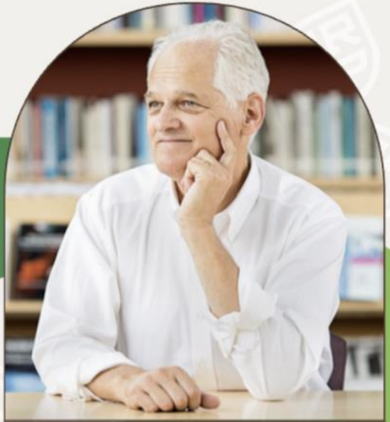
CNTR:
Center for Technological Responsibility, Reimagination, Redesign

# Fireside Chat : “AI Policy In The Next 4 Years”



Dr. Nicholas Ziegler
(International and Public Affairs)

Dr. Suresh Venkatasubramanian
(Directer of CNTR)

Join us for a post-election Fireside Chat, where we'll discuss the future directions of tech policy in a 45-minute conversation with expert insights and lively Q&A—don’t miss it!

**RSVP by scanning the code!** →

**November 13th, 4-5:30 PM**

**Joukowsky Forum @
Old Watson Institute**

In conversation with Emma Huang ‘25
(CNTR Undergraduate Coordinator)

# We want to hear about **your** high school digital safety education experience!

**Scan me!**

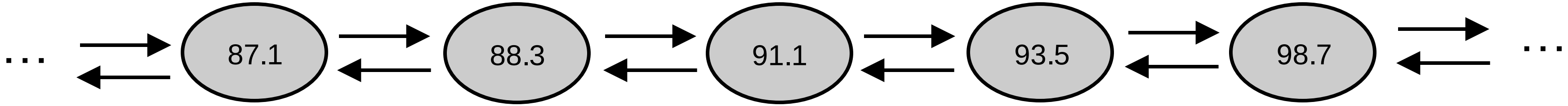
Help CS1953A students explore:

- The **current state of digital safety education** in U.S. high schools
- The **prevalence of tech-facilitated gender-inequities** (e.g. nonconsensual intimate imagery, explicit deepfakes, reproductive health surveillance)

# Doubly Linked List (1/3)

- Is there an easier/faster way to get to previous node while removing a node?
  - with Doubly Linked Lists, nodes have references both to next and previous nodes
  - can traverse list both backwards and forwards – Linked List still stores reference to front of list with `head` and back of list with `tail`
  - modify `Node` class to have *two* pointers: `next` and `prev`
  - eliminates pointer-chasing loop because `prev` points to predecessor of every `Node`, at cost of second pointer
    - classic space-time tradeoff!

# Doubly Linked List (2/3)




- For Singly Linked List, processing typically goes from first to last node, e.g. `search`, finding place to insert or delete

- Sometimes, particularly for sorted list, need to go in the opposite direction
  - e.g., sort CS15 students on their final grades in ascending order. Find lowest numeric grade that will be recorded as an "A". Then ask: who has a lower grade but is closer to the "A" cut-off, i.e., in the grey area, and therefore should be considered for "benefit of the doubt"?

# Doubly Linked List (3/3)

- This kind of backing-up can't easily be done with the Singly Linked List implementation we have so far
    - could build our own *specialized* `search` method, which would scan from the `head` and be, at a minimum, O(n)

- It is simpler for Doubly Linked Lists:
    - find student with lowest "A" using search
    - use `prev` pointer, which points to the predecessor of a node (O(1)), and back up until hit end of B+/A- grey area

# Lecture 19

## Stacks, Queues, and Trees

# Stacks and Queues


Abstractions that are Wrappers for `MyLinkedList`

# Outline

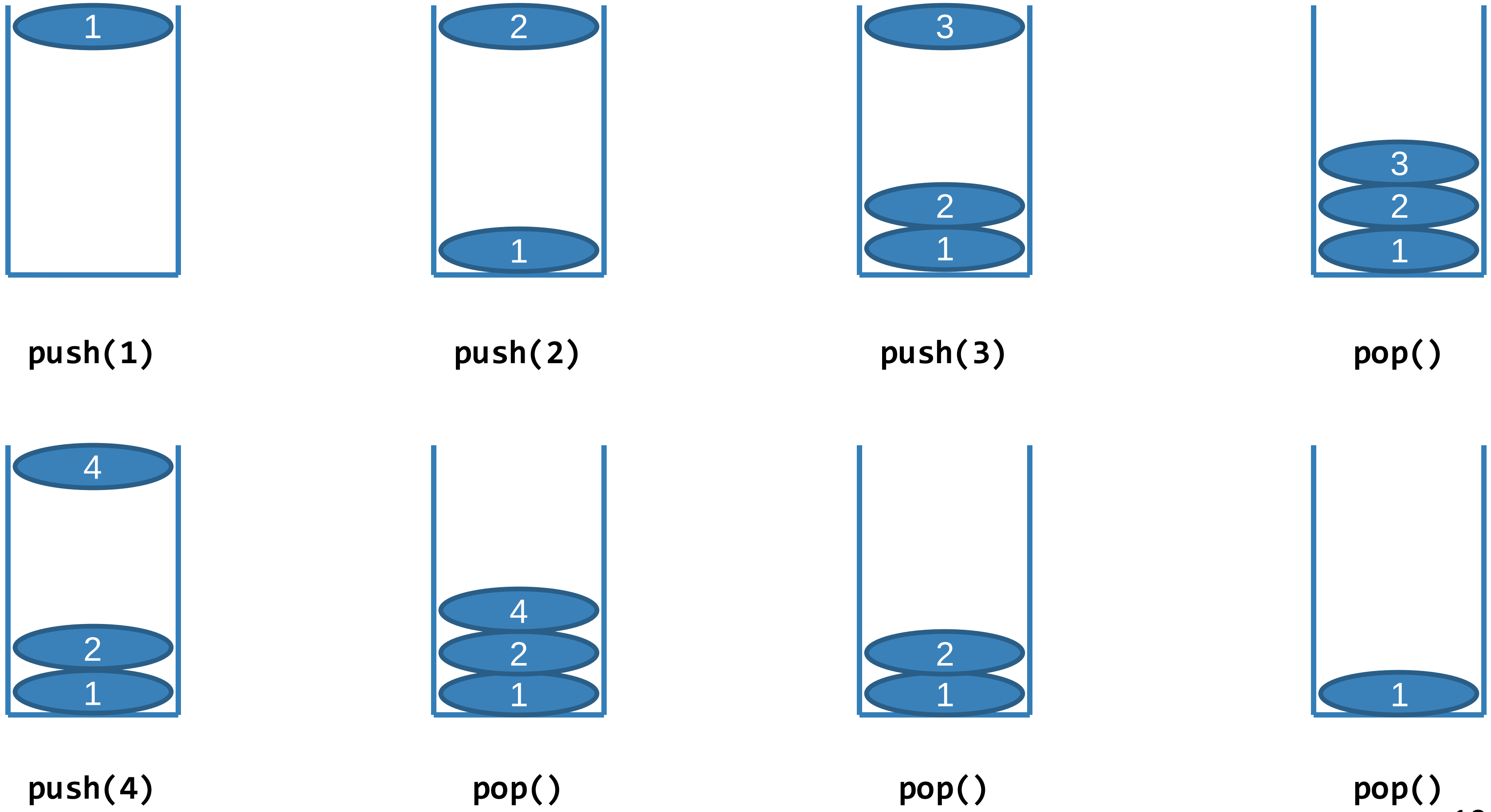
- Stacks and Queues
- Trees

# Stacks

- `Stack` has special methods for insertion and deletion, and two others for size
  - `push` and `pop`
  - `isEmpty, size`
- Instead of being able to insert and delete nodes from anywhere in the list, can only add and delete nodes from top of `Stack`
  - **L**IFO (**L**ast **I**n, **F**irst **O**ut)
- We'll implement a stack with a linked list

# Methods of a Stack

- Add element to top of `stack` — `public void push(Type el)`
- Remove element from top of `stack` — `public Type pop()`
- Returns whether `stack` has any elements — `public boolean isEmpty()`
- Returns number of elements in `stack` — `public int size()`



1
push(1)
2
1
push(2)
3
2
1
push(3)
3
2
1
pop()
4
2
1
push(4)
4
2
1
pop()
2
1
pop()
1
pop()

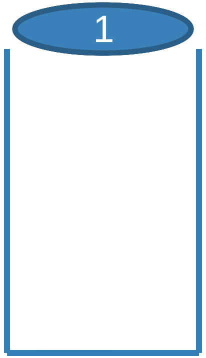
# Stack Constructor

```
public class Stack<Type> {
    private MyLinkedList<Type> list;
    public Stack() {
        this.list = new MyLinkedList<>();
    }
    /* other methods elided */
}
```

- When generic `Stack` is instantiated, it contains an empty `MyLinkedList`
- When using a stack, you will replace the generic `Type` with type of object your `Stack` will hold – enforces homogeneity
- Note: `Stack` uses classic “wrapper” pattern to modify functionality of the wrapped data structure, `MyLinkedList`, and to add other methods

# Implementing Push

```
//in the Stack<Type> class ...
public Node<Type> push(Type newData) {
    return this.list.addFirst(newData);
}
```

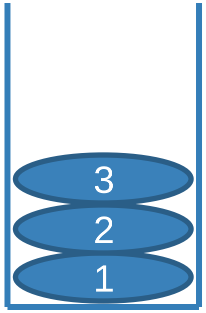



- Let’s see behavior...
- When element is `push`ed, it is always added to front of list
- Thus, `Stack` delegates to the `MyLinkedList`, `this.list` to implement `push`

# Implementing Pop

- Let's see what this does...
- When popping element, it is always removed from top of `Stack`, so call `removeFirst` on `MyLinkedList` – again, delegation
- `removeFirst` returns element removed, and `Stack` in turn returns it
- Remember that `removeFirst` method of `MyLinkedList` first checks to see if list is empty

```
//in the Stack<Type> class ...
public Type pop() {
    return this.list.removeFirst();
}
```



# isEmpty

- Stack will be empty if the MyLinkedList, list, is empty - delegation
- Returns true if Stack is empty; false otherwise

```
//in the Stack<Type> class ...
public boolean isEmpty() {
    return this.list.isEmpty();
}
```

# size

- Size of `Stack` will be number of elements that the `MyLinkedList, list` contains – delegation
- Size is updated whenever `Node` is added to or deleted from `list` during `push` and `pop` methods

```
//in the Stack<Type> class ...
public int size() {
    return this.list.size();
}
```

# TopHat Question

Look over the following code:

```
Stack<HeadTA> myStack = new Stack<>();
myStack.push(htaSarah);
myStack.push(htaGrace);
myStack.pop();
myStack.push(htaKarim);
myStack.pop();
```

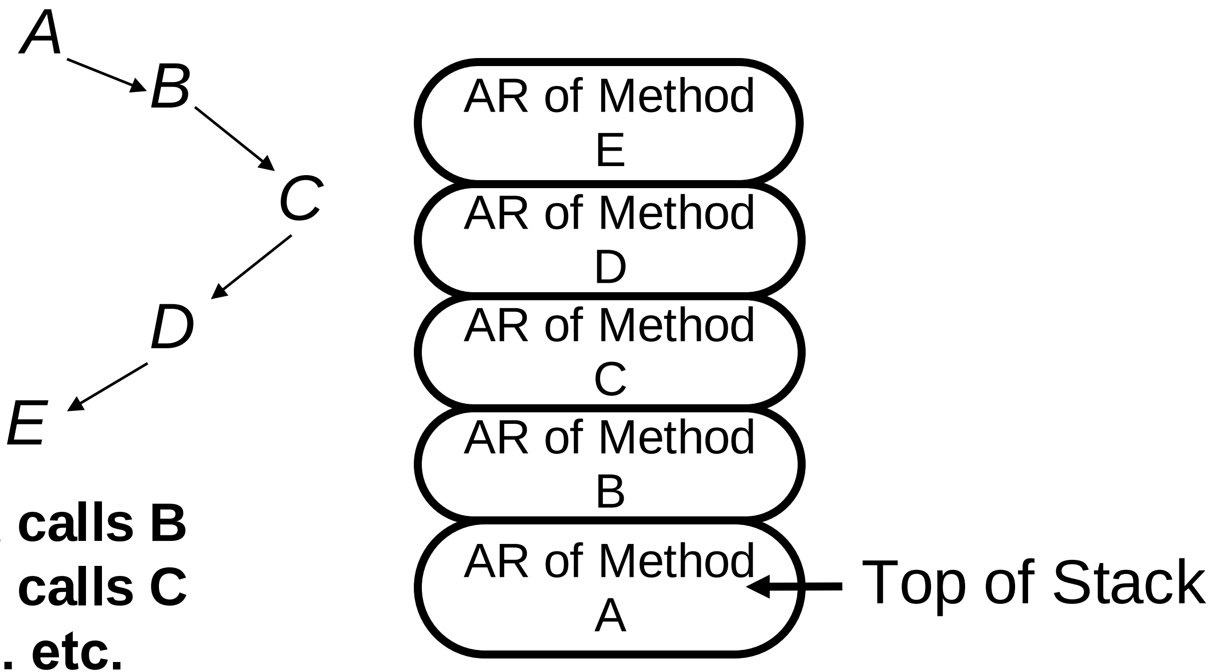
Who’s left in the stack?

