Count of Words: Data Structures Edition

Generated by Doxygen 1.9.8

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

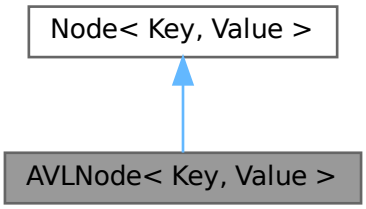
Class Documentation

4.1 AVLNode< Key, Value > Struct Template Reference

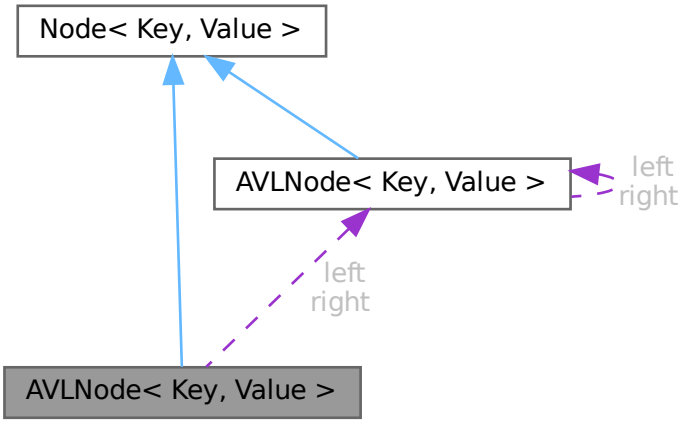
AVL tree node structure extending a generic Node.

`#include <AVLNode.hpp>`

Inheritance diagram for AVLNode< Key, Value >:



Collaboration diagram for AVLNode< Key, Value >:



Public Member Functions

- AVLNode (const Key &k, const Value &v)

  Constructs a Node object with the given key and value.

Public Member Functions inherited from Node< Key, Value >

- Node (const Key &key, const Value &value)

  Constructs a Node with a given key and value.
- const Key & getKey () const

  Gets the key stored in the node.
- void setKey (const Key &key)

  Sets the key in the node.
- const Value & getValue () const

  Gets the value stored in the node (read-only).
- Value & getValue ()

  Gets the value stored in the node (modifiable).
- void setValue (const Value &value)

  Sets the value in the node.
- void update (const Key &key, const Value &value)

  Updates both the key and the value of the node.
- std::string show () const

  Returns a string representation of the node as (key, value).
- ~**Node** ()=default

  Default destructor.

Public Attributes

- AVLNode * **left**

  Pointer to the left child.
- AVLNode * **right**

  Pointer to the right child.
- size_t **height**

  Height of the node in the AVL tree.

4.1.1 Detailed Description

template<typename Key, typename Value>
struct AVLNode< Key, Value >

AVL tree node structure extending a generic Node.

Template Parameters

Key	The type of the key.
Value	The type of the value.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 AVLNode()

```
template<typename Key , typename Value >
AVLNode< Key, Value >::AVLNode (
            const Key & k,
            const Value & v )  [inline]
```

Constructs a Node object with the given key and value.

Parameters

k	The key associated with the node.
v	The value associated with the node.

The documentation for this struct was generated from the following file:

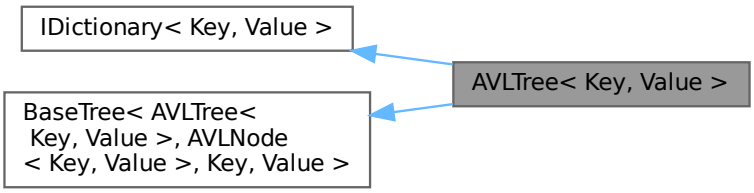
- include/Trees/AVL/AVLNode.hpp

4.2 AVLTree< Key, Value > Class Template Reference

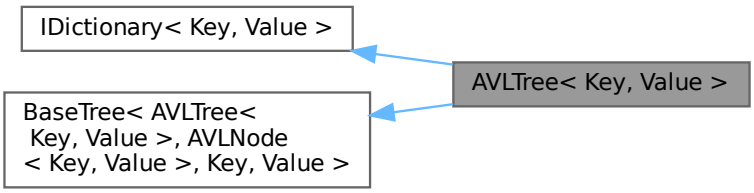
A class representing an AVL Tree.

```
#include <AVLTree.hpp>
```

Inheritance diagram for AVLTree< Key, Value >:



Collaboration diagram for AVLTree< Key, Value >:



Public Member Functions

- **AVLTree** ()

  Constructs an empty AVL tree.
- ~**AVLTree** ()

  Destroys the AVL tree and deallocates all resources.
- void insert (const Key &key, const Value &value) override

  Inserts a new key-value pair into the AVL tree.
- bool find (const Key &key, Value &outValue) const override

  Searches for a key in the AVL tree and retrieves its associated value.
- void update (const Key &key, const Value &value) override

  Updates the value associated with a given key in the AVL tree.
- void remove (const Key &key) override

  Removes a node with the specified key from the AVL tree.
- void clear () override

  Clears the AVL tree by deallocating all nodes.
- void printInOrder (std::ostream &out) const override

  Prints the elements of the AVL tree in in-order traversal.
- size_t getComparisonsCount () const override

  Retrieves the count of comparisons made during operations on the AVL tree.
- Value & operator[ ] (const Key &key) override

  Accesses the value associated with a given key.

- const Value & operator[] (const Key &key) const override

  Accesses the value associated with a given key (const version).
- void **print** () const

  Prints the AVL tree structure.
- size_t getRotationsCount () const

  Retrieves the total number of rotations performed by the AVL tree.

Public Member Functions inherited from IDictionary< Key, Value >

- virtual ~**IDictionary** ()=default

  Virtual destructor.

Static Public Attributes

- static const int **IMBALANCE** = 2

  The imbalance threshold for the AVL tree.

Additional Inherited Members

Protected Member Functions inherited from IDictionary< Key, Value >

- void incrementCounter (size_t n) const

  Increments the number of comparisons by a given amount.
- void **resetCounter** () const

  Resets the comparisons counter to zero.

Protected Member Functions inherited from BaseTree< AVLTree< Key, Value >, AVLNode< Key, Value >, Key, Value >

- BaseTree (AVLNode< Key, Value > *r)

  Constructs a BaseTree with the given root node.
- const AVLNode< Key, Value > * findNode (const Key &key, AVLNode< Key, Value > *comp=nullptr) const

  Finds a node with the specified key in the tree.
- AVLNode< Key, Value > * minimum (AVLNode< Key, Value > *node) const

  Finds the node with the minimum key in the subtree rooted at the given node.
- void clearNode (AVLNode< Key, Value > *node, AVLNode< Key, Value > *comp)

  Recursively clears (deletes) nodes in a subtree, avoiding a comparison node.
- void reset (AVLNode< Key, Value > *node, AVLNode< Key, Value > *comp=nullptr, AVLNode< Key, Value > *defaultRoot=nullptr)

  Resets the tree by clearing all nodes starting from the given node, except for the comparison node, and sets the root to the default root.
- void inOrderTransversal (std::ostream &out, AVLNode< Key, Value > *node, AVLNode< Key, Value > *comp) const

  Performs an in-order traversal of the subtree and prints node information to an output stream.
- const Value & at (const Key &key, AVLNode< Key, Value > *comp=nullptr) const

  Accesses the value associated with a given key.
- void setMaxKeyLen (const Key &key)

  Updates the maximum key length stored in the tree.
- void setMaxValLen (const Value &value)

  Updates the maximum length of the value in the tree.
- void incrementRotationsCount (size_t amount=1)

  Increments the count of rotations performed on the tree.

Protected Attributes inherited from IDictionary< Key, Value >

- size_t comparisonsCount = 0

  Tracks the number of comparisons made during dictionary operations.

Protected Attributes inherited from BaseTree< AVLTree< Key, Value >, AVLNode< Key, Value >, Key, Value >

- AVLNode< Key, Value > * root

  Pointer to the root node of the tree.
- size_t **maxKeyLen**

  Represents the maximum length of a key that can be stored in the tree.
- size_t maxValLen

  Represents the maximum length of a value in the tree.
- size_t rotationsCount

  Tracks the number of rotations performed in the tree.

4.2.1 Detailed Description

template<typename Key, typename Value>
class AVLTree< Key, Value >

A class representing an AVL Tree.

The AVLTree class implements a self-balancing binary search tree that maintains the AVL property. It supports operations such as insertion, deletion, and search while ensuring logarithmic time complexity.

Template Parameters

Key	The type of the keys stored in the tree.
Value	The type of the values associated with the keys.

4.2.2 Member Function Documentation

4.2.2.1 clear()

```
template<typename Key , typename Value >
void AVLTree< Key, Value >::clear ( )  [override], [virtual]
```

Clears the AVL tree by deallocating all nodes.

Implements IDictionary< Key, Value >.

4.2.2.2 find()

```
template<typename Key , typename Value >
bool AVLTree< Key, Value >::find (
            const Key & key,
            Value & outValue ) const  [override], [virtual]
```

Searches for a key in the AVL tree and retrieves its associated value.

Parameters

key	The key to search for.
outValue	A reference to store the associated value if the key is found.

Returns

true If the key is found.

false If the key is not found.

Implements IDictionary< Key, Value >.

4.2.2.3 getComparisonsCount()

```
template<typename Key , typename Value >
size_t AVLTree< Key, Value >::getComparisonsCount ( ) const  [override], [virtual]
```

Retrieves the count of comparisons made during operations on the AVL tree.

Returns

The total number of comparisons made.

Implements IDictionary< Key, Value >.

4.2.2.4 getRotationsCount()

```
template<typename Key , typename Value >
size_t AVLTree< Key, Value >::getRotationsCount ( ) const
```

Retrieves the total number of rotations performed by the AVL tree.

This function returns the count of rotations (both single and double) that have been executed to maintain the balance of the AVL tree during insertions, deletions, or updates.

Returns

size_t The number of rotations performed.

4.2.2.5 insert()

```
template<typename Key , typename Value >
void AVLTree< Key, Value >::insert (
            const Key & key,
            const Value & value )  [override], [virtual]
```

Inserts a new key-value pair into the AVL tree.

Parameters

key	The key to be inserted.
value	The value associated with the key.

This method updates the `root` of the AVL tree after insertion and also updates `maxKeyLen` and `maxValLen` based on the display size of the inserted key and value, respectively.

Implements IDictionary< Key, Value >.

4.2.2.6 operator[]() [1/2]

```
template<typename Key , typename Value >
const Value & AVLTree< Key, Value >::operator[] (
            const Key & key ) const  [override], [virtual]
```

Accesses the value associated with a given key (const version).

Parameters

key	The key to access.

Returns

A const reference to the associated value.

Implements IDictionary< Key, Value >.

4.2.2.7 operator[]() [2/2]

```
template<typename Key , typename Value >
Value & AVLTree< Key, Value >::operator[] (
            const Key & key )  [override], [virtual]
```

Accesses the value associated with a given key.

Parameters

key	The key to access.

Returns

A reference to the associated value.

Implements IDictionary< Key, Value >.

4.2.2.8 printInOrder()

```
template<typename Key , typename Value >
void AVLTree< Key, Value >::printInOrder (
            std::ostream & out ) const  [override], [virtual]
```

Prints the elements of the AVL tree in in-order traversal.

Parameters

out	The output stream where the traversal result will be written.

Implements IDictionary< Key, Value >.

4.2.2.9 remove()

```
template<typename Key , typename Value >
void AVLTree< Key, Value >::remove (
            const Key & key )  [override], [virtual]
```

Removes a node with the specified key from the AVL tree.

Parameters

key	The key of the node to be removed.

Implements IDictionary< Key, Value >.

4.2.2.10 update()

```
template<typename Key , typename Value >
void AVLTree< Key, Value >::update (
            const Key & key,
            const Value & value )  [override], [virtual]
```

Updates the value associated with a given key in the AVL tree.

Parameters

key	The key to update.
value	The new value to associate with the key.

Exceptions

KeyNotFoundException	If the key is not found in the tree.

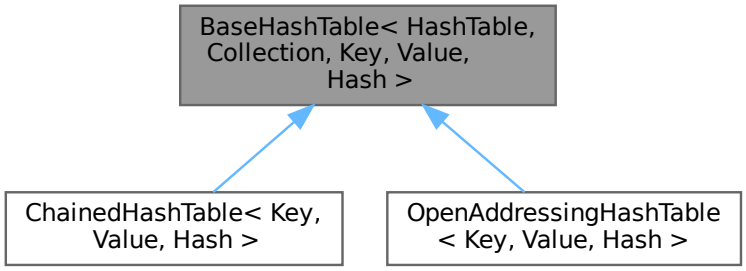
Implements IDictionary< Key, Value >.

The documentation for this class was generated from the following files:

- include/Trees/AVL/AVLTree.hpp
- include/Trees/AVL/AVLTree.impl.hpp

4.3 BaseHashTable< HashTable, Collection, Key, Value, Hash > Class Template Reference

Inheritance diagram for BaseHashTable< HashTable, Collection, Key, Value, Hash >:



Public Member Functions

- BaseHashTable (size_t size=7, float mlf=0.7)

  Constructs a new BaseHashTable object.
- float getLoadFactor () const

  Calculates and returns the current load factor of the hash table.
- void clearHashTable ()

  Clears all elements from the hash table, resetting it to an empty state.
- void incrementCollisionsCount (size_t m=1) const

  Increments the count of collisions in the hash table.

Protected Member Functions

- size_t getNextPrime (size_t num) const

  Calculates and returns the next prime number greater than or equal to a given number.
- void checkAndRehash ()

  Checks the current load factor and triggers a rehash if necessary.

Protected Attributes

- std::vector< Collection > **table**

  The hash table's internal storage, composed of collections (e.g., lists or buckets).
- size_t **tableSize**

  The current size of the hash table (number of slots).
- float **maxLoadFactor**

  The maximum load factor before the table is resized.
- size_t **numberOfElements**

  The number of elements currently stored in the table.
- Hash **hashing**

  Hash function object used to compute the index for each key.
- size_t **collisionsCount**

  Number of collisions occurred during insertions and searches.

4.3.1 Constructor & Destructor Documentation

4.3.1.1 BaseHashTable()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
BaseHashTable< HashTable, Collection, Key, Value, Hash >::BaseHashTable (
            size_t size = 7,
            float mlf = 0.7 )
```

Constructs a new BaseHashTable object.

Initializes a new hash table with a specified initial size and maximum load factor. The actual size of the hash table is set to the next prime number greater than or equal to the provided `size` to optimize hash distribution. The `table` (likely a vector of `Collection`s) is then resized accordingly.

The `maxLoadFactor` is set based on the `mlf` parameter. If `mlf` is less than or equal to 0, it defaults to 0.7 to ensure a reasonable threshold for rehashing. The `numberOfElements` is initialized to 0, as the table is empty upon construction.

Parameters

size	The desired initial size of the hash table. This will be adjusted to the next prime.
mlf	The maximum load factor for the hash table. If the load factor exceeds this value, a rehash operation will be triggered.

4.3.2 Member Function Documentation

4.3.2.1 checkAndRehash()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
void BaseHashTable< HashTable, Collection, Key, Value, Hash >::checkAndRehash ( )  [protected]
```

Checks the current load factor and triggers a rehash if necessary.

This method is responsible for maintaining the efficiency of the hash table. It compares the current load factor (the ratio of elements to table size) against a predefined maximum load factor (`maxLoadFactor`). If the current load factor exceeds this maximum, it indicates that the hash table is becoming too dense, potentially leading to increased collision rates and slower operations.

In such a scenario, the method initiates a rehash operation by calling the `rehash` method of the derived `Hash↩Table` class (using CRTP) with a new table size that is double the current `tableSize`. This effectively resizes the hash table and redistributes existing elements, improving performance.

4.3.2.2 clearHashTable()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
void BaseHashTable< HashTable, Collection, Key, Value, Hash >::clearHashTable ( )
```

Clears all elements from the hash table, resetting it to an empty state.

This method effectively empties the hash table while retaining its current capacity. It first clears all elements from the underlying `table` (which likely holds the collections for each bucket). After clearing, it resizes the `table` back to its original `tableSize`, ensuring that the structure remains intact but empty. Finally, `numberOfElements` is reset to 0, accurately reflecting the empty state.

4.3.2.3 getLoadFactor()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
float BaseHashTable< HashTable, Collection, Key, Value, Hash >::getLoadFactor ( ) const
```

Calculates and returns the current load factor of the hash table.

The load factor is defined as the ratio of the number of elements stored in the hash table to the total number of slots (buckets) in the table.

Returns

size_t The current load factor as a floating-point value.

4.3.2.4 getNextPrime()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
size_t BaseHashTable< HashTable, Collection, Key, Value, Hash >::getNextPrime (
            size_t num ) const  [protected]
```

Calculates and returns the next prime number greater than or equal to a given number.

This method finds the smallest prime number that is greater than or equal to the input `num`. It uses a lambda function `isPrime` to efficiently check for primality. The search starts from `num` (or `num + 1` if `num` is even) and increments by 2 to only check odd numbers, optimizing the search.

Parameters

num	The starting number from which to find the next prime.

Returns

The next prime number greater than or equal to `num`.

4.3.2.5 incrementCollisionsCount()

```
template<typename HashTable , typename Collection , typename Key , typename Value , typename
Hash >
void BaseHashTable< HashTable, Collection, Key, Value, Hash >::incrementCollisionsCount (
            size_t m = 1 ) const
```

Increments the count of collisions in the hash table.

This method increases the internal counter that tracks the number of collisions encountered during operations on the hash table. Collisions occur when multiple keys are hashed to the same index in the table.

Parameters

m	The amount by which to increment the collision count. Defaults to 1.

The documentation for this class was generated from the following files:

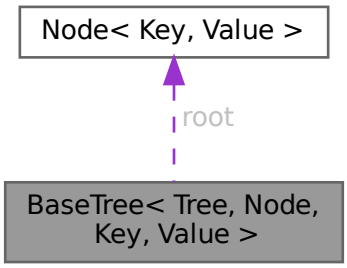
- include/HashTables/Base/BaseHashTable.hpp
- include/HashTables/Base/BaseHashTable.impl.hpp

4.4 BaseTree< Tree, Node, Key, Value > Class Template Reference

A base template class for tree structures.

`#include <BaseTree.hpp>`

Collaboration diagram for BaseTree< Tree, Node, Key, Value >:



Protected Member Functions

- BaseTree (Node ∗r)

  Constructs a BaseTree with the given root node.
- const Node ∗ findNode (const Key &key, Node ∗comp=nullptr) const

  Finds a node with the specified key in the tree.
- Node ∗ minimum (Node ∗node) const

  Finds the node with the minimum key in the subtree rooted at the given node.
- void clearNode (Node ∗node, Node ∗comp)

  Recursively clears (deletes) nodes in a subtree, avoiding a comparison node.
- void reset (Node ∗node, Node ∗comp=nullptr, Node ∗defaultRoot=nullptr)

  Resets the tree by clearing all nodes starting from the given node, except for the comparison node, and sets the root to the default root.
- void inOrderTransversal (std::ostream &out, Node ∗node, Node ∗comp) const

  Performs an in-order traversal of the subtree and prints node information to an output stream.
- const Value & at (const Key &key, Node ∗comp=nullptr) const

  Accesses the value associated with a given key.
- void setMaxKeyLen (const Key &key)

  Updates the maximum key length stored in the tree.
- void setMaxValLen (const Value &value)

  Updates the maximum length of the value in the tree.
- void incrementRotationsCount (size_t amount=1)

  Increments the count of rotations performed on the tree.

Protected Attributes

- Node ∗ root

  Pointer to the root node of the tree.
- size_t **maxKeyLen**

  Represents the maximum length of a key that can be stored in the tree.
- size_t maxValLen

  Represents the maximum length of a value in the tree.
- size_t rotationsCount

  Tracks the number of rotations performed in the tree.

4.4.1 Detailed Description

template<typename Tree, typename Node, typename Key, typename Value>
class BaseTree< Tree, Node, Key, Value >

A base template class for tree structures.

This class provides common functionality for various tree implementations, such as searching for nodes, finding minimum elements, clearing nodes, in-order traversal, and accessing values by key. It uses the curiously recurring template pattern (CRTP) to interact with derived tree classes.

Template Parameters

Tree	The derived tree class (CRTP).
Node	The node type used in the tree.
Key	The type of the keys stored in the tree.
Value	The type of the values stored in the tree.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 BaseTree()

```
template<typename Tree , typename Node , typename Key , typename Value >
BaseTree< Tree, Node, Key, Value >::BaseTree (
            Node * r )  [protected]
```

Constructs a BaseTree with the given root node.

Parameters

r	Pointer to the root node of the tree.

This constructor initializes the BaseTree with the provided root node. It also sets the initial values for the maximum key length, maximum value length, and the rotations count to zero. Additionally, it clears the counter used for tracking operations.

4.4.3 Member Function Documentation

4.4.3.1 at()

```
template<typename Tree , typename Node , typename Key , typename Value >
const Value & BaseTree< Tree, Node, Key, Value >::at (
            const Key & key,
            Node * comp = nullptr ) const  [protected]
```

Accesses the value associated with a given key.

Parameters

key	The key whose associated value is to be returned.

Returns

A const reference to the value associated with the key.

Exceptions

KeyNotFoundException	If the key is not found in the tree.

4.4.3.2 clearNode()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::clearNode (
            Node * node,
            Node * comp )  [protected]
```

Recursively clears (deletes) nodes in a subtree, avoiding a comparison node.

This method is typically used in destructors to deallocate tree nodes.

Parameters

node	The current node to clear.
comp	A comparison node (e.g., a sentinel or a node not to be deleted).

4.4.3.3 findNode()

```
template<typename Tree , typename Node , typename Key , typename Value >
const Node * BaseTree< Tree, Node, Key, Value >::findNode (
            const Key & key,
            Node * comp = nullptr ) const  [protected]
```

Finds a node with the specified key in the tree.

Parameters

key	The key to search for.

Returns

A const pointer to the node if found, nullptr otherwise.

4.4.3.4 incrementRotationsCount()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::incrementRotationsCount (
            size_t amount = 1 )  [protected]
```

Increments the count of rotations performed on the tree.

This function increases the internal counter that tracks the number of rotations performed during tree operations, such as balancing. By default, the counter is incremented by 1, but a custom amount can be specified.

Parameters

amount	The number by which to increment the rotations count. Defaults to 1 if not specified.

4.4.3.5 inOrderTransversal()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::inOrderTransversal (
            std::ostream & out,
            Node * node,
            Node * comp ) const  [protected]
```

Performs an in-order traversal of the subtree and prints node information to an output stream.

Parameters

out	The output stream to print to.
node	The current node in the traversal.
comp	A comparison node (e.g., a sentinel node to stop traversal).

4.4.3.6 minimum()

```
template<typename Tree , typename Node , typename Key , typename Value >
Node * BaseTree< Tree, Node, Key, Value >::minimum (
            Node * node ) const  [protected]
```

Finds the node with the minimum key in the subtree rooted at the given node.

Parameters

node	The root of the subtree to search within.

Returns

A pointer to the node with the minimum key.

4.4.3.7 reset()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::reset (
            Node * node,
            Node * comp = nullptr,
            Node * defaultRoot = nullptr )  [protected]
```

Resets the tree by clearing all nodes starting from the given node, except for the comparison node, and sets the root to the default root.

Parameters

node	The starting node to clear.
comp	The comparison node that will not be cleared. Defaults to nullptr.
defaultRoot	The new root node to set after clearing. Defaults to nullptr.

4.4.3.8 setMaxKeyLen()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::setMaxKeyLen (
            const Key & key )  [protected]
```

Updates the maximum key length stored in the tree.

This function calculates the length of the given key and updates the `maxKeyLen` member variable if the length of the provided key is greater than the current value of `maxKeyLen`.

Parameters

key	The key whose length is to be compared and potentially used to update the maximum key length.

4.4.3.9 setMaxValLen()

```
template<typename Tree , typename Node , typename Key , typename Value >
void BaseTree< Tree, Node, Key, Value >::setMaxValLen (
            const Value & value )  [protected]
```

Updates the maximum length of the value in the tree.

This function calculates the size of the given value and updates the `maxValLen` member variable if the size of the provided value is greater than the current `maxValLen`.

Parameters

value	The value whose size is to be compared and potentially used to update the maximum value length.

4.4.4 Member Data Documentation

4.4.4.1 maxValLen

```
template<typename Tree , typename Node , typename Key , typename Value >
size_t BaseTree< Tree, Node, Key, Value >::maxValLen  [protected]
```

Represents the maximum length of a value in the tree.

This variable is used to define the maximum number of characters or bytes that a value can have when stored in the tree structure.

4.4.4.2 root

```
template<typename Tree , typename Node , typename Key , typename Value >
Node* BaseTree< Tree, Node, Key, Value >::root  [protected]
```

Pointer to the root node of the tree.

