Is It A Red-Black Tree

黄文杰

3210103379

Date：2023-11-1

Chapter 1: Introduction

A balanced binary search tree known as a red-black tree is found in data structures. It possesses the following five characteristics:

- **Each node is either red or black.**
- **The root is black.**
- **Every leaf (NULL) is black.**
- **If a node is red, both of its children are black.**
- **For any given node, all straightforward paths from that node to its descendant leaves have an equal number of black nodes.**

In this context, our challenge is to determine whether a given binary search tree adheres to the properties of a Red-Black Tree. This is a critical problem, as the validity of Red-Black Trees directly impacts their applications in computer science, including databases, operating systems, compilers, and more. This report will comprehensively describe how our program determines whether a given binary search tree is a valid Red-Black Tree, highlighting the underlying principles and importance of this process.

Chapter 2: Algorithm Specification

- **Structure of a red-black tree node**

```
// Structure for a Red-Black Tree node
typedef struct RBTreeNode {
    int value;              // Node's key
    int color;            // Node's color, can be red (RED) or black (BLACK)
    struct RBTreeNode* left;    // Left child
    struct RBTreeNode* right;   // Right child
} Node;
```

- **createNode(int value, int color)**

```
Node* createNode(int value, int color)
    newNode <- allocate memory for a new RBTreeNode
    newNode.value <- value
    newNode.color <- color
    newNode.left <- NULL
    newNode.right <- NULL
    return newNode
```

> - **Parameter Description :**
>   - Input Parameters: An integer `value` representing the node key, an integer `color` representing the node color.
>   - Output: Returns a pointer to a new node.

- Function's purpose: Creates and returns a red-black tree node.

- **Algorithm Description :**
  1. Create a new node `newNode`.
  2. Set the `value` of `newNode` to the input parameter `value` and the `color` to the input parameter `color`.
  3. Initialize the `left` and `right` pointers of `newNode` to NULL.
  4. Return `newNode`.

- **insertNode(Node* root, Node* node)**

```
Node* insertNode(Node* root, Node* node)
    if root == NULL
        return node
    if node.value < root.value and root.left == NULL
        root.left <- node
        return root
    if node.value > root.value and root.right == NULL
        root.right <- node
        return root
    if node.value < root.value
        root.left <- insertNode(root.left, node)
        return root
    if node.value > root.value
        root.right <- insertNode(root.right, node)
        return root
```

- **Parameter Description :**
  - Input Parameters: `root` represents the root of the current subtree, `node` represents the node to be inserted.
  - Output: Returns the updated root of the subtree.
  - Function's purpose: Inserts a new node into the red-black tree.
- **Algorithm Description :**
  1. If `root` is NULL, return `node`.
  2. If the value of `node` is less than the value of `root` and the left subtree of `root` is empty, insert `node` as the left child of `root` and return `root`.
  3. If the value of `node` is greater than the value of `root` and the right subtree of `root` is empty, insert `node` as the right child of `root` and return `root`.
  4. If the value of `node` is less than the value of `root`, recursively call `insertNode` with `root->left` and `node`, and set the result as `root->left` and return `root`.
  5. If the value of `node` is greater than the value of `root`, recursively call `insertNode` with `root->right` and `node`, and set the result as `root->right` and return `root`.

- **buildRBTree(int *node, int size)**

```
Node* buildRBTree(int *node, int size)
    root <- NULL
    for i <- 0 to size - 1
        if node[i] < 0
            color <- RED
            value <- -node[i]
        else
            color <- BLACK
            value <- node[i]
        newNode <- createNode(value, color)
        root <- insertNode(root, newNode)
    return root
```

- **Parameter Description :**
  - Input Parameters: An integer array `node` representing node colors and keys, an integer `size` representing the array size.
  - Output: Returns a pointer to the root node.
  - Function's purpose: Builds and returns a red-black tree.
- **Algorithm Description :**
  1. Initialize `color` and `value` to 0.
  2. Initialize `root` as NULL.
  3. Iterate through elements of the array `node` :
     - If `node[i]` is negative, set `color` to RED and `value` to `-node[i]`; otherwise, set `color` to BLACK and `value` to `node[i]`.
     - Create a new node `newNode` using the `createNode` function, passing `value` and `color`.
     - Call the `insertNode` function to insert `newNode` into the red-black tree.
  4. Return the root node `root`.

