CS211, Lecture 19 Binary Search Trees

Readings: Weiss, chapter 19, sections 19.1--19.2.

I think that I shall never scan A tree as lovely as a man. . . . . A tree depicts divinest plan, But God himself lives in a man. Joyce Kilmer

I like trees because they seem more resigned to the way they have to live than other things do.
Author: Willa Sibert Cather

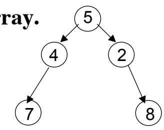
Binary search trees

You know that searching is faster in a sorted array than in a nonsorted array. If the data set has size n:

On the average: O(n) for a nonsorted array.

Always: O(log n) for a sorted array

On the average, O(n) for a binary tree.

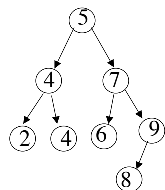


We now see how to restrict trees so that they can be searched in average time O(log n).

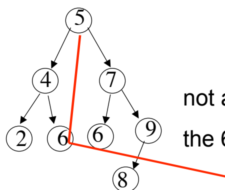
Binary search tree

Binary search tree: a binary tree in which, for each node n,

- **All the values in subtree n.left are <= the value in node n**
- **All the values in subtree n.right are >= the value in node n.**



binary search tree



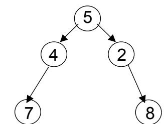
not a binary search tree: the 6 is > 5.

Binary tree

Binary tree: tree in which each node can have at most two children.

Redefinition of binary tree to allow empty tree

A **binary tree** is either
(1) Ø (the empty binary tree)
or
(2) a root node (with a value),
a left binary tree,
a right binary tree



Binary tree

tree with root 4 has an empty right binary tree
tree with root 2 has an empty left binary tree
tree with root 7 has two empty children.

Class for binary search tree nodes

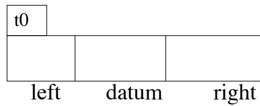
```
/** An instance is a nonempty binary search tree */
public class BSTNode {
   / ** class invariant: the left and right subtrees satisfy the
          binary search tree property */
   private Object datum;
   private BSTNode left;
   private BSTNode right;

   /** Constructor: a one-node tree with root value ob */
   public BSTNode(Object ob)
      { datum= ob; }

   **getter and setter methods for all three fields**
}
```

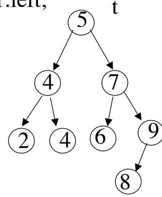

Method toString

```
/** An instance is a nonempty binary search tree */
public class BSTNode {
    /** = nodes of tree, separated by , */
   public String toString() {
      String result= "";
      if (left != null) {
         result= result + left.toString() + ", ";
      }
      result= result + datum;
      if (right != null) {
         result= result + ", " + right.toString();
      }
      return result;
   }
}
```

Searching a binary search tree for a value

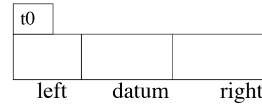
```
/** = a node of tree t that contains value x (null if none)  */
public BSTNode elementAt (Comparable x, BSTNode t) {
   BSTNode t1= t;
   // invariant: x is in t iff x is in tree t1
   while (t1 != null) {
        if (x.compareTo(t1.datum) == 0) return t1;
        if (x.compareTo(t1.datum) < 0) t1= t1.left;
        else t1= t1.right;
   }
   return null;
}
```

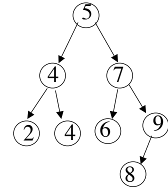




Searching a binary search tree for a value (using recursion)

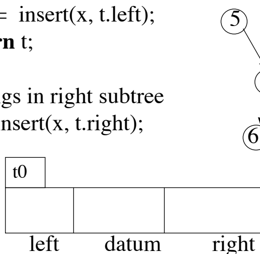
```
/** = a node of tree t that contains value x (null if none)  */
public BSTNode elementAt (Comparable x, BSTNode t) {
   if (t == null) return null;
   if (x.compareTo(t.datum) == 0) return t;
   if (x.compareTo(t.datum) < 0)
        return elementAt(x, t.left);
   return elementAt(x, t.right);

}
```

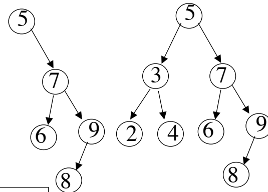





```
/** Insert item x into tree t and return the new root  */
public BSTNode insert (Comparable x, BSTNode t) {
   if  (t == null) {
        t= new BSTNode(x);
        return t;
   }
   if (x.compareTo(t.datum) <= 0) {
        t.left=  insert(x, t.left);
        return t;
   }
   // x belongs in right subtree
   t.right=  insert(x, t.right);
   return t;
}
```

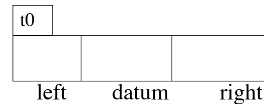
Duplicates are allowed. They are arbitrarily put in the left subtree. If duplicates not allowed, have method throw an exception if x already in t

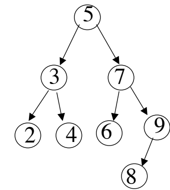




A method to create binary trees easily

```
/** = a tree with Integer values given by array b */
public static BSTNode newTree(int[] b) {
     BSTNode root= null;
     for (int k= 0; k != b.length; k= k+1) {
        root= insert(new Integer(b[k]), root);
     }
     return root;
}
```