This member variable represents the starting point of the tree structure. It is used to access and manage all nodes within the tree.

4.4.4.3 rotationsCount

```
template<typename Tree , typename Node , typename Key , typename Value >
size_t BaseTree< Tree, Node, Key, Value >::rotationsCount  [protected]
```

Tracks the number of rotations performed in the tree.

This variable is used to count the total number of rotations (e.g., left or right rotations) that have been executed to maintain the balance of the tree structure.

The documentation for this class was generated from the following files:

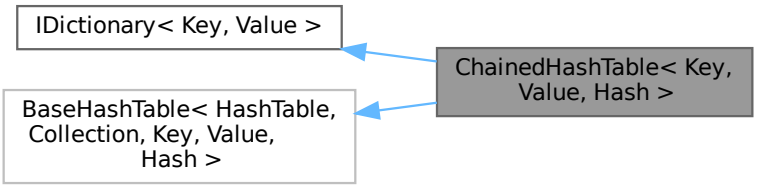
- include/Trees/Base/BaseTree.hpp
- include/Trees/Base/BaseTree.impl.hpp

4.5 ChainedHashTable< Key, Value, Hash > Class Template Reference

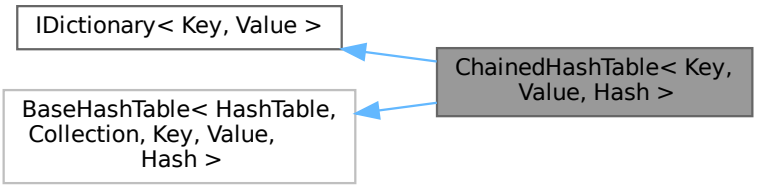
Hash table implementation using separate chaining.

```
#include <ChainedHashTable.hpp>
```

Inheritance diagram for ChainedHashTable< Key, Value, Hash >:



Collaboration diagram for ChainedHashTable< Key, Value, Hash >:



Public Member Functions

- **ChainedHashTable** (size_t size=7, float mlf=1.0)
- void insert (const Key &key, const Value &value) override

  Inserts a key-value pair into the hash table.
- bool find (const Key &key, Value &outValue) const override

  Searches for a key in the hash table and retrieves its associated value if found.
- void update (const Key &key, const Value &value) override

  Updates the value associated with a given key in the hash table.
- void remove (const Key &key) override

  Removes the key-value pair associated with the given key from the hash table.
- void clear () override

  Clears the hash table by removing all elements.
- void printInOrder (std::ostream &out) const override

  Prints the key-value pairs in the hash table to the output stream in ascending order of keys.
- size_t getComparisonsCount () const override

  Returns the current value of the comparisons count.
- Value & operator[ ] (const Key &key) override

  Provides read/write access to the value associated with the given key.
- const Value & operator[ ] (const Key &key) const override

  Provides read-only access to the value associated with the given key (const version).
- void rehash (size_t m)

  Rehashes the hash table to a new size.
- size_t getCollissionsCount () const

  Retrieves the total number of collisions that have occurred in the hash table.
- size_t getTableSize () const

  Retrieves the current size of the hash table.

Public Member Functions inherited from IDictionary< Key, Value >

- virtual ~**IDictionary** ()=default

  Virtual destructor.

Public Member Functions inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- BaseHashTable (size_t size=7, float mlf=0.7)

  Constructs a new BaseHashTable object.
- float getLoadFactor () const

  Calculates and returns the current load factor of the hash table.
- void clearHashTable ()

  Clears all elements from the hash table, resetting it to an empty state.
- void incrementCollisionsCount (size_t m=1) const

  Increments the count of collisions in the hash table.

Additional Inherited Members

Protected Member Functions inherited from IDictionary< Key, Value >

- void incrementCounter (size_t n) const

  Increments the number of comparisons by a given amount.
- void **resetCounter** () const

  Resets the comparisons counter to zero.

Protected Member Functions inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- size_t getNextPrime (size_t num) const

  Calculates and returns the next prime number greater than or equal to a given number.
- void checkAndRehash ()

  Checks the current load factor and triggers a rehash if necessary.

Protected Attributes inherited from IDictionary< Key, Value >

- size_t comparisonsCount = 0

  Tracks the number of comparisons made during dictionary operations.

Protected Attributes inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- std::vector< Collection > **table**

  The hash table's internal storage, composed of collections (e.g., lists or buckets).
- size_t **tableSize**

  The current size of the hash table (number of slots).
- float **maxLoadFactor**

  The maximum load factor before the table is resized.
- size_t **numberOfElements**

  The number of elements currently stored in the table.
- Hash **hashing**

  Hash function object used to compute the index for each key.
- size_t **collisionsCount**

  Number of collisions occurred during insertions and searches.

4.5.1 Detailed Description

template<typename Key, typename Value, typename Hash = std::hash<Key>>
class ChainedHashTable< Key, Value, Hash >

Hash table implementation using separate chaining.

A hash table implementation using chaining for collision resolution.

Template Parameters

Key	The type of the keys.
Value	The type of the values.
Hash	The hash function to be used (defaults to std::hash<Key>).

This class provides a hash table implementation that uses separate chaining to handle collisions. It supports dynamic resizing and rehashing to maintain an efficient load factor.

Parameters

size	The initial size of the hash table. Defaults to 7.
mlf	The maximum load factor before rehashing occurs. Defaults to 1.0.

4.5.2 Member Function Documentation

4.5.2.1 clear()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::clear ( )  [override], [virtual]
```

Clears the hash table by removing all elements.

This function resets the hash table to its initial state by clearing all buckets and resizing the table to its current size. The number of elements in the table is also reset to zero.

Implements IDictionary< Key, Value >.

4.5.2.2 find()

```
template<typename Key , typename Value , typename Hash >
bool ChainedHashTable< Key, Value, Hash >::find (
            const Key & key,
            Value & outValue ) const  [override], [virtual]
```

Searches for a key in the hash table and retrieves its associated value if found.

Parameters

key	The key to search for in the hash table.
outValue	A reference to a variable where the associated value will be stored if the key is found.

Returns

true If the key is found in the hash table.

false If the key is not found in the hash table.

Implements IDictionary< Key, Value >.

4.5.2.3 getCollissionsCount()

```
template<typename Key , typename Value , typename Hash >
size_t ChainedHashTable< Key, Value, Hash >::getCollissionsCount ( ) const
```

Retrieves the total number of collisions that have occurred in the hash table.

A collision occurs when two different keys are hashed to the same index in the hash table. This method provides a count of such occurrences, which can be useful for analyzing the efficiency of the hash function and the load factor of the table.

Returns

size_t The number of collisions that have occurred in the hash table.

4.5.2.4 getComparisonsCount()

```
template<typename Key , typename Value , typename Hash >
size_t ChainedHashTable< Key, Value, Hash >::getComparisonsCount ( ) const  [override], [virtual]
```

Returns the current value of the comparisons count.

This function provides access to the `comparisonsCount` attribute, which is expected to track the number of key comparisons performed by certain operations within the hash table (e.g., search, insertion).

Returns

The current number of comparisons as a `size_t`.

Note

Not passed to the base hash table by the function no need to add complexity to something relatively simple

Implements IDictionary< Key, Value >.

4.5.2.5 getTableSize()

```
template<typename Key , typename Value , typename Hash >
size_t ChainedHashTable< Key, Value, Hash >::getTableSize ( ) const
```

Retrieves the current size of the hash table.

This function returns the total number of buckets currently allocated in the hash table. It provides insight into the capacity of the table and can be useful for debugging or performance analysis.

Returns

size_t The number of buckets in the hash table.

4.5.2.6 insert()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::insert (
            const Key & key,
            const Value & value )  [override], [virtual]
```

Inserts a key-value pair into the hash table.

If the load factor exceeds the maximum load factor, the hash table will be rehashed to accommodate more elements. If the key already exists in the hash table, an exception of type KeyAlreadyExistsException will be thrown.

Parameters

key	The key to be inserted.
value	The value associated with the key.

Exceptions

KeyAlreadyExistsException	If the key already exists in the hash table.

Implements IDictionary< Key, Value >.

4.5.2.7 operator[]() [1/2]

```
template<typename Key , typename Value , typename Hash >
const Value & ChainedHashTable< Key, Value, Hash >::operator[] (
            const Key & key ) const  [override], [virtual]
```

Provides read-only access to the value associated with the given key (const version).

If the key exists in the hash table, this operator returns a constant reference to the existing value. This version of the operator is used when the hash table object itself is constant, preventing accidental modification of its contents. If the key is not found, it throws a `KeyNotFoundException`.

Parameters

key	The key whose associated value is to be accessed.

Returns

A const reference to the value associated with the key.

Exceptions

KeyNotFoundException	If the key does not exist in the hash table.

Note

This operator assumes the existence of a `findPairIterator` member function (or a `const` overloaded version of it) that returns a `FindResult` struct/object with `wasElementFound()` and `iterator` (an iterator to the found element).

Implements IDictionary< Key, Value >.

4.5.2.8 operator[]() [2/2]

```
template<typename Key , typename Value , typename Hash >
Value & ChainedHashTable< Key, Value, Hash >::operator[] (
            const Key & key )  [override], [virtual]
```

Provides read/write access to the value associated with the given key.

If the key already exists in the hash table, this operator returns a reference to the existing value, allowing it to be modified. If the key does not exist, a new key-value pair is inserted into the hash table with the provided key and a default-constructed value, and a reference to this new value is returned.

Parameters

key	The key whose associated value is to be accessed or inserted.

Returns

A reference to the value associated with the key.

Note

This operator modifies the hash table if the key does not exist. It assumes the existence of `findPair↩Iterator` member function that returns a `FindResult` struct/object with `wasElementFound()` and `bucketRef` (a reference to the bucket list/vector) and `iterator` (an iterator to the found element). It also assumes `Value` is default-constructible.

Implements IDictionary< Key, Value >.

4.5.2.9 printInOrder()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::printInOrder (
            std::ostream & out ) const  [override], [virtual]
```

Prints the key-value pairs in the hash table to the output stream in ascending order of keys.

This function iterates through all elements in the hash table, stores them in a temporary vector, sorts the vector based on the keys, and then prints each key-value pair to the provided output stream. The output is formatted such that keys and values are right-aligned within a field whose width is determined by the maximum length of the keys and values, respectively, plus 2 for padding.

Parameters

out	The output stream to which the key-value pairs will be printed. Typically `std::cout` or a file stream.

Note

This function requires `StringHandler::toString()` to be defined for `Key` and `Value` types to correctly calculate string lengths for formatting. It also assumes that the `Key` type supports the less-than operator (<) for sorting.

Implements IDictionary< Key, Value >.

4.5.2.10 rehash()

```
template<typename Key , typename Value , typename Hash >
```

```
void ChainedHashTable< Key, Value, Hash >::rehash (
            size_t m )
```

Rehashes the hash table to a new size.

This function resizes the hash table to a new size that is the next prime number greater than or equal to the specified size m. It redistributes all existing key-value pairs into the new table, ensuring that the hash table maintains its integrity and performance.

Parameters

m	The minimum size for the new hash table. The actual size will be the next prime number greater than or equal to m.

4.5.2.11 remove()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::remove (
            const Key & key )  [override], [virtual]
```

Removes the key-value pair associated with the given key from the hash table.

If the key exists in the hash table, the corresponding key-value pair is removed. If the key does not exist, no action is taken.

Parameters

key	The key of the key-value pair to be removed.

Exceptions

None	

Implements IDictionary< Key, Value >.

4.5.2.12 update()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::update (
            const Key & key,
            const Value & value )  [override], [virtual]
```

Updates the value associated with a given key in the hash table.

If the key exists in the hash table, its associated value is updated to the provided value. If the key does not exist, a KeyNotFoundException is thrown.

Parameters

key	The key whose associated value is to be updated.
value	The new value to associate with the given key.

Exceptions

KeyNotFoundException	If the key does not exist in the hash table.

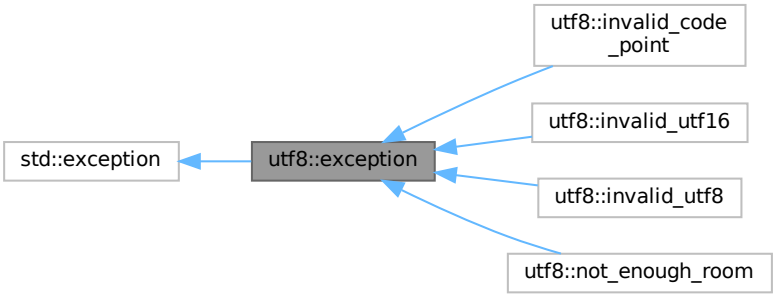
Implements IDictionary< Key, Value >.

The documentation for this class was generated from the following files:

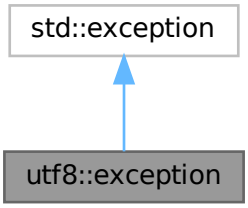
- include/HashTables/Chained/ChainedHashTable.hpp
- include/HashTables/Chained/ChainedHashTable.impl.hpp

4.6 utf8::exception Class Reference

Inheritance diagram for utf8::exception:



Collaboration diagram for utf8::exception:



The documentation for this class was generated from the following file:

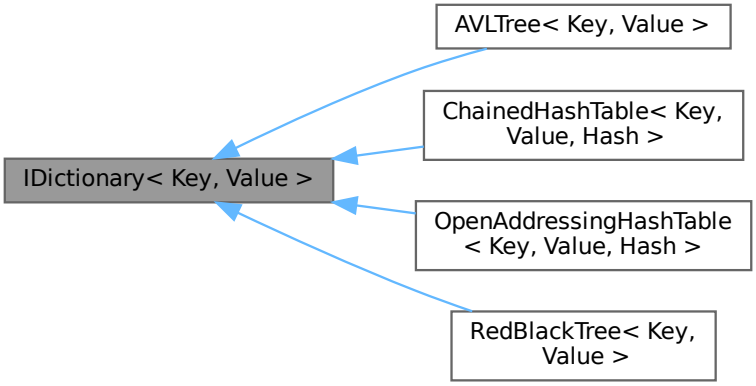
- include/utf8/checked.h

4.7 IDictionary< Key, Value > Class Template Reference

Interface for a generic dictionary data structure.

`#include <IDictionary.hpp>`

Inheritance diagram for IDictionary< Key, Value >:



Public Member Functions

- virtual void insert (const Key &key, const Value &value)=0
  Inserts a key-value pair into the dictionary.
- virtual bool find (const Key &key, Value &outValue) const =0
  Searches for a key in the dictionary.
- virtual void update (const Key &key, const Value &value)=0
  Updates the value associated with an existing key.
- virtual void remove (const Key &key)=0
  Removes a key and its associated value from the dictionary.
- virtual void clear ()=0
  Removes all entries from the dictionary.
- virtual void printInOrder (std::ostream &out) const =0
  Prints the contents of the dictionary in order.
- virtual size_t getComparisonsCount () const =0
  Retrieves the number of comparisons made in the last operation.
- virtual Value & operator[ ] (const Key &key)=0
  Provides access to the value associated with a key (modifiable).
- virtual const Value & operator[ ] (const Key &key) const =0
  Provides access to the value associated with a key (read-only).
- virtual ~**IDictionary** ()=default
  Virtual destructor.

Protected Member Functions

- void incrementCounter (size_t n) const

  Increments the number of comparisons by a given amount.
- void **resetCounter** () const

  Resets the comparisons counter to zero.

Protected Attributes

- size_t comparisonsCount = 0

  Tracks the number of comparisons made during dictionary operations.

Friends

- template<typename Tree , typename Node , typename K , typename V >
  class **BaseTree**
- template<typename HashTable , typename Collection , typename K , typename V , typename Hash >
  class **BaseHashTable**

4.7.1 Detailed Description

template<typename Key, typename Value>
class IDictionary< Key, Value >

Interface for a generic dictionary data structure.

This interface defines the basic operations for a dictionary, including insertion, search, update, removal, and traversal. It also provides functionality to track the number of comparisons made during operations.

Template Parameters

Key	The type of the keys used in the dictionary.
Value	The type of the values stored in the dictionary.

4.7.2 Member Function Documentation

4.7.2.1 clear()

```
template<typename Key , typename Value >
virtual void IDictionary< Key, Value >::clear ( )  [pure virtual]
```

Removes all entries from the dictionary.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H
and AVLTree< Key, Value >.

4.7.2.2 find()

```
template<typename Key , typename Value >
virtual bool IDictionary< Key, Value >::find (
            const Key & key,
            Value & outValue ) const  [pure virtual]
```

Searches for a key in the dictionary.

Parameters

key	The key to find.
outValue	The value associated with the key, if found.

Returns

true if the key is found; false otherwise.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.3 getComparisonsCount()

```
template<typename Key , typename Value >
virtual size_t IDictionary< Key, Value >::getComparisonsCount ( ) const  [pure virtual]
```

Retrieves the number of comparisons made in the last operation.

Returns

The number of comparisons.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.4 incrementCounter()

```
template<typename Key , typename Value >
void IDictionary< Key, Value >::incrementCounter (
            size_t n ) const  [inline], [protected]
```

Increments the number of comparisons by a given amount.

Parameters

n	The number of comparisons to add.

4.7.2.5 insert()

```
template<typename Key , typename Value >
virtual void IDictionary< Key, Value >::insert (
            const Key & key,
            const Value & value )  [pure virtual]
```

Inserts a key-value pair into the dictionary.

Parameters

key	The key to insert.
value	The value associated with the key.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.6 operator[]() [1/2]

```
template<typename Key , typename Value >
virtual const Value & IDictionary< Key, Value >::operator[] (
            const Key & key ) const  [pure virtual]
```

Provides access to the value associated with a key (read-only).

Parameters

key	The key to access.

Returns

A const reference to the value.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.7 operator[]() [2/2]

```
template<typename Key , typename Value >
virtual Value & IDictionary< Key, Value >::operator[] (
            const Key & key )  [pure virtual]
```

Provides access to the value associated with a key (modifiable).

Parameters

key	The key to access.

Returns

A reference to the value.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.8 printInOrder()

```
template<typename Key , typename Value >
virtual void IDictionary< Key, Value >::printInOrder (
            std::ostream & out ) const  [pure virtual]
```

Prints the contents of the dictionary in order.

Parameters

out	The output stream to print to.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.9 remove()

```
template<typename Key , typename Value >
virtual void IDictionary< Key, Value >::remove (
            const Key & key )  [pure virtual]
```

Removes a key and its associated value from the dictionary.

Parameters

key	The key to remove.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.2.10 update()

```
template<typename Key , typename Value >
virtual void IDictionary< Key, Value >::update (
            const Key & key,
            const Value & value )  [pure virtual]
```

Updates the value associated with an existing key.

Parameters

key	The key to update.
value	The new value to associate with the key.

Implemented in OpenAddressingHashTable< Key, Value, Hash >, RedBlackTree< Key, Value >, ChainedHashTable< Key, Value, H and AVLTree< Key, Value >.

4.7.3 Member Data Documentation

4.7.3.1 comparisonsCount

```
template<typename Key , typename Value >
size_t IDictionary< Key, Value >::comparisonsCount = 0  [mutable], [protected]
```

Tracks the number of comparisons made during dictionary operations.

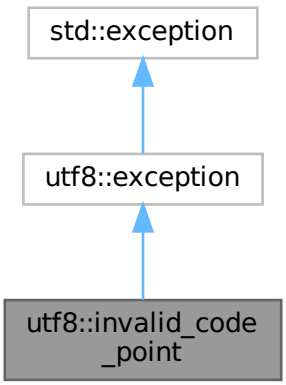
This mutable member variable is used to count the number of comparisons performed, allowing for performance analysis or debugging. Being mutable allows it to be modified even in const member functions.

The documentation for this class was generated from the following file:

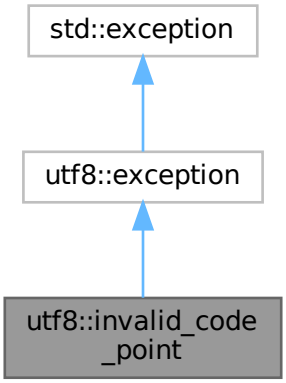
- include/Dictionary/IDictionary.hpp

4.8 utf8::invalid_code_point Class Reference

Inheritance diagram for utf8::invalid_code_point:



Collaboration diagram for utf8::invalid_code_point:



Public Member Functions

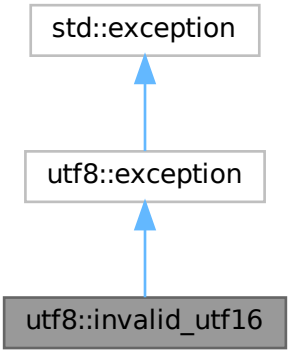
- **invalid_code_point** (utfchar32_t codepoint)
- virtual const char * **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE
- utfchar32_t **code_point** () const

The documentation for this class was generated from the following file:

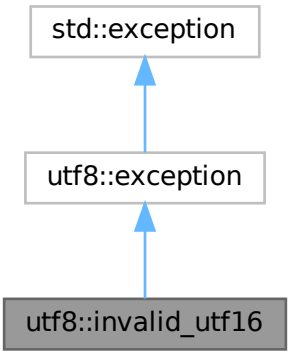
- include/utf8/checked.h

4.9 utf8::invalid_utf16 Class Reference

Inheritance diagram for utf8::invalid_utf16:



Collaboration diagram for utf8::invalid_utf16:



Public Member Functions

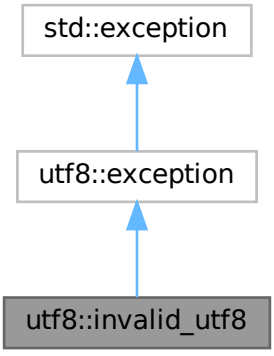
- **invalid_utf16** (utfchar16_t u)
- virtual const char * **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE
- utfchar16_t **utf16_word** () const

The documentation for this class was generated from the following file:

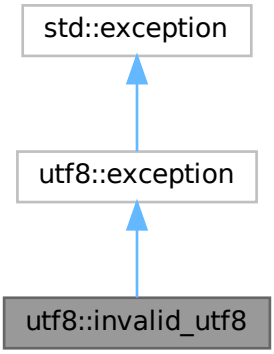
- include/utf8/checked.h

4.10 utf8::invalid_utf8 Class Reference

Inheritance diagram for utf8::invalid_utf8:



Collaboration diagram for utf8::invalid_utf8:



Public Member Functions

- **invalid_utf8** (utfchar8_t u)
- **invalid_utf8** (char c)
- virtual const char * **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE
- utfchar8_t **utf8_octet** () const

The documentation for this class was generated from the following file:

- include/utf8/checked.h

4.11 utf8::iterator< octet_iterator > Class Template Reference

Public Types

- typedef utfchar32_t **value_type**
- typedef utfchar32_t * **pointer**
- typedef utfchar32_t & **reference**
- typedef std::ptrdiff_t **difference_type**
- typedef std::bidirectional_iterator_tag **iterator_category**

Public Member Functions

- **iterator** (const octet_iterator &octet_it, const octet_iterator &rangestart, const octet_iterator &rangeend)
- octet_iterator **base** () const
- utfchar32_t **operator*** () const
- bool **operator==** (const iterator &rhs) const
- bool **operator!=** (const iterator &rhs) const
- iterator & **operator++** ()
- iterator **operator++** (int)
- iterator & **operator--** ()
- iterator **operator--** (int)

The documentation for this class was generated from the following file:

- include/utf8/checked.h

4.12 utf8::unchecked::iterator< octet_iterator > Class Template Reference

Public Types

- typedef utfchar32_t **value_type**
- typedef utfchar32_t * **pointer**
- typedef utfchar32_t & **reference**
- typedef std::ptrdiff_t **difference_type**
- typedef std::bidirectional_iterator_tag **iterator_category**

Public Member Functions

- **iterator** (const octet_iterator &octet_it)
- octet_iterator **base** () const
- utfchar32_t **operator*** () const
- bool **operator==** (const iterator &rhs) const
- bool **operator!=** (const iterator &rhs) const
- iterator & **operator++** ()
- iterator **operator++** (int)
- iterator & **operator--** ()
- iterator **operator--** (int)

The documentation for this class was generated from the following file:

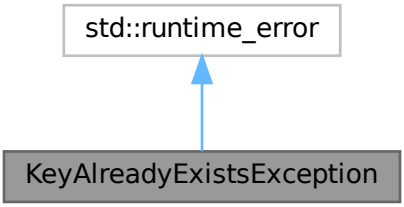
- include/utf8/unchecked.h

4.13 KeyAlreadyExistsException Class Reference

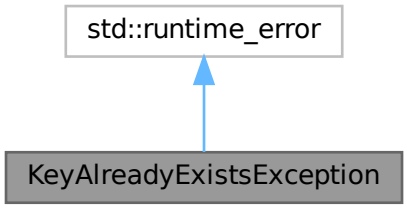
Exception thrown when attempting to insert a key that already exists in a dictionary.

`#include <KeyExceptions.hpp>`

Inheritance diagram for KeyAlreadyExistsException:



Collaboration diagram for KeyAlreadyExistsException:



4.13.1 Detailed Description

Exception thrown when attempting to insert a key that already exists in a dictionary.

This exception is derived from std::runtime_error and is used to indicate that a key being inserted into a dictionary or similar data structure already exists.

Example usage:

```
try {
    dictionary.insert(key, value);
} catch (const KeyAlreadyExistsException& e) {
    std::cerr « e.what() « std::endl;
}
```

The documentation for this class was generated from the following file:

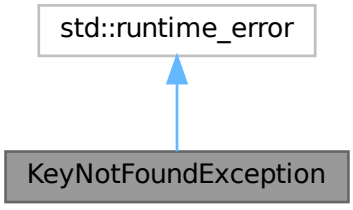
- include/Exceptions/KeyExceptions.hpp

4.14 KeyNotFoundException Class Reference

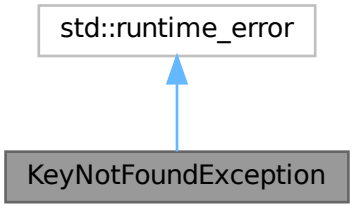
Exception thrown when a key is not found in a dictionary or map.

`#include <KeyExceptions.hpp>`

Inheritance diagram for KeyNotFoundException:



Collaboration diagram for KeyNotFoundException:



4.14.1 Detailed Description

Exception thrown when a key is not found in a dictionary or map.

This exception is derived from std::runtime_error and is used to indicate that an operation attempted to access a key that does not exist in the dictionary or map.

Example usage:

```
try {
    throw KeyNotFoundException();
} catch (const KeyNotFoundException& e) {
    std::cerr « e.what() « std::endl;
}
```

The documentation for this class was generated from the following file:

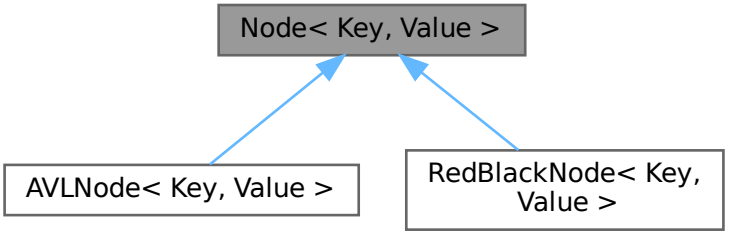
- include/Exceptions/KeyExceptions.hpp

4.15 Node< Key, Value > Class Template Reference

Represents a basic node that stores a key-value pair.

`#include <Node.hpp>`

Inheritance diagram for Node< Key, Value >:



Public Member Functions

- Node (const Key &key, const Value &value)

  Constructs a Node with a given key and value.
- const Key & getKey () const

  Gets the key stored in the node.
- void setKey (const Key &key)

  Sets the key in the node.
- const Value & getValue () const

  Gets the value stored in the node (read-only).
- Value & getValue ()

  Gets the value stored in the node (modifiable).
- void setValue (const Value &value)

  Sets the value in the node.
- void update (const Key &key, const Value &value)

  Updates both the key and the value of the node.
- std::string show () const

  Returns a string representation of the node as (key, value).
- ~**Node** ()=default

  Default destructor.

4.15.1 Detailed Description

template<typename Key, typename Value>
class Node< Key, Value >

Represents a basic node that stores a key-value pair.

Template Parameters

Key	The type of the key.
Value	The type of the value.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Node()

```
template<typename Key , typename Value >
Node< Key, Value >::Node (
            const Key & key,
            const Value & value )  [inline]
```

Constructs a Node with a given key and value.

Parameters

key	The key to store.
value	The value associated with the key.

4.15.3 Member Function Documentation

4.15.3.1 getKey()

```
template<typename Key , typename Value >
const Key & Node< Key, Value >::getKey ( ) const  [inline]
```

Gets the key stored in the node.

Returns

A constant reference to the key.

4.15.3.2 getValue() [1/2]

```
template<typename Key , typename Value >
Value & Node< Key, Value >::getValue ( )  [inline]
```

Gets the value stored in the node (modifiable).

Returns

A reference to the value.

4.15.3.3 getValue() [2/2]

```
template<typename Key , typename Value >
const Value & Node< Key, Value >::getValue ( ) const  [inline]
```

Gets the value stored in the node (read-only).

Returns

A constant reference to the value.

4.15.3.4 setKey()

```
template<typename Key , typename Value >
void Node< Key, Value >::setKey (
            const Key & key )  [inline]
```

Sets the key in the node.

Parameters

key	The new key to assign.

4.15.3.5 setValue()

```
template<typename Key , typename Value >
void Node< Key, Value >::setValue (
            const Value & value )  [inline]
```

Sets the value in the node.

Parameters

value	The new value to assign.

4.15.3.6 show()

```
template<typename Key , typename Value >
std::string Node< Key, Value >::show ( ) const  [inline]
```

Returns a string representation of the node as (key, value).

Returns

A formatted string showing the key and value.

4.15.3.7 update()

```
template<typename Key , typename Value >
void Node< Key, Value >::update (
            const Key & key,
            const Value & value )  [inline]
```

Updates both the key and the value of the node.

Parameters

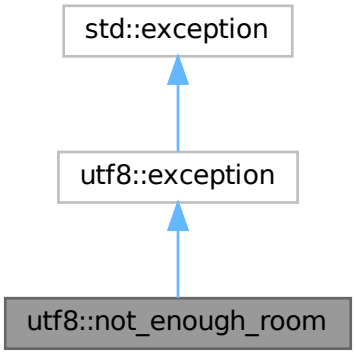
key	The new key to assign.
value	The new value to assign.

The documentation for this class was generated from the following file:

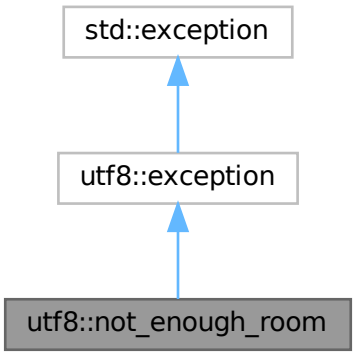
- include/Trees/Base/Node.hpp

4.16 utf8::not_enough_room Class Reference

Inheritance diagram for utf8::not_enough_room:



Collaboration diagram for utf8::not_enough_room:



Public Member Functions

- virtual const char * **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE

The documentation for this class was generated from the following file:

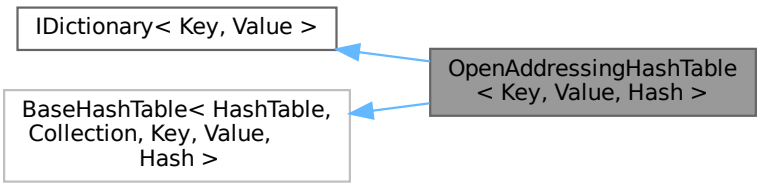
- include/utf8/checked.h

4.17 OpenAddressingHashTable< Key, Value, Hash > Class Template Reference

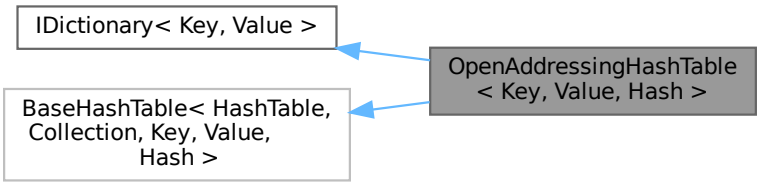
Hash table implementation using open addressing.

`#include <OpenAddressingHashTable.hpp>`

Inheritance diagram for OpenAddressingHashTable< Key, Value, Hash >:



Collaboration diagram for OpenAddressingHashTable< Key, Value, Hash >:



Public Member Functions

- OpenAddressingHashTable (size_t size=8, float mlf=0.7)

  Constructs an OpenAddressingHashTable with a specified initial size and maximum load factor.
- void insert (const Key &key, const Value &value)

  Inserts a key-value pair into the hash table.
- bool find (const Key &key, Value &outValue) const

  Searches for a key in the hash table and retrieves its associated value if found.
- void update (const Key &key, const Value &value)

  Updates the value associated with the given key in the hash table.
- void remove (const Key &key)

  Removes the element associated with the given key from the hash table.
- void clear ()

  Clears the hash table by removing all elements and resetting its state.
- void printInOrder (std::ostream &out) const

  Prints the contents of the hash table in order of keys.

- size_t getComparisonsCount () const

  Retrieves the total number of comparisons made during hash table operations.
- Value & operator[ ] (const Key &key)

  Accesses or inserts a value associated with the given key.
- const Value & operator[ ] (const Key &key) const

  Accesses the value associated with the given key in the hash table.
- void rehash (size_t m)

  Resizes the hash table to a new size and rehashes all existing elements.
- size_t getCollisionsCount () const

  Retrieves the total number of collisions that have occurred in the hash table.
- size_t getTableSize () const

  Retrieves the size of the hash table.

Public Member Functions inherited from IDictionary< Key, Value >

- virtual ~**IDictionary** ()=default

  Virtual destructor.

Public Member Functions inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- BaseHashTable (size_t size=7, float mlf=0.7)

  Constructs a new BaseHashTable object.
- float getLoadFactor () const

  Calculates and returns the current load factor of the hash table.
- void clearHashTable ()

  Clears all elements from the hash table, resetting it to an empty state.
- void incrementCollisionsCount (size_t m=1) const

  Increments the count of collisions in the hash table.

Additional Inherited Members

Protected Member Functions inherited from IDictionary< Key, Value >

- void incrementCounter (size_t n) const

  Increments the number of comparisons by a given amount.
- void **resetCounter** () const

  Resets the comparisons counter to zero.

Protected Member Functions inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- size_t getNextPrime (size_t num) const

  Calculates and returns the next prime number greater than or equal to a given number.
- void checkAndRehash ()

  Checks the current load factor and triggers a rehash if necessary.

Protected Attributes inherited from IDictionary< Key, Value >

- size_t comparisonsCount = 0

  Tracks the number of comparisons made during dictionary operations.

Protected Attributes inherited from BaseHashTable< HashTable, Collection, Key, Value, Hash >

- std::vector< Collection > **table**

  The hash table's internal storage, composed of collections (e.g., lists or buckets).
- size_t **tableSize**

  The current size of the hash table (number of slots).
- float **maxLoadFactor**

  The maximum load factor before the table is resized.
- size_t **numberOfElements**

  The number of elements currently stored in the table.
- Hash **hashing**

  Hash function object used to compute the index for each key.
- size_t **collisionsCount**

  Number of collisions occurred during insertions and searches.

4.17.1 Detailed Description

template<typename Key, typename Value, typename Hash = std::hash<Key>>
class OpenAddressingHashTable< Key, Value, Hash >

Hash table implementation using open addressing.

Template Parameters

Key	The type of the keys.
Value	The type of the values.
Hash	The hash function to be used (defaults to std::hash<Key>).

4.17.2 Constructor & Destructor Documentation

4.17.2.1 OpenAddressingHashTable()

```
template<typename Key , typename Value , typename Hash >
OpenAddressingHashTable< Key, Value, Hash >::OpenAddressingHashTable (
            size_t size = 8,
            float mlf = 0.7 )
```

Constructs an OpenAddressingHashTable with a specified initial size and maximum load factor.

Parameters

size	The initial size of the hash table. Defaults to 8 if not specified.
mlf	The maximum load factor (a value between 0 and 1) that determines when the table should be rehashed. Defaults to 0.7 if not specified.

4.17.3 Member Function Documentation

4.17.3.1 clear()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::clear ( )  [virtual]
```

Clears the hash table by removing all elements and resetting its state.

This function removes all active elements from the hash table and resets the internal data structure to its initial state. After calling this function, the hash table will be empty, and all slots will be marked as empty.

Implements IDictionary< Key, Value >.

4.17.3.2 find()

```
template<typename Key , typename Value , typename Hash >
bool OpenAddressingHashTable< Key, Value, Hash >::find (
            const Key & key,
            Value & outValue ) const  [virtual]
```

Searches for a key in the hash table and retrieves its associated value if found.

Parameters

key	The key to search for in the hash table.
outValue	A reference to a variable where the value associated with the key will be stored if found.

Returns

true If the key is found in the hash table.

false If the key is not found in the hash table.

Exceptions

None	

Implements IDictionary< Key, Value >.

4.17.3.3 getCollisionsCount()

```
template<typename Key , typename Value , typename Hash >
size_t OpenAddressingHashTable< Key, Value, Hash >::getCollisionsCount ( ) const
```

Retrieves the total number of collisions that have occurred in the hash table.

A collision occurs when two different keys are hashed to the same index in the table. This function provides a count of such collisions, which can be useful for analyzing the efficiency of the hash function and the overall performance of the hash table.

Returns

size_t The number of collisions that have occurred.

4.17.3.4 getComparisonsCount()

```
template<typename Key , typename Value , typename Hash >
size_t OpenAddressingHashTable< Key, Value, Hash >::getComparisonsCount ( ) const  [virtual]
```

Retrieves the total number of comparisons made during hash table operations.

This method returns the count of comparisons performed while searching for, inserting, or updating elements in the hash table. It is useful for analyzing the efficiency of the hash table operations.

Returns

size_t The number of comparisons made.

Implements IDictionary< Key, Value >.

4.17.3.5 getTableSize()

```
template<typename Key , typename Value , typename Hash >
size_t OpenAddressingHashTable< Key, Value, Hash >::getTableSize ( ) const
```

Retrieves the size of the hash table.

This function returns the total number of slots available in the hash table, which represents its capacity. It does not indicate the number of elements currently stored in the table.

Returns

size_t The total number of slots in the hash table.

4.17.3.6 insert()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::insert (
            const Key & key,
            const Value & value )  [virtual]
```

Inserts a key-value pair into the hash table.

This function attempts to insert the given key and value into the hash table. If the key already exists in the table, a KeyAlreadyExistsException is thrown. If the table is full or requires rehashing, the function will handle rehashing before proceeding with the insertion.

Parameters

key	The key to be inserted into the hash table.
value	The value associated with the key to be inserted.

Exceptions

KeyAlreadyExistsException	If the key already exists in the hash table.

Implements IDictionary< Key, Value >.

4.17.3.7 operator[]() [1/2]

```
template<typename Key , typename Value , typename Hash >
Value & OpenAddressingHashTable< Key, Value, Hash >::operator[] (
            const Key & key )  [virtual]
```

Accesses or inserts a value associated with the given key.

If the key exists in the hash table, this operator returns a reference to the associated value. If the key does not exist, a new entry is created with the given key and a default-constructed value, and a reference to the newly created value is returned.

Parameters

key	The key to search for or insert into the hash table.

Returns

Value& A reference to the value associated with the given key.

Note

This function may trigger a rehash if the load factor exceeds the maximum load factor.

Exceptions

std::bad_alloc	If memory allocation fails during rehashing or insertion.

Implements IDictionary< Key, Value >.

4.17.3.8 operator[]() [2/2]

```
template<typename Key , typename Value , typename Hash >
const Value & OpenAddressingHashTable< Key, Value, Hash >::operator[] (
            const Key & key ) const  [virtual]
```

Accesses the value associated with the given key in the hash table.

This operator provides read-only access to the value corresponding to the specified key. If the key is not found in the hash table, a KeyNotFoundException is thrown.

Parameters

key	The key whose associated value is to be accessed.

Returns

const Value& A constant reference to the value associated with the key.

Exceptions

KeyNotFoundException	If the key is not found in the hash table.

Implements IDictionary< Key, Value >.

4.17.3.9 printInOrder()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::printInOrder (
            std::ostream & out ) const  [virtual]
```

Prints the contents of the hash table in order of keys.

This function iterates through all active slots in the hash table, collects them into a vector, and sorts them by their keys. It then outputs the key-value pairs in a formatted manner to the provided output stream.

Parameters

out	The output stream where the formatted key-value pairs will be printed.

Implements IDictionary< Key, Value >.

4.17.3.10 rehash()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::rehash (
            size_t m )
```

Resizes the hash table to a new size and rehashes all existing elements.

This function increases the size of the hash table to the specified value m (if m is greater than the current table size) and rehashes all active elements into the new table. The rehashing process ensures that the hash table maintains its integrity and performance after resizing.

Parameters

m	The new size of the hash table. Must be greater than the current table size.

4.17.3.11 remove()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::remove (
            const Key & key )  [virtual]
```

Removes the element associated with the given key from the hash table.

If the key is found in the hash table, the corresponding slot's status is marked as DELETED. If the key is not found, the function does nothing.

Parameters

key	The key of the element to be removed.

Implements IDictionary< Key, Value >.

4.17.3.12 update()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::update (
            const Key & key,
            const Value & value )  [virtual]
```

Updates the value associated with the given key in the hash table.

If the key exists in the hash table, its associated value is updated to the provided value. If the key does not exist, a KeyNotFoundException is thrown.

Parameters

key	The key whose associated value is to be updated.
value	The new value to associate with the given key.

Exceptions

KeyNotFoundException	If the key is not found in the hash table.

Implements IDictionary< Key, Value >.

The documentation for this class was generated from the following files:

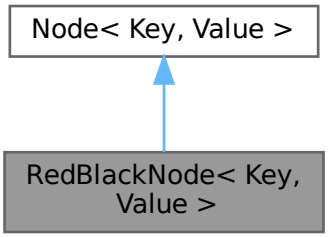
- include/HashTables/OpenAddressing/OpenAddressingHashTable.hpp
- include/HashTables/OpenAddressing/OpenAddressingHashTable.impl.hpp

4.18 RedBlackNode< Key, Value > Class Template Reference

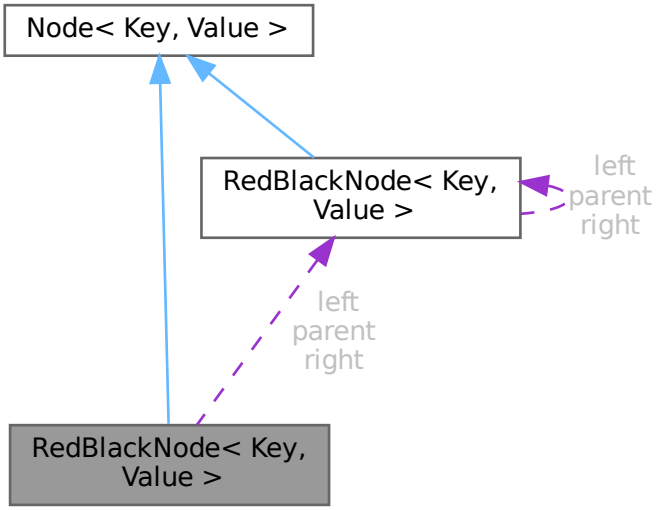
Represents a node in a Red-Black Tree.

`#include <RedBlackNode.hpp>`

Inheritance diagram for RedBlackNode< Key, Value >:



Collaboration diagram for RedBlackNode< Key, Value >:



Public Member Functions

- RedBlackNode (const Key &k, const Value &v, RedBlackNode *l, RedBlackNode *r, RedBlackNode *p, Color c)

  Constructs a RedBlackNode.
- **RedBlackNode** (Color color=BLACK)

Public Member Functions inherited from Node< Key, Value >

- Node (const Key &key, const Value &value)

  Constructs a Node with a given key and value.
- const Key & getKey () const

  Gets the key stored in the node.
- void setKey (const Key &key)

  Sets the key in the node.
- const Value & getValue () const

  Gets the value stored in the node (read-only).
- Value & getValue ()

  Gets the value stored in the node (modifiable).
- void setValue (const Value &value)

  Sets the value in the node.
- void update (const Key &key, const Value &value)

  Updates both the key and the value of the node.
- std::string show () const

  Returns a string representation of the node as (key, value).
- ~**Node** ()=default

  Default destructor.

Public Attributes

- RedBlackNode * left

  Pointer to the left child node in the Red-Black Tree.
- RedBlackNode * right

  Pointer to the right child node in the Red-Black Tree.
- RedBlackNode * parent

  Pointer to the parent node in the Red-Black Tree.
- Color color

  Represents the color of a node in a Red-Black Tree.

4.18.1 Detailed Description

template<typename Key, typename Value>
class RedBlackNode< Key, Value >

Represents a node in a Red-Black Tree.

This class extends the generic Node class and includes additional properties specific to Red-Black Trees, such as color and self-referencing pointers for left, right, and parent nodes.

Parameters

color	The color of the node, either RED or BLACK. Defaults to BLACK.

The constructor initializes the node with default key and value, and sets the left, right, and parent pointers to point to itself. This is useful for representing sentinel nodes (e.g., NIL nodes) in a Red-Black Tree.

4.18.2 Constructor & Destructor Documentation

4.18.2.1 RedBlackNode()

```
template<typename Key , typename Value >
RedBlackNode< Key, Value >::RedBlackNode (
            const Key & k,
            const Value & v,
            RedBlackNode< Key, Value > * l,
            RedBlackNode< Key, Value > * r,
            RedBlackNode< Key, Value > * p,
            Color c )  [inline]
```

Constructs a RedBlackNode.

Parameters

k	The key for the node.
v	The value for the node.
l	Pointer to the left child node.
r	Pointer to the right child node.
p	Pointer to the parent node.
c	The color of the node (Red or Black).

4.18.3 Member Data Documentation

4.18.3.1 color

```
template<typename Key , typename Value >
Color RedBlackNode< Key, Value >::color
```

Represents the color of a node in a Red-Black Tree.

The color can typically be either RED or BLACK, and it is used to maintain the balancing properties of the Red-Black Tree.

4.18.3.2 left

```
template<typename Key , typename Value >
RedBlackNode* RedBlackNode< Key, Value >::left
```

Pointer to the left child node in the Red-Black Tree.

This pointer references the left child of the current node. It is used to traverse the tree structure and maintain the Red-Black Tree properties.

4.18.3.3 parent

```
template<typename Key , typename Value >
RedBlackNode* RedBlackNode< Key, Value >::parent
```

Pointer to the parent node in the Red-Black Tree.

This pointer is used to maintain the hierarchical relationship between nodes in the Red-Black Tree. It points to the parent of the current node, or is set to nullptr if the current node is the root of the tree.

4.18.3.4 right

```
template<typename Key , typename Value >
RedBlackNode* RedBlackNode< Key, Value >::right
```

Pointer to the right child node in the Red-Black Tree.

This pointer references the right child of the current node. It is used to traverse or manipulate the right subtree of the Red-Black Tree.

The documentation for this class was generated from the following file:

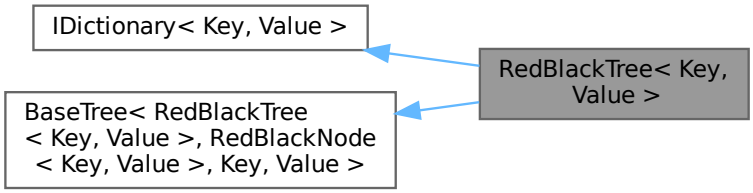
- include/Trees/RedBlack/RedBlackNode.hpp

4.19 RedBlackTree< Key, Value > Class Template Reference

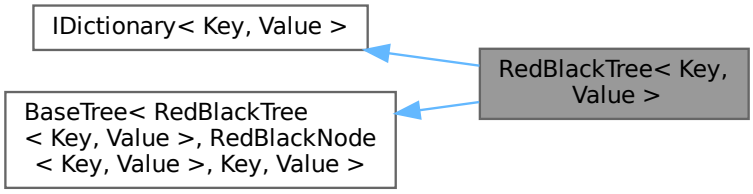
A class representing a Red-Black Tree.

