Lecture 8

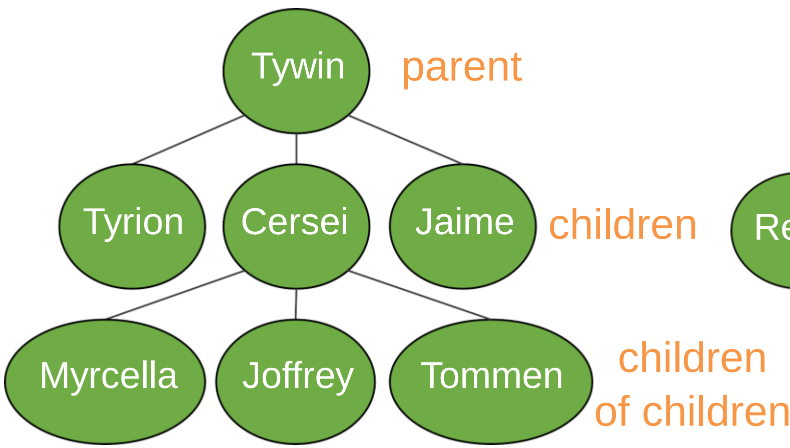
Binary Search Tree vs. Trie

Jianchen Shan
Department of Computer Science
Hofstra University

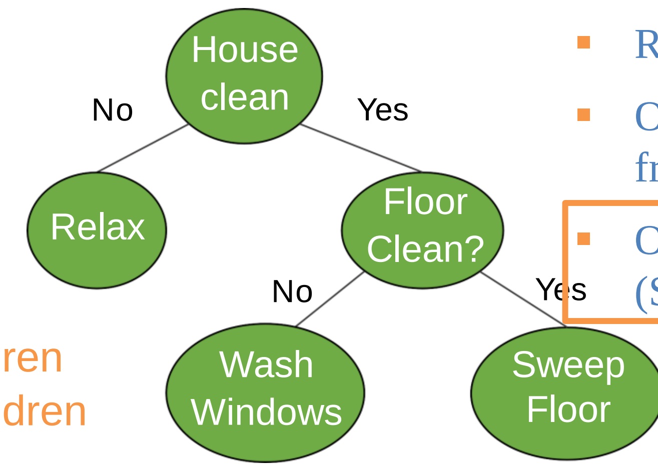
Lecture Goals

- Describe the value of trees and their data structure
- Explain the need to visit data in different orderings
- Perform pre-order, in-order, post-order and level-order traversals
- Define a Binary Search Tree
- Perform search, insert, delete in a Binary Search Tree
- Explain the running time performance to find an item in a BST
- Compare the performance of linked lists and BSTs
- Explain what a trie data structure is
- Describe the algorithm for finding keys in and adding keys to a trie
- Compare the time to find a key in a BST to a trie
- Implement a trie data structure in Java

Different Trees in Computer Science



Family Trees

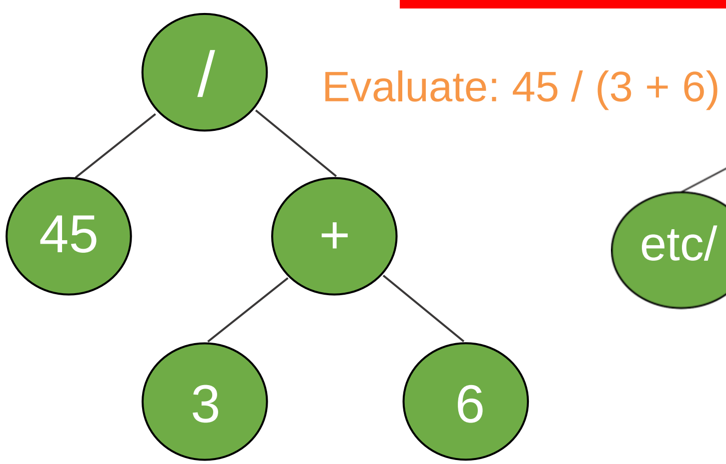


Decision Trees

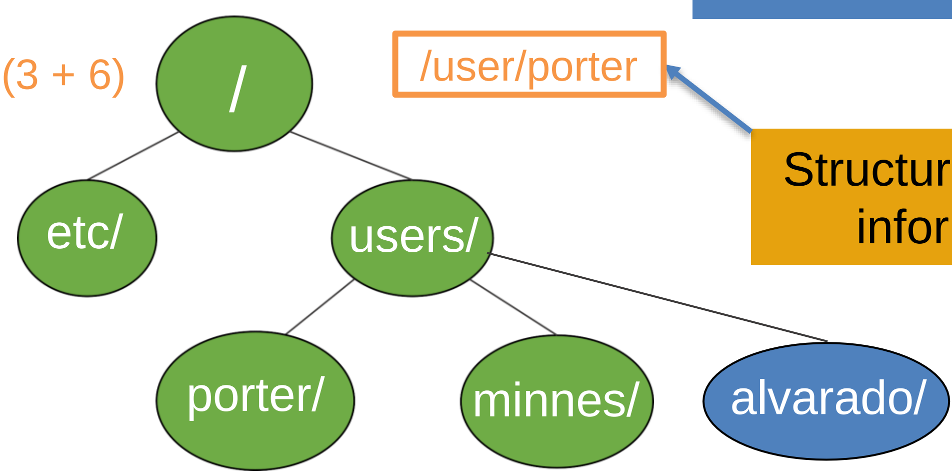
- Root is most important (Heap)
- Organized by character frequency (Huffman Tree)
- Organized by node ordering (Search Trees)
- Etc…

Why trees?

Different Organizations → Different Trees



Expression Trees

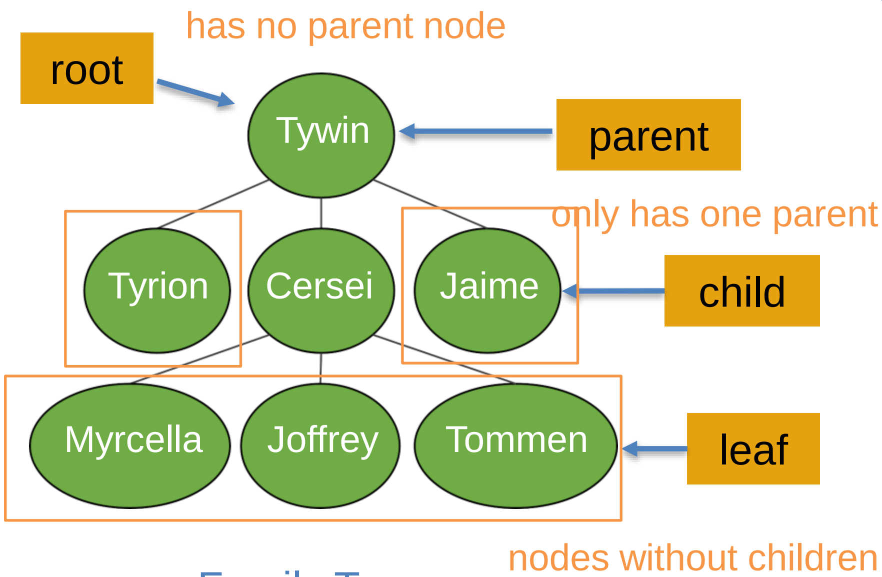


File System

Structure conveys information

Dynamic Data Structure

Defining Trees

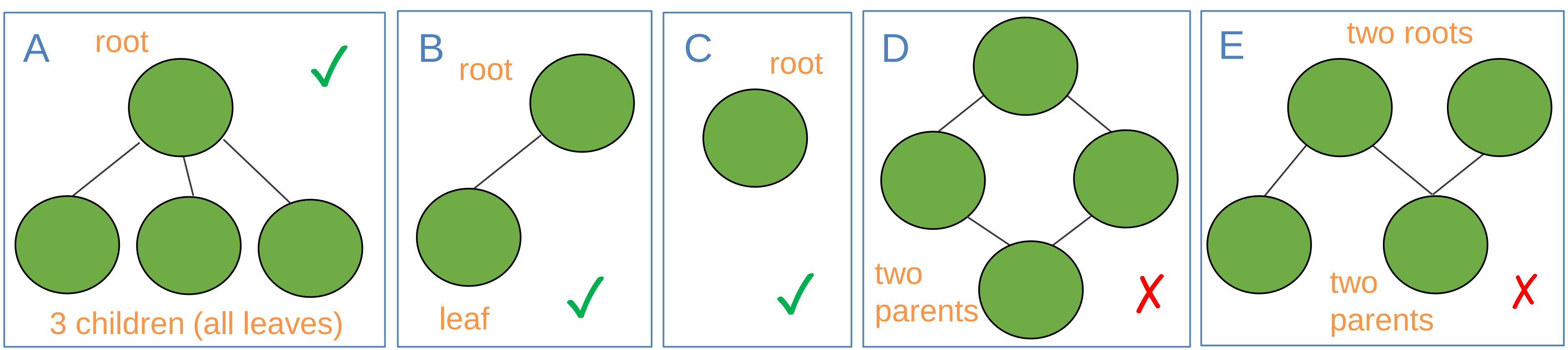


Family Trees

What defines a tree?

- Single root
- Each node can have only one parent (except for root)
- No cycles in a tree

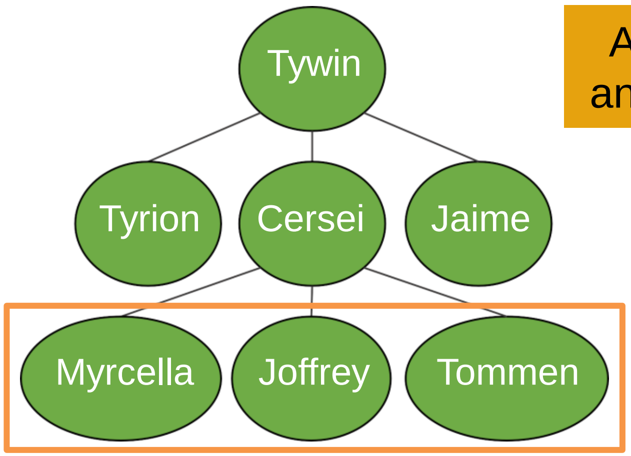
Which are trees?



