UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 12

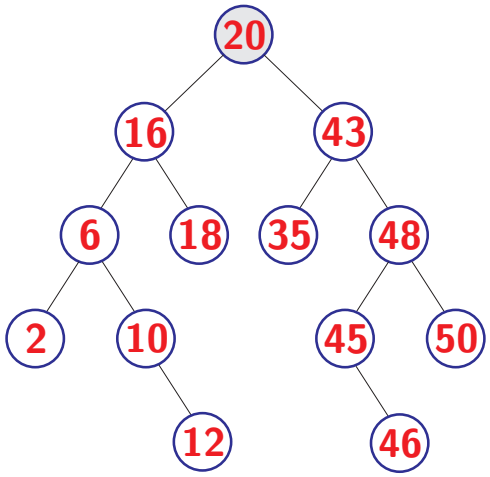
Dictionary Data Structure (Part 2)

Yoan Pinzón

Table of Content Session 12

Dictionary Data Structure
Binary Search Tree Representation

A Binary Search Tree (**BST**) is a binary tree where for every node v, all the elements in the left (right) subtree of v are smaller (larger) than v.



An in-order traversal produces the elements in sorted order: 2, 6, 10, 12, 16, 18, 20, 35, 43, 45, 46, 48, 50.

Searching for an Element with Key k

- Start at root. If k equals the key of the root, then stop.
- If the key is less (greater) than the key of the root, then search the left (right) subtree.

Time Complexity: $O(h)$

Inserting an Element with Key k

- Search if there is an element with the same key.
- If the search is unsuccessful, then insert the element at the point where the search was terminated.

Time Complexity: $O(h)$

Deleting an Element with Key k

- Search if there is an element e with the same key.
- If the search is successful, consider:
  - e is a leaf ⇒ delete this node.
  - e has exactly one nonempty subtree ⇒ If e is the root, then it is deleted and the root of its single subtree becomes the new root. Otherwise, change the pointer from the parent $p(e)$ of e so that it points to the e's only child, and delete e.
  - e has exactly two nonempty subtrees ⇒ replace this element either with the largest element in its left subtree, or with the smallest element in its right subtree. To find the largest (smallest) element in the left (right) subtree simply follow the right-child (left-child) pointers from the root of the subtree until a node with null right-child (left-child) pointer is reached.

Time Complexity: $O(h)$

Class Definition of BST

```
package unal.datastructures;

import java.util.*;

public class BST<K extends Comparable<? super K>, E> extends ↙
  ↳ LinkedBinaryTree<DataDict<K, E>> implements Dictionary<K, E>
{
   // constructor
   public BST( ) { /* ... */ }

   // methods
   public E get( K theKey ) { /* ... */ }
   public E put( K theKey, E theElement ) { /* ... */ }
   public E remove( K theKey ) { /* ... */ }
   public void ascend( ) { /* ... */ }
   public static void main( String[] args ) { /* ... */ }
}
```


DataDict class

```
5  class DataDict<K extends Comparable<? super K>, E>
6  {
7     // fields
8     K key;     // its key
9     E element; // element in node

11    // constructor
12    DataDict( )
13    {
14       key = null;
15       element = null;
16    }

18    DataDict( K theKey, E theElement )
19    {
20       key = theKey;
21       element = theElement;
22    }
```

```
24    @Override
25    public String toString( )
26    {
27       return "[" + Objects.toString( element ) +
28          ",␣key=" + Objects.toString( key ) + "]";
29    }
30 }
```


constructor

```
10    public BST( )
11    {
12       super( );
13    }
```

get

```
17  /** @return element with specified key
18    * @return null if no matching element */
19  public E get( K theKey )
20  {
21     // pointer p starts at the root and moves through
22     // the tree looking for an element with key theKey
23     BinaryTreeNode<DataDict<K, E>> p = root;
24     while( p != null )
25        if( theKey.compareTo( p.element.key ) < 0 )
26           p = p.leftChild;
27        else
28           if( theKey.compareTo( p.element.key ) > 0 )
29              p = p.rightChild;
30           else // found matching element
31              return p.element.element;
32     // no matching element
33     return null;
34  }
```


put

```
36  /** insert an element with the specified key
37    * overwrite old element if there is already an
38    * element with the given key
39    * @return old element (if any) with key theKey */
40  public E put( K theKey, E theElement )
41  {
42     BinaryTreeNode<DataDict<K, E>> p = root, // search pointer
43                                    pp = null; // parent of p
44     // find place to insert theElement
45     while( p != null )
46     {  // examine p.element.key
47        pp = p;
48        // move p to a child
49        if( theKey.compareTo( p.element.key ) < 0 )
50           p = p.leftChild;
51        else if( theKey.compareTo( p.element.key ) > 0 )
52                p = p.rightChild;
53             else
54             {  // overwrite element with same key
```

```
55                E elementToReturn = p.element.element;
56                p.element.element = theElement;
57                return elementToReturn;
58             }
59       }

61       // get a node for theElement and attach to pp
62       BinaryTreeNode<DataDict<K, E>> r = new BinaryTreeNode<>
63          ( new DataDict<K, E>( theKey, theElement ) );
64       if( root != null )
65          // the tree is not empty
66          if( theKey.compareTo( pp.element.key ) < 0 )
67             pp.leftChild = r;
68          else
69             pp.rightChild = r;
70       else // insertion into empty tree
71          root = r;
72       return null;
73    }
```


