Event Counter Using Red Black Tree

Submitted by:
Abhinav Rathi
UFID: 55935636

Table of Contents

Summary

Language Used: c++
Compiler Used: g++ (xcode)

All available test cases passed.

Running Time for test_100.txt file: <1 msec on local machine
Running Time for test_1000000.txt file: <1 sec on local machine
Running Time for test_10000000.txt file: around 5-6 seconds on local machine
Running Time for test_100000000.txt file: around 60-70 seconds on local machine

Files Description

Following files are provided as Source Code:

1. **Makefile**
   Compiles main.cpp, red_black_tree.cpp and creates executable file **bbst**
   Also use “make cl” to delete all object files and executable file bbst.

   Using Makefile we get our executable file as "bbst".
   We then execute by: ./bbst "file_to_read" < "user_commands_file" > "my_program's_output_file"

2. **main.cpp**
   Here we initially read the contents of a sorted input file and initialize the tree with it. Next we wait for User command line arguments to provide various commands and the corresponding output in each case is displayed, until the user inputs "quit".

   Command Line Functions available:
   increase id m
   reduce id m
   count id
   inrange id1 id2
   next id
   previous id
   quit

3. **red_black_tree.hpp**
   Here we define the structure of a “node” of Red Black Tree
   It contains:

Sl. No.	Name	Type	Description
1	ID	int	ID of Event Counter
2	count	int	No. of Events associated with the ID
3	color	char	‘b’ for BLACK; ‘r’ for RED
4	p	node*	Parent Pointer
5	l	node*	Left Child Pointer
6	r	node*	Right Child Pointer

We also define a constructor to initialize node with given ID and count, and to color it to ‘b’. Also pointers p, l and r are set to NULL.

Also the class definition of red_black_tree has following members:

Data Member:

Sl. No.	Name	Type	Description
1	root	node*	To hold the root of the Red Black Tree

Member Functions:

Sl. No.	Function Prototype	Description
1	red_black_tree()	Constructor (sets root = NULL)
2	void insert_from_sorted_list(int id[],int count[], int start,int end,int height)	Tree Initialization
3	int increase(int id,int m)	Increase count of "id" by "m", if present else insert
4	int reduce(int id,int m)	Reduce count of "id" by "m", if present else insert
5	int count(int id)	Return count of "id"
6	int inrange(int id1,int id2)	Total count of events between "id1" & "id2"
7	node* search(int id)	Search for "id" in Red-Black-Tree
8	node* search_next(int id)	Search next of "id", given "id" is not present in Tree
9	node* search_previous(int id)	Search previous of "id", given "id" is not present in Tree
10	void insert(node *t)	Insert single node "t" into Red-Black-Tree
11	void remove(node *t)	Removes node "t" from Red-Black-Tree
12	node* next(node *t)	Search for next node of a given node "t"
13	node* previous(node *t)	Search for previous node of a given node "t"
14	node* getroot()	Return root of Red-Black-Tree
15	node* sorted_insert(int i[],int c[],int s,int e,int h)	Recursive Tree Initialization (Sorted List)
16	int inrange_recursive(node *t,int i1,int i2)	Recursive function for tree traversing and counting
17	void display(node *t,int n,char c)	Displays Red-Black-Tree
18	int max_height(node *t)	Returns Maximum Height of Tree
19	void color(node *t,int h)	Colors all nodes black except last level (given BBST)

Note: Functions 17,18,19 are extra implementations.

4. **red_black_tree.cpp**
This class defines all the functions defined in the header file above. Short Description of each function follows:

- **red_black_tree::red_black_tree()**
Constructor: Initializes the root to NULL

- **void red_black_tree::insert_from_sorted_list(int id[],int count[],int start,int end,int height)**
Tree Initialization. Call to Function sorted_insert with arrays "id" and "count" to build the tree; "start" & "end" indicate the length of list to be put in the tree; "height" represents the maximum height of the tree

- **node* red_black_tree::sorted_insert(int i[],int c[],int s,int e,int h)**
Sorted_insert: Recursive function to build tree and return its root

- **int red_black_tree::increase(int id,int m)**
  User Command: Increase count of "id" by "m", if present else insert

- **void red_black_tree::insert(node *t)**
  Insert: Insert single node "t" into Red-Black-Tree with "root"
  All cases discussed are exactly what was discussed in class. The appropriate reference is given.

