COP 5536 Project

Implementation of Event Counter Using Red-Black Tree

Ying Wang UFID:13917195 University of Florida

Contents:

I. Background

A. Problem Description

This project asks us to implement an event counter using red-black tree. Each event has two fields: *ID* and *count*, where *count* is the number of active events with the given *ID*. The event counter stores only those *ID*'s whose *count* is > 0. Once a *count* drops below 1, that *ID* is removed. Initially, your program must build red-black tree from a sorted list of *n* events (i.e., *n* pairs (*ID*, *count*) in ascending order of *ID*) in O(*n*) time. The counter should support the following operations in the specified time complexity.

B. Input and Output Requirements

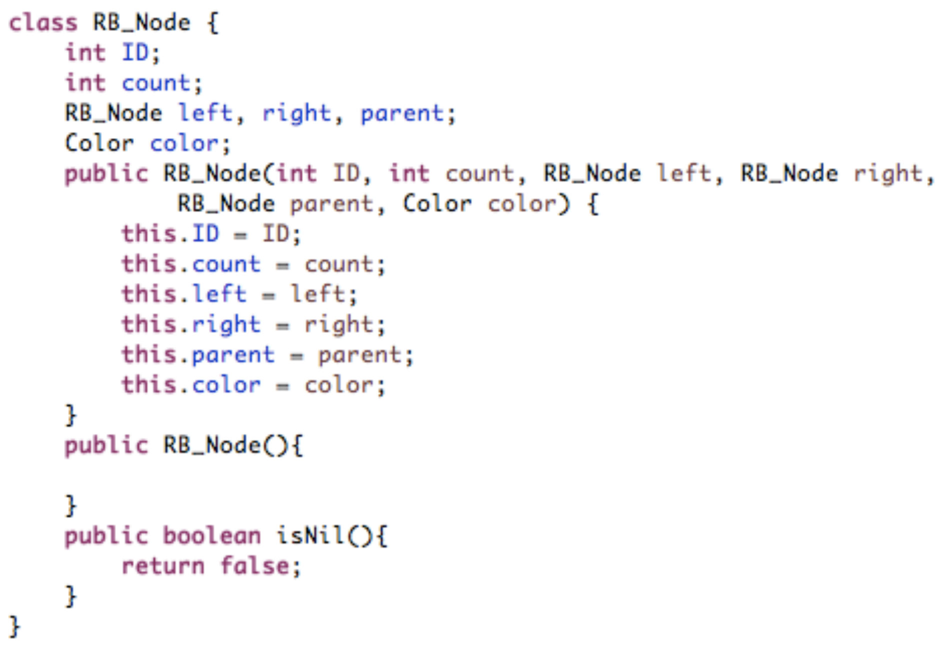
We are required to write a makefile document which creates an executable. The names of the executable must be bbst.
The program has to support redirected input from a file "file-name" which contains the initial sorted list. I programmed this project using Java, so the command line should be:
$bbst file-name $java bbst file-name
And the Input format is:
n ID1 count1
ID2 count2
...
 IDn countn
Assume that *IDi* < *IDi+1* where *IDi* and *counti* are positive integers and the total count fits in 4-byte integer limits.

II. Implementation

A. How to build a red-black tree

A.1Class of internal nodes and nil nodes design

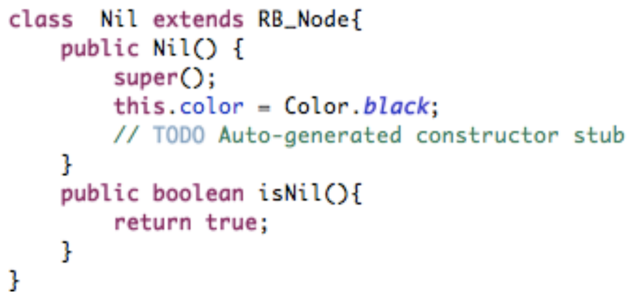
A traditional ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.
In this project, I use red-black tree to implement an event counter, so I need to define an extra ID and count for each of the node indicating an event. Here, inside the RB_Node class, I also use a Boolean function names "IsNil" to indicate the internal nodes are not nil nodes. The class of internal nodes is shown as Figure1 below:



```java
class RB_Node {
    int ID;
    int count;
    RB_Node left, right, parent;
    Color color;
    public RB_Node(int ID, int count, RB_Node left, RB_Node right,
            RB_Node parent, Color color) {
        this.ID = ID;
        this.count = count;
        this.left = left;
        this.right = right;
        this.parent = parent;
        this.color = color;
    }
    public RB_Node(){

    }
    public boolean isNil(){
        return false;
    }
}
```

Figure1 Define the Internal Nodes Properties

Since in red-black tree, external nodes are indicated as nil nodes instead of null, I define a Nil class to inherit all properties of internal nodes, except for coloring them as black. Also, the Boolean function isNil() is override and set to true instead.

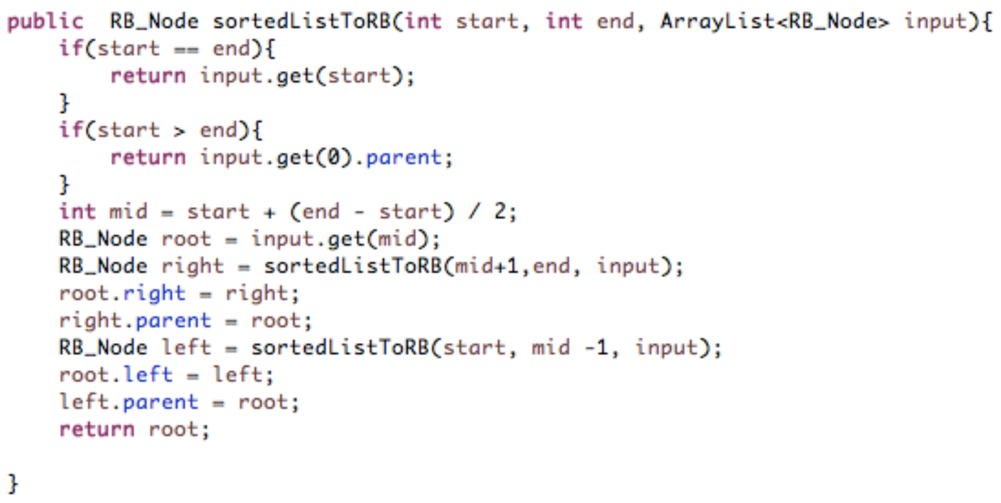


```java
class  Nil extends RB_Node{
    public Nil() {
        super();
        this.color = Color.black;
        // TODO Auto-generated constructor stub
    }
    public boolean isNil(){
        return true;
    }
}
```

Figure2 Define Nil Nodes Properties

A.2 Initialization of red-black tree through input file

Given the input file indicating each event's ID and count, I read the file and initialized each node as a black nil node with corresponding ID and count and put them into an arraylist. Now that I have the sorted list of nodes, I can build a balanced binary search tree using the list. The way I build a tree is recursive. I first get the middle element of the list as the root, put the nodes whose IDs are less than the root in the left subtree, and the rest in the right subtree. In the meantime set their parent pointers to the root, and the root points to its left and right. Recursively I get the middle element of the left subtree as the root of left, and split into smaller left subtree and right subtree. The right subtree could also be built in the same way. Figure3 indicates how I use a sorted list to build a binary search tree.

