Is it a red-black tree

author: A Smart student

date: 2020-10

1. Introduction

In computer science, a red–black tree is a kind of self-balancing binary search tree. Each node stores an extra bit representing color, used to ensure that the tree remains approximately balanced during insertions and deletions. See http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Red-black trees are self-balancing so these operations are guaranteed to be O($\log n$), they are useful when we need insertion and deletion relatively frequent. A red-black tree has the following 5 properties:

- (1) Every node is either red or black.
- (2) The root is black.
- (3) Every leaf (NULL) is black.
- (4) If a node is red, then both its children are black.
- (5) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Now, given cases of the preorder traversal sequences of some trees with N (≤ 30) nodes, the goal is to design an algorithm to tell if these trees are legal red-black trees.

2. Algorithm Specification

- **Main data structure** : Binary Search Tree

Definition:
(1) Every node has a key which is an integer, and the keys are distinct.
(2) The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
(3) The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
(4) The left and right subtrees are also binary search trees.

Main Idea: To implement a searching algorithm that finds a item efficiently when the items are totally ordered.

- **Algorithm 1** : Building a binary search tree by inserting

Input: an empty tree node T, integer array x[n]

Output: The root node of a built binary search tree T

Main Idea: The algorithm locates each item x in a binary search tree according to the properties of this kind of tree. If the item x is not present, the algorithm add a new vertex with x as its key. A binary search tree is built after n insertion.

Pseudo Code:

```
v:= root of T
{a vertex not present in T has the value null,the key value of v is label(v)}
for all  x to be inserted
if v is null then add a node r to the tree and label it with x
return r
else
r:=v
```

```
while v is not null and label(v) is not x
begin
    if x < label(v) then
    if left child of v is not null then v:= left child of v
    else add new vertex as a left child of v and label it with x
    else
    if right child of v is not null then v:= right child of v
    else add new vertex as a right child of v and label it with x
end
return r
end
```

- **Algorithm 2**: A Recursive judging algorithm

Input: binary search tree T

Output: True (integer 1) or False (integer 0)

Main Idea: The algorithm judges whether a given tree T is a legal red-black tree , by judging if both of the subtrees of the root of T is a legal red-black tree and if the number of black nodes in all simple path from the root of both subtrees are the same.

Pseudo Code:

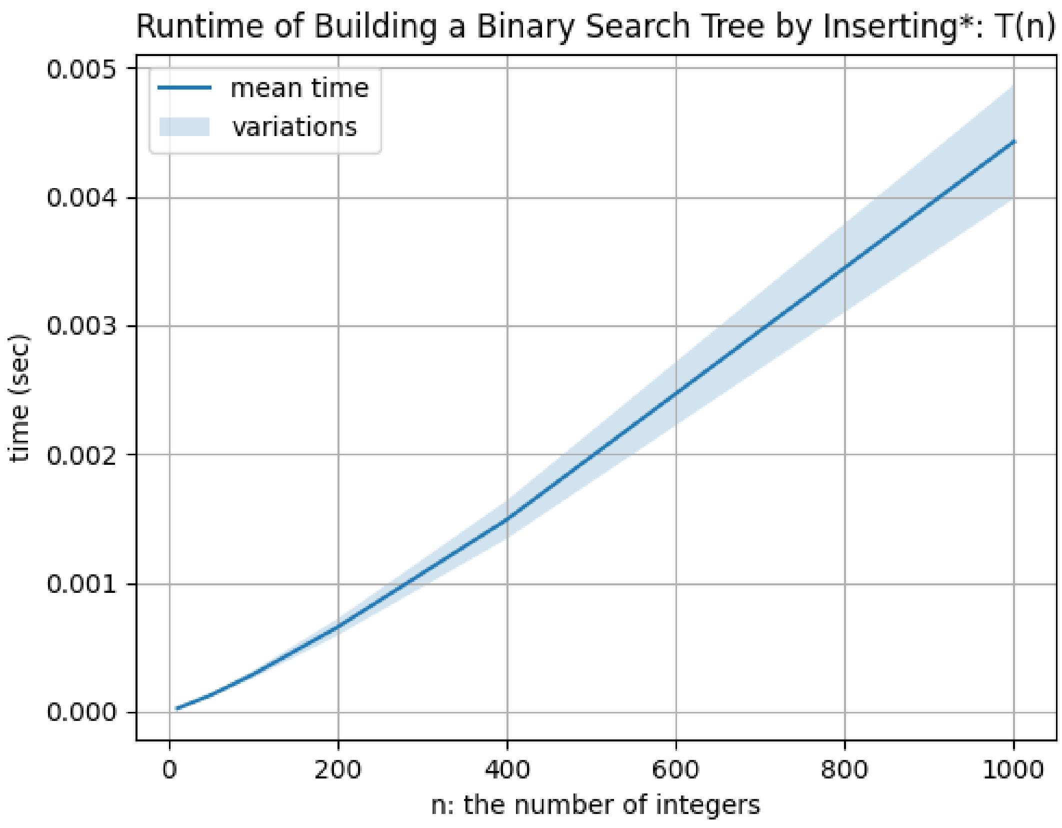
```
Procedure islegalr(v: the root of a tree T)
{a vertex not present in T has the value null.}{v was ensured to be a black node
and not to be null }{the number of black nodes in simple path from the node k to
all its leaves is n(k)}
if both children is null then
if v is a black node then n(v):=1
else n(v):=0
return True
else if one of the children of v is null
if the child that is not null is a black node then return False
else
if islegalr(vp:the child that is not null) is True
if v is a black node then n(v)=n(vp)+1
else n(v)=n(vp)
return True
else return False

if islegalr(vleft:the left child of v) is True and islegalr(vright:the right
child of v) is True and n(vleft) == n(vleft) is True then
if v is a black node then n(v)=n(vleft)+1
else n(v)=n(vleft)
else return False
```

