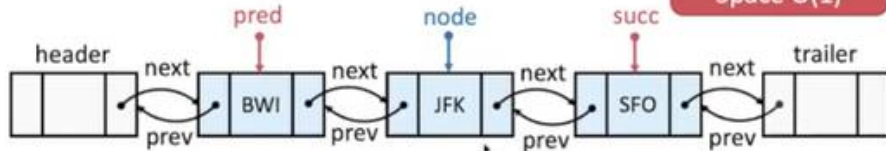
```
private void addBetween(E e, Node<E> pred, Node<E> succ) { ... }
```

67

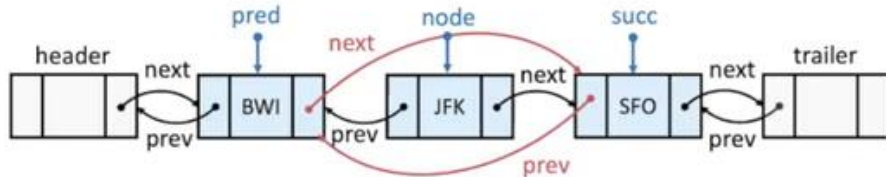
## DLL: remove

Time  $O(1)$   
Space  $O(1)$

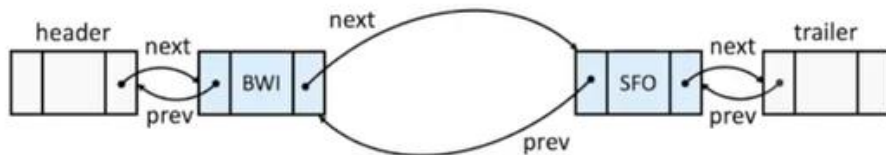
- Get predecessor and successor nodes:
- Set  $pred$  as  $node$ 's  $prev$ ;
  - Set  $succ$  as  $node$ 's  $next$ .



- Link out old node:
- Set  $pred$ 's  $next$  as  $succ$ ;
  - Set  $succ$ 's  $prev$  as  $pred$ .



Let garbage collection reclaim removed node.



Java code 3.17

```
private E remove(Node<E> node) { ... }
```

70

## DLL: insert at the head/tail

Time  $O(1)$   
Space  $O(1)$

How do we insert or remove at the head or tail of a DLL?  
Hint: make use of the following methods

```
private void addBetween(E e, Node<E> predecessor, Node<E> successor)
private E remove(Node<E> node)
```

Insert at the head: 

```
public void addFirst(E e) {
    addBetween(e, header, header.getNext());
}
```

Insert at the tail: 

```
public void addLast(E e) {
    addBetween(e, trailer.getPrev(), trailer);
}
```

Java code 3.17: page 126

72

## DLL: remove head/tail

Time  $O(1)$   
Space  $O(1)$

How do we insert or remove at the head or tail of a DLL?  
Hint: make use of the following methods

```
private void addBetween(E e, Node<E> predecessor, Node<E> successor)
private E remove(Node<E> node)
```

Remove head: 

```
public E removeFirst() {
    if(isEmpty()) return null;
    else return remove(header.getNext());
}
```

Remove tail: 

```
public E removeLast() {
    if(isEmpty()) return null;
    else return remove(trailer.getPrev());
}
```

Java code 3.17: page 126

73

For a(n) array/list with n elements:

	Array	SLL	CLL	DLL	
Access element (by index)	<b><math>O(1)</math></b>	$O(n)$	$O(n)$	$O(n)$	
Search element	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Insertion					
<b>At head</b>	$O(n)$	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<code>addFirst(e)</code>
<b>At tail</b>	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<code>addFirst(e)</code>
Deletion					
<b>At head</b>	$O(n)$	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<b><math>O(1)</math></b>	<code>removeFirst( )</code>
<b>At tail</b>	<b><math>O(1)</math></b>	$O(n)$	$O(n)$	<b><math>O(1)</math></b>	<code>removeLast( )</code>
Clone/copy	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

**Advantages:** efficient expansion, efficient insertion/deletion at head and tail.

**Disadvantages:** accessing elements requires traversal, added space for **prev** references.

## Arrays and lists [Chapter 3 extras]

## java.util Methods for Arrays

- equals(A, B):** Returns true if and only if the array *A* and the array *B* are equal. Two arrays are considered equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal. That is, *A* and *B* have the same values in the same order.
- fill(A, x):** Stores value *x* in every cell of array *A*, provided the type of array *A* is defined so that it is allowed to store the value *x*.
- copyOf(A, n):** Returns an array of size *n* such that the first *k* elements of this array are copied from *A*, where  $k = \min\{n, A.length\}$ . If  $n > A.length$ , then the last  $n - A.length$  elements in this array will be padded with default values, e.g., 0 for an array of **int** and **null** for an array of objects.
- copyOfRange(A, s, t):** Returns an array of size  $t - s$  such that the elements of this array are copied in order from  $A[s]$  to  $A[t - 1]$ , where  $s < t$ , padded as with `copyOf()` if  $t > A.length$ .
- toString(A):** Returns a String representation of the array *A*, beginning with `[`, ending with `]`, and with elements of *A* displayed separated by string `,` . The string representation of an element  $A[i]$  is obtained using `String.valueOf(A[i])`, which returns the string `"null"` for a **null** reference and otherwise calls `A[i].toString()`.
- sort(A):** Sorts the array *A* based on a natural ordering of its elements, which must be comparable. Sorting algorithms are the focus of Chapter 12.
- binarySearch(A, x):** Searches the *sorted* array *A* for value *x*, returning the index where it is found, or else the index of where it could be inserted while maintaining the sorted order. The binary-search algorithm is described in Section 5.1.3.

## java.util Methods for Random

- nextBoolean():** Returns the next pseudorandom **boolean** value.
- nextDouble():** Returns the next pseudorandom **double** value, between 0.0 and 1.0.
- nextInt():** Returns the next pseudorandom **int** value.
- nextInt(n):** Returns the next pseudorandom **int** value in the range from 0 up to but not including *n*.
- setSeed(s):** Sets the seed of this pseudorandom number generator to the **long** *s*.

```

/**
 * Program showing some array uses.
 */
public class RandomArray {
    public static void main(String[] args) {
        int data[] = new int[10];
        Random rand = new Random(); // a pseudo-random number generator
        rand.setSeed(System.currentTimeMillis()); // use current time as a
seed
        // fill the data array with pseudo-random numbers from 0 to 99,
inclusive
        for (int i = 0; i < data.length; i++)
            data[i] = rand.nextInt(100); // the next pseudo-random number
        int[] orig = Arrays.copyOf(data, data.length); // make a copy of the
data array
        System.out.println("arrays equal before sort: " + Arrays.equals(data,
orig));
        Arrays.sort(data); // sorting the data array (orig is unchanged)
        System.out.println("arrays equal after sort: " + Arrays.equals(data,
orig));
        System.out.println("orig = " + Arrays.toString(orig));
        System.out.println("data = " + Arrays.toString(data));
    }
}

```

## Cryptography

This field involves the process of encryption, in which a message, called the plaintext, is converted into a scrambled message, called the ciphertext. Decryption: turning a ciphertext back into its original plaintext.

Caesar cipher: simplest encryption = offsetting all the characters of a string by a fixed constant, wrapping around Z->A

Strings in java are immutable, so we can't change the string characters but we would need to create an equivalent array of characters, edit the array, and then reassemble a (new) string based on the array.

## Cryptogrphay Implementation CaesarCipher available in IntelliJ workspace

### Two-Dimensional Arrays

In a two-dimensional array, where we use two indices, say i and j, the first index usually refers to a row number and the second to a column number.

Eventhough Java arrays can only be one-dimensional what we are actually doing is an array of arrays. Nevertheless, Java provides a built-in pseudo 2 dimensional array declaration method that makes it feel like a real 2-dimensional array:

```
int[ ][ ] data = new int[8][10];
```

This statement creates a two-dimensional "array of arrays," data, which is 8×10, having 8 rows and 10 columns. Data is an array of length 8 such that each, element of data is an array of length 10 of integers.

[TicTacToe Implementation available in IntelliJ workspace](#)

[Singly Linked List Implementation available in IntelliJ workspace](#)

[Circularly Linked Lists](#)

[Round-Rogin scheduling](#)

In order to support the responsiveness of an arbitrary number of concurrent processes, most operating systems allow processes to effectively share use of the CPUs, using some form of an algorithm known as round-robin scheduling. A process is given a short turn to execute, known as a time slice, but it is interrupted when the slice ends, even if its job is not yet complete. Each active process is given its own time slice, taking turns in a cyclic order. New processes can be added to the system, and processes that complete their work can be removed.

(So, it's a circular linked list waiting pool and we just append processes to the tail and remove them as they are completed).

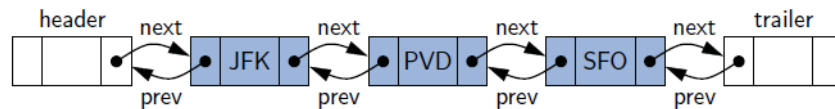
[Circularly Linked List Implementation available in IntelliJ workspace](#)

[Doubly Linked Lists](#)

[Sentinels](#)

### Header and Trailer Sentinels

In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a *header* node at the beginning of the list, and a *trailer* node at the end of the list. These “dummy” nodes are known as *sentinels* (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure 3.19.



**Figure 3.19:** A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

### Full implementation in IntelliJ workspace

`size()`: Returns the number of elements in the list.

`isEmpty()`: Returns **true** if the list is empty, and **false** otherwise.

`first()`: Returns (but does not remove) the first element in the list.

`last()`: Returns (but does not remove) the last element in the list.

`addFirst(e)`: Adds a new element to the front of the list.

`addLast(e)`: Adds a new element to the end of the list.

`removeFirst()`: Removes and returns the first element of the list.

`removeLast()`: Removes and returns the last element of the list.

If `first()`, `last()`, `removeFirst()`, or `removeLast()` are called on a list that is empty, we will return a **null** reference and leave the list unchanged.

## Equivalence testing

The author of each class has a responsibility to provide an implementation of the equals method, which overrides the one inherited from Object (that only checks `a == b`). The equals method must follow the mathematical definition of “equivalence relation”:

- Treatment of null:** For any nonnull reference variable `x`, the call `x.equals(null)` should return **false** (that is, nothing equals **null** except **null**).
- Reflexivity:** For any nonnull reference variable `x`, the call `x.equals(x)` should return **true** (that is, an object should equal itself).
- Symmetry:** For any nonnull reference variables `x` and `y`, the calls `x.equals(y)` and `y.equals(x)` should return the same value.
- Transitivity:** For any nonnull reference variables `x`, `y`, and `z`, if both calls `x.equals(y)` and `y.equals(z)` return **true**, then call `x.equals(z)` must return **true** as well.

### Available Equals for arrays

- `a == b`: Tests if `a` and `b` refer to the same underlying array instance.
- `a.equals(b)`: Interestingly, this is identical to `a == b`. Arrays are not a true class type and do not override the `Object.equals` method.
- `Arrays.equals(a,b)`: This provides a more intuitive notion of equivalence, returning **true** if the arrays have the same length and all pairs of corresponding elements are “equal” to each other. More specifically, if the array elements are primitives, then it uses the standard `==` to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons `a[k].equals(b[k])` in evaluating the equivalence.

To support the more natural notion of multidimensional arrays being equal if they have equal contents, the class provides an additional method:

- `Arrays.deepEquals(a,b)`: Identical to `Arrays.equals(a,b)` except when the elements of `a` and `b` are themselves arrays, in which case it calls `Arrays.deepEquals(a[k],b[k])` for corresponding entries, rather than `a[k].equals(b[k])`.

## Cloning

Each class in Java is responsible for defining whether its instances can be copied, and if so, precisely how the copy is constructed. The universal Object superclass defines a method named `clone`, which can be used to produce what is known as a shallow copy of an object. This uses the standard assignment semantics to assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied.

A shallow copy is not always appropriate for all classes, and therefore, Java intentionally disables use of the `clone()` method by declaring it as **protected**, and by having it throw a **CloneNotSupportedException** when called.



The author of a class must explicitly declare support for cloning by formally declaring that the class implements the Cloneable interface, and by declaring a public version of the clone( ) method. That public method can simply call the protected one to do the field-by-field assignment that results in a shallow copy, if appropriate. However, for many classes, the class may choose to implement a deeper version of cloning, in which some of the referenced objects are themselves cloned.

```

/*A method for creating a deep copy of a two-dimensional array of integers*/
public static int[][] deepClone(int[][] original) {
    int[][] backup = new int[original.length][]; // create top-level array of
arrays
    for (int k = 0; k < original.length; k++)
        backup[k] = original[k].clone(); // copy row k
    return backup;
}

```

```

/* Implementation of the SinglyLinkedList.clone method */
public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
    // always use inherited Object.clone() to create the initial copy
    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe
cast
    if (size > 0) { // we need independent chain of nodes
        other.head = new Node<>(head.getElement(), null);
        Node<E> walk = head.getNext(); // walk through remainder of original
list
        Node<E> otherTail = other.head; // remember most recently created
node
        while (walk != null) { // make a new node storing same element
            Node<E> newest = new Node<>(walk.getElement(), null);
            otherTail.setNext(newest); // link previous node to this one
            otherTail = newest;
            walk = walk.getNext();
        }
    }
    return other;
}

```

## Stacks

### Stack: definition

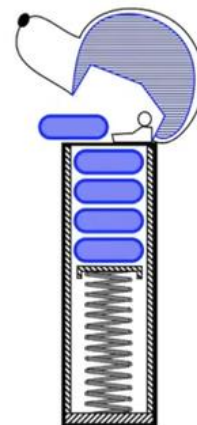
Stack (or pile) of elements.  
We can only access/modify the top of the stack.

#### Last-in, first-out principle (LIFO):

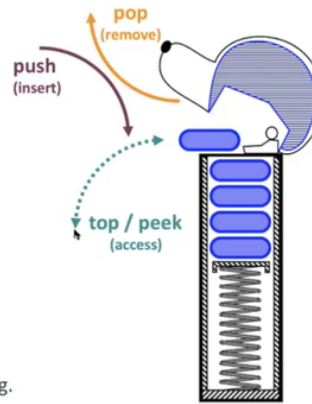
- elements can be inserted at any time;
- only the last inserted element (top of the stack) can be accessed or deleted.

#### Applications:

- Candy dispenser.
- Text editor undo: most recent change undone first.



## Stack: definition



Stack: **last-in, first-out** (LIFO) collection.

### Operations:

- **push**: insert element onto the stack;
- **pop**: remove element at the top of the stack;
- **top/peek**: access top element, without removing.

## Stack: abstraction

### Abstract Data Structure (ADT)

Abstraction of a data structure, cannot be instantiated. It specifies:

- Signatures of operations on the data structure;
- May also contain constants, default or static methods, ...

### Stack ADT

What methods do you expect in a Stack ADT?

## Stack: abstraction

### Abstract Data Structure (ADT)

Abstraction of a data structure, cannot be instantiated. It specifies:

- Signatures of operations on the data structure;
- May also contain constants, default or static methods, ...

### Stack ADT

#### Update methods

`void push(E e)`

adds element *e* to the top of the stack

`E pop()`

removes and returns element at the top of the stack

#### Accessor methods

`E top()`

returns the top element on the stack, without removing it

`int size()`

returns the number of elements in the stack

# Stack ADT: Java interface

Java code 6.1  
public interface Stack<E> { ... }

Generic parameterized type (allows specification of element type at declaration)

```
public interface Stack<E> {
    /**
     * Returns the number of elements in the stack.
     * @return number of elements in the stack
     */
    int size();

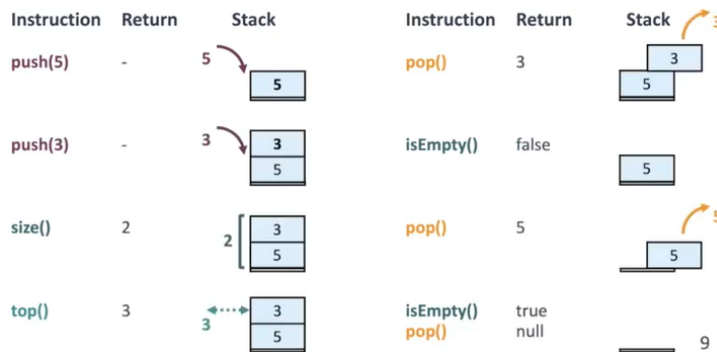
    /**
     * Tests whether the stack is empty.
     * @return <code>true</code> if the stack is empty, <code>false</code> otherwise
     */
    boolean isEmpty();

    /**
     * Inserts an element at the top of the stack.
     * @param e the element to insert
     */
    void push(E e);
    ...
}
```

Methods in an interface are implicitly public.

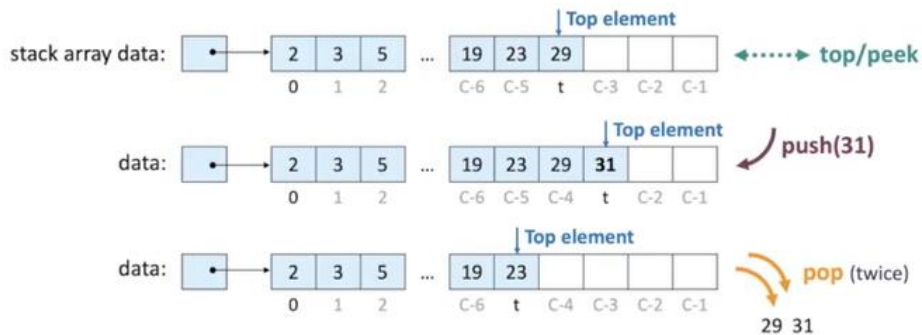
Javadoc-style comments. Keyword examples: @return describes output values @param describes parameters

## Stack: example



Java code 6.2  
public class ArrayStack<E> { ... }

## Stack: array-based

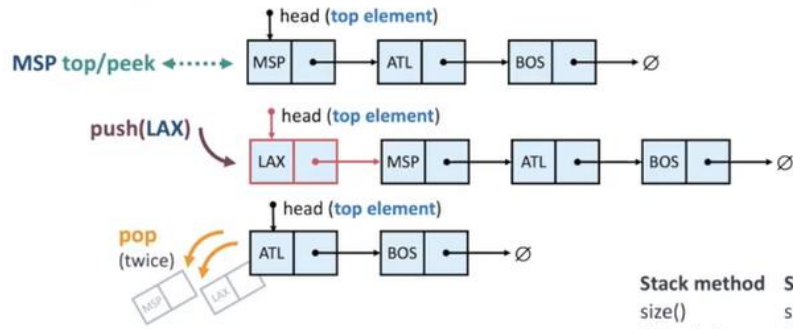


- Stores elements in an array of fixed capacity C.
- Top element: index t (data[t]).
- Stack size: t+1.

Why is the top element at the end of the array?  
Efficient element insertion/deletion! O(1)

## Stack: list-based

Java code 6.4  
public class LinkedStack<E> { ...



- **Top of the stack:** head of the list, for  $O(1)$  insertions and deletions.
- **Implementation:** Reuse SLL methods (*adapter* design pattern).

Stack method	SLL method
size()	size()
isEmpty()	isEmpty()
push(e)	addFirst(e)
pop()	removeFirst()
top()	first()

13

This is called an adapter design pattern

## Stack: complexity

Data structure	Space	Time
Array	$O(C)$	$C$ , array capacity
List	$O(n)$	$n$ , number of elements in the stack
<b>Operations</b>		
Obtain size		size() $O(1)$
Check if empty		isEmpty() $O(1)$
Get top element		top() $O(1)$
Insert element $e$ at the top		push( $e$ ) $O(1)$
Remove and return top element		pop() $O(1)$

**Array:** fixed capacity, wastes space if overdimensioned, costs time if underdimensioned (expansion).  
**List:** grows efficiently.

## Stack: matching symbols

```

1 public static boolean isMatched(String expression) {
2     final String open = "{{" ;
3     final String close = "}}";
4     Stack<Character> buffer = new LinkedStack<>();

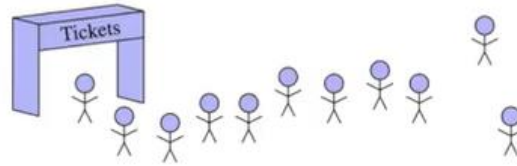
5     for (char c : expression.toCharArray()) {
6         if (open.indexOf(c) != -1)
7             buffer.push(c);
8         else if (close.indexOf(c) != -1) {

9             if (buffer.isEmpty())
10                return false;
11            if (close.indexOf(c) != open.indexOf(buffer.pop()))
12                return false;
13        }
14    }
15    return buffer.isEmpty();
16 }
    
```

**Time:  $O(n)$**   
**Space:  $O(n)$**

<p><b>Space</b></p> <ul style="list-style-type: none"> <li><math>O(1)</math> open <math>O(1)</math></li> <li><math>O(1)</math> close <math>O(1)</math></li> <li><math>O(1)</math> buffer <math>O(1)</math></li> <li><math>O(n)</math> toCharArray() <math>O(n)</math></li> <li><math>O(n)</math> buffer <math>O(n)</math></li> <li><math>O(1)</math> open <math>O(1)</math></li> <li><math>O(1)</math> close <math>O(1)</math></li> </ul>	<p>All operations within for loop are <math>O(1)</math>.</p> <p><math>O(1)</math></p>	<p><b>Time</b></p> <ul style="list-style-type: none"> <li><math>O(1)</math></li> <li><math>O(n)</math> toCharArray takes <math>O(n)</math> time but is called <b>once</b>, result stored in a temporary variable. stack operations <math>O(1)</math> push, pop, isEmpty.</li> <li><math>O(1)</math> indexOf is <math>O(k)</math>, with <math>k</math> the # of chars: since <math>k=3</math>, small constant, doesn't vary with <math>n</math>, so <math>O(1)</math>.</li> <li><math>O(1)</math></li> </ul>
---	---	---

Queues



Collection of objects.

Insertion and deletion follow **first-in, first-out** principle (FIFO):

- Elements can be inserted at any time, at the back of the queue.
- Only the element that has been the longest in the queue can be removed.

**Applications:**

- Handle calls to a call center.
- Handle printing jobs.

## Queue: abstraction

Java code 6.7  
`public interface Queue<E> { ... }`

### Queue ADT

**Update methods**

`void enqueue(E e)` adds element *e* to the tail of the queue  
`E dequeue()` removes & returns the head element (or null)

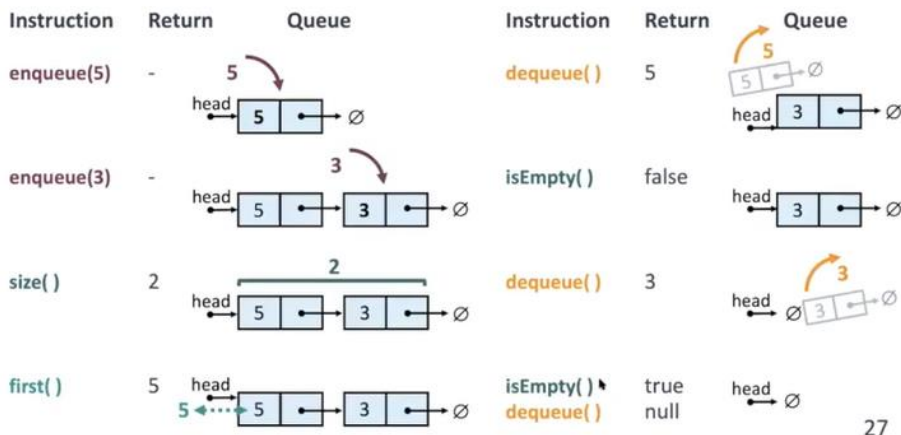
**Accessor methods**

`E first()` returns the head element, without removing it  
`int size()` returns the number of elements in the queue  
`boolean isEmpty()` returns true if the queue is empty, false otherwise

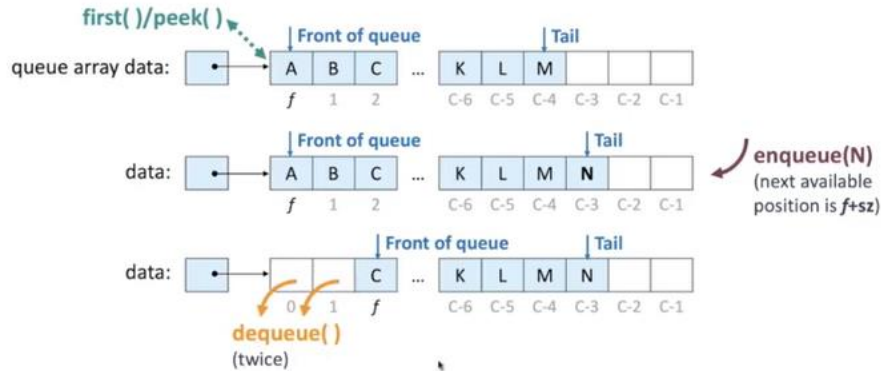
**java.util.Queue methods**

throws Exception	returns special value
<code>add(e)</code>	<code>offer(e)</code>
<code>remove()</code>	<code>poll()</code>
<code>element()</code>	<code>peek()</code>
	<code>size()</code>
	<code>isEmpty()</code>

## Queue: example



## Queue: array-based



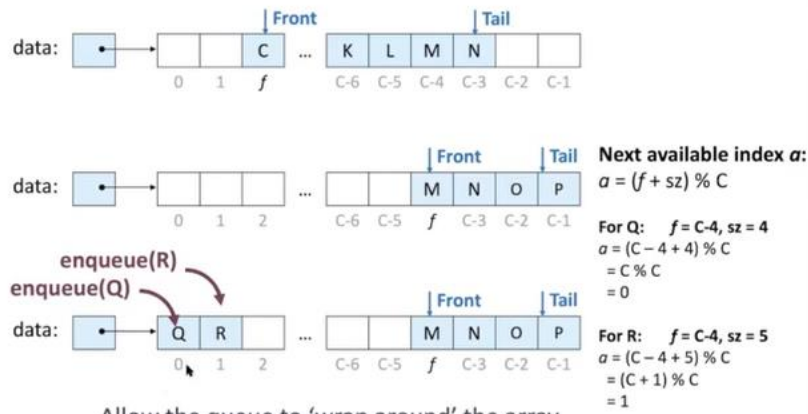
The front of the queue drifts away from 0 with each dequeue operation.

29

It's done like that so adding and enqueueing only takes  $O(1)$  time!

## Queue: array-based

JAVA CODE 6.8  
`public class ArrayQueue<E> { ... }`



Allow the queue to 'wrap around' the array (using the modulo operator %, the remainder of integer division).

32

## Queue: array-based

Time:  $O(1)$   
 Space:  $O(C)$

Java code 6.8  
`public class ArrayQueue<E>`



### Time complexity

Method	big-Oh	Reasoning
size	$O(1)$	stored in variable $sz$ , constant access
isEmpty	$O(1)$	relies on size $sz$ , same reasoning
first	$O(1)$	constant access via stored index $f$
enqueue	$O(1)$	insertion at available index $(f + sz) \% C$
dequeue	$O(1)$	deletion at index $f$ , update $f = (f + 1) \% C$

No shifting of elements in enqueue/dequeue!

### Space complexity

Fields	big-Oh	Reasoning
array data	$O(C)$	fixed capacity $C$ , independent of queue size $sz$
$f$	$O(1)$	primitive type int
$sz$	$O(1)$	primitive type int

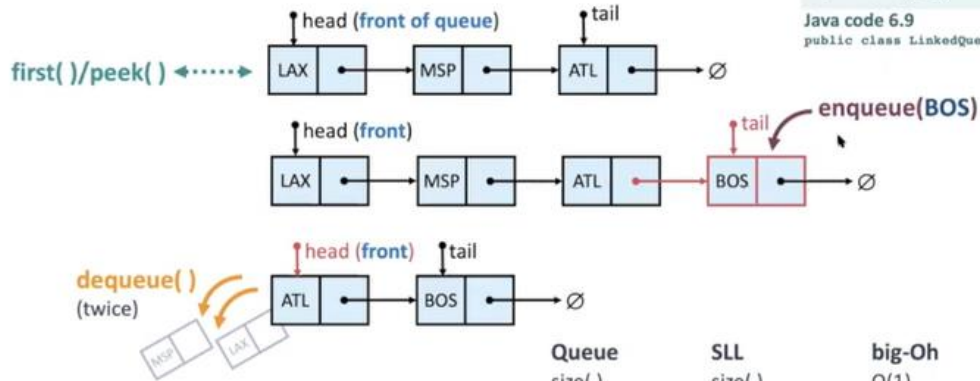
Overall  $O(C)$

33

## Queue: list-based

Time:  $O(1)$   
Space:  $O(C)$

Java code 6.9  
`public class LinkedQueue<E>`



- **Implementation:** Reuse SLL (*adapter* design pattern).
- **Front:** head, since tail deletion is not efficient.
- **Trade-off:** Grows efficiently. More space/time than array.

Queue	SLL	big-Oh
size( )	size( )	$O(1)$
isEmpty( )	isEmpty( )	$O(1)$
enqueue(e)	addLast(e)	$O(1)$
dequeue( )	removeFirst( )	$O(1)$
first( )	first( )	$O(1)$

34

## Queue

### Hot potato game

- n children in a circle passing a hot potato around;
- potato is passed until a bell rings;
- at that point, the child holding the potato:
  - passes the hot potato to the next;
  - leaves the game.
- Game continues with the remaining children, until only one child is left.

What data structure should we use to implement this game efficiently?

Note: We don't know beforehand how many children are joining.

ArrayQueue

LinkedQueue

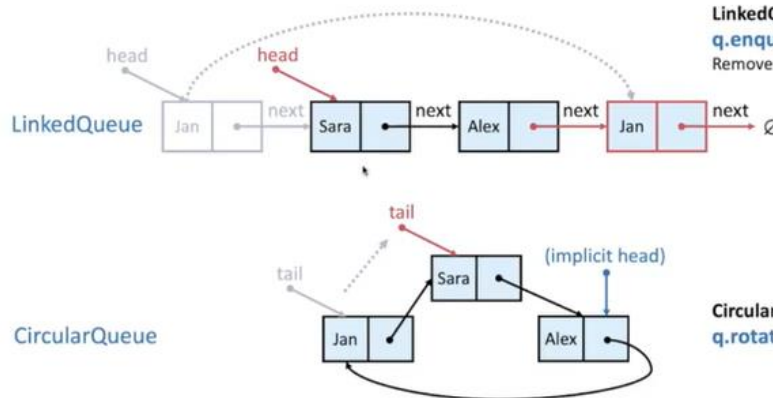
something else?

## Queue: circular

Java code 6.10,6.11  
`public interface CircularQueue<E>`

Passing to next

LinkedQueue  
`q.enqueue(q.dequeue( ))`  
Remove first, insert it at the end.



CircularQueue  
`q.rotate( )`

36

Dequeues

# Deque

Java code 6.12  
`public interface Deque<E>`

**Waiting line at restaurant:**

- first person is called, but finds out the table is not yet available;
- last person gets impatient and leaves.

**To support these use cases, we need efficient insertion and deletion at the front and tail of the queue.**

**Double-Ended Queue (Deque) ADT** (deque is pronounced 'deck')

`public interface Deque<E>`

**Update methods**

`addFirst(e)` insert element *e* at the front  
`addLast(e)` insert element *e* at the tail  
`removeFirst()` remove and return first element  
`removeLast()` remove and return last element

**Accessor methods**

`first()` returns first element without removing  
`last()` returns last element without removing  
`size()` returns number of elements in the deque  
`isEmpty()` true if deque is empty, false otherwise

## Deque: array and list

Array implementation

**Circular array (as in queue)**

Extra concern: when inserting first, decrement *f* in circular fashion, to make sure *f* doesn't become negative:

$$f = (f - 1 + C) \% C$$

List implementation

**Doubly-linked list**

DoublyLinkedList class (code 3.16,3.17) implements all methods of our Deque ADT:

```
public class DoublyLinkedList<E>
implements Deque<E> { ... }
```

What are the space complexities of these data structures?  
 What are the time complexities of their operations?

## Deque: array and list

Array implementation

**Circular array (as in queue)**

Space  $O(C)$  where *C* is array capacity

List implementation

**Doubly-linked list**

Space  $O(n)$  where *n* is number of elements

Deque ADT	Time	java.util.Deque (java.util.ArrayDeque / java.util.LinkedList)	throws exceptions	returns special value
<code>size()</code>	$O(1)$			<code>size()</code>
<code>isEmpty()</code>	$O(1)$			<code>isEmpty()</code>
<code>first()</code>	$O(1)$	<code>getFirst()</code>		<code>peekFirst()</code>
<code>last()</code>	$O(1)$	<code>getLast()</code>		<code>peekLast()</code>
<code>addFirst()</code>	$O(1)$	<code>addFirst()</code>		<code>offerFirst()</code>
<code>addLast()</code>	$O(1)$	<code>addLast()</code>		<code>offerLast()</code>
<code>removeFirst()</code>	$O(1)$	<code>removeFirst()</code>		<code>pollFirst()</code>
<code>removeLast()</code>	$O(1)$	<code>removeLast()</code>		<code>pollLast()</code>



## The Stack and (De)Queues [Chapter 6 Extras]

A stack is **Last In First Out** abstract data type (ADT) that supports the following two update methods:

`push(e)`: Adds element *e* to the top of the stack.

`pop()`: Removes and returns the top element from the stack (or null if the stack is empty).

Additionally, a stack supports the following accessor methods for convenience:

`top()`: Returns the top element of the stack, without removing it (or null if the stack is empty).

`size()`: Returns the number of elements in the stack.

`isEmpty()`: Returns a boolean indicating whether the stack is empty.

By convention, we assume that elements added to the stack can have arbitrary type and that a newly created stack is empty.

Java's Stack class remains only for historic reasons, and its interface is not consistent with most other data structures in the Java library. In fact, the current documentation for the Stack class recommends that it not be used, as LIFO functionality (and more) is provided by a more general data structure known as a double-ended queue. Therefore we have our own stack interface, which we will implement for our own stack-based classes.

```
public interface Stack<E> {
    int size();

    boolean isEmpty();

    void push(E e);

    E top();

    E pop();
}
```

### Array based Stack

Pros:

1. Very efficient implementation when the user knows how much memory he needs.
2. Returning the (popped) cell to a null reference is not mandatory but we do it to assist Java's garbage collection mechanism, which searches memory for objects that are no longer actively and wipes'em out.

Cons:

1. Otherwise, there could be a big waste of data
2. or lead to an IllegalStateException if we run out of memory

```

public class ArrayStack<E> implements Stack<E> {
    public static final int CAPACITY = 1000; // default array capacity
    private E[] data; // generic array used for storage
    private int t = -1; // index of the top element in stack

    public ArrayStack() {
        this(CAPACITY);
    } // constructs stack with default capacity

    public ArrayStack(int capacity) { // constructs stack with given capacity
        data = (E[]) new Object[capacity]; // safe cast; compiler may give
warning
    }

    public int size() {
        return (t + 1);
    }

    public boolean isEmpty() {
        return (t == -1);
    }

    public void push(E e) throws IllegalStateException {
        if (size() == data.length) throw new IllegalStateException("Stack is
full");
        data[++t] = e; // increment t before storing new item
    }

    public E top() {
        if (isEmpty()) return null;
        return data[t];
    }

    public E pop() {
        if (isEmpty()) return null;
        E answer = data[t];
        data[t] = null; // dereference to help garbage collection
        t--;
        return answer;
    }
}

```

### Singly Link based Stack

Unlike our array-based implementation, the linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit. With the top of the stack stored at the front of the list, all methods execute in constant time.

### Adapter Patern

The adapter design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

Implementation of a Stack using a SinglyLinkedList as storage

```

public class LinkedStack<E> implements Stack<E> {
    private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty
list

    public LinkedStack() {
        } // new stack relies on the initially empty list

    public int size() {
        return list.size();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public void push(E element) {
        list.addFirst(element);
    }

    public E top() {
        return list.first();
    }

    public E pop() {
        return list.removeFirst();
    }
}

```

## Matching Parentheses

```

/**
 * Tests if delimiters in the given expression are properly matched.
 */
public static boolean isMatched(String expression) {
    final String opening = "{[("; // opening delimiters
    final String closing = ")]}"; // respective closing delimiters
    Stack<Character> buffer = new LinkedStack<>();
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1) // this is a left delimiter
            buffer.push(c);
        else if (closing.indexOf(c) != -1) { // this is a right delimiter
            if (buffer.isEmpty()) // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
                return false; // mismatched delimiter
        }
    }
    return buffer.isEmpty(); // were all opening delimiters matched?
}

```

## Matching HTML tags

```

/**
 * Tests if every opening tag has a matching closing tag in HTML string.
 */
public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new LinkedStack<>();
    int j = html.indexOf('<'); // find first '<' character (if any)
    while (j != -1) {
        int k = html.indexOf('>', j + 1); // find next '>' character
        if (k == -1)
            return false; // invalid tag
        String tag = html.substring(j + 1, k); // strip away < >
        if (!tag.startsWith("/")) // this is an opening tag
            buffer.push(tag);
        else { // this is a closing tag
            if (buffer.isEmpty())
                return false; // no tag to match
            if (!tag.substring(1).equals(buffer.pop()))
                return false; // mismatched tag
        }
        j = html.indexOf('<', k + 1); // find next '<' character (if any)
    }
    return buffer.isEmpty(); // were all opening tags matched?
}

```

## Queues

First in First Out data structure. Elements enter a queue at the back and are removed from the front. The queue abstract data type (ADT) supports the following two update methods:

**enqueue(*e*):** Adds element *e* to the back of queue.

**dequeue():** Removes and returns the first element from the queue (or null if the queue is empty).

The queue ADT also includes the following accessor methods (with first being analogous to the stack's top method):

**first():** Returns the first element of the queue, without removing it (or null if the queue is empty).

**size():** Returns the number of elements in the queue.

**isEmpty():** Returns a boolean indicating whether the queue is empty.

By convention, we assume that elements added to the queue can have arbitrary type and that a newly created queue is empty.

```

public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E e);
    E first();
    E dequeue();
}

```

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue( <i>e</i> )	add( <i>e</i> )	offer( <i>e</i> )
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

## Implementing a Queue with a Singly Linked List

```

/**
 * The type Linked queue.
 *
 * @param <E> the type parameter
 */
public class LinkedQueue<E> implements Queue<E> {
    private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty
list

    /**
     * Instantiates a new Linked queue.
     */
    public LinkedQueue() {
    } // new queue relies on the initially empty list
    public int size() {
        return list.size();
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public void enqueue(E element) {
        list.addLast(element);
    }
    public E first() {
        return list.first();
    }
    public E dequeue() {
        return list.removeFirst();
    }
}

```

## A Circular Queue

```

/**
 * Rotates the front element
of the queue to the back of
the queue. This does nothing
if the queue is empty.
 */
public interface
CircularQueue<E> extends
Queue<E> {
    void rotate( );
}

```

## Double-Ended (Deck) Queues

Such a structure is called a double-ended queue, or deque, which is usually pronounced “deck” to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation “D.Q.”

## 6.3.1 The Deque Abstract Data Type

The deque abstract data type is richer than both the stack and the queue ADTs. To provide a symmetrical abstraction, the deque ADT is defined to support the following update methods:

- addFirst(*e*):** Insert a new element *e* at the front of the deque.
- addLast(*e*):** Insert a new element *e* at the back of the deque.
- removeFirst():** Remove and return the first element of the deque (or null if the deque is empty).
- removeLast():** Remove and return the last element of the deque (or null if the deque is empty).

Additionally, the deque ADT will include the following accessors:

- first():** Returns the first element of the deque, without removing it (or null if the deque is empty).
- last():** Returns the last element of the deque, without removing it (or null if the deque is empty).
- size():** Returns the number of elements in the deque.
- isEmpty():** Returns a boolean indicating whether the deque is empty.

A Java interface, Deque, describing the double-ended queue ADT. Note the use of the generic parameterized type, E, allowing a deque to contain elements of any specified class.

```
public interface Deque<E> {
    int size();

    boolean isEmpty();

    E first();
    E last();

    void addFirst(E e);
    void addLast(E e);

    E removeFirst();
    E removeLast();
}
```

Our Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first()	getFirst()	peekFirst()
last()	getLast()	peekLast()
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst()	removeFirst()	pollFirst()
removeLast()	removeLast()	pollLast()
size()	size()	
isEmpty()	isEmpty()	

Tail recursion example [ $O(n)$  time,  $O(1)$  memory vs  $O(2^n)$  time and  $O(n)$  memory]

```
fac n = go n 1
go 1 a = a
go n a = go (n-1) (a * n)
```

<https://www.youtube.com/watch?v=JtPhF8MshA>

This saves time (you go only one way) and space (you don't need to create an extremely large expression as you will keep the returning value in just the accumulator).

But it's important to note that the language/compiler will need to support tail call optimisation, otherwise it's not helping as much: For example with the accumulator, a "naive" language implementation would still keep all the stack frames around until the end, and return the just-received value back up the stack. Only if the language supports TCO will it recognize the tail call and replace (overwrite) the current stack frame for the next call - which is where the optimisation helps to reduce memory usage.

