Universidade Federal do Rio Grande do Norte
Dept^{o} de Informática e Matemática Aplicada

Basic Data Structure I • DIM0119
◁ Implementing the List Abstract Data Type ▷
10 de junho de 2023

Overview

This document describes the *List Abstract Data Type* (ADT). We focus on two aspects of the list: the core operations that should be supported, and how the data is stored and maintained internally. For this particular project, you should use a **doubly linked list** as the data storage structure.

In the next sections we introduce the basic definition of terms related to a List ADT, its properties and operations, followed by coding aspects related to a List ADT implemented with a doubly linked list.

Sumário

1 Definition of a List

We define a *general linear list* as the set of $n \geq 0$ elements $A[0], A[1], A[2], \ldots, A[n-1]$. We say that the size of the list is n and we call a list with size $n = 0$ an **empty list**. The structural properties of a list comes, exclusively, from the relative position of its elements:-

- if $n > 0$, $A[0]$ is the first element,
- for $0 < k \leq n$, the element $A[k-1]$ precedes $A[k]$.

Therefore, the first element of a list is $A[0]$ and the last element is $A[n-1]$. The **position** of element $A[i]$ in a list is i.

In addition to the structural properties just described, a list is also defined by the set of operations it supports. Typical list operations are to print the elements in the list; to make it empty; to access one element at a specific position within a list; to insert a new element at one of the list's ends, or at a specific location within the list; to remove one element at a given location within a list, or a range of elements; to inquire whether a list is empty or not; to get the size of the list, and so on.

Depending on the **implementation** of a list ADT, we may need to support other operations or suppress some of them. The basic factor that determines which operations we may support in our implementation is their performance, expressed in terms of time complexity. For example, the time complexity of inserting elements at the beginning of a List ADT implemented with **array** is $O(n)$, with the undesired side effect of having to shift all the elements already stored in the list to make space for the new element. On the other hand, if the List ADT is implemented with a **linked list** is perfectly acceptable to support insertion at the beginning, since the cost of this operation becomes $O(1)$ for such underlying data structure.

In this document we discuss how to implement a list ADT based on *doubly linked list*. This version of a list is equivalent to the `std::list` classes of the STL library.

2 The List ADT

In this section we present the core set of operations a list ADT should support, regardless of the underlying data structure one may choose to implement a list with.

Most of the operations presented here and in the next sections follow the naming convention and behavior adopted by the STL containers.

2.1 Constructors, Destructors, and Assignment

Usually a class provides more than one type of constructor. Next you find a list of constructors that should be supported by your `list` class. In these specifications consider that the `T` represents the template type.

Notice that all references to either a return type or variable declaration related to the size of a list are defined as `size_type`. This is basically an alias to some unsigned integral type, such

as `long int`, `size_t`, for example. The use of an alias such as this is a good programming practice that enables better code maintenance. Typically we are going to define an alias at the beginning of our class definition with `typedef` or `using` keywords, as for instance in `using size_type = unsigned long`.

`list( );`	(1)
`explicit list( size_type count );`	(2)
`template< typename InputIt >` `list( InputIt first, InputIt last );`	(3)
`list( const list& other );`	(4)
`list( std::initializer_list<T> ilist );`	(5)
`~list( );`	(6)
`list& operator=( const list& other );`	(7)
`list& operator=( std::initializer_list<T> ilist );`	(8)

(1) Default constructor that creates an empty list.
(2) Constructs the list with `count` default-inserted instances of `T`.
(3) Constructs the list with the contents of the range `[first, last)`.
(4) Copy constructor. Constructs a new list with the content of the `other`. No memory should be shared in this operation.
(5) Constructs the list with the contents of the initializer list `init`.
(6) Destructs the list. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.
(7) Copy assignment operator. Replaces the contents with a copy of the contents of `other`.
(8) Replaces the contents with those identified by initializer list `ilist`.

All three `operator=()` (overloaded) assign methods return `*this` at the end, so we may have multiple assignments in a single command line, such as `a = b = c = d;`.

Parameters

`count` - the size of the list.
`value` - the value to initialize the list with.
`fist, last` - the range to copy the elements from.
`other` - another list to be used as source to initialize the elements of the list with.
`ilist` - initializer list to initialize the elements of the list with.

2.2 Access operators

These are operators that only access and return specific list information and do not alter the list's internal structure.