A. `htaSarah`
B. `htaGrace`
C. `htaKarim`
D. none of them!

# Example: Execution Stacks

- Each method has an Activation Record (AR) – recall recursion lecture
  - contains execution pointer to next instruction in method
  - contains all local variables and parameters used by method
- When methods execute and call other methods, Java uses a `Stack` to keep track of the order of execution: "stack trace"
  - when a method calls another method, Java adds activation record of called method to `Stack`
  - when new method is finished, its AR is removed from `Stack`, and previous method is continued
  - method could be different or a recursively called clone, when execution pointer points into same immutable code, but different values for variables/parameters

# Execution Stacks




**A calls B**
**B calls C**
**… etc.**

When E finishes, its AR is popped. Then D’s AR is popped, etc. Note this handles the tracking of invocations (clones) in recursion automatically


Andries van Dam © 2024 11/12/24

# Stack Trace

- When an exception is thrown in a program, get a long list of methods and line numbers known as a stack trace

```
Exception in thread "main" java.lang.NullPointerException
   at DoodleJump.scroll(DoodleJump.java:94)
   at DoodleJump.updateGame(DoodleJump.java:44)
   ...
```

- A stack trace prints out all methods currently on execution stack
- If exception is thrown during execution of recursive method, prints all calls to recursive method

# Bootstrapping Data Structures

- This implementation of the stack data structure uses a wrapper of a contained `MyLinkedList`, but user has no knowledge of that
- Could also implement it with an `Array` or `ArrayList`
  - `Array` implementation could be more difficult--`Array`'s have fixed size, so would have to copy our `Array` into a larger one as we push more objects onto the `Stack`
  - User's code should not be affected even if the implementation of `Stack` changes (true for methods as well, if their semantics isn't changed) – loose coupling!
- We'll use the same technique to implement a `Queue`

# What are Queues?

- Similar to stacks, but elements are removed in different order
  - information retrieved in the same order it was stored
  - **FIFO**: **F**irst **I**n, **F**irst **O**ut (as opposed to stacks, which are **LIFO**: **L**ast **I**n, **F**irst **O**ut)
- Examples:
  - standing in line for merch at the Eras Tour
  - waitlist for TA hours after randomization


*Server at Seattle restaurant reminding herself what order customers get served in*

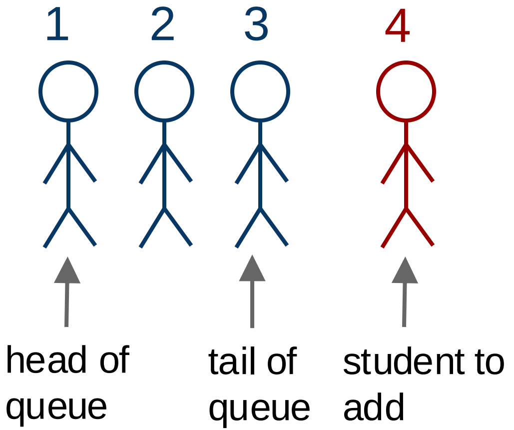
# Methods of a Queue

- Add element to end of `queue` — `public void enqueue(Type el)`
- Remove element from beginning of `queue` — `public Type dequeue()`
- Returns whether `queue` has any elements — `public boolean isEmpty()`
- Returns number of elements in `queue` — `public int size()`

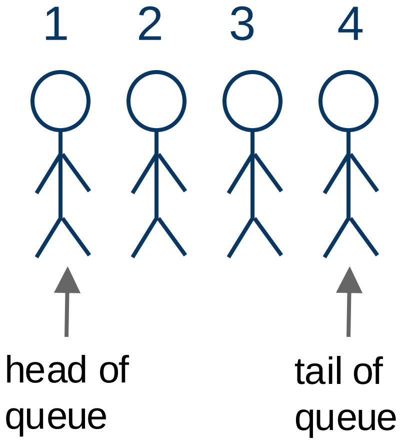
# Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Enqueuing



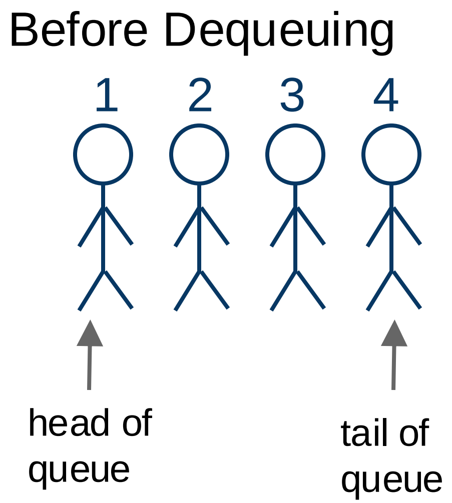

After Enqueuing

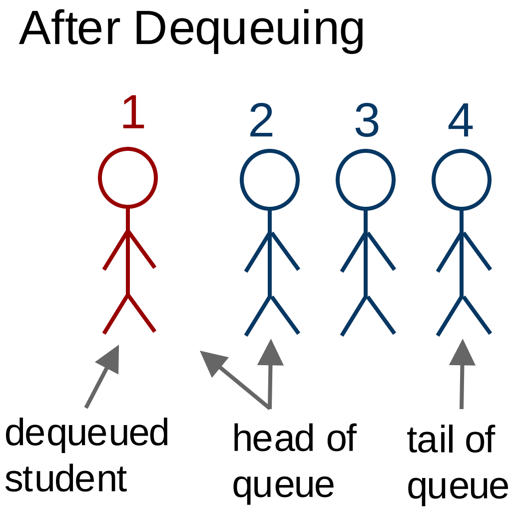



Andries van Dam © 2024 11/12/24

# Enqueuing and Dequeuing

- Enqueuing: adds a node to the back
- Dequeuing: removes a node from the front



# Our Queue

- Again use a wrapper for a contained `MyLinkedList.` As with `Stack`, we'll hide most of MLL's functionality and provide special methods that delegate the actual work to the MLL

```
public class Queue<Type> {

        private MyLinkedList<Type> list;

        public Queue() {
                this.list = new MyLinkedList<>();
        }
        // Other methods elided
}
```

- Contain a `MyLinkedList` within `Queue` class
    - `enqueue` will add to the end of `MyLinkedList`
    - `dequeue` will remove the first element in `MyLinkedList`

# enqueue

- Just call list's addLast method – delegation

```
public void enqueue(Type newNode) {
    this.list.addLast(newNode);
}
```

- This will add newNode to end of list

# dequeue

- We want first node in `list`
- Use `list`'s `removeFirst` method – delegation

```
public Type dequeue() {
        return this.list.removeFirst();
}
```

- What if `list` is empty? There will be nothing to dequeue!
- Our `MyLinkedList` class's `removeFirst()` method returns `null` in this case, so `dequeue` does as well

# `isEmpty()` and `size()`

- As with `Stack`s, very simple methods; just delegate to our wrapped `MyLinkedList`

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

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

# TopHat Question

In order from head to tail, a queue contains the following: `katara`, `sokka`, `aang`, `momo`. We remove each avatar from the queue by calling `dequeue()` and then immediately `push()` each dequeued avatar onto a stack.

At the end of the process, what is the order of the stack from top to bottom?

A. `katara, sokka, aang, momo`
B. `katara, momo, sokka, aang`
C. `momo, aang, sokka, katara`
D. It's random every time.

# Outline

- Stacks and Queues
- Trees

# Trees

# Searching in a Linked List (1/2)

- Searching for element in `LinkedList` involves pointer chasing and checking consecutive `Node`s to find it (or not)
  - it is sequential access
  - O(N) – can stop sooner for element not found if list is sorted
- Getting $N^{th}$ element in an `Array` or `ArrayList` by index is random access (which means O(1)), but (content-based) searching for particular element, even with index, remains sequential O(N)
- Even though `LinkedList`s support indexing (dictated by Java's `List` interface), getting the `i`th element is also done (under the hood) by pointer chasing and hence is O(N)

# Searching in a Linked List (2/2)

- For N elements, search time is O(N)
  - unsorted: sequentially check every node in list until element (“search key”) being searched for is found, or end of list is reached
    - if in list, for a uniform distribution of keys, average search time for a random element is N/2
    - if not in list, it is N
  - sorted: average* search time is N/2 if found, N/2 if not found (the win!)
  - we ignore issue of duplicates
- No efficient way to access $N^{th}$ node in list (via index)
- Insert and remove similarly have average search time of N/2 to find the right place

**Actually more complicated than this – depends on distribution of keys*

# Searching, Inserting, Removing

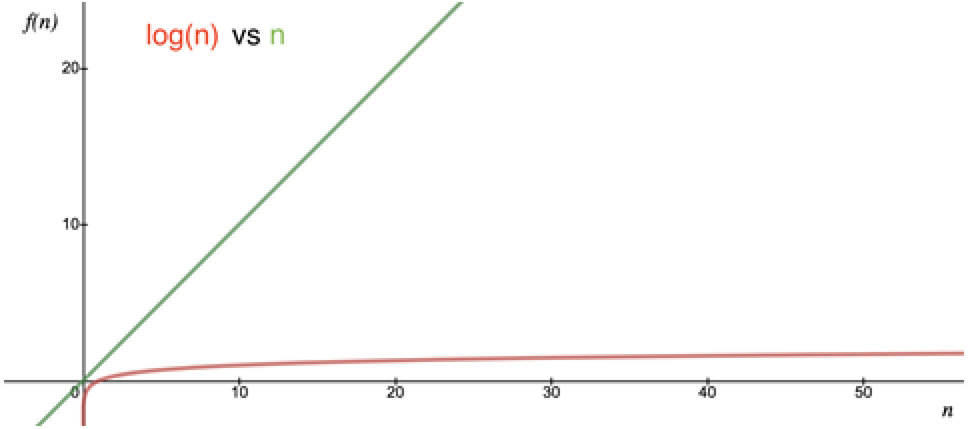
| | Search if unsorted | Search if sorted | Insert/remove after search |
|---|---|---|---|
| Linked list | O(N) | O(N) | O(1) |
| Array | O(N) | O(log N) [coming next] | O(N) |

# Binary Search (1/4)

- Searching sorted linked list is sequential access
- We can do better with a sorted array that allows random access at any index to improve sequential search
- Remember merge sort with search $O(\log_2 N)$ where we did "bisection" on the array at each pass
- If we had a sorted array, we could do the same thing
  - start in the middle
  - keep bisecting array, deciding which portion of the sub-array the search key lies in, until we find that key or can't subdivide further (not in array)
  - For N elements, search time is $O(\log_2 N)$ (since we reduce number of elements to search by half each time), very efficient!

# Binary Search (2/4)

- $\log_2 N$ grows much more slowly than N, especially for large N




**relatively small n in this graph, but imagine how large the difference is as n increases*

| N | (int) log(N) |
|---|---|
| 1 | 0 |
| 10 | 3 |
| 100 | 7 |
| 1,000 | 10 |
| 10,000 | 13 |
| 1,000,000 | 17 |
| 10,000,000 | 20 |
| 100,000,000 | 23 |
| 1,000,000,000 | 27 |

# Binary Search (3/4)

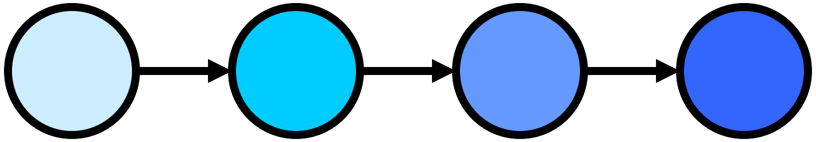
- A sorted `array` can be searched quickly using bisection because arrays are indexed
- `ArrayLists` (implemented in Java using arrays) are indexed too, so a sorted `ArrayList` shares this advantage! But inserting and removing from `ArrayLists` is slow (except for insertion and removal at either end)!
  - Inserting into or deleting from an arbitrary index in `ArrayList` causes all successor elements shift over. Thus insertion and deletion have same worst-case run time O(N)
- Advantage of `linkedLists` is insert/remove by manipulating pointer chain is faster [O(1)] than shifting elements [O(N)], but search can’t be done with bisection 😔, a real downside if search is done frequently

# Binary Search (4/4)

- Is there a data structure that provides both search speed of sorted arrays and `ArrayList`s and insertion/deletion efficiency of linked lists?
- Yes, indeed! `Tree`s! They provide much faster searching than linked lists and much faster insertions than arrays!

# Trees vs Linked Lists (1/2)

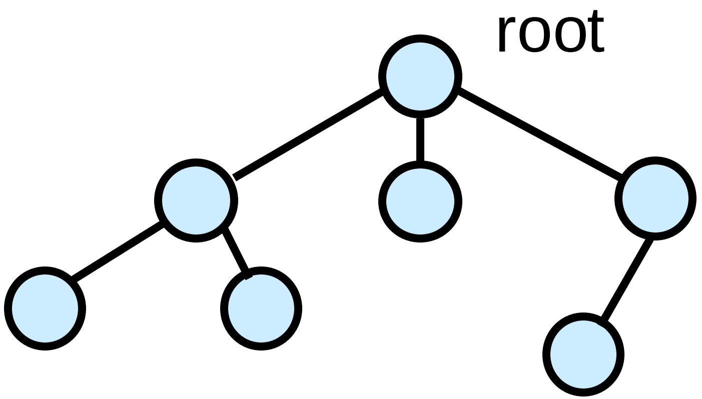
- Singly linked list – collection of nodes where each node references only one neighbor, the node's successor:



# Trees vs Linked Lists (2/2)

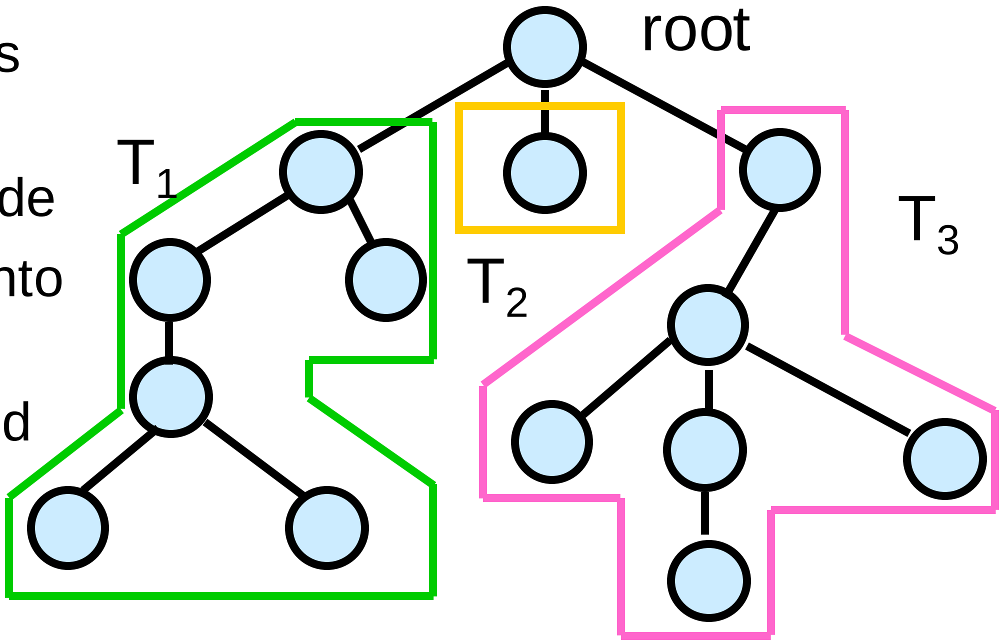
- Tree – also collection of nodes, but each node may reference multiple successors/children

- Trees can be used to model a hierarchical organization of data



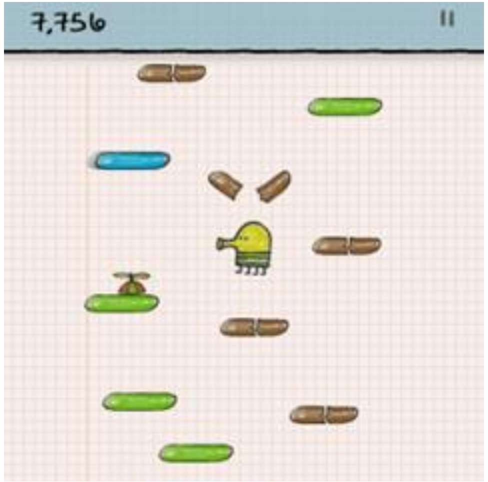
# Technical Definition of a Tree

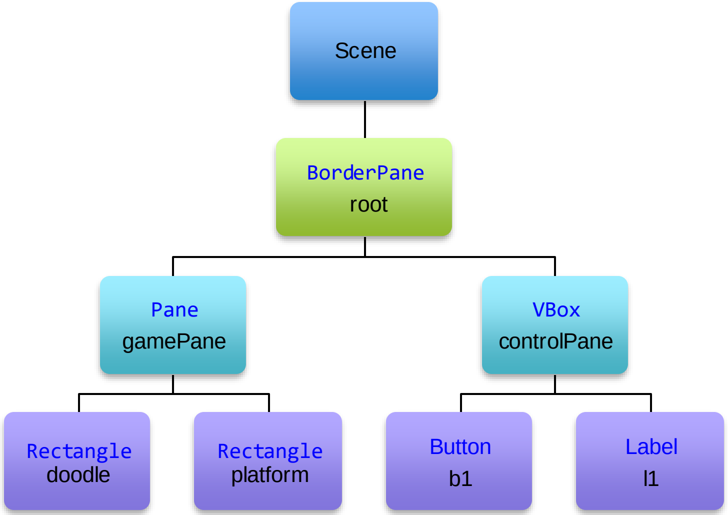
- Finite set, T, of one or more nodes such that:
  - T has one designated root node
  - remaining nodes partitioned into disjoint sets: $T_1$, $T_2$, … $T_n$
  - each $T_i$ is also a self-contained tree, called subtree of T
- Look at the image on the right- where have we seen seen such hierarchies like this before?



