04.12.2020

EADS ring class documentation

Author: Sławomir Batruch
Student ID: 303827
Documentation version: 1
Date of documentation: 04.12.2020

Header file name: EADS_lab2.hpp
Testing program file name: EADS_lab2_test.cpp

Overview

This is the documentation of the ring data structure. Here is the list of the methods and the data structure of the class. For more information about the methods please review the header file. Comments present in the file are the foundation of the documentation provided below. Furthermore, some comments explaining certain parts and instructions used inside the methods are present in the header file

The data structure of the class (private part) is the following:

```
template<typename Key, typename Info>
class Ring { // Doubly linked ring
    private:
        struct Node {
            Key key;
            Info info;
            Node* next;
            Node* prev;
        };
        Node* head; /* is the start of the ring, i.e. the node from
which we create new nodes.
        the last added node will point to this. */
        void copy (const Ring<Key, Info>& srcRing); /* void type
because it modifies this. no need to return anything
        this method copies srcRing to "this". Used by deep copy
constructor */
```

Description: This is a typical structure of a doubly linked ring with two data fields – Key and Info. It contains the "head" pointer, which in actuality is "any" pointer, but for convenience

reason I have named it head to mark the first created node in the ring structure.
Furthermore, copy method is in the part. It is used in the deep copy constructor (as mentioned in the comment.
Please refer to the comments highlighted in blue for more information regarding the methods.

Then, the public part of the class, first and foremost, contains the Iterator class. The structure of it is shown below:

```
class Iterator {

        public:

            Iterator() : iter(nullptr) {} /* DEFAULT CONSTRUCTOR. It
just assigns nullptr to the iterator.

             Not much useful, because we need to assign the iterator
later on anyway.*/

            Iterator(const Iterator& cpyIt) : iter(cpyIt.iter) {} //
COPY CONTRUCTOR (DEEP COPY)

            Iterator& operator=(const Iterator& cpyIter) { //
SHALLOW COPY

                iter = cpyIter.iter;

                return *this;

            }


            ~Iterator() { // DESTRUCTOR

                free(iter);

            }


            Info& getInfo() const { // Imporant method. Allows us to
read the Info from the Node iterator is pointing to.

                return iter->info;

            }

            Key& getKey() const { // Imporant method. Allows us to
read the Key from the Node iterator is pointing to.

                return iter->key;

            }
            /*Node& operator*() const {
                return *iter;
            }*/

            bool operator==(const Iterator& cmpIter) { // Used to
compare if iterators are pointing to the same node
                return iter->prev == cmpIter.iter->prev &&
                iter->next == cmpIter.iter->next &&
                iter->key == cmpIter.getKey() &&
                iter->info == cmpIter.getInfo();


            }

            bool operator!=(const Iterator& cmpIter) { // Used to
compare if iterators are pointing to the same node
                return iter->prev != cmpIter.iter->prev ||
                iter->next != cmpIter.iter->next ||
                iter->key != cmpIter.getKey() ||
                iter->info != cmpIter.getInfo();
            }

            /*Iterator operator+(int incr) {
                Iterator newIter = this;
                for (unsigned int i = 0; i < incr; i++) {
                    newIter = newIter->next;
                }
                return newIter;
            }*/

            Iterator& operator++() { // Move the iterator one node
to the right
```

```
            iter = iter->next;

            return *this;

        }


        Iterator& operator+=(int incr) { // Move the iterator
incr nodes to the right

            for (unsigned int i = 0; i < incr; i++) {

                iter = iter->next;

            }

            return *this;

        }


        /*Iterator operator-(int decr) {

            Iterator newIter = this;

            for (unsigned int i = 0; i < decr; i++) {

                newIter = newIter->prev;

            }

            return newIter;

        }*/


        Iterator& operator--() { // Move the iterator one node
to the left

            iter = iter->prev;

            return *this;

        }

        Iterator operator-=(int decr) { // Move the iteratror by
the decr nodes to the left

            for (unsigned int i = 0; i < decr; i++) {

                iter = iter->prev;

            }

            return *this;
        }
        // void setKey(const Key& key);
        // void setInfo(const Info& info);
    private:
        friend class Ring<Key, Info>;
        Node* iter;
        Iterator(Node* cpyIter) : iter(cpyIter) {}
};
```

Note: grey code snippets is to be interpreted as obsolete code left over in case of some need to use it.

