* Data Structure :- Logical organization of data in a particular manner.

or The logical organization of data to solve a particular problem in a particular manner is called data structure.

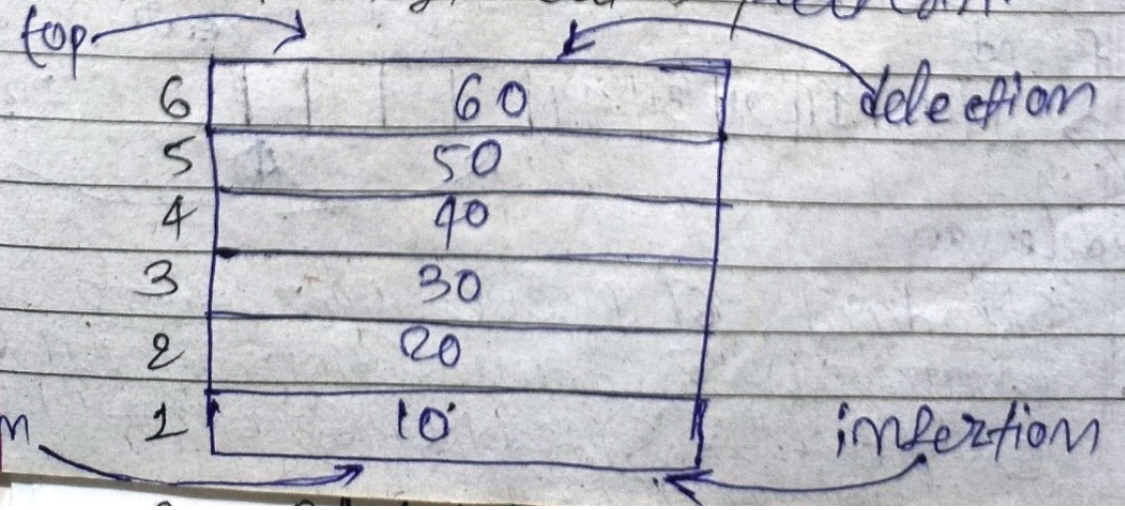
* What do you mean by time complexity ?

ans → Rate of growth of time taken by an algorithm to solve a particular problem with respect to input size.

* What do you mean by space complexity.

ans → space complexity refers to the amount of memory or storage space an algorithm uses to solve a problem with respect to input size.

* stack :- Stack is a linear data structure which operates in a LIFO (last in First out) or FILO (first in last out) pattern.



Date / /
Page No.

* Algorithm : Algorithm is a step-by-step procedure for solving a particular problem

* Types of data structure ⟸

(i) Linear
- arrays
- stack
- queue
- linked list

Ex ⇒

(ii) non-linear
- Trees
- graph

Ex ⇒

Q what are the operation perform on array

Ans ⇒ Traversing, Searching, Merging, Insertion, deletion, shorting.

Standard stack operations :-

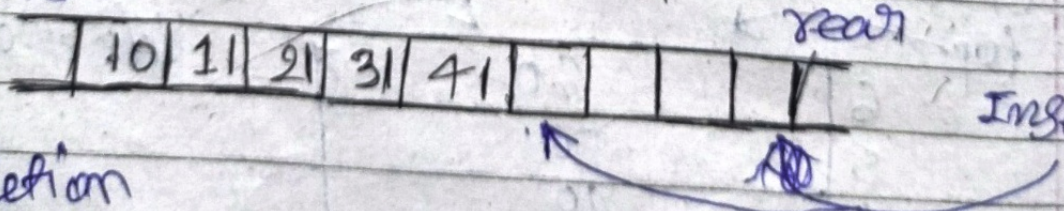
① push() - ② pop() : ③ isEmpty()
↓ ↓ ↓
Insert element — Deletion element — stack is empty

④ isfull() ⑤ peek() ⑥ count()
↓ ↓ ↓
Stack is full — Access the item at the i position — Get the no of item in ~~the~~ stack.

⑦ change() ⑧ display()
↓ ↓
change the item at the i position — Display all items in the stack.

* What is Queue data structure?

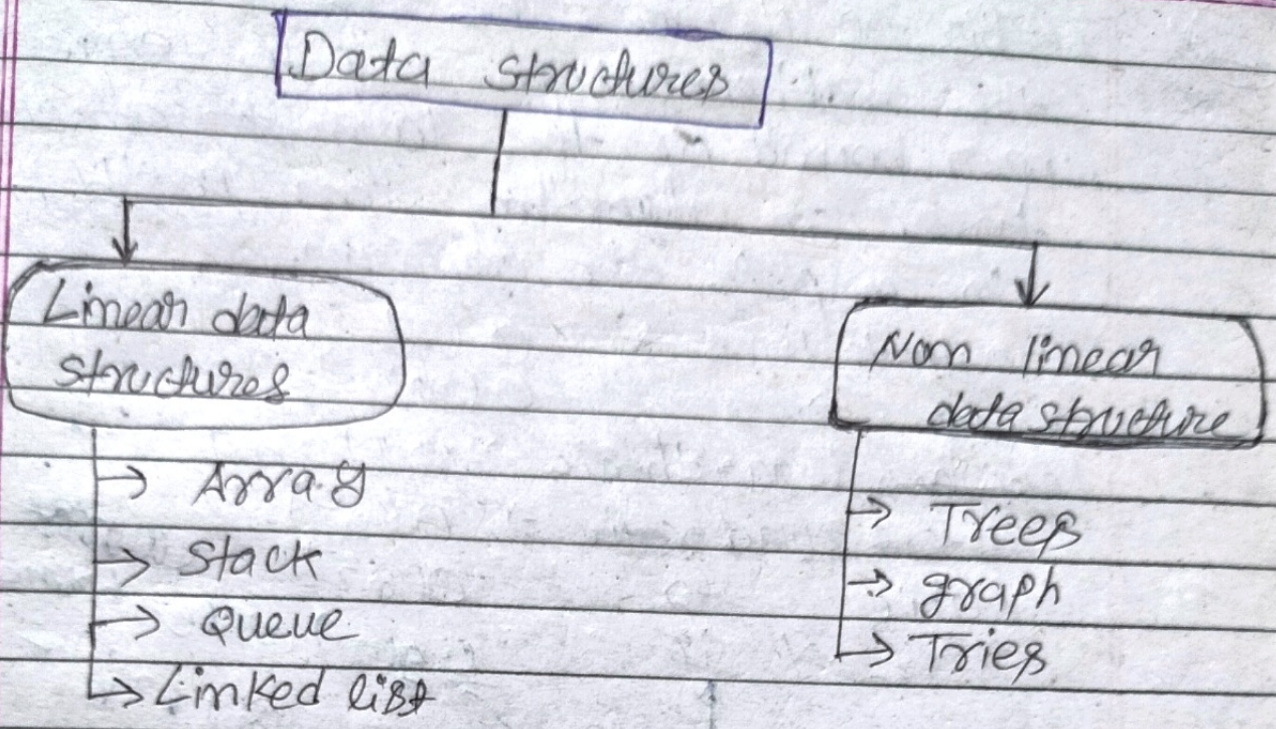
ans⇒ Queue is a linear data structure which operates in a ~~First~~ (FIFO) first in first out or (LILO) last in last out pattern



Standard Queue operations

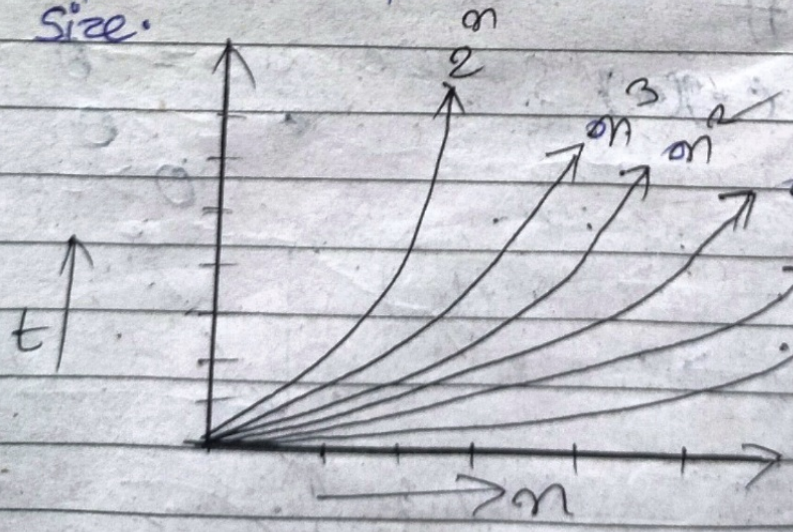
① enqueue() ② dequeue() ③ isempty()
④ isfull() ⑤ peek() ⑥ count() ⑦ change()
⑧ display()

Types of data structure



Complexity of algorithm

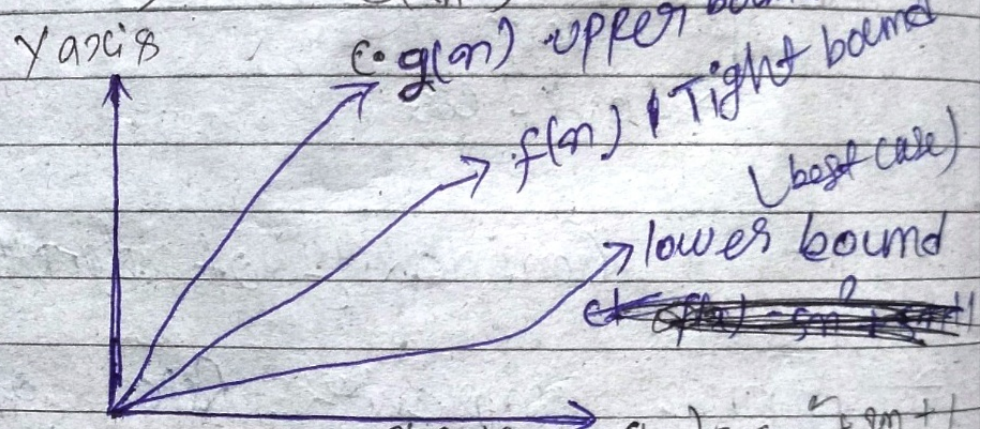
* Time complexity :- Rate of growth of time taken by an algorithm to slove a particular problem with respect to input size.



Asymptotic Notation ($O(n), \Theta, \Omega$)

1. Big O Notation :- The Big O notation is the mathematical way to express the upper bound or the longest amount of time an algorithm can possibly take to complete the program.

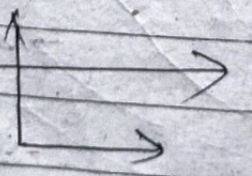
Constant time → $O(1)$
linear → $O(n)$
logarithm logarithmic → $O(\log n)$
quad quadratic → $O(n^2)$
cubic → $O(n^3)$



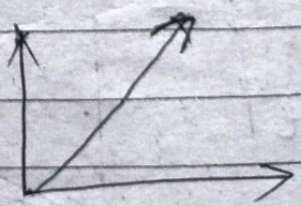
$f(n) = O(g(n))$
Iff

$f(n) <= c \cdot g(n)$ { where $n >= n0$, $c > 0$, $n0 >= 1$

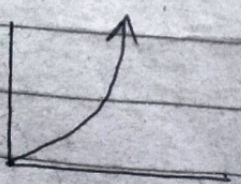
Ex → $O(1)$



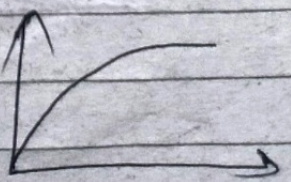
$O(n)$



$O(n^2)$



$O(\log n)$



Questions

$f(n) = 2n^2 + 3n \Rightarrow O(n^2)$

$f(n) = 4n^4 + 3n^3 \Rightarrow O(n^4)$

$f(n) = N^2 + \log n \Rightarrow O(n^2)$

$f(n) = 12\log 1 \Rightarrow O(1)$

$f(n) = 3n^3 + 2n^2 + 5 \Rightarrow O(n^3)$

$f(n) = \frac{n^3}{300} \Rightarrow O(n^3)$

$f(n) = 5n^2 + \log n \Rightarrow O(n^2)$

$f(n) = \frac{n}{4} \Rightarrow O(n)$

$f(n) = \frac{n+4}{4} \Rightarrow O(n)$

Note → (i)
```
for ( )
{ }
```
→ $O(n)$

```
for ( )
{ }
```
→ $O(m)$

TC = $O(n+m)$

(ii)
```
for (0 — N) → O(n)
{ for (0 — N) → O(n)
  { }
}
```
$O(n^2)$

```
for (0 — N) → O(m)
{
}
```
$O(m)$

$O(n^2) + O(m) \Rightarrow$ TC $O(n^2+m)$

* What is the time complexity of the following code

```
a = 0;
for (i=0; i<n; i++)
{ for (j=N; j<i; j--)
  { a = a+i+j; }
}
```

```
for(0 — N)
{ for(0 — N) { }
}
```
$O(n^2)$

* What is the time complexity

```
int a=0 , b =0;
for(i=0 ; i<N ; i++)          } for (0 - N) O(n)
{  a=a + rand();
}
for ( j=0; j<M ; j++)         } for(0- M) O(M)
{ b=b+rand();
}
```

TC = O(n+m)

(3)

```
int a=0 , b=0;
for(i=0 ; i<N ; i++)          } for(0 - N) O(n)
{ for( j=0 ; j<N ; j++)       { for(0 - N) O(n)
  { a=a+j;                    } TC = O(n^2)
}}
for ( k=0 ; k<N ; k++)        -> for(0 - N)
  {  b= b+k;                     O(n)
  }
```

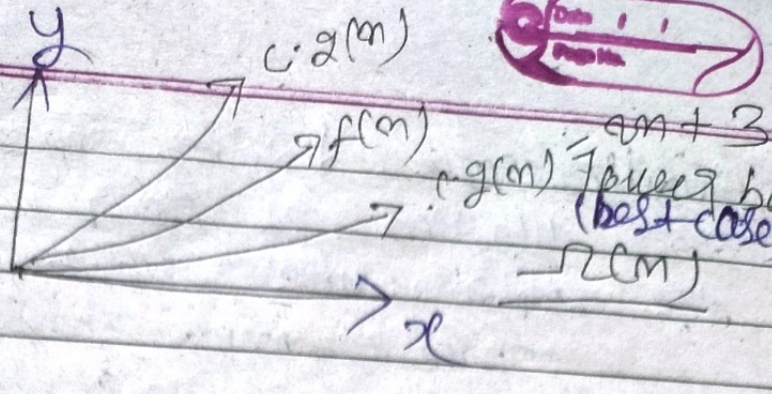
$O(n^2) + O(n) \Rightarrow O(n^2)$

(4) Big Omega (Ω) : Omega notation specifically describes best case scenario it represents the lower bound time complexity of an algorithm.

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

Iff

$f(n) >= c \cdot g(n)$ {where $n > n0$, $c > 0$, $n0 >= 1$



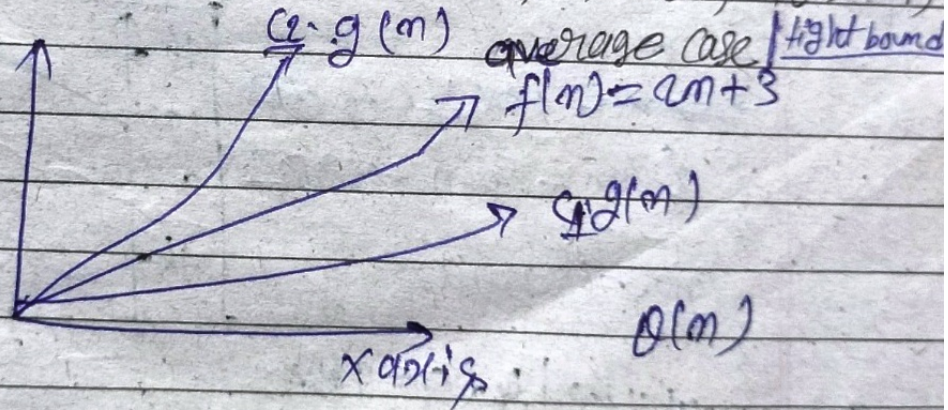
3. Big theta (Θ) :- Big theta notation specifically describes average case scenario, it represents the most realistic time complexity of an algorithm.

$f(n) = \Theta(g(n))$

Iff

$\Theta(g(n)) = C1 \cdot g(n) <= f(n) <= C2 \cdot g(n)$

(where $n > n0$, $C1, C2 > 0$, $n > n0$, $n0 >= 1$)



* Space complexity :- Space complexity refers to the amount of memory an algorithm uses to slove a problem with respect to input size.

Space complexity includes both Auxiliary space and space used by input.

