

# TREE

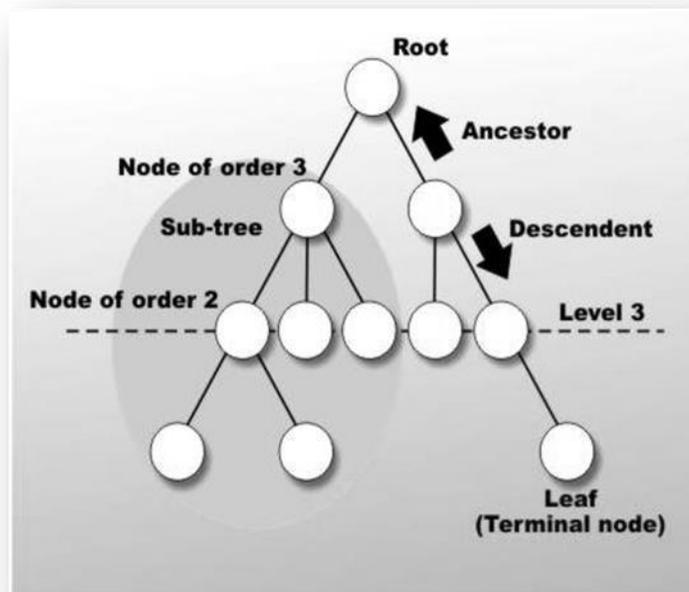
## Trees Basic Concepts:

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

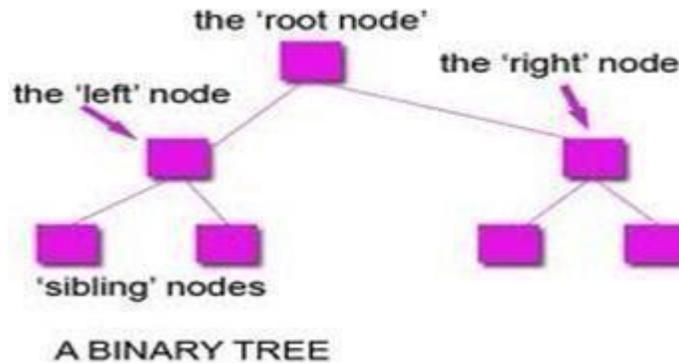
- If T is not empty, T has a special tree called the root that has no parent.
- Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.

Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.



## Binary Tree:

In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.



### Tree Terminology:

#### Leaf node

A node with no children is called a leaf (or external node). A node which is not a leaf is called an internal node.

**Path:** A sequence of nodes  $n_1, n_2, \dots, n_k$ , such that  $n_i$  is the parent of  $n_{i+1}$  for  $i = 1, 2, \dots, k - 1$ . The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

**Siblings:** The children of the same parent are called siblings.

**Ancestor and Descendent** If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

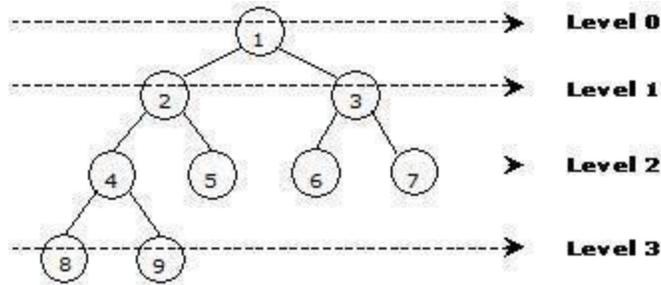
**Subtree:** Any node of a tree, with all of its descendants is a subtree.

**Level:** The level of the node refers to its distance from the root. The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its parent.

*The maximum number of nodes at any level is  $2^n$ .*

**Height:** The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. **Depth:** The depth of a node is the number of nodes along the path from the root to that node.

**Assigning level numbers and Numbering of nodes for a binary tree:** The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



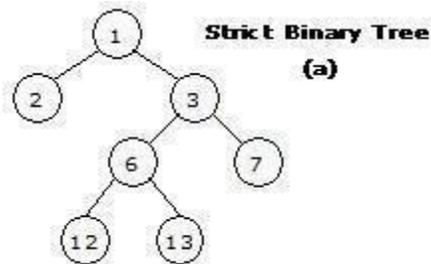
### Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If  $h$  = height of a binary tree, then
  - a. Maximum number of leaves =  $2^h$
  - b. Maximum number of nodes =  $2^{h+1} - 1$
2. If a binary tree contains  $m$  nodes at level  $l$ , it contains at most  $2m$  nodes at level  $l + 1$ .
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most  $2^l$  nodes at level  $l$ .
4. The total number of edges in a full binary tree with  $n$  nodes is  $n - 1$ .

### Strictly Binary tree:

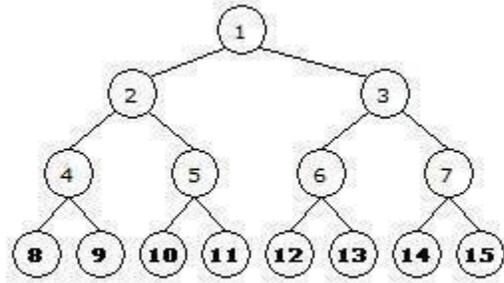
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with  $n$  leaves always contains  $2n - 1$  nodes.



### Full Binary Tree:

A full binary tree of height  $h$  has all its leaves at level  $h$ . Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height  $h$  has  $2^{h+1} - 1$  nodes. A full binary tree of height  $h$  is a *strictly binary tree* all of whose leaves are at level  $h$ .



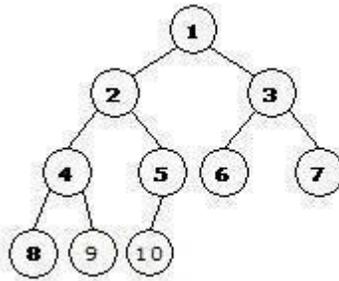
**Full binary tree (d)**

For example, a full binary tree of height 3 contains  $2^{3+1} - 1 = 15$  nodes.

**Complete Binary Tree:**

A binary tree with  $n$  nodes is said to be **complete** if it contains all the first  $n$  nodes of the above numbering scheme.

A complete binary tree of height  $h$  looks like a full binary tree down to level  $h-1$ , and the level  $h$  is filled from left to right.

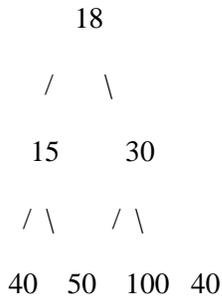


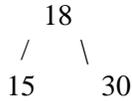
**Complete binary tree (c)**

**Perfect Binary Tree:**

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

Following are examples of Perfect Binary Trees.





A Perfect Binary Tree of height  $h$  (where height is number of nodes on path from root to leaf) has  $2^h - 1$  node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

**Balanced Binary Tree:**

A binary tree is balanced if height of the tree is  $O(\log n)$  where  $n$  is number of nodes. For Example, AVL tree maintain  $O(\log n)$  height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain  $O(\log n)$  height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide  $O(\log n)$  time for search, insert and delete.

