Binary Search Tree AP DSSC 20/21

Group Romina Doz, Davide Basso, Imola Fodor

Generated by Doxygen 1.9.1

1 Advanced programming exam 2020/2021 1

2 Class Index 3

2.1 Class List . . . 3

3 Class Documentation 5

3.1 bst< key_type, value_type, comparison > Class Template Reference . . . 5

3.1.1 Detailed Description . . . 7

3.1.2 Member Typedef Documentation . . . 7

3.1.2.1 const_iterator . . . 7

3.1.2.2 iterator . . . 7

3.1.2.3 node_type . . . 7

3.1.2.4 pair_type . . . 8

3.1.3 Member Function Documentation . . . 8

3.1.3.1 _find() . . . 8

3.1.3.2 _insert() . . . 8

3.1.3.3 _print2D() . . . 8

3.1.3.4 balance() . . . 8

3.1.3.5 begin() [1/2] . . . 9

3.1.3.6 begin() [2/2] . . . 9

3.1.3.7 cbegin() . . . 9

3.1.3.8 cend() . . . 9

3.1.3.9 clear() . . . 9

3.1.3.10 emplace() . . . 10

3.1.3.11 end() [1/2] . . . 10

3.1.3.12 end() [2/2] . . . 10

3.1.3.13 erase() . . . 10

3.1.3.14 find() [1/2] . . . 10

3.1.3.15 find() [2/2] . . . 11

3.1.3.16 insert() [1/2] . . . 11

3.1.3.17 insert() [2/2] . . . 11

3.1.3.18 insert_balanced_node() . . . 11

3.1.3.19 operator[]() [1/2] . . . 11

3.1.3.20 operator[]() [2/2] . . . 12

3.1.3.21 print2D() . . . 12

3.1.3.22 repopulate() . . . 12

3.1.3.23 size() . . . 12

3.1.4 Friends And Related Function Documentation . . . 12

3.1.4.1 operator<< . . . 13

3.1.5 Member Data Documentation . . . 13

3.1.5.1 _size . . . 13

3.1.5.2 comp . . . 13

3.1.5.3 head . . . 13

3.2 Iterator< pair_type, node_type > Class Template Reference . . . 13
3.2.1 Detailed Description . . . 14
3.2.2 Constructor & Destructor Documentation . . . 15
3.2.2.1 Iterator() . . . 15
3.2.3 Member Function Documentation . . . 15
3.2.3.1 next() . . . 15
3.2.3.2 operator*() . . . 15
3.2.3.3 operator++() [1/2] . . . 15
3.2.3.4 operator++() [2/2] . . . 16
3.2.3.5 operator->() . . . 16
3.2.4 Friends And Related Function Documentation . . . 16
3.2.4.1 operator"!= . . . 16
3.2.4.2 operator== . . . 16
3.2.5 Member Data Documentation . . . 17
3.2.5.1 current . . . 17
3.3 node< pair_type > Class Template Reference . . . 17
3.3.1 Detailed Description . . . 18
3.3.2 Constructor & Destructor Documentation . . . 19
3.3.2.1 node() [1/5] . . . 19
3.3.2.2 node() [2/5] . . . 19
3.3.2.3 node() [3/5] . . . 19
3.3.2.4 node() [4/5] . . . 19
3.3.2.5 node() [5/5] . . . 20
3.3.3 Member Function Documentation . . . 20
3.3.3.1 _leftmost() . . . 20
3.3.3.2 get_data() [1/2] . . . 20
3.3.3.3 get_data() [2/2] . . . 20
3.3.3.4 get_left() . . . 21
3.3.3.5 get_parent() . . . 21
3.3.3.6 get_right() . . . 21
3.3.3.7 leftiest() [1/2] . . . 21
3.3.3.8 leftiest() [2/2] . . . 22
3.3.3.9 operator=() . . . 22
3.3.4 Member Data Documentation . . . 22
3.3.4.1 data . . . 22
3.3.4.2 left_child . . . 22
3.3.4.3 parent_node . . . 23
3.3.4.4 right_child . . . 23

Chapter 1

Advanced programming exam 2020/2021

Group: Romina Doz [SM3500441], Davide Basso [SM3500450], Imola Fodor [SM3500474]

Problem statement

Implement a templated binary search tree, a data structure with the following properties:

- The left subtree of a node contains only nodes with keys smaller than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- Left and right subtree each must also be a binary search tree

Each node of the tree has to store a key and a value and the tree has to be traversed from the smaller key to the biggest. This is the so called in-order traversal and is only one among the different options that we have to actually traverse the tree.

