BST_Doxygen

Generated by Doxygen 1.8.16

Chapter 1

Binary Search Tree

Author

Thorben Fröhlking

Date

30 August 2019

1.1 Abstract

A Binary Search Tree (BST) class was developed. A BST is a node-based ordered data structure. Left subtrees of a node contain only nodes with keys lesser and right subtrees only contain nodes with keys greater than the node's key. The left and right subtree are also BSTs. The framework offers storing pairs of key and value, iterating from smallest to largest key through nodes, balancing the tree such that lookup complexity reduces to approximately Olog(N), finding the value of a given key, copy and move semantic as well as printing the tree traversed in ascending order.

1.1.1 Files

The documented files are the BST class 'BST.h' containing the entire functionality, the 'Test.cpp' being the main method for demonstration purposes and the 'Performance.cpp' is the method needed for the performance run regarding average lookup times.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Class Documentation

5.1 BST< key, value, comparator > Class Template Reference

`#include <BST.h>`

Classes

- class ConstIterator

  ConstIterator is using the functionality of the Iterator class, but is preventing the accessed elements to be manipulated by returning operator∗() const.
- class Iterator

  Iterator uses struct node of BST() class.

Public Member Functions

- BST ()

  Each BST class is constructed with a smart pointer to the root node and a comparison functor.
- Iterator begin ()

  Iterator to the node with lowest key.
- Iterator end ()

  Iterator Returns nullptr.
- ConstIterator cbegin () const

  ConstIterator to the node with lowest key.
- ConstIterator cend () const

  ConstIterator Returns nullptr.
- void insert (const key k, value v)

  Making use of the comparison functor allows adding a pair of key and value as soon as a nullptr is encountered in the next smart pointer of the current node.
- void clear ()

  Resets the root-node back to nullptr.
- void balance ()

  Not in-place balancing of the tree in a recursive manner.
- ConstIterator find (const key k) const

  Searches for a given key.

- value & operator[ ] (const key &k)

  Operator allowing value access via key.
- const value & operator[ ] (const key &k) const

  Const operator allowing value access via key.
- BST (const BST &bst_rhs)

  Copy constructor which constructs a deepcopy with new root-node following the original node structure in a recursive manner and inserting new nodes with the same std::pair as the original.
- BST & operator= (const BST &bst_rhs)

  Copy assignment.
- BST (BST &&bst_rhs)

  Move constructor which allows that the ownership of the root-node is transferred to the lvalue.
- BST & operator= (BST &&bst_rhs)

  Move assignment.
- ~BST ()=default

  Default destructor.

5.1.1 Detailed Description

template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
class BST< key, value, comparator >

The BST class is templated on the type of the const key and the type of the value associated with it allowing for variability in input data and the Functor which template type is derived via decltype(). Nodes are realised as nested struct containing data, smart pointers to the next left and right node as well as a pointer to the local root.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 BST() [1/3]

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
BST< key, value, comparator >::BST ( )  [inline]
```

Each BST class is constructed with a smart pointer to the root node and a comparison functor.

5.1.2.2 BST() [2/3]

```
template<class key , class value , class comparator >
BST< key, value, comparator >::BST (
            const BST< key, value, comparator > & bst_rhs )
```

Copy constructor which constructs a deepcopy with new root-node following the original node structure in a recursive manner and inserting new nodes with the same std::pair as the original.

Parameters

in	*bst_rhs*	BST object to copy from.

a root-node and a functor are initialised. The private member function deepcopy_recursive(), which is taking a reference to the current node of the original tree, is used to build a new tree with the same components as the original by using insert(). First the entire left tree side is copied recursively until a nullptr is encountered in rhs tree and then the same is repeated recursively for the right half of the tree.

5.1.2.3 BST() [3/3]

```
template<class key , class value , class comparator >
BST< key, value, comparator >::BST (
            BST< key, value, comparator > && bst_rhs )
```

Move constructor which allows that the ownership of the root-node is transferred to the lvalue.

Parameters

in	*bst_rhs*	BST object to acquire ownership from.

Takes rvalue reference && and uses std::move() to obtain ownership over the rhs root-node.

5.1.2.4 ~BST()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
BST< key, value, comparator >::~BST ( )  [default]
```

Default destructor.

5.1.3 Member Function Documentation

5.1.3.1 balance()

```
template<class key , class value , class comparator >
void BST< key, value, comparator >::balance ( )
```

Not in-place balancing of the tree in a recursive manner.

