Advanced Data Structure

Programming Project

COP 5563 Spring 2016

Name: Nikhil Singh
UF ID: 6415-5048
Email: singh0520@ufl.edu

Complier used: g++

How to compile: *make*

How to run:

Without input/output redirection: *./bbst file_name*
Where file_name is input file name

With input/output redirection: *./bbst file_name < commands.txt > out.txt*
where < is used for input redirection and > is used for output redirection

Structure of Program:

- **int main():** *Int main* method will take input from the command line and store them in corresponding variables.
  It is also responsible for calling out every other functions like increase, reduce, count, inrange, next and previous.

- **Minimum(nod *):** *Minimum* function takes a root node of red black tree as an argument. It traverse the whole tree recursively and return node with the lowest value.

- **Maximum(nod *):** *Maximum* function takes a root node of red black tree as an argument. It traverse the whole tree recursively and return node with the highest value.

- **Search(nod *, int):** *Search* function takes root node and a integer value as an argument. Integer value is the value of the node to be searched. The function works in a recursive manner, it goes to LEFT node and RIGHT node, comparing at each stage whether it value is equal to the integer value taken as an argument. When the values matches, it return that node. Otherwise, EMPTY_POINTER is returned, which means that node is not present in the tree.

- **Rotate_left(nod *, nod *):** *rotate_left* function takes root node and a node x as an argument. In the function, we rotate node x to the left side of its parent in order to satisfy the red black tree condition that both the child of a red node is black. So, we check whether the PARENT or LEFT child of its PARENT is empty or node. Then we make a left rotate taking its PARENT as a center point. Resulting in the node becoming the PARENT and previous PARENT becomes its LEFT child.

- **Rotate_right(nod *, nod *):** *rotate_right* function takes root node and a node x as an argument. In the function, we rotate node x to the right side of its parent in order to satisfy the red black tree condition that both the child of a red node is black. So, we check whether the PARENT or RIGHT child of its PARENT is empty or node. Then we make a right rotate taking its PARENT as a center point. Resulting in the node becoming the PARENT and previous PARENT becomes its RIGHT child.

- **Insert_fix(nod *, nod *):** *insert_fix* function takes root node and a node x as an argument. In this function, all the red black tree conditions are checked and corrected. First, root node should always be black. Second, red node must always have both its child black. Using WHILE, IF and ELSE condition

changes to the position of the node are made to satisfy the red black tree conditions.

- **Insert_tree(nod *, char *):** *insert_tree* function takes root node and a char value a an argument. The char value stores the input file name which is to be open. We have used IFSTREAM function to open a file. After opening the input file, it first read and stores into an integer, the number of nodes to be inserted. After that it reads id and count of each node and insert into the tree. After each insertion, it checks whether the insertion of the node follows the rules of red black tree, then we use INSERT_FIX function.

- **Delete_fix(nod *, nod *):** *delete_fix* function takes root node and a node x as an argument. In this function, after deleting a node all the red black tree conditions are checked and corrected. First, root node should always be black. Second, red node must always have both its child black. Using WHILE, IF and ELSE condition changes to the position of the node are made to satisfy the red black tree conditions.

- **Delete_tree(nod *, int):** *delete_tree* function takes root node and int x as an argument. First, it uses SEARCH function to get the node with value equals to x. If there is no such value, then EMPTY_POINTER will be returned. Otherwise it will delete the node by replacing its pointer values to null and making the required changes in the tree using DELETE_FIX function.

- **Increase(nod *, int ,int):** *increase* function takes root node, value of the id whose count is to be increased and integer value to increase the count, as an argument. First, it uses SEARCH function to get the node with id value equal to the one taken in

the argument. Then, it increases the values of its count and prints the ID and COUNT. If, there is no such node, then we INSERT a new node with id and count into red black tree.

- **Reduce(nod *, int, int):** *reduce* function takes root node, value of the id whose count is to be reduced and integer value to reduce the count, as an argument. First, it uses SEARCH function to get the node with id value equal to the one taken in the argument. Then, it reduces the count value of that node. After every reduction it checks that whether the value of the count is less then or equal to zero or not, if so, then we call DELETE function to delete that node from tree.

- **Count(nod *, int):** *count* function takes root node and value of the node whose count is to be displayed. First, it searches a node with the id equal to the integer values and then prints its count value. If the node is not found then it prints “0 0”.

