

Data Structures and Algorithms – Week 2 Code Samples

Lecturer: Arran Stewart

This zip file contains code samples we will refer to during week 2.

To understand the code samples well, you should make use of the textbook, which is *Data Structures and Problem Solving Using Java* (4th edition) by Mark Allen Weiss.

Stack abstract data type

This is the *interface* for the Stack abstract data type.

Refer to the Weiss textbook if you are not familiar with interfaces.

It is used in the problem sheet for this week.

```
1  /**
2   * Abstract data type (ADT) for a stack
3   * @author rachel cardell-oliver
4   * @version nov 2015
5   */
6  public interface StackADT<T> {
7
8      void push( T x );    // add
9      void pop( );        // remove
10     T top( );            // get value of top element
11     boolean isEmpty( ); // test for empty stack
12 }
```

Queue abstract data type

This is the *interface* for the Queue abstract data type.

```
1  /**
2   * Abstract data type (ADT) for a queue
3   * @author rachel cardell-oliver
4   * @version nov 2015
5   */
6  public interface QueueADT<T> {
7
8      public void enqueue(T a);    //add to the back
9      public T dequeue();          //remove from the front
10     public T examine();          //get value of top element
11     public boolean isEmpty();     //test for empty queue
12 }
```

Binary Tree

Implementation of a binary tree.

```
1 public class BinaryTreeNode<T> {
2
3     public T value;           // data value associated with a node
4     public BinaryTreeNode<T> left; // reference for the left child
5     public BinaryTreeNode<T> right; // reference for the right child
6
7     //see Weiss Fig 18.12 for a safer implementation
8     // in which these fields are hidden
9     // and getter methods and setter methods are used for access
10
11     /**
12      * Construct a binary tree node
13      * @param value data value of type T associated with this node
14      * @param lefttree reference to the left child
15      * @param righttree reference to the right child
16      */
17     public BinaryTreeNode( T value, BinaryTreeNode<T> lefttree,
18                           BinaryTreeNode<T> righttree ) {
19         this.value = value;
20         left = lefttree;
21         right = righttree;
22     }
23
24     /**
25      * traverse a binary tree in order and
26      * print out the value of each node visited
27      * @param root of the binary tree to start from
28      */
29     public void printPreOrder () {
30         if (this != null)
31             System.out.println(value.toString()); //visit root
32         if (left != null)
33             left.printPreOrder(); //visit left
34         if (right != null)
35             right.printPreOrder(); //visit right
36     }
37
38
39     public void printPostOrder () {
40         //TODO write your code here
41     }
42
43     public void printInOrder () {
44         //TODO write your code here
45     }
46 }
```

Binary Search Tree (BST)

Implementation of a binary search tree.

```
1  import CITS2200.DuplicateItem;
2  import CITS2200.ItemNotFound;
3
4  /**
5   * TODO binary search tree with insert and delete
6   * @author mark allen weiss with modifications by rachelcardell-oliver
7   * @version nov 2015
8   *
9   */
10 public class BinarySearchTree<T extends Comparable<? super T>> {
11
12     BinaryTreeNode<T> root;
13
14     public BinarySearchTree() {
15         root = null;
16     }
17
18     public boolean isEmpty() {
19         return ( root == null );
20     }
21
22     /**
23      * @return item that matches a or null if no match
24      */
25     private BinaryTreeNode<T> find(T a, BinaryTreeNode<T> t) {
26         if (t == null) { return null; }
27         int whatnext = t.value.compareTo(a);
28         if (whatnext == 0) {
29             return t;
30         } else if ( whatnext < 0 ) {
31             return ( find(a, t.left) );
32         } else { // whatnext > 0
33             return ( find(a, t.right) );
34         }
35     }
36
37     /**
38      * public find a in main tree
39      * @param a
40      */
41     public BinaryTreeNode<T> find(T a) {
42         return find(a,root);
43     }
44
45     /**
46      * internal method to find the minimum element in
47      * @param t subtree to be searched
48      * @return reference to the smallest element
49      */
50     private BinaryTreeNode<T> findMin( BinaryTreeNode<T> t ) {
51         if( t != null )
52             while( t.left != null ) t = t.left;
53         return t;
54     }
55
56
57     /**
```

```

58     * Insert is fairly straightforward
59     * perform a search for the element as in find
60     * if the element is found, report duplicate item
61     * if an empty node is reached, insert a new node containing the element
62     */
63     public BinaryTreeNode<T> insert(T a, BinaryTreeNode<T> t) {
64         if (t == null) {
65             t = new BinaryTreeNode<T>(a,null,null);
66         } else if ( t.value.compareTo(a) < 0 ) {
67             t.left = insert(a,t.left);
68         } else if ( t.value.compareTo(a) > 0 ) {
69             t.right = insert(a,t.right);
70         } else { //tried to insert duplicate
71             throw new DuplicateItem( a.toString() );
72         }
73         return t;
74     }
75
76     /**
77     * Standard insert into the main tree
78     * @param a value to insert
79     */
80     public void insert(T a) {
81         root = insert(a,root);
82     }
83
84     /**
85     * internal method to remove minimum element from a subtree
86     * @param t the root of the subtree
87     * @return
88     */
89     private BinaryTreeNode<T> removeMin(BinaryTreeNode<T> t) {
90         if (t==null) {
91             throw new ItemNotFound("Min not found");
92         } else if (t.left != null) {
93             t.left = removeMin( t.left );
94             return t;
95         } else {
96             return t.right;
97         }
98     }
99
100    /**
101    * remove element a from the tree
102    * @param a
103    * @param t
104    * @return the altered tree t
105    * @throws ItemNotFoundException
106    */
107    public BinaryTreeNode<T> remove(T a, BinaryTreeNode<T> t) {
108        //if element found at a node with at least one external child
109        //then standard delete
110        if( t == null )
111            throw new ItemNotFound( a.toString() );
112        if ( a.compareTo( t.value ) < 0 )
113            t.left = remove( a, t.left );
114        else if ( a.compareTo( t.value ) > 0 )
115            t.right = remove( a, t.right );
116        //if we get here we've found a - now remove it
117        else if ( t.left != null && t.right != null ) {
118            //case where there are two children
119            //1. replace the deleted element with its predecessor

```

```

120         //note that the predecessor will always have an empty right child
121         //2. delete the predecessor
122         t.value = findMin( t.right ).value;
123         t.right = removeMin( t.right );
124     } else {
125         if ( t.left != null ) {
126             t = t.left;
127         } else {
128             t = t.right;
129         }
130     }
131     return t;
132 }
133
134 /**
135  * Delete an item from the main tree
136  * @param a value to delete
137  */
138 public void remove(T a) {
139     root = remove(a,root);
140 }
141
142 }

```