DATA STRUCTURES

TREES

Prof. Khalil Al-Shqeerat

CS Dept. – Qassim University

TREES

1. Introduction
2. Binary Trees
3. Binary Search Tree
4. Operations on a Binary Search Tree
5. AVL Trees

مصطلحات مهمة :

* **Root** → أول عقدة في الشجرة.
* **Parent** → العقدة الأب.
* **Child** → العقدة الابن.
* **Leaf** → عقدة ليس لها أبناء.
* **Subtree** → جزء من الشجرة.
* **Height** → طول أطول مسار من الجذر إلى ورقة.
* **Level** → مستوى العقدة داخل الشجرة.

INTRODUCTION

الشجرة هي هيكل بيانات (Data Structure) يُستخدم لتنظيم البيانات بطريقة هرمية (Hierarchical)

- A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.

تسهل عمليات البحث و التنقل

الشجرة تتكون من:
(روابط) Links و (عُقد) Nodes

- It is a collection of nodes that are connected by **links** and has a hierarchical relationship between the nodes.

أعلى عقدة في الشجرة تسمى Root.

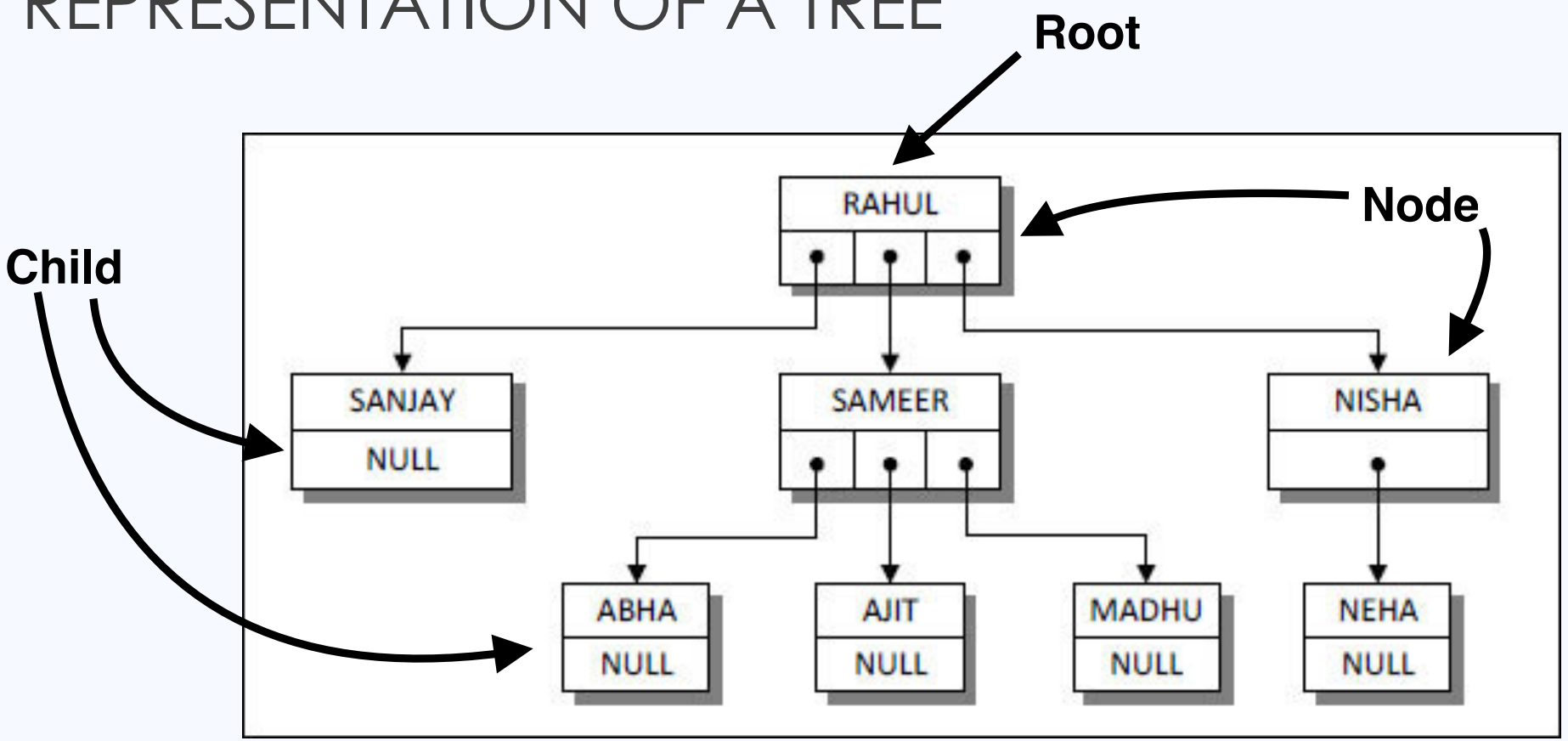
- The topmost node of the tree is called the **root**, and the nodes below it are called the **child** nodes..

العقد الموجودة أسفل عقدة معينة تسمى Children.

- Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a **recursive structure**.

لذلك تتكرر البنية نفسها داخل الشجرة، وهذا يسمى Recursive Structure.

REPRESENTATION OF A TREE



Binary Tree هي شجرة يكون لكل عقدة فيها بحد أقصى طفلان (Two Children):

BINARY TREES

* Left Child (الابن الأيسر)
* Right Child (الابن الأيمن)

- A tree that has at the most two pointers to the other tree nodes is called **Binary**.

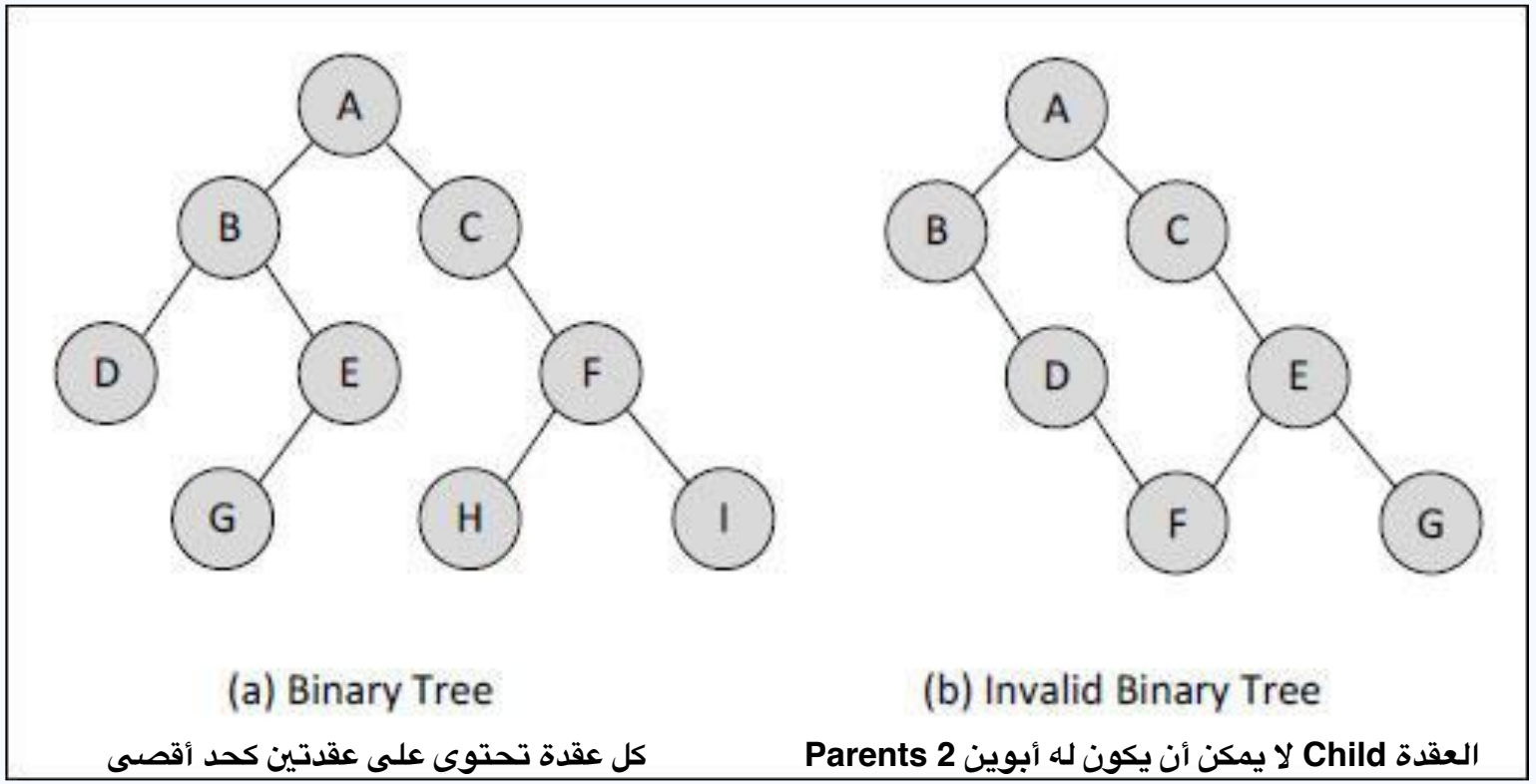
يعني أن كل Node تحتوى على مؤشرين فقط

- A **binary tree** is a finite set of elements that is either empty or is partitioned into three disjoint sub-sets.
  - The first sub-set contains a single element called the **root** of the tree.
  - The other two sub-sets are themselves binary trees, called the **left** and **right** sub-trees of the original tree. الجزء الأيسر والجزء الأيمن يعتبران Binary Trees مستقلة.
    - A left or right sub-tree can be empty. ليس من الضرورى وجود الفرعين معًا.

- Each element of a binary tree is called a **node** of the tree.

كل عنصر داخل الشجرة يسمى Node.

