حوری دهش 9821413

به نام خدا

تمرین تئوری سری سوم ساختمان داده

==========

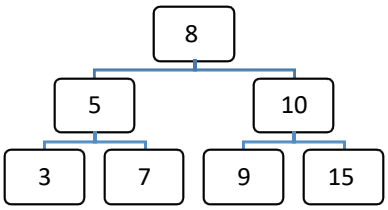
سوال 1:

الف) غلط است چون در یک درخت دودویی حداقل ارتفاع logn می باشد یعنی درخت ما یک درخت دودویی کامل است و حداکثر ارتفاع n-1 می باشد یعنی در درخت ما همه نودها در یک طرف قرار گرفته است.

ب) غلط است به درختی، درخت دودویی کامل می گویند که هر گره ان دقیقا دو فرزند داشته باشد نه حداکثر دو فرزند.

ج) غلط است --> مثال نقض :

در درخت روبه رو ما 4 تا برگ و 7 تا گره داریم -->



در صورتی که سوال می گوید ما $(x+1)^2$ تا گره داریم یعنی ما 25 تا گره داریم که این غلط می باشد.

د) درست است چون در ابتدای کار می توانیم ان نودی که از همه ارتفاعش بیشتر است را انتخاب کنیم و ان نود، نود ریشه ما میشود سپس نود ریشه را در پیمایش inorder پیدا می کنیم و هر نودی که قبل از نود ریشه، در پیمایش inorder باشد در زیر درخت سمت چپ ریشه قرار می گیرد و هر نودی که بعد از نود ریشه، در پیمایش inorder باشد در زیر درخت سمت راست ریشه قرار می گیرد و به صورت بازگشتی دوباره برای هر کدام از زیر درخت ها با کمک ارتفاعشون و همین پیمایش inorder جلو می رویم تا درختمان کامل درست شود که در این حالت ما یک درخت یکتا داریم.

ه) درست است چون هرم یک درخت k تایی کامل است و ارتفاع یک درخت k تایی کامل برابر با logn می باشد.

سوال 2 :

زمانی ما ارتفاع کمینه داریم که درخت ما کامل باشد.

ارتفاع کمینه --> $\lfloor \log n \rfloor = \lfloor \log 25 \rfloor = \lfloor 4.6 \rfloor = 4$ پس تا سطح چهارم درخت ما کامل است و نودهای باقی مانده در سطح پنجم قرار دارند --> این سوال حالت های مختلفی دارد که در انتها جمع این حالت ها می شود تعداد درخت های دودویی ما :

حالت اول این است که ما می دانیم تا سطح چهارم، درخت ما کامل است که برای این کار 15 تا نود مصرف شده است برای اینکه درخت ما کامل شود پس از 25 تا نود 10 نود باقی می ماند که این 10 تا نود را می توانیم در 16 تا جایگاه قرار دهیم یعنی $\binom{16}{10} = 8008$

حالت دوم این است که ما بیایم یک نود را از سطح چهارم حذف کنیم که در این حالت در سطح چهارم 7 تا نود داریم که این 7 تا نود را می توانیم در 8 جایگاه قرار دهیم یعنی $\binom{8}{7} = 8$ پس الان در سطح پنجم ما 11 تا نود داریم که این 11 تا نود را می توانیم در 14 تا جایگاه قرار دهیم یعنی $\binom{14}{11} = 364$ پس کلا حالت دوم می شود $8 * 364 = 2912$

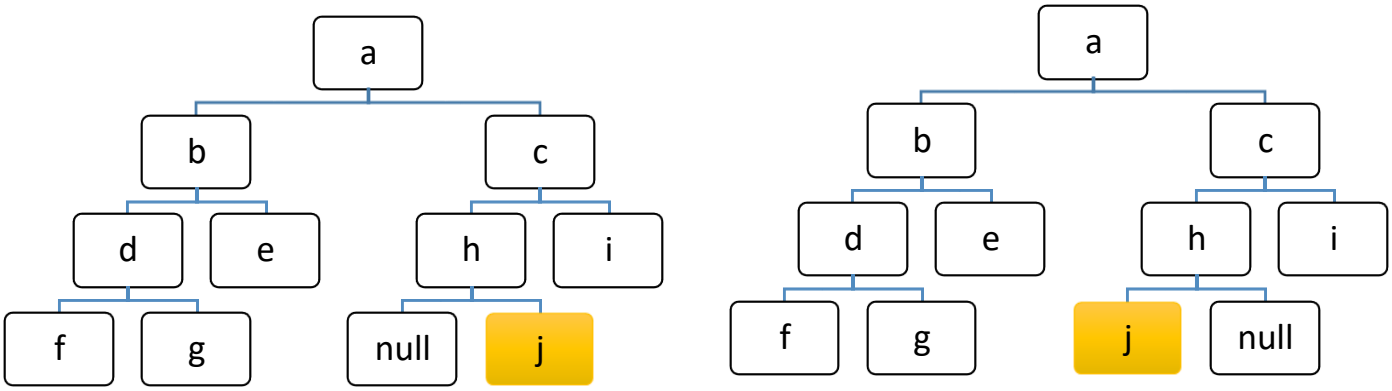
حالت سوم این است که ما بیایم دو نود را از سطح چهارم حذف کنیم که در این حالت در سطح چهارم 6 تا نود داریم که این 6 تا نود را می توانیم در 8 جایگاه قرار دهیم یعنی $\binom{8}{6} = 28$ پس الان در سطح پنجم ما 12 تا نود داریم که این 12 تا نود را می توانیم در 12 تا جایگاه قرار دهیم یعنی $\binom{12}{12} = 1$ پس کلا حالت سوم می شود $28 * 1 = 28$ که در این مرحله سطح پنجم کاملا پر می شود.

جمع همه حالت های بالا : $10948 = 8008 + 2912 + 28$ پس ما 10948 تا درخت دودویی داریم.

سوال 3 :

الف)

توجه : نودی که null فرض شده است به این خاطر است که نشان دهم کدام نود فرزند سمت راست یا فرزند سمت چپ است در غیر اینصورت همشون شبیه ستون قرار می گرفتن !!!



هر دو درخت بالا می تواند باشد.

مراحل میانوندی: اولین عنصر در preorder و اخرین عنصر در postorder ریشه ما را معلوم می کند که در این حالت می شود a سپس در پیمایش preorder ما عنصر بعد از a، ریشه زیر درخت سمت چپ می شود که در این حالت b است سپس در پیمایش postorder عنصر b را معلوم می کنیم و هر چی قبل از b در پیمایش postorder باشد متعلق به زیر درخت سمت چپ و راست ریشه b است و هر چی بعد از عنصر b در پیمایش postorder قرار گیرد تا قبل از ریشه a مربوط به زیر درخت سمت راست ریشه a می شود و در پیمایش postorder عنصری که قبل از ریشه a قرار گرفته باشد ریشه زیر درخت سمت راست ما می شود یعنی c و در پیمایش preorder هر چی بعد از عنصر c قرار گرفته باشد متعلق به زیردرخت سمت چپ و راست ریشه c است حالا برای ریشه های b , c هم به صورت بازگشتی این کار را تکرار می کنیم تا درخت میانوندی ما درست شود.

