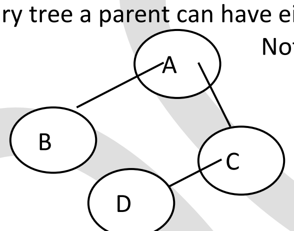
5. The depth of a node n is the unique path from root node to the node n. The depth of the tree is the unique path from root node to the deepest leaf node.
6. The height of a node n is the unique path from the node n to the root node. The height of the tree is the unique path from deepest leaf node to the root node.
7. The height of the tree must be equal to the depth of the tree.
   Therefore H(T)= D(T). Where H represents the Height, D represents the Depth and T represents the Tree.
8. All the leaves at height zero and the depth of the root is zero.
9. If there is a direct path from n1 to n2 then n1 is an ancestor of n2 and n2 is descendant of n1.
10. A tree should not have multiple paths from node n1 to node n2.

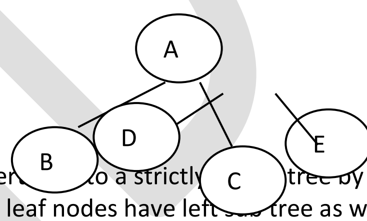
The above tree, height and depth of the tree: 3
Height of root node: 3 ; Depth of all the leaf nodes: 3
Depth of Root node: 0 ; Height of all the leaves : 0

10.2.1 Binary Tree: A tree is called binary tree if all the nodes in the tree has at the most two children. In a binary tree a parent can have either zero child or one child or two children.



Note: All the nodes have at the most two children.

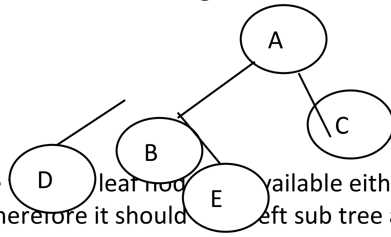
10.2.2 Strictly Binary Tree: A binary tree is called strictly binary tree if every non leaf node in a binary tree has non empty left sub-tree and right sub-tree.



Any binary tree can be converted into a strictly binary tree by adding left sub-tree or right sub-tree. In the above tree all the non leaf nodes have left sub tree as well as right sub tree also. A is a non leaf node has left and right sub tree similarly C is a non leaf node has left and right sub tree. Therefore the above tree is a strictly binary tree.

10.2.3 Almost complete Binary Tree: A binary tree is called almost complete binary tree if it satisfys the following two properties:

1. All the leaf nodes should be present either at level d or d-1.
2. Any non leaf none if it has right sub-tree then there should be left sub-tree also.

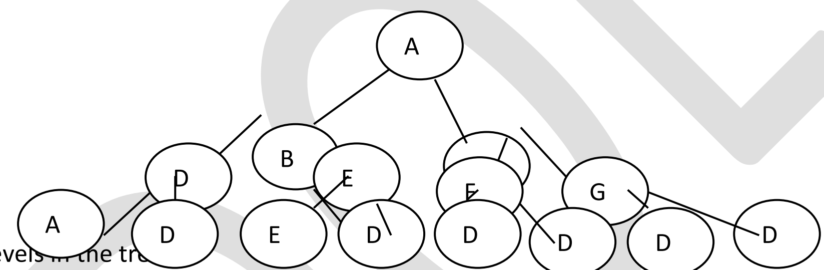


In this tree C, D, E are leaf nodes available either at depth d or d-1.
B has right sub tree therefore it should have left sub tree as well. The node B has both the sub trees hence it is an almost complete binary tree.

10.2.4 Complete Binary Tree: A binary tree is called complete binary tree if it satisfies the following properties:

1. Each node in tree must have at the most two children.
2. In a complete binary tree with n node at position i , the left child node must be present at position 2i such that 2i<=N where N is the total number of nodes in a tree and the right child node must be present at position 2i+1 such that 2i<=N where N is the total number of nodes in a tree.
3. The parent of left child node must be present at position i/2 such that i/2<N where N is the total number of nodes in a tree and the parent of right child node must be present at position (i-1)/2 such that (i-1)/2< N where N is the total number of nodes in a tree.
4. The complete binary tree at depth d is a strictly binary tree in which all the leaves are present at depth d.
5. The complete binary tree of level l contains 2^l at any level of a complete binary tree.
6. The total number of nodes in a complete binary tree will be calculated E(l=0 to d) 2^l.

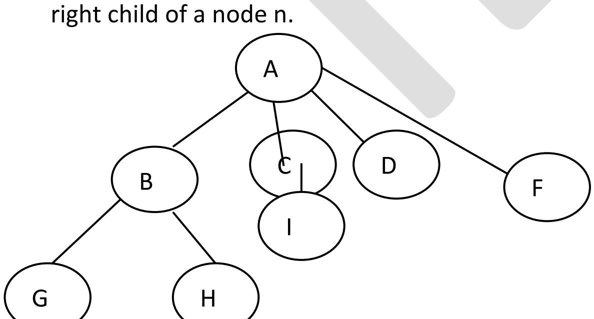
L0

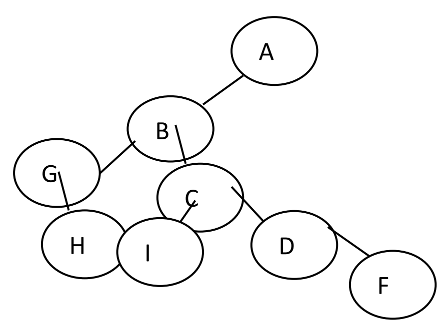


L1
L2
L3
L4

There are 4 levels in the tr
Each level number of nodes in Complete Binary : 2^0
Total Number of nodes at level 0: 2^0=1
Total Number of nodes at level 1: 2^1=2
Total Number of nodes at level 2: 2^2=4
Total Number of nodes at level 2: 2^3=8
Total Number of nodes in the tree: L0 + L1 + L2 + L3= 1+2+4+8= 15

10.3 Conversion of Tree to Binary tree: A tree can be converted to a binary tree with the help of following statement:
The left child of a node n will remain the left child of a node n and the right sibling will become the right child of a node n.



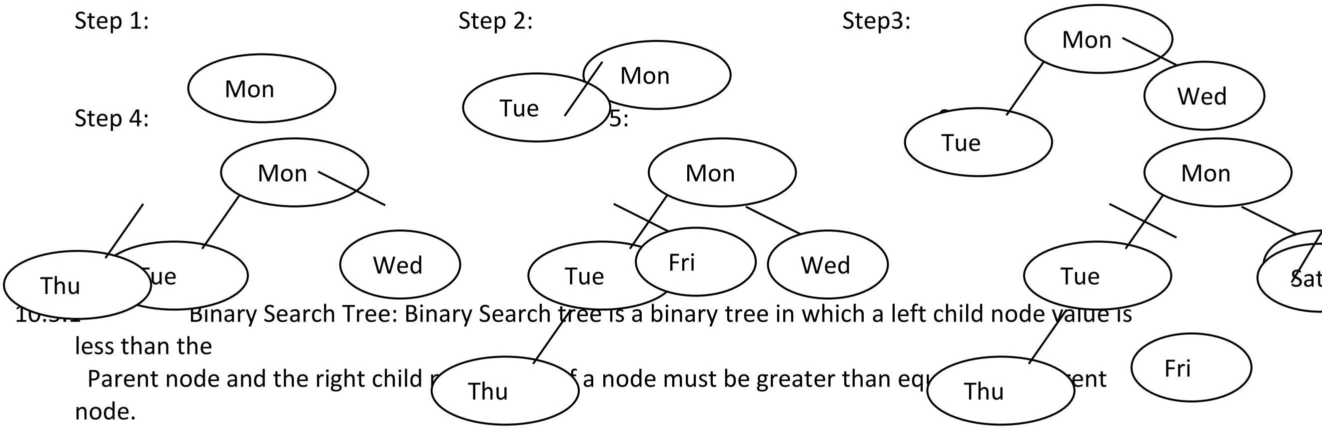


10.4 Construction of Binary Tree: The binary tree can be constructed with the help of following norms:

1. Start with the first value, become the root node.
2. The next value will be added at the last position of the tree.
3. Always first add the left child of the parent then the right child to the parent.

Problem: Construct Binary tree of Mon, Tue, Wed, Thu, Fri, Sat

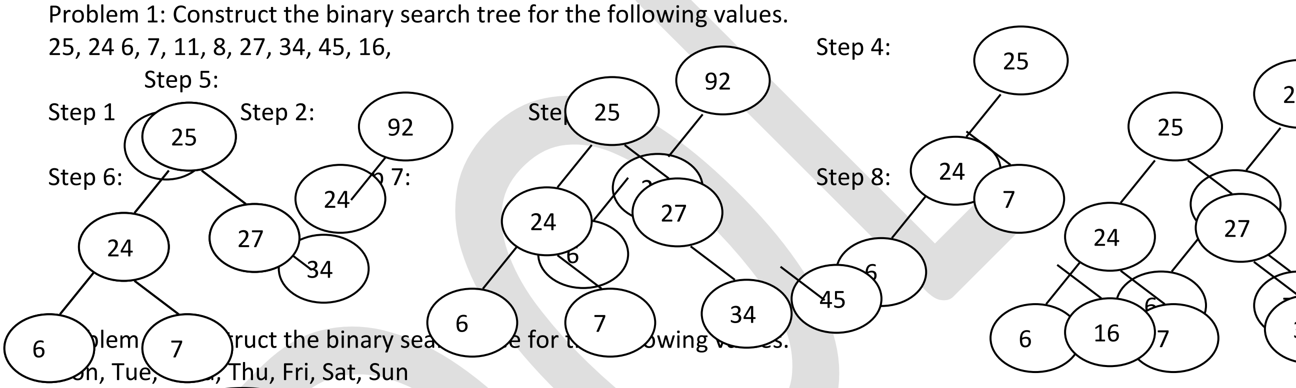
Step 1: Step 2: Step3: Step 4: Step 5: Step 6:



10.5.1 Binary Search Tree: Binary Search tree is a binary tree in which a left child node value is less than the
Parent node and the right child n… f a node must be greater than eq… ent node.

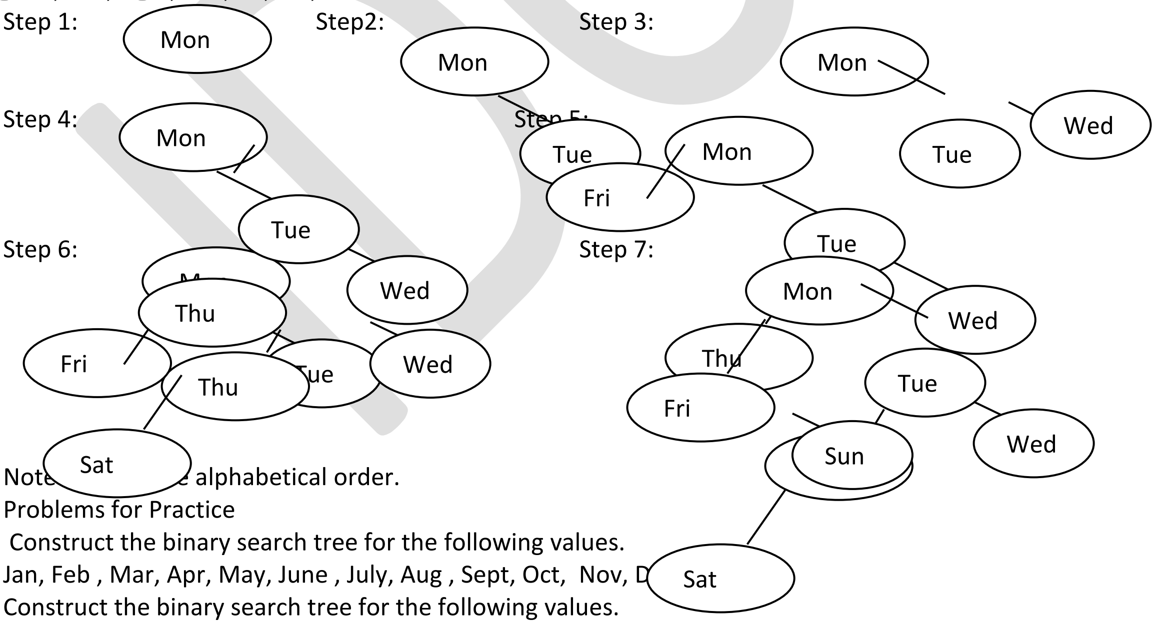
Problem 1: Construct the binary search tree for the following values.
25, 24 6, 7, 11, 8, 27, 34, 45, 16,

Step 1 Step 2: Step 3: Step 4: Step 5: Step 6: Step 7: Step 8:



…blem … truct the binary sea… e for t… owing values.
…on, Tue, …, Thu, Fri, Sat, Sun

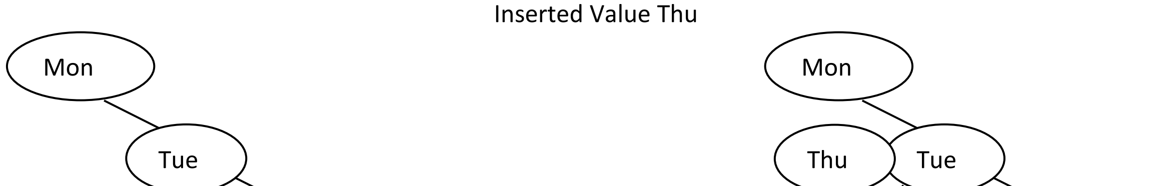
Step 1: Step2: Step 3: Step 4: Step 5: Step 6: Step 7:



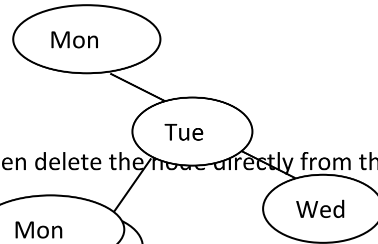
Note … alphabetical order.
Problems for Practice
Construct the binary search tree for the following values.
Jan, Feb , Mar, Apr, May, June , July, Aug , Sept, Oct, Nov, D…
Construct the binary search tree for the following values.
Dec, Nov, Oct, Sept, Aug, July, June , May, Apr, Mar, Feb, Jan

10.5.2 Operations on Binary Search Tree: The following operations can be performed on Binary Search.
How to insert a node in a binary search tree: If the new node value is less than the parent node value then the new node value will be the part of left sub-tree otherwise right sub-tree.