BINARY TREE



كل عقدة تحتوى على عقدتين كحد أقصى

العقدة Child لا يمكن أن يكون له أبوين Parents 2

SOME TERMINOLOGY

* A = Parent (الأب)
* B = Child (الابن)

- **Parent**: If A is the root of a binary tree and B is the root of its left or right sub-tree then, A is parent of B and B is left or right child of A
- An **ancestor** of a node is any other node on the path from the node to the root. هو أي عقدة تقع فوق العقدة الحالية في طريق الوصول إلى الـ Root.
  - An ancestor node of a node is either the parent of the node or the parent of some ancestor of the node.
- A **descendant** of a node is either a child of the node or a child of some descendant of the node (also called **successors**).

هو أى عقدة تقع أسفل العقدة الحالية.

TERMINOLOGY – CONT.

- **Climbing**, **Descending**: The root of the tree is at the top and the leaves at the bottom.
  - **Climbing** the tree when going from the leaves to the root.
  - **Descending** the tree when going from the root to the leaves.

Climbing (صعود): من الأسفل إلى الأعلى.
Descending (نزول): من الأعلى إلى الأسفل.

- **Degree** of a node is the number of nodes connected to this node.

عدد الأبناء (Children) للعقدة.

- **Depth** of a node is the maximum number of links from root to that node.

عدد الروابط (Links) من الـ Root إلى العقدة.

  - The depth of a binary tree is the maximum level of any leaf in the tree.
    - The length of the longest path from the root to any leaf.

TERMINOLOGY – CONT.

هو Height للـ Root

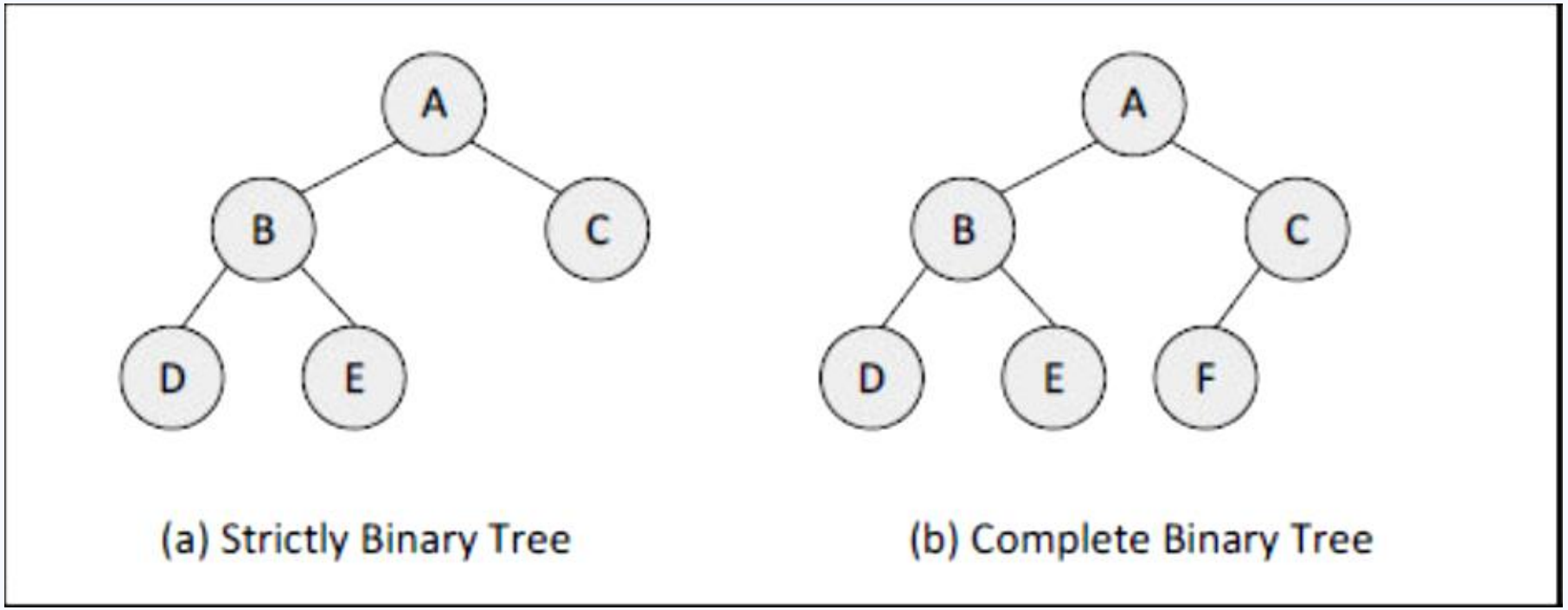
- **Height** of a node is the maximum number of links from that node to leaf node.
  - Height of a binary tree is height of its root node.

كل عقدة غير ورقية (Non-Leaf) يجب أن يكون لها طفلان بالضبط.

- **Strictly binary tree**: If every non-leaf node in a binary tree has nonempty left and right sub-trees.

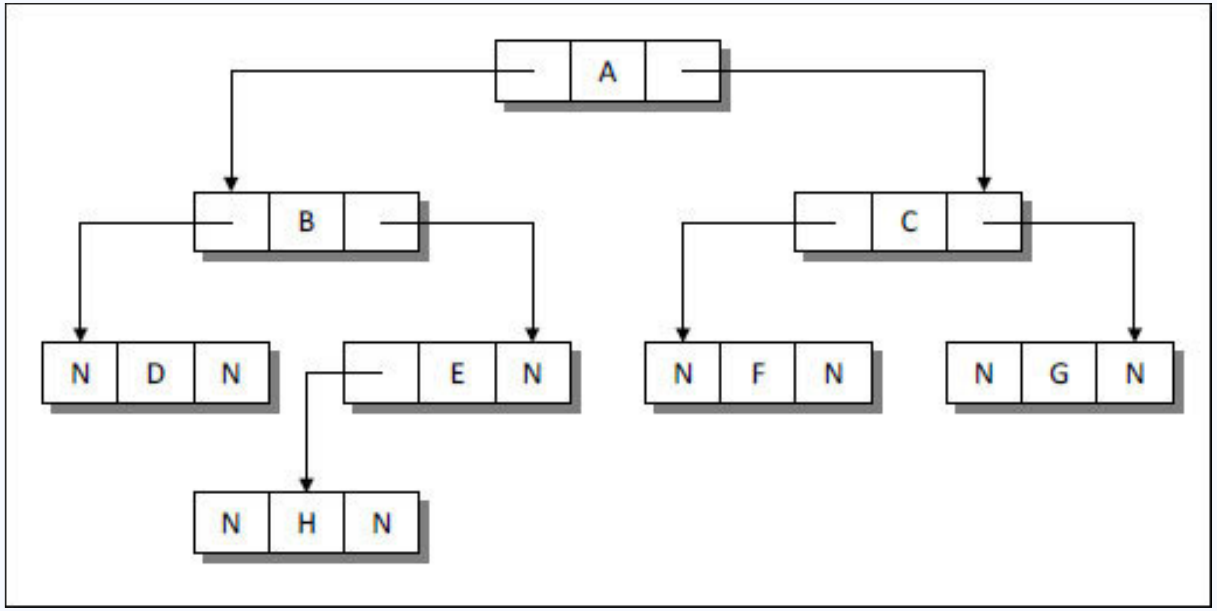
- **Complete binary tree**: the maximum number of possible nodes at all levels except the last level, and all the nodes of the last level appear as far left as possible. كل المستويات ممتلئة ما عدا الأخير، والأخير يُملأ من اليسار إلى اليمين.

STRICTLY AND COMPLETE BINARY TREE



LINKED REPRESENTATION OF BINARY TREES

- Each node contains addresses of its left child and right child.
  - If a child is absent, the link contains a NULL value.

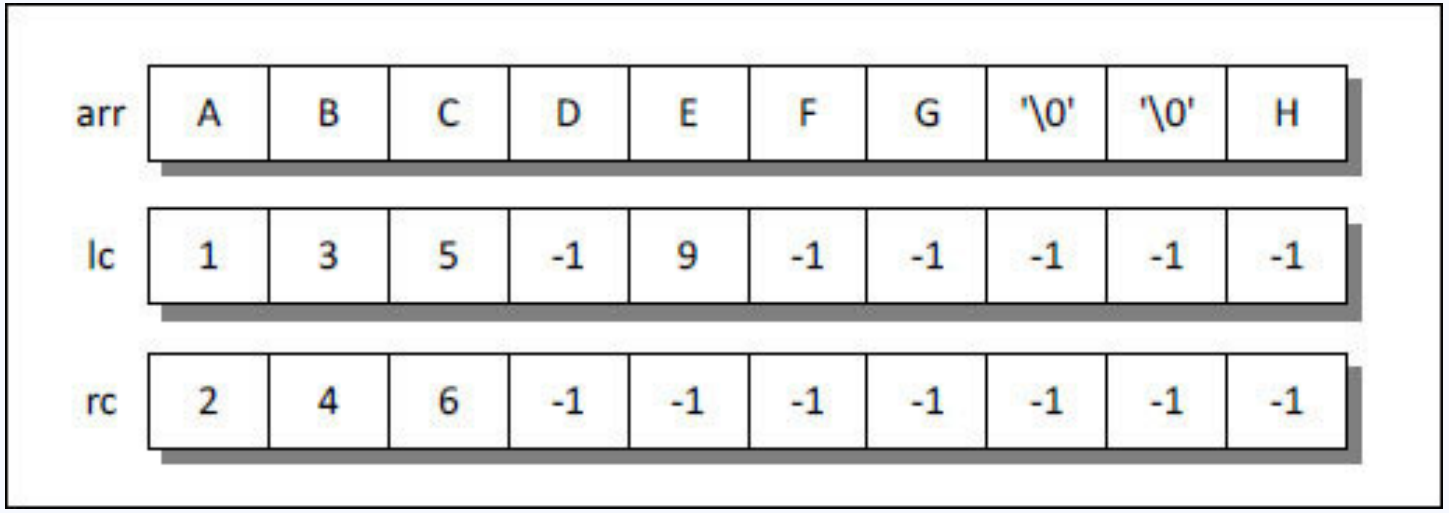


العقدة الواحدة تحتوي على :
1- مؤشر للعقدة اليمين
2- مؤشر للعقدة اليسار

```
Node* left;
Node* right;
```

ARRAY REPRESENTATION OF BINARY TREES

- When a binary tree is represented by arrays three separate arrays are required.
  - One array stores the data fields of the trees.
  - The other two arrays represent the left child and right child of the nodes.



arr	A	B	C	D	E	F	G	'\0'	'\0'	H
lc	1	3	5	-1	9	-1	-1	-1	-1	-1
rc	2	4	6	-1	-1	-1	-1	-1	-1	-1

Arrays 3 :الطريقة الأولى
نستخدم 3 مصفوفات:

Data[] → تخزن القيم
Left[] → تخزن موقع الابن الأيسر
Right[] → تخزن موقع الابن الأيمن

```
  A
 /\
B  C
```

Data = [A, B, C]
Left = [1, -1, -1]
Right = [2, -1, -1]

CONT.

