Red-Black Tree (Insertion)

Generated by Doxygen 1.10.0

Chapter 1

Topic Index

1.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Topic Documentation

4.1 Binary Search Tree

Functions

- static Node ∗ node_init (const int data)

  Initializes a new node for a Red-Black tree with given data.
- static void node_destroy (Node ∗node)

  Frees the memory allocated for a Red-Black tree node.
- static Node ∗ bst_insert (Node ∗root, const int data, Node ∗∗z)

  Inserts a new node using standard BST rules and updates the starting point for fixups.
- static Node ∗ bst_search (Node ∗root, const int data)

  Searches for a node with a given value in a BST.

4.1.1 Detailed Description

This section covers a subset of the fundamental operations related to managing nodes within a Red-Black Tree, including initialization and destruction of nodes, as well as the BST insert and search operations. These functions form the basis for maintaining the structural and color properties that define Red-Black Trees. Key operations include:

- `node_init()`: Initializes a new node with specified data, setting it to RED.
- `node_destroy()`: Frees the memory allocated for a node, ensuring no access to its children.
- `bst_insert()`: Inserts a new node into the tree following BST rules, setting up for Red-Black fixups.
- `bst_search()`: Recursively searches for a node by its value, adhering to BST search semantics.

4.1.2 Function Documentation

4.1.2.1 bst_insert()

```
static Node * bst_insert (
            Node * root,
            const int data,
            Node ** z )   [static]
```

Inserts a new node using standard BST rules and updates the starting point for fixups.

This function inserts a new node into the tree rooted at `root`, based on the given `data`. It performs the insertion recursively, ensuring the BST properties are maintained. Upon insertion, the function updates the `z` pointer to reference the newly inserted node. `z` serves as the starting point for subsequent fixup operations to preserve the Red-Black properties.

Parameters

root	The root of the binary search tree. If `root` is NULL, indicating an empty tree, a new node is created, and `z` is set to this node, effectively starting a new tree.
data	The integer value for the new node.
z	A double pointer to a Node, explicitly updated to point to the newly inserted node. This node is used for any necessary fixup operations to adjust the tree's structure and color properties post-insertion.

Returns

The root of the tree after the insertion. This may be the same as the input `root`, or, in the case of inserting into an empty tree, the new node itself.

Note

We set equality to the left of the subtree.

4.1.2.2 bst_search()

```
static Node * bst_search (
            Node * root,
            const int data )  [static]
```

Searches for a node with a given value in a BST.

This function recursively searches for a node containing the specified `data` within a BST. It navigates through the tree based on the binary search property: left subtree contains values less than the node's value, and right subtree contains values greater than the node's value.

Parameters

root	A pointer to the current node of the BST.
data	The value to search for.

Returns

A pointer to the node containing `data` if found, NULL otherwise.

4.1.2.3 node_destroy()

```
static void node_destroy (
            Node * node )  [static]
```

Frees the memory allocated for a Red-Black tree node.

This function releases the memory allocated for a Red-Black tree node and sets its left and right pointers to NULL to prevent any further access. It is designed to be called on a single node after ensuring its children (if any) have been properly handled or deleted to avoid memory leaks.

Parameters

node	A pointer to the node to be destroyed.

Note

This function does not recursively delete the children of the node; it only deletes the node itself. It is the caller's responsibility to handle the deletion of any child nodes to prevent memory leaks.

4.1.2.4 node_init()

```
static Node * node_init (
             const int data )  [static]
```

Initializes a new node for a Red-Black tree with given data.

This function dynamically allocates memory for a new Red-Black tree node, sets its color to RED, and initializes its left, right, and parent pointers to NULL. The node's data is set to the provided value.

Parameters

data	The integer value to be stored in the new node.

Returns

A pointer to the created node if successful, error and exit otherwise.

Note

The function uses malloc for memory allocation and checks for its success. If memory allocation fails, the function prints an error message and exits the program.