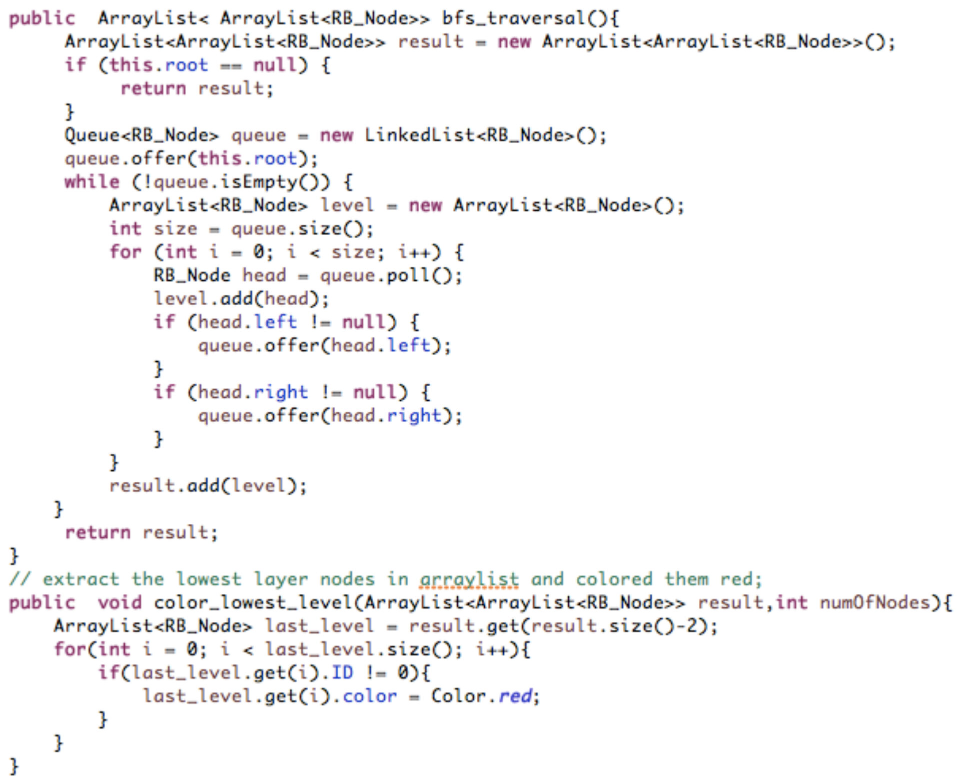


```java
public  RB_Node sortedListToRB(int start, int end, ArrayList<RB_Node> input){
    if(start == end){
        return input.get(start);
    }
    if(start > end){
        return input.get(0).parent;
    }
    int mid = start + (end - start) / 2;
    RB_Node root = input.get(mid);
    RB_Node right = sortedListToRB(mid+1,end, input);
    root.right = right;
    right.parent = root;
    RB_Node left = sortedListToRB(start, mid -1, input);
    root.left = left;
    left.parent = root;
    return root;

}
```

Figure3 Convert Sorted List to A Balanced Binary Tree

A.3 How to color red and black

By constraining the node colors on any simple path from the root to a leaf, red-black tree ensure that no such path is more than twice as long as any other, so that the tree is approximately *balanced*. Thus the easiest way to ensure the number of black nodes are the same on any path from the root to leaf is to color the lowest layer nodes as red (not nil nodes), and all internal nodes as black. The way I find lowest layer nodes is to use breath first traversal to find the lowest layer of nodes and recolored them. The algorithm is shown in Figure4 below.



```java
public  ArrayList< ArrayList<RB_Node>> bfs_traversal(){
     ArrayList<ArrayList<RB_Node>> result = new ArrayList<ArrayList<RB_Node>>();
     if (this.root == null) {
         return result;
     }
     Queue<RB_Node> queue = new LinkedList<RB_Node>();
     queue.offer(this.root);
     while (!queue.isEmpty()) {
         ArrayList<RB_Node> level = new ArrayList<RB_Node>();
         int size = queue.size();
         for (int i = 0; i < size; i++) {
             RB_Node head = queue.poll();
             level.add(head);
             if (head.left != null) {
                 queue.offer(head.left);
             }
             if (head.right != null) {
                 queue.offer(head.right);
             }
         }
         result.add(level);
    }
     return result;
}
// extract the lowest layer nodes in arraylist and colored them red;
public  void color_lowest_level(ArrayList<ArrayList<RB_Node>> result,int numOfNodes){
    ArrayList<RB_Node> last_level = result.get(result.size()-2);
    for(int i = 0; i < last_level.size(); i++){
        if(last_level.get(i).ID != 0){
            last_level.get(i).color = Color.red;
        }
    }
}
```

Figure4 Using BFS Traversal and Color the Lowest Layer Nodes as Red

One noteworthy point is that since I also stored nil nodes during bfs traversal, the lowest layer nodes are nil nodes instead of leaf nodes. So the leaf nodes are one layer above, to extract all leaf nodes, I have to exclude the nil nodes by judging whether is ID is 0(default value) at that layer also. The built tree is similar to the tree below:

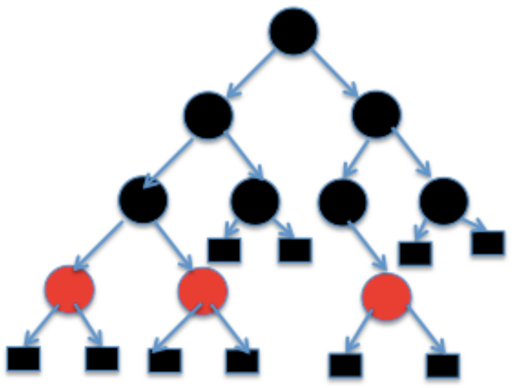


Figure5 The Initialized Structure of the Red-Black Tree in My Program

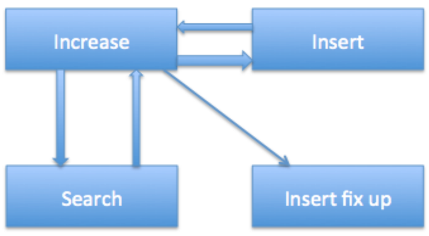
B. Design of Event Counter

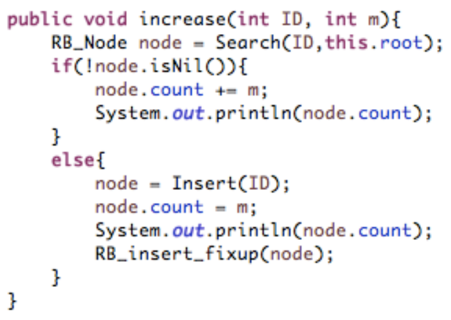
B.1 Increase (theID, m) function

The function asks us to increase the *count* of the event *theID* by *m*. After analysis, I found there are two scenarios of the function. The given ID could already exist in the tree or it doesn't exist in the tree. So we first need to search the tree to find the ID exists or not.

