# CS2110
# Software Development Methods

Panagiotis Apostolellis, PhD

Last Class!!!

# Reminders

### When & Where
01

Saturday May 2nd throughout the day on your own laptop; find a stable connection

### How? (format)
02

Two-hours long on Collab & external website (time stamped on open/submit)

### What (is allowed)?
03

Exam is *open books/notes/IDE*; no collaboration; find details under "Labs & Exam Review"

### Extra Credit
04

Complete Course Evals on Collab before midnight May 1st to get 1% extra credit added your final grade

# Final Exam Review

Get ready to participate!

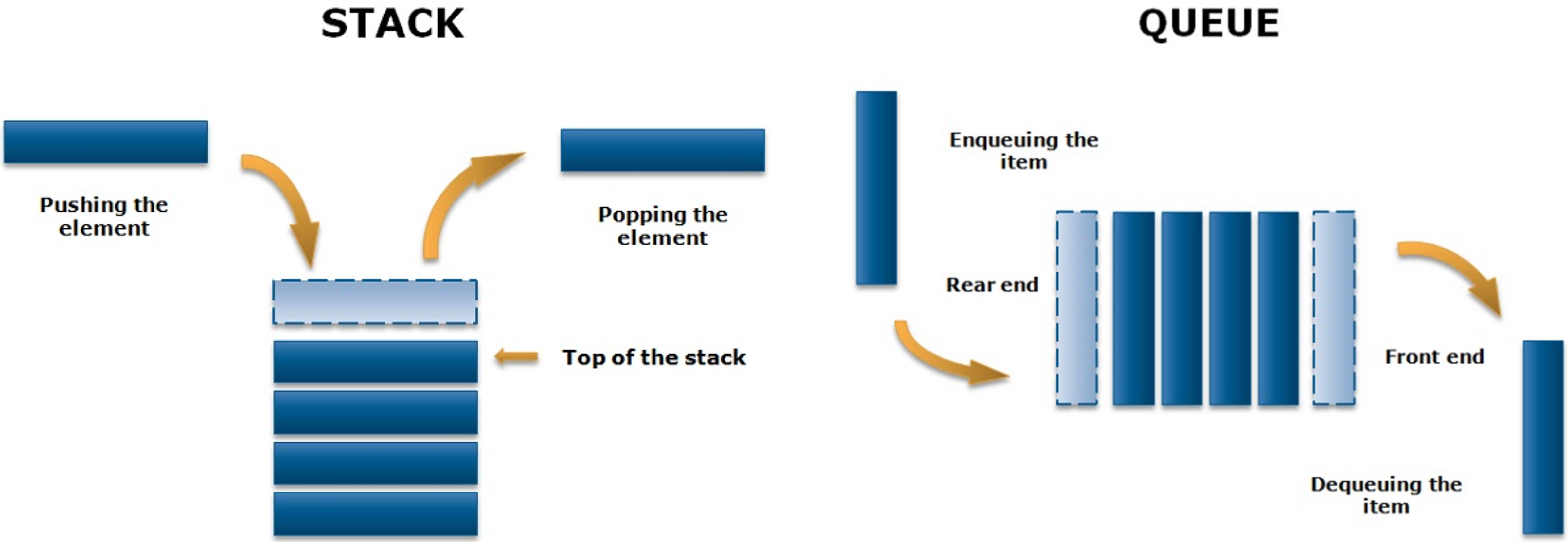
Go to **www.menti.com** and use the code **44 97 31**

# Stacks & Queues

Data structures for storing and retrieving elements

# Stacks vs Queues

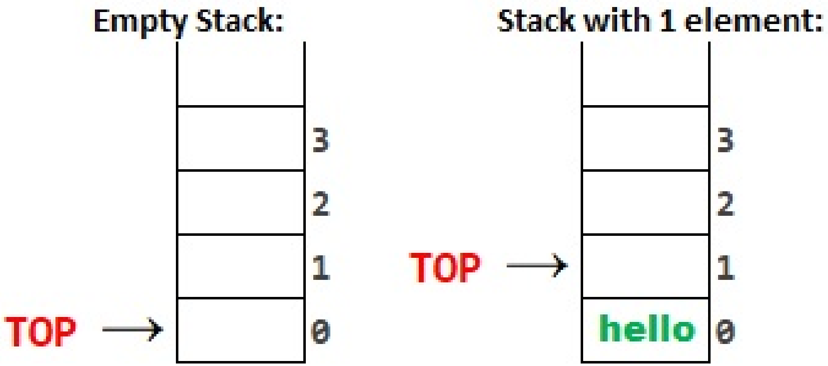
## Last-In, First-Out (LIFO) vs. First-In, First-Out (FIFO)

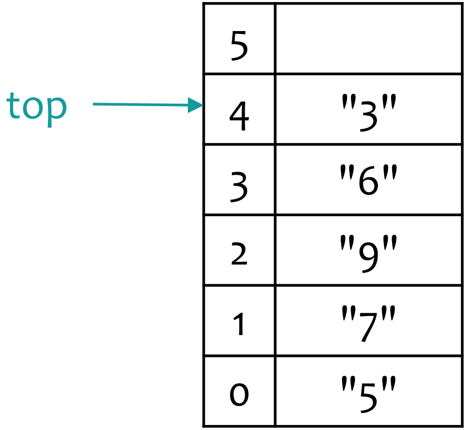


# Stack – push()

Adding an item to the top of the stack

```
public void push(String s){
    System.out.print("Push! ");
    growIfNecessary(); // if running out of room...
    theStack[top] = s; // new item inserted at position "top"
    top++;  // increment top pointer
}
```



