CS151 Intro to Data Structures

Trees

Outline

- HW comments
- Iterator
- Trees:
  - Overview
  - Binary Search Tree
    - Inserting
    - Searching

Announcements

- HW03 (Stacks & Queues) – due Friday 10/27
  - Must include your own Junit tests
  - 10% of grade
  - Have one file that contains all the Unit tests

- Lab 04, 05, 06 due Friday 10/27
  - Lab 06 (last week's lab) no checkoff, due on Gradescope

Homework Comments

HW00 grades/feedback returned

HW01 grades/feedback will be returned later today

HW02 will be returned this week

Homework Comments

Code must compile!

Not compiling is obvious disregard for your work

Code must be efficient

When querying, don’t re-load the entire dataset each time

Don’t resize if we don’t have to

- If we know how many datapoints when are going to have, create one array/Arraylist of that size

Class design:

Don’t just use Strings. Use ints/doubles when appropriate

Homework Comments

Make sure your code passes batch testing using the given `in.txt` and `out.txt`

To make sure your code passes, run

`“java Driver … < in.txt | diff out.txt –”`

Homework - No Hardcoding

- `int`/`double`/`String` literals in code
- Anything that has reason to change later should be a constant variable
- `public static final`

```
//in LookupZip
public static final int ZIP = 0;
public static final int TOWN = 1;
public static final int STATE = 2;
…
//zipData = line.split(…);
Place p = new Place(zipData[ZIP],
zipData[TOWN], zipData[STATE]);


//in Main
public static final String quit =
"00000";
…
if (userInput.equals(quit))
```

Homework Comments

Don’t do while(true)

Bad programming

Use a Boolean variable or operator

Why use Iterators?

Encapsulate traversal

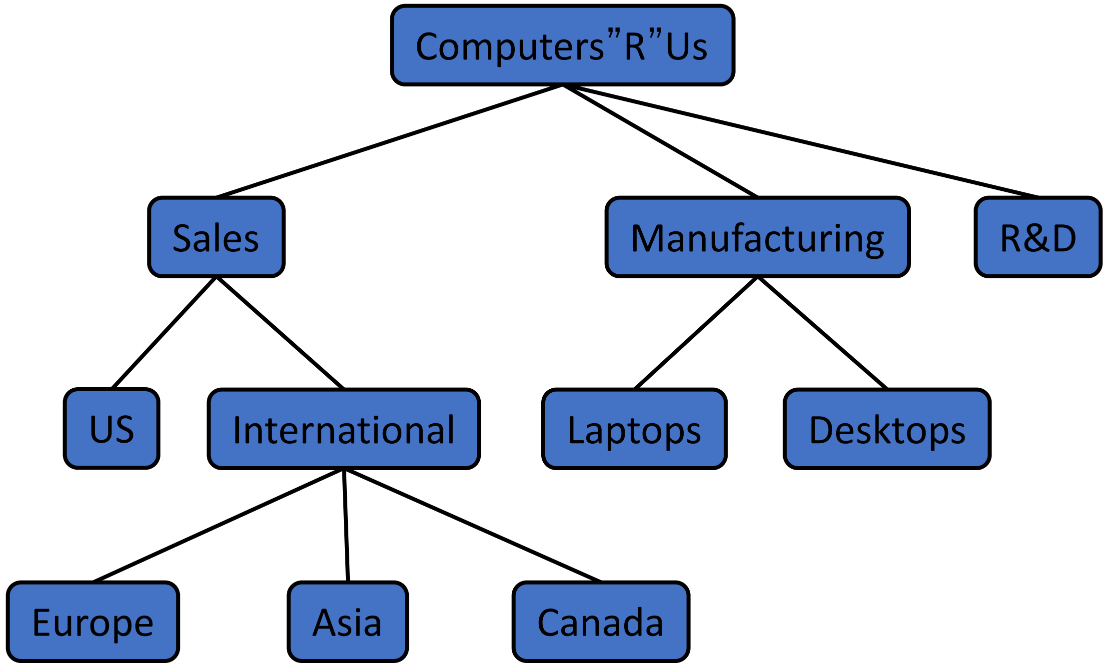
Container independence

- allows traversal without knowledge of underlying data structure implementation, i.e. `.length` or `.size()` or linked list
- allows switching out the underlying data structure while causing the least amount of code change elsewhere