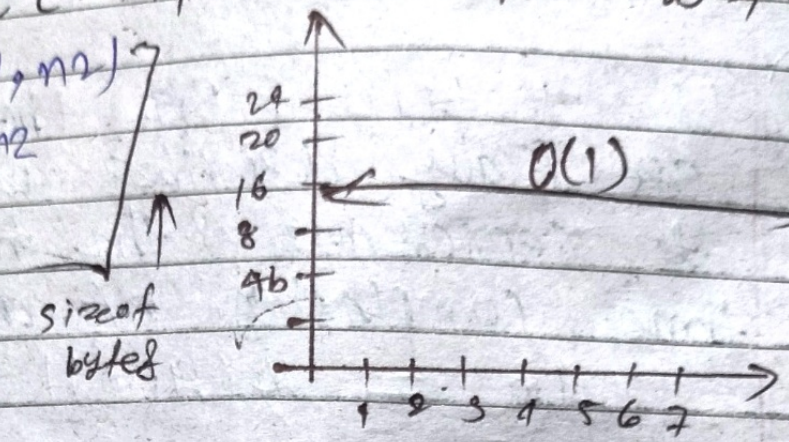
Space C= input size + Auxiliry space

* Auxiliary space: Auxiliary space is the temporary space allocated by your algorithm to solve the problem, with respect to input size.

space C = input size + Auxiliary space

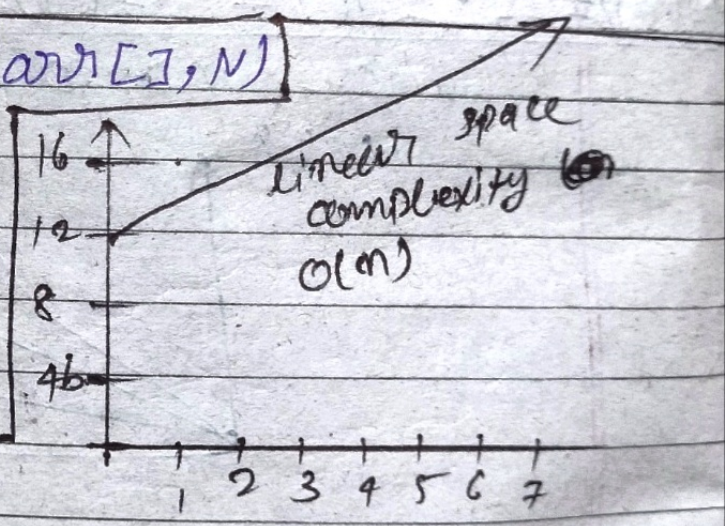
A-1 → fun add(n1, n2)
{ sum = n1+n2
return sum
}

n1 → 4 bytes
n2 → 4 bytes
sum = 4 bytes
Aux sp = 4 bytes
Total = 16 bytes



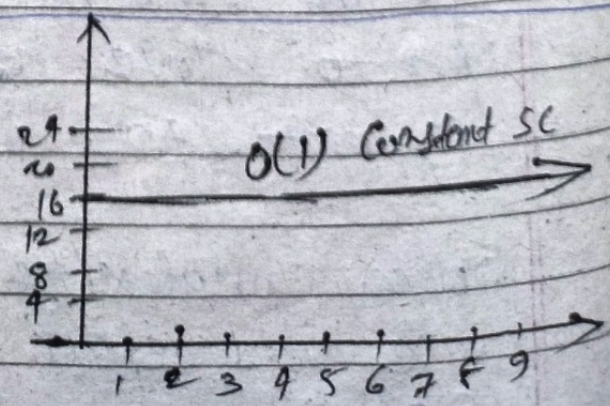
A2 → fun sumofNumbers(arr[], N)
{ sum = 0
for(i=0 to N)
{ sum = sum + arr[i]
}
print(sum)
}

arr → N x 4 bytes
sum = 4 bytes
i = 4 bytes
Aux Sp = 4 bytes
tot = 4N + 12b
O(n)
4(n) + c



A→3→ int fact=1;
for(int i=1; i<=n; ++i)
{ fact *= i; }
return fact;

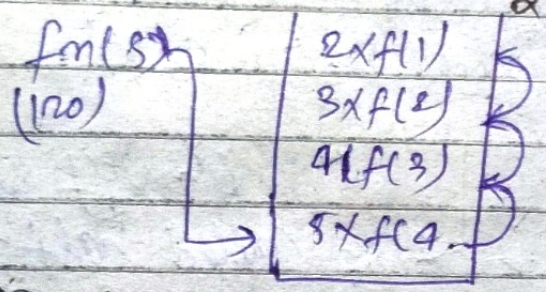
fact = 4 bytes
i = 4 bytes
n = 4 byte
Aux = 4 byte
TSP = 16 by



```
int factorial(int n)
{ if (n <= 1)
  { return 1;
  }
  else {
  return (n * factorial(n-1));
  }
}
```

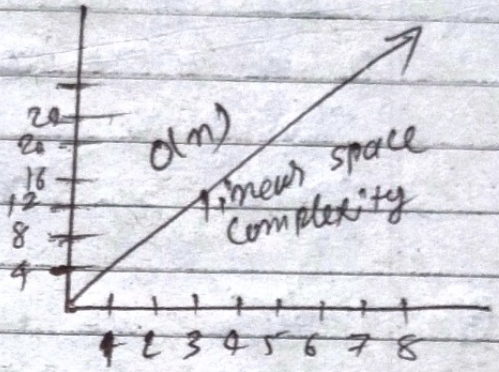
5 * f(4)

// n → 4byte, AUSP = 4byte



Total ⇒ n × 4bytes

S = 4bytes + n bytes



Q what do you meant by time space Tradeoff

ans ⇒ Time space trade off is a way to solve a problem in less time by using more storage space or by solving a problem in very little space by spending a long time.

Q1 differemle b/w linear search and binary search.

	linear search	binary search
(i)	Also called sequential search	Also called half-interval search & logarithmic search
(ii)	less efficiemcy	High efficiemcy
(iii)	less complex than binary	More complex than linear
(iv)	time complexity O(logn)	time complexity O(n)
(v)	Best case to find the element in the first position	Best case to find the element in the middle position.
(vi)	No required of sorted array.	required of sorted array.

Imp Q1. Efficiency of binary search. or How binary search efficient to linear search.

ans → linear search

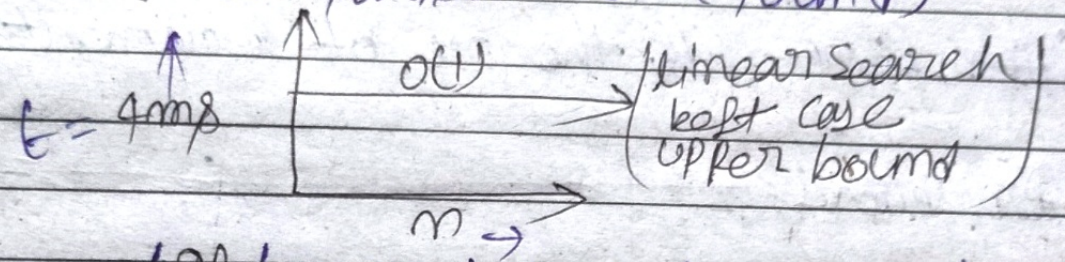
9	10	11	12	13
0	1	2	3	4

size = 5

x = 9

at index = 0 best case O(1)

No of comparison = 1 (found)



(It becomes independent of n)

for worst case → (i) last

(ii) not found

linear search

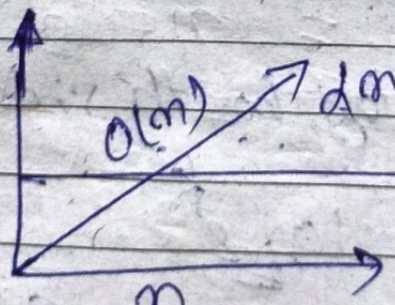
9	10	11	12
0	1	2	3

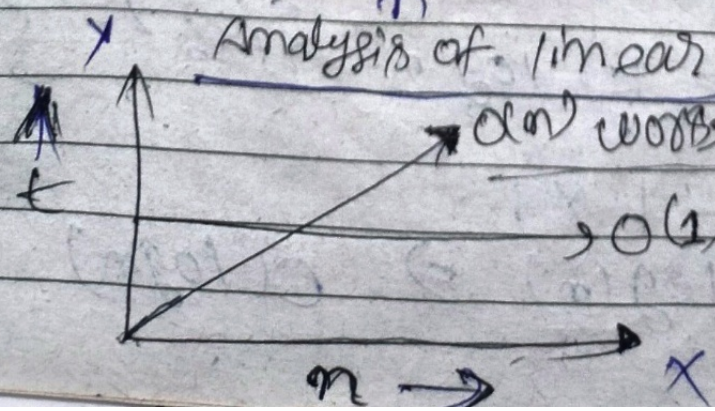
size = 4

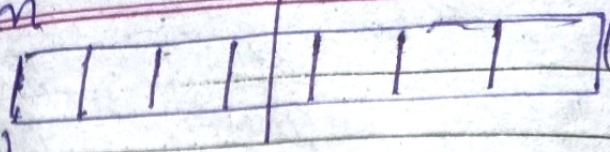
x = 12 No of comparison = 4 → found O(n)

x = 17 No of comparison = 4 → not found O(n)

(size of array)







a[mid] == x ? found

Analysis of binary search

After first call the no of element = $\frac{n}{2} = \frac{n}{2^1}$

After second ——— = $\frac{n}{2} \times \frac{1}{2} = \frac{n}{2^2}$

" third ——— = $\frac{n}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{n}{2^3}$

" nth ——— = $\frac{n}{2^k}$

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

Analysis of worst case

nth call = $n = 2^k$

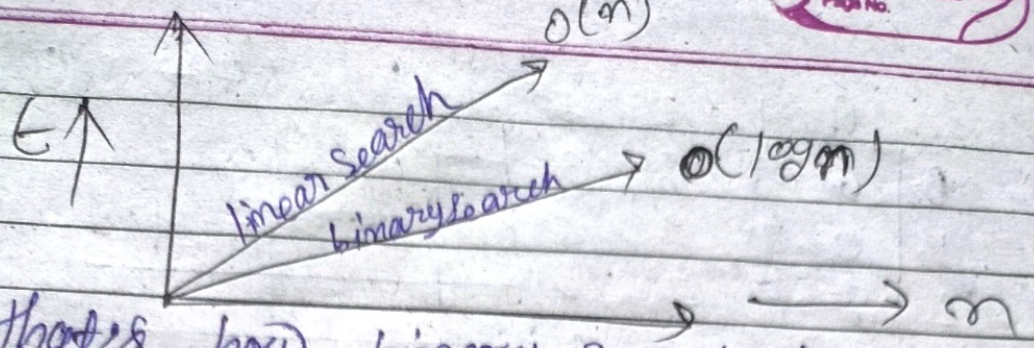
Taking log both side

$\log n = \log_2 2^k$

$\log n = k \cdot \log_2 2$ $\{\log_2 2 = 1\}$

$\log n = k \times 1$

$k = \log(n) \Rightarrow O(\log n)$



That's how binary search is efficient to linear search.

* How a multidimensional array is stored in a computer system. & representation of column and row major order

ans→

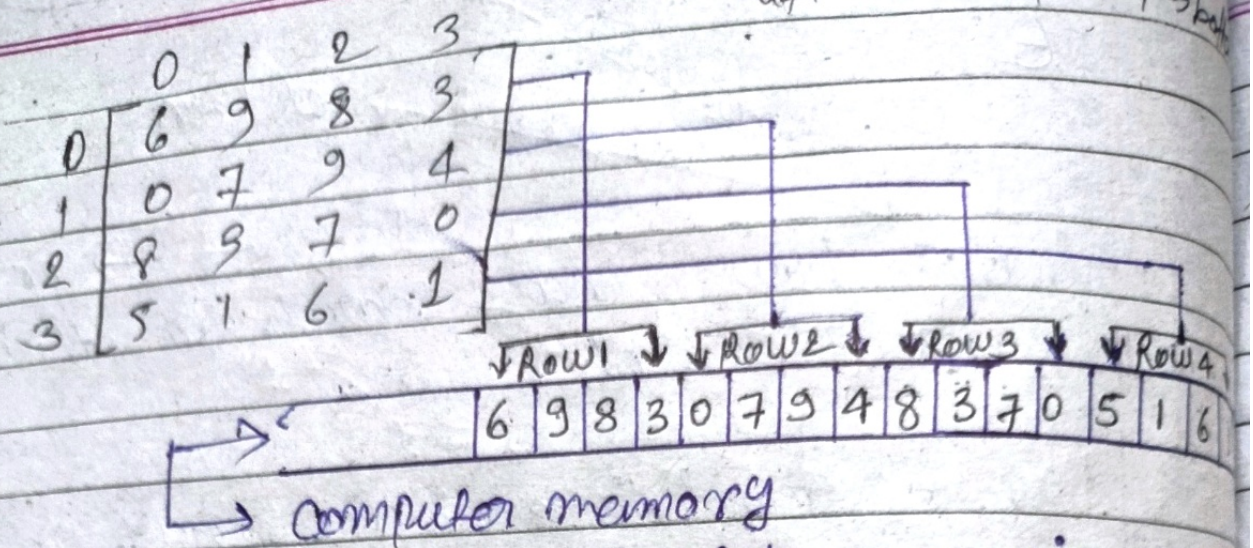
A multidimensional array is stored in a computer system in a linear fashion even if it is multidimensional array because memory is a one dimensional thing.

There are two main ways to store a multidimensional array in memory

(i) Row-major order (ii) Column-major order

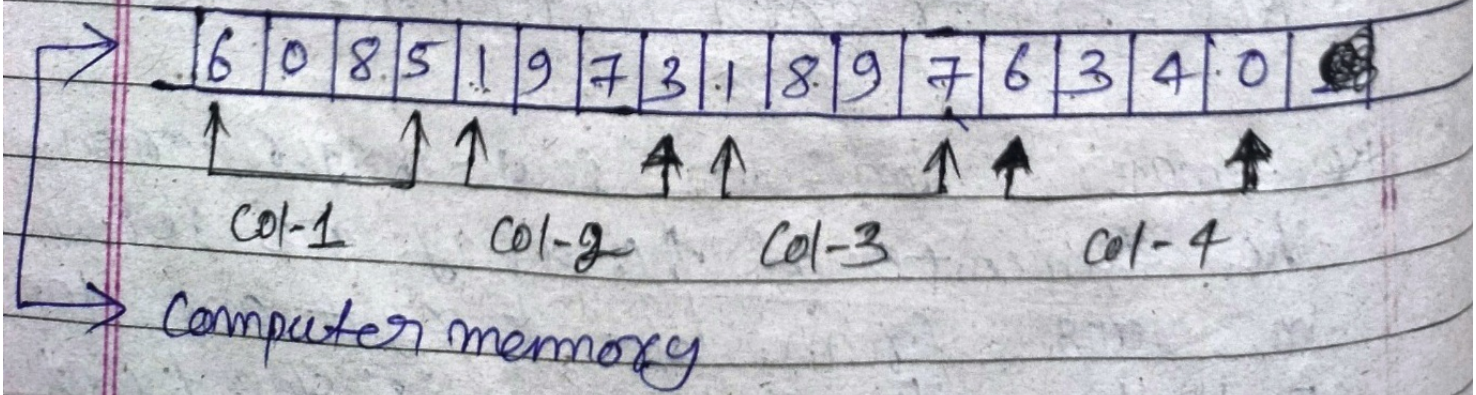
(i) Row-major order: In row-major order, the elements of the array are stored in rows, from left to right and top to bottom. The first row is stored first, followed by the second row, and so on.

left → right / Top → bottom



② Column-major order: In Column-major order the elements of the array are stored in columns, from top to bottom and left to right. The first column is stored first, followed by the second column, and so on.

	0	1	2	3
0	6	9	8	3
1	0	7	9	4
2	8	3	7	0
3	5	1	6	1



* Application of stack

(1) Conversion of infix to prefix and postfix expression using stack.

* Infix Expression: <operand><operator><operand>
(a) (+) (b)
↳ infix

* prefix expression: <operator><operand><operand>
+ab → prefix

* postfix expression: <operand><operand><operator>
ab+ → postfix

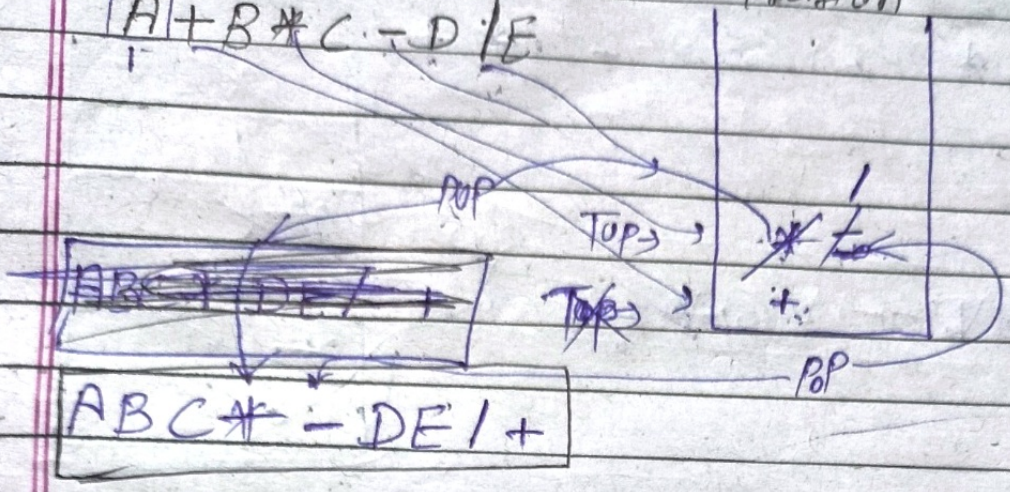
* Rules for convert infix to postfix.