Thus inside the increase function, I first call search() function given ID and root as parameters to do a binary search. If the ID exists, return the node with corresponding ID, otherwise return null. In the first case, I just need to increase the count of the returned node by m and print it, in the latter scenario it's much more complex.

If the node doesn't exist, I should find where to insert it. So in the second case, I call a insert() function to insert the node into the existing tree. In order not to destroy the number of black nodes on any path from root to leaf, the newly inserted node must be colored red. However, it's possible that the node's parent could be red, in this way, it disobeys the rule in red-black tree that two consecutive red nodes are not allowed, so after insertion, I have to call a Insert_fixup() function to make the tree satisfy the properties again. In the latter part, I will talk about search and insert function in detail and illustrate the relationship among them. After insertion, I printed out the count of the node. The structure and algorithm is shown in Figure5 below:



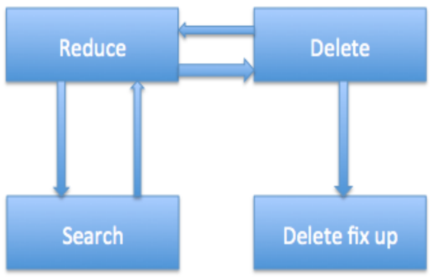


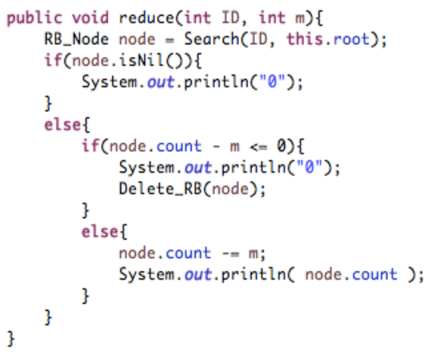
```
public void increase(int ID, int m){
    RB_Node node = Search(ID,this.root);
    if(!node.isNil()){
        node.count += m;
        System.out.println(node.count);
    }
    else{
        node = Insert(ID);
        node.count = m;
        System.out.println(node.count);
        RB_insert_fixup(node);
    }
}
```

Figure5 The Structure of Increase() Function

B.2 Reduce(theID, m) function

The function asks us to reduce a given ID by m. In the similar way as Increase() function, there are also two scenarios. For the first case, if the ID already exists in the tree, reduce its count by m, however, after the reduction, the ID could be less or equal to 0. In this case, the corresponding node should be deleted. Thus inside the first case, I first judge whether the count of node is less or equal to 0 after reduction, if it is, I call a delete() function(which I will talk about later) to remove the node, and print its count as “0”, otherwise print the node’s count reduced by m. In the second case, which the node doesn’t exist, I simply print “0” to indicate it’s not in the tree. The structure and function is shown as below:



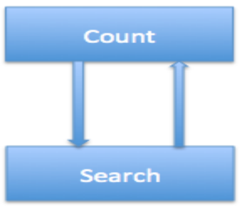


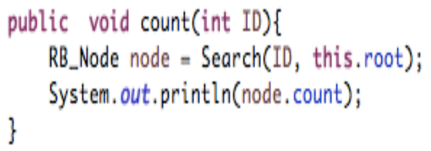
```
public void reduce(int ID, int m){
    RB_Node node = Search(ID, this.root);
    if(node.isNil()){
        System.out.println("0");
    }
    else{
        if(node.count - m <= 0){
            System.out.println("0");
            Delete_RB(node);
        }
        else{
            node.count -= m;
            System.out.println( node.count );
        }
    }
}
```

Figure6 The Structure of Reduce() function

B.3 Count(theID)

This function requires us to print the count *of theID*. If not present, print 0. The function is pretty straightforward. I simply call a search() function to see the returned node is null or not. If it’s not null, just print its count, otherwise print”0” instead.



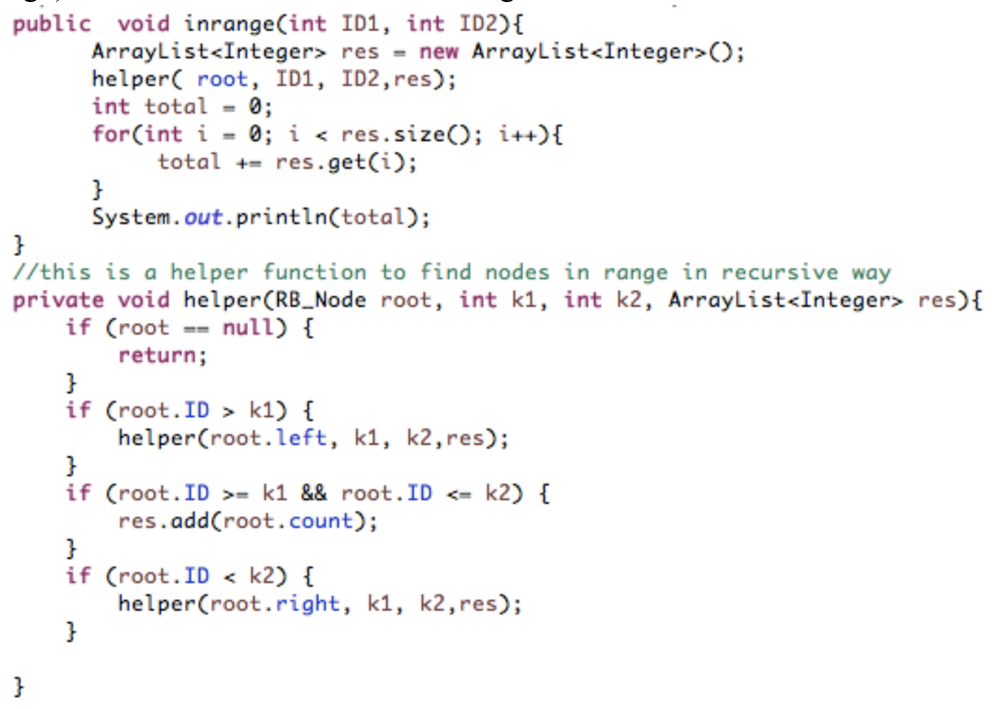


```
public  void count(int ID){
    RB_Node node = Search(ID, this.root);
    System.out.println(node.count);
}
```

Figure7 The Structure of Count() Function

B.4 InRange(ID1, ID2)

The requirement is to print the total count for *ID*s between *ID1* and *ID2* inclusively. Note, *ID1* ≤ *ID2*. There are many algorithms to achieve this, a naïve way is to do an inorder traversal and find all IDs within the range, but since the function is required to complete in O(logn) time, I use an algorithm similar to binary search. Starting at root point, If ID is less than ID2, go to its right child recursively until there is one ID greater than ID2, then we reach the right bound node, If ID is greater than ID1, go to its left child recursively until there is one ID less than ID1, then we reach the left bound node. When doing the search, add all nodes whose IDs are within the range, and compute the sum of all the nodes' count. In this way, I can search all nodes in O(logn) time. The function is shown in Figure8.