Tree

A tree is an abstract model of a hierarchical structure

Nodes have a parent-child relation



Terminology

root: no parent

A

external node/leaf: no children

E, I, J, K, G, H, D

internal node: - node with at least one child

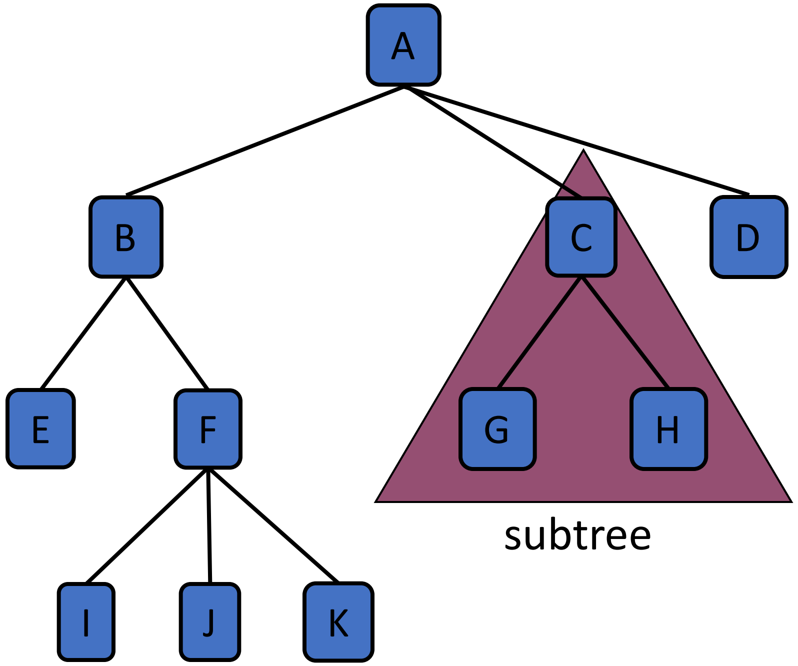
A, B, C, F

ancestor/descendent

depth - # of ancestors

Height - max depth

- **Subtree**: tree consisting of a node and its descendants

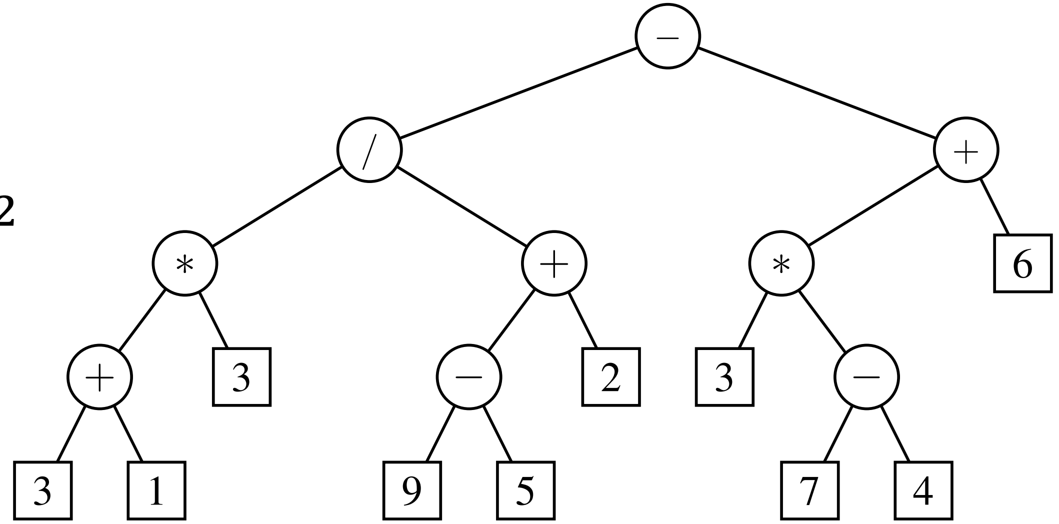


Binary Tree

An (ordered) tree with every node having at most two children – left and right

Recursive definition:

- base case: empty tree
- recursion: root with two subtrees $T_1 \cdot T_2$



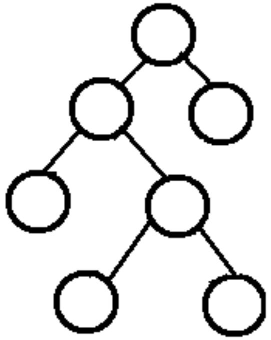
Types of Binary Trees

A binary tree is **full** (or proper) if each node has zero or two children

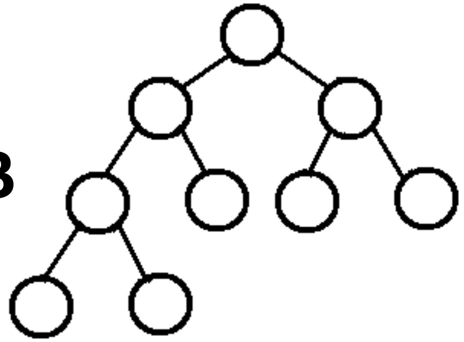
A binary tree is **complete** if every level (except possibly the last) is filled

If a complete binary tree is filled at every level, it is **perfect**

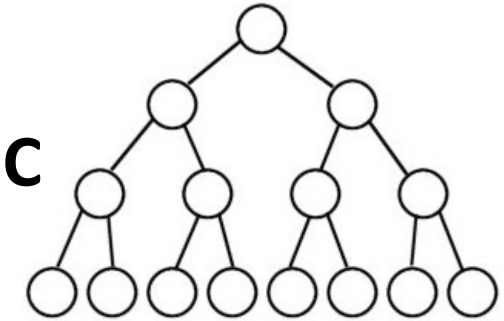
A



B



C

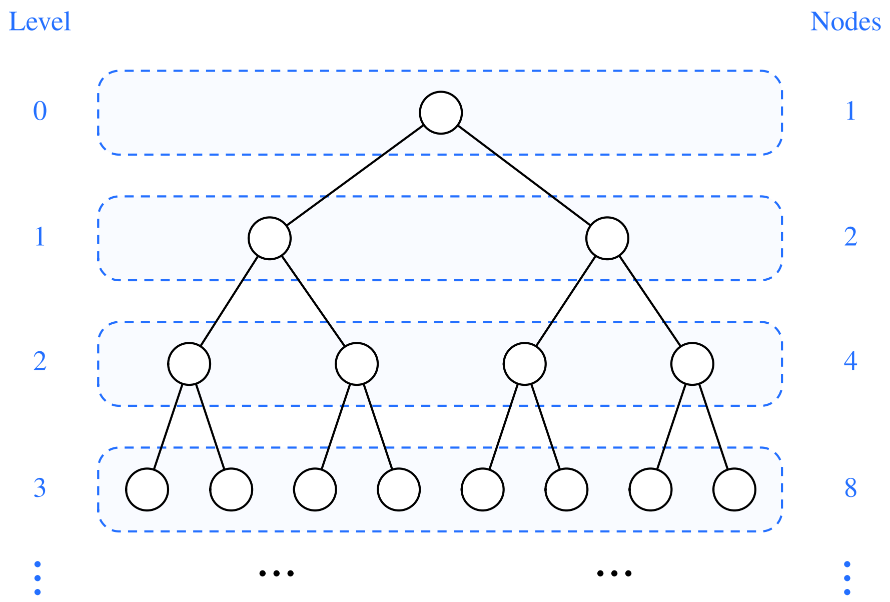


Binary Tree Properties

Let n denote the number of nodes and h the height of a binary tree

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log(n + 1) - 1 \leq h \leq n - 1$

Height of a complete binary tree is $O(logn)$ of the max number of nodes



Interface

```
public interface BinaryTree<E extends
Comparable<E>> {
  E getRootElement();
  int size();
  boolean isEmpty();
  void insert(E element);
  boolean contains(E element);

  //…
}
```

