The Tree Data Structure

Tree Definitions

Mathematically speaking, we may define a tree $T = (V, E)$ as a pair of sets, where V is the **vertex** (or **node**) set, and E is the **edge** set. Moreover, elements of E have the form (u, v) where $u \in V$ is the **parent** of $v \in V$, and v is the **child** of u. The **root** of T is the unique vertex $r \in V$ that does not have a parent. In other words, there are no edges of the form (u, r). The following definitions further describe properties of $T = (V, E)$.

Tree Size the number of vertices of T

Leaf any vertex v for which children$(v) = \emptyset$.

Internal Vertex any vertex v for which children$(v) \neq \emptyset$

parent(v) the unique vertex u for which $(u, v) \in E$, assuming v is not the root.

children(u) the set of vertices v for which $(u, v) \in E$

siblings(v) the set of vertices who have the same parent as v

ancestors(v) the parent of v along with the ancestors of the parent

dedcendants(u) the children of u along with their descendants

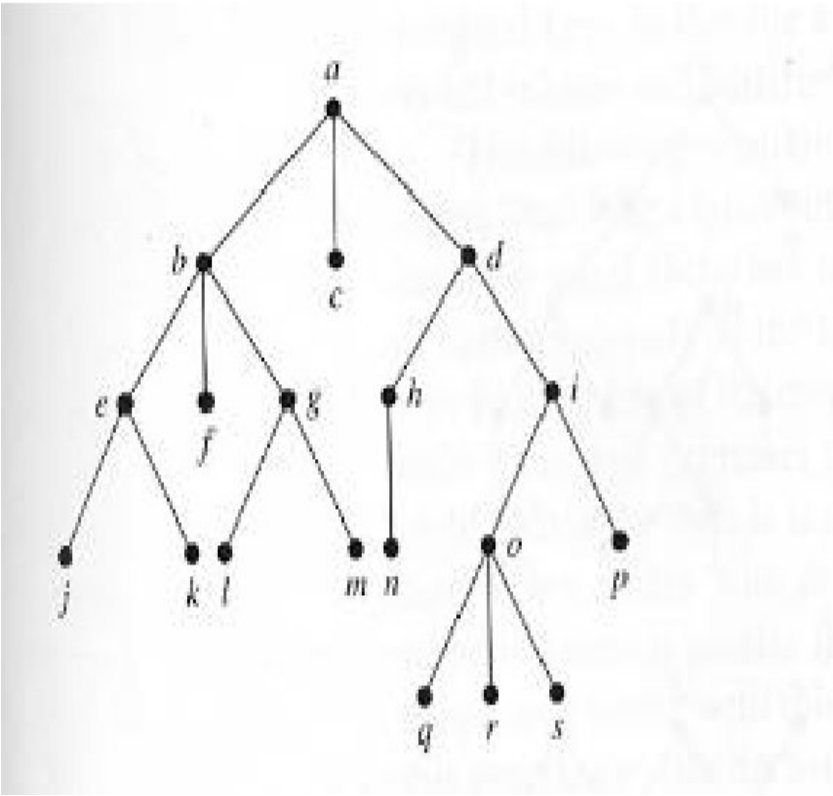
depth(v) the length of the path (in terms of number of edges traversed) from the root to v

height(v) the length of the longest path from v to a leaf of T

m**-ary** the maximum number of children that a parent may possess. For example, in a binary tree, each parent is allowed at most two children.

Full Tree one in which every parent has the maximum number of children

Perfect Tree a full tree for which every leaf has the same depth


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s

Common Tree Data Structures

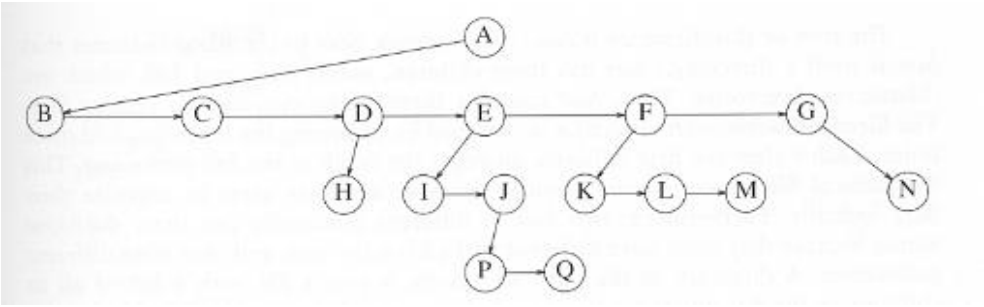
```
Node
{
    Node left; //the left child of this node
    Node right; //the right child of this node
}
```

In case the nodes of a tree are allowed to have any number of children, the `left` and `right` attributes may be replaced with a single list attribute.

```
Node
{
    List children; //a list of nodes that are the children of this node
}
```

It is sometimes necessary to have a `parent` attribute that references the parent of the node.