1. Scan Expression from Left to Right.
2. Print OPERANDs as the the arrive.
3. If OPERANDs arrives & stack is empty, Push this operator onto the stack
4. If incoming OPERATOR has Higher precedence than the Top of the stack, push it on stack.
5. If incoming Operator has lower precendance than TOP. ~~Then Start~~ of the stack then POP and Print the TOP. Then test the incoming operator against the NEW Top of stack.
6. If incoming operato has equal precendence with Top of stack, use Associativity rules.
7. For associativity of Left to Right. POP and print the Top of stack, then PUSH the incoming operator.
8. For associativity of Right to Left - PUSH incoming Operator on stack.
9. At the end of Expression, pop & print all Operators from the stack.
10. If incoming SYMBOL is '(' Push it onto stack
11. If incoming SYMBOL is ')' POP the stack and Print Operators till '(' is found & discard it
12. If Top of stack is '(' Push operator on stack

(1) Application

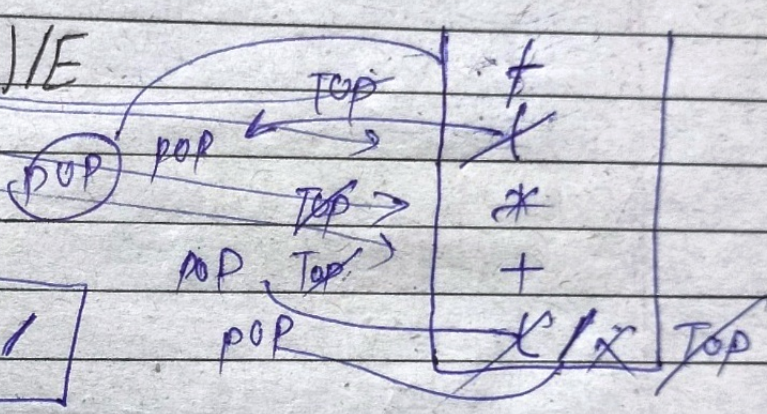
(1) Infix to postfix expression

A+B*C-D/E



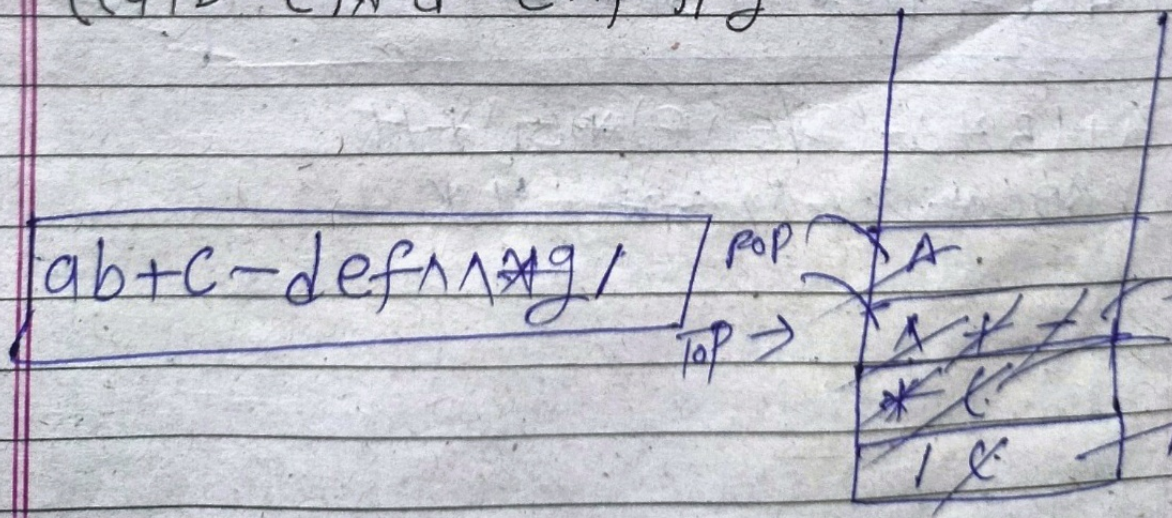
ABC*-DE/+

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



ABCD-*+E/

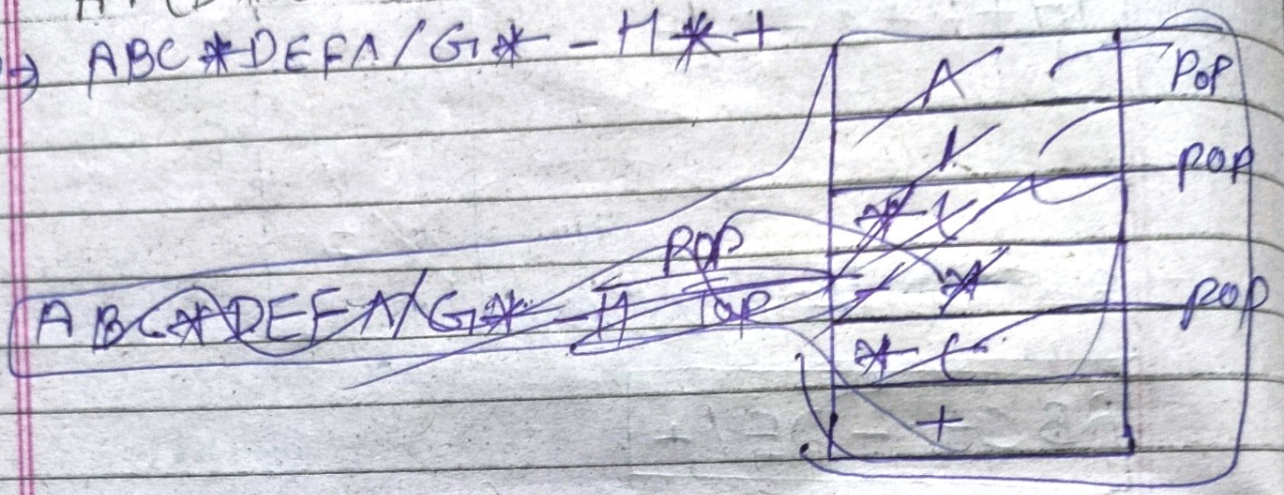
* infix: A+(B*C-(D/E^F)*G)*H
* ((a+b-c)*d^e^f)/g

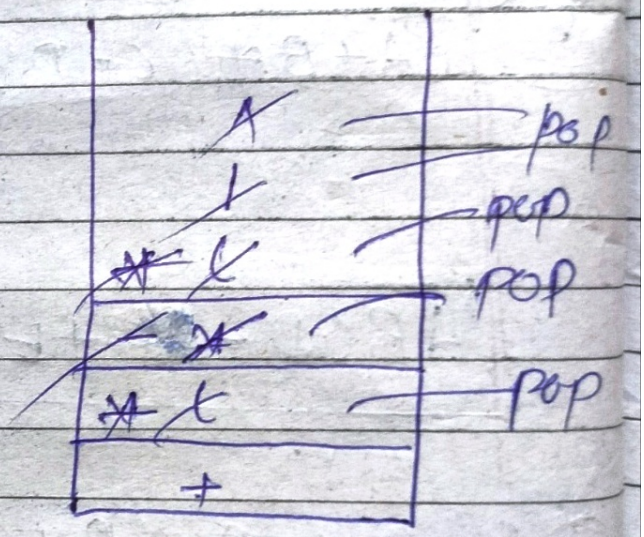


ab+c-def^^*g/

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

ans → ABC*DEF^/G*-H*+





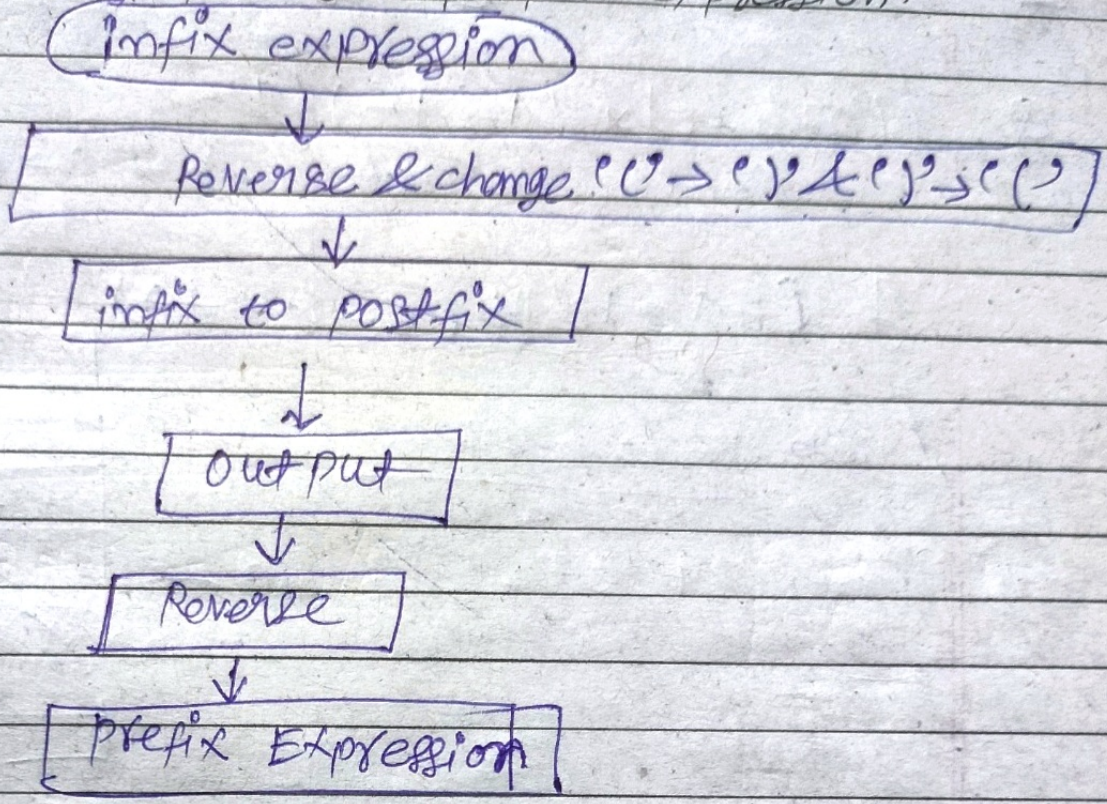
ABC*DEF^/G*-H*+

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

ABC*DEF^/G*-H*+

2. Applic

* Convert infix to prefix expression.



1. Reverse infix expression & swap '(' to ')' & ')' to '('
2. Scan Expression from left to Right.
3. print Operands as the arrive.
4. If operands arrive & stack is empty, Push to stack
5. ~~then the Top of the stack, push it on stack.~~
5. If incoming operator has higher precedence than the Top of the stack, push it on stack.
6. If incoming operator has equal precedence with Top of stack & incoming operator is '^' pop & print Top of stack. Then test the incoming operator against the New Top of stack.
7. If incoming operator has equal precedence with Top of stack, push it on stack.
8. and same as infix to postfix

13. At the end Reverse output string again.

(2) APP

* Convert infix to prefix

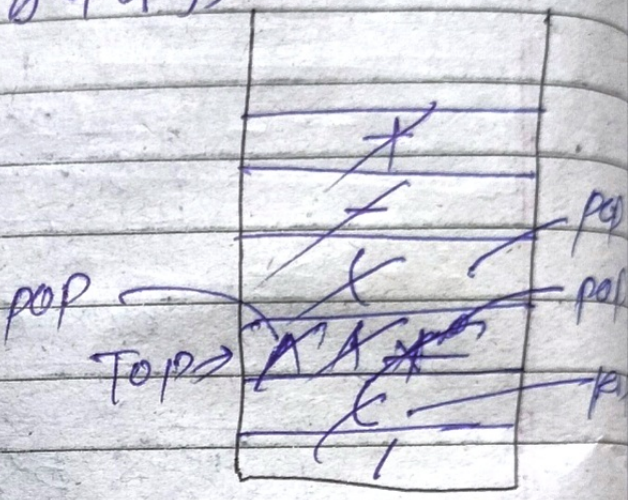
Q ((a+b-c)*d^e^f)/g

Reverse & swap (→) #) → ((

g/((f^e^d*(c-b+a))

↓

C into in to postfix



g f e ^ d ^ c b a + - * / → output

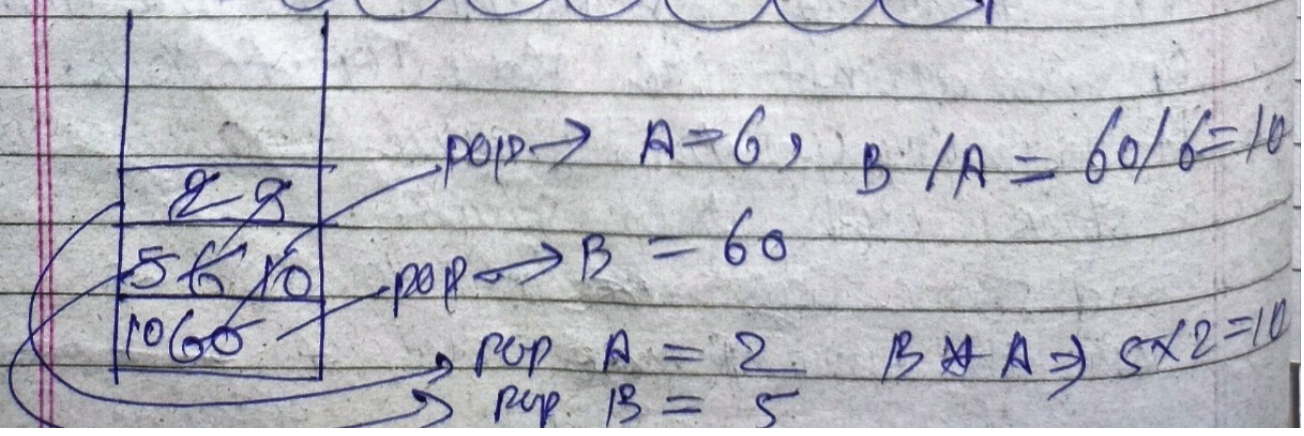
Reverse

/ * - + a b c ^ d ^ e f g

(3) APP

* Evaluation of postfix expression

Q 60, 6, /, 5, 2, *, 5, -, +



A = 5 , B = 10

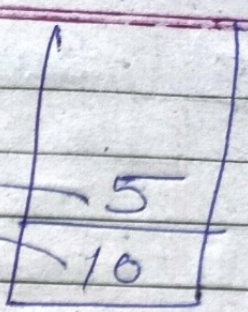
B - A = 10 - 5 = 5

A = 5

B = 10

B + A = 10 + 5 = 15

Answer = 15

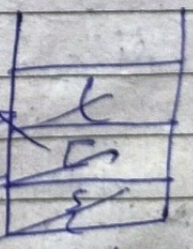


→ Stack में digit value को left से right तक read करते हुए जब भी कोई भी operator मिले तो stack से top से दो element pop कर लो A और B में और दिए गए operator से value find कर लो और फिर इस value को stack में डाल दो फिर आगे जब कोई digit value हो उसे stack में फिर डालो आगे कोई operator हो तो ~~value find~~ दो element pop करो value find करो ~~same~~ फिर store करो Stack में same process.

app

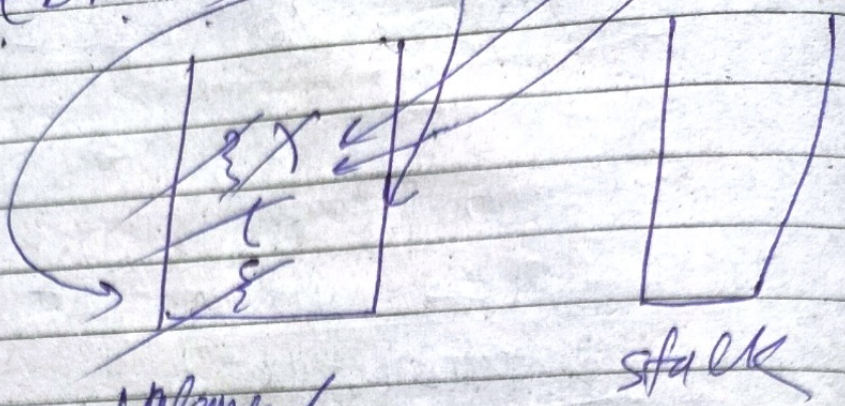
(4) Balanced parenthesis checking

(1) {[()]}



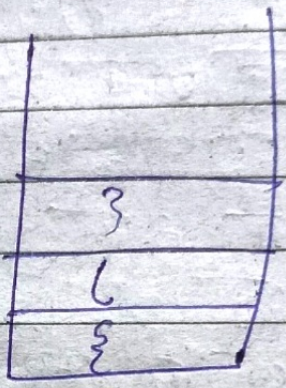
balanced

(ii) {a+(b+c)}



balanced

(iii) {a+(b+c})



Not balanced

app

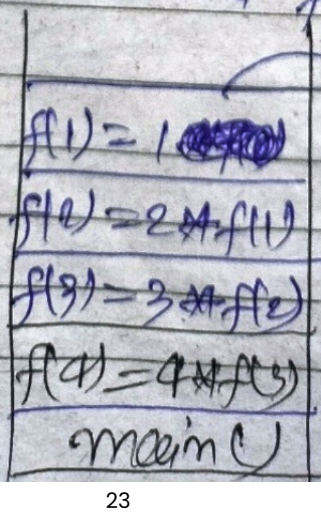
(5) factorial :

```
int factorial (n)
{ if (n==0)
  { return 1;
  }
else {
return n* factorial (n-1);
}
```

```
int main()
{
int x=factorial (4);
cout<<x;
}
```

n=4

f(4) = 4 * f(3)
f(3) = 3 * f(2)
f(2) = 2 * f(1)
f(1) = 1 * f(0)
f(0) = 1



* Why we need to convert infix to postfix expression?

Ans ⇒

Infix expression are readable and solvable by humans. We can easily distinguish the order of operators and ~~also parenthesis to solve~~ we can use the paranthesis to solve that part first during mathematical expression. The computer cannot differentiate the operators and paranthesis easily. that's why we need to convert infix to postfix expression.

Q. write Algorithm that how you insert & delect element in corray.

Sol^n :-

Insert element in corray.

1. start
2. Enter your position of array to insert element.
3. input: pos.
4. pos = pos - 1.
5. for loop → for(j=size-1; 1>=pos; j--)
6. arr[j+1] = arr[j].
7. Enter element to be insert.
8. input element.
9. arr[pos] = ele;
10. print new array. 11. end

To delect element from corray.

1. start
2. Enter position of arry to be delect.
3. input pos.
4. pos = pos - 1.
5. for loop → for(i=pos; i<=size-1; i++)
6. arr[i] = arr[i+1]
7. print new array.
8. end.

Q. ~~Operation on stack, push, pop, overloading~~

Q. Enlist the various operations that can be performed on a data structure.

(1) Traversing :- Visiting each element of a data structure one by one.

(2) Insertion : Adding a new element to a data structure.

(3) Deletion : Removing an element from data structure.

(4) Searching : Finding an element in a data structure.

(5) Sorting : Arranging the elements of a data structure.

(6) Merging : Combining two data structures into one.

(7) Update : Changing the value of an element in a data structure.

(8) Count : Counting the elements which are present in a data structure.

(9) Reversal : Reversing the order of the elements in a data structure.

