Report

DATABASE: LINKED LIST

Callum Drennan| DSA |

Introduction

This report will explain the implementation of the data-base using a singly linked list. It will go into detail on what a singly linked list is and why I chose to use it over other data basing structures, and the difference between them. It will also cover how the Linked List performs with big O notations and timing on the methods I used.

Data Structures

LINKED LIST:

A linked list is a way of storing information. The information is stored in a class called a node, the node holds the data your wanting to store and a pointer to another node. With this way of storing information, you can link one node to another forming a list of data that is connected to the next one. There are a few types of linked lists, one of these is a singly linked list where with the node there is only one pointer that points to the next node in the list. This is very similar to the doubly linked list which holds two pointers, one to the next node, and one to the previous node. This helps with finding data in a sorted linked list as you can check from ether end. These data structures are good for when you don't know how big or how much data you are adding, as both don't need to be set up before you start to add to them with a big O notation of one for adding to the front list this makes them very good at loading. The major criticism of doing this is that when you need to find something in them, as you must look at every node moving on to the next until you find the item you are looking; for this makes removing or modifying it a big O notation of n.

BINARY TREE:

A binary tree is another type of data structure, where there are two left nodes and a right node which hold the data, and pointers to its parent and its child's both left and right. It works very similar to a family tree, but there are only two children and one parent. This type of data structure has an add big O notation of 'n(log(n))', which makes it very fast at adding things to the tree. Finding things in the tree is also 'n(log(N))', which makes it very fast at returning the Keys in the database. This seems like a good way of implementing the database; however, it is complex to set up a run effectively.

HASH TABLE:

A Hash Table is another type of Data structure that employs an associative array. It uses hash codes to get a Key to store an item into a bucket, within that array. This makes adding items to the hash table fast and efficient with a big O notion of one on average; but with this way of storing items you can have collisions where the hash code of one item is the same as another item. In this case, it's common to implement a Hash Table of a linked list, meaning if there is a collision it will just be added to the linked list in the Hash Table. The Hash Table implemented in this way can make the Big O notations to O(N), because there could be more than one item in the Hash Table at that location.

Implementation:

For this Data-base I have implemented a singular Linked List for its ease of use, and maintenance. It can hold any number of Key Value pairs, and with an add method with a 'BigO(1)' this make it fast to add items. I did not implement a doubly Linked List, as the keys will not be sorted when going into the list; making the implementation of a doubly Linked List not worth it, because it will still be a BigO(N) to find items. What this structure lacks in speed, it makes up in simplicity, and ease of implementation.

In the Data-base I set up some methods to find the key, and change the value that relates to that key. As this will be the iteration over the whole Linked List, as this is a big O notion of N; so, with a smaller number of nodes in the Data-base this will take less time to find the key. But as n is larger; ie: 5 million, the time it will take for n to be found within the Link List will take far longer to locate the key.

I additionally created a method to remove at the key specified by the input, this will go through the Linked List looking for the key inputted. It also uses the method above to do this, so this will make it a BigO on N again meaning that smaller Linked Lists are better than larger. Once it finds the key, it removes the node that has that key within it. If, however the key it found is within the middle of the list, where there are multiple nodes connected to each other, it removes that node, and reconnects the list.

Another method that I set up, is a modification of this one; as again it iterates thought the list attempting find the key that you have asked for using the find key method. Once it finds the key located in the node, it allows you to change the value for another value.

As I explained earlier, all the above tasks are a BigO(n), as they must look at each node in the Linked List to find the key that you want.

The last method that I have implemented is adding to the linked list. It add to the front of the list, making it a BigO(1) it then connects the new first node to the old node.

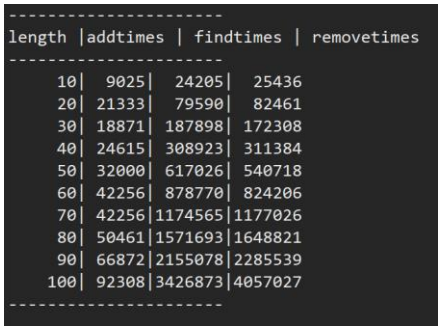
Expectations

With all the methods noted in the implementation, with most of them being BigO(N) we can expect that it will take some time to modify, remove, and find keys in a larger list. However, in a smaller list my expectations are that it will perform reasonably with times that are not inefficient – by taking long times to execute.

Results

In this section, I'll show you some tables on times that are measured in nano seconds, on how long it takes to do the methods stated in the implementation. I will also show tests working their way up to larger lists I've implemented.

This first one is working on list up to a size of 10000; so, this length in the table is multiped by 100 to get the results.



length	addtimes	findtimes	removetimes
10	9025	24205	25436
20	21333	79590	82461
30	18871	187898	172308
40	24615	308923	311384
50	32000	617026	540718
60	42256	878770	824206
70	42256	1174565	1177026
80	50461	1571693	1648821
90	66872	2155078	2285539
100	92308	3426873	4057027

As we can see here the times are very efficient. In a list this size we can see that the Linked List Data-base works very effectively. With the add times we can see that its increasing in time, but it's not by much, which reinforces the fact that it is a BigO notion of 1. In the find and remove times however; we can see that it is growing exponentially. This confirms that it is a big O notation of (N). But as we can see, there not overly large; meaning that it will work well with a list of 10000 nodes in size.

This next table will show a list up to the size of 1000000 in length, length here is again multiplied by 10000.

length	addtimes	findtimes	removetimes
10	291692	322640133	356790711
20	435693	1348170809	1450480184
30	486564	3422309917	3613752047
40	680616	6473554656	6592937268
50	18298264	8310874794	9201317006
60	976821	15979304196	16941748899
70	1273026	22546483917	23916167350
80	28528422	20830720341	22873209384
90	2049232	38592267320	40906406936
100	2820925	49340176139	49809447717

As you can see here the add times are large in time, but they do not grow exponentially. This reassures that it is still a bigO of (1), and again the find and remove confirm that it is a big O of (n). However, with this size of this Linked List Data-base we can see that the execution time is very inefficient, as the table above demonstrates.

With these two tables in comparison we can say that the larger the list is, the less effective the database becomes. But with smaller list it works in a reasonable time to be executed.

Conclusions

The Data-base that has been implemented as a Linked List, and as we can tell from the results we had: from the times it takes to do things grows fast, as the Data-base has more contained within it. We did expect this, but not a difference this large compared to smaller databases. So, it seems that Linked List Data-bases although simple to implement and works well is smaller sizes, becomes its downfall in larger lists making it a bad choice to choose for databases that you expect to grow to larger sizes exponentially.

A better choice for a Data-base of say over 100000 units would be a Hash Table. What the Hash Table gives in complexity it makes up for in efficiency. And to deal with the collisions that you will have in the Hash Table, is to give it a chaining of a Linked List. This will make the finding keys much faster, as there will be less nodes to literate over to find the key you want. And as your Data-base grows you can add a method that doubles the size of the Hash Table when it gets to a certain capacity, then reload your data to keep the efficiency of the Hash Table to workable times.