4.2 Red-Black Tree Helper Functions

Functions

- static Node ∗ grandparent (Node ∗z)

  Returns the grandparent of a given node in a Red-Black tree.
- static Node ∗ uncle (Node ∗z)

  Returns the uncle of a given node in a Red-Black tree.
- static Node ∗ recolor (Node ∗z)

  Recolors the parent, uncle, and grandparent of a given node in a Red-Black tree.
- static Node ∗ left_rotate (Node ∗x)

  Performs a left rotation on the given node within a Red-Black tree.
- static Node ∗ right_rotate (Node ∗x)

  Performs a right rotation on the given node within a Red-Black tree.
- static Node ∗ restructure (Node ∗z)

  Restructures the tree to maintain Red-Black properties after insertion.
- static Node ∗ fixup (Node ∗z)

  Fixes up the Red-Black tree after insertion to maintain its properties.

4.2.1 Detailed Description

This section describes the helper functions used for maintaining the Red-Black tree properties. These functions are essential for internal operations such as rotations, insertions, deletions, and color adjustments which ensure the tree remains balanced and conforms to red-black tree rules. Key operations include:

- `grandparent()`: Returns the grandparent of a given node.
- `uncle()`: Returns the uncle of a given node.
- `recolor()`: Handles the recolor case, fixing a double red violation.
- `left_rotate()`: Rotates around a node `x` once to the left.
- `right_rotate()`: Rotates around a node `x` once to the right.
- `restructure()`: Restructures the tree with respect to a node `z`, fixing a double red violation.
- `fixup()`: Recursively fixes up the Red-Black tree after insertion.

4.2.2 Function Documentation

4.2.2.1 fixup()

```
static Node * fixup (
             Node * z )  [static]
```

Fixes up the Red-Black tree after insertion to maintain its properties.

This function adjusts the colors and structure of a Red-Black tree starting from node `z` upwards to ensure that the tree maintains balance after an insertion. The fixup procedure includes recoloring nodes and performing rotations as necessary.

Parameters

z	A pointer to the newly inserted node or a node that may cause a double red violation.

Returns

A pointer to the root of the (valid Red-Black) tree after the adjustments.

Note

The function recursively addresses two main cases:

1. If `z's` parent is NULL, `z` is the root and is colored BLACK.
2. If both `z` and its parent are RED, then it checks the color of `z's` uncle.
   - If the uncle is RED, we recolor.
   - If the uncle is BLACK or NULL, we restructure with respect to `z`. After (recursively) addressing these cases, the function ascends the tree to its root to return it.

4.2.2.2 grandparent()

```
static Node * grandparent (
            Node * z )  [static]
```

Returns the grandparent of a given node in a Red-Black tree.

This helper function makes the code easier to read.

Parameters

z	A pointer to the grandchild of the node we want.

Returns

A pointer to the grandparent node if it exists; NULL otherwise.

Note

This function assumes that the provided node `z` and its parent are non-NULL. It is the caller's responsibility to ensure this condition is met before calling the function. Failing to do so may result in undefined behavior.

4.2.2.3 left_rotate()

```
static Node * left_rotate (
            Node * x )  [static]
```

Performs a left rotation on the given node within a Red-Black tree.

This function applies a left rotation to the node pointed to by `x`, adjusting the tree's structure to maintain Red-Black properties. Below is an ASCII art diagram illustrating the left rotation:

```
*    p              p
*    |              |
*    x              y
*   / \            / \
*  _   y    =>    x   z
*     / \        / \
*    T   z      _   T
*
```

Here, we label the left child of `x` as _ because it is irrelevant to the rotation. Additionally, `x's` parent `p` is directly above `x` since we need to check if `x` is a left or right child of `p`.

Parameters

x	Pointer to the node to rotate. This node becomes the left child of its right child after rotation.

Returns

The new root of the subtree after the rotation, which was the right child of `x` before the rotation.

