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

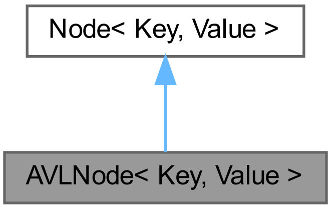
Class Documentation

4.1 AVLNode< Key, Value > Struct Template Reference

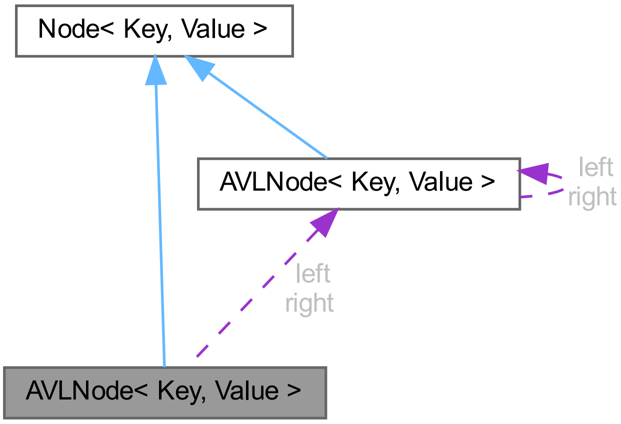
AVL tree node structure extending a generic Node.

```
#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:

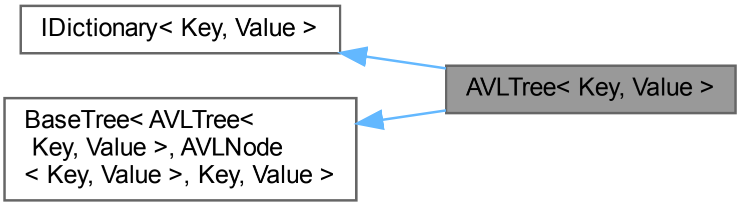
- include/Trees/AVL/AVLNode.hpp

4.2 AVLTree< Key, Value > Class Template Reference

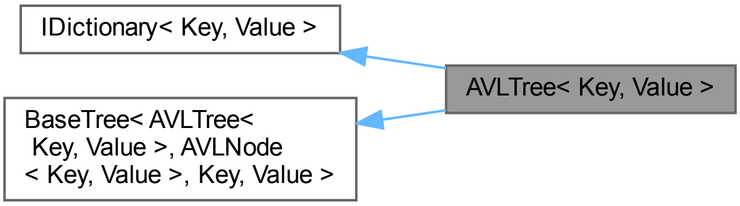
A class representing an AVL Tree.

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

Inheritance diagram for AVLTree< Key, Value >:


IDictionary< Key, Value >
BaseTree< AVLTree< Key, Value >, AVLNode < Key, Value >, Key, Value >
AVLTree< Key, Value >

Collaboration diagram for AVLTree< Key, Value >:


IDictionary< Key, Value >
BaseTree< AVLTree< Key, Value >, AVLNode < Key, Value >, Key, Value >
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.

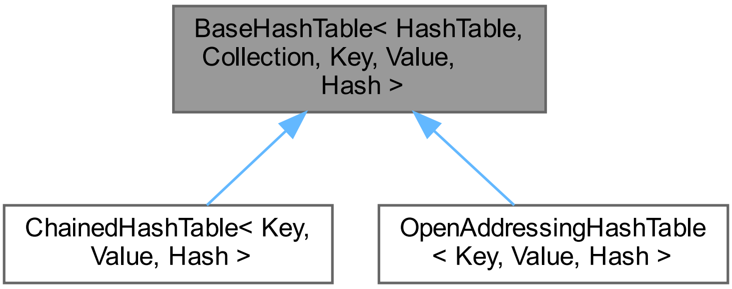
Implements IDictionary< Key, Value >.

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 >:


BaseHashTable< HashTable, Collection, Key, Value, Hash >
ChainedHashTable< Key, Value, Hash >
OpenAddressingHashTable < 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 `HashTable` 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:

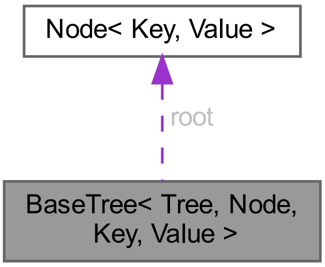
- include/HashTables/Base/BaseHashTable.hpp
- include/HashTables/Base/BaseHashTable.impl.hpp

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:

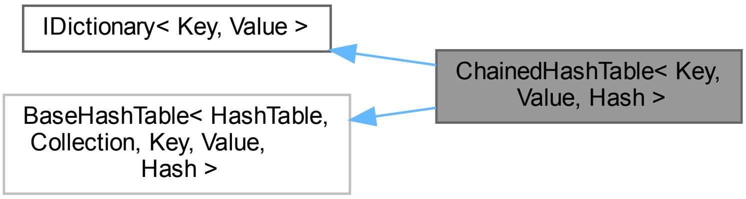
- include/Trees/Base/BaseTree.hpp
- include/Trees/Base/BaseTree.impl.hpp

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

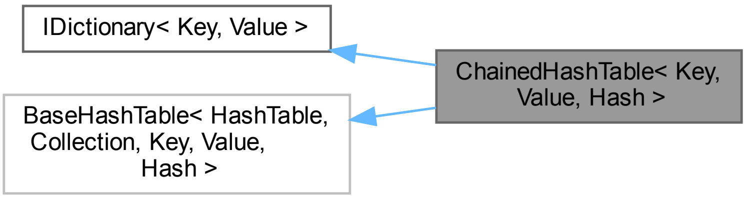
Hash table implementation using separate chaining.

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

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


IDictionary< Key, Value >
ChainedHashTable< Key, Value, Hash >
BaseHashTable< HashTable, Collection, Key, Value, Hash >

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


IDictionary< Key, Value >
ChainedHashTable< Key, Value, Hash >
BaseHashTable< HashTable, Collection, 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.
- void print () const

  Prints the contents of the hash table to the standard output.

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 `findPairIterator` 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 print()

```
template<typename Key , typename Value , typename Hash >
void ChainedHashTable< Key, Value, Hash >::print ( ) const
```

Prints the contents of the hash table to the standard output.

Each slot of the hash table is printed, showing the key-value pairs stored in that slot. If a slot is empty, it will display "Empty". This function is primarily used for debugging and visualization purposes.

4.5.2.10 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 >.

4.5.2.11 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`.

4.5.2.12 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 >.

4.5.2.13 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.

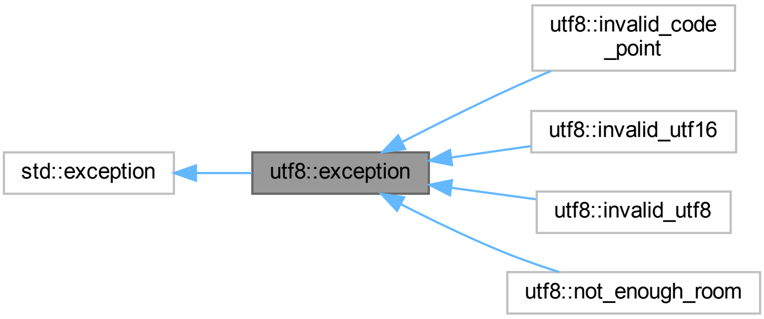
Implements IDictionary< Key, Value >.

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

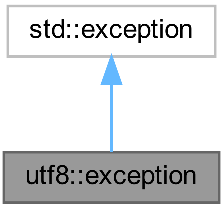
- include/HashTables/Chained/ChainedHashTable.hpp
- include/HashTables/Chained/ChainedHashTable.impl.hpp

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:

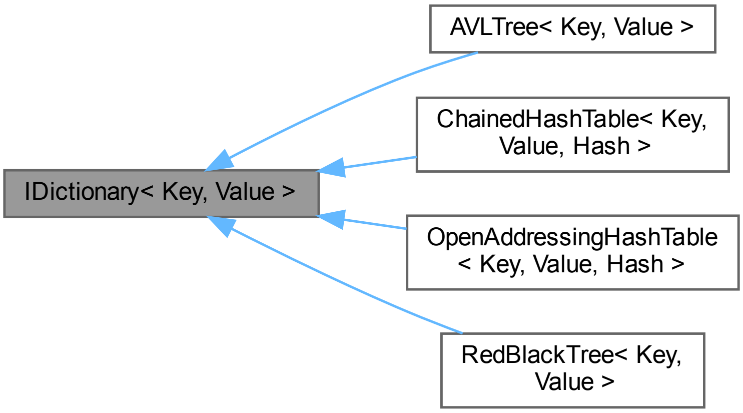
- include/utf8/checked.h

4.7 IDictionary< Key, Value > Class Template Reference

Interface for a generic dictionary data structure.

```
#include <IDictionary.hpp>
```

Inheritance diagram for IDictionary< Key, Value >:


AVLTree< Key, Value >
ChainedHashTable< Key, Value, Hash >
IDictionary< Key, Value >
OpenAddressingHashTable < Key, Value, Hash >
RedBlackTree< 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, Ha
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, Ha
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, Ha
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, Ha
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, Ha
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, Ha
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.

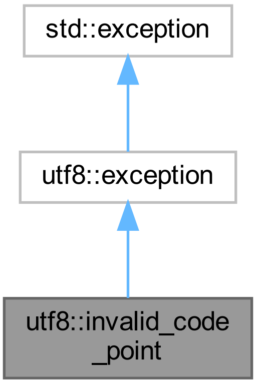
This mutable member variable is used to count the number of comparisons performed, allowing for performance analysis or debugging. Being mutable allows it to be modified even in const member functions.

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

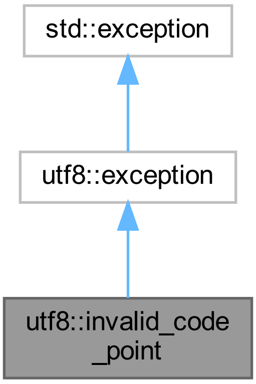
- include/Dictionary/IDictionary.hpp

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

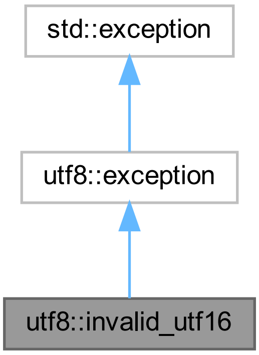
- **invalid_code_point** (utfchar32_t codepoint)
- virtual const char ∗ **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE
- utfchar32_t **code_point** () const

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

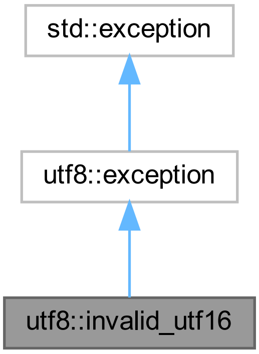
- include/utf8/checked.h

4.9 utf8::invalid_utf16 Class Reference

Inheritance diagram for utf8::invalid_utf16:



Collaboration diagram for utf8::invalid_utf16:



Public Member Functions

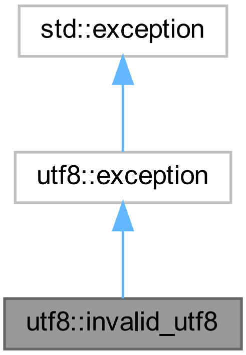
- **invalid_utf16** (utfchar16_t u)
- virtual const char ∗ **what** () const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE
- utfchar16_t **utf16_word** () const

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

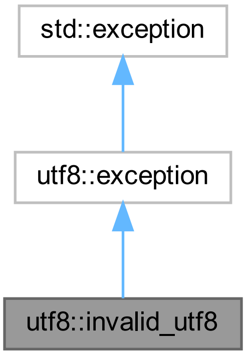
- include/utf8/checked.h

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:

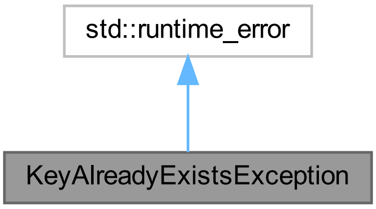
- include/utf8/unchecked.h

4.13 KeyAlreadyExistsException Class Reference

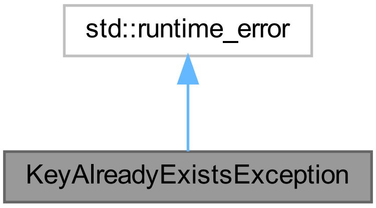
Exception thrown when attempting to insert a key that already exists in a dictionary.

```
#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:

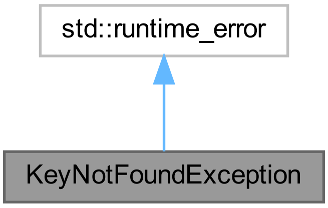
- include/Exceptions/KeyExceptions.hpp

4.14 KeyNotFoundException Class Reference

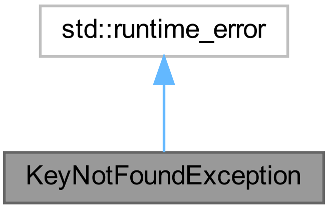
Exception thrown when a key is not found in a dictionary or map.