ب) خیر چون ما نمی توانیم از postorder , preorder یک درخت یکتا درست کنیم چون اگر در درخت preorder یک گره فقط یک فرزند داشته باشد و در درخت postorder هم همین طور باشد در این حالت ما اطلاعات کافیه نداریم که این فرزند چپ ان گره است یا راست ان گره مانند همین سوال برای نود h

مثلا این دو درخت یکسان نیستند ولی پیمایش postorder , preorder هر دو یکی است <--

پس ما تنها از postorder , inorder یا inorder , preorder می توانیم یک درخت یکتا بسازیم.

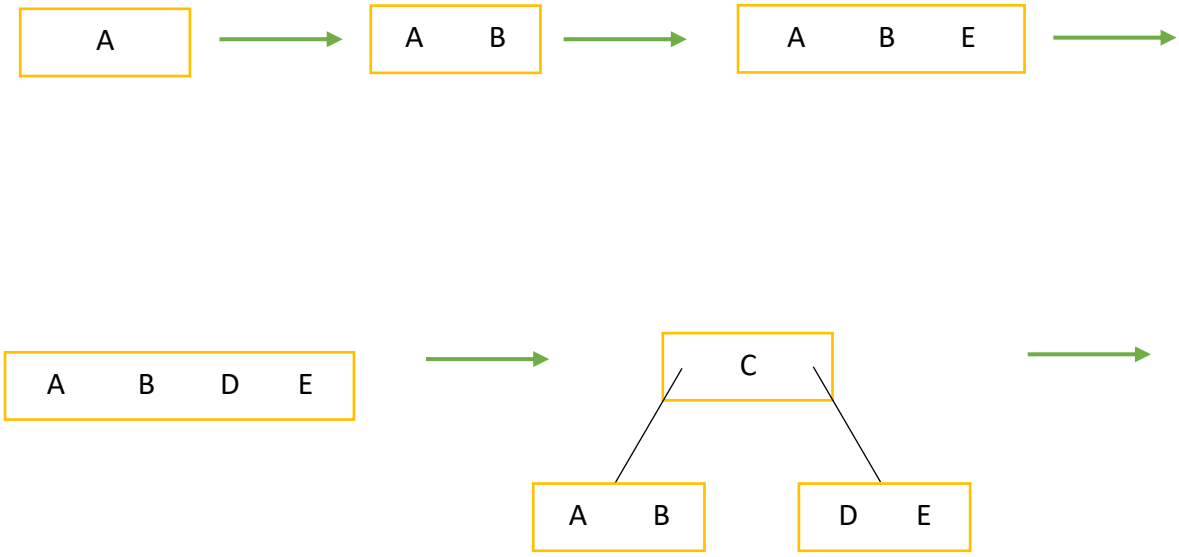
سوال 4 : A B E D C G F J H I ؟؟

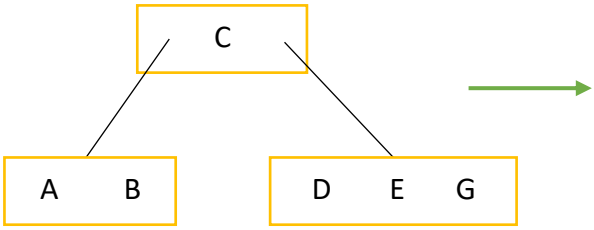
الف) AVL : **در صفحه اخر نوشته شده است!!! <-- از صفحه 18 تا اخر را شامل میشود**

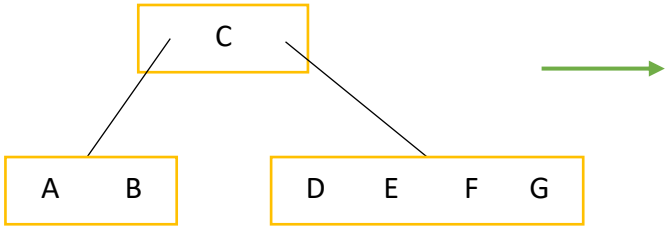
ب) Red-Black : **در صفحه اخر نوشته شده است!!! <-- از صفحه 18 تا اخر را شامل میشود**

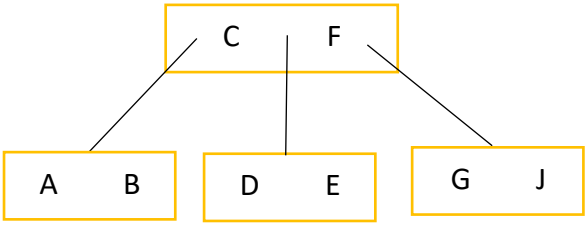
ج) B-tree & degree=3 :

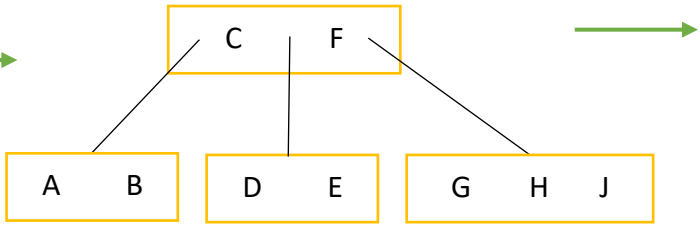
چون degree=3 است یعنی t=3 است پس حداکثر کلیدهای ما 2t-1=5 می تواند باشد.

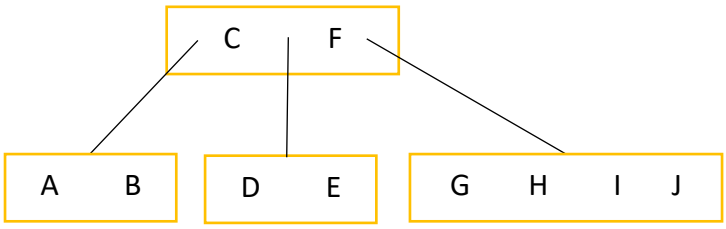




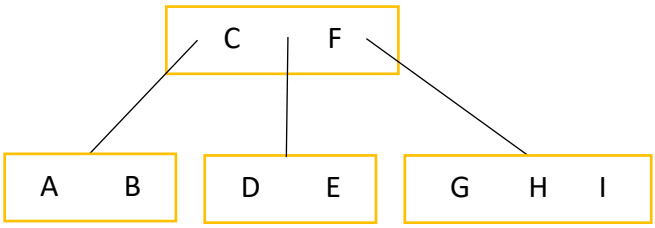




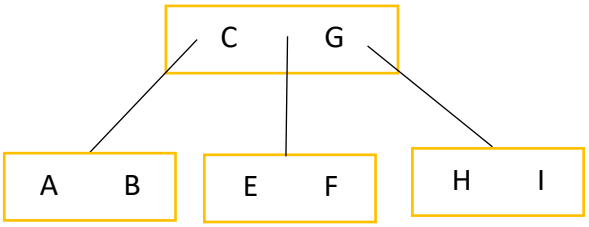




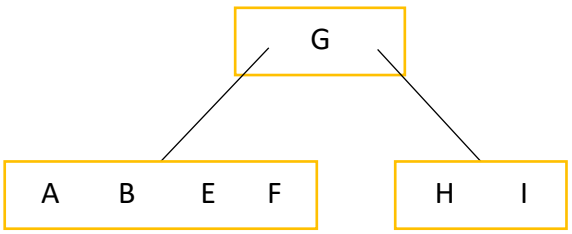
Delete J →



Delete D →



Delete C →



سوال 5 :