remove

```
75    /** @return matching element and remove it
76      * @return null if no matching element */
77    public E remove( K theKey )
78    {
79       // set p to point to node with key searchKey
80       BinaryTreeNode<DataDict<K, E>> p = root, // search pointer
81                                      pp = null; // parent of p
82       while( p != null && !p.element.key.equals( theKey ) )
83       {  // move to a child of p
84          pp = p;
85          if( theKey.compareTo( p.element.key ) < 0 )
86             p = p.leftChild;
87          else
88             p = p.rightChild;
89       }

91       if( p == null ) // no element with key searchKey
92          return null;

94       // save element to be removed
```

```
95      E theElement = p.element.element;
97      // restructure tree
98      // handle case when p has two children
99      if( p.leftChild != null && p.rightChild != null )
100     {  // two children
101        // convert to zero or one child case
102        // find element with largest key in left subtree of p
103        BinaryTreeNode<DataDict<K, E>> s = p.leftChild,
104                                       ps = p; // parent of s
105        while( s.rightChild != null )
106        {  // move to larger element
107           ps = s;
108           s = s.rightChild;
109        }
111        // move largest element from s to p
112        p.element = s.element;
113        p = s;
114        pp = ps;
115     }
```



```
117     // p has at most one child, save this child in c
118     BinaryTreeNode<DataDict<K, E>> c;
119     if( p.leftChild == null )
120        c = p.rightChild;
121     else
122        c = p.leftChild;
124     // remove node p
125     if( p == root ) root = c;
126     else
127     {  // is p left or right child of pp?
128        if( p == pp.leftChild )
129           pp.leftChild = c;
130        else
131           pp.rightChild = c;
132     }
134     return theElement;
135  }
```

ascend

```
137   /** output elements in ascending order of key */
138   public void ascend()
139   {
140      inOrderOutput();
141   }
```


main

```
143   /** test program */
144   public static void main( String[] args )
145   {
146      BST<Integer, Character> y = new BST<>( );

148      // insert a few elements
149      y.put( new Integer( 1 ), new Character( 'a' ) );
150      y.put( new Integer( 6 ), new Character( 'c' ) );
151      y.put( new Integer( 4 ), new Character( 'b' ) );
152      y.put( new Integer( 8 ), new Character( 'd' ) );

154      System.out.println( "Elements␣in␣ascending␣order␣are" );
155      y.ascend( );
156      System.out.println( );

158      // remove an element
```

```
159       System.out.println( "Removed␣element␣" +
160          y.remove( new Integer( 4 ) ) + "␣with␣key␣4" );
161       System.out.println( "Elements␣in␣ascending␣order␣are");
162       y.ascend( );
163       System.out.println( );

165       // remove another element
166       System.out.println( "Removed␣element␣" +
167          y.remove( new Integer( 8 ) ) + "␣with␣key␣8" );
168       System.out.println( "Elements␣in␣ascending␣order␣are" );
169       y.ascend( );
170       System.out.println( );

172       // remove yet another element
173       System.out.println( "Removed␣element␣" +
174          y.remove( new Integer( 6 ) ) + "␣with␣key␣6" );
175       System.out.println( "Elements␣in␣ascending␣order␣are" );
176       y.ascend( );
```



```
177       System.out.println( );

179       // try to remove a nonexistent element
180       System.out.println( "Removed␣element␣" +
181          y.remove( new Integer( 6 ) ) + "␣with␣key␣6" );
182       System.out.println( "Elements␣in␣ascending␣order␣are" );
183       y.ascend( );
184       System.out.println( );
185    }
```

Compiling `BST.java`

```
C:\2016699\code> javac unal\datastructures\BST.java ↵
C:\2016699\code> java unal.datastructures.BST ↵
Elements in ascending order are
[a, key=1] [b, key=4] [c, key=6] [d, key=8]
Removed element b with key 4
Elements in ascending order are
[a, key=1] [c, key=6] [d, key=8]
Removed element d with key 8
Elements in ascending order are
[a, key=1] [c, key=6]
Removed element c with key 6
Elements in ascending order are
[a, key=1]
Removed element null with key 6
Elements in ascending order are
[a, key=1]
```