DATA STRUCTURES AND ALGORITHMS

Binary Trees

LearnOA®

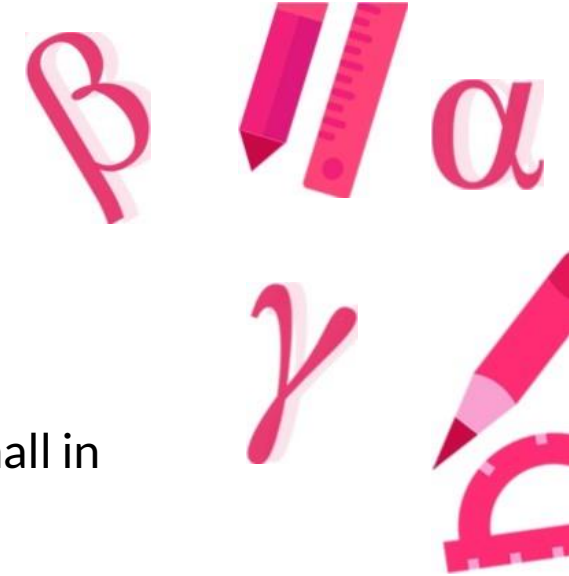
Objectives

At the end of this lesson, you will be able to:

- Define a binary tree
- Describe the terminologies associated with a binary tree
- Use a dynamically allocated data structure to represent a binary tree
- Traverse a binary tree
- Add nodes to a binary tree
- Remove nodes from a binary tree
- Search a binary tree

Eliminative or a Binary Search

- The mode of accessing data in a linked list is linear.
- Therefore, in the worst case scenario of the data in question being stored at the extremes of the list, it would involve starting with the first node, and traversing through all the nodes till one reaches the last node of the list to access the data.
- Therefore, search through a linked list is always linear.



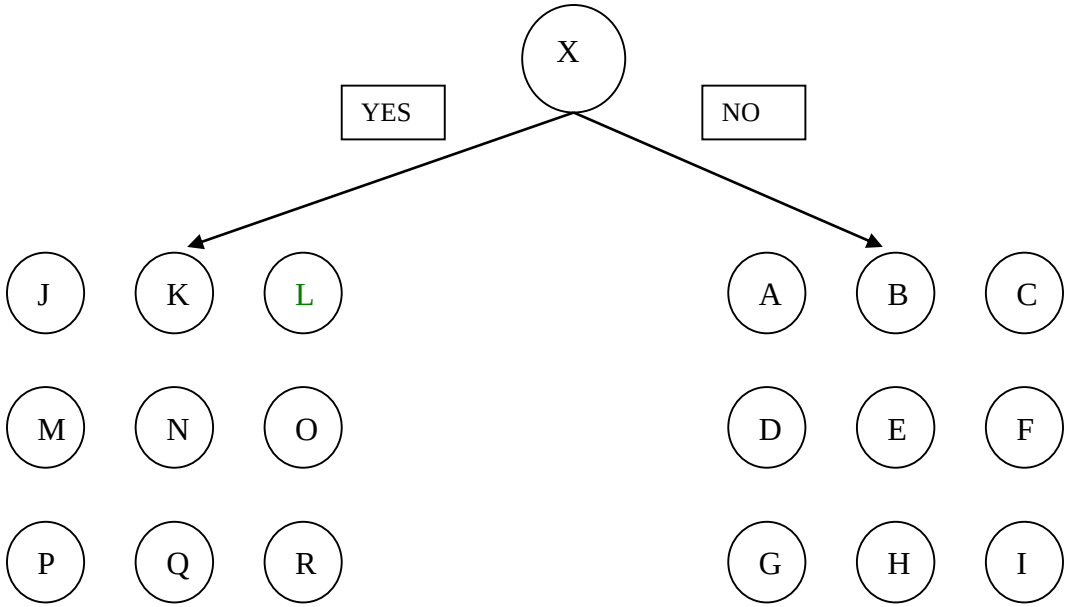
Eliminative or a Binary Search

- A linear search is fine if the nodes to be searched in a linked list are small in number.
- But the linear search becomes ineffective as the number of nodes in a linked list increase.
- The search time increases in direct proportion with the size of the linked list.
- It becomes imperative to have better searching mechanisms than a linear search.

Eliminative or a Binary Search

- You will now be exposed to a game that will highlight a new mechanism of searching.

Coin Search in a Group



Eliminative or a Binary Search

- A coin is with one of the members in the audience divided into the two sections on the left and the right respectively as shown in the diagram.
- The challenge facing X, the protagonist, is to find the person with the coin in the least number of searches.

Employing the Linear Search

- X, familiar with a linear search, starts using it to search for the coin among the group.
- Let us assume the worst-case scenario of the coin being with R.
- If X was to start the search with A and progress linearly through B, C, ….M and finally to R, he would have taken a minimum of 18 searches (R being the 18th person searched in sequence to find the coin.

Employing the Linear Search

- As you can see, this kind of search is not very efficient especially when the number of elements to be searched is high.
- An eliminative search, also called a binary search, provides a far better searching mechanism.

Employing the Eliminative or The Binary Search

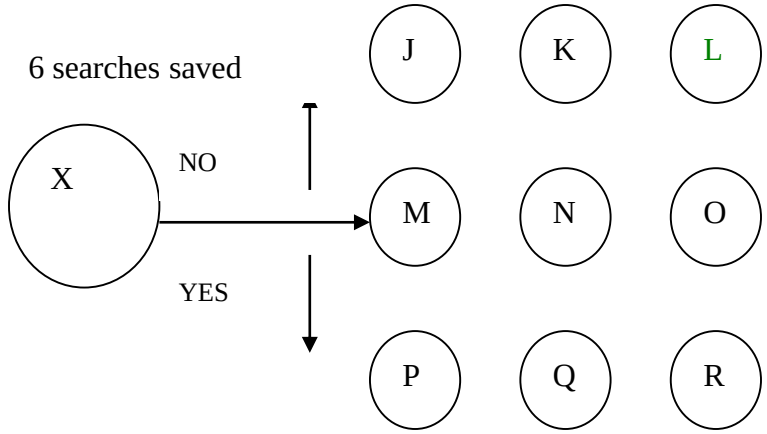
- Assume that X can pose intelligent questions to the audience to cut down on the number of searches.
- A valid question that he could pose is “Which side of the audience has the coin?”
- ‘A’ in the audience to the left of him says that his side of the audience does not have the coin.
- So X can completely do away with searching the audience to his left. That saves him 9 searches.

Employing the Eliminative or The Binary Search

- X has now got to search the audience to the right of him for searching out the coin.
- Here too, he can split the audience into half by standing adjacent to the middle row, and posing the same question that he asked earlier “Which side of the audience has the coin?”
- ‘M’ in the middle row replies that the coin is with the audience to the left of him.

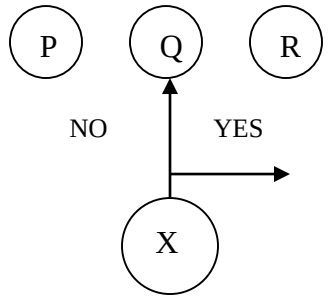
Employing the Eliminative or The Binary Search

- X has in the process now eliminated 6 more searches, that is, the middle row and the row to the left of the middle as shown in the diagram below.



Employing the Eliminative or The Binary Search

- X is now left with row to the right of the middle row containing P, Q, and R that has to be searched.
- Here too, he can position himself right at the middle of the row adjacent to Q and pose the same question,
- “Which side of the audience has the coin?” ‘Q’ replies that the coin is to his left.



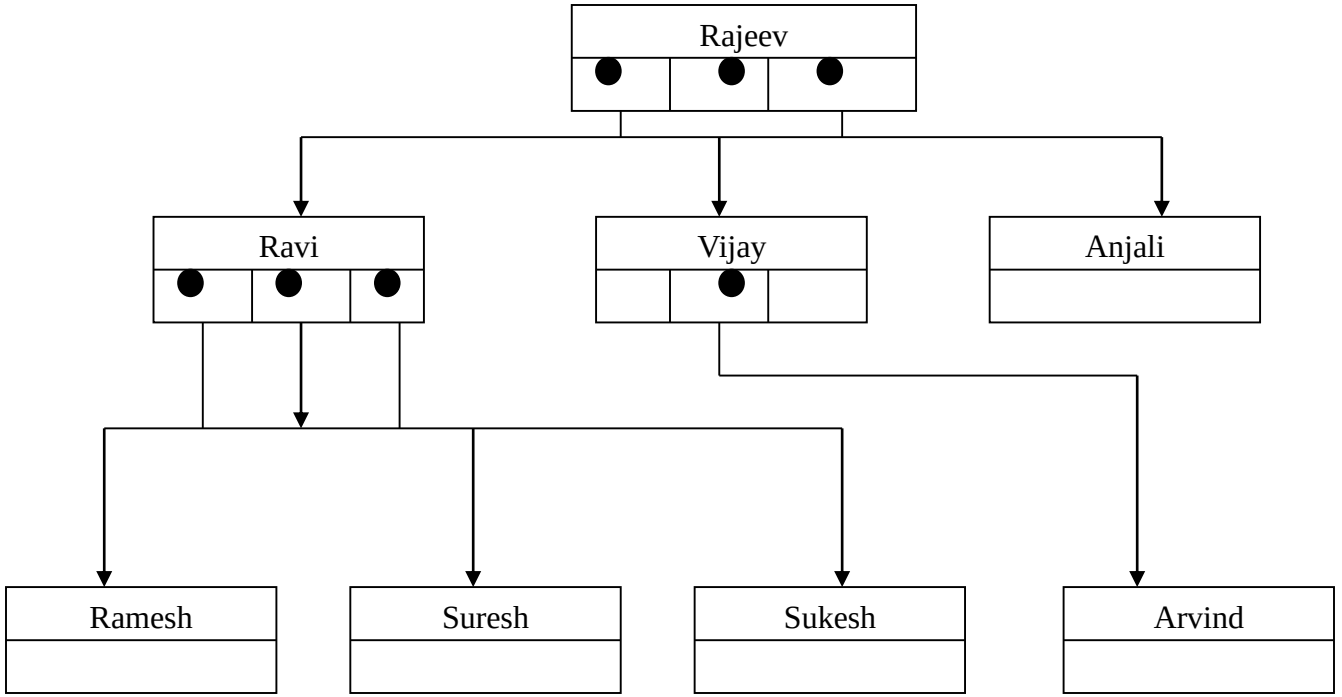
Eliminative or Binary Search

- That completes our eliminative or binary search.
- It is called a binary search because at each stage of the search, the search is cut by more than half.
- This kind of search forms the basis for the searching mechanism employed in a data structure wherein the data is represented in a hierarchal manner unlike the linear mechanism of storage employed in a linked list.

Trees

- Compared to linked lists that are linear data structures, trees are non-linear data structures.
- In a linked list, each node has a link which points to another node.
- In a tree structure, however, each node may point to several nodes, which may in turn point to several other nodes.
- Thus, a tree is a very flexible and a powerful data structure that can be used for a wide variety of applications.

Trees



Trees

- A tree consists of a collection of nodes that are connected to each other.
- A tree contains a unique first element known as the root, which is shown at the top of the tree structure.
- A node which points to other nodes is said to be the parent of the nodes to which it is pointing, and the nodes that the parent node points to are called the children, or child nodes of the parent node.

Trees

- The root is the only node in the tree that does not have a parent.
- All other nodes in the tree have exactly one parent.
- There are nodes in the tree that do not have any children. Such nodes are called leaf nodes.
- Nodes are siblings if they have the same parent.

Trees

- A node is an ancestor of another node if it is the parent of that node, or the parent of some other ancestor of that node.
- The root is an ancestor of every other node in the tree.
- Similarly, we can define a node to be a descendant of another node if it is the child of the node, or the child of some other descendant of that node.
- You may note that all the nodes in the tree are descendants of the root node.

Tree

- An important feature of a tree is that there is a single unique path from the root to any particular node.
- The length of the longest path from the root to any node is known as the depth of the tree.
- The root is at level 0 and the level of any node in the tree is one more than the level of its parent.
- In a tree, any node can be considered to be a root of the tree formed by considering only the descendants of that node. Such a tree is called the subtree that itself is a tree.

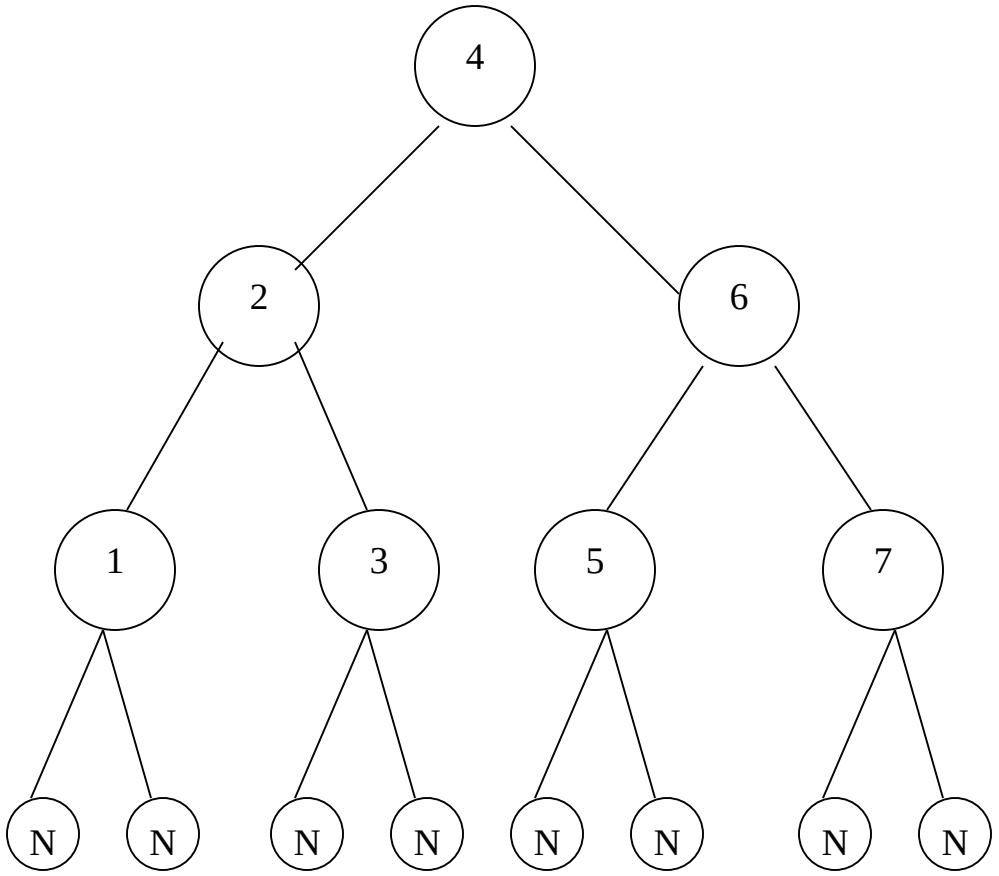
Binary Tree

- If you can introduce a restriction that each node can have a maximum of two children or two child nodes, then you can have a binary tree.
- You can give a formal definition of a binary tree as a tree which is either empty or consists of a root node together with two nodes, each of which in turn forms a subtree.
- You therefore have a left subtree and a right subtree under the root node.

Binary Search Tree

- A complete binary tree can be defined as one whose non-leaf nodes have non-empty left and right subtrees and all leaves are at the same level.
- This is also called as a balanced binary tree.
- If a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree are greater than the contents of n, such a tree is called a binary search tree.
- The following is an example of a balanced binary search tree.

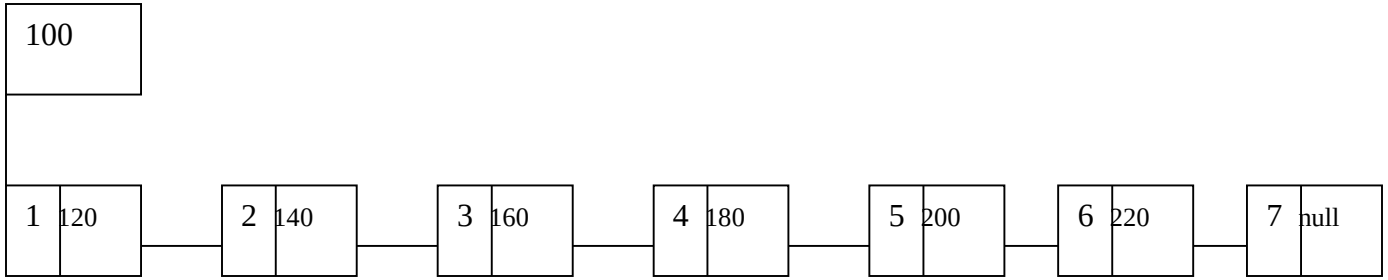
Balanced Binary Search Tree



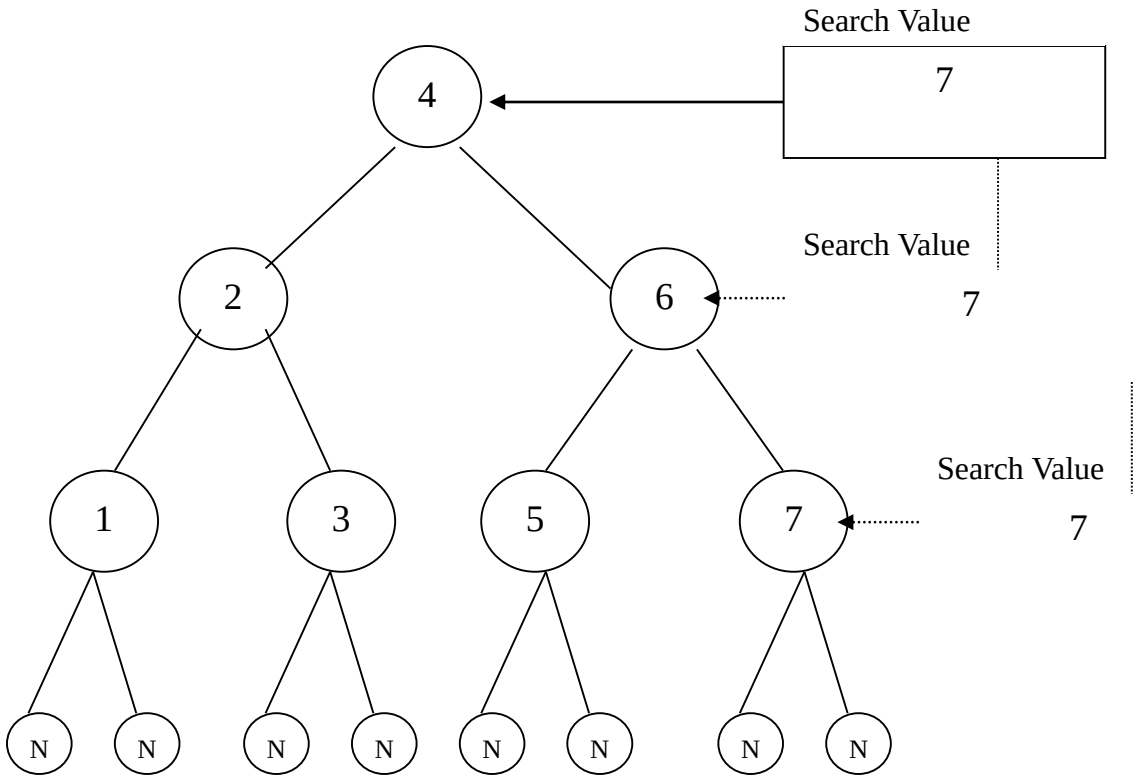
Binary Vs. Linear Search

- As the name suggests, balanced binary search trees are very useful for searching an element just as with a binary search.
- If we use linked lists for searching, we have to move through the list linearly, one node at a time.
- If we search an element in a binary search tree, we move to the left subtree for smaller values, and to the right subtree for larger values, every time reducing the search list by half approximately.

Linear Search in a Linked List



Binary Search in a Binary Search Tree



The Essence of a Binary Search

- To summarize, you have completely done away with searching with the entire left subtree of the root node and its descendant subtrees, in the process doing away with searching one-half of the binary search tree.
- Even while searching the right subtree of the root node and its descendant subtrees, we keep searching only one-half of the right subtree and its descendants.
- This is more because of the search value in particular, which is 7. The left subtree of the right subtree of the root could have been searched in case the value being searched for was say 5.

The Essence of a Binary Search

- Thus we can conclude that while searching for a value in a balanced binary search tree, the number of searches is cut by more than half (3 searches in a balanced binary search tree) compared to searching in a linked list (7 searches).

- **Thus a search that is hierarchical, eliminative and binary in nature is far efficient when compared to a linear search.**

Data Structure Representation of a Binary Trees

- A tree node may be implemented through a structure declaration whose elements consist of a variable for holding the information and also consist of two pointers, one pointing to the left subtree and the other pointing to the right subtree.
- A binary tree can also be looked at as a special case of a doubly linked list that is traversed hierarchically.
- The following is the structure declaration for a tree node:

Data Structure Representation of a Binary Trees

```
struct btreenode
{
  int info;
  struct btreenode *left;
  struct btreenode *right;
};
```

Traversing a Binary Tree

- Traversing a binary tree entails visiting each node in the tree exactly once.
- Binary tree traversal is useful in many applications, especially those involving an indexed search.
- Nodes of a binary search tree are traversed hierarchically.
- The methods of traversing a binary search tree differ primarily in the order in which they visit the nodes.

Traversing a Binary Tree

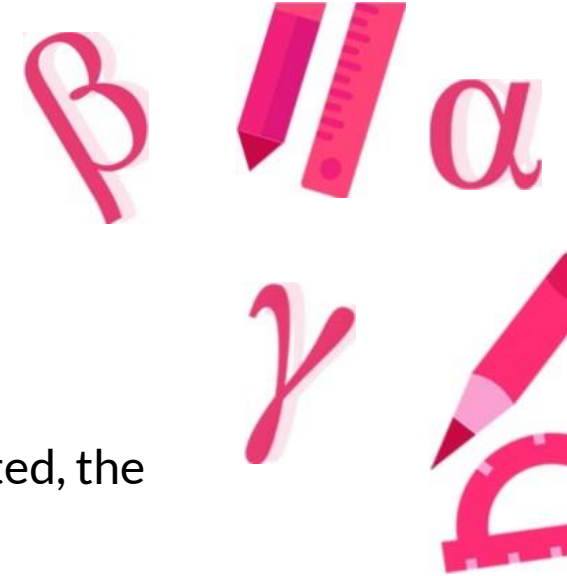
- At a given node, there are three things to do in some order. They are:
  - ✓ To visit the node itself
  - ✓ To traverse its left subtree
  - ✓ To traverse its right subtree
- We can traverse the node before traversing either subtree.
- Or, we can traverse the node between the subtrees.
- Or, we can traverse the node after traversing both subtrees.

