Simple Linked Lists

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open a window on a different way of storing data, namely through a simple linked list. There are, of course, certain advantages of using this data structure, however, among others, the main focus of this experiment will be the *append* operation and simulating a stack using a linked list.

My approach: Checking the accuracy

Before running any benchmarks, my priority was to first check the accuracy of the new methods I created. To illustarte this, in order to see that the elements in the list are connected correctly or that the right element is removed when the specific method is called, I decided to create a new method in order to display the linked list under the form of an array, as follows:

```
int[] asArray() {

    //call method 'length' which returns the list's size
    int [] array = new int [length()];
    Node index = first; //copy the reference of the first node

    //navigate through the list and copy the stored values into the array
    for (int i = 0; i < length(); i++) {
        array[i] = index.number;
        index = index.next;
    }

    return array;
}
```

Appending two lists

In order to append two lists, the goal was to navigate through the first one until **null** was reached. In other words, the last node was identified whose reference we then have to change so that it points to the first node of the second list. It should be taken into consideration that the analysis of the benchmark consists of two parts.

First and foremost, the time complexity of appending a list of fixed size to a list whose size is varying will be **O(n)** since we always have to navigate though the data structure of length 'n' to get to the end point.

On the other hand, when I started the experiment, at first I was expecting a roughly constant execution time for the second case when the linked list with a varying size is appended to one of fixed length. However, initially the terminal was indicating other results. Litle did I know that my fixed linked list was not as 'fixed' as it should have been... I was always appending to it new lists, without ever reseting it as below:

```
long t0 = System.nanoTime();
linked.append(linked_2);
long t1 = System.nanoTime();
double t_append_varyA = (t1 - t0);

//reset fixed linked list
linked_3.add(numbers_2[0]);
linked_3.createlist(numbers_2);
```

After having done so, the execution time was approximately *constant*.

Appending two arrays

Appending two arrays, will of course, take up more resources since apart from creating a new array the size of the two previous arrays summed up, there is a lot of copying involved as well. So, this time the time complexity will no longer depend on the size of *one array only*, but on both lengths. Suppose the first array has length 'a' and that the second has length 'b'. Then, the time complexity will be **O(a+b)**. All in all, according to my benchmark, this operation is on average **3 times slower**.

A stack under the form of a linked list

Implementing a stack using a linked list was undoubtedly much easier to implement since the *pop* and *push* operations behave as if we removed and

returned the first element of the list, then respectively added a new element at its top. As far as the time complexity is concerned, both operations are *O(1)* since we don't have to go through the whole array in order to execute them. We simply have to create a new node and change the references.

In addition, there are several advantages of using a stack under the form of a linked list. For instance, when we *push* an element, apart from not worrying anymore about reaching the maximum allocated space, we no longer create a new array and we also skip all the copying phase! Moreover, the questions related to how much we should increase the new stack or when the perefect moment to remove the empty spaces is, are no longer an issue! By and large, it is more efficient!

Conclusion

All in all, a linked list is an important data structure which presents benefits given several operations or when implemented under the form of a stack. However, it is highly important that one doesn't lose the reference to the first node of the list. If otherwise, the whole list is lost, which is something to be avoided.