```
#include <RedBlackTree.hpp>
```

Inheritance diagram for RedBlackTree< Key, Value >:



Collaboration diagram for RedBlackTree< Key, Value >:



Public Member Functions

- **RedBlackTree** ()

  Constructs an empty Red-Black Tree.
- void insert (const Key &key, const Value &value)

  Inserts a key-value pair into the Red-Black Tree.
- bool find (const Key &key, Value &outValue) const

  Searches for a key in the Red-Black Tree and retrieves its associated value.
- void update (const Key &key, const Value &value)

  Updates the value associated with a given key in the Red-Black Tree.
- void remove (const Key &key)

  Removes a node with the specified key from the Red-Black Tree.
- void clear ()

  Clears the Red-Black Tree by deallocating all nodes.
- void printInOrder (std::ostream &out) const

  Prints the elements of the Red-Black Tree in in-order traversal.
- size_t getComparisonsCount () const

  Retrieves the count of comparisons made during operations on the Red-Black Tree.
- virtual Value & operator[ ] (const Key &key)

  Accesses the value associated with a given key.
- virtual const Value & operator[ ] (const Key &key) const

  Accesses the value associated with a given key (const version).
- void **print** () const

  Prints the structure of the Red-Black Tree.
- size_t getRotationsCount () const

  Retrieves the total number of rotations performed by the Red-Black Tree.

Public Member Functions inherited from IDictionary< Key, Value >

- virtual ~**IDictionary** ()=default

  Virtual destructor.

Additional Inherited Members

Protected Member Functions inherited from IDictionary< Key, Value >

- void incrementCounter (size_t n) const

  Increments the number of comparisons by a given amount.
- void **resetCounter** () const

  Resets the comparisons counter to zero.

Protected Member Functions inherited from BaseTree< RedBlackTree< Key, Value >, RedBlackNode< Key, Value >, Key, Value >

- BaseTree (RedBlackNode< Key, Value > *r)

  Constructs a BaseTree with the given root node.
- const RedBlackNode< Key, Value > * findNode (const Key &key, RedBlackNode< Key, Value > *comp=nullptr) const

  Finds a node with the specified key in the tree.
- RedBlackNode< Key, Value > * minimum (RedBlackNode< Key, Value > *node) const

  Finds the node with the minimum key in the subtree rooted at the given node.
- void clearNode (RedBlackNode< Key, Value > *node, RedBlackNode< Key, Value > *comp)

  Recursively clears (deletes) nodes in a subtree, avoiding a comparison node.
- void reset (RedBlackNode< Key, Value > *node, RedBlackNode< Key, Value > *comp=nullptr, RedBlackNode< Key, Value > *defaultRoot=nullptr)

  Resets the tree by clearing all nodes starting from the given node, except for the comparison node, and sets the root to the default root.
- void inOrderTransversal (std::ostream &out, RedBlackNode< Key, Value > *node, RedBlackNode< Key, Value > *comp) const

  Performs an in-order traversal of the subtree and prints node information to an output stream.
- const Value & at (const Key &key, RedBlackNode< Key, Value > *comp=nullptr) const

  Accesses the value associated with a given key.
- void setMaxKeyLen (const Key &key)

  Updates the maximum key length stored in the tree.
- void setMaxValLen (const Value &value)

  Updates the maximum length of the value in the tree.
- void incrementRotationsCount (size_t amount=1)

  Increments the count of rotations performed on the tree.

Protected Attributes inherited from IDictionary< Key, Value >

- size_t comparisonsCount = 0

  Tracks the number of comparisons made during dictionary operations.

Protected Attributes inherited from BaseTree< RedBlackTree< Key, Value >, RedBlackNode< Key, Value >, Key, Value >

- RedBlackNode< Key, Value > * root

  Pointer to the root node of the tree.
- size_t **maxKeyLen**

  Represents the maximum length of a key that can be stored in the tree.
- size_t maxValLen

  Represents the maximum length of a value in the tree.
- size_t rotationsCount

  Tracks the number of rotations performed in the tree.

4.19.1 Detailed Description

template<typename Key, typename Value>
class RedBlackTree< Key, Value >

A class representing a Red-Black Tree.

The RedBlackTree class implements a self-balancing binary search tree that maintains the Red-Black Tree properties. It supports operations such as insertion, deletion, and search while ensuring logarithmic time complexity.

Template Parameters

Key	The type of the keys stored in the tree.
Value	The type of the values associated with the keys.

4.19.2 Member Function Documentation

4.19.2.1 clear()

```
template<typename Key , typename Value >
void RedBlackTree< Key, Value >::clear ( )  [virtual]
```

Clears the Red-Black Tree by deallocating all nodes.

Implements IDictionary< Key, Value >.

4.19.2.2 find()

```
template<typename Key , typename Value >
bool RedBlackTree< Key, Value >::find (
            const Key & key,
            Value & outValue ) const  [virtual]
```

Searches for a key in the Red-Black Tree and retrieves its associated value.

Parameters

key	The key to search for.
outValue	A reference to store the associated value if the key is found.

Returns

true If the key is found.

false If the key is not found.

Implements IDictionary< Key, Value >.

4.19.2.3 getComparisonsCount()

```
template<typename Key , typename Value >
size_t RedBlackTree< Key, Value >::getComparisonsCount ( ) const  [virtual]
```

Retrieves the count of comparisons made during operations on the Red-Black Tree.

Returns

The total number of comparisons made.

Implements IDictionary< Key, Value >.

4.19.2.4 getRotationsCount()

```
template<typename Key , typename Value >
size_t RedBlackTree< Key, Value >::getRotationsCount ( ) const
```

Retrieves the total number of rotations performed by the Red-Black Tree.

This method returns the count of rotations (both left and right) that have been executed during insertions or deletions to maintain the Red-Black Tree properties.

Returns

size_t The total number of rotations performed.

4.19.2.5 insert()

```
template<typename Key , typename Value >
void RedBlackTree< Key, Value >::insert (
            const Key & key,
            const Value & value )  [virtual]
```

Inserts a key-value pair into the Red-Black Tree.

Parameters

key	The key to insert.
value	The value associated with the key.

Implements IDictionary< Key, Value >.

4.19.2.6 operator[]() [1/2]

```
template<typename Key , typename Value >
Value & RedBlackTree< Key, Value >::operator[] (
            const Key & key )  [virtual]
```

Accesses the value associated with a given key.

Parameters

key	The key to access.

Returns

A reference to the associated value.

Implements IDictionary< Key, Value >.

4.19.2.7 operator[]() [2/2]

```
template<typename Key , typename Value >
const Value & RedBlackTree< Key, Value >::operator[] (
            const Key & key ) const  [virtual]
```

Accesses the value associated with a given key (const version).

Parameters

key	The key to access.

Returns

A const reference to the associated value.

Implements IDictionary< Key, Value >.

4.19.2.8 printInOrder()

```
template<typename Key , typename Value >
void RedBlackTree< Key, Value >::printInOrder (
            std::ostream & out ) const  [virtual]
```

Prints the elements of the Red-Black Tree in in-order traversal.

Parameters

out	The output stream where the traversal result will be written.

Implements IDictionary< Key, Value >.

4.19.2.9 remove()

```
template<typename Key , typename Value >
void RedBlackTree< Key, Value >::remove (
            const Key & key )  [virtual]
```

Removes a node with the specified key from the Red-Black Tree.

Parameters

key	The key of the node to be removed.

Implements IDictionary< Key, Value >.

4.19.2.10 update()

```
template<typename Key , typename Value >
```

```
void RedBlackTree< Key, Value >::update (
            const Key & key,
            const Value & value )  [virtual]
```

Updates the value associated with a given key in the Red-Black Tree.

Parameters

key	The key to update.
value	The new value to associate with the key.

Exceptions

KeyNotFoundException	If the key is not found in the tree.

Implements IDictionary< Key, Value >.

The documentation for this class was generated from the following files:

- include/Trees/RedBlack/RedBlackTree.hpp
- include/Trees/RedBlack/RedBlackTree.impl.hpp

4.20 StringHandler::SetWidthAtLeft< Object > Struct Template Reference

A manipulator to set the width and left-align an object when streamed.

`#include <StringHandler.hpp>`

Public Member Functions

- SetWidthAtLeft (const Object &o, size_t w)

  Constructs a SetWidthAtLeft manipulator.

Public Attributes

- const Object & **obj**

  The object to be formatted.
- size_t **width**

  The desired total width for the formatted output.

4.20.1 Detailed Description

template<typename Object>
struct StringHandler::SetWidthAtLeft< Object >

A manipulator to set the width and left-align an object when streamed.

Template Parameters

Object	The type of the object to be formatted.

4.20.2 Constructor & Destructor Documentation

4.20.2.1 SetWidthAtLeft()

```
template<typename Object >
StringHandler::SetWidthAtLeft< Object >::SetWidthAtLeft (
            const Object & o,
            size_t w )  [inline]
```

Constructs a SetWidthAtLeft manipulator.

Parameters

o	The object to be formatted.
w	The desired total width for the formatted output.

The documentation for this struct was generated from the following file:

- include/Utils/StringHandler.hpp

4.21 Slot< Key, Value > Struct Template Reference

Represents a slot in an open addressing hash table.

`#include <Slot.hpp>`

Public Member Functions

- Slot ()
  Default constructor. Initializes the slot as EMPTY.
- Slot (const Key &k, const Value &v)
  Constructs a slot with a key and value. Sets status to ACTIVE.

Public Attributes

- Key key
  The key associated with this slot.
- Value value
  The value associated with the key.
- Status status
  The current status of the slot (e.g., EMPTY, ACTIVE, DELETED).

4.21.1 Detailed Description

template<typename Key, typename Value>
struct Slot< Key, Value >

Represents a slot in an open addressing hash table.

The Slot structure is used to store a key-value pair along with its status in an open addressing hash table. The status indicates whether the slot is empty, active, or deleted.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Slot() [1/2]

```
template<typename Key , typename Value >
Slot< Key, Value >::Slot ( )  [inline]
```

Default constructor. Initializes the slot as EMPTY.

Default constructor that initializes the slot with an EMPTY status.

4.21.2.2 Slot() [2/2]

```
template<typename Key , typename Value >
Slot< Key, Value >::Slot (
            const Key & k,
            const Value & v )  [inline]
```

Constructs a slot with a key and value. Sets status to ACTIVE.

Parameters

k	The key to store.
v	The value associated with the key.

Parameterized constructor that initializes the slot with a given key and value, and sets the status to ACTIVE.

Note

The `Key` and `Value` types, as well as the `Status` enumeration, are assumed to be defined elsewhere in the codebase.

4.21.3 Member Data Documentation

4.21.3.1 key

```
template<typename Key , typename Value >
Slot< Key, Value >::key
```

The key associated with this slot.

The key associated with the slot.

4.21.3.2 status

```
template<typename Key , typename Value >
Slot< Key, Value >::status
```

The current status of the slot (e.g., EMPTY, ACTIVE, DELETED).

The status of the slot, which can be EMPTY, ACTIVE, or DELETED.

4.21.3.3 value

```
template<typename Key , typename Value >
Slot< Key, Value >::value
```

The value associated with the key.

The value associated with the slot.

The documentation for this struct was generated from the following file:

- include/HashTables/OpenAddressing/Slot.hpp

Chapter 5

File Documentation

5.1 IDictionary.hpp

```
00001 #ifndef IDICTIONARY_HPP
00002 #define IDICTIONARY_HPP
00003
00004 #include "Trees/Base/BaseTree.hpp"
00005
00016 template<typename Key, typename Value>
00017 class IDictionary {
00018 protected:
00026     mutable size_t comparisonsCount = 0;
00027
00033     void incrementCounter(size_t n) const { comparisonsCount += n; }
00034
00038     void resetCounter() const { comparisonsCount = 0; }
00039
00040 public:
00047     virtual void insert(const Key& key, const Value& value) = 0;
00048
00056     virtual bool find(const Key& key, Value& outValue) const = 0;
00057
00064     virtual void update(const Key& key, const Value& value) = 0;
00065
00071     virtual void remove(const Key& key) = 0;
00072
00076     virtual void clear() = 0;
00077
00083     virtual void printInOrder(std::ostream& out) const = 0;
00084
00090     virtual size_t getComparisonsCount() const = 0;
00091
00098     virtual Value& operator[](const Key& key) = 0;
00099
00106     virtual const Value& operator[](const Key& key) const = 0;
00107
00111     virtual ~IDictionary() = default;
00112
00113     template <typename Tree, typename Node, typename K, typename V>
00114     friend class BaseTree;
00115
00116     template <typename HashTable, typename Collection, typename K, typename V, typename Hash>
00117     friend class BaseHashTable;
00118 };
00119
00120 #endif
```

5.2 KeyExceptions.hpp

```
00001 #ifndef KEY_EXCEPTIONS_HPP
00002 #define KEY_EXCEPTIONS_HPP
00003
00004 #include <stdexcept>
00005 #include <string>
00006
00024 class KeyAlreadyExistsException : public std::runtime_error {
00025 public:
```

```
00026     explicit KeyAlreadyExistsException()
00027         : std::runtime_error("Key already exists in the dictionary.") {}
00028 };
00029
00047 class KeyNotFoundException : public std::runtime_error {
00048 public:
00049     explicit KeyNotFoundException()
00050         : std::runtime_error("Key not found in the dictionary.") {}
00051 };
00052
00053 #endif
```

5.3 BaseHashTable.hpp

```
00001 #ifndef BASE_HASH_TABLE_HPP
00002 #define BASE_HASH_TABLE_HPP
00003
00004 #include <vector>
00005
00006 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00007 class BaseHashTable {
00008 protected:
00012     std::vector<Collection> table;
00013
00017     size_t tableSize;
00018
00022     float maxLoadFactor;
00023
00027     size_t numberOfElements;
00028
00032     Hash hashing;
00033
00037     mutable size_t collisionsCount;
00038
00039
00051     size_t getNextPrime(size_t num) const;
00052
00067     void checkAndRehash();
00068 public:
00084     BaseHashTable(size_t size = 7, float mlf = 0.7);
00085
00094     float getLoadFactor() const;
00095
00105     void clearHashTable();
00106
00116     void incrementCollisionsCount(size_t m = 1) const;
00117 };
00118
00119 #include "HashTables/Base/BaseHashTable.impl.hpp"
00120
00121 #endif
```

5.4 BaseHashTable.impl.hpp

```
00001 #include "HashTables/Base/BaseHashTable.hpp"
00002
00003 #include <cmath>
00004
00005 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00006 size_t BaseHashTable<HashTable, Collection, Key, Value, Hash>::getNextPrime(size_t num) const {
00007     auto isPrime = [&num](size_t x) -> bool {
00008         if (x <= 1) return false;
00009         if (x == 2 or x == 3) return true;
00010         if (x % 2 == 0) return false;
00011
00012         for (int i = 3; i <= sqrt(x); i += 2) {
00013             if (x % i == 0) return false;
00014         }
00015
00016         return true;
00017     };
00018
00019     size_t candidate;
00020     if (num % 2 == 0) candidate = num + 1;
00021     else candidate = num + 2;
00022     while (true) {
00023         if (isPrime(candidate)) return candidate;
00024         candidate += 2;
00025     }
```

```
00026 }
00027
00028 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00029 void BaseHashTable<HashTable, Collection, Key, Value, Hash>::checkAndRehash() {
00030     if (getLoadFactor() >= maxLoadFactor)
00031         static_cast<HashTable*>(this)->rehash(2 * tableSize);
00032 }
00033
00034 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00035 BaseHashTable<HashTable, Collection, Key, Value, Hash>::BaseHashTable(size_t size, float mlf) {
00036     tableSize = size;
00037     table.resize(tableSize);
00038     maxLoadFactor = mlf <= 0 ? 0.7 : mlf;
00039     numberOfElements = 0;
00040     collisionsCount = 0;
00041 }
00042
00043 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00044 float BaseHashTable<HashTable, Collection, Key, Value, Hash>::getLoadFactor() const {
00045     return static_cast<float>(this->numberOfElements) / this->tableSize;
00046 }
00047
00048 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00049 void BaseHashTable<HashTable, Collection, Key, Value, Hash>::clearHashTable() {
00050     table.clear();
00051     table.resize(tableSize);
00052     numberOfElements = 0;
00053     collisionsCount = 0;
00054     static_cast<HashTable*>(this)->resetCounter();
00055 }
00056
00057 template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
00058 void BaseHashTable<HashTable, Collection, Key, Value, Hash>::incrementCollisionsCount(size_t amount)
      const {
00059     collisionsCount += amount;
00060 }
```

5.5 ChainedHashTable.hpp

```
00001 #ifndef CHAINED_HASH_TABLE_HPP
00002 #define CHAINED_HASH_TABLE_HPP
00003
00004 #include <vector>
00005 #include <list>
00006 #include <utility>
00007 #include <functional>
00008
00009 #include "HashTables/Base/BaseHashTable.hpp"
00010 #include "Dictionary/IDictionary.hpp"
00011
00019 template <typename Key, typename Value, typename Hash = std::hash<Key»
00020 class ChainedHashTable : public IDictionary<Key, Value>, public BaseHashTable<ChainedHashTable<Key,
      Value, Hash>, std::list<std::pair<Key, Value», Key, Value, Hash> {
00021
00028     template <typename Iterator, typename BucketRef>
00029     struct GenericFindResult {
00033         Iterator iterator;
00034
00038         BucketRef bucketRef;
00039
00046         GenericFindResult(Iterator it, BucketRef bRef);
00047
00053         bool wasElementFound() const;
00054     };
00055
00059     using FindResult = GenericFindResult<
00060         typename std::list<std::pair<Key, Value»::iterator,
00061         std::list<std::pair<Key, Value»&>;
00062
00066     using ConstFindResult = GenericFindResult<
00067         typename std::list<std::pair<Key, Value»::const_iterator,
00068         const std::list<std::pair<Key, Value»&>;
00069
00080     size_t hashCode(const Key& key) const;
00081
00082
00099     ConstFindResult findConstPairIterator(const Key& key) const;
00100
00112     FindResult findPairIterator(const Key& key);
00113 public:
00125     ChainedHashTable(size_t size = 7, float mlf = 1.0);
00126
00139     void insert(const Key& key, const Value& value) override;
```

```
00140
00149     bool find(const Key& key, Value& outValue) const override;
00150
00163     void update(const Key& key, const Value& value) override;
00164
00175     void remove(const Key& key) override;
00176
00184     void clear() override;
00185
00202     void printInOrder(std::ostream& out) const override;
00203
00214     size_t getComparisonsCount() const override;
00215
00234     Value& operator[](const Key& key) override;
00235
00253     const Value& operator[](const Key& key) const override;
00254
00266     void rehash(size_t m);
00267
00277     size_t getCollissionsCount() const;
00278
00288     size_t getTableSize() const;
00289 };
00290
00291 #include "HashTables/Chained/ChainedHashTable.impl.hpp"
00292
00293 #endif
00294
```

5.6 ChainedHashTable.impl.hpp

```
00001 #include "HashTables/Chained/ChainedHashTable.hpp"
00002
00003 #include <cmath>
00004 #include <iomanip>
00005
00006 #include "Exceptions/KeyExceptions.hpp"
00007 #include "Utils/StringHandler.hpp"
00008
00009 template <typename Key, typename Value, typename Hash>
00010 template <typename Iterator, typename BucketRef>
00011 ChainedHashTable<Key, Value, Hash>::GenericFindResult<Iterator, BucketRef>::GenericFindResult(
00012     Iterator it, BucketRef bRef)
00013     : iterator(it), bucketRef(bRef) {}
00014
00015 template <typename Key, typename Value, typename Hash>
00016 template <typename Iterator, typename BucketRef>
00017 bool ChainedHashTable<Key, Value, Hash>::GenericFindResult<Iterator, BucketRef>::wasElementFound()
      const {
00018     return iterator != bucketRef.end();
00019 }
00020
00021 template <typename Key, typename Value, typename Hash>
00022 ChainedHashTable<Key, Value, Hash>::ChainedHashTable(size_t size, float mlf)
00023     : BaseHashTable<ChainedHashTable<Key, Value, Hash>, std::list<std::pair<Key, Value>>, Key, Value,
      Hash>(this->getNextPrime(size), mlf) {}
00024
00025 template <typename Key, typename Value, typename Hash>
00026 size_t ChainedHashTable<Key, Value, Hash>::hashCode(const Key& key) const {
00027     size_t pos = this->hashing(key) % this->tableSize;
00028
00029     if (!this->table[pos].empty())
00030         this->incrementCollisionsCount();
00031
00032     return pos;
00033 }
00034
00035 template <typename Key, typename Value, typename Hash>
00036 void ChainedHashTable<Key, Value, Hash>::rehash(size_t m) {
00037     size_t newTableSize = this->getNextPrime(m);
00038
00039     if (newTableSize > this->tableSize) {
00040         std::vector<std::list<std::pair<Key, Value>>> copy = this->table;
00041         this->table.clear();
00042         this->table.resize(newTableSize);
00043         this->tableSize = newTableSize;
00044         this->numberOfElements = 0;
00045
00046         for (auto& line : copy) {
00047             for (auto& [k, v] : line)
00048                 insert(k, v);
00049             line.clear();
00050         }
```

```
00051     }
00052 }
00053
00054 template <typename Key, typename Value, typename Hash>
00055 typename ChainedHashTable<Key, Value, Hash>::FindResult
      ChainedHashTable<Key, Value, Hash>::findPairIterator(const Key& key) {
00056     size_t slot = hashCode(key);
00057
00058
00059
00060     std::list<std::pair<Key, Value»& lst = this->table[slot];
00061
00062     auto it = std::find_if(lst.begin(), lst.end(), [this, &key](const std::pair<Key, Value>& p) {
00063         this->comparisonsCount++;
00064         return p.first == key;
00065     });
00066
00067     return FindResult(it, lst);
00068 }
00069
00070 template <typename Key, typename Value, typename Hash>
00071 typename ChainedHashTable<Key, Value, Hash>::ConstFindResult
      ChainedHashTable<Key, Value, Hash>::findConstPairIterator(const Key& key) const {
00072     size_t slot = hashCode(key);
00073
00074
00075
00076     const std::list<std::pair<Key, Value»& lst = this->table[slot];
00077
00078     auto it = std::find_if(lst.begin(), lst.end(), [this, &key](const std::pair<Key, Value>& p) {
00079         this->comparisonsCount++;
00080         return p.first == key;
00081     });
00082
00083     return ConstFindResult(it, lst);
00084 }
00085
00086 template <typename Key, typename Value, typename Hash>
00087 void ChainedHashTable<Key, Value, Hash>::insert(const Key& key, const Value& value) {
00088     this->checkAndRehash();
00089
00090     size_t slot = hashCode(key);
00091
00092
00093     for (const auto& p : this->table[slot]) {
00094         this->comparisonsCount++;
00095         if (p.first == key) throw KeyAlreadyExistsException();
00096     }
00097
00098     this->table[slot].push_back({key, value});
00099     this->numberOfElements++;
00100 }
00101
00102 template <typename Key, typename Value, typename Hash>
00103 bool ChainedHashTable<Key, Value, Hash>::find(const Key& key, Value& outValue) const {
00104     ConstFindResult response = findConstPairIterator(key);
00105
00106     bool wasFound = response.wasElementFound();
00107
00108     if (wasFound) outValue = response.iterator->second;
00109
00110     return wasFound;
00111 }
00112
00113 template <typename Key, typename Value, typename Hash>
00114 void ChainedHashTable<Key, Value, Hash>::update(const Key& key, const Value& value) {
00115     FindResult response = findPairIterator(key);
00116
00117     if (!response.wasElementFound()) throw KeyNotFoundException();
00118
00119     response.iterator->second = value;
00120 }
00121
00122 template <typename Key, typename Value, typename Hash>
00123 void ChainedHashTable<Key, Value, Hash>::remove(const Key& key) {
00124     FindResult response = findPairIterator(key);
00125
00126     if (response.wasElementFound()) {
00127         response.bucketRef.erase(response.iterator);
00128         this->numberOfElements--;
00129     }
00130 }
00131
00132 template <typename Key, typename Value, typename Hash>
00133 void ChainedHashTable<Key, Value, Hash>::clear() {
00134     this->clearHashTable();
00135 }
```

```
00136
00137 template <typename Key, typename Value, typename Hash>
00138 void ChainedHashTable<Key, Value, Hash>::printInOrder(std::ostream& out) const {
00139     size_t maxKeyLen = 0, maxValLen = 0;
00140     std::vector<std::pair<Key, Value» vec(this->numberOfElements);
00141
00142     size_t i = 0;
00143     for (const auto& line : this->table) {
00144         for (const auto& p : line) {
00145             maxKeyLen = std::max(maxKeyLen, StringHandler::size(p.first));
00146             maxValLen = std::max(maxValLen, StringHandler::size(p.second));
00147
00148             vec[i++] = p;
00149         }
00150     }
00151
00152     std::sort(vec.begin(), vec.end(), [](const auto& pa, const auto& pb) {
00153         return pa.first < pb.first;
00154     });
00155
00156
00157     for (const auto& p : vec) {
00158         out « StringHandler::SetWidthAtLeft(p.first, maxKeyLen) « " | " «
      StringHandler::SetWidthAtLeft(p.second, maxValLen) « "\n";
00159     }
00160 }
00161
00162 template <typename Key, typename Value, typename Hash>
00163 size_t ChainedHashTable<Key, Value, Hash>::getComparisonsCount() const {
00164     return this->comparisonsCount;
00165 }
00166
00167 template <typename Key, typename Value, typename Hash>
00168 Value& ChainedHashTable<Key, Value, Hash>::operator[](const Key& key) {
00169     this->checkAndRehash();
00170
00171     FindResult response = findPairIterator(key);
00172
00173     if (!response.wasElementFound()) {
00174
00175         response.bucketRef.push_back({key, Value()});
00176         this->numberOfElements++;
00177         return response.bucketRef.back().second;
00178     } else {
00179         return response.iterator->second;
00180     }
00181 }
00182
00183 template <typename Key, typename Value, typename Hash>
00184 const Value& ChainedHashTable<Key, Value, Hash>::operator[](const Key& key) const {
00185     ConstFindResult response = findConstPairIterator(key);
00186
00187     if (!response.wasElementFound()) {
00188         throw KeyNotFoundException();
00189     } else {
00190         return response.iterator->second;
00191     }
00192 }
00193
00194 template <typename Key, typename Value, typename Hash>
00195 size_t ChainedHashTable<Key, Value, Hash>::getCollissionsCount() const {
00196     return this->collisionsCount;
00197 }
00198
00199 template <typename Key, typename Value, typename Hash>
00200 size_t ChainedHashTable<Key, Value, Hash>::getTableSize() const {
00201     return this->tableSize;
00202 }
```

5.7 OpenAddressingHashTable.hpp

