Date
Page

S.N	Units	Marks	Page No
1.	Introduction	4	6-9
2	Stack And Queue	11	10-19
3	List	14	20-27
4	Recursion	5	28--31
5	Trees	11	31-43
8	~~Sorting~~ Graph	12	44-51
6	~~Search~~ Sorting	12	52-58
7	~~Graph~~ Searching	12	59-65
		Total = 80	

Unit-1: I

Data struc
It is wa
efficientl
relations
mathema
It main

P

Linear
are

Non
that

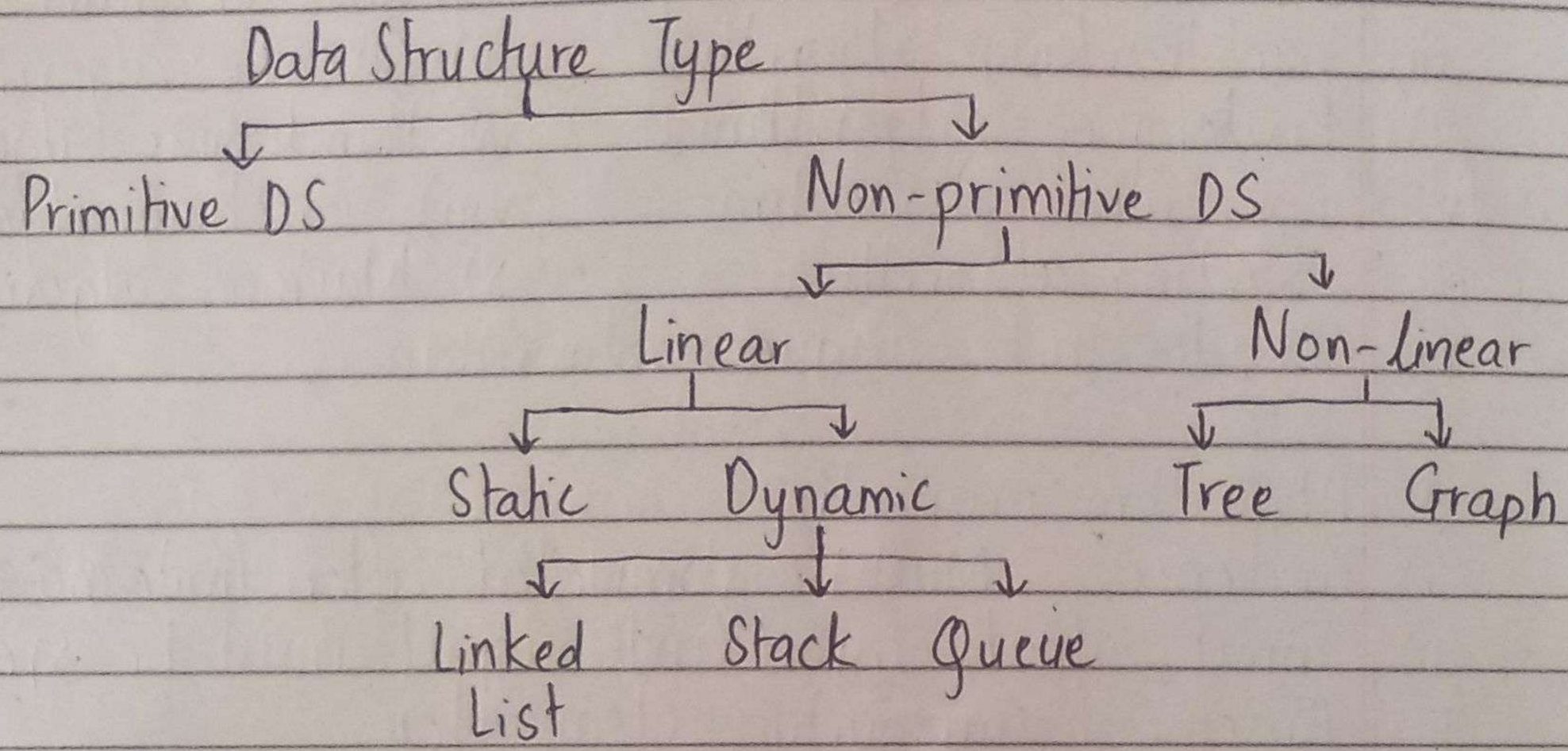
Oper
i) Sear
v) Upd

Algo
tak

Unit-1: Introduction

Data structure

It is way to store and organize data so that it can be efficiently used. It is structure representation of logic relationship between elements and data. It is logicall mathematical model of particular organization of data. It mainly specifies organization of data.



Linear DS: Data structure is called linear if all its elements are arranged in linear order.

Non linear DS: This data structure doesn't perform sequence that each item/element is connected with one or more item.

Operations performed in DS

i) Searching ii) Sorting iii) Insertion iv) Deletion
v) Updation vi) Traversing

Algorithm: It is sequence of steps executed by computer that takes input and transforms it into target output.

- How do data structure and algorithm work together?

Ans: Data structures hold the data while algorithms solve problem using data. They are one of the foundat discipline in the computer science.

* Types of algorithm

i) Recursive algorithm
ii) Dynamic programming algorithm
iii) Backtracking algorithm
iv) Greedy algorithm
v) Bruteforce algorithm
vi) Randomize algorithm
vii) Encryption Algorithm
viii) Searching algorithm
ix) Sorting algorithm
x) Mashing algorithm
xi) Divide and Conquer algorithm

* Algorithm properties

i) Finiteness: Must be completed after finite no. of steps
ii) Input/output: Input and output must be specified
iii) Absence of ambiguity: Clear steps
ix) Feasibility: Each instruction should be executed

Example of Algorithm to add two numbers

Step 1: Start
Step 2: Read A and B
Step 3: Calculate C = A+B
Step 4: Print C
Step 5: Stop

Write an algorithm to find if number is +ve or -ve

Step 1: Start

Step 2: Read number N
Step 3: If (N>0), print " N is +ve".
Else if (N==0), print "N is neither +ve nor -ve".
Else, print " N is -ve".
Step 4: Stop

Types of algorithm

i) Recursive algorithm: It is one of the simplest to devise algorithm as it doesn't require to think about every sub problems. Eg:

```
// factorial of number
int fact (int n) {
    if(x==0)  return 1;
    return x * fact (x-1);  }
```

ii) Divide And Conquer Algorithm: It is one of the most used algorithm that divides a problem into sub problems and solve each of them and combine them to form solution. It is extensively used as it is quite stable and optimal for most of problems asked.

iii) Greedy algorithm: In this, a decision is made that is good at that point without considering future. The two properties are
- Greedy choosing best option
- Optimal substructure property: If an optimal solution can be found by retrieving optimal solution to its sub problems. It doesn't always work but if it works, it works like

iv) Backtracking algorithm: It is form of recursion to backtrack to previous option for solution when a given option can't do so.

v) Randomized algorithm: It is algorithm that makes its decision based on random numbers that it uses in its logic. Eg: choosing pivot element in quick sort

vi) Bruteforce algorithm: It is ~~dev~~ algorithm devised to solve a problem

Eg:
```
//searching for element in sorted array
int main () {   int num = 5;
                boolean isFound = false;
                for (int i=0; i<5; i++){   //   O(n)
                    // code }
```

Algorithm to make telephone call

Step 1: Start
Step 2: Read telephone number
Step 3: Lift the receiver
Step 4: Is there a dial tone?
If yes, then dial telephone. Go to step 5.
If no, put down receiver. Go to step 3.
Step 5: Speak and put down receiver.
Step 6: Stop

Algorithm to calculate interest

Step 1: Start

Step 2: Read principal, rate and time.
Step 3: Multiplying them all and divide by 100
Step 4: Print out answer.
Step 5: Stop

Tools to write algorithm
- Code Editor / IDE (Integrated Development Environment)

Asymptotic Notation : It describes running time and space complexity of an algorithm for infinity 'n'.
i) Big O ii) Big Ω iii) Big θ

Types of analysis
i) Time complexity: It refers to amount of time taken for algorithm to complete as a function of size of input data.
ii) Space complexity : It refers to amount of memory space an algorithm takes to complete as a function of size of input data.

Algorithm analysis: It is process of evaluating performance of algorithm in terms of its time and space complexity. It is essential for predicting behaviours of algorithms and selecting best one for intended purpose. It helps to understand how an algorithm will perform in different situations.

i) Big O notation
It is mathematical notation to describe upper

bound of worst case time complexity of an algorithm is denoted by $O(n)$.

ii) Big Ω notation (Omega)

It is mathematical notation to describe low bound of best case time complexity of an algorithm denoted by $\Omega(n)$

iii) Big θ notation (Theta)

It is mathematical notation to describe average case time complexity of an algorithm denoted by $\theta(n)$.

Unit-1

Algorithm

An algorithm is a step by step procedure or set of rules for solving a specific problem or performing a particular task. It is a well defined computational procedure that takes some input, process it and produce desired output.

Types of algorithms

i) Sorting Algorithms: This algorithm that takes a collection of elements and rearrange them in specific order (ascending or descending). Eg: bubble sort, quick sort

ii) Searching algorithms: This algorithm search for particular

element in
value or
search, bi
iii) Graph al
like find
life proble
1) Shortest p
2) Minimum
3) Connectivit

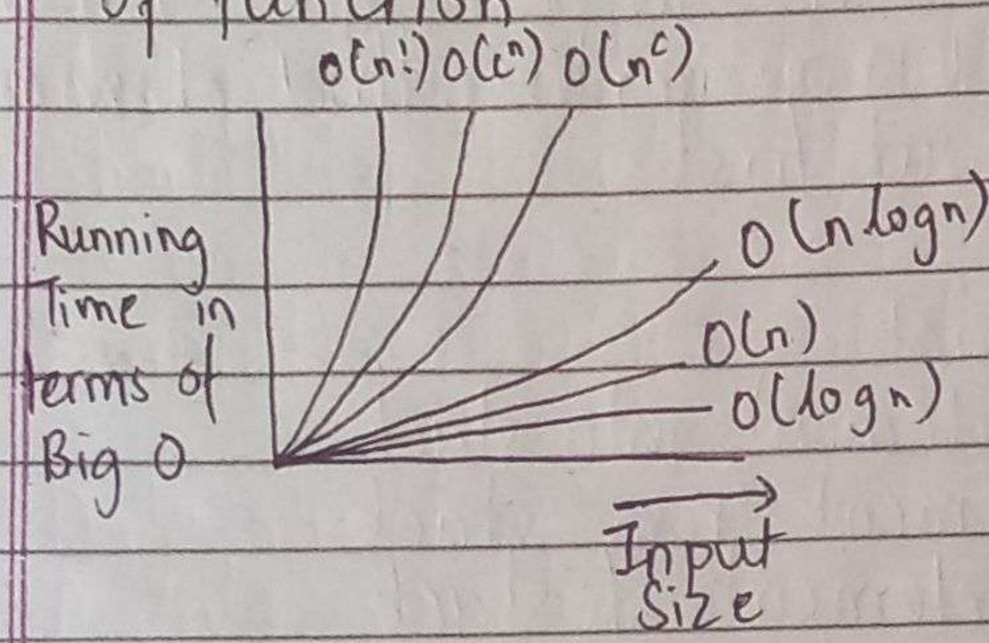
Data Stru
- A data s
data on
- A data
data on
updated

Types of
- Linear dat
elements
is attac
- Non lin
elements
can't
Eg: fro

i) Arrays:
or key.
element

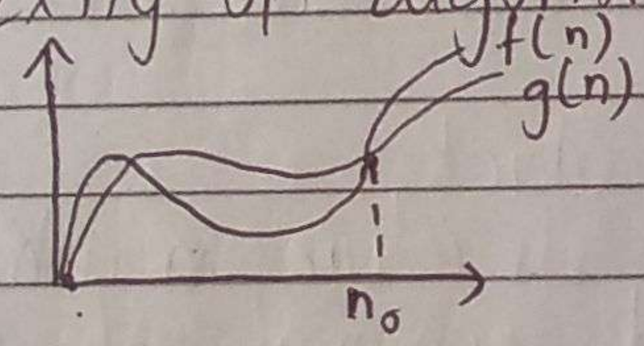
Asymptotic notation

i) Big O (worst case): It represents upper bound of time or space complexity of algorithm. It provides an asymptotic upper bound on growth rate of function

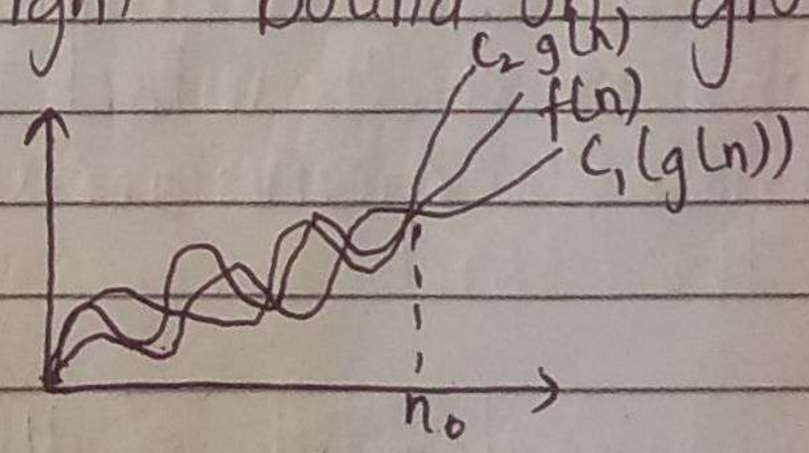


$O(n!), O(c^n), O(n^c)$ - worst
$O(n \log n)$ - Bad
$O(n)$ - Fair
$O(\log n)$ - Good
$O(1)$ - Best

ii) Big Ω (best case): It represents lower bound of time or space complexity of algorithm.



iii) Big Θ (average case): It represents both upper and lower bound of time or space complexity of an algorithm. It provides a tight bound on growth rate of function



element in data structure such as array. It finds value or record that user demands. Eg: linear search, binary search

iii) Graph algorithms: It finds solutions to problems like finding shortest path betn cities and real life problems like travelling sale man problems. Eg:
1) Shortest path algorithm (Dijkstra, Bellman-Ford A")
2) Minimum spanning algorithm (Kruskal, Prim)
3) Connectivity algorithm (eg: BFS, DFS).