Note

This function assumes `x's` right child is non-NULL and it's the caller's responsibility to ensure this. These conditions are met by definition of how we call the function. We call the function with respect to `z`, so `y` is defined.

4.2.2.4 recolor()

```
static Node * recolor (
            Node * z )  [static]
```

Recolors the parent, uncle, and grandparent of a given node in a Red-Black tree.

This function changes the color of the parent and uncle of the node `z` to BLACK, and the color of the grandparent to RED.

Parameters

z	A pointer to the node we are recoloring with respect to.

Returns

A pointer to the grandparent of the node `z` after recoloring.

Note

This function assumes that the node `z`, its parent, and its grandparent are non-NULL, and that it has an uncle. These conditions are met by definition since by property 3, nil nodes are treated as BLACK.

4.2.2.5 restructure()

```
static Node * restructure (
            Node * z )  [static]
```

Restructures the tree to maintain Red-Black properties after insertion.

This function performs rotations on a given node `z` in a Red-Black tree to fix any violations caused by insertion. There are four cases considered for rotation: left-left, right-right, left-right, and right-left, each requiring specific rotations to maintain the tree's balancing properties.

Parameters

z	A pointer to the node that we rotate with respect to.

Returns

A pointer to the node that becomes the new root of the subtree after the rotations are completed.

Note

The function modifies the tree structure through rotations and recolors the nodes to adhere to Red-Black tree properties. The specific rotations performed depend on the position of `z` relative to its parent and grandparent, addressing the different cases of tree imbalance:

- Left-Left Case: A single right rotation is needed.
- Right-Right Case: A single left rotation is needed.
- Left-Right Case: A right rotation followed by a left rotation is needed.
- Right-Left Case: A left rotation followed by a right rotation is needed. After the rotations, the function correctly recolors affected nodes.

The code is purposefully verbose as it serves an educational purpose. Given `z`, `p`, `x`, we can reduce redundancy by nesting cases as follows:

```
*  if @p z is a left child:
*    if @p p is a left child:
*      left_rotate()
*    else:
*      right_rotate(left_rotate())
*
*    recolor appropriately
*  else:
*    if @p p is a right child:
*      right_rotate()
*    else:
*      left_rotate(right_rotate)
*
*    recolor appropriately
*
*  return z
*
```

4.2.2.6 right_rotate()

```
static Node * right_rotate (
            Node * x )  [static]
```

Performs a right rotation on the given node within a Red-Black tree.

This function applies a right rotation to the node pointed to by `x`, adjusting the tree's structure to maintain Red-Black properties. Below is an ASCII art diagram illustrating the right rotation:

```
*       g              g
*       |              |
*       x              y
*      / \            / \
*     y   _    =>    z   x
*    / \                / \
*   z   T              T   _
*
```

Here, we label the right child of `x` as _ because it is irrelevant to the rotation. Additionally, `x's` parent `p` is directly above `x` since we need to check if `x` is a left or right child of `p`.

Parameters

x	Pointer to the node to rotate. This node becomes the right child of its left child after rotation.

Returns

The new parent node after the rotation, which was the left child of `x` before the rotation.

Note

Assumes `x's` left child is non-NULL and it's the caller's responsibility to ensure this.

4.2.2.7 uncle()

```
static Node * uncle (
            Node * z )  [static]
```

Returns the uncle of a given node in a Red-Black tree.

The function returns the uncle of a node, which is defined as the sibling of the node's parent.

Parameters

z	A pointer to the node whose uncle is desired.

Returns

A pointer to the uncle node if it exists; NULL otherwise. This is because if the grandparent does not exist, it implies that the uncle cannot exist.

4.3 Tree Formatter

Functions

- static int height (Node ∗root)

  Calculates the height of the tree.
- int col (int h)

  Calculates the column position for a given height.
- static void print_tree (int ∗∗mat_tree, char ∗∗mat_color, Node ∗root, int c, int r, int h)

  Utility function to print the tree structure.
- void rbt_print_tree (Node ∗root)

  Prints the entire Red-Black Tree.
- void rbt_inorder (Node ∗root)

  Performs an inorder traversal of the Red-Black Tree.