الف)

زمانی کمترین تعداد برگ را داریم که درخت ما مورب سمت راست یا مورب سمت چپ باشد یعنی همه نودها یک طرف باشند در این حالت یک برگ داریم.

و زمانی بیشترین تعداد برگ را داریم که درخت ما درخت دودویی کامل باشد در این حالت $\left\lceil \frac{n}{2} \right\rceil$ برگ داریم.

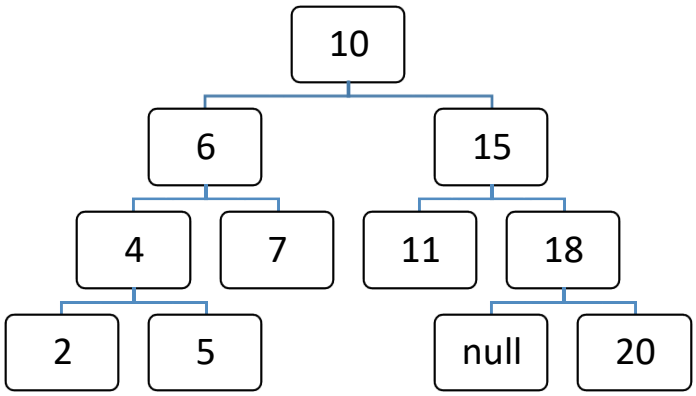
ب) با 1000 نود کمترین برگی که داریم یک است و بیشترین برگی که داریم $\left\lceil \frac{1000}{2} \right\rceil = 500$ است.

ج) با n تا نود کمترین برگی که داریم یک است و بیشترین برگی که داریم $\left\lceil \frac{n}{2} \right\rceil$ است.

حوری دهش 9821413

سوال 6 :

به فرض مثال ما همچین درختی داریم :



برای حل این سوال به صورت بازگشتی عمل می کنیم و در ابتدای کار هم از برگ شروع می کنیم به حرکت کردن و مجموع مقادیر را روی قطر به دست می اوریم به این صورت که :

اگر زیر درخت سمت راست = R و زیر درخت سمت چپ = L در نظر بگیریم، در هر مرحله ریشه را با L , R جمع می کنیم و به سمت بالا می رویم --> یعنی برای نود 4 ما داریم L = 2 , R = 5 پس 5+4+2=11 حالا سراغ نود 6 می رویم و از بین دو زیر درخت چپ و راست نود 4 بزرگترین مسیر یعنی قطر را در نظر می گیریم پس 5+4 را در نظر می گیریم حالا برای نود 6 داریم 6+7+9=22

این کار را برای زیر درخت سمت راست نود 10 هم انجام می دهیم به این صورت که برای نود 18 قطر ما می شود 20+18=38 و برای نود 15 هم قطر ما می شود 15+11+38=64

در اخر برای نود 10 هم از بین دو زیر درخت چپ و راست قطر ها را انتخاب می کنیم و با ریشه جمع می زنیم یعنی برای زیر درخت سمت راست قطر ما می شود 15+38=53 و برای زیر درخت سمت چپ قطر ما می شود 6+9=15 و در اخر مقدار کل ما روی قطر نهایی یا طولانی ترین مسیر می شود 10+15+38=78

اردر این الگوریتم از O(n) می باشد.

عضو میانی این مسیر مرکز قطر ما می شود که همان مرکز درخت است و ویژگی این مرکز درخت این است که بیشینه ی فاصله اش از سایر رئوس کمینه باشد --> مرکز درخت هم به این صورت به دست می اوریم که در هر مرحله برگهای درخت را جدا می کنیم و این کار را انقدر انجام می دهیم تا در نهایت به مرکز درخت برسیم حالا ممکن است درخت ما یک مرکز داشته باشد یا دو مرکز. در صورتی یک مرکز دارد که طول قطر ما عدد فردی شده باشد که به این درخت، درخت مرکزی می گویند ولی اگر طول قطر ما عدد زوجی شده باشد ما دو مرکز داریم که به این درخت، درخت دو مرکزی می گویند.

سوال 7 :

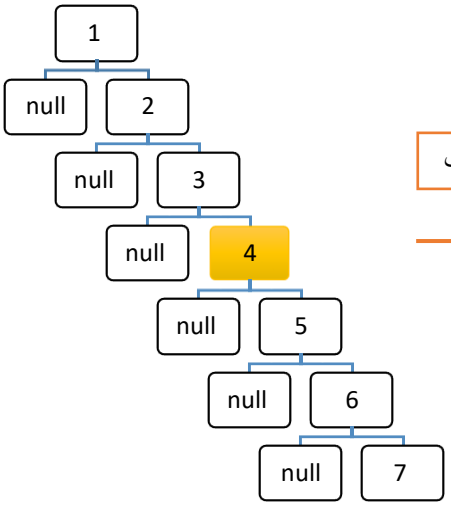
فرض می کنیم 7 نود به صورت 1 , 2 , 3 , 4 , 5 , 6 , 7 داریم :

توجه : نودی که null فرض شده است به این خاطر است که نشان دهم کدام نود فرزند سمت راست یا فرزند سمت چپ است در غیر اینصورت همشون شبیه ستون قرار می گرفتن !!!

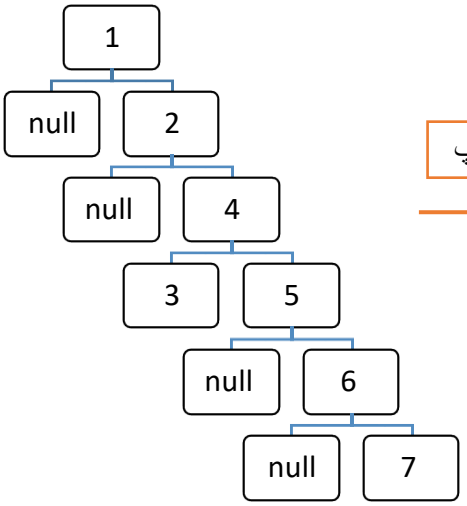
برای اینکه بتوانیم یک درخت مورب را به درخت کامل تبدیل کنیم فقط کافی است نود میانی را پیدا کنیم و ان را به ریشه برسانیم و از انجایی که درخت کامل $2^n - 1$ نود دارد پس حتما نود میانی ما فقط یک گره می شود چون تعداد نودهای ما فرد است.

