ITT201 Data Structures

Module 4 : Trees and Graphs

Anoop S K M
(https://sites.google.com/site/skmanoop)
Department of Information Technology
Govt. Engg. College, Sreekrishnapuram, Palakkad

Acknowledgements

- All the pictures are taken from the Internet using Google search.
- Wikipedia also referred.

Lecture 28

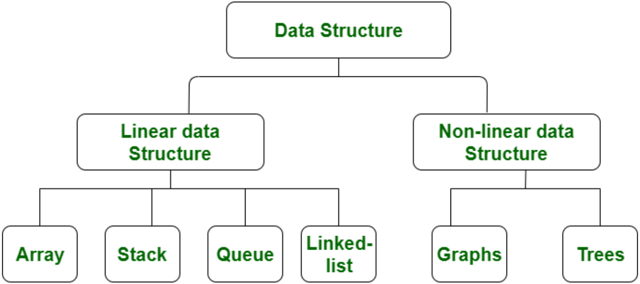
Recap & Goals

Till Now We Saw...

- Module 1 : Introduction to Data Structures
    - Searching - Linear and Binary Searches
    - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
    - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
    - Singly, Doubly, Circular Linked Lists
    - Dynamic Memory Management
- Module 3 : Stacks and Queues
    - Stacks and its applications
    - Queues and Types of queues

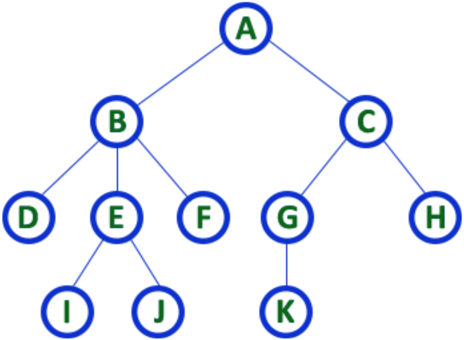
Today We Will See...

- Module 4 : Trees and Graphs
    - Trees : Basic Terminologies



- Array ✓
- Stack ✓
- Queue ✓
- Linked List ✓

Tree

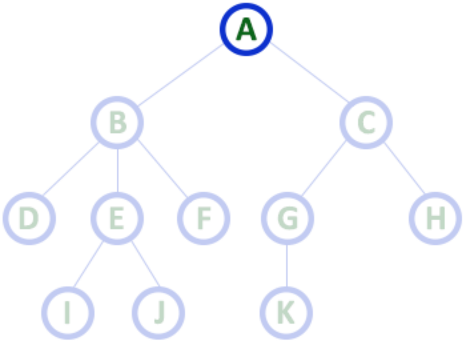


TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

img src: http://www.btechsmartclass.com

Root Node

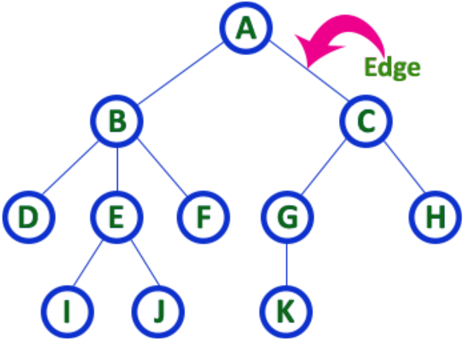


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

img src: http://www.btechsmartclass.com

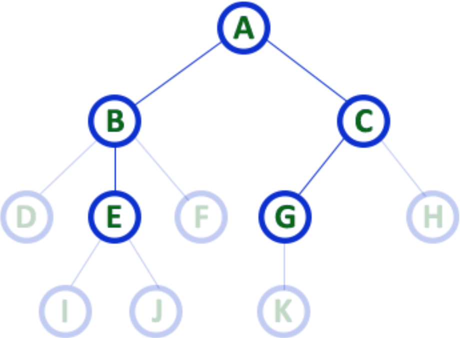
Edges



- In any tree, 'Edge' is a connecting link between two nodes.

img src: http://www.btechsmartclass.com

Parent Nodes

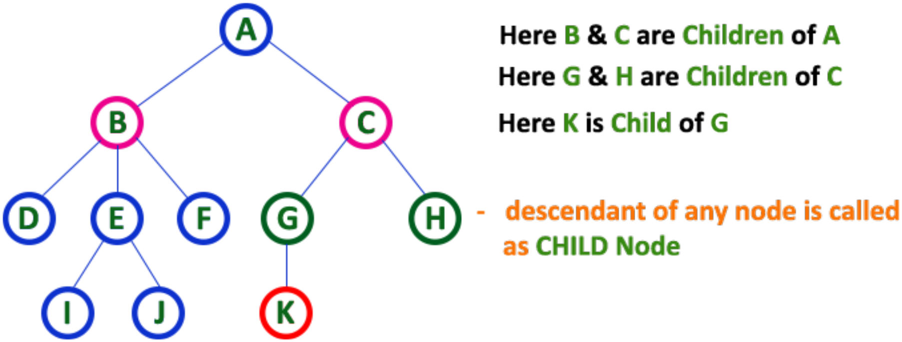


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

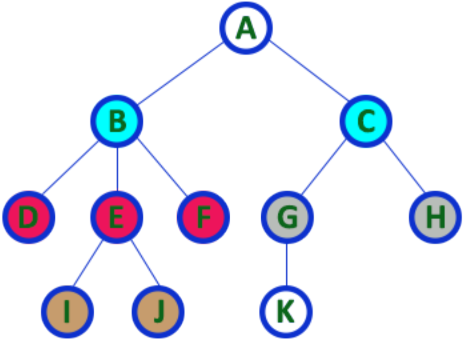
img src: http://www.btechsmartclass.com

Child Node



img src: http://www.btechsmartclass.com

Siblings

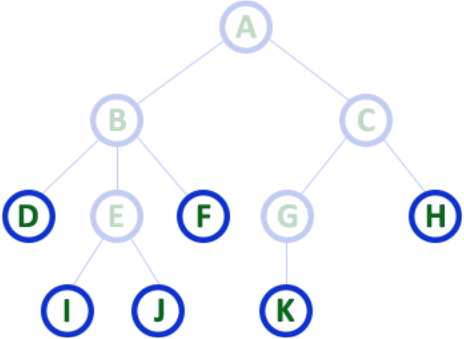


Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

img src: http://www.btechsmartclass.com

Leaf Node

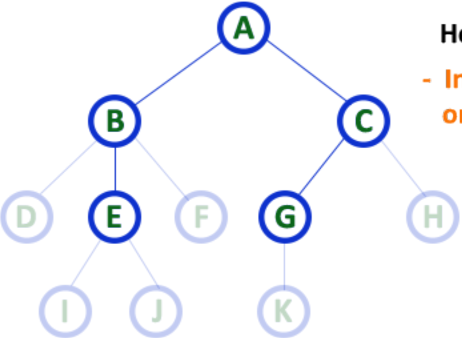


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

img src: http://www.btechsmartclass.com

Internal Node

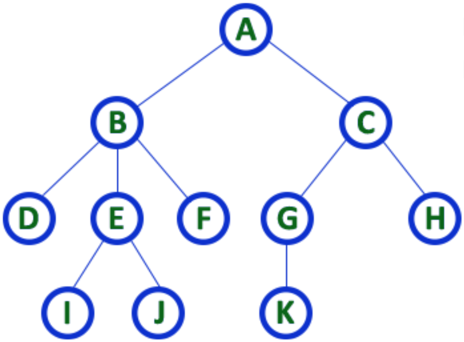


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

img src: http://www.btechsmartclass.com

Degree of a Node



Here Degree of B is 3

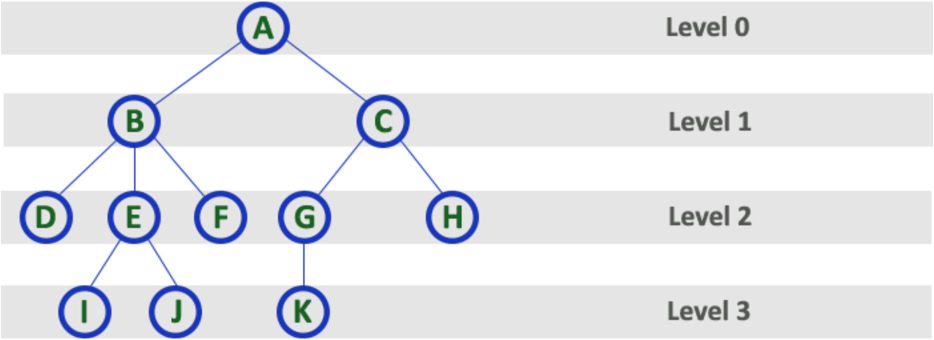
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

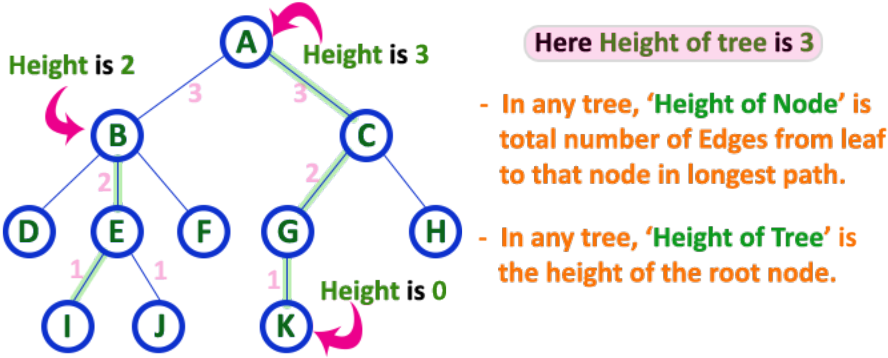
img src: http://www.btechsmartclass.com

Levels in a Rooted Tree



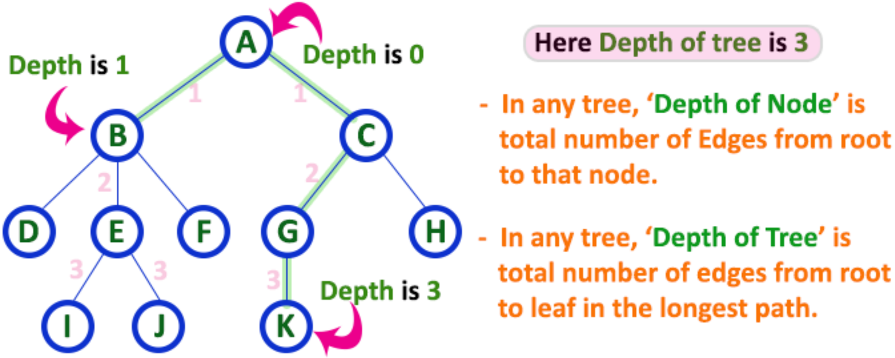
img src: http://www.btechsmartclass.com

Height



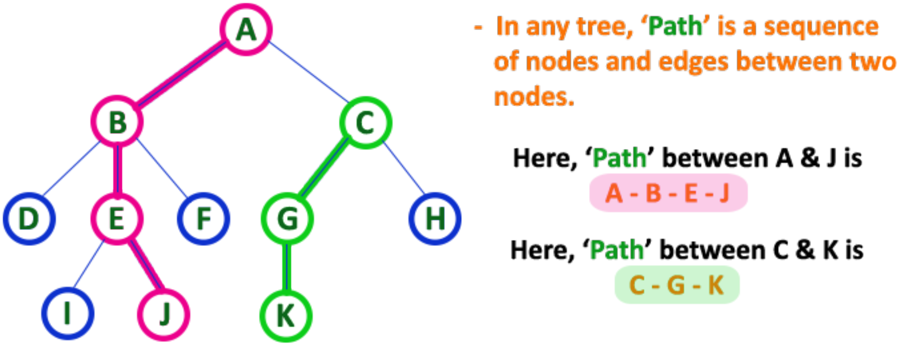
img src: http://www.btechsmartclass.com

Depth



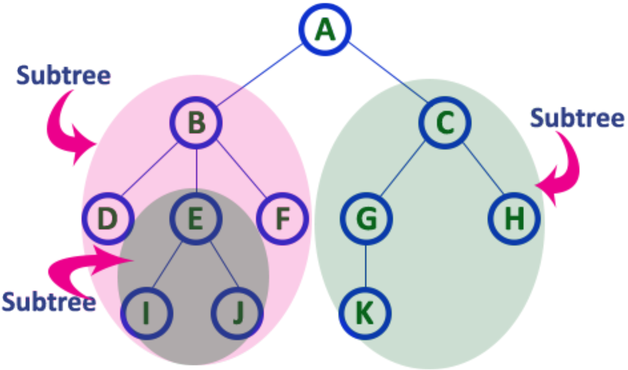
img src: http://www.btechsmartclass.com

Path



img src: http://www.btechsmartclass.com

Subtree



img src: http://www.btechsmartclass.com

Lecture 29

Recap & Goals

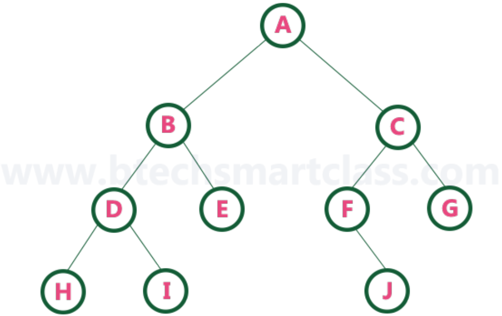
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees : Basic Terminologies

Today We Will See...

