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 that use doubly linked lists. 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?

Solution 1.
We assume S to be a doubly linked list and we use the attribute *head* to point to the top of the stack. Then, the stack is empty if the list is. Pushing an element on top of the stack corresponds to list insertion as in CLRS. Finally, the pop procedure corresponds to the deletion of the head of the list.

```
STACK-EMPTY($S$)
1  return $S.head$ == NIL
```

```
PUSH($S, x$)
1  LIST-INSERT($S, x$)
```

```
POP($S$)
1  if STACK-EMPTY($S$)
2      error "underflow"
3  else
4      $x = S.head$
5      LIST-DELETE($S, x$)
6      return $x$
```

The worst-case running time is $\Theta(1)$ for each operation as the corresponding operations on list used therein have constant worst-case running time too. This new list-based implementation can still produce underflow when trying to pop from an empty stack. However, it does not suffer any longer from overflow problems because the list can be arbitrarily long.

Exercise 2.
Exercises 10.4-2 and 10.4-3 CLRS pp. 248.

Solution 2.

(a) We have to write a $O(n)$ time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. The following procedure prints the tree rooted at node x.

```
REC-PRINT($x$)
1  if $x \neq$ NIL
2      print $x.key$
3      REC-PRINT($x.left$)
4      REC-PRINT($x.right$)
```

Given a rooted binary tree T we can print it by calling REC-PRINT($T.root$). The worst-case running time of the above algorithm follows the following recurrence

$$T(n) = \max_{0 \leq q \leq n-1} T(q) + T(n-q-1) + \Theta(1)$$

where the parameter q ranges from 0 to $n-1$ because the two subproblems have total size $n-1$. We guess that $T(n) \leq cn$ for some constant $c > 0$. Substituting this guess into the recurrence we obtain

$$
\begin{aligned}
T(n) &\le \max_{0\le q\le n-1} cq + c(n-q-1) + \Theta(1) \\
&= cn - c + \Theta(1) \\
&\le cn
\end{aligned}
$$

The last inequality holds by choosing a constant c large enough so that it dominates the $\Theta(1)$ term. Thus $T(n) = O(n)$.

(b) We have to write a $O(n)$ time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree using a stack as auxiliary data structure. The following procedure prints the tree T without using recursion.

```
ITER-PRINT($T$)
1  Initialise an empty stack $S$
2  PUSH($S, T.root$)
3  while ¬STACK-EMPTY($S$)
4      $x$ = POP($S$)
5      if $x \ne$ NIL
6          print $x.key$
7          PUSH($S, x.right$)
8          PUSH($S, x.left$)
```

Note that the above algorithm runs in linear time, because the body of the while loop takes constant time and the number of iterations is linearly bounded by the number of nodes in T since each node is inserted exactly once in the stack S and at each iteration one element is deleted from S. To be precise the number of insertions in S is $O(n)$ since also the NIL pointers are inserted.

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?

Solution 3.
The insert operation in singly linked lists is very similar to that for doubly linked lists.

```
LIST-INSERT($L, x$)
1  $x.next = L.head$
2  $L.head = x$
```

Clearly, the worst-case running time of LIST-INSERT(L, x) is $\Theta(1)$.
As for the procedure LIST-DELETE(L, x), our goal is to skip the element x in L by making its predecessor point to the element that follows x. However, differently from the case for doubly linked lists, we don't have the predecessor of x ready in our hands: we have to retrieve it by performing a linear search in L starting form its head.

```
LIST-DELETE($L, x$)
 1  $p$ = NIL
 2  $c = L.head$
 3  while $c \ne x$
 4      $p = c$
 5      $c = c.next$
 6  // Now $c = x$ and $p$ is its predecessor
 7  if $p \ne$ NIL
 8      $p.next = x.next$
 9  else // $x$ was the head of $L$
10      $L.head = x.next$
```

In the above pseudocode we assume that x is an element of L, hence we will eventually encounter it while scanning the list. Once we find it we update the pointers so to skip x in L. The worst-case running time is $\Theta(n)$ where n is the length of L, because if x is the last element in the list the procedure scans all the n elements in L.

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

Solution 4.
The following pseudocode used tree pointers p, c, and n to store respectively the previous, current, and next element in the original list. Initially, p and n are set to nil NIL and c points to the head of the list L. While c does not point to the end of the list, we do the following: store the new next element in n, update the attribute *next* of the current element to point to p. Finally, we move one step forward by updating the values of p and c.

```
REVERSE($L$)
1  $p = $ NIL
2  $c = L.head$
3  $n = $ NIL
4  while $c \neq$ NIL
5      $n = c.next$
6      $c.next = p$
7      $p = c$
8      $c = n$
```

The above procedure scans the list once, and each iteration takes constant time. Therefore the running time of REVERSE takes $\Theta(n)$. Furthermore, we use only three additional variables beyond the space needed for the list L.

Exercise 5.
Implement a queue by a singly linked list Q. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

Solution 5.
We assume that the singly linked list Q has two attributes: $Q.head$ which is the usual head of the linked list and $Q.tail$ which points to the last element of the linked list (as usual, $Q.tail = $ NIL when the list is empty). The operations ENQUEUE and DEQUEUE are implemented as follows.

```
ENQUEUE($Q, x$)
1  if $Q.tail \neq$ NIL
2      $Q.tail.next = x$
3  $Q.tail = x$
```

```
DEQUEUE($Q$)
1  if $Q.head \neq$ NIL
2      $x = Q.head$
3      $Q.head = x.next$
4      return $x$
5  else
6      error "underflow"
```

Clearly, both operations take $O(1)$ time in the worst case.