- `size_type size() const`: return the number of elements in the container.
- `bool empty() const`: returns `true` if the container contains no elements, and `false` otherwise.
- `const T & back() const`: returns the object at the end of the list.
- `const T & front() const`: returns the object at the beginning of the list.

2.3 Modifier operators

These are operators that change some of the list's internal structure.

- `void clear()`: remove all elements from the container. All the memory associated with the list should be released.
- `void push_front( const T & value )`: adds `value` to the front of the list.
- `void push_back( const T & value )`: adds `value` to the end of the list.
- `void pop_back()`: removes the object at the end of the list.
- `void pop_front()`: removes the object at the front of the list.

2.4 Operator overloading — non-member functions

Lastly, we need to provide a couple of binary operator on lists. They may be coded either as an external template function or as members of the `list` class. They are:

- `bool operator==( const list& lhs, const list& rhs );`: Checks if the contents of `lhs` and `rhs` are equal, that is, whether `lhs.size() == rhs.size()` and each element in `lhs` compares equal with the element in `rhs` at the same position.
- `bool operator!=( const list& lhs, const list& rhs ;`: Similar to the previous operator, but the opposite result.

In both cases, the type `T` stored in the list must be `EqualyComparable`, in other words, it must support the `operator==()`.

Note that these **are not methods nor a friend function**, but some plain-old regular functions. Therefore, the implementation of these functions must rely only on public methods provided by each class. This is an alternative way of providing operator overloading for classes. The `lhs` and `rhs` represent, respectively, the *left-hand-side* and the *right-hand-side* elements of a comparison operation.

3 Iterators

There are other operations common to all implementations of a list. These operations require the ability to insert/remove elements in the middle of the list. For that, we require the notion of *position*, which is implemented in STL as a nested type `iterator`.

Iterators may be defined informally as *a class that encapsulate a pointer to some element within the list*. This is an object oriented way of providing some degree of access to the list without exposing the internal components of the class, thus preserving the **encapsulation**[1] principle.

3.1 Getting an iterator

- `iterator begin()`: returns an iterator pointing to the first item in the list (see Figure 1).
- `const_iterator begin() const`: returns a constant iterator pointing to the first item in the list.
- `iterator end()`: returns an iterator pointing to the end mark in the list, i.e. the position just after the last element of the list.
- `const_iterator end() const`: returns a constant iterator pointing to the end mark in the list, i.e. the position just after the last element of the list.

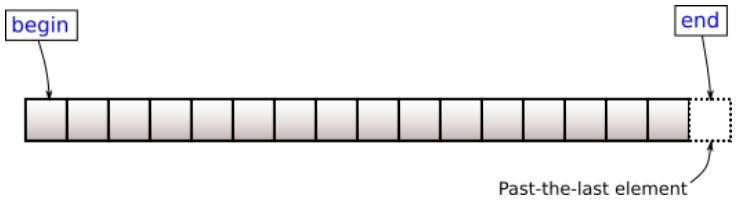

Figura 1: Visual interpretation of iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

The constant versions of the iterator are necessary whenever we need to use an iterator inside a `const` method, for instance. The `end()` method may seem a bit unusual, since it returns a pointer "out of bounds". However, this approach supports typical programming idiom to iterate along a container, as seen in the previous code example.

3.2 Iterator operations

- `operator++()`: advances iterator to the next location within the vector. This signature corresponds to the prefix form, or `++it`.
- `operator++(int)`: advances iterator to the next location within the vector. This signature corresponds to the posfix form, or `it++`.
- `operator--()`: recedes iterator to the previous location within the vector. This signature corresponds to the prefix form, or `--it`.
- `operator--(int)`: recedes iterator to the previous location within the vector. This signature corresponds to the posfix form, or `it--`.
- `operator*()` as in `*it`: return a reference to the object located at the position pointed by the iterator. The reference may or may not be modifiable.

[1] Besides **encapsulation**, the other fundamental principles of the Object-Oriented Programming paradigm are **inheritance**, **abstraction**, and **polymorphism**.

