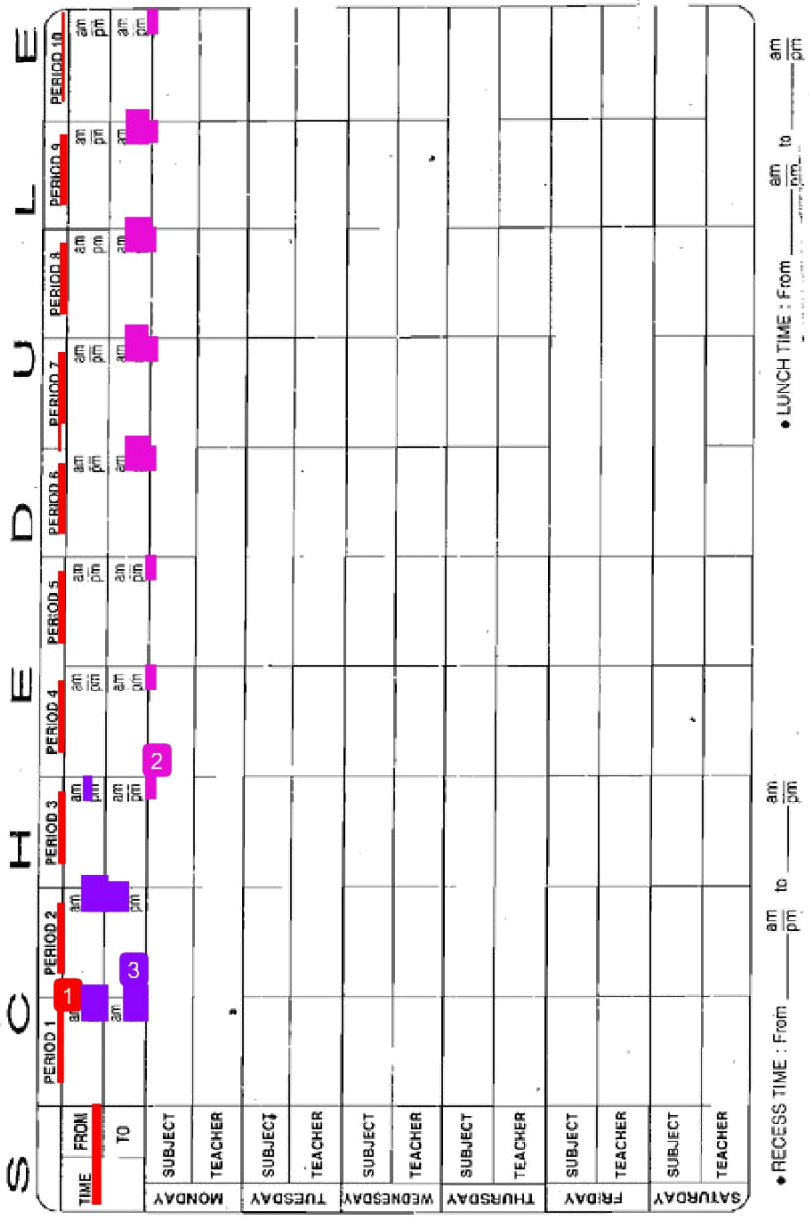
DS_2000

by Dr. D. B. Hanchate

FILE DS_2000.PDF (2.57M)
TIME SUBMITTED 29-JAN-2017 06:54PM WORD COUNT 545
SUBMISSION ID 763814637 CHARACTER COUNT 3119



This book is made from **TNPL'S** environment friendly bagasse based paper.

Hanchate

UNIT I :→

Introduction to Data Structures.

Algorithm :- An algorithm is a finite set of instruction to accomplish a particular task. Every algorithm must satisfy the foll. criteria.

i) Input :- There are zero or more quantities which are externally supplied.

ii) output :- At least one quantity is produced.

iii) Definiteness :- Each instruction must be clear and unambigious.

iv) Finiteness :- If we trace out the instruction, then for all cases the algorithm will teminate after finite no. of steps.

v) Effectiveness :- Every instruction must be sufficiently basic that it can be carried out by a person easily.

ex :- a) Write algorithm to search an element from the array of 'n' elements.

Start

```
SEARCH ( A, n, x )
/* 'A' is array of size 0 to n-1 */
/*  x is the element to be
    Searched      */
declare  i, j : integers
  i = 0
  j = n
while (i<j)
{
    if  (A[i] = = x)
    {
      print " Found"
      Return
    }
   i = i+1
}
print " Not found"
end   /* End of Search */
```

* Data :- In general data is input quantity or value and is processed by algorithm

* Data Types :- It refers to the type of quantities that variables may hold in a programming language. Every programming language has a set of inbuilt data types (int, char, etc) and instructions which manipulates the variables of this data types, but some other data types cannot be handled / manipulate by instruction (eg :- structure, record) known as user defined data types.

* Data object :- Data object is a term refering to a set of elements denoted by 'D'

Ex D = { JAN, FEB, MARCH - - - }.

D = { 0, 1, 2, - - - }.

Now, we want to describe the set of operation which may legally applied to

This implies that we must specify the set of operation and show how they work.

Ex:- Structure natno

declare ISZERO(x) → ZERO boolean — 1
SUCC (natno) → natno — 2
ADD (natno, natno) → natno — 3
ZERO ( ) → TRUE — 4
for all x, y ∈ natno Let — 5
ISZERO(ZERO) : : = TRUE — 6
ADD (x, ZERO) : : = x — 7
ADD (SUCC(x), y) : : = SUCC (ADD(x,y)) — 8

end for — 9

end natno — 10.

⇒ 0 → ZERO

1 → SUCC (ZERO)

2 → SUCC (SUCC (ZERO))

* Data Structure :- A data structure is a set of domains 'D', a designated domain 'd', a set of functions 'F' and a set of axioms 'A' and d ∈ D. The triple (D, F, A) denote a data structure 'd'.

Here; Data Structure (d):

D = { boolean, natno }

d = { natno }

F = { ISZERO, SUCC, ADD, ZERO }

AXIOMS = { statements from 6 to 8 } (in previous ex.)

ex:- Structure boolean

declare

ISTRUE (boolean) → boolean

AND (boolean, boolean) → boolean

for all a, b ∈ boolean, let.

ISTRUE (TRUE) ::= TRUE

ISTRUE (FALSE) ::= FALSE

ISTRUE (a) ::= if a is TRUE then TRUE else FALSE.

AND (a, b) ::= if ISTRUE (a) and ISTRUE (b) is TRUE then TRUE, else FALSE.

* Triple (DFA) is called abstract data type → The set of axioms describe the meaning & show that how function works, but they cannot be in the form of representation so that one can implement it directly (i.e in executable form). At this stage data structure should be designed so that we know what it does, but not necessarily how it will do it.

* How to analysis the algorithm :-

(i) Frequency count :- It is the total no of times the statement executed.

(ii) Time complexity of algorithm :- It is the time required to execute a algorithm.

ex- Addition of 'n' numbers.

```
Add ()                      — 1
{                             freq^n Count
   read n                   — 1
   x=0
   for (i=1 to n)           — n+1
   {
      x = x+i               — n
   }
   Print x                  — 1
}
```

Total freq^n count = 3+2n
≅ n.

Time complexity = O(n)
[OR = O(n)]

ex:- A program to calculate & print max value &

```
LARG (A, n)                 — 1
{   /* A (1:n) */ (Array is from 1 to n)
   max = A(1)               — 1
   for (i=2 to n)           — n
   {
      if (max < A(i))       } x(n-1)
      {
         max = A(i)         }
      }
   }
   print 'max =', max       — 1
}
```

Total freq^n count = 3+n+2(n-1)
≅ 2(n-1)
≈ n-1
∴ Time complexity = O(n-1)

ex- write a algorithm for matrix addition & calculate its time complexity.

```
MAT_ADD (A, B, C, m, n)
{  /* A (0:m-1, 0:n-1) */
   // B (0:m-1, 0:n-1)
   // C (0:m-1, 0:n-1)
   for (i=0 to m-1)         — (m+1)
      for (j=0 to n-1)      — m(n+1)
      {                                  } mxn
         c(i,j) = A(i,j) + B(i,j)        }
      }
}
```

Time complexity = mxn.

ex:- To sort n element using Bubble sort & calculate time complexity.

```
Bubble (A, n)
{  // A (0:n-1)
   for (i=0 to n-1)         n+1
      for (j=0 to n-i-1)    — n(n-1)
      {
         if (  -  )
         {
            exchange
         }
      }
}
```

Time complexity = $(n-1)+(n-2)+(n-3)$
$+ \cdots + 3+2+1$
$= \left[n + (n-1)+(n-2)+(n-3)+\cdots+3+2+1\right] - n$

$= \frac{n(n+1)}{2} - n$

$= \frac{n(n-1)}{2}$

ex - Calculate time complexity of algorithm of sequential search.

→
```
Search (A, n, x)
{ // A(0 : n-1)
  for (i=0 to n-1)
  {
    if (x == A(i))        } min, 1 time
    {                     } max. n times.
      print 'Found'
      return
    }
  }
  print 'not Found'
}
```

Time complexity = $\frac{1+2+\cdots+n}{n}$

$= \frac{n(n+1)}{2n}$

$= \frac{n+1}{2}$

* Ordered Lists →

It is one of the simplest & commonly used data types. In this items are arranged by some order.

ex:- Days of week = (MON, TUE, WED - - SUN).

operation of given list:-
i) Reading / List
ii) Searching
iii) Insertn — (do not overwrites)
iv) Addition — (overwrites)
v) Deletion.

* Array :- The most common way to represent an ordered list is by an array.

ex:- Array A = $(a_1, \cdots a_n)$

where a_i is item / value & i is the index.

Array item a_i & a_{i+1} store into consecutive memory location i & i+1 respectively. This is known as sequential mapping technique. Array elements can be accessed randomly

* Data Structure of the array :-

structure ARRAY (Value, index)

declare CREATE() → Array

RETRIVE (Array, index) → value

STORE (Array, Index, value) → Array

for all A ∈ Array, i,j ∈ index, x ∈ value Let

CREATE() → declare A(1:n)

RETRIVE (~~create~~ CREATE, i) → error

RETRIVE (A, i) → if 1 ≤ i ≤ n then A(i) else out of size

RETRIVE (STORE (A, i, x), j) → if i = = j then x else RETRIVE (A, j).

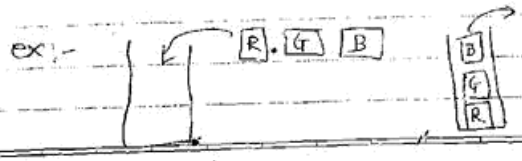
end

end ARRAY

* STACKS :-

A stack is an ordered list in which all insertion & deletion are made at one end called top.

In the stack the last element to be inserted in the stack will be the 1st to be removed. Hence stack is one as last in, first out (LIFO) list.

ex:- 

* Data Structure of Stack :-

Structure STACK (item)

declare CREATE() → Stack

INSERTION (Stack) →

ADD (Stack, item) → Stack

DELETE (Stack) → Stack

TOP (Stack) → Item

ISEMTS (Stack) → boolean.

for all S ∈ Stack, x ∈ item let

CREATE() → declare S(1:n), top ← 0

ADD (S, x) → if top >= n then print 'STACK = FULL' else { top = top + 1 S(top) ← x }

DELETE (CREATE) → error

ISEMTS (CREATE) → TRUE.

ISEMTS (ADD(S, x)) → FALSE.

Top (CREATE) → error

TOP (ADD(S, x)) → if top >= n then print 'STACK_FUL' else x

TOP (S) → if ISEMTS then print 'STACK_EMPTY' else S(top).

end

end Stack

PAGE NO. :
DATE : 18/01/2017

* Algorithm for ADD and DELETE :-
OR (PUSH) OR (POP)

```
Add (Stack, top, n, item)
{  // stack (1:n)
 if (top = n)
 {
   print 'STACK_FULL'
   return
 }
 top = top+1
 stack(top) = item
}

Delete (Stack, n, top)
{
   if (top <= 0)
   {
     print 'STACK_EMPTY'
     return
   }
   top = top-1
}
```

* Implement the foll. funcn calls :-

(1)
```
main()
{
  =
  fun1();
}

fun1()
{
  =
  fun3();
}

fun3()
{
  =
  x = fun4(5);
}
```

Note down the status of Stack at the time of each funcn execution. [MENU – Call Stack]

(2) i)
```
main()
{
  int i = 0;
  printf("%d %d %d", &i++, ++i, i);
}
```

ii)
```
main()
{
  int i = 0
  printf("%d %d %d", &i++, i, i);
}
```

iii)
```
main()
{
  i = 0
  printf("%d, %d %d", i, i++, ++i);
}
```

What will be o/p after execution of printf. Justify your answer.

* Data object Queue :-

A Queue is a type of ordered list in which all insertions take place at one end called rear and all the deletn take place at

other end called front. The first element which is inserted into the queue will be the first one to be removed. For this reasons queue are also known as first in first out (FIFO) list.

To implement a queue variables front and rear are required. Front is always less than actual front of queue and rear always points to the last element in the queue.

* Data Structure of queue :-

Structure QUEUE (item)
declare CREATE ( ) → queue
ADD (item, queue) → queue
DELETE (queue) → queue
FRONT (queue) → item
ISEMPTY (queue) → Boolean

for all Q ∈ queue, i ∈ item let
CREATE ( ) :: = declare Q(1:n), front = rear = 0
ISEMPT (CREATE) :: = TRUE
ISEMT (ADD (i, Q)) :: = FALSE
DELETE (CREATE) :: = error
ADD (i, Q) :: = if (rear = n)
{
print 'QUEUE_FULL'
return
}
else
{
rear = rear + 1
Q (rear) = i
}
DELETE (Q) :: = if (front = rear)
{
printf 'QUEUE_EMPTY'
return
}
else
front = front + 1
FRONT (CREATE) :: = error
FRONT (ADD (i, Q)) :: = if ISEMT (Q) then i
else FRONT (Q).
end
end QUEUE

* Algorithm for ADD & DELETE in queue :-

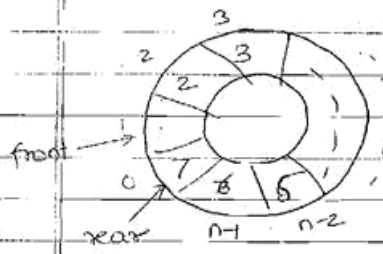
ADD (item, Q, n, rear)
{ // Q (1:n)
if (rear = n)
{
print 'QUEUE_FULL'
return
}

```
        rear = rear+1
        Q(rear) = item
    }
                rear
    DELETE (Q, MX, front)
    {
        if (front = rear)
        {
            print 'Q_EMPTY'
            return
        }
        front = front + 1
    }

#include<stdio.h>
#define N 20
top = -1;
Global

/* Stack operations */
void Delete (int stack[20], int n);
{ int tmp;                          if (top<=-1)
  tmp = stack[top]                  {
  top = top - 1;                      printf("Stack_Empty");
  return tmp;                         return;
}                                   }

void Add(int stack[20], int item, int n)
{
  if (top == n-1)
  { printf("STACK_FULL");
    return;
  }

    top = top + 1
    stack[top] = item;
}
void print(int stack[20])
{
  if (top <= -1)
  {printf("\n STACK_EMPTY);
   return;
  }
  printf("%d", stack[top]);
}
main()
{ int top=-1;
  char choice; int x, stack[20];
  while(1)
  {
    printf("Add");
    printf("Delete");
    printf("Print");
    printf("Exit");
    Scanf("%c", &choice);
    switch(choice)
    {
      case 'A':
      {
        int tmp;
        Scanf("%d", &tmp);
        Add(stack, tmp, 20);
      }
      break;
```

To declare variable tmp.

```
case 'D': Delete (stack, 20);
          break;
case 'P': Print (stack);
          break;
case 'E': exit (0);
    }
  }
}
```

* Circular queue :- The effcient way is to represent linear queue with circular queue.



