CMSC 132, Object-Oriented Programming II **Summer 2017**

Lecture 13

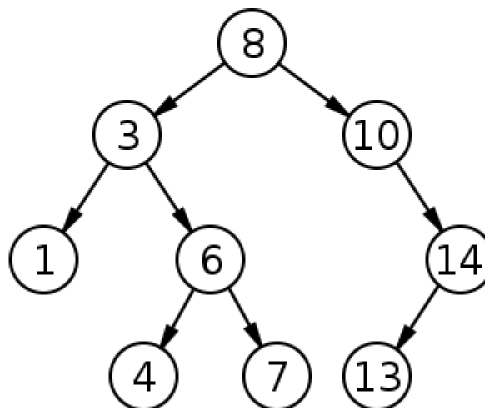
Lecturer: Anwar Mamat

Disclaimer: *These notes may be distributed outside this class only with the permission of the Instructor.*

13.1 Binary Search Tree

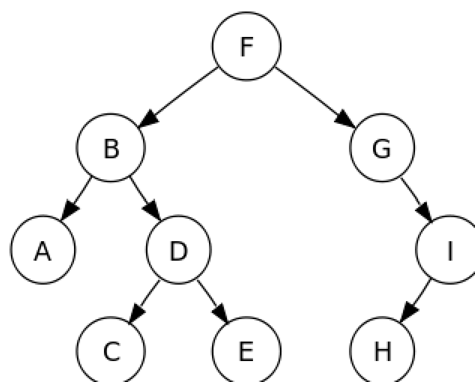
A binary search tree is a rooted binary tree, in which key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree. InOrder traversal of a binary search tree sorts all keys.

Figure 13.1: Binary Search Tree



InOrder traversal of the binary search tree shown in Figure 13.1: 1,3,4,6,7,8,10,13,14

Figure 13.2: Binarty Search Tree



InOrder traversal of the binary search tree shown in Figure 13.1: A,B,C,D,E,F,G,H,I

13.1.1 Binary Search Tree Implementation

Listing 1: Binary Search Tree

```
public class BST<Key extends Comparable<Key>, Value> {
    private Node root;
    private class Node{
        private Key key;
        private Value value;
        private Node left, right;
        public Node(Key k, Value v){
            key = k;
            value = v;
        }
    }
    BST(){
        root = null;
    }
    BST(Node r){
        this.root = r;
    }


    public void put(Key key, Value val){
        if(val == null) {
            delete(key);
            return;
        }
        root = put(root, key, val);
        //dumpToGraphViz();
    }

    private Node put(Node x, Key key, Value val){
        if(x == null) { return new Node(key,val);}
        int cmp = key.compareTo(x.key);
        if(cmp < 0) {x.left = put(x.left, key, val);}
        else if (cmp > 0) {x.right = put(x.right, key, val);}
        else {x.value = val;}
        return x;

    }

    public Value get(Key key){
        return get(root, key);
    }
    private Value get(Node x, Key key){
        if(x == null) return null;
        int cmp = key.compareTo(x.key);
        if(cmp < 0) return get(x.left, key);
        else if(cmp > 0) return get(x.right, key);
        else return x.value;
    }


    public void delete(Key key){
        root = delete(root, key);
    }

    public Value value(){
        return root.value;
    }
    public String info(){
        return root.key + "/" + root.value;
    }
```

```
61
62      public BST<Key,Value> getLeft(){
63          return new BST(root.left);
64      }
65      public BST<Key, Value> getRight(){
66          return new BST(root.right);
67      }
68
69      private Node delete(Node x, Key k){
70          if(x == null) return null;
71          int cmp = k.compareTo(x.key);
72          if(cmp < 0) x.left = delete(x.left, k);
73          else if (cmp > 0) x.right = delete(x.right, k);
74          else{
75              if(x.right == null) return x.left;
76              if(x.left == null) return x.right;
77              Node t = x;
78              x = min(t.right);
79              x.right = deleteMin(t.right);
80              x.left = t.left;
81          }
82          return x;
83
84      }
85      public void deleteMin(){
86          deleteMin(root);
87      }
88
89      private Node deleteMin(Node x){
90          if(x.left == null) return x.right;
91          x.left = deleteMin(x.left);
92          //System.out.println("Key=" + x.key  + "\tValue=" + x.value);
93          return x;
94      }
95      public void deleteMax(){
96          root = deleteMax(root);
97      }
98
99      private Node deleteMax(Node x){
100         if(x.right == null) return x.left;
101         x.right = deleteMax(x.right);
102         return x;
103     }
104
105     public Key min(){
106         if(isEmpty()) return null;
107         return min(root).key;
108     }
109
110     private Node min(Node x){
111         if(x.left == null) return x;
112         return min(x.left);
113     }
114
115     public Key max(){
116         if(isEmpty()) return null;
117         return max(root).key;
118     }
119
120     private Node max(Node x){
121         if(x.right == null) return x;
122         return max(x.right);
123     }
124     public boolean isEmpty(){
125         return size() == 0;
126     }
127
128     public int size(){
```

```
129         return size(root);
130     }
131 
132     private int size(Node x){
133         if(x == null) return 0;
134         else return 1 + size(x.left) + size(x.right);
135     }
136 
137     private int height(Node r){
138         if(r == null) return -1;
139         return 1 + Math.max(height(r.left), height(r.right));
140     }
141     public int height(){
142 
143         return height(root);
144     }
145     public boolean contains(Key key){
146         return get(key) == null;
147     }
148     public void preOrder(){
149         preOrder(root);
150     }
151     private void preOrder(Node r){
152         if(r == null) return;
153         System.out.print(r.key+",");
154         preOrder(r.left);
155         preOrder(r.right);
156     }
157 
158     public void postOrder(){
159         postOrder(root);
160     }
161     private void postOrder(Node r){
162         if(r == null) return;
163         postOrder(r.left);
164         postOrder(r.right);
165         System.out.print(r.key+",");
166     }
167 
168      public void inOrder(){
169         inOrder(root);
170     }
171     private void inOrder(Node r){
172         if(r == null) return;
173         inOrder(r.left);
174         System.out.print(r.key+",");
175         inOrder(r.right);
176     }
177 
178     public void levelOrder(){
179         levelOrder(root);
180     }
181     private void levelOrder(Node r){
182         if( r == null) return;
183         Queue<Node> q = new LinkedList();
184         q.offer(r);
185         while(!q.isEmpty()){
186             Node t = q.poll();//pop
187             System.out.print("<"+t.key+","+t.value+">");
188             if(t.left != null) q.offer(t.left);
189             if(t.right != null) q.offer(t.right);
190         }
191     }
192     public static void main(String[] args) {
193         BST<Integer, String> bst = new BST();
194         bst.put(100,"Alice");
195         bst.put(200,"Bob");
196      }
```

```
197 }
```

