Linked lists

Mahdi Nazari

September 2022

Linked list

A linked list is a type of linked data structure which contains nodes with some properties like a value and a reference to the next node in the list. If the reference in a node is null then the node is the last node in the list. Some operations which can be performed with a linked list is to add a node to a linked list or remove a node from the list.

Appending or Linking Two Linked Lists

In this section two linked lists will be appended to each others in order to build a new larger linked list. In a benchmark the time complexity of the execution time of the append operation regard to the number of the nodes in the list will be presented. At the end it will will be discussed why the result is as it is. In the first task a varied size list will be appended to a fixed size list. To do this task a new function which would generate a list will be necessary. This list generator is constructed as follows:

```
public static linkedList listGenerator(int size){
       Random rnd = new Random();
       linkedList listB = new linkedList(rnd.nextInt(100), null);
       for(int i=0; i<size-1; i++){
           linkedList list2 = new linkedList(rnd.nextInt(100), null);
           listB.append(list2);
       }
       return listB;
   }
```

Now a varied size list which will start from 100 nodes and will continue up to 600 nodes will be appended to a fixed size list which contains 100 nodes. The result from benchmarking is presented in the first column in the table below.

number	Appending a(v) to b(f)	Appending a(f) to b(v)	ratio
100	170	170	1
200	170	380	2.2
300	130	580	3.4
400	130	790	4.6
500	130	1000	5.8
600	130	1200	7

Table 1: The runtime for benchmarking of appending a varied size list (v) to a fixed size list (f) and vice versa. The runtime is in ns

The result from benchmarking shows that the execution time for appending a varied size list to a fixed size list will be almost constant and the time complexity for this operation will be an O(1) type.
In the second task a fixed size list will be appended to a varied size list and the runtime from benchmarking is as the second column in the Table 1 above.
As the result shows the execution time increase as the number of node in the varied size list increase. To understand how the runtime increase the ratio of the runtime is presented in the Table 1. It shows as the number of node increases twice the runtime is also increasing almost twice and this trend seems to continue. In other words the runtime is linear to the number of nodes in the varied size list. Then the time complexity will be an O(n) type.

Disscussion

In the first task the runtime is constant regardless to the number of the nodes in the varied size list because the Append-function will go through the whole list in the fixed size list and will append the varied size list by making the last node in the fixed size list refer to the head of the varied size list. The runtime for this operation will be constant as long as the first list which Append-function goes through is constant. In second task the Append-function was appending a fixed size list to a varied size list and therefore the execution time was increasing as the size of the varied size list was increasing.

Linked List VS Array

In this part a corresponding operation as Append was constructed for arrays. To see the differences in the execution time the same scenario as the last part was constructed but this time for arrays. In other words, first a varied size array was appended to a fixed size array and then a fixed size

array was appending to varied size arrays. The result from benchmarking is as in the table below:

number	**Appending a(v) to b(f)**	**Appending a(f) to b(v)**
100	125	125
200	125	166
300	208	208
400	250	208
500	292	250
600	334	291

Table 2: The runtime for benchmarking of appending a varied size array (v) to a fixed size array (f) and vice versa. The runtime is in ns

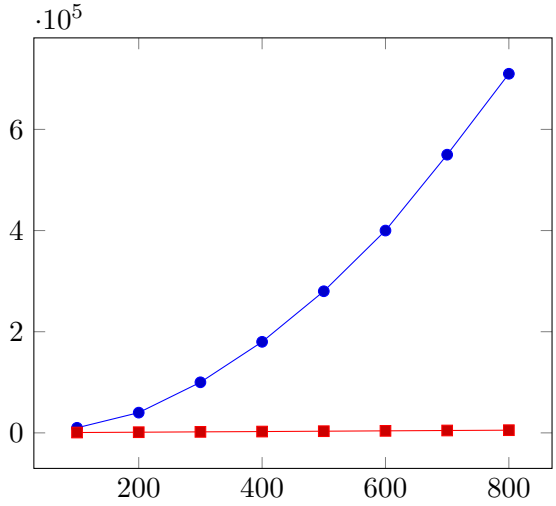
As the result shows unlike the linked lists the execution time is almost the same regardless to if a varied size array was appended to a fixed size array or vice versa. This is because in the both scenarios Append-function will allocate a bigger size array which can contain both fixed size array and varied size array and then it will copy the elements from these arrays to the new bigger array. This process will take same time if the Append-function copy first fixed size array and then varied size array or vice versa. Another point from benchmarking is that the execution time slightly increases as the number of elements in the varied array increases because Append-function has to copy more elements.

Allocating array vs linked list

In this part the execution time to create an array of a certain length with a linked list of the same size will be compared. The result is as the table and graph below.

Size	**Linked List**	**Array**
100	10000	670
200	40000	1300
300	100000	2000
400	180000	2600
500	280000	3300
600	400000	4000

Table 3: The runtime for benchmarking of creating a linked list of a certain size compared to creating an array of the same size. The runtime is in ns



Graph 1: Results from benchmarking of creating a linked list(blue line) and creating an array(red line). x-axis, size of arrays, y-axis execution time in ns.

As the Table 2 and Graph 1 shows the time complexity of creating a linked list is much longer than creating an array. Graph 1 shows that time complexity of creating a linked list with n number of nodes is an $O(n^2)$ type but time complexity of creating an array is an O(n) type.

Dynamic Stack with a Linked List

Dynamic stack with a linked list and dynamic stack with an array have differences in access, search and push and pop operations. In a dynamic stack with a linked list to access an element(node) in the stack the program should go through the all elements(nodes) before the targeted element(node) and this will cause a longer execution time which would depends on the position of the element(node) in the list. This means that the time complexity for the access and search function in a stack with a linked list will be O(n) type. But in a stack which is constructed with an array the time complexity for access and search operations will be constant and O(1) type.
Regarding to the push and pop operations the dynamic stack with a linked list has no limits (disregard to the systems hardware like stack and heap). This means that increasing and decreasing the stack will not cost a longer execution time. But in the dynamic stack constructed with an array the size of the stack will be decided in the beginning of the program and when the all plats in the stack is occupied then the stack should expand the size of array and copy the all elements from the old array to the new larger array and that will cost longer execution time for push operation. In the same way when the stack wants to decrease the size then it should allocate a smaller array and copy all elements to the new smaller array and this will also cause

a longer execution time.