Data Structure and Algorithm

EG2202CT

Fourth Semester

Prepared by Santoshi kumari Pandit

Unit - 1

Date 20/01/2076
Page

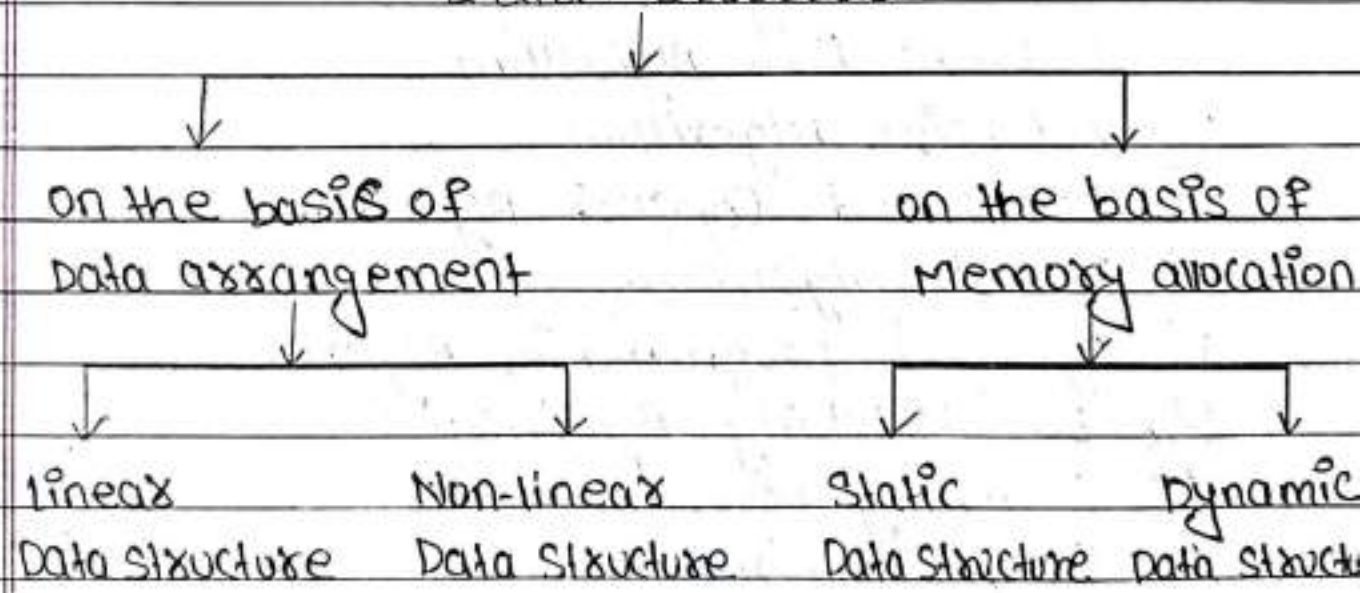
* Data Structure

Data structure is a way of organizing data items by considering it's relationship to each other or, data structure is a way of storing data in a computer so that it can be used efficiently.

* Da Algorithm

An algorithm is a well defined list of steps for solving a particular problem.

* Types of Data structure.



1.1 * Algorithm

Algorith is a set of commands that must be followed for a computer to perform calculations or other problem solving operations. it is a finite set of instructions carried out in a specific order to perform a particular task. It is not the entired program or code. it is simple logic to a problem represented as informal description in the form of statements or pseudo code.

* Types of Algorithm.

1.) Brute force Algorithm
2.) Recursive Algorithm
3.) Divide and Conquer Algorithm
4.) Greedy Algorithm
5.) Dynamic programming Algorithm
6.) Backtracking Algorithm
7.) Searching Algorithm
8.) Sorting Algorithm.

1.) Brute Force Algorithm

→ Brute force Algorithm are simple and straight ~~State~~ forward solutions ~~straight~~ that use trial and error to solve problems. it blindly iterates all possible solutions to search one or more than one solution

that make Solve a function.

2.) Recursive Algorithm.

→ A method that breaks a problem into smaller, soft problems and repeatedly applies itself to solve them until reaching a base case is recursive algorithm.

3.) Divide and Conquer Algorithm

→ This algorithm breaks a problem into sub problems, solves a single sub problem and merge the solutions to get the final solutions. it consist of the following three steps divide, solve, integrate.

4.) Greedy Algorithm.

→ it makes locally optimum choices at each step in the hope of finding a global optimum useful for optimization problems but may not always lead to the best solution.

5.) Dynamic programming Algorithm

The algorithm work by remembering the results of the past run and using them to find new results. In other words, it solves complex problems by breaking them into multiple simple sub problems and then it solves each of them once and then stores

them for future use. Fibonacci series is a good example for Dynamic programming Algorithm.

6.) Backtracking Algorithm

→ Whenever a solutions fails we track back to the failure points build on the next solutions and continue this process till we find the solutions.

7.) Searching algorithm.

Searching algorithm are the once that are used for searching elements or group of elements from a particular data structure.

8.) Sorting Algorithm

Sorting Algorithm are the once that are used for arranging a group of data in a particular manner (e.g ~~ascent~~ accending, decending) according to the requirement.

06.19

1.2 * Data structure

Data structure is a way of organizing, ~~all data items~~ Managing and storing all data items that considers their relationship to eachother. Data structure mainly specifies the following four things:

1.) Organization of data

2.) Accessing methods.

3.) Degree of Associativity
4.) processing alternatives for information.

Examples of different data structures are: Array, stack, Queue, Linked list, Tree, Graph etc.

* Operations on Datastructure.

Following are the datastructure operations

(i.) Create :- Create operation reserved memory space for the program elements.

(ii.) Inserting :- Adding new record into the data structure.

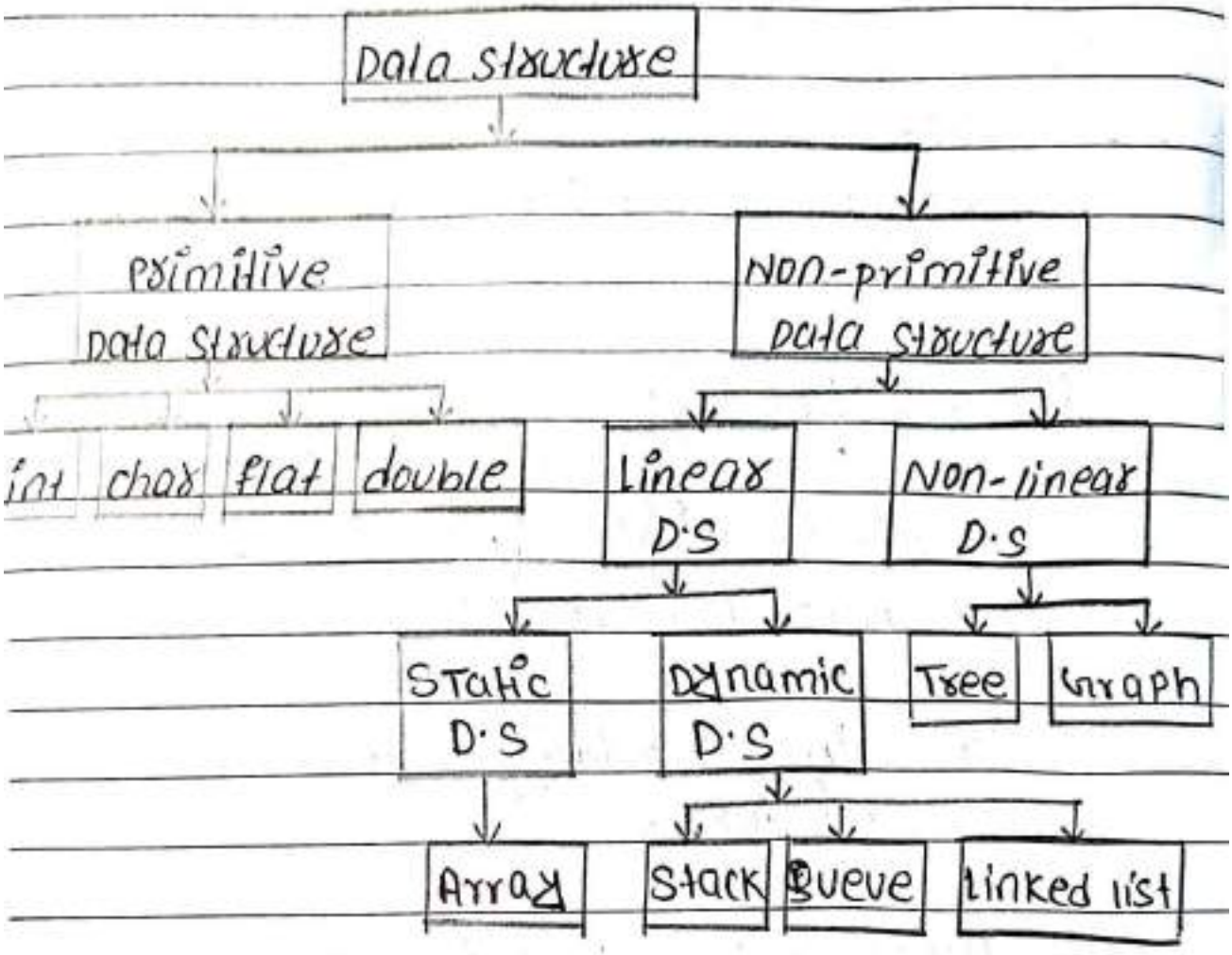
(iii.) Deleting :- Deleting record from the data structure

(iv.) Sorting :- Arranging the records in some logical

(v.) Searching :- finding data items and its location in data structure.

(vi.) Traversing :- Accessing each record exactly once.

Types of data structure.



primitive Data Structure

A primitive Data structure is pre-define by the programming language. The size and type of variable values are specified and it has no additional method. Examples are integer, character, float, Double, pointer etc.

Non-primitive Datastructure.

These data types are not actuall define by the programming language but are created by the programmer. Examples are: Array, stack, queue, Linked list, Tree, Graph etc.

Date
Page

Non-primitive data structure are further devided into two categories.

(i) Linear Datastructure
(ii) Non-Linear Datastructure

(i) Linear Datastructure

Linear datastructure consist of data elements arranged in a sequential manner where every element is connected to its previous and next elements. This connection helps to traverse a linear arrangement in a single level and in a single run. Example are: Array, stack, queue, linked list etc.

(ii) Non-Linear Data structure

In Non-linear datastructure, data are stored without focusing on it's sequence and the data cannot be arranged and accessed in a sequence. Examples are: Tree, Graph etc.

* Static Datastructure

Static datastructure are datastructures where the size is allocated at the compile time. So the maximum size is fixed and cannot be changed. Examples are: Array.

* Dynamic Datastructure

Dynamic datastructures are Data-Strcture where the size is allocated at the run time. hence the maximum size is flexible and can be changed as per requirements. Examples are: Stack, Queue, Linked list etc.

06-21

* Unit-2 *

Stack

2.1 * Stack

Stack is a non-primitive linear Datastructure. it is an ordered list in which addition of new data item or deletion of an already existing data item is done at one end. Called top of the stack. The insertion operation is called push operation and the deletion operation is called pop-operation.

Since all insertion and deletion takeplace at the same end, The last element added to the stack will be the first element removed/accessed from the stack. Therefore stack is called LIFO (Last in first out) datastructure.

Date
Page

* push operation

The processing of puting a new data item onto stack is known as push operation which involves a series of steps:

(i.) checks if the stack is full.

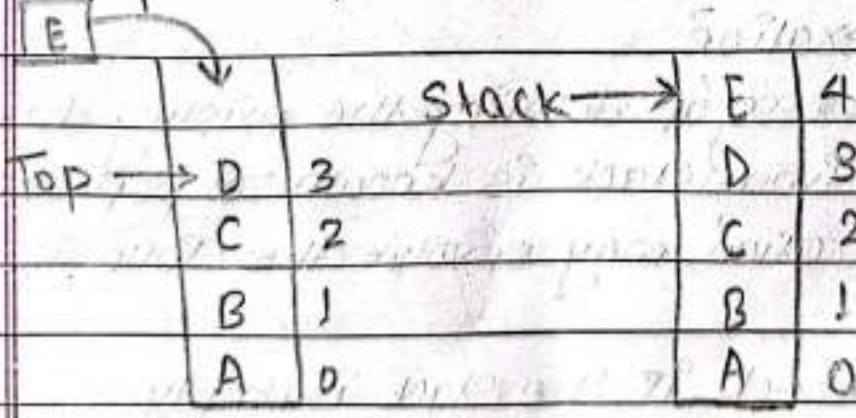
(ii.) if the stack is full, we cannot insert data into the stack.

(iii.) if stack is not full, increments ~~top~~ Top to point Next empty space.

(iv.) At data item to the stack location at where Top is pointing.

(v.) Returns Success.

The below figure illustrate the concept of push operation.



Push operation

* Algorithm for push operation.

Description:-

let ~~Stack~~ STACK is an array. let MAX is the maximum of the array STACK and

ITEM is the element to be inserted.
Top denotes the position of the Top element in the stack and initially we set Top = -1 to denote stack is empty.

Steps :-

(i.) If [Top is equal to MAX -1] THEN Display "Stack is overflow".
(ii.) ELSE
(a.) Set Top = Top + 1
(b.) Set STACK[TOP] = ITEM
(c.) Display "ITEM is inserted"
(iii.) END IF
(iv.) EXIT.

6-22

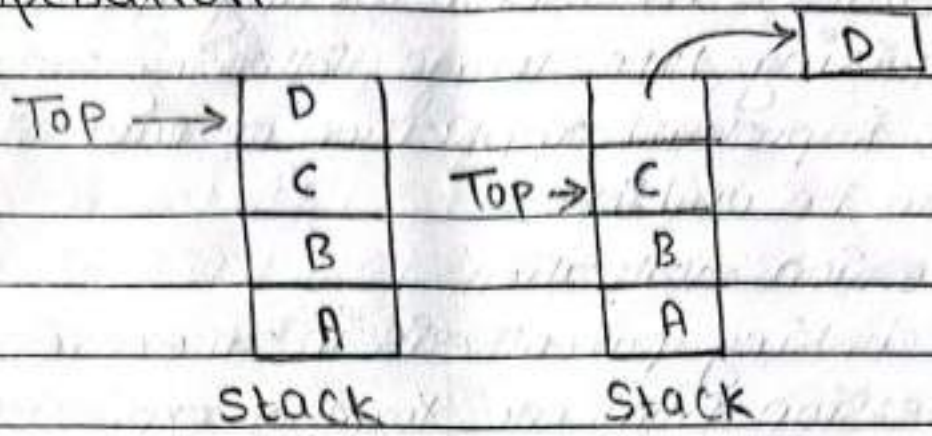
* pop-operation.

The process of deleting the existing data element from stack is known as pop operation. A pop operation may involve the following steps.

Step 1 :- check if the stack is empty.
Step 2 :- if the stack is empty, Display stack empty message and exit.
Step 3 :- if stack is not empty, delete the data element at which Top is pointing.
Step 4 :- Decrease the value of Top by 1.
Step 5 :- Returns success.

Date
Page

The below figure imustrate the concept of pop opexation.



* Algorith fox pop opexation.

Description:-

Let STK is an Array having size MAX, Top is used to points the Top most element of the array and DEL is the value to be Deleted

Steps:-

(i) if [Top $\overset{=-1}{<0}$] THEN Display "STK is empty |underfl

(ii) ELSE

(a.) Set DEL = STK[TOP]

(b) Set Top = TOP - 1

(c.) Display DEL is deleted.

(iii.) END IF

(iv) EXIT.

Date
Page

2.2 * Application of STACK.

STACKs are useful for any application requiring LIFO storage following are some of the important application of data structure. It can be used for:

(i.) Expression evaluation
(ii.) Sha:cheking parenthesis in an expression.
(iii.) Conversion from one form of expression to another
(iv.) Memory management.
(v.) Management of undo and redo operation.
(vi.) Function call management.
(vii.) Traversing trees and graphs.
(viii.) Recursive call management.

* Expression.

Expression is define as the number of operands combined with sevel operators.

There are basically three types of Notation (way of writing expression) for an expression. These are: ~~In-fix~~.

(i.) In-fix Notation → A+B, x*y.
(ii.) pre-fix Notation → +AB, *xy.
(iii.) post-fix Notation → AB+, xy*.

Date 06-2-23
Page

Expression Notation conversion

priority of operators Table

operations	symbols	priority
paranthesis	()	first
Exponentiation	$, ↑	second
Multiplication, Division	*, /	third
Addition, substration	+, −	fourth

2.2.1 * Infix to prefix conversion Rules.

Rules :-

(i) The expression is scanned from left to right untill an operator of higher priority is found.

(ii) once an operator of higher priority is found, the operator is shifted to it's left.

(iii) Scanning is done again from left to right and all the operators are shifted in this manner.

Examples :-

(i) a + b * c (Infix)

a + *bc

+a * bc (prefix)

(ii) (a+b) * c (infix)

+ab * c

* +abc (Prefix)

(iii) a+b$c*d (Infix)

a + $bc * d

~~+$*bcd+a~~ a+ * $bcd

~~$*+abcd~~ +a * $bcd (prefix)

Date
Page

2.2.1 Rules :- (Infix to postfix convesion).

(i) The expression is scanned from left to right untill an operator of higher priority is found.

(ii) Once an operator of highex priority is found, the operator is shifted to it's right.

(iii) Scanning is done again from left to right and all the operators are shifted in this manner.

(i) a+b*C (Infix)
a+ bC*
abC*+ (Postfix)

(ii) (a+b) * C (Infix)
ab+ *C
ab+C* (postfix)

(iii) a+b$C *d (Infix)
a+bC$ *d
a + bC$d*
bC$d*a+ (postfix)
abC$d+

(iv) Convert into prefix notation.
(A% + (B*C)
(A% + *BC)
+ A% * BC prefix

Date
Page

(2.) ((Ax + (B * Cy)) / (DE))

((Ax + * BCy) / (DE))

(+ Ax * BCy) / (DE)

/ + Ax * BCy DE prefix

Assigment

Infix to prefix and postfix.

(i.) ((A+B) * (C+E))

(+ AB * + CE)

* + AB + CE (prefix)

(i) ((A+B) * (C+E))

(AB + * CE +)

AB + CE + * (postfix)

(ii.) (Ax * (Bx * (((Cy + Ay) + By) * Cx)))

(Ax * (Bx * ((+ CyAy + By) * Cx)))

(Ax * (Bx * (+ CyAy + By * Cx)))

(Ax * (Bx * * + CyAy + By Cx))

(Ax * **yAy + By Cx * Bx * (yAy + ByCx)

* Ax * Bx * CyAy + ByCx (prefix)

(ii.) (Ax * (Bx * (((Cy + Ay) + By) * Cx)))

(Ax * (Bx * ((+ CyAy + By) * Cx)))

(Ax * (Bx * (+ + CyAy By * Cx)))

(Ax * (Bx * * + + CyAy By Cx))

(Ax * * Bx * + + CyAy By Cx)

* Ax Bx * + + CyAy By Cx

* * * + + Ax Ay Bx By Cx Cy

* Ax * Bx * + + CyAy ByCx

Date
Page

(iii.) A+B↑C*D/(E-F)

A+B↑C*D/-EF

A+↑BC*D/-EF

A+*↑BCD/-EF

A+/*↑BCD-EF

+A/*↑BCD-EF prefix

(iii.) A+B↑C*D/(E-F)

A+B↑C*D/EF-

A+BC↑*D/EF-

A+BC↑D*/EF-

A+BC↑D*EF-/

ABC↑D*EF-/+ POSTFIX POSTFIX

(ii.) (AX*(BX*(((CY+AY)+BY)*CX)))

(AX*(BX*((CXAY++BY)*CX)))

(AX*(BX*(CXAY+BY+*CX)))

(AX*(BX*CXAY+BY+CX*))

(AX*(BXCXAY+BY+CX*))

(AY*BXCXAY+BY+CX**)

AYBXCXAY+BY+CX***POSTFIX.

Date 06-24
Page

2075

A*(B+C$D)-E$F*(G/H)

In postfix:-

A*(BCD$+)-E$F*(G/H)

A*BCD$+-E$F*(G/H)

A*BCD$+-E$F*GH/

A*BCD$+-EF$*GH/

A*BCD$+*-EF$*GH/

ABCD$+*-EF$GH/*

ABCD$+*EF$GH/*-

* WAP for stack operations.

```
#include<stdio.h>
#include<conio.h>
  void push();
  void pop();
  void display();
#define max 10
 int stk[max],i,choice,top=-1;
  void main()
{
home:
   printf("\n1.push");
   printf("\n2.pop");
   printf("\n3.display");
   printf("\n4.Exit");
   printf("Enter your choice:");
    scanf("%d",&choice);
```

Date
Page

```
switch (choice)
{
case 1 : push(); getch(); clrscr(); goto home;
case 2 : pop(); getch(); clrscr(); goto home;
case 3 : Display(); getch(); clrscr(); goto home;
case 4 : Exit(0);
default :
        printf("\n Invalid choice");
        getch(); clrscr(); gotohome;
    }
}
```

⇒

```
void push()
{
int n;
 if (top == max-1)
   {
    printf("\n stack is full");
   }
  else
   {
    printf("\n Enter no to be inserted");
    scanf("%d", &n);
    top++;
    stk[top]=n;
    printf("\n %d is inserted ", n);
   }
```

⇒
```
void pop()
{
if (top == -1)
{
printf (" the stack is empty");
}
else
{
printf ("%d is deleted ", STK[TOP]);
    top = top-1;
}
}
```

⇒
```
void pop()
{
int del;
if (top<0)
{
printf("\n stack is empty");
}
else
{
del = STK[top];
top = -;
printf("\n %d is deleted ", del);
}
}
```

4.

⇒

```
void display()
{
int p=top;
if (top==-1)
 {
printf("\n Stack is empty");
 }
else
 {
  for(; p>=0; p--)
  {
  printf("\n %d", Stk[p]);
  }
 }
}
```

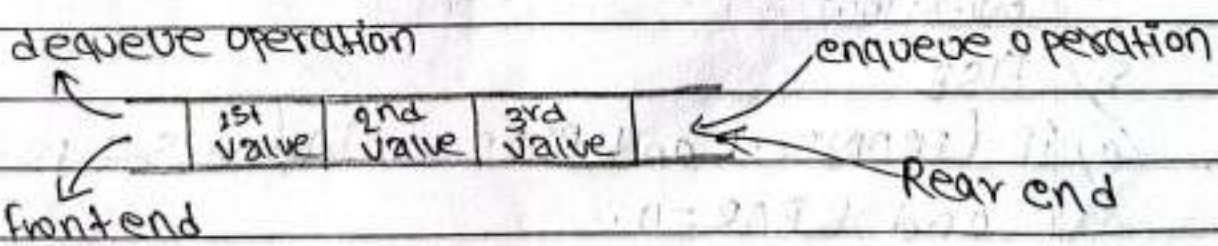
Unit 2

Date 06-25
Page

2. Queue

A queue is a non-primitive linear data structure. It is open at both of its ends. It is an ordered list in which one end is always use to insert data called rear and the other end is always used to remove data. Enqueue datastructure the insertion operation is called Enqueue operation and deletion operation is called dequeue.

Since the insertion of new data item is done only through the dedicated end (rear) and deletion of an existing data item is done through the other dedicated end (front), the first element added to the queue will be the first element to remove from the queue. Therefore, queue is also called FIFO (First in First out) ordered data structure:



2.3.1 Enqueue operation

Enqueue operations envolves the following series of steps.

1.) checks whether the queue is full or not.
2.) If the queue is full then display "queue is full".

Date
Page

3) If the queue is ~~not~~ Empty, then increments Rear to point ~~makes~~ next empty space.
4) Insert data element to the queue location where Rear is pointing.
5) Returns Success.

06-29

2.3.2 Algorith for Enqueue operation.

5

* Description

Let ~~Queue (QUEUE)~~ QUE is an array, MAX is the maximum size of the array QUE, ITEM is an element to be inserted, FRONT and REAR denotes the front and Rear in of the queue. An initiany it's value is set to -1 to denote Queue is empty.

Steps:-

1) IF [REAR=MAX-1] THEN Display "QUE is full / overflow".
2) ELSE
a) if (FRONT=-1 and REAR=-1) then Set FRONT=0 and REAR=0.
b) else set REAR=REAR+1.
c) end If.
d) QUE(REAR)=ITEM.
e) Display ITEM is inserted.
3) END IF.

```
void enqueue()
 {
if(r==max-1)
 {
printf("queue is full")
 }
 else
 {
if(f==-1 && r==-1)
 {
 f=0;
 r=0;
 }
else
 {
 r++;
 }
printf("Enter a no");
scanf("%d", &item);
 QUE(r)=item;
 }
}
```

Date 06-30
Page

2.3.2 Algorith for Dequeue operation.

* Description

Let QUEUE is an array, MAX is it's maximum size and DEL is an element to be deleted. FRONT and REAR denotes the front and Read end of the queue.

Steps :-

1.) If [FRONT = -1 and REAR = -1] THEN Display "Queue is underflow / Empty".
2.) ELSE
(a.) Set DEL = QUEUE [FRONT]
(b.) If [FRONT = REAR] then
(a.) Set FRONT = -1
(b.) Set REAR = -1
(c.) else
Set FRONT = FRONT + 1
(d.) end if
(e.) Display : DEL is deleted
(3.) END IF
4.) EXIT

Date
Page

2.3.6 * Applications of QUEUE

(1) QUEUES are widely used as waiting list for a single shared resource like ~~pat~~ printer, disk, CPU.

(2) When data is transferred asynchronously between two processes, que is used for synchronization for example IO Buffers, file input output (I/O) etc.

(3) QUEUES are used in operating systems for handling interrupts in real time system.

(4) Call center phone systems use queues to hold people calling them in order.

* Limitation of QUEUE:

As we know after ENQUEUEing and DEQUEUEing operations, the size of the queue has been reduced. If REAR reaches the last Index and FRONT is not at Index zero, we cannot insert element in the QUEUE even there is free spaces in the QUEUE. This is the limitation of (linear) QUEUE.