13.1.2 Non Recursive Implementation

Listing 2: Binary Search Tree

```
1
2  /*************************************************************************
3   * Copyright   2002?2010, Robert Sedgewick and Kevin Wayne.
4   *  Compilation:  javac NonrecursiveBST.java
5   *  Execution:    java  NonrecursiveBST < input.txt
6   *
7   *  A symbol table implemented with a binary search tree using
8   *  iteration instead of recursion for put(), get(), and keys().
9   *   *
10  *************************************************************************/
11
12 public class NonrecursiveBST<Key extends Comparable<Key>, Value> {
13
14     // root of BST
15     private Node root;
16
17     private class Node {
18         private Key key;             // sorted by key
19         private Value val;           // associated value
20         private Node left, right;    // left and right subtrees
21
22         public Node(Key key, Value val) {
23             this.key = key;
24             this.val = val;
25         }
26     }
27
28
29    /*********************************************************************
30     *  Insert key-value pair into symbol table (nonrecursive version)
31     *********************************************************************/
32     public void put(Key key, Value val) {
33         Node z = new Node(key, val);
34         if (root == null) {
35             root = z;
36             return;
37         }
38
39         Node parent = null, x = root;
40         while (x != null) {
41             parent = x;
42             int cmp = key.compareTo(x.key);
43             if      (cmp < 0) x = x.left;
44             else if (cmp > 0) x = x.right;
45             else {
46                 x.val = val;
47                 return;
48             }
49         }
50         int cmp = key.compareTo(parent.key);
51         if (cmp < 0) parent.left  = z;
52         else         parent.right = z;
53     }
54
55
56    /*********************************************************************
57     *  Search BST for given key, nonrecursive version
58     *********************************************************************/
59     Value get(Key key) {
```

```
60          Node x = root;
61          while (x != null) {
62              int cmp = key.compareTo(x.key);
63              if      (cmp < 0) x = x.left;
64              else if (cmp > 0) x = x.right;
65              else return x.val;
66          }
67          return null;
68      }
69
70     /*********************************************************************
71      *  Level-order traversal - need to make nonrecursive.
72      *********************************************************************/
73      public Iterable<Key> keys() {
74          Stack<Node> stack = new Stack<Node>();
75          Queue<Key> queue = new Queue<Key>();
76          Node x = root;
77          while (x != null || !stack.isEmpty()) {
78              if (x != null) {
79                  stack.push(x);
80                  x = x.left;
81              }
82              else {
83                  x = stack.pop();
84                  queue.enqueue(x.key);
85                  x = x.right;
86              }
87          }
88          return queue;
89      }
90
91
92     /***************************************************************************
93      *  Test client
94      ***************************************************************************/
95      public static void main(String[] args) {
96          String[] a = StdIn.readAllStrings();
97          int N = a.length;
98          NonrecursiveBST<String, Integer> st = new NonrecursiveBST<String, Integer>();
99          for (int i = 0; i < N; i++)
100             st.put(a[i], i);
101         for (String s : st.keys())
102             StdOut.println(s + "␣" + st.get(s));
103     }
104
105 }
```

13.2 BST exercies

- check if a given binary tree is a BST
- Find the minimum key
- Find the maximum key
- Build a BST from a preOrder traversal