MultiMap::findEqual()

This function begins with a pointer to the head, then compares its key value to the one passed. If they’re equal it constructs and returns an Iterator corresponding to that node. If they’re not equal it checks if the node has a child on the appropriate side (larger or smaller), if so it tries again with that child, if not it uses the default Iterator constructer to return an invalid Iterator.

This function satisfies its Big O requirements: for a MultiMap with N Nodes it will run in average case O(log(N)) time, and even the worst case will run in O(N) time.

MultiMap::Iterator::next()

This function first checks that the Iterator is valid and returns false if not. Then it looks in the corresponding Node’s right subtree if it’s not null and goes left as long as that’s not null, setting the Iterator to the leftmost member of that subtree and returning true. If the right subtree is null, it then looks at each ancestor of the corresponding Node and checks if the Node is in it’s left subtree. If so, it set the Iterator to that ancestor and returns true. If it reaches a null parent while only being in right subtree, it sets the Iterator to invalid and returns false.

This function satisfies its Big O requirements since it never looks both at children and parents since the children with be closer if they exist. This allows it to traverse the tree in much the same way as findEqual() yielding average case O(log(N)) for N nodes, but O(N) for a horrifically unbalanced tree.

Database::search()

The search function clears the results, then makes sure that all the search criteria are indexed member of the schema by iterating through the criteria and the schema for each one, returning ERROR\_RESULT otherwise.

Then for each search criterion if it has a minValue specified it uses findEqualOrSuccessor() to find the first value checks for a specified maxValue. If there is one, it keeps going using next(), adding each value to an unordered map as long as the value’s not more than the max and the Iterator is valid. If no maxValue is specified, it keeps going all the way until the Iterator is invalid. If there’s no minValue, but is a maxValue, it does the same thing but using findEqualOrPredecessor() and prev() instead of findEqualOrSuccessor() and next(). If there’s no min or max, it instead returns an ERROR\_RESULT.

This meets the Big O requirement of average case O(M log(N)) time for M matching items and N rows because it simply uses the log(N) time Iterator functions a constant number of times for each of the M matching rows.

After doing the above for each of the search criteria, the function checks for the existence of each member of the first criterion’s results in each other criterion’s results, deleting each that doesn’t appear in all of them.

This is a O(1) operation because these matches are stored in unordered\_sets. Therefore this adds an O(CM) operation for C criteria and M matches, but since the above piece had to be run C times and was previously determined to be O(M log(N)) with N items, C\*O(M log(N)) + O(CM) = O(CM log(N)) as expected.

Lastly, the matches are sorted and placed into results. The function turns the vector of results into a heap by starting from the last position and going until the 0th, using a comparison function to determine if each element it exceeds it’s parent (the data at index (i-1)/2 where i is its index).

The comparison function checks if the order is ascending or descending, then if ascending returns whether the element of the first passed row corresponding to the first search criterion is greater than the corresponding value for the second row passed. If order is descending, it uses the less than operator instead. But before looking at either of those, it checks if those values are equal for the two rows, and if so recursively calls itself starting with the next criterion down the list.

If the element compared to its parent does exceed it, they’re swapped. After the heap is created, the function starts at the first element in the vector and swaps it into the position just before size (which is initialized to the size of the vector), then it checks if the size is great enough for that element to have children ((2\*i)+1 where i is its index), and if so notes that number as the position of it’s maxChild. Then it checks if the size is big enough for it to have a right child (one further) and if so, compares the values of those children, storing the location of the child that exceeds the other. Finally, it compares that child to the parent, and swaps them if the child exceeds to parent and on down to it’s new children to restore the structure to a heap, thus resulting in a sorted vector of results.

This piece also meets its Big O requirements: O(SR log(R)) for S sort criteria and R matching rows. Heapsort (which I’ve recreated from scratch here) runs in R log(R) time with a comparison that’s O(1), but each can be compared S times because the comparison calls itself if they match. Most are unlikely to be equal, but even if one in a thousand is, that will multiply the base Heapsort runtime by (S/1000) which is the same as multiplying it by S as far as Big O is concerned.