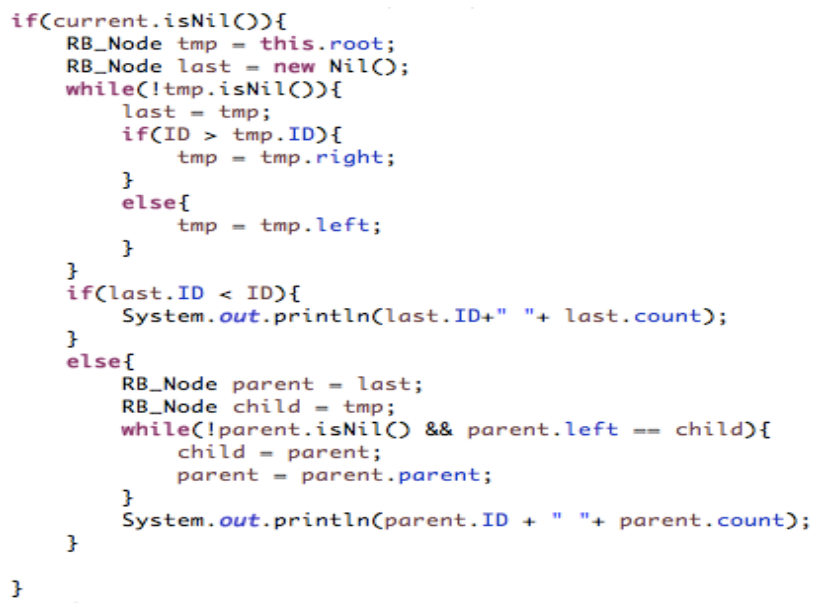
```
public  void inrange(int ID1, int ID2){
      ArrayList<Integer> res = new ArrayList<Integer>();
      helper( root, ID1, ID2,res);
      int total = 0;
      for(int i = 0; i < res.size(); i++){
           total += res.get(i);
      }
      System.out.println(total);
}
//this is a helper function to find nodes in range in recursive way
private void helper(RB_Node root, int k1, int k2, ArrayList<Integer> res){
    if (root == null) {
        return;
    }
    if (root.ID > k1) {
        helper(root.left, k1, k2,res);
    }
    if (root.ID >= k1 && root.ID <= k2) {
        res.add(root.count);
    }
    if (root.ID < k2) {
        helper(root.right, k1, k2,res);
    }

}
```

Figure8 Inrange() Function of O(logn) Time Complexity

B.5 Previous(theID)

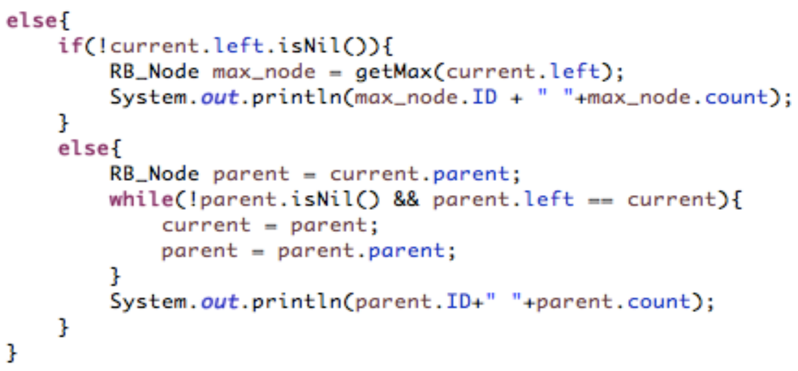
The requirement is to print the *ID* and the *count* of the event with the lowest *ID* that is greater that *theID*. If there is no next *ID*, just print "0 0". Again, there are two cases. If the ID exists in the tree, we just need to find its predecessor, if there is no predecessor, in other words, its predecessor is a nil node, just print "0 0". In another case, if the ID doesn't exist, I need to find its last node where it inserts into(the last node also indicated as the parent of the newly inserted node). If its parent's ID is less than the given ID, parent is its previous; otherwise the previous will be the predecessor of its parent.

One noteworthy point is how to find the predecessor in the above two scenarios. If the ID is present, the node with maximum ID in its left subtree will be the predecessor if the node has left child, (here I called getMax() function to return the node with maximum ID), otherwise its predecessor will be its lowest ancestor where the node lies in right subtree of the ancestor. If the ID is not present and its ID is smaller than the parent, since it is a leaf node, the previous will be its lowest ancestor where its parent lies in right subtree of the ancestor.



```
if(current.isNil()){
    RB_Node tmp = this.root;
    RB_Node last = new Nil();
    while(!tmp.isNil()){
        last = tmp;
        if(ID > tmp.ID){
            tmp = tmp.right;
        }
        else{
            tmp = tmp.left;
        }
    }
    if(last.ID < ID){
        System.out.println(last.ID+" "+ last.count);
    }
    else{
        RB_Node parent = last;
        RB_Node child = tmp;
        while(!parent.isNil() && parent.left == child){
            child = parent;
            parent = parent.parent;
        }
        System.out.println(parent.ID + " "+ parent.count);
    }

}
```

Figure9 the ID not Present in the Tree



```
else{
    if(!current.left.isNil()){
        RB_Node max_node = getMax(current.left);
        System.out.println(max_node.ID + " "+max_node.count);
    }
    else{
        RB_Node parent = current.parent;
        while(!parent.isNil() && parent.left == current){
            current = parent;
            parent = parent.parent;
        }
        System.out.println(parent.ID+" "+parent.count);
    }
}
```

Figure10 the ID is Present in the Tree

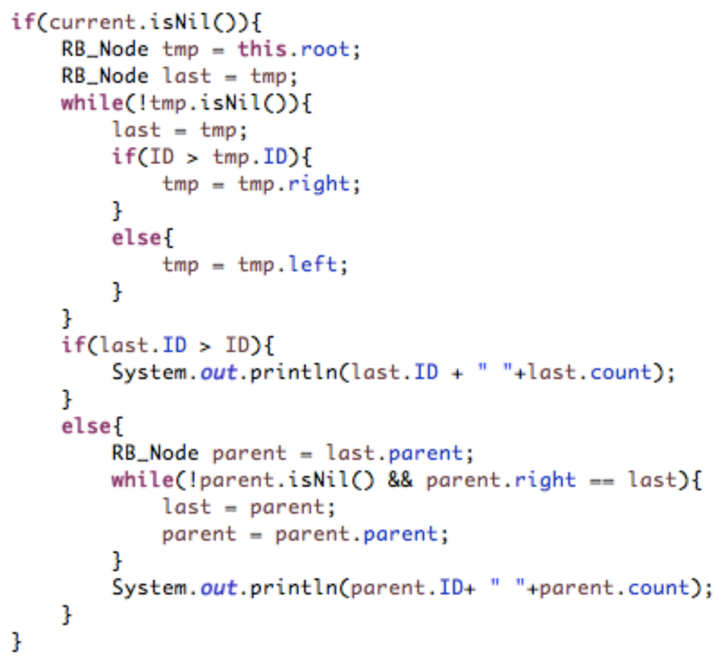
B.6 Next (theID)

This function needs to print the *ID* and the *count* of the event with the lowest *ID* that is greater that *theID*. if there is no next *ID*, just print “0 0”.