You use the "accumulator", that is an extra parameter that you use in the recursive function signature, to update the returning value as you go deep into the recursion, and once you hit the bottom of the recursion, you can directly return the accumulator (instead of bouncing back like in the classic factorial definition).

```
fib n = go n (0,1)
go 0 (a,b) = a
go 1 (a,b) = b
go n (a,b) = go (n-1) (b,a+b)

(2,3) ↦ (3,5)
```

```
fac 4 = 4 * fac 3
      = 4 * (3 * fac 2)
      = 4 * (3 * (2 * fac 1))
      = 4 * (3 * (2 * 1))
      = 4 * (3 * 2)
      = 4 * 6
      = 24
```

✓

```
fac 4 = go 4 1
      = go (4-1) (1*4)
      = go 3 4
      = go 2 12
      = go 1 24
      = 24
```

```
fib 4 = go 4 (0,1)
      = go 3 (1,1)
      = go 2 (1,2)
      = go 1 (2,3)
      = 3
```

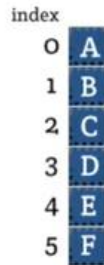
## Chapter 7. List Abstractions

### List Abstractions

Chapter 7.1 List ADT  
 Chapter 7.2 Array-Based Lists  
 Dynamic Arrays

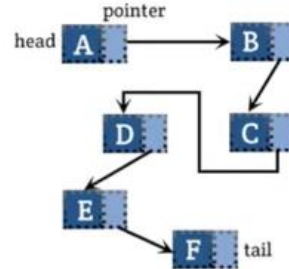
**Efficient access!**  
 (positional access)

#### Arrays



**Efficient expansion!**

#### Lists



List Abstractions aim to get both efficient access and expansion

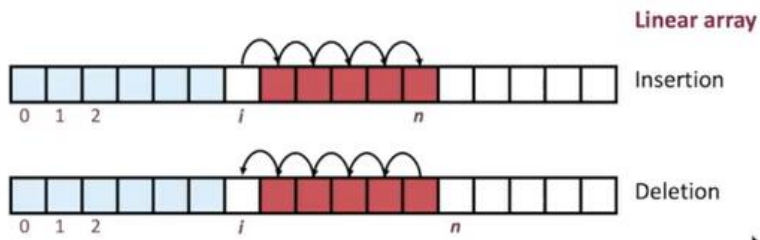
### List ADT

```
public interface List<E> {
    int size();
    boolean isEmpty();
    E get(int i) throws IndexOutOfBoundsException;
    E set(int i, E e) throws IndexOutOfBoundsException;
    void add(int i, E e) throws IndexOutOfBoundsException;
    E remove(int i) throws IndexOutOfBoundsException;
}
```

#### List ADT operations

size()	returns the number of elements in the list	
isEmpty()	return true if the list is empty, false otherwise	
get(i)	returns the element at index <i>i</i> or an error if <i>i</i> is not in {0, ..., size()-1}	} error if <i>i</i> is not in range {0, ..., size()-1} {0, ..., size()} for add
set(i, e)	replaces the element at index <i>i</i> with <i>e</i> and returns the replaced element	
add(i, e)	adds <i>e</i> at index <i>i</i> , moving subsequent elements one index forward	
remove(i)	removes and returns the element at index <i>i</i> , moving subsequent elements an index backward	

### List: array-based



Lists can be implemented using arrays:

- **Linear array:** fixed-capacity array,  $k^{\text{th}}$  element is always stored at index  $i = k-1$ .
- **Circular array:** fixed-capacity array, list can wrap around the array (see ArrayQueue, section 6.2).

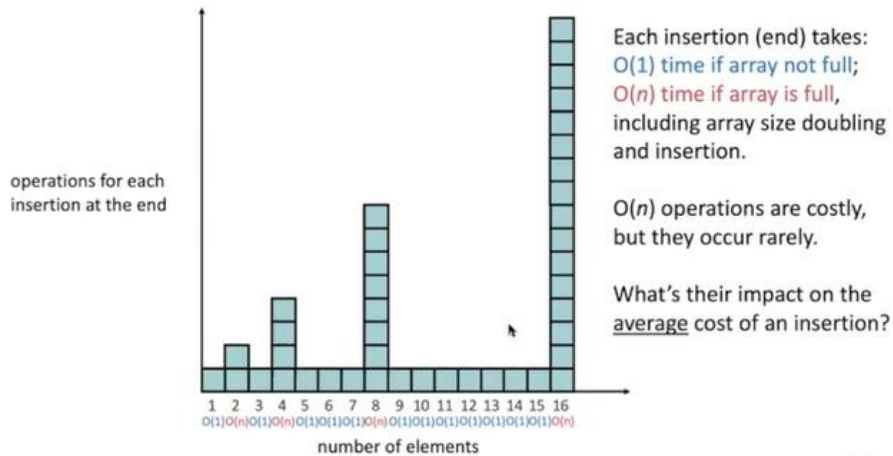
## List: array-based

Array-based list operations		Time	Efficient operations per implementation		
Obtain size	<code>size()</code>	$O(1)$	<b>Linear array</b>		
Check if it's empty	<code>isEmpty()</code>	$O(1)$	<code>add(n, e)</code>	add last	$O(1)$
Get element at index $i$	<code>get(i)</code>	$O(1)$	<code>remove(n-1)</code>	remove last	$O(1)$
Set element $e$ at index $i$	<code>set(i, e)</code>	$O(1)$	<b>Circular array</b>		
Add element $e$ at index $i$	<code>add(i, e)</code>	$O(n)$	<code>add(0, e)</code>	add first	$O(1)$
Remove element at index $i$	<code>remove(i)</code>	$O(n)$	<code>add(n, e)</code>	add last	$O(1)$
			<code>remove(0)</code>	remove first	$O(1)$
			<code>remove(n-1)</code>	remove last	$O(1)$

Many applications add items last to the collection, mainly for storage/access, without frequent changes. Arrays are great for this! Major drawback of arrays is fixed capacity. Circular arrays provide more space (wrap around), but linear arrays are easier to grow. **But what about the cost of expanding?**

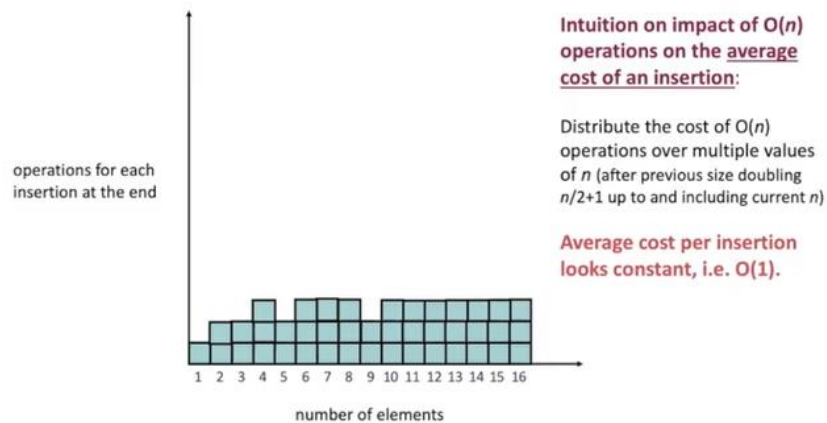
48

## Dynamic array: amortized analysis



50

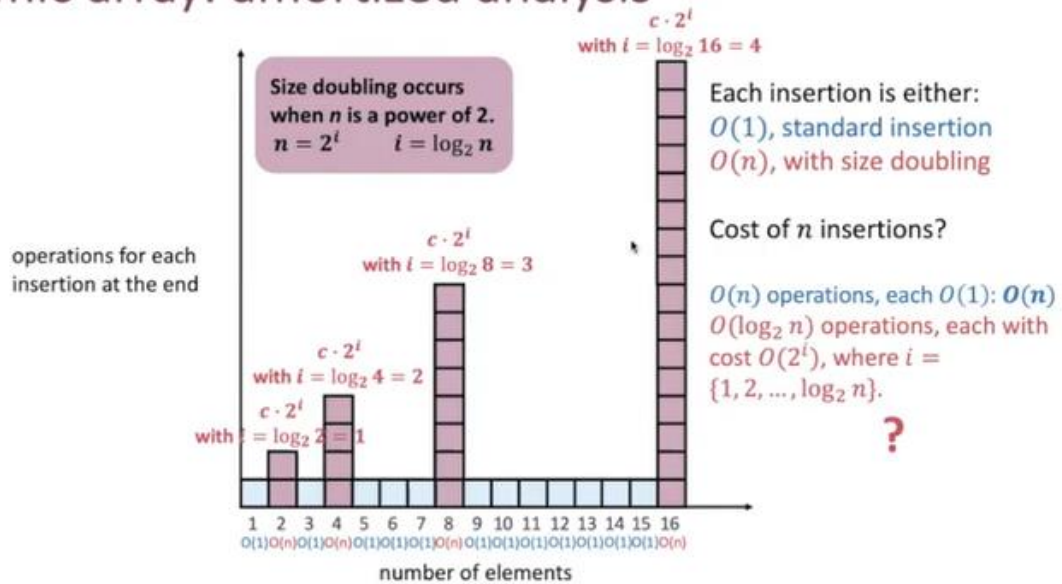
## Dynamic array: amortized analysis



54



## Dynamic array: amortized analysis



57

## Dynamic array: amortized analysis

Overall cost of  $n$  insertions?

$O(n)$  operations costing  $O(1)$  each.  $O(n)$

$O(\log_2 n)$  operations costing  $O(2^i)$  each, with  $i = \{1, 2, \dots, \log_2 n\}$  ?

$$\begin{aligned} \sum_{i=1}^{\log_2 n} 2^i &= \frac{2 \times (1 - 2^{\log_2 n})}{1 - 2} && \text{sum of first } m \text{ terms} \\ &= 2 \times (2^{\log_2 n} - 1) && \text{arithmetic} \\ &= 2 \times (n - 1) && \text{arithmetic} \\ &= 2n - 2 && \mathbf{O(n)} \end{aligned}$$

Sum of first  $m$  terms of a geometric progression:

$$S_m = \frac{a_1(1 - r^m)}{1 - r}, r \neq 1$$

where:

$m$  is the number of terms  
 $a_1$  is the first term  
 $r$  is the common ratio

Here:

$$\begin{aligned} m &= \log_2 n \\ a_1 &= 2 \\ r &= 2 \end{aligned}$$

Cost of  $n$  insertions is  $O(n)$ . **Amortized** (or average) cost per insertion is  $O(1)$ .

61

Amortized  $O(1)$  time complexity per insertion at the end of an array holds:

- for growing factors other than 2 (Java uses 3/2),
- as long as the increase in size is proportional to the size of the array (geometric progression).

With that said, Java defines a general interface, `java.util.List`, that includes the following index-based methods (and more):

- `size()`: Returns the number of elements in the list.
- `isEmpty()`: Returns a boolean indicating whether the list is empty.
- `get(i)`: Returns the element of the list having index *i*; an error condition occurs if *i* is not in range  $[0, \text{size}() - 1]$ .
- `set(i, e)`: Replaces the element at index *i* with *e*, and returns the old element that was replaced; an error condition occurs if *i* is not in range  $[0, \text{size}() - 1]$ .
- `add(i, e)`: Inserts a new element *e* into the list so that it has index *i*, moving all subsequent elements one index later in the list; an error condition occurs if *i* is not in range  $[0, \text{size}())$ .
- `remove(i)`: Removes and returns the element at index *i*, moving all subsequent elements one index earlier in the list; an error condition occurs if *i* is not in range  $[0, \text{size}() - 1]$ .

```

1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3      /** Returns the number of elements in this list. */
4      int size();
5
6      /** Returns whether the list is empty. */
7      boolean isEmpty();
8
9      /** Returns (but does not remove) the element at index i. */
10     E get(int i) throws IndexOutOfBoundsException;
11
12     /** Replaces the element at index i with e, and returns the replaced element. */
13     E set(int i, E e) throws IndexOutOfBoundsException;
14
15     /** Inserts element e to be at index i, shifting all subsequent elements later. */
16     void add(int i, E e) throws IndexOutOfBoundsException;
17
18     /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19     E remove(int i) throws IndexOutOfBoundsException;
20 }

```

**Code Fragment 7.1:** A simple version of the List interface.

```

1  public class ArrayList<E> implements List<E> {
2      // instance variables
3      public static final int CAPACITY=16;    // default array capacity
4      private E[] data;                       // generic array used for storage
5      private int size = 0;                   // current number of elements
6      // constructors
7      public ArrayList() { this(CAPACITY); } // constructs list with default capacity
8      public ArrayList(int capacity) {       // constructs list with given capacity
9          data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
10     }

```

**Code Fragment 7.2:** An implementation of a simple ArrayList class with bounded capacity. (Continues in Code Fragment 7.3.)

```

11 // public methods
12 /** Returns the number of elements in the array list. */
13 public int size() { return size; }
14 /** Returns whether the array list is empty. */
15 public boolean isEmpty() { return size == 0; }
16 /** Returns (but does not remove) the element at index i. */
17 public E get(int i) throws IndexOutOfBoundsException {
18     checkIndex(i, size);
19     return data[i];
20 }
21 /** Replaces the element at index i with e, and returns the replaced element. */
22 public E set(int i, E e) throws IndexOutOfBoundsException {
23     checkIndex(i, size);
24     E temp = data[i];
25     data[i] = e;
26     return temp;
27 }
28 /** Inserts element e to be at index i, shifting all subsequent elements later. */
29 public void add(int i, E e) throws IndexOutOfBoundsException,
30                 IllegalStateException {
31     checkIndex(i, size + 1);
32     if (size == data.length) // not enough capacity
33         throw new IllegalStateException("Array is full");
34     for (int k=size-1; k >= i; k--) // start by shifting rightmost
35         data[k+1] = data[k];
36     data[i] = e; // ready to place the new element
37     size++;
38 }
39 /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40 public E remove(int i) throws IndexOutOfBoundsException {
41     checkIndex(i, size);
42     E temp = data[i];
43     for (int k=i; k < size-1; k++) // shift elements to fill hole
44         data[k] = data[k+1];
45     data[size-1] = null; // help garbage collection
46     size--;
47     return temp;
48 }
49 // utility method
50 /** Checks whether the given index is in the range [0, n-1]. */
51 protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52     if (i < 0 || i >= n)
53         throw new IndexOutOfBoundsException("Illegal index: " + i);
54 }
55 }

```

**Code Fragment 7.3:** An implementation of a simple ArrayList class with bounded capacity. (Continued from Code Fragment 7.2.)

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get( $i$ )	$O(1)$
set( $i, e$ )	$O(1)$
add( $i, e$ )	$O(n)$
remove( $i$ )	$O(n)$

**Table 7.1:** Performance of an array list with  $n$  elements realized by a fixed-capacity array.

### Dynamic Array

The ArrayList implementation in Code Fragments 7.2 and 7.3 (as well as those for a stack, queue, and deque from Chapter 6) has a serious limitation; it requires that a fixed maximum capacity be declared, throwing an exception if attempting to add an element once full. This is a major weakness, because if a user is unsure of the maximum size that will be reached for a collection, there is risk that either too

large of an array will be requested, causing an inefficient waste of memory, or that too small of an array will be requested, causing a fatal error when exhausting that capacity.

Java's ArrayList class provides a more robust abstraction, allowing a user to add elements to the list, with no apparent limit on the overall capacity. To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a dynamic array.

```

/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
    E[] temp = (E[]) new Object[capacity]; // safe cast; compiler may give warning
    for (int k=0; k < size; k++)
        temp[k] = data[k];
    data = temp; // start using the new array
}

```

**Code Fragment 7.4:** An implementation of the ArrayList.resize method.

```

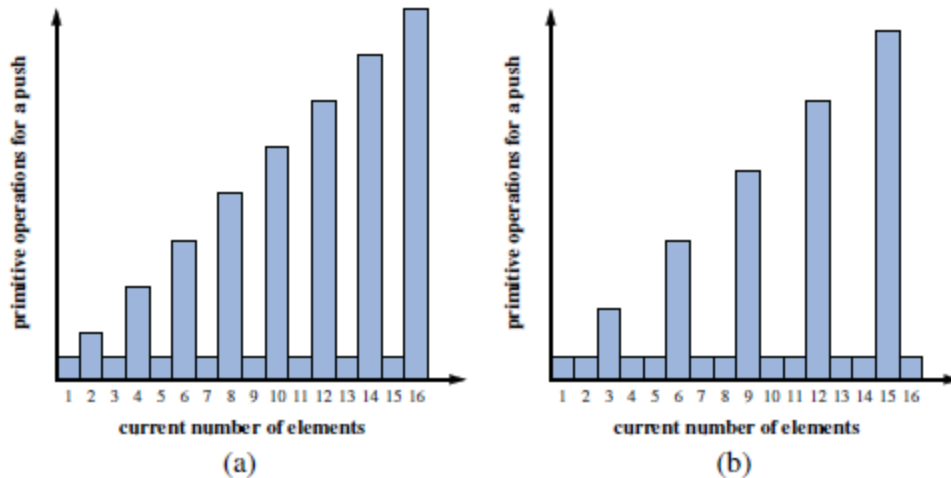
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size + 1);
    if (size == data.length) // not enough capacity
        resize(2 * data.length); // so double the current capacity
    // rest of method unchanged...
}

```

**Code Fragment 7.5:** A revision to the ArrayList.add method, originally from Code Fragment 7.3, which calls the resize method of Code Fragment 7.4 when more capacity is needed.

## Beware of Arithmetic Progression

To avoid reserving too much space at once, it might be tempting to implement a dynamic array with a strategy in which a constant number of additional cells are reserved each time an array is resized. Unfortunately, the overall performance of such a strategy is significantly worse. At an extreme, an increase of only one cell causes each push operation to resize the array, leading to a familiar  $1 + 2 + 3 + \dots + n$  summation and  $\Omega(n^2)$  overall cost. Using increases of 2 or 3 at a time is slightly better, as portrayed in Figure 7.4, but the overall cost remains quadratic.



**Justification:** Let  $c > 0$  represent the fixed increment in capacity that is used for each resize event. During the series of  $n$  push operations, time will have been spent initializing arrays of size  $c, 2c, 3c, \dots, mc$  for  $m = \lceil n/c \rceil$ , and therefore, the overall time is proportional to  $c + 2c + 3c + \dots + mc$ . By Proposition 4.3, this sum is

$$\sum_{i=1}^m ci = c \cdot \sum_{i=1}^m i = c \frac{m(m+1)}{2} \geq c \frac{\frac{n}{c}(\frac{n}{c} + 1)}{2} \geq \frac{1}{2c} \cdot n^2.$$

Therefore, performing the  $n$  push operations takes  $\Omega(n^2)$  time. ■

### StringBuilder vs concatenation

The `StringBuilder` class represents a mutable string by storing characters in a dynamic array. It guarantees that a series of `append` operations resulting in a string of length  $n$  execute in a combined time of  $O(n)$ . (Insertions at positions other than the end of a string builder do not carry this guarantee, just as they do not for an `ArrayList`).

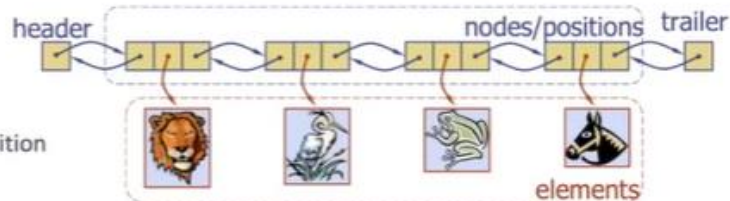
## Week 3. Position-based lists, iterators, trees, and priority queues

### Positional List and iterators

# List: position-based

#### Position-based list

Sequence of positions, each with 1 element. Traversal in both directions. Elements accessed via positions, no direct access to nodes.



#### Position ADT

`getElement()` returns element stored at position

#### Positional List ADT

##### Accessor methods

`first()` returns position of first element (**null** if empty)  
`last()` returns position of last element (**null** if empty)  
`before(p)` returns position just before *p* (**null** if *p* is first)  
`after(p)` returns position just after *p* (**null** if *p* is last)  
`isEmpty()` returns true if list is empty, false otherwise  
`size()` returns the number of elements in list

##### Update methods

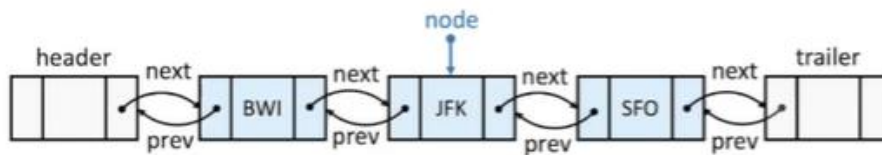
`addFirst(e)` inserts *e* at the front, returns position  
`addLast(e)` inserts *e* at the back, returns position  
`addBefore(p, e)` inserts *e* before *p*  
`addAfter(p, e)` inserts *e* after *p*  
`set(p, e)` replaces element at position *p* with *e*  
`remove(p)` removes and returns element at *p*

No direct access to the nodes in the list, but need to traverse the list (in both directions possible).

# List: position-based

The most natural way to implement a positional list is using a doubly-linked list (DLL).

Should we use references to nodes as positions? Not recommended.

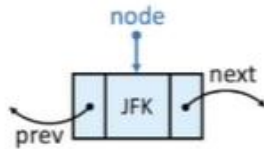


If we give direct access to a node, user has access to the entire list. Allows **changes to list** using external methods. To prevent this, public DLL methods receive or return elements (not nodes).

**Here we use position: reference to a node that prevents access to the rest of the list.**

## List: position-based

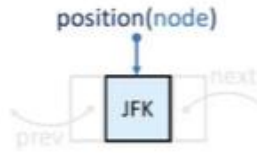
Define a position (reference to a node) that does not allow access to the rest of the list.



```
public class DoublyLinkedList<E> {
    private static class Node<E> { ... }
}
```

In Java we can define an abstraction of a node using an interface.

**Position interface allows access to element only!**



```
public interface Position<E> {
    public E getElement()
        throws IllegalStateException;
}
public class LinkedPositionalList<E> {
    private static class Node<E>
        implements Position<E> { ... }
}
```

## Iterator

Abstracts the process of scanning through a sequence, one element at a time.

Java.util.Iterator

hasNext() returns true if there's at least one additional element in the sequence  
 next() returns the next element in the sequence (or NoSuchElementException)  
 remove() optional method, removes the element returned by the most recent call to next()

If `iter` is an instance of type `Iterator<String>`:

```
while(iter.hasNext()) {
    String value = iter.next();
    System.out.println(value);
}
```

**Provides a single pass through a collection. No way to reset the iterator.**

Some iterators have `previous()` method, which can be used to go back. But going all the way back is  $O(n)$ . To reset, it's better to just create new iterator (which is  $O(1)$  for iterator operating on the original data structure).

# Iterable interface

To be implemented by data structures that allow iterations through them.

## Java Iterable interface

`iterator()` returns an iterator of the elements in the collection

**Example:** Java class `ArrayList` implements `Iterable`, it is not itself an `Iterator`.

### Supports for-each loop:

```
for(String student : csel305students) { // syntax does not allow remove
    System.out.println(student);
}
```

### Is shorthand for:

```
Iterator<String> iter = csel305students.iterator();
while(iter.hasNext()) {
    String student = iter.next();
    System.out.println(student);
}
```

# Iterator: implementations

## Snapshot iterator

- Maintains own private copy of the collection, constructed at creation.
- Unaffected by changes to the primary collection.
- Requires  $O(n)$  space and  $O(n)$  time upon construction (copy and keep collection of  $n$  elements).

## Lazy iterator

- Does not make a copy, directly traverses the primary data structure.
- Affected by changes to the primary collection.
- Requires  $O(1)$  space and  $O(1)$  construction time.
- 'fail-fast' behavior invalidates an iterator if its underlying collection is modified unexpectedly.

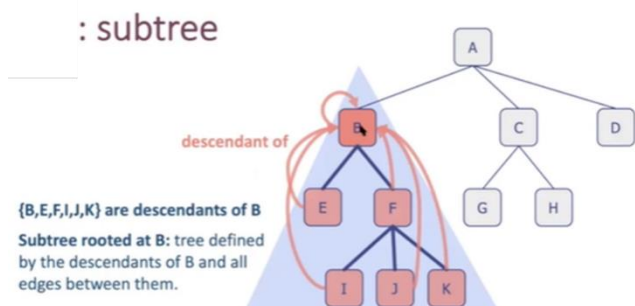
## Trees

### Tree

Abstract model of a hierarchical structure.

Examples:

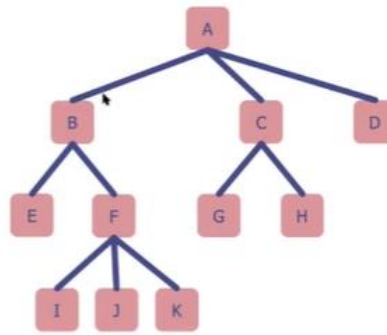
- Tree of languages, tree of life (evolution)
- Inheritance between user-defined classes in Java



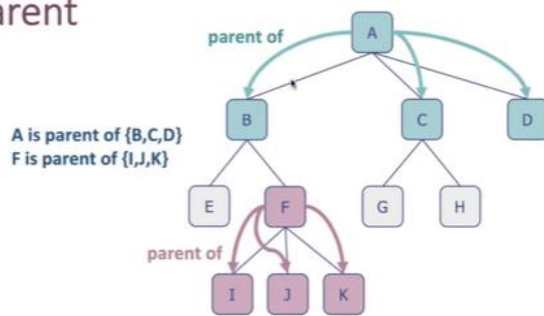


## Tree terminology

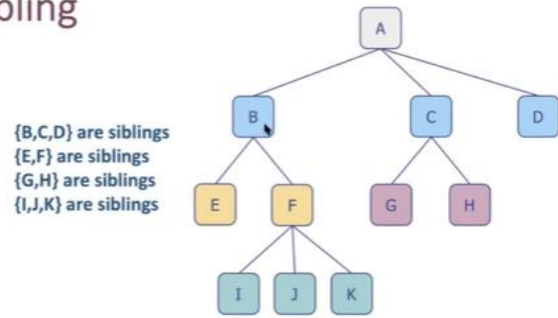
A tree comprises **nodes** and **edges**.  
 Every edge denotes a hierarchical, parent-child relation between two nodes.



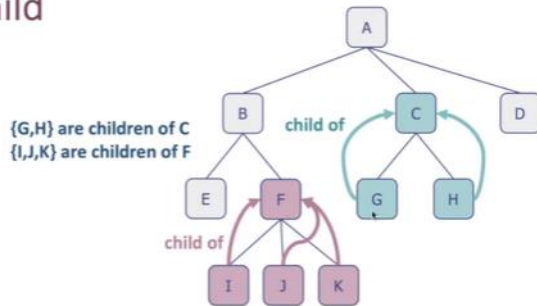
: parent



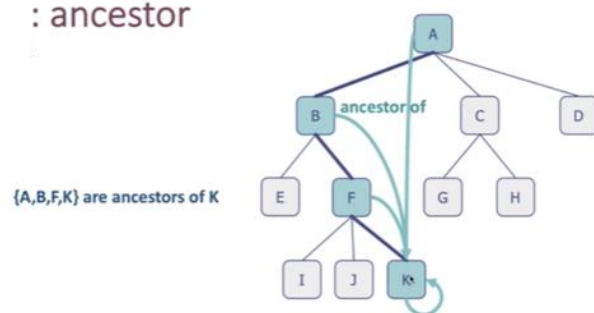
: sibling



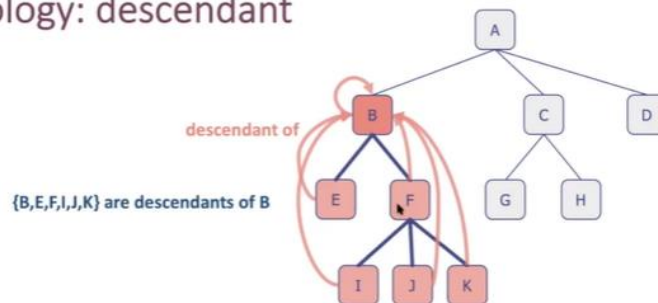
: child



: ancestor



## Tree terminology: descendant

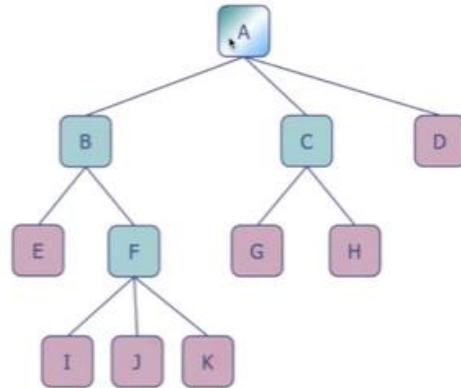


Relationships between nodes

- Parent of  $x$ : node  $y$  at the other end of the unique upward edge from  $x$  (B is the parent of E).
- Child of  $x$ : node  $y$  at lower end of an edge from  $x$  (K is a child of F)
- Sibling of  $x$ : any node  $y$  other than  $x$  with same parent of  $x$  (G is sibling of H)
- Ancestor of  $x$ : any node  $y$  in the unique upward path from  $x$  to the root, including  $x$  (ancestors of K: A,B,F,K)
- **Descendant of  $x$** : any node  $y$  in a downward path from  $x$  to the leaves, including  $x$  (descendants of B: B,E,F,I,J,K)

## Tree terminology: root, internal/external nodes

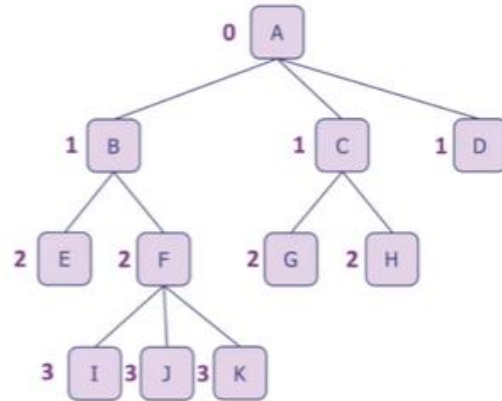
- **Root:** node without parent (A)
- **Internal nodes:** nodes with at least one child (B,C,F)
- **External nodes (leaves):** without children (D,E,G,H,I,J,K)



## Node depth

Depth of node  $x$ : number of ancestors of  $x$  other than  $x$  itself

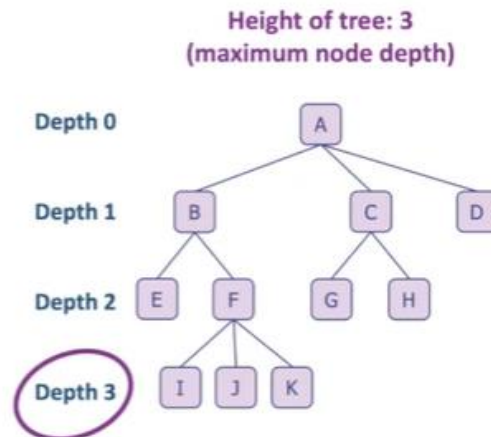
```
public int depth(Position<E> p) {
    if (isRoot(p))
        return 0;
    else
        return 1 + depth(parent(p));
}
```



## Tree height

Depth of node  $x$ : number of ancestors of  $x$  other than  $x$  itself

**Height of tree:**  
maximum depth of all of its nodes, if non-empty tree  
or zero, if tree is empty



# Tree height

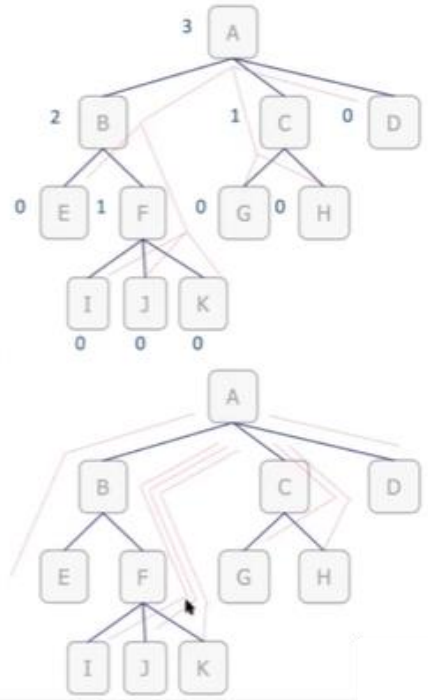
Which of these methods is most efficient? Why?

```
public int height(Position<E> p) {
    int h = 0; // base case (p is leaf)
    for (Position<E> c : children(p))
        h = Math.max(h, 1 + height(c));
    return h;
}
```

`children(p)` visits every node in subtree of  $p$  once:  $O(n)$   
`height(c)` called recursively, only once per node from top to bottom:  $O(n)$

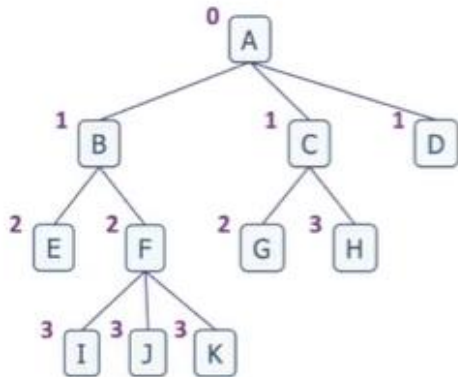
```
public int height() {
    int h = 0;
    for (Position<E> p : positions())
        if (isExternal(p))
            h = Math.max(h, depth(p));
    return h;
}
```

`positions()` is executed once; visits each node once:  $O(n)$   
`depth(p)` called per leaf, each time visits all nodes in path up to root; cost proportional to sum of depths of all leaves; worst-case  $O(n^2)$

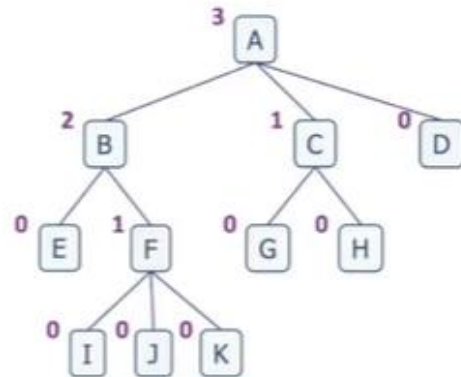


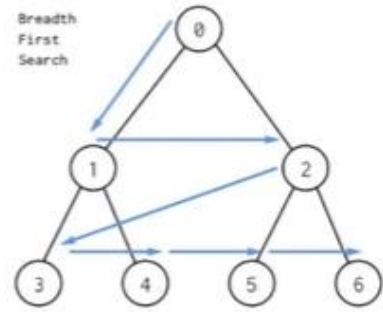
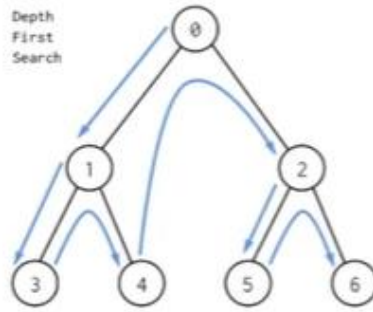
# Depth vs. height

Depth



Height





<http://mishadoff.com/blog/dfs-vs-bfs-in-binary-tree-ways/>

# Tree traversals

Systematic ways of visiting all nodes in a tree.

- Depth-first: pre-order
- Depth-first: post-order
- Depth-first: in-order (only for binary trees)
- Breadth-first

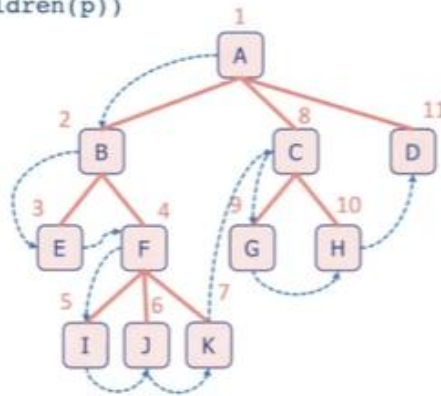
## Pre-order (depth-first)

Node is visited before its children.

```
public void preorder(Position<E> p) {
    visit(p); // visit performs some operation at the node, e.g. print its element
    for (Position<E> c : children(p))
        preorder(c);
}
```

### Legend

- A before visit (unvisited node)
- A during visit
- A after visit (visited node)
- unexplored edge
- explored (downward, preorder call)
- explored (upward, return from call)
- 1 order of visit to node
- points to next visited node



Visited nodes:  
A, B, E, F, I, J, K, C, G, H, D

*return to preorder(A)*  
*return from preorder(A)*

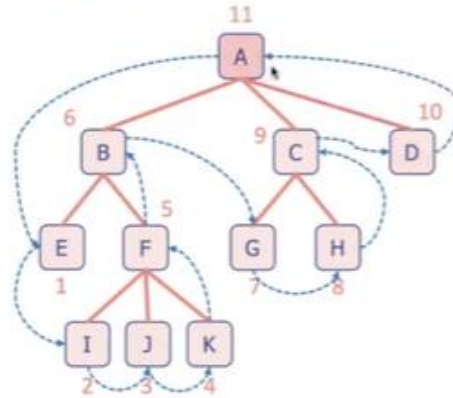
# Post-order (depth-first)

Node is visited after its children.

```
public void postorder(Position<E> p) {
    for (Position<E> c : children(p))
        postorder(c);
    visit(p);
}
```

**Legend**

- A before visit (unvisited node)
- A during visit
- A after visit (visited node)
- unexplored edge
- explored (downward, postorder call)
- explored (upward, return from call)
- 1 order of visit to node
- points to next visited node



Visited nodes:  
E, I, J, K, F, B, G, H, C, D, A

*return to postorder(A)*  
visit(A)

# Breadth-first

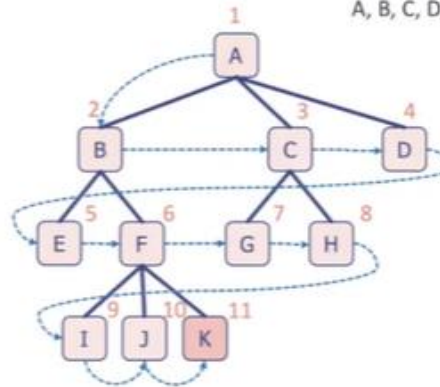
Visits nodes per level (depth).

```
public void breadthfirst(Position<E> p) {
    Queue<Position<E>> q = new Queue();
    q.enqueue(p);
    while (!q.isEmpty()) {
        Position<E> p = q.dequeue();
        visit(p);
        for (Position<E> c : children(p))
            q.enqueue(c);
    }
}
```

**Legend**

- A before visit (unvisited node)
- A during visit
- A after visit (visited node)
- unexplored edge
- explored (downward, enqueue call)
- 1 order of visit to node
- points to next visited node

q.dequeue(K)  
q.visit(K)



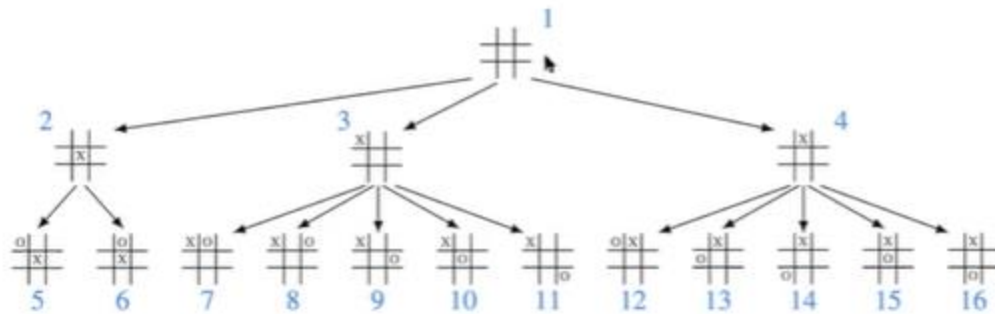
Queue:

-  
Visited nodes:  
A, B, C, D, E, F, G, H, I, J, K

# Breadth-first traversal: applications

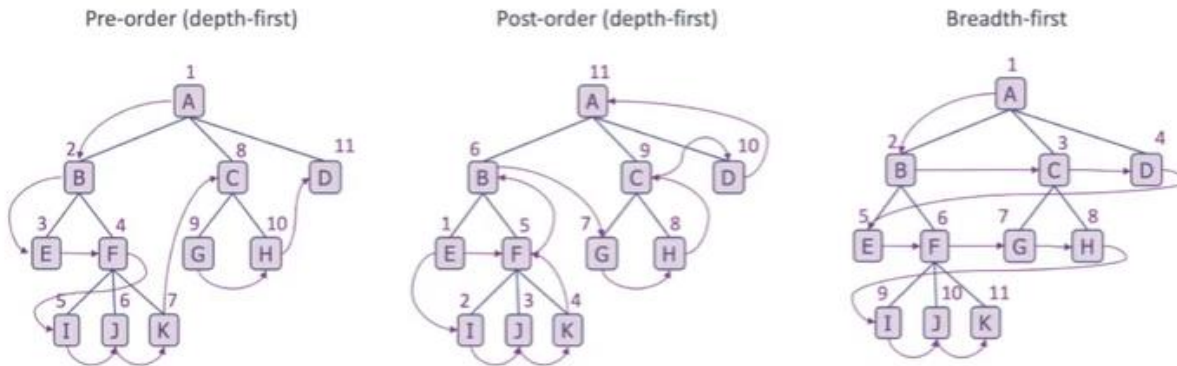
## Example: Tic tac toe game

Use a *game tree* to represent all possible moves that can be made by a player.  
The root of the tree represents the initial configuration for the game.