زمانی که نود میانی را پیدا کردیم فقط کافی است ان را به ریشه برسانیم که برای این کار هم می توانیم روی هر نودی که قبل از نودی میانی است یک دوران چپ انجام دهیم و برای بقیه زیردرخت ها هم باز به همین صورت عمل میکنیم که به درخت کامل برسیم پس مسئله را به صورت بازگشتی حل می کنیم.

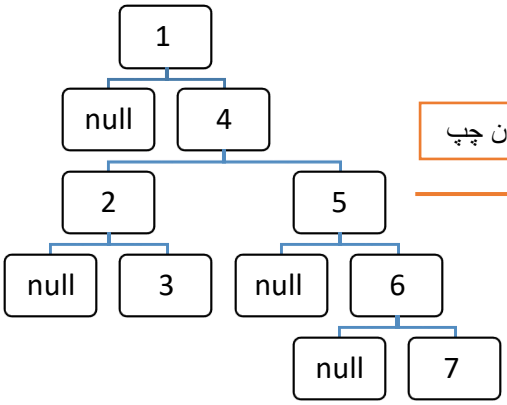
در این سوال نود میانی ما 4 می باشد پس برای نودهای قبل از 4 دوران چپ را انجام می دهیم و بعد از ان برای هر کدام از زیر درخت ها هم باز به همین صورت عمل میکنیم.



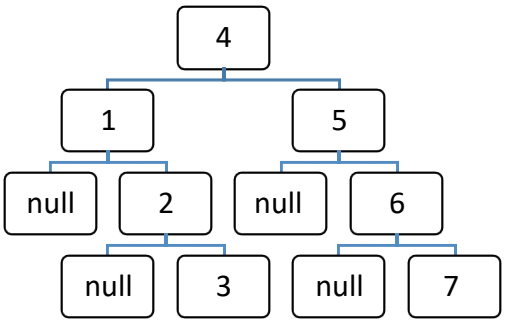
دوران چپ



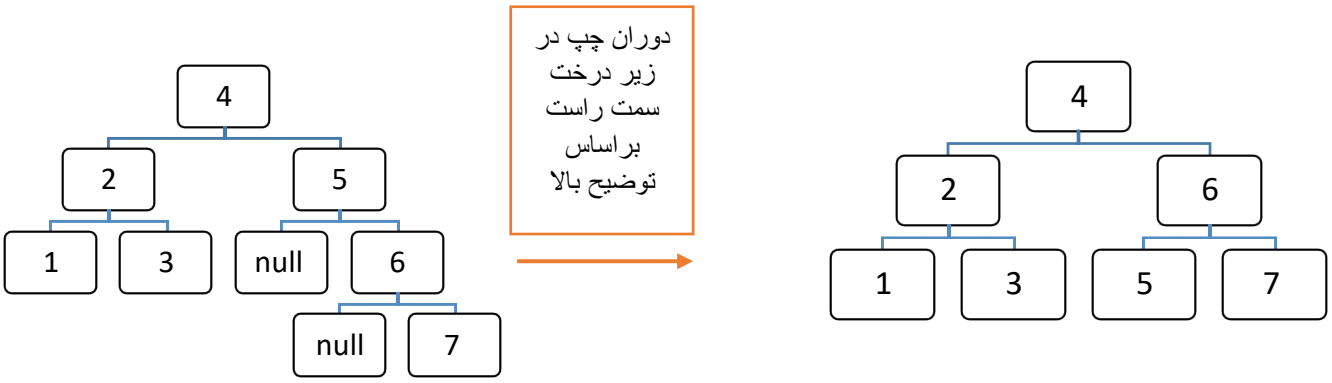
دوران چپ



دوران چپ



دوران چپ در زیر درخت سمت چپ براساس توضیح بالا



سوال 8 :

Time complexity: O(n)

برای حل این سوال به صورت بازگشتی عمل می کنیم.

در ابتدای کار به صورت بازگشتی طولانی ترین مسیر را از زیردرخت سمت چپ و راست پیدا می کنیم سپس نود فعلی را به زیر درختی که بیشترین طول را دارد اضافه می کنیم و این طولانی ترین مسیر از نود فعلی تا برگ می شود.

با شروع از ریشه مراحل زیر را به صورت بازگشتی برای هر نود انجام می دهیم:

اگر ریشه ما null بود هیچ مسیری وجود ندارد و یک وکتور خالی برمی گردانیم.

بزرگترین مسیر را از زیر درخت سمت راست در vector<int> rightvect به صورت پیمایش بازگشتی قرار می دهیم root->right

بزرگترین مسیر را از زیر درخت سمت چپ در vector<int> leftvect به صورت پیمایش بازگشتی قرار می دهیم root->left

سپس طول vector<int> rightvect با طول vector<int> leftvect را مقایسه می کنیم و نود فعلی را به ان وکتوری که بزرگترین است اضافه می کنیم و ان وکتور را برمی گردانیم.

در انتهای این پیمایش ما طولانی ترین مسیر ممکن را داریم پس فقط کافی است وکتور را به صورت معکوس چاپ کنیم تا طولانی ترین مسیر از ریشه تا برگ را به دست اوریم.

Code:

```
using namespace std;

// Tree node Structure
struct Node {
    int data;
    Node *left, *right;
};

struct Node* newNode(int data)
{
    struct Node* node = new Node;
```

```
    node->data = data;
    node->left = node->right = NULL;

    return (node);
}

// Function to find and return the
// longest path
vector<int> longestPath(Node* root)
{

    // If root is null means there
    // is no binary tree so
    // return a empty vector
    if (root == NULL) {
        vector<int> temp = {};
        return temp;
    }

    // Recursive call on root->right
    vector<int> rightvect = longestPath(root->right);

    // Recursive call on root->left
    vector<int> leftvect = longestPath(root->left);

    // Compare the size of the two vectors
    // and insert current node accordingly
    if (leftvect.size() > rightvect.size())
        leftvect.push_back(root->data);

    else
        rightvect.push_back(root->data);

    // Return the appropriate vector
    return (leftvect.size() > rightvect.size() ? leftvect : rightvect);
}
```

سوال 9 :

پیمایش inorder غیر بازگشتی و بدون پشته :

با استفاده از الگوریتم morris می توانیم درختی به صورت غیر بازگشتی و بدون پشته درست کنیم به این صورت که مبنای الگوریتم morris براساس Threaded Binary Tree است.

در ابتدای پیمایش ما ابتدا یک لینک به successor غیر ترتیبی می سازیم و با استفاده از این لینک ها می توانیم داده ها را چاپ کنیم و در اخر تغییرات را به حالت درخت اولیه برمی گردانیم.

