ADVANCED DATA STRUCTURES (COP 5526)

Spring 2016

Instructor: Dr.Sartaj Sahni

Submitted By:
Swati Sisodia
UFID: 04065889
Email : ssisodia@ufl.edu

1. Introduction

This report serves to address the structure and flow of the program. The project has been written in C++ and has been compiled using the g++ compiler. A make file has been provided with the source code. “make clean” will delete all the binary and .out files. “make” will compile the project. The compilation will generate a binary called ‘bbst’ which can be run.

2. Program Structure

The Program can be divided into implantation of following three parts.

- Structure and operations of Red Black trees
- Operations of event counter using the red black tree structure as specified by the requirements.
- Command driven interface

2.1 Implementation of Red Black Tree Data Structure

For the implementation of Red Black Trees, we have a node class to create nodes of tree as shown below:

```
struct node
{
    int key;
    int count;
    node *parent;
    char color;
    node *left;
    node *right;
};
```

Each object has a left and right pointer, pointing to the node’s two children, parent pointer pointing to parent, variable color to fold the color of node. The key and count are used to store id and count of the id.

The object Red Black tree is created using class rbtree. Definition of class is as shown below:

```
class rbtree
{
    node *root;
    node *q;
    public:
    rbtree()                        // Constructor to initialize parameters
    {
            q = NULL;
            root = NULL;
    }
    void insert(int id, int count);
    void complete_insertion(node *);
    void leftrotate(node *);
    void rightrotate(node *);
    void del(node *p);
    void complete_deletion(node *);
```

```
        void next(node *root ,int id,int &set);
        void inrange(node *root, int id1,int id2,int* sum);
        void  previous(node *root, int id, int *set  ,int* pkey,int* pcount);
        void reduce(int id, int m);
        void increase(int id, int m);
        node* successor(node *);
        node* createExternalNode(node *);
        node* search(int id);
        node* getroot();


};
```

The functionalities increase and reduce involves insertion and deletion of node into/from the tree. Insert and Delete functions have been implemented in two steps using following functions:

Insertion :

void insert(int id, int count);

void complete_insertion(node *);

Deletion :

void del(node *p);

void complete_deletion(node *);

- The **insert** function takes id and count as parameters , traverses the tree comparing id with key of nodes in the trees , finds the correct position for the node and adds it to the free by adjusting the pointer of parent and children. The new node is given color red by default. Since addition of node can cause imbalance in red black tree , complete_insertion is run to readjust the colors of nodes to balance the tree.
- **complete_insertion** takes the new node(x) as parameter and uses recoloring/rotation to balance the tree. The technique to use (recoloring /rotation) is determined by first determining the uncle node i.e the sibling of parent node. If uncle node has color red, recoloring is performed by swapping the colors of parent (p) and grandparent (g) and reassess the grandparent for imbalance. If uncle node has color black, then left rotate and right rotate are performed based on the the following four cases:

Case	**Condition**	**Action**
Left Left Case	p is left child of g and x is left child of p	perform right rotate on g and swap colors of p and g
Left Right Case	p is left child of g and x is right child of p	left rotate on p and repeat steps of left left case
Right Right Case	p is right child of g and x is right child of p	perform right rotate on g and swap colors of p and g
Right Left Case	p is right child of g and x is left child of p	right rotate on p and repeat steps of right right case

Since position is determined using binary search and readjusting is done in constant time. The total time for insertion of new node is O(logn).

- The **del** function takes pointer to the node which is to be removed as argument. Finds the successor node i.e. the node which will be taking place of removed node. The successor node is determined by successor function :

  node* successor(node *);

  The function takes node as argument and finds if right or left child or external node will replace the node. Once successor node is determined , pointers of parent and successor node are readjusted to remove the node from the tree. Removal of node can again cause imbalance.delete_insertion is performed on the successor node to balance the tree.

- **delete_insertion** takes node pointer as argument and uses recoloring/rotation to balance the tree. Contrary to insertion first sibling is determined. Since the main concern while replacing deleting a node is change in number of black nodes from root to leaf, if either of node to be removed or its successor is red , node is removed and successor is colored black. If both node to be removed (u) and its successor (v) are black , then color of sibling (s) is checked and actions are performed according to the following conditions :

Conditions	**Action**
s is black and at least one of sibling's children is red (r) : *Left Left Case*: s is left child of its parent and r is left child of s or both children of s are red *Left Right Case* : s is left child of its parent and r is right child *Right Right Case* : s is left child of its parent and r is right child *Right Left Case* : s is right child of its parent and r is left child of s	Remove u, left rotate on sibling Remove u, Right rotate on sibling and left rotate on parent Remove u, right rotate on sibling Remove u, Left rotate on sibling and Right rotate on parent
s is black and its both children are black	Change color of s as red and recur for the parent if parent is black
s is red :	

Left Case (s is left child of its parent) Right Case (s is right child of its parent)	Remove u and right rotate parent p Remove u and right rotate parent p
u is root	Remove u and keep color of v as black

- **Rotation Functions**

  Rotation is performed using following member functions of class rbtree :

  void leftrotate(node *);

  void rightrotate(node *);

  The functions take pointer to the pivot node about which rotation is to performed as argument and performs pointer readjustments to rotate the tree about pivot node.