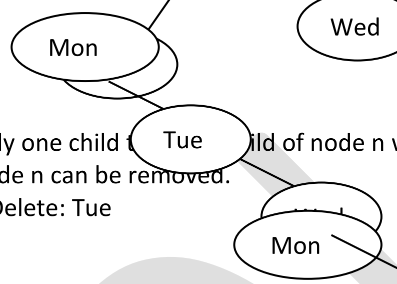
Inserted Value Thu



How to delete a node from a binary search tree: Any node can be deleted from the bellow binary search tree.

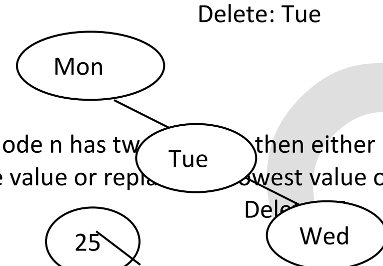


If the deleted node is a leaf node then delete the node directly from the binary tree.
Delete Thu :



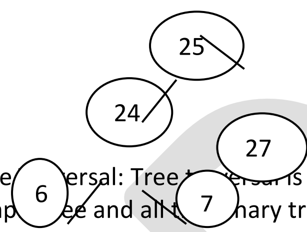
If the node to be deleted has only one child t[illegible]ild of node n will be connected with the parent of the node n and the node n can be removed.

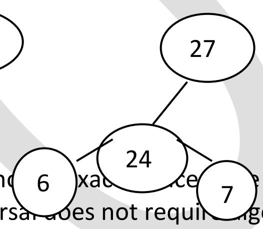
Delete: Tue



If a deleted node n has tw[illegible] then either replace the highe[illegible] left sub-tree from the deleted node value or repla[illegible]owest value of right sub-tree from the deleted node value.

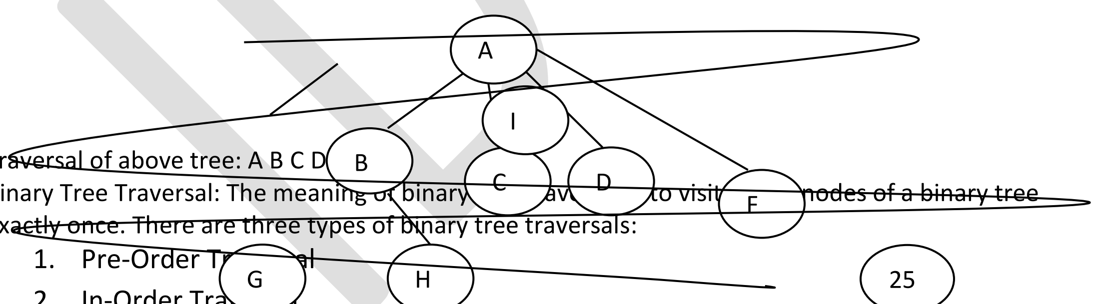
Dele[illegible]





Replaced by lowest value from RST

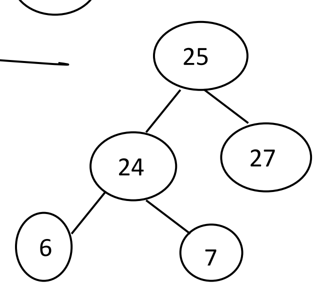
10.6Tree [illegible]ersal: Tree t[illegible] is to visit all the no[illegible]xac[illegible]ce[illegible] traversal is possible only for Simp[illegible]ee and all t[illegible]nary trees. Tree traversal does not requir[illegible]lgorithm. In case of Binary tree the traverse will start from root not and covers all the nodes from left to right.



Traversal of above tree: A B C D

Binary Tree Traversal: The meaning of binary [illegible]ave[illegible] to visit[illegible] nodes of a binary tree exactly once. There are three types of binary tree traversals:

1. Pre-Order Tr[illegible]al
2. In-Order Trav[illegible]
3. Post-Order Traversal



Dig 8

The Algorithms of Binary Tree Traversals:

Pre-Order Traversal

1. Visit the root node.
2. Traverse the left sub-tree in Pre-Order.
3. Traverse the Right sub-tree in Pre-Order.

Pre Order Traversal of tree in dig 8: 25, 24, 6, 7, 27

In-Order Traversal

1. Traverse the left sub-tree in In-Order.
2. Visit the root node.
3. Traverse the Right sub-tree in In-Order.

In Order Traversal of tree in dig 8: 6, 24, 7, 25, 27

Post-Order Traversal

1. Traverse the left sub-tree in In-Order.
2. Traverse the right sub-tree in In-Order.
3. Visit the root node.

Post Order Traversal of tree in dig 8: 6, 7, 24 , 27, 25

Problems for Practice:

Problem 1: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

92, 24 6,7,11,8,22,4,5,16,19,20,78

Problem 2: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Mon, Tue, Wed, Thu, Fri, Sat, Sun

Problem 3: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Jan, Feb , Mar, Apr, May, June , July, Aug , Sept, Oct, Nov, Dec.

Problem 4: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Dec, Nov, Oct, Sept, Aug, July, June , May, Apr, Mar, Feb, Jan

10.7 Construction of Binary Tree from the Traversal of the tree:

If any of the traversal pre-order or post-order is given the tree can be constructed using following method:

Pre- Oder and In-Order

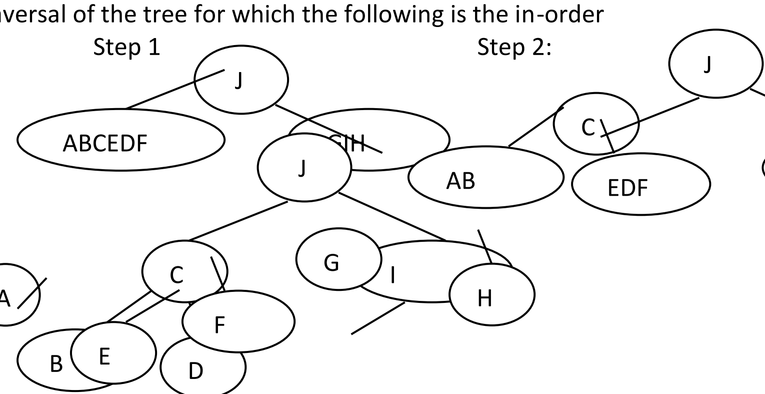
1. The first value of pre-order traversal becomes the root node value.
2. All the values lying left to the same value in in-order will be the part of left sub-tree and the values which are right to the in-order of the same value will be part of right sub-tree.

Problem: Construct a binary tree of a traversal of the tree for which the following is the in-order and Pre-order Traversal.

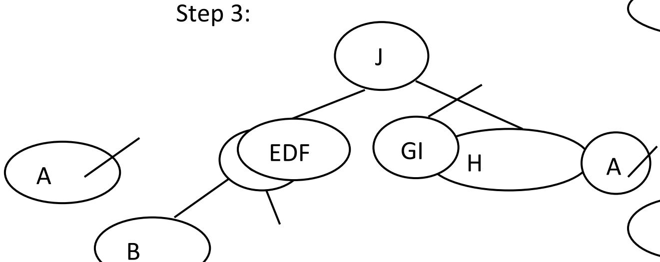
In-Order: A B C E D F J G I H

Pre-Order: J C B A D E F I G H

Step 1 Step 2:



Step 3:

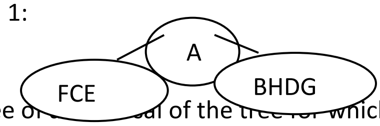


Problem: Construct a binary tree of a traversal of the tree for which the following is the in-order and Post-order Traversal.

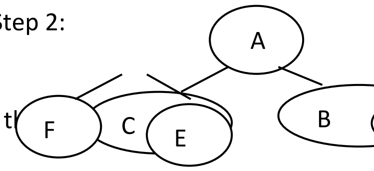
Post-Order: F E C H G D B A

In-Order: F C E A B H D G

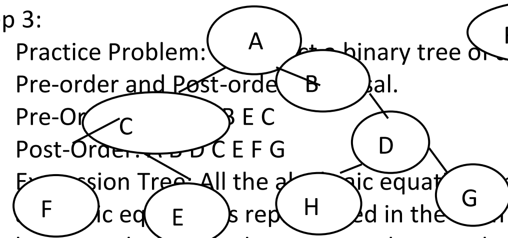
Step 1:



Step 2:



Step 3:



Practice Problem: [illegible] binary tree of [illegible] of the tree for which th[illegible]

Pre-order and Post-orde[illegible]sal.

Pre-Or[illegible] B E C

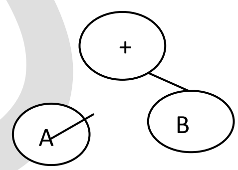
Post-Order: [illegible] D C E F G

10.8 E[illegible]sion Tree: All the al[illegible]ic equat[illegible]an be represented in the form of binary tree. An [illegible]ic eq[illegible]s rep[illegible]ed in the [illegible] of infix notation in which the operator is coming between the operands. For example A+B where A and B are the operands and + is an operator which is coming between operands A and B. While representing an algebraic equation in the form of binary tree, the entire operator will be either root node or internal nodes and all the operands will be the leaf nodes.

Expression tree satisfy following properties:

1. It represents an algebraic equation in the form of binary tree.
2. All the operators will be either root node or an internal node and all the operands will always be the leaf nodes.
3. Expression is an infix notation of a binary tree.
4. If you traverse the expression tree in an in-order then it is converted in to an algebraic equation.

Algebraic equation A+B can be represented as



Problem:

How to construct an expression Tree:

Step1: Convert the algebraic equation either in to pre fix notation or postfix notation.

Step 2: By Prefix / Postfix notation identify the root.

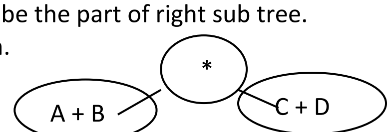
Step 3: All the values which comes left to prefix value in infix notation will be part of left sub tree and the values come to right to the prefix value in infix notation will be the part of right sub tree.

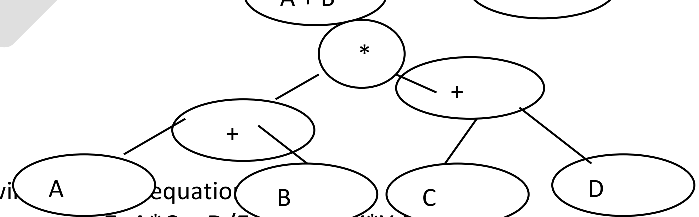
Construct an expression tree of the following algebraic equation.

A + B * C+ D

Infix Notation: A + B * C+ D

Prefix Notation: *+AB+CD





Construct an expression Tree for the followi[illegible] equatio[illegible]

A. A+B*C B. (A+B)*(C-D) E. A*C + D/F F. X*Y-Z*U

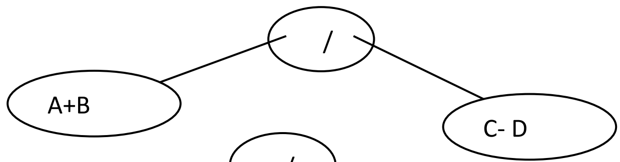
Construct an expression Tree for the following algebraic equation:

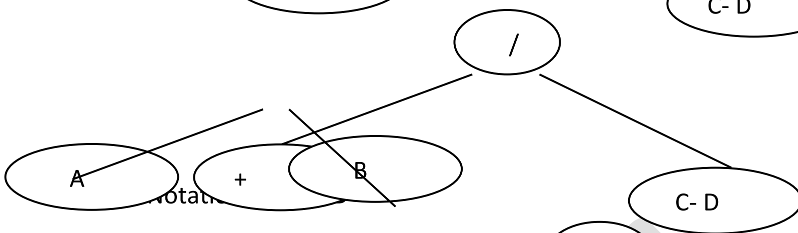
A. 2+P+Q*C B. (A-B)/(C+D) E. A*C * D/F F. P/Q+Z-U

Convert the expression in prefix notation and post fix notation and then construct the tree.