Enqueue
Display
Exit

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
int que[10], item, r=-1, f=-1, choice;
void main()
{
home:
        printf("\n 1. Enqueue");
        printf("\n 2. Display");
        printf("\n 3. Exit");
        printf("\n Enter your choice:");
        scanf("%d", &choice);
        switch(choice)
{
        case 1: enq(); getch(); clrscr(); goto home;
        case 2: display(); getch(); clrscr(); goto home;
        case 3: exit(0);
default: printf("\n wrong choice"); getch(); clrscr();
                goto home;
    }
}
void enq()
{
if (r==9)
{
```

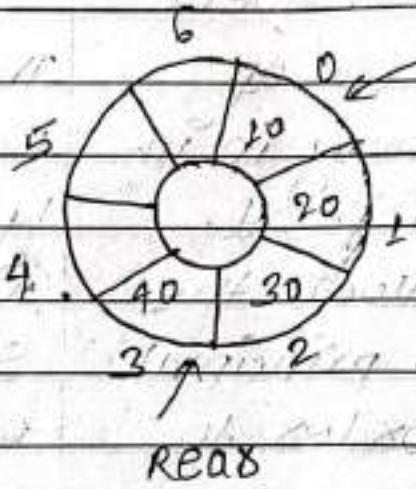
```
printf("\n Queue is full / overflow");
 }
else
 {
if(f==-1 && r==-1)
 {
f=0, r=0;
 }
else
 {
r++;
 }
printf("\n Enter a number");
scanf("%d", &item);
que[r] =item;
printf("\n %d is inserted", item);
}
}
void display()
 {
int t=f;
if(f==-1 && r==-1)
{
printf("\n Queue is empty");
}
else
{
for(; t<=r; t++)
```

```
}
printf("\n %d ", queue[i]);
    }
  }
}
```

Date 2080-07-12
Page

Circular Queue

Circular Queue is just a variation of the linear Queue in which front and rear end are connected to each others to optimize of the space wastage of the linear Queue and make it efficient. The below figure illustrate the concept of Circular Queue.



Difference between linear Queue and Circular Que

Basis	Linear	Circular
1.) Meaning	The linear Queue is a type of linear data structure that contains the elements in a sequential manner.	The Circular Queue is also linear data structure in which the last element of the Queue is connected to the first element of the Queue
2.) Insertion and deletion	In linear Queue insertion is done from the rear end	In Circular Queue, the insertion and deletion can take

Date
Page

		and deletion is done from the front end	~~In circular~~ from any end.
3.)	Memory Space	The memory space occupied by the linear queue is more than the circular queue	It requires less memory as compared to linear queue.
4.)	Memory Utilization	The use of Memory in in-efficient.	The memory can be more efficient ly utilized.
5)	order of execution	It follows the FIFO principal in order to perform the task.	It has no specific order for execution.

Advantage of circular queue over linear queue.

i.) Linear queue consumes more memory as compared to circular queue.

ii.) A circular queue uses an efficient way for memory utilization.

iii.) In a circular queue, new data can be inserted again at a particular position after deleting previous data on that ~~provision~~ position.

iv.) In a circular queue, an overflow message is shown when the queue is full.

Types of queue

Mainly there are three types of queue. These are

(i) Linear queue
(ii) Circular queue
(iii) priority queue

Enqueue
Dequeue
Display
Exit

```
#includ <stdio.h>
#include<conio.h>
#include<process.h>
 void enqueue();
 void dequeue();
void display();
#define max 10
int enque[10], item, choice, r=-1, f=1, del;
 void main()
  {
home:
    printf("\n1. Enqueue");
    printf("\n2. Dequeue");
    printf("\n3. Display");
    printf("\n4. Exit");
```

```
printf("Enter your choice:-");
scanf("%d",&choice);
switch(choice)
{
case 1: enque(); getch() clrscr(); goto home;
case 2: deque(); getch(); clrscr(); goto home;
case 3: display(); getch(); clrscr(); goto home;
case 4: Exit(0);
default: printf("\n wrong choice") getch(); clrscr(); goto home;
}
}
void enqueue()
{
if (r==9)
{
printf("\n queue is full");
}
else
{
if(r==-1 && f==-1)
{
f=0; r=0;
}
else
{
r++;
}
printf("\n Enter a number");
```

```
scanf("%d", &item);
queve[r] = item;
printf("\n %d is inserted", item);
 }
}
void dequeve()
  {
if (f==-1 && r==-1)
 {
printf("\n queve is empty");
  }
else
del = queve[f];
 if (f==r)
{
f=-1, r=-1)
 }
else
 {
f++;
}
printf("\n %d is deleted", del);
  }
}
void display()
  {
int t=f;
if (f==-1 && r==-1)
```

Date
Page

```
{
printf("\n queue is empty");
}
else
{
for(; t<=r; t++)
{
printf("\n %d", queue[t]);
}
}
}
```

Unit-3

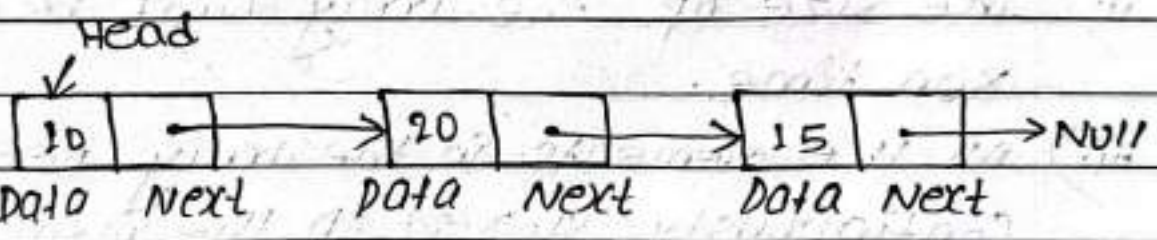
List / Linked List

Date 2080-08-10
Page

1. * List / Linked List (singly linked list)

A linked list is a linear datastructure that includes a series of connected nodes. Each element in a linked list datastructure is stored in the form of a node. Here each node has two parts. One is used to store the data and the other is used to store the address of the next node.

The elements in a linked lists are linked using pointers as shown in below figure.



* Structure of linked list :-

Each node Consist :-

(i) A data item.

(ii) An address of next node

We combine both the dataitem and the next node address in a struct as :

```
struct node
{
    int data;
    struct node *next;
};
```

* Application of Linked list.
(i) Dynamic Memory Allocation (DMA)
(ii) Implemented in stack and queue for creating Dynamic envisoment.
(iii) A undo functionality of softwares.
(iv) Hashtables and ~~graphs~~ graphs implementation.

08-11

* Advantage of linked list over Array / Limitation of Array.
i.) The size of array must be known inadvance before using it in the program.
ii.) The size of the array can't be change at run time.
iii.) All the elements in the array need to be continuously stored in the memory.
iv.) Inserting and deleting any element in the array needs shifting of elements.

Linked list datastructure overcome all the limitation of an array Some of the advantage of linked list over array are:

1.) Sizing is no longer a problem since we don't need to define its size at the time of declaration.
ii.) It allocates the memory dynamically so the size of linked list can be changed when ever require.

iii) All the nodes of linked list are non-continuously stored in the memory and linked together with the help of pointers.

iv) Insertion and deletion of nodes is easy just by adjusting the pointer parts of few nodes.

* operation on linked list:

1.) Node and list creation.
2.) Insert elements to a linked list.
a.) Insert at the begining of list.
b.) Insert at the end of list.
c.) Insert at the middle of list.
3.) Delete elements from linked list.
a.) Delete the begining element.
b.) Delete the end element.
c.) Delete the middle element.

1.) Node creation

```
struct node
{
int data;
struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *) Malloc (sizeof(struct node));
```

Date
Page

12

Q.) WAP to create a single node, and display its contents.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
int data;
struct node * next;
} * head, * ptr;
void main()
{
int item;
printf("Enter a Number");
scanf("%d", &item);
ptr = (struct node *) Malloc(sizeof(struct node));
ptr -> data = item;
ptr -> Next = NULL;
clrscr();
printf("Elements of Node is : %d", ptr->data);
getch();
}
```

Q.)2) WAP to create two nodes and link each node with eachother.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
Struct node
{
int data;
Struct node * next;
};
Struct node *ptr1,*ptr2;
void main()
{
int item1, item2;
printf (" Enter two number");
Scanf ("%d %d ", &item1,&item2);
ptr1 = (struct node *) malloc ( sizeof (struct node));
ptr2 = (struct node* ) malloc (sizeof (struct node));
ptr1 -> data = item1;
ptr2 -> data = item2;
ptr1 -> next = ptr2;
ptr2 -> next = NULL;
printf ("\n Elements of list are :\n");
printf ("%d", ptr1 -> data);
printf ("%d", ptr2 -> data);
printf ("%d", ptr -> next -> data);
getch();
}
```

WAP to create 5 nodes and link these nodes :

Q.No.4 WAP to create n nodes and link these with each other.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct mynode
{
int d;
struct mynode *link;
} *start=NULL, *ptr, *temp;
void main()
{
int n, i, item;
printf("Enter total No. of nodes");
scanf("%d", &n);
for(i=1; i<=n; i++)
{
ptr=(struct mynode *)malloc(sizeof(struct mynode));
printf("Enter an element");
scanf("%d", &item);
ptr->d=item;
ptr->link=NULL;
if (start==NULL)
{
start=ptr;
}
else
{
temp=start;
```

```
while(temp->link != NULL)
{
temp = temp->link;
}
temp->link = ptr;
}
}
temp = start;
while (temp != NULL)
{
printf("%d", temp->d");
temp = temp->link;
}
getch();
}
```

Q.No5 WAP to create n nodes and links these which each other. (Addition of n node).

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct mynode
{
int d;
struct mynode *link;
} *start = NULL, *ptr, *temp;
void main()
{
```

```
int n,i,item,sum=0;
printf("Enter total no of nodes");
scanf("%d",&n);
for(i=1; i<=n; i++)
{
ptr=(struct mynode *)malloc(sizeof(struct myno
printf("Enter an element");
scanf("%d",&item);
ptr->d=item;
ptr->link=NULL;
if(start==NULL)
{
start=ptr;
}
else
{
temp=start;
}
while(temp->link!=NULL)
{
temp=temp->link;
}
temp->link=ptr;
}
temp=start;
while(temp!=NULL)
```

Date
Page

(12)

```
Sum = Sum + temp -> data;
printf("%d", temp->data);
temp = temp -> link;
}
getch();
}
```

18-17

Insertion of Node in Linked List.

(1.) Algorith to insert node at the Beginning of list

Description.

Suppose START is the first position in the linked list, DATA is the element to be inserted in the new node and next NEXT is the pointer to the next node in the list.

Steps → i. Create a newNode.

ii. Input ~~DATA~~ VAL to be inserted.

iii. NewNode → DATA = VAL

iv. if [START is equal to NULL] THEN
    Set START = NewNode

v. ELSE
    (a.) set NewNode → NEXT = START
    (b.) Set START = NewNode

vi. END IF

vii. Display "VAL is inserted"

viii. EXIT.

* Function for inserting a node at the Begining of list.

=> Void insertatBeg ()

```
{
    int val;
    Struct node * newnode;
    newnode =(struct node *)malloc(sizeof (struct node
    printf ("\n Enter a No.");
    scanf("%d", &val);
    newnode -> data =val;
    if ( start == NULL)
     {
       Start= Newnode;
     }
     else
     {
      newnode ->next=start;
       Start = newnode;
      }
     printf("%d is inserted ",val);
}
```

Q.) 1.)

1. create
2. Display
3. insertatBegining
4. Exit

Date
Page

(15)

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
Struct node
{
    int data;
    struct node * next;
}*start=NULL, *ptr, *newnode, *temp;
void create();
void display();
void insertatBeg();
void main()
{
  int choice;
  home:printf ("\n 1.create");
       printf ("\n 2. Display");
       printf ("\n 3. insertatBeg");
       printf ("\n 4. Exit");
       printf("\n Enter your choice\n");
       scanf ("%d", &choice);
Switch(choice)
{
  Case1 : create(); getch(); clrscr(); goto home;
  case2 : Display(); getch(); clrscr(); goto home;
  case 3 : insertatBeg(); getch(); clrscr(); goto home;
  case 4 : exit(0);
  default : printf ("\n Invalid"); getch(); clrscr(); goto home;
}
}
```

```
void create()
{
  int n, i, val;
  printf("\n Enter total no of node");
  scanf("%d", &n);
  for (i=1; i<=n; i++)
  {
    ptr=(start node*)malloc(sizeof(struct node));
    printf("\n Enter a No");
    scanf("%d", &val);
    ptr->data=val;
    ptr->next=NULL;
    if (start==NULL)
    {
      Start=ptr;
    }
    else
    {
      temp=start;
      while(temp->next!=NULL)
      {
        temp=temp->Next;
      }
      temp->next=ptr;
    }
  }
```

Date 2080-08-28
Page

(2.) Algorithm to insert node at the end of list.

* Description

Suppose START is the first position in linked list, VAL is the element to be inserted in the new node, NEXT is the pointer to the next node in the list and TEMP is the temporary pointer to hold the node address.

Steps→(i.) Create a NewNode.

(ii.) Input VAL to be inserted.

(iii.) a) Set NewNode → DATA = VAL

b) Set NewNode → NEXT = NULL

(iv.) IF [START is equal to NULL] THEN

(a.) START = NewNode

(v.) ELSE

(a) TEMP = START

(b) WHILE [TEMP → NEXT != NULL]

(a.) TEMP = TEMP → NEXT

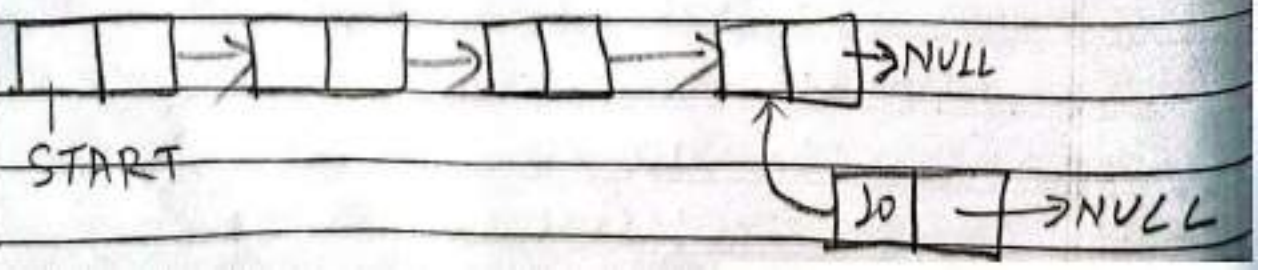
(c) End of while

(d) NewNode → NEXT = TEMP

(vi.) END IF

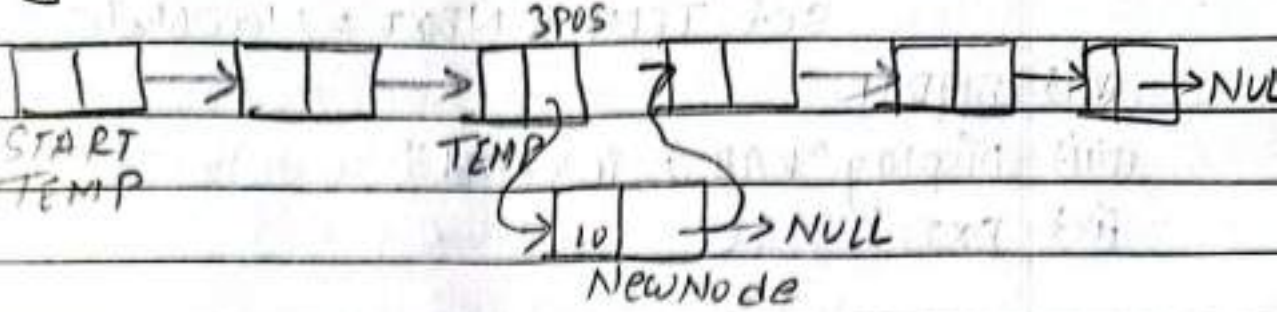
(vii.) Display "VAL is inserted".

(viii) EXIT.



Date
Page

(3.) Algorithm to insert node at the middle of list.



Description

Suppose START is the first position in linked list, VAL is the element to be inserted in the new node, NEXT is the pointer to the next node in the list and TEMP is the temporary pointer to hold the node address. pos is the position where newnode is inserted.

(i.) Create a NewNode.
(ii.) Input VAL to be inserted
(iii.) Input pos where the NewNode is to be inserted.
(iv.) a) Set NewNode -> DATA = VAL
b) Set NewNode -> NEXT = NULL
(v.) if [START is equal to NULL] THEN
(a.) START = NewNode
(vi.) ELSE
(a.) TEMP = START
(b) COUNTER = 1
(c) WHILE [COUNTER <= POS]
{
TEMP = TEMP -> NEXT
COUNTER is increased by 1.
}

(d.) Set NewNode → NEXT = TEMP → NEXT
Set TEMP → NEXT = NewNode

(vi.) ENDIF

(viii.) Display "VAL is inserted"

(ix.) EXIT

(1)	TEMP = START
(2)	START = TEMP → NEXT
(3)	free(TEMP)

for deletion at beginning

(4) Algorithm to delete first Node.

Description

Let 9 and START are the pointer variables that hold the address of the first node.

Steps - (i.) if [START is equal to NULL] THEN
Display "List is not created"

(ii.) ELSE
(a.) set 9 = START
(b.) Set START = START → NEXT
(c.) Display "9 → DATA is deleted"
(d.) free(9)

(iii.) END IF

(iv.) EXIT

(5.) Algorithm to delete last Node.

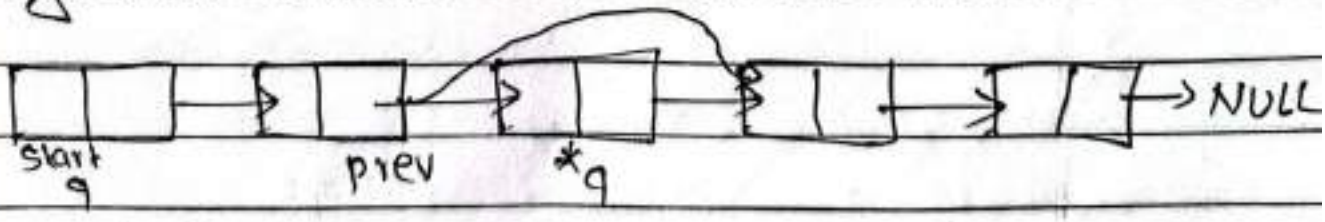
Description

Let q and START are the pointer variable that holds the address of first Node in the list and PREV is the temporary pointer variable.

eps-(i) IF [START is equal to NULL] THEN
Display "List is not created"
(ii) ELSE
(a.) q = START
(b.) WHILE [q = NEXT not equal TO NULL]
PREV = q
q = q -> NEXT
(c.) WEND (END of while loop)
(d.) PREV -> NEXT = NULL
(e.) Display q -> DATA is deleted
(f.) Free the node(q)
(iii.) END IF
(iv) EXIT

080/08/24

(6.) Algorithm to delete the middle node.



Description

Let q and START are the pointer variable that holds the address of the first node in the list, PREV is the temporary pointer variabl and pos in the position of the node to be deleted.

Step- (i) If [START is equal to NULL] THEN
Display "List Is Empty".

(2) ELSE

(a) Set q = START and Counter = 1.

(b) WHILE [counter < pos.]

(i) Set PREV = q

(ii) q = q → NEXT

(iii) Counter = Counter + 1

(c) WEND (End of WHILE loop).

(d) PREV → NEXT = q → NEXT

(e) fxee the node (q)

(3) END If

* Before the node
(counter < pos)

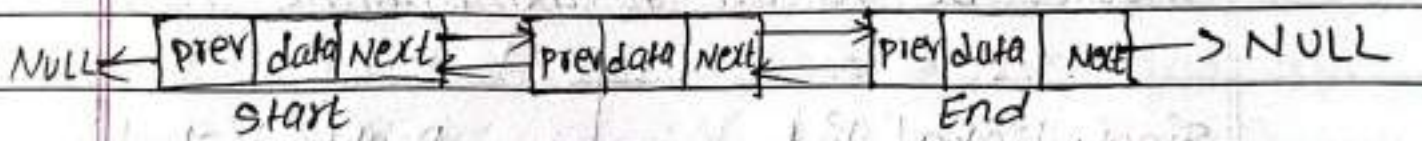
* After the node
(counter <= pos).

Date 2080/08/24
Page

2. * Doubly linked list

Doubly linked list is a group of pointers and nodes such that each node is pointing at it's preceding as well as it's succeeding node in the list.

In Dubly linked list, each node consist of two pointers and the data. These two pointers enables the traversals of link list in both the direction, backward and forward. The below figure illustrate the concept of doubly linked list.



* Structure of doubly linked list.

The below structure illustrates doubly linked list structure.

```
Struct dnode
{
    Struct dnode *prev;
    int data;
    Struct dnode *next;
} *start, *end, *ptr;
```

Date
Page

* Advantage of Doubly linked list.

(i) Traversal in either / both direction because convinent.

(ii) it reduces time requirement for the program execution.

(iii) Tree and graph datastructure can be easly represented using a doubly linked list.

* Disadvantage of Doubly linked list:

(i) Each node requires extra space for storing the additional pointer.

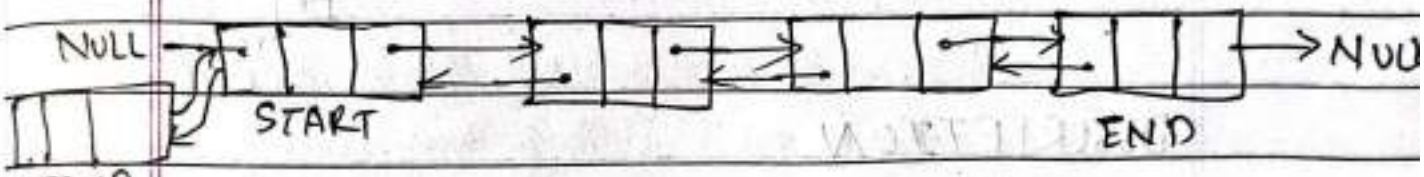
(ii) While manupulating the list, Extra care should be taken to manupulate both link.

	Singly linked list	Doubly linked list
(i)	Traversing is convient in only one direction.	(i) Traversing can be done in both direction.
(ii)	while deleting a node, its predecessor (previous) is required and can be found only after traversing from the begining of the list.	(ii) While deleting a node, it's predecessor can be obtained using 'previous-link' of node. No need to traverse the list.
(iii)	occupies less memory.	(iii) occupies more memory.

(iv.)	Care is take to modify only one link of a node.	(iv.) Care is taken to modify both links of a node.
(v.)	Tree & Graph is not so easily simple method.	(v.)

25/8/25

1.) Insertion at Beginning of doobly linked list



Description

Let START and END are the pointer variable that hold the address of the first and last element of the doobly linked list, VAL is the element to be inserted at the beginning of the list.

Step -> i) Create a node TEMP.

ii.) a.) set TEMP → DATA = VAL
b.) set TEMP ↔ PREV = NULL
c.) Set TEMP → NEXT = NULL

iii.) if [START is equal to NULL] THEN
a.) set START = TEMP
b.) set END = TEMP

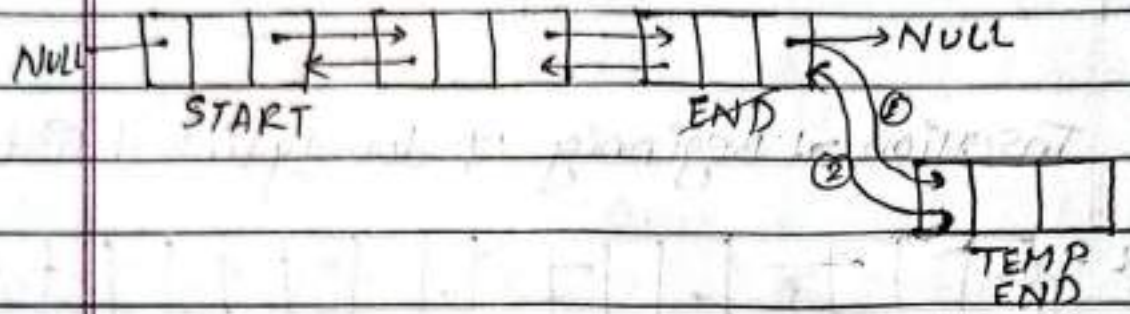
iv.) ELSE
a.) TEMP → NEXT = START
b.) START → PREV = TEMP

c) START = TEMP

v.) END IF

vi.) EXIT.

2.) INSERTION at the END of doubly linked list.



DESCRIPTION

let START and END are the pointer variables that hold the address of the First and last element of the doubly linked list, VAL is the element to be inserted at the beginning of the list.

Step→ 1.) Create a node TEMP

2.) a.) set TEMP→DATA = VAL

b.) set TEMP→PREV = NULL

c.) set TEMP→NEXT = NULL

3.) IF [START is equal to NULL] THEN

a.) set START = TEMP

b.) set END = TEMP

4.) ELSE

a.) END→NEXT = TEMP

b.) TEMP→PREV = END

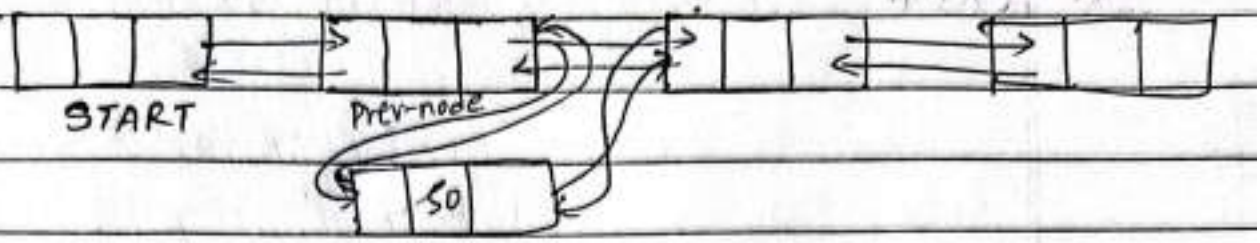
c.) END = TEMP

5.) END IF

6.) EXIT.

