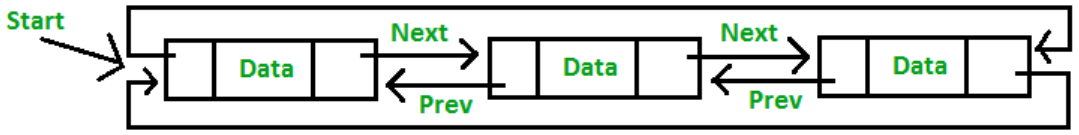
# EADS-Lab 2

# Ring Doubly-linked list




Student: Xián García Nogueira K-6077
Lecturer: Turlej Dariusz

26-04-2021

# Index:

**Index:** 1

**Description:** 2

**Aim of the Lab:** 2

**Methods:** 3

**Methods From the Iterators** 6

**Testing:** 8

Constructors tests: 8

Output: 8

Delete and Search tests: 9

Output: 9

Insert after tests: 9

Output: 9

Insert before tests: 10

Output: 10

Merge by position tests: 10

Output: 10

Merge by key tests: 11

Output: 11

Merge by key and position mixxed test: 11

Output: 11

# Description:

This project consists of a circular doubly linked list implemented in c++ for structure data with a key and the information for each node of the structure.
In the source files are the **Ring.ccp** and **Ring.h** that are full Ring doubly linked list implemented, they must be included in the in the desired code with:
**“#include "Ring.cpp"**
and the file must be in the project folder.
Then in the main.cpp are the tests ,one function for testing and the 2 externals functions merge_key and merge_pos

For initialization a new Circular Doubly linked list this is the code:

**Ring<Key, Info> name;**

# Aim of the Lab:

The aim of the Lab is to implement a full Circle DoublyLinkedList with Iterators, one constant and one static. With 2 external functions which one merges 2 lists with the starting position and other with the starting key with the parameter of the direction. Then testing all the methods.

# Methods:

**Ring();**
Constructor for the list

**~Ring();**
Deconstructor for the list

**Ring (const Ring<Key, Info>&);**
Copy constructor, that create a Ring initializing it with a deep copy of another Ring

**Ring<Key, Info>& operator= (const Ring<Key, Info>&copy);**
Assign a copy of a Ring to an existing Ring.

**Ring<Key, Info>& operator++ ();**
iterate "clockwise" once

**Ring<Key, Info>& operator++ (int);**
iterate "clockwise" as many times as entered by parameters

**Ring<Key, Info>& operator+= (const Ring<Key, Info>&newRing);**
Add to the current Ring the Ring given by parameters

**Ring<Key, Info>& operator-- ();**
iterate "counterclockwise" once

**Ring<Key, Info>& operator-- (int);**
iterate "counterclockwise" as many times as entered by parameters

**Ring<Key, Info>& operator-= (const Ring<Key, Info>&subRing);**
Subtract to the current Ring the Ring given by parameters

**friend Ring<key, info> operator+ (const Ring<Key, Info>&newRing,const Ring<Key, Info>&newRing2);**
Add to the first Ring given by parameters with the second, this constructor uses the constructor “operator+=”

**friend Ring<key, info> operator- (const Ring<Key, Info>&subRing,const Ring<Key, Info>&subRing2);**
Subtract to the first Ring given by parameters with the second.

**bool operator==(const Ring<Key,Info> &other);**
Comparison operator
**bool operator!=(const Ring<Key,Info> &other);**
Inverse comparison operator

**bool insertAtBeg (const Key& newKey, const Info& newInfo,bool direct);**
Insert a new Node with the values by parameters at the beginning of the list
return true if all go good and false if there is an error on the request.if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertAtEnd (const Key& newKey, const Info& newInfo,bool direct);**
Insert a new Node with the values by parameters at the end of the list
return true if all go good and false if there is an error on the request.if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertAfter (const Key& newKey, const Info& newInfo, const Key& where, int occurance,bool direct);**
Insert new node to the list after the key given by parameter and in which occurrence it should be in case the keys are repeated, if the keys are not repeated just occurance=1
return true if all go good and false if there is an error on the request. if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertBefore (const Key& newKey, const Info& newInfo, const Key& where, int occurance,bool direct);**
Insert new node to the list before the key given by parameter and in which occurrence it should be in case the keys are repeated, if the keys are not repeated just occurance=1
return true if all go good and false if there is an error on the request. if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**int size()const ;**
Returns the size of the current Ring.

**int search (const Key& key)const;**
This search method returns the number of occurrences from the given key in the ring, and returns 0 if it does not have that key.

**bool remove (const Key& key, int occurance, bool direct);**
Delete the node with the key given by parameter and in which occurrence it should be in case the keys are repeated, if the keys are not repeated just occurrence=1
return true if all go good and false is there is an error on the request. if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**void print(bool direct);**
Print in the console the info from all the nodes in the current Ring

**void print2(bool direct);**
Print in the console all the nodes in the current Ring in a format of "[key,info]"

**void reverse();**
Reverse the Ring.

**bool isEmpty() const;**
Returns true if the current Ring is empty or false if not

**bool search2(const Key& key, int occurance);**
Search if exists in the node given by parameters in the actual list.
Return false if not, true if yes.