`#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:

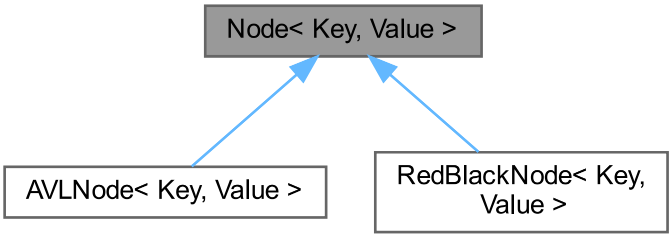
- include/Exceptions/KeyExceptions.hpp

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

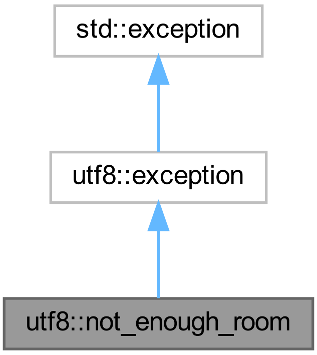
key	The new key to assign.
value	The new value to assign.

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

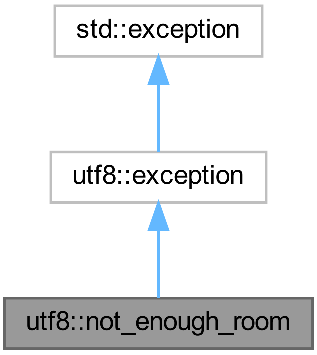
- include/Trees/Base/Node.hpp

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:

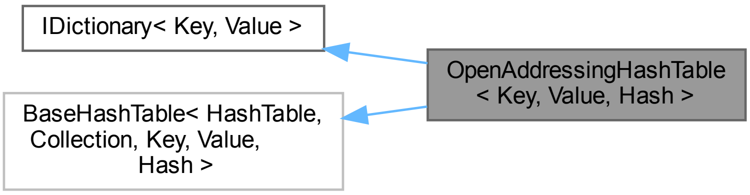
- include/utf8/checked.h

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

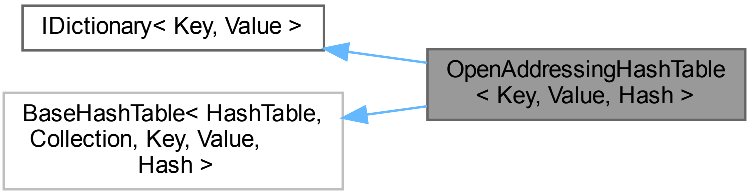
Hash table implementation using open addressing.

`#include <OpenAddressingHashTable.hpp>`

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


IDictionary< Key, Value >
BaseHashTable< HashTable, Collection, Key, Value, Hash >
OpenAddressingHashTable < Key, Value, Hash >

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


IDictionary< Key, Value >
BaseHashTable< HashTable, Collection, Key, Value, Hash >
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.
- void print () const

  Prints all slots in the hash table, including empty and deleted ones.

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 print()

```
template<typename Key , typename Value , typename Hash >
void OpenAddressingHashTable< Key, Value, Hash >::print ( ) const
```

Prints all slots in the hash table, including empty and deleted ones.

This function iterates through all slots in the hash table and prints their status (EMPTY, DELETED, or OCCUPIED) along with their key-value pairs if applicable.

Parameters

out	The output stream where the slot information will be printed.

4.17.3.10 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 >.

4.17.3.11 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.

4.17.3.12 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 >.

4.17.3.13 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:

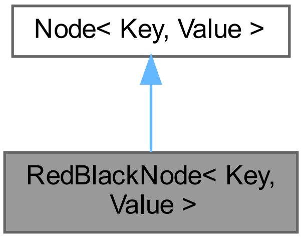
- include/HashTables/OpenAddressing/OpenAddressingHashTable.hpp
- include/HashTables/OpenAddressing/OpenAddressingHashTable.impl.hpp

4.18 RedBlackNode< Key, Value > Class Template Reference

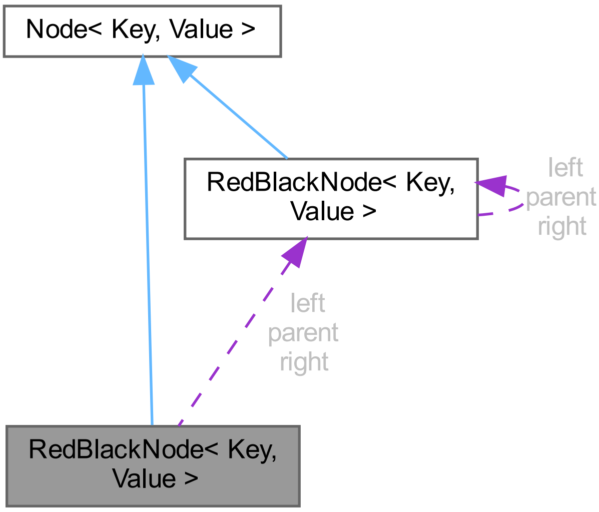
Represents a node in a Red-Black Tree.

```
#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:

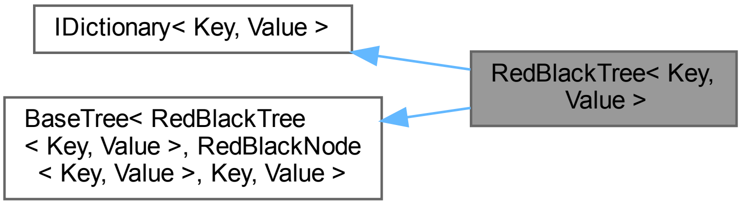
- include/Trees/RedBlack/RedBlackNode.hpp

4.19 RedBlackTree< Key, Value > Class Template Reference

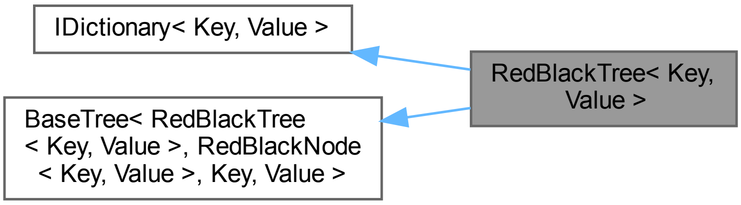
A class representing a Red-Black Tree.

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

Inheritance diagram for RedBlackTree< Key, Value >:


IDictionary< Key, Value >
BaseTree< RedBlackTree < Key, Value >, RedBlackNode < Key, Value >, Key, Value >
RedBlackTree< Key, Value >

Collaboration diagram for RedBlackTree< Key, Value >:


IDictionary< Key, Value >
BaseTree< RedBlackTree < Key, Value >, RedBlackNode < Key, Value >, Key, Value >
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

```
#ifndef IDICTIONARY_HPP
#define IDICTIONARY_HPP

#include "Trees/Base/BaseTree.hpp"

template<typename Key, typename Value>
class IDictionary {
protected:
    mutable size_t comparisonsCount = 0;

    void incrementCounter(size_t n) const { comparisonsCount += n; }

    void resetCounter() const { comparisonsCount = 0; }

public:
    virtual void insert(const Key& key, const Value& value) = 0;

    virtual bool find(const Key& key, Value& outValue) const = 0;

    virtual void update(const Key& key, const Value& value) = 0;

    virtual void remove(const Key& key) = 0;

    virtual void clear() = 0;

    virtual void printInOrder(std::ostream& out) const = 0;

    virtual size_t getComparisonsCount() const = 0;

    virtual Value& operator[](const Key& key) = 0;

    virtual const Value& operator[](const Key& key) const = 0;

    virtual ~IDictionary() = default;

    template <typename Tree, typename Node, typename K, typename V>
    friend class BaseTree;

    template <typename HashTable, typename Collection, typename K, typename V, typename Hash>
    friend class BaseHashTable;
};

#endif
```

5.2 KeyExceptions.hpp

```
#ifndef KEY_EXCEPTIONS_HPP
#define KEY_EXCEPTIONS_HPP

#include <stdexcept>
#include <string>

class KeyAlreadyExistsException : public std::runtime_error {
public:
    explicit KeyAlreadyExistsException()
        : std::runtime_error("Key already exists in the dictionary.") {}
};

class KeyNotFoundException : public std::runtime_error {
public:
    explicit KeyNotFoundException()
        : std::runtime_error("Key not found in the dictionary.") {}
};

#endif
```

5.3 BaseHashTable.hpp

```
#ifndef BASE_HASH_TABLE_HPP
#define BASE_HASH_TABLE_HPP

#include <vector>

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
class BaseHashTable {
protected:
    std::vector<Collection> table;

    size_t tableSize;

    float maxLoadFactor;

    size_t numberOfElements;

    Hash hashing;

    mutable size_t collisionsCount;


    size_t getNextPrime(size_t num) const;

    void checkAndRehash();
public:
    BaseHashTable(size_t size = 7, float mlf = 0.7);

    float getLoadFactor() const;

    void clearHashTable();

    void incrementCollisionsCount(size_t m = 1) const;
};

#include "HashTables/Base/BaseHashTable.impl.hpp"

#endif
```

5.4 BaseHashTable.impl.hpp

```
#include "HashTables/Base/BaseHashTable.hpp"

#include <cmath>

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
size_t BaseHashTable<HashTable, Collection, Key, Value, Hash>::getNextPrime(size_t num) const {
    auto isPrime = [&num](size_t x) -> bool {
        if (x <= 1) return false;
        if (x == 2 or x == 3) return true;
        if (x % 2 == 0) return false;

        for (int i = 3; i <= sqrt(x); i += 2) {
            if (x % i == 0) return false;
        }

        return true;
    };

    size_t candidate;
    if (num % 2 == 0) candidate = num + 1;
    else candidate = num + 2;
    while (true) {
        if (isPrime(candidate)) return candidate;
        candidate += 2;
    }
}

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
void BaseHashTable<HashTable, Collection, Key, Value, Hash>::checkAndRehash() {
    if (getLoadFactor() >= maxLoadFactor)
        static_cast<HashTable*>(this)->rehash(2 * tableSize);
}

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
BaseHashTable<HashTable, Collection, Key, Value, Hash>::BaseHashTable(size_t size, float mlf) {
    tableSize = size;
    table.resize(tableSize);
    maxLoadFactor = mlf <= 0 ? 0.7 : mlf;
    numberOfElements = 0;
    collisionsCount = 0;
}

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
float BaseHashTable<HashTable, Collection, Key, Value, Hash>::getLoadFactor() const {
    return static_cast<float>(this->numberOfElements) / this->tableSize;
}

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
void BaseHashTable<HashTable, Collection, Key, Value, Hash>::clearHashTable() {
    table.clear();
    table.resize(tableSize);
    numberOfElements = 0;
    collisionsCount = 0;
    static_cast<HashTable*>(this)->resetCounter();
}

template <typename HashTable, typename Collection, typename Key, typename Value, typename Hash>
void BaseHashTable<HashTable, Collection, Key, Value, Hash>::incrementCollisionsCount(size_t amount) const {
    collisionsCount += amount;
}
```

5.5 ChainedHashTable.hpp

```
#ifndef CHAINED_HASH_TABLE_HPP
#define CHAINED_HASH_TABLE_HPP

#include <vector>
#include <list>
#include <utility>
#include <functional>

#include "HashTables/Base/BaseHashTable.hpp"
#include "Dictionary/IDictionary.hpp"

template <typename Key, typename Value, typename Hash = std::hash<Key»
class ChainedHashTable : public IDictionary<Key, Value>, public BaseHashTable<ChainedHashTable<Key, Value, Hash>, std::list<std::pair<Key, Value», Key, Value, Hash> {

    template <typename Iterator, typename BucketRef>
    struct GenericFindResult {
        Iterator iterator;

        BucketRef bucketRef;

        GenericFindResult(Iterator it, BucketRef bRef);

        bool wasElementFound() const;
    };

    using FindResult = GenericFindResult<
        typename std::list<std::pair<Key, Value»::iterator,
        std::list<std::pair<Key, Value»&>;

    using ConstFindResult = GenericFindResult<
        typename std::list<std::pair<Key, Value»::const_iterator,
        const std::list<std::pair<Key, Value»&>;

    size_t hashCode(const Key& key) const;


    ConstFindResult findConstPairIterator(const Key& key) const;

    FindResult findPairIterator(const Key& key);
public:
    ChainedHashTable(size_t size = 7, float mlf = 1.0);

    void insert(const Key& key, const Value& value) override;
    bool find(const Key& key, Value& outValue) const override;

    void update(const Key& key, const Value& value) override;

    void remove(const Key& key) override;

    void clear() override;

    void printInOrder(std::ostream& out) const override;

    size_t getComparisonsCount() const override;

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

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

    void rehash(size_t m);

    size_t getCollissionsCount() const;

    size_t getTableSize() const;

    void print() const;
};

#include "HashTables/Chained/ChainedHashTable.impl.hpp"

#endif

```

5.6 ChainedHashTable.impl.hpp

```
#include "HashTables/Chained/ChainedHashTable.hpp"

#include <cmath>
#include <iomanip>

#include "Exceptions/KeyExceptions.hpp"
#include "Utils/StringHandler.hpp"

template <typename Key, typename Value, typename Hash>
template <typename Iterator, typename BucketRef>
ChainedHashTable<Key, Value, Hash>::GenericFindResult<Iterator, BucketRef>::GenericFindResult(
    Iterator it, BucketRef bRef)
    : iterator(it), bucketRef(bRef) {}

template <typename Key, typename Value, typename Hash>
template <typename Iterator, typename BucketRef>
bool ChainedHashTable<Key, Value, Hash>::GenericFindResult<Iterator, BucketRef>::wasElementFound() const {
    return iterator != bucketRef.end();
}

template <typename Key, typename Value, typename Hash>
ChainedHashTable<Key, Value, Hash>::ChainedHashTable(size_t size, float mlf)
    : BaseHashTable<ChainedHashTable<Key, Value, Hash>, std::list<std::pair<Key, Value», Key, Value, Hash>(this->getNextPrime(size), mlf) {}

template <typename Key, typename Value, typename Hash>
size_t ChainedHashTable<Key, Value, Hash>::hashCode(const Key& key) const {
    return this->hashing(key) % this->tableSize;
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::rehash(size_t m) {
    size_t newTableSize = this->getNextPrime(m);

    if (newTableSize > this->tableSize) {
        std::vector<std::list<std::pair<Key, Value»> copy = this->table;
        this->table.clear();
        this->table.resize(newTableSize);
        this->tableSize = newTableSize;
        this->numberOfElements = 0;

        for (auto& line : copy) {
            for (auto& [k, v] : line)
                insert(k, v);
            line.clear();
        }
    }
}

```