Example 1. Below are the `children` lists for nodes of a tree. Draw the tree.



Example 2. Assuming the binary `Node` data structure above, recursively implement the function

```
int size(Node n)
```

that returns the size of the tree rooted at n.

Example 3. Assuming the binary `Node` data structure above, recursively implement the function

```
int height(Node n)
```

that returns the height of the tree rooted at n.

Binary Search Trees

A **binary search tree** is a binary tree for which each node has an additional integer (or some other comparable) attribute, which we'll refer to as `value`, and for which the value of a node is greater than (respectively, less than) the values of all its left (respectively, right) descendants. In what follows, we assume the binary In what follows we assume the `Node` data structure defined above, but assume the additional `value` attribute.

Example 4. A binary search tree is to be constructed by adding integer nodes (in the order in which they are shown) $n_1, n_2, \ldots, n_7$ to an initially empty tree. If the integer values of the nodes are respectively $5, 8, 2, 9, 6, 10, 1$, what will the tree look like? Label each node with its value.

Essential operations for a binary search tree.

- **insert():** insert a node into the tree
- **find():** find a node with a given value in the tree
- **find_min():** find the node with the least value
- **find_max():** find the node with the greatest value
- **delete():** delete a node from the tree

```
//Assume: there is no node n in the tree for which n.value = value
//Insert a value into the tree
void insert(Node n, int value)
{
    if(value < n.value)
    {
        //insert into left subtree
        if(n.left == null)
        {
            n.left = new Node(value,null,null);
            return;
        }
        else
            insert(n.left, value)
    }

    //insert into right subtree
    if(n.right == null)
    {
        n.right = new Node(value,null,null);
        return;
    }

    insert(n.right,value);
}
```

Example 5. Insert a node with integer value 7 into the tree constructed during **Example 4.**

```
//find and return the node n for which n.value = value
Node find(Node n, int value)
{
    if(n == null)
        return null;

    int diff = n.value - value;

    if(diff == 0)
        return n;

    if(diff > 0)
        return find(n.left,value);

    //must be in right subtree
    return find(n.right,value);
}
```

Example 6. Trace through the above `find` code using **Example 4**, along with values 7, and 12.

```
//find and return the least element
Node find_min(Node n)
{
    if(n.left == null)
        return n;

    return find_min(n.left);
}
```

Deleting a Node from a Binary Search Tree

Let $n \in T$ be the node to be deleted.

- **case 1:** n has no children. Then directly remove n.
- **case 2:** n has one child. Upon removing n make $child(n)$ a child of $parent(n)$.
- **case 3:** n has two children. Replace n with the least right descendant $n^{'}$ of n. $n^{'}$ will have at most one child, so either case 1 or case 2 applies to its relocation.

Example 7. Demonstrate all three cases using the tree from **Example 4**.

Complexity analysis of bst operations

- **worst case:** occurs when all n data exist on one branch of the tree. In this case, all operations will equal $O(n)$.
- **best case:** occurs when the bst is perfect. In this case the tree will have depth $\log n$ and the complexity of all operations equal $O(\log n)$.
- **average case:** the average-case complexity depends on the average depth of a node in a randomly constructed tree.

Theorem. The average depth of a node in a randomly constructed binary search tree is $O(\log n)$.

Proof: Given a tree T, the sum of the depths of all the nodes of the tree is called the *internal path length*. Let $D(n)$ denote the internal path length of a binary tree having n nodes. Then $D(1) = 0$ and, for $n > 1$,

$$D(n) = D(i) + D(n - i - 1) + n - 1,$$

assuming that the tree has i nodes in its left subtree, and $n - i - 1$ nodes in its right subtree. The $n - 1$ term comes from the fact that $D(i)$ and $D(n - i - 1)$ gives a measurement of the internal path lengths of the left and right subtrees from the perspective that both roots start at depth 0. But since they are subtrees of one level down, each of the $n - 1$ nodes will have an extra unit of depth. Hence, we must add $n - 1$ to the recurrence relation. Now, in the case of a binary search tree for which the root node is randomly selected from a list of n choices, each with equal probability, we see that the average internal path length for a randomly constructed tree is

$$\frac{1}{n}[(D(0) + D(n-1) + (n-1)) + (D(1) + D(n-2) + (n-1)) + \cdots + (D(n-1) + D(0) + (n-1))] =$$

$$\frac{2}{n}\sum_{i=0}^{n-1} D(i) + (n-1).$$

We now show that this recurrence relation yields $D(n) = \mathrm{O(n \log n)}$. We prove this by induction.