Upon calling the function stores all std:pairs contained in the tree inside a std::vector in ascending order according to key. The tree is set to empty via clear() and rebuild in a recursive manner using the private member function balance_recursive(std::vector<std::pair<const key, value> >& vec, std::size_t start, std::size_t end). This function recursively determines a median and inserts its data-pair back into the tree via insert(). Initially the entire left side of the tree is rebuild with the call to balance_recursive(). As soon as this half of the vector is down to a single data-pair the same procedure is carried out for the right side of the vector.

5.1.3.2 begin()

```
template<class key , class value , class comparator >
BST< key, value, comparator >::Iterator BST< key, value, comparator >::begin ( )
```

Iterator to the node with lowest key.

5.1.3.3 cbegin()

```
template<class key , class value , class comparator >
BST< key, value, comparator >::ConstIterator BST< key, value, comparator >::cbegin ( ) const
```

ConstIterator to the node with lowest key.

5.1.3.4 cend()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
ConstIterator BST< key, value, comparator >::cend ( ) const  [inline]
```

ConstIterator Returns nullptr.

5.1.3.5 clear()

```
template<class key , class value , class comparator >
void BST< key, value, comparator >::clear ( )
```

Resets the root-node back to nullptr.

5.1.3.6 end()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
Iterator BST< key, value, comparator >::end ( )  [inline]
```

Iterator Returns nullptr.

5.1.3.7 find()

```
template<class key , class value , class comparator >
BST< key, value, comparator >::ConstIterator BST< key, value, comparator >::find (
            const key k ) const
```

Searches for a given key.

Parameters

in	*k*	target key.
out	*ConstIterator*	to the corresponding node.

The search starts with a temporary raw pointer to the root-node and continues until either the target k is found and returned as ConstIterator to the corresponding nodes or a nullptr is met and returned.

5.1.3.8 insert()

```
template<class key , class value , class comparator >
void BST< key, value, comparator >::insert (
            const key k,
            value v )
```

Making use of the comparison functor allows adding a pair of key and value as soon as a nullptr is encountered in the next smart pointer of the current node.

Parameters

in	*k,v*	arbitrary input key and value.

When the tree is empty a first node is constructed and the smart pointer to root-node is moved. Adding additional nodes is allowed via a the private member function add_node_recursive(std::pair<const key, value> p, node* current). It uses the functor for comparison and moves along the branches accordingly until a nullptr is encountered. If the key was not existing a new node is contructed and previous left or right smart pointer is moved to the new node. if the key was existing the value is overwritten.

5.1.3.9 operator=() [1/2]

```
template<class key , class value , class comparator >
BST< key, value, comparator > & BST< key, value, comparator >::operator= (
            BST< key, value, comparator > && bst_rhs )
```

Move assignment.

Parameters

in	*=bst_rhs*	BST object to obtain ownership from.
out	*lhs&*	Reference to lhs BST object.

Takes rvalue reference && and uses std::move() to obtain ownership over the rhs root-node.

5.1.3.10 operator=() [2/2]

```
template<class key , class value , class comparator >
BST< key, value, comparator > & BST< key, value, comparator >::operator= (
            const BST< key, value, comparator > & bst_rhs )
```

Copy assignment.

Parameters

in	*=bst_rhs*	BST object to copy from.
out	*lhs&*	Reference to lhs BST object which is a deepcopy.

First the copy assignement clears the lhs and proceeds like the copy constructor afterwards. It also contains a handling for self assignment.

5.1.3.11 operator[]() [1/2]

```
template<class key , class value , class comparator >
value & BST< key, value, comparator >::operator[] (
             const key & k )
```

Operator allowing value access via key.

Parameters

in	*k*	target key.
out	*value&*	reference to the corresponding value.

find() is used to search and return the key. If the key can not be found a new key is inserted with value{} and returned.

5.1.3.12 operator[]() [2/2]

```
template<class key , class value , class comparator >
const value & BST< key, value, comparator >::operator[] (
             const key & k ) const
```

Const operator allowing value access via key.

Parameters

in	*k*	target key.
out	*value&*	constant reference to the corresponding value.

find() is also used for the const override. However if the key can not be found a std::runtime error is thrown.

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

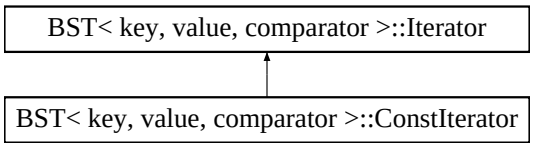
- D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/↩ Exam/C++/Class/Doxygen/BST.h

5.2 BST< key, value, comparator >::ConstIterator Class Reference

ConstIterator is using the functionality of the Iterator class, but is preventing the accessed elements to be manipulated by returning operator*() const.