- A binary tree can also be represented using one single array.
- Nodes are numbered from left to right level by level from top to bottom.

الطريقة الثانية: Single Array (الأهم)

نخزن الشجرة في مصفوفة واحدة فقط.

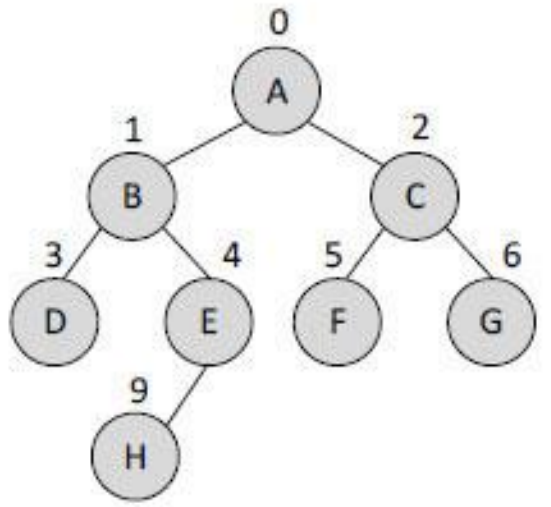
يتم ترقيم العقد:

* من اليسار إلى اليمين
* مستوى بمستوى (Level by Level)
* من الأعلى إلى الأسفل

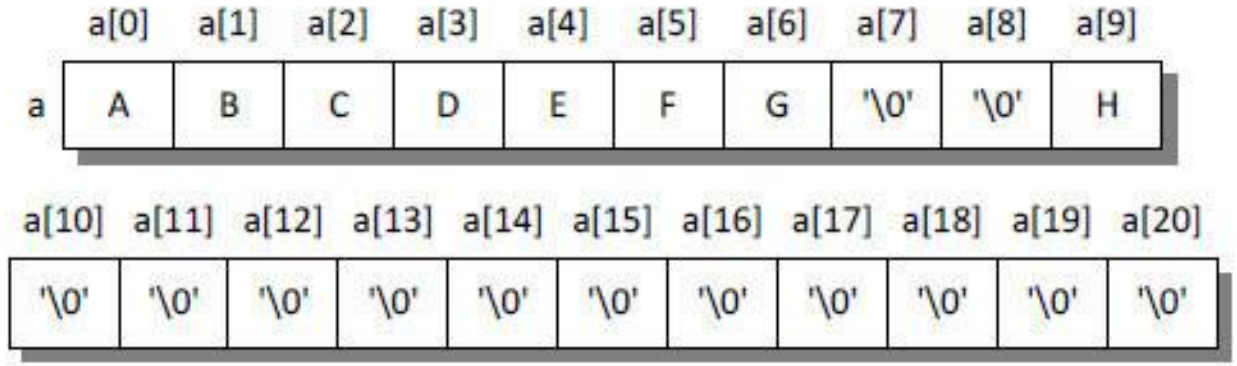
مثال:

```
    A
   / \
  B   C
 /\  /
D  E F
```

```
     A(0)
    /   \
  B(1)   C(2)
 / \    /
D(3) E(4) F(5)
```



(a) Numbering of nodes in Binary Tree



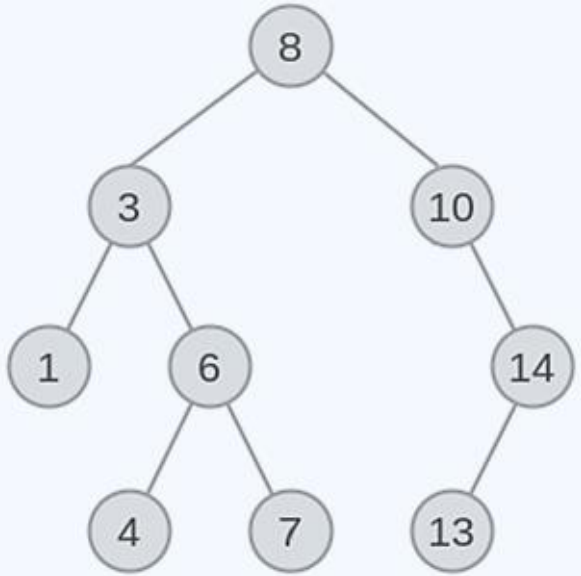
(b) Array representation of Binary Tree

هي Binary Tree لكن العناصر فيها مرتبة بطريقة معينة لتسهيل البحث.
القاعدة الأساسية

لكل عقدة (Node):
القيمة ≥ قيمة العقدة
إذًا العقدة بتكون على اليسار
القيم في اليمين ≤ قيمة العقدة
إذًا العقدة بتكون على اليمين

BINARY SEARCH TREES

- Binary search tree (BST) is a variant of binary tree in which the nodes are **arranged** in a particular manner.
  - The value of left node must be smaller than or equal to the parent node
  - The value of right node must be greater than or equal to the parent node.
- This rule is applied recursively to the left and right subtrees of the root.



ماذا يعني "Recursively"؟

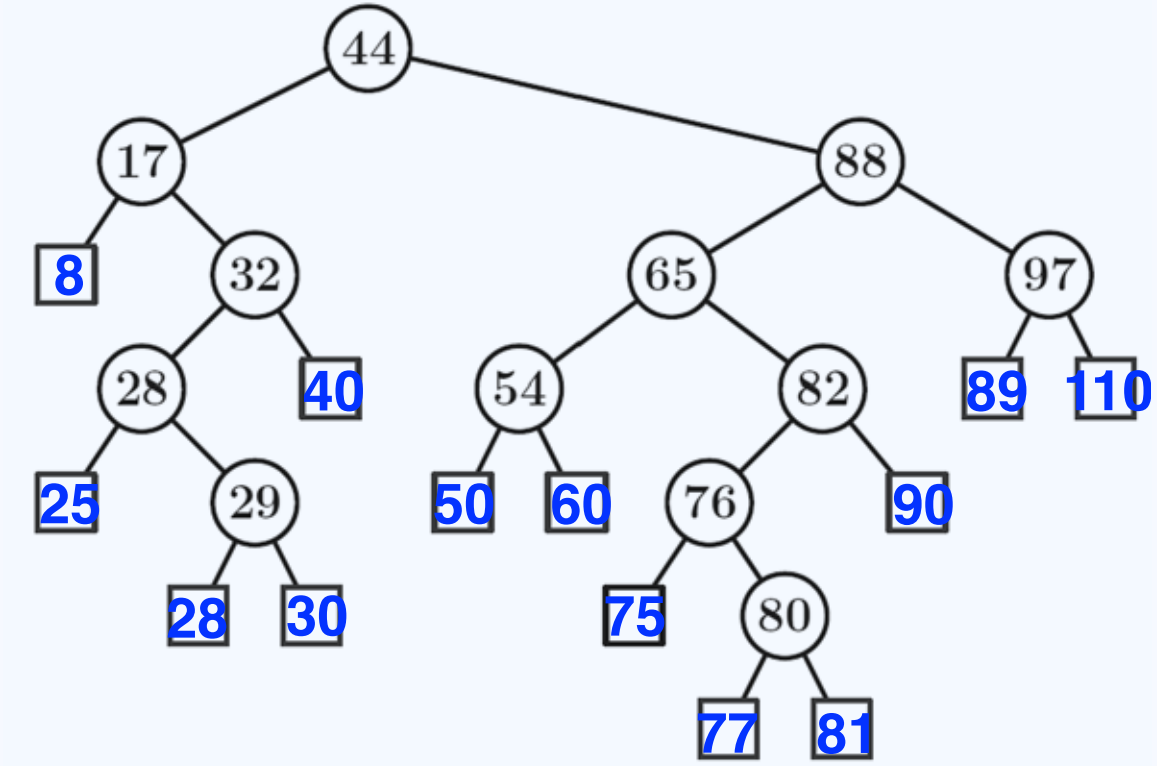
يعني أن القاعدة لا تطبق فقط على الـ Root، بل على كل Subtree أيضًا.

لماذا BST مهمة؟

لأنها تجعل عملية البحث أسرع

EXERCISE..

- Complete the following tree to satisfy BST data structure.



فكرة الإضافة

عندما نريد إضافة قيمة جديدة، نبدأ من الـ Root ونقارن القيمة الجديدة مع العقدة الحالية.

القاعدة

* إذا كانت القيمة أصغر → نذهب إلى اليسار.
* إذا كانت القيمة أكبر أو تساوي → نذهب إلى اليمين.
* نكرر العملية حتى نجد مكانًا فارغًا (NULL).
* نضيف العقدة الجديدة هناك.