To begin, we certainly can find a constant $C > 0$ such that $D(1) = 0 \leq C \cdot 1 \log 1 = 0$. Any C will work.

Now assume that there is a constant $C > 0$ such that, for all $m < n$, $D(m) \leq Cm \log m$. We now show that $D(n) \leq Cn \log n$, by choosing the right condition on C. To start,

$$D(n) = \frac{2}{n}\sum_{i=0}^{n-1} D(i) + (n-1) \leq$$

(by the inductive assumption)

$$\frac{2}{n}\sum_{i=0}^{n-1} Ci\log i + (n-1) \leq$$

$$\frac{2}{n}\int_1^n Cx\log x dx + (n-1) =$$

$$\frac{2}{n}(\frac{Cn^2}{2}\log n - \frac{1}{2\ln 2}\int_1^n x dx) + (n-1) =$$

$$Cn\log n - \frac{C(n^2-1)}{2n\ln 2} + (n-1) \leq Cn\log n,$$

provided that $C > 2\ln 2 = \ln 4$, and n is sufficiently large.

Therefore, the average internal path length is $D(n) = \mathrm{O(n\log n)}$. Dividing this quanity by n yields the average depth of a node in a randomly generated binary search tree, which in this case is $\mathrm{O(\log n)}$. Note: this same analysis can be used to analyze the average-case running time of Quicksort when the median is chosen at random.

Exercises.

1. Prove that, when a binary tree with n nodes is implemented by using links to the left and right child (as was done in the `Node` structure), then there will be a total of $n + 1$ null links.

2. Prove that the maximum number of nodes in a binary tree with height h is $2^{h+1} - 1$.

3. A full node for a binary tree is one that has two children. Prove that the number of full nodes plus one equals the number of leaves of a binary tree.

4. Using the `Node` class described in lecture, implement the function

   ```
   Boolean is_balanced(Node n)
   ```

   Hint: a binary tree is balanced if the heights of its left and right subtrees are no more than 1 apart in absolute value, and both its left and right subtrees are themselves balanced. Note: an empty tree is assumed to have a height of -1.

5. Using the `Node` class described in lecture, and assuming an additional integer `value` attribute, implement the function

   ```
   Boolean is_bst(Node n)
   ```

   which returns `true` iff the tree rooted at n is a binary search tree.

6. Using the `Node` class described in lecture, and assuming an additional integer `value` attribute, implement the function

   ```
   void print(Node n)
   ```

   which has the effect of printing all the integer values stored in the tree rooted at n.

7. Suppose a binary search tree is to hold keys 1,4,5,10,16,17,21. Draw possible binary search trees for these keys, and having heights 2,3,4,5, and 6. For each tree, compute its internal path length. Hint: the height-2 tree should be a perfect tree.

8. Write a method called `print()` for the BinarySearchTree class presented in lecture, and which prints all the objects of the tree in sorted order. You may assume that each object has a `toString()` method that may be called.

9. Prove that it takes $\Omega(n \log n)$ steps in the best case to build a binary search tree having n keys. Show work and explain.

10. Suppose we have a bst that stores keys between 1 and 1000. If we perform a find on key 363, then which of the following sequences could *not* be the sequence of nodes examined when performing using the `find()` method provided in lecture.
    a) 2,252,401,398,330,344,397
    b) 924,220,911,244,898,258,362,363
    c) 925,202,911,240,912,245,363
    d) 2,399,387,219,266,382,381,278,363
    e) 935,278,347,621,299,392,358,363

11. Suppose a bst is constructed by repeatedly inserting distinct keys into the tree. Argue that the number of nodes examined when searching for a key is equal to one more than the number examined when inserting that key.

12. Prove or disprove: deleting keys x and y from a bst is commutative. In other words, it does not matter which order the keys are deleted. The final trees will be identical. If true, provide a proof. If false, provide a counterexample.