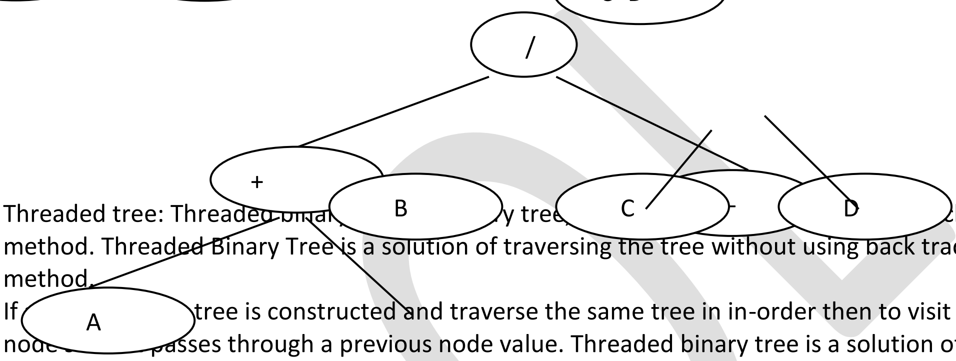
A. A/B*C B. (A+B*C)/(E-D) E. A*C * D/F F. X/Y-Z*U

Example: Infix Notation: (A+B)/(C-D)





Notatio



10.9 Threaded tree: Threaded binar... y tree... cking method. Threaded Binary Tree is a solution of traversing the tree without using back tracking method.

If ... tree is constructed and traverse the same tree in in-order then to visit the next node ... passes through a previous node value. Threaded binary tree is a solution of back tracking method in which while traversing the tree in in-order, it will not pass through the previous node value.

Process: Traverse the binary tree in in-order and mark all the successor node of all the leaf node of in-order traversal tree of the same tree. These leaf nodes will be connected to their successor node value by a thread which is redirecting the leaf node value to connect with successor node and avoid going to previous node value.

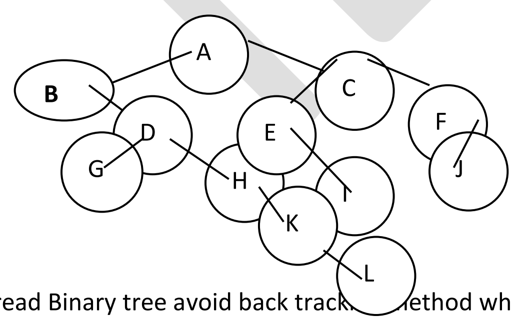
In-order: B G D H K L A E I C J F

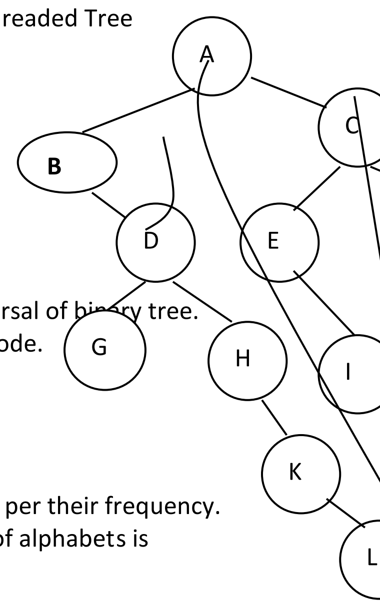
Leaf Nodes: G L I J

Successor Leaf nodes : G-> D; L->A I-> C J->F

Therefore G will be connected with D by a threaded. Similarly L by A , I by C and J by F.

Binary Threaded Tree





Thread Binary tree avoid back track... method which is a drawback of traversal of binary tree. Threads are proving an alternate path to move from one node to another node.

10.10 Huffman tree: Huffman Tree is a representation of alphabets or numbers as per their frequency. Many times we see an alphabet is repeated multiple times or combination of alphabets is repeated in the same combination. Huffman coding is a lossless data compression technique. In this all the alphabets are assigned variable length unique code by forming the tree. This tree is called Huffman Tree.

How to construct a Huffman Tree:

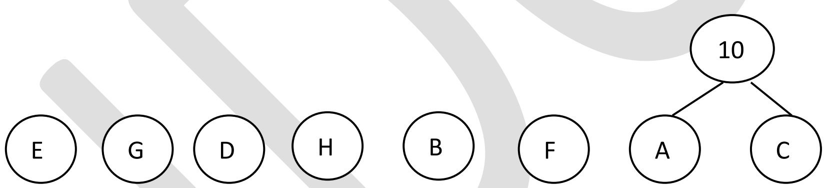
1. Arrange all the alphabets in ascending or descending order and place them a leaf node of a tree.
2. Add two lowest frequency nodes and form a new node which will be one level up. The value of this new node will be the addition of both the nodes which are added.
3. Keep adding two lowest frequency nodes which are not yet added.
4. Repeat step number three un till a root is formed.
5. Assign 0 to all the left branch of the tree and 1 to all the right branch of the tree.
6. A code of each alphabet is identified by reading the branch values from top to bottom which is a unique path from root to the leaf node.

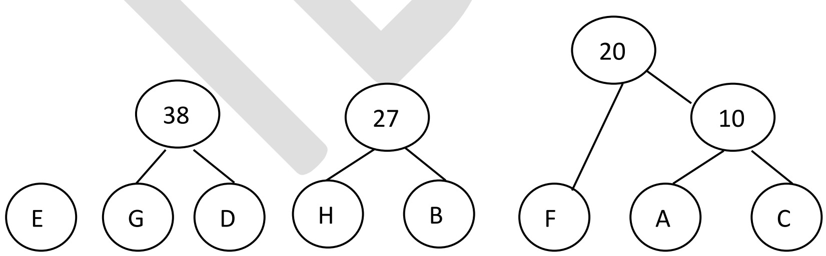
Problem: Construct Huffman Tree for the following data values:

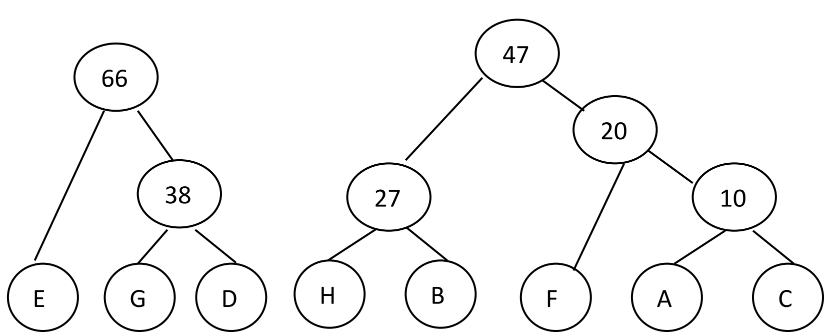
Symbol	A	B	C	D	E	F	G	H
Frequency	6	13	4	18	28	10	20	14

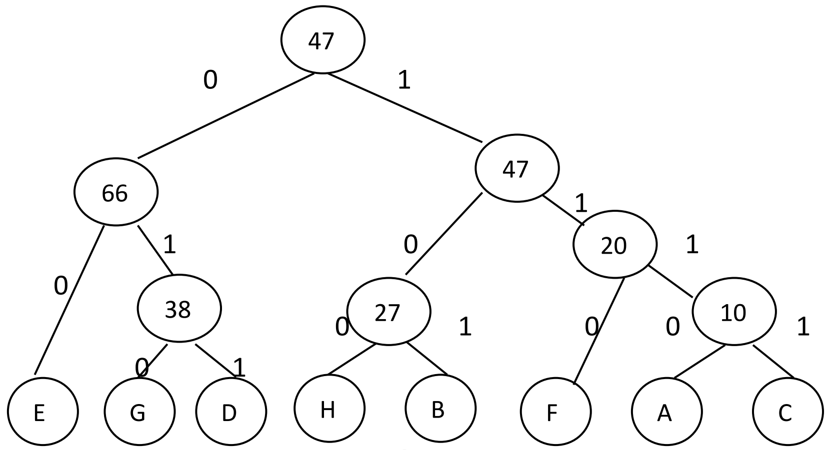
Arrange all the alphabets in descending order as per their frequency:

E	G	D	H	B	F	A	C
28	20	18	14	13	10	6	4









Huffman Code:

E: 00	D: 011	B: 101	A: 1110
G: 010	H:100	F: 110	C:1111

10.11 AVL tree: AVL Tree is identified by Adelson Velskii and Landis. That is why it is called as AVL Tree.

AVL tree is a solution of a binary search tree which is constructed for either all the values of increasing order or decreasing order. It has following problems:

1. When a binary search tree is constructed of ascending or descending values it will go one side either left hand side or right hand side.
2. The tree will not look like a tree at the same time and it will not be balanced.

Therefore AVL Tree is a solution to overcome from both the above problems.

AVL Tree is a binary search tree in which the difference between the height of left sub tree and the height of right sub tree must be less than equal to one. The condition can be represented by following equation.

$$H_L \sim H_R <= 1$$

H_L is Height of Left sub tree

H_R is Height of Right sub-tree.

Since the tree will be balanced therefore it is called as Height Balance tree or Balanced Binary Search Tree.

AVL Tree must satisfy the following properties:

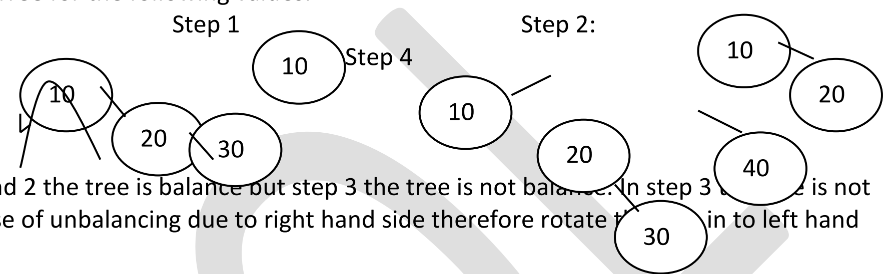
1. The difference between the height of left sub tree and the height of right sub tree must be less than equal to one.
2. A new node can be added to the AVL tree only if the tree is balanced.
3. All the nodes must have balance factor either 0 or 1.

How to Balance Binary search tree: Use rotation by finding the unbalancing is due to which sub tree.

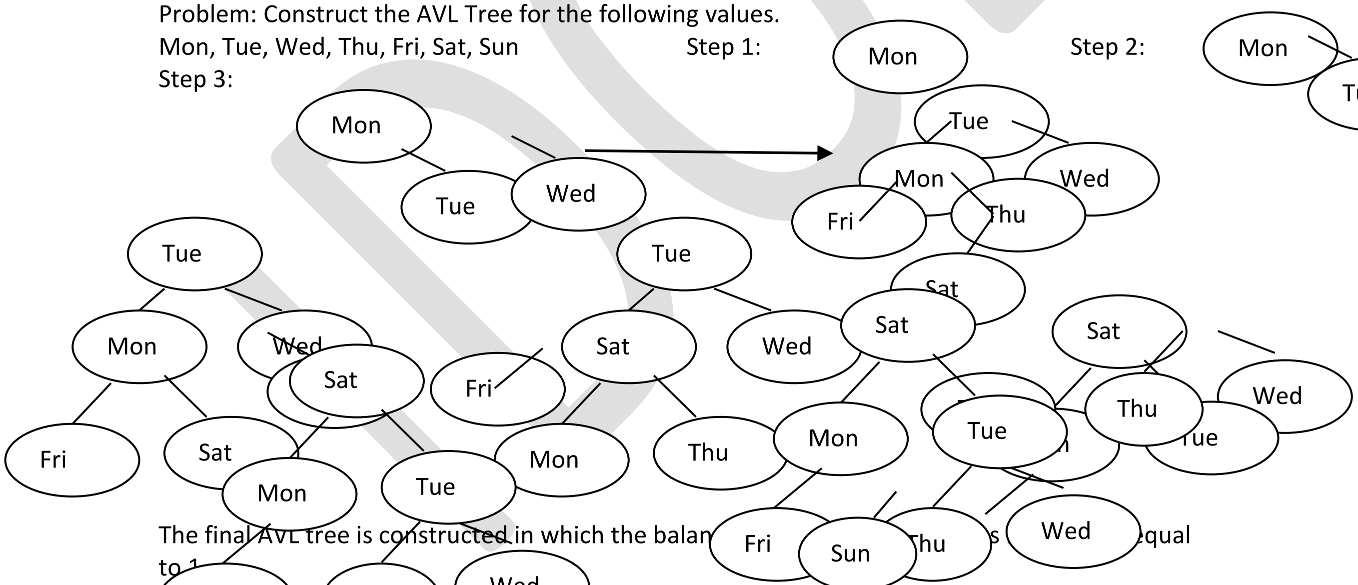
1. If the tree is unbalance due to the height of left sub tree then rotate the tree in to right hand side. Dig