- **int red_black_tree::reduce(int id,int m)**
  User Command: reduce count of "id" by "m", if present (removes node if count becomes 0 or less)

- **void red_black_tree::remove(node *t)**
  Remove: Deletes node "t" from Red-Black-Tree with "root"
  All cases discussed are exactly what was discussed in class. The appropriate reference is given.

- **int red_black_tree::count(int id)**
  User Command: count of "id", if present we return thr count else we return 0

- **int red_black_tree::inrange(int id1,int id2)**
  User Command: inrange between "id1" & "id2", if present we all counts between these two id's (inclusive)

- **int red_black_tree::inrange_recursive(node *t,int i1,int i2)**
  Recursive function to sum up all counts between "i1" and "i2"

- **node* red_black_tree::search(int id)**
  Search for "id" in Red-Black-Tree

- **node* red_black_tree::search_next(int id)**
  Search next of "id", given "id" is not present in Tree

- **node* red_black_tree::search_previous(int id)**
  Search previous of "id", given "id" is not present in Tree

- **node* red_black_tree::getroot()**
  Function returns root of the red-black-tree

- **node* red_black_tree::next(node *t)**
  User Command: next of node t, return NULL if t is righmost leaf child

- **node* red_black_tree::previous(node *t)**
  User Command: previous of node t, return NULL if t is leftmost leaf child

- **void red_black_tree::display(node *t,int n,char c)**
  Function displays the red-black-tree as a stair-case (also lists count, color, parent and children)

- **int red_black_tree::max_height(node *t)**
  Function calculates maximum height of any binary tree

- **void red_black_tree::color(node *t,int h)**
  Function colors a Balanced Binary Search Tree according to red-black tree properties, given its maximum height h

Detailed description pertaining to each function is mentioned as Comments in the source code.

Structure of Program

Now, lets look at the Structure of our Program by Running through the following steps:

Step 1: Execute “make” at terminal window (our current directory is the folder containing source code files)

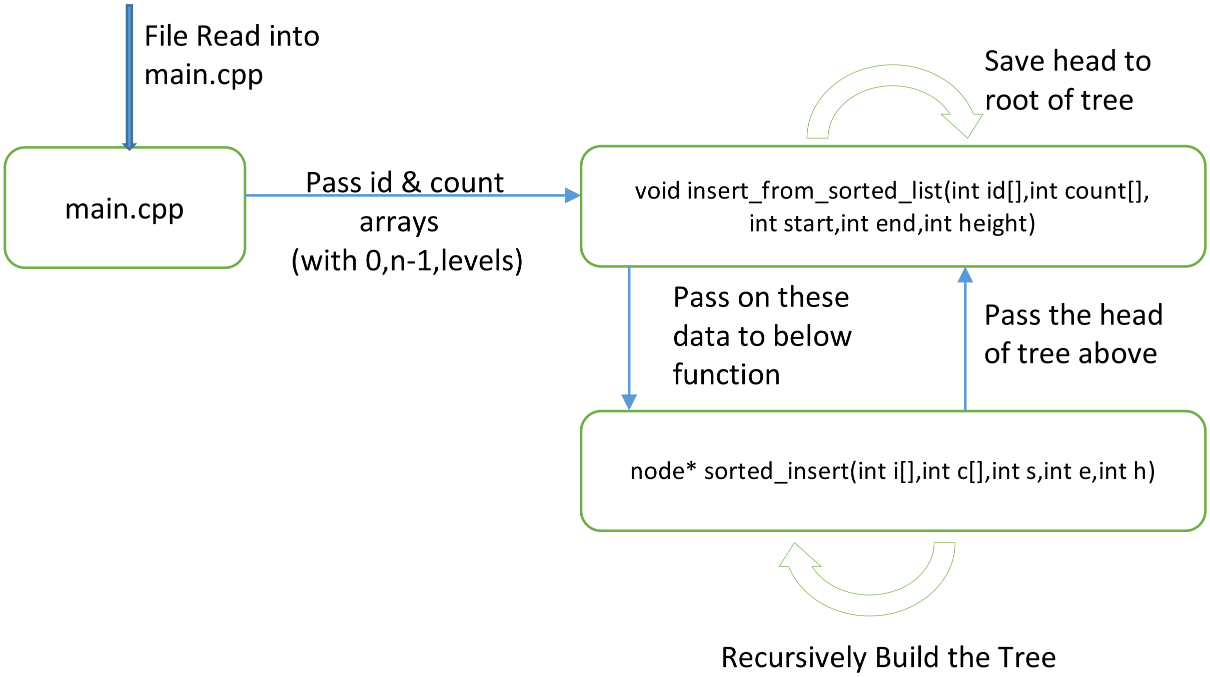
Executable file **bbst** will be created.