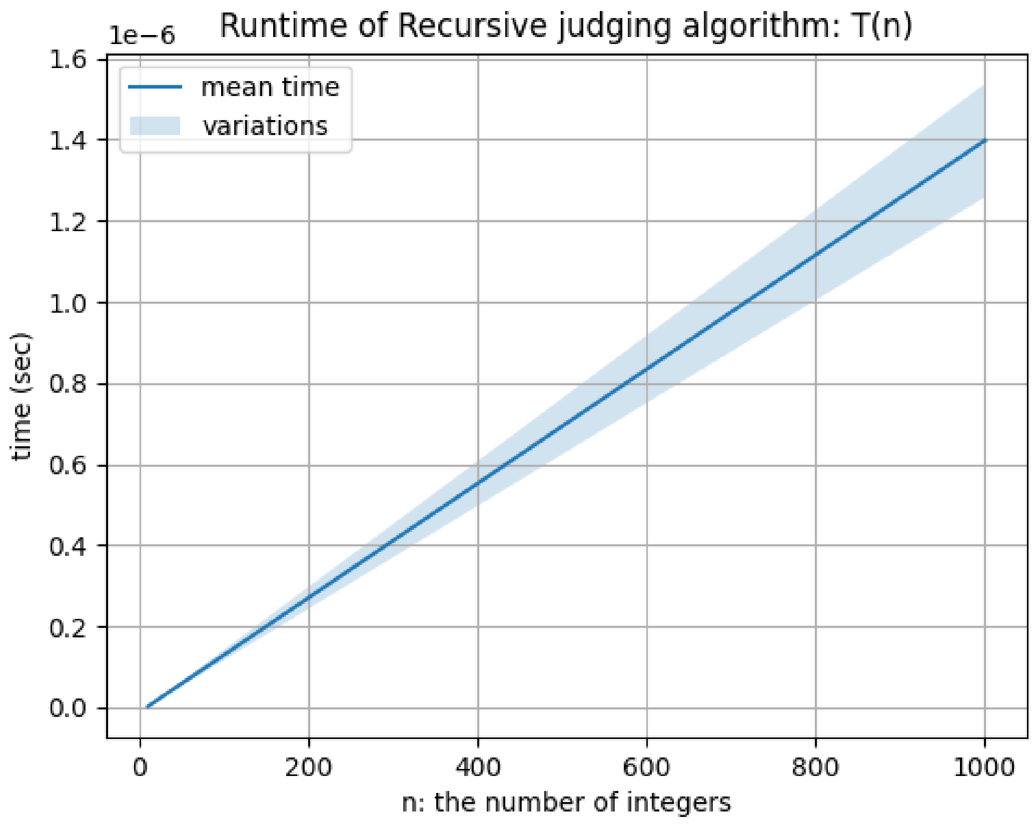
3. Testing Results

Test cases	Design Purpose	expected result	actual behavior	possible cause for a bug	status
[1 0]	An empty tree	[Yes]	[Yes]	/	*pass*
[1 1 -7]	The root is red	[No]	[No]	/	*pass*
[1 1 1]	A tree with only one black node	[Yes]	[Yes]	/	*pass*
[1 9 11 -2 1 -7 5 -4 8 14 -15]	A tree with a red node that one of its children is red	[No]	[No]	/	*pass*
[1 8 10 -7 5 -6 8 15 -11 17]	There exist two simple paths from the root of descendant leaves that contain different number of black nodes	[No]	[No]	/	*pass*
[1 6 7 1 -9 8 10 -12	A red-balck tree whose simple paths from the root of descendant leaves have different length	[Yes]	[Yes]	/	*pass*
[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]	A comprehensive case	[Yes No No]	[Yes No No]	/	*pass*
...	...	...	....	...	...

Table: Test cases for the algorithms implementation.



Figures 1 shows the running time of the *Binary Search Tree Inserting* implementation. We observe a curve between $y = x^2$ and $y = x \log x$, which implies an algorithm with time complexity between $O(n^2)$ and $O(n \log n)$ (see analysis in the next section).



Figures 2 shows the running time of the *Recursive judging algorithm* implementation. We observe a linear-like curve from the figure, which implies an $O(n)$ algorithm (see analysis in the next section).

4. Analysis and Comments

1. Building a binary search tree by inserting

For the runtime of Binary Search Tree Inserting, there is a *while* loop:

- The most comparisons need to add a new item is the length of the longest path in T from the root to a leaf, which is the height of the tree.
- In the full binary tree, there are t nodes with a key and t+1 leaves(null). Suppose height of the tree is h. $\log_2(t+1) \leq h \leq\ t+1$