Using a breadth-first traversal we can consider all possible moves up to a certain depth.

## Tree traversals



This tree has at most 2 children, an order paired composed of the left child and right child.

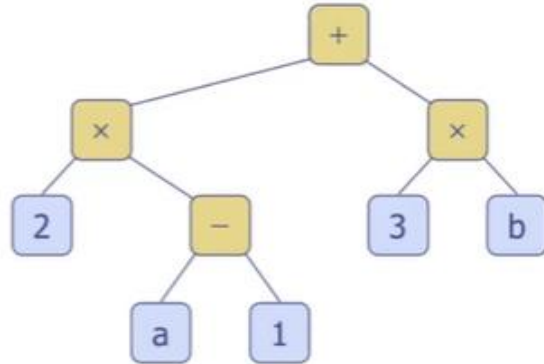
An arithmetic expression can be expressed with internal node operators and external nodes operands where the hierarchy of the tree denotes the order in which these operations must be performed.

The higher the depth, the earlier it needs to be computed.

# Binary tree

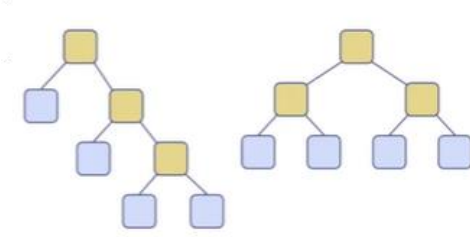
**Binary tree:** a tree with the following properties:

- Each internal node has at most two children (either 1 or 2).
- The children of a node are an ordered pair: left child, right child.



## Proper binary tree

**Proper binary tree:** tree in which every node has either zero or two children.



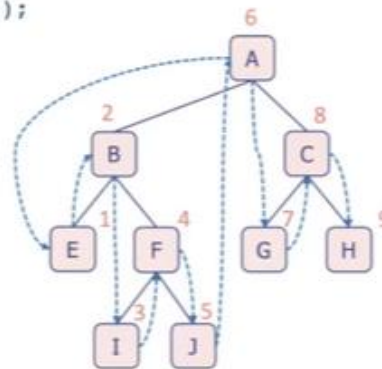
## In-order traversal in binary trees (depth-first)

Node is visited after left child and before right child.

```
public void inorder(Position<E> p) {
    if(left(p) != null) inorder(left(p));
    visit(p);
    if(right(p) != null) inorder(right(p));
}
```

Visited nodes:

E, B, I, F, J, A, G, C, H



# Properties of binary trees

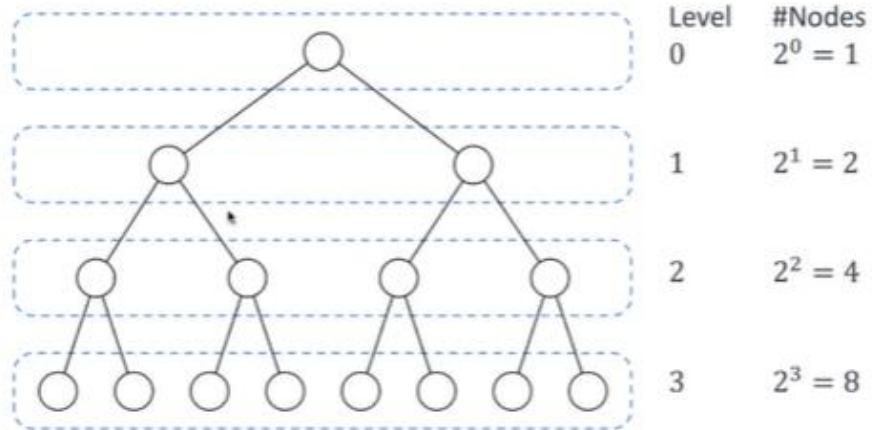
Relationship between number of nodes  $n$  and height  $h$ .

What is the maximum number of nodes in a binary tree with height  $h$ ?

(d)  $2^{h+1} - 1$

Sum the maximum number of nodes at all depths/levels.

$$\sum_{d=0}^h 2^d = 2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$$



## Linked structure for trees

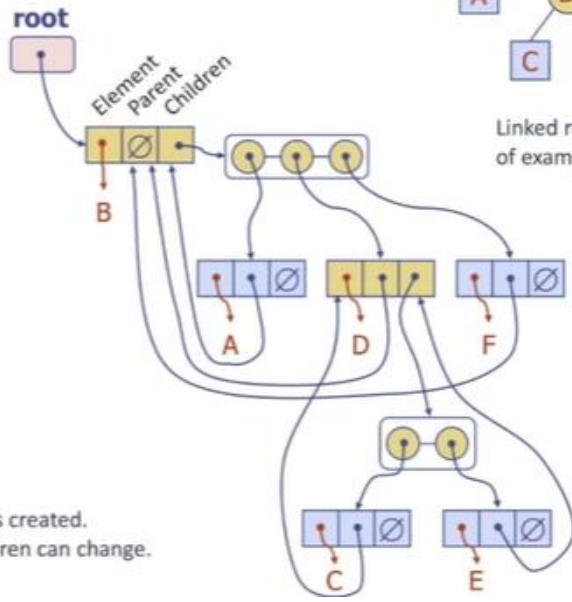
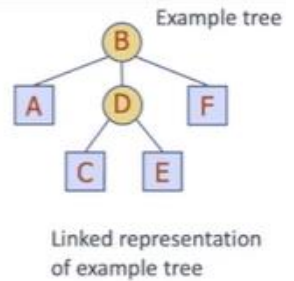
Tree node represented by object storing:

- Element
- Parent node
- Sequence of children nodes

Node class implements Position ADT.

```
public class LinkedTree<E>
    implements Tree<E> {
    protected static class Node<E>
        implements Position<E> {
        private E element;
        private Node<E> parent;
        private ?????<Node<E>> children;
        // ...
    }
    protected Node<E> root;
}
```

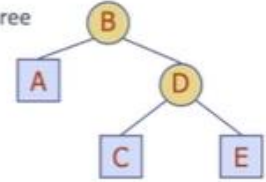
- **Array** if number of children is fixed when a node is created.
- **Dynamic array** or **linked list** if the number of children can change.





# Linked structure for binary trees

Example binary tree



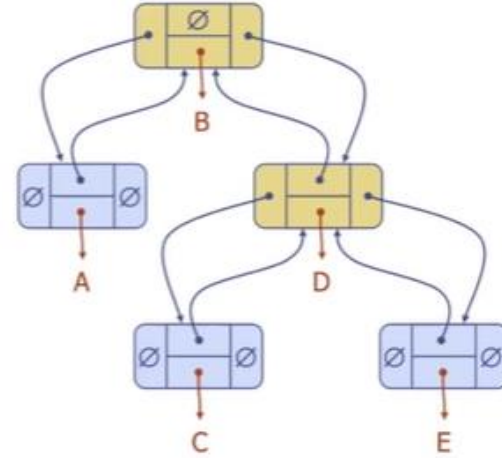
Binary tree node is an object storing

- Element
- Parent node
- Left child node
- Right child node

Node objects implement the Position ADT

```
public class LinkedBinaryTree<E> implements Tree<E> {
    protected static class Node<E> implements Position<E> {
        private E element;
        private Node<E> parent;
        private Node<E> left;
        private Node<E> right;
        // ...
    }
    protected Node<E> root;
}
```

Linked representation of example binary tree



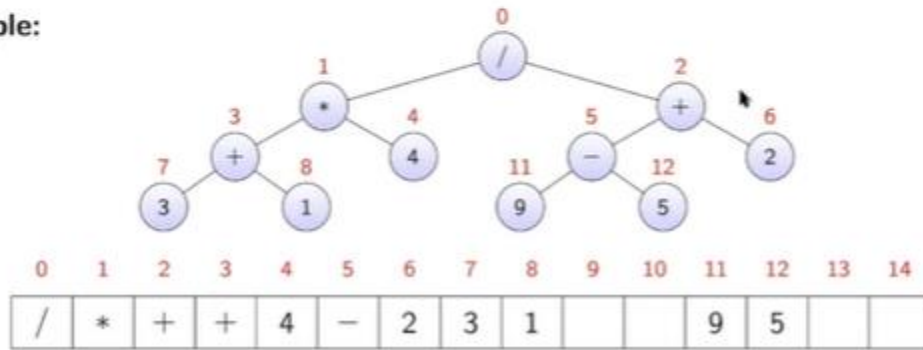
# Array-based binary trees

Nodes are stored in an array.

Node  $v$  is stored at index  $f(v)$ , given by:

- $f(v) = 0$  if  $v$  is the root
- $f(v) = 2 \cdot f(p) + 1$  if  $v$  is the left child of  $p$
- $f(v) = 2 \cdot f(p) + 2$  if  $v$  is the right child of  $p$

Example:



## Linked tree vs. array for binary trees

### Linked tree implementation

- $O(n)$  space (where  $n$  is the size of the tree)
- $O(1)$  operations for insertion/removal (see book, or previous lectures)

### Array-based implementation

- $O(2^h) / O(2^n)$  space. Exponential!
- Many update operations require changing the entire array.

**In summary:** array-based is only efficient for binary trees of particular shape, where only specific operations are performed (we'll cover an example in the next lecture: heaps).

## Priority queues

Priority cannot always be solved using FIFO (queue) or LIFO (stack) rules.

**Air traffic control:** decision on which plane lands first might depend on

- distance from runway
- time spent waiting in a holding pattern
- amount of remaining fuel

### Priority:

- **Key** associated with an element establishes its **priority**.
- Usually numeric. Any type, as long as keys can be compared.

## Priority: total order relation

Keys, or priorities, of different elements **must be comparable**.

For a comparison rule  $\leq$  to be self-consistent, it must define a **total order relation**.

It satisfies the following properties:

<b>Comparability</b>	$k_1 \leq k_2$ or $k_2 \leq k_1$ Relation $\leq$ is defined for every pair of keys.
<b>Anti-symmetry</b>	If $k_1 \leq k_2$ and $k_2 \leq k_1$ , then $k_1 = k_2$ If $\leq$ holds independently of the order of the pair of keys, then the keys in the pair are identical.
<b>Transitivity</b>	If $k_1 \leq k_2$ and $k_2 \leq k_3$ , then $k_1 \leq k_3$ If $\leq$ establishes orders $k_1 \leq k_2$ and $k_2 \leq k_3$ , then $k_1 \leq k_2 \leq k_3$ and thus $k_1 \leq k_3$ .
<b>Reflexivity</b>	$k \leq k$

If the relation  $\leq$  defines a **total order** over a finite set of keys, **minimal key**  $k_{min}$  is defined as:

$$k_{min} \leq k, \text{ where } k \text{ is any other key in the set}$$

## Comparable *versus* Comparator

**Natural ordering:** a class defines a natural ordering of instances by implementing `java.lang.Comparable`

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

**External ordering** can be defined for another class by implementing `java.util.Comparator` interface:

```
public interface Comparator<T> {
    int compare(T a, T b);
}
```

The syntaxes `a.compareTo(b)` and `compare(a, b)` must return an integer  $i$  as follows:

Negative	$i < 0$ ,	if $a < b$
Zero	$i = 0$ ,	if $a = b$
Positive	$i > 0$ ,	if $a > b$

**Example:** Ordering of String objects.

**Natural:** `compareTo` of String class defines natural ordering as lexicographic.

**lion**  $\leq$  **parrot**  $\leq$  **pig**

**External:** compare strings based on length.

**pig**  $\leq$  **lion**  $\leq$  **parrot**

```
public class StringLengthComparator
implements Comparator<String> {
    public int compare(String a, String b){
        if (a.length() < b.length()) return -1;
        if (a.length() == b.length()) return 0;
        return 1;
    }
}
```

## Priority Queue ADT

**Priority Queue:** collection of prioritized elements.

- Insertion at arbitrary positions.
- Removal of element with first priority.

**Priority Queue entries:**



**Priority Queue ADT**

<code>insert(k, v)</code>	creates an entry with key $k$ and value $v$ , $(k, v)$ , in the priority queue
<code>min()</code>	returns (does not remove) an entry $(k, v)$ with minimal key ( <b>null</b> if empty)
<code>removeMin()</code>	removes and returns an entry $(k, v)$ with minimal key ( <b>null</b> if empty)
<code>size()</code>	returns the number of entries
<code>isEmpty()</code>	returns <b>true</b> if empty, <b>false</b> otherwise

**Multiple entries with identical keys:** `min` and `removeMin` may report one of them arbitrarily.

## Priority Queue ADT

(*key*, *value*) entry:

- contains both the element and its priority
- composition design pattern: class that pairs key and value as a single object

```
public interface Entry<K,V> {
    K getKey();           // key is the priority
    V getValue();        // value is the element
}

public interface PriorityQueue<K,V> {
    int size();
    boolean isEmpty();
    Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
    Entry<K,V> min();
    Entry<K,V> removeMin();
}
```

## Priority Queue: implementations

### Unsorted list

**Insertion:** New entries are added at the end of the list, in  $O(1)$  time.

**Removal:** Access (*min*) and removal (*removeMin*) of minimal key is done by traversing list,  $O(n)$  time.

`public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> { ... } Code 9.6, p. 335`

### Sorted list

**Insertion:** Entries kept in non-decreasing order of keys. Insertion needs traversal to find position:  $O(n)$ .

**Removal:** Minimal key is always first in the list. Access (*min*) and removal (*removeMin*) are thus  $O(1)$ .

`public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> { ... } Code 9.7, p. 337`

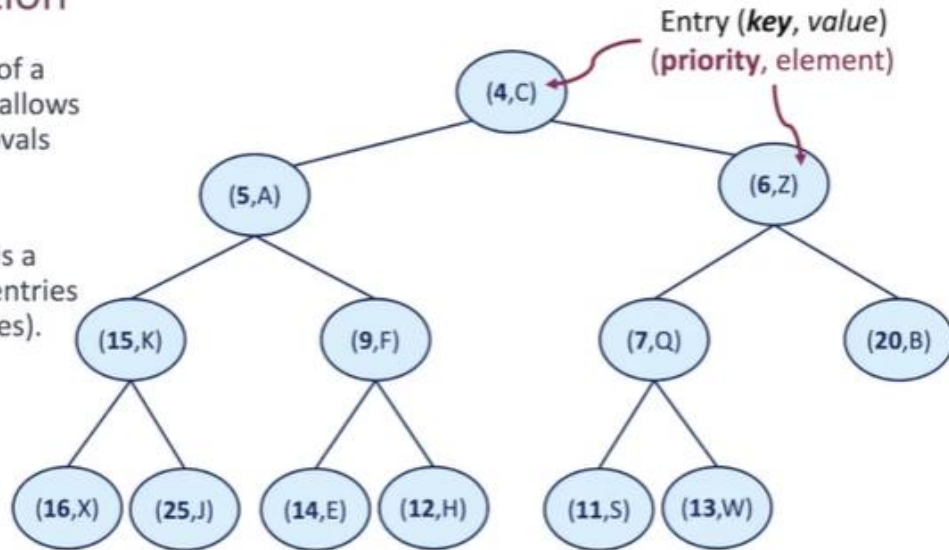
	Unsorted	Sorted
insert( <i>k,v</i> )	$O(1)$	$O(n)$
min()	$O(n)$	$O(1)$
removeMin()	$O(n)$	$O(1)$
size()	$O(1)$	$O(1)$
isEmpty()	$O(1)$	$O(1)$

Binary heaps

### Heap: definition

Efficient realization of a Priority Queue that allows insertions and removals in  $O(\log_2 n)$  time.

**Definition:** Heap  $\mathcal{T}$  is a binary tree storing entries at its positions (nodes).



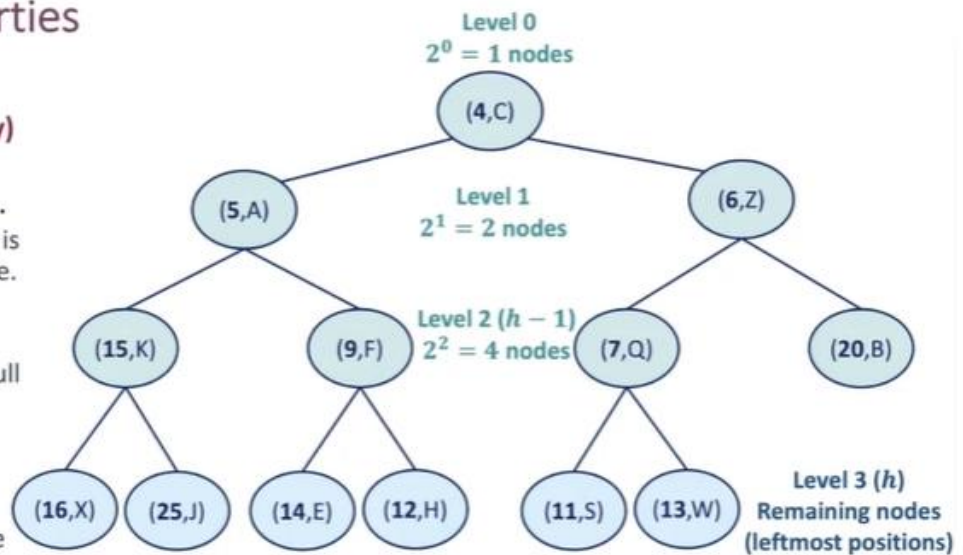
### Heap: properties

**(Structural property)**

**Complete binary tree.**  
Heap  $\mathcal{T}$  with height  $h$  is a complete binary tree.

All levels of  $\mathcal{T}$  except possibly the last are full (level  $i$  has  $2^i$  nodes, for  $0 \leq i \leq h - 1$ ).

Remaining nodes in last level  $h$  occupy the leftmost positions.

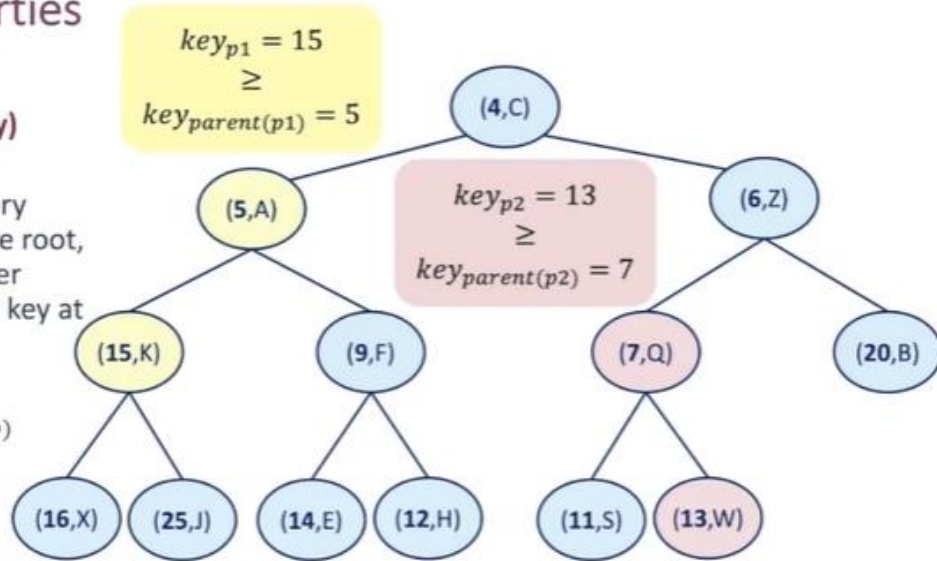


## Heap: properties

### (Relational property)

**Heap-order.** For every position  $p$  except the root, the key at  $p$  is greater than or equal to the key at its parent.

$$key_p \geq key_{parent(p)}$$



## Heap: height

### Height

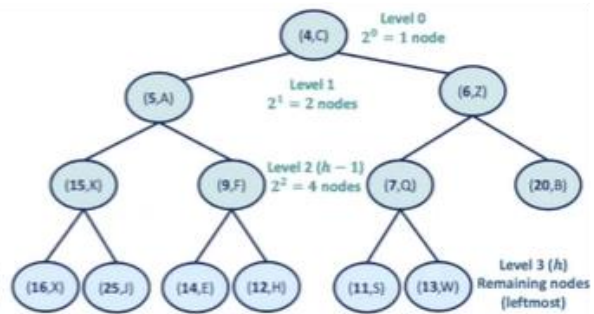
A heap  $\mathcal{T}$  with  $n$  entries has height  $h = \lfloor \log_2 n \rfloor$ .

Since  $\mathcal{T}$  is complete, number of nodes in levels 0 to  $h - 1$ :  
 $1 + 2 + \dots + 2^{h-1} = 2^h - 1$  (geometric series)

Number of nodes in level  $h$  is at least 1 and at most  $2^h$ .

Total number of nodes  $n$ :  $n \geq 2^h - 1 + 1 = 2^h$   
 $n \geq 2^h$   
 $h \leq \log_2 n$


and  $n \leq 2^h - 1 + 2^h$   
 and  $n + 1 \leq 2^{h+1}$   
 and  $h + 1 \geq \log_2(n + 1)$   
 and  $h \geq \log_2(n + 1) - 1$   
 Note that:  $\log_2(n + 1) - 1 \leq \log_2(n)$   
 arithmetic  
 take logarithm



Since  $h$  is a natural number, these inequalities imply that  $h = \lfloor \log_2 n \rfloor$ .

# Heap: insert

**insert( $k, v$ )** in a heap  $\mathcal{T}$   
 (we represent only the key)

#1 Create new node with entry. 

#2 **insert( $k, v$ )** in a heap  $\mathcal{T}$

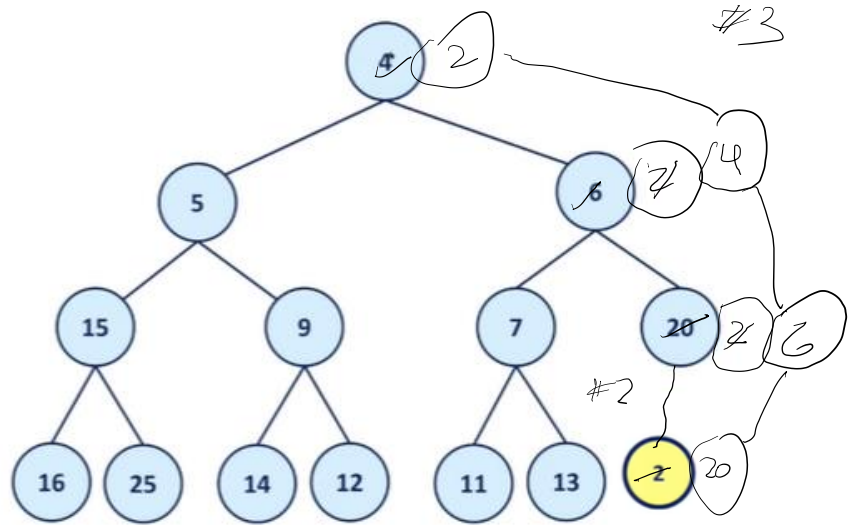
To preserve **completeness**:  
 Insert new node with  $(k, v)$  at position  $p$ :  
 - Just beyond rightmost node at last level  
 - As leftmost position of a new level

#3 **Process position  $p$ :**

To preserve **heap-order**:  
 If  $p$  is the root of  $\mathcal{T}$ , heap-order is satisfied.

Otherwise, compare key at  $p$  to that of  $p$ 's parent, denoted as  $q$ :

- If  $k_p \geq k_q$ , heap-order is satisfied.
- If  $k_p < k_q$ , restore heap-order by swapping.



# Heap: up-heap bubbling

**insert( $k, v$ )** in a heap  $\mathcal{T}$

Starts at the bottom.

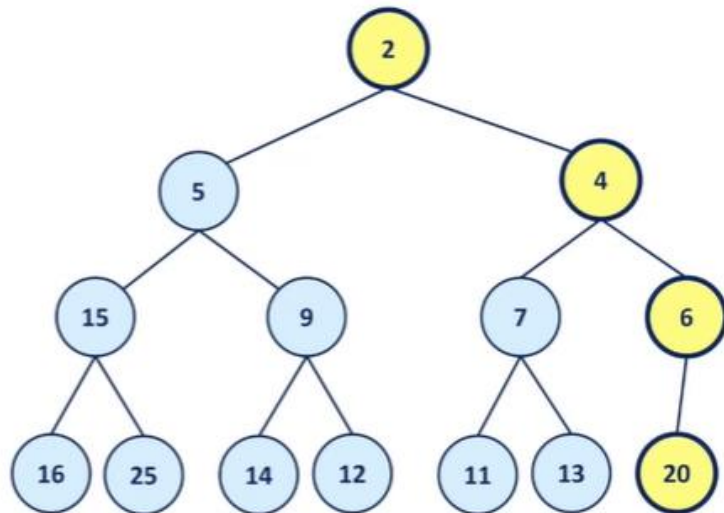
Each pairwise swap between a node and its parent either:

- Reestablishes heap-order
- Propagates the issue one level up

**Worst-case:**

- Entry (or key) moves all the way up to root.
- Number of moves is proportional to the height  $h$  of heap (recall that  $h = \lceil \log_2 n \rceil$ ).

$$O(\log_2 n)$$

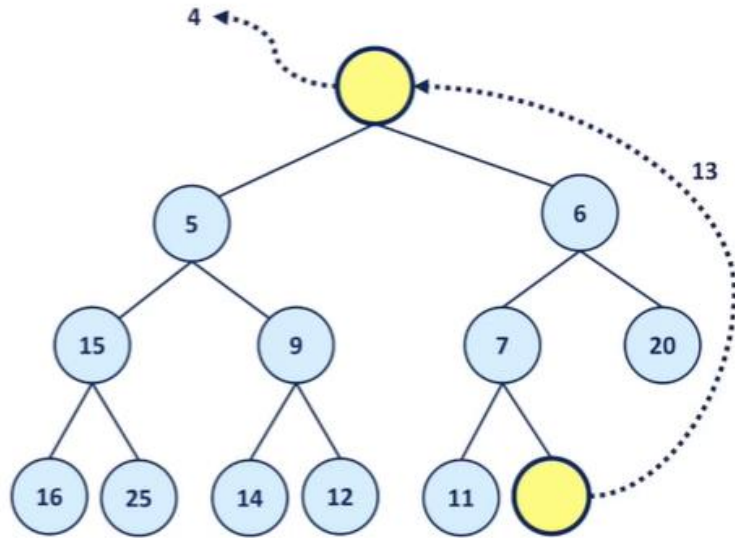


## Heap: removeMin

Entry with smallest key is at the root.

Cannot delete root node!  
To preserve completeness, replace root by last node.

To preserve heap-order ...



## Heap: down-heap bubbling after removal

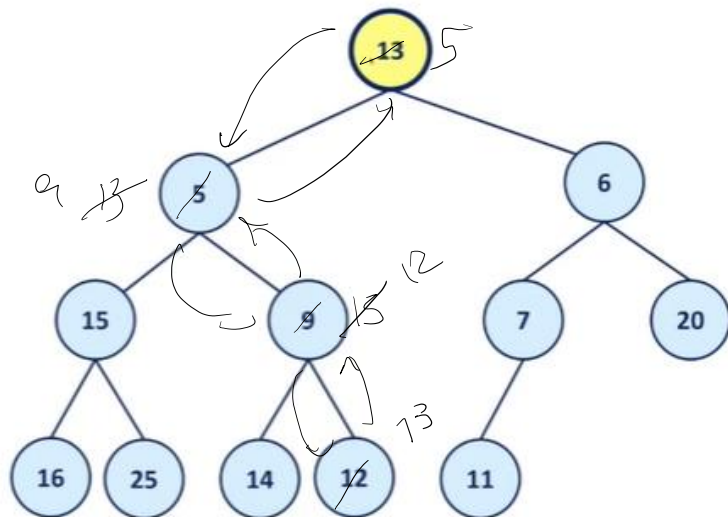
To preserve **heap-order**:

**Process position  $p$  (initially root of  $\mathcal{T}$ ):**

**[One node]** If  $\mathcal{T}$  has one node, heap-order is satisfied.

**[Multiple nodes]**

- If  $p$  has no right child,  $c$  is left child;
- Otherwise,  $c$  is child with minimal key.
- If  $k_p \leq k_c$ , the heap-order is satisfied.
- If  $k_p > k_c$ , restore heap-order, i.e. swap entries at  $p$  and  $c$ .





# Heap: down-heap bubbling

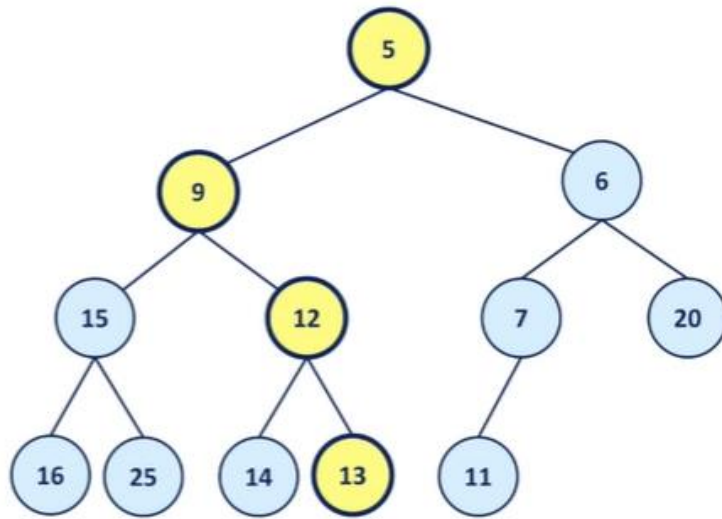
Starts at the root.

Each pairwise swap between a node and its child either:

- Reestablishes heap-order
- Propagates the issue one level down.

Worst-case:

- Entry (or key) moves all the way down to the bottom.
- Number of moves is proportional to height  $h$  of heap (recall  $h = \lceil \log_2 n \rceil$ ).



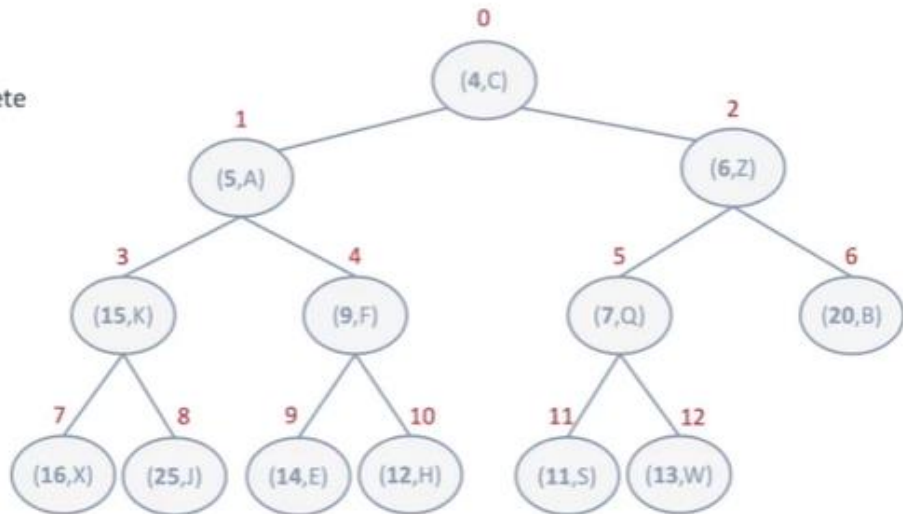
$$O(\log_2 n)$$

Heap representations

## Array heap

Arrays are suited for complete binary trees (e.g. heap).

Neatly "packed": no empty spaces in-between.



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

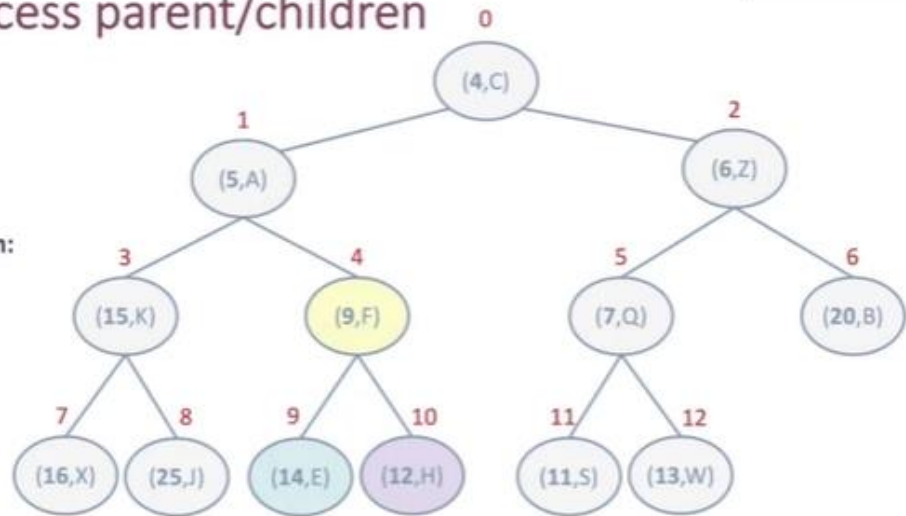
## Array heap: access parent/children

Accessing parent or children:

For position with index  $p$ :

Left child index  $2p + 1$

Right child index  $2p + 2$



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

## Array-based representation

Additional properties:

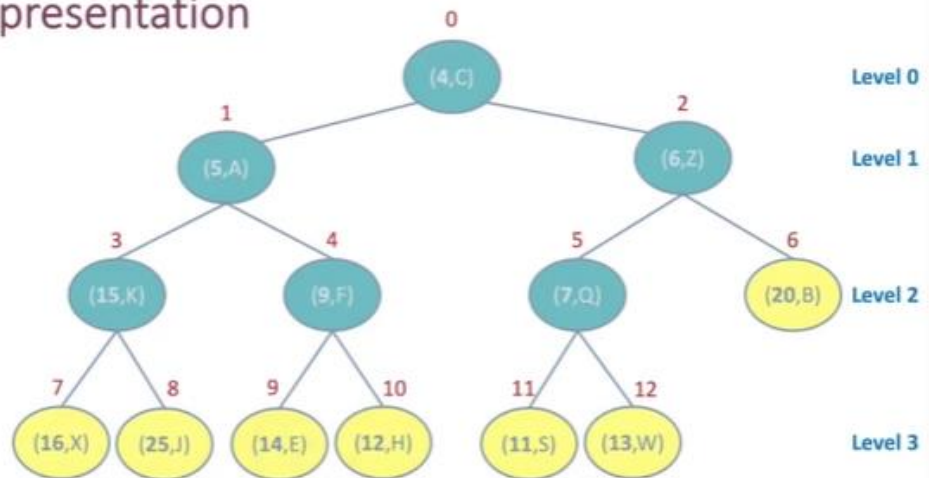
Levels are kept together.

Internal nodes leftmost

$$\left\{ 0, \dots, \left\lfloor \frac{n-1}{2} \right\rfloor - 1 \right\}$$

Leaf nodes rightmost

$$\left\{ \left\lfloor \frac{n-1}{2} \right\rfloor, \dots, n-1 \right\}$$



	(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Level	0	1	1	2	2	2	2	3	3	3	3	3	3

## Heap: array *versus* linked tree

Methods `insert` and `removeMin` rely on:

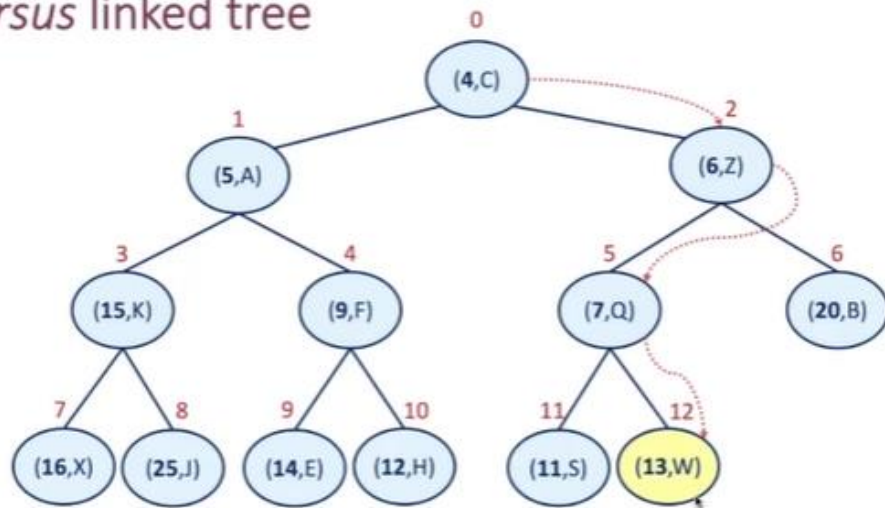
- locating the last position;
- local swapping between parent and child.

**Local swapping:**  $O(1)$  per swap,  $O(\log_2 n)$  in total (both array and linked tree).

**Accessing last position:**

In **array**, always index  $n - 1$ , thus access is  $O(1)$ .

In **linked tree**, requires traversal in  $O(\log_2 n)$ .



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

## Linked-tree: access last position

Tree path (binary):  
 Left 0  
 Right 1

We are interested in last node at last level.

Let us give indexes to nodes in last level.  
 Binary path denotes node index!

Node index is (#nodesLastLevel - 1).  
 How many nodes are in the last level?

Number of nodes in last level:

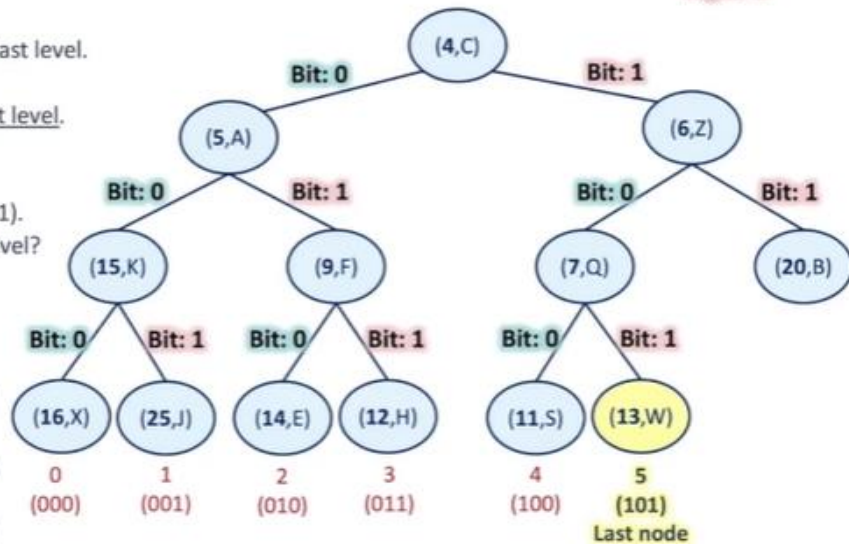
$$l = n - \left( \sum_{i=0}^{h-1} 2^i \right)$$

( $n$  - sum of all full levels, 0 up to  $h - 1$ )

$$l = n - 2^h + 1$$

$$= n - 2^{\lfloor \log_2 n \rfloor} + 1 \quad (h = \lfloor \log_2 n \rfloor)$$

Index of last node is  $n - 2^{\lfloor \log_2 n \rfloor}$ .



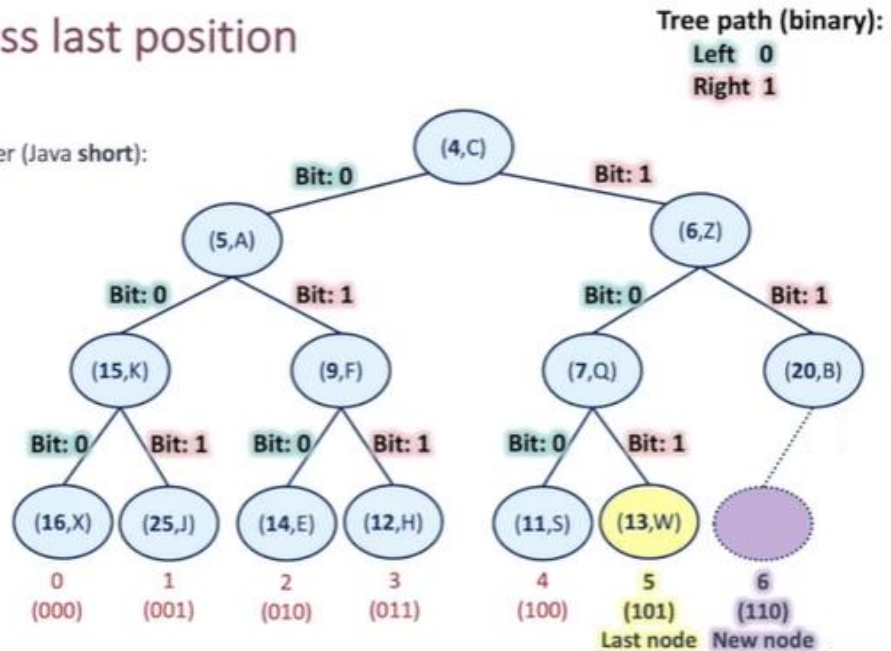
## Linked-tree: access last position

If index is encoded in 16-bit integer (Java short):  
00000000 00000101

**Index of last node:**  
 $n - 2^{\lfloor \log_2 n \rfloor}$   
Path: take  $\lfloor \log_2 n \rfloor$  rightmost bits  
00000000 00000101

**Index of a new node:**  
 $n + 1 - 2^{\lfloor \log_2(n+1) \rfloor}$   
Path of new node:  
 $\lfloor \log_2(n+1) \rfloor$  rightmost bits  
00000000 00000110

**Path of parent:**  
 $\lfloor \log_2(n+1) \rfloor$  rightmost **but one** bits  
00000000 00000110



## Priority Queue: complexity

Implementations of a Priority Queue with  $n$  elements.  
Assumes that two keys can be compared in  $O(1)$  time.