# Graphical Containment Hierarchies as Trees

- Levels of containment of **GUI** components



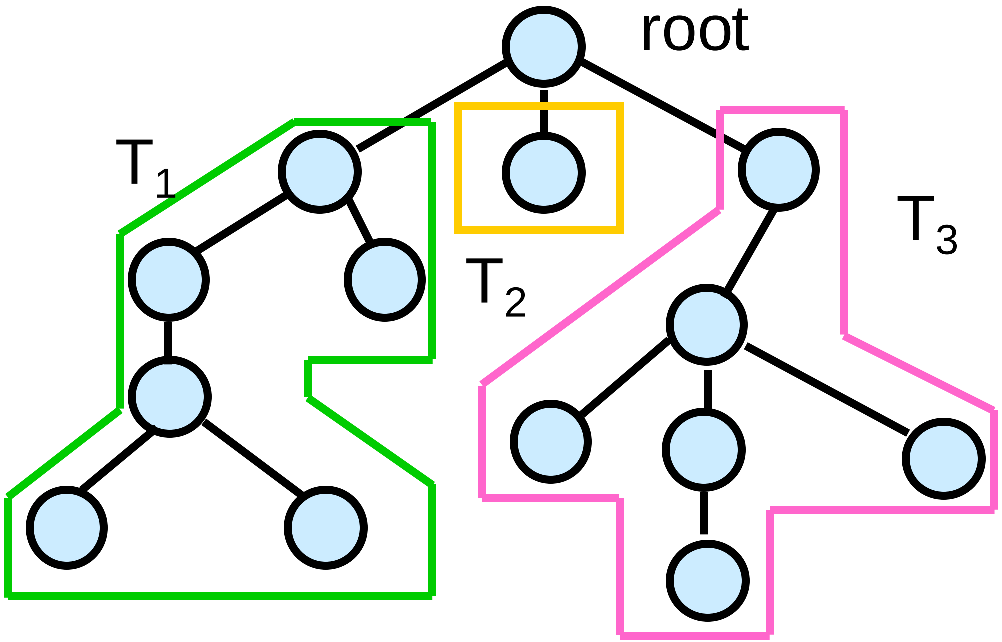




- Higher levels contain more components
- Lower levels contained by all above them
  - Panes contained by **root** pane, which is contained by Scene

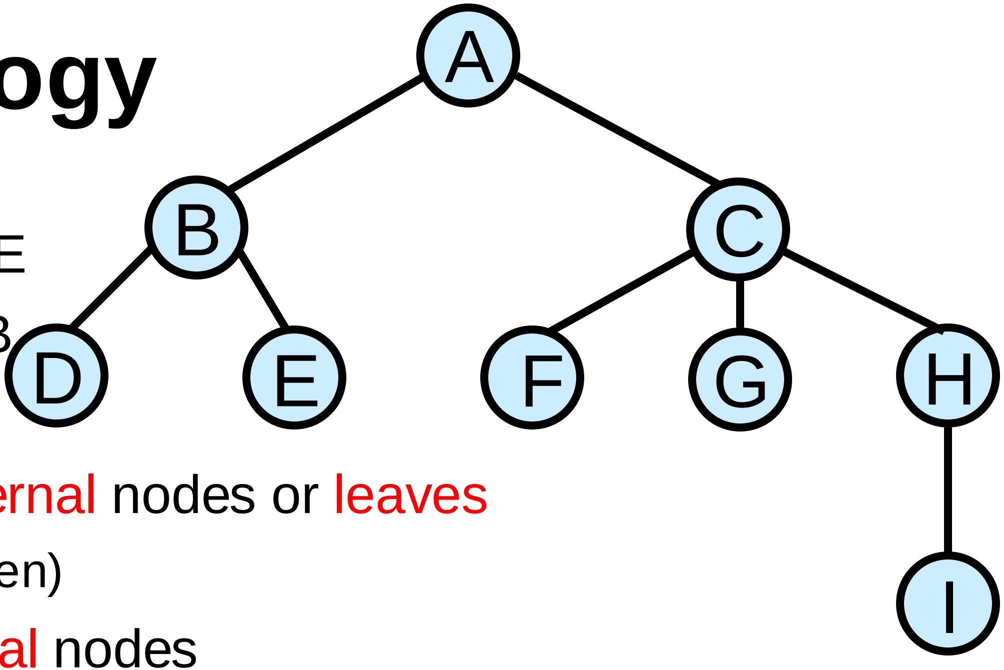
# Tree Structure

- Note that the tree structure has meaning
  - any subtree of T, $T_i$, is also a tree with specific values

- Can be useful to only examine specific subtrees of T



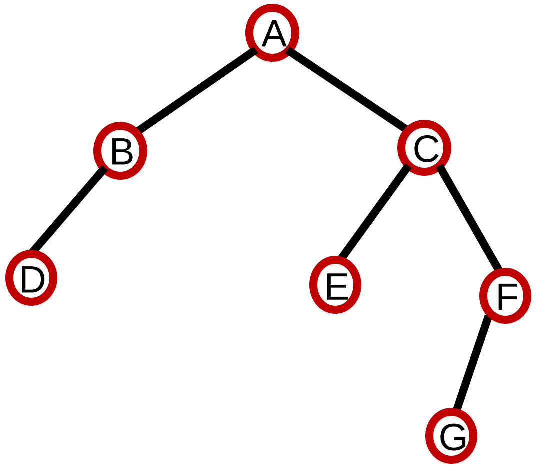
# Tree Terminology




- A is the root node
- B is the parent of D and E
- D and E are children of B
- (C ─── F) is an edge
- D, E, F, G, and I are external nodes or leaves
  - (i.e., nodes with no children)
- A, B, C, and H are internal nodes
- depth (level) of E is 2 (number of edges to root)
- height of the tree is 3 (max number of edges in path from root)
- degree of node B is 2 (number of children)

# Binary Trees

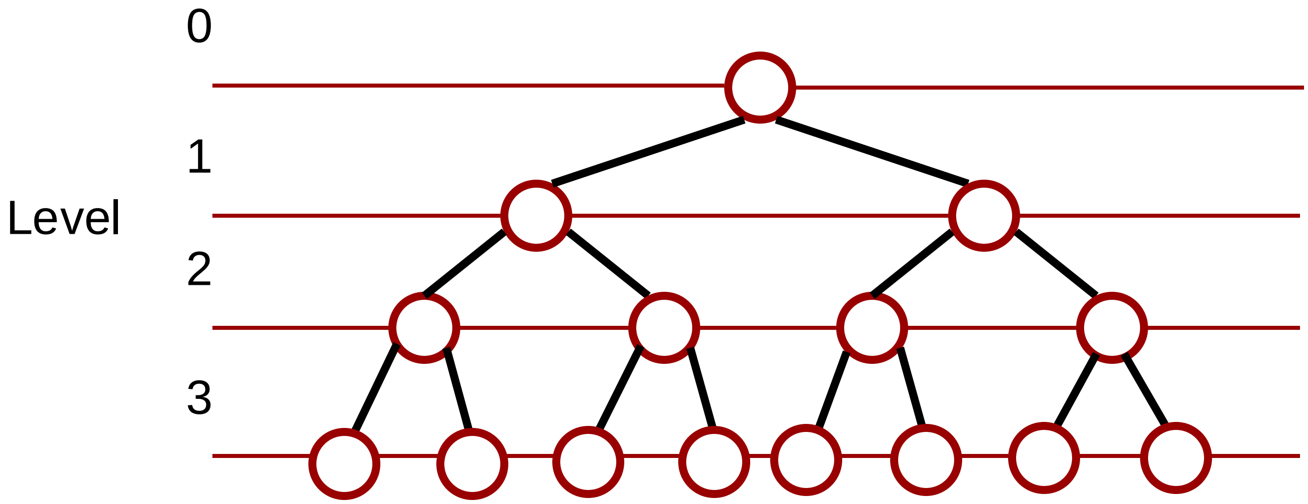
- Each internal node has a maximum of 2 successors, called children
  - i.e., each internal node has degree 2 at most
- Recursive definition of binary tree: A binary tree is either an:
  - external node (leaf), or
  - internal node (root) with one or two binary trees as children (left subtree, right subtree)
  - empty tree (represented by a null pointer)




- *Note:* These nodes are similar to the linked list nodes, with one data and two child pointers – we show the data element inside the circle

# Properties of Binary Trees (1/2)

- A binary tree is full when each node has exactly zero or two children
- Binary tree is perfect when, for every level i, there are $2^i$ nodes (i.e., each level contains a complete set of nodes)
  - thus, adding anything to the tree would increase its height

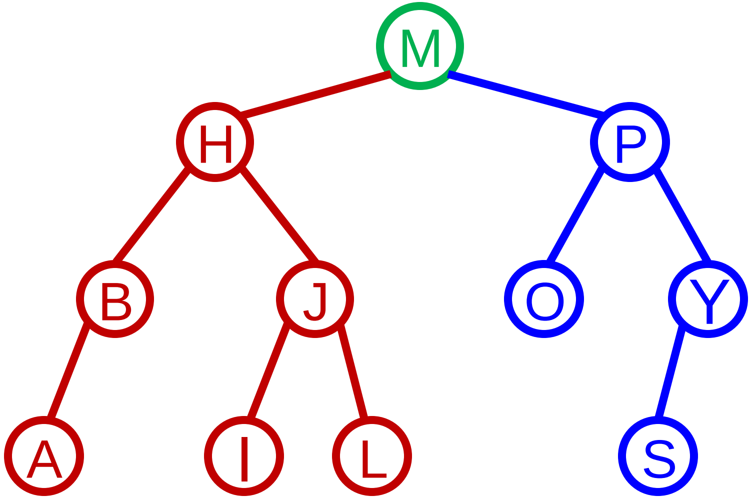


# Properties of Binary Trees (2/2)

- *In a full Binary Tree:* (# leaf nodes) = (# internal nodes) + 1
- *In a perfect Binary Tree:* (# nodes at level i) = $2^i$
- *In a perfect Binary Tree:* (# leaf nodes) <= $2^{(height)}$
- *In a perfect Binary Tree:* (height) >= $\log_2$(# nodes) - 1

# Binary Search Tree a.k.a BST (1/2)

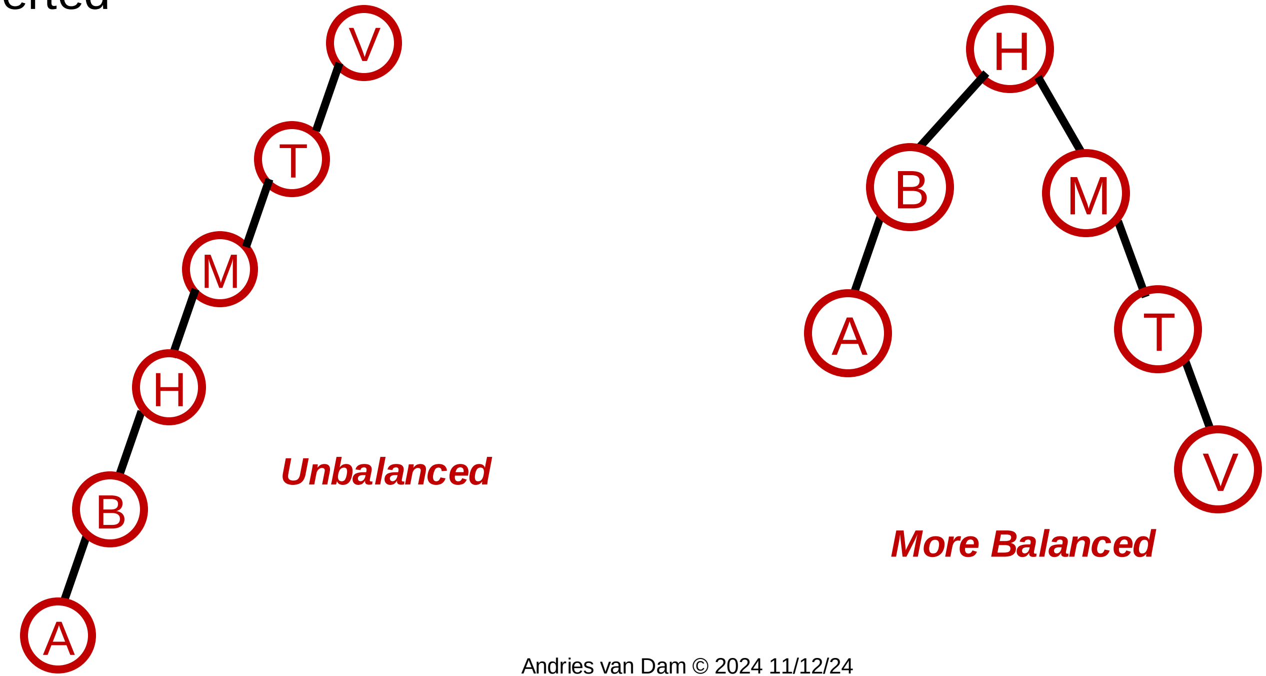
- Binary search tree stores keys in its nodes such that, for every node, keys in left subtree are smaller, and keys in right subtree are larger