`#include <BST.h>`

Inheritance diagram for BST< key, value, comparator >::ConstIterator:



Public Types

- using parent = const BST< key, value, comparator >::Iterator

Public Member Functions

- std::pair< const key, value > operator* () const
- Iterator (node *n)

  Iterator is constructed with a raw pointer to the current node.

5.2.1 Detailed Description

template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
class BST< key, value, comparator >::ConstIterator

ConstIterator is using the functionality of the Iterator class, but is preventing the accessed elements to be manipulated by returning operator*() const.

5.2.2 Member Typedef Documentation

5.2.2.1 parent

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
using BST< key, value, comparator >::ConstIterator::parent = const BST<key, value, comparator>↩
::Iterator
```

5.2.3 Member Function Documentation

5.2.3.1 Iterator()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
BST< key, value, comparator >::Iterator::Iterator  [inline]
```

Iterator is constructed with a raw pointer to the current node.

5.2.3.2 operator∗()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
std::pair<const key, value> BST< key, value, comparator >::ConstIterator::operator* ( ) const
[inline]
```

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

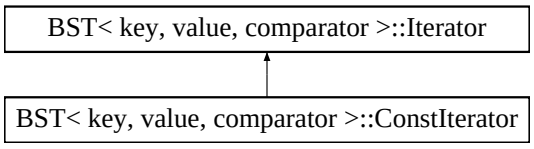
- D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/↩ Exam/C++/Class/Doxygen/BST.h

5.3 BST< key, value, comparator >::Iterator Class Reference

Iterator uses struct node of BST() class.

```
#include <BST.h>
```

Inheritance diagram for BST< key, value, comparator >::Iterator:



Public Member Functions

- Iterator (node ∗n)

  Iterator is constructed with a raw pointer to the current node.
- std::pair< const key, value > & operator∗ () const

  operator∗() is overloaded to return reference to the data-pair.
- Iterator & operator++ ()

  Pre-increment operator++() is overloaded to move to the next right node and if that is not possible to keep moving up in the tree until the next largest key is encountered.
- Iterator operator++ (int)

  Post-increment operator++(int) is enabled.
- bool operator== (const Iterator &other)

  Operator for equality check between two iterators.
- bool operator!= (const Iterator &other)

  Operator for equality check between two iterators.

5.3.1 Detailed Description

template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
class BST< key, value, comparator >::Iterator

Iterator uses struct node of BST() class.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Iterator()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
BST< key, value, comparator >::Iterator::Iterator (
            node * n )  [inline]
```

Iterator is constructed with a raw pointer to the current node.

5.3.3 Member Function Documentation

5.3.3.1 operator"!=()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
bool BST< key, value, comparator >::Iterator::operator!= (
            const Iterator & other )  [inline]
```

Operator for equality check between two iterators.

Parameters

out	*bool*	true if not equal.

5.3.3.2 operator*()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
std::pair<const key, value>& BST< key, value, comparator >::Iterator::operator* ( ) const
[inline]
```

operator*() is overloaded to return reference to the data-pair.

5.3.3.3 operator++() [1/2]

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
Iterator& BST< key, value, comparator >::Iterator::operator++ ( )  [inline]
```

Pre-increment operator++() is overloaded to move to the next right node and if that is not possible to keep moving up in the tree until the next largest key is encountered.

Parameters

out	*this	Iterator reference of current node.

5.3.3.4 operator++() [2/2]

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
Iterator BST< key, value, comparator >::Iterator::operator++ (
            int  )  [inline]
```

Post-increment operator++(int) is enabled.

5.3.3.5 operator==()

```
template<class key, class value, class comparator = decltype(& Functor<const key,value>)>
bool BST< key, value, comparator >::Iterator::operator== (
            const Iterator & other )  [inline]
```

Operator for equality check between two iterators.

Parameters

out	*bool*	true if equal.

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

- D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/↩ Exam/C++/Class/Doxygen/BST.h

Chapter 6

File Documentation

6.1 D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/Exam/C++/Class/Doxygen/BST.h File Reference

```
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>
```

Classes

- class BST< key, value, comparator >
- class BST< key, value, comparator >::Iterator

  Iterator uses struct node of BST() class.