# Stack – pop()

## Removing an item from the top of the stack

```java
public String pop(){
    if(top == 0){ // if nothing in the Stack (when top is at 0)
        return null;
    }
    top--; // decrement top pointing at current top item
    return theStack[top];
    // return the item that was at the top
    // during the next push operation,
    // new item will be added here
}
```

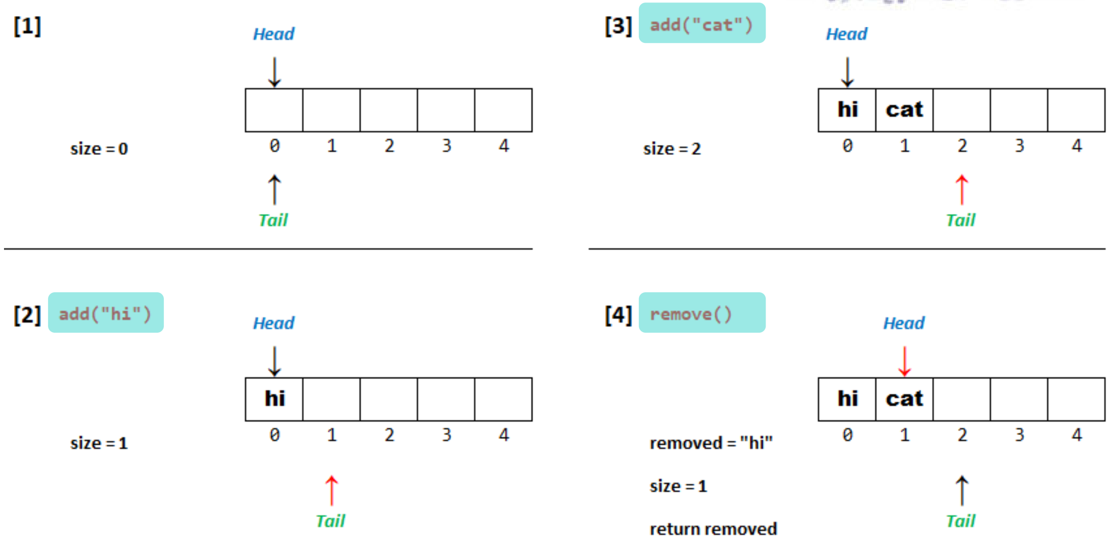
top →

| | |
|---|---|
| 5 | |
| 4 | "3" |
| 3 | "6" |
| 2 | "9" |
| 1 | "7" |
| 0 | "5" |

- What would we change to **peek()** – Look but don’t remove?
    - Do not decrement top
    - Return `theStack[top-1]`

# Queues

## First-In, First-Out (FIFO)



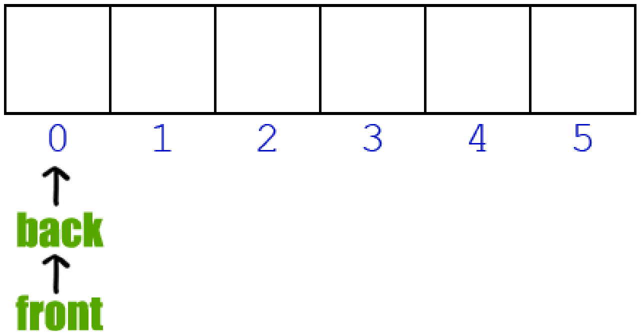

Although "hi" isn't removed, what is valid is between the "Head and "Tail" pointers

- Remember: work is done at BOTH ends of the queue:
  - Adding to the *tail*; Removing from the *head*

# Queue – basic operations

**Implementing add() and remove() methods**

- Think about how you would keep track of where to **insert** (*enqueue*) into the array and where you would **remove** (*dequeue*) from the array [**Hint**: *pointers*]
- Think about how you would handle the fact that when you remove from an array, you have an empty slot
  [**Hint**: either shift all the elements inside the array, or just keep track, via int pointers, of the location of the *head* and *tail*]




dequeue()

ue(x)

# Queues

## What would be the output?

```
Queue<String> q = new LinkedList<String>();
queue.add("Wishing you ");
queue.add("A happy ");
queue.add("Summer!");
queue.remove();
System.out.println(q.peek());
System.out.println(q.remove());
q.peek();
```

This does not print anything!

| Choice | Output |
|---|---|
| A | Wishing you<br>A happy |
| B | A happy<br>Summer! |
| C | A happy<br>A happy<br>Summer |
| D | A Happy<br>A Happy |
| E | Wishing you<br>A Happy<br>Summer! |

# Stacks

## What would be the output?

```java
Stack<String> stack = new Stack<String>();
stack.add(" Fall Semester!");
stack.add(" Great");
stack.add("And a");
System.out.println(stack.pop() +
    stack.peek() + stack.pop());
```

This does not remove the element!

| Choice | Output |
|---|---|
| A | And a Great Fall Semester! |
| B | Fall Semester! GreatAnd a |
| C | And a Great Great |
| D | And a Fall Semester! |
| E | Compiler error |