Traversing a Binary Tree

- If we designate the task of visiting the root as R', traversing the left subtree as L and traversing the right subtree as R, then the three modes of tree traversal discussed earlier would be represented as:
  - ✓ R'LR – Preorder
  - ✓ LRR' – Postorder
  - ✓ LR'R – Inorder

Traversing a Binary Tree

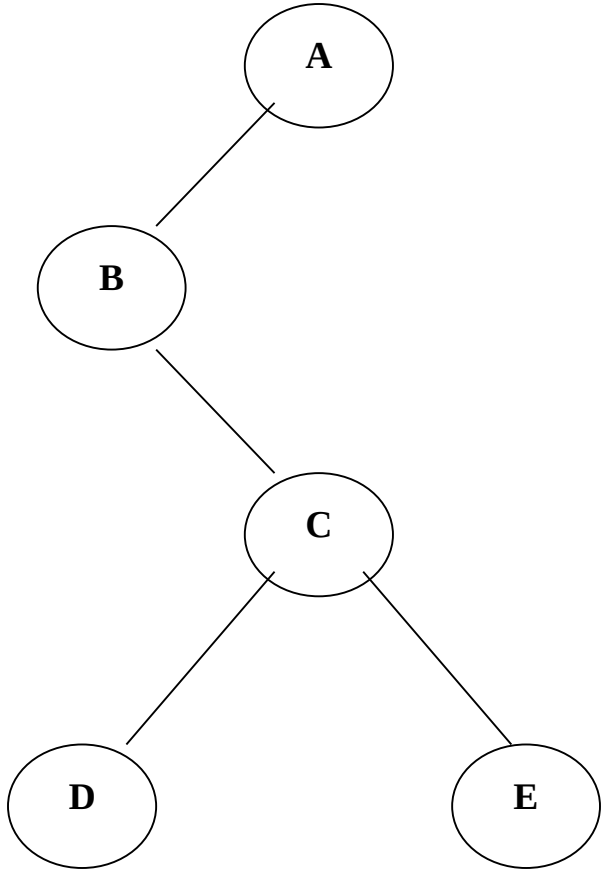
- The functions used to traverse a binary tree using these methods can be kept quite short if we understand the recursive nature of the binary tree.
- Recall that a binary tree is recursive in that each subtree is really a binary tree itself.
- Thus traversing a binary tree involves visiting the root node, and traversing its left and right subtrees.
- The only difference among the methods is the order in which these three operations are performed.



Traversing a Binary Tree

- Depending on the position at which the given node or the root is visited, the name is given.
- If the root is visited before traversing the subtree, it is called the preorder traversal.
- If the root is visited after traversing the subtrees, it is called postorder traversal.
- If the root is visited in between the subtrees, it is called the inorder traversal.

Traversing a Binary Tree



Preorder Traversal

- When we traverse the tree in preorder, the root node is visited first. So, the node containing A is traversed first.
- Next, we traverse the left subtree. This subtree must again be traversed using the preorder method.
- Therefore, we visit the root of the subtree containing B and then traverse its left subtree.
- The left subtree of B is empty, so its traversal does nothing. Next we traverse the right subtree that has root labeled C.

Preorder Traversal

- Then, we traverse the left and right subtrees of C getting D and E as a result.
- Now, we have traversed the left subtree of the root containing A completely, so we move to traverse the right subtree of A.
- The right subtree of A is empty, so its traversal does nothing. Thus the preorder traversal of the binary tree results in the values **ABCDE.**

Inorder Traversal

- For inorder traversal, we begin with the left subtree rooted at B of the root.
- Before we visit the root of the left subtree, we must visit its left subtree, which is empty.
- Hence the root of the left subtree rooted at B is visited first. Next, the right subtree of this node is traversed inorder.

Inorder Traversal

- Again, first its left subtree containing only one node D is visited, then its root C is visited, and finally the right subtree of C that contains only one node E is visited.
- After completing the left subtree of root A, we must visit the root A, and then traverse its right subtree, which is empty.
- Thus, the complete inorder traversal of the binary tree results in values **BDCEA.**

Postorder Traversal

- For postorder traversal, we must traverse both the left and the right subtrees of each node before visiting the node itself.
- Hence, we traverse the left subtree in postorder yielding values D, E, C and B.
- Then we traverse the empty right subtree of root A, and finally we visit the root which is always the last node to be visited in a postorder traversal.
- Thus, the complete postorder traversal of the tree results in **DECBA.**

Code - Preorder Traversal

```
void preorder (p)
struct btreenode *p;
{
 /* Checking for an empty tree */
 if ( p != null)
  {
   /* print the value of the root node */
   printf("%d", p->info);
   /* traverse its left subtree */
   preorder(p->left);
   /* traverse its right subtree */
   preorder(p->right);
  }
}
```

Code – Inorder Traversal

```
void inorder(p)
struct btreenode *p;
{
 /* checking for an empty tree */
 if (p != null)
  {
   /* traverse the left subtree inorder */
   inorder(p->left);
   /* print the value of the root node */
   printf("%d", p->info);
   /*traverse the right subtree inorder */
   inorder(p->right);
  }
}
```

Code – Postorder Traversal

```
void postorder(p)
struct btreenode *p;
 {
  /* checking for an empty tree */
  if (p != null)
   {
    /* traverse the left subtree */
    postorder(p->left);
    /* traverse the right subtree */
    postorder(p->right);
    /* print the value of the root node */
    printf("%d", p->info);
   }
 }
```

Accessing Values From a Binary Search Tree Using Inorder Traversal

- You may note that when you traverse a binary search tree inorder, the keys will be in sorted order because all the keys in the left subtree are less than the key in the root, and all the keys in the right subtree are greater than that in the root.
- The same rule applies to all the subtrees until they have only one key.
- Therefore, given the entries, we can build them into a binary search tree and use inorder traversal to get them in sorted order.

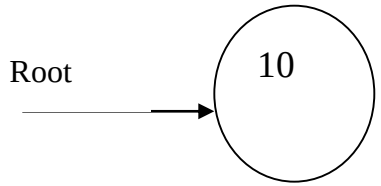
Insertion into a Tree

- Another important operation is to create and maintain a binary search tree.
- While inserting any node, we have to take care the resulting tree satisfies the properties of a binary search tree.
- A new node will always be inserted at its proper position in the binary search tree as a leaf.
- Before writing a routine for inserting a node, consider how a binary tree may be created for the following input:

  10, 15, 12, 7, 8, 18, 6, 20.

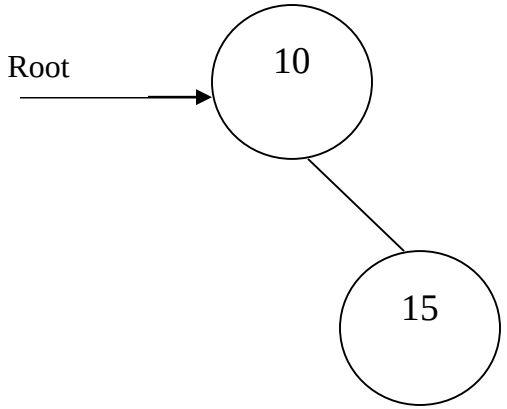
Insertion into a Tree

- First of all, you must initialize the tree.
- To create an empty tree, you must initialize the root to null. The first node will be inserted into the tree as a root node as shown in the following figure.



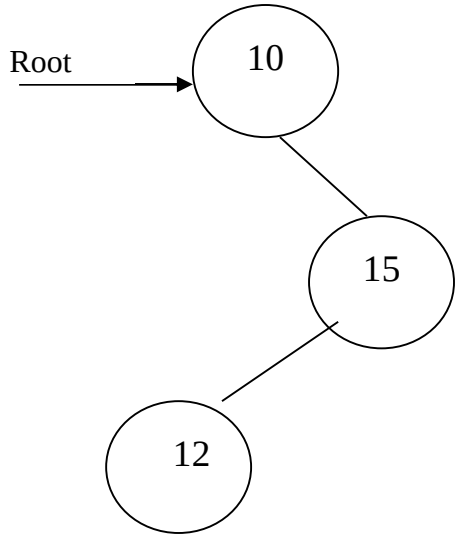
Insertion into a Tree

- Since 15 is greater than 10, it must be inserted as the right child of the root as shown in the following figure.



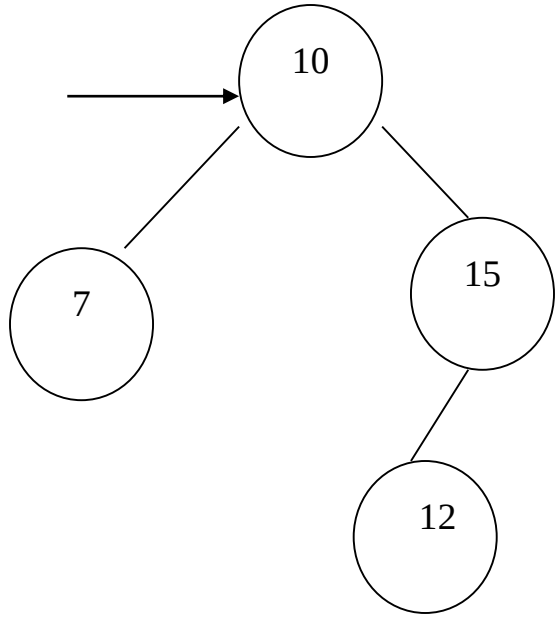
Insertion into a Tree

- Now 12 is larger than the root; it must go to the right subtree of the root.
- Further, since it is smaller than 15, it must be inserted as the left child of the root as shown below.



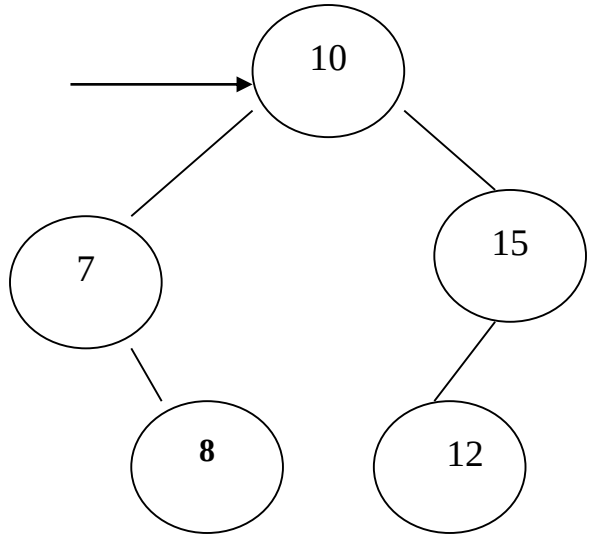
Insertion into a Tree

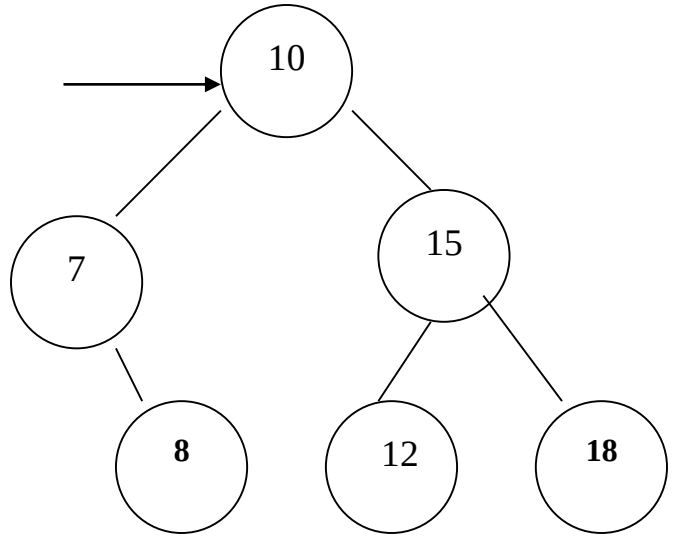
- Next, 7 is smaller than the root. Therefore, it must be inserted as the left child of the root as shown in the following figure.



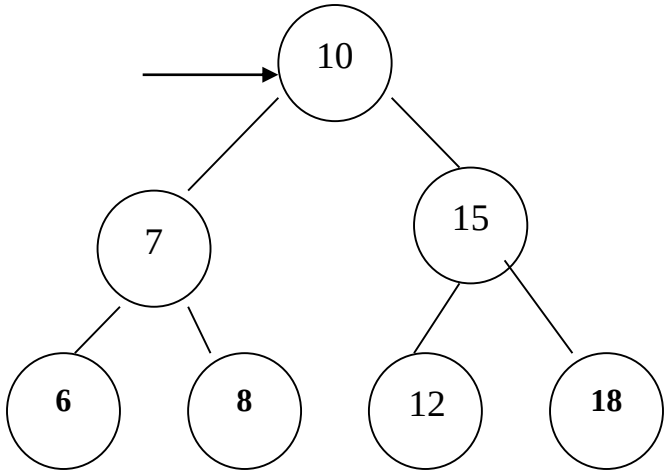
Insertion into a Tree

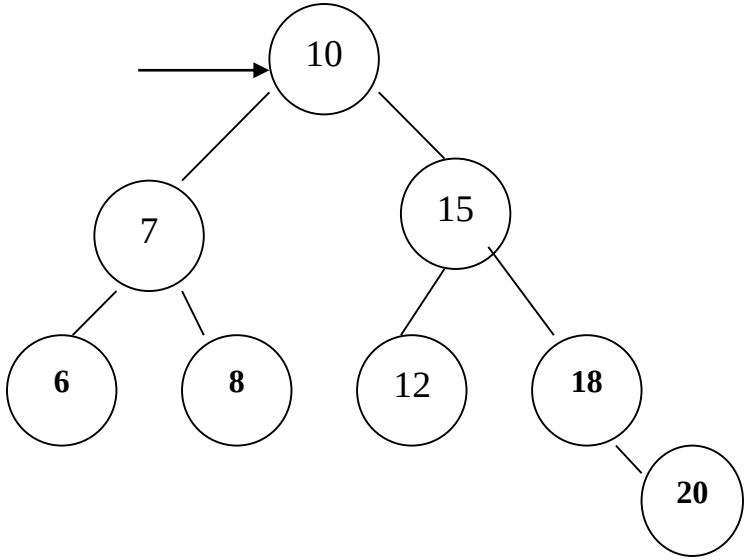
- Similarly, 8, 18, 6 and 20 are inserted at the proper place as shown in the following figures.





Insertion into a Tree





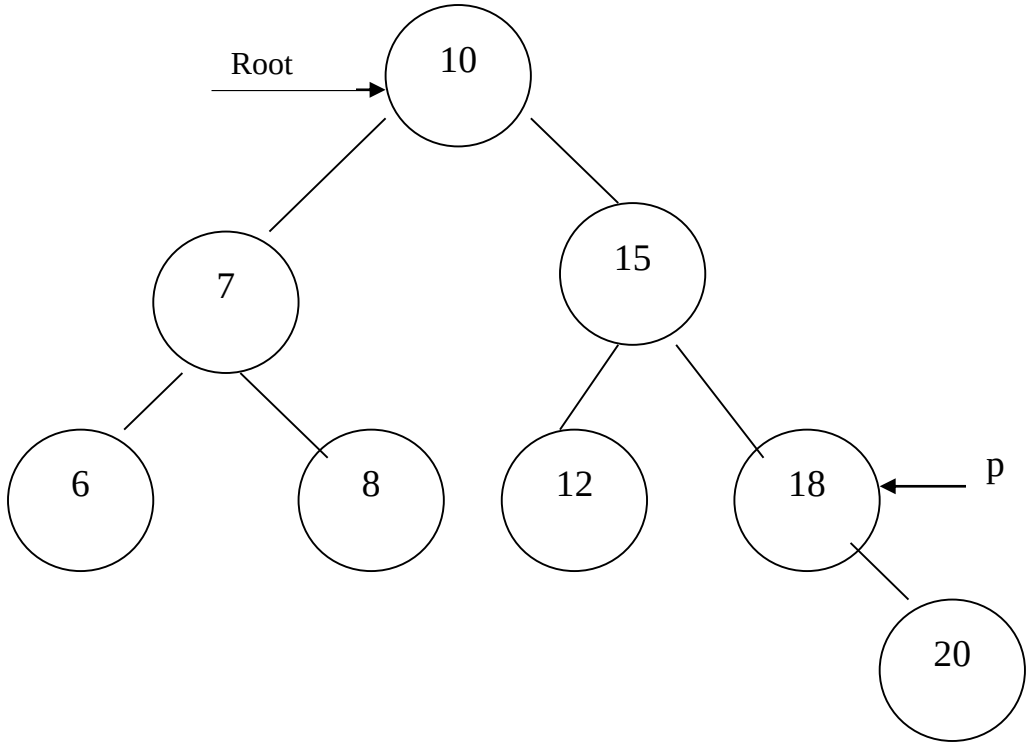
Insertion into a Tree

- This example clearly illustrates that given the root of a binary search tree and a value to be added to the tree, we must search for the proper place where the new value can be inserted.
- We must also create a node for the new value and finally, we have to adjust the left and right pointers to insert the new node.
- To find the insertion place for the new value, say 17, we initialize a temporary pointer p, which points to the root node.

Insertion into a Tree

- We can change the contents of p to either move left or right through the tree depending on the value to be inserted.
- When p becomes null, we know that we have found the insertion place as in the following figure.

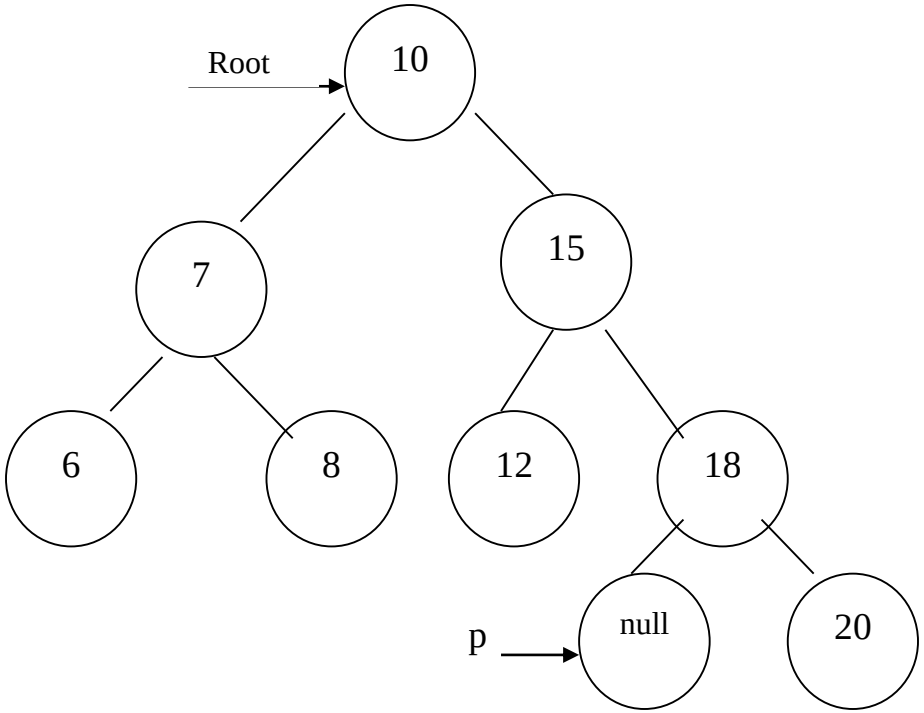
Insertion into a Tree



Insertion into a Tree

- But once p becomes null, it is not possible to link the new node at this position because there is no access to the node that p was pointing to (node with value 18) just before it became null.
- From the following figure, p becomes null when we have found that 17 will be inserted at the left of 18.

Insertion into a Tree



Insertion into a Tree

- You therefore need a way to climb back into the tree so that you can access the node containing 18, in order to make its left pointer point to the new node with the value 17.
- For this, you need a pointer that points to the node containing 18 when p becomes null.
- To achieve this, you need to have another pointer (trail) that must follow p as p moves through the tree.

Insertion into a Tree

- When p becomes null, this pointer will point to the leaf node (the node with value 18) to which you must link the new node (node with value 17).
- Once you know the insertion place, you must adjust the pointers of the new node.
- At this point, you only have a pointer to the leaf node to which the new node is to be linked.
- You must determine whether the insertion is to be done at the left subtree or the right subtree of the leaf node.

Insertion into a Tree

- To do that, you must compare the value to be inserted with the value in the leaf node.
- If the value in the leaf node is greater, we insert the new node as its left child; otherwise we insert the new node as its right child.

Creating a Tree – A Special Case of Insertion

- A special case of insertion that you need to watch out for arises when the tree in which you are inserting a node is an empty tree.
- You must treat it as a special case because when p equals null, the second pointer (trail) trailing p will also be null, and any reference to info of trail like trail->info will be illegal.
- You can check for an empty tree by determining if trail is equal to null. If that is so, we can initialize root to point to the new node.

Code Implementation For Insertion Into a Tree

- The C function for insertion into a binary tree takes two parameters; one is the pointer to the root node (root), and the other is the value to be inserted (x).
- You will implement this algorithm by allocating the nodes dynamically and by linking them using pointer variables. The following is the code implementation of the insert algorithm.

Code Implementation For Insertion Into a Tree

```
tree insert(s,x)
int x;
tree *s;
{
 tree *trail, *p, *q;
 q = (struct tree *) malloc (sizeof(tree));
 q->info = x;
 q->left = null;
 q->right = null;
 p = s;
 trail = null;
```

Code Implementation For Insertion Into a Tree

```
while (p != null)
{
 trail = p;
 if (x < p->info)
 {
  p = p->left;
 }
 else
 {
  p = p->right;
 }
}
```

Code Implementation For Insertion Into a Tree

```
/*insertion into an empty tree; a special case of insertion */
if (trail == null)
{
 s = q;
 return (s);
}
if(x < trail->info)
{
 trail->left = q;
}
else
{
 trail->right = q;
}
return (s);
}
```

Code Implementation For Insertion Into a Tree Using Recursion

- You have seen that to insert a node, you must compare x with root->info.
- If x is less than root->info, then x must be inserted into the left subtree.
- Otherwise, x must be inserted into the right subtree.
- This description suggests a recursive method where you compare the new value (x) with the one in the root and you use exactly the same insertion method either on the left subtree or on the right subtree.

Code Implementation For Insertion Into a Tree Using Recursion

The base case is inserting a node into an empty tree.

You can write a recursive routine (rinsert) to insert a node recursively as follows:

```
tree rinsert (s,x)
tree *s;
int x;
 {
  /* insertion into an empty tree; a special case
  of insertion */
  if (!s)
   {
   s=(struct tree*) malloc (sizeof(struct tree));
```

Code Implementation For Insertion Into a Tree Using Recursion

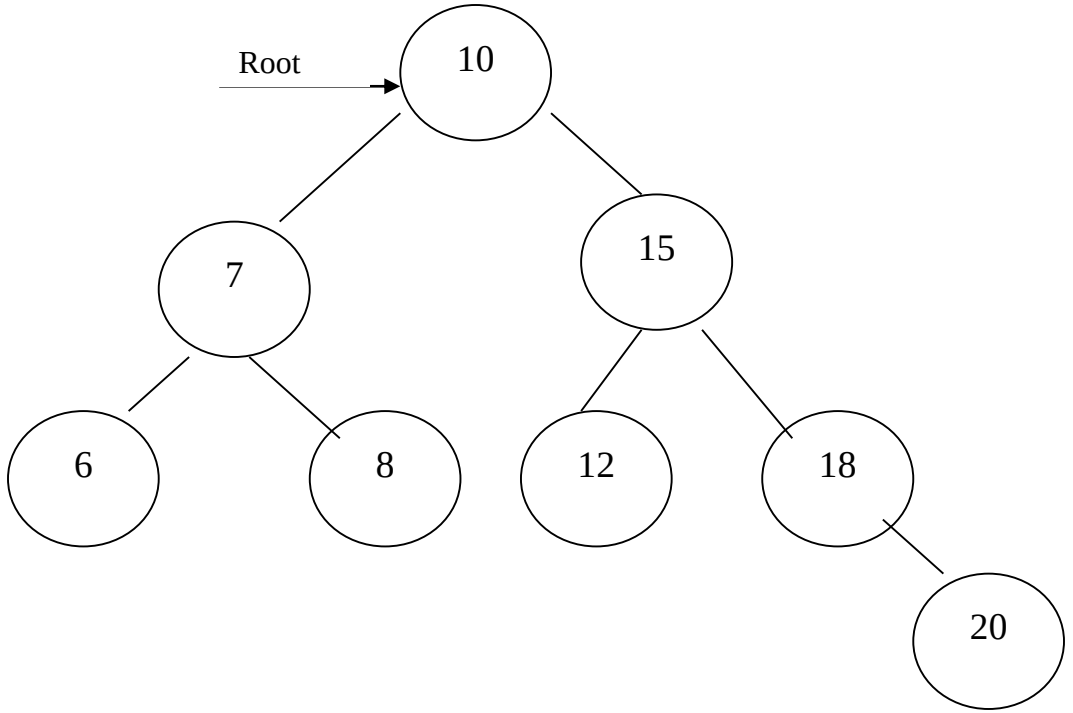
```
s->info = x;
  s->left = null;
  s->right = null;
  return (s);
 }
 if (x < s->info)
  s->left = rinsert(x, s->left);
 else
  if (x > s->info)
  s->right = rinsert(x, s->right);
return (s);
}
```

Circumstances When a Binary Tree Degenerates Into a Linked List

- The shape of a binary tree is determined by the order in which the nodes are inserted.
- Given the following input, their insertion into the tree in the same order would more or less produce a balanced binary search tree as shown below:

Input values: 10, 15, 12, 7, 8, 18, 6, 20

Circumstances When a Binary Tree Degenerates Into a Linked List

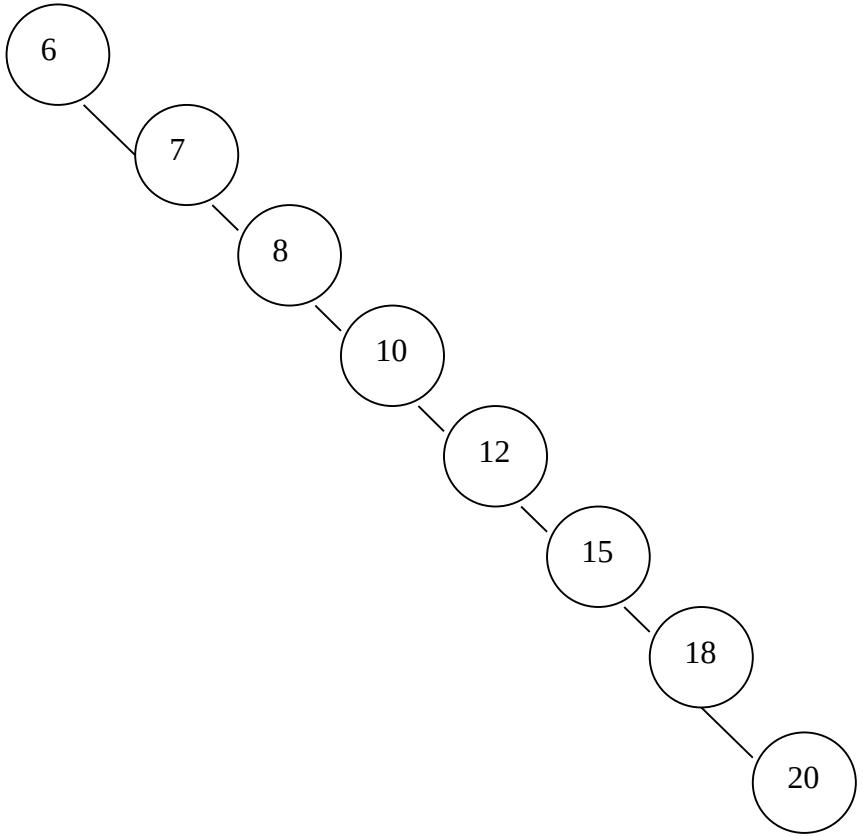


Circumstances When a Binary Tree Degenerates Into a Linked List

If the same input is given in the sorted order as

- 6, 7, 8, 10, 12, 15, 18, 20, you will construct a lopsided tree with only right subtrees starting from the root.
- Such a tree will be conspicuous by the absence of its left subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



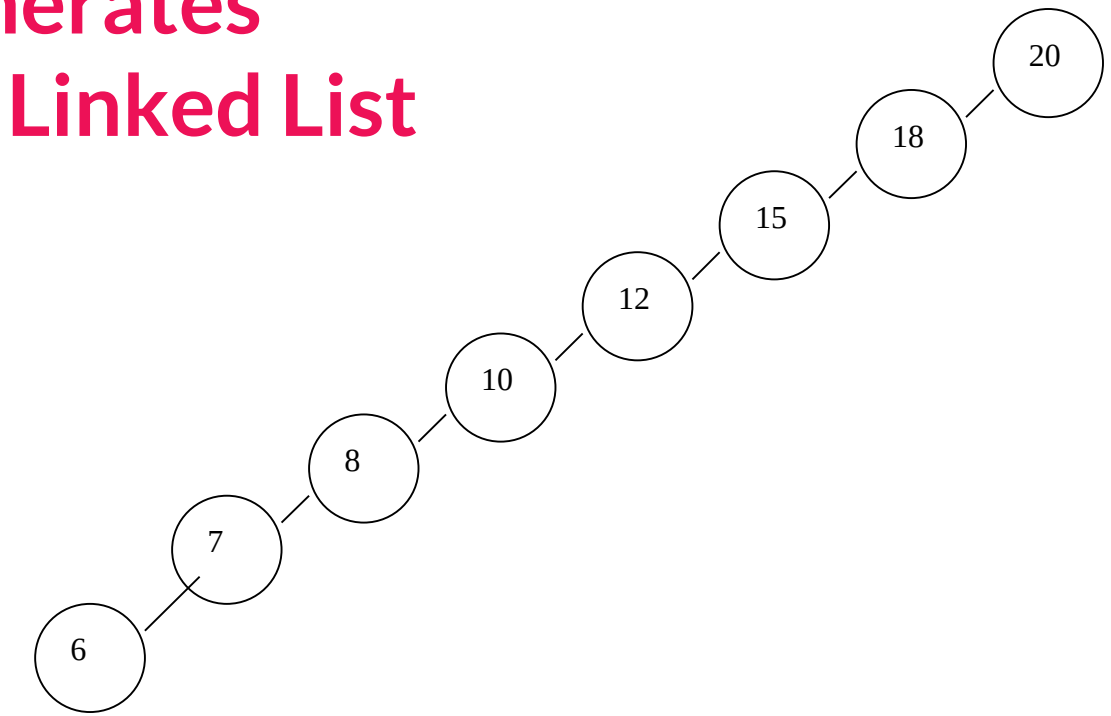
A Lopsided Binary Tree With Only Right Subtrees

Circumstances When a Binary Tree Degenerates Into a Linked List

However if you reverse the input as

- 20, 18, 15, 12, 10, 8, 7, 6, and insert them into a tree in the same sequence, you will construct a lopsided tree with only the left subtrees starting from the root.
- Such a tree will be conspicuous by the absence of its right subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



A Lopsided Binary Tree With Only Left Subtrees

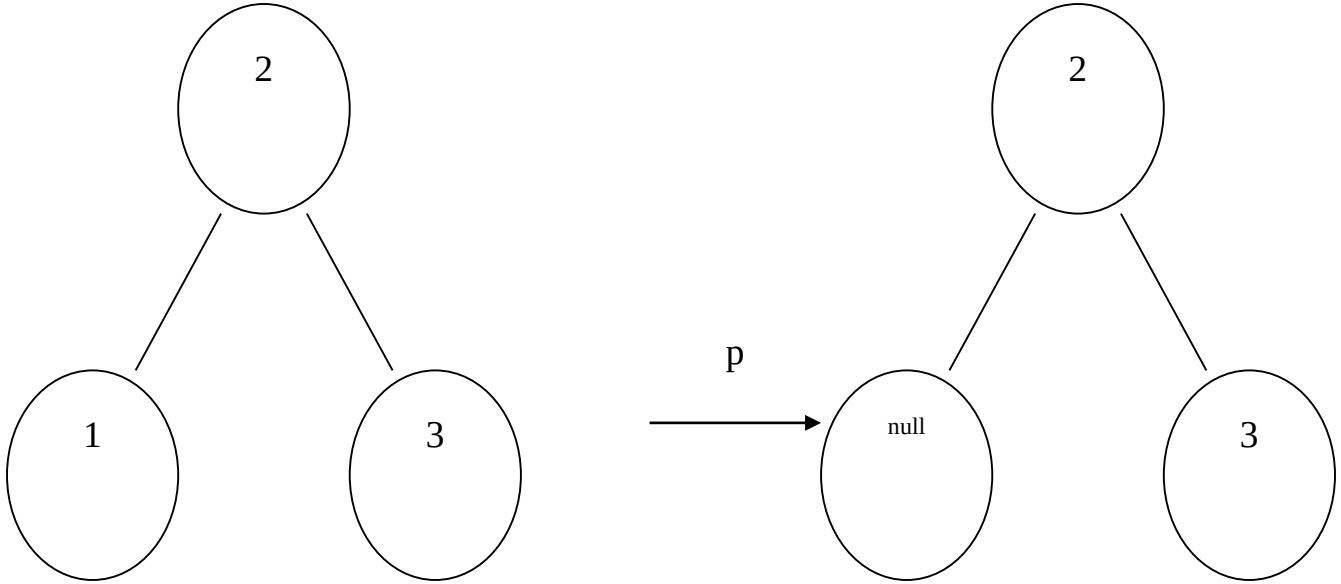
Deletion from a Binary Search Tree

- An important function for maintaining a binary search tree is to delete a specific node from the tree.
- The method to delete a node depends on the specific position of the node in the tree.
- The algorithm to delete a node can be subdivided into different cases.

Case I – Deletion Of The Leaf Node

- If the node to be deleted is a leaf, you only need to set appropriate link of its parent to null, and do away with the node that is to be deleted.
- For example, to delete a node containing 1 in the following figure, we have to set the left pointer of its parent (pointing to 1) to null.
- The following diagram illustrates this.

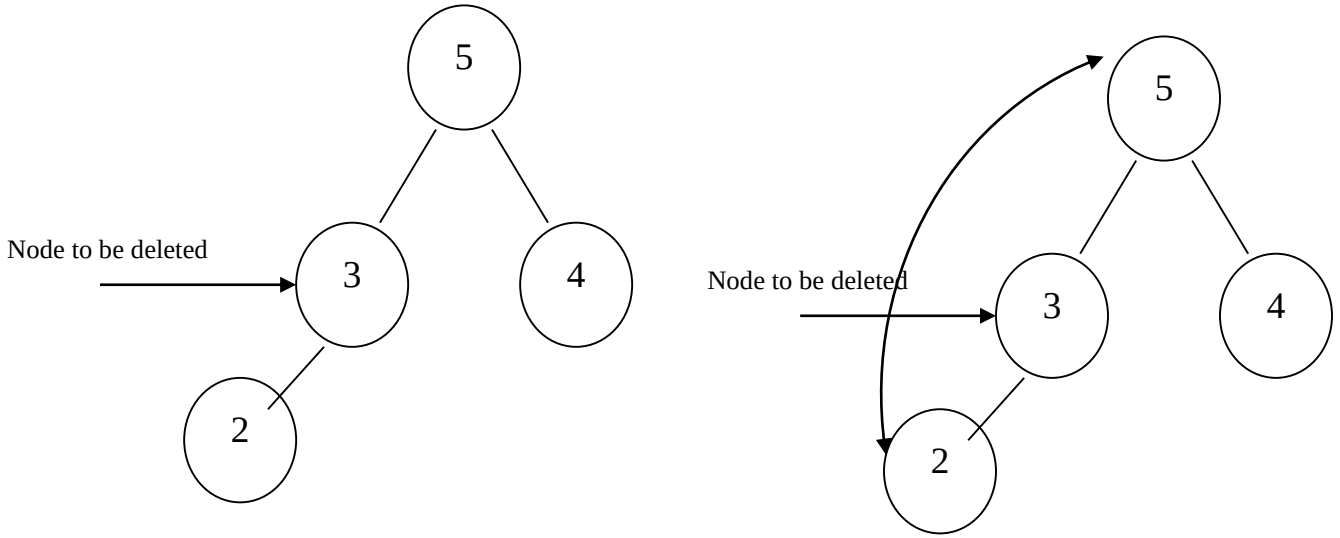
Case I – Deletion Of The Leaf Node



Case II – Deletion Of a Node With a Single Child

- If the node to be deleted has only one child, you cannot simply make the link of the parent to nil as you did in the case of a leaf node.
- Because if you do so, you will lose all of the descendants of the node that you are deleting from the tree.
- So, you need to adjust the link from the parent of deleted node to point to the child of the node you intend to delete. You can subsequently dispose of the deleted node.

Case II – Deletion Of a Node With a Single Child



- To delete node containing the value 3, where the right subtree of 3 is empty, we simply make the link of the parent of the node with the value 3 (node with value 5) point to the child of 3 (node with the value 2).

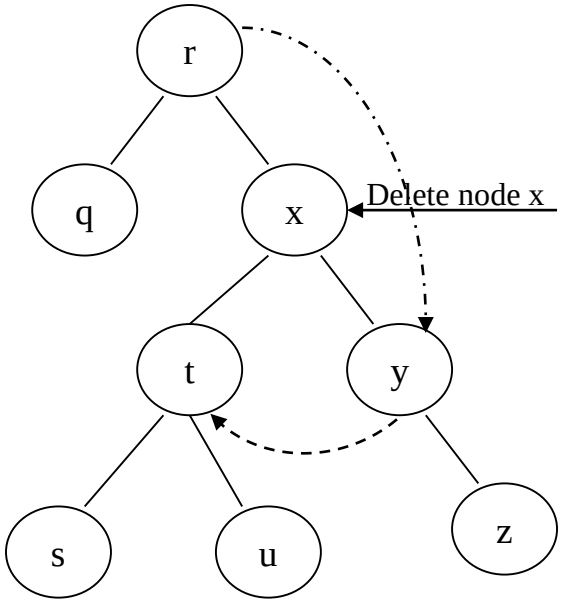
Case III – Deletion Of a Node With Two Child Nodes

- Complications arise when you have to delete a node with two children.
- There is no way you can make the parent of the deleted node to point to both of the children of the deleted node.
- So, you attach one of the subtrees of the node to be deleted to the parent, and then link the other subtree onto the appropriate node of the first subtree.

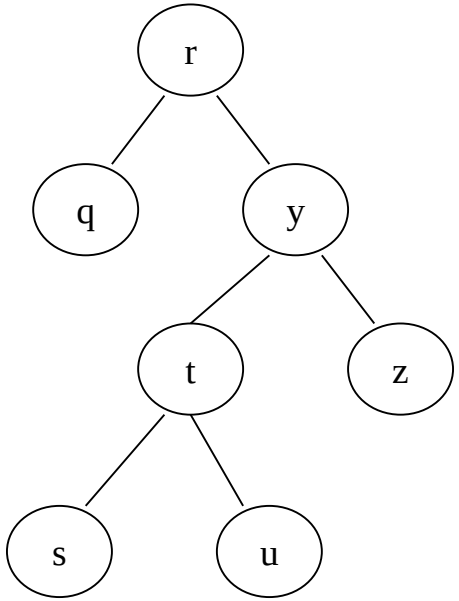
Case III – Deletion Of a Node With Two Child Nodes

- You can attach the right subtree to the parent node and then link the left subtree on to the appropriate node of the right subtree.
- Therefore, you must attach the left subtree as far to the left as possible. This proper place can be found by going left until an empty left subtree is found.
- For example, if you delete the node containing x as shown in the following figure, you make the parent of x (node with the value r) point to the right subtree of x (node containing y) and then go as far left as possible (to the left of the node containing y) and attach the left subtree there.

Case III – Deletion Of a Node With Two Child Nodes



Before Deletion of Node x



After Deletion of Node x

Code Implementation for Node Deletion for Cases I, II & III

```
void delete (p)
Struct tree *p
 {
  struct tree *temp
  if (p == null)
  printf("Trying to delete a non-existent node");
  else if (p->left == null)
  {
   temp = p;
   p = p->right;
   free(temp);
  }
  else if (p->right == null)
  {
   temp = p;
   p = p->left;
   free (temp);
  }
```

Code Implementation for Node Deletion for Cases I, II & III

```
else if(p->left != null && p->right!= null)
   {
    temp = p->right;
    while (temp->left != null)
     {
      temp = temp->left;
     }
    temp->left = p->left;
    temp = p;
    p = p->right;
    free (Temp);
   }
 }
```

Code Implementation for Node Deletion for Cases I, II & III

- Note that the while loop stops when it finds a node with an empty left subtree so that the left subtree of the node to be deleted can be attached here.
- Also, note that you first attach the left subtree at the proper place and then attach the right subtree to the parent node of the node to be deleted.

Search The Tree

- To search a tree, you employ a traversal pointer p, and set it equal to the root of the tree.
- Then you compare the information field of p with the given value x. If the information is equal to x, you exit the routine and return the current value of p.
- If x is less than p->info, you search in the left subtree of p.
- Otherwise, you search in the right subtree of p by making p equal to p->right.

Search the Tree

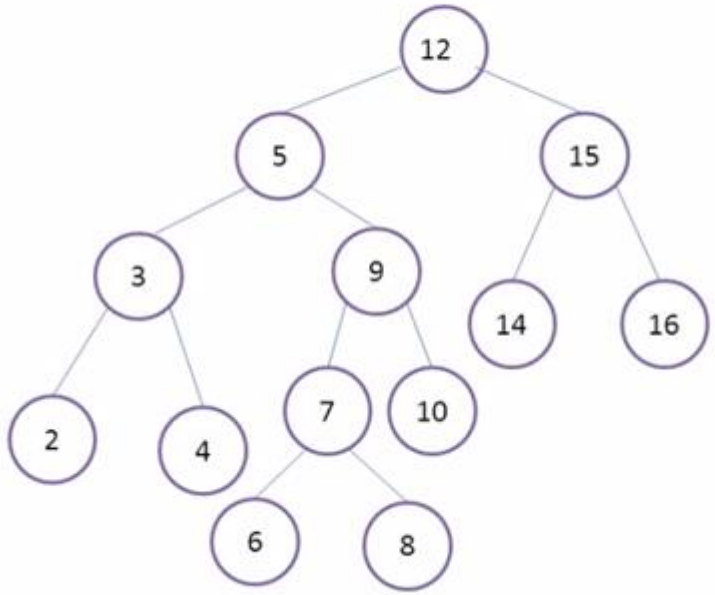
You continue searching until you have found the desired value or reach the end of the tree. You can write the code implementation for a tree search as follows:

```
search (p,x)
int x;
struct tree *p;
 {
  p = root;
  while (p != null && p->info != x)
   {
    if (p->info > x)
     p = p->left;
    else
     p = p->right;
   }  return (p);
  }
```

Binary Tree Delete

LearnOA®

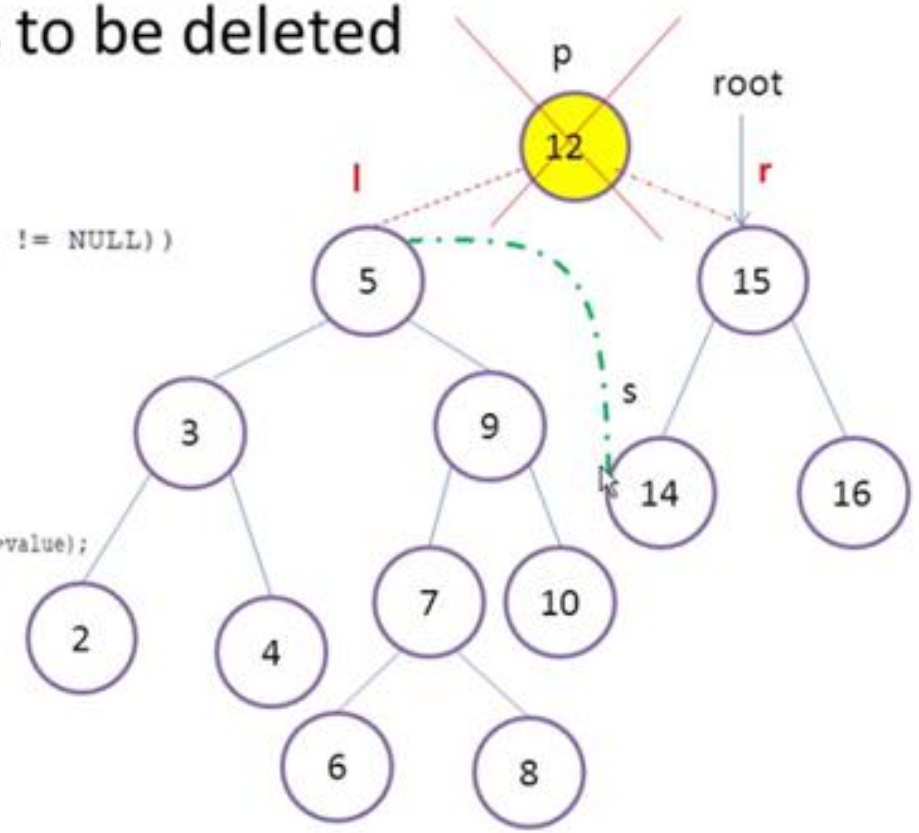
DELETION OF NODE IN BINARY TREE



Deletion of element from the binary tree takes place in different cases

CASE – 1 :- ROOT is to be deleted

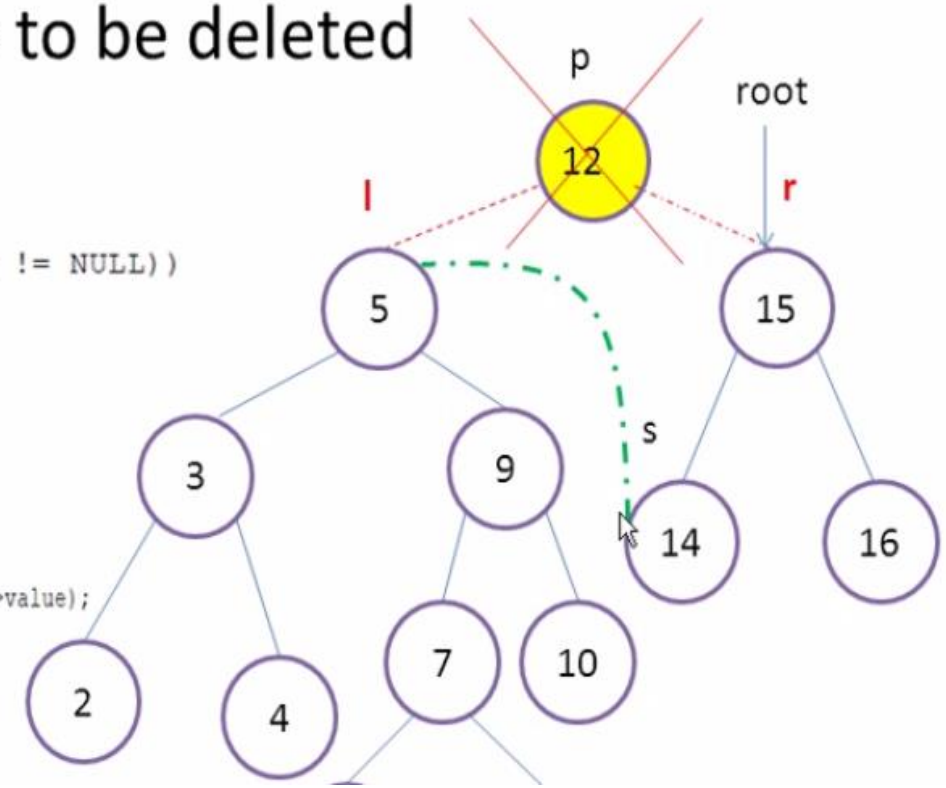
```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;


      if(r!= NULL)
      {    root = r ; s->left = l; }
      else
           root=l;

      printf("CASE-0 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
  }
```



Deletion of element from the binary tree takes place in different cases

CASE – 1 :- ROOT is to be deleted

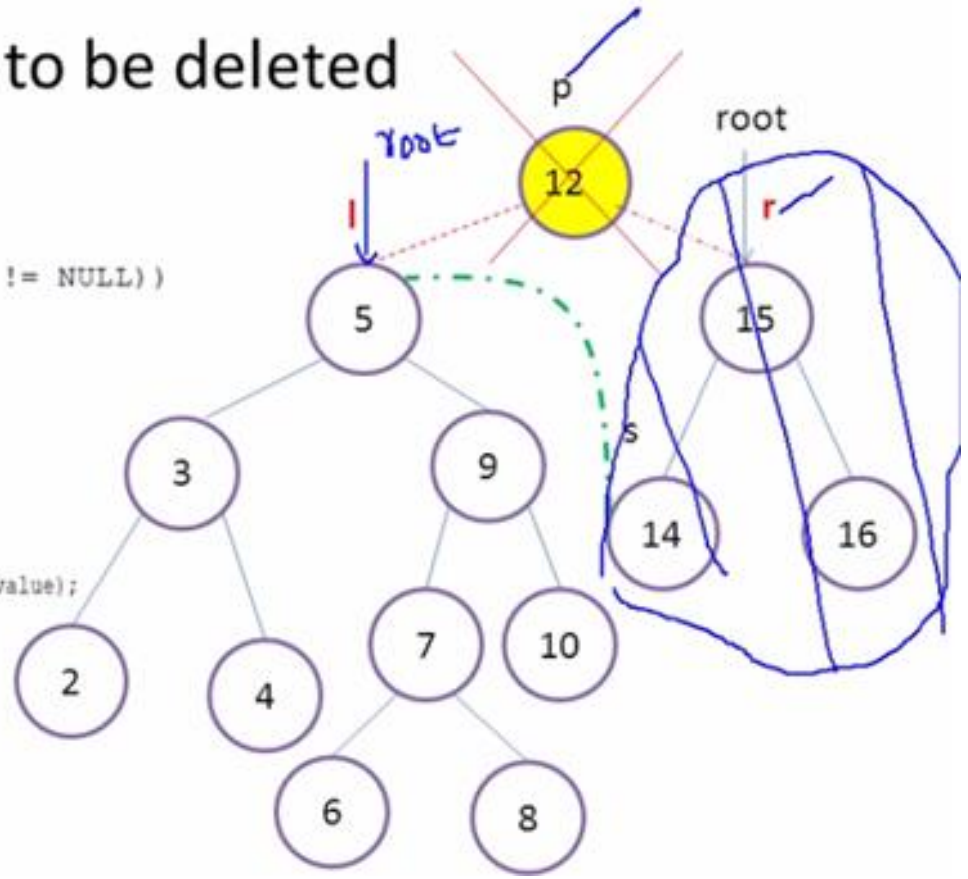
```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;


      if(r!= NULL)
      {     root = r ; s->left = l; }
      else
            root=l;

      printf("CASE-8 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
  }
```



Deletion of element from the binary tree takes place in different cases

CASE – 1 :- ROOT is to be deleted

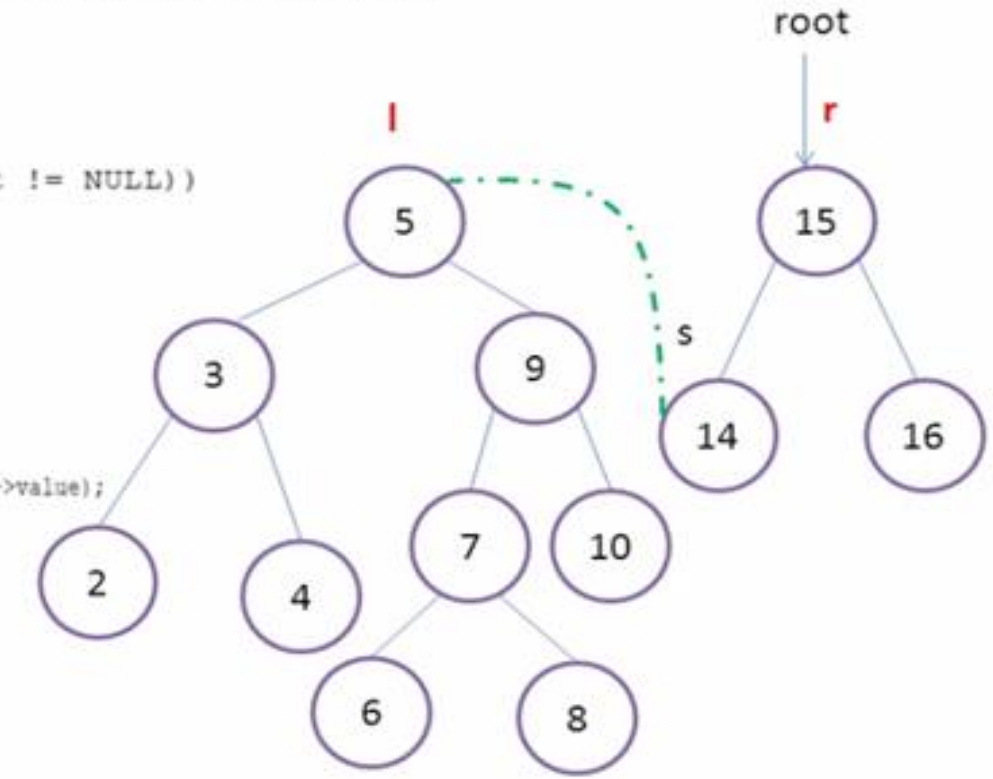
```
if(p!=NULL && value == p->value)
{
       l=p->left;  r=p->right;
       s=r;
       while((s!= NULL) && (s->left != NULL))
             s=s->left;


   + if(r!= NULL)
     {    root = r ; s->left = l; }
     else
          root=l;

     printf("CASE-8 Element deleted is %d\n",p->value);
     free(p);
     return(DONE);
  }
```



Deletion of element from the binary tree takes place in different cases

CASE – 1 :- ROOT is to be deleted

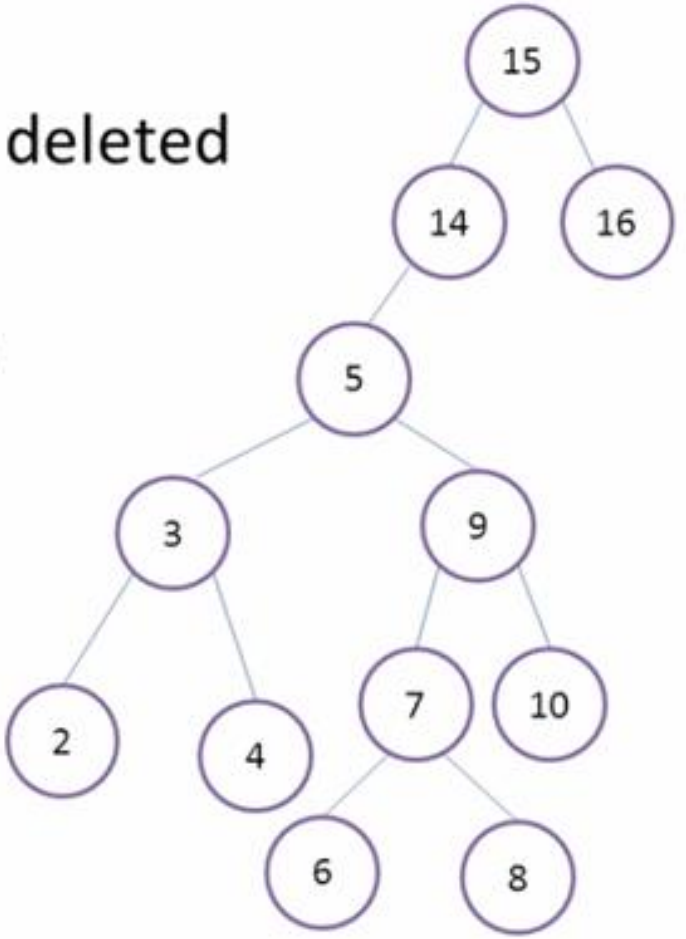
```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;


      if(r!= NULL)
      {     root = r ; s->left = l; }
      else
           root=l;

      printf("CASE 0 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
  }
```



Deletion of element from the binary tree takes place in different cases

CASE – 1 :- ROOT is to be deleted

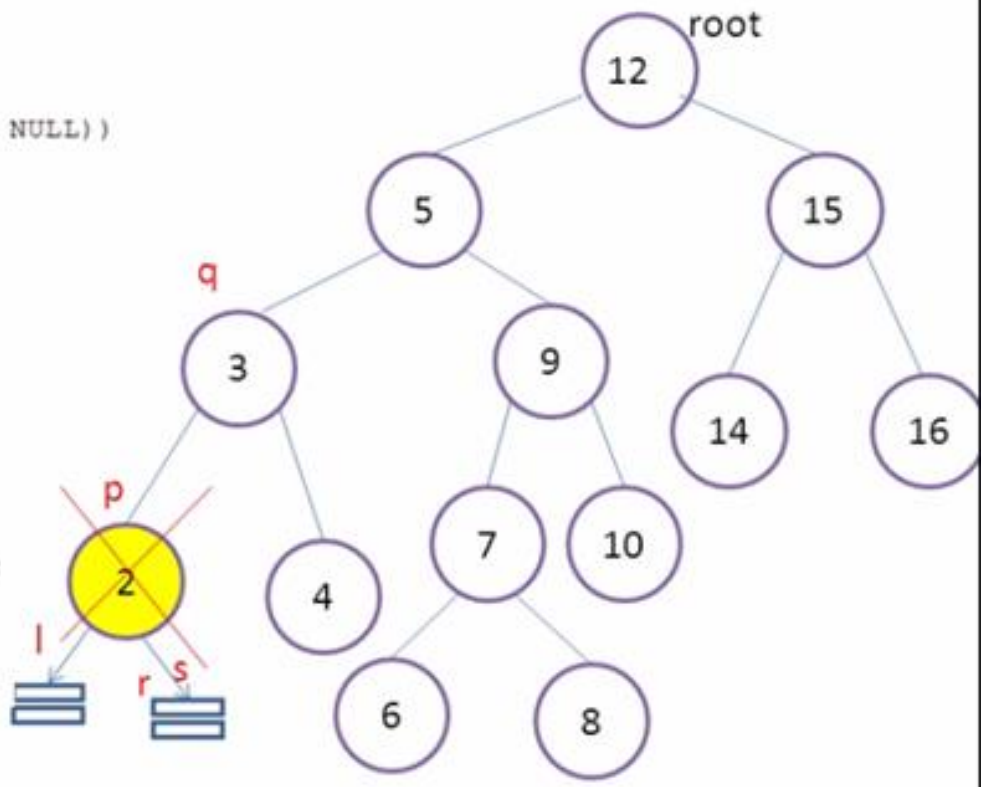
```
if(p!=NULL && value == p->value)
{
      l=p->left;  r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;


      if(r!= NULL)
      {    root = r ; s->left = l; }
      else
           root=l;

      printf("CASE 3 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
  }
```



CASE – 2 :- Node to be deleted is left of it's parent say value 2

```
q=p;
if(value < p->value) p=p->left;
```

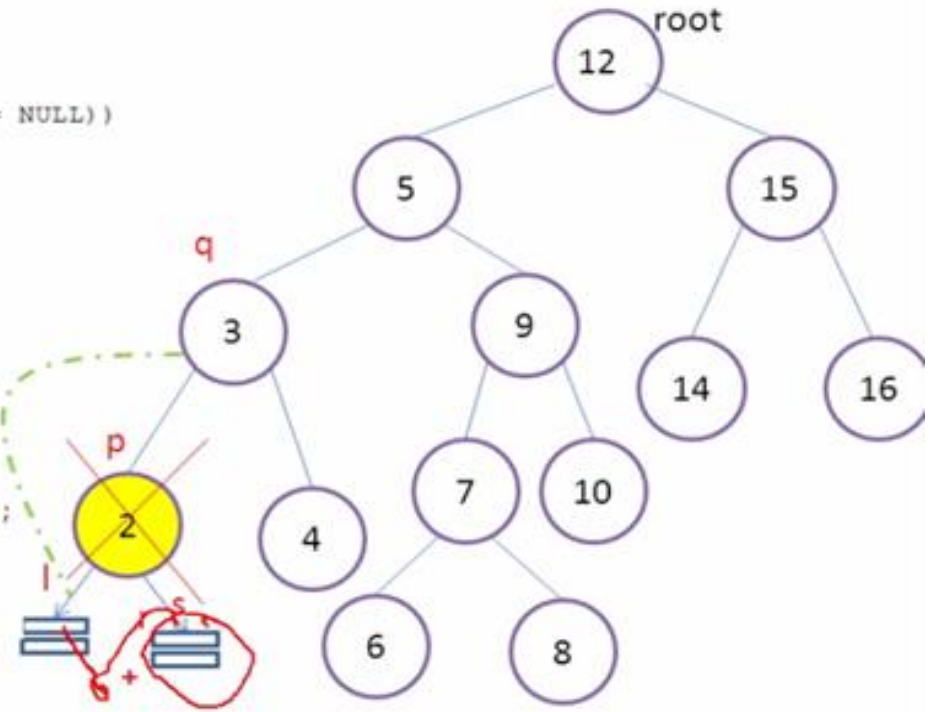
```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;

 else if(q->left == p )
 {
      q->left = l;
      while((l!= NULL) && (l->right != NULL))
            l=l->right;


      if(l!=NULL)
           l->right =r;
      printf("CASE-7 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
 }
```



CASE – 2 :- Node to be deleted is left of it's parent say value 2

```
q=p;
if(value < p->value) p=p->left;
```

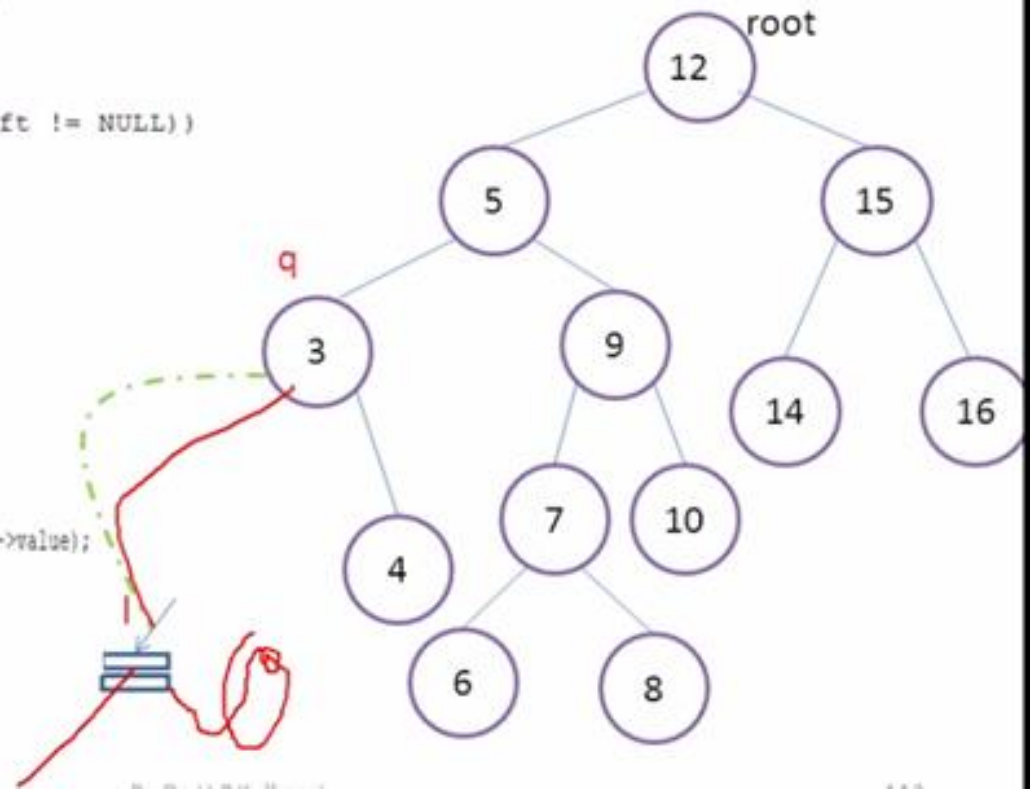
```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;

 else if(q->left == p )
 {
      q->left = l;
      while((l!= NULL) && (l->right != NULL))
            l=l->right;


      if(l!=NULL)
           l->right =r;
      printf("CASE-7 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
 }
```



CASE – 2 :- Node to be deleted is left of it's parent say value 2

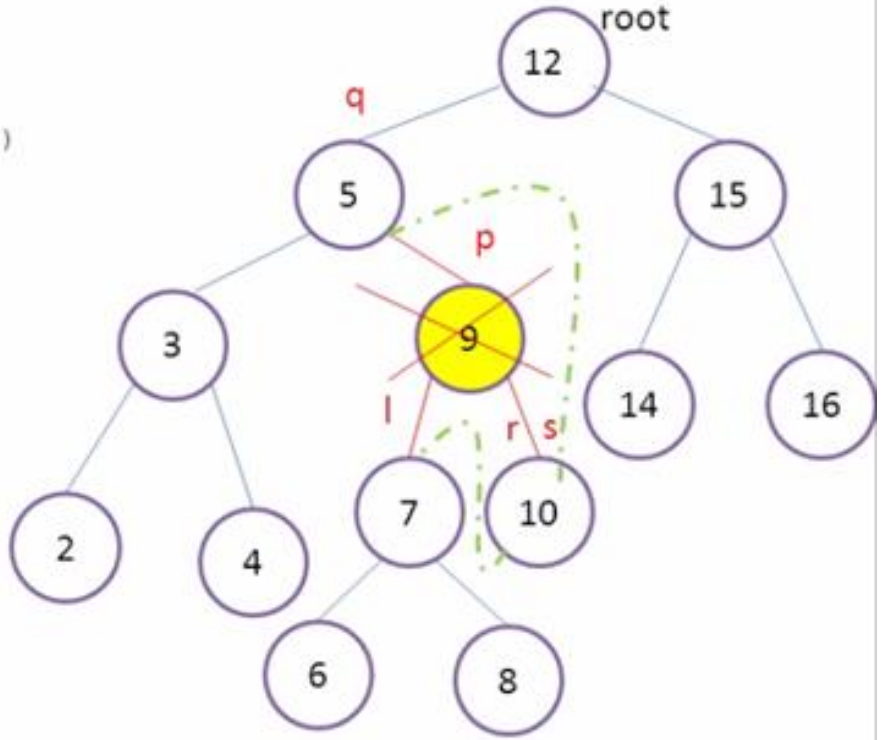
```
q=p;
if(value < p->value) p=p->left;
```

```
if(p!=NULL && value == p->value)
{
      l=p->left;   r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;

 else if(q->left == p )
 {
      q->left = l;
      while((l!= NULL) && (l->right != NULL))
                l=l->right;


      if(l!=NULL)
          l->right =r;
      printf("CASE-7 Element deleted is %d\n",p->value);
      free(p);
      return(DONE);
 }
```



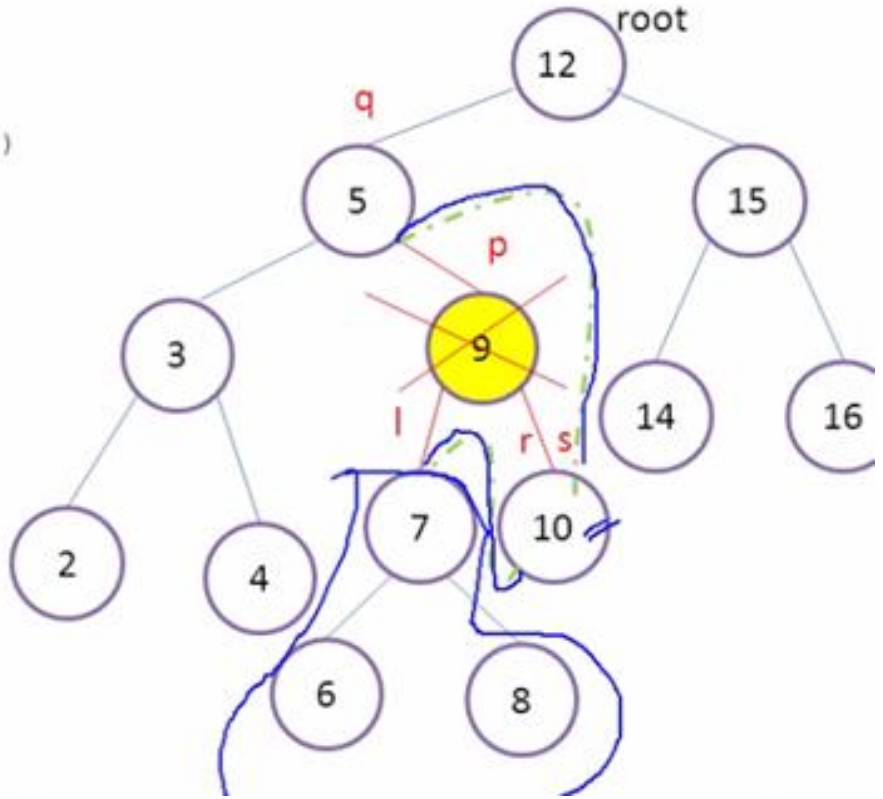
CASE – 3 :- Node to be deleted is right of it's parent node say value 9

```
 q=p;


 if(value > p->value) p=p->right;

if(p!=NULL && value == p->value)
{
      l=p->left;  r=p->right;
      s=r;
      while((s!= NULL) && (s->left != NULL))
            s=s->left;


  if(q->right == p)
  {
        if(r!=NULL)
              q->right = r;
        else


      q->right = l;

 if(s!=NULL)
      s->left = l;

 printf("CASE-6 Element deleted is %d\n",p->value);
 free(p);
 return(DONE);
```



CASE – 3 :- Node to be deleted is right of it's parent node say value 9

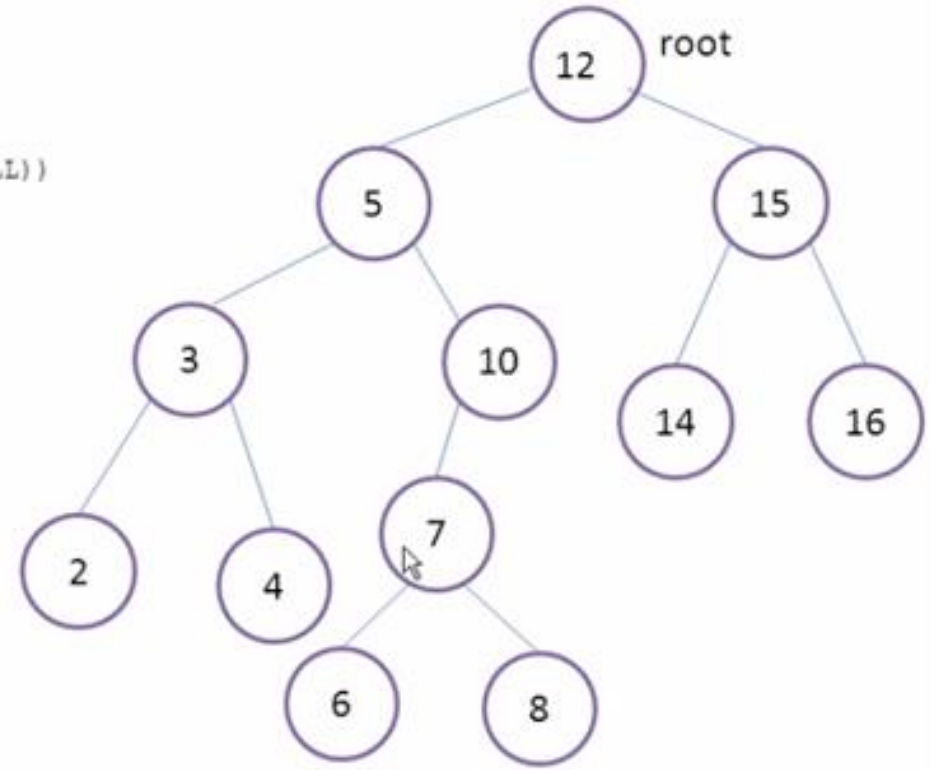
```
 q=p;

else
if(value > p->value) p=p->right;

if(p!=NULL && value == p->value)
{
     l=p->left;  r=p->right;
     s=r;
     while((s!= NULL) && (s->left != NULL))
          s=s->left;


 if(q->right == p)
 {
       if(r!=NULL)
            q->right = r;
       else


     q->right = l;

if(s!=NULL)
     s->left = l;

printf("CASE-6 Element deleted is %d\n",p->value);
free(p);
return(DONE);
```



CASE – 3 :- Node to be deleted is right of it's parent node say value 9

```
 q=p;

 else
 if(value > p->value) p=p->right;
```

```
if(p!=NULL && value == p->value)
{
     l=p->left;  r=p->right;
     s=r;
     while((s!= NULL) && (s->left != NULL))
          s=s->left;


  if(q->right == p)
  {
        if(r!=NULL)
             q->right = r;
        else


     q->right = l;

 if(s!=NULL)
     s->left = l;

 printf("CASE-6 Element deleted is %d\n",p->value);
 free(p);
 return(DONE);
```



```
int delete(struct tree *p,int value)
{
    struct tree *q,*s,*l,*r;

    while(p!=NULL)
    {
        q=p;
        if(value < p->value) p=p->left;
        else
        if(value > p->value) p=p->right;

        if(p!=NULL && value == p->value)
        {
            l=p->left;  r=p->right;
            s=r;
            while((s!= NULL) && (s->left != NULL))
                s=s->left;

        if(q->right == p)
        {
            if(r!=NULL)
                q->right = r;
            else
                q->right = l;

            if(s!=NULL)
                s->left = l;

            printf("CASE-6 Element deleted is %d\n",p->value);
            free(p);
            return(DONE);


        }
        else if(q->left == p )
        {
            q->left = l;
            while((l!= NULL) && (l->right != NULL))
                    l=l->right;


            if(l!=NULL)
                l->right =r;
            printf("CASE-7 Element deleted is %d\n",p->value);
            free(p);
            return(DONE);
        }
        else
        {
            if(r!= NULL)
            (   root = r ; s->left = l; )
            else
                root=l;

            printf("CASE-8 Element deleted is %d\n",p->value);
            free(p);
            return(DONE);
        }
```

AVL TREE

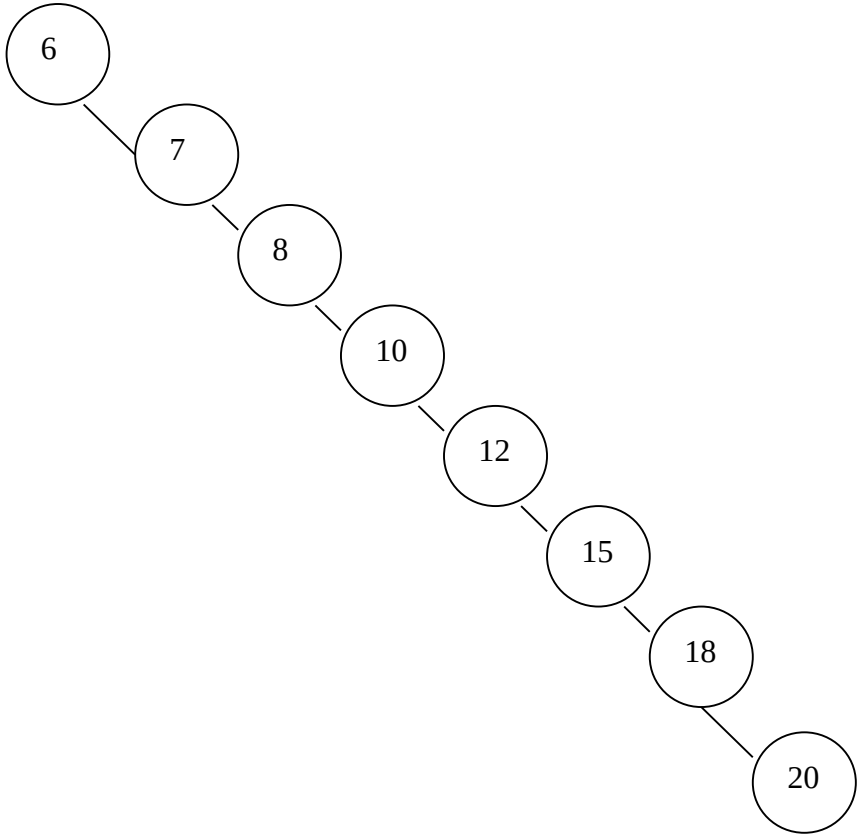
LearnOA®

Circumstances When a Binary Tree Degenerates Into a Linked List

If the same input is given in the sorted order as

- 6, 7, 8, 10, 12, 15, 5, 3,18, 20, you will construct a lopsided tree with only right subtrees starting from the root.
- Such a tree will be conspicuous by the absence of its left subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



A Lopsided Binary Tree With Only Right Subtrees

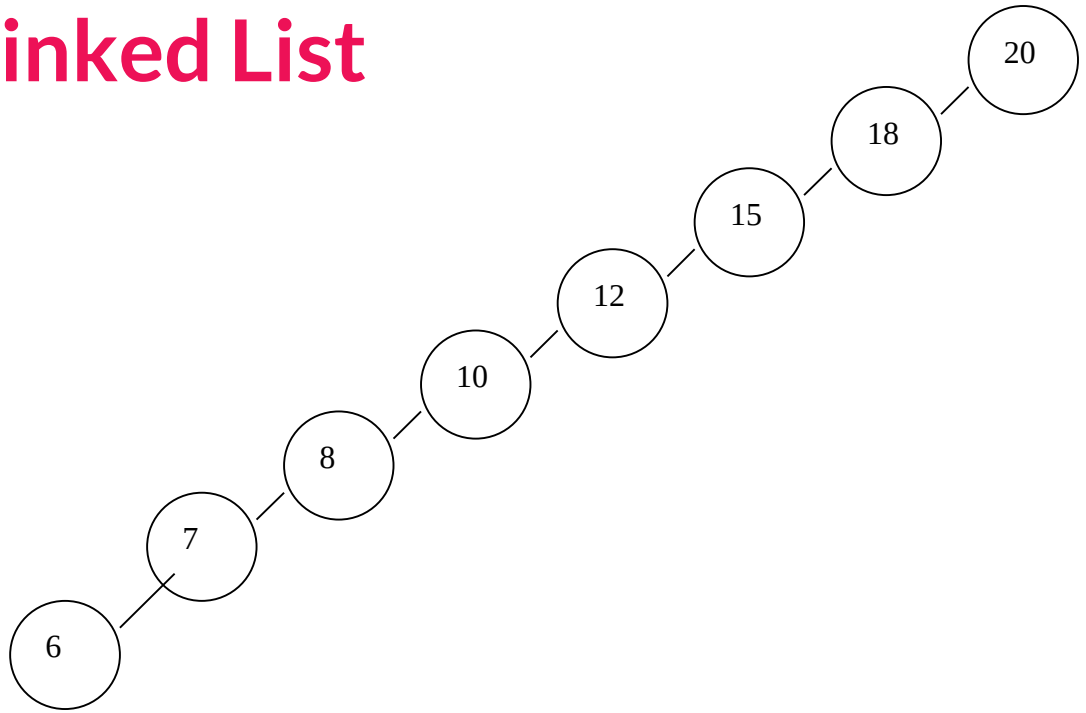
Circumstances When a Binary Tree Degenerates Into a Linked List

However if you reverse the input as

- 20, 18, 15, 12, 10, 8, 7, 6, and insert them into a tree in the same sequence, you will construct a lopsided tree with only the left subtrees starting from the root.

- Such a tree will be conspicuous by the absence of its right subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



A Lopsided Binary Tree With Only Left Subtrees

Introduction

Adelson-Velskii and Landis

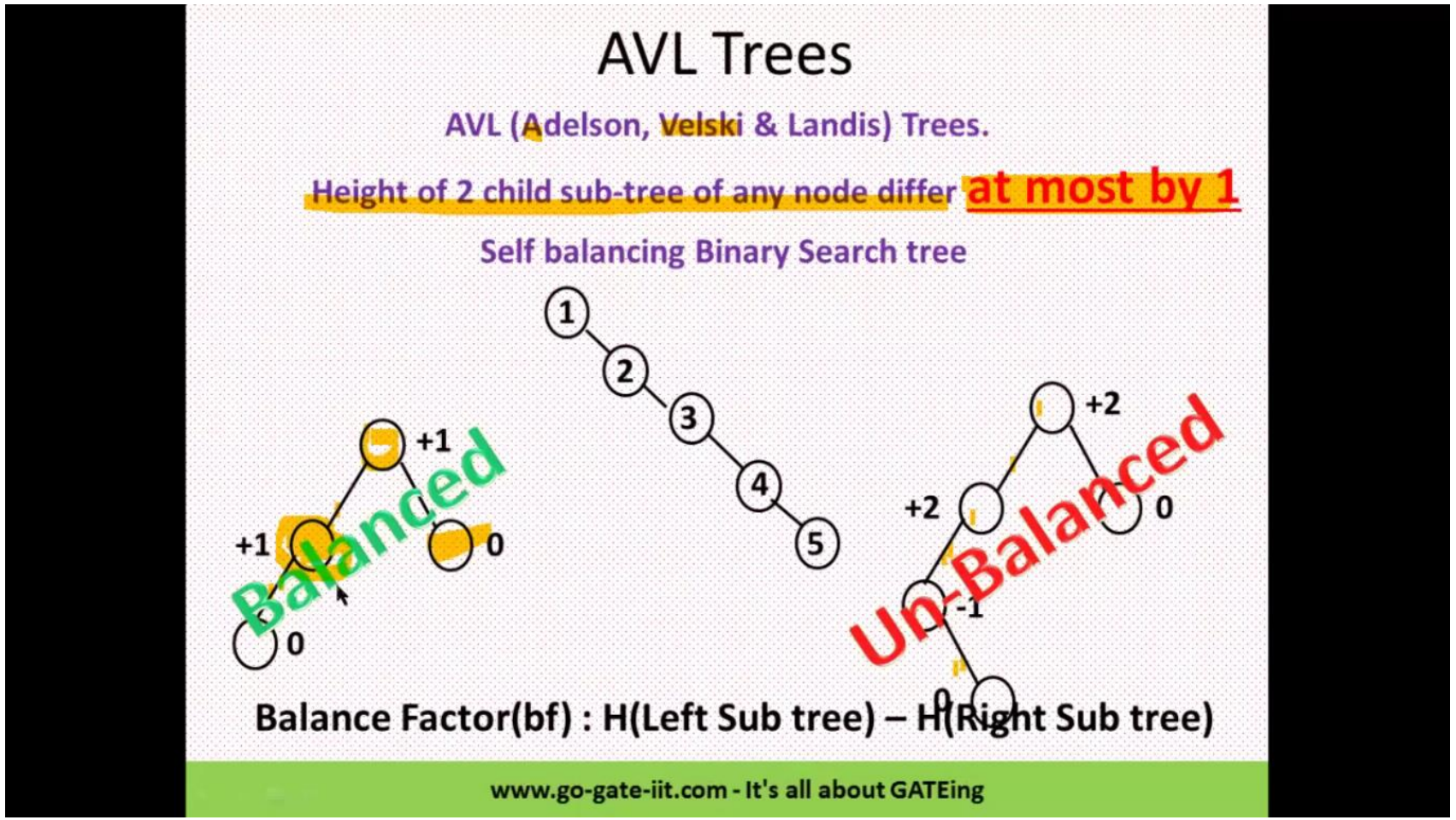
Complete binary tree is hard to build when we allow dynamic insert and remove.

We want a tree that has the following properties

Tree height = O(log(N))

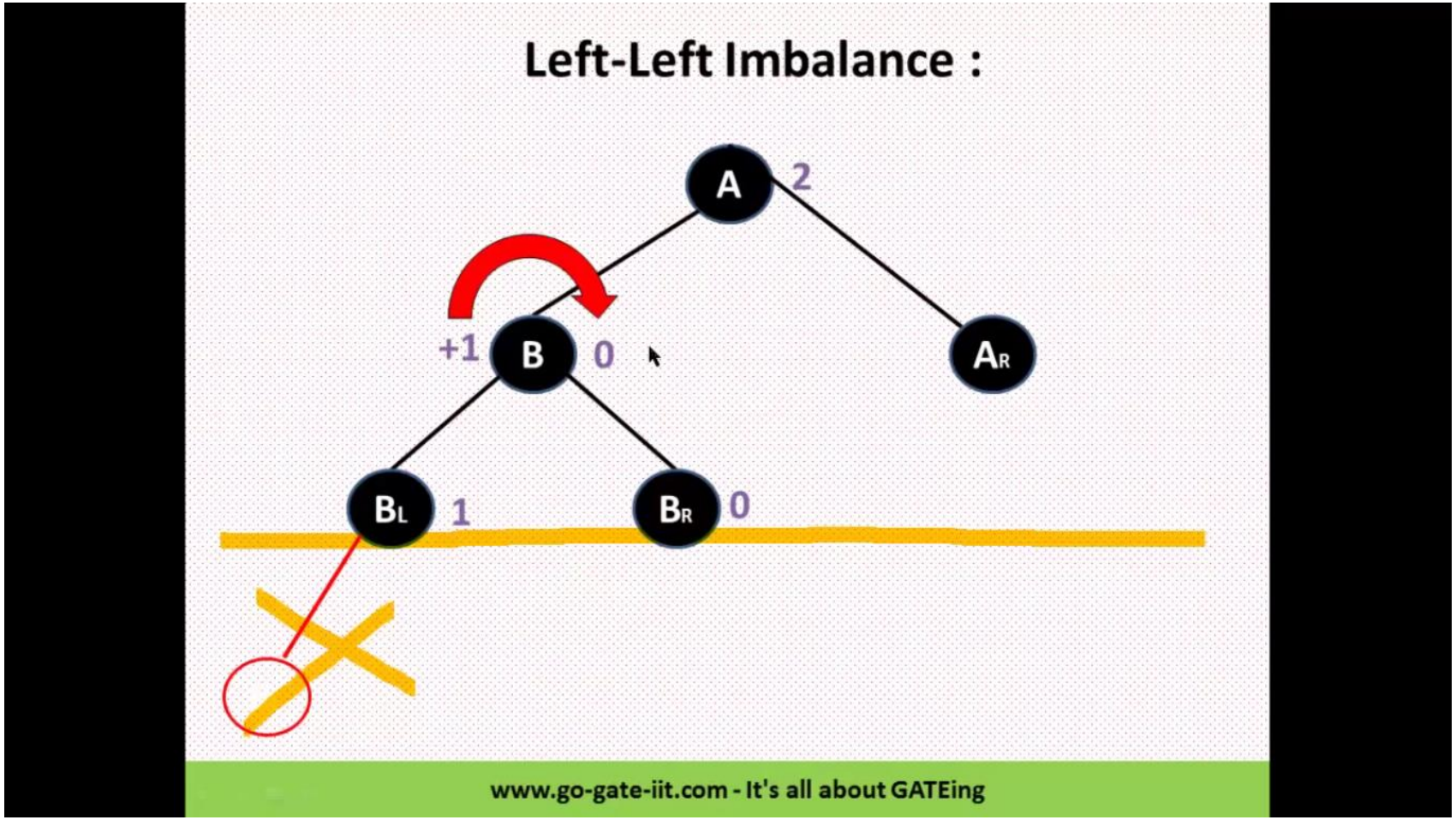
allows dynamic insert and remove with O(log(N)) time complexity.

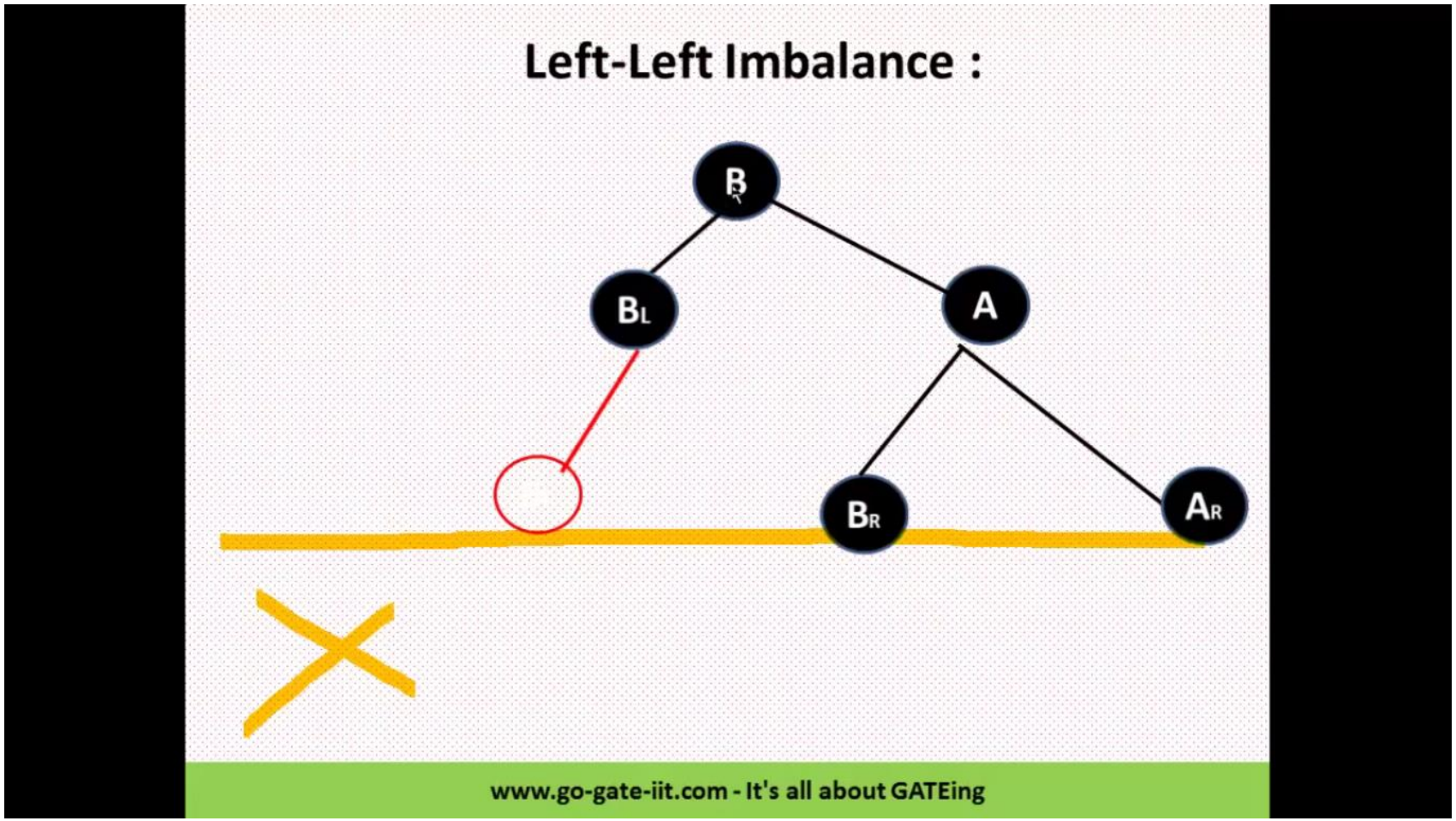
The AVL tree is one of this kind of trees.