4.3.1 Detailed Description

This section covers all the helper functions for formatting so we can pretty-print the tree. Key operations include:

- `height()`: Returns the height of the tree.
- `col()`: Returns the column position for a given height.

4.3.2 Function Documentation

4.3.2.1 col()

```
int col (
             int h )
```

Calculates the column position for a given height.

This function determines the column position based on the height of the tree, which is used for pretty printing purposes.

Parameters

h	The height of the tree.

Returns

The calculated column position.

4.3.2.2 height()

```
static int height (
             Node * root )  [static]
```

Calculates the height of the tree.

This function computes the height of the tree recursively.

Parameters

root	A pointer to the root node of the tree.

Returns

The height of the tree.

4.3.2.3 print_tree()

```
static void print_tree (
             int ** mat_tree,
             char ** mat_color,
             Node * root,
             int c,
             int r,
             int h )  [static]
```

Utility function to print the tree structure.

4.3.3 Printing Section

This section is dedicated to the functions required for printing the tree. These functions are specifically designed for pretty printing and do not directly relate to the tree's logical operations. Key operations include:

- `print_treee()`: Utility function to pretty print the tree.
- `rbt_print_tree()`: Pretty prints the tree.
- `rbt_inorder()`: Traverses the tree in order.

This function prints the tree structure. It uses matrix representation for the tree where each node's position is calculated based on its height and position in the tree.

Parameters

mat_tree	The matrix representation of the tree's structure.
mat_color	The matrix representing the color of each node in the tree.
root	A pointer to the root node of the tree.
c	The column position for the root.
r	The row position for the root.
h	The height of the tree.

4.3.3.1 rbt_inorder()

```
void rbt_inorder (
            Node * root )
```

Performs an inorder traversal of the Red-Black Tree.

Prints the elements of the tree in an in-order fashion. This function can be used for debugging purposes to visualize the tree structure and contents.

Parameters

root	A pointer to the root node of the Red-Black Tree.

4.3.3.2 rbt_print_tree()

```
void rbt_print_tree (
            Node * root )
```

Prints the entire Red-Black Tree.

Prints the tree in a structured manner to visualize its layout, including the color of each node. This is useful for debugging and verifying the structure of the tree.

Parameters

root	A pointer to the root node of the Red-Black Tree.

4.4 Red-Black Tree

Functions

- void rbt_destroy (Tree ∗tree)

  Destroys a Red-Black tree and frees its memory.
- Node ∗ rbt_insert (Tree ∗tree, const int data)

  Inserts a value into the Red-Black tree.
- Node ∗ rbt_search (Tree ∗tree, const int data)

  Searches for a node with a given value in a Red-Black Tree.

4.4.1 Detailed Description

This section documents the public facing API for the red-black tree. It includes functions for creating the tree, inserting nodes, searching for values, and other utilities that facilitate interaction with the red-black tree data structure. Key operations include:

- `rbt_init()`: Initializes a new tree with a NULL root and size of 0.
- `rbt_destroy()`: Frees the memory allocated for the entire tree.
- `rbt_insert()`: Inserts a new node into the Red-Black tree, fixing up recursively to maintain the balance properties.
- `rbt_search()`: Recursively searches for a node by its value, adhering to BST search semantics.

4.4.2 Function Documentation

4.4.2.1 rbt_destroy()

```
void rbt_destroy (
            Tree * tree )
```

Destroys a Red-Black tree and frees its memory.

This function recursively frees the memory allocated for the nodes of a Red-Black tree. It ensures that all associated memory is released to prevent memory leaks.

Parameters

tree	A pointer to the Red-Black tree to be destroyed.

Note

This function recursively destroys the entire tree. To destroy a single node, use `node_destroy()` instead.