Data Structure

- A data structure is storage used to store and organize data on computer
- A data structure is a way of arranging and storing data on computer so that it can be accessed and updated efficiently.

Types of data structure

- Linear data structure: Data structure in which data elements are arranged linearly where each element is attached to its previous and next adjacent element
- Non linear data structure: Data structure where data elements aren't placed sequentially. In this, we can't traverse all the element in a single run only. Eg: tree, graph

i) Arrays: A collection of elements each identified by index or key. It offers constant time access to any element based on index

ii) Linked list: It is linear collection of elements where each element point to next one in sequence. They are dynamic in size and allow for efficient insertion or deletion.

iii) Stack: Last In First Out (LIFO) data structure where elements are added and removed from same end known as top. Common operations are push, pop, etc

iv) Queue: First In First Out (FIFO) data structure where elements are added and removed from front.

v) Trees: Hierarchial data structure with root element and a set of child elements.

vi) Graphs: It is collection of nodes and edges connecting those nodes. It can be directed or undirected and used to represent relationships

Types of Algorithm Analysis

i) Time Complexity: It measures actual time to run the algorithm. It can be implemented using built-in functions

ii) Space Complexity: It is total space taken by algorithm with respect to input size. It includes both auxiliary space (temporary) and space used by input.

Unit-2
Stack And Queue

Stack: It is linear data structure that follows a particular order LIFO in which operation are performed. The order LIFO implies that element is inserted last comes out first.

Basic Operations on Stack

i) push(): to insert an element into stack.
ii) pop(): to remove element from stack
iii) top(): Returns top element of stack
iv) isEmpty(): returns true if stack is empty else false
v) size(): returns size of stack

Implementation of Stack

i) A stack is data structure in which insertion of a new element and removal of an existing element takes place at same end represented as top of stack
ii) To implement stack, it is required to maintain pointer to top of stack which is last element to inserted because we can access the element only on top of the stack.

1) Top : returns top element of the stack

Algorithm for Top

begin
return
end pro

2) Push: Adds
to be an

Algorithm
begin
if stack
retu
endif
else
in

end
end pro

3) Pop: remove
order in
then it

Algorithm
begin
if sta
re
endif
else
s

```
begin
    return stack [top]
end procedure
```

2) Push: Adds an item to stack. If stack is full, then it is said to be an over flow condition.

Algorithm for Push

```
begin
    if stack is full
        return
    endif
    else
        increment top
        stack [top] assign value
    end else
end procedure
```

3) Pop: removes an item for stack. Items are popped in reverse order in which they are pushed. If stack is empty, then it is called underflow condition.

Algorithm for pop

```
begin
    if stack is empty
        return
    endif
    else
        store value of stack [top]
```

```
        decrement top
        return value
    end else
end procedure
```

4) isEmpty() : Returns true if stack is empty, else false.

Algorithm for isEmpty:

```
begin
    if top<1
        return true
    else
        return false
end procedure
```

Implementation of stack using C

```
# include <stdio.h>
# define N 10
int stack [N];
top = -1;

void push (int x) {
    if (top > (N-1)) { printf ("Stack is full.") }
    else { top ++;  stack [top] = x;
            printf(" %d is pushed into stack", x); }
}

void pop( ){
    if (top<0) { printf (" Stack is empty") }
    else { int x = stack [top];  top--;
```

```
    pr
void pee
    i
    e
}
void displ
    e
}
int main
    push
```

Output:

Applicati
- Infix ex
  (a+b)
- Postfix

```
    printf(" %d is deleted", x); }

void peek (){
        if (top<0) { printf("Stack is empty"); }
        else { int x = stack [top];
                printf("%d is top element in stack", x); }
}

void display (){
        if (top<0) { printf ("Stack is Empty"); }
        else {
                for (int i = top ; i<=0 ; i--){
                        int x = stack [i];
                        printf("%d "); }
}

int main (){
    push(5);    push(4);  peek();  display();  pop(); }
```

Output:

Application of stack

- Infix expression: The expression of form "a operator b" (a+b) i.e operator betn every pair of operands
- Postfix expression: The expression of form "a b operator"

(abt) i.e ~~when~~ operator after ever pair of operand

Eg: Input: A+(B×C)-D
Output: ABC×+D-

Converting Infix to Postfix Expression

1) Scan infix expression from left to right
2) If scanned character is operand, put it in postfix expression
3) Otherwise do following:
   i) If precedence and associativity of scanned operator are greater than operator in stack, then push into stack.
   ii) Else, pop all operators from stack which are greater than or equal to in precedence than that of scanned operator
4) If scanned character is '(', push into stack.
5) If scanned character is ')', pop stack and output it until '(' is encountered and discard both parenthes
6) Repeat steps 2 to 5 until infix expression is scanned.
7) Once scanning is over, pop the stack and add th operators in postfix expression until it is not empty
8) Finally, print postfix expression

Queue And Operation

A Queue is defined as linear data structure that is open at both ends and operations are performed in First In First Out (FIFO) order.

* Queue Representation

- Queue : name of array storing queue elements
- Front: index where first element is stored in array representing queue
- Rear: index where last element is stored in array representing queue.

* Basic Operations

1) Enqueue() : Add element to end of queue
2) Dequeue() : Removal of elements from queue
3) Peek() / Front(): Acquires data element available at front node. of queue without deleting it
4) Rear(): Returns element at rear end without deleting
5) isFull(): Validates if queue is full
6) isNull(): checks if queue is empty

1) Enqueue:

Step 1: Check if queue is full
Step 2: If queue is full, return overflow error and exit
Step 3: If queue isn't full, increment rear pointer to pointer to point to next empty space
Step 4: Add data element to queue location where rear is pointing
Step 5: Return success

2) Dequeue

Step 1: Check if
Step 2: If queue exit
Step 3: If queue front is p
Step 4: Increment data elem
Step 5: Return

Implemen

```
# include
# define
int Q[
int front =
void en
```

2) Dequeue

Step 1: Check if queue is empty

Step 2: If queue is empty, return underflow error and exit

Step 3: If queue isn't empty, access the data where front is pointing

Step 4: Increment front pointer to point next available data element

Step 5: Return success

Implementation of queue using C

```
# include <stdio.h>
# define N 10
int Q[N];
int front = -1, rear = -1;

void enQ (int x) {
```

Linear Queue

It is most basic version of queue. Here, insertion of an element i.e Enqueue operation takes place at rear end and removal of an element i.e it it takes place at front end.

Here problem is that if we pop some item from front and then rear reach to the capacity of queue and although there are empty space before front means queue isn't full but as per condition in isFull() function, it will show the

queue is full then. To solve this, we use circular queue.

Circular queue

It is special type of queue where last position is connected back to the first position. Here also operations are performed in FIFO order.

In circular queue, element of queue acts as circular ring. Working of circular queue is similar to the linear queue except for the fact that last element is connected to first element.

Its advantage is that memory is utilized in better way. This is because if there is an empty space i.e if no element is present at certain position in queue, then element can be easily added at that position using modulo capacity (i%n)

Priority Queue

It is special type of queue. Its speciality is that it arranges elements in queue based on some priority. The priority can be something where element with highest value has priority, so it creates a queue with decreasing order of values. The priority can also be such that element with lowest value gets highest priority so in turn it creates a queue with increasing order of values. In pre-define priority queue, C++ gives priority to highest value whereas Java gives priority to lowest value.

Unit-3
List

Linked List

A linked list is linear data structure that stores a collection of data elements dynamically (non-contigous). It is dynamic arrangement that contains a link to the structure containing subsequent items called node. Node represents those data elements and links/pointers connect each node. Each node consists of 2 fields the information stored in linked list and a pointer that stores the address of its next node. The last node contains NULL in its second field because it will point to no node. Linked list can grow and shrink its size as per the requirement.

Representation of Linked List

This representation depicts that each node consists of 2 fields i.e data and pointer (points to another node. The start pointer stores address of the first node and at the end, there is null pointer that states the end of linked list.

Advantages

i) Dynamic data structure: It can grow and shrink at runtime by allocating and deallocating memory.

ii) No memory wastage: No need to pre-allocate memory
iii) Scalability: Contains ability to add or remove elements at any position
iv) Insertion and Deletion Operations: no need to shift elements after insertion or deletion of an element only the address present in next pointer needs to be updated.
v) Efficient for large data and flexible

Disadvantages
i) Random Access: random access isn't possible due to link betn nodes
ii) Complex Implementation: Requires complex programming understanding
iii) Memory Usage: Requires extra memory for pointer to save address of the next node
iv) Lower efficiency: Searching or looping can be slower
v) More time consuming traversal
vi) Reverse traversing isn't possible

Operations in Singly Linked List
Insertion at the beginning and end, after node, before node
A singly linked list is linear data structure in which elements aren't stored in contigous memory locations and each element is connected only to its next element using pointer.

Inserting at beginning
- Create a memory for new node
- Store data in a new node
- Change ~~head~~ ~~to~~ next of new node to head
- Change head to tell recently created node.

Inserting at the end
- Insert a new node and store data in it
- Traverse the last node of linked list
- Change next pointer of last node to newly created node

Inserting at middle
- Check if the given node exists or not
- If not, terminate process.
- Else, make element to be inserted as new node. Change next pointer of given node to new node. Now, shift original next pointer of given node to next pointer of new node.

* Deletion at start, end, after and before node

Deletion at start
- Store address of first node in pointer
- Move head node to the next node
- Dispose or free memory of pointer node

Deletion at the end
- Traverse linked list to second element
- Change its next pointer to null
- Free memory of the last node

Deletion at Middle
- Store address of deleted node as key node
- Store address of preceding node in pointer. eg P
- Store address of key node in pointer. eg. X
- Make successor of key node as successor of the node pointed by P
- Free node X

* Definition and structure of Doubly Linked List (DLL)

A DLL is special type of linked list in which each node contains pointer to previous node, data and next node's pointer of the linked list. This allows for bidirectional traversal of the list.

Advantages
- Bidirectional traversal (forward & backward)
- Deleting previous node using previous pointer is possible

Disadvantages
- Requires extra space for previous pointer
- All operation require extra pointer to be maintai

Applications
- Used by web browser to backward & forward web pa
- In OS, it is used by thread scheduler to track processe
- LRU / MRU (Most Recently Used) Cache is constructed.

* Operations in Doubly Linked List

1) Insertion

i) Inserting at the start
- Firstly allocate a new node
- Now put required data in it.
- Make next of new node point to current head of DL
- Make previous of the current head point to new no
- Lastly, point head to new node.

ii) Insertin
- Create a
- Make n
- If list i
- Else, tr
- Now ma
new no
- Change
of the l

iii) Insertin
- Allocate
- Put da
- Set pre
of the
- Set p
- Set
- Now,
1) If pr
point
2) Else
new

iv) Inse
- Cre

ii) Inserting at the end
- Create a new node and put data in it
- Make next pointer of new node as null
- If list is empty, make new node as head
- Else, travel to end of DLL.
- Now make next pointer of last node point to new node.
- Change previous pointer of new node to last node of the list.

iii) Inserting a node before given node
- Allocate memory for the new node.
- Put data in it.
- Set previous pointer of new node as previous node of the next node
- Set previous pointer of next node as new node.
- Set next pointer of new node as next node
- Now, set previous pointer of new node.
1) If previous node of new node isn't NULL, set next pointer of this previous node as new_node.
2) Else, if prev of new node is NULL, it'll be new head node.

iv) Inserting node after given node in DLL
- Create a new node

- Put data in it
- Point next of new node to next of previous node
- Point next of previous node to new node
- Point previous of new node to previous node
- Change pointer of ~~new~~ next node's previous pointer to new node

2) Deletion

i) Deleting at the start

- Let node to be deleted be temp
- Copy head pointer to temp (temp = head)
- Shift the head pointer to its next (head = head → nex
- Make previous node pointer of this new head node point to NULL.
  head → prev = NULL;
- Free the pointer temp. [free(temp)]

ii) Deleting at the end

- Check if list is empty or not
  If head = = null, terminate
- Else, let node to be deleted be temp (temp = head)
- Traverse list to reach last node and the temp would point to last node of the list at the end of loop
  while (temp → next != NULL) { SET temp = temp → next
- Make next pointer of previous node of temp to NU
  SET temp → prev → next = NULL
- Free the pointer temp.
  free (temp)

iii) Delete

- Check
  If h
- Else,
  tem
- Traver
  while
- Check
  If t
- Make
  temp
- Make
  tem
- Free

iii) Delete from middle

- Check if list is empty or not
  If head == null, terminate
- Else, let node to be deleted be temp.
  temp = head
- Traverse list until found desired data
  while (temp -> data != data) { temp = temp -> next }
- Check if found node is last or not.
  If temp -> next = NULL, terminate
- Make next of previous node of temp point to next node
  temp -> prev -> next = temp -> next
- Make previous of next node of temp point to prev node
  temp -> next -> prev = temp -> prev
- Free pointer temp
  free(temp)

Unit-4

Recursion

Recursion is programming and mathematical concept in which function calls itself during its execution. A recursive function is function which solves a problem by solving smaller instances of same prob. The process continues until a base case is reach at which point the function returns a result without making further recursive calls. It solves problems like tower of hanoi (TOH), fibonacci and factorial, traversing trees and graphs, etc.

Properties of Recursion

- It performs same operation multiple times
- Function calls itself with simplified version of the original problem
- Base case is needed to stop recursion. Otherwise, infi loop will occur.

* Recursion Vs Iteration

	Recursion	Iteration
Defn.	Function calls itself	Repetition of set of instructions
Application	For functions	For loops
Termination	Through base case, it ends.	When termination condition is it ends.
Code Size	Smaller code size	larger Code Size
Speed	Slow execution	Faster execution
Stack	Uses stack to store variables	No use of stacks
Memory	Uses more memory	Uses less memory

Space Complexity

Higher space complexity	Lower space complexity

* Tower of Hanoi and its solution

Tower of Hanoi (TOH) is mathematical puzzle where we have 3 rods (A, B and C) and N disks. Initially, all disks are stacked in decreasing value of diameter i.e smallest disk placed on top of rod A.