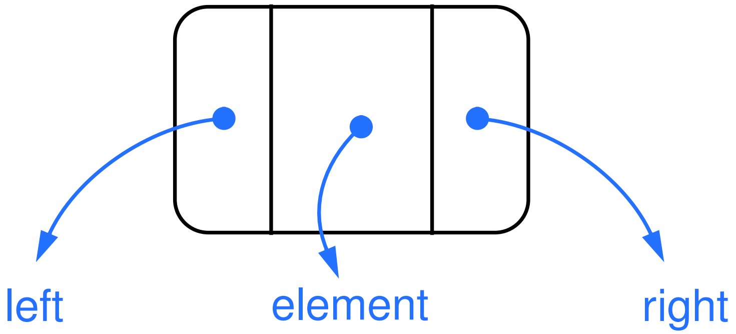
Implementation

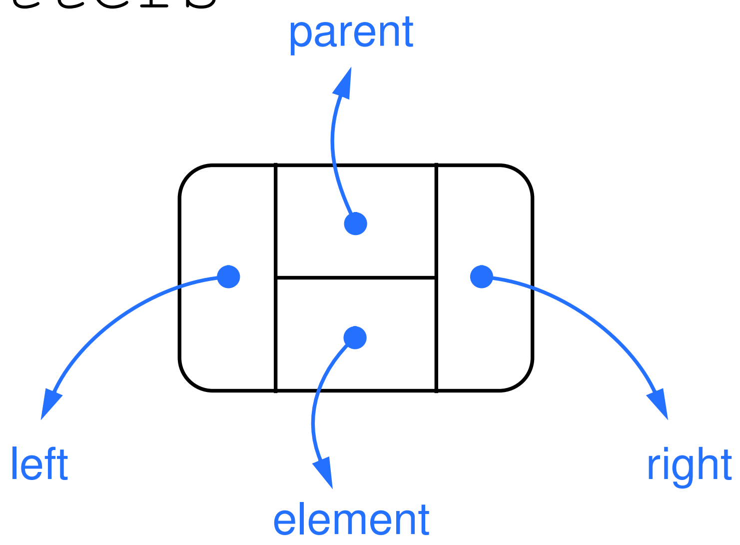
```
public class Node<E> {
  private E element;
  …

  …
}
```

Implementation

```
public class Node<E> {
  private E element;
  private Node<E> left;
  private Node<E> right;
  //constructors, getters, setters
  public boolean isLeaf(){
    //?
  }
}
```