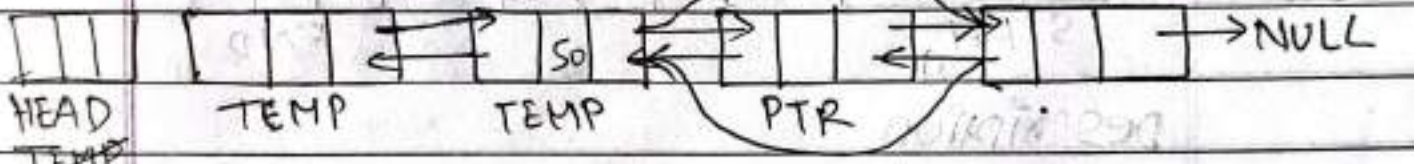
Date
Page

3. INSERTION at Middle of doubly linked list.



1. Create a new-node.
2. Set new-node → data = VAL.
3. Set new-node → next = prev-node → next
4. set prev-node → next = new-node
5. set new-node → prev = prev-node
6. Set ~~prev-node~~ prev → prev (prev-node → next) → prev = new-node
7. Exit.

6. Deletion of Middle node in doubly linked list.

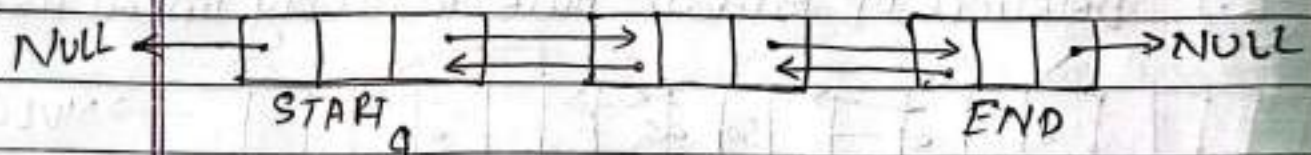


1. IF [HEAD = NULL] THEN
   Display "List EMPTY"
2. ELSE
   (a) Set TEMP = HEAD
   (b) Repeat While TEMP → DATA ! = ITEM
       TEMP = TEMP → NEXT
   (c) END of While
3. Set PTR = TEMP → NEXT
4. Set TEMP → NEXT = PTR → NEXT
5. Set PTR → NEXT → PREV = TEMP

Date
Page

(6) FREE (PTR)
(7) EXIT.

4.) Deletion of 1st / Begining node in doubly linked list



Description

Let START and END are the pointer variable that hold the address of 1st and last node of the doubly linked list and q is the temporary node variable.

step->1.) IF [START is equal to NULL] THEN
Display "list is Empty".
2.) ELSE IF [START = END] THEN
START = END = NULL
3.) ELSE
a) q = START

Date
Page

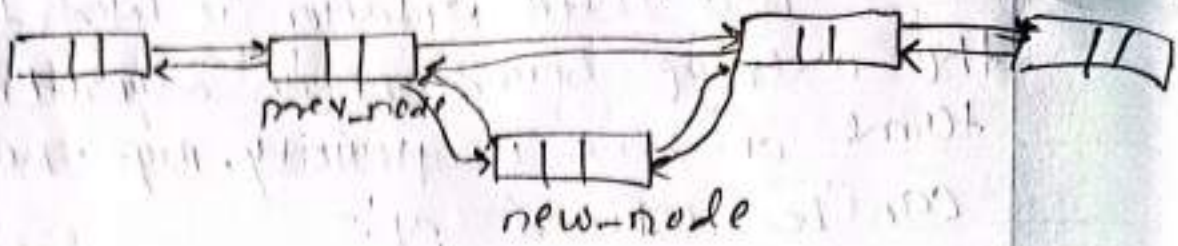
b.) (9->NEXT)-> PREV = NULL
c.) START = 9 -> NEXT
4.) END IF
5.) Free the Node(9)
6.) END IF.

5/96
5.) Deletion of last node in Doubly linked list.

(1.) IF [HEAD IS NULL]
Display "List is Empty".
(2.) ELSE
(a.) Set TEMP = HEAD.
(b.) Repeat while TEMP -> NEXT != NULL.
TEMP = TEMP -> NEXT.
(c.) End of While
(3.) Set TEMP -> PREV -> NEXT = NULL
(4.) FREE (TEMP)
(5.) EXIT.

bly Linked list.

Insertion at Middle in Doubly Linked list.

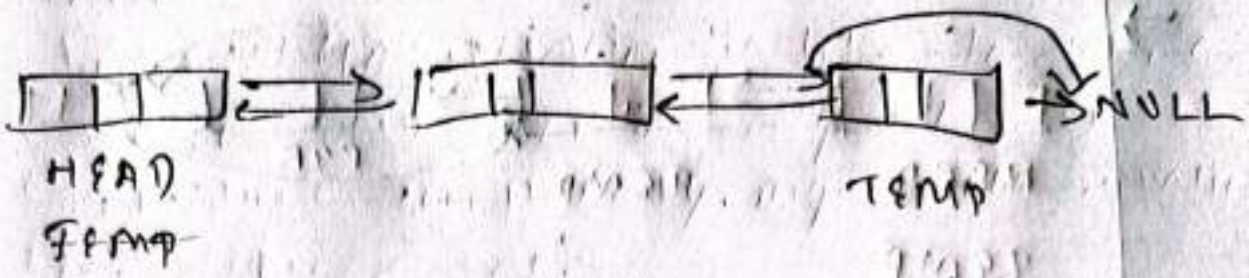


steps :-

1) Create a New_Node.
2) Set New_Node → data = VAL.
3) Set New_Node → next = prev_node → next
4) Set prev_node → next = new_node
5) Set New_node → prev = prev_node
6) Set (prev_node → Next) → prev = new_node
7) Exit.

* Deletion of Last Node in Doubly Linked list.

→



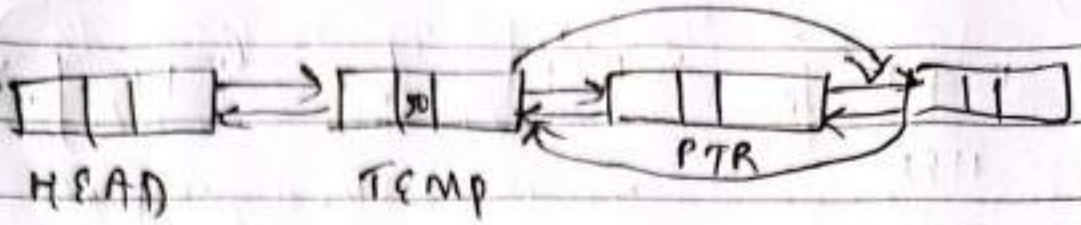
steps :-

1) IF [HEAD is NULL] THEN
   Display ("List is Empty".
2) ELSE (a) SET TEMP = HEAD.
   (b) Repeat while TEMP → NEXT != NULL
   TEMP = TEMP → NEXT
   (c) End of While
3) Set TEMP → PREV → NEXT = NULL
4) FREE (TEMP)
5) EXIT.

(After the node)

* Deletion of Middle Node in Doubly linked list.

→



TEMP

Steps :-

1) IF [HEAD = NULL] THEN
Display "List is empty".

2) ELSE
a) Set TEMP = HEAD
b) Repeat while TEMP → DATA != ITEM
TEMP = TEMP → NEXT
c) End of while
3) Set PTR = TEMP → NEXT
4) Set TEMP → NEXT = PTR → NEXT
5) Set PTR → NEXT → PREV = TEMP
6) FREE (PTR)
7) EXIT.

Tree

* Tree

Trees are very flexible, versatile and powerful non-linear data structure that can be used to represent data items by processing hierarchal relationship between the grand-father and his children and grand children and so on.

* Binary Tree

A binary tree is a tree in which the degree of each node is less than or equal to two i.e. each node in the tree can have zero or one or two child.

or

A Binary tree is a tree in which no node can have more than two sub-tree that means each node can have zero or one or two sub-trees.

* Tree traversal

Tree traversal is a way in which each node in the tree is visited exactly one in a systematic manner. There are three ways of traversing a binary treee. These are:

1.) pre-order Traversal
2.) In-order Traversal
3.) post-order Traversal

Date
Page

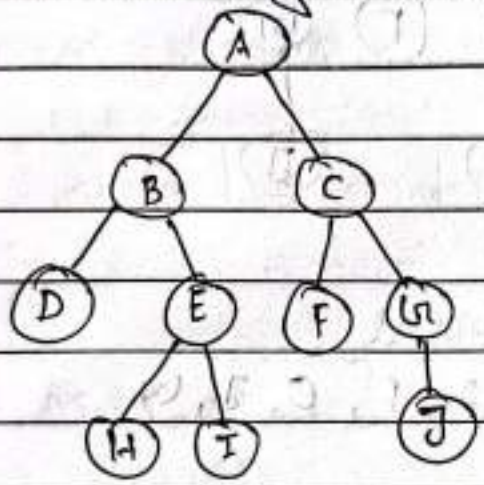
1.) pre-order Traversal [ROOT-Left-Right]

The pre-order traversal of a non-empty binary tree is defined as :

i.) Visit the root node.

ii.) Traverse the left sub-tree in pre-order

iii.) Traverse the right sub-tree in pre-order.



pre-order Traversal [ROOT-LEFT-Right]

A, B, D, E, H, I, C, F, G, J

2.) In-order Traversal [Left - ROOT - Right]

In the in-order traversal of a non-empty binary tree, The left sub-tree is traverse recursively before visiting the root. After visiting the root The right sub tree is traverse recursively in the in order way.

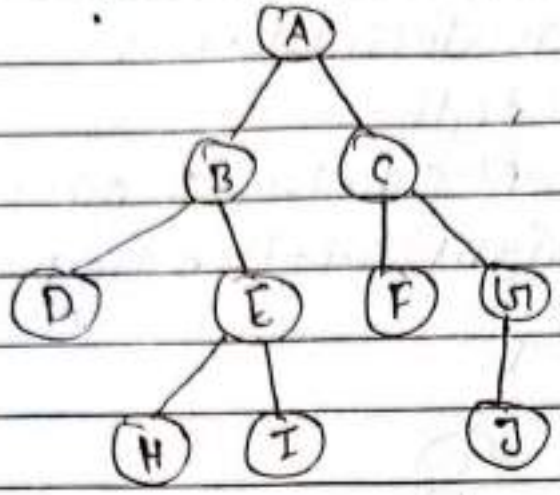
V.V.I Algorithm

The in-order traversal of a non-empty binary tree is defined as :

(i.) In-order traverse the left Sub-Tree.

(ii.) Visit the root of the tree.

(iii) Inorder traverse the right sub-tree.

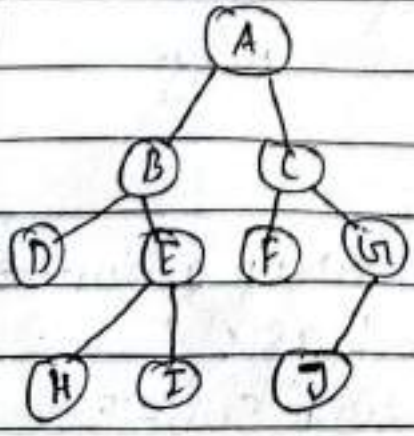


In-order traversal
D, B, H, E, I, A, F, C, J, G

3. post-order Traversal

The post-order traversal of a non-empty binary tree can be defined as:-

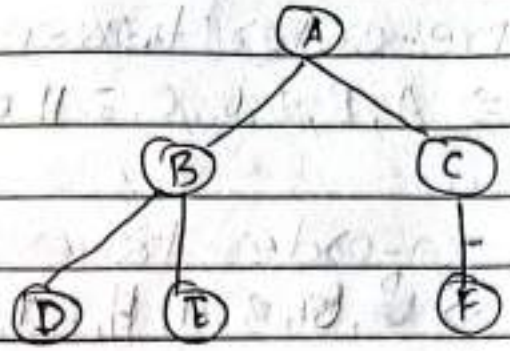
(i) post order traverse the left sub-tree
(ii) post order traverse the right sub-tree
(iii) visit the root node.



post-order Traversal
D, H, I, E, B, F, J, G, C, A

Date
Page

1.

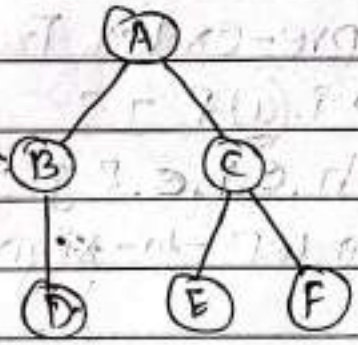


⇒ pre-order traversal
A, B, D, E, C, F

⇒ In-order traversal
D, B, E, A, F, C

⇒ post-order traversal
D, E, B, F, C, A

2.

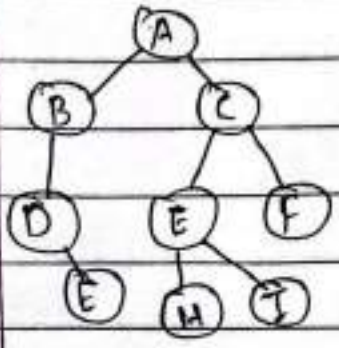


⇒ pre-order traversal
A, B, D, C, E, F

⇒ In-order traversal
B, D, A, E, C, F

⇒ post-order traversal
D, B, E, f, C, A

3.

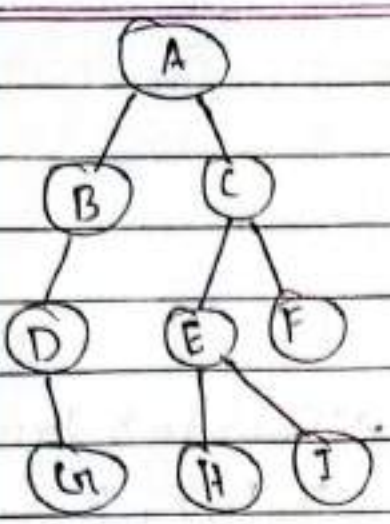


pre-order traversal
⇒ A, B, D, E, C, E, H, I, F

In-order traversal
⇒ D, E, B, A, H, E, I, E, F

post-order traversal ⇒ E, D, B, H, I, E, F, C, A

Q.



pre-order traversal
⇒ A, B, D, G, C, E, H, I, F

In-order traversal
⇒ G, D, B, A, H, E, I, C, F

post-order traversal
⇒ G, D, B, H, I, E, F, C, A

2078/08/27

V.V.I

* Binary Tree Construction

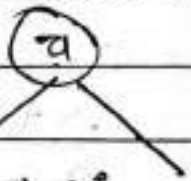
Q. From In-order and pre-order Traversal

In-order :- g d h b e i a f j c

pre-order :- a b d g h e i c f j

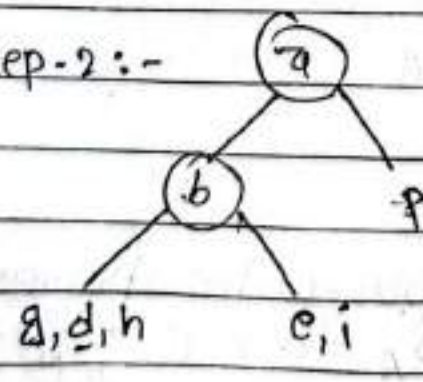
scan from left-to-right

Step-1 :-



g, d, h, b, e, i f, j, c

Step-2 :-

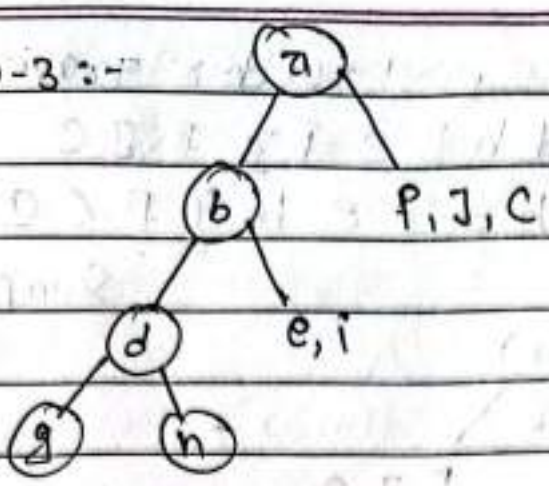


f, j, c

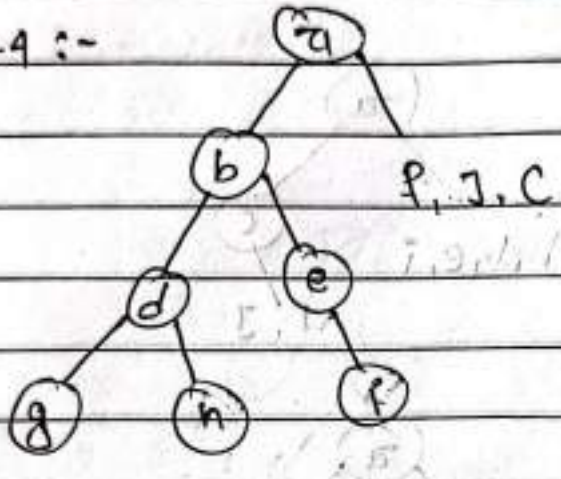
g, d, h e, i

Date
Page

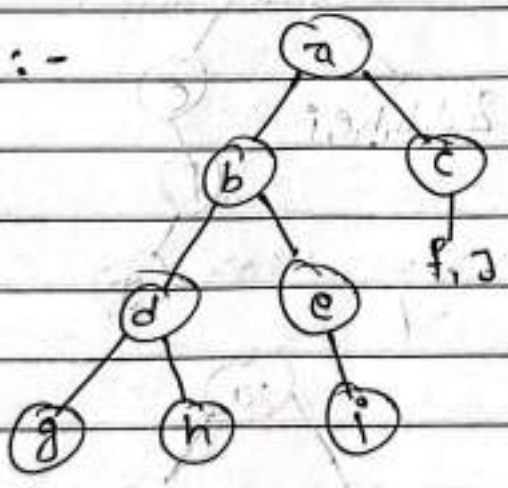
Step-3 :-



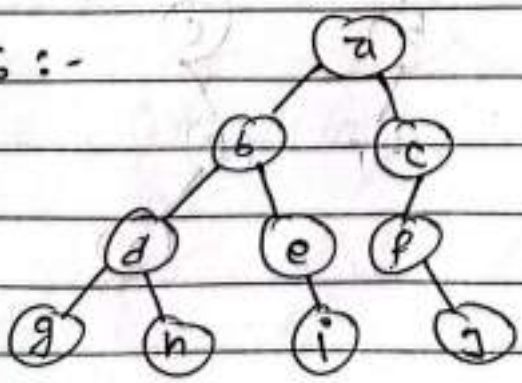
Step-4 :-



Step-5 :-



Step-6 :-



This is the required Binary Tree.

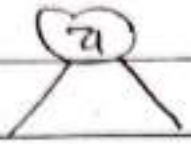
(ii) From In-order and post-order traversal

Inorder :- g d h b e i a f J C

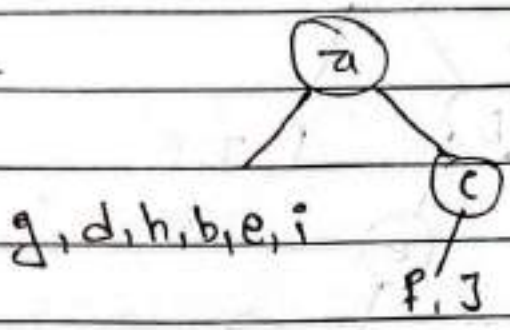
postorder :- g h d i e b J f C a

Scan from Right to left

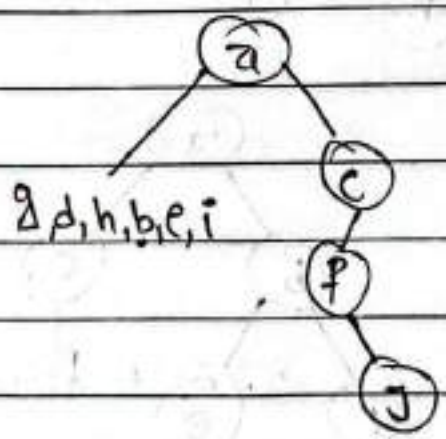
Step-1 :-



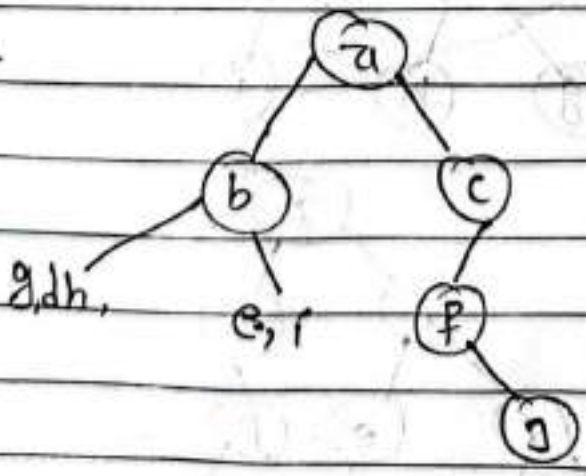
Step-2 :-



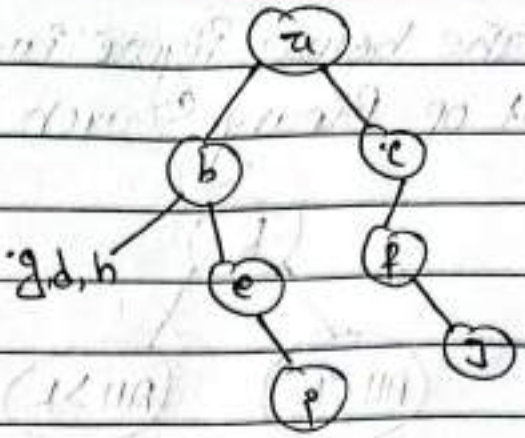
Step-3 :-



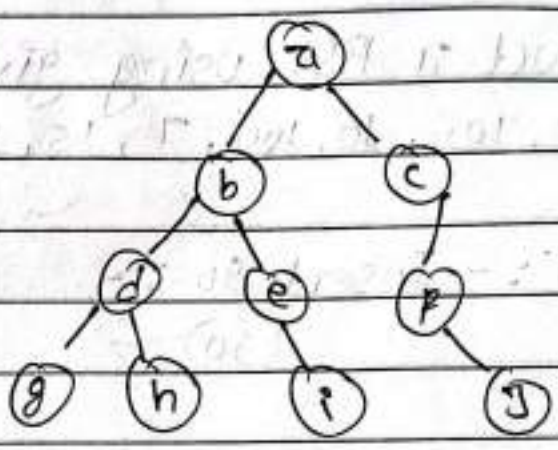
Step-4 :-



Step-5 :-



Step-6 :-



This is required Binary Tree.

~~Step~~

* BST (Binary Search Tree)

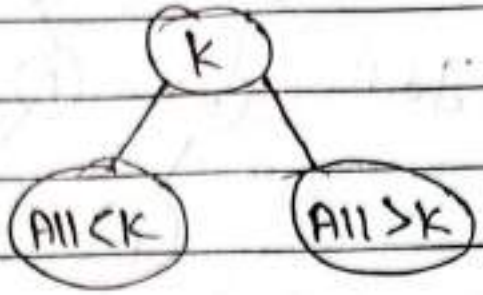
A Binary Search tree is a binary tree which is either empty or satisfies the following properties

(i) every node has a value and No two nodes have the same values.

(ii) If there exist a left child or left sub-tree then ~~this~~ it's value is less than the value of the root node.

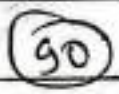
(iii) If there exist a right child or right sub-tree then it's value is larger than the value of the root node.

The below figure illustrate the concept of Binary Search Tree.