**Representation of Binary Trees:**

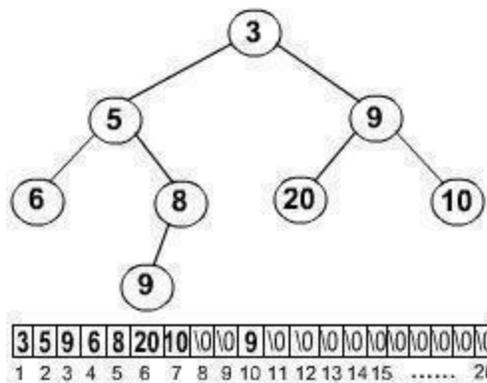
1. Array Representation of Binary Tree
2. Pointer-based.

**Array Representation of Binary Tree:**

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index  $i$  is put into the array as its  $i^{\text{th}}$  element.

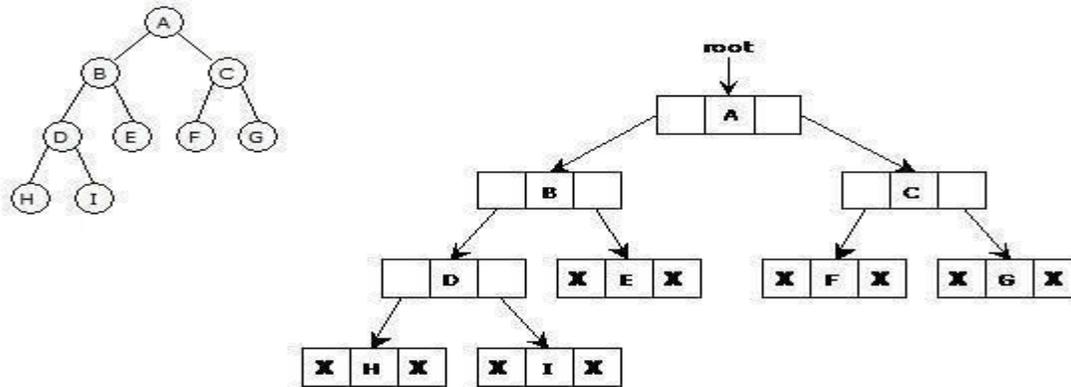
In the figure shown below the nodes of binary tree are numbered according to the given scheme.



The figure shows how a binary tree is represented as an array. The root 3 is the 0<sup>th</sup> element while its leftchild 5 is the 1<sup>st</sup> element of the array. Node 6 does not have any child so its children i.e. 7<sup>th</sup> and 8<sup>th</sup> element of the array are shown as a Null value.

It is found that if  $n$  is the number or index of a node, then its left child occurs at  $(2n + 1)^{\text{th}}$  position and right child at  $(2n + 2)^{\text{th}}$  position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

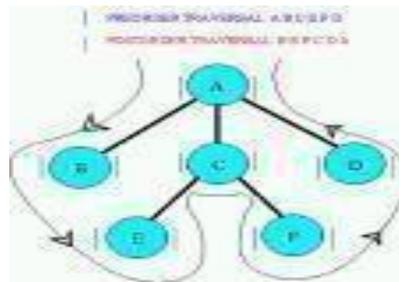
### Linked Representation of Binary Tree (Pointer based):



Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

### Binary Tree Traversals:

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all th it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. Th ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- ✦ Inorder Traversal
- ✦ Preorder Traversal
- ✦ Postorder Traversal

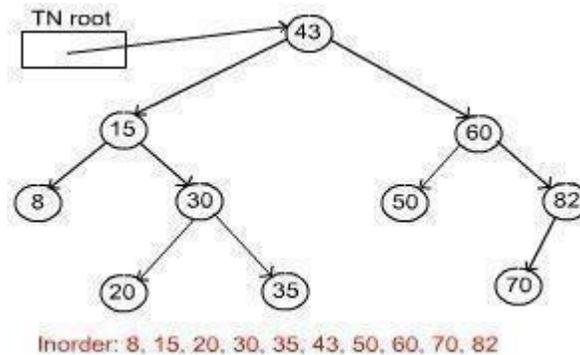
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are based functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

### Inorder Traversal:

To traverse a non empty tree in inorder the following steps are followed recursively.

- ✦ Visit the Root
- ✦ Traverse the left subtree
- ✦ Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



### Preorder Traversal:

Algorithm Pre-order(tree)

1. Visit the root.
2. Traverse the left sub-tree, i.e., call Pre-order(left-sub-tree)
3. Traverse the right sub-tree, i.e., call Pre-order(right-sub-tree)

### Post-order Traversal:

Algorithm Post-order(tree)

1. Traverse the left sub-tree, i.e., call Post-order(left-subtree)
2. Traverse the right sub-tree, i.e., call Post-order(right-sub tree)
3. Visit the root.

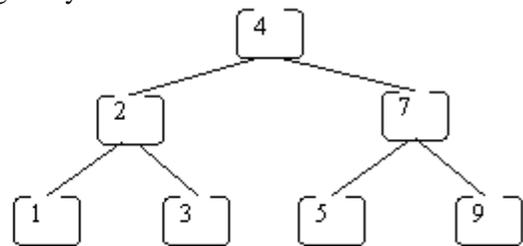
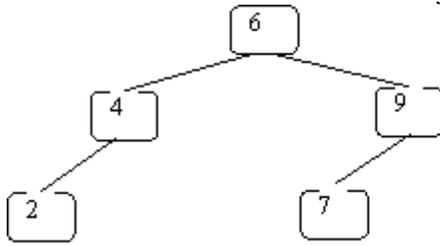
### Binary Search Trees:

An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node  $n$  in the tree:

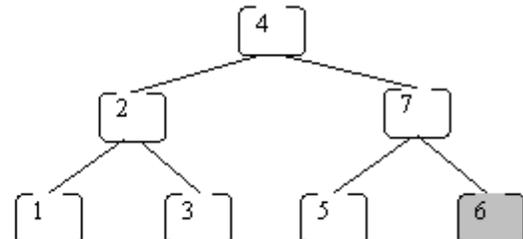
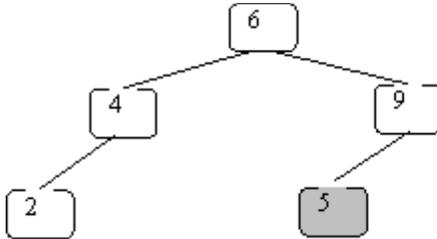
- All keys in  $n$ 's left subtree are less than the key in  $n$ , and
- All keys in  $n$ 's right subtree are greater than the key in  $n$ .

In other words, binary search trees are binary trees in which all values in the node's left subtree are less than node value all values in the node's right subtree are greater than node value.

Here are some BSTs in which each node just stores an integer key:



These are not BSTs:

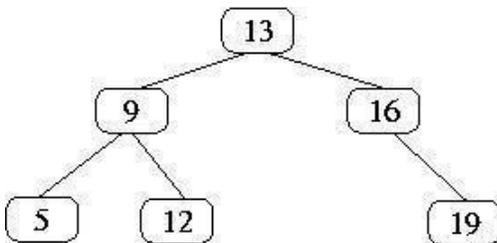


In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

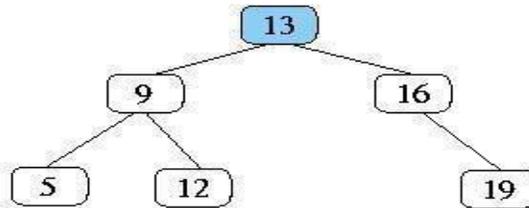
- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

Let's illustrate what happens using the following BST:

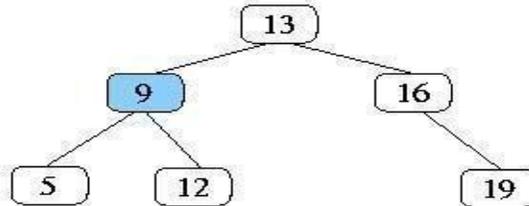


and searching for 12:

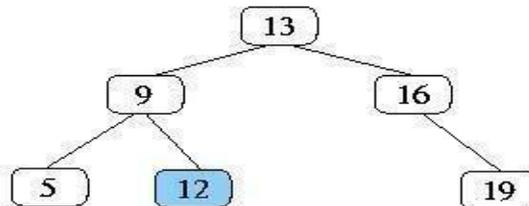
12 < 13 so go to  
left subtree



12 > 9 so go to  
right subtree.