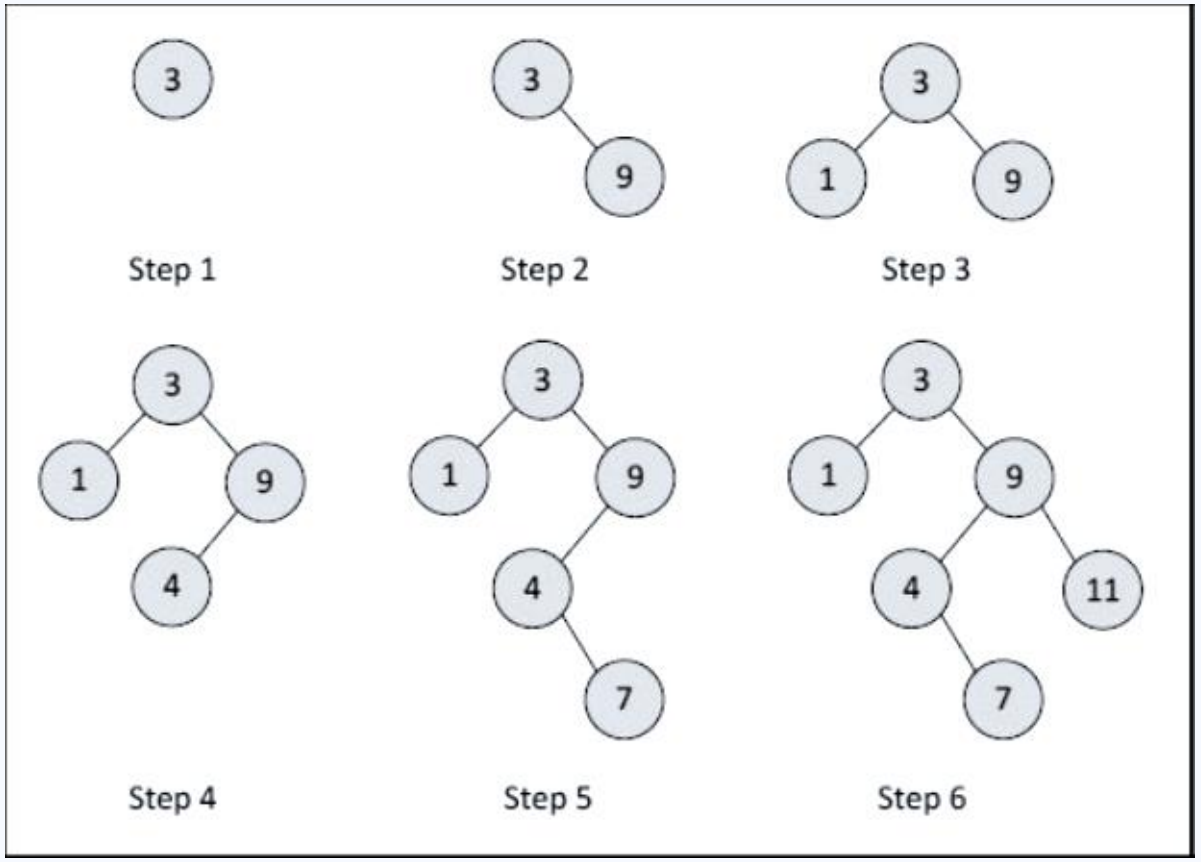
OPERATIONS ON A BINARY SEARCH TREE

- There are many operations that can be performed on binary search trees (such as Insertion, Traversal, Searching and Deletion).

Insertion of a Node

- While inserting a node in a BST the value being inserted is compared with the root node.
  - A left sub-tree is taken if the value is smaller than the root node.
  - A right sub-tree is taken If the value is greater than or equal to the node.
- this operation is repeated at each level till a node is found whose left or right sub-tree is empty.
- Finally, the new node is appropriately made the left or right child.

EXAMPLE


3
Step 1
3
9
Step 2
3
1
9
Step 3
3
1
9
4
Step 4
3
1
9
4
7
Step 5
3
1
9
4
11
7
Step 6

OPERATIONS ON A BINARY SEARCH TREE – CONT.

- **Traversal of a BST** هو عملية زيارة كل Node في الشجرة مرة واحدة فقط.
  - The traversal of a BST is to visit each node in the tree exactly once.
  - There are three popular methods of BST traversal: inorder traversal, preorder traversal and postorder traversal.
  - Traversing a BST involves visiting the root node and traversing its left and right sub-trees.
  - To traverse a non-empty BST in preorder:
    (1) Visit the root
    (2) Traverse the left sub-tree in preorder
    (3) Traverse the right sub-tree in preorder

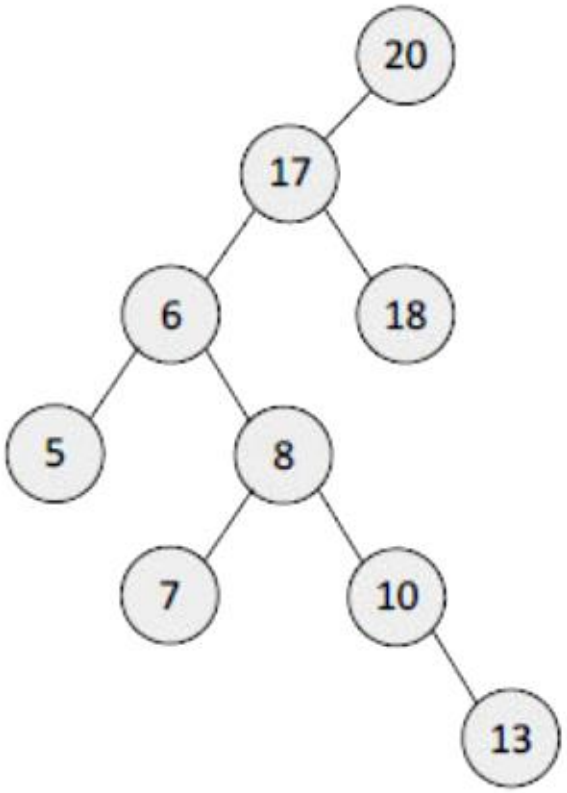
أنواعه :

1- PreOrder : Root -> Left -> Right
2- InOrder : Left -> Root -> Right
3- PostOrder : Left -> Right -> Root

TRAVERSAL OF A BST – CONT.

- To traverse a non-empty BST in inorder (or symmetric order):
  - (1) Traverse the left sub-tree in inorder
  - (2) Visit the root
  - (3) Traverse the right sub-tree in inorder
- To traverse a non-empty BST in postorder:
  - (1) Traverse the left sub-tree in postorder
  - (2) Traverse the right sub-tree in postorder
  - (3) Visit the root

TRAVERSAL OF A BST – EXAMPLE مهم جدًا



Inorder : 5, 6, 7, 8, 10, 13, 17, 18, 20

Prorder : 20, 17, 6, 5, 8, 7, 10, 13, 18

Postorder : 5, 7, 13, 10, 8, 6, 18, 17, 20

OPERATIONS ON A BINARY SEARCH TREE – CONT.

- **Searching of a Node**
  - To search any node in a binary tree, initially the value is compared with the root node.
    - If they match then the search is successful.
    - If the value is greater than the root node then searching process proceeds in the right sub-tree of the root node,
    - Otherwise, it proceeds in the left sub-tree of the root node.
  - BST search operation is very efficient because while searching an element we do not need to traverse the entire tree.



مهم جدًا جدًا
الـBST جدًا ممتازة في عمليات البحث، لأن مايحتاج نبحث في الشجرة كامل..

1- نقارن القيمة مع الـ ROOT
2- إذا كان القيمة أعلى من الـ ROOT مباشرة نروح لجزء الأيمن
3- غير كذا نبحث في الجزء الأيمن

OPERATIONS ON A BINARY SEARCH TREE – CONT.

- **Deletion of a Node**
  - Four cases are considered when deleting a node from a BST:
  - Case 1: Node to be deleted is absent.
    - If the node is not found when traversing the BST, we only need to display the message that the node does not exist.
  - Case 2: Node to be deleted has no children (leaf)
    - Simply delete the node from the tree
    - The left link or the right link of the parent of this node should be set to NULL.

❑DELETION OF A NODE – CONT.

- ❑Case 3: Node to be deleted has one child
  - ❑Replace that node with its child node
  - ❑The pointer of the parent of the node to be deleted should points to the child of the node being deleted.
  - ❑Remove the child node from its original position

- ❑Case 4: Node to be deleted has two children
  - ❑Get the inorder successor of that node.
  - ❑Replace the node with the inorder successor.
  - ❑Remove the inorder successor from its original position.

