Part 1. Linked Lists

1. Imagine we have the linked list shown in the figure. Explain what happens if we use the statement head = head->next in the search method as follows?

```plaintext
algorithm search (ref dataOut <data>)
1. while (head is not NULL)
   1. if (head->data.key = dataOut.key)
      1. dataOut = head->data
      2. return true
   2. head = head->next
2. return false
end search
```

2. Imagine we have the method to delete an element from a list as shown in the figure. Which of the following code can be used in searching the target element (i.e. pred and tmp pointers)? Explain.

- a. pred=null; tmp = head;
  loop (tmp is not null
  and tmp.data is not the target)
  pred = pred->next;
  tmp = tmp->next;
  end loop

- b. pred=null; tmp = head;
  loop (tmp is not null
  and tmp.data is not the target)
  pred = tmp;
  tmp = tmp->next;
  end loop

3. Suppose that we have two linked lists as shown in the figure. The two pointers list1 and list2 are used to handle the first elements of the lists. What happens if we perform the statement list1=list2?

4. What would happen if we applied the following statements to the two lists in the above question?

   1. temp = list1
   2. if (temp is not null)
      1. loop (temp->link is not null)
         1. temp = temp->next
      2. end loop
      3. temp->next = list2
   3. end if

5. Let call a node in an integer linked list that has the value is a positive number be “positive node”. Write an algorithm for a method of the Linked List ADT that traverses the linked list and deletes the single node if any that is immediately followed any positive node. Note that,
for any positive node, delete only one successor node. For example, the result of the algorithm for the list

a) \{12, 13, -5, -6, 7, -8, -9, -10, 3\} is \{12, -5, -6, 7, -9, -10, 3\}
b) \{-5, -6, -7, -8, -9, -10, 13\} is \{-5, -6, -7, -8, -9, -10, 13\}

In case a), the node 13 is deleted since it is after the positive node 12. After removing 13, the algorithm does not touch nodes -5, -6 and continues to another positive node 7. The node -8 is then removed. When the last positive node 3 has no successor node, nothing will happen. In case b), the only positive node 13 has no successor node, the list is unchanged.

```
algorithm delAfterPositive()
  This algorithm traverses the linked list and deletes the302(236,385),(733,404)
  that are immediately after any node with a positive key

Pre None
Post Nodes are deleted
Return false if empty list or true otherwise
```

### Part 2. Stacks

Suppose that the following algorithms are implemented:
- `pushStack (ref stack <Stack>, val n <data>)`: push the value n to the stack stack
- `popStack(ref s <Stack>, ref n <data>)`: remove the top element of the stack stack
  and assign the data of that top element to n
- `emptyStack(val s <Stack>)`: check whether the stack stack is empty

1. Imagine we have two empty stacks of integers, s1 and s2. Draw a picture of each stack after the following operations:

```
pushStack (s1, 3);
pushStack (s1, 5);
pushStack (s1, 7);
pushStack (s1, 9);
pushStack (s1, 11);
pushStack (s1, 13);
while (!emptyStack (s1)) {
  popStack (s1, x);
  pushStack (s2, x);
} //while
```

2. Imagine we have two empty stacks of integers, s1 and s2. Draw a picture of each stack after the following operations:

```
pushStack (s1, 3);
pushStack (s1, 5);
pushStack (s1, 7);
pushStack (s1, 9);
pushStack (s1, 11);
pushStack (s1, 13);
while (!emptyStack (s1)) {
  popStack (s1, x);
  popStack (s1, x);
  pushStack (s2, x);
} //while
```
3. Write an algorithm for a function called `duplicateStack` that duplicates the contents of one stack into another. The algorithm must have two parameters of type stack, one for the source stack and one for the destination stack. The order of the elements in the stacks must be identical.

Hint: use a temporary stack to preserve the order.

```python
algorithm duplicateStack (ref sourceStack <Stack>,
                          ref destStack <Stack>)
    This algorithm copies all elements in the sourceStack into the destStack. The order of the elements of the stacks must be identical.
    Pre  None
    Post the sourceStack is copied into the destStack
    Return None

end duplicateStack
```

### Appendix

**Formal parameters and actual parameters**

Simply speaking, formal parameters are those that are declared in algorithms/functions prototypes, meanwhile actual parameters are those that are passed when the algorithms/function are actually invoked.