```
template <typename Key, typename Value, typename Hash>
typename ChainedHashTable<Key, Value, Hash>::FindResult
ChainedHashTable<Key, Value, Hash>::findPairIterator(const Key& key) {
    size_t slot = hashCode(key);



    std::list<std::pair<Key, Value»& lst = this->table[slot];

    auto it = std::find_if(lst.begin(), lst.end(), [this, &key](const std::pair<Key, Value>& p) {
        this->comparisonsCount++;
        return p.first == key;
    });

    return FindResult(it, lst);
}

template <typename Key, typename Value, typename Hash>
typename ChainedHashTable<Key, Value, Hash>::ConstFindResult
ChainedHashTable<Key, Value, Hash>::findConstPairIterator(const Key& key) const {
    size_t slot = hashCode(key);



    const std::list<std::pair<Key, Value»& lst = this->table[slot];

    auto it = std::find_if(lst.begin(), lst.end(), [this, &key](const std::pair<Key, Value>& p) {
        this->comparisonsCount++;
        return p.first == key;
    });

    return ConstFindResult(it, lst);
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::insert(const Key& key, const Value& value) {
    this->checkAndRehash();

    size_t slot = hashCode(key);

    if (!this->table[slot].empty())
        this->incrementCollisionsCount();

    for (const auto& p : this->table[slot]) {
        this->comparisonsCount++;
        if (p.first == key) throw KeyAlreadyExistsException();
    }

    this->table[slot].push_back({key, value});
    this->numberOfElements++;
}

template <typename Key, typename Value, typename Hash>
bool ChainedHashTable<Key, Value, Hash>::find(const Key& key, Value& outValue) const {
    ConstFindResult response = findConstPairIterator(key);

    bool wasFound = response.wasElementFound();

    if (wasFound) outValue = response.iterator->second;

    return wasFound;
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::update(const Key& key, const Value& value) {
    FindResult response = findPairIterator(key);

    if (!response.wasElementFound()) throw KeyNotFoundException();

    response.iterator->second = value;
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::remove(const Key& key) {
    FindResult response = findPairIterator(key);

    if (response.wasElementFound()) {
        response.bucketRef.erase(response.iterator);
        this->numberOfElements--;
    }
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::clear() {
    this->clearHashTable();
}

```

```
template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::printInOrder(std::ostream& out) const {
    size_t maxKeyLen = 0, maxValLen = 0;
    std::vector<std::pair<Key, Value» vec(this->numberOfElements);

    size_t i = 0;
    for (const auto& line : this->table) {
        for (const auto& p : line) {
            maxKeyLen = std::max(maxKeyLen, StringHandler::size(p.first));
            maxValLen = std::max(maxValLen, StringHandler::size(p.second));

            vec[i++] = p;
        }
    }

    std::sort(vec.begin(), vec.end(), [](const auto& pa, const auto& pb) {
        return pa.first < pb.first;
    });


    for (const auto& p : vec) {
        out « StringHandler::SetWidthAtLeft(p.first, maxKeyLen) « " | " «
StringHandler::SetWidthAtLeft(p.second, maxValLen) « "\n";
    }
}

template <typename Key, typename Value, typename Hash>
size_t ChainedHashTable<Key, Value, Hash>::getComparisonsCount() const {
    return this->comparisonsCount;
}

template <typename Key, typename Value, typename Hash>
Value& ChainedHashTable<Key, Value, Hash>::operator[](const Key& key) {
    this->checkAndRehash();

    FindResult response = findPairIterator(key);

    if (!response.wasElementFound()) {

        response.bucketRef.push_back({key, Value()});
        this->numberOfElements++;
        return response.bucketRef.back().second;
    } else {
        return response.iterator->second;
    }
}

template <typename Key, typename Value, typename Hash>
const Value& ChainedHashTable<Key, Value, Hash>::operator[](const Key& key) const {
    ConstFindResult response = findConstPairIterator(key);

    if (!response.wasElementFound()) {
        throw KeyNotFoundException();
    } else {
        return response.iterator->second;
    }
}

template <typename Key, typename Value, typename Hash>
size_t ChainedHashTable<Key, Value, Hash>::getCollissionsCount() const {
    return this->collisionsCount;
}

template <typename Key, typename Value, typename Hash>
size_t ChainedHashTable<Key, Value, Hash>::getTableSize() const {
    return this->tableSize;
}

template <typename Key, typename Value, typename Hash>
void ChainedHashTable<Key, Value, Hash>::print() const {
    for (size_t i = 0; i < this->table.size(); ++i) {
        std::cout « "Slot " « i « ": ";
        if (this->table[i].empty()) {
        std::cout « "Empty";
        } else {
        for (const auto& pair : this->table[i]) {
            std::cout « "[" « pair.first « ": " « pair.second « "] ";
        }
        }
        std::cout « "\n";
    }
}
```

5.7 OpenAddressingHashTable.hpp

```
#ifndef OPEN_ADDRESSING_HASH_TABLE_HPP
#define OPEN_ADDRESSING_HASH_TABLE_HPP

#include <vector>

#include "Dictionary/IDictionary.hpp"
#include "HashTables/Base/BaseHashTable.hpp"
#include "HashTables/OpenAddressing/Slot.hpp"

template <typename Key, typename Value, typename Hash = std::hash<Key»
class OpenAddressingHashTable : public IDictionary<Key, Value>, public BaseHashTable<OpenAddressingHashTable<Key, Value, Hash>, Slot<Key, Value>, Key, Value, Hash>{
private:
    template <typename Entry>
    struct GenericFindResult {
        Entry* slot;

        Entry* availableSlot;

        GenericFindResult(Entry* e, Entry* as = nullptr);

        bool wasElementFound() const;
    };

    using FindResult = GenericFindResult<Slot<Key, Value»;

    using ConstFindResult = GenericFindResult<const Slot<Key, Value»;


    size_t hashCode(const Key& key, size_t i) const;

    ConstFindResult findConstSlot(const Key& key) const;

    FindResult findSlot(const Key& key);

    size_t nextBase2Of(size_t m) const;
public:
    OpenAddressingHashTable(size_t size = 8, float mlf = 0.7);

    void insert(const Key& key, const Value& value);

    bool find(const Key& key, Value& outValue) const;

    void update(const Key& key, const Value& value);

    void remove(const Key& key);

    void clear();

    void printInOrder(std::ostream& out) const;

    size_t getComparisonsCount() const;

    Value& operator[](const Key& key);

    const Value& operator[](const Key& key) const;

    void rehash(size_t m);

    size_t getCollisionsCount() const;

    size_t getTableSize() const;

    void print() const;
};

#include "HashTables/OpenAddressing/OpenAddressingHashTable.impl.hpp"

#endif
```

5.8 OpenAddressingHashTable.impl.hpp

```
#include "HashTables/OpenAddressing/OpenAddressingHashTable.hpp"

#include <iostream>
#include <algorithm>

#include "Exceptions/KeyExceptions.hpp"

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

```
OpenAddressingHashTable<Key, Value, Hash>::GenericFindResult<Entry>::GenericFindResult(Entry* e, Entry* as)
    : slot(e), availableSlot(as) {}

template <typename Key, typename Value, typename Hash>
template <typename Entry>
bool OpenAddressingHashTable<Key, Value, Hash>::GenericFindResult<Entry>::wasElementFound() const {
    return slot != nullptr;
}

template <typename Key, typename Value, typename Hash>
size_t OpenAddressingHashTable<Key, Value, Hash>::hashCode(const Key& key, size_t i) const {
    return (this->hashing(key) + ((i + (i * i)) / 2)) % this->tableSize;
}

template <typename Key, typename Value, typename Hash>
typename OpenAddressingHashTable<Key, Value, Hash>::ConstFindResult OpenAddressingHashTable<Key, Value, Hash>::findConstSlot(const Key& key) const {
    const Slot<Key, Value>* tableSlot = nullptr;

    for (size_t i = 0; i < this->tableSize; i++) {
        size_t slotIdx = hashCode(key, i);

        const Slot<Key, Value>& slot = this->table[slotIdx];

        if (slot.status == EMPTY) {
            this->incrementCounter(1);
            break;
        }

        if (slot.status == ACTIVE and slot.key == key)
            tableSlot = &slot;

        this->incrementCounter(2);
    }

    return ConstFindResult(tableSlot);
}

template <typename Key, typename Value, typename Hash>
typename OpenAddressingHashTable<Key, Value, Hash>::FindResult OpenAddressingHashTable<Key, Value, Hash>::findSlot(const Key& key) {
    Slot<Key, Value> *tableSlot = nullptr, *availableSlot = nullptr;

    for (size_t i = 0; i < this->tableSize; i++) {
        size_t slotIdx = hashCode(key, i);
        Slot<Key, Value>& slot = this->table[slotIdx];

        if (slot.status == EMPTY) {
            if (!availableSlot)
                availableSlot = &slot;

            this->incrementCounter(2);

            break;
        }

        if (slot.status == ACTIVE and slot.key == key) {
            this->incrementCounter(2);

            tableSlot = &slot;
            break;
        }

        if (slot.status == DELETED and !availableSlot)
            availableSlot = &slot;

        this->incrementCounter(3);

    }

    return FindResult(tableSlot, availableSlot);
}

template <typename Key, typename Value, typename Hash>
size_t OpenAddressingHashTable<Key, Value, Hash>::nextBase2Of(size_t m) const {
    if (m <= 0)
        return 1;



    size_t n = m, bits = sizeof(size_t) * 8;

    for (size_t i = 1; i < bits; i *= 2)
        n |= n » i;

    return (n + 1);
```

```
}

template <typename Key, typename Value, typename Hash>
OpenAddressingHashTable<Key, Value, Hash>::OpenAddressingHashTable(size_t size, float mlf)
    : BaseHashTable<OpenAddressingHashTable<Key, Value, Hash>, Slot<Key, Value>, Key, Value, Hash>(nextBase2Of(size), mlf) {}

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

    if (m > this->tableSize) {
        std::vector<Slot<Key, Value» copy = this->table;
        this->table.clear();
        this->table.resize(m);
        this->tableSize = m;
        this->numberOfElements = 0;

        for (auto& slot : copy) {
            if (slot.status == ACTIVE)
                insert(slot.key, slot.value);
        }
    }
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::insert(const Key& key, const Value& value) {
    this->checkAndRehash();

    int lastDeletedSlot = -1;

    for (int i = 0; i < this->tableSize; i++) {
        size_t slotIdx = hashCode(key, i);
        Slot<Key, Value>& slot = this->table[slotIdx];

        if (slot.status == EMPTY) {
            if (lastDeletedSlot == -1) {
                this->incrementCounter(1);
                slot = Slot(key, value);
                this->numberOfElements++;
                return;
            }

            break;
        } else if (slot.status == ACTIVE and slot.key == key) {
            this->incrementCounter(2);
            throw KeyAlreadyExistsException();
        } else if (slot.status == ACTIVE and slot.key != key) {
            this->incrementCollisionsCount();
            this->incrementCounter(3);
        } else if (slot.status == DELETED and lastDeletedSlot == -1) {
            this->incrementCounter(4);
            lastDeletedSlot = slotIdx;
        } else {
            this->incrementCounter(4);
        }
    }

    this->table[lastDeletedSlot] = Slot(key, value);
}

template <typename Key, typename Value, typename Hash>
bool OpenAddressingHashTable<Key, Value, Hash>::find(const Key& key, Value& outValue) const {
    ConstFindResult response = findConstSlot(key);
    bool wasElementFound = response.wasElementFound();

    if (wasElementFound)
        outValue = response.slot->value;

    return wasElementFound;
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::update(const Key& key, const Value& value) {
    FindResult response = findSlot(key);
    bool wasElementFound = response.wasElementFound();

    if (!wasElementFound)
        throw KeyNotFoundException();

    response.slot->value = value;
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::remove(const Key& key) {
    FindResult response = findSlot(key);

    if (response.wasElementFound())
```

```
        response.slot->status = DELETED;
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::clear() {
    this->clearHashTable();
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::printInOrder(std::ostream& out) const {
    size_t maxKeyLen = 0, maxValLen = 0, i = 0;
    std::vector<Slot<Key, Value» vec(this->numberOfElements);

    for (const Slot<Key, Value>& slot : this->table) {
        if (slot.status == ACTIVE) {
            maxKeyLen = std::max(maxKeyLen, StringHandler::size(slot.key));
            maxValLen = std::max(maxValLen, StringHandler::size(slot.value));

            vec[i++] = slot;
        }
    }

    std::sort(vec.begin(), vec.end(), [](const Slot<Key, Value>& slotA, const Slot<Key, Value>& slotB) {
        return slotA.key < slotB.key;
    });

    for (const Slot<Key, Value>& slot : vec)
        if (slot.status == ACTIVE)
            out « StringHandler::SetWidthAtLeft(slot.key, maxKeyLen) « " | " « StringHandler::SetWidthAtLeft(slot.value, maxValLen) « "\n";
}

template <typename Key, typename Value, typename Hash>
size_t OpenAddressingHashTable<Key, Value, Hash>::getComparisonsCount() const {
    return this->comparisonsCount;
}

template <typename Key, typename Value, typename Hash>
Value& OpenAddressingHashTable<Key, Value, Hash>::operator[](const Key& key) {
    this->checkAndRehash();

    FindResult response = findSlot(key);

    if (response.wasElementFound())
        return response.slot->value;

    this->numberOfElements++;
    response.availableSlot->key = key;
    response.availableSlot->value = Value();
    response.availableSlot->status = ACTIVE;
    return response.availableSlot->value;
}

template <typename Key, typename Value, typename Hash>
const Value& OpenAddressingHashTable<Key, Value, Hash>::operator[](const Key& key) const {
    ConstFindResult response = findConstSlot(key);

    if (!response.wasElementFound())
        throw KeyNotFoundException();

    return response.slot->value;
}

template <typename Key, typename Value, typename Hash>
size_t OpenAddressingHashTable<Key, Value, Hash>::getCollisionsCount() const {
    return this->collisionsCount;
}

template <typename Key, typename Value, typename Hash>
size_t OpenAddressingHashTable<Key, Value, Hash>::getTableSize() const {
    return this->tableSize;
}

template <typename Key, typename Value, typename Hash>
void OpenAddressingHashTable<Key, Value, Hash>::print() const {
    for (size_t i = 0; i < this->table.size(); ++i) {
        const auto& slot = this->table[i];
        std::cout « "Slot " « i « ": ";
        if (slot.status == EMPTY) {
            std::cout « "EMPTY";
        } else if (slot.status == DELETED) {
            std::cout « "DELETED";
        } else if (slot.status == ACTIVE) {
            std::cout « "ACTIVE [" « slot.key « ": " « slot.value « "]";
        }
        std::cout « '\n';
    }
}
```

