COP 5536: ADVANCED DATA STRUCTURES

PROGRAMMING PROJECT

The red-black tree based event counter has been implemented in C++ using gcc compiler version 4.9.2. The structure of the project and the method prototypes are described below:

```
struct node
{
   int ID;//ID of the node
   node *parent;//parent of the node
   char color;//color of the node
   node *left;//left child of the node
   node *right;//right child of the node
   int count;//count of the node
};
```

i) The functional unit of the Red-Black tree is the node which holds the ID and count for the event counter. Here, I have described the node's structure definition.

To implement the event counter based Red-Black Tree, I have implemented all the methods in a single class. As you can notice, I have considered methods Increase and Reduce directly as part of insertion/deletion of nodes since insertion/deletion of nodes takes place only under special cases of Increase/Reduce. (Complete explanation is given ahead):

```
void Increase(int, int);
void Count(int);
void InRange(int, int);
void Next(int);
void Previous(int);
void Reduce(int, int);
```

ii) Increase: This method increases the count of the event ID by amount m. If ID is not present, it inserts it. (Hence, here instead of calling a separate method like insert_node which inserts a new node into the Red-Black Tree, I have merged the two methods together so as to facilitate program codability.) It then prints the final count. This runs in O(lg n) time. This method calls the insertfix(t) method from RBTree class.

iii) Reduce: This method decreases the count of the event ID by amount m. If the ID's count becomes less than or equal to 0, it removes the ID from the counter. (Hence, here instead of calling a separate method like delete_node which deletes a node from the Red-Black Tree, I have merged the two methods together so as to facilitate

program codability.) It then prints the count of the ID after deletion or 0 if the ID is removed or is not present. This runs in O(lg n) time.

iv) Count: This function searches the counter for the event ID and returns the count. If it is not present, it returns 0.

v) InRange: It is a recursive function which returns the total count for IDs between ID1 and ID2.

vi) Next: It returns the event with the lowest ID that is greater than ID.

vii) Previous: It returns the event with the greatest ID that is less than ID.

```
node* Initialize(int dp, int low, int high)
{
    int N, mid;
    N=high-low+1;
    if(N<=0) return obj.nil;
    mid=low-1+(N+1)/2;
    node *u=new node;
    u->left=Initialize(dp+1, low, mid-1);
    u->left->parent = u;
    u->right=Initialize(dp+1, mid+1, high);
    u->right->parent = u;
    u->key=ID[mid];
    u->count=count[mid];
    if(dp==H) u->color='r';//the node whose height is H only has color 'r'
    else u->color='b';
    return u;
}
```

viii) Initialize: We know that while building Red-Black Trees or BBST's in general, if we build using the conventional insert node function, it takes O(nlgn) time. Hence, in order to build a red-black tree from sorted list of events in O(n) time, I have used the Initialize method.

Given below are some of the functions which I have called from the functions described above:

ix) insertfix: This method is used to restore the red-black properties which might have been violated after insertion of a new node. It uses a sequence of color flips and rotations to rebalance the tree. It handles the cases where the inserted node's uncle is RED, inserted node's uncle is BLACK and the node is a right child, the inserted node's uncle is BLACK and the node is a left child.

x) delfix: This method is used to restore the red black properties that might have been violated after deletion of a node. It uses a sequence of color flips and rotations to rebalance the tree. It handles the cases where the successor node's sibling is RED, successor's sibling is BLACK and both of the sibling's children are BLACK, the successor's sibling is BLACK whose left child is RED and right child is BLACK, successor's sibling is BLACK whose right child is RED.

xi) leftrotate: Performs left rotation on the subtree to restore balance. The left rotation pivots around the link from the node to its right child.

xii) rightrotate: Performs right rotation on the subtree to restore balance. The right rotation pivots around the link from the node to its left child.

Following is the running times for the different test files given:

1) 100 input events: 0.007s
2) 10^6 input events: 0.265s
3) 10^7 input events: 2.401s
4) 10^8 input events: 96.036s