← circular-queue.

"Always keep one cell empty to avoid 'queue empty condition' i.e rear = front".

queue full condition — ~~front = rear+1 = front~~ ~~rear = front+1~~.

rear + 1 = front

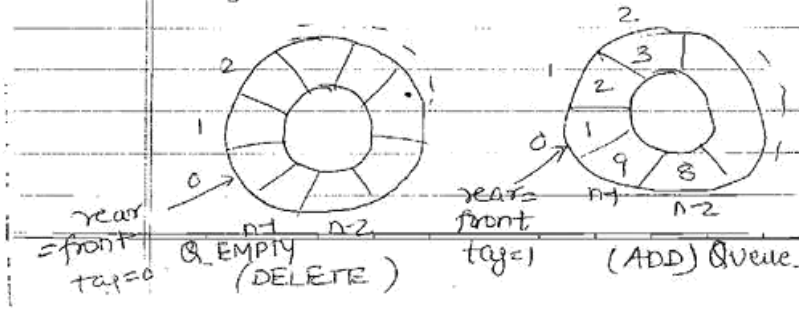
* Algorithm for ADD/DELETE in circular queue :→

```
ADD (Q, front, rear, n, item)
{
  // Q (0 : n-1)
  rear = (rear+1) mod n      (To restart queue, i.e after n=rear1 0 cell will be active)
  if (front = rear)
  {
    print ('Q_FULL'
    rear = (rear-1) mod n
    return
  }
  Q (rear) = item.
}
```

```
DELETE (Q, front, rear, n)
{
  // Q (0 : n-1).
  if (front = rear)
  {
    print 'Q_EMPTY'
    return'
  }
  front = (front+1) mod n
}
```

* Algo Algorithm for ADD-DEL, in circular queue with tag →

```
ADD ( Q, front, rear, n, item, tag)
{  \\ tag is boolean variable which store either 0 or 1
   if ((rear = front) and tag=1)
   {
      print 'Q_FULL'
      return
   }
   tag = 1
   rear = (rear+1) mod n
   Q (rear) = item.
}

DELETE (Q, front, rear, n, tag)
{
   if ((rear=front) and tag=0)
   {
      print 'Q- EMPTY'
      return
   }
   tag = 0
   front = (front+1) mod n
}
```



Assign 3 →

Implement a circular queue using array. Implement all operation such as Addition, Delem and print rear most element of queue. (without stack).

Application of stack →

Types of expression:- { Polish expr — infix, postfix, prefix }

i) Infix expr →

ex:- [(a+b/c) / (bxc)]
(unary, operand)

In infix expr all operator (excluding unary operator) comes in betn operand.

ii) Postfix expr :- In this expr operators comes after operand on which it operates.

	Infix		Postfix
ex :-	a+b	⇒	ab+
	((a*b)+c)	⇒	ab*c+

iii) Prefix expr :- In this expr the operator comes before the operand on which it operate.

ex :- a+b ⇒ +ab

((a*b)+c) ⇒ +(*ab)c

* Conversion of infix expr to postfix expr :-

i) (a+b) ⇒ ab+

ii) ((a+b)*c) ⇒ ab+c*

iii) ((a+b)/(c+d)) ⇒ ab+cd+/

iv) a+b+c
= ((a+b)+c) ⇒ ab+c+

* Infix to Postfix →

Advantages of postfix expr :-

i) No paranthesis is required.

ii) Priority is also not important.

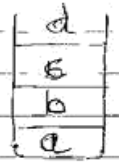
iii) Easy evaluatn of expr than infix expr.

iv) Expression can be evaluated by just scanning left to right (in single scan)

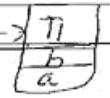
eg (a+(b/(c*d)))

⇒ abcd*/+

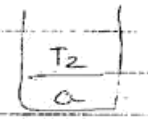
i) $T_1 = c*d$



ii) $T_2 = b/T_1$



iii) $T_3 = a+T_2$

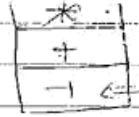


* Algorithm for conversion of infix to postfix

{ Priorities of operators →

Symbol	In stack priority (ISP)	In-coming priority (ICP)
)		
(Exponential) ** or ^ or ↑	3	4
*, /	2	2
+, -	1	1
(	0	4
#	-1	-

ex:- a+b*c



if ISP < ICP push

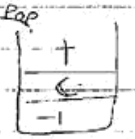
ⓐ -1 < 1 , ⓑ 1 < 2

print → abc*+

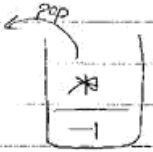
if ISP >= ICP then POP

ex:- (a+b)*c

ⓐ ab+

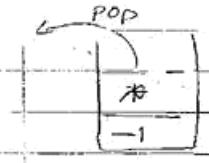


ⓑ ab+c*



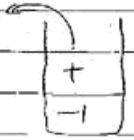
ex:- a * b + c

(a) ab*



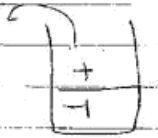
(AS ISP > ICP)

(b) ab*c+



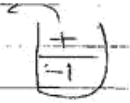
ex:- a+b+c

~~ab~~ab+



(∵ ISP = ICP)

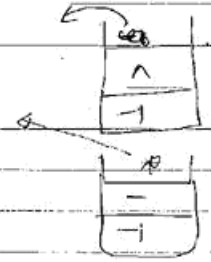
ab+c+



ex:- ~~(a+b-a~~ a∧b - a*b

a b ∧

ab∧ab*-



Asgn - 4 →

Algorithm for conversion from infix to postfix exp^n.

```
Define stacksize n:20
declare gioble top: integer
top = -1
POSTFIX ()
{
 declare stack (0:n-1) : character
 top = 0
 stack (top) = '#'
 while (1)
 {
  read (x)    // x : character
  {
  case : x = ')' : while (stack (top) != '(')
                   {
                    print stack (top)
                    top = top - 1
                   }
                   top = top - 1 // remove stack
                   top '(' symbol.
                   break;
case : x = ';' : while (stack (top) != '#')
                 {
                  print stack (top)
                  top = top - 1
                 }
                 return;
```

```
case ; x = '*' or '/' or '+' or '-' or '^';
    while ( ISP (stack (top)) >= ICP(x) )
    {
      print stack (top)
    }
    ADD (stack, x) // Add x onto stack
    break                          top

default: // x is operand
      print (x)
  }
// end of while
// end of POSTFIX.
```

```
*  ISP (y)
   // y : character
   {  x=
      case: '^' : return (3)

      case : x = '*' or '/' : return (2)
      case : x = '+' or '-' : return (1)
      case : x = '#' : return (-1)
      case : x = '(' : return (0)
```

```
*  ADD (stack, x)
   {
      if top = n-1
      {
        print STACK_FULL
        exit (0);
      }
```

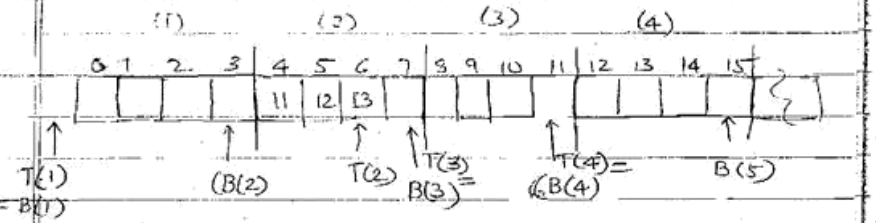
top = top + 1
stack (top) = x
}

* Assgn 5 – Postfix to Infix :–

* Multiple Stacks :–

A single dimensional array of size (1:m) can be used to represent more than one stack. Initially array can be added into required no. of stacks (say n)

ex :– The foll. fig. shows division of single array into no. of (arrays) stacks



B- (Boundary) T → (top).

B	
-1	1
3	2
7	3
11	4
15	5

T	
-1	1
6	2
7	3
11	4

1st stack – B(1) to B(2)
2nd – " – B(2) to B(3)
3rd – " – B(3) to B(4)
4th – " – B(4) upto end of array

if T(1) = B(2) — Stack 1 is full.
if T(1) = B(1) — Stack 1 is empty.

& so on ...

Initial conditions :-

a) Initially :-

B

-1
3
7
11

T

-1
3
7
11

for $0 \le i \le 3$

$B(i) = T(i)$

For each stack i, B(i) is used to hold the boundary values & shows the initial values of T(i).

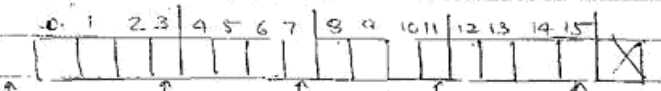
* Algorithms for ADD & DELETE :-

```
ADD (i, item)
{ //n is no. of stacks,
        0 ≤ i ≤ n-1 //

if T(i) = B(i+1)
{
    print STACK i is FULL
    return ;
}
else T(i) = T(i) + 1
STACK (T(i)) = item
}
```

```
Delete (i)
{
 if T(i) = B(i)
 {
 print stack is is EMPTY
  return
 }
 Top(i) = T(i) - 1
}
```

* Multiple Queue :→

(Short note)



initially F(1)=R(1) F(2)=R(2) F(3)=R(3) F(4)=R(4)

	(B)	(R)	(F)
1	-1	-1	-1
2	-3	3	3
3	7	7	7
4	11	11	11
5	15		

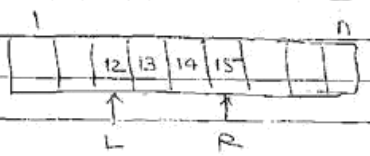
Stack EMPTY :- $B(i) = F(i) = R(i)$

FULL :- $R(i) = B(i+1)$.

* Double ended Queue :- (DEQUE)

Theory

It is a one dimensional array. We can use both the ends of array for the addition & deletion operations.

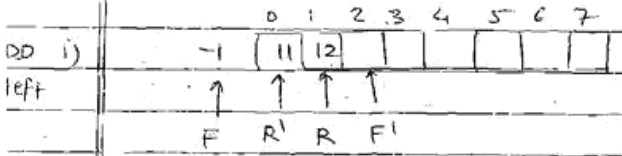


if L = R — Queue empty.

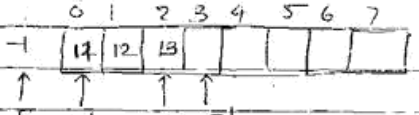
if (L = 1) & (R = n) — Queue FULL.

(*) initially —

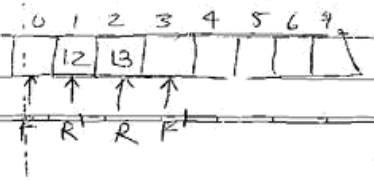
DD i)



ii)



iii)



FULL — R = n-1 (left)

R' = 0 (right).

Empty — F = R

F' = R'

* Priority queue :- (Theory).

i) Explanation.
ii) Types
iii) Operation.

1) Ascending priority queue.
2) Descending priority queue

* Assign :- A program to implement a circular queue without tag.
(ADD - DELETE - PRINT).

Device a formula in terms of front, rear, n (n - size of queue) & (1:n), to calculate total no of element in that circular queue

* LINKED LIST :-

Node.

Data	Link
field	field

- Dynamic mem. allocation.
- d.

100	10	150
150	11	200
200	12	ϕ

linked list.

a) ADDITION OF ELEMENT IN LINKED LIST-

100	10	300
150	11	200
200	12	ϕ
300	q	150

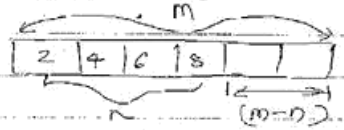
b) DELETION -

* Disadvantages of sequential representation-
  i) Order list may be vary in size.
  ii) causes the wastage of memory.
  ex:- Array of max. size 'm'.
  Order list is monitoin in this array having n elements such that

* Adv - i) No consecutive mem locum.
  ii) Insertion is possible without shifting data.
  iii) Size of array can be increased at RUN time.

* One of way to represent order list is by sequential representation. In this elements are stored into consecutive memory location.
  ex:- array, stack, queue.

no. of element < m, then (m-n) is the wastage.



ii) Once the array of max. size is declared one cannot add elements larger than the array size. This is because static memory allocation.

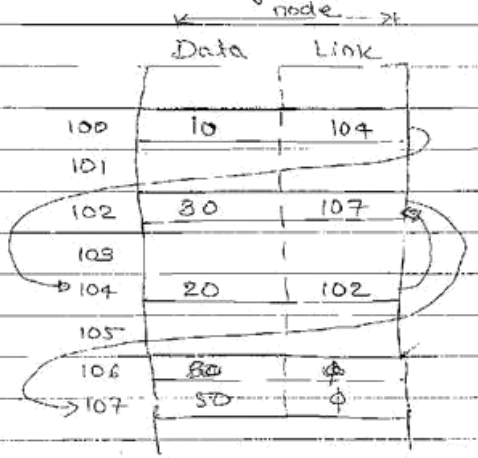
iii) Insertion & deletion operation requires large of data movement and hence more time is required.

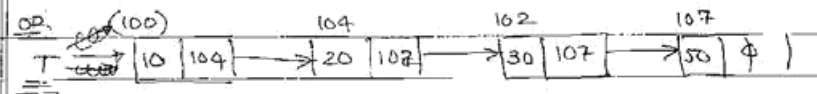
To overcome the above drawback another representation known as 'linked lists' are used.

* Linked representation :-

In linked representation element may be placed anywhere in memory. To access the element in correct order with each element we store the address of the next element. With each data element a pointer is associated to point the next element, called link. A link and data element collectively called 'node'.

Refer fig (a), (b), (c)





* Insertion :-

Consider a ordered list (10, 20, 30, 50) is maintained using linked representation. Now, insert new element 40 in between 30 and 50.

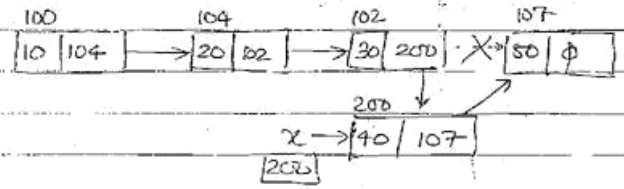
Algorithm :-

Step 1 :- Allocate a new node, say x

Step 2 :- Set the data field of x to 40.

Step 3:- Set the linked field of x to node after 30 with contain add. of 50.

Step 4:- Set the link field of a node containing 30 to x.



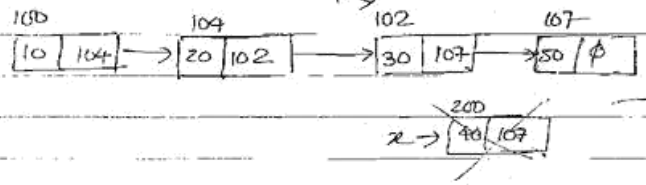
* Deletion:-

Delete 40 in above list.

Step 1:- Find the node which is before 40, say y.

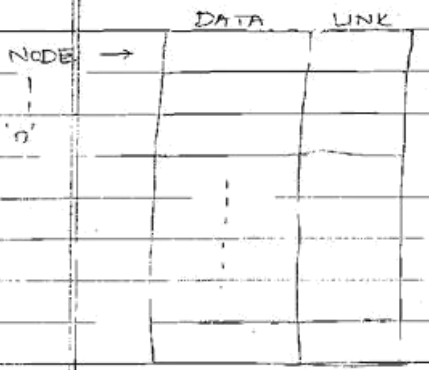
Step 2:- Set a link field of y to the node pointed by link field of x

Step 3:- Free the node x (deallocate it).