Cycle: two different paths between a pair of nodes

Binary Trees

Generic Tree

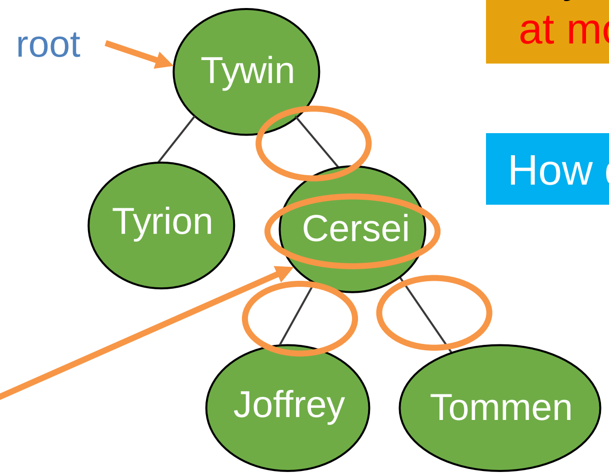


Any Parent can have any number of children

How would a general tree node differ?

A general tree would just have a list for children

Binary Tree



Any Parent can have at most two children

A tree just needs a root node

like the head and tail for linked list

How do we construct a tree?

Each node needs:

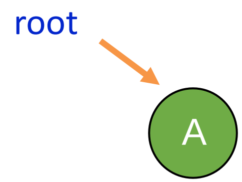
1. A value
2. A parent
3. A left child
4. A right child

Like Linked Lists, Trees have a "Linked Structure"

nodes are connected by references

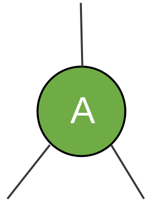
Write Code for Binary Tree

```
public class BinaryTree<E> {
    TreeNode<E> root;
    // more methods
}
```




```
public class TreeNode<E> {
        private E value;
        private TreeNode<E> parent;
        private TreeNode<E> left;
        private TreeNode<E> right;
        public TreeNode(E val, TreeNode<E> par) {
                this.value = val;
                this.parent = par;
                this.left = null;
                this.right = null;
        }
        public TreeNode<E> addLeftChild(E val) {
                this.left = new TreeNode<E>(val, this);
                return this.left;
        }
}
```

For root: TreeNode(val, null)



Let's write a constructor together

Next Step is to able to set/get children

Fill in the blank:

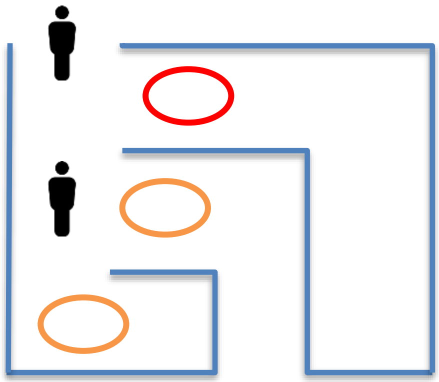
A. this.parent
B. this.left
C. this.right
D. this

Tree Traversal - Motivation

Warning: These first examples are really graphs. We'll visit graphs in detail in the next course. Here they are used as motivating examples

start

Strategy: go until hit a dead end, then retrace steps and try again



finish

Maze Traversal

Imagine this is a hedge maze

What's my next step?

Mazes benefit from "Depth First Traversals"

Bottom line: Order we visit matters and we'll make choices based on our needs

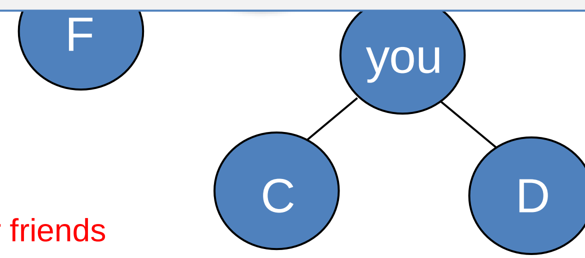
Suppose you have a list of your friends and each of your friends have lists

How closely are you connected with D?

What's my next step?

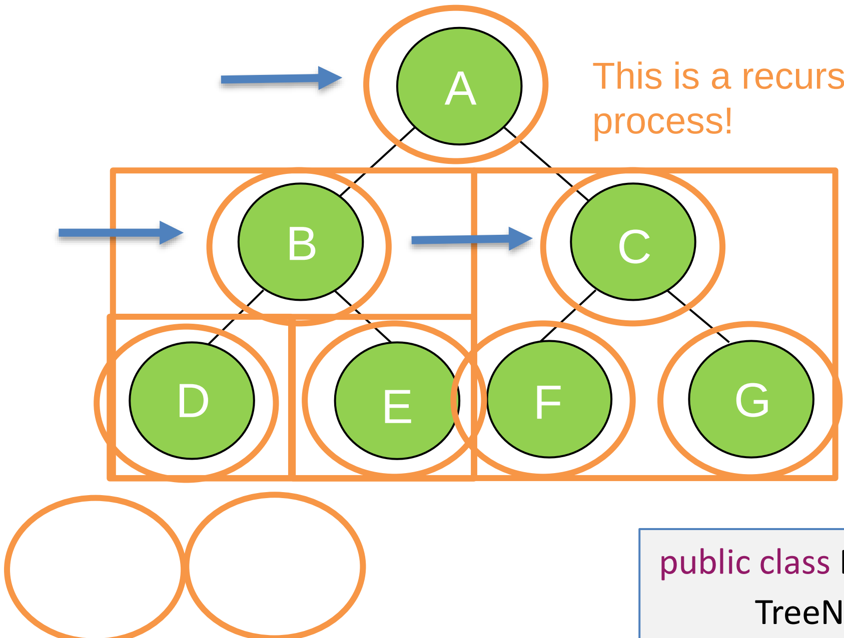
Strategy: look at all of your friends first, and then branch out.

This problem benefits from "Breadth First Traversals"



Social Network

Pre-order Traversal (Recursively)



This is a recursive process!

Idea:
- Visit yourself
- Then visit all your left subtree
- Then visit all your right subtree

Visited:

A B D E C F G

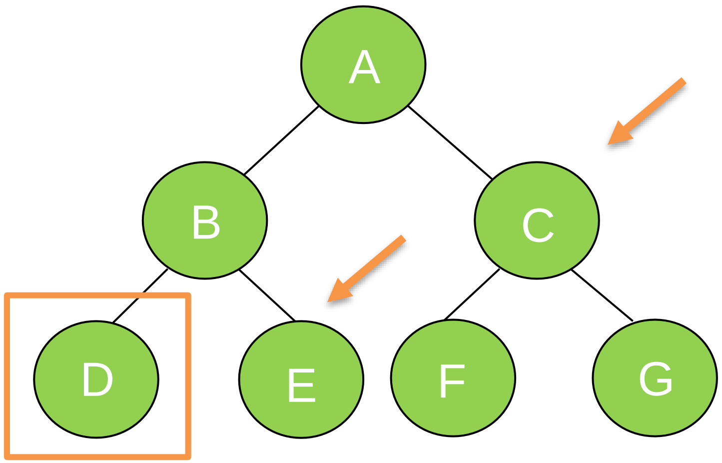
What's the order in which you think the nodes will be visited?

Recursion will help us do this!

This can be done iteratively

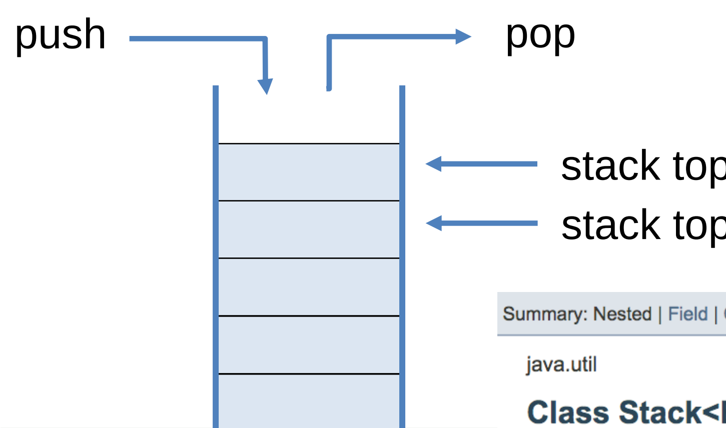
```
public class BinaryTree<E> {
        TreeNode<E> root;
        private void preOrder(TreeNode<E> node) {
                if(node!= null) {
                        node.visit();
                        preOrder(node.getLeftChild());
                        preOrder(node.getRightChild());
                }
        }
        public void preOrder() {
                this.preOrder(root);
        }
}
```

Pre-order Traversal (Iteratively)



Challenging: When we finish D, how do we go to E and C next?

Idea: Keep a list and keep adding to it and removing from end.



Visit: A B D E C F G

List: ~~A~~ ~~C~~ ~~B~~ ~~E~~ ~~D~~ ~~G~~ ~~F~~

We used this list like a "Stack"

- Add to the top
- Remove from the top
- Last-In, First-Out (LIFO)

Summary: Nested | Field | Constr | Method

java.util

Class Stack<E>

Methods

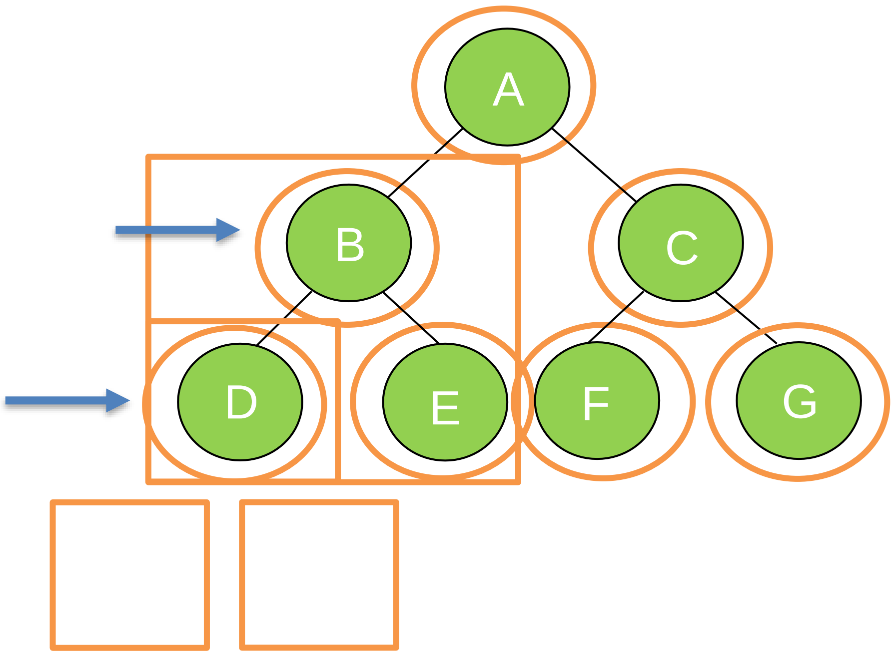
Modifier and Type	Method and Description
boolean	`empty()` Tests if this stack is empty.
E	`peek()` Looks at the object at the top of this stack w
E	`pop()` Removes the object at the top of this stack
E	`push(E item)` Pushes an item onto the top of this stack.