5.9 Slot.hpp

```
#ifndef SLOT_HPP
#define SLOT_HPP

enum Status { EMPTY, ACTIVE, DELETED };

template <typename Key, typename Value>
struct Slot {
    Key key;

    Value value;

    Status status;

    Slot(): status(EMPTY) {}

    Slot(const Key& k, const Value& v): key(k), value(v), status(ACTIVE) {}

};

#endif
```

5.10 AVLNode.hpp

```
#ifndef AVL_NODE_HPP
#define AVL_NODE_HPP

#include "Trees/Base/Node.hpp"

template <typename Key, typename Value>
struct AVLNode : public Node<Key, Value> {
    AVLNode *left;

    AVLNode *right;

    size_t height;

    AVLNode(const Key& k, const Value& v)
        : Node<Key, Value>(k, v), left(nullptr), right(nullptr), height(1) {}
};


#endif
```

5.11 AVLTree.hpp

```
#ifndef AVL_TREE_HPP
#define AVL_TREE_HPP

#include <functional>
#include <iostream>

#include "Dictionary/IDictionary.hpp"
#include "Trees/Base/Node.hpp"
#include "Trees/Base/BaseTree.hpp"
#include "Trees/AVL/AVLNode.hpp"

template <typename Key, typename Value>
class AVLTree : public IDictionary<Key, Value>, public BaseTree<AVLTree<Key, Value>, AVLNode<Key, Value>, Key, Value> {
private:
    size_t height(AVLNode<Key, Value>* node) const;

    size_t calcHeight(AVLNode<Key, Value>* node) const;

    int getBalanceFactor(AVLNode<Key, Value>* node) const;

    void printTree(AVLNode<Key, Value>* node, size_t depth = 0) const;

    AVLNode<Key, Value>* rotateLeft(AVLNode<Key, Value>*& y);

    AVLNode<Key, Value>* rotateRight(AVLNode<Key, Value>*& y);

    AVLNode<Key, Value>* fixupNode(AVLNode<Key, Value>* node);

    AVLNode<Key, Value>* removeSuccessor(AVLNode<Key, Value>* root, AVLNode<Key, Value>* node);

    AVLNode<Key, Value>* insert(const Key& key, const Value& value, AVLNode<Key, Value>* node);

    AVLNode<Key, Value>* update(const Key& key, const Value& value, AVLNode<Key, Value>* node);

    AVLNode<Key, Value>* remove(const Key& key, AVLNode<Key, Value>* node);

    AVLNode<Key, Value>* upsert(const Key& key, AVLNode<Key, Value>* node, Value*& outValue);

    public:
    static const int IMBALANCE = 2;

    AVLTree();

    ~AVLTree();

    void insert(const Key& key, const Value& value) override;

    bool find(const Key& key, Value& outValue) const override;

    void update(const Key& key, const Value& value) override;

    void remove(const Key& key) override;

    void clear() override;

    void printInOrder(std::ostream& out) const override;

    size_t getComparisonsCount() const override;

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

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

    void print() const;

    size_t getRotationsCount() const;
};

#include "Trees/AVL/AVLTree.impl.hpp"

#endif
```

5.12 AVLTree.impl.hpp

```
#include "Trees/AVL/AVLTree.hpp"

#include <iostream>
#include <cmath>

#include "Utils/StringHandler.hpp"

template <typename Key, typename Value>
size_t AVLTree<Key, Value>::height(AVLNode<Key, Value>* node) const {
    if (!node) return 0;

    return node->height;
}

template <typename Key, typename Value>
size_t AVLTree<Key, Value>::calcHeight(AVLNode<Key, Value>* node) const {
    if (!node) return 0;

    size_t leftHeight = height(node->left),
           rightHeight = height(node->right);

    return 1 + std::max(leftHeight, rightHeight);
}

template <typename Key, typename Value>
int AVLTree<Key, Value>::getBalanceFactor(AVLNode<Key, Value>* node) const {
    if (!node) return 0;
    return height(node->right) - height(node->left);
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::rotateLeft(AVLNode<Key, Value>*& y) {
    AVLNode<Key, Value>* x = y->right;
    y->right = x->left;
    x->left = y;

    y->height = calcHeight(y);
    x->height = calcHeight(x);

    this->incrementRotationsCount();

    return x;
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::rotateRight(AVLNode<Key, Value>*& y) {
    AVLNode<Key, Value>* x = y->left;

    y->left = x->right;
    x->right = y;

    y->height = calcHeight(y);
    x->height = calcHeight(x);

    this->incrementRotationsCount();

    return x;
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::fixupNode(AVLNode<Key, Value>* y) {
    if (!y) return nullptr;

    int balanceFactor = getBalanceFactor(y);

    if (std::abs(balanceFactor) == IMBALANCE) {
        if (balanceFactor < 0) {
            if (getBalanceFactor(y->left) <= 0) {
                y = rotateRight(y);

            } else {
                y->left = rotateLeft(y->left);
                y = rotateRight(y);

            }
        } else {
            if (getBalanceFactor(y->right) >= 0) {
                y = rotateLeft(y);

            } else {
                y->right = rotateRight(y->right);
                y = rotateLeft(y);

            }
        }
    }

    y->height = calcHeight(y);

    return y;
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::removeSuccessor(AVLNode<Key, Value>* root, AVLNode<Key, Value>* node) {
    if (node->left) {
        node->left = removeSuccessor(root, node->left);
    } else {
        root->setKey(node->getKey());
        root->setValue(node->getValue());
        AVLNode<Key, Value>* aux = node->right;
        delete node;
        return aux;
    }

    return fixupNode(node);
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::insert(const Key& key, const Value& value, AVLNode<Key, Value>* node) {
    // It'll never be called w/ root == nullptr
    if (!node)
        return new AVLNode(key, value);

    if (key < node->getKey()) {
        this->incrementCounter(1);
        node->left = insert(key, value, node->left);
    } else if (key > node->getKey()) {
        this->incrementCounter(2);
        node->right = insert(key, value, node->right);
    } else {
        this->incrementCounter(2);
        throw KeyAlreadyExistsException();
    }

    return fixupNode(node);
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::update(const Key& key, const Value& value, AVLNode<Key, Value>* node) {
    if (!node) throw KeyNotFoundException();

    if (key < node->getKey()) {
        this->incrementCounter(1);
        node->left = update(key, value, node->left);
    } else if (key > node->getKey()) {
        this->incrementCounter(2);
        node->right = update(key, value, node->right);
    } else {
        this->incrementCounter(2);
        node->setValue(value);
    }

    return fixupNode(node);
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::remove(const Key& key, AVLNode<Key, Value>* node) {
    if (!node) return nullptr;

    if (key < node->getKey()) {
        node->left = remove(key, node->left);
    } else if (key > node->getKey()) {
        node->right = remove(key, node->right);
    } else if (!node->right) {
        AVLNode<Key, Value>* leftChild = node->left;
        delete node;
        return leftChild;
    } else {
        node->right = removeSuccessor(node, node->right);
    }

    return fixupNode(node);
}

template <typename Key, typename Value>
AVLNode<Key, Value>* AVLTree<Key, Value>::upsert(const Key& key, AVLNode<Key, Value>* node, Value*& outValue) {
    this->setMaxKeyLen(key);

    if (!node) {
        AVLNode<Key, Value>* newNode = new AVLNode<Key, Value>(key, Value());
        outValue = &(newNode->getValue());
        this->setMaxValLen(*outValue);
        return newNode;
    }

    if (key < node->getKey()) {
        this->incrementCounter(1);
        node->left = upsert(key, node->left, outValue);
    } else if (key > node->getKey()) {
        this->incrementCounter(2);
        node->right = upsert(key, node->right, outValue);
    } else {
        this->incrementCounter(2);
        outValue = &(node->getValue());
        this->setMaxValLen(*outValue);
        return node;
    }

    return fixupNode(node);
}

template <typename Key, typename Value>
AVLTree<Key, Value>::AVLTree()
    : BaseTree<AVLTree<Key, Value>, AVLNode<Key, Value>, Key, Value>(nullptr) {}

template <typename Key, typename Value>
AVLTree<Key, Value>::~AVLTree() { clear(); }

template <typename Key, typename Value>
void AVLTree<Key, Value>::insert(const Key& key, const Value& value) {
    this->root = insert(key, value, this->root);
    this->setMaxKeyLen(key);
    this->setMaxValLen(value);
}

template  <typename Key, typename Value>
bool AVLTree<Key, Value>::find(const Key& key, Value& outValue) const {
    const AVLNode<Key, Value>* node = this->findNode(key);

    if (!node) return false;

    outValue = node->getValue();
    return true;
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::update(const Key& key, const Value& value) {
    this->root = update(key, value, this->root);
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::remove(const Key& key) {
    this->root = remove(key, this->root);
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::clear() {
    this->reset(this->root);
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::printTree(AVLNode<Key, Value>* node, size_t depth) const {
    if (!node) return;

    printTree(node->right, depth+1);

    for (int i = 0; i < depth; i++)
        std::cout « "    ";
    std::cout « node->show() « std::endl;

    printTree(node->left, depth+1);
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::print() const {
    printTree(this->root);
}

template <typename Key, typename Value>
void AVLTree<Key, Value>::printInOrder(std::ostream& os) const {
    this->inOrderTransversal(os, this->root, nullptr);
}

template <typename Key, typename Value>
size_t AVLTree<Key, Value>::getComparisonsCount() const {
    return this->comparisonsCount;
}

template <typename Key, typename Value>
Value& AVLTree<Key, Value>::operator[](const Key& key) {
    Value* insertedValue = nullptr;
    this->root = upsert(key, this->root, insertedValue);
    return *insertedValue;
}

template <typename Key, typename Value>
const Value& AVLTree<Key, Value>::operator[](const Key& key) const {
    return this->at(key);
}

template <typename Key, typename Value>
size_t AVLTree<Key, Value>::getRotationsCount() const {
    return this->rotationsCount;
}
```

5.13 BaseTree.hpp

```
#ifndef BASE_TREE_HPP
#define BASE_TREE_HPP

#include <iostream>

#include "Exceptions/KeyExceptions.hpp"

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

```
class BaseTree {
    void count(size_t n) const;

    void clearCounter();
protected:
    Node* root;

    size_t maxKeyLen;

    size_t maxValLen;

    size_t rotationsCount;

    BaseTree(Node* r);

    const Node* findNode(const Key& key, Node* comp = nullptr) const;

    Node* minimum(Node* node) const;

    void clearNode(Node* node, Node* comp);

    void reset(Node* node, Node* comp = nullptr, Node* defaultRoot = nullptr);

    void inOrderTransversal(std::ostream& out, Node* node, Node* comp) const;

    const Value& at(const Key& key, Node* comp = nullptr) const;

    void setMaxKeyLen(const Key& key);

    void setMaxValLen(const Value& value);

    void incrementRotationsCount(size_t amount = 1);
};

// Include the implementation file to provide the definitions for the template methods.
// This must be at the end of the header file.
#include "Trees/Base/BaseTree.impl.hpp"

#endif
```

5.14 BaseTree.impl.hpp

```
#ifndef BASE_TREE_IMPL_HPP
#define BASE_TREE_IMPL_HPP

#include "Trees/Base/BaseTree.hpp"

#include <cmath>

#include "Utils/StringHandler.hpp"