Therefore, The time complexity of inserting an item in a binary search tree with n nodes(with key) :

$$O(\log t) = \log_2(t+1) \leq T_1(t) \leq= t+1 = O(t)$$

Since there are n insertions, the total time complexity $T(n)$:

$$T(n) = \sum_{k=0}^{n-1} T_1(k) \in [\log_2 n!, \frac{n+n^2}{2}]$$
$$O(n\log n) \leq T(n) \leq O(n^2)$$

For the space requirement, since we merely need n tree nodes to store the n integers, the space complexity is $\Theta(n)$ and should be optimal.

2. A Recursive judging algorithm

For the runtime of the recursive judging algorithm $T(n)$

- First there are several steps of O(1)
- then there are one or two recursive steps with the runtime $T(k)(k < n)$
- in the worst case:

$$T(n) = c + 2 \times T(\frac{n-1}{2}) = c + 2c + 4 \times T(\frac{n-1-2}{4}) = \ldots$$
$$= c + 2c + .. + 2^k c + 2^{k+1} \times T(\frac{n - 2^{k+1} + 1}{2^{k+1}})$$
$$when\ k\ is\ \log_2(n+1) - 2, T(n) = c + 2c + .. + 2^k c + 2^{k+1} \times T(1) = O(2^{k+1}) = O(n)$$

For the space requirement , in the worst case:

$$\Theta(n) = 2 \times \Theta(\frac{n-1}{2}) = 2^2 \times \Theta(\frac{n-1-2}{2^2}) = 2^k \times \Theta(\frac{n-2^{k+1}+1}{2^k})$$
$$when\ k\ is\ \log_2(n+1) - 2,\ \ \Theta(n) = 2^k = O(n)$$

There's still room for improvement. In this algorithm, a tree node has for members: `color (int)`, `Element  (int)`, `Left (Tree)`, `Right(Tree)` and `n(int)`. `n` is used to store the number of black nodes in all simple paths from the node to its descendant leaves. But if we create a global variable K to store the number of black nodes in the first simple path from the node to its descendant leaves, and pass the value of the number of black nodes in each recursive steps in other simple paths to compare with K, we can save the space for the member `n` in the tree node.

