Tutorial (week 4)

All the answers can be given with pseudocode or with Python. Remember that most of the time, many solutions are possible.

1. Write a function `sorted_insert(h,t,e)` which given the head sentinel `h` and the tail sentinel `t` of an already sorted list (in increasing order) and a single node `e`, inserts the node into the correct sorted position in the list.

   Solution:
   Simply start from the header sentinel `h` of the list and keep link hopping forward from it until you reach an element in the list bigger than `e`, or if you hit the end of the list. Then, insert `e` at this position.

```
def sorted_insert(h,t,e):
    '''
    INPUT: a head node h and tail node t of a doubly linked list
           and an element node e
    OUTPUT: none
    '''

    # keep link hopping forward from the head until you reach an element
    # bigger than e's element. Then insert e at this position
    node_temp = h

    while True:
        node_temp = node_temp.next

        # if we reach the end of the list, we break the while loop
        if node_temp == t:
            break

        # if we found where to place e, we break the while loop
        if e.element <= node_temp.element:
            break

    # insert e at this position
    e.prev = node_temp.prev
    e.next = node_temp
    (e.prev).next = e
    node_temp.prev = e
```

2. Write a function `insertion_sorted(h,t)` which given the head sentinel `h` and the tail sentinel `t` of a list, will rearrange the nodes so that they are sorted in increasing order. You can use the function `sorted_insert(h,t,e)` from previous exercise.

   Solution:
   You can simply start from an empty list and then populate it by ranging from the input list and inserting these elements using the `sorted_insert(h,t,e)` function.

```
def insertion_sorted(h,t):
```

```
    '''
    INPUT: a head node h  and tail node t of a doubly linked list
           and an element node e
    OUTPUT: none
    '''

    # create a new empty list
    new_header = Node()
    new_tail = Node()
    new_header.next  = new_tail
    new_tail.prev  = new_header

    # take all nodes from the original list and insert them in the new sorted list
    # using the previous functionsorted_insert
    node_temp = h
    while node_temp.next != t:
        node_temp = node_temp.next
        new_node = Node()
        new_node.element = node_temp.element
        sorted_insert(new_header,new_tail,new_node)

    # once over, update the head and tail
    h.next = new_header.next
    t.prev = new_tail.prev
    (h.next).prev = h
    (t.prev).next = t
```

3. Describe, using Python or pseudocode, an implementation of the function `insertBefore(p, e)`, for a linked list, assuming the list is implemented using a doubly linked list.

Solution:

```
def insertBefore(p,e):
    '''
    INPUT: a position p and an element e
    OUTPUT: the newly create node v, inserted before p in the doubly linked list
    '''
    v = Node()    # create the node
    v.element = e
    v.prev = p.prev
    v.next = p
    (p.prev).next = v
    p.prev = v
    return v
```

4. A double-ended queue, or *deque*, is a list that allows for insertions and removals at either its head or its tail. Describe a way to implement a deque using a doubly linked list, so that every operation runs in $O(1)$ time.

Solution:
With a doubly linked list, the methods `first()`, `last()`, `insertAfter(p, e)`, `insertBefore(p, e)` and `remove(p)` are all $O(1)$. Thus, in order to perform the insertion of an element `e` at the head of the dequeue, you can use `insertBefore(first(), e)` which is $O(1)$. In order to perform the insertion of an element `e` at the tail of the dequeue, you can use `insertAfter(last(), e)` which is $O(1)$. Finally, removing an element at the head or at the tail of the dequeue can be done with `remove(first())` and `remove(last())`, which are also $O(1)$.

5. Describe, in Python or pseudocode, a link-hopping method `middle_node(h,t)` for finding the middle node of a doubly linked list with header and trailer sentinels (`h` and `t`), and an odd number of real nodes between them (note that this method must only use link hopping, it cannot use a counter). What is the running time of this method ?

