

Sequence template class

Documentation

Author : Wiktor Łazarski

Index number : 281875

Field of study : Computer Science

Faculty : Electronics and Information Technology, Warsaw University of Technology

Table of content

1. General information 3

- 1.1 *Sequence* template overview 3
- 1.2 Template parameters 3
- 1.3 Member types 4
- 1.4 Member functions 5
- 1.5 Non-member functions 6

2. Inner classes 7

- 2.1 *SequenceInvalidArgument* exception class 7
- 2.2 *iterator* and *const_iterator* class 7

3. Member and non-member functions implementation details 8

- 3.1 Member functions 8
 - 3.1.1 Standard member functions 8
 - 3.1.2 Operators 9
 - 3.1.3 Element access 11
 - 3.1.4 Iterators 12
 - 3.1.5 Capacity 13
 - 3.1.6 Modifiers 14
 - 3.1.7 Operations 17
- 3.2 Non-member functions 19

4. Shuffle method 20

- 4.1 Before invoking *shuffle* function 20
- 4.2 Implementation 20

5. Testing approach 22

- 5.1 *ErrorMessenger* class 22
- 5.2 Organisation of tests 23

1.1 *Sequence* template overview

Sequence class was implemented and compiled by MinGW for Windows compiler in CodeBlocks. Flags that were used during compilation are as follow :

```
mingw32-g++.exe -Wall -fexceptions -O2 -std=c++11
```

C++11 standard is highly recommended for proper use of *Sequence* class template.

Link to full project : <https://github.com/Wiktos/Sequence>

Sequence is a class template implemented as a single linked list, abstract data structure. It supports constant complexity of insertion and removal of new elements, *Nodes*. Template class code is available in *sequence.h* file. Implementation of all methods and *SequenceInvalidArgument* class are available in *sequence.hpp*. *iterator* class implementation is available in *seq_iterator.hpp*. To call *shuffle* method on two *Sequences* you need to include *shuffle.h* file where this method is implemented with others ones supporting shuffling process.

1.2 Template parameters

```
template <typename Key, typename Info>
class Sequence
```

Key	Typename of key by which <i>Node</i> will be recognized in <i>Sequence</i> .
Info	Typename of data stored in particular <i>Node</i> .

1.3 Member types

Private member types :

Member type	Definition
<code>struct Node</code>	<i>Node</i> is a structure that contain Key value, Info value and pointer to next <i>Node</i> .
<code>Node *head</code>	<i>Node</i> pointer to the start of <i>Sequence</i> .
<code>std::size_t length</code>	<i>length</i> private variable that contain information about current <i>Sequence</i> size.

Public member types :

Member type	Definition
<code>class SequenceInvalidArgument</code>	<i>SequenceInvalidArgument</i> is an exception class. <i>SequenceInvalidArgument</i> exception are thrown where member function is called with invalid arguments.
<code>class iterator</code>	Design pattern <i>iterator</i> class
<code>typedef const iterator const_iterator</code>	<i>const iterator</i> allows to perform only methods from <i>iterator</i> class which are <i>const</i> .

1.4 Member functions

Constructors :

<code>Sequence()</code> <code>noexcept</code> ;
<code>Sequence(const Sequence<Key, Info>&);</code>
<code>Sequence(Sequence<Key, Info>&& source)</code> <code>noexcept</code> ;

(All constructor are defined and implemented in header file)

Operators :

<code>Sequence<Key, Info>& operator=(const Sequence<Key, Info>& rhs)</code>
<code>Sequence<Key, Info>& operator=(Sequence<Key, Info>&& rhs)</code> <code>noexcept</code>
<code>Sequence<Key, Info> operator+(const Sequence<Key, Info>& rhs)</code> <code>const</code>
<code>Sequence<Key, Info>& operator+=(const Sequence<Key, Info>& rhs)</code>
<code>bool operator==(const Sequence<Key, Info>& rhs)</code> <code>const</code> <code>noexcept</code>
<code>bool operator!=(const Sequence<Key, Info>& rhs)</code> <code>const</code> <code>noexcept</code>

