Introduction

Multiple data structures are often interlinked or merged to perform complicated tasks. In this assignment, you will implement such a joint data structure that we call the *double list*. A double list represents two distinct lists over the same group of elements. For simplicity, we refer to the first list as the red list and the second list as the blue list. To be more precise, at any time, you can view a double list D as a pair of ordered sequences:

$$D = (\langle r_1, r_2, \ldots, r_n \rangle, \langle b_1, b_2, \ldots, b_n \rangle)$$

where the sequences $\langle r_i \rangle$ and $\langle b_i \rangle$ are permutations of each other.

Notice that since the red and the blue lists are defined on the same collection of elements, they are not independent. One cannot update the red list without updating the blue list and vice versa. If an element is removed from the red list, it should also be removed from the blue list. If an element is inserted into the red list, then it should also be inserted into the blue list. The position of insertion, however, may differ for each list.

In this assignment, we ask you to complete the implementation of a double list that consists of *two non-circular singly linked lists*. Since the values stored in both linked lists are identical, the double list uses a single set of nodes to represent both lists at the same time. (That is, the double list does **not** create two nodes for each element.) To be able to represent two lists with a single copy of nodes, the double list nodes have two separate *next* pointers, one for the red list and one for the blue list. Similarly, the main structure maintains two *head* pointers, one for each list.

The code below declares the class template for the double list. Note that some member functions are missing; they will be introduced in the next section.

```
enum Color // Represents a list color.
{
    Red  = 0,
    Blue  = 1
};

class InvalidIndexException : public std::exception {};

template<typename T>
class DoubleList
{
    public:
        class Node // Represents a node of the double list.
        {
            friend class DoubleList;
            private:
                T mValue; // Value of the node.
                Node *mNext[2]; // A next pointer for each color.
            public:
                T value() const { return mValue; } // Getter for the value.

                // Getter for the next pointers.
                Node *next(Color color) const { return mNext[color]; }
        };

    private:
        Node *mHead[2]; // A head pointer for each color.

    public:
        DoubleList() : mHead {nullptr, nullptr} {} // Creates an empty double list.

        // Getter for the head pointers.
        Node *head(Color color) const { return mHead[color]; }
};
```

Task

You are expected to implement the following additional member functions for the `DoubleList` class.

- `DoubleList(T *a, unsigned n)`

  Constructs a double list from the given array `a` of size `n`. In this construction, both lists are constructed as identical and contain the elements of `a` in order.

- `DoubleList(const DoubleList &dl)`

  Constructs a double list that is a copy of `dl`. This copy should be a "clone", that is, it should become independent from `dl` once the construction is complete. Note: This is probably the most difficult piece and we suggest you attempt this the last.

- `Node * get(unsigned index, Color color) const`

  Returns the node at the 0-based `index` in the list whose color is `color`. If `index` is larger than or equal to the size of the list, an `InvalidIndexException` is thrown.

- `Node * insert(T value, unsigned redIndex, unsigned blueIndex)`

  Inserts a new value into the double list. The (new) 0-based positions of the value in the individual lists are given by the `redIndex` and `blueIndex` parameters. If either index is larger than the size of the original list, an `InvalidIndexException` is thrown. Notice that an index equal to 0 adds the value to the beginning of the corresponding list. An index equal to the size of the original list adds the value to the end of the corresponding list. A pointer to the node for the newly inserted element is returned.

- `void remove(unsigned index, Color color)`

  Removes a node from the double list. The node to remove is identified by its 0-based `index` in the list whose color is `color`. Again, if the index is invalid, an `InvalidIndexException` is thrown.

- `void append(DoubleList &dl)`

  Appends `dl` onto this double list and empties `dl`. The appending is performed by appending the red and the blue lists individually. To be more precise, consider two double lists A and B:

  $$A = (\langle r_1^A, \ldots, r_n^A \rangle, \langle b_1^A, \ldots, b_n^A \rangle) \qquad B = (\langle r_1^B, \ldots, r_m^B \rangle, \langle b_1^B, \ldots, b_m^B \rangle)$$

  After we call `A.append(B)`, they become:

  $$A = (\langle r_1^A, \ldots, r_n^A, r_1^B, \ldots, r_m^B \rangle, \langle b_1^A, \ldots, b_n^A, b_1^B, \ldots, b_m^B \rangle) \qquad B = (\langle\rangle, \langle\rangle)$$

- `~DoubleList()`

  Destroys this double list instance. All memory associated with the double list should be freed.