The objective of this puzzle is to move entire stack to another rod (C) obeying rules i.e

- Only one disk can be moved at a time
- No disk may be placed on top of smaller disk
- Disk can only be moved if it is the top disk on stack

Algorithm

Moving n disks from one rod to another can be broken down into 3 steps:

1) Move n-1 disks from source to auxiliary rod
2) Move largest disk from source to destination
3) Move the n-1 disks from auxiliary rod to destination

Pseudocode:

```
tower_of_hanoi (n, source, target, aux_rod) :
    if n==1
        move_disk (source, target)
    else
        tower_of_hanoi (n-1, source, aux, target)
        move_disk (source, target)
        tower of hanoi (n-1, aux, target, source

move_disk (source, target)
    print ("Move disk from", source rod, "to", target)
```

Solution of fibonacci sequence and factorial

Fibonacci sequence is series of number in which each number is sum of two preceeding ones usually starting with 0 and 1.

Solution:

```
fibo (n)
    if n <= 1.
        return n
    else
        return fibo (n-1) + fibo (n-2)
```

Factorial of non negative number (n!) is product of all positive integers less than or equal to n.

Solution:
```
fact (n)
    if n<=1
        return 1
    else
        return n x fact (n-1)
```

Unit - 5
Trees

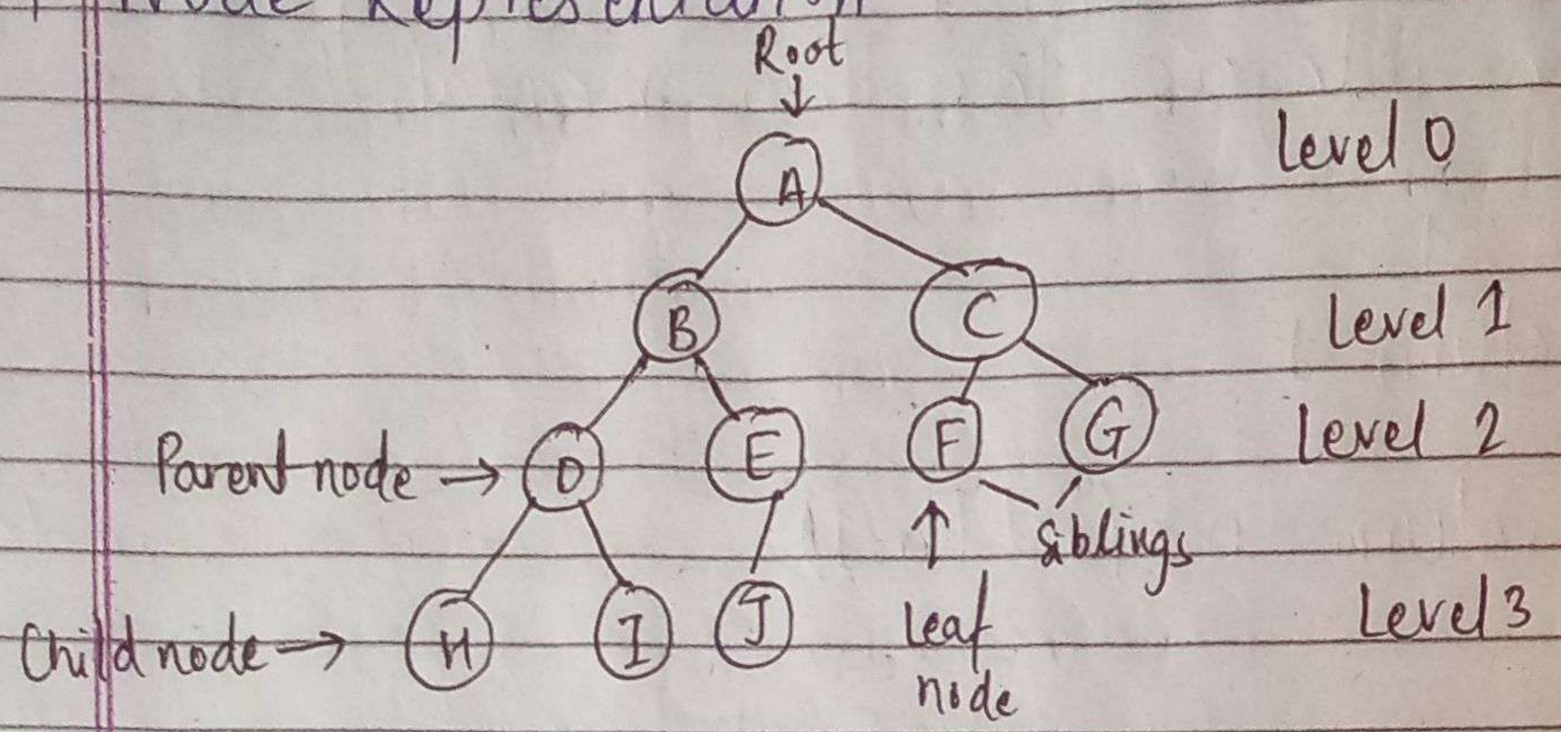
5.1 Tree Concepts

A tree is a hierarchial data structure that consists of nodes connected by edges. Each node in tree has a parent child relationship with other nodes except for topmost node called root which has no parent and nodes at bottom which have no children.

Trees provide efficient way to represent hierarchial structures and are used in many applications like file system, network routing algorithms.

→ Node: Basic building block of tree containing data & reference to its child nodes
→ Parent: A node in a tree that has one or more child nodes
→ Child: A node in tree that has parent node
→ Subtree: A tree formed by a node and its descendants
→ Depth: The level of node from root. Depth of node is no. of edges from root to node. Depth of root is 0.

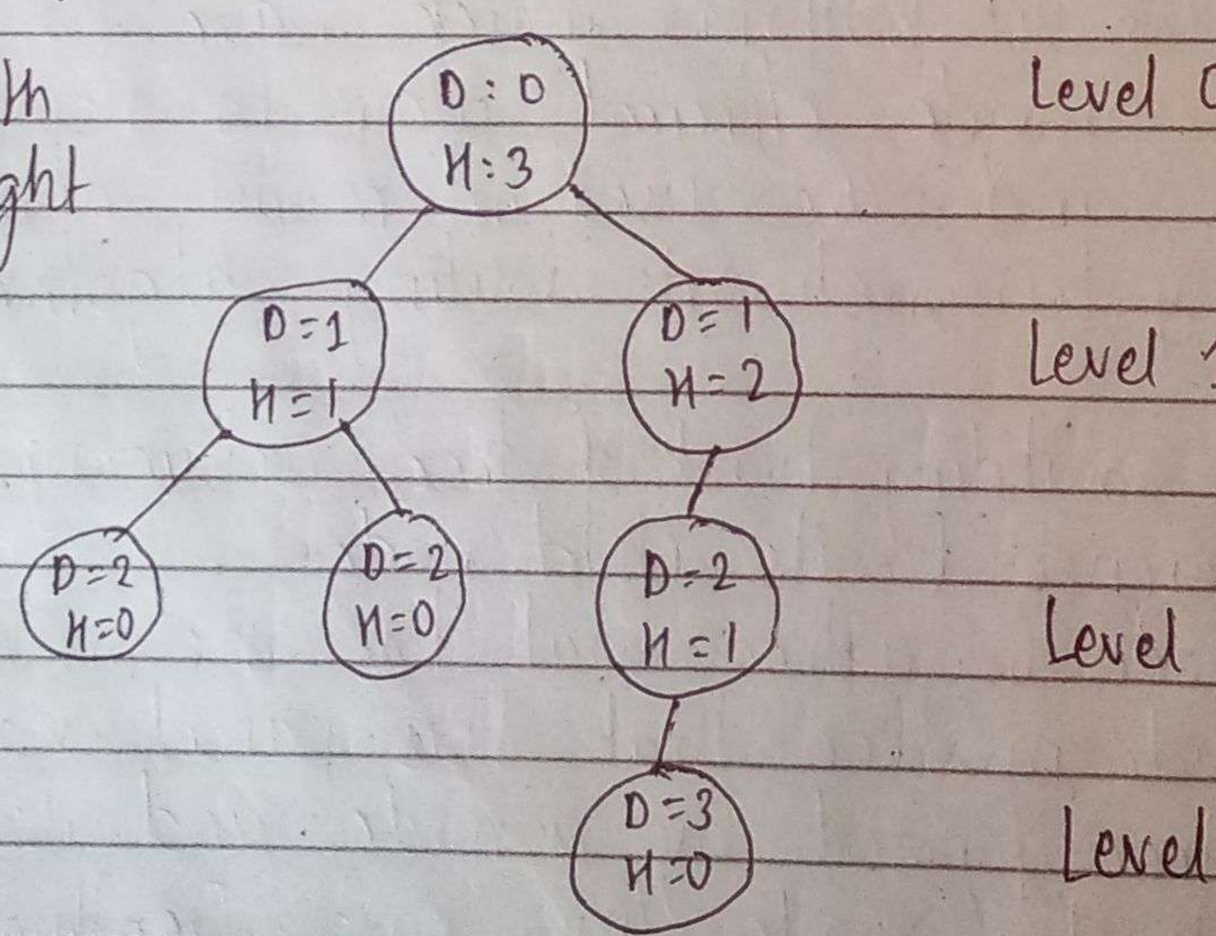
5.4 Node Representation



5.8 Height, Level, Depth of Tree And Its Importance

Level is no. of steps from top to bottom in tree. It starts with 0 and increments by 1 at each step. Depth is total no. of edges from root node to the leaf node in the longest path. Height is total no. of edges present in longest path connecting that node to leaf node

D : Depth
H : Height



Binary Tree

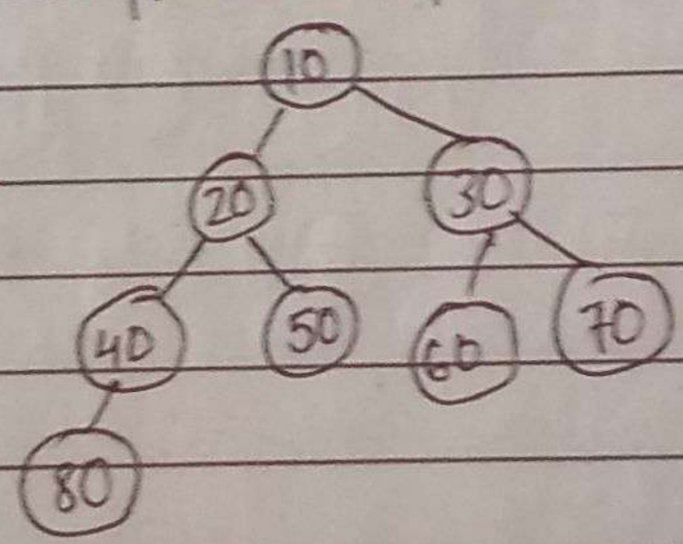
It is tree type non linear data structure with max of two children for each parent. Every node has left and right reference along with data element. The node at top of hierarchy is root node.

It has its applications in computer science including data storage and retrieval, expression evaluation, network routing and game AI. It can be used for algorithms like search, sort & graph.

Types of Binary Tree

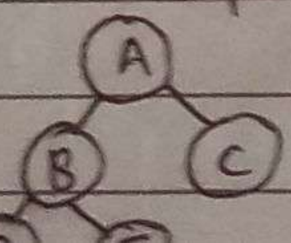
i) Complete Binary Tree

It is tree in which all nodes are completely filled except last level. In last level, all nodes must be left as possible. In complete binary tree, nodes should be added from left.

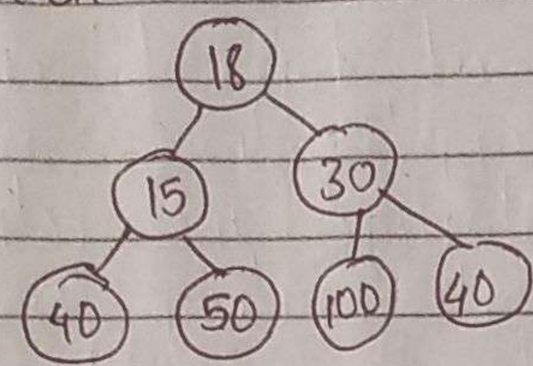


ii) Full Binary Tree

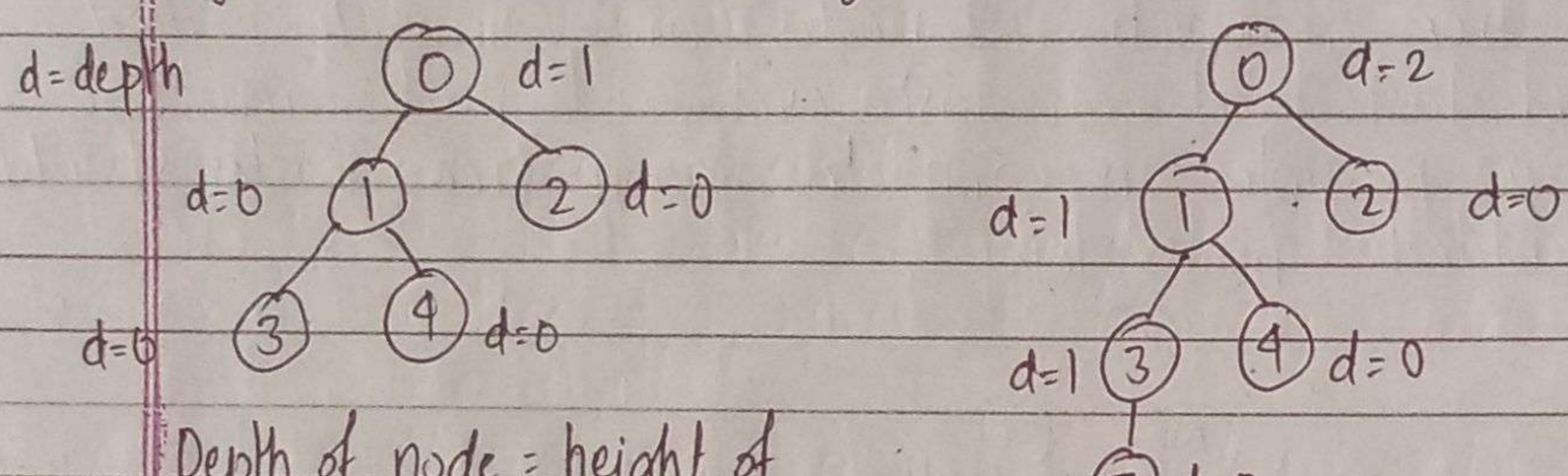
Also known as strict binary tree. A tree can be full binary if each node must contain either 0 or 2 children. It is tree in which each node must contain 2 children except leaf nodes.



iii) Perfect Binary Tree
It is binary tree type where all leaf nodes are on same level and every node except leaf nodes have 2 children.