2. If the tree is unbalance due to the height of right sub tree then rotate the tree in to left hand side. Dig
3. If the tree is unbalance due to the right sub tree of left sub tree then rotate the tree first left then right hand side. [In this case there will be multiple rotations to balance the tree.]
4. If the tree is unbalance due to the left sub tree of right sub tree then rotate the tree first right then left hand side. [In this case there will be multiple rotations to balance the tree.]
5. Dig 14.1

Construct AVL Tree for the following values:
10 20 30 40 Step 1 Step 2:
Step 3: Step 4



Note: Step 1 and 2 the tree is balance but step 3 the tree is not bal... In step 3 ... is not balance because of unbalancing due to right hand side therefore rotate t... in to left hand side.
The AVL Tree is constructed at step 4.
Problem: Construct the AVL Tree for the following values.
Mon, Tue, Wed, Thu, Fri, Sat, Sun Step 1: Step 2:
Step 3:



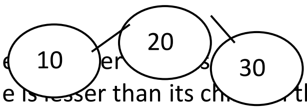
The final AVL tree is constructed in which the balan... s ... equal to 1.
...tio... an... to left of right sub tree then first rotate the sub tree in to right side than rotate the sub tree in to left side and if unbalancing is due to right of left sub tree then first rotate the sub tree in to left side than rotate the sub tree in to right.
Practice Problems:

- Construct and AVL Tree for the following values.
  92, 24 6,7,11,8,22,4,5,16,19,20,78
- Construct and AVL Tree for the following values.
  Sun, Mon, Tue, Wed, Thu, Fri, Sat
- Construct and AVL Tree for the following values.
  Jan, Feb , Mar, Apr, May, June , July, Aug , Sept, Oct, Nov, Dec.

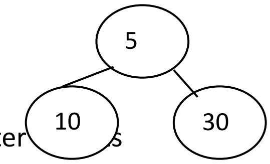
- Construct and AVL Tree for the following values.
  Dec, Nov, Oct, Sept, Aug, July, June , May, Apr, Mar, Feb, Jan

10.12.1 Heap: Heap Tree is binary tree in which parent node is either greater than its children or lesser than its children. There are two types of Heap Tree.

Max Heap



Min Heap



If the parent node is greater than its children then it is called as Max Heap.
If the parent node is lesser than its children then it is called as Min Heap.

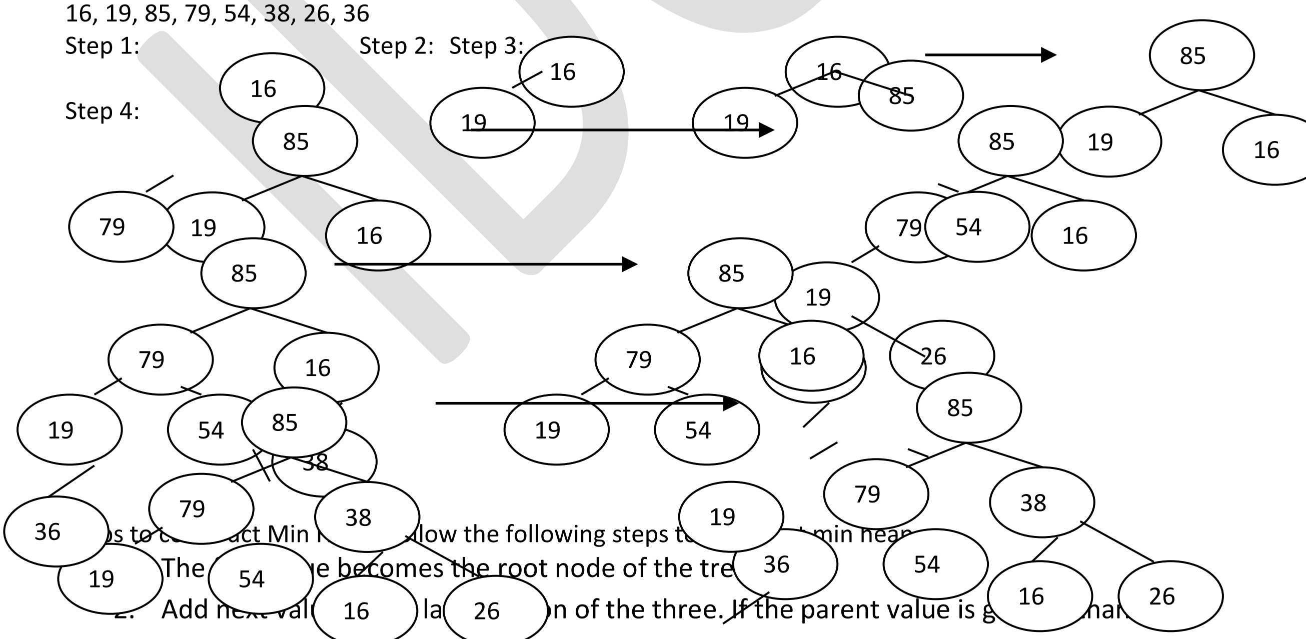
1. Max Heap: Max Heap Tree is binary tree in which parent node is greater than its children. Using max heap tree the values will be coming in descending order if the same heap tree is used for shorting.
2. Min Heap: Min Heap Tree is binary tree in which parent node is greater than its children. Using min heap tree the values will be coming in ascending order if the same heap tree is used for shorting.

10.12.2 How to construct a Heap Tree

Steps to construct Max Heap: Follow the following steps to construct max heap.

1. The first value becomes the root node of the tree.
2. Add next value at the last position of the three. If the parent value is lesser than its children the replace the parent node value to the greater child value and the tree is converted in to heap.
3. Repeat the step number 2 until all the values are added in to the heap tree.

Construct Max Heap for the following values:
16, 19, 85, 79, 54, 38, 26, 36



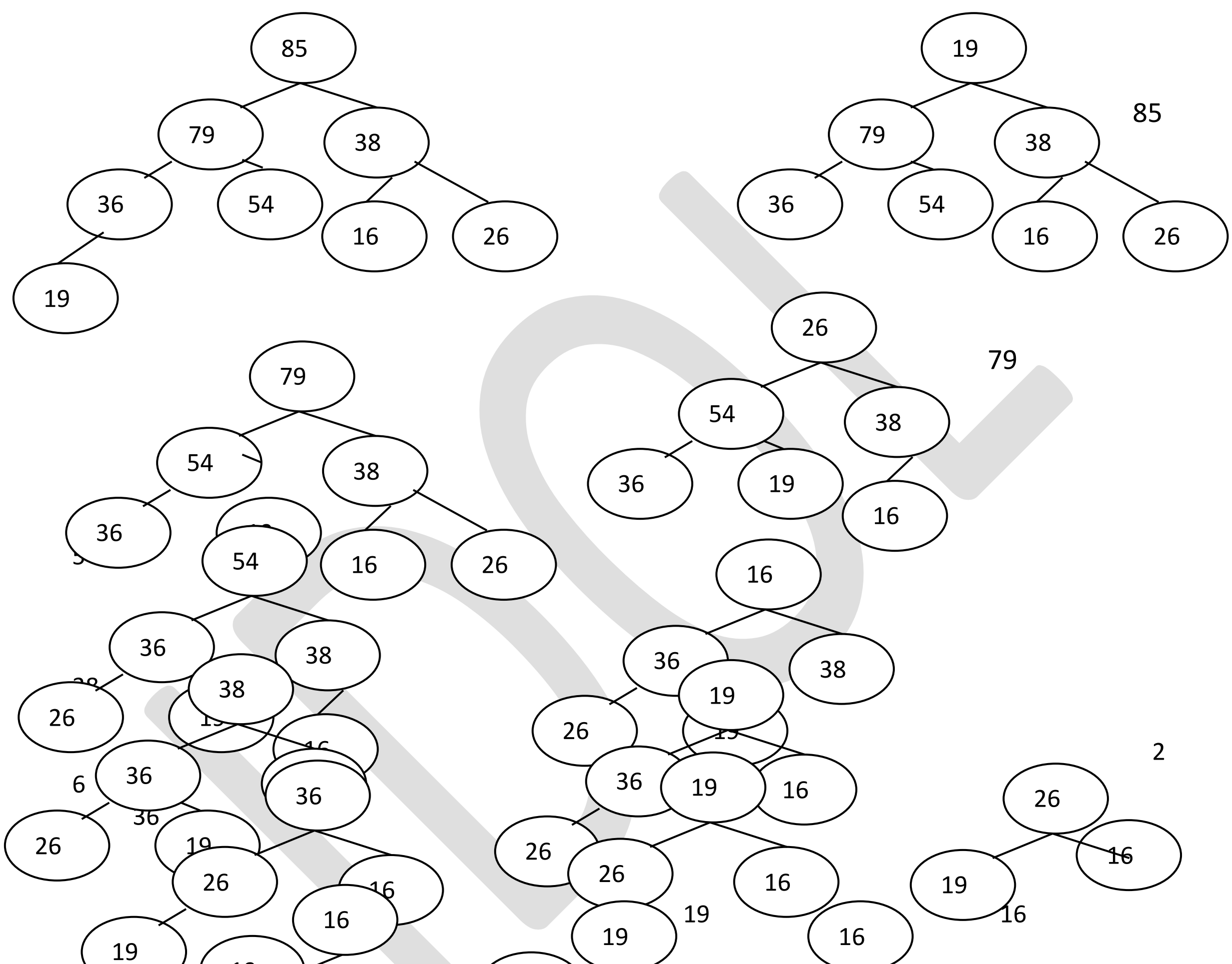
Steps to construct Min Heap: Follow the following steps to construct min heap.

1. The first value becomes the root node of the tree.
2. Add next value at the last position of the three. If the parent value is greater than its children the replace the parent node value to the lowest child value and the tree is converted in to heap.
3. Repeat the step number 2 until all the values are added in to the heap tree.

The way max heap is constructed; min heap can also be constructed similarly.

10.12.3 Steps to apply Heap Short:

1. Convert the tree in to heap.
2. Take out the value from the root.
3. Replace last node value from the root node value.
4. Go to step 1 until number of nodes are greater than one.



85, [illegible], 54, 3[illegible], 19, 16

Hence the values are in sorted order fro[illegible]ghest value to the lowest value. Max heap sort will arrange the values in descending order while min heap arrange the values in ascending order.

Reheap up and Reheap down: The concept of reheap up and reheap down is how the values are shifting upward direction or downward direction while constructing the heap tree. If the values are shifting upward direction then it is call Reheapup and if the values are shifting downward direction then it is call reheap down.

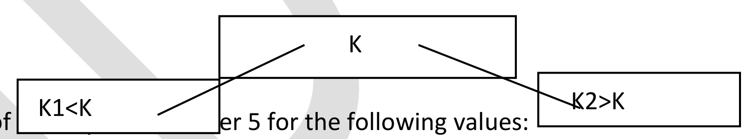
Unit 5: Chapter 11

M- Way Tree

11.1 M- Way Tree
11.1.1 M Way-Tree
11.1.2 Construction of M-Way Tree
11.2 B-Tree
11.2.1 B- Tree
11.2.2 Construction of B-Tree
11.3 B* Tree
11.3.1 B* Tree
11.3.2 Construct of B*- Tree
11.4 Deletion from B-Tree/ B* Tree
11.5 Similarities and Difference from B-Tree and B* Tree
11.5.1 Similarities in B-Tree and B* Tree
11.5.2 Difference from B-Tree and B* Tree
11.6 Practice Problem based on B-Tree and B* Tree

11.1.1 M-Way Tree: M-way tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

1. M-Way tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then the new key value will be either left child value of the parent node if it is lesser then its parent node value or right pointer child value if it is greater than its parent value.
3. A new key value will be added at the leaf node value.
4. The leaves may not be at the same level.



