

BBM-201 Data Structures Exercise: Lists (2)

Instructor: Prof. Dr. Emirhan Yalçın

Topics: Linked List, Array List

November 1, 2023

Name:				Id:					
Surname:				Section:					
Total 100 Points									
Question	1 (10pt)	2 (18pt)	3 (12pt)	4 (12pt)	5 (12pt)	6 (18pt)	7 (8pt)	8 (10pt)	Total: (100pt)
Deserved:									

- This examination consist of 8 questions about list data structure.
- Like the first exercise, this questions also mostly focus on basic operations of the list. But those basic implementations are important learn for solving and implementing more complicated algorithms in future.
- You are expected to use C++ language to implement solutions. If you want, you can solve them using Java first and convert to C++ later.
- You are adviced to solve this questions with a time limit and do your best with detailed explanations in order to provide active learning.
- If you want you can print this papers in order to feel more like you are in a real examination.
- Even if you can't solve some questions first or feels hard to build solutions, you are adviced to repeatedly solve questions and review them until you fully understand the solutions and feels easy to implement. You should be able to solve this question almost automatically in order to master list data structure and solving more complext problems in future.
- Unless stated otherwise, assume the implementations on the next page.
- Good luck.

```

1  struct Node {
2      int value;
3      Node* next;
4      Node(int value) :
5          value(value), next(nullptr) {}
6  };
7  struct DoublyNode {
8      int value;
9      DoublyNode* before;
10     DoublyNode* next;
11     DoublyNode(int value) :
12         value(value), before(nullptr), next(nullptr) {}
13 };
14
15 class LinkedList {
16     int size;
17     Node* head;
18     LinkedList() :
19         size(0), head(nullptr) {}
20 };
21 class ArrayList {
22     int size;
23     int* main_array;
24     ArrayList(int maxSize){
25         this->main_array = new int[maxSize];
26         size = 0;
27     }
28 };
29 class DoublyLinkedList {
30     int size;
31     DoublyNode* head;
32     DoublyNode* tail;
33     DoublyLinkedList() :
34         size(0), head(nullptr), tail(nullptr) {}
35 };
36

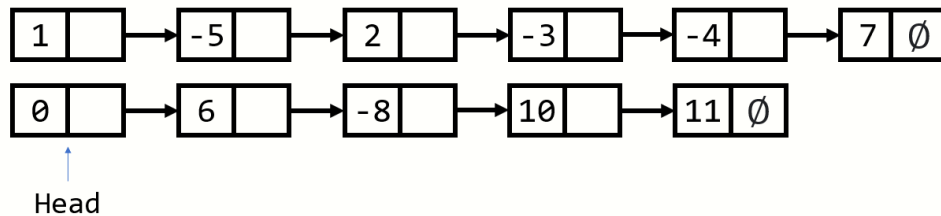
```

1. (10 points) **Linked List Returning Function**

(a) What does the following function do?

```
1 Node* someFunction(Node* head1, Node* head2) {
2     Node* mergedHead = nullptr;
3     while (head1 != nullptr || head2 != nullptr) {
4         if (head1 != nullptr && head1->value > 0) {
5             Node* newNode = new Node(head1->value);
6             newNode->next = mergedHead;
7             mergedHead = newNode;
8             head1 = head1->next;
9         }
10        if (head2 != nullptr && head2->value > 0) {
11            Node* newNode = new Node(head2->value);
12            newNode->next = mergedHead;
13            mergedHead = newNode;
14            head2 = head2->next;
15        }
16    }
17    return mergedHead;
18 }
```

(b) Draw the output linked list of the function when head of two linked list below given as parameter.



2. (18 points) **Delete At Index:** Implement `pop(int index) \Rightarrow int` method which removes item and index and returns the value. You should consider the cases where list may be empty, or has only one or two elements. Also if given index is invalid, don't delete anything and return -1. ($0 \leq \text{index} < \text{size}$) Can you tell the number of operations in your implementations using tilda notation?

(a) Implement delete method for Singly Linked List.

(b) Implement delete method for Doubly Linked List.

(c) Implement delete method for Array List.

