CS2110: SW Development Methods

Final Exam
~REVIEW~

Final Exam on Saturday, May 2, 2020

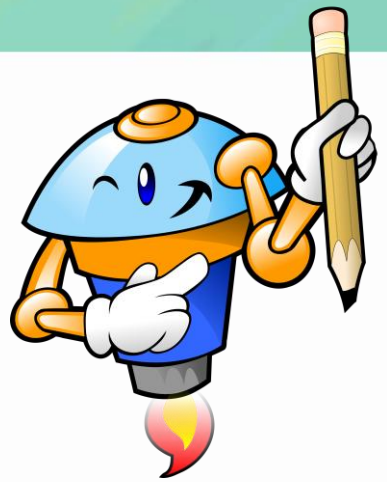
Spring 2020

Announcements

- Homework #6 – *Binary Search Trees*
  - Due: by 11:30pm on Tuesday, April 28, 2020
  - Remember to submit your **JUnit** tests
  - Web-CAT: 100% auto graded (score out of **100**)
- CS 2110 – Semester Course Evaluations
  - Everyone who completes them before Friday, May 1, at 11:59pm ET gets **1% extra credit** added their final grade
    - Still anonymous!
    - Only see a list of names

Similar to Exam 2, the Final Exam is:

- OPEN-book
- OPEN-notes
- OPEN-IDE (i.e. Eclipse or similar)
- OPEN-browser



- However…
  - You are **NOT** permitted to collaborate in any way
  - Completion of the test must represent your **individual** work
- If we find collaboration of ***any*** kind it will result in an automatic F / NC in the course (*unappealable*)

Exam Date, Duration, Location

- The fact that the final exam is given virtually, all students can start the exam ***any time*** on **Sat., May 2, 2020** (00:00:01am to 11:59:59pmET)

- You **must** COMPLETE the test **within two (2) hours**
  - The time you open the exam to the time you submit both parts of the exam should NOT exceed two (2) hours
  - To make use of the full time, don't start after 10pm

- This test is to be take at your own location

Exam Format

- **100% electronic** (virtual) – no in-class component at all
- All of the exam will be on **Collab** (under Quizzes) and the **Ford** code testing site (which you'll be linked to)
- The exam will automatically close at **11:59pm ET**!
- The format of the exam will consist of multiple-choice, fill-in-the-blank, short answer, and coding questions

- **You will need:**
  - A **laptop** (sufficiently charged to run for two hours) or a **desktop** computer
  - A stable and **reliable internet connection**

SDAC

- **SDAC** students please make sure your SDAC arrangements have been *finalized* with SDAC so we can provide you with appropriate accommodations


So, Any Tips For Success?


just breathe
You Can Do It!

Collab Tip…

- When taking the exam, **use only one tab on Collab!**

- Opening multiple tabs will cause issues with the active quiz (exam) and you will most probably lose your work and/or mess up the timer.

- We will not make special accommodations, if we realize you had multiple tabs trying to access other pages on Collab (e.g., prior Quizzes).
- (Suggestion: Collab material is freely downloadable!)

Important Stipulations (I)

- **Electronic timestamps** are used and recorded starting at the time you open the exam and the time you submit.
  - After completing and submitting your code for the **coding questions on the Ford website**, you will **receive a code**. You must place this code into the text field of the last question on the Collab quiz portion of the exam.
  - If no code is entered here, you will get a zero in the coding portion of the exam (no exceptions will be made under ANY circumstance.)
    So, plan accordingly to submit on **Ford** a few minutes *before* the time expires on **Collab**.

Important Stipulations (II)

- **Electronic timestamps** are used and recorded starting at the time you open the exam and the time you submit.
  - If the **time spent on Ford** is **more than 1 hour** after your start time, it will **not** be accepted and you will get a zero on the coding portion of the exam (no exceptions will be made under ANY circumstance.)
    So, plan accordingly to submit on Ford a few minutes before the time expires on Collab.

Test Taking Tips / Strategies

- Complete the higher-point value questions first
- Complete the questions you feel confident about first
- Toggle back and forth between questions you feel confident about, and questions you feel less confident about
- You can go back and forth between the Collab quiz questions and the Ford coding questions

- Keep a watch/clock next to you, try not to spend too much time on any one question
- Budget time to answer coding questions
- Budget time to copy the code received after submitting the coding questions into the text field of the last Collab question

Can we talk about what's on the exam?

You Should Be Able to Code/Do…

(Not exhaustive list)

- Write/read/trace **recursive** code
- A **node-level** operation for a **binary tree**
- A **tree-level** operation for a **binary tree**
- Perform **insert()** and **find()** methods on a **Binary Search Tree** (BST)
- Perform **tree traversal** (you may need to know how to code this, too - may be useful in assisting you with Tree coding question)
- Demonstrate the steps in **adding and removing from a Binary (Min) Heap** or **Binary (Max) Heap** (study BOTH! Once you know one, the other is similar, just opposite)
- Use multiple **threads** to solve problems
- Be able to recognize and understand code using **locks and/or condition objects** (not whole program from scratch)
- Write a general Java method – pay attention to appropriate data structure

Recursion and Tree Operations

- Recursive code for tree operations is simple, natural, elegant
- Example: **pseudo-code** in TreeNode

```
boolean find(Comparable target) { //find target
      Node next = null;
      if (this.data matches target) //found it!
            return true
      else if (target's data < this.data)
            next = this.leftChild //look left
      else
            next = this.rightChild //look right
      // 'next' points to left or right subtree
      if (next == null ) return false // no subtree
      return next.find(target) // search on
  }
```

Classes for Binary Trees

We will define **TWO** classes (a simplified version of a binary tree)

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

- class **BiniaryTreeNode {..}** – defines a node in the tree!
  - **data**: an object (usually of some **Comparable** type)
  - **left**: references root of left-subtree (or null)
  - **right**: references root of right-subtree (or null)
  - **parent**: this node's parent node (optional)
    - Could this be null? When should it be?
  - **methods**: node-level operations