Pre-order Traversal (Iteratively)

```
public class BinaryTree<E> {
        TreeNode<E> root;

        void iterativePreorder(TreeNode<E> par) {
                if (par == null) { return; }
        Stack<TreeNode<E>> nodeStack = new Stack<TreeNode<E>>();
        nodeStack.push(par);

                while (nodeStack.empty() == false) {
                        TreeNode<E> node = nodeStack.peek();
                node.visit();
                nodeStack.pop();
                if (node.right != null) {
                                nodeStack.push(node.right);
                }
                if (node.left != null) {
                nodeStack.push(node.left);
                }
                }
        }
   void iterativePreorder() {
      iterativePreorder(root);
   }
}
```

1) Create an empty stack *nodeStack* and push root node to stack.
2) Do following while *nodeStack* is not empty.
….a) Pop an item from stack and print it.
….b) Push right child of popped item to stack
….c) Push left child of popped item to stack
Right child is pushed before left child to make sure that left subtree is processed first.

Post-order and In-order Traversal



Visit: Post-order

D E B F G C A

REARRANGE:
- ? Visit yourself
- ? Visit all your left subtree
- ? Visit all your right subtree

- Visit all your left subtree
- Visit all your right subtree
- Visit yourself

Visit: In-order

____

What does this do?
- Visit all your left subtree
- Visit yourself
- Visit all your right subtree

Fill in the Blank:

A. A B C D E F G

B. D B E A F C G

C. D B A E F C G

Recursion will help us do these!

They can also be done iteratively with Stack.

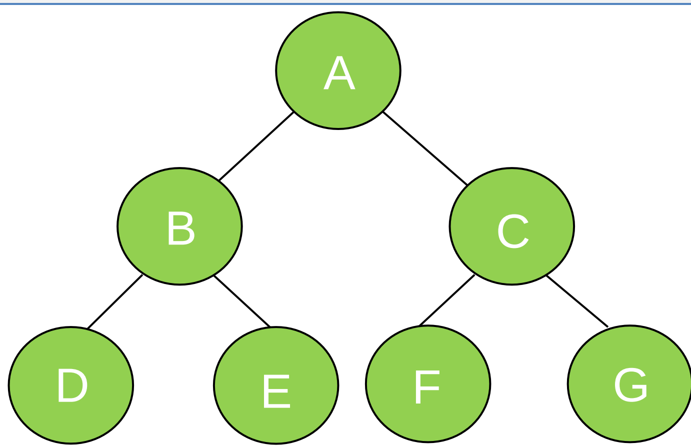
Post-order Traversal (Recursively and Iteratively)

Recursive

```
public class BinaryTree<E> {
   TreeNode<E> root;

   public void Postorder(TreeNode<E> node) {
      if (node == null)
         return;

      Postorder(node.left);
      Postorder(node.right);
      node.visit();
   }
   void Posterorder() {Postorder(root); }
}
```

For iterative version, the idea is to push reverse postorder traversal to a stack. Then, we can just pop all items one by one from the stack and visit them. To get reversed postorder elements in a stack – the second stack is used for this purpose. We can observe that this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child.

1. Push root to first stack.
2. Loop while first stack is not empty
….2.1 Pop a node from first stack and push it to second stack
….2.2 Push left and right children of the popped node to first stack
3. Visit contents of second stack



Visit: D E B F G C A

Stack 2: ~~A~~ ~~C~~ ~~G~~ ~~F~~ ~~B~~ ~~E~~ ~~D~~

Stack 1: ~~A~~ ~~B~~ ~~C~~ ~~F~~ ~~G~~ ~~D~~ ~~E~~

Iterative

```
public class BinaryTree<E> {
  TreeNode<E> root;
  public void iterativePostorder() {
   Stack<TreeNode<E>> s1 = new Stack<TreeNode<E>>();
   Stack<TreeNode<E>> s2 = new Stack<TreeNode<E>>();
   if (root == null)
      return;
   s1.push(root);
   while (!s1.isEmpty()) {
      TreeNode<E> temp = s1.pop();
      s2.push(temp);

      if (temp.left != null)
         s1.push(temp.left);
      if (temp.right != null)
         s1.push(temp.right);
   }
   while (!s2.isEmpty()) {
        TreeNode<E> temp = s2.pop();
        temp.visit();
   }
  }
}
```

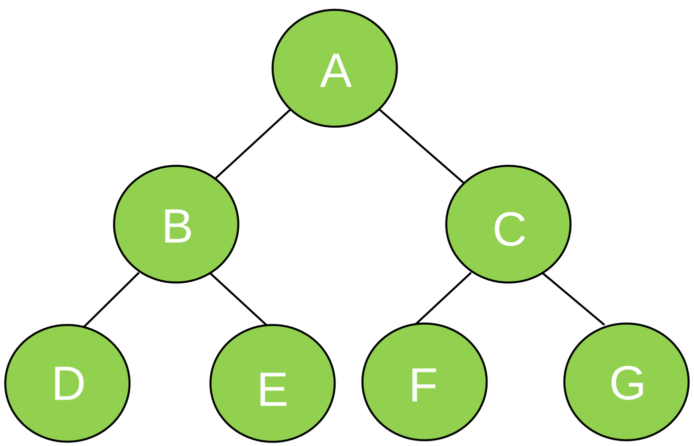
visit all elements of second stack

In-order Traversal (Recursively and Iteratively)

Recursive

```java
public class BinaryTree<E> {
  TreeNode<E> root;

  public void Inorder(TreeNode<E> node) {
    if (node == null)
      return;

    Inorder(node.left);
    node.visit();
    Inorder(node.right);
  }
  void Inorder() { Inorder(root); }
}
```

1) Create an empty stack S.
2) Initialize current node as root
3) If current is not NULL, push the current node to S and set current = current->left. Repeat until current is NULL
4) If current is NULL and stack is not empty then
….a) Pop the top item from stack.
….b) Print the popped item, set current = popped_item->right
….c) Go to step 3.
5) If current is NULL and stack is empty then we are done.



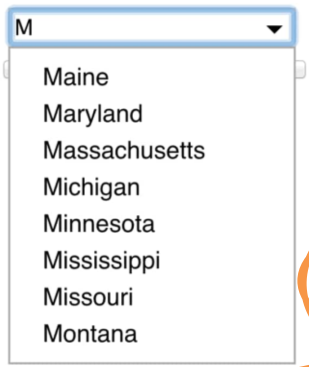
Visit: D B E A F C G

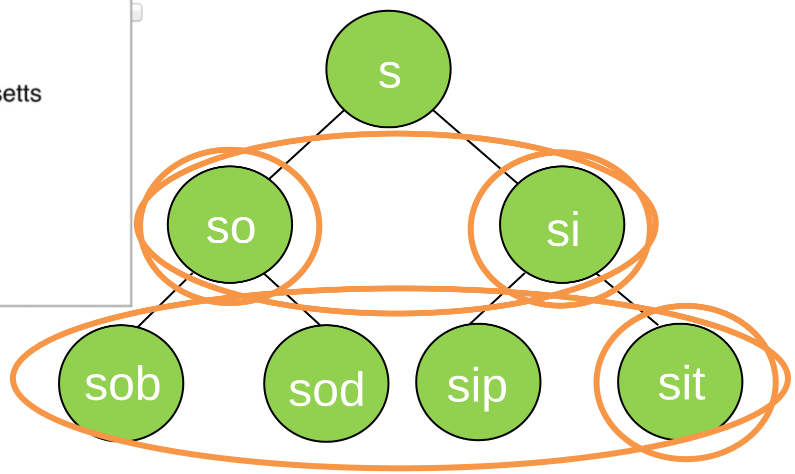
Stack: ~~A~~ ~~B~~ ~~D~~ ~~E~~ ~~C~~ ~~F~~ ~~G~~

Iterative

```java
public class BinaryTree<E> {
  TreeNode<E> root;

  public void iterativeInorder() {
    if (root == null)
      return;

    Stack<TreeNode<E>> s = new Stack<TreeNode<E>>();
    TreeNode<E> curr = root;

    while (curr != null || s.empty() == false) {
      while (curr !=  null) {
        s.push(curr);
        curr = curr.left;
      }
      curr = s.pop();
      curr.visit();
      curr = curr.right;
    }
  }
}
```

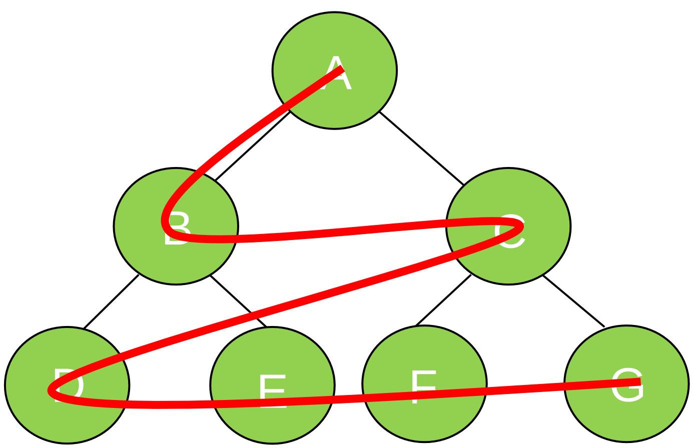
Level-order Traversal





- You've typed "s" What words should we suggest?
- Most frequent?
- Most frequent for whom?
- How about "closest"?

"Breadth First Traversal"



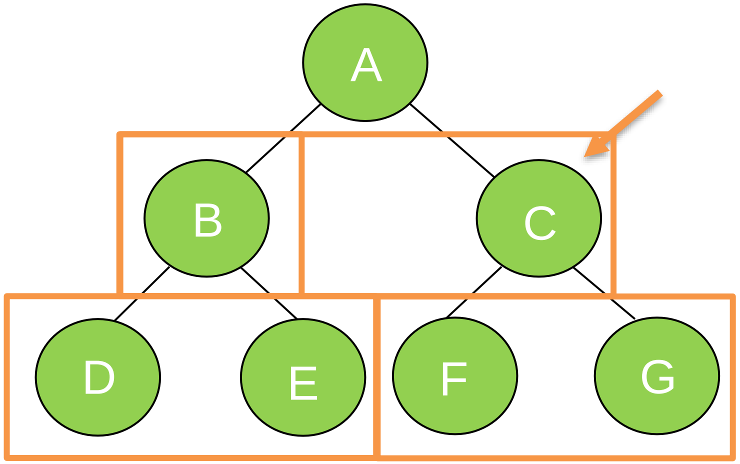
Visit: Level-order

A B C D E F G

Level-order is "Breadth First Traversal”

Post/Pre Order are: "Depth First Traversals"

Level-order Traversal (Contd.)