*Note: the keys here are sorted alphabetically!*

# BST (2/2)

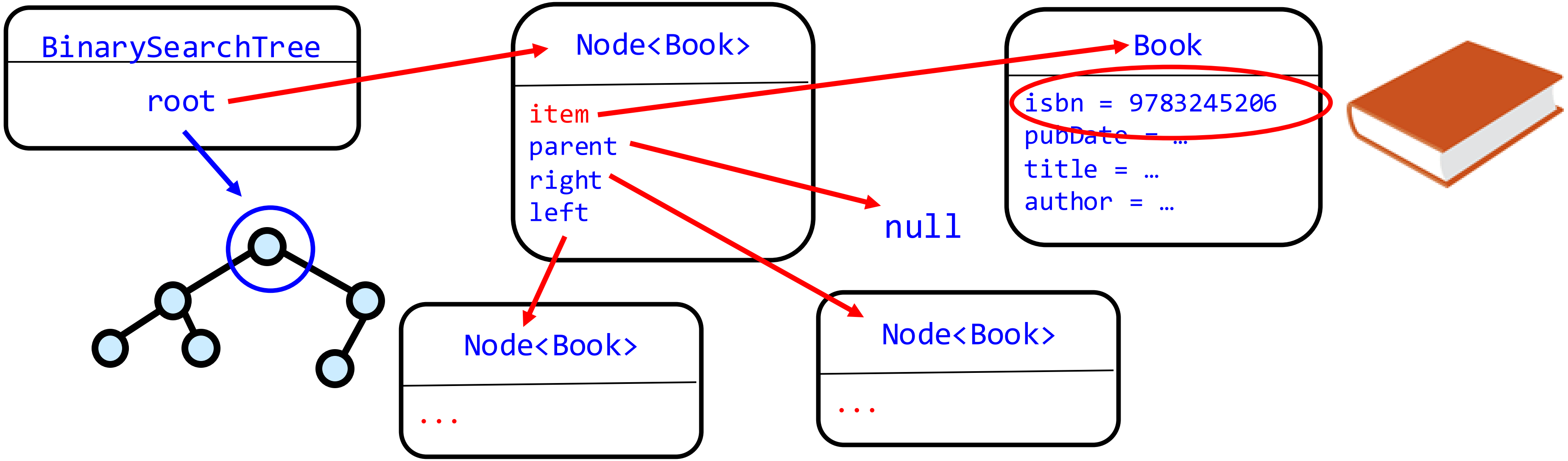
- Below is also BST but much less balanced. Gee, it looks like a linked list!
- The shape of the trees is determined by the order in which elements are inserted



# BST Class (1/4)

- What do BSTs know how to do?
  - much the same as sorted linked lists: *insert*, *remove*, *size, empty*
  - BSTs also have their own search method – a bit more complicated than simply iterating through its nodes
- What would an implementation of a BST class look like…
  - in addition to data, left, and right child pointers, we'll add a parent "back" pointer for ease of implementation (for the `remove` method – analogous to the `previous` pointer in doubly-linked lists!)
  - you'll learn more about implementing data structures in CS200!

# Nodes, data items, and keys




- item is a composite that can contain many properties,
- one of which is a key that Nodes are sorted by (here, ISBN #)

# Java's `Comparable<Type>` interface (1/3)

- Previously we used `==` to check if two things are equal
  - this only works correctly for primitive data types (e.g., `int`), or when we are comparing two variables referencing the exact same object
  - to compare `String`s, need a different way to compare things

- We can implement the `Comparable<Type>` generic interface provided by Java

- It specifies the `compareTo` method, which returns an `int`

- Why don't we just use ==, even when using something like ISBN, which is an `int`?
  - can treat ISBNs as `int`s and compare them directly, but more generally we implement the `Comparable<Type>` interface, which could easily accommodate comparing `String`s, such as author or title, or any other property

# Java's Comparable<Type> interface (2/3)

- The Comparable<Type> interface is specialized (think of it as parameterized) using generics

```
public interface Comparable<Type> {
    int compareTo(Type toCompare);
}
```

- Call compareTo on a variable of same type as specified in implementator of interface (Book, in our case)
  - currentBook.compareTo(bookToFind);

# Java's `Comparable<Type>` interface (3/3)

- `compareTo` method must return an `int`
  - **negative** if element on which `compareTo` is called is *less* than element passed in as the parameter of the search
  - **0** if element is *equal* to element passed in
  - **positive** if element is *greater* than element passed in
  - sign of `int` returned is all-important, magnitude is not and is implementation dependent
- `compareTo` not only used for numerical comparisons–it could be used for alphabetical or geometric comparisons as well–depends on how you implement `compareTo`

# “Comparable” Book Class

- Recall format for `compareTo`:
  - `elementA.compareTo(elementB)`
- Book class now implements `Comparable<Book>`
  - this means we can compare books, using `bookA.compareTo(bookB)`
- `compareTo` is defined according to these specifications
  - returns number that is `<0`, `0` or `>0`, depending on the ISBN numbers
  - < 0 if stored `this.isbn` < `toCompare`
  - == 0 if `this.isbn == toCompare`
  - > 0 if `this.isbn > toCompare`

```
public class Book implements Comparable<Book> {
    // variable declarations, e.g., isbn, elided
    public Book(String author, String title,
                       int isbn){
        //variable initializations elided
    }

    public int getISBN(){
        return this.isbn;
    }

    //other methods elided

    //compare isbn of book passed in to stored one

  @Override
    public int compareTo(Book toCompare){
        return (this.isbn - toCompare.getISBN());
    }
}
```

# BST Class (2/4)

- In our example, use Book as Type
- In generics, extends is used both for ‘extends’ and ‘implements’

- Using keyword extends in this way ensures that Type implements Comparable<Type>
  - note nested <>; shows it modifies Type and not the class
  - for generics, extends is used instead of implements for interfaces
- All elements stored in MyLinkedList must now have compareTo method for Type; thus restricts generic

```
public class BinarySearchTree<Type extends
                                Comparable<Type>> {

    private Node<Type> root;

    public BinarySearchTree(Type item) {
        //Root of the tree
        this.root = new Node(item, null);
    }

    // other methods shown next slide
}
```

# BST Class (3/4)

```
public class BinarySearchTree<Type extends
  Comparable<Type>> {

    private Node<Type> root;

    public BinarySearchTree(Type item) {
        //Root of the tree
        this.root = new Node(item, null);
    }

    public void insert(Type newData) {
          // . . .
     }
```

```
   //class continued
   public void remove(Type dataToRemove) {
      // . . .
   }

   public Node<Type> search(Type dataToFind) {
      // . . .
   }

   public int size() {
      // . . .
   }

} // end of class
```

# BST Class (4/4)

- Our implementations of `LinkedLists`, `Stacks`, and `Queues` are “smart” data structures that chain “dumb” nodes together
  - the lists did all the work by maintaining previous and current pointers and did the operations to search for, insert, and remove information – thus, nodes were essentially data containers
- Now we will use a “dumb” tree with “smart” nodes that will delegate using recursion
  - tree will delegate action (such as searching, inserting, etc.) to its root, which will then delegate to its appropriate child, and so on
  - creates specialized `Node` class that stores its item, parent, and children, and can perform operations such as `insert` and `remove`

# BST: Node Class (1/3)

- “Smart” Node includes the following methods:

```
// pass in entire data item, containing key, so compareTo() will work
public Node<Type> search(Type itemToFind);
public Node<Type> insert(Type newItem);

/* remove deletes Node pointing to dataToRemove, which contains key;
removing Node also will remove the matched data element instance unless
there’s another reference to it */
public Node<Type> remove(Type itemToRemove);
```

- Plus setters and getters of instance variables, defined in the next slides …

Andries van Dam © 2024 11/12/24

# BST: Node Class (2/3)

- `Node`s have a maximum of two non-`null` children that hold data implementing `Comparable<Type>`
  - four instance variables: `item`, `parent`, `left`, and `right`, with each having a `get` and `set` method.
  - `item` represents the data that `Node` stores. It also contains the key attribute that `Node`s are sorted by – we'll make a `Tree` that stores `Book`s
  - `parent` represents the direct parent (another `Node`) of `Node`–only used in `remove` method
  - `left` represents `Node`'s left child and contains a subtree, all of whose data is less than `Node`'s data
  - `right` represents `Node`'s right child and contains a subtree, all of whose data is greater than `Node`'s data
  - arbitrarily select which child should contain data equal to `Node`'s data


Andries van Dam © 2024 11/12/24

# BST: Node Class (3/3)

```
public class Node<Type implements Comparable<Type>> {
    private Type item;
    private Type parent;
    private Node<Type> left;
    private Node<Type> right;
    public Node(Type item, Node<Type> parent){ //construct a leaf node as default
        this.item = item;
        this.parent = parent;
        //child ptrs null for leaf nodes; set for internal nodes when child is created
        this.left = null;
        this.right = null;
    }
   // will define other methods in next slides…
}
```

# Smart Node Approach

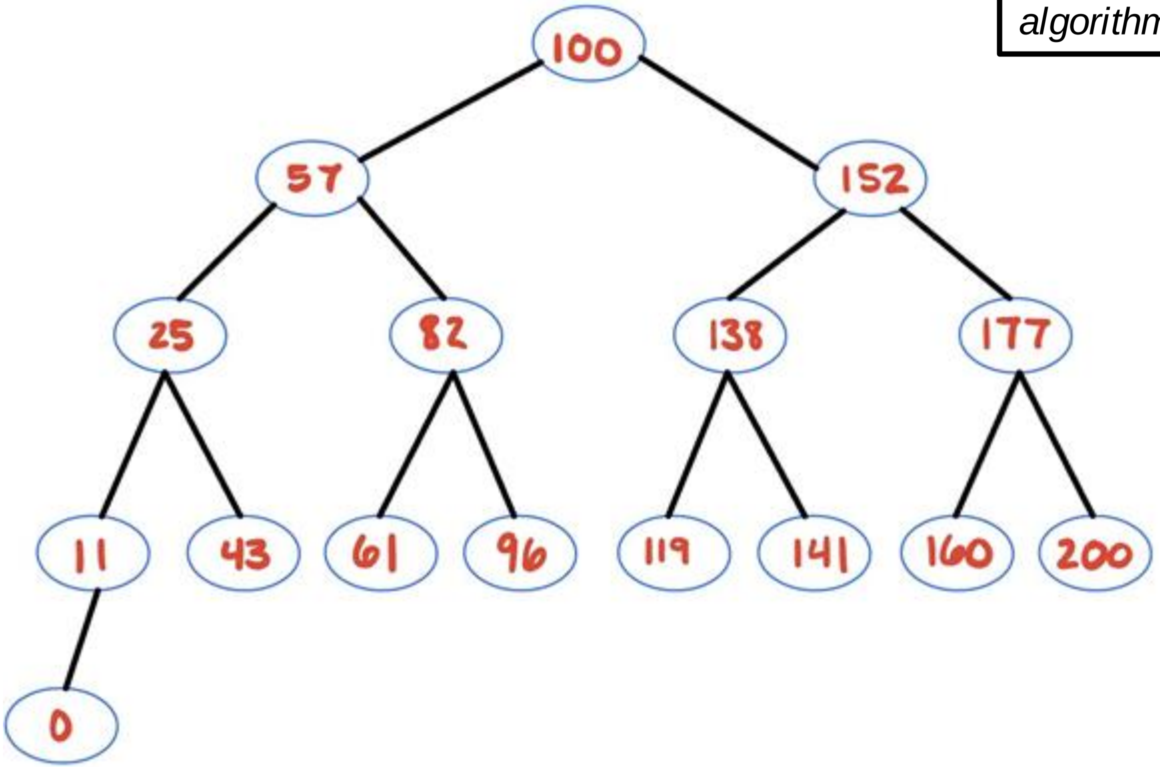
- `BinarySearchTree` is “dumb,” so it delegates to root, which in turn will delegate recursively to its left or right child, as appropriate

```
// search method for entire BinarySearchTree:
public Node<Type> search(itemToFind) {
    return this.root.search(itemToFind);
}
```

- Smart node approach makes our code clean, simple and elegant
  - non-recursive method is much messier, involving explicit bookkeeping of which node in the tree we are currently processing
    - we used the non-recursive method for sorted linked lists, but trees are more complicated, and recursion is easier – a tree is composed of subtrees!

# Let’s Search a BST

*For a step-by-step walkthrough of this algorithm,* *see slide 82*



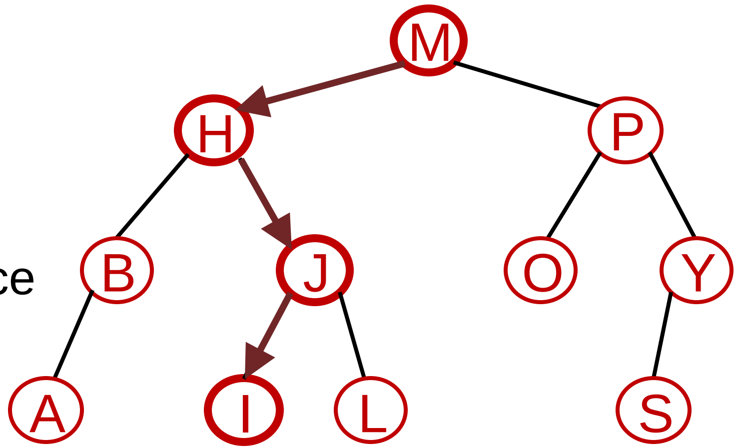
# TopHat Question

What's the runtime of (recursive) search in a BST and why?

A. O(n) – because you only iterate once
B. O(2n) – because you go visit both the left and right subtrees
C. O(n/2) – because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
D. $O(\log_2 n)$ - because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
E. $O(n^2)$ – because recursion makes your runtime quadratic

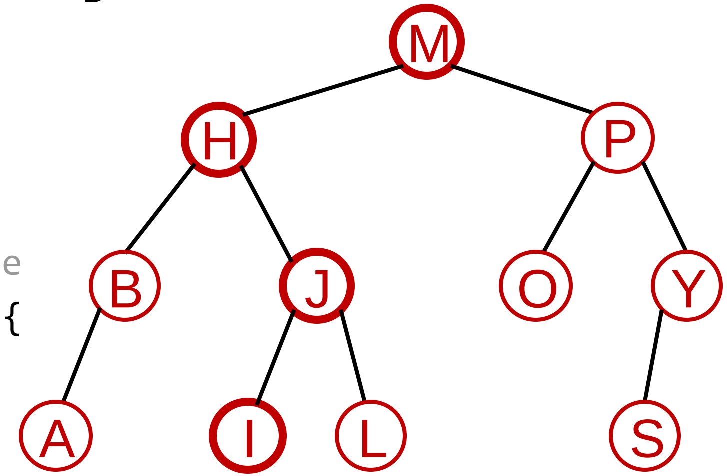
# Searching a BST Recursively Is $O(\log_2 N)$

- Search path: start with root **M** and choose path to **I** (for a reasonably balanced tree, **M** will be more or less "in the middle," and left and right subtrees will be roughly the same size)
  - structurally, the height of a reasonably balanced tree with *n* nodes is about $\log_2 n$
  - at most, we visit each level of the tree once
  - so, runtime performance of searching is $O(\log_2 N)$ as long as tree is reasonably balanced, which will be true if entry order is reasonably random
  - $O(\log_2 N)$ is much less than N, this is thus much more efficient!