**Space complexity**  $O(n)$

### Time complexity

Method	List	Sorted List	Heap	Motivation (heap with height $O(\log_2 n)$ )
<code>min()</code>	$O(n)$	$O(1)$	$O(1)$	Root contains minimal key.
<code>removeMin()</code>	$O(n)$	$O(1)$	$O(\log_2 n)^*$	Down-heap bubbling performs $O(\log_2 n)$ swaps.
<code>insert(k, v)</code>	$O(1)$	$O(n)$	$O(\log_2 n)^*$	Up-heap bubbling performs $O(\log_2 n)$ swaps.
<code>size</code>	$O(1)$	$O(1)$	$O(1)$	Size is stored by an instance variable.
<code>isEmpty</code>	$O(1)$	$O(1)$	$O(1)$	Checks size variable.

\*Amortized time complexity for dynamic array-based representation, due to occasional resizing.

Heap construction

## Heap construction

Building a heap with  $n$  keys.

**Keys not known in advance:**

- Start with empty heap.
- Call `insert` method for every new key.  
 $O(n \log_2 n)$

**Keys known beforehand:**

How do we build a heap efficiently?

### Bottom-up heap merging (idea)

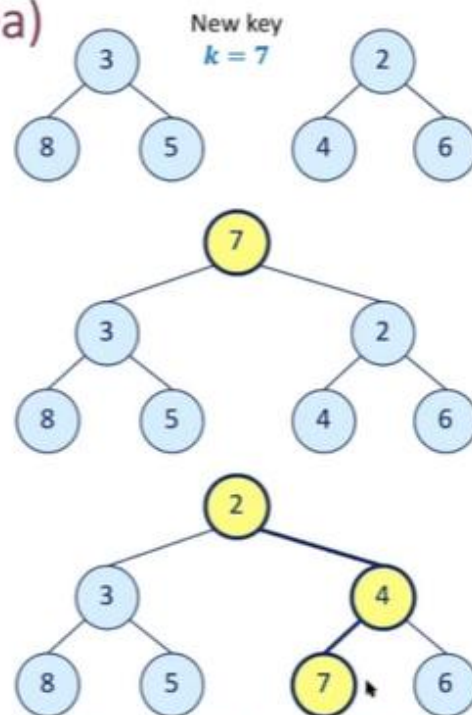
(merging idea as prelude for construction only)

**Input:** two heaps and new key  $k$ .

**Create new heap:**

- Root node storing  $k$ .
- Two given heaps as subtrees.

**Down-heap bubbling to restore heap-order property.**



**Note:** simplified example using a linked-tree structure and two heaps with all levels full.

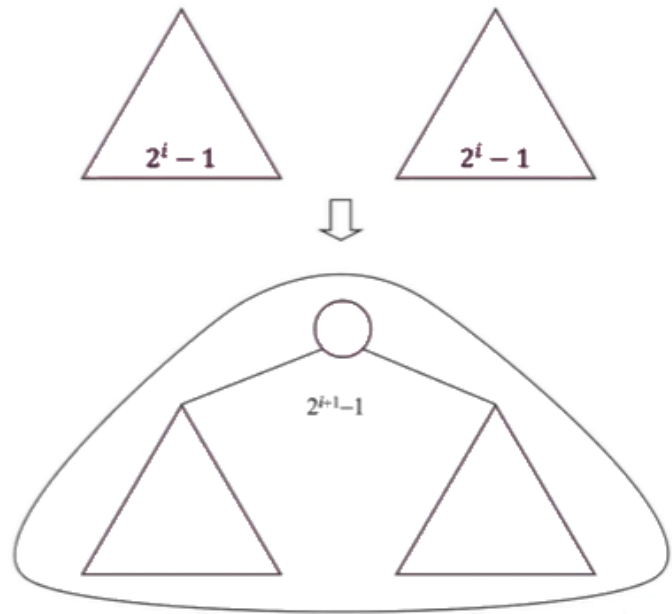
## Bottom-up heap construction

Building a heap with  $n$  keys.

Keys known beforehand:

Build heap using bottom-up approach.

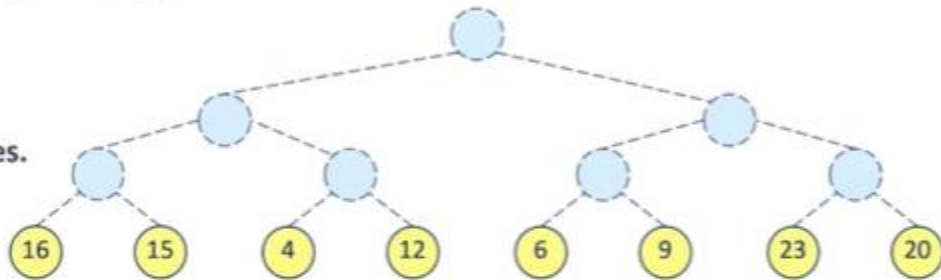
At iteration  $i$ , pairs of heaps with  $2^i - 1$  keys are merged into heaps with  $2^{i+1} - 1$ .



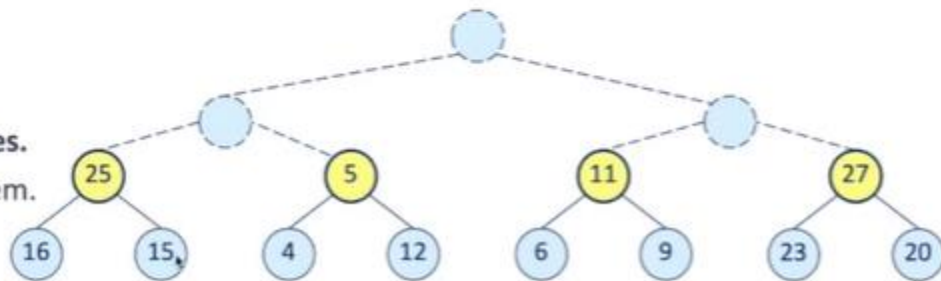
Leaf keys do not need to preserve any particular order since by not having any children nor parents and just siblings they are heap compliant already.

## Bottom-up heap construction

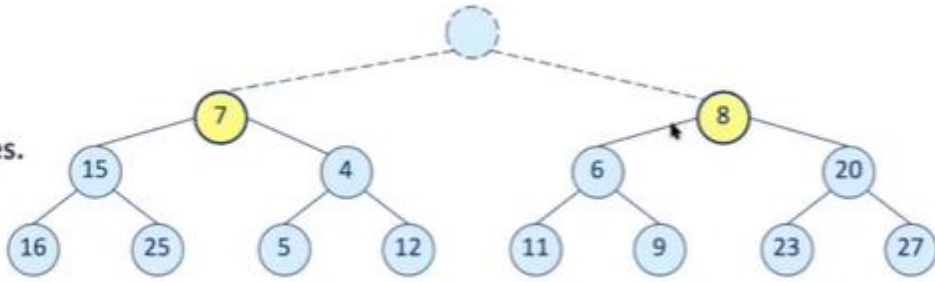
Iteration 1:  
Insert  $\frac{n+1}{2}$  nodes.



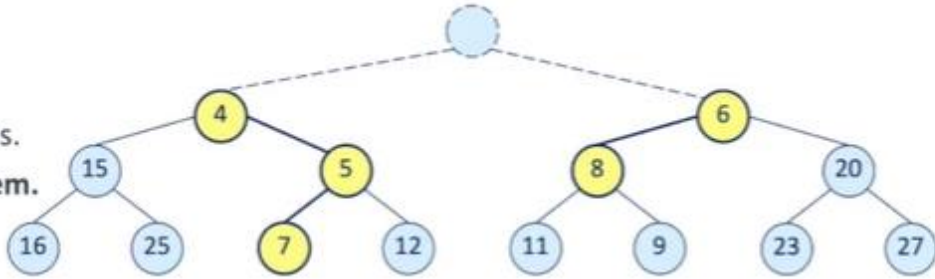
Iteration 2:  
Insert  $\frac{n+1}{4}$  nodes.  
Down-heap them.



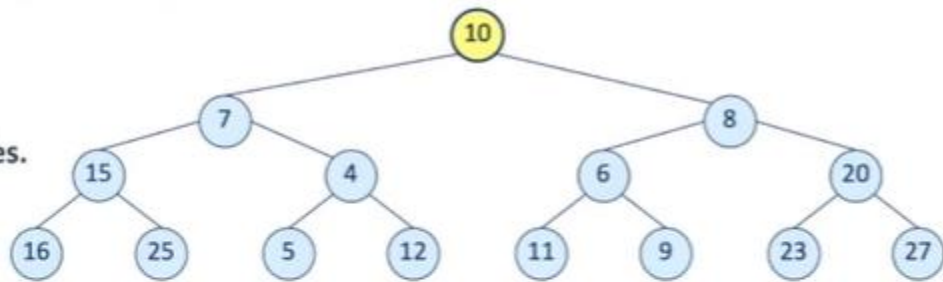
Iteration 3:  
 Insert  $\frac{n+1}{8}$  nodes.



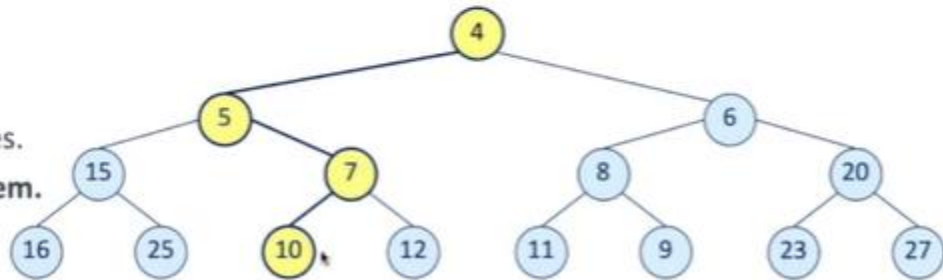
Iteration 3:  
 Insert  $\frac{n+1}{8}$  nodes.  
 Down-heap them.



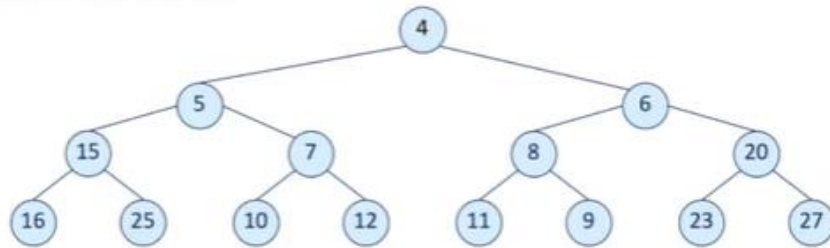
Iteration 4:  
 Insert  $\frac{n+1}{16}$  nodes.



Iteration 4:  
 Insert  $\frac{n+1}{16}$  nodes.  
 Down-heap them.



## Bottom-up heap construction



At Iteration  $i$ , with  $i \geq 0$ :

Insert  $\frac{n+1}{2^i}$  nodes.

Down-heap them.

Worst-case: down-heaping cost  $i$

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{n+1}{2^i} \times i$$

$$= (n+1) \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{i}{2^i}$$

$$= 2n + 2$$

Using identity for the sum of a geometric series, we can show the sum converges to 2.

**Bottom-up heap construction takes  $O(n)$  time.**

Faster than construction by entry insertion in  $O(n \log_2 n)$  time.

## Heapify: bottom-up heap construction (array)

Java code 9.10, page 350

```

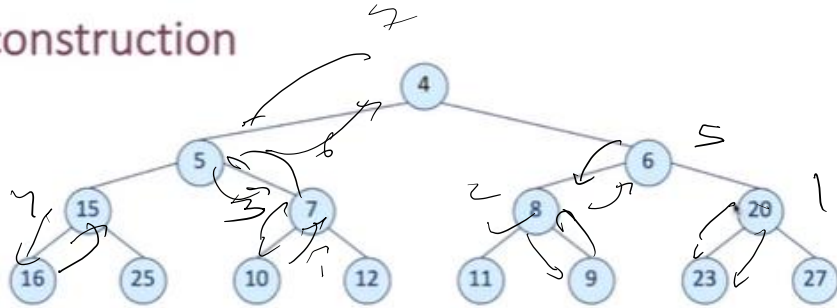
/**
 * Constructor of array-based heap with keys ks and values vs.
 */
public HeapPriorityQueue(K[] ks, V[] vs) {
    super(); // initialize array variable heap
    for (int j = 0; j < Math.min(ks.length, vs.length); j++) // iterate over keys and values
        heap.add(new PQEntry<>(keys[j], values[j])); // add all entries to the heap
    heapify(); // call heapify to bottom-up restore heap-order
}
/**
 * Heapify: Restores heap-order property in a bottom-up fashion.
 * Starting from the parent of the rightmost position (last level doesn't need
 * down-heap bubbling), and traverses backwards through all indices up to 0 (root).
 * For each node, performs down-heap bubbling to restore the heap-order property.
 */
protected void heapify() {
    int startIndex = parent(size()-1);
    for (int j = startIndex; j >= 0; j--)
        downheap(j);
}

```

1. Dump all the elements into the array
2. Heapify fixes the order in a bottom-up fashion. It will start from the parent of the right most position (the last level doesn't need to be sorted) and then traverse backwards through all indices up to 0/root and then perform down-heap bubbling.



## Bottom-up heap construction



Bottom-up heap construction in  $O(n)$  is intuitive using linked tree heaps.  
What if we have an array-based heap?

## Adaptable Priority Queue

Earlier, our Priority Queue ADT defined operations: `min`, `removeMin`, `insert`, `size`, `isEmpty`.

Some scenarios require additional operations.

At the airport:

- If a passenger withdraws from a waiting list, entry needs to be removed from the PQ.
- If a passenger finds gold frequent-flyer card, priority in the PQ needs to be updated.
- If the name of a passenger is misspelled, the entry needs to be corrected.

Adaptable PQ ADT:

`remove(e)`

Removes entry  $e$  from the priority queue (error if  $e$  is invalid; e.g. not in PQ anymore)

`replaceKey(e,k)`

Replaces the key of entry  $e$  with  $k$  (error if  $e$  is invalid; e.g. not in PQ anymore)

`replaceValue(e,v)`

Replaces the value of entry  $e$  with  $v$  (error if  $e$  is invalid; e.g. not in PQ anymore)

`min()`

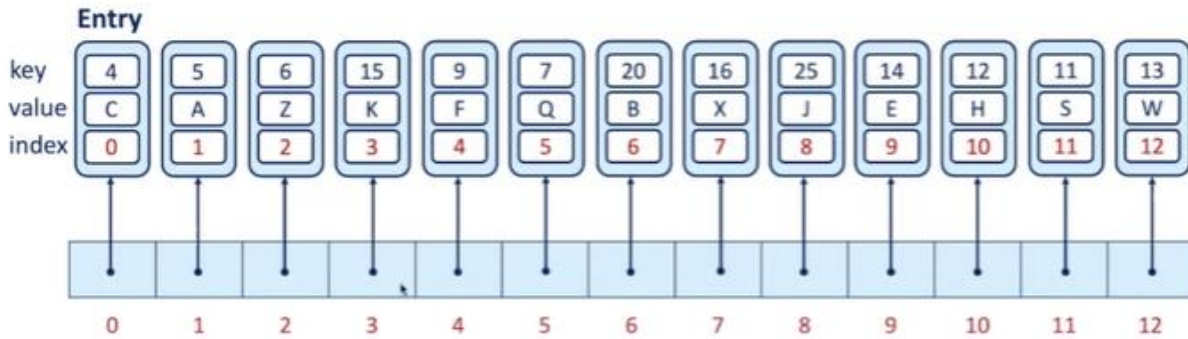
`removeMin()`

`insert(k, v)`

`size()`

`isEmpty()`

## Adaptable PQ



### Array heap with location-aware entries:

To implement `remove(e)`, `replaceKey(e, k)` and `replaceValue(e, v)` efficiently, we need fast way to locate entry `e`. Store the index in the entry itself.

```
public class HeapAdaptablePQ<K,V> extends HeapPQ<K,V> implements AdaptablePQ<K,V> { ... }
```

Note: in list or linked-tree implementations, entry contains a reference to its encapsulating node.

## Adaptable Priority Queue: complexity

Implementations of a Priority Queue with  $n$  elements.  
Assumes that two keys can be compared in  $O(1)$  time.

Space complexity  $O(n)$

### Time complexity

Method	List	Sorted List	Heap	Adaptable Heap
<code>min()</code>	$O(n)$	$O(1)$	$O(1)$	$O(1)$
<code>removeMin()</code>	$O(n)$	$O(1)$	$O(\log_2 n)^*$	$O(\log_2 n)^*$
<code>insert(k, v)</code>	$O(1)$	$O(n)$	$O(\log_2 n)^*$	$O(\log_2 n)^*$
<code>size()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>isEmpty()</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>remove()</code>	-	-	-	$O(\log_2 n)^*$
<code>replaceKey(e, k)</code>	-	-	-	$O(\log_2 n)^*$
<code>replaceValue(e, v)</code>	-	-	-	$O(1)$

\*Amortized time complexity for dynamic array-based representation, due to occasional resizing.

### 7.3 Position-based lists

Numeric indices are not a good choice for describing positions within a linked list because, knowing only an element's index, the only way to reach it is to traverse the list incrementally from its beginning or end, counting elements along the way.

Unfortunately, the public use of nodes in the ADT would violate the object-oriented design principles of abstraction and encapsulation, which were introduced in Chapter 2. There are several reasons to prefer that we encapsulate the nodes of a linked list, for both our sake and for the benefit of users of our abstraction:

1. It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation, such as low-level manipulation of nodes, or our reliance on the use of sentinel nodes
2. We can provide a more robust data structure if we do not permit users to directly access or manipulate the nodes.
3. By better encapsulating the internal details of our implementation, we have greater flexibility to redesign the data structure and improve its performance.

We introduce the concept of position, which formalizes the intuitive notion of the “location” of an element relative to others in the list.

A position

$p$ , which is associated with some element  $e$  in a list  $L$ , does not change, even if the index of  $e$  changes in  $L$  due to insertions or deletions elsewhere in the list. Nor does position  $p$  change if we replace the element  $e$  stored at  $p$  with another element. The only way in which a position becomes invalid is if that position (and its element) are explicitly removed from the list.

It has support for the method `getElement()`, which returns the element stored at this position.

A positional list is a collection of positions, each which stores an element. It should support the following

- `first()`: Returns the position of the first element of  $L$  (or null if empty).
- `last()`: Returns the position of the last element of  $L$  (or null if empty).
- `before( $p$ )`: Returns the position of  $L$  immediately before position  $p$  (or null if  $p$  is the first position).
- `after( $p$ )`: Returns the position of  $L$  immediately after position  $p$  (or null if  $p$  is the last position).
- `isEmpty()`: Returns true if list  $L$  does not contain any elements.
- `size()`: Returns the number of elements in list  $L$ .

The `first()` and `last()` methods of the positional list ADT return the associated positions, not the element. To get the element you could do something like `first().getElement()`

## Updated Methods of a Positional List

The positional list ADT also includes the following *update* methods:

- addFirst(*e*):** Inserts a new element *e* at the front of the list, returning the position of the new element.
- addLast(*e*):** Inserts a new element *e* at the back of the list, returning the position of the new element.
- addBefore(*p*, *e*):** Inserts a new element *e* in the list, just before position *p*, returning the position of the new element.
- addAfter(*p*, *e*):** Inserts a new element *e* in the list, just after position *p*, returning the position of the new element.
- set(*p*, *e*):** Replaces the element at position *p* with element *e*, returning the element formerly at position *p*.
- remove(*p*):** Removes and returns the element at position *p* in the list, invalidating the position.

If the `getElement()` method is called on a `Position` instance that has previously been removed from its list, an `IllegalStateException` is thrown. If an invalid `Position` instance is sent as a parameter to a method of a `PositionalList`, an `IllegalArgumentException` is thrown.

### Doubly Linked List Implementation

The obvious way to identify locations within a linked list are node references. Therefore, we declare the nested `Node` class of our linked list so as to implement the `Position` interface, supporting the required `getElement` method. So the nodes are the positions.

DoublyLinkedList //Implementation code available in IntelliJ workspace

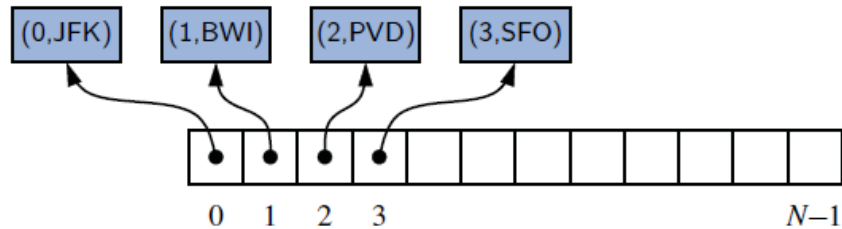
The positional list ADT is ideally suited for implementation with a doubly linked list, as all operations run in worst-case constant time. Index-based methods require traversing the nodes which would be  $O(n)$ .

Method	Running Time
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>first()</code> , <code>last()</code>	$O(1)$
<code>before(p)</code> , <code>after(p)</code>	$O(1)$
<code>addFirst(e)</code> , <code>addLast(e)</code>	$O(1)$
<code>addBefore(p, e)</code> , <code>addAfter(p, e)</code>	$O(1)$
<code>set(p, e)</code>	$O(1)$
<code>remove(p)</code>	$O(1)$

**Table 7.2:** Performance of a positional list with  $n$  elements realized by a doubly linked list. The space usage is  $O(n)$ .

## Array Implementation

Hence, if we are going to implement a positional list with an array, we need a different approach. We recommend the following representation. Instead of storing the elements of  $L$  directly in array  $A$ , we store a new kind of position object in each cell of  $A$ . A position  $p$  stores the element  $e$  as well as the current index  $i$  of that element within the list. Such a data structure is illustrated in Figure 7.8.



**Figure 7.8:** An array-based representation of a positional list.

With this representation, we can determine the index currently associated with a position, and we can determine the position currently associated with a specific index. We can therefore implement an accessor, such as `before( $p$ )`, by finding the index of the given position and using the array to find the neighboring position.

When an element is inserted or deleted somewhere in the list, we can loop through the array to update the index variable stored in all later positions in the list that are shifted during the update.

In this array implementation of a sequence, the `addFirst`, `addBefore`, `addAfter`, and `remove` methods take  $O(n)$  time, because we have to shift position objects to make room for the new position or to fill in the hole created by the removal of the old position (just as in the insert and remove methods based on index). All the other position-based methods take  $O(1)$  time.

## 7.4 Iterator

**hasNext():** Returns true if there is at least one additional element in the sequence, and false otherwise.

**next():** Returns the next element in the sequence.

The combination of these two methods allows a general loop construct for processing elements of the iterator. For example, if we let variable, `iter`, denote an instance of the `Iterator<String>` type, then we can write the following:

```
while (iter.hasNext()) {
    String value = iter.next();
    System.out.println(value);
}
```

The `java.util.Iterator` interface contains a third method, which is *optionally* supported by some iterators:

`remove()`: Removes from the collection the element returned by the most recent call to `next()`. Throws an `IllegalStateException` if `next` has not yet been called, or if `remove` was already called since the most recent call to `next`.

A single iterator instance supports only one pass through a collection; calls to `next` can be made until all elements have been reported, but there is no way to “reset” the iterator back to the beginning of the sequence.

Java defines another parameterized interface, named `Iterable`, that includes the following single method:

`iterator()`: Returns an iterator of the elements in the collection.

An instance of a typical collection class in Java, such as an `ArrayList`, is *iterable* (but not itself an *iterator*); it produces an iterator for its collection as the return value of the `iterator()` method. Each call to `iterator()` returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

Java’s `Iterable` class also plays a fundamental role in support of the “for-each” loop syntax (described in Section 1.5.2). The loop syntax,

```
for (ElementType variable : collection) {
    loopBody                                // may refer to "variable"
}
```

is supported for any instance, *collection*, of an iterable class. *ElementType* must be the type of object returned by its iterator, and *variable* will take on element values within the *loopBody*. Essentially, this syntax is shorthand for the following:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
    ElementType variable = iter.next();
    loopBody                                // may refer to "variable"
}
```

We note that the iterator's `remove` method cannot be invoked when using the `for-each` loop syntax. Instead, we must explicitly use an iterator. As an example, the following loop can be used to remove all negative numbers from an `ArrayList` of floating-point values.

```
ArrayList<Double> data; // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator();
while (walk.hasNext())
    if (walk.next() < 0.0)
        walk.remove();
```

## Iterator: implementations

### Snapshot iterator

- Maintains own private copy of the collection, constructed at creation.
- Unaffected by changes to the primary collection.
- Requires  $O(n)$  space and  $O(n)$  time upon construction (copy and keep collection of  $n$  elements).

### Lazy iterator

- Does not make a copy, directly traverses the primary data structure.
- Affected by changes to the primary collection.
- Requires  $O(1)$  space and  $O(1)$  construction time.
- 'fail-fast' behavior invalidates an iterator if its underlying collection is modified unexpectedly.

## 8.1-8.4 Trees

### Formal Tree Definition

Formally, we define a *tree*  $T$  as a set of *nodes* storing elements such that the nodes have a *parent-child* relationship that satisfies the following properties:

- If  $T$  is nonempty, it has a special node, called the *root* of  $T$ , that has no parent.
- Each node  $v$  of  $T$  different from the root has a unique *parent* node  $w$ ; every node with parent  $w$  is a *child* of  $w$ .

Note that according to our definition, a tree can be empty, meaning that it does not have any nodes. This convention also allows us to define a tree recursively such that a tree  $T$  is either empty or consists of a node  $r$ , called the root of  $T$ , and a (possibly empty) set of subtrees whose roots are the children of  $r$ .

concept of a *position* as an abstraction for a node of a tree. An element is stored at each position, and positions satisfy parent-child relationships that define the tree structure. A position object for a tree supports the method:

`getElement()`: Returns the element stored at this position.

The tree ADT then supports the following *accessor methods*, allowing a user to navigate the various positions of a tree  $T$ :

`root()`: Returns the position of the root of the tree (or null if empty).

`parent( $p$ )`: Returns the position of the parent of position  $p$  (or null if  $p$  is the root).

`children( $p$ )`: Returns an iterable collection containing the children of position  $p$  (if any).

`numChildren( $p$ )`: Returns the number of children of position  $p$ .

If a tree  $T$  is ordered, then `children( $p$ )` reports the children of  $p$  in order.

In addition to the above fundamental accessor methods, a tree supports the following *query methods*:

`isInternal( $p$ )`: Returns true if position  $p$  has at least one child.

`isExternal( $p$ )`: Returns true if position  $p$  does not have any children.

`isRoot( $p$ )`: Returns true if position  $p$  is the root of the tree.

These methods make programming with trees easier and more readable, since we can use them in the conditionals of **if** statements and **while** loops.

Trees support a number of more general methods, unrelated to the specific structure of the tree. These include:

`size()`: Returns the number of positions (and hence elements) that are contained in the tree.

`isEmpty()`: Returns true if the tree does not contain any positions (and thus no elements).

`iterator()`: Returns an iterator for all elements in the tree (so that the tree itself is iterable).

`positions()`: Returns an iterable collection of all positions of the tree.



Computing Depth

```

1  /** Returns the number of levels separating Position p from the root. */
2  public int depth(Position<E> p) {
3      if (isRoot(p))
4          return 0;
5      else
6          return 1 + depth(parent(p));
7  }

```

**Code Fragment 8.3:** Method `depth`, as implemented within the `AbstractTree` class.

Computing Height

```

1  /** Returns the height of the subtree rooted at Position p. */
2  public int height(Position<E> p) {
3      int h = 0; // base case if p is external
4      for (Position<E> c : children(p))
5          h = Math.max(h, 1 + height(c));
6      return h;
7  }

```

**Code Fragment 8.5:** Method `height` for computing the height of a subtree rooted at a position  $p$  of an `AbstractTree`.

## 8.2.1 The Binary Tree Abstract Data Type

As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:

- `left(p)`:** Returns the position of the left child of  $p$  (or null if  $p$  has no left child).
- `right(p)`:** Returns the position of the right child of  $p$  (or null if  $p$  has no right child).
- `sibling(p)`:** Returns the position of the sibling of  $p$  (or null if  $p$  has no sibling).

## Operations for Updating a Linked Binary Tree

- addRoot( $e$ ):** Creates a root for an empty tree, storing  $e$  as the element, and returns the position of that root; an error occurs if the tree is not empty.
- addLeft( $p, e$ ):** Creates a left child of position  $p$ , storing element  $e$ , and returns the position of the new node; an error occurs if  $p$  already has a left child.
- addRight( $p, e$ ):** Creates a right child of position  $p$ , storing element  $e$ , and returns the position of the new node; an error occurs if  $p$  already has a right child.
- set( $p, e$ ):** Replaces the element stored at position  $p$  with element  $e$ , and returns the previously stored element.
- attach( $p, T_1, T_2$ ):** Attaches the internal structure of trees  $T_1$  and  $T_2$  as the respective left and right subtrees of leaf position  $p$  and resets  $T_1$  and  $T_2$  to empty trees; an error condition occurs if  $p$  is not a leaf.
- remove( $p$ ):** Removes the node at position  $p$ , replacing it with its child (if any), and returns the element that had been stored at  $p$ ; an error occurs if  $p$  has two children.

Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth( $p$ )	$O(d_p + 1)$
height	$O(n)$

**Table 8.1:** Running times for the methods of an  $n$ -node binary tree implemented with a linked structure. The space usage is  $O(n)$ .

## 9.1 Priority queues

When an element is added to a priority queue, the user designates its priority by providing an associated key. The element with the minimal key will be the next to be removed from the queue.

We define the priority queue ADT to support the following methods:

- `insert( $k, v$ )`: Creates an entry with key  $k$  and value  $v$  in the priority queue.
- `min()`: Returns (but does not remove) a priority queue entry  $(k,v)$  having minimal key; returns null if the priority queue is empty.
- `removeMin()`: Removes and returns an entry  $(k,v)$  having minimal key from the priority queue; returns null if the priority queue is empty.
- `size()`: Returns the number of entries in the priority queue.
- `isEmpty()`: Returns a boolean indicating whether the priority queue is empty.

A priority queue may have multiple entries with equivalent keys, in which case methods `min` and `removeMin` may report an arbitrary choice among those entry having minimal key. Values may be any type of object.

## 9.2 List-based priority queues

### 9.2.2 Comparing Keys with Total Orders

In defining the priority queue ADT, we can allow any type of object to serve as a key, but we must be able to compare keys to each other in a meaningful way. More so, the results of the comparisons must not be contradictory. For a comparison rule, which we denote by  $\leq$ , to be self-consistent, it must define a *total order* relation, which is to say that it satisfies the following properties for any keys  $k_1, k_2$ , and  $k_3$ :

- **Comparability property:**  $k_1 \leq k_2$  or  $k_2 \leq k_1$ .
- **Antisymmetric property:** if  $k_1 \leq k_2$  and  $k_2 \leq k_1$ , then  $k_1 = k_2$ .
- **Transitive property:** if  $k_1 \leq k_2$  and  $k_2 \leq k_3$ , then  $k_1 \leq k_3$ .

The comparability property states that comparison rule is defined for every pair of keys. Note that this property implies the following one:

- **Reflexive property:**  $k \leq k$ .

Such a rule defines a linear ordering among a set of keys; hence, if a (finite) set of elements has a total order defined for it, then the notion of a minimal key, `kmin`, is well defined.

## The Comparable Interface

Java provides two means for defining comparisons between object types. The first of these is that a class may define what is known as the *natural ordering* of its instances by formally implementing the `java.lang.Comparable` interface, which includes a single method, `compareTo`. The syntax `a.compareTo(b)` must return an integer  $i$  with the following meaning:

- $i < 0$  designates that  $a < b$ .
- $i = 0$  designates that  $a = b$ .
- $i > 0$  designates that  $a > b$ .

For example, the `compareTo` method of the `String` class defines the natural ordering of strings to be *lexicographic*, which is a case-sensitive extension of the alphabetic ordering to Unicode.

## The Comparator Interface

In some applications, we may want to compare objects according to some notion other than their natural ordering. For example, we might be interested in which of two strings is the shortest, or in defining our own complex rules for judging which of two stocks is more promising. To support generality, Java defines the `java.util.Comparator` interface. A *comparator* is an object that is external to the class of the keys it compares. It provides a method with the signature `compare(a, b)` that returns an integer with similar meaning to the `compareTo` method described above.

```

1 public class StringLengthComparator implements Comparator<String> {
2     /** Compares two strings according to their lengths. */
3     public int compare(String a, String b) {
4         if (a.length() < b.length()) return -1;
5         else if (a.length() == b.length()) return 0;
6         else return 1;
7     }
8 }

```

**Code Fragment 9.3:** A comparator that evaluates strings based on their lengths.