Appendix: Source Code (in C)

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

```c
#include <stdlib.h>
#define Black 0
#define Red 1

typedef struct TreeNode *Tree;
struct TreeNode {
    int color;
    int Element;
    int n;     // n stores the number of black nodes in all simple paths from
the node to its descendant leaves
    Tree  Left;
    Tree  Right;
};

Tree addkey(Tree T,int a);
int islegal(Tree T);
int islegalr(Tree T);

main(){
    // get the number of inputs and creat an array of all the roots
    int k;
    scanf("%d",&k);
    int result[k];
    //for each root(the root will be set null first),build a tree using binary
search tree inserting
    for (int i=0;i<k;i++){
        int n;
        scanf("%d",&n);
        Tree Treehead;
        Treehead= NULL;
        for(int j=0;j<n;j++){
            int a;
            scanf("%d",&a);
            Treehead= addkey(Treehead,a);
            if(islegal(Treehead)) result[i]=1;
            else result[i]=0;
        }
    }
    //print the results
    for(int i=0;i<k;i++){
        if(result[i]) printf("Yes\n");
        else printf("No\n");
    }
    return 0;

}
/*
 * binary search tree inserting
 * ------------------------------------------------
 *   for a binary search tree, if a< the key of p, the position of a must be in
the left subtree of p;if a> the key of p, the position of a must be in the right
subtree of p
 *   T: the root of the tree
 *   a: the integer to be inserted
 *   h: a new node for a
 *   p: set a pointer of the root
 *   pb: Store p in pb, before setting p=p->Left or p=p->Right and do the next
round
```

```
 *   while p is not null, do the search for the proper postition of a in the
child of p.
 *   end: p=null,then put h in the position of p

 *   return: T
 */

Tree addkey(Tree T,int a){
    //create a new node for a
    Tree h;
    h=(Tree)malloc(sizeof(struct TreeNode));
    if(a<0){
        h->color = Red;
        h->Element= (-1)*a;
    }
    else{
        h->color = Black;
        h->Element= a;
    }
    h->Left= NULL;
    h->Right=NULL;
    h->n=0;
    //if T is null, let h be the root of the tree
    if(T == NULL) return h;
    else{
        Tree p;Tree pb;
        pb=T;p=T;
        while(p!= NULL){
            if(h->Element<p->Element){
                pb=p;
                p=p->Left;
                if(p == NULL){pb->Left=h;break;}  //pb is the father of p when p
is null
            }
            else{
                pb=p;
                p=p->Right;
                if(p == NULL){pb->Right=h;break;}
            }
        }
        return T;
    }
}

 // judging if T is an empty tree or if the root of T is black

int islegal(Tree T){

    if(T == NULL) return 1;
    else if(T->color == Red) return 0;
    else
        return islegalr(T);
}

/*
 *  Recursive judging
 * ------------------------------------------------
 *   T: the root of the tree
 *   While one or two children of T is presnt, iterate to judge if both of the
subtrees of the root of T is a legal red-black tree and if the number of black
nodes in all simple path from the root of both subtrees are the same.
 *   T->n = (the present child of T)->n+1 if T is a black node
 *   return 1 when T is a legal red-black tree ; return 0 when T is not a legal
red-black tree
 */
int islegalr(Tree T){
    // case1 when both children is not present
    if (T->Left == NULL && T->Right == NULL) {
        T->n = ((T->color == Black)?1:0);  //T->n add 1 if T is a black node
        return 1;
    }
    //case2 when one child is present and the other is not
    else if(T->Left == NULL||T->Right == NULL){
        Tree tem;
        tem= (T->Left == NULL? T->Right : T->Left);
        //case 2.1 when the child that is present is black
        if(tem->color == Black) return 0;
        //case 2.2 when the child that is present is red
        else {
            if (islegalr(tem)){
                if(T->color == Black) T->n=tem->n+1;
                else T->n=tem->n;
                return 1;
            }
            else return 0;
        }
    }
    //case 3 when both children is present
    //judge if both children of a red node is black
    if(T->color == Red){
        if(T->Left->color == Red || T->Right->color == Red)
            return 0;
    }
    if(islegalr(T->Left) && islegalr(T->Right)){      //judge if both subtrees
is a red-black tree
        if (T->Left->n == T->Right->n){               //judge if the number of
black nodes in all simple paths from both children to thier descendant leaves is
the same
            if(T->color == Black) T->n=T->Left->n+1;
            else T->n=T->Left->n;
            return 1;
        }
        else return 0;
    }
    else return 0;
}
```

Declaration

I hereby declare that all the work done in this project is of my independent effort.