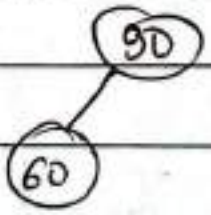


Q.) Construct a BST using given data.
90, 60, 105, 10, 100, 7, 15, 200

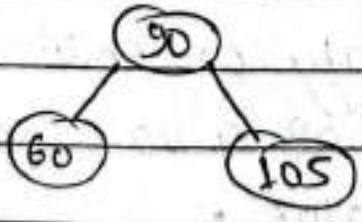
Step-1 :- Insert 90 :-



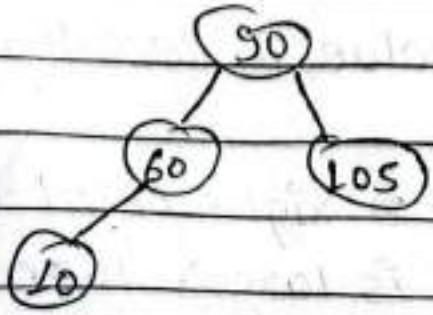
Step-2 :- insert 60 :-



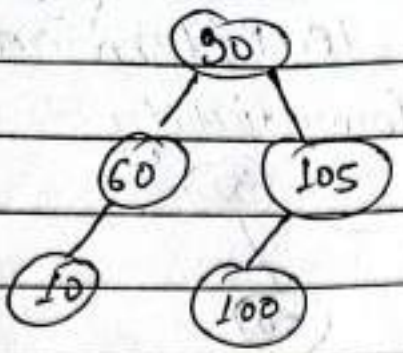
Step-3 :- insert 105 :-



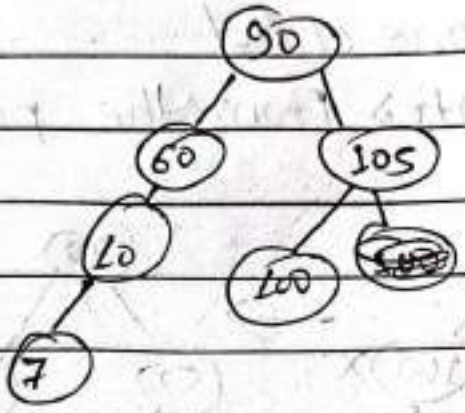
Step-4 :- insert 10 :-



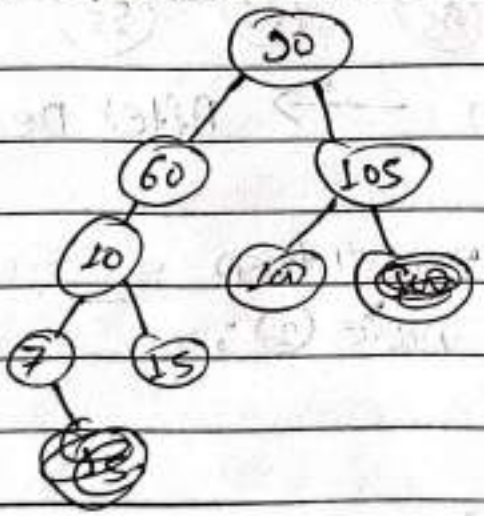
Step-5 :- Insert 100 :-



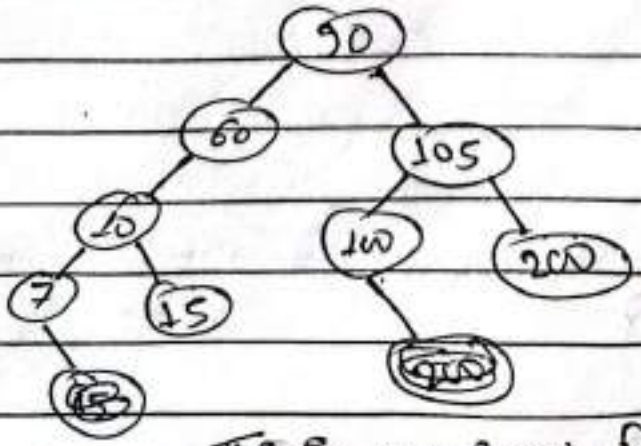
Step-6 :- Insert 7 :-



Step-7 :- Insert 15 :-



Step-8 :- Insert 200 :-



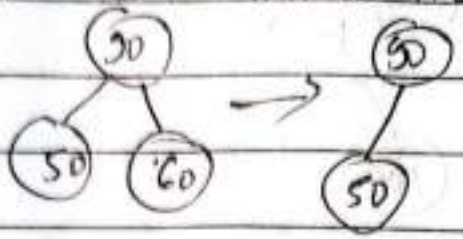
This is required Binary Search Tree.

Date: Page:

5.5) Three cases while deleting binary Tree's node.

(I.) The node is a leaf/terminal.

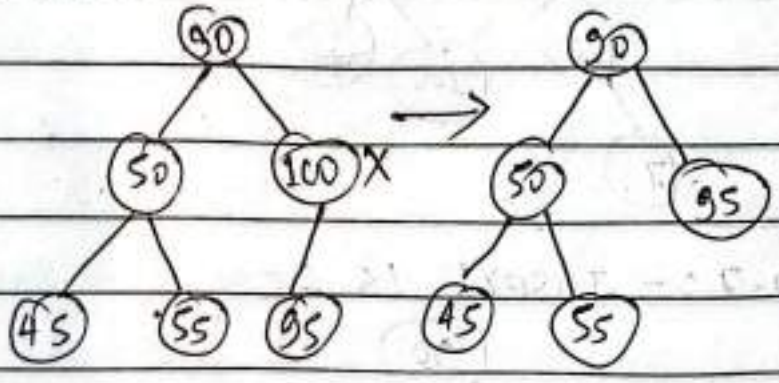
* Delete it immediately



Before Deletion ⟶ After Deletion

(II) The node has one child.

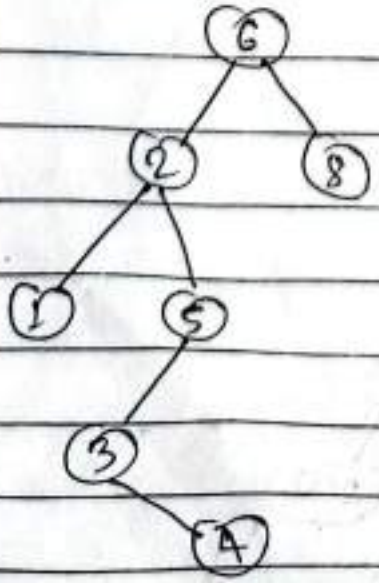
⇒ Adjust a pointer from the parent to bypass that node.



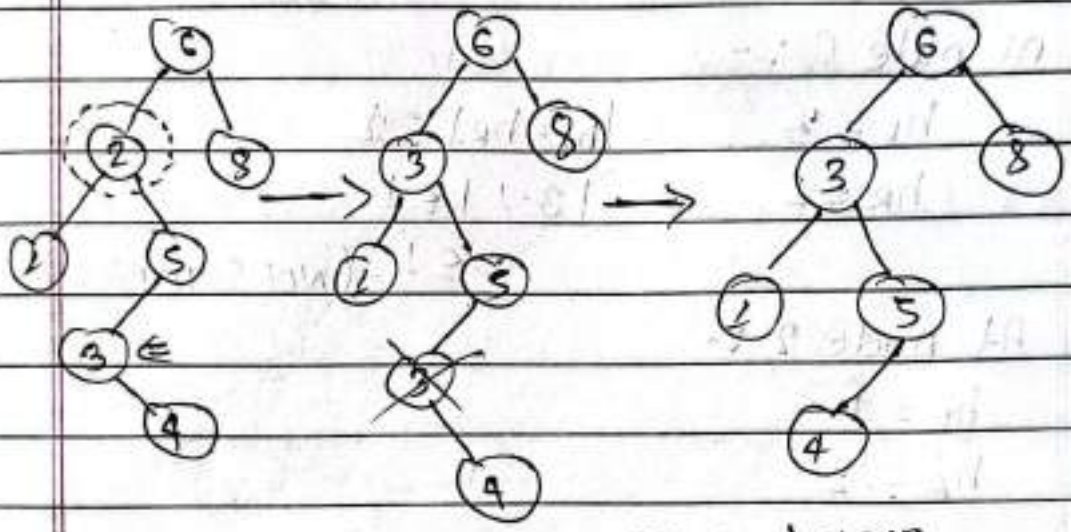
Before deletion ⟶ After Deletion

(III.) The node has 2 children

For deletion of node ②:-



Steps:-
- Find the minimum element at a right sub-tree of the node to be deleted.
- Copy that found minimum element to the node to be deleted.
- delete the found minimum element at the right sub-Tree.

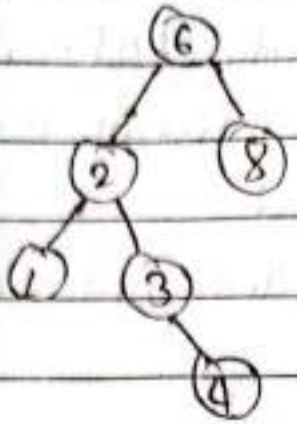


After deletion of node 2, we have the above BST Tree.

AVL Tree.

AVL Tree is a self balancing, height balanced Binary Search Tree whose left sub-tree and the right-sub tree differ in height by almost 1 unit and whose left and right sub-tree are themselves AVL Tree that means for AVL Tree every node must satisfy the balance factor. $|h_L - h_R| \leq 1$ Condition where $h_L \rightarrow$ hight of left sub-tree, $h_R \rightarrow$ height of Right sub-Tree.

Example :-



At node 6 :-

$h_L = 3$ $\quad |h_L - h_R| \leq 1$

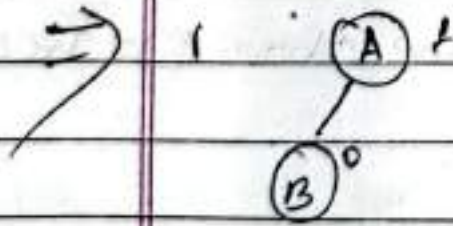
$h_R = 1$ $\quad |3-1| \leq 1$

$2 \leq 1$ (NOT SATISFIED)

At node 2 :-

$h_L = 1$

$h_R = 2$



AVL Tree

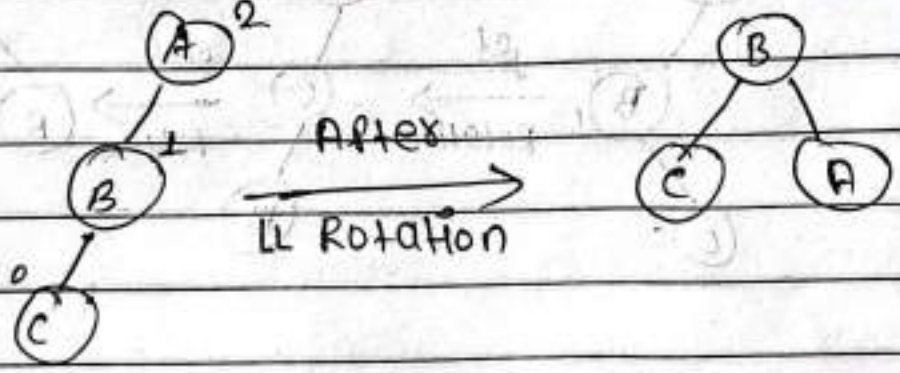
Date
Page

* Rules for Balancing unbalance tree.

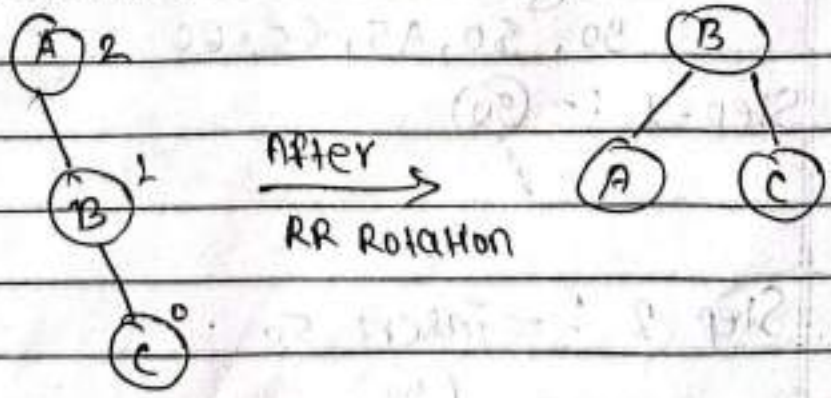
1.) The unbalance is tree is balance through Rotation of nodes. There are two types of rotation
(i.) Single Rotation
(ii) Double Rotation

i.) Single Rotation
it is performed by:

(a) LL Rotation :-



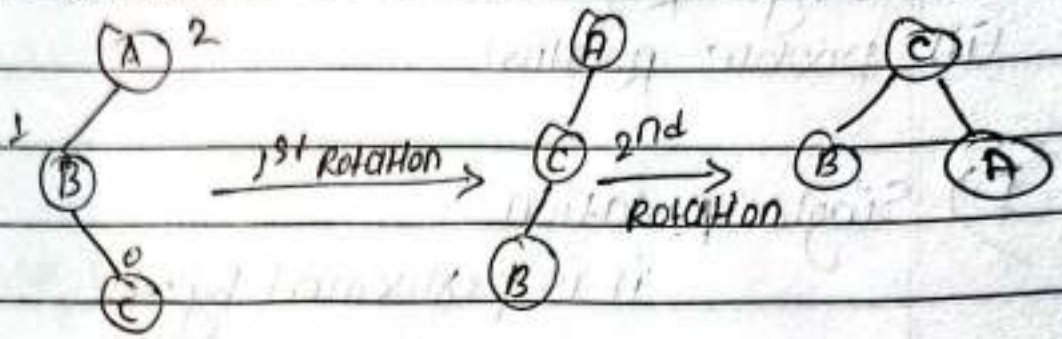
(b.) RR Rotation



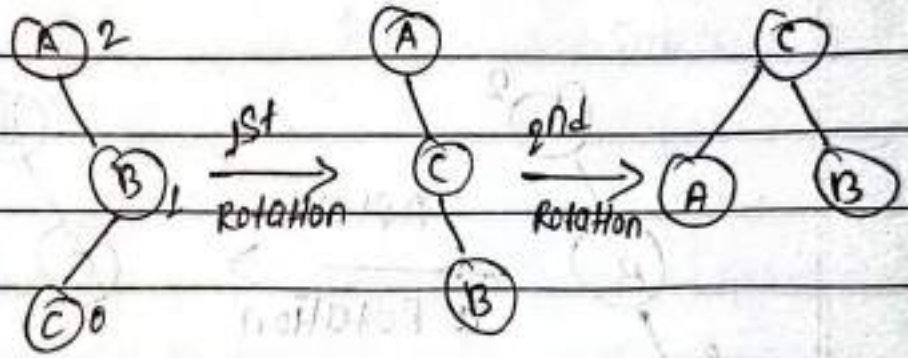
(ii) Double Rotation.

It is performed by: ~~LR Rotation~~

a) LR Rotation :-



b) RL Rotation :-



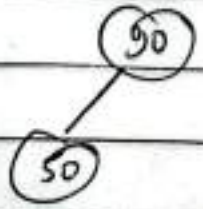
2080/08/29

Construct BST and AVL Tree for the given data :-

90, 50, 45, 65, 66

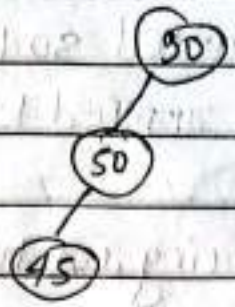
Step-1 :- 90

Step 2 :- insert 50

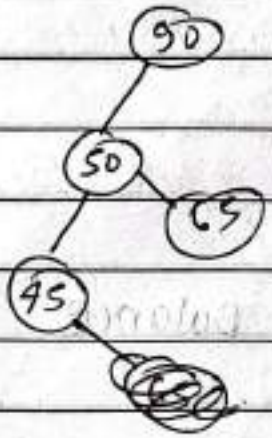


Date
Page

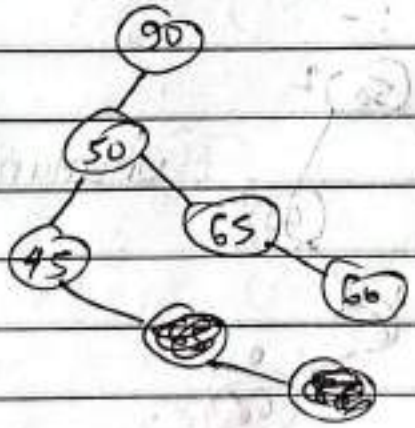
Step 3 :- insext 45



Step 4 :- insert 65



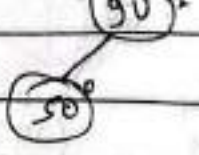
Step 5 :- insert 66



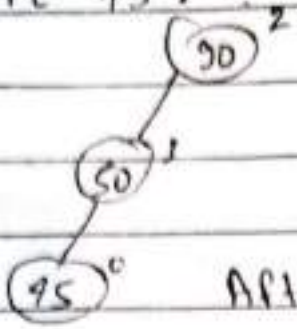
For AVL Tree

Step (i.) insert 90 :- 90 Balance

Step (ii) Insert 50 :- 90 Balance

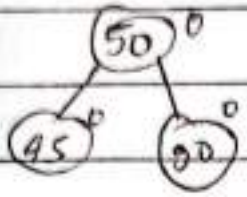


Step (iii) Insert 45 :-

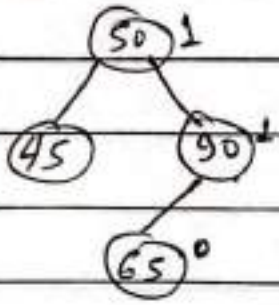


unbalanced so LL Rotation is applied to balance it.

After balancing, we have

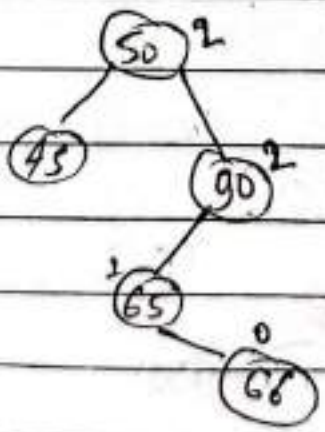


Step (4) Insert 65 :-



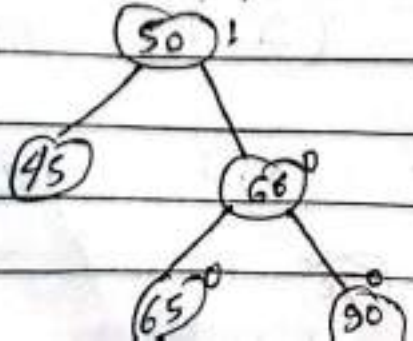
Balance

Step (5) Insert 66 :-



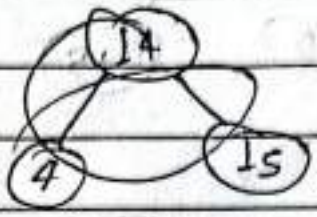
unbalanced

To balance it, double Rotation is applied it



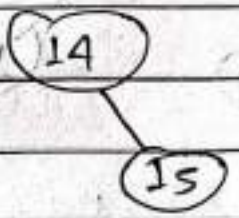
Balance

Q.1.) Draw BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 9, 20, 17, 9, 14, 5.

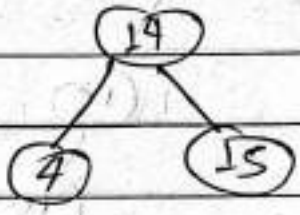


Step - (1)

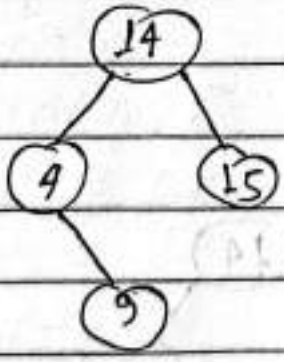
Step - (2)



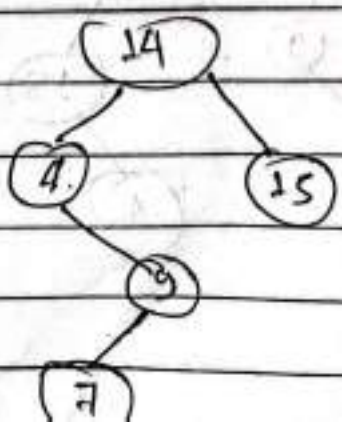
Step - (3)



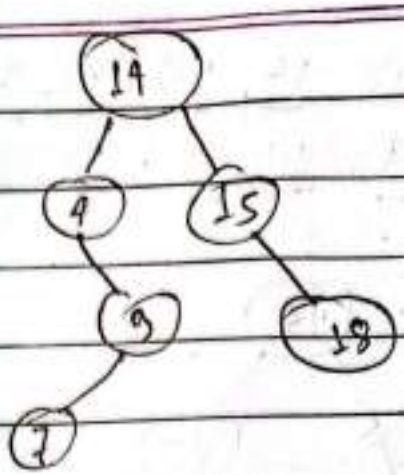
Step - (4)



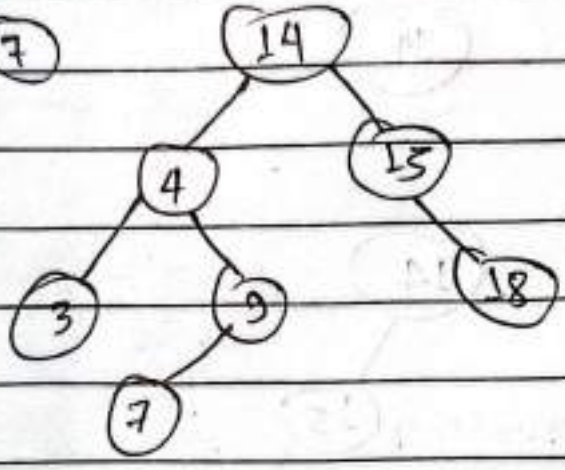
Step - (5)



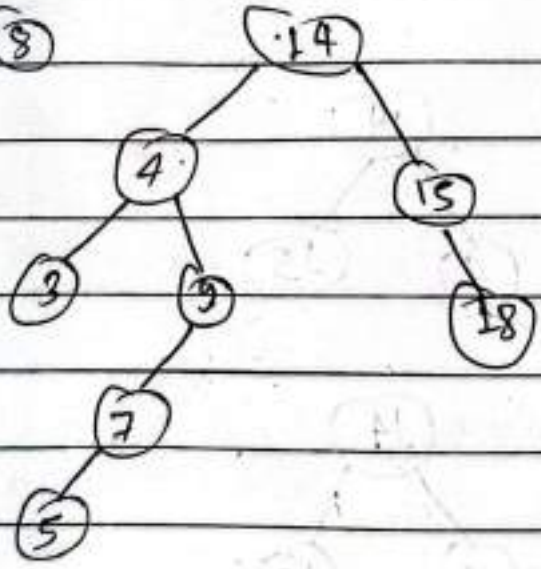
Step - 6



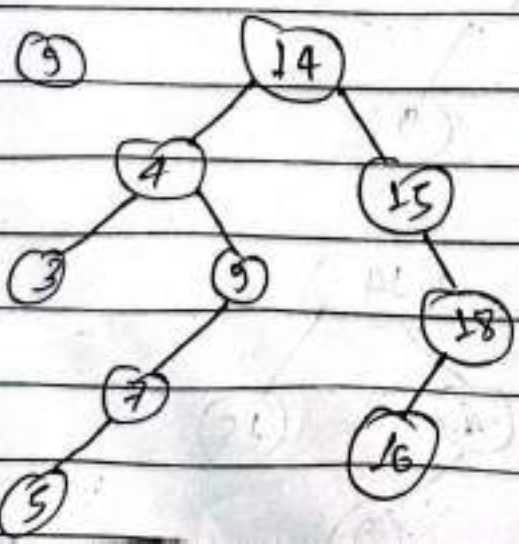
Step - 7



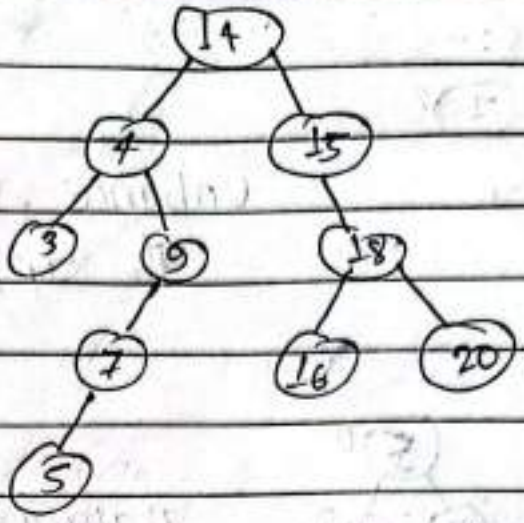
Step - 8



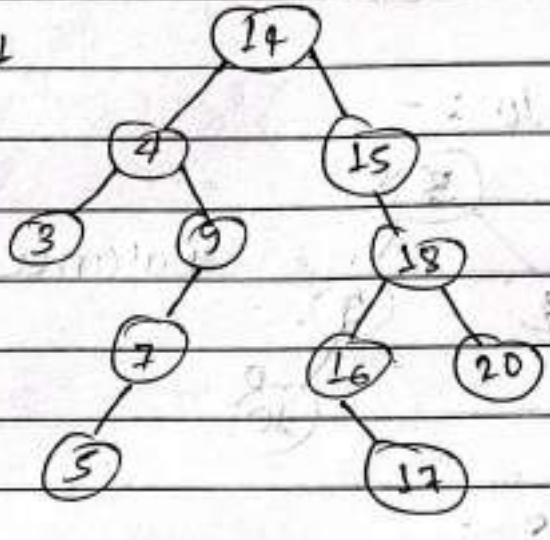
Step - 9



Step-10



Step-11



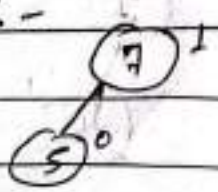
~~Step~~. This is required BST.

Q.) 2.) Construct an AVL tree from following.
7, 5, 3, 10, 2, 8, 9

Step-1 Insert 7 :-

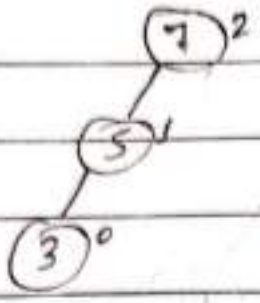
Balance

Step-2 Insert 5 :-

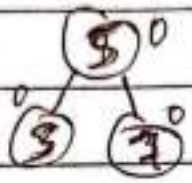


Balance

Step-3 Insert 3 :-

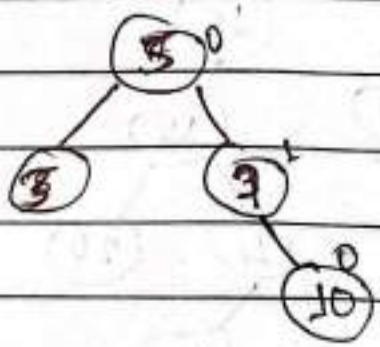


Unbance, so double rotation is applied it.



