COP 5536 Programming Project

By
Rishab Goel

Problem description

You are 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. Your counter should support the following operations in the specified time complexity.

Solution:

Compiler and Programming used:
g++ compiler used to code the single C++ source file

Code Description:

The Code could be largely divided into three main parts:

1) `struct node` : A Structure to define the variables needed to create a node suited for a node in the Red Black Tree.
2) `class EventCounter`: It creates a class using RedBlackTree data structure with all the required functions.
3) `int main(int argc, char *argv[])`: It takes the Input Test file using File Handling and reads the commands file from standard input stream as well as outputs the desired results.

Detailed Description:

1) `struct node`
   - `char color :` To assign the Red or Black color of the node.(1 byte size)
   - `int id :` To assign the ID of the Event Counter inputted. This behaves as the key to search a node in Red Black Tee. (4 byte size)
   - `int count :`To assign the count of the Event Counter inputted. This behaves as the value stored in a Red Black tree Node. (4 byte size)
   - `node * left ,*right *parent :` These are node pointers to link with left child , right child and parent of the current node respectively. (12 byte size)

   Total size of each node = 21 bytes.

2) `class EventCounter`

`private` Methods`:` These functions are declared private as they need not be accessed by the world outside the class for the current problem description.

- `node* root` – To store the pointer to the root node of type `struct node` for the Red Black Tree .
- `int compare(int left, int right)` – This function compares left and right key values and return -1 if left less than right, returns +1 if left greater than right and returns zero when both are equal.

- `int computeRedLevel(int size):` This function actually computes the level at which the node while inserted should be set to red level. This level is actually equivalent the lowest level of the tree.
- `node* buildFromSorted(int level, int lo, int hi, int redLevel, vector<pair<int, int> > &idCountPairs, int &currentIndex)` – The function takes six arguments and is used to construct the red black tree from a sorted array in O(n).The level, lo and hi gets the current height from the root node, index of the lowest value and index of the highest key in the subtree respectively. idCountPairs is the vector of pairs used to store the input file sorted keys with values during the file read and is passed as reference. The `currentIndex` is passed as a reference as it updates each iteration and used to iterate through the input vector for accessing each ID and count.

The buildfromSorted is build by doing postorder traversal through the BST and left and right node followed by parent node gets built from top to bottom. As the bottommost leaf would become red to ensure black nodes keep same from external node to each node. The middle node of each subtree is returned and finally leading upto returning the root.

- `node* grandparent(node* n)` – returns the parent of the parent of the current node if present.
- `node* sibling(node* n)` – returns the left or right child of the parent if the node is right or left child respectively.
- `node* uncle(node* n)`- returns the left or right child of the grandparent of the current node if the parent of node is right or left child respectively.
- `char nodeColor(node* n)` – gets the color of the node. 'R' for RED and 'B' for Black.
- `node* newNode(int id, int, char color, node*, node*)` – takes the id, count , color to create the node. It also receives the left child pointer as well as right child pointer of the current node. It updates the parent pointer of the left and right child of the current node created.
- `node* maxNode(node* root)` – It recursively iterates to get the rightmost node which is the maximum ID of the Red Black Tree.
- `void replaceNode(node* old, node* cur)`- receives the pointer to the old and new value. Updates the node if it's the root otherwise checks whether left or right child of its parent and replaces the current node accordingly. It is followed by updating the parent pointer of the inserted node.
- `node* search(node* n, int id)` – takes root node initially and id to be searched. It recursively searches the node each time checking whether it is left subtree or right subtree. If the id matches finally then the node is found and it is returned, else it returns NULL.
- `node* next(node* n, int id)` – takes root node initially and id whose next need to be searched. It id is less than current it goes left else right until the right child becomes NULL. It traces back and stops at node where current node is actually its left child of its parent. If it stops the node which is the left child of its parent, then parent is the next id. If the current node is the root then the next id of the desired does not exist in the tree.
- `node* previous(node* n, int id)`- takes root node initially and id whose next need to be searched. It id is greater than current it goes right else left until the left child becomes NULL. It traces back and stops at node where current node is actually its right child of its parent. If it stops the node which is the right child of its parent, then parent is the previous id. If the current node is the root then the previous id of desired key does not exist in the tree.
- `long long int inrange(node* cur, int k1, int k2, long long int sum)` – receives the root nodes pointer and lower and higher id with an initial sum value zero. It compare the in range IDs with current node. If the node is greater than ID1 call recursively this function return the count sum. If the node with ID in between the ID1 and ID2 is found then add its count value. Then continue the search for ID which is lesser than ID2 recursively return the count sum.
- `void rotateLeft(node* cur)` – receives the node pointer. The current node is replaced by its right Node. It then updates the right child of the replaced node by the left child of the new node. It

updates the parent of new node's left child as the replaced node. It replaces the new node's left child with the replaced node and finally making the new node the parent of the replaced node.