Visit: A B C D E F G

List: ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~G~~

We used this list like a "Queue"

- Add to the end
- Remove from the front
- First-In, First-Out (FIFO)

Visit:

A B C D E F G

Challenging: When we finish B, how do we go to C next?

Idea: Keep a list and keep adding to it and removing from start.

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface Queue<E>

	Throws exception
Insert	add(e)
Remove	remove()
Examine	element()

look at the first element

Level-order Traversal Implementation

Linkedlist implements both list and queue interfaces

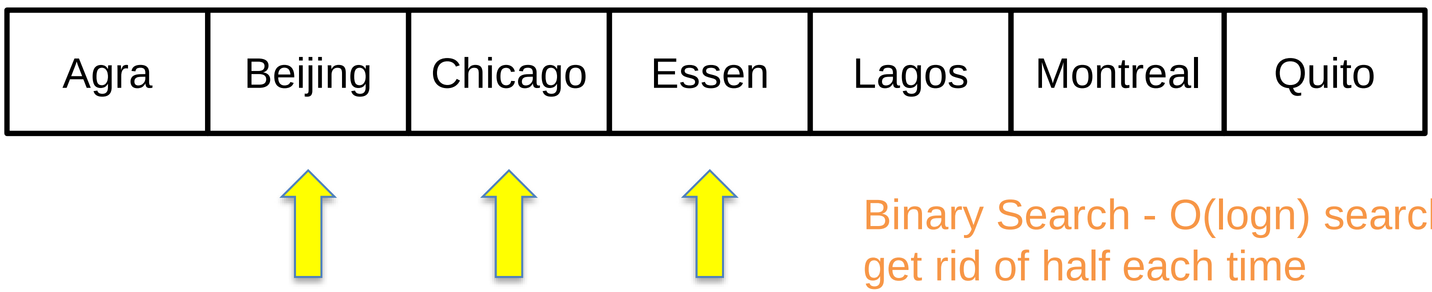
```
public class BinaryTree<E> {
        TreeNode<E> root;
        public void levelOrder() {
                Queue<TreeNode<E>> q = new LinkedList<TreeNode<E>>();
                q.add(root);
                while(!q.isEmpty()) {
                        TreeNode<E> curr = q.remove();
                if(curr != null) {
                        curr.visit();
                        q.add(curr.getLeftChild());
                        q.add(curr.getRightChild());
                }
        }
    }
}
```

Could also check for null children before adding

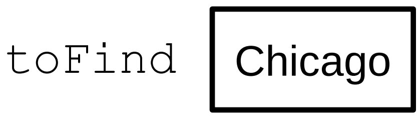
Binary Tree Traversals Analysis

- In recursive tree traversals, the Java execution stack keeps track of where we are in the tree
- In iterative traversals, the programmer needs to keep track!
  - An iterative traversal uses a container to store references to nodes not yet visited (*stack*, *queue*, etc.).
- Consider performance of a binary tree with n nodes
  - How many recursive calls are there at most?
    - For each node, 2 recursive calls at most
    - So, 2*n recursive calls at most
  - So, a traversal is O(n)
  - For iterative traversals, each node is visited constant times. Thus, it's also O(n).

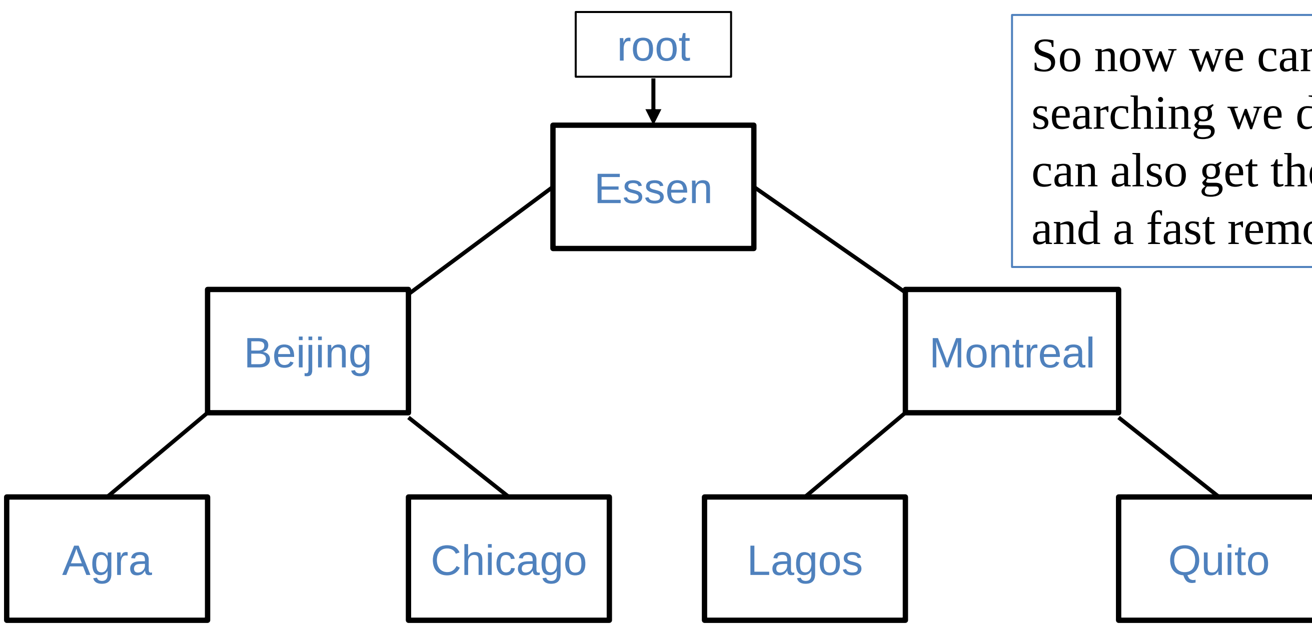
Motivation for Binary Search Tree



Binary Search - O(logn) search:
get rid of half each time

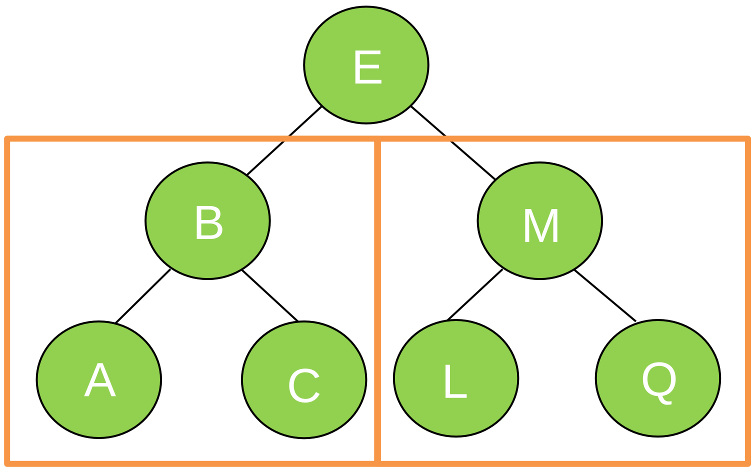


Sorted arrays are good for search,
but bad for insertion/removal



So now we can do the same kind of fast searching we did within an array, but we can also get the benefit of a fast insert and a fast removal that a tree provides.

Defining a Binary Search Tree



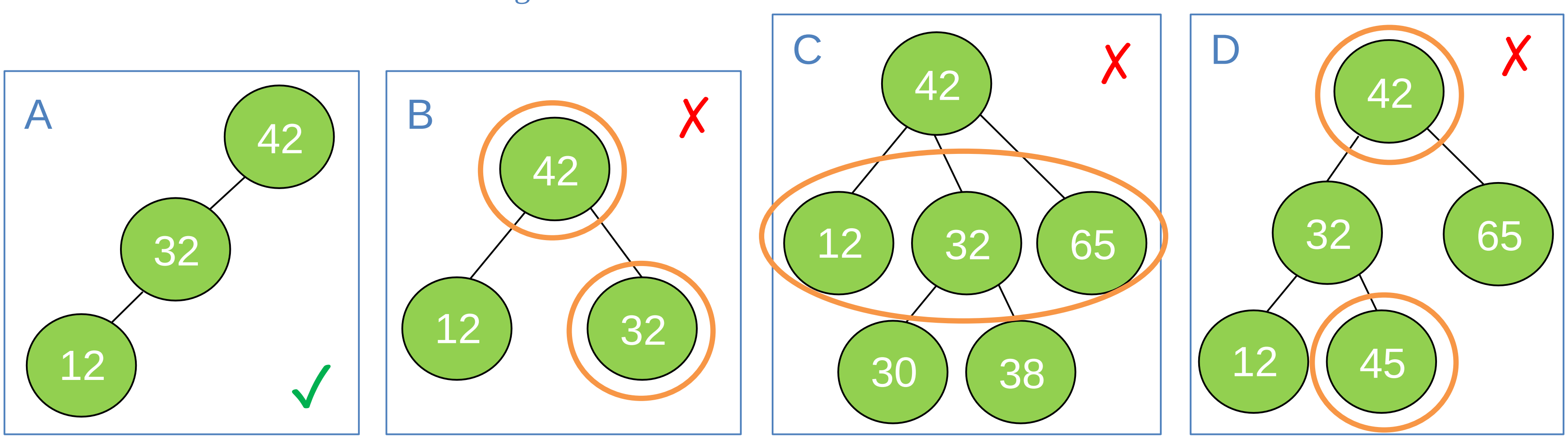
Left subtree's values must be lesser

Right subtree's values must be greater

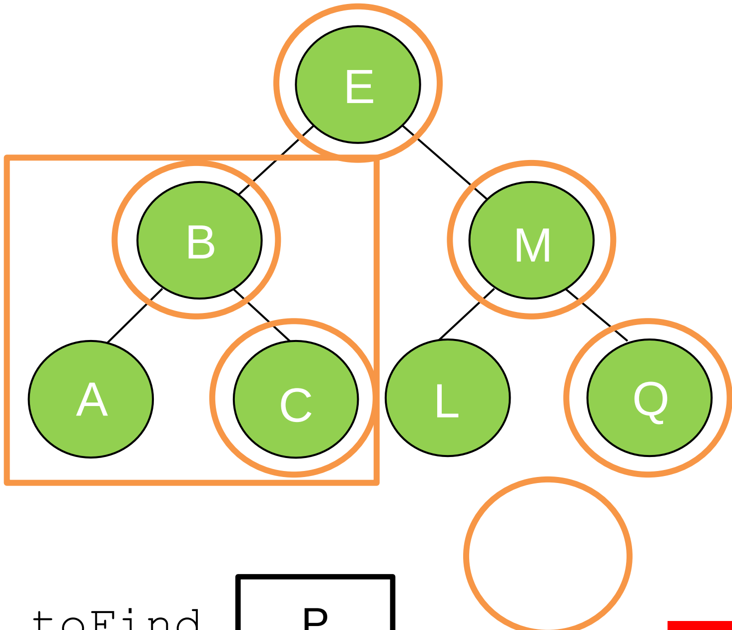
Binary Search Tree:

1. Binary Tree
2. Left subtrees are less than parent
3. Right subtrees are greater than parent

Which of these are binary search trees?