The algorithm is similar to Previous() function. If ID is present, its successor will be its next. In order to find its successor, first need to check whether it has right child. If it does, the node with minimum ID will be next, otherwise find the lowest ancestor where the node resides in left subtree of the ancestor.

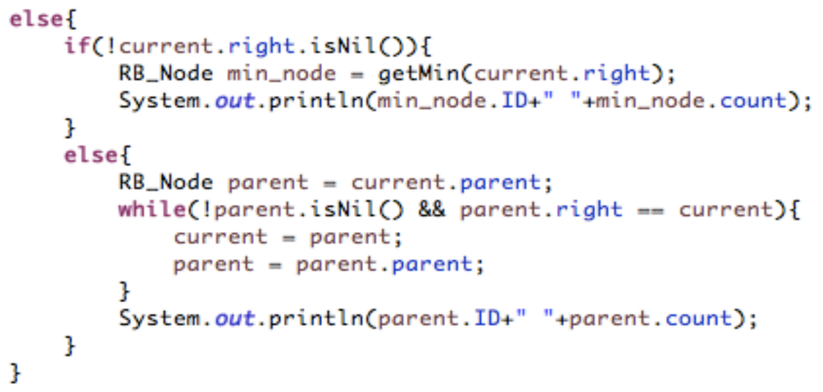
If the ID isn’t present, similarly, I check whether its last node’s ID is bigger. If it is, its last node, in other word, its parent will be the next node, otherwise I need to find the lowest ancestor where the node’s parent resides in the left subtree of the ancestor.

The algorithm shows as below:



```
if(current.isNil()){
    RB_Node tmp = this.root;
    RB_Node last = tmp;
    while(!tmp.isNil()){
        last = tmp;
        if(ID > tmp.ID){
            tmp = tmp.right;
        }
        else{
            tmp = tmp.left;
        }
    }
    if(last.ID > ID){
        System.out.println(last.ID + " "+last.count);
    }
    else{
        RB_Node parent = last.parent;
        while(!parent.isNil() && parent.right == last){
            last = parent;
            parent = parent.parent;
        }
        System.out.println(parent.ID+ " "+parent.count);
    }
}
```

Figure11 the ID not present



```
else{
    if(!current.right.isNil()){
        RB_Node min_node = getMin(current.right);
        System.out.println(min_node.ID+" "+min_node.count);
    }
    else{
        RB_Node parent = current.parent;
        while(!parent.isNil() && parent.right == current){
            current = parent;
            parent = parent.parent;
        }
        System.out.println(parent.ID+" "+parent.count);
    }
}
```

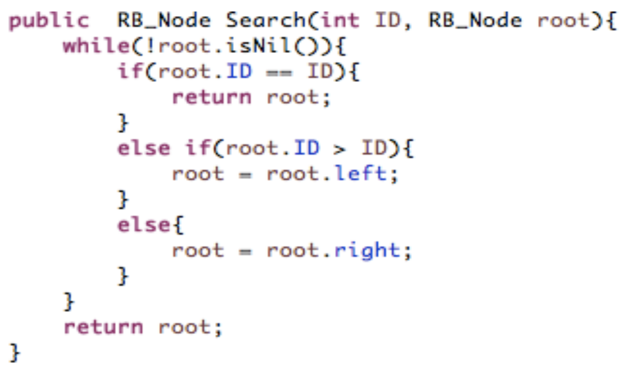
Figure12 the ID is present in the tree

B. Structure and Related Function

There are several function related to the event counter functions. In this part, I will talk about increase() related function insert(), reduce related function delete(), count() related function search() and their relationships in detail.

C.1 Search() function

The search function is straightforward. To find whether a node exists or not in the red-black tree, I use depth first binary search. Starting from the root, if given ID is equal to the root's ID, return the root, if the given ID is bigger, go to the right subtree, otherwise go to the left subtree. If the node's not present, return null. The function is shown in below Figure13.



```
public  RB_Node Search(int ID, RB_Node root){
    while(!root.isNil()){
        if(root.ID == ID){
            return root;
        }
        else if(root.ID > ID){
            root = root.left;
        }
        else{
            root = root.right;
        }
    }
    return root;
}
```

Figure13 Search() Function

C.2 Insert()

This function is called inside the insert() function when the given ID is not present in the tree. To ensure the number of black nodes are the same on any path from root to leaf, the newly inserted node has to be colored red. But when encountered the situation that the parent of the newly inserted node is also red, inset_fixup() function needs to be called to satisfy the properties of red-black tree. Here the newly inserted node is indicated as z. There are 3 cases in this situation.

Case 1: z's uncle y is red, Case 2: z's uncle y ***is black and is a right child, Case 3: z's uncle*** y ***is black and is a left child.***

The three cases are as follows:

Figure 14 shows the situation for case 1, which occurs when both z.parent and y are red. Because z.parent.parent is black, we can color both z.parent and y black, thereby fixing the problem of z and z.parent both being red, and we can color z.parent.parent red, thereby maintaining property 5. We then repeat the while loop with z.parent.parent as the new node z. The pointer z moves up two levels in the tree.

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases according to whether z is a right or left child of *z.parent*. Case 2 and 3 are shown in Figure15. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3, in which node is a left child. Because both z and z.parent are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z's uncle y is black, since otherwise we would have executed case 1. Additionally, the node z.parent.parent exists, since we have argued that this node existed at the time, and after moving´ up one level and then down one level, the identity of z.parent.parent remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The **while** loop does not iterate another time, since z.parent is now black.