4.4.2.2 rbt_insert()

```
Node * rbt_insert (
            Tree * tree,
            const int data )
```

Inserts a value into the Red-Black tree.

function first inserts the new node using standard BST rules, empirically setting equality to the left subtree. After insertion, we perform a (potentially) recursive fixup starting at the newly inserted node.

Note

The root node may change as a result of these adjustments. This is because we may rotate around the root of the tree. It is important to use this new root for future operations on the tree.

Parameters

tree	A pointer to the the tree.
data	The integer value to be inserted.

Returns

A pointer to the root of the tree.

4.4.2.3 rbt_search()

```
Node * rbt_search (
            Tree * tree,
            const int data )
```

Searches for a node with a given value in a Red-Black Tree.

This function acts as a wrapper for the `bst_search()` function, initiating a search for a node containing the specified `data` within a Red-Black Tree.

Parameters

tree	A pointer to the Red-Black Tree we want to search.
data	The value to search for.

Returns

A pointer to the node containing `data` if found, NULL otherwise.

Chapter 5

Data Structure Documentation

5.1 Node Struct Reference

Node structure for a Red-Black Tree.

`#include <rbt.h>`

Data Fields

- Color color
- struct Node ∗ left
- struct Node ∗ right
- struct Node ∗ parent
- int data

5.1.1 Detailed Description

Node structure for a Red-Black Tree.

Each node in the Red-Black Tree contains the data it stores, pointers to its left and right children, a pointer to its parent, and its color.

5.1.2 Field Documentation

5.1.2.1 color

`Node::color`

The color of the node is either red or black.

5.1.2.2 data

`Node::data`

The data stored in the node. For simplicity, this implementation considers an integer.

5.1.2.3 left

`Node::left`

Pointer to the left child of the node. If the node does not have a left child, this pointer is NULL.

5.1.2.4 parent

`Node::parent`

Pointer to the parent of the node. For the root node, this pointer is NULL.

5.1.2.5 right

`Node::right`

Pointer to the right child of the node. If the node does not have a right child, this pointer is NULL.

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

- rbt.h

5.2 Tree Struct Reference

Structure representing a Red-Black Tree.

`#include <rbt.h>`

Data Fields

- Node ∗ root
- size_t size

5.2.1 Detailed Description

Structure representing a Red-Black Tree.

This structure is used to maintain a Red-Black Tree. It encapsulates the root of the tree and the total number of nodes within the tree, providing a high-level interface for operations on the tree such as insertion, and search.

5.2.2 Field Documentation

5.2.2.1 root

`Tree::root`

Pointer to the root node of the Red-Black Tree. It points to NULL when the tree is empty.

5.2.2.2 size

`Tree::size`

The total number of nodes in the tree. This count helps in operations that may require knowledge of the tree's size, such as balancing, validation, and traversal optimizations.

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

- rbt.h

Chapter 6

File Documentation

6.1 main.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include "rbt.h"
#include <unistd.h>
```

Functions

- bool valid_int (const char ∗str)
- int main ()

6.1.1 Function Documentation

6.1.1.1 main()

```
int main ( )
```

6.1.1.2 valid_int()

```
bool valid_int (
            const char * str )
```

6.2 rbt.c File Reference

Implementation of a Red-Black Tree in C (Insertion Only).

```
#include "rbt.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

Functions

- static Node ∗ node_init (const int data)

  Initializes a new node for a Red-Black tree with given data.
- static void node_destroy (Node ∗node)

  Frees the memory allocated for a Red-Black tree node.
- static Node ∗ bst_insert (Node ∗root, const int data, Node ∗∗z)

  Inserts a new node using standard BST rules and updates the starting point for fixups.
- static Node ∗ bst_search (Node ∗root, const int data)

  Searches for a node with a given value in a BST.
- static Node ∗ grandparent (Node ∗z)

  Returns the grandparent of a given node in a Red-Black tree.
- static Node ∗ uncle (Node ∗z)

  Returns the uncle of a given node in a Red-Black tree.
- static Node ∗ recolor (Node ∗z)