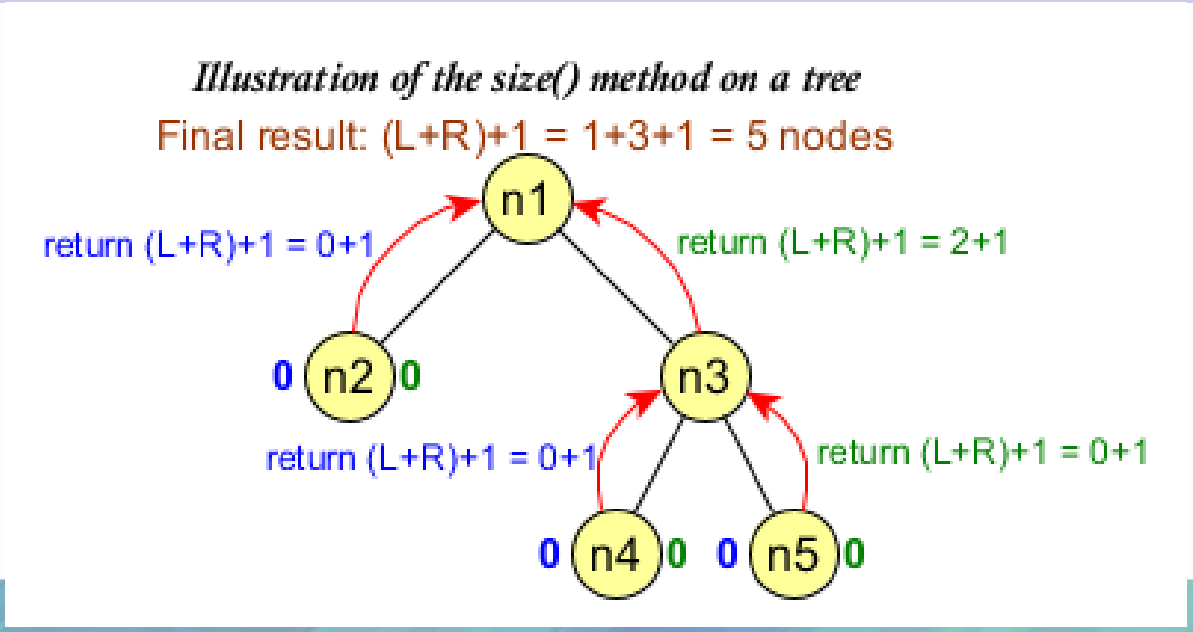
Size() method… [in Tree Class]

- **Tree-wide size()** should check for empty tree (root is null), then ask root for its size (call the node-version size() method on root)

```
public int size() {
    if (root == null) { // empty tree
        return 0; // size is zero
    }
    // otherwise, call size starting at
    // the ROOT of the tree
    return root.size(); // node level method
}
```

public int size( ) method [in Node Class]

- Initialize size variable to 0 (variable to keep track of # nodes)
- Check if current node has a **LEFT** child (not null)
  - If so, size = **size + left.size()**
- Check if current node has a **RIGHT** child (not null)
  - If so, size = **size + right.size()**
- Finally, **return size + 1** *(adding 1 to account for the current node)*



Size() method… [in Node Class]

- **Node-level size()** should count its children's sizes, add one for itself, and return the result (to be used by its parent)

```
public int size() {
    int size = 0;
    if(left != null) // there is a left subtree
        size += left.size(); // recursively call size() on left
    if(right != null) // there is a right subtree
        size += right.size(); // recursively call size() on right
    size += 1; // add one to account for current (this) node
    return size; // return
}
```

Cumulative Material (not exhaustive)

- Java keywords
- Object concepts
- Methods and method passing
- Data structures and how they are used / their applications
- ArrayLists, Arrays, Sets, Maps, Stacks, Queues, Trees, Heaps, etc
- Software development life cycle and concepts
- Run-time polymorphism
- Inheritance concepts
- Comparables / Comparators
- Algorithms (Big-O notation, given code find the complexity, asymptotic complexity, complexity classes)
- How exceptions are handled in code (including try-catch-finally)
- Event-driven programming (action listener and what it means to add an action listener to a component, purpose of actionPerformed() method, etc.)

Example: Writing compareTo()

- Imagine something like an entry in a **phonebook**
  - Order by last name, first name, then number

```
int compareTo(PhoneBookEntry item2 ) {
      int retVal= this.last.compareTo(item2.last);
      if ( retVal != 0 ) return retVal;
      retVal = this.first.compareTo(item2.first);
      if ( retVal != 0 ) return retVal;
      retVal = this.phNum - item2.phNum;
      return retVal;
}
```

The type is the type of the class! (Not "Object" like the equals() method!)

compareTo() for Strings!

Use of subtraction when dealing with numbers (a primitive) – will still be pos/neg/zero

PhoneBookEntry
last: String first: String phNum: int
compareTo(PhoneBookEntry, item) : int

compareTo() and various types

- Strings:
  - **compareTo()** with Strings uses alphabetical order to give you an "order" of Strings
  - Format: stringA.compareTo(stringB); // returns an int
- Numbers (ints) – e.g. sort students by score
  - Use **subtraction method** (*not* compareTo())
    - If "this.score" is 80 and "o.score" is 90
    - this.score – o.score is: 80-90 = -10 (negative)
  - This will sort student scores in ascending order (Question: how to sort in descending order??)
- Object /reference types: use **compareTo()** !

```
@Override
public int compareTo(Student o) {
    return this.score - o.score;
}
```

A Comparator Example: Dog class

```
public class CmpDogByBreedAndName implements Comparator<Dog> {
    // WAY 1: first by breed, then by name
    public int compare(Dog d1, Dog d2) {
      int retVal = d1.getBreed().compareTo(d2.getBreed());
      if (retVal != 0) return retVal;
      return d1.getName().compareTo(d2.getName());
    }
}

public class CmpDogByNameAndBreed implements Comparator<Dog> {
    // WAY 2:  first by name, then by breed
    public int compare(Dog d1, Dog d2) {
      int retVal = d1.getName().compareTo(d2.getName());
      if (retVal != 0) return retVal;
      return d1.getBreed().compareTo(d2.getBreed());
    }
}
```