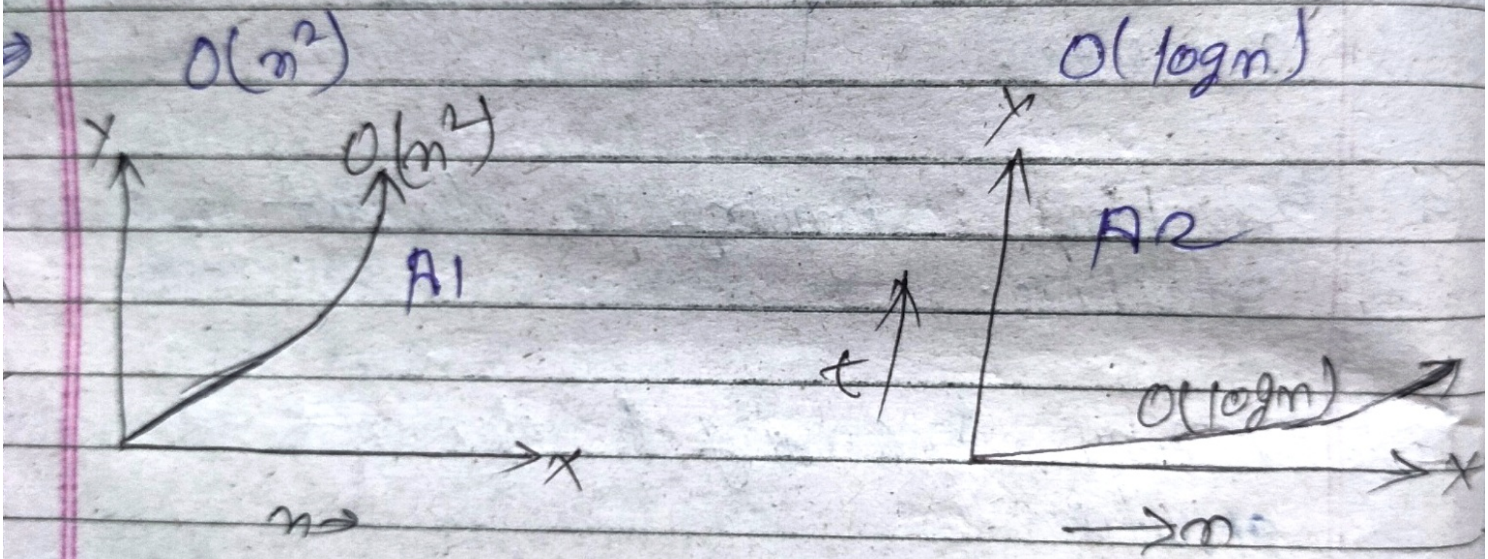
(10) Minimum/Maximum : Finding the minimum or maximum element in a data structure.

(11) Sum/Average : Finding the sum or average of the element in a data structure.

For the same problem P, if you have two different algorithm A1 and A2 with time complexities as follows:

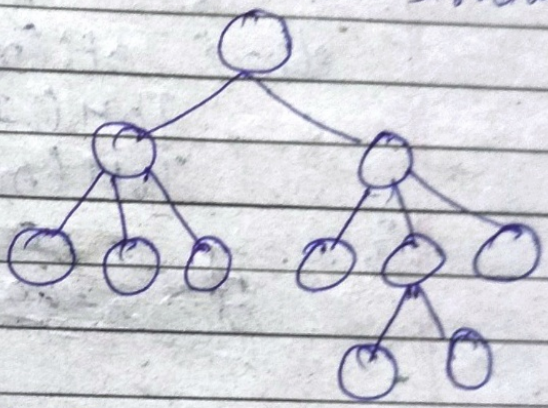
Algorithm A1 ⇒ $O(n^2)$

Algorithm A2 ⇒ $O(\log n)$ which algorithm you will choose?



As we can see A2 has lower bound and A1 has tight bound that's why I choose A2 Algorithm.

Q What is difference between linear and a non linear data structure with example.

	Linear	non-linear
(i)	Examples: Arrays, stack queue, linklist.	Examples: Trees, graph Sets, Tables
(ii)	Linear data structure 	Non Linear data structure 
(iii)	It can be traversed in single run.	It cannot be traversed in a single run.
(iv)	Not very memory efficient	More memory efficient
(v)	Used in simple application	Used in Complex applict
(vi)	data elements are arranged sequentially in a single line.	data elements are arranged hierarchically in multiple levels.

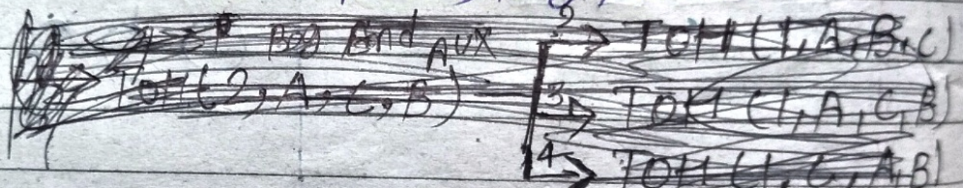
* Demonstrate the use of stack data structure is used to solve toh problem.

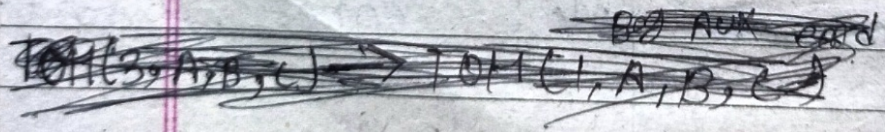
Date / /
Page No.

* Write a Recursive Algorithm to solve Tower of Hanoi problem.

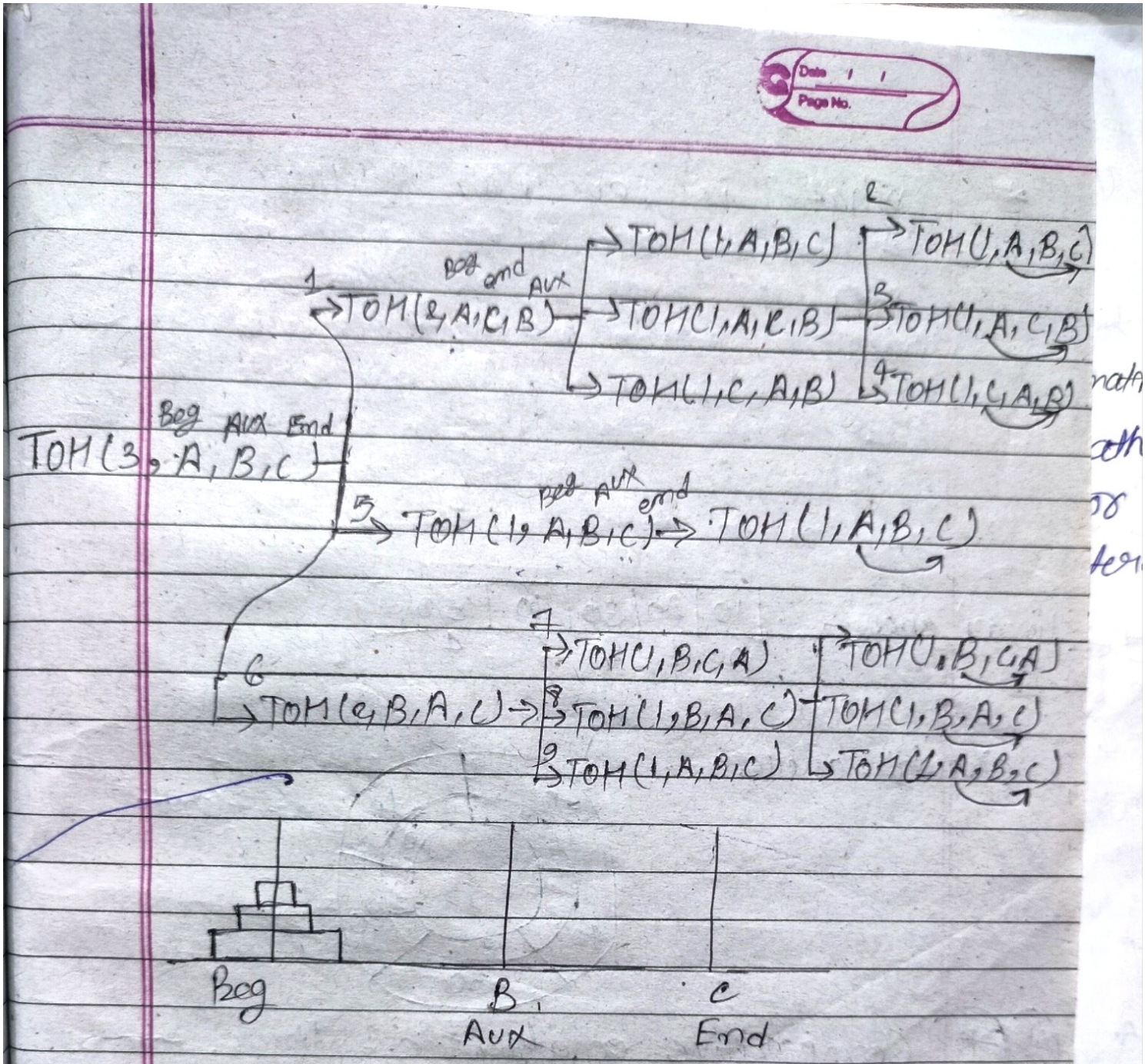
Soln →

```
TOH(N, Beg, AUX, end)
{   if (n==1)
    { Beg → End;
      return;
    }
  else { TOH(n-1, Beg, End, AUX)
         TOH( 1, Beg, AUX, End)
         TOH(n-1, AUX, Beg, End)
}
```





disk 1 moved from A → C
disk 2 moved from A → B
disk 1 moved from C → B
" 3 " " A → C
" 1 " " B → A
" 2 " " B → C
" 1 " " A → C



* What is TOH ~~graph~~.

ans⇒ Tower of Hanoi is a classic puzzle or mathematical problem that involves moving a stack of disks from one rod to another while adhering to certain rules.

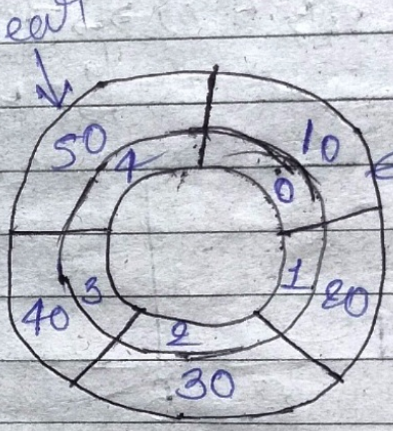
* What is circular queue?

ans ⇒ A circular queue, also known as a circular buffer or ring buffer that serves as a linear collection of elements with a fixed size unlike a regular queue, it is a linear data structure in which the last position is connected back to the first position to make a circle.

linear queue ⇒

10	20	30	40	50
0	1	2	3	4

circular queue.



que * Comperitive analysis of linear queue and circular queue

ans ⇒

f	0	1	2	3	4	r

initialization - int arr[5];
int rear = -1;
int front = -1;
int size = 5-1;

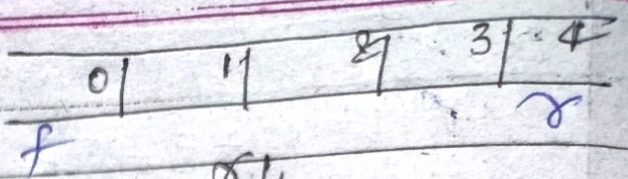
is empty()
{ if (front == -1 && rear == -1)
{ return true
}
else { return false
}

```
isfull()
{ if (rear == size)
  { return true;
  }
else {
      return false;
     }
```

```
enque(value)
{ if(isfull())
  { return; }
else if(isempty())
{ rear=front=0;
  arr[rear]=value;
}
else { rear++;
  arr[rear]=value;
  }
```

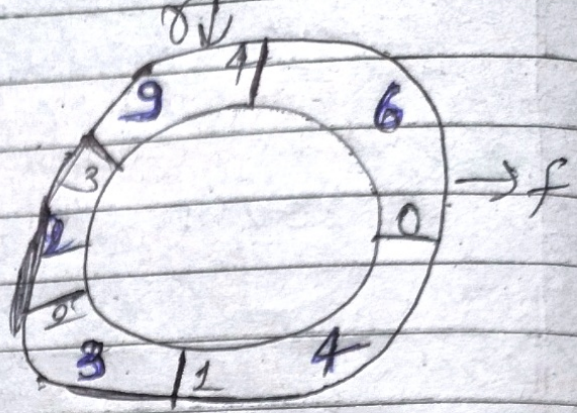
```
dequeue()
{ int x=0;
 if(isempty())
 { return; }
 else if (front == rear)
 { x = arr[front]
 front = rear = -1;
 }
 else
 { x = arr[front]
  front++;
 }return x;
  }
```

Initialization -




```
int size=5; int arr[5];
N=size;     int rear=-1;
            int front=-1;
```

```
isEmpty()
{ if(front==-1 && rear==-1)
      return true
  else
      return false
}
```




```
isFull()
{ if((rear+1)%N==front)
      return true;
  else
      return false;
}
```

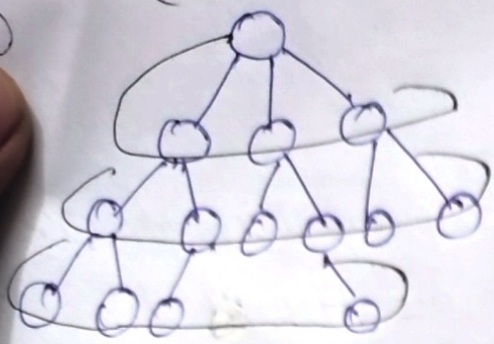
```
enque(value
{ if(isFull())
      return;
  else if(isEmpty())
  { rear=front=0;
    arr[rear]=value;
  }
  else { rear=(rear+1)%N;
         arr[rear]=value
  }
```

```
deque()
{ int x=0;
  if(isEmpty())
      return;
  else if(front==rear)
  { x=arr[front];
    front=rear=-1;
  }else { x=arr[front];
          front++;
  }return x;
}
```

(b) Differentiate between the graph traversal technique of BFS (Breath first search) & DFS (Depth first search)

(BFS)

(i)



(ii) Need queue data structure.

(iii) Need more space.

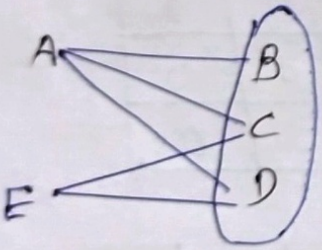
(iv) It focus all the neighbours of a ~~node~~ at a time

(v) Time complexity is O(V+E)

(vi) Applications: Shortest path finding problem

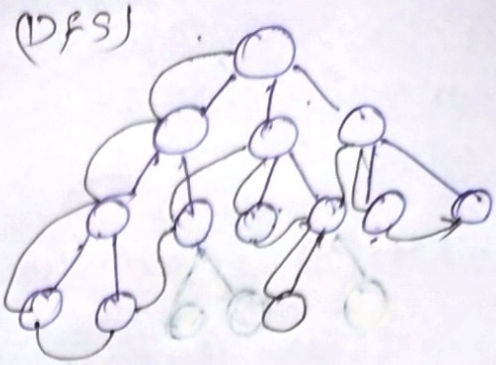
(b) To check wheather the graph is Bipartide or not.

(vii)



(DFS)

(i)



(ii) Need stack data structure.

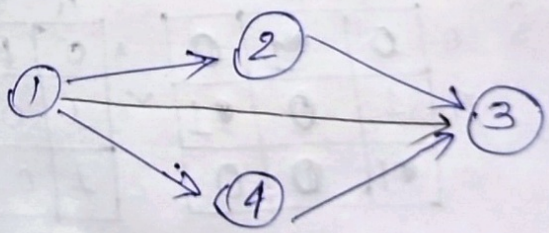
(iii) Need less space than BFS.

(v) We focus on childs of one Node.

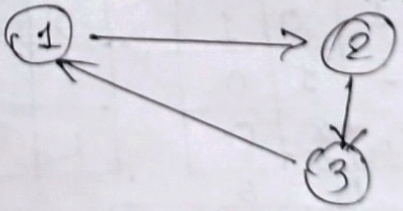
(v) Time complexity is O(V+E)

(vi) DFS is used ~~[redacted]~~ topological sorting process, scheduling, cycle detection.

(vii)



1→2→4→3→ [1, 2, 4, 3]



Q2. Compare the techniques and complexities of various sorting algorithms. → buble sort shell sort.
→ insertion sort
→ Selection sort
→ quick sort
→ merge sort

What are sparse matrices? Discuss the type of sparse matrices. Mention the efficient method to store sparse matrices in a computer system.

→ A sparse matrix is a two-dimensional data object made of m rows and n columns, therefore having total m×n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Ex →

$$\begin{bmatrix} 0 & 0 & 3 & 4 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \end{bmatrix}_{m \times n}$$

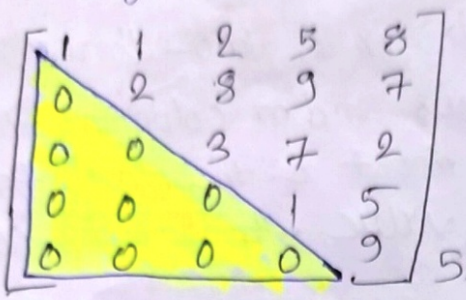
Types of sparse matrices

1. Lower triangular sparse matrix
2. Upper triangular sparse matrix
3. Tri-diagonal matrix

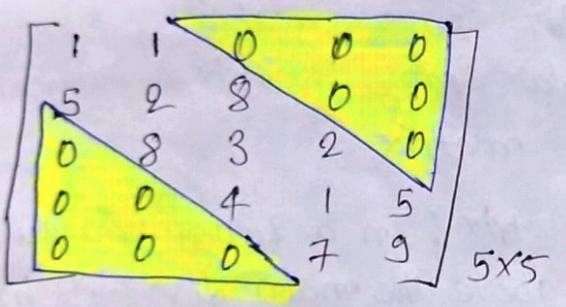
1. Lower Triangular Matrix: In a lower triangular sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also known as a lower triangular matrix. If you see its pictorial representation, then you find that all the elements having non-zero value are appear below the diagonal.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 & 0 \\ 1 & 4 & 3 & 0 & 0 \\ 9 & 8 & 7 & 1 & 0 \\ 1 & 2 & 7 & 8 & 9 \end{bmatrix}_{5 \times 5}$$

② Upper Triangular Matrix: In the upper triangular sparse matrix, all elements below the main diagonal have zero value. This type of sparse matrix is also known as an upper triangular matrix.