Command Line Arguments

The command line arguments are specified in the Makefile, so the program can be executed with the *make* command. However, these are the main commands:

```
-I include -g -std=c++17 -Wall -Wextra
```

- The folder *include* contains the following files: *include/bst.hpp include/node.hpp include/iterator.hpp*
- *-std=c++17* specifies the version of C++ to be used
- *-Wall -Wextra* asks for almost all warnings

Output

In *main()* all the functions that were implemented for the binary search tree are tested.

First, *insert()* and *emplace()* are used to create a bst. Then, with the *find()* function, it is possible to see whether a node is present and, in that case, show his value. *erase()* can delete a node and repopulate the subtree without the erased node. Once all these steps have been done, it is probable that the bst is not balanced (it means that it has not the minimum possible height). To solve this problem, *balance()* is used and the tree can be printed as an ordered sequence (using the operator $*<<*$) or in its 2D shape (using *print2D()*). Another tree with the same nodes of the previous one can be obtained using the *deep copy constructor*. A change in the second tree (made with *subscripting operator[]*) will not modify the original tree, as shown in the output. *Move assignment* is tested too and works properly. Finally all the trees generated are deleted, using the *clear()* function.

Note For all the calls to the above functions, the microseconds taken by the execution are printed, in order to have a performance index.

Design Decisions & Issues

The methods of bts described in the previous section are public but use private fuctions for the implementation. Iterators are often constructed inside these functions and can be used to traverse the tree.

The most important design decisions concern the following operations:

- *insert()* -> This function returns a pair of an iterator (pointing to the node) and a bool (true if the new node is added, false otherwise). When a new node has to be added to a bst object, it is inserted as a leaf node (without children). For this reason the tree is traversed from the top to the bottom (and not from left to right). This decision is made to avoid a self-balancing binary search tree and improve the performance of the program. If the node is already present in the bts, the original one will not be substituted.
- *emplace()* -> Inserts a new element into the container constructed in-place with the given args if there is no element with the key in the container. By in-place, we mean the element object is built in-place from the passed arguments.
- *erase()* -> Removes the node (if one exists) with a corresponding key. Once the node is found, its link with the parent is removed and the node is deleted. All the nodes of the subtree over the erased node are re-inserted recursively in the bst to keep the right ordered stucture of the binary search tree.
- *balance()* -> Can be used to change an existing tree in order to have the minimum possible height. To achieve this purpose, the nodes are stored in an ordered (by key) vector and the current tree is cleared. Then, the node in the center of the vector becomes the head of the tree and the vector is splitted in left and right part. The nodes placed in middle position of these new vectors are inserted then. Following this execution path, all the nodes are inserted recursively in the new tree.

Chapter 2

Class Index

2.1 Class List

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

bst< key_type, value_type, comparison > 5
Iterator< pair_type, node_type > 13
node< pair_type > 17

Chapter 3

Class Documentation

3.1 bst< key_type, value_type, comparison > Class Template Reference

`#include "include/node.hpp include/iterator.hpp"`

Public Member Functions

- bst ()=default
  Default bst Constructor.
- ~bst () noexcept=default
  Default bst Destructor.
- bst (bst &&other) noexcept=default
  Move constructor.
- bst & operator= (bst &&other) noexcept=default
  Move assignment.
- bst (const bst &other)
  Deep-copy constructor.
- bst & operator= (const bst &x)
  Deep-copy assignment.
- void clear ()
  delete tree
- void print2D () const noexcept
  pretty print
- iterator begin () noexcept
  begin of for loop with iterator
- const_iterator begin () const noexcept
  const begin of for loop with iterator
- const_iterator cbegin () const noexcept
  const begin of for loop with iterator
- iterator end () noexcept
  end of for loop with iterator
- const_iterator end () const noexcept
  const end of for loop with iterator
- const_iterator cend () const noexcept
  const end of for loop with iterator