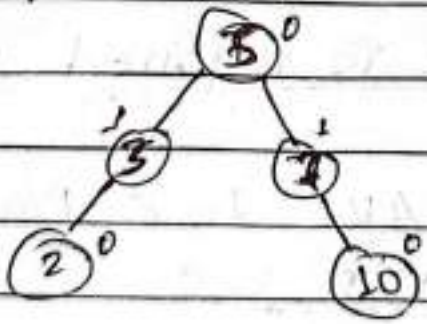
Balance

Step-4 Insert 10 :-



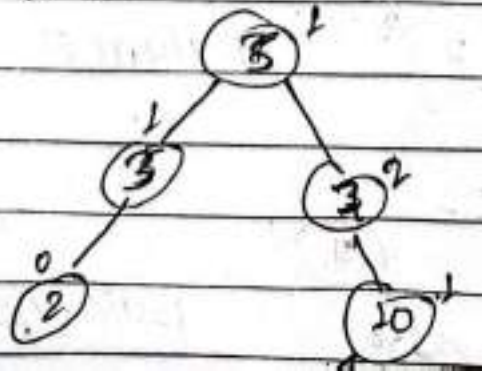
Balance

Step-5 Insert 2 :-

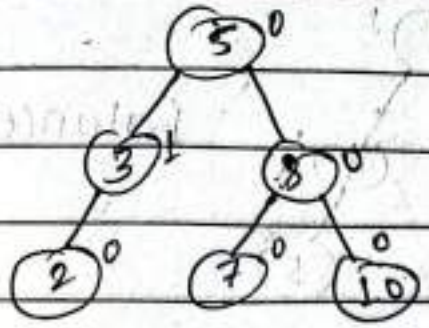


Balance

Step-6 Insert 8 :-



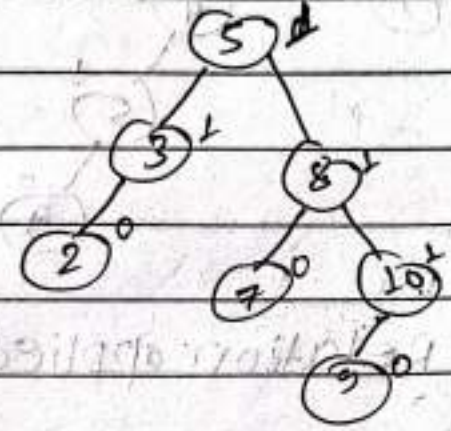
UnBalanced, so double rotation applied it.



Balanced

Insert.

Step-7. Insert 9 :-



Balanced

So, The required AVL tree is.

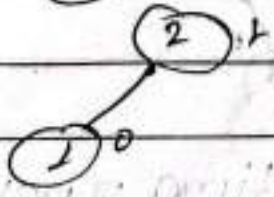
Q.) 3.) Construct an AVL tree from following.

2, 1, 4, 5, 9, 3, 6, 7

Step-1 insert (2) :-

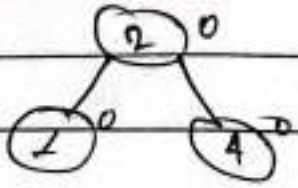
Balanced

Step-2 Insert (1) :-



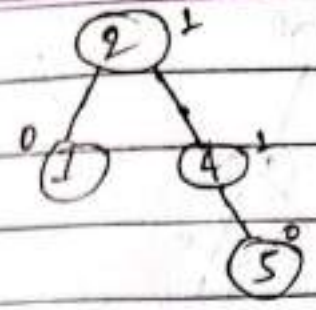
Balanced

Step-3 Insert (4) :-



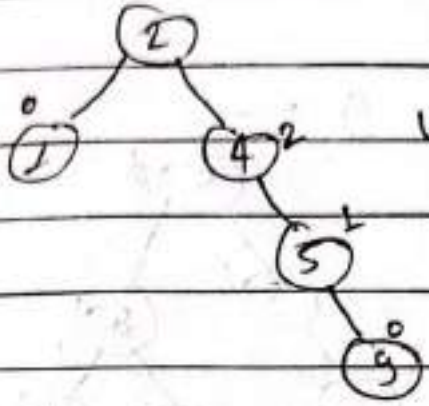
Balanced

Step-4 Insert (5) :-



Balanced

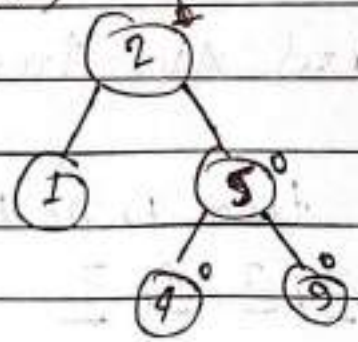
Step-5 Insert (9) :-



unbalanced

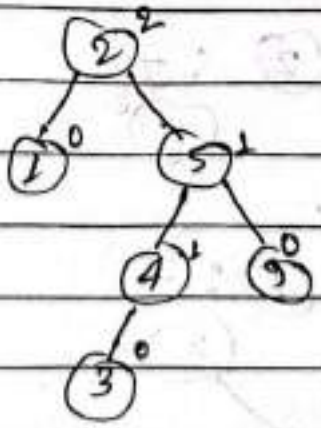
So, double Relation applied it.

Now,



Balanced

Step-6 insert (3) :-

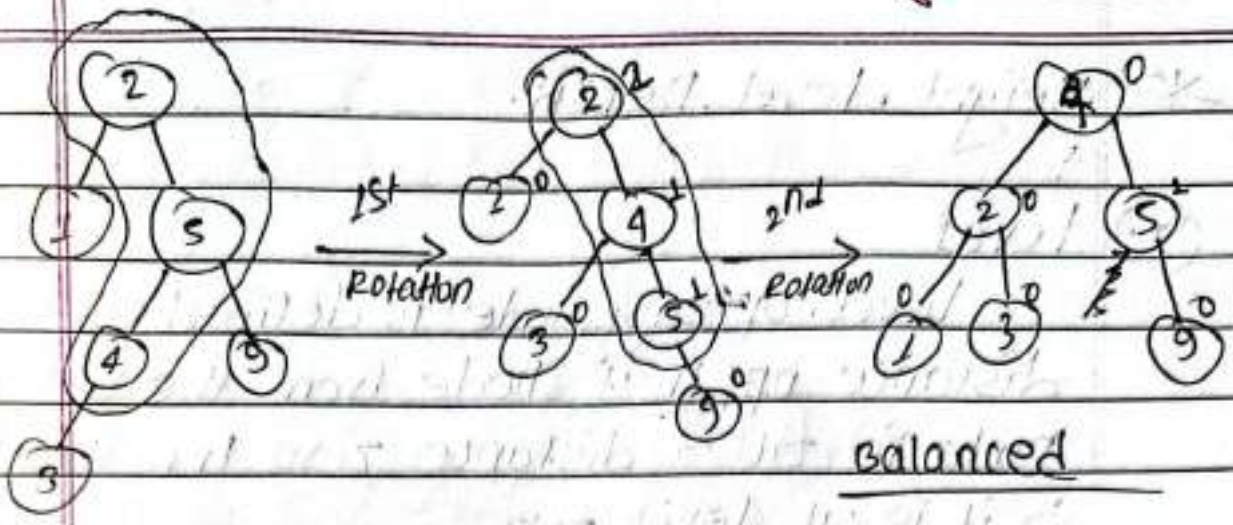


unbalanced

So, double Relation applied it,

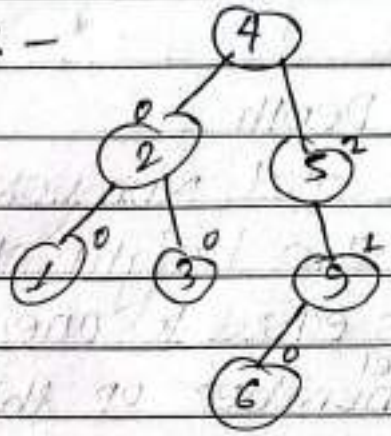
Now,

Date
Page

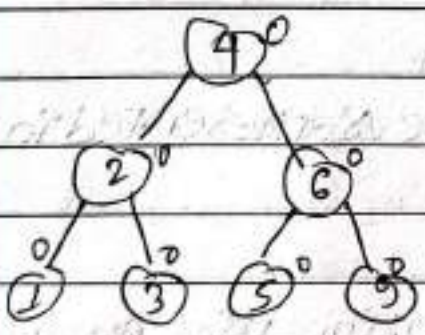


Balanced

Step-7 insert (6) :-

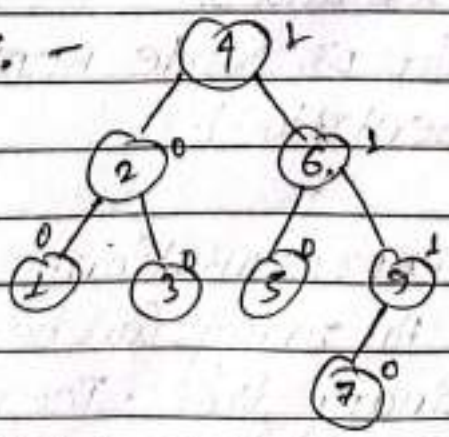


unbalanced
So, RL Rotation applied it,
Now,



Balanced

Step-8 Insert (7) :-



Balanced

So, The required AVL Tree.

Date
Page

* Height, level, Depth.

(1.) level

level of any node is defined as the distance of that node from the root. Root node is at a distance zero from itself so it is at level zero.

(2.) Height & Depth

The total number of levels in a tree is the height of the tree. So, ~~Hei~~ Height is equal to one more than the largest level number of the tree.

* Node Representation

The tree can be represented in two ways :-

(1.) Sequential Representation using Array (static Representation).

(2.) Linked list or, Node Representation (dynamic Representation).

1.) Array Representation.

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially.

Date 2080/09/02
Page

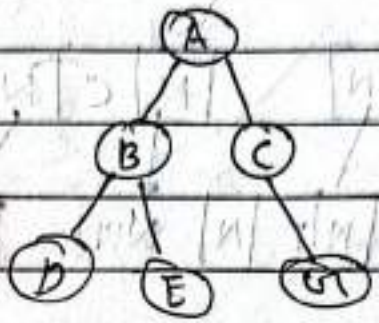


Fig :- Binary Tree of Depth 3.

In array the above tree can be represented below :-

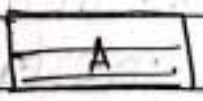
	0	1	2	3	4	5	6
A[7] =	A	B	C	D	E		G

Fig :- Array representation of the Binary tree.

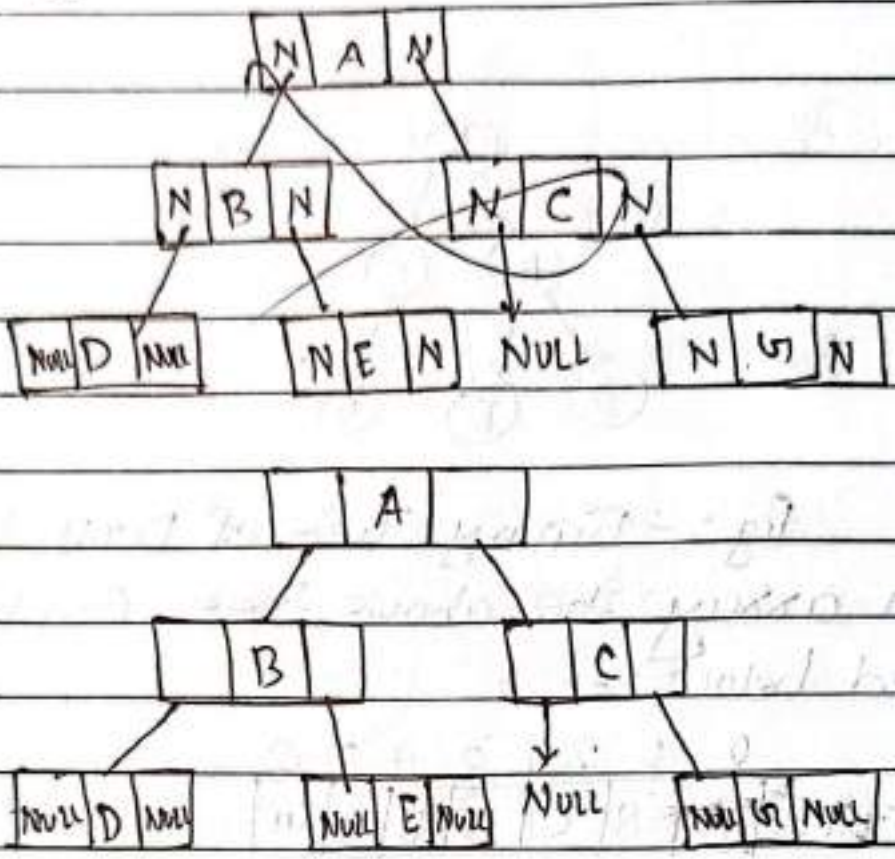
(2.) Dynamic Representation.

The most popular and practical way of representing a binary tree is using linked list. In link list, every element is represented as nodes. A node consist of three field such as :

left child, Information of the node and right child



The above binary tree can be represented using link list as follows:



* Application of Binary tree

(i.) It is used in interpreter or compiler design for programming language.

(ii.) It is used for obtaining the pre-fix and post-fix equivalent of a given in-fix notation or an expression.

(iii.) Searching a binary tree for a particular number is easy and takes less number of comparisions in compared to linear search

(iv.) It is used in representing arithmatic and boolen expression.

(v.) It is used in representing priority queue.

Date
Page

3.5 priority queue

A priority queue is a queue where each element is assigned a priority. In priority queue the elements are deleted and processed by following rules:

(i) An element of higher priority is process before any element of lower priority.
(ii) Two elements with the same priority are processed according to the order in which they are added to the queue.

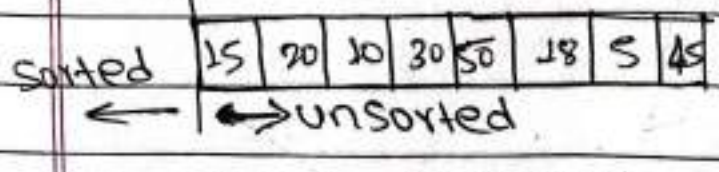
Unit-6

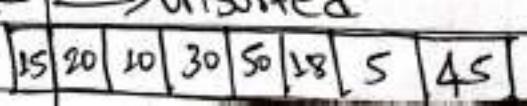
Sorting :-

* Insertion Sort :-

Insertion sort algorithm arranges a list of elements in a particular order. In this algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

example :- 15, 20, 10, 30, 50, 18, 5, 45





Date
Page

Sorted ← | → unsorted

15	20	10	30	50	18	5	45

Sorted ← | → unsorted

10	15	20	30	50	18	5	45

Sorted ← | → unsorted

10	15	20	30	50	18	5	45

Sorted ← | → unsorted

10	15	20	30	50	18	5	45

Sorted ← | → unsorted

10	15	18	20	30	50	5	45

Sorted ← | → unsorted

5	10	15	18	20	30	50	45

Sorted ← | → unsorted

5	10	15	18	20	30	45	50

2080|05|02

* Types of sorting

There are two types of sorting method.

(i) Internal Sorting

(ii) External Sorting

Date
Page

(i) Internal Sorting

An Internal sorting is a sorting process in which all of the data are held in primary memory during the sorting process.

Internal sorting is applied when the entire collection of data to be sorted is small enough that the sorting can take place within the main memory.

(ii) External Sorting

An external sort uses primary memory for the data currently being ~~use~~ sorted and secondary storage for any data that will not ~~feet~~ fit in primary memory.

External sorting method is applied to large collection of data which reside on secondary devices.

* Sorting :- Sorting is a process of arranging a list of element in a particular order (ascending and decending).

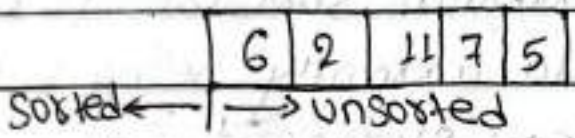
2. * Selection Sort

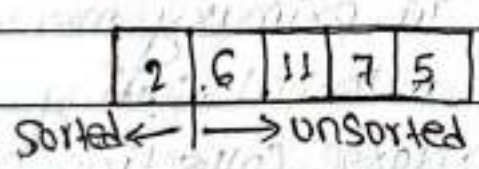
This sorting algorithm is an inplace comparision based algorithm in which the list is devided into two parts, the sorted part at the left end and the unsorted part at the right end.

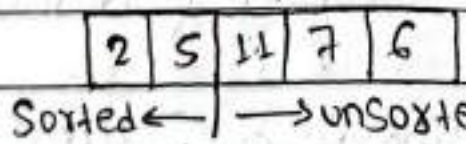
Initially, the sorted part is empty and the unsorted part is the entired list.

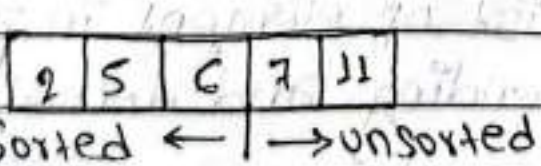
The smallest element is selected from the unsorted array and the swapped with the left most element and that element becomes a part of the sorted array. These process continues moving unsorted array boundary by one element to the right. eg:-

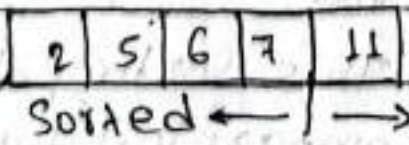
Selection Sort :- 6, 2, 11, 7, 5

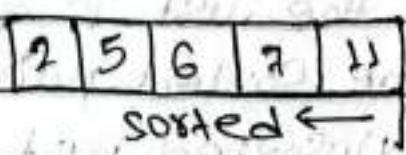








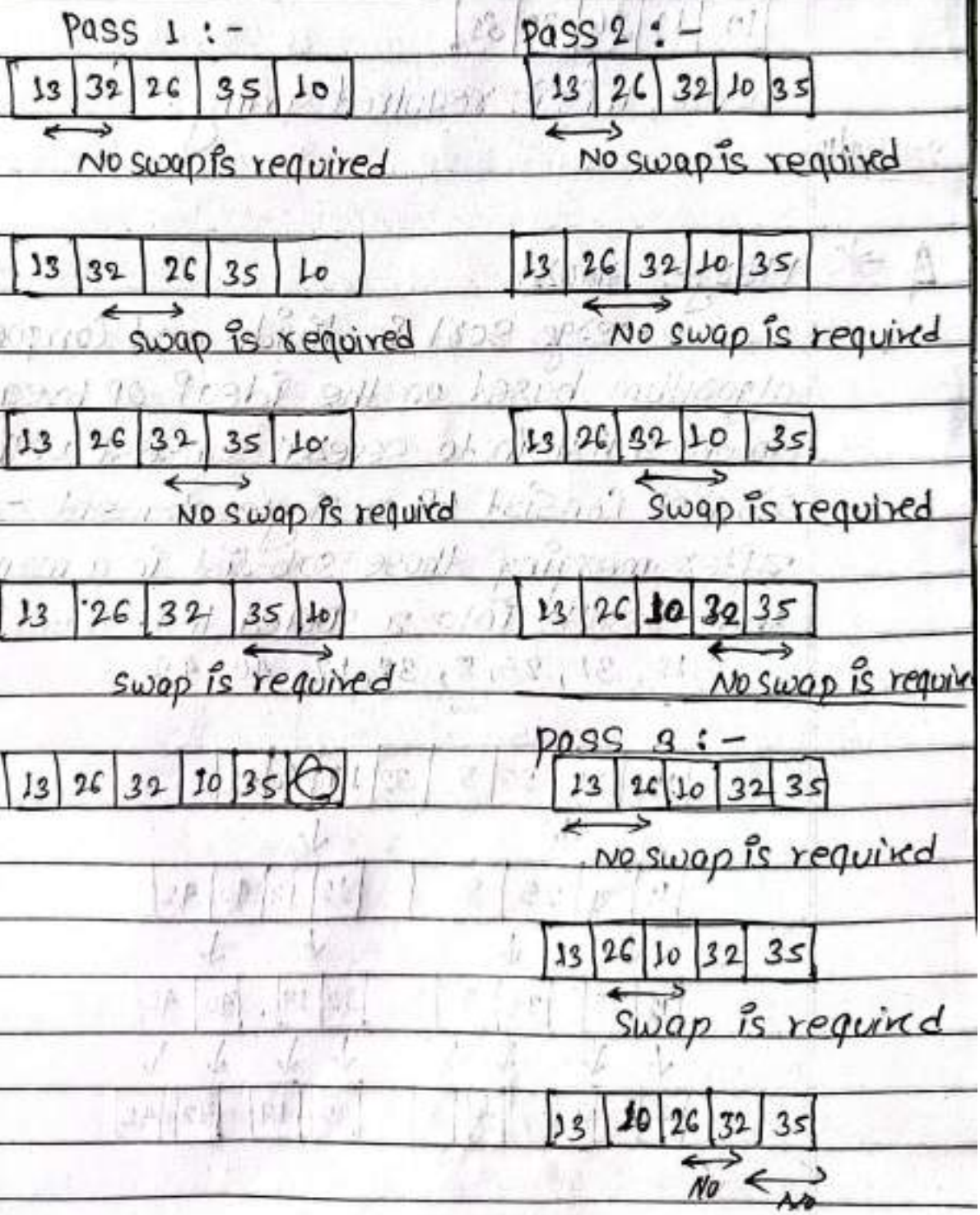




Hence, the required sorted :

2. * Bubble Sort

Bubble Sort algorithm is comparision based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. Eg :- 13, 32, 26, 35, 10



Pass 4 :-

13	10	26	32	35

swap is required

10	13	26	32	35

NO NO NO

10	13	26	32	35

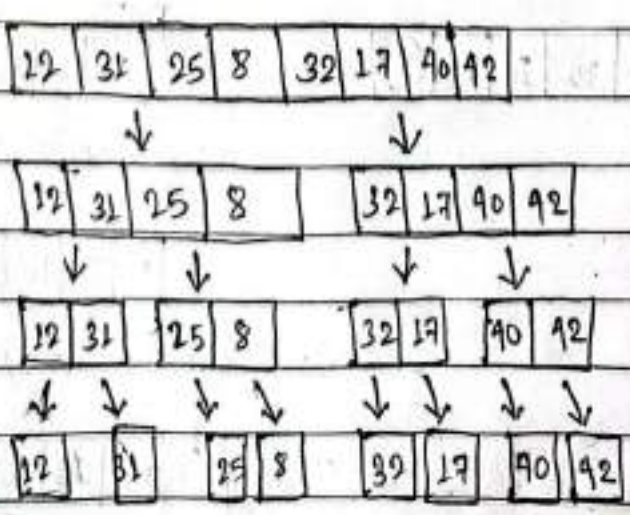
Hence, this is required sorting.

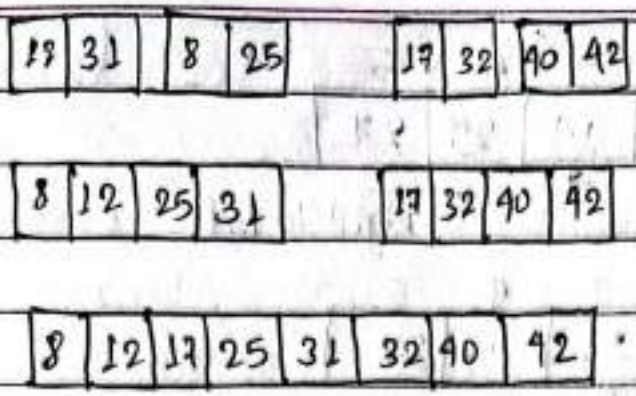
2080/09/12

4. * Merge Sort

Merge sort is divide and conquer algorithm based on the ideaf of breaking down a list in to several sub-list until each sub-list consist of a single element and then after merging those sub-list in a manner that results into a sorted list. Examples :-

12, 31, 25, 8, 32, 17, 40, 42



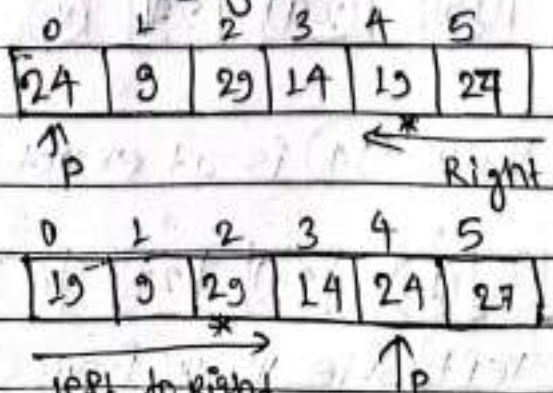


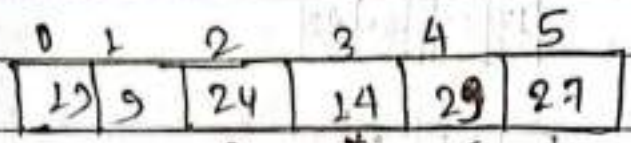
∴ This is required Merge sort data.

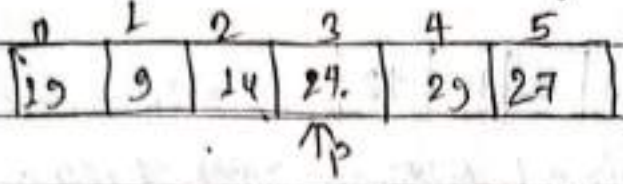
5. * Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partioning of array of data into smaller array. In Quick sort a large array is divided into two arrays in which one holds values that are smaller than the specified value (pivot) and another array holds the values that are greater than the (pivot).

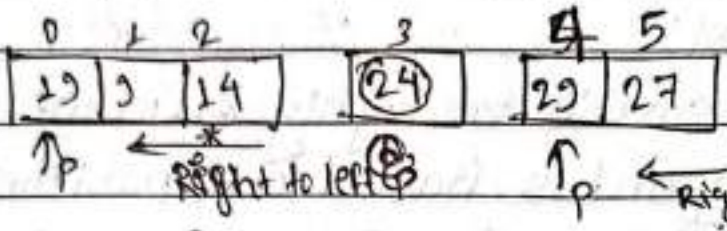
After that left and right sub arrays are also partioning using the same approach. it will continue until the single elements remains in the sub array. The below figure illustrate the concept of Quick sort. eg :- 24, 9, 29, 14, 19, 27

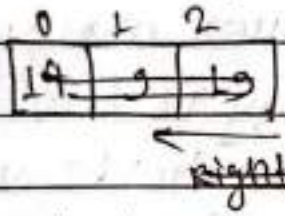


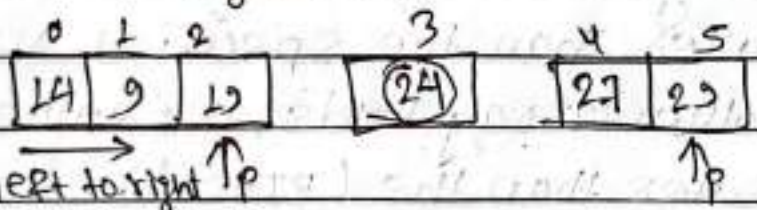




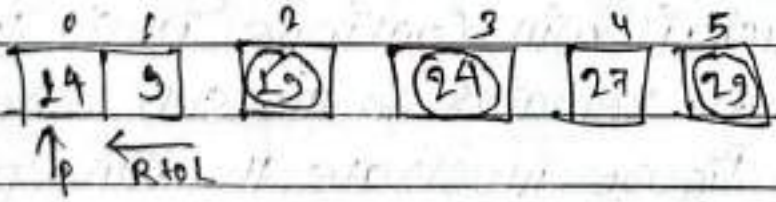
Now, Pivot value (24) is exact position.

