
Algorithms and Data Structures

Double Linked Ring

Documentation



Author: Kacper Wojakowski

kwojakow

293064

TABLE OF CONTENTS

1. General information	2
1.1. Overview of the <i>DoubleLinkedRing</i> template	2
1.2. Template parameters.....	2
1.3. Member types	2
1.4. Overview of the methods	3
 2. Iterator class	 4
2.1. Overview.....	4
2.2. Available operators	4
2.3. Access to elements.....	4
 3. Method details	 5
3.1. Standard methods	5
3.2. Operators.....	6
3.3. General methods	7
3.4. Printing	8
3.5. Iterator related methods.....	9
3.6. Insertion and removal	9
 4. Produce method	 11
4.1. Overview.....	11
4.2. Details of implementation.....	11

1

GENERAL INFORMATION

1.1 Overview of the *DoubleLinkedListRing* template.....

The C++11 standard is required to properly use the *DoubleLinkedListRing* class, as it uses some of its features (such as *auto* and *nullptr*). The class was written and compiled using Visual Studio 2017, and tested on the *lab011* server.

DoubleLinkedListRing (or *Ring* for short) is a class template, implemented as a double linked ring, abstract data structure. The elements of the list, *Nodes*, store two values: a *Key*, by which the *Nodes* are recognized and *Info*, the information stored in the *Nodes*. *DoubleLinkedListRing* allows multiple occurrences of the same *Key*. The class *DoubleLinkedListRing* supports multiple ways for inserting, removing and accessing its *Nodes*. The class includes an *Iterator*. The code to the class template and function declarations is stored in the *DoubleLinkedListRing.h* header file.

The *produce* method is defined and declared in a separate file, *produce.h* which has to be included in order to call the function.

1.2 Template parameters.....

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

Key	Typename of key, by which the <i>Nodes</i> are being differentiated
Info	Typename of data that are stored in <i>Nodes</i>

1.3 Member types.....

Private member types:

struct Node	<i>Node</i> is a structure containing a <i>Key</i> value, an <i>Info</i> value and a pointer to the next and previous <i>Node</i>
Node *any	Pointer to a <i>Node</i> in the <i>Double Linked Ring</i>

Public member types:

class Iterator	The iterator class, which allows access to elements in the ring and changing them
typedef const Iterator ConstIterator;	A define of constant iterator, which can only use the constant methods of the <i>Iterator</i> class

1.4 Overview of the methods

Standard methods:

<code>DoubleLinkedListRing()</code>
<code>DoubleLinkedListRing(const DoubleLinkedListRing<Key, Info> &source);</code>
<code>~DoubleLinkedListRing();</code>

Available operators:

<code>DoubleLinkedListRing<Key, Info>& operator=(const DoubleLinkedListRing<Key, Info> &rhs);</code>
<code>bool operator==(DoubleLinkedListRing<Key, Info> &rhs) const;</code>
<code>bool operator!=(DoubleLinkedListRing<Key, Info> &rhs) const;</code>
<code>friend std::ostream& operator<<(std::ostream &os, const DoubleLinkedListRing<Key, Info> &ring)</code>

General methods:

<code>void clear();</code>
<code>int size() const;</code>
<code>bool isEmpty() const;</code>
<code>bool search(const Key &key) const;</code>

Printing:

<code>void print(std::ostream &os = std::cout) const;</code>

Iterator related methods:

<code>Iterator begin() const;</code>
<code>Iterator find(const Key &key, int occurrence = 1) const;</code>

Insertion and removal:

<code>void insertAfter(const Key &newKey, const Info &newInfo, const Iterator &location);</code>
<code>void insertAfter(const Key &newKey, const Info &newInfo, const Key &location, int occurrence = 1);</code>
<code>void insertBefore(const Key &newKey, const Info &newInfo, const Iterator &location);</code>
<code>void insertBefore(const Key &newKey, const Info &newInfo, const Key &location, int occurrence = 1);</code>
<code>void remove(const Iterator &location);</code>
<code>void remove(const Key &location, int occurrence = 1);</code>

2

ITERATOR CLASS

2.1 Overview

Iterator is a public inside class of *DoubleLinkedListRing* which can be used to traverse the list and access its elements. It can also be passed and returned from some *DoubleLinkedListRing* methods.

Private members of the class:

<code>mutable Node *current;</code>	Pointer to the current <i>Node</i> of the iterator
-----------------------------------------	----------------------------------------------------

Public members of the class:

<code>struct Content</code>	A structure used for holding the elements of the <i>Iterator</i> , further explained in 2.3
-----------------------------	---------------------------------------------------------------------------------------------

The class *ConstIterator*, which can only access the const methods of *Iterator* is declared as: `typedef const Iterator ConstIterator;`