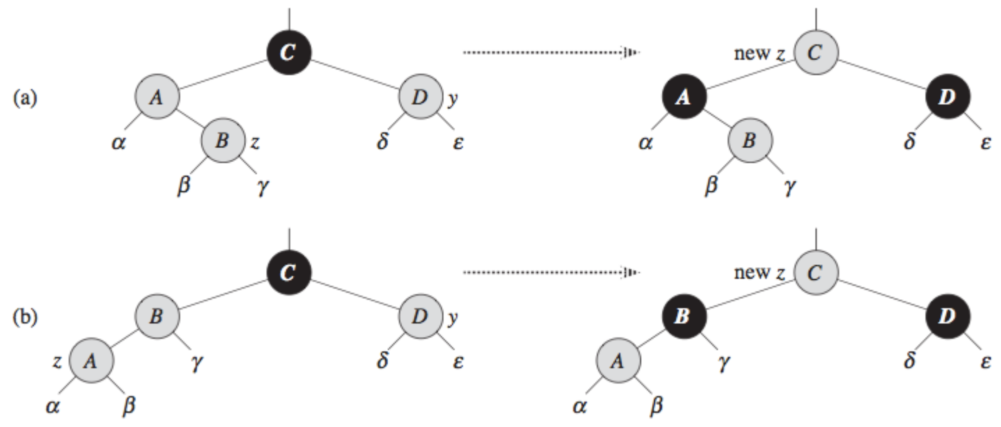


Figure14 Case1: z's Uncle is Red

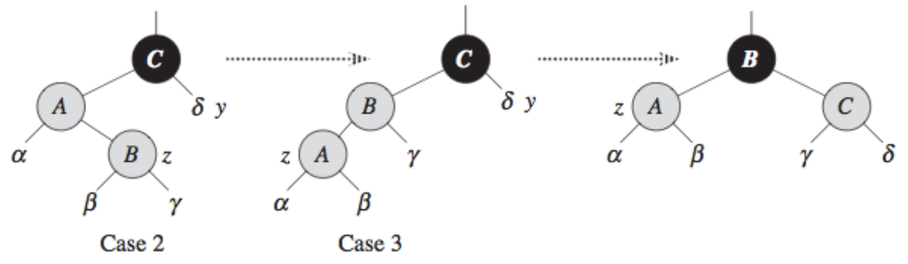
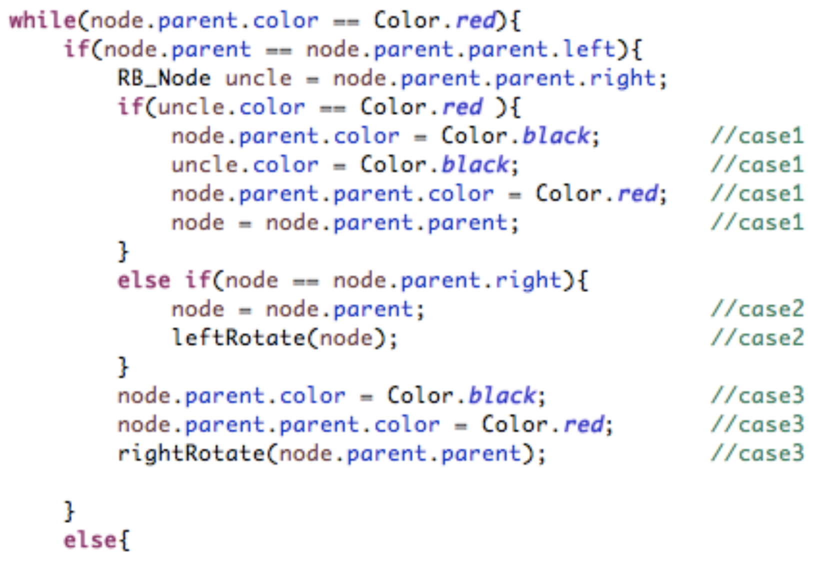


Figure15 Case 2 and 3: z's Uncle is Black and z is the Left Child and Right Child Respectively

In the following code, I just list the three cases when the newly inserted node's parent is the left subtree of its parent, in the right subtree case, situations are quite similar, just same as **then** clause with "right" and "left" exchanged. One thing to point is since the root of red black tree must be black, we have to color the root with black color after fixing up to avoid the case that the new root may be red.



```
while(node.parent.color == Color.red){
    if(node.parent == node.parent.parent.left){
        RB_Node uncle = node.parent.parent.right;
        if(uncle.color == Color.red ){
            node.parent.color = Color.black;          //case1
            uncle.color = Color.black;                //case1
            node.parent.parent.color = Color.red;     //case1
            node = node.parent.parent;                //case1
        }
        else if(node == node.parent.right){
            node = node.parent;                       //case2
            leftRotate(node);                         //case2
        }
        node.parent.color = Color.black;              //case3
        node.parent.parent.color = Color.red;         //case3
        rightRotate(node.parent.parent);              //case3

    }
    else{
```

Figure16 Inserted Node's Parent is the Left Subtree

C.3 Delete()

This function is called inside reduce() function when the given ID is present, and its count is less than or equal to 0 after reduction. If the node with given ID is red, nothing needs to be done, all properties of the red-black tree are remained. If the node happens to be black, some fix up needs to be done to ensure the properties are still satisfied. Thus, if the deleted node is black, Delete_fixup function needs to be called. There are 4 cases for fix up situation. Here we denoted the replaced node of deleted node as x.

Case 1: x***'s sibling*** w ***is red, Case 2: x's sibling w is black, and both of w's children are black, Case 3: x's sibling w is black, w's left child is red, and w's right child is black, Case 4: x's sibling w is black, and w's right child is red***

