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

Due on March 20th 2015

Robert Rigler : 4939377

Contents

Week 10: BinTreeNode class Implementation

Listing 1: Commented Implementatino of BinTreeNode in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tree
{
    class BinTreeNode
    {
        //Variables
        int key;
        BinTreeNode parent;
        BinTreeNode left;
        BinTreeNode right;

        BinTreeNode(int value) // Constructor
        {
            this.key = value;
            this.left = null;
            this.right = null;

        }

        /**
         * In_Order_Walk prints the contents of the tree in sorted order.
         *
         * @param BinTreeNode $x
         * @return void
          *
         **/
       void  In_Order_Walk(BinTreeNode x){

            if(x != null){
                In_Order_Walk(x.left);
                Console.Write(x.key);
                In_Order_Walk(x.right);
            }

        }
```

```
        /**
         * Post_Order_Walk prints the tree in Post Order:
         *  Which prints the root AFTER the values in its subtrees
         *
         */

        void Post_Order_walk(BinTreeNode x){
            if (x!=null){
                Post_Order_walk(x.left);
                Post_Order_walk(x.right);
                Console.Write(x.key);
            }

        }
        /**
         * Pre_Order_Walk prints the tree in Pre Order:
         *  which prints the root before the value in its subtrees
         */

        void Pre_Order_walk(BinTreeNode x)
        {
            if (x != null)
            {
                Console.Write(x.key);
                Pre_Order_walk(x.left);
                Pre_Order_walk(x.right);

            }

        }
```

 continued on next page...

```java


    /**
     *  Tree_Search searches the given tree for a specifik integer.
     *
     *  @param BinTreeNode x
     *  @param integer k
     *  @return BinTreeNode
     *
     * **/

    BinTreeNode Tree_Search(BinTreeNode x, int k)
    {
        while(x!=null && k != x.key){
            if (k < x.key)
                x = x.left;
            else
                x = x.right;
        }
        return x;
    }

    /**
     * Get_Minimum returns the minimum (leftmost) child of a given node or tree.
     *
     * @param BinTreeNode x
     * @return BinTreeNode
     */
    BinTreeNode Get_Minimum(BinTreeNode x)
    {
        while (x.left != null)
        {
            x = x.left;
        }
        return x;

    }
















    /**
```

```
        * Get_Minimum returns the maximum (rightmost) child of a given node or tree.
        *
        * @param BinTreeNode x
        * @return BinTreeNode
        */
        BinTreeNode Get_Maximum(BinTreeNode x)
        {
            while (x.right != null)
            {
                x = x.right;
            }
            return x;

        }

        /**
        * Get_Next returns the in_order successor to a given node
        *
        * @param BinTreeNode x
        * @return BinTreeNode
        */
        BinTreeNode Get_Next(BinTreeNode x)
        {
            //If a node has a right child;
            //then the in_order previous node must be the leftmost node of that
         //subtree
            if (x.right != null)
                return Get_Minimum(x.right);

            //If node does not have right child;
            // Traverse up the tree to the first node on the right
            BinTreeNode y = x.parent;
            while (y != null && x == y.right)
            {
                x = y;
                y = y.parent;
            }
            return y;
        }
```

```
        /**
        * Get_previous returns the in_order predecessor to a given node
        *
        * @param BinTreeNode x
        * @return BinTreeNode
        */
        BinTreeNode Get_Previous(BinTreeNode x)
        {
            //If a node has a left child;
            //then the in_order previous node must be the rightmost node of that
         //subtree
            if (x.left != null)
                return Get_Maximum(x.left);

            //if the node does not have a left child;
            // then Traverese up the tree to the first node on the left.
            BinTreeNode y = x.parent;
            while (y != null && x == y.left)
            {
                x = y;
                y = y.parent;
            }
            return y;
        }
```

```
        /**
         * # Tree_Insert inserts a new node (z) into the correct position in the
         *          tree (t).
         *  Begin checking at the root of the tree.
         *  x traverses the tree looking for the correct null position.
         *  (x moves left or right depending on the comparison).
         *  y stores the parent of the null position.
         *  set pointers
         *
         * @param BinTreeNode t
         * @param BinTreeNode z
         * @return void
         */
        void Tree_Insert(BinTreeNode t, BinTreeNode z)
        {

            BinTreeNode y = null;
            BinTreeNode x = t;

            while (x != null)
            {
                y = x;
                if (z.key < x.key)
                    x = x.left;
                else
                    x = x.right;
            }

            z.parent = y;
            if (y == null)
                t = z;
            else if (z.key < y.key)
                y.left = z;
            else y.right = z;

        }

    }

}
```

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 that 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".

In 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 *GraohNode* will contain a list of its neighbours (connected edges).

Algorithm 1 GraphNode

```
Class GraphNode
        int data;
        List Edges[];
```

Algorithm 2 Graph