IMPLEMENTATION OF BST OPERATIONS

```
#include <iostream>
using namespace std;
struct btreenode
{
    btreenode *leftchild;
    int data;
    btreenode *rightchild;
};
class btree
{
```

Node تحتوي
مؤشر للعقدة اليمين
مؤشر للعقدة اليسار
Data جزء الـ

```
private:
    btreenode *root;
    void inorder (btreenode *sr);
    void preorder (btreenode *sr);
    void postorder (btreenode *sr);
    void insert (btreenode **sr, int);
    bool search (struct btreenode *, int);
    void locate (btreenode **sr, int num,
btreenode **par, btreenode **x, bool
*found);
    void rem (btreenode **sr, int num);
    void del (btreenode *sr);
```

مؤشر للجذر

فنكشن تعرض لي المحتوى بالـ Inorder

فنكشن تعرض لي المحتوى بالـ Preorder

فنكشن تعرض لي المحتوى بالـ Postorder

IMPLEMENTATION OF BST OPERATIONS – CONT.

```
public:
        btree();
        void buildtree (int num);
        void display();
        bool searchbst (int);
        void remove (int num);
        ~btree();
};
```

```
// initialises data members
btree:: btree()
{
        root = NULL;
}
// calls insert() to build tree
void btree:: buildtree (int num)
{
        insert (&root, num);
}
```

INSERTING A NEW NODE IN BST

```
void btree:: insert (btreenode **sr, int
num)
{
      if (*sr == NULL)
      {
            *sr = new btreenode;
            (*sr)->leftchild = NULL;
            (*sr)->data = num;
            (*sr)->rightchild = NULL;
      }
// search the node to which new node
will be attached};
      else
      {
      // if new data is less, traverse to left
            if (num < (*sr)->data)
                  insert (& ((*sr)->leftchild), num);
            else
      // else traverse to right
            insert (& ((*sr)->rightchild), num);
      }
}
```

insert() continues to move down the levels of BST until it reaches a leaf node. When it does, the new node gets inserted in the left/right sub-tree

TRAVERSING BST

```
// traverse BST in Left-Root-Right fashion
void btree:: inorder (btreenode *sr)
{
    if (sr != NULL)
    {
        inorder (sr->leftchild);
        cout << sr->data << "\t";
        inorder (sr->rightchild);
    }
}
```

```
// traverse BST in Root-Left-Right fashion
void btree:: preorder (btreenode *sr)
{
    if (sr != NULL)
    {
        cout << sr->data << "\t";
        preorder (sr->leftchild);
        preorder (sr->rightchild);
    }
}
```

TRAVERSING BST – CONT.

```
// traverse BST in Left-Right-Root fashion
void btree:: postorder (btreenode *sr)
{
    if (sr != NULL)
    {
        postorder (sr->leftchild);
        postorder (sr->rightchild);
        cout << sr->data << "\t";
    }
}
```

```
bool btree:: searchbst (int num)
{
    bool flag;
    flag = search (root, num);
    return flag;
}
```

SEARCHING BST

```
bool btree:: search (struct btreenode
*sr, int num)
{
      while (sr != NULL)
      {
            if (num == sr->data)
                  return true;
            else if (num < sr->data)
                  sr = sr->leftchild;
            else
                  sr = sr->rightchild;
      }
      return false;
}

// calls rem to delete node
void btree:: remove (int num)
{
      rem (&root, num);
}
```

DELETING A NODE FROM BST

```
// deletes a node from the BST
void btree:: rem (btreenode **sr, int
num)
{
      bool found;
      btreenode *parent, *x, *xsucc;
      // if tree is empty
      if (*sr == NULL)
      {
      cout << endl << "Tree is empty";
      return;
      }
parent = x = NULL;
// call the locate function to find the
node being deleted
locate (sr, num, &parent, &x, &found);
// if the node to deleted is not found
if (found == false)
{
      cout << endl << "Data to be deleted,
not found";
      return;
}
```

DELETING A NODE FROM BST – CONT.

```
// If the node to be deleted has two children
if (x->leftchild != NULL && x->rightchild != NULL)
{
      parent = x;
      xsucc = x->rightchild;
      while (xsucc->leftchild != NULL)
      {
            parent = xsucc;
            xsucc = xsucc->leftchild;
      }
      x->data = xsucc->data;
      x = xsucc;
}
// if the node to be deleted has no child
if (x->leftchild == NULL && x->rightchild == NULL)
{
      if (parent->rightchild == x)
      parent->rightchild = NULL;
      else parent->leftchild = NULL;
      delete x;
return;
}
```

DELETING A NODE FROM BST – CONT.

```
// if the node to be deleted has only rightchild
if (x->leftchild == NULL && x->rightchild != NULL)
{
      if (parent->leftchild == x)
      parent->leftchild = x->rightchild;
      else
      parent->rightchild = x->rightchild;
      delete x;
      return;
}
```

```
// if the node to be deleted has only left child
if (x->leftchild != NULL && x->rightchild == NULL)
{
      if (parent->leftchild == x)
            parent->leftchild = x->leftchild;
      else
            parent->rightchild = x->leftchild;
      delete x;
      return;
}
}
```

DELETING A NODE FROM BST – CONT.

```
// returns the address of the node to
be deleted, address of its parent
// and whether the node is found or
not
void btree:: locate (btreenode **sr, int
num, btreenode **par,
btreenode **x, bool *found)
{
      btreenode *q;
      q = *sr;
      *found = false;
      *par = NULL;
      while (q != NULL)
      {
// if the node to be deleted is found
            if (q->data == num)
            {
                  *found = true;
                  *x = q;
                  return;
            }
            *par = q;
```

DELETING A NODE FROM BST – CONT.

```
        if (q->data > num)
                q = q->leftchild;
        else
                q = q->rightchild;
        }
}

// calls del to deallocate memory
btree:: ~btree()
{
        del (root);
}
```

```
// deletes nodes of a BST
void btree:: del (btreenode *sr)
{
        if (sr != NULL)
        {
                del (sr->leftchild);
                del (sr->rightchild);
        }
        delete sr;
}
```

هو الفرق بين ارتفاع الشجرة اليسرى وارتفاع الشجرة اليمنى.

BalanceFactor = Height(Left Subtree) - Height(Right Subtree)

30
/
20
/
10

Height(left) = 2
* Height(right) = 0

إذن:
BF = 2
❌ غير متوازنة.

AVL TREES

لماذا نحتاج AVL؟

في BST العادية قد تصبح الشجرة طويلة جداً من جهة واحدة.
فيصبح البحث أبطأ.

- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.
  - It controls the height of the binary search tree

تراقب ارتفاع الشجرة باستمرار وتحافظ على توازنها AVL.

  - The balance condition ensures that the depth of the tree is O(logN).
- The difference between the heights of left and right subtrees, (**Balance Factor**), must be 0 or at most one (1, -1).

BALANCE FACTOR = HEIGHT(LEFT SUBTREE) – HEIGHT(RIGHT SUBTREE)

  - Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.

فتقوم AVL تلقائيًا
بعملية Rotation (تدوير)

تذكري القاعدة:

BALANCE FACTOR

BF = Height(Left)-Height(Right)

- If balance factor of any node is 1:
  - The left sub-tree is one level higher than the right sub-tree.

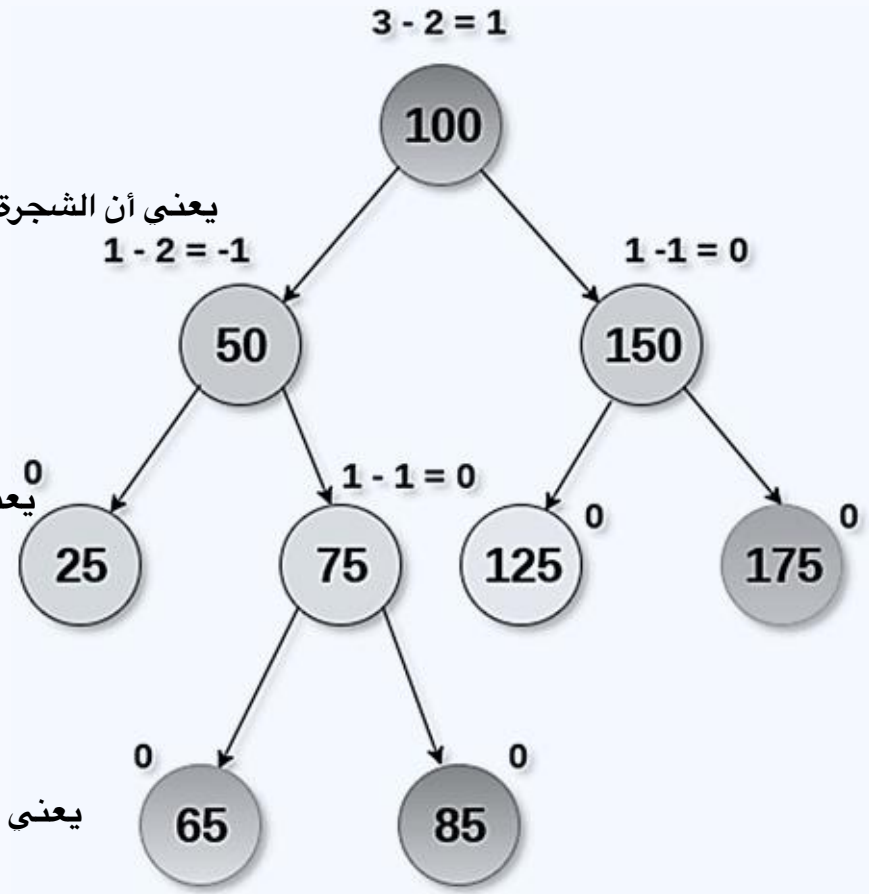
يعني أن الشجرة اليسرى أعلى بمستوى واحد.

- If balance factor of any node is 0:
  - The left sub-tree and right sub-tree contain equal height.

يعني أن ارتفاع اليسار = ارتفاع اليمين.

- If balance factor of any node is -1:
  - The left sub-tree is one level lower than the right sub-tree.

يعني أن الشجرة اليمنى أعلى بمستوى واحد.