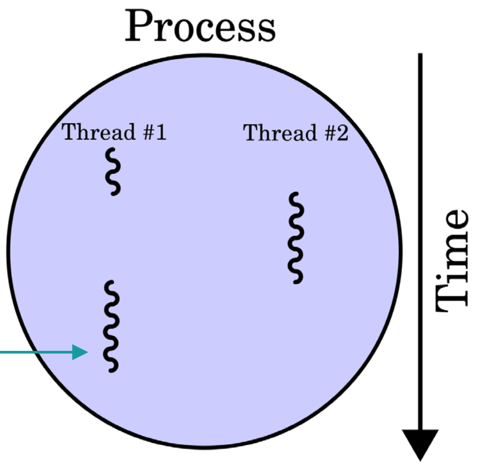
# Concurrency

Executing multiple threads simultaneously

# Running Threads

## Independent execution of program parts

- Often it is useful for a program to carry out two or more tasks at the same time. This can be achieved by implementing ***threads***
- **Thread**: a program unit that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a short amount of time (“*time slice*”)
- This gives the impression of parallel execution
- If a computer has multiple central processing units (CPUs), then some of the threads can run in parallel, one on each processor



# Running Threads

## Implementing the Runnable interface

1. Create a task to be run in a thread by implementing the **Runnable interface**

```
public interface Runnable
{
    void run(); // one method stub
}
```

2. Place your code for the task into the **run()** method of your class

```
public class MyRunnable implements
                            Runnable
{
    public void run() {
        // Task statements
        // . . .
    }
}
```

# Running Threads

## Constructing and running a thread

3. Create an object of your subclass (e.g., MyRunnable)

```
Runnable task = new MyRunnable();
```

4. Construct a **Thread** object from the MyRunnable object

```
Thread t = new Thread(task);
```

5. Call the start() method (from the Thread class) to start the thread (eventually, it will execute the run() method)

```
t.start();
```

# Threads

## What are the chances we can get this exact output every time?

```java
GreetingRunnable r1 = new GreetingRunnable("Hello");
GreetingRunnable r2 = new GreetingRunnable("Goodbye");
// Create TWO threads and run them
Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);
t1.start();
t2.start();
```

Console

<terminated> GreetingThreadRunner [Java Application] C:\Program Files\Java\jdk-12.0.2\bin\java

```
Sun Nov 10 13:22:59 EST 2019 Goodbye
Sun Nov 10 13:23:00 EST 2019 Hello
Sun Nov 10 13:23:00 EST 2019 Goodbye
Sun Nov 10 13:23:01 EST 2019 Goodbye
Sun Nov 10 13:23:01 EST 2019 Hello
Sun Nov 10 13:23:02 EST 2019 Hello
Sun Nov 10 13:23:02 EST 2019 Goodbye
```

| Choice | Output |
|---|---|
| A | 100% |
| B | Depends only on CPU speed |
| C | Depends on size of the input |
| D | Only God knows! |

# Thread Scheduler

## In charge of running each thread for a time slice

- **Thread scheduler:** runs each thread for a short amount of time (a *time slice*)
- Then the scheduler activates another thread
- There will always be *slight variations in running times* – especially when calling operating system services (e.g. input and output)
- ***There is no guarantee about the order in which threads are executed!***
  - As we can see with the **GreetingThreadRunner.java** example: the “Hello” and “Goodbye” statements are *not* perfectly interleaved

```
Console
<terminated> GreetingThreadRunner [Java Application] C:\Program Files\Java\jdk-12.0.2\bin\javaw.exe
Sun Nov 10 13:22:59 EST 2019 Goodbye
Sun Nov 10 13:23:00 EST 2019 Hello
Sun Nov 10 13:23:00 EST 2019 Goodbye
Sun Nov 10 13:23:01 EST 2019 Goodbye
Sun Nov 10 13:23:01 EST 2019 Hello
Sun Nov 10 13:23:02 EST 2019 Hello
Sun Nov 10 13:23:02 EST 2019 Goodbye
```

# 4 Terminating Threads

## What is the best way to terminate a thread?

| Choice | Output |
|---|---|
| A | Use the `Thread.stop()` method |
| B | Use a try/catch for an `InterruptedException` |
| C | Use the `Thread.interrupted()` method |
| D | Use the `Thread.interrupt()` method |

This is unsafe and the method is deprecated

# Terminating Threads

## How to interrupt the execution of threads

- A thread terminates when its **run()** method terminates
- Do not terminate a thread using the *deprecated* `stop()` method
- Instead, notify a thread that it should terminate:

```
t.interrupt(); // notifies the thread that
               // it should terminate
```

- **interrupt()** does not cause the thread to terminate – it sets a boolean variable in the thread data structure
- **sleep()** suspends execution of the thread
  - When interruption happens while sleeping, the thread is oblivious of the interruption!

5

# Terminating Threads

## What is the best way to handle a terminating thread?

| Choice | Output |
|---|---|
| A | Use the `Thread.stop()` method |
| B | Use a try/catch for an `InterruptedException` |
| C | Use the `Thread.interrupted()` method |
| D | Use the `Thread.interrupt()` method |

Thread will not wake up if *sleeping* while interrupted

# Threads

## Running and handling threads

- A **thread** is a program unit that is executed concurrently with other parts of the program.
- The **start()** method of the *Thread class* starts a new thread that executes the **run()** method of the associated *Runnable object*.
- The **sleep()** method puts the current thread to sleep for a given number of milliseconds.
- The **sleep()** method throws a *checked* **InterruptedException** so all calls to sleep must be wrapper in a *try/catch statement* (or declared).
- When a thread is **interrupted**, using the Thread's **interrupt()** call, you should commonly terminate the run() method (but must be done *explicitly*).
- The **thread scheduler** runs each thread for a short amount of time, called a **time slice**.