- `operator-()` as in `it1-it2`: return the difference between two iterators.
- `friend operator+(int n, iterator it)` as in `2+it`: return a iterator pointing to the `n`-th successor in the vector from `it`.
- `friend operator+(iterator it, int n)` as in `it+2`: return a iterator pointing to the `n`-th successor in the vector from `it`.
- `friend operator-(iterator it, int n)` as in `it-2`: return a iterator pointing to the `n`-th predecessor in the vector from `it`.
- `operator==()` as in `it1 == it2`: returns `true` if both iterators refer to the same location within the vector, and `false` otherwise.
- `operator!=()` as in `it1 != it2`: returns `true` if both iterators refer to a different location within the vector, and `false` otherwise.

Notice how theses operations involving iterators are (intentionally) very similar to the way we manipulate regular pointers to access, say, elements in an array.

3.3 List container operations that require iterators

- `iterator insert( iterator pos, const T & value )`: adds `value` into the list *before* the position given by the iterator `pos`. The method returns an iterator to the position of the inserted item.
- `template < typename InItr>`
  `iterator insert( iterator pos, InItr first, InItr last )`: inserts elements from the range `[first; last)` before `pos`.
- `iterator insert( const_iterator pos, std::initializer_list<T> ilist )`: inserts elements from the initializer list `ilist` before `pos`. Initializer list supports the user of insert as in `myList.insert( pos, {1, 2, 3, 4} )`, which would insert the elements 1, 2, 3, and 4 in the list before `pos`, assuming that `myList` is a list of `int`.
- `iterator erase( iterator pos )`: removes the object at position `pos`. The method returns an iterator to the element that follows `pos` before the call. This operation *invalidates* `pos`, since the item it pointed to was removed from the list.
- `iterator erase( iterator first, iterator last )`: removes elements in the range `[first; last)`. The entire list may be erased by calling `a.erase(a.begin(), a.end());`
- `template < typename InItr >`
  `void assign( InItr first, InItr last )`: replaces the contents of the list with copies of the elements in the range `[first; last)`.
- `void assign( std::initializer_list<T> ilist )`: replaces the contents of the list with the elements from the initializer list `ilist`.
  We may call, for instance, `myList.assign( {1, 2, 3, 4} )`, to replace the elements of the list with the elements 1, 2, 3, and 4, assuming that `myList` is a list of `int`.

For each operation described above, you must provide a const version, which means replacing `iterator` by `const_iterator`.

3.4 Utility operations exclusive for the linked list implementation

Here are a few list manipulation operations that can be efficiently implemented with linked lists.

3.4.1 `void merge( list &other )`

This method **merges** the following two lists into one: the list invoking the action and the list passed as argument. The result of the merge is stored in the `*this` list, which is the list calling the method. Both lists must be already sorted for the merge to work properly (precondition). Your solution should take advantage of this pre-sorted situation and perform the merge in a **linear** time complexity, $O(n+m)$, where n and m are, respectively, the lengths of the `*this` and the `other`.

You solution should not copy content of nodes from one node to another, nor create new nodes. Rather, your algorithm must rely solely on moving nodes from `other` to the target `*this` list. After the operation is concluded the `other` list must become *empty*, since it should not contain any node. On the other hand, the `*this` list must contain all the noded in sorted order.

This operation must be *stable*: for equivalent elements in the two lists, the elements from the `*this` list shall always precede the elements from other, and the order of equivalent elements of `*this` and `other` does not change. The method uses the `operator<` to compare elements.

The function should do nothing if `other` refers to the same object as `*this`. See Code 1 for an example.

Parameters

`other` - another container to transfer the content from.

Code 1 This code demonstrates the application of the `merge()` method.

```
1  #include <iostream>
2  using std::cout;
3  #include  "list.h"
4
5  std::ostream& operator<<(std::ostream& ostr, const sc::list<int>& list) {
6      for (auto &i : list)
7          ostr << " " << i;
8      return ostr;
9  }
10 int main() {
11     sc::list<int> list1 = { 0,1,3,5,9 }; // Already sorted in ascending order.
12     sc::list<int> list2 = { 2,4,6,7,8 }; // Already sorted in ascending order.
13     cout << "list1:  " << list1 << "\n"; // Output: list1:   0 1 3 5 9
14     cout << "list2:  " << list2 << "\n"; // Output: list2:   2 4 6 7 8
15     list1.merge(list2);                  // Calling merge...
16     cout << "merged: " << list1 << "\n"; // Output: merged:  0 1 2 3 4 5 6 7 8 9
17 }
```

3.4.2 `void splice( const_iterator pos, list &other )`

This method **transfers** all elements from `other` into `*this`. The elements are inserted *before* the element pointed to by `pos`. The container `other` becomes empty after the operation. The behavior is undefined if `other` refers to the same object as `*this`.

