**ADVANCED DATA STRUCTURES**

**COP 5536**

**RED – BLACK TREE**

**PROGRAMMING PROJECT**

**REPORT**

Submitted To: Dr. Sartaj Sahini

**ANJALI VERMA**

**UFID: 04862395**

**Mail:** anjaliverma661@ufl.edu

Submission Date: 22nd March, 2016

**PROGRAM STRUCTURE**

Class : **bbst**

* public static void **createTree**(File f)

Creates a Red Black by reading data from file ‘f’.

Complexity=

Each insertion takes log(size of tree)

* public static int **delete**(node z)

Deletes a node and then calls on transplant(node u, node v) to replace the subtree as a child of the parent with another subtree.

Complexity of delete: O(log n)

* public static node **tree\_min**(node x)

Determines the minimum value for a subtree with root x

Complexity : O (log(size of tree))

* public static void **transplant**(node u, node v)

Since, it basically involves changing pointers, it’s complexity is linear.

Complexity: O(1)

* public static void **Delete\_Fix**(node x)

It restores the properties of the Red-Black tree.

Complexity : O(log n)

* public static void **printTree**(node a)

Prints the tree whose root is ‘a’

Complexity: O(log n)

* public static node **insert**(node x)

Inserts a new node into the tree. Then, calls Insert\_Fix(node x) to restore the properties.

Complexity: O(log n)

* public static void **Insert\_Fix**(node x)

Restores the Red-Black properties.

Complexity: O(log n)

* public static void **Left\_Rotate**(node x)

Changes the pointer structure to preserve the binary search tree property

Complexity: O(1)

* public static void **Right\_Rotate**(node x)

Changes the pointer structure to preserve the binary search tree property

Complexity: O(1)

* public static int **Increase**(int ID, int m)

Uses Find\_Node(node a, int ID) to see if the node exists, if it does, it increases it’s count by ‘m’ else, it inserts a new node into the tree by calling Insert(node x).

Complexity: O(log n)

* public static int **Reduce**(int ID, int m)

Uses Find\_Node(node a, int ID) to see if the node exists, if it does it decreases it’s count, and if count becomes less than or equal to zero it deletes the node by calling delete(node x)

Complexity: O(log n)

* public static node **Find\_Node**(node a, int ID)

Finds a node with id as ID in a tree with root ‘a’

Complexity: O(log n)

* public static int **Count**(int ID)

Uses Find\_Node (node a, int Id) to determine if such a node exists, and retrieves the count attributes associated with that node.

Complexity: O (log n)

* public static node **Next**(int k)

Determines the next closest greater node to a node with ID k

Complexity: O(log n)

* public static node **Previous**(int k)

Determines the next closest smaller node to a node with ID k

Complexity: O(log n)

* public static int **inRange**(node root,int k1,int k2)

It finds the sum of count attributes of ID’s between k1 and k2.

Complexity= O(log n + number of ID’s)

Class : **node**

Defines the structure of node with the following attributes:

1. Color (Either 0 or 1 ; 0: Black, 1: Red)
2. Parent (Points to the parent node)
3. Left (Points to the left child of the node)
4. Right(Points to the right child of the node)
5. ID (Holds the ID)

Constructor:

node(int ID, int cnt) : Assigns value to each new node.

**Test Run Time for Tree Construction**

* Test\_100 : almost 0ms
* Test\_1000000 : 944ms
* Test\_10000000 : 12377ms
* Test\_100000000 : 165313ms (on 8GB)

**Compiler**

* Netbeans 8.1

**Resources**

* Introduction to Algorithms – Cormen
* Computer Algorithms C++ - Sartaj Sahini

**ScreenShot**