- iterator find (const key_type &x) noexcept

  find element in tree
- const_iterator find (const key_type &x) const noexcept

  const find element in tree by key
- std::pair< iterator, bool > insert (const pair_type &x)

  insert node by pair
- std::pair< iterator, bool > insert (pair_type &&x)

  insert node by pair
- template<class... Types>
  std::pair< iterator, bool > emplace (Types &&... args)

  emplace element
- void erase (const key_type &x)

  erase element from tree
- void balance ()

  balance tree
- std::size_t size () const noexcept

  depth of tree
- value_type & operator[ ] (const key_type &x)

  subscripting l-value
- value_type & operator[ ] (key_type &&x)

  subscripting r-value
- template<typename O >
  std::pair< typename bst< key_type, value_type, comparison >::iterator, bool > **_insert** (O &&x)
- template<typename T >
  node< std::pair< const key_type, value_type > > * **_find** (T &&x) const noexcept

Private Types

- using pair_type = std::pair< const key_type, value_type >
- using node_type = node< pair_type >
- using const_iterator = Iterator< const pair_type, node_type >
- using iterator = Iterator< pair_type, node_type >

Private Member Functions

- template<typename O >
  std::pair< iterator, bool > _insert (O &&x)

  internal insert
- template<typename T >
  node_type * _find (T &&x) const noexcept

  internal find
- void _print2D (node_type *root, int space) const noexcept

  internal print2D
- void repopulate (node_type *child)

  repopulate the tree
- void insert_balanced_node (std::vector< pair_type > vec)

  recursive insert for balance

Private Attributes

- std::unique_ptr< node_type > head
  head
- std::size_t _size
  size
- comparison comp
  compare two nodes

Friends

- std::ostream & operator<< (std::ostream &os, const bst &x)
  put-to

3.1.1 Detailed Description

template<typename key_type, typename value_type, typename comparison = std::less<key_type>>
class bst< key_type, value_type, comparison >

Custom Binary Search Tree Template class. Every instance of the bst class is a hierarchical (ordered) data structure.

3.1.2 Member Typedef Documentation

3.1.2.1 const_iterator

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
using bst< key_type, value_type, comparison >::const_iterator = Iterator<const pair_type,
node_type>  [private]
```

using declaration for pair_type. Represents a constant iterator class, defined by a pair type and node type.

3.1.2.2 iterator

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
using bst< key_type, value_type, comparison >::iterator = Iterator<pair_type, node_type>
[private]
```

using declaration for pair_type. Represents an iterator class, defined by a pair type and node type.

3.1.2.3 node_type

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
using bst< key_type, value_type, comparison >::node_type = node<pair_type>  [private]
```

using declaration for pair_type. Represents a node type of a pair of key and associated value.

3.1.2.4 pair_type

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
using bst< key_type, value_type, comparison >::pair_type = std::pair<const key_type, value_↩
type>  [private]
```

using declaration for pair_type. Represents a pair type of key and associated value.

3.1.3 Member Function Documentation

3.1.3.1 _find()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
template<typename T >
node_type* bst< key_type, value_type, comparison >::_find (
            T && x ) const  [private], [noexcept]
```

internal find

Private function for finding a node based on key. x passed as r-value, of typename T.

3.1.3.2 _insert()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
template<typename O >
std::pair<iterator, bool> bst< key_type, value_type, comparison >::_insert (
            O && x )  [private]
```

internal insert

Private function to insert node. x passed as r-value, of typename O.

3.1.3.3 _print2D()

```
template<typename key_type , typename value_type , typename comparison >
void bst< key_type, value_type, comparison >::_print2D (
            node_type * root,
            int space ) const  [private], [noexcept]
```

internal print2D

Private function for printing tree in 2D

3.1.3.4 balance()

```
template<typename key_type , typename value_type , typename comparison >
void bst< key_type, value_type, comparison >::balance
```

balance tree

Function to balance the tree.