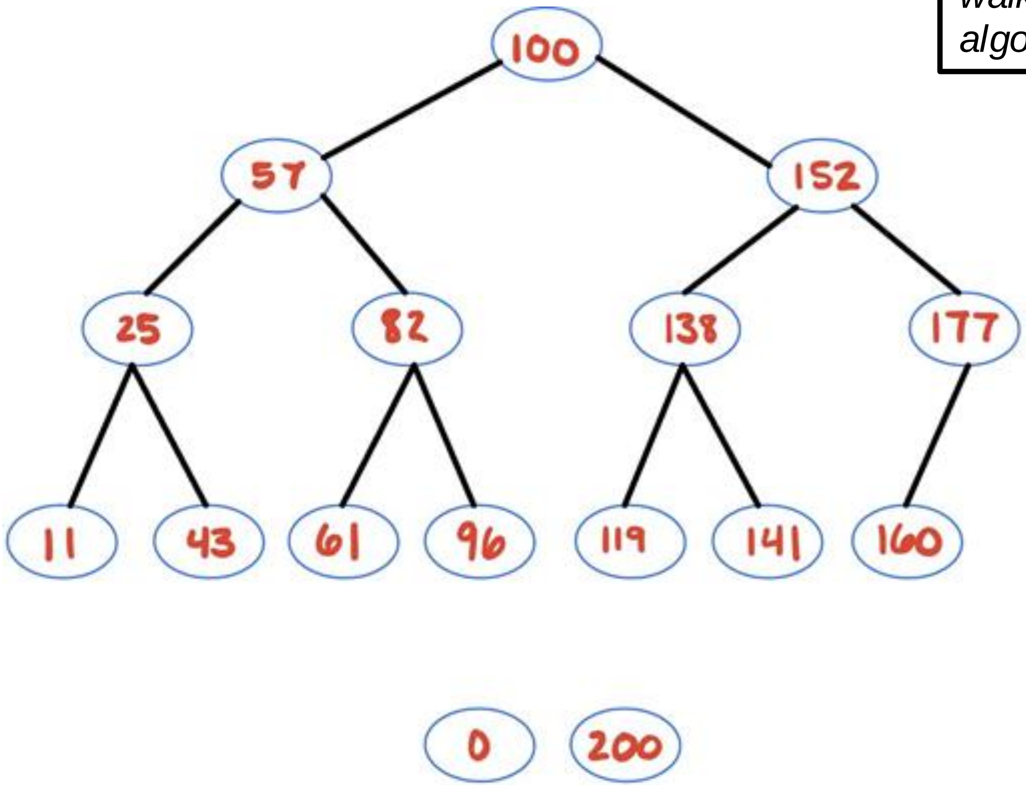
# Searching a BST Recursively

```
public Node<Type> search(Type itemToFind) {
    //if item is the thing we’re searching for
    if(this.item.compareTo(itemToFind) == 0) {
        return this.item;
    //if item > itemToFind, can only be in left tree
    } else if(this.item.compareTo(itemToFind) > 0) {
        if(this.left != null) {
            return this.left.search(itemToFind);
        }
    //if item < itemToFind, can only be in right tree
    } else if (this.right != null) {
            return this.right.search(itemToFind);
        }
    }
    //Only get here if itemToFind isn’t in tree, otherwise would’ve returned sooner
    return null;
}
```



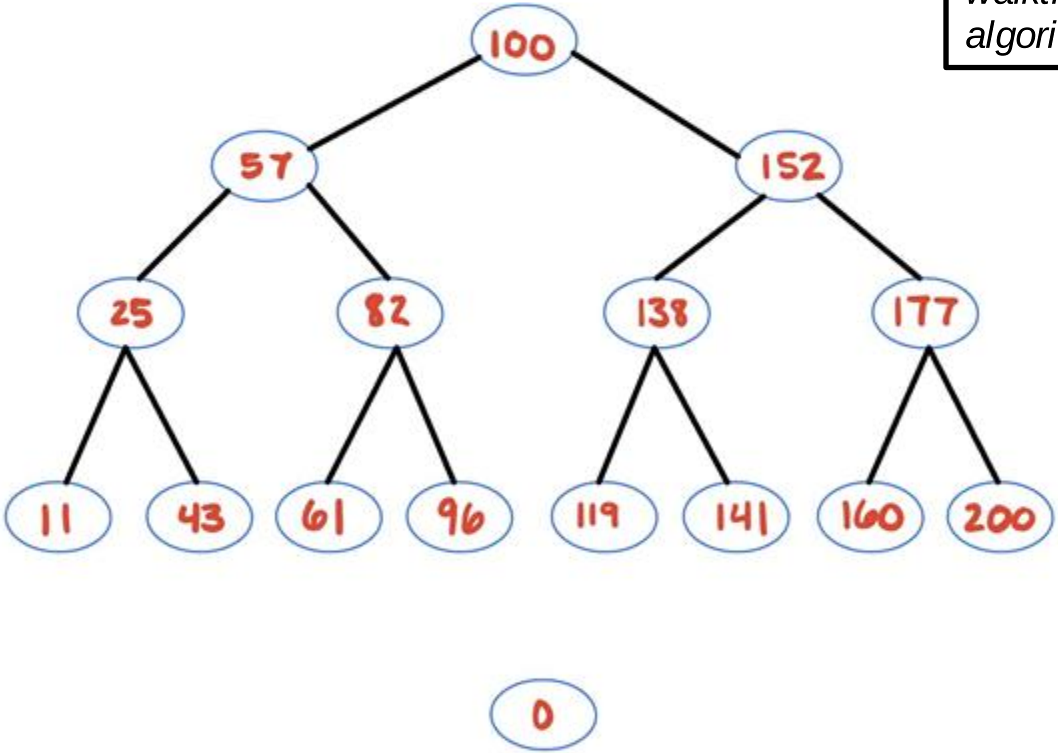
# Let’s Add to a BST (1/3)

*For a step-by-step walkthrough of this algorithm, see slide 90*



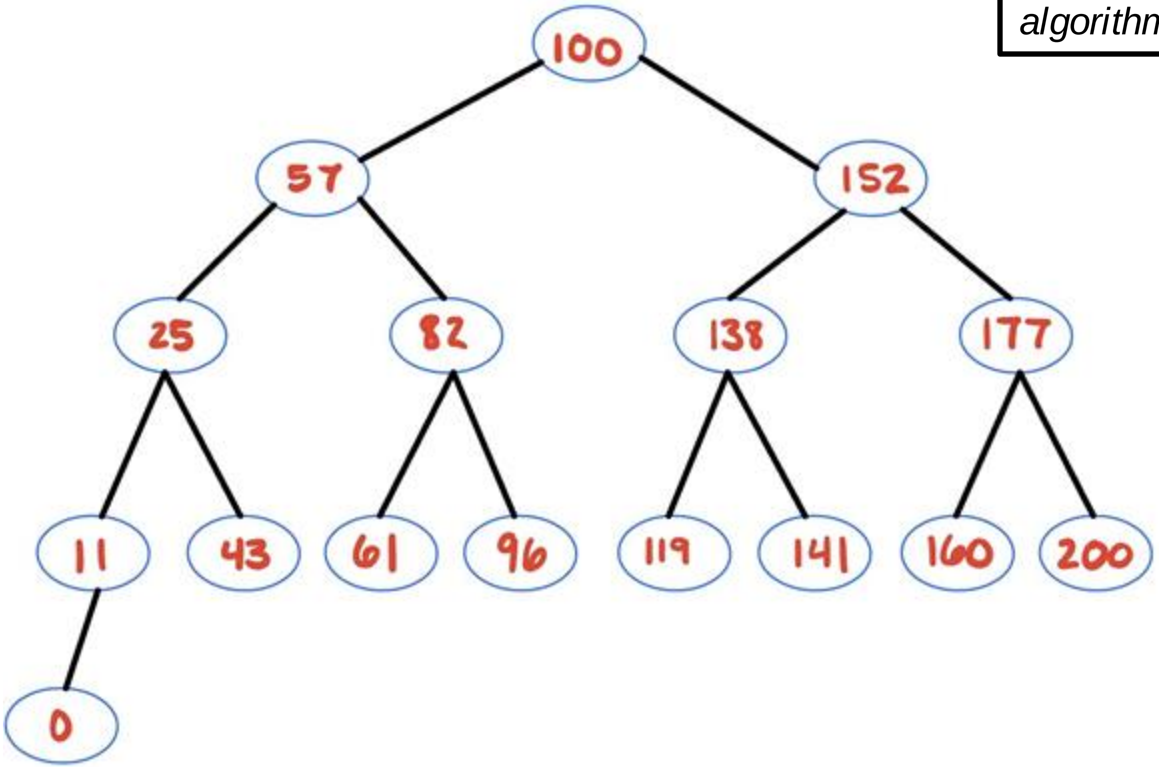
# Let’s Add to a BST (2/3)

*For a step-by-step walkthrough of this algorithm, see slide 90*



# Let’s Add to a BST (3/3)

*For a step-by-step walkthrough of this algorithm, see slide 90*



# Insertion into a BST

- Search BST starting at root until we find where the data to insert belongs
    - insert data when we reach a `Node` whose appropriate L or R child is `null`
- That `Node` makes a new `Node`, sets the new `Node`’s `data` to the data to insert, and sets child reference to this new `Node`
- Runtime is $O(\log_2 N)$, yay!
    - $O(\log_2 N)$ to search the nearly balanced tree to find the place to insert
    - constant time operations to make new `Node` and link it in

Andries van Dam © 2024 11/12/24

# Insertion Code in BST

- Again, we use a "Smart Node" approach and delegate

```
//Tree's insert delegates to root
public Node<Type> insert(Type newItem) {
  //if tree is empty, make first node. No traversal necessary!
  if(this.root == null) {
      this.root = new Node(newItem, null); //root's parent is null
      return this.root;
  } else {
      //delegate to Node's insert() method
      return this.root.insert(newItem);
  }
}
```

# Insertion Code in Node

```
public Node<Type> insert(Type newItem) { //insert method continued!
    if (this.item.compareTo(newItem) > 0) { //newItem should be in left subtree
       if(this.left == null) { //left child is null – we've found the place to insert!
           this.left = new Node(newItem, this);
           return this.left;
       } else { //keep traversing down tree
           return this.left.insert(newItem);
       }
    } else { //newItem should be in right subtree
       if(this.right == null) { //right child is null-we've found the place to insert!
           this.right = new Node(newItem, this);
           return this.right;
       } else { //keep traversing down tree
           return this.right.insert(newItem);
       }
    }
}
```

*Reference to the new `Node` is passed up the tree so it can be returned by the tree*

# Notes on Trees (1/2)

- Different insertion order of nodes results in different trees
  - if you insert a node referencing data value of 18 into empty tree, that node will become root
  - if you then insert a node referencing data value of 12, it will become left child of root
  - however, if you insert node referencing 12 into an empty tree, it will become root
  - then, if you insert one referencing 18, that node will become right child of root
  - even with same nodes, different insertion order makes different trees!
  - on average, for reasonably random (unsorted) arrival order, trees will look similar in depth so order doesn’t play a major role in runtime

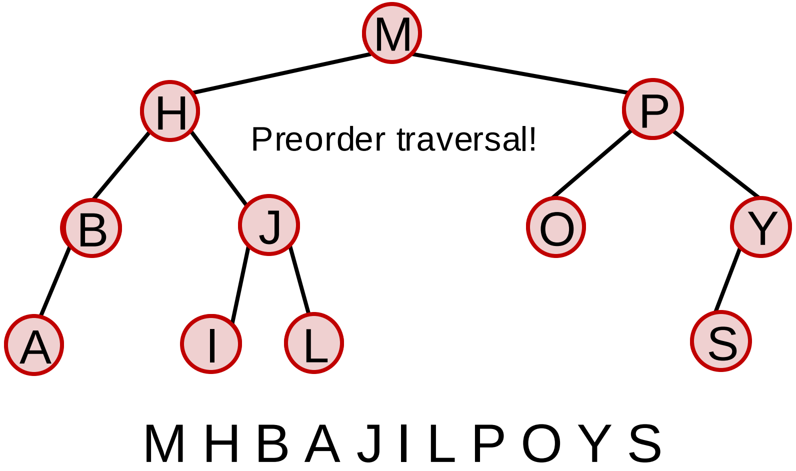
# Notes on Trees (2/2)

- When searching for a value, reaching another value that is greater than the one being searched for does not mean that the value being searched for is not present in tree (whereas it does in linked lists!)
  - it may well still be contained in left subtree of node of greater value that has just been encountered
  - thus, where you might have given up in linked lists, **you can't give up here until you reach a leaf** (but depth is roughly $\log_2 N$ for a nearly balanced tree, which is much smaller than N/2!)

# Preorder Traversal of BST

- Preorder traversal
  - “pre-order” because self is visited before (“pre-”) visiting children
  - again, use recursion!

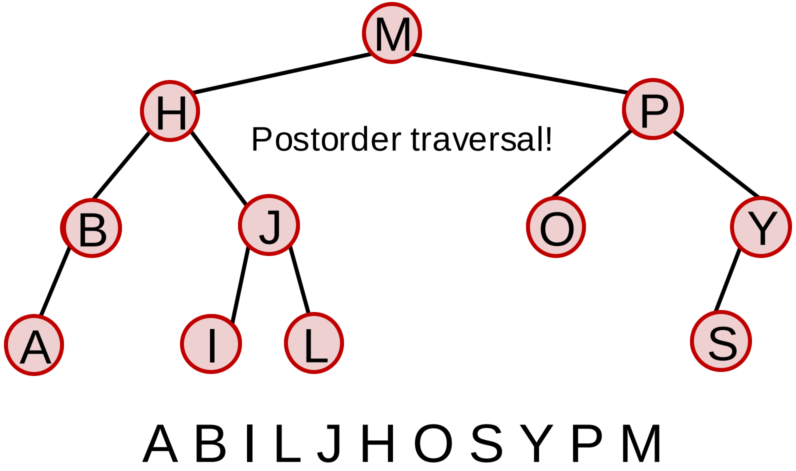
```
public void preOrder() {
    //Check for null children elided
     System.out.println(curr.item);
    this.left.preOrder();
    this.right.preOrder();
}
```



# Postorder Traversal of BST

- Postorder traversal
  - “post-order” because self is visited after (“post-”) visiting children
  - again, use recursion!

