Sample Solution for Assignment 1

10.1-2
Implementing 2 stacks in one array A[1…n] could be achieved by representing one stack at the beginning of the array (A[1]) and the beginning of the second array at the end of the array (A[n]). The tops of the stacks would be approaching each other as elements are added to either stack. Overflow will occur when the tops of the stacks hit each other.

10.1-3
//the following pseudo-codes use square brackets to denote the array index and parentheses to denote function call.
ENQUEUE-HEAD(Q,x)
   IF head = 0
      IF tail = length(Q)
         RETURN -1  //overflow
      ELSE
         head = length(Q)
   ELSE
      head = head – 1
   Q[head] = x
   size = size + 1

DEQUEUE-HEAD(Q)
   IF size = 0
      return -1  //underflow
   x ← Q[head]
   IF head = length(Q)
      Head ← 0
   ELSE
      head ← head + 1
   size = size - 1
   RETURN x

ENQUEUE-TAIL(Q,x)
   IF tail = length(Q)
      IF head = 0
         return -1  //overflow
      ELSE
         tail = 0
   ELSE
      tail = tail + 1
   Q[tail] = x
   size = size + 1

DEQUEUE-TAIL(Q)
IF size = 0
    RETURN -1 //underflow
x ← Q[tail]
IF tail = 0
    tail ← length(Q)
ELSE
    tail ← tail - 1
size = size - 1
RETURN x

10.1-6
Implement a queue using 2 stacks:
Use stack A for ENQUEUE operations, and stack B for DEQUEUE operations. Simulate
ENQUEUE by pushing new element onto stack A. Simulate DEQUEUE by popping top
element from stack B, but if stack B is empty when a DEQUEUE is requested, first
empty stack A into stack B by popping elements one at a time from stack A and pushing
them onto stack B. (Note that copying the stack reverses its order, so the oldest element is
then on top and can be removed with DEQUEUE.)

ENQUEUE takes time O(1). In the average case, DEQUEUE also takes time O(1), but
we could get unlucky and have to transfer n elements from one stack to another, so it is
worst case O(n). (However, we can amortize the cost of the transfer over all of the
ENQUEUE and DEQUEUE operations. It is clear that each element must be popped
from A and pushed onto B exactly once, so an amortized analysis gives an average worst
case time of O(1) for DEQUEUE.)

10.2-1
Both can be implemented in O(1) time on a singly linked list, assuming the position for
insertion and deletion are provided as parameters to the operations (See the solution for
10.2-5).

10.2-2
PUSH(L, x) //x is of type “node” and has been initialized; runs in O(1) time
    Next[x] ← head[L]
    head[L] ← x

POP(L) // runs in O(1) time
    IF head[L] = nil
        RETURN -1 //underflow
    x ← head[L]
    head[L] ← next[x]
RETURN x

10.2-3
ENQUEUE(L, x) //enqueue to tail; runs in O(1) time
    next[tail[L]] ← x
    tail[L] ← x
DEQUEUE(L) //dequeue from head; runs in $O(1)$ time
    IF head[L] = nil
        RETURN -1 //underflow
    x ← head[L]
    head[L] ← next[x]
    RETURN x

10.2-4
Store the value you are searching for as the sentinel and put it to the end of the list. So you can eliminate the bound check in each iteration and do it only once after the element is found. It’s either the one already in the list or the sentinel we have inserted.

10.2-5
INSERT(p, x)  //insert x after p; runs in $O(1)$ time
    next[x] ← next[p]
    next[p] ← x

DELETE(p)  //delete the node after p; runs in $O(1)$ time
    obsolete ← next[p]
    next[p] ← next[next[p]]
    destroy[obsolete]

10.2-6
UNION takes the pointer from the tail element of the first set and points it to the head of the other set linking the two sets into one set. This can be done in $O(1)$ time.

10.2-7
REVERSE-LIST(L)
    first ← head[L]
    temp1 ← nil
    temp2 ← nil
    WHILE first != nil
        head[L] ← first
        temp1 ← next[first]
        next[first] ← temp2
        temp2 ← first;
        first ← temp1;

10.2-8
The following equation followed from logical axiom is essential to the implementation of an xor-list

$$p \oplus p = 0.$$  
Suppose that x is directly before y in the list $[y = \text{next}[x]]$ and the addresses of both x and y are known. We can then evaluate next[y] by $\text{np}[y] \oplus x$, since from the definition of
np[y] this is equal to previous[y] ? next[y] ? x, which in turn is equal to next[y] since previous[y] = x so that previous[y] ? x = 0.

So in order to start the traversal, we must know the address of two nodes in direct succession of each other. Starting in the head, which still has the previous and next pointers, we can traverse the entire list forwards by using h and next[h] in place of x and y in the calculation above. Starting with h and previous[h] instead traverses the list backwards. We know that the end of the list is reached when we get back to the head.

The following algorithm shows in detail how to traverse the list forwards to look for a node with key k.

```
SEARCH(h, k)
    IF next[h] != h
        last ← h
        curr ← next[h]
    WHILE curr != h
        IF key[curr] ← k
            RETURN curr
        temp ← curr
        curr ← last ? np[curr]
        last ← temp
    RETURN nil
```