③ Tri-diagonal matrix: Tri-diagonal matrix is also another type of a sparse matrix, where elements with a non-zero value appear only on the diagonal or immediately below or above the diagonal.



The most efficient method to store sparse matrices in a computer system is to use a compressed storage formate. Compressed storage formats store only the non-zero elements of the matrix, along with additional information to allow for efficient access to these elements.

In CSR formate the matrix is stored as three arrays:

- Values Array: Store the non-zero values of the matrix row-wise.
- Column Indices Array: Store the column indices corresponding to each non-zero value in the same order as the values array.

Row Pointers Array: Store the index positions in the values and column indices arrays where each row starts.

Example:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 3 & 4 & 0 & 0 \\ 5 & 0 & 0 & 6 & 0 \\ 0 & 0 & 7 & 0 & 8 \end{bmatrix}$$

- Values array (A): [1, 2, 3, 4, 5, 6, 7, 8]
- Column indices array: [0, 4, 1, 2, 0, 3, 2, 4]
- Row pointers array: ~~[0, 2, 4, 7, 8]~~ [0 0 1 1 2 2 3 3]

Q2. Describe the implementation of priority queue.

ans ⇒ There are two way to implement priority queue.

(i) one-way list implementation

(ii) 2D-array implementation

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules.

(i) An element of higher priority is processed before any element of lower priority.

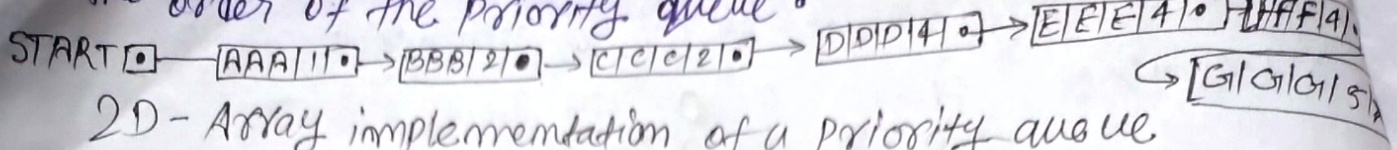
(ii) Two elements with the same priority are processed according to the order in which they were added to the queue.

One-way List representation of a priority queue

(a) Each node in the list will contain three items of information : an information field INFO, a priority number PRN and a link number LINK.

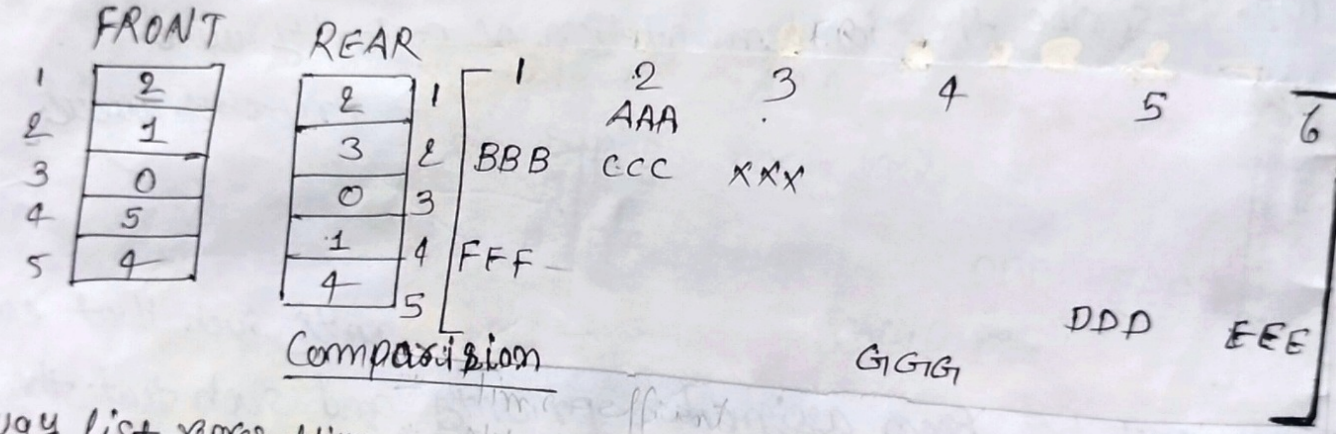
(b) A node x precedes a node y in the list (1) when x has higher priority than y or (2) when both have the same priority but x was added to the list before y.

This means that the order in the one-way list corresponds not to the order of the priority queue.



2D-Array implementation of a priority queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority. Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space a two dimensional array QUEUE can be used instead of the linear arrays.



One way list representation

(i) less time efficient

(ii) more space-efficient than array

(iii) overflow occurs only when the total number of elements exceeds the total capacity.

array presentation of priority queue

(i) more time efficient

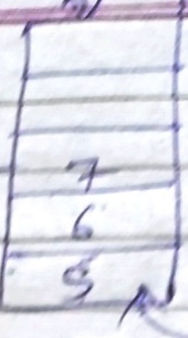
(ii) Less space-efficient than one way

(iii) overflow occurs when the number of elements in any single priority level exceeds the capacity for that level.

Stack algorithm:

```
is_empty()
{ if(top==-1)
{ return true;
} else
{
return false;
}}
```

deletion




```
arr[5];
top=-1
size=4;
insertion
```

```
is_full()
{ if(top==size)
{ return true;
}
else {
return false;
}}
```

```
Push(int value)
{ if(is_full())
{ cout<<" overload";
}
else { top++;
arr[top]=value;
}
}
```

```
pop()
{ if(is_empty())
{ cout<<" underflow";
}
else { int popv=arr[top];
arr[top]=0;
top--;
return popv;
}
}
```

Q why is circular queue better than linear queue?

ans ⇒ (i) Efficient use of memory: In a circular queue, when the rear pointer reaches the end of the queue. It wraps around to the beginning, which allows for efficient use of memory compared to a linear queue.

(ii) Easier for insertion-deletion: In the circular queue, if the queue is not fully occupied then the elements can be inserted easily in the vacant locations. but in linear queue insertion not possible. once the rear reaches the last index.

(iii) Better Performance: Circular queue offers better performance in situations where data is frequently added and removed from the queue as compared to a linear queue.

(iv) Reduced overflow risk: Circular queues are less likely to overflow than ~~linear~~ linear queues. This is because circular queues can use all of the available space. even if the queues is not full.

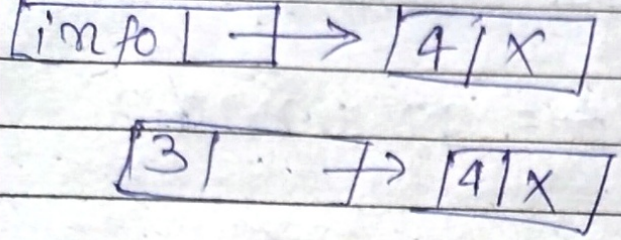
Comparison of sorting ÷ Time complexity

Sorting Techniques	Best case	Average case	Worst case
1. Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
2. Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
3. Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
4. Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
5. Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
6. Shell sort	$O(n)$	$O(n(\log n)^2)$	$O(n(\log n)^2)$
7. Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Sorting techniques	space complixity	stability	Auxiliary space	Locality of reference
1. Bubble sort	$O(1)$	yes	Not required	poor
2. Insertion sort	$O(1)$	yes	Not required	Good
3. Selection sort	$O(1)$	No	Not required	fair
4. Quick sort	$O(\log n)$	No	Not required	Good
5. Merge sort	$O(n)$	yes	Required	Poor
6. Heap sort	$O(1)$	NO	Not required	fair
7. Shell sort	$O(1)$	NO	Not required	Good.

To create a linear linked list

Node



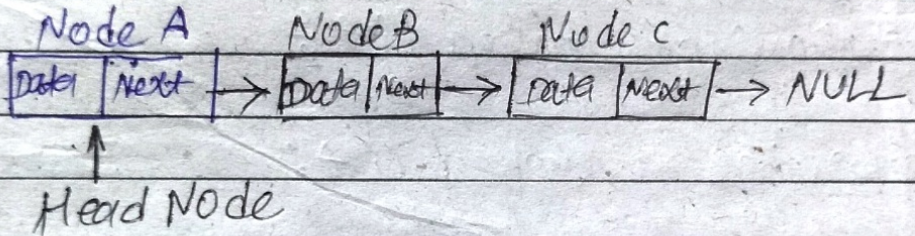

```
#include<iostream>
#include<stdio.h>
using namespace std;
struct Node
{ int data;
  Node *next;
};
Node *head = NULL;


void insertinLL(int val)
{ Node *newnode = (Node *)malloc(sizeof(Node));
newnod -> data = val;
newnode -> next = NULL;
Node *last = head;
if (last == NULL)
{
  head = newnode;
    return;
}
```

```
else { while ( last -> next != NULL )
    { last = last -> Next;
    }
    last -> Next = newnode;
      return;
    }
}
struct node
{ int data;
  Node * next;
};
 Node * head = NULL;
void insertionLL(int val)
{ Node * newnode = (Node*) malloc.
```

* What is Linked List: A Linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers (entity that point to the next element)

other words

A linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.



Linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

Linked List vs Arrays

Advantages of Linked List over arrays:

(i) Dynamic size

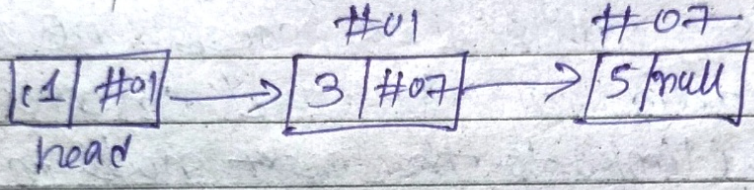
(ii) Ease of insertion / deletion

Disadvantages of Linked List over Array:

(i) Random access is not allowed. We have to access elements sequentially starting from the first node.

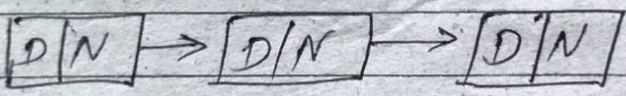
(2) Extra memory space for a pointer is required with each element of the list.

(3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.



Operations of linked list :-

1. Traversing a linked list:



2. Append a New node (to the end) of a list.

3. Prepend a new node (to start)

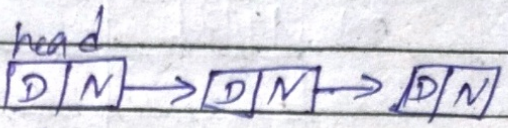
(4) Inserting a new node to a specific position in the list.

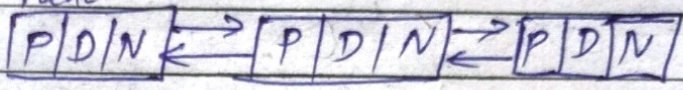
(5) Deleting a node from the list.

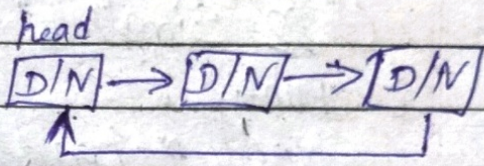
(6) updating a node in the list.

Types of linked list

D = Data ⇒ P = Previous N = Next

1. Singly linked list → 

2. Doubly linked list → 

3. Circular linked list → 

Some application of linked list

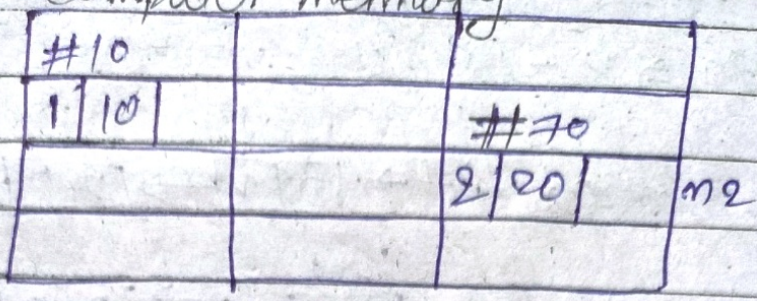
- Linked lists can be used to implement stack, queue,
- Linked lists can also be used to implement Graphs,
- Implementing Hash Tables
- Undo functionality in photoshop or word

Creating a node

```
class Node {
public: int key;
        int data;
        Node * next;
        Node()
    { key = 0;
      data = 0;
      next = NULL;
    }
```

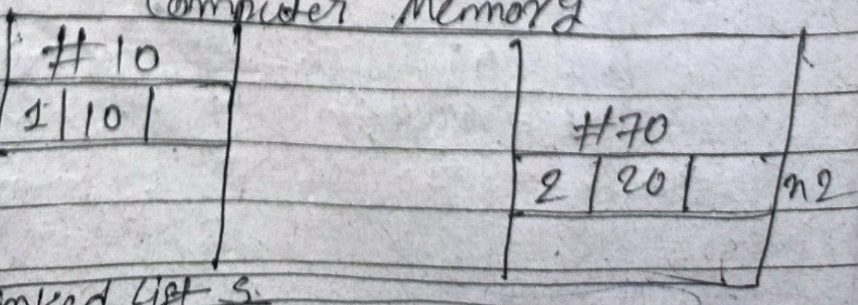
```
Node (int k, int d)
{ key = k;
  data = d;
  next = NULL;
}
};
int main()
{ Node n1(1,10);
  Node n2(2,20);
}
```

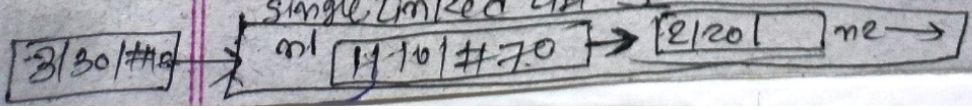
Computer Memory




```
class SinglyLinkedList
{ public:
    Node *head;
SinglyLinkedList(Node *h)
void PrependNode(Node *h)
void appendNode(key)
void insertNode(key)
void deleteNode(key)
void updateNode(key)
}
    int main()
  { Node n1(1,10);
    Node n2(2,20);
    Node n3(3,30);
  SinglyLinkedList S(&n1);
  S.appendNote(&n2);
  S.PrependNote(&n3);
}
```

Computer Memory





```
#include<iostream>
#include<stdlib.h>
using namespace std;
struct Node
{
int data;
Node *next;
};
Node *head=NULL;
void insertinLL(int val)
{
Node *newnode=(Node *)malloc(sizeof(Node));
newnode->data=val;
newnode->next=NULL;
Node *last=head;
if(last==NULL)
{ head=newnode;
return;
}
else
{ while(last->next!=NULL)
{
last=last->next;
}
last->next=newnode;
return;
}
}
```

```
void display()
{ Node *ptr = head;
  if (ptr == NULL)
{ cout<<"No elements are there in
  Linked List";
}
  else
{ cout<<"Elements in linked list are:\n";
  while (ptr != NULL)
  {
   cout<<ptr->data;
   ptr = ptr->next;
  }
}
}
void deletedata(int todel)
{
 Node *temp = head;
 Node *prev;
while(temp != NULL && temp->data != todel)
 {
   prev = temp;
   temp = temp->next;
 }
  if(temp == NULL)
{
cout<<"\n"<<todel<<" not found\n";
 return;
}
```

```
else {
cout << "\n deleting " << todel << " from the linked list \n";
prev -> next = temp -> next;
free(temp);
}
int main()
{ cout << " inserting 3 in the linked list \n";
insertinLL(3);
cout << " Inserting 4 in the linked list \n";
insertinLL(4);
cout << " Inserting 55 in the linked list \n";
insertinLL(55);
cout << " Inserting 184 in the linked list \n";
insertinLL(184);
cout << " Inserting -120 in the linked list \n";
insertinLL(-20);
cout << "\n currently the elements present in the
linked list are: \n";
display();
cout << "..... \n";
cout << " delete 30";
deletedata(30);
cout << " delete 55";
delete.data(55);
cout << " delete -20";
deletedata(-20);
cout << " delete 7";
deletedata(7);
```

```
cout<<"\n.....\n";
cout<<"\n Now, the elements present in the
    linked list are: \n";
    display();
    return 0;
}
```

output

Inserting 3 in the linked list
" 4 " " " "
" 55 " " "
" 184 " "
" -20 " "

currently the elements the elements present in the linked list are:
elements in linked list are;
3
4
55
184
-20

.....

delete 30
30 not found
delete 55
deleting 55 from the linked list
deleting -20
deleting -20 from the linked list
delete 7
7 not found

.....

Now, the elements present in the linked list are:

Elements in linked list are:

3

4

189

Q Demonstrate with example an efficient method of storing a tridiagonal matrix in computer memory.

ans ⇒

$$A = \begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ & a_{32} & a_{33} & & \\ & & \ddots & \ddots & \\ & & & a_{n1} & a_{nn} \end{pmatrix}$$

$B[1] = a_{11}, B[2] = a_{21}, B[3] = a_{22}, B[3] = a_{31}, \ldots$

observe first B.

$1+2+3+4+\ldots+n = \boxed{\frac{n(n-1)}{2}}$

$\boxed{B[L] = a_{jk}}$

L represent the no of elements in the list upto the a_{jk}.

