Binary Search Trees

Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements the nodes. If the elements are numbers then there is obviously an ordering. If the elements are strings, then there is also a natural ordering, namely the dictionary ordering, also known as “lexicographic ordering”.

Definition: binary search tree

A *binary search tree* is a binary tree such that

- each node contains an element, called a *key*, such that the keys are comparable, namely there is a strict ordering relation < between keys of different nodes
- any two nodes have different keys (i.e. no repeats, i.e. duplicates)
- for any node,
  - all keys in the left subtree are less than the node’s key
  - all keys in the right subtree are greater than the node’s key.

  Note this is stronger than just saying that the left child’s key is less than the node’s key which is less than the right child’s key. For example, the following is not a binary search tree.

```
    5
   /
  2
 / \
1   6
```

One important property of binary search trees is that *an inorder traversal of a binary search tree gives the elements in their correct order.* Here is an example with nodes containing keys `abcdefghijklm`. Verify that an inorder traversal gives the elements in their correct order.

```
            g
         /     \
       d          k
     /  \       /   \
   b     f     i      l
 /  \   /     / \      \
a    c  e    h   j      m
```

Here is another example with the same set of keys:

```
                  k
               /     \
             e         l
           /  \         \
         b      g         m
       /  \    / \
      a    c  f   i
            \    / \
             d  h   j
```

BST as an abstract data type (ADT)

One performs several common operations on binary search trees:

- `find(key)`: given a key, return a reference to the node containing that key (or null if key is not in tree)
- `findMin()` or `findMax()`: find the node containing the smallest or largest key in the tree and return a reference to the node containing that key
- `add(key)`: insert a new node into the tree such that the node contains the key and the node is in its correct position (if the key is already in the tree, then do nothing)
- `remove(key)`: remove from the tree the node containing the key (if it is present)

Here are algorithms for implementing these operations.

```
find(root,key){                  //    returns a node
  if (root == null)
     return null
  else if (root.key == key))
    return root
  else if (key < root.key)
     return  find(root.left, key)
  else
     return  find(root.right, key)
}

findMin(root){            //  returns a node
   if (root == null)      // only necessary for the first call
     return null
   else if (root.left == null)
      return root
   else
      return findMin(root.left)
}
```

For example, here the minimum key is `a`.

```
         f
      /     \
     c       m
   /   \     /
  a    e   g
```

Notice however that the minimum key is not necessarily a leaf i.e. it can occur if the key has a right child but no left child. Here the minimum key is `c`:

```
           f
        /     \
       c       m
        \     /
         e   g
        /
       d
```

The reasoning and method is similar for `findMax`.

```
findMax(root){                 // returns a node
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMax(root.right)
}
```

Let's next consider adding (inserting) a key to a binary search tree. If the key is already there, then do nothing. Otherwise, make a new node containing that key, and insert that node into its *unique* correct position in the tree.

```
add(root,key){      // returns root
   if (root == null)              //  base case:
      root =  new BSTnode(key)    //  makes a new node and returns it
   else if (key < root.key){
      root.left = add(root.left,key)
   else if (key > root.key){
      root.right = add(root.right,key)
   return root
}
```

The new node is always added at a leaf and so the base case is always reached eventually. For the non-base case, the situation is subtle. The code says that a new node is added to either the left or right subtree and then the reference to the left or right subtree is re-assigned. It is reassigned to the root of the left or right subtree, which has the new node added it. Why is it necessary to reassign the reference like that?

[ASIDE: I had forgotten to review this detail before the Sec. 001 class (lecture recording) and I promised I would fill it into the lecture notes: here we are!]
Suppose we add the new node to the left subtree. If the new node is a descendent of the root node of the left subtree , then the assignment `root.left = add(root.left,key)` doesn't change the reference `root.left` in the `root` node; it just assigns it to the same node it was referencing beforehand. *However,* if `root.left` was null and then the new node was added to the left subtree of `root`, then the call `add(root.left,key)` will create and return the new node, namely it is the root of a subtree with one node. If we don't assign that node to `root.left` as the code says, then the new node that is created will not in fact be added to the tree.

Next, consider the problem of removing a node from a binary search tree.

```
remove(root, key){                  // returns root
    if( root  == null )
         return null
    else if ( key < root.key )                                    (*)
         root.left = remove( root.left, key )
    else  if ( key > root.key )                                    (*)
         root.right = remove( root.right, key)
    else  if root.left == null                                     (**)
         root  =  root.right         // or just "return root.right"
    else  if root.right == null                                    (**)
         root  = root.left           // or just "return root.left"
    else{                                                          (***)
         root.key = findMin( root.right).key
         root.right = remove( root.right, root.key )
    }
    return root
}
```

The (*) conditions handle the case that the key is not at the root, and in this case, we just recursively remove the key from the left or right subtree. Note that we replace the left or right subtree with a subtree that doesn't contain the key. To do this, we use recursion, namely we remove the key from the left or right subtree.

The more challenging case is that the key that we want to remove is at the root (perhaps after a sequence of recursive calls). In this case we need to consider four possibilities:

- the root has no left child
- the root has no right child
- the root has no children at all
- the root has both a left child and a right child

In the first two cases – see (**) in the algorithm – we just replace the root node with the subtree in the non-empty child. Note that the third case is accounted for here as well; since both children are null, the root will become null.

In the fourth case – (***) – we take the following approach. We replace the root with the minimum node in the right subtree. It does so in two steps. It copies the key from the smallest node in the right subtree into the root node, and then it removes the smallest node node from the the right subtree.

Example (remove)

Take the following example with nodes `abcdefghijklm`:

```
            g
         /     \
       d          k
     /  \       /   \
   b     f     i      l
 /  \   /     / \      \
a    c  e    h   j      m
```

and then remove elements `g,f,k` in that order.

```
           h
        /     \
     d           k
   /   \       /   \
  b     f     i      l
 /  \   /       \      \
a    c  e        j      m
```

```
           h
        /     \
      d          k
     /  \       /   \
   b     e     i      l
 /  \           \      \
a    c           j      m
```

```
            h
         /     \
       d          l
     /  \       /   \
   b     e     i      m
 /  \           \
a    c           j
```

Let's consider the best and worst case performance. The best case performance for this data structure tends to occur when the keys are arranged such that the tree is balanced, so that all leaves are at the same depth (or depths of two leaves differ by at most one). However, if one adds keys into the binary tree and doesn't rearrange the tree to keep it balanced, then there is no guarentee that the tree will be anywhere close to balanced, and in the worst case a BST with n nodes could have height $n-1$. This implies the following best and worst cases:

Operations/Algorithms for Lists	Best case, $t_{best}(n)$	Worst case, $t_{worst}(n)$
findMax(), findMin()	$\Theta(1)$	$\Theta(n)$
find(key)	$\Theta(1)$	$\Theta(n)$
add(key)	$\Theta(1)$	$\Theta(n)$
remove(key)	$\Theta(1)$	$\Theta(n)$

In COMP 251, you will learn about balanced binary search trees, e.g. AVL trees or red-black trees. If a tree is balanced, then the operations are all $\Theta(\log n)$. This is an interesting topic, but technically involved so I won't say anything more about it here.