Dictionary< Key, Info > Class Template Reference

Classes

class	**iterator**
class	**reverse_iterator**

Public Types

enum	**TravelType** { **Preorder**, **Inorder**, **Postorder** } Types of traversing the tree.
typedef const **iterator**	**const_iterator**
typedef const **reverse_iterator**	**const_reverse_iterator**

Public Member Functions

iterator	**begin** () const
iterator	**end** () const
reverse_iterator	**rbegin** () const
reverse_iterator	**rend** () const
	Dictionary ()
	Dictionary (const **Dictionary**< Key, Info > &x)
	~Dictionary ()
int	**getHeight** () const
bool	**empty** () const
void	**clear** ()
Dictionary &	**operator=** (const **Dictionary**< Key, Info > &x)
bool	**operator==** (const **Dictionary**< Key, Info > &x) const
bool	**operator!=** (const **Dictionary**< Key, Info > &x) const
bool	**insert** (const Key &key, const Info &info)
bool	**insert** (const Key &key, const Info &info, **iterator** &it)
bool	**remove** (const Key &key)
template<typename ToDo >	
void	**executeForAllNodes** (ToDo method, **TravelType** type=TravelType::Preorder) const
void	**traversal** (const **TravelType** type, std::ostream &out=std::cout) const
void	**preorder** (std::ostream &out=std::cout) const
void	**inorder** (std::ostream &out=std::cout) const
void	**postorder** (std::ostream &out=std::cout) const
iterator	**find** (const Key &key) const
template<typename ToDo >	
void	**executeForAllNodes** (ToDo method, **Dictionary**< Key, Info >::**TravelType** type) const

Member Typedef Documentation

◆ const_iterator

template<typename Key , typename Info >

typedef const **iterator** **Dictionary**< Key, Info >::**const_iterator**

The iterator class is capable of traversing throught a constant **Dictionary** because Node* is specified as mutable. Instead of writing a seperate class ill specift a constant iterator like that:

◆ const_reverse_iterator

template<typename Key , typename Info >

typedef const **reverse_iterator Dictionary**< Key, Info >::**const_reverse_iterator**

The reverse iterator class is capable of traversing throught a constant **Dictionary** because Node* is specified as mutable. Instead of writing a seperate class ill specift a constant **reverse_iterator** like that:

Constructor & Destructor Documentation

◆ Dictionary() [1/2]

template<typename Key , typename Info >

Dictionary< Key, Info >::**Dictionary**

Constructor creates an empty tree.

◆ Dictionary() [2/2]

template<typename Key , typename Info >

Dictionary< Key, Info >::**Dictionary** (const **Dictionary**< Key, Info > & x)

Copy constructor.

◆ ~Dictionary()

template<typename Key , typename Info >

Dictionary< Key, Info >::**~Dictionary**

Destructor.

Member Function Documentation

◆ begin()

template<typename Key , typename Info >

Dictionary< Key, Info >::**iterator Dictionary**< Key, Info >::begin

An iterator to the smalles element of the tree. Complexity O(logN) where N is a number of element or O(h) here h is a height of the tree.

◆ clear()

template<typename Key , typename Info >

void **Dictionary**< Key, Info >::clear

Deletes all of the elements from the tree.

◆ empty()

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::empty

Checks whether the tree is empty.

◆ end()

template<typename Key , typename Info >

Dictionary< Key, Info >::**iterator Dictionary**< Key, Info >::end

An iterator to the end.

◆ executeForAllNodes()

template<typename Key , typename Info >

template<typename ToDo >

void **Dictionary**< Key, Info >::executeForAllNodes (ToDo method,
TravelType type = `TravelType::Preorder`
) const

Functionality: Executes set of instruction for all of the nodes of the AVL tree. Approche: Recursive method executes ToDo method according to the TravelType (preorder by default) ToDo method do need to have const_iterator to the element at the imput. In order to work properly. Proper construction of method: [](**Dictionary<int,int>::const_iterator**& in)->void {...} Auto is also fine: [](auto in)->void {...} param[in] method : Aforementioned method specifing todo operations in every step. param[in] type : By default TravelType::Preorder. Specifies the type of traversing.