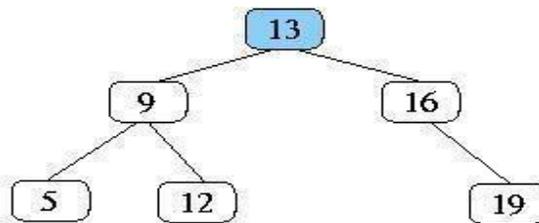


found!

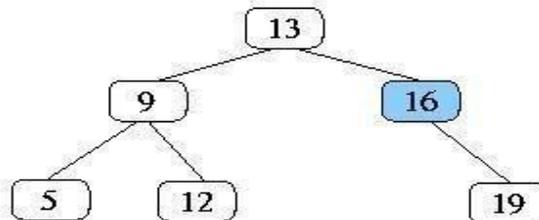


What if we search for 15:

15 > 13 so go to  
right subtree



15 < 16 so go to  
left subtree. It  
does not exist so  
search fails and it  
returns false



## Properties and Operations:

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are  $<$  (less) than the key in its parent node
3. The keys in the right subtree  $>$  (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

### Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node  $N$  to tree  $T$ . if the tree is empty, the we return new node  $N$  as the tree. Otherwise, the problem of inserting is reduced to inserting the node  $N$  to left of right sub trees of  $T$ , depending on  $N$  is less or greater than  $T$ . A definition is as follows.

$$\begin{aligned}\text{Insert}(N, T) &= N \quad \text{if } T \text{ is empty} \\ &= \text{insert}(N, T.\text{left}) \quad \text{if } N < T \\ &= \text{insert}(N, T.\text{right}) \quad \text{if } N > T\end{aligned}$$

### Searching for a node

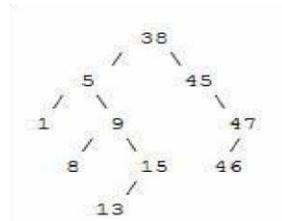
Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for  $T.\text{left}$  or  $T.\text{right}$ , depending on  $N < T$  or  $N > T$ . A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not. Search( $N$ ,

$$\begin{aligned}T) &= \text{false} \quad \text{if } T \text{ is empty} \\ &= \text{true} \quad \text{if } T = N \\ &= \text{search}(N, T.\text{left}) \quad \text{if } N < T \\ &= \text{search}(N, T.\text{right}) \quad \text{if } N > T\end{aligned}$$

### Deleting a node

A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.

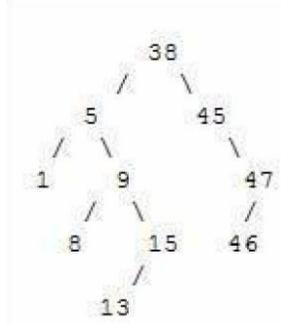


Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with

just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

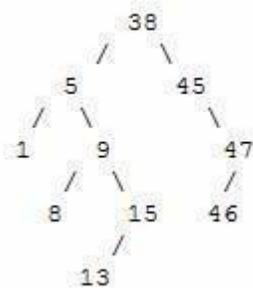
**Case 1 : The node to delete is a leaf node**

This is a very easy case. Just delete the node 46. We are done



**Case 2 : The node to delete is a node with one child.**

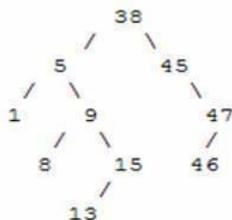
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.



**Case 3: The node to delete is a node with two children**

This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

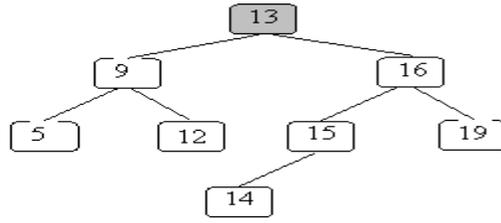
Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.



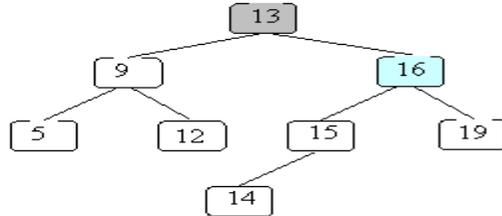
Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node.

Let's see when we delete 13 from that tree.

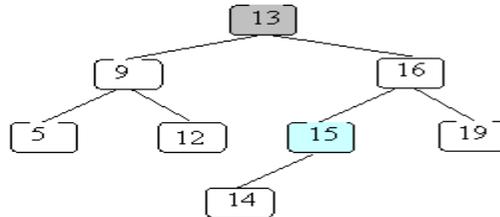
Original BST with 13 located



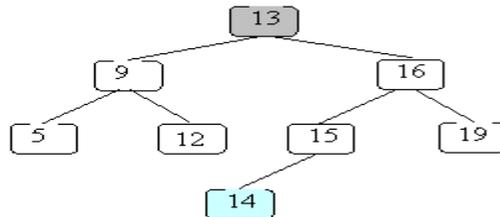
Step into right subtree.



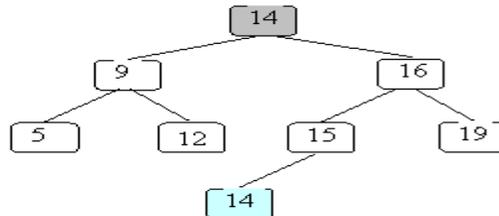
Go to left child.



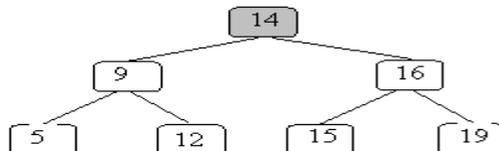
Continue to left child. This is last one.



Replace node to delete with far left child of right subtree.



Remove far left child of right subtree.



## Balanced Search Trees:

A **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions. The red–black tree, which is a type of selfbalancing binary search tree, was called symmetric binary B-tree. Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of selfbalancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Selfbalancing BSTs have better worst-case lookup performance than hash tables ( $O(\log n)$  compared to  $O(n)$ ), but have worse average-case performance ( $O(\log n)$  compared to  $O(1)$ ).

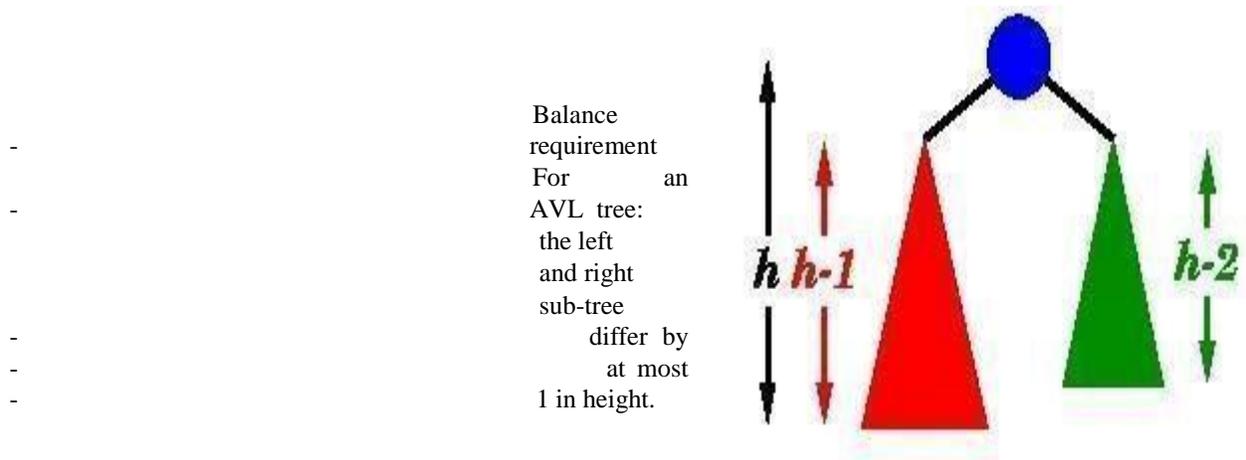
Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal  $O(n \log n)$  sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than merge sort, quicksort, or heapsort, because of the tree balancing overhead as well as cache access patterns.)

Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in  $O(\log n)$  time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms. **AVL Trees:**

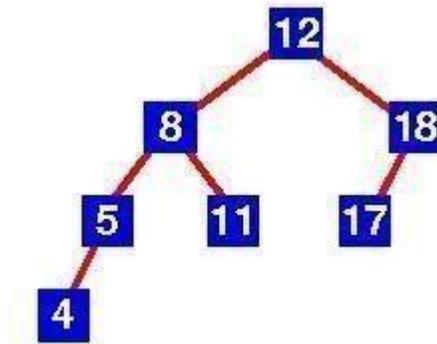
An **AVL tree** is another balanced binary search tree. Named after their inventors, **Adelson Velskii** and **Landis**, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an  $O(\log n)$  search time. Addition and deletion operations also take  $O(\log n)$  time.

**Definition of an AVL tree:** An AVL tree is a binary search tree which has the following properties:

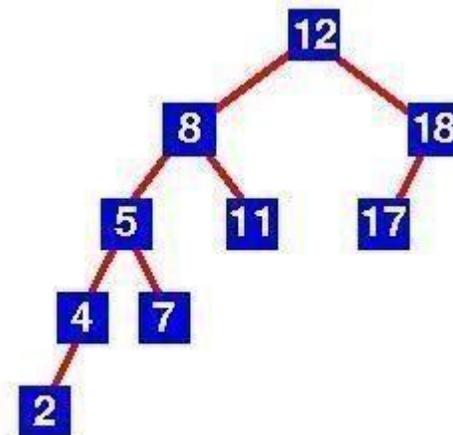
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



For example, here are some trees:



Yes this is an AVL tree. Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.



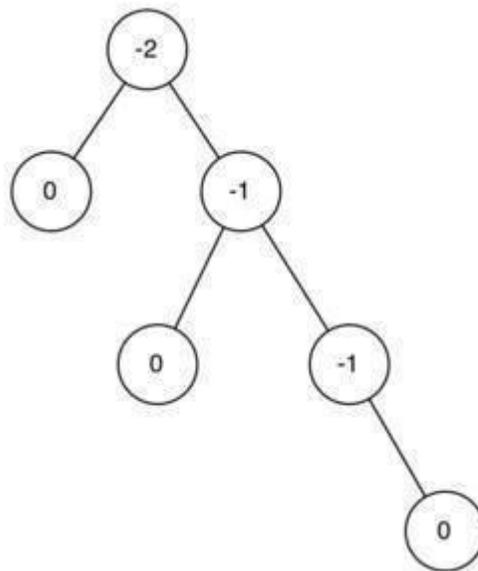
No, this is not an AVL tree. Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2.

An AVL tree implements the Map abstract data type just like a regular binary search tree, the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a

**balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$\text{balanceFactor} = \text{height}(\text{leftSubTree}) - \text{height}(\text{rightSubTree})$$

Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance. For purposes of implementing an AVL tree, and gaining the benefit of having a balanced tree we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance. Figure shows an example of an unbalanced, right-heavy tree and the balance factors of each node.



## Properties of AVL Trees

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 (Weiss, 1993, p:108). AVL trees are HB-k trees (height balanced trees of order k) of order HB-1.

The following is the height differential formula:

$$|\text{Height}(T_L) - \text{Height}(T_R)| \leq k$$

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same. Updates of AVL trees require up to  $O(\log n)$  rotations, whereas updating redblack trees can be done using only one or two rotations (up to color changes). For this reason, they (AVL trees) are considered a bit obsolete by some.

### Sparse AVL trees

Sparse AVL trees are defined as AVL trees of height  $h$  with the fewest possible nodes. Figure 3 shows sparse AVL trees of heights 0, 1, 2, and 3.

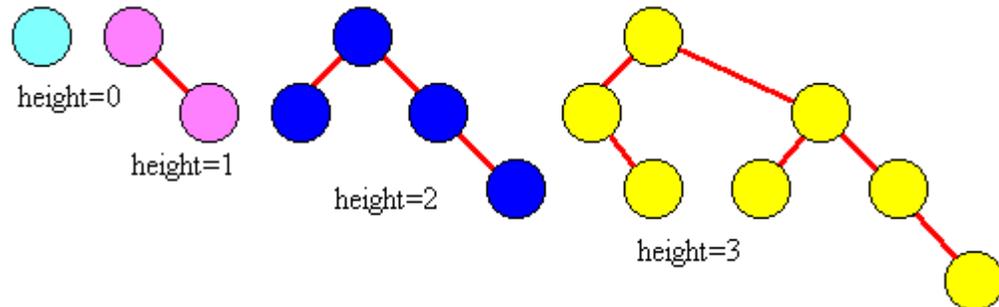
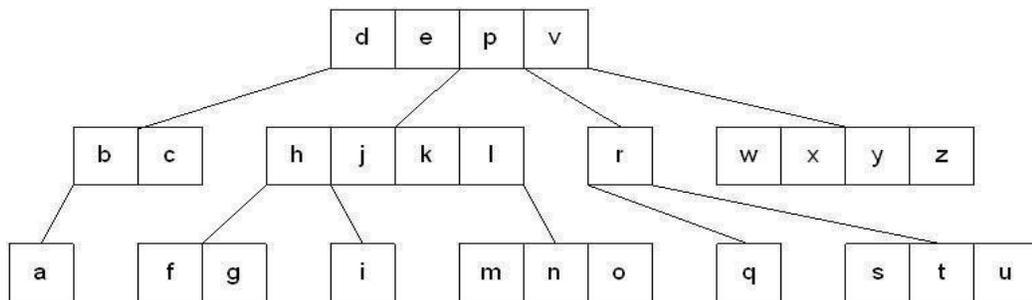


Figure Structure of an AVL tree

### Introduction to M-Way Search Trees:

A **multiway tree** is a tree that can have more than two children. A **multiway tree of order  $m$**  (or an  **$m$ -way tree**) is one in which a tree can have  $m$  children.