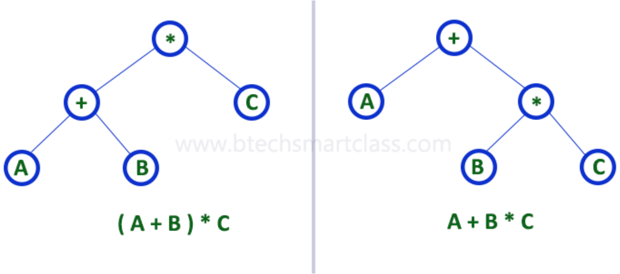
- Trees : Binary Trees

Binary Tree

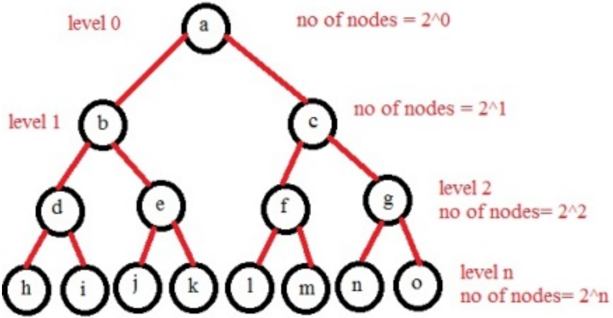


- Tree in which every node has at most 2 children

Binary Tree

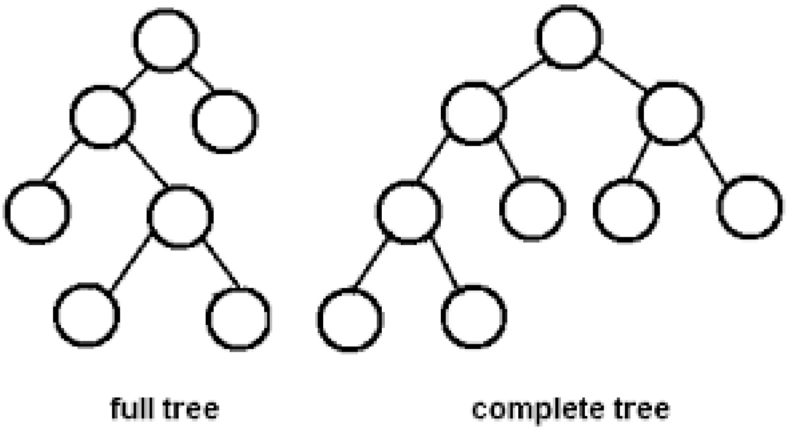


Binary Tree - Nodes at each level



- Number of nodes at level $i = 2^i$

Full Vs Complete Binary Tree



Full Binary Trees

A Full Binary Tree is a binary tree in which every node has either 0 or 2 children.

Full Binary Trees

A Full Binary Tree is a binary tree in which every node has either 0 or 2 children.

Recursive Definition : A full binary tree is either:

- A single vertex

Full Binary Trees

A Full Binary Tree is a binary tree in which every node has either 0 or 2 children.

Recursive Definition : A full binary tree is either:

- A single vertex
- A tree whose root node has two subtrees, both of which are full binary trees.

Full Binary Trees

A Full Binary Tree is a binary tree in which every node has either 0 or 2 children.

Recursive Definition : A full binary tree is either:

- A single vertex
- A tree whose root node has two subtrees, both of which are full binary trees.

Properites of Full Binary Trees

- A Full binary tree with ℓ leaves always contains $\ell - 1$ internal nodes

Full Binary Trees

A Full Binary Tree is a binary tree in which every node has either 0 or 2 children.

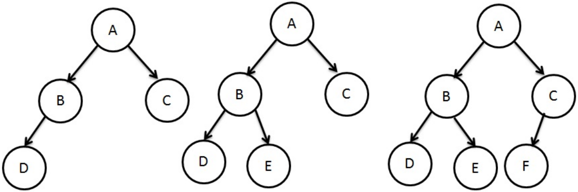
Recursive Definition : A full binary tree is either:

- A single vertex
- A tree whose root node has two subtrees, both of which are full binary trees.

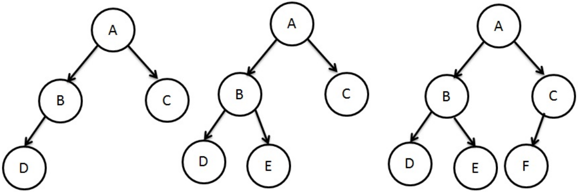
Properites of Full Binary Trees

- A Full binary tree with ℓ leaves always contains $\ell - 1$ internal nodes
- A Full binary tree with ℓ leaves always contains $2\ell - 1$ nodes

Complete Binary Trees

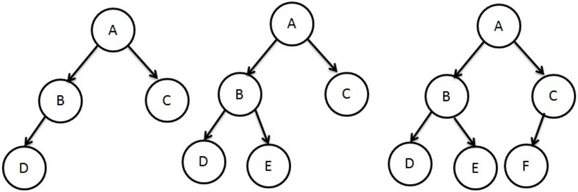


Complete Binary Trees



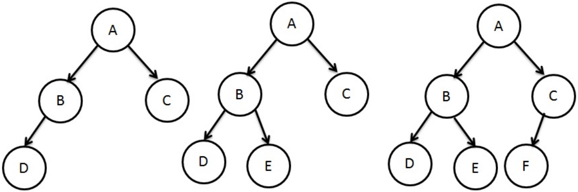
- every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Complete Binary Trees



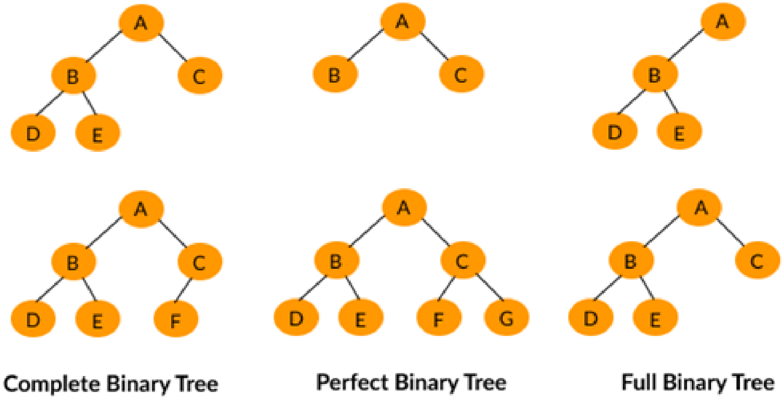
- every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- It can have between 1 and 2^{ℓ} nodes at the last level ℓ

Complete Binary Trees



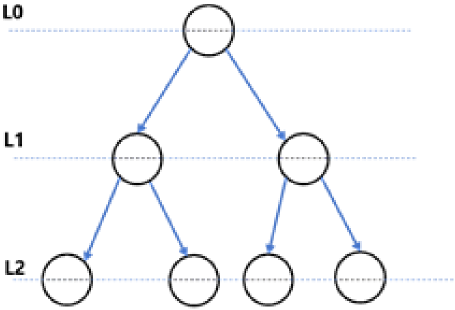
- every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- It can have between 1 and 2^ℓ nodes at the last level ℓ
- All other levels ℓ have 2^ℓ nodes.
- If there are 2^ℓ nodes at the last level, it is a Perfect Binary Tree

Complete,Perfect & Full Binary Trees



Maximum No. of Nodes

Max number of nodes at level



Ht = 0	$2^0 = 1$
Ht = 1	$2^1 = 2$
Ht = 2	$2^2 = 4$

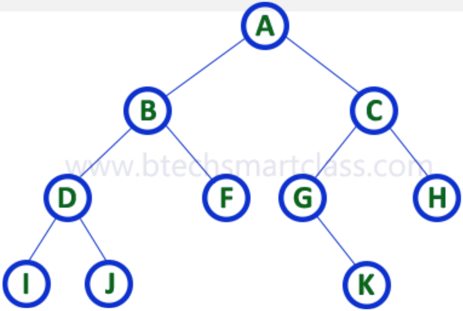
Max no. of nodes of binary tree of height "0" $= 2^0 = 1$

Max no. of nodes of binary tree of height "1" $= 2^0 + 2^1 = 1 + 2 = 3$

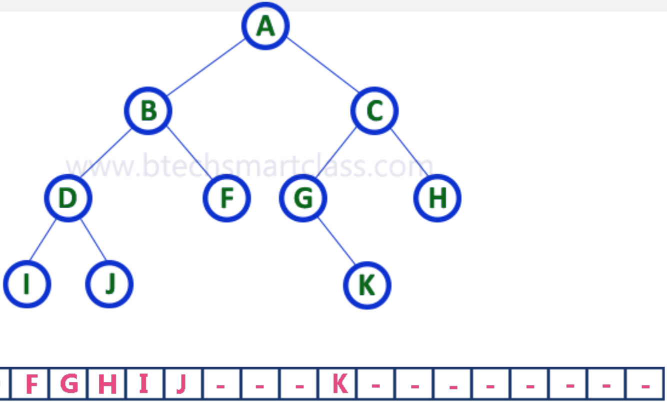
Max no. of nodes of binary tree of height "2" $= 2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7$

Max no. of nodes of binary tree of height "h" $= 2^0 + 2^1 + 2^2 + \ldots + 2^h$
$= 2^{h+1} - 1$

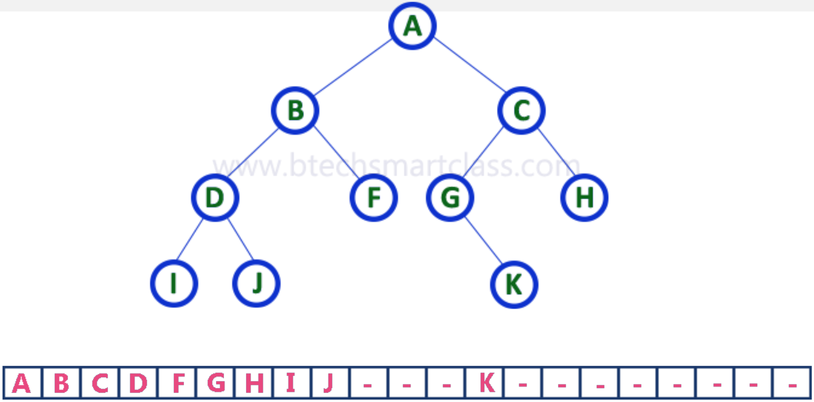
Array Representation of Binary Trees



Array Representation of Binary Trees



Array Representation of Binary Trees



- left and right child of a node at index i is at $2 * i + 1$ and $2 * i + 2$ respectively?
- parent of a node at index k is at $\lfloor \frac{k-1}{2} \rfloor$?

Lecture 30

Recap & Goals

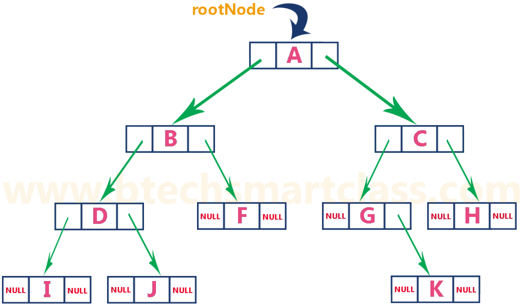
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees : Basic Terminologies

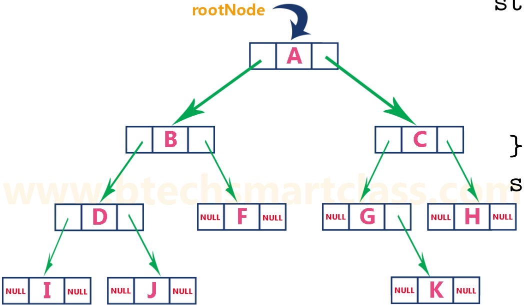
Today We Will See...