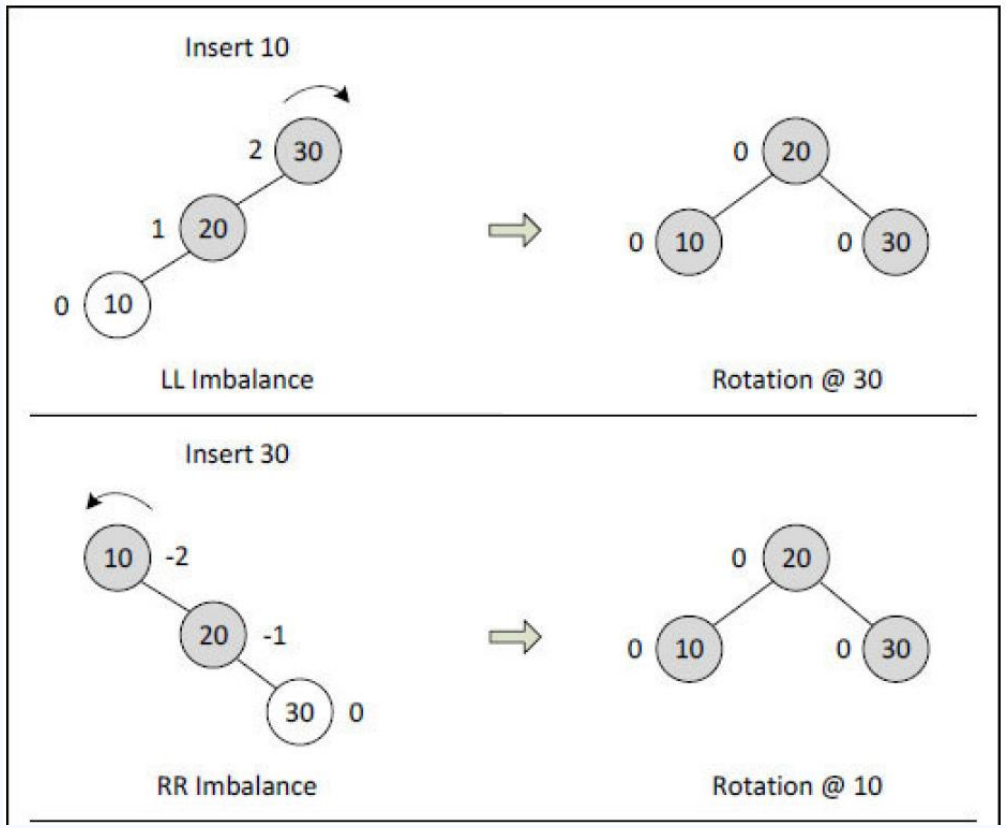


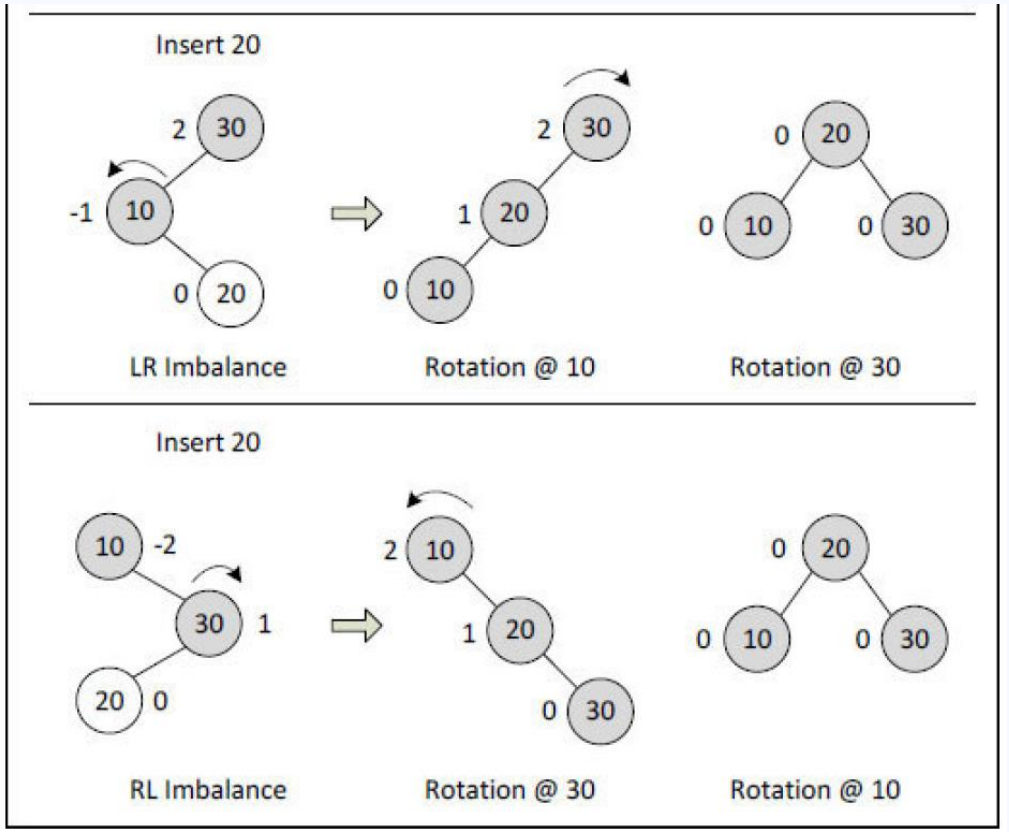
نستخدم Rotation لإعادة التوازن.
* الأنواع:
  * Right Rotation
  * Left Rotation
  * LR Rotation
  * RL Rotation
* إذا وجدت عدة عقد غير متوازنة:
* نختار أول Ancestor غير متوازن ونطبق الـ Rotation عليه.

BALANCE ROTATIONS

- If balance factor of any node is other than these three values then the tree is not balanced.
- To re-balance and make it an AVL tree the nodes need to be properly adjusted.
  - One of the 4 types of rotations – Left rotation, Right rotation, Left Right rotation, and Right Left rotation.
    - First two involve a 1 step process (single rotation), whereas the next two involve a 2 step process (double rotations).
- If 2 nodes are unbalanced then we need to rotate about the first **ancestor** that caused imbalance.

AVL ROTATIONS مثال



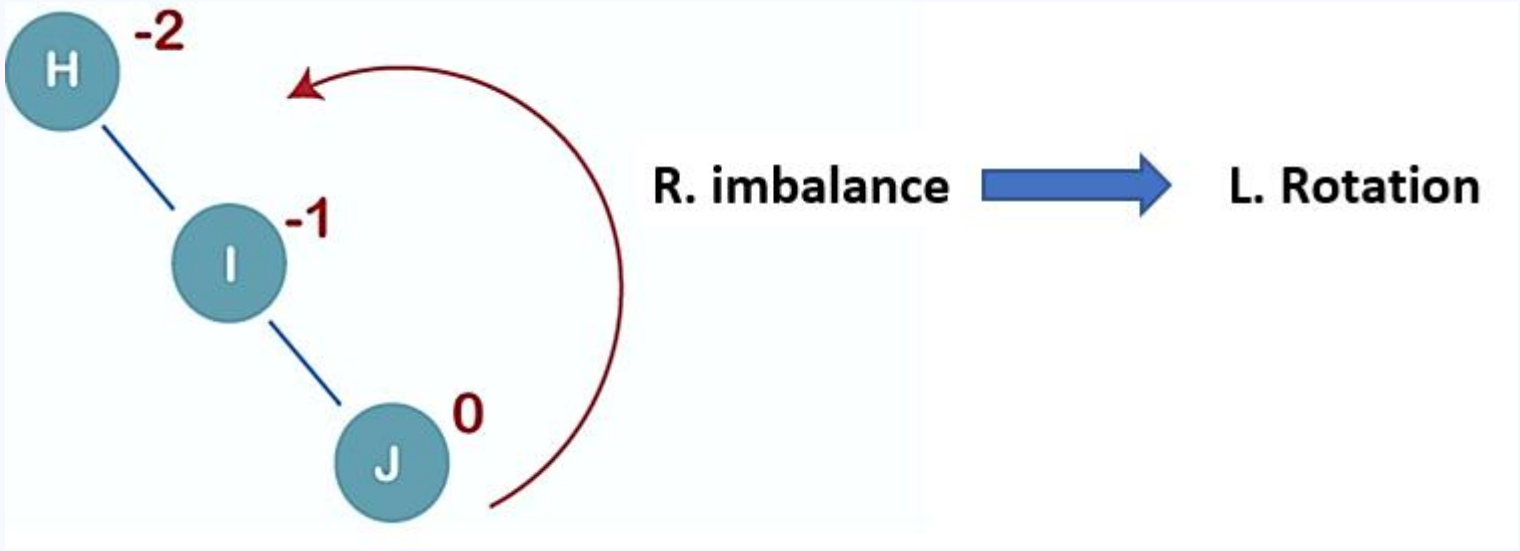


EXAMPLE

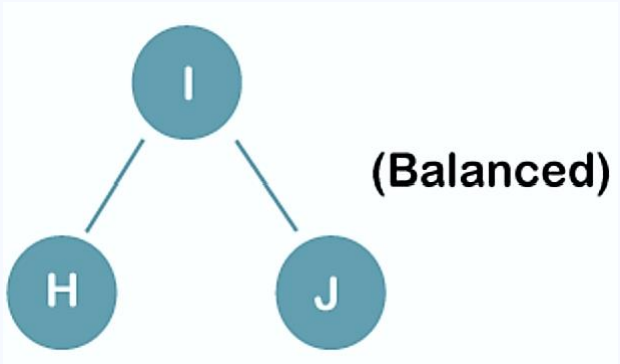
Construct an AVL tree having the following elements:

H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J



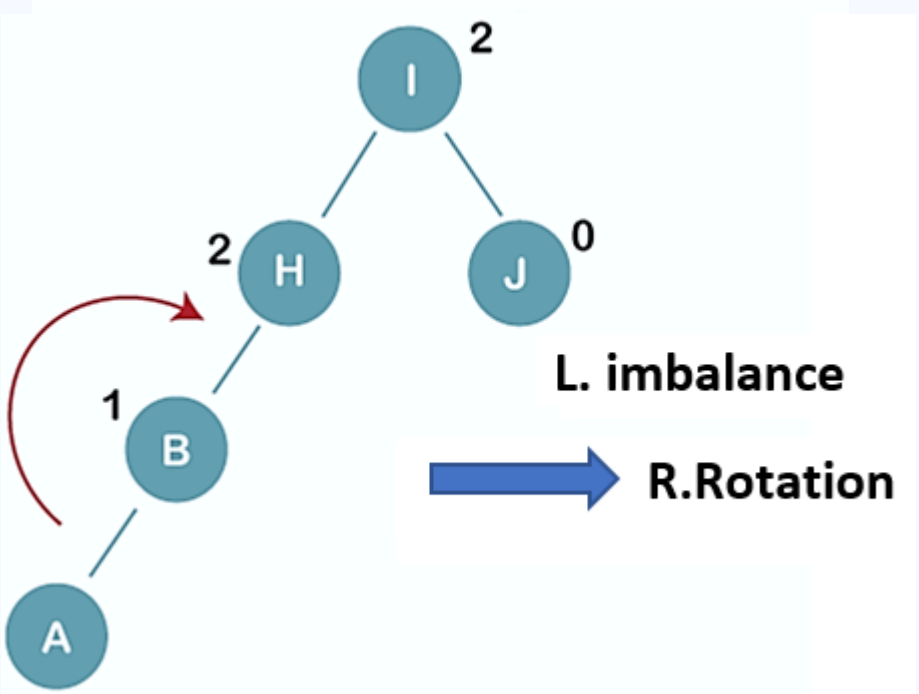
The resultant balance tree is:



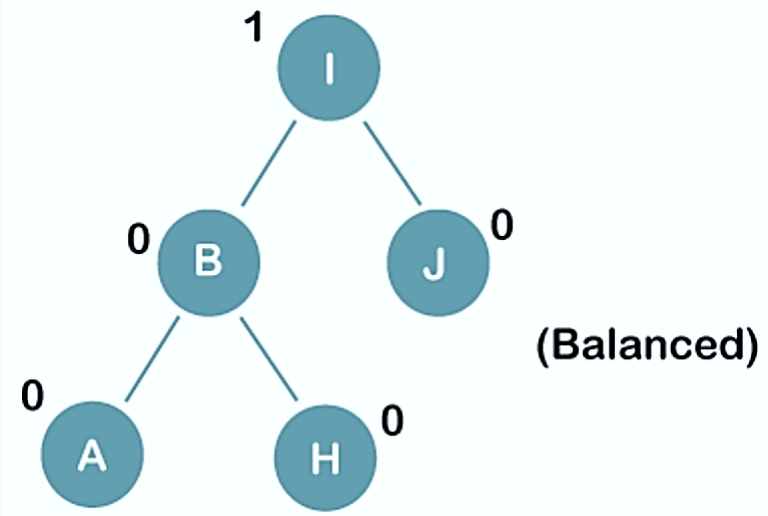
EXAMPLE – CONT.

H, I, J, B, A, E, C, F, D, G, K, L

2. Insert B, A



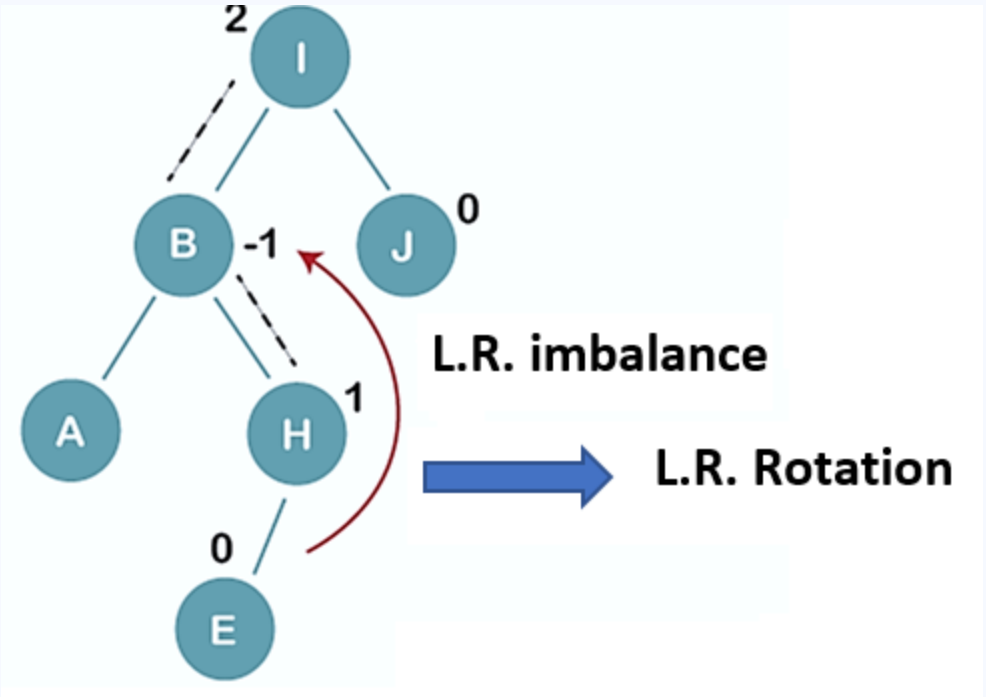
The resultant balance tree is:



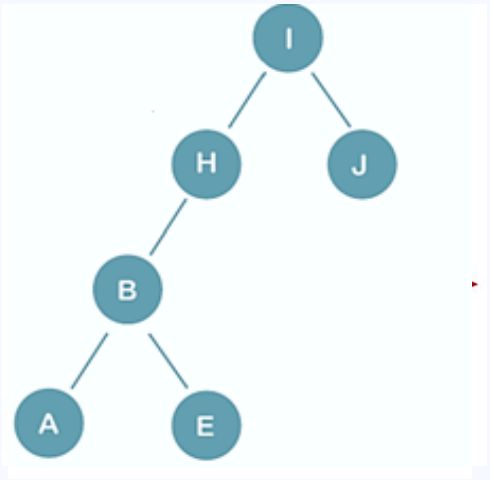
EXAMPLE – CONT.

H, I, J, B, A, E, C, F, D, G, K, L

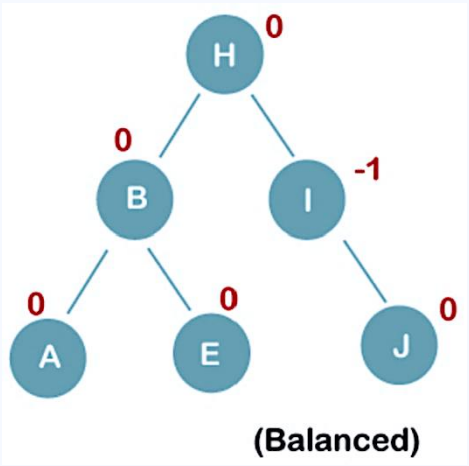
3. Insert E



The resultant tree after L rotation is:



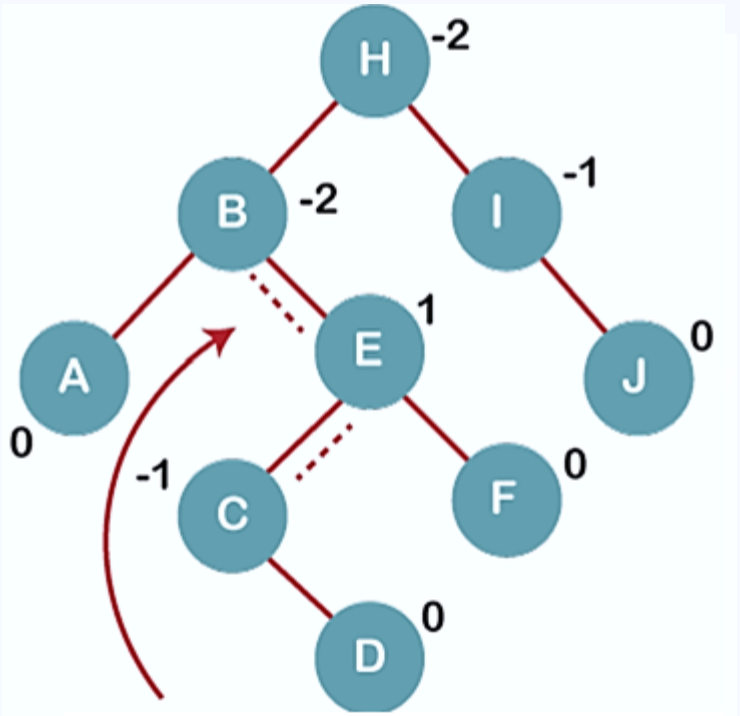
The resultant balanced tree after R rotation is:



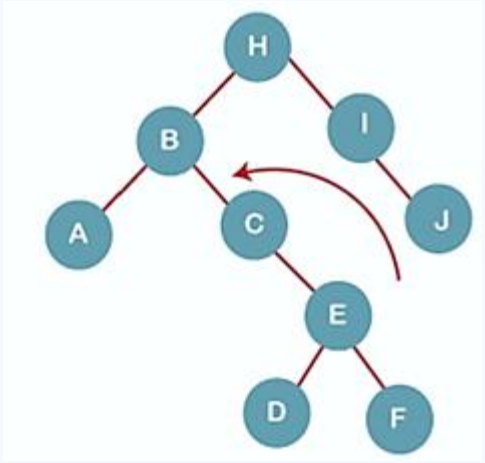
EXAMPLE – CONT.