$1+2+3+\ldots+(j-1) = \frac{j(j-1)}{2}$

element in the row i above a_{jk} and k are elements in row j.

$$\boxed{L = \frac{j(j-1)}{2} + k}$$

* Analyze and compare the mechanism to implement priority queue.

ans ⇒

Compare	One way list implem.	2D array implementation
①	less time efficient	more time efficient
②	more space-efficient than array	less space efficient the one way.
③	overflow occurs only when the total number of elements exceeds the total capacity.	overflow occurs when the total number of element in any single priority level exceeds the capacity for that level.

one way → START → [A | 1 | •] → [B | 2 | •] → [C | 2 | •]

→ [D | 4 | •] → [E | 4 | •] → [F | 4 | •] → [G | 5 | X]

array →

	front	Rear
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

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

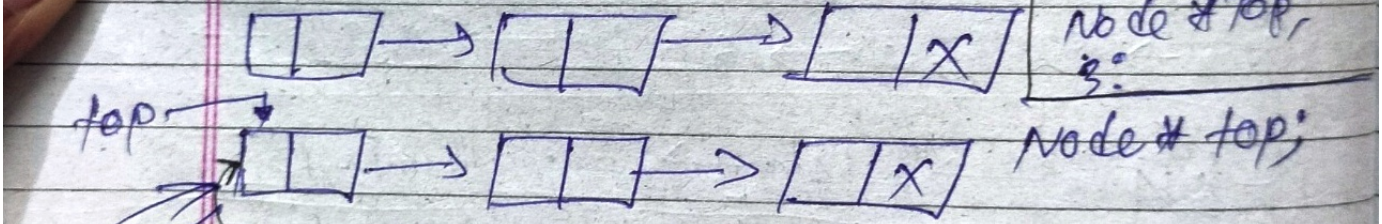
MST - 2

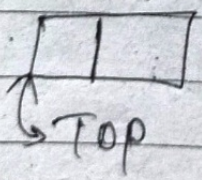
* Sequential representation and linked representation.
  - stack
  - queue

Implement stack and queue using:
- (i) sequential representation (array) → stack, queue
- (ii) linked representation → stack, queue

* linked representation of stack

```
structure node
{ int data;
node *top;
};
node *top;
```



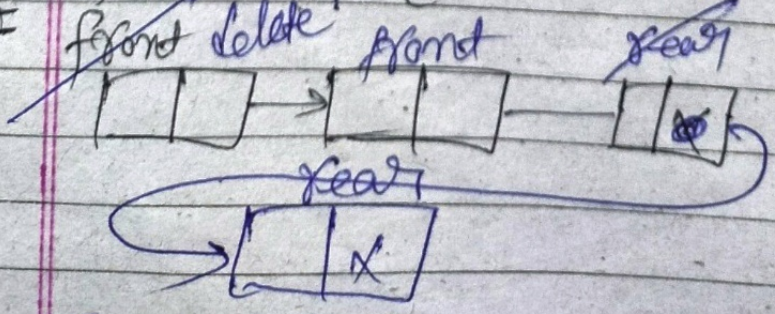


for delete → ;; Top → next ;

Top

for queue:

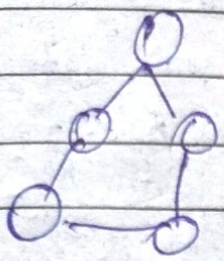
```
struct node
{ int data;
node *front;
node *rear;
};
```



for insertion

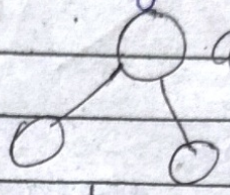
Trees (non-linear data structure)

Hierarchical relationship

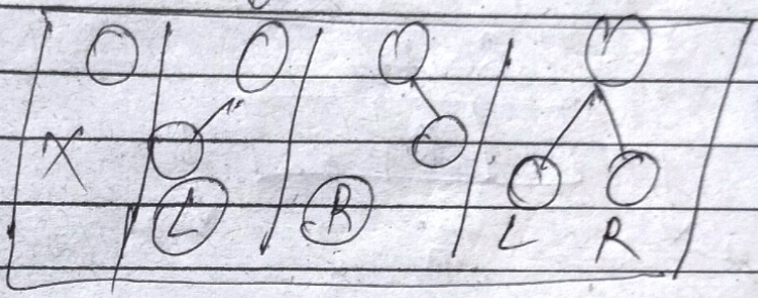


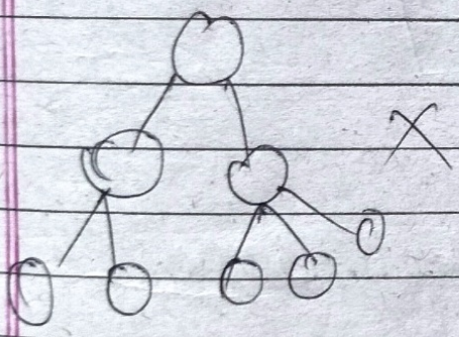
level - 0
level - 1
level - 2

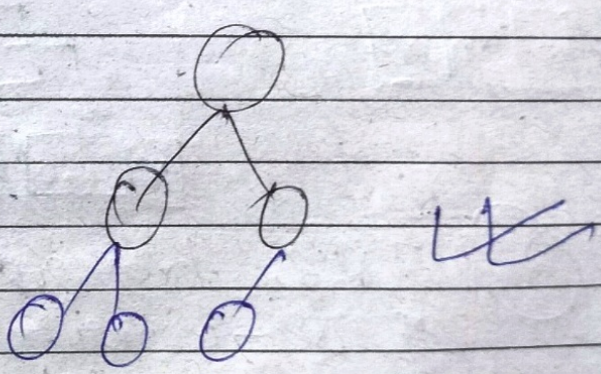
Binary tree



at the most 2 children







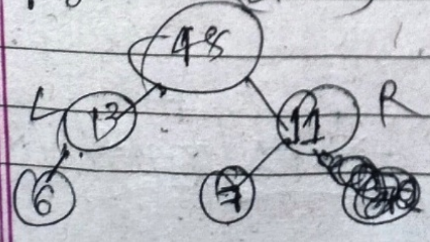
Representation of Trees (Array)

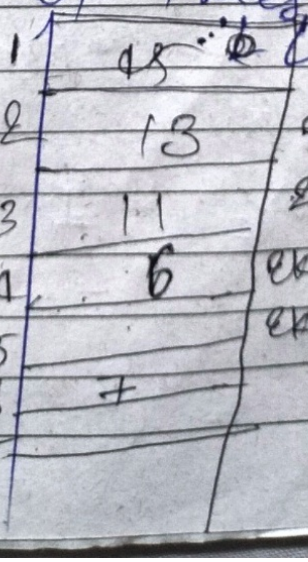
Array

Node → k^{th}

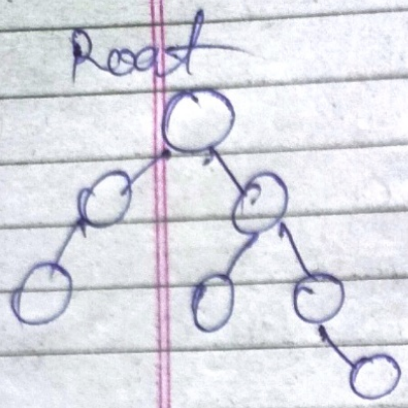
left = $2k^{th}$

right = $(2k+1)^{th}$

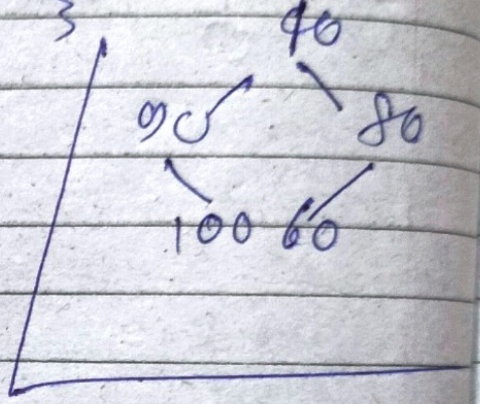


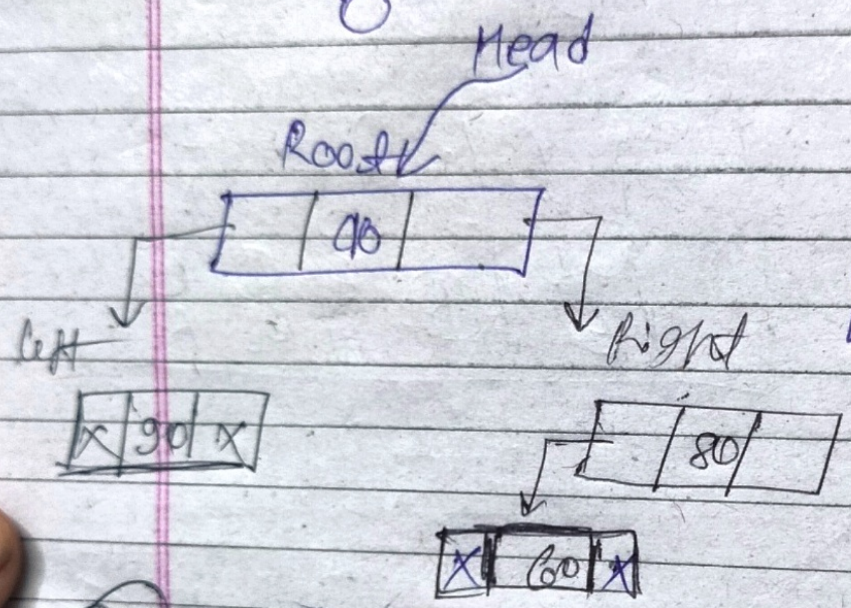


* Representation of Trees using linked




```
struct Node
{ int data;
  Node * left;
  Node * right;
}
```

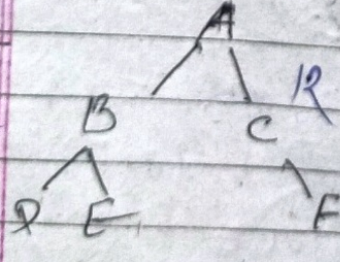




(*) Tree traversal technique.

* pre-order
* In-order
* Post-order

B.T



* pre-order → Root → left → Right

* In-order → left → Root → Right

Recursive in nature.

* Post-order → left → right → Root

2 Mole → Binary tree Apply this pre-In, post order and traverse

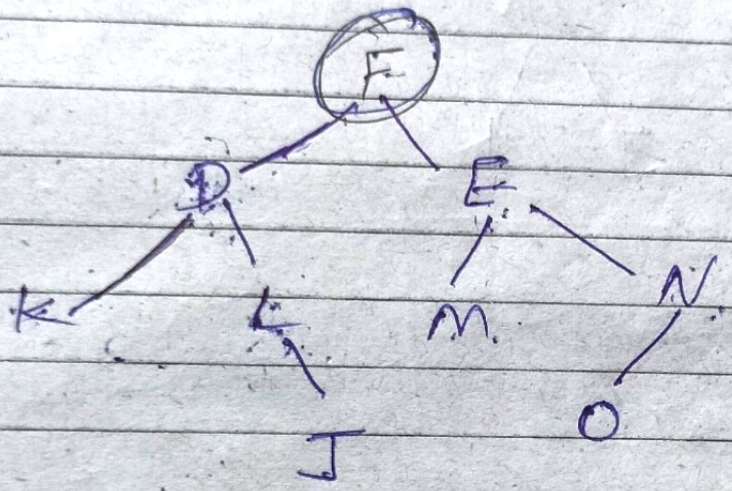
4

pre order: A | BDE | CF ≈ ABDECF

In-order: BDBE | A | CF ≈ DBEACF

post order: DEB | FC | A ≈ DEBFCA

9



pre: F | DKLJ | EMNO ≈ FDKLJEMNO

Inorder: KDLJ | F | MEON ≈ KDLJFMEON

Post: Left | Right | Root

KJLD | MONE | F

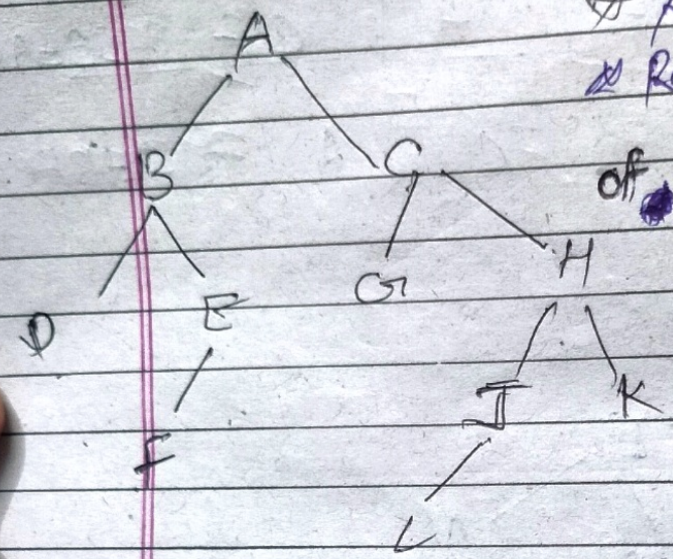
≈ KJLDMONEF

eans that the nodes ... by default in order

5

Threaded Binary tree

* Avoid using NULL pointer
* Rather than using Null ptr. store the address of the next node in the traversal.



Threaded

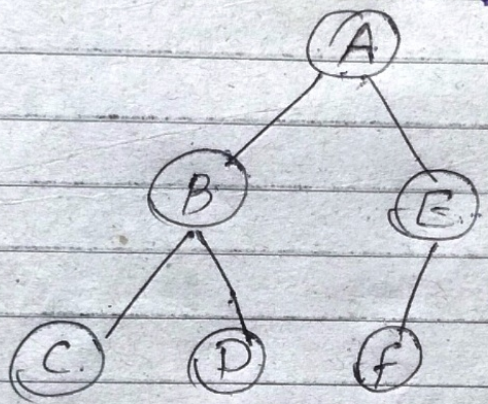
- one way threaded (only previous store) (or address)
- two way threaded (previous & next both address store)

eM

(yes) Apply two-way threading with header node in a tree.

one> Left pointer of the first node and right pointer of last node will contain NULL value.

→ two way threading stores previous as well as next address. It Avoid using Null ptr Rather than using Null pointer Store the address of the next Node in the traversal.



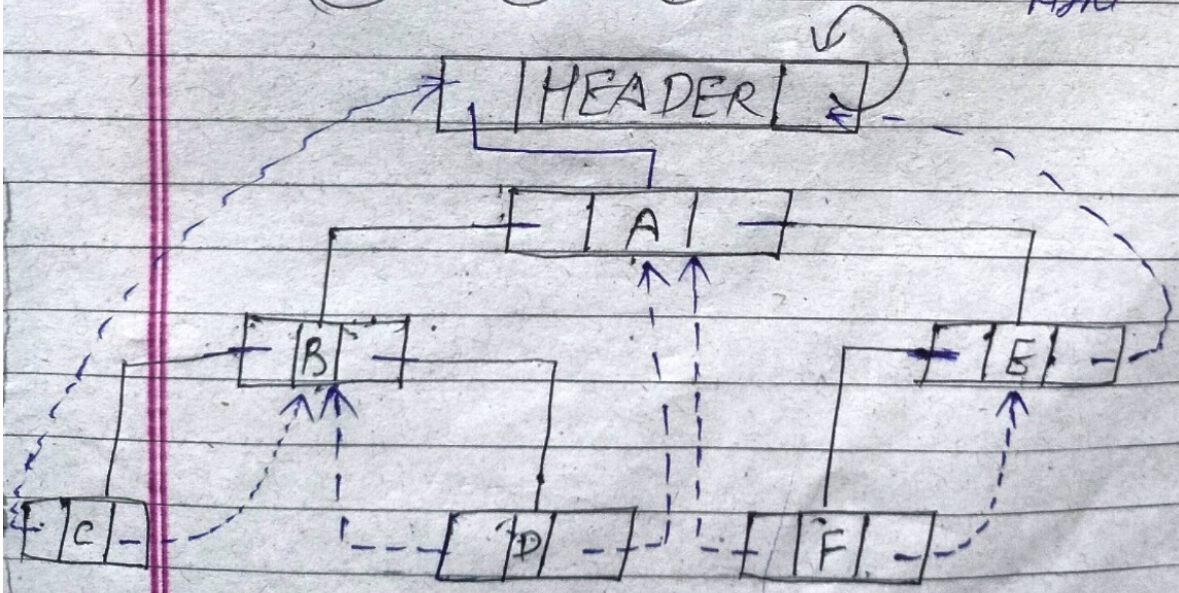
inorder

Left	Root	Right
CBD	A	FE

≈ CBDAFE

left

Right

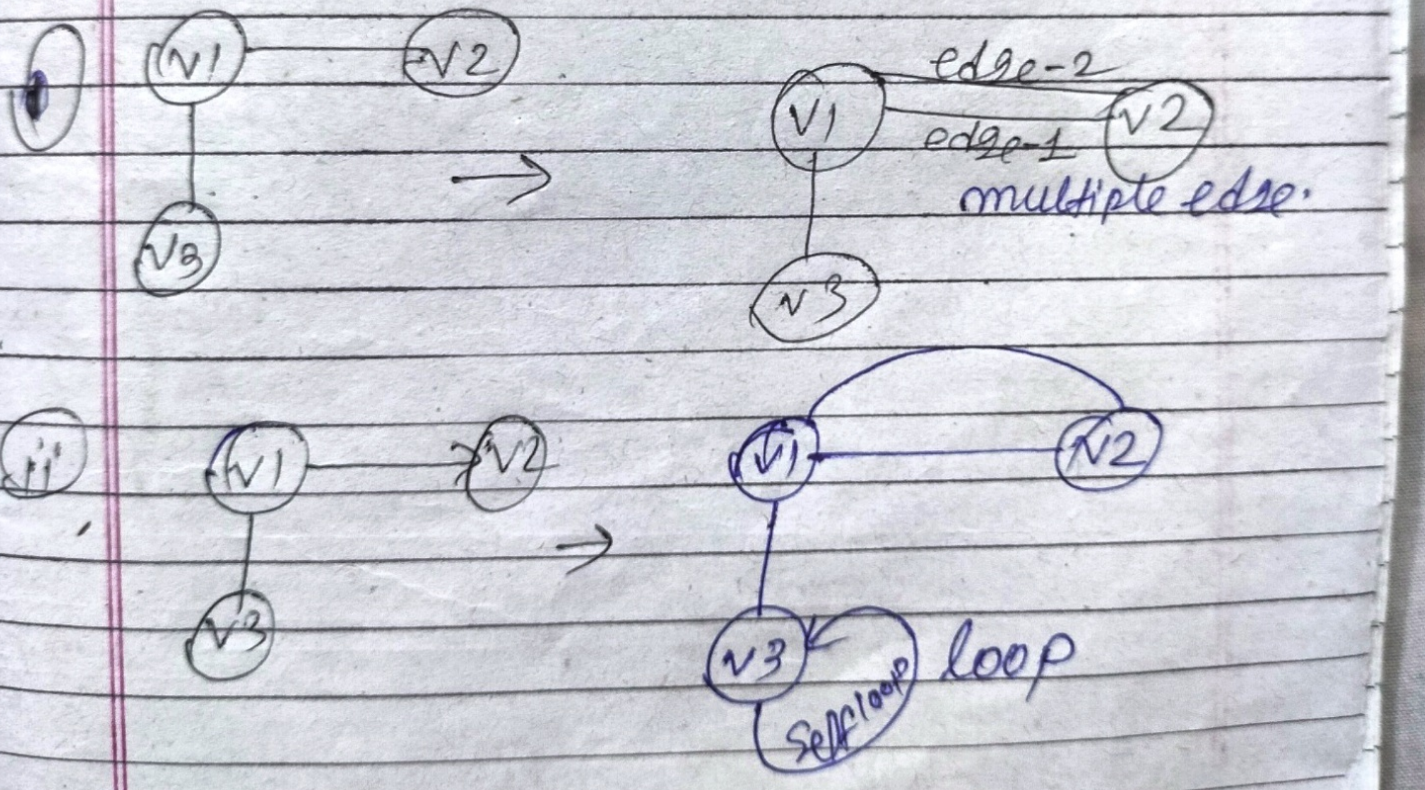


Q9) Analyze the features that make a graph as a multigraph.