Element access :

<code>Info& front();</code>
<code>const Info& front()</code> <code>const</code> ;
<code>Info& back();</code>
<code>const Info& back()</code> <code>const</code> ;

Iterators :

<code>iterator begin();</code>
<code>const_iterator begin()</code> <code>const</code> ;

Capacity :

<code>bool is_empty()</code> <code>const</code> <code>noexcept</code> ;
<code>std::size_t size()</code> <code>const</code> <code>noexcept</code> ;

1.4 Member functions

Modifiers :

<code>void push_front(const Key& key, const Info& info;</code>
<code>void push_back(const Key& key, const Info& info);</code>
<code>void insert_after(const Key& loc, const Key& new_key, const Info& new_info, int key_occurence = 1);</code>
<code>void pop_front() noexcept;</code>
<code>void pop_back() noexcept;</code>
<code>void remove(const Key& loc, int key_occurence = 1);</code>
<code>void clear() noexcept;</code>
<code>void swap(Sequence<Key, Info>& seq) ;</code>

Operations :

<code>Sequence<Key, Info> subsequence(const Key& loc, int size, int key_occurence=1) const;</code>
<code>Sequence<Key, Info> subsequence(const_iterator begin, const_iterator end) const;</code>
<code>Sequence<Key, Info> merge(const Sequence<Key, Info> seq) const;</code>
<code>bool compare(const Sequence<Key, Info>& rhs, std::function<bool(const Sequence<Key, Info>&, const Sequence<Key, Info>&>> comparator);</code>
<code>bool contain(const Key& loc, int key_occurence = 1) const</code>

Destructor :

<code>~Sequence() noexcept;</code>

1.5 Non-member functions

Operator :

<code>template <typename K, typename I> friend std::ostream& operator<<(std::ostream& os, const Sequence<K, I>& seq) (private member function)</code>
--

2.1 *SequenceInvalidArgument* exception class

SequenceInvalidArgument is an exception class that derive from *std::invalid_argument*. This class does not really extends properties of *std::invalid_argument* but just change its name. The purpose of having that class is to throw exception with different *typeid* than the *std::invalid_argument* one so it will be easier to define that method from *Sequence* template class throws it.

Implementation :

```
template <typename Key, typename Info>
class Sequence<Key, Info>::SequenceInvalidArgument final :public std::invalid_argument
{
public:
    using std::invalid_argument::invalid_argument;
};
```

2.2 *iterator* and *const_iterator* class

Iterator design pattern for *Sequence*. Provide basic operators on *Node* pointer, which is hide in *private* section of that class. *iterator* class also contain inner public structure called *NodeView* :

```
struct NodeView
{
    Key& key;
    Info& info;
};
```

I created this structure to avoid returning *Node* to "external world". I wanted to avoid possibility to get to *Node* object. *NodeView* contain all information from *Node* despite pointer to next *Node* object. Such solution minimize possibility of outer class changes of next pointer.

To see the hole implementation of this class go to *seq_iterator.hpp* file.

const_iterator is just a typedef name for *const iterator* :

```
typedef const iterator const_iterator;
```

Defining an object of *const_iterator* class allows you to call only those method of that class on that object which contain *const* at the end of declaration.

3.1 Member functions

3.1.1 Standard member functions

Default constructor
<code>Sequence() noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Assign <i>head</i> equals <i>nullptr</i> and <i>length</i> equals 0.

Copy constructor
<code>Sequence(const Sequence<Key, Info>& source);</code>
Parameters : source – constant reference to <i>Sequence</i> copy pattern
Complexity : linear $O(n)$
Exception : <code>std::bad_alloc</code> may be thrown
Notes : Copy constructor call <i>operator=</i> .

