Implementation of dictionary part -2

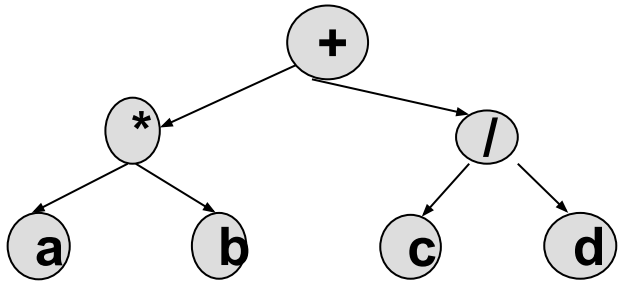
1. Through Binary search tree
2. Through AVL tree
3. Through B-Tree

Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

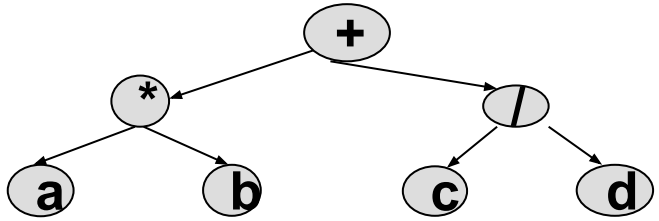
Example



A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...

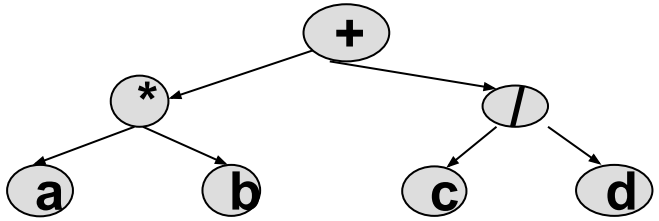


Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...

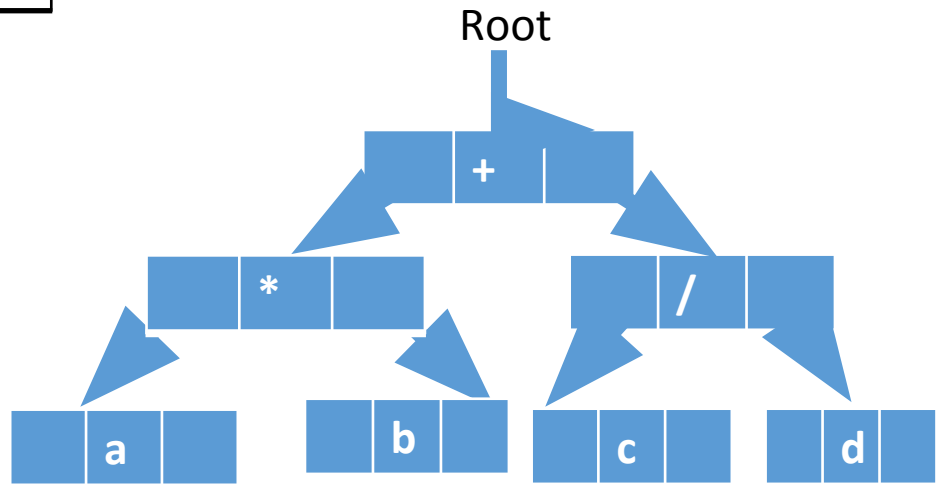
+	*	/	a	b	c	d

2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

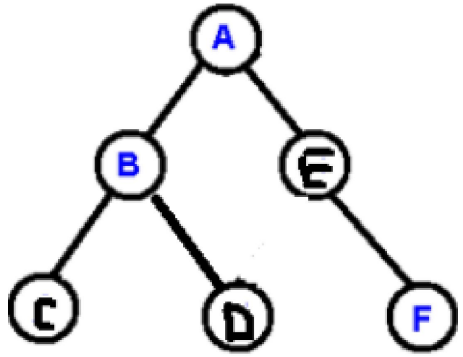
In this linked list representation, a node has the following structure...

null	40	null



Binary Tree Traversals:

- visiting order of nodes in a binary tree is called as Binary Tree Traversal. There are three types of binary tree traversals.
- 1.Pre - Order Traversal
- 2. In - Order Traversal
- 3.Post - Order Traversal
- Consider this binary tree for traversal:



Preorder Traversal(root-left-right) :

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child.

This pre-order traversal is applicable for every root node of all subtrees in the tree.

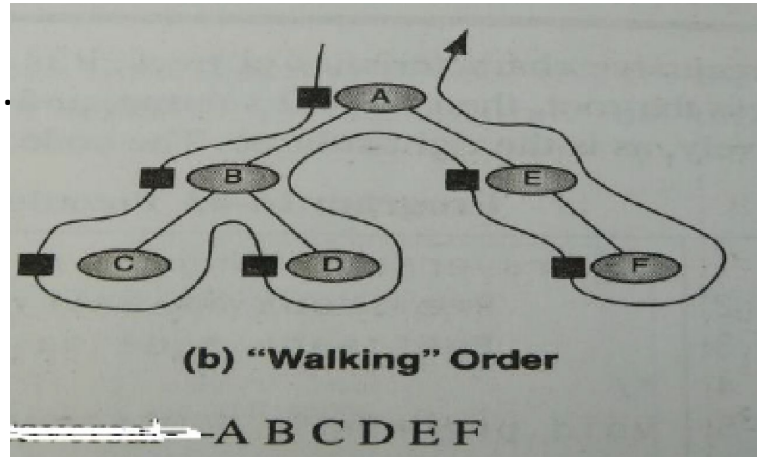
Steps:

Step 1: visit root node

Step 2: Traverse towards left node recursively

Step 3: Traverse towards right node recursively

Example:

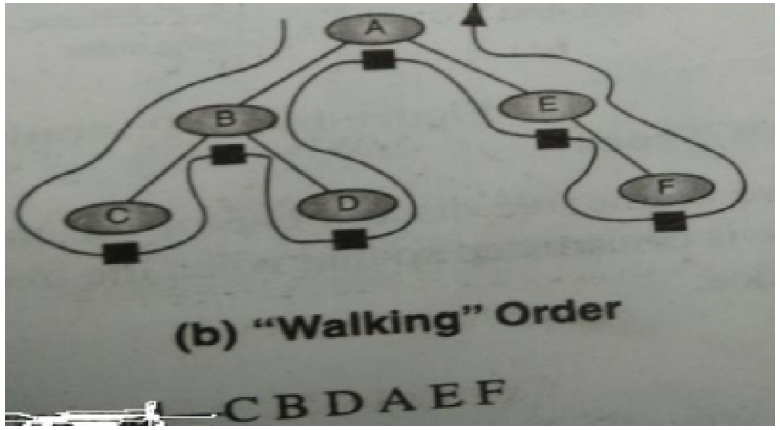


In this example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for C and D. So, we visit B's left child 'C' that is the left most child. Next we go for visiting right child of B that is D. With this we have completed root, left and right parts of node B and root, left parts of node A. So, we go for A's right child 'E' which is a root node for F. After visiting E , F gets visited.

• In-order traversal(Left-root-right)

- In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.
- STEPS:
- Step1: Traverse towards left node recursively
- Step 2: Visit root node
- Step 3: Traverse towards right node recursively

Example

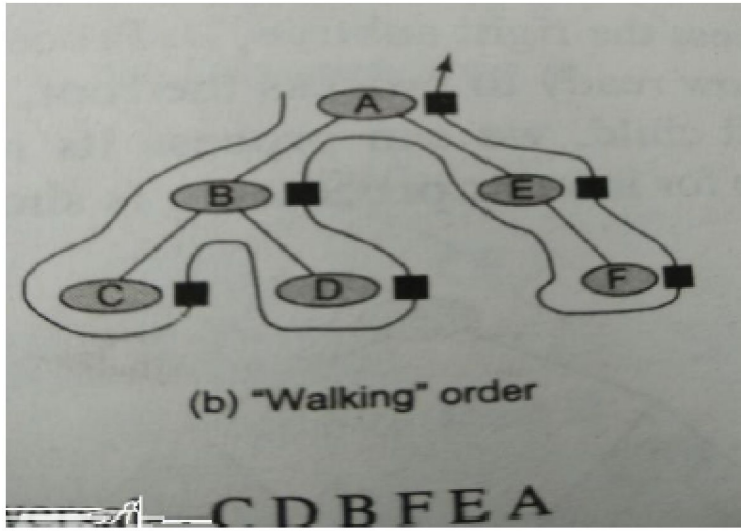


JAVA CODE

```
void   In-Order(Root)
{
 In-Order(Root->left);
 System.out.println(root);
In-Order(Root-right);
}
```

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child B is a root node for subtree with nodes B, C and D. So, we try to visit left child C of B and it is the left most child. So, first we visit C then go for it's root node B and later we visit B's right child D. With this we have completed the left part of node A. Then visit A and then go to its right child i.e E. Then we visit E and then F.

- **3. Post -Order Traversal (left - right - root)**
- In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited. STEPS: Step 1: Traverse towards left node recursively Step 2: Traverse towards right node recursively Step 3: Visit root node Example



JAVA CODE

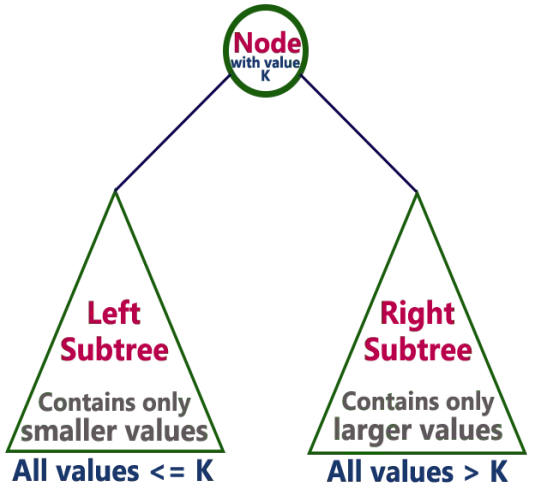
```
void   Post-Order(Root)
{
 Post-Order(Root->left);
 Post-Order(Root-right);
 System.out.println(root);
}
```

1.Binary Search Tree

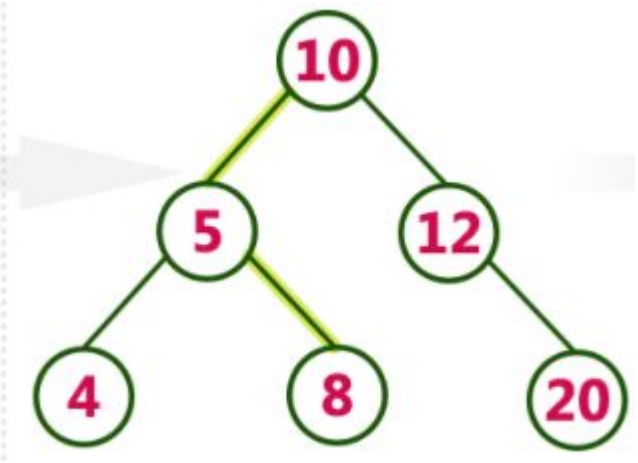
- A binary search tree is a binary tree that may be empty. A nonempty binary search tree satisfy the following properties:

1.Every element has a key , and no two elements have the same key; therefore all key elements are distinct.
2. The keys in the left subtree of the root are smaller than the keys in the root
3.The keys in the right subtree of the root are greater than the keys in the root
4. The left and right subtree of the root are also subtrees

- In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...



Example



- Operations on binary search trees:
- The following operations are performed on a binary earch tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Time Complexity of search and insertion: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Time Complexity of deletion: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

Searching a key

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Algorithm for search operation

- In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...
- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Illustration to search 6 in below tree:

1. Start from root.
2. Since the element to be searched is less than root ,then traverse for left, then towards right .

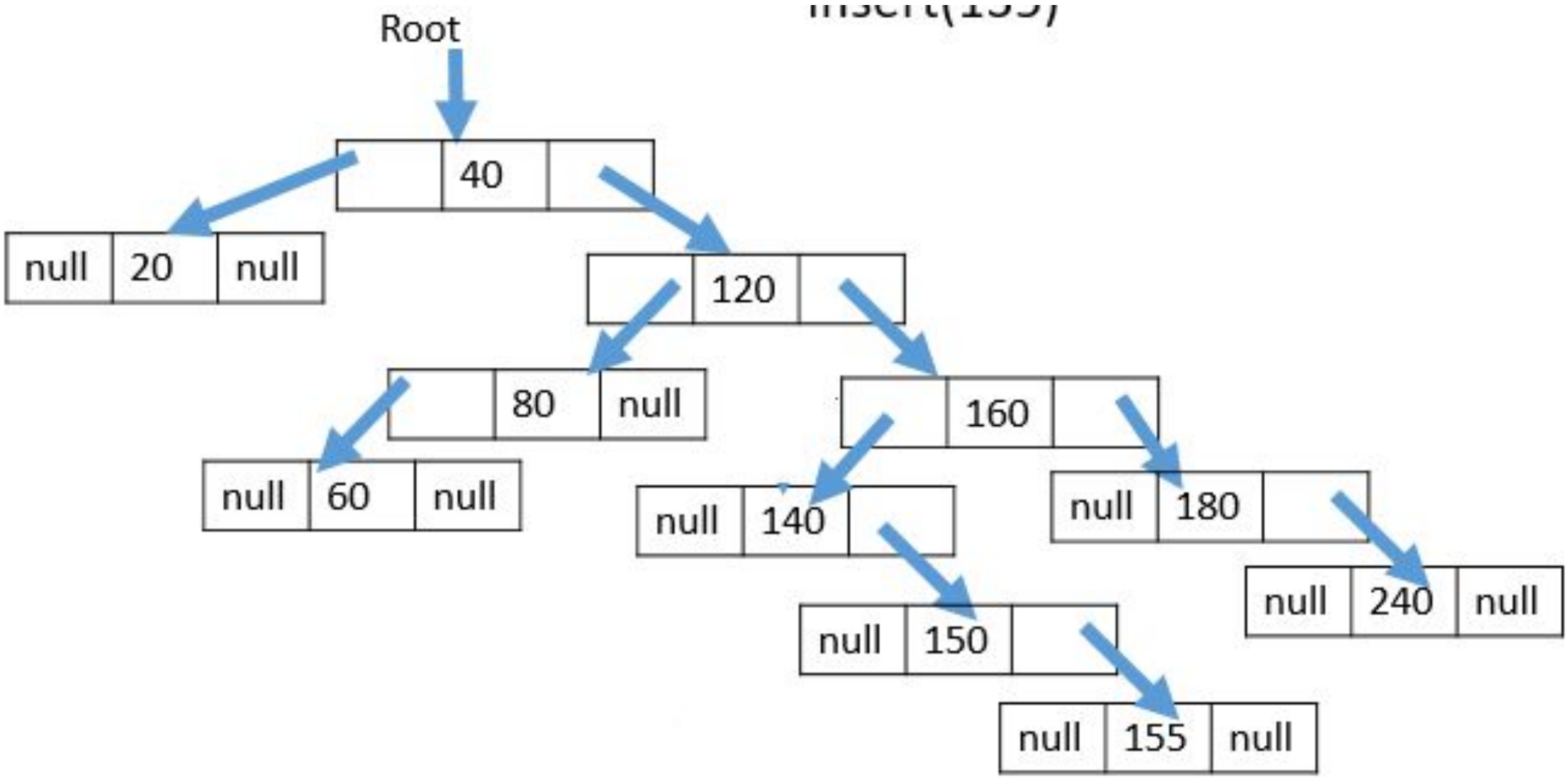
Insertion of a key into binary search tree

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

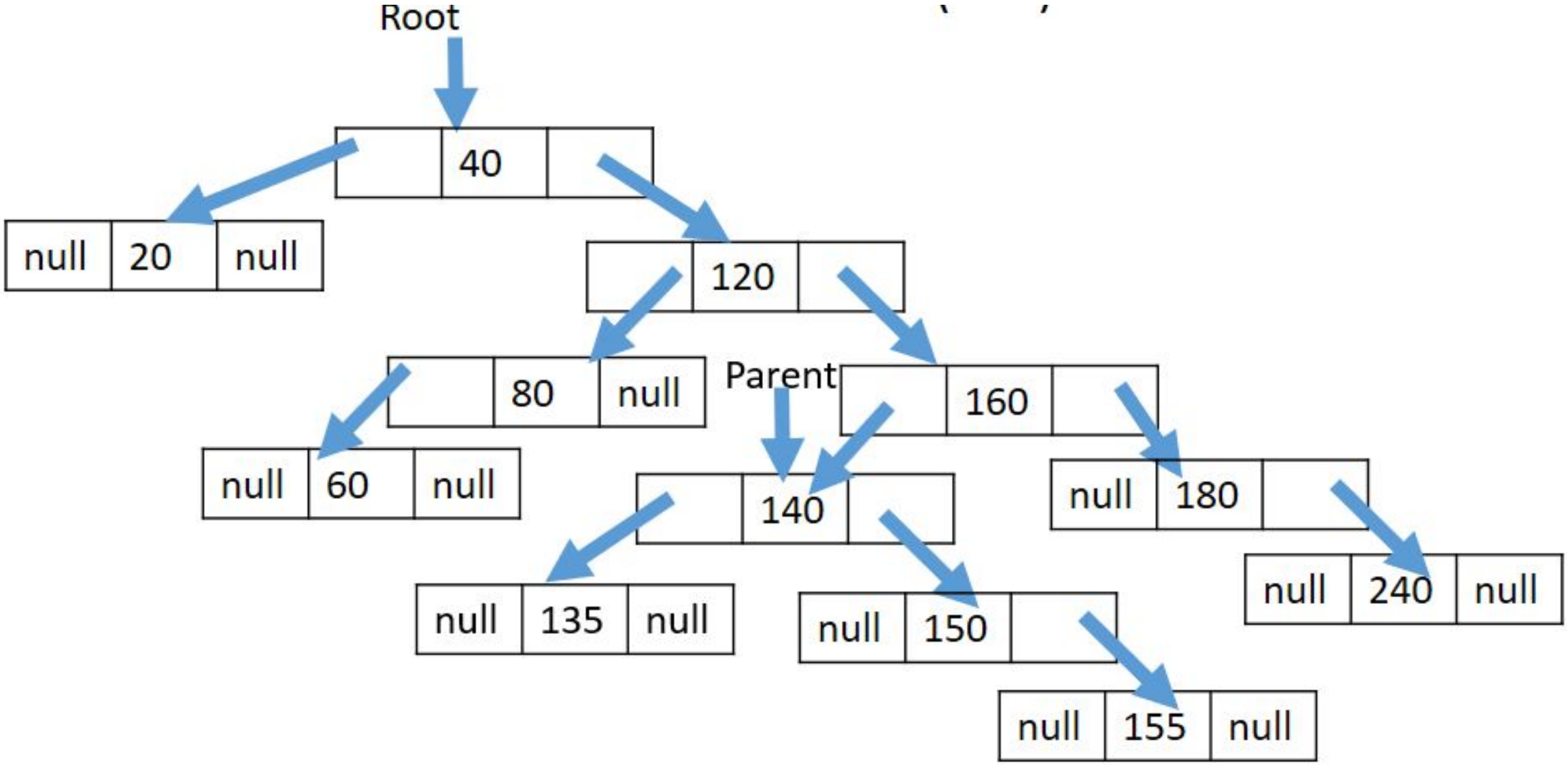
The insertion operation is performed as follows...

- **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set **root** to **newNode**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then insert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Example:Before insertion of 135 into binary search tree



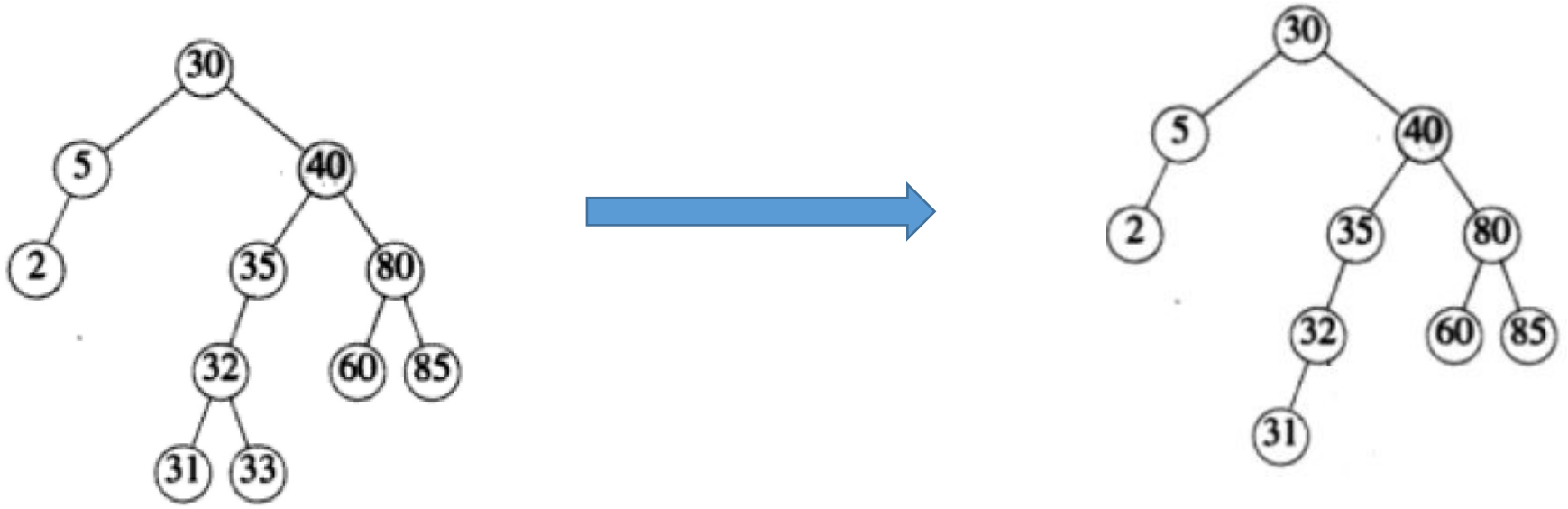
After insertion into BST



Deletion of node from Binary search tree

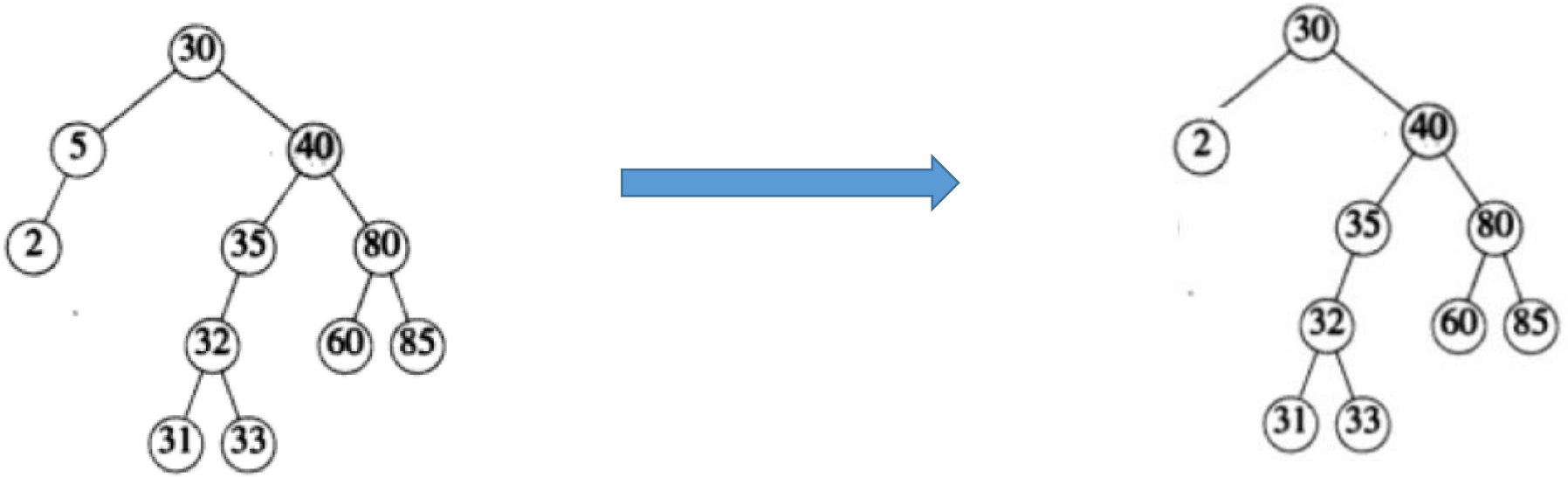
Case 1: Deleting a leaf node

- We use the following steps to delete a leaf node from BST...
- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function.
- Example: To remove 33 from thee below tree, we set right child of 32 to null, and the node is discarded.



Case 2: Deletion of a node with one child

- We use the following steps to delete a node with one child from BST...
- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node and terminate the function.
- Example: To delete 5 from the below tree , we change the left child field of its parent to point to the node containing 2.



Case 3: Deletion of a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using **search operation**

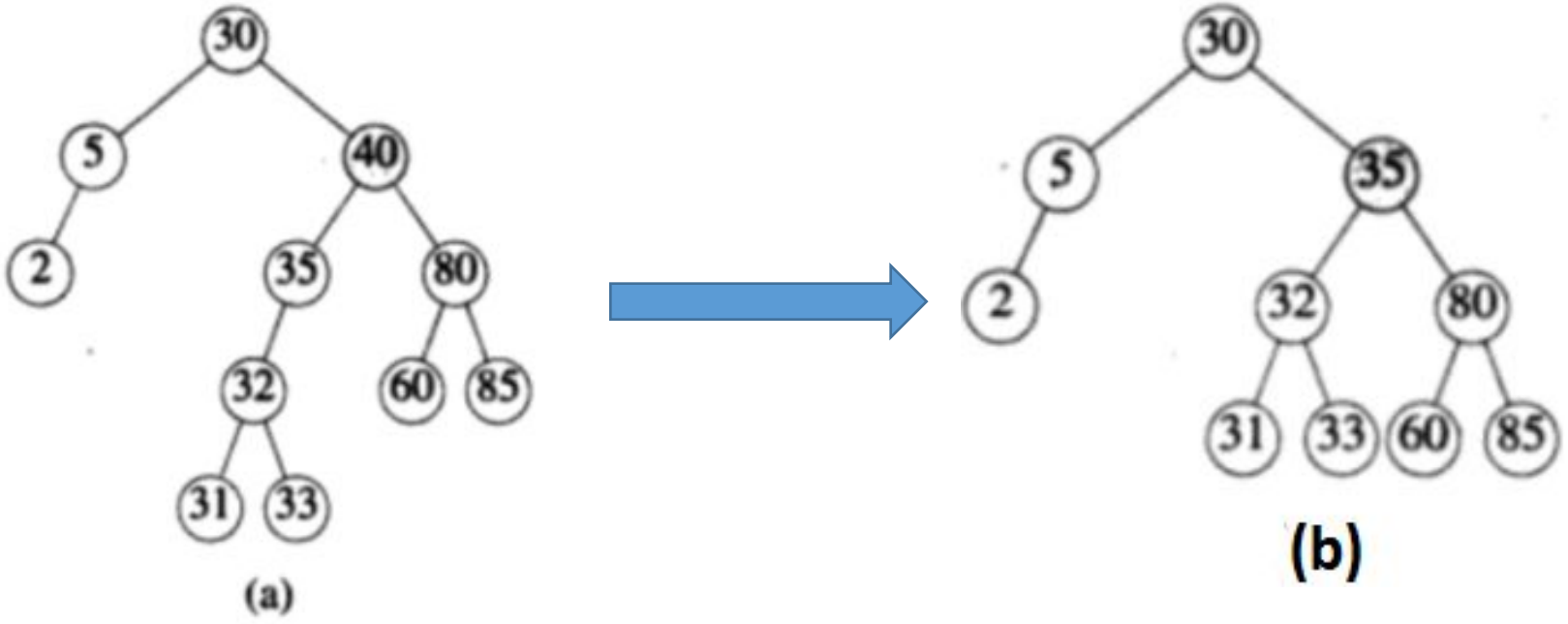
Step 2: If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.

Step 3: Then , we replace deleting element with an element which found in above step.

Step 4: Following the replacement, the replacing element is removed from the original node

Example: Suppose we wish to remove an element with key 40 from the tree of the following figure a. Either the largest element in its left subtree or smallest element in its right subtree replace this. If we opt for largest element in its left sub tree , we move the key element with key 35 to the node from which 40 was removed; in addition the

node from which 35 is moved is removed. The resultant tree appears as in the following figure b.



2. Balanced Binary Search Trees

- If the height of a binary tree is always O(log n), we can guarantee O(log n) performance for each search tree operation
- Trees with a worst-case height of O(log n) are called **balanced trees**
- An example of a balanced tree is **AVL** (Adelson-Velsky and Landis) tree

AVL Tree

Definition

- Binary tree.
- If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff
  1. T_L and T_R are AVL trees, and
  2. $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively

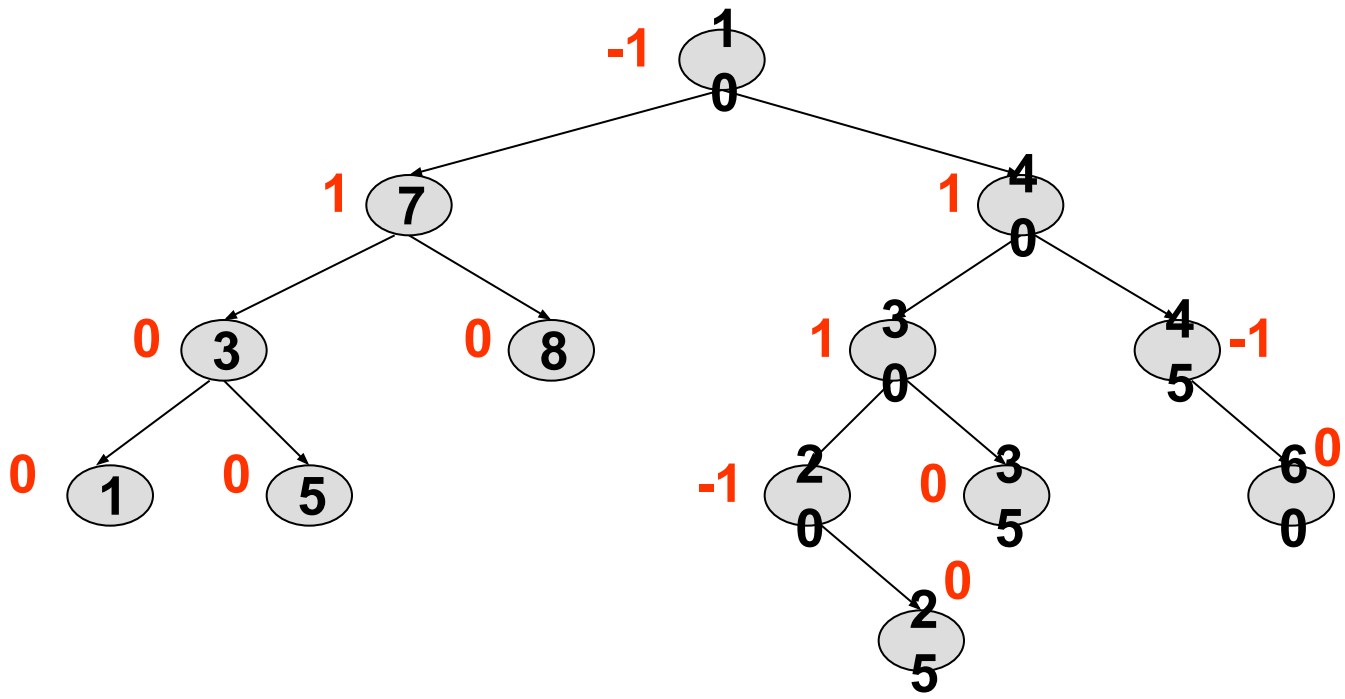
Properties of AVL Tree

1. The height of an AVL tree with n nodes is O(log n)
2. For every value of n, n ≥ 0, there exists an AVL tree
3. An n-node AVL search tree can be searched in O(height) = O(log n) time
4. A new node can be inserted into an n-node AVL search tree so that the result is an n+1 node AVL tree and insertion can be done in O(log n) time
5. A node can be deleted from an n-node AVL search tree, n>0, so that the result is an n-1 node AVL tree and deletion can be done in O(log n) time

Balance Factor

- AVL trees are normally represented using the linked representation
- To facilitate insertion and deletion, a **balance factor (bf)** is associated with each node
- The balance factor bf(x) of a node x is defined as
  height(x→leftChild) – height(x→rightChild)
- Balance factor of each node in an AVL tree must be –1, 0, or 1

AVL Tree with Balance Factors



Operations on AVL Trees

The following operations are performed on an AVL tree...

1.Search

2.Insertion

3.Deletion

Search Operation in AVL Tree:

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1:Find the place to insert the new element by following path from the root ,then insert the new node and set the balance factor of new node as zero(0).During this process , keep track of most recently visited node with Say “A” with balance factor as either 1 or -1.

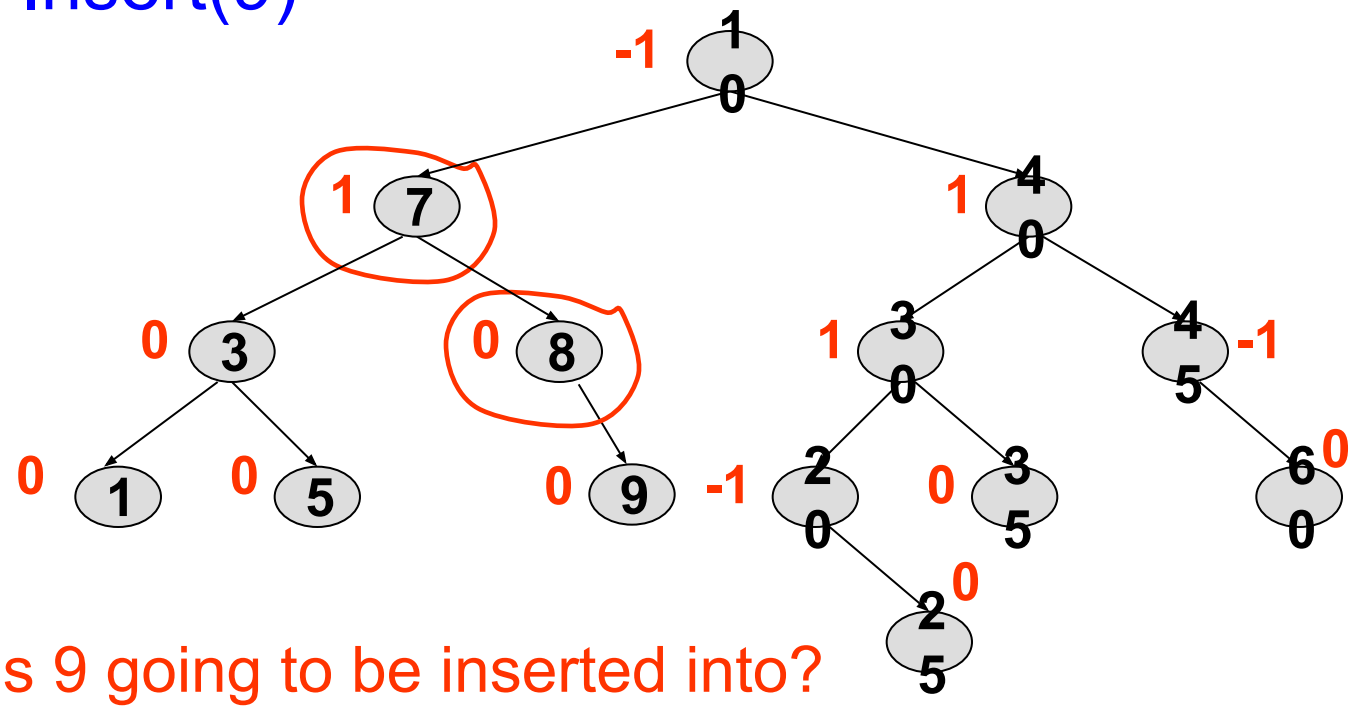
Step 2:If there is no node “A” ,then make another pass from the root , updating balance factors .Then terminate,Otherwise go to step 3

Step 3: If bf(A) =1 and the new node was inserted on the right side of “A” or if bf(A)=-1 and the new node was inserted on the left side of “A” ,then set the bf(A)=0 and terminate. Otherwise go to step 4.

Step 4: classify the imbalance at “A” and perform the appropriate rotations .Change balance factors as required by the rotations.

Inserting into an AVL Search Tree

Insert(9)



- Where is 9 going to be inserted into?
- After the insertion, is the tree still an AVL search tree? (i.e., still balanced? yes)

Imbalance Types in AVL insertion

After an insertion, when the balance factor of node A is –2 or 2, the node A is one of the following four imbalance types

1. **LL**: new node is in the left subtree of the left subtree of A
2. **LR**: new node is in the right subtree of the left subtree of A
3. **RR**: new node is in the right subtree of the right subtree of A
4. **RL**: new node is in the left subtree of the right subtree of A

LL Imbalance

A generic LL type imbalance is shown in the following figure .Figure (a) shows the conditions before the insertion ,(b) shows the situation following the insertion of an element into the left subtree BL of B. The subtree movement needed to restore balance at A appears in the figure (c). B becomes the root of the subtree that A was previously root of , B'L remains the left subtree of B, A becomes the root of B's right subtree, BR becomes the left subtree of A, and the right subtree of A is unchanged. The balance factors of nodes in B'L that are on the path from B to the newly inserted node change as does the balance factors A.

This kind of rotation is called LL rotation. Here , only single rotation is required.

An LL Rotation

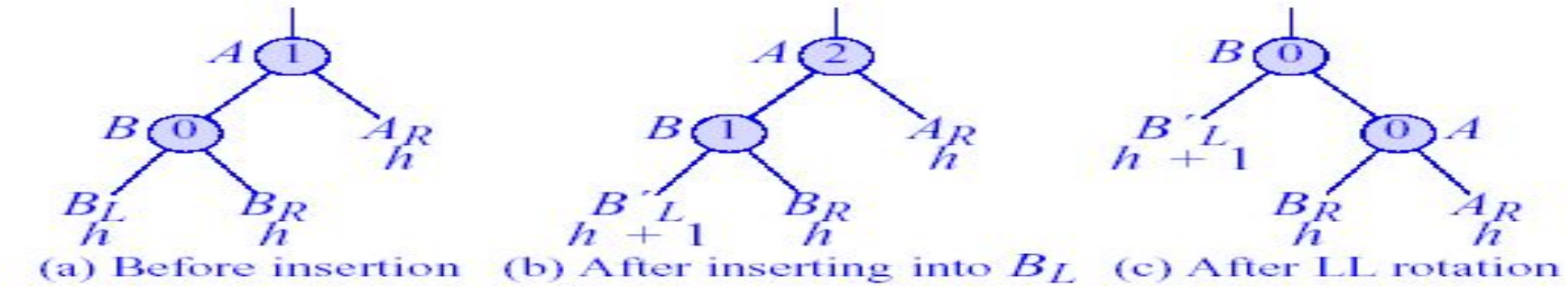
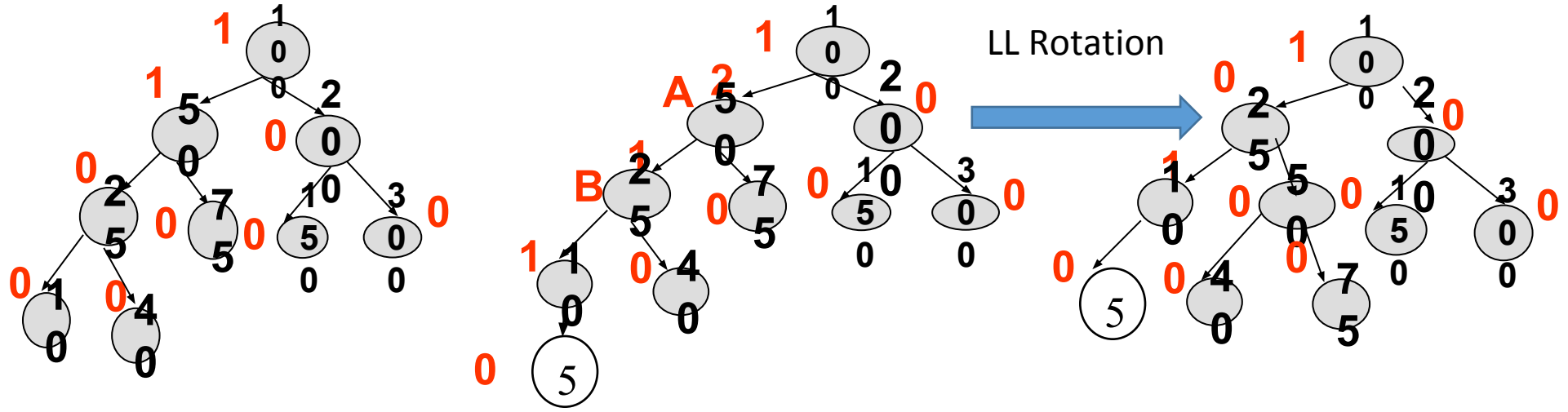


Figure An LL Rotation



An LR Rotation

LR rotation:Following figure LR type imbalance . Here, the insertion takes place in the right subtree of B.The rearrangement of subtrees needed to rebalance appears in the figure (c) .

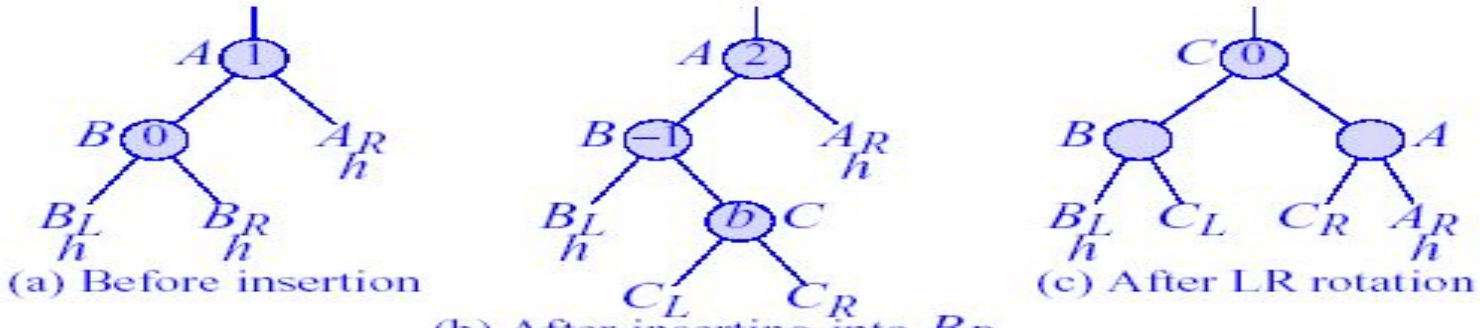
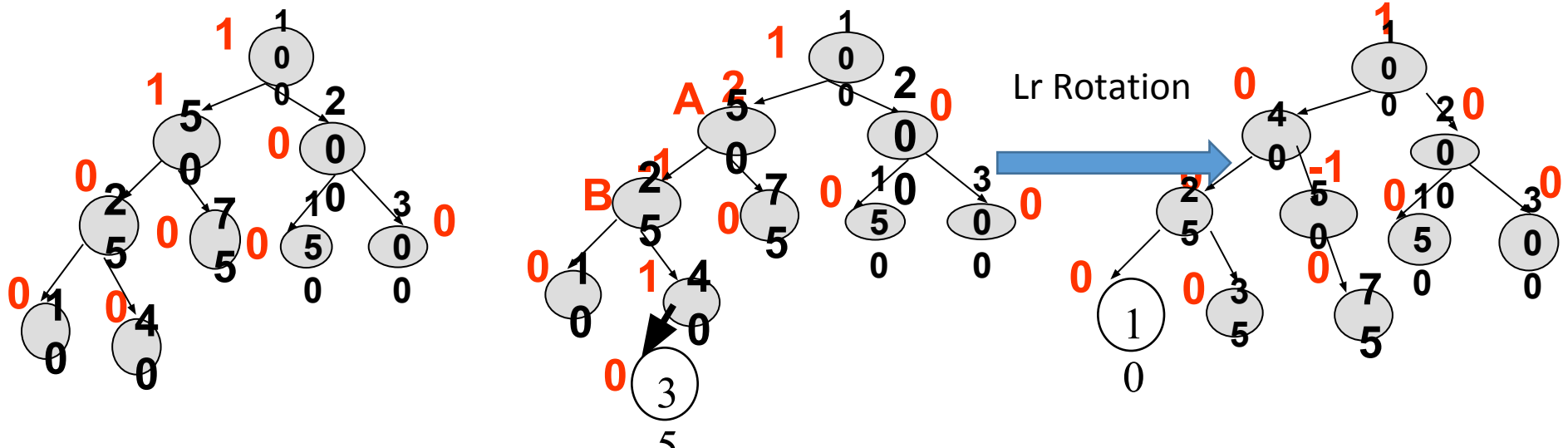


Figure An LR Rotation



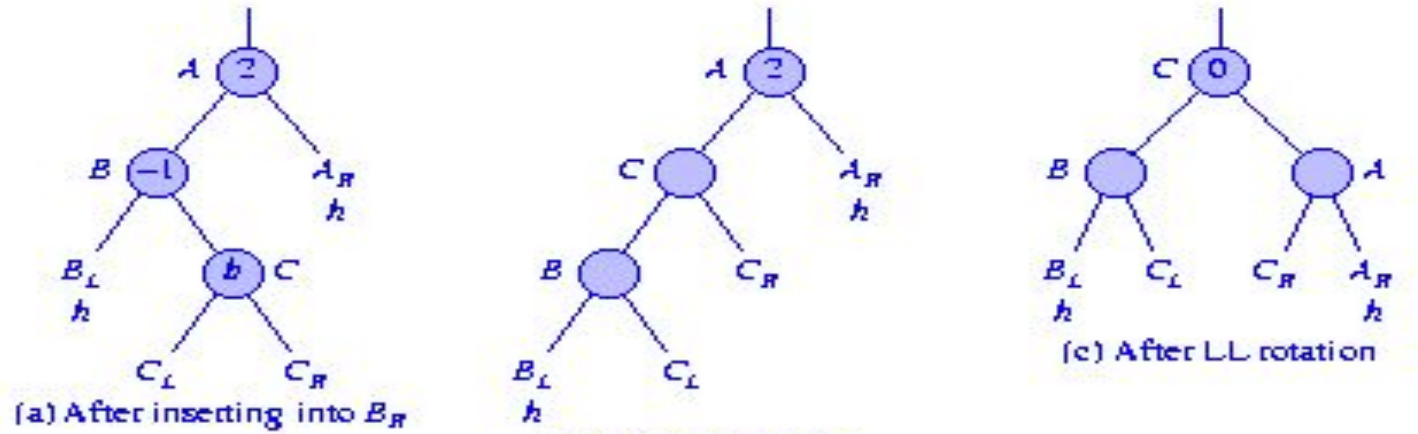
Single and Double Rotations

- Single rotations: the transformations done to correct LL and RR imbalances
- Double rotations: the transformations done to correct LR and RL imbalances
- The transformation to correct LR imbalance can be achieved by an RR rotation followed by an LL rotation

Single and Double Rotations

- Single rotations: the transformations done to correct LL and RR imbalances
- Double rotations: the transformations done to correct LR and RL imbalances
- The transformation to correct LR imbalance can be achieved by an RR rotation followed by an LL rotation as shown below

LR Rotation



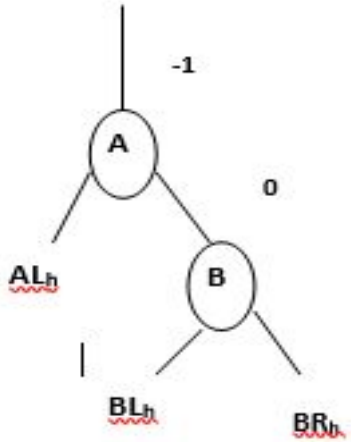
$b = 0 \Rightarrow bf(B) = bf(A) = 0$ after rotation
$b = 1 \Rightarrow bf(B) = 0$ and $bf(A) = -1$ after rotation
$b = -1 \Rightarrow bf(B) = 1$ and $bf(A) = 0$ after rotation

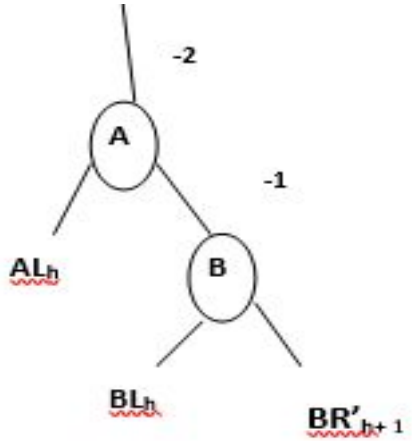
An LR rotation

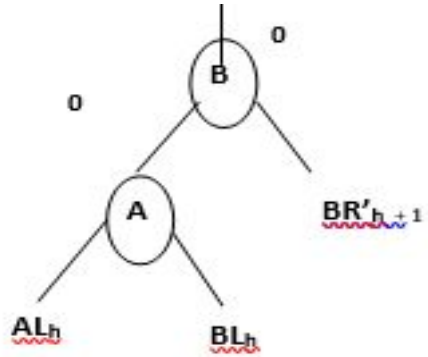
- Figure (b) shows the tree after an RR rotation at vertex B
- Figure (c) shows the result of performing an LL rotation at vertex A of the tree of Figure (b).
- The resulting tree is the same as that shown in Figure (c).

RR imbalance: A generic RR type imbalance is shown in the following figure . Figure (a) shows the conditions before the insertion ,(b) shows the situation following the insetion of an element into the right subtree BR of B. The subtree movement needed to restore balance at A appears in the figure (c). B becomes the root of the subtree that A was previously root of , B'R remains the right subree of B, A becomes the left subtree of B,BL becomes the fight subtree of A, and the left subtree of A is unchanged. The balance factors of nodes in B'R that are on the path from B to the newly inserted node will be changed.

An RR Rotation

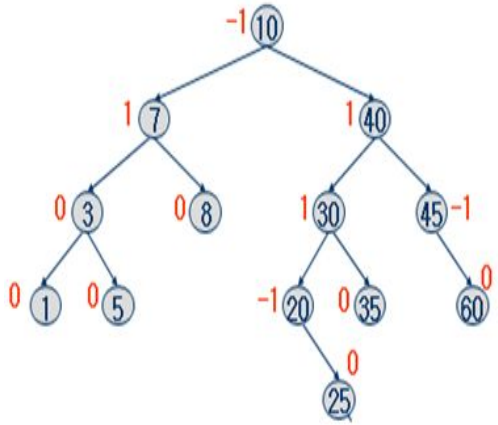


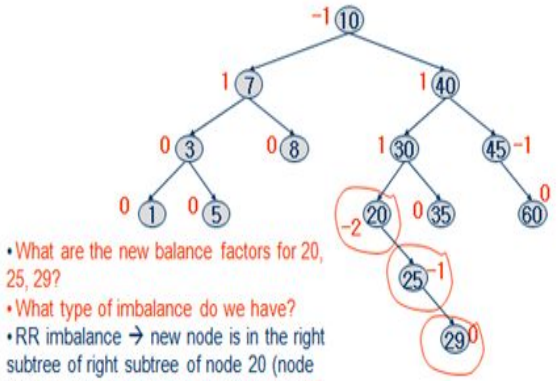




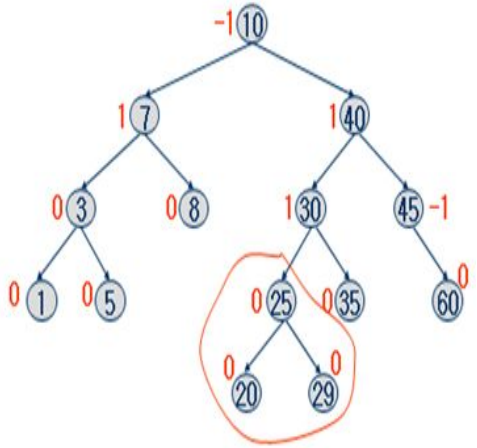
a)Before insertion b) After insertion c) After RR rotation

Example:

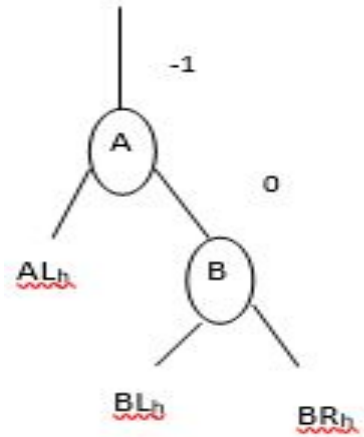


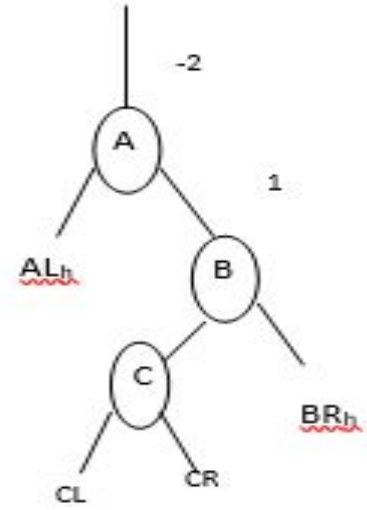


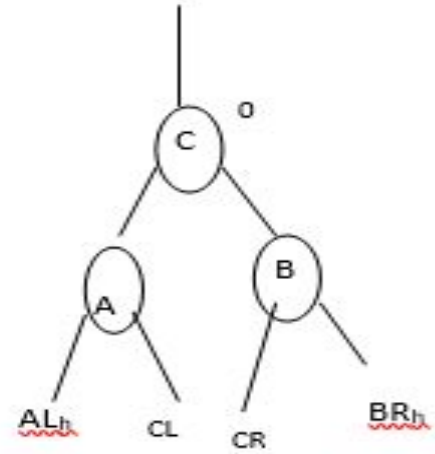
- What are the new balance factors for 20, 25, 29?
- What type of imbalance do we have?
- RR imbalance → new node is in the right subtree of right subtree of node 20 (node with bf = -2) → what rotation do we need?
- What would the left subtree of 30 look like after RR rotation?



RL rotation:





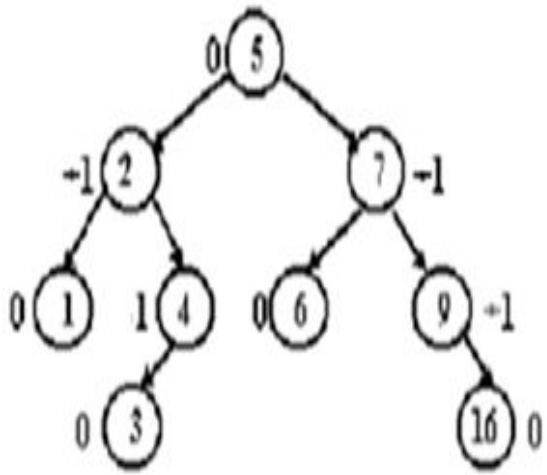


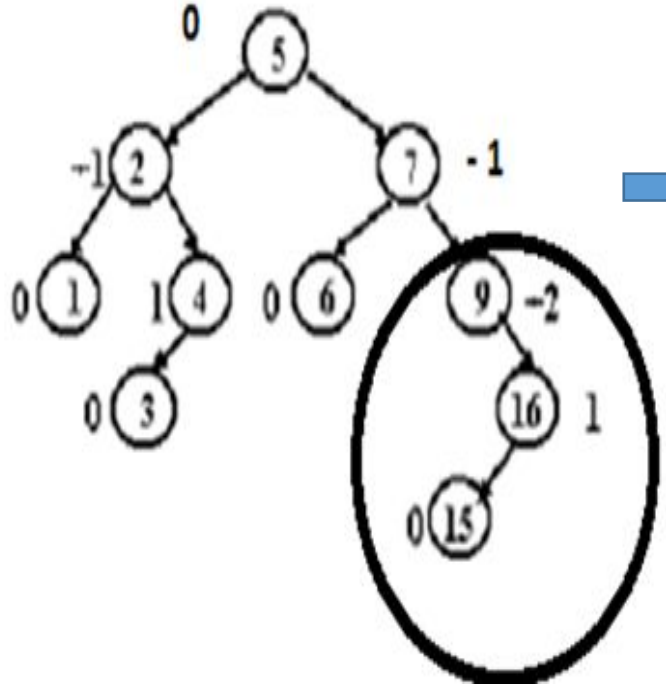
a)Before insertion

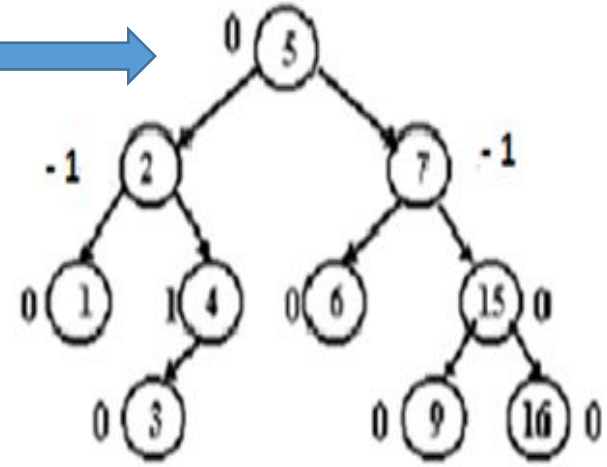
b) after insertion

c) After RL rotation

-

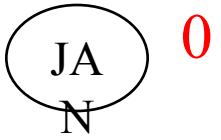




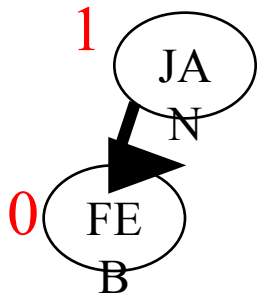


Example for insertion into an AVL

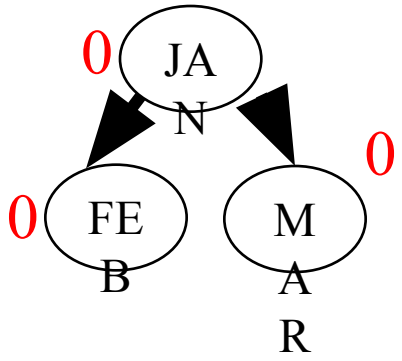
1)Insert “JAN”



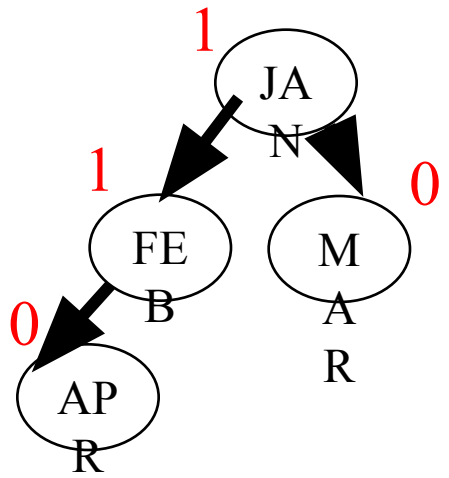
2)Insert “FEB”



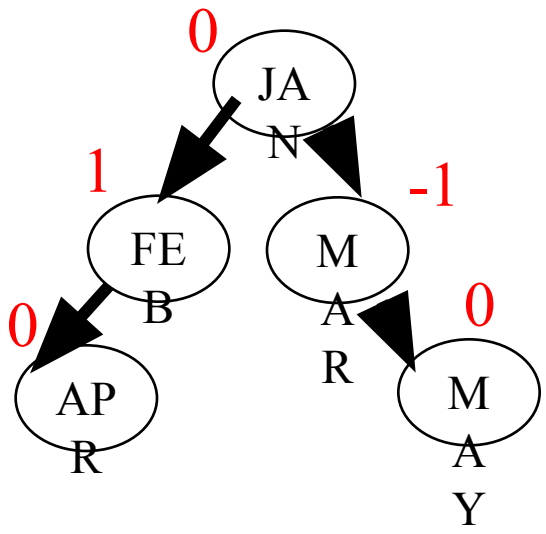
3)Insert “MAR”



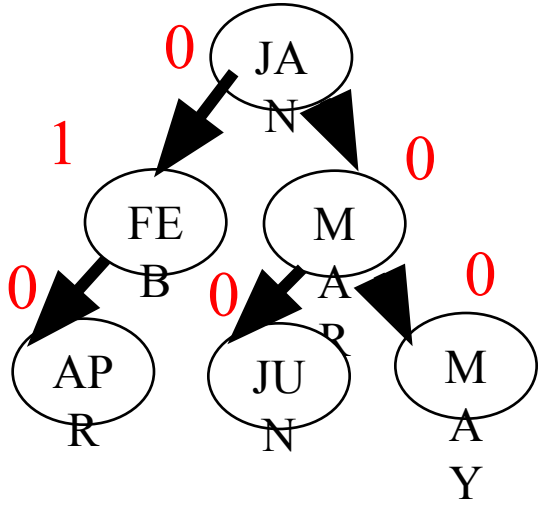
4)Insert “APR”



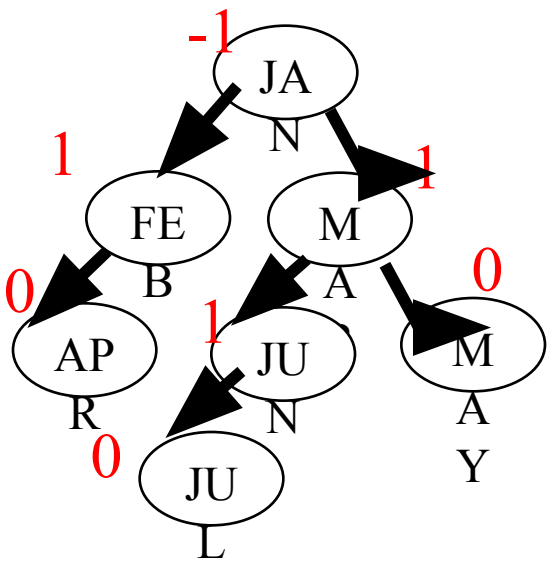
5)Insert “MAY”



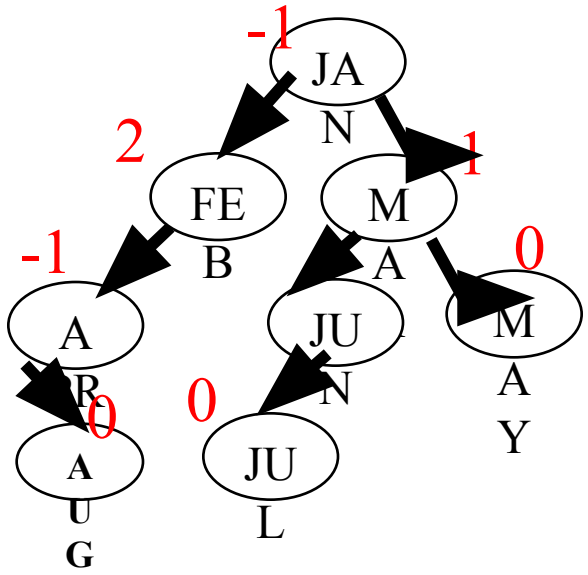
6)Insert “JUN”



6)Insert “JUL”



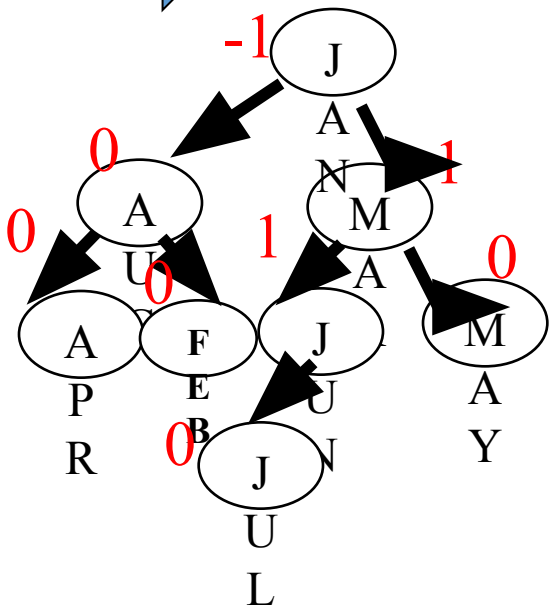
6)Insert “AUG”



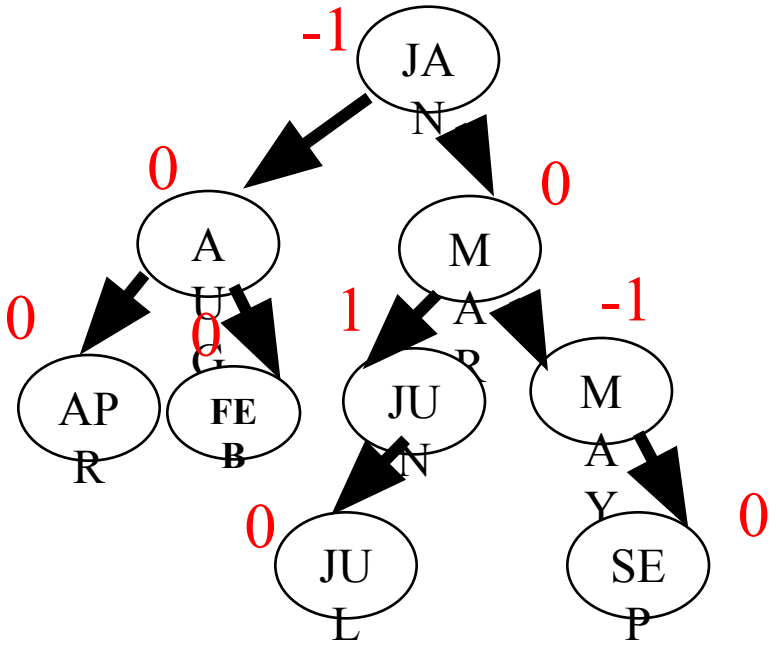
LR imbalance

LR Rotation

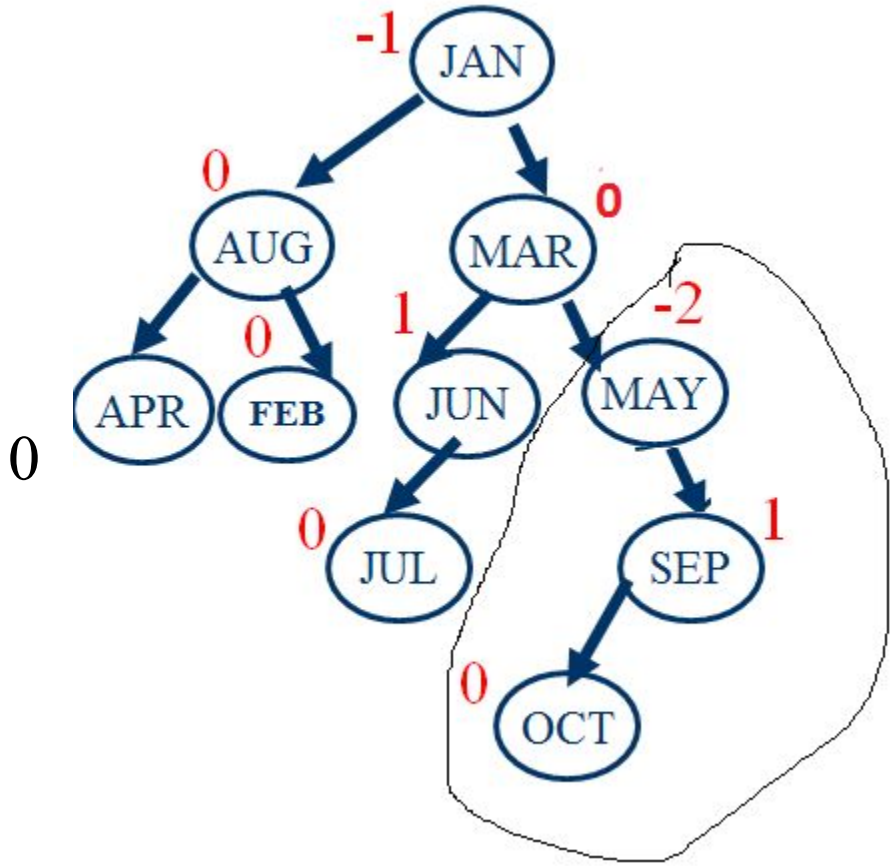
6)Insert “AUG”



6)Insert “SEP”

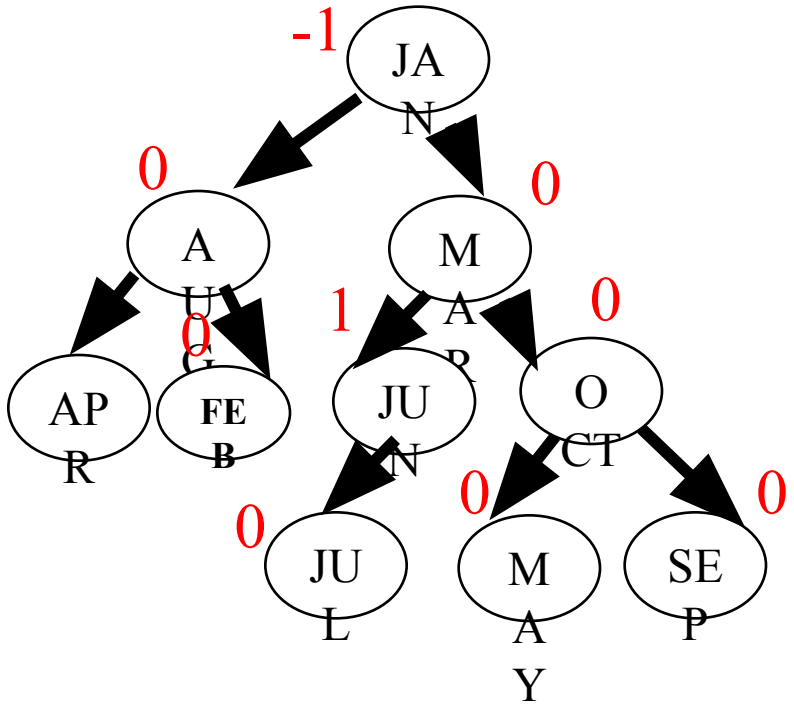


7)Insert “OCT”

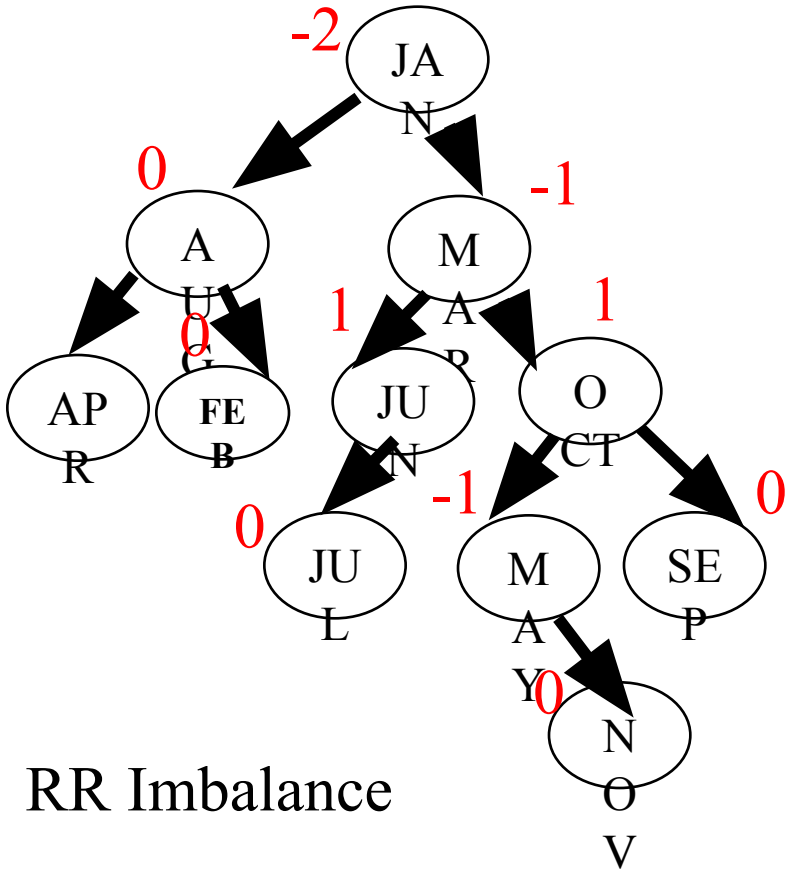


RL imbalance

RL rotation

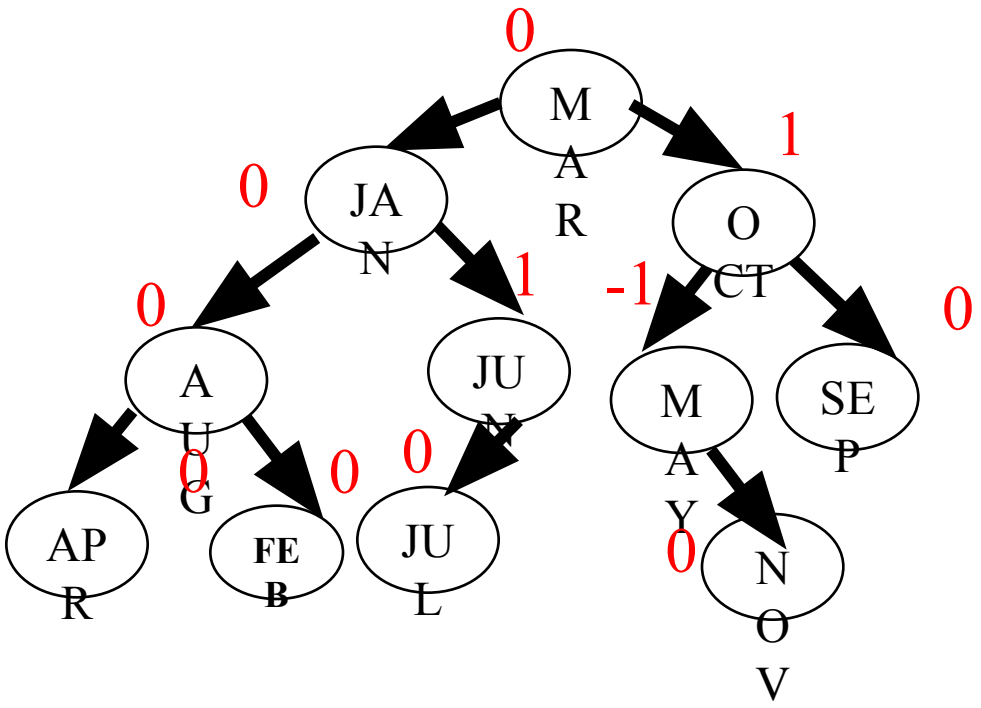


Insert NOV



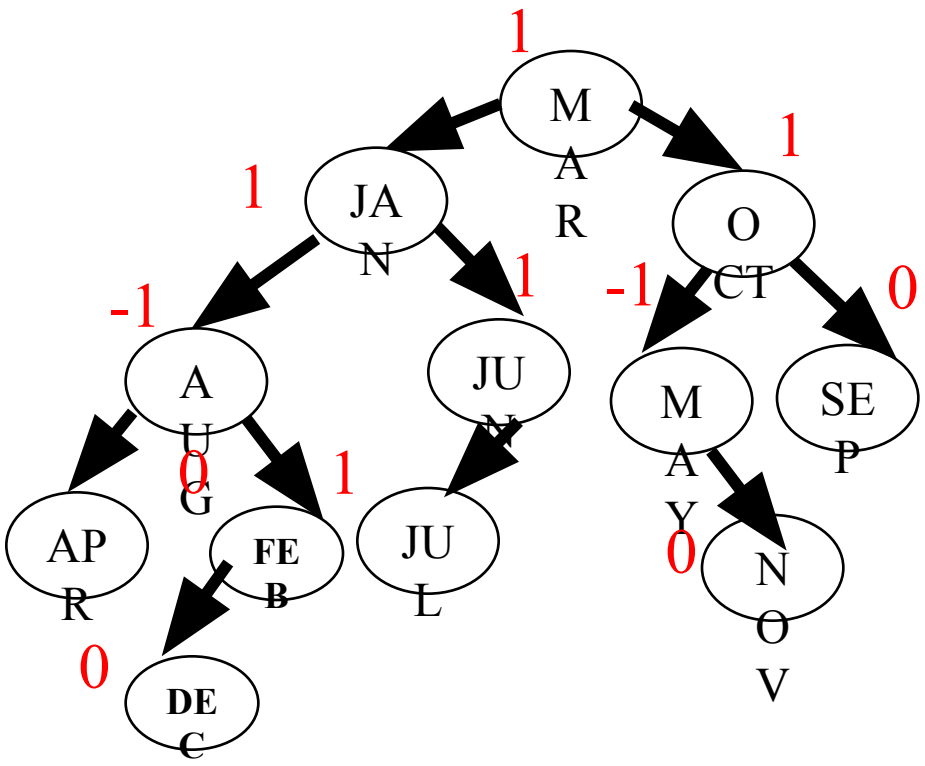
RR Imbalance

RR Rotation

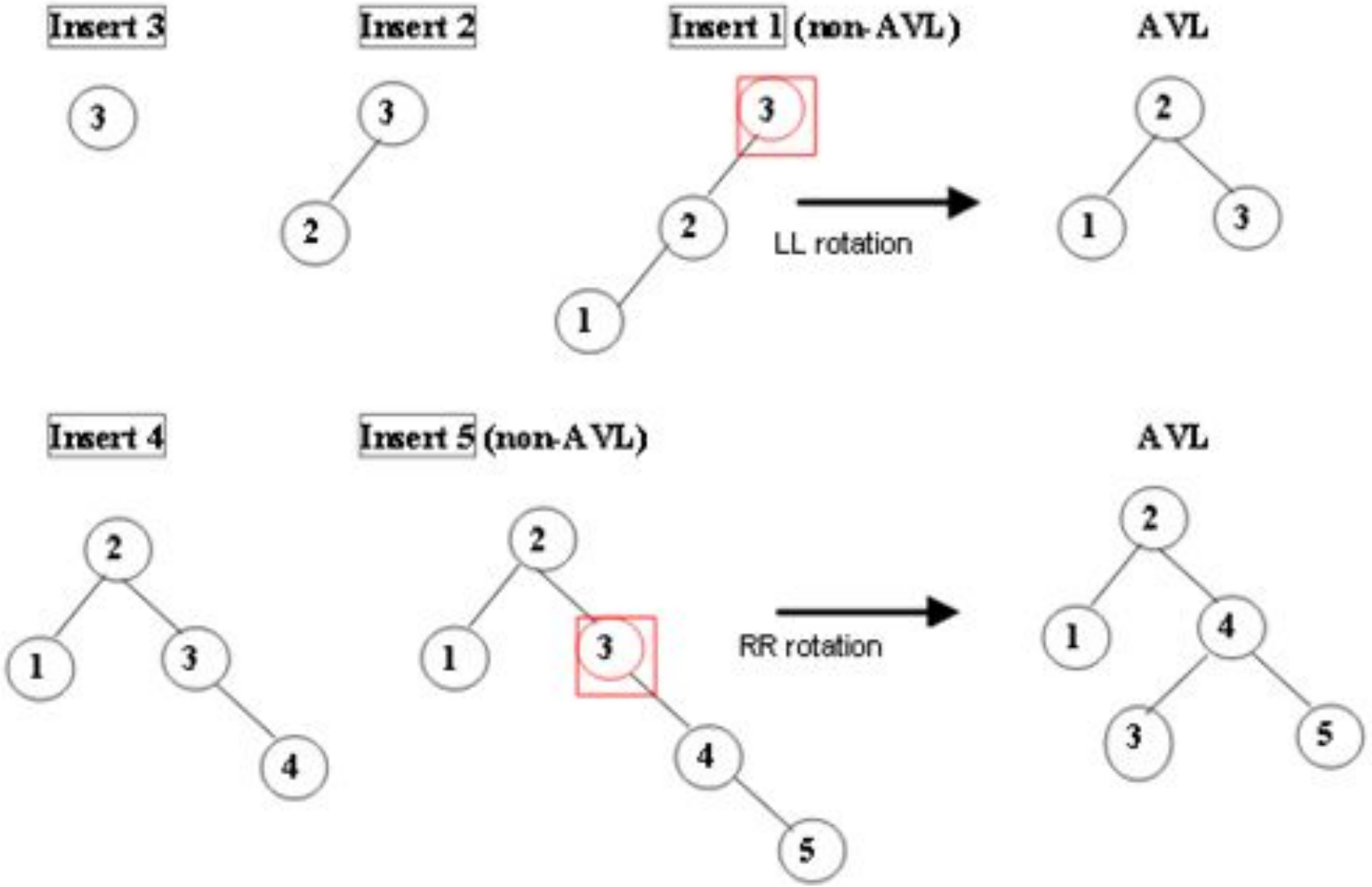


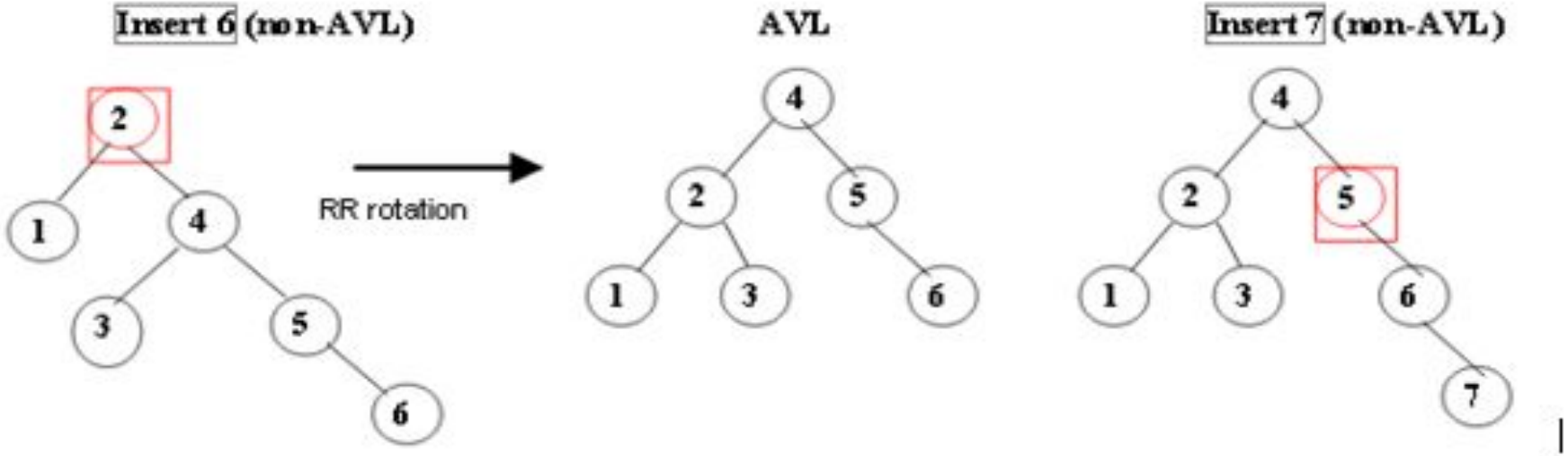
RR Imbalance

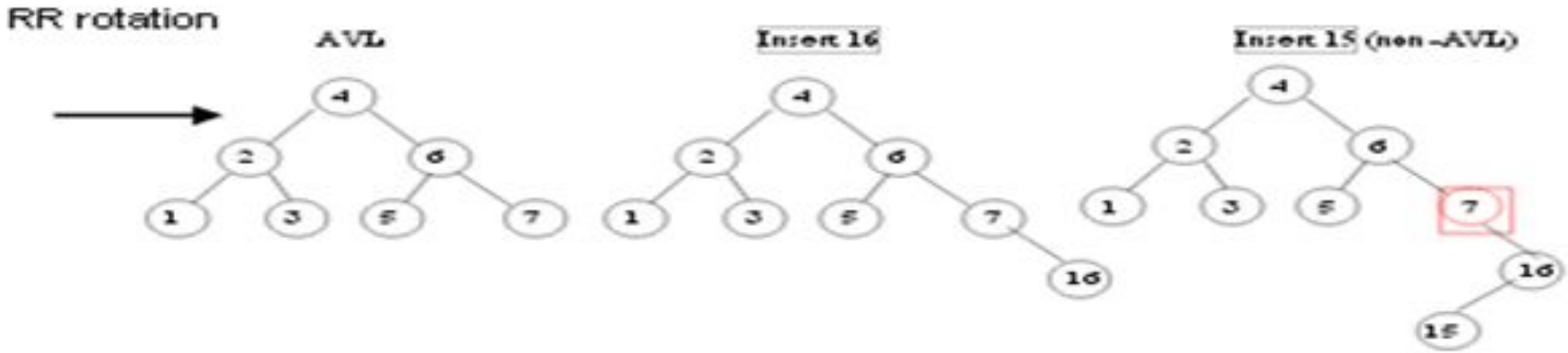
Insert “DEC”



Example for insertion into an AVL




Insert 6 (non-AVL)
AVL
Insert 7 (non-AVL)
RR rotation


RR rotation
AVL
Insert 16
Insert 15 (non-AVL)


Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
RL rotation

Deletion from an AVL Search Tree:

To delete a node from an AVL tree , the following procedure to be used.

Algorithm Deletion from AVL

Step 1:Search for the node say Q to be deleted. Let PQ is the parent of Q
Step 2:Delete Q by using similar procedure whichever is used for deletion of node from binary search tree , then go to step 3.

Step 3: //updation of balance factor of PQ which is the parent of Q
if Q is on the left side of PQ decrease balance factor of PQ by 1. If Q is on the right side of PQ increase the balance factor of PQ by 1. Then go to step 4 , 5 or 6 based on updated balance factor of PQ.

Step 4: if updated balance factor of PQ is zero means that the height of PQ is reduced by one so update the balance factors of its ancestors and then terminate.

Step 5: if updated balance factor of PQ is 1 or -1 means that the height of PQ is not changed, so no need of changing balance factors of its ancestors. Simply terminate.

Step 6: if updated balance factor of PQ is 2 or -2 means the tree is imbalanced at PQ. Possible imbalances are R0 , R1, R-1 , L0,L-1 and L1. So perform appropriate rotation to make the tree as balanced and then terminate. In case of R1 , R-1 ,L1 and L-1 imbalances, after performing concerned rotations, ancestors balance factors also to be updated.

Imbalances in the AVL deletion:

After deletion of a node Q from AVl tree , balance factor changes may propagate up the tree along the path from Q to the root .During this process ,it is possible for the balance factor of a node on this path to become -2 or 2. Let A be the first such node on this path. To restore balance at node A, we classify the type of imbalance.

L imbalance occurs , if the deletion takes place from the left subtree of A.

R imbalance occurs , if the deletion takes place from the right subtree of A.

R imbalance is further classified into three types such as R0, R1 and R-1 imbalances depends on balance factor of B, where B is left child of A.

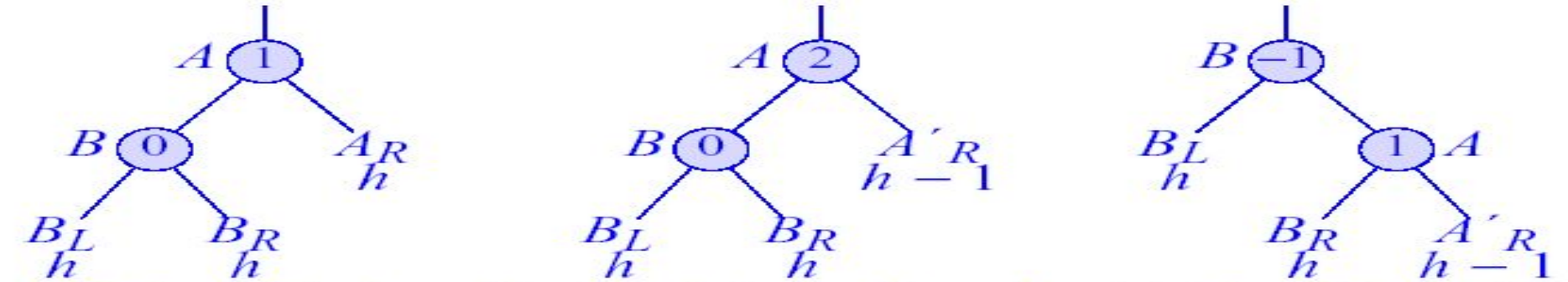
The Ro refers to the case when the deletion took place from the right subtree of A and bf(B)=0 .
The R1 refers to the case when the deletion took place from the right subtree of A and bf(B)=1.
The R-1 refers to the case when the deletion took place from the right subtree of A and bf(B)=-1.

L imbalance is further classified into three types such as L0, L1 and L-1 imbalances depends on balance factor of B, where B is the right child of A.

The Lo refers to the case when the deletion took place from the left subtree of A and bf(B)=0 .
The L1 refers to the case when the deletion took place from the left subtree of A and bf(B)=1.
The L-1 refers to the case when the deletion took place from the left subtree of A and bf(B)=-1.

An R0 Rotation

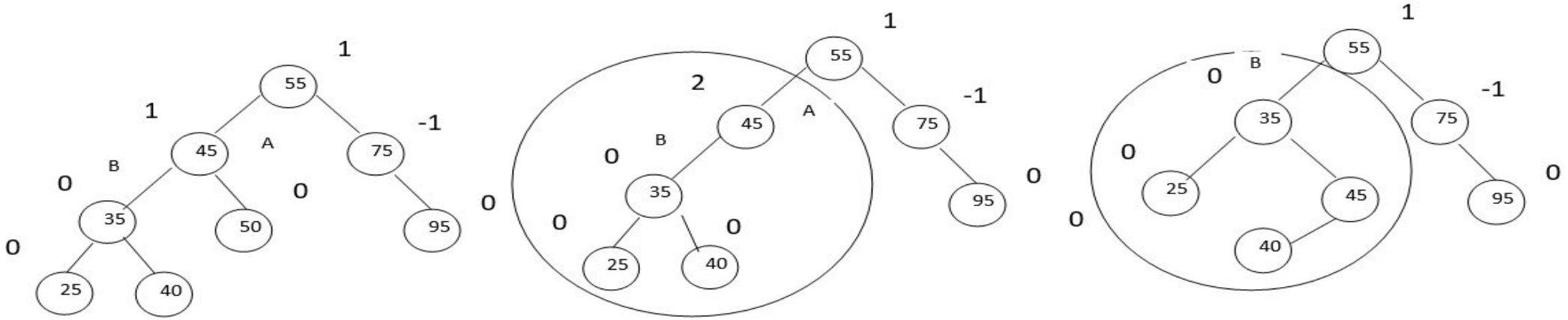
An R0 imbalance at A can be rectified by performing the rotation shown in the following figure. Notice that the height of the subtree was h+2 before deletion and it is h+2 after deletion also. So , the balance factors of the remaining nodes on the path to the root are unchanged.



(a) Before deletion (b) After deleting from A_R (c) After R0 rotation

Figure 11.11 An R0 rotation (single rotation)

Figure : An R0 rotation (single rotation)



An R1 Rotation

Following figure shows how to handle an R1 imbalance. While he pointer changes ae the same as for an R0 imbalance, the new balance factors for A and B are different and the height of the subtree following the rotations is now h+1, which is 1 less than before deletion. So, if A is not the root, the balance factors of some of its ancestors will change and further rotations may be necessary. Hence, following an R1 rotation , we must continue to examine nodes on the path to the root.

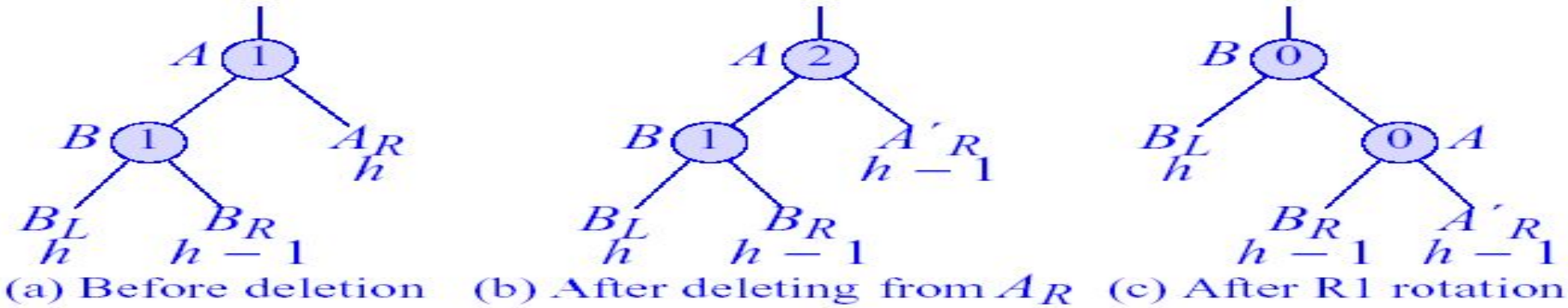
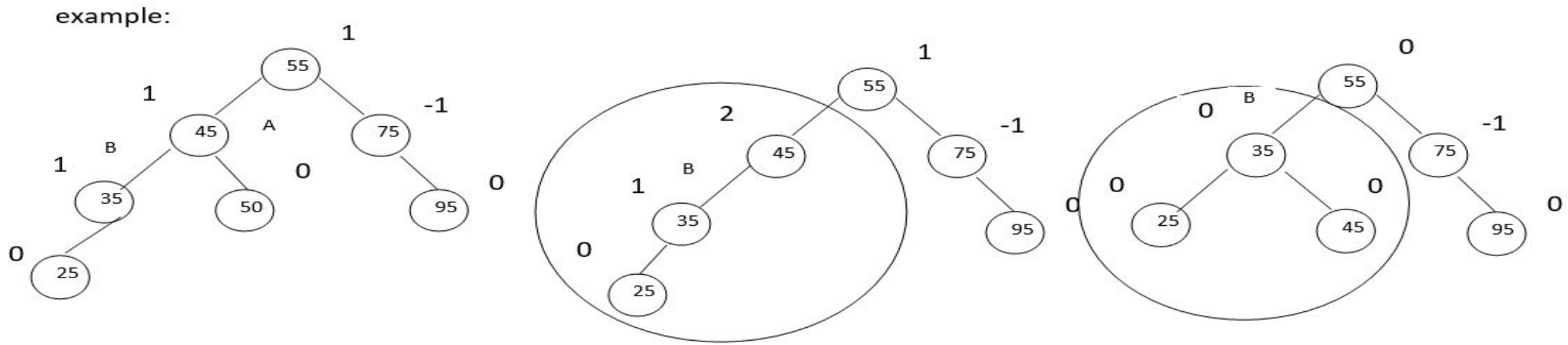


Figure .An R1 rotation (single rotation)



An R-1 Rotation

An R-1 Rotation : The transformation needed when the imabalance is of type R-1 is illustrated in the following figure. The balance factors of A and B following the rotation depends on the balance factor "b" of the right child of B. the rotation leaves a subtree of height h+1, while the subtree height prior to the deletion was h+2. So , we need to continue to examine nodes on the path to the root.

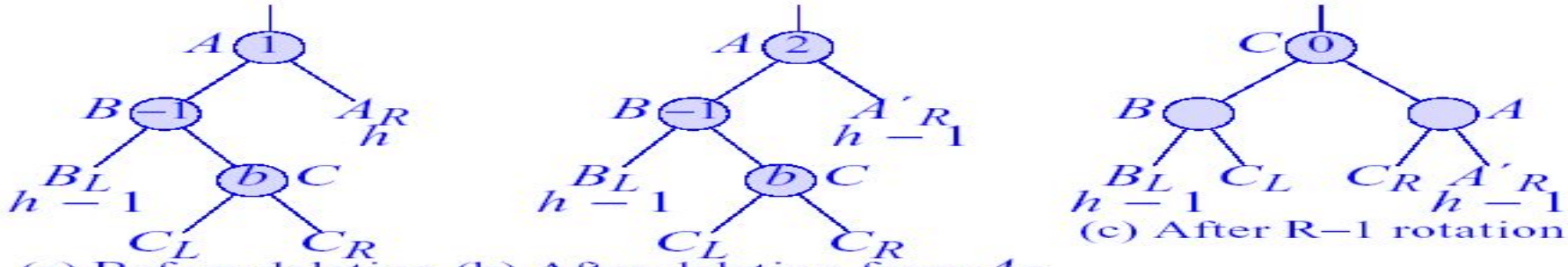
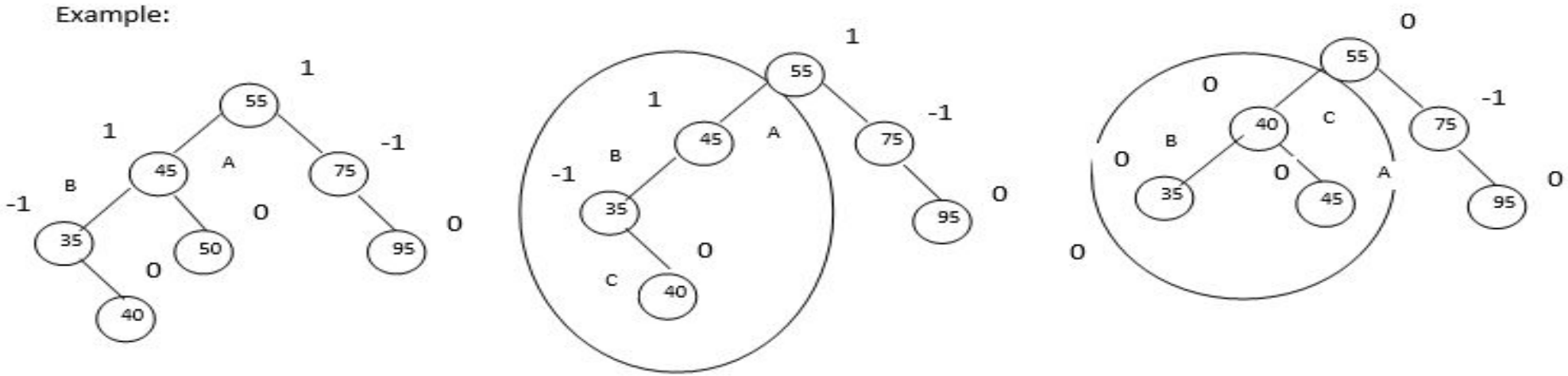
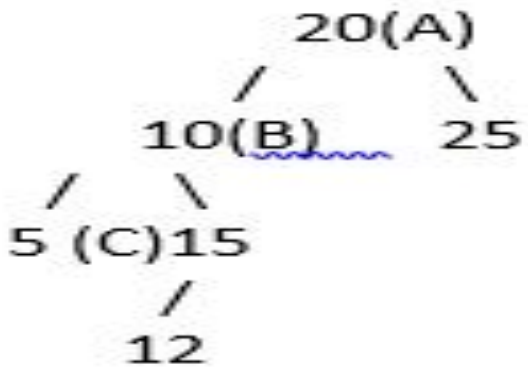


Figure An R-1 rotation (double rotation)



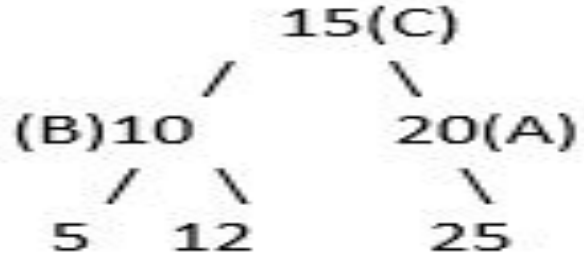
Example 1: Deletion of 30 from the following tree

Before deletion



After deletion

The restructure is as follows:



After R-1 imbalances

m-way search tree of order m, is a tree in which

A. The nodes hold between 1 to m-1 distinct keys
B. The keys in each node are sorted
C. A node with k keys has k+1 sub trees, where the sub trees may be empty.
D. The ith sub tree of a node [v1, ..., vk], 0 < i < k, may hold only values v in the range vi < v < vi+1 (v0 is assumed to equal - , and vk+1 is assumed to equal infinity).

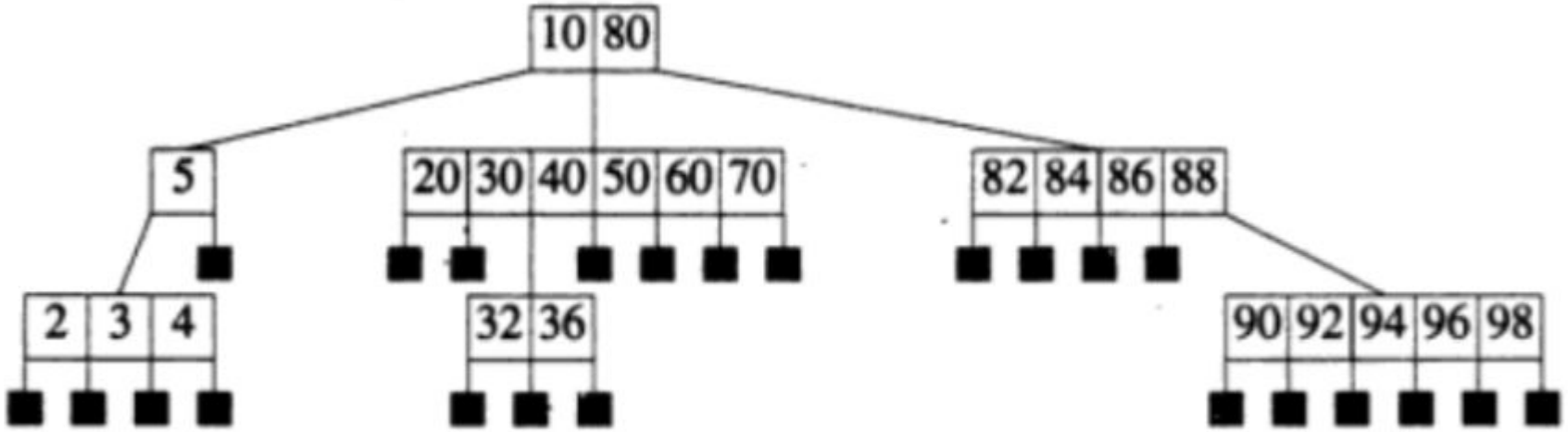


Figure A seven-way search tree

3.B-Tree

A B-tree of order m is an m-way search tree. If the B-tree is not empty, the corresponding extended tree satisfy the following properties:

Property #1 - All the leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

Property #4 - If the root node is a non leaf node, then it must have at least 2 children.

Property #5 - All the key values within a node must be in Ascending Order.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

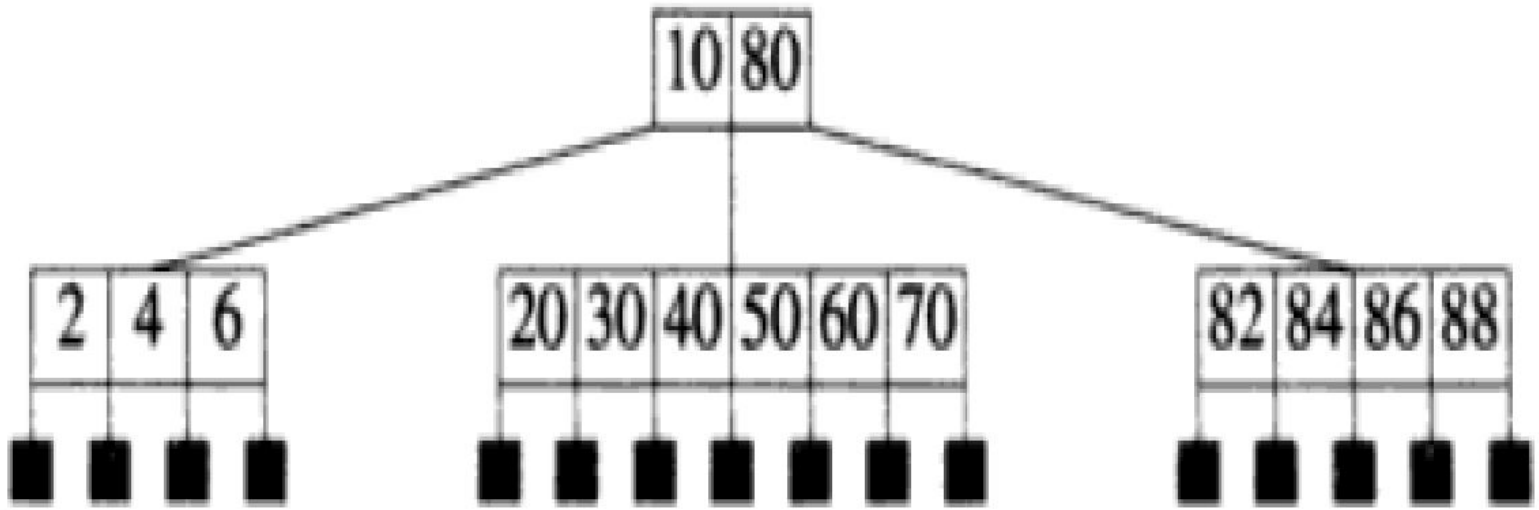


Figure 15.24 A B-tree of order 7

In a B-tree of order 3 , no internal nodes have either 2 or 3 children. So , a B-tree of order 3 is also known as 2-3 tree. Example for 2-3 B Tree is shown below

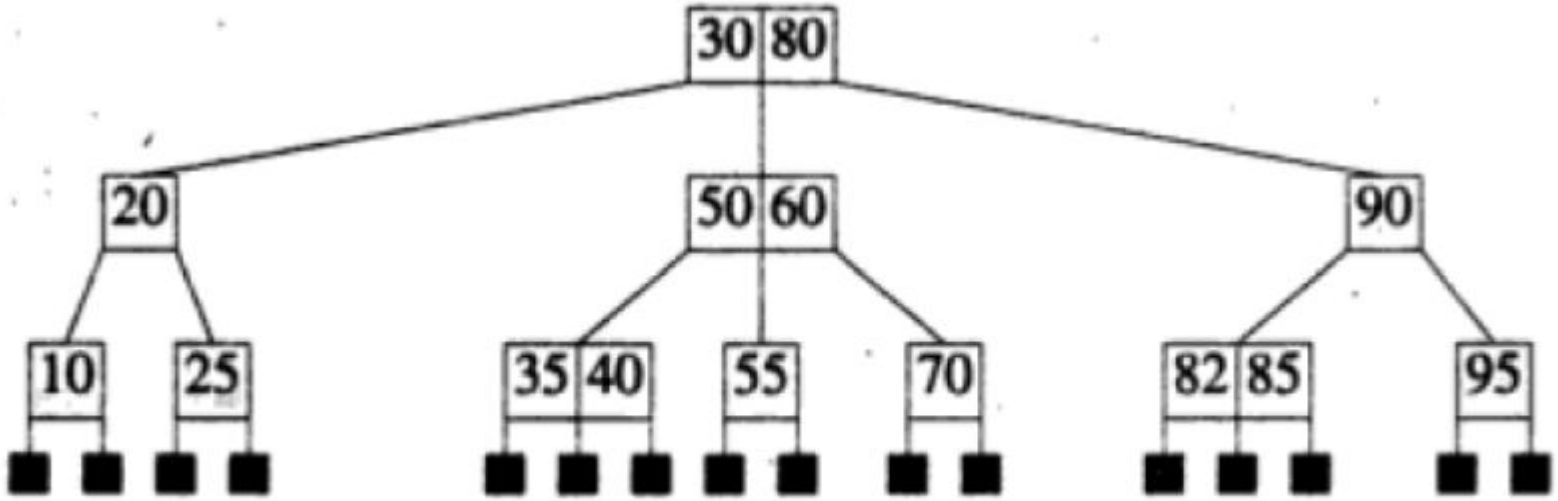


Figure . A 2-3 tree or B-tree of order 3

Operations on B-tree:

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with first key value of root node in the tree.
- Step 3: If both are matching, then display "Given node found!!!" and terminate the function
- Step 4: If both are not matching, then check whether search element is smaller or larger than that key value.
- Step 5: If search element is smaller, then continue the search process in left subtree.
- Step 6: If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparison completed with last key value in a leaf node.
- Step 7: If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

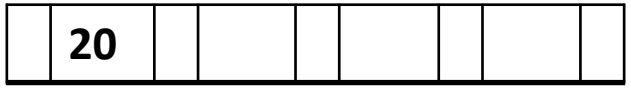
Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

Step 1: Check whether tree is Empty.
Step 2: If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
Step 5: If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sent value is fixed into a node.
Step 6: If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

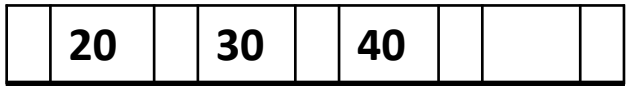
Example: **Construct B-tree order 5 for the following data**
20,40,30,350,75,375,360,80,60,90,70, 100,110 ,120, 385, 390,380

Step 1: insert (20)



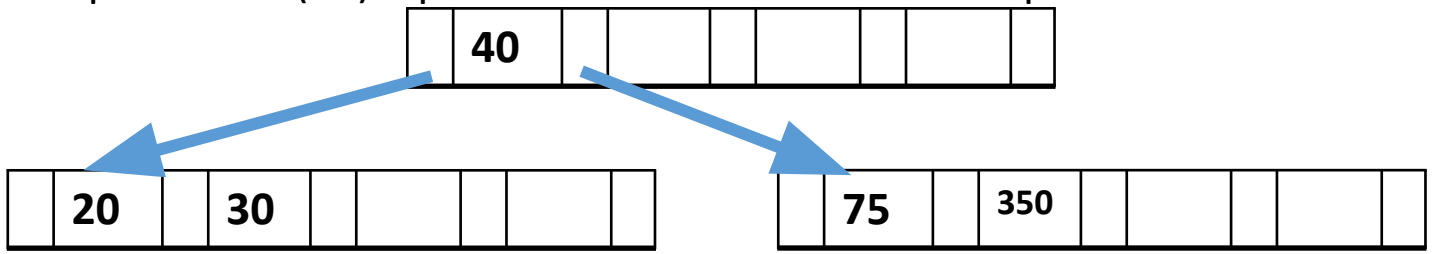
Step 2: insert (40)

Step3: insert (30)

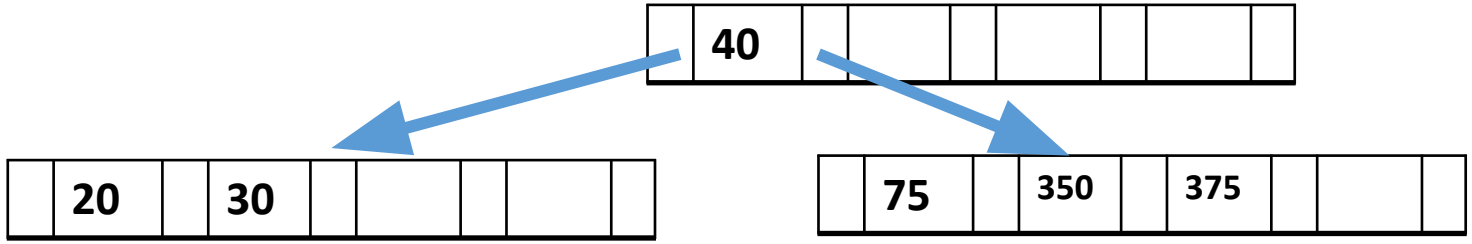


Step 4: insert (350)

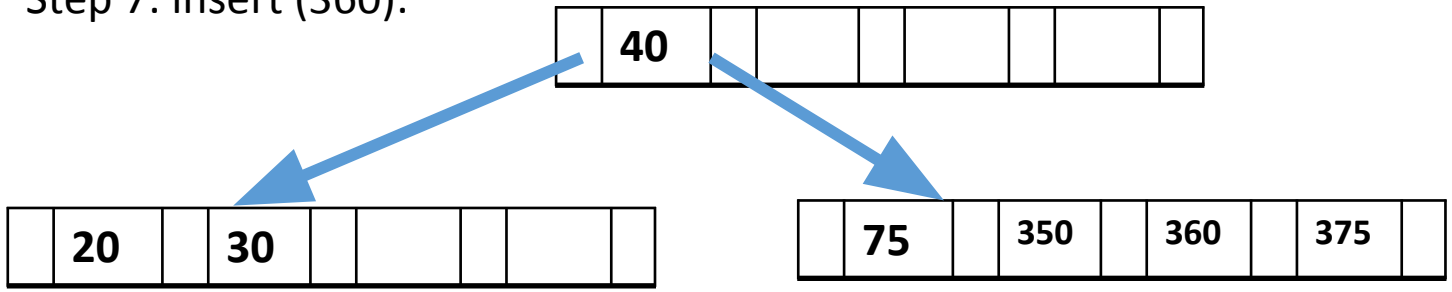
Step 5: insert (75): Split and send middle element up



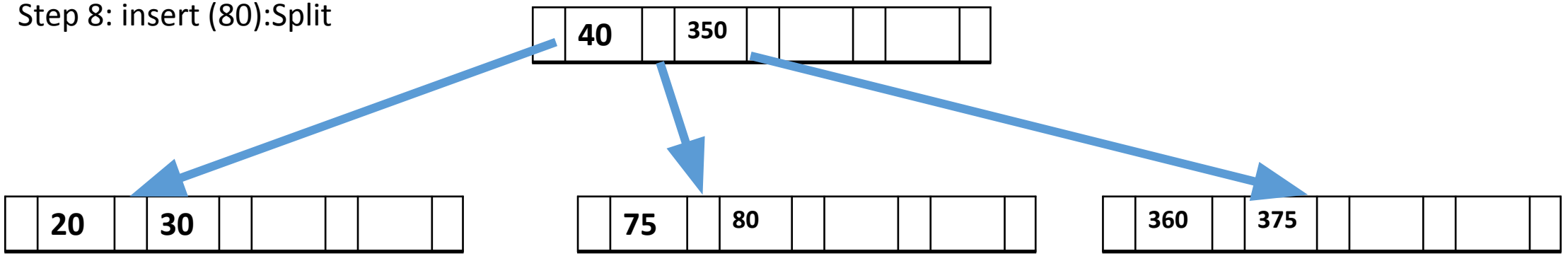
Step 6: insert (375):



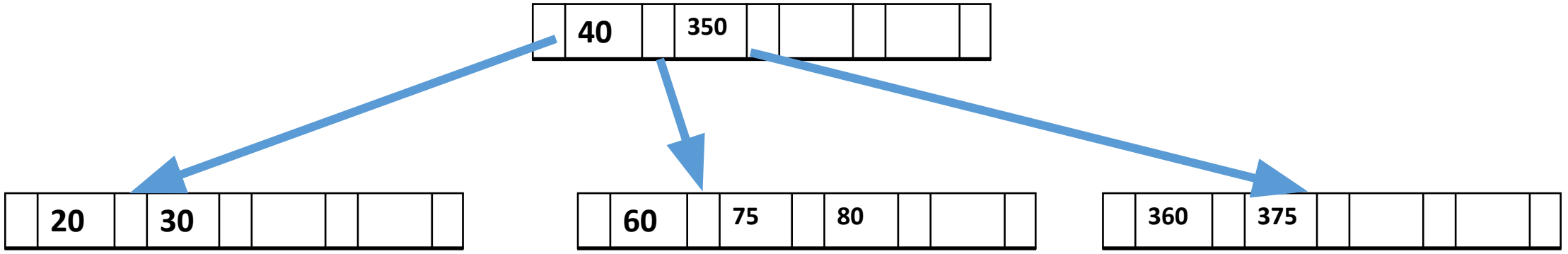
Step 7: insert (360):



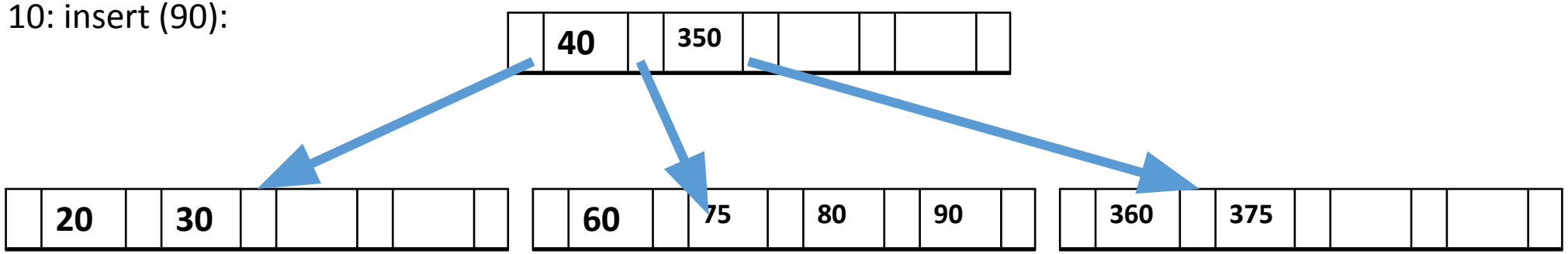
Step 8: insert (80):Split



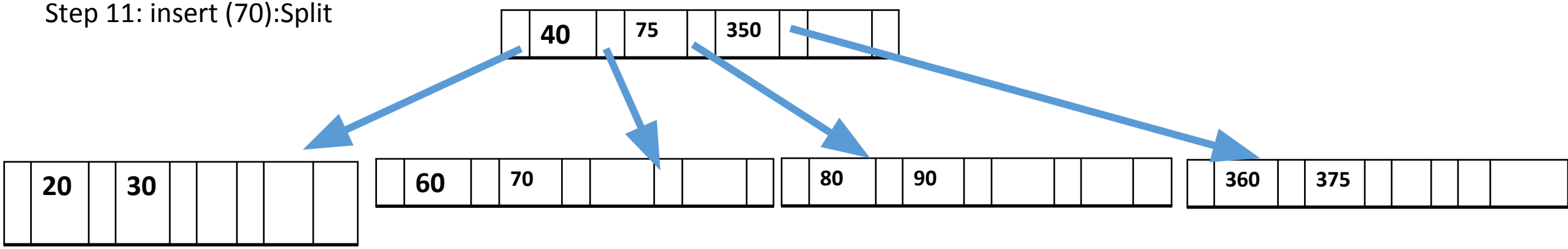
Step 9: insert (60):



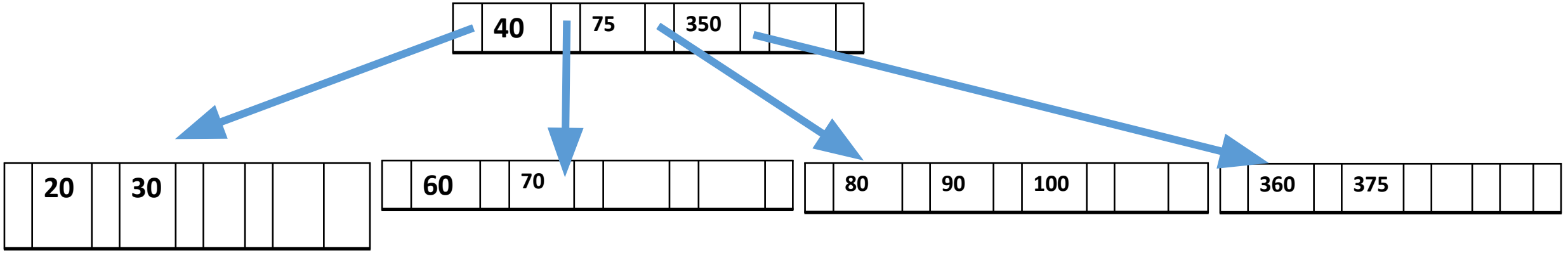
Step 10: insert (90):



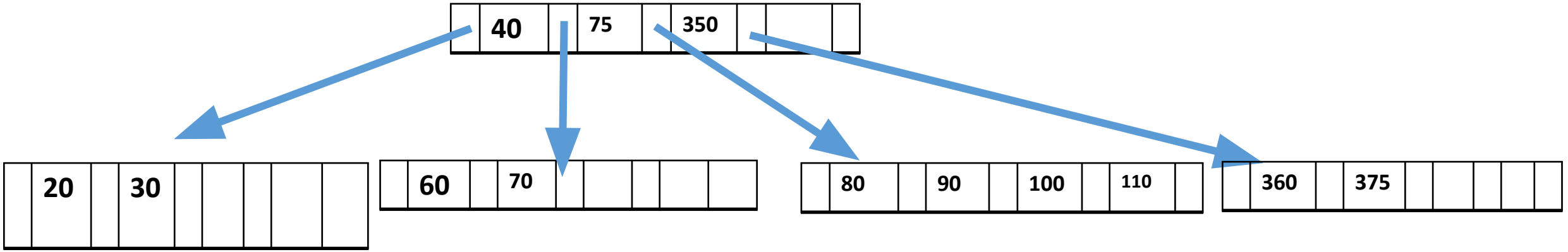
Step 11: insert (70):Split



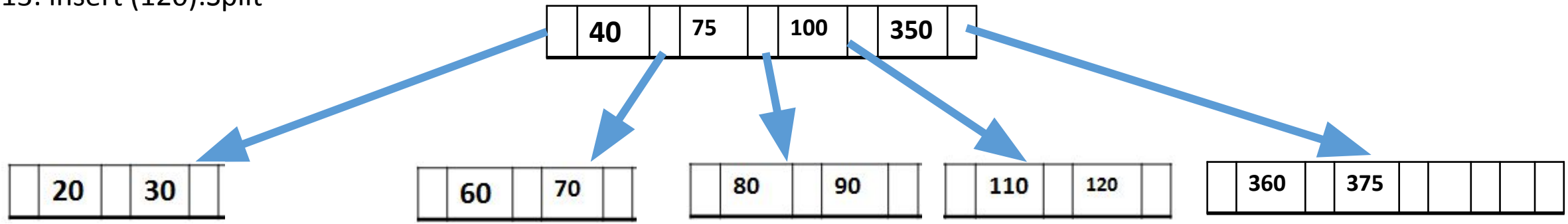
Step 12: insert (100)



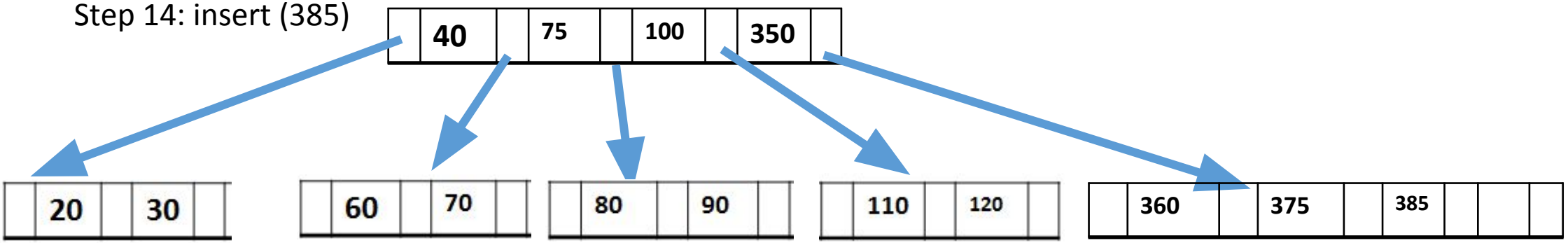
Step 13: insert (110)



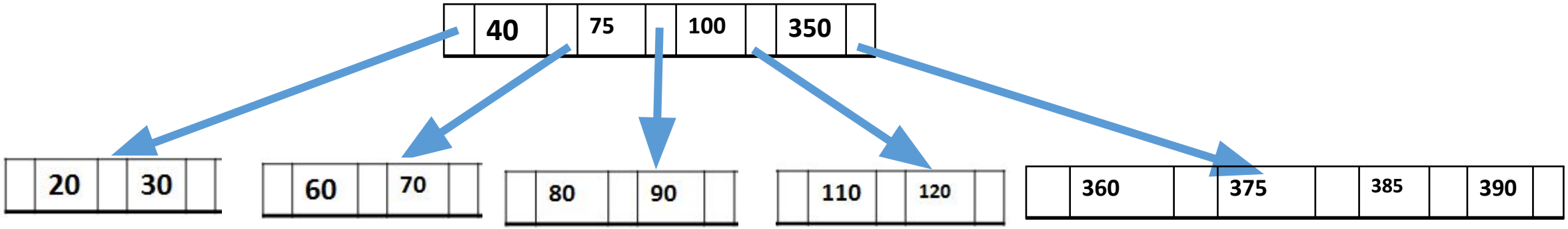
Step 13: insert (120):Split



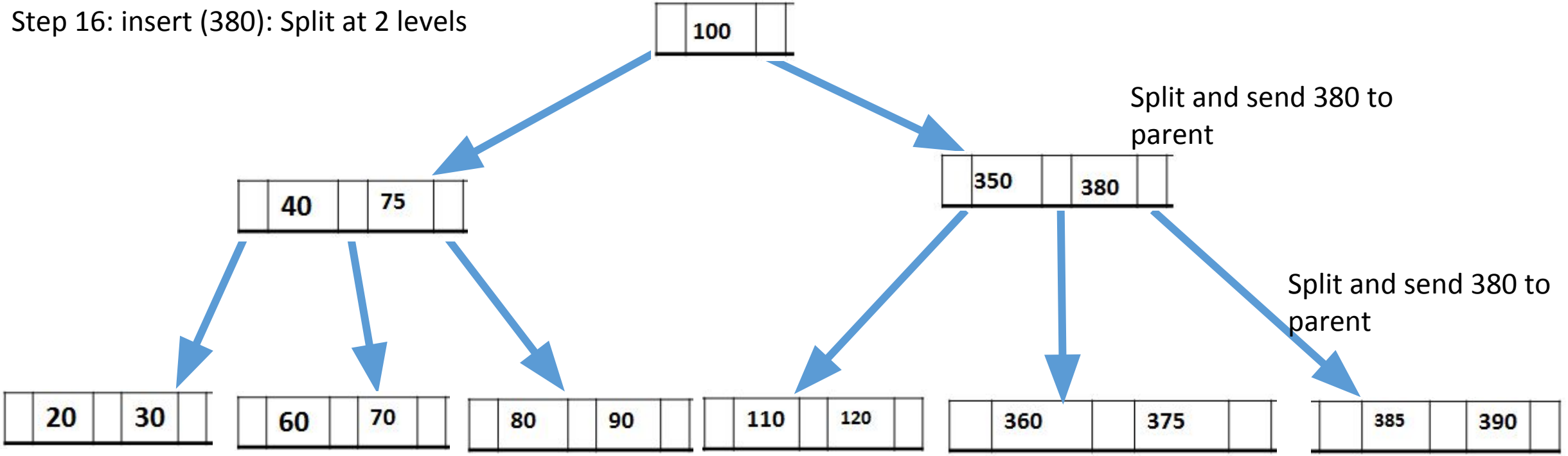
Step 14: insert (385)



Step 15: insert (390)



Step 16: insert (380): Split at 2 levels



Split and send 380 to parent

Split and send 380 to parent

Deleting an Element From a B-Tree

- Deletion is divided into two cases:

1. The element to be deleted is in a leaf node
2. The element to be deleted is in a non leaf node

Case 1:

To delete value X from a B-tree, starting at a leaf node, there are 2 steps:

1.Remove X from the current node. Being a leaf node there are no subtrees to worry about.
2.Removing X might cause the node containing it to have *too few*

If underflow does not occur, then we are finished the deletion process. If it does occur, it must be fixed.

NOTE: After deleting an element from B tree, the resultant tree must stratify B-tree Properties

Example 1: Let us take as a specific example, deleting 6 from this B-tree (of degree 5):

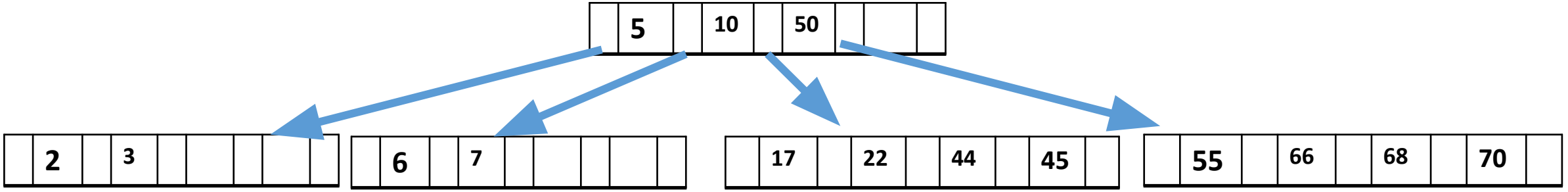
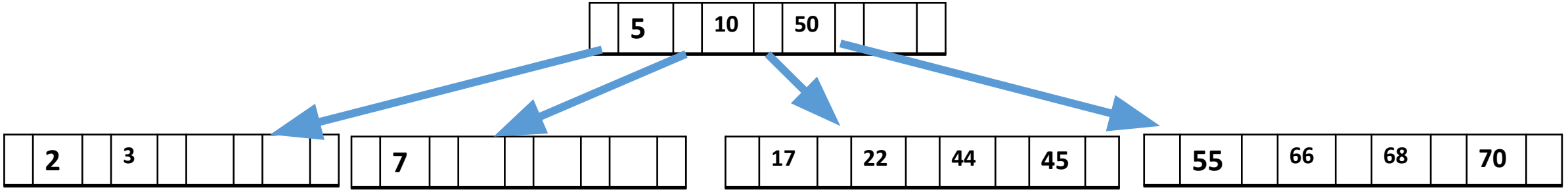
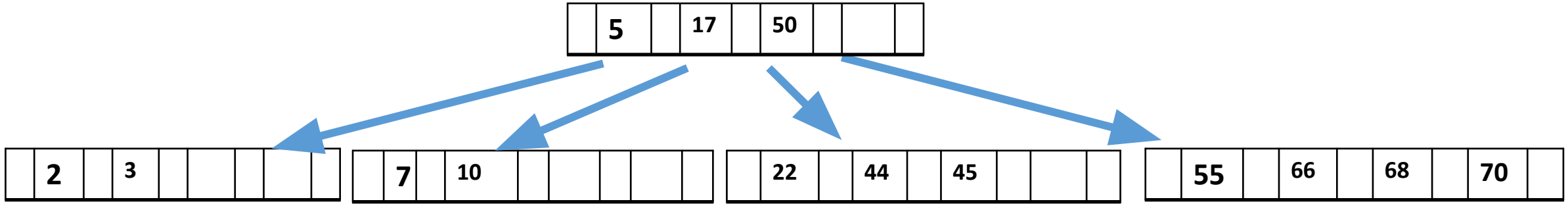


Fig.Before Deletion

Here 6 is in the leaf , so simply delete and observe the resultant.



Removing 6 causes the node it is in to underflow, as it now contains just 1 value (7). Our strategy for fixing this is to try to `borrow' values from a neighbouring node. After borrowing from the neighbour, the resultant tree is shown below:



Example 2: Let us take as a specific example, deleting 3 from this B-tree (of degree 5):

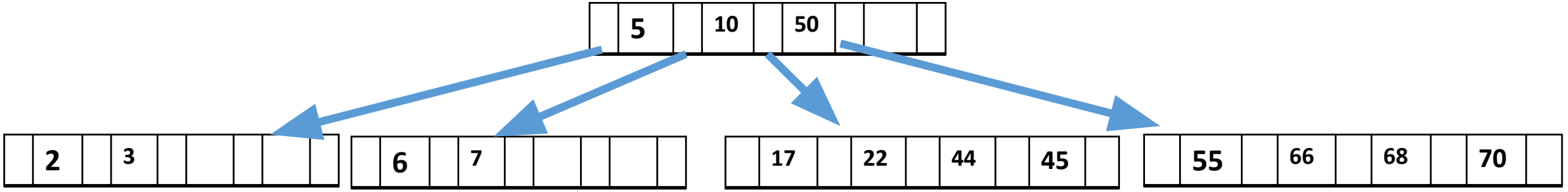
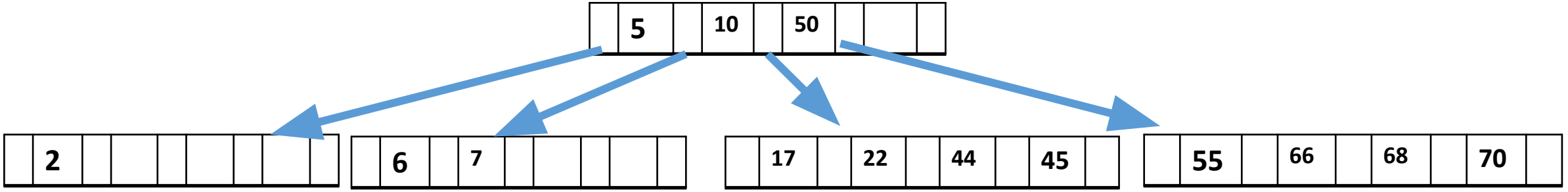
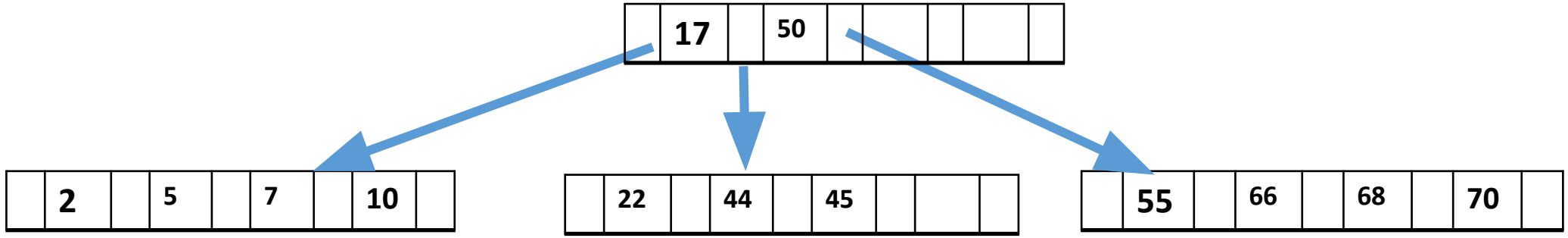


Fig.Before Deletion

Here 6 is in the leaf , so simply delete and observe the resultant.



Removing 3 causes the node it is in to underflow, as it now contains just 1 value (2). Our strategy for fixing this is to try to `borrow' values from a neighbouring node. However ,sib link too does not have ufficient number of elements .So , follow the combine strategy. The resultant tree is shown below:



Case 2:The element to be deleted is in a non leaf node

Case 2 is transformed into case 1 by replacing deleted element with either largest element in its left sub tree or smallest element in its right sub tree and then the replacing element is deleted from its original position. The replacing element is guaranteed to be in leaf.

Example :Delete 350 from below tree

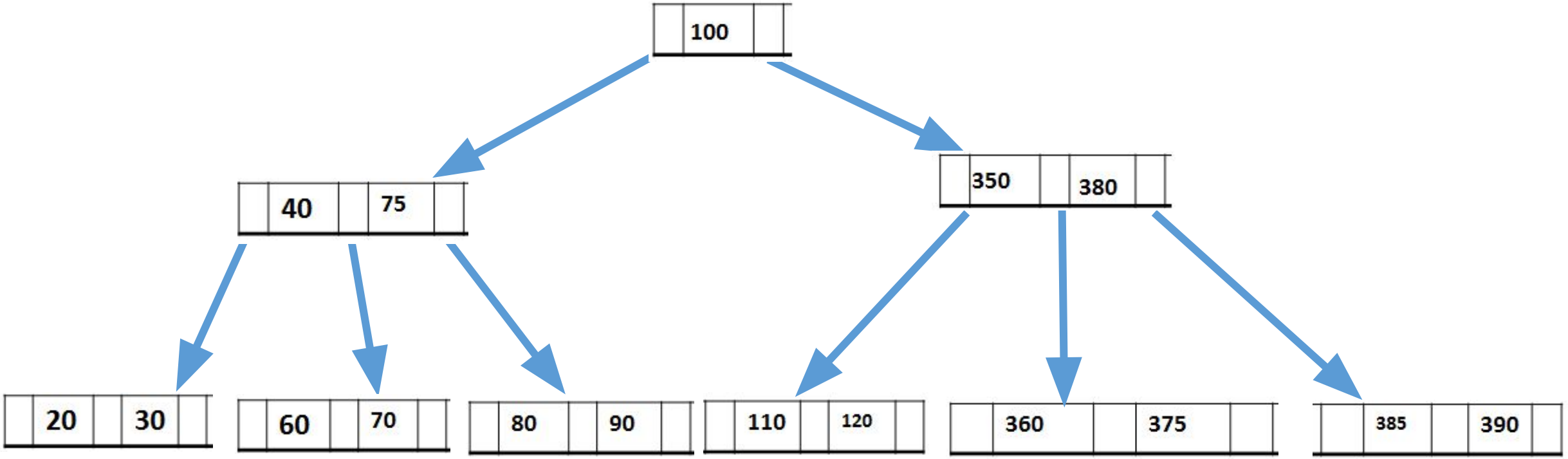


Fig a: Before deletion

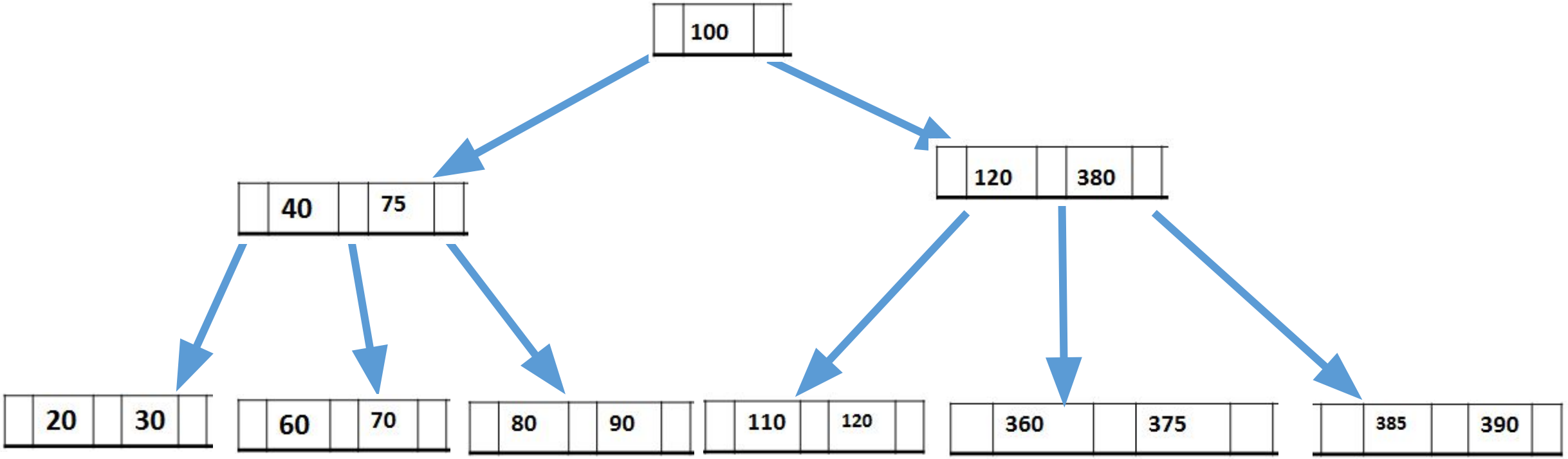


Fig a: After replacing 350 with largest element in its left subtree .

Now , delete 120 from its original position. The resultant tree is shown below

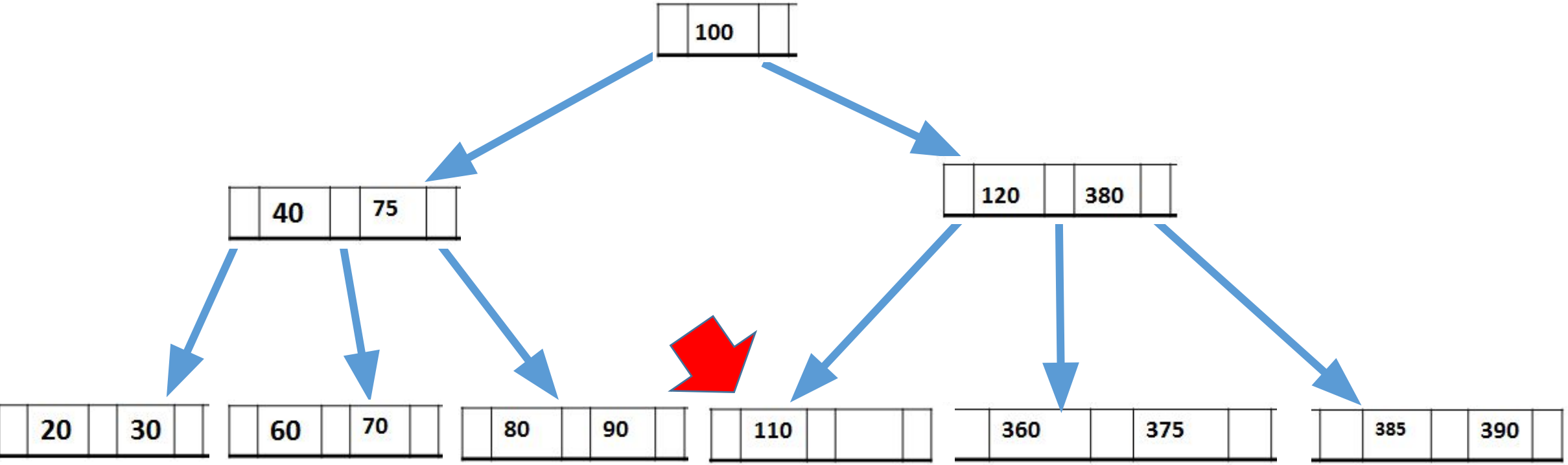


Fig b: After deleting 120 from its original position. Resultant tree is not B tree

Observe the previous tree. After deleting 120, the corresponding node suffers from underflow. So, borrow from neighbour. However neighbour does not have sufficient number of elements. Then the solution is combine sib links along with their parent. The resultant tree is shown next.

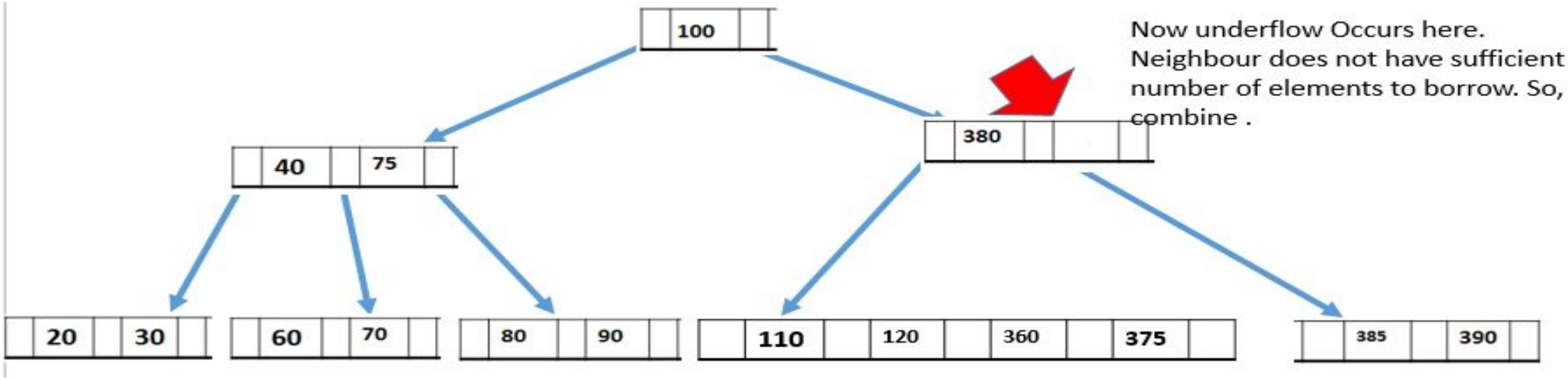


Fig c: After combining at leaf level

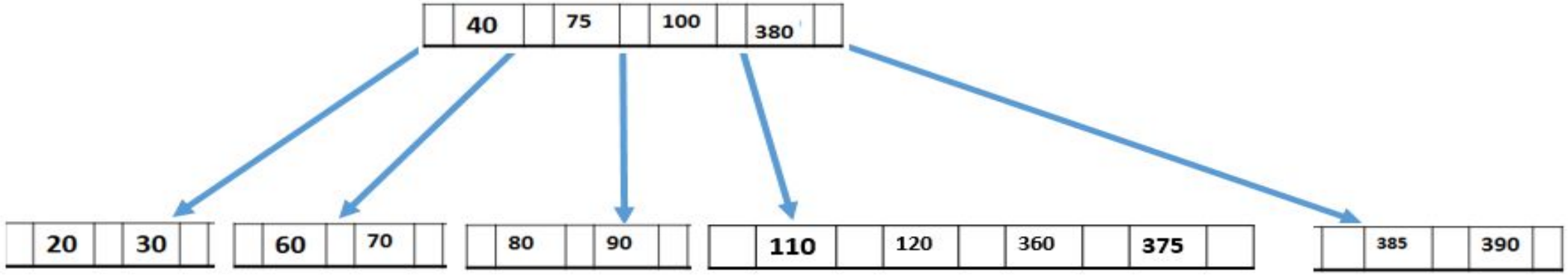


Fig d: After combining at 2nd level. Final tree

- Note 1: Refer Sahani& Good Rich text books to know the applications of various data structures
- Note 2: Refer Record for programs