Unit 5: Binary Tree

Trees

Linked list usually provide greater flexibility than array, but they are linear structures and it is difficult to use them to organize a hierarchical representation of objects. To overcome these limitations, we create a new data type called a tree that consists of nodes and arcs. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If t1,……tk are disjointed trees, then the structure whose root has its children the roots of t1,,…… tk is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

Key Terminologies

Root: A tree contains a unique first node which is shown at the top of the tree structure. This node is called root of the tree.

Leaf Nodes: The nodes which do not have children are called leaf nodes.

Interior Nodes: The nodes which have children nodes are called interior nodes.

Siblings: If two or more nodes have same parent, then these nodes are called siblings to each other.

Ancestor: A node is called ancestor of another node if either it is the parent of that node or it is the parent of some other ancestor of that node.

Descendents: A node is called Descendents of another node if it is the child of that node or the child of some other descendents of that node.

Depth of tree: The length of the longest path from root to any other node is known as the depth of the tree. Path is the number of edges from root to any node.

Binary Tree

A binary tree is type of tree with finite number of elements and is divided into three main parts. The first part is called root of the tree and other two parts are itself binary tree which exists towards left and right of the tree. Each of the elements in the binary tree is considered a node and a node will have three piece of information: data, two references

Some information of binary tree

- The binary tree starts at root and grows downward.
- The topmost node is called the root. The root will not have parents. All other nodes can be reached from it by following edges or links.
- The link between two nodes is termed as edge or arc.
- If X is the root of the tree and L and R are its left child/ and right child respectively then X is the father of both L and R. L and R are called siblings or brother
- A node with no left or right child is termed as leaf node. It is also called terminal node.
- An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.
- A path is sequence of edges from some node to another node with more than one edge.
- A path ending in a leaf is known as branch.
- Going from root of the tree to any of the leaf node is termed as descending the tree.
- Going from any leaf of the tree towards root of the tree is termed as climbing the tree.
- The root has level 0 and level of any other node is one more than the level of its parents.

- The height of the binary tree is one more than the number of levels. Another definition of height is the number of nodes in a branch of tree.
- The depth of the binary tree is the maximum level number.
- Number of sons for a node is considered as degree of that node.
- Two binary trees are said to be copies when they have the identical structures and identical nodes at all levels of tree.
- Two binary trees are said to be similar when they have the identical structures only.

Strictly Binary Tree

It is a binary tree with non-empty right and left sub trees. In other words, it is binary tree with every node N has either 0 or 2 tree. The strictly binary tree is also known as extended 2 Tree or simply 2 –tree. Sometimes nodes with 2 children's are known as internal nodes and nodes with 0 children are known as external nodes.

Complete Binary Tree

It is a special type of strictly binary tree where all the leaves of the tree reside at the same level. Using the depth we always say a complete binary tree of depth d where all leaves with be at level d

Binary Search Tree

The application of binary tree is searching and sorting. By enforcing certain rules on the values of the elements stored in a binary tree, it could be used to search and sort.

Binary search tree is a tree in which value of each node in the tree is greater than the value of node in its left (if exists) and it is less than the value in its right child (if it exists).

As name suggests, binary search tree (BST) is used for searching purpose. For every node N in the BST, the following property will be true.

- The data at left node will be smaller than data at node N
- The data at right node will be larger than data at node N

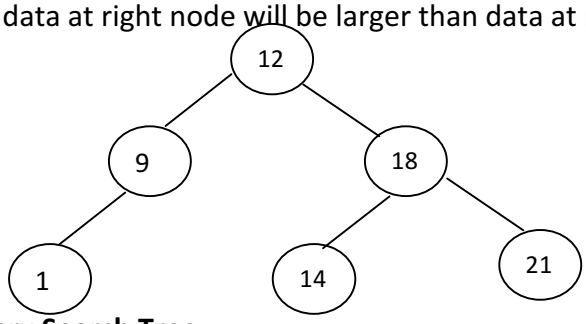


Figure: Binary Search Tree

Implementing Binary Trees

Binary trees can be implemented in at least two ways: as arrays and as linked structures. To implement a tree as an array, a node is declared as an object with information filed two "reference" fields.

Array Implementation of Binary Tree

The sequential representation of binary tree uses array for storing the data for each node. This is very simple. If any parent node is stored at index I then its left child will be stored at 2*I+1 and right child will be stored at 2*I+2. The root of the tree is stored at first index of the array (index 0).

The array representation of above BST is as shown below:

0	1	2	3	4	5	6	7	8	9
12	9	18	1		14	21			

It is clear that the most of the locations in array are empty. This causes wastage of memory space. That is, the array representation of binary tree is quite inefficient. In general for a binary tree with height H, the size of the array will be approximately be 2^H.

Besides this, locations of children must be known to insert a new node and these locations may need to be located sequentially. After deleting a node from a tree, a hole in the array would have to be eliminated; this may lead to populating the array with many unused cells.

Linked List Representation of Binary Tree

The linked list representation is the most popular, efficient and most frequently used representation of binary tree. In the linked list representation every node is represented by data, a reference to left child and a reference to right child. The node of binary tree can be created as follows:

```
class BTNode{
   public BTNode left;
   public int data;
   public BTNode right;
   public BTNode() {
      left = null;
      data = 0;
      right = null;
   }
   public BTNode(int n) {
      left = null;
       data = n;
      right = null;
   }
}
```

Searching a Binary Search Tree

An algorithm for locating an element in binary search tree is quite straightforward. For every node, compare the key to be located with the value stored in the node root. If the key is equal to value then stop. If the key is less than the value, go to the left subtree and try again. If key is greater than that value, try the right subtree. If it is same, obviously the search can be discontinued.

A method for searching a Binary Search Tree

```
Node BST( int el)
{
        Node p = root;
        while(p.el != el)
        {
                if(el < p.el)
                        p = p.left;
                else
                        p = p.right;
                if( p == null)
                        return null;
        }
        return p;
}
```

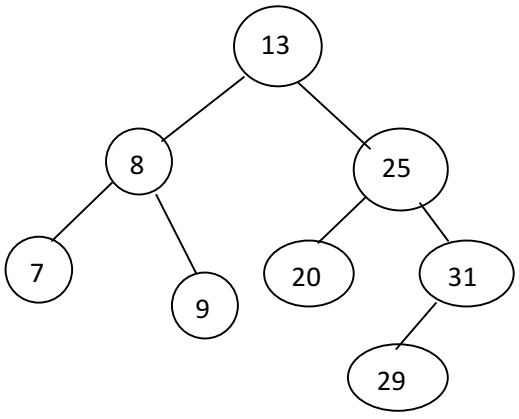
Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. Traversal may be interpreted as putting all nodes on one line or linearizing a tree.
The definition of traversal specifies only one condition-visiting each node only one time but it does not specify the order in which the nodes are visited. For a tree with n nodes, there are n! different traversals. Most of the traversal has no use, so we restrict our attention two classes only, namely, breadth first search and depth first search traversal.

Breadth First Traversal
Breadth first search traversal is visiting each node from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left). There are thus four possibilities, and one such possibility, a top – down, left to right, breadth first traversal of the tree.
Implementation of this kind of traversal is straightforward when a queue is used.
Consider a top-down, left to right, breadth first traversal. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited. The restriction is that all nodes on level n must be visited before visiting any node on level n+1 is accomplished.

```
Void breadthfirstTraversal()
{
        Node p = root;
        Queue queue = new Queue<Node>();
        if(p != null)
        {
                queue.enqueue(p);
                while(!queue.isempty())
                {
                        p = queue.dequeue();
                        visit(p);
                        if(p.left != null)
                                queue.enqueue(p.left);
                        if(p.right != null)
                                queue.enqueue(p.right);
                }
        }
}
```

For example, consider the following tree:



The Breadth first search traversal will result in the sequence: 13 8 25 7 9 20 31 29

Depth First Traversal

Depth first traversal proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right). We repeat this process until all nodes are visited. There are some variations of the depth first search traversal.

There are three tasks of interest in this types of traversal:

V – Visiting a node

L – Traversing the left subtree.

R – Traversing the right subtree.

Preorder Traversal(VLR):

To traverse a nonempty binary tree in **preorder**, we perform the following three operations:

- Visit the root
- Traverse the left tree in preorder
- Traverse the right sub tree in preorder

```
void preorder(Node p)
{
        if(p != null)
        {
                visit (p);
                preorder(p.leftChild);
                preorder(p.rightChild);
        }
}
```

Inorder Traversal(LVR):

To traverse a nonempty binary tree in **inorder**, we perform the following three operations:

- Traverse the left sub tree in inorder
- Visit the root
- Traverse the right sub tree inorder

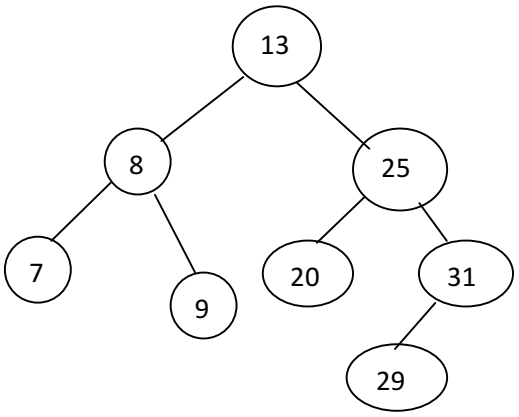
```
void inorder(Node p)
{
        if(p != null)
        {
                inorder(p.leftChild);
                visit (p);
                inorder(p.rightChild);
        }
}
```

Postorder Traversal(LRV)

To traverse a non-empty tree in **Postorder**, we perform the following

- Traverse the left sub tree in postoder
- Traverse the right sub tree in postorder
- Visit the root

```
void postorder(Node p)
{
        if(p != null)
        {
                postorder(p.leftChild);
                postorder(p.rightChild);
                visit (p);
        }
}
```



The preorder traversal will print: 13 8 7 9 25 20 31 29
The inorder traversal will print: 7 8 9 13 20 25 29 31
The postorder traversal will print: 7 9 8 20 29 31 25 30

Insertion

Searching a binary tree does not modify the tree. It scans the tree in a predetermined way to access some or all of the keys in the tree but the tree itself remains undisturbed. There are certain operations that always make some change on the tree, such as adding nodes, deleting nodes, and modifying elements, merging trees and balancing trees to reduce their height.

To insert a new node with key el, a tree node with a dead end has to be reached, and new node has to be attached to it. Such a tree node is found using the same technique as used in tree searching.

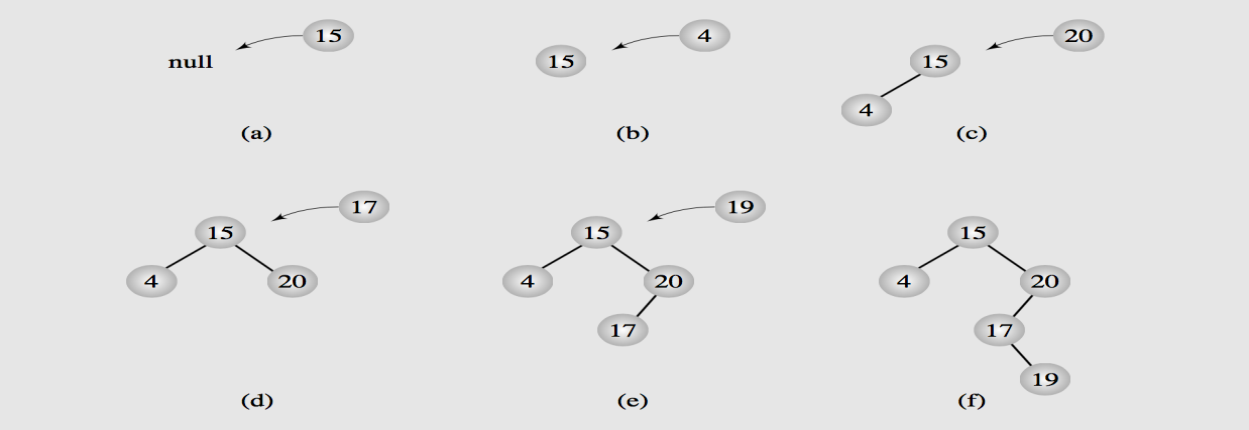
The key el is compared to the key of the node currently being examined during tree scan. If el is less than that key, the left child (if any) of p is tried; otherwise the right child is tested. If the child of p to be tested is empty, the scanning is discontinued and the new node becomes this child.

Insertion Algorithm:

If root is NULL
then create root node
return
If root exists
then compare the data with node.data
while until
insertion position is located
If data is greater than node.data
goto right subtree
else goto left subtree
endwhile
insert data
end If

Method to insert element in BST:

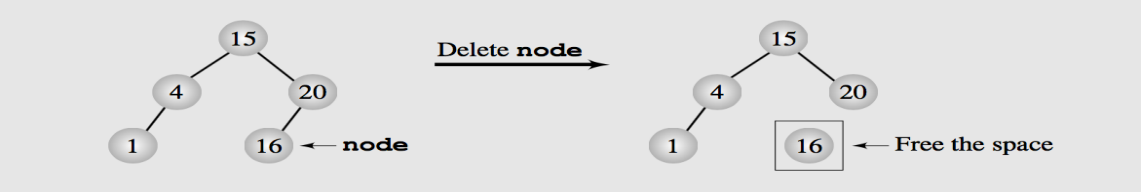
```
void insert(int el)
{

        Node p = root, prev = null;
        while(p != null)
        {
                        prev = p;
                        if(el.compareTo(p.el) < 0)
                                p = p.left;
                        else
                                p = p.right;
        }
        if(root == null)
                root = new Node(el);
        else if(el.compareTo(prev.el) < 0)
                prev.left = new Node(el);
        else
                prev.right = new Node(el);
}
```



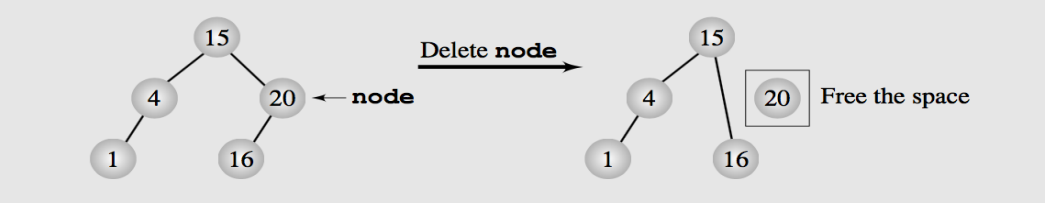
Deletion

Deleting a node is another operation to maintain a binary Search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree.

1. The node is a leaf; it has no children: This is the easiest case to deal with. The appropriate reference of its parent is set to null and space occupied the deleted node is later claimed by the garbage collector.



2. The node has one child: This case is not complicated. The parent's child reference to the node is reset to refer to the node's child. In this way, the node's children are lifted up be one level



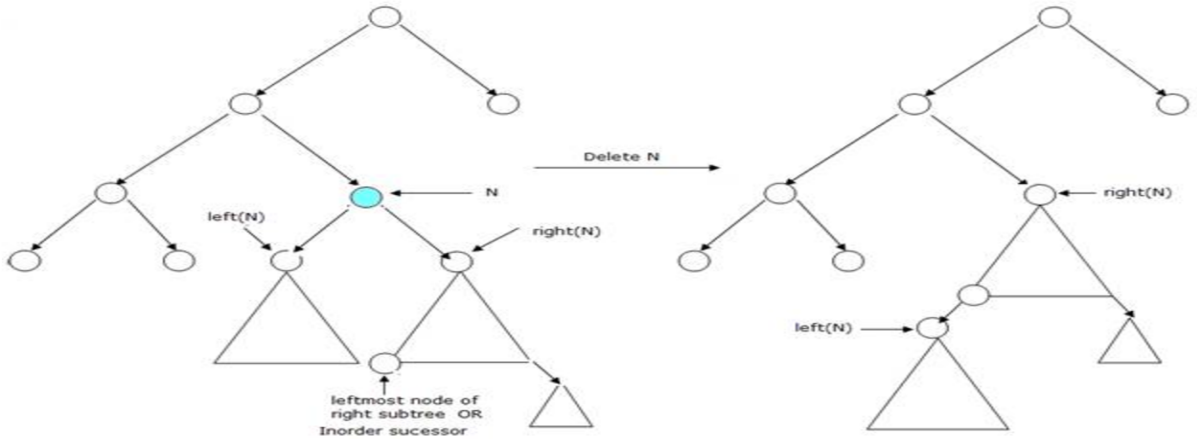
3. The node has two children. In this case, no one step option can be performed because the parent's right or left reference cannot refer to both the node's children at the same time.
   There are two solutions to this problem:

a) Deletion by merging

This solution makes one tree out of the two subtrees of the node and then attaches it to the node's parent. This technique is called deleting by merging. By nature of the binary search trees, every key of the right subtree is greater than every key of the left subtree, so the best thing to do is to find in left tree The node with the greatest key and make it a parent of the right subtree. Symmetrically, the node with the lowest key can be found in the right subtree and made a parent of the left subtree.

The desired node is the rightmost node of the left subtree. It can be located by moving along this subtree and taking right reference until null is encountered. This means that this node will not have a right child, and there is no danger of violating the property of binary search trees in the original tree by setting that right most node's right reference to right subtree.

Since in a binary search tree value at every node is always less than or equal to the values in right subtree and always greater than to the values in left subtree, to merge a right subtree into a left subtree we find largest value in left subtree and make it a parent of right subtree of node N.



Deleting node N with two children by merging right subtree into left subtree

To delete node 25 by merging we will merge right subtree into left subtree. First we have to find the node with largest key value (20) in left subtree. Now make the node 20 as a parent of right subtree of node 25. Next set the link of parent node of 25 which is node 35 to point to root of left subtree of 25.

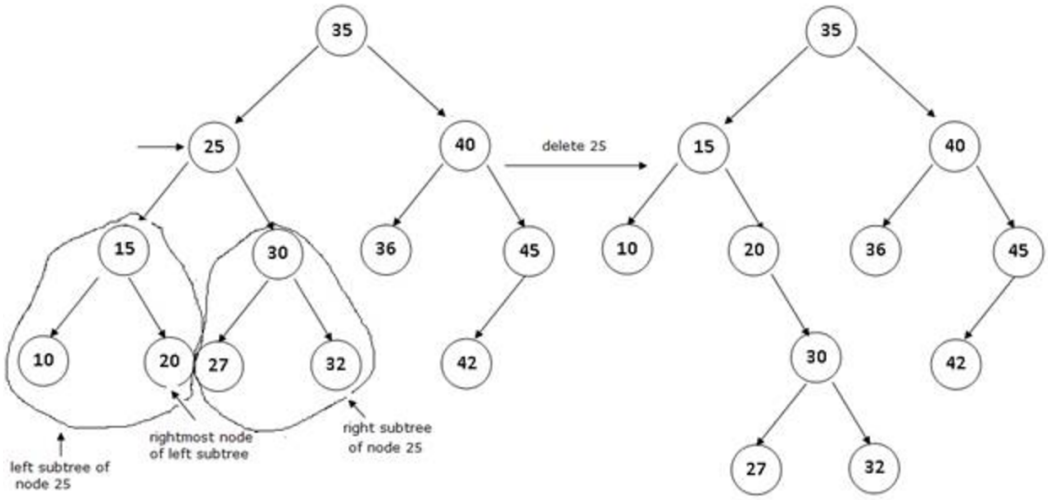
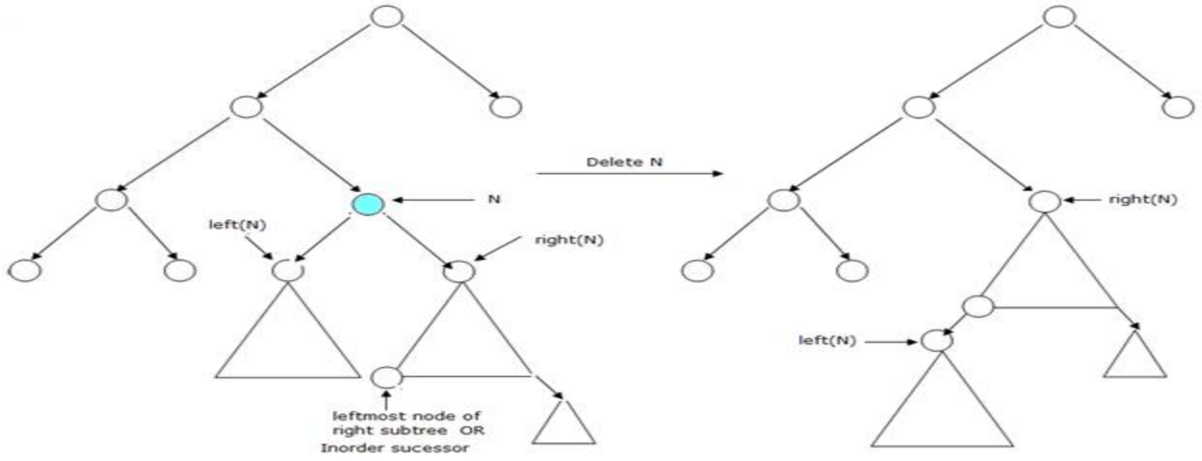


Figure 3.15 Deleting 25 by merging right subtree into left subtree



Deleting node N with two children by merging left subtree into right subtree

In order to merge left subtree into right subtree find the smallest value in right subtree and make it a parent of left subtree. The smallest value in a right subtree will be the leftmost node i.e. node with no left child. This can be located by moving down in right subtree towards left until left child is NULL. This node is the immediate successor of the node to be deleted in inorder traversal.

Similarly, node 25 can also be deleted by merging left subtree into right subtree and link right subtree to the parent node of 25. This has been illustrated in figure 3.16.

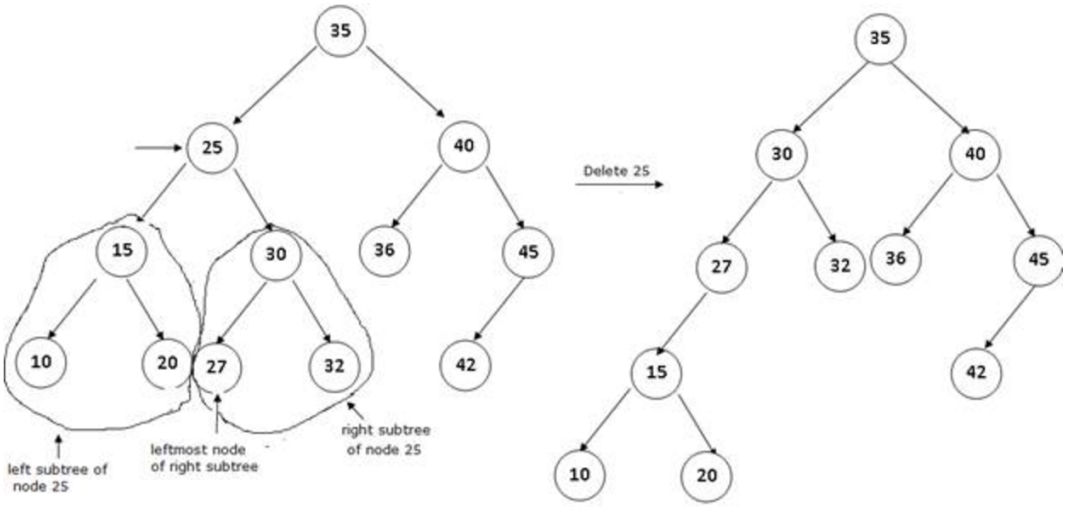


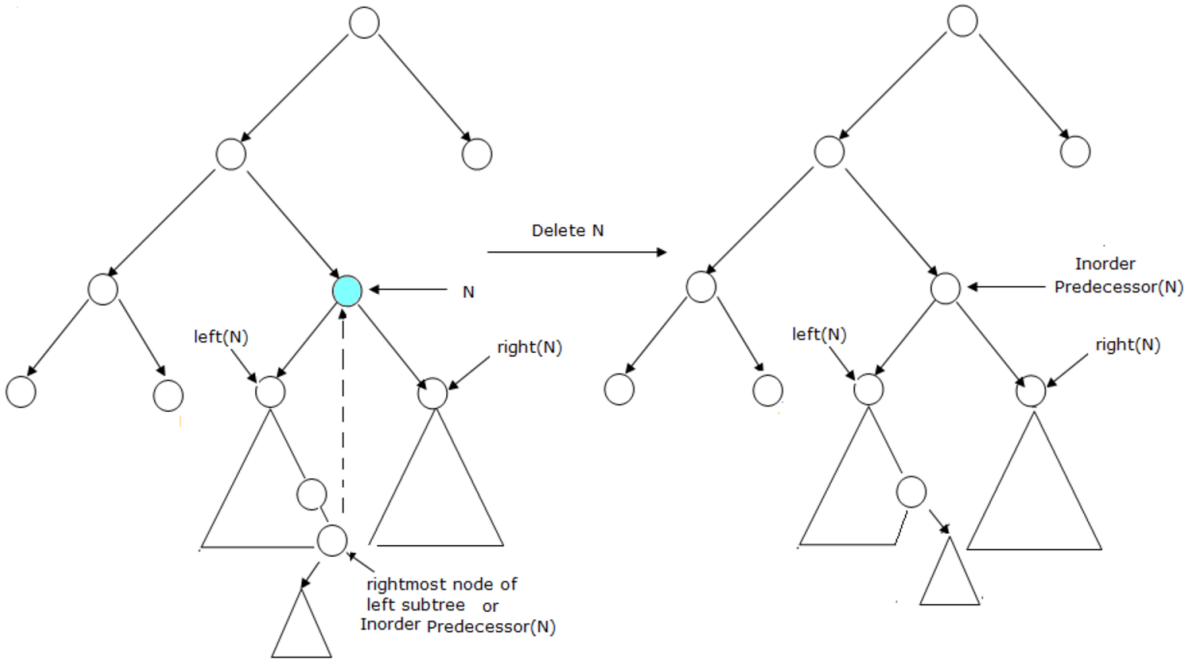
Figure 3.16 Deleting 25 by merging left subtree into right subtree

b) **Deletion by Copying**

Another solution, called deletion by copying, was proposed by Thomas Hibbard and Donald Knuth. If the node has two children, the problem can be reduced to one of two simple cases. The node is a leaf or the node has only one nonempty child. This can be done by replacing the key being deleted with its immediate predecessor (or successor). As deletion by merging, a key's predecessor is the key in the rightmost node in the left subtree. First, the predecessor has to be located. This is done, again, by moving as far to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible. Next, the key of the located node replaces the key to be deleted. And that is where one of two simple cases comes into play. If the rightmost node is a leaf, the first case applies; however, if it has one child, the second case is relevant. In this way, deletion by copying removes a key k1 by overwriting it by another key k2 and then removing the node that holds k2, whereas deletion by merging consisted of removing a key k1 along with the node that holds it.

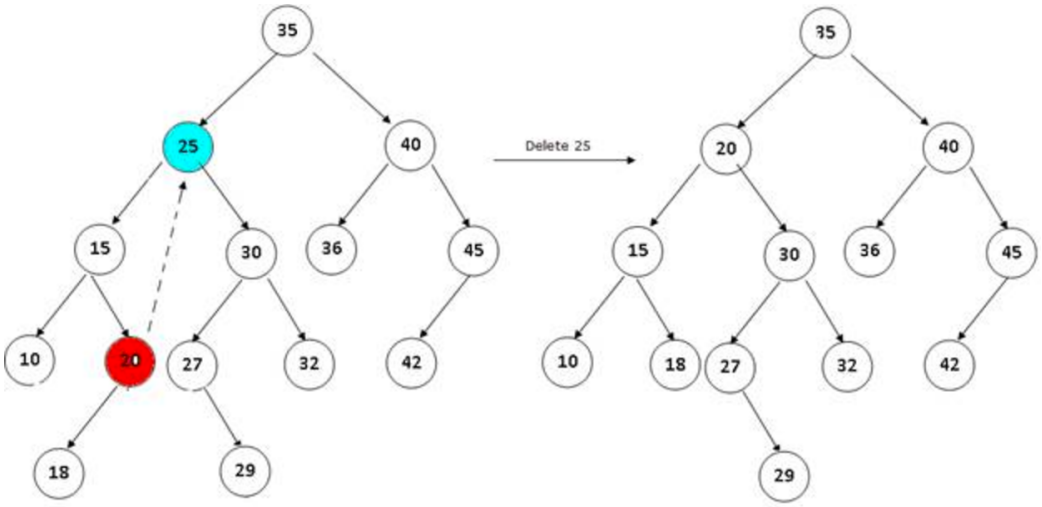
Deletion by copying method is simple and better than the deletion by merging method. Sometimes height of the tree increases in deletion by merging resulting in unbalanced tree. But in Deletion by Copying method height of the tree remains balanced.

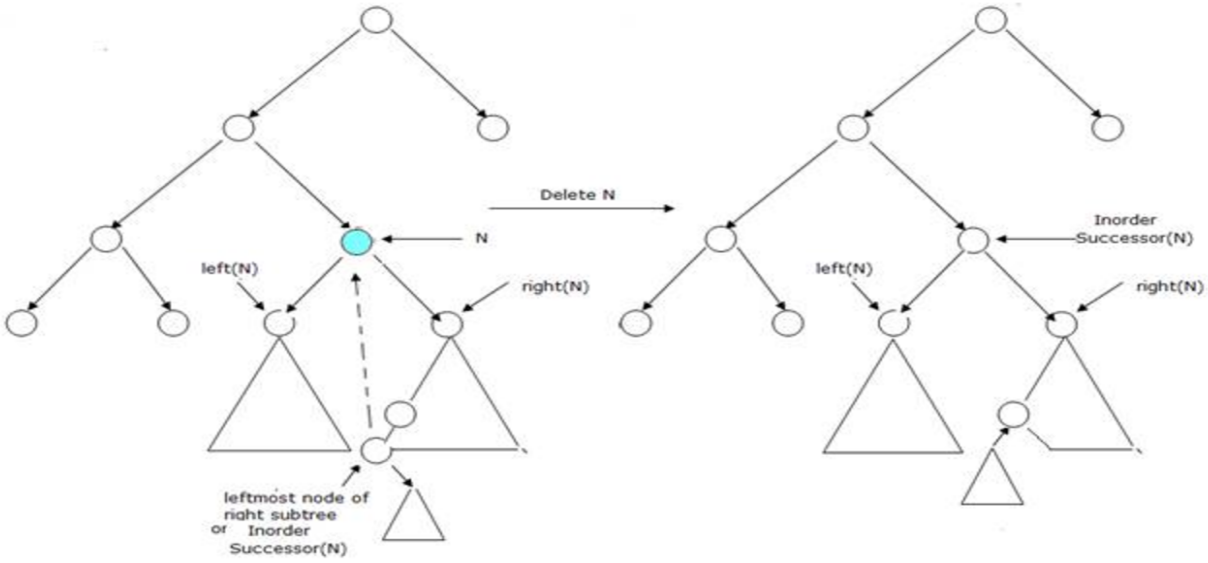
In deletion by copying key of inorder predecessor/successor of the node to be deleted are copied at the node's place. First we find inorder predecessor/r successor of the node as discussed in previous section. Than copy the value of Inorder predecessor/successor at the place of key value of the node and then delete the inorder predecessor/successor



Deleting N by copying In-order Predecessor

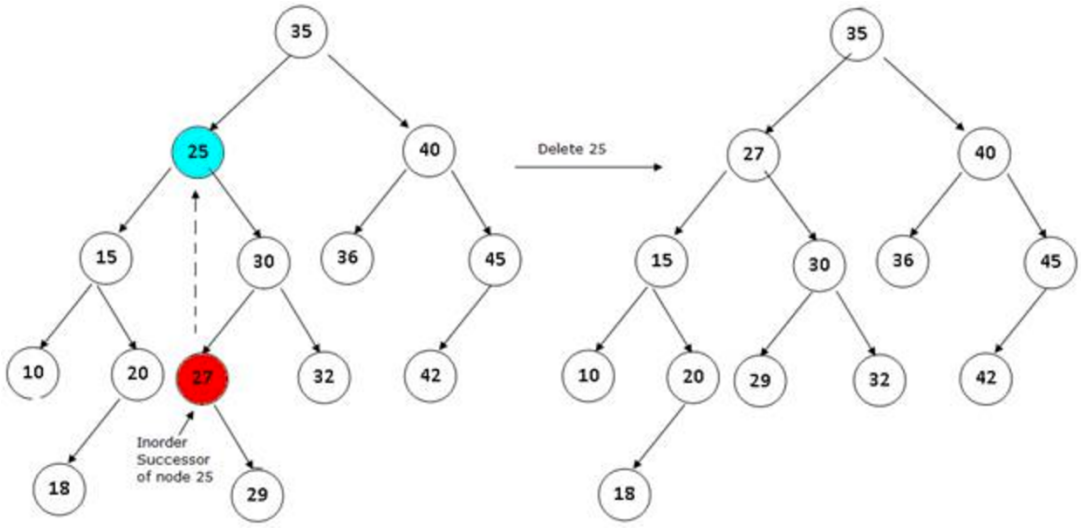
Deleting 25 using Deletion by copying inorder predecessor





Deleting N by copying Inorder Successor

Deleting node 25 using Deletion by copying inorder successor



Balancing a Tree

There are two arguments that support the use of tree. The first one is tree is used to represent the hierarchical structure of a certain domain and search process is much faster is using trees than using linked lists.

The second argument, however, does not always hold. It all depends on what the tree looks like. If tree is not balanced tree it is more or less similar to linked list.

A binary tree is height balanced tree or simply balanced tree if the difference in height of both subtrees of any node in the tree is either zero or one.

Let us consider the following table.

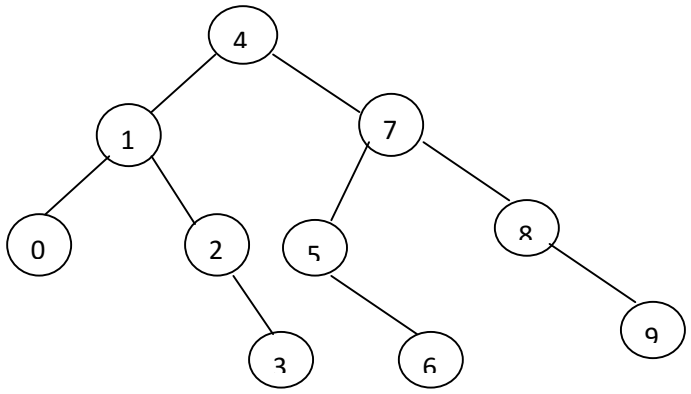
Height	Nodes at one Level	Nodes at All Levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 1$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
….		
11	$2^{10} = 1024$	$2027 = 2^{11} - 1$
…		
14	$2^{13} = 8192$	$16383 = 2^{14} - 1$
…		
H	2^{h-1}	$N = 2^h - 1$

For example, if 10000 elements are stored in a perfectly balanced tree, then the tree is of height log 10000 = 13.289 = 14. In practical terms, this means that if 10000 elements are stored in a perfectly balanced tree, then at most 14 nodes have to be checked to locate a particular element. This is a

substantial difference compared to 10000 tests needed in a linked list (in the worst case). Therefore, it is worth the effort to build a balanced tree or modify an existing tree so that it is balanced.
There are number of techniques to properly balance a binary tree. Some of them consist of constantly restructuring the tree when elements arrive and lead to an unbalanced tree. Some of them consist of reordering the data themselves and then building a tree.
The technique is based on binary search technique. This allows for using the following simple recursive implementation.

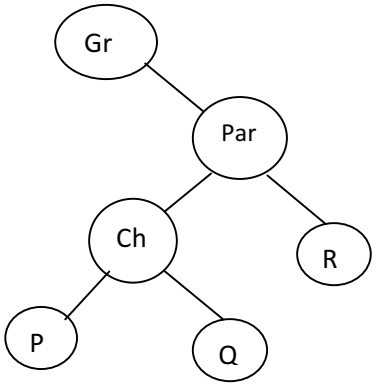
```
void balance (data[] int first, int last){
if(first<=last){
int middle = (first+last)/2;
insert(data[middle]);
balance(data, first, middle-1);
balance(data, middle+1, last);
}
}
```

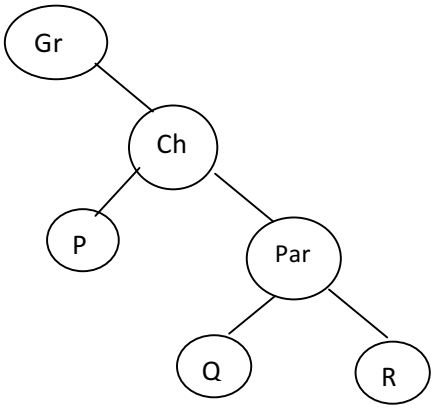
Let us consider the following example:
Stream of data: 5 1 9 8 7 0 2 3 4 6
Array of sorted data: 0 **1** 2 3 **4** 5 6 7 8 9



The DSL Algorithm
The earlier method is no efficient in that it requires an array and needs array to be sorted. To avoid sorting, it required deconstructing the tree after placing elements in the array using the inorder traversal, and then reconstructing the tree, which is inefficient except for relatively small tree. There are, however, algorithms that require little additional storage for intermediate variables and use no sorting procedures. The very elegant DSW algorithm was devised by Colin day and later improved by block Quentin F, Stout
The building block for tree transformation is this algorithm is the rotation. There are two of rotations, left and right, which are symmetrical to each other. The right rotation of the node **ch** about its parent **par** is performed according to the following algorithm:

```
rotateRight(Gr, Par, Ch)
if  Par is not the root of the tree
grandparent Gr of child Ch becomes Ch's parent
right subtree of Ch becomes left subtree of Ch's parent Par;
node Ch acquires Par as its right child;
```

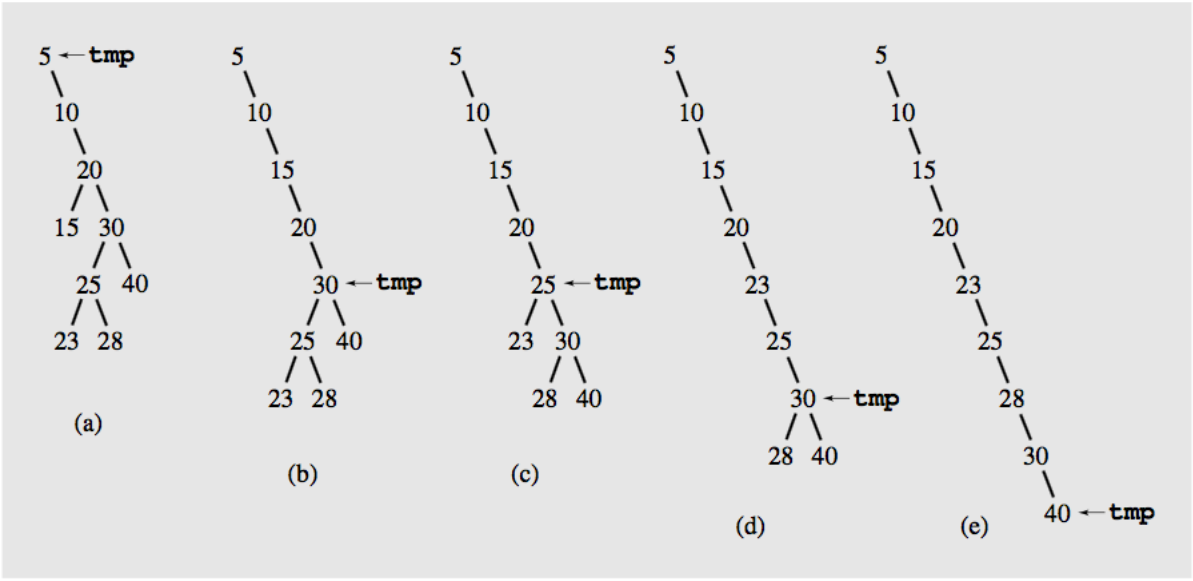




Basically, the DSW algorithm transforms an arbitrary binary search tree into a linked list called a backbone or vine. Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent
In the first phase, a backbone is created using the following routine.

```
CreateBackbone (root, n)
tmp = root;
while (tmp!=null)
if(tmp has a left child
        rotate this about tmp;
set tmp to the child that just became parent;
else set tmp to its right child;
```

FIGURE 6.38 Transforming a binary search tree into a backbone.

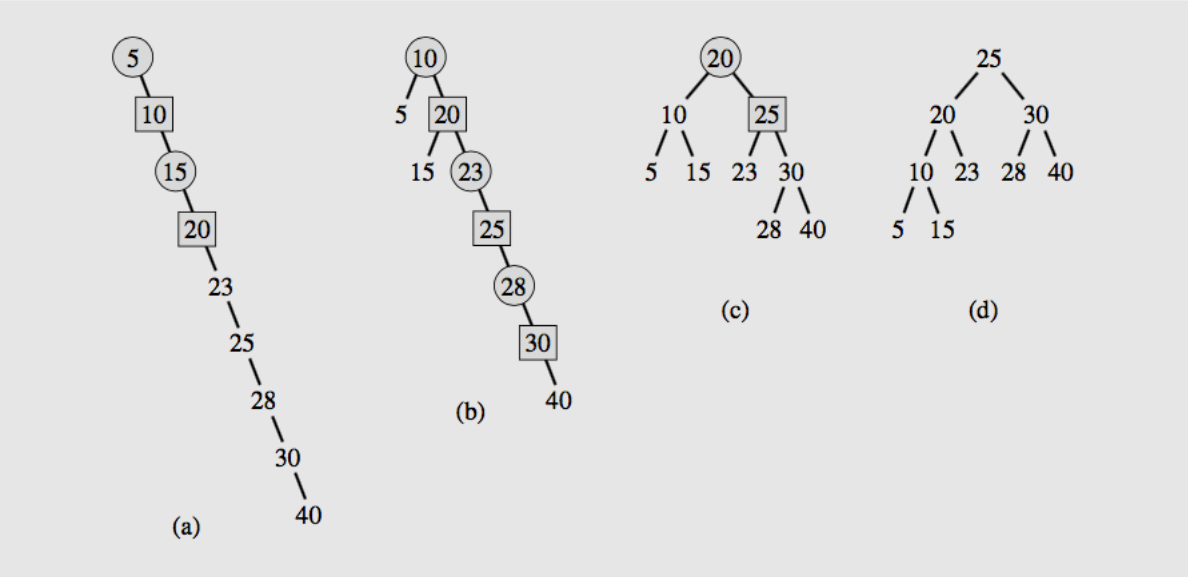


In the best case, when the tree is already a backbone, the while loop is executed n times and no rotation is performed. In worst case, when the root does not have a right child, the while loop is executed 2n-1 times with n-1 rotations performed, where n is the number of nodes in the tree; that is, the run time of the first phase is O(n).
In the second phase, the backbone is transformed into a tree, but this time the tree is perfectly balanced by having leaves only on two adjacent leaves. In each pass down the backbone, every second node down to a certain point is routed about its parent. The first pass may not reach the end of the backbone. It is used to account for the difference between the number n of nodes in the current tree and the number $2^{\lfloor \log(n+1) \rfloor}-1$ of nodes in the closest complete binary tree.
That is, the overflowing nodes are repeated separately.

createPerfectTree(n)
m = $2^{\lfloor \log(n+1) \rfloor}$ -1;
make n-m rotations starting from the top of the backbone;
while(m>1)
m = m/2;
make m rotations staring from the top of the backbone;

To compute the complexity of the tree building phase, observe that the number of iterations performed by the while loop equals.
$(2^{\log(m+1)-1}-1)+\ldots\ldots\ldots\ldots\ldots\ldots+15+7+3+1 = m - \log(m+1)$

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



The number of rotations can now be given by the formula
n-m+(m-log (m+1)) = n-log(m+1) = n- $\lfloor \log(n+1 \rfloor$

that is, the number of rotations is O(n). Because creating a backbone also requires at most O(n) rotations, the cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of additional storage

The java method for DSW is as follows

```
public void DSW(){
        if(root!=null){
        createBackBone();
        createPerfectBST();
        }
    }
    public void createBackBone(){
        BSTNode grandParent = null;
        BSTNode parent = root;
        BSTNode leftChild;
        while(parent!=null){
            leftChild = parent.left;
            if(leftChild!=null){
                grandParent = rotateRight(grandParent,parent,leftChild);
               parent = leftChild;
            }
            else{
                grandParent = parent;
                parent = parent.right;
            }
        }
    }
    public BSTNode rotateRight(BSTNode grandParent, BSTNode parent, BSTNode leftChild){
        if(grandParent!=null){
            grandParent.right = leftChild;
        }
        else{
            root = leftChild;
        }
        parent.left = leftChild.right;
        leftChild.right = parent;
        return grandParent;
}
 private void createPerfectBST() {
int n = 0;
for (BSTNode tmp = root; null != tmp; tmp = tmp.right) {
n++;
}
int m = greatestPowerOf2LessThanN(n + 1) - 1;
makeRotations(n - m);
while (m > 1) {
makeRotations(m /= 2);
}
}
private int greatestPowerOf2LessThanN(int n) {
int x = MSB(n);
```

```
return (1 << x);
}
public int MSB(int n) {
int ndx = 0;
while (1 < n) {
n = (n >> 1);
ndx++;
}
return ndx;
}
private void makeRotations(int bound) {
BSTNode grandParent = null;
BSTNode parent = root;
BSTNode child = root.right;
for (; bound > 0; bound--) {
try {
if (null != child) {
rotateLeft(grandParent, parent, child);
grandParent = child;
parent = grandParent.right;
child = parent.right;
}
else {
break;
}
}
catch (NullPointerException convenient) {
break;
}
}
}
private void rotateLeft(BSTNode grandParent, BSTNode parent, BSTNode rightChild) {
if (null != grandParent) {
grandParent.right = rightChild;
} else {
root = rightChild;
}
parent.right = rightChild.left;
rightChild.left = parent;
}
```

AVL Trees

The AVL tree was proposed by **Adelson Velskii and Landies**. An AVL tree (originally called an admissible tree) is one in which the height of the left and right subtrees of every node differ by at most one. The definition of AVL tree is same as definition of the balanced tree. However, the concept of the AVL tree always implicitly includes the techniques for balancing the tree. The technique for balancing AVL trees does not guarantee that the resulting tree is perfectly balanced.

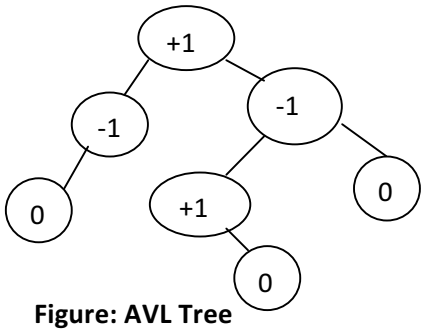
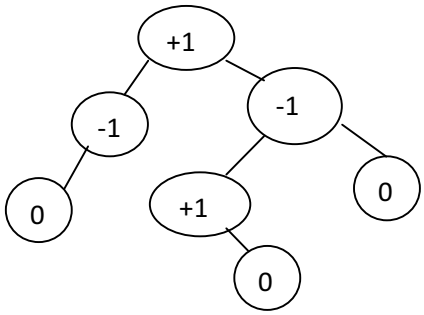


Figure: AVL Tree

Numbers in the nodes indicate the balance factors that are difference between the heights of the right subtree minus the height of the left subtree. For an AVL tree, all balance factors should be +1, 0 or -1.

The definition of an AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation.

$AVL_h = AVL_{h-1}+AVL_{h-2}+1$



Where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions. This formula leads to the following bounds on the height h of an AVL tree depending on the number of nodes n.

$\text{Log}(n+1) \leq h \leq 1.44(n+2)-0.328$

Therefore, h is bounded by O(logn); the worst case search requires O(logn) comparisons. For a perfectly balanced binary tree of the same height. Therefore, the search time in the worst case in an AVL tree is 44% worse than in the best tree configuration. Empirical studies indicate that the average number of searches is much closer to the best case than to the worst and is equal to logn+0.25 for large n. Therefore, AVL trees are definitely worth studying.

If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced.

Deletion may be more time consuming than insertion. First, we can apply deleteBycopying () to delete. This technique allows us to reduce the problem of deleting a node with two descendents to deleting a node with at most one descendents.

After a node has been deleted from the tree, balance factor are updated from the parent of the deleted node up to the root. For each node in this path whose balance factor becomes ±2, a single or double

rotation has to be performed to restore the balance of the tree. Importantly, the rebalancing does not stop after the first node is found for which the balance factor would become ±2, as is the case with insertion. This also means that deletion leads to at most O(logn) rotations, because in the worst case, every node on the path from the deleted node to the root may require rebalancing.

FIGURE 6.41 Balancing a tree after insertion of a node in the right subtree of node *Q*.

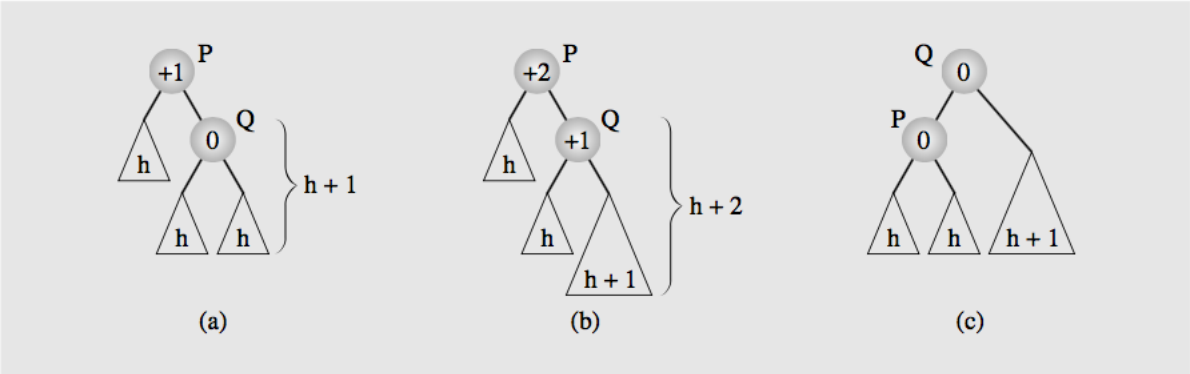


FIGURE 6.42 Balancing a tree after insertion of a node in the left subtree of node *Q*.

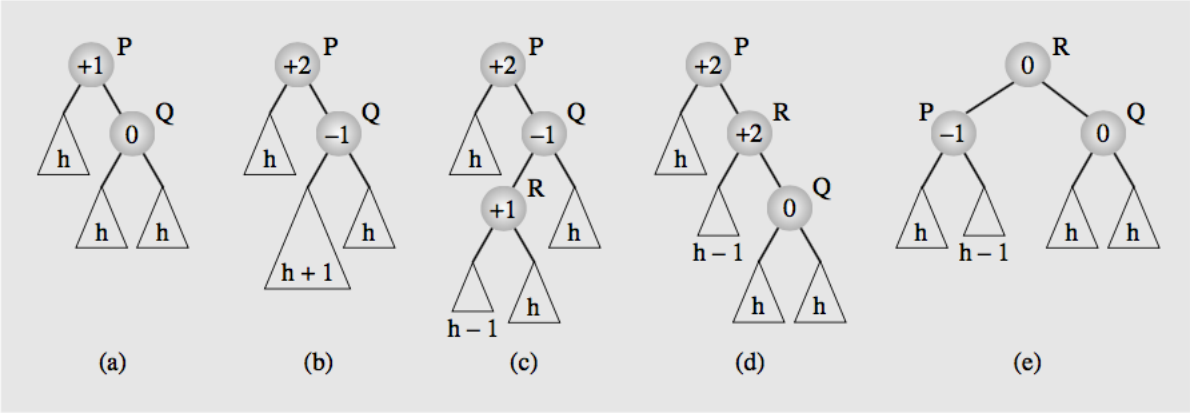


FIGURE 6.43 An example of inserting a new node (b) in an AVL tree (a), which requires one rotation (c) to restore the height balance.

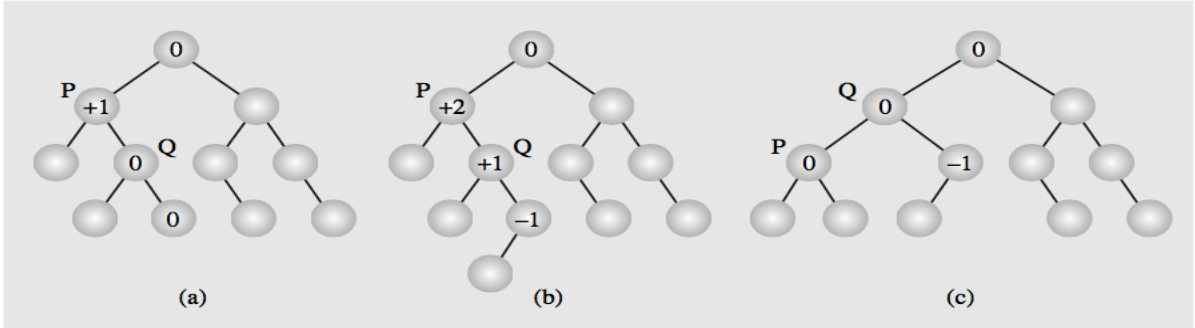


FIGURE 6.44 In an AVL tree (a) a new node is inserted (b) requiring no height adjustments.

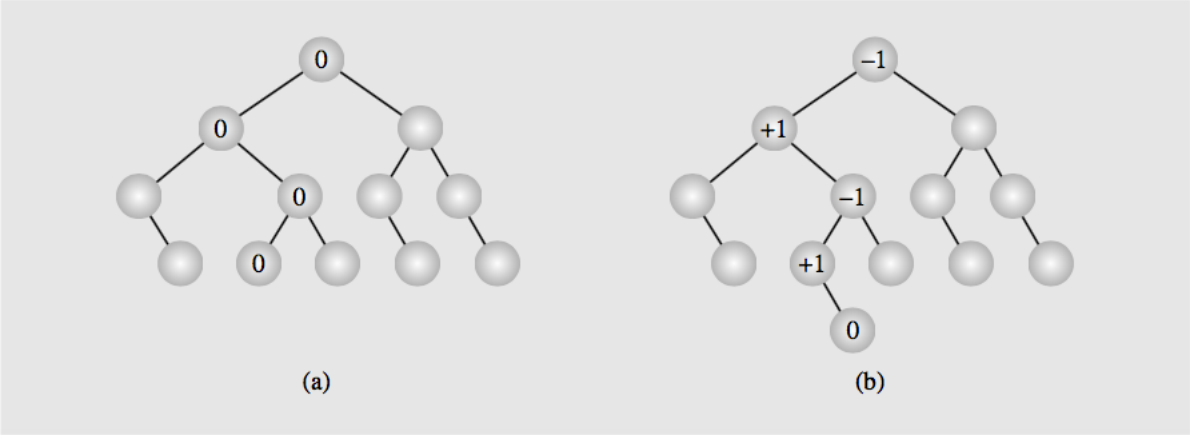
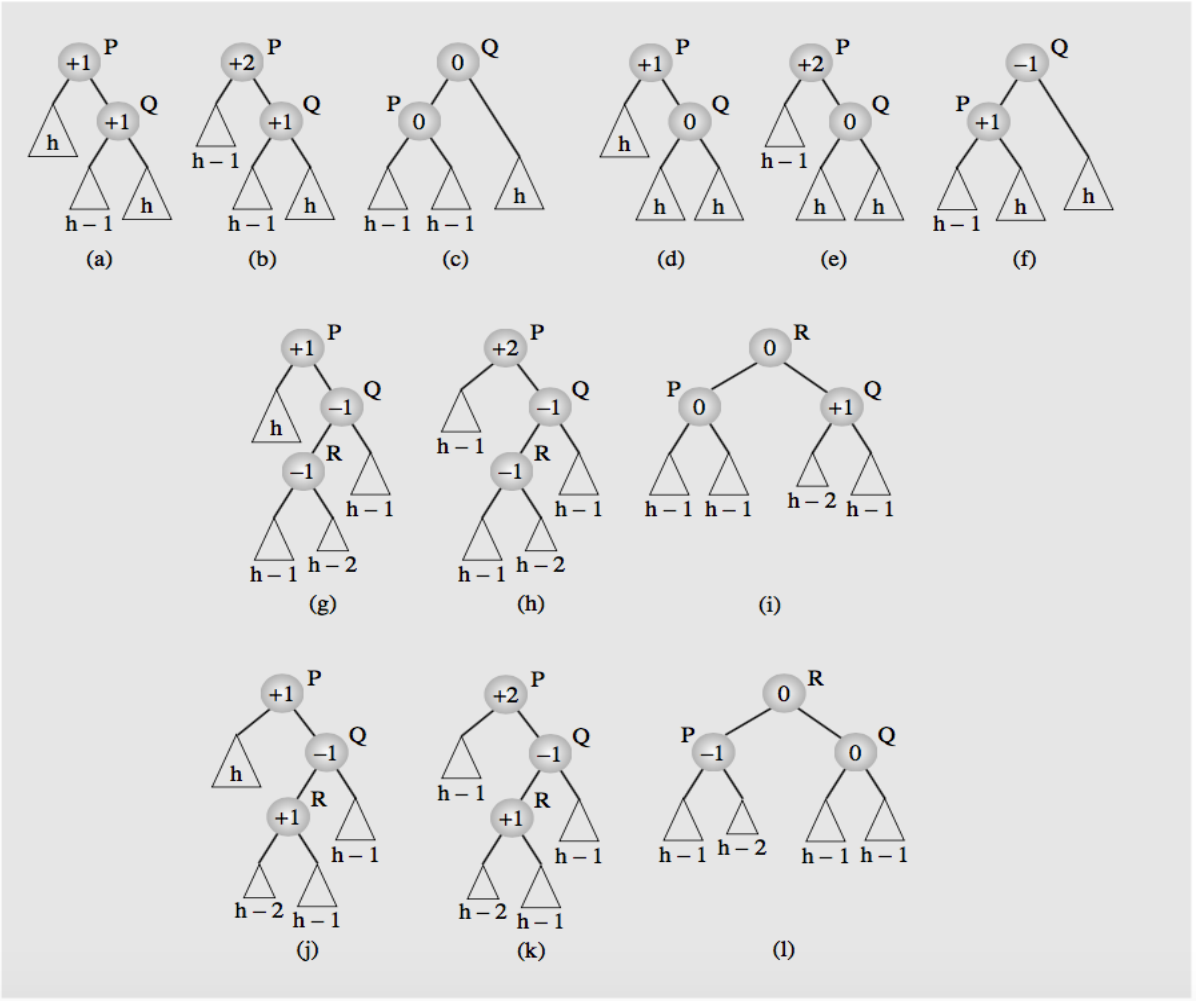


FIGURE 6.45 Rebalancing an AVL tree after deleting a node.



Self-Adjusting Trees

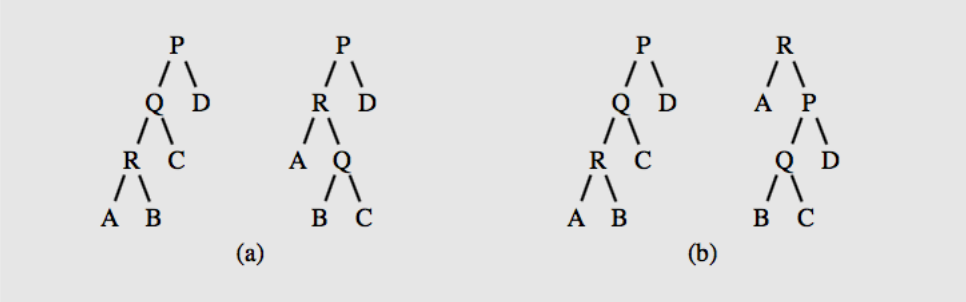
The main concern in balancing the tree is to keep them from becoming lopsided and ideally, to allow leaves to occur only at one or two levels. Therefore, if a newly arriving element endangers the tree balance, the problem is immediately rectified by restructuring the tree locally (AVL tree) or re-creating the tree). However, we may question whether such a restructuring is always necessary. Binary search tree is used to insert, retrieve and delete elements quickly, and the speed of performing these operations is the issue, but not the shape of the tree. Performance can be improved by balancing the tree, but this, is not the only method that can be used.

Another approach begins with the observation that not all the elements are used with the same frequency. For example, if an element on the 10^{th} level of the tree is used only infrequently, then the execution of the entire program is not greatly impaired by accessing this level. However, if the same element is constantly being accessed, then it makes a big difference whether it is on the tenth level or close to root. Therefore, the strategy in self- adjusting tree is to restructure trees by moving up the tree only those elements that are used more often, creating a kind of "priority tree". The frequency of accessing nodes can be determined in a variety of ways. Each node can have a counter field that records the number of times the element has been used for any operation. Then the tree can be scanned to move the most frequently accessed elements towards the root. In a less sophisticated approach, it is assumed that an element being accessed has good chance of being accessed again soon. Therefore, it is moved up to promoting elements that occasionally accessed, but the overall tendency is to move up elements with a higher frequency of access, and for the most part, these elements will populate the first few levels of the tree.

Self-Restructuring Trees

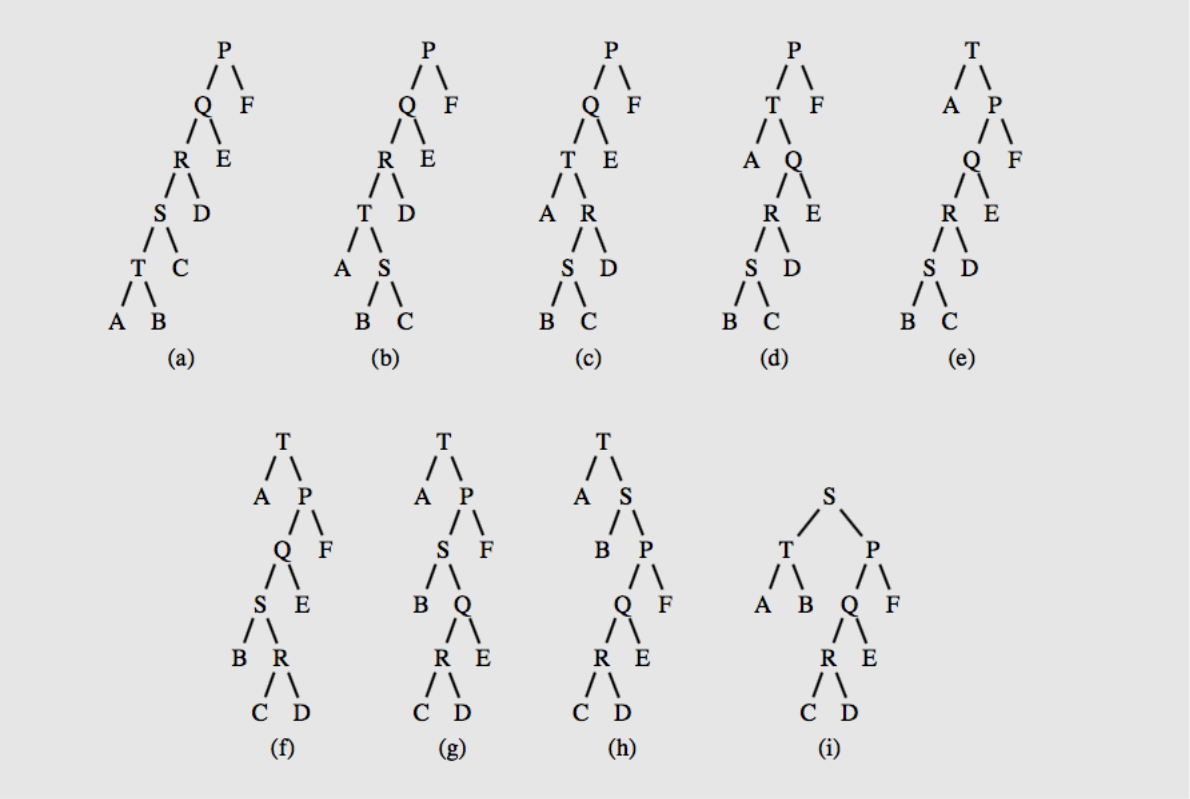
A strategy proposed by Brain Allen and Ian Munro and by James Bitner consists of two possibilities:

1. **Single rotation:** Rotate a child about its parent if an element in a child is accessed, unless it is the root. Figure(a).
2. **Moving to the root:** Repeat the child –parent rotation until the element being accessed is in the root. Figure(b).



Using the single rotation strategy, frequently accessed elements are eventually moved up close to the root so that later accesses are faster than previous ones. In the move to the root strategy, it is assumed that the element being accessed has a high probability to be accessed again, so it percolates right away up to the root. These strategies, however, do not work very well in unfavorable situations, when the binary tree is elongated. In this case, the shape of the tree improves slowly. Nevertheless, it has been determined that the cost of moving a node to the root, converges to the cost of accessing the nodes in an optimal tree times 2log 2; that is, it converges to (2ln2)logn. The result holds for any probability distribution. However, the average search time when all requests are equally likely is, for the single rotation technique, equal to $\sqrt{\pi n}$.

FIGURE 6.47 (a–e) Moving element *T* to the root and then (e–i) moving element *S* to the root.



Splaying

A modification of the move to the root is called splaying, which applies single rotation in pairs in an order depending on the links between the child, parent, and grandparent. First, three cases are distinguished depending on the relationship between a node R being accessed and its parent Q and grandparent P (if any) nodes:

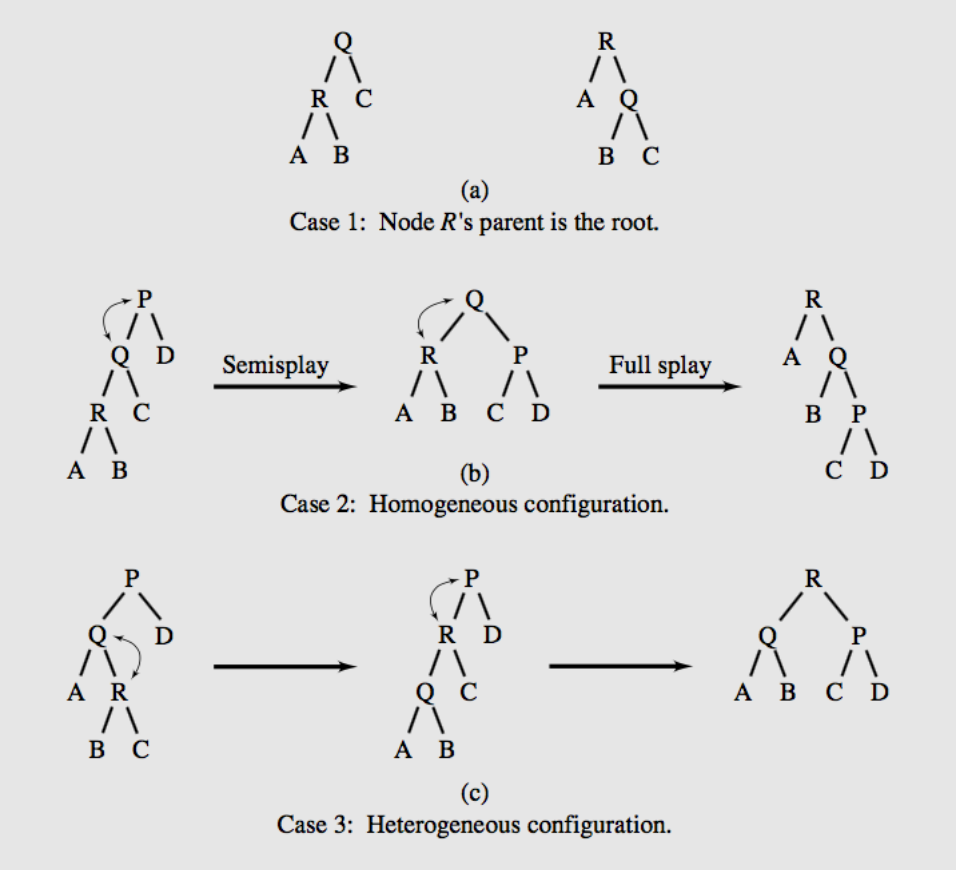
Case 1: Node R's parent is the root.

Case 2: Homogeneous configuration: Node R is the left child of its parent Q, and Q is the left child of its parent P or R and Q are both right children.

Case 3: Heterogeneous configuration: Node R is the right child of its parent Q, and Q is the left child of its parent P, or R is the left child of Q and Q is the right child of P.

The algorithm to move to a node R being accessed to the toot of the tree is as follows:

```
Spalying (P,Q,R)
while R is not the root
        If R's parent is the root
                Perform a singular splay, rotate R about its parent.
        else if R is in a homogeneous configuration with its predecessor
                perform a homogeneous splay, first rotate Q about P
                and then R about Q.
        else
                perform a heterogeneous splay; first rotate R about Q
                and then about P
```



Although splaying is a combination of two rotations except when next to the root, these rotations are not always used in the bottom up fashion as in the self-adjusting trees. For the homogeneous case (left –left or right –right), first the parent and the grandparent of the node being accessed are rotated, and only afterward are the node and its parent rotated. This has the effect of moving an element to the root and flattening the tree, which has a positive impact on the accesses to be made.

The number of rotations may seem excessive, and it certainly would be if an accessed element happened to be in a leaf every time. In case of a leaf, the access time is usually O(logn), except for some initial accesses when the tree is not balanced. But accessing elements close to root may make the tree unbalanced. For example, if the left child of the root is always accessed, then eventually, the tree would also be elongated, this time extending to the right.

Heaps

A particular kind of binary tree, called a heap, has the following two properties.

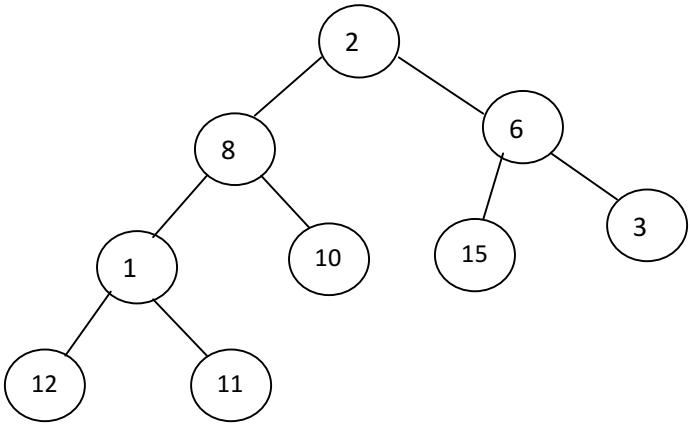
1. The value of each node is greater than or equal to the values of stored in each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

To be exact, these two properties define a max heap. If “greater” in the first properties is replaced with “less” then the definition specifies a min heap. This means that the root of a max heap contains the largest element, whereas the root of a min heap contains the smallest. A tree has the heap property if each nonleaf has the first property. Due to the second condition, the number of leaves in the tree is O(logn).

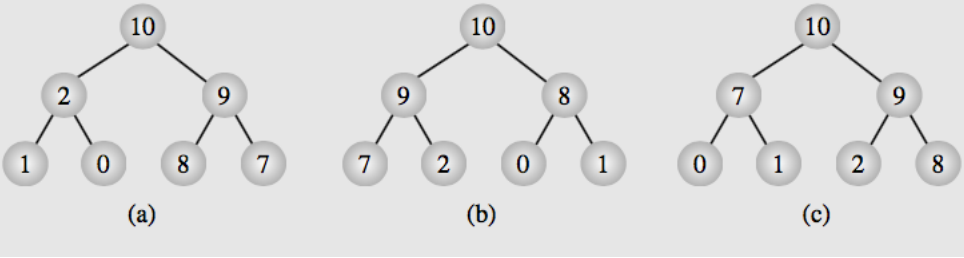
Interestingly, heaps can be implemented by arrays. The elements are placed at sequential locations representing the nodes from top to bottom and in each level from left to right. The second property

reflects the fact that the array is packed, with no gaps. Now, a heap can be defined as an array heap of length n in which:
Heap[i]≥heap[2.i+1], for 0≤i<(n-1)/2
And heap[i] ≥heap[2.i+1], for 0≤i<(n-2)/2.

The array [2 8 6 1 10 15 3 12 11] seen as a tree.



Elements in a heap are not perfectly ordered. We know only that the largest element is in the root node and that, for each node, all its descendents are less than or equal to that node. But the relation between siblings (brothers) nodes or to continue the kinship terminology, between uncle and nephew nodes is not determined. The order of the elements obeys a linear line of descent, disregarding lateral lines.



Heaps as Priority Queues
A heap is an excellent way to implement a priority queue. Priority queue can be implemented using linked list for which the complexity was expressed in O(n). For large n , this may be too inefficient. On the other hand, a heap is a perfectly balanced tree, hence, reaching a leaf requires O(logn) searches. This efficiency is promising. Therefore, heaps can be used to implement priority queue. To this end, however, two procedures have to be implemented to enqueue and dequeue elements on a priority queue.
To enqueue an element, the element is added at the end of the heap as the last leaf. Restoring the heap priority in the case of enqueuing is achieved by moving from the leaf towards the root.
The algorithm for enqueuing is as follows:
heapEnque (el)
put el at the end of heap;
where el is not in the root and el >parent (el)
swap el with its parents;

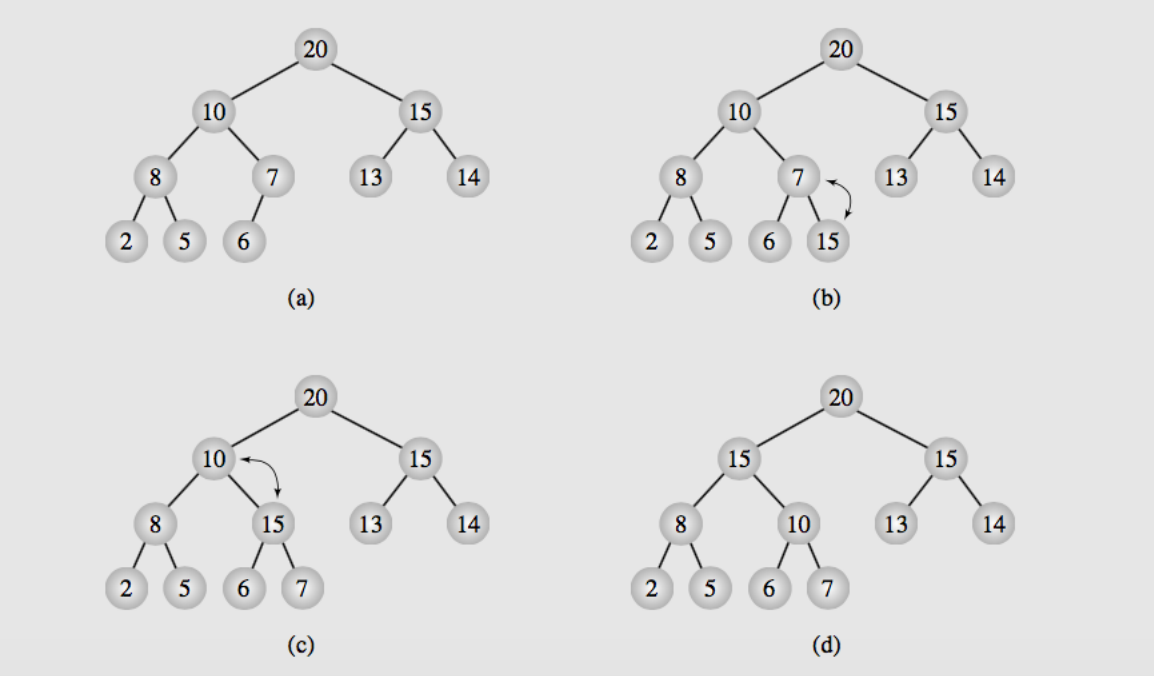
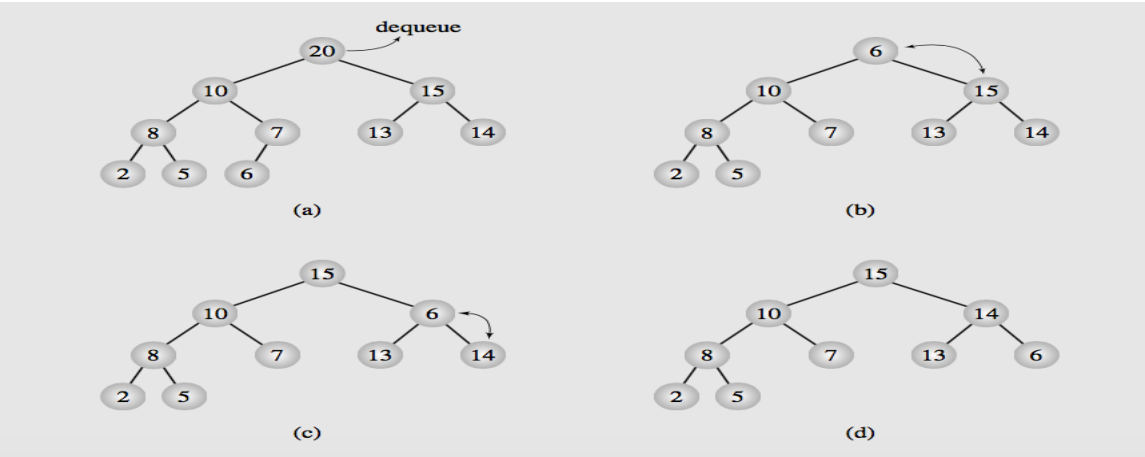


Fig: Enqueueing an element to a heap

Dequeuing an element from the heap consists of removing the root element from the heap, because by the heap property it is the element with the greatest priority. Then the last leaf is put in its place and the heap property almost certainly has to be restored, this time by moving from the root down the tree.

The algorithm for Dequeuing is as follows:

heapDequeue()
extract the element from the root;
put the element from the last leaf in its place;
remove the last leaf;
p = the root;
where p is not a leaf and p < any of its children
swap p with the larger child;



Algorithm to move the root element down a tree:

```
void moveDown(Object[] data, int first, int last)
{
        int largest = 2*first + 1;
        while (largest <= last)
        {
                if (largest < last && data[largest]).compareTo(data[largest+1]) < 0)
                        largest++;
                if (((Comparable)data[first]).compareTo(data[largest]) < 0)
                {
                        swap (data, first, largest);
                        first = largest;
                        largest = 2*first + 1;
                }
                else
                        largest = last + 1;
        }
}
```

Organizing Arrays as Heaps

Heaps can be implemented as arrays, and in that sense, each heap is an array, but all arrays are not heaps. In some situations, however, most notably in heap sort, we need to convert an array into a heap. There are several ways to do this, but the simplest way is to start with an empty heap and sequentially include elements into a growing heap. **This is a top down method** and it was proposed by John Williams; it extends the heap by enqueuing new element in the heap.

(i) A top down method:

Figure 6.57 contains a complete example of the top-down method. First, the number 2 is enqueued in the initially empty heap (6.57a). Next, 8 is enqueued by putting it at the end of the current heap (6.57b) and then swapping with its parent (6.57c). Enqueuing the third and fourth elements, 6 (6.57d) and then 1 (6.57e), necessitates no swaps. Enqueuing the fifth element, 10, amounts to putting it at the end of the heap (6.57f), then swapping it with its parent, 2 (6.57g), and then with its new parent, 8 (6.57h) so that eventually 10 percolates up to the root of the heap. All remaining steps can be traced by the reader in Figure 6.57.

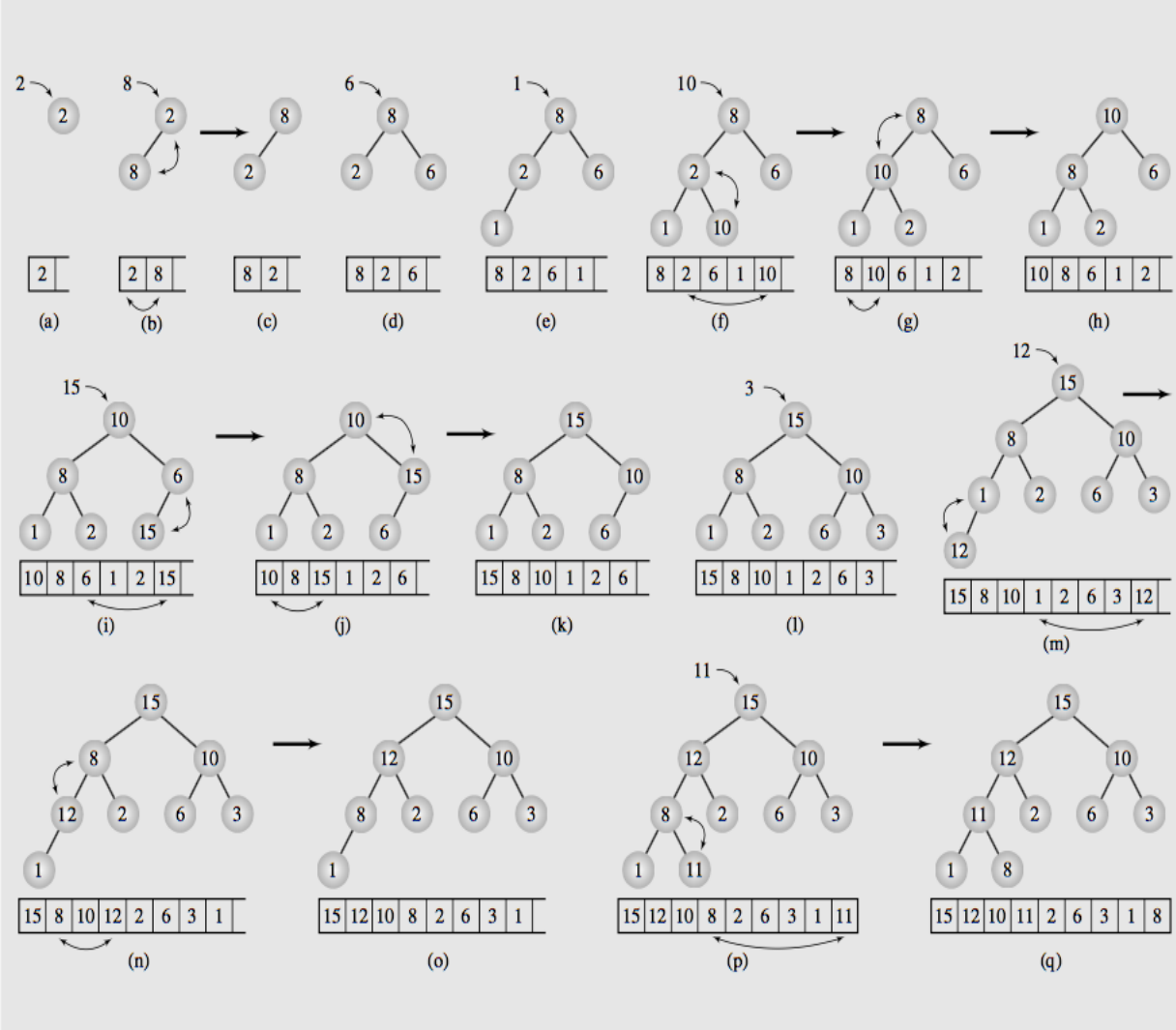


Figure: 6.57 (A Top-Down Method)

To check the complexity of the algorithm, observe that in the worst case, when a newly added element has to be moved up to the root of the tree, log k exchanges are made in a heap of k nodes. Therefore, if n elements are enqueued, then in the worst case

Log 1+log 2+log 3…….+log n = O(nlogn)

Exchanges are made during execution of the algorithm and the same number of comparisons.

(ii) A Bottom-Up Method (Floyd Algorithm):

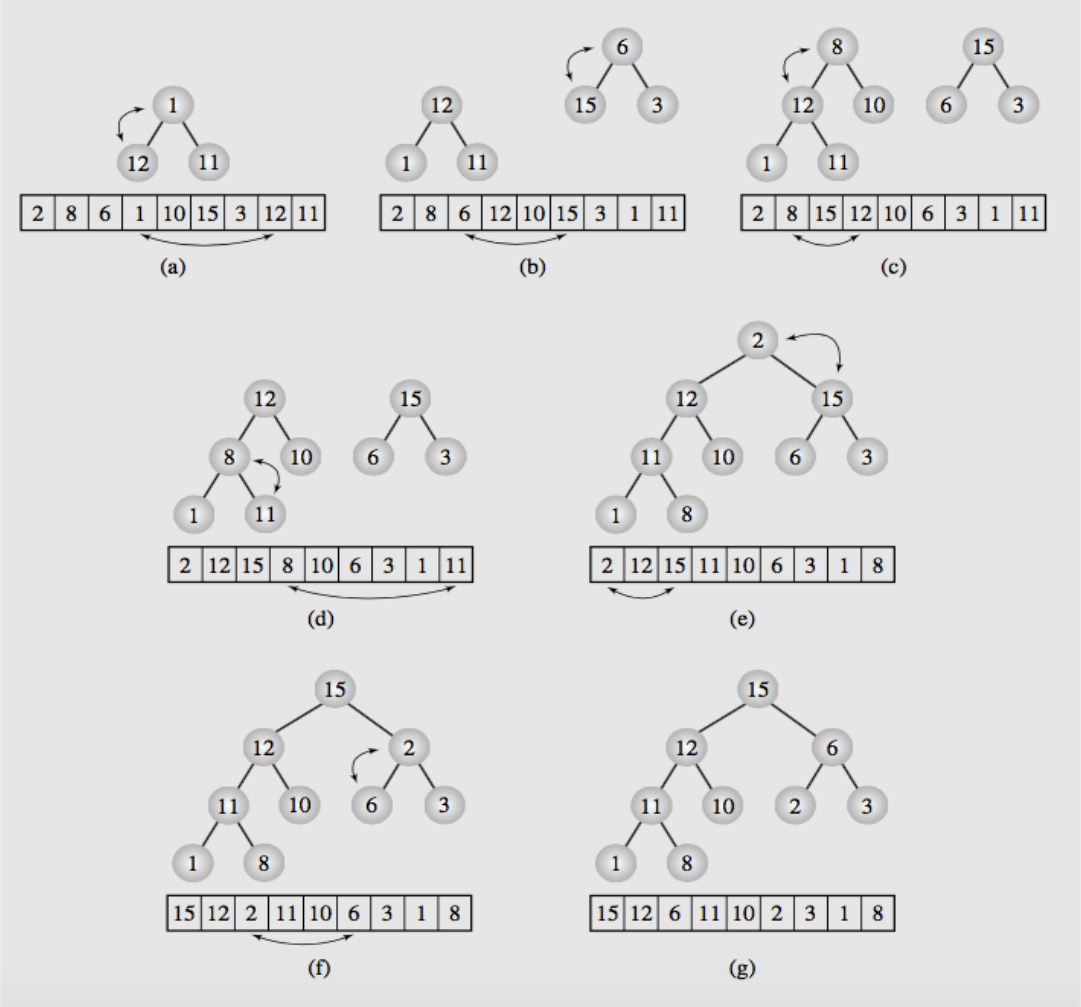
In another algorithm, developed by Robert Floyd, a heap is built bottom-up. In this approach, small heaps are formed and repetitively merged into larger heaps.

Algorithm:

```
FloydAlgorithm (data[])
    for i = index of the last nonleaf down to 0
        restore the heap property for the tree whose root is data[i] by calling
        moveDown(data, i, n-1);
```

We start from the last nonleaf node, which is data[n/2-1], *n* being the array size. If data[n/2-1] is less than one of its children, it is swapped with the larger child. In the tree in Figure 6.58a, this is the case for data [3] = 1 and data [7] = 12. After exchanging the elements, a new tree is created, shown in Figure 6.58b. Next the element data[n/2-2] = data [2] = 6 is considered. Because it is smaller than its child data [5] = 15, it is swapped with that child and the tree is transformed to that in Figure 6.58c. Now data[n/2-3] = data [1] = 8 is considered. Because it is smaller than one of its children, which is data [3] = 12, an interchange occurs, leading to the tree in Figure 6.58d. But now it can be noticed that the order established in the subtree whose root was 12 (Figure 6.58c) has been somewhat disturbed because 8 is smaller than its new child 11. This simply means that it does not suffice to compare a node's value with its children's, but a similar comparison needs to be done with grandchildren's, great-grandchildren's, and so on until the node finds its proper position. Taking this into consideration, the next swap is made, after which the tree in Figure 6.58e is created. Only now is the element data[n/2-4] = data [0] = 2 compared with its children, which leads to two swaps (Figure 6.58f).

Figure below contains an example of transforming the array data [] = [2 8 6 1 10 1531211] into a heap:



The following program creates heap from Array using bottom approach

```
public class HeapArray {
      public static void buildHeap(int []a){
      int n=a.length-1;
      for(int i=n/2-1;i>=0;i--){
         maxheap(a,i,n);
      }
   }
   public static void maxheap(int[] a, int i,int n){
      int left=2*i+1;
      int right=2*i+2;
      int largest;
      if(left <= n && a[left] > a[i]){
               largest=left;
      }
      else
               largest=i;
      if(right <= n && a[right] > a[largest]){
         largest=right;
      }
      if(largest!=i){
         exchange(a,i,largest);
         maxheap(a, largest,n);
      }
   }
   public static void exchange(int a[],int i, int j){
      int t=a[i];
      a[i]=a[j];
      a[j]=t;
      }
   public static void main(String[] args) {
      int arr[] = {1,6,7,8,2,3};
     buildHeap(arr);
      for(int i=0;i<arr.length;i++)
      System.out.print(arr[i]+ " ");
   }
}
```

Similarly, we can also create heap from top down approach. The procedure is almost same. The following program illustrates this technique very clearly.

Polish Notation and Expression Tree

One of the applications of binary trees is an unambiguous representation of arithmetical, relational or logical expressions. In the early 1920s, a Polish logician, Jan Lukasiewich invented a special notation, called Polish notation, results in less readable formulas than the parenthesized originals and it was not widely used. It was proved useful after the emergence of computers, especially for writing compilers and interpreters.

To maintain readability and prevent from ambiguity of formulas, extra symbols such as parentheses have to be used. However, if avoiding ambiguity is the only goal, then these symbols can be omitted at the cost of changing the order of symbols used in the formulas. This is exactly what the compiler does. It rejects everything that is not essential to retrieve the proper meaning of formulas, rejecting it as "syntactic sugar".

To understand how notation works, let us take following example:

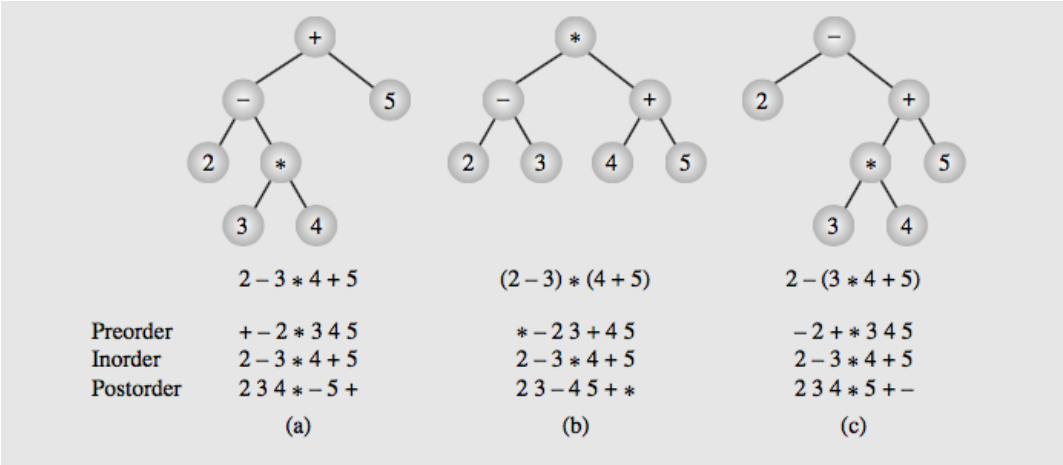
2-3.4+5

The result depends on the order in which the operations are performed. If we multiply first and then subtract and add, the result is -5 as expected. If subtraction is done first, then addition and multiplication as in:

(2-3).(4+5)

Then the result of evaluation is -15. If we see the first expression, then we know in what order to evaluate it. But the computer does not know that, in such a case, multiplication has precedence over addition and subtraction. If we want to override the precedence, then parentheses are needed.

Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations. Therefore, all expressions have to be broken down unambiguously into separate operations and put into their proper order. That is where the Polish notation is useful. It allows us to create an expression an expression tree, which imposes an order on execution of operations. For example, the first expression 2-3.4+5 which is same as 2-(3.4)+5 is represented by the following expression tree.



There is no ambiguity involved in this tree representation. The final result can be computed only if intermediate results as calculated first.
It can be noted that tree do not use parentheses and yet no ambiguity arises. We can maintain this parentheses free notation if the expression tree is linearized. The three traversal methods relevant here are preorder, inorder and postorder.
Because of the importance of these different conventions, special terminology is used. Preorder traversal generates prefix notation, inorder traversal generates infix notation and postorder traversal generates postfix notation.

Operations on Expression Trees

Binary trees can be created in two different ways: top down or bottom up. In the implementation of insertion, the first approach was used. But here we second approach by creating expression trees bottom up while scanning infix expressions from left to right.
The most important part of this construction is retaining the same precedence of operations as in the expression being scanned. If parentheses are not allowed, the task is simple, as parentheses allow for many levels of nesting. Therefore, an algorithm should be powerful enough to process any number of nesting levels in an expression. A natural approach is a recursive implementation.