Classes, subclasses, abstract classes, and interfaces

Type	Description
Class	Make a **CLASS** that does not extend anything (besides Object) when your new class ***does not pass the IS-A test*** for any other type
Subclass	Make a **SUBCLASS** (i.e., extend another class) ONLY when you need to make a ***more specific version*** of a class and need to override or add new behaviors
Abstract class	Use an **ABSTRACT CLASS** when you want to define a ***template (contract)*** for a group of subclasses, and you have at least some implementation code that all subclasses should use (not all abstract methods); also when you want to ensure that ***no objects of that type can be instantiated***
Interface	Use an **INTERFACE** when you want to define a ***role*** that other classes can play (i.e., subclass implements interface), regardless of where these classes are in the inheritance tree

Another Run-time Polymorphism Example: UVAperson

```
public class UVAperson {     // UVAperson class
      public UVAperson() { }; // constructor

      public department() { S.O.P("Work for a dept"); }
      public getAddress() { S.O.P("Mailing Address (UVA)"); }
      public getJobOffer() { S.O.P("Offered a job!"); }
}
      Note: "S.O.P" means "System.out.println"
```

```
public class Student extends UVAperson {   // UVAperson is superclass
      public Student() { }; // constructor

      public department() { S.O.P("Student-This is my major!"); }
      public enroll() { S.O.P("Student-Took a class!"); }
}
```

```
public class UgradSt extends Student { // Student is superclass
      public UgradSt() { }; // constructor

      public getJobOffer() { S.O.P("Ugrad-Got a Job!"); }
}
```

In main somewhere (e.g. in UVA_person):

```
      UVAperson s1  = new Student();
      Student   uG1 = new UgradSt();
      UVAperson uG2 = new UgradSt();
```

QUESTIONS:

```
s1.department(); // Student(specific), so "Student-This is my major!"
uG2.enroll();    // ILLEGAL - UVAperson doesn't have enroll()!
s1.enroll();     // ILLEGAL - UVAperson doesn't have enroll()!
s1.getAddress();// "Mailing Address (UVA)" - doesn't exist in Student so in UVAp
uG1.getJobOffer(); // "Ugrad-Got a job!" - Student inherits mthd, Ugrad version used
uG2.department(); // "Student-This is my major!" - UVAp has mthd, Student overrides it
                     - Ugrad inherits Student.department()
uG1.getAddress(); // "Mailing Address (UVA)" - UgradSt inherits from UVAperson
```

Run-time Polymorphism

Many forms: adapting behavior during run-time

Light-morph jaguar

Dark-morph or melanistic

Calling the getColor() method on two Jaguar objects

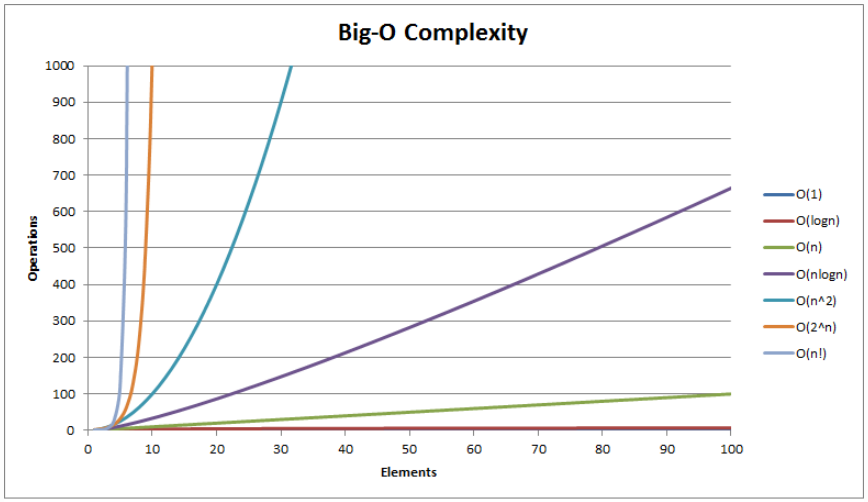
```
class LJaguar extends Jaguar {
     public String getColor() {
          return "I'm the light-morph one!";    }
}

class BJaguar extends Jaguar {
     public String getColor() {
          return "I'm the melanistic one!";   }
}

Jaguar j1 = new LJaguar();
Jaguar j2 = new BJaguar();
System.out.println(j2.getColor())
```

Who are you?

Algorithms

- Order classes group "equivalently" efficient algorithms
  - Algorithms that grow at the SAME RATE

Class	Name	Example
O(1)	Constant time	Input size doesn't matter
O(log n)	Logarithmic time (Very efficient)	Binary search (on sorted list)
O(n)	Linear time	Linear search (unsorted list)
O(n log n)	Log-linear time	Best sorting algorithms
O(n^2)	Quadratic time	Poorer sorting algorithms
O(n^3)	Cubic time	3 variables equation solver
O(2^n)	Exponential time	Many important problems, often about optimization
O(n!)	Factorial time	Find all permutations of a given string/set



Increasing Complexity

Final Exam Topics

Abstract Data Types

- What is an ADT?
- **Stacks**
  - LIFO
  - Common stack operations: push(), pop(), peek()
- **Queues**
  - FIFO
  - Common queue operations: add(), remove()

