# Linked Lists

William Frid

Fall 2023

## Introduction

When writing software there are different ways of storing data the developer can choose between. Two of those ways, that are common, are arrays and linked lists. The array is based on an allocated set of memory with an index system for targeting the correct data for reading and writing. The linked list is based on elements/cells that are linked to each other via reference. These cells can be linked either singly or doubly. That is to say, linked to the next cell in line, or both to the previous and next cell allowing for easier navigation between cells.

One can easily conclude based on the description that these two data structures have different use cases. The goal of this report is to learn how to build a singly linked list from scratch using Java and compare it to Java's built-in array data structure. This will be done using analysis or code and benchmarks to come to a conclusion when which should be used over the other and how well/bad a linked list performs.

## Linked List

### Structure

The first step in building the singly linked list is making it's structure (class). And since this type of object is going to hold cells, it's very convenient to create an inner class for that. See the snippet below for the base structure.

```
class LinkedList {

    Cell first;

    private class Cell {
        int val;
        Cell nxt;

        Cell(int target_val, LinkedList list) { ... }
```

```
    }

    public void add(int val) { ... }
    public int length() { ... }
    public boolean find(int key) { ... }
    public void remove(int key) { ... }
    public void append(LinkedList second_list) { ... }

}
```

The snipped above contains the methods and an inner class required for making a fully usable singly linked list. The add method creates a new cell and inserts it at the start of the list. Length iterates through the list and returns the number of cells while finding returns true or false based on the fact if it finds the key or not. The other two methods, remove and append are more complex and will be spoken of further down.

Some of these methods can be more effective if one were to use doubly linked lists since this structure (singly linked) requires additional references to work properly since there is no way to check the previous value if it's not saved individually. This can be seen in the remove method which is the most complex operation used.

```
public void remove(int key) {
    Cell target = this.first;
    Cell nxt_target = this.first.nxt;

    if (target.val == key) {
        this.first = nxt_target;
        return;
    }

    if (nxt_target == null) {
        return;
    }

    while (nxt_target.nxt != null) {
        if (nxt_target.val == key) {
            target.nxt = nxt_target.nxt;
            return;
        }
        target = target.nxt;
        nxt_target = nxt_target.nxt;
    }
```

```
    if (nxt_target.val == key) {
        target.nxt = null;
    }
}
```

Following is the append operation which is going to benchmarks under the section `Benchmarking`. By analyzing this snippet, we get that the time complexity is $O(n)$ where n is equal to the current list size because of the while loop. This means that the append operation shouldn't take longer as the size of the second linked list increases since it doesn't have to loop that one.

```
public void append(LinkedList second_list) {
    Cell last = this.first;
    while (last.nxt != null) {
        last = last.nxt;
    }
    last.nxt = second_list.first;
    second_list.first = null;
}
```

## Benchmarking

The first benchmark run tested the speed of the append operation using two linked lists of which one had fixed size, and the other varied. This was also done vise versa to give results that would show a clear connection to the calculated big-O complexity.

| List A | List B | Time(µs) | Diff. |
|---|---|---|---|
| 100 | 100 | 0.19 | x |
| 100 | 200 | 0.20 | 1.05 |
| 100 | 400 | 0.20 | 1.00 |
| 100 | 800 | 0.20 | 1.00 |
| 100 | 1600 | 0.23 | 1.16 |
| 100 | 3200 | 0.24 | 1.04 |
| 100 | 6400 | 0.18 | 0.75 |
| 100 | 12800 | 0.18 | 1.00 |
| 100 | 25600 | 0.23 | 1.27 |
| 100 | 51200 | 0.22 | 0.96 |

Table 1: Benchmark comparing append method with varying size of list being appended.

| List A | List B | Time(µs) | Diff. |
|---|---|---|---|
| 100 | 100 | 0.22 | x |
| 200 | 100 | 0.32 | 1.46 |
| 400 | 100 | 0.59 | 1.84 |
| 800 | 100 | 1.10 | 1.86 |
| 1600 | 100 | 2.19 | 1.99 |
| 3200 | 100 | 4.28 | 1.95 |
| 6400 | 100 | 8.78 | 2.05 |
| 12800 | 100 | 17.99 | 2.05 |
| 25600 | 100 | 37.55 | 2.09 |
| 51200 | 100 | 78.98 | 2.10 |

Table 2: Benchmark comparing append method with varying size of list being append to.