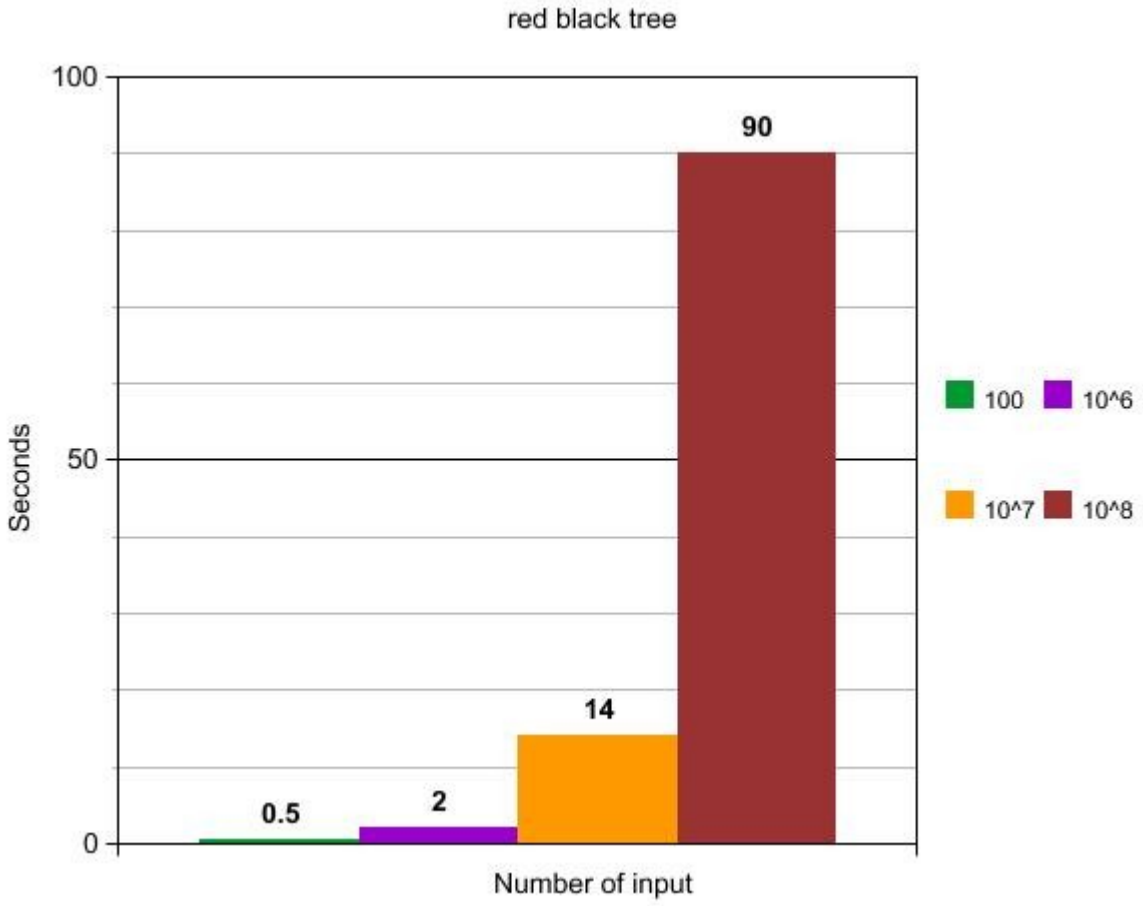
- **Inrange(nod *, int, int):** *inrange* function takes root node, first id and second id as an argument. It is a recursive function, in which it traverse the complete tree and checking at every point whether the id value is inbetween first and second id, inclusively, if so, then add its count value to a global variable. Print the value of the global variable in the end.

- **Next(nod, int):** *next* function take root node and id of the node whose next largest value is to be found. It is also a recursive function in which it goes to LEFT and RIGHT child, comparing the values of the node at it stage. It then returns the desired values in the end.