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::count(size_t n) const {
    static_cast<const Tree*>(this)->incrementCounter(n);
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::clearCounter() {
    static_cast<Tree*>(this)->resetCounter();
}

template <typename Tree, typename Node, typename Key, typename Value>
BaseTree<Tree, Node, Key, Value>::BaseTree(Node* r)
    : root(r), maxKeyLen(0), maxValLen(0), rotationsCount(0) {
        clearCounter();
    }

template <typename Tree, typename Node, typename Key, typename Value>
const Node* BaseTree<Tree, Node, Key, Value>::findNode(const Key& key, Node* comp) const {
    const Node* aux = root;

    while (aux != comp) {
        if (key < aux->getKey()) {
            count(1);
            aux = aux->left;
        } else if (key > aux->getKey()) {
            count(2);
            aux = aux->right;
        } else {
            count(2);
            return aux;
        }
    }

    return nullptr;
}

template <typename Tree, typename Node, typename Key, typename Value>
Node* BaseTree<Tree, Node, Key, Value>::minimum(Node* node) const {
    if (!node->left) return node;
    return minimum(node->left);
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::clearNode(Node* node, Node* comp) {
    if (node != comp) {
        clearNode(node->left, comp);
        clearNode(node->right, comp);
        delete node;
    }
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::reset(Node* node, Node* comp, Node* defaultRoot) {
    clearNode(node, comp);
    root = defaultRoot;
    maxKeyLen = 0;
    maxValLen = 0;
    rotationsCount = 0;
    clearCounter();
}


template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::inOrderTransversal(std::ostream& out, Node* node, Node* comp) const {
    if (node != comp) {
        inOrderTransversal(out, node->left, comp);

        out « StringHandler::SetWidthAtLeft(node->getKey(), maxKeyLen) « " | "
            « StringHandler::SetWidthAtLeft(node->getValue(), maxValLen) « ’\n’;

        inOrderTransversal(out, node->right, comp);
    }
}

template <typename Tree, typename Node, typename Key, typename Value>
const Value& BaseTree<Tree, Node, Key, Value>::at(const Key& key, Node* comp) const {
    const Node* aux = root;

    while (aux != comp) {
        if (key < aux->getKey()) {
            count(1);
            aux = aux->left;
        } else if (key > aux->getKey()) {
            count(2);
            aux = aux->right;
        } else {
            count(2);
            return aux->getValue();
        }
    }

    throw KeyNotFoundException();
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::setMaxKeyLen(const Key& key) {
    maxKeyLen = std::max(maxKeyLen, StringHandler::size(key));
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::setMaxValLen(const Value& value) {
    maxValLen = std::max(maxValLen, StringHandler::size(value));
}

template <typename Tree, typename Node, typename Key, typename Value>
void BaseTree<Tree, Node, Key, Value>::incrementRotationsCount(size_t amount) {
    rotationsCount += amount;
}

#endif
```

5.15 Node.hpp

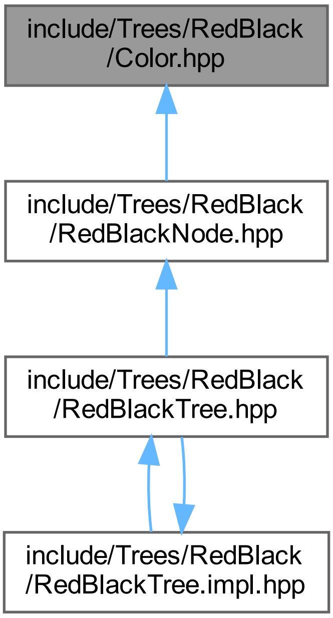
```
#ifndef INODE_HPP
```

```
#define INODE_HPP

#include <utility>
#include <sstream>

template <typename Key, typename Value>
class Node {
    std::pair<Key, Value> data;

public:
    Node(const Key& key, const Value& value): data({key, value}) {}

    const Key& getKey() const { return data.first; }

    void setKey(const Key& key) { data.first = key; }

    const Value& getValue() const { return data.second; }

    Value& getValue() { return data.second; }

    void setValue(const Value& value) { data.second = value; }

    void update(const Key& key, const Value& value) {
        setKey(key);
        setValue(value);
    }

    std::string show() const {
        std::ostringstream os;
        os « "(" « getKey() « ", " « getValue() « ")";
        return os.str();
    }

    ~Node() = default;
};


#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.

```
#ifndef COLOR_HPP
#define COLOR_HPP

enum Color { RED, BLACK };

#endif
```

5.18 RedBlackNode.hpp

```
#ifndef RED_BLACK_NODE_HPP
#define RED_BLACK_NODE_HPP

#include "Trees/Base/Node.hpp"
#include "Trees/RedBlack/Color.hpp"

template <typename Key, typename Value>
struct RedBlackNode : public Node<Key, Value> {
    RedBlackNode* left;

    RedBlackNode* right;

    RedBlackNode* parent;

    Color color;

    RedBlackNode(const Key& k, const Value& v,
                    RedBlackNode* l, RedBlackNode* r,
                    RedBlackNode* p, Color c)
        : Node<Key, Value>(k, v), left(l), right(r), parent(p), color(c) {}

    RedBlackNode(Color color = BLACK) : Node<Key, Value>(Key(), Value()) {
        this->left = this;
        this->right = this;
        this->parent = this;
        this->color = color;
    }
};

#endif
```

5.19 RedBlackTree.hpp

```
#ifndef RED_BLACK_TREE_HPP
#define RED_BLACK_TREE_HPP

#include <functional>
#include <iostream>

#include "Dictionary/IDictionary.hpp"
#include "Trees/RedBlack/RedBlackNode.hpp"
#include "Trees/Base/BaseTree.hpp"

template <typename Key, typename Value>
class RedBlackTree : public IDictionary<Key, Value>, public BaseTree<RedBlackTree<Key, Value>, RedBlackNode<Key, Value>, Key, Value> {
private:
    static RedBlackNode<Key, Value> NIL_NODE;
    static constexpr RedBlackNode<Key, Value>* const NIL = &NIL_NODE;

    RedBlackNode<Key, Value>* rotateLeft(RedBlackNode<Key, Value>* y);

    RedBlackNode<Key, Value>* rotateRight(RedBlackNode<Key, Value>* y);

    void insertFixup(RedBlackNode<Key, Value>* z);

    void deleteFixup(RedBlackNode<Key, Value>* x);


    void deleteNode(RedBlackNode<Key, Value>* z);

    void printTree(RedBlackNode<Key, Value>* node, int indent = 0) const;

public:
    RedBlackTree();

    void insert(const Key& key, const Value& value);

    bool find(const Key& key, Value& outValue) const;

    void update(const Key& key, const Value& value);

    void remove(const Key& key);

    void clear();

    void printInOrder(std::ostream& out) const;

    size_t getComparisonsCount() const;

    virtual Value& operator[](const Key& key);

    virtual const Value& operator[](const Key& key) const;

    void print() const;

    size_t getRotationsCount() const;
};

#include "Trees/RedBlack/RedBlackTree.impl.hpp"

#endif
```

5.20 RedBlackTree.impl.hpp

```
#include "Trees/RedBlack/RedBlackTree.hpp"

template <typename Key, typename Value>
RedBlackNode<Key, Value>* RedBlackTree<Key, Value>::rotateLeft(RedBlackNode<Key, Value>* y) {
    RedBlackNode<Key, Value>* x = y->right;

    y->right = x->left;
    if (y->right != NIL) y->right->parent = y;
    x->left = y;

    x->parent = y->parent;
    y->parent = x;

    if (x->parent != NIL) {
        if (x->getKey() < x->parent->getKey())
            x->parent->left = x;
        else
            x->parent->right = x;
    } else {
        this->root = x;
    }

    this->incrementRotationsCount();

    return x;
}

template <typename Key, typename Value>
RedBlackNode<Key, Value>* RedBlackTree<Key, Value>::rotateRight(RedBlackNode<Key, Value>* y) {
    RedBlackNode<Key, Value>* x = y->left;

    y->left = x->right;
    if (y->left != NIL) y->left->parent = y;
    x->right = y;

    x->parent = y->parent;
    y->parent = x;

    if (x->parent != NIL) {
        if (x->getKey() < x->parent->getKey())
            x->parent->left = x;
        else
            x->parent->right = x;
    } else {
        this->root = x;
    }

    this->incrementRotationsCount();

    return x;
}

```

```
template <typename Key, typename Value>
void RedBlackTree<Key, Value>::insertFixup(RedBlackNode<Key, Value>* z) {
    while (z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            if (z->parent->parent->right->color == RED) { // Case 1
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                z->parent->parent->right->color = BLACK;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) { // Case 2
                    z = z->parent;
                    z = rotateLeft(z);
                    z = z->left;

                }

                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                z = rotateRight(z->parent->parent);

            }
        } else { // Symmetrical case
            if (z->parent->parent->left->color == RED) { // Case 1
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                z->parent->parent->left->color = BLACK;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) { // Case 2
                    z = z->parent;
                    z = rotateRight(z);
                    z = z->right;

                }

                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                z = rotateLeft(z->parent->parent);

            }
        }
    }

    this->root->color = BLACK;
}

template <typename Key, typename Value>
RedBlackNode<Key, Value> RedBlackTree<Key, Value>::NIL_NODE = RedBlackNode<Key, Value>();

template <typename Key, typename Value>
RedBlackTree<Key, Value>::RedBlackTree()
    : BaseTree<RedBlackTree<Key, Value>, RedBlackNode<Key, Value>, Key, Value>(NIL) {
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::insert(const Key& key, const Value& value) {
    RedBlackNode<Key, Value> *x = this->root, *y = NIL;

    while (x != NIL) {
        y = x;

        if (key < x->getKey()) {
            this->incrementCounter(1);
            x = x->left;
        } else if (key > x->getKey()) {
            this->incrementCounter(2);
            x = x->right;
        } else {
            this->incrementCounter(2);
            throw KeyAlreadyExistsException();
        }

    }

    this->setMaxKeyLen(key);
    this->setMaxValLen(value);

    RedBlackNode<Key, Value> *z = new RedBlackNode<Key, Value>(key, value, NIL, NIL, NIL, RED);

    z->parent = y;
    if (y == NIL) {
        this->incrementCounter(1);
        this->root = z;
    } else if (z->getKey() < y->getKey()) {
        this->incrementCounter(2);
        y->left = z;
    } else {
        this->incrementCounter(2);
        y->right = z;
    }

    insertFixup(z);
}

template <typename Key, typename Value>
void RedBlackTree<Key,Value>::deleteFixup(RedBlackNode<Key, Value>* x) {
    while (x != this->root and x->color == BLACK) {
        if (x == x->parent->left) {
            RedBlackNode<Key, Value>* w = x->parent->right;

            if (w->color == RED) { // Case 1
                x->parent->color = RED;
                w->color = BLACK;
                x->parent = rotateLeft(x->parent);
                w = x->parent->right;
            }

            if (w->left->color == BLACK and w->right->color == BLACK) { // Case 2
                w->color = RED;
                x = x->parent;
            } else {
                if (w->right->color == BLACK) { // Case 3
                    w->left->color = BLACK;
                    w->color = RED;
                    w = rotateRight(w);

                    w = x->parent->right;
                }

                // Case 4
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                w = rotateLeft(x->parent);


                x = this->root;
            }
        } else { // Symetrical case
            RedBlackNode<Key, Value>* w = x->parent->left;

            if (w->color == RED) { // Case 1
                x->parent->color = RED;
                w->color = BLACK;
                x->parent = rotateRight(x->parent);

                w = x->parent->left;
            }

            if (w->right->color == BLACK and w->left->color == BLACK) { // Case 2
                w->color = RED;
                x = x->parent;
            } else {
                if (w->left->color == BLACK) { // Case 3
                    w->right->color = BLACK;
                    w->color = RED;
                    w = rotateLeft(w);

                    w = x->parent->left;
                }

                // Case 4
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                w = rotateRight(x->parent);


                x = this->root;
            }
        }
    }

    x->color = BLACK;
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::deleteNode(RedBlackNode<Key, Value>* z) {
    RedBlackNode<Key, Value>* y;
    if (z->left == NIL or z->right == NIL)
        y = z;
    else
        y = this->minimum(z->right);

    RedBlackNode<Key, Value>* x;
    if (y->left != NIL)
        x = y->left;
    else
        x = y->right;

    x->parent = y->parent;

    if (y->parent == NIL) {
        this->root = x;
    } else {
        if (y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;
    }

    if (y != z)
        y->setKey(z->getKey());

    if (y->color == BLACK)
        deleteFixup(x);

    delete y;
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::printTree(RedBlackNode<Key, Value>* node, int indent) const {
    if (node != NIL) {
        printTree(node->right, indent + 4);

        if (indent > 0) {
            std::cout « std::string(indent, ' ');
        }

        std::cout « node->getKey() « " (" « (node->color == RED ? "R" : "B") « ")" « std::endl;

        printTree(node->left, indent + 4);
    }
}

template <typename Key, typename Value>
bool RedBlackTree<Key, Value>::find(const Key& key, Value& outValue) const {
    const RedBlackNode<Key, Value>* node = this->findNode(key, NIL);

    if (!node) return false;

    outValue = node->getValue();
    return true;
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::update(const Key& key, const Value& value) {
    RedBlackNode<Key, Value>* aux = this->root;
    while (aux != NIL) {
        if (key < aux->getKey()) {
            this->incrementCounter(1);
            aux = aux->left;
        } else if (key > aux->getKey()) {
            this->incrementCounter(2);
            aux = aux->right;
        } else {
            this->incrementCounter(2);
            aux->setValue(value);
            return;
        }
    }

    throw KeyNotFoundException();
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::print() const {
    printTree(this->root);
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::remove(const Key& key) {
    RedBlackNode<Key, Value>* p = this->root;

    while (p != NIL and p->getKey() != key) {
        if (key < p->getKey()) p = p->left;
        else p = p->right;

        this->incrementCounter(1);
    }
```

```cpp

    if (p != NIL)
        deleteNode(p);
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::clear() {
    this->reset(this->root, NIL, NIL);
}

template <typename Key, typename Value>
void RedBlackTree<Key, Value>::printInOrder(std::ostream& os) const {
    this->inOrderTransversal(os, this->root, NIL);
}

template <typename Key, typename Value>
size_t RedBlackTree<Key, Value>::getComparisonsCount() const {
    return this->comparisonsCount;
}

template <typename Key, typename Value>
Value& RedBlackTree<Key, Value>::operator[](const Key& key) {
    this->setMaxKeyLen(key);

    RedBlackNode<Key, Value> *x = this->root, *y = NIL;

    while (x != NIL) {
        y = x;

        if (key < x->getKey()) {
            this->incrementCounter(1);
            x = x->left;
        } else if (key > x->getKey()) {
            this->incrementCounter(2);
            x = x->right;
        } else {
            this->incrementCounter(2);
            this->setMaxValLen(x->getValue());
            return x->getValue();
        }
    }

    RedBlackNode<Key, Value> *z = new RedBlackNode<Key, Value>(key, Value(), NIL, NIL, NIL, RED);

    this->setMaxValLen(z->getValue());

    z->parent = y;
    if (y == NIL) {
        this->incrementCounter(1);
        this->root = z;
    } else if (z->getKey() < y->getKey()) {
        this->incrementCounter(2);
        y->left = z;
    } else {
        this->incrementCounter(2);
        y->right = z;
    }

    insertFixup(z);

    return z->getValue();
}

template <typename Key, typename Value>
const Value& RedBlackTree<Key, Value>::operator[](const Key& key) const {
    return this->at(key);
}

template <typename Key, typename Value>
size_t RedBlackTree<Key, Value>::getRotationsCount() const {
    return this->rotationsCount;
}
```