Move constructor
<code>Sequence(Sequence<Key, Info>&& source) noexcept;</code>
Parameters : source – reference to <i>Sequence</i> that will be moved to current created object
Complexity : constant $O(1)$
Exception : exception safe
Notes : Set source as an empty set.

Destructor
<code>~Sequence() noexcept;</code>
Parameters : none
Complexity : linear $O(n)$
Exception : exception safe
Notes : Call <i>clear</i> function that delete all allocated memory.

3.1 Member functions

3.1.2 Operators

Copy assignment operator =
<code>Sequence<Key, Info>& operator=(const Sequence<Key, Info>& rhs);</code>
Parameters : rhs – constant reference to <i>Sequence</i> copy pattern
Complexity : linear $O(n)$
Exception : <code>std::bad_alloc</code> may be thrown
Notes : Return reference to <i>this</i> .

Move assignment operator =
<code>Sequence<Key, Info>& operator=(Sequence<Key, Info>&& rhs) noexcept;</code>
Parameters : rhs – reference to <i>Sequence</i> that will be moved to our object
Complexity : constant $O(1)$
Exception : exception safe
Notes : Set rhs <i>Sequence</i> as an empty set.

Add operator +
<code>Sequence<Key, Info> operator+(const Sequence<Key, Info>& rhs) const;</code>
Parameters : rhs – constant reference to <i>Sequence</i> that will be merged with object
Complexity : linear $O(n)$
Exception : <code>std::bad_alloc</code> may be thrown
Notes : Method does not modify our object, call merge.

Add - equal operator +=
<code>Sequence<Key, Info>& operator+=(const Sequence<Key, Info>& rhs);</code>
Parameters : rhs – constant reference to <i>Sequence</i> that will be added to the end of object
Complexity : linear $O(n)$
Exception : <code>std::bad_alloc</code> may be thrown
Notes : Method does modify our object, call merge and assign operator =.

3.1 Member functions

Compare operator ==
<code>bool operator==(const Sequence<Key, Info>& rhs) const noexcept;</code>
Parameters : rhs – constant reference to <i>Sequence</i> that will compared with object
Complexity : linear $O(n)$
Exception : exception safe
Notes : First function compare sizes and then each <i>Node</i> .

Compare operator !=
<code>bool operator!=(const Sequence<Key, Info>& rhs) const noexcept;</code>
Parameters : rhs – constant reference to <i>Sequence</i> that will compared with object
Complexity : linear $O(n)$
Exception : exception safe
Notes : Returns negation of operator == comparison.

3.1 Member functions

3.1.3 Element access

Get data stored in first <i>Node</i> in <i>Sequence</i>
<code>Info& front();</code>
Parameters : none
Complexity : constant $O(1)$
Exception : <code>std::runtime_error</code> thrown if <i>Sequence</i> is empty
Notes : Return reference to data so it is possible to change data stored in <i>Node</i> .

Get data stored in first <i>Node</i> in <i>Sequence</i> only for reading
<code>const Info& front() const;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : <code>std::runtime_error</code> thrown if <i>Sequence</i> is empty
Notes : Return constant reference to data so it is only possible to read data stored in <i>Node</i>

Get data stored in last <i>Node</i> in <i>Sequence</i>
<code>Info& back();</code>
Parameters : none
Complexity : linear $O(n)$
Exception : <code>std::runtime_error</code> thrown if <i>Sequence</i> is empty
Notes : Return reference to data so it is possible to change data stored in <i>Node</i> .

Get data stored in last <i>Node</i> in <i>Sequence</i> only for reading
<code>const Info& back() const;</code>
Parameters : none
Complexity : linear $O(n)$
Exception : <code>std::runtime_error</code> thrown if <i>Sequence</i> is empty
Notes : Return constant reference to data so it is only possible to read data stored in <i>Node</i>

3.1 Member functions

3.1.4 Iterators

Get <i>iterator</i> class object
<code>iterator begin() noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Return <i>iterator</i> with pointer set at the beginning of <i>Sequence</i> or return <i>nullptr</i> if <i>Sequence</i> is empty.

