Week 8: Trees

Readings

- **Required**
  - [Weiss] ch18.1 – 18.3
  - [Weiss] ch18.4.4
  - [Weiss] ch19.1 – 19.2
- **Exercises**
  - [Weiss] 18.1, 18.2, 18.3, 18.9
- **Fun**

Tree

- root
- internal nodes
- leaves

Relationship

- A is **parent** of B and C
- B and C are **children** of A
- B and C are **siblings**
A node is an ancestor of itself, and a descendant of itself.
Applications

- Family Tree
- Directory Tree
- Organization Chart

Tree is recursive!

Implementation

```
class TreeNode
{
  Object element;
  TreeNode firstChild;
  TreeNode nextSibling;
  // Methods...
}
```

Implementation
Binary Trees

Just like a tree, a binary tree is recursive in nature.

An empty binary tree is just a reference to null.

We can add other members, such as a reference to parent (see successor()) and size of the subtree (see findKth()).
Size of a Tree

\[ \text{size}(T) \]
if \( T \) is empty
  \[ \text{return} \ 0 \]
else
  \[ \text{return} \ 1 + \text{size}(T.\text{left}) + \text{size}(T.\text{right}) \]

Height of a Tree

\[ \text{height}(T) \]
if \( T \) is empty
  \[ \text{return} \ -1 \]
else
  \[ \text{return} \ 1 + \max(\text{height}(T.\text{left}), \text{height}(T.\text{right})) \]

In a full binary tree, every node must have either 0 or 2 children.

A complete binary tree is a full binary tree where all leaves are of the same depth.
Property

How many nodes in a complete binary tree of height h?

Number of nodes $= 2^{h+1} - 1$

Height is $O(\log N)$.

Binary Tree Traversal

Post-order Traversal

```
postorder(T)
if T is not empty then
    postorder(T.left)
    postorder(T.right)
    print T.element
```

Pre-order traversal

```
preorder(T)
if T is not empty then
    print T.element
    preorder(T.left)
    preorder(T.right)
```
In-order Traversal

inorder(T)
if T is not empty then
inorder(T.left)
print T.element
inorder(T.right)

Traversal Example

Post-order: 4 8 9 5 2 0 6 7 3 1

Traversal Example

Pre-order: 1 2 4 5 8 9 3 6 0 7

Traversal Example

In-order: 4 2 8 5 9 1 6 0 3 7
Level-order Traversal

Level-order: 1234567890

What do you get when you replace the queue with a stack?

Binary Search Tree

Dynamic Set Operation

- `insert` (key, data)
- `delete` (key)
- `data = search` (key)
- `key = findMin`()
- `key = findMax`()
- `key = findKth`(k)
- `data[] = findBetween` (low, high)
- `successor` (key)
- `predecessor` (key)
Running Time

<table>
<thead>
<tr>
<th></th>
<th>Unsorted Array/List</th>
<th>Sorted Array</th>
<th>Sorted LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>O(1)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>O(N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>find</td>
<td>O(N)</td>
<td>O(logN)</td>
<td></td>
</tr>
<tr>
<td>findMin</td>
<td>O(N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>findMax</td>
<td>O(N)</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>

Recap

<table>
<thead>
<tr>
<th></th>
<th>Unsorted array/list</th>
<th>Sorted Array</th>
<th>Sorted List</th>
</tr>
</thead>
<tbody>
<tr>
<td>findKth</td>
<td>O(N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>find Between</td>
<td>O(N)</td>
<td>O(k + logN)</td>
<td></td>
</tr>
<tr>
<td>successor</td>
<td>O(N)</td>
<td>O(log N)</td>
<td></td>
</tr>
</tbody>
</table>

Variable k is the size of the output of findBetween().

Binary Search Tree

- All operations O(log N)
- findBetween O(k + logN)

The BST property holds recursively, which means the left sub-tree and right sub-tree must be BST as well.
What do you get when you traverse a BST in inorder?

Finding Minimum Element

while T.left is not empty
  T = T.left
return T.element

Finding x in T

while T is not empty
  if T.element == x then
    return T
  else if T.elements < x then
    T = T.left
  else
    T = T.right
return NOT FOUND
How to Insert 6?

After Inserting 6

```
insert(x, T)
if T is empty
    return new BinaryNode(x)
else if x < T.element
    T.left = insert(x, T.left)
else if x > T.element
    T.right = insert(x, T.right)
else
    ERROR!
    return T
```

How to delete?

Method insert(x, T) returns the new tree after inserting x into T.
Method delete(x,T) returns the new tree after deleting x from T.
delete(x, T): Case 3

if T has two children
    if x == T.element
        T.element = findMin(T.right)
        T.right = delete(T.element, T.right)
    else if x < T.element
        T.left = delete(x, T.left)
    else
        T.right = delete(x, T.right)

return T

Successor returns the next larger element in the tree.
Successor(5) is 6.
Successor(4) is 5.
11 does not have a successor.

Successor(T)

// find next largest element
if T.right is not empty
    return findMin(T.right)
else if T is a left child
    return parent of T
else T is a right child
    let x be the first ancestor of T that is a left child
    return parent of x

• What happen if we cannot find such an x?
  This means that there is no successor for T.
  (i.e. T is the maximum).
• We need a reference to the parent for this operation, so that we can traverse up the tree.
• Second and third case can actually be combined into one.
• Question: why is the algorithm on the left correct? Think about it using the property of BST.
findKth(T,K)

Size of a Tree

findKthSmallest(T,K)

findKthLargest(T,K)

Observation:
- if a node, T, has 6 elements in its right sub-tree, we know that T is the 7th largest element in the tree.
- The 1st, 2nd, ..., 6th largest elements must be in the right sub-tree.
- The 9th largest element in T is the 2nd largest element in the left sub-tree of T. (9 – 6 – 1 = 2)
Running Time
- find $O(h)$
- findMin $O(h)$
- insert $O(h)$
- delete $O(h)$
- successor $O(h)$
- findKth $O(h)$

BUT

$h = O(N)$

When you insert nodes in increasing order, you get a skewed tree. Therefore $h$ is actually in $O(N)$. 
/** *
* Return the node containing the successor of x. This method is part of *
* BinarySearchTree class. I assume that BinaryNode has a member called *
* parent. If a node is the root, parent points to null, otherwise it *
* points to its parent. (Modifying insert/delete to maintain the parent *
* pointer is a good exercise to help you understand BinarySearchTree.) *
* *
* @param x the item whose successor we want to search for. *
* @return the successor or null if no successor exists. *
*/

public BinaryNode successor( Comparable x )
{
    BinaryNode t = find(x, root);
    if (t.right != null)
    {
        // right child is not empty, just call findMin on the right
        // child.
        return findMin(t.right);
    }
    else // t has no right child
    {
        if (t.parent == null)
        {
            // t is the root and has no right child. so t must be
            // the largest. (i.e. no successor).
            return null;
        }
        else if (t.parent.left == t)
        {
            // t is a left child, return the parent.
            return t.parent;
        }
        else if (t.parent.right == t)
        {
            // t is a right child. find the first ancestor that is
            // a left child.
            BinaryNode p = t.parent;
            while (p.parent != null)
            {
                if (p.parent.left == p)
                {
                    // p is the first ancestor that is a left child.
                    // return its parent.
                    return p.parent;
                }
                else
                {
                    // proceed to the next ancestor.
                    p = p.parent;
                }
            }
            // reach the root and found nothing. t must be the largest.
            return null;
        }
    }
    return null; // to make compiler happy.