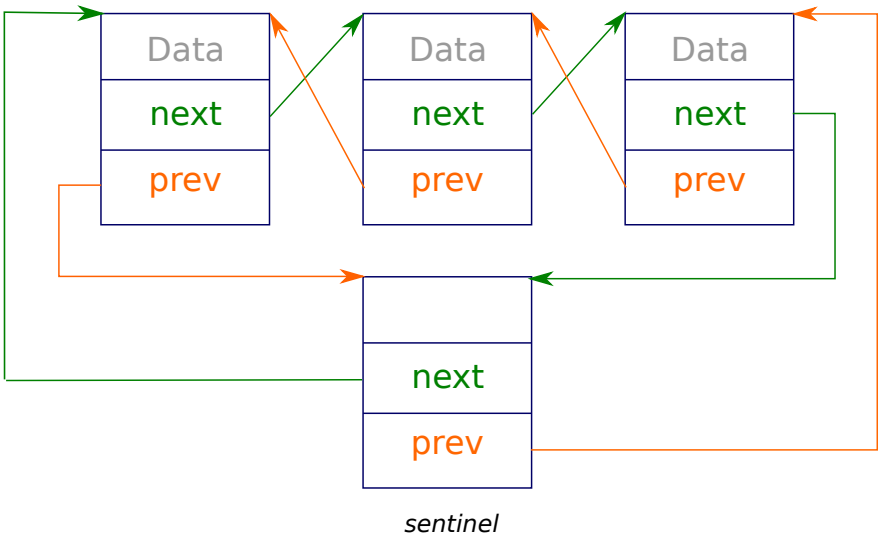
Problem 1

Doubly linked list contains nodes each of which holds a piece of data (of any type) and two pointers: to the previous node (prev) and to the next one (next). A convenient implementation consists of creating a special node (the so called *sentinel*) with irrelevant data whose member next points to the first "true" node and prev to the last one, as illustrated in the figure



This implementation simplifies operations of adding and removing elements of the list.

An empty list is represented by a sentinel whose both member pointers (prev and next) point to itself.

Write (and test) a template of class **DLL** objects of which represent doubly linked lists. Operations on the lists are defined by functions:

- constructor creating an empty list;
- **empty** — returns **true** if the list is empty;
- **push_front** — adds an element at the beginning of the list;
- **push_back** — adds an element at the end of the list;
- **print_fwd** — prints to a given stream (by default it is cout) all elements of the list (data from the nodes in one line, space separated);
- **print_rev** — as **print_fwd** but elements are printed in the reverse order;
- **find_first** — finds the first node with data equal to a given value and returns the pointer to this node (somewhat violating hermetization...) or **nullptr** if there is no such a node;
- **find_last** — as **find_first** but finds the last such node;
- **insert_after** — gets the pointer to a node (for example obtained earlier by calling **find_first**) and inserts a new node with a given data *after* it;

- **insert_before** — as **insert_after** but inserts the new node *before* the node it has got as argument;
- **delete_node** — gets the pointer to a node and deletes it; prints data from the node to be deleted;
- **reverse** — reverses the order of all nodes;
- **clear** — deletes all nodes (except for the sentinel) so the resulting object represents an empty list; prints data from the deleted nodes, so we can see that indeed all nodes which should be deleted are deleted;
- destructor deleting all the nodes, including the sentinel (it can call **clear**).

The program may have the following structure:

download *DoublyLL.cpp*

```
#include <iostream>
#include <utility>  // swap (may be useful)
#include <string>

template <typename T>
class DLL {
    struct Node {
        T     data;
        Node* next;
        Node* prev;
    };
    Node* sent; // sentinel
public:
    DLL() : sent(new Node{T(),nullptr,nullptr}) {
        sent->next = sent->prev = sent;
    }

    bool empty() const;
    void push_front(const T& t) const;
    void push_back(const T& t) const;
    void print_fwd(std::ostream& str = std::cout) const;
    void print_rev(std::ostream& str = std::cout) const;
    Node* find_first(const T& e) const;
    Node* find_last(const T& e) const;
    void insert_after(Node* a, const T& t) const;
    void insert_before(Node* b, const T& t) const;
    void delete_node(const Node* d) const;
    void reverse() const;
    void clear() const;
    ~DLL();
};

int main () {
```

```
    using std::cout; using std::endl; using std::string;

    DLL<std::string>* listStr = new DLL<std::string>();
    listStr->push_back("X");
    listStr->push_back("E");
    listStr->push_front("C");
    listStr->push_front("X");
    listStr->push_front("A");
    cout << "List printed in both directions:" << endl;
    listStr->print_fwd();
    listStr->print_rev();

    listStr->delete_node(listStr->find_first("X"));
    listStr->delete_node(listStr->find_last("X"));
    cout << "\nList after deleting X's:" << endl;
    listStr->print_fwd();

    listStr->insert_after(listStr->find_first("A"),"B");
    listStr->insert_before(listStr->find_last("E"),"D");
    cout << "List after inserting B and D:" << endl;
    listStr->print_fwd();

    listStr->reverse();
    cout << "List after reversing:" << endl;
    listStr->print_fwd();

    std::cout << "Is list empty? " << std::boolalpha
            << listStr->empty() << std::endl;
    std::cout << "Clearing the listt" << std::endl;
    listStr->clear();

    std::cout << "Adding one element (Z)" << std::endl;
    listStr->push_front("Z");

    std::cout << "Deleting the list" << std::endl;
    delete listStr;
}
```

All methods are declared as **const**, as none of them modifies the state of the object (which is the address of the sentinel).

The program above should print something like

```
List printed in both directions:
A X C X E
E X C X A
del:X del:X
```

```
List after deleting X's:
A C E
List after inserting B and D:
A B C D E
List after reversing:
E D C B A
Is list empty? false
Clearing the listt
DEL:E DEL:D DEL:C DEL:B DEL:A
Adding one element (Z)
Deleting the list
DEL:Z
```