For the explanations please refer to the provided comments marked as blue in the above code extract from the header. Most of the methods are relatively trivial and the explanations written in comments should suffice. Below there is a list enumerating all available methods for the Iterator class:

```
Default constructor – Iterator(); Destructor - ~Iterator()

Copy constructor - Iterator(const Iterator& cpyIt),
Iterator(Node* cpyIter). First is a copy constructor from another
Iterator reference, second is a copy constructor from a Node.

Operators – operator= (shallow copy), operator==, operator!=,
operator++, operator+=, operator--, operator-=

Getters – getInfo(), getKey()
```

Below can be found ring public method list:

```
    Ring(); // DEFAULT CONSTRUCTOR

    ~Ring(); // DESTRUCTOR

    Ring(const Ring<Key, Info>&); // COPY CONSTRUCTOR (DEEP COPY)

    Ring<Key, Info>& operator=(const Ring<Key, Info>&); // SHALLOW
COPY

    void addNode(Key newKey, Info newInfo); // add a new node with
newKey as Key and newInfo as Info at the index position.

    bool removeNode(Key rmKey, Info rmInfo); // Remove a Node with
given Key and Info. Returns false if such node doesn't exist and
wasn't removed
```

```
    Key getKey(unsigned int index) const; /* Get a key at a given
index in the ring

    If ring has for example 5 elements and index 7 is passed, it
will choose the 2nd element*/

    Info getInfo(unsigned int index) const; // Same rules as with
getKey

    int seekKey(const Key& searchKey, unsigned int occurence) const;
/* Searches a occurence of a key and returns the index of the Node
at which the key is found

    this function returns -1 if the key was not found (when a whole
loop over the ring was done and the key wasnt found) */


    unsigned int size() const; // returns the size of the Ring


    Iterator begin() const {return head;} // Move the iterator to
the beginning of the ring

    Iterator end() const {return head->prev;} // Move the iterator
to the last Node before the starting one


    void display(); // used for testing by comparing the operation
done on the ring to its structure from memory

    void clear(); // delete all nodes. Used by the destructor
```

For this methods please fully refer to the provided comments highlighted in blue above. Here is the list of all methods that are shown above:

`Default constructor – Ring(); Destructor - ~Ring()`

`Copy constructor – Ring(const Ring<Key, Info>&)`

`Operators – operator= (shallow copy)`

`Mutators – addNode(Key newKey, Info newInfo), removeNode(Key rmKey, Info rmInfo)`

`Getters and other – getKey(unsigned int index), getInfo(unsigned int index), seekKey(const Key& searchKey, unsigned int occurrence), size(), display(), clear()`

`Iterator mutators – begin(), end()`

Extensive documentation (i.e. explanation of the methods) is present in the header file.

Produce methods

The produce methods works the same as they did in the Sequence class. This short extraction from the Sequence documentation and the comment block from the task should provide necessary information regarding the produce methods. For further information please refer to the header file and to the testing file:

```
template <typename Key, typename Info>
Ring <Key, Info> produce1 (
const Ring <Key, Info>& ring1, int start1, bool direct1, int len1,
const Ring <Key, Info>& ring2, int start2, bool direct2, int len2,
int limit)
```

```
template <typename Key, typename Info>
Ring <Key, Info> produce2 (
const Ring <Key, Info>& ring1, const Key& start_key1, int start1,
bool direct1, int len1,
const Ring <Key, Info>& ring2, const Key& start_key2, int start2,
bool direct2, int len2,
int limit)
```

From testing file:

```
  // produce1 is called in produce2, hence calling produce2 proves
                          that produce1 works properly.

        // all work regarding ring creation is done in produce1

                     // expected result (key) is:

   // {8}{9}{60}{70} {1}{2}{80}{90} {3}{4}{10}{20} {5}{6}{30}{40}
 // limit:  1              2              3              4
```

And from the header file, an example provided in the lab hours:

```
/*

ring1={1,}{2,}{3,}{4,}{5,}

ring2={10,}{20,}{30,}{40,}

start1=2, len1=2,

start2=3, len2=1,

limit=4


{2,}{3,}{30,} {4,}{5,}{40,} {1,}{2,}{10,} {3,}{4,}{20,}


*/
```

Produce2 works similar way to produce1, but it first needs to move the start index to a proper place (i.e. to a node with a given start_key, but one that is also after the indicated start argument)

Testing

For the testing, it was done through if condition checking, but also some tests were done by simply displaying the ring structure to the standard output. The user himself, in this case, must compare the shown ring structure in the standard output with the expected result. This is for example present in the produce testing.

All the tests in the time of writing this documentation are passed when run.

Compilation

For the header compilation and to run the testing console program, enter the directory the source files are present in and execute the following commands in the terminal. First command will compile the header, second will compile the test program, and the 3rd command will run the obtained compiled executable file:

```
g++ EADS_lab2.hpp
g++ EADS_lab2_testing.cpp -o test.exe
./test.exe
```