Searching a BST



Same fundamental idea as binary search of an array

toFind C

Compare: E and C

Compare: B and C

Compare: C and C

Found it!

toFind P

Compare: E and P

Compare: M and P

Compare: Q and P

Node is null

Not Found!

How to implement this?

You could solve this with recursion.

You could also solve it with iteration by keeping track of your current node.

Searching a BST Iteratively

```
public class BinaryTree<E> {
        TreeNode<E> root;
        public boolean search(E toSearch) {
                TreeNode<E> curr = root;
                while (curr != null) {
                        int comp = toSearch.compareTo(curr.getValue());
                        if (comp < 0)
                                curr = curr.getLeftChild();
                else if (comp > 0)
                        curr = curr.getRightChild();
                else // comp = 0
                        return true;
                }
                return false;
        }
}
```

```
<E extends Comparable<? super E>> {
```

It means that either the class E itself or one of its super classes implements Comparable

Doesn't work with objects

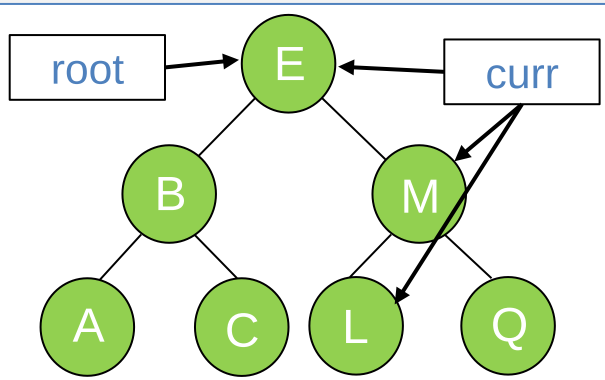
Do NOT change root pointer!

We need to do this over and over if not found

if calling object is less than parameter, compareTo returns a value < 0

if calling object is greater than parameter, compareTo returns a value > 0

Are we done?




```
t.search('L')
```

if calling object is equal to parameter, compareTo returns 0

Traverse down tree until:

a) end is reached
b) element is found

Searching a BST Recursively

```
public class BinaryTree<E extends Comparable<? super E>> {
   TreeNode<E> root;

      private boolean search(TreeNode<E> p, E toSearch) {
            if (p == null)
                  return false;
            int comp = toSearch.compareTo(p.getValue());
            if (comp == 0)
                  return true;
            else if (comp < 0)
                  return search(p.left, toSearch);
            else // comp > 0
                  return search(p.right, toSearch);
      }
      public boolean search(E toSearch) {
            return search(root, toSearch);
      }
}
```

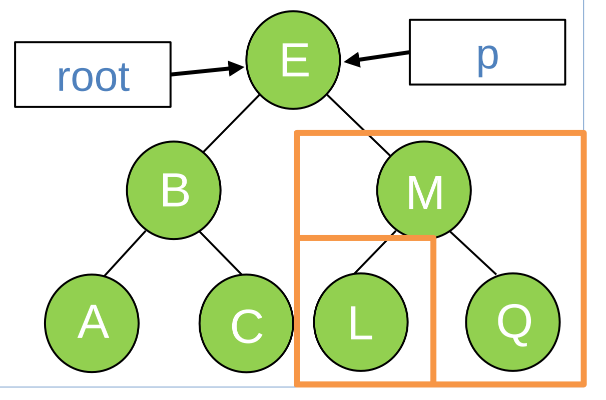
Root of the tree we look at

Tree is empty

Found it!

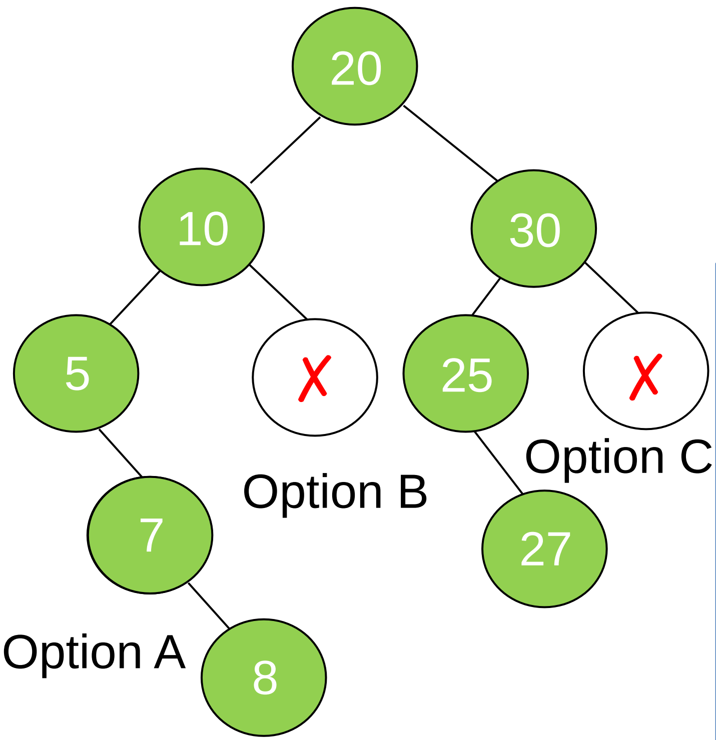
look left

look right




```
t.search('L')
```

Inserting into a BST



Where should we insert 7?

Insert 27?

Insert 8?

✗ Option D: Either Option A or Option B are fine.

Again, this is solved cleanly with either recursion or iteration.

```java
public class BinaryTree<E extends Comparable<? super E>> {
   TreeNode<E> root;

   public boolean recursizeInsert(TreeNode<E> p, E toInsert) {
      if (p == null) {
            p = new TreeNode<E>(toInsert, null);
                  return true;
      }
      int comp = toInsert.compareTo(p.getValue());
      if (comp == 0) { return false; }
      else if (comp > 0) {
            if (p.right == null) {
                  p.addRightChild(toInsert);
                  return true;
            } else { return recursizeInsert(p.right, toInsert);}
      }
      else { // comp < 0
            if (p.left == null) {
                  p.addLeftChild(toInsert);
                  return true;
            } else { return recursizeInsert(p.left, toInsert);}
      }
   }
   public boolean insert(E toInsert) {
      return recursizeInsert(root, toInsert);
   }
}
```

tree is empty

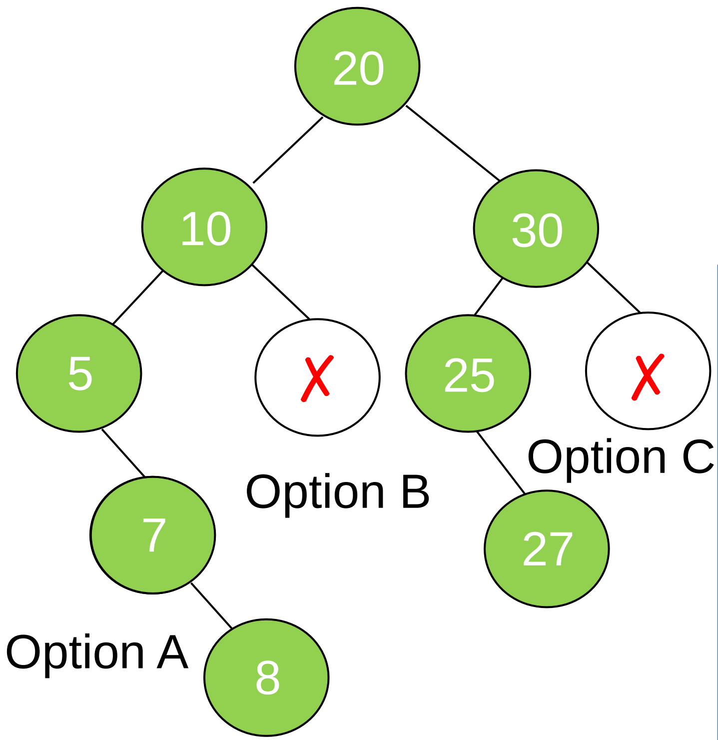
duplication is not allowed

add to the right three

add to the left three

Recursive

Inserting into a BST



Where should we insert 7?

Insert 27?

Insert 8?

```
public class BinaryTree<E extends Comparable<? super E>> {
   TreeNode<E> root;

   public TreeNode<E> recursizeInsert(TreeNode<E> p, E toInsert){
      if (p == null) {
            p = new TreeNode<E>(toInsert, null);
                  return p;
      }
      int comp = toInsert.compareTo(p.getValue());
      if (comp == 0) {}
      else if (comp > 0) {
                  p.right = recursizeInsert(p.right, toInsert);}
      }
      else { // comp < 0
            p.left = recursizeInsert(p.left, toInsert);}
      }

            return p;
   }
   public TreeNode<E> insert(E toInsert) {
     root = recursizeInsert(root, toInsert);
     return root;
   }
}
```

tree is empty

duplication is not allowed

✗ Option D: Either Option A or Option B are fine.

Again, this is solved cleanly with either recursion or iteration.