```

1 public class DefaultComparator<E> implements Comparator<E> {
2     public int compare(E a, E b) throws ClassCastException {
3         return ((Comparable<E>) a).compareTo(b);
4     }
5 }

```

**Code Fragment 9.4:** A `DefaultComparator` class that implements a comparator based upon the natural ordering of its element type.

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

**Table 9.2:** Worst-case running times of the methods of a priority queue of size  $n$ , realized by means of an unsorted or sorted list, respectively. We assume that the list is implemented by a doubly linked list. The space requirement is  $O(n)$ .

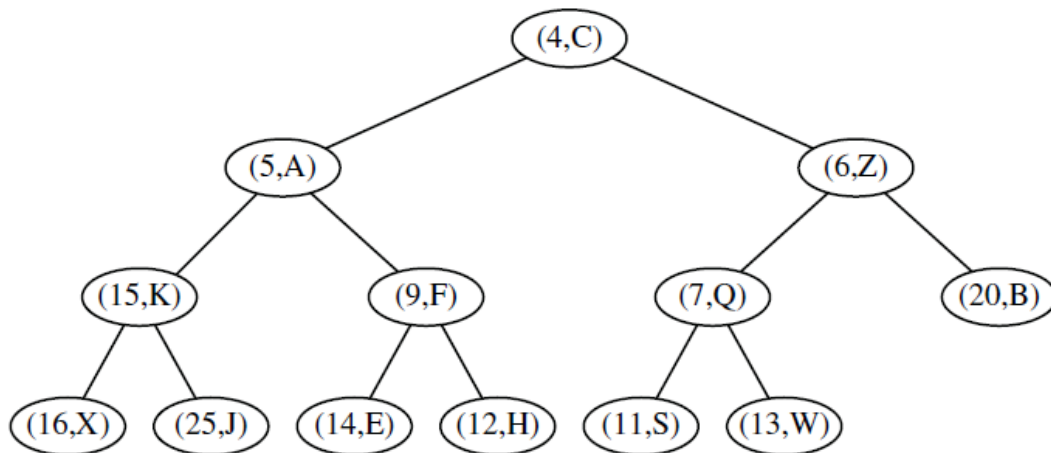
### 9.3 Tree-based priority queues: heaps

Trees allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations.

The fundamental way the heap achieves this improvement is to use the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted.

**Heap-Order Property:** In a heap  $T$ , for every position  $p$  other than the root, the key stored at  $p$  is greater than or equal to the key stored at  $p$ 's parent.

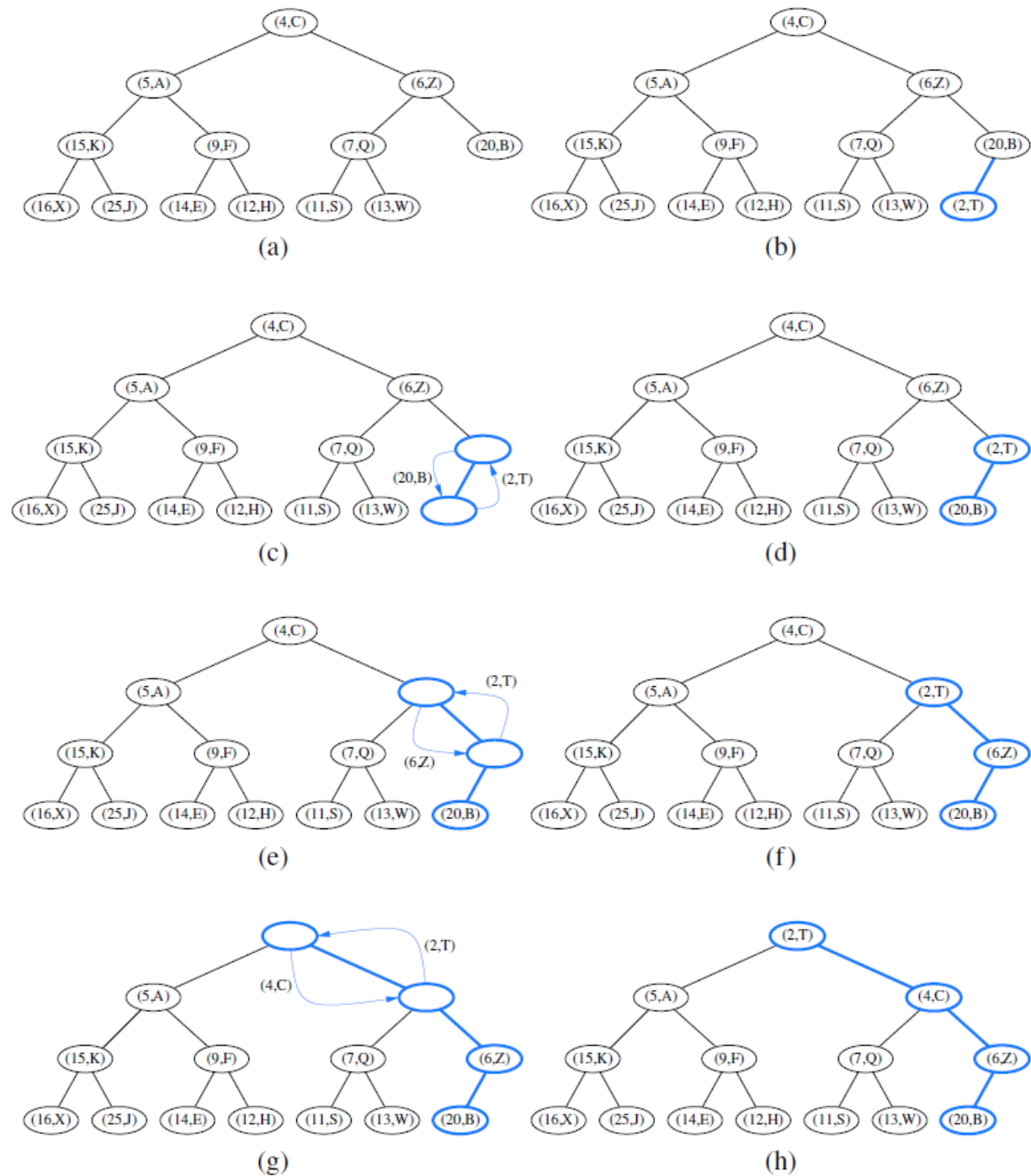
**Complete Binary Tree Property:** A heap  $T$  with height  $h$  is a *complete* binary tree if levels  $0, 1, 2, \dots, h-1$  of  $T$  have the maximal number of nodes possible (namely, level  $i$  has  $2^i$  nodes, for  $0 \leq i \leq h-1$ ) and the remaining nodes at level  $h$  reside in the leftmost possible positions at that level.



**Figure 9.1:** Example of a heap storing 13 entries with integer keys. The last position is the one storing entry  $(13, W)$ .

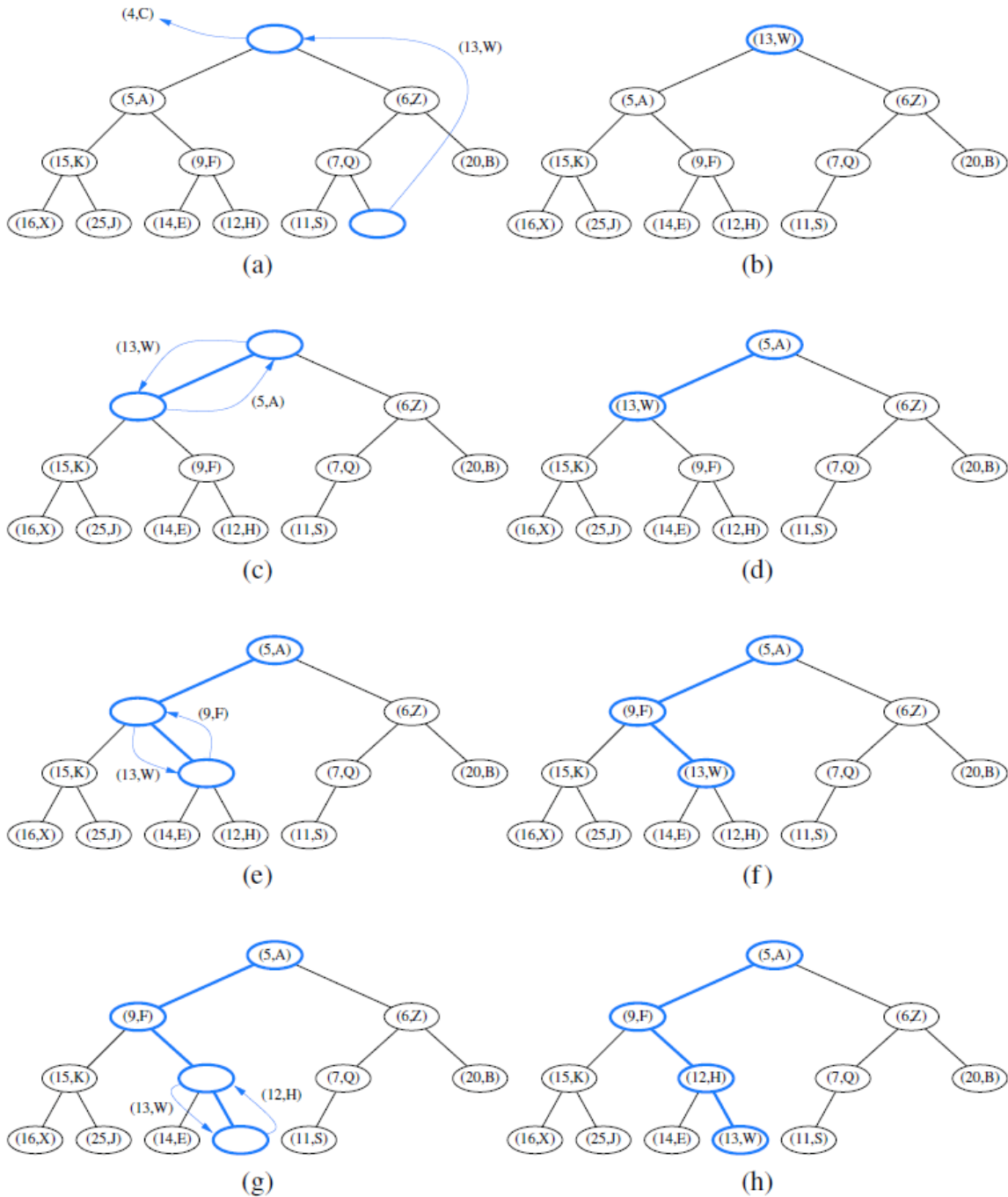
A heap  $T$  storing  $n$  entries has height  $h = \lceil \log n \rceil$ .

Up-heap Bubbling After an Insertion (at the bottom)



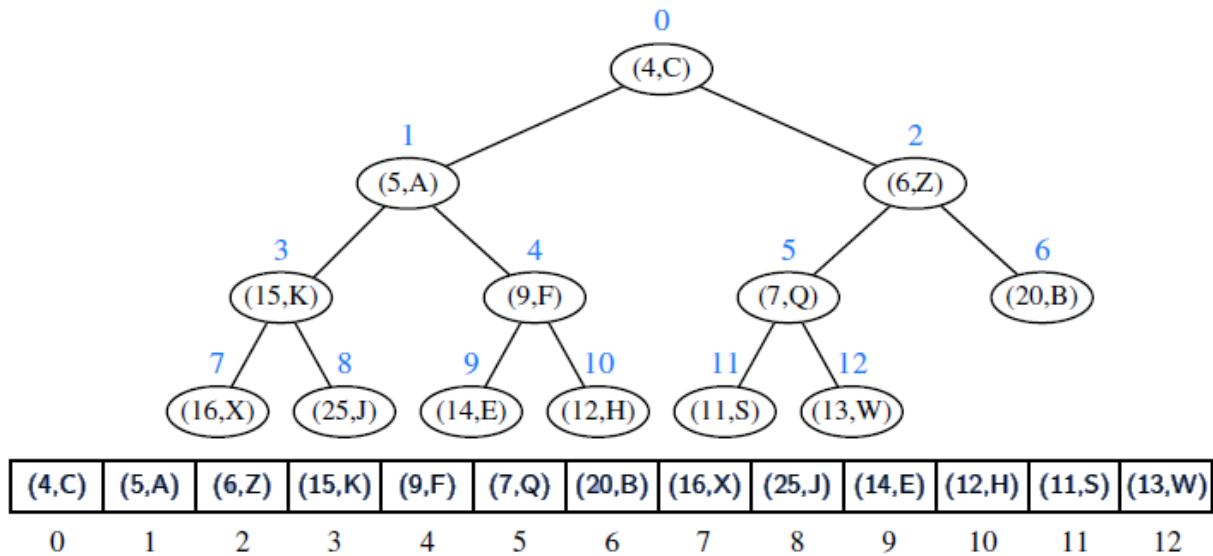
**Figure 9.2:** Insertion of a new entry with key 2 into the heap of Figure 9.1: (a) initial heap; (b) after adding a new node; (c and d) swap to locally restore the partial order property; (e and f) another swap; (g and h) final swap.

Down-heap bubbling after a removal



**Figure 9.3:** Removal of the entry with the smallest key from a heap: (a and b) deletion of the last node, whose entry gets stored into the root; (c and d) swap to locally restore the heap-order property; (e and f) another swap; (g and h) final swap.

Array-Based Representation of a Complete Binary Tree



**Figure 9.4:** Array-based representation of a heap.

Breadth first traversal leads to inner nodes being kept in the left side indexes and leaves on the right.

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

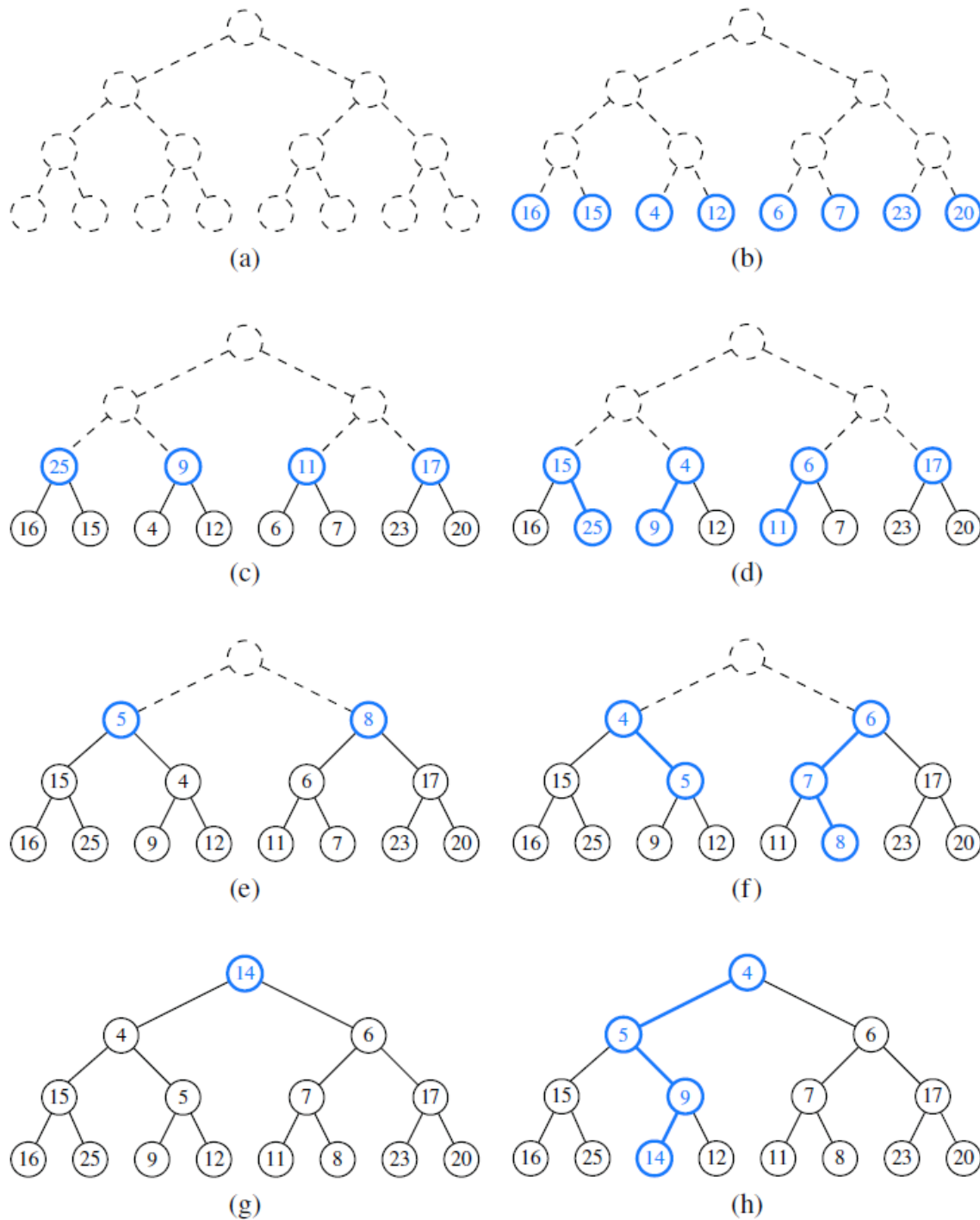
\*amortized, if using dynamic array

**Table 9.3:** Performance of a priority queue realized by means of a heap. We let  $n$  denote the number of entries in the priority queue at the time an operation is executed. The space requirement is  $O(n)$ . The running time of operations min and removeMin are amortized for an array-based representation, due to occasional resizing of a dynamic array; those bounds are worst case with a linked tree structure.

**Proposition 9.3:** Bottom-up construction of a heap with  $n$  entries takes  $O(n)$  time, assuming two keys can be compared in  $O(1)$  time.



Bottom-up construction



**Figure 9.5:** Bottom-up construction of a heap with 15 entries: (a and b) we begin by constructing 1-entry heaps on the bottom level; (c and d) we combine these heaps into 3-entry heaps; (e and f) we build 7-entry heaps; (g and h) we create the final heap. The paths of the down-heap bubblings are highlighted in (d, f, and h). For simplicity, we only show the key within each node instead of the entire entry.

## 9.5 Adaptable priority queues

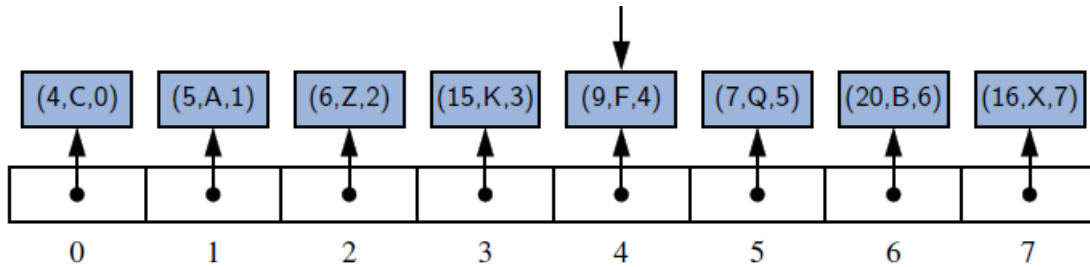
Formally, the adaptable priority queue ADT includes the following methods (in addition to those of the standard priority queue):

`remove( $e$ )`: Removes entry  $e$  from the priority queue.

`replaceKey( $e, k$ )`: Replaces the key of existing entry  $e$  with  $k$ .

`replaceValue( $e, v$ )`: Replaces the value of existing entry  $e$  with  $v$ .

A third value “token” corresponds to the index of an entry.



**Figure 9.10:** Representing a heap using an array of location-aware entries. The third field of each entry instance corresponds to the index of that entry within the array. Identifier token is presumed to be an entry reference in the user’s scope.

Method	Running Time
size, isEmpty, min	$O(1)$
insert	$O(\log n)$
remove	$O(\log n)$
removeMin	$O(\log n)$
replaceKey	$O(\log n)$
replaceValue	$O(1)$

**Table 9.5:** Running times of the methods of an adaptable priority queue with size  $n$ , realized by means of our array-based heap representation. The space requirement is  $O(n)$ .

## Week 4.1 Sort

The algorithm for sorting a sequence  $S$  with a priority queue  $P$  is quite simple and consists of the following two phases:

1. In the first phase, we insert the elements of  $S$  as keys into an initially empty priority queue  $P$  by means of a series of  $n$  insert operations, one for each element.
2. In the second phase, we extract the elements from  $P$  in nondecreasing order by means of a series of  $n$  removeMin operations, putting them back into  $S$  in that order.

Insertion vs selection sorts both  $O(n^2)$

They both follow the steps above and they take the name in relation to the bottleneck of their approach. In selection popping the sequence into the priority que is done fast by just adding the at the end, then they are selected before they are popped back into the final sequence. In Insertion they are inserted in order into the priority queue, then first element is popped back into  $S$  until  $P$  is empty.

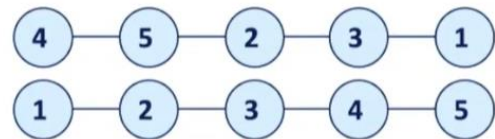
## Sorting with a priority queue

Sorting a sequence  $S$  with a priority queue.

**Algorithm  $PQ\text{-Sort}(S, C)$**

**Input** sequence  $S$ , comparator  $C$  for elements of  $S$

**Output** sequence  $S$  in non-decreasing order based on  $C$



## Sorting with list-based PQs

**Algorithm  $PQ\text{-Sort}(S, C)$**

**Input** sequence  $S$ , comparator  $C$

**Output** ordered sequence  $S$

$P \leftarrow$  priority queue using  $C$

**while**  $\neg S.isEmpty()$

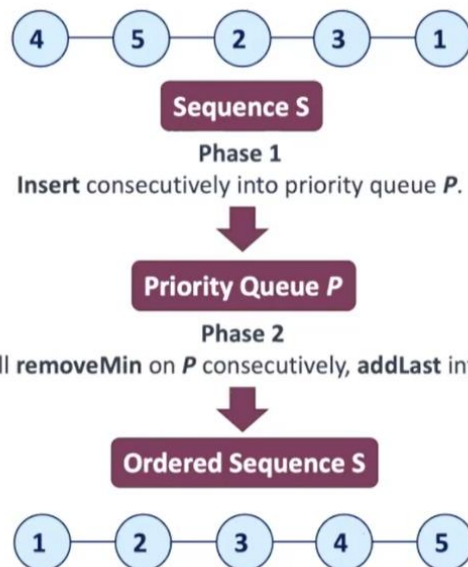
$e \leftarrow S.remove(S.first())$

$P.insert(e, \emptyset)$

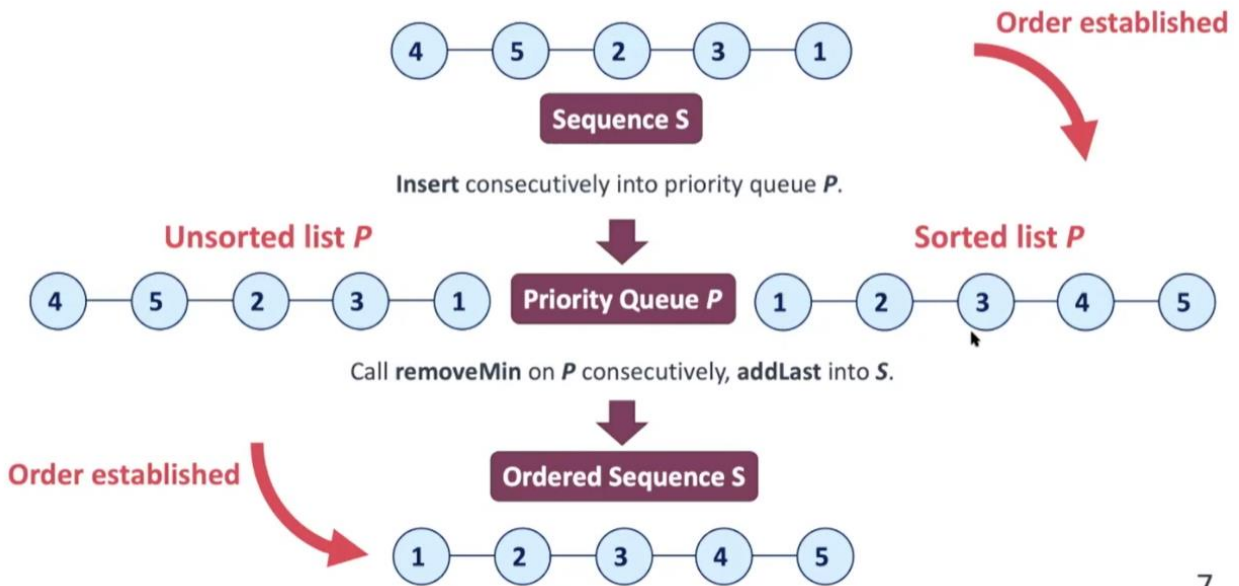
**while**  $\neg P.isEmpty()$

$e \leftarrow P.removeMin()$

$S.addLast(e)$



## Sorting with list-based PQs



7

## Insertion sort

Sorting  $n$  elements.  
Sorted list priority queue  $P$ .

### Phase 1: insertion

Repeated insertions into priority queue  $P$ .  
Entries inserted at final sorted position.  
Runtime of `insert` proportional to size of  $P$ :

$$1c + 2c + \dots + (n-1)c + n \cdot c$$

$$= c \sum_{i=1}^n i = c \frac{n(n+1)}{2} \quad O(n^2)$$

### Phase 2: removal

Repeated removal of minimal key from  $P$ .  
Each removal is  $O(1)$ ,  $n$  removals  $O(n)$ .

Input:	Sequence $S$	Priority queue $P$
	(7,4,8,2,5,3,9)	()
<b>Phase 1</b>		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
<b>Phase 2</b>		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
(c)	(2,3,4)	(5,7,8,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

## Selection sort

Sorting  $n$  elements.

Unsorted list priority queue  $P$ .

### Phase 1: insertion

Insertion into priority queue.

An insertion is  $O(1)$ ,  $n$  insertions  $O(n)$ .

### Phase 2: removal

Repeated removal of minimal key from  $P$ .

Runtime of each `removeMin` is proportional to the size of  $P$ :

$$n \cdot c + (n-1)c + \dots + 2c + 1c$$

$$= c \sum_{i=1}^n i = c \frac{n(n+1)}{2} \quad O(n^2)$$

	Sequence S	Priority Queue P
<b>Input</b>	(7,4,8,2,5,3,9)	( )
<b>Phase 1</b>		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
(c)	(2,5,3,9)	(7,4,8)
(d)	(5,3,9)	(7,4,8,2)
(e)	(3,9)	(7,4,8,2,5)
(f)	(9)	(7,4,8,2,5,3)
(g)	( )	(7,4,8,2,5,3,9)
<b>Phase 2</b>		
(a)	(2)	(7,4,8,2,5,3,9)
(b)	(2,3)	(7,4,8,5,3,9)
(c)	(2,3,4)	(7,4,8,5,9)
(d)	(2,3,4,5)	(7,8,5,9)
(e)	(2,3,4,5,7)	(7,8,9)
(f)	(2,3,4,5,7,8)	(8,9)
(g)	(2,3,4,5,7,8,9)	(9)
		( )

10

## Insertion sort: space complexity

(sequence  $S$  is a linked list and modified)

Sorting  $n$  elements.

Sorted list priority queue  $P$ .

### Space complexity:

Depends on implementation.

If  $S$  is linked list whose size is modified by insert/remove.

The total number of elements in  $S$  and  $P$  combined is always equal to the number of elements originally in  $S$  (given as input).

Then space is  $O(1)$ .

*first*

What if  $S$  is an array list?

	Sequence S	Priority queue P	
<b>Input:</b>	(7,4,8,2,5,3,9)	( )	$n$ elements
<b>Phase 1</b>	<b>insert</b>		
(a)	(4,8,2,5,3,9)	(7)	$n$ elements
(b)	(8,2,5,3,9)	(4,7)	$n$ elements
(c)	(2,5,3,9)	(4,7,8)	$n$ elements
(d)	(5,3,9)	(2,4,7,8)	$n$ elements
(e)	(3,9)	(2,4,5,7,8)	$n$ elements
(f)	(9)	(2,3,4,5,7,8)	$n$ elements
(g)	( )	(2,3,4,5,7,8,9)	$n$ elements
<b>Phase 2</b>	<b>remove</b>		
(a)	(2)	(3,4,5,7,8,9)	$n$ elements
(b)	(2,3)	(4,5,7,8,9)	$n$ elements
(c)	(2,3,4)	(5,7,8,9)	$n$ elements
(d)	(2,3,4,5)	(7,8,9)	$n$ elements
(e)	(2,3,4,5,7)	(8,9)	$n$ elements
(f)	(2,3,4,5,7,8)	(9)	$n$ elements
(g)	(2,3,4,5,7,8,9)	( )	$n$ elements

## Insertion sort: space complexity

(sequence  $S$  is array or read-only linked list)

Sorting  $n$  elements.

Sorted list priority queue  $P$ .

**Space complexity:**

Depends on implementation.

**If  $S$  is linked list modified by insert/remove.**

Then space is  $O(1)$ .

**If  $S$  is an array of capacity  $n$  (or any linked list that does not shrink, elements are overridden).**

Then we need  $O(n)$  space for  $P$ .

**Space complexity is  $O(n)$  with array.**

**Can we use less space?**

**Input:**

**Phase 1**

- (a)
- (b)
- (c)
- (d)
- (e)
- (f)
- (g)

**Phase 2**

- (a)
- (b)
- (c)
- (d)
- (e)
- (f)
- (g)

**Sequence S**  
Insert/remove  
(array list)  
(7,4,8,2,5,3,9)

**Sequence S**  
Read/override  
(array or linked list)  
(7,4,8,2,5,3,9)

**Priority queue P**

( )

(7)

(4,7)

(4,7,8)

(2,4,7,8)

(2,4,5,7,8)

(2,3,4,5,7,8)

(2,3,4,5,7,8,9)

( ,4,8,2,5,3,9)

(7,4,8,2,5,3,9)

( , ,8,2,5,3,9)

(7,4,8,2,5,3,9)

( , , ,2,5,3,9)

(7,4,8,2,5,3,9)

( , , , ,5,3,9)

(7,4,8,2,5,3,9)

( , , , , ,3,9)

(7,4,8,2,5,3,9)

( , , , , , ,9)

(7,4,8,2,5,3,9)

( , , , , , , ,)

(7,4,8,2,5,3,9)

(2, , , , , , ,)

(2,4,8,2,5,3,9)

(3,4,5,7,8,9)

(2,3, , , , , ,)

(2,3,8,2,5,3,9)

(4,5,7,8,9)

(2,3,4, , , , ,)

(2,3,4,2,5,3,9)

(5,7,8,9)

(2,3,4,5, , , ,)

(2,3,4,5,5,3,9)

(7,8,9)

(2,3,4,5,7, , ,)

(2,3,4,5,7,3,9)

(8,9)

(2,3,4,5,7,8, ,)

(2,3,4,5,7,8,9)

(9)

(2,3,4,5,7,8,9)

(2,3,4,5,7,8,9)

( )

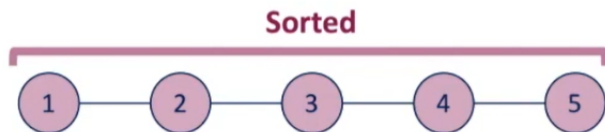
Similar concept applies to selection sort.

## In-place insertion sort

Selection and insertion sort can be implemented **in-place**.

**In-place sorting:**

Only  $O(1)$  space used (in addition to the sequence being sorted).



## In-place insertion sort

Insertion sort implemented **in-place**.

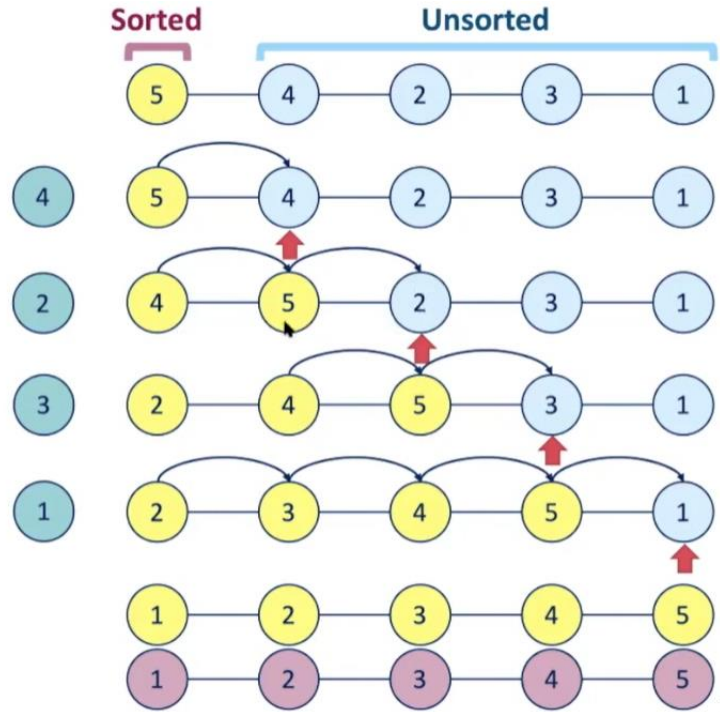
Only  $O(1)$  space used (in addition to the sequence being sorted).

A portion of the input sequence itself serves as the priority queue  $\mathcal{P}$ .

### In-place insertion sort:

Iterate left to right, one index  $i$  at a time (where  $i = 1$  up to  $i = n - 1$ ).

- Iterate backwards within  $0, \dots, i - 1$ :
  - If element larger than the element at index  $i$ , shift a position to the right.
- Insert element at its correct position.



## In-place selection sort

Selection sort implemented **in-place**.

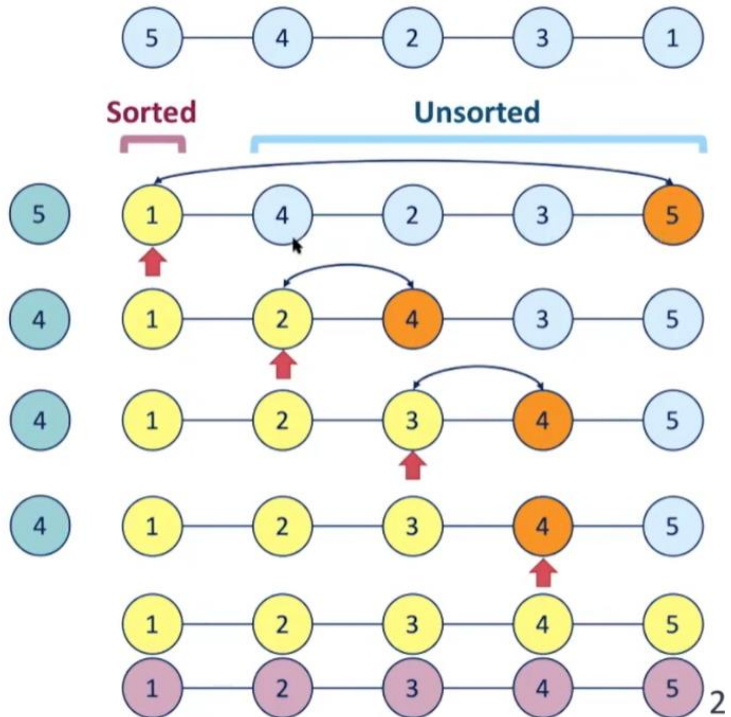
Only  $O(1)$  space used (in addition to the sequence being sorted).

A portion of the input sequence itself serves as the priority queue  $\mathcal{P}$ .

### In-place selection sort:

Iterate left to right, one index  $i$  at a time (where  $i = 0$  up to  $i = n - 2$ ).

- Find index  $m$  of minimum element within indexes  $i, \dots, n - 1$ .
- Swap elements at  $i$  and  $m$ .







Merge sort( $n \log n$ )

**Divide-and-conquer** is an **algorithmic design pattern** consisting of 3 steps:

1. **Divide:**

**[Small input: base case]**

If input is small (e.g. 1-2 elements), solve problem directly.

**[Larger input: recurrence]**

Divide the input into two or more disjoint subsets.

2. **Conquer:** Recursively solve the subproblems associated with the subsets.

3. **Combine:** Take the solutions of the subproblems and merge them into a solution to the larger problem.

**Merge sort** uses divide-and-conquer to sort a sequence  $S$  with  $n$  elements:

1. **Divide:**

**[Base case]**

If  $S$  has less than 2 element(s), return  $S$  (already sorted).

**[Recurrence]**

If  $S$  has at least 2 element(s), split elements of  $S$  into 2 sequences  $S_1$  and  $S_2$ .

$S_1$  and  $S_2$  contain each about half of the elements:  $S_1$  the first  $\lfloor n/2 \rfloor$ ,  $S_2$  the remaining  $\lceil n/2 \rceil$ .

2. **Conquer:** Recursively sort sequences  $S_1$  and  $S_2$ .

3. **Combine:** Put elements back into  $S$  by merging the sorted sequences  $S_1$  and  $S_2$  into a sorted sequence.

Execution of merge sort depicted by its **recursion tree**.

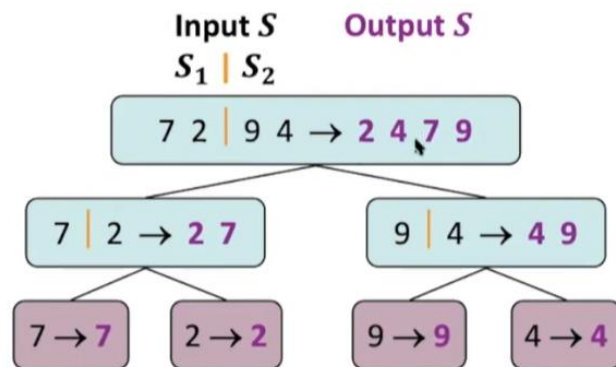
**Warning: This is not a tree data structure!**

Each node represents a recursive call of merge sort with:

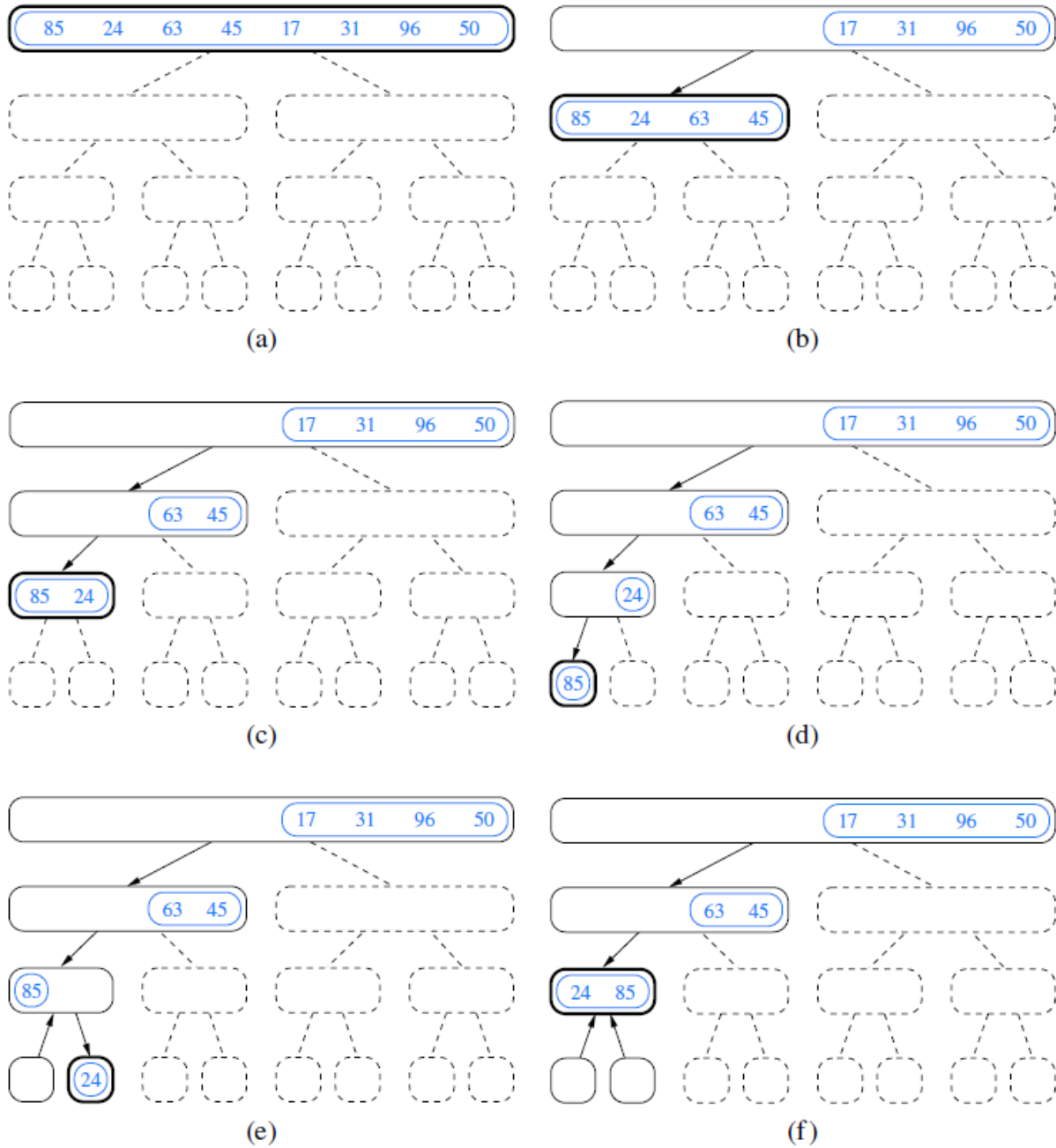
- unsorted sequence before the execution and its partition (**left**);
- sorted sequence at the end of the execution (**right**).

**Root** contains initial call, and final result.

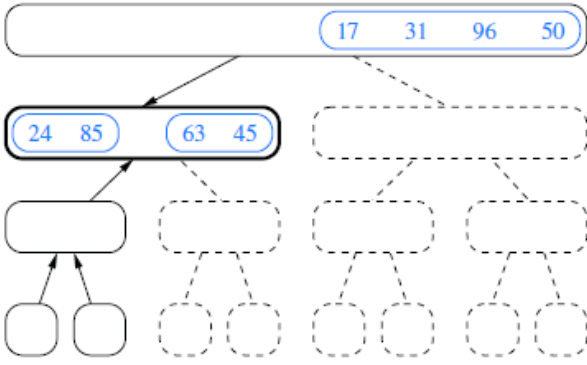
**Leaves** are calls on sequences of size 1.



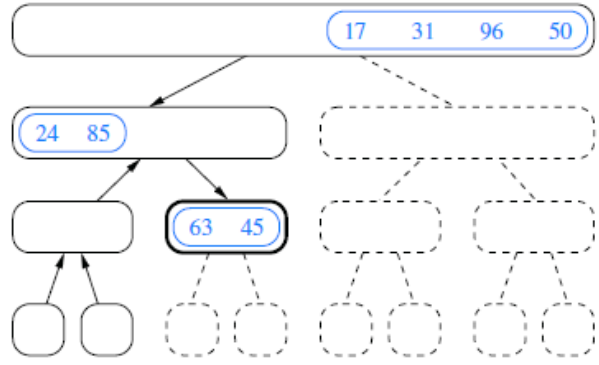
*Merge-sort-tree  
(calls, not a structure)*



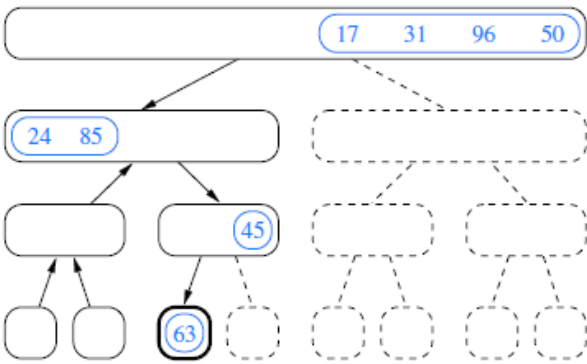
**Figure 12.2:** Visualization of an execution of merge-sort. Each node of the tree represents a recursive call of merge-sort. The nodes drawn with dashed lines represent calls that have not been made yet. The node drawn with thick lines represents the current call. The empty nodes drawn with thin lines represent completed calls. The remaining nodes (drawn with thin lines and not empty) represent calls that are waiting for a child call to return. (Continues in Figure 12.3.)



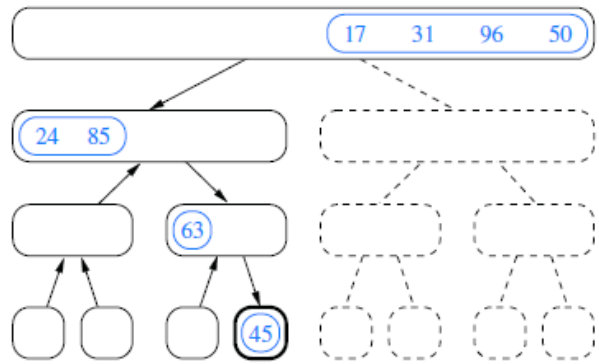
(g)



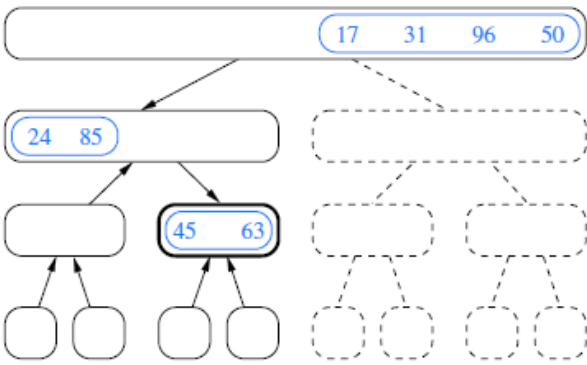
(h)



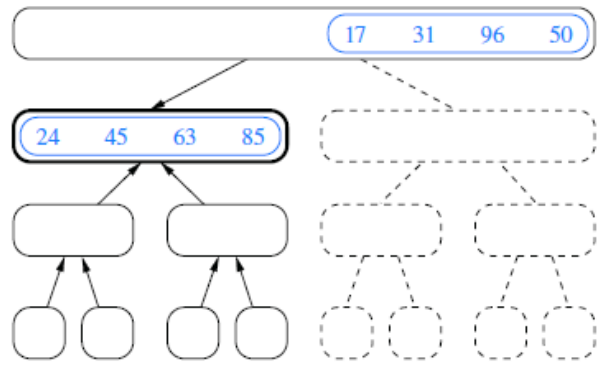
(i)



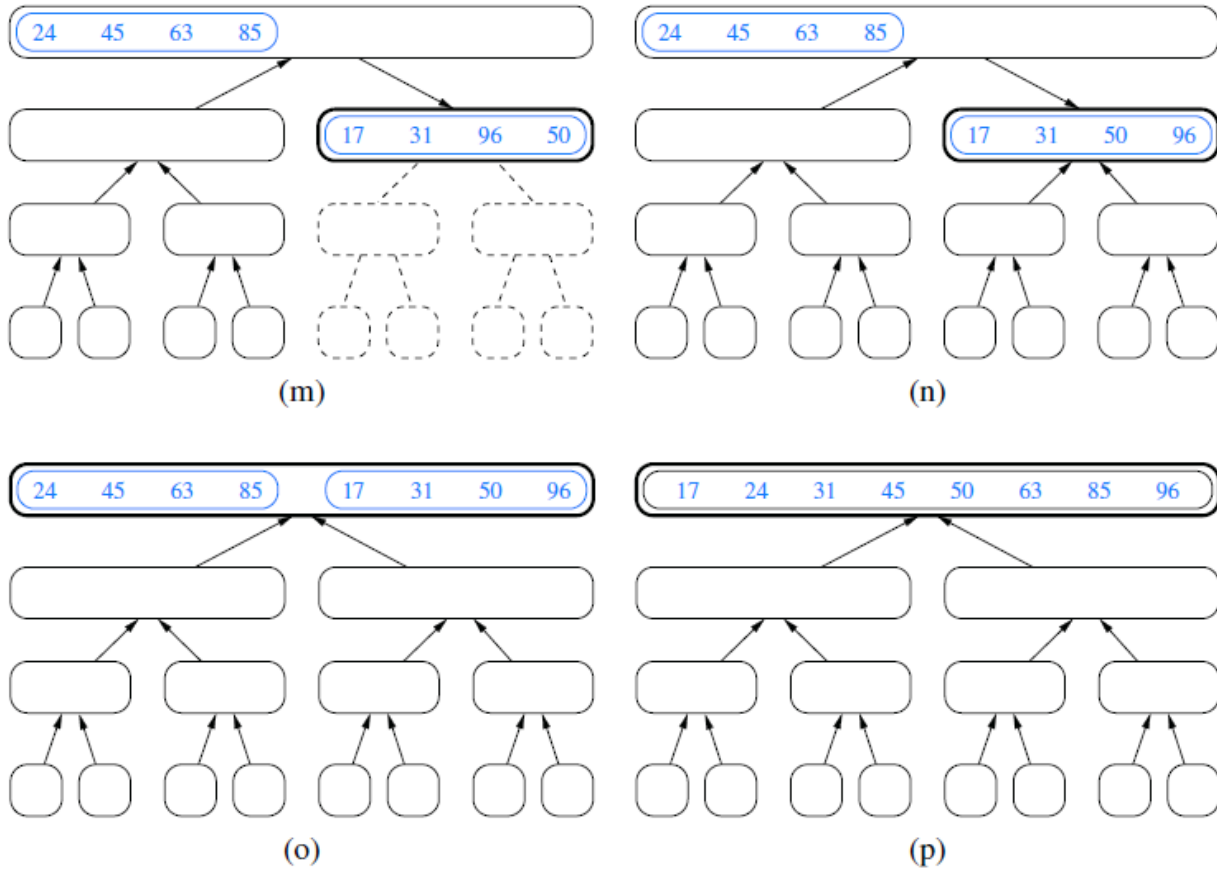
(j)



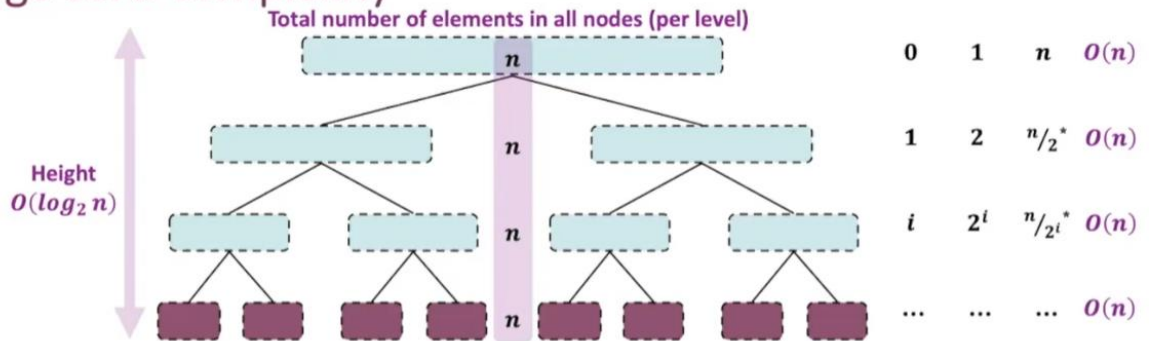
(k)



(l)



### Merge sort: complexity



The height  $h$  of the merge sort recursion tree is  $O(\log_2 n)$ .

- At each call, divide, make 2 recursive calls: recursion tree is a complete binary tree.

The total amount of work done by all the calls at level  $i$  is always  $O(n)$ :

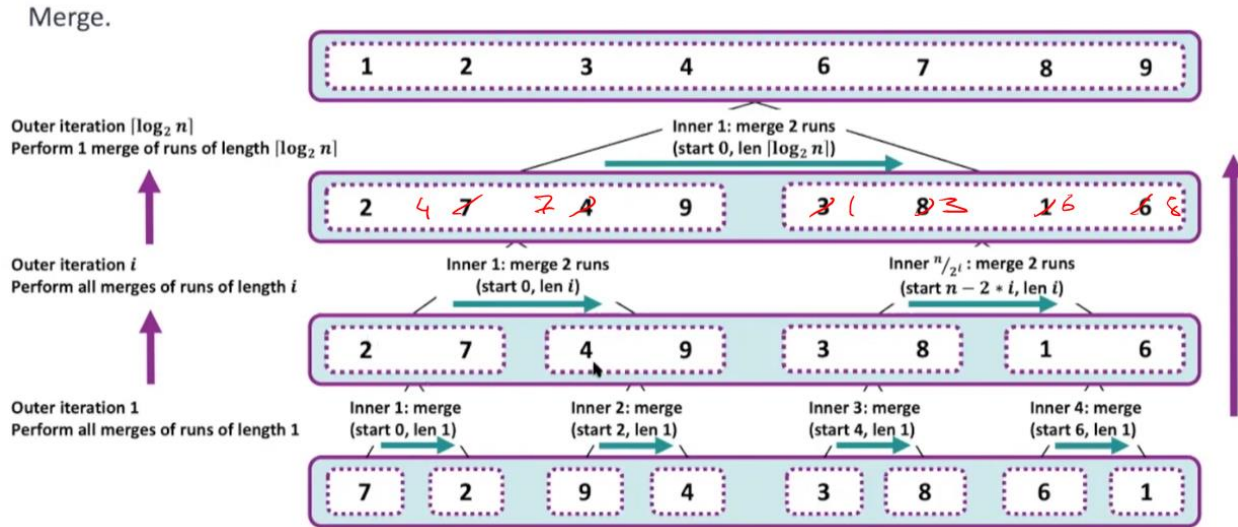
- Partition and merge  $2^i$  sequences of approximately size  $n/2^i$ .
- Make  $2^{i+1}$  recursive calls.

Total running time of merge sort is  $O(n \log_2 n)$ .

\*may differ by 1 for sequences with an odd number of elements.

The merged arrays are supposed to be sorted, therefore only the first index of each array are compared, then popped into the (sub) final sequence. Since eventually every element of the array must have been evaluated, we get  $O(n)$  for those (comparisons are  $O(1)$ ). The number of evaluations is relative to the height of the tree  $O(\log n)$ . Multiplying these yields  $O(n \log n)$

## Merge sort (bottom-up, non-recursive)



## Sorting algorithms: comparison

Algorithm	Time (worst)	Time (average)	Time (best)	Space	Properties
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Slow. In-place. For small datasets (< 1K).
Insertion	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	Generally slow. In-place. For small datasets (< 1K). Can be $O(n)$ time for nearly sorted sequences.
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)^1$	$O(1)$	Fast. In-place. For large datasets (1K — 1M). <sup>1</sup> Best $O(n)$ time only if all elements are equal
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)^2$	$O(n)^3$	Fast. Sequential data access. For huge data sets (> 1M). <sup>2</sup> Can be made to have best $O(n)$ time, but only if sequence is sorted. <sup>3</sup> Difficult to implement in-place, beyond scope of this course.

### Merge sort notes (see also Section 13.1.5 Alternative Implementations of Merge Sort):

Merge sort can be implemented both top-down and bottom-up on arrays and linked lists with same complexities. Non-recursive merge sort also runs in  $O(n \log_2 n)$ , but it is faster in practice, since it avoids overhead of recursive calls and temporary memory allocation.

Realizing a priority queue with a heap has the advantage that all the methods in the priority queue ADT run in logarithmic time or better. In general, we say that a sorting algorithm is in-place if it uses only a small amount of memory in addition to the sequence storing the objects to be sorted.

The running time of merge-sort is equal to the sum of the times spent at the nodes. the overall time spent at all the nodes of T at depth  $i$  is  $O(2^i \cdot n/2^i)$ , which is  $O(n)$ , the height of T is  $\lceil \log n \rceil$  thus the overall time complexity is  $O(n \log n)$ .

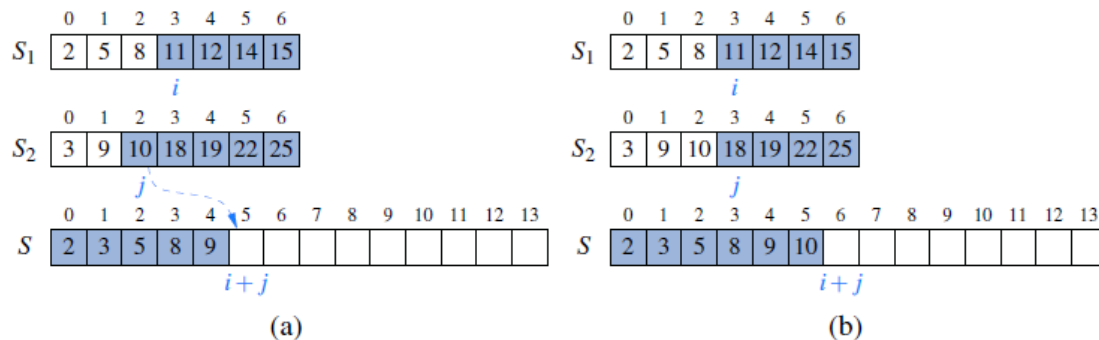
## Array-based implementation of Merge-Sort