Step 2: Execute “./bbst input_file.txt < commands_file.txt > output_file.txt

Step 3: By executing ./bbst, main function will be called.

i) It will first check if “input_file.txt” is valid. If not, display error message and end program.
ii) Now we read the first line of file to store in n, the number of data points.
iii) We then compute levels for a Balanced Binary Search Tree, given n.
iv) Here on we initialize two arrays id & count (both of size n) by reading n sets of data from file
v) Now we call for a function in red_black_tree class using its object “T”.

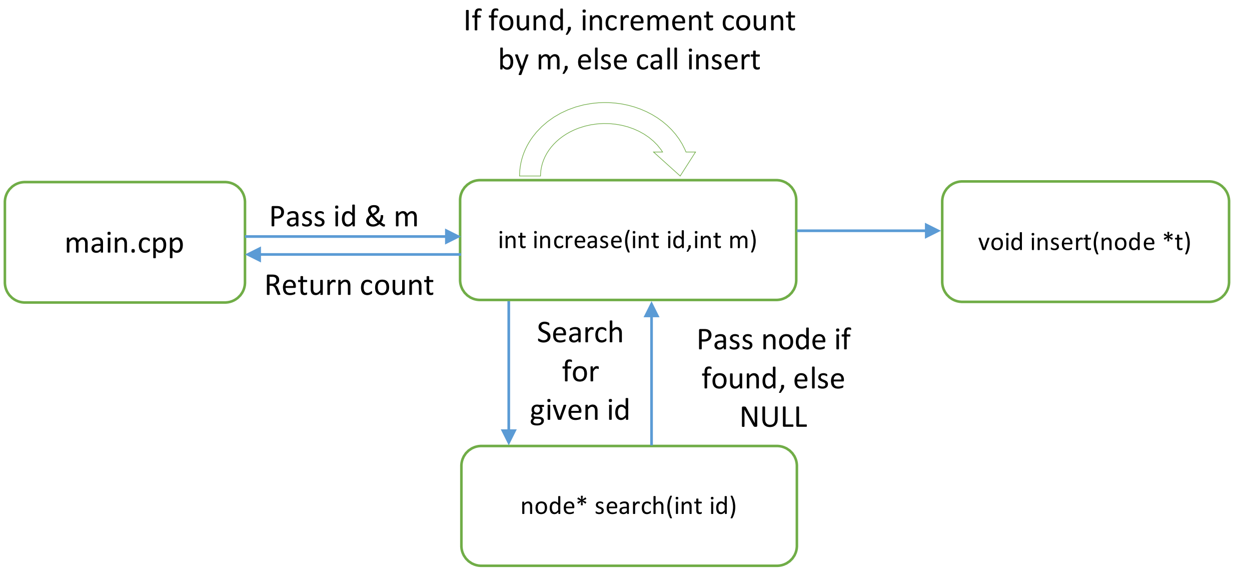
Initialization from Sorted List:


File Read into main.cpp
main.cpp
Pass id & count arrays (with 0,n-1,levels)
void insert_from_sorted_list(int id[],int count[], int start,int end,int height)
Save head to root of tree
Pass on these data to below function
Pass the head of tree above
node* sorted_insert(int i[],int c[],int s,int e,int h)
Recursively Build the Tree