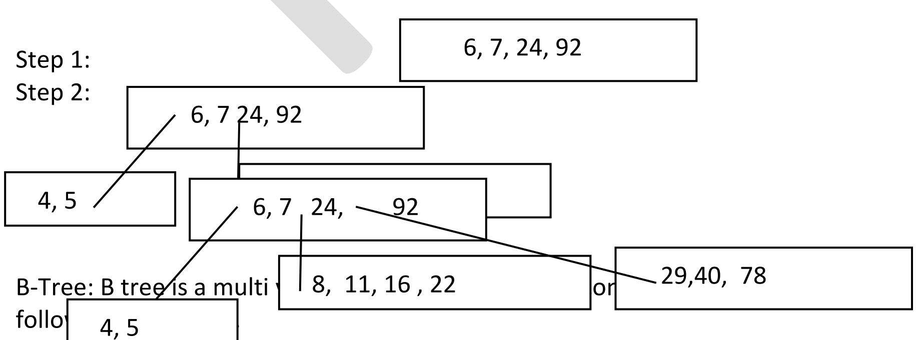
11.1.2 Construction of ... er 5 for the following values:

92, 24 6,7,11,8,22,4,5,16,29,40,78

Maximum Number of key values in a node: (m-1) = 5-1= 4

Step 1:

Step 2:



11.2.1 B-Tree: B tree is a multi v... or... satisfy the follo...

1. B tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then split the node. Select the median value which becomes the parent value. The values

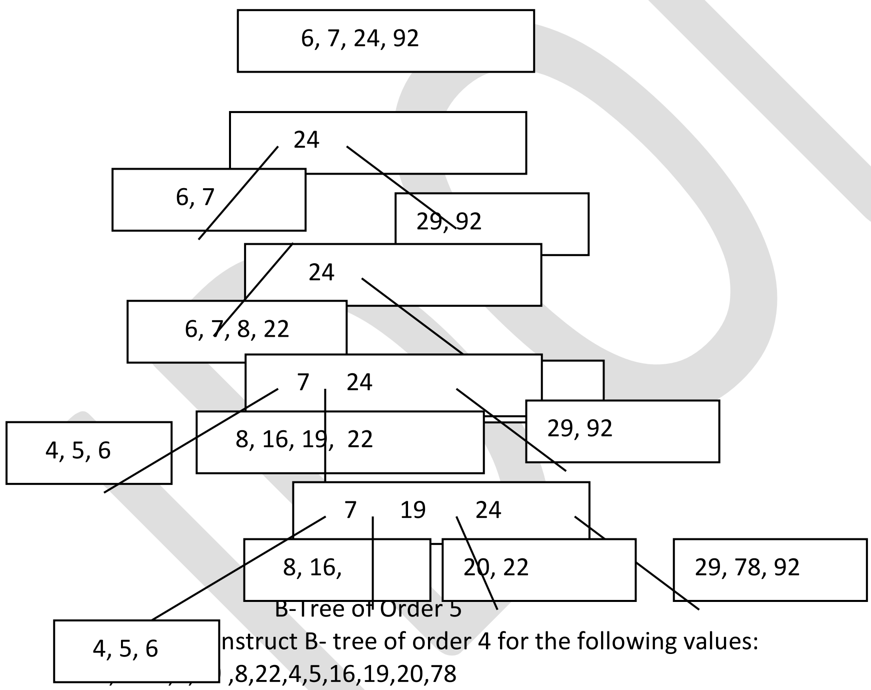
coming left of median value will become the left child of parent value and the values coming to right to the median value will become the right child of the median value. The new key value will be inserted either left child of the parent node (if it is lesser then its parent node value) or right child value of the parent node (if it is greater than its parent value).

3. A new key value will be added at the leaf node value.
4. The leaves must be at the same level.
5. The height of B-tree will always be lesser than M-Way Tree.
6. The new value will be inserted in the leaf node.

11.2.2 Construction of B Tree: Construct B- tree of order 5 for the following values:

92, 24 6,7,29 ,8,22,4,5,16,19,20,78

Maximum Number of Key values in a node= m-1=5-1=4

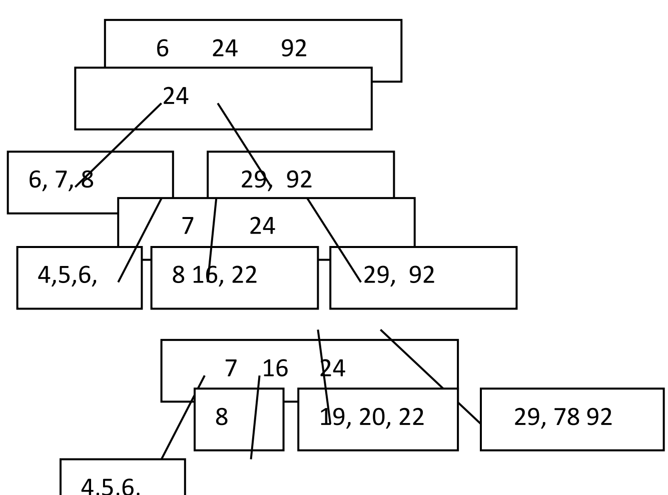


B-Tree of Order 5

nstruct B- tree of order 4 for the following values:

,8,22,4,5,16,19,20,78

Maximum Number of Key values in a node= m-1=4-1=3

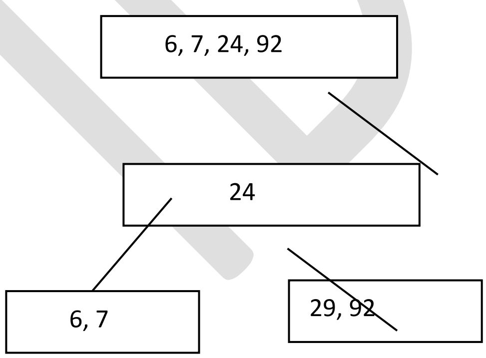


B Tree of Order 4

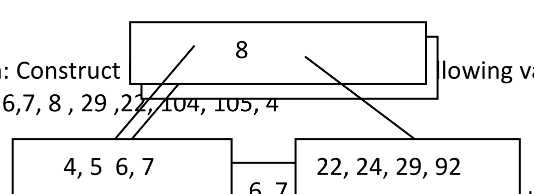
11.3.1 B*-Tree: B* tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

1. B* tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then split the node only if all the siblings are full. If all the siblings are full then only select the median value which becomes the parent value. The values coming left of median value will become the left child of parent value and the values coming to right to the median value will become the right child of the median value. The new key value will be inserted either left child of the parent node (if it is lesser then its parent node value) or right child value of the parent node (if it is greater than its parent value). If siblings are not full the rotate the values of the leaf node and make the place empty for the new key value.
3. A new key value will be added at the leaf node value.
4. The leaves must be at the same level.
5. The height of B*-tree will always be lesser than B Tree.
6. B* tree is referred for classification of topic. B* tree is also referred for the purpose of indexing.
7. The new value will be inserted in the leaf node.

11.3.2 Construction of B* Tree: Construct B*- tree of order 5 for the following values:
92, 24 6,7,29 ,8,22,4,5
Maximum Number of Key values in a node= m-1=5-1=4



Problem: Construct llowing values:
94, 25 6,7, 8 , 29 ,22, 104, 105, 4



Next valu values 6, 7, 8, 24, 92. 8 will become the root node value and the values which are coming left to 8 (6,7) will become the left child of root node and the values which are coming right to 8 (24, 82) will become the right child of median value.

8

Next value is 29. It will be inserted between 25 and 94.

6, 7,

25, 94

8

Next value is 22. It will be inserted before 25 as the node can consist of maximum 4 key values.

6, 7,

25,29 , 94

8

Next value is 104. It will be inserted after 94 as the node can consist of maximum 4 key values But the node is full th… ’t be inserted in to right c… can be inserted after rotating … de and 8 will come down …

6, 7,

22, 25, 29 , 94

22

Once the key values … ace is empty at rig…

6, 7, 8

7,

25,29 , 94, 104

Next value is 105. It … 4 as the node can … ut the node is full therefore 105 can’t be inserted in to right child of 8. Hence new value can be inserted after rotating 22 as a root node and 8 will come down to the left hand side.

25

Now the place is empty at the right child of 25 therefore 105 can be inserted in to the right child of 25.

6, 7, 8, 22

29, 94,104

25

Next value is 4 but both the siblings are full therefore break the left node of 25. Since 4 is the left most value of the same node then median value will become 7. 7 will move up and 4,6 will become the left child … become the right ch…

6, 7, 8,22

29, 94, 104, 105

7 25

Here the new value is inserted below.

6

8,22

29, 94, 104, 105

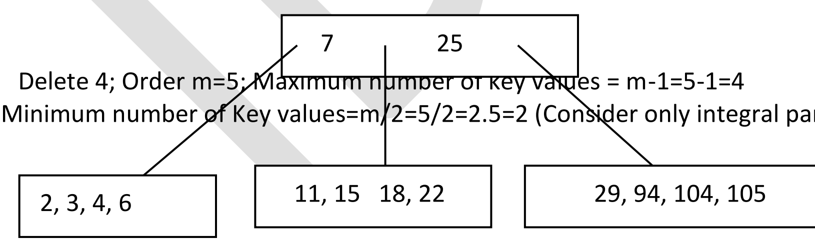
7 25

11.4 Deletion from B Tree:

Deletion from B tree is the deleting the key values from a node. Below are the rules to delete the key values from the node.

1. The key values are deleted from the leaf node only.
2. At a time only one key value is deleted.
3. The key value can't be deleted from root node and internal node.
4. Key value can't be deleted if the number of key values is less the m/2 key values.
5. If the key value which is available at leaf node is supposed to be deleted and the node have more than m/2 key values than delete the key value from the leaf node directly.
6. If the key value which is available at leaf node is supposed to be deleted and the node have m/2 key values or less than m/2 key values then merge the leaf node siblings and then delete the key value from the leaf node directly.
7. If the key value which is not available at leaf node is supposed to be deleted and the node have more than m/2 key values then move key value at the leaf node and then delete the key value from the leaf node directly.

Delete the following key values from the below B-Tree.



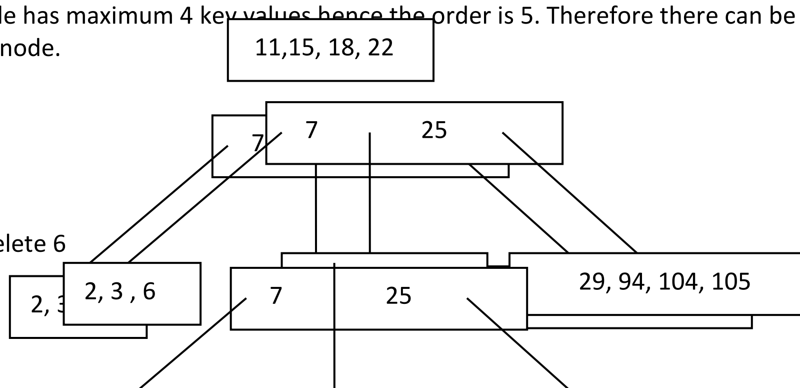
Delete 4; Order m=5; Maximum number of key values = m-1=5-1=4

Minimum number of Key values=m/2=5/2=2.5=2 (Consider only integral part)

Four is deleted from the leaf node directly since it was available at the leaf node and the node has more than m/2 Key values.

Since a node has maximum 4 key values hence the order is 5. Therefore there can be minimum 2 key values in a node.

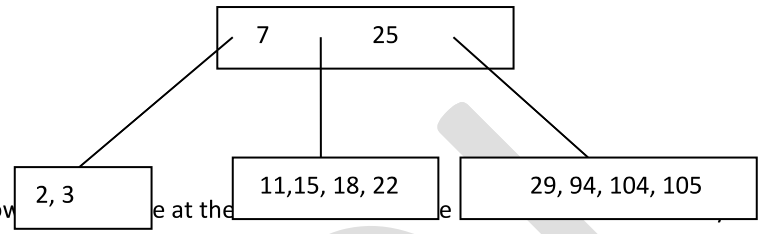
Delete 6



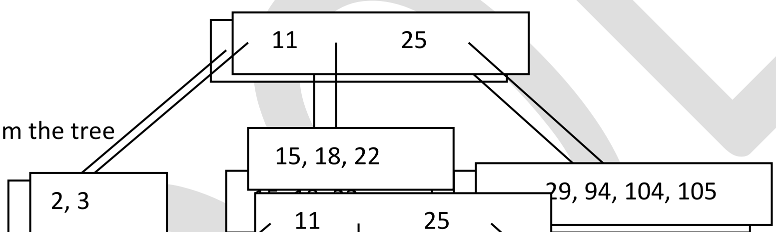
6 is deleted from the leaf node since it was available at the leaf node and number of key values in a node was more than m/2.

Delete 7

7 can't be deleted directly since it is not available at the leaf node therefore rotate the key values.