Concurrency

- Understanding of what is a thread and how to use them.
- Understand what a race condition is and ways to avoid them.
  - What are locks? How do they work? Where are they placed in the code?
- Be able to describe a deadlock as well as how to avoid them.
  - What are condition objects? How do they work? How do they work with locks? Where are they placed in the code?

```
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;  // lock
    private Condition sufficientFundsCondition; // Add condition object


    /**
      Constructs a bank account with a zero balance
    */
    public BankAccount(){
        balance = 0;
        balanceChangeLock = new ReentrantLock(); // lock
        // condition object associated with specific
        // lock object (balanceChangeLock)
        sufficientFundsCondition = balanceChangeLock.newCondition();
```

```
public void deposit(double amount){

    balanceChangeLock.lock();  // lock!
    try
    {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
        // Funds added to balance...
        // Unblock other threads waiting on the condition by "signalAll"
        sufficientFundsCondition.signalAll();
    }
    finally
    {
        balanceChangeLock.unlock(); // unlock!
    }
}
```

```
public void withdraw(double amount) throws InterruptedException {

    balanceChangeLock.lock();  // lock!
    try
    {
        // Check condition:
        while (balance < amount) {
            // If balance is less than withdrawal amount...
            // Condition object calls "await"
            sufficientFundsCondition.await();   // await!
            // Another thread can now acquire the lock object
        }
        System.out.print("Withdrawing " + amount);
        double newBalance = balance - amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
    finally
    {
        balanceChangeLock.unlock(); // unlock!
    }
}
```

Recursion

- Definitions, concepts, use in mathematics
- How to write recursive code
- How to read recursive code
- Asymptotic analysis of recursive code. This topic is not on exam, nor is Section 5.5.2 of the MSD book which covers this.

Recursion (Reminder)

- **Recursion** breaks a difficult problem into one or more simpler versions of itself
- A definition is **recursive** if it is defined in terms of itself
- Questions to ask yourself:
  - How can we **reduce** the problem into smaller version of the same problem?
  - How does each call make the problem **smaller**?
  - What is the **base case?**
  - Will we always **reach the base case?**

Definitions

- Base case
  - The case for which the solution can be stated non-recursively (*solved directly*)
- Recursive case
  - The case for which the solution is expressed in terms of a smaller version of itself

```
public int factorial(int n) {
    // base case (always first)
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Recursion can be Tricky! [Summary]

- Always put the base case (or base cases) FIRST
- The base case (or base cases) should eventually be reached (happen) given *any* input
- The recursive call must break the problem down into a smaller version of itself (making progress towards the goal)
- A recursive solution may not always be the best solution!

Recursion

- Implementing basic recursive methods in programming languages
  - Three parts of recursion (a base case, a recursive case, and that the recursive case makes progress towards the base case). Also, don't forget that the base case(s) should come before the recursive call(s)
  - Activation records and the run-time stack (see page 351)
  - Simple examples: factorial, Fibonacci, binary search
- When recursion is not a good idea?
  - Repeated sub-problems e.g. Fibonacci
  - Bottom-up approach that remembers small solutions better
- Recursive binary search

Recursion vs. Iteration

Recursion

```
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Iteration

```
public int factorial(int n) {
    int fact_n = 1;

    for (int i = 1; i <= n; i++){
        fact_n = fact_n * i;
    }
    return fact_n;
}
```

Build solution from ***top down***

Build solution from ***bottom up***

Exercise: Trace Mergesort Execution

- Can you trace MergeSort() on this list? *(**even** # of elements)*

A = {8, 3, 2, 9, 7, 1, 5, 4}; ← original list; to be sorted (8 *elements*)

8, 3, 2, 9 7, 1, 5, 4	← ***divide*** into 2 lists of 4
8, 3 2, 9 7, 1 5, 4	← ***divide*** 2 lists of 4 into 4 lists of 2
8, 3, 2, 9, 7, 1, 5, 4	← ***divide*** into **SINGLE** items
3,8 2,9 1,7 4,5	← ***merge*** single items into pairs
2,3,8,9 1,4,5,7	← ***merge*** 4 pairs into 2 lists of 4
{1,2,3,4,5,7,8,9}	← ***merge*** 2 lists of 4 into 1 list (Result)

Exercise: Trace Mergesort Execution

- Can you trace MergeSort() on this list? *(**odd** # of elements)*

A = {8, 3, 2, 9, 7, 1, 5, 4, 6}; ← original list; to be sorted (9 *elements*)

8, 3, 2, 9 7, 1, 5, 4, 6 ← ***divide*** two lists are not even (ok!)

8, 3 2, 9 7, 1 5, 4 6 ← ***divide*** into pairs + 1

8, 3, 2, 9, 7, 1, 5, 4, 6 ← ***divide*** into **SINGLE** items

---

3,8 2,9 1,7 4,5,6 ← ***merge*** single items into pairs + 3

2,3,8,9 1,4,5,6,7 ← ***merge***

{1,2,3,4,5,6,7,8,9} ← ***merge*** into 1 list (Result)

Efficiency of Mergesort

- ***Mergesort*** is **O(n lg n)**

  same order-class as the most efficient sorts (quicksort and heapsort)
- It **is** more efficient than Selection Sort, Bubble Sort, and Insertion Sort
- Wait for CS2150 and CS4102 to study efficiency of this and other recursive algorithms!
- The Divide and Conquer approach matters, and in this case, is a "win!"

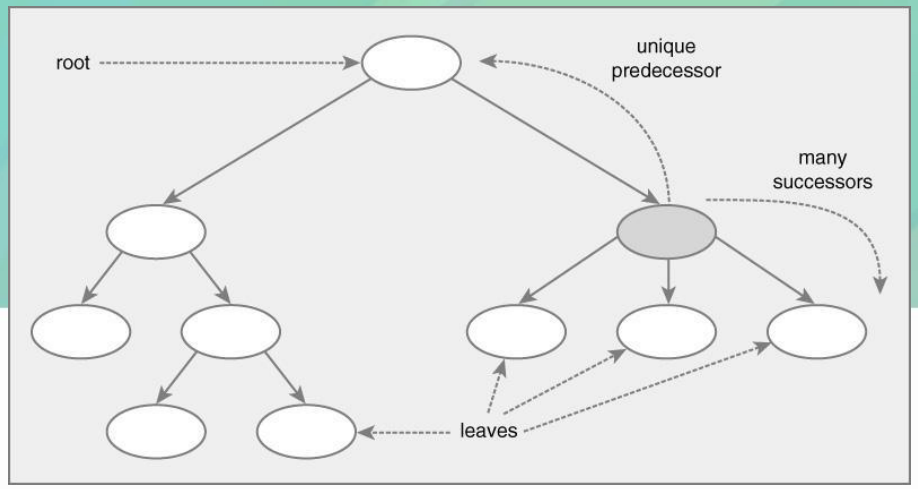
Data Structures // Linked Lists

- The ideas of abstract data types, data structures, code implementations, and how they differ
- **Linked Lists:** definitions and common terms
  - Note recursive definition of a linked list
  - Linked lists can be implemented both as a **stack and as a queue**
  - Understand what goes into the Node class and the List class
  - Operations defined for linked lists
    - **push() and pop() methods** (linked list implementation of a stack)
    - **add()/enqueue() and remove()/dequeue() methods** (linked list implementation of a queue)
    - How to implement methods like **toString(), find(), contains()**
    - **get(int index) method, add(int index, int value)** - adding a value in the middle of a linked list (at position 'index')

