ALGORITHMS AND DATA STRUCTURES

PROJECT 2

Julia Czmut 300168

1. DESCRIPTION OF THE DECISIONS

1.1 CONCERNING FUNCTIONS AND CLASSES' INTERFACES

The program consists of the main template class **BiRing** and two smaller classes: **iterator** and **const_iterator**. There is also an external function **shuffle**.

BiRing is a bidirectional ring with two parameters: Key and Info. It has a private **Node** struct, as well as a **KeyInfoPair** struct which is public. Each Node has its KeyInfoPair and pointers to next and previous elements. I decided to create those two structs, because Node on its own has some information which could easily be public, like the Key and Info, but giving public access to the pointers would facilitate breaking the structure and losing the information it contains. That is where KeyInfoPair struct comes in, keeping Key and Info of the nodes.

As ring is a circular structure, it has no end. End of the ring is abstract, but is defined either way, as it is next to the anchor. It is also needed for the situation when the ring is empty: then end and anchor are equal. Anchor is the "head" of the ring. It could be any of the elements, but the structure has to start somewhere after all.
The class has two options for inserting an element: using a function **insert** which takes a KeyInfoPair and an iterator before which we want to place that element in the structure. The second option is function **push_back.** Its name may seem to lack sense, as the ring has no end, but it is just because the element is inserted before the anchor, so in some sense at the end.
The program allows to **erase** an element by providing the iterator which points to it and also to erase all occurrences of a given key using **eraseKey**.
Function **advance**, given a key and a number of occurrence, allows to advance the iterator to the desired occurrence of this key. It is helpful in a private function **internalFind**, which in turn is used by two public functions **find:** one returning a const_iterator and the other returning an iterator.
As for less important functionalities, there are also functions determining whether a given key exists, how many times and whether the ring is empty.
Lastly, in the class we can print the whole ring starting from the anchor or starting from a specified key.

Iterator and **const_iterator** are there to help traversing through the ring. Iterators are used in functions which modify some data. Const_iterators on the other hand are used in functions which do not need to modify anything, so to "read only". Their methods are just operators, because their main job is to hide pointers and as mentioned before, to move along the structure.

External template function **shuffle**, given references to two non-empty rings, number of elements to extract from the first one, from the second one and number of repetitions of this procedure, returns a shuffled ring. The idea can be presented by

$$(extractThisManyFromFirstRing+thisManyFromSecondRing)*repeatThisManyTimes$$

1.2 CONCERNING TESTS

Tests are performed on rings with integer keys and string values representing the things which helped me go through the process of studying and coding and it is music, specifically music bands.

2. INTERFACES OF CLASSES

```
template <typename Key, typename Info>
class BiRing {
	public:
	struct KeyInfoPair {
		Key key;
		Info info;

		KeyInfoPair() = default;
		KeyInfoPair(Key newKey, Info newInfo);
	};

	private:
	struct Node {
		KeyInfoPair keyAndInfo;
		Node* next;
		Node* prev;
		Node();
		Node(Key newKey, Info newInfo, Node* node, Node* prevNode);
	};
	Node* anchor;
	int nodeCount;
	void copyNodes(const BiRing&);

	public:

	BiRing();
	~BiRing();
	BiRing(const BiRing&);
	BiRing(BiRing&&);
	BiRing<Key, Info>& operator=(const BiRing<Key, Info>&);
	BiRing<Key, Info>& operator=(BiRing<Key, Info>&&);
	bool operator==(const BiRing<Key, Info>&) const;
	bool operator!=(const BiRing<Key, Info>&) const;

	class iterator;
	class const_iterator;

	iterator begin();
	iterator end();
	const_iterator cbegin() const;
	const_iterator cend() const;
	iterator insert(const KeyInfoPair&, iterator& beforeThis);	// inserts the node before the specified element
	void push_back(const KeyInfoPair&);	// inserts the node at the abstract "end", so simply before the anchor
	iterator erase(const iterator&);	// returns the next iterator
```

```
        iterator find(const Key&, int which = 0);
        const_iterator find(const Key&, int which = 0) const;
        void advance(const Key&, const_iterator&) const;
        void clear();
        void eraseKey(const Key&);
        bool empty() const;
        bool doesExist(const Key&) const;
        int size() const;
        int howMany(const Key&) const;
        void print();
        void printFrom(const Key&, int which = 0);

        private:
        const_iterator internalFind(const Key&, int) const;
};
```

```
template <typename Key, typename Info>
class BiRing<Key, Info>::iterator {

        friend class BiRing;

        private:
        Node* node;
        iterator(const const_iterator&);

        public:
        iterator();
        KeyInfoPair& operator*();
        KeyInfoPair* operator->();
        iterator& operator++();
        iterator operator++(int);
        iterator& operator--();
        iterator operator--(int);
        bool operator==(const iterator&) const;
        bool operator!=(const iterator&) const;

};
```

```
template <typename Key, typename Info>
class BiRing<Key, Info>::const_iterator {

        friend class BiRing;

        private:
        Node* node;
        public:
        const_iterator();
        const KeyInfoPair& operator*();
        const KeyInfoPair* operator->();
        const_iterator& operator++();
        const_iterator operator++(int);
        const_iterator& operator--();
        const_iterator operator--(int);
        bool operator==(const const_iterator&) const;
        bool operator!=(const const_iterator&) const;
};
```

3. EXTERNAL FUNCTION SHUFFLE

```
template <typename Key, typename Info>
BiRing<Key, Info> shuffle(const BiRing<Key, Info>& first, const BiRing<Key, Info>& second, int fromFirst, int fromSecond, int
                                                                                                    reps) {
        BiRing<Key, Info> shuffledRing;
        if(first.empty() || second.empty()) return shuffledRing;
        if(fromFirst < 0 || fromSecond < 0 || reps < 1) return shuffledRing;
        if(fromFirst + fromSecond == 0) return shuffledRing;

        typename BiRing<Key, Info>::const_iterator firstIt;
        firstIt = first.cbegin();
        typename BiRing<Key, Info>::const_iterator secondIt;
        secondIt = second.cbegin();

        for(int r = 0; r < reps; r++){
                for(int f = 0; f < fromFirst; f++){
                        Key key = firstIt->key;
                        Info info = firstIt->info;
                        shuffledRing.push_back(typename BiRing<Key, Info>::KeyInfoPair(key, info));
                        ++firstIt;
                }
                for(int s = 0; s < fromSecond; s++){
                        Key key2 = secondIt->key;
                        Info info2 = secondIt->info;
                        shuffledRing.push_back(typename BiRing<Key, Info>::KeyInfoPair(key2, info2));
                        ++secondIt;
                }
        }
        return shuffledRing;
}
```