



كلية الهندسة قسم المعلوماتية

بنى معطيات 1

Data Structure 1

ا.د. علي عمران سليمان

محاضرات الأسبوع التاسع

الأشجار 2

Tree 2

الفصل الثاني 2023-2024

| Trees 1 +2 | الأشجار 1 + 2                               |
|------------|---|
|            | 1- مقدمة                                    |
|            | 2- الأشجار العامة.                          |
|            | 3- خوارزميات التجول عبر الشجرة.             |
|            | 4- الأشجار الثنائية.                        |
|            | 5- تعريف شجرة البحث الثنائية كقاموس بيانات. |
|            | 6- تحقيق خوارزميات البحث.                   |
|            | 7- حساب أداء خوارزميات البحث.               |
|            | 8- تحقيق طرائق التبديل على شجرة بحث ثنائية. |
|            | 9- تحقيق شجرة البحث الثنائية بلغة ++C       |

## References

- Deitel & Deitel, Java How to Program, Pearson; 10th Ed(2015)

- د.علي سليمان، بني معطيات بلغة JAVA، بني معطيات بلغة ++C، بني معطيات بلغة Pascal جامعة تشرين 2014، 2007، 1998

شجرة البحث الثنائي BST – بتعريف بسيط - عبارة عن بنية معطيات من نوع شجرة tree ثنائية, تملك الخصائص التالية،

- كل عقدة  $v$  من الشجرة مخصصة لتخزين مفتاح مميز  $(k, x)$  بحيث إن:
- المفاتيح المخزنة في العقد التي في الشجرة الفرعية اليسارية لـ  $v$  أصغر من  $k$ .
- المفاتيح المخزنة في العقد التي في الشجرة الفرعية اليمينية لـ  $v$  أكبر من  $k$ .

إن المفاتيح المخزنة في عقد  $T$  تتيح طريقة لتنفيذ البحث من خلال إجراء مقارنة عند كل عقدة داخلية  $v$ , ويمكن أن تتوقف عند  $v$  أو تتابع عند الابن اليساري أو اليميني لـ  $v$ . وبالتالي، فإن أشجار البحث الثنائية هي أشجار ثنائية مكتملة proper غير فارغة، وتستخدم العقد الخارجية كمقايض placeholders. إن هذا الأسلوب يبسط الكثير من خوارزميات البحث والتعديل الخاصة بها.

يمكن في بعض الأحيان أن نسمح باستخدام أشجار بحث ثنائية غير مكتملة improper، فهي تتيح استخداماً أفضل لمساحة الذاكرة، إلا أنها أكثر كلفة وتعقيداً في تحقيق خوارزميات البحث والتعديل.

إن الميزة الهامة لشجرة البحث الثنائية هي تحقيق قاموس مرتب. أي أن شجرة البحث الثنائية يجب أن تمثل أو تعبر بشكل شجري أو هرمي hierarchically عن ترتيب لمفاتيحها، باستخدام علاقة بين الأب والابن. وبتحديد أكبر، فإن التجول المرتب inoreder traversal لجميع عقد شجرة بحث ثنائية  $T$  يجب أن يقوم بزيارة جميع المفاتيح بالترتيب.

• خوارزمية اختبار فيما إذا كانت شجرة بحث ثنائي أو لا.

- If the current node is **null** then return **true**.
- If the value of the left child of the node is greater than or equal to the current node then return **false**.
- If the value of the right child of the node is less than or equal to the current node then return **false**.
- If the left subtree or the right subtree is not a BST then return **false**.
- Else return **true**

# Basic Operations on BST

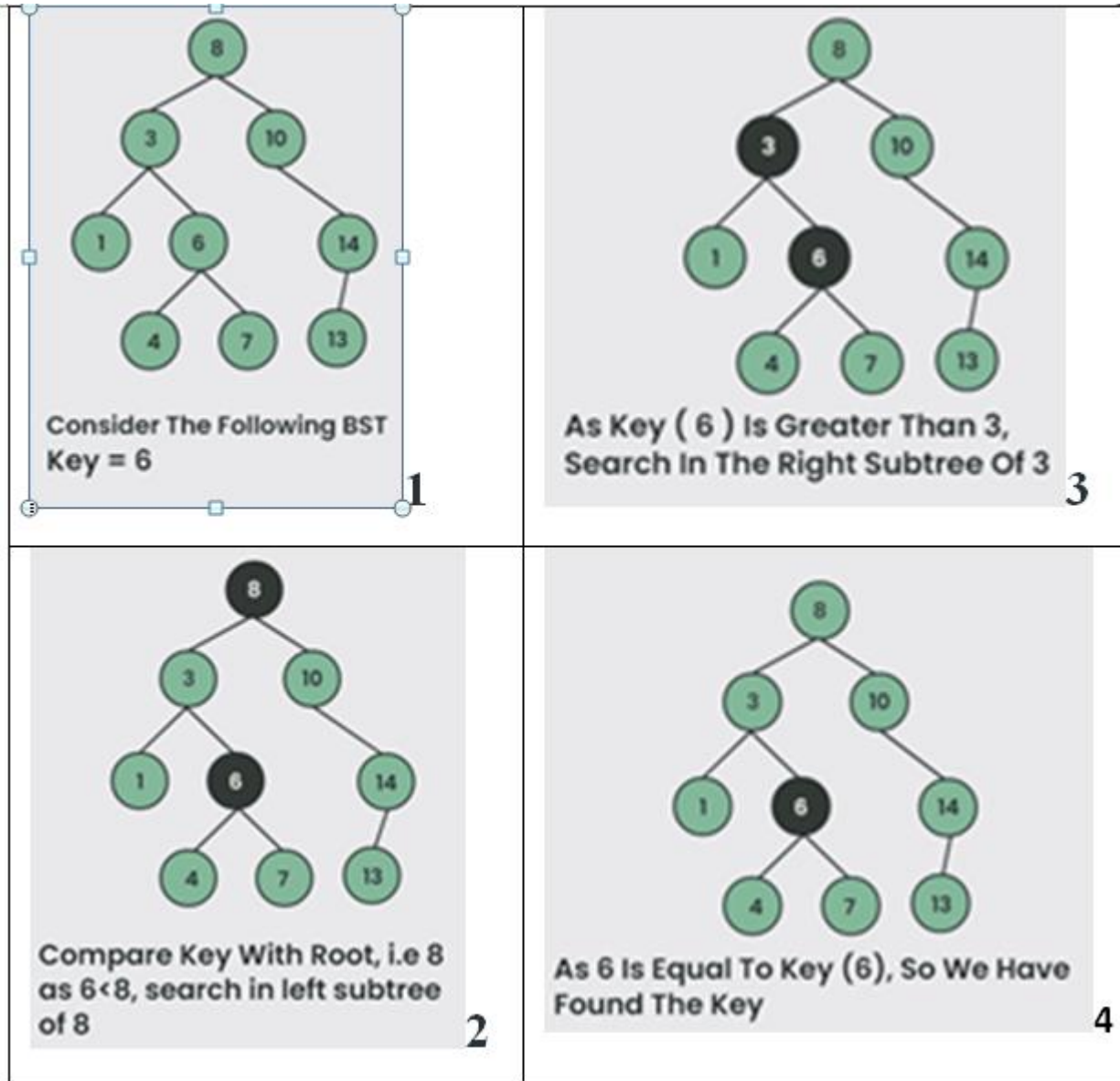


## العمليات الأساسية في أشجار البحث الثنائية 1

### Basic Operations on BST:

- Searching in Binary Search Tree
- Insertion in Binary Search Tree
- Deletion in Binary Search Tree
- Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order





لتنفيذ العملية  $find(k)$  في شجرة بحث ثنائية  $T$ ، ننظر إلى الشجرة على أنها شجرة قرار  $decision\ tree$ ، وتتم المقارنه عند كل عقدة داخلية  $v$  مع مفتاح البحث  $k$  وإذا كانت نتيجة مقارنته مع القيمة المخزنه في العقدة  $v$ ، والمشار إليه بـ  $key(v)$ .

- أصغر عندئذ يستمر البحث في الشجرة الفرعية اليسارية،
- أكبر عندئذ يستمر البحث في الشجرة الفرعية اليمينية.
- وإلا إنه (يساوي) عندئذ فإن البحث ينتهي بنجاح  $return\ true$
- إذا وصلنا إلى عقدة خارجية ولا تملك مفتاح البحث، فإن البحث ينتهي بالسلب ويعيد  $return\ false$ .

يبين المقطع الخوارزمي 1-7 وصفاً مفصلاً لهذا الأسلوب، حيث  $k$  هو مفتاح البحث، و  $v$  هي عقدة من  $T$ .  
 تعيد الطريقة  $TreeSearch$  عقدة (موقع)  $w$  من الشجرة الفرعية  $T(v)$  من الشجرة  $T$  والتي جذرها  $v$ ، حيث يحصل أحد الأمور التالية:

-  $w$  هي عقدة داخلية ويملك العنصر  $w$  مفتاحاً مساوي لـ  $k$ .

-  $w$  هي عقدة خارجية تمثل موقع  $k$  في حالة تجول ترتيب عبر  $T(v)$ ، إلا أن  $k$  ليست مفتاحاً متضمناً في  $T(v)$ .

وبالتالي، يمكن إنجاز الطريقة  $find(k)$  من خلال استدعاء  $TreeSearch(k, root())$ .

لتكن  $w$  هي العقدة من الشجرة  $T$  المعادة بهذا الاستدعاء. إذا كانت  $w$  هي عقدة داخلية، عندئذ نعيد قيمة  $w$  وإلا فإننا

نعيد  $null$ . ندرج خوارزمية البحث العودية

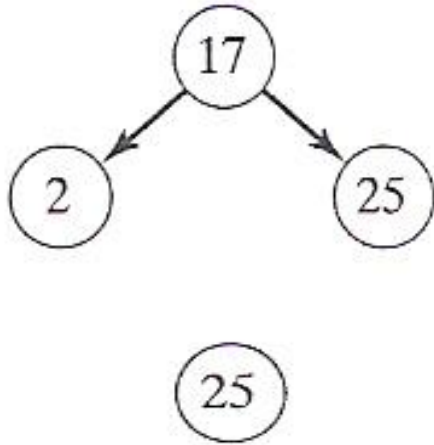
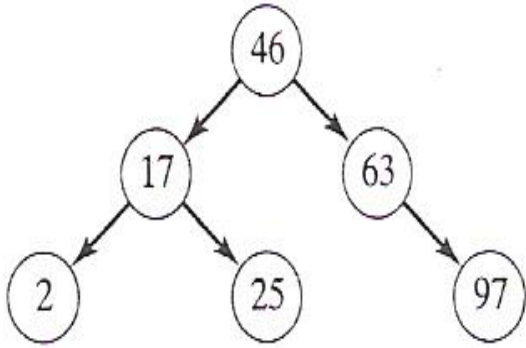
**Algorithm**  $TreeSearch(k,v)$ :

```

if  $T.isExternal(v)$  then    return  $v$ 
if  $k < key(v)$  then         return  $TreeSearch(k, T.left(v))$ 
else if  $k > key(v)$  then    return  $TreeSearch(k, T.right(v))$ 
return  $v$                     {we know  $k = key(v)$ }
  
```



## مثال 1 عن البحث في شجرة البحث الثنائية 1



نفرض أننا نريد البحث ضمن الشجرة المجاورة عن القيمة 25. نبدأ بالجزر وبما أن 25 اصغر من القيمة في الجذر أي 46، فإننا نستنتج أن القيمة المرغوبة قد تكون موجودة في الجزء الذي على يسار الجذر وبالتالي تصبح العملية هي البحث ضمن الشجرة الفرعية اليسارية والتي جذرها هو 17.

نتابع الآن عملية البحث بمقارنة القيمة المراد البحث عنها 25 مع القيمة في جذر الشجرة الفرعية وبما أنها أكبر منها وبالتالي يجب أن يتم البحث في الجزء الذي على يمين العقدة الجذر أي: باختبار القيمة في الجذر في الشجرة الفرعية المؤلفة من عقدة واحدة نصل إلى إيجاد العقدة المطلوبة. وعندما يصل البحث إلى ورقة خارجية ولا تملك القيمة المبحوث عنها عندها نصل لنتيجة البحث السلبية القسمة غير موجوده.



## مثال 1 عن البحث في شجرة البحث الثنائية 2

وبالتالي يجب إضافة التصريح عن التابع Search() ضمن القسم public للصف BST :

```
bool Search(const int & item) const;
```

ومن كتابة الشيفرة التالية في الملف:

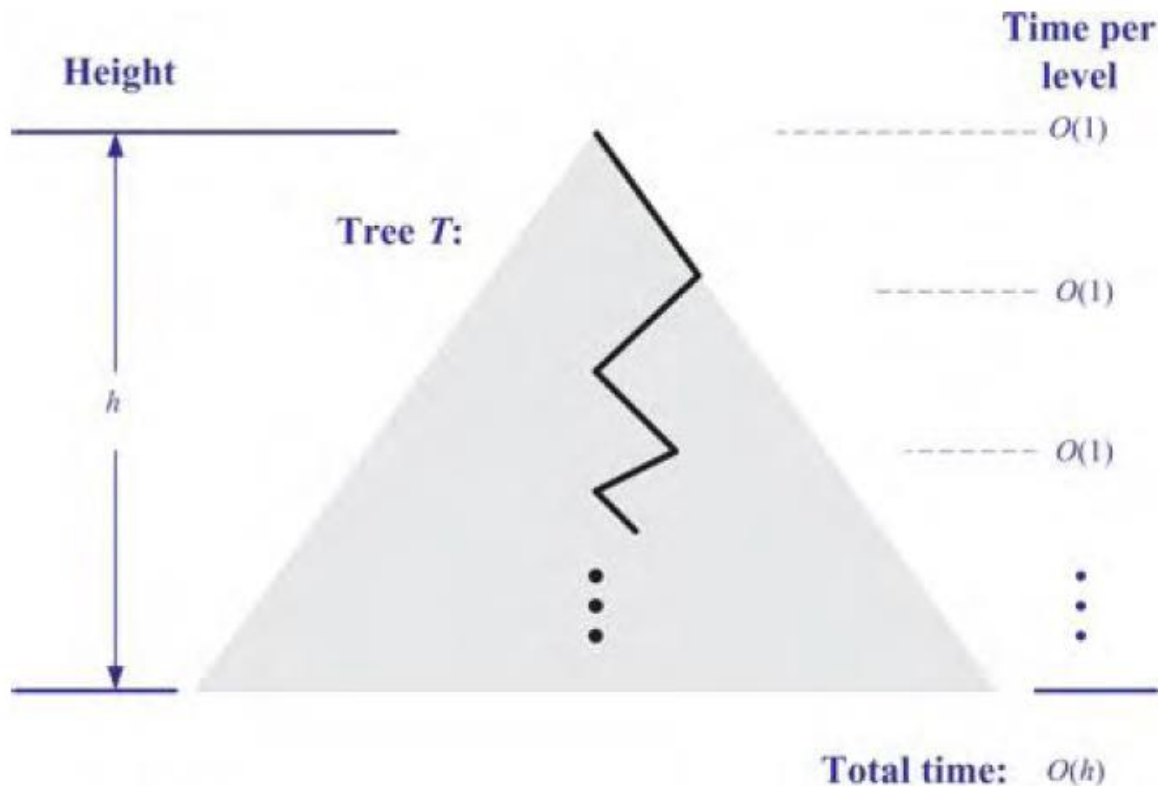
```
bool BST :: Search(const int & item) const
{
BST * locptr = root;  bool found = false;
for ( ;; )
{  if (found || locptr == 0) break;
  if (item < locptr->data)      // descend left
    locptr = locptr->left;
  else if (item > locptr->data) // descend right
    locptr = locptr->right;
  else      found = true; // item found
}
return found;
}
```

حيث يبدأ المؤشر locptr عند جذر شجرة البحث الثنائي ومن ثم ينتقل إلى الرابط اليميني أو اليساري للعقدة الحالية بحسب كون العنصر الذي نبحث عنه أصغر أو أكبر من القيمة المخزنة في العقدة. تستمر هذه العملية حتى يتم إيجاد العنصر المرغوب أو يصبح locptr صفرياً مشيراً إلى شجرة فرعية فارغة والتي تعني أن العنصر غير موجود في الشجرة.

```
// Utility function to search a key in a BST
BST BST::search(BST * root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL) return NULL;
    if (root->key == key) return root ;
    // Key is greater than root's key
    if (root->key < key) return search(root->right, key);
    // Key is smaller than root's key
    return search(root->left, key);
}
```

# Analysis of Binary Tree Searching

## تحليل البحث في شجرة ثنائية 1



إن تحليل زمن تنفيذ الحالة الأسوأ لعملية البحث في شجرة بحث ثنائية T يعتبر أمراً بسيطاً. فالخوارزمية TreeSearch هي خوارزمية عودية وتنفذ عدداً ثابتاً من العمليات البسيطة في كل عملية استدعاء عودي. يتم تنفيذ كل استدعاء عودي للطريقة TreeSearch على إبن من أبناء العقدة السابقة. أي أن، الطريقة TreeSearch يتم استدعاؤها على عقد مسار من T يبدأ عند الجذروينزل هبوطاً مستوى واحد في كل مرة. وبالتالي، عدد هذه العقد محدد بـ  $h+1$ ، حيث  $h$  هو ارتفاع T.

الشكل 7-2- كلفة البحث على شجرة ثنائية. إن القيمة المقدمة الخوارزمية السابقة لتنفيذ العملية  $\text{findAll}(k)$  هي زمن  $O(h+s)$ ، حيث  $s$  عدد القيم التي يتم إيجادها (إعادتها). إلا أن هذه الطريقة أكثر تعقيداً بقليل وستدرسها لاحقاً.

بكلام آخر، بما أننا ننفق زمناً مقداره  $O(1)$  عند كل عقدة في البحث، فإن الطريقة  $\text{find}$  تنفذ في زمن  $O(h)$ ، حيث  $h$  هو ارتفاع شجرة البحث الثنائية T المستخدمة لتحقيق الشجرة. (يبين الشكل 7-2 هذه العملية).

تتيح أشجار البحث الثنائية تحقيقات للعمليات `insert` و `remove` باستخدام خوارزميات مباشرة تماماً.

1- الحشر Insertion: لنفرض شجرة ثنائية  $T$  تتضمن عملية التعديل التالية:

- `insertAtExternal(v,e)`: إضافة العنصر  $e$  في عقدة خارجية  $v$  وتوسيع  $v$  لتكون داخلية من خلال إعطائها المؤشرين الصادرين عنها القيمة `NULL`.

باستخدام هذه الطريقة، يمكن تنفيذ `insert(k,x)` والمبينة في المقطع 2-7.

```
BST* BST::Insert(BST* root, int value)
```

```
{if (!root) {return new BST(value);} // Insert the first node, if root is NULL.
```

```
if (value > root->data) {// Insert data.
```

```
    /*Insert right node data, if the 'value' to be inserted is greater than 'root'.Process right nodes.*/
```

```
    root->right = Insert(root->right, value);}
```

```
else if (value < root->data) {
```

```
    /* Insert left node data, if the 'value' to be inserted is smaller than 'root' node data. Process left nodes.*/
```

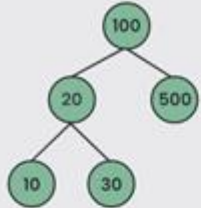
```
        root->left = Insert(root->left, value);}
```

```
return root;} // Return 'root' node, after insertion
```

## Update Operations(Insertion)

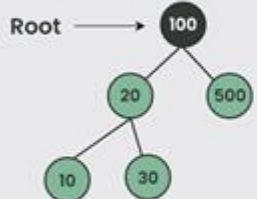
## طرائق التعديل (الحشر) 2

Consider The Following BST



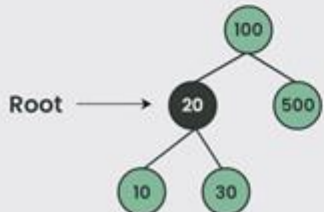
X = 40 ( The Node To Be Inserted )

**STEP 1 :** Comparing X with Root Node



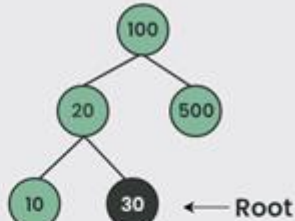
Since 100 Is Greater Than 40.  
Move Pointer To The Left Child ( 20 )

**STEP 2 :** Comparing X with left child of root node



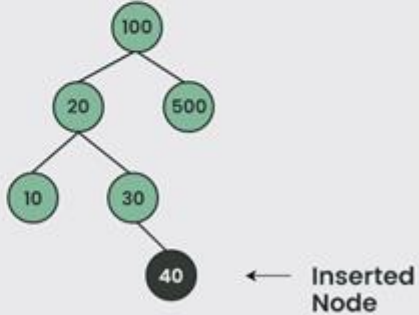
Since 20 Is Less Than 40, Move  
Pointer To The Right Child ( 30 )

**STEP 3 :** Comparing x with the right child of 20



Again 40 Is Greater Than 30  
Move Pointer To The Right Side  
Of 30

**STEP 4 :** Insert item to the right of 30



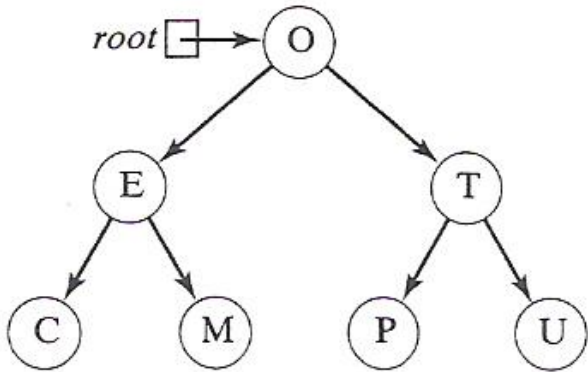
As 40 Is Greater Than The Node 30,  
Thus It Will Be Inserted To The Right  
Of 30

تقوم هذه الخوارزمية بتتبع مسار بدءاً من جذر T إلى عقدة خارجية، والتي توسع إلى عقدة داخلية جديدة لاحتواء القيمة الجديدة. يبين الشكل 3-7 مثلاً للحشر في شجرة بحث ثنائية.

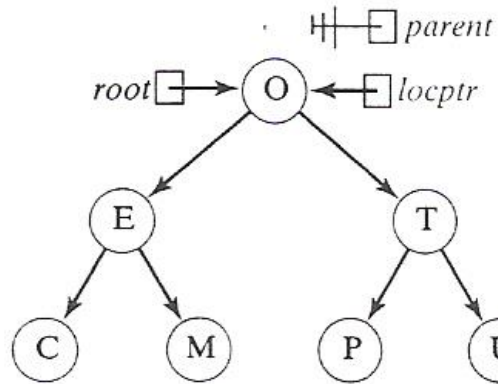
الشكل 3-7- حشر عنصر قيمته 40 في شجرة البحث المبينة في الشكل 1-7. إيجاد الموقع (a)

(b) الشجرة الناتجة.

## مثال 2 عن الحشر 1



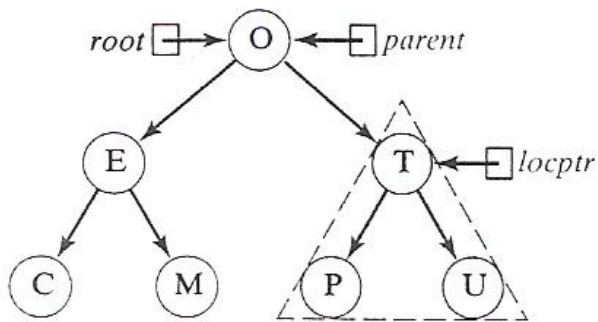
يمكن بناء شجرة البحث الثنائية بالاستدعاء المتكرر لتابع يقوم بحشر العناصر في شجرة البحث الثنائية التي هي فارغة في الحالة الابتدائية ( *root* يشير إلى المؤشر الصفرى). الطريقة المستخدمة لتحديد المكان الذي سيتم حشر العنصر فيه مشابهة لتلك المستخدمة في عملية البحث، وبالتالي نحتاج فقط لأن نعدل التابع (*Search*) للمحافظة على مؤشر إلى العقدة الأب للعقدة المختبرة حالياً عندما ننزل عبر الشجرة باحثين عن مكان لحشر العنصر.



لتوضيح ذلك، نفرض أن شجرة البحث الثنائية المجاورة تم بناؤها:

وأننا نريد إضافة الحرف 'R'. نبدأ عند الجذر ونقارن 'R' مع الحرف الموجود ضمنها:

بما أن 'R' > 'O' نزل إلى الشجرة الفرعية اليمنى:



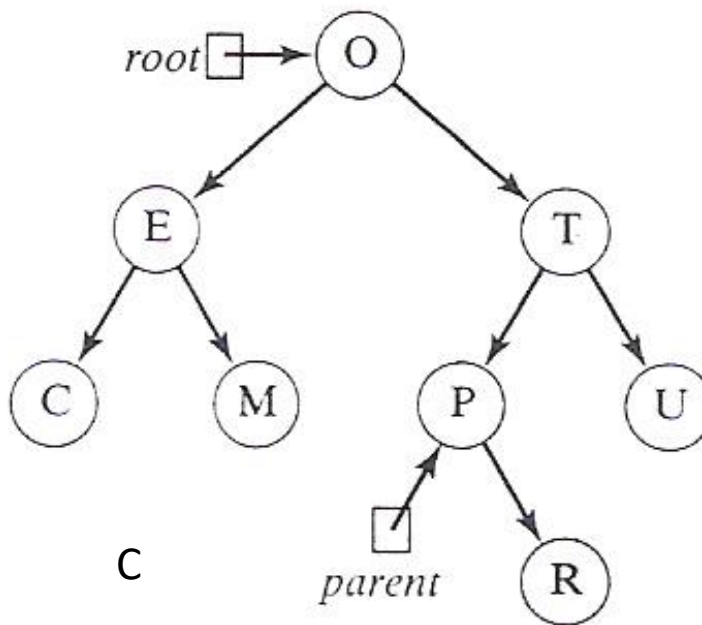
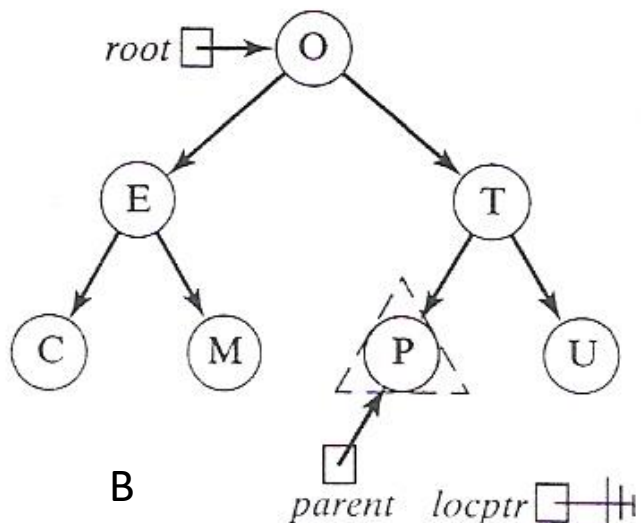
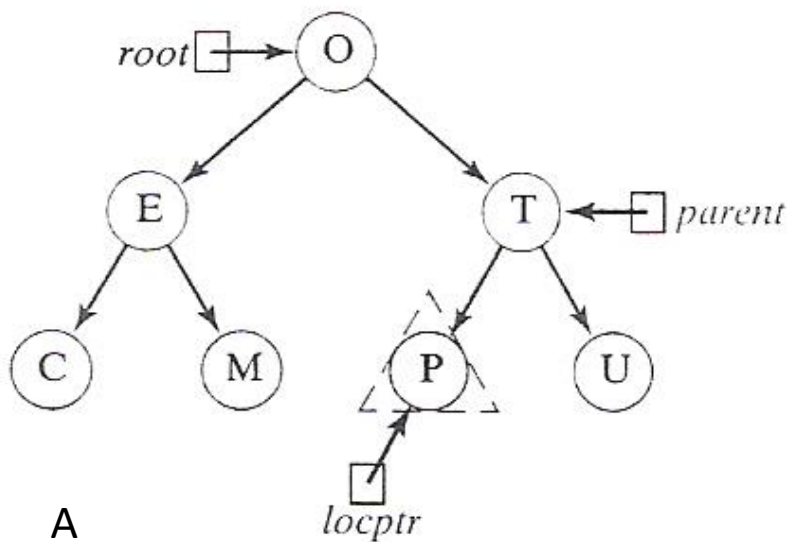
وبعد مقارنة 'R' مع 'T' المخزن في جذر هذه الشجرة الجزئية المشار إليها بواسطة *locptr*، نهبط إلى الشجرة الجزئية اليسرى بما أن 'R' < 'T':





## مثال 2 عن الحشر 2

وبما أن 'R' > 'P' نزل إلى الشجرة الجزئية اليمنى من هذه الشجرة الجزئية ذات العقدة الوحيدة الحاوية على 'P':  
 إن حقيقة أن الشجرة الجزئية اليمنى فارغة ( أي  $locptr == null$  )  
 تعني أن 'R' ليست موجودة في شجرة البحث الثنائية ويجب أن  
 تحشركابن يميني لهذه لعقدة الأب 'P'.





### Delete

### الحذف

- يعتبر تحقيق العملية  $delete(k)$  شجرة بحث ثنائية  $T$  أمراً أكثر تعقيداً بقليل من عمليات الحشر، وذلك لأننا لانرغب بإنشاء أي ثقب أو فجوات في الشجرة  $T$ . نفرض في هذه الحالة، أن أي شجرة ثنائية تدعم عملية التحديث الإضافية التالية:
- نبدأ تحقيقنا للعملية  $delete(k)$  كنمط بيانات مجرد باستدعاء  $TreeSearch(k, T.root())$  على  $T$  لإيجاد العقدة من  $T$  التي تخزن عنصراً ذو مفتاح يساوي  $k$  هناك احتمالين.
  - الأول: أن نصل لعقدة خارجية ولم نجد القيمة المبحوث عنها عندها تعيد الخوارزمية  $TreeSearch$  القيمة  $null$  ولن يتم الحذف.
  - الثاني: العنصر موجود وهناك ثلاث حالات أن تكون ورقه، أولها ابن وحيد أولها ابنان:
- 1-2 - إذا أعاد  $TreeSearch$  عقدة خارجية (ورقة)، عندها يتم وضع المؤشر الذي يشير إليها يساوي  $null$  ونعيد العقدة المحذوفة لخزان العقد الفارغة.
- 2-2- أما إذا أعاد  $TreeSearch$  عقدة داخلية  $w$  ولها ابن وحيد، عندئذ تخزن  $w$  العنصر الذي نرغب بحذفه، - بكل بساطة- بحذف  $w$  من  $T$  وتقوم بإعادة تشكيل  $T$  من خلال استبدال  $w$  بابنها الوحيد ونعيد العقدة المحذوفة لخزان العقد الفارغة. ، (الشكل 4-7).

# Update Operations

## طرائق التعديل ( الحذف ) 2

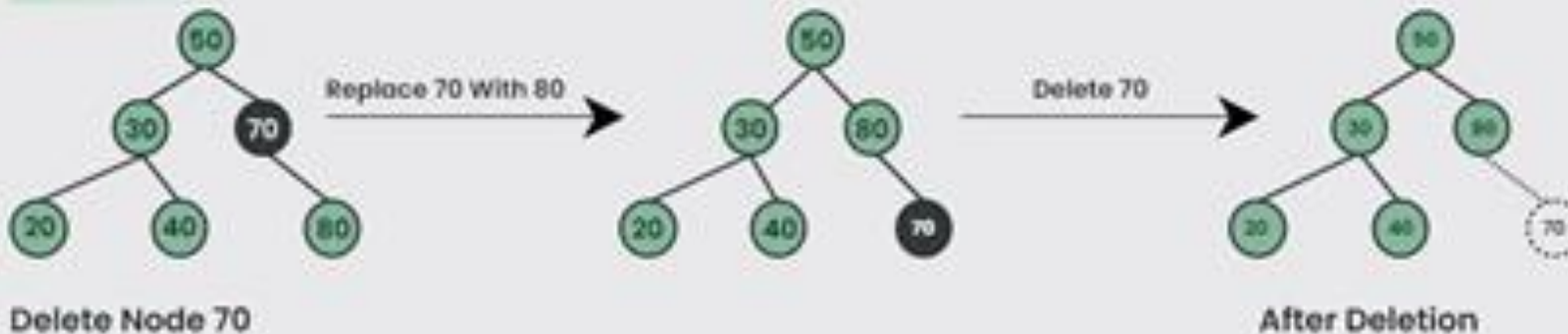
Delete

الحذف

### Case 1 : Delete A Leaf Node In BST



### Case 2: Delete A Node With Single Child In BST



1-2 - إذا أعاد TreeSearch عقدة خارجية (ورقة)، عندها يتم وضع المؤشر الذي يشير إليها يساوي null ونعيد العقدة المحذوفة لخزان العقد الفارغة.

## Delete

## الحذف

3-2- إذا كانت العقدة المرغوب حذفها  $w$  تملك ابنين، لانستطيع ببساطة حذف العقدة  $w$  من  $T$  لأن ذلك سيؤدي إلى إنشاء فجوة أو ثقب في  $T$ . بدلاً من ذلك فإننا نقوم بما يلي (الشكل 5-7):

○ نقوم بإيجاد العقدة الخلف Successor للعقدة المحذوفة وتكون أول عقدة داخلية مثل  $y$  تلي  $w$  في تجول مرتب عبر  $T$ . تكون العقدة  $y$  هي العقدة التي تقع في أقصى اليسار من الشجرة الفرعية اليمينية لـ  $w$ ، ويتم إيجادها من خلال الذهاب إلى الابن اليميني لـ  $w$  أولاً ومن ثم هبوط  $T$  باتجاه اليسار، (عندما نجد عقده ابناً اليساري NULL تكون الخلف. بتتبع الأبناء اليساريين. الابن اليساري  $x$  لـ  $y$  هو أيضاً عقدة خارجية تلي مباشرة العقدة  $w$  في التجول المرتب عبر  $T$ .

○ نقوم بحفظ العنصر المخزن في  $w$  في متحول مؤقت  $t$ ، ونسخ القيمة المخزنه في  $y$  إلى  $w$ . إن هذا التصرف له مفعول حذف العنصر السابق المخزن في  $w$ .

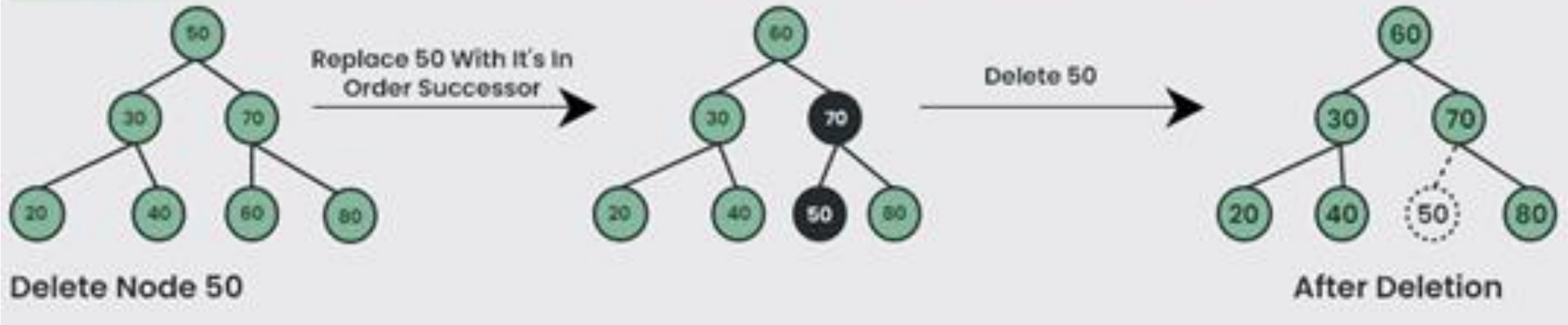
○ نقوم بحذف العقد  $y$  من خلال استدعاء تابع الحذف على  $T$ . هذا التصرف يستبدل  $y$  بأبنها اليميني إن كان لها ابن يمين.

○ نقوم بإعادة العنصر المخزن سابقاً في  $w$  والذي قمنا بحفظه في المتحول المؤقت  $t$ .

وكما في حالتى البحث والحشر، فإن خوارزمية الحذف تتجول عبر مسار من الجذر إلى عقدة خارجية، وتقوم ربما بنقل عنصرين عقدتين من هذا المسار، ومن ثم تنفذ عملية `removeExternal` عند تلك العقدة الخارجية.

ملاحظة: قد تجد في بعض المراجع إستبدال العقدة المحذوفة بأكبر قيمة في الفرع اليساري وبمحاكات مشابهه يمكن الوصول لها.

### Case 3 : Delete A Node With Both Children In BST



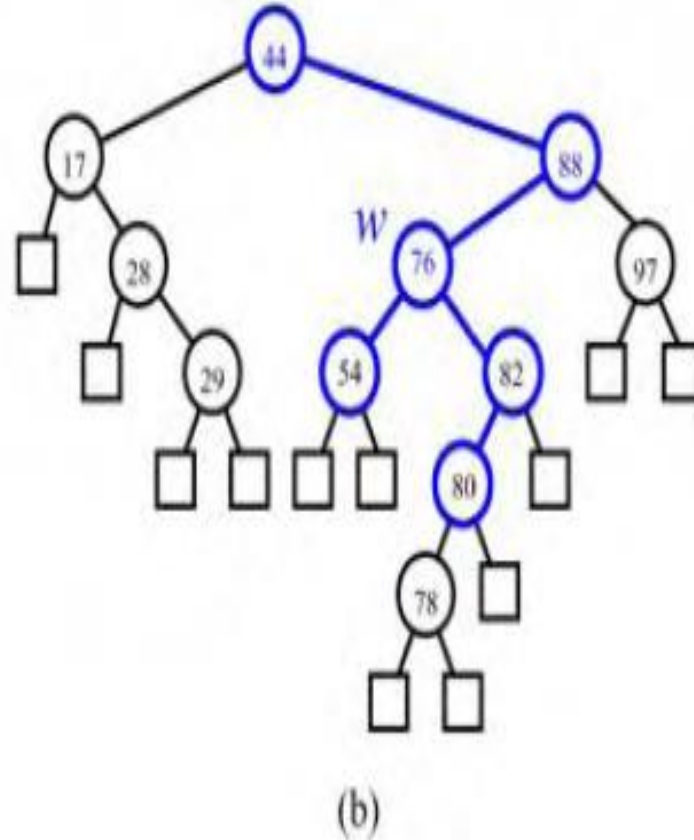
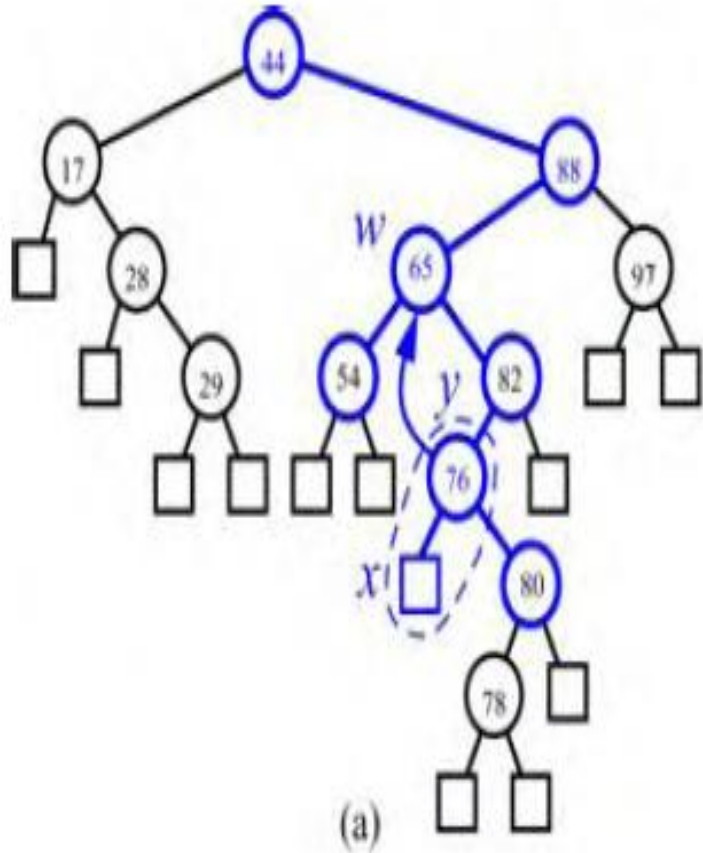
2-2- لعنصر المراد حذفه (ذو المفتاح 50) مخزن في الجزروهي عقدة  $w$  ونشيرلها ب  $r$  ذات إبنين لذلك نبحث عن Successor وهو العنصر في الفرع اليميني و اقصى اليسارويتحقق بالإتجاه أول مره لليمين ونبحث عن العنصر الذي مؤشره اليساري NULL وفي حالتنا 50 وهي عقدة خارجيه وهنا نشيرلها ب  $y$  وننسخ محتواها 50 إلى  $w$  ونقوم بحذف  $y$  وكذلك المؤشر  $r$ .

## Update Operations

## طرائق التعديل ( الحذف ) 5

Removal

الحذف



تابع إيجاد أصغر العناصر:

Node \*findmin( Node \*r)

```

{
  if (r==NULL) return NULL;
  else if (r->left==NULL) return r;
  else return findmin(r->left);
}
  
```

الشكل 5-7- الحذف من شجرة البحث الثنائية المبينة في الشكل 3-7 حيث العنصر المراد حذفه (ذو المفتاح 65) مخزن في عقدة w كلا إبنها داخليين (a قبل، b بعد).



## Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order

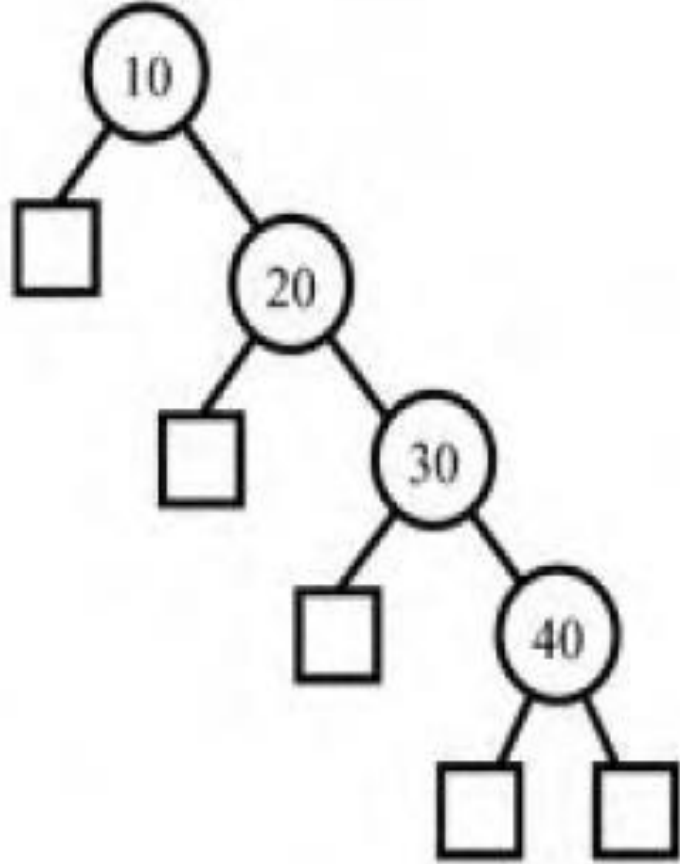
Preorder (PLR) *At first visit the **root** then traverse **left subtree** and then traverse the **right subtree**.*

Postorder (LRP) *:At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**.*

Inorder (LPR) *: At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**.*

## Performance of a Binary Search Tree

## أداء شجرة بحث ثنائية



إن تحليل خوارزميات البحث، الحشر والحذف متشابه. نحن ننفق زمناً  $O(1)$  عند كل عقدة نزورها، وفي الحالة الأسوأ، عدد العقد التي تتم زيارتها تتناسب مع الارتفاع  $h$  للشجرة  $T$ . وبالتالي، في قاموس  $D$  تم تحقيقه بشجرة بحث ثنائية  $T$  فإن الطرائق  $find$ ،  $insert$ ، و  $remove$  تنفذ في زمن  $O(h)$ ، حيث  $h$  هو ارتفاع  $T$ . وبالتالي، إن شجرة البحث الثنائية  $T$  هي تحقيق فعال أو مجدي لقاموس ذي  $n$  عنصر فقط إذا كان ارتفاع الشجرة صغيراً. في الحالة الأفضل، يكون ارتفاع  $T$  هو  $h = \log(n+1)$  الأمر الذي ينتج أداء ذو زمن لوغاريتمي لجميع عمليات القاموس. في حين أنه، في الحالة الأسوأ، يكون ارتفاع  $T$  هو  $n$ ، وفي هذه الحالة قد يكون أفضل أن نستخدم تحقيقاً للقاموس على شكل لائحة مرتبة.

تحدث الحالة الأسوأ عندما نقوم بإدخال سلسلة من العناصر ذات مفاتيح بترتيب متزايد أو متناقص (الشكل 6-7).

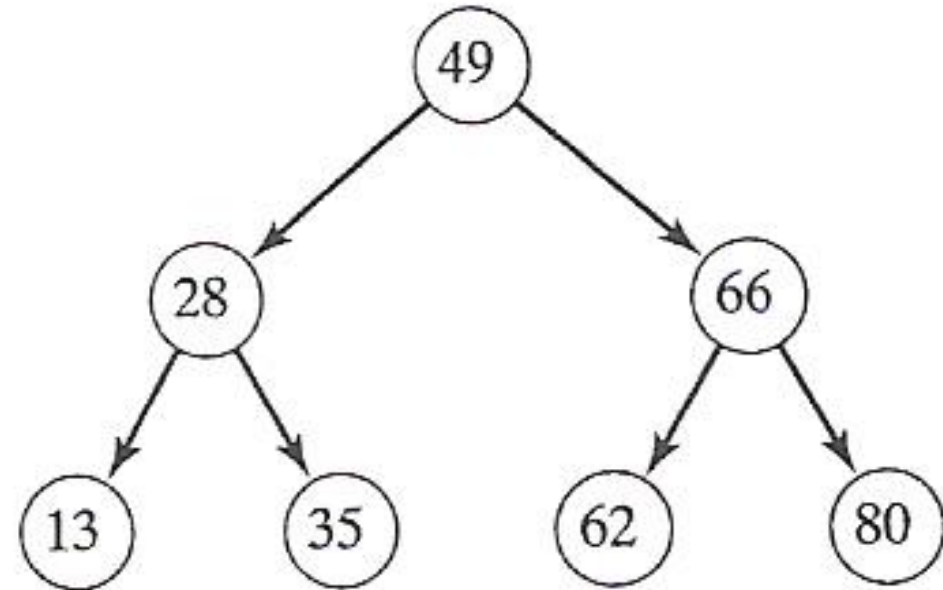
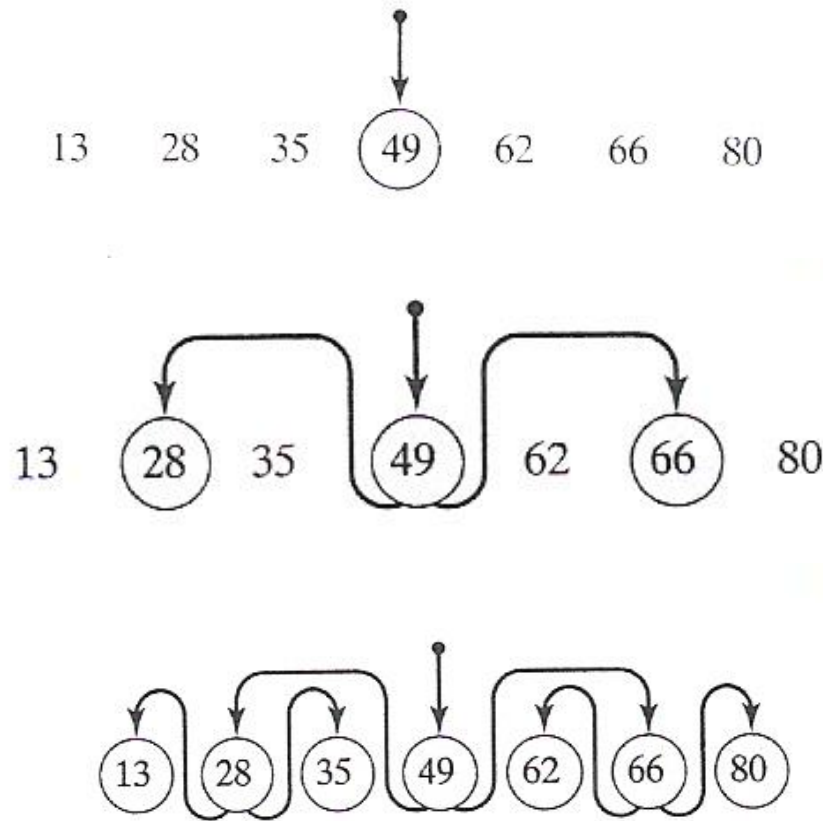
الشكل 6-7- مثال لشجرة ثنائية ذات ارتفاع خطي.



# Binary Search Tree Applications

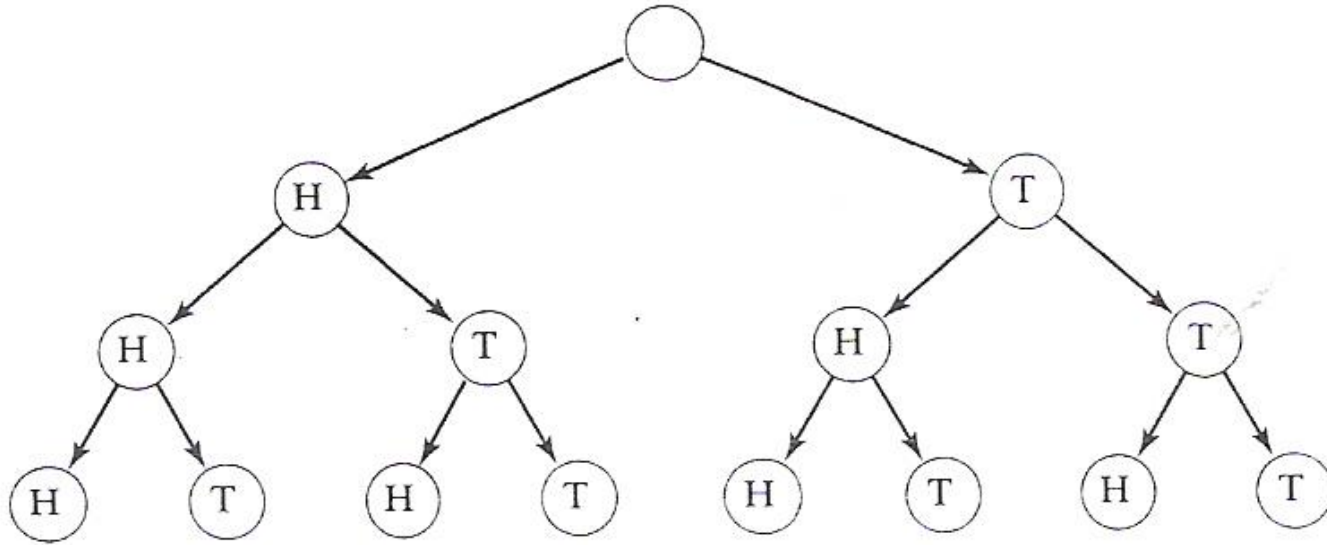
## تطبيقات بحث ثنائية 1

إن تمثيل لائحة مرتبة كشجرة ثنائية يذكرنا بعملية البحث الثنائي حيث أن mad تمثل العقد على أن يتم التجوال بكلا الجزئين.



الشكل 7-7- يمثل شجرة البحث الثنائي.

شجرة ثنائية لتمثيل الاحتمالات الممكنة لعملية رمي قطعة نقدية ثلاث مرات (وتطبق على التجارب والاختبارات التي يكون لها احتمالين ممكنين (مثل on أو off ، 0 أو 1 ، false أو true ، down أو up)).

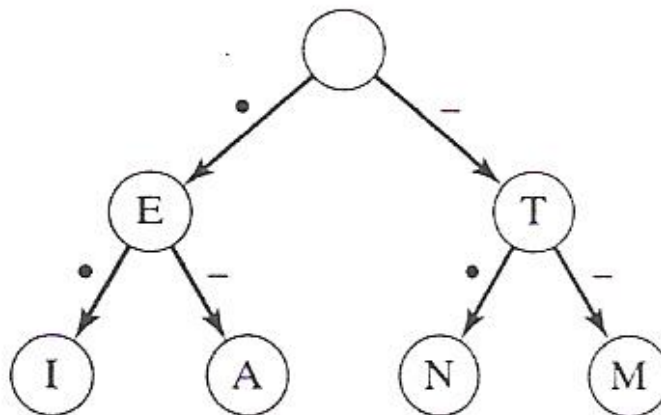


يمثل كل مسار بدءاً من الجذر وحتى إحدى الأوراق ناتجاً محتملاً مثل THT (نقش ثم طرة ثم نقش) كما هو مبين في المخطط.

الشكل 7-8- يمثل شجرة البحث الثنائي رمي قطعة نقدية ثلاث مرات.

في مسائل التشفير مثل تشفير وفك تشفير coding and decoding الرسائل المرسله باستخدام شيفرة مورس Morse code، وفي هذه الطريقة يتم تمثيل الحروف كتال من النقط والشرطات كما هو مبين في الجدول التالي:

|        |        |        |
|--------|--------|--------|
| A ·—   | M—     | Y—     |
| B —··· | N —·   | Z —··· |
| C —··· | O —··· | 1 ·—   |
| D —··  | P ·—   | 2 ·—   |
| E ·    | Q —··· | 3 ·—   |
| F ···  | R ···  | 4 ···  |
| G —··  | S ···  | 5 ···  |
| H ···  | T —    | 6 —··· |
| I ··   | U ·—   | 7 —··· |
| J —··· | V —··  | 8 —··· |
| K —··  | W ·—   | 9 —··· |
| L ···  | X —··· | 0 —··· |



تستخدم العقد في الشجرة الثنائية لتمثيل الحروف وكل قوس من عقدة إلى أبنائها تميز إما بنقطة أو شرطة بحسب فيما إذا كانت تقود إلى ابن يميني أو ابن يساري على التوالي والتالي فإن أجزاء الشجرة الخاصة بشيفرة مورس هي:

تتالي النقط والشرطات المميز للمسار من الجذر إلى عقدة ما يمثل شيفرة مورس لذلك الحرف

الشكل 7-9- يمثل جزء من شيفرة مورس .

تستخدم الأشجار الثنائية في العديد من مسائل التشفير وفك التشفير.

- شيفرة مورس، التي تمثل كل حرف بتتال من النقط والشرطات. تستخدم شيفرة مورس متتاليات متغيرة الطول.

- ترميزات الـ ASCII، EBCDIC، UNICODE التي يكون فيها طول الرمز هو نفسه لكل المحارف.

- شيفرة هوفمان Huffman code والتي تستخدم رموزاً بأطوال متغيرة.

الفكرة الأساسية في طرق التشفير متغير الطول هي استخدام شيفرات أقصر للأحرف الأكثر استخداماً، وشيفرات أطول للأحرف

الأقل استخداماً. على سبيل المثال 'E' في شيفرة مورس هي عبارة عن نقطة واحدة، في حين أن 'Z' ممثلة بالشكل (-.-).

إن الهدف هو تقليل الطول المتوقع لشيفرة الحرف، هذا يقلل عدد البتات الواجب إرسالها عند نقل الرسائل المشفرة. إن هذه

الطرق في الترميز المتغيرة الطول مفيدة عند ضغط البيانات لأنها تقلل عدد البتات الواجب تخزينها

لتوضيح المسألة بشكل أدق، نفرض أننا أعطينا مجموعة المحارف  $\{C_1, C_2, \dots, C_n\}$  وأن أوزاناً محددة أرفقت مع هذه المحارف

$w_1, w_2, \dots, w_n$ ، حيث  $w_i$  هي الوزن المرفق بالمحرف  $C_i$  وهو يدل على تكرار استخدام هذا المحرف في الرسائل المراد تشفيرها. إذا كانت

$l_1, l_2, \dots, l_n$  هي أطوال الشيفرات الخاصة بالأحرف  $C_1, C_2, \dots, C_n$  على التوالي، **عندئذ فإن الطول المتوقع لشيفرة أي من الأحرف**

**تُحسب كما يلي:**

$$\text{expected length} = w_1 l_1 + w_2 l_2 + \dots + w_n l_n = \sum_{i=1}^n w_i l_i$$

كمثال بسيط، لندرس الأحرف الخمسة A,B,C,D,E ونفرض أنها تحصل بالأوزان التالية:

| character | A   | B   | C   | D    | E    |
|-----------|-----|-----|-----|------|------|
| weight    | 0.2 | 0.1 | 0.1 | 0.15 | 0.45 |

في شيفرة مورس إذا استبدلنا النقطة بصفر والشرطة بواحد فإن هذه المحارف مشفرة كما يلي:

| character | A  | B    | C    | D   | E |
|-----------|----|------|------|-----|---|
| code      | 01 | 1000 | 1010 | 100 | 0 |

وبالتالي فإن الطول المتوقع لكل من هذه الأحرف الخمسة في هذه الطريقة هو:

$$0.2 \times 2 + 0.1 \times 4 + 0.1 \times 4 + 0.15 \times 3 + 0.45 \times 1 = 2.1$$

قابلية فك التشفير مباشرة immediate decidability:

الميزة المفيدة الأخرى لبعض طرق التشفير هو أنها قابلة لفك التشفير مباشرة immediately decodable. وهذا يعني ليس هناك تتالي من البتات يمثل محرفاً يمكن أن يكون جزءاً سابقاً prefix من تتالي أطول لأحرف أخرى. وبالتالي عند استقبال تتالي من البتات فهو شيفرة لحرف، فهو يمكن أن يفك تشفيره إلى ذلك المحرف مباشرة بدون انتظار فيما إذا كانت الخانات التي تلي تجعل منه شيفرة لمحرف آخر.

لاحظ أن شيفرة مورس السابقة ليست قابلة لفك التشفير مباشرة، لأنه، وعلى سبيل المثال، الشيفرة للمحرف E هي ( 0 ) وهي جزء سابق من شيفرة الحرف A أي ( 01 )، وكذلك شيفرة الحرف D أي ( 100 ) هي جزء سابق من شيفرة الحرف B أي ( 1000 ). تستخدم شيفرة مورس من أجل عملية فك التشفير خانة إضافية هي الفراغ للفصل بين المحارف.

### شيفرة هوفمان Huffman codes:

يمكن استخدام الخوارزمية التالية المقدمة من قبل D.A.Huffman في عام 1952 لبناء طريقة تشفير قابلة للفك مباشرة وتملك طولاً أصغرياً متوقعاً لشيفرة كل حرف:

#### HUFFMAN'S ALGORITHM

1-initialize a list of one-node binary trees containing the weights  $w_1, w_2, \dots, w_n$  one for each of the characters  $C_1, C_2, \dots, C_n$ .

2-do the following  $n-1$  times:

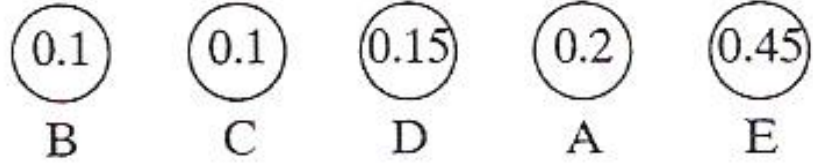
a-find two trees  $T'$  and  $T''$  in this list with roots of minimal weights  $w'$  and  $w''$ .

b-replace these two trees with a binary tree whose root is  $w'+w''$ , and whose subtrees are  $T'$  and  $T''$ , and label the pointers to these subtrees 0 and 1 respectively.

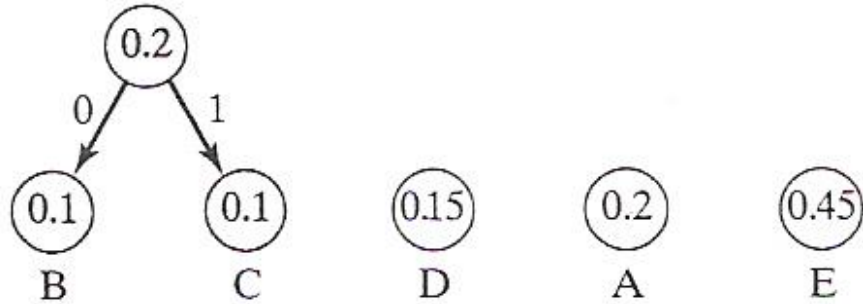
3-the code for character  $C_i$  is the bit string labeling a path in the final binary tree from the root to the leaf for  $C_i$ .



كتوضيح لخوارزمية هوفمان، ندرس مجدداً المحارف A,B,C,D,E بالأوزان المحددة سابقاً. نبدأ ببناء لائحة من أشجار ثنائية مؤلفة من عقدة واحدة لكل محرف:



لشجرتين الأولى والثانية التي سيتم اختيارهما هي تلك الممثلة للمحارف B و C وذلك لأنها تملك الأوزان الأصغر. وبتجميع هاتين الشجرتين بشكل شجرة تملك الوزن  $0.1+0.1=0.2$  ولها شجرتين كشجرتين جزئيتين:



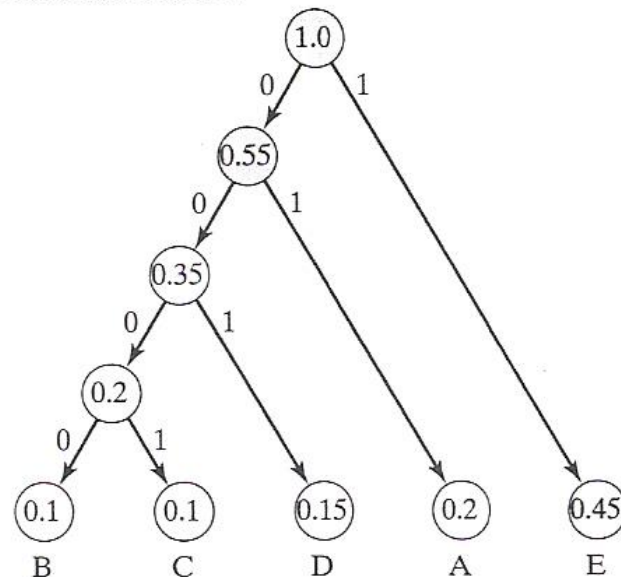
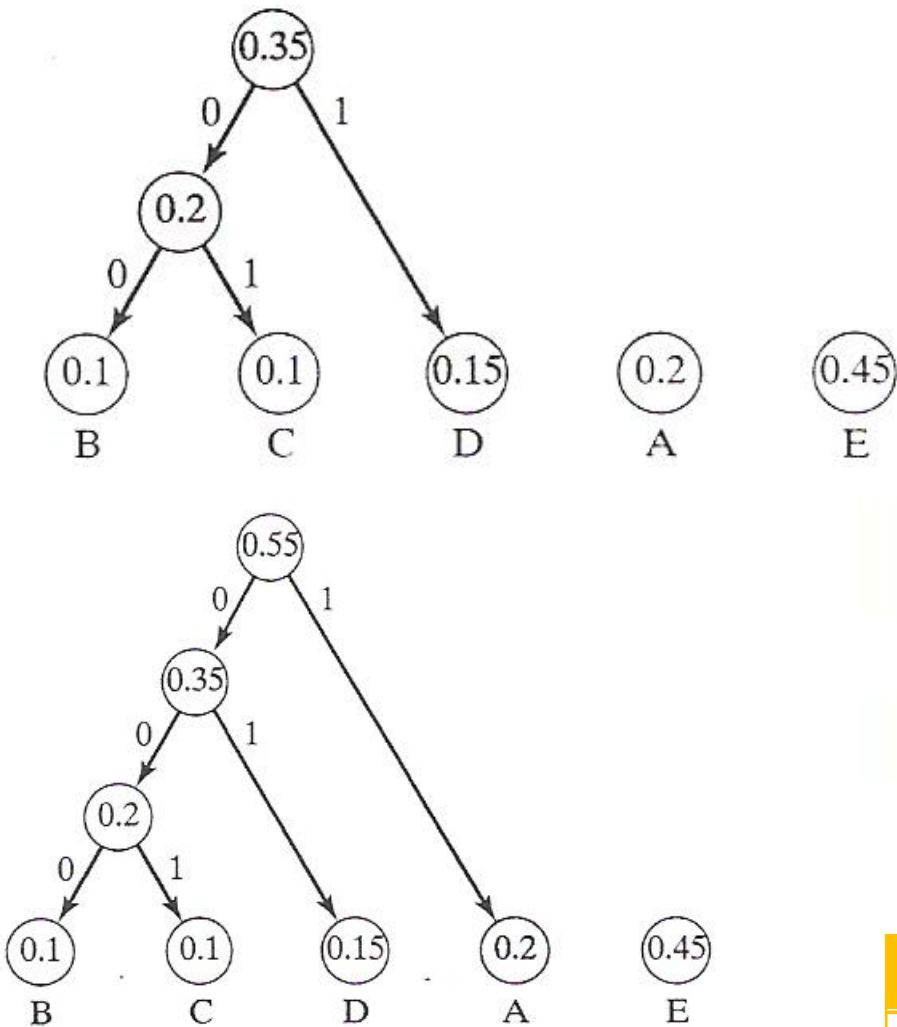
نختار من جديد من لائحة الأشجار الأربعة وزنين أصغريين، الأولى والثانية (أو الثانية والثالثة) ونقوم باستبدالهما كما في المرة السابقة بشجرة أخرى وزنها  $0.2+0.15=0.35$  ولها هاتين الشجرتين كشجرتين جزئيتين:



## application of binary trees: Huffman codes

## تطبيق على الأشجار الثنائية: شيفرة هوفمان 5

وبالمثل نتابع فنحصل على الشجرة التالية:



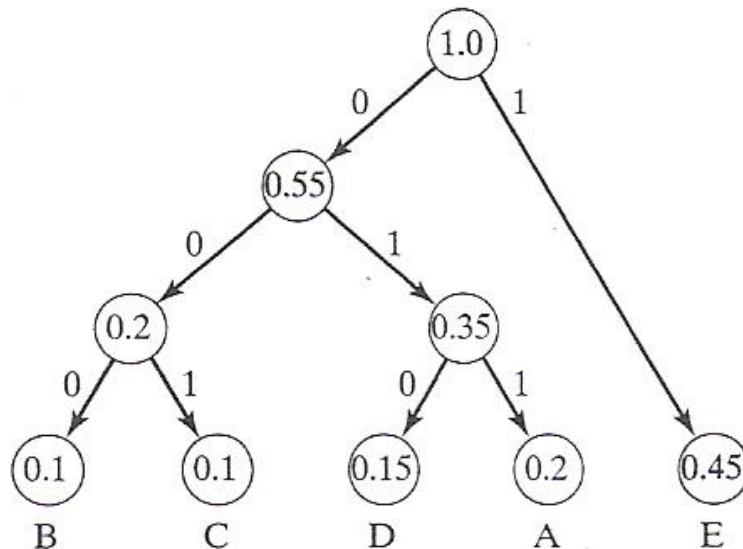
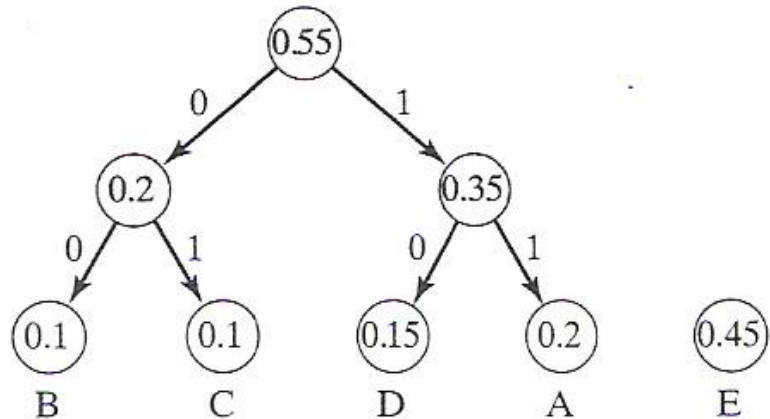
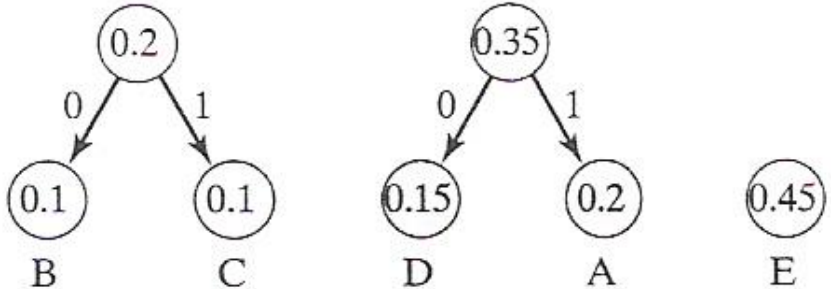
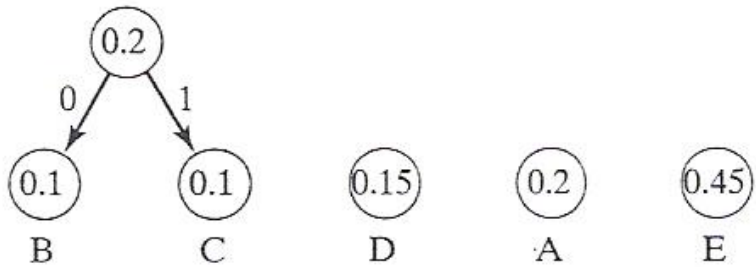
وبالتالي تكون شيفرة هوفمان الناتجة عن هذه الشجرة لهذه الأحرف كما يلي:  
وكما بينا سابقاً فإن الطول المتوقع لشيفرة كل حرف هو 2.1.

| character    | A  | B    | C    | D   | E |
|--------------|----|------|------|-----|---|
| Huffman code | 01 | 0000 | 0001 | 001 | 1 |

## application of binary trees: Huffman codes

## تطبيق على الأشجار الثنائية: شيفرة هوفمان 6

يمكن تكوين شيفرة هوفمان بطريقة أخرى  
نكتفي بعرضها من خلال الرسوم كما يلي:



وتكون الشيفرة الناتجة كما يلي:

| character    | A   | B   | C   | D   | E |
|--------------|-----|-----|-----|-----|---|
| Huffman code | 011 | 000 | 001 | 010 | 1 |

إن قابلية فك التشفير مباشرة واضحة في شيفرة هوفمان. كل محرف مرتبط بعقدة ورقة في شجرة هوفمان وهناك مسار وحيد من الجذر إلى كل ورقة. وبالتالي ليس هناك تتالي من البتات يتضمن شيفرة لمحرف يمكن أن يكون جزءاً سابقاً من تتالي أطول من البتات لمحرف آخر.  
إن خورازمية فك التشفير سهلة جداً بسبب ميزة قابلية فك التشفير مباشرة:

### HUFFMAN DECODING ALGORITHM

1-initialize pointer p to the root of the Huffman tree.

2-while the end of the message string has not been reached , do the following:

a- let x be the next bit in the string.

b- if  $x=0$  then set p equal to its left child pointer.

else set p equal to its right child pointer.

c-if p points to a leaf then

i-display the character associated with that leaf.

ii-reset p to the root of the Huffman tree.

## application of binary trees: Huffman codes

## تطبيق على الأشجار الثنائية: شيفرة هوفمان 8

كتوضيح، لنفرض أن الرسالة التالية:

0 1 0 1 0 1 1 0 1 0

قد استلمت، وأن هذه الرسالة مشفرة باستخدام شجرة هوفمان الثانية المبينة سابقاً، يتبع المؤشر المسار التالي 010 من جذر هذه الشجرة يتم اكتشافه وجعله جذر الشجرة:

0 1 0 1 0 1 1 0 1 0

D

0 1 0 1 0 1 1 0 1 0

D

E

الخانة التالية 1 تقود مباشرة إلى المحرف E:

0 1 0 1 0 1 1 0 1 0

D

E

A

D

وهكذا إلى أن تصبح الرسالة من الشكل:

```
// C++ program to demonstrate insertion
// in a BST recursively
#include <iostream>
using namespace std;
class BST {
public: int data;  BST *left, *right;

BST();           // Default constructor.
BST(int);       // Parameterized constructor.
BST* Insert(BST*, int); // Insert function.
void Inorder(BST*); // Inorder traversal.
void Postorder(BST*); // Postorder traversal
void Preorder(BST*); // Preorder traversal.
BST* deleteNode(BST*, int); // Delete function.
};
```

```
#include <iostream>
#include "BST.h"
using namespace std;
// Default Constructor definition.
BST::BST(): data(0) , left(NULL), right(NULL){}
// Parameterized Constructor definition.
BST::BST(int value){data = value; left = right = NULL;}
// Insert function definition.
BST* BST::Insert(BST* root, int value)
{if (!root) {// Insert the first node, if root is NULL.
return new BST(value);}
// Insert data.
if (value > root->data) {root->right = Insert(root->right, value);}
/*Insert right node data, if the 'value' to be inserted is greater than 'root' .Process right nodes.*/
```

```

else if (value < root->data) {
/* Insert left node data, if the 'value' to be inserted is smaller than 'root' node data. Process left nodes.*/
    root->left = Insert(root->left, value);}

    return root;} // Return 'root' node, after insertion

// Preorder, Postorder, Inorder traversal function.This gives data in sorted order.

void BST::Preorder(BST* r) // root ->left->right
    { if (r == NULL) return;    cout << r->data << "\t";    Preorder(r->left);    Preorder(r->right);}

void BST::Postorder(BST* r) // left--> right->root
{ if (r == NULL) return;    Postorder(r->left);    Postorder(r->right);    cout << r->data << "\t";}

void BST::Inorder(BST* root) // left--> root-> right
{if (root == NULL) {return;}Inorder(root->left);    cout << root->data << " ";    Inorder(root->right);}

```



// Function that returns the node with minimum key value found in that tree

```
BST * minValueNode(BST * node)
```

```
{ BST * current = node; // Loop down to find the leftmost leaf
```

```
  while (current && current->left != NULL)    current = current->left;
```

```
  return current;}
```

// Function that deletes the key and returns the new root

```
BST* BST::deleteNode(BST* root, int value)
```

```
{ // base Case
```

```
  if (root == NULL)    return root;
```

```
  // If the key to be deleted is smaller than the root's key,
```

```
  // then it lies in left subtree
```

```
  if (value < root->data)
```

```
    {root->left = deleteNode(root->left, value);}
```

```
  // If the key to be deleted is greater than the root's key,
```

```

// then it lies in right subtree
else if (value > root->data)
    { root->right= deleteNode(root->right, value);}
// If key is same as root's key, then this is the node to be del
else { // Node with only one child or no child
    if (root->left == NULL)           // one child on the right
        { BST * temp = root->right;   free(root);  return temp;}
    else if (root->right == NULL)     // one child on the left
        { BST * temp = root->left;    free(root);  return temp; }
    // Node with two children: Get the inorder successor(smallest in the right subtree)
    BST * temp = minValueNode(root->right);

    // Copy the inorder successor's content to this node
    root->data = temp->data;

```

```
// Delete the inorder successor
```

```
    root->right= deleteNode(root->right, temp->data); }
```

```
    return root;
```

```
}
```

```
#include "BST.h"
```

```
using namespace std;
```

```
// Driver code
```

```
int main(){
```

```
    /* Let us create following BST
```

```
    50
```

```
   /  \
```

```
  30   70
```

```
 / \   / \
```

```
20 40 60 80 */
```



```

BST b, *root = NULL;          root = b.Insert(root, 50);          b.Insert(root, 30);
b.Insert(root, 20);          b.Insert(root, 40);          b.Insert(root, 70);
b.Insert(root, 60);          b.Insert(root, 80);
printf("\n Original BST in Preorder:\t");    b.Preorder(root);
printf("\n Original BST in Postorder:\t");    b.Postorder(root);
printf("\n Original BST: ");                b.Inorder(root);
printf("\n\nDelete a Leaf Node: 20\n");      root = b.deleteNode(root, 20);
printf("Modified BST tree after deleting Leaf Node:\n");    b.Inorder(root);
printf("\n\nDelete Node with single child: 70\n");    root = b.deleteNode(root, 70);
printf("Modified BST tree after deleting single child Node:\n");    b.Inorder(root);
printf("\n\nDelete Node with both child: 50\n");    root = b.deleteNode(root, 50);
printf("Modified BST tree after deleting both child Node:\n");    b.Inorder(root);
system("pause");              return 0;
}

```



Original BST in Preorder: 50 30 20 40 70 60 80

Original BST in Postorder: 20 40 30 60 80 70 50

Original BST in Inorder: 20 30 40 50 60 70 80

Delete a Leaf Node: 20

Modified BST tree after deleting Leaf Node:

30 40 50 60 70 80

Delete Node with single child: 70

Modified BST tree after deleting single child Node:

30 40 50 60 80

Delete Node with both child: 50

Modified BST tree after deleting both child Node:

30 40 60 80

Press any key to continue . . .



| AVLTrees                                    |  |
|---|--|
| 1-  |  |
| 2- Why AVL Trees?.                          |  |
| 3-Operations on an AVL Tree                 |  |
| 4-Rotating the subtrees in an AVL Tree      |  |
| 5- Advantages and Disadvantages of AVL Tree |  |
| 6- Insertion in an AVL Tree                 |  |
| 7-Deletion in an AVL Tree                   |  |
| 8-Searching in an AVL Tree                  |  |
| 9-Illustration of Insertion at AVL Tree     |  |

## References

- Deitel & Deitel, Java How to Program, Pearson; 10th Ed(2015)

<https://www.geeksforgeeks.org> ›



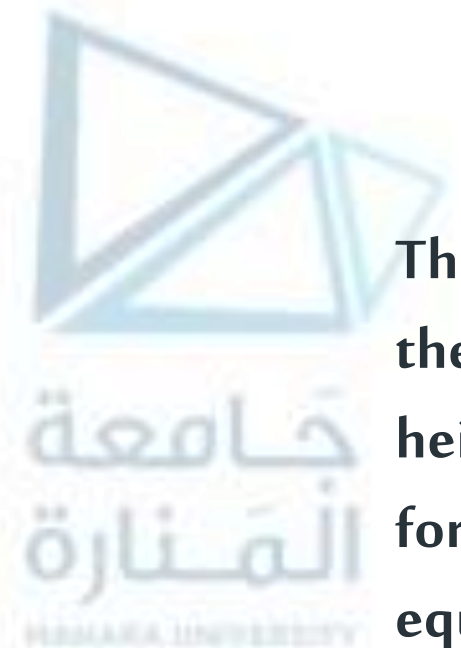
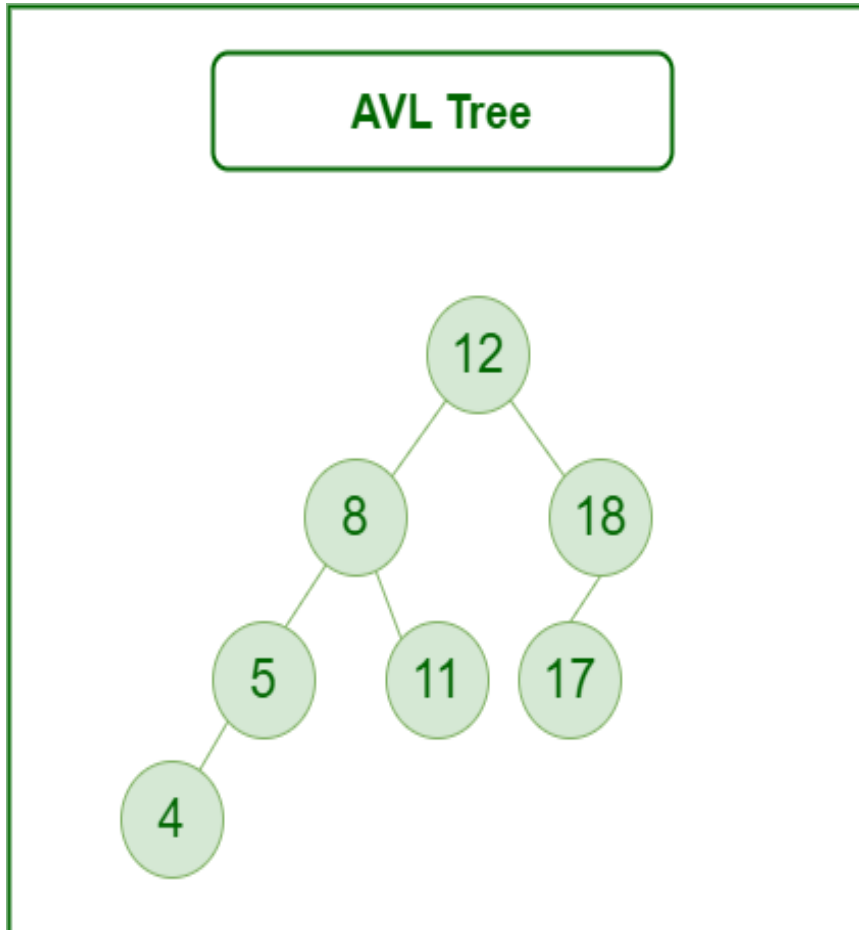
# AVL tree 1

*An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*

The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.

The AVL tree is named after its inventors, Georgy Adelson- Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

## AVL tree 2



The tree on the left is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

# Why AVL Trees ?



## Why AVL Trees?

*Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a skewed Binary tree. If we make sure that the height of the tree remains  $O(\log(n))$  after every insertion and deletion, then we can guarantee an upper bound of  $O(\log(n))$  for all these operations. The height of an AVL tree is always  $O(\log(n))$  where  $n$  is the number of nodes in the tree.*

# Operations on an AVL Tree 1



## Operations on an AVL Tree:

- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

## Rotating the subtrees in an AVL Tree:

An AVL tree may rotate in one of the following four ways to keep itself balanced:

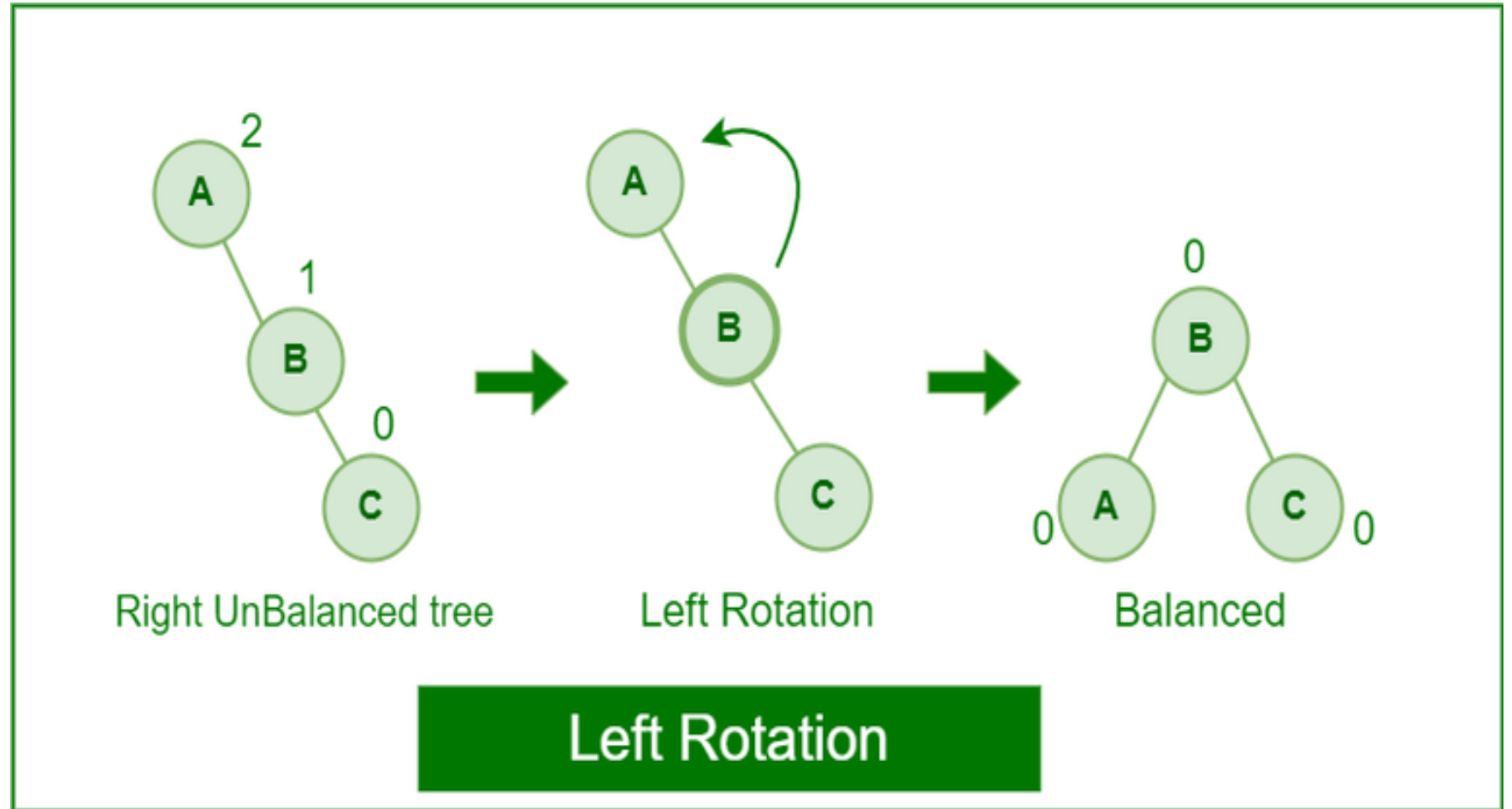
### Left Rotation:

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.

# Operations on an AVL Tree 2

## Left Rotation:

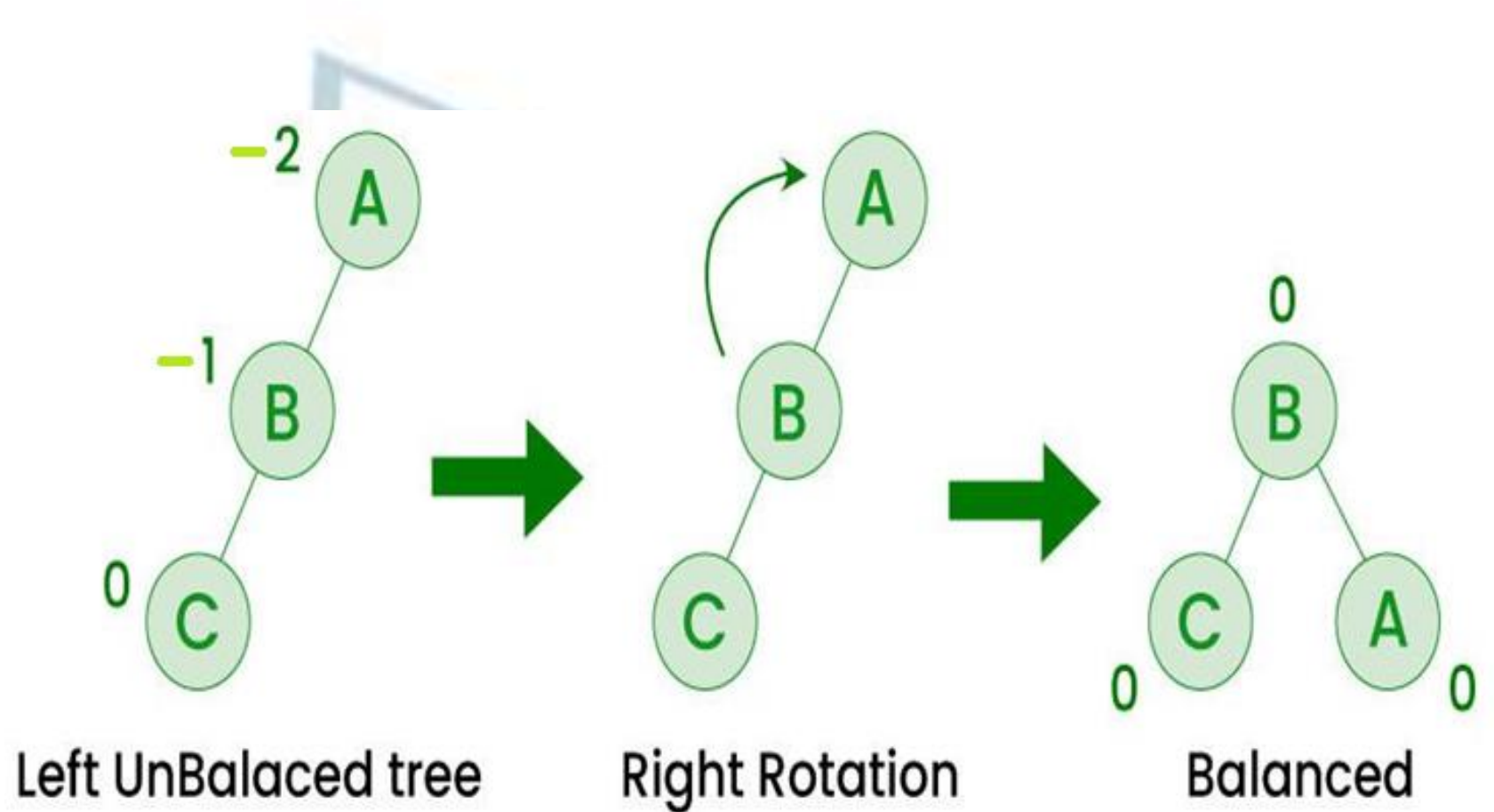
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation (anticlockwise rotation).



# Operations on an AVL Tree 3

## Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation (clockwise rotation).



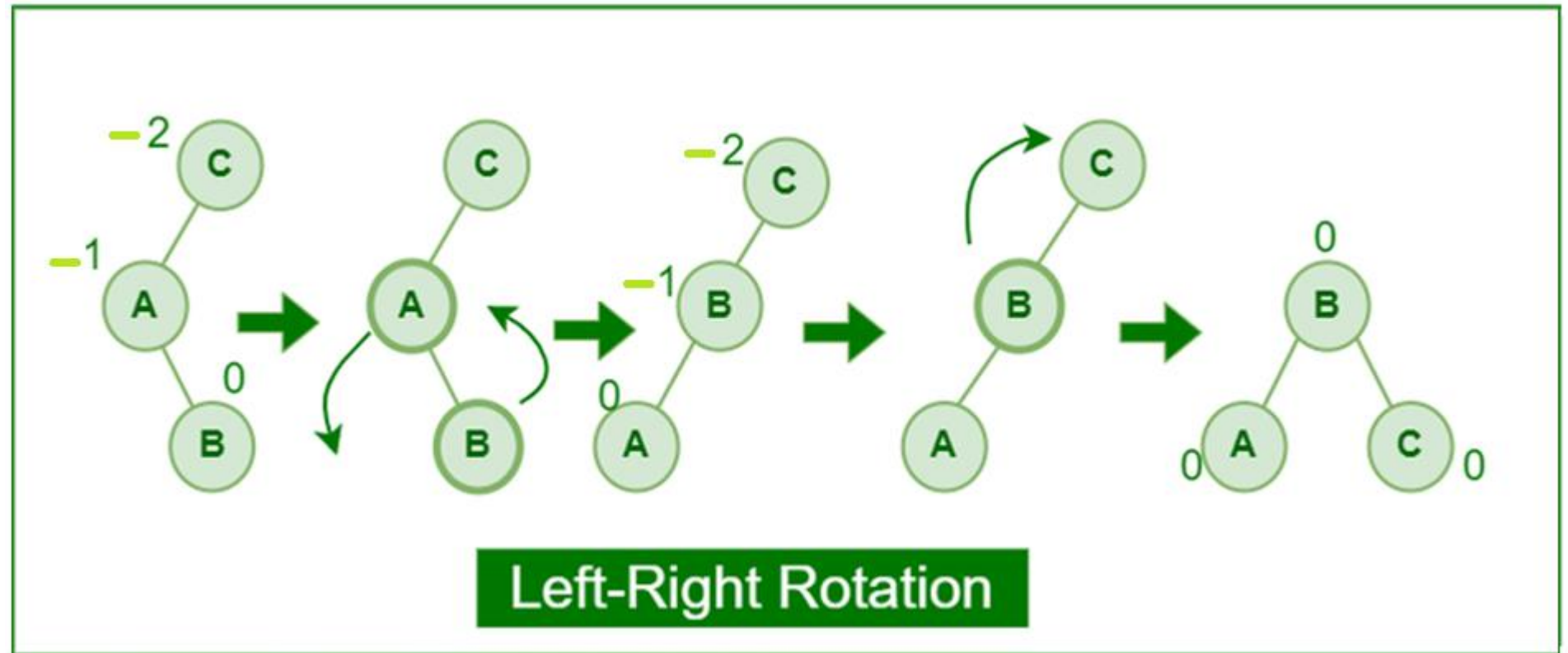


# Operations on an AVL Tree 4

## Left-Right

### Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

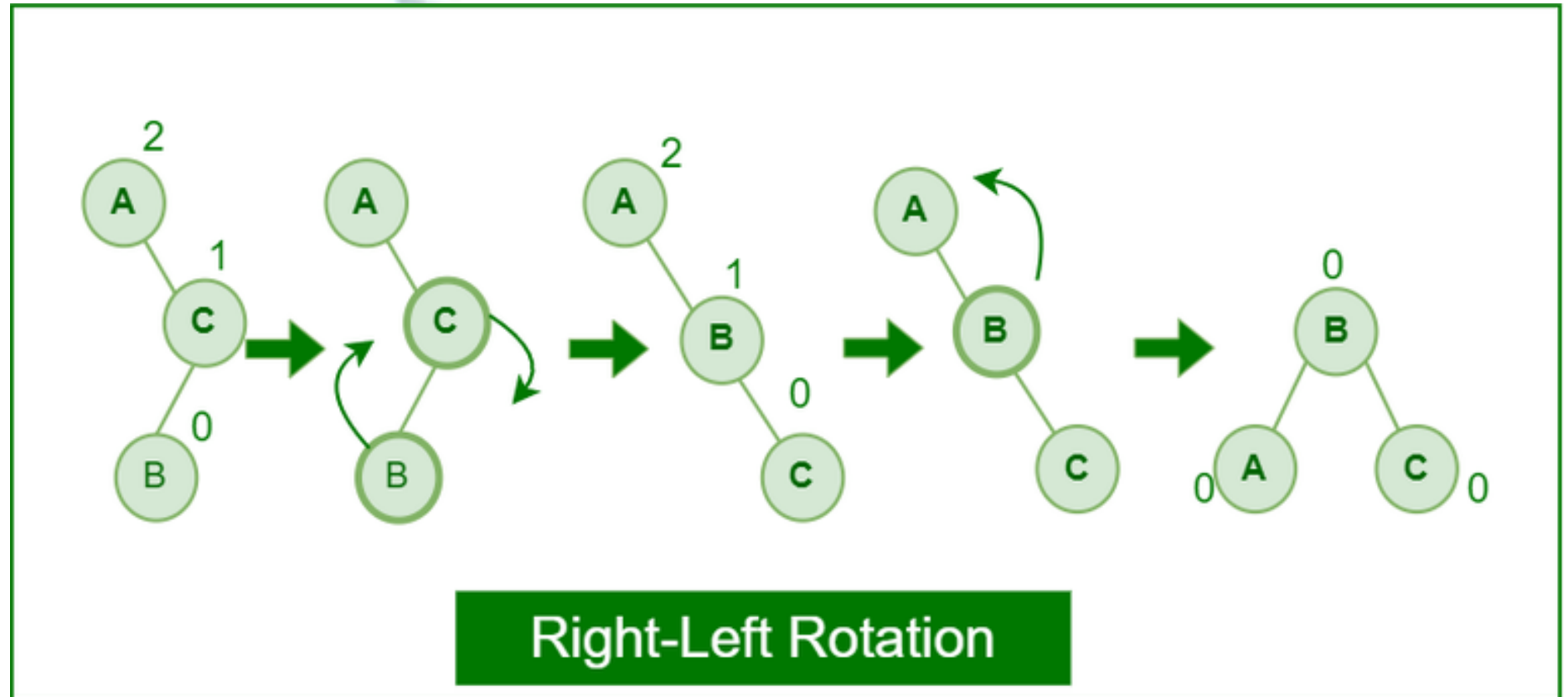


# Operations on an AVL Tree 5

## Right-Left

### Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



# Advantages of AVL Tree 7



## Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed  $\log_2(n)$ , where, N is the total number of nodes in the tree.

# Disadvantages of AVL Tree 8



## Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

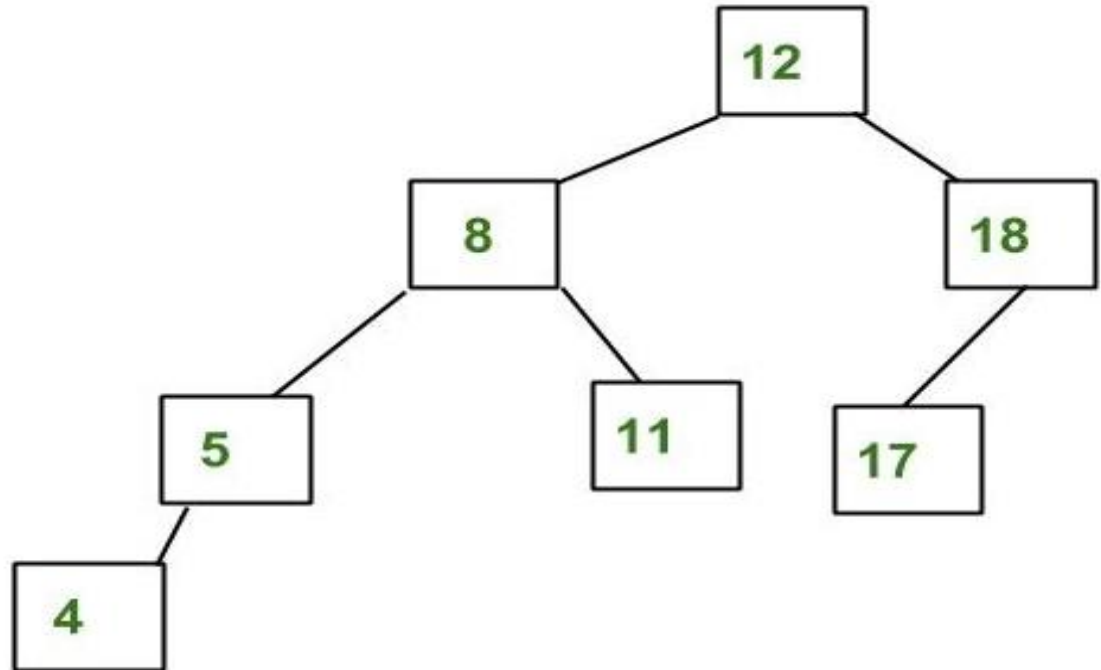
# Insertion in an AVL Tree 1

## Insertion in an AVL Tree

### AVL Tree:

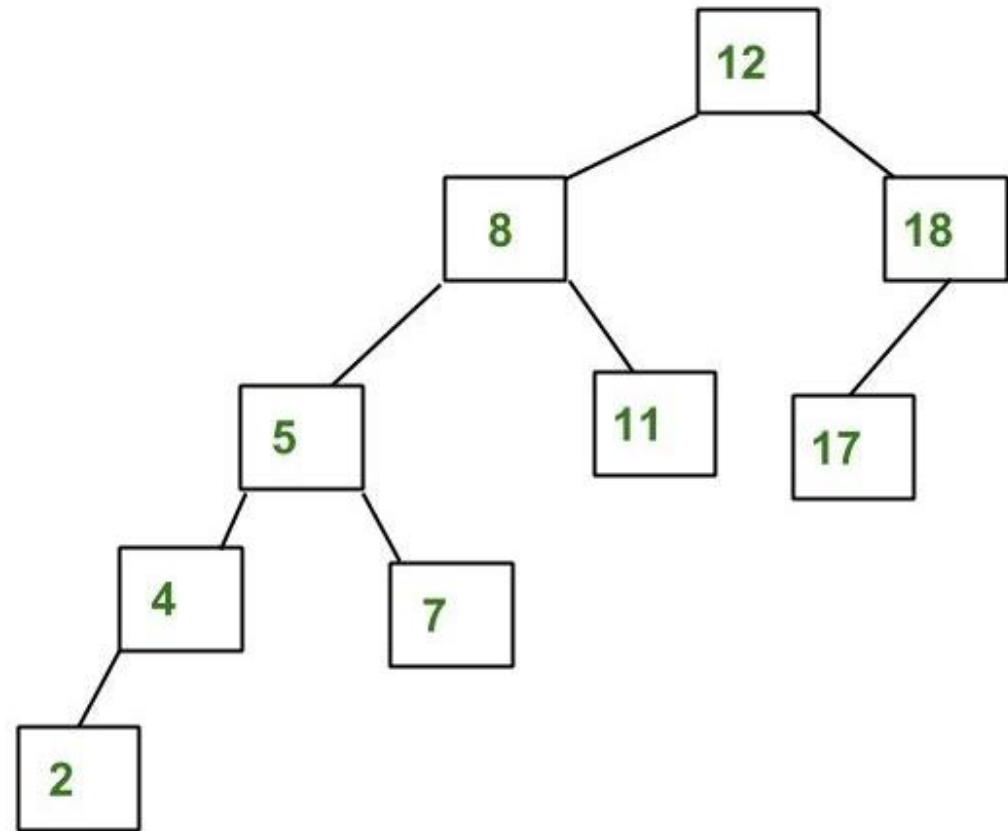
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Insert a node with value 2 The tree is not an AVL tree



## Insertion in an AVL Tree 2

The tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.





# Insertion in an AVL Tree 3



## Insertion in AVL Tree:

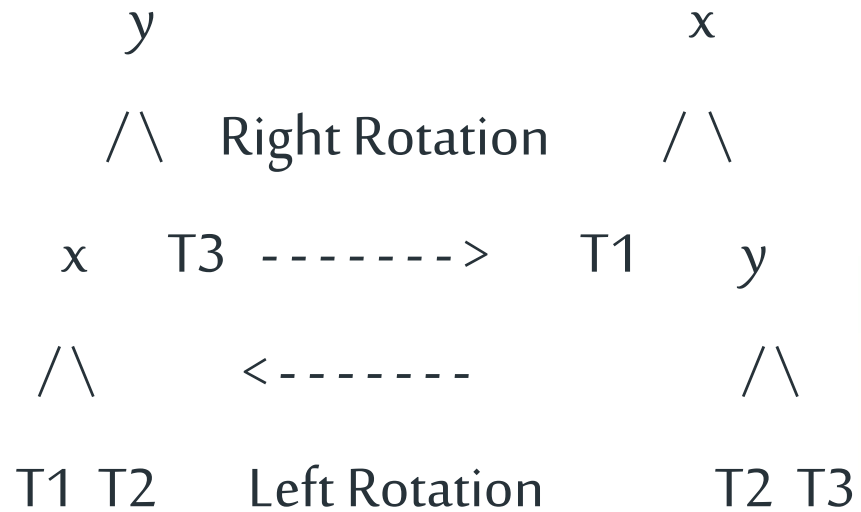
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property ( $\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$ ).

- Left Rotation
- Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with  $y$  (on the left side) or  $x$  (on the right side)

## Insertion in an AVL Tree 4



Keys in both of the above trees follow the following order

$$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$$

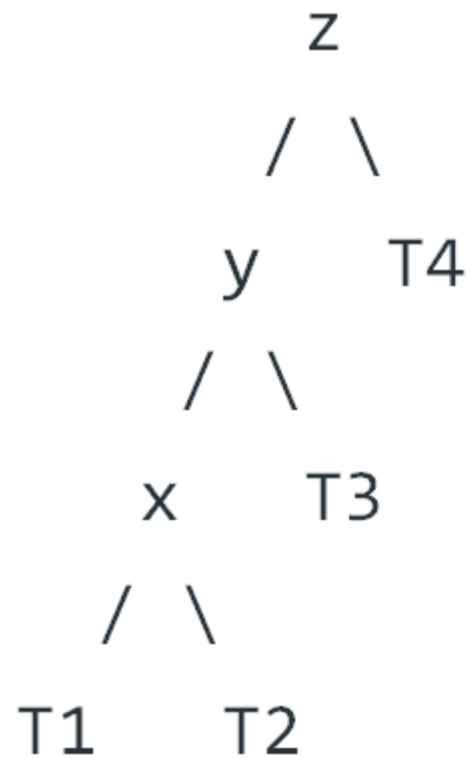
So BST property is not violated anywhere.



## Left Left Case

### 1. Left Left Case

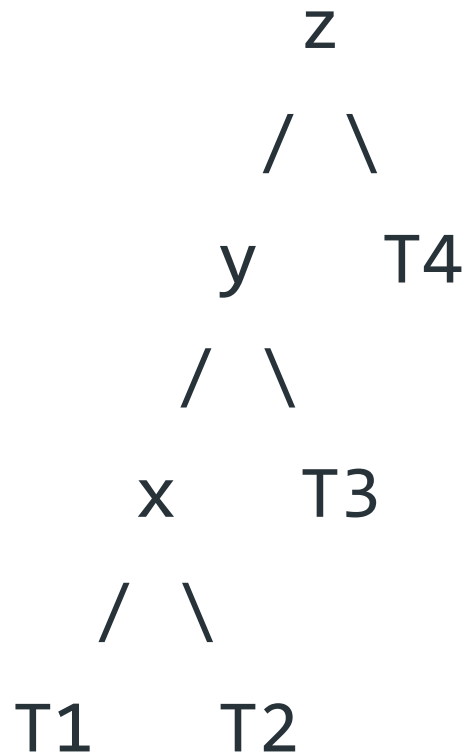
T1, T2, T3 and T4 are subtrees.



# Left Left Case

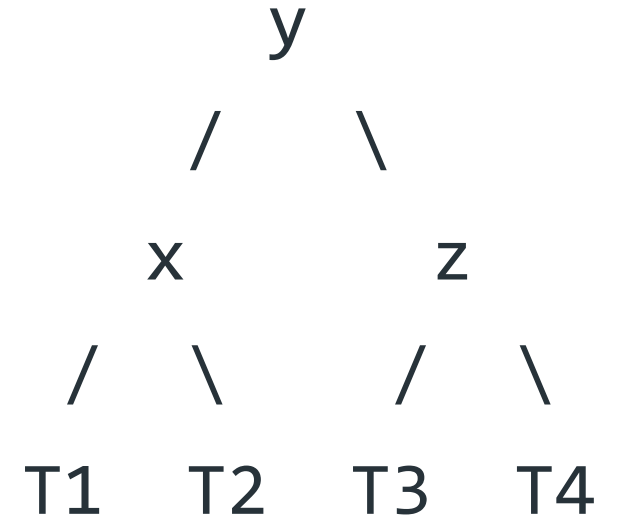
## 1. Left Left Case

T1, T2, T3 and T4 are subtrees.



Right Rotate (z)

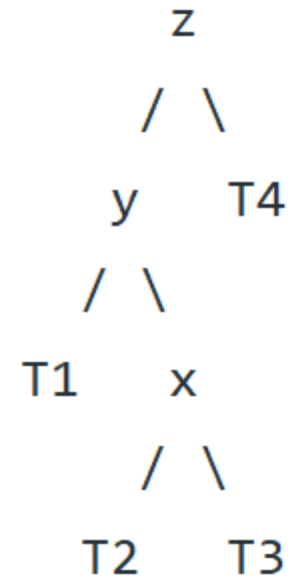
- - - - ->



# Left Right Case



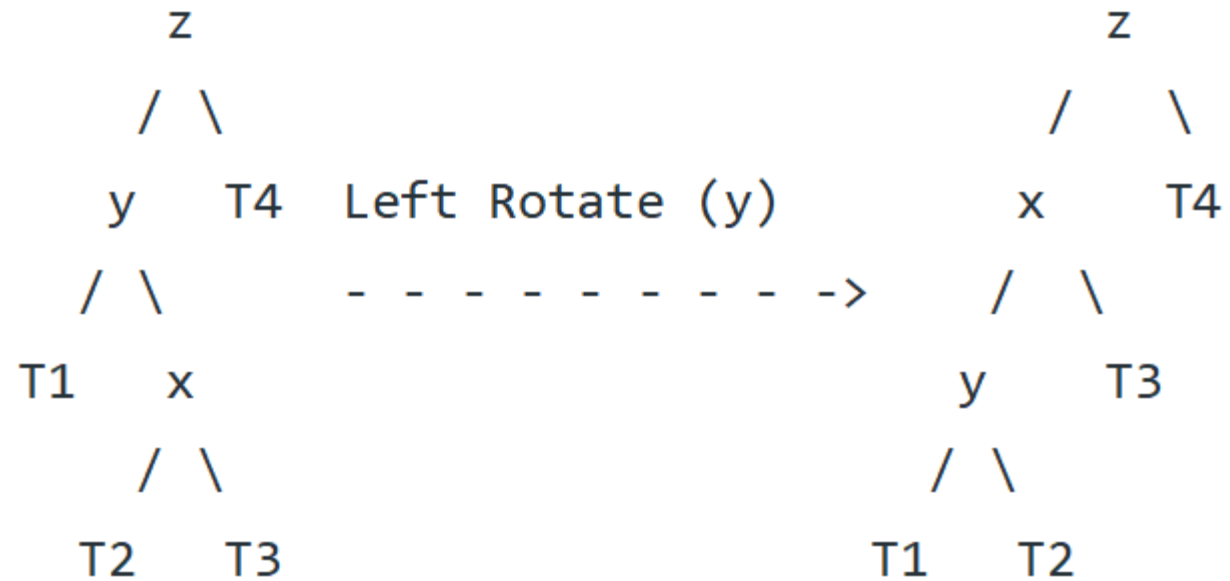
## 2. Left Right Case



# Left Right Case



## 2. Left Right Case

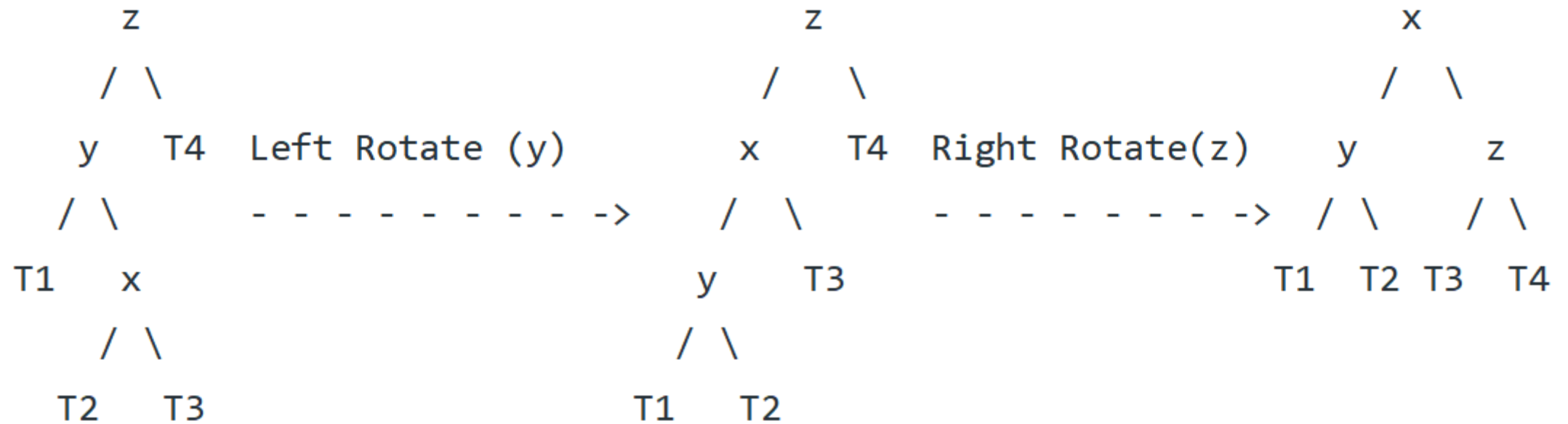




# Left Right Case



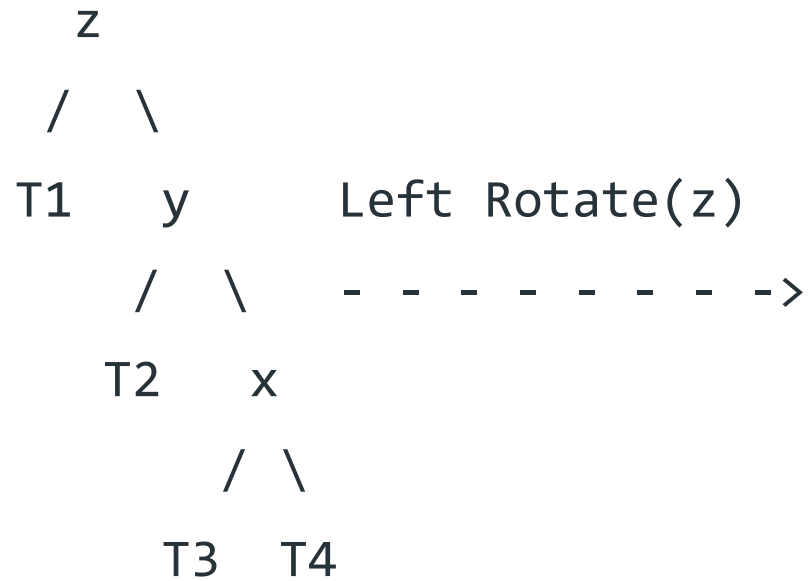
## 2. Left Right Case



# Right Right Case



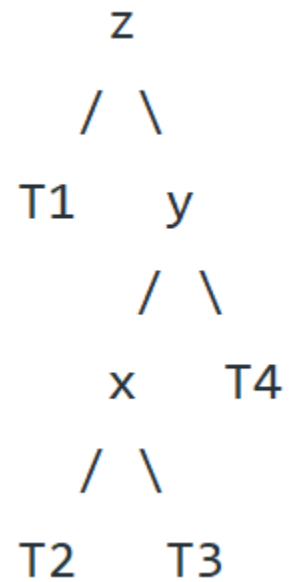
## 3. Right Right Case



# Right Left Case



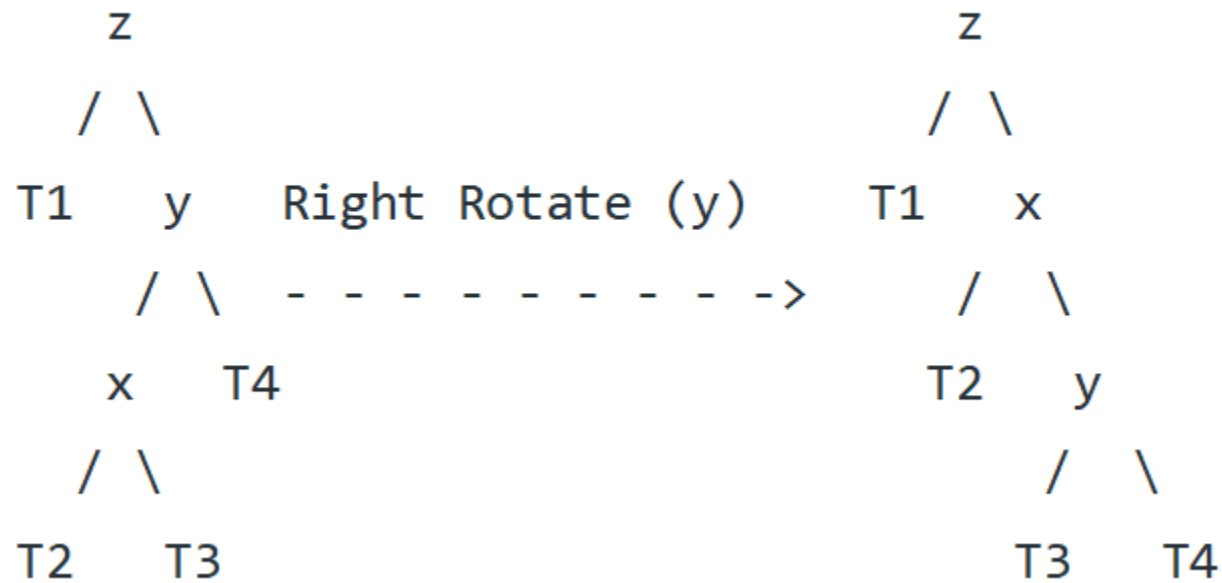
## 4. Right Left Case



# Right Left Case



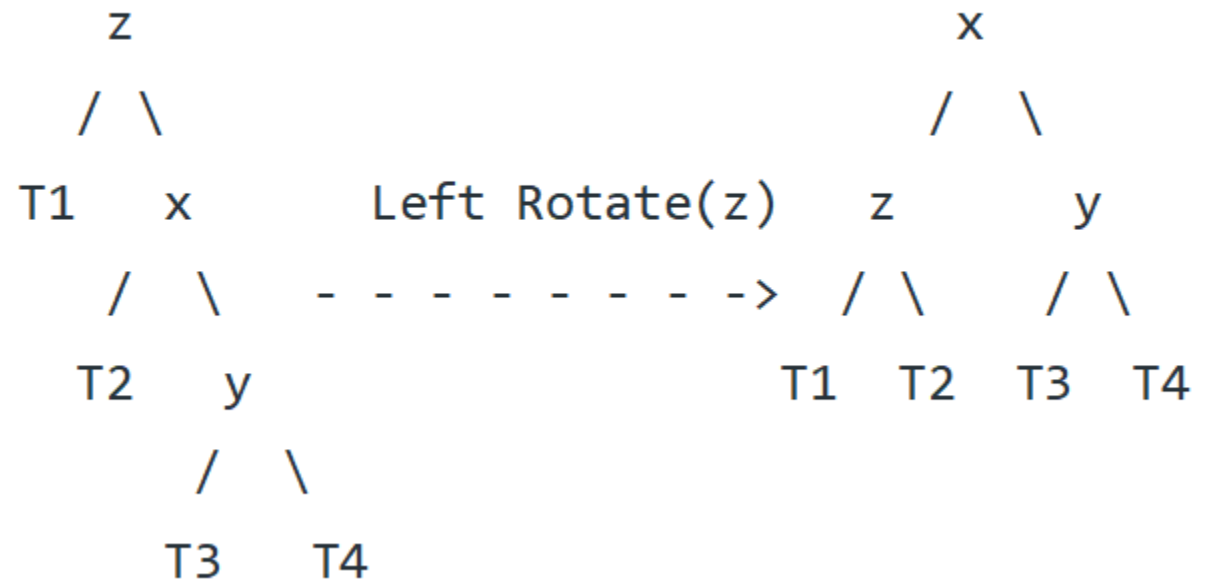
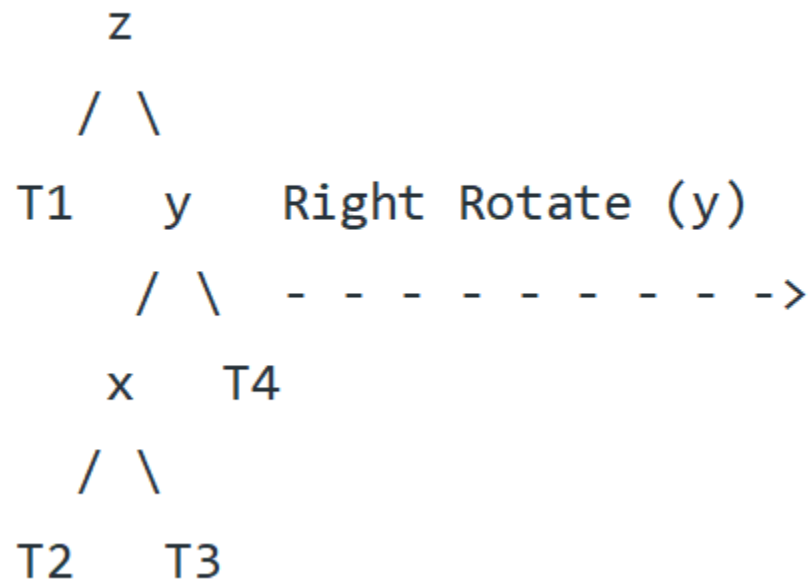
## 4. Right Left Case



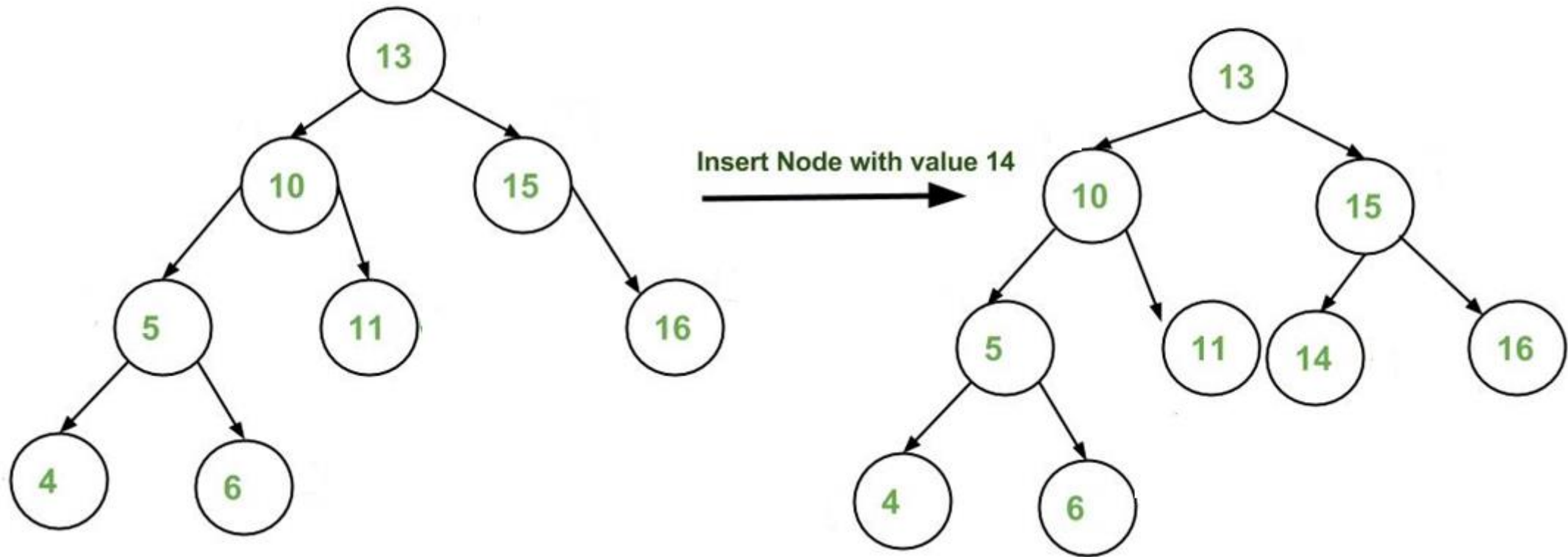
# Right Left Case



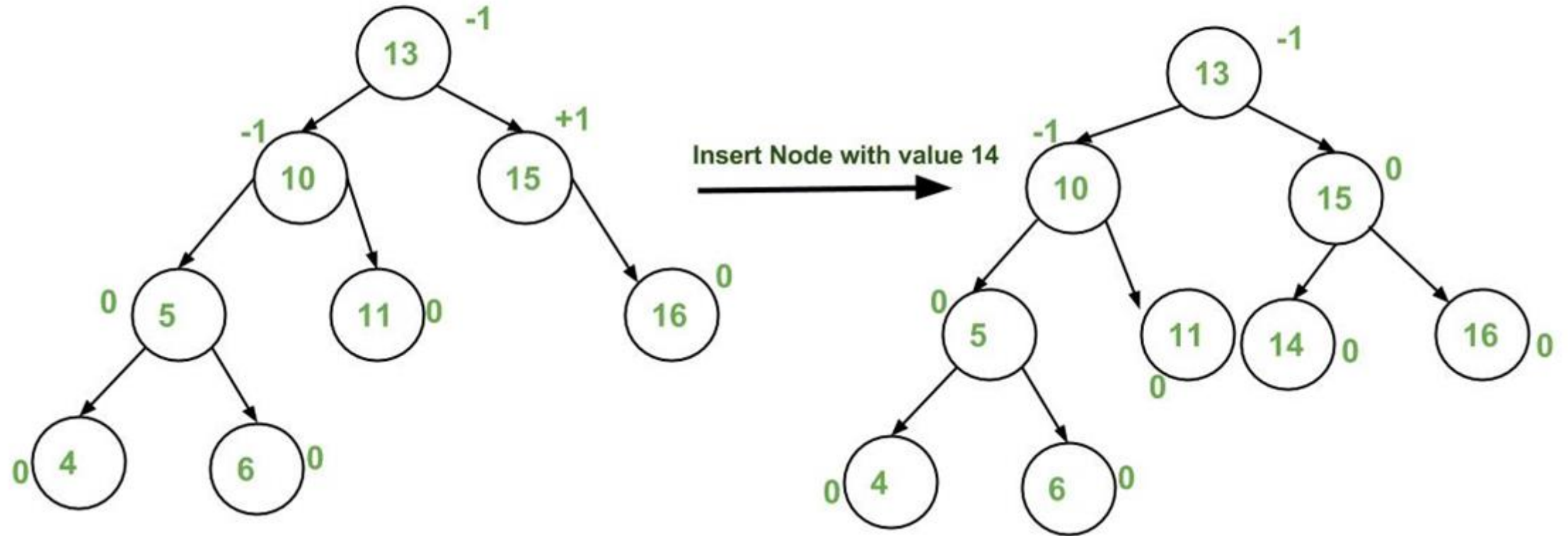
## 4. Right Left Case



# Illustration of Insertion at AVL Tree 1

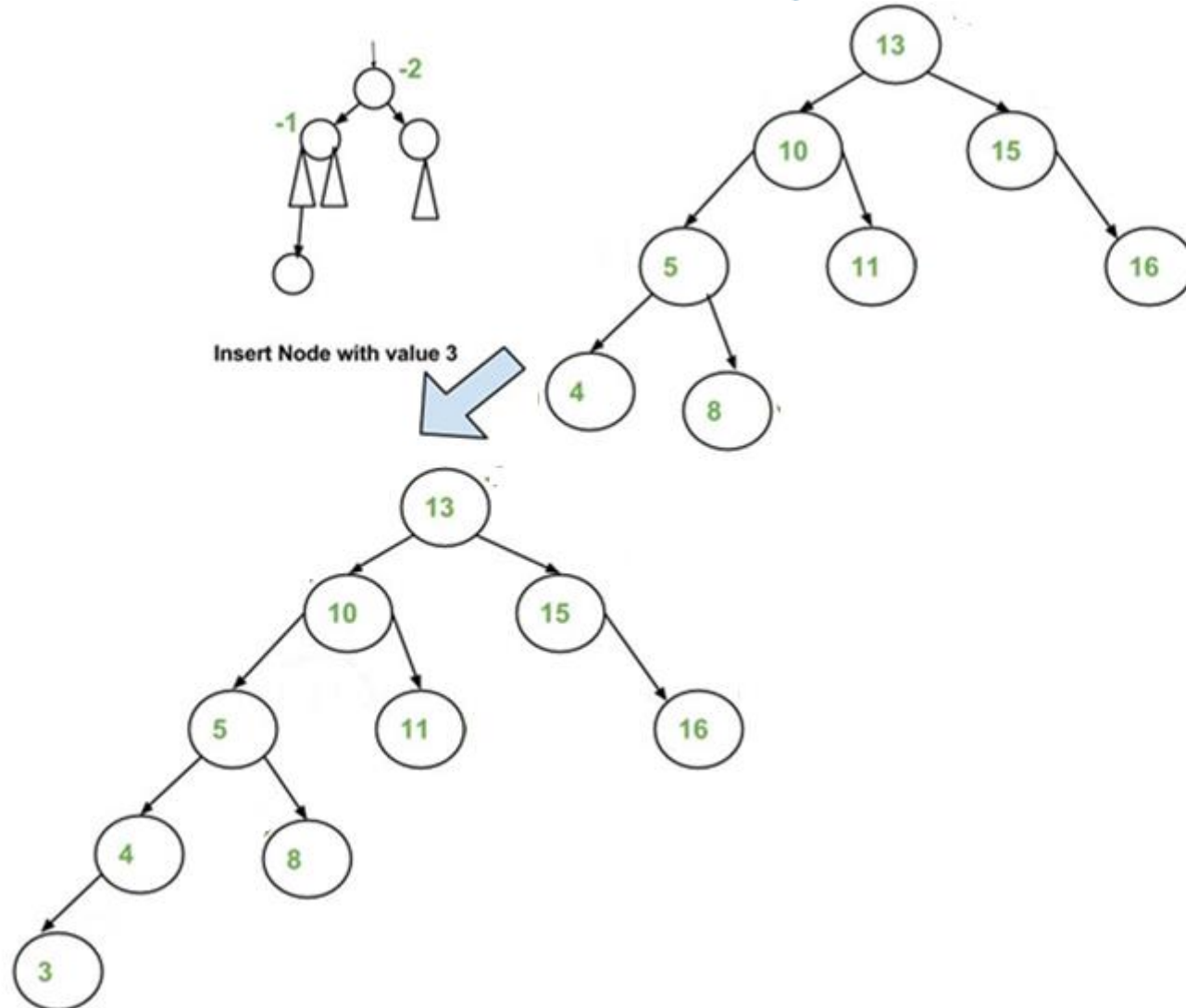


# Illustration of Insertion at AVL Tree 1

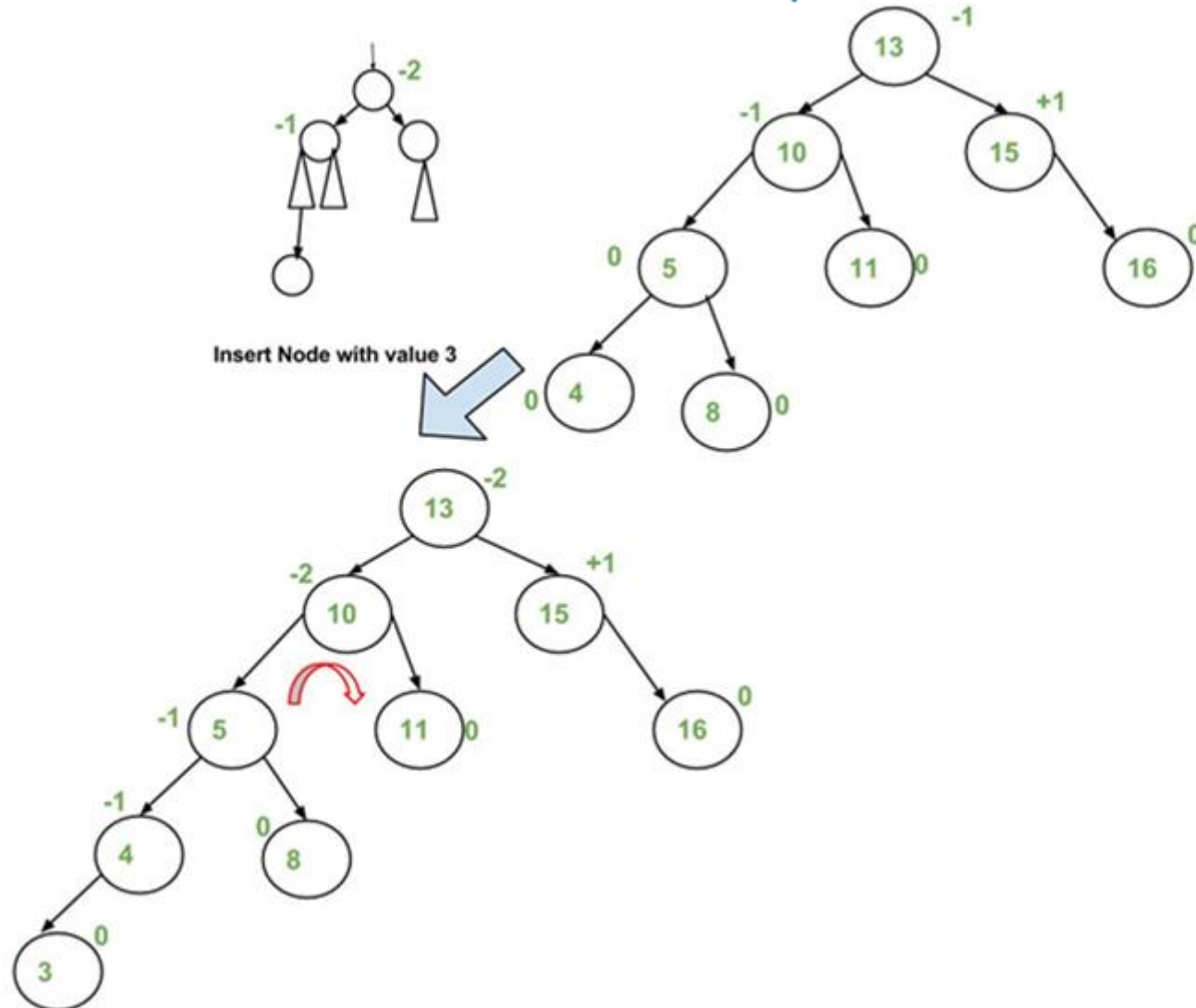




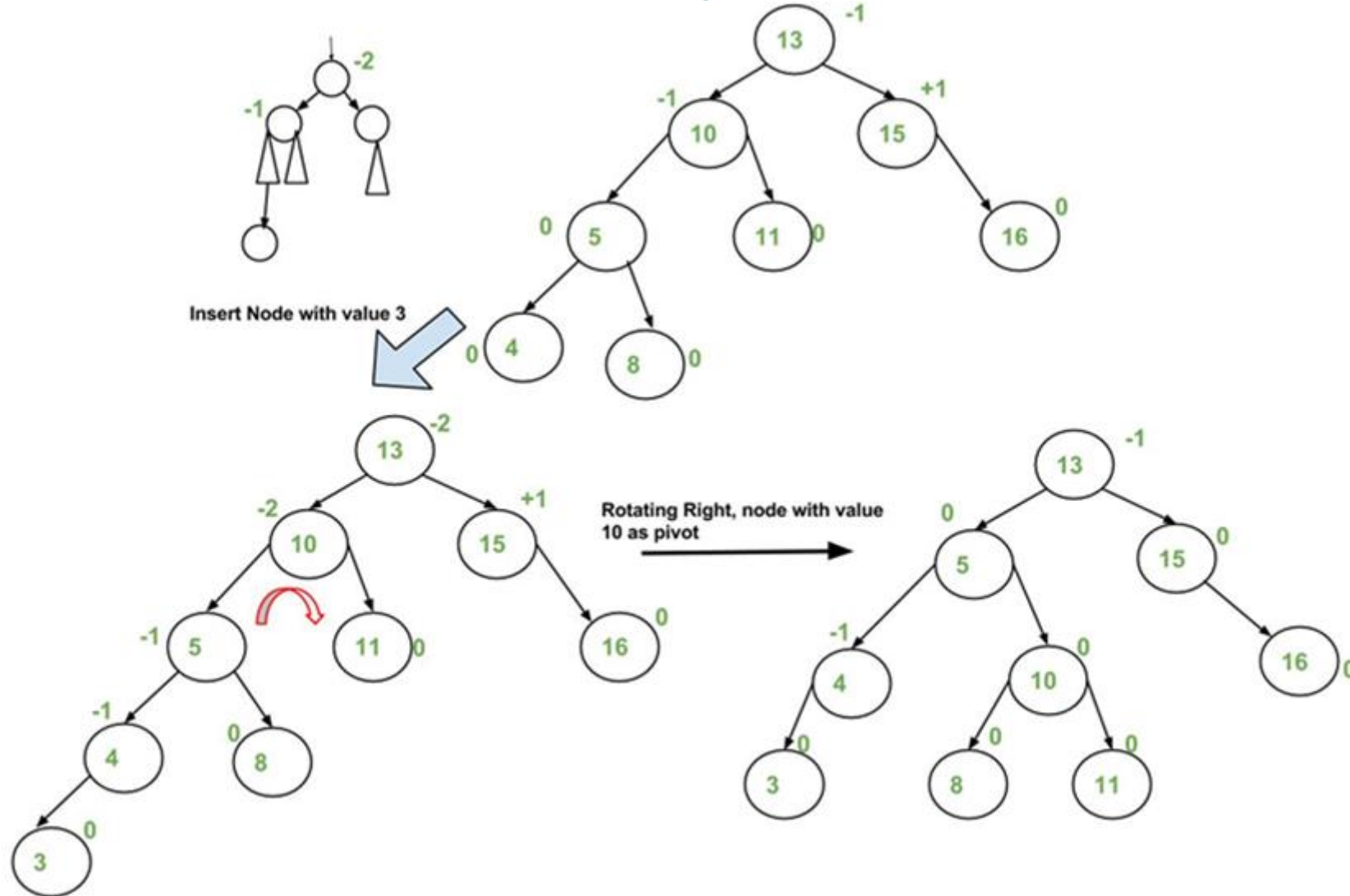
# Illustration of Insertion at AVL Tree 2



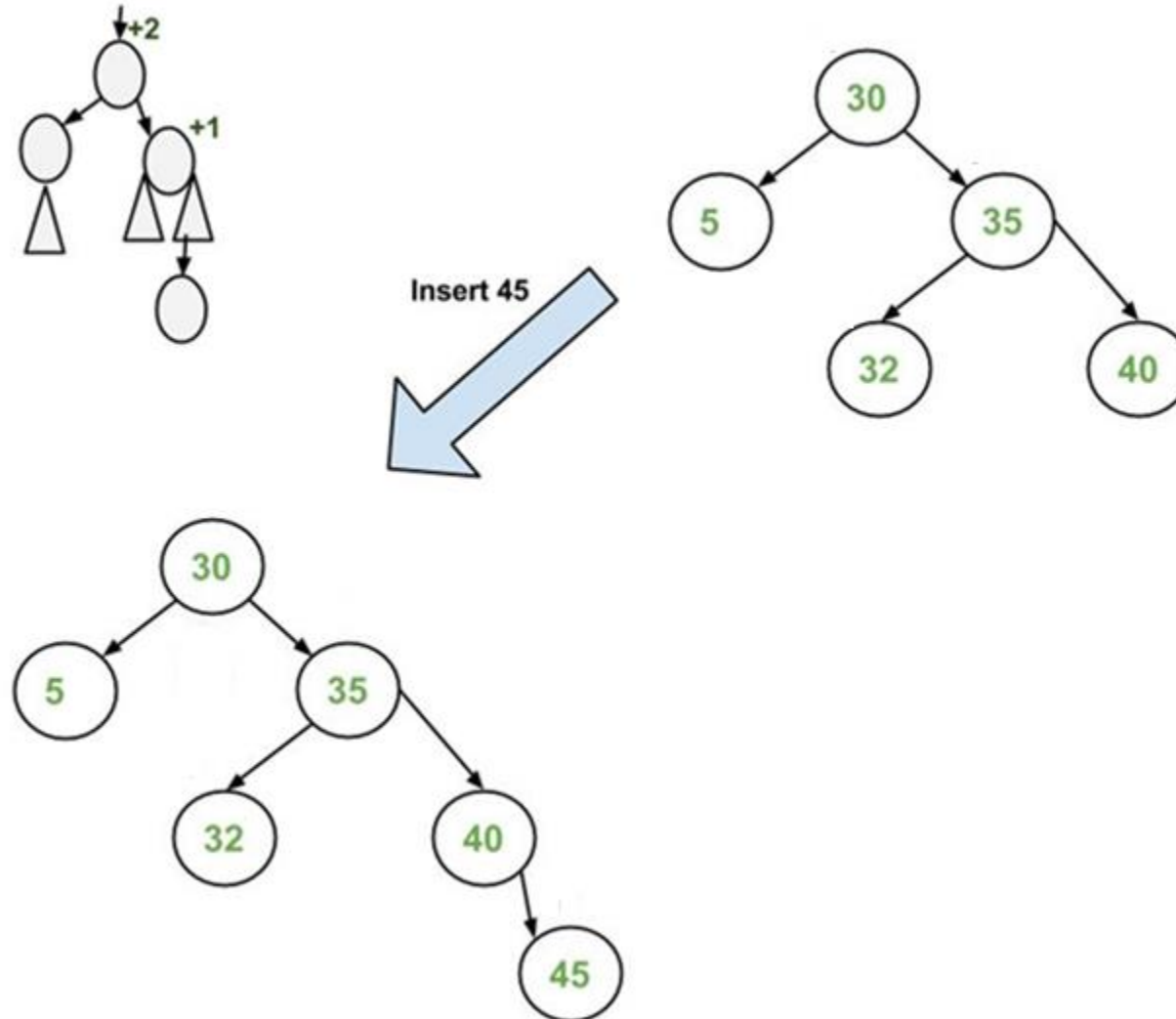
# Illustration of Insertion at AVL Tree 2



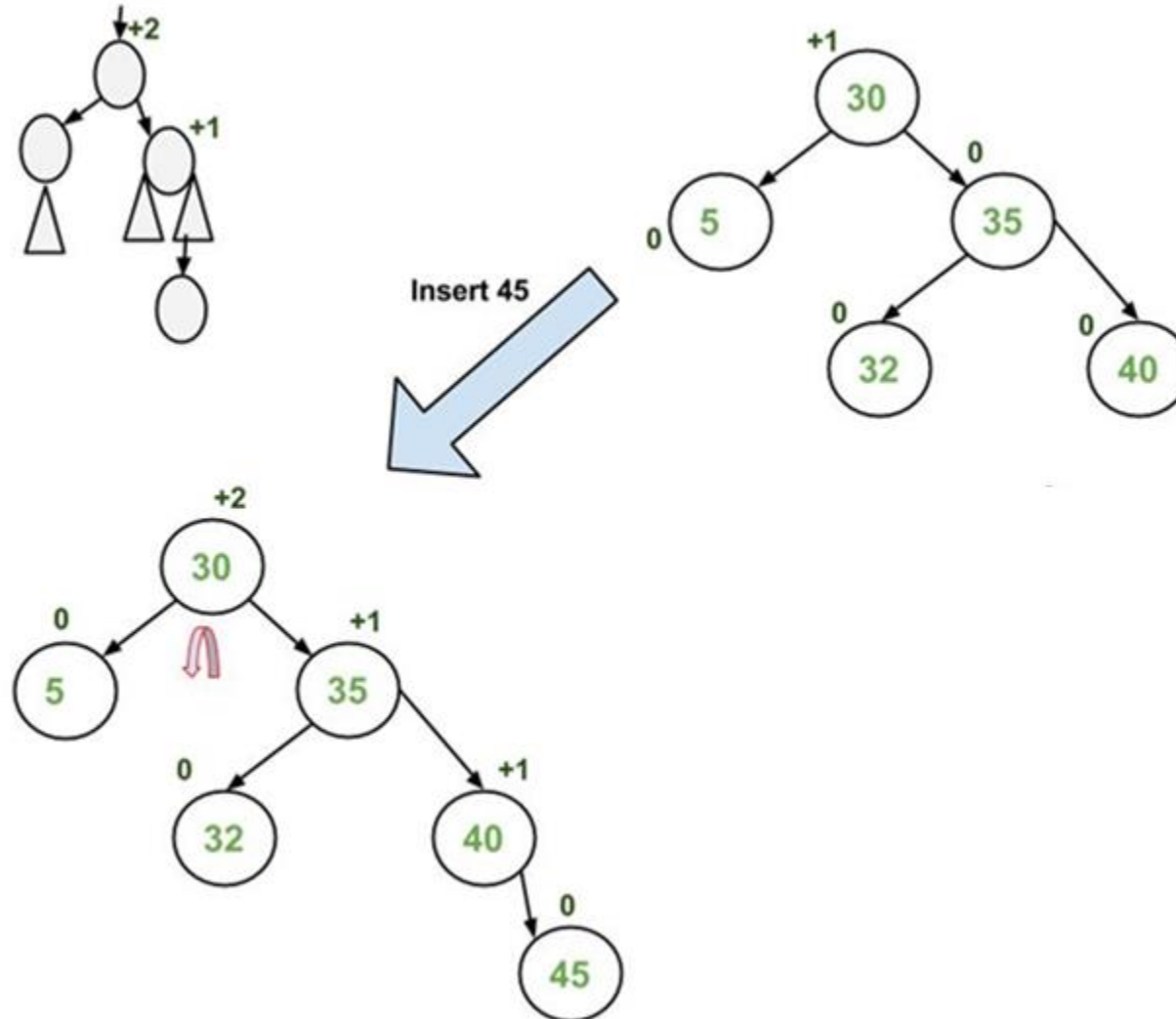
# Illustration of Insertion at AVL Tree 2



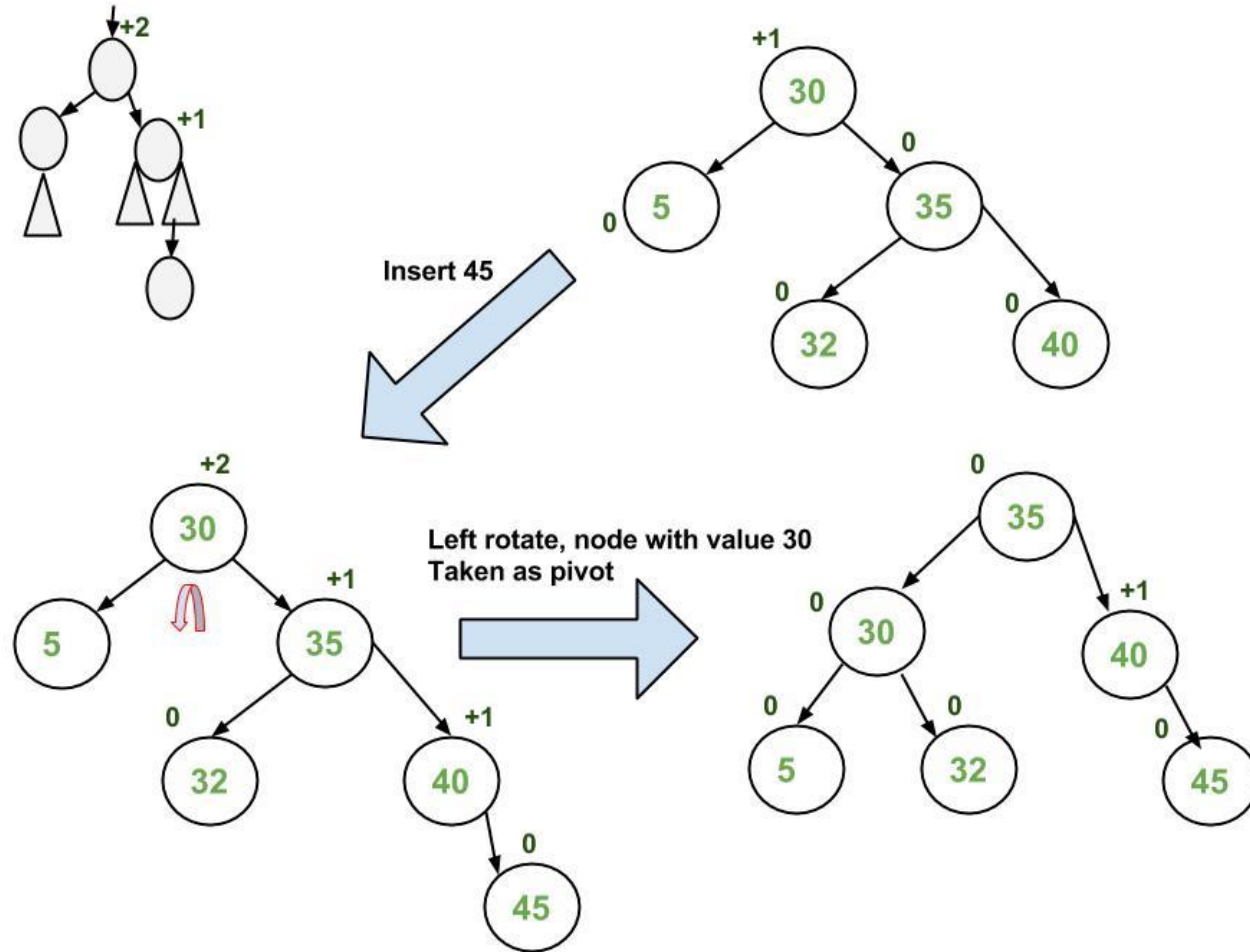
# Illustration of Insertion at AVL Tree 3



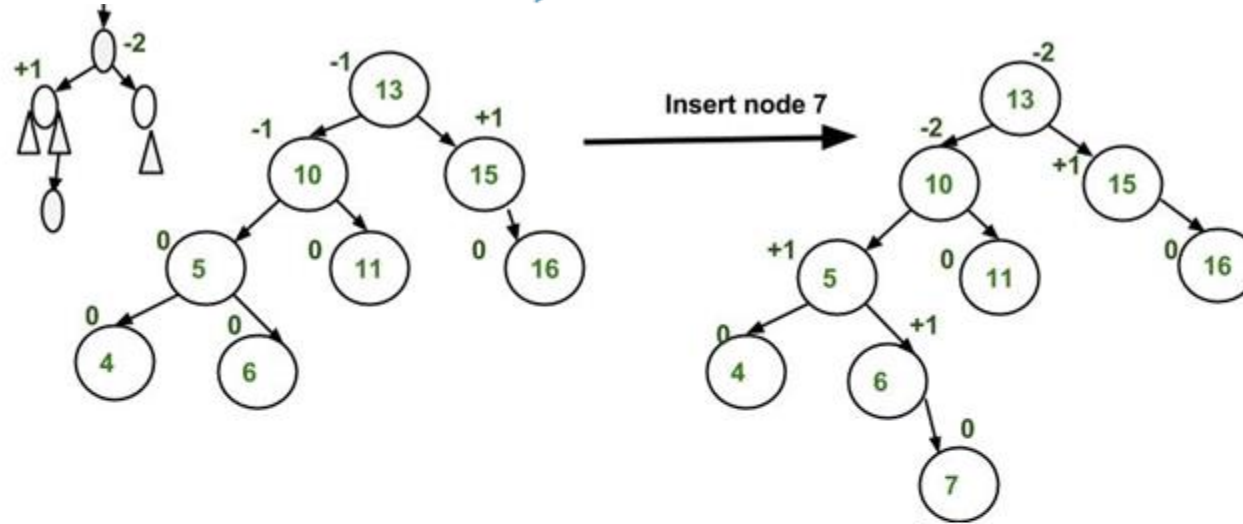
# Illustration of Insertion at AVL Tree 3



# Illustration of Insertion at AVL Tree 3

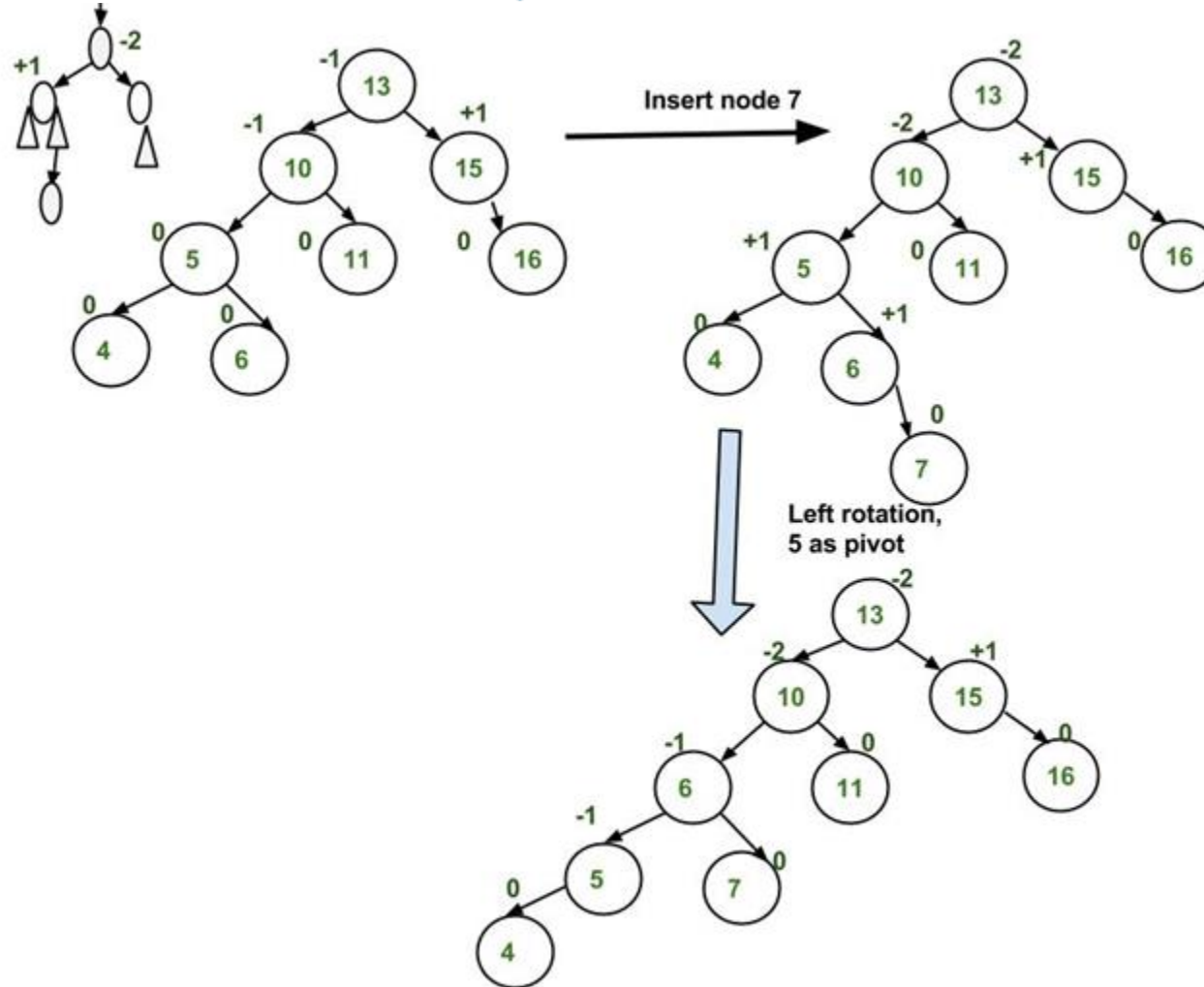


# Illustration of Insertion at AVL Tree 4

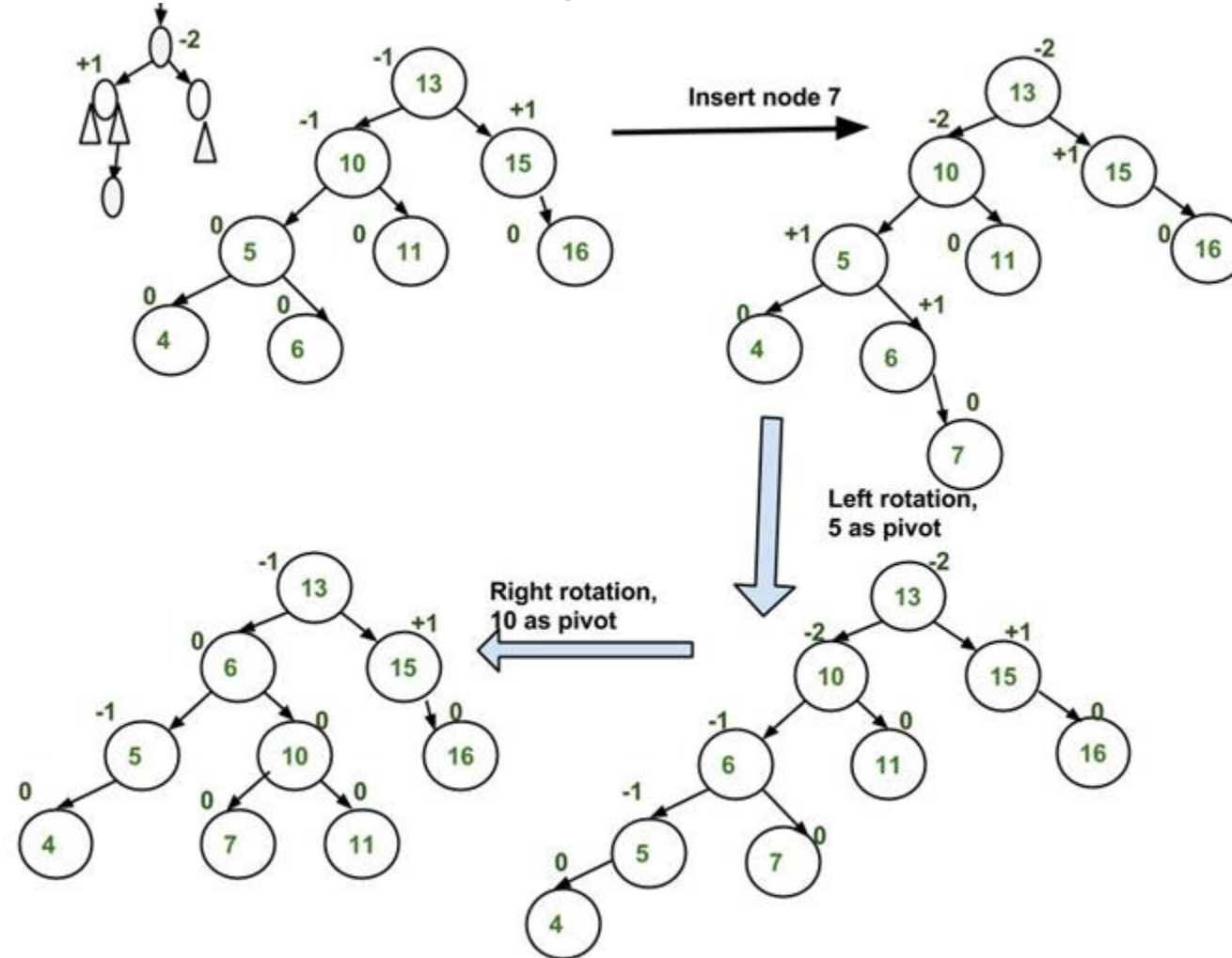




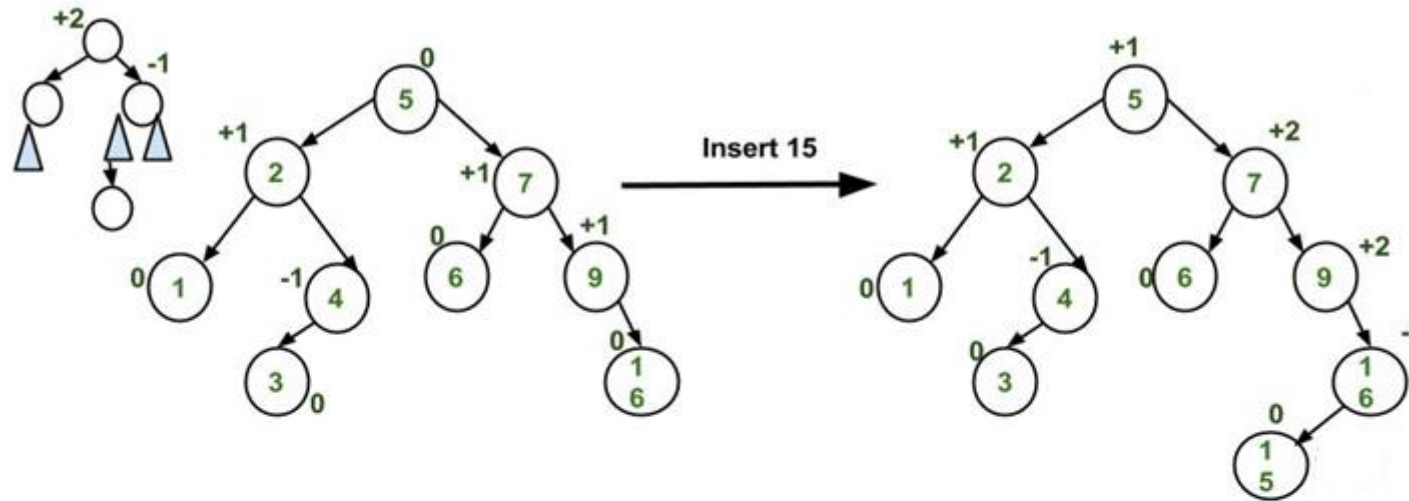
# Illustration of Insertion at AVL Tree 4



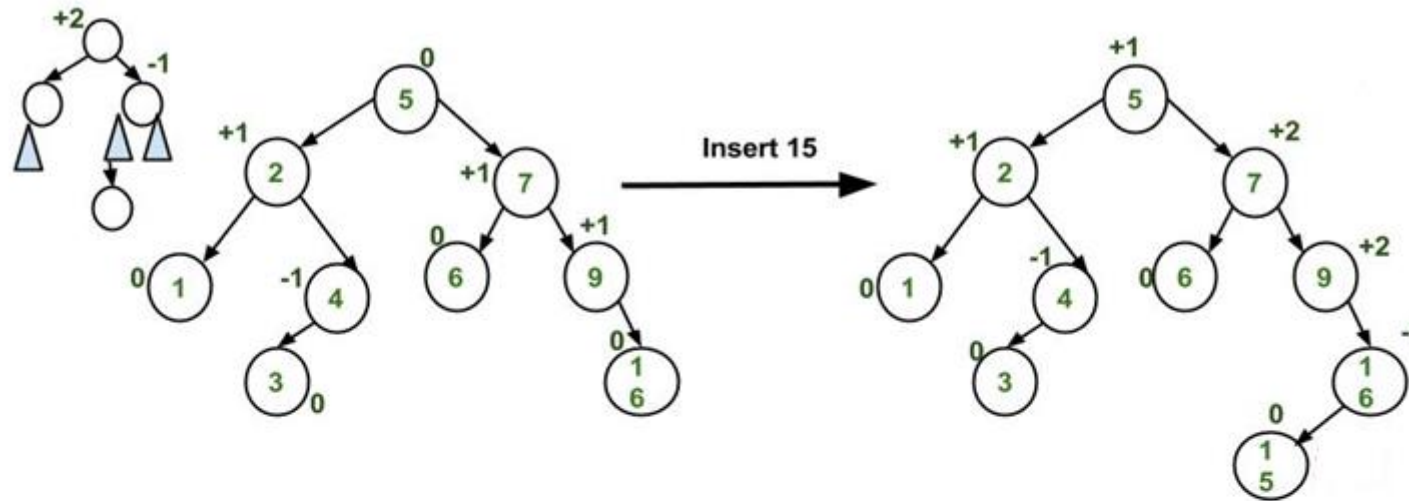
# Illustration of Insertion at AVL Tree 4



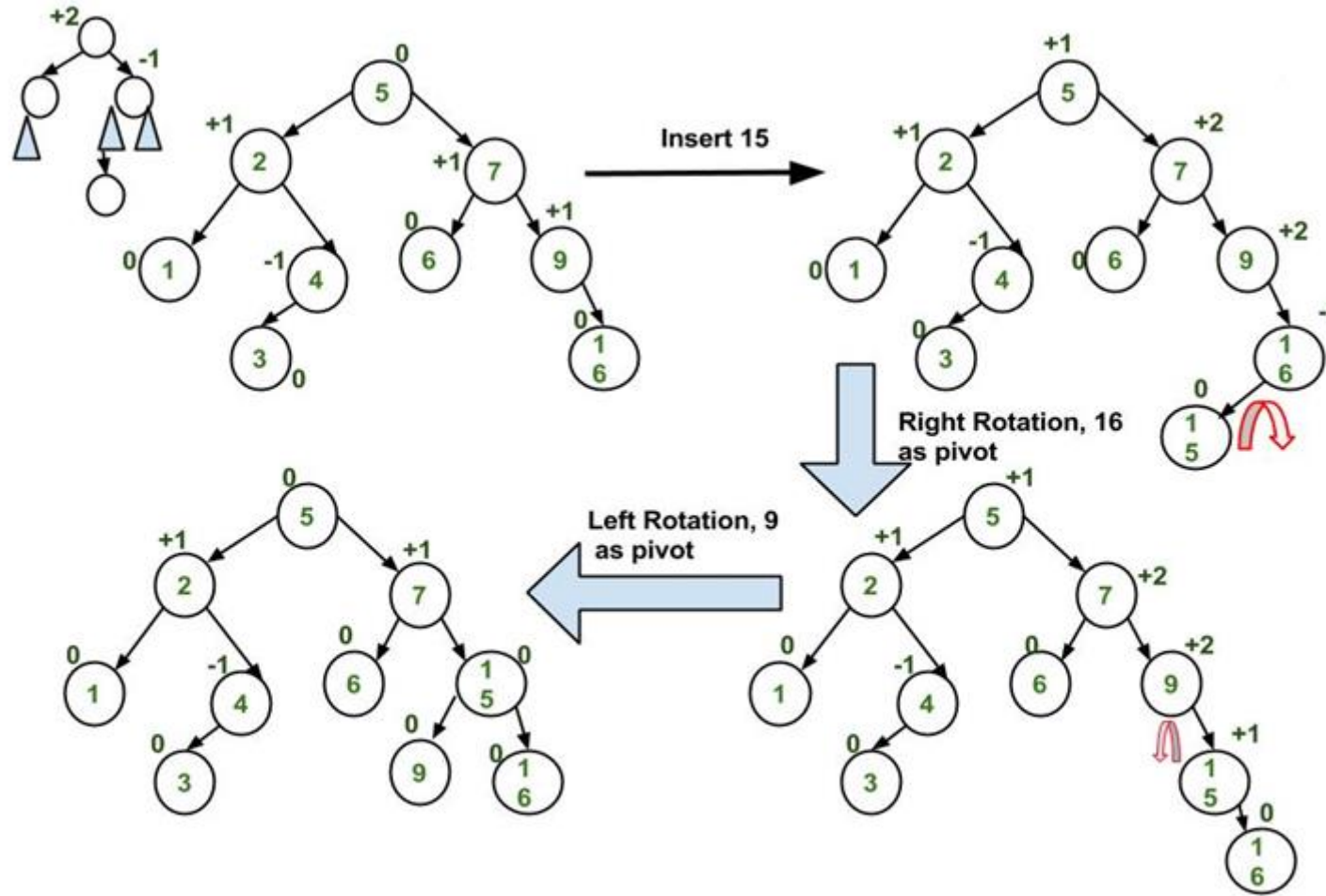
# Illustration of Insertion at AVL Tree 5



# Illustration of Insertion at AVL Tree 5



# Illustration of Insertion at AVL Tree 5



## Illustration of Insertion at AVL Tree 5



```
// C++ program to delete a node from AVL Tree
#include<iostream>
using namespace std;

// An AVL tree node
class Node { public: int key; Node *left; Node *right; int height; };

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(Node *N) { if (N == NULL) return 0; return N->height; }

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key) { Node* node = new Node(); node->key = key;
node->left = NULL; node->right = NULL;
node->height = 1; // new node is initially added at leaf
return(node); }
```

## Illustration of Insertion at AVL Tree 5



جامعة  
المنارة

```
// A utility function to right rotate subtree rooted with y See the diagram given above.
```

```
Node *rightRotate(Node *y) { Node *x = y->left; Node *T2 = x->right;
```

```
x->right = y; y->left = T2; // Perform rotation
```

```
// Update heights
```

```
y->height = max(height(y->left), height(y->right)) + 1;
```

```
x->height = max(height(x->left), height(x->right)) + 1;
```

```
return x; // Return new root
```

```
}
```

```
// A utility function to left rotate subtree rooted with x
```

```
// See the diagram given above.
```

```
Node *leftRotate(Node *x) { Node *y = x->right; Node *T2 = y->left;
```

```
y->left = x; x->right = T2; // Perform rotation
```

```
// Update heights
```

```
x->height = max(height(x->left), height(x->right)) + 1;
```

```
y->height = max(height(y->left), height(y->right)) + 1;
```

```
return y; // Return new root
```

```
}
```



# Illustration of Insertion at AVL Tree 5



```
// Get Balance factor of node N
int getBalance(Node *N) { if (N == NULL) return 0;
    return height(N->left) - height(N->right); }

Node* insert(Node* node, int key) { /* 1. Perform the normal BST rotation */
    if (node == NULL) return(newNode(key));
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
    else return node; // Equal keys not allowed

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left), height(node->right));

    /* 3. Get the balance factor of this ancestor node to check whether
    this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases Left Left Case
    if (balance > 1 && key < node->left->key) return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key) return leftRotate(node);
```

# Illustration of Insertion at AVL Tree 5



```
// Left Right Case
```

```
if (balance > 1 && key > node->left->key){ node->left = leftRotate(node->left);  
return rightRotate(node); }
```

```
// Right Left Case
```

```
if (balance < -1 && key < node->right->key){ node->right = rightRotate(node->right);  
return leftRotate(node); }
```

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

```
/* Given a non-empty binary search tree, return the node with minimum key value  
found in that tree. Note that the entire tree does not need to be searched. */
```

```
Node * minValueNode(Node* node) { Node* current = node;
```

```
/* loop down to find the leftmost leaf */
```

```
while (current->left != NULL) current = current->left; return current;
```

```
}
```

## Illustration of Insertion at AVL Tree 5



```
// Recursive function to delete a node with given key from subtree with  
// given root. It returns root of the modified subtree.
```

```
Node* deleteNode(Node* root, int key) { // STEP 1: PERFORM STANDARD BST DELETE  
    if (root == NULL) return root;
```

```
    // If the key to be del is smaller than the root's key, then it lies in left subtree  
    if ( key < root->key ) root->left = deleteNode(root->left, key);
```

```
    // If the key to be del is greater than the root's key, then it lies in right subtree  
    else if( key > root->key ) root->right = deleteNode(root->right, key);
```

```
    // if key is same as root's key, then This is the node to be deleted  
    else { // node with only one child or no child
```

```
        if( (root->left == NULL) || (root->right == NULL) )  
        { Node *temp = root->left ? root->left : root->right;
```

```
            if (temp == NULL) { temp = root; root = NULL; } // No child case
```

```
                else // One child case
```

```
                    *root = *temp; // Copy the contents of the non-empty child
```

```
                    free(temp); }
```

## Illustration of Insertion at AVL Tree 5



```
else { // node with two children: Get the inorder
Node* temp = minValueNode(root->right); // successor (smallest in the right subtree)
root->key = temp->key; // Copy the inorder successor's data to this node
root->right = deleteNode(root->right, temp->key); // Delete the inorder successor
}
}

if (root == NULL) return root; // If the tree had only one node then return

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = 1 + max(height(root->left), height(root->right));

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether this
// node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases // Left Left Case
if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root);
```

## Illustration of Insertion at AVL Tree 5



```
// Left Right Case
```

```
if (balance > 1 && getBalance(root->left) < 0) {root->left = leftRotate(root->left);  
    return rightRotate(root);    }
```

```
// Right Right Case
```

```
if (balance < -1 && getBalance(root->right) <= 0) return leftRotate(root);
```

```
// Right Left Case
```

```
if (balance < -1 && getBalance(root->right)>0){root->right=rightRotate(root->right);  
    return leftRotate(root);
```

```
}
```

```
return root;
```

```
}
```

```
// A utility function to print preorder traversal of the tree.
```

```
// The function also prints height of every node
```

```
void preOrder(Node *root) {    if(root != NULL)  
    {cout << root->key << " "; preOrder(root->left);preOrder(root->right);    }  
}
```

# Illustration of Insertion at AVL Tree 5



```
// Driver Code
```

```
int main()
```

```
{ Node *root = NULL;
```

```
/* Constructing tree given in the above figure */
```

```
root = insert(root, 9);
```

```
root = insert(root, 5);
```

```
root = insert(root, 10);
```

```
root = insert(root, 0);
```

```
root = insert(root, 6);
```

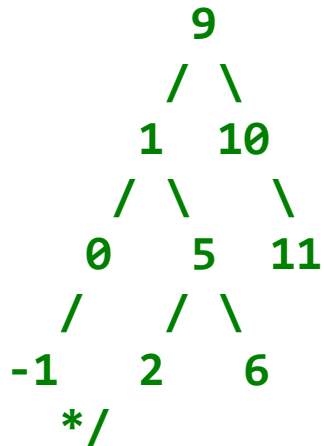
```
root = insert(root, 11);
```

```
root = insert(root, -1);
```

```
root = insert(root, 1);
```

```
root = insert(root, 2);
```

```
/* The constructed AVL Tree would be
```

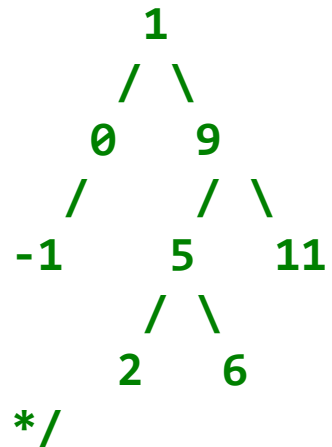


# Illustration of Insertion at AVL Tree 5



```
cout << "Preorder traversal of the constructed AVL tree is \n";  
preOrder(root);
```

```
root = deleteNode(root, 10);  
/* The AVL Tree after deletion of 10
```



```
cout << "\nPreorder traversal after deletion of 10 \n";  
preOrder(root);  
system("pause"); return 0;  
}
```



انتهت محاضرة الأسبوع 10