Data Structures // Trees

- **Trees**: definitions and common terms
  - Note recursive definition using sub-trees
  - Ability to draw trees and discuss illustrations of them
- **Binary Trees**
  - What operations might we define for binary trees
  - Java classes for tree-nodes and for tree (using code demoed in class). Study the fields and the methods we covered in class, in particular the following things:
    - In the TreeNode class, the recursive methods find(), size() and height()
    - In the BinTree class, see how calls of the same name as the above methods are coded. (It's simple: just call the method with the same name on the root. May need to check if there is a root node

Trees



- Trees are composed of:
  - **Nodes**
    - Elements in the data structure (hold data)
    - Only one parent (***unique predecessor***)
    - Zero, one, or more children (***successors***)
    - **LEAF** nodes: nodes without children (***terminal***)
    - **ROOT** node: **top** or start node; with no parent
    - **INTERNAL** node: notes with children (***non-terminal***)
    - Measure of **DEGREE**: how many children
  - **Edges**
    - Link parent node with children node (if applicable)
- The **HEIGHT** of a tree is the longest path (# nodes) from root to leaf

Trees: 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
- For these, it's natural to have recursive algorithms
- **Tree operations may come in two flavors:**
  - **NODE-SPECIFIC** (e.g. `hasParent()` or `hasChildren()`)
  - **TREE-WIDE** (e.g. `size()` or `height()`) – requires tree traversal

Classes for Binary Trees

We will define **TWO** classes (a simplified version of a binary tree)

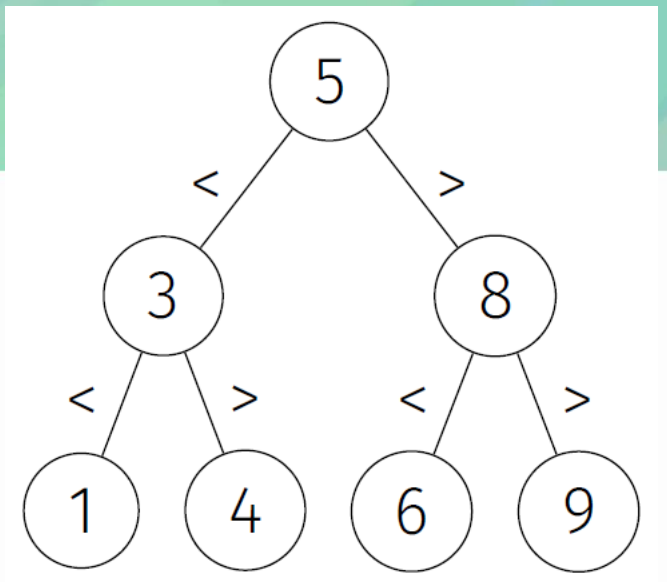
- class **BinaryTree {..}** – defines the tree
  - reference pointer to the **root node**
  - methods: tree-level operations (like size())

- class **BiniaryTreeNode {..}** – defines a node in the tree!
  - **data**: an object (usually of some **Comparable** type)
  - **left**: references root of left-subtree (or null)
  - **right**: references root of right-subtree (or null)
  - **parent**: this node's parent node (optional)
    - Could this be null? When should it be?
  - **methods**: node-level operations

Data Structures // Trees

- **Binary Search Tree (BST)**
  - How different than "plain" or "just" binary trees. (Answer: nodes are stored in a particular order based on their key value.)
  - How this is done in Java using the **Comparable interface** for nodes.
  - The important methods for BSTs: **insert()**, **find()**
  - How insert knows where to place an object