- **checkRBTree(Node* root, int* blackHeight)**

```
int checkRBTree(Node* root, int* blackHeight)
    if root == NULL
        blackHeight <- 0
        return 1
    if root.color == RED
        if root.left != NULL and root.left.color == RED
            return 0
        if root.right != NULL and root.right.color == RED
            return 0
    leftBlackHeight <- 0
    rightBlackHeight <- 0
    if not checkRBTree(root.left, leftBlackHeight)
        return 0
    if not checkRBTree(root.right, rightBlackHeight)
        return 0
    if leftBlackHeight != rightBlackHeight
        return 0
    if root.color == BLACK
```

```
        blackHeight <- leftBlackHeight + 1
    else
        blackHeight <- leftBlackHeight
    return 1
```

- **Parameter Description :**
  - Input Parameters: `root` represents the root of the current subtree, `blackHeight` represents the black height of the subtree.
  - Output: Returns 1 (true) if it is a red-black tree, 0 (false) otherwise.
  - Function's purpose: Checks if a subtree satisfies red-black tree properties and updates the black height of the subtree.
- **Algorithm Description :**
  1. If `root` is NULL, set `blackHeight` to 0 and return 1.
  2. If the color of `root` is RED, check its left and right child nodes, and return 0 if either child is RED.
  3. Initialize `leftBlackHeight` and `rightBlackHeight` to 0.
  4. Recursively call `checkRBTree` with `root->left` and `leftBlackHeight`, and store the result as `leftResult`.
  5. Recursively call `checkRBTree` with `root->right` and `rightBlackHeight`, and store the result as `rightResult`.
  6. If `leftBlackHeight` and `rightBlackHeight` are not equal, return 0.
  7. If the color of `root` is BLACK, update `blackHeight` to `leftBlackHeight + 1`; otherwise, set `blackHeight` to `leftBlackHeight`.
  8. Return 1 (Because all the preceding conditions have been met).

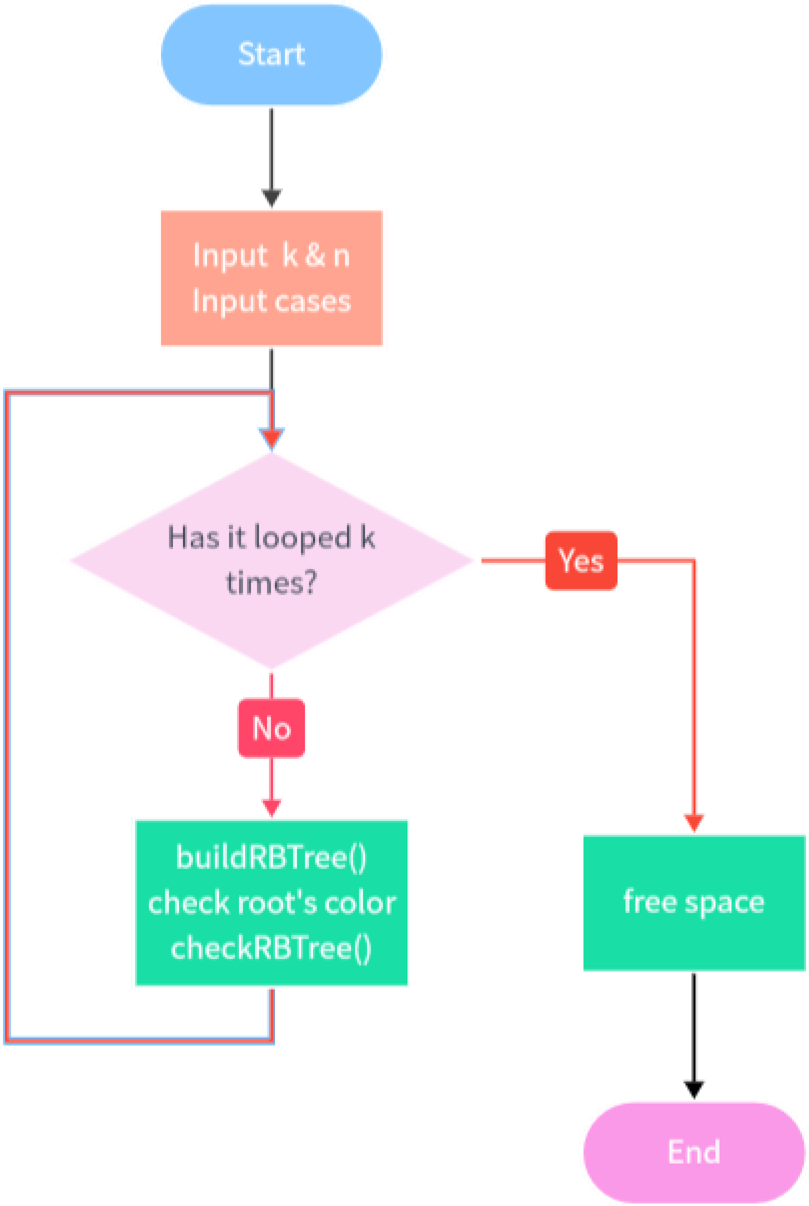
- **freeNodeSpace(Node* root)**

```
void freeNodeSpace(Node* root)
    if root == NULL
        return
    if root.left == NULL
        freeNodeSpace(root.right)
        free(root)
        return
    if root.right == NULL
        freeNodeSpace(root.left)
        free(root)
        return
    freeNodeSpace(root.left)
    freeNodeSpace(root.right)
    free(root)
    return
```