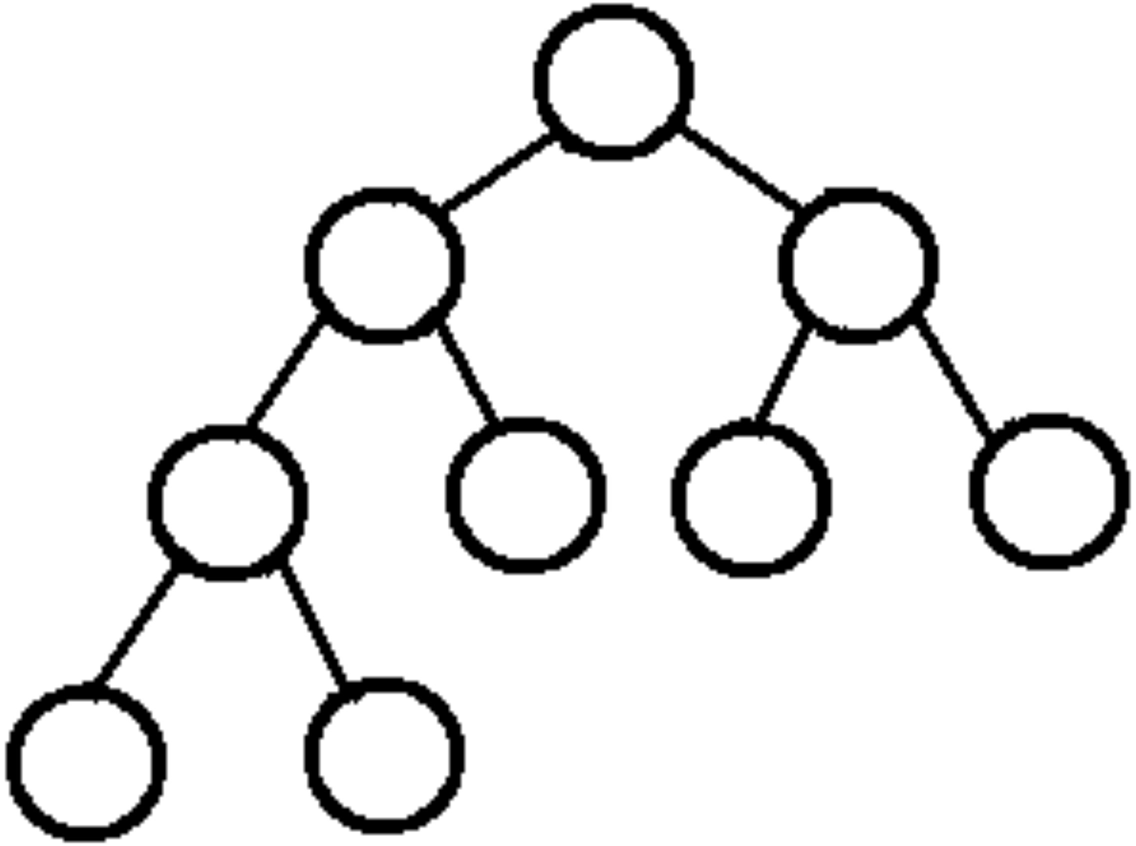


Class

```
public class LinkedBinaryTree<E extends
Comparable<E>> implements BinaryTree<E> {
    // what instance variables?
    // nested Node class


}
```