```
Class Graph
        List Vertices[]
```

Algorithm 3 Graph AddVertice

```
procedure AddVertice(int u)

    vertices.add(new GraphNode(u))
end procedure
```

Algorithm 4 Graph AddVertice

```
procedure AddVertice(int u)

    vertices.add(new GraphNode(u))
end procedure
```

Week 16: Graph Search

Week 17: Lambda Functions

In this task, I had to:

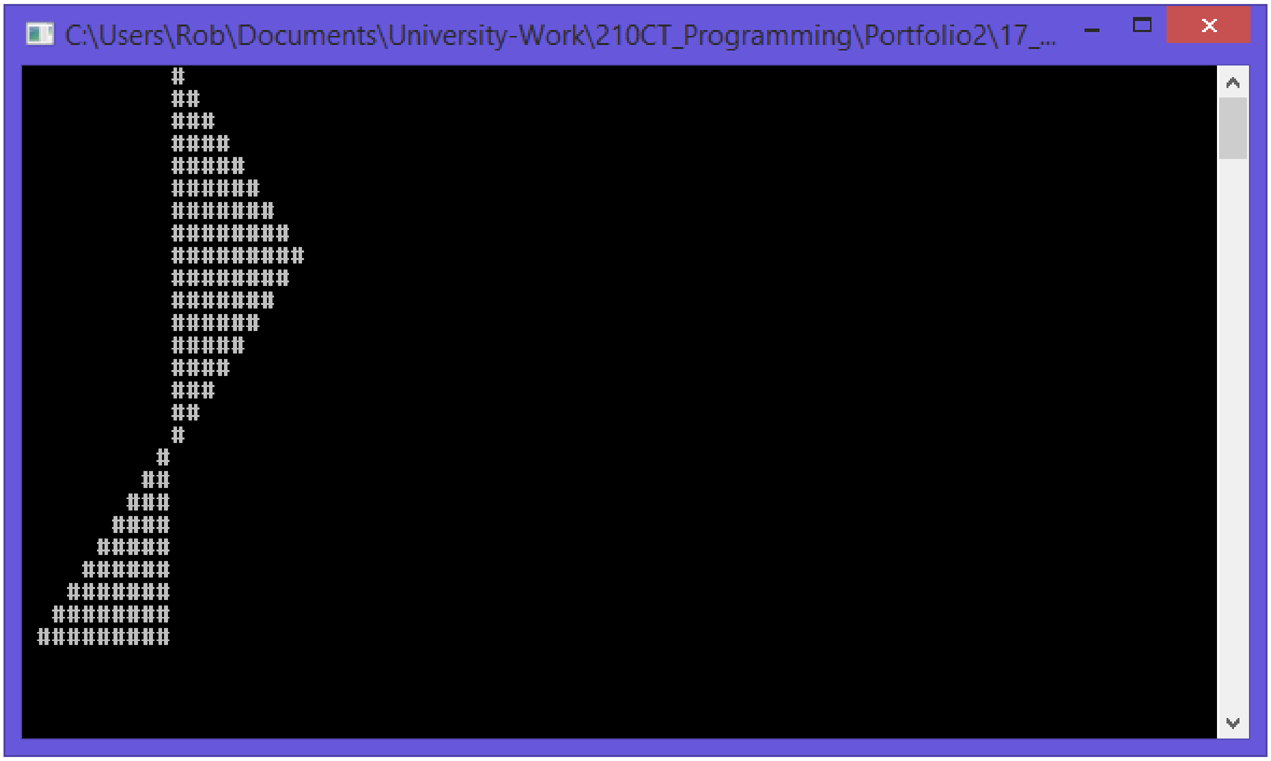
1. In either Python or C++, write a class that stores a series of numbers. The length of the sequence can be set in the constructor. All numbers will be floats in the range -1 to 1 (inclusive).
2. Add a function to display the sequence visually. This will be done by drawing it as a waveform, vertically.
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 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

```
// Wave.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>



void dWave(std::vector<double> v){

     //declaring control Variables for the loop.

     double j = 0;
     int y;
     int z;

/* If the element is negative; j will be less that 10
     and the inner loop will iterate 10 times, printing spaces while less than j.

     If the element is positive, j will be more than 10,
     the inner loop will iterate j times, printing spaces while less than 10.

*/

     for (int i = 0; i < v.size() ; i++){

          j = 10 + (v[i] * 10 );

          if (j > 10) y = j, z = 10;

          else y = 10, z = j;

          for (int x = 0; x <= y; x++){
               if (x < z)
                    std::cout << " ";
               else
                    std::cout << "#";
          }
          std::cout << "\n";

     }

     }
```



 continued on next page. . .

Tasks 3

Listing 3: Wave.cpp Version 2

```
// Wave.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

     /* Functor 'goround':
     *
     * transforms a double in range -1 <= x >= 1
     * to either -1 or 1 depending if abs(x) < 0.5 respectively.
     *
     */
struct rounder{

     int operator() (double x){ if (abs(x) > 0.5) return 1;
                                            else return -1; }
} goround ;


void dWave(std::vector<double> v){

     //declaring control Variables for the loop.
     double j = 0;
     int y;
     int z;

     /* If the element is negative; j will be less that 10
     and the inner loop will iterate 10 times, printing spaces while less than j.

     Iff the element is positive, j will be more than 10,
     the inner loop will iterate j times, printing spaces while less than 10.

     */
```

```
    /* STD::TRANSFORM,
    *
    *    Uses std:transform to transform all elements in :
    *    std::vector<double> v  TO their goround equivalent (either -1 or 1)
    *
    */


    std::transform(v.begin(), v.end(), v.begin(), goround);
    for (int i = 0; i < v.size() ; i++){

        j = 10 + (v[i] * 10 );

        if (j > 10) y = j, z = 10;

        else y = 10, z = j;

        for (int x = 0; x < y; x++){
            if (x < z)
                std::cout << " ";
            else
                std::cout << "#";
        }
        std::cout << "\n";

    }

    }
```