Текстовод Уникальность https://textovod.com/unique

Отчет проверки уникальности текста

Дата проверки: 2023-06-22 22:38:44

Уникальность 77%

Хорошо. Подойдет для большинства текстов.

Текст

#pragma once

#include < iostream>
#include < string>

#include "colors.h"

template < typename T> class BST {
public:
struct Node {
T value;
// Parent only used to beautifully display nodes, it is not used in anything else
Node *left, *right, *parent;

Node(T value, Node *parent): value(value), left(nullptr), right(nullptr), parent(parent) {
}
};

private:
BST< T> :: Node *tree = nullptr;

// Colored "left" and "right" words
const std: :string left_string = COLOR_YELLOW + "left" + COLOR_RESET;
const std: :string right_string = COLOR_CYAN + "right" + COLOR_RESET;
const std: :string moving_left_string = " - moving " + this-> left_string;
const std: :string moving_right_string = " - moving " + this-> right_string;

```
// Colored "value" output
std: :string valueString(T value) {
return "'" + COLOR_YELLOW + std: :to_string(value) + COLOR_RESET + "'";
}

// Shows colored node. Green - parent, Yellow - left of parent, Cyan - right of parent
void showNode(BST< T> :: Node *node) {
if (! node-> parent) {
std: :cout < < COLOR_GREEN;
} else if (node-> parent-> left == node) {
std: :cout < < COLOR_YELLOW;
} else {
std: :cout < < COLOR_CYAN;
}

std: :cout < < node-> value < < COLOR_RESET < < " ";
}

void showPreOrder(BST< T> :: Node *node) {
if (! node) {
return;
}

showNode(node);
showPreOrder(node-> left);
showPreOrder(node-> right);
}

void showInOrder(BST< T> :: Node *node) {
if (! node) {
return;
}

showInOrder(node-> left);
showNode(node);
showInOrder(node-> right);
}

void showPostOrder(BST< T> :: Node *node) {
if (! node) {
return;
```

```
}

showPostOrder(node-> left);
showPostOrder(node-> right);
showNode(node);
}

public:
void insert(T value) {
if (! this-> tree) {
std: :cout < < COLOR_GREEN < < "Tree is empty, creating the root" < <
COLOR_RESET < < std: :endl;

this-> tree = new BST< T> :: Node(value, nullptr);
return;
}

std: :cout < < "Searching for the suitable space: " < < std: :endl;
BST< T> :: Node *prev = nullptr;
BST< T> :: Node *current = tree;

while (current) {
prev = current;

if (current-> value > value) {
std: :cout < < moving_left_string < < std: :endl;
current = current-> left;
} else if (current-> value < value) {
std: :cout < < moving_right_string < < std: :endl;
current = current-> right;
} else {
std: :cout < < COLOR_RED < < "The same value is found. Returning" < <
COLOR_RESET < < std: :endl;

return;
}
}

if (prev-> value > value) {
std: :cout < < "Inserting new node " < < this-> left_string < < " with the " < <
valueString(value) < < std: :endl;
```

```
prev-> left = new BST< T> :: Node(value, prev);
} else if (prev-> value < value) {
std: :cout < < "Inserting new node " < < this-> right_string < < " with the " < <
valueString(value) < < std: :endl;
prev-> right = new BST< T> :: Node(value, prev);
}
}

BST< T> :: Node *search(T value) {
if (! this-> tree) {
std: :cout < < "The tree is empty, nothing to search" < < std: :endl;
return nullptr;
}

BST< T> :: Node *current = this-> tree;
while (current) {
if (current-> value > value) {
std: :cout < < moving_left_string < < std: :endl;
current = current-> left;
} else if (current-> value < value) {
std: :cout < < moving_right_string < < std: :endl;
current = current-> right;
} else {
std: :cout < < "The value " < < this-> valueString(value) < < COLOR_GREEN < < "
was found" < < COLOR_RESET
< < std: :endl;

return current;
}
}

std: :cout < < "The value " < < this-> valueString(value) < < COLOR_RED < < " was
not found" < < COLOR_RESET
< < std: :endl;

return nullptr;
}

void show() {
if (! this-> tree) {
std: :cout < < "The tree is empty, nothing to show" < < std: :endl;
```

```
return;
}

std: :cout < < "Pre-order traversal: ";
this-> showPreOrder(this-> tree);
std: :cout < < std: :endl;

std: :cout < < "In-order traversal: ";
this-> showInOrder(this-> tree);
std: :cout < < std: :endl;

std: :cout < < "Post-order traversal: ";
this-> showPostOrder(this-> tree);
std: :cout < < std: :endl;
}

BST< T> :: Node *remove(T value) {
if (! this-> tree) {
std: :cout < < "The tree is empty, nothing to remove" < < std: :endl;
return this-> tree;
}

std: :cout < < "Searching for the value " < < this-> valueString(value) < < std: :endl;

BST< T> :: Node *current = this-> tree;
BST< T> :: Node *previous = nullptr;

while (current) {
if (current-> value > value) {
previous = current;
std: :cout < < moving_left_string < < std: :endl;
current = current-> left;
} else if (current-> value < value) {
previous = current;
std: :cout < < moving_right_string < < std: :endl;
current = current-> right;
} else {
std: :cout < < "The value " < < this-> valueString(value) < < COLOR_GREEN < < "
was found" < < COLOR_RESET
< < std: :endl;
```

```
break;
}
}

if (! current) {
std: :cout < < "The value " < < this-> valueString(value) < < COLOR_RED < < " was
not found" < < COLOR_RESET
< < std: :endl;
return nullptr;
}

if (! current-> left ||! current-> right) {
std: :cout < < COLOR_YELLOW < < "Value has only one child" < < COLOR_RESET < <
std: :endl;

BST< T> :: Node *new_current;

if (! current-> left) {
new_current = current-> right;
} else {
new_current = current-> left;
}

if (! previous) {
std: :cout < < "The value is the root, replacing the root" < < std: :endl;
this-> tree = new_current;
return new_current;
}

if (current == previous-> left) {
std: :cout < < "Placing " < < left_string < < " child on the place of the current" < <
std: :endl;
previous-> left = new_current;
} else {
std: :cout < < "Placing " < < right_string < < " child on the place of the current" < <
std: :endl;
previous-> right = new_current;
}

free(current);
} else {
```

```
std: :cout < < COLOR_YELLOW < < "Value has 2 children" < < COLOR_RESET < < std: :endl;

previous = nullptr;
BST< T> :: Node *to_remove = nullptr;

std: :cout < < "Finding leftmost value of a right child" < < std: :endl;

to_remove = current-> right;

while (to_remove-> left) {
std: :cout < < moving_left_string < < std: :endl;

previous = to_remove;
to_remove = to_remove-> left;
}

std: :cout < < "Removing the value" < < std: :endl;
if (previous) {
previous-> left = to_remove-> right;
} else {
current-> right = to_remove-> right;
}

current-> value = to_remove-> value;

free(to_remove);
}

return this-> tree;
}
};
```

Источники

- https://www.tutorialspoint.com/binary-search-tree-delete-operation-in-cplusplus (7%)
- https://sodocumentation.net/cplusplus/topic/681/std--map (6%)
- https://radioprog.ru/post/1100 (5%)
- https://all-learning.com/binary-search-tree-in-c-c/ (4%)

- https://cplusplus.com/forum/beginner/275875/ (3%)
- https://coollib.com/b/151595/read (3%)
- https://pencilprogrammer.com/algorithms/inorder-preorder-postorder/ (2%)