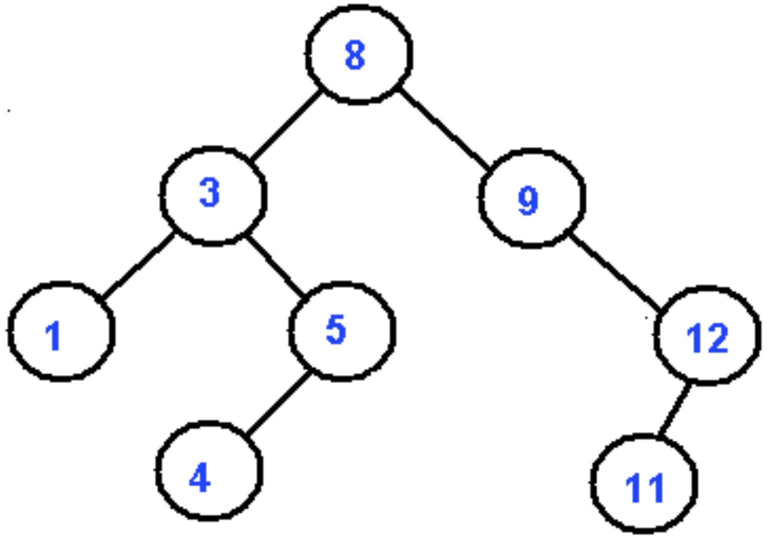
Insertion

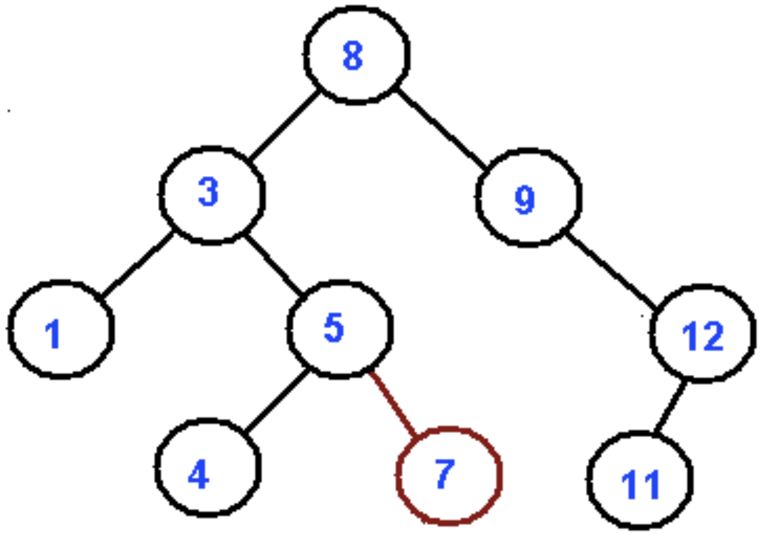


Insertion

Binary search trees are ordered

- Comparable elements
- Smaller to the left, bigger to the right





Draw a Binary Search Tree

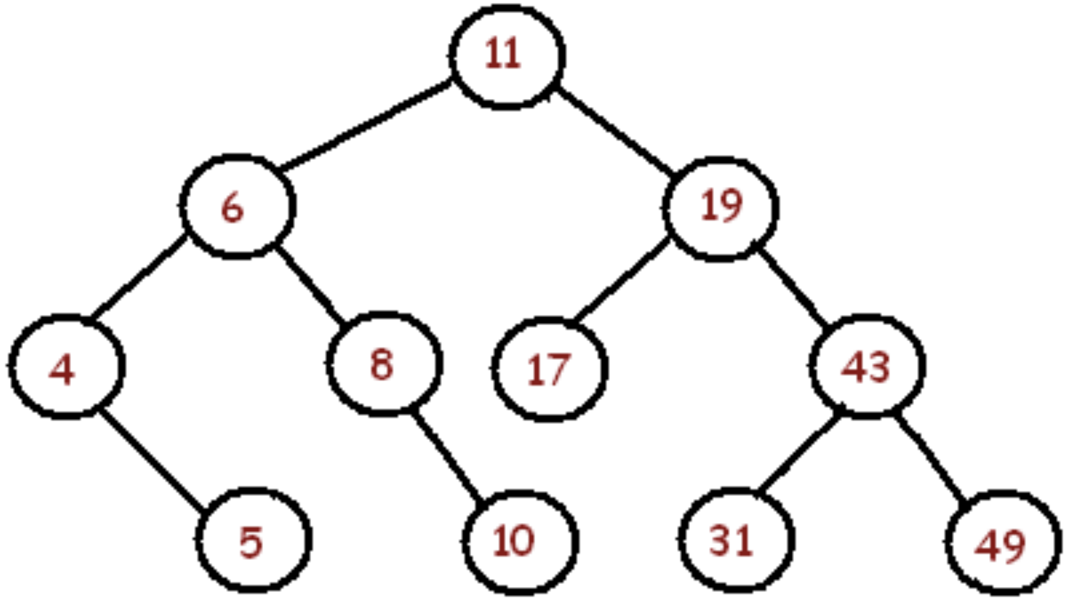
- 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Skip slide