Please pay attention to the following:

- Note that we will give a fixed declaration for the `DoubleList` class. This implies that you cannot add additional members to class. We expect you to stick to the declared member fields and use them as described in this document. We may check if the member fields are correctly populated as we defined them.
- Notice that the list definitions we provide do `not` have any sentinels. An empty red/blue list is simply a `nullptr`. The last element of either list has its corresponding next pointer as `nullptr`.
- Your code should not create a memory leak. If we detect a memory leak in your implementation, you may lose a portion of the points.
- The `Node` pointers returned by the `insert()` and `get()` functions should be persistent. That is, the pointer should remain valid and point to the same element until the element is removed or the corresponding list is destroyed. The persistence should live through an `append` operation as well.
- You can assume that the users of your class does not attempt to modify the `Nodes` and/or delete them without the involvement of your functions. The users also guarantee not to read a `Node` that has been removed.

Template Parameter

The template parameter T will be guaranteed to support copy, default construction and the assigment operator.

Submission

Submission is through ODTUClass. We will provide a *DoubleList.tpp* file that contains the declaration of the `DoubleList` class and its members. We expect a single *DoubleListImpl.tpp* from you that contains the implementations of the functions. Ideally, your file should start with

```
#pragma once
#include "DoubleList.tpp"
```

We will compile your code with `g++` using the options (beware the new C++ version):
`-std=c++2a -O3 -Wall -Wextra -Wpedantic`.

Grading

We will (in the coming days) provide an auto-grader that will evaluate your functions in black-box manner. To get full points, you need to implement all functions and your implementations should adhere to the description above. You will be able to get partial points if you implement a subset of the functions. Efficiency is not critical for this homework. Unless your solution is "unreasonably" inefficient, satisfying the above requirements is sufficient for getting full points.

The grade you get from the auto-grader is not final. We may do additional black-box and white-box evaluations and adjust your grade. Solutions that do not reasonably attempt to solve the given task may lose points.

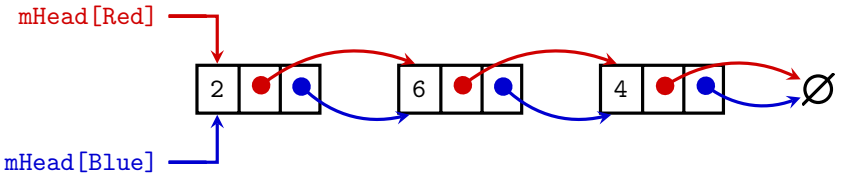
Please, start thinking about the problem early. If you do not have an idea on how you can implement the given tasks by the end of the first week, you are not well on track for this homework.

Example 1

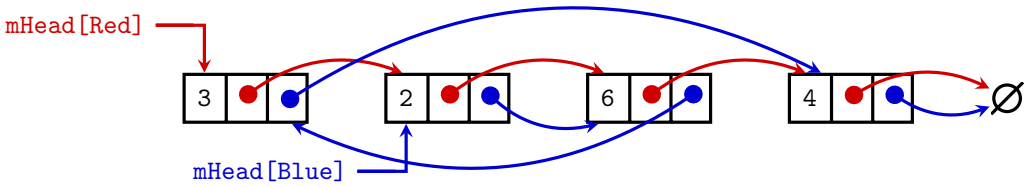
The following code:

```
int a[] = {2, 6, 4}; DoubleList<int> dl(a, 3);
```