نکته : ما پیمایش inorder غیر بازگشتی با پشته هم داریم ولی چون پیمایش بدون پشته هیچ فضای اضافه ای برای پیمایش در نظر نمی گیرد روش بهتری است نسبت به پیمایش با پشته.

Code:

```
/* A binary tree tNode has data, a pointer to left child and a pointer to right child */
struct tNode {
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Function to traverse the binary tree without recursion and without stack */
void MorrisTraversal(struct tNode* root)
{
    struct tNode *current, *pre;

    if (root == NULL)
        return;

    current = root;
    while (current != NULL) {

        if (current->left == NULL) {
            printf("%d ", current->data);
            current = current->right;
        }
        else {

            /* Find the inorder predecessor of current */
            pre = current->left;
            while (pre->right != NULL && pre->right != current)
                pre = pre->right;

            /* Make current as the right child of its inorder predecessor */
            if (pre->right == NULL) {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in the 'if' part to restore the original tree i.e., fix the
right child of predecessor */
            else {
                pre->right = NULL;
                printf("%d ", current->data);
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the given data and NULL left and right
pointers. */
```

```
struct tNode* newtNode(int data)
{
   struct tNode* node = new tNode;
   node->data = data;
   node->left = NULL;
   node->right = NULL;

   return (node);
}
```

سوال 10 :

طبق توضیحات سوال 7 ما این سوال را به صورت بازگشتی حل کردیم :

مثلا :

0 دوران	1 نود
1 دوران	3 نود
5 دوران	7 نود
17 دوران	15 نود

پس می توانیم این رابطه را به صورت بازگشتی برای تعداد دوران ها به این صورت بنویسیم :

$$T(n) = \left\lfloor \frac{n}{2} \right\rfloor + 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right)$$

علت این کار هم همان روشی است که در سوال 7 گفته شد.

سوال 11:

سوال 12:

نکته اول برای این سوال، اینه که باید در مرحله اول درخت دودویی را به درخت جستجو دودویی تبدیل کنیم چون درخت red-black یک درخت جستجو دودویی است و برای اینکه متوجه بشیم یک درخت red-black است یا نه باید در ابتدا ان درخت، درخت جستجو دودویی باشد.

اکنون با توجه به ویژگی های درخت red-black ما می دانیم که ارتفاع یک گره حداکثر دو برابر حداقل ارتفاع است پس برای هر درخت جستجو دودویی کافی است ببینیم این ویژگی زیر را دارد یا نه :

تعداد گره های مشاهده شده در طولانی ترین مسیر از هر گره تا یکی از برگ ها بیشتر از دوبرابر تعداد گره های مشاهده شده در کوتاه ترین مسیر از آن گره تا یکی از برگ ها نباشد.
پس ایده این است که چک کنیم ببینیم درخت جستجوی دودویی مان متعادل است یا نه اگر متعادل بود درخت ما یک درخت red-black است پس برای هر گره باید حداکثر و حداقل ارتفاع را به دست اوریم و با هم مقایسه کنیم که برای این کار درخت را طی می کنیم و چک می کنیم که هر گره متعادل است یا نه.

در نتیجه می توانیم یک تابع بازگشتی بنویسیم که سه چیز را برمی گرداند :

1- مقدار بولی برای نشان دادن متعادل بودن یا نبودن درخت

2- حداقل ارتفاع

3- حداکثر ارتفاع

Time complexity: O(n)

Code:

```
using namespace std;

struct Node
{
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node with the given key  */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// Returns returns tree if the Binary tree is balanced like a Red-Black
// tree. This function also sets value in maxh and minh (passed by
// reference). maxh and minh are set as maximum and minimum heights of root.
bool isBalancedUtil(Node *root, int &maxh, int &minh)
{
    // Base case
    if (root == NULL)
    {
        maxh = minh = 0;
        return true;
    }

    int lmxh, lmnh; // To store max and min heights of left subtree
    int rmxh, rmnh; // To store max and min heights of right subtree

    // Check if left subtree is balanced, also set lmxh and lmnh
    if (isBalancedUtil(root->left, lmxh, lmnh) == false)
        return false;

    // Check if right subtree is balanced, also set rmxh and rmnh
    if (isBalancedUtil(root->right, rmxh, rmnh) == false)
        return false;

    // Set the max and min heights of this node for the parent call
    maxh = max(lmxh, rmxh) + 1;
    minh = min(lmnh, rmnh) + 1;

    // See if this node is balanced
```

```
    if (maxh <= 2*minh)
        return true;
    return false;
}

// A wrapper over isBalancedUtil()
bool isBalanced(Node *root)
{
    int maxh, minh;
    return isBalancedUtil(root, maxh, minh);
}
```

سوال 13:

فرض می کنیم که ما یک مقدار صحیح K و یک درخت جستجوی دودویی یا BST را به اقای چنگ می دهیم و اون به ما دوتا درخت جستجوی دودویی می دهد که این دو درخت جستجوی دودویی، باید متعادل باشند به طوری که BST 1 شامل تمام گره هایی است که کمتر از K هستند و BST 2 شامل تمام گره هایی است که بزرگتر از K یا برابر با K هستند.

نکته : آرایش گره ها ممکن است هر چیزی باشد اما هر دو BST باید متعادل باشند.

برای این کار ابتداد پیمایش inorder درخت جستجو دودویی داده شده را در یک آرایه ذخیره می کنیم.

سپس این آرایه را با توجه به مقدار K به دو قسمت تقسیم می کنیم.

در نهایت دو آرایه حاصل شده را به دو درخت BST تبدیل می کنیم.

Time complexity: O(n)

Code:

```
using namespace std;

// Structure of each node of BST
struct node {
    int key;
    struct node *left, *right;
};

// A utility function to
// create a new BST node
node* newNode(int item)
{
    node* temp = new node();
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to insert a new
// node with given key in BST
struct node* insert(struct node* node,int key)
```