Recursive

Inserting into a BST (Iteratively)

```
public class BinaryTree<E extends Comparable<? super E>> {
   TreeNode<E> root;

   public boolean iterativeInsert(E toInsert) {
            TreeNode<E> curr = root;
            if (root == null) {
                     root = new TreeNode<E>(toInsert, null);
                     return true;
            }
            int comp = toInsert.compareTo(curr.value);
            while (comp < 0 && curr.left != null ||
        comp > 0 && curr.right != null) {
                     if (comp < 0) curr = curr.left;
                     else curr = curr.right;
                     comp = toInsert.compareTo(curr.value);
            }
            if (comp < 0)
                     curr.addLeftChild(toInsert);
            else if (comp > 0)
                     curr.addRightChild(toInsert);
            else // comp = 0
                     return false;
            return true;
        }
}
```

Where does curr point to?

tree is empty

search the location to insert from root

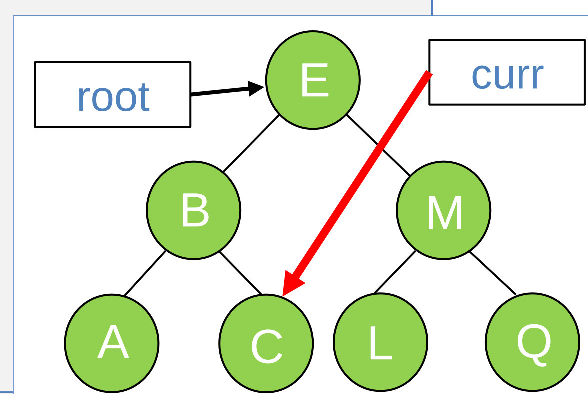
stop when find the location:

After the loop either: (1) curr points to the last node, or (2) we found the duplicate

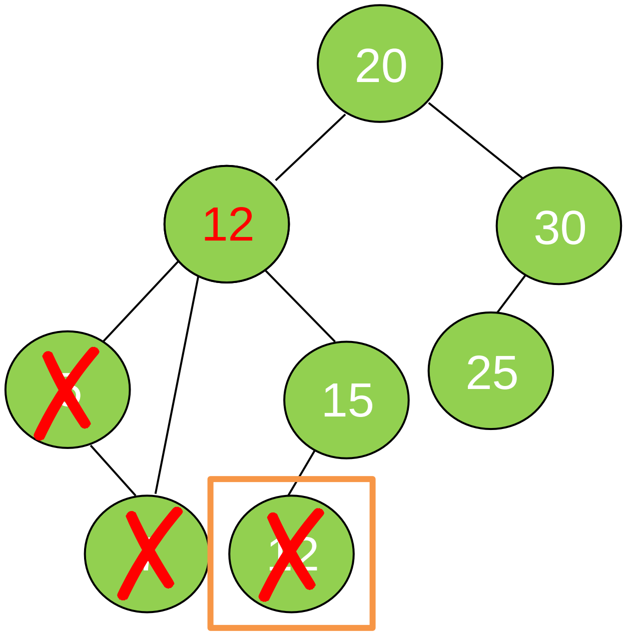
(1) curr points to the last node

(2) we found the duplicate

`t.iterativeInsert('D')`



Deleting from a BST



Which of the following is true about the smallest element in a node's right subtree?

A. Its left child is null

B. Its right child is null

C. Both of its children are null

Please implement it by yourself.

Delete 7

If leaf node: Delete parent's link 7

Delete 5

If only one child, hoist child

Delete 10

When a deleted node has two children, this gets tricky.

Find smallest value in right subtree

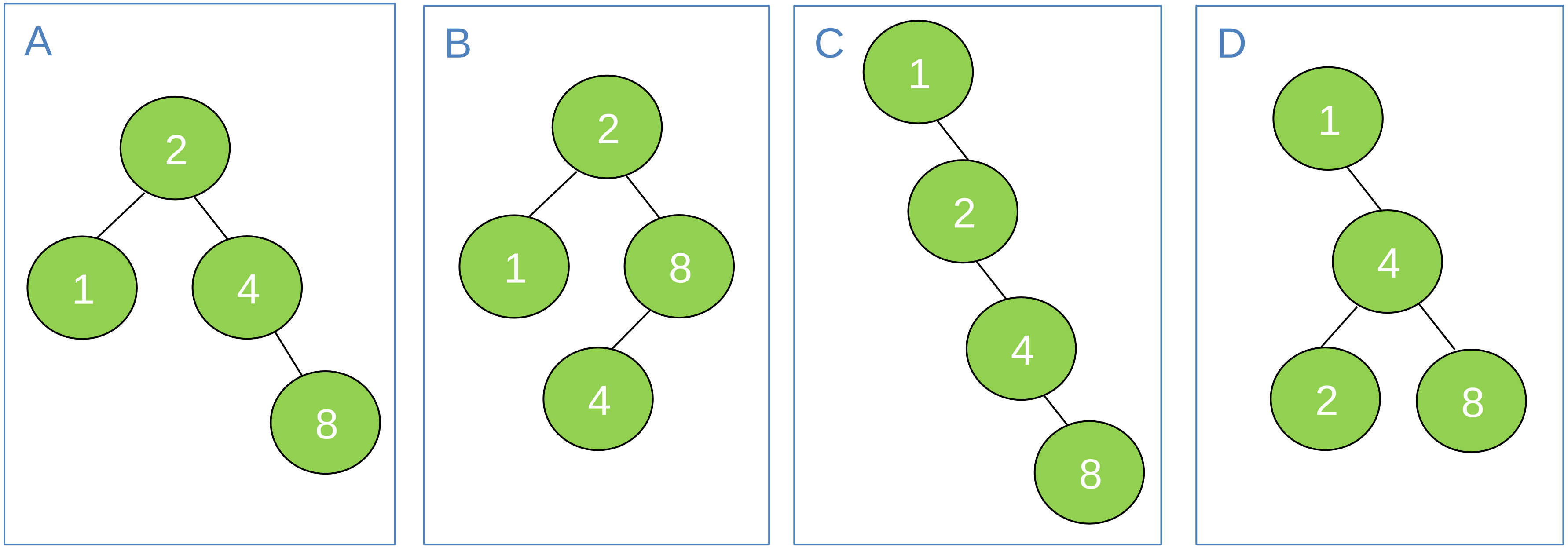
Replace deleted element with smallest right subtree value

Then delete right subtree duplicate (12)

Binary Search Tree Shape

Which of the following Binary Search Trees could be the result of adding elements: 1, 2, 4, and 8 in some order (select all). For valid trees, determine (on your own) an insertion order which would produce that tree?

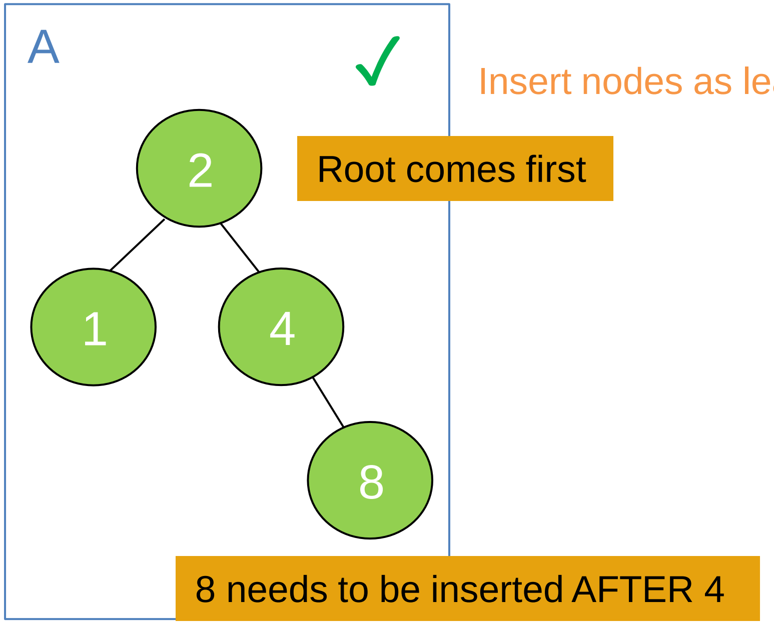
These are all valid binary search trees!



Binary Search Tree Shape (Contd.)

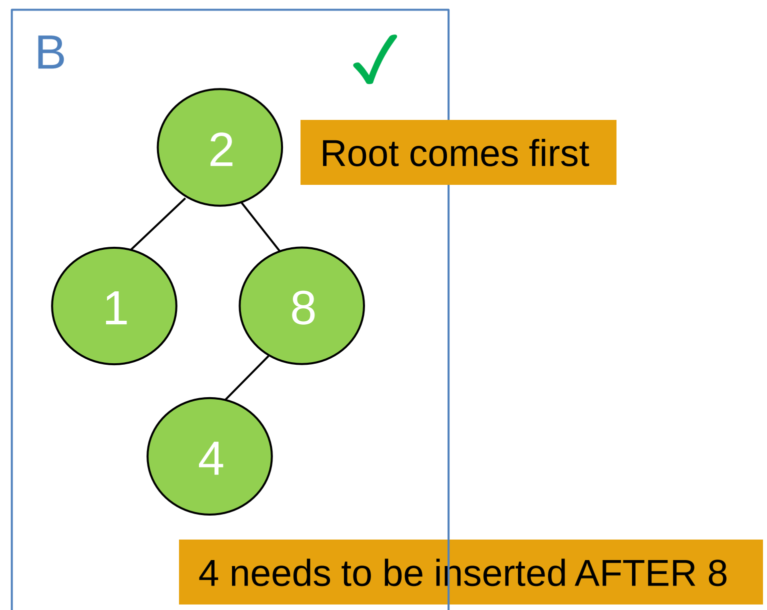
Inserting a node means making it a child of an existing node