- **Parameter Description :**
  - Input Parameters: `root` represents the root of the subtree to be freed.
  - Output: None.
  - Function's purpose: Releases memory for the red-black tree.
- **Algorithm Description :**
  1. If `root` is NULL, return.

2. If the left subtree of `root` is empty, recursively call `freeNodeSpace` with `root->right`, then free the memory for `root`.
3. If the right subtree of `root` is empty, recursively call `freeNodeSpace` with `root->left`, then free the memory for `root`.
4. Otherwise, recursively call `freeNodeSpace` with `root->left`, then recursively call `freeNodeSpace` with `root

- **a sketch of the main program**



- **the main program**

```
int main(void){
```

```
    int k,n; // k:total number of cases; n: the total number of nodes in the binary tree
    int i,j;
    printf("Input k: ");
    scanf("%d",&k);
    for(i=0;i<k;i++){
        // Input n
        printf("Input n: ");
        scanf("%d",&n);
        // Allocate a dynamic array of integers using malloc
        int* nodeArray = (int*)malloc(n * sizeof(int));
        if (nodeArray == NULL) {
            printf("Memory allocation failed.\n");
            return 1;
        }
        // Define the root of a RBTree
        Node* root = NULL;
        // Accept user input to initialize the array.
        printf("Input n integers: ");
        for(j=0;j<n;j++){
            scanf("%d",&nodeArray[j]);
        }
        root=buildRBTree(nodeArray,n);
        int blackHeight=0;
        // First check if the root is RED
        if(root!=NULL&&root->color==RED){
            printf("Results: \n");
            printf("No\n");
            printf("\n");
        }else if(checkRBTree(root,&blackHeight)){
            printf("Results: \n");
            printf("Yes\n");
            printf("\n");
        }else{
            printf("Results: \n");
            printf("No\n");
            printf("\n");
        }

        // Free the allocated memory
        freeNodeSpace(root);
        free(nodeArray);
    }
}
```

Chapter 3: Testing Results

1. ## Test Case 1: Comprehensive Test

Case1(question-provided test cases) :

- **Input :**

```
3
9
7 -2 1 5 -4 -11 8 14 -15
9
11 -2 1 -7 5 -4 8 14 -15
8
10 -7 5 -6 8 15 -11 17
```

- **Output & Expected Result :**

```
Output:
Yes
No
No

Expected Result:
Yes
No
No
```

- **Testing Purpose :**

  This test case includes a moderately complex Red-Black Tree with both red and black nodes and two wrong cases. It tests the program's ability to handle various node values and validate a balanced Red-Black Tree.

Case2(other test cases) :

- **Input :**

```
1
11
53 -34 -80 18 46 74 88 -17 -33 -50 -72
```

- **Output & Expected Result :**

```
Output:
Yes

Expected Result:
Yes
```

- **Testing Purpose :**

This test case includes a moderately complex Red-Black Tree with both red and black nodes . It tests the program's ability to handle various node values and validate a balanced Red-Black Tree.

2. **Test Case 2: Smallest & Largest Input Size**

`Small Size`

Blcak root :

- **Input :**

```
1
1
5
```

- **Output & Expected Result :**

```
Output:
Yes

Expected Result:
Yes
```

- **Testing Purpose :**

  This test case focuses on the smallest possible Red-Black Tree, having just one node. It checks if the program can correctly identify a valid Red-Black Tree with the minimum size (black root node).

Red root:

- **Input :**

```
1
1
-6
```

- **Output & Expected Result :**

```
Output:
No

Expected Result:
No
```

- **Testing Purpose :**

  This test case focuses on the smallest possible Red-Black Tree, having just one node. It checks if the program can correctly identify a valid Red-Black Tree with the minimum size (red root node).