H, I, J, B, A, E, C, F, D, G, K, L

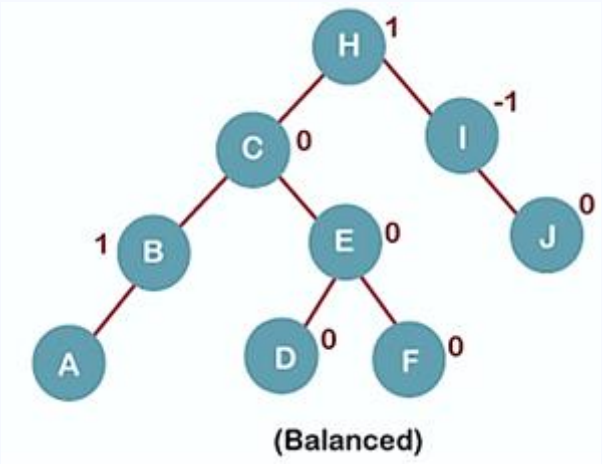
4. Insert C, F, D



The resultant tree after R rotation is:



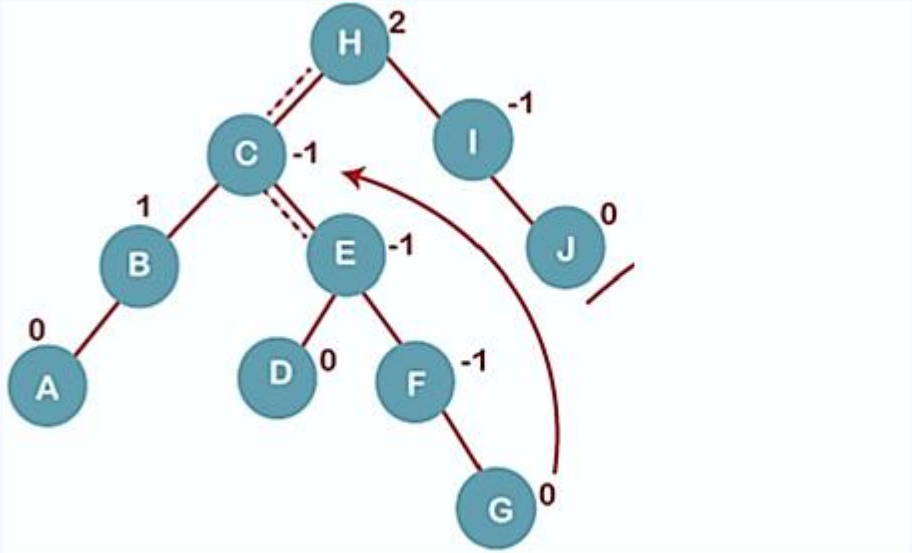
The resultant balanced tree after L rotation is:



EXAMPLE – CONT.

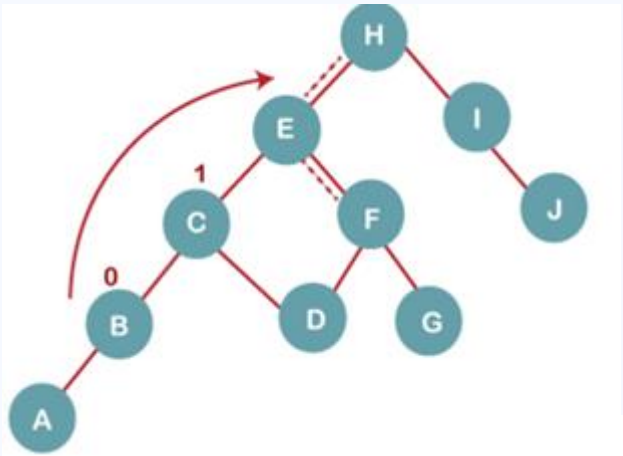
H, I, J, B, A, E, C, F, D, G, K, L

5. Insert G

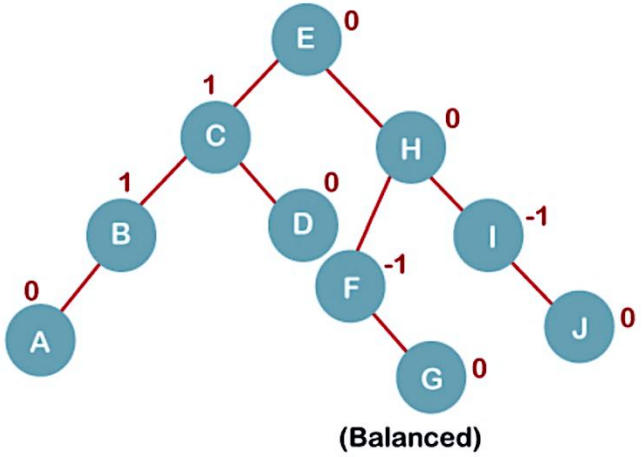


L.R. imbalance → **L.R. Rotation**

The resultant tree after L rotation is:



The resultant balanced tree after R rotation is:

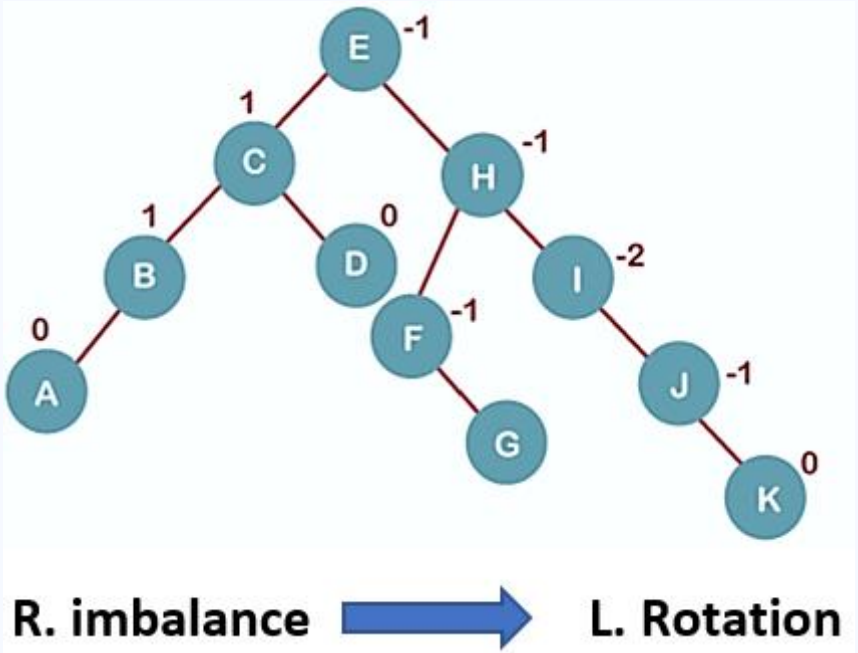


(Balanced)

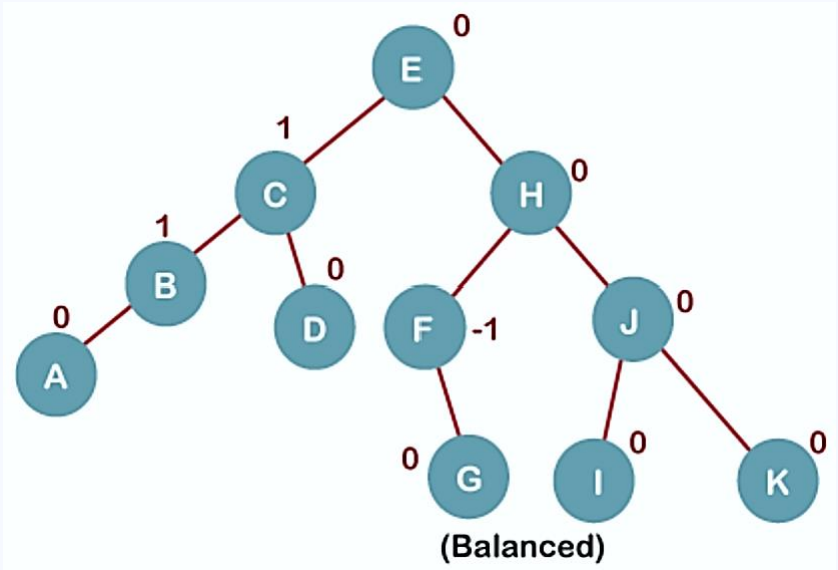
EXAMPLE – CONT.

H, I, J, B, A, E, C, F, D, G, K, L

6. Insert K



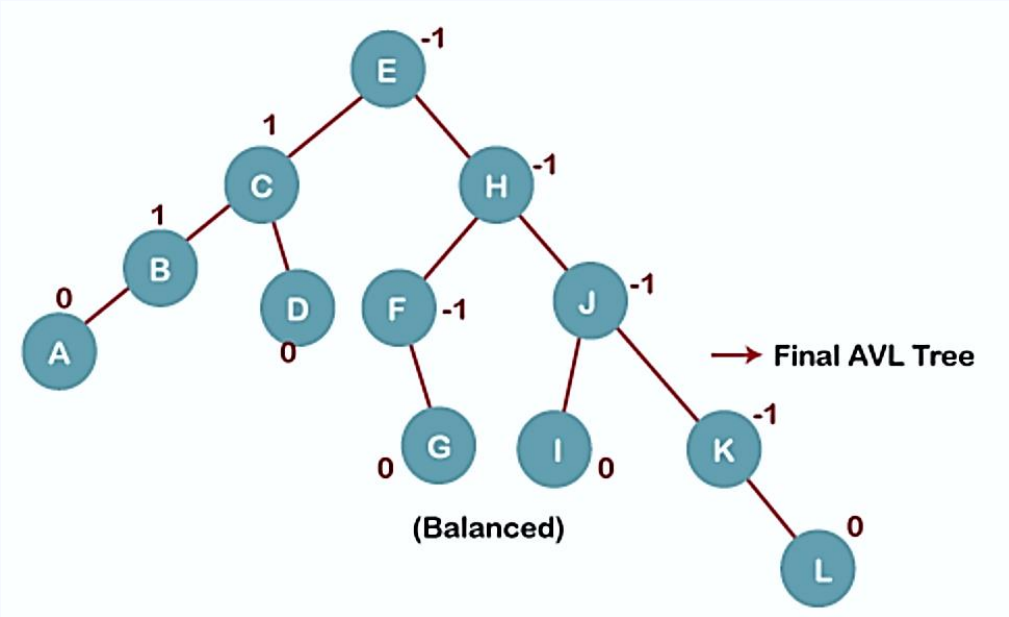
The resultant balanced tree after L rotation is:



EXAMPLE – CONT.

H, I, J, B, A, E, C, F, D, G, K, L

7. Insert L



EXERCISE

Construct an AVL tree having the following elements:

15, 20, 24, 10, 13, 7, 30, 36, 25

