Jason Tieu

304047667

Project 4 Report

***MultiMap::findEqual()***

The findEqual() function takes in a string argument and checks the MultiMap for a key matching the string. It returns a MultiMap::Iterator if it finds an association, and an invalid Iterator if it doesn't. We were required to make it run averge O(logN) time and worst case O(N) time, assuming that there are N elements in the MultiMap.

For this function I decided to create a recursive function for it to call. The following is the pseudocode for the recursive function I made for findEqual() to call.

MultiMap::Iterator MultiMap::findEqual(std::string key) const

{

return binarySearch(m\_root, key);

}

MultiMap::Iterator MultiMap::binarySearch(Node\* p, string s) const

{

//base cases

if p is a null pointer

return an invalid Iterator (default constructor)

if p's key is equal to s

return an Iterator to p

if s is greater than p's key

call binarySearch on p's right child (still using s)

else s is less than p's key

call binarySearch on p's left child (still using s)

}

For binarySearch(...), assuming that the MultiMap holds N items, the time complexity is on average O(log N). The if-statements perform at constant time, O(1), since they are just checking to see if the Node pointer is null or not. When we do a recursive call on binarySearch, we are splitting the elements searched, N, by half every time since this is a Binary Search Tree and we can narrow the elements based on comparison. If the MultiMap were in sequential order, then this function would have time complexity of O(N), or linear time. This is because each Tree node would only have one children (since it is all in order). This would cause the recursive calls to have to visit the elements linearly, like a linked-list.

***MultiMap::Iterator::next()***

The next() function gives the ability to move to the next Tree Node in sequential order. Since I am using the method where the values are stored in a linked-list, this function allows traveling through a value linked-list as well. For example, if the nodes in a BST are ("hello",2), ("bye",203), ("hello", 202), then my BST tree structure would be:

hello:2,3 hello:2 hello:2

/ / / \

bye:203 **instead of** bye:203 **or** bye:203 hello:3

\

hello:3

We were required to make it run averge O(logN) time and worst case O(N) time, assuming that there are N elements in the MultiMap.

The psuedo-code for next() is shown below:

bool MultiMap::Iterator::next()

{

if invalid

return false

if next value in value linked list is not null

next() moves to next value

return true

if no right child

if tree node less than the Root node

set to the parent's parent

return true

if tree nodes greater than Root

if parent is larger

next in sequence is the parent

if parent is smaller

node is the last in the BST

invalidate Iterator

return false

if there is a right child

the next in succession is the left most child of the next right child

return true;

}

For MultiMap::Iterator::next(), the time complexity is O(logN) assuming that there are N elements in the MultiMap. This is because the decision on which is the next Node is halved every time an if-statement is true due to the structure of a Binary Search Tree. At worst case, it would perform at O(N), linear time if the BST was in already formatted in sequential order since it would have to traverse the tree as if it were a linked-list.

***Database::search()***

The search function is used to search all rows that satisfy criteria passed in via the searchCriteria parameter. All rows that match the criteria then must be sorted in a particular order determined by the sortCriteria parameter. The function passes a vector of ints (results) by reference. After finding all the rows that satisfy the searchCriteria and sorting them, we set the passed into vector to the row numbers that have been found. The function returns an int representing the size of the vector of ints.

I encountered some difficulties with this part. I was able to get the searching and sorting to work properly, but due to time constraints, I could not test why there were cases where I would not be able to get the results from search(). I believe that my search() method fufilled the time complexity requirements. To determine all rows that met a single search criterion, I was able to make it run in average case O(M log N), assuming that the database holds N rows, and there are M matching items. I was also able to determine which rows meet all criteria at O(CM log N) time, assuming that the database holds N rows, there are M matching items, and there are C search criteria. As for sorting, I was able to make it run in average case O(SR log R) time, assuming R is my query results and S is the number of sort criterias. I made this possible by using the sort method from the algorithms library, it runs at O(R log R) time, assuming R is my query results.

The pseudo code is shown below:

bool Database::search(...)

{

...

if searchCriteria is empty

clear results vector

return ERROR\_RESULT

...

repeatedly:

...

if current searchCriteria min & max are ""

clear results vector

return ERROR\_RESULT

repeatedly:

go thru schema to find index of search criteria

if search criteria field name doesn't exist in schema

clear results vector

return ERROR\_RESULT

repeatedly:

if search criteria min and max aren't ""

if first search criteria

put satisfying row# into a vector

if not first search criteria

put satisfying row# into diff vector

combine both vectors to get combined results (intersection)

if sortCriteria is empty

set results equal to combined results

return size of combined results

...

repeatedly:

search thru schema to find index of sort criteria

if sort criteria doesn't exist

clear results vector

return ERROR\_RESULT

repeatedly:

push searched row fields onto a vector

call sort\_function (detailed below)

make a new vector of row#s for final results

set results to final results

return final results size

}

void Database::sort\_function(...)

{

sort a vector of Row fields based on ascending/descending

check for any fields that are equa

if there are fields that are equal

make a subvector for the fields that are equal

call sort\_function on subvector

}