# 6 Concurrency Issues

## Select the solutions to common concurrency issues

| Choice | Output |
| --- | --- |
| A | Race conditions |
| $\boxed{B}$ | Lock conditions |
| C | Deadlocks |
| $\boxed{D}$ | Synchronizing |
| $\boxed{E}$ | Lock objects |

These are the problems not the solutions

# Race Condition

## And how to avoid it using Synchronizing

- A **race condition** occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.
- The solution to a race condition is **synchronizing** the methods that manipulate the same resource(s):
  1. You can do this via a **Lock object** and using the **lock()** and **unlock()** methods on that object to lock access to the object’s methods;
     - no other thread can *acquire the lock* until it’s *released* by the first thread
  2. You can also use the **synchronized** method modifier when you want to prevent two threads entering the same method;
     - once a thread has entered a *synchronized method*, no other thread can enter any other synchronized method on the same object!

```
public synchronized void deposit(double amount){...}
```

# Deadlocks

## And how to avoid them using Condition objects

- A **deadlock** occurs if no thread can proceed because each thread is waiting for another to do some work first.
- A **Condition object** provides a thread with the ability to suspend its execution, until the *condition* (*test*) is true; it is necessarily bound to a *Lock* and can be obtained using the **newCondition()** method.
- Calling **await()** on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is *blocked* until another thread calls **signalAll()** on the *condition object* for which the thread is waiting.
    - **signal()** randomly picks just one thread waiting on the object and unblocks it
    - **signalAll()** can be more efficient, but you need to know that every waiting thread can proceed
    - **Recommendation:** always call **signalAll()**

7

# Condition Objects

## In which line you would release the lock?

```
1 public void withdraw(double amount) {
2
3    balanceChangeLock.lock();  // lock!
4    try
5    {
6       while (balance < amount) {  // if not enough balance…
7          sufficientFundsCondition.await();   // …wait!
8          ?
9       }
10      ?
11   }
12   finally
13   {
14      balanceChangeLock.unlock(); // unlock!
15   }
16   ?
17 }
```

You want this to happen no matter what!

| Choice | Line |
|---|---|
| A | Line 8 |
| B | Line 10 |
| C | Line 14 |
| D | Line 16 |

# Recursion

Breaking down a solution to smaller parts

# Recursion in Algorithms

## Different views of Recursion

- **Recursive Definition:**
  - defining the elements in a set, in terms of other elements in the set (non-math examples are common too):
  - `n! = n * (n-1)!`
- **Recursive Procedure:**
  - a procedure that calls itself (more coming up)
- **Recursive Data Structure:**
  - a data structure that contains a pointer to an instance of itself

```
public class ListNode {
      Object nodeItem;
      ListNode next, previous;
      ...
}
```

8

# Recursive Factorial

**Does this implementation work?**

```
public static int factorial(int n){
    int x = factorial(n-1);
    if (n <= 0)
      return 1;
    else
      return n * x;
}
```

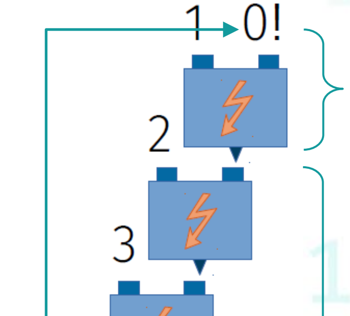
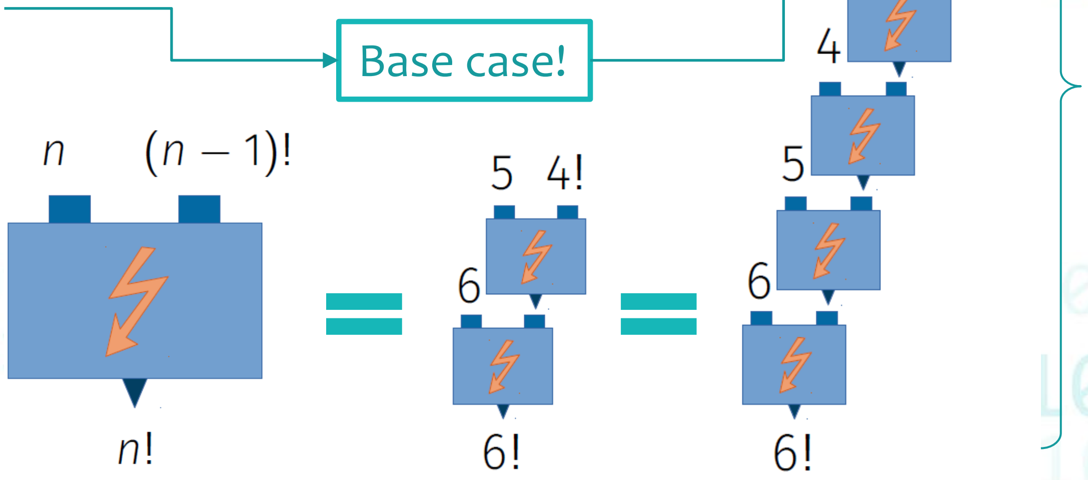
The *base case* is never checked, so we're trapped in an infinite recursion!

| Choice | Line |
| --- | --- |
| A | Yes, it's fine |
| B | No, because… |

# Recursive Example

## Implementing factorial recursively

- Factorial: n! = n * (n-1) * (n-2) * … * 2 * 1
- Can be written as: n! = n * (n-1)!
- Can be solved by multiplying two numbers…
- …and continuously applying the same process
- Note: 0! = 1! = 1