* Storage Pool :-

Linked list is a chain of nodes. To create a linked list memory is divided into nodes such that each node must have atleast one link field.



Memory Block

Initially the linked list is empty and all pre nodes are kept together to form a block. Such a block is known as 'Storage Pool', in another word the Storage pool contains all the node that are not currently being used.

The 2 operations performed on a storage pool :-

(i) GETNODE (X) :-

It removes one free node from storage pool and return its address in 'x'.

(eg. of procedure in text book Pg- 115)

same as (malloc in C)

i.e x is now a pointer to that node.
If no node is free in the pool, it return error message & stops.

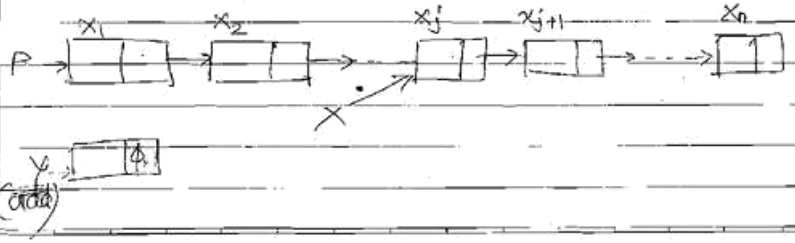
ii) RET (X) :-
(as procedure) pg-47
Ans It returns node x back to the storage pool.

Q) Consider a node has two fields, say data and link. Let X be a pointer to the first node in single linked list. Write a procedure to insert the new node at the end of linked list. If Initially if linked list is empty then X be a pointer to the newly created node.

NODE : DATA & LINK

```
Insert (X, data)
{
  // X be a pointer to 1st node
  // data is variable's value that
     you want to insert in node.
  // link field of last node contains ϕ

GETNODE (Y)      // create new node : Y
LINK (Y) := ϕ
DATA (Y) = data.
if X = ϕ
{
  X = Y
  return
}
else
{
  T = X
  while (LINK(T) ≠ ϕ)
  {
    T = LINK(T)
  }
  LINK (T) = Y
  return
}
}
```

Q) Let $P = (X_1, X_2 \dots X_j \dots X_n)$ be a linked list where X_i is a node having 2 fields DATA & LINK and $1 \le i \le n$. Let X be a pointer to X_j $(1 \le j \le n)$. Write a procedure to add new node 'Y' after X_j so that linked list will be,
$(X_1, X_2 \dots X_j, Y, X_{j+1} \dots X_n)$

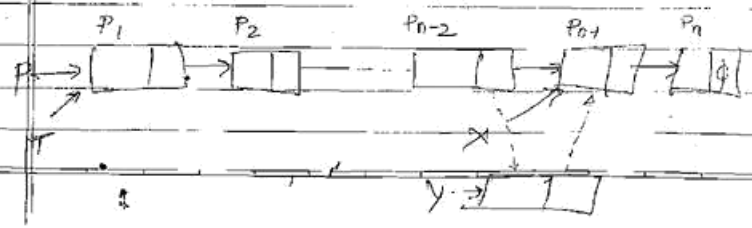


```
Insert (P,X,Y)
{
  // P is pointer to 1st node.
  // X is --"-- to xj node
  // Y is --"-- new node.

  LINK(Y) = LINK(X)
  LINK(X) = Y
}
```

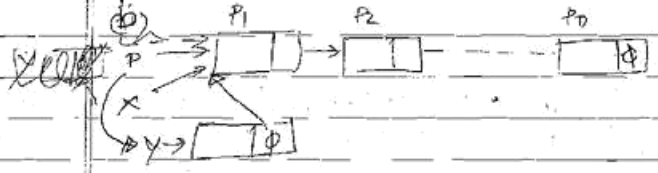
a) Let P be a pointer to the 1st node in a single linked list. Write a procedure to add a new node 'Y' before the node X where 'X' is an arbitary node in the list. If, P = ϕ then Y should be reset to point to the new first node.

Let 'P' be a pointer to first node and X is an arbitary node in the list. Write a procedure to add new node Y before X. If X = P then P should be reset to point to new first node Y.

(a)



(b)



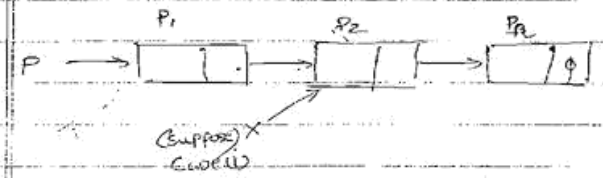
Code (b)

```
Insert (P,X,Y)
{
  if (P = X)
  {
    LINK(Y) = P
    P = Y
    return
  }
  else
  { T = P
    while (LINK(T) ≠ X)
    {
      T = LINK(T)
    }
    LINK(Y) = X
    LINK(T) = Y
    return
  }
}
```

Code (a)

To find addr of node Pn-2).

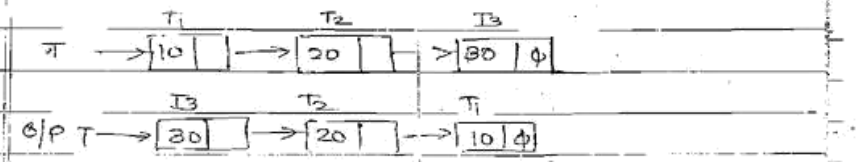
Q) Let $P = (P_1, P_2 \ldots P_n)$ be a single linked list where P_i, $1 \le i \le n$ is a node having two fields DATA and LINK. X is a pointer pointing to any one of the node of the linked list. Write the algorithm to delete node X from the linked list if X=P then P should be reset to point to the next node as a first node.




```
DELETE (X, P)
{
  if (X = P)
  {
    X = link(P)
    RET(P)
    P = X
  }
  GETNODE (t).
  t = P                    OR (LINK(T) = X).
  while (t -> data != X -> data)
  {
    t -> link = X -> link   Link[t] = Link[x]
  free RET (X)
}.
```

Q) Inverted Linked list :-

Ex1)



Ex2) Let $T = (T_1, T_2 \ldots T_n)$ be a linked list. Write algorithm to invert the linked list so that after the execut^n $T = (T_n \ldots T_2, T_1)$.

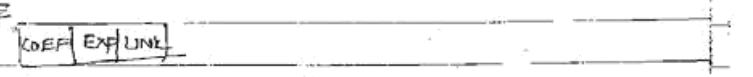
Ex2) Let P be a pointer to 1st node in single linked list and X is arbitary node in that list. Write an algorithm to add node Y before X. If X=P then P should be reset to pt. to the new first node Y and if X = NULL then add new node Y at the end of the list.

* Use of Linked list to represent polynomial :-

$3x^2 + x + 10$ — Polynomial.

Structure → NODE | COEF | EXP | LINK |

(Linked) Mem Representation.



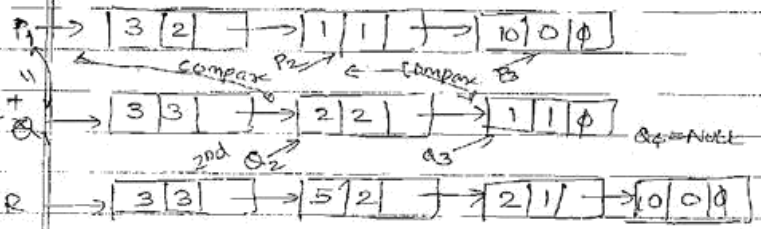
Polynomial addition using linked list :-

Eg:-

$3x^2 + x + 10$

$+ \quad 3x^3 + 2x^2 + x$

$3x^3 + 5x^2 + 2x + 10$



1st Compare Power of P and Q.
if equal add else increment Q
and again compare.
If addition take place, also increment P along with Q.

```
Struct NODE
{
  int data;
  Struct NODE * link;
} * Head, * New, * Tmp;
void Add (void)
{
  int n;
  printf ("\n Enter new value ");
  Scanf ("%d", &n);
  new = (struct Node *) malloc (sizeof (struct NODE));
  New -> link = '\0';
  New -> data = n;
  Tmp = Head;
  while (Tmp -> link ! = '\0')
  {
    Tmp = Tmp -> link
  }
  Tmp -> link = New;
}
Void Insert (void)
{
  int n, t;
  printf (" Enter new value");
  New = (struct Node *) malloc
  New -> link = '\0';
  New -> data = n;
  printf ("\n Enter the value where new
   value is to be inserted");
  Scanf ("%d", &t);
  Tmp = Head;
  if (Head -> data == t)
  {
  New -> link = Head;
  Head = New;
  } return;
```

```
Tmp = Head;
while (Tmp->link->data != t)
{
    Tmp = Tmp->link;
}
New->link = Tmp->link;
Tmp->link = New;
}

void Delete (void)
{
printf ("Enter the data to be deleted");
tmp2 = (struct Node*) malloc (sizeof(--))
scanf ("%d", &t);
if (Head->data == t)
{
    Tmp = Head;
    Head = Head->link;
    free (Tmp);
}
Tmp = Head;
while (Tmp->link->data != t)
{
    Tmp = Tmp->link;
} Tmp->link =
Tmp2 = Tmp->link;
Tmp->link = Tmp->link->link;
free (Tmp2);
}
```

```
main ()
{
char choice;                    'Head = Tmp = New = %0';
while (1)
{
printf ("Addition");
printf ("Insertion");
printf ("Deletion");
printf ("List");
printf ("Exit");
choice = getch ();
switch (choice)
{
case 'A' : if (Head = '\0')
           {
           Head = (struct Node*) malloc -
           scanf ("%d", &Head);
           Head->link = '\0';
           }
           else
           { Add ();  }
           break;
case 'I' : (A)
           else
           { Insert (); }
           break;
case 'D' :- if (Head == '\0')
           { printf ("EMPTY");
```

```
else
    { Delete (); }.
    break;
case 'P' - - -
case 'E' - - -
```

* Program for Polynomial addition :→

```
struct Poly
{
    int COEF;
    int EXP;
    struct poly  * link;
} *x, *y, *z, *tmp, *t;
main ()  int n, i, coef
    x = y = z = '\0'
    printf ("Enter first polynomial expr");
    scanf ("%d", &n)        (no of expr.
                             x^2 or x^3 ...
    for (i=n; i>=0; i--)
    {
      printf ("\n Enter the coefficient of
      position %d", i);
      scanf ("%d", &coef);
      if (coef != 0)
      {
      tmp= (struct poly *) malloc (sizeof
                  (struct Poly))
      tmp -> COEF = coef;
      tmp -> EXP = i;
```

```
tmp -> link = '\0';
if (x == '\0')
  {
    x = tmp;
  }
else
  {  while (tmp -> link != NULL)
     {
       tmp = tmp -> link;
     }
     tmp -> link = tmp;
  }
} /* End of for */

printf ("Enter 2nd polynomial expr");

                (scan 2nd polynomial)

while (x -> link != '\0' && y -> link != '\0')
{
  if (x -> EXP < y -> EXP)
  {
  tmp = (struct Poly*) malloc - - -
  tmp -> EXP = y -> EXP ;
  tmp -> COEF = y -> COEF;
  tmp -> link = ('\0');
  if (z == '\0')
```

```
{
    z = tmp;
}
else
{
    t = z;
    while (t->link != '\0')
    { t = t->link
    }
    t->link = tmp;
} y = y->link;
} /* End of if */
if (x -> EXP > y -> EXP)
{
  {store x in z using tmp &
    increment x}.
}
if (x -> EXP = y -> EXP)
{
  tmp = (struct POLY *) malloc
  tmp -> EXP = y -> EXP
  tmp -> COEF = x -> COEF + y -> COEF;
  tmp -> link = '\0'
  if(z == '\0')
  {
    z = tmp;
  }
  else
  { t = z;
```

```
while (t -> link != '\0')
{
    t = t -> link;
}
t -> link = tmp;
}
y = y -> link;
x = x -> link;
} /* End of if */
} /* End of while */

while (y -> link != '\0')
{
  tmp = (struct POLY *) mall
  tmp -> EXP = y -> EXP;
  tmp -> COEF = y -> COEF)
  tmp -> link = '\0';
  if (z == '\0')
  { z = tmp; }
  else
    { t = z; }.
  while (t -> link != '\0')
  {
    t = t -> link;
  }
  t -> link = tmp;
  y = y -> link;
} /* End of while
```

```
while (x->link != '\0')
{
    // (same as above)
}
Print -> while (z1 != 0)
{
    printf ("%d %d \n", z->COEF, z->EXP)
    z = z->link;
}
} /* END OF MAIN */
```

\# Algorithm for Polynomial addition.

```
Create Poly (x)
Create Poly (y)
z = Empty
if ( EXP(x) < EXP(y))
{
    Create New Node at end of z
    EXP(NEW) = EXP(y)
    COEF(NEW) = COEF(y)
    y = LINK(y)
}
if (EXP(x) > EXP(y))
{
    ||
}
```

Q) How are stacks implemented using linked lists?

```
if (EXP(x) == EXP(y))
{
    Create NEW at end of z
    EXP(NEW) = EXP(y)
    COEF(NEW) = COEF(x) + COEF(y)
    x = LINK(x)
    y = LINK(y)
}
while (y -> link != NULL)
{
    attach (y) to z
    y = link (y)
}
```

Similarly for x, LINK(x) = Null

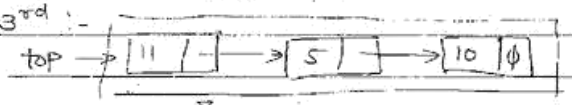
* LINKED STACK :-

(10, 5, 11) . top -> NULL

Initially :- 1st top -> [10|ϕ]

2nd :- top -> [5] -> [10|ϕ]

3rd :-



Algorithm :- (To add & delete in linked stack) st

```
Add (S, item)
{
    // S is a stack using single linked list.
    // top is a pointer pointing to top most element of stack S.
```

```
GETNODE (NEW)
LINK (NEW) = TOP
top = NEW
data (NEW) = item.
}
```

Refer algo for insertion of node at top

```
DELETE ( )
{
  if (TOP== NULL)
  {
    print 'Stack Empty'
    return
  }.
  X = top
  top = LINK (top)
  RET (X)
}
```

(X is a tmp. pointer).

let X be ptr. to node before top (node)

```
top = X
X = link (top)
free(X)
```

* Link(top) = ϕ

Linked Queue :-

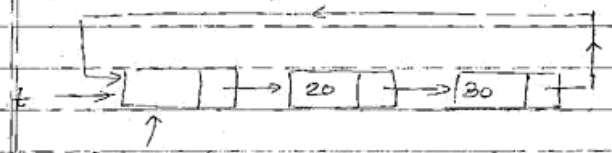
```
front = rear = NULL
Add Q ( item)
{
  GETNODE (New)  → LINK (NEW) = NULL
  DATA (New) = item
  if (rear == NULL)
  {
    front = rear = New
    return
  }
```

```
else
{ LINK (rear) = New
  rear = new
}
```

```
DELETE Q ( )
{
  if (front == rear = NULL)
  {
    print ' Q Empty'
    return
  }
  X = front
  front = link (front)
  RET (X)
}
```

(X is a temp. pointer).

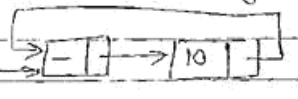
* Circular Singly Linked Queue :-



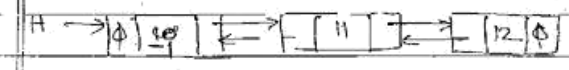
head node :- it contains no data.

a) Initial condition :-

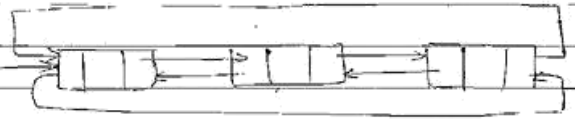
b) After adding one node.