- Binary Trees : Linked List Representation, Tree Traversal

Linked List Representation of Binary Trees

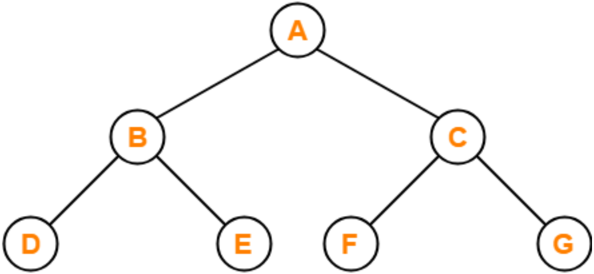


Linked List Representation of Binary Trees



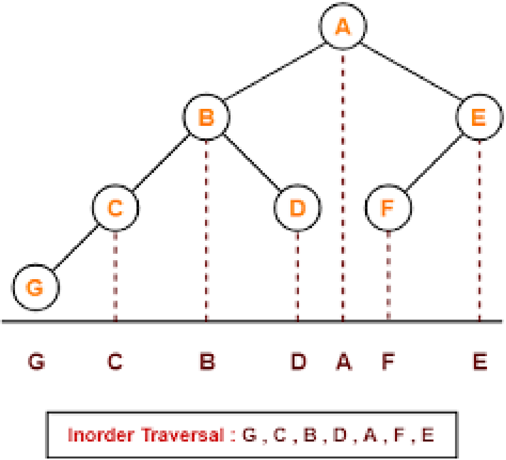

```
struct tnode{
    struct tnode *left;
    int data;
    struct tnode *right;
 };
 struct tnode *root = NULL;
```

Preorder: Node-Left-Right

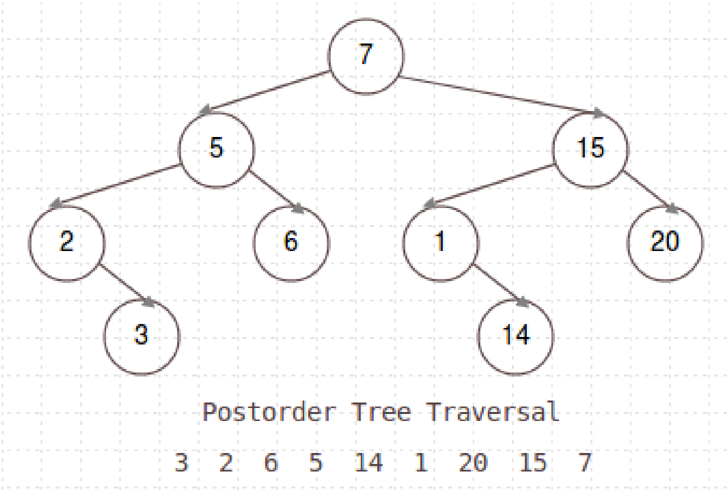


Preorder Traversal : A , B , D , E , C , F , G

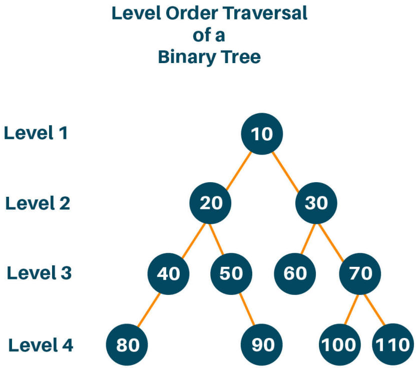
Inorder: Left-Node-Right



Postorder: Left-Right-Node

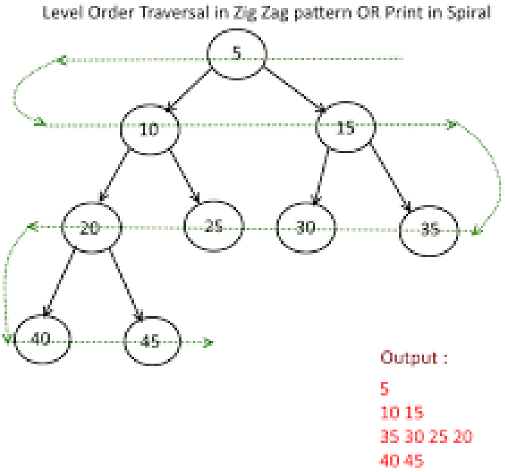


Levelorder

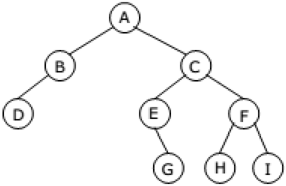


At level 1: [10]
At level 2: [20, 30]
At level 3: [40, 50, 60, 70]
At level 4: [80, 90, 100, 110]

Level Order-2



All traversals example



Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Lecture 31

Recap & Goals

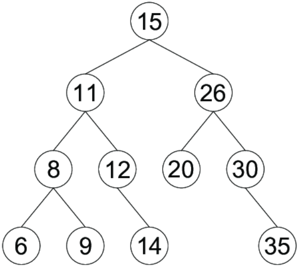
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees- Array and Linked List Representation
  - Tree Traversals -Preorder, Inorder, Postorder

Today We Will See...

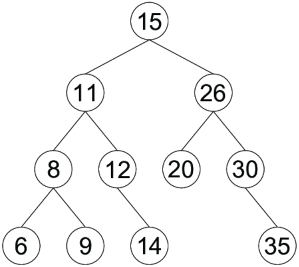
- Binary Search Trees

Binary Search Tree (BST)



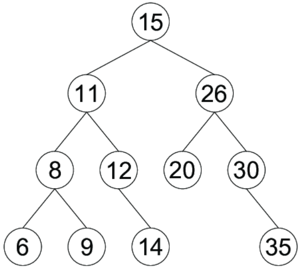
- left subtree of a node contains only nodes with data lesser than the node's data.

Binary Search Tree (BST)



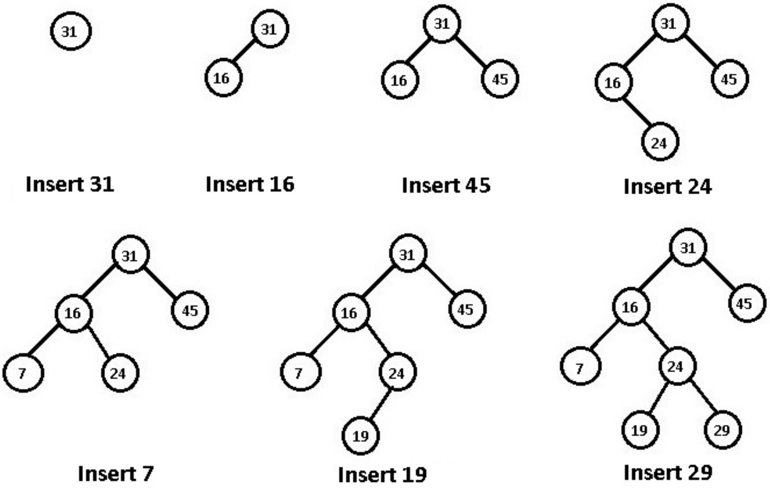
- left subtree of a node contains only nodes with data lesser than the node's data.
- right subtree of a node contains only nodes with data greater than or equal to the node's data.

Binary Search Tree (BST)

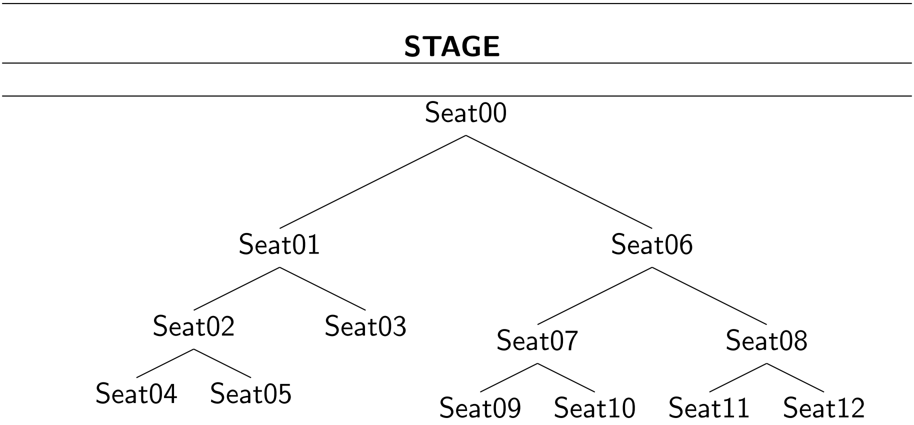


- left subtree of a node contains only nodes with data lesser than the node's data.
- right subtree of a node contains only nodes with data greater than or equal to the node's data.
- left and right subtree each must also be a binary search tree.

Insertion in BST - Example 1



Seat Arrangement in Auditorium



Rule of Seating

- Newcomer Always Goes to Seat00 and occupy if VACANT.

Rule of Seating

- Newcomer Always Goes to Seat00 and occupy if VACANT.
- If NOT VACANT, Go to the **Seat on Left** if in alphabetical order his name comes before the already seated person's.

Rule of Seating

- Newcomer Always Goes to Seat00 and occupy if VACANT.
- If NOT VACANT, Go to the **Seat on Left** if in alphabetical order his name comes before the already seated person's.
  Otherwise Go to the **Seat on Right**.

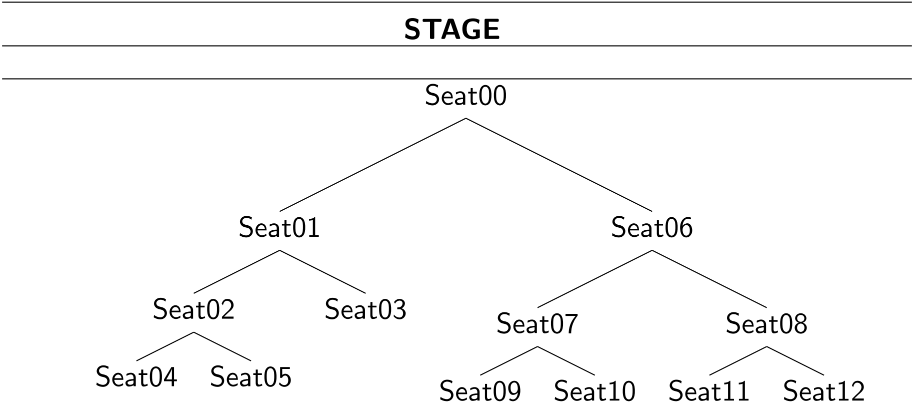
Rule of Seating

- Newcomer Always Goes to Seat00 and occupy if VACANT.
- If NOT VACANT, Go to the **Seat on Left** if in alphabetical order his name comes before the already seated person's.
  Otherwise Go to the **Seat on Right**.
- Apply the rule again till you find a vacant seat.

Place our Stars

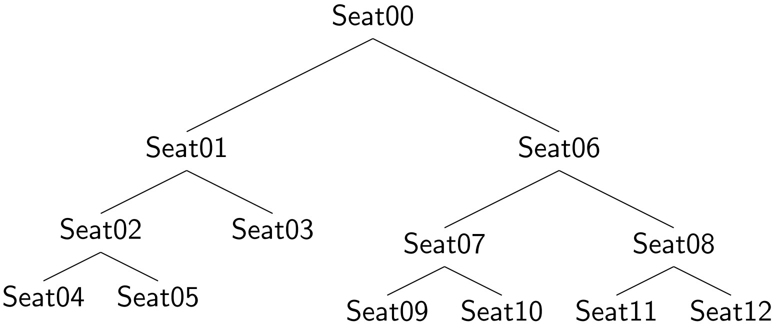
Let's seat our film stars one by one!

