

**ECU178 Computer Science:
210CT - Programming, Algorithms
and Data Structures
Coursework**

Due on March 20th 2015

Robert Rigler : 4939377

Contents

Week 10: BinTreeNode class Implementation	3
Week 14: Balanced Tree Research	9
Red-Black Trees	9
B - Trees	9
Week 15: Graphs	10
Task 1: Pseudocode	10
Week 16: Graph Search	11
Week 17: Lambda Functions	12
Tasks 1 & 2	12
Tasks 3	15

Week 10: BinTreeNode class Implementation

Listing 1: Commented Implementatino of BinTreeNode in C#

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Tree
8 {
9     class BinTreeNode
10     {
11         //Variables
12         int key;
13         BinTreeNode parent;
14         BinTreeNode left;
15         BinTreeNode right;
16
17         BinTreeNode(int value) // Constructor
18         {
19             this.key = value;
20             this.left = null;
21             this.right = null;
22
23         }
24
25         /**
26          * In_Order_Walk prints the contents of the tree in sorted order.
27          *
28          * @param BinTreeNode $x
29          * @return void
30          */
31         void In_Order_Walk(BinTreeNode x) {
32
33             if(x != null){
34                 In_Order_Walk(x.left);
35                 Console.Write(x.key);
36                 In_Order_Walk(x.right);
37             }
38
39         }
40     }
41
42
43
44
45
46
47
48
49
50
```

```
51
52     /**
53      * Post_Order_Walk prints the tree in Post Order:
54      * Which prints the root AFTER the values in its subtrees
55      *
56      */
57
58     void Post_Order_walk(BinTreeNode x) {
59         if (x!=null) {
60             Post_Order_walk(x.left);
61             Post_Order_walk(x.right);
62             Console.Write(x.key);
63         }
64     }
65
66     /**
67      * Pre_Order_Walk prints the tree in Pre Order:
68      * which prints the root before the value in its subtrees
69      */
70
71     void Pre_Order_walk(BinTreeNode x)
72     {
73         if (x != null)
74         {
75             Console.Write(x.key);
76             Pre_Order_walk(x.left);
77             Pre_Order_walk(x.right);
78         }
79     }
80
81 }
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
```

```
104
105
106     /**
107      * Tree_Search searches the given tree for a specifk integer.
108      *
109      * @param BinTreeNode x
110      * @param integer k
111      * @return BinTreeNode
112      *
113      * **/
114
115     BinTreeNode Tree_Search(BinTreeNode x, int k)
116     {
117         while(x!=null && k != x.key){
118             if (k < x.key)
119                 x = x.left;
120             else
121                 x = x.right;
122         }
123         return x;
124     }
125
126     /**
127      * Get_Minimum returns the minimum (leftmost) child of a given node or tree.
128      *
129      * @param BinTreeNode x
130      * @return BinTreeNode
131      */
132     BinTreeNode Get_Minimum(BinTreeNode x)
133     {
134         while (x.left != null)
135         {
136             x = x.left;
137         }
138         return x;
139     }
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156     /**
```

```
157      * Get_Minimum returns the maximum (rightmost) child of a given node or tree.
158      *
159      * @param BinTreeNode x
160      * @return BinTreeNode
161      */
162      BinTreeNode Get_Maximum(BinTreeNode x)
163      {
164          while (x.right != null)
165          {
166              x = x.right;
167          }
168          return x;
169      }
170
171
172      /**
173      * Get_Next returns the in_order successor to a given node
174      *
175      * @param BinTreeNode x
176      * @return BinTreeNode
177      */
178      BinTreeNode Get_Next(BinTreeNode x)
179      {
180          //If a node has a right child;
181          //then the in_order previous node must be the leftmost node of that
182          //subtree
183          if (x.right != null)
184              return Get_Minimum(x.right);
185
186          //If node does not have right child;
187          // Traverse up the tree to the first node on the right
188          BinTreeNode y = x.parent;
189          while (y != null && x == y.right)
190          {
191              x = y;
192              y = y.parent;
193          }
194          return y;
195      }
196
197
198
199
200
201
202
203
204
205
206
207
208
209
```

```
210      /**
211      * Get_previous returns the in_order predecessor to a given node
212      *
213      * @param BinTreeNode x
214      * @return BinTreeNode
215      */
216      BinTreeNode Get_Previous(BinTreeNode x)
217      {
218          //If a node has a left child;
219          //then the in_order previous node must be the rightmost node of that
220          //subtree
221          if (x.left != null)
222              return Get_Maximum(x.left);
223
224          //if the node does not have a left child;
225          // then Traversal up the tree to the first node on the left.
226          BinTreeNode y = x.parent;
227          while (y != null && x == y.left)
228          {
229              x = y;
230              y = y.parent;
231          }
232          return y;
233      }
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
```