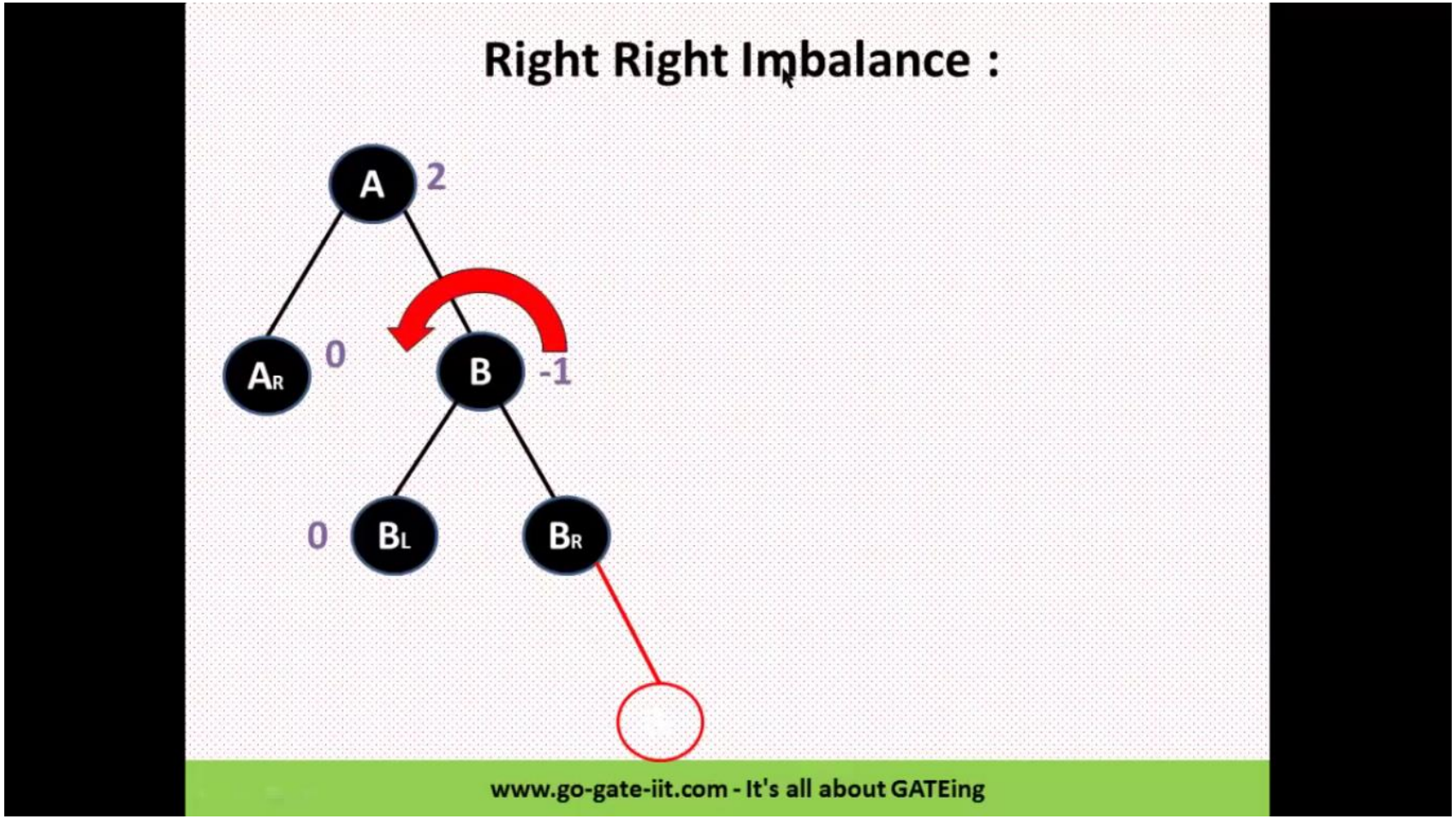

AVL Trees
AVL (Adelson, Velski & Landis) Trees.
Height of 2 child sub-tree of any node differ at most by 1
Self balancing Binary Search tree
1
2
3
4
5
+1
+1
0
0
Balanced
+2
+2
0
-1
0
Un-Balanced
Balance Factor(bf) : H(Left Sub tree) – H(Right Sub tree)
www.go-gate-iit.com - It's all about GATEing

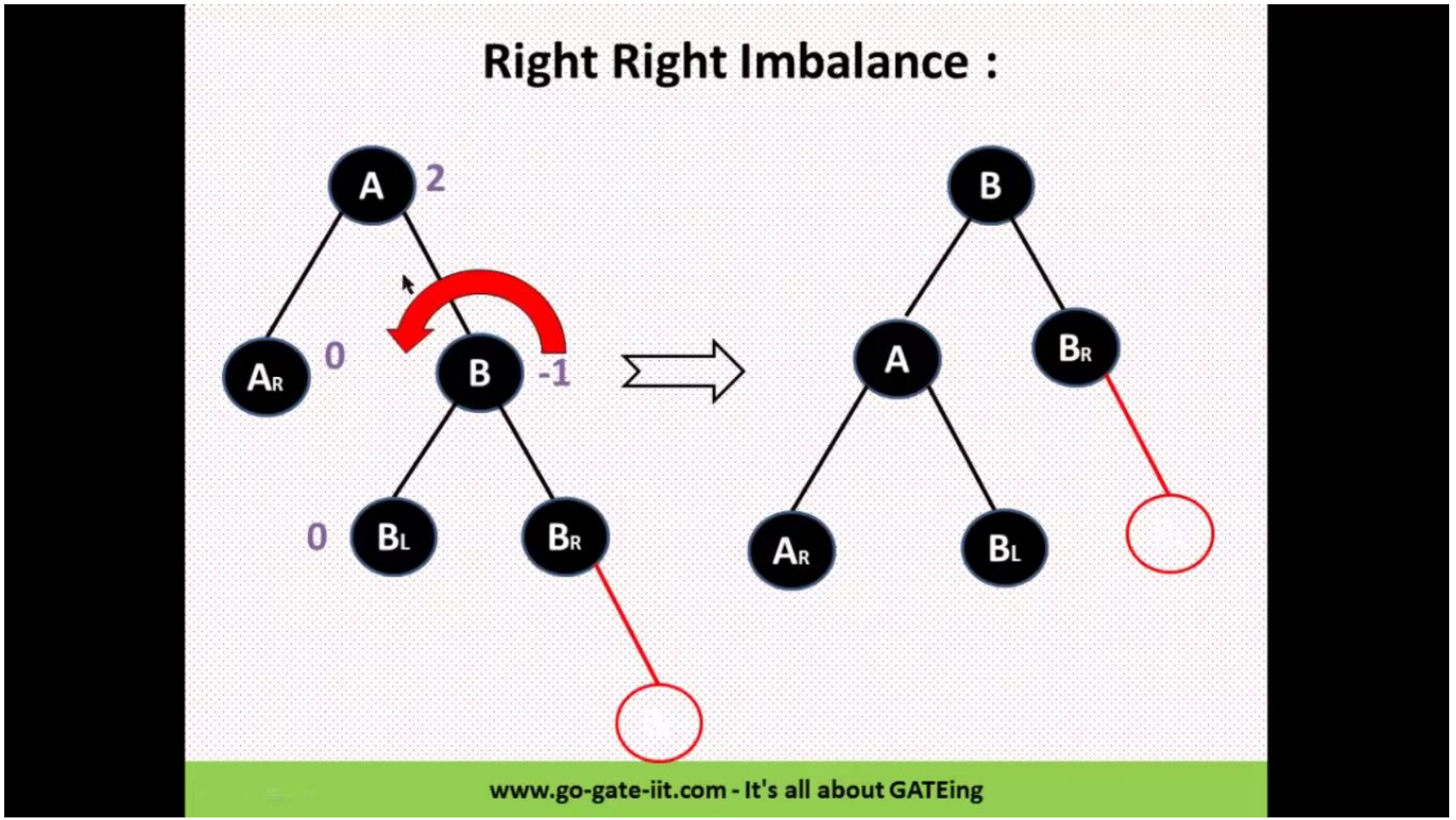
Types of imbalances

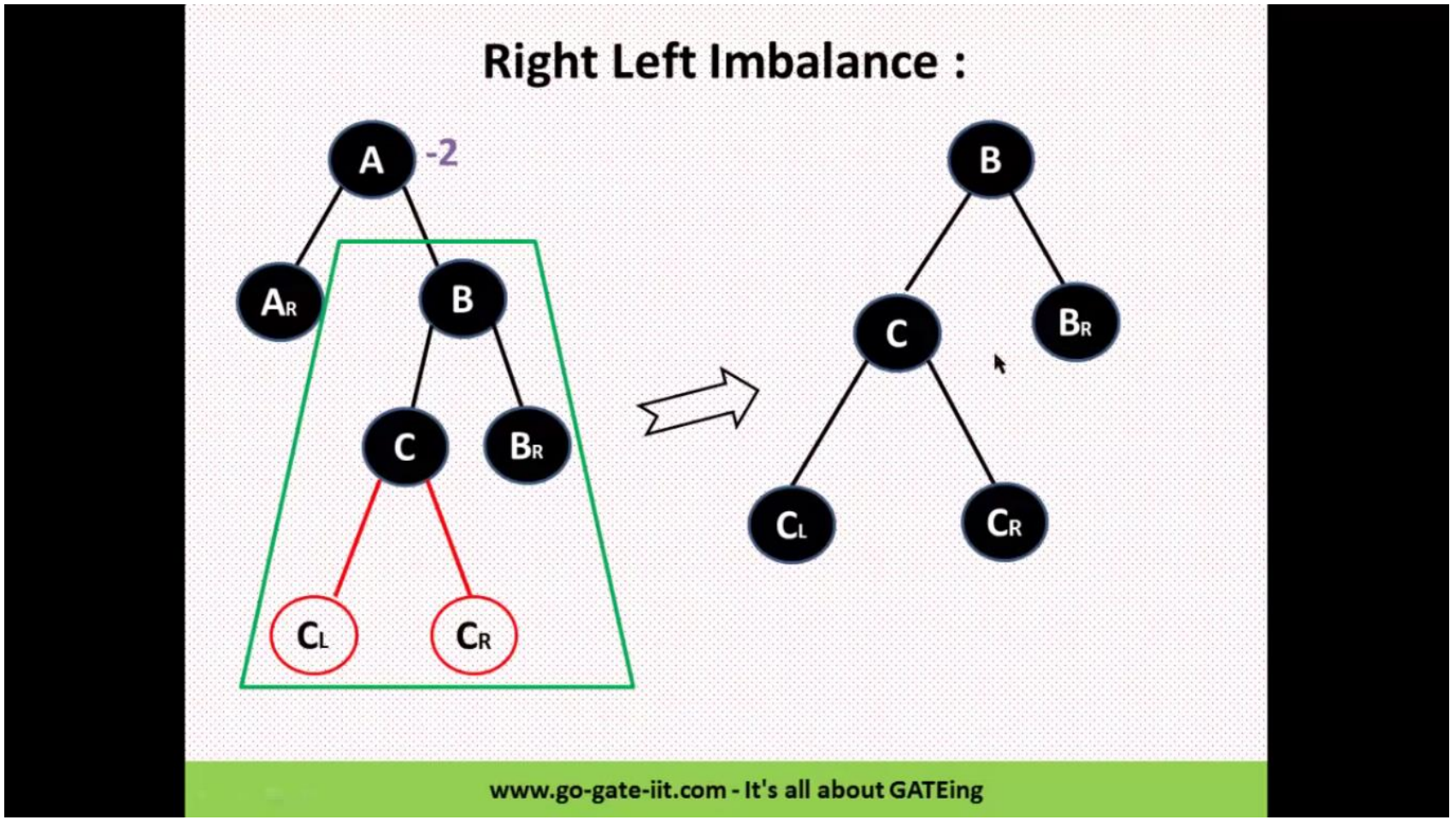
- **Violation cases at node k (deepest node)**
  1. An insertion into left subtree of left child of k
  2. An insertion into right subtree of right child of k
  3. An insertion into right subtree of left child of k
  4. An insertion into left subtree of right child of k
  - **Cases 1 and 2 equivalent**
    - Single rotation to rebalance
  - **Cases 3 and 4 equivalent**
    - Double rotation to rebalance

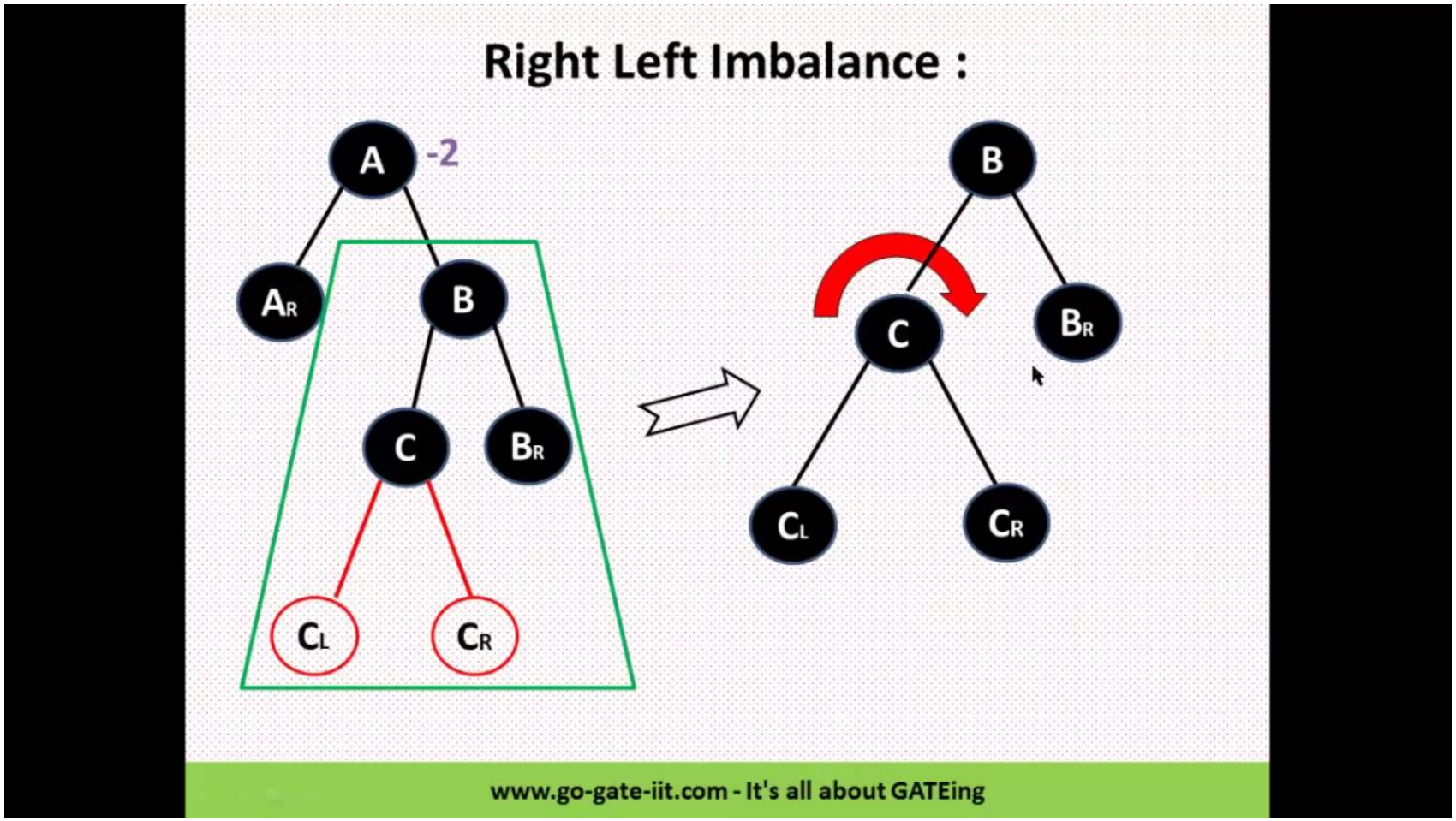

Left-Left Imbalance :
A
2
+1
B
0
AR
BL
1
BR
0
www.go-gate-iit.com - It's all about GATEing

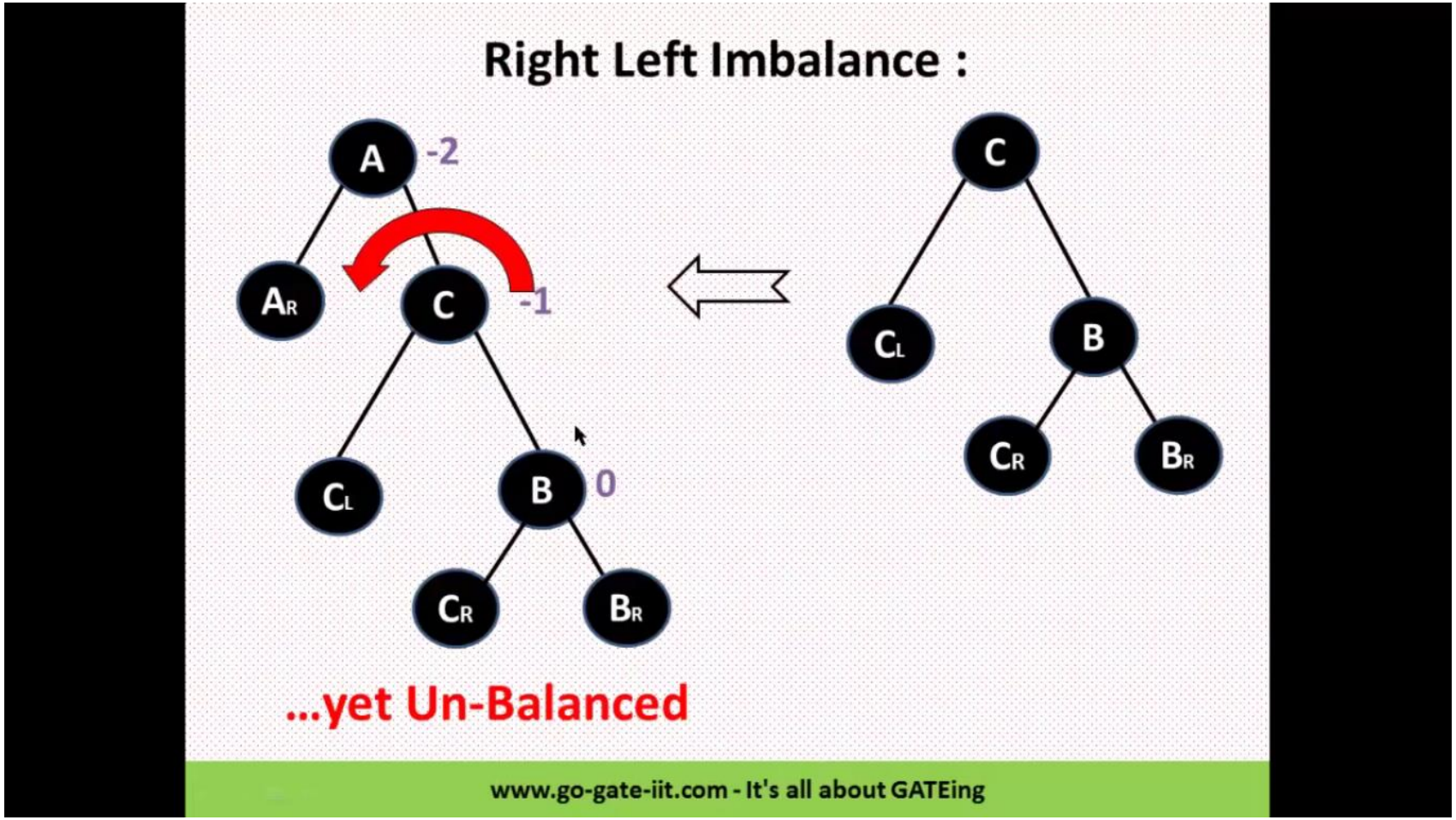

Left-Left Imbalance :
B
BL
A
BR
AR
www.go-gate-iit.com - It's all about GATEing

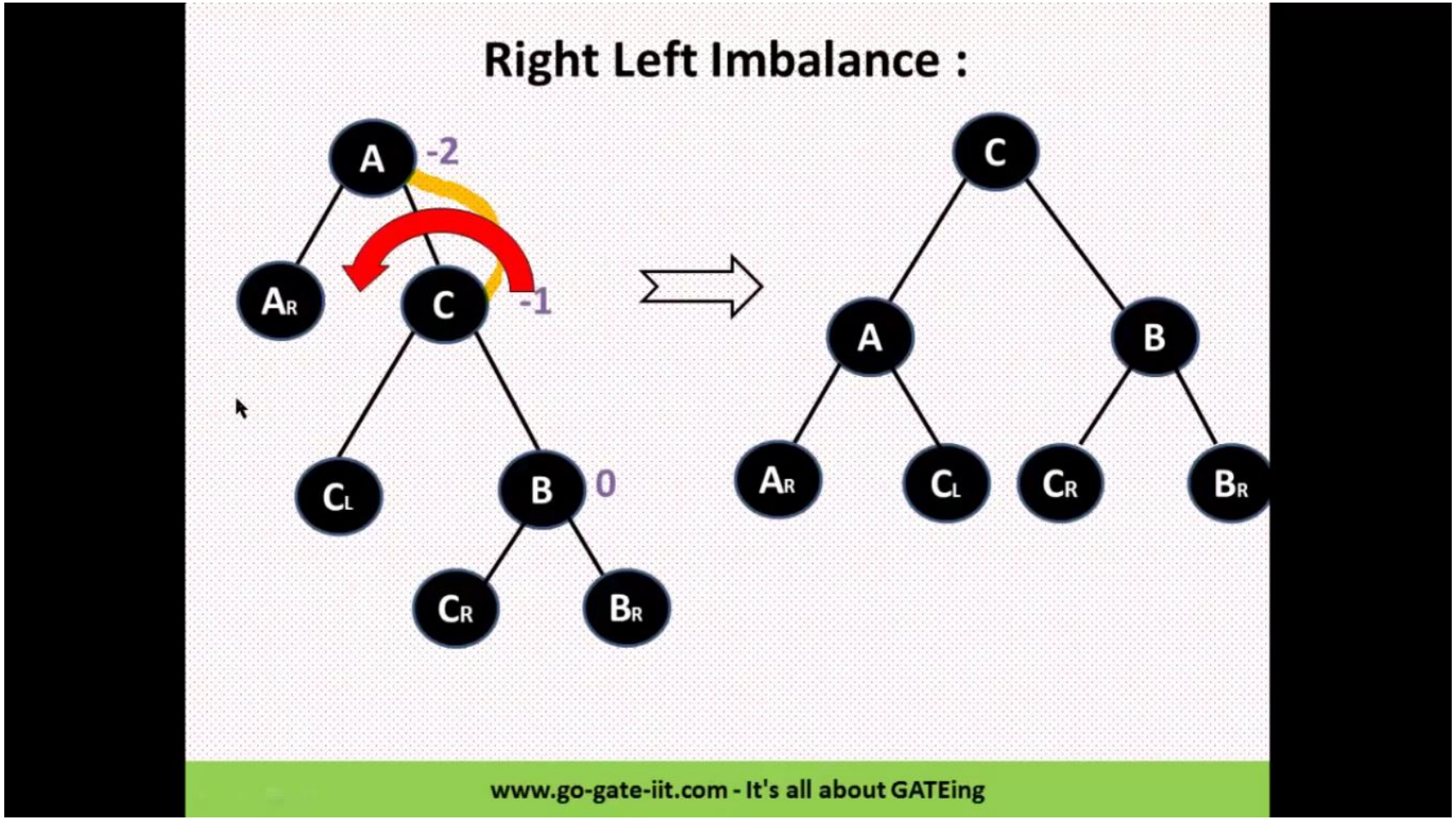

Right Right Imbalance :
A
2
A_R
0
B
-1
0
B_L
B_R
www.go-gate-iit.com - It's all about GATEing

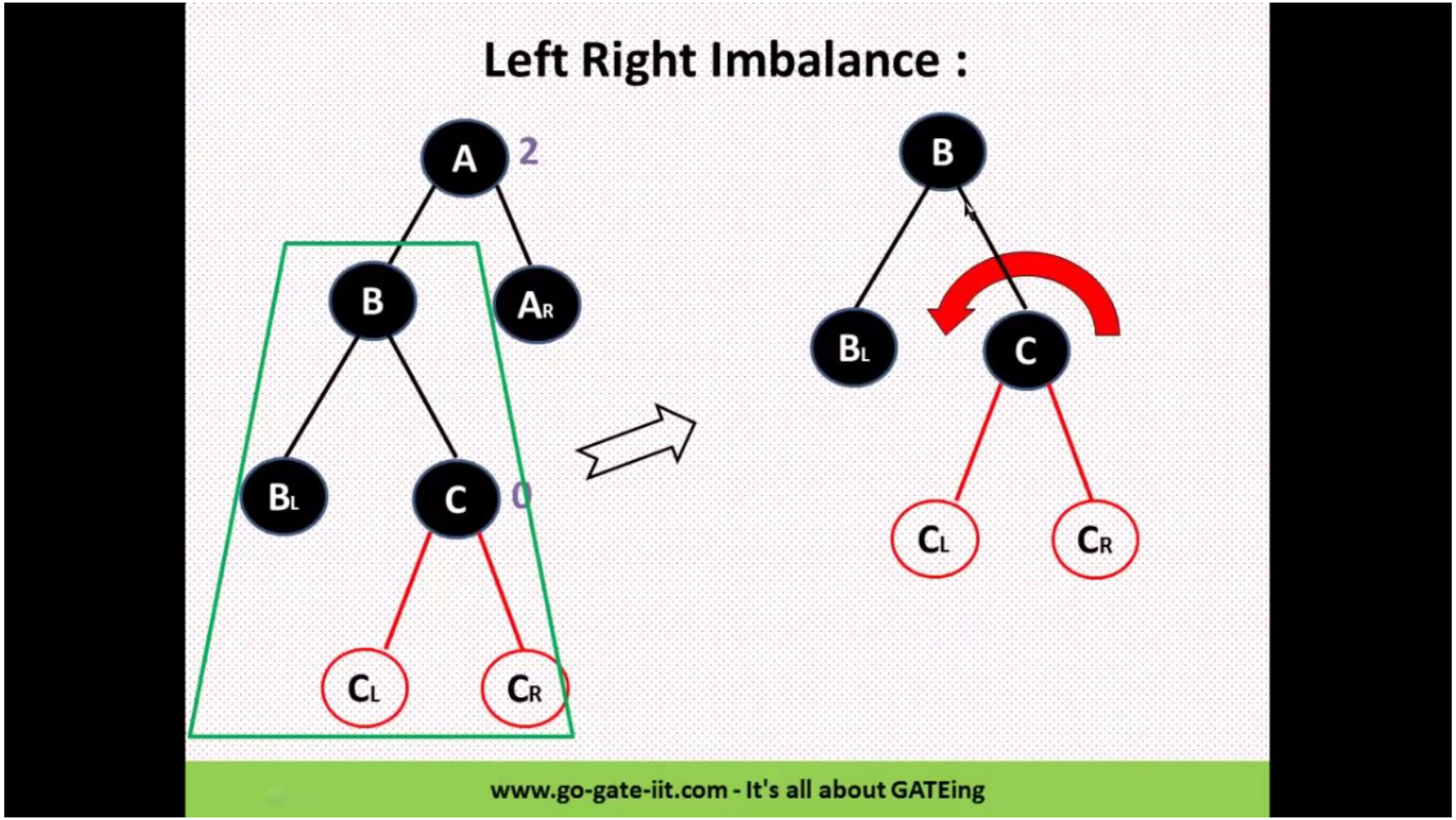

Right Right Imbalance :
A
2
A_R
0
B
-1
0
B_L
B_R
B
A
B_R
A_R
B_L
www.go-gate-iit.com - It's all about GATEing

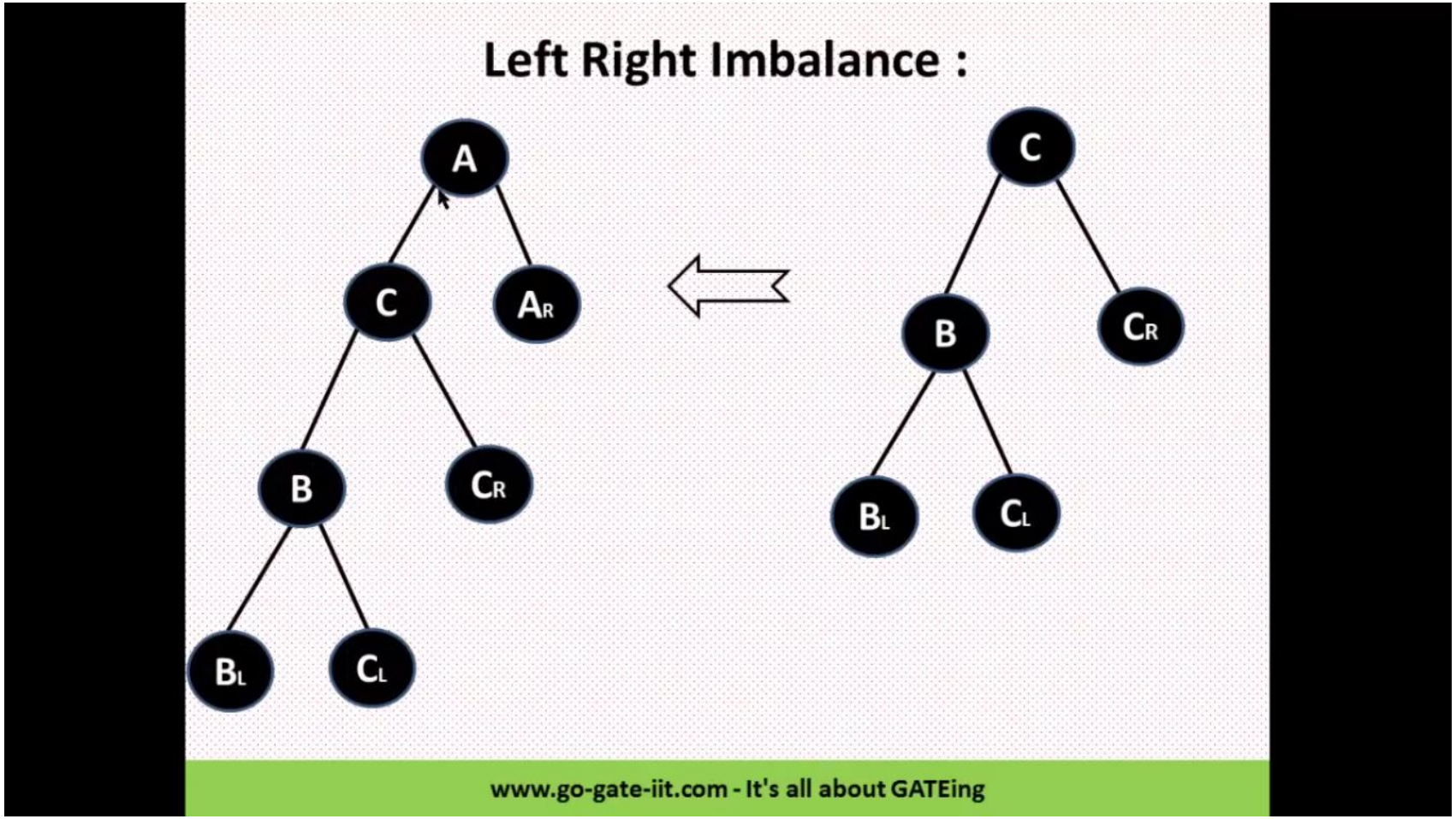

Right Left Imbalance :
A
-2
AR
B
C
BR
CL
CR
B
C
BR
CL
CR
www.go-gate-iit.com - It's all about GATEing

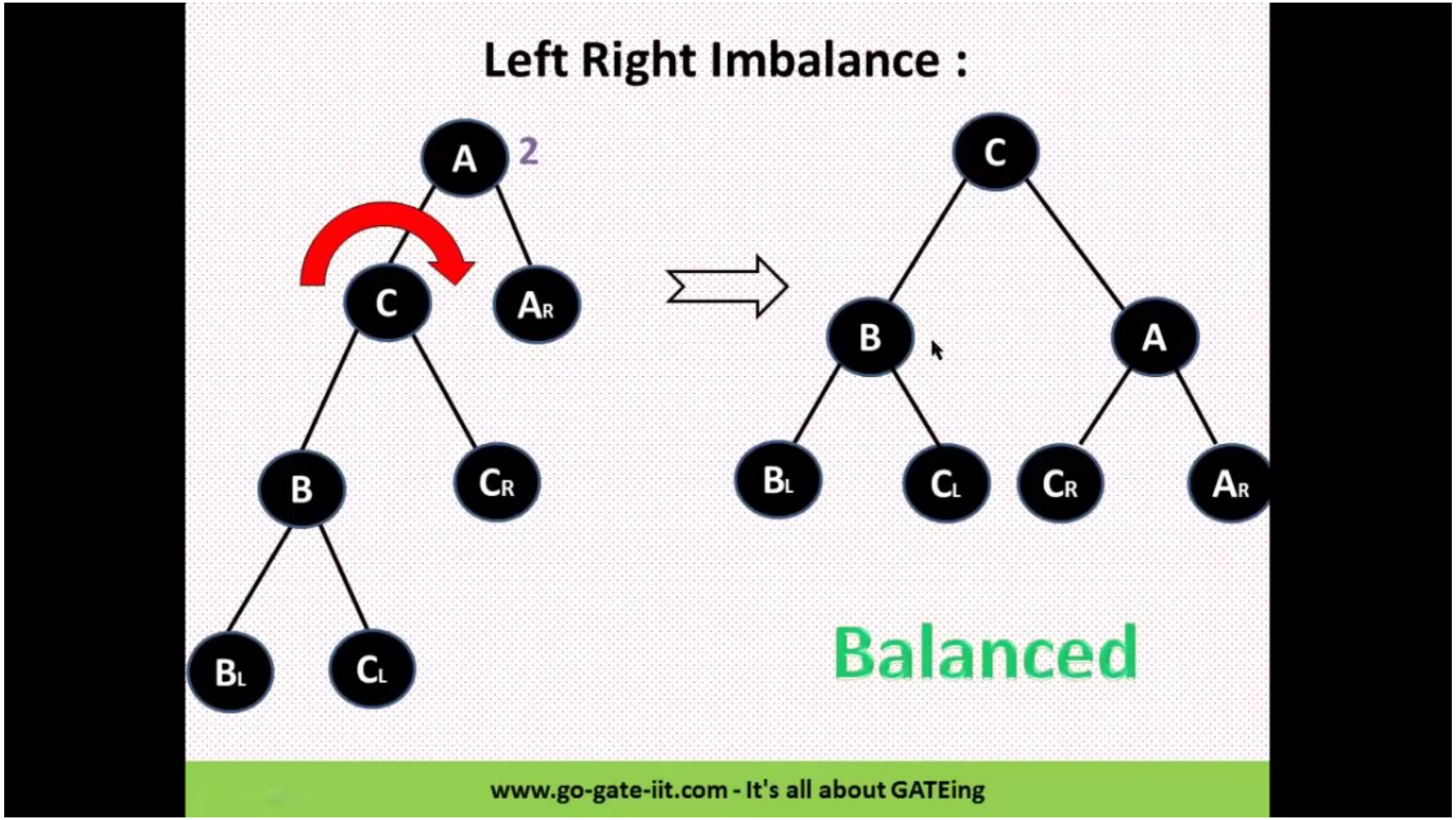

Right Left Imbalance :
A
-2
AR
B
C
BR
CL
CR
B
C
BR
CL
CR
www.go-gate-iit.com - It's all about GATEing


Right Left Imbalance :
A
-2
AR
C
-1
CL
B
0
CR
BR
C
CL
B
CR
BR
...yet Un-Balanced
www.go-gate-iit.com - It's all about GATEing


Right Left Imbalance :
A
-2
AR
C
-1
CL
B
0
CR
BR
C
A
B
AR
CL
CR
BR
www.go-gate-iit.com - It's all about GATEing


Left Right Imbalance :
A
2
B
AR
BL
C
0
CL
CR
B
BL
C
CL
CR
www.go-gate-iit.com - It's all about GATEing


Left Right Imbalance :
A
C
AR
B
CR
BL
CL
C
B
CR
BL
CL
www.go-gate-iit.com - It's all about GATEing


Left Right Imbalance :
A
2
C
AR
B
CR
BL
CL
C
B
A
BL
CL
CR
AR
Balanced
www.go-gate-iit.com - It's all about GATEing

Red-Black Trees

Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is balanced.
- Height is O(lg n), where n is the number of nodes.
- Operations take O(lg n) time in the worst case.

- Published by my PhD supervisor, Leo Guibas, and Bob Sedgewick.
- Easiest of the balanced search trees, so they are used in STL map operations…

Hysteresis : or the value of lazyness

- **Hysteresis,** n. [fr. Gr. to be behind, to lag.]
  a retardation of an effect when the forces acting upon a body are changed (as if from viscosity or internal friction); especially: a lagging in the values of resulting magnetization in a magnetic material (as iron) due to a changing magnetizing force

Red-black Tree

Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.

All other attributes of BSTs are inherited:

key, *left*, *right*, and *p*.

All empty trees (leaves) are colored black.

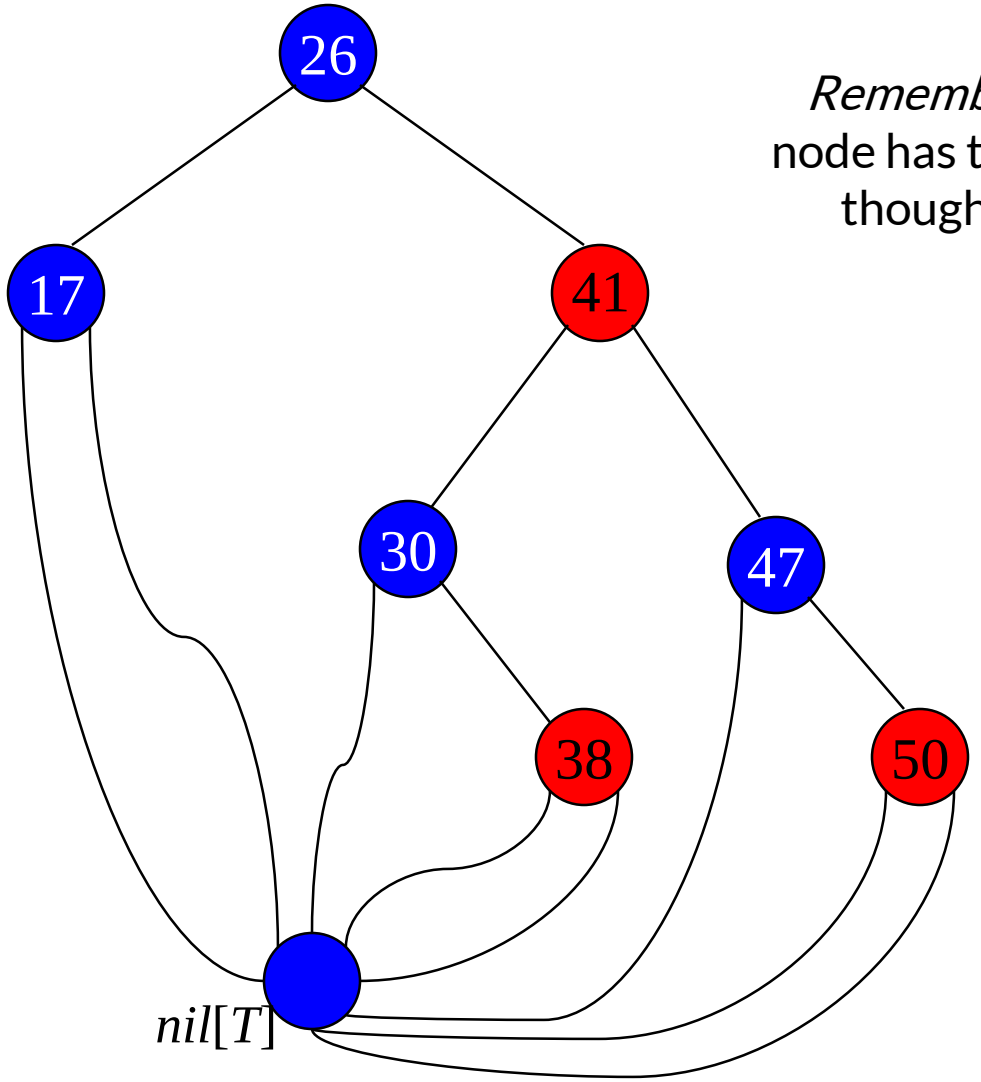
We use a single sentinel, nil, for all the leaves of red-black tree T, with color[nil] = black.

The root's parent is also nil[T].

Red-black Properties

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

Red-black Tree – Example



Remember: every internal node has two children, even though nil leaves are not usually shown.

Height of a Red-black Tree

Height of a node:

$h(x)$ = number of edges in a longest path to a leaf.

Black-height of a node x, $bh(x)$:

$bh(x)$ = number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.

Black-height of a red-black tree is the black-height of its root.

By Property 5, black height is well defined.

Height of a Red-black Tree

Example:

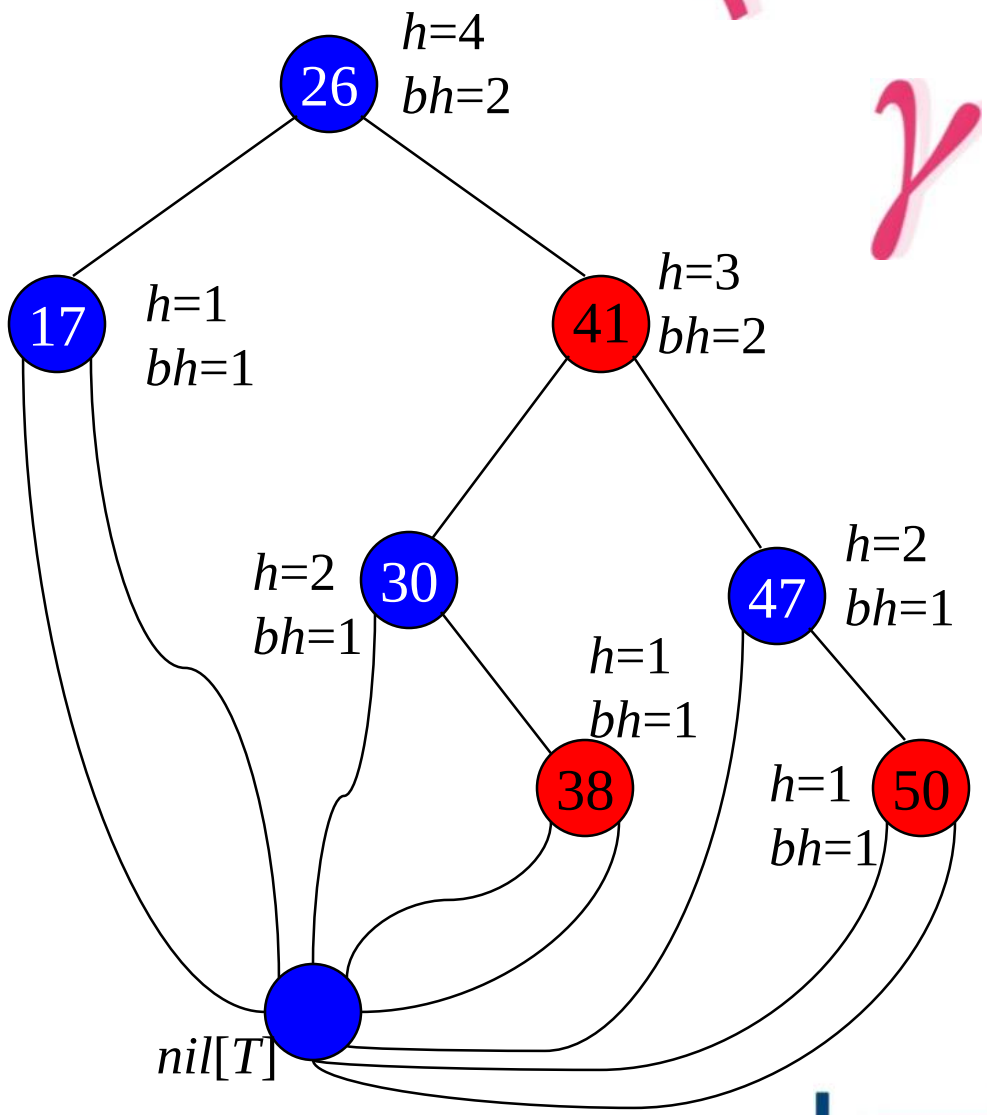
Height of a node:

$h(x)$ = # of edges in a longest path to a leaf.

Black-height of a node $bh(x)$ = # of black nodes on path from x to leaf, not counting x.

How are they related?

$bh(x) \le h(x) \le 2\, bh(x)$



Lemma “RB Height”

Consider a node x in an RB tree: The longest descending path from x to a leaf has length $h(x)$, which is at most twice the length of the shortest descending path from x to a leaf.

Proof:

black nodes on any path from x = $bh(x)$ (prop 5)

≤# nodes on shortest path from x, $s(x)$. (prop 1)

But, there are no consecutive red (prop 4),

and we end with black (prop 3), so $h(x) \leq 2\, bh(x)$.

Thus, $h(x) \leq 2\, s(x)$. **QED**

Bound on RB Tree Height

Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.

Proof: By induction on height of x.

Base Case: Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$.
Subtree has $2^0-1 = 0$ nodes. √

Induction Step: Height $h(x) = h > 0$ and $bh(x) = b$.

Each child of x has height h - 1 and
black-height either b (child is red) or b - 1 (child is black).

By ind. hyp., each child has $\geq 2^{\text{bh}(x)-1}-1$ internal nodes.

