Doubly linked circular list - an EADS project

Maciej Marcinkiewicz

January 2020

1 Introduction

The goal of the project is to implement doubly linked circular list (or as some call it – ring) in the C++ programming language with use of templates. As in the previous project the requirement is to create a container with two generic parameters – `Key` and `Info`. One more requirement was to use iterators, so `Ring` has to contain `iterator` and `const_iterator` classes.

The second task is to create `Split function` which produces two rings out of one. This function is meant to be external (and cannot be linked with keyword `friend`).

2 Overview of funtions

2.1 Constructors and destructor

```
Ring()
```

Default constructor.

```
~Ring()
```

Destructor.

```
Ring(const Ring<Key, Info> &other)
```

Copy constructor.

```
Data(const Key &key, const Info &info)
```

Constructor for `Data`. Creates new `Data` object with given key and information.

```
Node(const Key &key, const Info &info, Node *next, Node *previous)
     : data(key, info)
```

Constructor for `Node`. Assigns node's pointers.

```
iterator()
const_iterator()
```

Constructors for iterator classes.

2.2 Overloaded operators

```
Ring<Key, Info> &operator=(const Ring<Key, Info> &other)
```

Assignment operator.

```
bool operator==(const Ring<Key, Info> &other)
```

Comparison operator. Returns `true` if compared rings contain the same elements.

```
bool operator!=(const Ring<Key, Info> &other)
```

Comparison operator. Returns `false` if compared rings contain the same elements.

```
bool iterator::operator==(const Ring<Key, Info> &other)
bool const_iterator::operator==(const Ring<Key, Info> &other)
```

Comparison operators. Return `true` if compared nodes of iterators are the same.

```
bool iterator::operator!=(const Ring<Key, Info> &other)
bool const_iterator::operator!=(const Ring<Key, Info> &other)
```

Comparison operators. Return `true` if compared nodes of iterators are the same.

```
Data &iterator::operator*() const
Data &const_iterator::operator*() const
Data *iterator::operator->() const
Data *const_iterator::operator->() const
```

Dereference operators for iterators.

```
iterator &iterator::operator++()
const_iterator &const_iterator::operator++()
iterator iterator::operator++(int)
const_iterator const_iterator::operator++(int)
```

Preincrementation and postincrementation operators for iterators.

```
iterator &iterator::operator--()
const_iterator &const_iterator::operator--()
iterator iterator::operator--(int)
const_iterator const_iterator::operator--(int)
```

Predecrementation and postdecrementation operators for iterators.

2.3 Adding elements

```
void PushBack(const Key &key, const Info &info)
```

Method which adds new node of given key and information behind `first`.

```
bool Insert(const Key &key, const Info &info,
            const iterator &position)
```

Method which adds new node of given key and information before `position`. Returns `false` if `position` is `end()`.

2.4 Removing elements

```
bool PopBack()
```

Method which removes node behind `first`. Returns `false` if list is empty.

```
bool Remove(const iterator &position)
```

Method which removes node which is pointed by `position`. Returns `false` if list is empty or if `position` is `end()`.

```
bool Clear()
```

Removes all nodes. Returns `false` if list is empty.

```
bool RemoveAllOccurances(const Key &key)
```

Method which removes all nodes with `key`. Returns `false` if list is empty or there are no nodes with such a key.

2.5 Other methods

```
iterator Find(const Key &key, int whichOccurance = 1)
const_iterator Find(const Key &key, int whichOccurance = 1)
```

Returns iterator pointing on element with `key`. `whichOccurance` tells which occurance of key has to be found. If there is no such a node returns `end()`.

```
bool IsEmpty() const
```

Returns `true` if list contains no nodes.

```
int Size() const
```

Returns private field `size` which contains number of nodes in list.

```
int NumberOfOccurances(const Key &key) const
```

Returns number of occurances of nodes with given key.

2.6 Iterator methods

```
iterator begin()
const_iterator begin() const
```

Return iterator pointing on `first`.

```
iterator end()
const_iterator end() const
```

Return iterator pointing on `nullptr`.

2.7 Split function

```
void Split(const Ring<Key, Info> &source, bool direction,
           Ring<Key, Info> &result1, int sequence1, int rep1,
           Ring<Key, Info> &result2, int sequence2, int rep2)
```

Produces two rings from another one. `source` is ring from other are produced, `direction` tells if passage should be forward (`true`) or backward (`false`), `result1` and `result2` are rings in which produced ones are stored, `sequence1`and `sequence2` tell how many nodes are taken in which passage and eventually `rep1` and `rep2` tell how many passages have to be taken. If result1 or result2 are not empty, they are cleared.

Example

```
Input:
//info, keys for every node are the same
source = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

direction = false

result1 = r1
result2 = r2

sequence1 = 3
sequence2 = 2

rep1 = 2
rep2 = 4

Output:
//only info as keys are the same
r1 = {1, 12, 11, 8, 7, 6}
r2 = {10, 9, 5, 4, 3, 2, 1, 12}
```

3 Decisions and changes

Based on experience with previous project I decided to create two structs for keeping elements of list. One is `Data` which is public struct contining `Key` and `Info`. The second is private `Node` which contains `Data`, `next` and `previous`. This makes encapsulations unbroken while keeping the implementation unrevealed.

On the very beginning class contained `Print` method. Unfortunately, someone could make a list of non-printable objects and this could end with serious error. To make testing possible I implemented this function in testing source files.

4 Testing

4.1 Introduction

Once again I decided to use Catch2 testing framework. Tests are performed in two files. The first one contains all methods testing, the second one `Split` function testing. For the purpose of memory leaks checking I used Valgrind software.

4.2 Structure of tests

Empty list

- Printing list
- Checking if list is empty
- Using removal methods
- Using `Find` method

Filling list

- Adding nodes behind `first`
- Adding nodes inside the list

Removing nodes from list

- Removing nodes behind `first`
- Removing selected nodes
- Removing all nodes
- Removing all nodes with chosen key

Comparison operators

- Equality operator
- Inequality operator

Copying lists

- Copying with copy constructor
- Copying with assignment operator

Other methods

- Size method
- Find method
- Checking number of occurances

Split function

- Splitting empty list
- Splitting in normal cases
- Splitting with wrong parameters