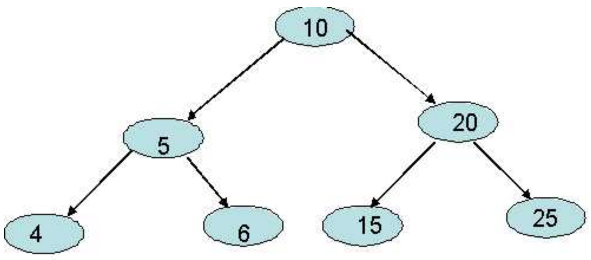
Binary Tree Definition

What is a Binary Tree?

A binary tree is a type of linked list where each node has two links to other nodes instead of just one. The model for the data structure is that the top-level node is called the “root” and that all nodes on the left side of the root have key values ***less*** than the root and all nodes on the right side of the root have key values that are ***greater*** than the root.

The following is a graphical depiction of the Binary Tree model:



A simple binary tree

Whenever a node points to another node, the pointing node is called the “parent” of the node it points to. The node being pointed to is called the “child” of the node pointing to it.

In the graphical example, the top node with the “10” key is the root. Both “5” and “20” are children of the root node. Therefore, “10” is considered the parent of both “5” and “20”. The relationships trickle down the binary tree.

Therefore, “5” is considered the parent of “4” and “6” and etc.

Purpose of a Binary Tree

Purpose of a Binary Tree

What purpose do Binary Trees serve?

The idea is that, if we have a perfectly balanced tree (where the difference between nodes in any given left tree and any given right tree differ by no more than 1), we can find any node in the tree in no worse than O(log n) time (logarithmic). This is an improvement over linear search on an unsorted array which is O(n) time in the worst case. This type of search is considered a type of binary search since it is on a sorted data structure and we are halving the amount of remaining values to look through with each iteration or recursive call.

Another benefit of this data structure is that it allows us to maintain a type of sort relatively easy. In an array, we must create a new array of larger size and copy over the old array being careful to place the new value in the proper spot in the new array. This is an O(n) time (or linear time) operation.

With binary trees, we should be able to insert in O(log n) time worst case assuming a balanced tree structure.

What if our tree isn't balanced?

Although the average case (assuming random key insertions) is favorable, when we need an absolute performance guarantee, a basic Binary Tree will not get us there. The worst case for insertion in a binary tree is still O(n) time (assume inserting values: 0, 1, 2, 3, 4, 5; for example). If we had this kind of lopsided tree, we would also get bad performance from the search portion (O(n) time worst case).

Because of this, there are different strategies for dealing with the issue of balancing a tree to improve performance in the average or worst case. Algorithms included in your text include: 2-3 trees and Red-Black trees. Read up about those in the text if you are interested.

Common Operations in a Binary Tree

What operations should a Binary Tree data structure support?

At the minimum, a class should support:

- insert – Put new nodes into the tree in the proper order
- search – Find whether the key exists in the tree (boolean) or return the node, if found (Node)
- delete – Remove the node from the tree based on key, if found
- size – Report how many nodes are currently in the tree
- clear – Reset the tree to empty status
- ^toString – Return a string that can be displayed to show current keys in the tree nodes

^ For this, we can choose from different ways to display the keys based on how you want to view the tree. In our “from scratch” example, we will use the “inorder” style which traverses the tree from left nodes first and then right nodes. This makes the data appear in order of smallest to largest.

Other displaying options are: preorder, post-order, and breadth-first (also called “level order”).

Creating and Using Binary Trees using the Java API

How do we use Binary Trees in Java API?

Java does not have a built-in equivalent to a basic binary tree class. They have a “TreeSet” class that can be used for advanced TreeMaps and routines like “Red-Black Trees.”

Guess we have to roll our own this time…

Creating Binary Trees from Scratch

How can we write a basic Binary Tree class from scratch?

The following code will implement the minimum operations we talked about previously. This is the code in Java:

```java
package Main;

import java.util.LinkedList;

import java.util.Queue;

public class BST{

      // Fields

      private Node root; // Top-level node in tree

      private int n; // Number of nodes in tree

      private Queue<Integer> q; // Used for the inorder toString method

      public class Node{

            int key;

            Node left;

            Node right;

            public Node(int key) {

                  this.key = key;

                  left = right = null;

            }

      }

      // Constructor

      public BST(){

            root = null;

            n = 0;

      }

      // Methods

      public void insert(int key){

            root = insertHelper(root, key);

      }

      private Node insertHelper(Node root, int key){

            if(root == null){

                  root = new Node(key);
            n++;

            return root;

        }

        if(key < root.key){

            root.left = insertHelper(root.left, key);

        }else if(key > root.key){

            root.right = insertHelper(root.right, key);

        }

        return root;

    }

    public boolean search(int key){

        return searchHelper(root, key);

    }

    private boolean searchHelper(Node root, int key){

        if(root == null)

            return false;

        if(root.key == key)

            return true;

        if(key < root.key)

            return searchHelper(root.left, key);

        return searchHelper(root.right, key);

    }

    public void delete(int key){

        root = deleteHelper(root, key);

    }

    private Node deleteHelper(Node root, int key){

        if(root == null)

            return null;

        if(key < root.key){

            root.left = deleteHelper(root.left, key);

        }else if(key > root.key){

            root.right = deleteHelper(root.right, key);

        }else{

            if(root.left == null){
                n--;

                return root.right;

            }

            else if(root.right == null){

                n--;

                return root.left;

            }

            // Here is where we decide how to delete...

            root.key = minNodeKey(root.right);

            root.right = deleteHelper(root.right, root.key);

        }

        return root;

    }

    private int minNodeKey(Node root){

        int min = root.key;

        while (root.left != null){

            min = root.left.key;

            root = root.left;

        }

        return min;

    }

    public int size(){

        return n;

    }

    public void clear(){

        root = null;

        n = 0;

    }

    /* This will return the keys of the BST in an inorder traversal */

    public String toString(){

        String ret = "[";

        q = new LinkedList<Integer>();

        inorder(root);

        while(!q.isEmpty()){
                ret += q.remove() + ",";

            }

            ret = ret.substring(0, ret.length()-1);

            ret +="]";

            return ret;

        }

        private void inorder(Node node){

            if(node == null) return;

            inorder(node.left);

            q.add(node.key);

            inorder(node.right);

        }

    }
```

Using our Binary Tree class in practice

As a tester of how we can use this class, I included the following Java code:

```java
package Main;

public class BSTTest{

      public static void main(String[] args){

            // Tester array to insert into BST

            int[] a = {5, 3, 7, 2, 4, 6, 8};

            BST bst = new BST();

            for(int i = 0; i < a.length; i++)

                  bst.insert(a[i]); // Insert values here...

            System.out.println(bst.size()); // Print the size

            for(int i = 0; i < 10; i++){

                  System.out.println("Searching for " + i + " and result is: " + bst.search(i));

            }

            System.out.println(bst.toString()); // Display the BST using inorder traversal

      }

}
```