Algorithms and Data Structures

Dictionary

Documentation

Author: Kacper Wojakowski

kwojakow

293064

TABLE OF CONTENTS

1 GENERAL INFORMATION

1.1 Overview of the *Dictionary* template

The C++11 standard is required to properly use the *Dictionary* class, as it uses some of it features (such as *auto* and *nullptr*). The class was written and compiled using Visual Studio 2017, and tested on the lab011 server.

Dictionary is a class template, implemented as an AVL Tree (height balanced binary search tree), 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*. *Dictionary* does not allow multiple occurrences of the same *Key*. The class *Dictionary* balances itself every time an element is added or removed to ensure best searching time. The code to the class template and function declarations is stored in the *Dictionary.h* header file.

1.2 Template parameters

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

`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, a balance factor and pointers to left and right *Nodes*
`Node *root`	Pointer to a *Node* which marks the start of the *Dictionary*

Public member types:

`class DictionaryError`	An exception class which is thrown whenever methods are called with incorrect data, or the class encounters an unexpected situation in the tree.

1.4 Overview of the methods .....

Standard methods:

```
Dictionary();
Dictionary( const Dictionary& src );
~Dictionary();
```

Available operators:

```
Dictionary& operator=( const Dictionary& rhs );
bool operator==( const Dictionary& rhs ) const noexcept;
bool operator!=( const Dictionary& rhs ) const noexcept;
```

General methods:

```
void clear();
int height() const noexcept;
bool empty() const noexcept;
```

Printing:

```
void display( std::ostream& os = std::cout ) const noexcept
```

Access to elements:

```
Info get( const Key& elem ) const;
```

Insertion/Removal

```
void insert( const Key& nKey, const Info& nInfo )
void remove( const Key& elem )
```

Most of the public methods call one or more of the private methods:

Recursive methods:

`Node* copy( Node* src );`
`void display( std::ostream& os, Node* node, int indent = 0 ) const noexcept;`
`void clear( Node*& node );`
`bool compare( Node* rhs, Node* lhs ) const noexcept;`
`bool insert( Node*& dest, const Key& nKey, const Info& nInfo );`
`bool remove( const Key& elem, Node*& dest );`
`int height( Node* dest ) const noexcept;`

Rotation methods:

`void lRotate( Node*& dest );`
`void rRotate( Node*& dest );`
`void lrRotate( Node*& dest );`
`void rlRotate( Node*& dest );`

2 METHOD DETAILS

2.1 Standard methods.....

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

Copy constructor	
`Dictionary( const Dictionary& src );`	
Parameters:	*seq* – constant reference to *Dictionary* to be copied from
Returns:	-
Complexity:	Exponential $O(2^n)$
Exceptions:	May throw *std::bad_alloc*
Notes:	Calls *operator=* (see 2.2)

Destructor	
`~Dictionary();`	
Parameters:	-
Returns:	-
Complexity:	Exponential $O(2^n)$
Exceptions:	Exception safe
Notes:	Calls the *clear* method (see 2.3)

2.2 Operators

Assignment operator =	
`Dictionary& operator=( const Dictionary& rhs );`	
Parameters:	*rhs* – constant reference to a *Dictionary* to be assigned
Returns:	Copy of *rhs*
Complexity:	Exponential $O(2^n)$
Exceptions:	May throw *std::bad_alloc*
Notes:	Clears the tree and copies the elements from *rhs*

Comparison operator ==	
`bool operator==( const Dictionary& rhs ) const noexcept;`	
Parameters:	*rhs* – constant reference to a *Dictionary* to compare to
Returns:	*true* if both *Dictionaries* are identical, *false* otherwise
Complexity:	Exponential $O(2^n)$
Exceptions:	Exception safe
Notes:	operator== must be defined for both *Key* and *Info*

Comparison operator !=	
`bool operator!=( const Dictionary& rhs ) const noexcept;`	
Parameters:	*rhs* – constant reference to a *Dictionary* to compare to
Returns:	*true* if both *Dictionary* are not identical, *false* otherwise
Complexity:	Exponential $O(2^n)$
Exceptions:	Exception safe
Notes:	Calls *operator==*, operator== must be defined for both *Key* and *Info*

2.3 General methods.....

clear	
`void clear();`	
Parameters:	-
Returns:	-
Complexity:	Exponential $O(2^n)$
Exceptions:	Exception safe
Notes:	Deletes every single element from the tree

height	
`int height() const noexcept;`	
Parameters:	-
Returns:	Height of the *Dictionary*
Complexity:	Linear O(n)
Exceptions:	Exception safe
Notes:	-