Get <i>const_iterator</i> class object
<code>const_iterator begin() const noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Return <i>const_iterator</i> with pointer set at the beginning of <i>Sequence</i> or return <i>nullptr</i> if <i>Sequence</i> is empty.

3.1 Member functions

3.1.5 Capacity

Check if <i>Sequence</i> is empty
<code>bool is_empty() const noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Check if <i>length</i> == 0.

Check if <i>Sequence</i> is empty
<code>std::size_t size() const noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Return current value of <i>length</i> .

3.1 Member functions

3.1.6 Modifiers

Add <i>Node</i> at the beginning of <i>Sequence</i>
<pre>void push_front(const Key& key, const Info& info);</pre>
Parameters : key – element’s key that will be stored in new <i>Node</i> info – element’s info that will be stored in new <i>Node</i>
Complexity : constant $O(1)$
Exception : <i>std::bad_alloc</i> may be thrown
Notes : none

Add <i>Node</i> at the end of <i>Sequence</i>
<pre>void push_back(const Key& key, const Info& info);</pre>
Parameters : key – element’s key that will be stored in new <i>Node</i> info – element’s info that will be stored in new <i>Node</i>
Complexity : linear $O(n)$
Exception : <i>std::bad_alloc</i> may be thrown
Notes : none

Add <i>Node</i> in the middle of <i>Sequence</i>
<pre>void insert_after(const Key& loc, const Key& new_key, const Info& new_info, int key_occurrence = 1)</pre>
Parameters : loc – key of element which will be before new <i>Node</i> new_key – element’s key that will be stored in new <i>Node</i> new_info – element’s info that will be stored in new <i>Node</i> key_occurrence – specify after which occurrence of loc user want to insert new <i>Node</i>
Complexity : linear $O(n)$
Exception : <i>SequenceInvalidArgument</i> is thrown when list is empty or key_occurrence < 1 <i>std::bad_alloc</i> may be thrown
Notes : key_occurrence is set by default to 1.

3.1 Member functions

Remove first <i>Node</i> from <i>Sequence</i>
<code>void pop_front() noexcept;</code>
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : none

Remove last <i>Node</i> from <i>Sequence</i>
<code>void pop_back() noexcept;</code>
Parameters : none
Complexity : linear $O(n)$
Exception : exception safe
Notes : none

Remove <i>Node</i> from the middle of <i>Sequence</i>
<code>void remove(const Key& loc, int key_occurrence = 1);</code>
Parameters : loc – key of element which will be before new <i>Node</i> key_occurrence – specify after which occurrence of loc user want to insert new <i>Node</i>
Complexity : linear $O(n)$
Exception : <i>SequenceInvalidArgument</i> is thrown when list is empty or key_occurrence < 1
Notes : key_occurrence is set by default to 1.

Remove all <i>Nodes</i> from <i>Sequence</i>
<code>void clear() noexcept;</code>
Parameters : none
Complexity : linear $O(n)$
Exception : exception safe
Notes : Clearing is pernament.

3.1 Member functions

Swap Sequences
<code>void swap(Sequence<Key, Info>& seq);</code>
Parameters : seq – reference to <i>Sequence</i> which will swap content with my object
Complexity : linear $O(n)$
Exception : <i>std::bad_alloc</i> may be thrown
Notes : Calls <i>operator=</i> .

3.1 Member functions

3.1.7 Operations

Get subsequence of <i>Sequence</i>
<code>Sequence<Key, Info> subsequence(const Key& loc, int size, int key_occurrence = 1) const;</code>
Parameters : loc – key of element which will be first in subsequence size – specify the size of subsequence key_occurrence – specify after which occurrence of loc user want to start subsequence
Complexity : linear $O(n)$
Exception : <i>SequenceInvalidArgument</i> is thrown if size or key_occurrence are less than 1 <i>std::bad_alloc</i> may be thrown
Notes : key_occurrence is set by default to 1.