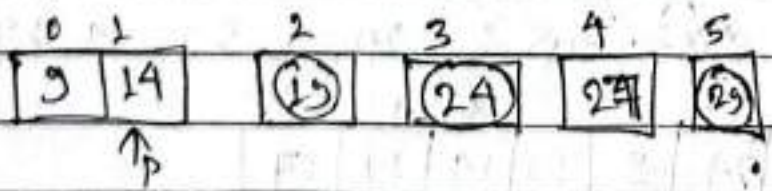




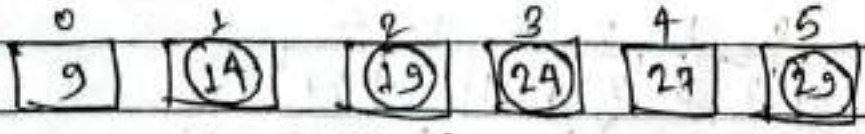


Now, pivot value (19) is at exact position.
Now, pivot value (29) is at exact position.





Now, Pivot value (14) is at exact position.



Since, all the left side value from (9) are smaller and Right side values are greater than (9), so it is also at

0	1	2	3	4	5
9	19	15	24	27	29

So, The Sorted data are :- 9, 19, 19, 24, 27, 29.

6. * Heap Sort

Heap Sort is the comparison based Sorting algorithm that sorts an array in specific order by transforming it into a binary heap data structure. It uses the concept of Max-heap and min-heap. In Max-heap the largest element is the root node and In Min-head, the smallest element is the root node.

* What is heap ?

→ Heap is a complete binary tree.

Complete binary tree is a binary tree in which all the levels ~~are~~ except the last level that is leaf node should be completely filled and all the node should be left justified. eg:

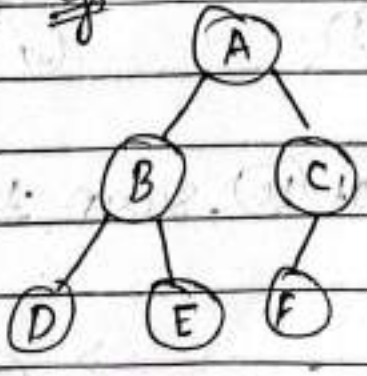
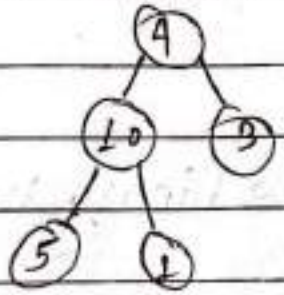


Fig :- Complete binary tree.

Steps 1) Construct a complete binary tree (heap tree) with given data
2) Transform the heap tree into max-heap tree.
3) Delete the root element from MAX-heap tree.
4) put the deleted element into the Sorted stack.
5) Repeats steps Two to four until max-heap tree become empty.
6) Display the Sorted stack elements.

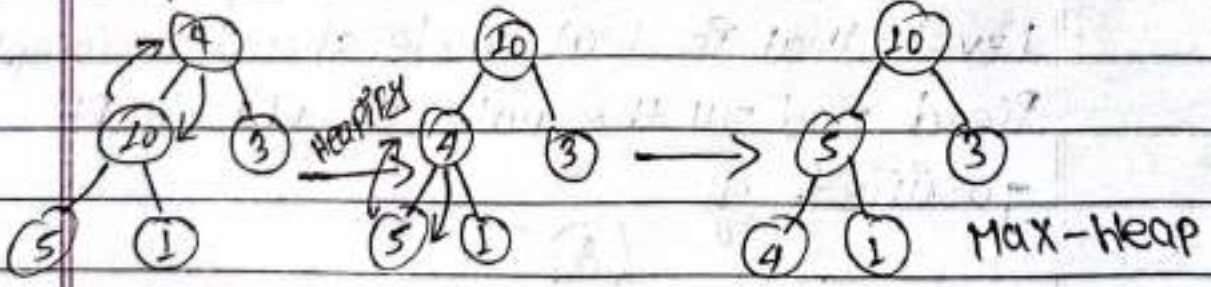
eg :- 4, 10, 3, 5, 1

Step-1 :- Construct Heap tree



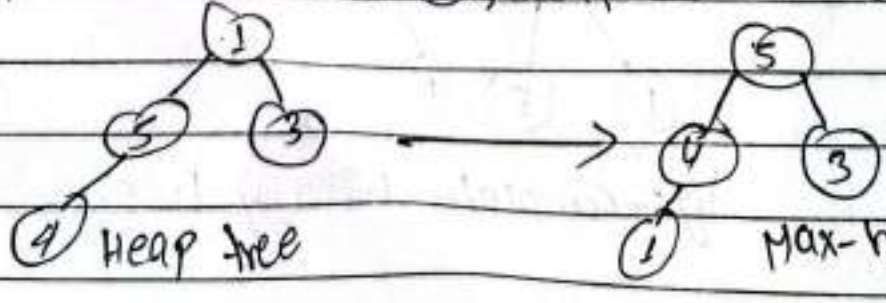
Heap tree

Step-2 :-



Max-Heap

Step-3 :- Delete root (10), swap 10 with 1

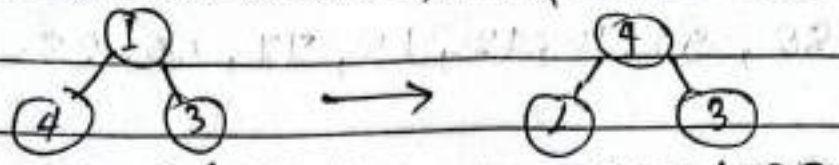


Heap tree
Max-heap

Sorted stack → LIFO

Date
Page

Step-4 :- Delete Root (5), swap with (1)



Heap tree → max tree

Step-5 :- Delete Root (4), swap with (3)

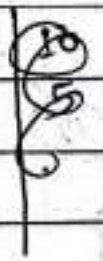
Heap tree → max tree

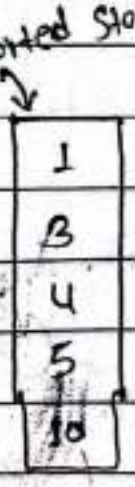
Step-6 :- Delete Root (3), swap with (1)

Heap tree → Max tree

Step-7 :- Delete Root (1).

Heap Tree empty.



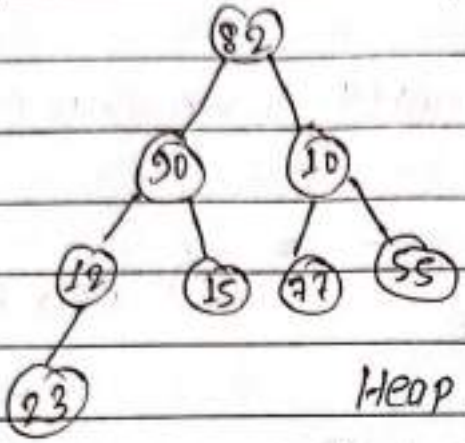


Sorted data :- 1, 3, 4, 5, 10

Q.1: Heap Tree Sort:

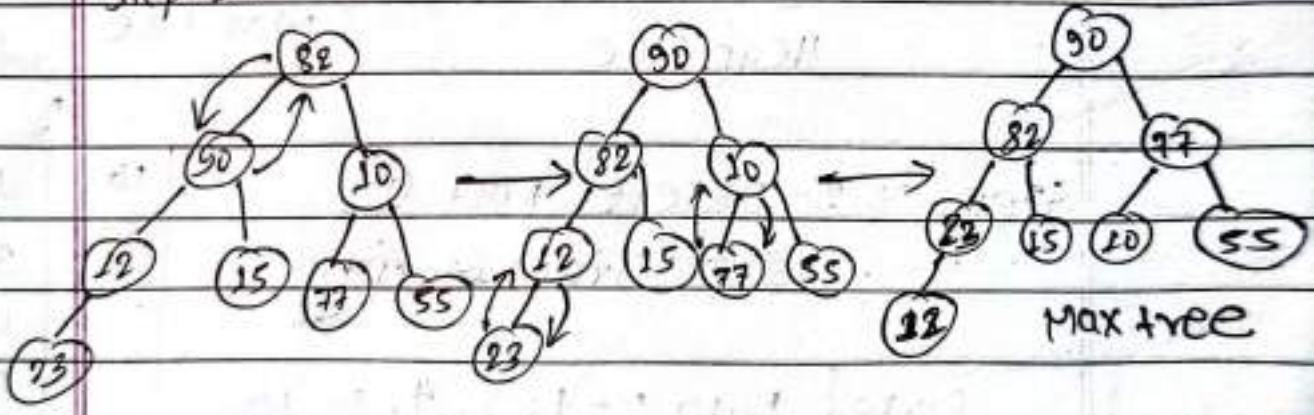
82, 90, 10, 12, 15, 77, 55, 23.

Step-1 :- Construct Heap tree



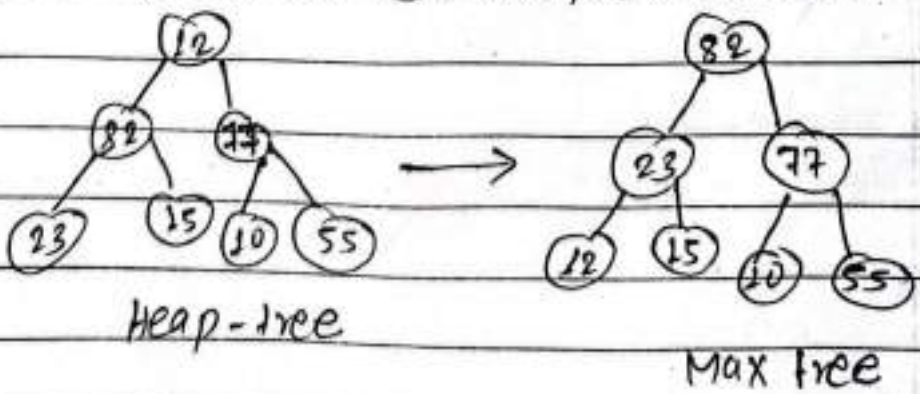
Heap tree

Step-2 :-



Max tree

Step-3 :- Delete root (90) Swap 90 with 12

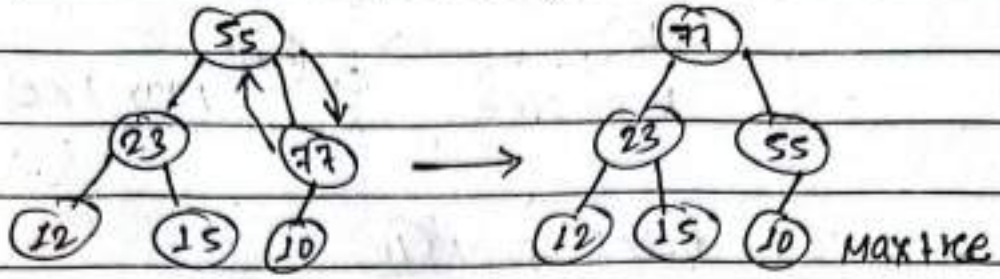


Heap-tree

Max tree

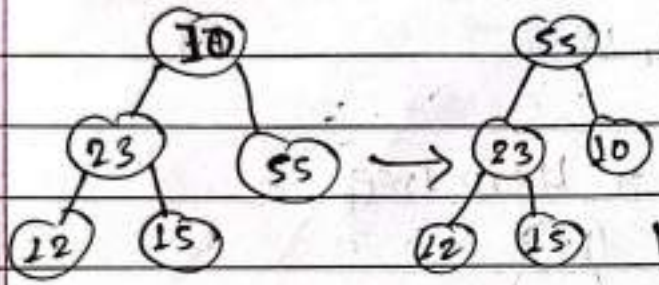
Date
Page

Step-4 :- Delete Root (82), swap with 55



Heap tree

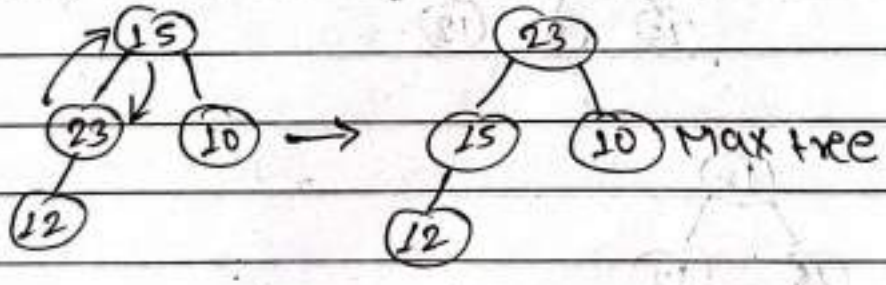
Step-5 :- Delete Root (77) swap ~~77~~ with 10



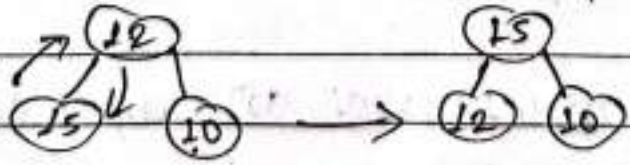
sorted stack

10
12
15
23
55
77
82
90

Step-6 :- Delete Root (55), swap 55 with 15

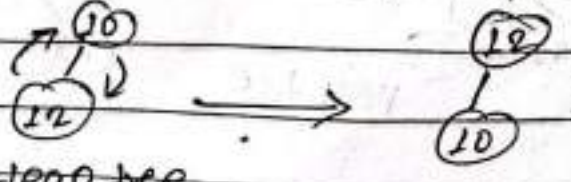


Step-7 :- Delete Root (23), swap (23) with 12



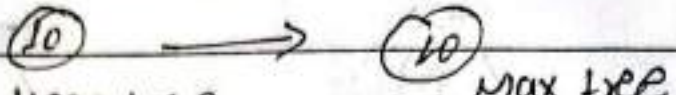
Heap tree

Step-8 :- Delete Root (15), swap 15 with (10)



Heap tree

Step-9 :- delete Root (12), swap 12 with 10.



Heap tree → Max tree

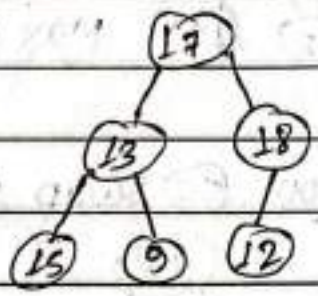
Step-10 :- Delete Root (10)

Heap tree empty.

∴ 10, 12, 15, 23, 55, 77, 82, 90 //

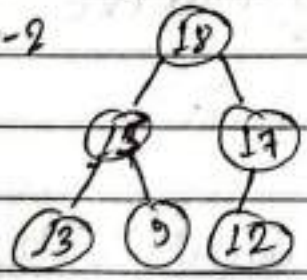
Q2. 17, 13, 18, 15, 9, 12

Step-1 Construct a Heap tree



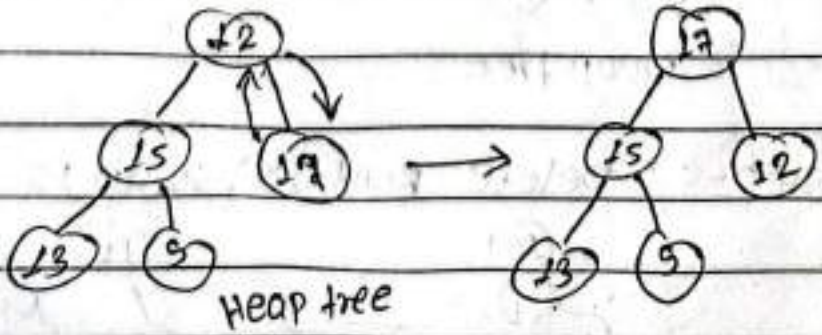
Heap tree

Step-2



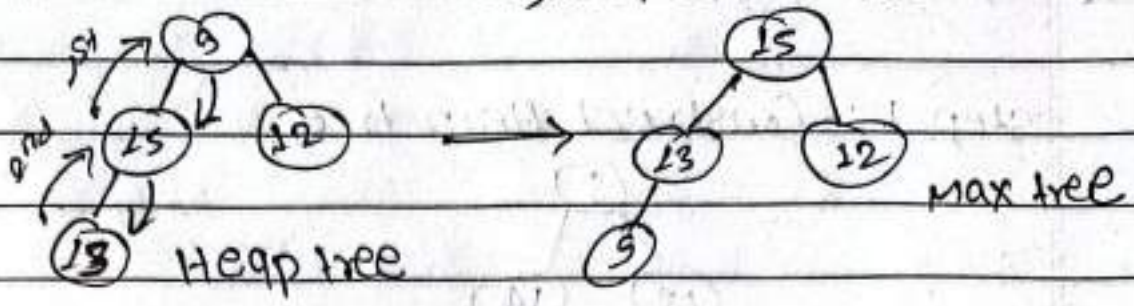
Max tree

Step-3 :- Delete root (18), swap (18) with (12)

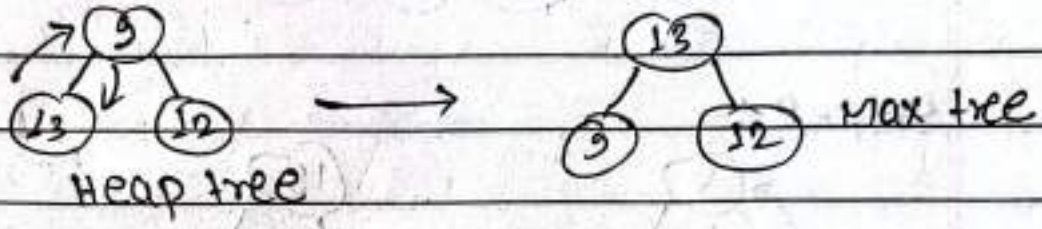


Heap tree

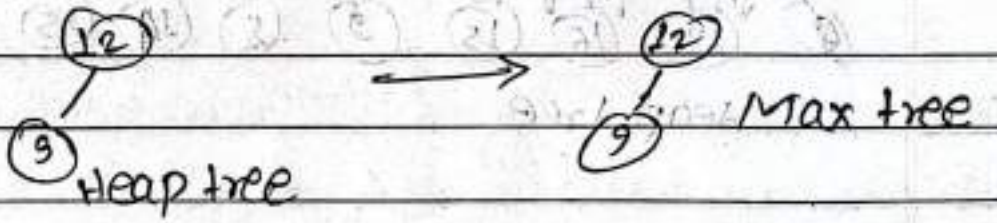
Step-4:- Delete Root (17), Swap (9) with 9.



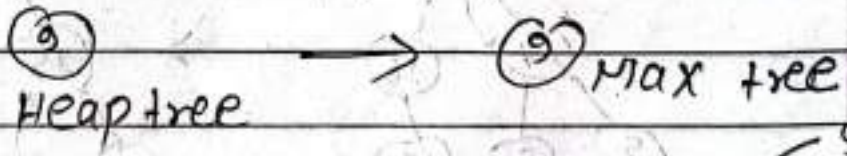
Step-5:- Delete Root (15), Swap (15) with (9).



Step-6:- Delete Root (13), Swap (13) with (12)



Step-7:- Delete Root (12), Swap (12) with (9)



Step-8:- Delete root (9)

Heap tree empty.

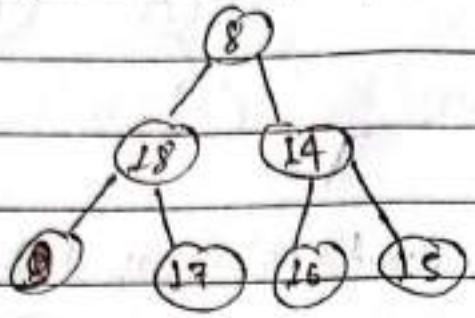
∴ 9, 12, 13, 15, 17, 18

Sorted stack.

9
12
13
15
17
18

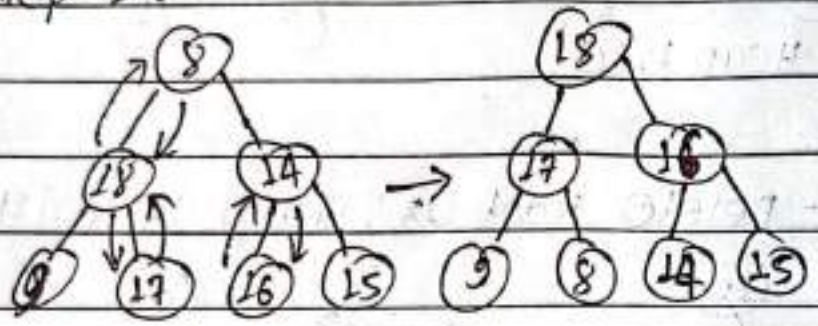
No.3 8, 18, 14, 9, 17, 16, 15

Step-1 :- Construct Heap tree.



Heap tree

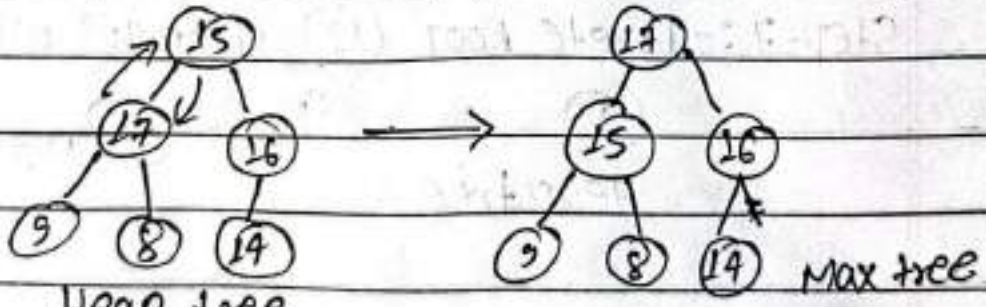
Step-2 :-



Maxtree

Heap tree

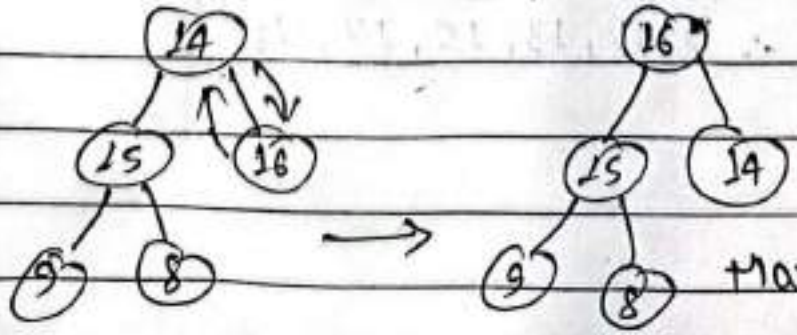
Step-3 :- Delete Root (18), swap (18) with (15).



Heap tree

Max tree

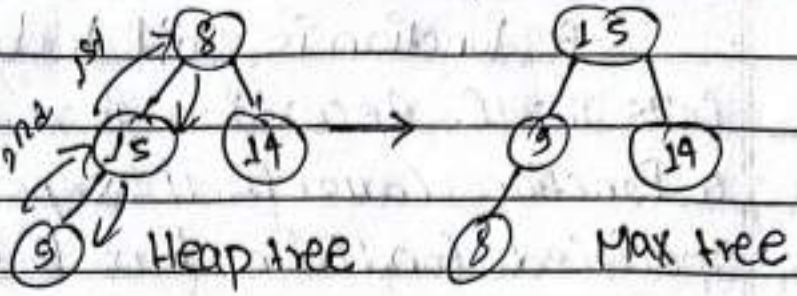
Step-4 :- Delete Root (17), swap (17) with (14).



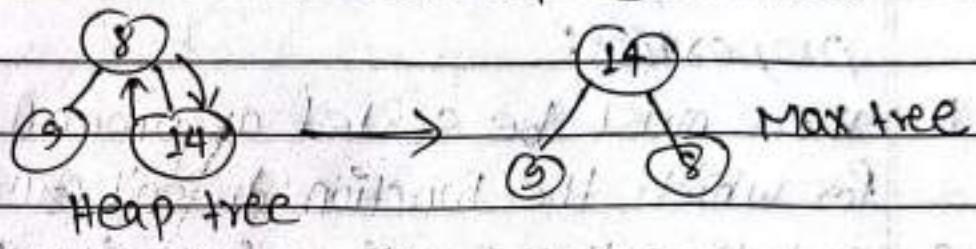
Heap tree

Max tree

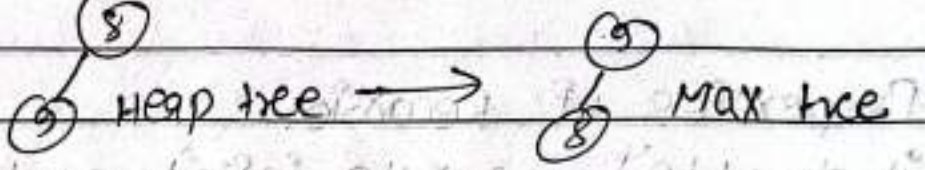
Step-5:- Delete Root (16), swap (16) with (8).



Step-6:- Delete Root (15), swap (15) with (8).



Step-7:- Delete Root (14), swap (14) with (8)

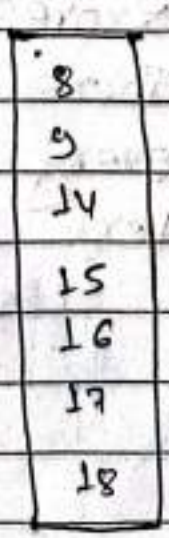


Step-8:- Delete Root (9), swap (9) with (8).