Binary Search Trees

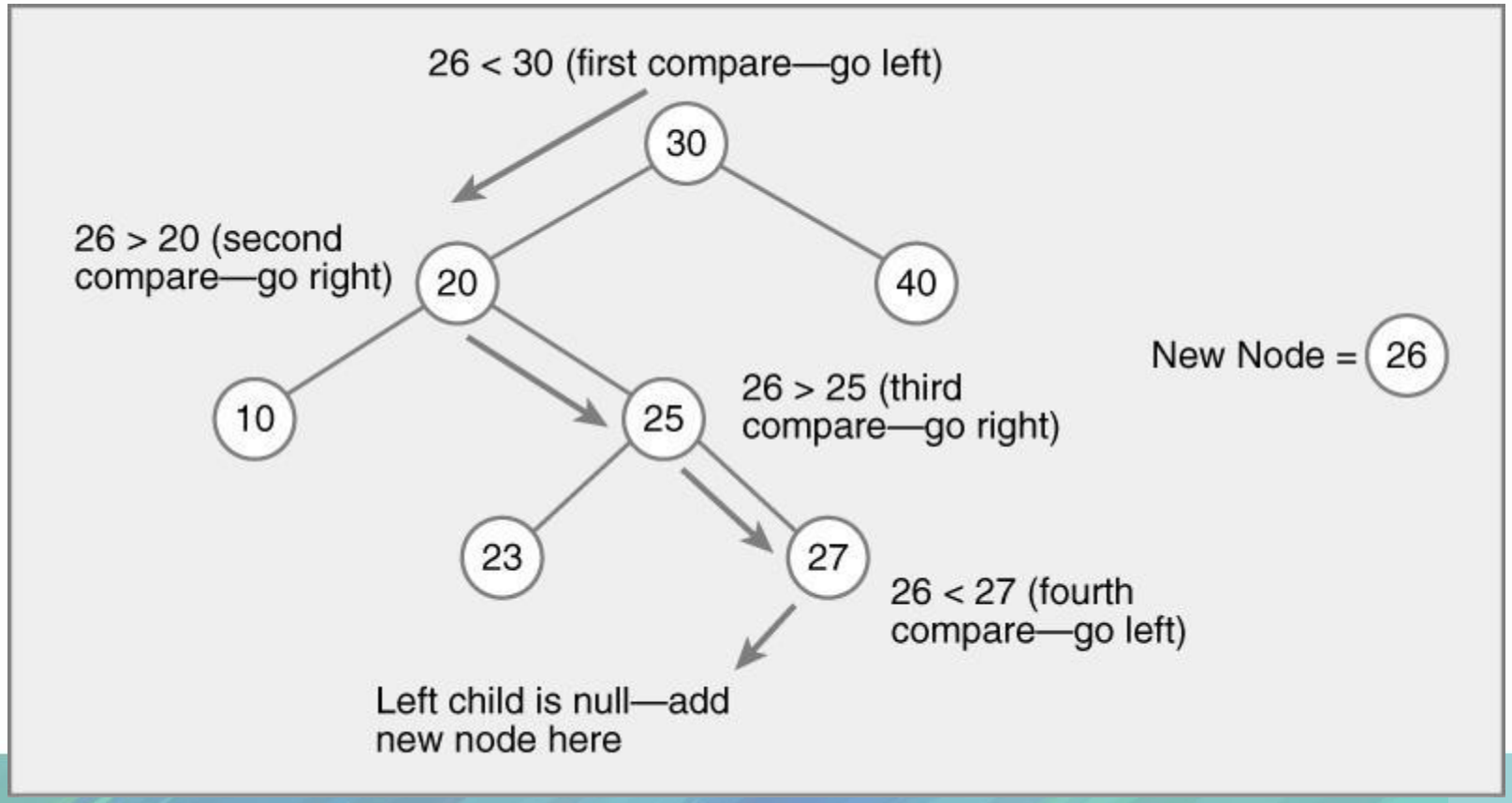


- Binary tree with **comparable** data values

- **Binary search tree (BST) property:**
  - The data values of all *descendants* to the **left** *subtree* are **less** than the data value stored in the parent node, and
  - The data values of all *descendants* to the **right** *subtree* are **greater** than the data value stored in the parent node
- BST requirement:
  - The data variable should be a **Comparable** type (not Object) in order for the data comparisons to work

Find and Insert in BST

- **Find**: look for where it should be
- If not there, that's where you **insert**



BST Find and Insert

- **Find** an element in the tree
  - Compare with root, if less traverse left, else traverse right; repeat
  - Stops when found or at a leaf
  - Sounds like binary search!
  - Time complexity: **O(log n)**, worst case height of the tree
- **Insert** a new element into the tree
  - Easy! Do a **find** operation. At the leaf node, add it!
  - Remember: add it to the correct side (left or right)