No elements are copied or moved, only the internal pointers of the list nodes are re-pointed. No iterators or references become invalidated, the iterators to moved elements remain valid, but now refer into `*this`, not into `other`. See Code 2 for an example.

Parameters

`pos` - element before which the content will be inserted.
`other` - another container to transfer the content from.

Code 2 This code demonstrates the application of the `splice()` method.

```
1  #include <iostream>
2  using std::cout;
3  #include  "list.h"
4  using namespace sc;
5
6  std::ostream& operator<<(std::ostream& ostr, const sc::list<int>& list) {
7      for (auto &i : list)
8          ostr << " " << i;
9      return ostr;
10 }
11 int main () {
12     sc::list<int> list1 = { 1, 2, 3, 4, 5 };
13     sc::list<int> list2 = { 10, 20, 30, 40, 50 };
14
15     auto it = list1.begin();
16     std::advance(it, 2); // Jump twice on the list: now 'it' points to '3'.
17
18     list1.splice(it, list2);
19     cout << "list1: " << list1 << "\n"; // Out-> list1:  1 2 10 20 30 40 50 3 4 5
20     cout << "list2: " << list2 << "\n"; // Out-> list2:
21 }
```

3.4.3 `void reverse( void )`

This method **reverses** the order of the elements in the container. No references or iterators become invalidated. This operation should be done in *linear* time complexity and should not allocate any extra memory proportional to the size of the list. In short: the time complexity must be $O(n)$ and the space (memory usage) complexity should be $O(1)$. See Code 3 for an example.

Code 3 This code demonstrates the application of the `reverse()` method.

```
#include <iostream>
using std::cout;
#include  "list.h"
using namespace sc;

std::ostream& operator<<(std::ostream& ostr, const sc::list<int>& list) {
    for (auto &i : list)
        ostr << " " << i;
    return ostr;
}
int main() {
    sc::list<int> list1 = { 0,1,2,3,4,5,6,7,8,9 }; // Sorted

    cout << "ascending:  " << list1 << "\n"; // ascending:   0 1 2 3 4 5 6 7 8 9
    list1.reverse();
    cout << "descending: " << list1 << "\n"; // descending:  9 8 7 6 5 4 3 2 1 0
}
```

3.4.4 `void unique( void )`

This method **removes all consecutive duplicate** elements from the container. Only the first element in each group of equal elements is left. The method uses `operator==` to compare the elements.

This operation should be done in *linear* time complexity and should not allocate any extra memory proportional to the size of the list. In short: the time complexity must be $O(n)$ and the space (memory usage) complexity should be $O(1)$. No elements are copied or moved, only the internal pointers of the list nodes are re-pointed. See Code 4 for an example.

3.4.5 `void sort( void )`

This method **sorts** the elements of the container in ascending order. The order of equal elements *must be* preserved (i.e. a stable sorting algorithm). The method uses `operator<` to compare the elements.

This operation should be done with approximately $n \log n$ comparisons, where n is the number of elements in the list. In short: the time complexity must be $O(n \log n)$. See Code 5 for an example.

Note that the sorting should be accomplished by modifying the linking pointers of the list **and not by copying or swapping node content among nodes**. This approach makes this method more efficient, since it does not moved/copy memory content but rather it only changes pointer's addresses.

Code 4 This code demonstrates the application of the `unique()` method.

```cpp
#include <iostream>
using std::cout;
#include  "list.h"
using namespace sc;

std::ostream& operator<<(std::ostream& ostr, const sc::list<int>& list) {
    for (auto &i : list)
        ostr << " " << i;
    return ostr;
}
int main () {
    sc::list<int> list1 = {1, 2, 2, 3, 3, 2, 1, 1, 2};

    std::cout << "original:  " << list1 << "\n"; // original:   1 2 2 3 3 2 1 1 2
    list1.unique();
    std::cout << "unique: " << list1 << "\n";    // unique:     1 2 3 2 1 2
}
```