3.1.3.5 begin() [1/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
const_iterator bst< key_type, value_type, comparison >::begin ( ) const  [inline], [noexcept]
```

const begin of for loop with iterator

Return a const iterator to the left-most node (which, likely, is not the root node). The returning value is obtained using the fucntion leftiest(), which finds the leaf node placed at the very far left

3.1.3.6 begin() [2/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
iterator bst< key_type, value_type, comparison >::begin ( )  [inline], [noexcept]
```

begin of for loop with iterator

Return an iterator to the left-most node (which, likely, is not the root node). The returning value is obtained using the fucntion leftiest(), which finds the leaf node placed at the very far left

3.1.3.7 cbegin()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
const_iterator bst< key_type, value_type, comparison >::cbegin ( ) const  [inline], [noexcept]
```

const begin of for loop with iterator

Return a const iterator to the left-most node (which, likely, is not the root node). The returning value is obtained using the fucntion leftiest(), which finds the leaf node placed at the very far left

3.1.3.8 cend()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
const_iterator bst< key_type, value_type, comparison >::cend ( ) const  [inline], [noexcept]
```

const end of for loop with iterator

Returns a const iterator to one-past the last element. Basically a const_iterator initialized to a nullptr

3.1.3.9 clear()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
void bst< key_type, value_type, comparison >::clear ( )  [inline]
```

delete tree

Function to clear the contents of the tree by setting head to null-pointer.

3.1.3.10 emplace()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
template<class...  Types>
std::pair<iterator,bool> bst< key_type, value_type, comparison >::emplace (
            Types &&...  args )  [inline]
```

emplace element

Inserts a new element into the container constructed in-place with the given args if there is no element with the key in the container.

3.1.3.11 end() [1/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
const_iterator bst< key_type, value_type, comparison >::end ( ) const  [inline], [noexcept]
```

const end of for loop with iterator

Returns a const iterator to one-past the last element. Basically a const_iterator initialized to a nullptr

3.1.3.12 end() [2/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
iterator bst< key_type, value_type, comparison >::end ( )  [inline], [noexcept]
```

end of for loop with iterator

Returns an iterator to one-past the last element. Basically an iterator initialized to a nullptr

3.1.3.13 erase()

```
template<typename key_type , typename value_type , typename comparison >
void bst< key_type, value_type, comparison >::erase (
            const key_type & x )
```

erase element from tree

Function that removes the element (if one exists) with the key equivalent to key. When element found, deleted. After erase, tree repopulated. Takes const x , l-value reference of type key.

3.1.3.14 find() [1/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
const_iterator bst< key_type, value_type, comparison >::find (
            const key_type & x ) const  [inline], [noexcept]
```

const find element in tree by key

Find a given key, x passed as l-value. If the key is present, returns a const_iterator to the proper node; if not the function _find() returns a nullptr, so equivalent result as cend().

3.1.3.15 find() [2/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
iterator bst< key_type, value_type, comparison >::find (
            const key_type & x )  [inline], [noexcept]
```

find element in tree

Find a given key, x passed as l-value. If the key is present, returns an iterator to the proper node; if not the function _find() returns a nullptr, so equivalent result as end().

3.1.3.16 insert() [1/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::pair<iterator, bool> bst< key_type, value_type, comparison >::insert (
            const pair_type & x )  [inline]
```

insert node by pair

Function to insert node based on x , as const l-value reference pair type. The function returns a pair of an iterator (pointing to the node) and a bool. The bool is true if a new node has been allocated, false otherwise.

3.1.3.17 insert() [2/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::pair<iterator, bool> bst< key_type, value_type, comparison >::insert (
            pair_type && x )  [inline]
```

insert node by pair

Function to insert node based on x , as r-value reference pair type. The function returns a pair of an iterator (pointing to the node) and a bool. The bool is true if a new node has been allocated, false otherwise.

3.1.3.18 insert_balanced_node()

```
template<typename key_type , typename value_type , typename comparison >
void bst< key_type, value_type, comparison >::insert_balanced_node (
            std::vector< pair_type > vec )  [private]
```

recursive insert for balance

Auxiliary function for balance tree, recursive. The vector that holds the tree nodes, gets to be divided by the median. Each splitted vector gets its' median inserted into the new, balanced tree. Process repeated for each section divided by the median. Process stops when median doesn't hold anymore a positive value.