Skip slide

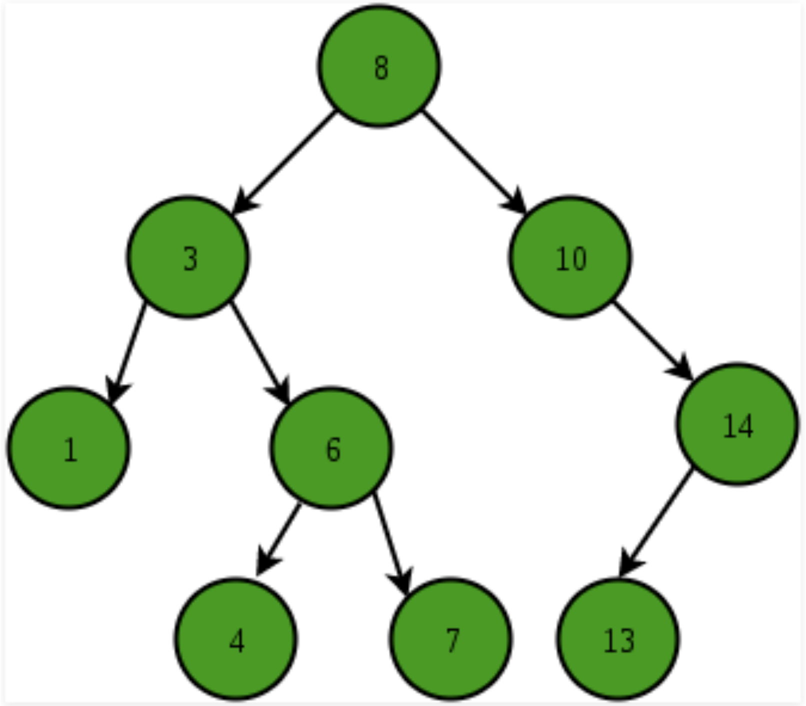
Draw a Binary Search Tree

- 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

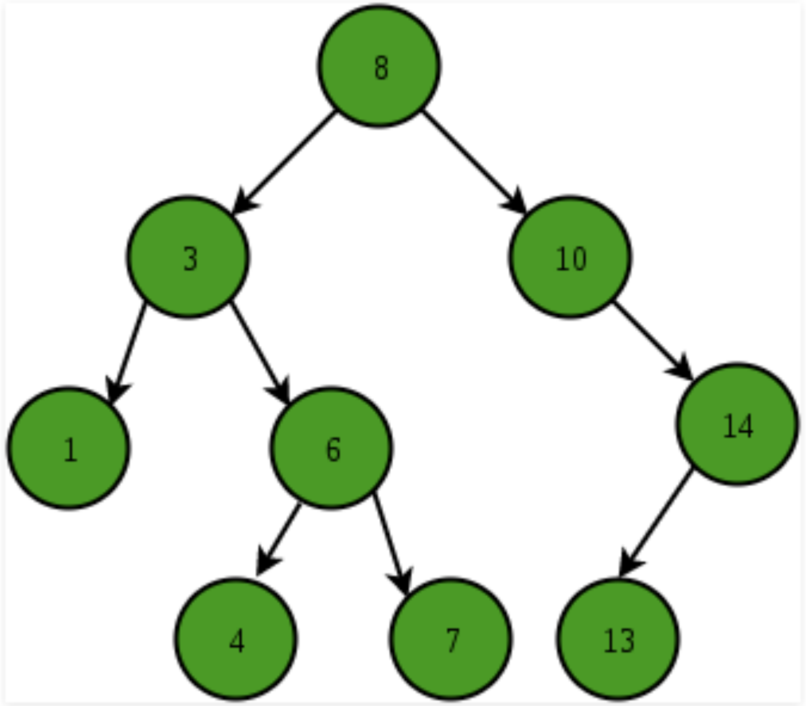


Search

- `boolean contains(E element);`
- returns `true` if found in the tree, `false` otherwise



Algorithm



Recursive solution:

compare with root of current subtree:

- root is empty – return false
- root == element – return true (we found the element ☺)
- root < element
  - recurse on right child
- root > element
  - recurse on left child

Pseudo Code

```
containsRec(root, key):
  if root == null:
    return false
  if root.key == key:
    return true
  if root.key > key:
    return containsRec(root.left, key)
  else
    return containsRec(root.right, key)
```

Recursive Helper Method

The signature of `contains` doesn't allow any `Node` references and isn't inducive to recursion

- `boolean contains(E element);`

```
boolean containsRec(Node<E> root, E element);
```

insert

```
void insert(E element);
```

new node is always inserted as a leaf

inserts to

- left subtree if element is smaller than subtree root
- right subtree if larger

What are the cases?

