Class Notes CS 3137

1 Binary Search Trees

- A Binary Search Tree is an ordered tree in which each node is represented by a key. The ordering is such that:

  Keys in Left Subtree $<$ Key K in node N $<$ Keys in Right Subtree

- To search for an item with key K in the tree:

  1. Compare K with the key in the root.
  2. If K = the key in the root, then we have found the key, and return the pointer to this node.
  3. If K $<$ the the key in the root, then we search the left subtree, else we search the rightsubtree.
  4. If we ever reach a NULL node (empty subtree), the item wasn't in the tree. Return NULL pointer.

- To insert an item with key K into an existing binary search tree:

  1. Compare the new key K with the key in the root.
  2. If K = the key in the root, then this item is already inserted into the tree. Either update the node with new information (essentially a find operation is being done here) or mark this as an error.
  3. If K $<$ the the key in the root, then we search the left subtree, else we search the rightsubtree.
  4. If we ever reach a NULL node (empty subtree), the item wasn't in the tree. We need to insert this new node as a child of the node we last visited.

- The power of using a Binary Search Tree is this: We never have to search more than the height of the tree, and the height of the tree is $\approx Log(N)$. This is a major speedup in search over sequential (i.e. $O(N)$ searches).

- CAVEAT: all this analysis assumes the tree is reasonably balanced (as many left children as right children).

- Note that a sequential linked list is nothing more than a degenerate binary tree with all the children being on one side of every node. In this case, the height of the tree is N.

- Another method you need to consider in a Binary Search Tree is a delete method. If you delete a node, you can change the shape of the tree. Careful analysis is needed to do this correctly.

- There are 3 cases that can occur at a deleted node (discussed in section 4.3.4 of your book):

1. The node is a leaf. Action: Simply delete the node.
2. The node has a 1 child. Action: adjust the tree by having the parent of the deleted node point to the single child of the deleted node.
3. The deleted node has 2 children. Action: Replace the deleted node with the successor node in the tree's ordering. That means find the leftmost item of the deleted node's right subtree. Once we find the successor node, we simply move its element's contents into the delete node's element's contents, and now delete the successor. This last deletion of the successor is easy since it is guaranteed by definition that the successor will have either no children or at most 1 child. This is because it is the leftmost item of the right subtree.

```
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
    {
        if( t == null )
            return new BinaryNode<AnyType>( x, null, null );

        int compareResult = x.compareTo( t.element );
        if( compareResult < 0 )
            t.left = insert( x, t.left );
        else if( compareResult > 0 )
            t.right = insert( x, t.right );
        else
            ;  // Duplicate; do nothing
        return t;
    }

private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
    {
        if( t == null )
            return t;   // Item not found; do nothing

        int compareResult = x.compareTo( t.element );
        if( compareResult < 0 )
            t.left = remove( x, t.left );
        else if( compareResult > 0 )
            t.right = remove( x, t.right );
        else if( t.left != null && t.right != null ) // Two children
        {
            t.element = findMin( t.right ).element;
            t.right = remove( t.element, t.right );
        }
        else
            t = ( t.left != null ) ? t.left : t.right;
        return t;
    }
```

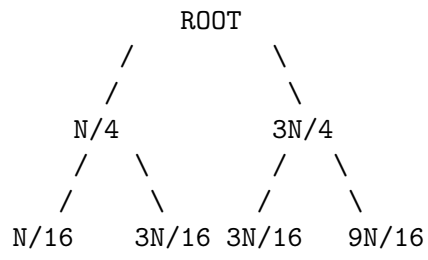
```
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

2 Efficiency of Binary Search Trees

- Binary search trees have an efficiency based upon the shape of the tree. We base our analysis on a *Find* operation, which searches the tree to locate a key in the tree.
- Worst Case: If the tree is degenerate - essentially a linked list - then the complexity is $O(N)$. As an example of this, inserting a set of words into the tree that are already in lexographic order will result in insertions to one side of the tree only.
- Best Case: If the tree is evenly balanced, then we can use our standard analysis that says:

$$T(N) = T(\frac{N}{2}) + 1 \tag{1}$$

  since we are cutting off half the tree each time we take a right or left link. This recurrence can be solved to yield a complexity of $O(Log\, N)$.
- Typical Case: What about something between the worst and best cases? Let's assume that the "split" in the data occurs each time about 3/4 of the way. So, for example, the root element will split the N data items into 2 subtrees, one with 3/4 N nodes, the other with 1/4 N nodes. Lets assume that at each successive split, this ratio holds. We put 1/4 of the elements into the left tree, and 3/4 in the right, all the way down to the leaf nodes. You can envision a tree that looks like this:



The largest subtree of this tree is the rightmost, And the rightmost subtree has exactly

$$\left(\frac{3}{4}\right)^d N$$

elements in it at level d.

- Given N, the total number of nodes, we can solve for the depth of the tree d at which the largest subtree becomes a single node. This is equivalent to the longest search path we need to travel to access a node. In other words, when does

$$\left(\frac{3}{4}\right)^d N \ \leq \ 1 \ ?$$

- To solve for d, we take the the Log of both sides of the equation:

$$\left(\frac{3}{4}\right)^d N \ \leq \ 1$$

$$dLog(\frac{3}{4}) \ + \ Log(N) \ \leq \ Log(1)$$

$$and \ since \ Log(\frac{3}{4}) \approx -.4, \quad , \quad d \ \geq \ \frac{Log(N)}{.4} \ \ or \ \ d \geq \ 2.5Log(N)$$

- So at a depth of about $2.5Log(N)$, we find the deepest leaf node. This is a small price to pay, since $2.5Log(N) \ \approx O(Log(N))$. So these trees are efficient even if we have a somewhat uneven split.