3.1.3.19 operator[]() [1/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
value_type& bst< key_type, value_type, comparison >::operator[] (
            const key_type & x )  [inline]
```

subscripting l-value

Subscripting operator, takes x as l-value reference, of type key. Returns a reference to the value that is mapped to a key equivalent to x, performing an insertion if such key does not already exist. Takes advantage of the already defined insert() function.

3.1.3.20 operator[]() [2/2]

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
value_type& bst< key_type, value_type, comparison >::operator[] (
            key_type && x )  [inline]
```

subscripting r-value

Subscripting operator, takes `x` as r-value reference, of type key. Returns a reference to the value that is mapped to a key equivalent to x, performing an insertion if such key does not already exist. Takes advantage of the already defined insert() function.

3.1.3.21 print2D()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
void bst< key_type, value_type, comparison >::print2D ( ) const  [inline], [noexcept]
```

pretty print

Function for a 2D design of the existing tree

3.1.3.22 repopulate()

```
template<typename key_type , typename value_type , typename comparison >
void bst< key_type, value_type, comparison >::repopulate (
            node_type * child )  [private]
```

repopulate the tree

Function to properly repopulate tree. `child` passed, pointer to a node type. Used after erasing a node in the tree.

3.1.3.23 size()

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::size_t bst< key_type, value_type, comparison >::size ( ) const  [inline], [noexcept]
```

depth of tree

Returns depth of the tree.

3.1.4 Friends And Related Function Documentation

3.1.4.1 operator<<

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::ostream& operator<< (
            std::ostream & os,
            const bst< key_type, value_type, comparison > & x )  [friend]
```

put-to

Put-to operator, takes instance of ostream, and x as l-value reference to bst type.

3.1.5 Member Data Documentation

3.1.5.1 _size

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::size_t bst< key_type, value_type, comparison >::_size  [private]
```

size

Size of the tree as_size .

3.1.5.2 comp

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
comparison bst< key_type, value_type, comparison >::comp  [private]
```

compare two nodes

Compare two nodes, ascomp.

3.1.5.3 head

```
template<typename key_type , typename value_type , typename comparison = std::less<key_type>>
std::unique_ptr<node_type> bst< key_type, value_type, comparison >::head  [private]
```

head

Root of the tree ashead

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

- include/bst.hpp

3.2 Iterator< pair_type, node_type > Class Template Reference

```
#include <iterator.hpp>
```

Public Types

- using **difference_type** = std::ptrdiff_t
- using **reference** = pair_type &
- using **pointer** = pair_type *
- using **iterator_category** = std::forward_iterator_tag

Public Member Functions

- node_type * next (node_type *cur) const noexcept

  next node
- Iterator ()=default

  Default iterator Constructor.
- ~Iterator () noexcept=default

  Default iterator Destructor.
- Iterator (node_type *other)

  Custom iterator Constructor.
- reference operator* () const noexcept

  star operator overload
- pointer operator-> () const noexcept

  -> overload
- Iterator & operator++ ()

  ++ overload
- Iterator operator++ (int)

  ++ overload

Private Attributes

- node_type * current

  pointer to current node

Friends

- bool operator== (const Iterator &lhs, const Iterator &rhs) noexcept

  == overload
- bool operator!= (const Iterator &lhs, const Iterator &rhs) noexcept

  != overload

3.2.1 Detailed Description

template<typename pair_type, typename node_type>
class Iterator< pair_type, node_type >

Custom Forward Iterator Template class for members of the binary search tree concept. Every instance of the Iterator class is a pointer to a Node type. Mainly used to tarverse the tree in order.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Iterator()

```
template<typename pair_type , typename node_type >
Iterator< pair_type, node_type >::Iterator (
            node_type * other )  [inline], [explicit]
```

Custom iterator Constructor.

Creates an iterator by receiving a pointer to node type as `other` .

3.2.3 Member Function Documentation

3.2.3.1 next()