`Large size:`

- **Input :**

```
2
30
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
100
1 -2 3 -4 5 -6 7 -8 9 -10 11 -12 13 -14 15 -16 17 -18 19 -20 21 -22 23 -24 25 -26 27 -28 29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98 -99 -100
```

- **Output & Expected Result :**

```
Output:
No
No

Expected Result:
No
No
```

- **Testing Purpose :**

  Test whether the program can handle cases with a large size.

3. ## Test Case 3: Extreme Case Test

Empty test case :

- **Input :**

```
0
```

- **Output & Expected Result :**

```
Output:

Expected Result:

```

- **Testing Purpose :**

  Test whether the program can handle zero input cases

An empty tree :

- **Input :**

```
1
0
(null)
```

- **Output & Expected Result :**

```
Output:
Yes

Expected Result:
Yes
```

- **Testing Purpose :**

  Test whether the program can handle the case of an empty tree and return the correct result.

Large Input Size with Unbalanced Red Nodes :

- **Input :**

```
1
10
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
```

- **Output & Expected Result :**

```
Output:
No

Expected Result:
No
```

- **Testing Purpose :**

  Confirm that the program detects an unbalanced tree with all red nodes.

Chapter 4: Analysis and Comments

- ## Time complexity analysis

> **buildRBTree Function:**
>
> - Iterating through the `node` array to create nodes and insert them into the Red-Black Tree: This operation takes O(n) time, where n is the number of nodes in the tree.
> - The `createNode` function takes constant time to allocate memory and initialize the node's attributes.

> - The `insertNode` function, when called for insertion, takes O(h) time for each node insertion, where h is the height of the tree. In the worst case, when the tree is highly unbalanced, it can take O(n) time.
> - So, the total time complexity of building the Red-Black Tree is O(n * h), where h can range from log(n) to n.

> **checkRBTree Function:**
>
> - The function is checking the properties of a Red-Black Tree recursively. In the worst case, it will visit every node in the tree, so the time complexity is O(n).

> **freeNodeSpace Function:**
>
> - This function recursively traverses and deallocates the memory for each node. Similar to the `checkRBTree` function, it can visit every node in the tree, resulting in O(n) time complexity.

- ### Space complexity analysis

> **Memory for nodeArray:**
>
> - In the `main` function, it allocates memory for an integer array `nodeArray` of size `n`. So, the space complexity here is O(n).

> **Memory for Red-Black Tree:**
>
> - The memory used by the Red-Black Tree itself is mainly for the nodes and their attributes. In the worst case, if the tree is highly unbalanced, the space complexity can be O(n).
> - Additionally, for the recursive calls in `checkRBTree` and `freeNodeSpace`, they have function call stack space. In the worst case, the stack space can be O(h), where h is the height of the tree. For a balanced tree, this is O(log(n)), but for an unbalanced tree, it can be O(n).
> - The `createNode` function also allocates memory for each node, but the space required is proportional to the number of nodes, so it's also O(n).

> **Total Space Complexity:**
>
> - The sum of the above space complexities is O(n) for `nodeArray` + O(n) for the Red-Black Tree + O(h) for the function call stack space. Therefore, the overall space complexity is O(n + h), where h is the height of the tree.

- ## Conclusion

  In conclusion, the time complexity of the program is O(n * h), where n is the number of nodes in the tree, and h is the height of the tree. The space complexity is O(n + h), where n is the number of nodes and h is the height of the tree. The actual time and space complexity can vary depending on the balance of the Red-Black Tree.