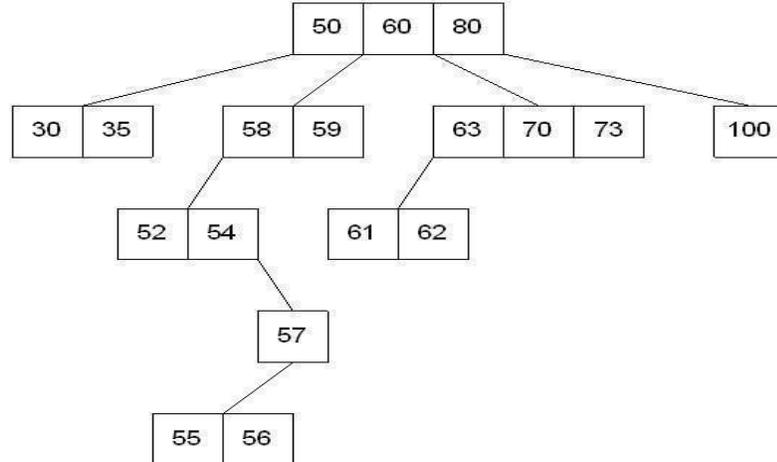
As with the other trees that have been studied, the nodes in an  $m$ -way tree will be made up of key fields, in this case  $m-1$  key fields, and pointers to children. Multiday tree of order 5



To make the processing of  $m$ -way trees easier some type of order will be imposed on the keys within each node, resulting in a **multiway search tree of order  $m$**  (or an  **$m$ -way search tree**). By definition an  $m$ -way search tree is a  $m$ -way tree in which:

- Each node has  $m$  children and  $m-1$  key fields
- The keys in each node are in ascending order.
- The keys in the first  $i$  children are smaller than the  $i$ th key
- The keys in the last  $m-i$  children are larger than the  $i$ th key

## 4-way search tree



M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update. However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance. **B Trees:**

An extension of a multiway search tree of order  $m$  is a **B-tree of order  $m$** . This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order  $m$  is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.
2. Each non root and each non leaf node have at most  $m$  nonempty children and at least  $m/2$  nonempty children.
3. The number of keys in each non root and each non leaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

## Searching a B-tree

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

## Insertion into a B-tree

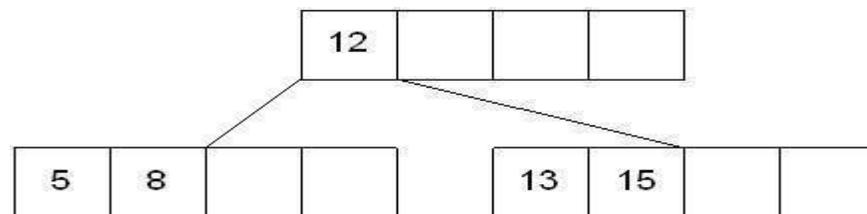
The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

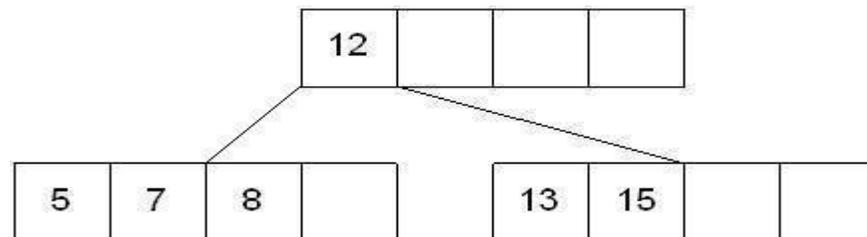
1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

### Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:

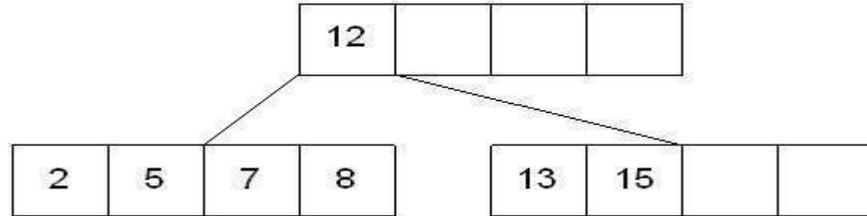


### Case 2: The leaf in which a key is to be placed is full

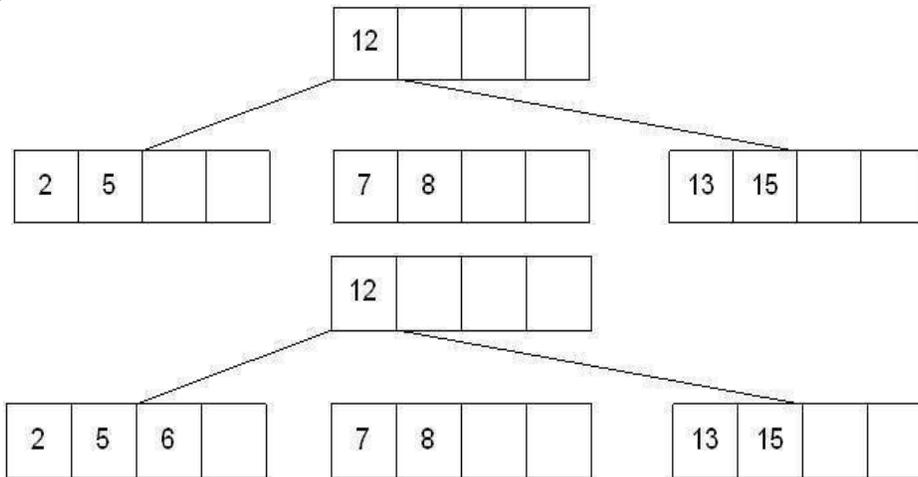
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process continues up the tree until all of the values have "found" a location.

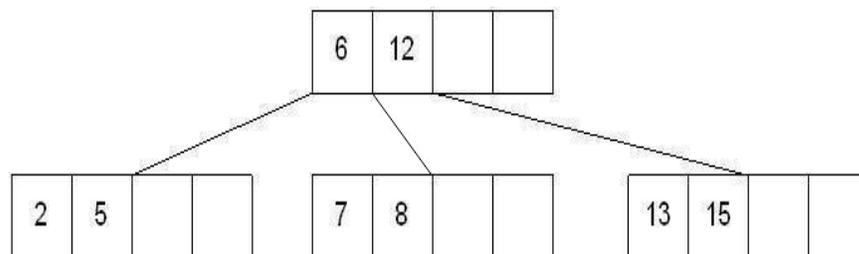
Insert 6 into the following B-tree:



results in a split of the first leaf node:



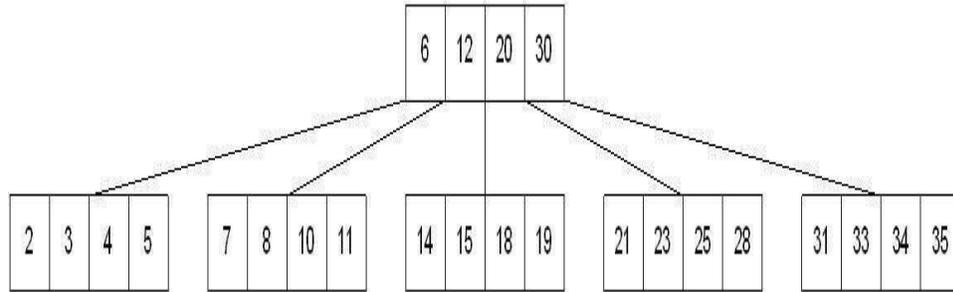
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