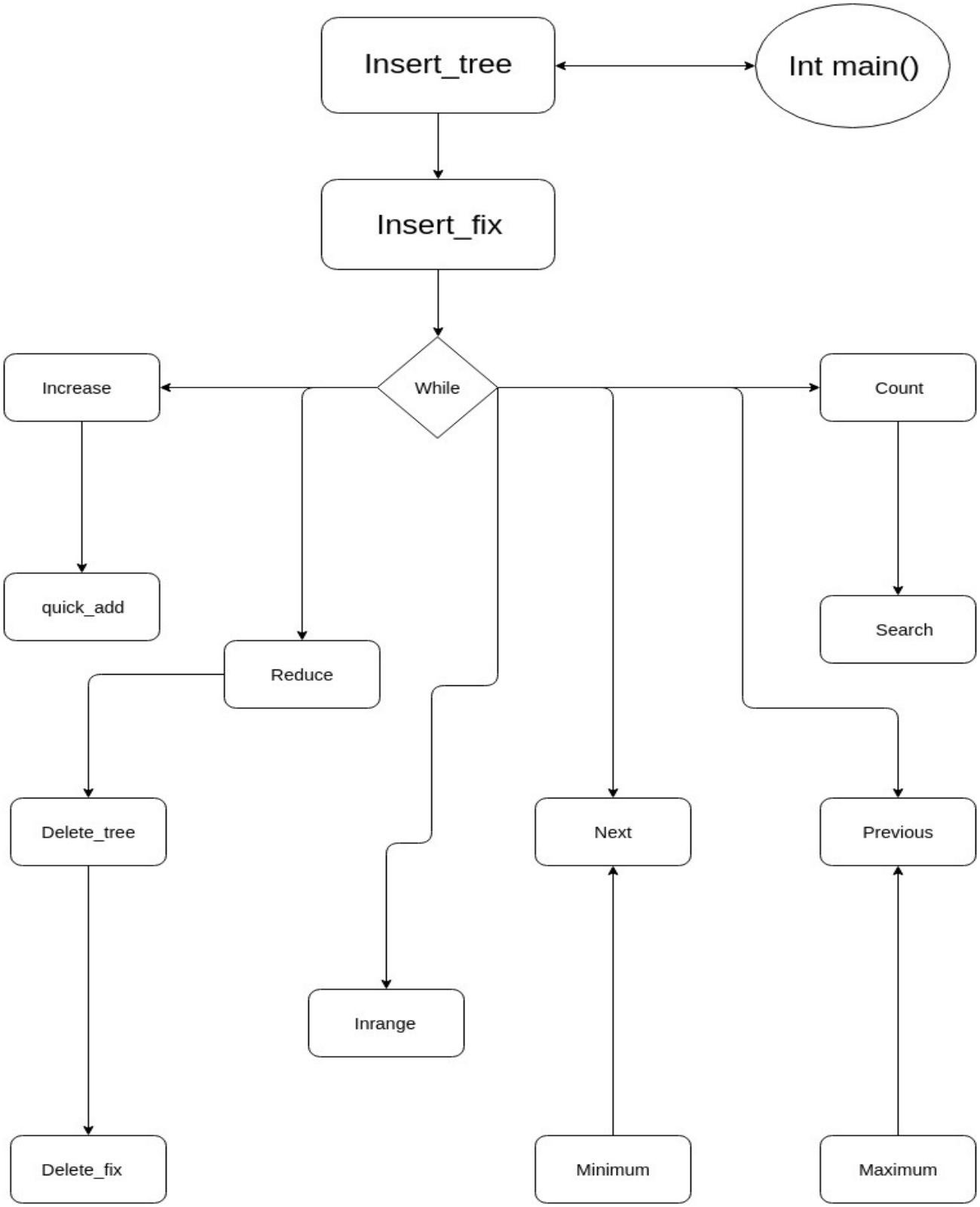
- **Previous(nod, int):** *previous* function take root node and id of the node whose previous largest value is to be found. It is also a recursive function in which it goes to LEFT and RIGHT child, comparing the values of the node at it stage. It then returns the desired values in the end.

Program perfromance:

1. Test_100.txt takes < 1 sec to insert and show output.
2. Test_1000000.txt takes around 3 secs to insert and show output.
3. Test_10000000.txt takes about 14 secs to insert and show output.
4. Test_100000000.txt takes about 90 sec to insert and show output.



Basic Program execution:



Conclusion and results:

- Red black tree was successfully implemented.
- All the values from the input file was inserted within O(n) time period.
- All the functions are working fine and with the required time period.
- Input and output redirection is working with the code.