Seat Filling



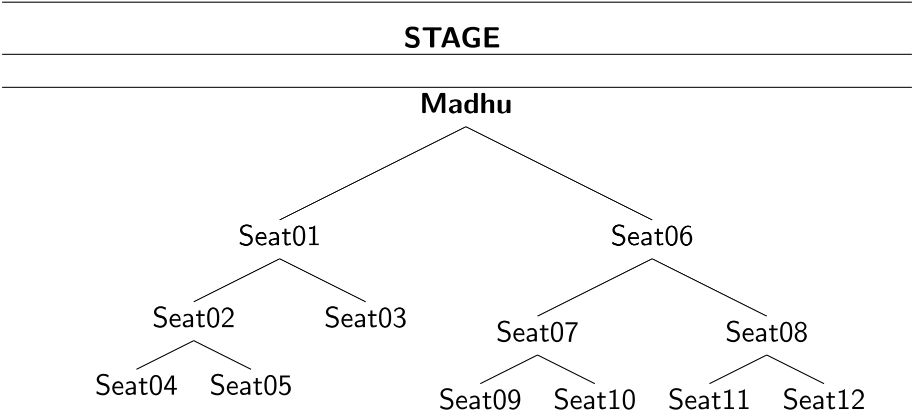
Seat Filling

STAGE



First Person : 

Seat Filling



Seat Filling

STAGE

Madhu

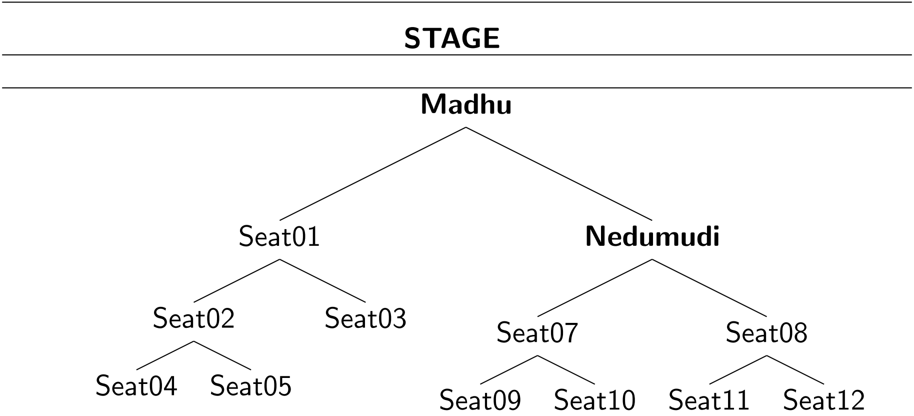
Seat01 Seat06

Seat02 Seat03 Seat07 Seat08

Seat04 Seat05 Seat09 Seat10 Seat11 Seat12

Second Person : 

Seat Filling



Seat Filling

STAGE

Madhu

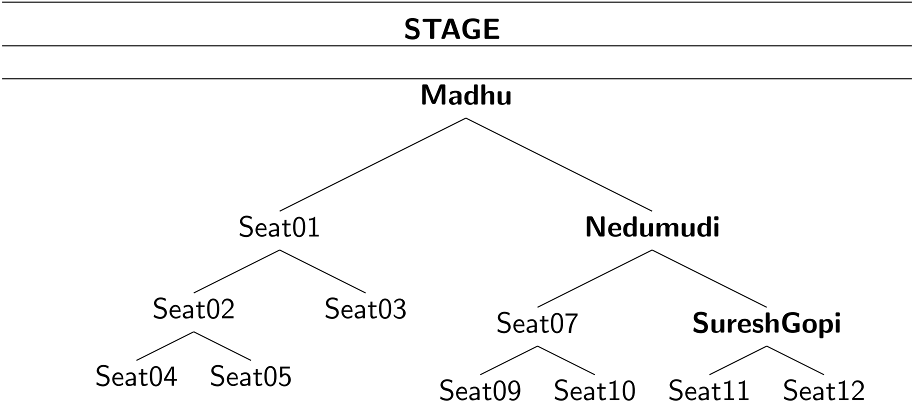
Seat01 **Nedumudi**

Seat02 Seat03 Seat07 Seat08

Seat04 Seat05 Seat09 Seat10 Seat11 Seat12

Third Person : 

Seat Filling



Seat Filling

STAGE

Madhu

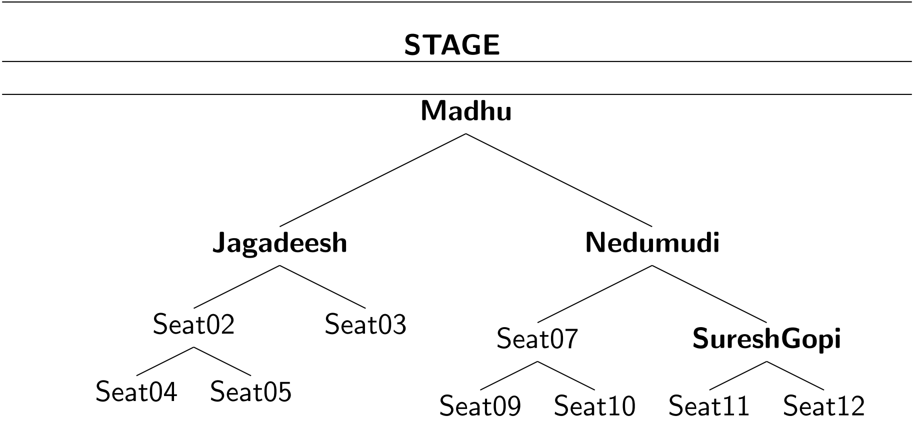
Seat01 **Nedumudi**

Seat02 Seat03 Seat07 **SureshGopi**

Seat04 Seat05 Seat09 Seat10 Seat11 Seat12

Fourth Person : 

Seat Filling



Seat Filling

STAGE

Madhu

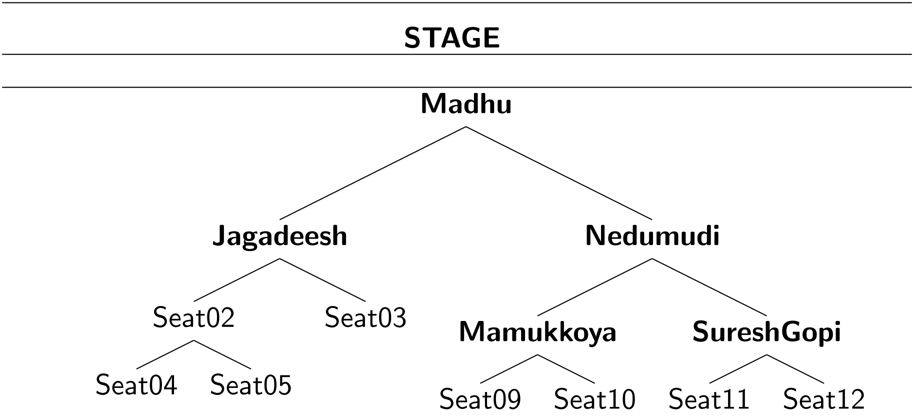
Jagadeesh **Nedumudi**

Seat02 Seat03 Seat07 **SureshGopi**

Seat04 Seat05 Seat09 Seat10 Seat11 Seat12

Fifth Person : 

Seat Filling



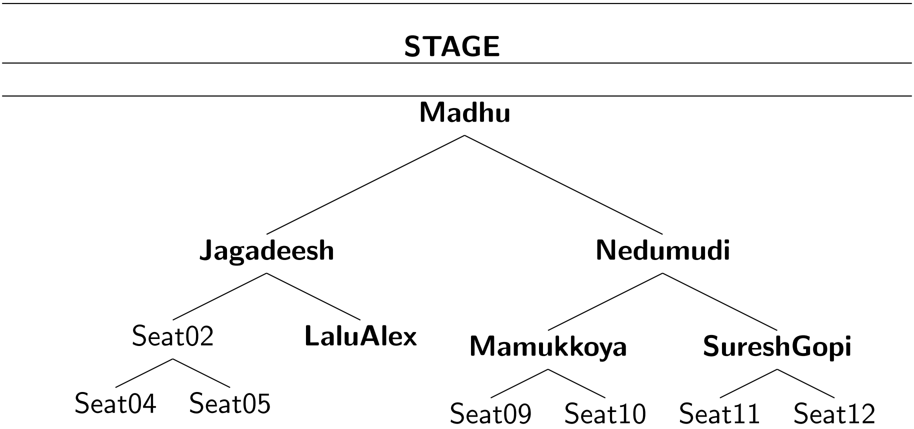
Seat Filling

STAGE

- **Madhu**
  - **Jagadeesh**
    - Seat02
      - Seat04
      - Seat05
    - Seat03
  - **Nedumudi**
    - **Mamukkoya**
      - Seat09
      - Seat10
    - **SureshGopi**
      - Seat11
      - Seat12

Last Person : 

Seat Filling



Let's Welcome One by One on to the Stage!

Rule for going to the stage

Anyone getting the invitation should

Rule for going to the stage

Anyone getting the invitation should

- Pass the invitation to the person on your left (if any)

Rule for going to the stage

Anyone getting the invitation should

- Pass the invitation to the person on your left (if any)
- Go to stage only after everyone on your left side has gone

Rule for going to the stage

Anyone getting the invitation should

- Pass the invitation to the person on your left (if any)
- Go to stage only after everyone on your left side has gone
- Invite the person on your right (if any)

Rule for going to the stage

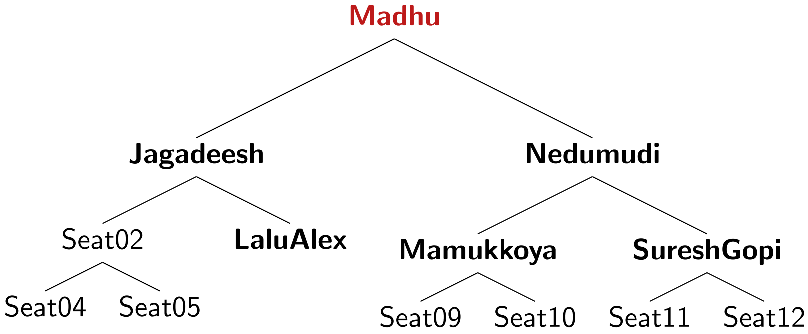
Anyone getting the invitation should

- Pass the invitation to the person on your left (if any)
- Go to stage only after everyone on your left side has gone
- Invite the person on your right (if any)

Let's Welcome **Sri. Madhu !**

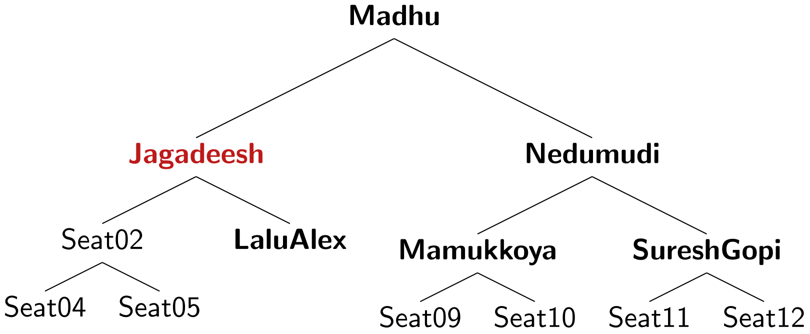
On to the Stage

STAGE



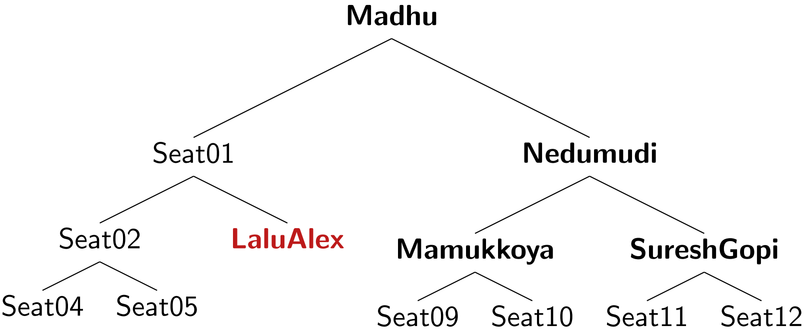
On to the Stage

STAGE



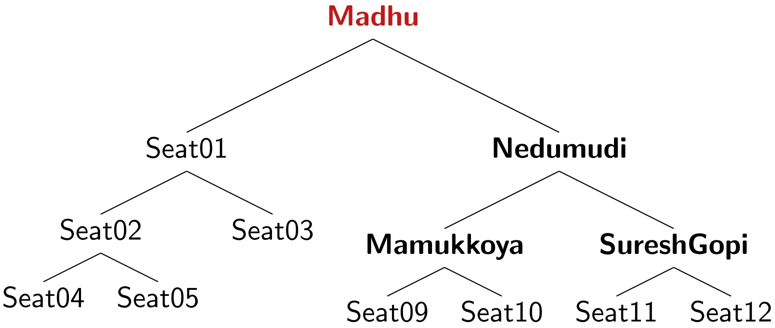
On to the Stage

STAGE



On to the Stage

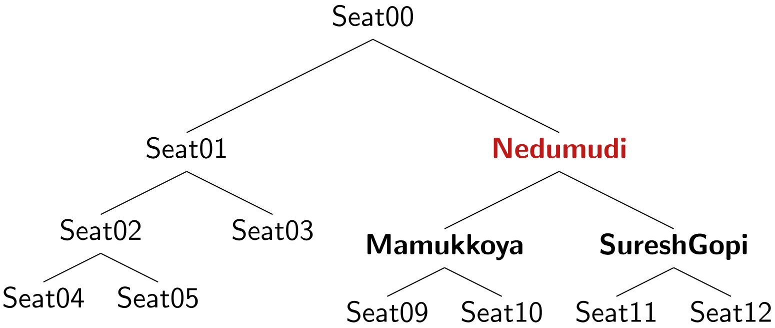
STAGE



On to the Stage

STAGE

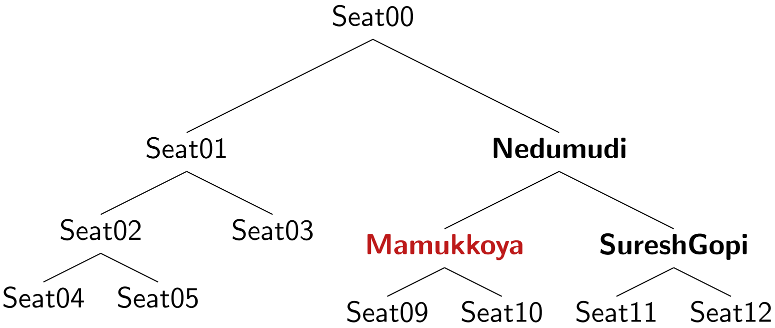




On to the Stage

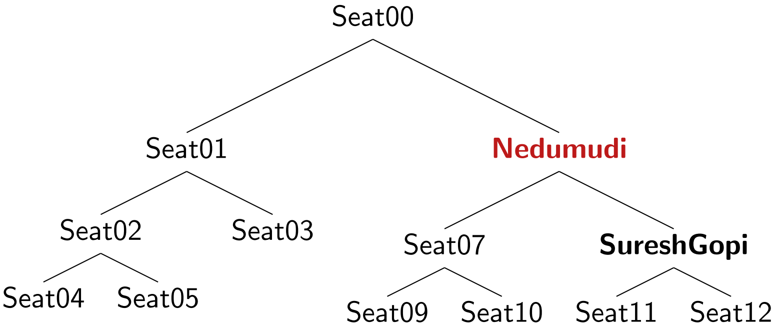
STAGE





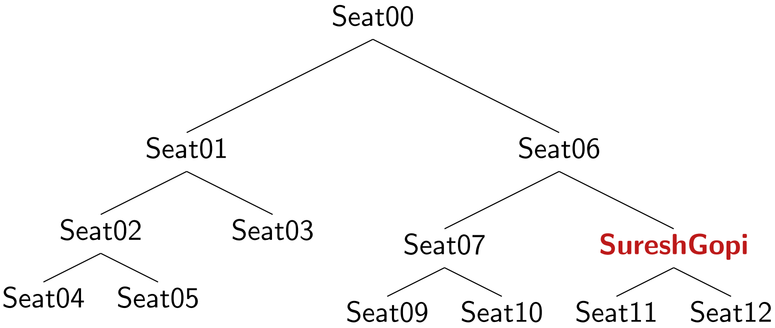
On to the Stage

STAGE



On to the Stage

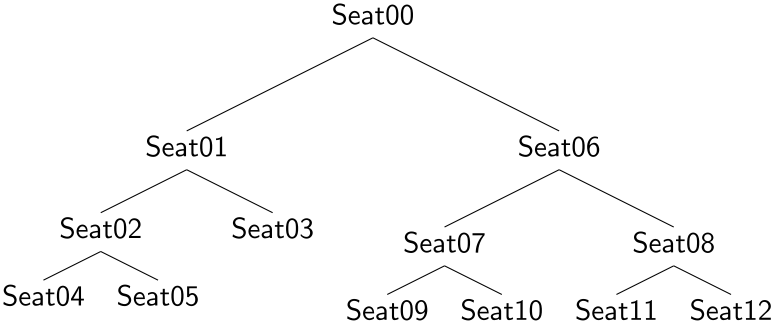
STAGE



On to the Stage

STAGE

Jagadeesh, LaluAlex, Madhu, Mamukkoya, Nedumudi, SureshGopi



They are standing in ALPHABETICAL ORDER!!!

Lecture 2^5

Recap & Goals

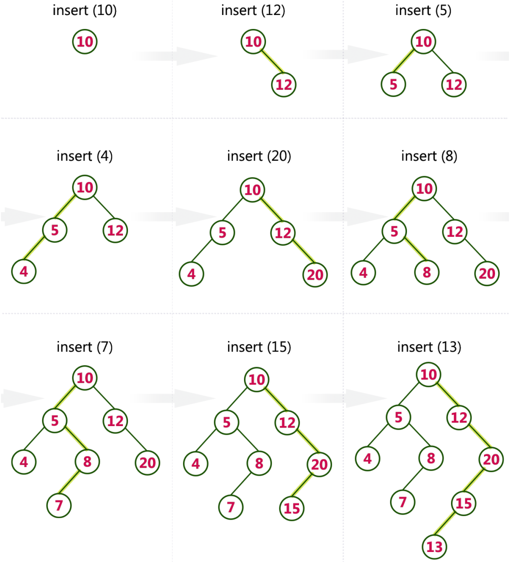
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees- Array and Linked List Representation
  - Tree Traversals -Preorder, Inorder, Postorder
  - Binary Search Trees

Today We Will See...

- Implementation : Binary Search Tree Insertion, Tree Traversals

Insertion in BST- Example 2



Node Structure in Binary Search Trees

Node Structure in Binary Search Trees


```
struct bstnode{
   struct bstnode *left;
   int data;
   struct bsttnode *right;
 };
 struct bstnode *root = NULL
```

BST : Insertion Algorithm

- Create a node by allocating space for it

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
    - if left of root is null, place the node on the left of root

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
  - if left of root is null, place the node on the left of root
  - if not null, do the same procedure starting from the node on the left of root

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
  - if left of root is null, place the node on the left of root
  - if not null, do the same procedure starting from the node on the left of root
- if the value of the newnode $>=$ value of the root,

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
  - if left of root is null, place the node on the left of root
  - if not null, do the same procedure starting from the node on the left of root
- if the value of the newnode $>=$ value of the root,
  - if right of root is null, place the node on the right of root

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
  - if left of root is null, place the node on the left of root
  - if not null, do the same procedure starting from the node on the left of root
- if the value of the newnode $>=$ value of the root,
  - if right of root is null, place the node on the right of root
  - if not null, do the same procedure starting from the node on the right of root

BST : Insertion Algorithm

- Create a node by allocating space for it
- Assign Data to the node
- Make the left and right of node point to null
- Now, using the property of BST insert this node by starting comparison with root.
- if the value of the newnode $<$ value of the root,
  - if left of root is null, place the node on the left of root
  - if not null, do the same procedure starting from the node on the left of root
- if the value of the newnode $>=$ value of the root,
  - if right of root is null, place the node on the right of root
  - if not null, do the same procedure starting from the node on the right of root
- continue the above procedure till we find a null

BST : Insertion Algorithm

```
temp = malloc(sizeof(struct bstnode));
temp->data = x;
temp->left = temp->right = NULL;
insertToBST(root, temp);
------------------------------------------
insertToBST(struct bstnode *atnode, struct bstnode *temp)
{
  if(root == NULL) then root = temp and return;
  if( temp->data < atnode->data){
    if(atnode->left == NULL){
      atnode->left = temp;
    }else{
      insertToBST(atnode->left, temp);
    }
  }else{
    /*same code replacing left with right*/
  }
```

Preorder Traversal (Node, Left, Right)

```
void preorder(struct bstnode *atnode)
{
 if(atnode!=NULL){
   printf("%d ",atnode->data);
   preorder(atnode->left);
   preorder(atnode->right);
 }
}
```

Inorder Traversal(Left, Node, Right)

```
void inorder(struct bstnode *atnode)
{
 if(atnode!=NULL){
   inorder(atnode->left);
   printf("%d ",atnode->data);
   inorder(atnode->right);
 }
}
```

Postorder Traversal(Left, Right, Node)

```
void postorder(struct bstnode *atnode)
{
 if(atnode!=NULL){
   postorder(atnode->left);
   postorder(atnode->right);
   printf("%d ",atnode->data);

 }
}
```

Lecture 33

Recap & Goals

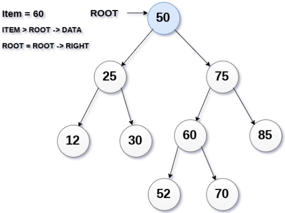
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees- Array and Linked List Representation
  - Tree Traversals -Preorder, Inorder, Postorder
  - Binary Search Tree : Insertion

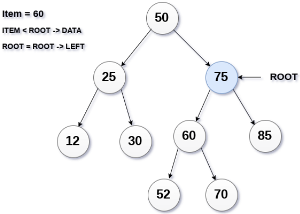
Today We Will See...

- Binary Search Tree : Search, Deletion

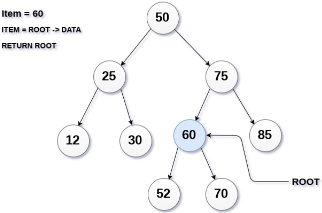
Search in BST



STEP 1

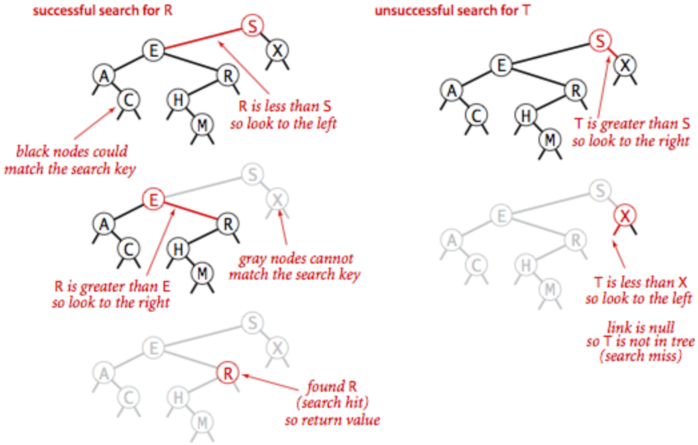


STEP 2



STEP 3

Search in Character Data BST



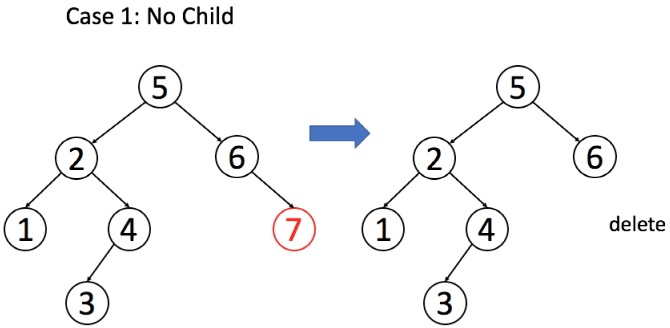
Successful (left) and unsuccessful (right) search in a BST

BST deletion

Three different cases

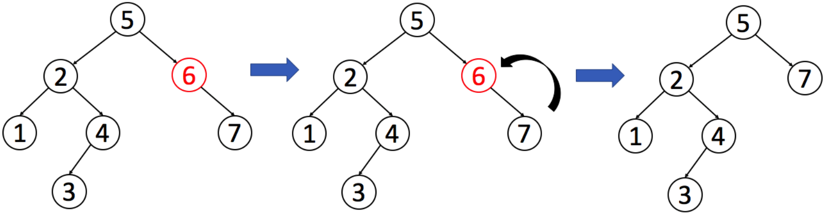
- Case 1: Node to be deleted is the leaf
- Case 2: Node to be deleted has only one child
- Case 3: Node to be deleted has two children

Case 1: Node to be deleted is the leaf

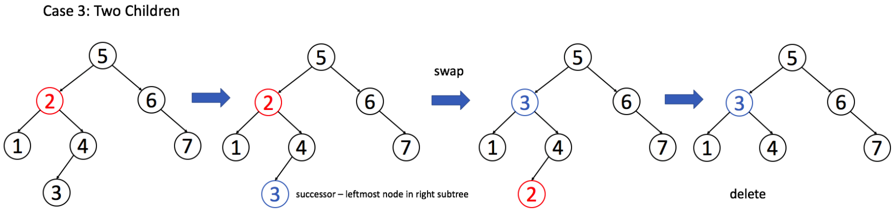


Case 2: Node to be deleted has only one child

Case 2: One Child



Case 3: Node to be deleted has two children



- Find the Inorder successor
- Swap with it
- Apply deletion algorithm again

Lecture 34

Recap & Goals

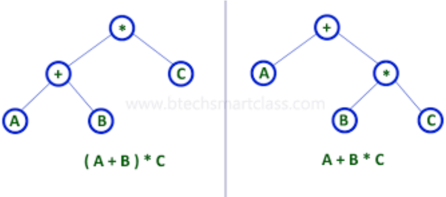
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees- Array and Linked List Representation
  - Tree Traversals -Preorder, Inorder, Postorder
  - Binary Search Tree : Insertion, Search, Deletion

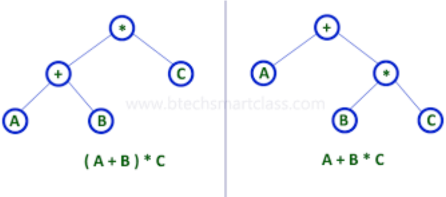
Today We Will See...

- Expression Tree, Heap Tree

Expression Tree : Properties

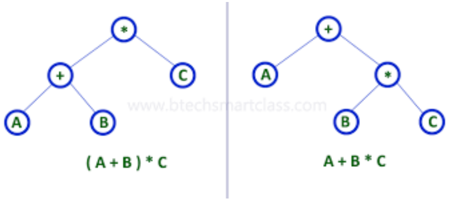


Expression Tree : Properties



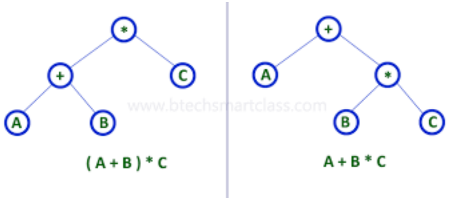
- Expression Trees are Full Binary Trees.