- class BST< key, value, comparator >::ConstIterator

  ConstIterator is using the functionality of the Iterator class, but is preventing the accessed elements to be manipulated by returning operator() const.*

Functions

- template<class key , class value >
  int Functor (std::pair< const key, value > &lhs, std::pair< const key, value > &rhs)
- template<class key , class value , class comparator >
  std::ostream & operator<< (std::ostream &os, BST< key, value, comparator > &l)

  Override to allow printing of the BST in the format 'key : value'.
- template<class key , class value , class comparator >
  std::ostream & operator<< (std::ostream &os, const BST< key, value, comparator > &l)

  Override to allow printing of a const BST in the format 'key : value' via a ConstIterator handling.

6.1.1 Detailed Description

The file contains the entire functionality of the BST. For demonstration purposes additional output stream statements can be enabled.

6.1.2 Function Documentation

6.1.2.1 Functor()

```
template<class key , class value >
int Functor (
            std::pair< const key, value > & lhs,
            std::pair< const key, value > & rhs )
```

A templated functor is setup for comparison taking references to two std::pairs

Parameters

in	*lhs,rhs*	Reference to the two data pairs that shall be compared.
out	*int*	0 if larger, 1 if smaller and 2 if equal.

6.1.2.2 operator<<() [1/2]

```
template<class key , class value , class comparator >
std::ostream& operator<< (
            std::ostream & os,
            BST< key, value, comparator > & l )
```

Override to allow printing of the BST in the format 'key : value'.

6.1.2.3 operator<<() [2/2]

```
template<class key , class value , class comparator >
std::ostream& operator<< (
            std::ostream & os,
            const BST< key, value, comparator > & l )
```

Override to allow printing of a const BST in the format 'key : value' via a ConstIterator handling.

6.2 D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/Exam/C++/Class/Doxygen/Performance.cpp File Reference

```
#include "BST_performance.h"
#include <map>
#include <algorithm>
#include <chrono>
#include <fstream>
```

Functions

- int main ()

6.2.1 Detailed Description

The file contains a procedure to investigate the performance of the lookup in unbalanced and balanced BST() via member function find() compared to std::map. Average lookup-times are measured using std::chrono::high↩ _resolution_clock and normalised with respect to the number of nodes present. The performance is tested for node numbers in the range of 10000 to 10000000. The output is a AverageLookupTimes.txt which is containing lookup-times in nanoseconds in the format 'std::map : BST(unbalanced) : BST(balanced)'.

6.2.2 Function Documentation

6.2.2.1 main()

```
int main ( )
```

Vector to store the lookup-times is constructed. And the node range is defined.

Vector for keys is constructed and contains keys in the node range.

Vector for input keys is constructed and contains the keys, but in a random order.

Vector for lookup keys is constructed and contains the keys, but in randomised lookup order.

A std::map is populated with the input keys.

The time to lookup all keys in the randomised lookup order is measured in nanoseconds.

Average lookup-times are stored.

A BST() is populated with the input keys.

The time to lookup all keys in the randomised lookup order is measured in nanoseconds.

Average lookup-times are stored.

The unbalanced BST() is copied and balanced afterwards.

The time to lookup all keys in the randomised lookup order is measured in nanoseconds.

Average lookup-times are stored.

The output-file is generated.

6.3 D:/PhD course SISSA-Physics and Chemistry of Biological Systems/Modules/Advanced Programming/Exam/C++/Class/Doxygen/Test.cpp File Reference

```
#include "BST.h"
```

Functions

- int main ()

6.3.1 Detailed Description

The file contains a demonstration of the functionality inside the Binary Search Tree class. In chronological order a BST class is constructed, data-pairs are inserted and the tree is cleared afterwards. Again data-pairs get inserted and Iterators begin()and end() are used to print all tree nodes. The ConstIterator is investigated regarding pre and post-increment operators. Commented lines can be enabled to see the read-only behaviour. The tree is cleared again and repopulated via the operator[]. The operator<< is used to print the tree. A const BST instance is constructed and its operator[] and operator<< are tested. The overwrite on insert() is checked. The functionality of find() is investigated and its returned iterator is checked. balance() is called for the non-const BST and it is printed multiple times to additionally test all copy and move semantics provided by the class. Commented lines can be enabled to see the causing of std::runtime_error.

6.3.2 Function Documentation

6.3.2.1 main()

```
int main ( )
```

testing constructor: BST()

testing function: void insert(const key k, value v)

testing function: void clear()

testing iterator functions: begin(), end(), cbegin(), cend()

testing value& operator[ ](const keye& k) non-const and const

testing operator <<

testing function: overwrite in void insert(const key k, value v)

testing function: ConstIterator find(const key k)

testing function: void balance()

testing copy constructor

testing copy assignment

testing move constructor

testing move assignment