proj2

Generated by Doxygen 1.12.0

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 BSTTree< T > Class Template Reference

Binary search tree implementation, accepting elements of type T.

`#include <BSTTree.hpp>`

Public Member Functions

- BSTTree (const T &element)
  Constructs a BSTTree with an initial element.
- BSTTree ()
  Constructs an empty BSTTree.
- ~BSTTree ()
  Destructor to clean up dynamically allocated nodes in the tree.
- void add (const T &element)
  Adds an element to the tree.
- void delete_tree ()
  Deletes all nodes in the tree, resetting it to an empty state.
- std::vector< T > traverse_preorder () const
  Traverse the tree in the preorder direction and return a vector the contents of each node.
- std::vector< T > traverse_inorder () const
  Traverse the tree in the inorder direction and return a vector the contents of each node.
- std::vector< T > traverse_postorder () const
  Traverse the tree in the postorder direction and return a vector the contents of each node.
- int delete_element (T value)
  Delete an element of a given value.
- std::vector< T > find_path (const T &value)
  Finds the path to a value in a binary search tree.

3.1.1 Detailed Description

template<typename T>
class BSTTree< T >

Binary search tree implementation, accepting elements of type T.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BSTTree() [1/2]

```
template<typename T >
BSTTree< T >::BSTTree (
            const T & element)  [inline]
```

Constructs a BSTTree with an initial element.

Parameters

element	The initial element to be stored in the root node.

3.1.2.2 BSTTree() [2/2]

```
template<typename T >
BSTTree< T >::BSTTree ()  [inline]
```

Constructs an empty BSTTree.

3.1.2.3 ~BSTTree()

```
template<typename T >
BSTTree< T >::~BSTTree ()  [inline]
```

Destructor to clean up dynamically allocated nodes in the tree.

3.1.3 Member Function Documentation

3.1.3.1 add()

```
template<typename T >
void BSTTree< T >::add (
            const T & element)  [inline]
```

Adds an element to the tree.

Parameters

element	The element to insert into the tree.

This operation is performed recursively.

3.1.3.2 delete_element()

```
template<typename T >
int BSTTree< T >::delete_element (
            T value)  [inline]
```

Delete an element of a given value.

Parameters

value	The value of the element to be deleted.

This does not differentiate between unique elements of the same value.

Returns

0 if the element was successfully deleted, -1 if the element was not found.

3.1.3.3 delete_tree()

```
template<typename T >
void BSTTree< T >::delete_tree ()  [inline]
```

Deletes all nodes in the tree, resetting it to an empty state.

3.1.3.4 find_path()

```
template<typename T >
std::vector< T > BSTTree< T >::find_path (
            const T & value)  [inline]
```

Finds the path to a value in a binary search tree.

Template Parameters

T	The type of the tree values.

Parameters

value	The value to find.

Returns

std::vector<T> Path to the value or empty if not found.

3.1.3.5 traverse_inorder()

```
template<typename T >
std::vector< T > BSTTree< T >::traverse_inorder () const  [inline]
```

Traverse the tree in the inorder direction and return a vector the contents of each node.

Returns

Inordered vector of the elements of the tree.

3.1.3.6 traverse_postorder()

```
template<typename T >
std::vector< T > BSTTree< T >::traverse_postorder () const  [inline]
```

Traverse the tree in the postorder direction and return a vector the contents of each node.

Returns

Postordered vector of the elements of the tree.

3.1.3.7 traverse_preorder()

```
template<typename T >
std::vector< T > BSTTree< T >::traverse_preorder () const  [inline]
```

Traverse the tree in the preorder direction and return a vector the contents of each node.

Returns

Preordered vector of the elements of the tree.

3.2 FileTree< T > Class Template Reference

```
#include <FileTree.hpp>
```

Public Member Functions

- void save_to_text (const std::string &filename, const std::vector< T > &elements)

  Save the tree's elements to a text file.
- void load_from_text (const std::string &filename, BSTTree< T > &tree, std::vector< T > &elements, bool append=false)

  Load elements from a text file into a tree.
- void save_to_binary (const std::string &filename, const std::vector< T > &elements)

  Save the tree's elements to a binary file.
- void load_from_binary (const std::string &filename, BSTTree< T > &tree, std::vector< T > &elements, bool append=false)

  Load elements from a binary file into a tree.

3.2.1 Member Function Documentation

3.2.1.1 load_from_binary()

```
template<typename T >
void FileTree< T >::load_from_binary (
            const std::string & filename,
            BSTTree< T > & tree,
            std::vector< T > & elements,
            bool append = false)  [inline]
```

Load elements from a binary file into a tree.

Parameters

filename	The name of the binary file to read.
tree	The tree to populate.
elements	A list to store the loaded elements.
append	Whether to add to an existing tree or start fresh. Without this, tree would have to deleted manually.

3.2.1.2 load_from_text()

```
template<typename T >
void FileTree< T >::load_from_text (
            const std::string & filename,
            BSTTree< T > & tree,
            std::vector< T > & elements,
            bool append = false)  [inline]
```

Load elements from a text file into a tree.