The control comes back to main.cpp

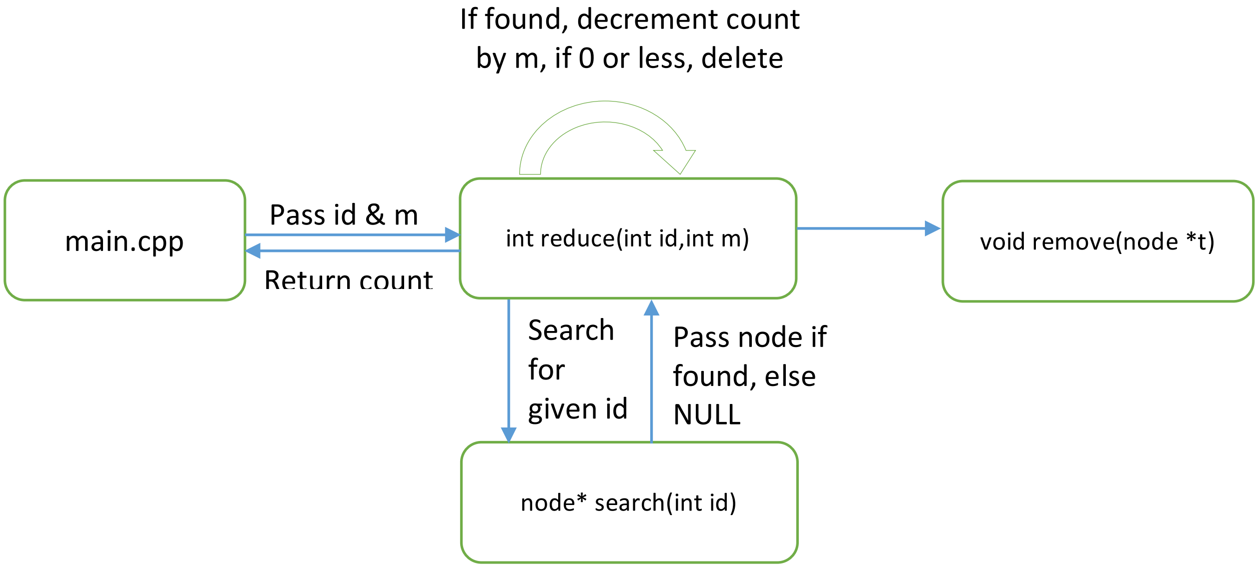
Step 4: Now we have various controls available for standard input to exercise following queries:

i) **increase id m**


If found, increment count by m, else call insert
main.cpp
Pass id & m
Return count
int increase(int id,int m)
void insert(node *t)
Search for given id
Pass node if found, else NULL
node* search(int id)

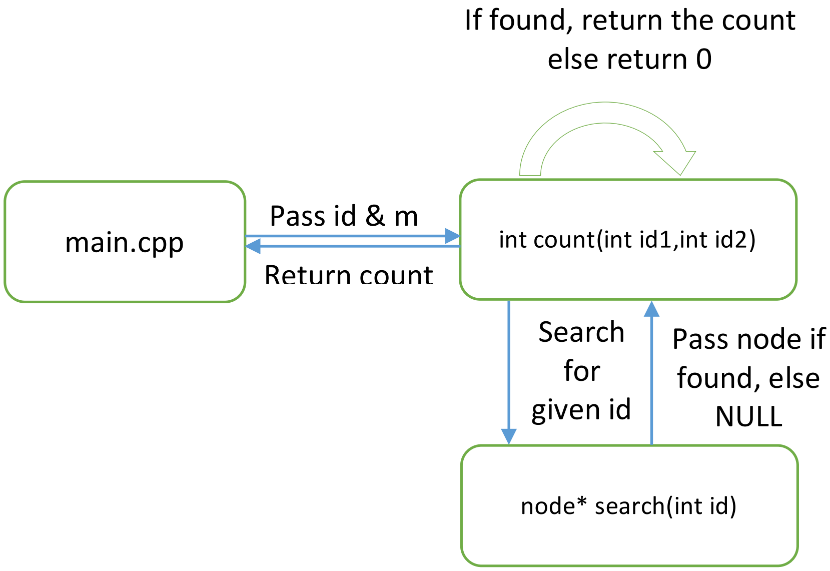
Increase function will return count plus m if ID was found, else return m after inserting.