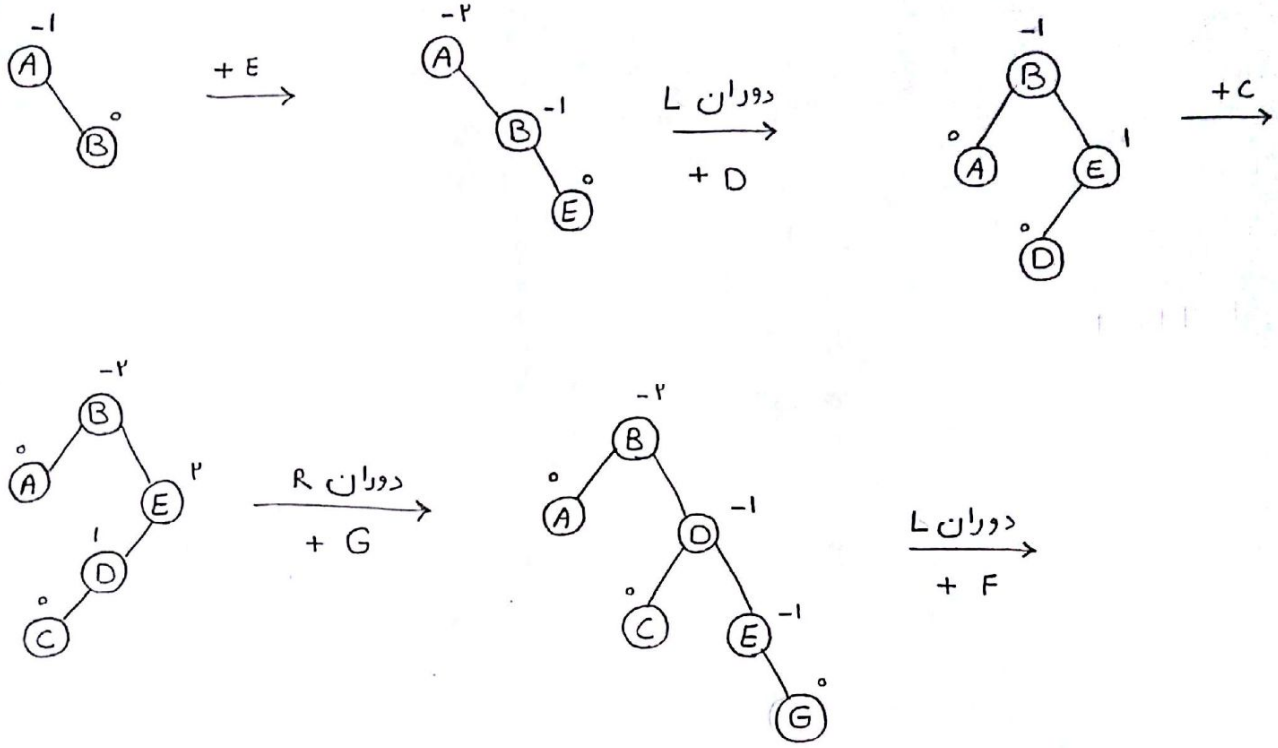
```
{
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
        node->left = insert(node->left,key);
    else if (key > node->key)
        node->right = insert(node->right,key);

    // return the (unchanged) node pointer
    return node;
}

// Function to return the size
// of the tree
int sizeOfTree(node* root)
{
    if (root == NULL) {
        return 0;
    }

    // Calculate left size recursively
    int left = sizeOfTree(root->left);

    // Calculate right size recursively
    int right = sizeOfTree(root->right);

    // Return total size recursively
    return (left + right + 1);
}

// Function to store inorder
// traversal of BST
void storeInorder(node* root,int inOrder[],int& index)
{
    // Base condition
    if (root == NULL) {
        return;
    }

    // Left recursive call
    storeInorder(root->left,inOrder, index);

    // Store elements in inorder array
    inOrder[index++] = root->key;

    // Right recursive call
    storeInorder(root->right,inOrder, index);
}
```

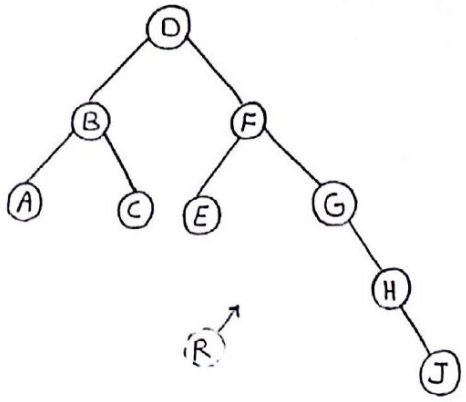
```
// Function to return the splitting
// index of the array
int getSplittingIndex(int inOrder[],int index, int k)
{
    for (int i = 0; i < index; i++) {
        if (inOrder[i] >= k) {
            return i - 1;
        }
    }
    return index - 1;
}

// Function to create the Balanced
// Binary search tree
node* createBST(int inOrder[],int start, int end)
{
    // Base Condition
    if (start > end) {
        return NULL;
    }

    // Calculate the mid of the array
    int mid = (start + end) / 2;
    node* t = newNode(inOrder[mid]);

    // Recursive call for left child
    t->left = createBST(inOrder,start, mid - 1);

    // Recursive call for right child
    t->right = createBST(inOrder,mid + 1, end);

    // Return newly created Balanced
    // Binary Search Tree
    return t;
}

// Function to traverse the tree
// in inorder fashion
void inorderTrav(node* root)
{
    if (root == NULL)
        return;
    inorderTrav(root->left);
    cout << root->key << " ";
    inorderTrav(root->right);
}

// Function to split the BST
// into two Balanced BST
void splitBST(node* root, int k)
```

```
{

    // Print the original BST
    cout << "Original BST : ";
    if (root != NULL) {
        inorderTrav(root);
    }
    else {
        cout << "NULL";
    }
    cout << endl;

    // Store the size of BST1
    int numNode = sizeOfTree(root);

    // Take auxiliary array for storing
    // The inorder traversal of BST1
    int inOrder[numNode + 1];
    int index = 0;

    // Function call for storing
    // inorder traversal of BST1
    storeInorder(root, inOrder, index);

    // Function call for getting
    // splitting index
    int splitIndex= getSplittingIndex(inOrder,index, k);

    node* root1 = NULL;
    node* root2 = NULL;

    // Creation of first Balanced
    // Binary Search Tree
    if (splitIndex != -1)
        root1 = createBST(inOrder, 0,splitIndex);

    // Creation of Second Balanced
    // Binary Search Tree
    if (splitIndex != (index - 1))
        root2 = createBST(inOrder,splitIndex + 1,index - 1);

    // Print two Balanced BSTs
    cout << "First BST : ";
    if (root1 != NULL) {
        inorderTrav(root1);
    }
    else {
        cout << "NULL";
    }
    cout << endl;
```

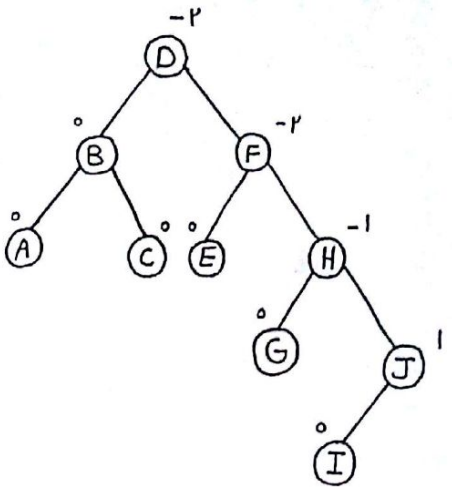
```
    cout << "Second BST : ";
    if (root2 != NULL) {
        inorderTrav(root2);
    }
    else {
        cout << "NULL";
    }
}
```

