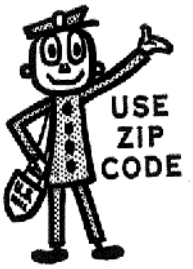
**CS 302 – Assignment #05**

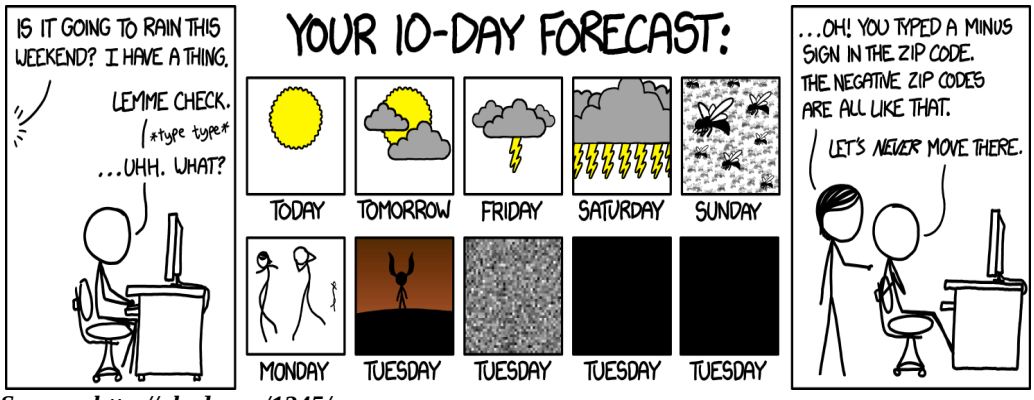
Purpose: Learn concepts regarding balanced binary trees.
Due: Tuesday (2/28) → Must be submitted on-line before class.
Points: Part A → 100 pts, Part B → 50 pts

**Assignment:**




**Part A:**
In order to qualify for the lower bulk mail rates, the addresses must be certified as being "correct" (i.e., at least ≥ 97% valid). For our purposes we will simplify the process and focus just on ensuring that the 5-digit ZIP Codes[1] are legal codes. Specifically, we will create a program to validate the ZIP codes for a file of addresses. To store the master set of valid zip codes, you will need to implement a Red-Black Tree[2] data structure. A test main will be provided that performs a series of tests using Red-Black tree with different data types. A main will be provided that performs the zip code validation process (using your objects).

**Part B:**
When completed, create and submit a write-up (PDF format) not too exceed 500 words including the following:



***Source: http://xkcd.com/1245/***

- Name, Assignment, Section
- Summary of the Red-Black tree data structure.
- Compare using an Red-Black tree to a binary search tree.
  - Include advantages and disadvantages of each implementation approach.
- Big-O for the various Red-Black tree operations.

**Visualization**
The following web site provides a visualization for Red-Black Trees, including the rotate operations.

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

**Submission:**

- Part A → Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.
- Part B → A copy of the write-up including the chart. Must use PDF format.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: https://en.wikipedia.org/wiki/ZIP_Code
2 For more information, refer to: http://en.wikipedia.org/wiki/Red-black_tree

## Class Descriptions

- Red-Black Tree Class
  The Red-Black Tree template stack class will implement functions specified below. We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType             keyValue;
      nodeColor          color;
      nodeType<myType>   *left;
      nodeType<myType>   *right;
      nodeType<myType>   *parent;  };
```

  Additionally, include the following enumeration definitions:

```
enum  nodeColor { RED, BLACK };
```

| **redBlackTree<myType>** |
|---|
| -nodeType<myType> *root |
| +redBlackTree() |
| +~redBlackTree() |
| +destroyTree(): void |
| +countNodes() const: unsigned int |
| +height() const: unsigned int |
| +printTree() const: void |
| +insert(myType): void |
| +search(myType): bool |
| -search(myType, nodeType<myType> *): bool |
| -destroyTree(nodeType<myType> *): void |
| -countNodes(nodeType<myType> *) const: unsigned int |
| -height(nodeType<myType> *) const: unsigned int |
| -printTree(nodeType<myType> *) const: void |
| -rightRotate(nodeType<myType> *): void |
| -leftRotate(nodeType<myType> *): void |

## Function Descriptions

- The *redBlackTree()* constructor should initialize the tree class variables to an empty state.
- The *~redBlackTree()* destructor should delete the tree by calling the private *destroyTree()* function.
- The public *destroyTree()* function should delete the tree by calling the private *destroyTree()* function.
- The private *destroyTree()* function should recursively delete the tree (including releasing all the allocated memory). Must be recursive.
- The public *countNodes()* function should return the total count of nodes in the tree by calling the private *countNodes()* function.
- The private *countNodes()* function should recursively return the total count of nodes in the tree. Must be recursive.
- The public *height()* function should return the maximum height of the tree by calling the private *height()* function.
- The private *height()* function returns maximum height of the tree. Must be recursive.

- The public *printTree()* function should call the private *printTree()* function to print the tree in the order passed.
- The private *printTree()* function should recursively print the tree in pre-order format. Must be recursive. Refer to the example output for formatting.
- The public *search()* function should call the private search function for the passed item, returning true if found and false if not.
- The private *search()* function should recursively search the tree, returning true if found and false if not. Must be recursive.
- The public *insert()* function should check to see if the passed key value is already in the tree using the *search()* function. If the passed key value is already in the tree, the function should do nothing. If the key value is not already in the tree, it should be inserted. If the tree becomes unbalanced, the appropriate rotations must be performed using the *leftRotate()* and/or *rightRotate()* functions.
- The private *rightRotate()* function should perform a right tree rotate operation (as described in class, in the text, and in the lecture notes).
- The public *leftRotate()* function should perform a left tree rotate operation (as described in class, in the text, and in the lecture notes).

- Zip Codes Class
  The zip codes class should inherit from the ***redBackTree*** class and will implement functions specified below.

| **ZipCodes public redBlackTree<string>** |
|---|
| -goodCount: int |
| -badCount: int |
| -masterZipCodes: redBlackTree<string> |
| +zipCodes() |
| +~zipCodes() |
| +readMasterZipCodes(const string): bool |
| +checkZips(const string): bool |
| +showStats() const: void |

**Function Descriptions**

- The *zipCodes()* constructor should initialize the class variables.
- The *~zipCodes()* destructor should reset the class variables.
- The *readMasterZipCodes()* function should read the passed master zip codes file and *insert()* the words in the Red-Black tree. If the master zip codes file read operations are successful, the function should return true and false otherwise.
- The *checkZips()* function should read the passed zip codes data file, parse out the zip code and see if the zip code is in the red-black tree. If so, the good counter should be incremented and if not, the bad counter should be incremented. If the data file read operations are successful, the function should return true and false otherwise.
- The *showStats()* function should display the statistics (in the format shown in the examples) including the calculation of the percentage good and percentage bad. If the percentage good is ≥ 97%, the message "Validation – PASSED" should be displayed otherwise the message "Validation – FAILED" should be displayed. Refer to the example output for formatting.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

**Red-Black Tree Insertion Algorithm**

*Note*, you must use this approach. Do not use other algorithms from the net.
Code copied from the net will result in a zero and formal academic misconduct write-up.

- red-black tree insertion algorithm
  - if node already in tree
    - exit routine
  - create new node, ***x***
    - set left, right, and parent pointers to NULL
    - set node key value to passed key value
    - set color to to RED
  - if root is NULL
    - set root = new ***x*** node
    - set ***x*** color to BLACK
    - exit routine
  - insert via standard binary search tree manner
    - initialize ***curr*** pointer to root
    - walk ***curr*** down from root following appropriate branch to find correct leaf
    - set pointers on appropriate leaf node, ***curr***, to new ***x*** node
    - set parent pointer for new ***x*** node to ***curr***
    - mark new ***x*** node as RED
  - while ***x*** is not root and color of ***x***'s parent is RED

    *note*, uncle's are: parent→parent→left OR parent→parent→right
    - if ***x***'s parent == ***x***'s left uncle (parent→parent→left)
      - if ***x***'s right uncle (parent→parent→right) exists and uncle is RED
        - change color of parent to BLACK
        - change color of uncle to BLACK
        - change color of grand parent (***x***→parent→parent) to RED
        - set ***x*** = ***x***'s grand parent (***x***→parent→parent)

          *note*, this moves the ***x*** pointer up the tree and leaves new node as is
      - else
        - if ***x*** == ***x*** parent right
          - set ***x*** = ***x*** parent
          - left rotate on ***x***
        - set ***x*** parent color to BLACK
        - set ***x*** grand parent (***x***→parent→parent) color to RED
        - right rotate on ***x*** grand parent (***x***→parent→parent)
    - else
      - if ***x***'s left uncle exists and uncle is RED
        - change color of parent to BLACK
        - change color of uncle to BLACK
        - change color of grand parent (***x***→parent→parent) to RED
        - set ***x*** = ***x***'s grand parent (***x***→parent→parent)

          *note*, this moves the ***x*** pointer up the tree and leaves new node as is
      - else
        - if ***x*** == ***x*** parent left
          - set ***x*** = ***x*** parent
          - right rotate on ***x***
        - set ***x*** parent color to BLACK
        - set ***x*** grand parent (***x***→parent→parent) color to RED
        - left rotate on ***x*** grand parent (***x***→parent→parent)
    - if ***x*** is root, set root color to BLACK

**Example Execution:**

Below is an example program execution for the tree test program.
*Note*, the **`ed-vm%`** is the prompt on my machine.

```
ed-vm%
ed-vm% ./treeTest
-----------------------------------------------------------------
CS 302 - Assignment #5

-----------------------------------------------------------------
Test Set #0
       Nodes:  7
       Height: 3

Pre-order traversal:
10 6 5 8 14 11 18

       deleting tree...

-----------------------------------------------------------------
Test Set #1
       Nodes:  24
       Height: 7

Pre-order traversal:
8 4 2 1 3 6 5 7 12 10 9 11 16 14 13 15 20 18 17 19 22 21 23 24

-----------------------------------------------------------------
Test Set #2
       Nodes:  15
       Height: 5

Pre-order traversal:
44 28 11 3 17 21 32 29 88 65 54 80 76 82 97

-----------------------------------------------------------------
Test Set #3
       Nodes:  50000
       Height: 29

       too much to print...
       deleting tree...

       Nodes:  0
       Height: 0

-----------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
```

Below is an example program execution for the check zips program.

```
ed-vm%
ed-vm% ./checkZips -z free-zipcode-database-Primary.csv -d addrsSM.csv
****************************************************************
CS 302 - Assignment #5
ZIP Code Checking Program

--------------------------------------
Zip Code Validation Statistics:

Master Zip Codes Tree Statistics:
   Tree Entries: 42522
   Tree Height: 24

Zip Codes Data File Statistics:
   Total Zip Codes: 500
     Good: 488
     Bad: 12
   Percentage Good: 97.6
   Percentage Bad: 2.4
   Validation - PASSED

****************************************************************
Game Over, thank you for playing.
ed-vm%
```