**Base case:**
- n = 0 → 0! = 1 (solved directly; no recursion)

**Recursive case:**
- n > 0 → n! = n * (n-1)!

**Advice:**
- *Always put the base case first!*

# Fibonacci

## Which is a better implementation (and why)?

```
public long fib(int n) {
      if ( n <= 2 ) return 1;
      return fib(n-1) + fib(n-2);
}
```

$O(2^n)$

```
long fib(int n) {
  if ( n <= 2 ) return 1;
  long answer;
  long prevFib=1, prev2Fib=1;
  for (int k = 3; k <= n; k++) {
      answer = prevFib + prev2Fib;
      prev2Fib = prevFib;
      prevFib = answer;
  }
  return answer;
}
```

$O(n)$

| Choice | Line |
|---|---|
| A | Recursive, because… |
| B | Iterative, because… |

***Memoization***

*An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again*

# Recursion vs Iteration

**Basic idea between recursion and iteration**

- **Recursion**
  - "Loop" is stopped by *base case*
  - Build solution from *top down*

```
public int recurse(int i) {
    if(i >= 5)
        return i;
    //do something
    return recurse(i++);
}
```

- **Iteration**
  - Loop *condition* determines when to stop
  - Build solution from *bottom up*

```
public int iterate(int i) {
    while (i < 5) {
        //do something
        i++;
    }
    return i;
}
```

- Any recursive solution may be written using iteration
- Recursive algorithm may appear simpler, more intuitive but is usually less efficient

# 10 Recursive Algorithm

## What is the output of printAna(“”, “to”)?

```
public static void printAna (String prefix, String word) {
   if(word.length() <= 1) {
      System.out.print(prefix + word);
   } else {
      for(int i = 0; i < word.length(); i++) {
            String cur = word.substring(i, i + 1);
            String before = word.substring(0, i); // letters before cur
            String after = word.substring(i + 1); // letters after cur
            printAna (prefix + cur, before + after);
      }
   }
}
```

*Anagram algorithm*: Breaks down the problem by cutting down each word to the first letter (*prefix*) and the remaining string (*word*)

| Record | Output |
|---|---|
| "", to | |
| "t", "o" | to |
| "o", "t" | ot |

> toot

# Binary Trees

Hierarchical data structures

# Binary Trees

## A recursive data structure

- **Recursive data structure:** *a data structure that contains references (or pointers) to an instances of that same type*

```
public class TreeNode<E> {
        private E data;
        private TreeNode<E> left;
        private TreeNode<E> right;
        …
}
```

- Recursion is a *natural way* to express many data structures (e.g., Trees)
  - For these, it's natural to have recursive algorithms (operations)
- Tree operations may come in two flavors:
  - **Node-specific** (e.g. `hasParent()` or `hasChildren()`)
  - **Tree-wide** (e.g. `size()` or `height()`) – requires ***tree traversal***

# Recursive Data Structure

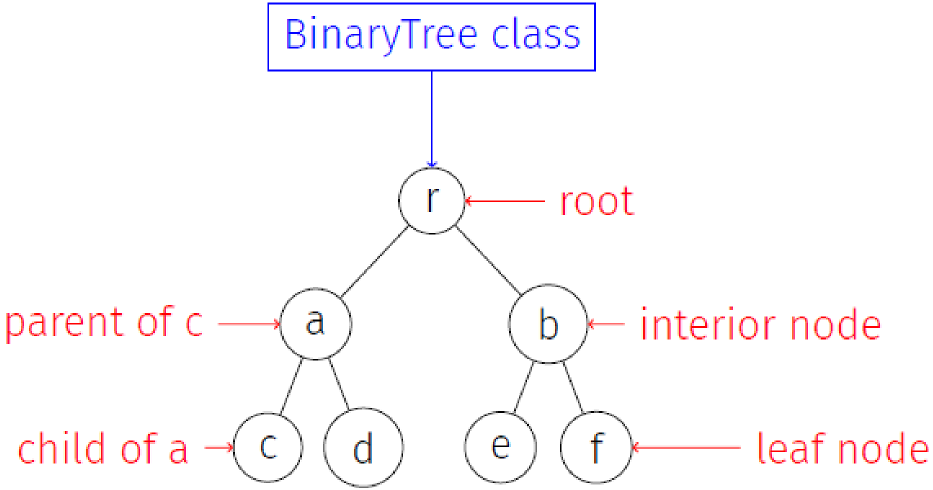
**Two-class strategy for implementing recursive data structures**

- A common design pattern: use one class for a *Tree/List*, another for *Nodes*
- **“Top” (tree) class**
  - Reference to “first” node
  - Methods and fields that apply to the entire data structure (i.e. the tree-object)
- **Node class**
  - Recursively defined: references to other node objects
  - Contains data stored at the node
  - Methods defined in this class are specific to this node or recursive (this node and its references)

# Binary Tree Classes

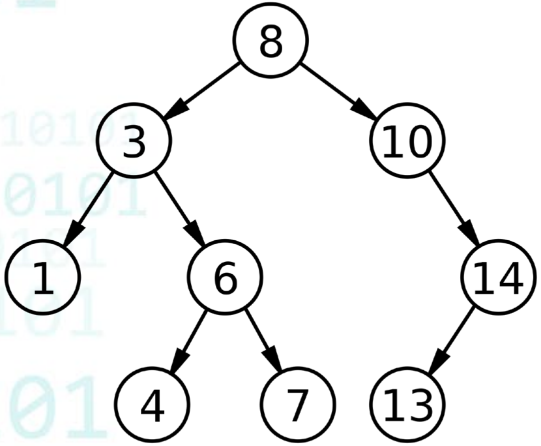
## Creating a simplified version of a binary tree