```

1  /** Merge contents of arrays S1 and S2 into properly sized array S. */
2  public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
3      int i = 0, j = 0;
4      while (i + j < S.length) {
5          if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
6              S[i+j] = S1[i++];           // copy ith element of S1 and increment i
7          else
8              S[i+j] = S2[j++];           // copy jth element of S2 and increment j
9      }
10 }

```

**Code Fragment 12.1:** An implementation of the merge operation for a Java array.



**Figure 12.5:** A step in the merge of two sorted arrays for which  $S_2[j] < S_1[i]$ . We show the arrays before the copy step in (a) and after it in (b).

```

1  /** Merge-sort contents of array S. */
2  public static <K> void mergeSort(K[] S, Comparator<K> comp) {
3      int n = S.length;
4      if (n < 2) return;           // array is trivially sorted
5      // divide
6      int mid = n/2;
7      K[] S1 = Arrays.copyOfRange(S, 0, mid);           // copy of first half
8      K[] S2 = Arrays.copyOfRange(S, mid, n);           // copy of second half
9      // conquer (with recursion)
10     mergeSort(S1, comp);           // sort copy of first half
11     mergeSort(S2, comp);           // sort copy of second half
12     // merge results
13     merge(S1, S2, S, comp);           // merge sorted halves back into original
14 }

```

**Code Fragment 12.2:** An implementation of the recursive merge-sort algorithm for a Java array (using the merge method defined in Code Fragment 12.1).

### 12.1.4 Merge-Sort and Recurrence Equations ★

There is another way to justify that the running time of the merge-sort algorithm is  $O(n \log n)$  (Proposition 12.2). Namely, we can deal more directly with the recursive nature of the merge-sort algorithm. In this section, we will present such an analysis of the running time of merge-sort, and in so doing, introduce the mathematical concept of a *recurrence equation* (also known as *recurrence relation*).

Let the function  $t(n)$  denote the worst-case running time of merge-sort on an input sequence of size  $n$ . Since merge-sort is recursive, we can characterize function  $t(n)$  by means of an equation where the function  $t(n)$  is recursively expressed in terms of itself. In order to simplify our characterization of  $t(n)$ , let us restrict our attention to the case when  $n$  is a power of 2. (We leave the problem of showing that our asymptotic characterization still holds in the general case as an exercise.) In this case, we can specify the definition of  $t(n)$  as

$$t(n) = \begin{cases} b & \text{if } n \leq 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

An expression such as the one above is called a recurrence equation, since the function appears on both the left- and right-hand sides of the equal sign. Although such a characterization is correct and accurate, what we really desire is a big-Oh type of characterization of  $t(n)$  that does not involve the function  $t(n)$  itself. That is, we want a *closed-form* characterization of  $t(n)$ .

We can obtain a closed-form solution by applying the definition of a recurrence equation, assuming  $n$  is relatively large. For example, after one more application of the equation above, we can write a new recurrence for  $t(n)$  as

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2t(n/2^2) + 2(cn/2) + cn = 2^2t(n/2^2) + 2cn. \end{aligned}$$

If we apply the equation again, we get  $t(n) = 2^3t(n/2^3) + 3cn$ . At this point, we should see a pattern emerging, so that after applying this equation  $i$  times, we get

$$t(n) = 2^i t(n/2^i) + icn.$$

The issue that remains, then, is to determine when to stop this process. To see when to stop, recall that we switch to the closed form  $t(n) = b$  when  $n \leq 1$ , which will occur when  $2^i = n$ . In other words, this will occur when  $i = \log n$ . Making this substitution, then, yields

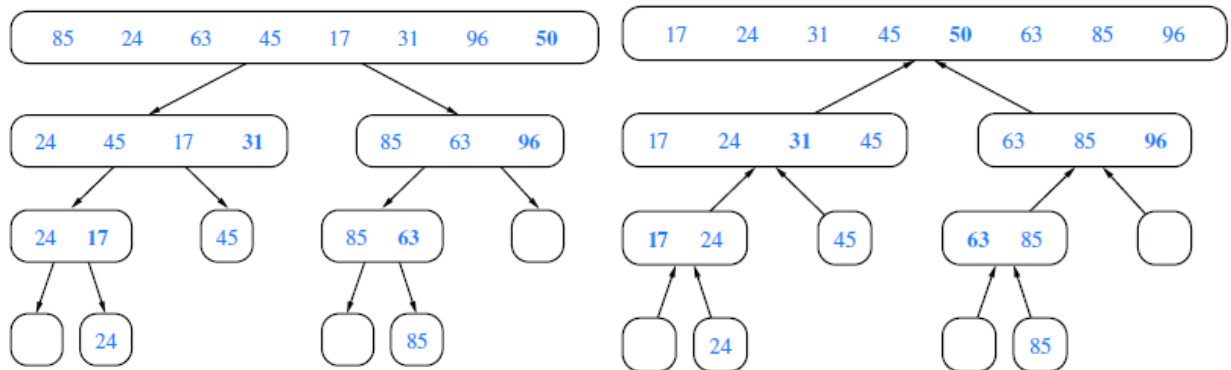
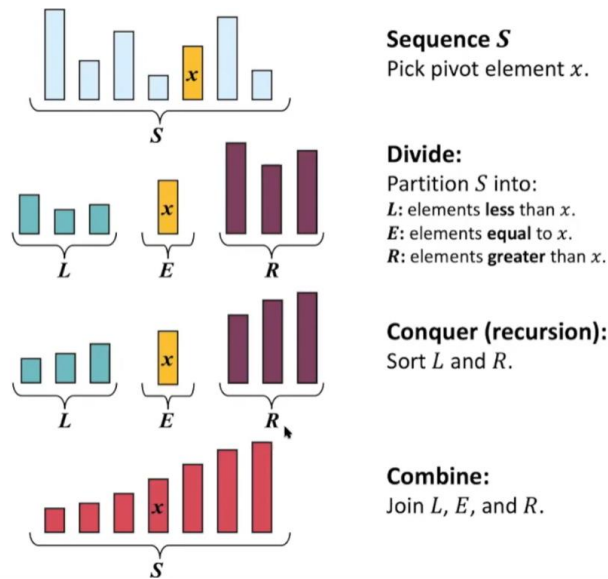
$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n. \end{aligned}$$

That is, we get an alternative justification of the fact that  $t(n)$  is  $O(n \log n)$ .

## Quick sort

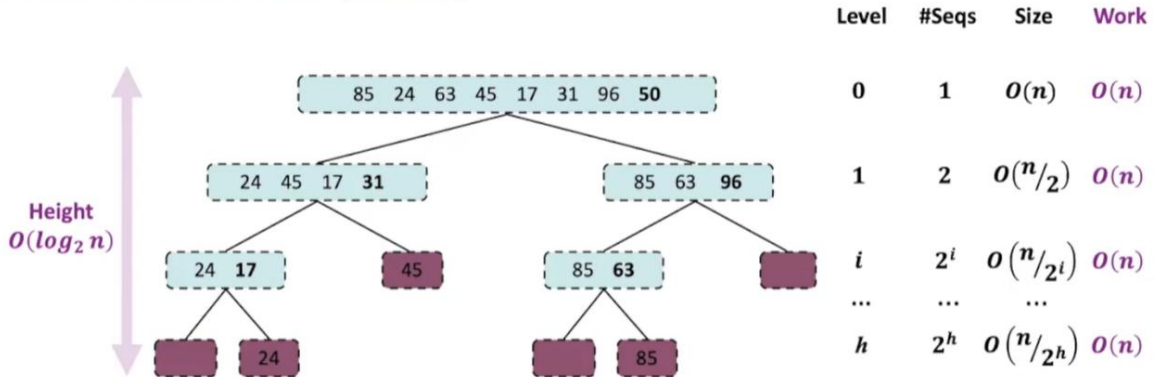
**Quick sort** uses divide-and-conquer to sort a sequence  $S$  with  $n$  elements. In quick sort the hard work is mostly done before the recursive calls.

1. **Divide:**  
**[Base case]**  
 If  $S$  has less than 2 element(s), return (nothing to do).  
**[Recurrence]**  
 If  $S$  has at least 2 element(s), select a specific element from  $S$ , called the **pivot**. E.g. choose **pivot** as the last element in  $S$  (other choices possible: e.g. middle).  
 Remove all elements from  $S$  and split them into 3 sequences:  
**Sequence  $L$ :** elements from  $S$  that are **smaller** than **pivot**.  
**Sequence  $E$ :** elements from  $S$  that are **equal** to **pivot**. (If all elements in  $S$  are unique, then only the pivot)  
**Sequence  $R$ :** elements from  $S$  that are **larger** than **pivot**.
2. **Conquer:** Recursively sort sequences  $L$  and  $R$ .
3. **Combine:** Put elements back into  $S$  as follows: **first** all elements of  $L$ , **then** elements of  $E$ , finally elements of  $R$ .





## Quick sort: time complexity



Good case: when pivot choice leads to nearly equally sized partitions.

Choice of pivot:  
last element

Then the height  $h$  of the quick sort tree is  $O(\log_2 n)$ .

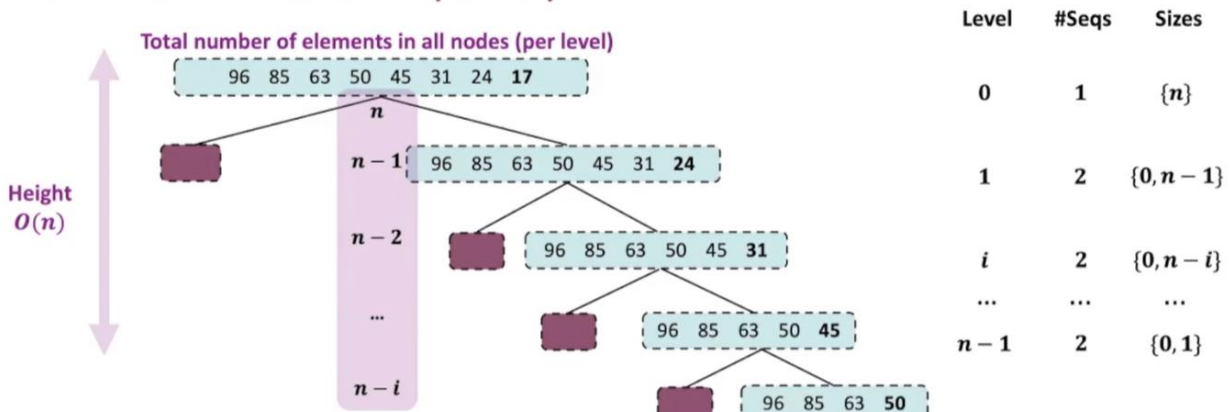
Overall amount of work done at level  $i$  is  $O(n)$ :

- Partition and combine  $2^i$  sequences of size  $n/2^i$ .
- Make  $2^{i+1}$  recursive calls.

Good case running time of quick sort is  $O(n \log_2 n)$ . Space in stack for recursion tree is  $O(\log_2 n)$ .

30

## Quick sort: time complexity



If  $S$  is sorted, last as pivot leads to unbalanced partition sizes!

Then the height of the quick sort tree is  $O(n)$ .

The work per level  $i$  is proportional to size of largest sequence,  $n-i$ .

Over all nodes:  $n + (n-1) + \dots + 2 + 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Worst-case running time of quick sort is  $O(n^2)$ . Space is  $O(n)$ .

33

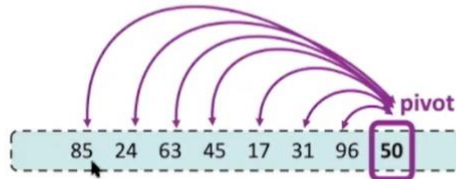
## Quick sort: randomized

To guarantee that quick sort is efficient, partitions should be balanced.  
 Deterministic choice of pivot can lead to worst-case performance for certain distributions of the input.

### Randomized quick sort

Introduces randomization when choosing the pivot:

Instead of always picking the first, last, or any other fixed element of  $S$ , we choose an element at random.



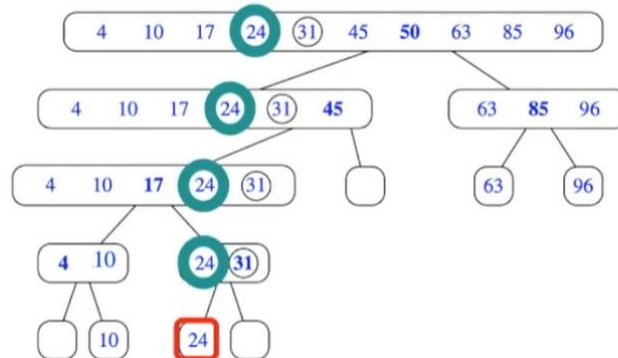
### Time complexity analysis

The running time of quick sort is proportional to the number of comparisons.

In each recursive call: **divide** compares each element to the pivot, so it can partition the input sequence into 3.

How do we calculate the total number of comparisons performed by quick sort?

## Quick sort: randomized



### Time complexity analysis (number of comparisons)

Non-pivot elements: per level, a non-pivot element is only compared once to the pivot, or not at all.

Specifically, an element  $x$  is involved as a non-pivot in only one comparison per level until it either:

- becomes the pivot (examples: elements 4, 17, 31, 45, 50, 85)
- becomes the only element (examples: 10, 24, 63, 96)

**What other examples do you see?**

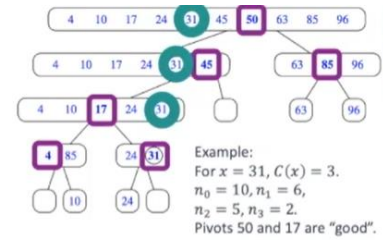
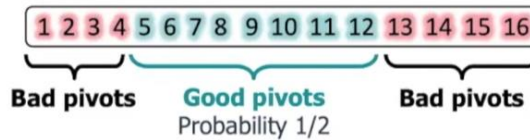
**Time complexity analysis**

The total number of comparisons is  $\sum_{x \in S} C(x)$ , where  $C(x)$  is the number of comparisons with  $x$  as a non-pivot.  $C(x)$  is path length, as  $x$  is in 1 comparison as non-pivot per level until it becomes the pivot or the only element.

**Input size (at level  $d$ ):** Let  $n_d$  be the input size for a node of such path at level  $d$  of the tree, for  $0 \leq d \leq C(x)$ . For the root,  $n_0 = n$ . For a recursive call at level  $d + 1$ , size is at most one less than the parent:  $n_{d+1} \leq n_d - 1$ .

Example:  
 For  $x = 31$ ,  $C(x) = 3$ .  
 $n_0 = 10, n_1 = 6, n_2 = 5, n_3 = 2$ .  
 Pivot choices of 50 and 17 are good.

**Quick sort**  
 randomized



**Time complexity analysis**

The runtime is proportional to the number of comparisons:  $\sum_{x \in S} C(x)$ , where  $C(x)$  is #comparisons w/  $x$  as non-pivot.

$C(x)$  is path length, as  $x$  is in one non-pivot comparison per level until it becomes the pivot or the only element.

Let  $n_d$  be the input size for node at level  $d$ , for  $0 \leq d \leq C(x)$ . Root,  $n_0 = n$ . For any call at  $d$ :  $n_{d+1} < n_d - 1$ .

Choice of pivot at level  $d$  is "good" if  $n_{d+1} \leq 3n_d/4$ .

Choice of pivot is "good" with probability at least  $1/2$ .

There are at least  $n_d/2$  elements in the input that, if chosen as pivot, leave at least  $n_d/4$  in a subproblem, and  $x$  in set w/ at most  $3n_d/4$  elements.

There are at most  $\log_{4/3} n$  good pivot choices before  $x$  is isolated. Since choice is good with probability  $1/2$ , expected number of choices to get  $\log_{4/3} n$  good choices is at most  $2 \log_{4/3} n$ . **Expected value of  $C(x)$  is  $O(\log_2 n)$ .**

The expected value of sum  $\sum_{x \in S} C(x)$  is the sum of expected values of its terms.

**Since  $C(x)$  is  $O(\log_2 n)$ , then  $\sum_{x \in S} C(x)$  is  $O(n \log_2 n)$ .**

## Quick sort (in-place)

Uses element swapping, no subsequences are created.

A subsequence of the input is represented implicitly by the range given by indices  $a$  and  $b$ .

Divide step scans the array simultaneously:

- forward, using index  $l$ , and
- backward, using index  $r$ .

When  $l$  and  $r$  cross, division is complete, and the algorithm recurs on the two subsequences.

No explicit combine needed.

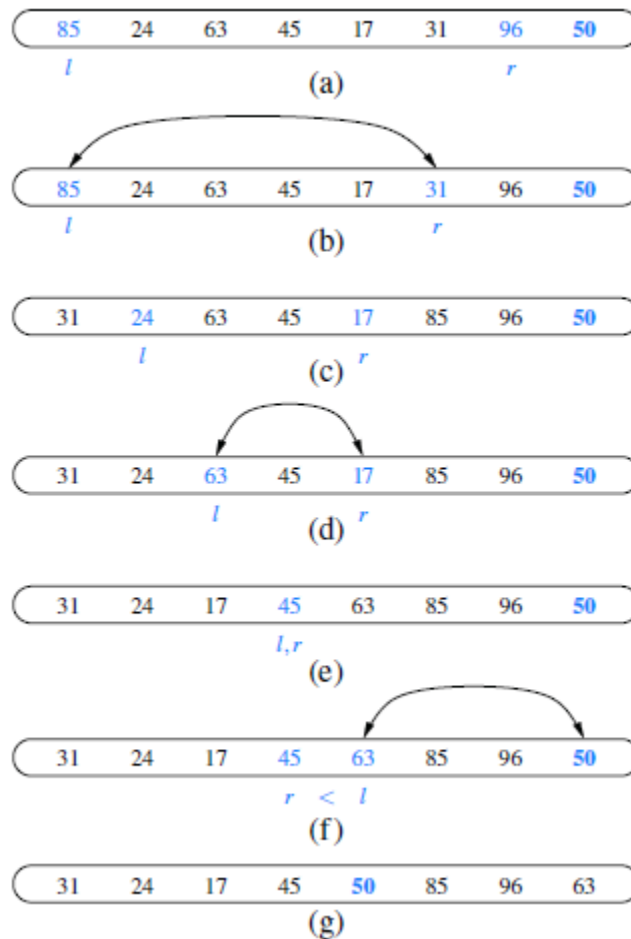
In-place quicksort:  
Java code 13.6, page 553.

```

1  /** Sort the subarray S[a..b] inclusive. */
2  private static <K> void quickSortInPlace(K[] S, Comparator<K> comp,
3                                          int a, int b) {
4      if (a >= b) return; // subarray is trivially sorted
5      int left = a;
6      int right = b-1;
7      K pivot = S[b];
8      K temp; // temp object used for swapping
9      while (left <= right) {
10         // scan until reaching value equal or larger than pivot (or right marker)
11         while (left <= right && comp.compare(S[left], pivot) < 0) left++;
12         // scan until reaching value equal or smaller than pivot (or left marker)
13         while (left <= right && comp.compare(S[right], pivot) > 0) right--;
14         if (left <= right) { // indices did not strictly cross
15             // so swap values and shrink range
16             temp = S[left]; S[left] = S[right]; S[right] = temp;
17             left++; right--;
18         }
19     }
20     // put pivot into its final place (currently marked by left index)
21     temp = S[left]; S[left] = S[b]; S[b] = temp;
22     // make recursive calls
23     quickSortInPlace(S, comp, a, left - 1);
24     quickSortInPlace(S, comp, left + 1, b);
25 }

```

46



## Sorting algorithms: comparison

Algorithm	Time	Properties
Selection sort	$O(n^2)$	In-place. Slow. OK for small input, but insertion sort is typically better.
Insertion sort	$O(n^2)$	In-place. Slow, good for small input. Can be $O(n)$ for nearly sorted input.
Quick sort	$O(n \log_2 n)^*$	In-place. Randomized. Fastest (good for large inputs). Worst-case $O(n^2)$ .
Heap sort	$O(n \log_2 n)$	In-place. Fast (good for large inputs).
Merge sort	$O(n \log_2 n)$	Sequential data access. Fast (good for huge inputs).

\*expected

**Hybrid approaches.** Sorts in library implementations usually combine multiple algorithms (e.g. insertion sort for small inputs, improved versions of quick sort or merge sort for large inputs).

Unlike merge-sort, however, the height of the quick-sort tree associated with an execution of quick-sort is linear in the worst case. This happens, for example, if the sequence consists of  $n$  distinct elements and is already sorted.

Quicksort has  $O(n^2)$  worst-case runtime and  $O(n \log n)$  average case runtime. However, it's superior to merge sort in many scenarios because many factors influence an algorithm's runtime, and, when taking them all together, quicksort wins out.

In particular, the often-quoted runtime of sorting algorithms refers to the number of comparisons or the number of swaps necessary to perform to sort the data. This is indeed a good measure of performance, especially since it's independent of the underlying hardware design. However, other things – such as locality of reference (i.e. do we read lots of elements which are probably in cache?) – also play an important role on current hardware. Quicksort in particular requires little additional space and exhibits good cache locality, and this makes it faster than merge sort in many cases.

In addition, it's very easy to avoid quicksort's worst-case run time of  $O(n^2)$  almost entirely by using an appropriate choice of the pivot – such as picking it at random (this is an excellent strategy).

In practice, many modern implementations of quicksort (in particular libstdc++'s `std::sort`) are actually [introsort](#), whose theoretical worst-case is  $O(n \log n)$ , same as merge sort. It achieves this by limiting the recursion depth, and switching to a different algorithm ([heapsort](#)) once it exceeds  $\log n$ .

share edit follow

edited May 19 '14 at 6:59

answered Sep 16 '08 at 9:14

Konrad Rudolph

468k ● 118 ● 869 ● 1120

## Week 6. lower bound & key-based sorting, selection

### Lower bound

I.e. minimum running time to sort any given sequence.

## Sorting algorithms

Algorithm	Time	Properties
Selection sort	$O(n^2)$	In-place. Slow, OK for small inputs but insertion sort is better.
Insertion sort	$O(n^2)$	In-place. Slow, good for small inputs, $O(n)$ for nearly sorted inputs.
Quick sort	$O(n \log n)^*$	In-place. Randomized. Fastest in practice (good for large inputs).
Heap sort	$O(n \log n)$	In-place. Fast (good for large inputs).
Merge sort	$O(n \log n)$	Sequential data access. Fast (good for huge inputs).

\*expected

Is there a lower bound on the time complexity of sorting?  
Can we design sorting algorithms faster than  $O(n \log n)$ ?

## Lower bound on sorting

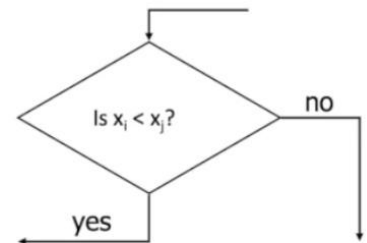
Sorting a sequence  $S = \{x_0, x_1, \dots, x_{n-1}\}$  with  $n$  elements.

All algorithms seen so far rely on **comparisons** to establish an order.

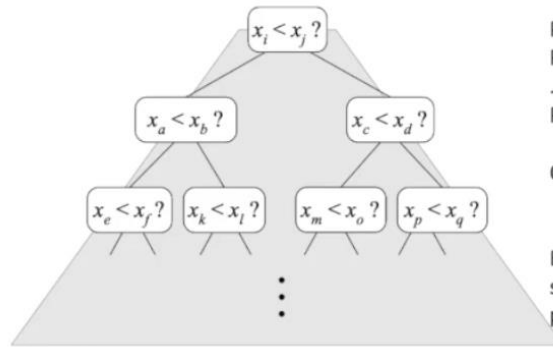
Let's only count comparisons for a **lower bound  $\Omega$**  on the **worst-case**:

- Implementation of  $S$  does not matter (only counting comparisons).
- Comparing  $x_i$  and  $x_j$  (is  $x_i < x_j$  true?), has 2 outcomes: YES, NO.
- Based on the result of a comparison, the algorithm performs some internal calculations and eventually another comparison.

We can represent a comparison-based sorting algorithm with a binary decision tree  $\mathcal{T}$ .



## Lower bound on sorting



Input:  
Any permutation of a sequence  $S$

Permutation  $P_1 = [1, 2, 3, 4, 5, 6]$   
 Permutation  $P_2 = [2, 1, 3, 4, 5, 6]$   
 ...  
 Permutation  $P_{n!} = [6, 5, 4, 3, 2, 1]$

Output: always the same (sorted  $S$ ):  
 $[1, 2, 3, 4, 5, 6]$

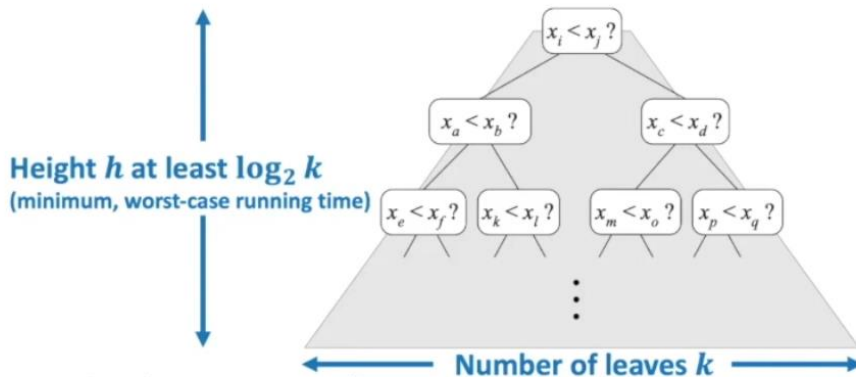
Each possible permutation leads to a specific series of comparisons in a path from the root to a leaf.

Sorting  $S = \{x_0, x_1, \dots, x_{n-1}\}$

Binary decision tree  $\mathcal{T}$  represents all comparisons made by a comparison-based sorting algorithm.

- Nodes denote comparison operations.
- Each possible run of the algorithm corresponds to a root-to-leaf path in the decision tree  $\mathcal{T}$ .

## Lower bound on sorting



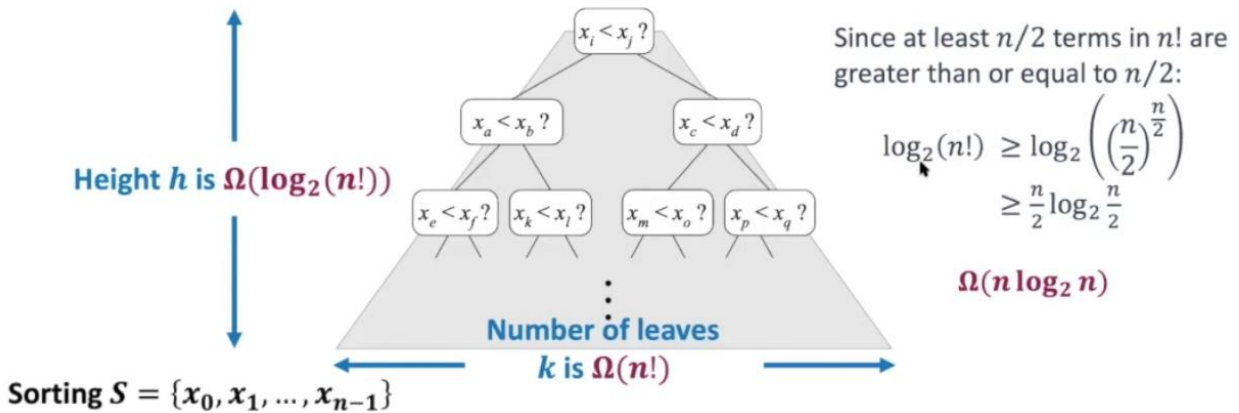
Sorting  $S = \{x_0, x_1, \dots, x_{n-1}\}$

Running time of comparison-based sorting is at least equal to the height  $h$  of tree  $\mathcal{T}$  (minimum).

Each leaf in  $\mathcal{T}$  denotes the sequence of comparisons for at most one permutation of  $S$ .

**Proof by contradiction.** If different permutations  $P_1$  and  $P_2$  are associated with the same leaf, then there are at least two elements  $x_i$  and  $x_j$  from  $S$  such that their order is different in  $P_1$  and  $P_2$ . This would mean that  $x_i$  and  $x_j$  are in the wrong order in  $P_1$  or  $P_2$ , which is not possible for a correct sorting algorithm.  $\square$

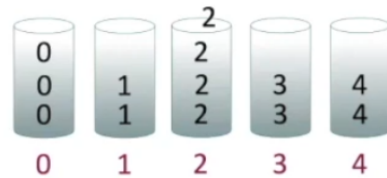
## Lower bound on sorting



Running time of comparison-based sorting is at least equal to the height  $h$  of tree  $\mathcal{T}$  (minimum).  
 Each leaf in  $\mathcal{T}$  denotes the sequence of comparisons for at most one permutation of  $S$ .  
 Each permutation of  $S$  results in a different leaf. **Number of leaves**  $n! = n(n-1)(n-2) \dots 2 \cdot 1$ .  
 Height of  $\mathcal{T}$  is at least  $\log_2(n!)$ .

## Bucket sort

4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



# Bucket sort

Chapter 13.3.2

0	0	0	1	1	2	2	2	2	2	3	3	4	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---



## Key-based sorting

Let  $S$  be a sequence of  $n$  (key, element) items.  
 Keys are (small) integers in the range  $\{0, \dots, N - 1\}$ .

### Bucket Sort:

No need for comparisons.  
 Can use key as index into an auxiliary array:  
 All entries with key  $k$  are placed in a “bucket” at index  $k$ .

### Applications.

- Sort string by first letter.
- Sort class list by group.
- Sort phone numbers by area code.
- **Subroutine in a sorting algorithm.**

input		sorted result (by section)	
name	section		
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

↑  
Keys are small integers

14

## Bucket sort

For simplicity, we only look at keys.

4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---

**Algorithm** bucketSort( $S$ ):

**Input:** Sequence  $S$  of entries w/ integer keys in range  $\{0, \dots, N - 1\}$ .

**Output:** Sequence  $S$  sorted in non-decreasing order of keys.

Let  $B$  be an array of  $N$  sequences, each initially empty.

**For** each entry  $e$  in  $S$  **do**

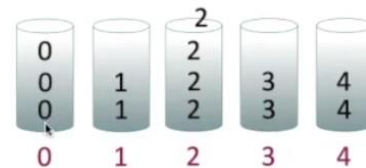
Let  $k$  be the key of entry  $e$ .

Remove entry  $e$  from  $S$  and insert it at the end of  $B[k]$ .

**For**  $i = 0$  to  $N - 1$  **do**

**For** each entry  $e$  in sequence  $B[i]$  **do**

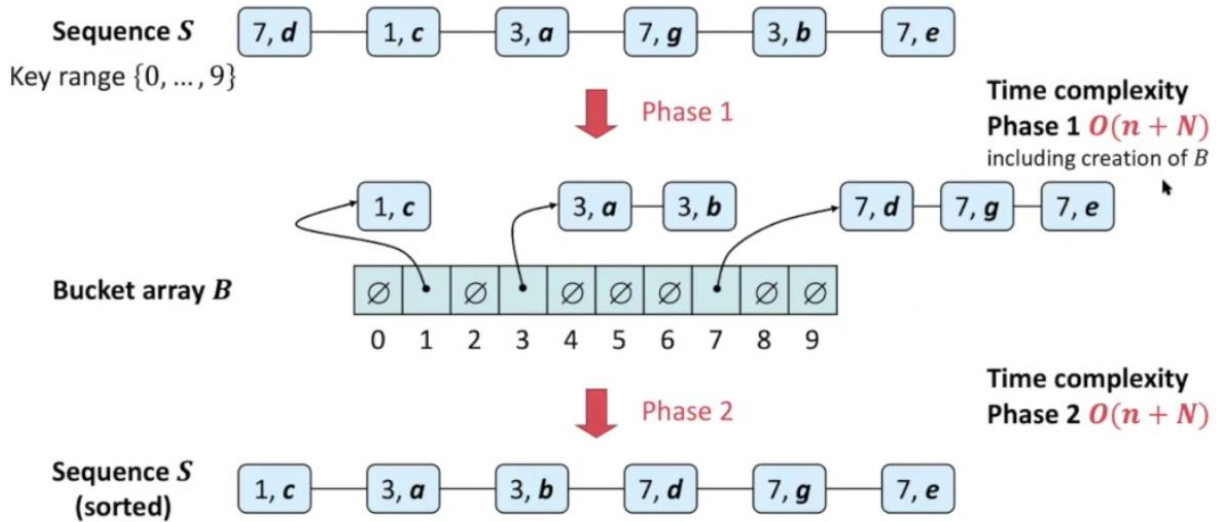
Remove entry  $e$  from  $B[i]$  and insert it at the end of  $S$ .



0	0	0	1	1	2	2	2	2	3	3	4	4
---	---	---	---	---	---	---	---	---	---	---	---	---

## Bucket sort

Sequence  $S$ :  $n$  elements.  
Keys:  $N$  possible values.



## Bucket sort

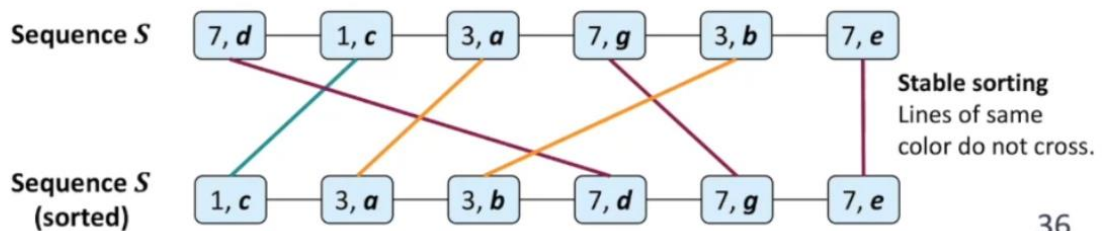
Time and space complexity:  $O(n + N)$ . Depending on implementation, space can be  $O(N)$  for buckets. Efficient when range of keys  $N$  is small compared to sequence size  $n$ : e.g.  $N$  is  $O(n)$  or  $O(n \log n)$ .

### Properties

**Key type:** keys are used as indices into an array, cannot be arbitrary objects.

**Stable sorting:** preserves the relative order of any two items with the same key.

For any two items  $(k_i, v_i)$  and  $(k_j, v_j)$  such that  $k_i = k_j$  and  $(k_i, v_i)$  precedes  $(k_j, v_j)$ , or  $i < j$ , in  $S$  before sorting, then  $(k_i, v_i)$  also precedes  $(k_j, v_j)$ , or  $i < j$ , in  $S$  after sorting.



## Stable sorting

A stable sorting algorithm preserves the relative order of items with the same or equivalent key.

### When is this relevant?

Imagine that we are sorting based on multiple criteria.

If an order has already been established so far, we don't want to break it if the item keys are equivalent according to the next criterium.

### Which of these algorithms are naturally stable?

Selection sort	yes	
Insertion sort	yes	
Quick sort	no	due to the swapping of elements relative to the pivot
Heap sort	no	due to the heap structure and operations
Merge sort	yes	
Bucket sort	yes	

Radix sorts (LSD, MSD)

## Key types

Type of key	Radix (N)	Alphabet (possible keys)
BINARY	2	01
DNA	4	ACTG
OCTAL	8	01234567
DECIMAL	10	0123456789
HEXADECIMAL	16	0123456789ABCDEF
PROTEIN	20	ACDEFGHIKLMNPQRSTVWY
LOWERCASE	26	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ ghijklmnopqrstuvwxyz
ASCII	128	<i>ASCII characters</i>
EXTENDED_ASCII	256	<i>extended ASCII characters</i>
UNICODE16	65536	<i>Unicode characters</i>

**ASCII codes**  
{65, 66, ..., 90}

**Keys: code - 65**  
{0, 1, ..., 25}

40

# Radix sort

**Composite keys:** keys in a sequence  $S$  are  $d$ -tuples of elementary keys  
 $key = (k_1, k_2, \dots, k_d)$

Examples:

Keys	Tuple description	Alphabet	Radix ( $N$ )
06 1234 5678	phone number is a 10-tuple of decimal digits	{0,1, ..., 9}	10 possible digits
AMS	airport code is a 3-tuple of capital letters	{A, B, ..., Z}	26 possible letters

## Idea of radix sort

To sort sequence  $S$  of  $n$  keys that are  $d$ -tuples of radix  $N$ :

Apply stable bucket sort sequentially, using a different elementary key of the tuple at each iteration.

**How should we process the elementary keys in the  $d$ -tuple?**

- A. Forward    AMS    ->    AMS    ->    AMS
- B. Backward    AMS    ->    AMS    ->    AMS

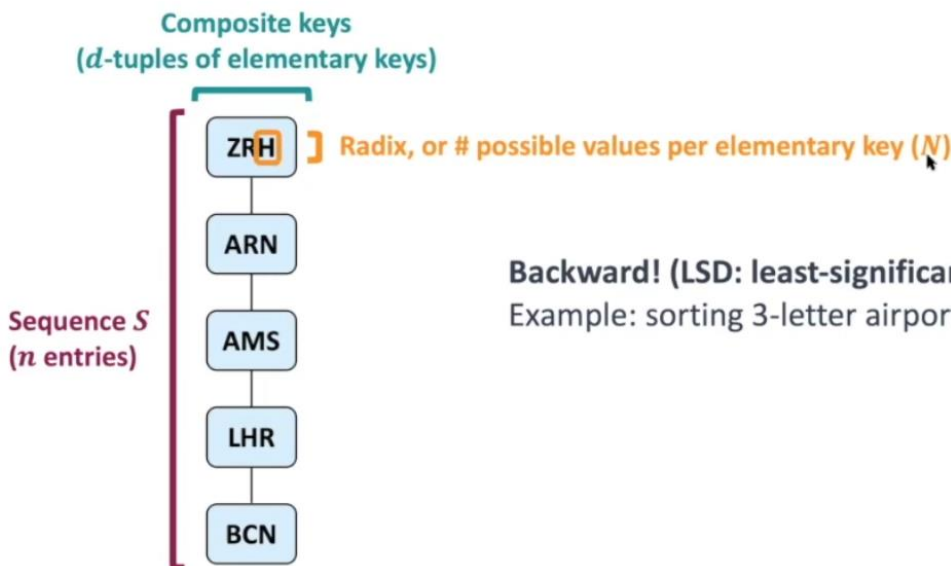
41

Because it preserves the lexicographic order.

## Radix sort (LSD)

Key length     $d = 3$   
 Radix         $N = 26$   
 #Elements     $n = 5$

$n \gg N$  in real applications



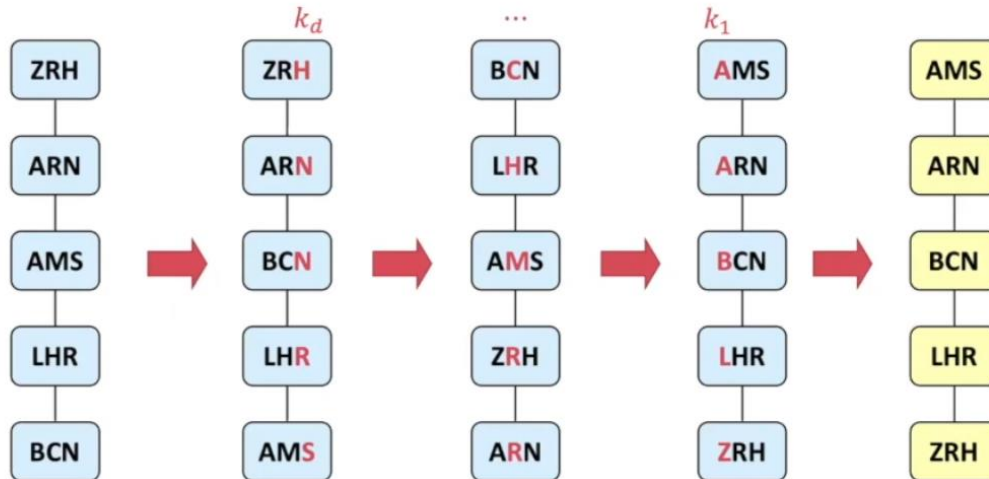
42

## Radix sort (LSD)

Bucket sort applied  $d$  times:  $O(d(n + N))$

$d = 3$   
 $N = 26$   
 $n = 5$

$n \gg N$  in real applications



48

## Radix sort (LSD)

Generally works with keys of equal-length.

Strings of equal-length (e.g. airport codes).

Integers of equal-length (e.g. integers using their bit representation, telephone numbers).

More:

Can be made to work with variable-length keys that align right.

But needs to either process the keys per length or use padding with zeros on the left.

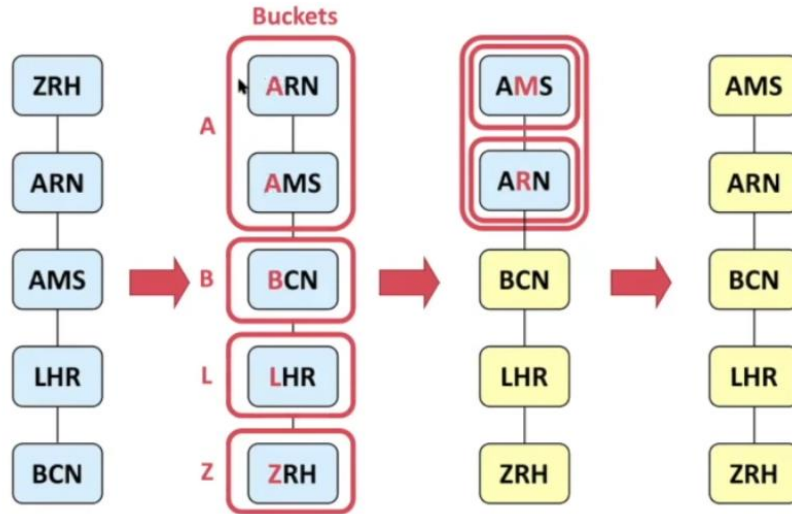
Example: multi-digit integers

0040  
 0002  
 0315  
 0008

Not really suited for variable-length keys that use lexicographic order (e.g. strings in general).

## Radix sort (MSD)

It works, if we apply bucket sort recursively within each bucket formed in the previous iteration!



51

## Radix sort (MSD)

**Works with keys of different length.**

General strings.

Integers (e.g. because it can make use of their bit representation).

pig  
lion  
parrot  
owl  
zebra

lion  
owl  
pig  
parrot  
zebra

lion  
owl  
parrot  
pig  
zebra

Three strings sorted after one character, 2 sorted after two characters.

Can be faster than LSD, since it may not need to process all elementary keys.

Overhead due to recursive calls.

**Improvements: check 3-way radix quicksort.**

## Sorting algorithms

	Stable	In-place	Best-case time	Worst-case time	Space	Application
Selection sort	Yes	Yes	$O(n^2)$	$O(n^2)$	$O(1)$	Small sequences, but insertion is better.
Insertion sort	Yes	Yes	$O(n + m)$	$O(n^2)$	$O(1)$	Small sequences. It's $O(n)$ for nearly sorted (since # inversions $m$ is small).
Heap sort	No	Yes	$O(n \log n)$	$O(n \log n)$	$O(1)$	Small to mid-size sequences that fit into memory. Slower than quick, merge sort.
Quick sort	No	Yes	$O(n \log n)^*$	$O(n^2)$	$O(\log n)^*$	General purpose if space is tight and stability is not a concern.
Merge sort	Yes	No	$O(n \log n)$	$O(n \log n)$	$O(n)$	General purpose for very large data that doesn't fit into main memory.
Radix sort	Yes	No	$O(d(n + N))$	$O(d(n + N))$	$O(n + N)$	Integers, strings, other $d$ -tuples. If $d(n + N) \ll n \log n$ , radix are faster than comparison-based algorithms.

Values for stable and in-place columns based on the most common implementations of the algorithms. Other variants may be possible.

\*also expected

54

Randomized (in-place) quick select

## Selection problem

### Selection problem.

Select the  $k^{\text{th}}$  smallest element from an unsorted collection of  $n$  comparable elements.  
Also called order statistics (selecting an element with a given rank).

For instance:

1 <sup>st</sup> smallest	(minimum)	rank 1
...		
$\lfloor n/2 \rfloor^{\text{th}}$ smallest	(median)	rank $\lfloor n/2 \rfloor$
...		
$n^{\text{th}}$ smallest	(maximum)	rank $n$

Solutions? Complexity?

## Selection problem

**Problem.** Select the  $k^{\text{th}}$  smallest element from an unsorted collection of  $n$  comparable elements.

**Solution 1.** Sort collection, pick element at index  $k - 1$  in sorted sequence. Takes  $O(n \log n)$  time.

**Solution 2.** Build heap with  $n$  elements in  $O(n)$  time, remove  $k$  elements. Takes  $O(n + k \log n)$  time.

Slow, since we can solve the selection problem for  $k = 1$ ,  $k = n$ , and other values of  $k$  in  $O(n)$  time.

Can we achieve  $O(n)$  running time for all values of  $k$ ?

For instance, finding the median, where  $k = \lfloor n/2 \rfloor$ .

Algorithm that uses design pattern **prune-and-search** or **decrease-and-conquer**:

Prune away a fraction of the  $n$  elements.

Recursively solve the smaller problem.

Find solution for base case using a brute-force method.

59

## Randomized quick-select (in-place)