Expression Tree : Properties



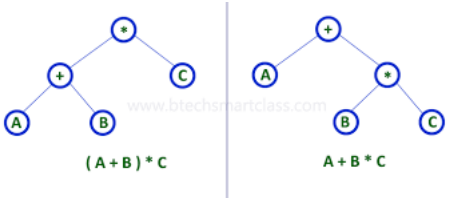
- Expression Trees are Full Binary Trees.
- For a binary operation of the form *operand*1 *operator* *operand*2, the operator will be the parent node, with *operand*1 its left child and *operand*2 its right child.

Expression Tree : Properties



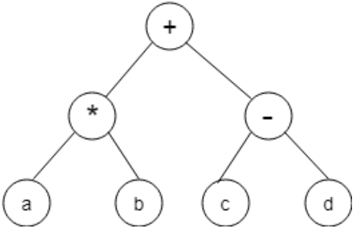
- Expression Trees are Full Binary Trees.
- For a binary operation of the form *operand*1 *operator* *operand*2, the operator will be the parent node, with *operand*1 its left child and *operand*2 its right child.
- operands are leaves, the number of leaves is equal to the number of operands, the number of internal nodes is equal to the number of operators.

Expression Tree : Properties

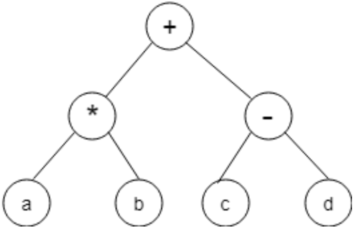


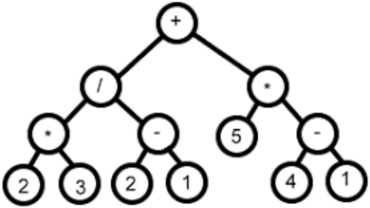
- Expression Trees are Full Binary Trees.
- For a binary operation of the form *operand*1 *operator* *operand*2, the operator will be the parent node, with *operand*1 its left child and *operand*2 its right child.
- operands are leaves, the number of leaves is equal to the number of operands, the number of internal nodes is equal to the number of operators.
- the preorder, inorder and postorder traversal of an expression tree gives the prefix, infix and postfix expression.

Expression Tree Examples



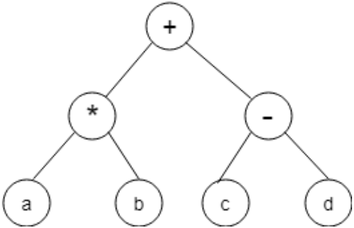
Expression Tree Examples

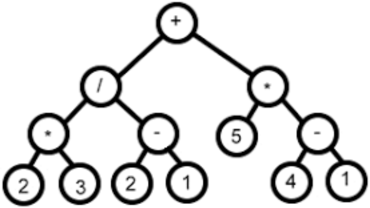




Expression tree for 2*3/(2-1)+5*(4-1)

Expression Tree Examples





Expression tree for 2*3/(2-1)+5*(4-1)

Expression Tree Creation from an Expression

- Identify the order in which operation is carried out in the given expression.

Expression Tree Creation from an Expression

- Identify the order in which operation is carried out in the given expression.
- for each operation of the form $op1$ $operator$ $op2$, with $op1$ and $op2$ as the left and right child make the parent node as $operator$

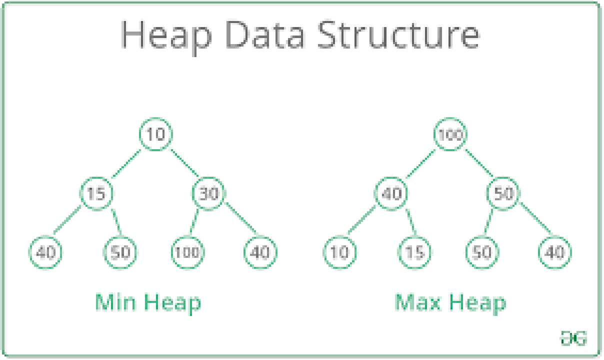
Expression Tree Creation from an Expression

- Identify the order in which operation is carried out in the given expression.
- for each operation of the form *op*1 *operator* *op*2, with *op*1 and *op*2 as the left and right child make the parent node as *operator*
- if an operation involves a sub expression, make the tree corresponding to it as the left/right subtree accordingly, with the operator as the root.

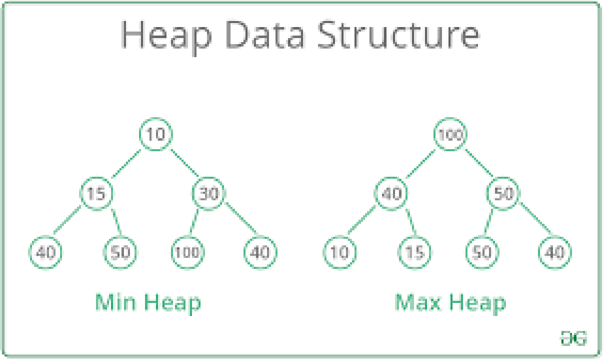
Expression Tree Creation from an Expression

- Identify the order in which operation is carried out in the given expression.
- for each operation of the form *op*1 *operator* *op*2, with *op*1 and *op*2 as the left and right child make the parent node as *operator*
- if an operation involves a sub expression, make the tree corresponding to it as the left/right subtree accordingly, with the operator as the root.
- the last operation that happens in an expression in the order of evaluation, will become the root of the final expression tree.

Heap Tree Types

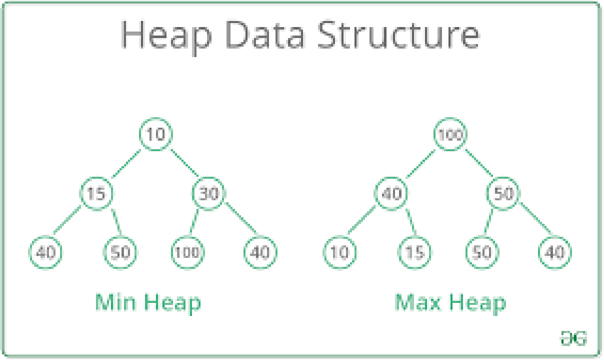


Heap Tree Types



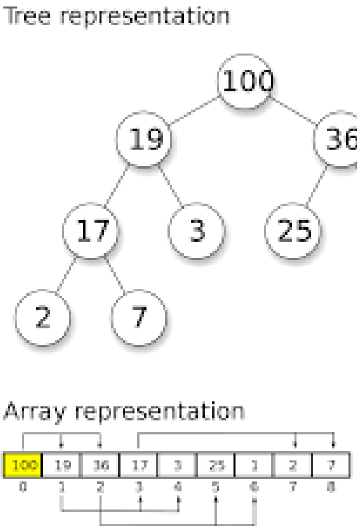
- Min Heap : all the nodes on left and right subtrees of a node contains values greater than(or equal to) the node.

Heap Tree Types



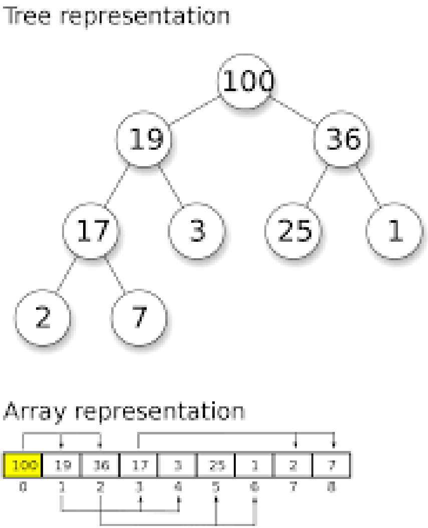
- Min Heap : all the nodes on left and right subtrees of a node contains values greater than(or equal to) the node.
- Max Heap : all the nodes on left and right subtrees of a node contains value lesser than (or equal to) the node.

Heap using Array



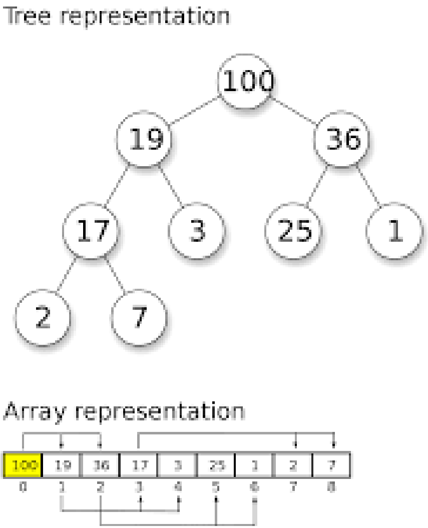
- Best data structure for Heap implementation ?

Heap using Array



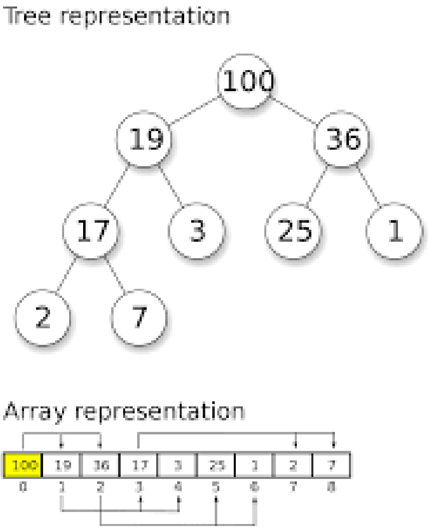
- Best data structure for Heap implementation ? Array!

Heap using Array

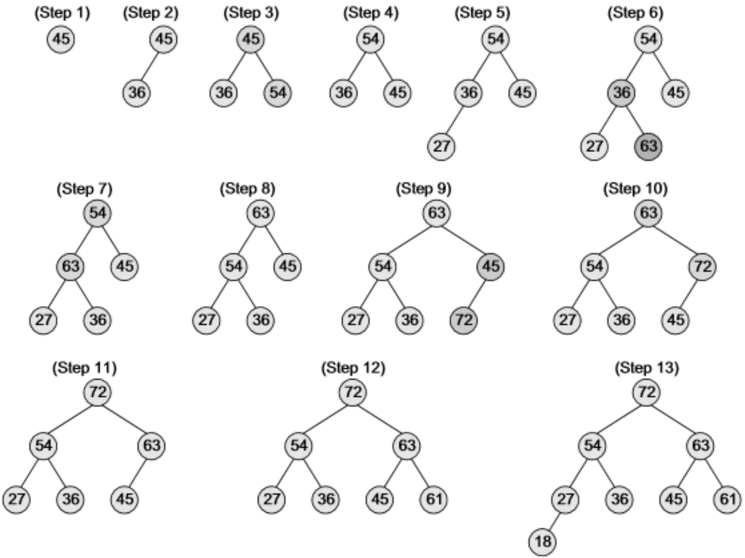


- Best data structure for Heap implementation ? Array! Why?

Heap using Array



- Best data structure for Heap implementation ? Array! Why?
- Heap is always a complete binary tree.


(Step 1)
45
(Step 2)
45
36
(Step 3)
45
36
54
(Step 4)
54
36
45
(Step 5)
54
36
45
27
(Step 6)
54
36
45
27
63
(Step 7)
54
63
45
27
36
(Step 8)
63
54
45
27
36
(Step 9)
63
54
45
27
36
72
(Step 10)
63
54
72
27
36
45
(Step 11)
72
54
63
27
36
45
(Step 12)
72
54
63
27
36
45
61
(Step 13)
72
54
63
27
36
45
61
18

Lecture 35

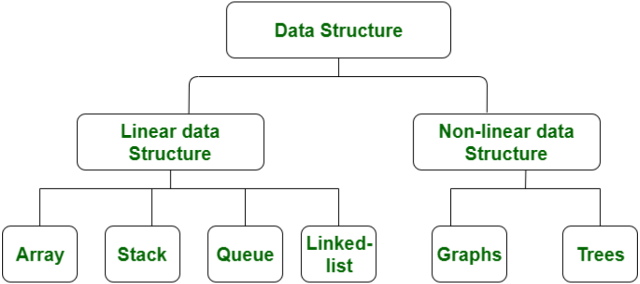
Recap & Goals

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees- Array and Linked List Representation
  - Tree Traversals -Preorder, Inorder, Postorder
  - Binary Search Tree, Expression Tree, Heap Tree

Today We Will See...