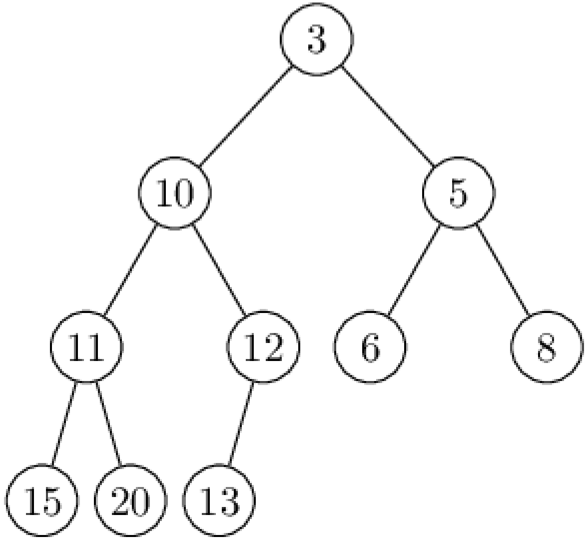
- class **`BinaryTree {…}`** – defines the tree
  - reference pointer to the **root node**
  - **methods:** *tree-level* operations (like `size()`)

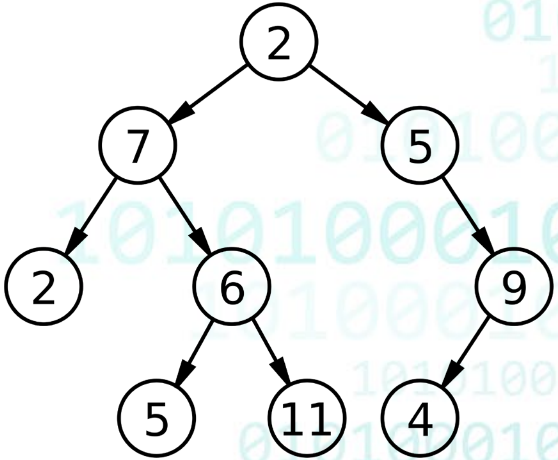


# Binary Trees

## Categorize the following trees



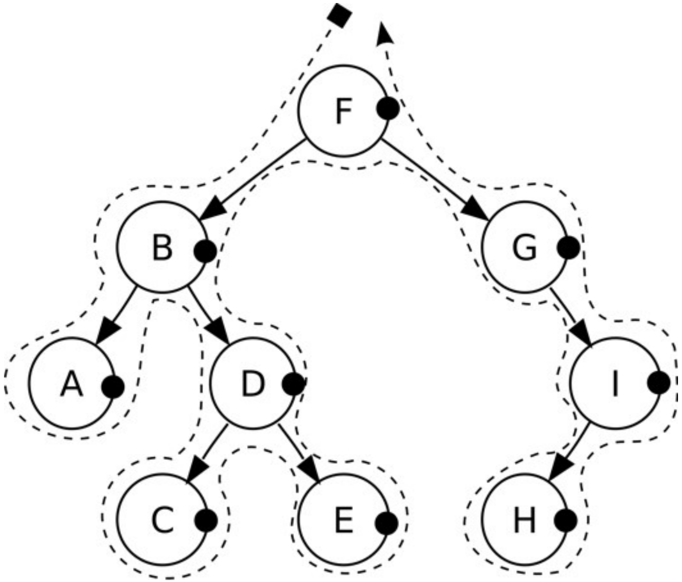




1. Binary Tree
2. Binary Heap
3. Binary Search Tree

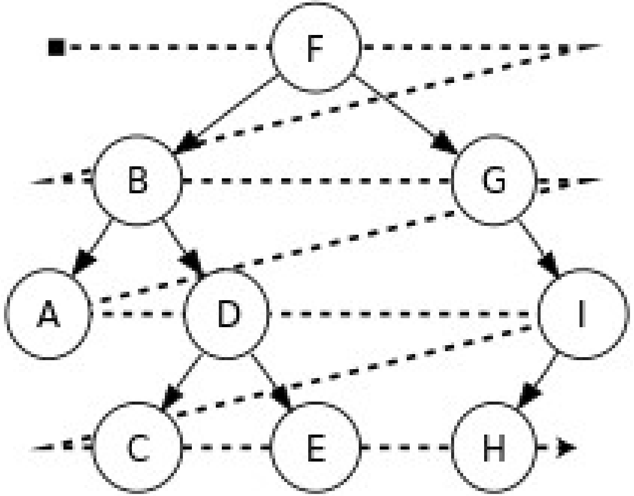
# Depth vs Breadth First

**Different ways to traverse a tree**

Depth-first




Breadth-first



# Depth-first Traversals

**"Order" defined by when the root of each subtree is visited**

- In **Pre-order,** the root is visited ***before (pre)*** the subtrees traversals
- In **In-order**, the root is visited ***in-between*** the left and right subtrees traversals
- In **Post-order,** the root is visited ***after (post)*** the subtrees traversals