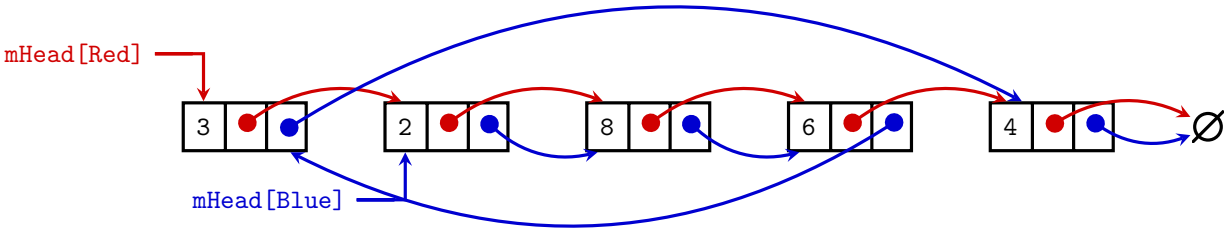
constructs the following double list:



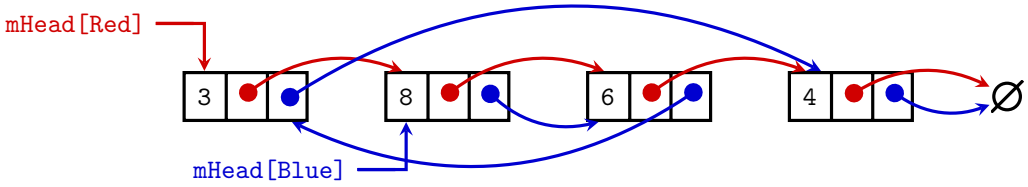
After we call, `dl.insert(3, 0, 2)`:



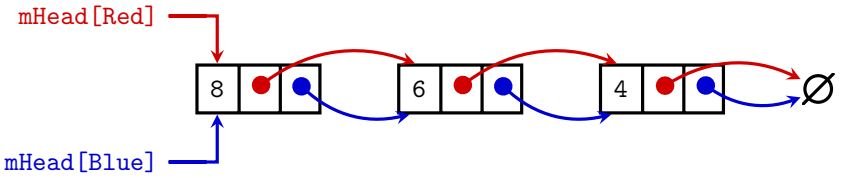
After we call, `dl.insert(8, 2, 1)`:



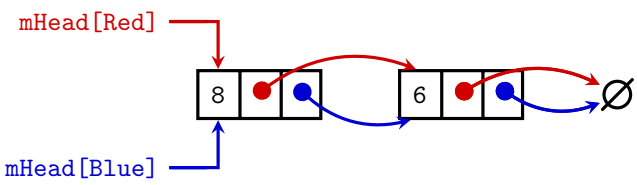
After we call `dl.remove(0, Blue)`:



After we call `dl.remove(0, Red)`:



After we call `dl.remove(2, Blue)`:



Example 2

The following code:

```
#include <iostream>
using namespace std;

#include "DoubleList.tpp"
#include "DoubleListImpl.tpp"
template <typename T>
void print(const DoubleList<T> &dl)
{
    for (auto p = dl.head(Red); p != nullptr; p = p -> next(Red))
        cout << p->value() << "␣";
    cout << "--␣";
    for (auto p = dl.head(Blue); p != nullptr; p = p -> next(Blue))
        cout << p->value() << "␣";
    cout << endl;
}

int main()
{
    DoubleList<char> dl;
    dl.insert('a', 0, 0);
    dl.insert('b', 1, 0);
    print(dl);
    cout << dl.get(0, Red)->value() << "␣" << dl.get(1, Blue)->value() << endl;

    char t[] = {'x','y'};
    DoubleList<char> other_dl(t, 2);
    dl.append(other_dl);
    print(dl);
    print(other_dl);

    dl.remove(1, Blue);
    print(dl);

    try {
        dl.remove(5, Red);
        cout << "Remove␣success!" << endl;
    } catch(const InvalidIndexException& e) {
        cout << "Invalid␣Index␣Exception" << endl;
    }

    DoubleList<char> clone_dl(dl);
    dl.insert('z', 0, 2);
    clone_dl.insert('p', 1, 0);
    print(dl);
    print(clone_dl);
    return 0;
}
```

is expected to produce the output:

```
a b -- b a
a a
a b x y -- b a x y
--
b x y -- b x y
Invalid Index Exception
z b x y -- b x z y
b p x y -- p b x y
```

Good Luck!