ii) **reduce id m**

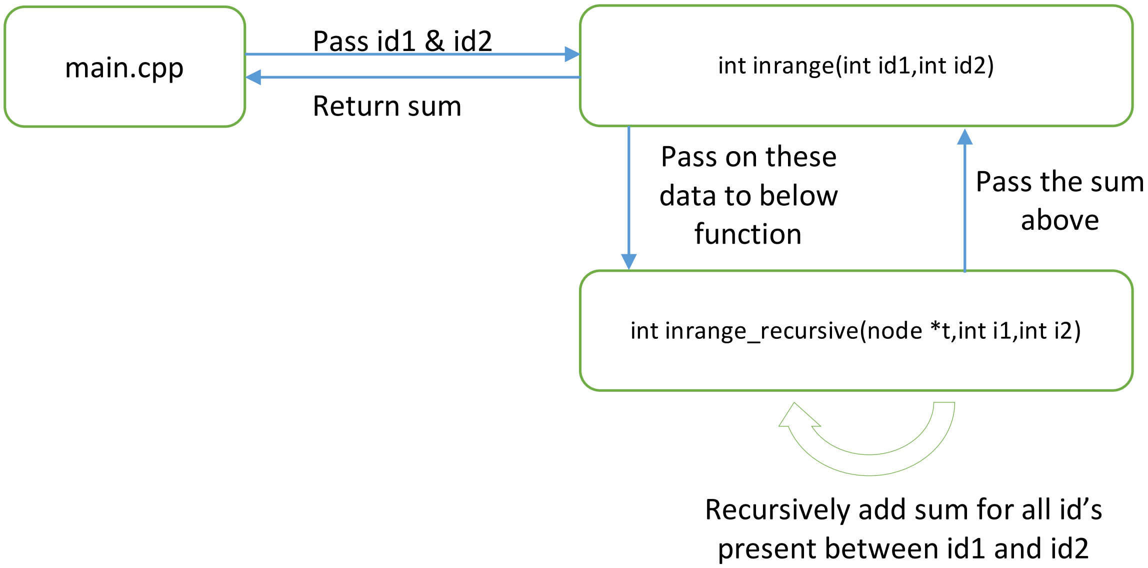


Reduce function will return count minus m if ID was found, else 0 (0 for deletion too).

