Exercise Session 06

Exercise 1.
In class we have seen that using arrays for implementing stack may lead to underflow and overflow problems. Propose an alternative implementation of the stack operations STACK-EMPTY, PUSH and, POP that make use of doubly linked lists to implement a stack.
What is the worst-case running time for the POP and PUSH operations? May we still incur in situations where the stack underflows or overflows?

STACK-EMPTY:
Check if the head of the list has a next pointer that is null. This means that the head is the only element in the stack/Linked list. $\Theta(1)$.

PUSH:
In terms of implementing a stack like linked list we would only be able to add elements from the tail. Therefor we would need to traverse the whole linked list from the head to the tail and add that node from the list and updating the previous nodes next pointer to null. $\Theta(n)$

POP
In terms of implementing a stack like linked list we would only be able to remove elements from the tail. Therefor we would need to traverse the whole linked list from the head to the tail and remove that node from the list and updating the previous nodes next pointer to null $\Theta(n)$

Underflow/overflow:
The linked list is dependent on the amount of elements from head to tail, hence, there will not be any underflow/overflow in this case. As it decreases/increases the length of the linked list dynamically dependent on the amount of nodes when a new tail is added/removed.

Exercise 2.

(a) [CLRS-3 10.4-2] Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

```
PRINTTREE(x)
1  if (x ≠ null)
2  print(x.key)
3  PrintTree(x.l)
4  PrintTree(x.r)
```

(b) [CLRS-3 10.4-3] Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

```
ITER-PRINT(T)
1  S be an empty array
2  Push(S, T.root)
3  while ¬StackEmpty(S)
4        x = Pop(S)
5        if x ≠ null
6              print(x.key)
7              Push(S, x.right)
8              Push(Sm x.left)
```

The iterative approach uses a stack in order to print all the elements in T. *Iter-Print* does this by iteratively pushing the current parents/root into a stack and printing its key. Hereafter we update the

stack with the current parent/roots left and right children, such that they now become the parents. This approach continues until a node has no children. The while loop terminates when the stack is completely empty, thus implying that we have traversed the whole rooted tree.

Exercise 3.
Singly linked lists are a variant of linked lists where each element x has two attributes, $x.key$ that stores the key, and $x.next$ that points to the next element in the list. Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

Insert: Insert can be done in $\Theta(1)$ by updating the head to be the new node and setting its next pointer to the current head. Thus we insert elements at the start of the list and therefor doesn't need to traverse the list at all.

Delete: Delete would still require the list to be traversed an can therefor not be done in $\Theta(1)$, but rather $O(n)$

Exercise 4.
Give a $\Theta(n)$-time nonrecursive procedure that reverses a *singly* linked list of n elements by updating its pointers. The procedure should use no more than constant storage beyond that needed for the list itself.

```
LISTREVERSE(L)
 1  head = L.head.next
 2  OGhead = L.head
 3  OGhead.next = Null
 4  prev = OGhead
 5  while (head ≠ Null)
 6        next = head.next
 7        head.next = prev
 8        prev = head
 9        head = next
10  head.next = prev
```

Exercise 5.
Implement a queue by a singly linked list Q. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.
Hint: You can assume that in addition to the usual attribute $Q.head$, pointing to the head of the list, the list Q has also the attribute $Q.tail$ which points to the last element of the linked list.
Enqueue:

```
ENQUEUE(Q, newNode)
1  Q.tail.next = newNode
2  Q.tail = newNode
```

Dequeue:

```
DEQUEUE(Q)
1  Temp = Q.head
2  Remove(Q.head)
3  Q.head = Temp
```