2.2 Available operators.....

<code>Iterator& operator=(const Iterator &rhs)</code>	Assigning value of existing <i>Iterator</i>
<code>Iterator& operator++() const Iterator operator++(int) const Iterator& operator--() const Iterator operator--(int) const Iterator operator+(int rhs) const Iterator operator-(int rhs) const</code>	Moving the <i>Iterator</i> forwards or backwards
<code>bool operator==(const Iterator &rhs) const bool operator!=(const Iterator &rhs) const</code>	Comparing two <i>Iterators</i>
<code>Content operator*() const Content operator*() const Content* operator->() const Content* operator->() const</code>	Access to elements of <i>Iterator</i> , further explained in 2.3

2.3 Access to elements

Iterator uses a nested public structure *Content*, which is implemented to be able to access both *Key* and *Info* with one operator without risk of accessing pointers *next* and *previous* of *Nodes*.

```

struct Content
{
    Key& key;
    Info& info;
};

```

Content stores references to *Key* and *Info*, which allows *Iterator* to change the data inside the *DoubleLinkedRing*. *ConstIterator*, however, is not able to do that, it can only see the data.

To access data without declaring a *Content* object two methods are also available: *getInfo* and *getKey*.

All the ways to access the data:

<pre> Info& getInfo() const Info& getInfo() const Key& getKey() const Key& getKey() const </pre>	Returns reference or constant reference to <i>Key</i> and <i>Info</i> , respectively
<pre> Content operator*() const Content operator*() const </pre>	Returns struct <i>Content</i> (see above)
<pre> Content* operator->() const Content* operator->() const </pre>	Returns pointer to <i>Content</i> (see above)

3

METHOD DETAILS

3.1 Standard methods.....

Default constructor	
DoubleLinkedRing()	
Parameters:	-
Returns:	-
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	Assigns <i>nullptr</i> to <i>head</i>

Copy constructor	
<code>DoubleLinkedRing(const DoubleLinkedRing<Key, Info> &source);</code>	
Parameters:	<i>source</i> – constant reference to <i>DoubleLinkedRing</i> to be copied from
Returns:	-
Complexity:	Linear O(n)
Exceptions:	May throw <i>std::bad_alloc</i>
Notes:	Calls <i>operator=</i> (see 3.2)

Destructor	
<code>~DoubleLinkedRing();</code>	
Parameters:	-
Returns:	-
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Calls the <i>clear</i> method (see 3.3)

3.2 Operators.....

Assignment operator =	
<code>DoubleLinkedRing<Key, Info>& operator=(const DoubleLinkedRing<Key, Info> &rhs);</code>	
Parameters:	<i>rhs</i> – constant reference to a <i>DoubleLinkedRing</i> to be assigned
Returns:	Copy of <i>rhs</i>
Complexity:	Linear O(n)
Exceptions:	May throw <i>std::bad_alloc</i>
Notes:	Clears the ring and copies the elements from <i>rhs</i>

Comparison operator ==	
<code>bool operator==(DoubleLinkedRing<Key, Info> &rhs) const;</code>	
Parameters:	<i>rhs</i> – constant reference to a <i>DoubleLinkedRing</i> to compare to
Returns:	<i>true</i> if both <i>Sequences</i> are identical, <i>false</i> otherwise
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	<i>operator==</i> must be defined for both <i>Key</i> and <i>Info</i>

Comparison operator !=	
<code>bool operator!=(DoubleLinkedRing<Key, Info> &rhs) const;</code>	
Parameters:	<i>rhs</i> – constant reference to a <i>DoubleLinkedRing</i> to compare to
Returns:	<i>true</i> if both <i>Ring</i> are not identical, <i>false</i> otherwise
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Calls <i>operator==</i> , <i>operator==</i> must be defined for both <i>Key</i> and <i>Info</i>

Output operator<<	
<code>friend std::ostream& operator<<(std::ostream &os, const DoubleLinkedRing<Key, Info> &ring)</code>	
Parameters:	A reference to an <i>std::ostream</i> object, a <i>Ring</i> to be printed
Returns:	Reference to the <i>std::ostream</i> that was passed
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	<i>operator<<</i> must be defined for <i>Key</i> and <i>Info</i>

3.3 General methods.....

clear	
<code>void clear();</code>	
Parameters:	-
Returns:	-
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Deletes every single element from the ring

size	
<code>int size() const;</code>	
Parameters:	-
Returns:	Number of elements in the <i>Ring</i>
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	-

isEmpty	
<code>bool isEmpty() const;</code>	
Parameters:	-
Returns:	<i>true</i> if the <i>DoubleLinkedRing</i> is empty, <i>false</i> otherwise
Complexity:	Constant $O(1)$
Exceptions:	Exception safe
Notes:	Checks whether <i>any</i> is <i>nullptr</i>

search	
<code>bool search(const Key &key) const;</code>	
Parameters:	<i>key</i> – constant reference to the <i>Key</i> to search for
Returns:	<i>true</i> if <i>key</i> is found, <i>false</i> otherwise
Complexity:	Linear $O(n)$
Exceptions:	Exception safe
Notes:	Searches for the first occurrence of <i>key</i>