  Recolors the parent, uncle, and grandparent of a given node in a Red-Black tree.
- static Node ∗ left_rotate (Node ∗x)

  Performs a left rotation on the given node within a Red-Black tree.
- static Node ∗ right_rotate (Node ∗x)

  Performs a right rotation on the given node within a Red-Black tree.
- static Node ∗ restructure (Node ∗z)

  Restructures the tree to maintain Red-Black properties after insertion.
- static Node ∗ fixup (Node ∗z)

  Fixes up the Red-Black tree after insertion to maintain its properties.
- static int height (Node ∗root)

  Calculates the height of the tree.
- int col (int h)

  Calculates the column position for a given height.
- static void print_tree (int ∗∗mat_tree, char ∗∗mat_color, Node ∗root, int c, int r, int h)

  Utility function to print the tree structure.
- void rbt_print_tree (Node ∗root)

  Prints the entire Red-Black Tree.
- void rbt_inorder (Node ∗root)

  Performs an inorder traversal of the Red-Black Tree.
- Tree ∗ rbt_init ()

  Initializes a new Red-Black tree.
- void rbt_destroy (Tree ∗tree)

  Destroys a Red-Black tree and frees its memory.
- Node ∗ rbt_insert (Tree ∗tree, const int data)

  Inserts a value into the Red-Black tree.
- Node ∗ rbt_search (Tree ∗tree, const int data)

  Searches for a node with a given value in a Red-Black Tree.

6.2.1 Detailed Description

Implementation of a Red-Black Tree in C (Insertion Only).

This file contains the definition and implementation details of a Red-Black Tree, a self-balancing binary search tree. In a Red-Black Tree, each node contains an extra bit for denoting the color of the node, either RED or BLACK. A Red-Black tree satisfies the following properties:

1. Every node is either RED or BLACK.
2. Red nodes cannot have RED children; i.e. RED nodes can only have BLACK children.
3. NULL nodes are BLACK.
4. The root is always BLACK.
5. Every path from a node to its descendant NULL nodes has the same number of BLACK nodes.

The tree structure supports operations such as insertion, deletion, and search with guaranteed log(n) maximum time complexity for each operation, making it efficient for data retrieval and manipulation. Space complexity is linear; i.e. O(n).

Key Components:

- Color: An enumeration defining the color (RED, BLACK) of nodes in the tree.
- Node: A structure representing a node in the Red-Black Tree, containing data, color, and pointers to its children and parent.
- Tree: A structure representing the entire Red-Black Tree, encapsulating its root and size.

Key Functions:

- rbt_init(): Initializes and returns a new Red-Black Tree.
- rbt_destroy(): Frees memory allocated for the Red-Black Tree.
- rbt_insert(): Inserts a new node with the given data into the tree.
- rbt_inorder(): Performs an inorder traversal of the tree.
- rbt_print_tree(): Prints the tree structure.

This file provides a basic implementation and can be extended for more complex operations and use cases.

Version

1.0

Date

2024-02-27

Author

Warren Kim

6.2.2 Function Documentation

6.2.2.1 rbt_init()

```
Tree * rbt_init ( )
```

Initializes a new Red-Black tree.

This function dynamically allocates memory for a new Red-Black tree structure, sets its root to NULL, and initializes its size to 0. It returns a pointer to the newly created tree.

Returns

A pointer to the newly initialized Red-Black tree if successful, error and exits otherwise.

Note

The function checks for successful memory allocation and exits the program with an error message if allocation fails. This ensures that the function returns a valid pointer to a Red-Black tree.

6.3 rbt.h File Reference

Declaration of Red-Black Tree data structures and functions.

```
#include <stddef.h>
```

Data Structures

- struct Node

  Node structure for a Red-Black Tree.
- struct Tree

  Structure representing a Red-Black Tree.

Typedefs

- typedef struct Node Node
- typedef struct Tree Tree

Enumerations

- enum Color { RED , BLACK }

  Represents the color of a node in a Red-Black tree.