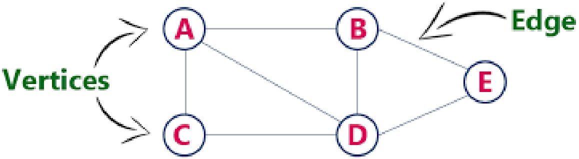
- Graphs & Graph Traversals (BFS & DFS)



- Array ✓
- Stack ✓
- Queue ✓
- Linked List ✓
- Trees ✓

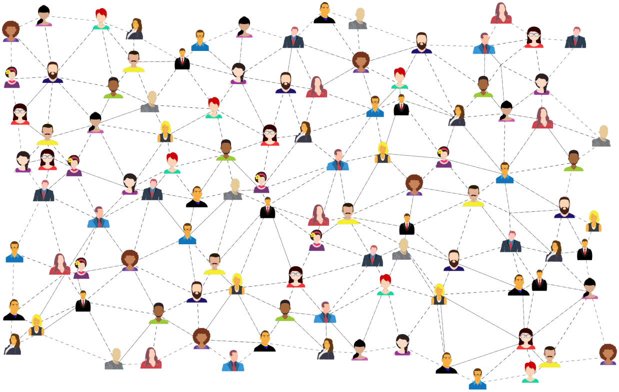
Graph

- Graph $G = (V, E)$
  - V, set of Vertices(commonly called Nodes)
  - E, set of Edges, i.e. set of pair of nodes(a, b), where $a, b \in V$

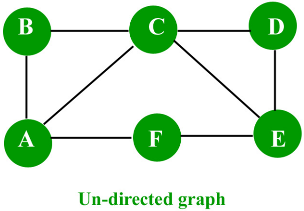


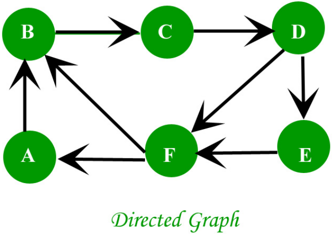
- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)\}$

Graph Example - Facebook Graph



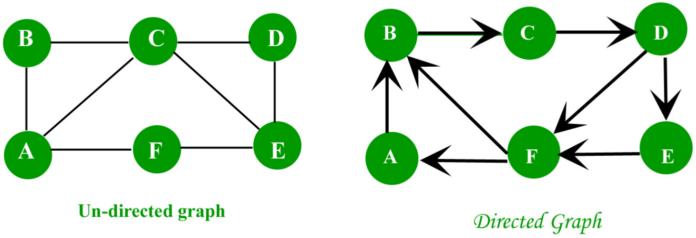
Undirected and Directed Graphs





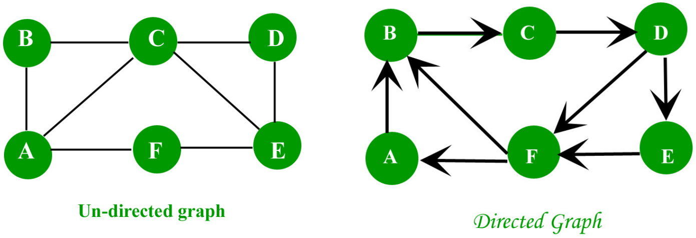
- Directed Graphs have directions for the edges.

Undirected and Directed Graphs



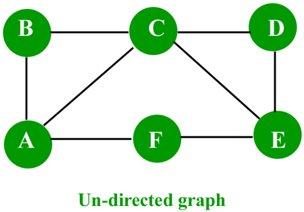
- Directed Graphs have directions for the edges.
- In Undirected, we may use either (a, b) or (b, a) to represent an edge between a and b

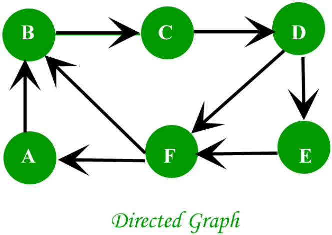
Undirected and Directed Graphs



- Directed Graphs have directions for the edges.
- In Undirected, we may use either (a, b) or (b, a) to represent an edge between a and b
- An edge (a, b) in Directed graph is different from the edge (b, a).

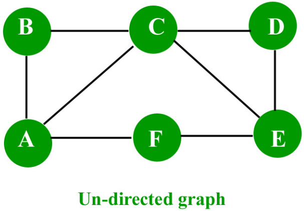
Undirected and Directed Graphs

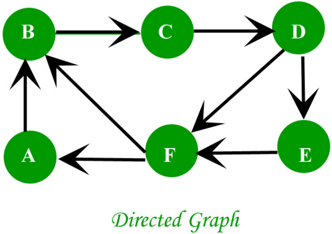




- Directed Graphs have directions for the edges.
- In Undirected, we may use either (a, b) or (b, a) to represent an edge between a and b
- An edge (a, b) in Directed graph is different from the edge (b, a).
- Undirected Graph Example : Facebook.

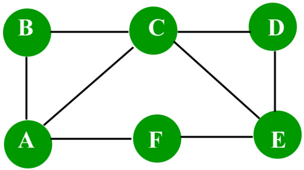
Undirected and Directed Graphs



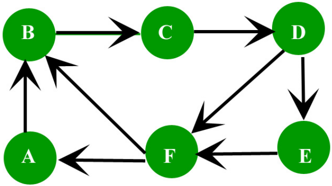


- Directed Graphs have directions for the edges.
- In Undirected, we may use either (a, b) or (b, a) to represent an edge between a and b
- An edge (a, b) in Directed graph is different from the edge (b, a).
- Undirected Graph Example : Facebook.
- Directed Graph Example : Twitter.

Degree of a node



Un-directed graph

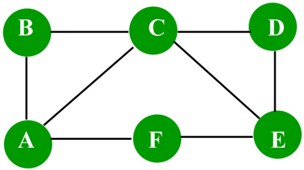


Directed Graph

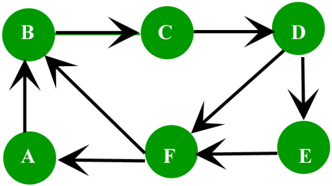
In Undirected Graphs,

- The number of edges connected to a node is its Degree

Degree of a node



Un-directed graph



Directed Graph

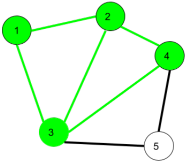
In Undirected Graphs,

- The number of edges connected to a node is its Degree

In Directed Graphs

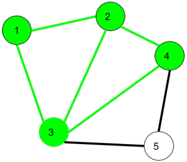
- Indegree : number of incoming edges to a node.
- Outdegree : number of outgoing edges to a node.

Walk in a Graph



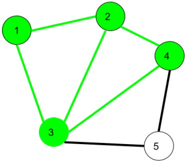
- Walk is a sequence of vertices and edges of a graph

Walk in a Graph



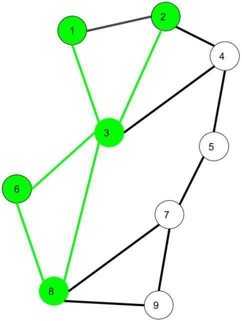
- Walk is a sequence of vertices and edges of a graph
- $1- > 2- > 3- > 4- > 2- > 1- > 3$ is a walk.

Walk in a Graph



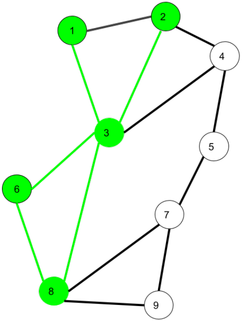
- Walk is a sequence of vertices and edges of a graph
- $1- > 2- > 3- > 4- > 2- > 1- > 3$ is a walk.
- if the starting and ending vertices are same, it is called a closed walk.

Trail in a Graph



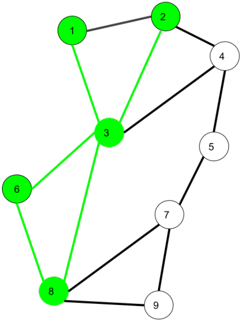
- Trail is an open walk in which no edge is repeated.

Trail in a Graph



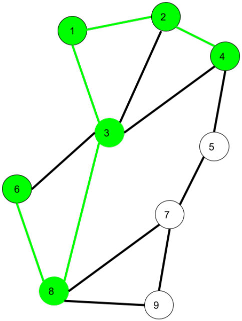
- Trail is an open walk in which no edge is repeated.
- Vertex can be repeated.

Trail in a Graph



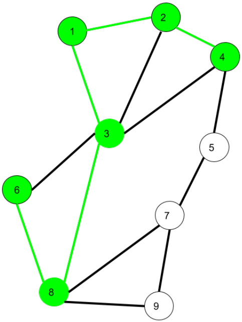
- Trail is an open walk in which no edge is repeated.
- Vertex can be repeated.
- $1->3->8->6->3->2$ is a trail

Path in a Graph



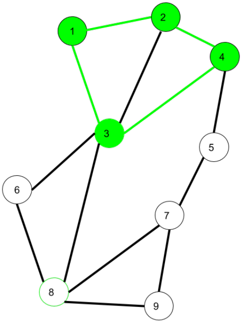
- Path is a trail in which neither vertices nor edges are repeated

Path in a Graph



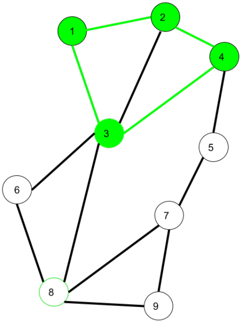
- Path is a trail in which neither vertices nor edges are repeated
- $6- > 8- > 3- > 1- > 2- > 4$ is a path

Cycle in a Graph



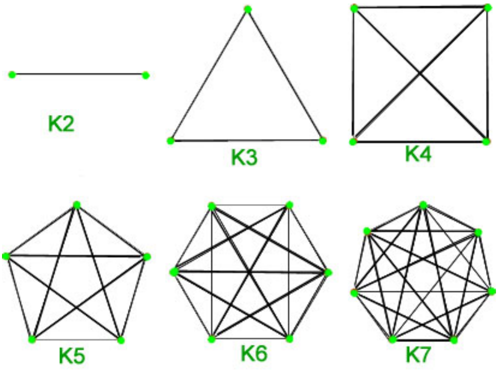
- Cycle is a path in which the first and last vertrices are same.

Cycle in a Graph

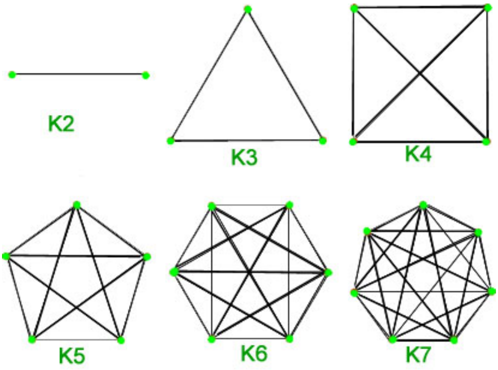


- Cycle is a path in which the first and last vertrices are same.
- No other vertex or edge gets repeated.

Complete Graph

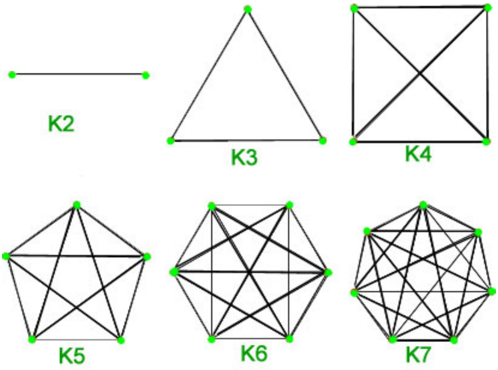


Complete Graph



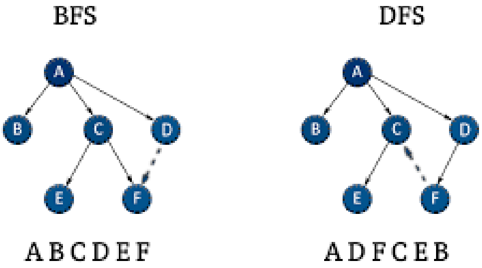
- It contains Edge between every pair of vertices.

