Kalin Khemka

304336969

Report Project 4

**MultiMap::findEqual()**

Basically, this function traverses through the multimap with a new pointer to the root node. This new pointer starts at the root node and then it is checked whether it is a nullptr. If it isn’t a nullptr, it compares the key of the temp node with the key that is trying to be found. If the key parameter is less than the key of the node, then the node is moved to the left child. If the key parameter is greater than the key of the node, then the node is moved to the right child. If the key parameter is equal to the key of the node, then an iterator is set to that node and the node moves to the left (this is because if there is another node that is equal in value to the node that has been checked, it will be always be on the left subtree of that first node).

This function runs an average of O(logN) times, where N is the number of nodes in the binary tree. The function can run up to O(N) though if the tree is very imbalanced. The reason for this is that every time you choose to go down the left or the right subtree, you eliminate 50% of the choices (in a balanced tree) (on average if not balanced).

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

The next function finds the next node in the binary tree through in-order-traversal. Basically, two node pointers are created to traverse this binary tree. If the right subtree is null, then it checks the top subtree. If there are parent nodes, then a bool function is triggered to call the function to keep going up the parent nodes till it finds the first parent node bigger (At the same time it checks, using the second node pointer, that the child node that you just came from was on the left side and not the right. If it was from the left, then you know that you have hit the next node.) If there was a right subtree however, then it moves into the next node on the right and trips a bool that makes it check all the way down the left subtree of the new node. Once it finds the leftmost node of the new node, then it stops there as that is the next node. Finally, if the node pointer points to the nullptr at the end, then it returns false.

This function runs on average O(logN) times where N is the number of nodes in the tree because as you take one path, (left or right), you are eliminating 50% of the tree that you have to traverse. At the same time, most of the times the function will run on O(1) because the next node may be literally the next node in the tree.

**Database::search()**

The search function was definitely the most fun function of them all… The search function starts with 2 main loops. The first loop goes through each of the searchCriteria and while it is doing that searches through the schema and matches the schemes with the searchCriteria. The second main loop goes through each vector, adding the row number of the rows that match the searchCriteria to an unordered set. The first time this second loop runs, it adds all the number to the unordered set. Every subsequent time it runs, it attempts to add the values to the unordered set and if it is unable to, then it adds the values to a second unordered set. This gives the second set all the duplicate values. At the end of each iteration, I change the value of the first unordered list to that of the second because those are all the duplicate values that you need. By doing this for each schema, you will eventually get only the row numbers of the rows that match all the criteria.

This part of the function has a time complexity of CMlogN, where C is the number of criteria, M is the number of matching rows, and N is the number of nodes. The log N comes from the fact that when you go through each of the vectors looking for matching nodes, you go through and eliminate 50% of the choices every time. The C comes from the number of times you loop through the initial outer loop. The M however, I feel may be less, because by using unordered sets, we are having an average of O(1) when we do any actions with the set.

The sort part of the function is divided into separate work by three different functions. The search part gives you a vector of integers that stores the row numbers that matter. Then, taking this vector of integers, you quicksort it. Taking the quicksort algorithm, basically you find the pivot points and keep exchanging them while dividing the vector into smaller vectors until the vector is sorted. To sort the vectors properly, and not simply by number, I added another function which compares two rows based on the sortCriteria and returns which one should come first and which should come second (It loops through all the sortcriteria till a suitable one is found that can determine which one comes first and which one comes second). This is used by the quicksort to properly sort the vector. At the end, the vector should be arranged by the sortCriteria.

The time complexity for this function is exactly as the spec said it should be, SRlogR, where S is the number of sort criterion and R is the number of matching rows as given by the size of the results vector.