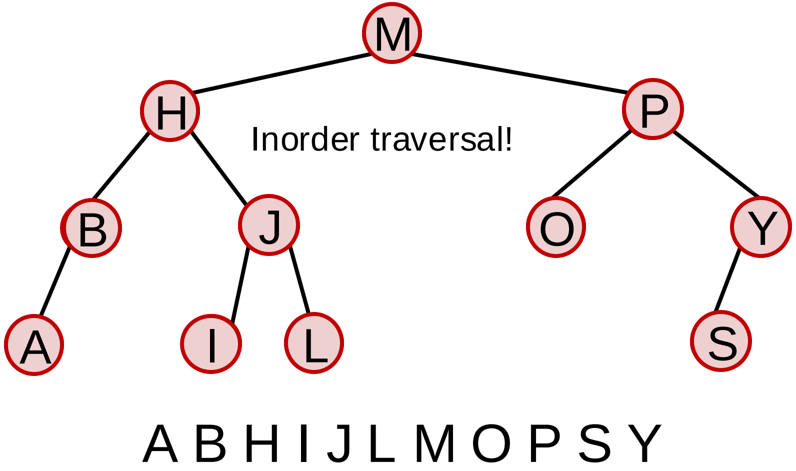
```
public void postOrder() {
    //Check for null children elided
    this.left.postOrder();
    this.right.postOrder();
    System.out.println(curr.item);
}
```



# Inorder Traversal of BST

- Inorder traversal
  - “in-order” because self is visited between (“in-”) visiting children
  - again, use recursion!

```
public void inOrder() {
    //Check for null children elided
    this.left.inOrder();
    System.out.println(curr.item);
    this.right.inOrder();
}
```




To learn more about the exciting world of trees, take CS200 (CSCI0200): Program Design with Data Structures and Algorithms!

# Tree Runtime

- Binary Search Tree has a search of $O(\log_2 n)$ runtime, can we make it faster?
- Could make a ternary tree! (each node has at least 3 children)
  - $O(\log_3 n)$ runtime
- Or a 10-way tree with $O(\log_{10} n)$ runtime
- Let’s try the runtime for a search with 1,000,000 nodes
  - $\log_{10} 1{,}000{,}000 = 6$
  - $\log_2 1{,}000{,}00 < 20$, so shallower but broader tree
- Analysis: the logs are not sufficiently different and the comparison (basically an n-way nested if-else-if) is far more time consuming, hence not worth it
- Furthermore, binary tree makes it easy to produce an ordered list

# Prefix, Infix, Postfix Notation for Arithmetic Expressions (1/2)

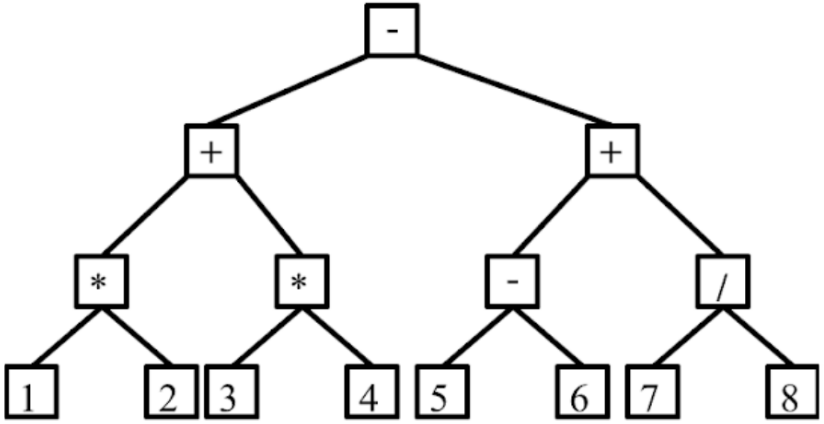
- When you type an equation into a spreadsheet, you use Infix; when you type an equation into many Hewlett-Packard calculators, you use Postfix, also known as “Reverse Polish Notation,” or “RPN,” after its inventor Polish Logician Jan Lukasiewicz (1924)
- Easier to evaluate Postfix because it has no parentheses and evaluates in a single left-to-right pass
- Use Dijkstra’s 2-stack shunting yard algorithm to convert from user-entered Infix to easy-to-handle Postfix – compile or interpret it on the fly


Andries van Dam © 2024 11/12/24

# Prefix, Infix, Postfix Notation for Arithmetic Expressions (2/2)

- Infix, Prefix, and Postfix refer to where the operator goes relative to its operands
  - Infix: (fully parenthesized)
    - ((1 * 2) + (3 * 4)) - ((5 - 6) + (7 / 8))
  - Prefix:
    - - + * 1 2 * 3 4 + - 5 6 / 7 8
  - Postfix:
    - 1 2 * 3 4 * + 5 6 - 7 8 / + -

- Graphical representation for equation:



# Announcements

- Tetris deadlines
  - late handin: Wednesday 11/13
  - keep an eye out for Tetris code debriefs
- COME TO LECTURE THURSDAY TO LEARN ABOUT FPs AND HAVE A FUN TIME!
- HTA Hours Friday 3-4pm (as always!) in CIT 209
  - come talk to us about which FP to do!
- Reminder to resubmit all non-MF projects by end of semester
- Reminder that you cannot use late days on FPs, and can only use max 2 on Tetris no matter how many you have left
- DoodleJump form on Ed
- Final mentor meetings – keep an eye on your emails - talk about registering for classes (coming up soon! come talk to us about 200!)

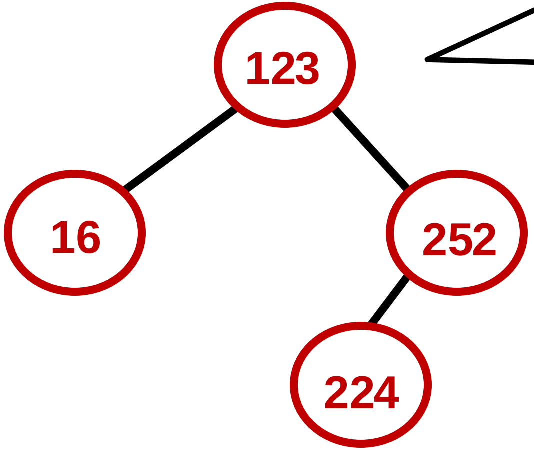
# Appendix

- Searching Simulation
- Insertion Demonstration
- Remove

# Searching Simulation (animated)

- What if we want to know if **224** is in Tree?
- Tree says:

"Hey Root! Ya got **224**?"

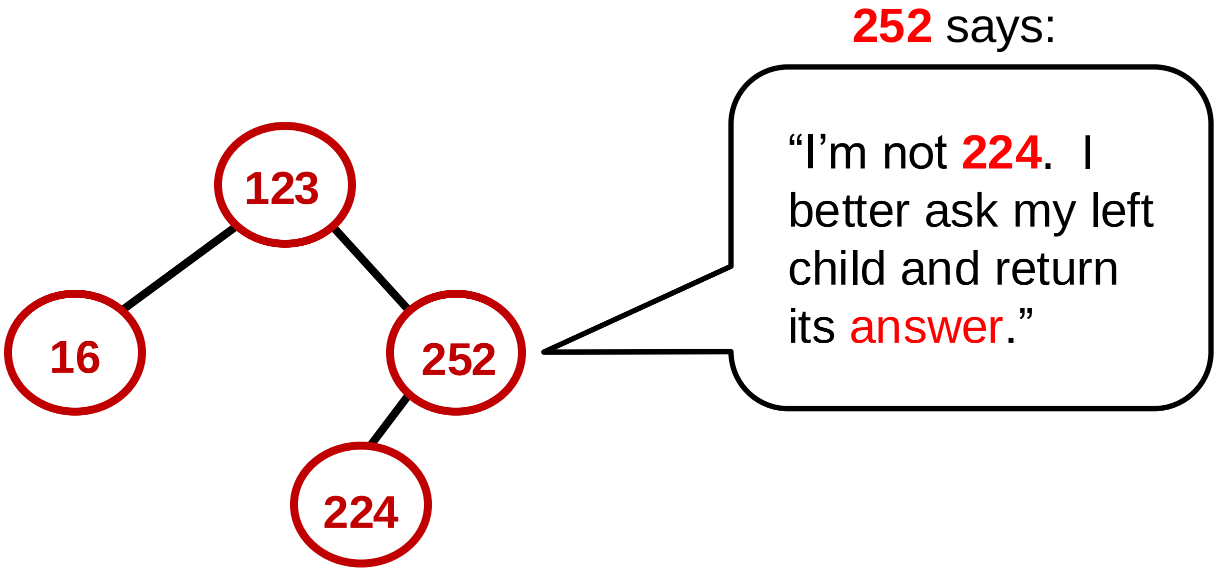



**123** says:

"Let's see. I'm not **224**. But if **224** is in tree, since it's larger, it would be to my right. I'll ask my right child and return its answer."

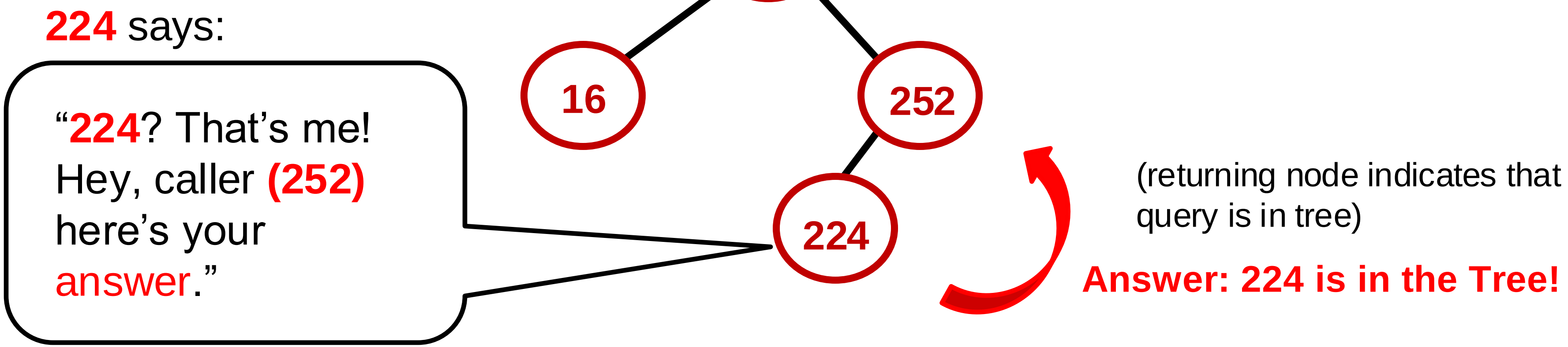
# Searching Simulation (animated)

- What if we want to know if **224** is in Tree?



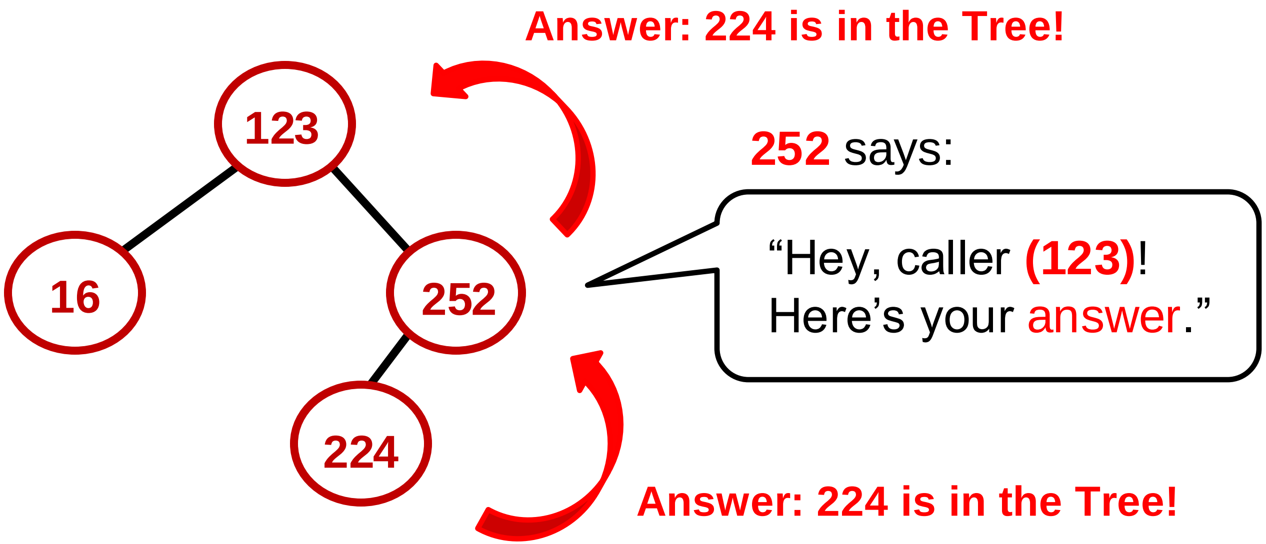
# Searching Simulation (animated)

- What if we want to know if **224** is in Tree?



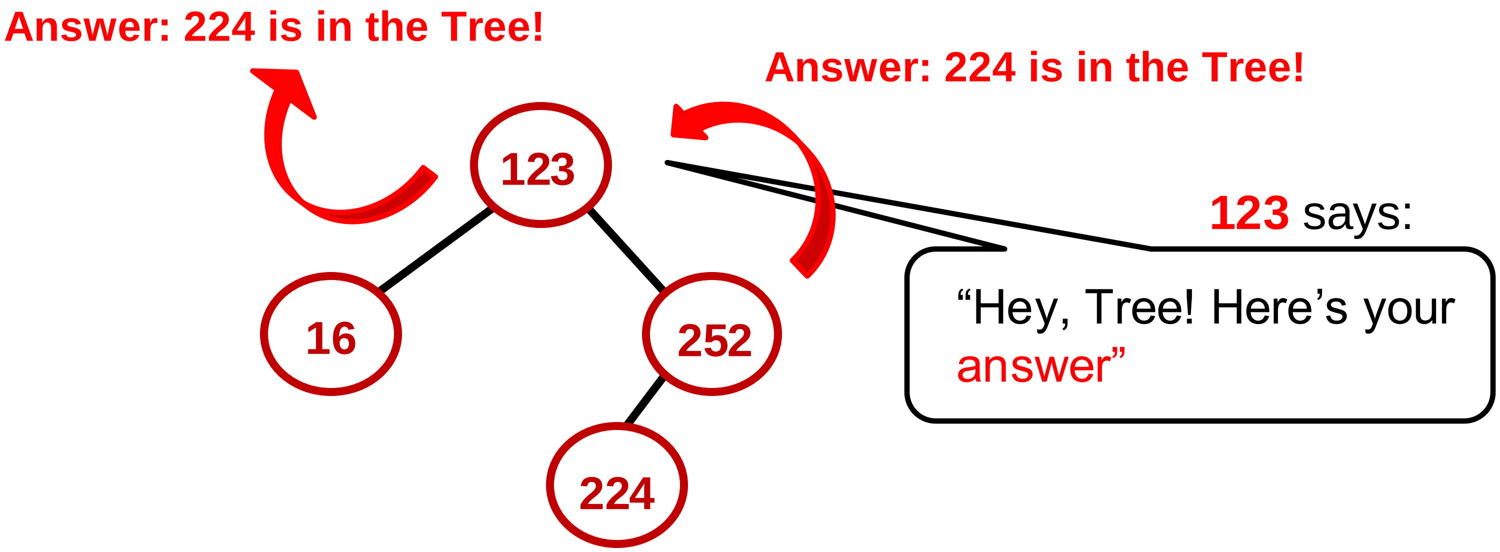
# Searching Simulation (animated)

- What if we want to know if **224** is in Tree?