Code 5 This code demonstrates the application of the `sort()` method.

```cpp
#include <iostream>
using std::cout;
#include  "list.h"
using namespace sc;

std::ostream& operator<<(std::ostream& ostr, const sc::list<int>& list) {
    for (auto &i : list)
        ostr << " " << i;
    return ostr;
}
int main () {
     sc::list<int> list = { 8,7,5,9,0,1,3,2,6,4 };

    std::cout << "original:  " << list1 << "\n"; // original:   8 7 5 9 0 1 3 2 6 4
    list1.sort();
    std::cout << "sorted: " << list1 << "\n";    // sorted:     0 1 2 3 4 5 6 7 8 9
}
```

4 Implementation

Your mission, should you choose to accept it, is to implement a class called `ls::list` that follows the list ADT design and stores its elements in a *doubly linked list*. Your implementation must create `head` and `tail` extra *nodes* (not pointers!) to allow for constant time complexity of some operations done on either end of the list. See Figure 2 for a abstract representation of the list with

its two head and tail nodes (doubly linked version).

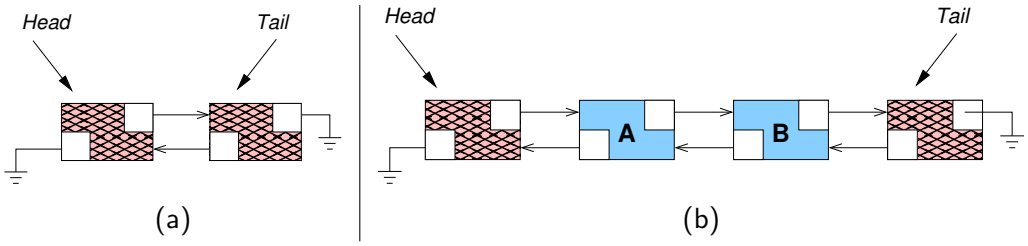

Figura 2: Abstract representation of a doubly linked list with a *head*, and *tail* nodes in (a) empty state, and (b) with two elements stored.

One possible implementation for the `list` may comprehend the following four classes:

- The main `list` class, which should create the tail and head nodes, keep track of the number of elements in the list, create/delete nodes as the list receives/looses elements, and provide a series of public methods the client code should invoke to operate on or access the list.
- The structure `Node`, which is a nested structure that describes the memory layout of a linked list node; basically it is a struct with two fields, the data content and a pointer to the next node in the list. Because the `Node` is a nested structure inside `list`, its fields may be accessed directly from within every `list`'s methods by using the '`.`' operator.
- The `const_iterator` class, which is class that represents a const iterator that may point to whichever element in the list. The class `const_iterator` stores a raw pointer to the "current" node, and provide the basic iterator operations, such as `=`, `==`, `!=` e `++`.
- The `iterator` class, which is class similar to the `const_iterator` class. This class provide the same functionality as the `const_iterator`, with the exception of `operator*()`, which returns a *regular reference* to a list item, rather than a *constant reference*. Notice that an `iterator` may be used in any method that requires a `const_iterator`, but the opposite is not true. In other words, the `iterator` *IS-A*[2] `const_iterator`.

The Codes 6 to 8 present an overview of the declaration of all suggested classes and some of their corresponding methods. Note that these code listing are only suggestions and they may be changed/adapted as long as all the requested methods' signature are not altered. Initially all methods's are **empty** or just have a *stub*, so that you can successfully compile the tests; obviously, most of the tests will fail. You may declare all classes in a single file, named `list.h`.

4.1 Unit Test Code

I **strongly** recommend that you make use of the *unit tests* provided with this project. The unit tests are an automatic way of testing each of the methods of your class under various situations. The application of the test suits during the development phase helps you pinpoint which methods are

[2]Object-oriented programming terminology for the concept of *inheritance*.

working as expected or not at all. Besides, if your class pass all tests it means you are in the right track to get full credits.

Note, however, that the unit tests do not ensure that your code is 100% correct. It only shows that the class works for some basic circumstances. Other aspects, such as memory leaks, or unforeseen bugs may not be caught by the tests alone.