Insert nodes as leaves

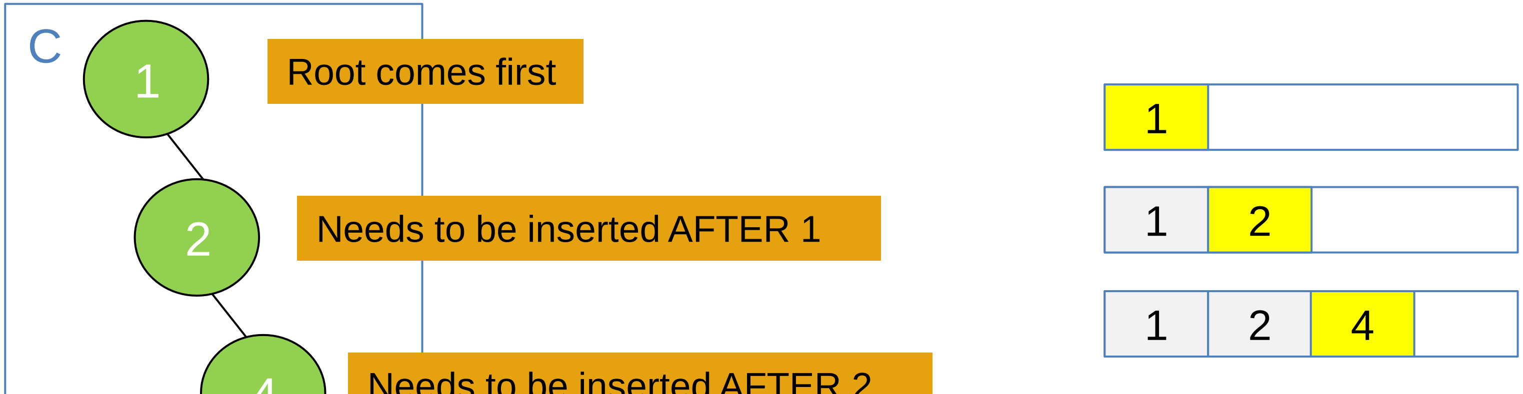


2			
2	4	1	8
2	1	4	8
2	4	8	1

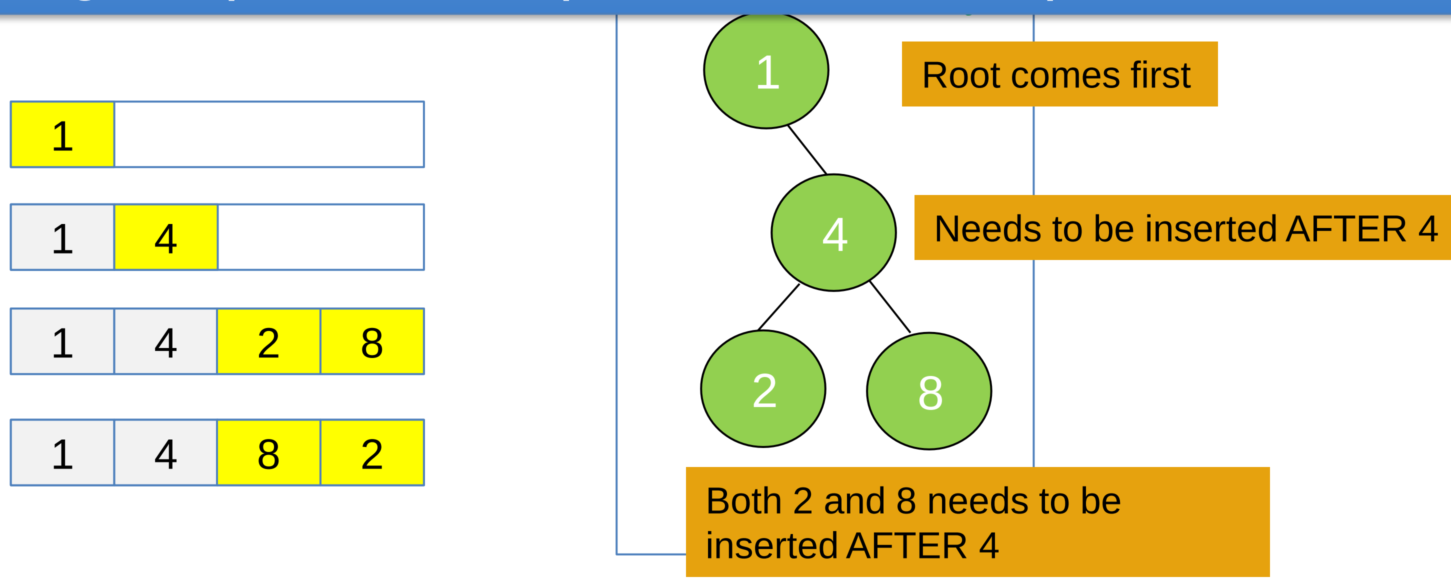
2			
2	8	1	4
2	1	8	4
2	8	4	1



Binary Search Tree Shape (Contd.)



The order in which we put elements into a BST impacts the shape, and what you‘ll see is that the shape of BST will have a huge impact on the performance of operations.

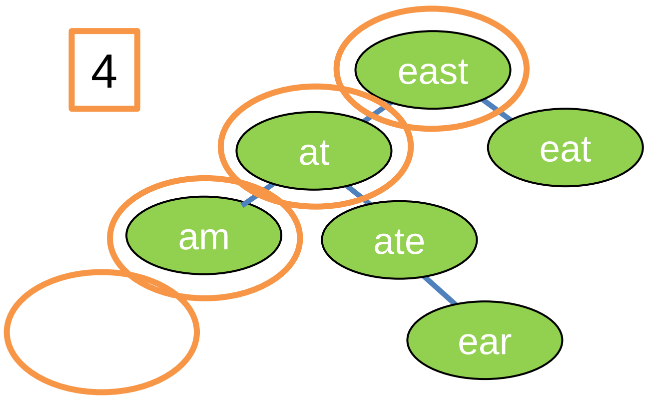


Performance Analysis of BST

Storing a dictionary as a BST

{ am, at, ate, ear, eat, east }

Structure of a BST depends on the order of insertion

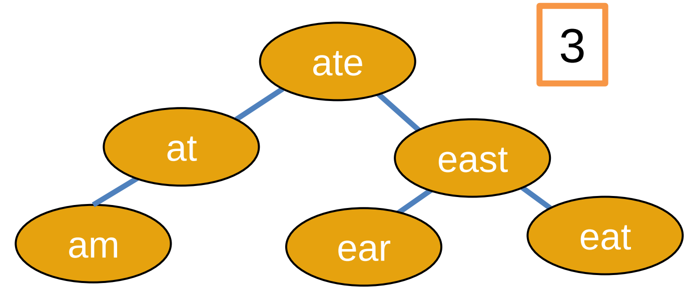


`isWord(east)`

Best case: O(1)

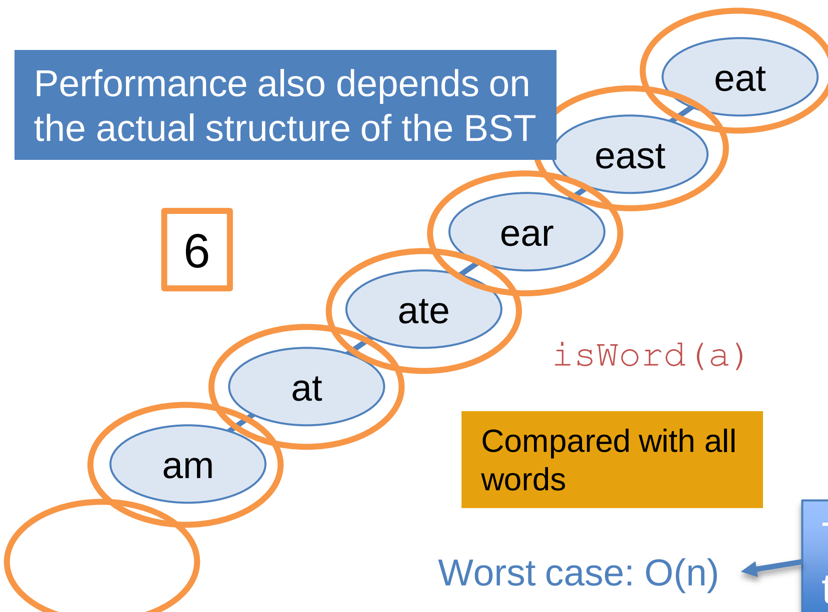
`isWord(a)`

Compared with 3 out of 7 words



How does the performance of isWord relate to input size n?

Performance also depends on the actual structure of the BST



`isWord(a)`

Compared with all words

`isWord(String wordToFind)`

1. Start at root
2. Compare word to current node
    1. If current node is null, return false
    2. If wordToFind is less than word at current node, continue searching in left subtree
    3. If wordToFind is greater than word at current node, continue searching in right subtree
    4. If wordToFind is equal to word at current node, return true

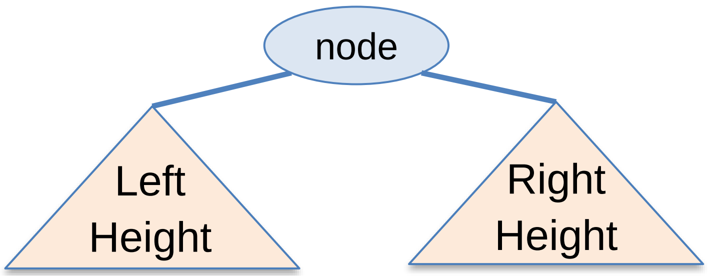
Worst case: O(n)

To optimize the worst case, we can modify the tree to control the max distance until leaf

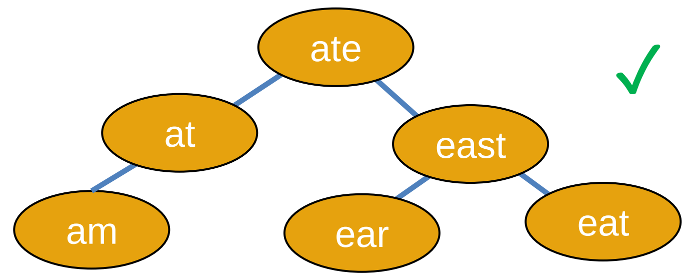
height

Balanced BST

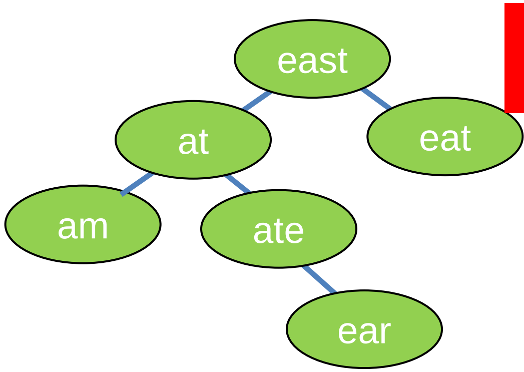
We want to keep the height down as much as we can while still maintaining the same number of nodes.

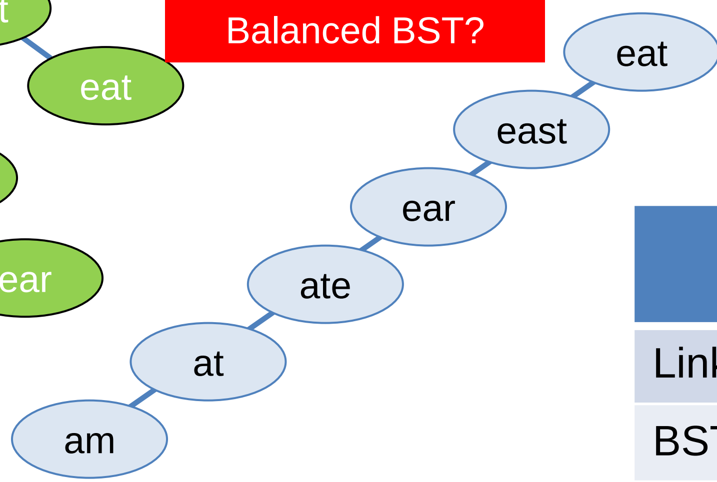


| LeftHeight – RightHeight | <=1



Which is the Balanced BST?





height ≈ log(n)

Especially if insert to BST in order!