```
00001 #ifndef OPEN_ADDRESSING_HASH_TABLE_HPP
00002 #define OPEN_ADDRESSING_HASH_TABLE_HPP
00003
00004 #include <vector>
00005
00006 #include "Dictionary/IDictionary.hpp"
00007 #include "HashTables/Base/BaseHashTable.hpp"
00008 #include "HashTables/OpenAddressing/Slot.hpp"
00009
00017 template <typename Key, typename Value, typename Hash = std::hash<Key»
00018 class OpenAddressingHashTable : public IDictionary<Key, Value>, public
      BaseHashTable<OpenAddressingHashTable<Key, Value, Hash>, Slot<Key, Value>, Key, Value, Hash>{
```

```
00019 private:
00025     template <typename Entry>
00026     struct GenericFindResult {
00030         Entry* slot;
00031
00035         Entry* availableSlot;
00036
00043         GenericFindResult(Entry* e, Entry* as = nullptr);
00044
00050         bool wasElementFound() const;
00051     };
00052
00056     using FindResult = GenericFindResult<Slot<Key, Value»;
00057
00061     using ConstFindResult = GenericFindResult<const Slot<Key, Value»;
00062
00063
00075     size_t hashCode(const Key& key, size_t i) const;
00076
00088     ConstFindResult findConstSlot(const Key& key) const;
00089
00102     FindResult findSlot(const Key& key);
00103
00114     size_t nextBase2Of(size_t m) const;
00115 public:
00122     OpenAddressingHashTable(size_t size = 8, float mlf = 0.7);
00123
00137     void insert(const Key& key, const Value& value);
00138
00149     bool find(const Key& key, Value& outValue) const;
00150
00162     void update(const Key& key, const Value& value);
00163
00172     void remove(const Key& key);
00173
00181     void clear();
00182
00194     void printInOrder(std::ostream& out) const;
00195
00205     size_t getComparisonsCount() const;
00206
00220     Value& operator[](const Key& key);
00221
00232     const Value& operator[](const Key& key) const;
00233
00244     void rehash(size_t m);
00245
00255     size_t getCollisionsCount() const;
00256
00266     size_t getTableSize() const;
00267 };
00268
00269 #include "HashTables/OpenAddressing/OpenAddressingHashTable.impl.hpp"
00270
00271 #endif
```

5.8 OpenAddressingHashTable.impl.hpp

```
00001 #include "HashTables/OpenAddressing/OpenAddressingHashTable.hpp"
00002
00003 #include <iostream>
00004 #include <algorithm>
00005
00006 #include "Exceptions/KeyExceptions.hpp"
00007
00008 template <typename Key, typename Value, typename Hash>
00009 template <typename Entry>
00010 OpenAddressingHashTable<Key, Value, Hash>::GenericFindResult<Entry>::GenericFindResult(Entry* e,
      Entry* as)
00011     : slot(e), availableSlot(as) {}
00012
00013 template <typename Key, typename Value, typename Hash>
00014 template <typename Entry>
00015 bool OpenAddressingHashTable<Key, Value, Hash>::GenericFindResult<Entry>::wasElementFound() const {
00016     return slot != nullptr;
00017 }
00018
00019 template <typename Key, typename Value, typename Hash>
00020 size_t OpenAddressingHashTable<Key, Value, Hash>::hashCode(const Key& key, size_t i) const {
00021     size_t pos = (this->hashing(key) + ((i + (i * i)) / 2)) % this->tableSize;
00022
00023     if (this->table[pos].status == ACTIVE)
00024         this->incrementCollisionsCount();
```

```
00025
00026     return pos;
00027 }
00028
00029 template <typename Key, typename Value, typename Hash>
00030 typename OpenAddressingHashTable<Key, Value, Hash>::ConstFindResult
      OpenAddressingHashTable<Key, Value, Hash>::findConstSlot(const Key& key) const {
00031     const Slot<Key, Value>* tableSlot = nullptr;
00032
00033     for (size_t i = 0; i < this->tableSize; i++) {
00034         size_t slotIdx = hashCode(key, i);
00035
00036         const Slot<Key, Value>& slot = this->table[slotIdx];
00037
00038         if (slot.status == EMPTY) {
00039             this->incrementCounter(1);
00040             break;
00041         }
00042
00043         if (slot.status == ACTIVE and slot.key == key)
00044             tableSlot = &slot;
00045
00046         this->incrementCounter(2);
00047     }
00048
00049     return ConstFindResult(tableSlot);
00050 }
00051
00052 template <typename Key, typename Value, typename Hash>
00053 typename OpenAddressingHashTable<Key, Value, Hash>::FindResult
      OpenAddressingHashTable<Key, Value, Hash>::findSlot(const Key& key) {
00054     Slot<Key, Value> *tableSlot = nullptr, *availableSlot = nullptr;
00055
00056     for (size_t i = 0; i < this->tableSize; i++) {
00057         size_t slotIdx = hashCode(key, i);
00058         Slot<Key, Value>& slot = this->table[slotIdx];
00059
00060         if (slot.status == EMPTY) {
00061             if (!availableSlot)
00062                 availableSlot = &slot;
00063
00064             this->incrementCounter(2);
00065
00066             break;
00067         }
00068
00069         if (slot.status == ACTIVE and slot.key == key) {
00070             this->incrementCounter(2);
00071
00072             tableSlot = &slot;
00073             break;
00074         }
00075
00076         if (slot.status == DELETED and !availableSlot)
00077             availableSlot = &slot;
00078
00079         this->incrementCounter(3);
00080
00081     }
00082
00083     return FindResult(tableSlot, availableSlot);
00084 }
00085
00086 template <typename Key, typename Value, typename Hash>
00087 size_t OpenAddressingHashTable<Key, Value, Hash>::nextBase2Of(size_t m) const {
00088     if (m <= 0)
00089         return 1;
00090
00091
00092
00093     size_t n = m, bits = sizeof(size_t) * 8;
00094
00095     for (size_t i = 1; i < bits; i *= 2)
00096         n |= n >> i;
00097
00098     return (n + 1);
00099 }
00100
00101 template <typename Key, typename Value, typename Hash>
00102 OpenAddressingHashTable<Key, Value, Hash>::OpenAddressingHashTable(size_t size, float mlf)
00103     : BaseHashTable<OpenAddressingHashTable<Key, Value, Hash>, Slot<Key, Value>, Key, Value,
      Hash>(nextBase2Of(size), mlf) {}
00104
00105 template <typename Key, typename Value, typename Hash>
00106 void OpenAddressingHashTable<Key, Value, Hash>::rehash(size_t m) {
00107
00108     if (m > this->tableSize) {
```

```
00109         std::vector<Slot<Key, Value» copy = this->table;
00110         this->table.clear();
00111         this->table.resize(m);
00112         this->tableSize = m;
00113         this->numberOfElements = 0;
00114 
00115         for (auto& slot : copy) {
00116             if (slot.status == ACTIVE)
00117                 insert(slot.key, slot.value);
00118         }
00119     }
00120 }
00121 
00122 template <typename Key, typename Value, typename Hash>
00123 void OpenAddressingHashTable<Key, Value, Hash>::insert(const Key& key, const Value& value) {
00124     this->checkAndRehash();
00125 
00126     int lastDeletedSlot = -1;
00127 
00128     for (int i = 0; i < this->tableSize; i++) {
00129         size_t slotIdx = hashCode(key, i);
00130         Slot<Key, Value>& slot = this->table[slotIdx];
00131 
00132         if (slot.status == EMPTY) {
00133             this->incrementCounter(1);
00134             slot = Slot(key, value);
00135             this->numberOfElements++;
00136             return;
00137         } else if (slot.status == ACTIVE and slot.key == key) {
00138             this->incrementCounter(2);
00139             throw KeyAlreadyExistsException();
00140         } else if (slot.status == DELETED and lastDeletedSlot == -1) {
00141             this->incrementCounter(3);
00142             lastDeletedSlot = slotIdx;
00143         } else {
00144             this->incrementCounter(3);
00145         }
00146     }
00147 
00148     this->table[lastDeletedSlot] = Slot(key, value);
00149 }
00150 
00151 template <typename Key, typename Value, typename Hash>
00152 bool OpenAddressingHashTable<Key, Value, Hash>::find(const Key& key, Value& outValue) const {
00153     ConstFindResult response = findConstSlot(key);
00154     bool wasElementFound = response.wasElementFound();
00155 
00156     if (wasElementFound)
00157         outValue = response.slot->value;
00158 
00159     return wasElementFound;
00160 }
00161 
00162 template <typename Key, typename Value, typename Hash>
00163 void OpenAddressingHashTable<Key, Value, Hash>::update(const Key& key, const Value& value) {
00164     FindResult response = findSlot(key);
00165     bool wasElementFound = response.wasElementFound();
00166 
00167     if (!wasElementFound)
00168         throw KeyNotFoundException();
00169 
00170     response.slot->value = value;
00171 }
00172 
00173 template <typename Key, typename Value, typename Hash>
00174 void OpenAddressingHashTable<Key, Value, Hash>::remove(const Key& key) {
00175     FindResult response = findSlot(key);
00176 
00177     if (response.wasElementFound())
00178         response.slot->status = DELETED;
00179 }
00180 
00181 template <typename Key, typename Value, typename Hash>
00182 void OpenAddressingHashTable<Key, Value, Hash>::clear() {
00183     this->clearHashTable();
00184 }
00185 
00186 template <typename Key, typename Value, typename Hash>
00187 void OpenAddressingHashTable<Key, Value, Hash>::printInOrder(std::ostream& out) const {
00188     size_t maxKeyLen = 0, maxValLen = 0, i = 0;
00189     std::vector<Slot<Key, Value» vec(this->numberOfElements);
00190 
00191     for (const Slot<Key, Value>& slot : this->table) {
00192         if (slot.status == ACTIVE) {
00193             maxKeyLen = std::max(maxKeyLen, StringHandler::size(slot.key));
00194             maxValLen = std::max(maxValLen, StringHandler::size(slot.value));
00195 
```

```
00196             vec[i++] = slot;
00197         }
00198     }
00199
00200     std::sort(vec.begin(), vec.end(), [](const Slot<Key, Value>& slotA, const Slot<Key, Value>& slotB)
      {
00201         return slotA.key < slotB.key;
00202     });
00203
00204     for (const Slot<Key, Value>& slot : vec)
00205         if (slot.status == ACTIVE)
00206             out « StringHandler::SetWidthAtLeft(slot.key, maxKeyLen) « " | " «
      StringHandler::SetWidthAtLeft(slot.value, maxValLen) « "\n";
00207 }
00208
00209 template <typename Key, typename Value, typename Hash>
00210 size_t OpenAddressingHashTable<Key, Value, Hash>::getComparisonsCount() const {
00211     return this->comparisonsCount;
00212 }
00213
00214 template <typename Key, typename Value, typename Hash>
00215 Value& OpenAddressingHashTable<Key, Value, Hash>::operator[](const Key& key) {
00216     this->checkAndRehash();
00217
00218     FindResult response = findSlot(key);
00219
00220     if (response.wasElementFound())
00221         return response.slot->value;
00222
00223     this->numberOfElements++;
00224     response.availableSlot->key = key;
00225     response.availableSlot->value = Value();
00226     response.availableSlot->status = ACTIVE;
00227     return response.availableSlot->value;
00228 }
00229
00230 template <typename Key, typename Value, typename Hash>
00231 const Value& OpenAddressingHashTable<Key, Value, Hash>::operator[](const Key& key) const {
00232     ConstFindResult response = findConstSlot(key);
00233
00234     if (!response.wasElementFound())
00235         throw KeyNotFoundException();
00236
00237     return response.slot->value;
00238 }
00239
00240 template <typename Key, typename Value, typename Hash>
00241 size_t OpenAddressingHashTable<Key, Value, Hash>::getCollisionsCount() const {
00242     return this->collisionsCount;
00243 }
00244
00245 template <typename Key, typename Value, typename Hash>
00246 size_t OpenAddressingHashTable<Key, Value, Hash>::getTableSize() const {
00247     return this->tableSize;
00248 }
```

5.9 Slot.hpp

```
00001 #ifndef SLOT_HPP
00002 #define SLOT_HPP
00003
00004 enum Status { EMPTY, ACTIVE, DELETED };
00005
00006 template <typename Key, typename Value>
00034 struct Slot {
00038     Key key;
00039
00043     Value value;
00044
00048     Status status;
00049
00053     Slot(): status(EMPTY) {}
00054
00061     Slot(const Key& k, const Value& v): key(k), value(v), status(ACTIVE) {}
00062
00063 };
00064
00065 #endif
```

5.10 AVLNode.hpp

```
00001 #ifndef AVL_NODE_HPP
00002 #define AVL_NODE_HPP
00003
00004 #include "Trees/Base/Node.hpp"
00005
00012 template <typename Key, typename Value>
00013 struct AVLNode : public Node<Key, Value> {
00017     AVLNode *left;
00018
00022     AVLNode *right;
00023
00027     size_t height;
00028
00035     AVLNode(const Key& k, const Value& v)
00036         : Node<Key, Value>(k, v), left(nullptr), right(nullptr), height(1) {}
00037 };
00038
00039
00040 #endif
```

5.11 AVLTree.hpp

```
00001 #ifndef AVL_TREE_HPP
00002 #define AVL_TREE_HPP
00003
00004 #include <functional>
00005 #include <iostream>
00006
00007 #include "Dictionary/IDictionary.hpp"
00008 #include "Trees/Base/Node.hpp"
00009 #include "Trees/Base/BaseTree.hpp"
00010 #include "Trees/AVL/AVLNode.hpp"
00011
00022 template <typename Key, typename Value>
00023 class AVLTree : public IDictionary<Key, Value>, public BaseTree<AVLTree<Key, Value>, AVLNode<Key,
      Value>, Key, Value> {
00024 private:
00031     size_t height(AVLNode<Key, Value>* node) const;
00032
00039     size_t calcHeight(AVLNode<Key, Value>* node) const;
00040
00050     int getBalanceFactor(AVLNode<Key, Value>* node) const;
00051
00058     void printTree(AVLNode<Key, Value>* node, size_t depth = 0) const;
00059
00066     AVLNode<Key, Value>* rotateLeft(AVLNode<Key, Value>*& y);
00067
00074     AVLNode<Key, Value>* rotateRight(AVLNode<Key, Value>*& y);
00075
00084     AVLNode<Key, Value>* fixupNode(AVLNode<Key, Value>* node);
00085
00095     AVLNode<Key, Value>* removeSuccessor(AVLNode<Key, Value>* root, AVLNode<Key, Value>* node);
00096
00106     AVLNode<Key, Value>* insert(const Key& key, const Value& value, AVLNode<Key, Value>* node);
00107
00117     AVLNode<Key, Value>* update(const Key& key, const Value& value, AVLNode<Key, Value>* node);
00118
00126     AVLNode<Key, Value>* remove(const Key& key, AVLNode<Key, Value>* node);
00127
00142     AVLNode<Key, Value>* upsert(const Key& key, AVLNode<Key, Value>* node, Value*& outValue);
00143
00144     public:
00145     static const int IMBALANCE = 2;
00146
00150     AVLTree();
00151
00155     ~AVLTree();
00156
00165     void insert(const Key& key, const Value& value) override;
00166
00175     bool find(const Key& key, Value& outValue) const override;
00176
00184     void update(const Key& key, const Value& value) override;
00185
00191     void remove(const Key& key) override;
00192
00196     void clear() override;
00197
00203     void printInOrder(std::ostream& out) const override;
00204
00210     size_t getComparisonsCount() const override;
```

```
00211
00218     Value& operator[](const Key& key) override;
00219
00226     const Value& operator[](const Key& key) const override;
00227
00231     void print() const;
00232
00242     size_t getRotationsCount() const;
00243 };
00244
00245 #include "Trees/AVL/AVLTree.impl.hpp"
00246
00247 #endif
```

5.12 AVLTree.impl.hpp

```
00001 #include "Trees/AVL/AVLTree.hpp"
00002
00003 #include <iostream>
00004 #include <cmath>
00005
00006 #include "Utils/StringHandler.hpp"
00007
00008 template <typename Key, typename Value>
00009 size_t AVLTree<Key, Value>::height(AVLNode<Key, Value>* node) const {
00010     if (!node) return 0;
00011
00012     return node->height;
00013 }
00014
00015 template <typename Key, typename Value>
00016 size_t AVLTree<Key, Value>::calcHeight(AVLNode<Key, Value>* node) const {
00017     if (!node) return 0;
00018
00019     size_t leftHeight = height(node->left),
00020            rightHeight = height(node->right);
00021
00022     return 1 + std::max(leftHeight, rightHeight);
00023 }
00024
00025 template <typename Key, typename Value>
00026 int AVLTree<Key, Value>::getBalanceFactor(AVLNode<Key, Value>* node) const {
00027     if (!node) return 0;
00028     return height(node->right) - height(node->left);
00029 }
00030
00031 template <typename Key, typename Value>
00032 AVLNode<Key, Value>* AVLTree<Key, Value>::rotateLeft(AVLNode<Key, Value>*& y) {
00033     AVLNode<Key, Value>* x = y->right;
00034
00035     y->right = x->left;
00036     x->left = y;
00037
00038     y->height = calcHeight(y);
00039     x->height = calcHeight(x);
00040
00041     this->incrementRotationsCount();
00042
00043     return x;
00044 }
00045
00046 template <typename Key, typename Value>
00047 AVLNode<Key, Value>* AVLTree<Key, Value>::rotateRight(AVLNode<Key, Value>*& y) {
00048     AVLNode<Key, Value>* x = y->left;
00049
00050     y->left = x->right;
00051     x->right = y;
00052
00053     y->height = calcHeight(y);
00054     x->height = calcHeight(x);
00055
00056     this->incrementRotationsCount();
00057
00058     return x;
00059 }
00060
00061 template <typename Key, typename Value>
00062 AVLNode<Key, Value>* AVLTree<Key, Value>::fixupNode(AVLNode<Key, Value>* y) {
00063     if (!y) return nullptr;
00064
00065     int balanceFactor = getBalanceFactor(y);
00066
00067     if (std::abs(balanceFactor) == IMBALANCE) {
```

```
00068         if (balanceFactor < 0) {
00069             if (getBalanceFactor(y->left) <= 0) {
00070                 y = rotateRight(y);
00071 
00072             } else {
00073                 y->left = rotateLeft(y->left);
00074                 y = rotateRight(y);
00075 
00076             }
00077         } else {
00078             if (getBalanceFactor(y->right) >= 0) {
00079                 y = rotateLeft(y);
00080 
00081             } else {
00082                 y->right = rotateRight(y->right);
00083                 y = rotateLeft(y);
00084 
00085             }
00086         }
00087     }
00088 
00089     y->height = calcHeight(y);
00090 
00091     return y;
00092 }
00093 
00094 template <typename Key, typename Value>
00095 AVLNode<Key, Value>* AVLTree<Key, Value>::removeSuccessor(AVLNode<Key, Value>* root,
      AVLNode<Key, Value>* node) {
00096     if (node->left) {
00097         node->left = removeSuccessor(root, node->left);
00098     } else {
00099         root->setKey(node->getKey());
00100         root->setValue(node->getValue());
00101         AVLNode<Key, Value>* aux = node->right;
00102         delete node;
00103         return aux;
00104     }
00105 
00106     return fixupNode(node);
00107 }
00108 
00109 template <typename Key, typename Value>
00110 AVLNode<Key, Value>* AVLTree<Key, Value>::insert(const Key& key, const Value& value,
      AVLNode<Key, Value>* node) {
00111     // It'll never be called w/ root == nullptr
00112     if (!node)
00113         return new AVLNode(key, value);
00114 
00115     if (key < node->getKey()) {
00116         this->incrementCounter(1);
00117         node->left = insert(key, value, node->left);
00118     } else if (key > node->getKey()) {
00119         this->incrementCounter(2);
00120         node->right = insert(key, value, node->right);
00121     } else {
00122         this->incrementCounter(2);
00123         throw KeyAlreadyExistsException();
00124     }
00125 
00126     return fixupNode(node);
00127 }
00128 
00129 template <typename Key, typename Value>
00130 AVLNode<Key, Value>* AVLTree<Key, Value>::update(const Key& key, const Value& value,
      AVLNode<Key, Value>* node) {
00131     if (!node) throw KeyNotFoundException();
00132 
00133     if (key < node->getKey()) {
00134         this->incrementCounter(1);
00135         node->left = update(key, value, node->left);
00136     } else if (key > node->getKey()) {
00137         this->incrementCounter(2);
00138         node->right = update(key, value, node->right);
00139     } else {
00140         this->incrementCounter(2);
00141         node->setValue(value);
00142     }
00143 
00144     return fixupNode(node);
00145 }
00146 
00147 template <typename Key, typename Value>
00148 AVLNode<Key, Value>* AVLTree<Key, Value>::remove(const Key& key, AVLNode<Key, Value>* node) {
00149     if (!node) return nullptr;
00150 
00151     if (key < node->getKey()) {
```

```
00152         node->left = remove(key, node->left);
00153     } else if (key > node->getKey()) {
00154         node->right = remove(key, node->right);
00155     } else if (!node->right) {
00156         AVLNode<Key, Value>* leftChild = node->left;
00157         delete node;
00158         return leftChild;
00159     } else {
00160         node->right = removeSuccessor(node, node->right);
00161     }
00162 
00163     return fixupNode(node);
00164 }
00165 
00166 template <typename Key, typename Value>
00167 AVLNode<Key, Value>* AVLTree<Key, Value>::upsert(const Key& key, AVLNode<Key, Value>* node, Value*&
    outValue) {
00168     this->setMaxKeyLen(key);
00169 
00170     if (!node) {
00171         AVLNode<Key, Value>* newNode = new AVLNode<Key, Value>(key, Value());
00172         outValue = &(newNode->getValue());
00173         this->setMaxValLen(*outValue);
00174         return newNode;
00175     }
00176 
00177     if (key < node->getKey()) {
00178         this->incrementCounter(1);
00179         node->left = upsert(key, node->left, outValue);
00180     } else if (key > node->getKey()) {
00181         this->incrementCounter(2);
00182         node->right = upsert(key, node->right, outValue);
00183     } else {
00184         this->incrementCounter(2);
00185         outValue = &(node->getValue());
00186         this->setMaxValLen(*outValue);
00187         return node;
00188     }
00189 
00190     return fixupNode(node);
00191 }
00192 
00193 template <typename Key, typename Value>
00194 AVLTree<Key, Value>::AVLTree()
00195     : BaseTree<AVLTree<Key, Value>, AVLNode<Key, Value>, Key, Value>(nullptr) {}
00196 
00197 template <typename Key, typename Value>
00198 AVLTree<Key, Value>::~AVLTree() { clear(); }
00199 
00200 template <typename Key, typename Value>
00201 void AVLTree<Key, Value>::insert(const Key& key, const Value& value) {
00202     this->root = insert(key, value, this->root);
00203     this->setMaxKeyLen(key);
00204     this->setMaxValLen(value);
00205 }
00206 
00207 template  <typename Key, typename Value>
00208 bool AVLTree<Key, Value>::find(const Key& key, Value& outValue) const {
00209     const AVLNode<Key, Value>* node = this->findNode(key);
00210 
00211     if (!node) return false;
00212 
00213     outValue = node->getValue();
00214     return true;
00215 }
00216 
00217 template <typename Key, typename Value>
00218 void AVLTree<Key, Value>::update(const Key& key, const Value& value) {
00219     this->root = update(key, value, this->root);
00220 }
00221 
00222 template <typename Key, typename Value>
00223 void AVLTree<Key, Value>::remove(const Key& key) {
00224     this->root = remove(key, this->root);
00225 }
00226 
00227 template <typename Key, typename Value>
00228 void AVLTree<Key, Value>::clear() {
00229     this->reset(this->root);
00230 }
00231 
00232 template <typename Key, typename Value>
00233 void AVLTree<Key, Value>::printTree(AVLNode<Key, Value>* node, size_t depth) const {
00234     if (!node) return;
00235 
00236     printTree(node->right, depth+1);
00237 
```