```
template<typename pair_type , typename node_type >
node_type * Iterator< pair_type, node_type >::next (
            node_type * cur ) const  [noexcept]
```

next node

Function that returns pointer to the node that is successive to the current one, which is passed as a pointer `cur` .

3.2.3.2 operator*()

```
template<typename pair_type , typename node_type >
reference Iterator< pair_type, node_type >::operator* ( ) const  [inline], [noexcept]
```

star operator overload

Overloading of operator * to return contents of the current Iterator node instance.

3.2.3.3 operator++() [1/2]

```
template<typename pair_type , typename node_type >
Iterator& Iterator< pair_type, node_type >::operator++ ( )  [inline]
```

++ overload

Operator ++ as pre-increment.

3.2.3.4 operator++() [2/2]

```
template<typename pair_type , typename node_type >
Iterator Iterator< pair_type, node_type >::operator++ (
            int  )  [inline]
```

++ overload

Operator ++ as post-increment with `int` .

3.2.3.5 operator->()

```
template<typename pair_type , typename node_type >
pointer Iterator< pair_type, node_type >::operator-> ( ) const  [inline], [noexcept]
```

-> overload

Accessing first member of iterator through a pointer.

3.2.4 Friends And Related Function Documentation

3.2.4.1 operator"!=

```
template<typename pair_type , typename node_type >
bool operator!= (
            const Iterator< pair_type, node_type > & lhs,
            const Iterator< pair_type, node_type > & rhs )  [friend]
```

!= overload

Operator != overloading. `lhs` const Iterator l-value reference used as lhs of the equation while `rhs` const Iterator l-value reference used as rhs of the equation. Returning value obtained using the == operator and taking the negation of the result.

3.2.4.2 operator==

```
template<typename pair_type , typename node_type >
bool operator== (
            const Iterator< pair_type, node_type > & lhs,
            const Iterator< pair_type, node_type > & rhs )  [friend]
```

== overload

Operator == overloading. `lhs` const Iterator l-value reference used as lhs of the equation while `rhs` const Iterator l-value reference used as rhs of the equation.

3.2.5 Member Data Documentation

3.2.5.1 current

```
template<typename pair_type , typename node_type >
node_type* Iterator< pair_type, node_type >::current  [private]
```

pointer to current node

Pointer to a node class, private to the user,current.

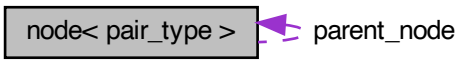
The documentation for this class was generated from the following file:

- include/iterator.hpp

3.3 node< pair_type > Class Template Reference

```
#include <node.hpp>
```

Collaboration diagram for node< pair_type >:



Public Member Functions

- node ()=default
  Default node Constructor.
- ~node () noexcept=default
  Default node Destructor.
- node (const pair_type &d, node *parent=nullptr)
  Custom node Constructor.
- node (const pair_type &&d, node *parent=nullptr)
  Custom Node Constructor.
- node (node *parent=nullptr)
  Custom Node Constructor.
- node (node &&other) noexcept=default
  move constuctor
- node & operator= (node &&other) noexcept=default
  move assignment

- node (const node &other)
  deep copy constructor
- node & operator= (const node &x)
  deep copy assignment
- node * get_left () noexcept
  get left child of a node
- node * get_right () noexcept
  get right child of a node
- node * get_parent () noexcept
  get parent of a node
- pair_type & get_data () noexcept
  get data contained in a node
- const pair_type & get_data () const noexcept
  get data contained in a node
- node * leftiest () noexcept
  get far left leaf node
- const node * leftiest () const noexcept
  get far left leaf node

Public Attributes

- node * parent_node
  parent node
- std::unique_ptr< node > left_child
  left child
- std::unique_ptr< node > right_child
  right child

Private Member Functions

- node * _leftmost (node *other) const noexcept
  pointer to far left node

Private Attributes

- pair_type data
  node content

3.3.1 Detailed Description

template<typename pair_type>
class node< pair_type >

Template class for members of the binary search tree concept. Every element of the binary search tree is a node. Each node stores a pair of a key and the associated value.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 node() [1/5]