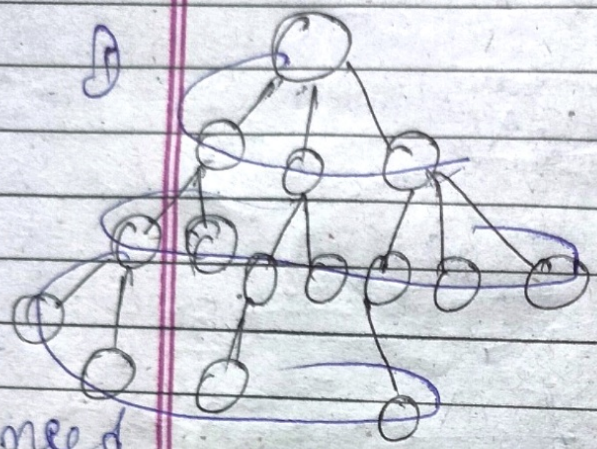
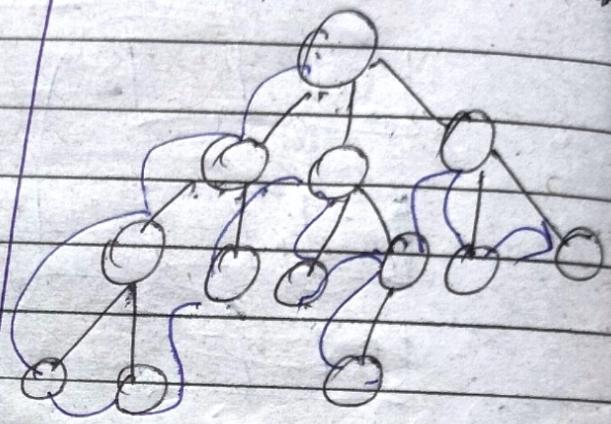
Ans ⇒ To make a graph as a multigraph

(i) graph should have multiple edges between two nodes.

(ii) graph can also have self loops.



Graph Traversal Techniques:

BFS	DFS
Breadth first search	Depth first search
	
① need queue data s to implement BFS	stack data s to implement DFS
② Needs more space	Needs less space than BFS
③ all the neighbours are saved at a time	③ we focus on childs of one Node.
④ Time complexity is $O(V+E)$.	④ Time complexity is $O(V+E)$.
⑤ a. Application: shortest path finding problam b. To check whether the graph is Bipartite or not	DFS is used for topological sorting scheduling, cycle detection.

* Hasing (saving the memory)

* Hasing techniques [Hash functions]
→ H(k) : K → L

Hashing
1. Division method
2. Mid-square method
3. folding method

k = 3205
h = k mod m
h = 3205 mod

To represent 4 digit no into two digit no mod h with m
m → prime no.
m → closest to maximum value and greater the record of the member

>m: 9699 (97)
>record
>no of employee
>65

prime no that value

97) 3205 (04
X

(ii) Mid-Square method: $k = 3205 \rightarrow k^2$

$k^2 = 10272025$

(72)

(iii) Folding method: 32 | 05 = 32 + 05 = 37 without reversing

(ii) reversing 3205

32 + 50 = (82) with reversing

R→ (i) 7148 = 71 + 48 = 119 = 19

(ii) 7148 = 71 + 84 = 155 = 55

R→ 3232 = 32 + 23 = 55

~~Collision in Hashing:~~

* Collision in Hashing:

$H(k) = h$

$H(k') = h$

H(k) → folding

H(7148) → 55

H(3232) → 55

Coll resolution tech: (i) linear probing

(ii) quadratic Probing

(iii) Double Hashing

practice formula

(1) Linear probing: $h, (h+1)\%N, (h+2)\%N$

$h, h+1, h+2, h+3, \ldots$

(2) Quadratic probing:

$h, h+1^2, h+2^2, h+3^2$ | $h, (h+1^2)\%N, (h+2^2)\%N$

$h, h+1, h+4, h+9$

(3) $H(k) = h$

$H'(k) = h'$

$h, h+h', h+2h', h+3h' + \ldots$

$h, (h+h')\%N, (h+2h')\%N$

(5) linear probing $n=8$

	A	B	C	D	E	X	Y	Z
h:	4	8	2	11	4	11	5	1

Mem loc: $(q+1)\%11$

1	2	3	4	5	6	7	8	9	10	11
X	C	Z	A	E	Y		B			D

Open addressing (closed Hashing)
[To resolve collision: involve probing]

Efficiency:

	A	B	C	D	E	X	Y	Z
H(k)	4	8	2	11	4	11	5	1
	X	C	Z	A	E	Y	B	D
M loc.	1	2	3	4	5	6	7	8	9	10	11

maximum = 11

Step: 1 $n = 8$, $m_{loc} = 11$

load factor $= \frac{n}{m} = \frac{8}{11} \Rightarrow \alpha = 0.73$

Step: 2 Compute $S(\alpha)$ & $U(\alpha)$

$S(\alpha)$ → It is the average no of probes for successful search

$U(\alpha)$ → It is the average no of probes for unsuccessful search.

Reminder

$S(\alpha) = ?$

$U(\alpha) = ?$

$S(\alpha) = \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)}\right] = 2.35$ | $S(\alpha) = \frac{1}{2}[1 + \frac{1}{}$

$U(\alpha) = \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)^2}\right] = 7.36$

$S = \frac{2.35}{n}$ ... $U = \frac{7.36}{m}$

$S = 0.29$ ... $U = 0.669$

try that the ...

A	B	C	D	E	X	Y	Z
1	1	1	1	2	2	2	3

$S = \frac{1+1+1+1+2+2+2+3}{8(m)}$

$S = \frac{13}{8}$ $\quad S = 1.625 \quad \approx 1.63$

$K \to \frac{1}{2}$ के लिए सबसे खोजना है $\quad 1.63 < 2.35$

$S < S(d)$

	1	2	3	4	5	6	7	8	9	10	11
$U =$	7	6	5	4	3	2	1	0	1	1	8

$11(m)$

$U = \frac{40}{11}$ $\quad U = 3.63$

$U < U(d)$

$3.63 < 7.36$

(*) Open channing (hashing)

Records	A	B	C	D	E	X	Y	Z
	4	8	2	11	4	11	5	1

$m = 8$, $m = 11$

	Info	Link
1	A	0
2	B	0
3	C	0
4	D	0
5	E	1
6	X	4
7	Y	0
8	Z	0
9		0
10		0
11		

Dimk

1	8
2	3
3	0
4	5
5	7
6	0
7	0
8	2
9	0
10	0
11	6

m =

faster method than above having but not space efficient because it using memory to save the pointer.

efficiency

$S(d) = 1 + \frac{1}{2}d$ $\quad U(d) = e^{-d} + 1$, $d = \frac{n}{m}$

$S_1 = ?$ $\quad U = ?$ $\quad e \rightarrow$ Euler's no. $= 2.718$

$d = \frac{8}{11}$ $\quad d = 0.73$

$S(d) = 1 + \frac{1}{2} \times 0.73$, $U(d) = e^{-d} + d$

$S(d) = 1.36$

$S(d) = 1.37$

$U(d) = 0.48 + 0.73 = 1.21$

eans that the ...

A B C D F X Y Z

S= 2 + 1 + 1 + 2 + 1 + 1 + 1 + 1

m = 8

$S = \frac{10}{8}$ | S = 1.25 | S(d) S

1.37 > 1.25

efficient

U= 1 2 3 4 5 6 7 8 9 10 11

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

m = 11

$U = \frac{11}{11}$ U = 1

U(d) U

1.21 > 1

efficient

Analysis of load factor:

Open Addressing

closed hashing

[Hash Table]

[m ≥ n] 11 = m

$d = \frac{n}{m} \leq 1$

chaining

open-hashing

n

m

[m < n]

$d = \frac{n}{m} > 1$

Rehashing:

④ Quick sort algorithm

step 1 (25), 57, 48, 37, 12, 92, 86, 33

set first element as pivot

Step 2 dn → dn = 0

A[dn] > pivot

UP = 7

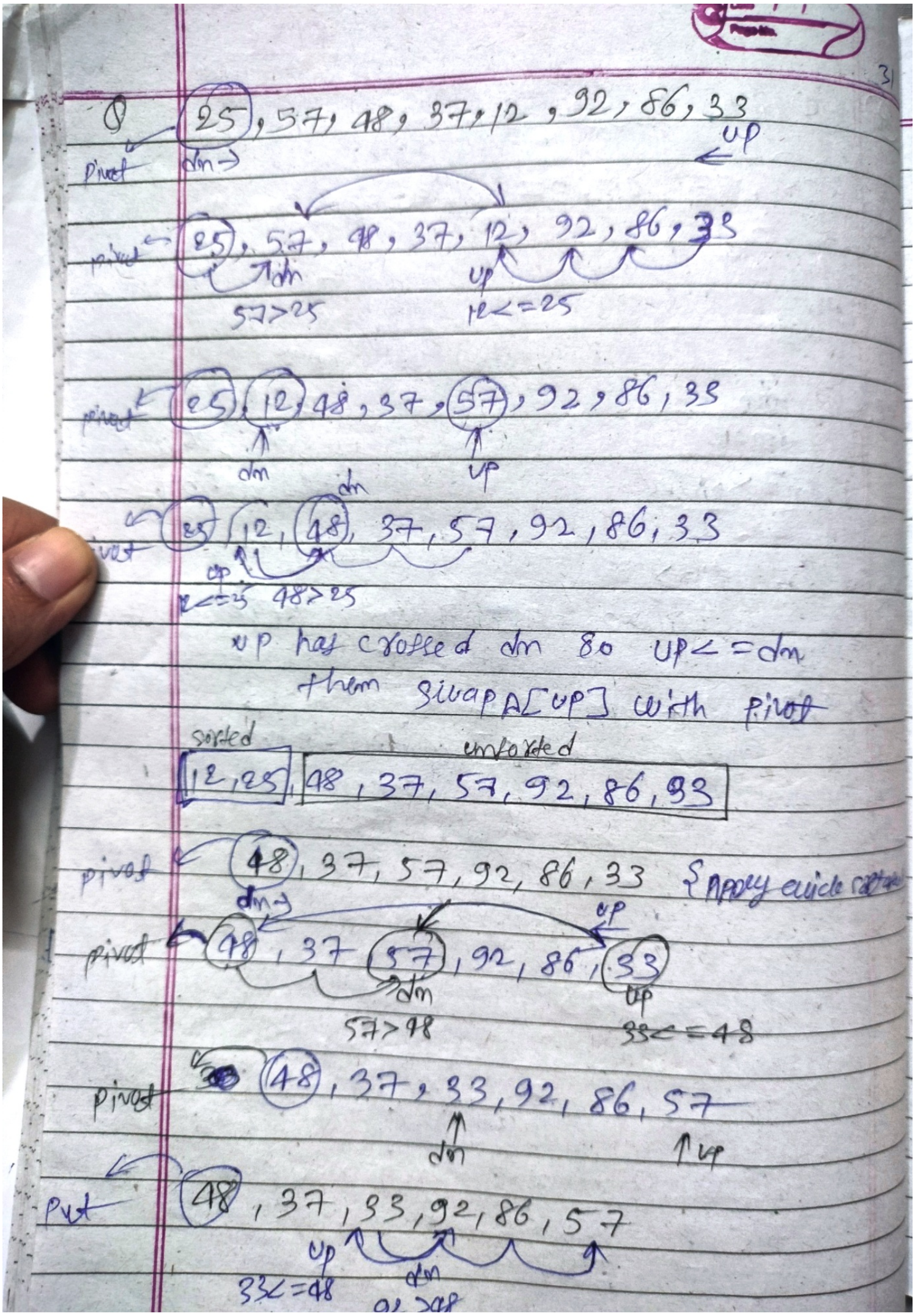
UP will decrement till A[UP] <= pivot

Step 3 if Up > dn

swap A[dn] with A[UP]

but if Up <= dn

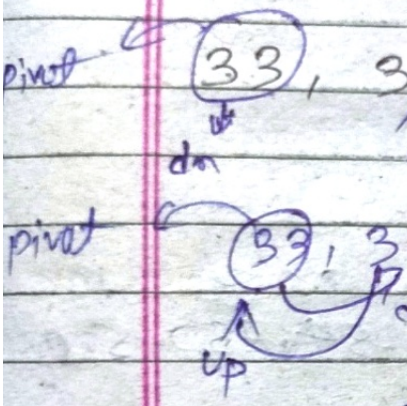
swap A[UP] with pivot



up has crossed dn so up<=dn then swap A[up] with pivot

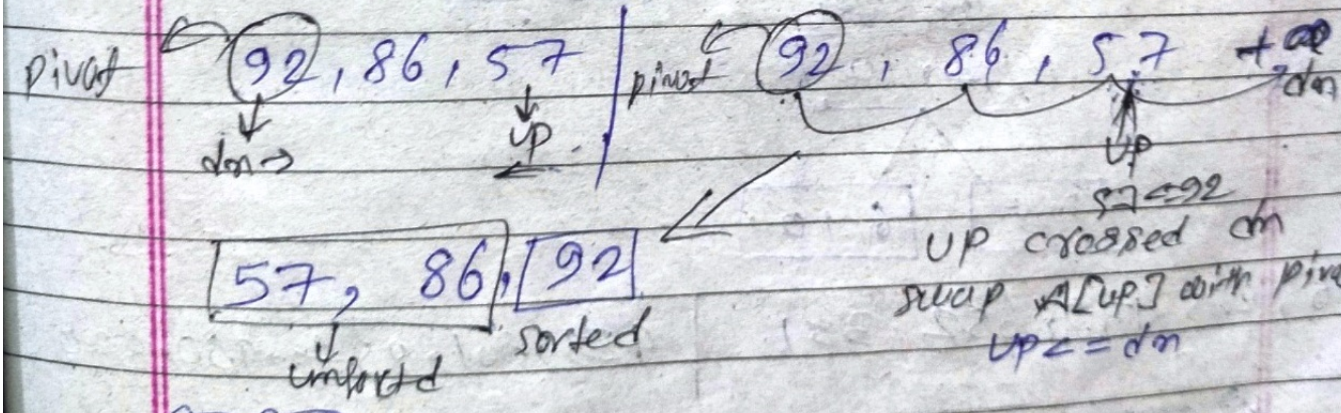
up has crossed dn
up <= dn so swap it with pivot
A[up] swap with pivot

[33, 37], 48, [92, 86, 57]
sorted unsorted



dn has crossed dn
up <= dn
then A[up] swap with pivot

[33, 37] 48 [92 86 57]
sorted unsorted

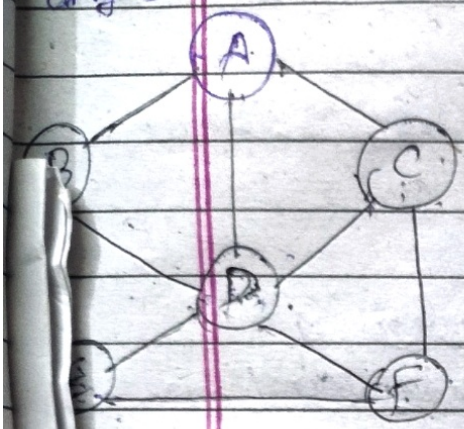


up crossed dn up <= dn
A[up] swap with pivot

[57, 86]
sorted

* Traverse the following graph using BFS starting from node B.

any ⇒



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	1	1	0
C	1	0	0	1	0	1
D	1	1	1	0	1	1
E	0	1	0	1	0	1
F	0	0	1	1	1	0

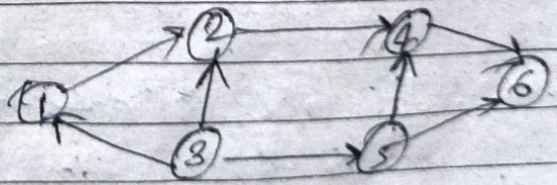
Q1	output Q2
B	ϕ
EDA	BE
DAF	BED
AFC	BEDA
FC	~~BEDAF~~
C	BEDAF
ϕ	BEDAFC

Pyq: Illustrate the use of adjancy matrix and list to represent a graph in memory

(i) Adjacency matrix

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	0	0	1	0	1
6	0	0	0	0	0	0

Pyq



(ii) Adjacency list

Node	Adjacency list
1	2
2	4
3	1, 2, 5
4	6
5	4, 6
6	

The adjacency matrix uses more space but allows for constant time to check whether edge exists b/w two vertices.

The A.L is more space efficient but requires more time for check whether edge exists b/w vertices.