```
00238     for (int i = 0; i < depth; i++)
00239         std::cout « "    ";
00240     std::cout « node->show() « std::endl;
00241 
00242     printTree(node->left, depth+1);
00243 }
00244 
00245 template <typename Key, typename Value>
00246 void AVLTree<Key, Value>::print() const {
00247     printTree(this->root);
00248 }
00249 
00250 template <typename Key, typename Value>
00251 void AVLTree<Key, Value>::printInOrder(std::ostream& os) const {
00252     this->inOrderTransversal(os, this->root, nullptr);
00253 }
00254 
00255 template <typename Key, typename Value>
00256 size_t AVLTree<Key, Value>::getComparisonsCount() const {
00257     return this->comparisonsCount;
00258 }
00259 
00260 template <typename Key, typename Value>
00261 Value& AVLTree<Key, Value>::operator[](const Key& key) {
00262     Value* insertedValue = nullptr;
00263     this->root = upsert(key, this->root, insertedValue);
00264     return *insertedValue;
00265 }
00266 
00267 template <typename Key, typename Value>
00268 const Value& AVLTree<Key, Value>::operator[](const Key& key) const {
00269     return this->at(key);
00270 }
00271 
00272 template <typename Key, typename Value>
00273 size_t AVLTree<Key, Value>::getRotationsCount() const {
00274     return this->rotationsCount;
00275 }
```

5.13 BaseTree.hpp

```
00001 #ifndef BASE_TREE_HPP
00002 #define BASE_TREE_HPP
00003 
00004 #include <iostream>
00005 
00006 #include "Exceptions/KeyExceptions.hpp"
00007 
00021 template <typename Tree, typename Node, typename Key, typename Value>
00022 class BaseTree {
00027     void count(size_t n) const;
00028 
00035     void clearCounter();
00036 protected:
00043     Node* root;
00044 
00048     size_t maxKeyLen;
00049 
00056     size_t maxValLen;
00057 
00065     size_t rotationsCount;
00066 
00077     BaseTree(Node* r);
00078 
00084     const Node* findNode(const Key& key, Node* comp = nullptr) const;
00085 
00091     Node* minimum(Node* node) const;
00092 
00099     void clearNode(Node* node, Node* comp);
00100 
00109     void reset(Node* node, Node* comp = nullptr, Node* defaultRoot = nullptr);
00110 
00117     void inOrderTransversal(std::ostream& out, Node* node, Node* comp) const;
00118 
00125     const Value& at(const Key& key, Node* comp = nullptr) const;
00126 
00137     void setMaxKeyLen(const Key& key);
00138 
00149     void setMaxValLen(const Value& value);
00150 
00161     void incrementRotationsCount(size_t amount = 1);
00162 };
00163 
```

```
00164 // Include the implementation file to provide the definitions for the template methods.
00165 // This must be at the end of the header file.
00166 #include "Trees/Base/BaseTree.impl.hpp"
00167
00168 #endif
```

5.14 BaseTree.impl.hpp

```
00001 #ifndef BASE_TREE_IMPL_HPP
00002 #define BASE_TREE_IMPL_HPP
00003
00004 #include "Trees/Base/BaseTree.hpp"
00005
00006 #include <cmath>
00007
00008 #include "Utils/StringHandler.hpp"
00009
00010 template <typename Tree, typename Node, typename Key, typename Value>
00011 void BaseTree<Tree, Node, Key, Value>::count(size_t n) const {
00012     static_cast<const Tree*>(this)->incrementCounter(n);
00013 }
00014
00015 template <typename Tree, typename Node, typename Key, typename Value>
00016 void BaseTree<Tree, Node, Key, Value>::clearCounter() {
00017     static_cast<Tree*>(this)->resetCounter();
00018 }
00019
00020 template <typename Tree, typename Node, typename Key, typename Value>
00021 BaseTree<Tree, Node, Key, Value>::BaseTree(Node* r)
00022     : root(r), maxKeyLen(0), maxValLen(0), rotationsCount(0) {
00023         clearCounter();
00024     }
00025
00026 template <typename Tree, typename Node, typename Key, typename Value>
00027 const Node* BaseTree<Tree, Node, Key, Value>::findNode(const Key& key, Node* comp) const {
00028     const Node* aux = root;
00029
00030     while (aux != comp) {
00031         if (key < aux->getKey()) {
00032             count(1);
00033             aux = aux->left;
00034         } else if (key > aux->getKey()) {
00035             count(2);
00036             aux = aux->right;
00037         } else {
00038             count(2);
00039             return aux;
00040         }
00041     }
00042
00043     return nullptr;
00044 }
00045
00046 template <typename Tree, typename Node, typename Key, typename Value>
00047 Node* BaseTree<Tree, Node, Key, Value>::minimum(Node* node) const {
00048     if (!node->left) return node;
00049     return minimum(node->left);
00050 }
00051
00052 template <typename Tree, typename Node, typename Key, typename Value>
00053 void BaseTree<Tree, Node, Key, Value>::clearNode(Node* node, Node* comp) {
00054     if (node != comp) {
00055         clearNode(node->left, comp);
00056         clearNode(node->right, comp);
00057         delete node;
00058     }
00059 }
00060
00061 template <typename Tree, typename Node, typename Key, typename Value>
00062 void BaseTree<Tree, Node, Key, Value>::reset(Node* node, Node* comp, Node* defaultRoot) {
00063     clearNode(node, comp);
00064     root = defaultRoot;
00065     maxKeyLen = 0;
00066     maxValLen = 0;
00067     rotationsCount = 0;
00068     clearCounter();
00069 }
00070
00071
00072 template <typename Tree, typename Node, typename Key, typename Value>
00073 void BaseTree<Tree, Node, Key, Value>::inOrderTransversal(std::ostream& out, Node* node, Node* comp)
      const {
00074     if (node != comp) {
```

```
00075         inOrderTransversal(out, node->left, comp);
00076
00077         out « StringHandler::SetWidthAtLeft(node->getKey(), maxKeyLen) « " | "
00078             « StringHandler::SetWidthAtLeft(node->getValue(), maxValLen) « '\n';
00079
00080         inOrderTransversal(out, node->right, comp);
00081     }
00082 }
00083
00084 template <typename Tree, typename Node, typename Key, typename Value>
00085 const Value& BaseTree<Tree, Node, Key, Value>::at(const Key& key, Node* comp) const {
00086     const Node* aux = root;
00087
00088     while (aux != comp) {
00089         if (key < aux->getKey()) {
00090             count(1);
00091             aux = aux->left;
00092         } else if (key > aux->getKey()) {
00093             count(2);
00094             aux = aux->right;
00095         } else {
00096             count(2);
00097             return aux->getValue();
00098         }
00099     }
00100
00101     throw KeyNotFoundException();
00102 }
00103
00104 template <typename Tree, typename Node, typename Key, typename Value>
00105 void BaseTree<Tree, Node, Key, Value>::setMaxKeyLen(const Key& key) {
00106     maxKeyLen = std::max(maxKeyLen, StringHandler::size(key));
00107 }
00108
00109 template <typename Tree, typename Node, typename Key, typename Value>
00110 void BaseTree<Tree, Node, Key, Value>::setMaxValLen(const Value& value) {
00111     maxValLen = std::max(maxValLen, StringHandler::size(value));
00112 }
00113
00114 template <typename Tree, typename Node, typename Key, typename Value>
00115 void BaseTree<Tree, Node, Key, Value>::incrementRotationsCount(size_t amount) {
00116     rotationsCount += amount;
00117 }
00118
00119 #endif
```

5.15 Node.hpp

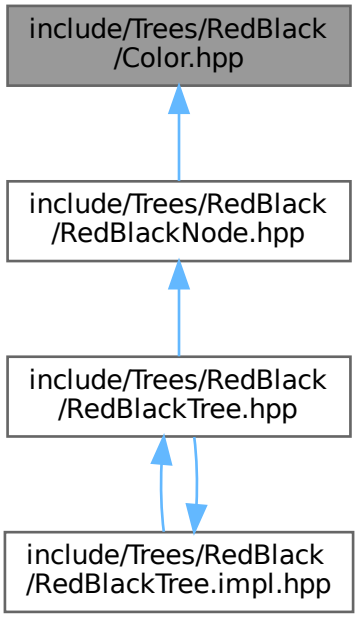
```
00001 #ifndef INODE_HPP
00002 #define INODE_HPP
00003
00004 #include <utility>
00005 #include <sstream>
00006
00013 template <typename Key, typename Value>
00014 class Node {
00018     std::pair<Key, Value> data;
00019
00020 public:
00027     Node(const Key& key, const Value& value): data({key, value}) {}
00028
00034     const Key& getKey() const { return data.first; }
00035
00041     void setKey(const Key& key) { data.first = key; }
00042
00048     const Value& getValue() const { return data.second; }
00049
00055     Value& getValue() { return data.second; }
00056
00062     void setValue(const Value& value) { data.second = value; }
00063
00070     void update(const Key& key, const Value& value) {
00071         setKey(key);
00072         setValue(value);
00073     }
00074
00080     std::string show() const {
00081         std::ostringstream os;
00082         os « "(" « getKey() « ", " « getValue() « ")";
00083         return os.str();
00084     }
00085
00089     ~Node() = default;
```

```
00090 };
00091
00092
00093 #endif
```

5.16 include/Trees/RedBlack/Color.hpp File Reference

Defines the Color enumeration used in Red-Black Tree nodes.

This graph shows which files directly or indirectly include this file:



Enumerations

- enum Color { RED , BLACK }

5.16.1 Detailed Description

Defines the Color enumeration used in Red-Black Tree nodes.

This header file contains the definition of the `Color` enumeration, which represents the color of a node in a Red-Black Tree. The two possible values are `RED` and `BLACK`.

5.16.2 Enumeration Type Documentation

5.16.2.1 Color

```
enum Color
```

Enumerator

RED	Represents a red-colored node in the Red-Black Tree.
BLACK	Represents a black-colored node in the Red-Black Tree.

5.17 Color.hpp

Go to the documentation of this file.

```
00001
00015 #ifndef COLOR_HPP
00016 #define COLOR_HPP
00017
00018 enum Color { RED, BLACK };
00019
00020 #endif
```

5.18 RedBlackNode.hpp

```
00001 #ifndef RED_BLACK_NODE_HPP
00002 #define RED_BLACK_NODE_HPP
00003
00004 #include "Trees/Base/Node.hpp"
00005 #include "Trees/RedBlack/Color.hpp"
00006
00007 template <typename Key, typename Value>
00008 struct RedBlackNode : public Node<Key, Value> {
00016     RedBlackNode* left;
00017
00024     RedBlackNode* right;
00025
00034     RedBlackNode* parent;
00035
00042     Color color;
00043
00054     RedBlackNode(const Key& k, const Value& v,
00055                  RedBlackNode* l, RedBlackNode* r,
00056                  RedBlackNode* p, Color c)
00057         : Node<Key, Value>(k, v), left(l), right(r), parent(p), color(c) {}
00058
00073     RedBlackNode(Color color = BLACK) : Node<Key, Value>(Key(), Value()) {
00074         this->left = this;
00075         this->right = this;
00076         this->parent = this;
00077         this->color = color;
00078     }
00079 };
00080
00081 #endif
```

5.19 RedBlackTree.hpp

```
00001 #ifndef RED_BLACK_TREE_HPP
00002 #define RED_BLACK_TREE_HPP
00003
00004 #include <functional>
00005 #include <iostream>
00006
00007 #include "Dictionary/IDictionary.hpp"
00008 #include "Trees/RedBlack/RedBlackNode.hpp"
00009 #include "Trees/Base/BaseTree.hpp"
00010
00021 template <typename Key, typename Value>
00022 class RedBlackTree : public IDictionary<Key, Value>, public BaseTree<RedBlackTree<Key, Value>,
      RedBlackNode<Key, Value>, Key, Value> {
00023 private:
00024     static RedBlackNode<Key, Value> NIL_NODE;
00025     static constexpr RedBlackNode<Key, Value>* const NIL = &NIL_NODE;
00026
00033     RedBlackNode<Key, Value>* rotateLeft(RedBlackNode<Key, Value>* y);
```

```
00034
00041     RedBlackNode<Key, Value>* rotateRight(RedBlackNode<Key, Value>* y);
00042
00051     void insertFixup(RedBlackNode<Key, Value>* z);
00052
00065     void deleteFixup(RedBlackNode<Key, Value>* x);
00066
00067
00079     void deleteNode(RedBlackNode<Key, Value>* z);
00080
00087     void printTree(RedBlackNode<Key, Value>* node, int indent = 0) const;
00088
00089 public:
00093     RedBlackTree();
00094
00101     void insert(const Key& key, const Value& value);
00102
00111     bool find(const Key& key, Value& outValue) const;
00112
00120     void update(const Key& key, const Value& value);
00121
00127     void remove(const Key& key);
00128
00132     void clear();
00133
00139     void printInOrder(std::ostream& out) const;
00140
00146     size_t getComparisonsCount() const;
00147
00154     virtual Value& operator[](const Key& key);
00155
00162     virtual const Value& operator[](const Key& key) const;
00163
00167     void print() const;
00168
00177     size_t getRotationsCount() const;
00178 };
00179
00180 #include "Trees/RedBlack/RedBlackTree.impl.hpp"
00181
00182 #endif
```

5.20 RedBlackTree.impl.hpp

```
00001 #include "Trees/RedBlack/RedBlackTree.hpp"
00002
00003 template <typename Key, typename Value>
00004 RedBlackNode<Key, Value>* RedBlackTree<Key, Value>::rotateLeft(RedBlackNode<Key, Value>* y) {
00005     RedBlackNode<Key, Value>* x = y->right;
00006
00007     y->right = x->left;
00008     if (y->right != NIL) y->right->parent = y;
00009     x->left = y;
00010
00011     x->parent = y->parent;
00012     y->parent = x;
00013
00014     if (x->parent != NIL) {
00015         if (x->getKey() < x->parent->getKey())
00016             x->parent->left = x;
00017         else
00018             x->parent->right = x;
00019     } else {
00020         this->root = x;
00021     }
00022
00023     this->incrementRotationsCount();
00024
00025     return x;
00026 }
00027
00028 template <typename Key, typename Value>
00029 RedBlackNode<Key, Value>* RedBlackTree<Key, Value>::rotateRight(RedBlackNode<Key, Value>* y) {
00030     RedBlackNode<Key, Value>* x = y->left;
00031
00032     y->left = x->right;
00033     if (y->left != NIL) y->left->parent = y;
00034     x->right = y;
00035
00036     x->parent = y->parent;
00037     y->parent = x;
00038
00039     if (x->parent != NIL) {
```

```
00040         if (x->getKey() < x->parent->getKey())
00041             x->parent->left = x;
00042         else
00043             x->parent->right = x;
00044     } else {
00045         this->root = x;
00046     }
00047
00048     this->incrementRotationsCount();
00049
00050     return x;
00051 }
00052
00053 template <typename Key, typename Value>
00054 void RedBlackTree<Key, Value>::insertFixup(RedBlackNode<Key, Value>* z) {
00055     while (z->parent->color == RED) {
00056         if (z->parent == z->parent->parent->left) {
00057             if (z->parent->parent->right->color == RED) { // Case 1
00058                 z->parent->color = BLACK;
00059                 z->parent->parent->color = RED;
00060                 z->parent->parent->right->color = BLACK;
00061                 z = z->parent->parent;
00062             } else {
00063                 if (z == z->parent->right) { // Case 2
00064                     z = z->parent;
00065                     z = rotateLeft(z);
00066                     z = z->left;
00067
00068                 }
00069
00070                 z->parent->color = BLACK;
00071                 z->parent->parent->color = RED;
00072                 z = rotateRight(z->parent->parent);
00073
00074             }
00075         } else { // Symmetrical case
00076             if (z->parent->parent->left->color == RED) { // Case 1
00077                 z->parent->color = BLACK;
00078                 z->parent->parent->color = RED;
00079                 z->parent->parent->left->color = BLACK;
00080                 z = z->parent->parent;
00081             } else {
00082                 if (z == z->parent->left) { // Case 2
00083                     z = z->parent;
00084                     z = rotateRight(z);
00085                     z = z->right;
00086
00087                 }
00088
00089                 z->parent->color = BLACK;
00090                 z->parent->parent->color = RED;
00091                 z = rotateLeft(z->parent->parent);
00092
00093             }
00094         }
00095     }
00096
00097     this->root->color = BLACK;
00098 }
00099
00100 template <typename Key, typename Value>
00101 RedBlackNode<Key, Value> RedBlackTree<Key, Value>::NIL_NODE = RedBlackNode<Key, Value>();
00102
00103 template <typename Key, typename Value>
00104 RedBlackTree<Key, Value>::RedBlackTree()
00105     : BaseTree<RedBlackTree<Key, Value>, RedBlackNode<Key, Value>, Key, Value>(NIL) {
00106 }
00107
00108 template <typename Key, typename Value>
00109 void RedBlackTree<Key, Value>::insert(const Key& key, const Value& value) {
00110     RedBlackNode<Key, Value> *x = this->root, *y = NIL;
00111
00112     while (x != NIL) {
00113         y = x;
00114
00115         if (key < x->getKey()) {
00116             this->incrementCounter(1);
00117             x = x->left;
00118         } else if (key > x->getKey()) {
00119             this->incrementCounter(2);
00120             x = x->right;
00121         } else {
00122             this->incrementCounter(2);
00123             throw KeyAlreadyExistsException();
00124         }
00125
00126     }
```

```
00127
00128     this->setMaxKeyLen(key);
00129     this->setMaxValLen(value);
00130
00131     RedBlackNode<Key, Value> *z = new RedBlackNode<Key, Value>(key, value, NIL, NIL, NIL, RED);
00132
00133     z->parent = y;
00134     if (y == NIL) {
00135         this->incrementCounter(1);
00136         this->root = z;
00137     } else if (z->getKey() < y->getKey()) {
00138         this->incrementCounter(2);
00139         y->left = z;
00140     } else {
00141         this->incrementCounter(2);
00142         y->right = z;
00143     }
00144
00145     insertFixup(z);
00146 }
00147
00148 template <typename Key, typename Value>
00149 void RedBlackTree<Key,Value>::deleteFixup(RedBlackNode<Key, Value>* x) {
00150     while (x != this->root and x->color == BLACK) {
00151         if (x == x->parent->left) {
00152             RedBlackNode<Key, Value>* w = x->parent->right;
00153
00154             if (w->color == RED) { // Case 1
00155                 x->parent->color = RED;
00156                 w->color = BLACK;
00157                 x->parent = rotateLeft(x->parent);
00158                 w = x->parent->right;
00159             }
00160
00161             if (w->left->color == BLACK and w->right->color == BLACK) { // Case 2
00162                 w->color = RED;
00163                 x = x->parent;
00164             } else {
00165                 if (w->right->color == BLACK) { // Case 3
00166                     w->left->color = BLACK;
00167                     w->color = RED;
00168                     w = rotateRight(w);
00169
00170                     w = x->parent->right;
00171                 }
00172
00173                 // Case 4
00174                 w->color = x->parent->color;
00175                 x->parent->color = BLACK;
00176                 w->right->color = BLACK;
00177                 w = rotateLeft(x->parent);
00178
00179
00180                 x = this->root;
00181             }
00182         } else { // Symetrical case
00183             RedBlackNode<Key, Value>* w = x->parent->left;
00184
00185             if (w->color == RED) { // Case 1
00186                 x->parent->color = RED;
00187                 w->color = BLACK;
00188                 x->parent = rotateRight(x->parent);
00189
00190                 w = x->parent->left;
00191             }
00192
00193             if (w->right->color == BLACK and w->left->color == BLACK) { // Case 2
00194                 w->color = RED;
00195                 x = x->parent;
00196             } else {
00197                 if (w->left->color == BLACK) { // Case 3
00198                     w->right->color = BLACK;
00199                     w->color = RED;
00200                     w = rotateLeft(w);
00201
00202                     w = x->parent->left;
00203                 }
00204
00205                 // Case 4
00206                 w->color = x->parent->color;
00207                 x->parent->color = BLACK;
00208                 w->left->color = BLACK;
00209                 w = rotateRight(x->parent);
00210
00211
00212                 x = this->root;
00213             }
```

```
00214         }
00215     }
00216
00217     x->color = BLACK;
00218 }
00219
00220 template <typename Key, typename Value>
00221 void RedBlackTree<Key, Value>::deleteNode(RedBlackNode<Key, Value>* z) {
00222     RedBlackNode<Key, Value>* y;
00223     if (z->left == NIL or z->right == NIL)
00224         y = z;
00225     else
00226         y = this->minimum(z->right);
00227
00228     RedBlackNode<Key, Value>* x;
00229     if (y->left != NIL)
00230         x = y->left;
00231     else
00232         x = y->right;
00233
00234     x->parent = y->parent;
00235
00236     if (y->parent == NIL) {
00237         this->root = x;
00238     } else {
00239         if (y == y->parent->left)
00240             y->parent->left = x;
00241         else
00242             y->parent->right = x;
00243     }
00244
00245     if (y != z)
00246         y->setKey(z->getKey());
00247
00248     if (y->color == BLACK)
00249         deleteFixup(x);
00250
00251     delete y;
00252 }
00253
00254 template <typename Key, typename Value>
00255 void RedBlackTree<Key, Value>::printTree(RedBlackNode<Key, Value>* node, int indent) const {
00256     if (node != NIL) {
00257         printTree(node->right, indent + 4);
00258
00259         if (indent > 0) {
00260             std::cout « std::string(indent, ' ');
00261         }
00262
00263         std::cout « node->getKey() « " (" « (node->color == RED ? "R" : "B") « ")" « std::endl;
00264
00265         printTree(node->left, indent + 4);
00266     }
00267 }
00268
00269 template <typename Key, typename Value>
00270 bool RedBlackTree<Key, Value>::find(const Key& key, Value& outValue) const {
00271     const RedBlackNode<Key, Value>* node = this->findNode(key, NIL);
00272
00273     if (!node) return false;
00274
00275     outValue = node->getValue();
00276     return true;
00277 }
00278
00279 template <typename Key, typename Value>
00280 void RedBlackTree<Key, Value>::update(const Key& key, const Value& value) {
00281     RedBlackNode<Key, Value>* aux = this->root;
00282     while (aux != NIL) {
00283         if (key < aux->getKey()) {
00284             this->incrementCounter(1);
00285             aux = aux->left;
00286         } else if (key > aux->getKey()) {
00287             this->incrementCounter(2);
00288             aux = aux->right;
00289         } else {
00290             this->incrementCounter(2);
00291             aux->setValue(value);
00292             return;
00293         }
00294     }
00295
00296     throw KeyNotFoundException();
00297 }
00298
00299 template <typename Key, typename Value>
00300 void RedBlackTree<Key, Value>::print() const {
```

```
00301     printTree(this->root);
00302 }
00303
00304 template <typename Key, typename Value>
00305 void RedBlackTree<Key, Value>::remove(const Key& key) {
00306     RedBlackNode<Key, Value>* p = this->root;
00307
00308     while (p != NIL and p->getKey() != key) {
00309         if (key < p->getKey()) p = p->left;
00310         else p = p->right;
00311
00312         this->incrementCounter(1);
00313     }
00314
00315     if (p != NIL)
00316         deleteNode(p);
00317 }
00318
00319 template <typename Key, typename Value>
00320 void RedBlackTree<Key, Value>::clear() {
00321     this->reset(this->root, NIL, NIL);
00322 }
00323
00324 template <typename Key, typename Value>
00325 void RedBlackTree<Key, Value>::printInOrder(std::ostream& os) const {
00326     this->inOrderTransversal(os, this->root, NIL);
00327 }
00328
00329 template <typename Key, typename Value>
00330 size_t RedBlackTree<Key, Value>::getComparisonsCount() const {
00331     return this->comparisonsCount;
00332 }
00333
00334 template <typename Key, typename Value>
00335 Value& RedBlackTree<Key, Value>::operator[](const Key& key) {
00336     this->setMaxKeyLen(key);
00337
00338     RedBlackNode<Key, Value> *x = this->root, *y = NIL;
00339
00340     while (x != NIL) {
00341         y = x;
00342
00343         if (key < x->getKey()) {
00344             this->incrementCounter(1);
00345             x = x->left;
00346         } else if (key > x->getKey()) {
00347             this->incrementCounter(2);
00348             x = x->right;
00349         } else {
00350             this->incrementCounter(2);
00351             this->setMaxValLen(x->getValue());
00352             return x->getValue();
00353         }
00354     }
00355
00356     RedBlackNode<Key, Value> *z = new RedBlackNode<Key, Value>(key, Value(), NIL, NIL, NIL, RED);
00357
00358     this->setMaxValLen(z->getValue());
00359
00360     z->parent = y;
00361     if (y == NIL) {
00362         this->incrementCounter(1);
00363         this->root = z;
00364     } else if (z->getKey() < y->getKey()) {
00365         this->incrementCounter(2);
00366         y->left = z;
00367     } else {
00368         this->incrementCounter(2);
00369         y->right = z;
00370     }
00371
00372     insertFixup(z);
00373
00374     return z->getValue();
00375 }
00376
00377 template <typename Key, typename Value>
00378 const Value& RedBlackTree<Key, Value>::operator[](const Key& key) const {
00379     return this->at(key);
00380 }
00381
00382 template <typename Key, typename Value>
00383 size_t RedBlackTree<Key, Value>::getRotationsCount() const {
00384     return this->rotationsCount;
00385 }
```

5.21 utf8.h