◆ find()

template<typename Key , typename Info >

Dictionary< Key, Info >::**iterator Dictionary**< Key, Info >::find (const Key & key) const

The method finds an element with a given key. If an element was not found the empty iterator is being returned. To check whether thew iterator has a walue use InNull method. param[in] key : Key to find a value.

◆ getHeight()

template<typename Key , typename Info >

int **Dictionary**< Key, Info >::getHeight

Outputs the height of a tree. O(1).

◆ inorder()

template<typename Key , typename Info >

void **Dictionary**< Key, Info >::inorder (std::ostream & out = `std::cout`) const

This is classic BST traversals. I have implemented bacause traversal method requires some additional typing. param[in] out : By default std::cout. Specifies the ostream variable.

◆ insert() [1/2]

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::insert (const Key & key,
const Info & info
)

Functionallity: Inserts the node to the AVL tree. Approche: I am using recursive insert. There is possible throw when key is already in the tree, however, public method outputs true/false in that case. param[in] key : Key of the element that is going to be inserted. param[in] info : Info of the element that is going to be inserted.

◆ insert() [2/2]

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::insert (const Key & key,
const Info & info,
iterator & it
)

Functionallity: Slightly modified insert method. It outputs the iterator to the a new element or an iterator to existing element. Approche: I am using recursive insert. There is possible throw when key is already in the tree, however, public method outputs true/false in that case. param[in] key : Key of the element that is going to be inserted. param[in] info : Info of the element that is going to be inserted. param[in] it : A reference to iterator to which we want to provide information about the added/existing element.

◆ operator!=()

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::operator!= (const **Dictionary**< Key, Info > & x) const

Comparision operator.

◆ operator=()

template<typename Key , typename Info >

Dictionary< Key, Info > & **Dictionary**< Key, Info >::operator= (const **Dictionary**< Key, Info > & x)

Assign operator.

◆ operator==()

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::operator== (const **Dictionary**< Key, Info > & x) const

Comparision operator.

◆ postorder()

template<typename Key , typename Info >

void **Dictionary**< Key, Info >::postorder (std::ostream & out = `std::cout`) const

This is classic BST traversals. I have implemented bacause traversal method requires some additional typing. param[in] out : By default std::cout. Specifies the ostream variable.

◆ preorder()

template<typename Key , typename Info >

void **Dictionary**< Key, Info >::preorder (std::ostream & out = `std::cout`) const

This is classic BST traversals. I have implemented bacause traversal method requires some additional typing. param[in] out : By default std::cout. Specifies the ostream variable.

◆ rbegin()

template<typename Key , typename Info >

Dictionary< Key, Info >::**reverse_iterator Dictionary**< Key, Info >::rbegin

An reverese iterator to the biggest element of the tree. Complexity O(logN) where N is a number of element or O(h) here h is a height of the tree.

◆ remove()

template<typename Key , typename Info >

bool **Dictionary**< Key, Info >::remove (const Key & key)

Functionality: Delete the node from the AVL tree. Approche: I am using recursive remove. There is possible throw when key is not a memeber of the tree, however, public method outputs true/false in that case. param[in] key : Key of the element that is going to be deleted.

◆ rend()

template<typename Key , typename Info >

Dictionary< Key, Info >::**reverse_iterator Dictionary**< Key, Info >::rend

An iterator to the end.

◆ traversal()

template<typename Key , typename Info >

void **Dictionary**< Key, Info >::traversal (const **TravelType** type,
std::ostream & out = `std::cout`
) const

There are also three classic BST traversals. param[in] type : Travel type. param[in] out : By default std::cout. Specifies the ostream variable.