3. (12 points) **List Insertion at Index:** Implement `insert(int index, int item)` method that inserts the new element to the specified position in the list. After insertion, you should locate the item at specified index.

- (a) Implement the method for Singly Linked List. But assume that your Singly Linked List implementation keeps both head and tail pointers as attribute.

- (b) Implement the same for Doubly Linked List.

4. (12 points) **Last nth Node of a Linked List:**

- (a) Given the head of the Singly Linked List, write a method that returns n^{th} last element of the list. Note that you are allowed to do only one iteration. i.e. you cannot use more than one for or while loop. Assume that n will have always a valid value, in range of list size.

```
int getNthFromEnd(ListNode* head, int n) {
```

- (b) Given the head of the Singly Linked List, write a method that removes n^{th} last element of the list, and return new head of the list. Again you must do this in one iteration.

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
```

5. (12 points) **Insert - Delete After:** Implement Singly Linked List methods that inserts a new element or removes one after the first occurrence of a specific element in the list.

- (a) Implement **insertAfter**(int **itemValue**, int **newItem**) method. Note the invalid case, which the item does not exist in the list. (Optional Solution Advice: You can define a **find**(ListNode* head, int itemToFind) method that returns index of the item, if not found returns -1.)

- (b) Implement **deleteAfter**(int **itemValue**) method. Note the invalid cases such that the given parameter is the last element of the list.

6. (18 points) **Doubly Linked List vs Singly Linked List:**

- (a) Implement the following methods which inserts and delete items at front and tail. What are the time complexities of your methods?

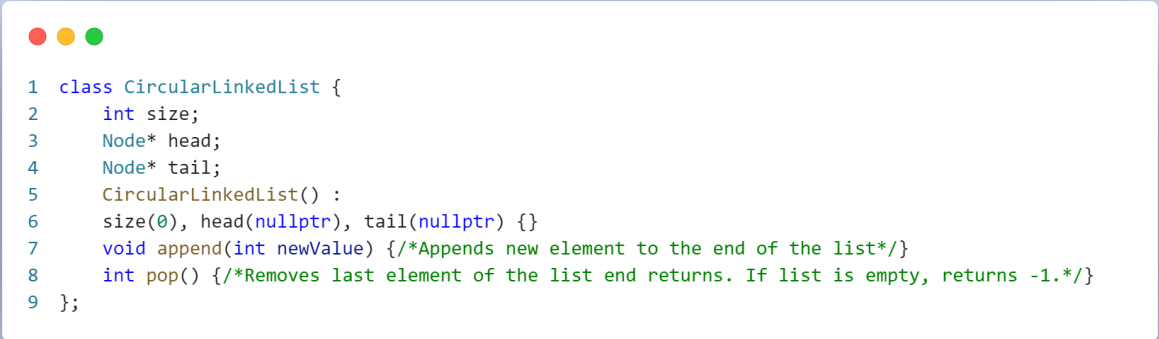
addFront()	\Rightarrow	$O(\quad)$
addBack()	\Rightarrow	$O(\quad)$
deleteFront()	\Rightarrow	$O(\quad)$
deleteBack()	\Rightarrow	$O(\quad)$

- (b) Compare Singly and Double Linked List data structures considering their memory usages and time complexities. What advantages and disadvantages has the Doubly Linked List over Singly Linked List?

7. (8 points) **Middle Element of a Linked List** Given a head of a singly linked list, return the middle element of the list. If number of elements in the list is an even number, return the first-middle one.

```
int getMiddleElement(ListNode* head) {
```

8. (10 points) **Circular Linked List** Circular Linked list a speacial version of lists, in which tail node shows head node as next. Below you can see basic structure of a Circular Linked List. implement the append and pop methods for the circular linked list. Note that tail node must always show the head after any operation.



```
1 class CircularLinkedList {
2     int size;
3     Node* head;
4     Node* tail;
5     CircularLinkedList() :
6         size(0), head(nullptr), tail(nullptr) {}
7     void append(int newValue) { /*Appends new element to the end of the list*/}
8     int pop() { /*Removes last element of the list end returns. If list is empty, returns -1.*/}
9 };
```