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

1.1 *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 is 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 suppors constant complexity of insertion and removal of new elements, *Node*. Template class code is available in *sequence.h* file. Implementation of all methods and *SequenceInvalidArgument* class are available in *sequence.tpp*. *iterator* class implementation is available in *seq_iterator.tpp*. To call *shuffle* method on two *Sequence* you need to include *shuffle.h* file where this method is implemented with others supported shuffling process.

1.2 Template parameters

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

`Key`	Typename of key by which *Node* will be recognized in *Sequence*.
`Info`	Typename of data stored in particular *Node*.

1.3 Member types

Private member types :

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

Public member types :

Member type	Definition
`class SequenceInvalidArgument`	*SequenceInvalidArgument* is an exception class. *SequenceInvalidArgument* exception are thrown where member function is called with invalid arguments.
`class iterator`	*iterator* class
`typedef const iterator const_iterator`	*const iterator* allows to perform only methods from *iterator* class which are *const*.

1.4 Member functions

Constructors :

```
Sequence() noexcept;
```

```
Sequence(const Sequence<Key, Info>&);
```

```
Sequence(Sequence<Key, Info>&&);
```

(All constructor are defined and implemented in header file)

Operators :

```
Sequence<Key, Info>& operator=(const Sequence<Key, Info>& rhs)
```

```
Sequence<Key, Info>& operator=(Sequence<Key, Info>&& rhs)
```

```
Sequence<Key, Info> operator+(const Sequence<Key, Info>& rhs) const
```

```
Sequence<Key, Info>& operator+=(const Sequence<Key, Info>& rhs)
```

```
bool operator==(const Sequence<Key, Info>& rhs) const noexcept
```

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

Element access :

```
Info& front();
```

```
const Info& front() const;
```

```
Info& back();
```

```
const Info& back() const;
```

Iterators :

```
iterator begin();
```

```
const_iterator begin() const;
```

Capacity :

```
bool is_empty() const noexcept;
```

```
std::size_t size() const noexcept;
```

1.4 Member functions

Modifiers :

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

Operations :

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

Destructor :

`~Sequence() noexcept;`

1.5 Non-member functions

Operator :

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

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 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 minimalize possibility of outer class changes of next pointer.

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

const_iterator is just a typedefine 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
`Sequence() noexcept;`
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Assign *head* equals *nullptr* and *lenght* equals 0

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

Move constructor
`Sequence(Sequence<Key, Info>&& source);`
Parameters : source – reference to *Sequence* that will be moved to current created object
Complexity : linear $O(n)$
Exception : *std::bad_alloc* may be thrown
Notes : Move constructor call *operator=*

Destructor
`Sequence(Sequence<Key, Info>&& source);`
Parameters : source – reference to *Sequence* that will be moved to current created object
Complexity : linear $O(n)$
Exception : exception safe
Notes : Call *clear* funtion that delete all allocated memory

3.1 Member functions

3.1.2 Operators

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

Move assigment operator =
`Sequence<Key, Info>& operator=(Sequence<Key, Info>&& rhs);`
Parameters : rhs – reference to *Sequence* that will be moved to current created object
Complexity : linear $O(n)$
Exception : *std::bad_alloc* may be thrown
Notes : Call copy assigment operator and return reference to *this*

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

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

3.1 Member functions

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

Compare operator ==
`bool operator!=(const Sequence<Key, Info>& rhs) const noexcept;`
Parameters : rhs – constant reference to *Sequence* 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 *Node* in *Sequence*
`Info& front();`
Parameters : none
Complexity : constant $O(1)$
Exception : *std::runtime_error* thrown if *Sequence* is empty
Notes : Return reference to data so it is possible to change data stored in *Node*

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

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

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

3.1 Member functions

3.1.4 Iterators

Get *iterator* class object
`iterator begin() noexcept;`
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Return *iterator* with pointer set at the beginning of *Sequence* or *nullptr* if *Sequence* is empty

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

3.1 Member functions

3.1.5 Capacity

Check if *Sequence* is empty
`bool is_empty() const noexcept;`
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Check if *length* == 0

Check if *Sequence* is empty
`std::size_t size() const noexcept;`
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : Return current value of *lenght*

3.1 Member functions

3.1.6 Modifiers

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

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

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

3.1 Member functions

Remove first *Node* from *Sequence*
`void pop_front() noexcept;`
Parameters : none
Complexity : constant $O(1)$
Exception : exception safe
Notes : none

Remove last *Node* from *Sequence*
`void pop_back() noexcept;`
Parameters : none
Complexity : linear $O(n)$
Exception : exception safe
Notes : none

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

Remove all *Nodes* from *Sequence*
`void clear() noexcept;`
Parameters : none
Complexity : linear $O(n)$
Exception : exception safe
Notes : Clearing is pernament

3.1 Member functions

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

3.1 Member functions

3.1.7 Operations

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

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

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

3.1 Member functions

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

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

3.2 Non-member functions – operator <<

Output stream operator <<
`template <typename K, typename I>` `friend std::ostream& operator<<(std::ostream& os, const Sequence<K, I>& seq);`
Parameters : os – output stream seq – *Sequence* 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 threat it as an anchor of my recursive funcion *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 *shuffle* arguments method

Full implementation :
 	
`bool check_shuffle_argument(int start1, int len1, int start2, int len2){`
`    return (start1 >= 0) && (len1 >= 0) && (start2 >= 0) && (len2 >= 0);`
`}`

Parameters : start1 – starting point of first *Sequence* len1 – number if element we want to insert from first *Sequence* before moving to second *Sequence* start2 – starting point of second *Sequence* len2 – number if element we want to insert from second *Sequence* before moving to first *Sequence*
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.

***shuffle* function**

Full implementation :
 	
`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);`
`}`

Parameters : s1 – first *Sequence* start1 – starting point of first *Sequence* len1 – number if element we want to insert from first *Sequence* before moving to second *Sequence* s2 – second *Sequence* start2 – starting point of second *Sequence* len2 – number if element we want to insert from second *Sequence* before moving to first *Sequence* repeat – number of algorithm calls
Complexity : linear $O(n)$
Exception : *std::bad_alloc* may be thrown
Notes : Function does not modify *Sequence s1* or *Sequence s2*. Recursion may *terminate* your program if stack is overflowded

5.1 *ErrorMessenger* class

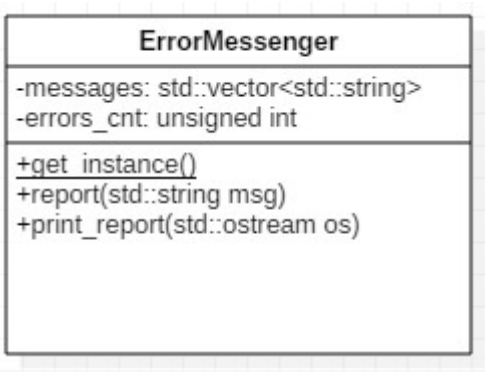
ErrorMessenger is a class implemented based on Singleton design pattern for collecting all fail tests information. Class belongs to *seq_test* namespace.

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*.