Appendix: Source Code (in C)

```c
#include <stdio.h>
#include <stdlib.h>

// Define colors for the Red-Black Tree
#define RED 0
#define BLACK 1

// Structure for a Red-Black Tree node
typedef struct RBTreeNode {
    int value;              // Node's key
    int color;            // Node's color, can be red (RED) or black (BLACK)
    struct RBTreeNode* left;    // Left child
    struct RBTreeNode* right;   // Right child
} Node;

// Build Red-Black Tree
Node* buildRBTree(int *node, int size);
// Create a new node
Node* createNode(int key, int color);
// Insert a node into a RBTree and return the root of the RBTree
Node* insertNode(Node* root, Node* node);
// Check if a Binary-Tree is RBTree (1:true 0:false)
int checkRBTree(Node* root, int* blackHeight);
// Free the memory allocated for Red-Black tree nodes
void freeNodeSpace(Node* root);

int main(void){
    int k,n; // k:total number of cases; n: the total number of nodes in the binary tree
    int i,j;
    printf("Input k: ");
    scanf("%d",&k);
    for(i=0;i<k;i++){
        // Input n
        printf("Input n: ");
        scanf("%d",&n);
        // Allocate a dynamic array of integers using malloc
        int* nodeArray = (int*)malloc(n * sizeof(int));
        if (nodeArray == NULL) {
            printf("Memory allocation failed.\n");
```

```
            return 1;
        }
        // Define the root of a RBTree
        Node* root = NULL;
        // Accept user input to initialize the array.
        printf("Input n integers: ");
        for(j=0;j<n;j++){
            scanf("%d",&nodeArray[j]);
        }
        root=buildRBTree(nodeArray,n);
        int blackHeight=0;
        // First check if the root is RED
        if(root!=NULL&&root->color==RED){
            printf("Results: \n");
            printf("No\n");
            printf("\n");
        }else if(checkRBTree(root,&blackHeight)){
            printf("Results: \n");
            printf("Yes\n");
            printf("\n");
        }else{
            printf("Results: \n");
            printf("No\n");
            printf("\n");
        }

        // Free the allocated memory
        freeNodeSpace(root);
        free(nodeArray);
    }
}

// Build Red-Black Tree
Node* buildRBTree(int *node, int size){
    int color = 0;
    int value = 0;
    int i;
    Node* root=NULL;
    for(i=0;i<size;i++){
        // Determine the color of the node based on its sign form.
        if(node[i]<0){
            color = RED;
            value=-node[i];
        }else{
            color=BLACK;
            value=node[i];
        }
        // Call functions to create nodes and insert nodes to build a Red-Black tree.
        Node* node = createNode(value,color);
        root=insertNode(root,node);
    }
    return root;
}

```

```c
// Create a new node
Node* createNode(int value, int color) {
    Node* newNode = (Node*)malloc(sizeof(struct RBTreeNode));
    newNode->value = value;
    newNode->color = color;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Insert node into a RBTree and return the root of the RBTree
Node* insertNode(Node* root, Node* node){
    if(root==NULL){
        return node;
    }
    if((node->value<root->value)&&(root->left==NULL)){
        root->left=node;
        return root;
    }
    if((node->value>root->value)&&(root->right==NULL)){
        root->right=node;
        return root;
    }
    // recursive insertion
    if(node->value<root->value){
        root->left=insertNode(root->left,node);
        return root;
    }
    if(node->value>root->value){
        root->right=insertNode(root->right,node);
        return root;
    }
}

// Check if a Binary-Tree is RBTree (1:true 0:false)
/* Tips:
The root node here is not the actual root node of the Red-Black tree;
the root node for any subtree can be the parameter 'root' for this function.
*/
int checkRBTree(Node* root, int* blackHeight){
    // 1. First, check if the node is empty.
    if(root==NULL){
        *blackHeight=0;
        return 1;
    }
    // 2. Second, Second, if the current root node's color is red at this point,
    // then check if the colors of its two child nodes are also red.
    if(root->color==RED){
        if((root->left!=NULL)&&((root->left)->color==RED)){
            return 0;
        }
        if((root->right!=NULL)&&((root->right)->color==RED)){
            return 0;
```

```
        }
    }
    // 3. Third, recursively check whether the left and right subtrees also satisfy
    // the properties of a Red-Black tree and pass the black height of the left and
    // right subtrees to the calling function using pointer parameters.
    int leftBlackHeight=0;
    int rightBlackHeight=0;
    if(!checkRBTree(root->left,&leftBlackHeight)){
        return 0;
    }
    if(!checkRBTree(root->right,&rightBlackHeight)){
        return 0;
    }
    // Check if the black heights of the left and right subtrees are equal.
    if(leftBlackHeight!=rightBlackHeight){
        return 0;
    }
    // 4. Pass the black height of the current node to the parent node using a pointer.
    if(root->color==BLACK){
        *blackHeight+=leftBlackHeight+1;
    }else{
        *blackHeight=leftBlackHeight;
    }
    return 1;
}


// Free the memory allocated for Red-Black tree nodes
void freeNodeSpace(Node* root){
    if(root==NULL){
        return;
    }
    // Add conditional checks, reduce additional stack overhead, and save memory space
    if(root->left==NULL){
        freeNodeSpace(root->right);
        free(root);
        return;
    }
    // Add conditional checks, reduce additional stack overhead, and save memory space
    if(root->right==NULL){
        freeNodeSpace(root->left);
        free(root);
        return;
    }
    // Recursively release the memory allocated to nodes
    // (first left subtree, then right subtree, and finally the root node).
    freeNodeSpace(root->left);
    freeNodeSpace(root->right);
    free(root);
    return;
```

```
}


```