Parameters

filename	The name of the file to read from.
tree	The tree to populate.
elements	A list to store the loaded elements.
append	Whether to add to an existing tree or start fresh.

3.2.1.3 save_to_binary()

```
template<typename T >
void FileTree< T >::save_to_binary (
            const std::string & filename,
            const std::vector< T > & elements)  [inline]
```

Save the tree's elements to a binary file.

Parameters

filename	The name of the binary file.
elements	A list of tree elements.

3.2.1.4 save_to_text()

```
template<typename T >
void FileTree< T >::save_to_text (
            const std::string & filename,
            const std::vector< T > & elements)  [inline]
```

Save the tree's elements to a text file.

Parameters

filename	The name of the file to save to.
elements	A list of tree elements.

Chapter 4

File Documentation

4.1 src/BSTTree.hpp File Reference

```
#include <ranges>
#include <vector>
#include <algorithm>
```

Classes

- class BSTTree< T >

  Binary search tree implementation, accepting elements of type T.

4.2 BSTTree.hpp

Go to the documentation of this file.

```
00001
00002 #pragma once
00003
00004 #include <ranges>
00005 #include <vector>
00006 #include <algorithm>
00007
00011 template <typename T>
00012 class BSTTree {
00013 private:
00014     struct Tree {
00015         T contents;
00016         Tree* parent;
00017         Tree* left;
00018         Tree* right;
00019
00020         Tree(T _contents, Tree* _parent = nullptr, Tree* _left = nullptr, Tree* _right = nullptr) :
00021             contents{ _contents }, parent{ _parent }, left{ _left }, right{ _right } {
00022         }
00023
00024         ~Tree() {
00025             delete left;
00026             delete right;
00027         }
00028     };
00029
00030     Tree* root;
00031
00032     void recursive_add(const T& element, Tree*& node, Tree* parentNode = nullptr) {
00033         if (node == nullptr) {
00034             node = new Tree(element);
```

```
00035             if (parentNode == nullptr) {
00036                 return;
00037             }
00038             node->parent = parentNode;
00039             if (node->contents >= parentNode->contents) {
00040                 parentNode->right = node;
00041             }
00042             else {
00043                 parentNode->left = node;
00044             }
00045             return;
00046         }
00047         if (element >= node->contents) {
00048             recursive_add(element, node->right, node);
00049         }
00050         else {
00051             recursive_add(element, node->left, node);
00052         }
00053     }
00054
00055     void preorder_traverse_recursive(Tree* node, std::vector<Tree*>& traversedTrees) const {
00056         if (node == nullptr) {
00057             return;
00058         }
00059         traversedTrees.push_back(node);
00060         preorder_traverse_recursive(node->left, traversedTrees);
00061         preorder_traverse_recursive(node->right, traversedTrees);
00062     }
00063
00064     void inorder_traverse_recursive(Tree* node, std::vector<Tree*>& traversedTrees) const {
00065         if (node == nullptr) {
00066             return;
00067         }
00068         inorder_traverse_recursive(node->left, traversedTrees);
00069         traversedTrees.push_back(node);
00070         inorder_traverse_recursive(node->right, traversedTrees);
00071     }
00072
00073     void postorder_traverse_recursive(Tree* node, std::vector<Tree*>& traversedTrees) const {
00074         if (node == nullptr) {
00075             return;
00076         }
00077         postorder_traverse_recursive(node->left, traversedTrees);
00078         postorder_traverse_recursive(node->right, traversedTrees);
00079         traversedTrees.push_back(node);
00080     }
00081
00082 public:
00087     BSTTree(const T& element) : root{ new Tree(element) } {}
00088
00092     BSTTree() : root{ nullptr } {}
00093
00097     ~BSTTree() { delete root; }
00098
00105     void add(const T& element) {
00106         recursive_add(element, root);
00107     }
00108
00112     void delete_tree() {
00113         delete root;
00114         root = nullptr;
00115     }
00116
00121     std::vector<T> traverse_preorder() const {
00122         std::vector<Tree*> traversedTrees;
00123         preorder_traverse_recursive(root, traversedTrees);
00124         return traversedTrees | std::ranges::views::transform([](const Tree* tree) { return
      tree->contents; }) | std::ranges::to<std::vector>();
00125     }
00126
00131     std::vector<T> traverse_inorder() const {
00132         std::vector<Tree*> traversedTrees;
00133         inorder_traverse_recursive(root, traversedTrees);
00134         return traversedTrees | std::ranges::views::transform([](const Tree* tree) { return
      tree->contents; }) | std::ranges::to<std::vector>();
00135     }
00136
00141     std::vector<T> traverse_postorder() const {
00142         std::vector<Tree*> traversedTrees;
00143         postorder_traverse_recursive(root, traversedTrees);
00144         return traversedTrees | std::ranges::views::transform([](const Tree* tree) { return
      tree->contents; }) | std::ranges::to<std::vector>();
00145     }
00146
00154     int delete_element(T value) {
00155         std::vector<Tree*> traversedTrees;
00156         inorder_traverse_recursive(root, traversedTrees);
```

```
00157
00158         auto elementOfValueIterator = std::find_if(traversedTrees.begin(), traversedTrees.end(),
00159                                                    [&value](const Tree* tree) { return tree->contents ==
      value; });
00160
00161         if (elementOfValueIterator == traversedTrees.end()) {
00162             return -1;
00163         }
00164
00165         Tree* elementOfValue{ *elementOfValueIterator };
00166         if (elementOfValue->left == nullptr && elementOfValue->right == nullptr) {
00167             if (elementOfValue->parent != nullptr) {
00168                 if(elementOfValue->parent->left == elementOfValue) {
00169                     elementOfValue->parent->left = nullptr;
00170                 } else {
00171                     elementOfValue->parent->right = nullptr;
00172                 }
00173             }
00174
00175             delete elementOfValue;
00176         } else {
00177             Tree* successor = *std::next(elementOfValueIterator);
00178             if (successor->parent != nullptr) {
00179                 if (successor->parent->left == successor) {
00180                     successor->parent->left = nullptr;
00181                 }
00182                 else {
00183                     successor->parent->right = nullptr;
00184                 }
00185             }
00186             elementOfValue->contents = successor->contents;
00187             delete successor;
00188         }
00189
00190         return 0;
00191     }
00192
00193
00201     std::vector<T> find_path(const T& value) {
00202         std::vector<T> path;
00203         Tree* curr = root;
00204
00205         while (root != nullptr) {
00206             path.push_back(curr->contents);
00207
00208             if (curr->contents == value) {
00209                 return path;
00210             }
00211             else if (value < curr->contents) {
00212                 curr = curr->left;
00213             }
00214             else {
00215                 curr = curr->right;
00216             }
00217         }
00218         return {};
00219     }
00220
00221 };
```