5.21 utf8.h

```cpp
// Copyright 2006 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
```

```
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_2675DCD0_9480_4c0c_B92A_CC14C027B731
#define UTF8_FOR_CPP_2675DCD0_9480_4c0c_B92A_CC14C027B731

/*
To control the C++ language version used by the library, you can define UTF_CPP_CPLUSPLUS macro
and set it to one of the values used by the __cplusplus predefined macro.

For instance,
    #define UTF_CPP_CPLUSPLUS 199711L
will cause the UTF-8 CPP library to use only types and language features available in the C++ 98 standard.
Some library features will be disabled.

If you leave UTF_CPP_CPLUSPLUS undefined, it will be internally assigned to __cplusplus.
*/

#include "utf8/checked.h"
#include "utf8/unchecked.h"

#endif // header guard
```

5.22 checked.h

```
// Copyright 2006-2016 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_CHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
#define UTF8_FOR_CPP_CHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731

#include "core.h"
#include <stdexcept>

namespace utf8
{
    // Base for the exceptions that may be thrown from the library
    class exception : public ::std::exception {
    };

    // Exceptions that may be thrown from the library functions.
    class invalid_code_point : public exception {
        utfchar32_t cp;
    public:
        invalid_code_point(utfchar32_t codepoint) : cp(codepoint) {}
        virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid code point"; }
        utfchar32_t code_point() const {return cp;}
    };

    class invalid_utf8 : public exception {
        utfchar8_t u8;
    public:
        invalid_utf8 (utfchar8_t u) : u8(u) {}
        invalid_utf8 (char c) : u8(static_cast<utfchar8_t>(c)) {}
        virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid UTF-8"; }
        utfchar8_t utf8_octet() const {return u8;}
    };

    class invalid_utf16 : public exception {
        utfchar16_t u16;
    public:
        invalid_utf16 (utfchar16_t u) : u16(u) {}
        virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Invalid UTF-16"; }
        utfchar16_t utf16_word() const {return u16;}
    };

    class not_enough_room : public exception {
    public:
        virtual const char* what() const UTF_CPP_NOEXCEPT UTF_CPP_OVERRIDE { return "Not enough space"; }
    };


    template <typename octet_iterator>
    octet_iterator append(utfchar32_t cp, octet_iterator result)
    {
        if (!utf8::internal::is_code_point_valid(cp))
            throw invalid_code_point(cp);

        return internal::append(cp, result);
    }

    inline void append(utfchar32_t cp, std::string& s)
    {
        append(cp, std::back_inserter(s));
    }

    template <typename word_iterator>
    word_iterator append16(utfchar32_t cp, word_iterator result)
    {
        if (!utf8::internal::is_code_point_valid(cp))
            throw invalid_code_point(cp);

        return internal::append16(cp, result);
    }

    template <typename octet_iterator, typename output_iterator>
    output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out, utfchar32_t replacement)
    {
        while (start != end) {
            octet_iterator sequence_start = start;
            internal::utf_error err_code = utf8::internal::validate_next(start, end);
            switch (err_code) {
                case internal::UTF8_OK :
                    for (octet_iterator it = sequence_start; it != start; ++it)
                        *out++ = *it;
                    break;
                case internal::NOT_ENOUGH_ROOM:
                    out = utf8::append (replacement, out);
                    start = end;
                    break;
                case internal::INVALID_LEAD:
                    out = utf8::append (replacement, out);
                    ++start;
                    break;
                case internal::INCOMPLETE_SEQUENCE:
                case internal::OVERLONG_SEQUENCE:
                case internal::INVALID_CODE_POINT:
                    out = utf8::append (replacement, out);
                    ++start;
                    // just one replacement mark for the sequence
                    while (start != end && utf8::internal::is_trail(*start))
                        ++start;
                    break;
            }
        }
        return out;
    }

    template <typename octet_iterator, typename output_iterator>
    inline output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out)
    {
        static const utfchar32_t replacement_marker = utf8::internal::mask16(0xfffd);
        return utf8::replace_invalid(start, end, out, replacement_marker);
    }

    inline std::string replace_invalid(const std::string& s, utfchar32_t replacement)
    {
        std::string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
        return result;
    }

    inline std::string replace_invalid(const std::string& s)
    {
        std::string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    template <typename octet_iterator>
    utfchar32_t next(octet_iterator& it, octet_iterator end)
    {
        utfchar32_t cp = 0;
        internal::utf_error err_code = utf8::internal::validate_next(it, end, cp);
        switch (err_code) {
            case internal::UTF8_OK :
                break;
            case internal::NOT_ENOUGH_ROOM :
                throw not_enough_room();
            case internal::INVALID_LEAD :
            case internal::INCOMPLETE_SEQUENCE :
            case internal::OVERLONG_SEQUENCE :
                throw invalid_utf8(static_cast<utfchar8_t>(*it));
            case internal::INVALID_CODE_POINT :
                throw invalid_code_point(cp);
        }
        return cp;
    }

    template <typename word_iterator>
    utfchar32_t next16(word_iterator& it, word_iterator end)
    {
        utfchar32_t cp = 0;
        internal::utf_error err_code = utf8::internal::validate_next16(it, end, cp);
        if (err_code == internal::NOT_ENOUGH_ROOM)
            throw not_enough_room();
        return cp;
    }

    template <typename octet_iterator>
    utfchar32_t peek_next(octet_iterator it, octet_iterator end)
    {
        return utf8::next(it, end);
    }

    template <typename octet_iterator>
    utfchar32_t prior(octet_iterator& it, octet_iterator start)
    {
        // can't do much if it == start
        if (it == start)
            throw not_enough_room();

        octet_iterator end = it;
        // Go back until we hit either a lead octet or start
        while (utf8::internal::is_trail(*(--it)))
            if (it == start)
                throw invalid_utf8(*it); // error - no lead byte in the sequence
        return utf8::peek_next(it, end);
    }

    template <typename octet_iterator, typename distance_type>
    void advance (octet_iterator& it, distance_type n, octet_iterator end)
    {
        const distance_type zero(0);
        if (n < zero) {
            // backward
            for (distance_type i = n; i < zero; ++i)
                utf8::prior(it, end);
        } else {
            // forward
            for (distance_type i = zero; i < n; ++i)
```

```
                utf8::next(it, end);
        }
    }

    template <typename octet_iterator>
    typename std::iterator_traits<octet_iterator>::difference_type
    distance (octet_iterator first, octet_iterator last)
    {
        typename std::iterator_traits<octet_iterator>::difference_type dist;
        for (dist = 0; first < last; ++dist)
            utf8::next(first, last);
        return dist;
    }

    template <typename u16bit_iterator, typename octet_iterator>
    octet_iterator utf16to8 (u16bit_iterator start, u16bit_iterator end, octet_iterator result)
    {
        while (start != end) {
            utfchar32_t cp = utf8::internal::mask16(*start++);
            // Take care of surrogate pairs first
            if (utf8::internal::is_lead_surrogate(cp)) {
                if (start != end) {
                    const utfchar32_t trail_surrogate = utf8::internal::mask16(*start++);
                    if (utf8::internal::is_trail_surrogate(trail_surrogate))
                        cp = (cp « 10) + trail_surrogate + internal::SURROGATE_OFFSET;
                    else
                        throw invalid_utf16(static_cast<utfchar16_t>(trail_surrogate));
                }
                else
                    throw invalid_utf16(static_cast<utfchar16_t>(cp));

            }
            // Lone trail surrogate
            else if (utf8::internal::is_trail_surrogate(cp))
                throw invalid_utf16(static_cast<utfchar16_t>(cp));

            result = utf8::append(cp, result);
        }
        return result;
    }

    template <typename u16bit_iterator, typename octet_iterator>
    u16bit_iterator utf8to16 (octet_iterator start, octet_iterator end, u16bit_iterator result)
    {
        while (start < end) {
            const utfchar32_t cp = utf8::next(start, end);
            if (cp > 0xffff) { //make a surrogate pair
                *result++ = static_cast<utfchar16_t>((cp » 10)   + internal::LEAD_OFFSET);
                *result++ = static_cast<utfchar16_t>((cp & 0x3ff) + internal::TRAIL_SURROGATE_MIN);
            }
            else
                *result++ = static_cast<utfchar16_t>(cp);
        }
        return result;
    }

    template <typename octet_iterator, typename u32bit_iterator>
    octet_iterator utf32to8 (u32bit_iterator start, u32bit_iterator end, octet_iterator result)
    {
        while (start != end)
            result = utf8::append(*(start++), result);

        return result;
    }

    template <typename octet_iterator, typename u32bit_iterator>
    u32bit_iterator utf8to32 (octet_iterator start, octet_iterator end, u32bit_iterator result)
    {
        while (start < end)
            (*result++) = utf8::next(start, end);

        return result;
    }

    // The iterator class
    template <typename octet_iterator>
    class iterator {
      octet_iterator it;
      octet_iterator range_start;
      octet_iterator range_end;
      public:
      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;
      iterator () {}
      explicit iterator (const octet_iterator& octet_it,
                         const octet_iterator& rangestart,
                         const octet_iterator& rangeend) :
               it(octet_it), range_start(rangestart), range_end(rangeend)
      {
          if (it < range_start || it > range_end)
              throw std::out_of_range("Invalid utf-8 iterator position");
      }
      // the default "big three" are OK
      octet_iterator base () const { return it; }
      utfchar32_t operator * () const
      {
          octet_iterator temp = it;
          return utf8::next(temp, range_end);
      }
      bool operator == (const iterator& rhs) const
      {
          if (range_start != rhs.range_start || range_end != rhs.range_end)
              throw std::logic_error("Comparing utf-8 iterators defined with different ranges");
          return (it == rhs.it);
      }
      bool operator != (const iterator& rhs) const
      {
          return !(operator == (rhs));
      }
      iterator& operator ++ ()
      {
          utf8::next(it, range_end);
          return *this;
      }
      iterator operator ++ (int)
      {
          iterator temp = *this;
          utf8::next(it, range_end);
          return temp;
      }
      iterator& operator -- ()
      {
          utf8::prior(it, range_start);
          return *this;
      }
      iterator operator -- (int)
      {
          iterator temp = *this;
          utf8::prior(it, range_start);
          return temp;
      }
    }; // class iterator

} // namespace utf8

#if UTF_CPP_CPLUSPLUS >= 202002L // C++ 20 or later
#include "cpp20.h"
#elif UTF_CPP_CPLUSPLUS >= 201703L // C++ 17 or later
#include "cpp17.h"
#elif UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
#include "cpp11.h"
#endif // C++ 11 or later

#endif //header guard