Get subsequence of <i>Sequence</i>
<code>Sequence<Key, Info> subsequence(const_iterator begin, const_iterator end) const;</code>
Parameters : begin – specify the starting <i>Node</i> of subsequence end – specify the last non inserted <i>Node</i> in subsequence
Complexity : linear $O(n)$
Exception : <i>std::bad_alloc</i> may be thrown
Notes : Method does not check if begin iterator is before end so it may cause program call <i>terminate</i> .

Merge <i>Sequences</i>
<code>Sequence<Key, Info> merge(const Sequence<Key, Info> seq) const;</code>
Parameters : seq – <i>Sequence</i> that will be added to the end of ours <i>Sequence</i> object
Complexity : linear $O(n)$
Exception : <i>std::bad_alloc</i> may be thrown
Notes : Method does not modify our object.

3.1 Member functions

Compare <i>Sequences</i>
<pre>bool compare(const Sequence<Key, Info>& rhs, std::function<bool(const Sequence<Key, Info>&, const Sequence<Key, Info>&)> comparator);</pre>
Parameters : rhs – right hand side of comparison comparator – specify the way <i>Sequences</i> will be compared
Complexity : unknown because depends from comparator
Exception : unknown because depends from comparator
Notes : none

Check if <i>Sequences</i> contain <i>Node</i>
<pre>bool contain(const Key& loc, int key_occurence = 1) const;</pre>
Parameters : loc – key of element looked for key_occurence – specify after the occurrence of loc in <i>Sequence</i>
Complexity : linear $O(n)$
Exception : <i>SequenceInvalidArgument</i> is thrown if key_occurence < 1
Notes : none

3.2 Non-member functions – operator <<

Output stream operator <<
<pre>template <typename K, typename I> friend std::ostream& operator<<(std::ostream& os, const Sequence<K, I>& seq);</pre>
Parameters : os – output stream seq – <i>Sequence</i> object that will be inserted to os
Complexity : linear $O(n)$
Exception : unknown depends from os
Notes : none

4.1 Before invoking *shuffle* function

For proper working shuffle method we need to check all integer that are contain in argument list of shuffle method. To check whether starting points and lenghts are correct *shuffle* function call *check_shuffle_argument*. *repeat* argument is not checked by this function because I treat it as an anchor of my recursive function *shuffle*.

If you want to increase performance of *shuffle* method remove line :

```
if(!check_shuffle_argument(start1, len1, start2, len2))  
    return retv;
```

from it but then don't forget that *shuffle* function may *terminate* your program.

By default this part of code is inserted.

Checking <i>shuffle</i> arguments method
Full implementation : <pre>bool check_shuffle_argument(int start1, int len1, int start2, int len2){ return (start1 >= 0) && (len1 >= 0) && (start2 >= 0) && (len2 >= 0); }</pre>
Parameters : start1 – starting point of first <i>Sequence</i> len1 – number if element we want to insert from first <i>Sequence</i> before moving to second <i>Sequence</i> start2 – starting point of second <i>Sequence</i> len2 – number if element we want to insert from second <i>Sequence</i> before moving to first <i>Sequence</i>
Complexity : constant $O(1)$
Exception : exception safe
Notes : Function check if arguments are proper. Negate result to check if arguments are improper.

4.2 Implementation

shuffle function is a recursive function. It is not a member function of *Sequence* class. Method returns empty *Sequence* if any of integer arguments is invalid. The result is create with the following algorithm :

- 1) Take *len1* arguments from first *Sequence* starting from *start1* position.
- 2) Take *len2* arguments from second *Sequence* starting from *start2* position.
- 3) Decrease by 1 *repeat* argument.
- 4) Go to 1 if *repeat* greater than 0.