الف) AVL

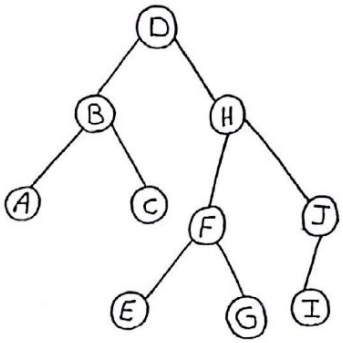

-۱
A
B
+ E
-۲
A
-۱
B
E
دوران L
+ D
-۱
B
A
E
+ C
-۲
B
A
E
۲
۱
D
C
دوران R
+ G
-۲
B
A
-۱
D
C
E
-۱
G
دوران L
+ F
-۱
D
B
E
-۲
A
C
G
۱
F
دوران RL
+ J
D
B
E
A
C
F
G
R
(L)
-۱
D
B
-۱
F
A
C
E
G
-۱
J
+ H
-۲
D
B
-۲
F
A
C
E
G
-۲
J
۱
H
دوران RL
+ I



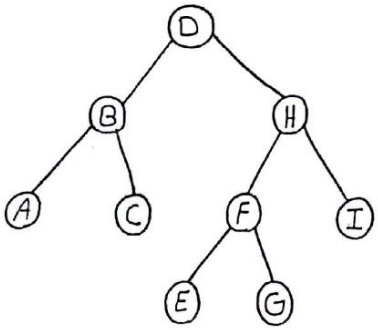
L



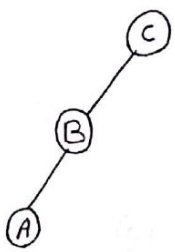
دوران L



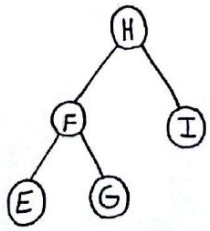
Delete J →



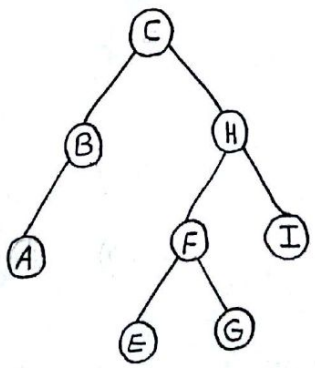