Step-9:- Delete Root (8)

Heap tree empty.

∴ 8, 9, 14, 15, 16, 17, 18. //



unit-4
Recursion

Date 2080/10/31
Page

* Recursion

A function is said to be recursive if it calls itself. Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

A Recursive Function should have two properties:

1. There must be sorted argument (called base value) for which the function doesn't call itself.
2. Each time the function calls itself, the argument must be closer to the base value.

4.1 * Properties of Recursion:

i) it shouldn't generate infinite number of calls on itself because such definitions will never terminate.

ii) Atleast one of the arguments shouldn't be expressed in terms of itself.

iii) There should be a terminating conditions in every recursive function which when satisfied text the control out of the function finally.

Date
Page

4.2 Recursion VS Iteration.

	Recursion	Iteration
(i.)	Recursion is a technique of defining anything in terms of itself.	It is a process of executing statements repeatedly until some specified condition is satisfied.
ii.)	There must be an "if" statement inside the recursive function to specify stopping condition.	Iteration involves four clear cut steps like initialization, condition, execution & updation.
iii.)	Not all problems have recursive solution.	Any recursive problem can be solved iteratively.
iv.)	Recursion is generally a worse ~~version~~ option ~~function~~ to for simple problem.	Iterative counter part of a problem is more efficient in terms of memory utilization and execution speed.
v.)	The Speed of ~~Iteration~~ recursive program is ~~faster than~~ slower ~~Recursive~~ because of stack overhead	The speed of iteration program is faster than Recursive program

4.3 V.V.I

Tower of Hanoi (TOH)

TOH has three towers, one is used as source tower, second one is used temporary tower and the thirst one is used as destination tower. TOH problem is moved to disk (plate) from one tower to another using a temporary tower.

Suppose we have a Source tower A, which has finite number of disk and these disk is placed on it in a decreasing order that is largest disk is at the bottom and the smallest disk is at the top. Now we have to place all these disks on destination tower C in the same order. We can use a temporary tower B to placed the disk temporarily ~~tower B~~ when ever required. Following are the rules to be followed during transfered:

i) At any point of transformation of disk, largest disk cannot be placed on smaller disc.

ii) Only one disk may be moved at a time.

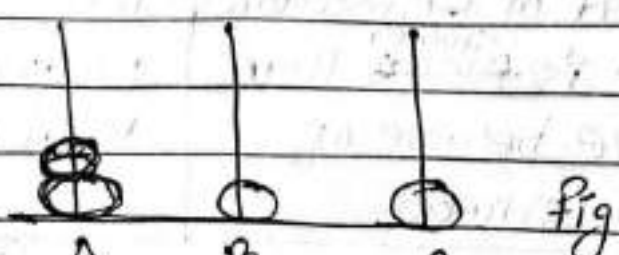


fig - TOH

Date
Page

The Recursive function for solving ToH problem can be define as;

$$Toh(n, Sorce, temp, dest) = \begin{cases} \text{Move disk 1 from Source to dest} & (\text{if } n=1) \\ toh(n-1, sorce, dest, temp) & (\text{if } n>1) \\ \text{Move } n^{th} \text{ disk from Soure to dest} & \\ toh(n-1, temp, Sorce, dest) & \end{cases}$$

* Algorithm for ToH.

To move n disk from tower A to C using B as temporary tower, the general solution can be written as:

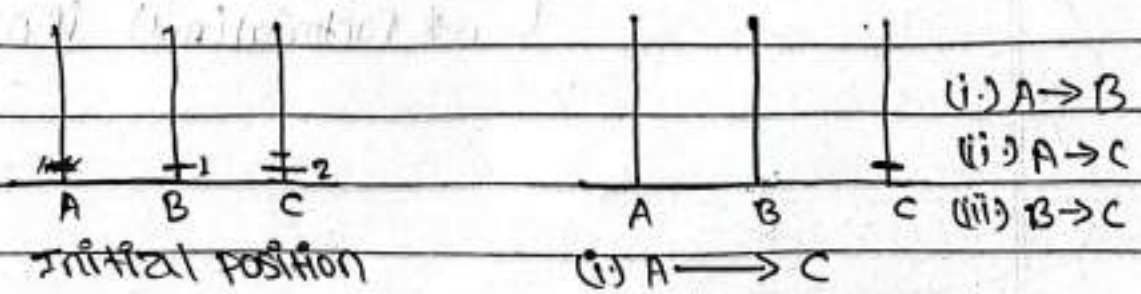
(i.) if n=1 Move the single disk from A to C and stop.

(ii.) Move upper n-1 disks from A to B using C as the temporary tower.

(iii.) Move n^{th} disk from A to C.

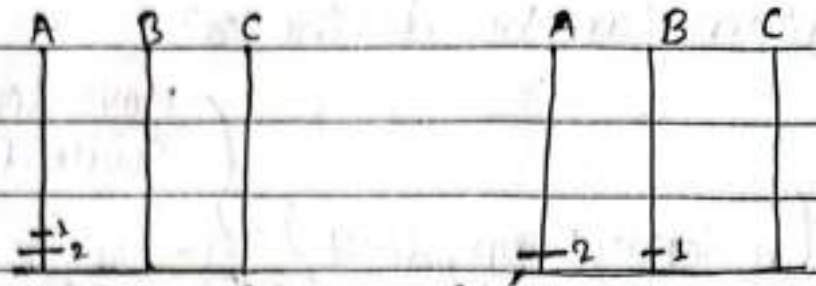
(iv.) Move n-1 disk from B to C using A as the temporary tower.

Eg:- for n=1 [i.e. no of disk on source tower A is one].

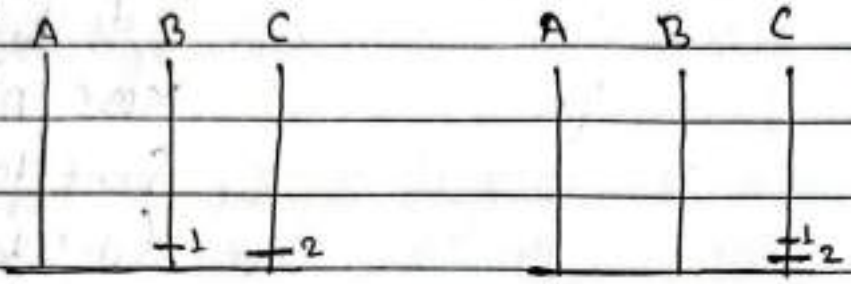


(i.) Move the disk from tower A to tower C. (A→C).

for n=2 [i.e. no of disks on source tower A is two].



(a) Initial position (i) A → B



(ii) A → C (iii) B → C

(i) Move disk 1 from A to B
(ii) Move disk 2 from A to C
(iii) Move disk 1 from B to C

08/09/18

4.4 Factorial

The factorial of a number is the product of the integer values from 1 to the number, recursively the factorial algorithm can be define as below:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n * \text{factorial}(n-1) & \text{if } n>0 \end{cases}$$

Date Page

~~program~~

```
#include <stdio.h>
#include <conio.h>
long int fact(int n)
{
  if(n=0)
    return 1;
  else
    return(n * fact(n-1));
}
void main()
{
  int x, y;
  printf("\n Enter an integer value");
  scanf("%d", &x);
  y = fact(n)
  printf("\n factorial = %d", y);
  getch();
}
```

The below figure illustrate the recursive concept of factorial of 3.

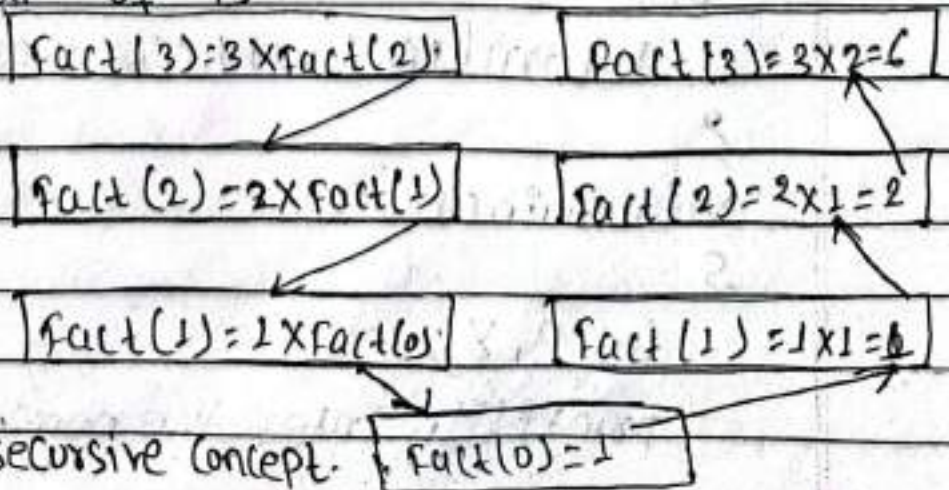


fig:- fact(3) recursive concept.

Page

*Fibonacci Numbers [Leonardo Fibonnacci]

Fibonacci Numbers are a series in which each number is the sum of the previous two numbers. The first few number in fibonacci series are 0, 1, 1, 2, 3, 5, 8, 13..... where, 0 & 1 is the base value of fibonacci series.

Recursively the fibonacci algorith can be defined as:

$$Fib(n) = \begin{cases} 1 & \text{if } n <= 1 \\ Fib(n-1) + Fib(n-2) & \text{if } n > 1 \end{cases}$$

Program

```
#include <stdio.h>
#include <conio.h>
int fib (int n)
{
  if (n<=1)
    return(n);
  else
    return(fib(n-1)+fib(n-2));
}
void main()
{
  int n, x;
  printf("\n Enter the number of terms");
```

Date
Page

```
scanf("%d", &n);
for(i=0; i<n; i++)
{
    x=fib(i);
    printf("%d", x);
}
getch();
}
```

Unit-8
Graph

Date 2080/09/18
Page

* Graph

Graph is a non-hexixichal data structure that consist of nodes and set of edges that establish relationships (connections) between the nodes. ox,

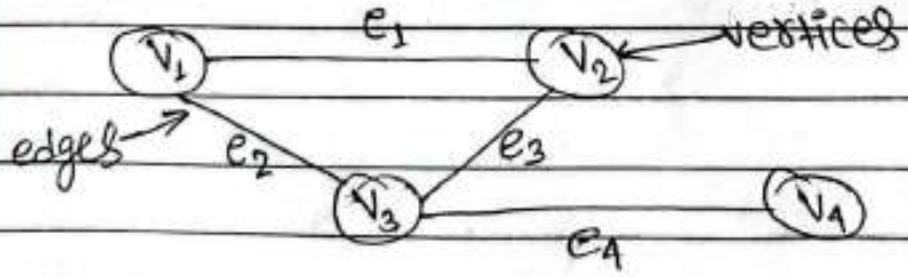
A graph is a collection of nodes called vertices and line segment (edge ox axc) that connect pairs of nodes.

ox,

A graph 'G' is defined as set of (V, E) where, V is a finite and non-empty set of vertices and
E is the set of edges.

* Components of Graph.

The below figure illustrates graph with it's edges and vertices component.



The two components are :-

* Vertex :- Vertex is the point ox node used to store the data elements. In the above graph the vertex 'V' can be represented as

$$V = \{V_1, V_2, V_3, V_4\}$$

* Edge :- Edge is a line or arc used to connect vertices in the above graph the edge 'E' can be represented as

$$E = \{e_1, e_2, e_3, e_4\}$$
$$= \{(v_1+v_2), (v_1+v_3), (v_2+v_3), (v_3+v_4)\}$$

* Types of Graph

A graph can be of Two Types:-

(i) Directed Graph [Digraph]

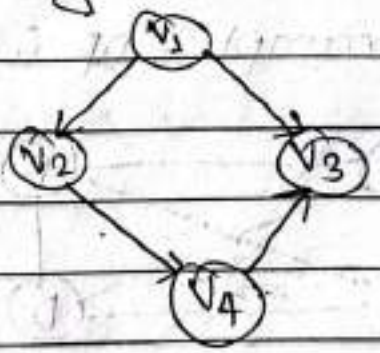
(ii) UnDirected Graph

(i) Directed Graph

A Directed Graph is a graph which has ordered pair of vertices (U,V). In Directed Graph, A direction is associated with each edge that means (U,V) is not equal to (V,U).

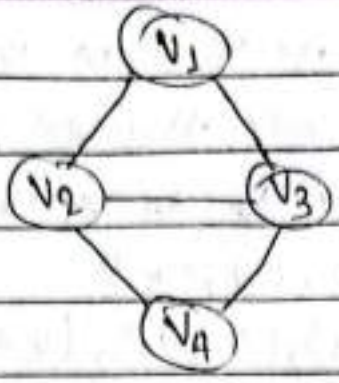
The below figure illustrate the directed

(ii) UnD Graph.



(ii) UnDirected Graph

UnDirected Graph is a graph which has unordered pair of vertes. If there is an edge between vertices (U & V) then it can be represented as either (U,V) or (V,U) in undirected graph. The below figure illustrate the concept of undirected graph.



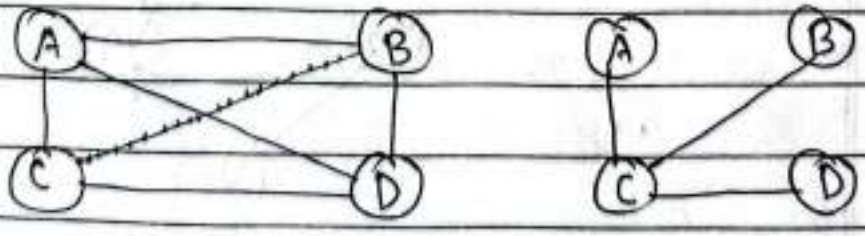
* Undirected Graph Connectivity.

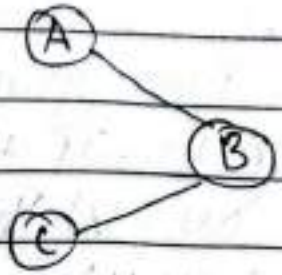
There are two types of Connectivity of Undirected graph.

(i.) Connected graph.
(ii.) Disconnected graph

(i.) Connected graph

An undirected graph is said to be Connected if their exist a path from any vertex to any other vertex. A connected graph of "N" vertices has at least "N-1" edges. Examples of connected graphs are

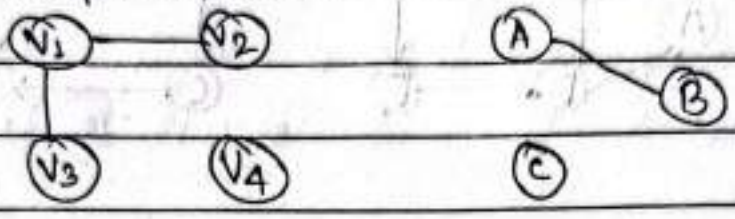




(ii.) Disconnected graph (unconnected graph)

An undirected graph is disconnected graph if there doesnot exist at least one path from any vertex to any other vertex.

Examples are :-



* Directed graph Connectivity.

There are two types of connectivity of directed graph.

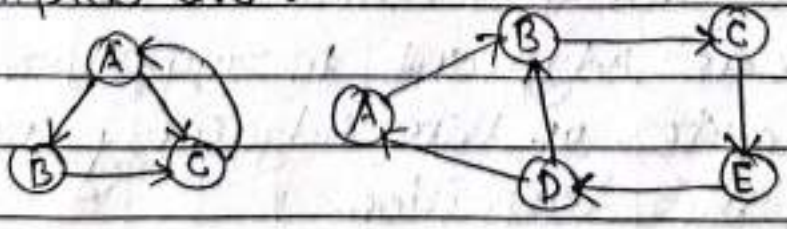
(i) Strongly connected graph.

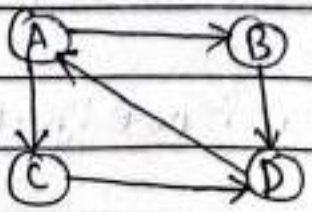
(ii) Weakly connected graph.

(i.) Strongly connected graph.

A directed graph is strongly connected if there is a directed path from any vertex of graph to any other vertex.

Examples are :-

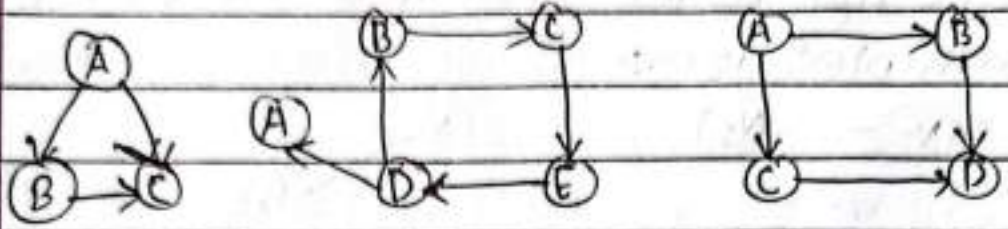




Date
Page

(ii.) Weakly Connected Graph.

A Directed graph is weakly connected is at least two vertices are not connected



* Representation of Graph.

There are two main standard ways of representing a graph in the memory of a computer.

(i.) Sequential representation using Adjacency Matrix.

(ii.) Link representation using linked list of Adjacency list.

(i.) Adjacency Matrix

Adjacency Matrix is a Matrix that maintains the information of adjacent vertices. That means this Matrix tells us whether a vertex is Adjacent to any other vertex or not. The entries of this Adjacency Matrix are fixed using this definition.

$$A(i,j) = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j. \\ 0 & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

Date
Page

In this method the size of the graph must be known before the program starts.
for example :-

(a) for Directed graph :-

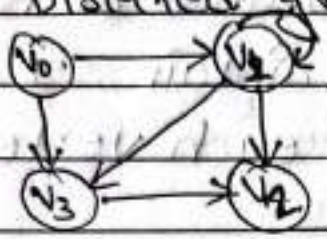


Fig :- Directed graph

The above directed graph can be represented in Adjacency Matrix form as below :-

	V_0	V_1	V_2	V_3
V_0	0	1	0	1
V_1	0	1	1	1
V_2	0	0	0	0
V_3	0	0	1	0

(b) for UnDirected graph :-

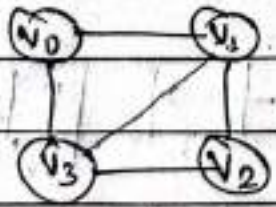


Fig :- UnDirected graph

	V_0	V_1	V_2	V_3
V_0	0	1	0	1
V_1	1	0	1	1
V_2	0	1	0	1
V_3	1	1	1	0

The above undirected graph can be represented in Adjacency Matrix form as above :-

ii.) Adjacency list

In Adjacency list method, we maintain two linked list. The first linked list is the vertex list that keeps track off all the vertices in the graph and second linked list is the edge list that maintains a list of adjacent vertices for each vertex.

The following figure shown a directed graph and its adjacency list structure.

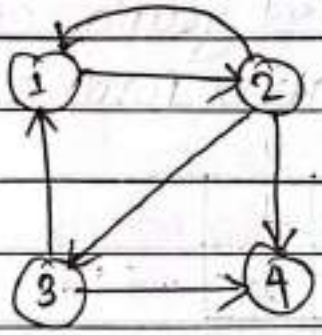
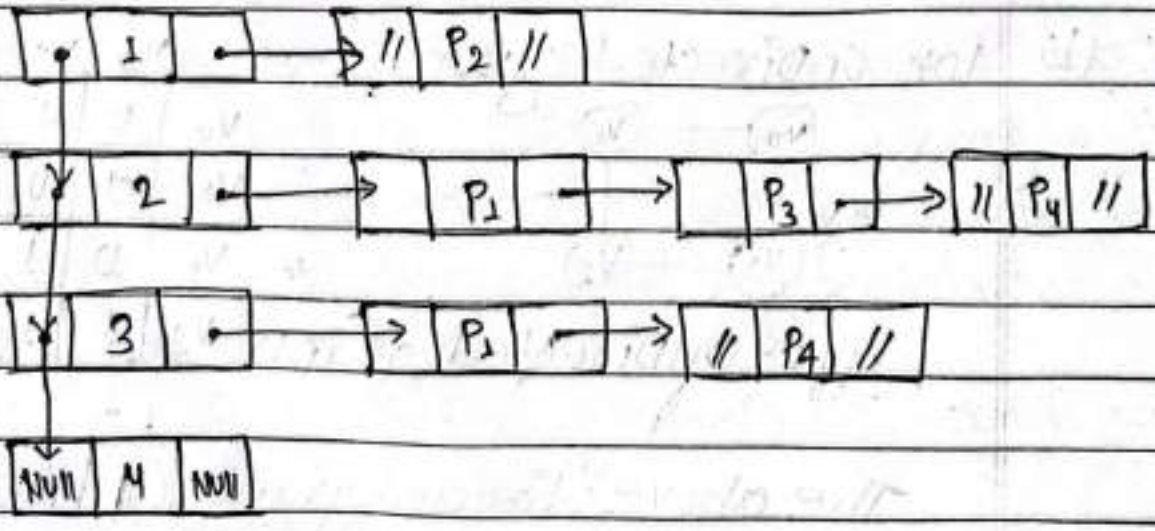


Fig:- Directed graph



Date
Page

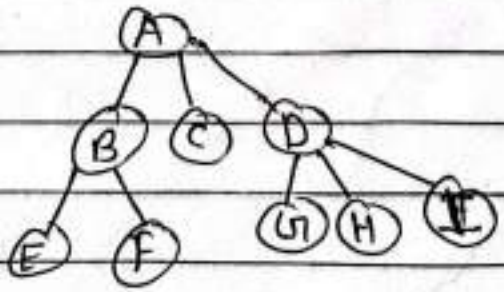
* Graph Traversal
1.) Depth first search / Traversal [DFS/T]
2.) Breadth first Search / Traversal [BFS/T]

1.) Depth First Search / Traversal [DFS/T]

The idea of this algorith is to make a path as long as possible and then go back to add branches also as long as possible.

In DFS, we process all of a vertex descendents before we move to an adjacent vertex. In this method, we first visit the starting vertex and then pick up any path that starts from the starting vertex and all the vertices in the path till we reach a data ending. It is implemented using Stack.

For example, let us take a graph and perform a depth first search traversal taking 'A' as starting vertex.



DFS :- A, B, E, F, D, G, H, I, C
A, C, B, F, E, D, H, G, H.

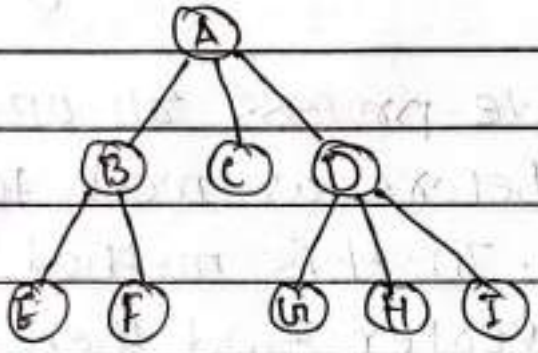
Date
Page

(2) Breadth first-Search (BFS) Traversal:

In this technique, the starting vertex is visiting first and then visit all the vertices adjacent to the starting vertex. After this way pick this adjacent vertices. One by one at visit these adjacent vertices and this process goes on.

BFS is implemented by using queue.

For example, the BFS Traversal of the below graph is written as



BFS :- A, B, C, D, E, F, G, H, I

Unit-7
Search

Date
Page

7.4) **Hashing :-**

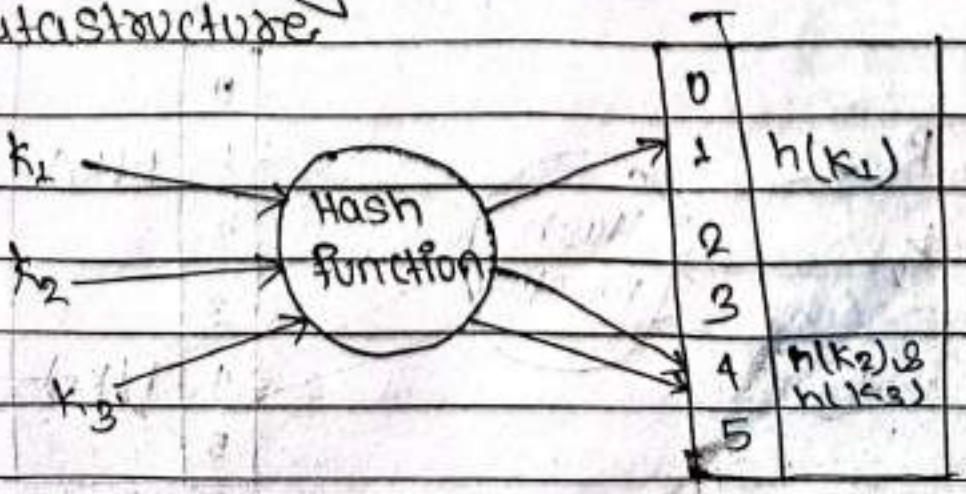
Hash is a datastructure that allow the storage and retrieval of data in an average time which does not depend at on the collection size. So Hashing is a method to store data in a way that storing, searching, Inserting and deleting data in fast.

7.4.1) **Hash Table,**

It is a data structure in which the location of a data item is determined directly as a function of the data item itself rather than by a sequence of comparisions.

A Hash Table is made up of two parts: an array (where actual data is stored) and a mapping function (Hash function).

The below figure illustrate the has table datastructure.



Date
Page

7.4.3) Hash collision or Hash clash

A Hash collision occurs when the Has function generates the same address for differe keys.