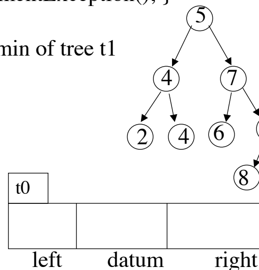


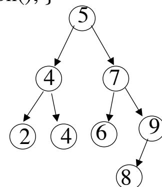


Searching a binary search tree for the minimum

```
/** = a node that contains the minimum value in tree t.
        Throw IllegalArgumentException if t == null) */
public BSTNode findMin (BSTNode t) {
   if  (t == null)
        { throw new IllegalArgumentException(); }
   BSTNode t1= t;
   // inv: the min of tree t is the min of tree t1
   while (t1.left != null) {
        t1= t1.left;
   }
   return t1;
}
```

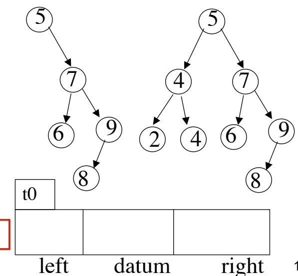
You write findMax(BTNode)

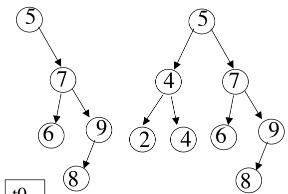




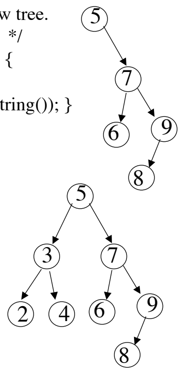

```
/** Remove the min node from tree t and return the root
      of the new tree.  Throw IllegalArgumentException if
      t ==  null */
public BSTNode removeMin (BSTNode t) {
   if  (t == null)
        { throw new IllegalArgumentException(); }
   if (t.left == null)
        { return t.right; }
   t.left= removeMin(t.left);
   return t;

}
```

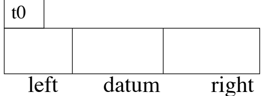
You write removeMax(BTNode)





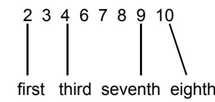
```
/** Remove node with item x from t; return root of new tree.
      Throw IllegalArgumentException if  x is not in t  */
public BSTNode remove (Comparable x, BSTNode t) {
   if  (t == null)
         { throw new IllegalArgumentException(x.toString()); }
   if (x.compareTo(x, t.datum) < 0)
        { t.left= remove(x, t.left);   return t; }
   if (x.compareTo(t.datum) > 0)
        { t.right= remove(t.right);   return t; }
   // { x is the value in root t }
   if (t.right != null) {
      t.datum= findMin(t.right).datum;
      t.right= removeMin(t.right);
      return t;
   }
   // { t.right is null }
   return t.left;
}
```

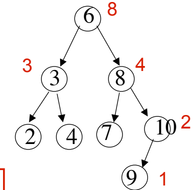




Order statistics

A binary search tree can be viewed as an ordered list. Suppose we want to find the kth smallest element.



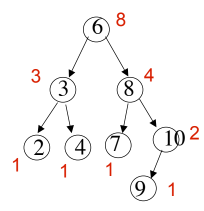


If we know the size of each tree, we can find the kth smallest element quite quickly.

Extend class to have a field that gives the size.

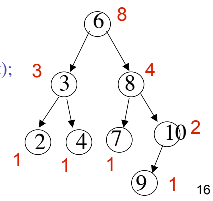
BSTNode with size

```
public class BSTNodeSize extends BSTNode {
  private int size;  // Number of nodes in tree

  // Constructor: a one-node tree with value ob
  public BSTNodeSize(Object ob) {
     super(ob);
     size= 1;
  }

}
```



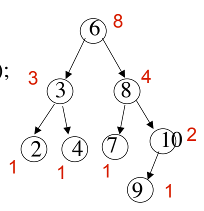
Find kth smallest value

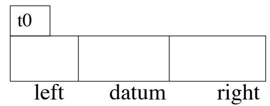
```
/** = rank k node of tree t. k < 0 or k > size of tree, throw exception */
public BSTNodeSize findKth (int k, BSTNodeSize t) {
  if (t == null)
     throw new IllegalArgumentException();
  int lsize= 0;  // size of left subtree
  if (t.left != null)
     lsize= ((BSTNodeSize)(t.left)).size;
  if (k = lsize + 1)
     return t;
  if (k <= lsize)
     return findKth(k, (BSTNodeSize)t.left);
return findKth(k – lsize – 1, (BSTNodeSize)t.right);
}
```



Methods that remove and insert have to be overriden

```
/** Remove the min node from tree t and return the root
      of the new tree.  Throw IllegalArgumentException if
       t ==  null */
public BSTNodeSize removeMin (BSTNodeSize t) {
   if  (t == null)
        { throw new IllegalArgumentException(); }
   if (t.left == null)
        { return (BSTNodeSize)t.right; }
   t.left= removeMin((BSTNodeSize)t.right);
   t.size= t.size – 1;
   return t;
}
```





You rewrite removeMax(BTNode)