Delete D →



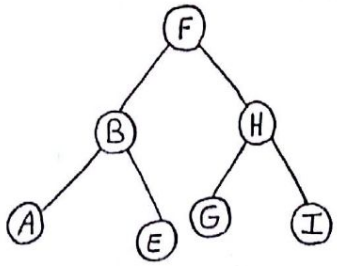
+



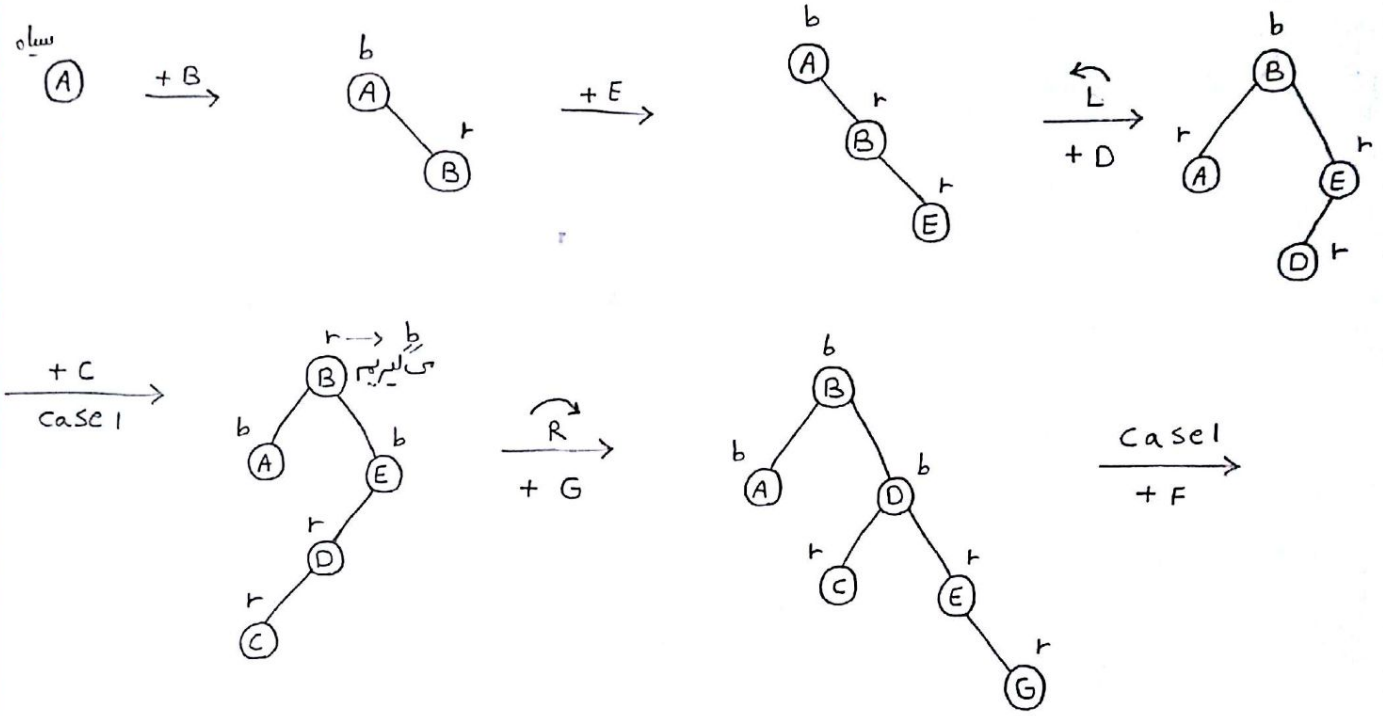
→

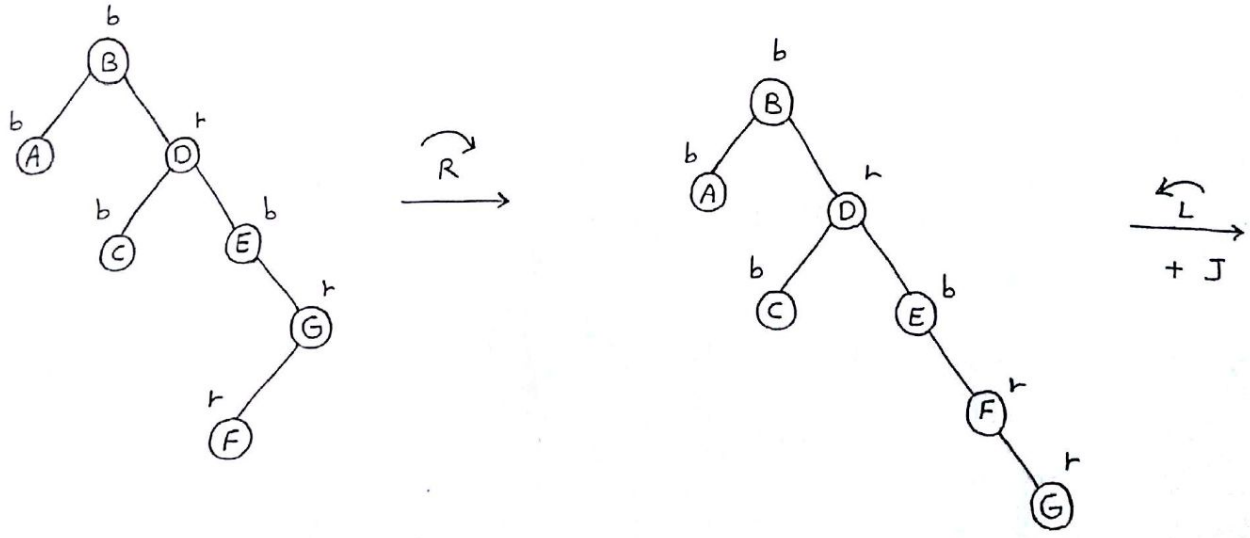


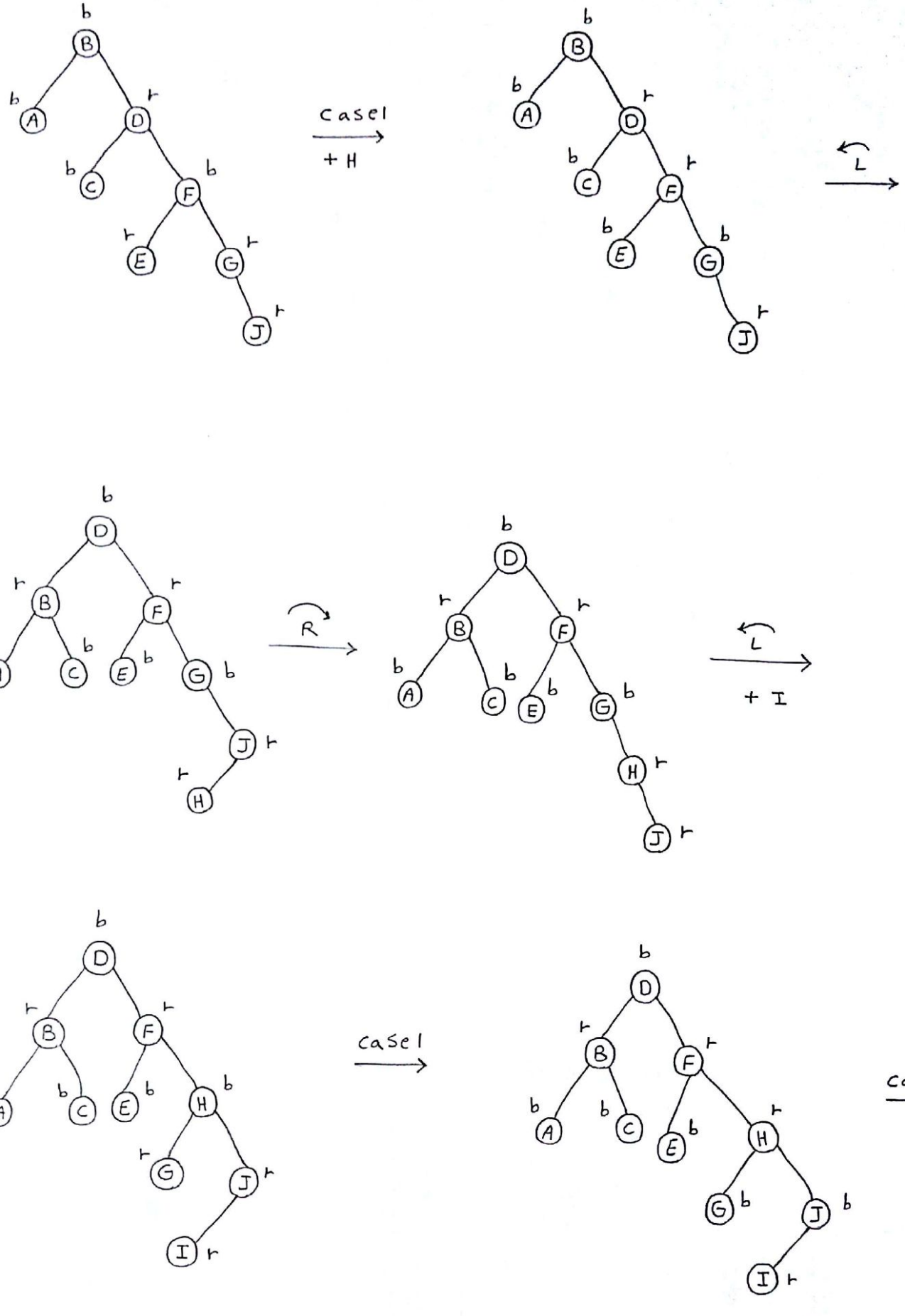
Delete c →

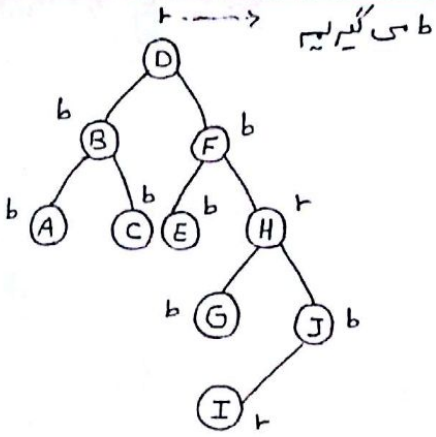


ب) red-black

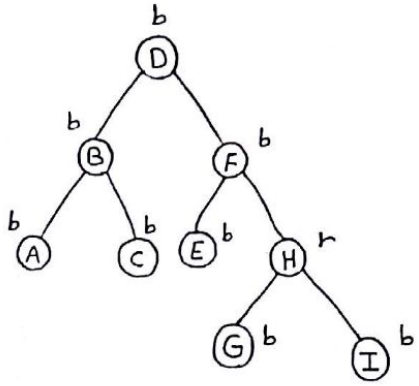




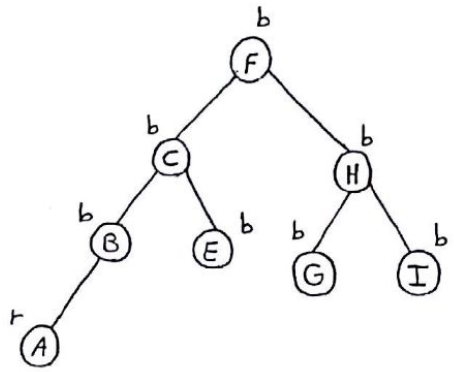

b
B
b
A
r
D
b
C
b
F
r
E
r
G
r
J
case1
+H
L
R
+I
case1
case1



Delete J →



Delete D →



Delete C →