**Pre-order Traversal**:
1. Visit the **root**
2. Traverse **left** subtree
3. Traverse **right** subtree

**In-order Traversal**:
1. Traverse **left** subtree
2. Visit the **root**
3. Traverse **right** subtree

**Post-order Traversal**:
1. Traverse **left** subtree
2. Traverse **right** subtree
3. Visit the **root**

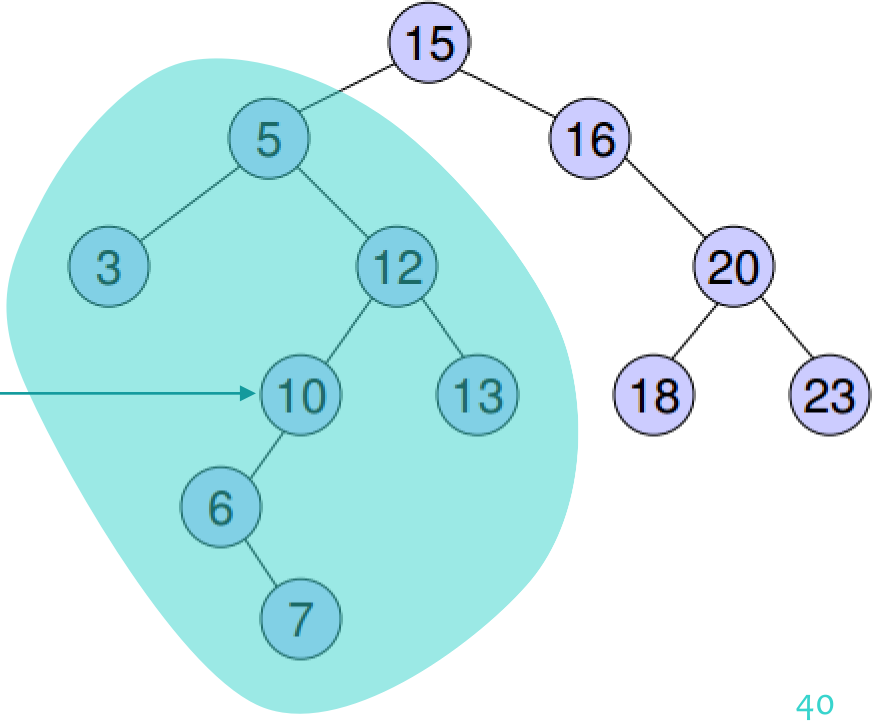
# Tree Traversals

12

## Type the post-order traversal of the highlighted subtree

3-7-6-10-13-12-5

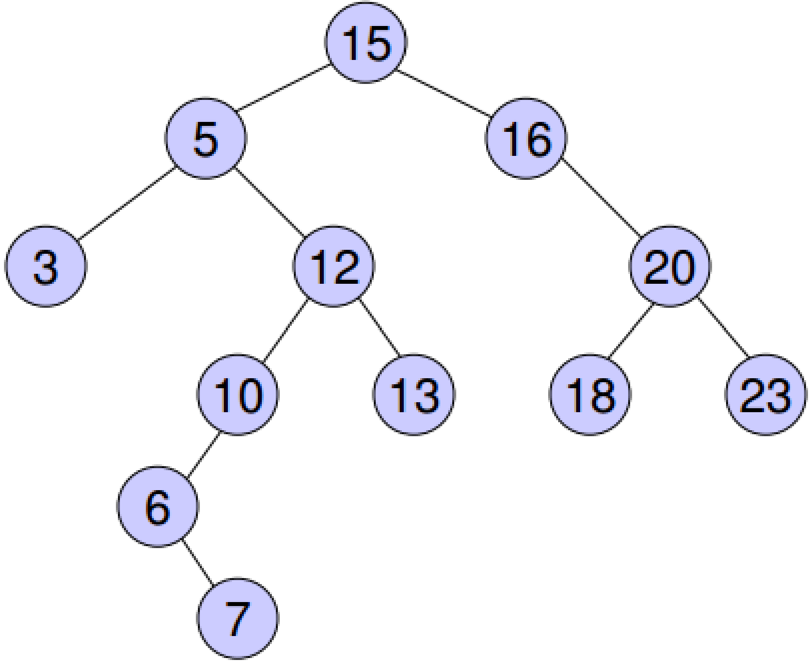
Don’t forget to apply the traversal *recursively* to every subtree!



# Binary Trees Traversal

## Example using the three depth-first traversal methods

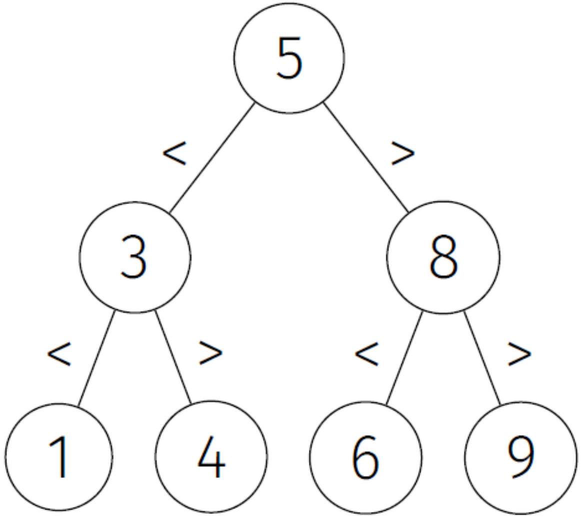
- **Pre-order (NLR):** (node, left, right)
  *15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23*
- **In-order (LNR):** (left, node, right)
  *3, 5, 6, 7, 10, 12, 13, 15, 16, 18, 20, 23*
- **Post-order (LRN):** (left, right, node)
  *3, 7, 6, 10, 13, 12, 5, 18, 23, 20, 16, 15*