Functions

- Tree ∗ rbt_init ()

  Initializes a new Red-Black tree.
- void rbt_destroy (Tree ∗tree)

  Destroys a Red-Black tree and frees its memory.
- Node ∗ rbt_insert (Tree ∗tree, const int data)

  Inserts a value into the Red-Black tree.
- void rbt_inorder (Node ∗root)

  Performs an inorder traversal of the Red-Black Tree.
- void rbt_print_tree (Node ∗root)

  Prints the entire Red-Black Tree.
- Node ∗ rbt_search (Tree ∗tree, const int data)

  Searches for a node with a given value in a Red-Black Tree.

6.3.1 Detailed Description

Declaration of Red-Black Tree data structures and functions.

This header file contains the declarations for the Red-Black Tree data structures and outlines the interface for operations on a Red-Black Tree. A Red-Black Tree is a kind of self-balancing binary search tree where each node stores an extra bit representing its color (RED or BLACK) to ensure the tree remains approximately balanced. This property enables the tree to support insertion, deletion, and search operations in O(log n) time, making it efficient for various applications. A Red-Black tree satisfies the following properties:

1. Every node is either RED or BLACK.
2. Red nodes cannot have RED children; i.e. RED nodes can only have BLACK children.
3. NULL nodes are BLACK.
4. The root is always BLACK.
5. Every path from a node to its descendant NULL nodes has the same number of BLACK nodes.

Key Components:

- Color: An enumeration that defines the possible colors (RED, BLACK) of nodes in the tree.
- Node: A structure representing a node within the Red-Black Tree. It includes the node's data, its color, and pointers to its children and parent.
- Tree: A structure representing the Red-Black Tree itself, encapsulating a pointer to its root and the total number of nodes in the tree.

Key Functions (Declared):

- rbt_init(): Creates and returns a new instance of a Red-Black Tree.
- rbt_destroy(): Destroys the Red-Black Tree, freeing all allocated memory.
- rbt_insert(): Inserts a new element with the specified data into the tree.
- rbt_inorder(): Conducts an inorder traversal of the tree.
- rbt_print_tree(): Prints the structure of the tree.

This header file should be included in any source file that intends to utilize the Red-Black Tree data structures or operations. The implementation details are provided in the corresponding source file (`rbtree.c`).

Version

1.0

Date

2024-02-27

Author

Warren Kim

6.3.2 Typedef Documentation

6.3.2.1 Node

```
struct Node
```

6.3.2.2 Tree

```
struct Tree
```

6.3.3 Enumeration Type Documentation

6.3.3.1 Color

```
enum enum Color
```

Represents the color of a node in a Red-Black tree.

This enumeration is used to represent the color property of nodes within a Red-Black tree.

Enumerator

RED	
BLACK	

6.3.4 Function Documentation

6.3.4.1 rbt_init()

```
Tree * rbt_init ( )
```

Initializes a new Red-Black tree.

This function dynamically allocates memory for a new Red-Black tree structure, sets its root to NULL, and initializes its size to 0. It returns a pointer to the newly created tree.

Returns

A pointer to the newly initialized Red-Black tree if successful, error and exits otherwise.

Note

The function checks for successful memory allocation and exits the program with an error message if allocation fails. This ensures that the function returns a valid pointer to a Red-Black tree.

6.4 rbt.h

Go to the documentation of this file.

```

#include <stddef.h>

typedef enum { RED, BLACK } Color;


typedef struct Node {
    Color color;
    struct Node  *left;
    struct Node  *right;
    struct Node  *parent;
    int data;
} Node;

typedef struct Tree {
    Node *root;
    size_t size;
} Tree;

Tree *rbt_init();

void rbt_destroy(Tree *tree);

Node *rbt_insert(Tree *tree, const int data);

void rbt_inorder(Node *root);

void rbt_print_tree(Node *root);

Node *rbt_search(Tree *tree, const int data);
```

Index