- `void rotateRight(node* n) --` receives the node pointer. The current node is replaced by its left Node. It then updates the left child of the replaced node by the right child of the new node. It updates the parent of new node's right child as the replaced node. It replaces the new node's right child with the replaced node and finally making the new node the parent of the replaced node.
- `void insertFixup(node* n) –` receives the newly inserted node in the red-black tree. This function alters the nodes of the tree to satisfy the red-black tree properties. This can be divided in five cases:
    - The node is the root node, i.e., first node of red–black tree
    - The node's parent is black.
    - The node parent and uncle are red
    - The node is added to right of left child of grandparent, or the node is added to left of right child of grandparent (parent is red and uncle is black)
    - The node is added to left of left child of grandparent, or the node is added to right of right child of grandparent (parent is red and uncle is black)
- `void deleteFixup(node* n) –` receives the node which needs to be eventually deleted. This node so received from the remove function will have at least one null child. This function does alterations in the tree to satisfy the red-black tree properties. This can be summarized in the following cases:
    - Node is root.
    - The sibling of node is red.
    - Parent is BLACK, sibling is black and both children of sibling are black.
    - Parent is RED, sibling is black and both children of sibling are black.
    - Node is the left child of parent, sibling is black, left child of sibling is RED, and right child of sibling is black
    - Node is the right child of parent, sibling is black, right child of sibling is RED, and left child of sibling is black.
- `void verifyProperties(node*) –` It is just a dummy function which just indicates after fix up the properties of red black tree should be satisfied. Since we follow the standard algorithm we don't need to explicitly verify these properties.

`Public Methods:` that need to be accessed by the main method.

- `EventCounter(vector<pair<int, int> > &idCountPairs) –` The constructor of the EventCounter is called which calls the buildFromSorted() to create the RedBlack Tree in O(n).
- `EventCounter() {} –` The default constructor for event counter class.
- `int insert(int, int)-` The insert method is called with id and count both so that if the id is found it updates it count value or else keep on searching until find a suitable NULL node where it has to be inserted. At each stage it compares whether it is less than current node or greater than and goes to left or right subtree respectively for further search. This is equivalent to inserting a node in a binary search tree, after this we need to satisfy the properties of red-black tree. This is taken care by insertFixup().
- `void remove(int) –` This function gets called from the main function using the id. The id is first searched, then deleted if found. If the node has both the left and right children, then the values(id, count) of its inorder predecessor are copied to this node and node to be deleted becomes the predecessor node. With this, the node to be deleted will always have fewer than two children (will have utmost 1 children). Also, if this node to be deleted has color red, we simple replace it with its child. If not so, we call delete fixup. Here logically the RedBlackTree properties should be checked, that is why the dummy funcitonn verifyProperties() is called.
- `node* search(int)-` This function receives the ID from the main function and calls the recursive private search by sending the root node and returns the node pointer.
- `node* next(int)-` This function receives the ID from the main function and calls the recursive private search by sending the root node and returns the node pointer.

- `node* previous(int)` - This function receives the ID from the main function and calls the recursive private search by sending the root node and returns the node pointer.
- `long long int inrange(int, int)`- This function receives the IDs from the main function and calls the recursive private inrange by sending the root node.

3) `main(int argc, char *argv[])`

- In the main() we read the input file using ifstream using the file name from the argument and read it into a vector and sends that vector to EventCounter class to construct a tree.
- We close the file and delete the vectors created to temporarily store values.
- We separate the command and its arguments.
- We compare the commands and call the respective required function.
    - Command = “increase"
      Call rbt->insert(id, count) and output the count value.
    - Command =”reduce"
      Lookup the node using rbt->search(id) if found then decrease value which if less than equal to zero then delete the node using rbt->remove(id) call and print the count value.
    - Command = "count"
      Call rbt->search(id) and print the value 0 if NULL and else the actual count value.
    - Command = "inrange"
      Get the lower bound and upper bound id and call rbt->inrange(id1, id2) and output its return value.
    - Command = "next"
      Call rbt->next(id) print its return value if found else print zero.
    - Command = previous"
      Call rbt->previous(id) print its return value if found else print zero.

- We exit the while loop when “quit” command is read.