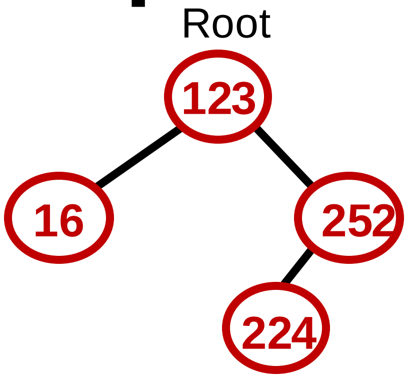


# Searching Simulation (animated)

- What if we want to know if **224** is in Tree?



# Searching Simulation - Recap

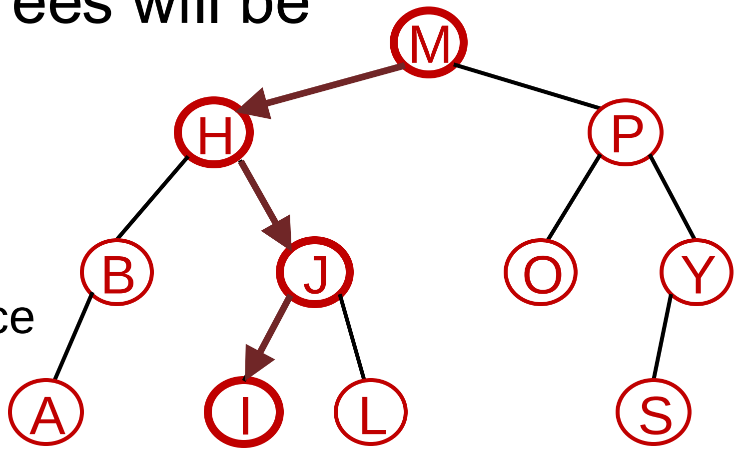



- What if we want to know if **224** is in Tree?
- Tree says "Hey Root! Ya got **224**?"
- **123** says: "Let's see. I'm not **224**. But if **224** is in tree, it would be to my right. I'll ask my right child and return its answer."
- **252** says: "I'm not **224**, it's smaller than me. I better ask my left child and return its answer."
- **224** says: "**224**? That's me! Hey, caller **(252)** here's your answer." (returning node indicates that query is in tree)
- **252** says: "Hey, caller **(123)**! Here's your answer."
- **123** says: "Hey, Tree! Here's your answer."

Andries van Dam © 2024 11/12/24

# Searching a BST Recursively Is $O(\log_2 N)$

- Search path: start with root **M** and choose path to **I** (for a reasonably balanced tree, **M** will be more or less "in the middle," and left and right subtrees will be roughly the same size)
  - structurally, the height of a reasonably balanced tree with *n* nodes is about $log_2 n$
  - at most, we visit each level of the tree once
  - so, runtime performance of searching is $O(\log_2 N)$ as long as tree is reasonably balanced, which will be true if entry order is reasonably random (slide 87)




Andries van Dam © 2024 11/12/24

# Appendix

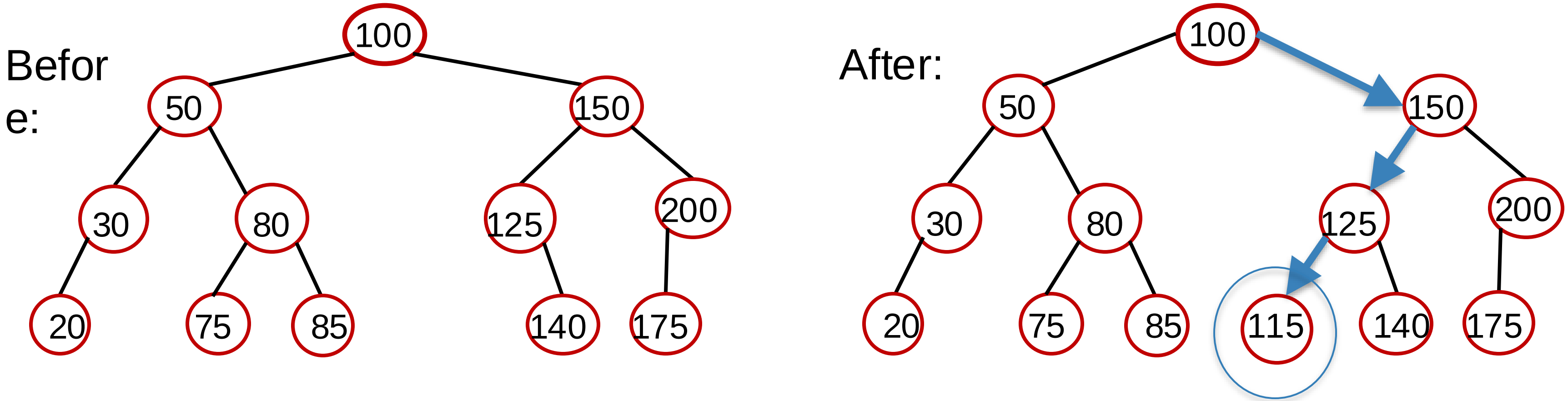
- Searching Simulation
- Insertion Demonstration
- Remove

# Insertion into a BST(1/2)

- Search BST starting at root until we find where the data to insert belongs
  - insert data when we reach a `Node` whose appropriate L or R child is `null`
- That `Node` makes a new `Node`, sets the new `Node`'s `data` to the data to insert, and sets child reference to this new `Node`
- Runtime is $O(\log_2 N)$, yay!
  - $O(\log_2 N)$ to search the nearly balanced tree to find the place to insert
  - constant time operations to make new `Node` and link it in

# Insertion into a BST(2/2)

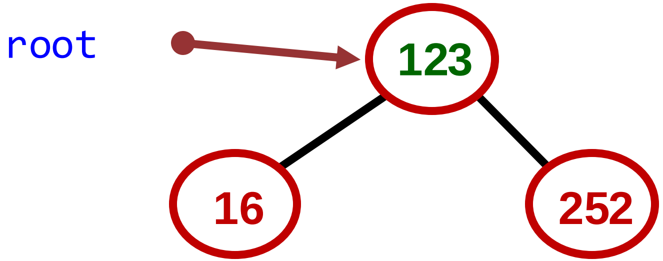
- Example: Insert **115**



# Insertion Simulation (1/4)

- Insert: **224**
- First call `insert` in BST:

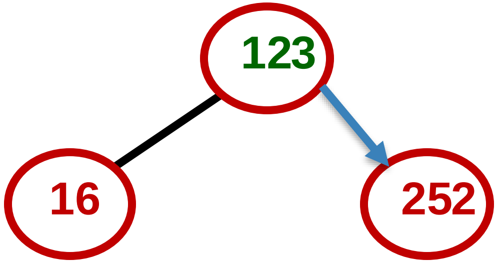
```
this.root = this.root.insert(newData);
```



# Insertion Simulation (2/4)

- **123** says: "I am less than **224**. I'll let my right child deal with it."

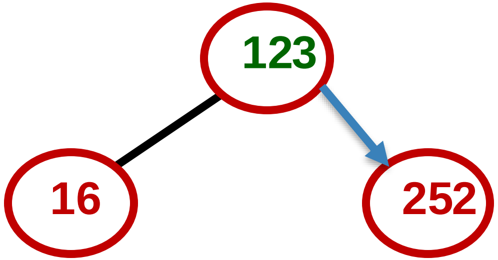
```
if (this.data.compareTo(newData) > 0) {
       //code for inserting left elided
} else {
       if(this.right == null) {
           //code for inserting with null right child elided
       } else {
           return this.right.insert(newData);
       }
}
```



# Insertion Simulation (3/4)

- **252** says: "I am greater than **224**.  I'll pass it on to my left child – but my left child is `null`!"

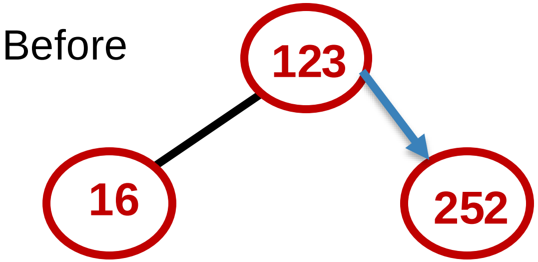
```
if (this.data.compareTo(newData) > 0) {
    if(this.left == null) {
        this.left = new Node(newData, this);
        return this.left;
    } else {
        //code for continuing traversal elided
    }
}
```

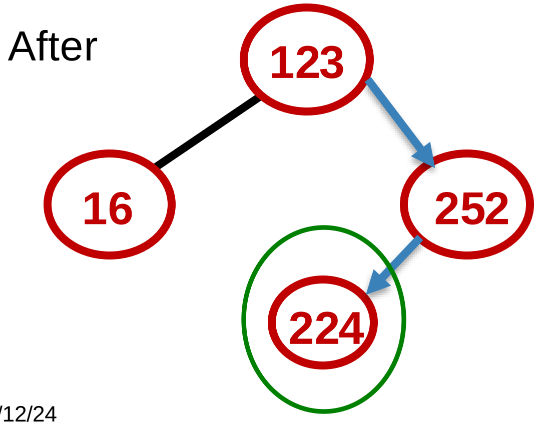


# Insertion Simulation (4/4)

- 252 says: "You belong as my left child, 224. Let me make a node for you, make this new node your home, and set that node as my left child. Lastly, I will return a pointer to the new left node". (And each node, as its recursive invocation ends, passes the pointer to the new 224 node up to its parent, eventually up to whatever method called on the tree's search)

```
this.left = new Node(newData, this);
return this.left;
```



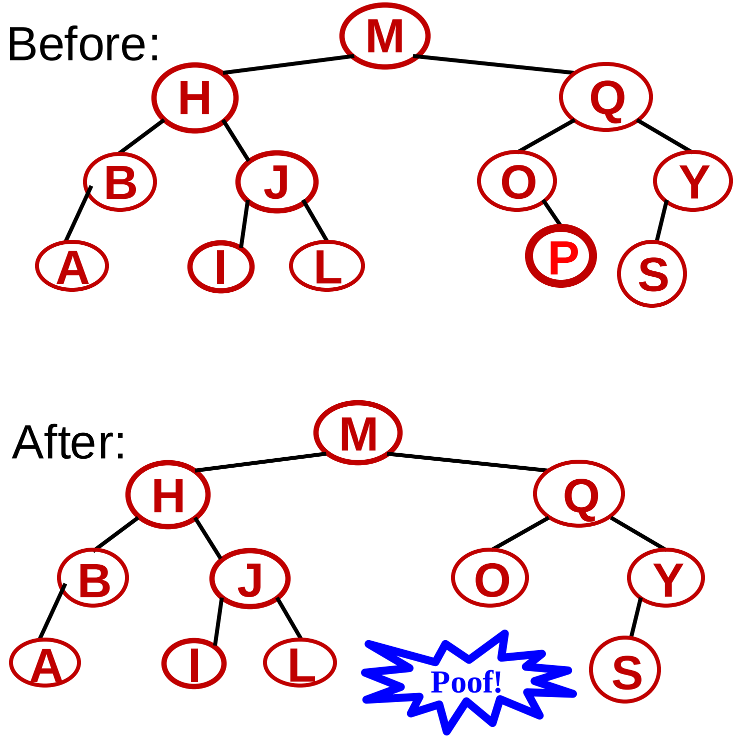



# Appendix

- Searching Simulation
- Insertion Demonstration
- Remove

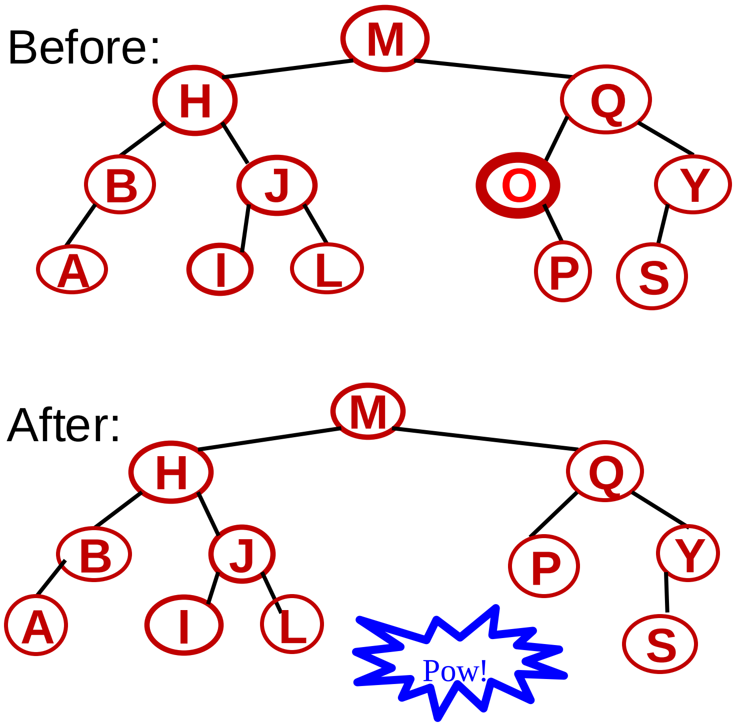
# Remove: No Child Case

- Node to remove has no children (is a leaf)
  - just set the parent's reference to this `Node` to `null` – no more references means the `Node` is garbage collected!
- Example: Remove **P**
  - set **O**'s right child to `null`, and **P** is gone!



# Remove: One-Child Case

- Harder case: Node to delete has one child
  - replace Node child
- Example: Remove **O**
  - **O** has one child
  - **Q** replaces **O** by replacing its left child, previously **O**, with **P**
  - we know that all of the children of **O** are less than **Q** and greater than **M**. So, making **O**'s child a child of **Q** results in a valid BST!