```

5.23 core.h

```
// Copyright 2006 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_CORE_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
#define UTF8_FOR_CPP_CORE_H_2675DCD0_9480_4c0c_B92A_CC14C027B731

#include <iterator>
#include <cstring>
#include <string>

// Determine the C++ standard version.
// If the user defines UTF_CPP_CPLUSPLUS, use that.
// Otherwise, trust the unreliable predefined macro __cplusplus

#if !defined UTF_CPP_CPLUSPLUS
    #define UTF_CPP_CPLUSPLUS __cplusplus
#endif

#if UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
    #define UTF_CPP_OVERRIDE override
    #define UTF_CPP_NOEXCEPT noexcept
#else // C++ 98/03
    #define UTF_CPP_OVERRIDE
    #define UTF_CPP_NOEXCEPT throw()
#endif // C++ 11 or later


namespace utf8
{
// The typedefs for 8-bit, 16-bit and 32-bit code units
#if UTF_CPP_CPLUSPLUS >= 201103L // C++ 11 or later
    #if UTF_CPP_CPLUSPLUS >= 202002L // C++ 20 or later
        typedef char8_t         utfchar8_t;
    #else // C++ 11/14/17
        typedef unsigned char   utfchar8_t;
    #endif
    typedef char16_t        utfchar16_t;
    typedef char32_t        utfchar32_t;
#else // C++ 98/03
    typedef unsigned char   utfchar8_t;
    typedef unsigned short  utfchar16_t;
    typedef unsigned int    utfchar32_t;
#endif // C++ 11 or later

// Helper code - not intended to be directly called by the library users. May be changed at any time
namespace internal
{
    // Unicode constants
    // Leading (high) surrogates: 0xd800 - 0xdbff
    // Trailing (low) surrogates: 0xdc00 - 0xdfff
    const utfchar16_t LEAD_SURROGATE_MIN  = 0xd800u;
    const utfchar16_t LEAD_SURROGATE_MAX  = 0xdbffu;
    const utfchar16_t TRAIL_SURROGATE_MIN = 0xdc00u;
    const utfchar16_t TRAIL_SURROGATE_MAX = 0xdfffu;
    const utfchar16_t LEAD_OFFSET         = 0xd7c0u;       // LEAD_SURROGATE_MIN - (0x10000 » 10)
    const utfchar32_t SURROGATE_OFFSET    = 0xfca02400u;   // 0x10000u - (LEAD_SURROGATE_MIN « 10) - TRAIL_SURROGATE_MIN

    // Maximum valid value for a Unicode code point
    const utfchar32_t CODE_POINT_MAX      = 0x0010ffffu;

    template<typename octet_type>
    inline utfchar8_t mask8(octet_type oc)
    {
        return static_cast<utfchar8_t>(0xff & oc);
    }
    template<typename u16_type>
    inline utfchar16_t mask16(u16_type oc)
    {
        return static_cast<utfchar16_t>(0xffff & oc);
    }

    template<typename octet_type>
    inline bool is_trail(octet_type oc)
    {
        return ((utf8::internal::mask8(oc) » 6) == 0x2);
    }

    inline bool is_lead_surrogate(utfchar32_t cp)
    {
        return (cp >= LEAD_SURROGATE_MIN && cp <= LEAD_SURROGATE_MAX);
    }

    inline bool is_trail_surrogate(utfchar32_t cp)
    {
        return (cp >= TRAIL_SURROGATE_MIN && cp <= TRAIL_SURROGATE_MAX);
    }

    inline bool is_surrogate(utfchar32_t cp)
    {
        return (cp >= LEAD_SURROGATE_MIN && cp <= TRAIL_SURROGATE_MAX);
    }

    inline bool is_code_point_valid(utfchar32_t cp)
    {
        return (cp <= CODE_POINT_MAX && !utf8::internal::is_surrogate(cp));
    }

    inline bool is_in_bmp(utfchar32_t cp)
    {
        return cp < utfchar32_t(0x10000);
    }

    template <typename octet_iterator>
    int sequence_length(octet_iterator lead_it)
    {
        const utfchar8_t lead = utf8::internal::mask8(*lead_it);
        if (lead < 0x80)
            return 1;
        else if ((lead » 5) == 0x6)
            return 2;
        else if ((lead » 4) == 0xe)
            return 3;
        else if ((lead » 3) == 0x1e)
            return 4;
        else
            return 0;
    }

    inline bool is_overlong_sequence(utfchar32_t cp, int length)
    {
        if (cp < 0x80) {
            if (length != 1)
                return true;
        }
        else if (cp < 0x800) {
            if (length != 2)
                return true;
        }
        else if (cp < 0x10000) {
            if (length != 3)
                return true;
        }
        return false;
    }

    enum utf_error {UTF8_OK, NOT_ENOUGH_ROOM, INVALID_LEAD, INCOMPLETE_SEQUENCE, OVERLONG_SEQUENCE, INVALID_CODE_POINT};

    template <typename octet_iterator>
    utf_error increase_safely(octet_iterator& it, const octet_iterator end)
    {
        if (++it == end)
            return NOT_ENOUGH_ROOM;

        if (!utf8::internal::is_trail(*it))
            return INCOMPLETE_SEQUENCE;

        return UTF8_OK;
    }

    #define UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(IT, END) {utf_error ret = increase_safely(IT, END); if (ret != UTF8_OK) return ret;}

    template <typename octet_iterator>
    utf_error get_sequence_1(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
            return NOT_ENOUGH_ROOM;

        code_point = utf8::internal::mask8(*it);

        return UTF8_OK;
    }

    template <typename octet_iterator>
    utf_error get_sequence_2(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
```

```
            return NOT_ENOUGH_ROOM;

        code_point = utf8::internal::mask8(*it);

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = ((code_point « 6) & 0x7ff) + ((*it) & 0x3f);

        return UTF8_OK;
    }

    template <typename octet_iterator>
    utf_error get_sequence_3(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
            return NOT_ENOUGH_ROOM;

        code_point = utf8::internal::mask8(*it);

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = ((code_point « 12) & 0xffff) + ((utf8::internal::mask8(*it) « 6) & 0xfff);

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = static_cast<utfchar32_t>(code_point + ((*it) & 0x3f));

        return UTF8_OK;
    }

    template <typename octet_iterator>
    utf_error get_sequence_4(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
           return NOT_ENOUGH_ROOM;

        code_point = utf8::internal::mask8(*it);

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = ((code_point « 18) & 0x1fffff) + ((utf8::internal::mask8(*it) « 12) & 0x3ffff);

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = static_cast<utfchar32_t>(code_point + ((utf8::internal::mask8(*it) « 6) & 0xfff));

        UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR(it, end)

        code_point = static_cast<utfchar32_t>(code_point + ((*it) & 0x3f));

        return UTF8_OK;
    }

    #undef UTF8_CPP_INCREASE_AND_RETURN_ON_ERROR

    template <typename octet_iterator>
    utf_error validate_next(octet_iterator& it, octet_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
            return NOT_ENOUGH_ROOM;

        // Save the original value of it so we can go back in case of failure
        // Of course, it does not make much sense with i.e. stream iterators
        octet_iterator original_it = it;

        utfchar32_t cp = 0;
        // Determine the sequence length based on the lead octet
        const int length = utf8::internal::sequence_length(it);

        // Get trail octets and calculate the code point
        utf_error err = UTF8_OK;
        switch (length) {
            case 0:
                return INVALID_LEAD;
            case 1:
                err = utf8::internal::get_sequence_1(it, end, cp);
                break;
            case 2:
                err = utf8::internal::get_sequence_2(it, end, cp);
            break;
            case 3:
                err = utf8::internal::get_sequence_3(it, end, cp);
            break;
            case 4:
                err = utf8::internal::get_sequence_4(it, end, cp);
            break;
        }

        if (err == UTF8_OK) {
            // Decoding succeeded. Now, security checks...
            if (utf8::internal::is_code_point_valid(cp)) {
                if (!utf8::internal::is_overlong_sequence(cp, length)){
                    // Passed! Return here.
                    code_point = cp;
                    ++it;
                    return UTF8_OK;
                }
                else
                    err = OVERLONG_SEQUENCE;
            }
            else
                err = INVALID_CODE_POINT;
        }

        // Failure branch - restore the original value of the iterator
        it = original_it;
        return err;
    }

    template <typename octet_iterator>
    inline utf_error validate_next(octet_iterator& it, octet_iterator end) {
        utfchar32_t ignored;
        return utf8::internal::validate_next(it, end, ignored);
    }

    template <typename word_iterator>
    utf_error validate_next16(word_iterator& it, word_iterator end, utfchar32_t& code_point)
    {
        if (it == end)
            return NOT_ENOUGH_ROOM;
        // Save the original value of it so we can go back in case of failure
        // Of course, it does not make much sense with i.e. stream iterators
        word_iterator original_it = it;

        utf_error err = UTF8_OK;

        const utfchar16_t first_word = *it++;
        if (!is_surrogate(first_word)) {
            code_point = first_word;
            return UTF8_OK;
        }
        else {
            if (it == end)
                err = NOT_ENOUGH_ROOM;
            else if (is_lead_surrogate(first_word)) {
                const utfchar16_t second_word = *it++;
                if (is_trail_surrogate(second_word)) {
                    code_point = static_cast<utfchar32_t>(first_word « 10) + second_word +
SURROGATE_OFFSET;
                    return UTF8_OK;
                } else
                    err = INCOMPLETE_SEQUENCE;

            } else {
                err = INVALID_LEAD;
            }
        }
        // error branch
        it = original_it;
        return err;
    }

    // Internal implementation of both checked and unchecked append() function
    // This function will be invoked by the overloads below, as they will know
    // the octet_type.
    template <typename octet_iterator, typename octet_type>
    octet_iterator append(utfchar32_t cp, octet_iterator result) {
        if (cp < 0x80)                        // one octet
            *(result++) = static_cast<octet_type>(cp);
        else if (cp < 0x800) {                // two octets
            *(result++) = static_cast<octet_type>((cp » 6)          | 0xc0);
            *(result++) = static_cast<octet_type>((cp & 0x3f)        | 0x80);
        }
        else if (cp < 0x10000) {              // three octets
            *(result++) = static_cast<octet_type>((cp » 12)         | 0xe0);
            *(result++) = static_cast<octet_type>(((cp » 6) & 0x3f) | 0x80);
            *(result++) = static_cast<octet_type>((cp & 0x3f)        | 0x80);
        }
        else {                                // four octets
            *(result++) = static_cast<octet_type>((cp » 18)         | 0xf0);
            *(result++) = static_cast<octet_type>(((cp » 12) & 0x3f)| 0x80);
            *(result++) = static_cast<octet_type>(((cp » 6) & 0x3f) | 0x80);
            *(result++) = static_cast<octet_type>((cp & 0x3f)        | 0x80);
        }
        return result;
    }

    // One of the following overloads will be invoked from the API calls

    // A simple (but dangerous) case: the caller appends byte(s) to a char array
    inline char* append(utfchar32_t cp, char* result) {
        return append<char*, char>(cp, result);
    }

    // Hopefully, most common case: the caller uses back_inserter
    // i.e. append(cp, std::back_inserter(str));
    template<typename container_type>
    std::back_insert_iterator<container_type> append
            (utfchar32_t cp, std::back_insert_iterator<container_type> result) {
        return append<std::back_insert_iterator<container_type>,
            typename container_type::value_type>(cp, result);
    }

    // The caller uses some other kind of output operator - not covered above
    // Note that in this case we are not able to determine octet_type
    // so we assume it’s utfchar8_t; that can cause a conversion warning if we are wrong.
    template <typename octet_iterator>
    octet_iterator append(utfchar32_t cp, octet_iterator result) {
        return append<octet_iterator, utfchar8_t>(cp, result);
    }

    // Internal implementation of both checked and unchecked append16() function
    // This function will be invoked by the overloads below, as they will know
    // the word_type.
    template <typename word_iterator, typename word_type>
    word_iterator append16(utfchar32_t cp, word_iterator result) {
        if (is_in_bmp(cp))
            *(result++) = static_cast<word_type>(cp);
        else {
            // Code points from the supplementary planes are encoded via surrogate pairs
            *(result++) = static_cast<word_type>(LEAD_OFFSET + (cp » 10));
            *(result++) = static_cast<word_type>(TRAIL_SURROGATE_MIN + (cp & 0x3FF));
        }
        return result;
    }

    // Hopefully, most common case: the caller uses back_inserter
    // i.e. append16(cp, std::back_inserter(str));
    template<typename container_type>
    std::back_insert_iterator<container_type> append16
            (utfchar32_t cp, std::back_insert_iterator<container_type> result) {
        return append16<std::back_insert_iterator<container_type>,
            typename container_type::value_type>(cp, result);
    }

    // The caller uses some other kind of output operator - not covered above
    // Note that in this case we are not able to determine word_type
    // so we assume it’s utfchar16_t; that can cause a conversion warning if we are wrong.
    template <typename word_iterator>
    word_iterator append16(utfchar32_t cp, word_iterator result) {
        return append16<word_iterator, utfchar16_t>(cp, result);
    }

} // namespace internal


    // Byte order mark
    const utfchar8_t bom[] = {0xef, 0xbb, 0xbf};

    template <typename octet_iterator>
    octet_iterator find_invalid(octet_iterator start, octet_iterator end)
    {
        octet_iterator result = start;
        while (result != end) {
            utf8::internal::utf_error err_code = utf8::internal::validate_next(result, end);
            if (err_code != internal::UTF8_OK)
                return result;
        }
        return result;
    }

    inline const char* find_invalid(const char* str)
    {
        const char* end = str + std::strlen(str);
        return find_invalid(str, end);
    }

    inline std::size_t find_invalid(const std::string& s)
    {
        std::string::const_iterator invalid = find_invalid(s.begin(), s.end());
        return (invalid == s.end()) ? std::string::npos : static_cast<std::size_t>(invalid - s.begin());
    }

    template <typename octet_iterator>
    inline bool is_valid(octet_iterator start, octet_iterator end)
    {
        return (utf8::find_invalid(start, end) == end);
    }

    inline bool is_valid(const char* str)
    {
        return (*(utf8::find_invalid(str)) == '\0');
    }

    inline bool is_valid(const std::string& s)
    {
        return is_valid(s.begin(), s.end());
    }



    template <typename octet_iterator>
    inline bool starts_with_bom (octet_iterator it, octet_iterator end)
    {
        return (
            ((it != end) && (utf8::internal::mask8(*it++)) == bom[0]) &&
            ((it != end) && (utf8::internal::mask8(*it++)) == bom[1]) &&
            ((it != end) && (utf8::internal::mask8(*it))   == bom[2])
           );
    }

    inline bool starts_with_bom(const std::string& s)
    {
        return starts_with_bom(s.begin(), s.end());
    }
} // namespace utf8

#endif // header guard