**Iterator search3 (const Key& where, int occurance);**
Search for the given key and return the Iterator that contains it.

**bool insertAfter (const Key& newKey, const Info& newInfo, Iterator where);**
Insert new node to the list after position given by the Iterator in the parameter and in which occurrence it should be in case the keys are repeated, if the keys are not repeated just occurance=1
return true if all go good and false if there is an error on the request. if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertBefore (const Key& newKey, const Info& newInfo, Iterator where);**
Insert new node to the list before position given by the Iterator in the parameter and in which occurrence it should be in case the keys are repeated, if the keys are not repeated just occurance=1
return true if all go good and false if there is an error on the request. if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertAtBeg (Iterator what, bool direct);**
Insert a new Node with the values taken from the Iterator given by parameters at the beginning of the list
return true if all go good and false if there is an error on the request.if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**bool insertAtEnd (Iterator what, bool direct);**
Insert a new Node with the values taken from the Iterator given by parameters at the end of the list
return true if all go good and false if there is an error on the request.if the direct is true, it will iterate from head to tail, and if it is false from tail to head.

**Iterator begin();**
Returns the Iterator starting in the head from the ring.

**Iterator end();**
Returns the Iterator starting in the tail from the ring.

**const_Iterator beginConst() const;**
Returns the Constant_ Iterator starting in the head from the ring.

**const_Iterator endConst() const;**
Returns the Constant_Iterator starting in the tail from the ring.

# Methods From the Iterators

This methods are in the both constructor, that changes the visibility from the attributes and the methods, but the functionality of the methods are the same:

**const_Iterator ();**
Constructor from the iterator.

**const Info& operator*();**
Operator that returns the info.

**const const_Iterator & operator++();**
iterate "clockwise" once.

**const const_Iterator operator++(int);**
iterate "clockwise" as many times as entered by parameters

**const const_Iterator& operator--();**
iterate "counterclockwise" once

**const const_Iterator& operator--(int);**
iterate "counterclockwise" as many times as entered by parameters

**const const_Iterator operator+(int);**
iterate "clockwise" as many times as entered by parameters

**const Key& getKey();**
iGet the key from the actual position of the iterator.

**const Info& getInfo();**
Get the info from the actual position of the iterator.

**bool setKey(const Key&);**
Change the key value in the actual position of the iterator.

**bool setInfo(const Info&);**
Change the info value in the actual position of the iterator.

**bool isEmpty();**
Returns true if the iterator is empty

**bool operator==(const Iterator &);**
Comparison operator

**bool operator!=(const Iterator &);**
Inverse comparison operator

**const_Iterator (const Ring<Key, Info>&newIt );**
Set the iterator for the given ring

**const_Iterator associeteWith (const Ring<Key, Info>&);**
Associate the iterator with the given ring

**friend Ring<Key, Info>;**

# Testing:

The testing methods are implemented in the main.cpp, It consist of a method that i create “AssertEqueals” that returns true if the data of the Ring are the same as the string given by me, for test that all is correct.
Then I created different methods that test the functions needed for this lab.

### Constructors tests:

Test the constructors and operators, this is the output that checks that all tested are correct.

**Output:**

```
--------Constructors---------
Test 0.1: true
[3,3][6,7][2,5]
Test 0.2(empty copyconstructor): true
Test 0.3: true
[6,7][2,5]
Test 0.4(+=operator, empty first List): true
[6,7][2,5]
Test 0.5(+=operator): true
[6,7][2,5][6,7][2,5]
Test 0.6(+=operator, empty second List): true
[6,7][2,5][6,7][2,5]
Test 0.7(+operator, empty second List): true
[6,7][2,5][6,7][2,5]
Test 0.8(+operator): true
[6,7][2,5][6,7][2,5][6,7][2,5][6,7][2,5]
Test 0.9(-=operator, empty first List): true
[]
Test 0.10(-=operator, empty second List): true
[2,2][1,1]
Test 0.11(-=operator): true
[2,2]
Test 0.12(-operator, empty first List): true
[]
Test 0.13(-operator, empty second List): true
[5,5]
Test 0.14(-operator): true
[1,1][2,5][3,3][4,5]
```

## Delete and Search tests:

Test the Delete method, in both directions and all cases, this is the output that checks that all tests are correct.