3.4 Printing

print	
<code>void print(std::ostream &os = std::cout) const;</code>	
Parameters:	A reference to an <i>std::ostream</i> object, defaults to <i>std::cout</i>
Returns:	-
Complexity:	Linear $O(n)$
Exceptions:	Exception safe
Notes:	operator<< must be defined for <i>Key</i> and <i>Info</i>

3.5 Iterator related methods.....

begin	
<code>Iterator begin() const;</code>	
Parameters:	-
Returns:	Iterator pointing to <i>any</i>
Complexity:	Constant $O(1)$
Exceptions:	Exception safe
Notes:	If the <i>Ring</i> is empty, the returned <i>Iterator</i> will point to <i>nullptr</i>

find	
<code>Iterator find(const Key &key, int occurrence = 1) const;</code>	
Parameters:	<i>key</i> – constant reference to the <i>Key</i> to search for <i>occurrence</i> – specified occurrence of the <i>key</i> in the <i>Ring</i>
Returns:	Iterator pointing to found element
Complexity:	Linear $O(n)$
Exceptions:	Exception safe
Notes:	If the element is not found, it will return an <i>Iterator</i> to <i>nullptr</i>

3.6 Insertion and removal.....

insertAfter	
<code>void insertAfter(const Key &newKey, const Info &newInfo, const Iterator &location);</code>	
Parameters:	<i>newKey</i> – constant reference to the <i>Key</i> of the new <i>Node</i> <i>newInfo</i> – constant reference to the <i>Info</i> of the new <i>Node</i> <i>location</i> – constant reference to the <i>Iterator</i> after which to insert a new <i>Node</i>
Returns:	-
Complexity:	Constant $O(1)$
Exceptions:	May throw <code>std::bad_alloc</code>
Notes:	Inserts a new <i>Node</i> at a specified place

insertAfter	
<pre>void insertAfter(const Key &newKey, const Info &newInfo, const Key &location, int occurrence = 1);</pre>	
Parameters:	<i>newKey</i> – constant reference to the <i>Key</i> of the new <i>Node</i> <i>newInfo</i> – constant reference to the <i>Info</i> of the new <i>Node</i> <i>location</i> – constant reference to the <i>Key</i> marking after which <i>Node</i> to insert a new one <i>occurrence</i> – integer denoting after which occurrence of <i>location</i> to insert the <i>Node</i> (defaults to 1)
Returns:	-
Complexity:	Linear O(n)
Exceptions:	May throw <i>std::bad_alloc</i>
Notes:	Inserts a new <i>Node</i> at a specified place

insertBefore	
<pre>void insertBefore(const Key &newKey, const Info &newInfo, const Iterator &location);</pre>	
Parameters:	<i>newKey</i> – constant reference to the <i>Key</i> of the new <i>Node</i> <i>newInfo</i> – constant reference to the <i>Info</i> of the new <i>Node</i> <i>location</i> – constant reference to the <i>Iterator</i> before which to insert a new <i>Node</i>
Returns:	-
Complexity:	Constant O(1)
Exceptions:	May throw <i>std::bad_alloc</i>
Notes:	Inserts a new <i>Node</i> at a specified place

insertBefore	
<pre>void insertAfter(const Key &newKey, const Info &newInfo, const Key &location, int occurrence = 1);</pre>	
Parameters:	<i>newKey</i> – constant reference to the <i>Key</i> of the new <i>Node</i> <i>newInfo</i> – constant reference to the <i>Info</i> of the new <i>Node</i> <i>location</i> – constant reference to the <i>Key</i> marking before which <i>Node</i> to insert a new one <i>occurrence</i> – integer denoting after which occurrence of <i>location</i> to insert the <i>Node</i> (defaults to 1)
Returns:	-
Complexity:	Linear O(n)
Exceptions:	May throw <i>std::bad_alloc</i>
Notes:	Inserts a new <i>Node</i> at a specified place

remove	
<code>void remove(const Iterator &location);</code>	
Parameters:	<i>location</i> – constant reference to <i>Iterator</i> marking which <i>Node</i> to delete
Returns:	-
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	Removes a specified <i>Node</i>

remove	
<code>void remove(const Key &location, int occurrence = 1);</code>	
Parameters:	<i>location</i> – constant reference to the <i>Key</i> marking which <i>Node</i> to delete <i>occurrence</i> – integer denoting which occurrence of <i>location</i> to delete (defaults to 1)
Returns:	-
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Removes a specified <i>Node</i>

4. PRODUCE METHOD

4.1 Overview

```
template <typename Key, typename Info>
DoubleLinkedList<Key, Info> produce( const DoubleLinkedList<Key, Info> &ring1,
                                     int start1, int step1, bool dir1,
                                     const DoubleLinkedList<Key, Info> &ring2,
                                     int start2, int step2, bool dir2,
                                     int num, bool dir )
```

The function “produces” a *DoubleLinkedList* from two *Rings* R1 and R2. If any of the arguments are incorrect ($num < 1$, for example) the function returns an empty *Ring*.

4.2 Details of implementation

The main part of the function is based on *Iterators* (see Chapter 2).

When copying from input *Rings* the function copies both *Key* and *Info* of each node.