```
public static int selectInPlace(int[] array, int k) {
    return selectInPlace(array, 0, array.length-1, k-1);
}

private static int selectInPlace(int[] array, int left, int right, int i) { //i index of k-th
    if (left == right)
        return array[left]; // if there's only one element, return that element

    // select random pivot index between left and right (same as quick sort)
    int pivotIdx = randomPivot(left, right);
    // swap pivot with last element, partition, assign updated pivot index (same as quick sort)
    pivotIdx = partition(array, left, right, pivotIdx);

    if (i == pivotIdx)
        return array[i]; // k-th element is the pivot
    else if (i < pivotIdx)
        return selectInPlace(array, left, pivotIdx - 1, i); // k-th element is <= pivot
    else
        return selectInPlace(array, pivotIdx + 1, right, i); // k-th element is >= pivot
}
```

61



## Randomized quick-select

**Algorithm** quickSelect( $S, k$ )

**Input:** Sequence  $S$  of  $n$  elements, and an integer  $k \in \{1, \dots, n\}$

**Output:** The  $k^{\text{th}}$  smallest element of  $S$

**If**  $n == 1$  **then**

**return** the (first) element of  $S$

Pick a random (pivot) element  $x$  of  $S$  and divide  $S$  into three sequences:

$L$ , storing the elements in  $S$  less than  $x$

$E$ , storing the elements in  $S$  equal to  $x$

$R$ , storing the elements in  $S$  greater than  $x$

**If**  $k \leq |L|$  **then**

**return** quickSelect( $L, k$ )

**else if**  $k \leq |L| + |E|$  **then**

**return**  $x$

**else**

**return** quickSelect( $R, k - |L| - |E|$ )

## Randomized quick-select (in-place)

Input array [7, 10, 4, 3, 20, 15] and  $k = 4$   
 Call: selectInPlace(array, 0, 5, 3)  
 Expected output: 10.

7 10 4 3 20 15

3 4 7 15 20 10  
 $p = 2$

3 4 7 15 20 10

3 4 7 10 20 15  
 $p = 3$

**Remember that  $i = 3$ .**

Pivot choice: 7 (swap with 15, partition)

Pivot index  $p$  is 2.

Is  $i = p$ ? No, so  $k^{\text{th}}$  is not the pivot.

Is  $i < p$ ? No, so  $k^{\text{th}}$  is not in left subarray.

Must be in right then, recur on right subarray.

Pivot choice: 10 (already last, partition)

Pivot index  $p$ : 3

Is  $i = p$  true? Yes, so  $k^{\text{th}}$  is the pivot. Return 10.

62

## Randomized quick-select

Quick-select on sequence of  $n$  elements.

### Best case:

Pivot choice always results in equally sized partitions (sizes of approximately  $1/2$  each).

Only recurs on one of the partitions, so number of comparisons is

$$n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{\lceil \log_2 n \rceil}} = n \sum_{i=0}^{\lceil \log_2 n \rceil} (1/2)^i$$

Since  $\sum_{i=0}^{\lceil \log_2 n \rceil} (1/2)^i$  is a geometric sum with base  $1/2$ , a positive number  $< 1$ , time complexity is  $O(n)$ .

Space is  $O(\log_2 n)$  if in-place, since we need space for the stack frames (given by height of recursion tree).

### Worst-case:

Pivot choice always results in one partition of size 0, and one partition of size  $n - 1$ .

Number of comparisons is  $n + (n - 1) + (n - 2) + \dots + 1$ . Gauss' sum, so time complexity is  $O(n^2)$ .

Space is  $O(n)$ , since we need space for the stack frames and height of recursion tree is proportional to  $n$ .

Expected given random pivot choice?

63

## Randomized quick-select

**Time expected:**  $O(n)$

**Time worst:**  $O(n^2)$

**Space expected:**  $O(\log n)$

**Space worst:**  $O(n)$

Random pivot choice at each recursive call results in:

- "good" partitions (one of size at least  $1/4$ , other of size at most  $3/4$ )
- with probability at least  $1/2$

For the upper bound (big-Oh), we recur on the largest partition (size at most  $3/4$ ).

Let  $t(n)$  be the running time of randomized quick-select on sequence of size  $n$  (total time for all recursive calls).

Running time  $t(n)$  depends on partition sizes at each recursive call resulting from random pivot choices.

So  $t(n)$  is a random variable, and we want to bound its expected value  $E(t(n))$ .

Say  $g(n)$  is number of consecutive recursive calls we make before we get a good one (including the current call).

The recurrence equation is  $t(n) \leq bn \cdot g(n) + t(3n/4)$ , where  $b \geq 1$  is a constant

The expectation for  $n > 1$  is  $E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4))$

A recursive call is good w/ prob.  $\geq 1/2$ , independent of previous calls being good: expected value  $E(g(n)) \leq 2$ .

If  $T(n)$  is shorthand for  $E(t(n))$ , for  $n > 1$   $T(n) \leq T(3n/4) + 2bn$

After unfolding  $T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i$  thus expected  $O(n)$  time

66



## Application: Counting Word Frequencies

As a case study for using a map, consider the problem of counting the number of occurrences of words in a document. This is a standard task when performing a statistical analysis of a document, for example, when categorizing an email or news article. A map is an ideal data structure to use here, for we can use words as keys and word counts as values.

1. We begin with an empty map, mapping words to their integer frequencies.
2. We first scan through the input, considering adjacent alphabetic characters to be words,
3. Which we then convert to lowercase.
4. For each word found, we attempt to retrieve its current frequency from the map using the `get` method, with a yet unseen word having frequency zero.
5. We then (re)set its frequency to be one more to reflect the current occurrence of the word.
6. After processing the entire input, we loop through the `entrySet()` of the map to determine which word has the most occurrences

```

1  /** A program that counts words in a document, printing the most frequent. */
2  public class WordCount {
3      public static void main(String[] args) {
4          Map<String,Integer> freq = new ChainHashMap<>(); // or any concrete map
5          // scan input for words, using all nonletters as delimiters
6          Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
7          while (doc.hasNext()) {
8              String word = doc.next().toLowerCase(); // convert next word to lowercase
9              Integer count = freq.get(word); // get the previous count for this word
10             if (count == null)
11                 count = 0; // if not in map, previous count is zero
12             freq.put(word, 1 + count); // (re)assign new count for this word
13         }
14         int maxCount = 0;
15         String maxWord = "no word";
16         for (Entry<String,Integer> ent : freq.entrySet()) // find max-count word
17             if (ent.getValue() > maxCount) {
18                 maxWord = ent.getKey();
19                 maxCount = ent.getValue();
20             }
21         System.out.print("The most frequent word is " + maxWord);
22         System.out.println(" with " + maxCount + " occurrences.");
23     }
24 }

```

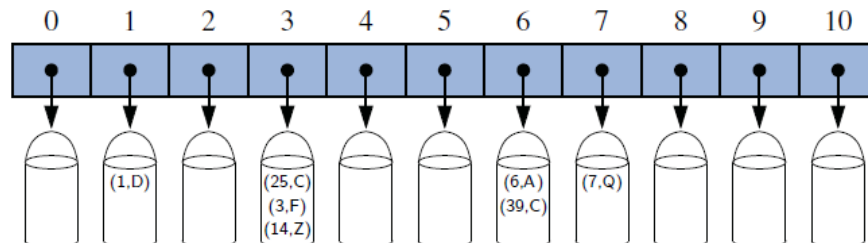
## Hash Tables

One of the most efficient data structures for implementing a map, and the one that is used most in practice. This structure is known as a hash table.

Intuitively, a map  $M$  supports the abstraction of using keys as “addresses” that help locate an entry.

The novel concept for a hash table is the use of a hash function to map general keys to corresponding indices in a table. Ideally, keys will be well distributed in the range from 0 to  $N - 1$  by a hash function, but in practice there may be two or more distinct keys that get mapped to the same index.

As a result, we will conceptualize our table as a bucket array, as shown in Figure 10.4, in which each bucket may manage a collection of entries that are sent to a specific index by the hash function.



**Figure 10.4:** A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

## Hash functions and has codes

The goal of a hash function,  $h$ , is to map each key  $k$  to an integer in the range  $[0, N - 1]$ , where  $N$  is the capacity of the bucket array for a hash table. Equipped with such a hash function,  $h$ , the main idea of this approach is to use the hash function value,  $h(k)$ , as an index into our bucket array,  $A$ , instead of the key  $k$  (which may not be appropriate for direct use as an index).

We say that a hash function is “good” if it maps the keys in our map so as to sufficiently minimize collisions (duplicates).

Types of has chodes: (be mindful that overflows may occur in most of them)

- **Treating the Bit Representation as an Integer:** for base types byte, short, int, and char, we can achieve a good hash code simply by casting a value to int. A better approach is to combine in some way the high-order and low-order portions of a 64-bit key to form a 32-bit hash code, which takes all the original bits into consideration
- **Polynomial Hash Codes:** For strings such as "stop", "tops", "pots", and "spot" a better hash code should somehow take into consideration the **positions of the xi's**, with a polynomial function:

$$x_0a^{n-1} + x_1a^{n-2} + \dots + x_{n-2}a + x_{n-1} \quad \text{or} \quad x_{n-1} + a(x_{n-2} + a(x_{n-3} + \dots + a(x_2 + a(x_1 + ax_0)) \dots)).$$

33, 37, 39, and 41 are particularly good choices for a when working with character strings that are English words, these produced fewer than 7 collisions in each case.

- **Cyclic-Shift Hash Codes:** A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits. In Java, a cyclic shift of bits can be accomplished through careful use of the bitwise shift operators.

```
static int hashCode(String s) {
    int h=0;
    for (int i=0; i<s.length(); i++) {
        h = (h << 5) | (h >>> 27);           // 5-bit cyclic shift of the running sum
        h += (int) s.charAt(i);              // add in next character
    }
    return h;
}
```

Our choice of a 5-bit shift is justified by experiments run on a list of just over 230,000 English words, comparing the number of collisions for various shift amounts

- **Hash Codes in Java:** The Object class, which serves as an ancestor of all object types, includes a default hashCode( ) method that returns a 32-bit integer of type int, which serves as an object’s hash code. Here any two objects that are viewed as “equal” to each other have the same hash code.

```
1 public int hashCode() {
2     int h = 0;
3     for (Node walk=head; walk != null; walk = walk.getNext()) {
4         h ^= walk.getElement().hashCode(); // bitwise exclusive-or with element's code
5         h = (h << 5) | (h >>> 27);       // 5-bit cyclic shift of composite code
6     }
7     return h;
8 }
```

**Code Fragment 10.6:** A robust implementation of the hashCode method for the SinglyLinkedList class from Chapter 3.

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

## Compression functions

The hash code for a key  $k$  will typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the capacity of the bucket array. Approaches:

- **The Division Method:** maps an integer  $i$  to  $i \bmod N$ , where  $N$  is the size of the bucket array. *if we insert keys with hash codes  $\{200, 205, 210, 215, 220, \dots, 600\}$  into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions. If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is  $1/N$ . Choosing  $N$  to be a prime number is not always enough, however, for if there is a repeated pattern of hash codes of the form  $pN + q$  for several different  $p$ 's, then there will still be collisions.*
- The MAD Method:

A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys, is the *Multiply-Add-and-Divide* (or “MAD”) method. This method maps an integer  $i$  to

$$[(ai + b) \bmod p] \bmod N,$$

where  $N$  is the size of the bucket array,  $p$  is a prime number larger than  $N$ , and  $a$  and  $b$  are integers chosen at random from the interval  $[0, p - 1]$ , with  $a > 0$ . This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a “good” hash function, that is, one such that the probability any two different keys collide is  $1/N$ . This good behavior would be the same as we would have if these keys were “thrown” into  $A$  uniformly at random.

## Hash functions



If we have the worst hash code possible, can we fix this with a good compression function?

- Yes, we can still have a good hash function
- Partially, we can have a decent hash function, but it will not be perfect
- No, but we can make it a little better
- No, not at all

## Hash codes



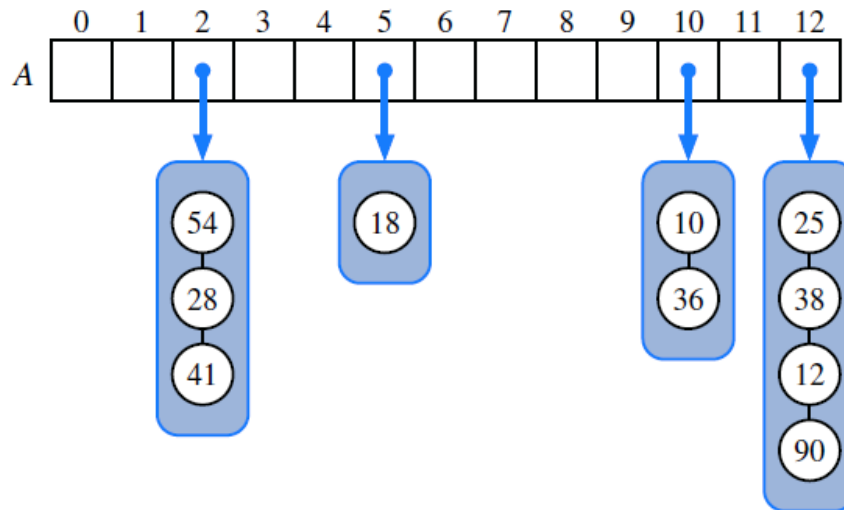
What is true about hash codes?

- If `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- If `x.hashCode() == y.hashCode()`, then `x.equals(y)`
- Both A & B are true
- Neither A nor B is true problem


## Collision-Handling Schemes

## Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket  $A[j]$  store its own secondary container, holding all entries  $(k, v)$  such that  $h(k) = j$ . A natural choice for the secondary container is a small map instance implemented using an unordered list, as described in Section 10.1.4. This *collision resolution* rule is known as *separate chaining*, and is illustrated in Figure 10.6.



## Separate Chaining

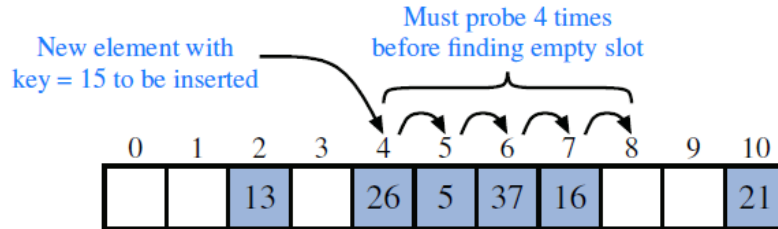
 Given an array-based map using hashing, with Separate Chaining as collision-handling scheme, how large should the array be to ensure a time complexity of  $\mathcal{O}(1)$  for the core operations (choose the last option from the correct answers)?

- (a)  $N > n$
- (b)  $N \geq n$
- (c)  $N$  is  $\Omega(n)$
- (d)  $N$  is prime
- (e)  $N$  is  $\Omega(1)$

## Open Addressing

This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to properly handle collisions.

**Linear Probing and Its Variants:** if we try to insert an entry  $(k, v)$  into a bucket  $A[j]$  that is already occupied, where  $j = h(k)$ , then we next try  $A[(j+1) \bmod N]$ . If  $A[(j+1) \bmod N]$  is also occupied, then we try  $A[(j+2) \bmod N]$ , and so on, until we find an empty bucket that can accept the new entry.




**Figure 10.7:** Insertion into a hash table with integer keys using linear probing. The hash function is  $h(k) = k \bmod 11$ . Values associated with keys are not shown.

**Quadratic probing:** iteratively tries the buckets  $A[(h(k) + f(i)) \bmod N]$ , for  $i = 0, 1, 2, \dots$ , where  $f(i) = i^2$ . It has *secondary clustering*, where the set of filled array cells still has a nonuniform pattern.

**Double hashing:** we choose a secondary hash function,  $h'$ , and if  $h$  maps some key  $k$  to a bucket  $A[h(k)]$  that is already occupied, then we iteratively try the buckets  $A[(h(k) + f(i)) \bmod N]$  next, for  $i = 1, 2, 3, \dots$ , where  $f(i) = i \cdot h'(k)$ .

## Open Addressing

 Which core function(s) of a map do(es) not work anymore if we do not replace a deleted entry with a "defunct" object?

- (a) get
- (b) put
- (c) remove
- (d) get and put
- (e) **get and remove**
- (f) put and remove
- (g) get, put and remove

Time complexity

Method	Unsorted List	Hash Table	
		expected	worst case
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$	$O(n)$

**Table 10.2:** Comparison of the running times of the methods of a map realized by means of an unsorted list (as in Section 10.1.4) or a hash table. We let  $n$  denote the number of entries in the map, and we assume that the bucket array supporting the hash table is maintained such that its capacity is proportional to the number of entries in the map.



## Sorted Maps

allows a user to look up the value associated with a given key, but the search for that key is a form known as an **exact search**. In this section, we will introduce an extension known as the **sorted map** ADT that includes all behaviors of the standard map, plus the following:

- firstEntry():** Returns the entry with smallest key value (or null, if the map is empty).
- lastEntry():** Returns the entry with largest key value (or null, if the map is empty).
- ceilingEntry(k):** Returns the entry with the least key value greater than or equal to  $k$  (or null, if no such entry exists).
- floorEntry(k):** Returns the entry with the greatest key value less than or equal to  $k$  (or null, if no such entry exists).
- lowerEntry(k):** Returns the entry with the greatest key value strictly less than  $k$  (or null, if no such entry exists).
- higherEntry(k):** Returns the entry with the least key value strictly greater than  $k$  (or null if no such entry exists).
- subMap( $k_1, k_2$ ):** Returns an iteration of all entries with key greater than or equal to  $k_1$ , but strictly less than  $k_2$ .

## Sorted Search Tables

We store the map's entries in an array list  $A$  so that they are in increasing order of their keys.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

**Figure 10.8:** Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

The sorted search table has a space requirement that is  $O(n)$ . The primary advantage of this representation, and our reason for insisting that  $A$  be array-based, is that it allows us to use the **binary search** algorithm for a variety of efficient operations.

while performing a binary search, we can instead return the index at or near where a target might be found. During a successful search, the standard implementation determines the precise index at which the target is found. During an unsuccessful search, although the target is not found, the algorithm will effectively determine a pair of indices designating elements of the collection that are just less than or just greater than the missing target.



What are the time complexities of the following methods for each of the map implementations?

Map( $K, V$ )	Unsorted arraylist	Sorted arraylist	Arrays based using hashing
get( $K$ key)	$O(n)$	$O(\log(n))$	$O(1)$ (expected)
put( $K$ key, $V$ value)	$O(n)$	$O(n)$	$O(1)$ (expected)
remove( $K$ key)	$O(n)$	$O(n)$	$O(1)$ (expected)

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$ ; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where $s$ items are reported
entrySet, keySet, values	$O(n)$

**Table 10.3:** Performance of a sorted map, as implemented with SortedTableMap. We use  $n$  to denote the number of items in the map at the time the operation is performed. The space requirement is  $O(n)$ .

## Sets, Multisets, and Multimaps

- ▶ A **set** is an unordered collection of elements, without duplicates, that typically supports efficient membership tests
- ▶ A **multiset** (also known as a bag) is a set-like container that allows duplicates.
- ▶ A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values

### 10.5.1 The Set ADT

The Java Collections Framework defines the `java.util.Set` interface, which includes the following fundamental methods:

`add( $e$ )`: Adds the element  $e$  to  $S$  (if not already present).

`remove( $e$ )`: Removes the element  $e$  from  $S$  (if it is present).

`contains( $e$ )`: Returns whether  $e$  is an element of  $S$ .

`iterator()`: Returns an iterator of the elements of  $S$ .

There is also support for the traditional mathematical set operations of *union*, *intersection*, and *subtraction* of two sets  $S$  and  $T$ :

$$S \cup T = \{e: e \text{ is in } S \text{ or } e \text{ is in } T\},$$

$$S \cap T = \{e: e \text{ is in } S \text{ and } e \text{ is in } T\},$$

$$S - T = \{e: e \text{ is in } S \text{ and } e \text{ is not in } T\}.$$

In the `java.util.Set` interface, these operations are provided through the following methods, if executed on a set  $S$ :

`addAll( $T$ )`: Updates  $S$  to also include all elements of set  $T$ , effectively replacing  $S$  by  $S \cup T$ .

`retainAll( $T$ )`: Updates  $S$  so that it only keeps those elements that are also elements of set  $T$ , effectively replacing  $S$  by  $S \cap T$ .

`removeAll( $T$ )`: Updates  $S$  by removing any of its elements that also occur in set  $T$ , effectively replacing  $S$  by  $S - T$ .

## Sorted Sets

For the standard set abstraction, there is no explicit notion of keys being ordered; all that is assumed is that the equals method can detect equivalent elements.

If, however, elements come from a Comparable class (or a suitable Comparator object is provided), we can extend the notion of a set to define the *sorted set ADT*, including the following additional methods:

- `first()`: Returns the smallest element in  $S$ .
- `last()`: Returns the largest element in  $S$ .
- `ceiling( $e$ )`: Returns the smallest element greater than or equal to  $e$ .
- `floor( $e$ )`: Returns the largest element less than or equal to  $e$ .
- `lower( $e$ )`: Returns the largest element strictly less than  $e$ .
- `higher( $e$ )`: Returns the smallest element strictly greater than  $e$ .
- `subSet( $e_1, e_2$ )`: Returns an iteration of all elements greater than or equal to  $e_1$ , but strictly less than  $e_2$ .
- `pollFirst()`: Returns and removes the smallest element in  $S$ .
- `pollLast()`: Returns and removes the largest element in  $S$ .

In the Java Collection Framework, the above methods are included in a combination of the `java.util.SortedSet` and `java.util.NavigableSet` interfaces.

## Implementing Sets

Although a set is a completely different abstraction than a map, the techniques used to implement the two can be quite similar. In effect, a set is simply a map in which (unique) keys do not have associated values.

Therefore, any data structure used to implement a map can be modified to implement the set ADT with similar performance guarantees. As a trivial adaption of a map, each set element can be stored as a key, and the null reference can be stored as an (irrelevant) value. Of course, such an implementation is unnecessarily wasteful; a more efficient set implementation should abandon the Entry composite and store set elements directly in a data structure.

The Java Collections Framework includes the following set implementations, mirroring similar data structures used for maps:

- `java.util.HashSet` provides an implementation of the (unordered) set ADT with a hash table.
- `java.util.concurrent.ConcurrentSkipListSet` provides an implementation of the sorted set ADT using a skip list.
- `java.util.TreeSet` provides an implementation of the sorted set ADT using a balanced search tree. (Search trees are the focus of Chapter 11.)

## Multiset

The Multiset interface should include the following behaviors:

- `add(e):` Adds a single occurrences of *e* to the multiset.
- `contains(e):` Returns true if the multiset contains an element equal to *e*.
- `count(e):` Returns the number of occurrences of *e* in the multiset.
- `remove(e):` Removes a single occurrence of *e* from the multiset.
- `remove(e, n):` Removes *n* occurrences of *e* from the multiset.
- `size():` Returns the number of elements of the multiset (including duplicates).
- `iterator():` Returns an iteration of all elements of the multiset (repeating those with multiplicity greater than one).

The multiset ADT also includes the notion of an immutable `Entry` that represents an element and its count, and the `SortedMultiset` interface includes additional methods such as `firstEntry` and `lastEntry`.

## Multimap

Like a map, a multimap stores entries that are key-value pairs (*k,v*), where *k* is the key and *v* is the value. Whereas a map insists that entries have unique keys, a multimap allows multiple entries to have the same key, much like an English dictionary, which allows multiple definitions for the same word. That is, we will allow a multimap to contain entries (*k,v*) and (*k,v'*) having the same key.

There are two standard approaches for representing a multimap as a variation of a traditional map. One is to redesign the underlying data structure to allow separate entries to be stored for pairs such as (*k,v*) and (*k,v'*). The other is to map key *k* to a secondary container of all values associated with that key (e.g., {*v,v'*}). An implementation should include the following:

- `get(k):` Returns a collection of all values associated with key *k* in the multimap.
- `put(k, v):` Adds a new entry to the multimap associating key *k* with value *v*, without overwriting any existing mappings for key *k*.
- `remove(k, v):` Removes an entry mapping key *k* to value *v* from the multimap (if one exists).
- `removeAll(k):` Removes all entries having key equal to *k* from the multimap.
- `size():` Returns the number of entries of the multiset (including multiple associations).
- `entries():` Returns a collection of all entries in the multimap.
- `keys():` Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).
- `keySet():` Returns a nonduplicative collection of keys in the multimap.
- `values():` Returns a collection of values for all entries in the multimap.

*Answer:* Just like a map, but instead of *V*, use some list of *V*'s. `get` works like normal, `put` adds a value to the list instead of just putting a value, `remove` removes a specific value from the list instead of removing *v* and `removeAll` works just like the old `remove` does.

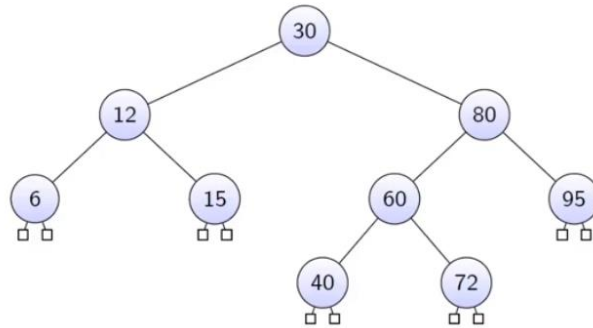
## Week 7. Search trees

### Binary Search Trees

#### Binary search trees

A **binary search tree** is a binary tree in which each node with key  $k$  satisfies the following properties:

- ▶ Keys stored in the left subtree are less than  $k$
- ▶ Keys stored in the right subtree are greater than  $k$



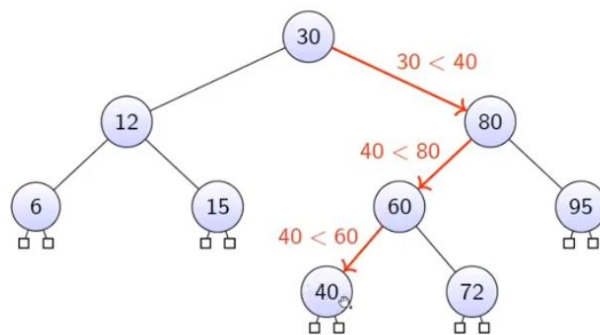
(Associated values omitted to ease drawing trees)

#### Binary search

**Binary search:** starting from the root, compare search key with key at node

- ▶ smaller: search left
- ▶ equal: success
- ▶ greater: search right

**Example:** search for 40:

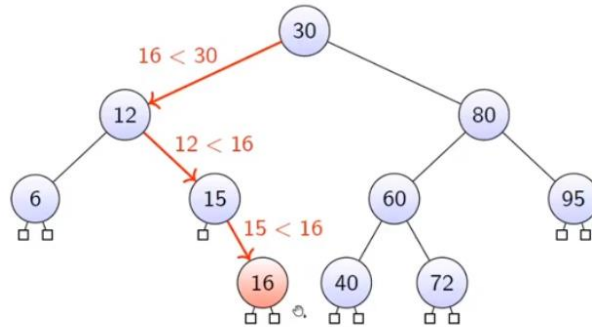


## Putting an element into a binary search tree

Find position for insertion:

- ▶ **not null**: key is already in the tree
- ▶ **null**: expand to actual node

**Example:** insert 16:

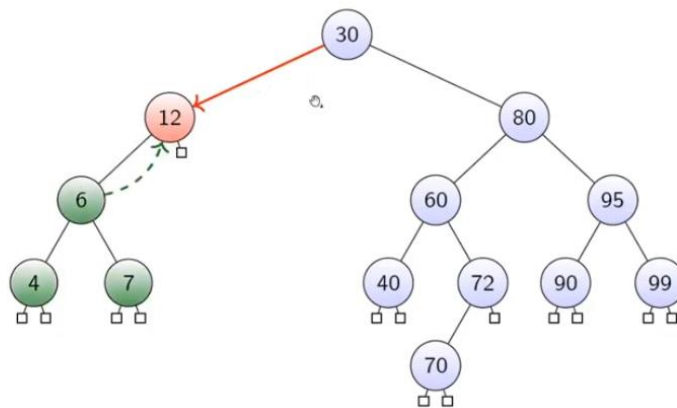


## Removing an element from a binary search tree (case 1)

**The easy case:** one of the children of the node is **null**

- ▶ remove node and replace it by the child

**Example:** remove 12:

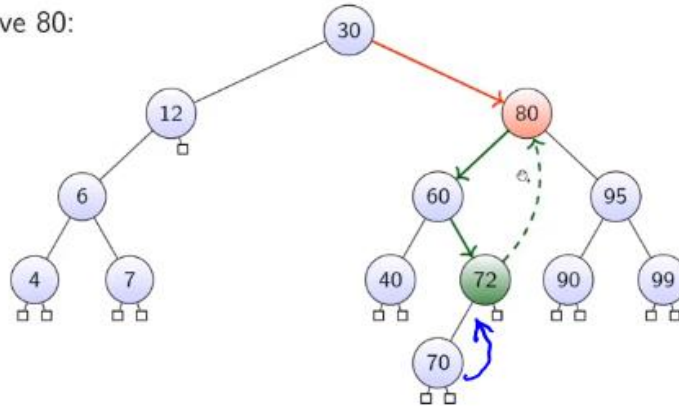


## Removing an element from a binary search tree (case 2)

**The difficult case:** both children of the node are not `null`

- ▶ Find maximum node in left child
- ▶ Replace the to be removed node by the maximum node
- ▶ Recursively proceed to remove the maximum node

**Example:** remove 80:



## Complexity of binary search