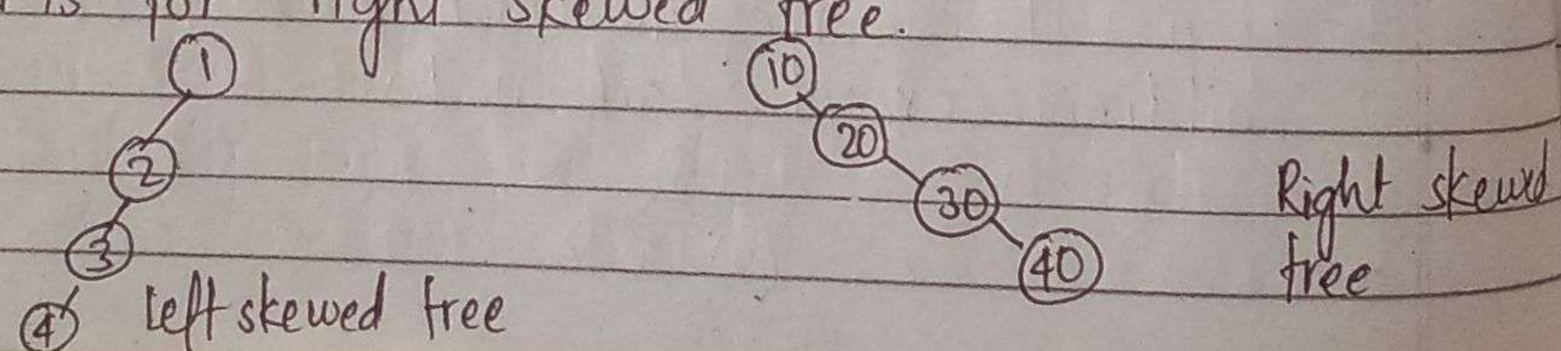


iv) Balanced Binary Tree
It is tree in which both left and right trees differ by at most 1. For eg: AVL and Red-Black Trees



Depth of node = height of (left child - right child)

v) Degenerate Binary Tree
It is tree in which all internal nodes can have only one children. If all nodes have only one child and stores as left child only, it is left skewed tree. Similar is for right skewed tree.



Left skewed tree

Right skewed tree

5.3 Applications
i) Binary Search
insertion, d
complexity
algorithms.
ii) Huffman co
and their
iii) File Systems
organize a
iv) Game Trees:
v) Network Ro
and direct
vi) DB indexing
DB indexin
and delete

5.7 Tree Traver
Traversal
may prin
via edges
we can't
ways fo
i) In-order

Pre-Ord
In this,
and rig
pre-ord

5.5 Applications of Binary Tree

i) Binary Search Tree (BST): It allows for efficient searching, insertion, deletion operations with average time complexity of O (logn) used in DBs and search algorithms.

ii) Huffman coding trees: It is used to represent characters and their variable length code for data compressions.

iii) File Systems: Used in hierarchial file systems to organize and represent directory and structure.

iv) Game Trees: It can model decision trees in games or AI scenarios like minmax algorithm

v) Network Routing Algorithms: It is used to find paths and direct data traffic

vi) DB indexing: Trees like AVL, Red-Black are used in DB indexing to perform efficient search, insertion and delete operations on large datasets.

5.7 Tree Traversal

Traversal is process to visit all nodes of tree and may print their values too. As all nodes are linked via edges, we always start from root node i.e we can't randomly access a node in tree. Three ways for tree traversal are:

i) In-order ii) Pre-order iii) Post-order

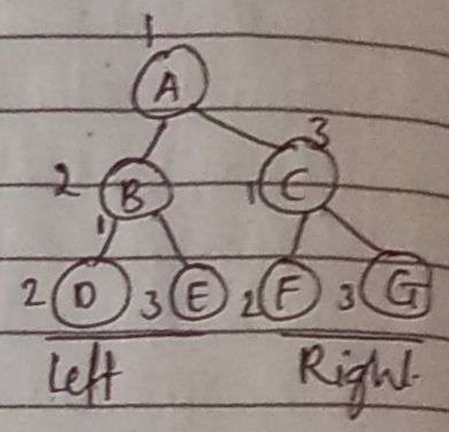
Pre-Order Tree Traversal

In this, root node is visited first and then left subtree and right subtree. We start from A and following pre-order traversal, we first visit A itself and move

on to its left subtree B. B is also traversed pre-order. This continues until all nodes are visited.

Algorithm

Step 1: Visit root node
Step 2: Recursively traverse left subtree
Step 3: Recursively traverse right "




```
preorder (struct node *root){
    print (root -> data)
    preorder (root -> left)
    preorder (root -> right)
}
```

Output of pre-order traversal : A → B → D → E → C → F → G

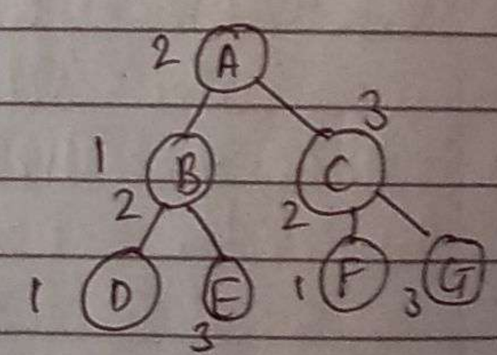
In-Order Tree Traversal

In this, left subtree is visited first, then root and later right subtree. Remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, output will produce sorted key values in ascending order.

Algorithm

Step 1: Recursively traverse left subtree
Step 2: Visit root node
Step 3: Recursively traverse right subtree




```
inOrder (struct node *root){
    inOrder (root -> left)
    ~~inOrder~~ print (root -> data)
    inOrder (root -> right)
}
```

Output of in-order traversal : D → B → E → A → F → C → G

Pre	root	left	right
In	left	root	right
Post:	left	right	root

Post Order Tree

In this, root n
is traversed, th
We start
we first visit B.
process goes w

Algorithm

Step 1: Recursively traverse
Step 2: " "
Step 3: Visit root node

```
postOrder (struc
    postOrder (r
    postOrder (r
    print (roo
}
```

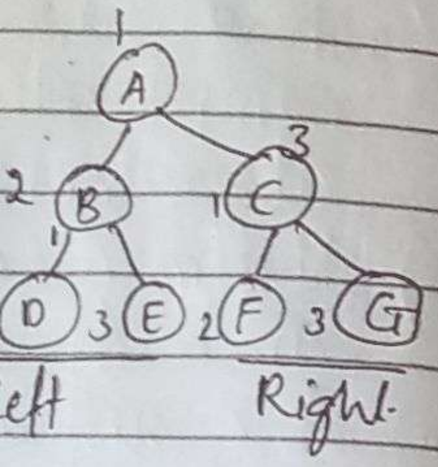
5.5 Operations in B

A binary tree i
node in tree
node in tree
pointer and r
need to creat

```
struct node {
    int data; s
}
```

In this, data
left node a
right node

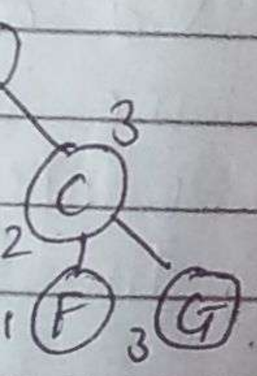
traversed pre-order
visited.



D → E → C → F → G

root and later
de may represent

er, output will
order



in-order traversal
→ A → F → C →

Pre	root	left	right
In	left	root	right
Post:	left	right	root

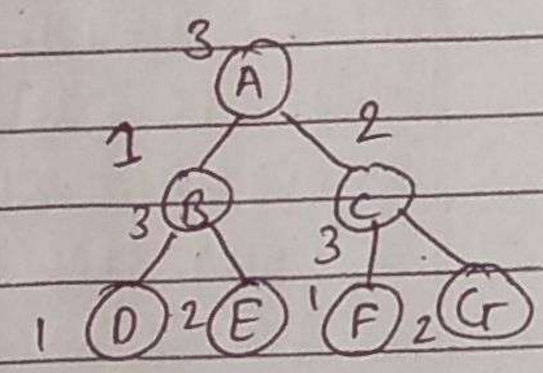
Post Order Tree Traversal

In this, root node is visited last. First left subtree is traversed, then right subtree and finally root node.

We start from A and following pre-order, we first visit B. B is also traversed post order. The process goes until all nodes are visited.

Algorithm

Step 1: Recursively traverse left subtree
Step 2: " " right "
Step 3: Visit root node




```
postOrder (struct node* root){
    postOrder (root->left)
    postOrder (root->right)
    print (root->data)
}
```

Output of post order
= D → E → B → F → G → C → A

5.5 Operations in Binary Tree

A binary tree is implemented using pointers. The first node in tree is represented by root pointer. Each node in tree consists of 3 parts i.e data, left pointer and right pointer. To create binary, we need to create the node as show below.

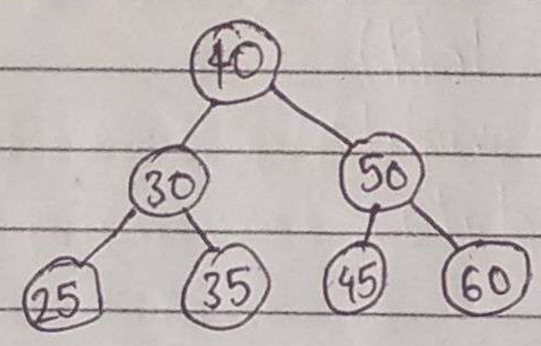
```
struct node{
    int data; struct node *left, *right;
}
```

In this, data is value, left pointer contains address of left node and right pointer contains address of right node.

Binary Search Tree

A binary search tree follows some order to arrange elements. In BST, value of left node must be smaller than parent node and value of right node must be greater than parent node. Ordering property of BST makes it suitable for efficient searching, insertion and deletion operations. BST is binary tree data structure with following properties:

i) Binary Tree Structure : Each node in BST can have at most 2 c
ii) Ordering Property - For each node 'n', all nodes in its left subtree have values less than n and all nodes in its right subtree have values greater than 'n'.



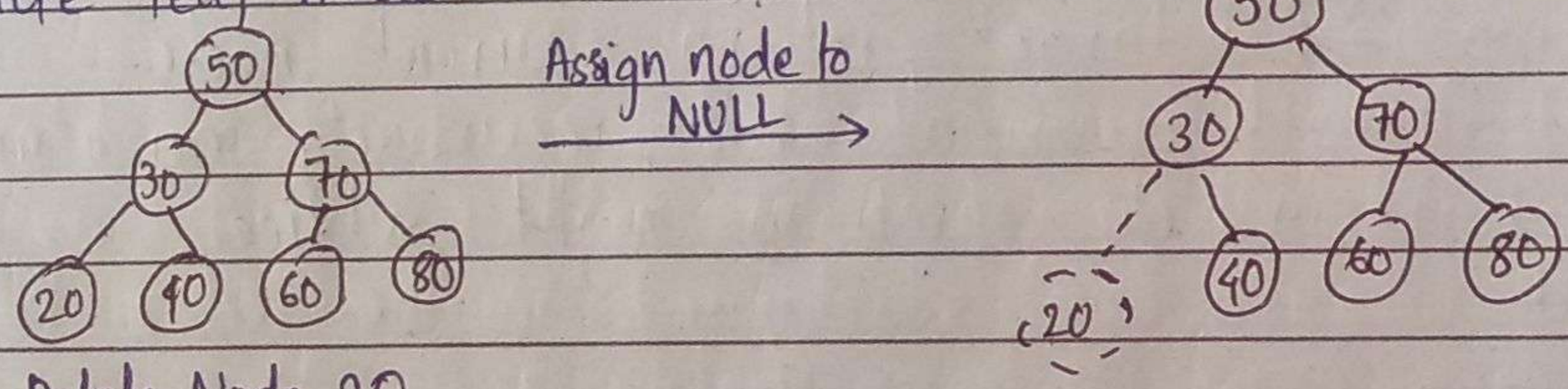
5.5.1 Insertion in BST

i) If root is null, create new node with given value and make it root of tree.
ii) Otherwise, recursively traverse tree to find appropriate position for new node based on following.
a) If value to be inserted < value of current node, go to left subtree.
b) Else, go to right subtree.
iii) Repeat (i) & (ii) until empty spot is found
iv) Insert new node at found location

Deletion in BST

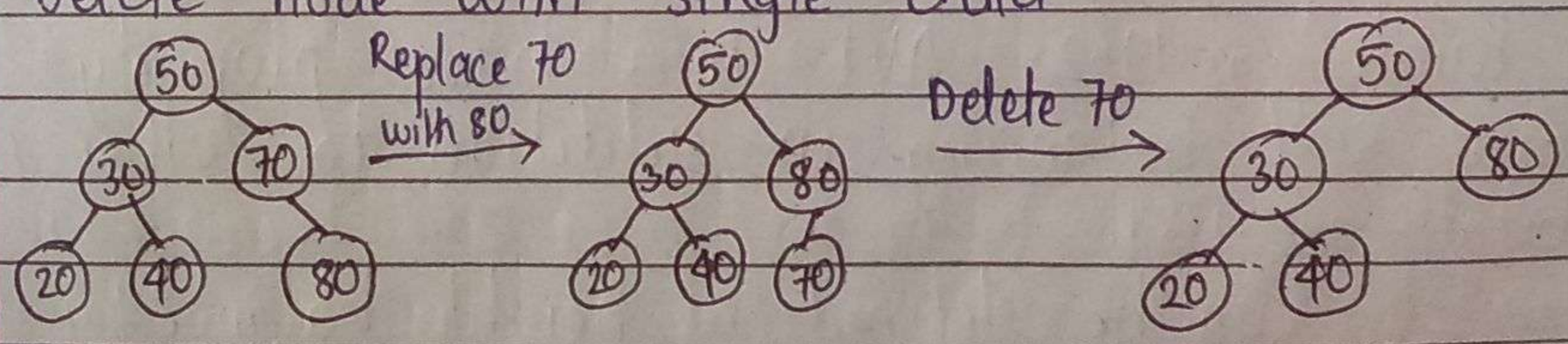
1) If root is null, tree is empty. Return null.
2) Otherwise, recursively search for node to be deleted in left or right subtree based on comparison of values.
   - → If value to be deleted < value of current node, go to left subtree.
   - → If value to be deleted > value of current node, go to right subtree.
   - → If values are equal, node to be deleted is found.
3) Once node to be deleted is found, consider:
   - → If node has no children, simply remove it.
   - → If node has one child, replace node with child.
   - → If node has 2 children, find node's in-order successor, replace node's value with successor's value and then recursively delete successor node.
4) After deletion, ensure BST properties are maintained.
5) Return modified root.

