Name: -Avnish Sengupta

UID: -705299746

Data structure implemented: -

I used an open Hash table where every bucket was linked to Binary Search Tree to halve the time it would take to locate the desired node. The name of the class is hashtable\_stringtoinfo.

I made a Node struct that consisted of a string value and a vector of indices where that string value appears in the Info vector. The buckets of my hash table were just root pointers to the Binary Search Trees made up of these nodes.

I used two vectors- one that kept track of the Info structs, and the other kept track of the info structs consisting of users in chats that left. I made two separate hash tables for both the vectors- one that mapped users to their indices of the respective vector and the other mapped the chat names to the indices of their respective vector.

I used the Info struct provided to us.

The hashtable\_stringtoinfo class consisted of the following functions: -

hashtable\_stringtoinfo(int buckets);

~hashtable\_stringtoinfo();

vector<int>\* search(string name);

bool searcher(string name);

void insert(string name, int indices);

void remove(string name);

unsigned int mapfunc(string name);

void freetree(Node\* root);

The searcher function traversed the Binary Search tree and returned false the moment it hit a leaf node whose string value was not equal to that of the argument.

Pseudocode: -

//map the argument to get the bucket index

//if the bucket is not used

//while we don’t hit a leaf node.

//from the headpointer

//if the name in the node is equal to the argument, return true

//if the argument is lesser than the current node value, go left.

// if the argument is greater than the current node value, go right.

//if you exit the loop, return false.

//if the bucket is used, return false.

The search function was identical to the searcher function but instead of returning a bool value, it returned a pointer to the vector of integers, the target node held of the indices where the Info with node’s value is present in the Info vector.. The searcher function was used to first verify if the value is present in the Info vector.

The insert function inserted a new node if it was not initially present in the Hash Table.

Pseudocode: -

//map the argument to get the bucket index

//if the mapped bucket is not used or the headptr does not point to a nullptr

//create a new node and push the indices argument into the index vector.

//else

//if the target string is present on the node, make a new node and pass the indices argument to the index vector.

//if the name is smaller, go left and if there is no left child, make a new node.

//if the name is larger, go right and if there is no right child, make a new node.

The remove function removed a node from the Binary Search Tree attached to a mapped bucket.

Pseudocode: -

//if the bucket is not used, return.

//else, from the headpointer use a parent pointer and a child pointer.

//while the child pointer is not a nullpointer

After this it was executing the deletion of nodes regarding the three cases of a leaf node, or one-child node, or a two-child node.

Freetree helped clear the tree after and destroy it recursively.

The functions I executed for ChatTrackerImpl: -

void join(string user, string chat);

//if the user is present in the vector: -

//traverse the vector of indices of the user node and traverse the actual vector using these indices only.

//Once the chat part of the Info is equal to the argument, erase that node from the vector and append it again.

//Set your indices accordingly.

int terminate(string chat);

//if the chat and user are present in their hash table, then

//walk through the user node index vector, and use the indices to find its different instantiations on the info vector.

//total the count only if its chat is equal to the argument.

//erase these indices from the chat and user index vectors after walking is complete.

//walk over the userswholeft chat and user hash tables and do the same thing.

//return total

//else return -1

int contribute(string user);

//if the user actually exists in the user hash table, then: -

//take the last element of the user node index vector and use this index to map the corresponding value on the info vector.

//add the count that this info struct holds.

//return count.

//else return 0.

int leave(string user, string chat);

//if the chat and user exist in their hash tables-

//walk the corresponding chat and user index vectors and find ones that are equal to each other.

//if they are equal erase them from both the vectors.

//push back this value’s corresponding info struct on the info vector to the userswholeft vector and erase it.

//else, return -1.

int leave(string user);

//if the user exists in its hash table

//get the last element of the user node index vector, map it to the corresponding info struct on the info vector.

//append this info struct on the userswholeft vector and pop it from the normal info vector.

//return count.

//else, return -1.

Problems I faced: -

Making the hash table proved more challenging and time consuming to me than I originally thought, especially since I was attempting it with a Binary Search Tree. I frequently tried to access the values of nullpointers.

Trying to iterate for loops a fixed number of times. State variables helped considerably.

I used a hash table that mapped the string value of either the user or the chat to its index value in the vector. However, I ran into an efficiency issue when everytime I erased an element, the indexes of all the subsequent elements shifted and I constantly readjust them. Making a vector of the indexes, and removing the indexes that have left the vector from them instead of the info struct itself, helped stabilizing the index values.