In addition to insertion and deletion operation functions and their utility functions, additional functions created are :

- **Search Function**:

  node* search(int id);

  It takes id to be searched for as argument ,performs binary search on Red Black Tree for the id and returns the node corresponding to the id.

- **External Node Creation Function**:

  node* createExternalNode(node *);

  Function creates external nodes with color black and key = -1. This is done to avoid null pointer problems while performing operations on leaf nodes ,

- **Root returning function**:

  node* getroot();

  Simply returns root of tree.

2.2 Implementing Event Counter using Red Black Tree Structure

To implement the event counter and its operations following functions were included in the member functions of class member tree :

- **increase**

  void increase(int id, int m);

  The function takes id and integer m by which count needs to increased as argument. It then performs look up for the id using search function. If id exists , it increases the count by m else it performs insert(id,m) which creates a new node with id as the key and m as the count.

- **reduce**

  void reduce(int id, int m);

  Simillar to increase ,he function takes id and integer m by which count needs to decreased as argument. It then performs look up for the id using search function. If id exists , it decreases the count by m else it simply prints 0. After decreasing the count , it checks if the count is less than zero. If yes, then it calls del(node* p) to remove the node from the tree.

- **next**

  void next(node *root ,int id,int &set**);**

  The Function is a recursive function for inorder traversal of Red Black Tree. According to requirements , the next command should return node id of smallest node whose id is greater than the given id.This is the node which comes just after the given id while traversing the tree inorder. Set bit is used to indicate that the previous node was the given id.Intially set variable is set to 0 . While traversing, if the given id matches the current node.set is set to 1. Whenever a node encounters set bit = 1 ,it prints its id and changes value of set to 2 which denotes completion of next. If given id is not found or there is no id greater than given id ,set either remains 0 or 1.

- **previous**

  void previous(node *root, int id, int *set ,int* pkey,int* pcount);

  Simillar to next previous performs inorder traversal. While traversing the tree, before moving to next node , current node is set in variable prev. Whenever current node id is same as given id, key corresponding to prev is printed and set is set to 1 denoting completion of next operation. If the node has no previous element i.e the node is first element while traversing tree inorder or given id is not found ,set value remains 0.

- **inrange**

  void inrange(node *root, int id1,int id2,int* sum);

  Inrange function again performs inorder traversal from id1 to id2 with set1 and set2 variables set as markers when corresponding values are encountered .The count of keys between id1 and id2 is added to the sum variable which hold the total count of id's between id1 and id2.

- **count**

  node* search(int id);

count functions is implemented using search function which performs binary search for given id and returns the node corresponding to the id whose count is then printed.If id is not found and search function returns null value , zero Is printed.

2.3 Implementation of the interactive part (main and run Methods)

In main method ,rbtree object is created .Then filename is taken as command line argument and read line by line to get id and count pair and create Red Black Tree nodes using insert (id, count). Once all nodes are read and corresponding nodes are inserted into the red black tree structure , run function is called.