**Example 1.** Let’s consider the following piece of pseudo code

```python
algorithm search(val n <datatype>)
    Return position of the list element whose data is equal to n

end search

//in another algorithm
... p = Search(number)
...```

In this example, the algorithm `search` takes a formal parameter named `n` and returns the position of the list element whose data is equal to `n`. Later then, in another example algorithm, `search` is invoked with the actual parameter `number`.

**Notation of parameter passing**

When developing algorithms, we adopt the following notations regarding mechanisms of parameter passing:

- **ref**: parameters are passed by reference. When passed by reference, the actual parameter is a reference of the formal parameter. In other (and simpler) words, any value change that occurs in the formal parameter will also apply immediately on the actual parameter.

- **val**: parameters are passed by value. When the function is invoked, the value of actual parameter will be copied to that of the formal parameter. Since then, any change on formal parameters will not affect the actual parameters and vice versa.

**Example 2.** Let’s consider the following piece of pseudo code representing a method in the class `List` implementing a linked list
algorithm countPositive (val n <int>)
   This algorithm counts the number of elements whose data are positive numbers (incorrect version).
   Pre None
   Post n holds the number of elements whose data are positive numbers

1. count = 0
2. pTemp = head;
3. loop (pTemp!=NULL)
   1. if (pTemp->data > 0) then
      1. count++
   2. end if
   3. pTemp = pTemp->link
4. end loop
5. n = count

end countPositive

As easily observed, this method intends to use the parameter n to return the positive data counted. However, since n is passed by value, the value cannot be passed properly to the actual parameters when countPositive is called.

One way to correct the method is to take away the parameter and directly return the result as follows.

algorithm countPositive ()
   This algorithm counts the number of elements whose data are positive numbers
   Pre None
   Post None
   Return the number of positive elements

1. count = 0
2. pTemp = head;
3. loop (pTemp!=NULL)
   1. if (pTemp->data > 0) then
      1. count++
   2. end if
   3. pTemp = pTemp->link
4. end loop
5. return count

end countPositive

Alternatively, we can use the passing by reference mechanism to correct the method as follows.

algorithm countPositive (ref n<int>)
   This algorithm counts the number of elements whose data are positive numbers.
   Pre None
   Post n holds the number of elements whose data are positive numbers

1. count = 0
2. pTemp = head;
3. loop (pTemp!=NULL)
   1. if (pTemp->data > 0) then
      1. count++
   2. end if
   3. pTemp = pTemp->link
4. end loop
5. n = count

end countPositive

Method and global function
In Example 2, we are in the situation of developing a method for the class \textit{List}, thus we can assume that we can access the (private) internal member \textit{head} of the class. However, in some case, you may be requested to write a \textit{global function}, or \textit{function} for short, which is generally unable to access internal members of the class. In that case, your algorithm should avoid touching any class properties and instead call the suitable method.

\textbf{Example 3.} Write a \textbf{function} that counts in a list the number of elements whose data are positive numbers

\begin{verbatim}
algorithm countPositive (ref list <Linked List>, ref n <int>)
\end{verbatim}

This algorithm counts the number of elements whose data are positive numbers

\begin{verbatim}
Pre None
Post n holds the number of elements whose data are positive numbers
\end{verbatim}

1. count = 0
2. i = 0
3. loop (i < list.size())
   1. list.retrieve(i, data)
   2. if (data > 0) then
      1. count++
   3. end if
   4. i++
4. end loop
5. n = count
\textbf{end countPositive}

In this example, we assume that the class \textit{List} has already been implemented with methods \textit{size()}, which returns the number of elements in the list, and \textit{retrieve(val pos <int>, ref dataOut <data>),} which retrieves the data field of the element at the position \textit{pos} in the list and assigns that data to the \textit{dataOut} parameter.