test plan

Tags

Introduction

I wish I would have more time to write this out. If you look at my functions, I went pretty in depth with my testing. It was completely crucial for the development of this project

BST API Testing

Creating and Destroying Test Cases

This case is focused on creating and destroying the Tree

Creating Tree

Creating tree takes two parameters, both are callback functions. There is really no way to test that they are legit since it is up to the user of the API to figure that out. But the endpoint should return a functioning bst control block with access to the two functions provided. We can test that the functions are being called by making dummy functions that prints hello.

tree_creaion_not_null:

> This test checks to make sure that when the tree is created it is not null. When then free the tree without any nulls to make sure that it there is no memory leaks

tree_destroy_node_null

> This unit test doesn't have any ck_assertions. It relies on valgrind since we are testing for memory leaks.

tree_create_replace

This test case tests to make sure that there is no memory leaks when a node is replaced. It is expected that the payload is freed before the new one is inserted. To test this, a new payload is created with the same key value that is already in the tree. A get of that value will be performed and the pointers and value of both node_payloads will be tested. At first, the values should match, but the pointer should not. When a insert is performed, the values should continue to match, but the pointers will also match since the key that has been fetched has has their payload replaced.

tree_traversal_test

This test will test the three methods of traversal.

```
bst_insert(tree, create_payload(30, 1), BST_REPLACE_FALSE);
bst_insert(tree, create_payload(10, 2), BST_REPLACE_FALSE);
bst_insert(tree, create_payload(28, 3), BST_REPLACE_FALSE);
bst_insert(tree, create_payload(50, 4), BST_REPLACE_FALSE);
bst_insert(tree, create_payload(29, 5), BST_REPLACE_FALSE);
bst_insert(tree, create_payload(55, 5), BST_REPLACE_FALSE);

// create a list of the order that we expected
int values_in_order[6] = {10, 28, 29, 30, 50, 55};
int values_pre_order[6] = {30, 10, 28, 29, 50, 55};
int values_post_order[6] = {29, 28, 10, 55, 50, 30};
```

The nods above are the nods saved to the tree. The three lists are the order of the nodes that we expect to receive. As the traversal method is called, the value for the current node will be checked against the list. On success all values should be equal.

tree_rotation_right

Rotation will be performed on 3 structures. The expectation is to rotate the node identified by the search and then confirm it's output using the pre order output

```
Rotation_node: 14
int values_pre_order_before_rotation[6] = {20, 15, 14, 13, 19};
int values_pre_order_after_rotation[6] = {20, 14, 13, 15, 19};

Rotation_node: 14
int values_pre_order_before_rotation[6] = {20, 15, 14};
```

```
int values_pre_order_after_rotation[6] = {20, 14, 15};

Rotation_node: 15
int values_pre_order_before_rotation[3] = {20, 15, 14};
int values_pre_order_after_rotation[3] = {15, 14, 20};
```