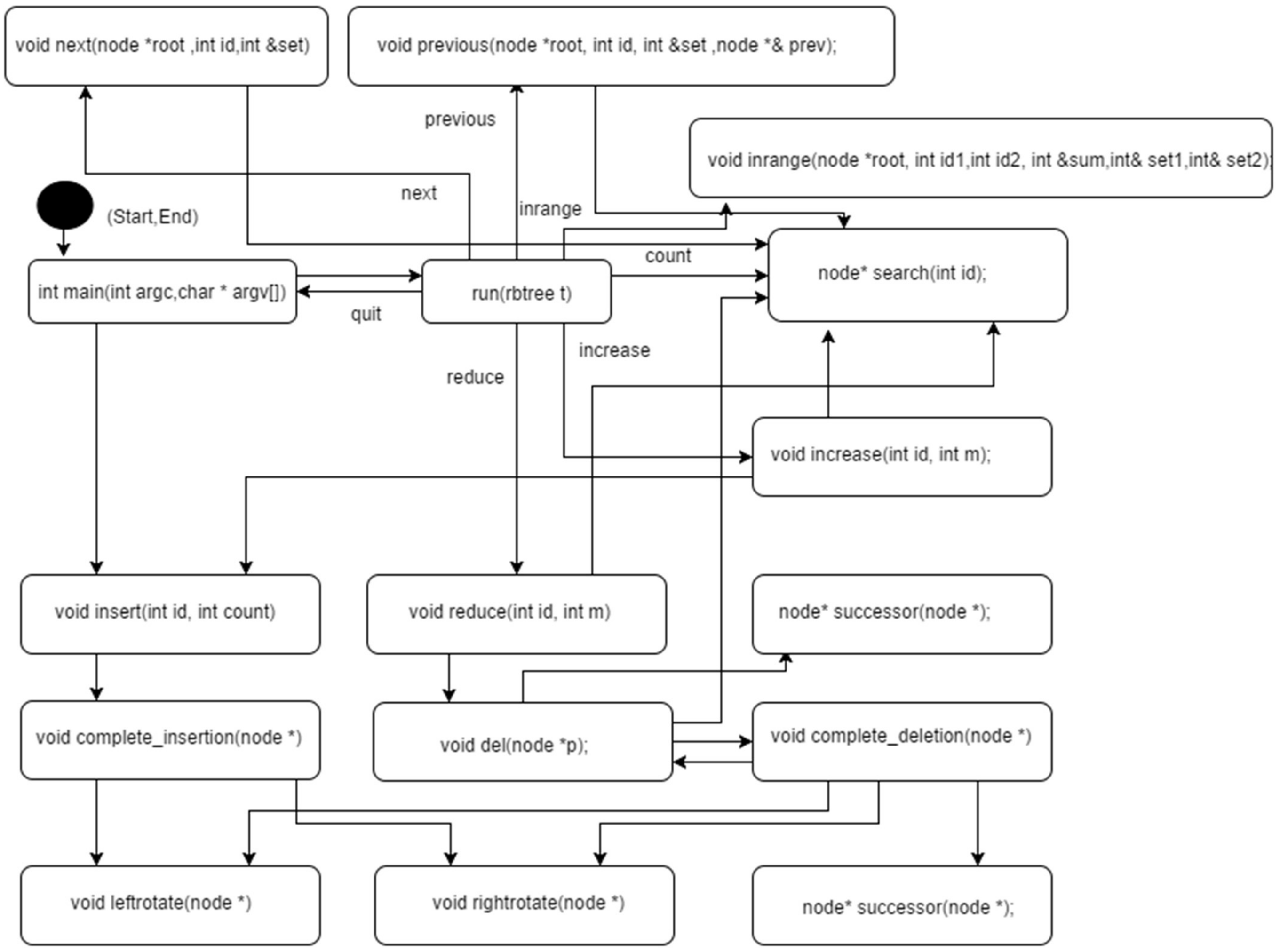
void run(rbtree t);

Run function takes following commands from console and calls corresponding functions on rbtree object(t).On completion of a command run function recursively calls itself unless it encounters quit command :

Console Command	**Corresponding Function**
increase id m	t.increase(atoi(arg[1].c_str()), atoi(arg[2].c_str()))
reduce id m	t.reduce(atoi(arg[1].c_str()), atoi(arg[2].c_str()));
count id	node *p = t.search(atoi(arg[1].c_str()));
inrange id1 id2	t.inrange(t.getroot(), atoi(arg[1].c_str()), atoi(arg[2].c_str()), sum, set1,set2);
next id	t.next(t.getroot(),atoi(arg[1].c_str()),set);
previous id	t.previous(t.getroot(),atoi(arg[1].c_str()),set,prev);
Quit	return

3. Flow Diagram

The following flow diagram represents the gist of the program :


void next(node *root ,int id,int &set)
void previous(node *root, int id, int &set ,node *& prev);
void inrange(node *root, int id1,int id2, int &sum,int& set1,int& set2);
previous
next
inrange
(Start,End)
count
int main(int argc,char * argv[])
run(rbtree t)
node* search(int id);
quit
increase
reduce
void increase(int id, int m);
void insert(int id, int count)
void reduce(int id, int m)
node* successor(node *);
void complete_insertion(node *)
void del(node *p);
void complete_deletion(node *)
void leftrotate(node *)
void rightrotate(node *)
node* successor(node *);

Note : To reduce confusion , return paths are omitted. Flow will always return to calling function after completion of execution.

4. References

- http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/
- http://www.coders-hub.com/2015/07/red-black-tree-rb-tree-using-c.html#.VutBXvkrLP6
- https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html
- http://algs4.cs.princeton.edu/33balanced/RedBlackBST.java.htmlhttp://algs4.cs.princeton.edu/33balanced/RedBlackBST.java.html
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree