INDEX

SJT 313 A-B

NAME: T. Prashanth Reddy STD.: B.Tech SEC.: IT ROLL NO.: 12BIT0077 SUB.: ITE103

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		Quiz-3- 24th Oct.		
		Budham Baker - Rose Colour book		
		1st Quiz - Aug. 8		
		Intr. to Al,		
		by Mark Allen Weis		
		cat-I : Unit-II & III		
		2nd Quiz - Thursday, 19 sep searching & sorting.		
		cat II - Unit-4-full		
		Unit-5-		
	UNIT-4	Sorting & Searching		
		Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Shellsort		
		Merge, Sort, Radix Sort, Heap Sort - Searching Techniques:		
		Sequential search, Binary search, Hash Table methods		
		Hashing functions, Sorting & Searching efficiencies		
	UNIT-5	Non-linear Data Structures:-		
		Trees- Binary Tree, Tree Traversals, Expressions		
		& Search Tree.		

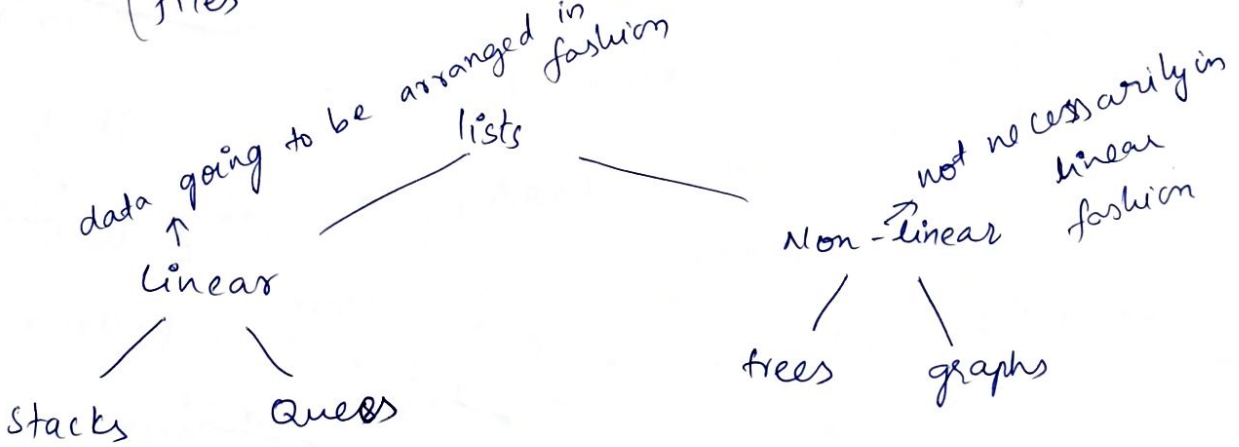
15/7/13

Datastructure:

Datastructure is a way of organising (or) arranging the data, stores them in the memory finally provides the relationship between data items.

Ex:

Simple d.s
- Arrays - Homogenous — d.s - collects same datatype
- structure - Heterogenous — d.s - collects datas of different datatypes
- lists
- files



insertion - push ; deletion - pop

→ Differentiate Static and Dynamic d.Structure

Static -	Dynamic
1) memory is fixed because memory is allocated during compile time itself	1) memory is allocated during execution time only when it is requisted
2) since, it is fixed continuous memory is allocated	2) Non- contiguous memory is allocated

3) wastage of memory is expected	3) No wastage of memory

Array: Insertion & Deletion and other operations at any place of the array

Stack: Insertion & Deletion (push & pop) are restricted to only one end called "top" pointer.

A algorithm should satisfy the following criteria

1) Input need to be supplied
2) Atleast one output it should be produced
3. Definiteness - steps
4. finiteness - algorithm should be terminated at some point
5. Algorithm should be simple.

A problem is supplied an algorithm should contain

1) Data items and their relationship
2. How to represent the data items in memory (memory representation)
3. Operations related to the data items

Analysing an Algorithm:

1. check for correctness of the algorithm
2. no. of lines used in the algorithm
3. Simplicity of the algorithm

These aspects alone not decides whether it is best algorithm. Important aspects are

1. Time complexity- How much CPU time utilised

2. Space complexity- how much memory utilised

Time Complexity:

This is calculated based on

1. what machine is used
2. what instructions set is followed
3. How much time is used to execute instruction set
4. Time taken by the compiler to convert Source language to machine language

Eventhough the above mentioned aspects are very important we are not going to consider them because machine varies and Compiler varies

So, based on number of lines and loops we are going to decide time complexity

Abstract Datatype (ADT).

18/7

Stack:

1. special linear data structure (array)
2. FILO (first in last out)
3. LIFO (Last in first out)

```
#define size 5
struct stack
{
    int a[size];
    int top;
}st;
int st_full()
{
    if (st.top == size-1)
        return 1;
    else
        return 0;
}
int st_empty()
{
    if (st.top == -1)
        return 1;
    else
        return 0;
}
```

```
void st_pop()
{
    int item;
    if (st_empty() == 1)
printf ("stack empty");
    else
    item = st.a[st.top--];
}
void st_display()
{
    int temp;
    if (st_empty() == 1)
printf ("stack empty");
else
{
    temp = st.top;
while (temp >= 0)
{printf ("%d", st.a[temp]);
    temp--;
} } }
```

Applications of Stack:

1. Expression conversion
   - In-fix to pre-fix
   - In-fix to post-fix

( )	1
exponent ^	2
/, *, %	3
+, -	4

Algorithm
Infix: to postfix:

The operator will be placed in b/w the operands

1) Read the given infix exp from left to right if any operand comes ~~post~~ print it on the output array

If any operator comes check the following Condn's.

i) if the incoming operator priority is greater than the stack operator priority push the incoming operator

ii) If the inc. op. pr. is less than the stack. ope priority go and pop the operator from top of the stack till the condn is true and then push the incoming operator

iii) If the incoming ope. priority & the stack op. priority are equal go and pop the operators from the top of the stack till the

condn is true and push the incoming operator

iv) If the incoming operator is "(" push the incoming operator

v) If the incoming op is ")" go and pop the oper. from the top of the stack till matching operator is found and delete the incoming ope. and stack operator

vi) If there is no further input to read go and pop the operators from the top of the stack till stack becomes empty

22/7

1Q:a) WAP to find out fibonocci series using recursive function.

b) Identify whether the given number is Armstrong (or) not. Use recursive function.

0, 1, 1, 2, 3, 5, 8, 13 ...

a) Fibonocci series:

0, 1, 1, 2, 3, 5, 8, 13 ....

```
int main()
{
  int a=1, b=1, c=0, n, i;
  cout<<" upto what number series is to printed? ";
  cin>>n;
  for(i=0; i<n; i++)
  {
    c = a+b;
    a = b;
    b = c;
    cout<<c<<" ";
  }
}
```

b) Armstrong number: 153, $1^3+5^3+3^3 = 153$

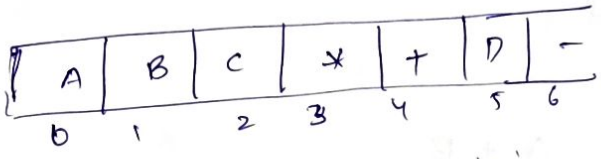
```
int main()
{ int n,nd=0;
  cout<<"enter the number";
  cin>>n;
  for( i=0; n!=0; i++)
  { n = n/10;
    nd++;
  }
```

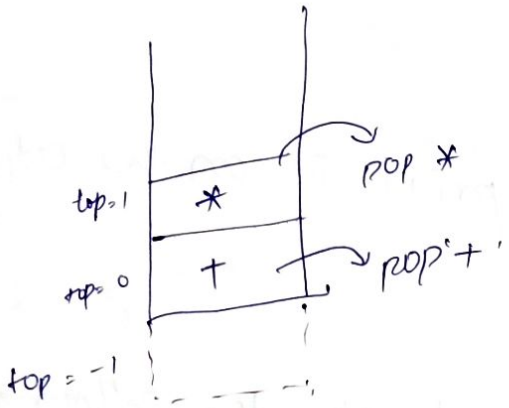
23/7

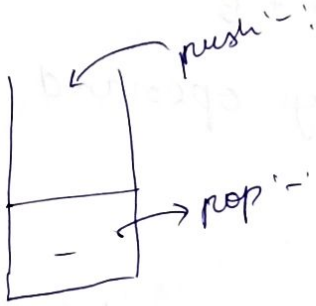
Infix to Postfix Conversion:

1. A + B * C - D

output array

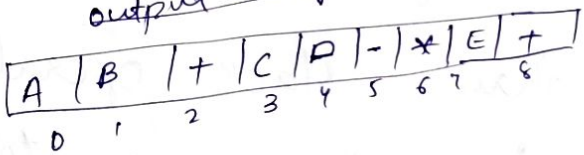


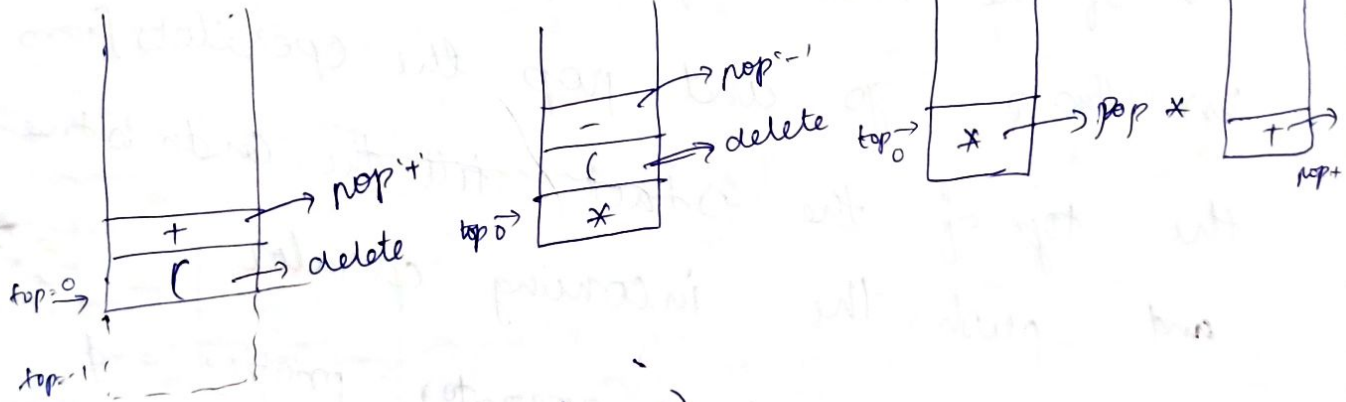




2. (A + B) * (C - D) + E.

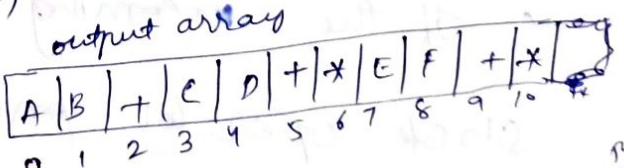
output array

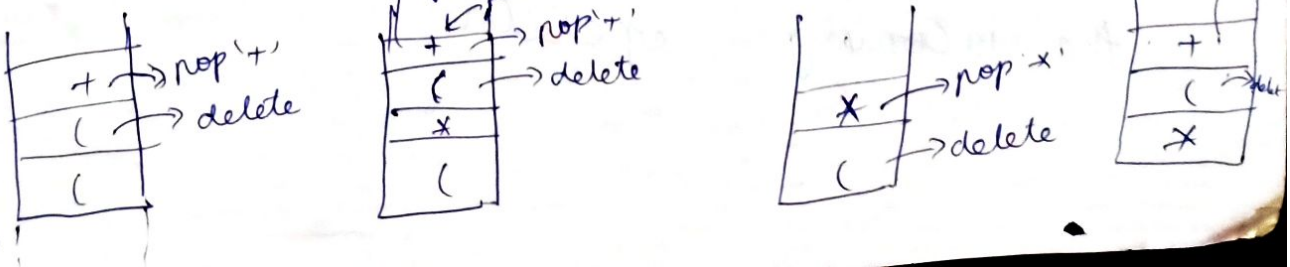




3. ((A + B) * (C + D)) * (E + F)

output array





Infix - pre fix conversion: (Algorithm)

1) Reverse the given infix expression

2. Read the reversed expression from left right

Exp

A + B

3. If any operand comes print it on the output array

4. If any operator comes check the following conditions

a) If the incoming operator priority is higher than stack operator priority push the incoming operator

b) If the incoming operator priority is less go and pop the operators from the top of the stack till the condn is true and push the incoming operator

c. If the incoming operator priority and stack operator priority are equal push the incoming operator

d. If the incoming operator is "(" push the incoming operator

e. If the incoming operator is ")" go and pop the operators from top of stack till matching operator is found then delete the incoming operator

5. Repeat step 2, 3, 4 till you reach the end of the input

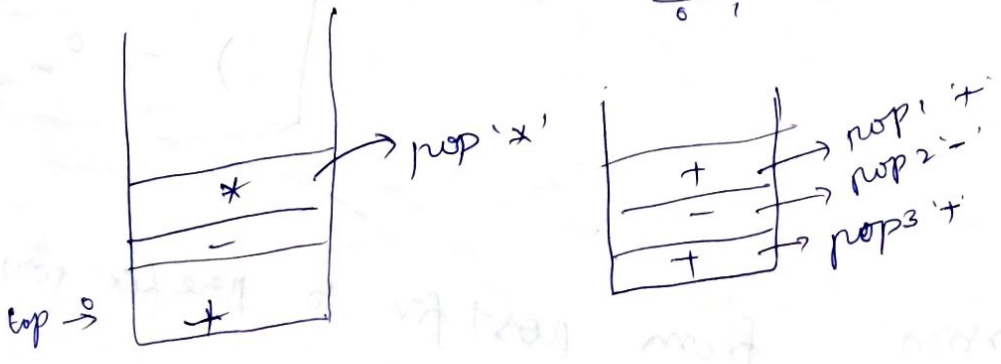
6. If there is no further input to read go and pop the operators from top of the stack till stack becomes empty

Ex1: A + B * C − D + E

↳ E + D − C * B + A

output array:

0	1	2	3	4	5	6	7	8
+	−	+	A	*	B	C	D	E

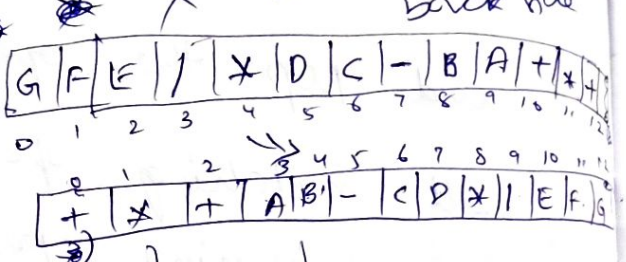


Ex 2:

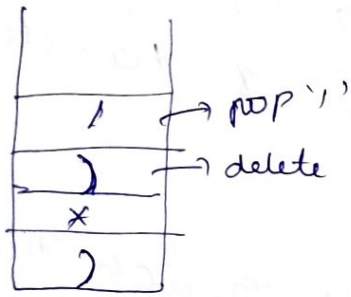
$$((A+B)*(C-D)+(E/F)*G)$$

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

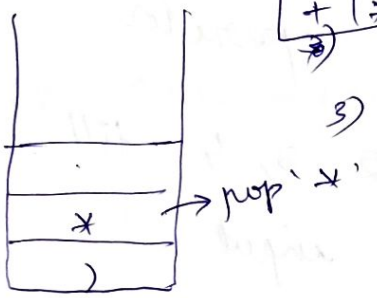
we have to fill from back side



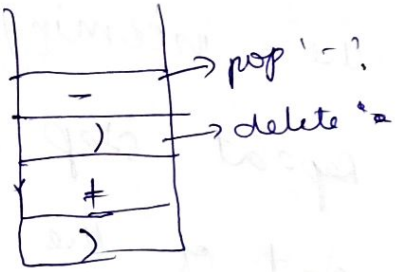
1)



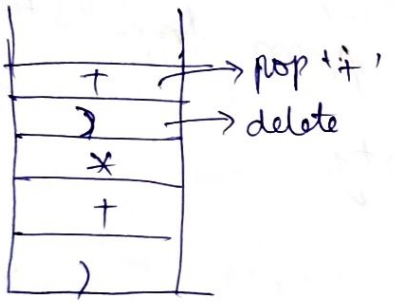
2)

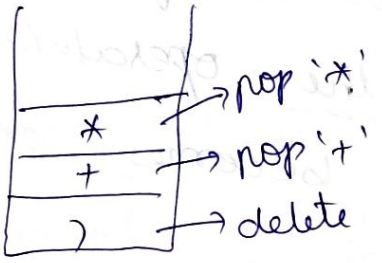


3)



4)





+	1
-	2
*	3
/	4
↑	5
)	0

H.W

conversion from postfix to prefix (or) prefix to postfix.

```
<iostream>
<conio.h>
#define size 20;
<string.h>
void pfix();
void push(int);
char pop();
char infix[20], prefix[20], revexp
```

25/7

Evaluation of postfix Expression:

Algorithm:-

1) Read the given postfix expression from left to right
2. If any operand comes push it on the stack
3. If any operator comes pop the two recently pushed items from the stack perform the operation and push the result back to the stack.
4. Repeat step '2' and '3' till you reach the end of the input
5. If there is no further input to read display the stack elements.

Ex: A + B * C - D Let A = 2; B = 3; C = 5; D = 4

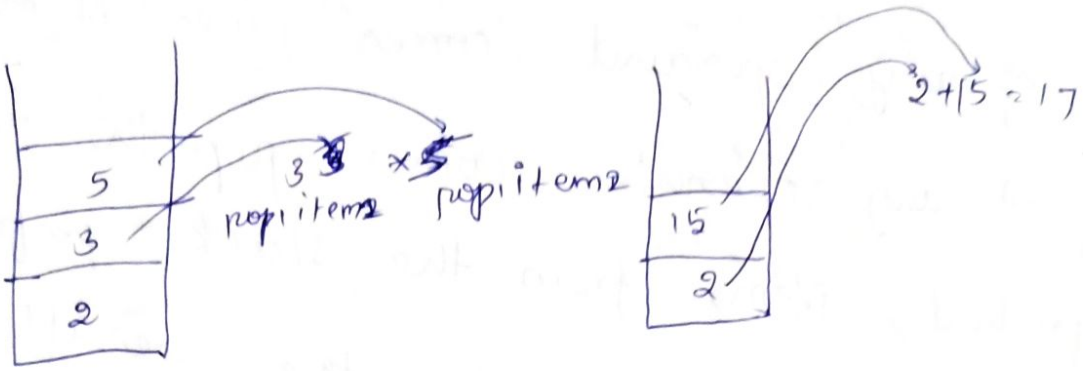
2 + 3 * 5 - 4

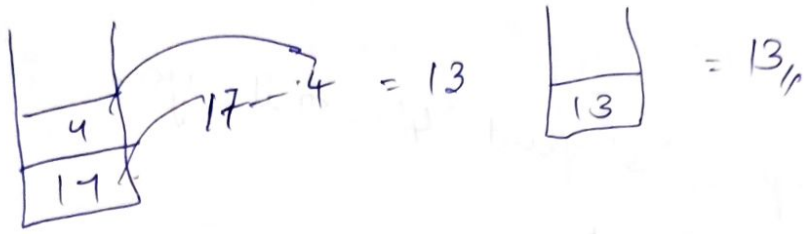
= 13

postfix exp:

A B C * + D -

2 3 5 * + 4 -





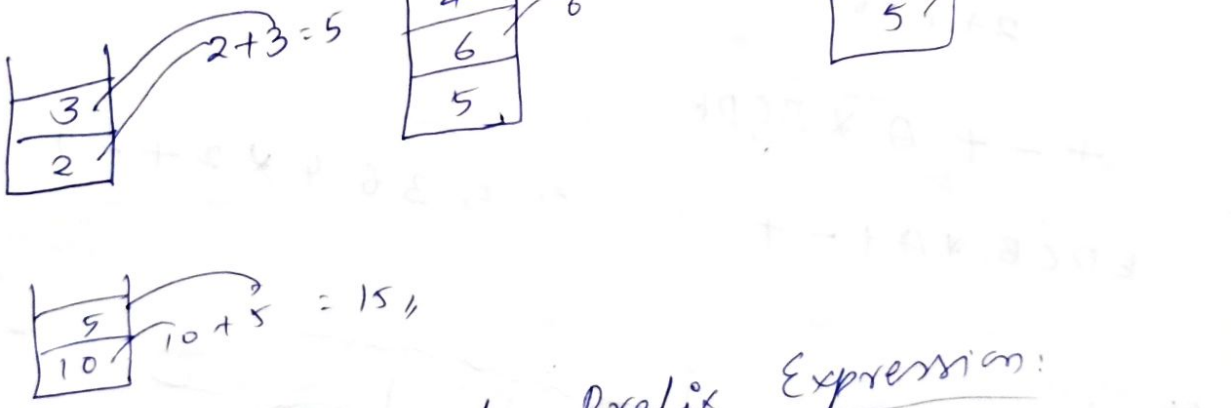
2) $(A+B) * (C-D)+E$

$A=2$; $B=3$; $C=6$; $D=4$; $E=5$

$(5) * (2) + 5 = 15//$

postfix exp: $AB+CD-*E+$

2) $23+64-*5+$



Evaluation of Prefix Expression:

1. Reverse the given prefix expression
2. Read the reversed expression from left to right

3. If any operand comes push it on the stack.

4. If any operator comes pop the two recently pushed items from the stack perform the operation and push the result back to the stack.

5. Repeat step '3' and '4' till you reach the end of the input

6. If there is no further input to read display the stack elements.

Ex: A + B * C - D + E

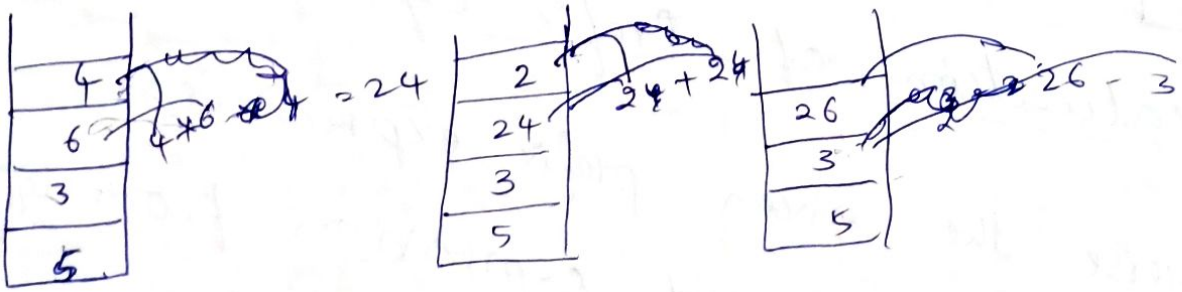
A = 2 ; B = 4 ; C = 6 ; D = 3 ; E = 5

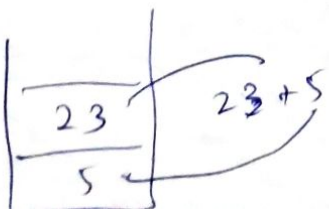
2 + 4 * 6 - 3 + 5 = 28

+ - + A * B C D E

E D C B * A + - +

⇒ 5 3 6 4 * 2 + - +



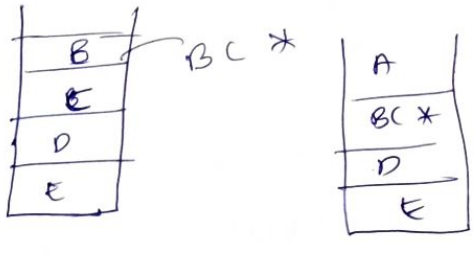


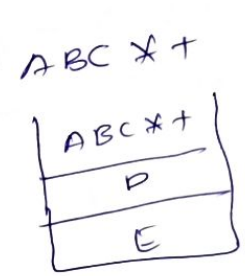
Conversion of prefix to postfix:

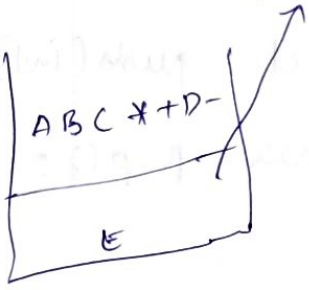
A + B * C - D + E

⇒ Prefix: + - + A * B C D E

E D C B * A + - +







H.W:
Write an algorithm to convert postfix expression to prefix infix form

- Evaluation of post-fix:

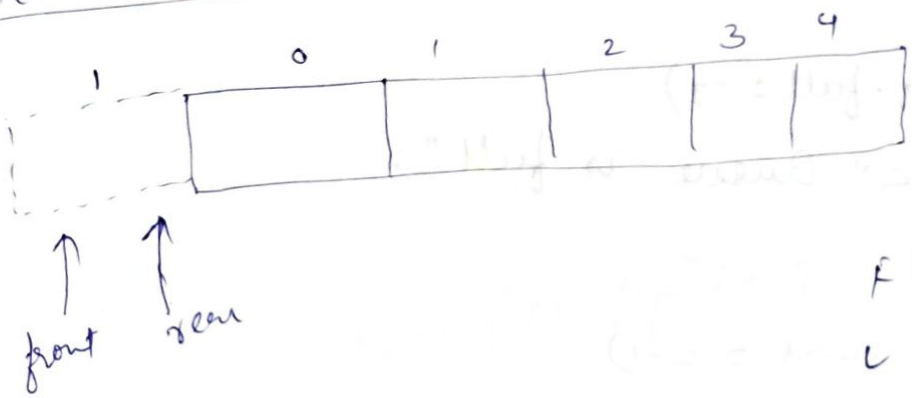
```
#<conio.h>
#define size 20;
void eval();
void push(int);
char pop();
```

Queue:-

- Linear Queue
- Circular Queue
- Double ended queue
- priority Queue — Ascending / descending

insertion - "enqueue"
deletion - "dequeue"

Linear Queue:-



FIFO
LILO

```
struct lqueue
{   int a[size];
    int front, rear;
} lq
```

functions performed:-

```
int lq_empty()
{ if (lq.rear == -1)
    return 1;
  else
    return 0;
}
```

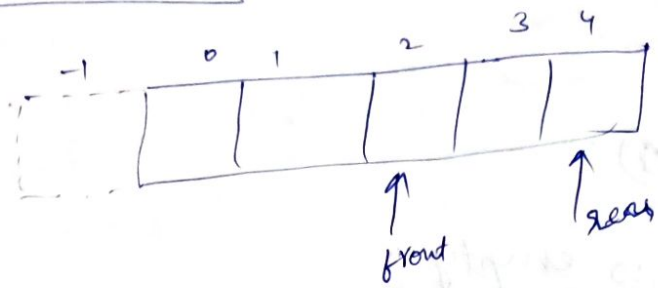
```
int lq_full()
{
    if (lq.rear == size-1)
    return 1;
    else
    return 0;
}

void lq_insert (int item)
{
    if (lq.full == 1)
    cout << "Queue is full";
    else
    { if (lq.rear == -1)
      {
        lq.front = 0;
        lq.rear = 0;
        lq.a[lq.rear] = item;
      }
    else
      lq.a[++lq.rear] = item;
    }
}
```

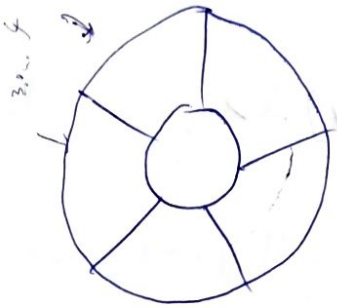
```
void lq.delete()
{
  int item;
  if (lq.empty() == 1)
  cout << " queue is empty ";
  else
  {
    if (lq.front == lq.rear)
    {
      item = lq.a[lq.front];
      lq.front = -1;
      lq.rear = -1;
    }
    else
      item = lq.a[lq.front++];
  }
}
void lq.display()
{
  int temp;
  if (lq.empty() == 1)
  printf(" queue is empty");
  else
  {
    temp = lq.front;
    while (temp <= lq.rear)
    {
      printf("%d", lq.a[temp]);
      temp++;
    }
  }
}
```

30/1

Linear Queue:



Circular Queue:



front = -1 ; rear = -1 ;

```
struct queue
{
    int a[size];
    int front, rear;
} cq;
```

```
int cq_empty()
{
    if ( cq.rear == -1)
    return 1;
    else
    return 0;
}
int { cq.full()
{
  if ( cq.rear == cq.(front + 1 % size)
    return 1;
    return 0;
}
```

front

rear

```
void cq_insert()
{
  if (cq_full()==-1)
  printf("queue full");
  else
  { if (cq.rear==-1)
  cq.rear = cq.rear + 1%size;
  cq.front = cq.front + 1%size;
   cq.a[cq.rear] = item;
  }
  else
  { rear = rear + 1%size;
    cq.a[cq.rear] = item
    }
  }
}
void cq_delete()
{
  if (cq.empty()==1)
  printf("queue empty");
  else
  { if (cq.rear == cq.front)
  cq.rear = cq.front = -1;
  else
  cq.front = (cq.front + 1)%size;
  }
}
```

display -> front -> rear

```
void cq_display()
{
 if (cq.empty()==1)
 cout<<"queue empty";
 else
 { temp = cq.front;
 while (temp!= cq.rear)
 {
 printf("%d", cq.a[temp]);
  temp = temp + 1%size
 }
 printf("%d", cq.a[temp]);
}
```

because the last element will not be printed

H.W

Propose a Queue data structure which supports insertion, deletion operations on both the ends

31/7/13

priority queues are also called as "heaps"

~~Appl~~

Scheduling Problem:

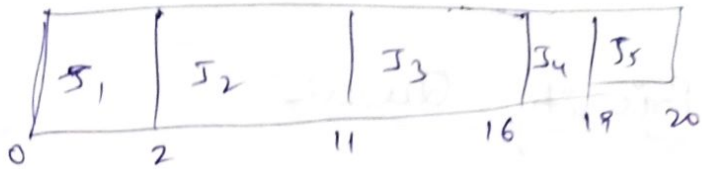
You are given five jobs $J_1, J_2, \ldots J_5$ with their known running times $t_1, t_2, t_3 \ldots t_5$ and you are given only one Processor, write an algorithm to find the overall average completion time. (Assume non-pre-emptive scheduling)

* Pre-emptive - if high priority job comes processor suspends its execution & accepts the high priority job.
* time calculated in clock cycles

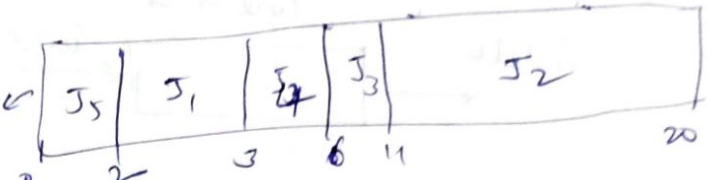
Shortest-time-job scheduling:-

$J_1 = 2$
$J_2 = 9$
$J_3 = 5$
$J_4 = 3$
$J_5 = 1$

$J_5 = 1$
$J_1 = 2$
$J_4 = 3$
$J_3 = 5$
$J_2 = 9$



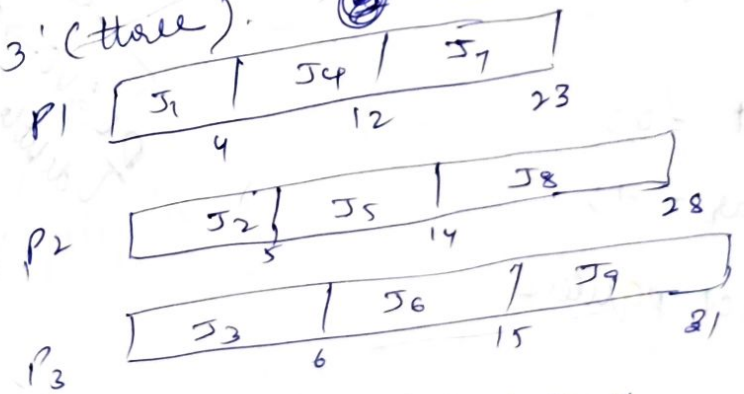
shortest time jobs first ←



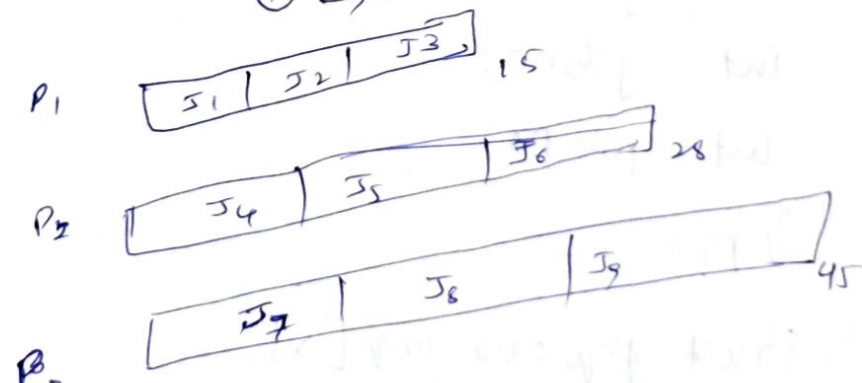
* Priority Queue is used to solve problem
* You are given jobs $J_1, J_2 \ldots J_9$ with their running times $t_1, t_2 \ldots t_9$. Write an algorithm to find the minimum overall completion time and number of processors given is '3' (three).

$J_1 = 4$
$J_2 = 5$
$J_3 = 6$
$J_4 = 8$
$J_5 = 9$
$J_6 = 11$
$J_7 = 14$
$J_8 = 15$
$J_9 = 16$

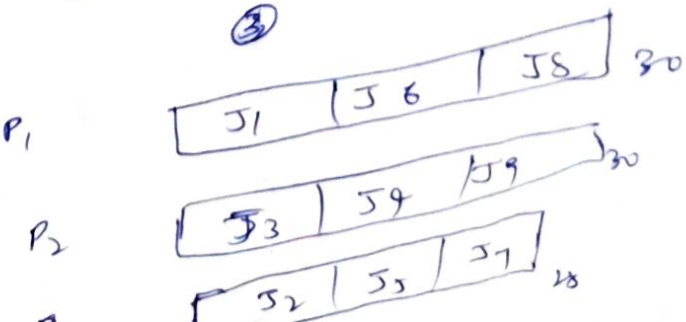
(2)



(1) → worst way.

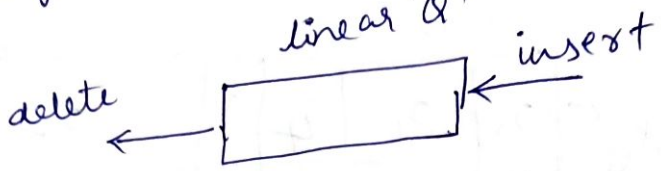


(3)

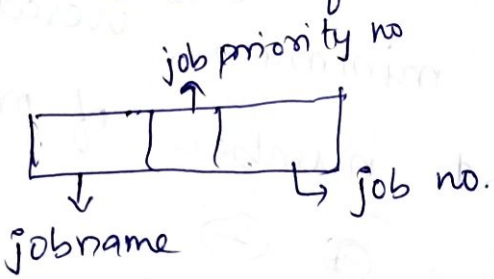


1/8/2013

Priority Queue:



'Queue' consists of a structure



DEQUE → double ended queue — refer once

```
front = 0;
rear = -1;
struct pqueue
{
char jobname[10];
int jobno;
int jobpr;
} pq;
Struct pqueue pq[5];
```

```
int prqueue_empty()
{
 if (front > rear)
 return 1;
 else
 return 0;
 }
int prqueue_full()
{
 if (rear = size-1)
 return 1;
 else
 return 0;
 }
 void prdelete()
{ if (rear == -1)
 printf(" queue empty");
 else
 front ++;
 }
 void pr_insertion()
 {

 }
```

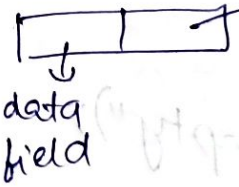
5/8/13

UNIT-3

Linked list:

Singly linear linked list
Singly circular linked list
Doubly linear linked list
Doubly circular linked list

Linked list is a basic linear data structure.
Linked list is a collection of nodes where each node will have minimum of two fields. one is data field and the otherone is address field.

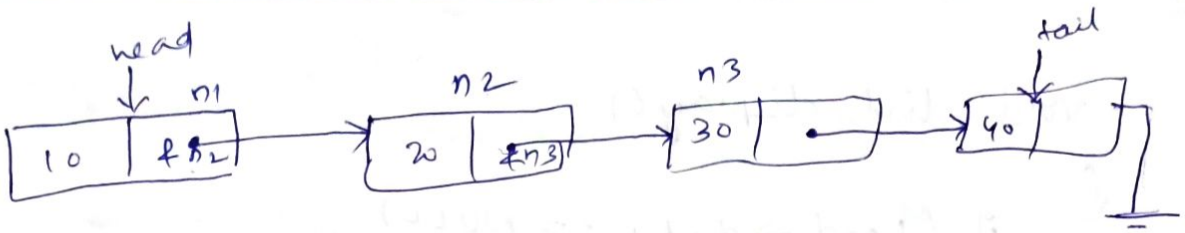



```
struct node
{
  int data;
  struct node * next;
};
```

`int * next;` → means pointer going to store address of integer data type

`struct node * next;` → means pointer going to store address of node

data field is used to store the data and address field is used to store the address of next node



array
linked list } basic data structure because these two d.s are used to implement other d.s Ex: stack, Queue, tree, graph.

1) W A 'C' program to create singly linear linked list with four nodes and display the same

```
struct node
{ int data;
  struct node *next;
}; struct node *temp, *new, *head, *tail;

void list-creation()
{
  if (head->data == NULL)
  head->data = item;
  else
  { new = (struct node *) malloc (sizeof(struct node));
    new->data = item;
    new->next = NULL;
    tail->next = new;
    tail = new;
  }}
```

```
void list_display()
{
    if (head->data == NULL)
    printf("list empty");
    else
    {
    temp = head;
    while (temp != NULL)
    {
    printf("%d", temp->data);
    temp = temp->next;
    } } }
void main()
{
    int choice, item;
    nw = (struct node *) malloc(sizeof(struct node));
    nw -> data = NULL;
    nw -> next = NULL;
    head = tail = nw;
do
{
printf("\n1. Creation \n2. display \n3. exit");
    printf("enter your choice");
    scanf("%d", &choice);
    switch(choice)
    {
```

6/8/13

```
case 1:
        list_creation();
        break;
case 2:
        list_display();
        break;
case 3:
        exit(0);
}} while(choice <=3)
}
```

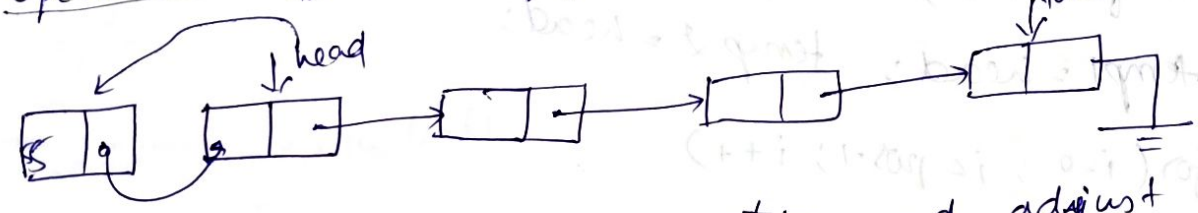
6/6/13

```
#include<stdio.h>
#include <malloc.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
}; struct node *nw, *head, *tail, *temp;
void list_creation()
{
int item;
printf("enter the item");
scanf("%d", &item);
if (head->data= NULL)
head->data = item;
else
{
```

```
nw = (struct node*) malloc(sizeof(struct node));
nw->data = item;
nw->next = NULL;
tail->next = nw;
tail = nw;
}}
void list_display()
{
if(head==NULL)
printf("\n List empty");
else
{
temp = head;
while(temp!=NULL)
{
printf("%d", temp->data);
temp = temp->next;
}}}
void main()
{ int choice;
nw = (struct node *) malloc(sizeof(struct node
nw->data = NULL; nw->next = NULL;
head = nw; tail = nw;
do {
printf("\n 1. creation \n 2. display \n 3. Exit");
printf("\nEnter your choice");
```

```
printf  scanf ("%d", &choice);
switch (choice)
{
case 1 : list_creation ();
         break;
case 2 :
         list_display ();
         break;
case 3 :
         exit(0);
         break;
} } while (choice <=3)
}.
```

Operations on singly linked list :-



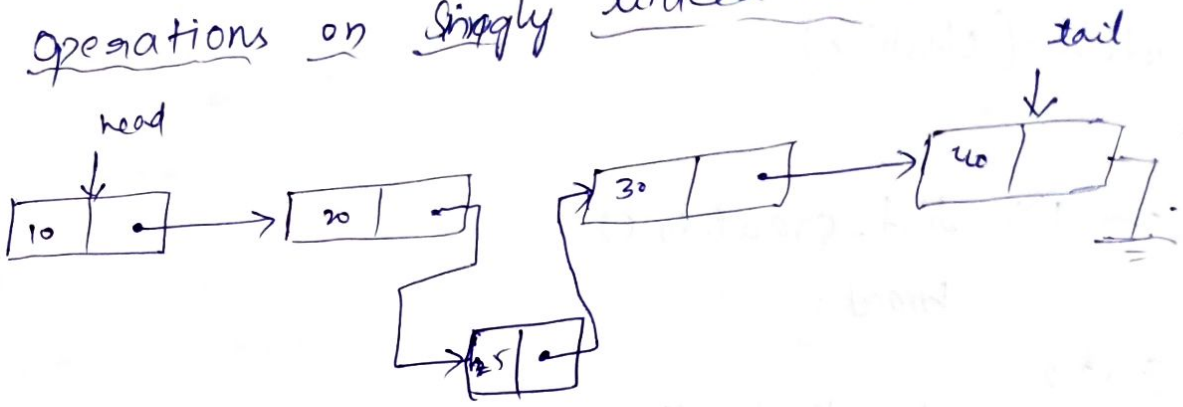
we have to create connection and adjust the head pointer.

function for insertion at the beginning :-

```
void insert_at_begin ()
{
    nw = (struct node *) malloc (sizeof (struct node));
    nw -> data = item;
    nw -> next = head;
    head = new;
}
```

7/8/13

Operations on Singly linked list:-

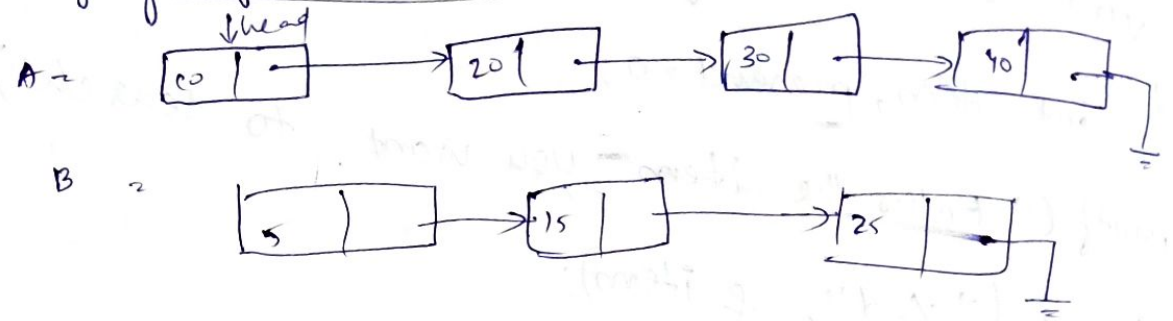



```
// void insert_at_pos()
{
    int pos, item, i;
    printf("enter the position:");
    scanf("%d", &pos);
    printf("enter the element:");
    scanf("%d", &item);
    temp1 = head; temp2 = head;
    for(i=0; i<pos-1; i++)
{
    temp1 = temp1 -> next;
    temp2 = temp2 -> next;
}  temp2 = temp2 -> next;
nw = (struct node *)malloc(sizeof(struct node));
    temp1 -> next = nw;
    nw -> next = temp2;
}
```

```
void list_searching()
{
    int item, present = 0;
printf ("Enter the item you want to search");
scanf ("%d", &item);
temp = head;
while (temp != NULL)
{
    if (temp -> data == item)
    present = 1;
    break;
else
    temp = temp -> next;
}
}

void no_of_nodes()
{   temp = head;
    int count = 0;
while (temp != NULL)
{ count ++;
  temp = temp -> next;
}
printf ("number of nodes = %d", count);
}
```

Merging of two lists:




```
void merge_list()
{
    if (head -> data == NULL) || (head -> data == NULL)
    printf ("merge not possible");
    tail -> next = head 2;
}
```

How we can perform insertion using one pointer temp1, instead of temp1, temp2?

H.co.

You are given two ~~linear~~ singly linear linked list of size 'A' and 'B'. write a function to merge the two lists so that the resulting list will have elements in ascending order.

Deletion at beginning

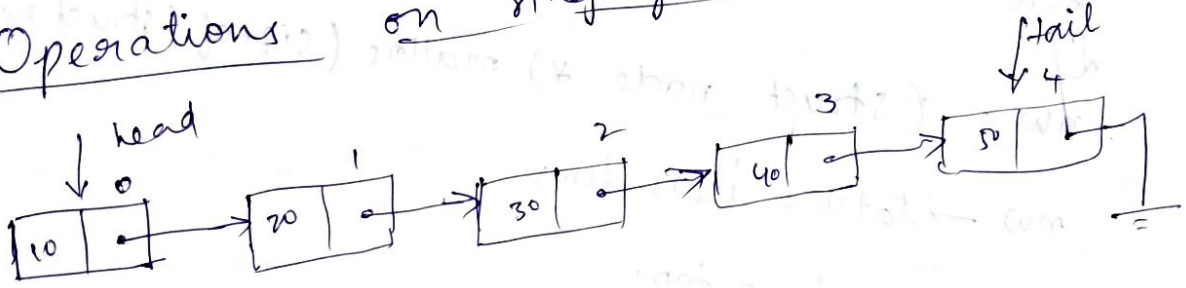
```
void dete_at_begins()
{
    if (head -> data == NULL)
    printf ("list empty");
    else
    { temp = head;
      head = head -> next;
      free (temp); }
```

Deletion at end:-

```
void del_at_end()
{
  if (head -> data == NULL)
  printf ("list empty");
else
  { temp = head;
  for (i=0; i<count-2; i++)
  { temp = temp -> next;
  tail = temp -> temp = temp->next; free(temp);
  tail -> next = NULL;
} }
```

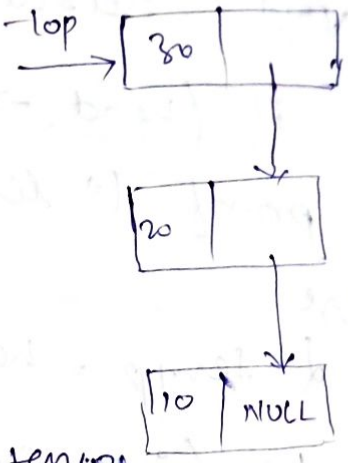
8/8/12

Operations on singly linked list:-




```
void del_at_mid()
{
  int pos;
  if (head -> data == NULL)
  printf ("list empty");
  else
  { temp1 = temp2 = head;
  for (i=0; i<pos-1; i++)
  { temp1 = temp1 -> next;
  temp2 = temp2 -> next;
  temp2 = temp2 -> next;
  temp1 -> next = temp -> next;
  }
}
```

Stack using linked list :-



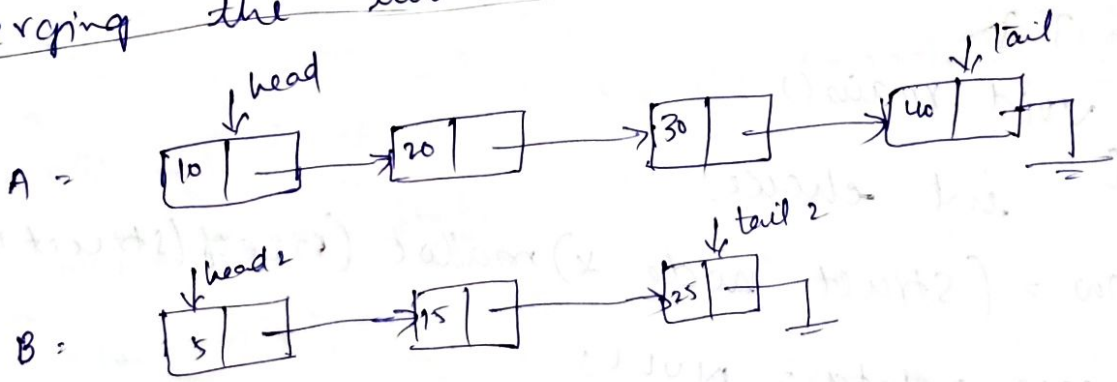

```
struct node
{
  int data;
  struct node * next;
}; struct node * top, *new, *temp;

void st_push()
{
    int item;
 if (top -> data == NULL)
  top -> data = item;
  else
  {
  nw = (struct node *) malloc (sizeof (struct node));
   nw -> data = item;
   nw -> next = top;
    top = nw;
   }
}
 void st_pop()
 {
      if (top -> data == NULL)
        printf ("stack underflow");
 else
```

```
{ printf ("%d", top->data);
  temp = top;
  top = top->next;
  free (temp);
 }
}
void st_display()
{
  if (top->data == NULL)
  printf ("stack underflow");
  else
  { temp = top;
  while (temp != NULL)
  { printf ("%d", temp->data);
    temp = temp->next;
  }
 }
}
void main()
{
  int choice;
nw = (struct node *) malloc (sizeof (struct node));
  nw->data = NULL;
  nw->next = NULL;
  top = nw;
do
{
  printf ("1. push in 2. pop in 3. Display in 4. Exit");
printf ("enter the choice");
  scanf ("%d", &choice);
```

```
switch (choice)
{
case 1: st_push();
        break;
case 2: st_pop();
        break;
case 3: st_display();
        break;
Case 4: exit(0);
        break;
} } while (choice <= 4);
}.
```

Merging the two linked lists in Ascending order




```
void merge_ascending()
{  temp1 = head1;  temp2 = head2;
while ((temp1 != NULL) && (temp2 != NULL))
{  if (temp1 -> data > temp2 -> data)
   {  nw3 -> data = temp2 -> data;
      temp2 = temp2 -> next; }
   else
   {  nw3 -> data = temp1 -> data;
      temp1 = temp1 -> next; } }
```

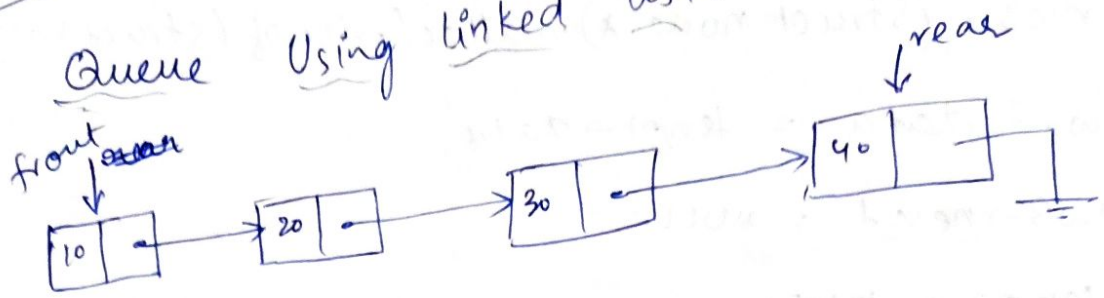
```
while ( temp != NULL)
{
    nw3 = (struct node *) malloc(size of (struct node));
    nw3 -> data = temp1 -> data;
    nw3 -> next = NULL;
    temp3 = nw3;
}
while ( temp2 != NULL)
{
    nw3 = (struct node *) malloc( size of (struct node));
    nw3 -> data = item;
    nw3 -> next = NULL;
    temp3 -> next = nw3;
    temp3 = nw3;
} }.
```

~~log 2~~

~~log 2~~

12/6/13

Queue Using linked list:




```
struct node()
{
    int data;
    struct node *next;
}; struct node *front, *rear, *nw, *temp;

void queue-insert()
{
    if (rear -> data == NULL)
    rear -> data = item;

else
{
    nw = (struct node *) malloc (sizeof(struct node));
    nw -> data = item;
     nw -> next = NULL;
     rear -> next = nw;
      rear = nw; } }

void queue-delete()
{
  if (front -> data) == NULL)
   printf ("queue is empty");
  else
  {  printf ("%d", front -> data);
```

```
temp = front;
front = front -> next;
free (temp);
}}
void queue_display()
{
    if (front -> data == NULL)
printf ("queue empty");
else
{   temp = front;
while ( temp != NULL)
{
printf ("%d", temp -> data);
temp = temp -> next;
}}}
```

Q. You are given two polynomial equations P_1 and P_2 of size 'm' and 'n' respectively WA program to add the two polynomial equations & display the same.

Polynomial Addition:

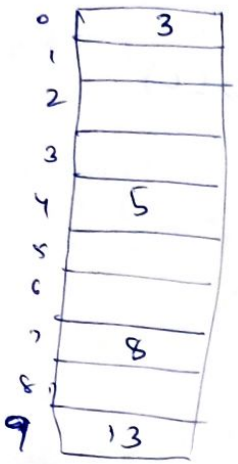
$P_1 = 13x^9 + 8x^7 + 5x^4 + 3$

$P_2 = 11x^{10} + 8x^9 + 6x^7 + 4x^3 + 2$

Ans:

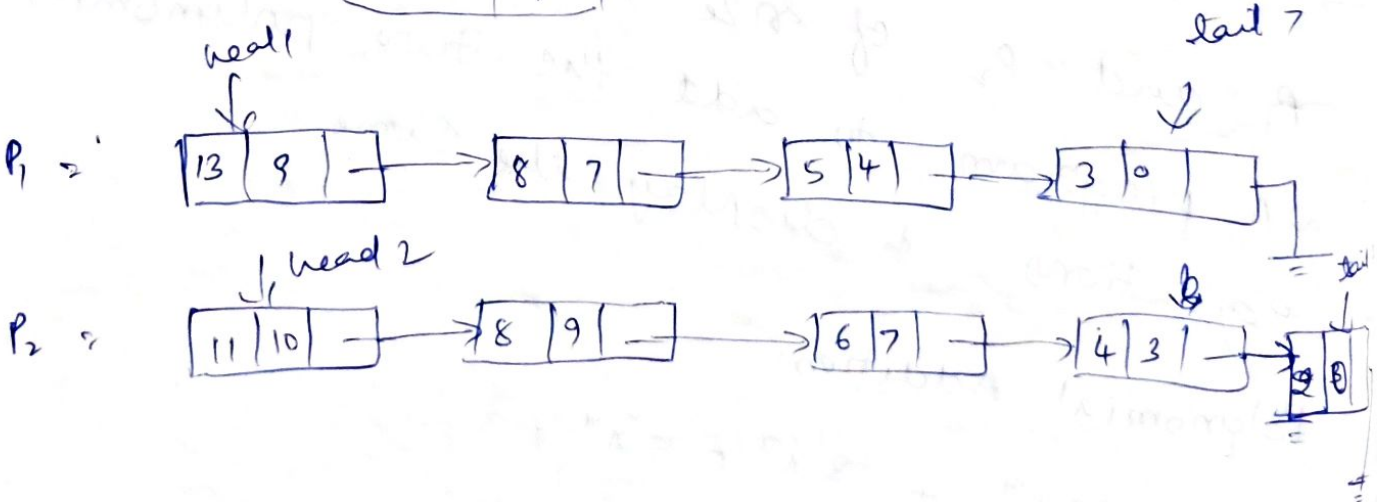
- If the powers of the polynomial eqn are reducing regularly / continuously array is efficient

Ex:



In this case if the size of polynomial is known as two-dimensional array is efficient

3	0
5	4
8	7
13	9



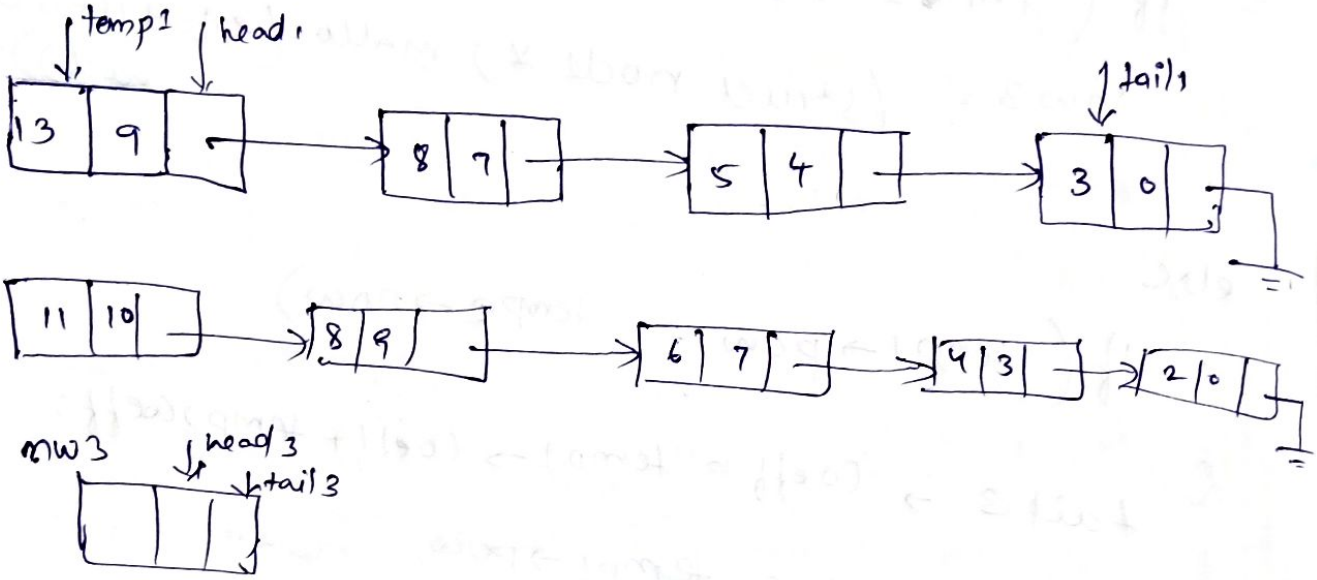
```
void poly_add()
{
  temp1 = head1;
  temp2 = head2;
while ( (temp1 != NULL) && (temp2 != NULL))
{  if ( tail3 -> data == NULL)
     nw3 = (struct node *) malloc(sizeof(struct
                                  node));
  :
else
    if( temp1 -> pow == temp2 -> pow)
{   tail3 -> coeff = temp1 -> coeff + temp2 -> coeff;
    tail3 -> pow = temp1 -> pow;
    temp1 = temp1 -> next;
    temp2 = temp2 -> next;
}
else if ( temp1 -> pow < temp2 -> pow)
{   tail3 -> coeff = temp2 -> coeff;
    tail3 -> pow = temp2 -> pow;
    temp2 = temp2 -> next;
}
else
{   tail3 -> coeff = temp1 -> coeff;
    tail3 -> pow = temp1 -> pow;
    temp1 = temp1 -> next;
}
```

14/8/13

Polynomial Addition:

$P_1 = 13x^9 + 8x^7 + 5x^4 + 3$

$P_2 = 11x^{10} + 8x^9 + 6x^7 + 4x^3 + 2$




```
void polyadd()
{
    temp1 = head1;
    temp2 = head2;
while ((temp1 != NULL) && (temp2 != NULL))
{
 if (tail3 -> coeff != NULL)
 {
    nw3 = (struct node *) malloc (sizeof(struct node));
    nw3 -> next = NULL;
    tail3 -> next = nw3;
    tail3 = nw3;
 }
```

```
if ( temp1 -> pow == temp2 -> pow)
{
    tail3 -> pow = temp1 -> pow;
    tail 3 -> coeff = temp1 -> coeff + temp2 -> coeff;
    temp1 = temp1 -> next;
    temp2 = temp2 -> next;
}
else if (temp1 -> pow > temp2 -> pow)
{
    tail3 -> pow = temp1 -> pow;
    tail3 -> coeff = temp1 -> coeff;
    temp1 = temp1 -> next;
}
else
{ tail3 -> pow = temp2 -> pow;
  tail3 -> co.eff = temp2 -> coeff;
  temp2 = temp2 -> next;
}
}
while ( temp1 != NULL)
{
  tail3 -> pow = temp1 -> pow;
  tail3 -> coeff = temp1 -> coeff;
  temp1 = temp1 -> next;
}
```

```
while (temp!=NULL)
{
  tail3 -> pow = temp2 -> pow;
  tail3 -> coeff = temp2 -> coeff;
  temp2 = temp2 -> next;
}
}.
```

function to display the Resultant polynomial equn:

```
void polydisplay()
{
  temp3 = head3;
  while (temp3 != NULL)
{
printf ("%d x ^ %d", temp3->coeff, temp3 -> pow);
  temp3 = temp3 -> next;
}
}
```

Problem:-

Q. You are given different weights $w_1, w_2, w_3 \ldots w_n$ with their profits $P_1, P_2, P_3, \ldots P_n$ and you are given a bucket of size 'm'. Write an algorithm to find the maximum profit (Napsak problem)

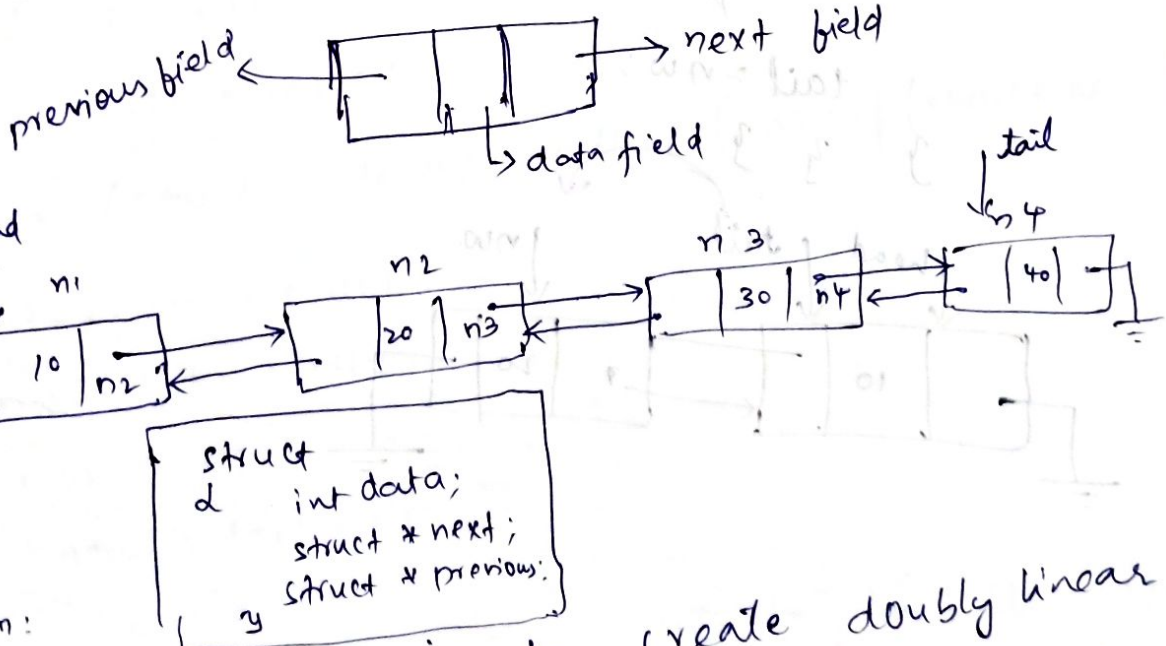
19/8/2013

Doubly linked list:

* want to travel in both the directions

It is a collection of nodes where each node will have '3' fields. one is data field mainly used to store data and nextfield to store the address of next node and a previous field. which is used to store the address of previous field.

In doubly linear linked list the first node of previous field and last node next field will be 'NULL'



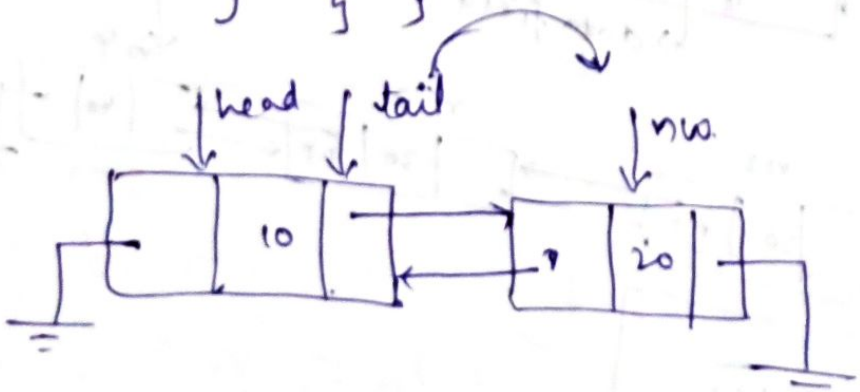

```
struct
{   int data;
    struct * next;
    struct * previous;
};
```

Problem: To create doubly linear linked list with 'n' nodes and display the same.

1) A 'C' Program

function for creation of doubly linear linked li

```
int n, i ; printf("enter the no of nodes:");
scanf("%d", &n);
for( i=0; i<n; i++)
void list_creation(int item)
{
{
if (head->data == NULL)
head -> data = item;
else
{
nw = (struct *node*) malloc (size of (struct no
nw -> data = item;
nw -> next = NULL;
tail -> next = nw;
nw -> prev = tail;
tail = nw;
} } }
```



function to display elements:

```
void double_link.display()
{
    if (head -> data == NULL)
     printf (" list Empty ");
    else
    { temp = tail;
    while ( temp != NULL)
    {
      printf (" %d", temp -> data);
      temp = temp -> prev;
      }
      } }.
```

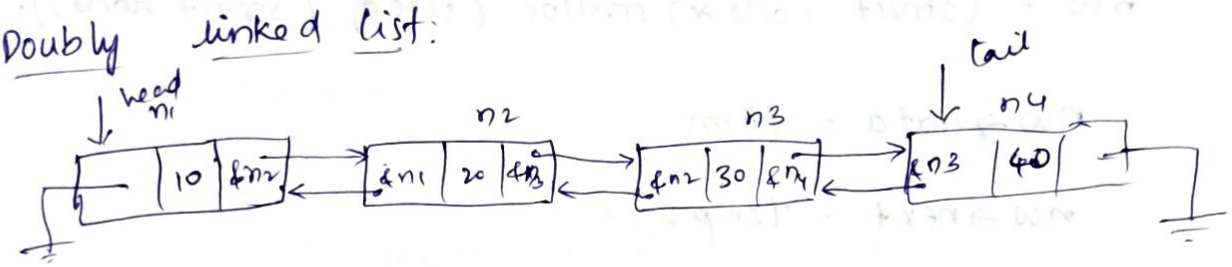
~~int~~ main function:

```
void main()
{       int choice;
nw = (struct node *) malloc (sizeof (struct node));
  nw -> prev = NULL;
  nw -> data = NULL;
  nw -> next = NULL;
  head = tail = nw;
do
{ printf (" 1. Creation \n 2. Display \n 3. Quit \n");
  printf (" Enter your choice ");
  scanf (" %d", &choice);
```

```
Switch (choice)
{
    case 1:  double_link_creation();
             break;
    case 2:
             double_link_display();
             break;
    Case 3:
             exit(0);
             break;
  } } (while ch<=3);
    }
void insert_at_begin()
{
    nw= (struct *node*) malloc (sizeof (struct
    nw→data = item;
    nw→prev = NULL;
    head→prev = nw;
    nw→next = head;
    head = nw;
}
```

29/8/13

Doubly linked list:




```
void insert_double()
{
  if (pos == 0)
  {
    nw = (struct node *) malloc (sizeof (struct node));

    nw->data = item;

    nw->prev = NULL;

    nw->next = head;

    head->prev = nw;

     head = nw;

  }
else if (pos == n)
{
    nw = (struct node *) malloc (sizeof(struct node));

    nw->data = item;

    nw->next = NULL;

    nw->prev = tail;

    tail->next = nw;

     tail = nw;

}
else {      temp1 = head;
            temp2 = head;
   for( i=0 ; i < pos-1 ; i++)
   {  temp1 = temp1->next; temp2 = temp2->next
      temp2 = temp2->next;
   }
```

or

```
for(i=0; i<pos-1; i++)
{ temp1 = temp1->next;
}
temp2 = temp1->next;
```

```
nw = (struct node *) malloc (sizeof (struct node));
    nw->data = item;
    nw->next = temp2;
    temp2->prev = nw;
    temp1->next = nw;
    nw->prev = temp1;
}}
```

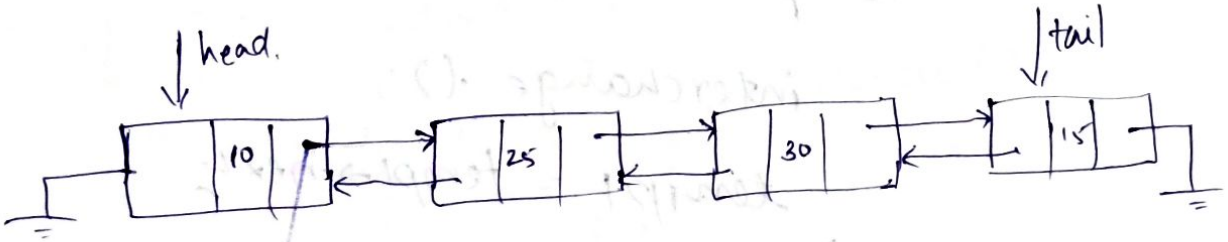
Deletion of node of Doubly linked list:

```
void del_double()
{
  if (pos == 0)
{
   temp = head;
 head = head->next;
   head->prev = NULL;
   temp->next = NULL;
   free (temp);
 }
 else if (pos == n)
{
      temp = tail;
       tail = tail->prev;
      tail->next = NULL;
       temp->prev = NULL;
   }   free (temp);
```

```
else
{   temp1 = head;
  for (i=0 ; i< pos-1 ; i++)
      temp1 = temp1->next;
   temp2 = temp1->next;
  temp1->next = temp2->next;
       temp3 = temp2;
   temp2 = temp2->next;
  temp2->prev = temp1;
   temp3->prev = NULL;
   temp3->next = NULL;
   free (temp3);
  } }
```

Q. Write a function to reverse the doubly linked list (n-1/2) steps

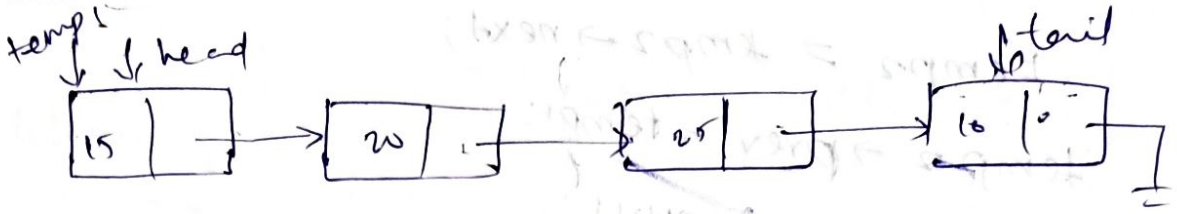
Sol:-




```
void reverse_double()
{   temp1 = head;
    temp2 = tail;
  for (i=0; i< count/2 ; i++)
{       temp = temp1->data;
        temp1->data = temp2->data;
```

```
        temp2->data = temp;
        temp1 = temp1->next;
        temp2 = temp2->prev;
    }
}
```

Singly linked list:




```
void reverse_link_list()
{
    temp1 = head;
    for(i=0; i<count/2; i++)
    {temp2 = head;
     for(j=0; j<count-i-1; j++)
      temp2 = temp2->next;
        interchange();
        temp1 = temp1->next;
    }
```

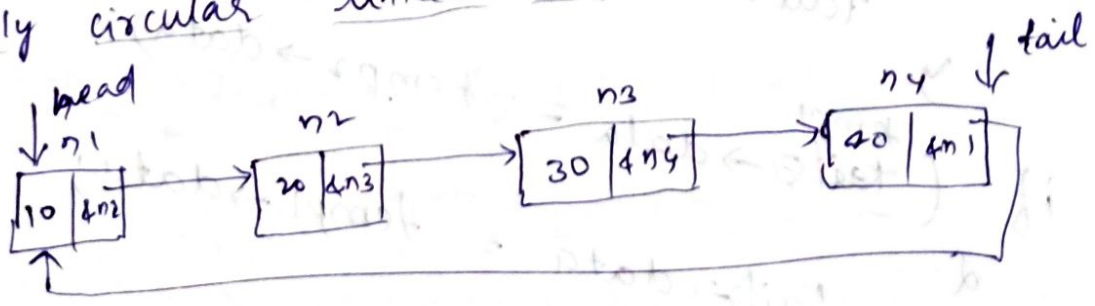
Q. You are given strings S1, S2 of sizes 'm' & 'n' respectively. Write an algorithm to compute the longest common subsequence of S1, S2.
(L.C.S)

Q. You are given two singly linked lists linear l_1, l_2 of size 'm1' and 'm2' respectively. Write a function to compute the intersection & union of l_1 & l_2.

```
wt:    void intersection()
{
       temp1 = head1;
       temp2 = head2;
  while (temp1 != NULL) && (temp2 != NULL))
{
  if (tail3 -> data != NULL)
  {
    nw3 = (struct node *) malloc (sizeof (struct node));
    nw3 -> next = NULL;
    tail3 -> next = nw3;
    tail3 = nw3;
  }
  if (temp1 -> data == temp2 -> data)
  {
    tail3 -> data = temp1 -> data;
  }
  temp1 = temp1 -> next;
  temp2 = temp2 -> next;
  }
  else
  {
```

22/8/13

Singly circular linked list:



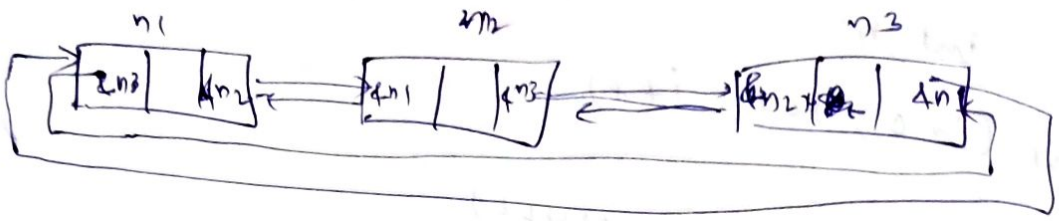

```
struct node
{
    int data;
    struct node * next;
}; struct node *nw, *head, *tail;
```

```
void creation_ circular_list()
{
    int n;
for (i=0; i<n; i++)
{
  if (head->data == NULL)
   head -> data = item;
else
{       nw = (struct node *) malloc (sizeof (struct node));
   nw ->data =item;
   nw ->next = head;
   tail ->next =nw;
    tail =nw;        } } }.
```

→ Deletion at end:

```
void   del_at_end()
{
        temp =head;
for ( i=0; i< count-1; i++)
    temp = temp1 -> next; tail = temp
temp= temp1 ->next
                tail -> next = head;
                free (temp);
}
```

Doubly circular linked list



```
struct node
{ int data;
struct node *next, *prev;
}; struct node .*new, *head, *tail, *temp;
void . double - circular_list()
{
   int n;
   for(i=0;i<n;n++)
   {
    if (head->data==NULL)
     head-> data = item;
else
{ nw = (struct node *) malloc (sizeof(struct node));
nw->data = item;
nw->prev = tail;
nw->next - head;
head->prev = nw;
  tail->next = nw;
   tail = nw;
  } }.
```

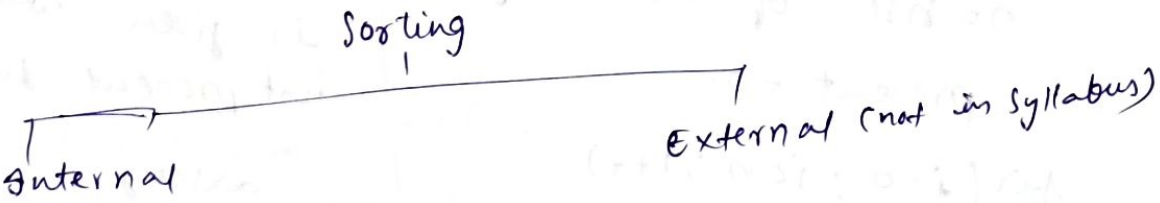
```
void del_at_end()
{
    temp = tail->prev;
    temp->next = head;
    head->prev = temp;
    tail->next = NULL;
    tail->prev = NULL;
    free(tail)
    tail = temp;
}
```

2/9/13

2/9/13

UNIT-4

Searching & Sorting Algorithms:



Searching

- basic searching algorithms
  - ⇒ → Linear search
  - → Binary search

To reduce time → Hashing (UNIT-5)

→ Linear search

Time complexity = O(n)
(n times)

→ Real time Application
(searching contacts without giving alphabets)

→ sorting of elements is not required.

O(n)
↳ Big 'O'

→ Binary Search

→ divide & conquer
(< n/2 times)

→ searching contacts by giving alphabets

→ Requires sorting of elements

0	1	2	3	4	5	6	7
15	22	11	9	13	52	16	17

n = size of the array

```
present = 0;
for(i=0 ; i<n ; i++)
{
    if (sn == a[i]);
    present = 1;
    break;
}
```

If the element is given which is not present in the array

* At first sort the elements
* check whether the element is in boundaries of first & last element

0	1	2	3	4	5	6	7
9	11	13	15	16	17	22	52

```
if (sn > a[0] && sn < a[n-1])
```

Binary Search:-

divide and conquer

0	1	2	3	4	5	6	7
9	11	13	15	16	17	22	52

min ↑ ↑ ↑ max

```
while (min <= max)
{
  mid = min + max/2;
 if (sn == a[mid])
{ present = 1; }
 else if (sn > a[mid])
{  min = mid + 1;
  else
   max = mid - 1;
}
```

3/9/13

Sorting:-

Internal sorting:

Internal sorting algorithms are capable of sorting small amount of datas which are stored in the main memory

Ex: 1) Bubble sort
2) Selection sort
3) Insertion sort
4) Heap sort
5. merge sort
6. Quick sort
7. radix sort

* drawback:
The output cannot stored permanently

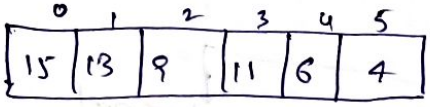
* External:

These sorting algorithms are capable of sorting reasonably large amount of data (or) which are stored on external memory

Ex:

* polyphase merging

→ Bubble Sort:- O(n²):



* works on iterations
* (n-1) iterations.
* Comparisons (n-1) + (n-2) + ... + (n-(n-1)).

Iteration 1:

15 13 9 11 6 4

13 15 9 11 6 4

13 9 15 11 6 4

13 9 11 15 6 4

13 9 11 6 15 4

13 9 11 6 4 15

iteration 2

9 13 11 6 4 15

9 11 13 6 4 15

9 11 6 13 4 15

9 11 6 4 13 15

iteration 3

9 11 6 4 13 15

9 6 11 4 13 15

9 6 4 11 13 15

iteration 4

9 6 4 11 13 15

6 9 4 11 13 15

6 4 9 11 13 15

iteration 5:

4 6 9 11 13 15

Drawback:

Even it is fully (or) partially sorted it takes n^2 iterations (or) times.

```
for (i=0; i<n-1; i++)
{
for (j=0; j<n-i-1; j++)
{
  if (a[j] > a[j+1])
  {
    temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;
  }
}
}
```

4/9/13

Selection sort:- $O(n^2)$

opposite to bubble sort (finds the smallest 4 places at beginning

	0	1	2	3	4	5
a	35	11	16	10	3	19

* number of iterations = n-1;

* number of comparisons = (n-1) + (n-2) + (n-3) ... + n-(n-1).

Iteration -1:

11 35 16 10 3 19

11 35 16 10 3 19

10 35 16 11 3 19

3 35 16 11 10 19

3 35 16 11 10 19

iteration-2:

3 35 16 11 10 19

3 16 35 11 10 19

3 11 35 16 10 19

3 10 35 16 11 19

3 10 35 16 11 19

iteration 3

3 10 35 16 11 19

3 10 16 35 11 19

3 10 11 35 16 19

3 10 11 35 16 19

iteration 4:

3 10 11 35 16 19

3 10 11 16 35 19

3 10 11 16 35 19

iteration 5

3 10 11 16 35 19

3 10 11 16 19 35

i=0

j=1,0
j=2,0
j=3,0
j=4,0
j=5,0

i=1

j=2,1
j=3,1
j=4,1
j=5,1

i=2

j=3,2
j=4,2
j=5,2

i=3

j=4,3
j=5,3

i=4

j=5,4

Code:

```
for (i=0; i<n-1; i++)
{  int temp;
   for (j=i+1; j<n; j++)
   {
      if(a[i] > a[j])
      {
         temp = a[i];
         a[i] = a[j];
         a[j] = temp;
      }
   }
}
```

no need of swapping every time

```
for (i=0; i<n-1; i++)
{  int min = i;
   for (j=i+1; j<n; j++)
   {  if (a[min] > a[j])
         min = j;
   }
   temp = a[i];
   a[i] = a[min];
   a[min] = temp;
}
```

Insertion Soot :- O(n²) :-

* Improvement of Bubble & selection sorting algorithms
* takes n^2 times only when elements are fully unsorted.

	0	1	2	3	4	5
a	32	19	55	42	13	9

number of iterations : n-1 :

number of comparisons : $n-(n-1)+(n-2)+(n-3)+\dots+(n-(n-1))$

$\dots + (n-1)$.

iteration 1 :-

32 19 55 42 13 9

19 32 55 42 13 9

iteration 2 :

19 32 55 42 13 9

19 32 55 42 13 9

iteration 3 :

19 32 55 42 13 9

19 32 42 55 13 9

iteration 4 :

19 32 42 55 13 9

13 19 32 42 55 9

iteration : 5 :

13 19 32 42 55 9

9 13 19 32 42 55

s/g

```
for (i=1; i<n; i++)
{
  for (j=0; j<i; j++)
  {
    if (a[j] > a[i])
      temp = a[j];
      a[j] = a[i];
      for (k = i; k > j; k--)
      { a[k-1] = a[k]; } a[k+1] = temp; } }
```

NOTE 2: You are allowed to move next level only when current level is complete

property of B.T:

can hold only two childs (0 or 1)

* Complete binary tree if all nodes are attached with element

$2^n - 1$ inputs

n → no. of levels

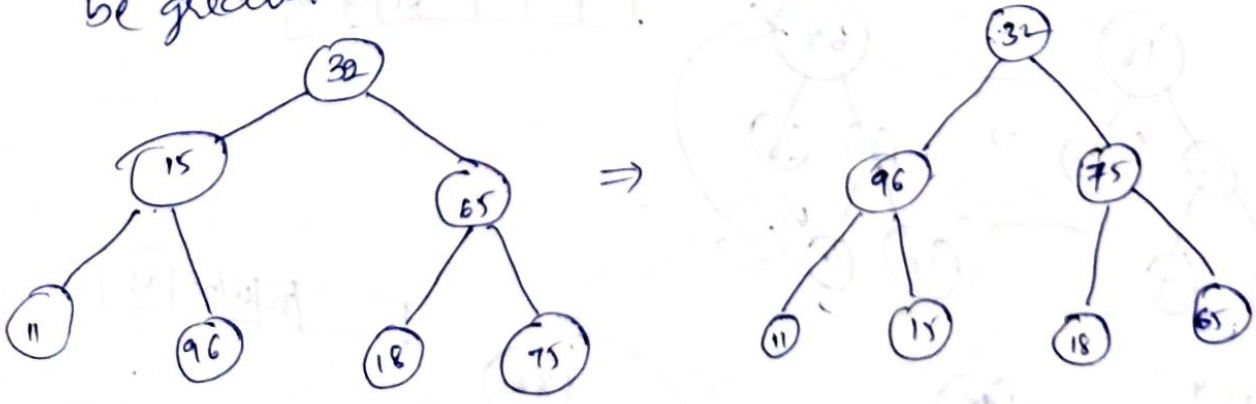
if not ($2^n - 1$ inputs) → incomplete binary tree

* attach dummy nodes if ($2^n - 1$) inputs not given.

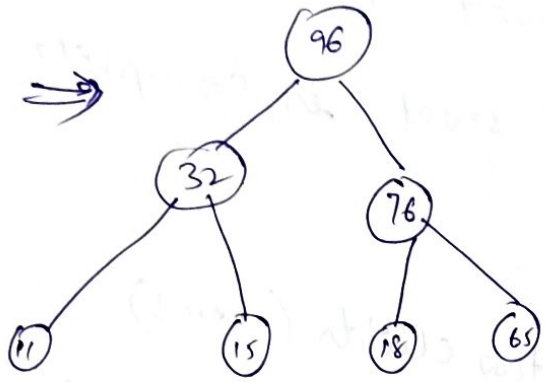
Step 2:

Heap tree:

In Heap tree the root node will be greater than its left & Right children.



Binary Tree



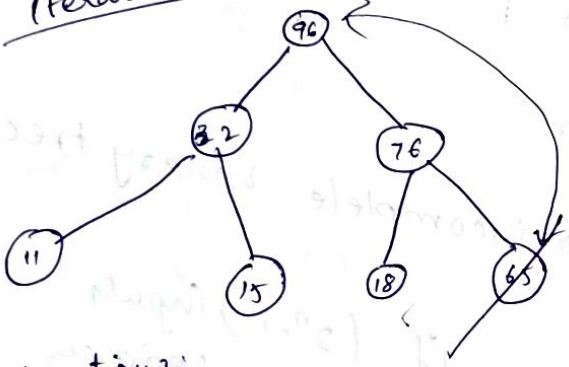
Heap Tree / Max Tree

* continue upto no. of nodes in tree becomes "2"

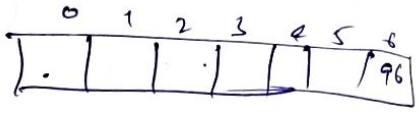
Parent: node having min 1 degree other than root

Leaf: node having 0 degree

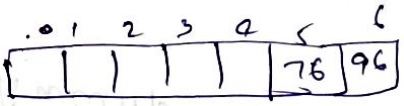
Iteration 1:

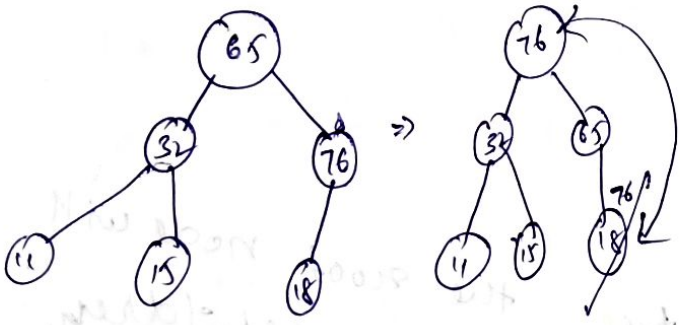


output array:

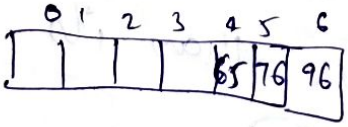


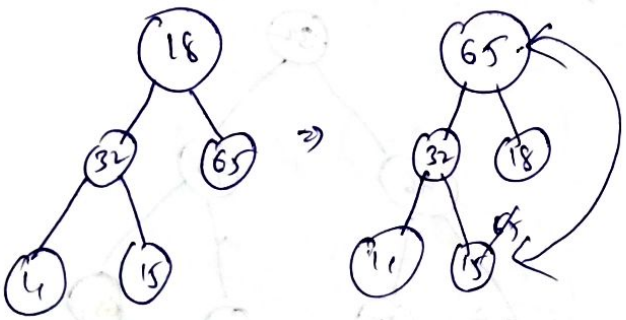
iteration 2:



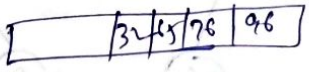


i=3

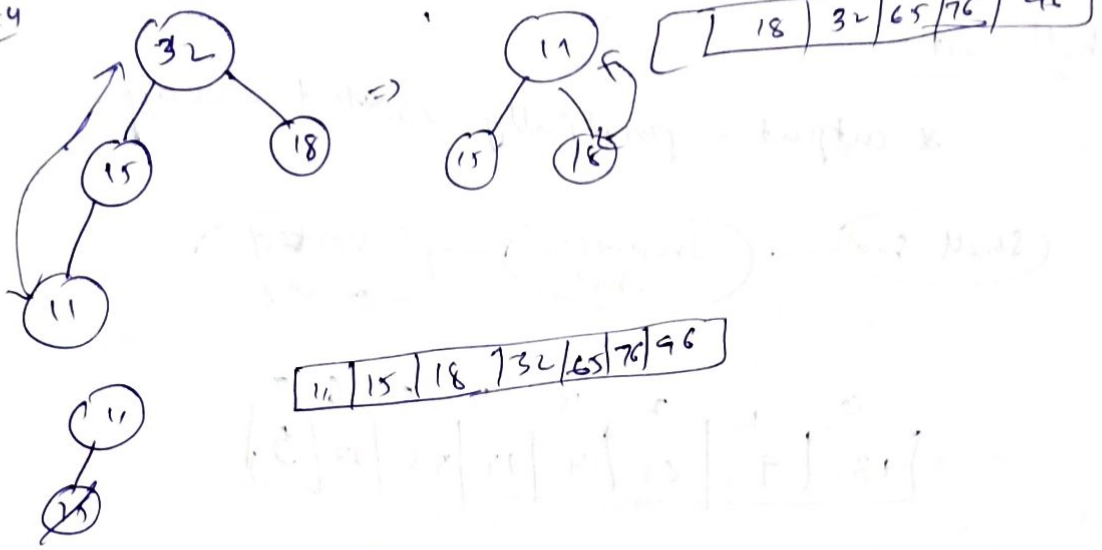




=> i=4



i=4



LDR
DLR
LRD

LDR - 11, 15, 96, 32, 18, 65, 75

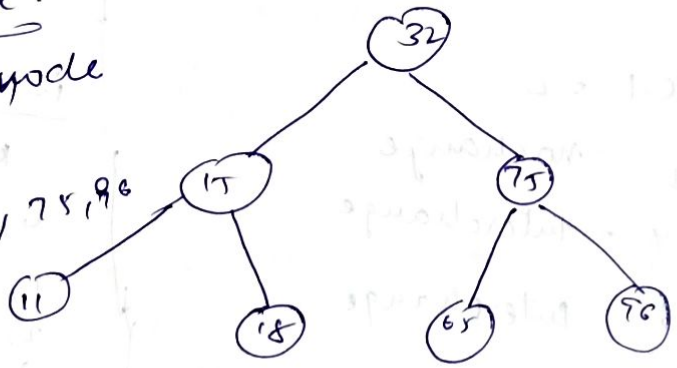
DLR - 32, 15, 11, 96, 65, 18, 75

LRD - 11, 96, 15, 18, 75, 65, 32

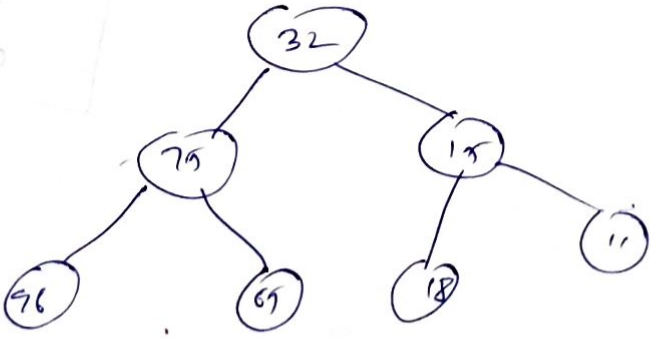
Binary Search Tree:

leftnode < root < rightnode

11, 15, 18, 32, 65, 75, 96



descending order



Shell sort:

* output - partially sorted array

0	1	2	3	4	5	6	7
13	9	65	14	19	53	23	5

no. of iterations: $max = n/2$

$K = n/2$ (distance)

$K = 4$

i=1

1) 0, 4, no change

i=2

2) $K = K-1 = 3$

0, 3 - no change

3, 6 - no change

i=3

$K = K-1 = 2$

0, 2 - no change

2, 4 - interchange

4, 6 - interchange

i=4

$K = K-1 = 1$

0, 1, — interchange

1, 2 — no change

2, 3 — interchange

3, 4 — no change

4, 5 — no "

5, 6 — " "

6, 7 - interchange

19/9/13

Hashing:

(no implementation is given in any text-book)

Hashing is the way to reduce the no of comparisons in large databases.

Hash function:-

A function which is used to return the address of given record is called as "Hash function".

Hash key:-

The ~~func~~ value which is returned by the hash function is called hash key

Collission:-

when two diff records demands for same index position is called collission.

Q. Construct Hash table for the following input

73, 55, 69, 78, 45, 10, 67, 85, 42, 36

using "linear probing", "chain without replacement" & "chain with replacement" methods.

Hash function :- H(I) = I/P % 10 :

Hash Table using ~~index~~ linear probing

H(key) = 73 % 10 = 3 (key)

= 55 % 10 = 5 (key)

= 69 % 10 = 9

= 78 % 10 = 8

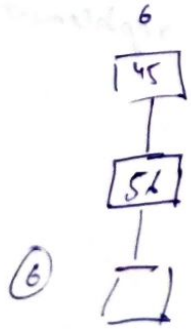
= 45 % 10 = 5

collision occurs (To ~~avoid~~ so scan the table & store where first free record available)

index position	Id	Record
0	45	
1	10	
2	85	
3	73	
4	42	
5	55	
6	36	
7	67	
8	78	
9	69	

Hash table using "chain without replacement"

* chain is used to keep track of element of same key

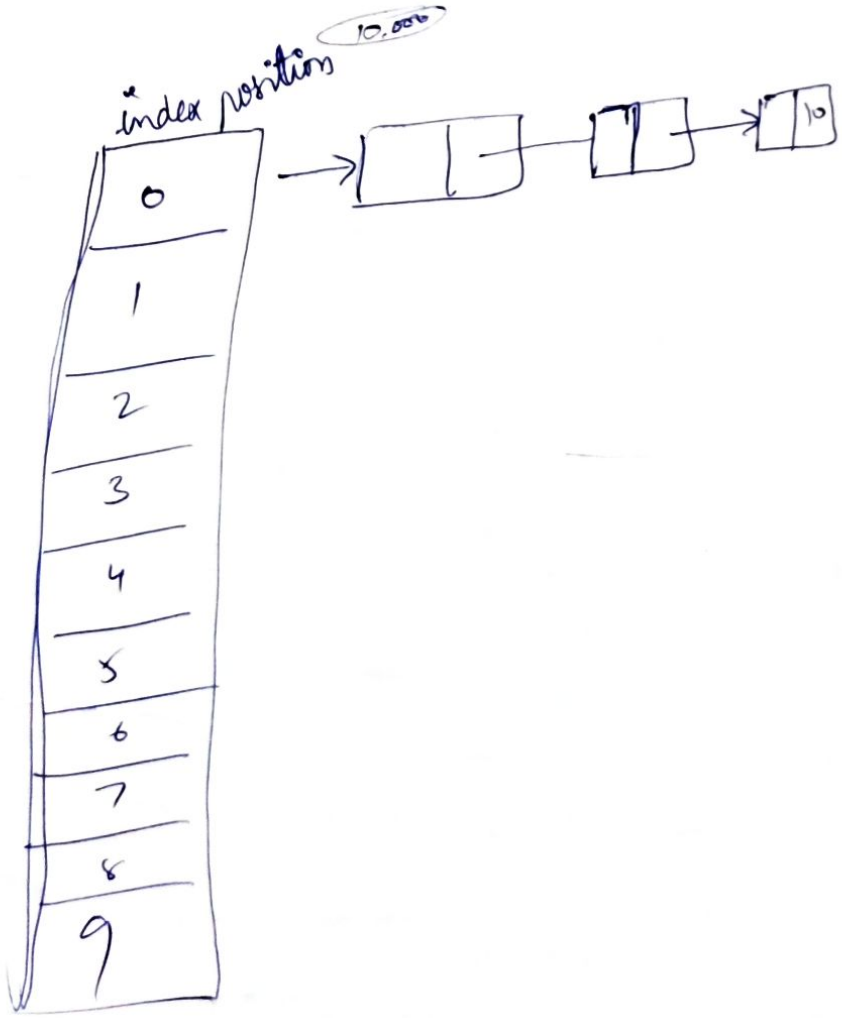


index position	Id	chain	Record
0	45	7	
1	10	-1	
2	85	-1	
3	73	-1	
4	42	-1	
5	55	0	
6	36	-1	
7	67	-1	
8	78	-1	
9	69	4	

Chain with replacement

* when digital data comes move the stored data one position forward.

Index no	Id	chain	Record
0	10	1	
1	45	4	
2	42	-1	
3	73	-1	
4	85	-1	
5	55	-1	
6	36	-1	
7	67	-1	
8	78	-1	
9	69	-1	



Sorting Algorithm	Best	Worst
Bubble sort	n^2	n^2
Selection sort	n^2	n^2
Insertion sort	n	n^2
Shell sort	$n \log n$	$n \log^2 n$
Merge sort	$n \log n$	$n \log n$
Quick sort	$n \log n$	n^2
Heap sort	$n \log n$	$n \log n$
Radix sort	n	n

Unit - 5:

Tree and Graphs

→ Trees

→ Tree Terminology

→ different types of trees

→ tree representation — array, linked list

→ tree traversals — in-order, pre-order, post-order

→ applications of tree — searching, sorting, Expression conversion.

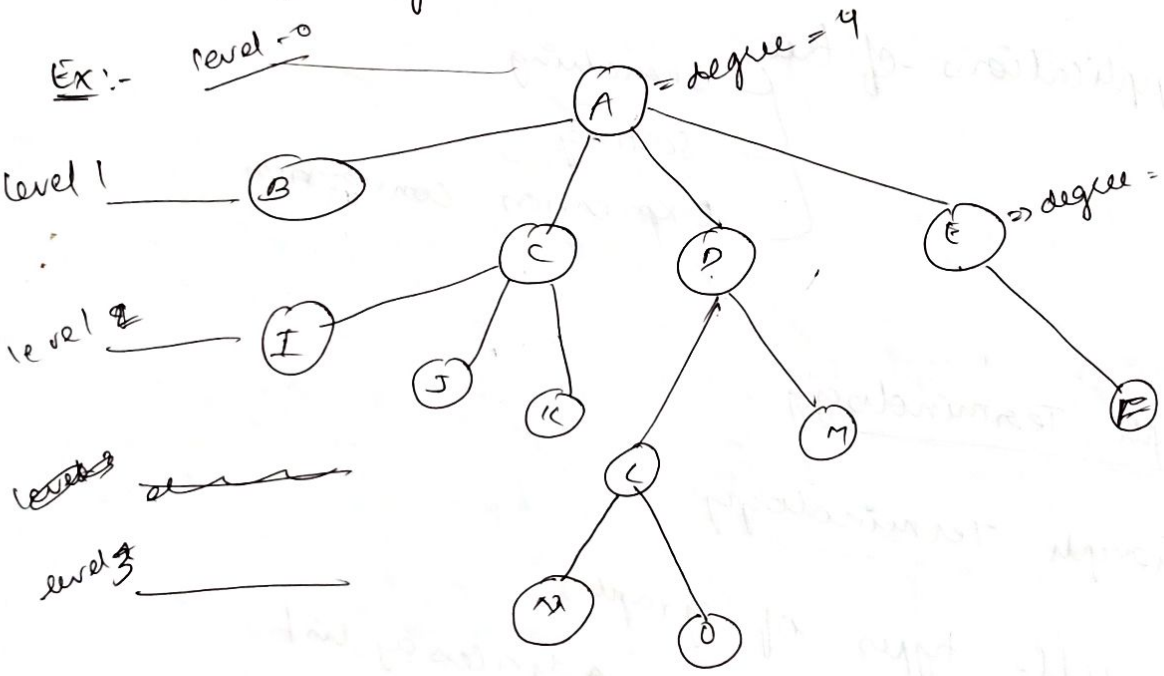
→ Graph Terminology:-

1. Graph Terminology
2. diff. types of graph
3. Graph representation — Adjacency list, Adjacency
4. Graph transversal — Breadth first, Depth first
5. Application — shortest path - Dijkstra's; M → maximum; (M)ST → minimum spanning tree - Prim's, Kruskal's

General Tree:

Tree is a collection of nodes where there is a special designated node called a root. in which further branches are attached.

In general tree a node can have maximum of 'n' children.



Parent node:-

Node which ever is having minimum one degree other than root is called "parent".

Ex:- root - A

Parents: C, D, E, L

leaf node:

Node whichever is having zero degree is called "leaf Node".

Ex: leaf: B, I, J, K, N, O, M, F.

degree of a node:

No. of childrens attached to a particular node is called degree of the node:

A - degree: - 4
C - " - 3
E - " - 1
F - " - 0

Degree of the tree:

The maximum no. of children attached particularly to a node

Ex. A - 4

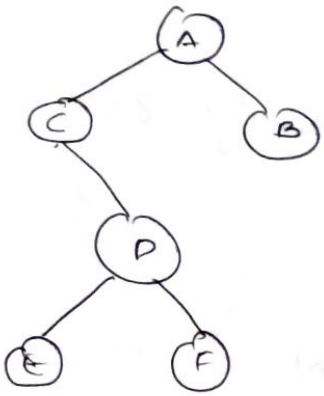
Height of the Tree:

The no. of levels present in the tree is called "height" of the Tree

Binary tree:

In B T a node can have max. of two children

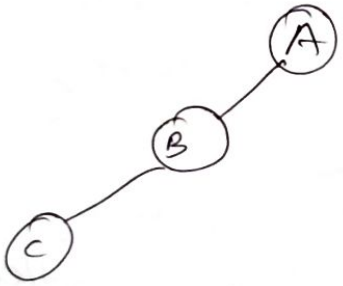
Ex:



→ left skewed B.T:-

The absence of right child in each level is called "left skewed B.T".

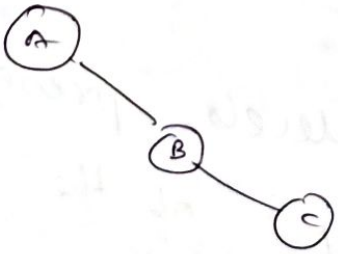
Ex:-



→ Right skewed B.T:-

The absence of left child in each level is called "Right skewed B.T".

Ex:



→ Complete Binary Tree:-

In Complete B.T there will be $(2^n - 1)$ nodes where 'n' represents no of levels.

→ Binary Search Tree :-

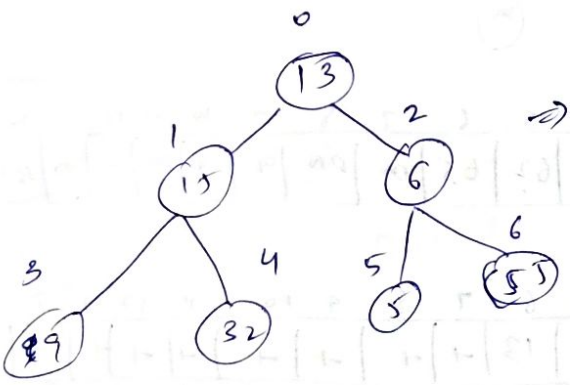
left child < parent (root) < Right child

25/9

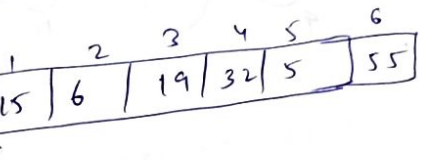
Tree Representation :-

13, 15, 6, 19, 32, 5, 55

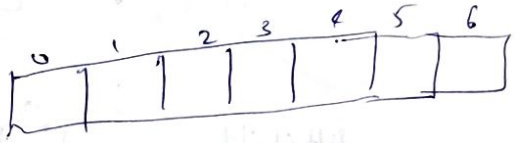
Array Representation:



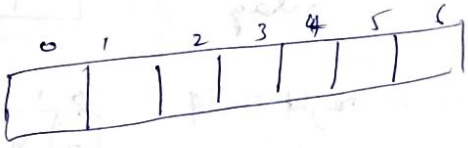
⇒ root

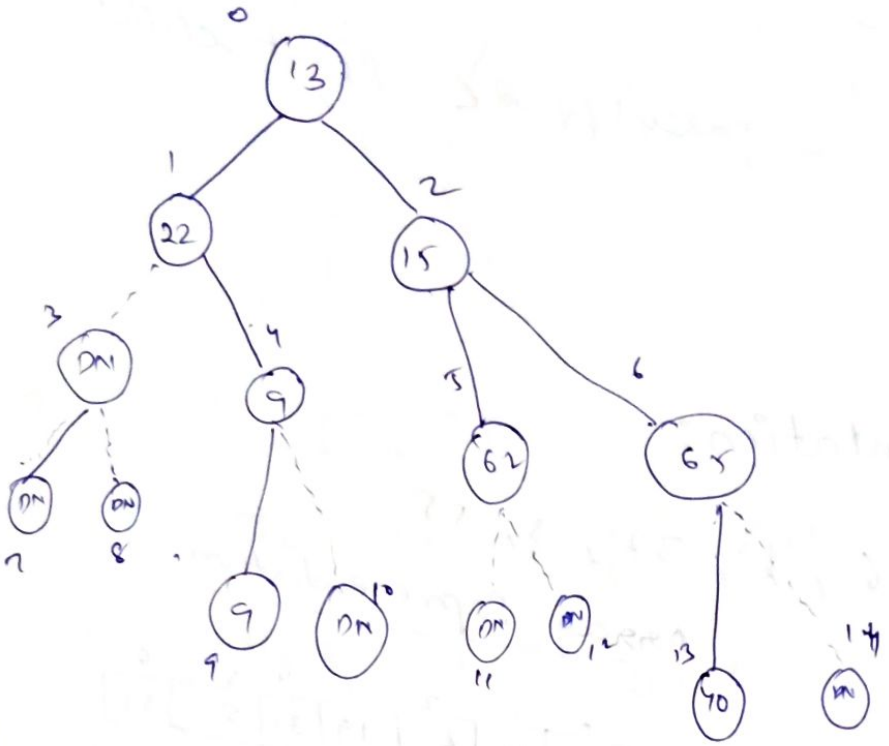


⇒ left child array



right child array





root array

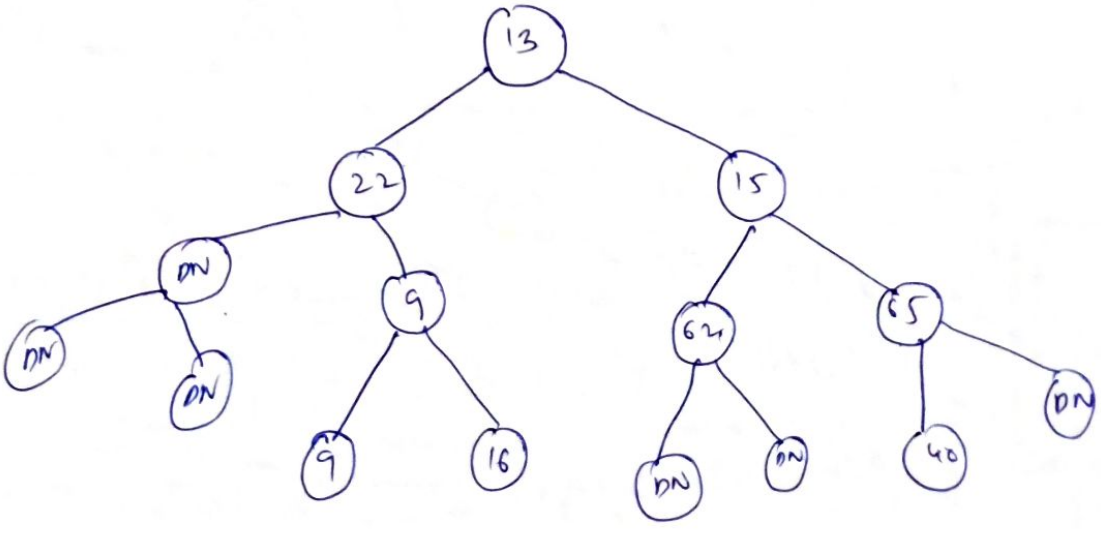
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	22	15	DN	9	62	65	DN	DN	9	DN	DN	DN	40	DN

left child array

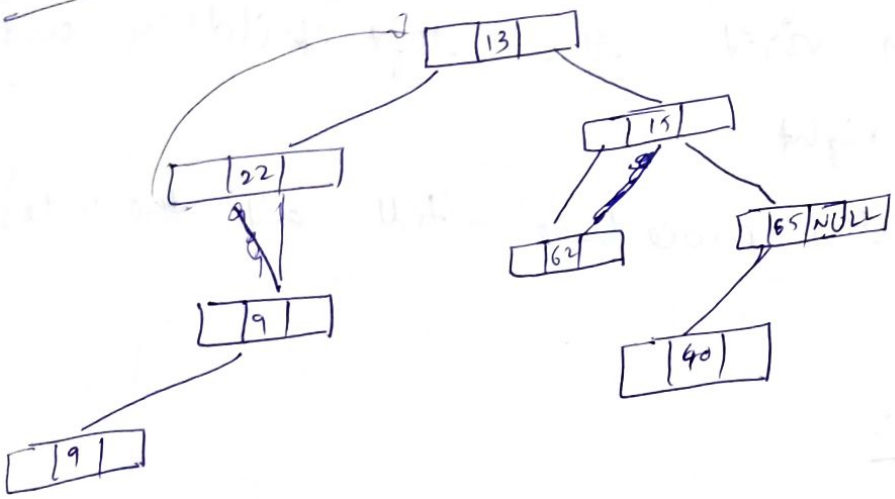
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	5	7	9	11	13	-1	-1	-1	-1	-1	-1	-1	-1

right child array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	4	6	8	10	12	14	-1	-1	-1	-1	-1	-1	-1	-1

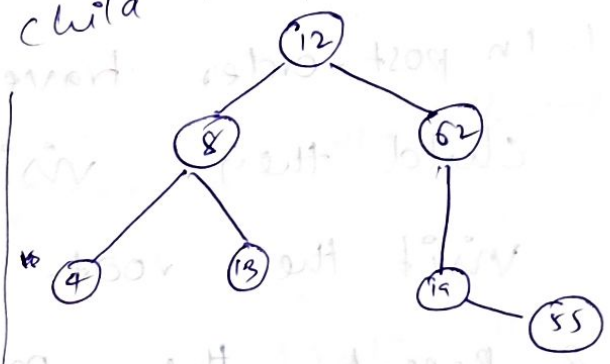


Linked - list Representation:-



* Threaded binary Tree:-

Thread from child to parent node



26/9

→ Binary Tree Traversal:-

1. In-order (LDR)
2. pre-order (DLR)
3. post-order (LRD).

Inorder: 1. In the Inorder traversal first visit the leftmost children then root finally visit the right child

2. Repeat the procedure till all the nodes are visited.

In-order: 4, 8, 13, 12, 19, 55, 62

Pre-order: 1. In pre-order traversal first visit the root then visit the left children and finally visit the right.

2. Repeat the procedure till all the nodes are visited.

Pre-order:

12, 8, 4, 13, 62, 19, 55

Post-Order:-

1. In post-order traversal first visit the leftmost child then visit the right child finally visit the root.

2. Repeat the procedure till all the nodes are visited

Post order:

4, 13, 8, 55, 19, 62, 12

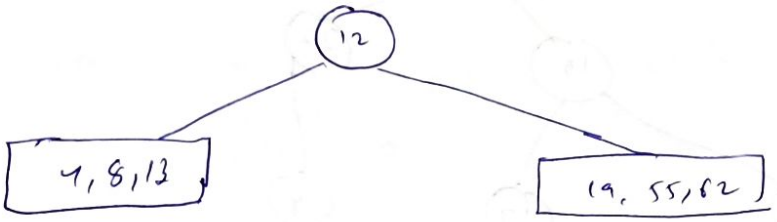
* Reconstruction is possible only when In-order and pre-order (or) inorder and post order are given

* If only post order (or) pre-order is given reconstruction is not possible

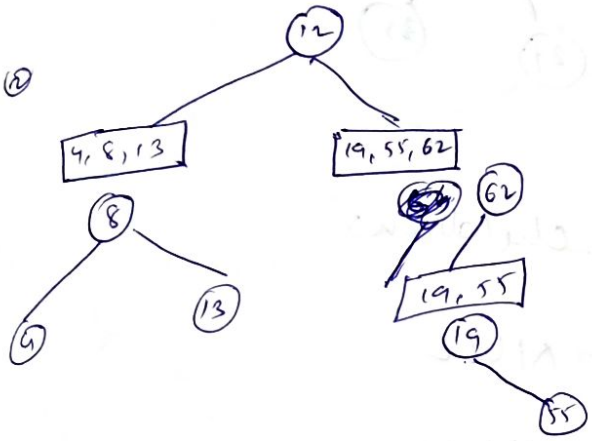
Q. In-order :- 4, 8, 13, 12, 19, 55, 62 (LDR)

pre-order : 12, 8, 4, 13, 62, 19, 55 (DLR)

(1)



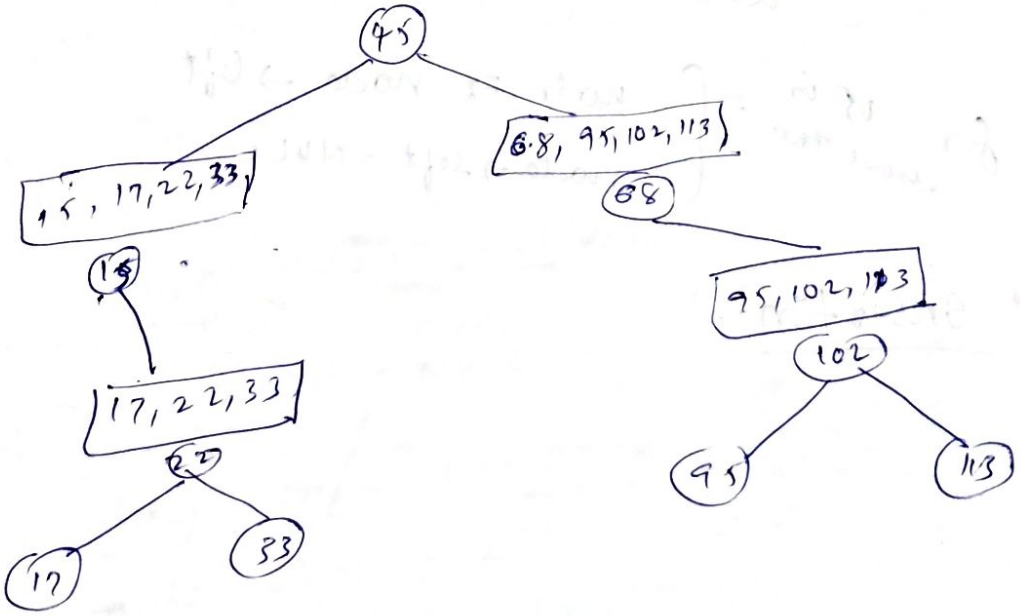
(2)



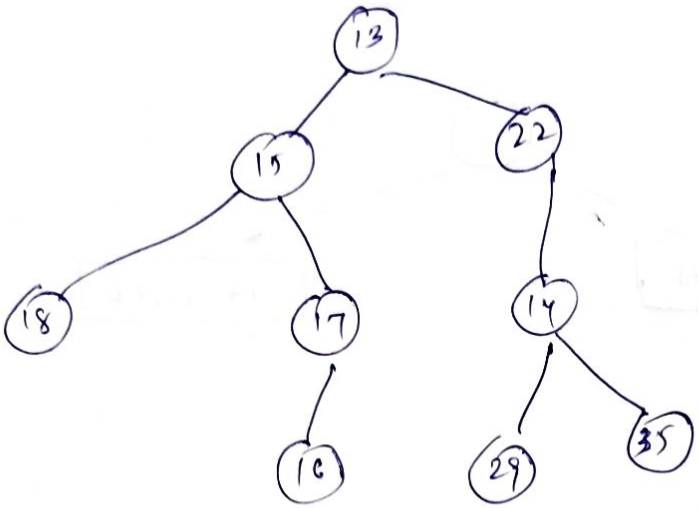
1. Reconstruct the binary tree from the following tree traversals

In-order :- 15, 17, 22, 33, 45, 68, 95, 102, 113

post order : 17, 33, 22, 15, 95, 113, 102, 68, 45



Operations on Binary Tree:-



Deletion:-

i) node having zero children:

parent → left = NULL.

ii) node having one children:

node = node → left

node → left = NULL.

iii) node having two children:-

whichever takes at less cost

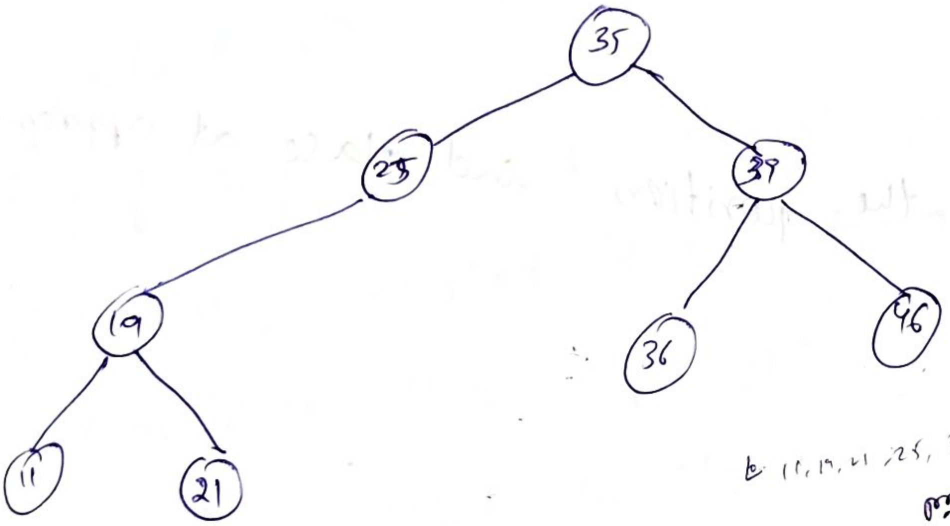
for 15 in above tree ⇒ { node = node → left
node → left = NULL

→ Insertion:-

Q. construct binary search Tree for the following input:

35, 25, 19, 39, 46, 21, 11, 36 and

show how binary search is used for searching & sorting.



11, 19, 21, 25, 35, 36, 39, 46

pre order: DLR

In-order: → ascending order

(LDR)

11, 19, 21, 25, 35, 36, 39, 46.

* also used in searching.

Deletion:-

i) node having zero children:

parent → left = NULL

ii) node having one children:

node = node → left

node → left = NULL

iii) node having two children:-

node = node → left

node → left = NULL

iv) Parent (special) node : here (33) is not allowed to delete

Insertion :-

* Identify the position and place at appropriate position.

2/10

Applications of Binary tree :-

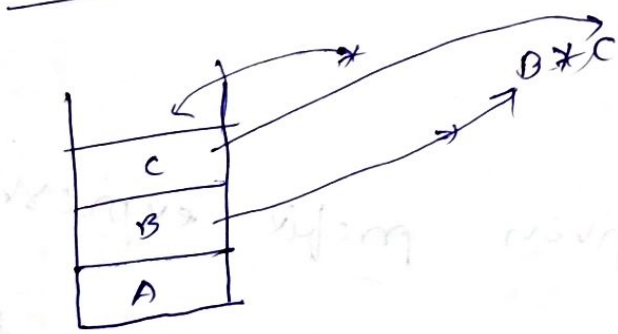
* searching
* sorting
* exp- conversion.
* exp-Tree cannot be constructed using infix exp.
* can be converted only if prefix (or) postfix form is given.

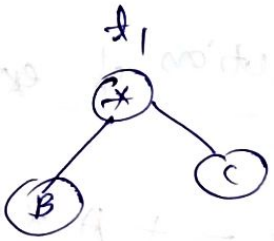
postfix : ABC*+D-

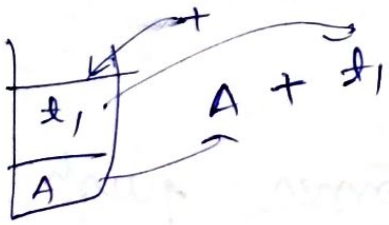
Algorithm for const.

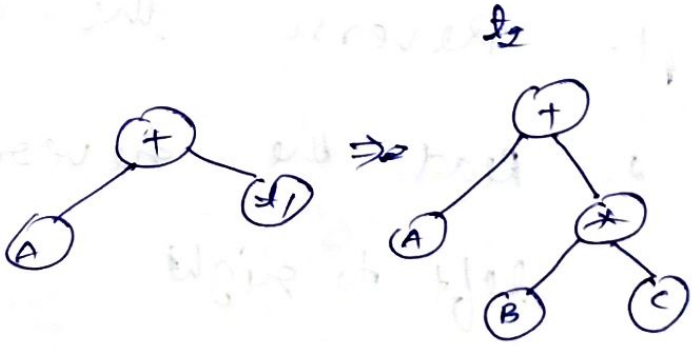
1. Read the postfix expression from left to right
2. If any operand comes push it on the stack.
3. If any operator comes pop the two recently pushed items from stack construct the expression tree and push the tree back to stack.
4. Repeat step '2' & '3' till you reach the end of input.

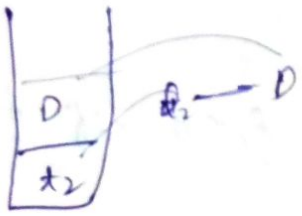
ABC*+D-

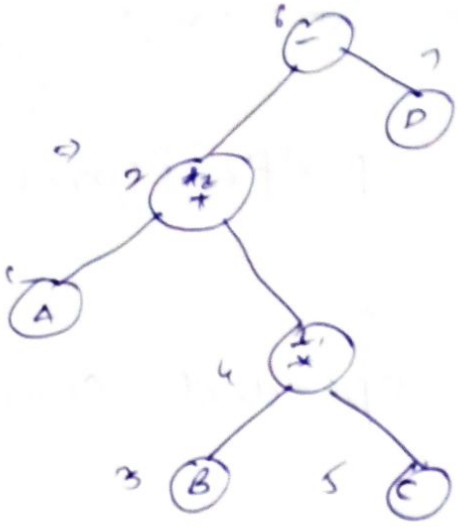












Inorder: (LDR)

↓

Infix:

A + B * C - D

pre-order (DLR)

↓

prefix : - + A * B C D

post-order:

postfix : A B C * + D -

* Construction of expression tree from prefix exp:

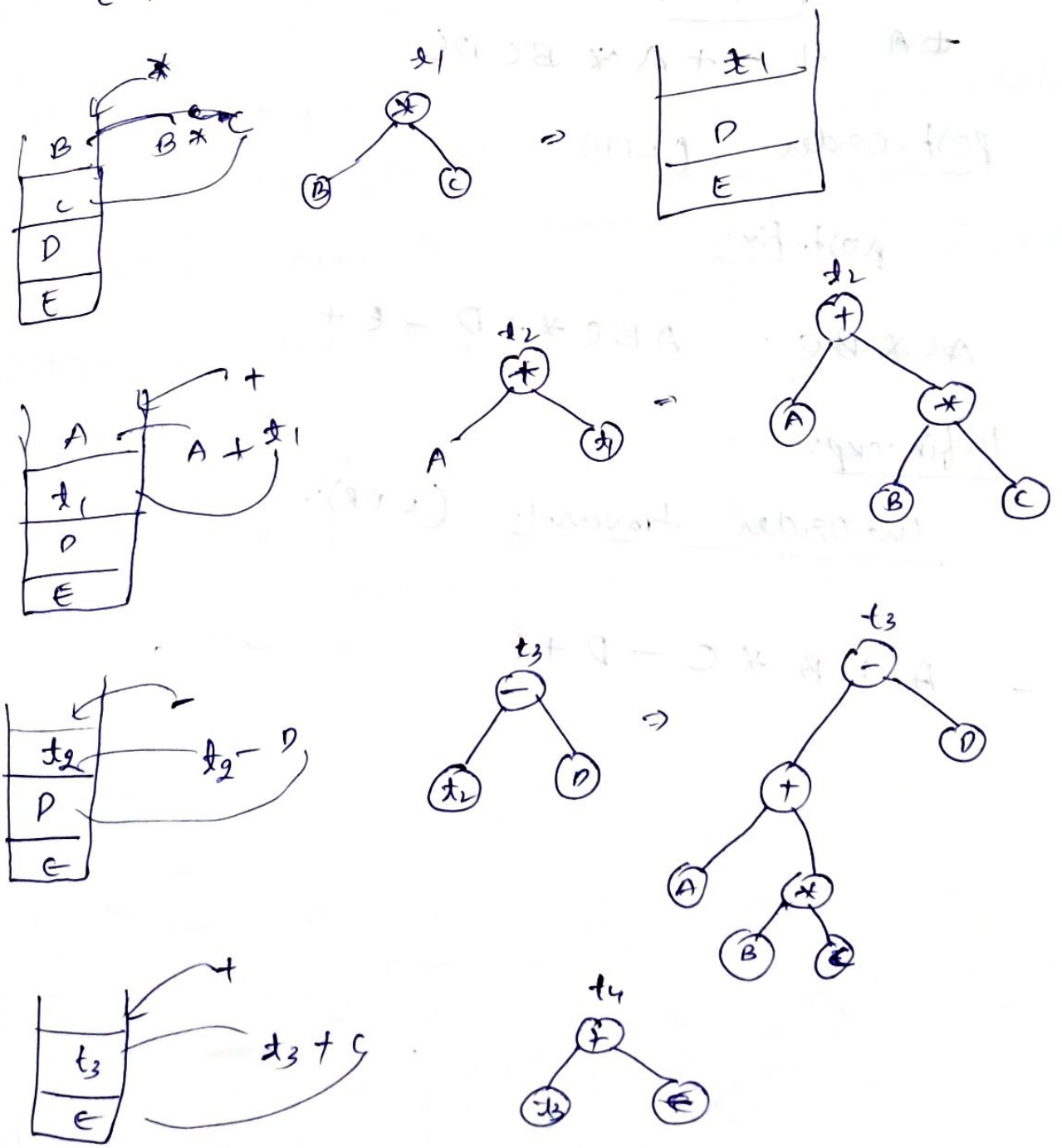
* + - + A * B C D E

1. Reverse the given prefix expression.
2. Read the reversed prefix exp from left to right
3. If any operand comes push it on the stack.

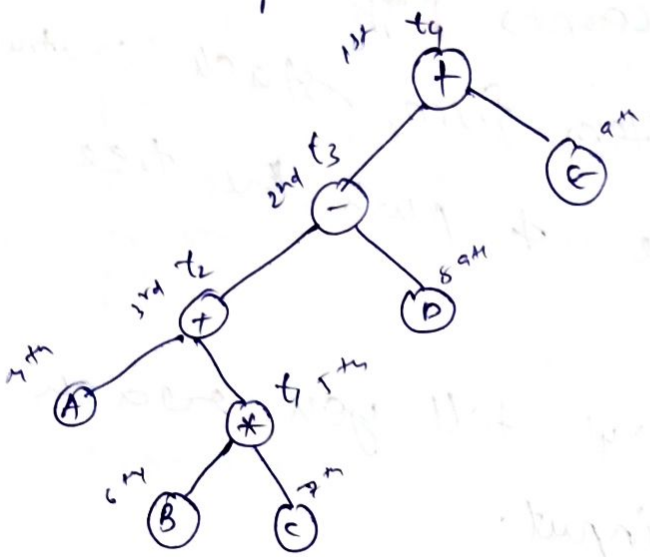
4. If any operator comes pop the two recently pushed item from stack construct the expression tree & push the tree back to stack

5. Repeat step '3' & '4' till you reach the end of the input.

E D C B * A + − +.



⇒ Expression tree



pre-order traversal: (DLR)

prefix exp:

+ - + A * B C D E

post-order (LRD)

post-fix:

A B C * + D - E +

Infix exp:

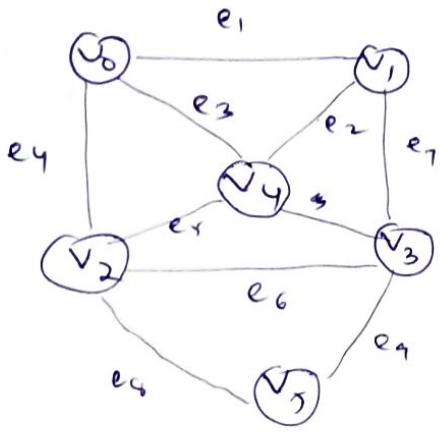
In order traversal: (LDR):

A + B * C - D + E.

Graph:

Any graph, G is defined as finite non empty sets of edges and vertices

$G = \{e, v\}$

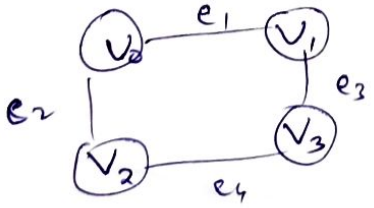


Undirected Graph:

In this type of graph the distance between the vertices and the direction will not be shown on the edges. If their exist an edge from v_i, v_j

v_i, v_j

v_j, v_i



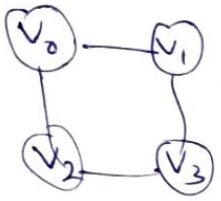
$G = \{V, E\}$

In this type of graph the direction between the vertices will be shown on the edges but not distance. If there exist the

2.

Graph Representations :-

Adjacency Matrix Representation :-

G = 

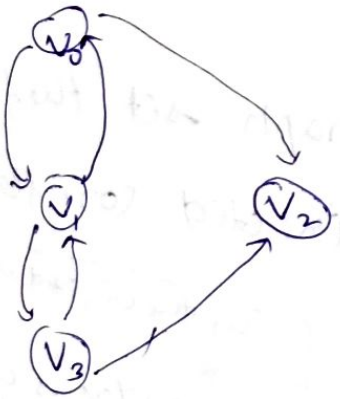
Order of matrix = no. of vertices

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

4 × 4

If there is a path mark '1' else mark it '0':

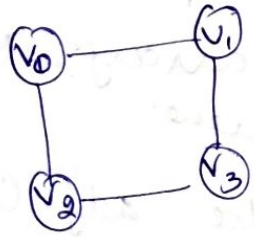
Ex: $V_0 - V_0 = 0$
$V_0 - V_1 = 1$

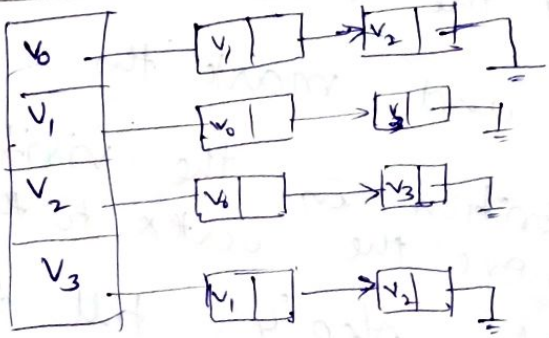


	V_0	V_1	V_2	V_3
V_0	0	1	1	0
V_1	1	0	0	1
V_2	0	0	0	0
V_3	0	1	1	0

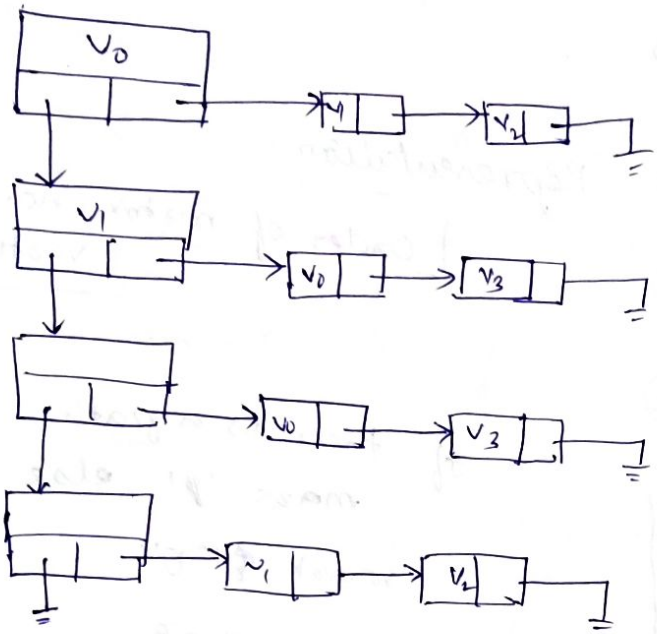
* drawback: static data structure

Adjacency list Representation:-

G = 



4 × 2




```
struct head
{ struct head *adj;
  struct head *next;
  int data;
};
struct adjnode
{
    int data;
struct adjnode *next;
};
```

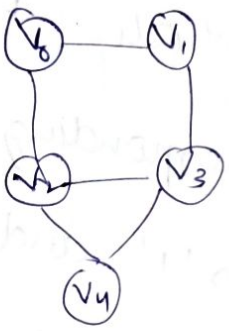
Graph Traversals:

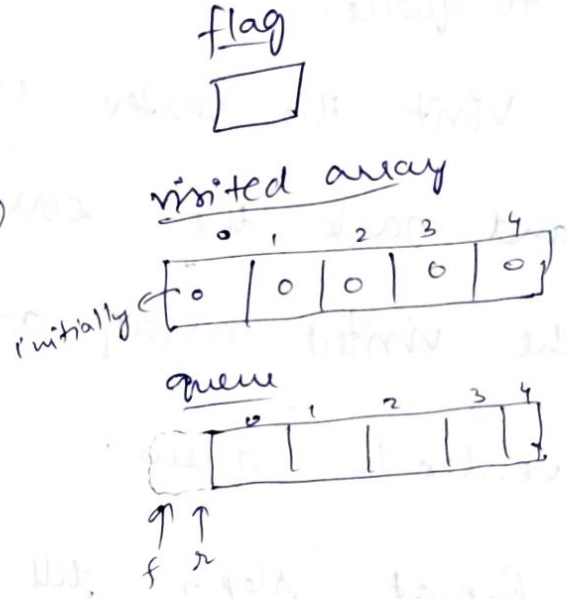
DFT: (Back tracking).

1. Create a graph
2. Depending upon the type of graph set the flag equal to zero if it is undirected (or) flag=1 if it is directed.
3. Start the traversal from any vertex say V_i and mark the corresponding index position in the visited array as '1' and move the vertex to queue.
4. Visit the vertex which is adjacent to 'V_i' say 'W_i' and mark the corresponding index position in the visited array as 1. and move the vertex to the queue.
5. Repeat step '4'. till all the adjacent vertices of 'V_i' are visited
6. On reaching a vertex whose all adjacent vertices are visited go back to the vertex

whichever is having unvisited vertex and repeat step '4' & '5'

7. If all the index positions of the visited array becomes '1'. stop the traversal and display the 'queue' elements





21/10/13

the vertex to queue'

DFT:

1. Create a graph
2. Depending upon the type of graph set the flag equal to zero if it is undirected (or) flag if it is directed
3. Start the traversal from any vertex say "v_i" and mark the corresponding index position in the visited array as '1' and move the vertex

to 'queue'

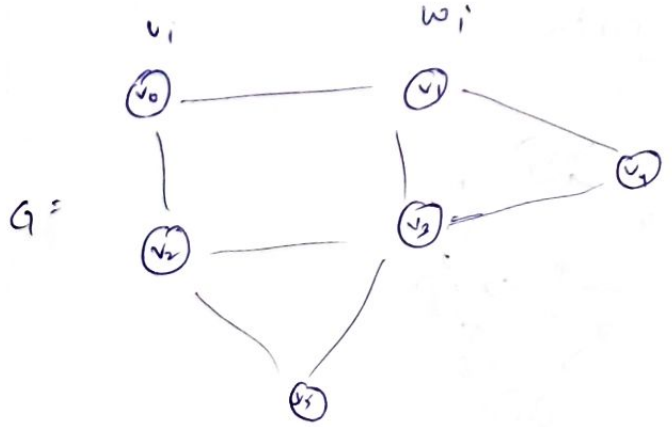
4. Visit the vertex 'Wi' which is adjacent to 'Vi'. and mark the corresponding index position in the visited array as '1'. and move the vertex to queue.

5. Visit the vertex which is adjacent to 'Wi'. and mark the corresponding index position in the visited array as '1' and move the ~~index~~ vertex to queue

6. Repeat step 5 till all the vertices are covered

7. On reaching a vertex whose all adjacent vertices are visited go back to the vertex whichever is having unvisited and repeat 'Step 4' & 'Step 5'.

8. if All the index positions of the visited array becomes '1' stop the traversal and display the queue



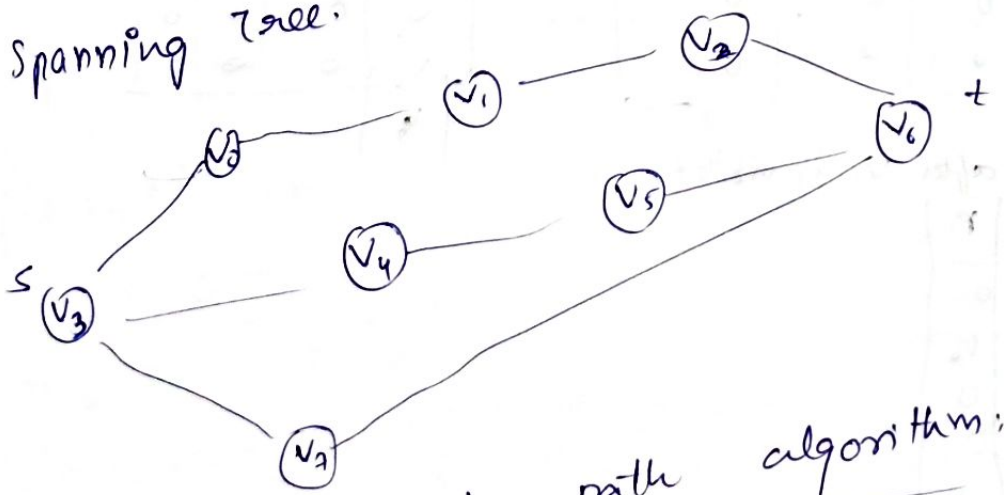
flag [: 0]

visited array

Shortest path Shortest path Spanning Tree

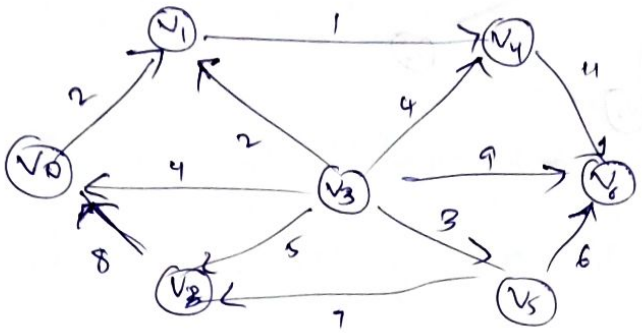
Applications of Graph:

1) Shortest path
2) Spanning Tree.



General shortest path algorithm:

Dijkstra's shortest path:



Condn:-

1. Only directed & weighted graph
2. should not be any negative weights on edge
3. should be atleast one outgoing edge.

Initial configuration table → from source to destination

distance b/w vertices → d_v

predecessor → p_v

parameters which cause change in d_v

v	k	d_v	p_v
v_0	0	0	0
v_1	0	∞	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0

after v_0 is visited

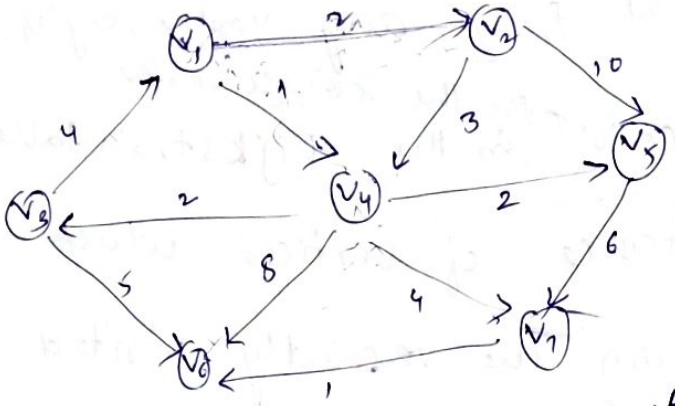
v	k	d_v	p_v
v_0	1	0	0
v_1	0	2	v_0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0

after v_1 is visited

v	k	d_v	p_v
v_0	1	0	0
v_1	1	2	v_0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	11	v_1
v_5	0	∞	0
v_6	0	∞	0

Algorithm

1. start the traversal from any vertex say V_i.
2. mark the distance '0' for the source vertex in the Dijkstra's table
3. Update the distance of vertices which are reachable from the recently visited vertex only if the current distance is less than the previous distance.
4. Select the unvisited vertex 'w' which is adjacent to the recently visited vertex whichever is having less distance.
5. Repeat step '4' till all the vertices are visited (or) covered.
6. If all the vertices are visited construct the spanning tree from Dijkstra's table by considering only the edges which are present in the Dijkstra's table.



single source shortest path algorithm:-

works on Greedy - algorithm

Initial config table

v	k	dv	pv
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

after v_1 visited

v	k	dv	pv
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

after v_4 visited

v	k	dv	pv
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	2	v_4
v_6	0	8	v_4
v_7	0	4	v_4

after v_2 visited

v	k	dv	pv
v_1	1	0	0
v_2	1	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	2	v_4
v_6	0	8	v_4
v_7	0	4	v_4

after v_3 is visited

v	k	dv	pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	2	v_4
v_6	0	5	v_3
v_7	0	4	v_4

after v_5 is visited

v	k	dv	pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	2	v_4
v_6	0	5	v_3
v_7	0	4	v_4

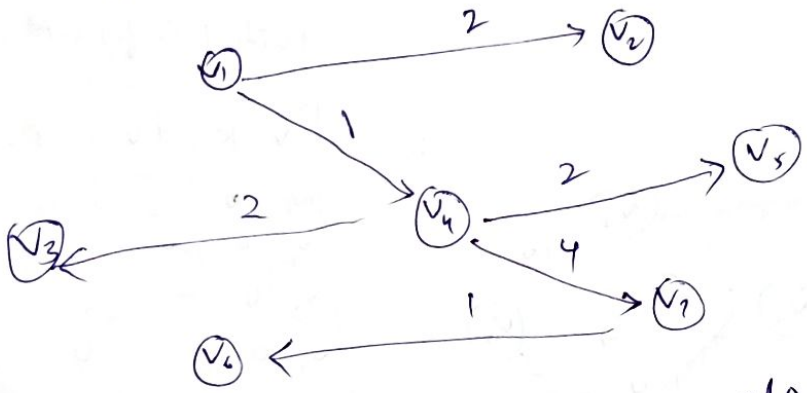
after v_7 is visited

v	k	dv	pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	2	v_4
v_6	0	1	v_7
v_7	1	4	v_4

v	k	dv	pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	2	v_4
v_6	1	1	v_7
v_7	1	4	v_4

Spanning Tree → w (shortest path)

Spanning Tree 'G' of a graph, G is also a graph in which it will have all the vertices but not all the edges



* do not produce any closed circuit. because it produces more than one route.

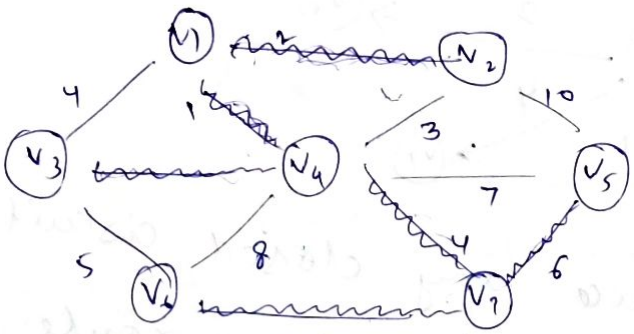
Minimum Spanning Tree:

Prim's Minimum Spanning Tree Algorithm:

1. Start the traversal from any vertex say v_i. and mark the distance source to source itself '0'. and source to all other vertices as '∞'.
2. Update the distance of vertices which are reachable from source only if the current distance is less than the previous distance.
3. Select the unvisited vertex w_i which is adjacent to 'v_i' whichever is having less distance
4. adjacent to 'v_i'. & Repeat step '2'.

4. Repeat Step '2' & '3' till all the vertices are covered.

5. If all the vertex becomes visited stop the traversal and construct the spanning tree from the prim's table



Initial Configuration

V	K	dV	PV
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

after 'v_1' is visited

V	K	dV	PV
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

after 'v_4' is visited

V	K	dV	PV
v_1	1	0	0
v_2	1	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

after 'v_3' is visited

V	K	dV	PV
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

after 'v_2' is visited

V	K	dV	PV
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

$v_1 - v_2 - 2$
$v_1 - v_3 - 3$
$v_1 - v_4 - 1$
$v_1 - v_5 - 11$
$v_1 - v_6 - 6$
$v_1 - v_7 - 5$

$1+2+4+6+4 = 16$

$2+2+1+6+1+4 = 16$

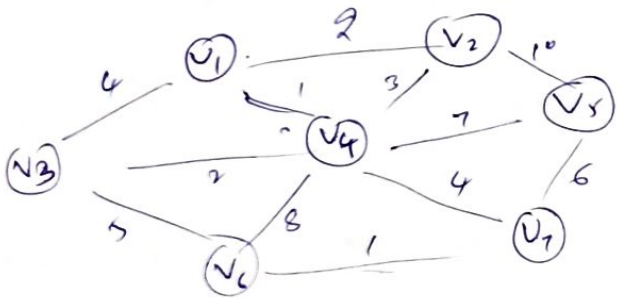
Kruskal - Minimum Spanning Tree;

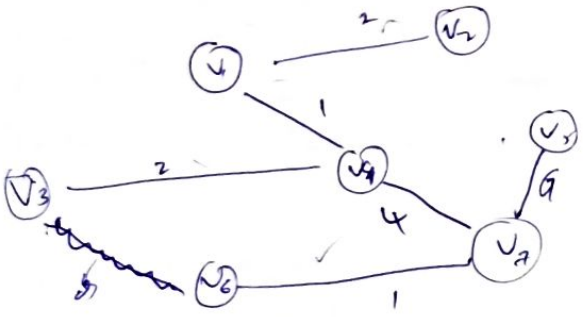
↳ mathematician

Algorithm:

This algorithm is considered as

1. Construct a spanning tree by including the edges in ascending order
2. If the inclusion of the edge forms any closed circuit. then reject the cycle.
3. Repeat 'step 1' & 'step 2' till all the vertices are connected.





* Including kruskal table for every action

* why Kruskal is not used widely?

UNIT-1

Problem solving approaches

→ Asymptotic Notations

→ time complexity Analysis

→ Space complexity.

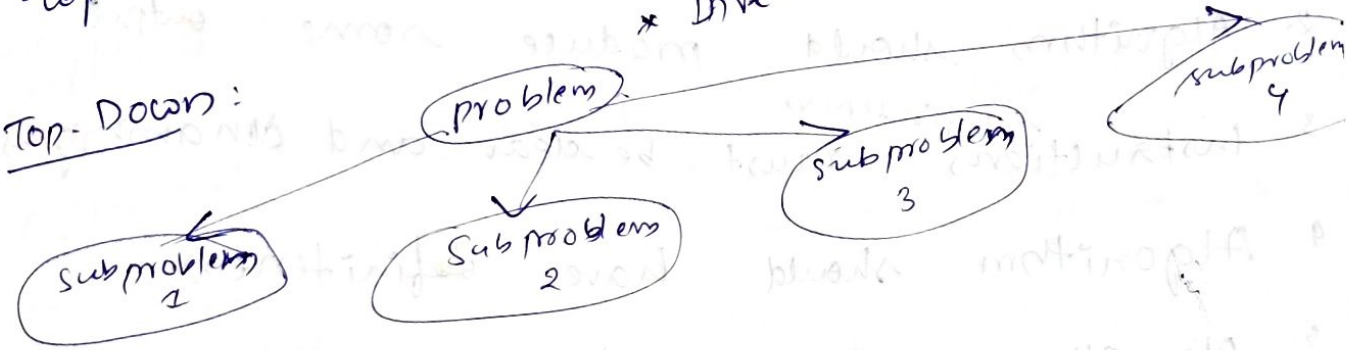
→ Problem solving Approaches:

top down

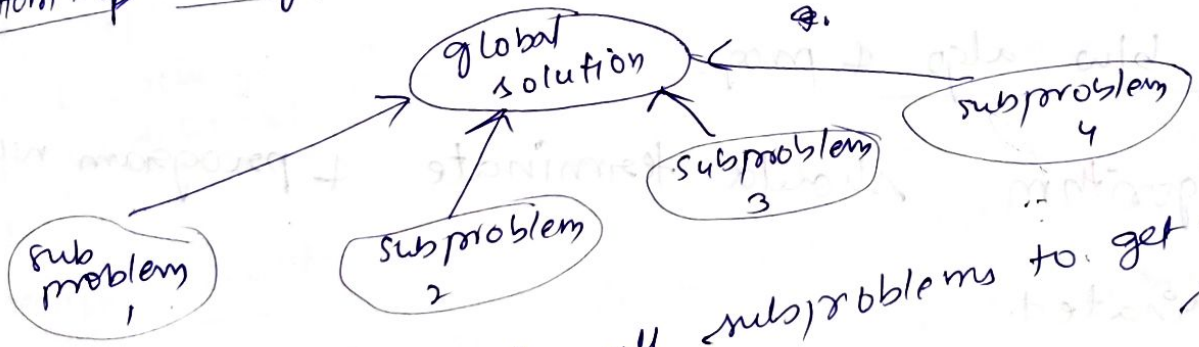
Bottom up

Top-Down:

* Divide a problem into subproblems.



Bottom-up-Design:



* Integrate all subproblems to get global solution

1. Write an algorithm for finding given no is odd (or) even.

6/11/13

Algorithm Analysis:-

Need for algo:

1) algo can convert into any prog. lang
2) effic. of prog ↑ ∝ eff of algo. ↑ not versa

chara. of good Algo:

1. Algorithm should accept some input
2. Algorithm should produce some output
3. Instructions → definiteness must be clear and unambiguous
4. Algorithm should have definiteness.
5. Algorithm should terminate.

Diff b/w algo & prog:

Algorithm should terminate & program needn't terminated.

* data type should be discussed in algo

Algo for odd (or) even

① → bad

1. Take the input
2. divide by 2
3. If rem is '0' then print even
4. else print odd

② ⇒ good

1. Declare the variable 'n' as integer
2. Get the input in variable
3. if n%2 == 0
4. print n is even
5. else n is odd.

Definiteness:

How to choose best algorithm:

1. Time complexity (execution time / Runtime)
2. Space complexity

Frequency count:

frequency - no. of executions

```
if (n%2 == 0)
printf("n is even");
else
printf("n is odd");
```

Differentiate executable & non-executable statements

executable statements	non-executable statements
1. assignment operators (int i=0;)	1. Declaration (int i;)
2. looping statements	2. Input " "
3. branching " "	3. output " "
4. conditional " "	4. Comment lines.

Frequency count:-

1. Write an algorithm to find the smallest number in the given set of integers. and also determine its frequency count. (unsorted).

1. Declare the variables a[], i, n, small as integer
2. Get the no. of elements in 'n'.
3. for i=0 to n
4. get the array values in a[];
5. small = a[0]
6. for i=1 to n
7. if small > a[i];
8. small = a[i]
9. print small.

Frequency count:

1.	for i=0 to n	; (n)+1;	n+1
2.	small = a[0]	; 1	1
3.	for i=1 to n	; (n-1)+1	n
4.	small > a[i]	; (n-1).	n
5.	small = a[i]	; (n-1)	n
		4n	

10

To with '4n' we can't say whether it is max time complexity, min t.c so we use notations

Asymptotic notations:

Big-Oh 'O':

Any function f(n), is defined as $f(n) = O(g(n))$ where Big 'O' denotes the maximum bound on f(n) or the max. time taken by the function, f(n).

'Θ' (Theta):

Omega (Ω):

Any function, f(n), is defined as $f(n) = \Omega(g(n))$ where $\Omega(g(n))$ denotes the lower bound on f(n).

Theta (Θ):-

$f(n) = \Theta(g(n))$ denotes both lower & upper bounds on f(n).

write an algorithm to find the given element in the array (or) not.

Declare a[], i, n, elem, present as integers

get the no. of elements in 'n'.

for i=0 to n

get the array values

get the searching element in elem

for i=0 to n

if elem == a[i]:

present = 1;

if (present == 1)

search successful

else search not successful

Frequency count:

Big oh analysis:-

3. for i=0 to n-1 — n+1
4. for i=0 to n-1 — n+1
7. if elem == a[i] — n
8. present = 1 — 1
9. if(present == 1) — 1

$O(3n+2)$

Omega Analysis:-

Best case

3. for i=0 to n-1 — n+1
4. for i=0 to n-1 — 1 (or) n-(n-1)+1
7. if(elem == a[i]) — 1
8. present = 1 — 1
9. if (present == 1) — 1

$\Omega(n+5)$

Frequency Count:

Determine the frequency count of the following program

i)
```
for(i=0 ; i<n ; i++)
    sum = sum+10;
```

ii)
```
for(i=0 ; i<n ; i++)
    for(j=0; j<m; j++)
    sum = sum+10;
```

iii)
```
for(i=0 ; i<n ; i++)
    for(j=i ; j<m ; j++)
    sum = sum+10;
```

iv)
```
for(i=0; i<n ; i++)
    for(j=0; j<m; j++)
    for(k=0; k<m; k++)
    sum = sum+10;
```

---

i)

i=0	1
i<n	n+1
i++	n
sum = sum+10	n
	3n+3

→ omit the constant

$O(3n)$

i=0	1
i<n	n+1
i++	n
j=0	n
j<m	n * m+n
j++	n*m
sum=sum+10	n*m

frequency count $O(3nm)$

$O(3nm+4n+2)$

iii)

i=0	1
i<n	n+1
i++	n
j=i	n
j<m	n*m+n
j++	n*m
sum=sum+10	n*m

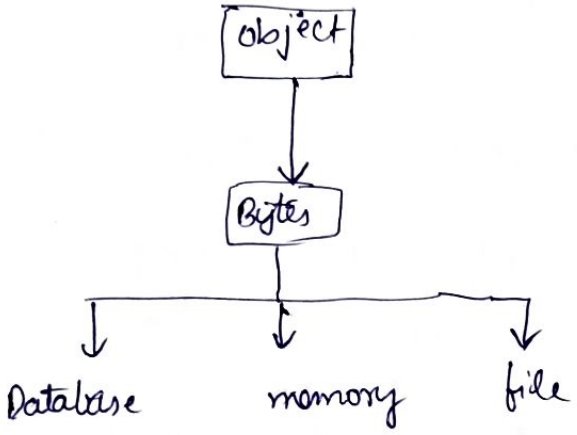
wrong (constraint)

Serialisation

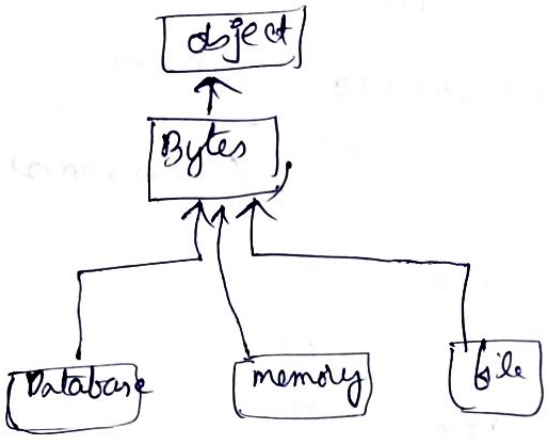
Serialisation:
process of converting object into a stream

Deserialisation:
process of reconstructing an object that has been serialised before

Serialisation



De serialisation



why Serialise ?

1. Store data / state of an object
   So, After appln terminates we loose objects
   so, to store objects we can use serialisation

objects are created in ↓ heap → temporary storage
↓ when appln terminates objects are destroyed

2. Transmit data / state of an object
3. clone an object without overriding clone

FileOutputStream
⇓
Sequence of bytes → file

vs

ObjectOutputStream
object → sequence of bytes

new ObjectOutputStream(new FileOutputStream(".txt"));

object → seq of bytes → file

FileInputStream
↓
sequence of bytes ⇐ file

vs ObjectInputStream

Object ⇐ seq of bytes

new ObjectInputStream(new FileInputStream("...txt"));

object ← seq of bytes ← file

readObject()

Interface Serializable

- required. if a class wants to have its state

serialised | deserialised

18/9/14

PHP - database connectivity

→

connection

selection is mandatory

AJAX

→ Asynchronous java script and XML

→ combination of technologies

Infix to postfix:

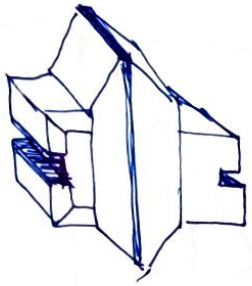
```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1;
#define FALSE 0;
int isoperand(char);
int precedence(char , char);
char pop();
 struct stack
{
 char data[100];
 int top;
 } s;
main()
{
char in[50]; post[50], aa, set, x;
 int j=0,k=0, i;
 s.top = -1;
 printf("enter infix expression");
 scanf("%s", in);
 for(i=0; in[i]!='\0'; i++)
 {
 if (isoperand (in[i])
 {
 post[j] = in[i];
 j++;
 }
```

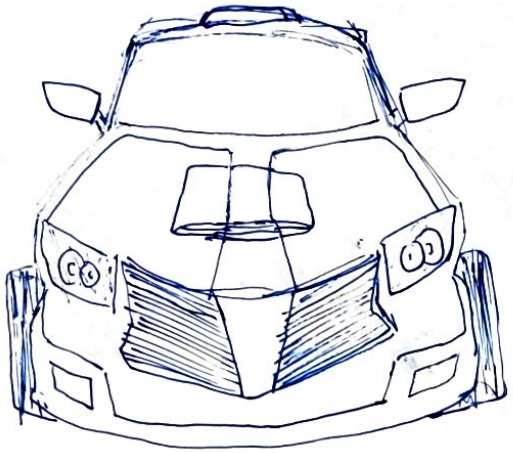
```
else
{
if (in[i]=='(')
{
s.data[++s.top] = in[i];
}
else if (in[i]==')')
{
while (s.data[s.top] != '(')
{
aa = pop();
post[j] = aa;
j++;
}
pop();
}
else
{
set = TRUE;
while(set==TRUE){
if (s.top == -1)
{
s.data[++s.top] = in[i];
set = FALSE; }
else
{
aa = s.data[s.top];
x = precedence(aa, in[i]);
if (x==1)
{
s.data[++s.top] = in[i];
set = FALSE;
}
else
{
post[j] = aa;
j++;
s.top--;
set = TRUE;
} } } } }
printf("postfix is");
for (i=s.top; i>=0; i--)
post[j++] = s.data[i];
for (i=0; i<j; j++)
printf("%c", post[i]);
}
int operand (char a)
{
a = tolower(a);
if (a>96 && a<123)
return 1;
else
return 0;
}
```

```
int precedence(char s, char i)
{
  int a, b;
  if (s=='+'||s=='-')
  a=1;
  else if (s=='*'||s=='/')
  a=2;
  else if (s=='(')
  a=0;
  if (i=='+'||i=='-')
  b=1
  else if (i=='*'||i=='/')
  b=2
  else if (i=='(')
     b=3;
    if (a<b)
     return 1;
     else
     return 2;
}
char pop()
{
    char p;
    p=s.data[s.top];
    s.top--;
    return p;
}
```

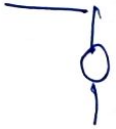
40 - (
41 -)
42 - *
43 - +
44 - ,
45 - -
46 - .
47 - /
48 - 0
49 - 1
:
57 - 9
65 - A
:
90 - Z
:
97 - a
98 - b
:
122 - z

ROUGH





Step 1. Think of a word
2. Write down dashes corresponding to each letter
3. Other person guesses



_ _ E A _

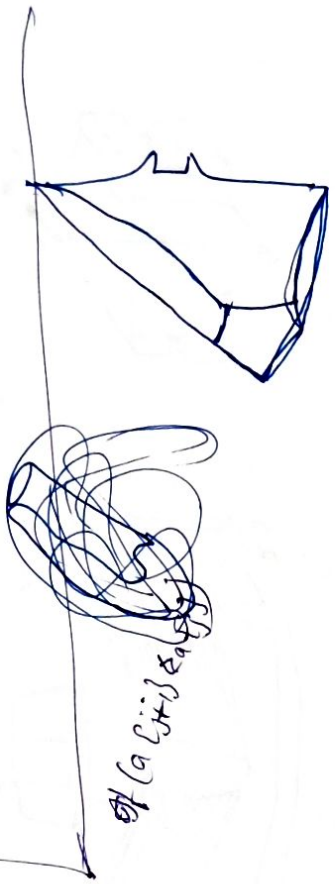
SPEAK OUT !

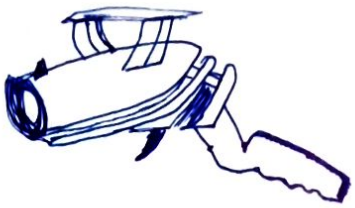
ABCDEFGHIJKLMNOPQRSTUVWXYZ

HANGMAN

Bubble Sort:

```
for(i=0; i<n; i++)
for(j=0; j<n-i; j++)
if (a[j] < a[j-1])
{ temp = a[j];
  a[j] = a[j-1];
  a[j-1] = temp;
}
```





A B C D E F G H I J K L M N
O P Q R S T U V W X Y

$2^2 = 4$

$4^2 = 16$

$16^2 = 256$

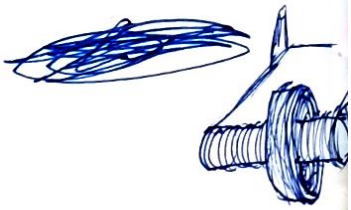
$256^2 = 65536$

$65536^2 =$

quick sort()

3, 4 }

```
   256
   256
  ----
  1536
 12800
 51200
 ------
 65536
 65536
 ------
393216
2006080
```



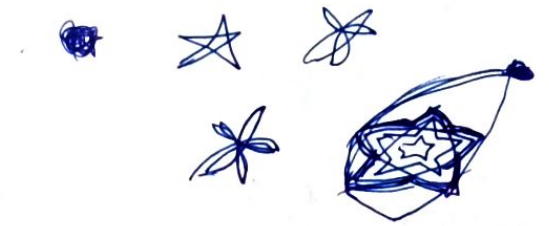
```
65536
    3
-------
196608
```

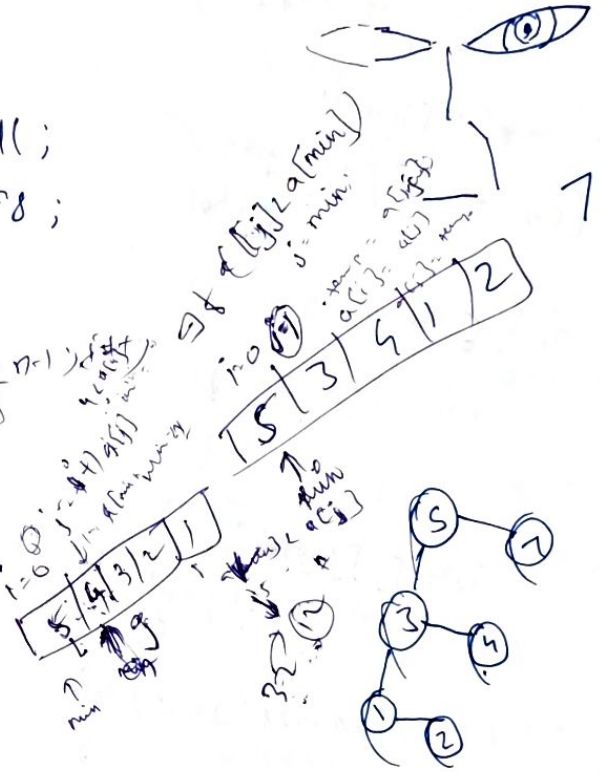
```
struct node
{
   int data;
   struct * child;
   struct * next sibling;
};
```

```
#Include<iostream>
using namespace std;
main()
{
   int a
```

```
int new_node
{
if (root == NULL)
{
new node;
child = null;
next = sibling;
curr = node;
root = node;
}
else
{
new node;
node.child = null;
node_sibling = null;
cur.child = node
cur = node
}
```

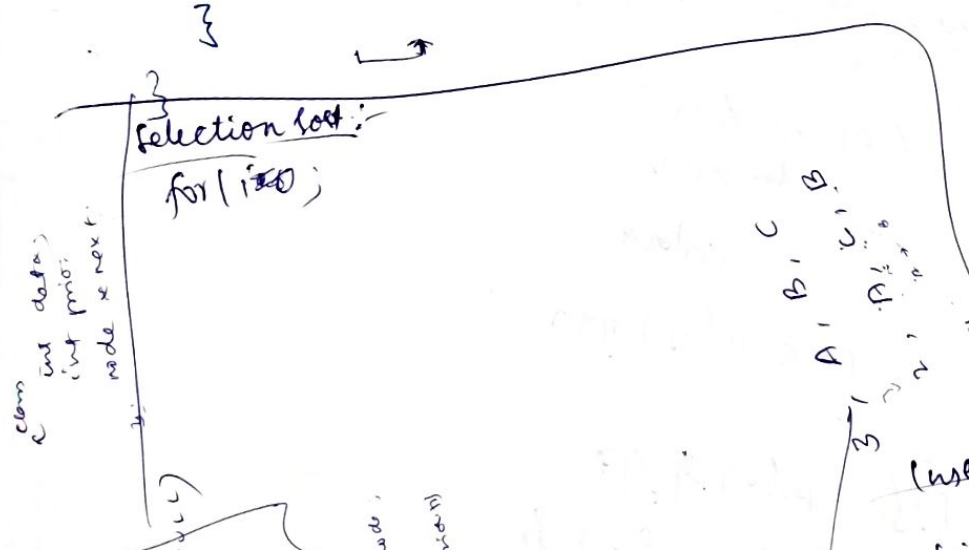
Quick sort



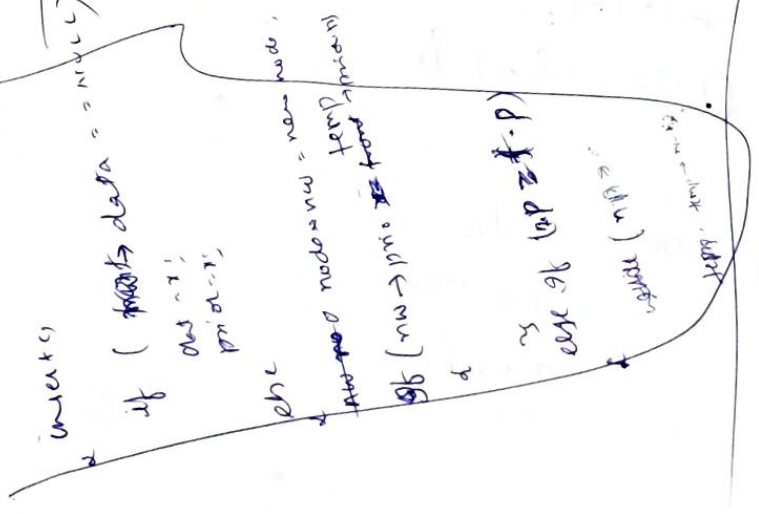


Selection sort:

```
for(i=0;
```



Syn. pr. Calc.



```
if (data > root)
check left(RST)
else if (data < root)
check left(LST)
```

Insertion sort:

```
for(i=1; i<n; i++)
{ k=a[i];
for(j=i-1; a[j]>k && j>=0; j--)
a[j+1] = a[j];
a[j+1] = k;
```

UNIT-4 Searching & Sorting

Linear search

```
for (i=0; i<n; i++)
if (a[i]==key)
{
  flag=1;
  break;
}
```

Binary search

```
low=0; high=n-1;
while (high>=low)
{
  mid=(low+high)/2;
  if (a[mid]==key)
  { flag=1; break;
  }
  else if (a[mid]>key)
  {  high=mid-1;
  }
  else if (a[mid]<key)
  {
     low=mid+1;
  }
}
```

* Input should be sorted array for binary search
* "Divide & conquer" strategy

Sorting.

Bubble Sort:-

```
for(i=0; i<n-1; i++)
for(j=0; j<n-1-i; j++)
if (a[j]>a[j+1])
{  temp=a[j];
   a[j]=a[j+1];
   a[j+1]=temp;
}
```

Selection sort:-

```
for(i=0; i<n-1; i++)
{
  min = i;
for(j=i+1; j<n; j++)
if(a[j] < a[min])
  imin = j;
temp = a[min];
a[min] = a[j];
a[i] = temp;
}
```

Insertion Sort:-

```
for(i=0; i<n; i++)
{
  k = a[i];
  for(j=i-1; (j>=0) && (a[j]>k)); j--)
    a[j+1] = a[j];
    a[j+1] = k;
}
```

Quick Sort:-

```
int split (int a[], int lower, int upper)
{
  int temp, new_lower = lower+1, pivot = a[lower], new_upper
                                                  upper
  while (new_upper >= new_lower)
  {
    while (pivot > a[new_lower] && new_lower <= new_upper
    new_lower++;
    while (pivot < a[new_upper])
      new_upper--;
```

```
if ( new_upper > new_lower)
{
        temp = a[new_lower];
    a[new_lower] = a[new_upper];
      a[new_upper] = temp;
      new_upper--;
       new_lower++;
  }  else
    new_lower++;
}
 a[lower] = a[new_upper];
  a[new_upper] = pivot;
  return new_upper;
}
void quick_sort (int a[], int lower, int upper)
{
        int i;
    if (upper <= lower)
    return;
     i = split (a, lower, upper);
     quick_sort (a, lower, i-1);
     quick_sort (a, i+1, upper);
}
```