<i>shuffle</i> function
<p>Full implementation :</p> <pre> template <typename Key, typename Info> Sequence<Key, Info> shuffle(const Sequence<Key, Info>& s1, int start1, int len1, const Sequence<Key, Info>& s2, int start2, int len2, int repeat) { Sequence<Key, Info> retv; if(!check_shuffle_argument(start1, len1, start2, len2)) return retv; if(repeat <= 0) return retv; //getting value from first list if(static_cast<unsigned int>(start1 + len1) <= s1.size()) retv += s1.subsequence(s1.begin() + start1, s1.begin() + (start1 + len1)); else{ if(static_cast<unsigned int>(start1) < s1.size()) retv += s1.subsequence(s1.begin() + start1, s1.begin() + s1.size()); } //getting value from second list if(static_cast<unsigned int>(start2 + len2) <= s2.size()) retv += s1.subsequence(s2.begin() + start2, s2.begin() + (start2 + len2)); else{ if(static_cast<unsigned int>(start2) < s2.size()) retv += s1.subsequence(s2.begin() + start2, s2.begin() + s2.size()); } return retv + shuffle(s1, start1 + len1, len1, s2, start2 + len2, len2, repeat - 1); } </pre>
<p>Parameters : s1 – first <i>Sequence</i> start1 – starting point of first <i>Sequence</i> len1 – number if element we want to insert from first <i>Sequence</i> before moving to second <i>Sequence</i> s2 – second <i>Sequence</i> start2 – starting point of second <i>Sequence</i> len2 – number if element we want to insert from second <i>Sequence</i> before moving to first <i>Sequence</i> repeat – number of algorithm calls</p>
<p>Complexity : linear $O(n)$</p>
<p>Exception : <i>std::bad_alloc</i> may be thrown</p>
<p>Notes : Function does not modify <i>Sequence s1</i> or <i>Sequence s2</i>. Recursion may <i>terminate</i> your program if stack is overflowed.</p>

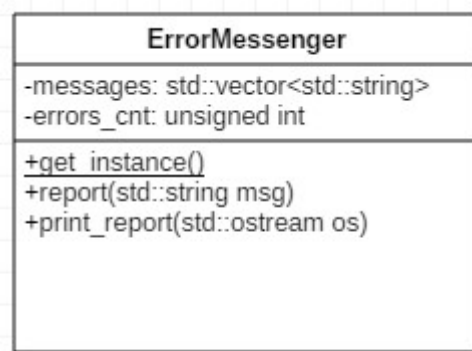
5.1 *ErrorMessenger* class

ErrorMessenger is a class implemented based on Singleton design pattern for collecting all fail tests information. Class belongs to *seq_test* namespace.

Contain two methods :

- **void** report(std::string msg) – function add messege to vector of messeges. Called when test fail.
- **void** print_report(std::ostream& os) – function called at the end of tests. To print number of errors and all messeges.

UML



Full implementation of *ErrorMessenger* class is placed in folder *./tests/*.

5.2 Organisation of tests

All test are performed in *main.cpp* file. It could be find in *./source_code/* folder. Performing test code looks as follow :

```
int main()
{
    seq_test::test_constructors();
    seq_test::test_push_front_and_push_back();
    seq_test::test_insert_after();
    seq_test::test_pop_back();
    seq_test::test_pop_front();
    seq_test::test_remove();
    seq_test::test_clear();
    seq_test::test_subsequence();
    seq_test::test_merge_and_binary_op();
    seq_test::test_accessing_elem_methods();
    seq_test::test_contain_method();
    seq_test::test_compare_method();
    seq_test::test_iterator_class();
    seq_test::test_swap_method();

    shuffle_test::test_check_param_method();
    shuffle_test::test_shuffle_method();

    seq_test::error_messenger().print_report(std::cout);
    return 0;
}
```

Implementation of particular test could be find in *./tests/* folder.

All *Sequence* test functions are declared in *seq_tests.h* and implementation of them is in file *seq_tests.cpp*. Tests belongs to namespace *seq_test*.

All *shuffle* function test method are declared in *shuffle_tests.h* and implementation of them is in file *shuffle_tests.cpp*. Tests belongs to namespace *shuffle_test*.