```

5.24 cpp11.h

```
// Copyright 2018 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_a184c22c_d012_11e8_a8d5_f2801f1b9fd1
#define UTF8_FOR_CPP_a184c22c_d012_11e8_a8d5_f2801f1b9fd1

#include "checked.h"

namespace utf8
{
    inline void append16(utfchar32_t cp, std::u16string& s)
    {
        append16(cp, std::back_inserter(s));
    }

    inline std::string utf16to8(const std::u16string& s)
    {
        std::string result;
        utf16to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u16string utf8to16(const std::string& s)
    {
        std::u16string result;
        utf8to16(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::string utf32to8(const std::u32string& s)
    {
        std::string result;
        utf32to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u32string utf8to32(const std::string& s)
    {
        std::u32string result;
        utf8to32(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }
} // namespace utf8

#endif // header guard

```

5.25 cpp17.h

```
// Copyright 2018 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_7e906c01_03a3_4daf_b420_ea7ea952b3c9
#define UTF8_FOR_CPP_7e906c01_03a3_4daf_b420_ea7ea952b3c9

#include "cpp11.h"

namespace utf8
{
    inline std::string utf16to8(std::u16string_view s)
    {
        std::string result;
        utf16to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u16string utf8to16(std::string_view s)
    {
        std::u16string result;
        utf8to16(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::string utf32to8(std::u32string_view s)
    {
        std::string result;
        utf32to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u32string utf8to32(std::string_view s)
    {
        std::u32string result;
        utf8to32(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::size_t find_invalid(std::string_view s)
    {
        std::string_view::const_iterator invalid = find_invalid(s.begin(), s.end());
        return (invalid == s.end()) ? std::string_view::npos : static_cast<std::size_t>(invalid - s.begin());
    }

    inline bool is_valid(std::string_view s)
    {
        return is_valid(s.begin(), s.end());
    }

    inline std::string replace_invalid(std::string_view s, char32_t replacement)
    {
        std::string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
        return result;
    }

    inline std::string replace_invalid(std::string_view s)
    {
        std::string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline bool starts_with_bom(std::string_view s)
    {
        return starts_with_bom(s.begin(), s.end());
    }

} // namespace utf8

#endif // header guard

```

5.26 cpp20.h

```
// Copyright 2022 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_207e906c01_03a3_4daf_b420_ea7ea952b3c9
#define UTF8_FOR_CPP_207e906c01_03a3_4daf_b420_ea7ea952b3c9

#include "cpp17.h"

```

```
namespace utf8
{
    inline std::u8string utf16tou8(const std::u16string& s)
    {
        std::u8string result;
        utf16to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u8string utf16tou8(std::u16string_view s)
    {
        std::u8string result;
        utf16to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u16string utf8to16(const std::u8string& s)
    {
        std::u16string result;
        utf8to16(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u16string utf8to16(const std::u8string_view& s)
    {
        std::u16string result;
        utf8to16(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u8string utf32tou8(const std::u32string& s)
    {
        std::u8string result;
        utf32to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u8string utf32tou8(const std::u32string_view& s)
    {
        std::u8string result;
        utf32to8(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u32string utf8to32(const std::u8string& s)
    {
        std::u32string result;
        utf8to32(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::u32string utf8to32(const std::u8string_view& s)
    {
        std::u32string result;
        utf8to32(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline std::size_t find_invalid(const std::u8string& s)
    {
        std::u8string::const_iterator invalid = find_invalid(s.begin(), s.end());
        return (invalid == s.end()) ? std::string_view::npos : static_cast<std::size_t>(invalid -
s.begin());
    }

    inline bool is_valid(const std::u8string& s)
    {
        return is_valid(s.begin(), s.end());
    }

    inline std::u8string replace_invalid(const std::u8string& s, char32_t replacement)
    {
        std::u8string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
        return result;
    }

    inline std::u8string replace_invalid(const std::u8string& s)
    {
        std::u8string result;
        replace_invalid(s.begin(), s.end(), std::back_inserter(result));
        return result;
    }

    inline bool starts_with_bom(const std::u8string& s)
    {
        return starts_with_bom(s.begin(), s.end());
    }

} // namespace utf8

#endif // header guard

```

5.27 unchecked.h

```
// Copyright 2006 Nemanja Trifunovic

/*
Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
*/


#ifndef UTF8_FOR_CPP_UNCHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731
#define UTF8_FOR_CPP_UNCHECKED_H_2675DCD0_9480_4c0c_B92A_CC14C027B731

#include "core.h"

namespace utf8
{
    namespace unchecked
    {
        template <typename octet_iterator>
        octet_iterator append(utfchar32_t cp, octet_iterator result)
        {
            return internal::append(cp, result);
        }

        template <typename word_iterator>
        word_iterator append16(utfchar32_t cp, word_iterator result)
        {
            return internal::append16(cp, result);
        }

        template <typename octet_iterator, typename output_iterator>
        output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out, utfchar32_t replacement)
        {
            while (start != end) {
                octet_iterator sequence_start = start;
                internal::utf_error err_code = utf8::internal::validate_next(start, end);
                switch (err_code) {
                    case internal::UTF8_OK :
                        for (octet_iterator it = sequence_start; it != start; ++it)
                            *out++ = *it;
                        break;
                    case internal::NOT_ENOUGH_ROOM:
                        out = utf8::unchecked::append(replacement, out);
                        start = end;
                        break;
                    case internal::INVALID_LEAD:
                        out = utf8::unchecked::append(replacement, out);
                        ++start;
                        break;
                    case internal::INCOMPLETE_SEQUENCE:
                    case internal::OVERLONG_SEQUENCE:
                    case internal::INVALID_CODE_POINT:
                        out = utf8::unchecked::append(replacement, out);
                        ++start;
                        // just one replacement mark for the sequence
                        while (start != end && utf8::internal::is_trail(*start))
                            ++start;
                        break;
                }
            }
            return out;
        }

        template <typename octet_iterator, typename output_iterator>
        inline output_iterator replace_invalid(octet_iterator start, octet_iterator end, output_iterator out)
        {
            static const utfchar32_t replacement_marker = utf8::internal::mask16(0xfffd);
            return utf8::unchecked::replace_invalid(start, end, out, replacement_marker);
        }

        inline std::string replace_invalid(const std::string& s, utfchar32_t replacement)
        {
            std::string result;
            replace_invalid(s.begin(), s.end(), std::back_inserter(result), replacement);
            return result;
        }

        inline std::string replace_invalid(const std::string& s)
        {
            std::string result;
            replace_invalid(s.begin(), s.end(), std::back_inserter(result));
            return result;
        }

        template <typename octet_iterator>
        utfchar32_t next(octet_iterator& it)
        {
            utfchar32_t cp = utf8::internal::mask8(*it);
            switch (utf8::internal::sequence_length(it)) {
                case 1:
                    break;
                case 2:
                    it++;
                    cp = ((cp « 6) & 0x7ff) + ((*it) & 0x3f);
                    break;
                case 3:
                    ++it;
                    cp = ((cp « 12) & 0xffff) + ((utf8::internal::mask8(*it) « 6) & 0xfff);
                    ++it;
                    cp = static_cast<utfchar32_t>(cp + ((*it) & 0x3f));
                    break;
                case 4:
                    ++it;
                    cp = ((cp « 18) & 0x1fffff) + ((utf8::internal::mask8(*it) « 12) & 0x3ffff);
                    ++it;
                    cp = static_cast<utfchar32_t>(cp + ((utf8::internal::mask8(*it) « 6) & 0xfff));
                    ++it;
                    cp = static_cast<utfchar32_t>(cp + ((*it) & 0x3f));
                    break;
            }
            ++it;
            return cp;
        }

        template <typename octet_iterator>
        utfchar32_t peek_next(octet_iterator it)
        {
            return utf8::unchecked::next(it);
        }

        template <typename word_iterator>
        utfchar32_t next16(word_iterator& it)
        {
            utfchar32_t cp = utf8::internal::mask16(*it++);
            if (utf8::internal::is_lead_surrogate(cp))
                return (cp « 10) + *it++ + utf8::internal::SURROGATE_OFFSET;
            return cp;
        }

        template <typename octet_iterator>
        utfchar32_t prior(octet_iterator& it)
        {
            while (utf8::internal::is_trail(*(--it))) ;
            octet_iterator temp = it;
            return utf8::unchecked::next(temp);
        }

        template <typename octet_iterator, typename distance_type>
        void advance(octet_iterator& it, distance_type n)
        {
            const distance_type zero(0);
            if (n < zero) {
                // backward
                for (distance_type i = n; i < zero; ++i)
                    utf8::unchecked::prior(it);
            } else {
                // forward
                for (distance_type i = zero; i < n; ++i)
                    utf8::unchecked::next(it);
            }
        }

        template <typename octet_iterator>
        typename std::iterator_traits<octet_iterator>::difference_type
        distance(octet_iterator first, octet_iterator last)
        {
            typename std::iterator_traits<octet_iterator>::difference_type dist;
            for (dist = 0; first < last; ++dist)
                utf8::unchecked::next(first);
            return dist;
        }

        template <typename u16bit_iterator, typename octet_iterator>
        octet_iterator utf16to8(u16bit_iterator start, u16bit_iterator end, octet_iterator result)
        {
            while (start != end) {
                utfchar32_t cp = utf8::internal::mask16(*start++);
                // Take care of surrogate pairs first
                if (utf8::internal::is_lead_surrogate(cp)) {
                    if (start == end)
                        return result;
                    utfchar32_t trail_surrogate = utf8::internal::mask16(*start++);
                    cp = (cp « 10) + trail_surrogate + internal::SURROGATE_OFFSET;
                }
                result = utf8::unchecked::append(cp, result);
            }
            return result;
        }

        template <typename u16bit_iterator, typename octet_iterator>
        u16bit_iterator utf8to16(octet_iterator start, octet_iterator end, u16bit_iterator result)
        {
            while (start < end) {
                utfchar32_t cp = utf8::unchecked::next(start);
                if (cp > 0xffff) { //make a surrogate pair
                    *result++ = static_cast<utfchar16_t>((cp » 10)    + internal::LEAD_OFFSET);
                    *result++ = static_cast<utfchar16_t>((cp & 0x3ff) +
internal::TRAIL_SURROGATE_MIN);
                }
                else
                    *result++ = static_cast<utfchar16_t>(cp);
            }
            return result;
        }

        template <typename octet_iterator, typename u32bit_iterator>
        octet_iterator utf32to8(u32bit_iterator start, u32bit_iterator end, octet_iterator result)
        {
            while (start != end)
                result = utf8::unchecked::append(*(start++), result);

            return result;
        }

        template <typename octet_iterator, typename u32bit_iterator>
        u32bit_iterator utf8to32(octet_iterator start, octet_iterator end, u32bit_iterator result)
        {
            while (start < end)
                (*result++) = utf8::unchecked::next(start);

            return result;
        }

        // The iterator class
        template <typename octet_iterator>
          class iterator {
            octet_iterator it;
            public:
            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;
            iterator () {}
            explicit iterator (const octet_iterator& octet_it): it(octet_it) {}
            // the default "big three" are OK
            octet_iterator base () const { return it; }
            utfchar32_t operator * () const
            {
                octet_iterator temp = it;
                return utf8::unchecked::next(temp);
            }
            bool operator == (const iterator& rhs) const
            {
                return (it == rhs.it);
            }
            bool operator != (const iterator& rhs) const
            {
                return !(operator == (rhs));
            }
            iterator& operator ++ ()
            {
                ::std::advance(it, utf8::internal::sequence_length(it));
                return *this;
            }
            iterator operator ++ (int)
            {
                iterator temp = *this;
                ::std::advance(it, utf8::internal::sequence_length(it));
                return temp;
            }
            iterator& operator -- ()
            {
                utf8::unchecked::prior(it);
                return *this;
            }
            iterator operator -- (int)
            {
                iterator temp = *this;
                utf8::unchecked::prior(it);
                return temp;
            }
          }; // class iterator

    } // namespace utf8::unchecked
} // namespace utf8


#endif // header guard

```

5.28 StringHandler.hpp

```
#ifndef STRING_HANDLER_HPP
#define STRING_HANDLER_HPP

#include <string>

namespace StringHandler {
    template <typename Object>
    struct SetWidthAtLeft {
        const Object& obj;
        size_t width;

        SetWidthAtLeft(const Object& o, size_t w)
            : obj(o), width(w) {}
    };


    template <typename Object>
    std::string toString(const Object& obj);

    template <typename Object>
    size_t size(const Object& obj);

    template <typename Object>
    std::ostream& operator«(std::ostream& os, const SetWidthAtLeft<Object>& manip);
}

#include "Utils/StringHandler.impl.hpp"

#endif
```

5.29 StringHandler.impl.hpp

```
#include "Utils/StringHandler.hpp"
```

```

#include <sstream>

#include "utf8.h"

namespace StringHandler {
    template <typename Object>
    std::string toString(const Object& obj) {
        std::ostringstream oss;
        oss « obj;
        return oss.str();
    }

    template <typename Object>
    size_t size(const Object& obj) {
        std::string str = toString(obj);
        size_t count = 0;

        auto it = str.begin();
        auto end = str.end();

        while (it != end) {
            utf8::next(it, end);
            count++;
        }

        return count;
    }

    template <typename Object>
    std::ostream& operator«(std::ostream& os, const SetWidthAtLeft<Object>& manip) {
        std::string str = toString(manip.obj);

        size_t realSize = StringHandler::size(manip.obj),
            padding = (manip.width > realSize ? manip.width - realSize : 0);

        os « str « std::string(padding, ’ ’);

        return os;
    }
}
```

Index