Linked List

Fredrik Berzins

Fall 2023

Introduction

So far we have only been working with primitive data structures and arrays. More complicated structures are better described as structures that are linked to each other using references, also called pointers or links. The reference is one-way so the object that has the property of course knows that it is referring to another object but the other object is unaware.

As an example a class that describes a person. The person will of course have a name, an address etc but it could also have a father and a mother. These properties could then be references to other objects rather than strings with the name of the parents. A person could of course also have an array of children where each child is a reference to another person object.

In this assignment we will look at simple linked structures and find out their properties rather than actually represent anything.

Linked structure explanation

A linked structure also called a node structure is a structure made from nodes and references to other nodes. Each node can have some information and some references.

Linked list explanation

A linked list is a node structure that has one known origin node and every node links to one other node, only the origin node is known the rest are only known by reference from the node before them. Each node can hold multiple data types of information. The last node in a linked list still has a reference but it's referencing a null object.

Implementation

Linked list implementation

The linked List is implemented by creating nodes with a null pointer from the beginning that will later be changed it also always has space for an int, this is shown in the node constructor method below.

```
private class Node {
    public int value;
    public Node next;

    public Node (int value) {
        this.value = value;
        this.next = null;
    }
}
Node first;
```

When doing benchmarks later the function that will be used is a append function that appends one linked list to the end of another linked list. When doing this the only change is the last cells reference (the null reference), this is changed to a reference to the origin node of the second linked list.

Array implementation

When doing append for an array it sums the length of both arrays creates a new array with the sum of the lengths and copy one number at a time.

Benchmarks

The testing was done with length between 100 and 25600 values(doubling each time, 100,200,400...), the fixed length for all test was 400. Each time test was redone 1'000 times witch included appending 1'000 different arrays to get as accurate results as possible. This gives a min time showing avg from the group with the 1'000 quickest runs.

The test in Table 1 is the to append a linked list/array to the end of a linked list/array, while the test int Table 2 is the time to create 1'000 linked lists/arrays of the first length and one of the second length. ("n+Fixed L", 1'000 n length lists and one fixed length list.)

Result

As shown below in table 1 the time to append a linked list to a linked list takes O(n) where n is length of first list, in comparison it takes the O(n) to

append to a array but in this case n is the sum of length for both arrays. When appending to a fixed length list it takes a consistent time, but if done to an array it will sit be dependent on the sum of the lengths.

In table 2 it's shown how the time differs when creating linked lists/arrays. It shows how creating a linked list and array is only dependent on length of the lists or arrays. This gives it both data types a big o notation of O(n) n being length of list/array when being cheated and filled with numbers.

n	**n+Fixed L**	**Fixed+n L**	**n+Fixed A**	**Fixed+n A**
100	417	1628	705	743
200	835	1628	862	912
400	1640	1643	1143	1187
800	3257	1633	1659	1746
1600	6505	1630	2758	2893
3200	12935	1633	4949	5215
6400	25903	1643	9457	9993
12800	51700	1641	17927	18091

Table 1: Shows time to append a fixed or a n length linked list/array to a linked list/array(ex: "n+Fixed L" means a Fixed length list is appended to a n length list.) in ns.

n	**n+Fixed L**	**Fixed+n L**	**n+Fixed A**	**Fixed+n A**
100	710	2796	2660	10377
200	1406	2797	5222	10378
400	2796	2808	10311	10383
800	5595	2816	20490	10396
1600	11224	2799	40757	10418
3200	22509	2824	81644	10455
6400	44939	2836	163758	10539
12800	139664	2901	325311	10699

Table 2: Shows time to create a linked list/array(ex: "n+Fixed L" means creating 1'000 n length list and one fixed length list.) in ns.

Conclusion

In conclusion a linked list is faster when appending since it only has to get to the end and change one reference. It' significantly faster to make and fill a linked list, compared to creating and filling an array of the same length. Since when making a linked list it always needs a value when adding the

node while an array is first created and then filled one value at a time. Both of these are down to the implementation of the code since both are O(n), n being the length of the list/array being created.

A good example of linked list being slower to create is if it's made all nodes consecutively this means you progress in the list as you add to it. Compared to adding each node to the end, this would make creating a linked list significantly slower, it would also end up with a big o notation of $O(n^2)$.

If a linked list structure was used for a stack it would be a space efficient dynamic array but since it always has to "resize" it will be slower in most cases but more consistent in time wise, this might be prefers since it wont have as bad of a worst case but it also wont be as quick in general. Since the linked list can be made to always add new nodes to the beginning it can be as efficient when poping values.