Algorithms and Data Structures

Class Sequence

Documentation

Author: Kacper Wojakowski

kwojakow

293064

TABLE OF CONTENTS

1 GENERAL INFORMATION

1.1 Overview of the *Sequence* template

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

Sequence is a class template, implemented as a single linked list, 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*. *Sequence* allows multiple occurrences of the same *Key*. The class *Sequence* supports multiple ways for inserting, removing and accessing its *Nodes*. The code to the class template and function declarations is stored in the *sequence.h* header file.

The *shuffle* method is defined and declared in a separate file, *shuffle.h* which has to be included in order to call the function.

1.2 Template parameters

```
template <typename Key, typename Info> class Sequence
```

`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 *Node*
`Node *head`	Pointer to a *Node* which marks the start of the *Sequence*

1.4 Overview of the methods .....

Standard methods:

```
Sequence();
Sequence( const Sequence<Key, Info> &seq );
~Sequence();
```

Available operators:

```
Sequence<Key, Info>& operator=( const Sequence<Key, Info> &rhs );
bool operator==( const Sequence<Key, Info> &rhs ) const;
bool operator!=( const Sequence<Key, Info> &rhs ) const;
Sequence<Key, Info> operator+( const Sequence<Key, Info> &rhs ) const;
Sequence<Key, Info>& operator+=( const Sequence<Key, Info> &rhs );
```

General methods:

```
void clear();
int length() const;
bool isEmpty() const;
```

Printing:

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

Access to first/last element:

```
void pushFront( const Key &key, const Info &info );
void pushBack( const Key &key, const Info &info );
Info popFront();
Info popBack();
Info front() const;
Info back() const;
```

Access to elements by Key:

```
bool insertAfter( const Key &location, const Key &newKey,
                  const Info &newInfo, int occurence = 1 );
bool removeByKey( const Key &location, int occurence = 1 );
Info getInfo( const Key &key, int occurence = 1 );
bool search( const Key &key ) const;
```

Access to elements by index:

```
bool getNode( int index, Key &key, Info &info ) const;
bool removeByIndex( int index );
```

Modifiers:

```
Sequence<Key, Info> merge( const Sequence<Key, Info> &seq ) const;
```

2 METHOD DETAILS

2.1 Standard methods.....

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

Copy constructor	
`Sequence( const Sequence<Key, Info> &seq );`	
Parameters:	*seq* – constant reference to *Sequence* to be copied from
Returns:	-
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Calls *operator=* (see 2.2)

Destructor	
`~Sequence();`	
Parameters:	-
Returns:	-
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Calls the *clear* method (see 2.3)

2.2 Operators

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

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

Addition operator +	
`Sequence<Key, Info> operator+( const Sequence<Key, Info> &rhs ) const;`	
Parameters:	*rhs* – a constant reference to a *Sequence* to add
Returns:	A *Sequence* merged from both arguments
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Calls *merge* (see 2.8)

Addition/Assignment operator +=	
`Sequence<Key, Info>& operator+=( const Sequence<Key, Info> &rhs );`	
Parameters:	*rhs* – a constant reference to a *Sequence* to add
Returns:	Reference to a *Sequence* merged from both arguments
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Calls *merge* (see 2.8)

2.3 General methods.....

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

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

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

2.4 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*

printBackwards	
`void printBackwards( 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*

2.5 Access to first/last element .....

pushFront	
`void pushFront( const Key &key, const Info &info );`	
Parameters:	*key* – constant reference to the *Key* of the new *Node* *info* – constant reference to the *Info* of the new *Node*
Returns:	-
Complexity:	Constant O(1)
Exceptions:	May throw *std::bad_alloc*
Notes:	Adds a new *Node* at the beginning of the *Sequence*

pushBack	
`void pushBack( const Key &key, const Info &info );`	
Parameters:	*key* – constant reference to the *Key* of the new *Node* *info* – constant reference to the *Info* of the new *Node*
Returns:	-
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Adds a new *Node* at the end of the *Sequence*

popFront	
`Info popFront();`	
Parameters:	-
Returns:	*Info* of the deleted *Node*
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	Deletes the first *Node* of the *Sequence*

popBack	
`Info popBack();`	
Parameters:	-
Returns:	*Info* of the deleted *Node*
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Deletes the last *Node* of the *Sequence*

front	
`Info front() const;`	
Parameters:	-
Returns:	*Info* of the first *Node*
Complexity:	Constant O(1)
Exceptions:	Exception safe
Notes:	If the list is empty, returns default value of *Info*

Name	
`Info back() const;`	
Parameters:	-
Returns:	*Info* of the last *Node*
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	If the list is empty, returns default value of *Info*

2.6 Access to elements by Key.....

insertAfter	
`bool insertAfter( const Key &location, const Key &newKey, const Info &newInfo, int occurence = 1 );`	
Parameters:	*location* – constant reference to the *Key* marking after which *Node* to insert a new one *newKey* – constant reference to the *Key* of the new *Node* *newInfo* – constant reference to the *Info* of the new *Node* *occurrence* – integer denoting after which occurrence of *location* to insert the *Node* (defaults to 1)
Returns:	*true* if insertion was successful, *false* otherwise
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Inserts a new *Node* at a specified place, if occurrence is <1 it always returns *false*

removeByKey	
`bool removeByKey( 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:	*true* if removal was successful, *false* otherwise
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Removes a specified *Node*, always returns *false* if occurrence <1

getInfo	
`Info getInfo( const Key &key, int occurence = 1 );`	
Parameters:	*location* – constant reference to the *Key* to retrieve *Info* from *occurrence* – integer denoting which occurrence of *location* to retrieve from (defaults to 1)
Returns:	*Info* of the specified *Node*
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Gets *Info* of *Node* specified by *Key*, if it can't be found returns default value of *Info*

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*

2.7 Access to elements by index

getNode	
`bool getNode( int index, Key &key, Info &info ) const;`	
Parameters:	*index* – an integer denoting the index to retrieve *Node* from *key* – a reference to a *Key* to which *Node* should be outputted *info* – a reference to an *Info* to which *Node* should be outputted
Returns:	*true* if successful, *false* otherwise
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	Retrieves *Node* based on index, if unsuccessful *key* and *info* stay unchanged

Name	
`bool removeByIndex( int index );`	
Parameters:	*index* – an integer denoting the index of the *Node* to delete
Returns:	*true* if deletion was successful, *false* otherwise
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	-

2.8 Modifiers

merge	
`Sequence<Key, Info> merge( const Sequence<Key, Info> &seq ) const;`	
Parameters:	*seq* – constant reference to *Sequence* to merge with
Returns:	A new *Sequence* merged from the two
Complexity:	Linear O(n)
Exceptions:	May throw *std::bad_alloc*
Notes:	Merges two *Sequences* together, calls copy constructor (see 2.1)

3. SHUFFLE METHOD

3.1 Overview

```
template <typename Key, typename Info>
Sequence<Key, Info> shuffle( const Sequence<Key, Info> &S1, int start1, int len1,
                             const Sequence<Key, Info> &S2, int start2, int len2,
                                                                        int repeat)
```

The function “shuffles” two *Sequences* S1 and S2. If any of the arguments are incorrect (len2 < 1, for example) the function returns an empty *Sequence*.

3.2 Details of implementation

The main part of the function is based mostly on two methods:

- *getNode* (see 2.7)
- *pushBack* (see 2.5)

It also calls other methods in the process.
When copying from input *Sequences* the function copies both *Key* and *Info* of each node.