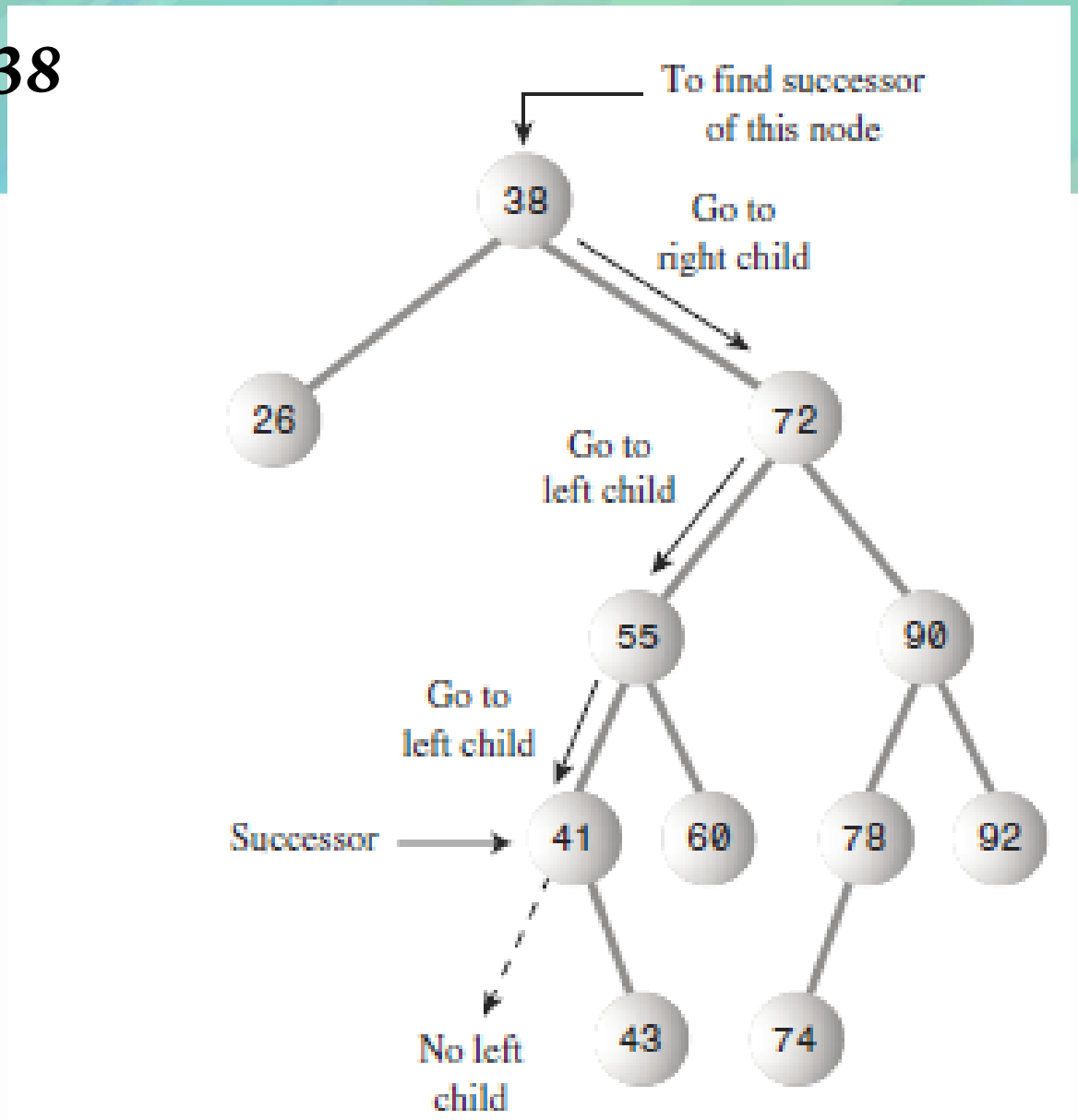
Recursion and Tree Operations

- Recursive code for tree operations is simple, natural, elegant
- Example: **pseudo-code** in TreeNode

```
boolean find(Comparable target) { //find target
      Node next = null;
      if (this.data matches target) //found it!
            return true
      else if (target's data < this.data)
            next = this.leftChild //look left
      else
            next = this.rightChild //look right
      // 'next' points to left or right subtree
      if (next == null ) return false // no subtree
      return next.find(target) // search on
  }
```

Find Successor of 38

- *Minimum* of **right** subtree (*leftmost node*)
- Which is the next largest number



Finding the successor.

Data Structures // Trees

- **Tree traversal** (in-order, pre-order, and post-order traversals)
  - Be able to perform each of these traversals given a binary tree
  - Properties of each kind of traversal / when would you use one over the other, etc.
- **Heaps (Binary Heap)**
  - Heap definition
  - Minheap / Maxheap (understand how to construct and remove from BOTH -- you are responsible for both minheaps and max heaps)
  - Heap in memory: one-dimensional array
  - Application of heaps
  - Methods for heap: adding and removal

Preoder, Inorder, Postorder

- In Preorder, the root is visited **before** (pre) the subtrees traversals
- In Inorder, the root is visited **in-between** left and right subtree traversal
- In Postorder, the root is visited **after** (post) the subtrees traversals

Preorder Traversal:
1. Visit the **root**
2. Traverse **left** subtree
3. Traverse **right** subtree

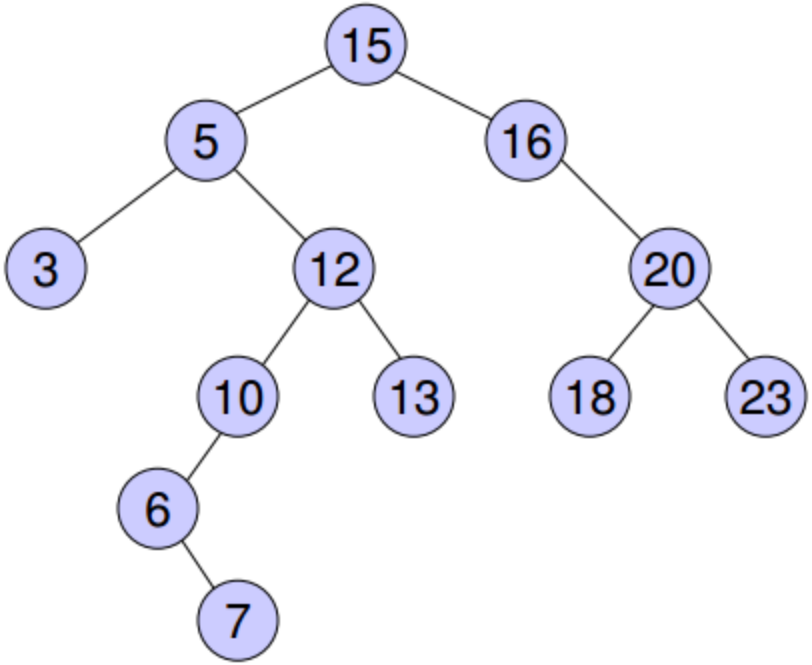
Inorder Traversal:
1. Traverse **left** subtree
2. Visit the **root**
3. Traverse **right** subtree

Postorder Traversal:
1. Traverse **left** subtree
2. Traverse **right** subtree
3. Visit the **root**

Tree Traversal Example [*3 methods*]

Let's do an example first…

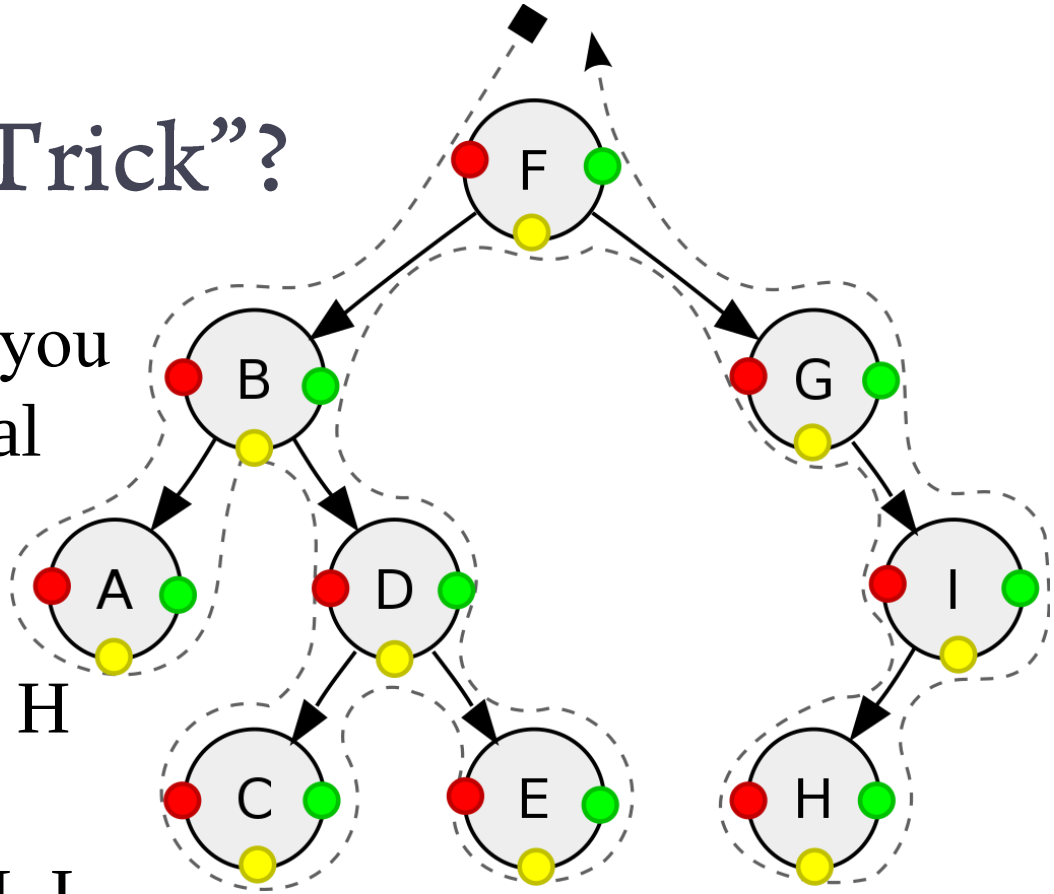
(Notice: this is a Binary Search Tree!)