# Binary Search Trees

## Binary Tree with comparable data values

- **Binary Search Tree (BST) properties:**
  - The data values of all *descendants* to the *left subtree* are **less** than the data value stored in the node
  - The data values of all *descendants* to the *right subtree* are **greater** than the value stored in the node
  - Follows a *Set implementation* and no duplicate values are allowed




- **BST requirement:**
  - The data variable should have type **Comparable**, not Object, in order for the data comparisons to work (and node values to be sorted)

# Binary Heaps

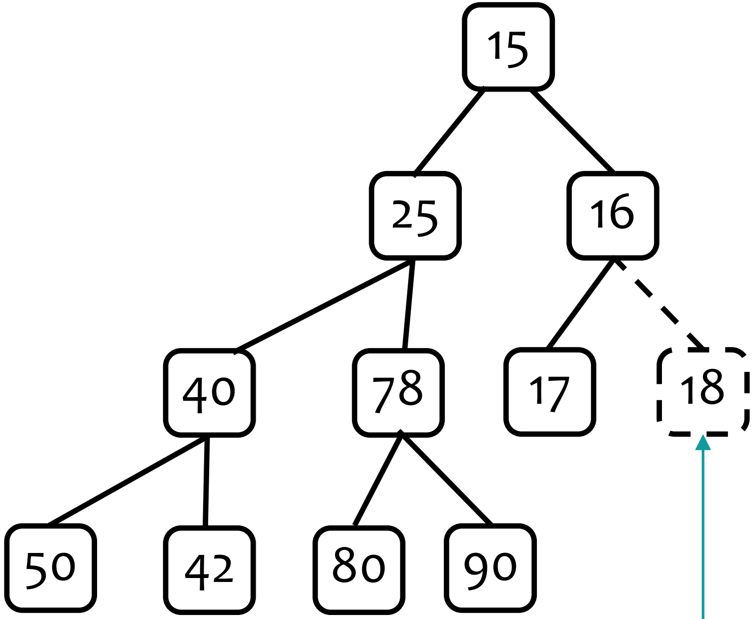
## An example of a balanced binary tree

- A **binary heap** is a heap data structure created using a binary tree
- It can be seen as a binary tree with two additional constraints:
    - **Shape property:**
        - A heap is a *complete binary tree*, a binary tree of height *i* in which all leaf nodes are located on level *i* or level *i-1* (must be *complete*), and all the leaves on level *i* are as far to the left as possible
    - **Order (heap) property:**
        - The data value stored in a node is less than or equal to the data values stored in all of that node's descendants
        - (Value stored in the root is always the smallest value in the heap)

# Minheap

## Is this a binary minheap?

No!



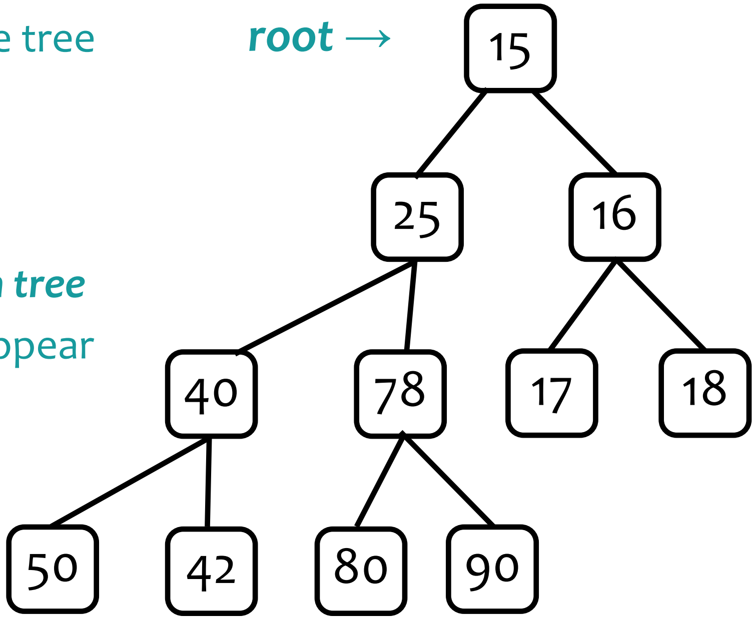

Every level from 1 to *i-1* *inclusive* should be **complete!**

# Minheap

## Storing values in ascending order

- The *smallest* value is the *root* of the tree
- All nodes are *smaller* than ALL its *descendants*

- **Note:** ***a heap is NOT a binary search tree*** – values larger than the root can appear on either side as children



# Binary Heap

## The most important methods on heaps

- The two most important *mutator* methods on heaps are:
  - (1) **inserting** a new value into the heap and
  - (2) **retrieving** the smallest value from the heap (in other words, *removing the root*)
- The **`insertHeapNode()`** method **adds** a new data value to the heap.
  - It must ensure that the insertion maintains both the *order* and *shape* properties of the heap
- The retrieval method, **`getSmallest()`**, **removes and returns** the smallest value in a *minheap*, which must be the value stored in the root
  - This method also rebuilds the heap because it removes the root, and all non-empty trees must have a root by definition

# Good luck!

Take a big breath and all will be good!