```
263      /**
264       * # Tree_Insert inserts a new node (z) into the correct position in the
265       *     tree (t).
266       * Begin checking at the root of the tree.
267       * x traverses the tree looking for the correct null position.
268       * (x moves left or right depending on the comparison).
269       * y stores the parent of the null position.
270       * set pointers
271       *
272       * @param BinTreeNode t
273       * @param BinTreeNode z
274       * @return void
275       */
276     void Tree_Insert(BinTreeNode t, BinTreeNode z)
277     {
278
279         BinTreeNode y = null;
280         BinTreeNode x = t;
281
282         while (x != null)
283         {
284             y = x;
285             if (z.key < x.key)
286                 x = x.left;
287             else
288                 x = x.right;
289         }
290
291         z.parent = y;
292         if (y == null)
293             t = z;
294         else if (z.key < y.key)
295             y.left = z;
296         else y.right = z;
297
298     }
299
300 }
301
302 }
```


Week 14: Balanced Tree Research

A balanced tree is one in which both left and right sub-trees are of a similar height. In Binary trees the insertion, lookup and deletion performance is $O(h)$ time, where h is the height of the sub-tree. By balancing the tree, making sure that both sub-trees are of a similar height, it is guaranteed that the performance of basic operations is $O(\log n)$ time in the worst case scenario.

There are many types of balanced trees:

- Red-Black trees,
- AVL Trees,
- 2-3-4 Trees,
- B-Trees,
- etc.

For this particular task, I will be comparing Red-Black Trees and B - Trees to AVL Trees.

Red-Black Trees

Red-Black Trees, as well as storing the *value*, *left node* and *right node* attributes, it also stores the color of the nodes. It ensures the tree is balanced by.

AVL trees are more balanced in comparison to Red-Black Trees, However AVL trees take longer to insert and delete nodes from the tree because they do more rotations per operation.

In contrast, AVL Trees, being more strictly balanced than Red-Black trees, make lookup(Search) operations much faster than in a Red-Black Tree.

So In application, Red-Black trees are much more suited for programs which do more insertion and deletion than search operations. For example Linux uses Red-Black Trees for process scheduling instead of a priority queue.

AVL Trees are more suited for programs which do a lot of lookup operations.

B - Trees

A B-Tree is another type of balanced tree. It differs from AVL Trees and Red-Black Trees because it can have a variable number of keys and children per parent. B-Trees are typically used in scenarios where there is a large amount of data that needs to be quickly accessed. For example, B-Trees are often used when the nodes are stored on a physical disk. This is because it takes much longer to access data on a disk than from RAM, so by using many nodes with many branches, the speed of accesses is greatly improved. B-Trees allow a desired node to be located faster, but the decision process at each branch is more complicated.

Week 15: Graphs

Task 1: Pseudocode

This task asked me to: "Write the pseudocode for an unweighted graph data structure. You either use an adjacency matrix or an adjacency list approach. Also write a function to add a new node and a function to add an edge".

I will be using the adjacency list approach. To complete this task I will create two data structures: Graph and GraphNode. *Graph* will contain the list of vertices in the graph and also control the operation for adding new vertices and edges. *GraphNode* will represent the individual nodes in the graph. Each *GraphNode* will contain a list of its neighbours (connected edges).

Algorithm 1 GraphNode

```
1: Class GraphNode
2:     int data;
3:     List Edges[];
```

Algorithm 2 Graph

```
1: Class Graph
2:     List Vertices[]
```

Algorithm 3 Graph AddVertex

```
1: procedure ADDVERTICE(int  $u$ )
2:
3:     vertices.add(new GraphNode( $u$ ))
4: end procedure
```

Algorithm 4 Graph AddVertice