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

2.4 Printing .....

display	
`void display( std::ostream& os = std::cout ) const noexcept`	
Parameters:	A reference to an *std::ostream* object, defaults to *std::cout*
Returns:	-
Complexity:	Exponential $O(2^n)$
Exceptions:	Exception safe
Notes:	operator<< must be defined for *Key* and *Info* Works best only when *Keys* are no more than 8 characters long Used mostly for testing (checking whether the tree is properly balanced)

2.5 Access to elements .....

get	
`Info get( const Key& elem ) const;`	
Parameters:	*key* – constant reference to the *Key* of the element to be retrieved
Returns:	*Info* of the element being retrieved
Complexity:	Logarithmic $O(\log_2 n)$
Exceptions:	May throw *DictionaryError*
Notes:	Throws an exception when *elem* does not exist in the tree

2.6 Insertion/Removal.....

insertAfter	
`void insert( const Key& nKey, const Info& nInfo )`	
Parameters:	*nKey* – constant reference to Key to be inserted *nInfo* – constant reference to *Info* to be inserted
Returns:	-
Complexity:	Logarithmic $O(\log_2 n)$
Exceptions:	May throw *std::bad_alloc* and *DictionaryError*
Notes:	Throws an exception when the *Key* already exists in the tree

remove	
`void remove( const Key& elem )`	
Parameters:	*elem* – constant reference to Key to be removed
Returns:	-
Complexity:	Logarithmic $O(\log_2 n)$
Exceptions:	May throw *DictionaryError*
Notes:	Throws an exception when the *Key* does not exist in the tree

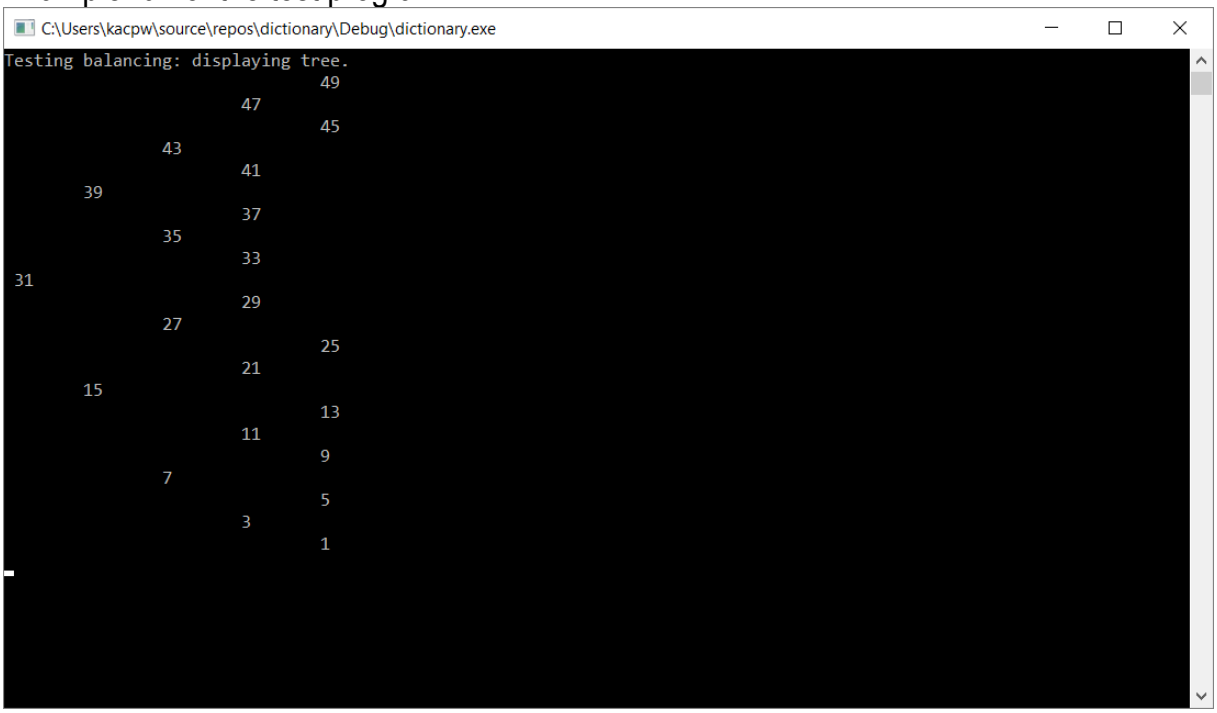
3. TESTING APPROACH

3.1 Overview

All the tests are written and implemented in *main.cpp*, in the *sourcefiles* folder. Should any errors occur, the information about them will be printed into the *std::cerr* error stream. Testing the balance of the tree is done visually, by looking at the results of the method *display()*.

3.2 Example

Example run of the test program:



No error messages were printed, printed tree is properly balanced