5 Project Evaluation

You should hand in a complete program, without compiling errors, tested and fully documented. The assignment will be credit according to the following criteria:-

1. Correct implementation of special members (Section 2.1) (20 credits);
   (a) Regular constructor (3 credits);
   (b) Constructor (size) (3 credits);
   (c) Destructor (3 credits);
   (d) Copy constructor (3 credits);
   (e) Constructor from range (2 credits);
   (f) Constructor from initialize list (2 credits);
   (g) Assignment operator: `opereator=(list)` (2 credits);
   (h) Assignment operator: `opereator=(initializer list)` (2 credits);
2. Correct implementation of the get iterators (Section 3.1) (4 credits);
   (a) `begin()` (1 credits);
   (b) `end()` (1 credits);
   (c) `cbegin()` (1 credits);
   (d) `cend()` (1 credits);
3. Correct implementation of access operations (Section 2.2) (6 credits);
   (a) `empty()` (1 credits);
   (b) `size()` (1 credits);
   (c) `front()` (2 credits);
   (d) `back()` (2 credits);
4. Correct implementation of modifier operations (Section 2.3) (21 credits);
   (a) `clear()` (3 credits);
   (b) `push_front()` (3 credits);
   (c) `push_back()` (3 credits);
   (d) `pop_front()` (3 credits);
   (e) `pop_back()` (3 credits);
   (f) `assign(range)` (3 credits);
   (g) `assign(initialize list)` (3 credits);
5. Correct implementation of operator overloading (Section 2.4) (4 credits);

   (a) `operator==()` (2 credits);
   (b) `operator!=()` (2 credits);

6. Correct implementation of methods that require iterator (Section 3.3) (19 credits);
   (a) `insert(iterator, value)` (4 credits);
   (b) `insert(iterator, range)` (4 credits);
   (c) `insert(iterator, initializer list)` (4 credits);
   (d) `erase(iterator)` (3 credits);
   (e) `erase(range)` (4 credits);
7. Correct implementation of utility methods (Section 3.4) (40 credits);
   (a) `merge(other)` (8 credits);
   (b) `splice(pos, other)` (8 credits);
   (c) `reverse()` (8 credits);
   (d) `unique()` (8 credits);
   (e) `sort()` (8 credits);
8. Correct implementation of the iterators (Section 3) (15 credits);
   (a) Special members (3 credits);
      i. Regular constructor (1 credits);
      ii. Copy constructor (1 credits);
      iii. Assignment operator (1 credits);
   (b) Navigation methods (7 credits);:
      i. increment operator `++it` and `it++` (2 credits);
      ii. decrement operator `--it` and `it--` (2 credits);
      iii. dereference operator `*it` (1 credits);
      iv. equality/difference operators `it1==it2` and `it1!=it2` (2 credits);

The following situations may *take credits out* of your assignment, if they happen during the evaluation process:-

- Project does not support the unit tests provided (up to -20 credits)
- Compiling and/or run time errors (up to -20 credits)
- Missing code documentation in Doxygen style (up to -10 credits)
- Memory leak (up to -10 credits)
- Missing `author.md` file (up to -20 credits).

The `author.md` file (Markdown file format recommended here) should contain a brief description of the project, and how to run it. It also should describe possible errors, limitations, or issues found. Do not forget to include the author(s) name(s)!

Good Programming Practices

During the development process of your assignment, it is strongly recommend to use the following tools:-

- Doxygen: professional code documentation;
- Git: version control system;
- Valgrind: tracks memory leaks, among other features;
- gdb: debugging tool, and;
- Makefile: helps building and managing your programming projects.

Try to organize you code in several folders, such as `src` (for `.cpp` files), `include` (for header files `.h`, and `.inl`), `bin` (for `.o` and executable files) and `data` (for storing input files).

6 Authorship and Collaboration Policy

This is a pair assignment. You may work in a pair or alone. If you work as a pair, comment both members' names atop every code file, and try to balance evenly the workload. Only one of you should submit the program via Sigaa.

Any team may be called for an interview. The purpose of the interview is twofold: to confirm the authorship of the assignment and to identify the workload assign to each member. During the interview, any team member should be capable of explaining any piece of code, even if he or she has not written that particular piece of code. After the interview, the assignment's credits may be re-distributed to better reflect the true contribution of each team member.

The cooperation among students is strongly encouraged. It is accepted the open discussion of ideas or development strategies. Notice, however, that this type of interaction should not be understood as a free permission to copy and use somebody else's code. This is may be interpreted as plagiarism.