* Doubly Linked list :-



Circularly doubly linked list ->.



Circular Doubly linked list ->

```
* struct NODE
  {
    int data;
    struct NODE *LLink;
    struct NODE *RLink;
  } *Head, *New, *Tmp;

void Insert (void)
{
   int x,h,

   printf("\n Enter value to insert");
   scanf("%d", &x);
   printf("\n After which value you
     have to insert");
   scanf("%d", &a);
```

```
/* Search value 'a' first */
Tmp = Head;
do while (Tmp != Head)
  {
    if (Tmp -> data == x)
    { New = (struct Node *) malloc(sizeof(-)
      New -> LLink = Tmp;
      New -> RLink = Tmp -> RLink
      Tmp -> RLink -> LLink = New;
      Tmp -> RLink = new;
      Tmp -> data = x;
      return;
    }
    Tmp = Tmp -> RLink;
  } while ( Tmp != Head);
  printf (" Value not found ");
} /* end of insert */

Void Delete (void)
{
  int x;
  printf (" which element to be delete ?");
  scanf ("%d", &x);
  Tmp = Head;
  do
  {
    if (Tmp -> data == x)
```

```
{
Tmp -> LLink -> RLink = Tmp -> RLink
Tmp -> RLink -> LLink = Tmp -> LLink
return;
}
Tmp = Tmp -> RLink;
} while (Tmp != Head);

void print (void)
{
}

main ()
{
char choice;
while(1)
{printf ("\n [I]nsertion");
printf ("\n [D]eletion");
printf ("\n [P]rint");
printf ("\n [E]xit");
printf ("\n Enter your choice");
choice = getch ();

Head = (struct NODE *) malloc (sizeof
(struct NODE));
Head -> LLink = Head -> RLink = Head;

switch (choice)
{:
```

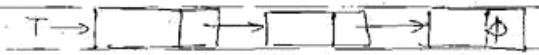
```
case 'i': Insert ();
break;
case 'd': Delete ();
break;
case 'p': Print (); break;
case 'e': exit(0);
}
}
}
/* end of main */
```

Linear linked list :- (chain)

It is a set of ordered list and connected together with the help of a linked field. Each element is stored in the node. In linear linked list the last link field of last node contain 'NULL'.

(S.N)

eg.



In a linear linked list only way to search a node which is before p is to start back from the begining of. To avoid the searching from first node again & again the linked field of last node points to the first and such a list is called as Circular singly linked list.

Again by making the list circular, it is very easy to manipulate the list. eg. To add the node, one can add it before any node in a circular linked list. But in linear linked list there are 2 cases, either before or after a node.

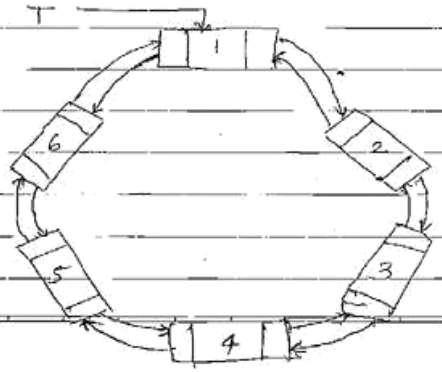
* Doubly Linked List :-

→ Doubly linked list is a collection of node. Each node contain three fields, i) DATA, ii) RLINK, iii) LLINK. DATA field is used to store the element. RLINK is used to point right node & LLINK is used to point left node.

eg:-

Linear doubly linked list.

ii) Circular doubly linked list.



a) Head node :- It is the first node in a linked list.

Zero linked list has to be created as a special case in algorithm.

eg:- Addition of new node. In this case if the linked list is zero then new node will become now 1st node in linked list. To avoid this special case the head node is used.

```
Add (T, N)
{
  // T is a pointer to a node
  // after which new node will be added.

  LLINK(N) = T
  RLINK(N) = RLINK(T)
LLINK(RLINK(T)) = N
  RLINK(T) = N
```

```
Del (T, N)
{
  // T is a pointer to first node.
  // N → node to be deleted.

RLINK(LLINK(N)) = RLINK(N)
LLINK(RLINK(N)) = LLINK(N)
free(N)
}
```

* Strings :-

$S = s_1 s_2 \cdots s_n$ set

where s_i is the character over the character set

then S is called string.

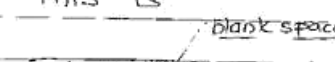
length of S = no. of character in that string

if length(S) = 0 → it is called as Null string.

* Data representations for string:-

i) Array (Sequential representation)

eg:- "This is"



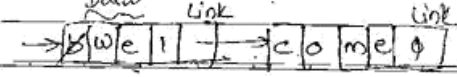
a) Sequential representation is not efficient when insertn & deletn into and from resp. from middle of string are carried out.

b) Locatn are reserved for string having fixed size, Suppose n. If the String whose size is less than n then the remaining memory is wastage.

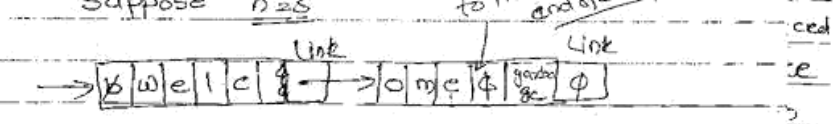
ii) Linked representation :-

→ a) with node size = n.

Suppose (n=4) & [String = "Welcome"

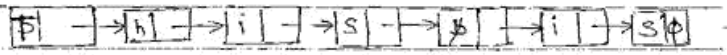


Suppose n=5



b) With node size always = 1

eg:- "This is"



Operations on string:-

→ Searching → Char.
→ substring.

strmp → String matching.

str_ → Pattern matching (whether given string is substring of given string)

strcat → Concatanation.

→ Reversing.

strlen → String length.

1) String matching:-

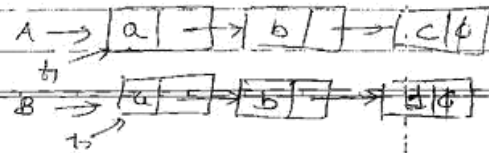
Q) Suppose there are 2 strings.

$A = \{x_1, x_2 \cdots x_m\}$

& $B = \{y_1, y_2 \cdots y_m\}$

Write algorithm which print +1, 0 or -1 if A>B, A=B, A<B respectively.

→



logic →

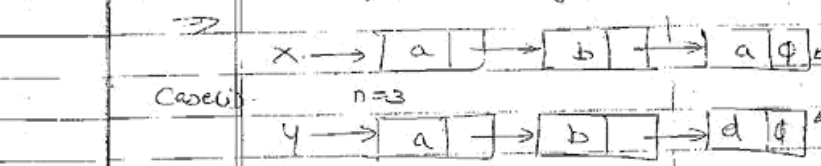
```
while (t1 -> data == t2 -> data)
{
    increment t1, t2
}
if (t1 -> data > t2 -> data)
{
    A is greater
}
else — B is greater.
```

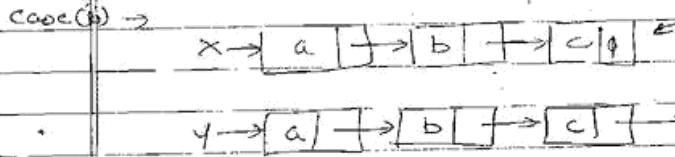
If (length of string (A) > length of string (B) & vice A is greater & vice versa.

Algorithm →

```
Strmat (A, B)
{
// A — is pointing to firststring.
// B — is pointing to 2nd string.
// t1 be a pointer pointing to A.

while ( A -> link != ϕ And B -> link != ϕ)
{
```

2) If $X = (x_1, x_2 \ldots x_m)$ & $Y = (y_1, y_2 \ldots y_n)$ are the strings where x_i & y_i are the letters of the alphabet then $X < Y$, if $x_i = y_i$ for $1 \le i \le j$ and $x_j < y_j$ or if $x_i = y_i$ for $1 \le i \le m$ and $m < n$. Write an algorithm which takes two strings x & y and returns either -1, 0 or +1 if $x < y$, $x = y$ or $x > y$ respectively.





Algorithm →

```
STRCMP(X, Y)
{
while (X != NULL and Y != NULL)
{
    if (DATA(X) < DATA(Y))
    {
        return (-1)
    }
```

```
if (DATA(X) > DATA(Y)
{
    return (+1);
}
if (DATA(X) = DATA(Y))
{
    X = LINK(X)
    Y = LINK(Y)
}
}
if (X= NULL and y= NULL)
{
    return (0)
}
if (x ! = NULL)
{   return(+1)
}
if (y ! = NULL)
{
    return(-1)
}.
```

3) Consider two strings S and PAT. Write a program to check is whether 'PAT' is substring of S.

⇒ eg:- S = "abcdef"
PAT = "de"

∴ PAT ⊂ S.

PAT = "cd"

Algorithm :-

```
STRMAT (S, PAT)
{
T = S          // T is a pointer pointing to S
X = PAT        // X is pointer pointing to PAT
While (T ! = NULL)
{
  While (PAT ! = NULL)
  {
    if (DATA(S) = DATA (PAT))
    {
      S → link (S)
      PAT → link (PAT).
    }
    else
    {
      break;
    }
  }
  if (PAT = NULL)
  {
    "Substring matched"
    return(S).
  }
  else
  {
    T = link (T)
    S = T
    PAT = X
  }
print " PAT is not substring".
```

Assigns →

1) Program to convert infix exp^r to postfix.
2) Prog. to convert postfix exp^r to infix.
3) Prog. to add element & delete element from Circular queue without flag.
4) Progr to add element in & delete element from stack using singly linked list.
5) Prog. to add element ——— from doubly linked list.
6) Prog. to add two polynomial using Singly linked list.
7) Prog. to compare 2 strings using singly linked list.
8) Binary tree traverses.
9) Depth first search.
10) Breadth first search
11) Prog. to add record & delete record from Sequential file.
12) Prog. to ——— from direct access file.

* A single link list, is used to store list & single variable polynomial.

Generalised lists :-

• A generalized list 'A' denoted by $A = (\alpha_1, \alpha_2, \ldots \alpha_n)$ where α_i are for, $1 \le i \le n$ are either atom or list. If α_i is the list known as sublist of list 'A'.

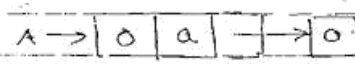
'A' is the name of the list and α_1 is the head of list A and $(\alpha_2 \ldots \alpha_n)$ is called tail.

#1) Use of generalised list to represent ordered list.

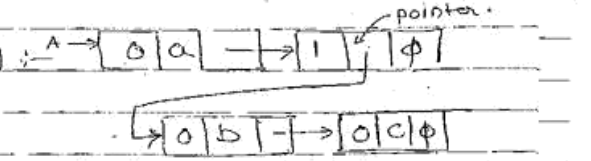
• Generalise list → 0 – atom, 1 – sublist. Tag | Data | Link

i) A = (*) : A = ϕ

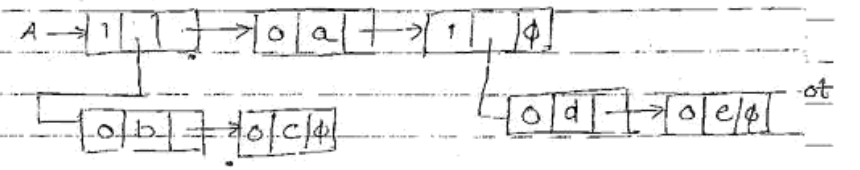
ii) A = (a,b)



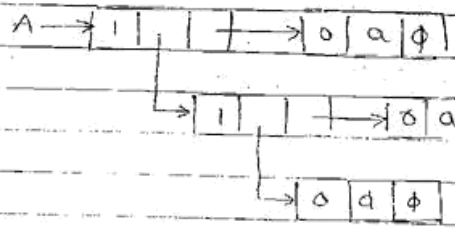
iii) A = (a, (b,c))



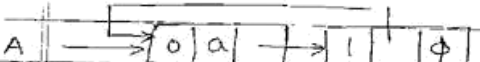
iv) A = ((b,c), a, (d,e))



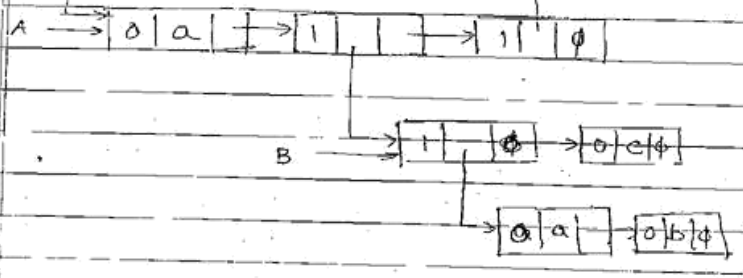
v) A = ((d), a), a)



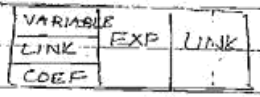
vi) A = (a, A)



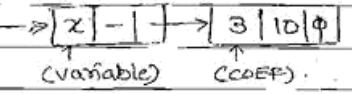
vii) A = (a, B, A)
B = ((a, b), c)



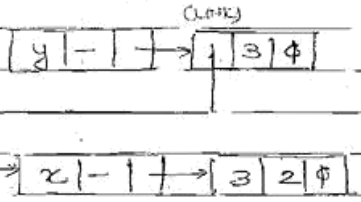
*2) Use of generalized list to store multivariable polynomial :-



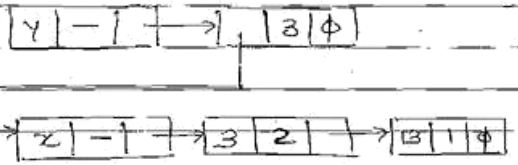
i) $3x^{10}$.



ii) $3x^2y^3 = (3x^2)(y^3)$.

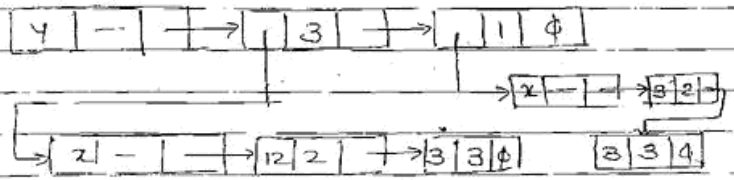


iii) $3x^2y^3 + 13xy^3 = (3x^2+13x)y^3$



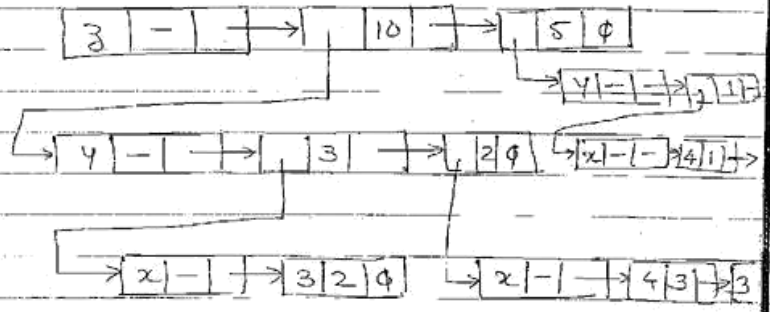
iv) $12x^2y^3 + 3x^2y + 3x^3y^3 + 3x^3y$.

$= (12x^2+3x^3)y^3 + (3x^2+3x^3)y$.



v) $3x^2y^3z^{10} + 4x^3y^2z^{10} + 3x^2y^2z^{10} + 4xyz^5 + 4x^2yz^5 + 14x^2y^2z^5$

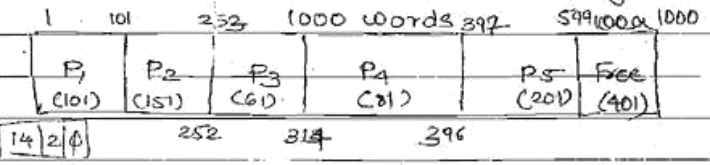
$= ((3x^2)y^3 + (4x^3 + 3x^2)y^2)z^{10} + ((4x + 4x^2)y + 14x^2y^2)z^5$



S.N * Dynamic Memory Management:-

In multiprocessing computer envoirnment several programs reside in memory at same time. Different programs have different mem. requirement. Each program may require a contiguous storage blocks in variety or sizes. When execution of program is complete it release the mem block allocated to it and this free block may now be allocated to another program. In dynamic environment mem required by a program is not known in advanced. Again the block of mem will be free in some order different from that in which they were allocated.

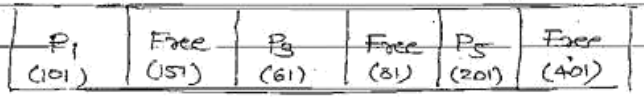
For eg:- Consider the small mem of size 1000 words and 5 programs



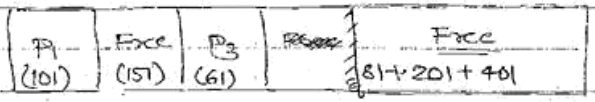
[Status of Memory]

P_1, P_2, P_3, P_4, P_5 make request of size 101, 151, 61, 81, 201 resp.

Assume that programs P_2, P_4 complete the execution & free the mem used by them.



We know have 3 free blocks but not "contiguous". Hence further large block request (eg of block size 501) cannot be satisfied. Now P_5 completes its execution, then this block should be combined with its adjacent block which is free further large request can be satisfied.

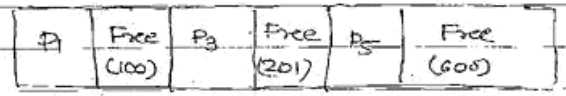


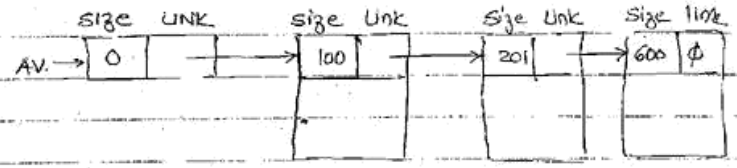
Status of memory.

* DATA STRUCTURE TO REPRESENT FREE BLOCK

In dynamic allocation environment to free block of required size it is necessary to keep the track of those blocks that are not in use - All this free blocks can be maintained in the linked list (chain form). Each node in the free list has two fields in its first word. First field is size & 2nd field is Link field.

Consider,



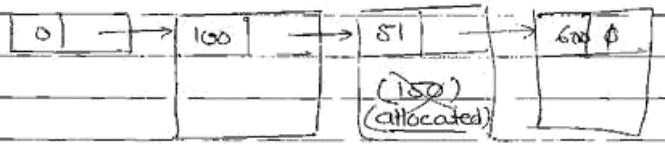


Free list / chain structure of free list

Allocation Strategies:-

i) First - Fit :- When a block of size 'n' is requested then the free list is traversed sequentially to find the first free block whose size is greater than or equal to n. once the block is found it is removed from the list and allocated (if size of block = n). otherwise the block is split in two blocks such that one block having the size equal to n & that block is allocated and remaining is still available in the free list.

eg:- n = 150

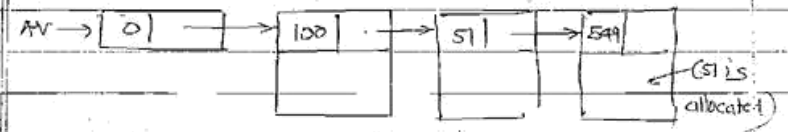


ii) Best - Fit :-

In this method when a request of size n is made the entire free list is traversed sequentially first & smallest free block whose size is greater than or equal to n is allocated.

eg:- n = 51

3) Worst - Fit :- In this method always the portion of largest free block is allocated.

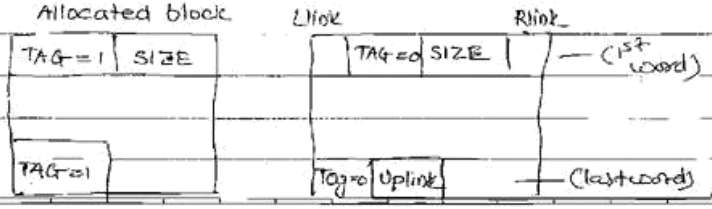


* Boundary Tag Method :-

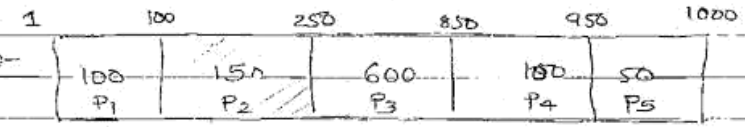
i) First operation :- when request of block size 'n' then search free block of size greater than equal to n & allocate it i.e remove it from free list.

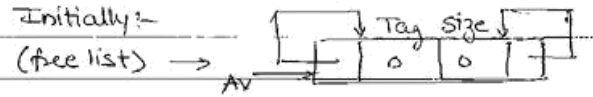
ii) 2nd operatn :- Returning free block back to free list as well as check whether its neighbour are also free. If so then they can be combine into a single block.

To implement 2nd operatn the node structure can be modified as shown below :-

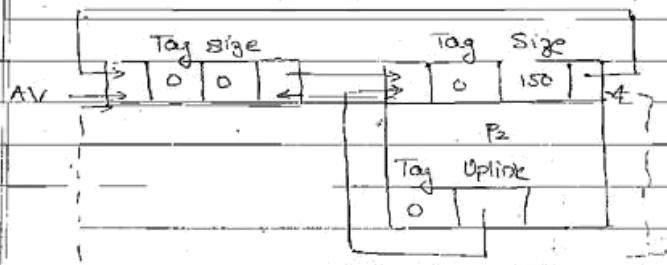


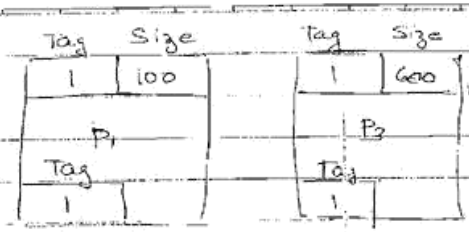
The 1st & last word of each block are reserved for allocatn information. First word of each free block has 4 fields, llink, rlink, tag and size. and the last word of free block having 2 fields namely tag & uplink. This free block structure assumes doubly linked list. Both tag fields are set to zero for free blocks. Uplink is used to point to the start of block.



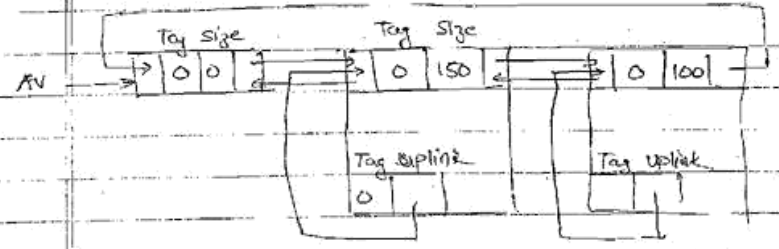


Suppose P2 is free.

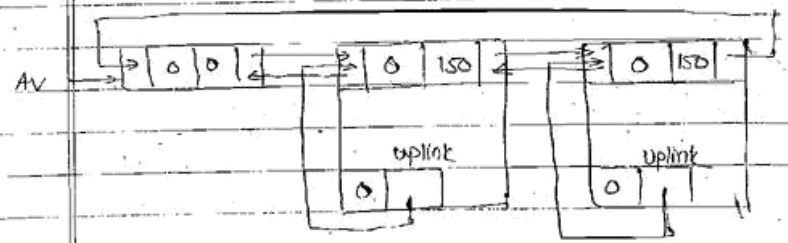




Suppose P4 is free.



Now, if P5 is also free (then P4 & P5 are merged)



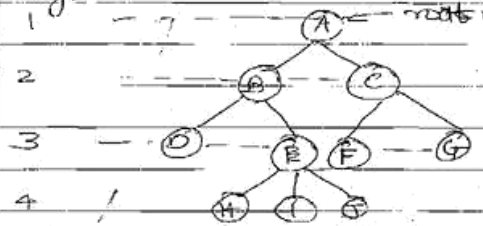
Condition of merging →
if tag field of ~~previous block~~ last word of previous block is zero the merge.

eg:- As tag field of (949) is zero merge P4 & P5

TREES :-

Def^n :- A tree is finite set of one or more nodes such that there is a specially designated node called the root node and the remaining nodes are partition into 'n' >= 0 disjoint sets $T_1, T_2, \ldots T_n$ where each of this set is a tree. $T_1, T_2 \ldots T_n$ are called the subtrees of a ~~loop~~ root.

eg:-



~~root~~ node.

disjoint sets,
T_1 { B, D, E }
T_2 { C, F, G }

i) The no. of subtree of a node is called its degree.

Node	degree
A	2
B D	0
E	3

ii) The nodes that have degree zero are called leaf nodes.

eg:- D, F, G, I, J, H.

iii) Degree of tree :- It is the max. degree of a node in the tree.

degree of tree = 3.

iv) The level of a node is defined by initially letting marking the root node at level one.

v) The ht. or depth of a tree is defined to be the maximum level of any node in the tree.

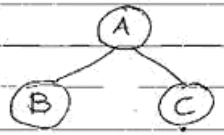
ht. of tree = 4.

vi) A forest is a set of 'n'>0 disjoint trees.

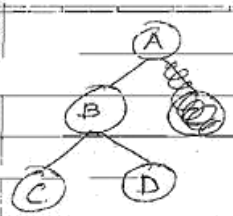
* Binary Tree :-

Def^n A binary tree is finite set of nodes which is either empty or consist of root and two disjoint binary tree called left subtree & right subtree.

Eg :- a)



b)



* Data Structure of binary tree :-

Structure BTREE

declare ISEMTBT ( ) → Boolean
CREATE ( ) → btree
MAKEBT (btree, item, btree) → btree
LCHILD (btree) → btree
RCHILD (btree) → btree
DATA (btree) → item

all l, r ∈ btree and d ∈ item let.

ISEMTBT (CREATE) ::= true
ISEMTBT (MAKEBT (l,d,r)) ::= false
LCHILD (MAKEBT (l,d,r)) ::= l
RCHILD (CREATE) ::= error
LCHILD (CREATE) ::= error.
DATA (MAKEBT (l,d,r)) ::= d
DATA (CREATE) ::= error

end

end.

{ Create — Creates a binary tree.
Make bt — Makes a new binary tree by adding l & r trees with d as.

root node }.

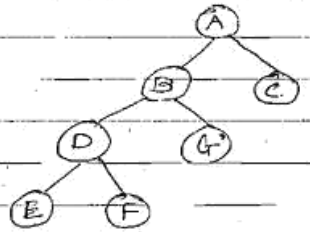
* Examples of binary tree :-

i)

ii)

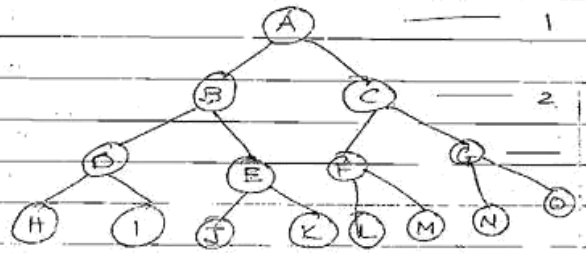
Above fig. shows 2 binary trees. In 1st binary tree the right subtree is empty. Such binary tree is known as ~~sq~~ skewed binary tree to left.

ii)



For this binary node each node have zero subtree or 2 subtree. Such BT is known as complete binary tree.

iii)



In this binary tree have either 2 subtree or zero subtree.

* All the nodes having ~~have~~ zero subtree are at same level. Such a binary tree is known as full binary tree

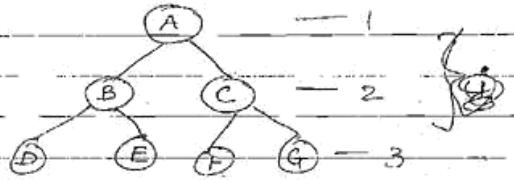
a) { In full binary tree, (n) no. of node in particular level = 2^{n-1} where n = level. }.

b) The max. no of nodes in full b.t of depth k = $2^{k+1}-1$

* Binary tree representation :-

i) Array representation :-

A full binary tree of depth k is a binary tree having $2^{k+1}-1$ nodes. In another word to represent binary tree of depth k the array size should be equal to $2^{k+1}-1$



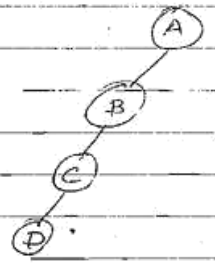
Size = $2^3 - 1$ = 7 nodes = max array size

1	2	3	4	5	6	7
A	B	C	D	E	F	G

lchild = 2 x i (i = index no. of array)

rchild = (2 x i) + 1

parent = $\frac{i}{2}$

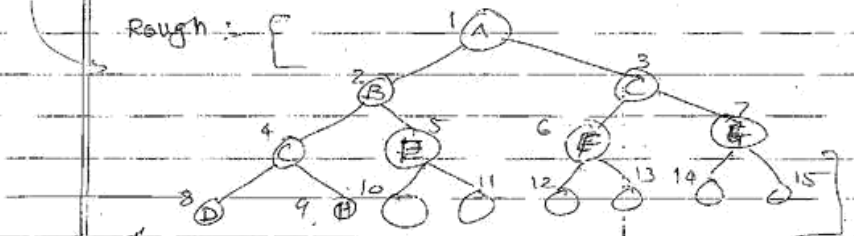


array size = $2^4 - 1$

= 15

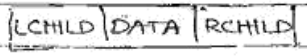
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	-	C	-	-	-	D	-	-	-	-	-	-	-

Rough :-

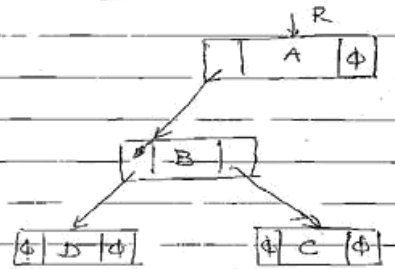


*b) Linked List to represent binary tree :-

Node Structure :-

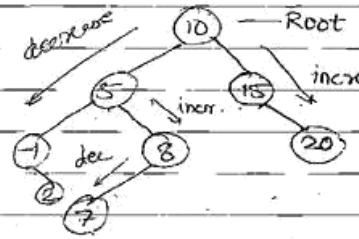


Eg :-



* Creation of Binary Tree :-

Eg :- 10, 5, 8, 7, 15, 20, 1, 2



Node
{ if element < root then left subtree
Node
if element > root then right subtree

* Algorithm to Create & Implement tree :-

CREATEBT ()

{ // R → Root
// Tmp → Node of tree
// x → data

} FIELDS — Data, LLINK, RLINK.

Create Root Node

```
R = NULL
Read x          (data)
GETNODE (R)
DATA (R) = x
LLINK (R) = R(RLINK (R) = NULL
```

```
Print "How many values?"
Read n.
for (1 to n)
{
Read x          — (new data).
GETNODE (Tmp)
DATA (Tmp) = x
LLINK (Tmp) = RLINK (Tmp) = NULL
```

A=R

```
while (1)
{
  if (DATA (T) < = x)
  {
    if (LLINK (T) = NULL)
    {
      LLINK (T) = Tmp;
      break;
    }
    T=LLINK (T)
  }
  if (DATA (T) > x)
  {
    if (RLINK (T) = NULL)
    {
      RLINK (T) = Tmp;
```

```
T= RLINK (T);
}
}
}
```

(4) * Binary Tree Traversal :- (To print binary tree).

There are 3 binary tree traversal algorithm available:-

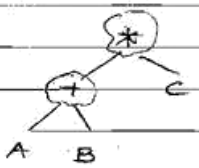
i) Inorder —"—

ii) Post order traversal

iii) Pre order traversal

1) Inorder :- (LDR)

consider B.T.

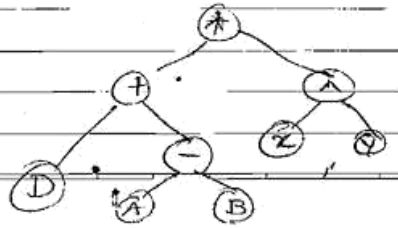


Print → A+B * C

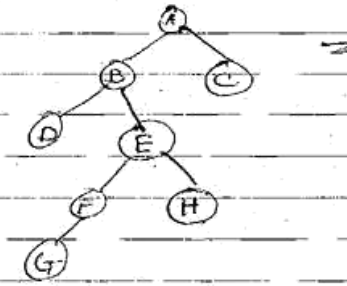
Left | Right

data

ii)



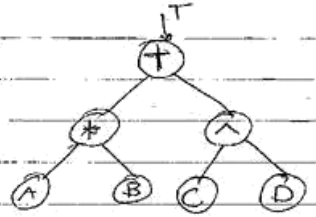
→ D+A-B*z^y



⇒ DBGFEHAC

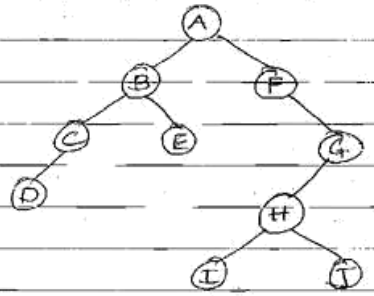
* Postorder :- (LRD) :→

e.g.:-1)



⇒ AB*CD^+

2)



Postorder

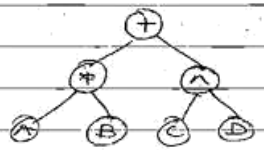
⇒ DCEBIJHGFA

Inorder :- ~~DCEBIJHGFA~~

DCBEAFIHJG

* Preorder :- (~~BR~~DLR) →

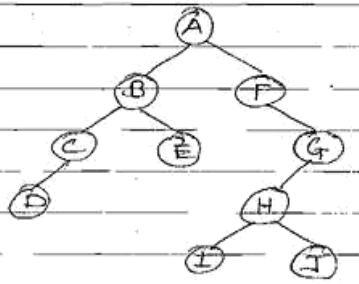
i)



Preorder :-

+*AB^CD

ii)



Preorder :- ABCDEFGHIJ

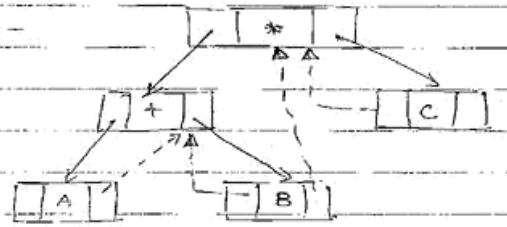
```
* Inorder (T)
{  // T is a pointer to tree
   // Node fields : DATA, RCHILD, LCHILD
   if (T not equal to NULL)
   {Inorder (LCHILD(T))
    Print DATA (T)
    Inorder (RCHILD(T))
   }}

* Postorder (T)
{
   if (T not equal to NULL)
```

```
Postorder (LCHILD(T))
Postorder (RCHILD(T))
print DATA(T)
}
}
```

```
* Preorder (T)
{
if (T != NULL)
{
print DATA(T)
preorder (LCHILD(T))
preorder (RCHILD(T))
}
}
```

we will replace it by pointer to node which would be printed after P when traversing the tree in inorder. Similarly when LCHILD(T) = NULL then it is replaced by a pointer to the node which immediately proceeds the node P in inorder.

eg:-

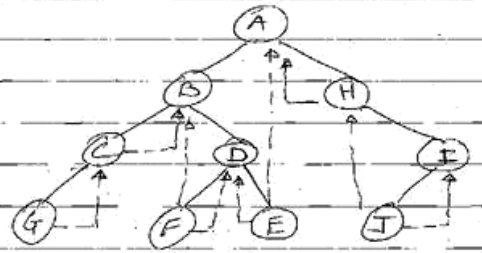


Prog. to create & traverse binary tree in inorder, preorder & postorder.

for eg:- (inorder)

(GCBFDEAHJI

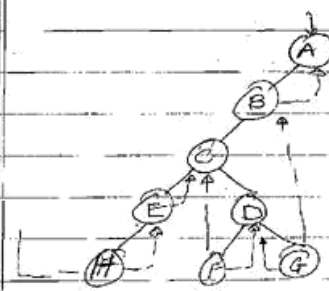
only in NULL fields



* Head Threaded Binary Tree:-

In a binary tree there are many null links at the leaf nodes. In threaded binary tree this NULL links can be utilised by a pointer known as threads. This threads are pointing to nodes as follows :-

If (RCHILD (P) is equal to NULL)

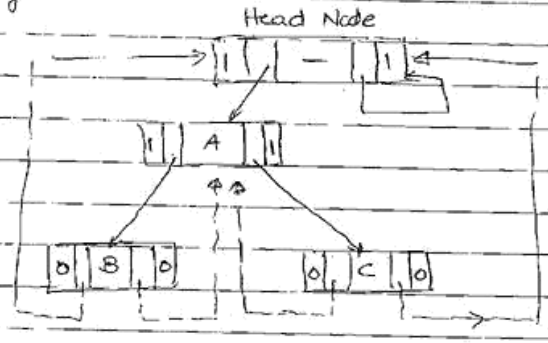


Inorder – HECFDGBA.

* i) Head node is used for the effecient programing.
ii) NULL conditions are avoided.

* To Distinguish betn the Thread & pointer two extra bit fields known as 1bit & 2bit are added to the structure of node. If LCHILD(P) or RCHILD(P) is a normal pointer then LBIT(P) is set to 1. If LBIT(P) or RBIT(P) is thread then it is set to 0.

eg:-

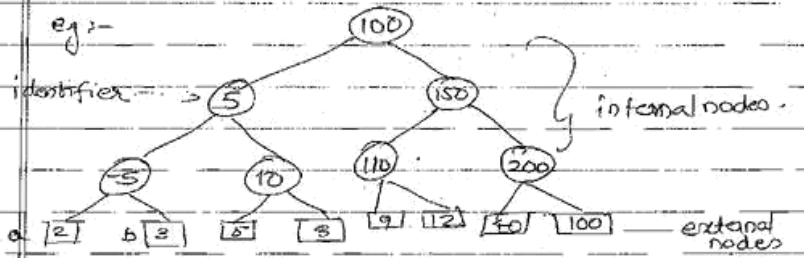


* Advantage:-
i) It simplifies inorder, preorder, postorder algorithms.
ii) In a binary tree if there are n nodes then there are (n+1) NULL links but in threaded binary tree this (n+1) NULL links are also utilized.

* Binary Search Tree:-

Def:- A binary search tree (T) is a binary tree, either empty or each node in the tree contains an identifier and,
i) All identifiers in left subtree of tree are less then the identifier in root node of T.
ii) All identifiers in right subtree of tree are greater than the identifier in root node of T.
iii) Left & right subtrees of T are also binary search tree.

eg:-



To search and identify 'x' in binary search tree x is compared with root. If x < identifier then search continues in left subtree. If x = identifier the search terminates.
In evaluating binary search tree it is useful to add special nodes

at every place there is NULL link known as external node, denoted by square boxes.

Each time a binary tree is examined for an identifier which is not in tree then search terminates at external node, i.e we can say that search is unsuccesful; hence external nodes are also known a "failure node"

* External path lenght :-

It is the path length of binary tree. It is evaluated by sum over all the external nodes of the lengths of path from the root to these nodes.

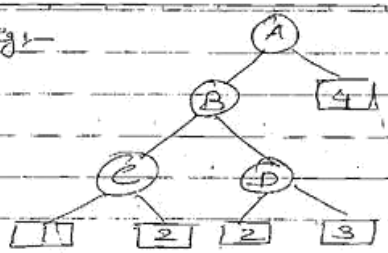
eg. External path length of a = 3.
Internal path length of -5 = 2

* Weighted external path length :-

It is the summation of $\sum_{1 \le i < n+1} q_i k_i$, where k_i is distance from root node to external node with weight q_i.

Wt. ext path length = $2 \times 3 + 3 \cdot 3 + 5 \times 3 + 8 \times 3 + \dots + 100 \times 3$

eg:-



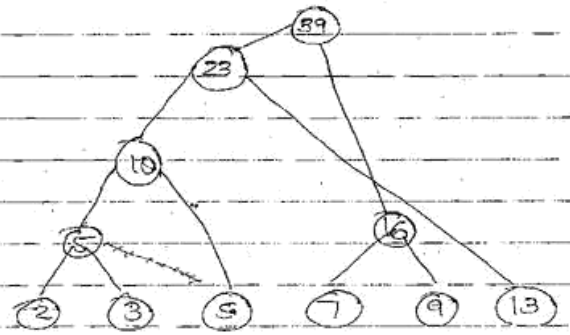
Wt. external path length = $1 \times 3 + 2 \times 3 + 2 \times 3 + 3 \times 3 + 4 \times 1$
$= 28$.

* Binary search tree with external path length :- minimal weighted

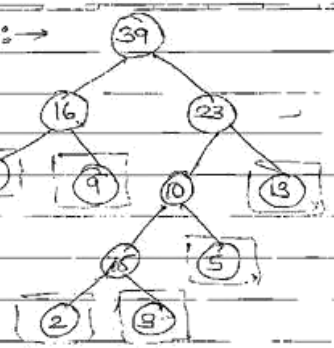
* Huffman algorithm to construct minimal or optimal weighted ext path length (binary tree).

eg:- $a_1 = 2$, $a_2 = 3$, $a_3 = 5$
$a_4 = 7$, $a_5 = 9$, $a_6 = 13$

(a)



(b) Final :→



Optimal Binary Tree

8) Construct minimal weighted external path length for above ex.

Minimal wt path length = 7×2+9×2+2×4+3×4 + 5×3+13×2 = 93

* Height Balance Binary Search tree:-

Defn:- An empty tree is height balanced. If T is non empty binary tree with T_L & T_R as its left and right subtrees, then T is height balance, if

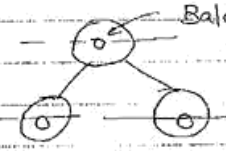
i) T_L and T_R are height balanced and,

ii) $|h_L - h_R| \leq 1$

where h_L & h_R are the heights of T_L & T_R respectively.

* Balance factor of a tree 'T' denoted by of a node 'T' in a binary tree 'T' is defined to be $h_L - h_R$, where h_L & h_R are height of left and right subtrees of T'. Balance factor always bet^n −1, 0 & +1

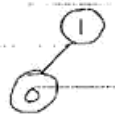
ex:- a) H = 0 ($h_L - h_R$)



Balance factor $(h_L - h_R)$.

$h_L = 1$ $h_R = 1$ H = 0 (∵ balanced)

b) balanced tree

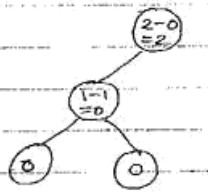


$h_R = 0$

$h_L = 1$

∴ H = 1 (∵ balanced)

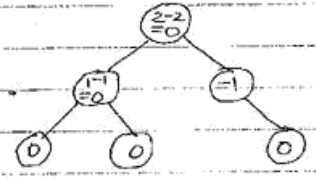
c)



Ht. of tree H = 2

∴ Tree is unbalanced

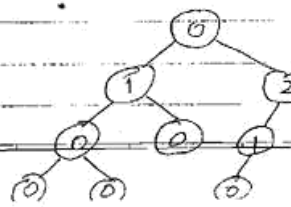
d)



Ht. of tree bet^n e & f 3-3 = 0

∴ balanced.

e)



* unbalanced

* When height of tree is unbalanced Re-balancing can be carried out by 4 types of rotatn :-

i) L-L
ii) L-R
R-L
& R-R

This rotatn are cha' by the nearest ancestor (A) of inserted node 'y' whose balance factor becomes ±2.

i) L-L :- New node 'y' is inserted in left subtree of left subtree of 'A'.
ii) L-R :- New node 'y' is inserted in right subtree of left subtree of 'A'.
iii) R-L :- New node 'y' is inserted in left subtree of right
iv) R-R :- in right subtree of right subtree.

* SEQUENTIAL FILE HANDLING :-

* Basic Operat° :-
i) Create
ii) Open
iii) Close

Additional* → i) Read → Search
ii) Write → addition.
iii) Read & write → update
iv) Deletion.
↳ a) Reorganisation.
b) Flag method.

* Assgn -9 :- /* Sequential file processing */

```
Struct STUD
{
    int Roll;
    char Name[30];
    char class[6];
} s;
char FILENAME[30];

main()
{
    char choice;
    while (1)
    {
        clrscr();
```

```
        printf ("\n Create");
        printf ("\n Open");
        printf ("\n Write");
        printf ("\n Search");
        printf ("\n Update");
        printf ("\n Delete");
        printf ("\n Exit");
        choice = getch();
        switch (choice)
        {
          case 'C' : Create();
                     break;
          case 'O' : Open();
                     break;
```

Before (main)

```
void Create (void)
{
  FILE *fp;
  printf ("\n Enter the name to Create");
  scanf ("%s", FileName);
  fp = fopen (Filename, "w");
  if (fp == NULL)
  {
    printf ("FILE CREATION ERROR");
    return;
  }
  fclose (fp);
}
```

```
void Open (void)
{
  FILE *fp;
  printf ("\n Enter the filename");
  scanf ("%s", FileName);
  fp = fopen (Filename, "r");
  if (fp == NULL)
  {
    printf ("CREATE FILE FIRST");
    return;
  }
  fclose (fp);
}

void Write (void)
{ File *fp;                                   or (a)
  fp = fopen (FileName, "r+");
  if (fp == NULL)
  {
    printf ("CREATE FILE FIRST");
    return;
  }
  printf ("\n Enter the Record");
  scanf ("%d %s %s", &s.Roll, s.Name,
         s.class);
  fseek (fp, 0, SEEK_END);
  fwrite (&s, sizeof(struct STUD), 1, fp);
  fclose (fp);
}
```

```
void search (void)
{
    FILE *fp; int n;
    fp = fopen (FILENAME, "r");
    printf (" Enter the Roll no to be
            searched");
    scanf (" %d", &n);

    while ( !feof (fp))
    {
        fread (&s, sizeof (struct STUD), 1, fp);
        if (s.Roll == n)
        {
            printf (" Record is = %s, %s", s.Name,
            fclose(fp); s.Class);
            return;
        }
    }
    printf (" Not Found");
    fclose(fp);
}

void update (void)
{
    FILE *fp; int n; long pos;
    fp = fopen (FILENAME, "r+");
    if (fp == NULL)
    ---
    2
```

```
    printf (" \n Roll no. of Record to be
            update");
    scanf ("%d", &n);
    while (! feof (fp))
    { pos = ftell (fp);
      fread (&s, sizeof (struct STUD), 1, fp);
      if (s.Roll == n)
      {
        printf ("\n Reenter the record of %d", n);
        scanf (" %s %s", s.Name, s.class);
        fseek (fp, pos, SEEK_SET);
        fwrite (&s, size of (struct STUD), 1, fp);
        fclose (fp);
        return;
      }
    }
    printf ("\n RECORD NOT FOUND");
    fclose (fp);
}

Void Delete (void)   [skip if Roll no = 0
                      assume deleted]
    SEARCH the roll no

    s.Roll = 0
    fseek (fp, pos, SEEK_SET);
    fwrite (&s, sizeof (---), 1, fp);
    fclose (fp);
    return;
}
```

```
void Listing (void).
{
    FILE *fp;
    int n;
    fp = fopen (Filename, "r");
    if (fp == NULL)
    {
    }
    while (! feof (fp))
    {
        fread (&s, size.of(struct STUD), 1, fp);
        if (s.Roll ! = 0)
        {
            printf ("In Roll No = %d, Name = %s
                Class = %s ", s.Roll, s.Name, s.class);
        }
    }
    fclose (fp);
    return)
}
```

Hash Table :-

Hash function.

pos = f(x)

buckets	slot 1	slot 2
1	a_1	a_2
2	b_{10}	b_1
3	a_5	c_1
26	z_2	

where $x = a_1, a_2, b_{10}, b_1, z_2, a_5, c_1$

collision

overflow element

overflow.

Hash functions :-

i) Divide & remainder method.

Roll no	Data
0	
1	
2	
3	
9	

size of array/struct

For Roll No 1 :-

formula :- Roll = (Roll - 1) mod 10

it returns 0 (i.e first roll no)

ii) Folding :-

acc → 1 3 3

caa → 3 1 1

pos = 100 + 30 + 3 = 133

pos = 300 + 10 + 1 = 311

iii) Mid & square :-

Q) * A hash table is used to store 12 records. The size of hash table is 6. Assume total slots = 2. Hash funct is divide & remainder. Key values are :- 10, 100, 87, 5, 12, 0, 9, 65, 50, 14, 21. Show the status of hash table after inserting key values in order.

OR. HASH METHOD USING LINEAR PROBING

N = 6.
Slots = 2.

	1	2
0	12	0
1	7	(21) ←
2	8	50
3	9	(14) ←
4	10	100
5	5	65

Hash f = Key mod N
funcn value

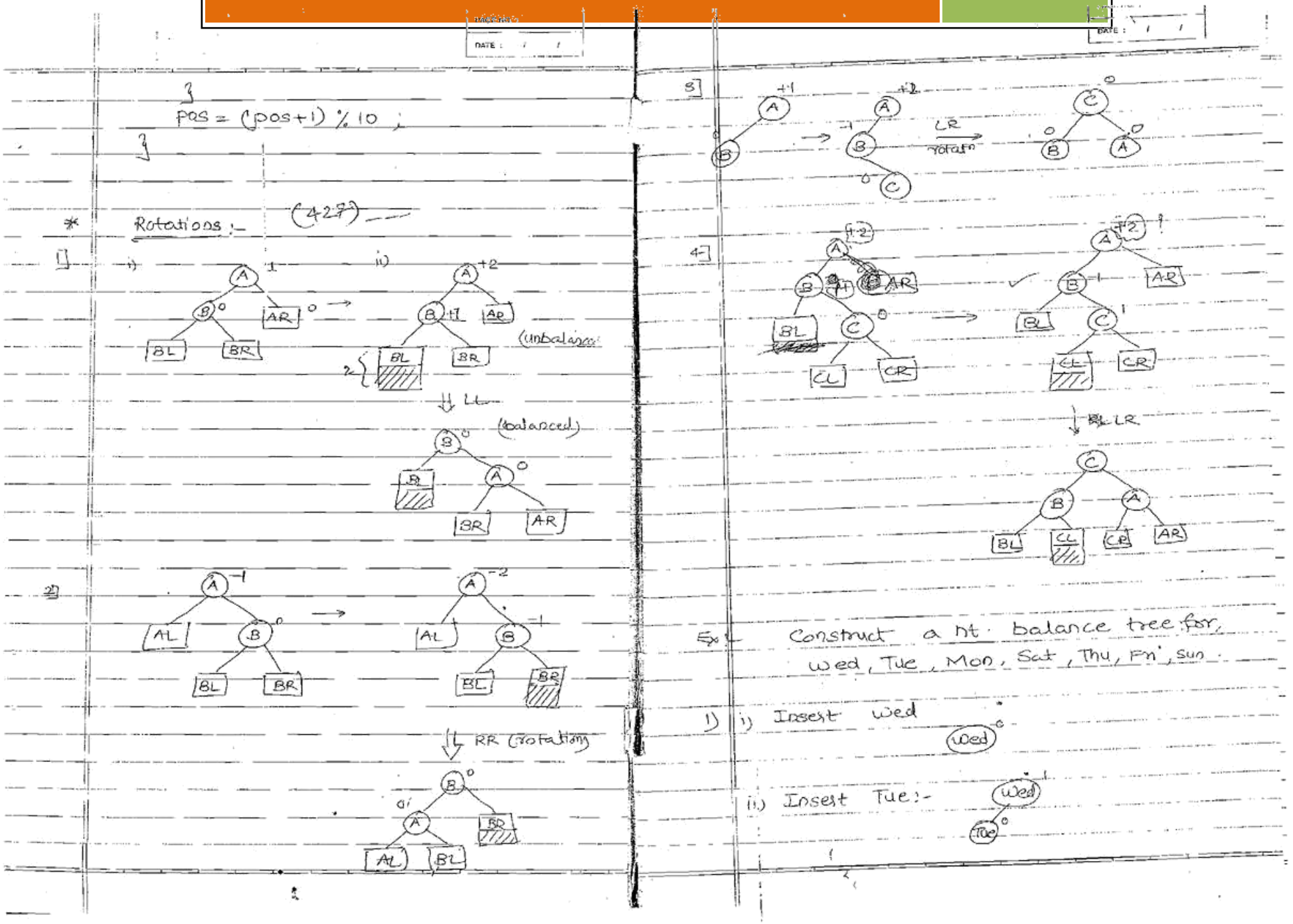
overflow eg :- (10 % 6 = 4
10 will be in 4th slot)

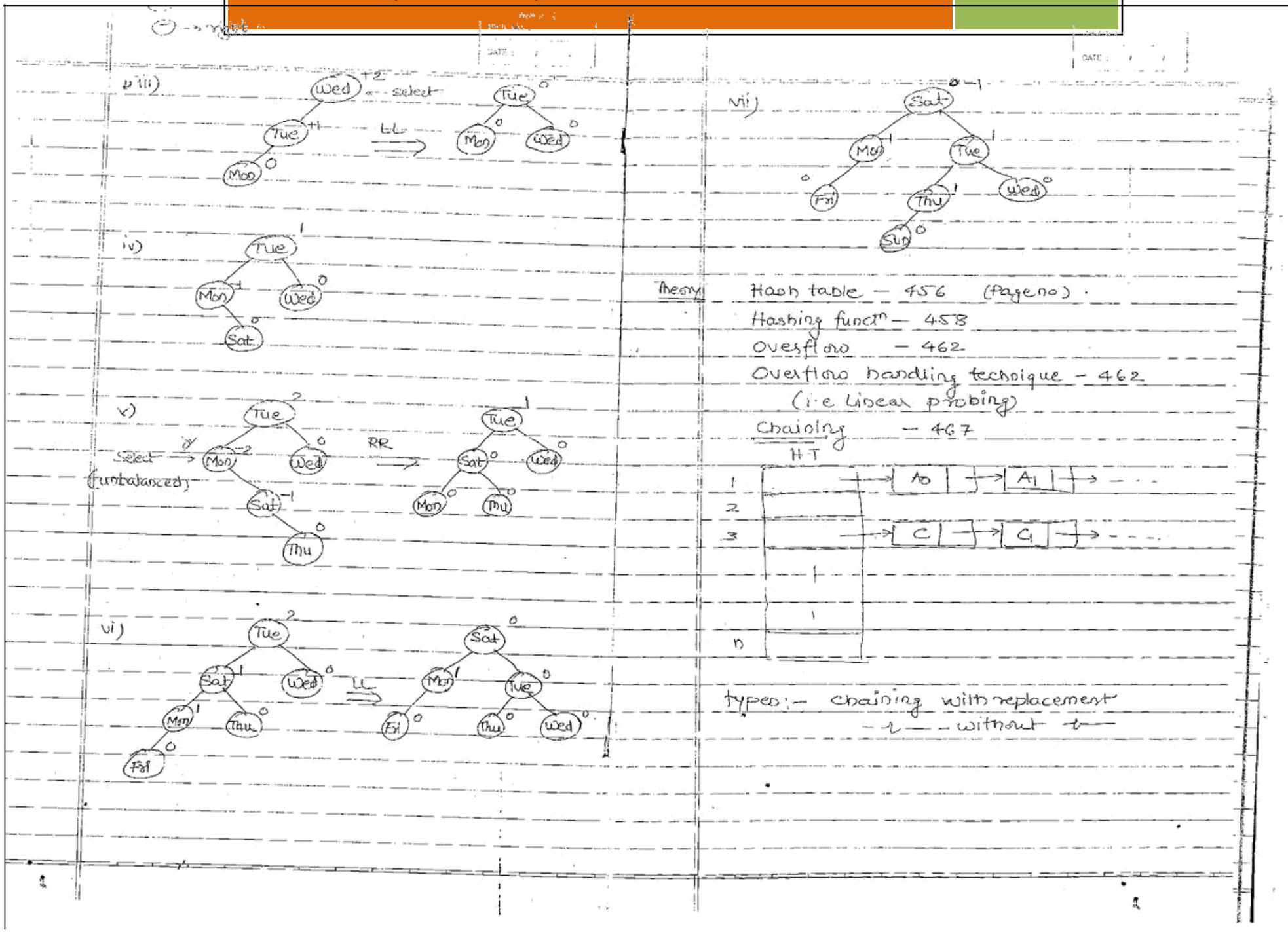
As hash table is Circular (21 mod 6 = 3) 21 is in posⁿ 3rd)

LINEAR PROBING is used to search Given overflow element. [For this jump to original position & then search until the pointer reaches the original position].

Asse - 10

```
/* Program to Create Hash table */
struct                          { Assume marks are
{                                 bet 0 to 99.
int marks;                        size = 10
char Name[30];                    slots = 2
} HT[10][2];

Void Insert(void)
{   pos,i,j
    int m; char n[30]
    printf("Enter the Marks & Name");
    scanf("%d %s", &m, n);
    pos = m % 10;
    for(i=0; i<2; i++)
    {
        if (HT[pos][i] * marks == -1)
        {
            HT[pos][i].marks = m;
            strcpy(HT[pos][i].name, n);
            return;
        }
    }
    pos = (pos+1) % 10
    for(i=0; i<10; i++)
    { for(j=0; j<2; j++)
      { if (HT[pos][0] == -1)
        {
            -1,
        }
        break;
```


pos = (pos+1) % 10 ;
Rotations :-
(427)
(unbalanced)
LL
(balanced)
RR (Rotation)
LR
Ex:- Construct a bt balance tree for,
wed, Tue, Mon, Sat, Thu, Fri, sun.
i) Insert wed
ii) Insert Tue:-


iii)
Wed
+2
select
Tue
+1
LL
Mon
Tue
Wed
Mon
0
iv)
Tue
Mon
Wed
Sat
v)
Tue
Select
(unbalanced)
RR
Mon
Wed
Sat
Thu
Tue
Sat
Mon
Thu
vi)
Tue
Sat
Wed
LL
Mon
Fri
Sat
Mon
Tue
Fri
Thu
Wed
Sun
vii)
Sat
Mon
Tue
Fri
Thu
Wed
Sun
Theory
Hash table – 456 (Page no).
Hashing funct^n – 458
Overflow – 462
Overflow handling technique – 462
(i.e Linear probing)
Chaining – 467
HT
1
2
3
n
A0
A1
C
C1
types:– chaining with replacement
– without –

DATE : 23/03/2009

* Continued in 'C-book'.

* Index sequential File :-

a) The record can be access using index. Index is nothing but a structure that contain the two information i) key value, ii) Location of Record.

b) Searching is faster as compared to seq., relative & direct access file.

Disadv → Delet^n is complicated.

* Primitive operat^n :-

i) All operat^n are same to as direct access file.

* Linear index file :-

If the index file is ordered by value (ascending / descending) then such file is known as linear index file.

* Tree indexes :-

Tree index is one in which their is herrarchy of index. The root of 1st level of the index points to record level of the index. Each level of the index pts to the lower level until the lowest node, (leaf node) are reached. The leaf node have pointers only to data file.

Tree indexes are classified into

DATE : / /

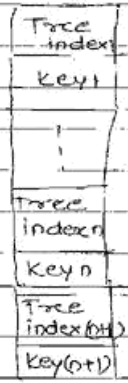
2 Types :-

i) Heterogenous tree index
ii) Homogenous

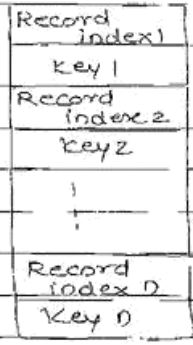
i) Heterogenous tree index :-

a) Structure of Node :-

All the each nodes except the leafnode contains two fields as shown below.



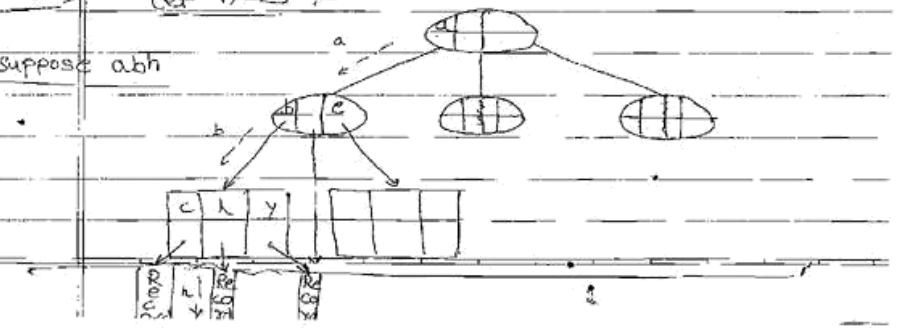
— structure of internal node.



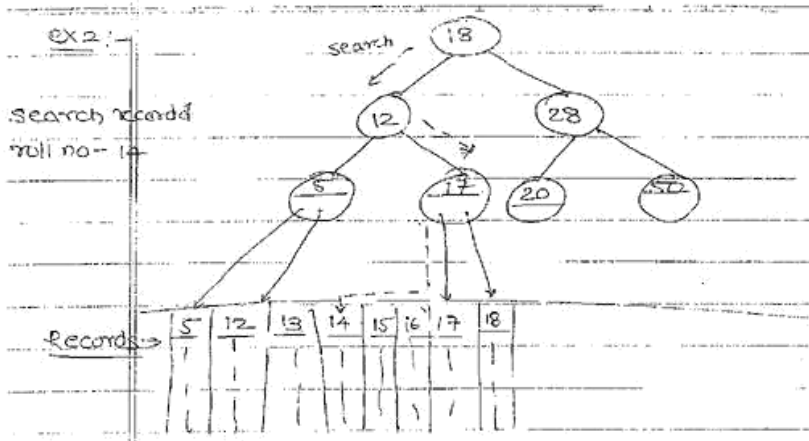
— structure of leaf node.

ex1) let n=3 :-

Suppose abh

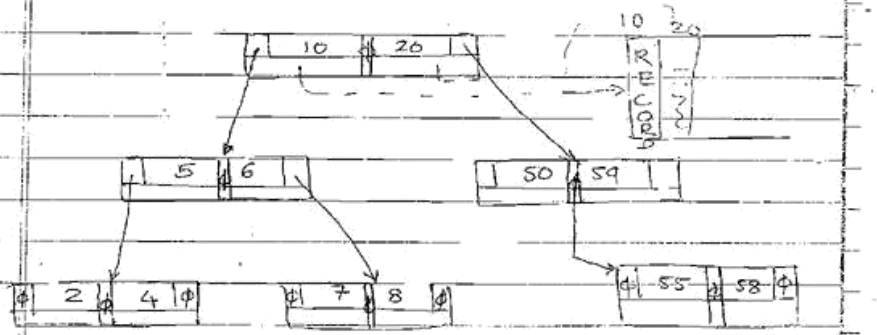


Ex 2 :



Search record with roll no = 14

pt. is used to access record, otherwise corresponding tree pointer is used to search the next node & process is repeated.



* Homogenous tree index :-

a) Structure of node →

Tree pointer 1	Each node contains 2 types of data pointer & other is tree pointer. Generally no. of keys is equal to no. of data pointers. To search any record searching begins from 1st record. Given key is compared with key value in node. Result will be either match is found or value lies betn 2 keys. If the match is found then corresponding data
Key 1	
Data pointer 1	
Tree pointer 2	
Key 2	
Data pointer 2	
⋮	
Tree pointer n	
Key n	
Data pointer n	
Tree pointer n+1	
Key n+1	
Data pt n+1	

Data pointers = No. of keys.

Tree pointers = Data pointer + 1

* Searching time is less, as record is found at each level.

*

* Relative File :-

```
Struct RECORD
{
  int Roll;
  char Name[30];
} S;
# define SIZE 70
char FileName[30];
void create (void)
{
   FILE *fp;
   fp = fopen (Filename, "w");
   if (fp == NULL)
   {
     printf ("      ");  }
   S.Roll = 0;
  for (i=0; i<SIZE; i++)
  {
   fwrite (&s, sizeof (struct RECORD), 1, fp);
  }
  fclose(fp);
}

Void Addition (Void)
{ int key, long pos;
FILE *fp  fp = fopen (FileName, "r+")
   if (fp == NULL)
```

check for duplicate records

```
printf ("Enter Roll");
Scanf ("%d", &key);
pos = (key -1) * sizeof (struct RECORD);
fseek (fp, pos, SEEK_SET);
fwrite printf ("Enter the name");
Scanf ("%s" S.Name);
S.Roll = key;
fwrite (&s, sizeof (struct RECORD), 1, fp);
fclose (fp);
}

void Search (Void)
{
int key, long pos;
FILE *fp;
fp = fopen(FileName, "r+");

printf ("Enter the Roll No");
Scanf ("%d", &key);
pos = (key-1) * sizeof (struct RECORD);
fseek (fp, pos, SEEK_SET);
fread (&s, sizeof (struct RECORD), 1, fp);
if (S.Roll == 0)
{
    printf ("Record is deleted");
    fclose (fp);
    return;
}
printf ("Roll No = %d, Name = %s", S.Roll, S.Name);
```

```
    fclose (fp);
}

* Index Seq. File :-

struct Index
{ int Roll
  long pos;
} I;

Struct RECORD
{
int Roll   /* Key */
char Name [30];
} S;
Indexfile
char Filename [30]; Datafile [30];

void Create (void)
{ FILE *fpIndex ; * fpData;
  printf (" Enter Index file & Data Filename"
  scanf ("%s %s", Indexfile, Datafile);
fpIndex = fopen (IndexFile, "w");

fpData =

  fclose (fpIndex);
  fclose (fpData);
}

void Addition (void)
{
long pos;
FILE * fpIndex, * fpData;
fpIndex = fopen (IndexFile, "a");
fpData = fopen (DataFile, "a");

printf (" Enter Roll");
scanf (" %d", & S.Roll);
printf (" Enter Name);
scanf (" %s " S.Name);
fwrite (& S, sizeof (---), 1, fpData);
pos = ftell (fpData);
pos = pos - sizeof (struct RECORD);
I.Roll = S.Roll;
I.Pos = Pos;
fwrite (&I, sizeof (struct Index), 1, fpIndex);
fclose (fpData);
fclose (fpIndex);
}

S.Name for storing only Name in Data-file

void Search (void)
{
  FILE *fpIndex, * fpData;
  long pos;
  fp Index = fopen (IndexFile, "r");
  fp Data =
  printf (" Enter Roll No");
  scanf (" %d ", & S.Roll);
  while ( ! feof (fpIndex)
  {
    fread (& I, sizeof (struct Index), 1, fpIndex
    if (I.Roll == key);
    {
      fseek (fpData, I.Pos, SEEK_SET);
      fread (& S, sizeof (struct RECORD), 1, fpData);
```

```
printf("Roll No = %d, Name = %s", S.Roll, S.Name);
g - fclose(---); return;
} /* End of while */
printf("Not Found");
fclose(fp); (fp Index);
}
```

B-Tree :-

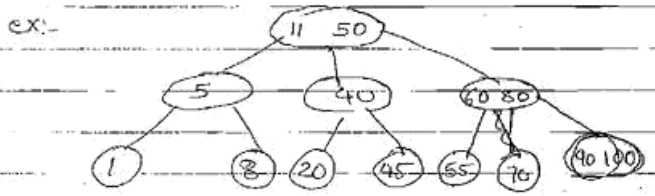
A B-Tree of order of 'm' is homogenous tree that is either empty or is of height greater than equal to one & satisfying the foll. properties,

i) The root node has atleast 2 children.

ii) All other nodes, (other than root nodes) and leaf node (failure node) have atleast n/2 children.

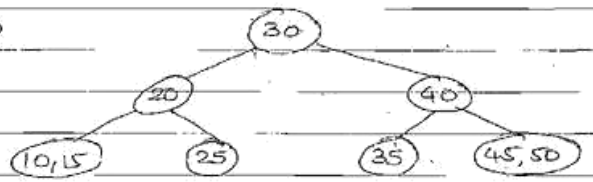
iii) All the leaf node are at the same level.

ex:-

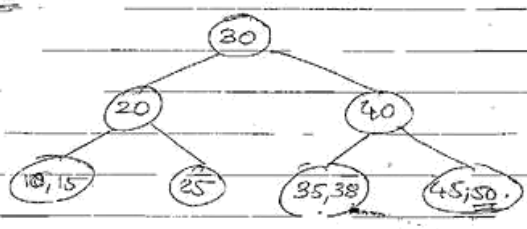


3-way search tree (i.e m=3)

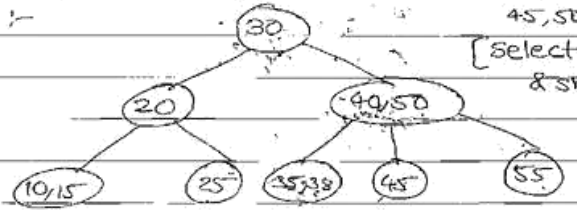
Foll. fig shows B-tree of order 3. Insert foll. key values in that tree. Key values — 38, 55, 37, 5, 18, 12



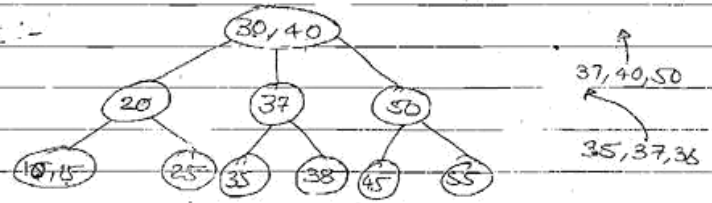
i) 38



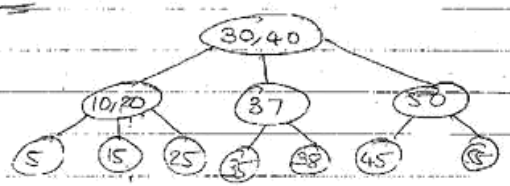
ii) 55 :-



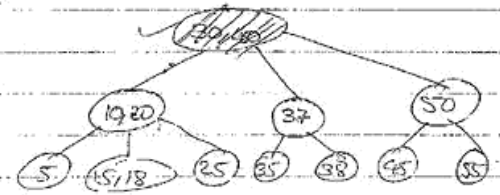
iii) 37 :-



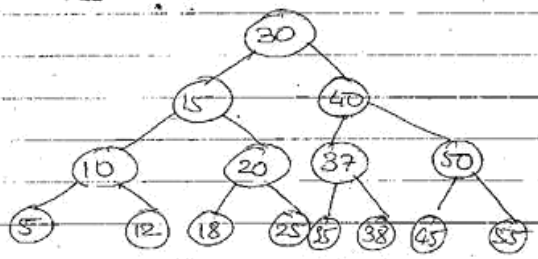
iv) 5



v) 18

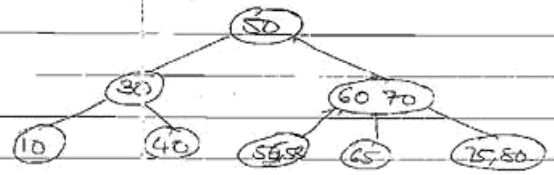


vi) 12.

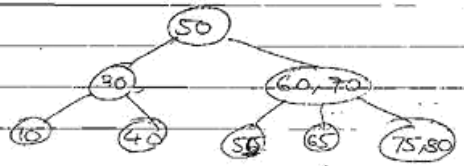


* Foll. fig. shows a btree of order 3. Delete the foll. key values, 58, 65, 55 & 40

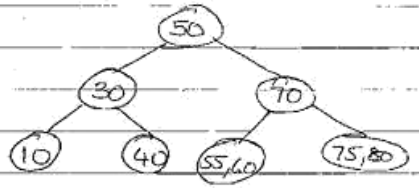
⇒



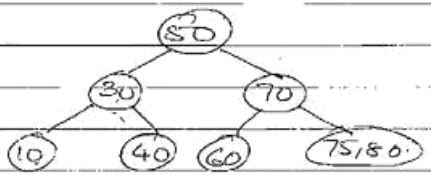
i) 58 :-



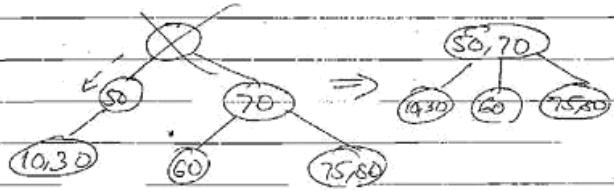
ii) 65 :-



iii) 55 :-



iv) 40



* **Overflow File :-**

In direct access file when a new identifier is inserted into a full bucket then overflow occur. A simple soln to handle this soln is to place a new indentifier into another file known as overflow file. overfile file can be organised as sequential/ direct access file.

* **Rehashing :-** It is the different method of handling the overflow. It requires 2 hashing functions. Main hashing funcn is applied to identifier first, if the slot is already occupied then overflow occur. In this situatn the 2nd hashing funcn is applied.

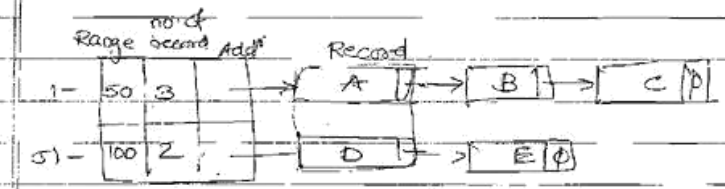
* **MultiList File organisation :-**

To provide the facility of searching on each field of record it is necessary to maintain several index. One index is for each field. Again several record with same key values are possible. To reduce the searching time & size of index file following criteria is used :-

i) If key is a primary key then the values of primary keys can be grouped into the ranges, so that searching time can be minimised.

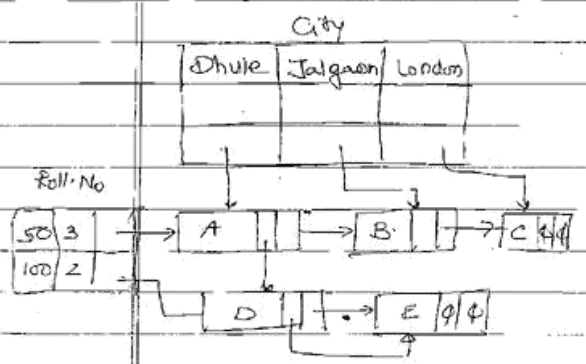
eg :-

# Record	Roll No	Name	City
A	10	xyz	Dhule
B	12	pqr	Jalgaon
C	50	abc	London
D	80	a1	Dhule
E	82	a2	Kesthi Dhule



ii) If key is secondary :-

Let (City) is the index in above ex.

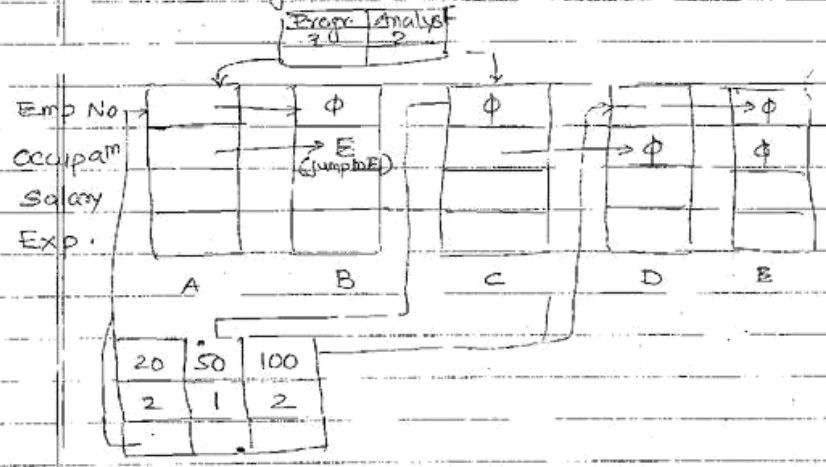


A file in which records that share the value of fields linked together is called the Multilist file.

* Construct the Multilist file of the given record below:- Assume employee no. range 10-20, 21-50, 51-100

Record No.	Employee no.	Occupatn	Salary	Experience
A	10	Programmer	8,000	4
B	15	-"-	12,000	8
C	50	Analyst	10,000	10
D	80	-"-	12,000	12
E	90	Programmer	12,000	12

Table @.



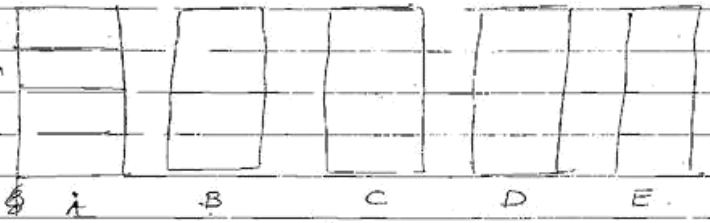
Also do it for salary, & exp.

* Inverted file :-

It is similar to multilist file. The diff. is that in the record with same key value are linked together with link informatn being kept in the individual records. But in inverted file the link informatn is kept in index itself so that the index entries are variable in length.

Ex :- Consider Table @.

20	50	100
2	1	2
A	C	D
B	-	E



Programmer	Analyst
A, B, E	C, D

DS_2000

ORIGINALITY REPORT

%7	%6	%3	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.southfayette.org Internet Source	%4
2	www.roseburg.k12.or.us Internet Source	%2
3	E. H. EASON. "The type specimens and identity of the species described in the genus Lithobius by F. Meinert, and now preserved in the Zoological Museum, Copenhagen University (Chilopoda: Lithobiomorpha)", Zoological Journal of the Linnean Society, 8/1974 Publication	%1

EXCLUDE QUOTES ON EXCLUDE MATCHES < 4 WORDS