```
00001 // Copyright 2006 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_2675DCD0_9480_4c0c_B92A_CC14C027B731
00029 #define UTF8_FOR_CPP_2675DCD0_9480_4c0c_B92A_CC14C027B731
00030
00031 /*
00032 To control the C++ language version used by the library, you can define UTF_CPP_CPLUSPLUS macro
00033 and set it to one of the values used by the __cplusplus predefined macro.
00034
00035 For instance,
00036     #define UTF_CPP_CPLUSPLUS 199711L
00037 will cause the UTF-8 CPP library to use only types and language features available in the C++ 98
      standard.
00038 Some library features will be disabled.
00039
00040 If you leave UTF_CPP_CPLUSPLUS undefined, it will be internally assigned to __cplusplus.
00041 */
00042
00043 #include "utf8/checked.h"
00044 #include "utf8/unchecked.h"
00045
00046 #endif // header guard
```

5.22 checked.h

```
00001 // Copyright 2006-2016 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_CHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00029 #define UTF8_FOR_CPP_CHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00030
00031 #include "core.h"
```

```
00032 #include <stdexcept>
00033
00034 namespace utf8
00035 {
00036     // Base for the exceptions that may be thrown from the library
00037     class exception : public ::std::exception {
00038     };
00039
00040     // Exceptions that may be thrown from the library functions.
00041     class invalid_code_point : public exception {
00042         utfchar32_t cp;
00043     public:
00044         invalid_code_point(utfchar32_t codepoint) : cp(codepoint) {}
00045         virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid code
      point"; }
00046         utfchar32_t code_point() const {return cp;}
00047     };
00048
00049     class invalid_utf8 : public exception {
00050         utfchar8_t u8;
00051     public:
00052         invalid_utf8 (utfchar8_t u) : u8(u) {}
00053         invalid_utf8 (char c) : u8(static_cast<utfchar8_t>(c)) {}
00054         virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid UTF-8"; }
00055         utfchar8_t utf8_octet() const {return u8;}
00056     };
00057
00058     class invalid_utf16 : public exception {
00059         utfchar16_t u16;
00060     public:
00061         invalid_utf16 (utfchar16_t u) : u16(u) {}
00062         virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid UTF-16";
      }
00063         utfchar16_t utf16_word() const {return u16;}
00064     };
00065
00066     class not_enough_room : public exception {
00067     public:
00068         virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Not enough
      space"; }
00069     };
00070
00072
00073     template <typename octet_iterator>
00074     octet_iterator append(utfchar32_t cp, octet_iterator result)
00075     {
00076         if (!utf8::internal::is_code_point_valid(cp))
00077             throw invalid_code_point(cp);
00078
00079         return internal::append(cp, result);
00080     }
00081
00082     inline void append(utfchar32_t cp, std::string& s)
00083     {
00084         append(cp, std::back_inserter(s));
00085     }
00086
00087     template <typename word_iterator>
00088     word_iterator append16(utfchar32_t cp, word_iterator result)
00089     {
00090         if (!utf8::internal::is_code_point_valid(cp))
00091             throw invalid_code_point(cp);
00092
00093         return internal::append16(cp, result);
00094     }
00095
00096     template <typename octet_iterator, typename output_iterator>
00097     output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out,
      utfchar32_t replacement)
00098     {
00099         while (start != end) {
00100             octet_iterator sequence_start = start;
00101             internal::utf_error err_code = utf8::internal::validate_next(start, end);
00102             switch (err_code) {
00103                 case internal::UTF8_OK :
00104                     for (octet_iterator it = sequence_start; it != start; ++it)
00105                         *out++ = *it;
00106                     break;
00107                 case internal::NOT_ENOUGH_ROOM:
00108                     out = utf8::append (replacement, out);
00109                     start = end;
00110                     break;
00111                 case internal::INVALID_LEAD:
00112                     out = utf8::append (replacement, out);
00113                     ++start;
00114                     break;
00115                 case internal::INCOMPLETE_SEQUENCE:
```

```
00116                 case internal::OVERLONG_SEQUENCE:
00117                 case internal::INVALID_CODE_POINT:
00118                     out = utf8::append (replacement, out);
00119                     ++start;
00120                     // just one replacement mark for the sequence
00121                     while (start != end && utf8::internal::is_trail(*start))
00122                         ++start;
00123                     break;
00124             }
00125         }
00126         return out;
00127     }
00128
00129     template <typename octet_iterator, typename output_iterator>
00130     inline output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator
      out)
00131     {
00132         static const utfchar32_t replacement_marker = utf8::internal::mask16(0xfffd);
00133         return utf8::replace_invalid(start, end, out, replacement_marker);
00134     }
00135
00136     inline std::string replace_invalid(const std::string& s, utfchar32_t replacement)
00137     {
00138         std::string result;
00139         replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
00140         return result;
00141     }
00142
00143     inline std::string replace_invalid(const std::string& s)
00144     {
00145         std::string result;
00146         replace_invalid(s.begin(), s.end(), std::back_inserter(result));
00147         return result;
00148     }
00149
00150     template <typename octet_iterator>
00151     utfchar32_t next(octet_iterator& it, octet_iterator end)
00152     {
00153         utfchar32_t cp = 0;
00154         internal::utf_error err_code = utf8::internal::validate_next(it, end, cp);
00155         switch (err_code) {
00156             case internal::UTF8_OK :
00157                 break;
00158             case internal::NOT_ENOUGH_ROOM :
00159                 throw not_enough_room();
00160             case internal::INVALID_LEAD :
00161             case internal::INCOMPLETE_SEQUENCE :
00162             case internal::OVERLONG_SEQUENCE :
00163                 throw invalid_utf8(static_cast<utfchar8_t>(*it));
00164             case internal::INVALID_CODE_POINT :
00165                 throw invalid_code_point(cp);
00166         }
00167         return cp;
00168     }
00169
00170     template <typename word_iterator>
00171     utfchar32_t next16(word_iterator& it, word_iterator end)
00172     {
00173         utfchar32_t cp = 0;
00174         internal::utf_error err_code = utf8::internal::validate_next16(it, end, cp);
00175         if (err_code == internal::NOT_ENOUGH_ROOM)
00176             throw not_enough_room();
00177         return cp;
00178     }
00179
00180     template <typename octet_iterator>
00181     utfchar32_t peek_next(octet_iterator it, octet_iterator end)
00182     {
00183         return utf8::next(it, end);
00184     }
00185
00186     template <typename octet_iterator>
00187     utfchar32_t prior(octet_iterator& it, octet_iterator start)
00188     {
00189         // can't do much if it == start
00190         if (it == start)
00191             throw not_enough_room();
00192
00193         octet_iterator end = it;
00194         // Go back until we hit either a lead octet or start
00195         while (utf8::internal::is_trail(*(--it)))
00196             if (it == start)
00197                 throw invalid_utf8(*it); // error - no lead byte in the sequence
00198         return utf8::peek_next(it, end);
00199     }
00200
00201     template <typename octet_iterator, typename distance_type>
```

```
00202     void advance (octet_iterator& it, distance_type n, octet_iterator end)
00203     {
00204         const distance_type zero(0);
00205         if (n < zero) {
00206             // backward
00207             for (distance_type i = n; i < zero; ++i)
00208                 utf8::prior(it, end);
00209         } else {
00210             // forward
00211             for (distance_type i = zero; i < n; ++i)
00212                 utf8::next(it, end);
00213         }
00214     }
00215
00216     template <typename octet_iterator>
00217     typename std::iterator_traits<octet_iterator>::difference_type
00218     distance (octet_iterator first, octet_iterator last)
00219     {
00220         typename std::iterator_traits<octet_iterator>::difference_type dist;
00221         for (dist = 0; first < last; ++dist)
00222             utf8::next(first, last);
00223         return dist;
00224     }
00225
00226     template <typename u16bit_iterator, typename octet_iterator>
00227     octet_iterator utf16to8 (u16bit_iterator start, u16bit_iterator end, octet_iterator result)
00228     {
00229         while (start != end) {
00230             utfchar32_t cp = utf8::internal::mask16(*start++);
00231             // Take care of surrogate pairs first
00232             if (utf8::internal::is_lead_surrogate(cp)) {
00233                 if (start != end) {
00234                     const utfchar32_t trail_surrogate = utf8::internal::mask16(*start++);
00235                     if (utf8::internal::is_trail_surrogate(trail_surrogate))
00236                         cp = (cp « 10) + trail_surrogate + internal::SURROGATE_OFFSET;
00237                     else
00238                         throw invalid_utf16(static_cast<utfchar16_t>(trail_surrogate));
00239                 }
00240                 else
00241                     throw invalid_utf16(static_cast<utfchar16_t>(cp));
00242
00243             }
00244             // Lone trail surrogate
00245             else if (utf8::internal::is_trail_surrogate(cp))
00246                 throw invalid_utf16(static_cast<utfchar16_t>(cp));
00247
00248             result = utf8::append(cp, result);
00249         }
00250         return result;
00251     }
00252
00253     template <typename u16bit_iterator, typename octet_iterator>
00254     u16bit_iterator utf8to16 (octet_iterator start, octet_iterator end, u16bit_iterator result)
00255     {
00256         while (start < end) {
00257             const utfchar32_t cp = utf8::next(start, end);
00258             if (cp > 0xffff) { //make a surrogate pair
00259                 *result++ = static_cast<utfchar16_t>((cp » 10)   + internal::LEAD_OFFSET);
00260                 *result++ = static_cast<utfchar16_t>((cp & 0x3ff) + internal::TRAIL_SURROGATE_MIN);
00261             }
00262             else
00263                 *result++ = static_cast<utfchar16_t>(cp);
00264         }
00265         return result;
00266     }
00267
00268     template <typename octet_iterator, typename u32bit_iterator>
00269     octet_iterator utf32to8 (u32bit_iterator start, u32bit_iterator end, octet_iterator result)
00270     {
00271         while (start != end)
00272             result = utf8::append(*(start++), result);
00273
00274         return result;
00275     }
00276
00277     template <typename octet_iterator, typename u32bit_iterator>
00278     u32bit_iterator utf8to32 (octet_iterator start, octet_iterator end, u32bit_iterator result)
00279     {
00280         while (start < end)
00281             (*result++) = utf8::next(start, end);
00282
00283         return result;
00284     }
00285
00286     // The iterator class
00287     template <typename octet_iterator>
00288     class iterator {
```

```
00289       octet_iterator it;
00290       octet_iterator range_start;
00291       octet_iterator range_end;
00292       public:
00293       typedef utfchar32_t value_type;
00294       typedef utfchar32_t* pointer;
00295       typedef utfchar32_t& reference;
00296       typedef std::ptrdiff_t difference_type;
00297       typedef std::bidirectional_iterator_tag iterator_category;
00298       iterator () {}
00299       explicit iterator (const octet_iterator& octet_it,
00300                          const octet_iterator& rangestart,
00301                          const octet_iterator& rangeend) :
00302               it(octet_it), range_start(rangestart), range_end(rangeend)
00303       {
00304           if (it < range_start || it > range_end)
00305               throw std::out_of_range("Invalid utf-8 iterator position");
00306       }
00307       // the default "big three" are OK
00308       octet_iterator base () const { return it; }
00309       utfchar32_t operator * () const
00310       {
00311           octet_iterator temp = it;
00312           return utf8::next(temp, range_end);
00313       }
00314       bool operator == (const iterator& rhs) const
00315       {
00316           if (range_start != rhs.range_start || range_end != rhs.range_end)
00317               throw std::logic_error("Comparing utf-8 iterators defined with different ranges");
00318           return (it == rhs.it);
00319       }
00320       bool operator != (const iterator& rhs) const
00321       {
00322           return !(operator == (rhs));
00323       }
00324       iterator& operator ++ ()
00325       {
00326           utf8::next(it, range_end);
00327           return *this;
00328       }
00329       iterator operator ++ (int)
00330       {
00331           iterator temp = *this;
00332           utf8::next(it, range_end);
00333           return temp;
00334       }
00335       iterator& operator -- ()
00336       {
00337           utf8::prior(it, range_start);
00338           return *this;
00339       }
00340       iterator operator -- (int)
00341       {
00342           iterator temp = *this;
00343           utf8::prior(it, range_start);
00344           return temp;
00345       }
00346     }; // class iterator
00347
00348 } // namespace utf8
00349
00350 #if UTF_CPP_CPLUSPLUS >= 202002L // C++ 20 or later
00351 #include "cpp20.h"
00352 #elif UTF_CPP_CPLUSPLUS >= 201703L // C++ 17 or later
00353 #include "cpp17.h"
00354 #elif UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
00355 #include "cpp11.h"
00356 #endif // C++ 11 or later
00357
00358 #endif //header guard
00359
```

5.23 core.h

```
00001 // Copyright 2006 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
```

```
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_CORE_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00029 #define UTF8_FOR_CPP_CORE_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00030
00031 #include <iterator>
00032 #include <cstring>
00033 #include <string>
00034
00035 // Determine the C++ standard version.
00036 // If the user defines UTF_CPP_CPLUSPLUS, use that.
00037 // Otherwise, trust the unreliable predefined macro __cplusplus
00038
00039 #if !defined UTF_CPP_CPLUSPLUS
00040     #define UTF_CPP_CPLUSPLUS __cplusplus
00041 #endif
00042
00043 #if UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
00044     #define UTF_CPP_OVERRIDE override
00045     #define UTF_CPP_NOEXCEPT noexcept
00046 #else // C++ 98/03
00047     #define UTF_CPP_OVERRIDE
00048     #define UTF_CPP_NOEXCEPT throw()
00049 #endif // C++ 11 or later
00050
00051
00052 namespace utf8
00053 {
00054 // The typedefs for 8-bit, 16-bit and 32-bit code units
00055 #if UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
00056     #if UTF_CPP_CPLUSPLUS >= 202002L // C++ 20 or later
00057         typedef char8_t         utfchar8_t;
00058     #else // C++ 11/14/17
00059         typedef unsigned char   utfchar8_t;
00060     #endif
00061     typedef char16_t        utfchar16_t;
00062     typedef char32_t        utfchar32_t;
00063 #else // C++ 98/03
00064     typedef unsigned char   utfchar8_t;
00065     typedef unsigned short  utfchar16_t;
00066     typedef unsigned int    utfchar32_t;
00067 #endif // C++ 11 or later
00068
00069 // Helper code - not intended to be directly called by the library users. May be changed at any time
00070 namespace internal
00071 {
00072     // Unicode constants
00073     // Leading (high) surrogates: 0xd800 - 0xdbff
00074     // Trailing (low) surrogates: 0xdc00 - 0xdfff
00075     const utfchar16_t LEAD_SURROGATE_MIN  = 0xd800u;
00076     const utfchar16_t LEAD_SURROGATE_MAX  = 0xdbffu;
00077     const utfchar16_t TRAIL_SURROGATE_MIN = 0xdc00u;
00078     const utfchar16_t TRAIL_SURROGATE_MAX = 0xdfffu;
00079     const utfchar16_t LEAD_OFFSET         = 0xd7c0u;       // LEAD_SURROGATE_MIN - (0x10000 » 10)
00080     const utfchar32_t SURROGATE_OFFSET    = 0xfca02400u;   // 0x10000u - (LEAD_SURROGATE_MIN « 10) -
      TRAIL_SURROGATE_MIN
00081
00082     // Maximum valid value for a Unicode code point
00083     const utfchar32_t CODE_POINT_MAX      = 0x0010ffffu;
00084
00085     template<typename octet_type>
00086     inline utfchar8_t mask8(octet_type oc)
00087     {
00088         return static_cast<utfchar8_t>(0xff & oc);
00089     }
00090     template<typename u16_type>
00091     inline utfchar16_t mask16(u16_type oc)
00092     {
00093         return static_cast<utfchar16_t>(0xffff & oc);
00094     }
00095
```

```
00096     template<typename octet_type>
00097     inline bool is_trail(octet_type oc)
00098     {
00099         return ((utf8::internal::mask8(oc) » 6) == 0x2);
00100     }
00101
00102     inline bool is_lead_surrogate(utfchar32_t cp)
00103     {
00104         return (cp >= LEAD_SURROGATE_MIN && cp <= LEAD_SURROGATE_MAX);
00105     }
00106
00107     inline bool is_trail_surrogate(utfchar32_t cp)
00108     {
00109         return (cp >= TRAIL_SURROGATE_MIN && cp <= TRAIL_SURROGATE_MAX);
00110     }
00111
00112     inline bool is_surrogate(utfchar32_t cp)
00113     {
00114         return (cp >= LEAD_SURROGATE_MIN && cp <= TRAIL_SURROGATE_MAX);
00115     }
00116
00117     inline bool is_code_point_valid(utfchar32_t cp)
00118     {
00119         return (cp <= CODE_POINT_MAX && !utf8::internal::is_surrogate(cp));
00120     }
00121
00122     inline bool is_in_bmp(utfchar32_t cp)
00123     {
00124         return cp < utfchar32_t(0x10000);
00125     }
00126
00127     template <typename octet_iterator>
00128     int sequence_length(octet_iterator lead_it)
00129     {
00130         const utfchar8_t lead = utf8::internal::mask8(*lead_it);
00131         if (lead < 0x80)
00132             return 1;
00133         else if ((lead » 5) == 0x6)
00134             return 2;
00135         else if ((lead » 4) == 0xe)
00136             return 3;
00137         else if ((lead » 3) == 0x1e)
00138             return 4;
00139         else
00140             return 0;
00141     }
00142
00143     inline bool is_overlong_sequence(utfchar32_t cp, int length)
00144     {
00145         if (cp < 0x80) {
00146             if (length != 1)
00147                 return true;
00148         }
00149         else if (cp < 0x800) {
00150             if (length != 2)
00151                 return true;
00152         }
00153         else if (cp < 0x10000) {
00154             if (length != 3)
00155                 return true;
00156         }
00157         return false;
00158     }
00159
00160     enum utf_error {UTF8_OK, NOT_ENOUGH_ROOM, INVALID_LEAD, INCOMPLETE_SEQUENCE, OVERLONG_SEQUENCE,
      INVALID_CODE_POINT};
00161
00163     template <typename octet_iterator>
00164     utf_error increase_safely(octet_iterator& it, const octet_iterator end)
00165     {
00166         if (++it == end)
00167             return NOT_ENOUGH_ROOM;
00168
00169         if (!utf8::internal::is_trail(*it))
00170             return INCOMPLETE_SEQUENCE;
00171
00172         return UTF8_OK;
00173     }
00174
00175     #define UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(IT, END) {utf_error ret = increase_safely(IT, END);
      if (ret != UTF8_OK) return ret;}
00176
00178     template <typename octet_iterator>
00179     utf_error get_sequence_1(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
00180     {
00181         if (it == end)
00182             return NOT_ENOUGH_ROOM;
```

```
00183
00184         code_point = utf8::internal::mask8(*it);
00185
00186         return UTF8_OK;
00187     }
00188
00189     template <typename octet_iterator>
00190     utf_error get_sequence_2(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
00191     {
00192         if (it == end)
00193             return NOT_ENOUGH_ROOM;
00194
00195         code_point = utf8::internal::mask8(*it);
00196
00197         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00198
00199         code_point = ((code_point << 6) & 0x7ff) + ((*it) & 0x3f);
00200
00201         return UTF8_OK;
00202     }
00203
00204     template <typename octet_iterator>
00205     utf_error get_sequence_3(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
00206     {
00207         if (it == end)
00208             return NOT_ENOUGH_ROOM;
00209
00210         code_point = utf8::internal::mask8(*it);
00211
00212         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00213
00214         code_point = ((code_point << 12) & 0xffff) + ((utf8::internal::mask8(*it) << 6) & 0xfff);
00215
00216         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00217
00218         code_point = static_cast<utfchar32_t>(code_point + ((*it) & 0x3f));
00219
00220         return UTF8_OK;
00221     }
00222
00223     template <typename octet_iterator>
00224     utf_error get_sequence_4(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
00225     {
00226         if (it == end)
00227            return NOT_ENOUGH_ROOM;
00228
00229         code_point = utf8::internal::mask8(*it);
00230
00231         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00232
00233         code_point = ((code_point << 18) & 0x1fffff) + ((utf8::internal::mask8(*it) << 12) & 0x3ffff);
00234
00235         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00236
00237         code_point = static_cast<utfchar32_t>(code_point + ((utf8::internal::mask8(*it) << 6) &
      0xfff));
00238
00239         UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)
00240
00241         code_point = static_cast<utfchar32_t>(code_point + ((*it) & 0x3f));
00242
00243         return UTF8_OK;
00244     }
00245
00246     #undef UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR
00247
00248     template <typename octet_iterator>
00249     utf_error validate_next(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
00250     {
00251         if (it == end)
00252             return NOT_ENOUGH_ROOM;
00253
00254         // Save the original value of it so we can go back in case of failure
00255         // Of course, it does not make much sense with i.e. stream iterators
00256         octet_iterator original_it = it;
00257
00258         utfchar32_t cp = 0;
00259         // Determine the sequence length based on the lead octet
00260         const int length = utf8::internal::sequence_length(it);
00261
00262         // Get trail octets and calculate the code point
00263         utf_error err = UTF8_OK;
00264         switch (length) {
00265             case 0:
00266                 return INVALID_LEAD;
00267             case 1:
00268                 err = utf8::internal::get_sequence_1(it, end, cp);
```

```
00269                 break;
00270             case 2:
00271                 err = utf8::internal::get_sequence_2(it, end, cp);
00272             break;
00273             case 3:
00274                 err = utf8::internal::get_sequence_3(it, end, cp);
00275             break;
00276             case 4:
00277                 err = utf8::internal::get_sequence_4(it, end, cp);
00278             break;
00279         }
00280 
00281         if (err == UTF8_OK) {
00282             // Decoding succeeded. Now, security checks...
00283             if (utf8::internal::is_code_point_valid(cp)) {
00284                 if (!utf8::internal::is_overlong_sequence(cp, length)){
00285                     // Passed! Return here.
00286                     code_point = cp;
00287                     ++it;
00288                     return UTF8_OK;
00289                 }
00290                 else
00291                     err = OVERLONG_SEQUENCE;
00292             }
00293             else
00294                 err = INVALID_CODE_POINT;
00295         }
00296 
00297         // Failure branch - restore the original value of the iterator
00298         it = original_it;
00299         return err;
00300     }
00301 
00302     template <typename octet_iterator>
00303     inline utf_error validate_next(octet_iterator& it, octet_iterator end) {
00304         utfchar32_t ignored;
00305         return utf8::internal::validate_next(it, end, ignored);
00306     }
00307 
00308     template <typename word_iterator>
00309     utf_error validate_next16(word_iterator& it, word_iterator end, utfchar32_t& code_point)
00310     {
00311         if (it == end)
00312             return NOT_ENOUGH_ROOM;
00313         // Save the original value of it so we can go back in case of failure
00314         // Of course, it does not make much sense with i.e. stream iterators
00315         word_iterator original_it = it;
00316 
00317         utf_error err = UTF8_OK;
00318 
00319         const utfchar16_t first_word = *it++;
00320         if (!is_surrogate(first_word)) {
00321             code_point = first_word;
00322             return UTF8_OK;
00323         }
00324         else {
00325             if (it == end)
00326                 err = NOT_ENOUGH_ROOM;
00327             else if (is_lead_surrogate(first_word)) {
00328                 const utfchar16_t second_word = *it++;
00329                 if (is_trail_surrogate(second_word)) {
00330                     code_point = static_cast<utfchar32_t>(first_word « 10) + second_word +
    SURROGATE_OFFSET;
00331                     return UTF8_OK;
00332                 } else
00333                     err = INCOMPLETE_SEQUENCE;
00334 
00335             } else {
00336                 err = INVALID_LEAD;
00337             }
00338         }
00339         // error branch
00340         it = original_it;
00341         return err;
00342     }
00343 
00344     // Internal implementation of both checked and unchecked append() function
00345     // This function will be invoked by the overloads below, as they will know
00346     // the octet_type.
00347     template <typename octet_iterator, typename octet_type>
00348     octet_iterator append(utfchar32_t cp, octet_iterator result) {
00349         if (cp < 0x80)                        // one octet
00350             *(result++) = static_cast<octet_type>(cp);
00351         else if (cp < 0x800) {                // two octets
00352             *(result++) = static_cast<octet_type>((cp » 6)          | 0xc0);
00353             *(result++) = static_cast<octet_type>((cp & 0x3f)        | 0x80);
00354         }
```