	Best case	Average case	Worst case
Linked List	O(1)	O(n)	O(n)
BST	O(1)	O(log n)	O(n)
Balanced BST	O(1)	O(log n)	O(log n)

How to keep balanced? TreeSet and TreeMap in Java API

`isWord(String wordToFind)`

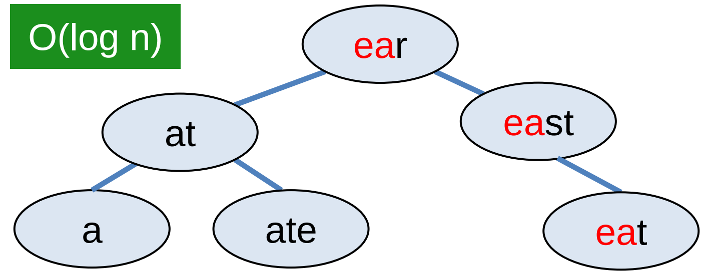
Introduction to Tries

re(TRIE)ve

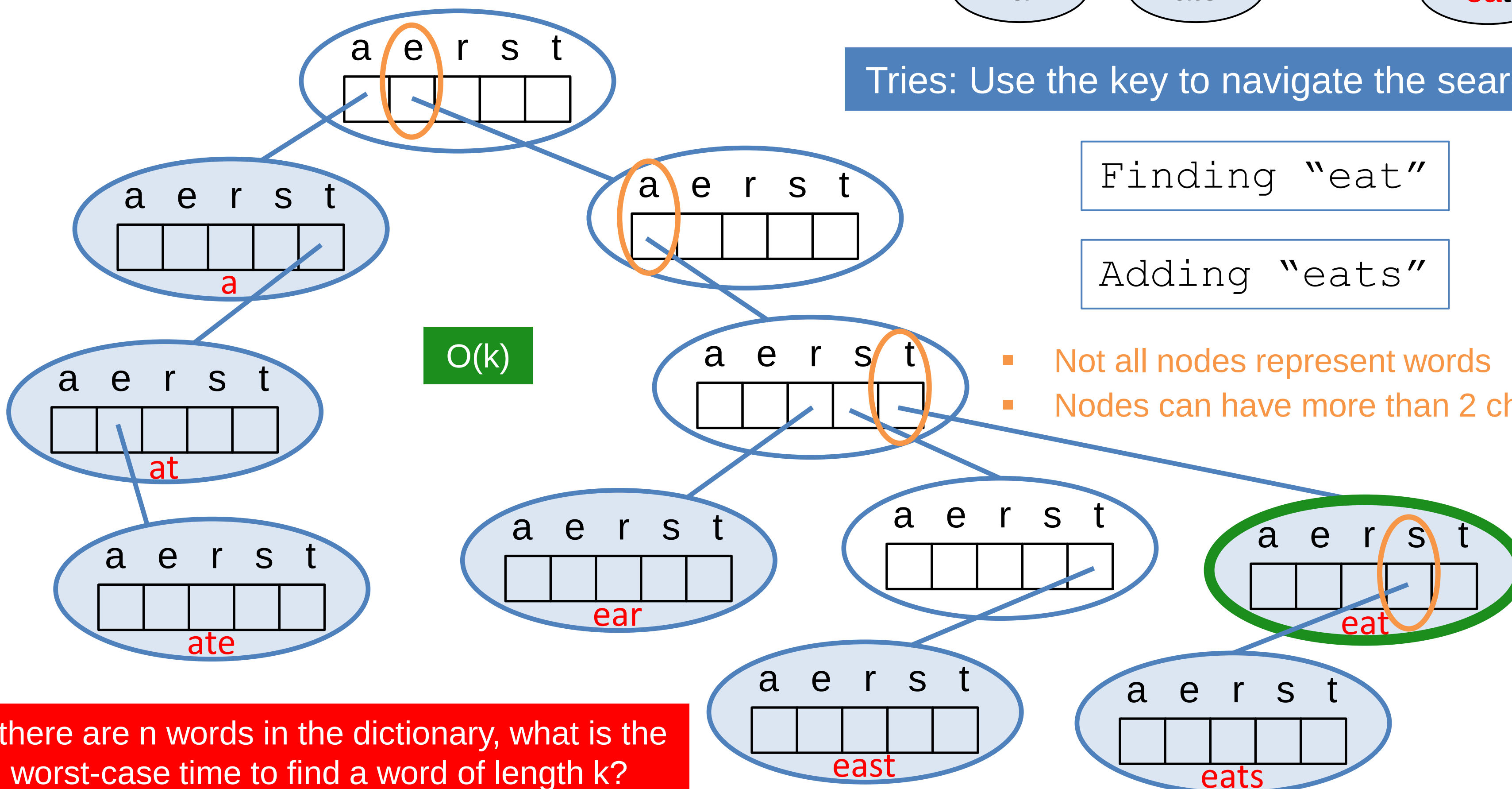
Storing a dictionary as a (balanced) BST

BSTs don't take advantage of shared structure

O(log n)



Tries: Use the key to navigate the search



O(k)

`Finding "eat"`

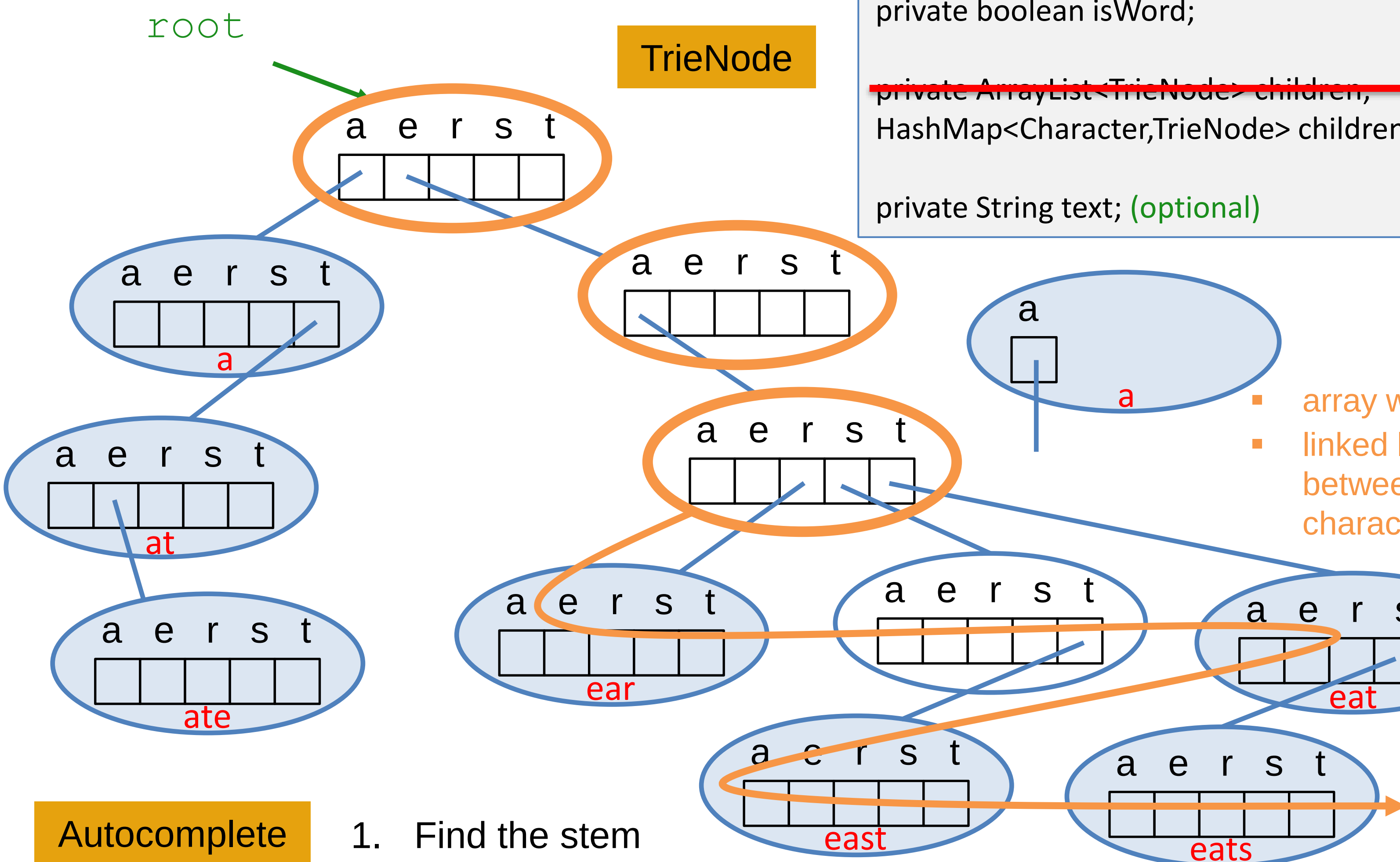
`Adding "eats"`

- Not all nodes represent words
- Nodes can have more than 2 children

If there are n words in the dictionary, what is the worst-case time to find a word of length k?

$\log_2(250000) \approx 18$

Implementing a Trie




```
private boolean isWord;

~~private ArrayList<TrieNode> children;~~
HashMap<Character,TrieNode> children;

private String text; (optional)
```

- array wastes space
- linked list lost association between position and character

Autocomplete

"ea"

1. Find the stem
2. Do a level order traversal from there

from shortest to longest

Additional Resources

- Trees and Binary Search Trees
  - http://www.openbookproject.net/thinkcs/archive/java/english/chap17.htm -- explains trees, how to build and traverse it
  - http://algs4.cs.princeton.edu/32bst/ -- about binary search trees
  - https://www.youtube.com/watch?v=pYT9F8_LFTM -- BST video
- Tries
  - https://www.toptal.com/java/the-trie-a-neglected-data-structure -- explains with solid example
  - https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/ -- explains as well as providing code

Upper Bound of Balanced BST Height

- Let N_h represent the minimum number of nodes that can form a balanced BST of height h. If we know N_{h-1} and N_{h-2}, we can determine N_h. Since this N_h-noded tree must have a height h, the root must have a child that has height h − 1. To minimize the total number of nodes in this tree, we would have this sub-tree contain N_{h-1} nodes.
- By the property of a balanced BST , if one child has height h − 1, the minimum height of the other child is h − 2. By creating a tree with a root whose left sub-tree has N_{h-1} nodes and whose right sub-tree has N_{h-2} nodes, we have constructed the balanced BST of height h with the least nodes possible.
- This tree has a total of $N_{h-1}+N_{h-2}+1$ nodes (N_{h-1} and N_{h-2} coming from the sub-trees at the children of the root, the 1 coming from the root itself). The base cases are $N_1 = 1$ and $N_2 = 2$. From here, we can iteratively construct N_h by using the fact that $N_h = N_{h-1} + N_{h-2} + 1$ that we figured out above. Using this formula, we can then reduce as such:

$$
\begin{aligned}
N_h &= N_{h-1} + N_{h-2} + 1 \\
N_{h-1} &= N_{h-2} + N_{h-3} + 1 \\
N_h &= (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \\
N_h &> 2N_{h-2} \\
N_h &> 2^{\frac{h}{2}} \\
\log N_h &> \log 2^{\frac{h}{2}} \\
2\log N_h &> h \\
h &= O(\log N_h)
\end{aligned}
$$