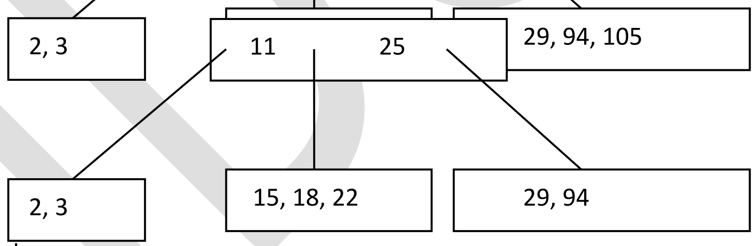


Now [illegible] e at the [illegible] e

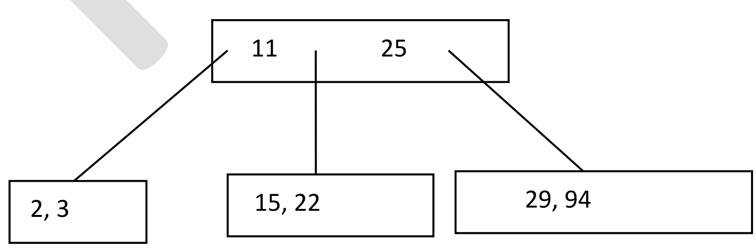


Delete 104 from the tree

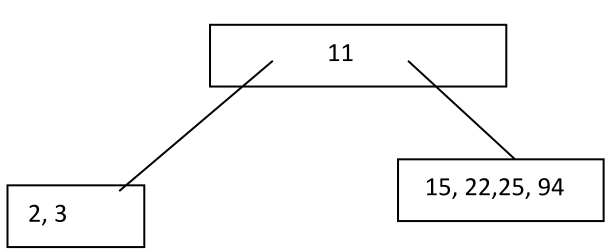
Delete 105 from the tree



Delete 18 from the tree

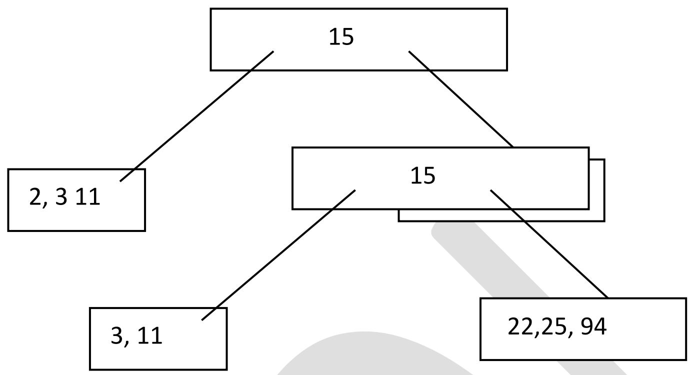


Delete 29 from the tree

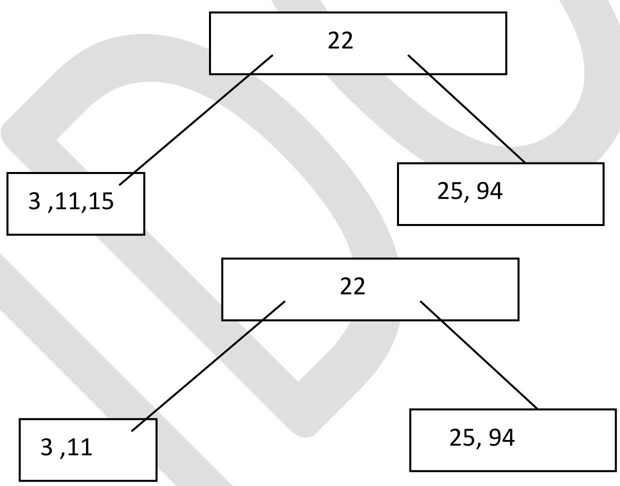


Since the node containing the value 29, will have less than m/2 key values therefore both the siblings will be merged.

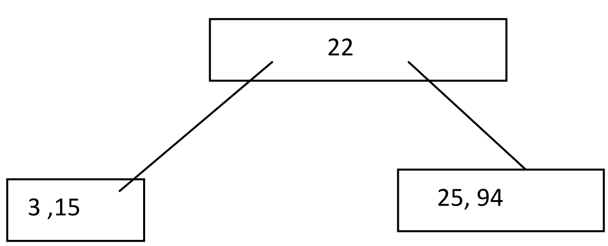
Delete 2 from the tree: First rotate the key values in which 11 will move down and 15 will move up.



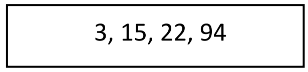
Delete 15 from the tree: 22 will move up and 15 will move down than 15 will be deleted from the leaf.



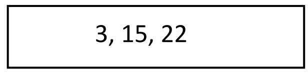
Delete 11 from the tree



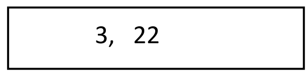
Delete 25 from the Tree: 25 is available at the leaf node but both siblings has m/2 key values therefore merger will take place.



Delete 94 from the Tree:



Delete 15 from the Tree:



Further values from the tree can't be deleted since the node is containing m/2 key values and neither rotation is possible nor merger is possible.

11.5 Similarities in B-Tree and B* Tree

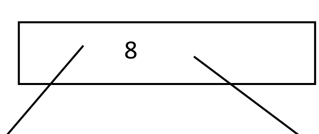
- Both have maximum m-1 key values in a node if the order of the tree is m.
- Both the trees can have maximum m pointers in a node if the order of the tree is m.
- Both the trees the key values in a node must be in increasing order.
- Both the trees leaves must be at the same level.
- At the time of deletion the rotation is possible in both the trees.

Difference in B-Tree and B* Tree

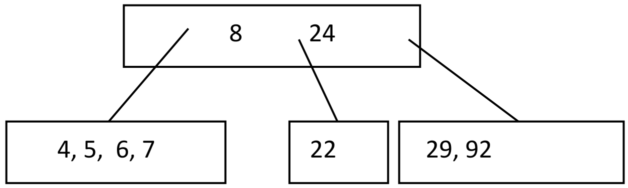
- The height of B*-Tree is either lesser or equal to the B-Tree for same number of key values and for the same order of the tree.
- In case of B* Tree, a node will have more number of key values compare to B Tree.
- B Tree, rotation of key values is not allowed while in B* Tree rotation of key values is allowed at the time of inserting the key values.
- We do not split the node in B*-Tree until sibling are full while B tree, we split the node once the node is full.

11.6 Practice Problems on B Tree and B* Tree

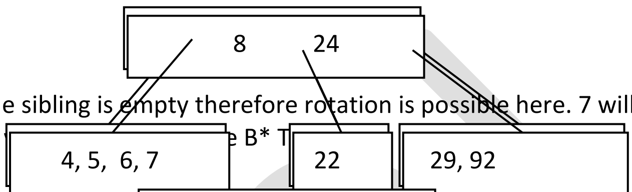
Problem: Insert 107 in the following B-tree of order 5.



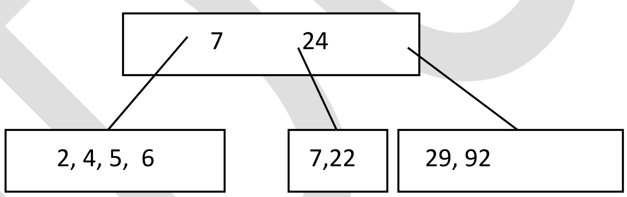
Solution: The value 107 can't be inserted directly in to the B tree of order 5 and a node can contain maximum 4 values in this case the node. Therefore split the right child node.



Problem: Insert 2 in the following B* Tree of Order 5.



Solution: Since the sibling is empty therefore rotation is possible here. 7 will move up and 8 will come down and then 2 [illegible] e B* T[illegible]



Unit 6: Chapter 12
Graphs

Introduction to Graphs

12.0 Objectives 274
12.1 Introduction 275
12.2 Basic Concepts of Graphs 275
12.2.1 Types of Graphs 277
12.2.2 Representing Graphs 278
12.2.2.1 Adjacency Matrix 278
12.2.2.2 Adjacency List: 280
12.2.2.3 Adjacency Multi-lists 281
12.2.2.4 Weighted edges 282
12.2.3 Operations on Graphs 282
12.2.3.1 Depth-First Search 282
12.2.3.2 Breadth-First Search 283
12.2.3.3 Connected Components 285
12.3 Graph Traversals 289
12.3.1 In-order 289
12.3.2 Pre-order 290
12.3.3 Post-order 290
12.4 Summary 291
12.5 Review Your Learning 291
12.6 Questions 291
12.7 Further Reading 292
12.8 References 292

12.0 Objectives

1. Explain basic graph terminologies.
2. Understand adjacency matrix and convert to graphs.
3. Describe various operations like BFS, DFS performed on graphs
4. Analyse the applications of graphs in day-to-day life

12.1 Introduction

A Graph in data structure is a type of non-linear data structure.Map is a well-established example of a graph. In a map, various cities are connected using links. These links can be considered as roads, railway lines or aerial network. Leonhard Euler was a scientist and he used graph theory to solve Seven Bridges of Konigsberg problem in 1736. He laid the foundations of Graph Theory idea of topology. The problem of Konigsberg bridge was to find whether there is a possible way to traverse every bridge exactly once.This is shown in below in figure (a) and is called as Euler's Tour.

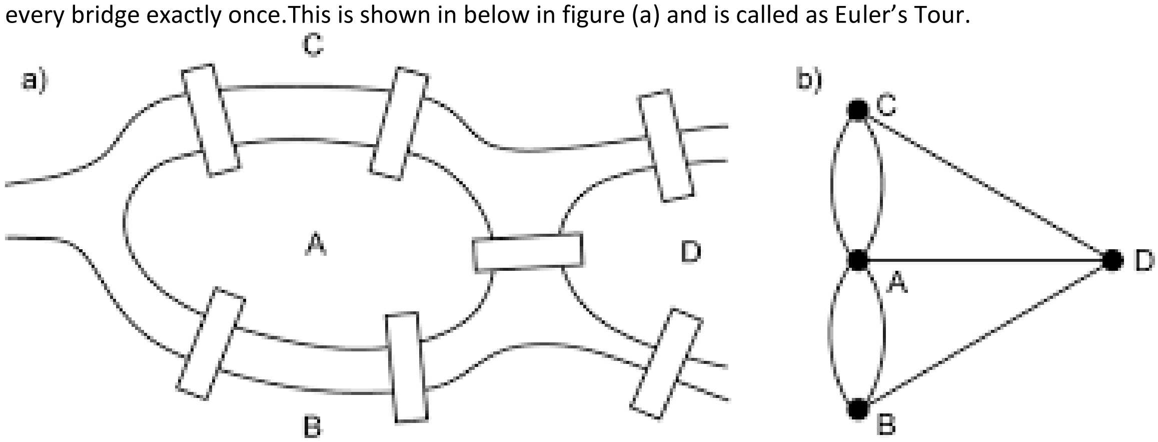


Figure: (a) Seven Bridges of Konigsberg and (b) Graphical Representation of figure (a)

As we can see the graphical representation of Konigsberg's seven bridges in figure b, here the points A, B, C and D are called as Vertex in graph terminologies and the paths joining to these vertices are called as edges.

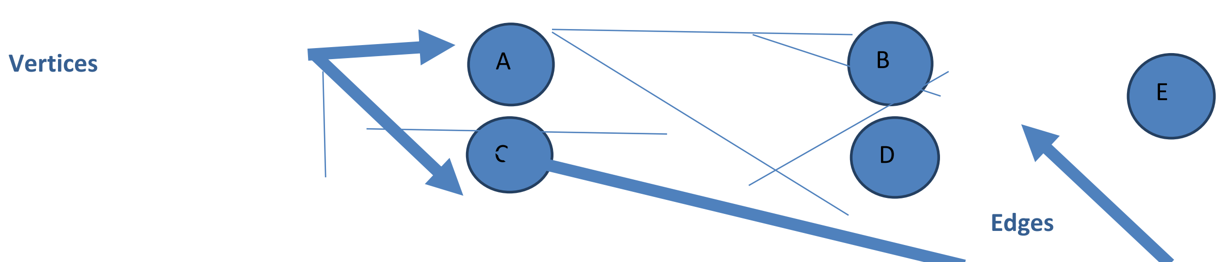
We can represent any graphical scenario with the help of graphs and find a solution for the same.

Applications of Graphs in real life:

1. Solving Electricity Distribution problem
2. Maps like Cities, Rivers, Countries and so on
3. Water distribution in various areas
4. CAD/CAM applications
5. Finding Disaster Relief Solutions

12.2 Basic Concepts of Graphs

Nodes / Vertices: A graph contains a set of points known as nodes or vertices
Edge / Link / Arc: A link joining any two-vertex known as edge or Arc.
Graph:A graph is a collection of vertices and arcs which connects vertices in the graph.
A graph G is represented as G = (V, E), where V is set of vertices and E is set of edges.



Example: Graph G can be defined as G = (V, E) where,
V = {A,B,C,D,E} and
E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

This is a graph with 5 vertices and 6 edges.