Solution:

```
def middle_node(h,t):
    '''
    INPUT: a head node h and a tail node t of a doubly linked list
           with an odd number of nodes
    OUTPUT: the middle node v of the doubly linked list
    '''

    # keep link hopping forward from the head and backward from the tail
    # until you reach the same node, which will necessarily be the middle node
    while t != h:
        t = t.prev
        h = h.next

    return t
```

6. Describe the structure and Python code/pseudocode for an array-based implementation of an index-based list that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the list. Your implementation should also provide for a constant-time `get` method.

Solution:
Maintain a capacity variable n. Also maintain the variables `indexFirst` and `indexLast`. Presuming that overflow doesn't occur, insertion at rank 0 involves inserting the element in array position `indexFirst`-1 mod n. Then `indexFirst` is updated to reflect the array index the new element was inserted into. Removal from rank 0 involves incrementing `indexFirst` mod n.

Similarly, insertion at the end of the list involves inserting the element in array position `indexLast`$+1$ mod n. Then `indexLast` is updated to reflect the array index the new element was inserted into. Removal from the end of the list involves decrementing `indexLast` mod n.

The array index of `get(x)` can be computed as `x + indexFirst` mod n.

7. Using an array-based list, describe an efficient way of putting a sequence representing a deck of n cards into random order. Use the function `random.randint(a,b)` from the `random` module, which returns a random integer between a and b inclusive. Your method should guarantee that every possible ordering is equally likely. What is the running time of your method?

Solution:
The proposed algorithm works with $O(n)$ complexity. The idea is to take the last card of the deck and swap it with any card (itself included). We then consider the deck without the last card (thus $n-1$ cards now) and repeat the process. We continue until the considered deck has only one card remaining.

```
import random

def shuffle_deck(S):
    '''
```

```
    INPUT: a sequence S representing a deck of n cards
    OUTPUT: a randomly shuffled sequence S
    '''
    # for all elements from n-1 to 0
    for i in reversed(range(len(S))):
        p = random.randint(0,i)   # pick a random position p in [0,i]
        temp = S[i]  # swap S[p] with S[i]
        S[i] = S[p]
        S[p] = temp

    return S
```

8. In the children's game "hot potato", a group of n children sit in a circle passing an object, called the "potato", around the circle (say in a clockwise direction). The children continue passing the potato until a leader rings a bell, at which point the child holding the potato must leave the game, and the other children close up the circle. This process is then continued until there is only one child remaining, who is declared the winner. Using a list, describe an efficient method for implementing this game. Suppose the leader always rings the bell immediately after the potato has been passed k times. What is the running time of your method in terms of n and k, assuming the list is implemented with a doubly linked list ? What if the list is implemented with an array ?

   Solution:
   Assume that the list is first initialized with all the participants of the game, with $O(n)$ complexity.

   Using a doubly linked list L, one simply jump from one participant to the other using the `next` links of the nodes. After k mod $L.size$ jumps (using a counter) and thus complexity $O(k)$, the current node `v` is removed from the list using the function `L.remove(v)`. We continue the process until only a single node remains in the list. Since we have $n-1$ nodes to remove in total, the complexity is $O(nk)$ if $k < n$, $O(n^2)$ otherwise (when $k > n$, the modulo reduction always ensure that we make at most $n-1$ jumps every time).

   Using an array A, we maintain a variable s that will hold the current number of participants in the array (initially $s = n$). Then, starting from a participant located at index x, one simply jump directly to the element located at index $x + k$ mod s. We remove this element e from the array by shifting all elements after e by one position down, and we of course decrement s. We continue the process until only a single element remains in the array, the winner. Since we have $n-1$ nodes to remove in total, the complexity is $O(n^2)$.

Hints

- **Question 3:** review the code for `insertAfter(p, e) in the textbook`.
- **Question 5:** consider a combined search from both ends.
- **Question 6:** think about how to extend the circular array implementation of a queue (given in the textbook).
- **Question 7:** consider randomly shuffling the deck one card at a time.