(Pyq) Specify the situation(s) when operating system performs the garbage collection mention the steps involved in performing collection.

ans ⇒ operating system perform garbage collection in the following situation.

- when a process terminates : when a process terminates, the operating system reclaims/collect all the memory that was allocated to that process.

- when a program requests more memory : if a program requests more memory than the operating system may perform garbage collection.

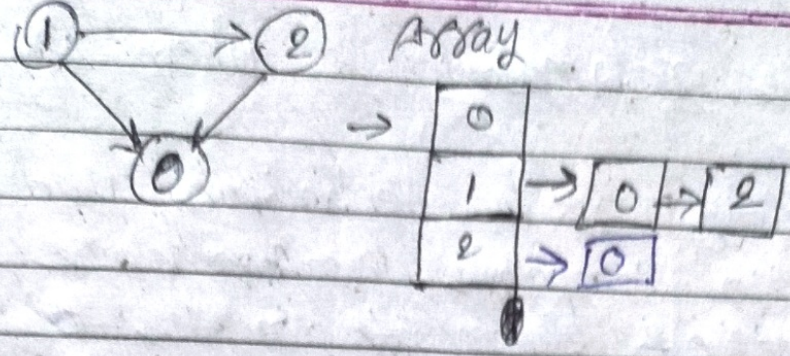
steps involve in performing collection

Step:1 → Gc will sequentially visit all nodes in memory and mark all nodes which are being used in program.

Step 2: It will collect all unmarked nodes and place them in free storage area.

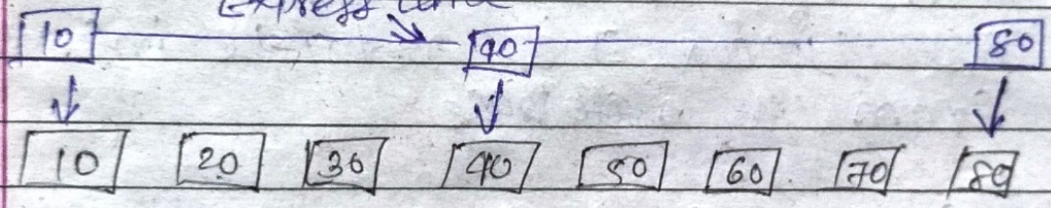
(Pyq) what is adjancency list?

ans⇒ An adjancemay list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.

Ex → 

Q9 What do you mean by skip list?

Ans A skip list is a data structure that allows for efficient search, insertion and deletion of elements in a sorted list. It skips over many of items in the full list in one step, that's why it is called skip list.



Q10 Why Balancing is important in Binary search Tree
- Faster search time
- More efficient for insertion & deletion.
- Reduced memory usage.
- Time complexity $O(\log_2 n)$

Q11 How rehashing is done?

Ans → Step 1: Create a new hash table with twice the no of buckets of the old table.

Step 2: Iterate over the elements in the old table and add them to the new table using their new hash values

Step 3: Discard the old table.

eam

Date / /
Page No.

* What is rehashing & when is rehashing

ans ⇒ Rehashing is a technique used in hash tables to improve performance as the number of elements in the table increases.

* When rehashing is done.
Rehashing is done when the load factor exceeds a certain threshold.

$$\text{load factor} = \frac{\text{no of element in the table}}{\text{no of buckets}}$$

* Need of hashing: hashing is need for index and retrieve information from a database.

(Pyq) Records: P Q R A B C V W X
H(K): 5 4 5 6 8 11 11 1 4

Suppose the records are entered into the table T in the above order.

(a) Examine the efficiency of the given hash function with linear probing as the collision resolution technique.

(b) formulate the memory organization if the records are stored using chaining.

eary "

Date / /
Page No.

* What is rehashing & when is rehashing

ary ⇒ Rehashing is a technique used in hash tables to improve performance as the number of elements in the table increases

* When rehashing is done.
Rehashing is done when the load factor exceeds a certain threshold.

$$\text{load factor} = \frac{\text{no of element in the table}}{\text{no of buckets}}$$

* Need of hashing: hasing is need for index and retrieve information from a database.

(Eg) Records: P Q R A B C V W X
H(K): 5 4 5 6 8 11 11 1 4

suppose the records are entered into the table T in the above order.

(a) Examine the efficiency of the given hash function with linear probing as the collision resolution technique.

(b) formulate the memory organization if the records are stored using chaining.

* What is rehashing & when is rehashing

ans ⇒ Rehashing is a technique used in hash tables to improve performance as the number of elements in the table increases.

* when rehashing is done.
Rehashing is done when the load factor exceeds a certain threshold.

$$\text{load factor} = \frac{\text{no of element in the table}}{\text{no of buckets}}$$

* Need of hashing: hasing is need for index and retrieve information from a database.

(Pyq) Records: P Q R A B C V W X
H(K): 5 4 5 6 8 11 11 1 4

suppose the records are entered into the table T in the above order.

(a) Examine the efficiency of the given hash function with linear probing as the collision resolution technique.

(b) formulate the memory organization if the records are stored using chaining.

* What is rehashing & when is rehashing

ans⇒ Rehashing is a technique used in hash tables to improve performance as the number of elements in the table increases

* When rehashing is done.

Rehashing is done when the load factor exceeds a certain threshold.

$$\text{load factor} = \frac{\text{no of element in the table}}{\text{no of buckets}}$$

* Need of hashing: hashing is need for index and retrieve information from a database.

Eg: Records: P Q R A B C V W X
H(K): 5 4 5 6 8 11 11 1 4

Suppose the records are entered into the table T in the above order.

(a) Examine the efficiency of the given hash function with linear probing as the collision resolution technique.

(b) formulate the memory organization if the records are stored using chaining.

Q)

				P	Q	R	A	B	X	C	
mlocs:	1	2	3	4	5	6	7	8	9	10	11

Step: 1 $n = 9$, $m = 11$

load factor $d = \frac{n}{m}$ $d = \frac{9}{11}$ $\boxed{d = 0.81}$

Step: 2 $S(d) = ?$ $U(d) = ?$

$$S(d) = \frac{1}{2}\left[1 + \frac{1}{(1-d)}\right], \quad U(d) = \frac{1}{2}\left[1 + \frac{1}{(1-d)^2}\right]$$

$$S(d) = \frac{1}{2}\left[1 + \frac{1}{1-0.81}\right], \quad U(d) = \frac{1}{2}\left[1 + \frac{1}{(1-0.81)^2}\right]$$

$\boxed{S(d) = 3.13}$, $\boxed{U(d) = 14.35}$

	P	Q	R	A	B	C	V	W	X
$S =$	1	+ 1	+ 2	+ 2	+ 1	+ 1	+ 2	+ 2	+ 6

$n(9)$

$S = \frac{18}{9}$ $\boxed{S = 2}$

S $S(d)$

$2 < 3.13$

efficient

$U =$	1	2	3	4	5	6	7	8	9	10	11
	3	+ 2	+ 1	+ 7	+ 6	+ 5	+ 4	+ 3	+ 2	+ 1	+ 4

$n(11)$

$U = \frac{38}{11}$ $\boxed{U = 3.45}$

U $U(d)$

$3.45 < 14.35$ efficient

(b)

	link
1	8
2	0
3	0
4	9
5	3
6	4
7	0
8	5
9	0
10	0
11	7

	Info	link
1	P	0
2	Q	0
3	R	1
4	A	0
5	B	0
6	C	0
7	V	6
8	W	0
9	X	2
10		
11		

Efficiency $n = 9$, $m = 11$

load factor $\alpha = \frac{n}{m}$ $\alpha = \frac{9}{11}$ $\boxed{\alpha = 0.81}$

$\boxed{S(\alpha) = 1 + \frac{1}{2}\alpha}$ $\boxed{U(\alpha) = e^{-\alpha} + \alpha}$

$S(\alpha) = 1 + \frac{1}{2} \times 0.81$, $U(\alpha) = 0.4459 + 0.8$

$\boxed{U(\alpha) = 1.25}$

$S(\alpha) = 1 + 0.40$

$\boxed{S(\alpha) = 1.40}$

$S =$ A A A A B C V W X

$2 + 2 + 1 + 1 + 1 + 2 + 1 + 1 + 1$

$n(9)$

$S = \frac{12}{9}$ $\boxed{S = 1.33}$ / $S(\alpha)$ S

1.40 > 1.33

efficient

$$U = \underbrace{1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1}_{n(11)}$$

$$U = \frac{11}{11} \qquad \boxed{U = 1}$$

$U(d)$		U
1.25	>	1

efficient

Q) List the areas of applications of data structure.

ans ⇒
- Database : data storage and retrieval.
- operating system
- Compiler design
- Artificial intelligence
- Game development
- Machine learning
- Blockchain

Q) with the help of code snippet explain Complexity analysis.

ans ⇒

```
int main()
{ int i, n=8;                    // Hello & time.
for (i=1; i<=n; i++)
{ cout << "Hello" << endl;
} return 0;                      O(n)
} // The time Complexity of above code is because hello is printed n time. on the screen
```

Auxiliary space : O(1)

You can give one more example from your own in 4 marks.

Page No.

(Pg8) which data structure is ideal to perform Recursion operation and why?

Ans⇒ Stack data structure is ideal for perform recursion operation because it allow for easily maintain functions calls and returns. Each function call is pushed onto stack and when a function completes it is popped off the stack.

(Pg8) why it is said that searching a node in a binary search tree is efficient than that of a simple binary tree?

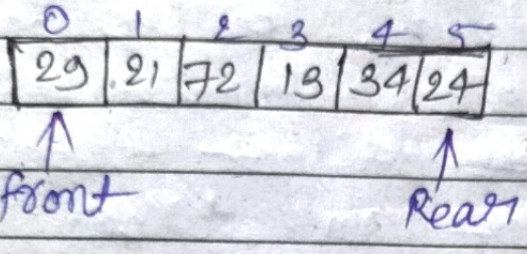
Ans⇒ Searching a node in a binary search tree (BST) is more efficient than a simple binary tree because a BST has a specific property: for each node, all in its left subtree have values less than the node, and all nodes in its right subtree have values greater than the node. This property allows for a more efficient search.

(Pg9) Discuss how circular queue overcomes limitations over linear queue with example.

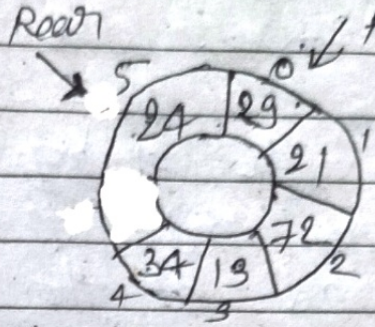
Ans⇒ A circular queue overcome limitations over linar queue by addressing the problem of wasted space at the front of the queue. In a linear queue, when elements are deleted the front space becomes unusable.

for example.

Consider both linear and circular queue of size 6 element

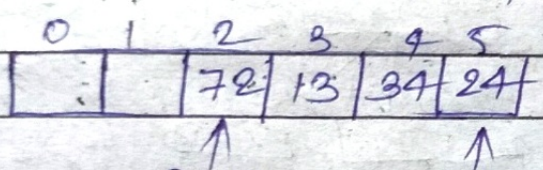


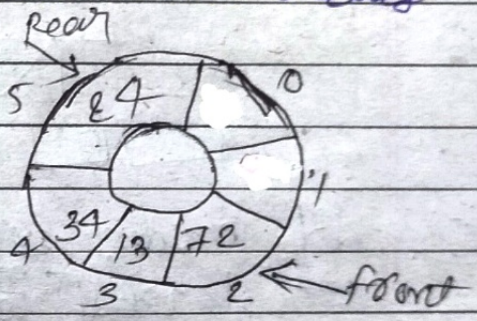
Linear queue



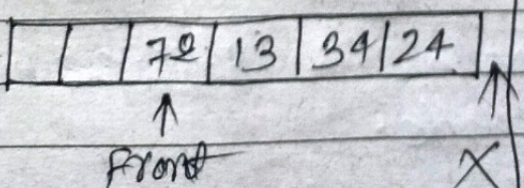
Circular queue

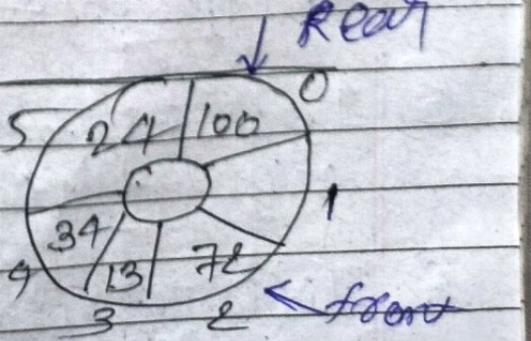
- When the dequeue operation is performed on the both the queues : Consider the ~~first~~ 2 elements that are deleted from bothe the queues.





- Now enqueue operation is performed : Consider an element with a value of 100, the insertion of element 100 in linear queue, is not possible but in circular queue rear will be gone at the 0 th index and value will be inserted.



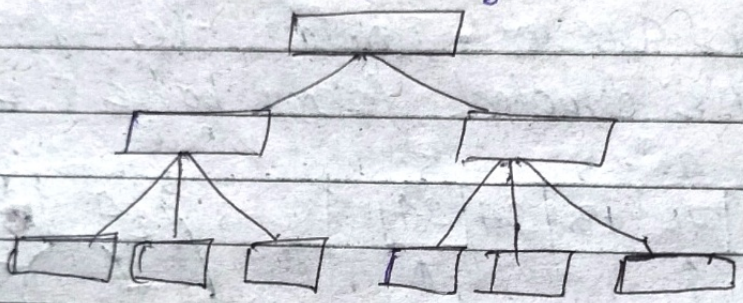


Q30 Write a brief Note on B-trees.

B-tree also called m-way tree. It is a self-balanced Tree in which every node contains multiple keys and has atleast two child and at the most m childs.

B-Tree of order m has the following properties

1. All the leaf nodes must be at same level.



2. All nodes except root node must have at least $\left(\left\lceil \frac{m}{2} \right\rceil - 1\right)$ keys and at the most $(m-1)$ key.

3. All the key value in a node must be in ascending order.

4. B-Tree have at least 2 child nodes and at the most m child nodes.

5. Insertion of a node in B-Tree happens only at leaf Node.

6. Time Complexity of B-Tree in
searching
inserting
deleting
→ O(log n)

Q) What is hashing, need of hashing? Discuss the various hash functions.

Ans → Hashing is the process of generating a fixed-size output from variable size input using mathematical formulas known as hash functions.

Need of hashing:

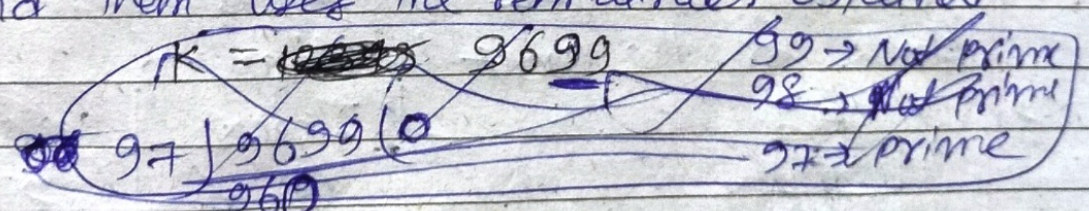
- Data retrieval • for index
- retrieve information from a database

hash functions

There are many hash functions

1. Division Method.
2. Mid square Method.
3. folding Method.

(1) Division method: In division method hash function divides the value K by M and then uses the remainder obtained.



K = 1276 M = 11

h(1276) = 1276 mod 11

1276 % 11 = 0

(ii) Mid square method: square the value of the key → k^2 and extract the middle r digits as the hash value.

Ex → K = 60, k^2 = 3600 = 60

(iii) folding method: It involve two method
(i) without reverse
(ii) with reverse

Ex → 32/05 32+05 = 37

(ii) Ex → with reverse: 32/05 → 32/50 →
32+50 = 82

Ex (i) 7148
71+48 = 119 = 19

Ex → (ii) 71+84 = 155 = 55

* open hashing of a technique to resolve collision in hashing.

ans → In linear probing the table is searched sequentially that starts from the original location of the hash if in case the location that we get is already occupied, then we check for the next location

~~H(1) = 5 % 5 = 0~~

Linear probing : $h, (h+1)\%N, (h+2)\%N, \ldots$

Ex→

A	B	C	D	E	X	Y	Z
4	8	2	11	4	11	5	1

Mloc →	1	2	3	4	5	6	7	8	9	10	11
	X	C	Z	A	E	Y		B			D

for E ⇒ $(h+1)\%N \Rightarrow (4+1)\%11 = 5\%11 = 5$

for X ⇒ $(h+1)\%N \Rightarrow (11+1)\%11 = 12\%11 = 1$

for Y = $(h+1)\%N \Rightarrow (5+1)\%11 = 6\%11 = 6$

for Z = $(h+1)\%N = (1+1)\%11 = 2\%11 = 2$ ✗

$(h+2)\%N = (1+2)\%11 = 3\%11 = 3$

Q) Describe quadratic probing as a technique to resolve collision in hashing

Ans→ Quadratic probing is a collision resolution technique in hash tables, when a collision occurs, quadratic probing searches for the next slot by using a quadratic function.

$H(k) = h, (h+1^2)\%N, (h+2^2)\%N, (h+3^2)\%N \ldots$

Ex→

A	B	C	D	E	X	Y	Z
4	8	2	11	4	11	5	1

	X	C		A	E	Y		B		Z	D
Mloc →	1	2	3	4	5	6	7	8	9	10	11

for E → $(h+1^2)\%N = (4+1^2)\%11$

$= 5\%11 = 5$

for X → $(h+1^2)\%N = (11+1)\%11 = 12\%11 = 1$

for Y → $(h+1^2)\%N = (5+1)\%11 = 6\%11 = 6$

for Z → $(h+1^2)\%N = (1+1)\%11 = 2\%11 = 2$ X

again $(h+2^2)\%N = (1+4)\%11 = 5\%11 = 5$ X

$(h+3^2)\%N = (1+9)\%11 = 10\%11 = 10$