```

1 public V get(Position<Entry<K,V>> p, K key) {
2   if (p == null) return null;
3
4   int comp = compare(key, p.getElement());
5   if (comp == 0) return p.getElement();
6   else if (comp < 0) return get(left(p), key);
7   else return get(right(p), key);
8 }

```

Recurrence equations (where  $h$  is the height of the tree):

$$T(0) = c_0 \quad T(h) = T(h - 1) + c_1$$

Closed form:

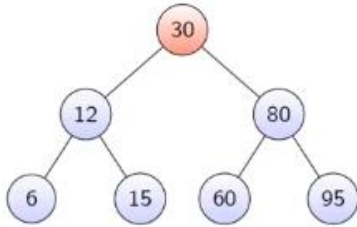
$$T(h) = h \cdot c_1 + c_0$$

The time-complexity of binary search is  $\mathcal{O}(h)$

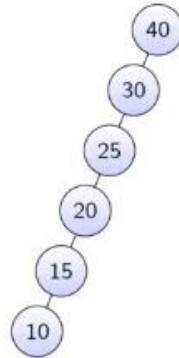
## Complexity of binary search

 How can we end up in the worst case?

**Best case:** height is  $\mathcal{O}(\log n)$



**Worst case:** height is  $\mathcal{O}(n)$



Insert a sequence whose keys is sorted (or inversely sorted), for example: 40, 30, 25, 20, 15, 10

## Balanced Search Trees

**Solution:** require a **balance condition** that:

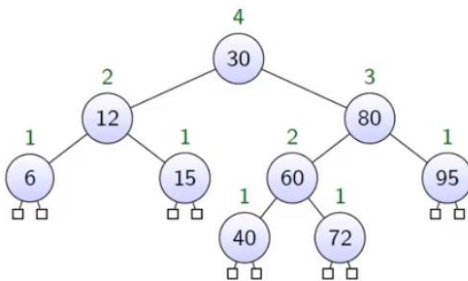
- ▶ ensures the depth is  $\mathcal{O}(\log n)$
- ▶ can be maintained in  $\mathcal{O}(\log n)$  time for each put and remove

## AVL trees

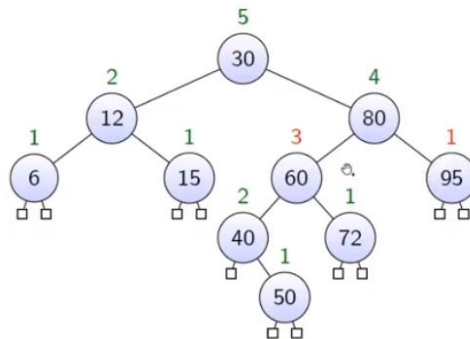
Invented by: **A**delson-**V**elskii and **L**andis

**AVL Balance condition:** the heights of the children of each node differ by at most 1

**Example 1:** a valid AVL tree



**Example 2:** an invalid AVL tree



*Note:* we consider **null** references as leaves when counting heights

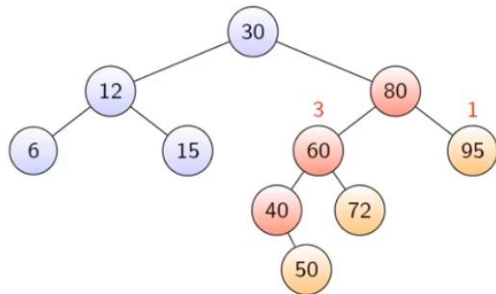


## Insertion into an AVL tree

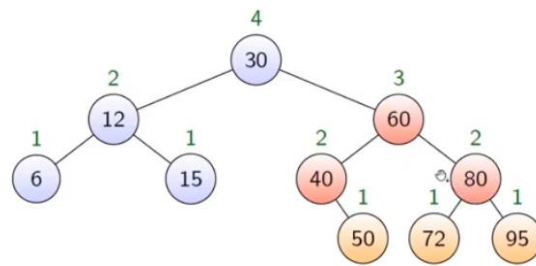
### Idea:

- ▶ use 'ordinary' binary tree insertion
- ▶ this may create unbalanced nodes, but only along the path to the inserted node
- ▶ "search and repair" while traversing to the root
- ▶ by "fixing" unbalanced nodes using "tri-node restructuring" along the way

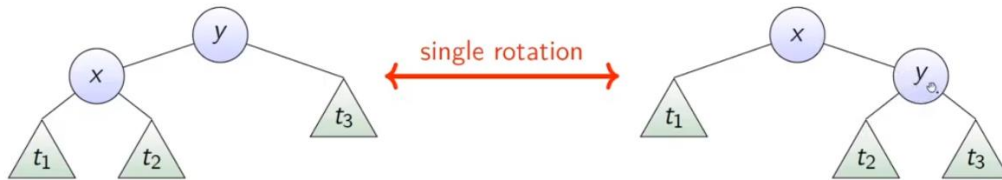
**Example:** insert 50



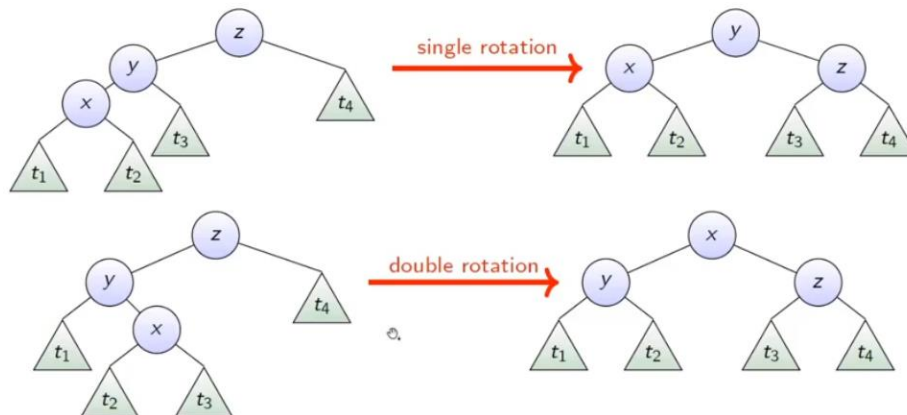
**Rebalance** to "fix" AVL property



## Rotations



## Tri-node restructurings



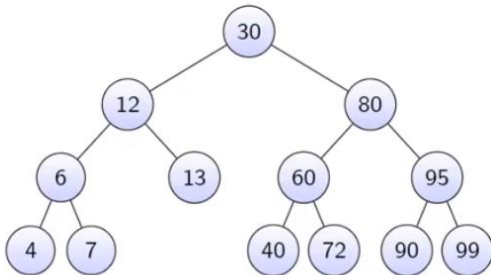
(And symmetrically)

## Removal from an AVL tree

**Idea:**

- ▶ use 'ordinary' binary tree removal
- ▶ this may create unbalanced nodes, but only along the path to the removed node
- ▶ "search and repair" while traversing to the root
- ▶ by 'fixing' unbalanced nodes using "tri-node restructuring" along the way

**Example:** remove 13

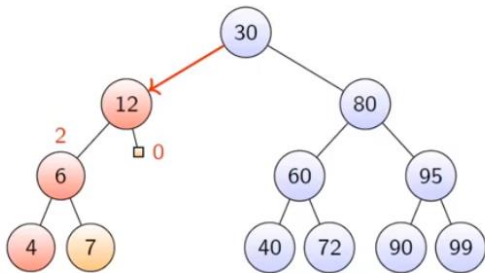


## Removal from an AVL tree

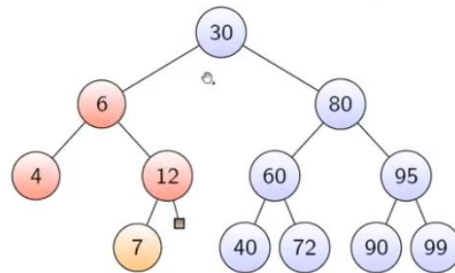
**Idea:**

- ▶ use 'ordinary' binary tree removal
- ▶ this may create unbalanced nodes, but only along the path to the removed node
- ▶ "search and repair" while traversing to the root
- ▶ by 'fixing' unbalanced nodes using "tri-node restructuring" along the way

**Example:** remove 13



**Rebalance to "fix" AVL property**

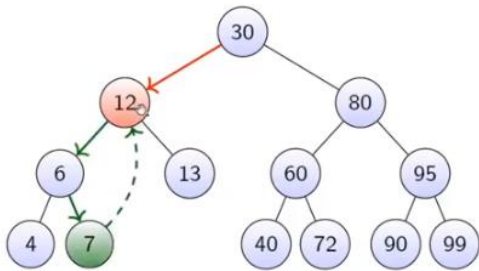


## Removal from an AVL tree

**Idea:**

- ▶ remove from the binary search tree
- ▶ might create unbalanced nodes, but only along the path to the removed node
- ▶ “search and repair” while traversing to the root
- ▶ by “fixing” unbalanced nodes using “tri-node restructuring” along the way

**Example:** remove 12

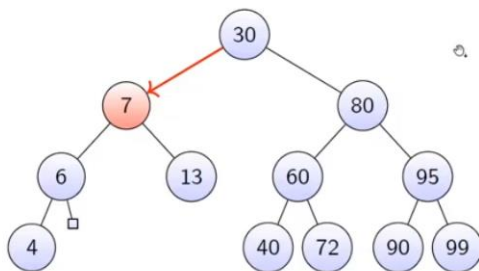


## Removal from an AVL tree

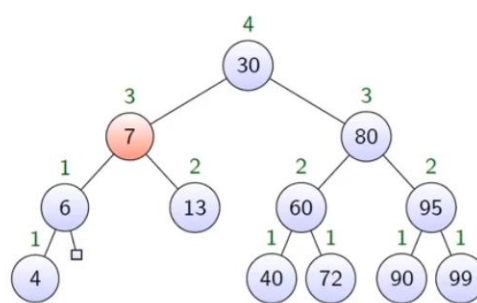
**Idea:**

- ▶ remove from the binary search tree
- ▶ might create unbalanced nodes, but only along the path to the removed node
- ▶ “search and repair” while traversing to the root
- ▶ by “fixing” unbalanced nodes using “tri-node restructuring” along the way

**Example:** remove 12



**No tri-node restructuring needed**



## Time complexity of AVL trees operations

The time-complexity of `get`, `put` and `remove` is  $\mathcal{O}(\log n)$

- ▶ A tri-node restructuring takes  $\mathcal{O}(1)$  time
- ▶ A `get`, `put` or `remove` visits  $\mathcal{O}(h)$  nodes and performs  $\mathcal{O}(h)$  tri-node restructurings
- ▶ The height of an AVL tree of size  $n$  is  $\mathcal{O}(\log n)$

Method	Running Time
<code>size</code> , <code>isEmpty</code>	$\mathcal{O}(1)$
<code>get</code> , <code>put</code> , <code>remove</code>	$\mathcal{O}(\log n)$
<code>firstEntry</code> , <code>lastEntry</code>	$\mathcal{O}(\log n)$
<code>ceilingEntry</code> , <code>floorEntry</code> , <code>lowerEntry</code> , <code>higherEntry</code>	$\mathcal{O}(\log n)$
<code>subMap</code>	$\mathcal{O}(s + \log n)$
<code>entrySet</code> , <code>keySet</code> , <code>values</code>	$\mathcal{O}(n)$

**Table 11.2:** Worst-case running times of operations for an  $n$ -entry sorted map realized as an AVL tree  $T$ , with  $s$  denoting the number of entries reported by `subMap`.



Explain in your own words the advantage of an AVL tree compared to an ordinary binary search tree.

*Answer:* The height of an AVL tree is  $\mathcal{O}(\log n)$ , which guarantees that the primary methods are  $\mathcal{O}(\log n)$ .



Explain in your own words why the height of an AVL tree with size  $n$  is  $\mathcal{O}(\log n)$  (no formal proof required).

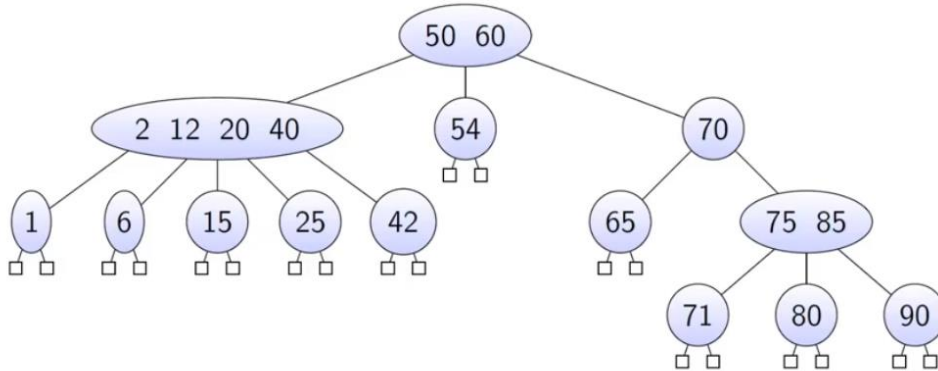
*Answer:* A tree with height  $h$  has as amount of nodes the sum of the nodes of the 2 subtrees plus the root itself. One of the subtrees must have height  $h - 1$  (otherwise the height wouldn't be  $h$ ), the other has at least  $h - 2$  (otherwise it wouldn't be balanced). This means the amount of nodes of a tree with height  $h$   $n(h) = 1 + n(h - 1) + n(h - 2)$ . If we unfold this, we can clearly see an exponential growth.

## (2,4) Trees

## Multiway search trees

A **multiway search tree** is a tree in which:

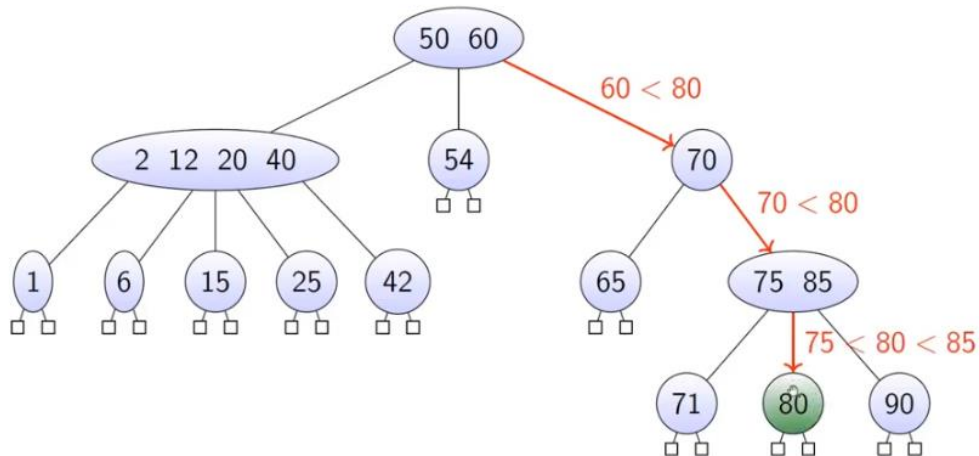
- ▶ each internal node has at least two children
- ▶ each internal node with  $d$  children contains an ordered list of  $k_1, \dots, k_{d-1}$  keys
- ▶ all keys  $k$  in the  $i$ th child of an internal node satisfy  $k_{i-1} < k < k_i$  where  $k_0 = -\infty$  and  $k_d = \infty$



## Multiway search

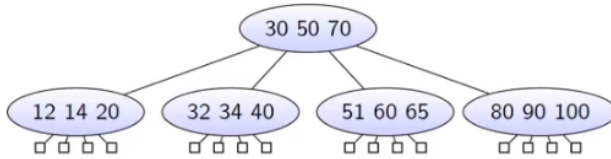
Multiway search is like searching in a binary search tree

**Example:** search for 80

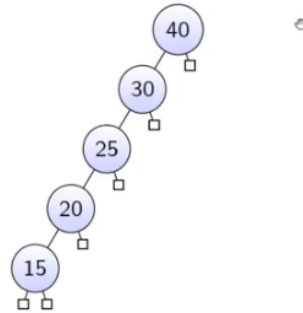


## Complexity of multiway search

**Best case:** height is  $\mathcal{O}(\log_d n)$ , i.e.  $\mathcal{O}(\log n)$   
 Multiway search is  $\mathcal{O}(\log n)$



**Worst case:** height is  $\mathcal{O}(n)$   
 Multiway search is  $\mathcal{O}(n)$



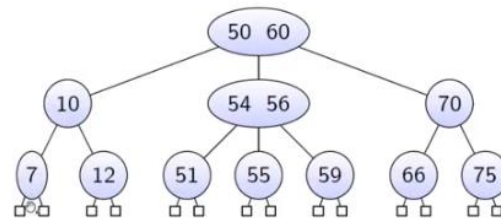
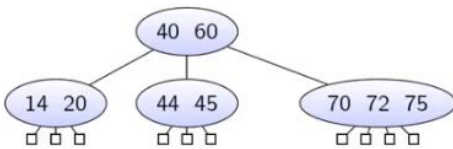
## (2,4) trees

A **(2,4) tree** is a multiway search tree that satisfies the following properties:

- ▶ **Size property:** every internal node has at most four children
- ▶ **Depth property:** all external nodes have the same depth

Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

**Examples:**

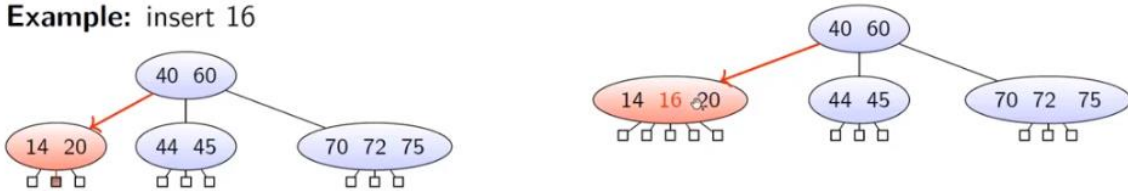


## Putting an entry into a (2,4) tree

Perform multiway search, and put the new entry at the *parent* of the leaf (i.e. parent of the `null`) where the search ended

- ▶ This preserves the *depth property*
- ▶ This may break the *size property* by causing an **overflow**: a node may become a 5-node

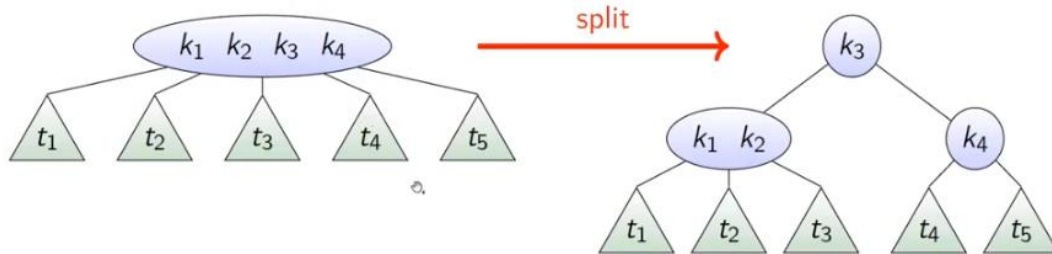
**Example:** insert 16



## Fixing overflow using split

Repairing an overflow:

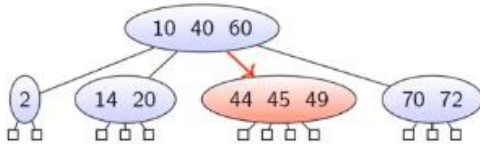
- ▶ We perform a **split** of the node that became a 5-node:



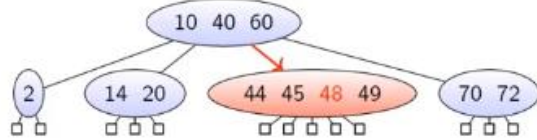
- ▶ If the node was the root, the resulting top root is the new root
- ▶ Otherwise, we merge the resulting top node with its parent
- ▶ If the parent overflows, we proceed recursively

## Split in action

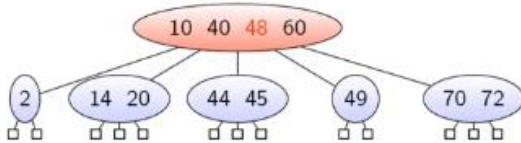
**Example:** insert 48



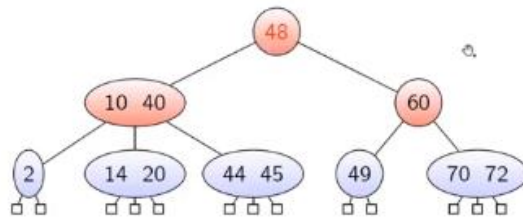
That causes an *overflow*:



That we fix by a *split*, causing another *overflow*:



That we fix by another *split*:



## Removing an entry from a (2,4) tree

Recipe for removing an entry:

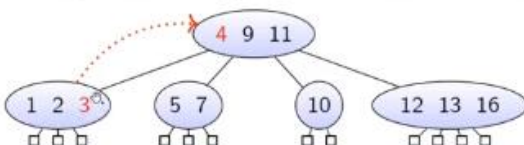
- ▶ Use multiway search to find the entry that needs to be removed
- ▶ If the entry is **not a node of level 1** (i.e. its children are not `null`) then:
  - ▶ Find the rightmost (=maximal) entry in the left subtree
  - ▶ Swap that entry with the key that has to be removed
  - ▶ Proceed with the rightmost entry
- ▶ Remove the entry, and 'repair' the tree while traversing upwards

## Examples of removing an entry from a (2,4) tree

**Remember:** if the entry is **not a node of level 1** (i.e. its children are not `null`) then:

- ▶ Find the rightmost (=maximal) entry in the left subtree
- ▶ Swap that entry with the key that has to be removed
- ▶ Proceed with the rightmost entry

**Example:** deleting 4 can be reduced to deleting 3



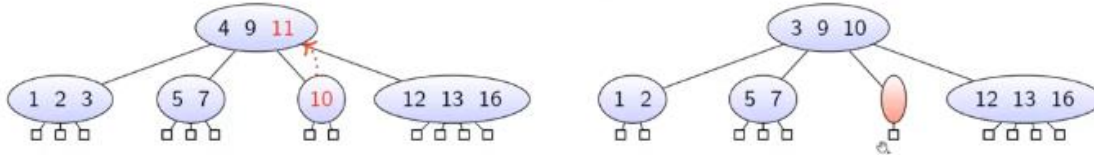


## Examples of removing an entry from a (2,4) tree

**Remember:** if the entry is **not a node of level 1** (i.e. its children are not **null**) then:

- ▶ Find the rightmost (=maximal) entry in the left subtree
- ▶ Swap that entry with the key that has to be removed
- ▶ Proceed with the rightmost entry

**Example:** deleting 11 can be reduced to deleting 10

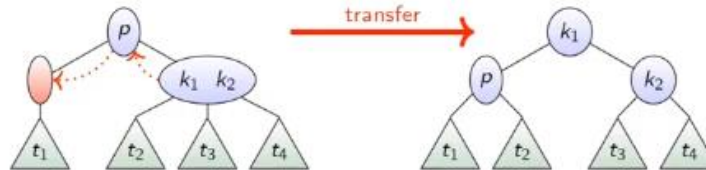


Removal may break the *size property* by causing an **underflow**: a node may become a 1-node

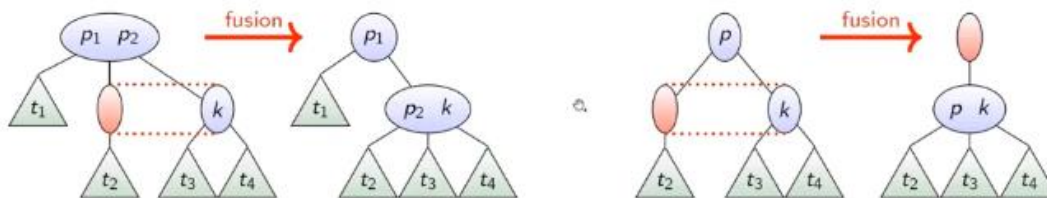
## Fixing underflow using transfer and fusion

Removal may break the *size property* by causing an **underflow**: a node may become a 1-node. To handle an underflow, we consider two cases (in given order):

(a) An adjacent sibling is a 3-node or a 4-node, we then perform a **transfer**:



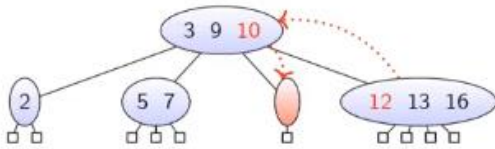
(b) An adjacent sibling is a 2-node, we then perform a **fusion**:



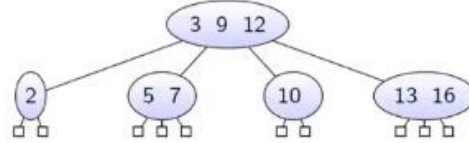
Tricky case: parent becomes a 1-node; proceed recursively to the root

## Transfer in action

**Example:** remove 11

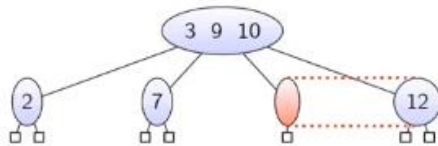


Result after a transfer:

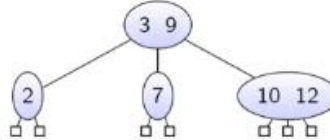


## Fusion in action

**Example:** remove 11



Result after fusion:




## Time complexity of (2,4) tree operations

The time-complexity of get, put and remove is  $\mathcal{O}(\log n)$

- ▶ The height of a (2,4) tree of size  $n$  is  $\mathcal{O}(\log n)$  (see proof in the book)
- ▶ A split, transfer, or fusion take  $\mathcal{O}(1)$  time
- ▶ A get, put or remove visits  $\mathcal{O}(h)$  nodes

## When to use balanced search trees or hash tables?

 Hash tables seem great with  $\mathcal{O}(1)$  (expected) operations? Why not use them always instead of balanced search trees?

Possible reasons not to use hash tables:

- ▶ They are unordered, they **do not** support fast operations for taking the minimal or maximal element, for in-order traversal, ...
- ▶ Lookup/insert/remove become  $\mathcal{O}(n)$  with poor hash-functions
- ▶ Periodic rehashing can be problematic in real-time systems

Possible reasons to use AVL/red-black trees:

- ▶ Lookup/insert/remove with  $\mathcal{O}(\log n)$  **worst case**
- ▶ They are ordered, they **do** support fast operations for taking the minimal or maximal element, for in-order traversal, ...

## Recap: The sorted map ADT

```

1 interface SortedMap<K,V> {
2     /* Returns the entry with smallest key value */
3     Entry<K,V> firstEntry();
4     /* Returns the entry with largest key value */
5     Entry<K,V> lastEntry();
6     /* Returns the entry with the least key value greater than or equal to k */
7     Entry<K,V> ceilingEntry(K k);
8     /* Returns the entry with the greatest key value less than or equal to k */
9     Entry<K,V> floorEntry(K k);
10    /* Returns the entry with the greatest key value strictly less than k */
11    Entry<K,V> lowerEntry(K k);
12    /* Returns the entry with the least key value strictly greater than k */
13    Entry<K,V> higherEntry(K k);
14 }

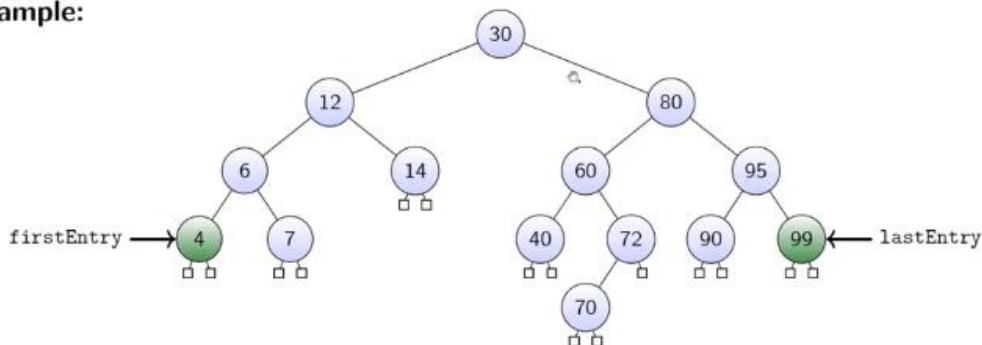
```

## First and last entry of binary search trees

Given a tree of height  $h$ :

- ▶ `Entry<K,V> firstEntry()`: pick the left-most node  $\mathcal{O}(\log h)$
- ▶ `Entry<K,V> lastEntry()`: pick the right-most node  $\mathcal{O}(\log h)$

Example:



## Summary

**Time complexity** (given a map with  $n$  elements):

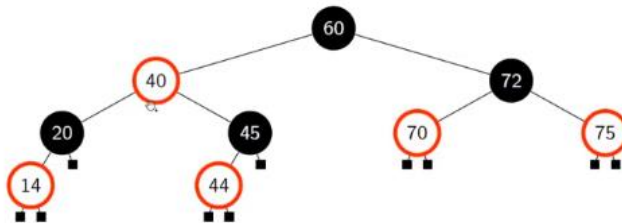
Operation	AVL/red-black trees	Hash tables
V get(K key)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ expected
V put(K key, V value)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ expected
V remove(K key)	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ expected
Entry<K,V> firstEntry()	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Entry<K,V> lastEntry()	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Entry<K,V> ceilingEntry(K k)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Entry<K,V> floorEntry(K k)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Entry<K,V> lowerEntry(K k)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Entry<K,V> higherEntry(K k)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

## Red-Black Trees

### Red-black trees

A **red-black tree** is a binary search tree with nodes colored red and black that enjoys:

- ▶ **Root property:** the root is black
- ▶ **External property:** every leaf (i.e. `null`) is black
- ▶ **Red property:** the children of a red node are black
- ▶ **Depth property:** all external nodes have the same black depth, defined as the number of *proper* ancestors that are black

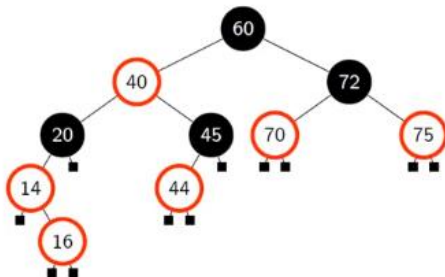


### Putting an entry into a red-black tree

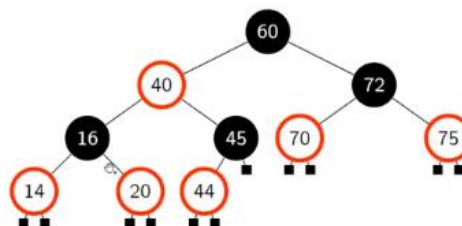
Perform binary search, and put the new red entry at the leaf (i.e. the `null`) where the search ended, and add two black leaves to the just created entry

- ▶ This preserves the *depth property*
- ▶ This may break the *red property*: two red nodes may be above each other

**Example:** insert 16

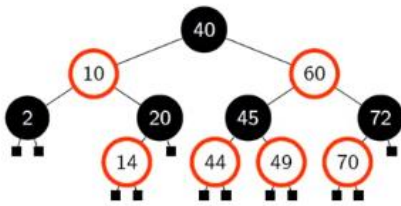


Restructure (like the AVL Tree)

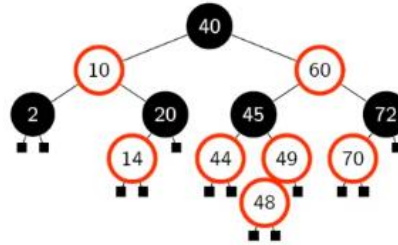


## Putting an entry into a red-black tree

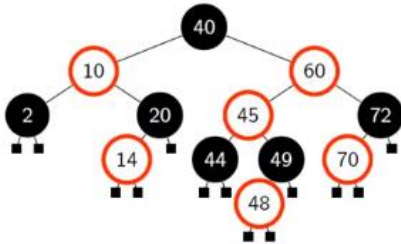
**Example:** insert 48



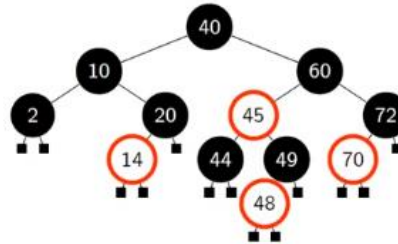
That causes a double red:



Recolor:



Another recolor:



6

## Removing an entry from a Red-Black tree

Recipe for removing an entry:

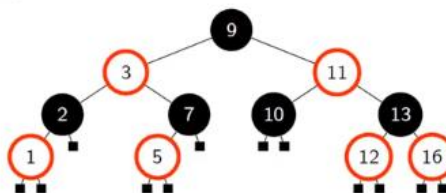
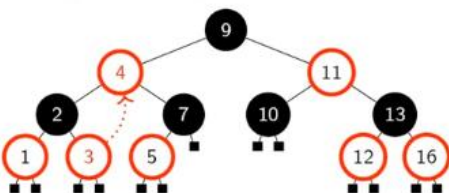
- ▶ Use binary search to find the entry that needs to be removed
- ▶ If the entry is **not a node of level 1** (i.e. its children are not `null`) then:
  - ▶ Find the rightmost (=maximal) entry in the left subtree
  - ▶ Swap that entry with the key that has to be removed
  - ▶ Proceed with the rightmost entry
- ▶ Remove the entry, and 'repair' the tree while traversing upwards

## Examples of removing an entry from a Red-Black tree

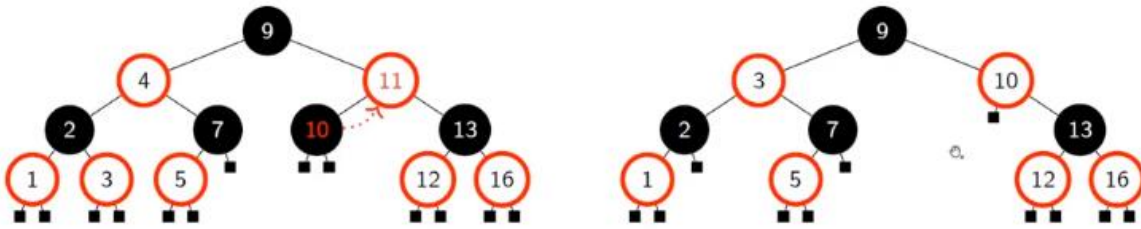
**Remember:** if the entry is **not a node of level 1** (i.e. its children are not `null`) then:

- ▶ Find the rightmost (=maximal) entry in the left subtree
- ▶ Swap that entry with the key that has to be removed
- ▶ Proceed with the rightmost entry

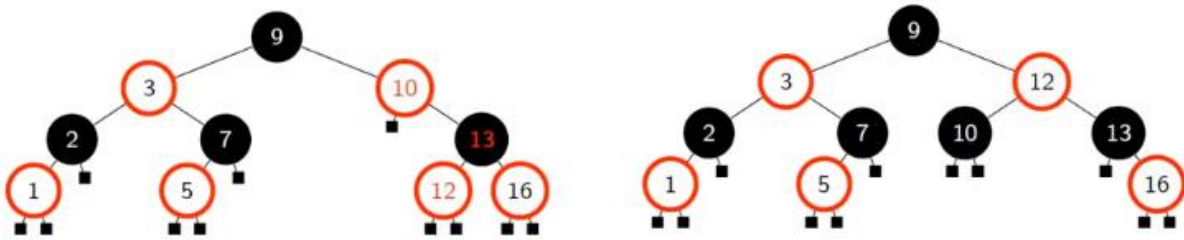
**Example:** deleting 4 can be reduced to deleting 3



**Example:** deleting 11 can be reduced to deleting 10

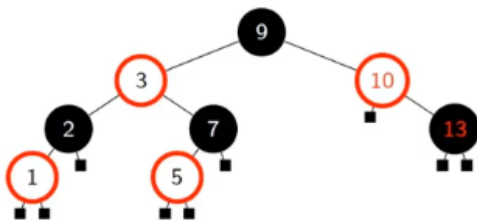


Removal may break the *depth property*

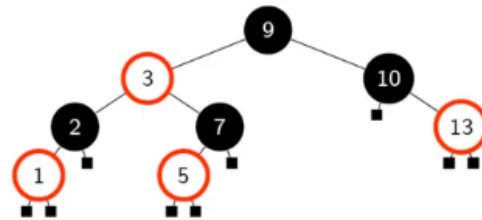


Restructure in action if black sibling has only black childs

**Example:** remove 11



Result after a restructure:



Time complexity of red-black tree operations

The time-complexity of `get`, `put` and `remove` is  $\mathcal{O}(\log n)$

- ▶ The height of a red-black tree of size  $n$  is  $\mathcal{O}(\log n)$  (see proof in the book)
- ▶ Recoloring operations take  $\mathcal{O}(1)$  time
- ▶ A `get`, `put` or `remove` visits  $\mathcal{O}(h)$  nodes


## Red-black trees versus AVL trees

Both have  $\mathcal{O}(\log n)$  time complexity for get, put and remove, so what is the difference?

AVL trees are better balanced, hence:

- ▶ get is faster for AVL trees
- ▶ put and remove of a red-black tree is faster, because red-black trees require less re-balancing <sup>ⓐ</sup>

## When to use balanced search trees or hash tables?

 Hash tables seem great with  $\mathcal{O}(1)$  (expected) operations? Why not use them always instead of balanced search trees?

Possible reasons not to use hash tables:

- ▶ They are unordered, they **do not** support fast operations for taking the minimal or maximal element, for in-order traversal, ...
- ▶ Lookup/insert/remove become  $\mathcal{O}(n)$  with poor hash-functions
- ▶ Periodic rehashing can be problematic in real-time systems

Possible reasons to use AVL/red-black trees:

- ▶ Lookup/insert/remove with  $\mathcal{O}(\log n)$  **worst case**
- ▶ They are ordered, they **do** support fast operations for taking the minimal or maximal element, for in-order traversal, ...

## Recap: The sorted map ADT

```

1 interface SortedMap<K,V> {
2     /* Returns the entry with smallest key value */
3     Entry<K,V> firstEntry();
4     /* Returns the entry with largest key value */
5     Entry<K,V> lastEntry();
6     /* Returns the entry with the least key value greater than or equal to k */
7     Entry<K,V> ceilingEntry(K k);
8     /* Returns the entry with the greatest key value less than or equal to k */
9     Entry<K,V> floorEntry(K k);
10    /* Returns the entry with the greatest key value strictly less than k */
11    Entry<K,V> lowerEntry(K k); ⓐ
12    /* Returns the entry with the least key value strictly greater than k */
13    Entry<K,V> higherEntry(K k);
14 }

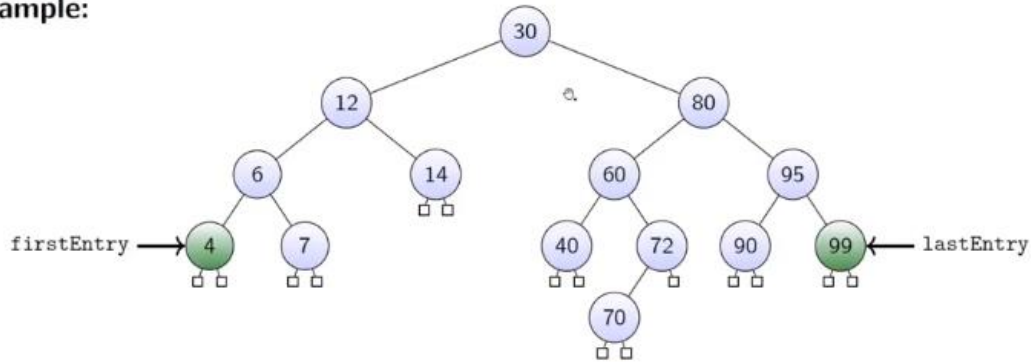
```

## First and last entry of binary search trees

Given a tree of height  $h$ :

- ▶  $\text{Entry}\langle K, V \rangle$  `firstEntry()`: pick the left-most node  $O(\log h)$
- ▶  $\text{Entry}\langle K, V \rangle$  `lastEntry()`: pick the right-most node  $O(\log h)$

**Example:**



## Summary

**Time complexity** (given a map with  $n$  elements):

Operation	AVL/red-black trees	Hash tables
$V$ <code>get(K key)</code>	$O(\log n)$	$O(1)$ expected
$V$ <code>put(K key, V value)</code>	$O(\log n)$	$O(1)$ expected
$V$ <code>remove(K key)</code>	$O(\log n)$	$O(1)$ expected
$\text{Entry}\langle K, V \rangle$ <code>firstEntry()</code>	$O(\log n)$	$O(n)$
$\text{Entry}\langle K, V \rangle$ <code>lastEntry()</code>	$O(\log n)$	$O(n)$
$\text{Entry}\langle K, V \rangle$ <code>ceilingEntry(K k)</code>	$O(\log n)$	$O(n)$
$\text{Entry}\langle K, V \rangle$ <code>floorEntry(K k)</code>	$O(\log n)$	$O(n)$
$\text{Entry}\langle K, V \rangle$ <code>lowerEntry(K k)</code>	$O(\log n)$	$O(n)$
$\text{Entry}\langle K, V \rangle$ <code>higherEntry(K k)</code>	$O(\log n)$	$O(n)$



Week 8. Lost due to crash. Hardcopy available...