Lab exercise: Linked lists
In this exercise, you will design and implement a set of functions (we call it a "toolkit") to manipulate linked lists. This is done in two steps:

First, you will design and implement five functions for insertion, removal and copying of linked list and test these operations thoroughly.

When you are done with the five first functions, you will design and implement a set of other operations. Through this, you will learn about nodes and linked lists, how to manipulate them, and polish off your template and pointer skills.

You are provided with an implementation of the class `Node` as a C++ template class in the `LinkedList` namespace. You are required to use this implementation for your linked list nodes, and your implementation of the functions must be in the same namespace, i.e. `LinkedList`

Note: The implementation of the linked list toolkit in this exercise deviates from the one in [Drozdek]. In [Drozdek], the linked list is a class that encapsulates the nodes. In our implementation, the linked list toolkit operates on nodes supplied from the caller. Make sure you understand this difference.

Exercise 1: The Linked list toolkit, part 1
Design (i.e. draw "cartoons" and/or write pseudocode) the following member functions for a linked list so that they obey the stated pre- and postconditions.

```
// Precondition: headPtr points to the head of a list
// Postcondition: Info has been inserted at the head of the list, and headPtr is updated
void headInsert(Node<T>*& headPtr, T info)

// Precondition: headPtr points to the head of a list
// Postcondition: The head element has been removed and headPtr is updated to point to the new
// head element
void headRemove(Node<T>*& headPtr);

// Precondition: prevPtr points to Node just before insertion point
// Postcondition: A new Node with data=info has been inserted into the list after prevPtr
void insert(Node<T>* prevPtr, T info);

// Precondition: prevPtr points to Node just before Node to remove
// Postcondition: The Node after prevPtr has been removed from the list
void remove(Node<T>* prevPtr);

// Precondition: sourcePtr is the head pointer of a linked list.
// Postcondition: A pointer to a deep copy of the linked list is returned. The original list is
// unaltered.
Node* copy(Node<T>* sourcePtr);
```

Exercise 2: Time complexity
State the time complexity of each of the methods in the toolkit.

Exercise 3: Implement your design
Have you done Exercises 1 and 2? Have you designed the methods, e.g. done cartoons, found the time complexity etc.? *Really* done it, or just *"I know how it should be done"*-done it? Shame on you. Go back to Exercise 1.

Okay, if you have *really* designed the first five operations of your linked list toolkit, you may implement them in the namespace `LinkedList`.

Exercise 4: The Linked List Toolkit, part 2

Add (i.e. design, implement and test) as many of the following operations to your linked list toolkit you can:

```
// Precondition: headPtr is the head pointer of a linked list.
// Postcondition: All nodes of the list have been deleted, and the headPtr is NULL.
void clear(Node*& headPtr);


// Precondition: headPtr is the head pointer of a linked list.
// Postcondition: The data item of each Node in the list has been printed to the screen in an
// easily readable way, e.g. “3 - 4 - 7 -/”
void print(Node* headPtr);


// Precondition: headPtr is the head pointer of a linked list.
// Postcondition: headPtr points to the start of a list that is reversed with respect to the
// original list
void reverse(Node*& headPtr);


// Precondition: splitPtr points to the node before the split point
// Postcondition: A pointer is returned that points to the first node after splitPtr. The
// original list ends at the node pointed to by splitPtr
Node* split(Node* splitPtr);


// Precondition: values points to an array of at least size n
// Postcondition: A pointer is returned that points to the head of a list in which the nodes
// contains values from the array values
Node* build(int* values, size_t n);


// Precondition: head1 and head2 each point to the head of linked lists
// Postcondition: head1 points to a list containing head1-lists' elements followed by head2-lists
// element.
void join(Node*& head1, Node* head2);


// Precondition: head points to the head of a linked list
// Postcondition: The list is rotated left by once - if it was 1-2-3-4, it is now 2-3-4-1
void rotateLeft(Node*& head);


// Precondition: head points to the head of a linked list
// Postcondition: The list is rotated right once - if it was 1-2-3-4, it is now 4-1-2-3
void rotateRight(Node*& head);
```