# Remove: Two-Children Case (1/3)

- Hard case: node to remove has two internal children
  - brute force: just flag node for removal, and rewrite tree at a later time -- bad idea, because now every operation requires checking that flag. Instead, do the work right away
  - this is tricky, because not immediately obvious which child should replace its parent
  - slow solution: re-insert each member of one of the sub-trees. Might be bad, O(n), even for a balanced tree.
  - non-obvious solution: first swap the data in Node to be removed with data in a Node that doesn't have two children, then remove Node using one of simpler remove cases


Andries van Dam © 2024 11/12/24

# Remove: Two-Children Case (2/3)

- Use an auxiliary method, `swapData`
  - swaps data in node to be removed with the data in the right-most node in its left subtree
  - this child has a key value less than all `Node`s in the to-be removed `Node`'s right subtree, and greater than all other nodes in its left subtree
  - since it is a right-most `Node`, it has at most one child because if it is the right most child, it won't have any right children
  - this swap is temporary–we then remove the node in the right-most position using simpler `remove`

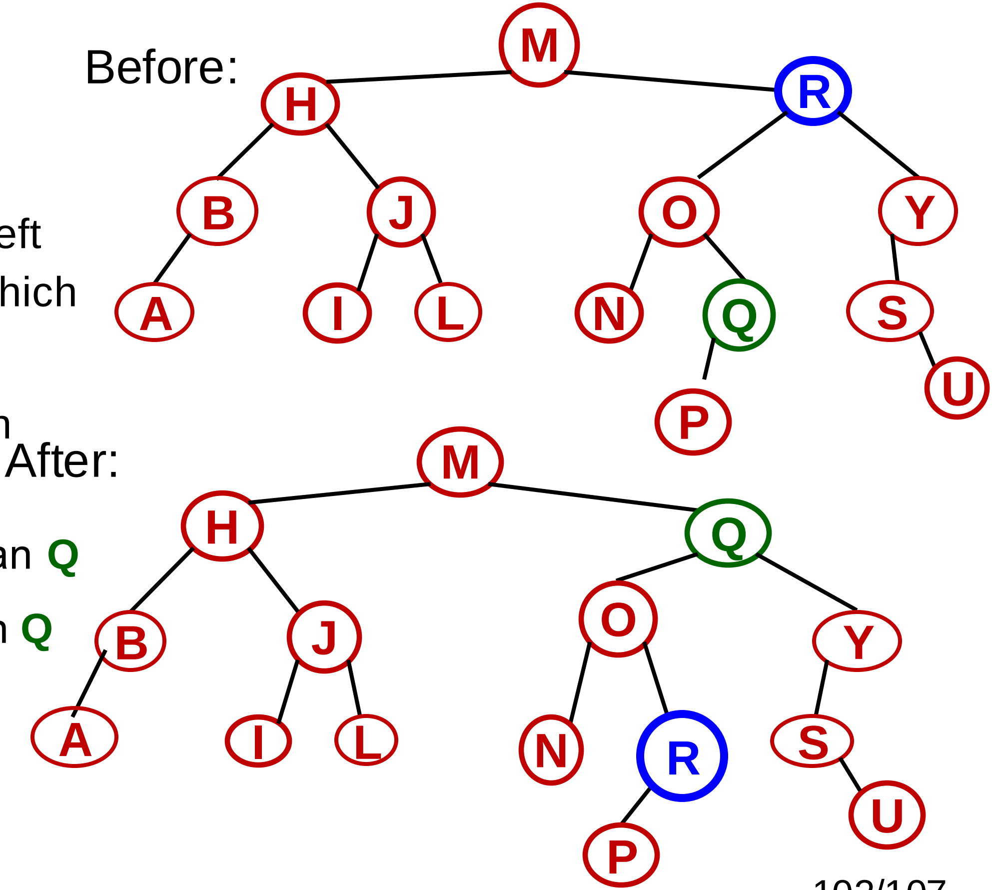
# Remove: Two-Children Case (3/3)

- How do we remove **R**?
  - **R** has two children
  - swap **R** with the right-most `Node` in the left subtree, the largest `Node` less than **R**, which will be **Q**
  - observe the following relationship, which must be maintained after the swap:

    - children in **R**'s left subtree are smaller than **Q**

    - children in **R**'s right subtree are larger than **Q**

    - **R** is in the wrong place but…
  - remove **R** (in its new position) using the one-child case



# Remove: BST Code

- Starts as usual with delegating to root
- Nodes are "smart," so they can remove themselves
- Need to first find the Node to remove; if not null, it removes itself
- $O(\log_2 N)$ because of searching in a nearly balanced tree

```
// in BinarySearchTree class:
//BinarySearchTree's remove takes a data element
public void remove(Type itemToRemove) {
    Node<Type> toRemove = this.root.search(itemToRemove);
    if (toRemove != null) {
        //smart node's remove takes no params
        toRemove.remove();
    }
}
```

# Remove: Node Code (1/3)

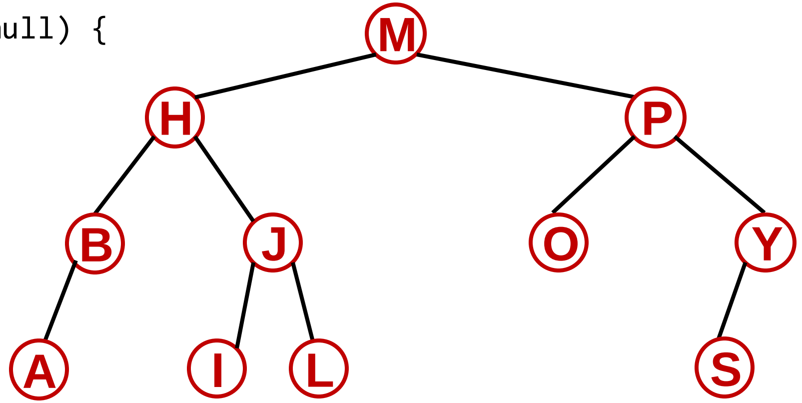
- In the Node class, remove method allows Node to remove itself

```
public Node<Type> remove() {
    //Case 1 – Node to remove is a leaf node
    //Set its parent's reference that originally refers to this Node to null
    if(this.left == null && this.right == null) { //if it's a leaf, set appropriate parent to null
        if(this.parent.getLeft() == this) {
            this.parent.setLeft(null);
        } else {
            this.parent.setRight(null);
        }
    }
    //Code for other cases on next slides…
}
```

Note: because a node removes itself, it compares the parents' child pointers to itself via `this`

# Remove: Node Code (2/3)

```
public Node<Type> remove() {  //code for case 1 elided
    //In a one-child case, we replace the _parent's reference to Node with the Node's child
    } else if (this.left != null && this.right == null) {
        //case 2.1 – Node only has left child
        if (this.parent.getLeft() == this) {
            this.parent.setLeft(this.left);
        } else {
            this.parent.setRight(this.left);
        }
    } else if (this.left == null && this.right != null) { //case 2.2 – Node has only right child
        if (this.parent.getLeft() == this) {
            this.parent.setLeft(this.right);
        } else {
            this.parent.setRight(this.right);
        }
    } //Case 3 on next slide …
}
```



# Remove: Node Code (3/3)

- Successor is guaranteed to have at most one child, so we remove with simpler remove case

```
public Node<Type> remove() {
    //code for case 1 (no children) elided
    //code for case 2 (one child) elided
    } else { //case 3 – both children
        Node<Type> toSwap = this.swapItem(); //swap data with successor
        toSwap.remove(); //now remove toSwap, which holds original Node’s data
        return toSwap; //return toSwap, since toSwap was data we removed
    }
    return this; //return this if we didn’t do any swapping since Node is removed
}
//swapData() defined on next slide
```

# Remove: `swapItem` Code

- We find the right-most `Node` in left subtree, but we can also find the left-most `Node` in right subtree

```
public Node<Type> swapItem(){
    Node<Type> curr = this.left; //first get left child
    while(curr.getRight() ! = null) { //go right as far as possible
        curr = curr.getRight();
    }
    //swap data of this Node and successor
    Type tempItem = this.item;
    this.data = curr.getItem();
    curr.setItem(tempItem);
    return curr;
}
```

# Tech for Good

Topics in Socially Responsible Computing

***Be a pragmatic techno-optimist!***

# Health Tech: AI & Medical Diagnostics

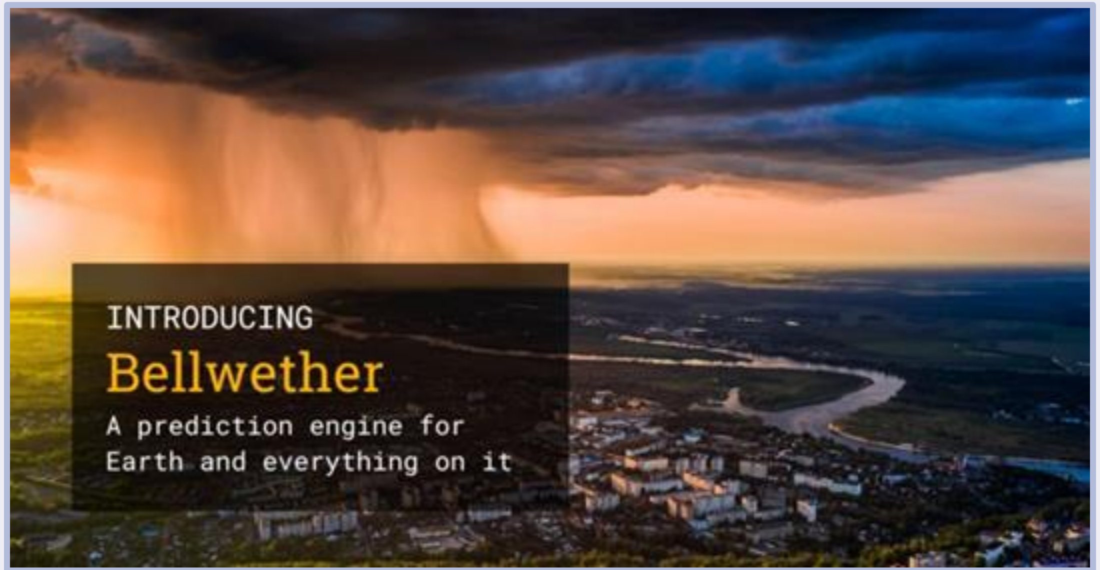
## In a first, FDA authorizes AI-driven test to predict sepsis in hospitals

3 min · 77

By Daniel Gilbert and Rachel Roubein

Updated April 3, 2024 at 5:31 p.m. EDT | Published April 3, 2024 at 12:02 p.m. EDT

# Environmental Tech: Monitoring & Response




TechCrunch

**Alphabet X's Bellwether harnesses AI to help predict natural disasters**

Brian Heater
April 17, 2024 · 2 min read

# EdTech: Digital Learning Platforms

## Howard University Expands College Math Course Developed with Khan Academy to Historically Underserved Schools Nationwide

NEWS PROVIDED BY
Khan Academy →
Oct 04, 2022, 09:00 ET

SHARE THIS ARTICLE

# Computational Linguistics in Research

## How AI Translation Is Helping Scientists Understand Ancient Languages

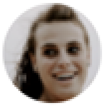
Deep learning algorithms tackle cuneiform writing and more

# Assistive Technologies

News > Initiative to use AI technologies to improve communication in ALS

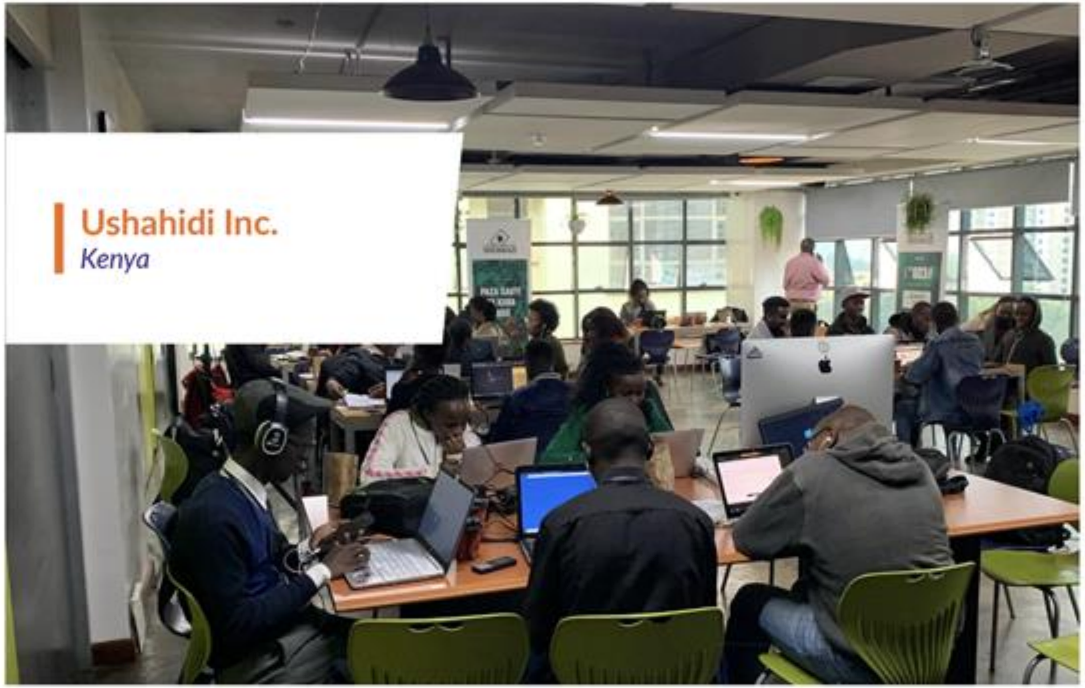
## Initiative to use AI technologies to improve communication in ALS

Scott-Morgan Foundation teaming up with tech companies on new project



by *Marisa Wexler, MS* | October 17, 2024

# Civic Tech




*Uchaguzi: Harnessing citizen power for peaceful and fair elections in Kenya*

# Not far from home...

**Humanoid Robots Teach Coping Skills to Children With Autism** Roboticist Ayanna Howard explains what inspired her to work on assistive technologies for kids

BY KATHY PRETZ | 10 JUL 2019 | 5 MIN READ |

zyrobotics.com

Ayanna Howard ’93

# Not far from home...

Amélie-Sophie Vavrovsky ’18

Formally

# Not far from home...

beam

Viveka Hulyalkar ’15 and co-founder Alex Sadhu

***“We must not fixate on what this new arsenal of digital technologies allows us to do without first inquiring what is worth doing.”***

- Evgeny Morozov, *To Save Everything, Click Here: The Folly of Technological Solutionism*