4.3 src/FileTree.hpp File Reference

A simple class to save and load a Binary Search Tree (BSTTree) from files.

```
#include "BSTTree.hpp"
#include <fstream>
#include <vector>
#include <iostream>
```

Classes

- class FileTree< T >

4.3.1 Detailed Description

A simple class to save and load a Binary Search Tree (BSTTree) from files.

4.4 FileTree.hpp

Go to the documentation of this file.

```
00001
00006 #pragma once
00007
00008 #include "BSTTree.hpp"
00009 #include <fstream>
00010 #include <vector>
00011 #include <iostream>
00012
00013 template <typename T>
00014 class FileTree {
00015 public:
00021     void save_to_text(const std::string& filename, const std::vector<T>& elements) {
00022         std::ofstream file(filename);
00023         if (!file) {
00024             std::cerr « "Error: Could not open file for writing.\n";
00025             return;
00026         }
00027         for (const auto& elem : elements) {
00028             file « elem « " ";
00029         }
00030     }
00031
00039     void load_from_text(const std::string& filename, BSTTree<T>& tree, std::vector<T>& elements, bool
      append = false) {
00040         std::ifstream file(filename);
00041         if (!file) {
00042             std::cerr « "Error: Could not open file for reading.\n";
00043             return;
00044         }
00045         if (!append) {
00046             tree.delete_tree();
00047             elements.clear();
00048         }
00049         T value;
00050         while (file » value) {
00051             elements.push_back(value);
00052             tree.add(value);
00053         }
00054     }
00055
00061     void save_to_binary(const std::string& filename, const std::vector<T>& elements) {
00062         std::ofstream file(filename, std::ios::binary);
00063         if (!file) {
00064             std::cerr « "Error: Could not open file for binary writing.\n";
00065             return;
00066         }
00067         size_t size = elements.size();
00068         file.write(reinterpret_cast<const char*>(&size), sizeof(size));
00069         for (const auto& elem : elements) {
00070             file.write(reinterpret_cast<const char*>(&elem), sizeof(T));
00071         }
00072     }
00073
00081     void load_from_binary(const std::string& filename, BSTTree<T>& tree, std::vector<T>& elements,
      bool append = false) {
00082         std::ifstream file(filename, std::ios::binary);
00083         if (!file) {
00084             std::cerr « "Error: Could not open file for binary reading.\n";
00085             return;
00086         }
00087         if (!append) {
00088             tree.delete_tree();
00089             elements.clear();
00090         }
00091         size_t size = 0;
00092         file.read(reinterpret_cast<char*>(&size), sizeof(size));
00093         for (size_t i = 0; i < size; ++i) {
00094             T elem;
00095             file.read(reinterpret_cast<char*>(&elem), sizeof(T));
00096             elements.push_back(elem);
00097             tree.add(elem);
00098         }
00099     }
00100 };
```

4.5 src/main.cpp File Reference

```
#include "BSTTree.hpp"
#include "FileTree.hpp"
#include <print>
#include <iostream>
#include <string>
```

Functions

- bool lauf ()
- int main (int argc, char ∗argv[ ])

4.5.1 Function Documentation

4.5.1.1 lauf()

```
bool lauf ()
```

4.5.1.2 main()

```
int main (
            int argc,
            char * argv[ ])
```

Index