Case I: Delete leaf node in BST

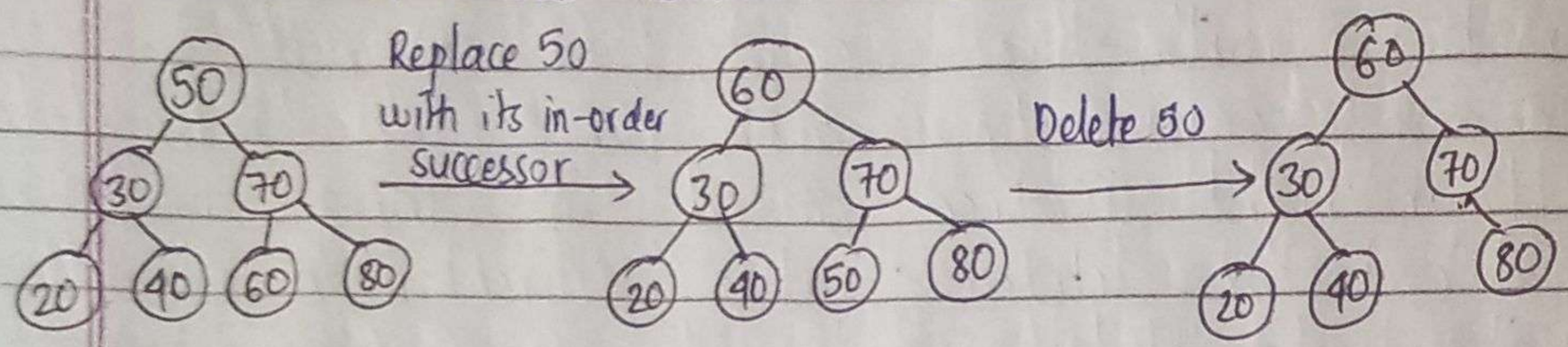


Delete Node 20

Case II: Delete node with single child



Case III: Delete a node with 2 children



Delete Node 50 After Deletion

5.6 Algorithm of Tree Search

Searching means to locate specific element /node in BST. It is easy because elements in BST are stored in specific order.

Search (root, item)

Step 1: Start

Step 2: Check if tree is empty or not

Step 3: Compare element to be searched with root node of tree

→ If ~~root~~ current node = target, return current node.

→ Else If target < current node, recursively search in left subtree

→ Else, recursively search in right subtree.

Step 4: End

5.9 AVL Balance Tree

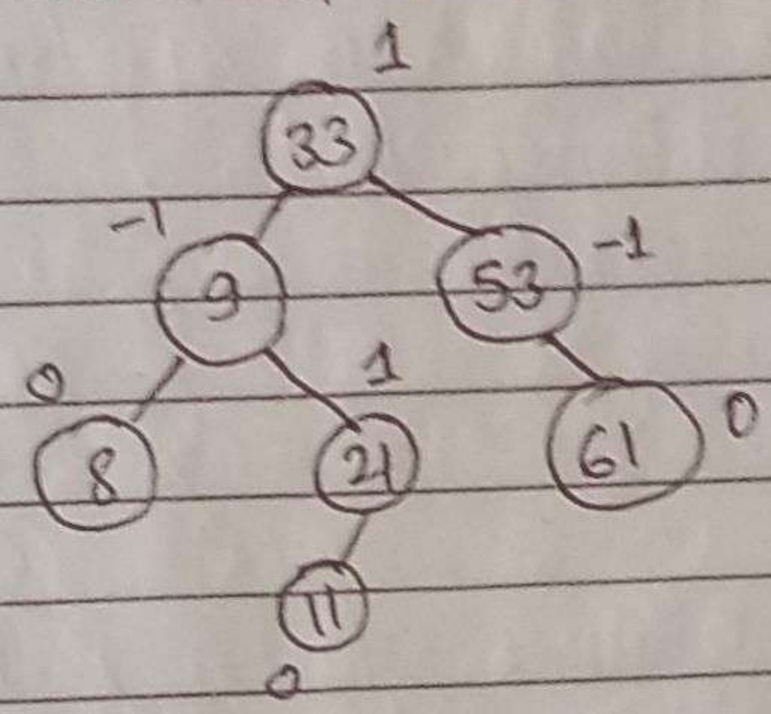
AVL Tree is named after its inventors Adeleson, Velskii and Landis in 1962. AVL Trees have property of dynamic self-balancing in addition to other properties exhibited by BST.

It can be defined as height balanced BST in which each node is associated with balance factor

calculated by difference of height of left subtree and right subtree.

Balance Factor = Height of (left subtree - right subtree)

AVL Tree is self balancing BST in which each node maintains extra information called balanced factor whose value is either -1, 0 or +1. If it isn't, tree will unbalanced and needs to be balanced.



Operations on AVL Tree

Searching and traversing don't lead to property violation in AVL. But insertion and deletion can. So, they need to be revisited.

In AVL Trees, after operation like insertion or deletion, balance factor of every node needs to be checked. If every node satisfies balance factor condition, operation can be concluded. Otherwise, tree needs to be rebalanced using rotation operations:

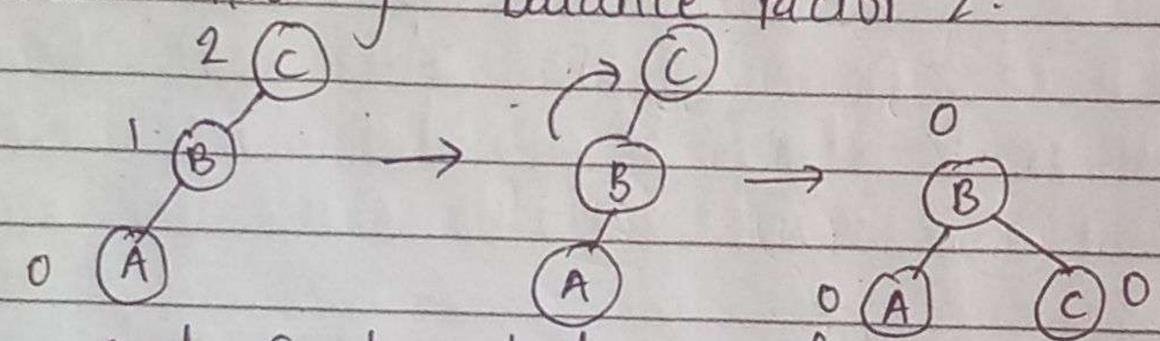
i) Imbalance in left child's left subtree, perform right rotation.
ii) " " right " right " , perform left rotation.
iii) " " right " left " , perform right-left "
iv) " " left " right " , perform left-right "

5.9.3 Single and Double rotation in balancing

First two rotations LL and RR are single rotations and next two rotations LR and RL are double rotations

i) LL Rotation

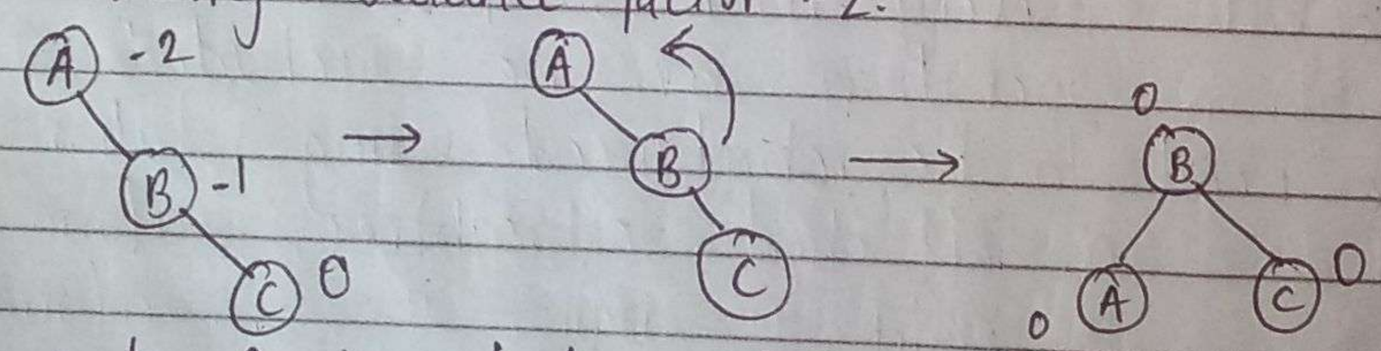
When BST becomes unbalanced due to insertion of node in left subtree of left subtree of C, we perform LL rotation (clockwise) which is applied on edge below a node having balance factor 2.



Here, node C has balance factor 2 because node A is inserted in left subtree of C left subtree. We perform LL operation on edge below C.

ii) RR Rotation

When BST becomes unbalanced due to insertion of node in right subtree of right subtree of A, we perform RR rotation (anti-clockwise) on the edge below a node having balance factor -2.



Here, node A has balance factor -2 because node C is inserted in right subtree of A's right subtree. We perform RR rotation on edge below A.

iii) LR rotation

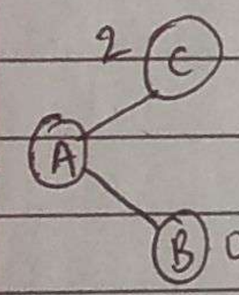
LR rotation

First RR rot

LL rotation

path of in

than 0, -1

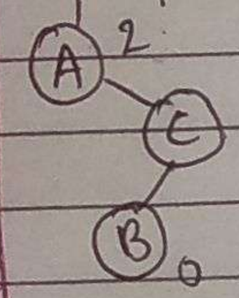


iv) RL rotation

RL rotation

First LL rot

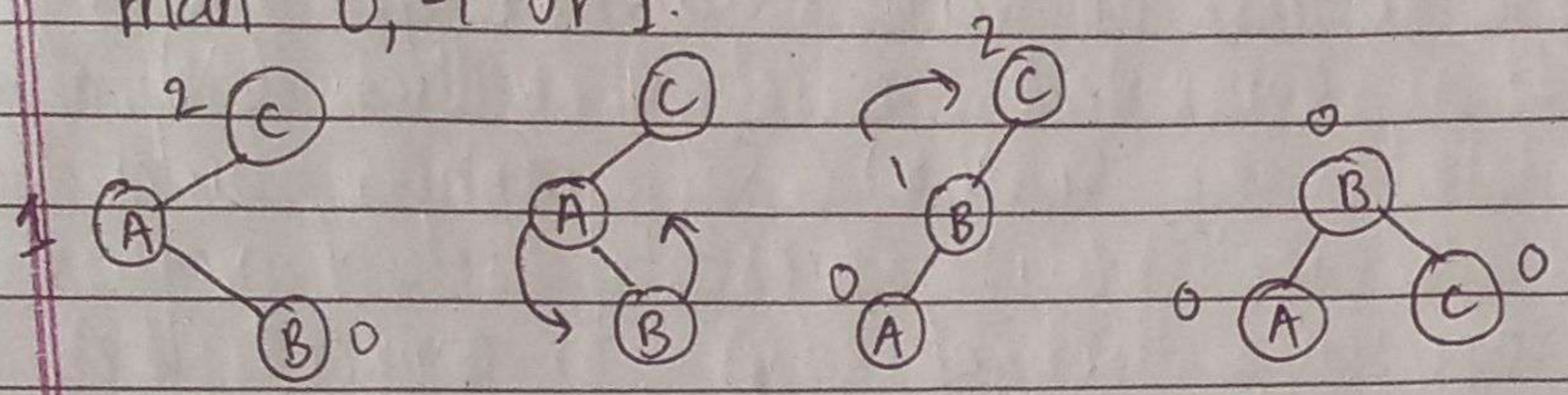
is performed



iii) LR rotation

LR rotation = RR rotation + LL rotation

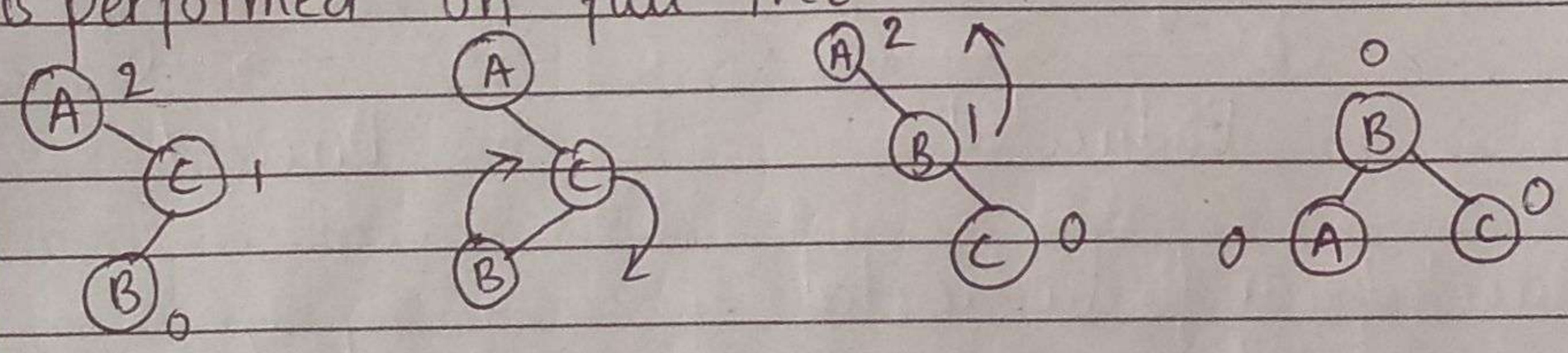
First RR rotation is performed on subtree and then LL rotation is performed on full tree (first node from path of inserted node whose balance factor is other than 0, -1 or 1.



iv) RL rotation

RL rotation = LL rotation + RR rotation

First LL rotation is performed on subtree and RR rotation is performed on full tree.