- root is empty, root is a leaf, root has one child, root has two children

Pseudo Code

```
insertRec(root, key):
  if root == null:
    return new Node(key)
  if root.key > key:
    root.left =
    insertRec(root.left, key)
  else
    root.right =
    insertRec(root.right, key)
  return root
```

Recursive Helper Method

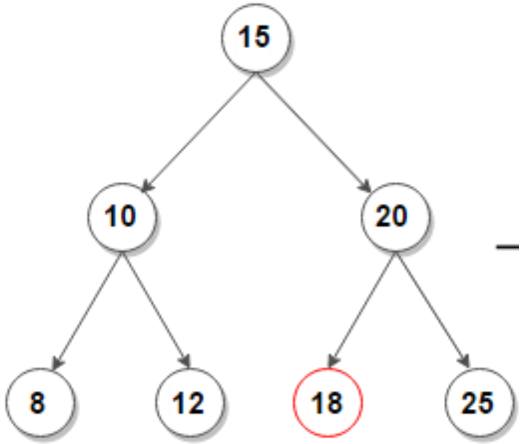
- `Node<E> insertRec(Node<E> root, E element);`

Remove

- `boolean remove(E element);`
- returns true if element existed and was removed and false otherwise
- Cases
    - element not in tree
    - element is a leaf
    - element has one child
    - element has two children

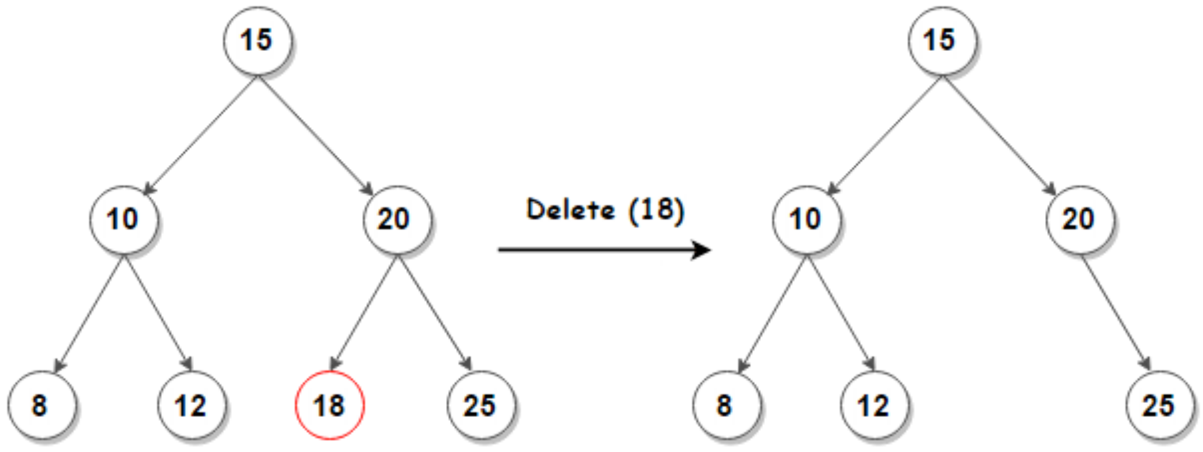
Leaf

- Just delete



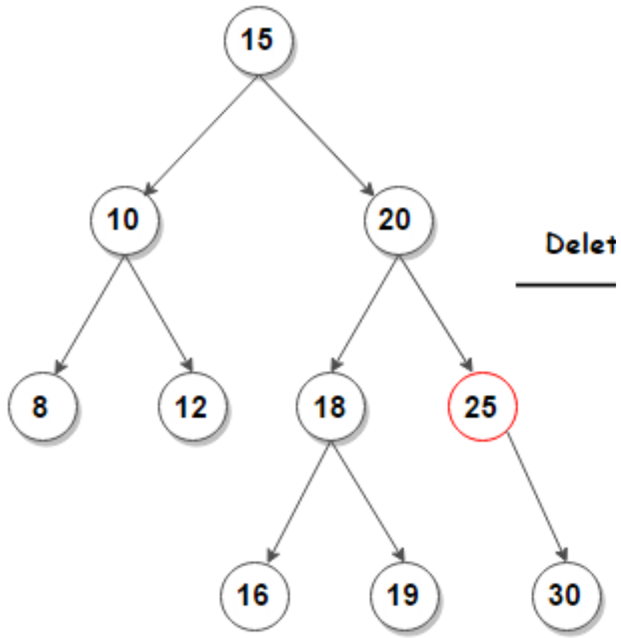
Leaf

- Just delete



One child

- Replace with child – skip over like in linked list



One child

- Replace with child – skip over like in linked list