```
1: procedure ADDVERTICE(int  $u$ )  
2:  
3:   vertices.add(new GraphNode( $u$ ))  
4: end procedure
```

Week 16: Graph Search

Week 17: Lambda Functions

In this task, I had to:

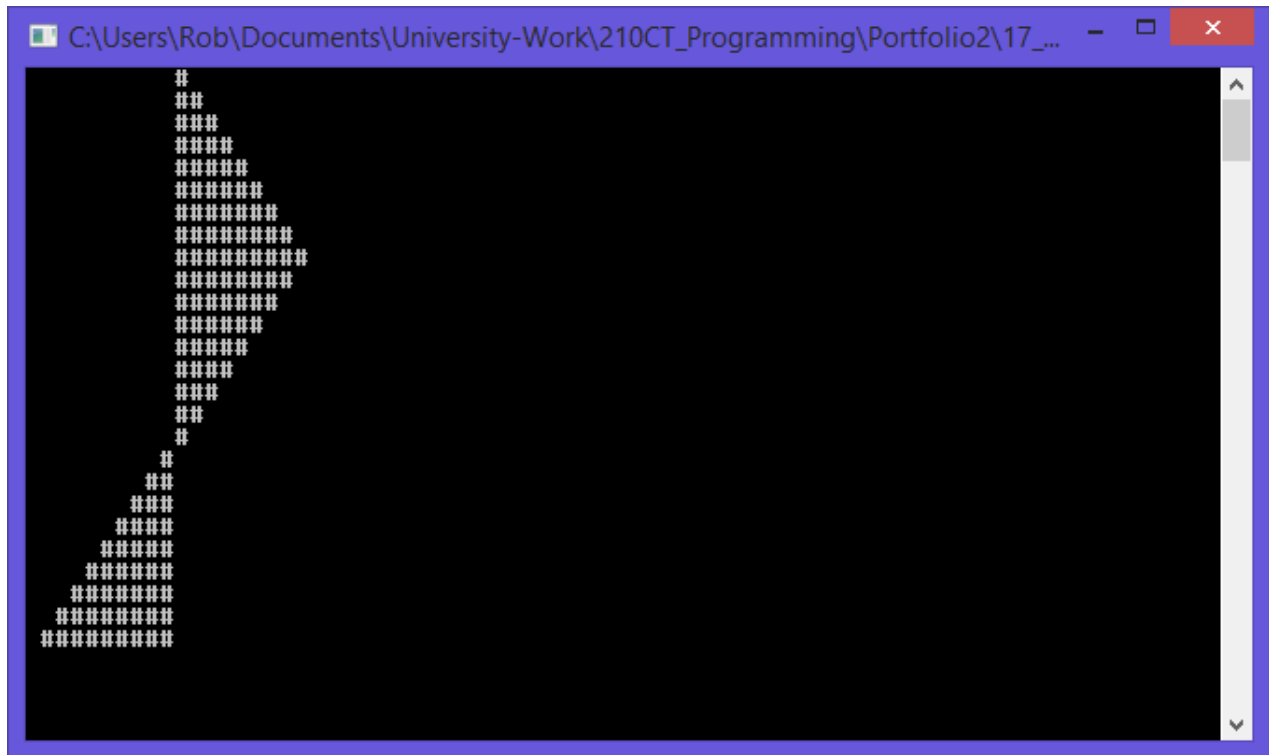
1. In either Python or C++, write a class that stores a series of numbers. The length of the sequence can be set in the constructor. All numbers will be floats in the range -1 to 1 (inclusive).
2. Add a function to display the sequence visually. This will be done by drawing it as a waveform, vertically.
3. Add a function that takes a functor/lambda that can be used to replace every value with another based on the original value. For example, all values such that $\text{abs}(x) > 0.5$ become -1 or 1 depending on whether they were originally negative or positive.

Tasks 1 & 2

Below is the commented code and a screenshot of the working program.

Listing 2: Wave.cpp Version 1

```
1  // Wave.cpp : Defines the entry point for the console application.
2  //
3
4  #include "stdafx.h"
5  #include <stdlib.h>
6  #include <iostream>
7  #include <vector>
8  #include <algorithm>
9  #include <cmath>
10
11
12
13 void dWave(std::vector<double> v) {
14
15     //declaring control Variables for the loop.
16
17     double j = 0;
18     int y;
19     int z;
20
21     /* If the element is negative; j will be less than 10
22        and the inner loop will iterate 10 times, printing spaces while less than j.
23
24        If the element is positive, j will be more than 10,
25        the inner loop will iterate j times, printing spaces while less than 10.
26
27     */
28
29     for (int i = 0; i < v.size() ; i++){
30
31         j = 10 + (v[i] * 10 );
32
33         if (j > 10) y = j, z = 10;
34
35         else y = 10, z = j;
36
37         for (int x = 0; x <= y; x++){
38             if (x < z)
39                 std::cout << " ";
40             else
41                 std::cout << "#";
42         }
43         std::cout << "\n";
44
45     }
46
47 }
```



Tasks 3

Listing 3: Wave.cpp Version 2

```
1  // Wave.cpp : Defines the entry point for the console application.
2  //
3  #include "stdafx.h"
4  #include <stdlib.h>
5  #include <iostream>
6  #include <vector>
7  #include <algorithm>
8  #include <cmath>
9
10     /* Functor 'goround':
11     *
12     * transforms a double in range -1 <= x <= 1
13     * to either -1 or 1 depending if abs(x) < 0.5 respectively.
14     *
15     */
16 struct rounder{
17
18     int operator() (double x){ if (abs(x) > 0.5) return 1;
19                               else return -1; }
20 } goround ;
21
22
23 void dWave(std::vector<double> v){
24
25     //declaring control Variables for the loop.
26     double j = 0;
27     int y;
28     int z;
29
30     /* If the element is negative; j will be less than 10
31     and the inner loop will iterate 10 times, printing spaces while less than j.
32
33     Iff the element is positive, j will be more than 10,
34     the inner loop will iterate j times, printing spaces while less than 10.
35
36     */
37
38
39
40
41
42
43
44
45
46
47
48
49
50
```

```

51  /* STD::TRANSFORM,
52  *
53  *      Uses std::transform to transform all elements in :
54  *      std::vector<double> v TO their goround equivalent (either -1 or 1)
55  *
56  */
57
58
59  std::transform(v.begin(), v.end(), v.begin(), goround);
60  for (int i = 0; i < v.size() ; i++){
61
62      j = 10 + (v[i] * 10 );
63
64      if (j > 10) y = j, z = 10;
65
66      else y = 10, z = j;
67
68      for (int x = 0; x < y; x++){
69          if (x < z)
70              std::cout << " ";
71          else
72              std::cout << "#";
73      }
74      std::cout << "\n";
75
76  }
77
78  }

```

