

**Binary Search Tree AP DSSC 20/21**  
Group Romina Doz, Davide Basso, Imola Fodor

Generated by Doxygen 1.9.1



<b>1 README</b>	<b>1</b>
<b>2 Class Index</b>	<b>3</b>
2.1 Class List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 <code>bst&lt; key_type, value_type, comparison &gt;</code> Class Template Reference	5
3.1.1 Detailed Description	7
3.1.2 Member Typedef Documentation	7
3.1.2.1 <code>const_iterator</code>	7
3.1.2.2 <code>iterator</code>	7
3.1.2.3 <code>node_type</code>	7
3.1.2.4 <code>pair_type</code>	8
3.1.3 Member Function Documentation	8
3.1.3.1 <code>_find()</code>	8
3.1.3.2 <code>_insert()</code>	8
3.1.3.3 <code>_print2D()</code>	8
3.1.3.4 <code>balance()</code>	8
3.1.3.5 <code>begin()</code> [1/2]	9
3.1.3.6 <code>begin()</code> [2/2]	9
3.1.3.7 <code>cbegin()</code>	9
3.1.3.8 <code>cend()</code>	9
3.1.3.9 <code>clear()</code>	9
3.1.3.10 <code>emplace()</code>	10
3.1.3.11 <code>end()</code> [1/2]	10
3.1.3.12 <code>end()</code> [2/2]	10
3.1.3.13 <code>erase()</code>	10
3.1.3.14 <code>find()</code> [1/2]	10
3.1.3.15 <code>find()</code> [2/2]	11
3.1.3.16 <code>insert()</code> [1/2]	11
3.1.3.17 <code>insert()</code> [2/2]	11
3.1.3.18 <code>insert_balanced_node()</code>	11
3.1.3.19 <code>operator[]()</code> [1/2]	11
3.1.3.20 <code>operator[]()</code> [2/2]	12
3.1.3.21 <code>print2D()</code>	12
3.1.3.22 <code>repopulate()</code>	12
3.1.3.23 <code>size()</code>	12
3.1.4 Friends And Related Function Documentation	12
3.1.4.1 <code>operator&lt;&lt;</code>	13
3.1.5 Member Data Documentation	13
3.1.5.1 <code>_size</code>	13
3.1.5.2 <code>comp</code>	13
3.1.5.3 <code>head</code>	13

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

## Chapter 1

# README

Advanced programming exam 2020/2021 Group Romina Doz, Davide Basso, Imola Fodor



## Chapter 2

# Class Index

### 2.1 Class List

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

<a href="#">bst&lt; key_type, value_type, comparison &gt;</a>	5
<a href="#">Iterator&lt; pair_type, node_type &gt;</a>	13
<a href="#">node&lt; pair_type &gt;</a>	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 iterator Constructor.*
- `~bst ()` noexcept=default  
*Default iterator 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)  
*const 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 function leftmost(), 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 function leftmost(), 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 function leftmost(), 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 same 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 same 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]
```

const 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 ahead

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 and 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.

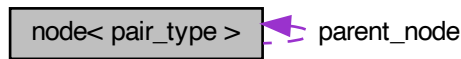
The documentation for this class was generated from the following file:

- 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 and 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 constructor*
- `node & operator= (node &&other) noexcept=default`  
*move semantics*
- `node(const node &other)`  
*deep copy semantics*
- `node & operator= (const node &x)`  
*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 constructor

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 semantics

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 semantics

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