1. Provide pseudo-code descriptions for the operation insertFirst on singly linked lists, and on doubly linked lists.

The operation insertFirst gets as input a list and a key, and adds a node with the input-key at the beginning of the input-list.

Give the worst-case time complexity of your operations in terms of $O$.

Solution:

We consider the implementation of singly linked lists.

Algorithm insertFirst($L, k$):

```
new Node x
x.key := k
x.next := L.head
L.head := x
```

The worst-case time complexity of this method is in $O(1)$, because we perform in every line an operation in constant time. The (worst-case) time complexity does not depend on the size of the input.

Now for doubly-linked lists:

Algorithm insertFirst($L, k$):

```
new Node x
x.key := k
x.next := L.head
if L.head ≠ nil then
    L.head.prev := x
L.head := x
x.prev := nil
```

This algorithm is in $O(1)$.

2. Provide pseudo-code descriptions for the operation insertBefore on singly linked lists, and on doubly linked lists.

The operation insertBefore gets as input a list and a node and a key. It adds a node with the input-key before the input-node. We are given the input-node, which means that we do not have to search it.
Give the worst-case time complexity of your operations in terms of $O$.

Solution:

First we give the procedure $\text{insertAfter}$ as more or less on the slides; here $n$ is a node and $k$ is a key.

\begin{algorithm}
\textbf{Algorithm $\text{insertAfter}(L, n, k)$:}
\begin{algorithmic}
\State \textbf{new} Node $x$
\State $x.key := k$
\State $x.next := n.next$
\State $n.next := x$
\end{algorithmic}
\end{algorithm}

Pseudo-code for the operation that takes as input a list, a node in that list, and a key.

\begin{algorithm}
\textbf{Algorithm $\text{insertBefore}(L, n, k)$:}
\begin{algorithmic}
\If {$L.head = n$}
\State $\text{insertFirst}(L, k)$
\Else
\State $m := L.head$
\While {$m.next \neq n$}
\State $m := m.next$
\State $\text{insertAfter}(L, m, k)$
\EndWhile
\EndIf
\end{algorithmic}
\end{algorithm}

(Because we cannot go back, we have to traverse the list even if we are given the node.)

Now for doubly-linked lists:

Pseudo-code for the operation that takes as input a list, a node in that list, and a key.

\begin{algorithm}
\textbf{Algorithm $\text{insertBefore}(L, n, k)$:}
\begin{algorithmic}
\If {$n = L.head$}
\State $\text{insertFirst}(L, k)$
\Else
\State \textbf{new} Node $x$
\EndIf
\State $x.key := k$
\State $n.prev.next := x$
\State $x.prev := n.prev$
\State $n.prev := x$
\State $x.next := n$
\end{algorithmic}
\end{algorithm}
3. Give a non-recursive procedure in $O(n)$ that reverses a singly linked list of $n$ elements. The procedure should use no more than constant storage space beyond the space needed for the list itself. (Is your procedure also in $\Theta(n)$?)

Solution:

Algorithm reverse($L$):
\[
\begin{align*}
\text{first} & := L.first \\
\text{if } \text{first} = \text{null} \text{ or } \text{first.next} = \text{null} \text{ then} \\
\text{return } L \\
\text{else} \\
\text{second} & := \text{first.next} \\
\text{third} & := \text{second.next} \\
\text{second.next} & := \text{first} \\
\text{first.next} & := \text{null} \\
L.last & := \text{first} \\
\text{current} & := \text{third} \\
\text{previous} & := \text{second} \\
\text{while not } \text{current} = \text{null} \text{ do} \\
\text{next} & := \text{current.next} \\
\text{current.next} & := \text{previous} \\
\text{previous} & := \text{current} \\
\text{current} & := \text{next} \\
L.first & := \text{previous}
\end{align*}
\]

As for all other answers: there are often several correct approaches.

4. Give an implementation of a stack using a singly linked list. What is the top of the stack? Provide operations for push and pop in $O(1)$.

Solution:

We implement a stack $S$ using a singly linked list $L$. The top of the stack is the first node in the list. The method for push:

Algorithm push($S, k$):
\[
\text{insertFirst}(L, k)
\]

Because insertFirst for singly linked lists is in $O(1)$, this implementation of push is also in $O(1)$.

The method pop (we should add an exception for the case we pop from an empty stack):
Algorithm pop(S):
    o := L.head.key
    L.head := L.head.next
    return o

If we first have to search for a node in the list in order to delete that node, the operation for remove is in $O(n)$. However, if we already ‘have’ the node that is to be removed, the operation for remove is in $O(1)$. The method above is in $O(1)$.

5. Give an implementation of a max-priority queue using a(n unsorted) singly linked list. What is the worst-case time complexity for the operations for adding and deleting?

Solution:

We use a singly linked list $L$.

We assume that we have operations first and last and also insertFirst (in the book: listInsert) on singly linked lists. The idea for adding an element is just adding it at the beginning of the list. This is not expensive. The idea for deleting an element is first traverse the list, and then remove a maximum of the list. This operation is expensive.

Pseudo-code for insertion:

Algorithm insertPQ(k):
    insertFirst(L, k)

The worst-case complexity of this algorithm is the worst-case complexity of insertFirst for singly linked lists which is in $O(1)$.

We implement removeMax($H$), the operation removing (and returning) the maximum key from $H$ as follows.
Algorithm `removeMax(H)`:  
if `H.first = null` then  
    return error  

$m := -\infty$  
$current := H.first$  
$previous := null$  

while not $current = null$ do  
    if $m < current.key$ then  
        $m := current.key$  
        $maxprevious := previous$  
        $previous := current$  
        $current := current.next$  
    
    if $maxprevious = null$ then  
        if $H.first = H.last$ then  
            $H.first := null$  
            $H.last := null$  
        else  
            $H.first := H.first.next$  
        
        if $maxprevious.next = H.last$ then  
            $H.last := maxprevious$  
            $maxprevious.next := maxprevious.next.next$  

    return $m$

This algorithm is in $O(n)$ because we walk with `current` and `previous` together through the list to find the maximum element.

6. For the sequence

$$30 \ 20 \ 56 \ 75 \ 31 \ 19$$

and the hash function $h(k) = k \mod 11$, construct the hash table using open addressing with linear probing.

Solution:

First we calculate the addresses:
key hash try
30 8 8
20 9 9
56 1 1
75 9 9, 10
31 9 9, 10, 0
19 8 8, 9, 10, 0, 1, 2

We get the following hash table:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>56</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>20</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

7. Add the following numbers (in this order) to an initially empty hash table of length 11:

12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5

using hash function \( h(k) = (2 \cdot k + 5) \mod 11 \) and solving collision using

(a) chaining,
(b) linear probing,
(c) double hashing, with second hash function \( h'(k) = 7 - (k \mod 7) \).

Solution:
The hash function:

<table>
<thead>
<tr>
<th>( k )</th>
<th>12</th>
<th>44</th>
<th>13</th>
<th>88</th>
<th>23</th>
<th>94</th>
<th>11</th>
<th>39</th>
<th>20</th>
<th>16</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(k) )</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The hashing:

(a) using chaining

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>[16,5]</td>
<td>[44,88,11]</td>
<td>[94,39]</td>
<td>[12,23]</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Using linear probing:
Then we get the following hash table:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>39</td>
<td>20</td>
<td>5</td>
<td>16</td>
<td>44</td>
<td>88</td>
<td>12</td>
<td>23</td>
<td>13</td>
<td>94</td>
</tr>
</tbody>
</table>

(c) Using double hashing, with second hash function $h'(k) = 7 - (k \mod 7)$:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>23</td>
<td>20</td>
<td>16</td>
<td>39</td>
<td>44</td>
<td>94</td>
<td>12</td>
<td>88</td>
<td>13</td>
<td>5</td>
</tr>
</tbody>
</table>

We use the second hash function for the first time when we try to insert the key 88. It is first hashed to the slot 5, which is already taken by 44. We have $h'(88) = 3$, so after slot 5 we try slot $(5 + 3) \mod 11$ which is slot 8.

We also use the second hash function for key 11, with probe sequence 5, 7, 9, 0, and for key 39 with probe sequence 6, 9, 1, 4.

This can also be seen from the following table:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$h(k)$</td>
<td>$h'(k)$</td>
<td>try</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>5</td>
<td>3</td>
<td>5, 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>5</td>
<td>7, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>2</td>
<td>5, 7, 9, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>3</td>
<td>6, 9, 1, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>5</td>
<td>4, 9, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4, 6, 8, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. Suppose we want to define a hash function for English words which are hashed to (only) 26 slots. Why is it not a good idea to hash each word to the first letter of this word?

Solution:
The first letters of English words are not uniformly distributed over the alphabet.

9. Determine the probability that all \( n \) keys are hashed to the same place in a hash table of size \( m \) (assuming that the hash function distributes keys evenly over all places in the hash table).

Solution:
\[
\left( \frac{1}{m} \right)^n
\]

10. Come up with a real-world problem where hashing is useful.

Solution: no solution.