13. Consider a branch of a binary search tree. This branch divides the tree into three sets: set A, the elements to the left of the branch; set B, the elements on the branch; and set C, the elements to the right of the branch. Provide a small counterexample to the statement "for all $a \in A$, $b \in B$, and $c \in C$, $a \leq b \leq c$."

14. Consider a bst T whose keys are distinct. We call key y a **successor** of key x iff $y > x$ and there is no key z for which $y > z > x$. Show that if the right subtree of the node storing key x in T is empty and x has a successor y, then the node that stores y must be the lowest ancestor of x whose left child is also an ancestor of x. Show two examples of this, one in which y is the parent of x, and one for which y is not the parent of x. Label both x and y.

Exercise Hints and Answers.

1. Using mathematical induction, if there is one node, then there are 2 null links. Now suppose a tree with n nodes has $n+1$ null links. Let T be a tree with $n+1$ nodes, and l a leaf of T. Removing l from T yields a tree with n nodes, and hence $n+1$ null links (by the inductive assumption). Hence, adding l back to T removes one null link, but adds two more, which implies that T has $n+1-1+2 = n+2$ null links.

2. This follows from the identity $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$.

3. Proof by induction. If a tree has one node then it has 0 full nodes, and 0+1 leaves. Now suppose that a tree with n full nodes has $n+1$ leaves. Let T be a tree with $n+1$ full nodes. Let l be a leaf of T. If the parent of l is not full, then removing l from T produces a new tree with the same number of leaves and full nodes. So, by continually plucking leaves from T, eventually we will find a leaf l whose parent is full. In this case, removing l from T produces a tree with n full nodes and hence, by the inductive assumption, $n+1$ leaves. Thus, T must have $n+1+1 = n+2$ leaves.

4. ```
   Boolean is_balanced(Node n)
   {
       if(n == null)
           return true;

       return is_balanced(n.left) && is_balanced(n.right) &&
           abs(height(n.left)-height(n.right)) <= 1;
   }
   ```

5. ```
   boolean is_bst(Node n)
   {
       if(n == null)
           return true;

       int value = n.value;

       if(n.left != NULL && value <= n.left.value)
        return false;

       if(n.right != NULL && value >= n.right.value)
        return false;

       return is_bst(n.left) && is_bst(n.right);

   }
   ```

6. ```
   void print(Node n)
   {
       if(n == null)
           return;
   ```
   ```

```
    print(n.left);
    print(" " + n.value + " ");
    print(n.right);
}
```

7. height 2: insert in the order of 10,4,17,1,5,16,21; height 4: insert in the order of 16,10,5,4,1,17,21, height 6: insert in order 21,17,16,10,5,4,1 Internal path lengths: height 2 = 10, height 4 = 13, height 6 = 21

8. See Exercise 6.

9. Best case number of steps for i th key is $O(\lfloor \log i \rfloor)$. Hence, $T(n) = \Omega(\sum_{i=1}^{n} \log i) = \Omega(n \log n)$ by the Integral theorem.

10. a) No. Does not end with 363
    c) No. 912 cannot be in the left subtree of 911

11. True since the key will be found along exactly the same path that it was inserted.

12. Hint: consider the binary search tree with nodes inserted in the following order: 1,10,5,2,8,7,9

13. Use the tree of the previous problem.

14. If y is the parent of x and x is the left child of y, then the statement is true, since technically x is an ancestor of itself. So suppose y is not the parent of x. Consider the first ancestor y of x whose left child is also an ancestor of x. Then we must have $x < y$ and $x > z$ for every descendant z of y that lies between y and x (exclusive) on the branch from y to x. It remains to show that there can be no element z for which $x < z < y$. Certainly no such z will be found in the left subtree of y, since x has no right subtree, and every ancestor of x between y and x (exclusive) is less than x. Also, no such z will be found in the right subtree of y, since all these elements are greater than y. Moreover, for any ancestor z of y, if y is in z's left subtree, then $z > y$. If y is in z's right subtree, then $y > z$, but also $x > z$. Finally, if z is an element for which z is in one subtree of some ancestor a of y, while y is in the other subtree, then, if $y < a$, then $y < z$. Similarly, if $y > a$, then $y > z$. But also, $x > a$, and hence $x > z$. Therefore, no such z can exist. Note: we call this a "proof by cases".