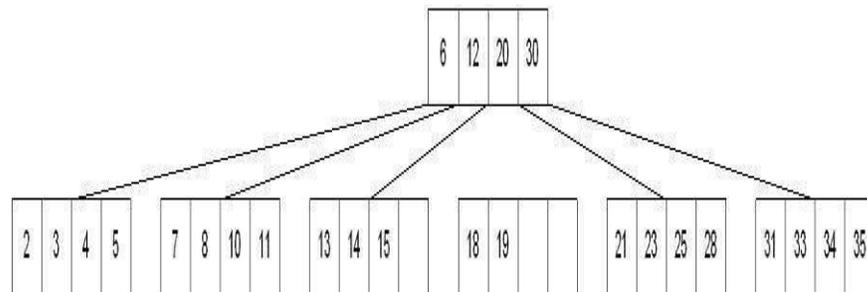
**Case 3: The root of the B-tree is full**

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the Btree.

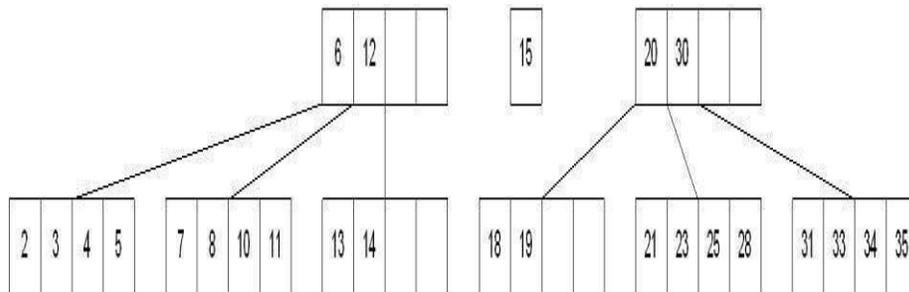
Inserting 13 into the following tree:



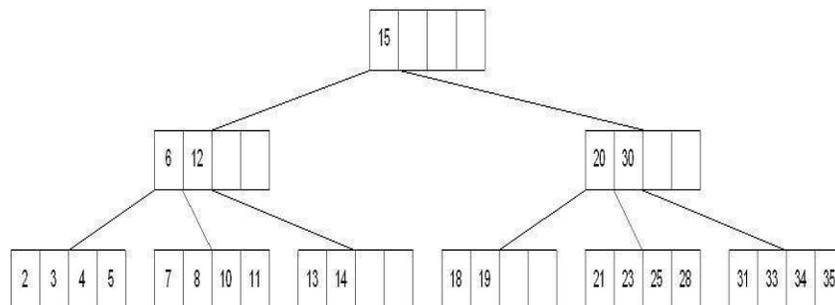
Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



## Deleting from a B-tree

As usual, this is the hardest of the processes to apply. The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

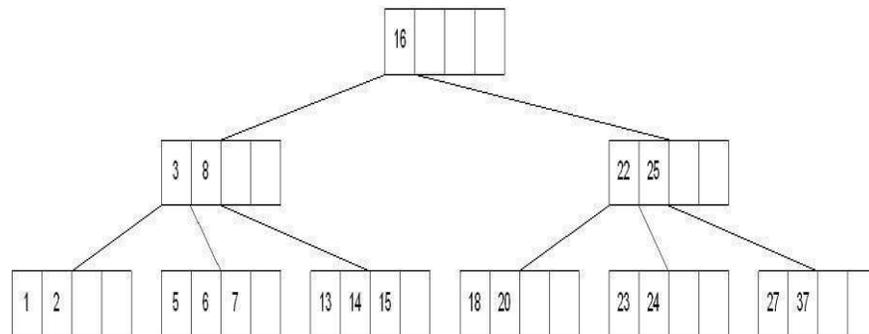
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

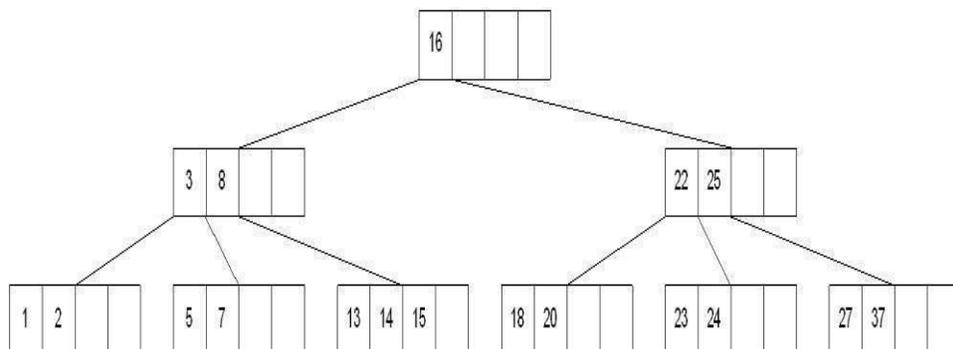
### Case 1: Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

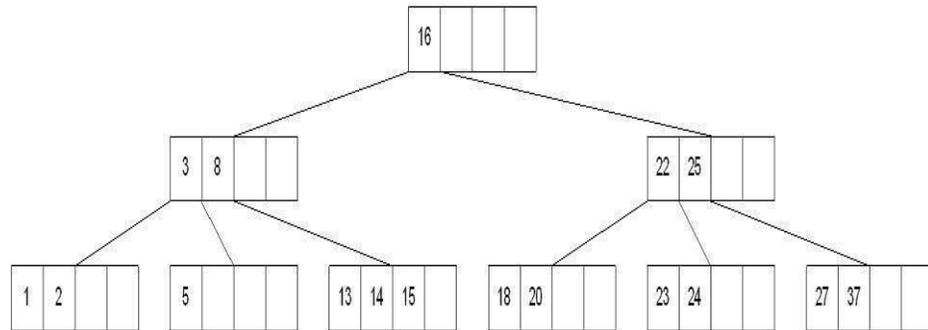


results in:

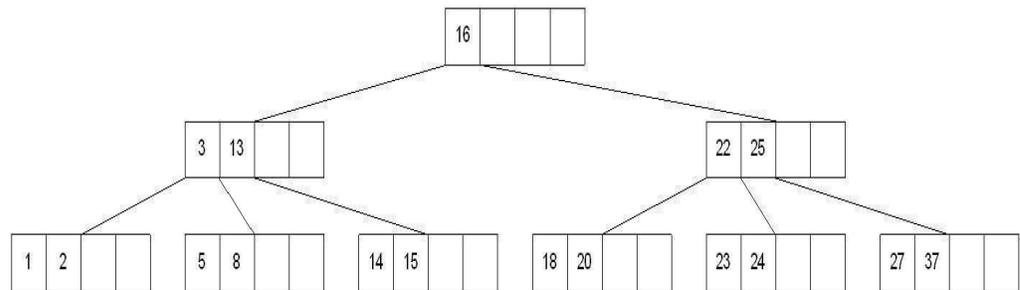


1b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

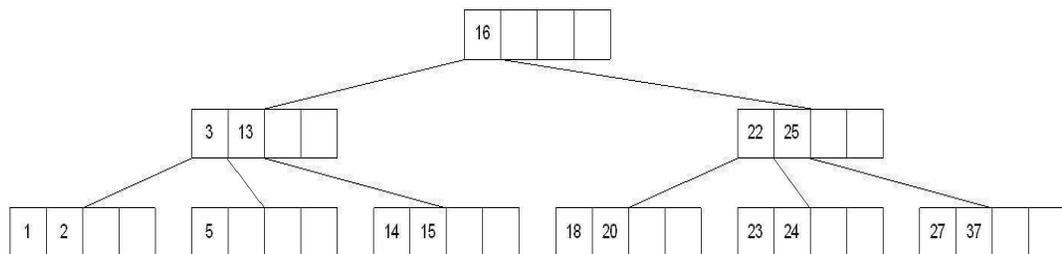
Deleting 7 from the tree above results in:



1b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

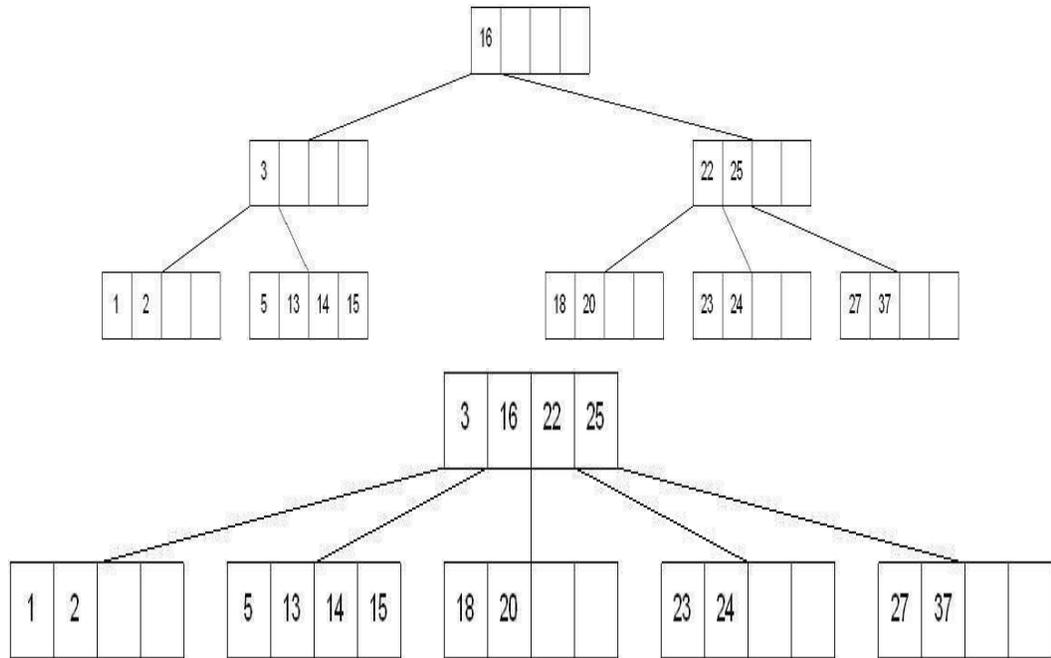


Now delete 8 from the tree:



1b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step 1b-2 until the minimum requirement is met or the root of the tree is reached.

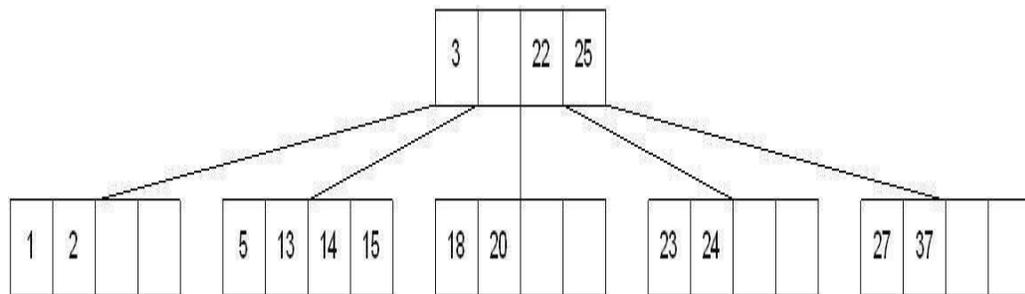
**Special Case for 1b-2:** When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will become the new root for the B-tree. Both the sibling and the old root will be discarded.



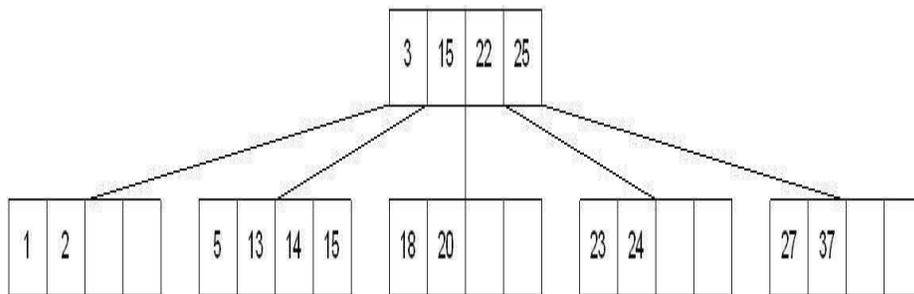
**Case 2: Deletion from a non-leaf**

This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

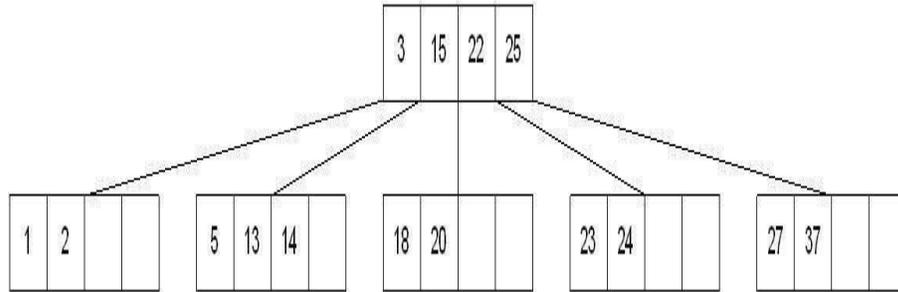
The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node. Deleting 16 from the tree above results in:



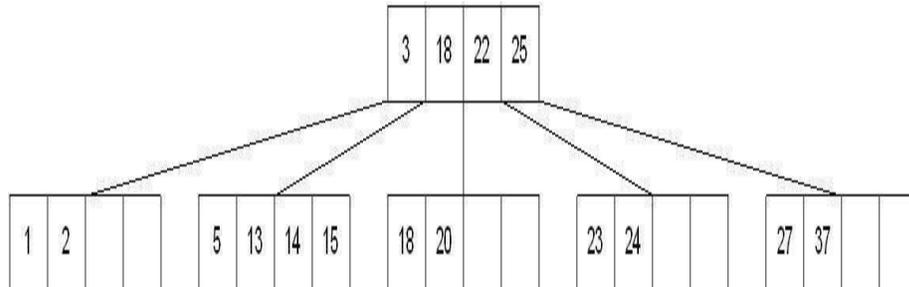
The "gap" is filled in with the immediate predecessor:



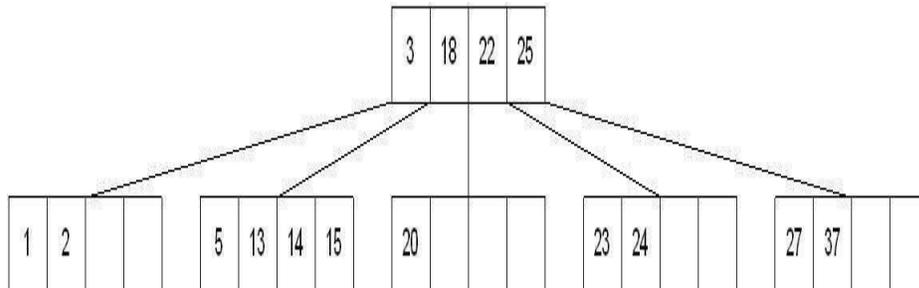
and then the immediate predecessor is deleted:



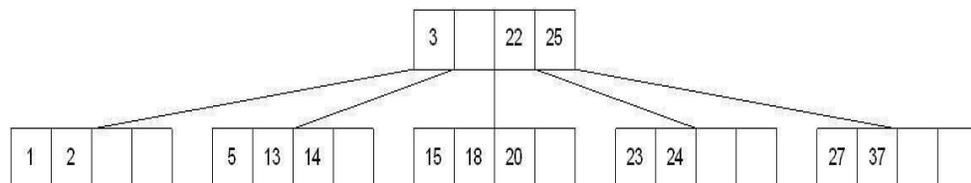
If the immediate successor had been chosen as the replacement:



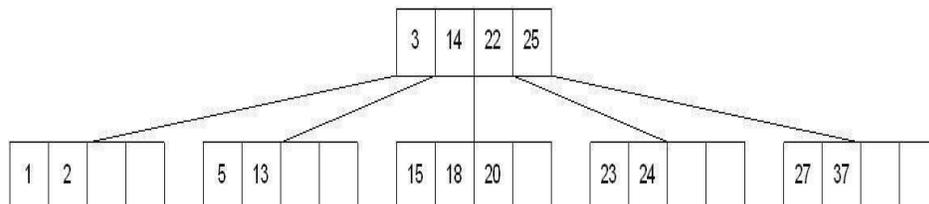
Deleting the successor results in:



The vales in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:



and then the middle value is moved to the parent:



# GRAPH

## Basic Graph Concepts:

Graph is a data structure that consists of following two components:

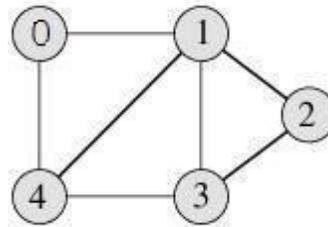
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge.

The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph (di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

## Graph and its representations:

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

## Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[i][j]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If  $adj[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .

The adjacency matrix for the above example graph is:

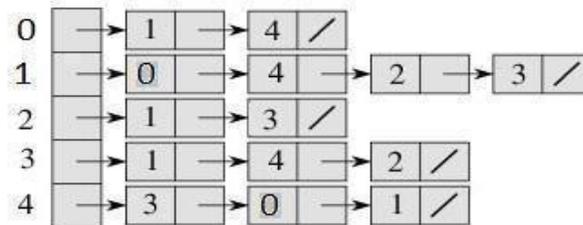
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation of the above graph

*Pros:* Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done  $O(1)$ .

*Cons:* Consumes more space  $O(V^2)$ . Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time.

**Adjacency List:** An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be `array[]`. An entry `array[i]` represents the linked list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

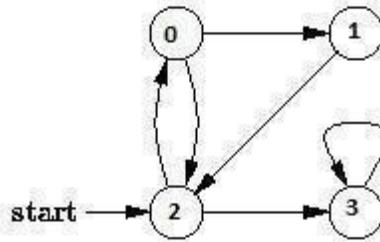


Adjacency List Representation of the above Graph

## Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.



### Algorithm: Breadth-First Search Traversal

BFS(V, E, s)

```

for each u in V - {s}
  do color[u] ← WHITE
     d[u] ← infinity
     π[u] ← NIL
  color[s] ← GRAY

  d[s] ← 0
  π[s] ← NIL
  Q ← {}
  ENQUEUE(Q, s)
while Q is non-empty
  do u ← DEQUEUE(Q)
     for each v adjacent to u
       do if color[v] ← WHITE
          then color[v] ← GRAY
             d[v] ← d[u] + 1
             π[v] ← u
             ENQUEUE(Q, v)
     DEQUEUE(Q)
  color[u] ← BLACK
  
```

### Applications of Breadth First Traversal

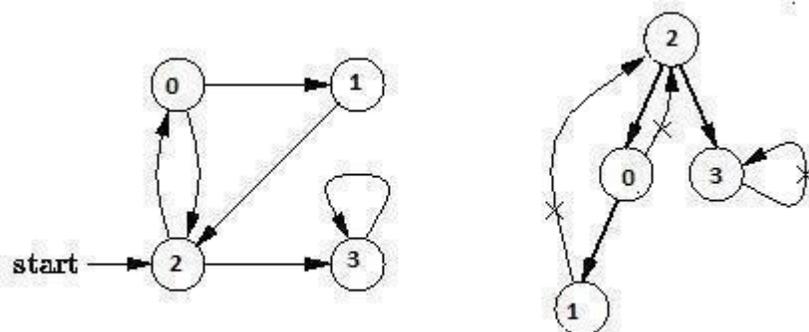
- 1) **Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
- 2) **Peer to Peer Networks.** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

- 3) **Crawlers in Search Engines:** Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
- 4) **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
- 5) **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- 6) **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- 7) **In Garbage Collection:** Breadth First Search is used in copying garbage collection using Cheney's algorithm.
- 8) **Cycle detection in undirected graph:** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
- 9) **Ford-Fulkerson algorithm** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to  $O(VE^2)$ .
- 10) **To test if a graph is Bipartite** We can either use Breadth First or Depth First Traversal.
- 11) **Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
- 12) **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

### Depth First Traversal for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the following graph is 2, 0, 1, 3



## Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges  $(u, v)$  such that  $u$  is gray and  $v$  is white when edge  $(u, v)$  is explored. The following pseudocode for DFS uses a global timestamp  $time$ .

### DFS (V, E)

```
for each vertex  $u$  in  $V[G]$ 
  do color[ $u$ ] ← WHITE
      $\pi[u]$  ← NIL
     time ← 0
  for each vertex  $u$  in  $V[G]$ 
    do if color[ $u$ ] ← WHITE
       then DFS-Visit( $u$ )
```

### DFS-Visit( $u$ )

```
color[ $u$ ] ← GRAY
time ← time + 1
d[ $u$ ] ← time
for each vertex  $v$  adjacent to  $u$ 
  do if color[ $v$ ] ← WHITE
     then  $\pi[v]$  ←  $u$ 
        DFS-Visit( $v$ )
color[ $u$ ] ← BLACK
time ← time + 1
f[ $u$ ] ← time
```

## Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

### 2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

### 3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices  $u$  and  $z$ . i) Call DFS( $G, u$ ) with  $u$  as the start vertex.

ii) Use a stack  $S$  to keep track of the path between the start vertex and the current vertex. iii)

As soon as destination vertex  $z$  is encountered, return the path as the contents of the stack

### 4) Topological Sorting

### 5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

- 6) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based also for finding Strongly Connected Components)