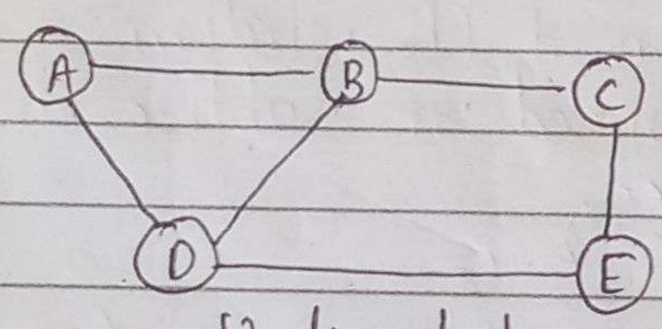
Unit - 8
Graph

8.1 Components of Graph

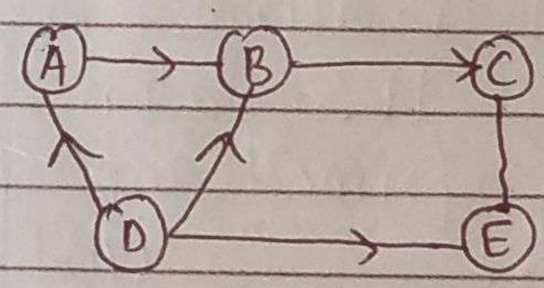
Graphs are non linear data structures made up of finite no. of nodes or vertices and edges that connect them. It is composed of set of vertices (V), edges (E) and denoted by G(V,E). So, graph is ordered set G(V,E) where V(G) represents set of vertices and E(G) represents set of edges to connect them.

→ Vertices are fundamental units of graph known as no

→ Edges are lines drawn to connect nodes of graph



Undirected



Directed

8.2 Directed and Undirected

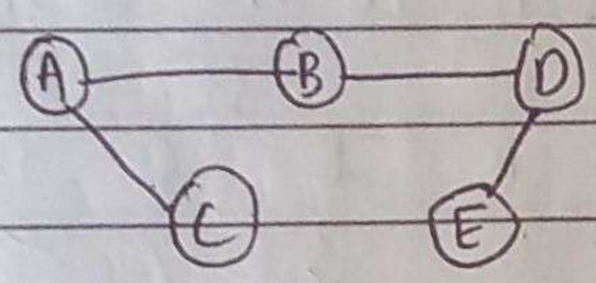
An undirected graph comprises a set of nodes and links connecting them. Order of 2 connected vertice is irrelevant and has no direction. It can be formed with finited no. of edges and vertices. The edges can be traversed bidirectionally.

A directed graph (digraph) is set of nodes connected by edges each with direction. In this, edges form an ordered pair. Edges represent a specific path from A to B for traversal where A is initial node and B is terminal node

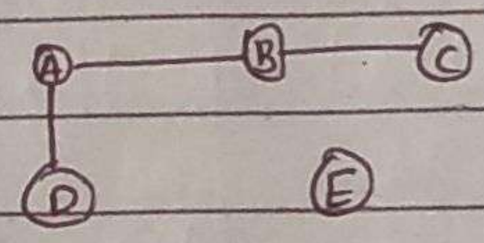
8.3 Connected And Unconnected

An unconnected graph is when there is no edge linking vertices. (eg: null graph). If there exists two vertices in G such that no path in G has these vertices as end points, it is unconnected graph. A graph with just one vertex is connected but edgeless graph with 2 or more vertices is disconnected.

A connected graph is one in which some path exists betn every two vertices (u, v) in V. There are no isolated nodes in this.



Connected

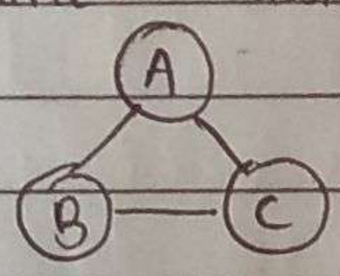


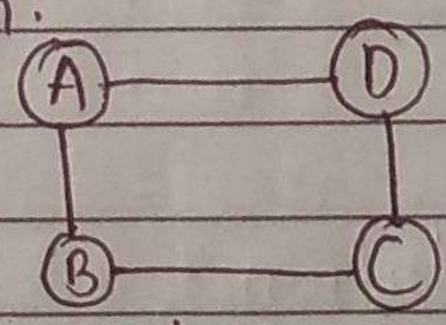
Disconnected

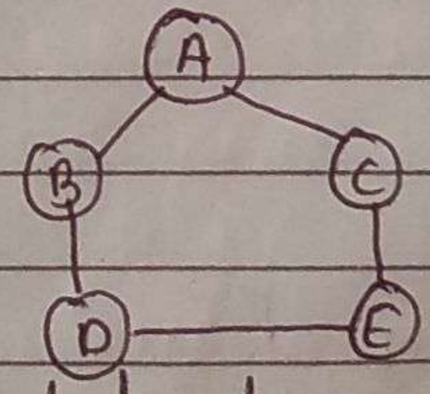
8.4 Path and Cycle

Path is sequence of nodes that are followed in order to reach some terminal node V from initial node U.

A path will be closed path if initial node is same as terminal node. A path will be closed if $V_{source} = V_{destination}$.







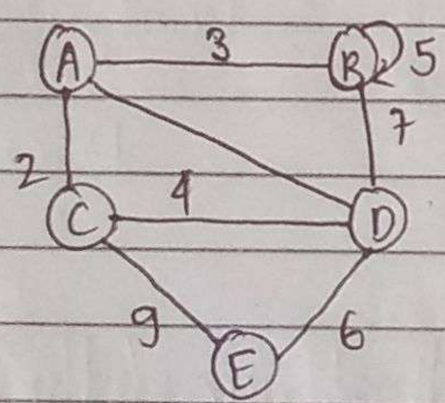
A cycle is path which has no repeated edges or vertices except first and last vertices. Cycle in graph is non empty trail in which only first and last vertices are equal.

Path: $A \rightarrow B \rightarrow C$ Cycle: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow A$

8.5 Adjacency sets and Tables.

If 2 nodes u and v are connected via an edge e, then nodes u and v are called neighbours or adjacent nodes. Adjacency table maps every vertices to set of its neighbours.

→ Adjacency matrix: It is sequential representation to show which nodes are next to one another (connection of nodes). You create (m x m) matrix G for representation. If an edge exists between node A and B, corresponding element of G will be G(i,j) = 1. Else, G(i,j) = 0. If it is weighted graph, you can record edge's weight instead of 1's and 0's.

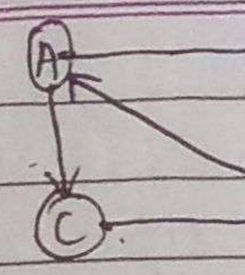


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 3 & 2 & 8 & 0 \\ B & 3 & 5 & 0 & 7 & 0 \\ C & 2 & 0 & 0 & 4 & 9 \\ D & 8 & 7 & 4 & 0 & 6 \\ E & 0 & 0 & 9 & 6 & 0 \end{array}$$

→ Adjacency table:

8.6 Array Based Representation

An adjacency list is array of list where each element (i) in array represent vertex (i) and corresponding list contains all vertices adjacent to vertex (i). Each vertex in graph has list of neighbouring vertices.

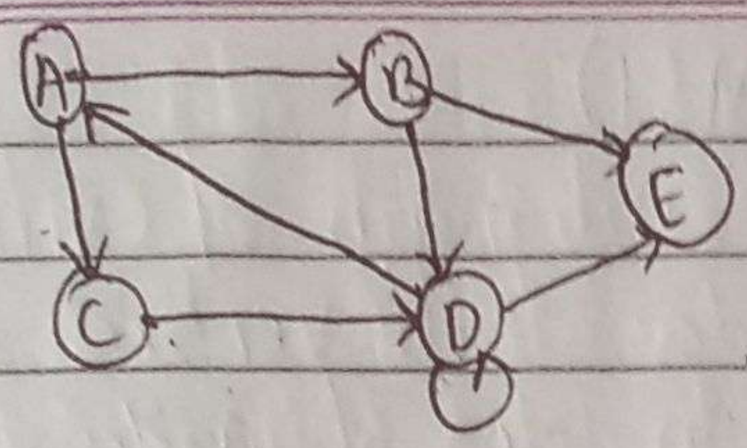


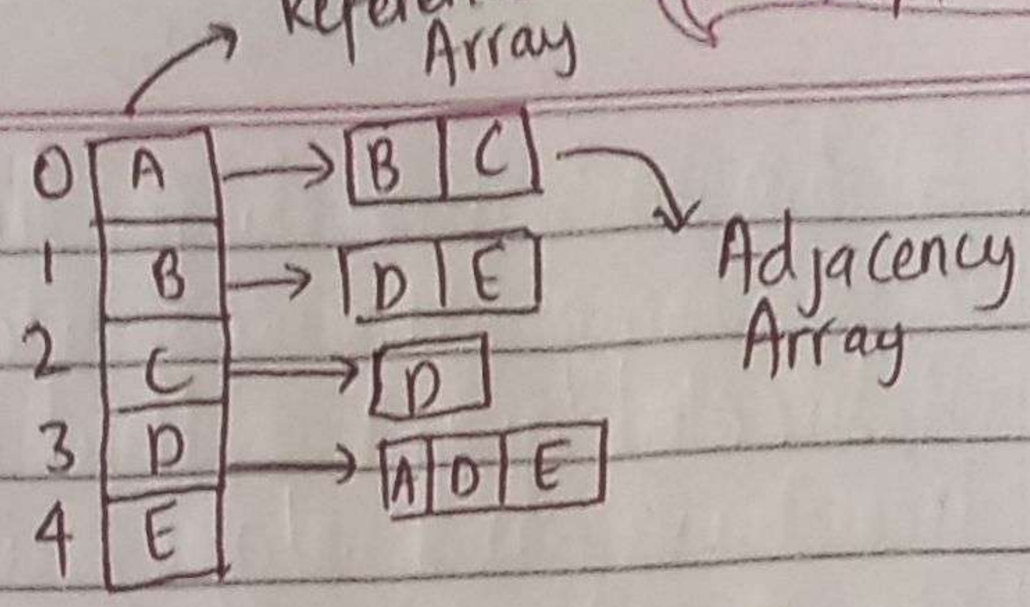
8.7 Linked Base

Adjacency Li
Each vertex
it. linked
adjacent t
for sparse-

Graph from

Mixed Rep
It combin
It uses
there are
sparse re
small d
combines



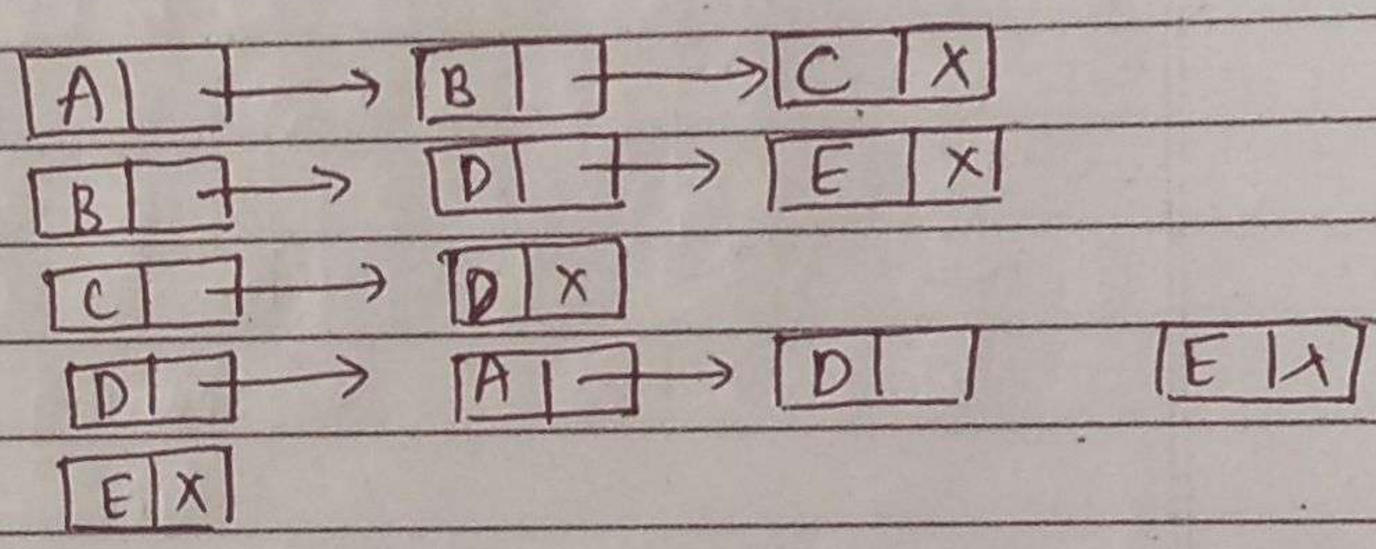


7 Linked Based and mixed implementation

Adjacency List with Linked Structures

Each vertex in graph has linked list associated with it. linked list contains nodes representing vertices adjacent to current vertex. It is memory efficient for sparse-graphs.