```
00355         else if (cp < 0x10000) {              // three octets
00356             *(result++) = static_cast<octet_type>((cp » 12)          | 0xe0);
00357             *(result++) = static_cast<octet_type>(((cp » 6) & 0x3f) | 0x80);
00358             *(result++) = static_cast<octet_type>((cp & 0x3f)         | 0x80);
00359         }
00360         else {                                // four octets
00361             *(result++) = static_cast<octet_type>((cp » 18)          | 0xf0);
00362             *(result++) = static_cast<octet_type>(((cp » 12) & 0x3f)| 0x80);
00363             *(result++) = static_cast<octet_type>(((cp » 6) & 0x3f) | 0x80);
00364             *(result++) = static_cast<octet_type>((cp & 0x3f)         | 0x80);
00365         }
00366         return result;
00367     }
00368
00369     // One of the following overloads will be invoked from the API calls
00370
00371     // A simple (but dangerous) case: the caller appends byte(s) to a char array
00372     inline char* append(utfchar32_t cp, char* result) {
00373         return append<char*, char>(cp, result);
00374     }
00375
00376     // Hopefully, most common case: the caller uses back_inserter
00377     // i.e. append(cp, std::back_inserter(str));
00378     template<typename container_type>
00379     std::back_insert_iterator<container_type> append
00380             (utfchar32_t cp, std::back_insert_iterator<container_type> result) {
00381         return append<std::back_insert_iterator<container_type>,
00382             typename container_type::value_type>(cp, result);
00383     }
00384
00385     // The caller uses some other kind of output operator - not covered above
00386     // Note that in this case we are not able to determine octet_type
00387     // so we assume it's utfchar8_t; that can cause a conversion warning if we are wrong.
00388     template <typename octet_iterator>
00389     octet_iterator append(utfchar32_t cp, octet_iterator result) {
00390         return append<octet_iterator, utfchar8_t>(cp, result);
00391     }
00392
00393     // Internal implementation of both checked and unchecked append16() function
00394     // This function will be invoked by the overloads below, as they will know
00395     // the word_type.
00396     template <typename word_iterator, typename word_type>
00397     word_iterator append16(utfchar32_t cp, word_iterator result) {
00398         if (is_in_bmp(cp))
00399             *(result++) = static_cast<word_type>(cp);
00400         else {
00401             // Code points from the supplementary planes are encoded via surrogate pairs
00402             *(result++) = static_cast<word_type>(LEAD_OFFSET + (cp » 10));
00403             *(result++) = static_cast<word_type>(TRAIL_SURROGATE_MIN + (cp & 0x3FF));
00404         }
00405         return result;
00406     }
00407
00408     // Hopefully, most common case: the caller uses back_inserter
00409     // i.e. append16(cp, std::back_inserter(str));
00410     template<typename container_type>
00411     std::back_insert_iterator<container_type> append16
00412             (utfchar32_t cp, std::back_insert_iterator<container_type> result) {
00413         return append16<std::back_insert_iterator<container_type>,
00414             typename container_type::value_type>(cp, result);
00415     }
00416
00417     // The caller uses some other kind of output operator - not covered above
00418     // Note that in this case we are not able to determine word_type
00419     // so we assume it's utfchar16_t; that can cause a conversion warning if we are wrong.
00420     template <typename word_iterator>
00421     word_iterator append16(utfchar32_t cp, word_iterator result) {
00422         return append16<word_iterator, utfchar16_t>(cp, result);
00423     }
00424
00425 } // namespace internal
00426
00428
00429     // Byte order mark
00430     const utfchar8_t bom[] = {0xef, 0xbb, 0xbf};
00431
00432     template <typename octet_iterator>
00433     octet_iterator find_invalid(octet_iterator start, octet_iterator end)
00434     {
00435         octet_iterator result = start;
00436         while (result != end) {
00437             utf8::internal::utf_error err_code = utf8::internal::validate_next(result, end);
00438             if (err_code != internal::UTF8_OK)
00439                 return result;
00440         }
00441         return result;
00442     }
```

```
00443
00444     inline const char* find_invalid(const char* str)
00445     {
00446         const char* end = str + std::strlen(str);
00447         return find_invalid(str, end);
00448     }
00449
00450     inline std::size_t find_invalid(const std::string& s)
00451     {
00452         std::string::const_iterator invalid = find_invalid(s.begin(), s.end());
00453         return (invalid == s.end()) ? std::string::npos : static_cast<std::size_t>(invalid -
      s.begin());
00454     }
00455
00456     template <typename octet_iterator>
00457     inline bool is_valid(octet_iterator start, octet_iterator end)
00458     {
00459         return (utf8::find_invalid(start, end) == end);
00460     }
00461
00462     inline bool is_valid(const char* str)
00463     {
00464         return (*(utf8::find_invalid(str)) == '\0');
00465     }
00466
00467     inline bool is_valid(const std::string& s)
00468     {
00469         return is_valid(s.begin(), s.end());
00470     }
00471
00472
00473
00474     template <typename octet_iterator>
00475     inline bool starts_with_bom (octet_iterator it, octet_iterator end)
00476     {
00477         return (
00478             ((it != end) && (utf8::internal::mask8(*it++)) == bom[0]) &&
00479             ((it != end) && (utf8::internal::mask8(*it++)) == bom[1]) &&
00480             ((it != end) && (utf8::internal::mask8(*it))   == bom[2])
00481            );
00482     }
00483
00484     inline bool starts_with_bom(const std::string& s)
00485     {
00486         return starts_with_bom(s.begin(), s.end());
00487     }
00488 } // namespace utf8
00489
00490 #endif // header guard
00491
00492
```

5.24 cpp11.h

```
00001 // Copyright 2018 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_a184c22c_d012_11e8_a8d5_f2801f1b9fd1
00029 #define UTF8_FOR_CPP_a184c22c_d012_11e8_a8d5_f2801f1b9fd1
```

```
00030
00031 #include "checked.h"
00032
00033 namespace utf8
00034 {
00035     inline void append16(utfchar32_t cp, std::u16string& s)
00036     {
00037         append16(cp, std::back_inserter(s));
00038     }
00039
00040     inline std::string utf16to8(const std::u16string& s)
00041     {
00042         std::string result;
00043         utf16to8(s.begin(), s.end(), std::back_inserter(result));
00044         return result;
00045     }
00046
00047     inline std::u16string utf8to16(const std::string& s)
00048     {
00049         std::u16string result;
00050         utf8to16(s.begin(), s.end(), std::back_inserter(result));
00051         return result;
00052     }
00053
00054     inline std::string utf32to8(const std::u32string& s)
00055     {
00056         std::string result;
00057         utf32to8(s.begin(), s.end(), std::back_inserter(result));
00058         return result;
00059     }
00060
00061     inline std::u32string utf8to32(const std::string& s)
00062     {
00063         std::u32string result;
00064         utf8to32(s.begin(), s.end(), std::back_inserter(result));
00065         return result;
00066     }
00067 } // namespace utf8
00068
00069 #endif // header guard
00070
```

5.25 cpp17.h

```
00001 // Copyright 2018 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_7e906c01_03a3_4daf_b420_ea7ea952b3c9
00029 #define UTF8_FOR_CPP_7e906c01_03a3_4daf_b420_ea7ea952b3c9
00030
00031 #include "cpp11.h"
00032
00033 namespace utf8
00034 {
00035     inline std::string utf16to8(std::u16string_view s)
00036     {
00037         std::string result;
00038         utf16to8(s.begin(), s.end(), std::back_inserter(result));
00039         return result;
```

```
00040     }
00041
00042     inline std::u16string utf8to16(std::string_view s)
00043     {
00044         std::u16string result;
00045         utf8to16(s.begin(), s.end(), std::back_inserter(result));
00046         return result;
00047     }
00048
00049     inline std::string utf32to8(std::u32string_view s)
00050     {
00051         std::string result;
00052         utf32to8(s.begin(), s.end(), std::back_inserter(result));
00053         return result;
00054     }
00055
00056     inline std::u32string utf8to32(std::string_view s)
00057     {
00058         std::u32string result;
00059         utf8to32(s.begin(), s.end(), std::back_inserter(result));
00060         return result;
00061     }
00062
00063     inline std::size_t find_invalid(std::string_view s)
00064     {
00065         std::string_view::const_iterator invalid = find_invalid(s.begin(), s.end());
00066         return (invalid == s.end()) ? std::string_view::npos : static_cast<std::size_t>(invalid -
    s.begin());
00067     }
00068
00069     inline bool is_valid(std::string_view s)
00070     {
00071         return is_valid(s.begin(), s.end());
00072     }
00073
00074     inline std::string replace_invalid(std::string_view s, char32_t replacement)
00075     {
00076         std::string result;
00077         replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
00078         return result;
00079     }
00080
00081     inline std::string replace_invalid(std::string_view s)
00082     {
00083         std::string result;
00084         replace_invalid(s.begin(), s.end(), std::back_inserter(result));
00085         return result;
00086     }
00087
00088     inline bool starts_with_bom(std::string_view s)
00089     {
00090         return starts_with_bom(s.begin(), s.end());
00091     }
00092
00093 } // namespace utf8
00094
00095 #endif // header guard
00096
```

5.26 cpp20.h

```
00001 // Copyright 2022 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
```

```
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_207e906c01_03a3_4daf_b420_ea7ea952b3c9
00029 #define UTF8_FOR_CPP_207e906c01_03a3_4daf_b420_ea7ea952b3c9
00030
00031 #include "cpp17.h"
00032
00033 namespace utf8
00034 {
00035     inline std::u8string utf16tou8(const std::u16string& s)
00036     {
00037         std::u8string result;
00038         utf16to8(s.begin(), s.end(), std::back_inserter(result));
00039         return result;
00040     }
00041
00042     inline std::u8string utf16tou8(std::u16string_view s)
00043     {
00044         std::u8string result;
00045         utf16to8(s.begin(), s.end(), std::back_inserter(result));
00046         return result;
00047     }
00048
00049     inline std::u16string utf8to16(const std::u8string& s)
00050     {
00051         std::u16string result;
00052         utf8to16(s.begin(), s.end(), std::back_inserter(result));
00053         return result;
00054     }
00055
00056     inline std::u16string utf8to16(const std::u8string_view& s)
00057     {
00058         std::u16string result;
00059         utf8to16(s.begin(), s.end(), std::back_inserter(result));
00060         return result;
00061     }
00062
00063     inline std::u8string utf32tou8(const std::u32string& s)
00064     {
00065         std::u8string result;
00066         utf32to8(s.begin(), s.end(), std::back_inserter(result));
00067         return result;
00068     }
00069
00070     inline std::u8string utf32tou8(const std::u32string_view& s)
00071     {
00072         std::u8string result;
00073         utf32to8(s.begin(), s.end(), std::back_inserter(result));
00074         return result;
00075     }
00076
00077     inline std::u32string utf8to32(const std::u8string& s)
00078     {
00079         std::u32string result;
00080         utf8to32(s.begin(), s.end(), std::back_inserter(result));
00081         return result;
00082     }
00083
00084     inline std::u32string utf8to32(const std::u8string_view& s)
00085     {
00086         std::u32string result;
00087         utf8to32(s.begin(), s.end(), std::back_inserter(result));
00088         return result;
00089     }
00090
00091     inline std::size_t find_invalid(const std::u8string& s)
00092     {
00093         std::u8string::const_iterator invalid = find_invalid(s.begin(), s.end());
00094         return (invalid == s.end()) ? std::string_view::npos : static_cast<std::size_t>(invalid -
      s.begin());
00095     }
00096
00097     inline bool is_valid(const std::u8string& s)
00098     {
00099         return is_valid(s.begin(), s.end());
00100     }
00101
00102     inline std::u8string replace_invalid(const std::u8string& s, char32_t replacement)
00103     {
00104         std::u8string result;
00105         replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
00106         return result;
00107     }
00108
```

```
00109     inline std::u8string replace_invalid(const std::u8string& s)
00110     {
00111         std::u8string result;
00112         replace_invalid(s.begin(), s.end(), std::back_inserter(result));
00113         return result;
00114     }
00115
00116     inline bool starts_with_bom(const std::u8string& s)
00117     {
00118         return starts_with_bom(s.begin(), s.end());
00119     }
00120
00121 } // namespace utf8
00122
00123 #endif // header guard
00124
```

5.27 unchecked.h

```
00001 // Copyright 2006 Nemanja Trifunovic
00002
00003 /*
00004 Permission is hereby granted, free of charge, to any person or organization
00005 obtaining a copy of the software and accompanying documentation covered by
00006 this license (the "Software") to use, reproduce, display, distribute,
00007 execute, and transmit the Software, and to prepare derivative works of the
00008 Software, and to permit third-parties to whom the Software is furnished to
00009 do so, all subject to the following:
00010
00011 The copyright notices in the Software and this entire statement, including
00012 the above license grant, this restriction and the following disclaimer,
00013 must be included in all copies of the Software, in whole or in part, and
00014 all derivative works of the Software, unless such copies or derivative
00015 works are solely in the form of machine-executable object code generated by
00016 a source language processor.
00017
00018 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00019 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00020 FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
00021 SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
00022 FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
00023 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
00024 DEALINGS IN THE SOFTWARE.
00025 */
00026
00027
00028 #ifndef UTF8_FOR_CPP_UNCHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00029 #define UTF8_FOR_CPP_UNCHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
00030
00031 #include "core.h"
00032
00033 namespace utf8
00034 {
00035     namespace unchecked
00036     {
00037         template <typename octet_iterator>
00038         octet_iterator append(utfchar32_t cp, octet_iterator result)
00039         {
00040             return internal::append(cp, result);
00041         }
00042
00043         template <typename word_iterator>
00044         word_iterator append16(utfchar32_t cp, word_iterator result)
00045         {
00046             return internal::append16(cp, result);
00047         }
00048
00049         template <typename octet_iterator, typename output_iterator>
00050         output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out,
      utfchar32_t replacement)
00051         {
00052             while (start != end) {
00053                 octet_iterator sequence_start = start;
00054                 internal::utf_error err_code = utf8::internal::validate_next(start, end);
00055                 switch (err_code) {
00056                     case internal::UTF8_OK :
00057                         for (octet_iterator it = sequence_start; it != start; ++it)
00058                             *out++ = *it;
00059                         break;
00060                     case internal::NOT_ENOUGH_ROOM:
00061                         out = utf8::unchecked::append(replacement, out);
00062                         start = end;
00063                         break;
```

```
00064                     case internal::INVALID_LEAD:
00065                         out = utf8::unchecked::append(replacement, out);
00066                         ++start;
00067                         break;
00068                     case internal::INCOMPLETE_SEQUENCE:
00069                     case internal::OVERLONG_SEQUENCE:
00070                     case internal::INVALID_CODE_POINT:
00071                         out = utf8::unchecked::append(replacement, out);
00072                         ++start;
00073                         // just one replacement mark for the sequence
00074                         while (start != end && utf8::internal::is_trail(*start))
00075                             ++start;
00076                         break;
00077                 }
00078             }
00079             return out;
00080         }
00081
00082         template <typename octet_iterator, typename output_iterator>
00083         inline output_iterator replace_invalid(octet_iterator start, octet_iterator end,
      output_iterator out)
00084         {
00085             static const utfchar32_t replacement_marker = utf8::internal::mask16(0xfffd);
00086             return utf8::unchecked::replace_invalid(start, end, out, replacement_marker);
00087         }
00088
00089         inline std::string replace_invalid(const std::string& s, utfchar32_t replacement)
00090         {
00091             std::string result;
00092             replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
00093             return result;
00094         }
00095
00096         inline std::string replace_invalid(const std::string& s)
00097         {
00098             std::string result;
00099             replace_invalid(s.begin(), s.end(), std::back_inserter(result));
00100             return result;
00101         }
00102
00103         template <typename octet_iterator>
00104         utfchar32_t next(octet_iterator& it)
00105         {
00106             utfchar32_t cp = utf8::internal::mask8(*it);
00107             switch (utf8::internal::sequence_length(it)) {
00108                 case 1:
00109                     break;
00110                 case 2:
00111                     it++;
00112                     cp = ((cp << 6) & 0x7ff) + ((*it) & 0x3f);
00113                     break;
00114                 case 3:
00115                     ++it;
00116                     cp = ((cp << 12) & 0xffff) + ((utf8::internal::mask8(*it) << 6) & 0xfff);
00117                     ++it;
00118                     cp = static_cast<utfchar32_t>(cp + ((*it) & 0x3f));
00119                     break;
00120                 case 4:
00121                     ++it;
00122                     cp = ((cp << 18) & 0x1fffff) + ((utf8::internal::mask8(*it) << 12) & 0x3ffff);

00123                     ++it;
00124                     cp = static_cast<utfchar32_t>(cp + ((utf8::internal::mask8(*it) << 6) & 0xfff));
00125                     ++it;
00126                     cp = static_cast<utfchar32_t>(cp + ((*it) & 0x3f));
00127                     break;
00128             }
00129             ++it;
00130             return cp;
00131         }
00132
00133         template <typename octet_iterator>
00134         utfchar32_t peek_next(octet_iterator it)
00135         {
00136             return utf8::unchecked::next(it);
00137         }
00138
00139         template <typename word_iterator>
00140         utfchar32_t next16(word_iterator& it)
00141         {
00142             utfchar32_t cp = utf8::internal::mask16(*it++);
00143             if (utf8::internal::is_lead_surrogate(cp))
00144                 return (cp << 10) + *it++ + utf8::internal::SURROGATE_OFFSET;
00145             return cp;
00146         }
00147
00148         template <typename octet_iterator>
```

```
00149         utfchar32_t prior(octet_iterator& it)
00150         {
00151             while (utf8::internal::is_trail(*(--it))) ;
00152             octet_iterator temp = it;
00153             return utf8::unchecked::next(temp);
00154         }
00155 
00156         template <typename octet_iterator, typename distance_type>
00157         void advance(octet_iterator& it, distance_type n)
00158         {
00159             const distance_type zero(0);
00160             if (n < zero) {
00161                 // backward
00162                 for (distance_type i = n; i < zero; ++i)
00163                     utf8::unchecked::prior(it);
00164             } else {
00165                 // forward
00166                 for (distance_type i = zero; i < n; ++i)
00167                     utf8::unchecked::next(it);
00168             }
00169         }
00170 
00171         template <typename octet_iterator>
00172         typename std::iterator_traits<octet_iterator>::difference_type
00173         distance(octet_iterator first, octet_iterator last)
00174         {
00175             typename std::iterator_traits<octet_iterator>::difference_type dist;
00176             for (dist = 0; first < last; ++dist)
00177                 utf8::unchecked::next(first);
00178             return dist;
00179         }
00180 
00181         template <typename u16bit_iterator, typename octet_iterator>
00182         octet_iterator utf16to8(u16bit_iterator start, u16bit_iterator end, octet_iterator result)
00183         {
00184             while (start != end) {
00185                 utfchar32_t cp = utf8::internal::mask16(*start++);
00186                 // Take care of surrogate pairs first
00187                 if (utf8::internal::is_lead_surrogate(cp)) {
00188                     if (start == end)
00189                         return result;
00190                     utfchar32_t trail_surrogate = utf8::internal::mask16(*start++);
00191                     cp = (cp « 10) + trail_surrogate + internal::SURROGATE_OFFSET;
00192                 }
00193                 result = utf8::unchecked::append(cp, result);
00194             }
00195             return result;
00196         }
00197 
00198         template <typename u16bit_iterator, typename octet_iterator>
00199         u16bit_iterator utf8to16(octet_iterator start, octet_iterator end, u16bit_iterator result)
00200         {
00201             while (start < end) {
00202                 utfchar32_t cp = utf8::unchecked::next(start);
00203                 if (cp > 0xffff) { //make a surrogate pair
00204                     *result++ = static_cast<utfchar16_t>((cp » 10)   + internal::LEAD_OFFSET);
00205                     *result++ = static_cast<utfchar16_t>((cp & 0x3ff) +
      internal::TRAIL_SURROGATE_MIN);
00206                 }
00207                 else
00208                     *result++ = static_cast<utfchar16_t>(cp);
00209             }
00210             return result;
00211         }
00212 
00213         template <typename octet_iterator, typename u32bit_iterator>
00214         octet_iterator utf32to8(u32bit_iterator start, u32bit_iterator end, octet_iterator result)
00215         {
00216             while (start != end)
00217                 result = utf8::unchecked::append(*(start++), result);
00218 
00219             return result;
00220         }
00221 
00222         template <typename octet_iterator, typename u32bit_iterator>
00223         u32bit_iterator utf8to32(octet_iterator start, octet_iterator end, u32bit_iterator result)
00224         {
00225             while (start < end)
00226                 (*result++) = utf8::unchecked::next(start);
00227 
00228             return result;
00229         }
00230 
00231         // The iterator class
00232         template <typename octet_iterator>
00233           class iterator {
00234             octet_iterator it;
```

```
00235             public:
00236             typedef utfchar32_t value_type;
00237             typedef utfchar32_t* pointer;
00238             typedef utfchar32_t& reference;
00239             typedef std::ptrdiff_t difference_type;
00240             typedef std::bidirectional_iterator_tag iterator_category;
00241             iterator () {}
00242             explicit iterator (const octet_iterator& octet_it): it(octet_it) {}
00243             // the default "big three" are OK
00244             octet_iterator base () const { return it; }
00245             utfchar32_t operator * () const
00246             {
00247                 octet_iterator temp = it;
00248                 return utf8::unchecked::next(temp);
00249             }
00250             bool operator == (const iterator& rhs) const
00251             {
00252                 return (it == rhs.it);
00253             }
00254             bool operator != (const iterator& rhs) const
00255             {
00256                 return !(operator == (rhs));
00257             }
00258             iterator& operator ++ ()
00259             {
00260                 ::std::advance(it, utf8::internal::sequence_length(it));
00261                 return *this;
00262             }
00263             iterator operator ++ (int)
00264             {
00265                 iterator temp = *this;
00266                 ::std::advance(it, utf8::internal::sequence_length(it));
00267                 return temp;
00268             }
00269             iterator& operator -- ()
00270             {
00271                 utf8::unchecked::prior(it);
00272                 return *this;
00273             }
00274             iterator operator -- (int)
00275             {
00276                 iterator temp = *this;
00277                 utf8::unchecked::prior(it);
00278                 return temp;
00279             }
00280           }; // class iterator
00281
00282     } // namespace utf8::unchecked
00283 } // namespace utf8
00284
00285
00286 #endif // header guard
00287
```

5.28 StringHandler.hpp

```
00001 #ifndef STRING_HANDLER_HPP
00002 #define STRING_HANDLER_HPP
00003
00004 #include <string>
00005
00006 namespace StringHandler {
00011     template <typename Object>
00012     struct SetWidthAtLeft {
00013         const Object& obj;
00014         size_t width;
00015
00021         SetWidthAtLeft(const Object& o, size_t w)
00022             : obj(o), width(w) {}
00023     };
00024
00025
00032     template <typename Object>
00033     std::string toString(const Object& obj);
00034
00041     template <typename Object>
00042     size_t size(const Object& obj);
00043
00051     template <typename Object>
00052     std::ostream& operator<<(std::ostream& os, const SetWidthAtLeft<Object>& manip);
00053 }
00054
00055 #include "Utils/StringHandler.impl.hpp"
```

```
00056
00057 #endif
```

5.29 StringHandler.impl.hpp

```
00001 #include "Utils/StringHandler.hpp"
00002
00003 #include <sstream>
00004
00005 #include "utf8.h"
00006
00007 namespace StringHandler {
00008     template <typename Object>
00009     std::string toString(const Object& obj) {
00010         std::ostringstream oss;
00011         oss « obj;
00012         return oss.str();
00013     }
00014
00015     template <typename Object>
00016     size_t size(const Object& obj) {
00017         std::string str = toString(obj);
00018         size_t count = 0;
00019
00020         auto it = str.begin();
00021         auto end = str.end();
00022
00023         while (it != end) {
00024             utf8::next(it, end);
00025             count++;
00026         }
00027
00028         return count;
00029     }
00030
00031     template <typename Object>
00032     std::ostream& operator«(std::ostream& os, const SetWidthAtLeft<Object>& manip) {
00033         std::string str = toString(manip.obj);
00034
00035         size_t realSize = StringHandler::size(manip.obj),
00036             padding = (manip.width > realSize ? manip.width - realSize : 0);
00037
00038         os « str « std::string(padding, ' ');
00039
00040         return os;
00041     }
00042 }
```

Index