Lab 4 How To Sort

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in "Linked lists" for ID1021. In this assignment, the goal is to analyse and understand usage of linked list in different situations, and how efficient they are through bench marking using the in built java clock, which measures in nanoseconds.

Background

This task enhances our understanding in linked lists, which are structures that are linked to each other using pointers. This pointer points to another object, and only works in one way, so the object being pointed to is unaware of where it is being referenced from. This assignment focuses on simple liked structures and analysing their properties, and discusses the difference between arrays, which we have worked with before compared to linked lists. We also discuss the instances where linked lists and arrays are used for stacks and compare their advantages and disadvantages.

Tasks

a linked list

In a simple linked list, each element will contain a reference which refers to the next element in the list. At the end of the list, the reference is a null-pointer. This tasks asks us to access the run time of the append operation in the linked list. The benchmark runs for 1000 iterations, and averages the total in order to find the most accurate data.

```
public void append(LinkedList b) {
    LinkedList nxt = this;
    // Traverse till last element
    while (nxt.tail != null) {
        nxt = nxt.tail;
    }
    //Change the tail of last element
    nxt.tail = b;
}
```

The append operation above adds a new element at the end. Our assignment is to append a linked list of varying sizes (a) to a linked list (b) with fixed sizes, and vice versa. The data received from this operation is.

Elements in array	**Microseconds (Fixed B)**	**Microseconds (Fixed A)**
200	0.12	1.25
400	0.34	1.33
600	0.46	1.27
800	0.58	1.28
1000	0.71	1.32
2000	1.45	1.29
4000	2.67	1.23

As shown above, the table indicates that when a dynamic array is being appended to a fixed array, as the array size increases, it takes more time to find the null pointer of the list (a). The time complexity of this appending method is O(n) which should display a linear trend, visible on the table above.

When list (a) is a fixed list which appended to the list (b), the time complexity is O(1), since (a) is a fixed list, and therefore the reference, or pointer will take the same amount of time traveling to the end of the list each time so that the head of (a) can be found.

Array linked list

The second task requires for us to find the method to append two arrays. In this case, the goal is to allocate a new array that is able to copy all the values from both array itself. In order to complete this task, the append operation must be changed a bit to fit the situation of the double array.

```
public int[] append(int[] arrayA, int[] arrayB) {
     int newArraySize = arrayA.length + arrayB.length;
     int[] newArray = new int[newArraySize];
     for (int i = 0; i < newArray.length; i++ ) {
         if(i < arrayA.length){
             newArray[i] = arrayA[i];
         }
         else{
```

```
                    newArray[i] = arrayB[i - arrayA.length];

                }
            }
            return newArray;
        }
.
.
.
```

After running the piece of code in a benchmark, we receive the following results.

Elements in array	Microseconds (A to B)	Microseconds (B to A)
200	0.32	0.35
400	0.52	0.53
600	0.73	0.77
800	0.98	0.95
1000	1.22	1.32
2000	2.42	2.49
4000	4.52	4.63

The two arrays for appending a to b and b to a does not serve much of a difference, since this method of appending ends up with the two complexity adding up, we end up with O(n).

In order to analyse the allocation time of linked list and array, both will have the time complexity of O(n), as we have analysed before. The arrays have an advantage in efficient search and access time, where each element can be accessed easily, linked list on the other hand requires to traverse through the entire linked list to find an element. However, insertion and deletion times in linked list is faster than that of the array. When the time used for allocation becomes more than the time saved in execution, the linked list will be slower than arrays. My hypothesis is that as arrays get larger, the time take for linked list will be longer than the arrays.

Stack

Similar to previous tasks, we are asked to implement a stack in this case, using linked list. The operation of last in first out structure, working with push and pop. The initial stack contains a pointer pointing at the top of the stack, which pushes and pops the element that is at the top. Dynamic linked list is able to shrink and grow, which discards the restriction that arrays have in stack operations, so the element will not get stack overflow as elements are dynamically allocated. But the allocation time for adding elements will be high in a dynamic situation. An array on the other hand has O(1) at the best, and O(n) at worst, they might be slower than linked lists in the worst case scenario, but with reference the previous section, we can conclude that linked lists usually take more time to allocate, so overall, linked lists will be slower.