Complete Graph



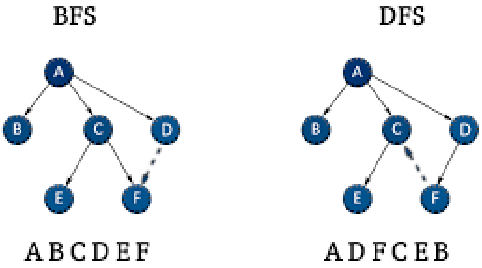
- It contains Edge between every pair of vertices.
- For an n vertex complete graph, the number of edges $= \frac{n(n-1)}{2}$

Graph Traversal



- We saw Preorder, Inorder and Postorder traversals for Binary Trees

Graph Traversal



- We saw Preorder, Inorder and Postorder traversals for Binary Trees
- In Graph we have
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

Graph Representations

Two types of Graph Representations are

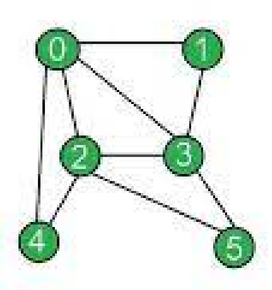
1. Adjacency Matrix

Graph Representations

Two types of Graph Representations are

1. Adjacency Matrix
2. Adjacency List

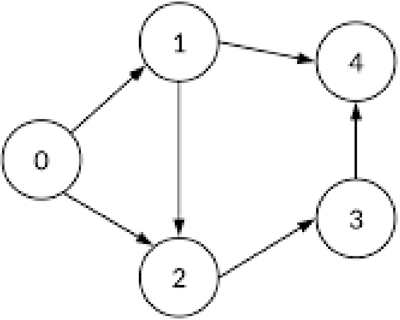
Adjacency Matrix(undirected)



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

- $M[i][j] = 1$, if $(i, j) \in E$

Adjacency Matrix(directed)

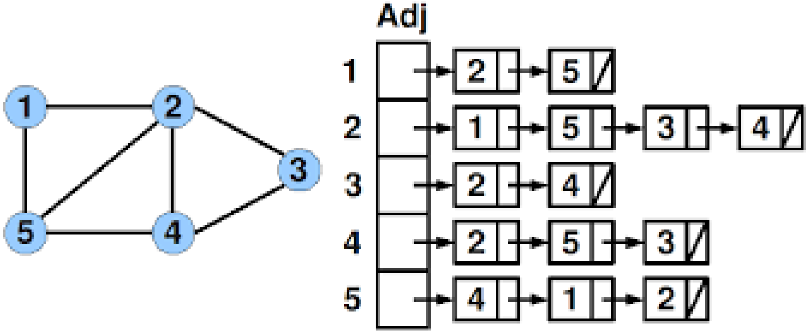


Adjacency Matrix

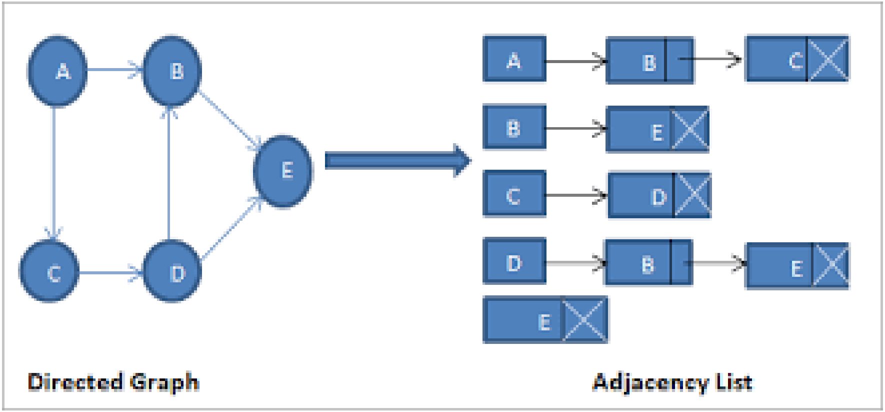
	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

- $M[i][j] = 1$, if $(i, j) \in E$

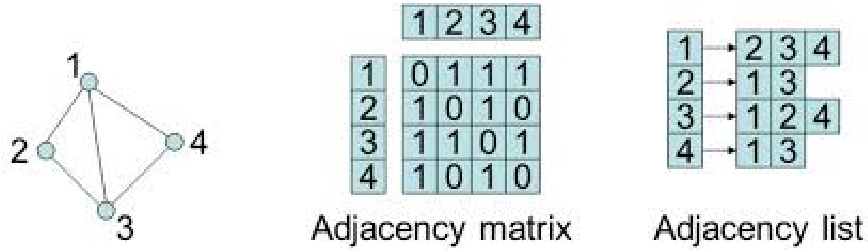
Adjacency List(undirected)



Adjacency List(directed)

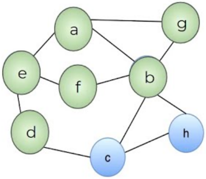


Adjacency Matrix &List



Breadth First Search(BFS)

Breadth First Search



Graph

a	e	b	g		
b	a	g	f	c	h
c	b	h	d		
d	c	e			
e	a	d	f		
f	e	b			
g	a	b			
h	b	c			

Adjacency List

Visited

a e b g d f

BFS order

a e b g d f c h

Q ⇦ | b | g | d | f | ⇦

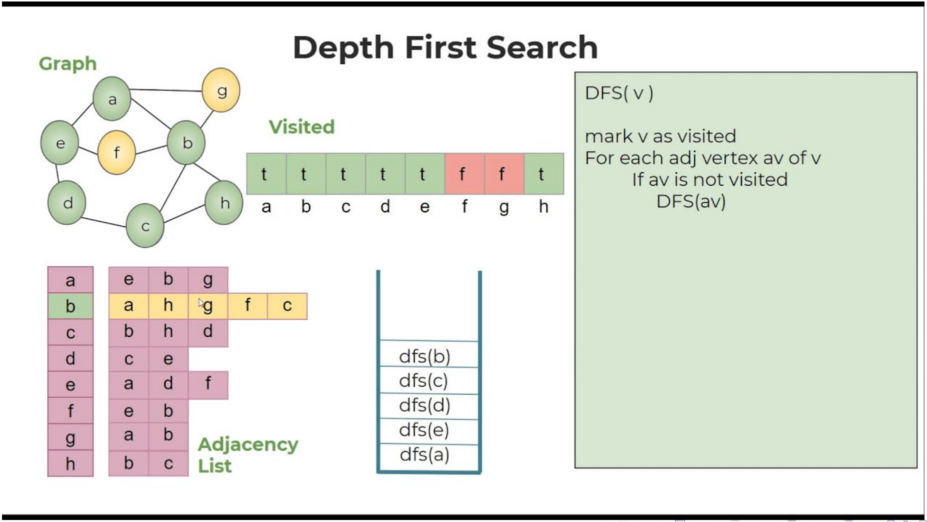
v | e |

BFS

add start vertex to Q
mark start as visited

while Q not empty
    v ← dequeue from Q
    for each adj vertex av of v
        //.... process vertex av....
        if av is not visited
            add av to Q
            mark av as visited

Depth First Search(DFS)



Lecture 36

Recap & Goals

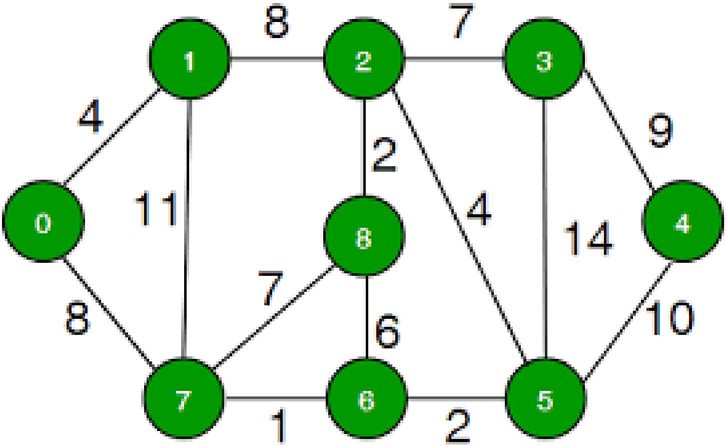
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees, Representation and Traversals
  - Binary Search Tree, Expression Tree, Heap Tree
  - Graphs, Graph Terminologies, Graph Traversals

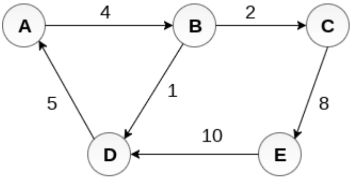
Today We Will See...

- Dijkstra's Shortest Path Algorithm, Minimum Spanning Tree.

Weighted Graph



Weighted Graph- Adjacency Matrix

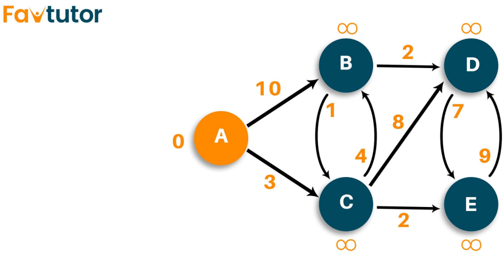


Weighted Directed Graph

$$
\begin{array}{c c}
 & \begin{array}{ccccc} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \end{array} \\
\begin{array}{c} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \\ \mathbf{D} \\ \mathbf{E} \end{array} &
\begin{bmatrix} 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 8 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \end{bmatrix}
\end{array}
$$

Adjacency Matrix

Dijkstra's Algorithm Example

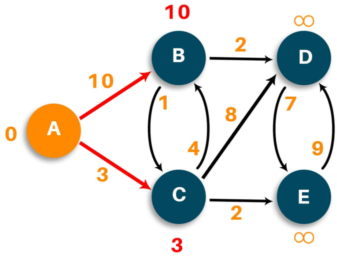


Q:

A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞

Dijkstra's Algorithm Example

Favtutor

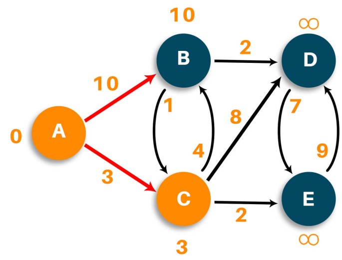


Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	3	∞	∞

S: {A}

Dijkstra's Algorithm Example

Favtutor

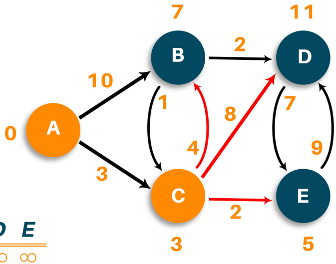


Q:

A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
	10	$\boxed{3}$	∞	∞

S: {A, C}

Dijkstra's Algorithm Example



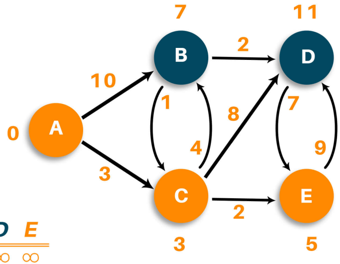
Q:

A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
	10	$\boxed{3}$	∞	∞
	7		11	5

S: {A, C}

Dijkstra's Algorithm Example

Favtutor

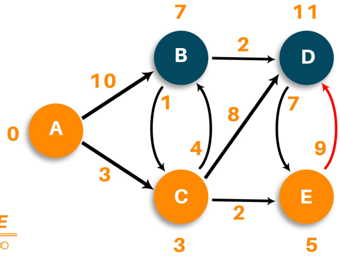


Q:

A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
	10	$\boxed{3}$	∞	∞
	7		11	$\boxed{5}$

S: {*A*, *C*, *E*}

Dijkstra's Algorithm Example

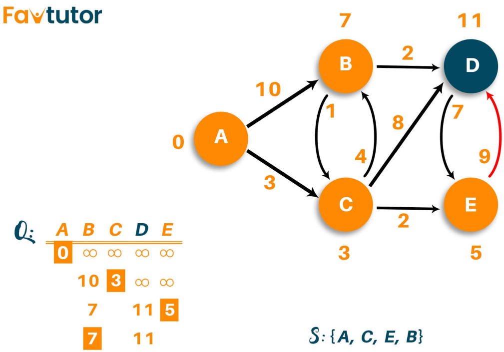


Q:

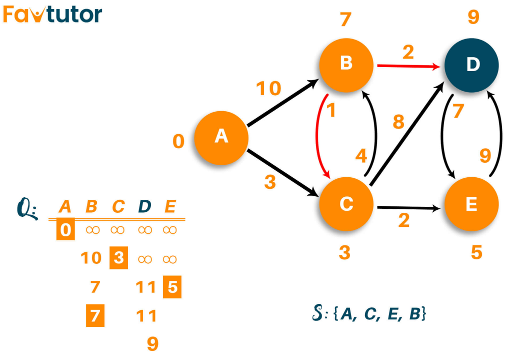
A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
	10	$\boxed{3}$	∞	∞
	7		11	$\boxed{5}$
	7		11	

S: {A, C, E}

Dijkstra's Algorithm Example

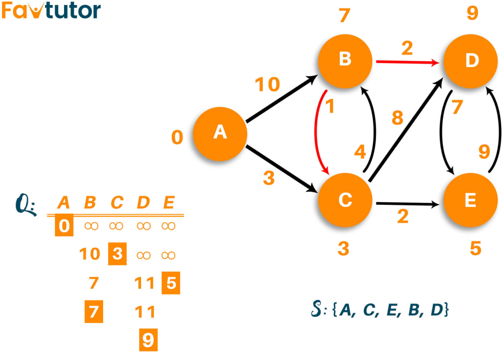


Dijkstra’s Algorithm Example



Dijkstra's Algorithm Example

Favtutor



Dijkstra's Algorithm Pseudocode

```
for each vertex v in Graph.Vertices:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
dist[source] ← 0

while Q is not empty:
    u ← vertex in Q with min dist[u]
    remove u from Q

    for each neighbor v of u still in Q:
        alt ← dist[u] + Graph.Edges(u, v)
        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
```