Looking at Table 1 and Table 2, we see that the values under column `Time(µs)` change very differently. The time doesn't increase when the size of linked list A increase, but does when linked list B do. This is, as mentioned under section `Structure`, because of the append operation algorithm. It needs to iterate through the whole list being appended to find its last cell before pointing that cell to the first cell in the list being appended.

Finally looking at the `Diff.` column, we get that the difference is close to double for each step the size of list A doubles. Thus proving that the big-O complexity for the operation is $O(n)$ where n is equal to the size of linked list A.

## Linked List Versus Array

When adding more data/cells to a linked list, there's no need to copy all the old data like one has to do when using arrays as those are of a fixed size. This means that the software first has to allocate a new array equal to the combined size, followed by populating that array with all data from the two arrays. But how big is this time penalty? This was checked using a benchmark that appended one array to another.

| Arr. A | Arr. B | List time(µs) | Arr. time(µs) |
|---|---|---|---|
| 100 | 100 | 0.23 | 0.13 |
| 100 | 200 | 0.22 | 0.17 |
| 100 | 400 | 0.23 | 0.24 |
| 100 | 800 | 0.21 | 0.36 |
| 100 | 1600 | 0.24 | 0.62 |
| 100 | 3200 | 0.25 | 1.17 |
| 100 | 6400 | 0.18 | 3.13 |
| 100 | 12800 | 0.18 | 4.41 |
| 100 | 25600 | 0.24 | 9.78 |
| 100 | 51200 | 0.23 | 17.93 |

Table 3: Benchmark comparing appending arrays and appending linked lists.

Table 3 shows a clear comparison of how the linked list append becomes faster as the amount of data grows. The append for the linked list starts slower because of the additional logic behind it, but since its big-O complexity is $O(n)$ where n is equal to the size of the list being appended to, we know that it eventually going to outrun the arrays append as that big-G complexity is $O(n + m)$ where n is the size of the fist array, and m the second array. This is visible when array B is large enough in the table, `Arr. time(µs)` gets close to doubling in size each alongside the size.

Let's say the goal is to only allocate a certain amount of data as fast as possible. Then which one is the fastest? This was tested using another benchmark with modified methods to only allocate, and not add any data to either the array or linked list.

| Size | List time(µs) | Arr. time(µs) |
|---|---|---|
| 100 | 0.29 | 0.08 |
| 200 | 0.52 | 0.09 |
| 400 | 1.00 | 0.10 |
| 800 | 1.87 | 0.13 |
| 1600 | 3.71 | 0.20 |
| 3200 | 8.01 | 0.33 |
| 6400 | 14.57 | 0.58 |
| 12800 | 29.03 | 1.16 |
| 25600 | 63.02 | 2.37 |
| 51200 | 118.00 | 4.18 |
| 102400 | 265.98 | 8.31 |

Table 4: Benchmark comparing time to allocate memory.

According to the benchmark results in Table 4, allocating memory for

an array is faster than for a linked list. The linked list big-O complexity is $O(n)$ and the array has a complexity that seems to be logarithmic. With that said, the time to allocate memory for a linked list will be faster as the size grows.

## Linked List Used As A Stack

Finally, we look at how a linked list performes against an array when used as a stack. The big advantage of linked lists for stacks is the ability to grow and shrink without having to copy any data as is needed when an array gets full. This means that the time it takes to push and pop always remains constant, while when using an array, additional actions might be required to adjust the size of the array containing the stack.

```
public class Stack {
    Element first;
    private class Element {
        int val;
        Element nxt;
        Element(int val, Stack stack) {
            this.val = val;
            this.nxt = stack.first;
        }
    }

    public void push(int val) {
        Element new_el = new Element(val, this);
        this.first = new_el;
    }

    public int pop() throws Exception {
        if (this.first == null) {
            throw new Exception("Tried to pop empty stack");
        }
        int tmp = this.first.val;
        this.first = this.first.nxt;
        return tmp;
    }
}
```

A quick overview of the code tells us that not only does the linked list perform better as a stack, but it also requires less logic to push and pop.

## Discussion & Conclusion

Linked lists have just like all other data structures both advantages and disadvantages. The linked lists prove to be more effective in some cases as they perform faster when deleting and appending than arrays. The arrays however perform faster for reading and writing as they are index-based and also allocate small amounts of memory faster.

Based on this, a developer needs to analyze the data that's going to be stored and calculate how it is going to be used before deciding which data structure to use as the linked list might have great usability but not be as fast as an array.

Additional benchmarks would be useful to compare all methods directly between linked lists and arrays, such as the add and remove operation. Finally, a benchmark for the use of the linked list as a stack would be preferred to visualize this difference that was concluded based on theory.