Graph from above :

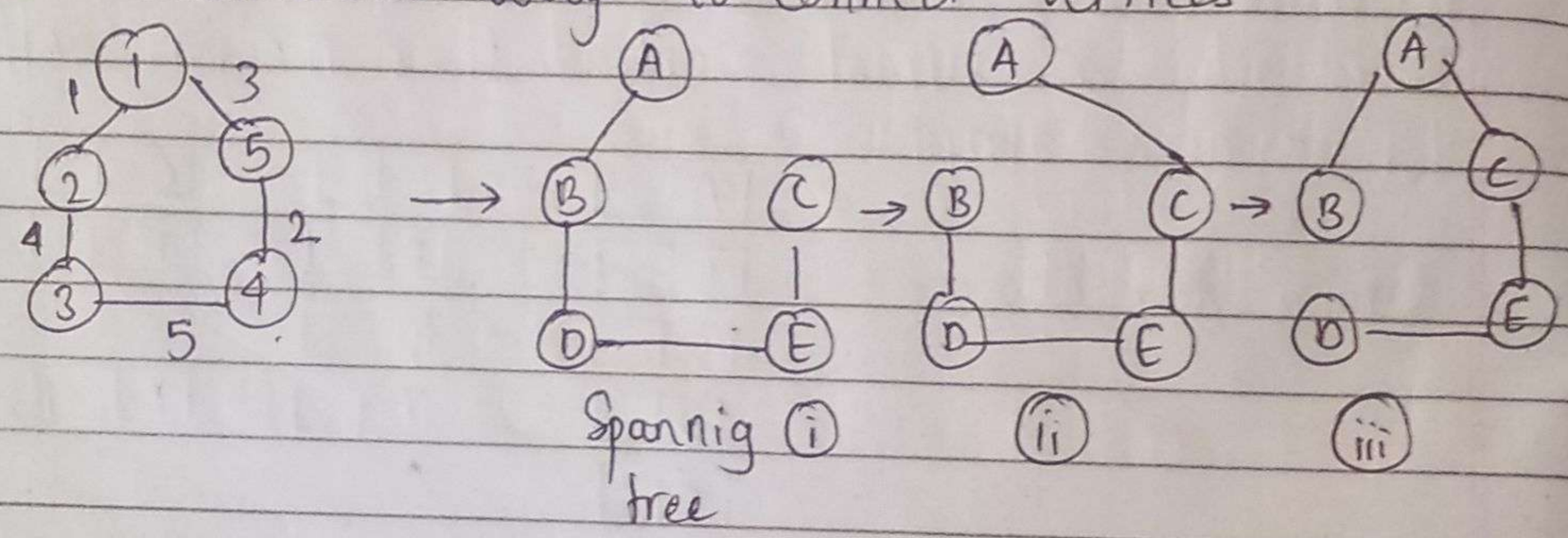


Mixed Representation (Adjacency Matrix + Adjacency list)

It combines an adjacency matrix with adjacency list. It uses matrix for dense regions (areas of graph where there are many edges betn vertices) and lists for sparse region (areas of graph with isolated vertices or small cluster of connected vertices). This approach combines strength of both representation.

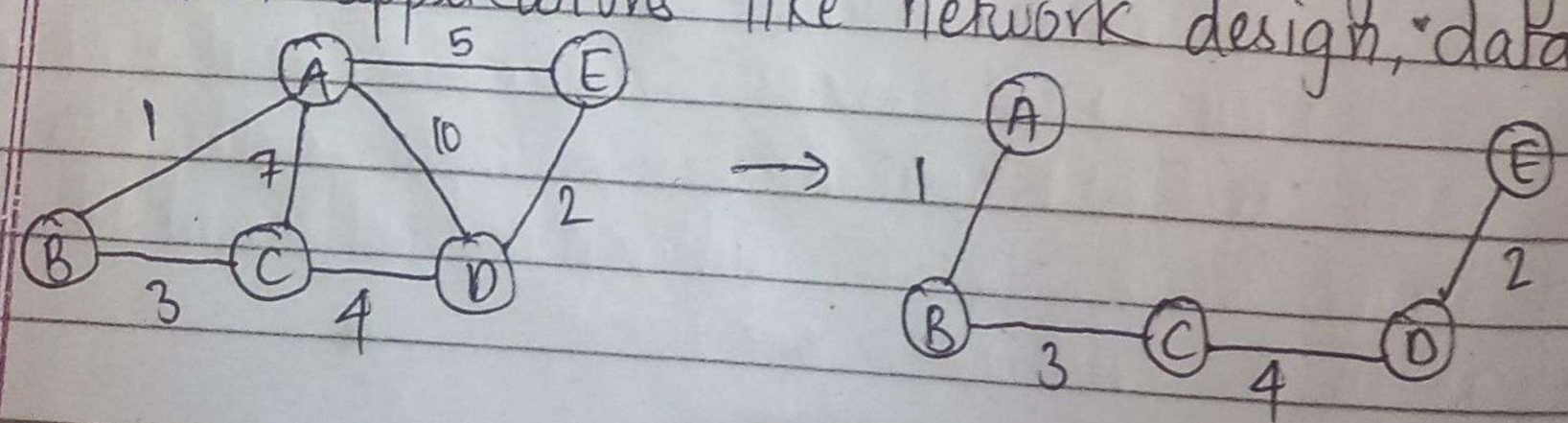
8.8 Minimum Spanning Trees (MST)

A spanning tree is tree-like subgraph of connected undirected graph that includes all vertices. A spanning tree has n-1 edges where n = no. of nodes. A minimum spanning tree is subset of edges of connected, edge-weighted graph that connects all vertices together without any cycles and with minimum possible total edge weight. It is way of finding paths in map i.e most economical way to connect vertices.



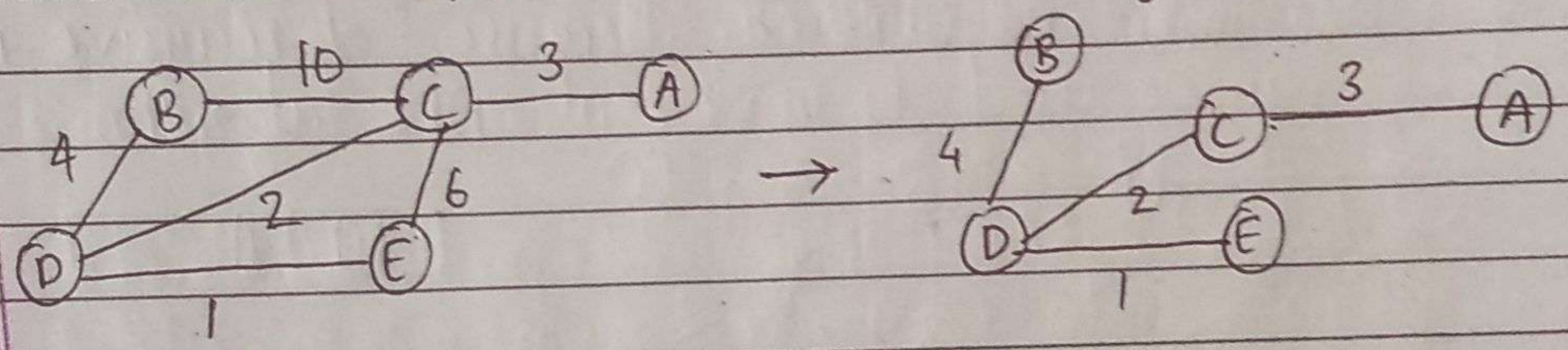
8.8.1 Kruskal's Algorithm and Prim's algorithm

Kruskal algorithm is popular algorithm for finding minimum spanning tree from connected, undirected graph. Its main target is to find subset of edges by using which we can traverse every vertex of graph. In this, we start from edge with lowest weight and keep adding edges until goal is reached. It is greedy algorithm used in applications like network design, data analysis



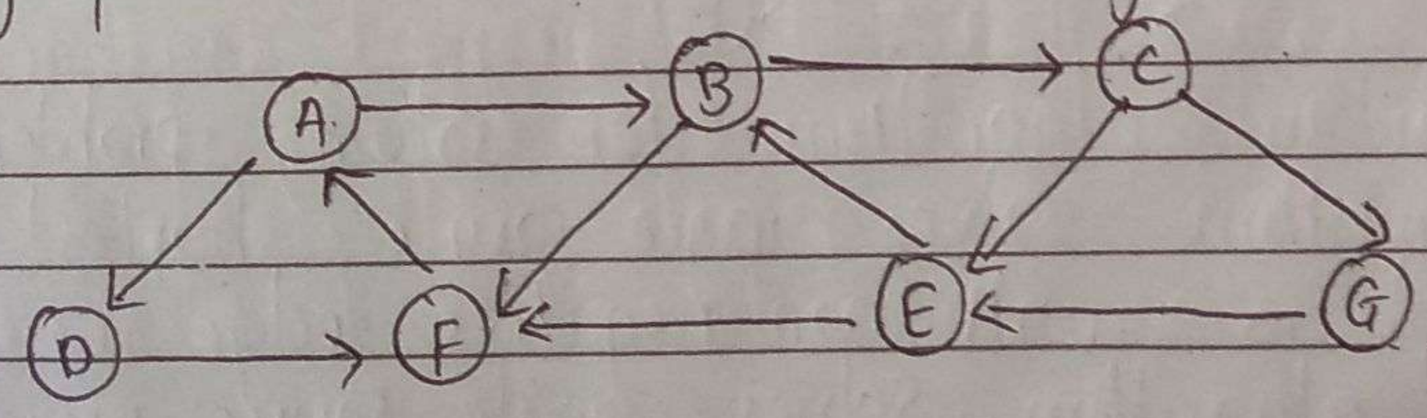
Prim's algorithm is greedy algorithm to find minimum spanning tree (MST) from graph. It finds subset of edges that includes every vertex of graph such that sum of weights of edges can be minimized.

It starts with single node and explores all adjacent nodes with all connecting edges at every step. Edges with minimal weights causing no cycles in graph are selected. It is used in image segmentation based on color/texture and routing for delivery trucks.



8.8.2 Algorithm of graph traversal (BFS, DFS)

Breadth first search (BFS) is graph traversal algorithm that starts traversing graph from root node and explores all neighbouring nodes. It is common approach for graph traversal and implemented using Queue. It puts every vertex of graph into 2 categories: visited and non-visited. It selects single node in graph and visits nodes adjacent to selected node.

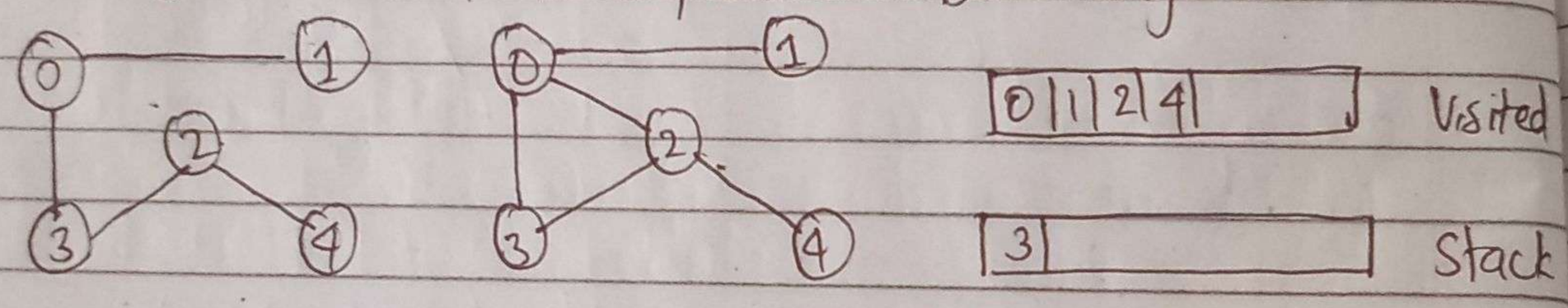


Adjacency List
A: B, D
B: C, F
C: E, G
G: E
E: B, F
F: A
D: F

Here, minimum path (P) can be found using BFS

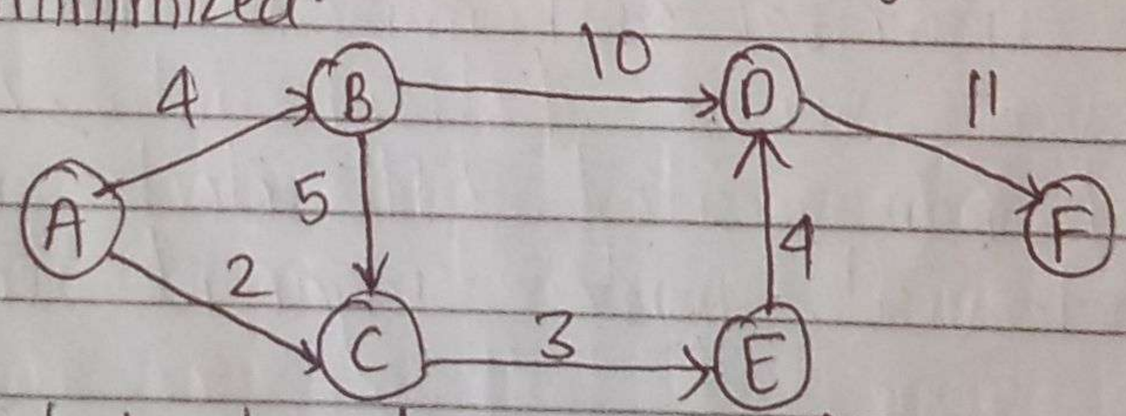
that starts from node A and ends at E. Algorithm uses two queues (Q1 and Q2). Q1 holds all nodes that are to be processed while Q2 holds all nodes that are processed and deleted from Q1. After BFS traversal, Q1 = {G} and Q2 = {A, B, D, C, F, E}

Depth First Search (DFS) is recursive algorithm to search all vertices of graph that starts with initial node of graph G and goes deeper until we find goal node or node with no children. Because of recursive nature stack can be used to implement DFS algorithm.



8.8.3 Shortest path algorithm

It is problem of finding path between two vertices in graph such that sum of weights of its constituent edges is minimized.



The most important algorithms for solving problem are:

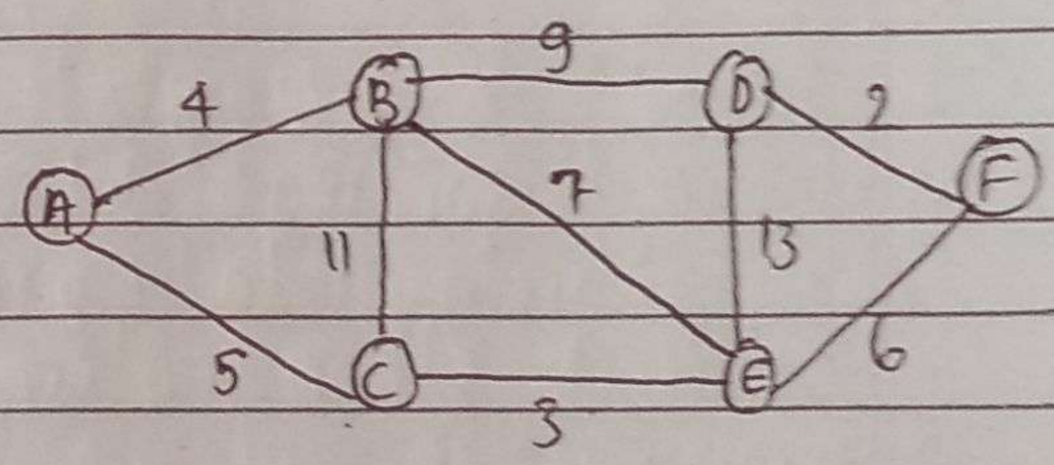
i) Dijkstra's algorithm : solves single source shortest path ~~algorithm~~ problem with non-negative edge weight

ii) Bellman-Ford algorithm : solves single source problem if edge weights may be negative

iii) A* search algorithm : solves for single pair shortest path using heuristics to try to speed up search
iv) Floyd-Warshall algorithm : solves all pairs shortest paths

Dijkstra's Algorithm

It is graph algorithm to find shortest path from source vertex to all other vertices in graph (single source shortest path). It is type of greedy algorithm that only works on weighted graph having positive weights.