- **pre-order:** (root, left, right)
  15, 5, 3, 12, 10, 6, 7,
  13, 16, 20, 18, 23
- **in-order:** (left, root, right)
  3, 5, 6, 7, 10, 12, 13,
  15, 16, 18, 20, 23
- **post-order:** (left, right, root)
  3, 7, 6, 10, 13, 12, 5,
  18, 23, 20, 16, 15

Tree Traversal "Trick"?

- Here's a trick to help you remember the traversal methods:
- ***pre-order*** *(red)*:
  F, B, A, D, C, E, G, I, H
- ***in-order*** *(yellow)*:
  A, B, C, D, E, F, G, H, I
- ***post-order*** *(green)*:
  A, C, E, D, B, H, I, G, F



Picture credit: Pluke, Miles, and Jochen Burghardt (overlay)

Heaps ("binary heaps")

- 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), 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)

Binary Heap

- 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 the heap, which must be the value stored in the **root**. This method also **rebuilds the heap** because it removes the root, and all nonempty trees must have a root by definition

Inserting a node into a Heap

- Add the element to the ***bottom level* of the heap** – *maintaining the* **shape property**
- Compare the added element with its parent; if they are in the correct order, stop
- If not, **swap** the element with its parent and return to the previous step (the parent must be less than or equal to its children – *maintaining the* **order property**)
- The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property
- **Time complexity:** $\mathbf{O(\log n)}$

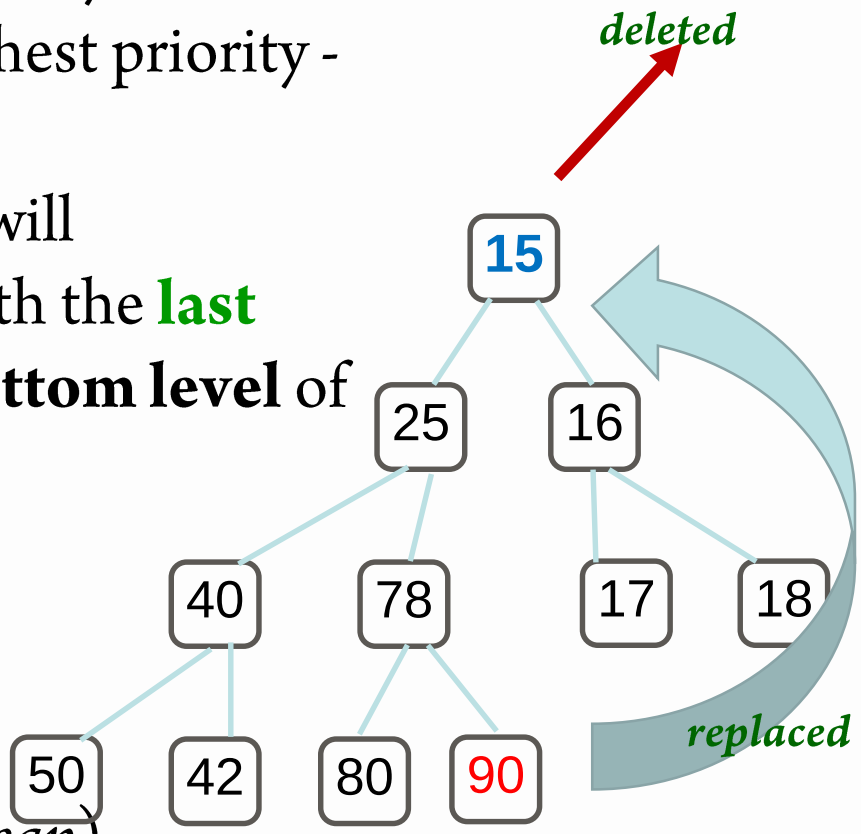
Deleting a node from a Heap

- Replace the **root** of the heap with the ***last element on the last level*** – *maintaining the* **shape property**
- Compare the new root with its children; if they are in the correct order, stop
- If not, **swap** the element with one of its children and return to the previous step. (Swap with its *smaller* child in a min-heap and its *larger* child in a max-heap – *maintaining the* **order property**)
- In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree. **Time complexity:** $O(\log n)$
- [When retrieving the *smallest element*, we delete the root node]

Removing an element from a heap

- For a **priority queue**, you always remove the least value element (highest priority - think priority #1)
- In this heap, **15** is least, we will **remove** it and **replace** it with the **last node on the right** at the **bottom level** of the heap (90)

  Note: no other node is appropriate to initially replace 15!
- When retrieving the *smallest element*, we delete the root node (*min heap*)



YOU GOT THIS!! ☺

You Got This! ☺