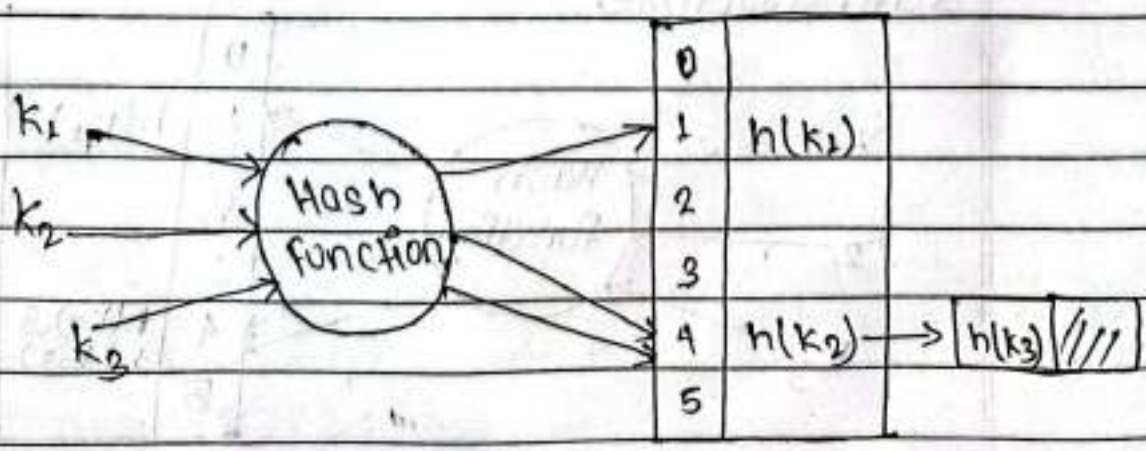
7.4.4) Techniques to solve Hash Collision.

A technique use to resolve the collision generated by a Hash function is called collision resolution Techniques. The possible collision resolutions & Techniques are as follows:

(1) chaining
(2) open addressing
(3) Re-hashing

(1) chaining Collision

It builts a linked list of all item whose keys has to the same values. In this method, linked lists are maintained for keys that have same Hash address.



Date
Page

Example:-

Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, to into a chained Hash table. Let us suppose that Has table has ~~mind~~ 9 ~~slots~~ slots and the Has function be "h(k) = k mod 9"

Soln

We create a chained table with 9 slots.

0	NULL
1	NULL → 28
2	///
3	///
4	///
5	/// → 5 ///
6	///
7	///
8	///

Fig:- Initial Step of chained Hash Table.

Insert (5):-
h(5) = 5 mode 9 ⇒ 5
Create a linked list at slot 5 and insert it at the linked list

Insert (28):-
h(28) = 28 mode 9 ⇒ 1

Insert (19):-
h(19) = 19 mode 9 ⇒ 1

Insert (15):-
h(15) = 15 mode 9 ⇒ 6

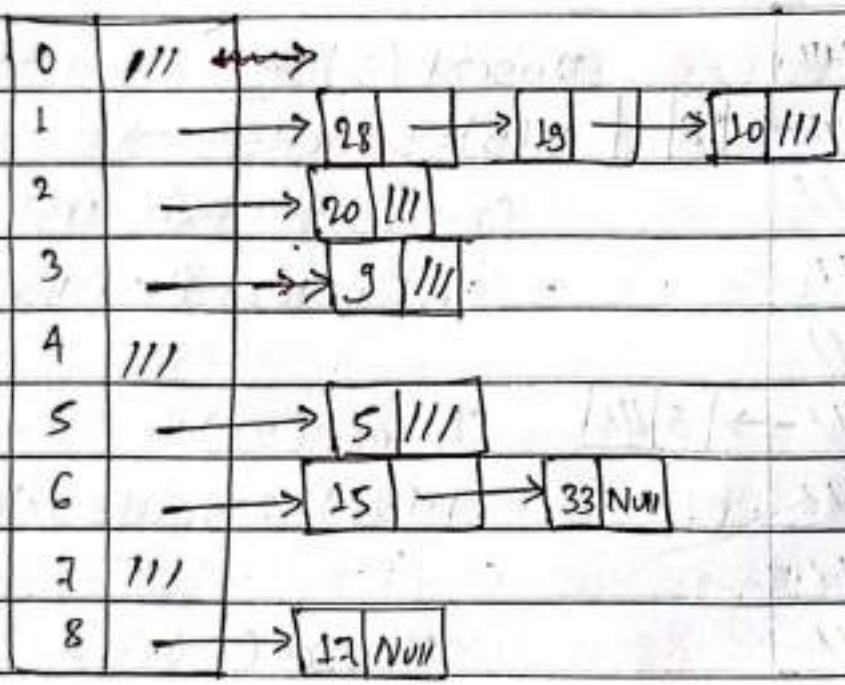
Insert (20):-
h(20) = 20 mode 9 ⇒ 2

Insert (33):-
h(33) = 33 mode 9 ⇒ 6

Insert (12):-
h(12)=12 mode 9 => 3

Insert (17):-
h(17)=17 mode 9 =>8

Insert (10):-
h(10)=10 mode 9 => 1



(2) Open Addressing

Open Addressing is a method of collision resolution in which a hash collision is resolved by "probing" or searching until either the target record is found or an unused array slot is found. Some of the well known probe sequences are:

(a) linear probing
(b) Quadratic probing
(c) Double Hashing

Date
Page

(9.) Linear probing

Linear probing uses the following Hash functions.

$$h(k,i) = [h'(k)+i] \bmod m \text{ for } i = 0,1,2,\cdots m-1$$

where,

m → is the size of Hash table.

$h'(k) = k \bmod m$

i = is the probe number.

Example :-

Consider inserting the keys 76, 26, 37, 59, 21, 65, 88 into a Hash table of size m=11 using linear probing. further Consider that the primary Hash function is $h'(k) = k \bmod m$.

Soln

	0	1	2	3	4	5	6	7	8	9	10
T	21	65			26	37	59		88		76

Fig:- Initial stage of the Hash Table.

As we know,

for linear probing :- $h(k,i) = [h'(k)+i] \bmod m$

We have $h'(k) = k \bmod m$

and m=11

Insert (76)

$h(76+0) = [76 \bmod 11 + 0] \bmod 11$

k i

$= 10$

11) 76 (6
66
10

Insert (26) :-
$h(26, 0) = [26 \bmod 11 + 0] \bmod 11$
$= 4$

Insert (37) :-
$h(37, 0) = [37 \bmod 11 + 0] \bmod 11$
$= 4$
Since, slot T[4] is occupied, the next prob sequence is
~~Insert (59) :-~~ Calculated as
$h(37, 1) = [37 \bmod 11 + 1] \bmod 11 = 5$

Insert (59) :-
$h(59, 0) = [59 \bmod 11 + 0] \bmod 11 = 4$
Since, slot T[4] is occupied, the next prob Sequence is calculated as
$h(59, 1) = [59 \bmod 11 + 1] \bmod 11 = 5$
Again, slot T[5] is occupied, the next prob sequence is calculated as
$h(59, 2) = [59 \bmod 11 + 2] \bmod 11 = 6$

Insert (21) :-
$h(21, 0) = [21 \bmod 11 + 0] \bmod 11 = 10$
Since, slot T[10] is occupied, the next prob sequence is calculated as
$h(21, 1) = [21 \bmod 11 + 1] \bmod 11 = 0$

Insert (65):-

$h(65, 0) = [65 \bmod 11 + 0] \bmod 11 = 10$

Since, slot T[10] is occupied, the next prob Sequence is calculated as

$h(65, 1) = [65 \bmod 11 + 1] \bmod 11 = 0$

Since, slot T[0] is occupied, the next prob Sequence is calculated as

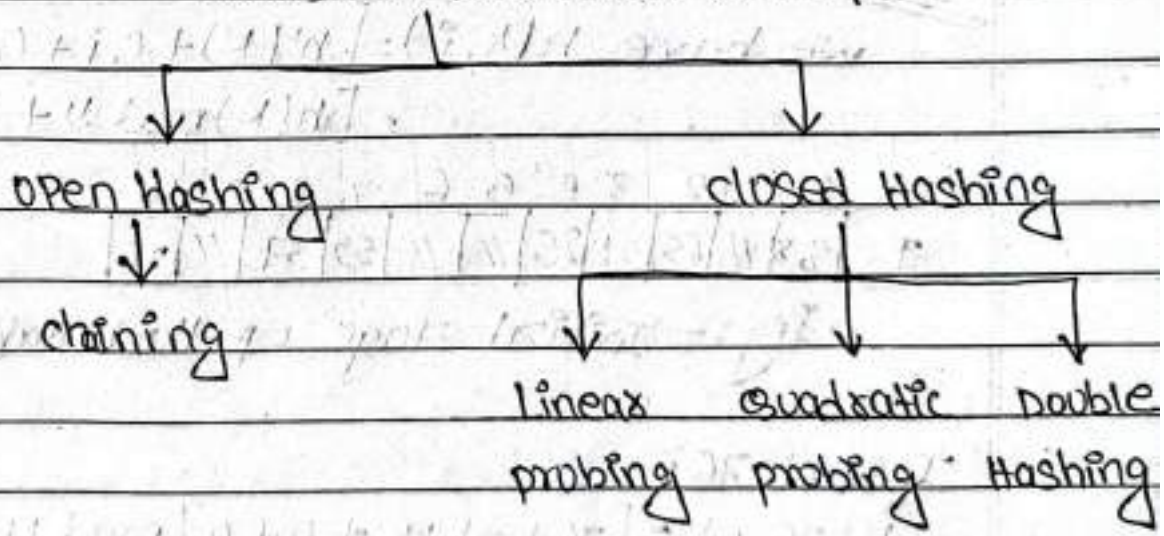
$h(65, 2) = [65 \bmod 11 + 2] \bmod 11 = 1$

Insert (88):-

$h(88, 0) = [88 \bmod 11 + 0] \bmod 11 = 8$

08/09/23 This is required linear probing.

7.9.9) Collision Resolution Technique



(b.) Quadratic probing

In quadratic probing, Instead of probing the next slot linearly, the algorithm uses a quadratic function to determine the next prob position. The probing sequence follows a quadratic patten until an empty slot is form

The quadratic probing uses the following Hash function.

$$h(k,i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$
for $i = 0, 1, 2, \ldots, m-1$.

Where, c_1 & $c_2 \neq 0$, are auxuliary constants.

Example:-

Consider Inserting the keys 76, 26, 37, 59, 21, 65, 88 into a Hash table of size M=11 using quadratic probing with $c_1=1$ & $c_2=3$. Further consider that the primary Hash function is "$h'(k) = k \bmod m$".

Soln

we have $h(k,i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$
$= [h(k) \bmod 11 + i + 3i^2] \bmod 11$

	0	1	2	3	4	5	6	7	8	9	10
T	88	//	65	21	26	//	//	59	37	//	76

Fig:- Initial stage of Hash Table.

Insert 76:-

$h(76,0) = [76 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 10$

Since, slot T[10] is free, insert 76 at slot 10.

Insert 26:-

$h(26,0) = [26 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 4$

Since, slot T[4] is free, Insert 26 at slot 4.

Insert 37:-

$h(37,0) = [37 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 4$

Since, T[4] is occupied, the next prob sequence

is computed as

$h(37,1) = [37 \bmod 11 + 1 + 3] \bmod 11 \Rightarrow 8$

Insert 59 :-

$h(59,0) = [59 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 4$

Since, T[4] is occupied, the next prob sequence is computed as

$h(59,1) = [59 \bmod 11 + 1 + 3] \bmod 11 \Rightarrow 8$

Now, T[8] is occupied, the next prob sequence is computed as

$h(59,2) = [59 \bmod 11 + 2 + 12] \bmod 11 \Rightarrow 7$

Insert 21 :-

$h(21,0) = [21 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 10$

Since, T[10] is occupied, the next prob sequence is computed as

$h(21,1) = [21 \bmod 11 + 1 + 3] \bmod 11 \Rightarrow 3$

Insert 65 :-

$h(65,0) = [65 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 10$

Since, T[10] is occupied, the next prob sequence is computed as

$h(65,1) = [65 \bmod 11 + 1 + 3] \bmod 11 \Rightarrow 3$

Since, T[3] is occupied, the next prob sequence is computed as

$h(65,2) = [65 \bmod 11 + 2 + 12] \bmod 11 \Rightarrow 2$

Insert 88 :-

$h(88,0) = [88 \bmod 11 + 0 + 0] \bmod 11 \Rightarrow 0$

c.) Double Hashing

it uses two has function to determine the prob sequence. it occupies the second Hash function to calculate and offset which is then used to find the next slot to prob. double Hashing uses a has function of the form

$$h(k,i) = [h_1 k + i h_2(k)] \bmod m$$
$$\text{for } i = 0, 1, 2, \ldots m-1$$

where, $h_1(k) = k \bmod m$

$h_2(k) = k \bmod m'$

and m' is lightly less than m.

Example:-

Considered inserting the keys 76, 26, 37, 59, 21, 65, 88 in a Hash table of size $m = 11$ using double Hashing. Forther Consider that the Auxuliary Hash function are :-

"$h_1(k) = k \bmod 11$" and "$h_2(k) = k \bmod 9$"

Soln

	0	1	2	3	4	5	6	7	8	9	10
T	88	65	21		26	37				59	76

We have, $h(k,i) = [(k \bmod 11) + i(k \bmod 9)] \bmod 11$

Date
Page

Insert 76:-

$h(76, 0) = [(76 \bmod 11) + 0] \bmod 11$

$= 10$

Insert 26:-

$h(26, 0) = [(26 \bmod 11) + 0] \bmod 11 = 4$

Insert 37:-

$h(37, 0) = [(37 \bmod 11) + 0] \bmod 11 \Rightarrow 4$

Since, T[4] is occupied, the next prob sequence is computed as

$h(37, 1) = [(37 \bmod 11) + (37 \bmod 9)] \bmod 11$

$= 5$

Insert 59:-

$h(59, 0) = [(59 \bmod 11) + 0] \bmod 11 \Rightarrow 4$

Since, T[4] is occupied, the next prob sequence is computed as

$h(59, 1) = [(59 \bmod 11) + (59 \bmod 9)] \bmod 11 \Rightarrow 9$

Insert 21:-

$h(21, 0) = [(21 \bmod 11) + 0] \bmod 11 \Rightarrow 10$

Since, T[10] is occupied, the nex prob sequence is computed as

$h(21, 1) = [(21 \bmod 11) + (21 \bmod 9)] \bmod 11 \Rightarrow 2$

Inserted 65 :-

$h(65,0) = [(65 \bmod 11) + 0] \bmod 11 \Rightarrow 10$

Since, T[10] is occupied, the next prob sequence is computed as

$h(65,1) = [(65 \bmod 11) + (65 \bmod 9)] \bmod 11 \Rightarrow 1$

Insert 88 :-

$h(88,0) = [(88 \bmod 11) + 0] \bmod 11 \Rightarrow 0$

Unit-7
Searching

Date 2080/09/24
Page

* Searching

Searching is the process of finding the presence of desired data items in the list of data items.

Basically there are two types of searching
(i) Sequential Search
(ii) Binary Search

(i) Sequential Search (Linear Search)

Sequential Search is a very simple Search algorithm. In this type of Search, we start searching at the beginning of a list of items and looks at each turn until we either found the required data or reach at end of list.

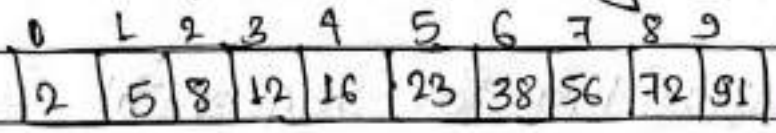
Algorithm for Sequential Searching.

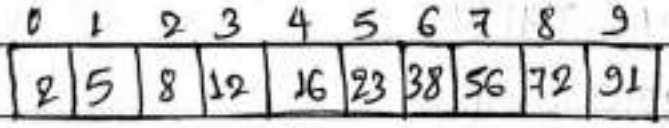
1. Set I = 1
2. If I > N THEN
   goto Step 7
3. IF A[I] = k Then
   goto Step 6
4. Set I = I + 1
5. Go to Step 2
6. print value k is found at index I and goto Step 8.
7. print value k is not found.
8. Exit.

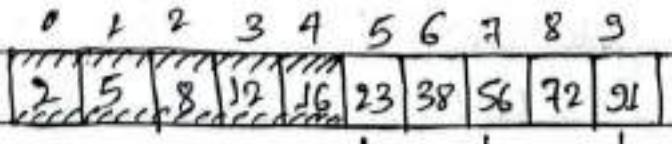
(ii) Binary Search

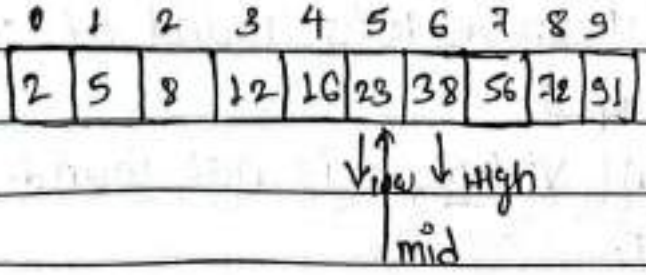
This search algorithm works on the principle of devide and conquer. For this algorith, the data collection should be in the sorted form.

Binary Seach looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. otherwise, the item is searched in the sub-array to the right of the middle item. This process continuous on the sub-array as well until the size of the sub-array reduces to zero









Date 2080/9/25
Page

* Spanning Tree

A spanning Tree is the sub-set of graph in which has all the vertices covered with minimum possible number of edges. Hence a spanning tree doesnot have cycles and cannot be disconnected.

Example :-

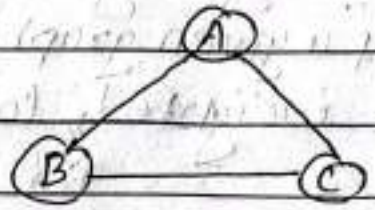


fig :- Graph

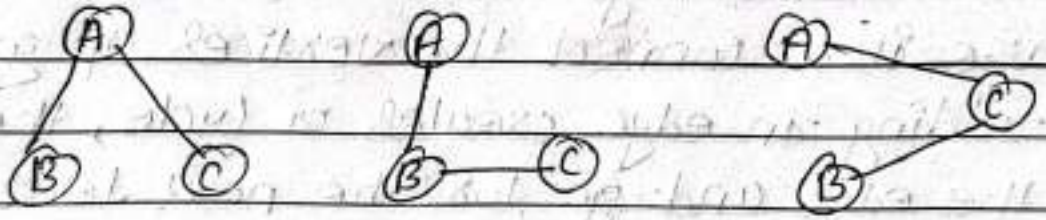


fig :- Spanning Trees of above graph.

* Minimum Spanning Tree (MST)

In a weighted graph, A minimum (MST) is a spanning Tree that has minimum weight than all other spanning tree of the same graph.

* MST Algorithm

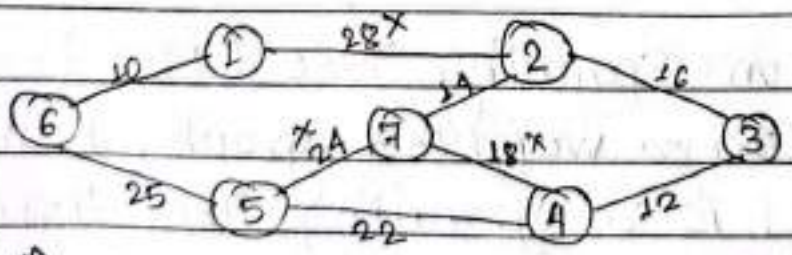
1. Kruskal's Algorithm
2. Prim's Algorith

1. Kruskal's Algorithm

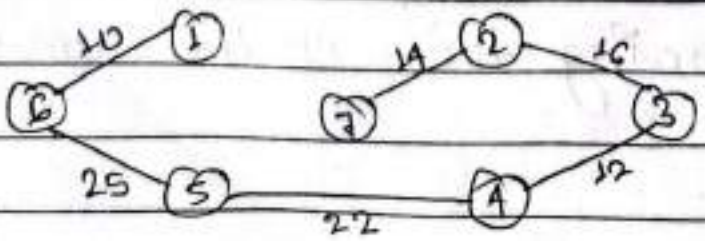
This is a famous greedy algorithm it is used for finding the MST of a given graph. It is applied to the graph which most be weighted, connected and undirected

Steps :-

1. Sort all the arrays from low wait to high weight
2. Take the arrays with the lowest weight and use it to connect the vertices of graph. If adding an edge creates a cycle, then reject the edge and go for the next least weight edge.
3. Keep adding edges until all the vertices are connected and a MST is obtained.



Soln



Weight of MST = 10 + 25 + 22 + 12 + 16 + 14 = 99 units

Date
Page

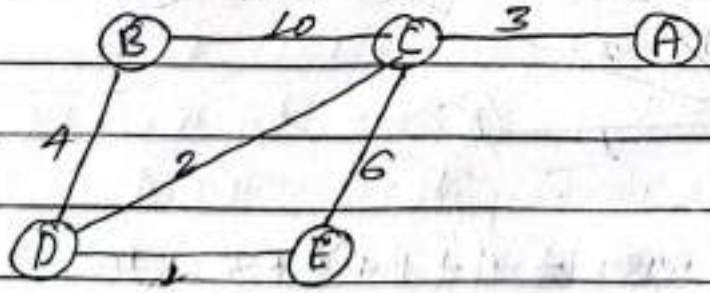
(2) prim's Algorithm

prim's Algorith is use to find the MST. it takes a graph as input and finds the sub-set of the edges of that graph which:

(1.) form a tree that includes every vertex.
(2.) Has the minimum some of weights among all the trees that can be formed from the graph

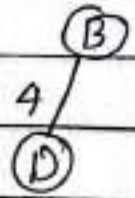
Steps :-

(a.) Initialized the minimum spanning tree with a vertex choosen at random:
(b.) find all the edges that connect the tree to new vertices, find the minimum and it to the tree.
(c.) keep repeating 'step-2' until we get a minimum spanning tree.

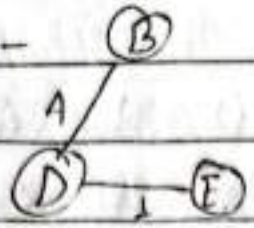


Step-1 :- B is choosen as random vertex for MST.

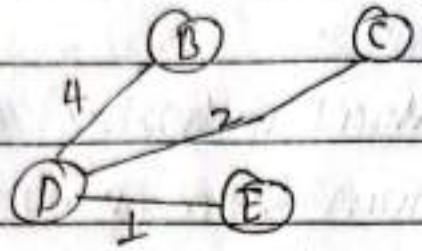
Step-2 :-



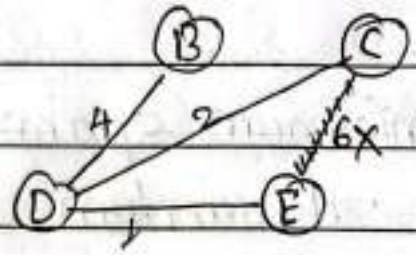
Step-3 :-



Step-4 :-

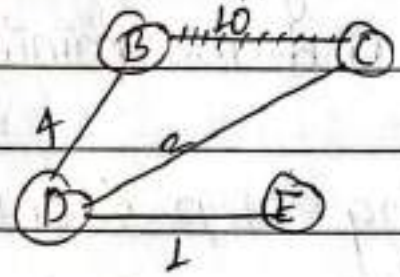


Step-5 :-

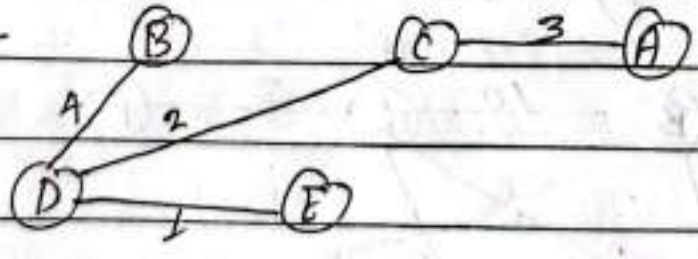


x this vertex is regected

Step-6 :-



Step-7 :-



Weight of MST = 9+1+2+3 = 10

Unit - 1

Date
Page

1.3.) Algorithm Analysis

Algorithm Analysis is an important part of Computational Complexity theory. It is use to determine the amount of time and space required to execute it. The analysis of algorithm is based on time Complexity and space Complexity.

(i.) Time Complexity
(ii.) Space Complexity

(i.) Time Complexity

It is the required time of the algorithm to solve the given problem. To estimate the the Time Complexity we need to consider the cost of each fundamental instruction and the number of time the instruction is executed.

(ii.) Space Complexity

problem Solving Using Computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory require by the algorithm to solve given problem is called space Complexity of the algorithm.

Date 2080-05-26
Page

* Asymptotic Notations :-

Asymptotic Notation of an algorithm is an mathematical representation of its time Complexity. In this Notation, we use only the most significant times in the complexity of that algorithm.

The time complexity is represented by three ways :-

(i) Big-oh (O)
(ii) Big-omega (Ω)
(iii) Big-Thita (Θ)

(i) Big-oh (O) :-

Big-oh Notation is used to define the upper bound of an algorithm in terms of time complexity. Big-oh can be defined as :-

The function $f(n) = O(g(n))$ iff $\exists$ positive constants such that

$f(n) \leq c * g(n) \; \forall \; n > n_0$.

Date
Page

OR,

def-2:- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.

The function $f(n) = O(g(n))$ iff $f(n) \leq c * g(n)$ for all $n \geq n_0$, $c > 0$ and $n_0 \geq 1$.

Example

$f(n) = 2n+3$

or, $2n+3 \leq 2n+3n$

$2n+3 \leq 5n$

$\therefore f(n) = O(n)$

2.) Big-Ω

Big-Ω is used to define the lower bound of an algorithm in terms of time complexity. Big-Ω notation can be define as:

The function $f(n) = \Omega(g(n))$ iff ∃ positive constants such that $f(n) \geq c * g(n)$ ∀ $n > n_0$.

Consider function f(n) as time complexity of an algorith function g(n) is the most significa term. The function $f(n) = \Omega(g(n))$ ip and only if $f(n) \geq C * g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$

for example:- $f(n) = 2n+3$

$2n+3 \geq 2n$

$f(n) \geq g(n)$

$\therefore f(n) = \Omega(g(n))$

Big-θ

Big-θ notation is used to define the averag bound of an algorithm in terms of time complexity. Big-θ notation can be defined as:

The function $f(n) = \theta(g(n))$ iff $\exists$ positive constants c_1, c_2 and n_0 such that

$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \; \forall n \geq n_0$