Graph Terminology

1. **Vertex**: An individual data element of a graph is called as Vertex. Vertex is also known as node.
   In above example graph, A, B, C, D & E are known as vertices.
2. **Edge**: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).
   Example: In above graph, the link between vertices A and B is represented as (A,B)

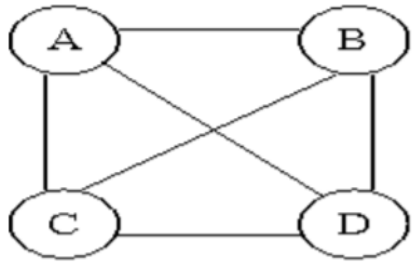
   Edges are of three types:
   a. **Undirected Edge**- An undirected edge is a bidirectional edge. If there is an undirected edge between vertices Aand B then edge (A, B) is equal to edge (B, A).
   b. **Directed Edge -** A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
   c. **Weighted Edge -** A weighted edge is an edge with cost as weight on it.

3. **Degree of a Vertex**: The degree of a vertex is said to the number of edges incident on it.

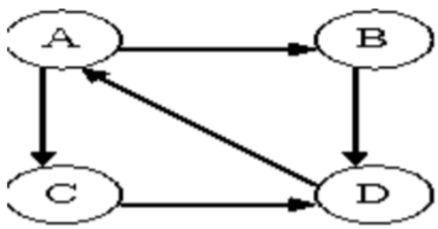
   Euler showed that there is a path beginning at any vertex, going through each edgeexactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian.
   Ex: There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.
4. Outgoing Edge: A directed edge is said to be outgoing edge on its origin vertex.
5. Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.
6. Degree: Total number of edges connected to a vertex is said to be degree of that vertex.
7. Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
8. Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
9. Parallel edges or Multiple edges: If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
10. Self-loop: An edge (undirected or directed) is a self-loop if its two endpoints coincide.
11. Simple Graph
12. A graph is said to be simple if there are no parallel and self-loop edges.

12.2.1 Types of Graphs

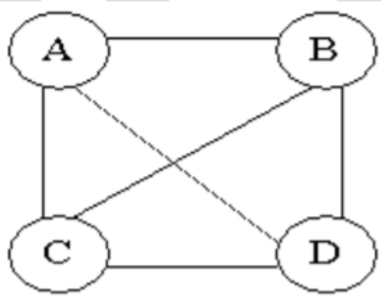
1. **Undirected Graph**: A graph with only undirected edges is said to be undirected graph.



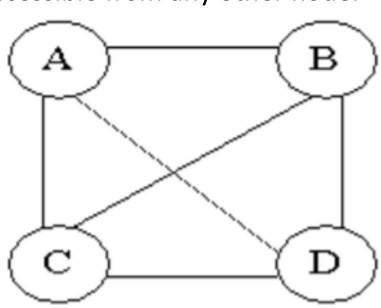
2. **Directed Graph:**A graph with only directed edges is said to be directed graph.



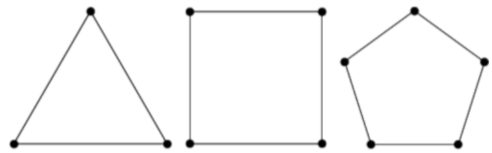
3. **Complete Graph:** A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



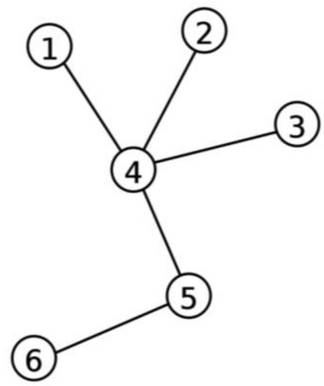
4. **Regular Graph:** Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



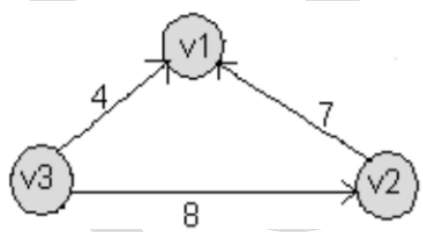
5. **Cycle Graph:** A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



6. **Acyclic Graph:** A graph without cycle is called acyclic graphs.



7. **Weighted Graph:** A graph is said to be weighted if there are some non-negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



12.2.2 Representing Graphs

Graph data structure is represented using following representation types:
1. Adjacency Matrix
2. Adjacency List
3. Adjacency Multi-list

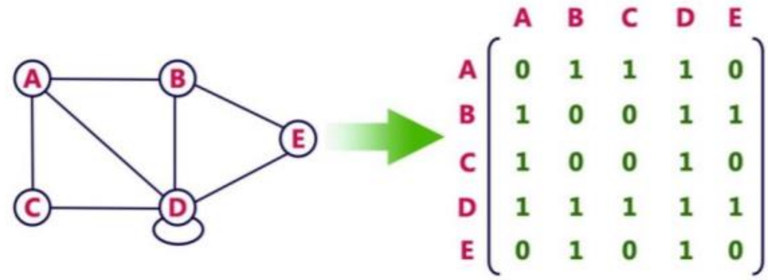
10.2.2.1Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

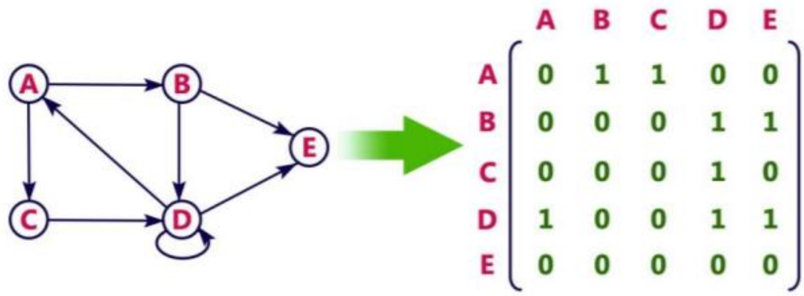
In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix: Let $G = (V, E)$ with n vertices, $n \geq 1$. The adjacency matrix of G is a 2-dimensional $n \times n$ matrix, A, $A(i, j) = 1$ iff $(v_i , v_j) \in E(G)$ ($\langle v_i , v_j \rangle$ for a diagraph), $A(i, j) = 0$ otherwise.

Example: For undirected graph



For a Directed graph



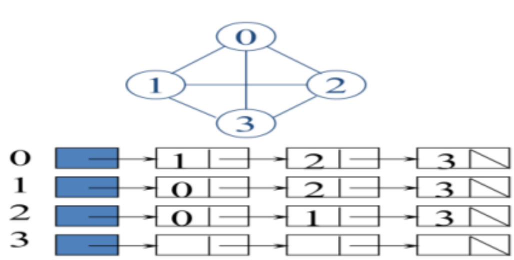
The adjacency matrix for an undirected graph is symmetric but the adjacency matrix for a digraph need not be symmetric.
Merits of Adjacency Matrix: From the adjacency matrix, to determine the connection of vertices is easy.
The degree of a vertex is

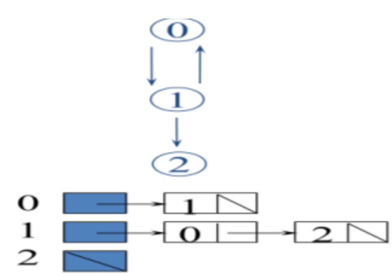
$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree.

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.
2. Adjacency List
In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.
It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:



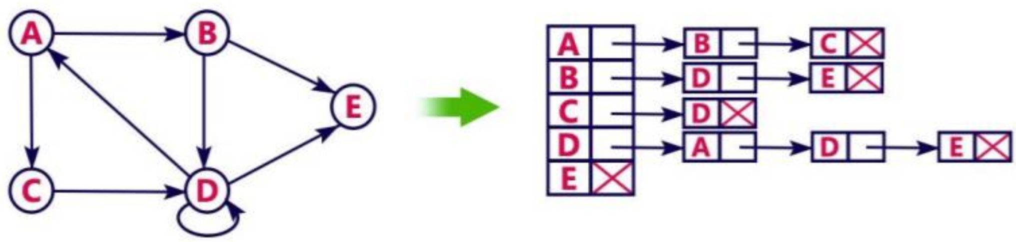


So that we can access the adjacency list for any vertex in O(1) time.
Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is
#define MAX_VERTICES 50

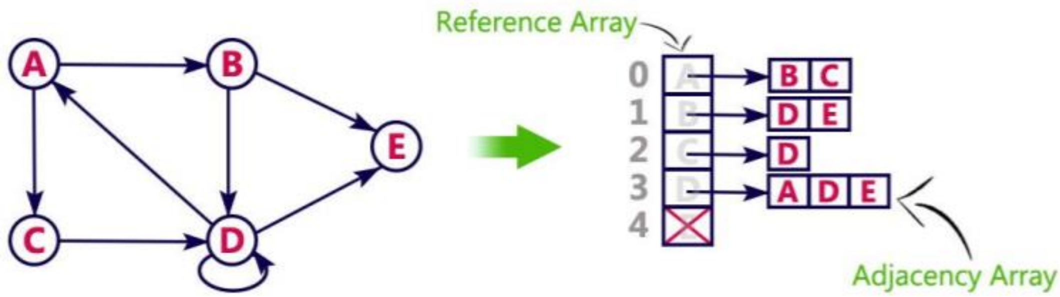
```
typedef struct node *node_pointer;
typedef struct node { int vertex; struct node *link; };
node_pointergraph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Another type of representation is given below.

Example: Consider the following directed graph representation implemented using linked list.



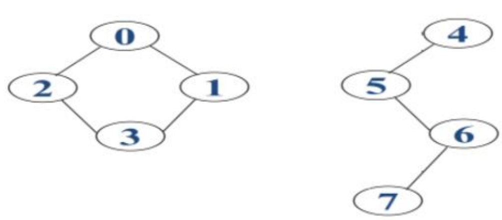
This representation can also be implemented using array



12.2.2.2 Adjacency List:

Sequential representation of adjacency list and its conversion to graph is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

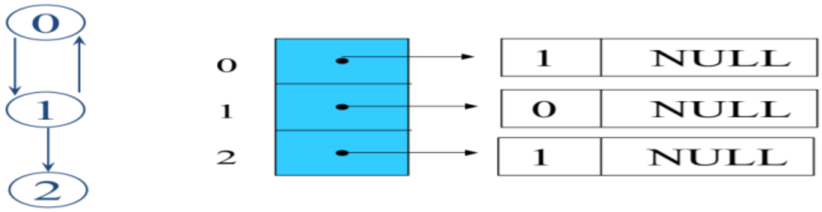


Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node I are stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined by counting the number of nodes in its adjacency list.

To find in-degree of vertex v, we must traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



Determine in-degree of a vertex in a fast way.

12.2.2.3 Adjacency Multi-lists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one onthe list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determine ie ~ ndenty for a particular edge and mark that edge as having been examined. This can be accomplished easilyif the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among severallists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists foreach of the two nodes to which it is incident).

For adjacency multilists, node structure is

```
typedef struct edge *edge_pointer;
typedef struct edge {
 short int marked;
 int vertex1, vertex2;
edge_pointer path1, path2;
};
edge_pointergraph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

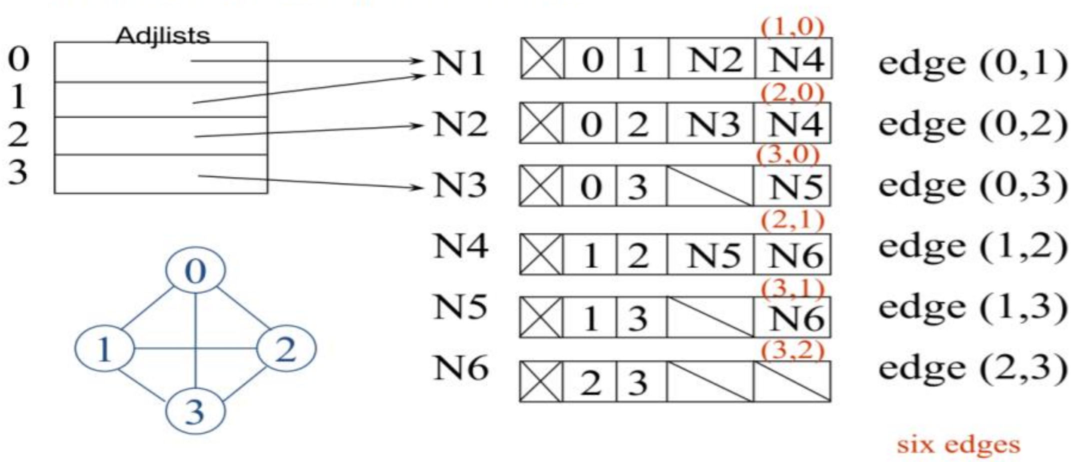
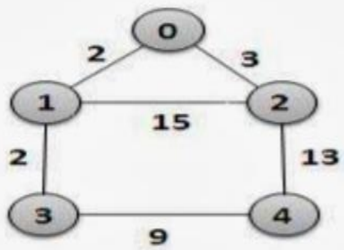


Figure: Adjacency multilists for given graph

12.2.2.4 Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list's nodes by including an additional field weight. A graph with weighted edges is called a network.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

12.2.3 Operations on Graphs

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

12.2.3.1 Depth-First Search

- Begin the search by visiting the start vertex v
  - o If v has an unvisited neighbor, traverse it recursively
  - o Otherwise, backtrack
- Time complexity
  - o Adjacency list: O(|E|)
  - o Adjacency matrix: O(|V|2)

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w.

