# Linked Lists

Pontus Hector

Fall 2022

## Introduction

This report is conducted as the fifth assignment in the course ID1021. In this report one explores the implementation of linked data structures in the form of linked lists and compare their efficiency with more primitive data structures.

## Linked lists

When storing objects in the form of linked lists, the objects are not stored at contiguous positions in the memory like they would be when stored in an array or a stack. Instead each element is stored with an individual pointer as a node, pointing to the memory location of the next node with the next element and pointer. In the first benchmark a List of varying size (n), filled with "dummy values", is linked with a list of fixed size with 100 elements. The following measurements were obtained:

| **Iterations** | **Array size (n)** | **Static array size** | **Lowest runtime** |
|---|---|---|---|
| 1000000 | 100 | 100 | 100$ns$ |
| 1000000 | 200 | 100 | 300$ns$ |
| 1000000 | 400 | 100 | 600$ns$ |
| 1000000 | 800 | 100 | 1100$ns$ |
| 1000000 | 1600 | 100 | 2400$ns$ |
| 1000000 | 3200 | 100 | 4700$ns$ |

Table 1: Linked list where the static sized list is appended onto the varying one

These measurements confirm that the linking method is of linear time complexity (O(n)). This is since the array of size n is iterated through, from the first element to the last element in which the pointer is manipulated to point to the first element of the second list.

```
while (nxt.tail != null) {
            nxt = nxt.tail;
```

```
    }
    nxt = b;
```

This means that if we swap which list is appended to which, one should achieve a different outcome in elapsed time and efficiency. The following measurements were obtained trough benchmarking:

| **Iterations** | **Static array size** | **Array size (n)** | **Lowest runtime** |
|---|---|---|---|
| 1000000 | 100 | 100 | $100ns$ |
| 1000000 | 100 | 200 | $100ns$ |
| 1000000 | 100 | 400 | $100ns$ |
| 1000000 | 100 | 800 | $100ns$ |
| 1000000 | 100 | 1600 | $100ns$ |
| 1000000 | 100 | 3200 | $100ns$ |

Table 2: Linked list where the varying sized list is appended onto the static one

These measurements confirm that the time complexity of the method is heavily dependent on the size of the list that is *appended onto*. To conclude these benchmarks one could argue that the time complexity of this method is in best case scenario constant (O(1)) and in worst case (but also general case) scenario, linear (O(n)), all depending on the size of the array appended onto.

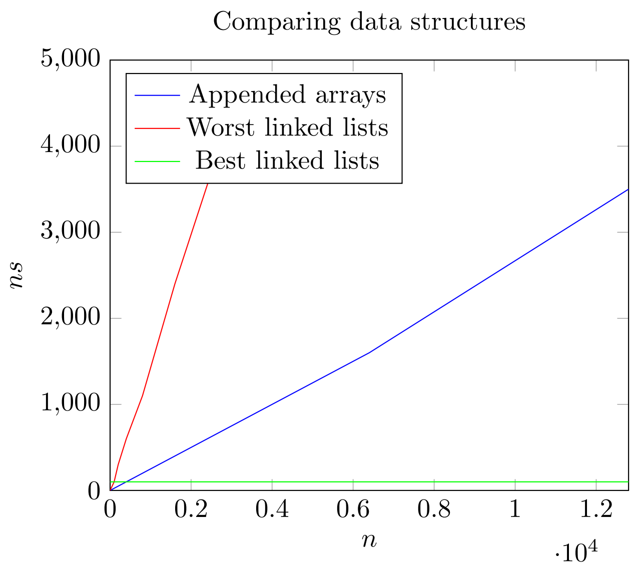
## Linked lists using arrays

Elements in an array are stored at contiguous positions in the memory which is beneficial in many ways. As an example this provides optimal search time of constant time complexity since any element in an array can be accessed by changing the offset of the array index, which is not possible in lists where one have to wander through the entire one-dimensional node tree, from the beginning to the target node, to access a particular element. However arrays are known for being formidable to manipulate and there is no effortless way to work around the slow insertion and deletion times. In the third benchmark an array of varying size (n) and an array of fixed size with 100 elements are filled with "dummy values" and copied into a larger array of size n + 100. The following measurements were obtained:

| **Iterations** | **Array size (n)** | **Static array size** | **Lowest runtime** |
|---|---|---|---|
| 1000000 | 400 | 100 | 100$ns$ |
| 1000000 | 800 | 100 | 200$ns$ |
| 1000000 | 1600 | 100 | 400$ns$ |
| 1000000 | 3200 | 100 | 800$ns$ |
| 1000000 | 6400 | 100 | 1600$ns$ |
| 1000000 | 12800 | 100 | 3500$ns$ |

Table 3: Appended arrays

These measurements confirm that the time complexity of this method is linear (O(n)). Together with the previously collected data, all measurements are graphically plotted to get a more evident picture of their relative efficiency.



This method clearly performs better than the worst case of linked lists, however as long as the linked list method link a larger list onto a smaller one, the linked list method will eventually perform better for an increasing size of the larger list and any of the arrays.

Accessing memory in lists are, as previously mentioned, not as efficient as in accessing memory in arrays. Therefore the creation of a list of size (n) should take longer time than creating an array of size (n). The following measurements were obtained through benchmarking:

| Iterations | Array size (n) | Lowest runtime List | Lowest runtime Array |
|---|---|---|---|
| 1000000 | 1000 | 2.6$\mu$s | 0.1$\mu$s |
| 1000000 | 2000 | 5.2$\mu$s | 0.2$\mu$s |
| 1000000 | 4000 | 10.4$\mu$s | 0.6$\mu$s |
| 1000000 | 8000 | 21.1$\mu$s | 1.2$\mu$s |
| 1000000 | 16000 | 43.9$\mu$s | 2.3$\mu$s |

Table 4: Cost of building a list and an array of size n

These measurements confirm that the cost of creating arrays are considerably less than the cost of creating lists of the same size.

## Stacks

Building a stack using a linked list implementation surely has its benefits compared to an array stack if implemented correctly. By implementing a stack with linked lists where each element is ”pushed” to the first position of the list, one does not have to iterate through the entire node tree to pop the most recently used element.

```
public LinkedListStack push(LinkedListStack stack) {
        LinkedListStack nxt = this; //elementet som matas in (element.push(stack)
        nxt.tail = stack; //ändrar pekaren till att peka på stacken
        return nxt; //uppdaterar stacken
    }
```

This leads to a linked list stack where the push and pop operations are constant just as in the stack implemented using arrays (even though the operations may take longer time than for the array). The main benefit of using the linked list implementation is that we have a dynamic stack in which there is no size dependent cost for increasing and decreasing the size, which is a heavy drawback for the dynamic array implementation. Furthermore the linked list stack will only allocate the memory needed to store the precise amount of elements whereas in the array implementation of the stack one often allocate more memory than needed, especially for larger sized arrays. For very small sized arrays the array implementation might be more efficient, but in general case and for very large sized arrays the linked list implementation will perform better.