Final paths:

1) A = 0
2) B = 4 (A → B)
3) C = 5 (A → C)
4) D = 4 + 9 = 13 (A → B → D)
5) E = 5 + 3 = 8 (A → C → E)
6) F = 5 + 3 + 6 = 14 (A → C → E → F)

Unit-6
Sorting

6.1 Definition

Sorting refers to ordering data in increasing or decreasing order according to some linear relationship among data items. It rearranges a given array of elements according to comparison operator on elements. Comparison operator decides new order of elements in respective data structure

6.2 Types of sorting

a) Internal sorting is sorting designed to sort data that fits entirely into computer's main memory (RAM). It is efficient when entire dataset can be loaded into memory.

b) External sorting is sorting done on data stored and processed in external storage devices (hard drives). It is done when dataset is too large for RAM.

6.3 Bubble Sort

It is sorting algorithm that repeatedly steps through list, compare adjacent elements and swaps them if they are in wrong order. It is simplest sort method which performs sorting by repeatedly moving largest element to highest index of array. It comprises of comparing each element to its adjacent element and replace them accordingly.

Algorithm

Suppose arr is array of n elements. The swap function swaps values of given array elements.

```
begin BubbleSort (arr)
    for all array elements
        if arr[i] > arr[i+1]
            swap (arr[i], arr[i+1])
        endif
    end for
    return arr
end BubbleSort
```

6.4 Insertion Sort

Insertion sort inserts each element of array to its proper place. It works similar to sorting of playing cards in hands. It is assumed that 1st card is already sorted in card game and we select an unsorted card. If selected unsorted card > 1st card, it'll be placed at right side. Else, it'll be placed at left side.

Algorithm

Step 1:

6.5 Selection sort

In this, smallest value among unsorted elements of array is selected in every pass and inserted to its appropriate position into array. It is in-place sorting algorithm. In this array is divided into 2 parts i.e sorted and unsorted parts. Initially, sorted part is empty and unsorted part is given array. Sorted part is placed left while unsorted is right.

In this, first smallest element is selected from unsorted array and placed at first position. After, second smallest element is selected and placed in second position. This process continues till array is sorted.

Algorithm

6.6 Quick Sort

It is widely used sorting algorithm that makes n log n comparisons in average case for sorting array of n elements. It is faster and efficient. It follows divide and conquer approach. Divide and conquer is technique of breaking down algorithms into subproblems, solving them and combining results back together to solve original problem.

Divide: First pick a pivot element. After that, partition array into 2 sub arrays such that each element in left sub-array is less than or equal to pivot element and each element in right sub-array is greater than pivot

Conquer: Recursively, sort 2 subarrays with Quicksort.

Combine: Combine already sorted array.

Algorithm

```
QuickSort (arr A, start, end)
{ if (start <end){
        p = partion (A, start, end);
        Quicksort (A, start, p-1);
        Quicksort (A, p+1, end);
    }
}
```

6.7 Merge Sort

It is similar to quick sort as it uses divide & conquer approach. It divides given list into two equal halves, calls itself for two halves and merges two sorted halves. Sub-lists are divided again & again into halves until list can't be divided further. Then we combine pair of one element list into two element list, sorting them in process and are merged until we get sorted list.

Algorithm

```
MergeSort (arr, beg, end) {
        if (beg < end) {
            int mid = (beg+end)/2
            MergeSort (arr, beg, mid)
            MergeSort (arr, mid+1, end)
            Merge (arr, beg, mid, end)
        End if
End MergeSort
```

6.8 Heap Sort

A heap is complete binary tree in which node can have utmost 2 children & all nodes should be left-justified. In heap sort, min heap or max heap is maintained from array elements depending upon choice and elements are sorted by deleting root element of heap. Min or max heap represents ordering of array in which root element represents min or max element of array.

First convert array into heap data structure

using heapify, then one by one delete root node of max-heap and replace it with last node in heap and then heapify root of heap. Repeat this until size of heap is greater than 1.

Algorithm

Unit-7
Search

Searching refers to process of finding desired element in set of items. Desired element is called target.

7.1 Sequential / Linear Search

It is search algorithm that starts at one end and goes through each element of list until target is found. Else, search continues till end of list.

Advantages
- can be used irrespective of whether array is sorted or not
- doesn't require extra memory
- well suited for small data

Drawbacks
- slow for large dataset i.e O(n) time complexity
- not suited for large data

Find 20

10	50	30	40	60	70	20	80	40
0	1	2	3	4	5	6	7	8

7.2 Binary Search

It is search algorithm for sorted array by repeatedly dividing search interval in half. Idea of binary search is to use information that array is sorted and reduce time complexity to O(log n).

Advantages
- faster than linear search
- well suited for large datasets

Drawbacks
- Array should be sorted
- Data structure need to stored in contigous memory location (eg: array

7.3 Tree Search Algorithm

In BST, searching element is easy because elements are stored in specific order. Steps to do so are:

i) Compare element to be searched with root node.
ii) If it matches, return node's location.
iii) If item is less than root, move to left subtree.
iv) Else, move to right subtree.
v) Repeat above procedure recursively until match is found
vi) If element is not found, return NULL.

7.4 Hashing

Hashing is popular technique in CS that involves mapping large data sets to fixed length values. It is process of converting data set of variable size into data set of fixed size. This algorithm converts an input (string, integer) into fixed sized output (hash code or value). Data is then stored and retrieved using this hash value as index in hash table.

Some Hashing Algorithms

i) MD5 (Message Digest 5): cryptographic hash function that produces 128 bit hash value
ii) SHA (Secure Hash algorithm): used in security protocols
iii) CRC: used in error detection in data transmission
iv) Argon 2: resist brute-force attacks

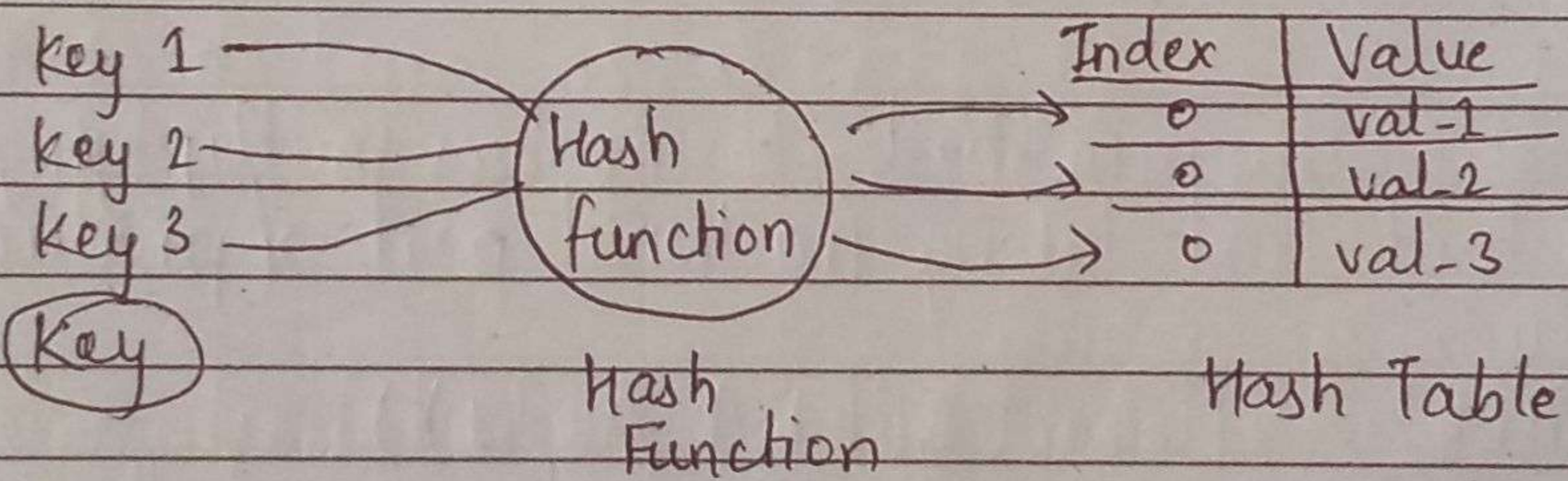
Advantages

i) Better synchronization than other structures
ii) Efficient than search trees
iii) Provides constant time for search, insert and delete

operations on average.

Disadvantages
i) Disallows null values
ii) Inefficient when there are many collisions
iii) Practically not avoided for large set of possible keys

Components of Hashing
1) Key: It can be string or integer fed as input in hash function, the technique that determines index for storage of an item in data structure.
2) Hash function: It receives input key and returns index of element in array called hash table. The index is called hash index
3) Hash table: It is data structure that maps keys to values using hash function. It stores data in associative manner in array where each data value has its own unique index.



7.4.2 Hash Function
It creates mapping between key and value through use of mathematical formula. Its result is hash value or hash. For eg: Consider an array as Map

where key is index and value is value at that index. So for array A, if we have index (i) which will be treated as key, we can find value by looking at A[i].

Types of Hash Functions

i) Division Method
ii) Mid Square Method
iii) Folding Method
iv) Multiplication Method

i) Division Method: It involves dividing key by table size and taking remainder as hash value. It is simplest and easiest method to generate hashvalue.

$h(k) = k \bmod M$

Here, 'k' is key value and M is size of hash table.

ii) Mid-square method: It involves two step to compute hash value as square value of key and extract middle 'r' digits as hash value

$h(k) = h(k \times k)$

iii) Multiplication method: It involves multiplying key by constant and taking fractional part of product as hash value.

$h(k) = \text{floor}(M(kA \bmod 1))$

M = size of table, A = constant value k = key value

iv) Digit Folding Method: It involves two steps as first divi key-value k into no. of parts i.e k1, k2, --- kn where each part has same no. of digits except for

last where that can have lesser digits than others and add individual parts. Hash value is obtained by ignoring last carry if any

$k = k1, k2, \ldots, kn$

$s = k1 + k2 + \ldots + kn$

$h(k) = s$ $\quad$ s = sum of parts of key (k)

Collision in Hashing

Hashing process generates small no for big key. So there is possibility that two keys could produce same value. The situation where newly inserted key maps to an already occupied and it must be handled using collision handling technology.

Two methods to handle it are:

1) Separate Chaining (Open hashing)
2) Open Addressing (Closed Hashing

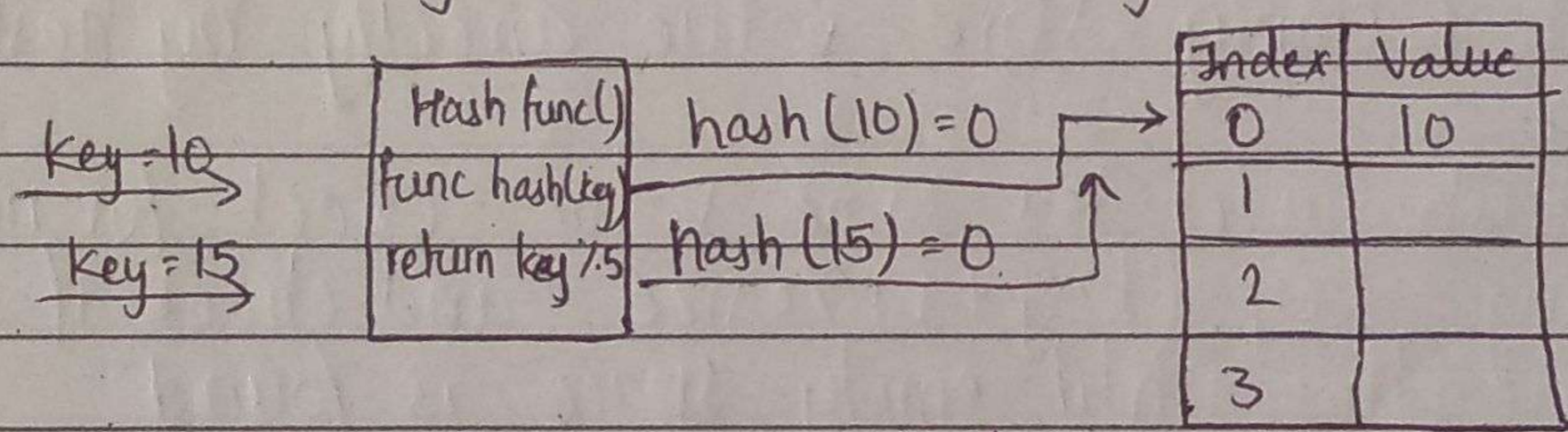


Fig: Hashing collision

Open and Closed Hashing

The idea of open hashing is to make each cell of hash table point to linked list of records that have same hash function value. Chaining is simple but requires extra memory outside table.

Slot	
0	15 → 25
1	
2	12 → 22
3	

In closed hashing, all elements are stored in hash table itself. Each table entry contains either record or NULL. When searching for element, we examine table slot one by one until target is found or missing.

Linear Probing

In this, hash table is searched sequentially that starts from original location of hash. If location we get is already occupied, we check for next location.

→ Algorithm

i) Find hash key: key = data % size

ii) Check if empty. Store value directly by hashTable[key] = data

iii) If hash index has some value, check for next index using key = (key + 1) % size and check if empty.

iv) Repeat this until empty space is found

Quadratic Probing

It operates by taking original hash index and adding successive values of arbitrary quadratic polynomial until open slot is found

$$h(k, i) = [h1(k) + i^2] \% n$$

where n = hash table size k = key to be hashed

i = non-ve no. that indicates collision no.

Double Hashing

It make use of 2 hash functions. The first hash function is h1(k) which takes key and gives out location on hash table. But if new location is occupied, we'll use secondary hash function h2(k) in combination with first hash function h1(k) to find new location on hash table.

$$h(k,i) = (h1(k) + i \times h2(k)) \% n$$

where, k = key n = hash table size
i = non negative no that indicates collision no