Subtree rooted at x has $\geq 2\,(2^{\text{bh}(x)-1} - 1)+1$
$= 2^{\text{bh}(x)} - 1$ internal nodes. (The +1 is for x itself.)

Bound on RB Tree Height

Lemma: The subtree rooted at any node x has
$\geq 2^{bh(x)} - 1$ internal nodes.

Lemma 13.1: A red-black tree with n internal nodes has height at most $2 \lg(n+1)$.

Proof:

By the above lemma, $n \geq 2^{bh} - 1$,

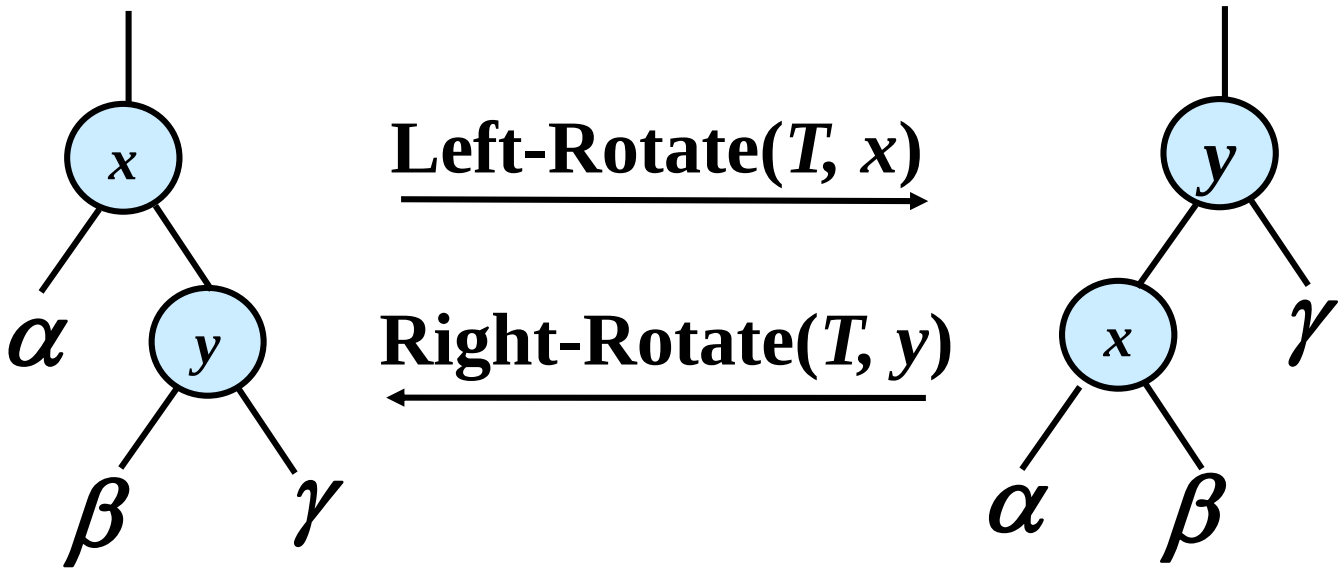
and since $bh \geq h/2$, we have $n \geq 2^{h/2} - 1$.

$\Rightarrow \; h \leq 2 \lg(n+1)$.

Operations on RB Trees

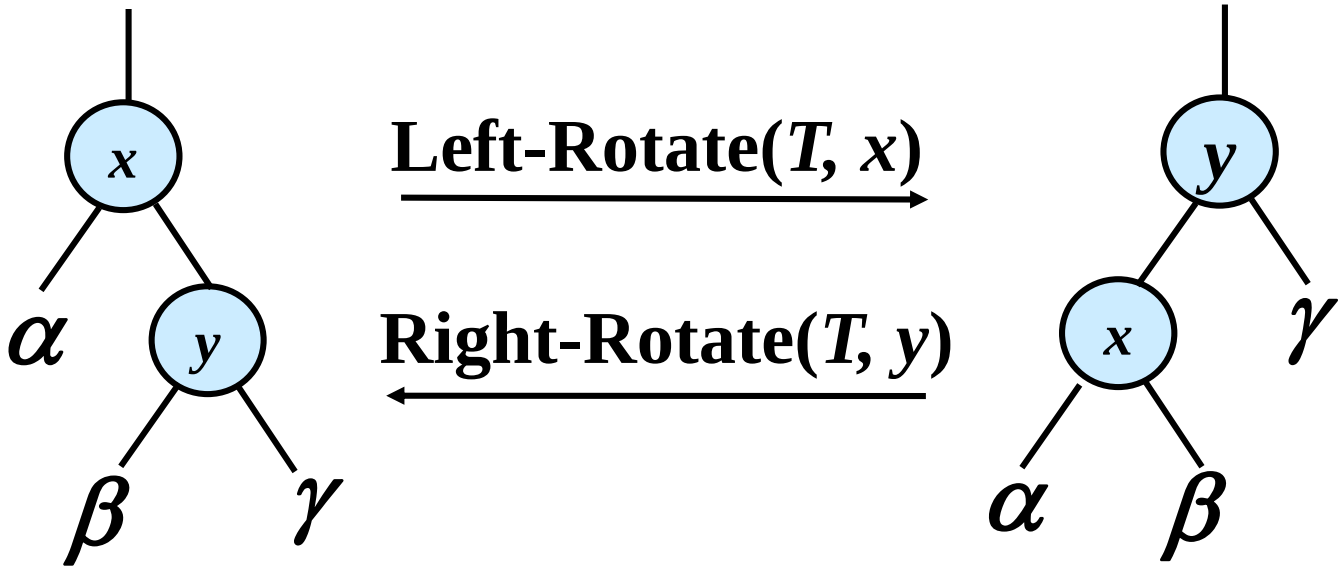
- All operations can be performed in $O(\lg n)$ time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in BSTs.
- Insertion and Deletion are not straightforward. Why?

Rotations



Rotations

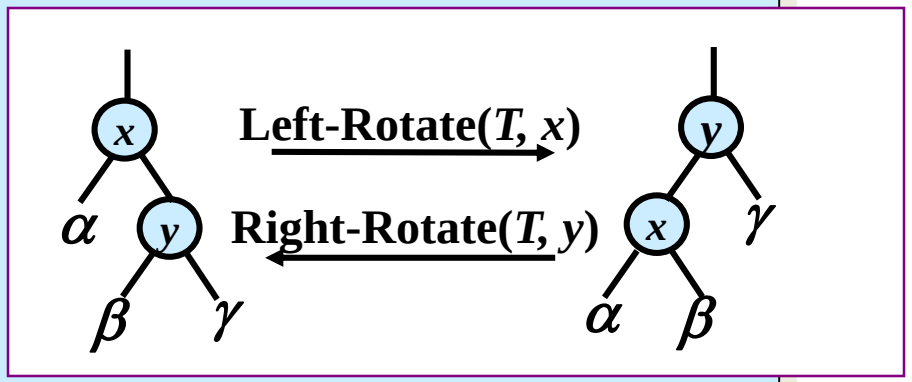
- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation – Pseudo-code

Left-Rotate (*T*, *x*)

```
1.   y ← right[x]   // Set y.
2.   right[x] ← left[y]   //Turn y's left subtree into x's right subtree.
3.   if left[y] ≠ nil[T ]
4.        then p[left[y]] ← x
5.   p[y] ← p[x]          // Link x's parent to y.
6.   if p[x] = nil[T ]
7.        then root[T ] ← y
8.        else if x = left[p[x]]
9.                then left[p[x]] ← y
10.               else right[p[x]] ← y
11.  left[y] ← x          // Put x on y's left.
12.  p[x] ← y
```



Rotation

- The pseudo-code for Left-Rotate assumes that

  right[x] $\neq$ nil[T], and

  root's parent is nil[T].

- Left Rotation on x, makes x the left child of y, and the left subtree of y into the right subtree of x.

- Pseudocode for Right-Rotate is symmetric: exchange left and right everywhere.

- Time: O(1) for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Reminder: Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Insertion in RB Trees

Insertion must preserve all red-black properties.

Should an inserted node be colored Red? Black?

Basic steps:

Use Tree-Insert from BST (slightly modified) to insert a node x into T.

Procedure **RB-Insert(x)**.

Color the node x red.

Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.

Procedure **RB-Insert-Fixup**.

Insertion

RB-Insert(T, z)

```
1.   y ← nil[T]
2.   x ← root[T]
3.   while x ≠ nil[T]
4.       do y ← x
5.          if key[z] < key[x]
6.             then x ← left[x]
7.             else x ← right[x]
8.   p[z] ← y
9.   if y = nil[T]
10.      then root[T] ← z
11.      else if key[z] < key[y]
12.         then  left[y] ← z
13.         else right[y] ← z
```

RB-Insert(T, z) Contd.

```
14.  left[z] ← nil[T]
15.  right[z] ← nil[T]
16.  color[z] ← RED
17.  RB-Insert-Fixup (T, z)
```

How does it differ from the Tree-Insert procedure of BSTs?

Which of the RB properties might be violated?

Fix the violations by calling RB-Insert-Fixup.

Insertion – Fixup

Insertion – Fixup

Problem: we may have one pair of consecutive reds where we did the insertion.

Solution: rotate it up the tree and away…
Three cases have to be handled…

http://www2.latech.edu/~apaun/RBTreeDemo.htm

Insertion – Fixup

RB-Insert-Fixup (*T*, *z*)

```
1.    while color[p[z]] = RED
2.        do if p[z] = left[p[p[z]]]
3.            then y ← right[p[p[z]]]
4.                 if color[y] = RED
5.                    then color[p[z]] ← BLACK  // Case 1
6.                         color[y] ← BLACK      // Case 1
7.                         color[p[p[z]]] ← RED  // Case 1
8.                         z ← p[p[z]]           // Case 1
```

Insertion – Fixup

RB-Insert-Fixup(*T*, *z*) (Contd.)

```
9.          else if z = right[p[z]]  // color[y] ≠ RED
10.             then z ← p[z]                          // Case 2
11.                  LEFT-ROTATE(T, z)                 // Case 2
12.             color[p[z]] ← BLACK                    // Case 3
13.             color[p[p[z]]] ← RED                   // Case 3
14.             RIGHT-ROTATE(T, p[p[z]])               // Case 3
15.     else (if p[z] = right[p[p[z]]])(same as 10-14
16.             with "right" and "left" exchanged)
17. color[root[T ]] ← BLACK
```

Correctness

Loop invariant:

At the start of each iteration of the **while** loop,

z is red.

If $p[z]$ is the root, then $p[z]$ is black.

There is at most one red-black violation:

Property 2: z is a red root, or

Property 4: z and $p[z]$ are both red.

Correctness – Contd.

Initialization: √

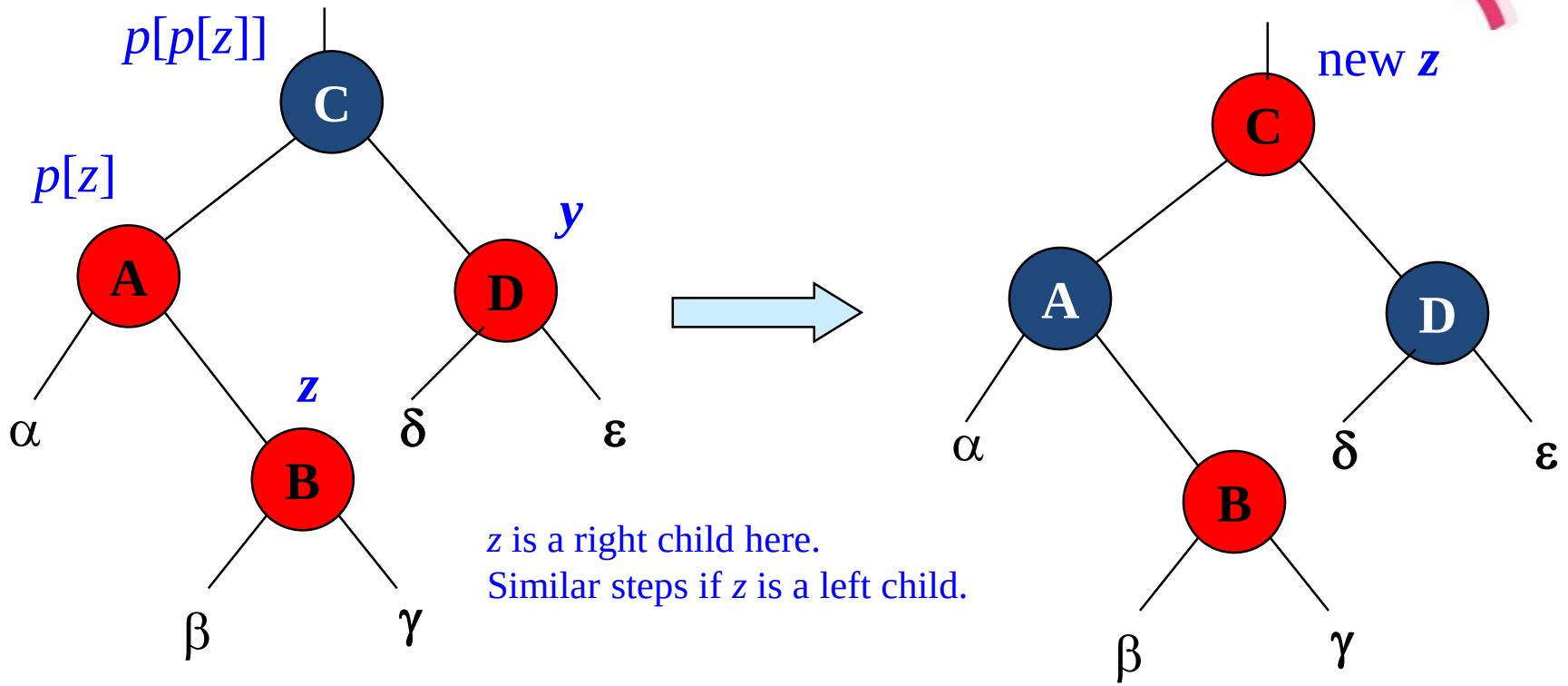
Termination: The loop terminates only if $p[z]$ is black. Hence, property 4 is OK. The last line ensures property 2 always holds.

Maintenance: We drop out when z is the root (since then $p[z]$ is sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.

There are 6 cases, 3 of which are symmetric to the other 3. We consider cases in which $p[z]$ is a left child.

Let y be z's uncle ($p[z]$'s sibling).

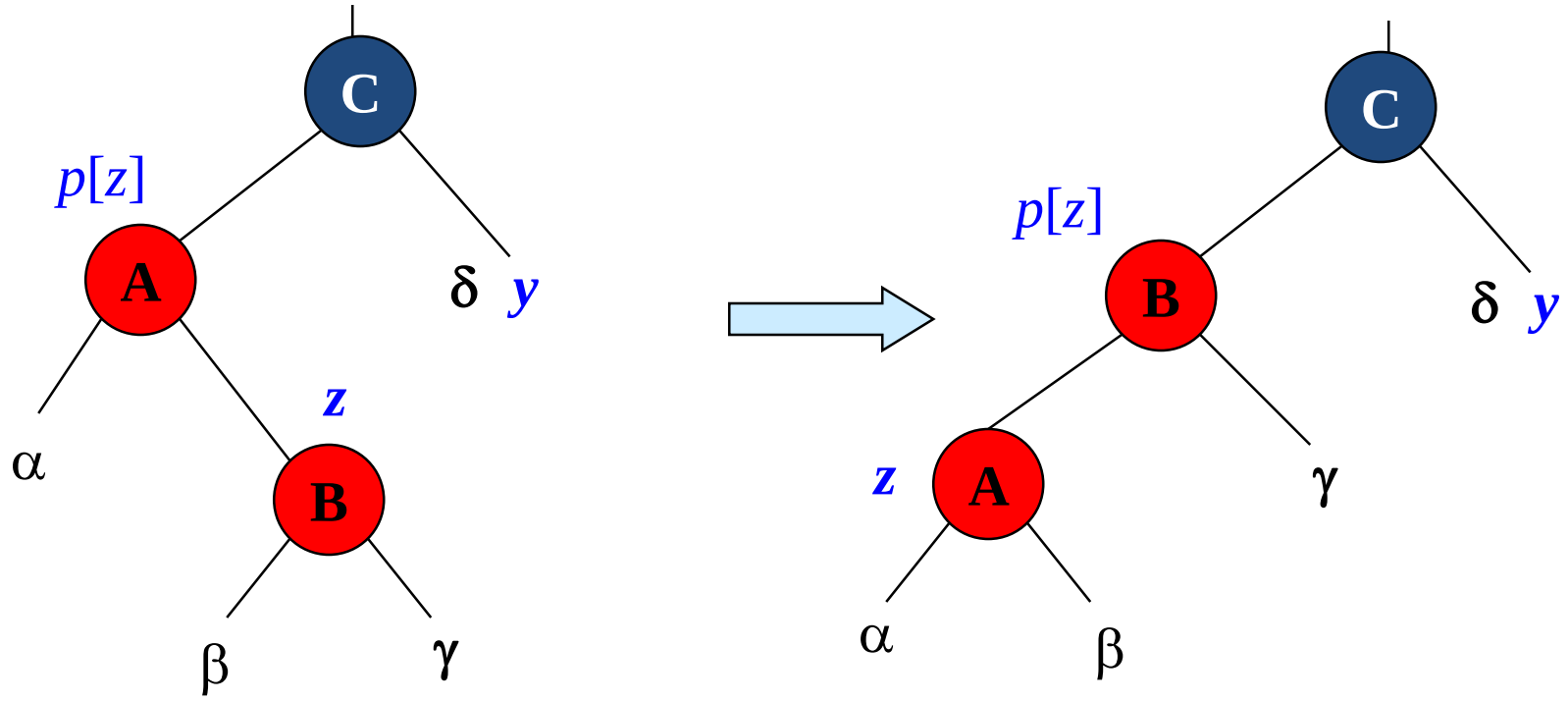
Case 1 – uncle y is red



z is a right child here.
Similar steps if *z* is a left child.

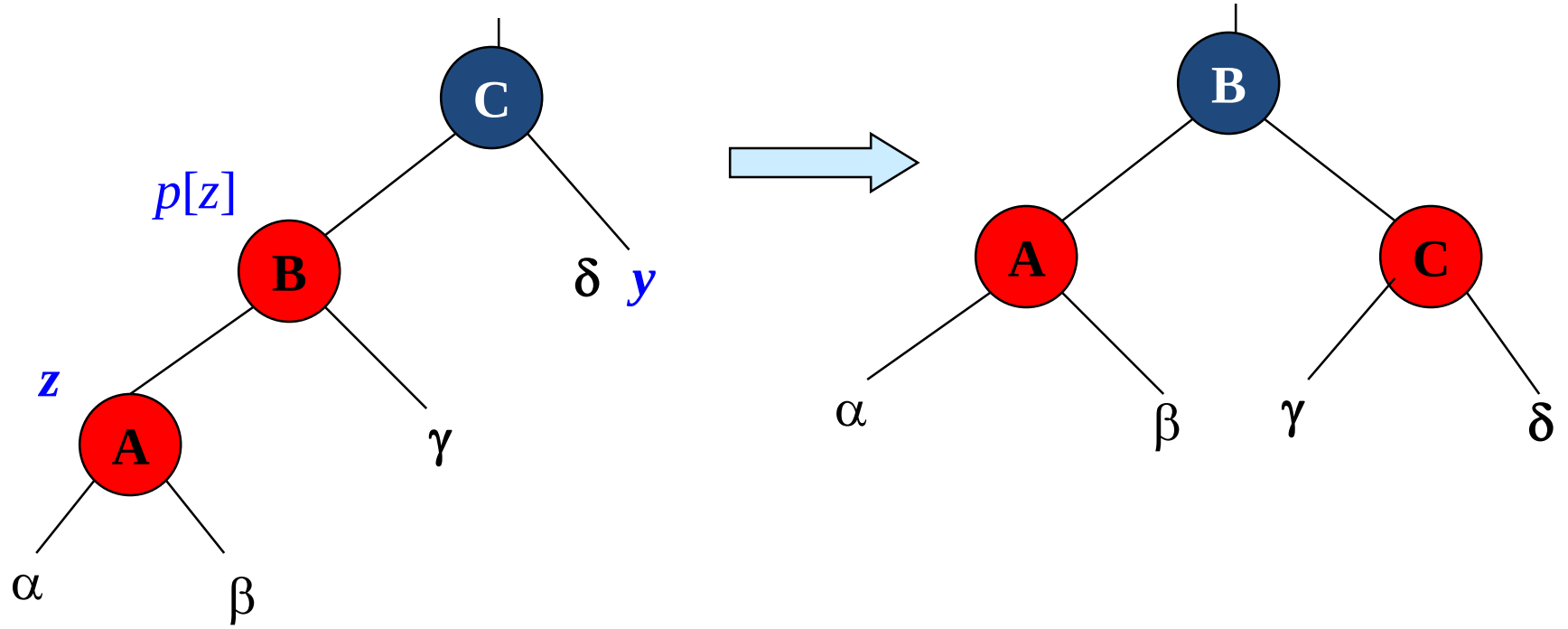
- *p*[*p*[*z*]] (*z*'s grandparent) must be black, since *z* and *p*[*z*] are both red and there are no other violations of property 4.
- Make *p*[*z*] and *y* black ⇒ now *z* and *p*[*z*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*z*]] red ⇒ restores property 5.
- The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).

Case 2 – y is black, z is a right child



- Left rotate around p[z], p[z] and z switch roles ⇒ now z is a left child, and both z and p[z] are red.
- Takes us immediately to case 3.

Case 3 – y is black, z is a left child



- Make p[z] black and p[p[z]] red.
- Then right rotate on p[p[z]]. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- p[z] is now black ⇒ no more iterations.

Algorithm Analysis

$O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.

Within RB-Insert-Fixup:

- Each iteration takes $O(1)$ time.
- Each iteration but the last moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
- Thus, insertion in a red-black tree takes $O(\lg n)$ time.
- Note: there are at most 2 rotations overall.

Deletion

Deletion, like insertion, should preserve all the RB properties.

The properties that may be violated depends on the color of the deleted node.

Red – OK. Why?

Black?

Steps:

Do regular BST deletion.

Fix any violations of RB properties that may result.

Deletion

RB-Delete(*T*, *z*)

```
1.   if left[z] = nil[T] or right[z] = nil[T]
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z)
4.   if left[y] = nil[T ]
5.       then x ← left[y]
6.       else x ← right[y]
7.   p[x] ← p[y]   // Do this, even if x is nil[T]
```

Deletion

RB-Delete (*T*, *z*) (Contd.)

```
8.  if p[y] = nil[T ]
9.      then root[T ] ← x
10.     else if y = left[p[y]]
11.             then left[p[y]] ← x
12.             else right[p[y]] ← x
13. if y = z
14.     then key[z] ← key[y]
15.     copy y's satellite data into z
16. if color[y] = BLACK
17.      then RB-Delete-Fixup(T, x)
18. return y
```

The node passed to the fixup routine is the lone child of the spliced up node, or the sentinel.

Thank You!