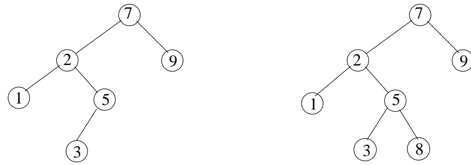
Binary Search Trees (Chapter 18)

A *binary search tree* is a binary tree that maintains an ordering property, which is what allows for fast search to determine whether or not an element is in the tree. The ordering property is very simple.

Ordering Property: for every node X in the tree, the values of all the keys in the left subtree are smaller than the key in X, and the values of all the keys in the right subtree are large than the key in X.

Note that the ordering property does not permit duplicates to appear in the binary search tree.

Question: do either of the following trees maintain the ordering property?



Find Operations on Binary Search Trees

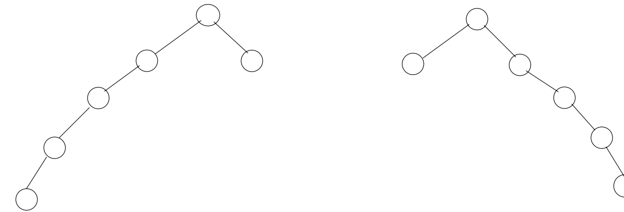
The operations on a search tree are primarily to find out whether or not an element is in the tree.

We can define a general Find algorithm, but we also want to define special FindMin and FindMax algorithms. In each case the algorithms will work recursively.

The FindMin operation simply follows the left child pointer of each node, starting at the root until the leftmost leaf node is found.

The FindMax operation simply follows the right child pointer of each node, starting at the root until the rightmost leaf node is found.

The FindMin and FindMax operations work very efficiently, O(lg n) on a binary search tree if the tree is reasonably balanced. An heavily unbalanced tree can lead to O(n) running time. Why?



Fiind, FindMin, and FindMax Recursive Algorithms

A SearchTree<T> class would define FindMin, FindMax and Find operations as follows: (assuming BinaryNode<T> components are accessible by the SearchTree<T> methods). We return a BinaryNode<T>*.

```
template<class T> BinaryNode<T>* SearchTree:findMin(BinaryNode<T>* node) const
{
      if (node != NULL)
         while (node->left != NULL)
           node = node->left;
      return node;
}

template<class T> BinaryNode<T>* SearchTree:findMax(BinaryNode<T>* node) const
{
      if (node != NULL)
         while (node->right != NULL)
           node = node->right;
      return node;
}

template<class T> BinaryNode<T>* SearchTree::find (const T& x, BinaryNode<T> *node)
{
      while (node != NULL) {
         if (x < node->elem)
           node = node->left;
         else if (x > node->elem)
           node = node->right;
         else
           return node;
      }
      return NULL;
}
```


Tree Maintenance Operations

Beside the find operations, we need operations to insert a new node and remove a node from the search tree. Addition is quite easy. We use an overloaded insert method. One for the user interface, another a recursive routine that takes a value of type T and a reference to a pointer to a node.

```
template<class T> int SearchTree<T>::insert(const T& x)
{
      return insert(x, root); // start recursive insert from the root
}
```

Note that we use a BinaryNode<T> *& (reference to a pointer to a node), so that we can update the actual pointer field. If we just passed a BinaryNode<T>*, then the assignment to a node would not change the value in the pointer variable since a pointer is passed by value. If node is root, and the tree is empty, then a new node is created and root is set to point a the new node.

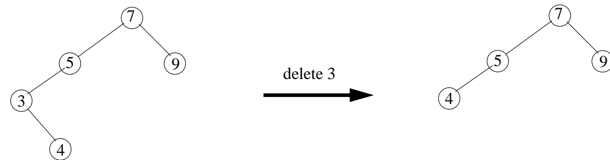
```
template<class T> int SearchTree<T>::insert(const T& x, BinaryNode<T> *& node)
{
      if (node == NULL) {
         node = new BinaryNode<T>(x);  // assign new node pointer to node
         return 1;                     // returning 1 indicates success
      }
      else
         if (x < node->elem)
           return insert(x, node->left);
      else
         if (x > node->elem)
           return insert(x, node->right);

      return 0;  // x == node->elem, which is not allowed to be inserted
}
```

The Remove Node Operation

Removal is a bit harder since we have to maintain the order property when we remove a node. Removal of a leaf node is trivial. Removal of an interior node with one child is also easy since we can just have the parent of the node being removed point at the child node of the node being removed.

Question: how would we remove the minimum (maximum) element from the following tree?

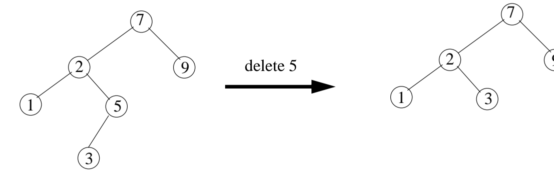



```
template<class T> int SearchTree<T>::removeMin(BinaryNode<T> *& node)
{
      if (node == NULL) // tree empty
         return 0;
      else
         if (node->left != NULL)
           return removeMin(node->left); // recurse down left branch

      BinaryNode<T>* tmp = node;
      node = node->right;
      delete tmp;
      return 1;
}
```


The Remove Node Operation

The complicated case is when you remove a node with one or two children. The children must then be pulled up to the next level, which may also require a bit of reorganization so that the ordering property is maintained.

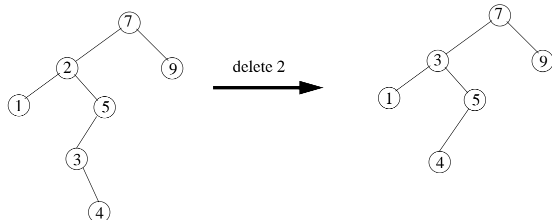


When you remove a node with a single child node, you make the parent of the node being removed point at the child.

The Remove Node Operation

Removing a node with two children is a bit more complicated. We may need to pull up a deeper child node to replace the node being deleted so that we maintain the ordering property. Otherwise, our find algoritms will be broken as they rely on branching left or right depending on whether or not the value being looked for is less than or greater than the value at some node.

So, in the following case, we have to pull up the left child of the node containing 5 and place it where node 2 used to be. this also requires that we re-attach the right child of the node containing 3 to be the left child of the node containing 5 since 4 is less than 5. So, the order property is then maintained in the resulting tree.



Note that we need to keep track of parent nodes in order for remove to work, but the BinaryNode<T> class we have defined does not have a pointer that points back up the tree, only left and right pointers that point down the tree. The fact that we use recursion, and pass a node as a reference to a pointer, we can avoid having the parent pointer in each node.

Recursive Remove Operation

The remove operation finds the element to remove, and depending on the number of child nodes, removes the element. Note that both findMin and removeMin are used to help in the removal operation.

```
template<class T> int SearchTree<T>::remove(const T& x, BinaryNode<T> *& node)
{
      BinaryNode<T> *tmp;

      if (node == NULL)
        return 0;  // return failure
      else
         if (x < node->elem) // go left
           return remove(x, node->left);  // recurse down left branch
      else
         if (x > node->elem) // go right
           return remove(x, node->right);  // recurse down right branch
      else
         if (node->left != NULL && node->right != NULL)  // 2 children
         {
           tmp = findMin(node->right);
           node->elem = tmp->elem;
           return removeMin(node->right);
         }

      // 0 or 1 child

      tmp = node;
      node = (node->left != NULL) ? node->left : node->right;  // re-root node
      delete tmp;  // delete old root

      return 1;  // return success
}
```