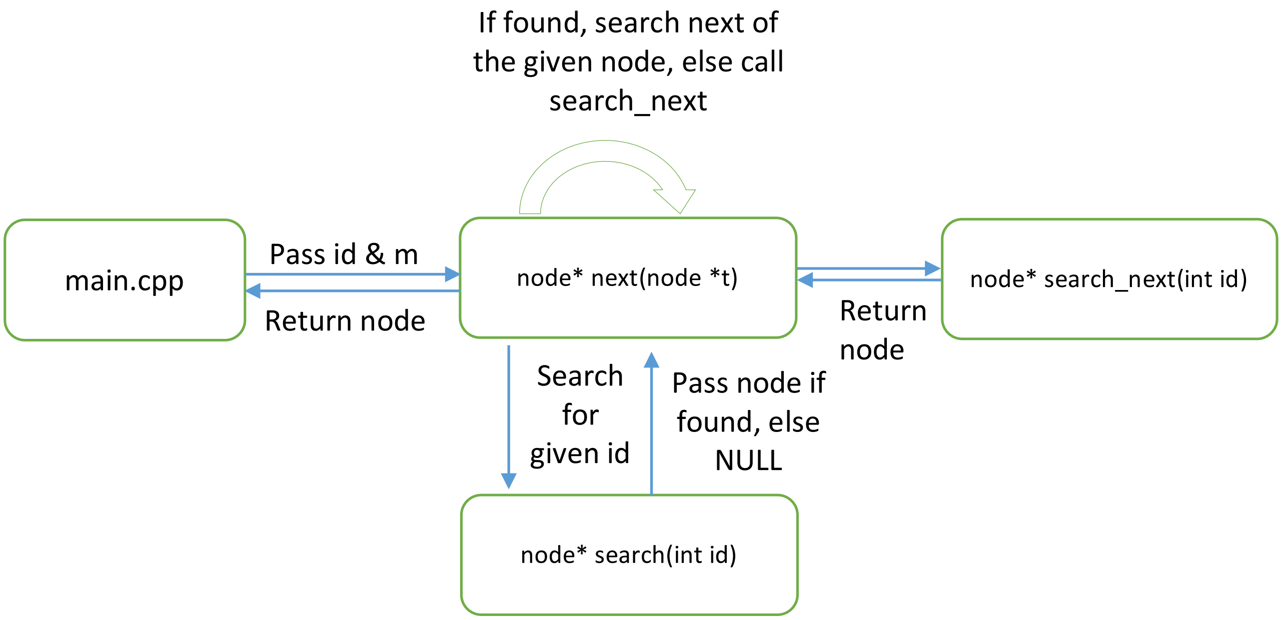
iii) count id



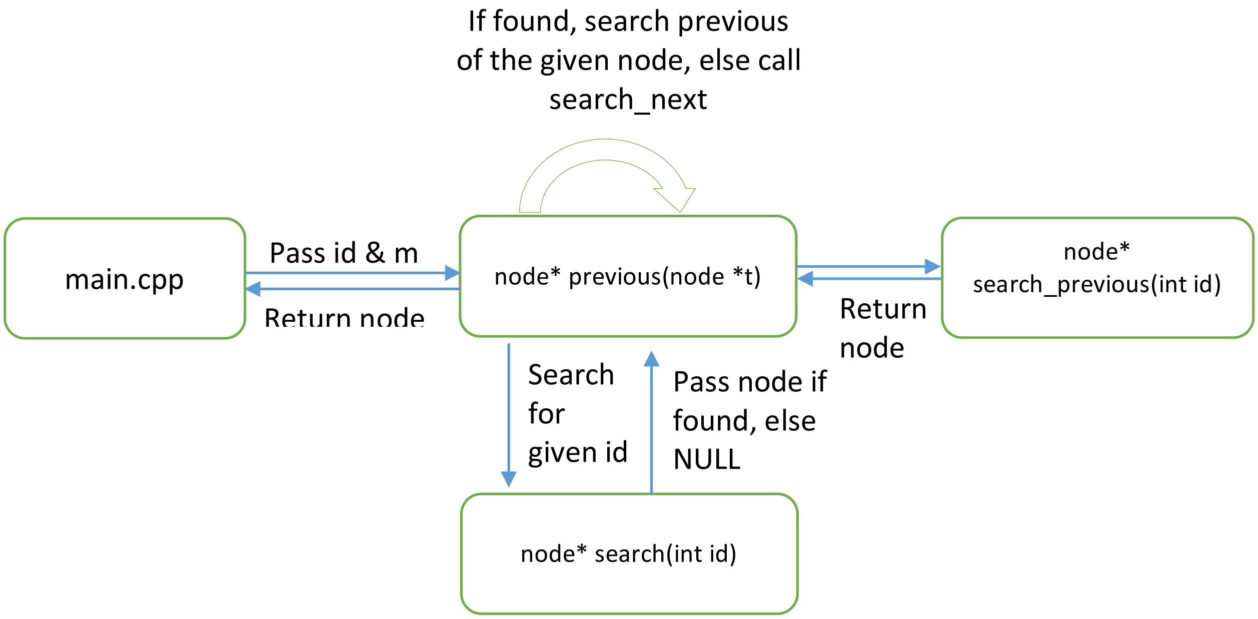
iv) inrange id1 id2


main.cpp
Pass id1 & id2
Return sum
int inrange(int id1,int id2)
Pass on these
data to below
function
Pass the sum
above
int inrange_recursive(node *t,int i1,int i2)
Recursively add sum for all id's
present between id1 and id2

v) next id


If found, search next of the given node, else call search_next
main.cpp
Pass id & m
Return node
node* next(node *t)
node* search_next(int id)
Return node
Search for given id
Pass node if found, else NULL
node* search(int id)

vi) previous id


If found, search previous of the given node, else call search_next
main.cpp
Pass id & m
Return node
node* previous(node *t)
node* search_previous(int id)
Return node
Search for given id
Pass node if found, else NULL
node* search(int id)

vii) quit

Quits the main.cpp and ends the program.

References

- Lecture Videos available on Canvas
- Presentation Slides available on **http://www.cise.ufl.edu/~sahni/cop5536/**