Algorithms and Data Structures

Double Linked Ring

Report

Author:Sotaro Suzuki

TABLE OF CONTENTS

1 GENERAL INFORMATION

1.1 Overview of the *DoubleLinkedRing* template .....

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

DoubleLinkedRing (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*. *DoubleLinkedRing* allows multiple occurrences of the same *Key*. The class *DoubleLinkedRing* supports multiple ways for inserting, removing and accessing its *Nodes*. The class includes and *Iterator.* The code to the class template and function declarations is stored in the *DoubleLinkedRing.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 DoubleLinkedRing
```

`Key`	Typename of key, by which the *Nodes* are being differentiated
`Info`	Typename of data that are stored in *Nodes*

1.3 Member types .....

Private member types:

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

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 *Iterator* class

1.4 Overview of the methods .....

Standard methods:

```
DoubleLinkedRing()
DoubleLinkedRing( const DoubleLinkedRing<Key, Info> &source );
~DoubleLinkedRing();
```

Available operators:

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

General methods:

```
void clear();
int size() const;
bool isEmpty() const;
bool search( const Key &key ) const;
```

Printing:

```
void print( std::ostream &os = std::cout ) const;
```

Iterator related methods:

```
Iterator begin() const;
Iterator find( const Key &key, int occurence = 1 ) const;
```

Insertion and removal:

`void insertAfter( const Key &newKey, const Info &newInfo, const Iterator &location );`
`void insertAfter( const Key &newKey, const Info &newInfo, const Key &location, int occurence = 1 );`
`void insertBefore( const Key &newKey, const Info &newInfo, const Iterator &location );`
`void insertBefore( const Key &newKey, const Info &newInfo, const Key &location, int occurence = 1 );`
`void remove( const Iterator &location );`
`void remove( const Key &location, int occurence = 1 );`

2 ITERATOR CLASS

2.1 Overview .....

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

Private members of the class:

`mutable Node *current;`	Pointer to the current *Node* of the iterator

Public members of the class:

`struct Content`	A structure used for holding the elements of the *Iterator*, further explained in *2.3*

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

2.2 Available operators .....

`Iterator& operator=( const Iterator &rhs )`	Assigning value of existing *Iterator*

`Iterator& operator++() const`
`Iterator operator++( int ) const`
`Iterator& operator--() const`
`Iterator operator--( int ) const`
`Iterator operator+( int rhs ) const`
`Iterator operator-( int rhs ) const`
	Moving the *Iterator* forwards or backwards

`bool operator==( const Iterator &rhs ) const` `bool operator!=( const Iterator &rhs ) const`	Comparing two *Iterators*

`Content operator*()`
`const Content operator*() const Content*`
`operator->()`
`const Content* operator->() const`
	Access to elements of *Iterator*, 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:

`Info& getInfo()`
`const Info& getInfo() const Key&`
`getKey()`
`const Key& getKey() const`
	Returns reference or constant reference to *Key* and *Info*, respectively

`Content operator*()` `const Content operator*() const`	Returns struct *Content* (see above)
`Content* operator->()` `const Content* operator->() const`	Returns pointer to *Content* (see above)

3 METHOD DETAILS

3.1 Standard methods .....

Default constructor	
`DoubleLinkedRing()`	
Parameters:	-
Returns:	-
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	Assigns *nullptr* to *head*

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

Destructor	
`~DoubleLinkedRing();`	
Parameters:	-
Returns:	-
Complexity:	Linear O(n)

Exceptions:	Exception safe
Notes:	Calls the *clear* method (see 3.3)

3.2 Operators .....

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

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

Notes:	Calls *operator==,* operator== must be defined for both *Key* and *Info*

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

3.3 General methods .....

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

size	
`int size() const;`	
Parameters:	-
Returns:	Number of elements in the *Ring*
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	-
isEmpty	

`bool isEmpty() const;`	
Parameters:	-
Returns:	*true* if the *DoubleLinkedRing* is empty, *false* otherwise
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	Checks whether *any* is *nullptr*

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

3.4 Printing .....

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

3.5 Iterator related methods .....

begin	
`Iterator begin() const;`	
Parameters:	-
Returns:	Iterator pointing to *any*
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	If the *Ring* is empty, the returned *Iterator* will point to *nullptr*

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

3.6 Insertion and removal .....

insertAfter
`void insertAfter( const Key &newKey, const Info &newInfo, const Iterator &location );`

Parameters:	*newKey* – constant reference to the *Key* of the new *Node newInfo* – constant reference to the *Info* of the new *Node* *location* – constant reference to the *Iterator* after which to insert a new *Node*
Returns:	-
Complexity:	Constant O(1)
Exceptions:	May throw *std::bad_alloc*
Notes:	Inserts a new *Node* at a specified place

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

insertBefore	
`void insertBefore( const Key &newKey, const Info &newInfo, const Iterator &location );`	
Parameters:	*newKey* – constant reference to the *Key* of the new *Node newInfo* – constant reference to the *Info* of the new *Node* *location* – constant reference to the *Iterator* before which to insert a new *Node*
Returns:	-

Complexity:	Constant O(1)
Exceptions:	May throw *std::bad_alloc*
Notes:	Inserts a new *Node* at a specified place

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

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

remove	
`void remove( const Key &location, int occurence = 1 );`	
Parameters:	*location* – constant reference to the *Key* marking which *Node* to delete *occurrence* – integer denoting which occurrence of *location* to delete (defaults to 1)

Returns:	-
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Removes a specified *Node*

4. PRODUCE METHOD

4.1 Overview .....

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

The function “produces” a *DoubleLinkedRing* 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.