Any two (or more) programs deemed as plagiarism will automatically receive **zero** credits, regardless of the real authorship of the programs involved in the case. If your project uses a (small) piece of code from someone else's, please provide proper acknowledgment in the `author.md` file.

7 Work Submission

You may submit your work in two possible ways: via GitHub Classroom (GHC), or, via Sigaa submission task. In case you decide to send your work via GHC you **must** also send a text file via Sigaa submission task with the github link to your repository. In case you choose to send your work via Sigaa only, send a zip file containing all the code necessary to compile and run the project.

I any of these two ways, remember to remove all the executable files (i.e. the `build` folder) from your project before handing in your work. Only one team member should submit a single zip file containing the entire project. This should be done only via the SIGAA's virtual class.

◀ The End ▶

Code 6 Partial listing of the class `ls::list`. This code has references to Codes 7, and 8, which contains the declaration of related classes.

```
1  template <typename T>
2  class list {
3    private:
4      struct Node { See Code 7 };
5
6    public:
7      class const_iterator { See Code 8 };
8      class iterator : public const_iterator { See Code 8 };
9
10     // [I] SPECIAL MEMBERS
11     list( ){};
12     ~list( ){};
13     list( const list & ){};
14     list & operator= ( const list & ){return *this;};
15     list & operator= ( const std::initializer_list & il ){return *this;};
16
17     // [II] ITERATORS
18     iterator begin( ){return iterator{};};
19     const_iterator cbegin( ) const { return const_iterator{}; }
20     iterator end( ) { return iterator{}; }
21     const_iterator cend( ) const { return const_iterator{}; }
22     // [III] Access
23     int size( ) const { return 0; };
24     bool empty( ) const { return false; };
25     const T & front( ) const { return T{}; };
26     const T & back( ) const { return T{}; };
27     // [IV] Modifiers
28     void clear( ){};
29     T & front( ){ return T{}; };
30     T & back( ) { return T{}; };
31     void push_front( const T & value ){};
32     void push_back( const T & value ){};
33     void pop_front( ){};
34     void pop_back( ){};
35     // [IV-a] Modifiers with iterators
36     iterator insert( const_iterator itr, const T & value ){return iterator{};};
37     template < class InItr >
38     iterator insert( const_iterator pos, InItr first, IntItr last ){return iterator{};};
39     iterator insert( const_iterator pos, std::initializer_list<T> ilist ){return iterator{};};
40     iterator erase( const_iterator itr ){return iterator{};};
41     iterator erase( const_iterator first, const_iterator last ){return iterator{};};
42     template < class InItr >
43     void assign( InItr first, InItr last ){};
44     void assign( std::initializer_list<T> ilist ){};
45
46   private:
47     int    m_size; //!< Current number of elements stored in the list.
48     Node *m_head; //!< Pointer to the head node.
49     Node *m_tail; //!< Pointer to the tail node.
50 };
```

Code 7 Partial listing of the structure `Node`, which is part of the class `Forward_list` (see Code 6).

```
struct Node {
    T  data;      //!< Data field
    Node  *prev;  //!< Pointer to the previous node in the list.
    Node  *next;  //!< Pointer to the next node in the list.
    //<! Basic constructor.
    Node( const T & d = T{}, Node * p = nullptr, Node * n = nullptr )
      : data{d}, prev{p}, next{n} { /* Empty */ }
};
```

Code 8 Partial listing of classes `const_iterator`, and `iterator`, part of the class `Forward_list` (see Code 6).

```
class const_iterator : public bidirection_iterator{
  public:
    const_iterator( );
    const T & operator* ( ) const;
    const_iterator & operator++ ( );   // ++it;
    const_iterator operator++ ( int ); // it++;
    const_iterator & operator-- ( );   // --it;
    const_iterator operator-- ( int ); // it--;
    bool operator== ( const const_iterator & rhs ) const;
    bool operator!= ( const const_iterator & rhs ) const;

  protected:
    Node *current;
    const_iterator( Node * p ) : current( p );
    friend class List<T>;
};
class iterator : public const_iterator {
  public:
    iterator( ) : const_iterator() { /* Empty */ }
    const T & operator* ( ) const;
    T & operator* ( );

    iterator & operator++ ( );
    iterator operator++ ( int );
    iterator & operator-- ( );
    iterator operator-- ( int );

  protected:
    iterator( Node *p ) : const_iterator( p );
    friend class List<Object>;
};
```