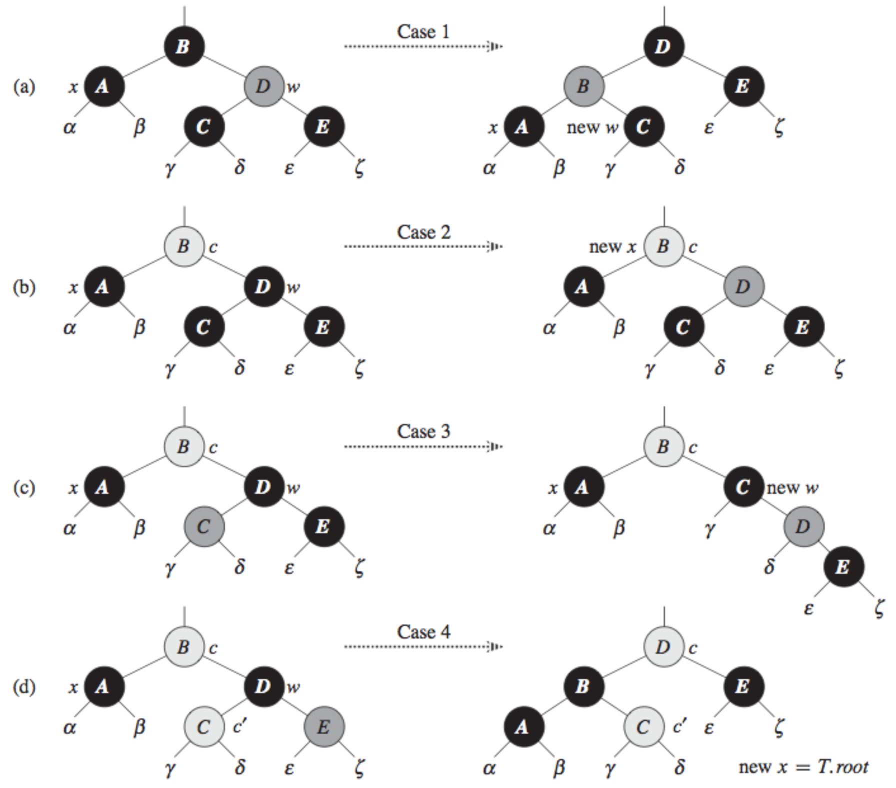
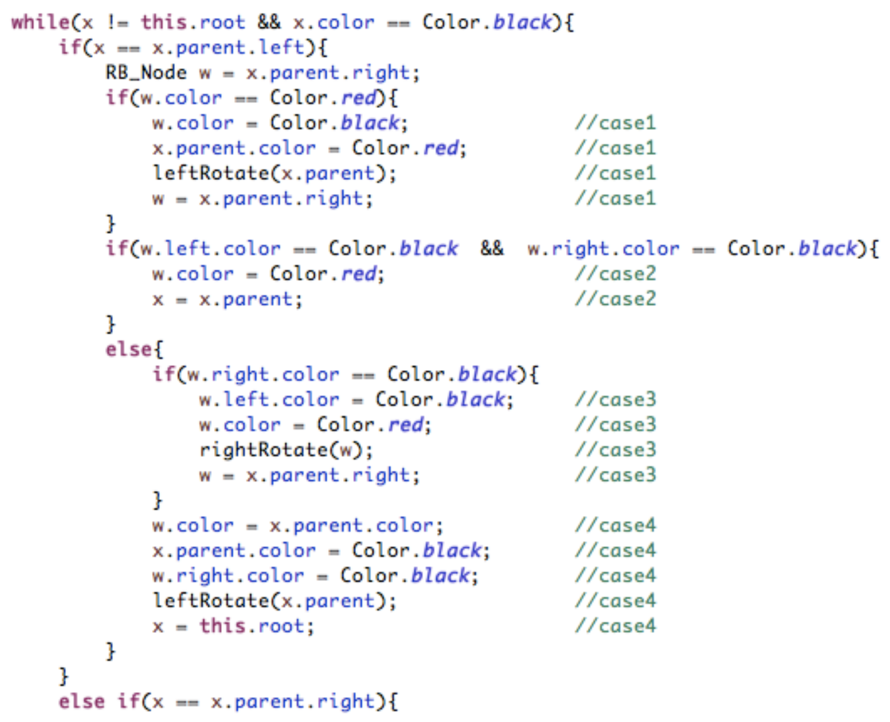


Figure17 Four Cases to Fix Up after Deletion

In the following code of Delete_fixup function, I just listed four cases when the replaced node is left child of its parent. For the case that node x is the right child of its parent, the situations are the similar, same as **then** clause with "right" and "left" exchanged.



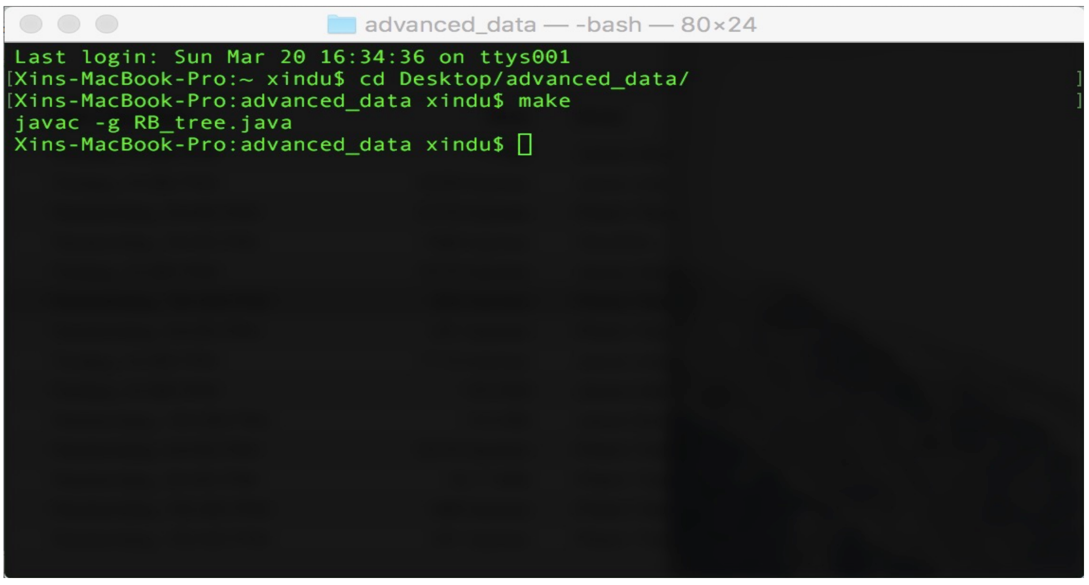
```
while(x != this.root && x.color == Color.black){
    if(x == x.parent.left){
        RB_Node w = x.parent.right;
        if(w.color == Color.red){
            w.color = Color.black;              //case1
            x.parent.color = Color.red;         //case1
            leftRotate(x.parent);               //case1
            w = x.parent.right;                 //case1
        }
        if(w.left.color == Color.black  &&  w.right.color == Color.black){
            w.color = Color.red;                //case2
            x = x.parent;                       //case2
        }
        else{
            if(w.right.color == Color.black){
                w.left.color = Color.black;     //case3
                w.color = Color.red;            //case3
                rightRotate(w);                 //case3
                w = x.parent.right;             //case3
            }
            w.color = x.parent.color;           //case4
            x.parent.color = Color.black;       //case4
            w.right.color = Color.black;        //case4
            leftRotate(x.parent);               //case4
            x = this.root;                      //case4
        }
    }
    else if(x == x.parent.right){
```

Figure18 Delete_fixup() Function

III. Test Mechanism

A. How to compile and run the file

As required, we must write a makefile document which creates an executable. The names of the executable must be bbst. Thus, I built a class in java file named bbst, and I put java file RB_tree.java, two input files named "test_100.txt" and "test_1000000.txt", and corresponding output files with correct answers. First go to the directory where all these files reside, then type the command "make", shown as in Figure below:

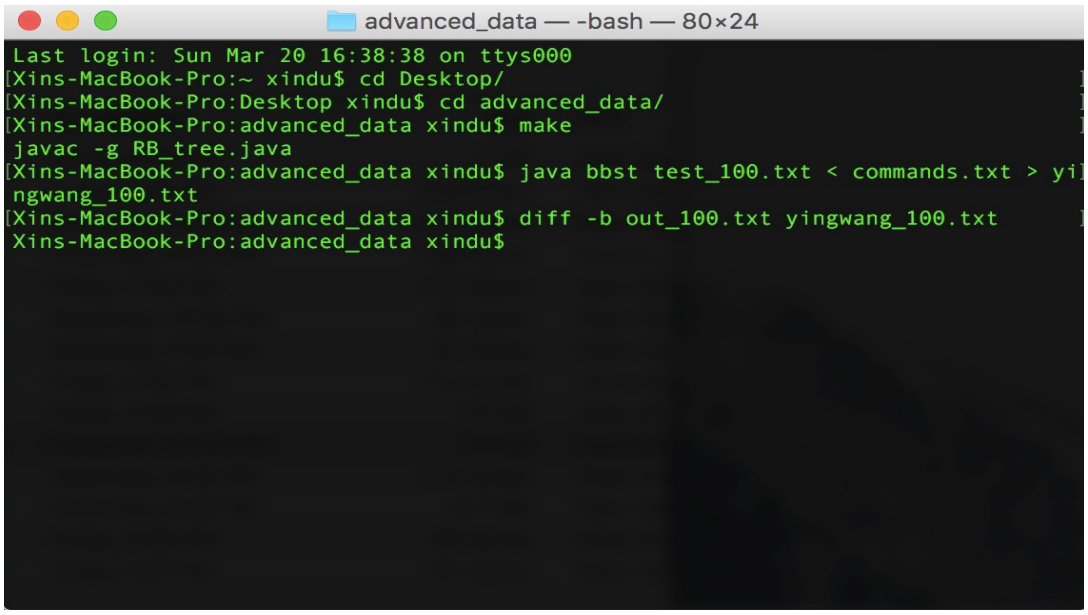


```
Last login: Sun Mar 20 16:34:36 on ttys001
Xins-MacBook-Pro:~ xindu$ cd Desktop/advanced_data/
Xins-MacBook-Pro:advanced_data xindu$ make
javac -g RB_tree.java
Xins-MacBook-Pro:advanced_data xindu$ 
```

Figure18 Run Makefile to Create an Executable

B. test_100.txt Test

In the command "java bbst test_100.txt < commands.txt > yigwang_100.txt" , bbst class is loaded, test_100.txt is the input file with 100 nodes, <commands> sends the output of my program to another newly created txt file named "yingwang_100.txt". Since we are already given the output_100.txt file of the correct answers, to test whether the output of my program is the same as the output file, I use the diff command to check that. In the following console, it shows nothing appeared after the "diff" command executes, it means the output of my program is exactly the same as the output file.

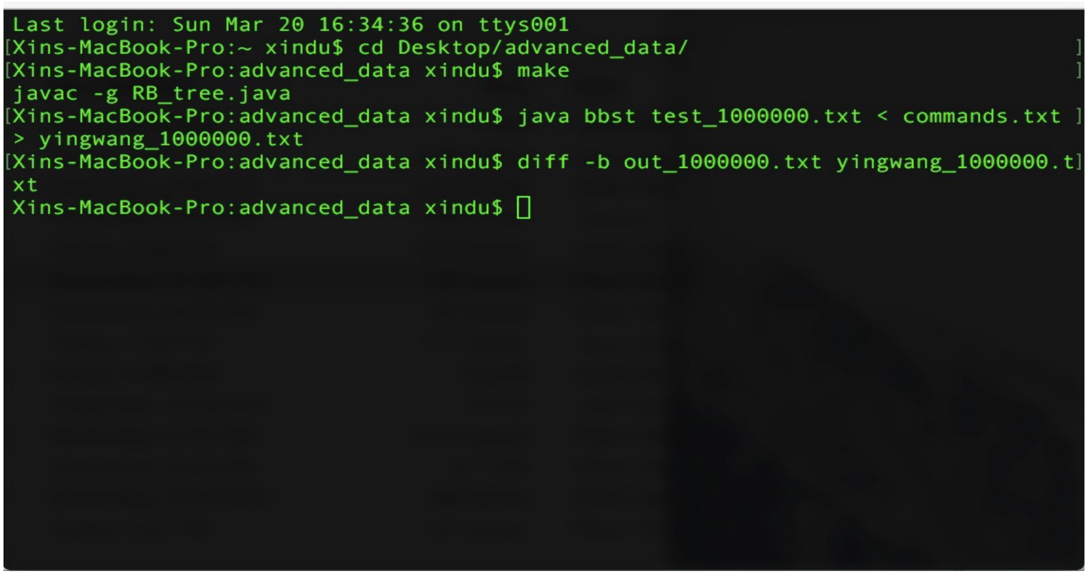


```
Last login: Sun Mar 20 16:38:38 on ttys000
[Xins-MacBook-Pro:~ xindu$ cd Desktop/
[Xins-MacBook-Pro:Desktop xindu$ cd advanced_data/
[Xins-MacBook-Pro:advanced_data xindu$ make
javac -g RB_tree.java
[Xins-MacBook-Pro:advanced_data xindu$ java bbst test_100.txt < commands.txt > yi
ngwang_100.txt
[Xins-MacBook-Pro:advanced_data xindu$ diff -b out_100.txt yingwang_100.txt
Xins-MacBook-Pro:advanced_data xindu$
```

Figure19 Load bbst Class and Test the Output of 100 Nodes

C. test_1000000.txt Test

In a similar way, I test whether the output of my program with 1000000 nodes has the exactly same output as out_1000000.txt. The Figure shown below indicates there is no difference between the output of my program and the correct answers.



```
Last login: Sun Mar 20 16:34:36 on ttys001
[Xins-MacBook-Pro:~ xindu$ cd Desktop/advanced_data/                          ]
[Xins-MacBook-Pro:advanced_data xindu$ make                                   ]
javac -g RB_tree.java
[Xins-MacBook-Pro:advanced_data xindu$ java bbst test_1000000.txt < commands.txt ]
> yingwang_1000000.txt
[Xins-MacBook-Pro:advanced_data xindu$ diff -b out_1000000.txt yingwang_1000000.t]
xt
Xins-MacBook-Pro:advanced_data xindu$ 
```

Figure20 Test Output of 1000000 Nodes

IV. Analysis and Summary

A brief analysis of event counter operations:

Increase ():	O(logn)
Reduce ():	O(logn)
Count ():	O(logn)
Previous ():	O(logn)
Next ():	O(logn)
InRange ():	O(logn + s)

The test_100.txt finishes all operations within 1s, and the test_1000000.txt file completed all the operations between 0.8s – 1.2s, both satisfy the efficiency requirement.

In this program, I used Eclipse to compile java program. The project is a good overview of red-black tree. Through this project, I practiced the initialization of a balanced binary search tree, basic operations of binary search tree like search, insert, delete, find successor and predecessor, etc. The red-black tree is the same except that it has some unique properties such as nil nodes and nodes with different color. In addition, I get to know more about how to process an input file and how to compile and run a java file.