The search terminates when no unvisited vertex can be reached from any of the visited vertices. DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops.

We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.
This function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
int visited[MAX_VERTICES];
void dfs(int v)
{
node_pointerw;
 visited[v]= TRUE;
printf("%d", v);
 for (w=graph[v]; w; w=w->link)
 if (!visited[w->vertex])
dfs(w->vertex);
}
```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depthfirst search is initiated from vertex 0 then the vertices of G are visited in the following order: 0 1 3 7 4 5 2 6.
Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

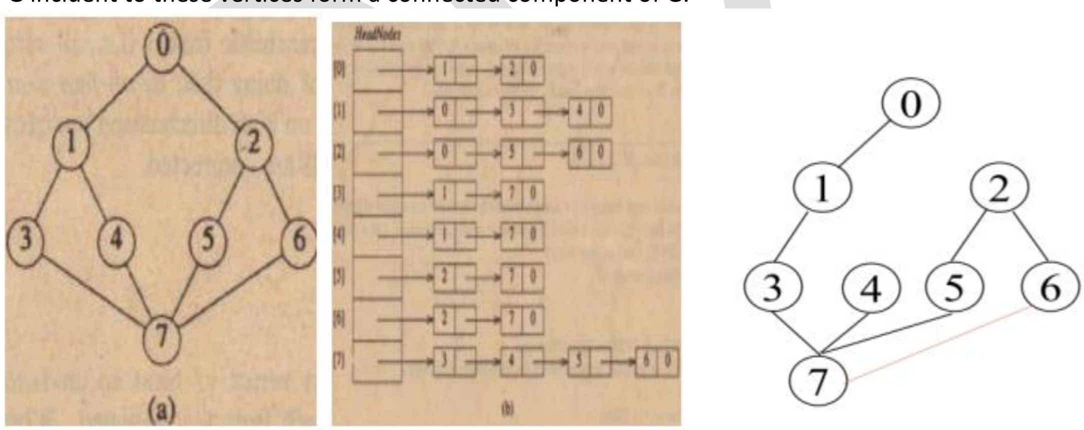


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n2).

12.2.3.2 Breadth-First Search

In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are

visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on.
Algorithm BFS.

Program:

```
typedef struct queue *queue_pointer;
typedef struct queue {
 int vertex;
queue_pointerlink;
};
void addq(queue_pointer *,
queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
node_pointerw;
queue_pointer front, rear;
 front = rear = NULL;
printf("%d", v);
 visited[v] = TRUE;
addq(&front, &rear, v);
while (front) {
 v= deleteq(&front);
 for (w=graph[v]; w; w=w->link)
 if (!visited[w->vertex]) {
printf("%d", w->vertex);
addq(&front, &rear, w->vertex);
 visited[w->vertex] = TRUE;
 }
 }
}
```

Steps:

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
Step 5: Repeat step 3 and 4 until queue becomes empty.
Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Analysis of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, O(n2). If adjacency lists are used the loop has a total cost of d0 + … + dn-1 = O(e), where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

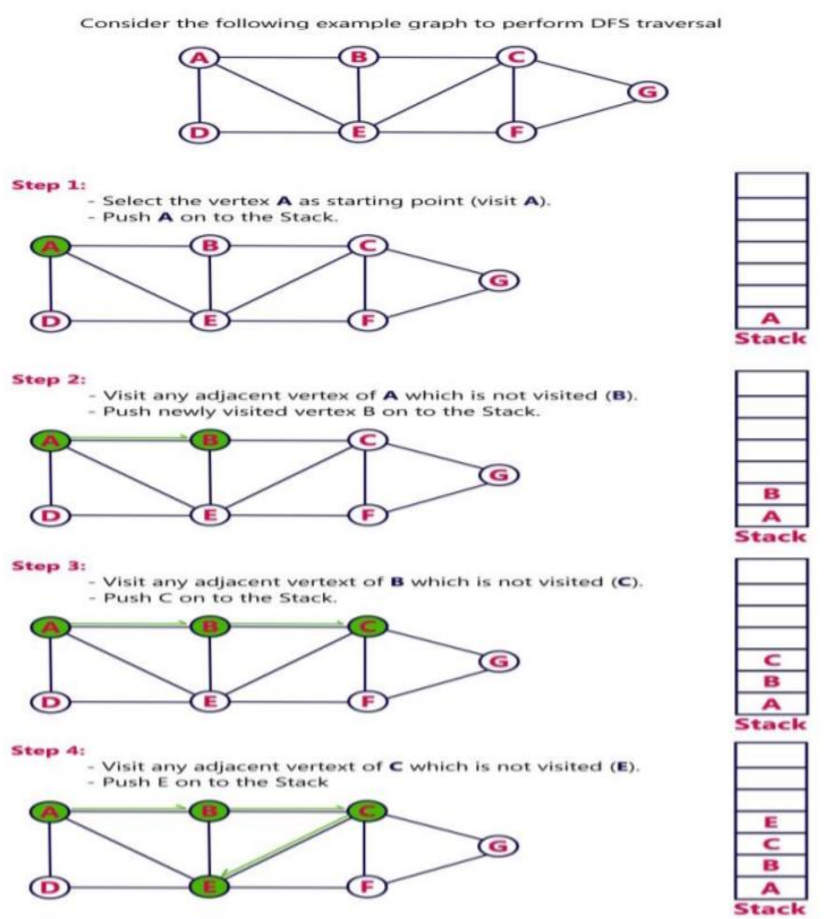
12.2.3.3 Connected Components

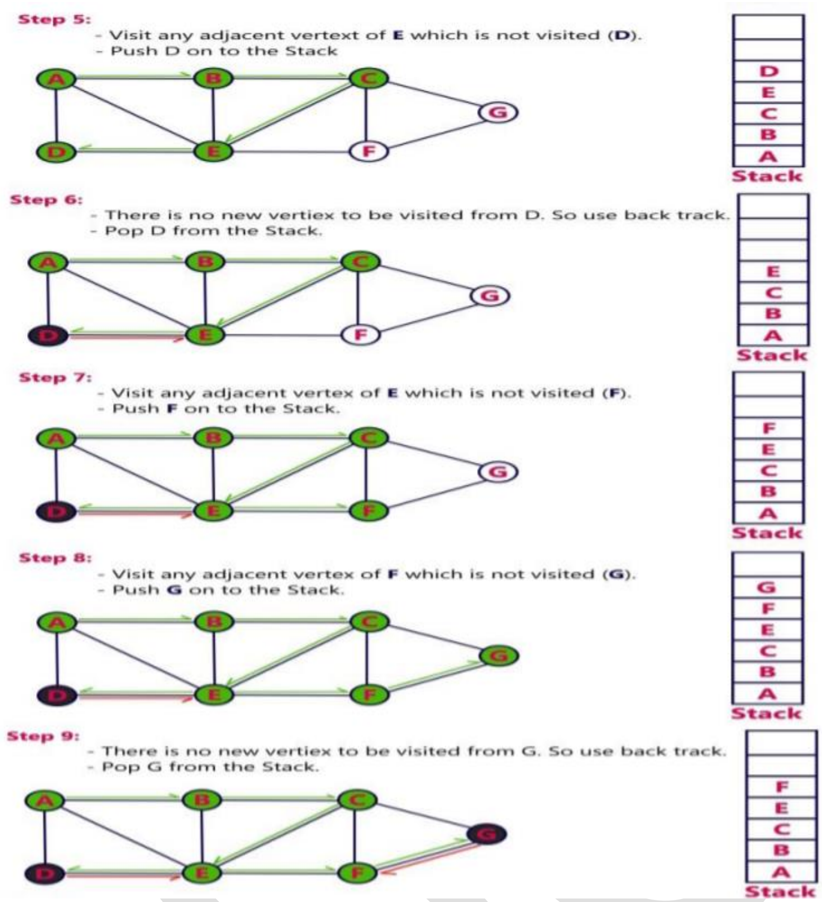
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected as given below in program which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
 for (i=0; i<n; i++) {
 if (!visited[i]) {
dfs(i);
printf("\n"); } } }
```

Analysis of Components:

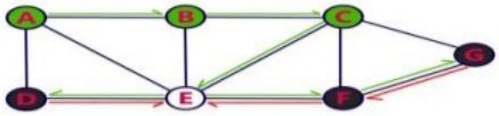
If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used, then the time required is O(n2).

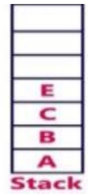



Step 5:
- Visit any adjacent vertext of E which is not visited (D).
- Push D on to the Stack
A
B
C
G
D
E
F
D
E
C
B
A
Stack
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.
A
B
C
G
D
E
F
E
C
B
A
Stack
Step 7:
- Visit any adjacent vertex of E which is not visited (F).
- Push F on to the Stack.
A
B
C
G
D
E
F
F
E
C
B
A
Stack
Step 8:
- Visit any adjacent vertex of F which is not visited (G).
- Push G on to the Stack.
A
B
C
G
D
E
F
G
F
E
C
B
A
Stack
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.
A
B
C
G
D
E
F
F
E
C
B
A
Stack

Step 10:

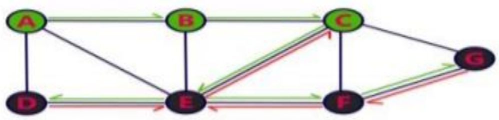
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

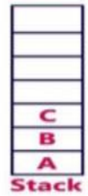




Step 11:

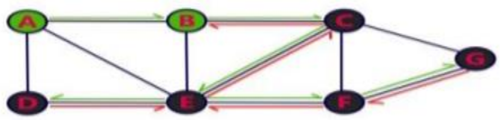
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





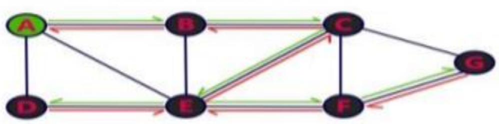
Step 12:

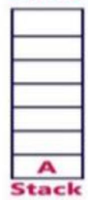
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:

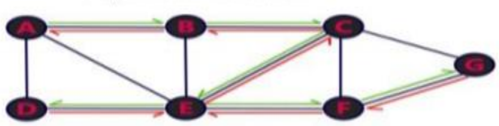
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

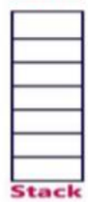




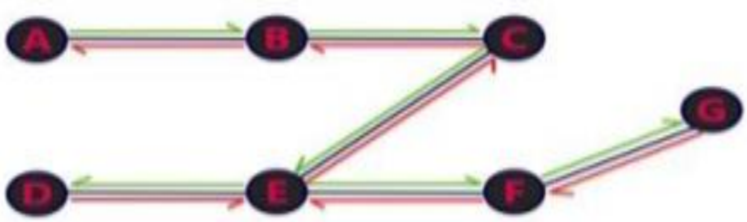
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

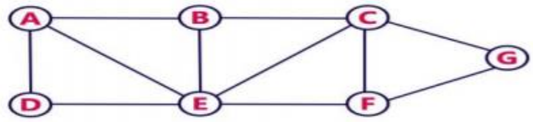




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

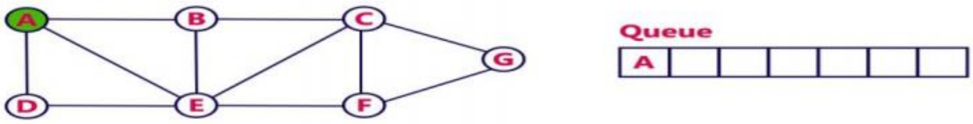


Consider the following example for BFS traversal.



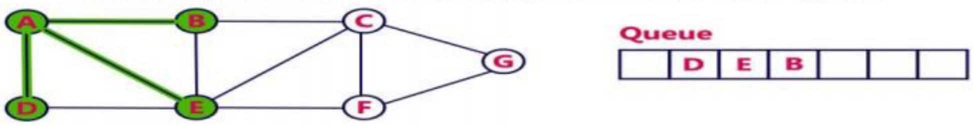
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



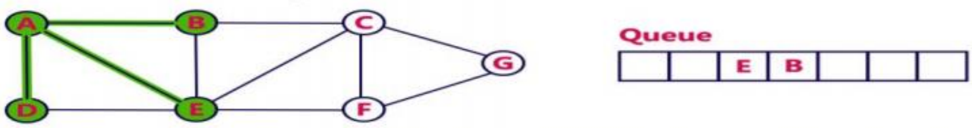
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.