```
template<typename pair_type >
node< pair_type >::node (
            const pair_type & d,
            node< pair_type > * parent = nullptr )  [inline]
```

Custom node Constructor.

Creates a node receiving an l-value reference to the content we want to store in it,as `d` , and parent node as `parent` with default value as nullptr.

3.3.2.2 node() [2/5]

```
template<typename pair_type >
node< pair_type >::node (
            const pair_type && d,
            node< pair_type > * parent = nullptr )  [inline]
```

Custom Node Constructor.

Creates a node receiving an r-value reference to the content we want to store in it,as `d` , and parent node as `parent` with default value as nullptr.

3.3.2.3 node() [3/5]

```
template<typename pair_type >
node< pair_type >::node (
            node< pair_type > * parent = nullptr )  [inline], [explicit]
```

Custom Node Constructor.

Creates a node receiving a parent node as `parent` with default value as nullptr.

3.3.2.4 node() [4/5]

```
template<typename pair_type >
node< pair_type >::node (
            node< pair_type > && other )  [explicit], [default], [noexcept]
```

move constuctor

Implementing move constructor using default compiler generated ones.

3.3.2.5 node() [5/5]

```
template<typename pair_type >
node< pair_type >::node (
            const node< pair_type > & other )  [inline], [explicit]
```

deep copy constructor

Managing the resources from the lowest level, to achieve a deep copy of the bst itself.

3.3.3 Member Function Documentation

3.3.3.1 _leftmost()

```
template<typename pair_type >
node* node< pair_type >::_leftmost (
            node< pair_type > * other ) const  [inline], [private], [noexcept]
```

pointer to far left node

Private function that returns the node on the far left side of the tree, used as the begin of the iterator.

3.3.3.2 get_data() [1/2]

```
template<typename pair_type >
const pair_type& node< pair_type >::get_data ( ) const  [inline], [noexcept]
```

get data contained in a node

Returns

by const reference, contents of node, as data

3.3.3.3 get_data() [2/2]

```
template<typename pair_type >
pair_type& node< pair_type >::get_data ( )  [inline], [noexcept]
```

get data contained in a node

Returns

by reference, contents of node, as data

3.3.3.4 get_left()

```
template<typename pair_type >
node* node< pair_type >::get_left ( )  [inline], [noexcept]
```

get left child of a node

Returns

pointer to left child

3.3.3.5 get_parent()

```
template<typename pair_type >
node* node< pair_type >::get_parent ( )  [inline], [noexcept]
```

get parent of a node

Returns

pointer to parent

3.3.3.6 get_right()

```
template<typename pair_type >
node* node< pair_type >::get_right ( )  [inline], [noexcept]
```

get right child of a node

Returns

pointer to right child

3.3.3.7 leftiest() [1/2]

```
template<typename pair_type >
const node* node< pair_type >::leftiest ( ) const  [inline], [noexcept]
```

get far left leaf node

Returns

const leftmost node

3.3.3.8 leftiest() [2/2]

```
template<typename pair_type >
node* node< pair_type >::leftiest ( )  [inline], [noexcept]
```

get far left leaf node

Returns

pointer to leftmost node

3.3.3.9 operator=()

```
template<typename pair_type >
node& node< pair_type >::operator= (
            node< pair_type > && other )  [default], [noexcept]
```

move assignment

Implementing move assignment using default compiler generated ones.

3.3.4 Member Data Documentation

3.3.4.1 data

```
template<typename pair_type >
pair_type node< pair_type >::data  [private]
```

node content

Pair type, containing a key and associated value, stored in vardata.

3.3.4.2 left_child

```
template<typename pair_type >
std::unique_ptr<node> node< pair_type >::left_child
```

left child

Unique pointer to the node that is left from the current, a node with a smaller key from the current node.

3.3.4.3 parent_node

```
template<typename pair_type >
node* node< pair_type >::parent_node
```

parent node

Pointer to the node that has the current node as one of its' childs. Widely used in the insert function in order to find the correct place for a new node.

3.3.4.4 right_child

```
template<typename pair_type >
std::unique_ptr<node> node< pair_type >::right_child
```

right child

Unique pointer to the node that is right from the current, a node with a bigger key from the current node.

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

- include/node.hpp