**Output:**

```
Test 3.1: true
Test 3.2: true
[]
[1,2][7,10][7,45][7,9][6,7][2,5]
Test 3.3: true
[7,10][7,45][7,9][6,7][2,5]
Test 3.4: true
[7,10][7,45][7,9][6,7]
Test 3.5: true
[7,10][7,9][6,7]
Test 3.7: true
Test 3.9: true
Test 3.10: true
Test 3.11: true
Test 3.12: true
[]
[1,2][7,10][7,45][7,9][6,7][2,5]
Test 3.13: true
[1,2][7,10][7,45][7,9][6,7]
Test 3.14: true
[7,10][7,45][7,9][6,7]
Test 3.15: true
[7,10][7,45][6,7]
Test 3.16: true
Test 3.17: true
Test 3.18: true
Test 3.19: true
```

## Insert after tests:

Test the insertAfter method, in both directions and all cases, this is the output that checks that all tests are correct.

**Output:**

```
--------INSERT AFTER---------
Test 4.1: true
Test 4.2: true
[6,7][8,10][2,5][8,9][1,1]
Test 4.3: true
[6,7][8,10][1,1][2,5][8,9][1,1]
Test 4.4: true
[6,7][8,10][1,1][2,5][8,9][1,1]
```

## Insert before tests:

Test the insertBefore method, in both directions and all cases, this is the output that checks that all tests are correct.

**Output:**

```
--------INSERT BEFORE---------
Test 5.1: true
Test 5.2: true
[8,10][6,7][1,1][8,9][2,5]
Test 5.3: true
[1,1][8,10][6,7][1,1][8,9][2,5]
Test 5.4: true
[1,1][8,10][6,7][1,1][8,9][2,5]
```

## Merge by position tests:

Test the merge_pos external function, in both directions and all cases, this is the output that checks that all tests are correct.

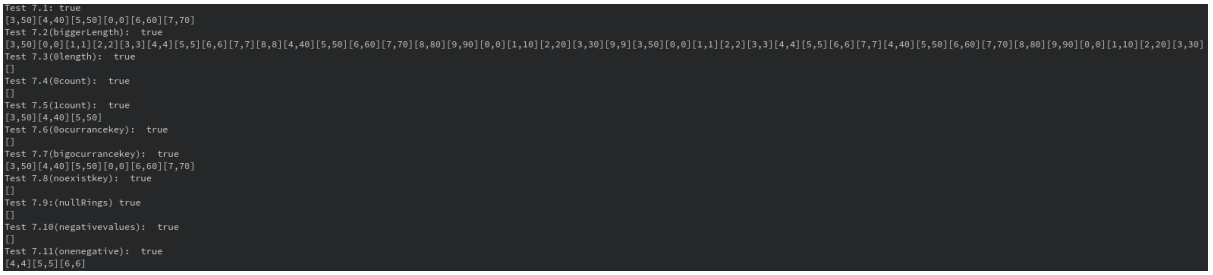
**Output:**

```
Test 6.2: true
[3,3][2,20][3,30][4,4][4,40][5,50][5,5][6,60][7,70][6,6][8,80][9,90]
Test 6.3(0count): true
[]
Test 6.4(bigcount): true
[3,3][2,20][3,30][4,4][4,40][5,50][5,5][6,60][7,70][6,6][8,80][9,90][7,7][0,0][1,10]
Test 6.5(0length): true
[]
Test 6.6(bigstartpos): true
[0,0][0,0][1,10][1,1][2,20][3,30][2,2][4,40][5,50][3,3][6,60][7,70][4,4][8,80][9,90]
Test 6.7:(0startpos) true
[0,0][0,0][1,10][1,1][2,20][3,30][2,2][4,40][5,50][3,3][6,60][7,70][4,4][8,80][9,90][5,5][0,0][1,10][6,6][2,20][3,30][7,7][4,40][5,50][8,8][6,60][7,70][9,9][8,80][9,90]
Test 6.8:(nullRings) true
[]
Test 6.9:(negatives) true
[]
Test 6.10(onenegative):  true
[4,4][5,5][6,6]
Test 6.11(bigstartpos): true
[3,3][2,20][3,30][4,4][4,40][5,50][5,5][6,60][7,70][6,6][8,80][9,90]
Test 6.12(examplegiven): true
[2,2][3,3][30,30][4,4][5,5][40,40][1,1][2,2][10,10][3,3][4,4][20,20]
Test 6.13(one false direction): true
[3,3][7,70][6,60][4,4][5,50][4,40][5,5][3,30][2,20][6,6][1,10][0,0]
Test 6.14(one false direction2): true
[6,6][2,20][3,30][5,5][4,40][5,50][4,4][6,60][7,70][3,3][8,80][9,90]
Test 6.15(two false direction): true
[6,6][7,70][6,60][5,5][5,50][4,40][4,4][3,30][2,20][3,3][1,10][0,0][2,2][9,90][8,80][1,1][7,70][6,60]
```

## Merge by key tests:

Test the merge_key external function, in both directions and all cases, this is the output that checks that all tests are correct.

### Output:




## Merge by key and position mixxed test:

Test the merge_key and merge_pos external functions, all checked together in the different cases.

### Output: