Unit-V STACK

Rainbow
PAGE:
DATE: / /

*

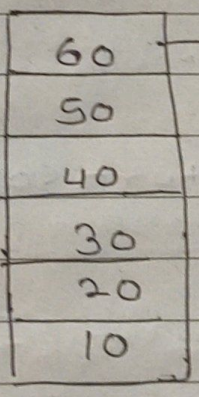
① What is stack? Explain stack as ADT?

→ - A stack is a data structure in which all insertion and deletion are performed from one end which is called TOP.

- Stack work on principle of LIFO manner means the element which is inserted at last will be remove first.

- For example:-

If we have to make stack of element 10, 20, 30, 40, 50, 60 then 10 will be boottomost element and 60 will be TOP most element.



STACK

* STACK AS AN ADT:-

STACK is collection of element in which insertion and deletion of element is done by one end called TOP.

Rainbow
PAGE:
DATE: / /

There are various operation performed on stack:-

① ~~SET~~ isFull():-

- This condition check whether the stack is Full or not.
- If the stack is Full then we cannot insert the element in stack.

• Pseudocode:-

```
int is_Full()
{
    if (top == SIZE-1)
    {   return(1);
        // Stack is Full
    }
    else
    {
        return(0);
    }
}
```

② isempty():-

- This condition check whether the the stack is empty or not.
- If the stack is empty then we cannot insert element in stack.

• Pseudocode:-

```
int is_empty()
{
    if (top == -1)
    {
        cout << "Stack is empty";
        return(1);
    }
    else
    {
        return 0;
    }
}
```

③ Push:-

- By this operation we can push element onto stack.
- Before performing push operation we must check isFull() condition.

• Pseudocode:-

```
void push()
{
    if (is_full() == 1)
    {
        cout << "Stack is full";
    }
    else
    {
        cout << "Enter element=";
        cin >> ele;
```

```
    top++;
    stack[top]=ele;
  }
}
```

④ POP :-

- By this operation one can remove the element from stack.
- Before popping element from stack we should check isempty() condition.

*Pseudocode :-

```
void pop()
{
  if(is_empty==1)
  {
    cout<<"stack is empty";
  }

  else
  {
    cout<<"Element popped"<<stack[top];
    top--;
  }
}
```

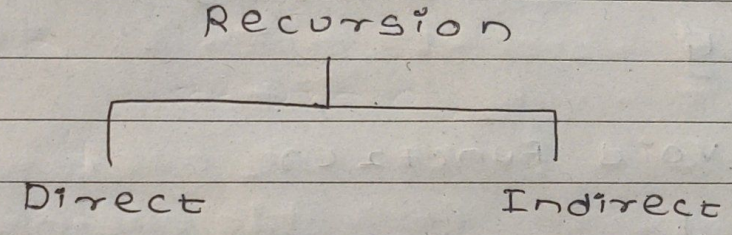
PAGE:
DATE: / /

② What is recursion? Explain use of stack in word processor for undo operation?

→

- Recursion is programming technique in which the function calls itself repeadely for some input.

- There are two type of Recursion



*Types of Recursion:-

① Direct Recursion:-

- Function calling itself from it's body is called Direct Recursion

```
void recursec()
{
    recursec();
}
int main()
{
    recursec();
}
```

Rainbow
PAGE:
DATE: / /

② Indirect Recursion:-
- Function call itself from another function viceversa is called indirect Recursion.

Example:-

```
void Func1()
{
    Func2();
}

void Func2()
{
    Func1();
}

int main()
{
    Func1();
}
```

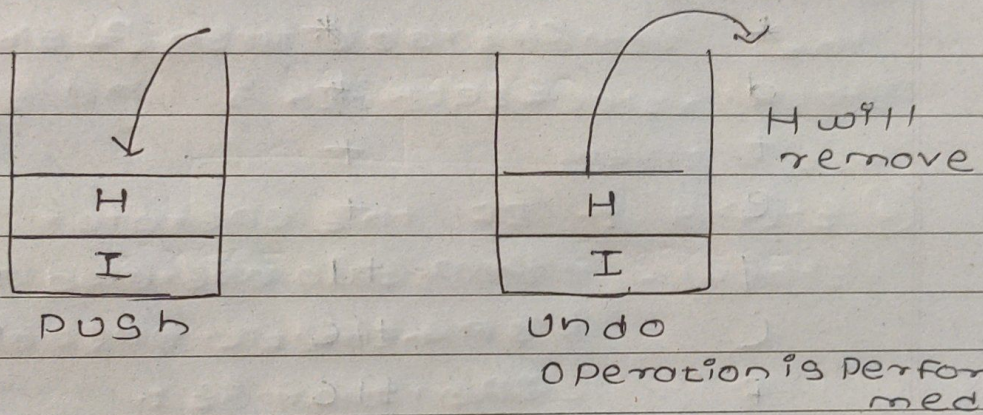
*Use of Recursion in Word:-

- Recursion concept is used in ms word for performing undo operation.
- When we type anything or perform any action that will push into stack.

- when we want to perform undo operation then the latest action or word is popped out from stack.

* For example :-

∴ Suppose we type HI it will push in stack



Push

Undo operation is performed.

Rainbow

PAGE:
DATE: / /

* (a+b) * d+e/(f+a *d)+c

→

Character	Stack	Postfix
(	(	-
a	(	a
+	(+	a
b	(+	ab
)	~~(+~~ -	ab+
*	*	ab+
d	*	ab+d
+	+	ab+d*
e	+	ab+d*e
/	+/	ab+d*e
(	+/(	ab+d*e
f	+/(	ab+d*ef
+	+/(+	ab+d*ef
a	+/(+	ab+d*efa
*	+/(+*	ab+d*efa
d	+/(+*	ab+d*efad
)	~~+/(+*~~ +/+	ab+d*efad*+
+	+	ab+d*efad*+/+
c	+	ab+d*efad*+/+c

ab+d*efad*+/+c+

Rainbow

PAGE :
DATE : / /

* 10, 2, *, 15, 3, /, +, 12, 3, 2, ^, +, +

1

Character	Stack	Evolution
10	10	
2	10, 2	
*	20	10 * 2
15	20, 15	
3	20, 15, 3	
/	20, 5	15/3
+	25	20+5
12	25, 12	
3	25, 12, 3	
2	25, 12, 3, 2	
^	25, 12, 9	3^2
+	25, 21	12+9
+	46	25+21

PAGE:
DATE: / /

* Algorithm for postfix expression:-

~~(1) Read P~~

(1) Step 1 :- START

(2) Step 2 :- Read postfix expression from left to Right

(3) Step 3 :- If input symbol read is operand then push it on stack.

(4) Step 4 :- If the operator is read POP two operand and perform arithematic operation if operator is :-

/ then result = operand1 / operand2
* then result = operand1 * operand2
+ then result = operand1 + operand2
- then result = operand1 - operand2

(5) Step 5 :-

Push the result onto stack.

(6) Step 6 :-

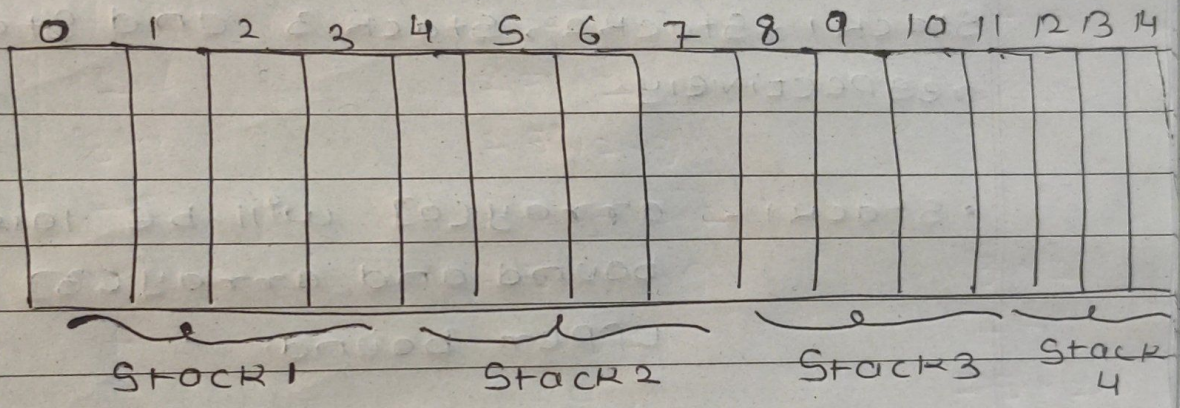
Repeat Step 1-4 till postfix expression is not over.

* Multi Stack :-

Q1) Explain concept of multiple stack with suitable concept.

→

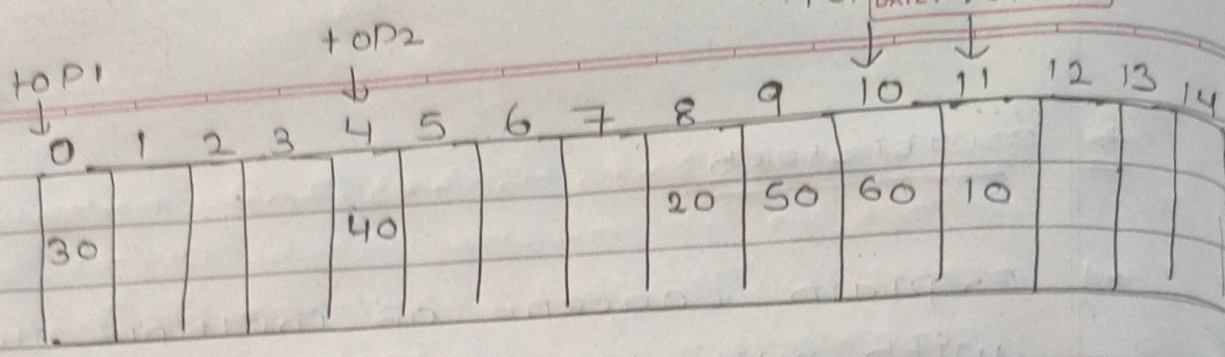
- In a single array any number of stack can be adjusted And push and push and pop operation on each individual stack can be performed.



- Each stack in one dimensional array can be of any size.

Example :-

1. Push 10 in Stack 4
2. Push 20 in Stack 3
3. Push 30 in Stack 1
4. Push 40 in Stack 2
5. Push 50 in Stack 3
6. Push 60 in Stack 3



- Here stack 3 is now full and we can not insert element in stack 3.

- the top1, top2, top3, top4 indicates the various top pointer for pointer stack1, stack2, stack 3 and stack 4 respectively.

- stack1 - array[0] will be lower bound and array[3] will be upper bound.

- stack 2 - The area for stack 2 will be from array[4] to array[7]

- stack 3 - From array[8] to array[10].

- stack 4 - From array[11] to array[14].

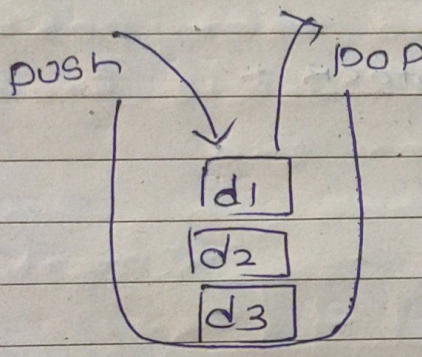
Unit-5

STACK

Rainbow
PAGE:
DATE: / /

(1) Define Stack?

→ A Stack is data structure in which the addition and removal of an element is performed from one end called as top of Stack.



(2) What is recursion? Explain use of Stack for undo operation in word processor.

→

- Calling a Function inside itself is called recursion and such Function is called as recursive Function.

```
void recurse()
{
    recurse();
}
int main()
{
    recurse();
}
```

recursive call

- stack can be used to keep recording of action taken by user for undo operation in word processor

c) Write pseudocode to represent stack as an ADT.

→

*Operation of stack :-

① Empty :-

```
int stempty()
{
    if(s->top==-1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

② Full :-

```
int sfull()
{
    if(s.top)==size-1)
    {
        return 1;
    }
    else
        return 0;
}
```

Rainbow
PAGE:
DATE: / /

③ Push ():-

```
void push (int item)
{
    st.top++;
    st.s[top] = item
}
```

④ POP():-

```
int pop()
{
    int item;
    item = st.s[top];
    st.top --
    return (item);
}
```

② Recursion:-

Rainbow

PAGE:
DATE: / /

Infix to Postfix

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

character	stack	post fix
(	(	-
(	((	-
a	((	a
+	((+	a
b	((+	ab
)	((	ab+
*	(*	ab+
(	(*(	ab+
c	(*(	ab+c
-	(*(-	ab+c
d	(*(-	ab+cd
)	(*	ab+cd-
)	-	ab+cd-*
/	/	ab+cd-*
E	/	ab+cd-*E
		ab+cd-*E/

*/
+-

PAGE:
DATE: / /

* a↑b*c-d+e/f/(g+h)

character	stack	POSTFIX
a	-	a
↑	↑	a
b	↑	ab
*	*	ab↑
c	*	ab↑c
-	-	ab↑c*
d	-	ab↑c*d
+	+	ab↑c*d-
e	+	ab↑c*d-e
/	+/	ab↑c*d-e
f	+/	ab↑c*d-ef
/	+/	ab↑c*d-ef/
(	+/(	ab↑c*d-ef/
g	+/(	g
+	+/(+	
h	+/(+	gh
)	+/	ab↑c*d-ef/gh+
		ab↑c*d-ef/gh+/+

* when precidence is small or equal ele will pop
* when precidence is large it will push into stack as it is
* when closing bracket is encounter all the element between opening & closing will be pop using Last in first out manner
* At the end all the remaining ele in stack will be pop in LIFO manner

Infix to postfix conversion

Rainbow
PAGE:
DATE: / /

(a+b)* d+e/(f+a*d)+c.

character	stack	postfix
(	(	
a	(	a
+	(+	a
b	(+	ab
)	—	ab+
*	*	ab+
d	*	ab+d
+	+	ab+d*
e	+	ab+d*e
/	+/	ab+d*e
(	+/(	ab+d*e
f	+/(	ab+d*ef
+	+/(+	ab+d*ef
a	+/(+	ab+d*efa
*	+/(+*	ab+d*efa
d	+/(+*	ab+d*efad
)	+/	ab+d*efad*+
+	+	ab+d*efad*+/+
c	+	ab+d*efad*+/+c

ab+d*efad*+/+c

ab+d*efad*+/+c+

Rainbow

PAGE:
DATE: / /

postfix Evaluation

① 10, 2, *, 15, 3, /, +, 12, 3, 2, 1, +, +

character	stack	Evaluation
10	10	
2	10, 2	
*	20	10 * 2
15	20, 15	
3	20, 15, 3	
/	20, 5	15/3
+	25	20+5
12	25, 12	
3	25, 12, 3	
2	25, 12, 3, 2	
9	25, 12, 9	3^2
+	25, 21	12+9
+	46	25+21

Algorithm : Post Evaluation

Input : post-fix expression

Output : solution post fix expression.

Step 1 : Start

Step 2 : Read postfix expression from left to right

Step 3 : If the input symbol read is an operand then push it on to stack

Rainbow

PAGE:
DATE: / /

step 4: If the operator is read POP two operands and perform arithmetic operations, if operator is

+ then res = op1 + op2
- then res = op1 - op2
* then res = op1 * op2
/ then res = op1 / op2

step 5: Push the result onto the stack.

step 6: Repeat steps 2-5 till the postfix expression is not over.

eg: 1,2,+,3,-,2,1,+,3,$

Expression	stack	Evaluation
1	1	-
2	1,2	-
+	3	1+2
3	3,3	-
-	0	3-3
2	2	-
1	2,1	-
+	3	2+1
3	3,3	-
$	27	3$3

```
        else

        if (strcmp(type,"operator")==0)/*if it is operator*/
        {
         op2 = pop();
         op1 = pop(); //popping two operands to perform
                        arithmetic operation
         switch(ch)
        {
        case '+' : result = op1 + op2;
                        break;
        case '-' : result = op1 - op2;
                        break;
        case '*' : result = op1 * op2;
                        break;
        case '/' : result = op1 / op2;
                        break;
        case '^' : result = pow(op1,op2);
                        break;
    }/* switch */
    push(result);
    }
        i++;
        ch=exp[i];
    } /* while */
    result = pop(); /*pop the result*/
    return(result);
}
```

Finally result will be pushed onto the stack.

5.9 : Linked Stack and Operations

Q.14 Explain the linked implementation of stack.

[SPPU : June-22, Marks 4]

Ans. : • The advantage of implementing stack using linked list is that we need not have to worry about the size of the stack.

- Since we are using linked list as many elements we want to insert those many nodes can be created. And the nodes are dynamically getting created so there won't be any stack full condition.
- The typical structure for linked stack can be

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

- Each node consists of data and the next field. Such a node will be inserted in the stack. Following figure represents stack using linked list.

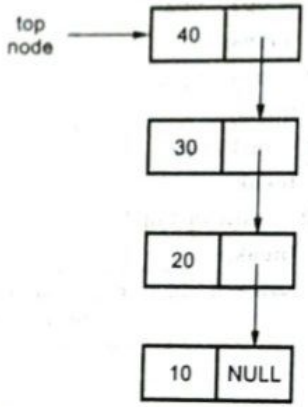


Fig. Q.14.1 Representing the linked stack

implementation.

[SPPU : June-22, Marks 4]

Ans. :

```
/*
-----------------------------------------
The Push function
-----------------------------------------
*/
void Lstack::Push(int Item, node **top)
{
```

UNIT - III

Rainbow
PAGE:
DATE: / /

* Binary search: very fast
- Binary search is ↑ searching algorithm.
- The algorithm is applied on sorted list
- mid = lower+upper this formula is used to find mid of array.
- If the search element is equal to mid then middle element is returned, otherwise we search into two halves according to result produced through the match.

* Algorithm for Binary search:

Input :- Sorted Array
Output:- element found or not.

(1) if (low > high)
(2) return;
(3) mid = (low+high)/2
(4) if (x == a[mid])
(5) return (mid);
(6) if (x < a[mid])
(7) search for x in a[low] to a[mid-1]
(8) else
(9) search for x in a[mid+1] to a[high];

Rainbow
PAGE:
DATE: / /

List (A) = 9, 17, 23, 38, 45, 45, 50, 57, 76, 90, 100
low ↑ ↑ high

$$mid = \frac{low + high}{2} = \frac{0+9}{2} = 4$$

checks 45 == 10 ≠ no, here 10 < 45 then search is continued in first part.

Then upper = a[mid] - 1
= 4 - 1
= 3

$$mid = \frac{lower + high}{2} = \frac{0+3}{2} = 1$$

∴ a[mid] > key
then search in first part and
high = a[mid] - 1 = 1 - 1 = 0

$$mid = \frac{0+0}{2} = 0$$

∴ key != a[mid]

∴ Element not found.

Rainbow

PAGE:
DATE: / /

Binary search example:

Key = 100

$$mid = \frac{low + high}{2}$$

$$= \frac{0 + 9}{2}$$

$$= 4$$

∵ Key > a[mid]

Then search second part & low = a[mid] + 1

i.e. 5

$$mid = \frac{low + high}{2}$$

$$= \frac{5 + 9}{2}$$

$$= \frac{14}{2}$$

$$= 7$$

∵ Key > a[mid]

then search second part & low = a[mid] + 1 i.e 8

$$mid = \frac{8 + 9}{2}$$

$$= \frac{17}{2}$$

$$= 8$$

Rainbow

PAGE:
DATE: / /

∵ key > a[mid] then ~~decrement~~

low = a[mid] + 1

= 8 + 1 = 9

∵ ~~key == a[mid]~~

mid = $\frac{low + high}{2}$

= $\frac{9 + 9}{2}$

= 9

∵ key == a[mid]

Elemend found at index 9

Rainbow

PAGE:
DATE: / /

Bubble sort:

* 81, 5, 27, -6, 61, 93, 4, 8, 104, 15

if a[i] > a[i+1] the swap a[i], a[i+1]

Pass 1:

81 5 27 -6 61 93 4 8 104 15

5 81 27 -6 61 93 4 8 104 15

5 27 81 -6 61 93 4 8 104 15

5 27 -6 81 61 93 4 8 104 15

5 27 -6 61 81 93 4 8 104 15

5 27 -6 61 81 93 4 8 104 15

5 27 -6 61 81 4 93 8 104 15

5 27 -6 61 81 4 8 93 104 15

5 27 -6 61 81 4 8 93 15 104

PAGE:
DATE: / /

Pass 2:

5 27 -6 61 81 4 8 93 15 104

5 27 -6 61 81 4 8 93 15 104

5 -6 27 61 81 4 8 93 15 104

5 -6 27 61 81 4 8 93 15 104

5 -6 27 61 4 81 8 93 15 104

5 -6 27 61 4 8 81 93 15 104

5 -6 27 61 4 8 81 15 93 104

Pass 3:

5 -6 27 61 4 8 81 15 93 104

-6 5 27 61 4 8 81 15 93 104

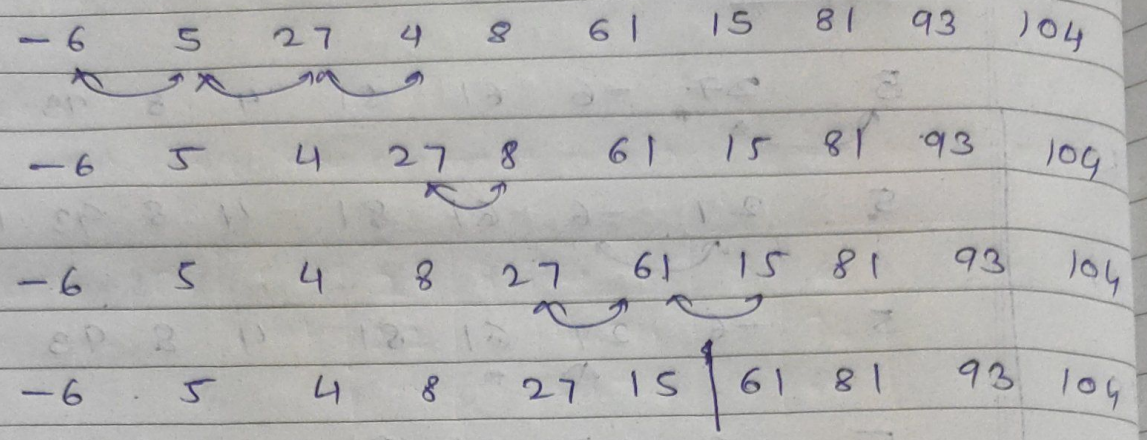
-6 5 27 4 61 8 81 15 93 104

-6 5 27 4 8 61 81 15 93 104

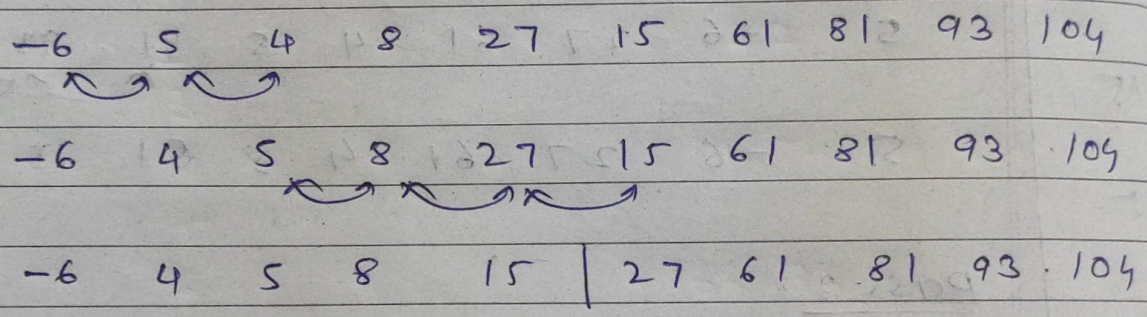
-6 5 27 4 8 61 15 81 93 104

-6 5 27 4 8 61 15 | 81 93 104

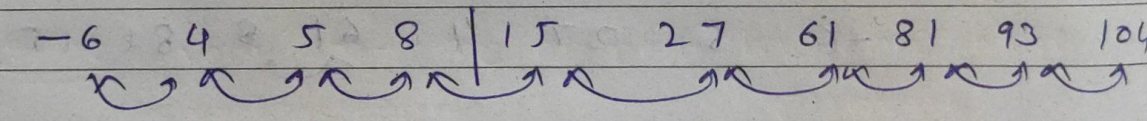
Pass 4:



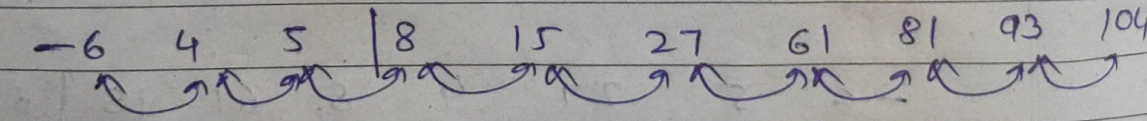
Pass 5:



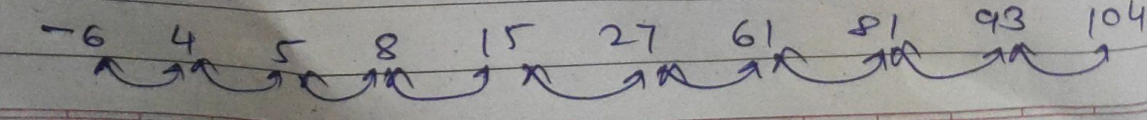
Pass 6:



Pass 7:



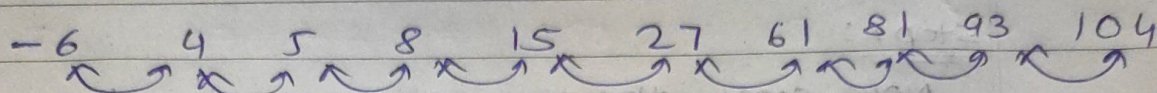
Pass 8:



Rainbow

PAGE:
DATE: / /

Pass 9:



Thus we got a sorted list of element as :-

-6	4	5	8	15	27	61	81	93	104

Rainbow
PAGE:
DATE: / /

Radix sort:

* 14, 1, 66, 74, 22, 36, 41, 59, 64, 54

Step1:
Now, sort the element according to last digit:

last	0	1	2	3	4	5	6
Element		1, 41	22		14, 54		36
					64, 74		66

ones digit	Element
0	
1	1, 41
2	22
3	
4	14, 74, 64, 54
5	
6	66, 36
7	
8	
9	59

Step2: Now, sort the element based on second last digit

second last digit	Element
0	01
1	14
2	22
3	36
4	41
5	54, 59
6	64, 66, ~~54~~
7	74
8	
9	

Rainbow

PAGE:
DATE: / /

since list of elements is of two digits that is why we stop comparing. Thus we get sorted list as :-

01 14 22 36 41 54 59 64 66 74

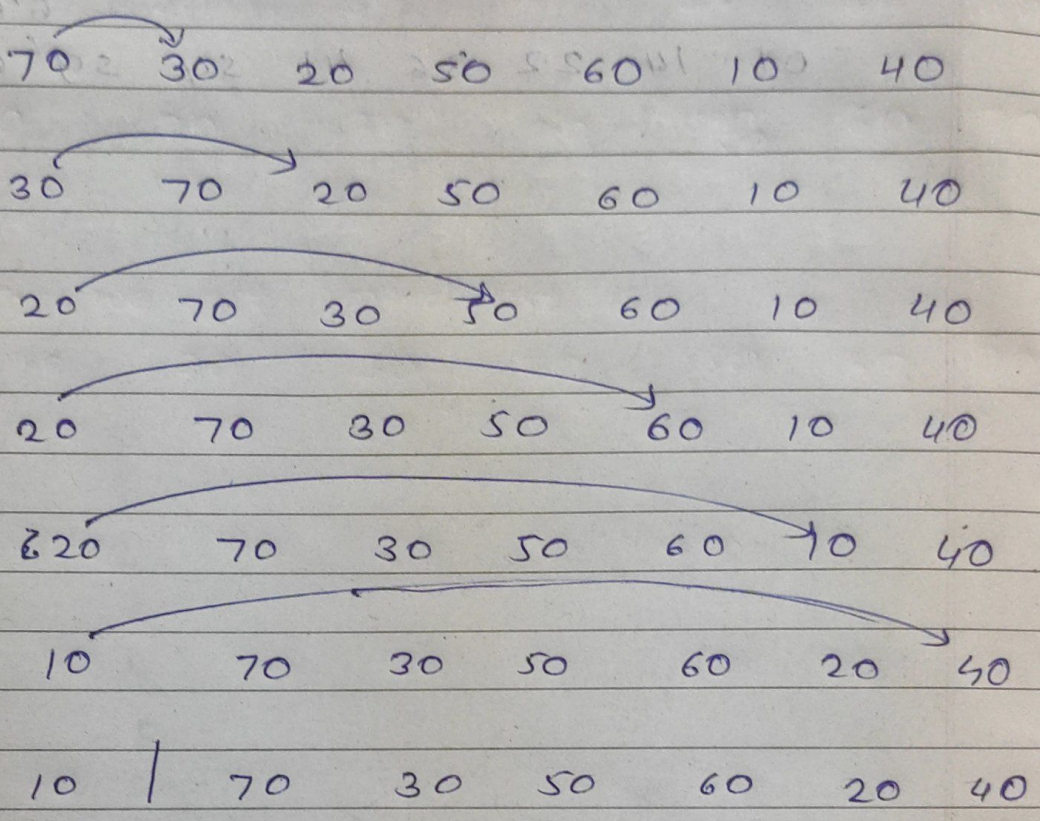
Rainbow

PAGE:
DATE: / /

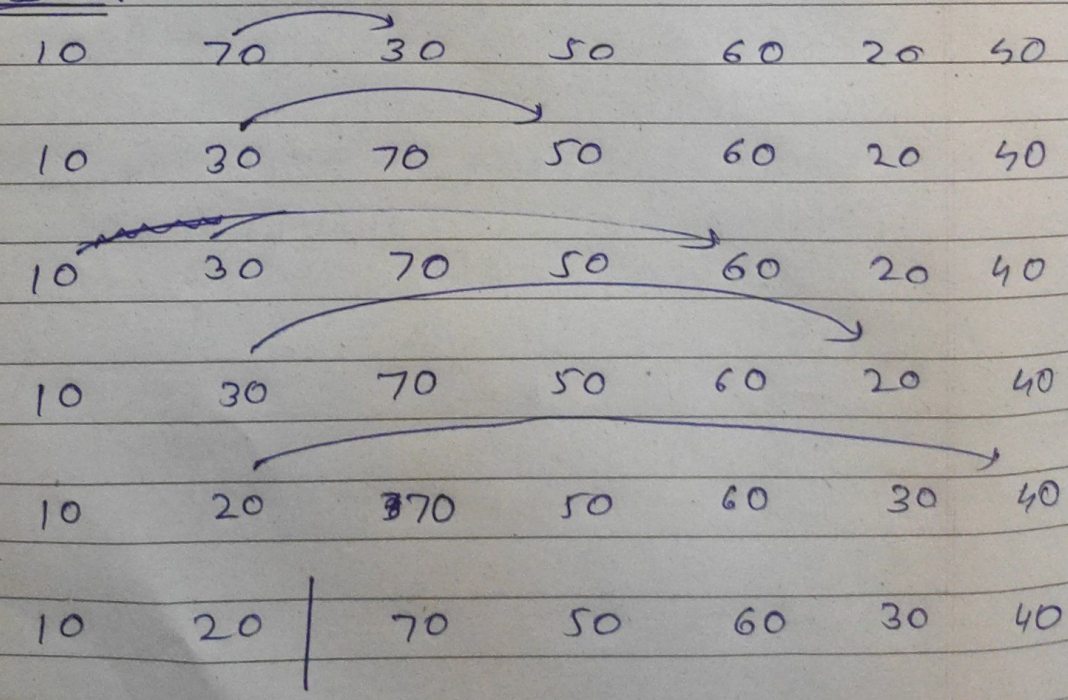
Selection sort :

* 70, 30, 20, 50, 60, 10, 40

Pass 1:



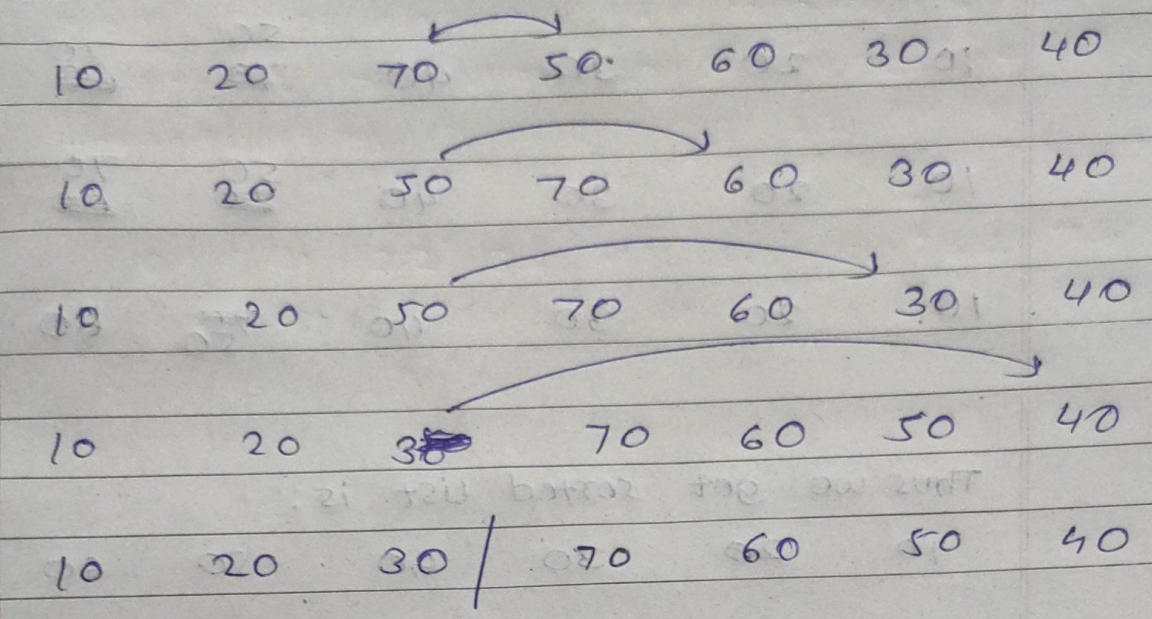
Pass 2:



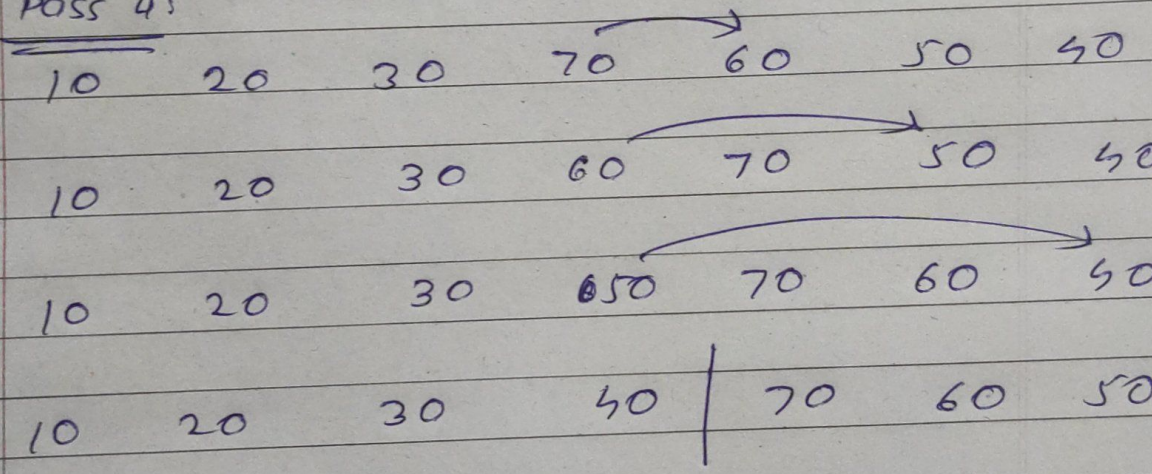
Rainbow

PAGE:
DATE: / /

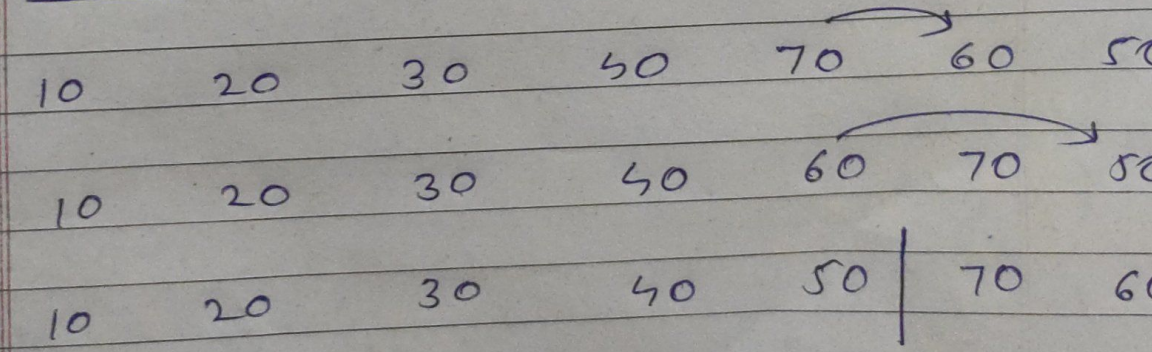
Pass 3:



Pass 4:



Pass 5:



Rainbow

PAGE:
DATE: / /

Pass 6:

10	20	30	40	50 ~~70~~	~~60~~ 70	80
10	20	30	40	50 ~~70~~	70 ~~60~~	80
10	20	30	40	~~70~~ 50	60	70

Thus we got sorted list is:

10	20	30	40	50	60	70

→ SHELL SORT

- Shell sort uses insertion sort to sort given list.
- In this sort, we require value of gap or interval.
- Initial value of gap = no. of elements in the list / 2

Example:

gap 4

23	29	15	19	31	7	9	5	2
23	29	15	19	31	7	9	5	2
23	7	15	19	31	29	9	5	2

pass 2

23	7	9	19	31	29	15	5	2
23	7	9	5	31	29	15	19	2
23	7	9	5	2	29	15	19	31
2	7	9	5	23	29	15	19	31

gap 2

2	7	9	5	23	29	15	19	31

pass 2

2	5	9	7	23	29	15	19	31
2	5	9	7	15	29	23	19	31
2	5	9	7	15	19	23	29	31

pass 3 gap 1

2	5	7	9	15	19	23	29	31

Algorithm:

1. Initialize the value of gap or interval as d/2
2. List is divided into sublists using gap.
3. Sort these sublists using insertion sort.
4. Repeat above steps until given list is sorted.

Analysis:

Complexity:

Best case: $\Omega(n)$

Worst case: $O(n^2)$

BUCKET SORT:

- Bucket sort is a sorting algorithm that works by distributing the elements of an array/list into a number of buckets.

algorithm:

1. set up an array of initially empty "buckets"
2. Scatter: Go over the Og array, putting each object in its corresponding bucket. sort each non-empty bucket
3. Gather: visit the buckets in order & PUll all elements back into the original array.

Example:

88, 12, 48, 96, 35, 78, 2, 56, 28, 61

2, 12	28, 35	48, 56	61, 78	88, 96
0-20	20-40	40-60	60-80	80-100

2, 12, 28, 35, 48, 56, 61, 78, 88, 96.

3.7 Comparison of all Sorting Methods and their Complexities

Sorting technique	Best case	Average case	Worst case
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Radix sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick sort	O(nlogn)	O(nlogn)	$O(n^2)$
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)
Shell sort	O(nlogn)	O(n)	O(n)

Rainbow

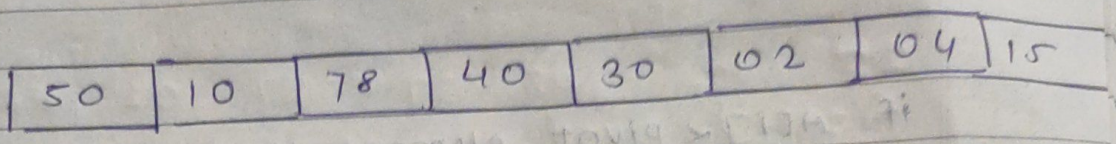
PAGE:
DATE: / /

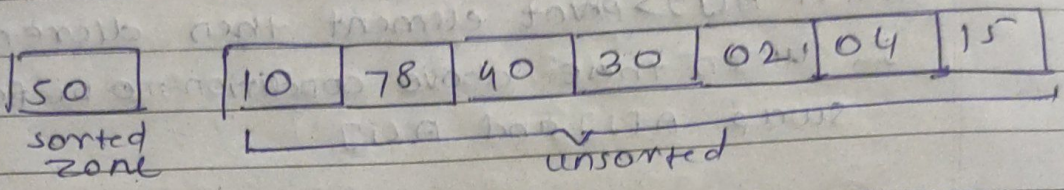
	Searching Technique	Best case	average case	worst case
1)	Binary search	O(1)	O(logn)	O(logn)
2)	Sentinel search	O(1)	O(n)	O(n)
3)	fibonacci search	O(1)	O(1)	O(logn)
4)	Linear search	O(1)	O(n)	O(n)

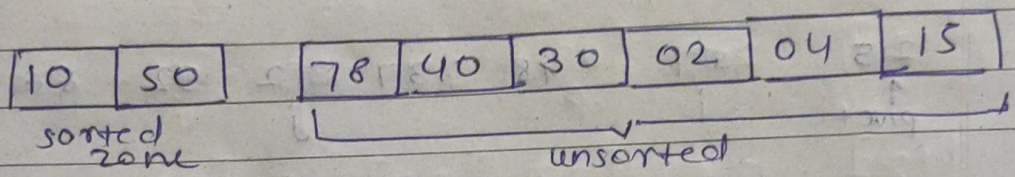
INSERTION SORT:

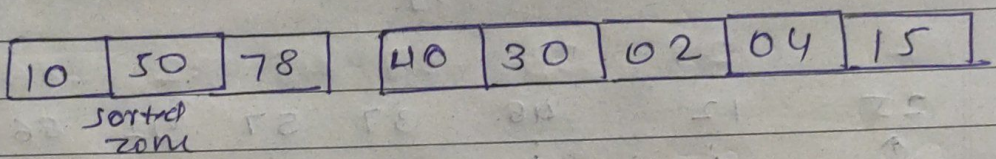
Rainbow
PAGE:
DATE: / /

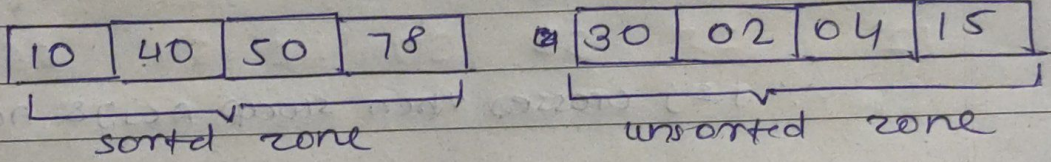
* 50 10 78 40 30 02 04 15

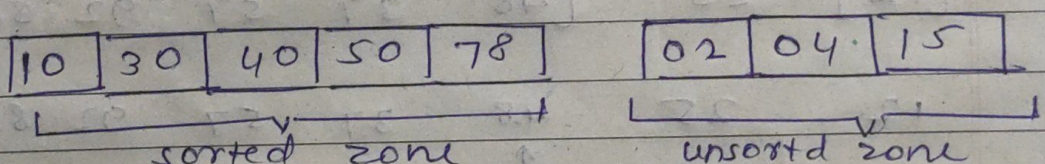


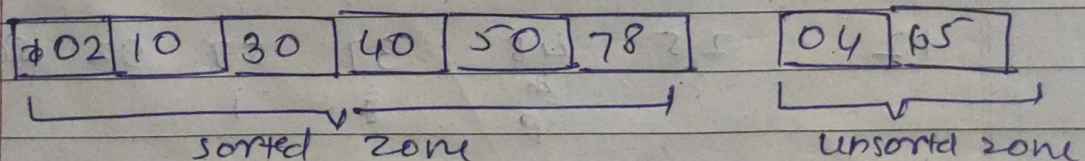


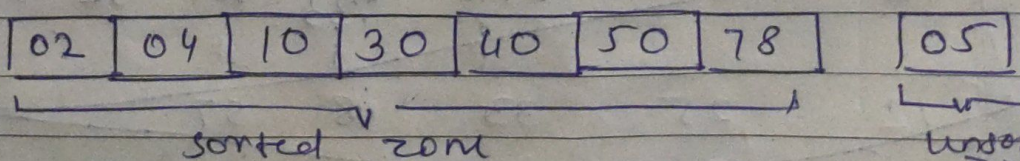


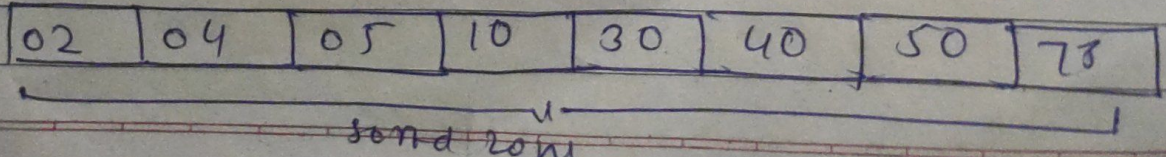












Rainbow

PAGE:
DATE: / /

QUICK SORT

25 57 48 37 12 92 86 33
p i j

25 12 48 37 57 92 86 33
p j i

[12] 25 [48 37 57 92 86 33]
p i j

[12] 25 [48 37 33 92 86 57]
p j i

[12] 25 [33 37] 48 [92 86 57]

12 25 33 37 48 92 86 57
p j i

12 25 33 37 48 57 86 92

Unit 4

Rainbow
PAGE:
DATE: / /

* Explain polynomial representation using linked list with an example.

→

- A polynomial p(x) is expression in variable x which is in the form $ax^n + bx^{n-1} + \dots + k$, where a, b, c, k ... are fall in category of real number and 'n' is non negative integer which is called degree of polynomial.

- An important characteristic of polynomial is that each polynomial mainly consist of two part:-
  1. coefficient
  2. exponent

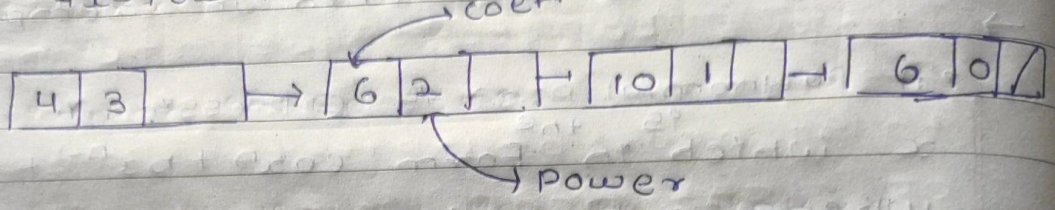
* Example:-

$-10x^2 + 26x$; Here 10 and 26 are coefficient and 2,1 are exponent value.

- Points to keep in mind while working with polynomial:-
  1. The sign of each coefficient and exponent are stored within the coefficient and exponent.
  2. The storage allocation for all terms in polynomial must be done in ascending and decending order of their exponent.

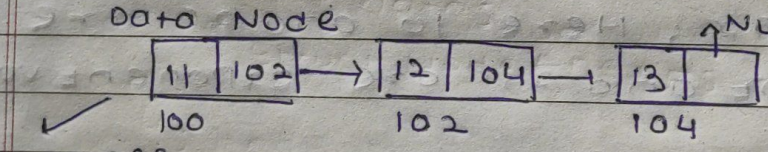
example:-

$4x^3+6x^2+10x+6$



b) Define linked list.

→ • Linked list is linear data structure which consists of element called as node.
• Every node contains two part data and next.
• Data consist of actual element while next contain address of next node.



c) Explain concept of generalized linked list.

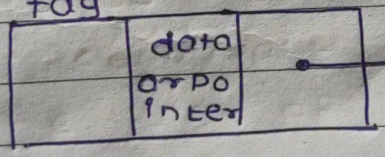
→ - All these type of linked list can contain only atomic values like integer, float, character etc.
- Generalized link list is special form of linked list in which element

Rainbow
PAGE:
DATE: / /

are either atoms or generalized list (sublist) themselves.

- Generalized linked list is finite set of zero or element (a0, a1, a2 ... an) such as ai is either atoms or ~~et.~~ sublist.

* Representation of Generalized List:
- The gll is same as of simple linked list, but there is need of extra field known as tag which indicate whether element is atom or sublist.
- The tag field contain value either 0 and 1. The value 1 indicates that element is sublist while value 0 indicates that element is atom.
- When node represents sublist, then it store address of starting node of sublist.

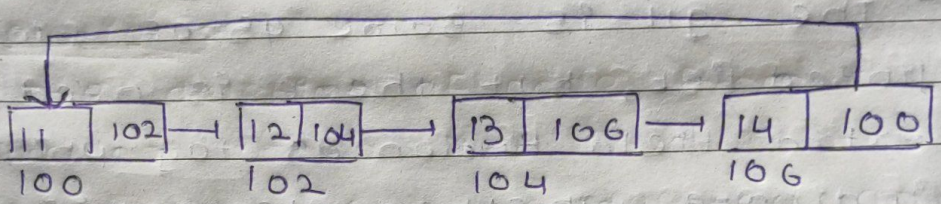


d) Write pseudo c++ code to represent circular linked list as an ADT.

→

* Circular linked list:-

- Circular linked list is linked list in which next part of last node contain address of 1st node to form a circle.



* To create Circular linked list:-

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

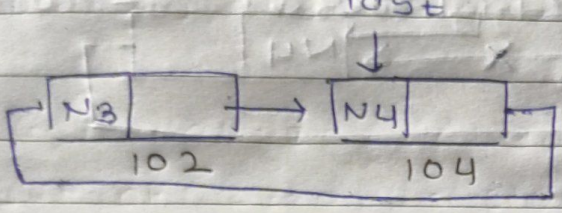
* ADT operation:-

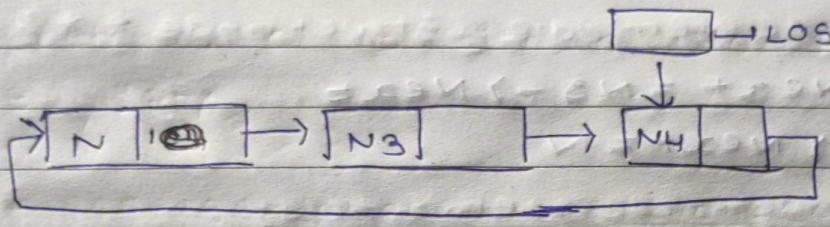
① Insertion:-

- We can insert a node in circular linked list at any position, that is at beginning, in end or middle.

Rainbow
PAGE:
DATE: / /

① Insert at beginning of list :-

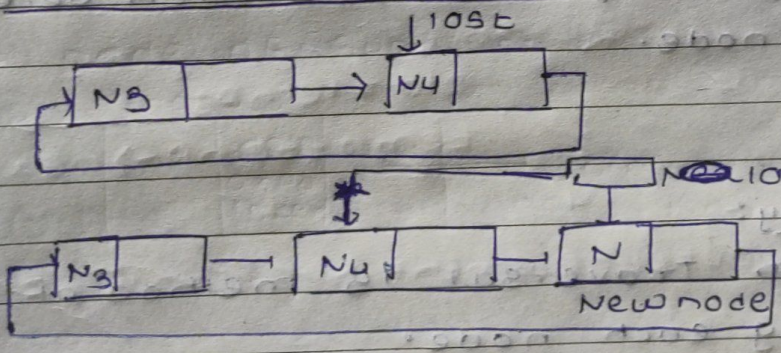




- When a new node is inserted at beginning of list, the next pointer of last node point to new node N thereby making it as 1st Node.

- n->next = last->next;
  last->next = n

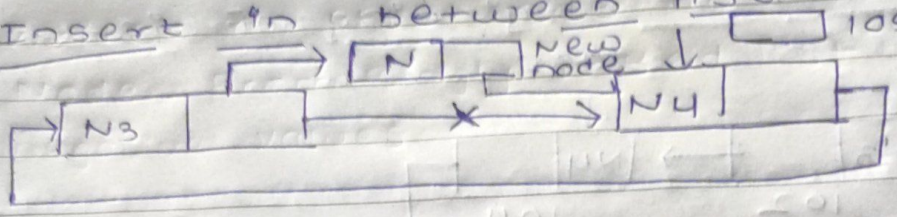
② Insert at end of list :-



n->next = last->next
last->next = n
last = n

PAGE:
DATE: / /

③ Insert in between list



N -> Next = N3 -> Next
N3 -> Next = N

④ Deletion :-

- The deletion of node from a circular linked list involves the node which is to be deleted and then freeing its memory
- For this purpose we need to maintain two pointer curr and prev and then traverse a list to find node.
- Based on location we set curr and prev pointer and then delete curr node.

⑤ Display :-

- ~~Trad~~ Traversal is method of visiting each node.
- In Circular linked list we start from 1st node and transverse each node we stop when we once again reach 1st Node.

```
void dll::display()
{
    node *temp;
    temp=head;
    if (temp==head)
    {
        cout<<"In Node is empty";
    }
    else
    {
        do
        {
            cout<<"\n"<<temp->data;
            temp=temp->next;
        } while (temp!=head)
    }
    getch();
}
```

e) Write an algorithm to delete intermediate node from doubly linked list.

→

Step 1 :- q=temp=head

Step 2 :- Read num

Step 3 :- Transverse temp

Step 4 :- Compare temp->data with num

Step 5 :- If data found :

If it is intermediate node

Link previous and next node of temp.

Step 6 :- Delete temp

Step 7 :- If data not found and temp!=NULL

Repeat From Step 3

Rainbow

PAGE:
DATE: / /

F) Write a c++ code to delete Node from singly linked list with negative values.

→

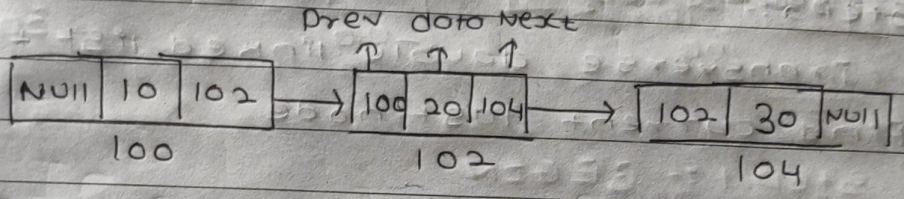
```
void deletemiddle (struct node **q, int num)
{
    struct node *temp, *old;
    temp = *q;

    while (temp->next != NULL)
    {
        if (temp->data < 0)
        {
            old->next = temp->next;
            free (temp);
            return
        }
        else
        {
            old = temp;
            temp = temp->next;
        }
    cout<<"element Not Found";
}
```

Rainbow
PAGE:
DATE: / /

* Linked list :-

5] Explain doubly linked list as ADT?

→ - Doubly linked list is a list in which every node contains three part :- data, previous, next.
- Data contain actual data while the previous and next part contain address of previous and next node.



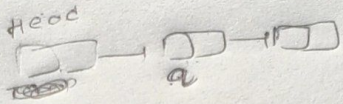
* Primitive operation :-

① creating a doubly linked list :-

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

② Insert Node at beginning of DLL.

Step 1 :- q = head
Step 2 :- Read data
Step 3 :- alloc (temp)
Step 4 :- Link temp's previous and next node to temp.



Rainbow
PAGE:
DATE: / /

(3) Insert Node at end :-

Step 1 :-
Create a node with given data

Step 2 :-
Set next pointer of new node to "NULL" since it will be last node

Step 3 :-
Transverse double linked list to find current last node

Step 4 :-
Set the next pointer of current last node to ~~next~~ new node

Step 5 :-
Set the previous pointe

Rainbow

PAGE:
DATE: / /

* ADT operation of doubly linked list:-

① Create:- To create a dll we have to create structure contain 3 element prev, next and data.

Algo:-

Step 1:- BEGIN

Step 2:- CREATE A NODE

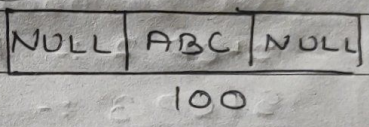
NEWNODE -> DATA = VALUE

NEWNODE -> PREVIOUS = NULL.

NEWNODE -> NEXT = NULL.

Step 3:- START = NEWNODE

Step 4:- EXIT.



② Insert at beginning:-

Node can be inserted at any place in linked list:-

① Beginning
② middle
③ end

Step 1:- BEGIN

Step 2:- CREATE A NODE

NEWNODE -> DATA = VALUE.

NEWNODE -> PREVIOUS = NULL

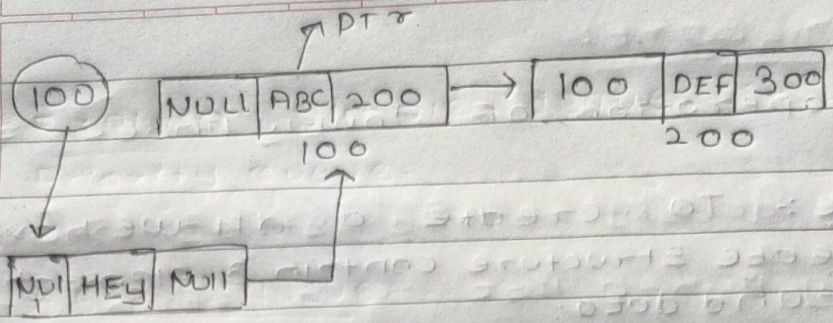
NEWNODE -> NEXT = NULL

Step 3:- NEWNODE -> NEXT = START.

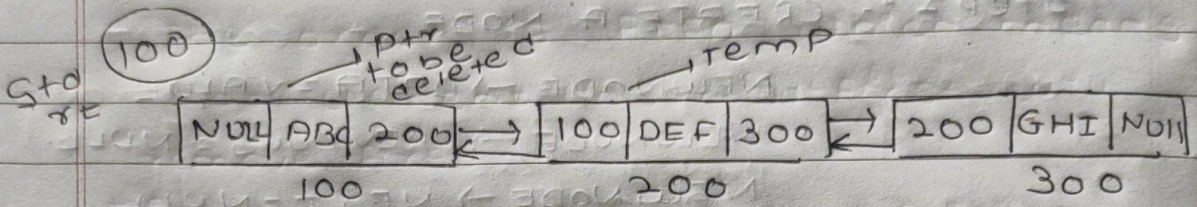
Step 4:- PTR -> PREV = NEWNODE.

Step 5:- START = NEWNODE.

Step 6:- EXIT.



(3) DELETE FROM BEGINNING :-



Step 1 :- BEGIN
Step 2 :- PTR = START
Step 3 :- START = PTR -> NEXT
Step 4 :- TEMP -> PREV = NULL
Step 5 :- FREE (PTR).
Step 6 :- EXIT.

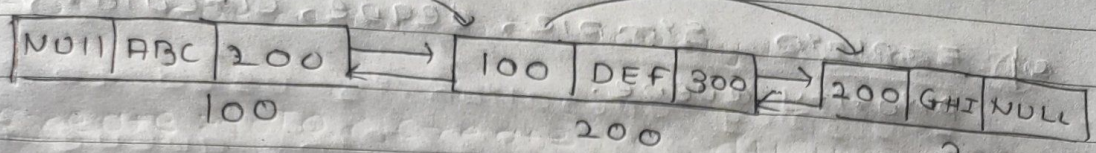
(4) Transversing Node in linked list :-

ALGO :-
Step 1 :- BEGIN
Step 2 :- IF START = NULL then PRINT LIST is EMPTY.
Step 3 :- ELSE PTR = START
Step 4 :- PRINT PTR
Step 5 :- WHILE (PTR -> NEXT != NULL).
{
PTR = PTR -> NEXT
PRINT PTR -> DATA;
}
Step 6 :- EXIT.

START

100

path



Rainbow
PAGE:
DATE: / /

3] Explain ~~stgn~~ singly linked list as ADT.

⇒ Singly linked list :-

- It is also called as linear linked list.
- It is a basic form of linked list.
- In this, every node is made up of two parts data and next. Data part contains the actual element while next part contains the address of next node.

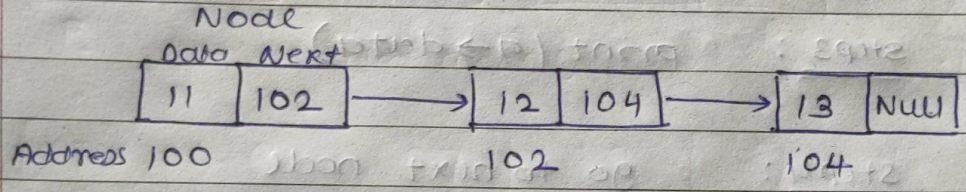


fig: Singly linked list.

Primitive operations on singly LL.

(i) Create :-

To create a ~~node~~ SLL we create a structure contains two elements:

Data: will contain actual data part

Next: will contain address of next node

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

Rainbow
PAGE:
DATE: / /

(ii) Traversing

Traversing means the process of visiting each node at least once.

Algorithm

Step 1 : if head == NULL
print (List is empty)
else

Step 2 : q = head

Step 3 : print (q → data)

Step 4 : go to next node

Step 5 : if q != NULL
Repeat from step 3 (3 & 4)

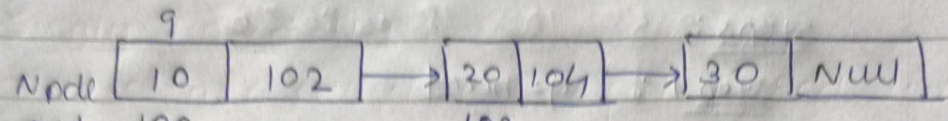
Step 6 : End.

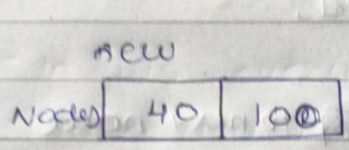
(iii) Inserting a Node in SLL :

A node can be inserted at any ~~place~~ in place in linked list :

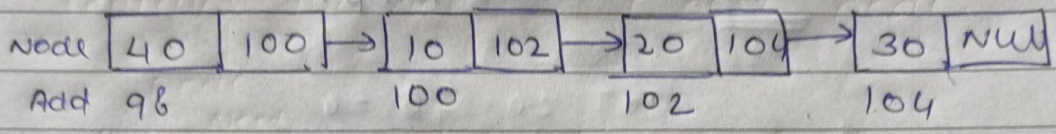
- At the end
- At the beginning
- In the middle of

* Adding at the beginning and end of SLL





create new node



Steps & Algorithm

Step 1 : set q at head

Step 2 : Read data

Step 3 : alloc(newNode)
newNode->data = data

Step 4 : newNode->next = q

Step 5 : Set q at newnode.

Rainbow

PAGE:
DATE: / /

*(iv) Deleting a Node in SLL:

Steps:

Step 1: q = head

Step 2: temp = head;

Step 3: traverse q to next node

Step 4: delete temp.

UNIT- VI

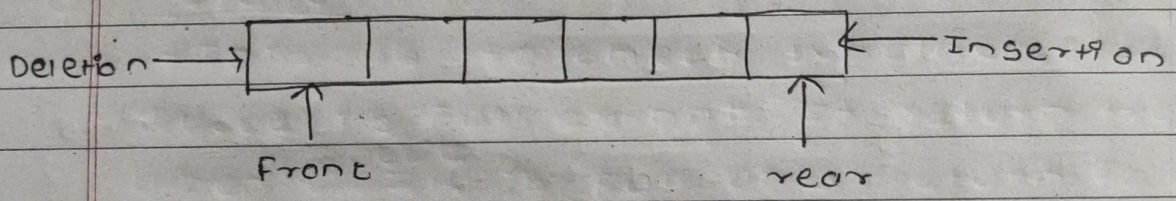
Rainbow
PAGE:
DATE: / /

Q1) Explain simple, deque, circular, priority queue.

→

* Simple queue:

- A queue is linear data structure which follows a particular order in which the operation are performed.
- The order is FIFO which means that element inserted first will remove first.
- Front pointer points to beginning of node and rear pointer points to end of node.



*Queue as an ADT:-

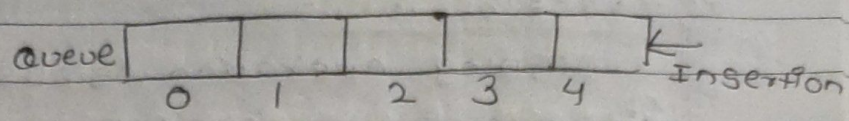
- Different operation are performed on queue:-

1. Enqueue - Add element
2. Dequeue - Remove element
3. isEmpty - Check if queue is empty
4. IsFull - Check if queue is full
5. Peek - Get value of Front of queue without removing it

Rainbow

PAGE:
DATE: / /

① Insertion:



Step 1:- IF CREAR = MAX-1 then
Write "Queue is overflow"
Goto step 4
[End of IF]

Step 2 :- IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
EISE
SET REAR = REAR + 1
[END OF IF]

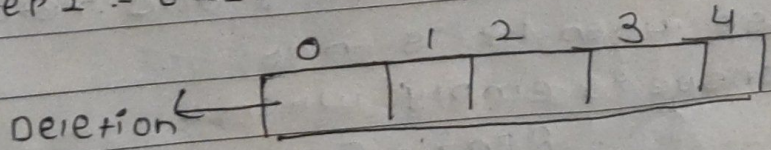
Step 3 :- SET QUEUE[REAR] = NUM
Step 4 :- EXIT

② DELETION :-

Step 1 :- IF FRONT = -1 OR FRONT > REAR
Write "Queue is underflow"

EISE
SET VAL = QUEUE[FRONT]
FRONT = FRONT + 1
[END OF IF]

Step 2 :- EXIT



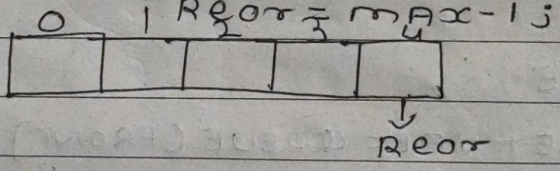
③ Display :-

- Display operation is used to display or tranverse the queue from front to rear.

```
For(int i=front; i<=rear; i++)
{
    cout<<queue[i]<<" ";
}
```

④ isFull() :-

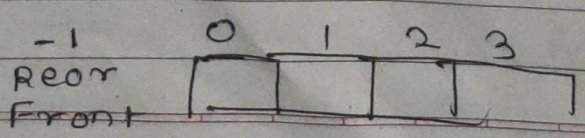
- This is to check whether the queue is full or not. It return true when ~~stack~~ queue is full and return false when it is not.
- queue is full when :-



⑤ isempty() :-

- It checks whether the queueue is empty or not. It return true when queue is empty and return false when it is not.
- queue is empty when :-

REar = front = -1

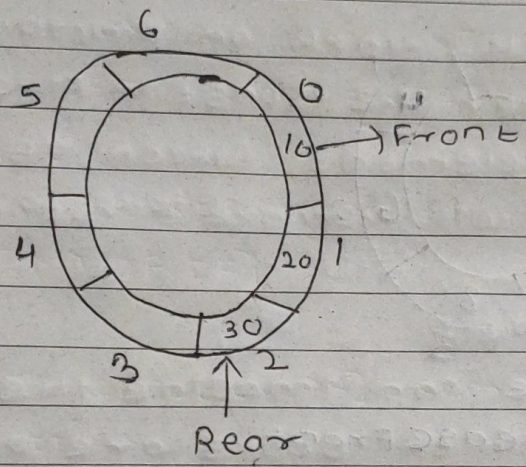


Rainbow
PAGE:
DATE: / /

* Circular queue :-

- Circular queue is linear data structure in which operation are performed based on FIFO principle.

- The last position is connected back to 1st position to make a circle.



* Operation :-

① Enqueue :-

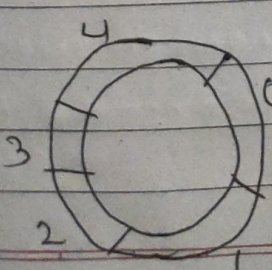
- Adding an element in queue if there is space in queue.

STEP 1 :- Check if queue is full

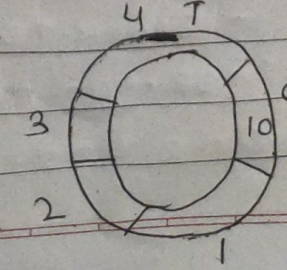
STEP 2 :- For 1st element, set value of Front to 0.

Step 3 :- Circularly increase rear index by 1.

Step 4 :- Add new element in position pointed by REAR



1) Initially Front=0 and Rear=-1



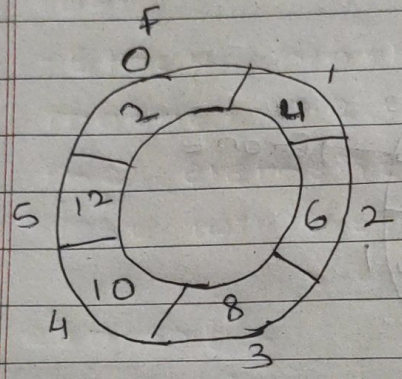
2) Add item 10 when front=0 rear=0

Rainbow

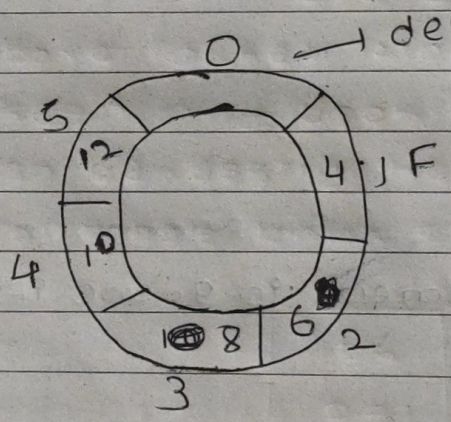
PAGE:
DATE: / /

② DEQUE operation :-

① Check if queue is empty
② Return value pointed by Front
③ Circularly increase Front index by 1
④ For last element set Front and rear to -1.



increase Front

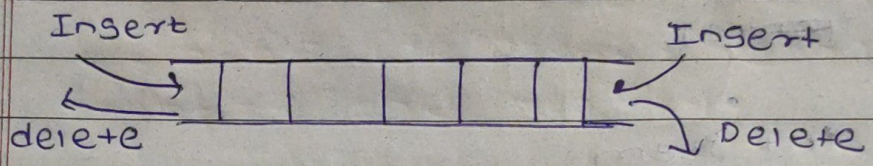


Queue

1) Explain deque with insert and delete operations performed on it.

→

- Double ended queue is data structure in which insertion and deletion are allowed at both end.
- That means, we can insert at both Front and rear position and as well we can delete from both Front and rear end.

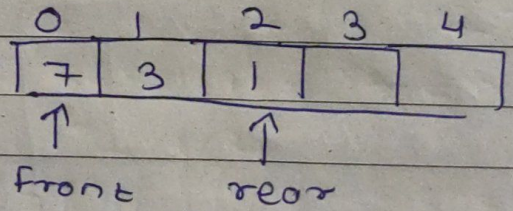


* Insert operation:

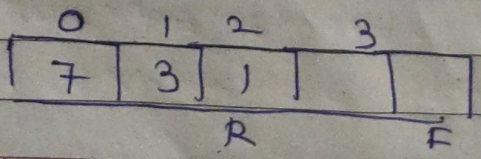
~~Enqueue~~ :

① Insert at Front :-

① Check position of Front



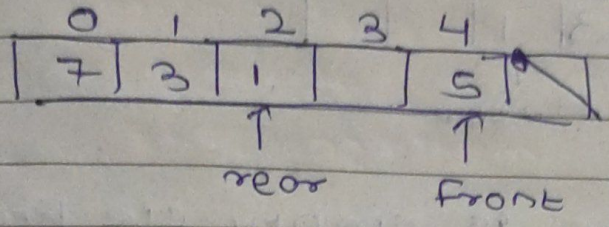
② If Front < 1, reinitialize Front = n-1 (last index)



Rainbow
PAGE:
DATE: / /

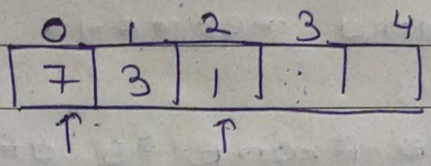
③ Else,
decrease Front by 1

④ Add Key 5 into array [front]



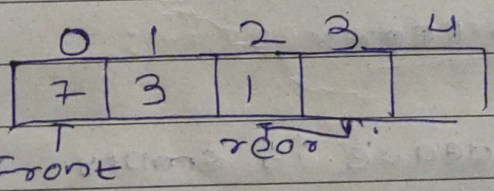
② Insert at Rear :-

① Check if array is Full

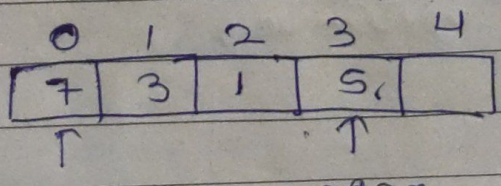


② If deque is Full,
reinitialize
rear=0.

③ Else, increase rear by 1

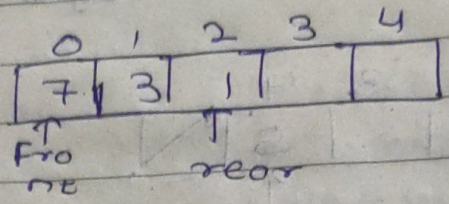


④ Add Key (5) into array [rear]



③ Delete From Front:-

① Check if deque is empty

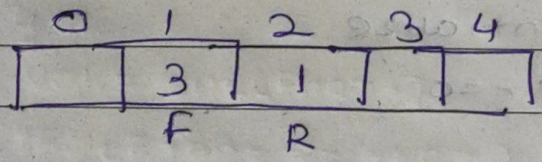


② If deque is empty (i.e Front = -1), deletion cannot be performed.

③ If deque has only one element (i.e Front = rear), set Front = -1 and rear = -1.

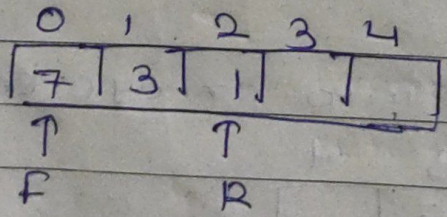
④ Else if if Front is at end Front = n-1, set go to Front = 0.

⑤ Else Front = Front + 1.



④ Delete From Rear:-

① Check if deque is empty.



② If deque is empty (i.e Front = -1) deletion cannot be performed.

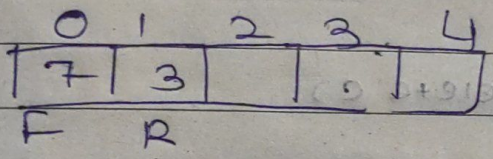
Rainbow
PAGE:
DATE: / /

③ If deque has only one element i.e Front == rear, set front = -1 and rear = -1, else follow step below.

④ If rear is at Front set go Front rear = n-1.

⑤ Else,

rear = rear - 1.



②

② Write pseudo code to implement a circular queue using array

→

```
//
Insert function
//

void queue :: insert (int item)
{
    if (front == (rear+1) % max)
    {
        cout << "queue is full";
    }
    else
    {
        if (front == -1)
            front = rear = 0;
```

```
else
    rear = (rear+1) % max;
    queue[rear] = item;
  }
}
```

1.

Delete Function

1.

```
int
void queue ::delete()
{
    int val;
    if (front==-1)
    {
        cout<<"queue is empty";
        return 0;
    }
    val = que[front];
    if (front==rear)
    {
        front=rear=-1;
    }
    else
    {
        front=(front+1) % MAX;
        return val;
    }
}
```

Rainbow
PAGE:
DATE: / /

③ What is priority queue? Describe operation on priority queue and explain its application.

→

• Priority queue is considered as an extension of queue with following given properties:-

① Every element in queue has priority associated with it.
② An element which has high priority is deqeued before element which has low priority.
③ If two elements have same priorities, they are served according to their order in queue.

• Application of priority queue:-

① Priority queues can be used in operating system for purpose of job scheduling. Here job with higher priority gets processed first.
② Graph algorithm such as Dijkstra's shortest path algorithm, Prim's minimum spanning tree etc.
③ Heap sort.
④ Data compression:- It is used in Huffman code which is used to compress data.

* what is backtracking algorithm design strategy?

→ - Backtracking is a problem-solving strategy used in algorithm design, particularly in problems involving optimization, combinatio and decision making. It is often employed when you need to explore all possible solutions to find the best one.

- It involves systematically trying different options, and if a chosen option does not lead to a valid solution, the algorithm backtracks to the previous decision point and explores another option.

- The general steps of a backtracking algorithm are:

(i) choose: make a decision on the next option to try.

(ii) Explore: move forward with choosen option & explore further.

(iii) Unchoose: If the current option does not lead to a solution, backtrack to the previous decision point and explore other options.

- This process continues recursively until a valid solution is found or all possibilities have been explored.
- Backtracking is commonly used in problems such as N-Queen problem, Sudoku and the subset sum problem.

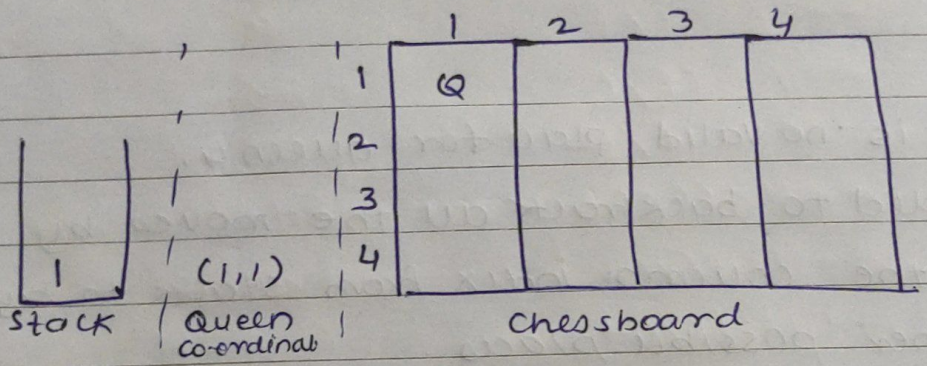
* Explain use of backtracking in 4-Queen's problem.

⇒ USE OF STACK IN BACTRACKING.

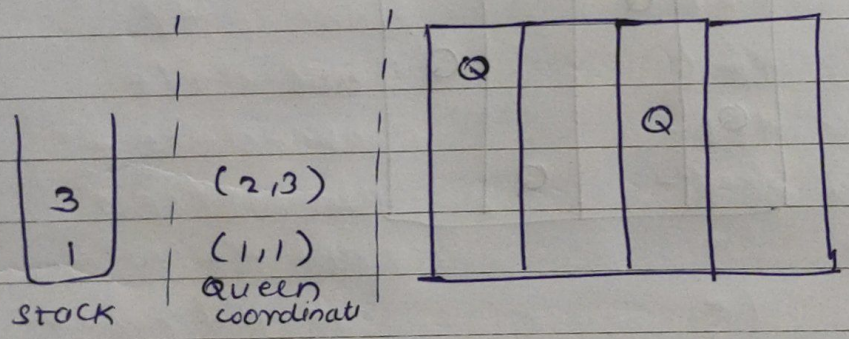
(i) Let us take 4 queens and 4×4 chessboard.

- Now we start with empty chessboard.

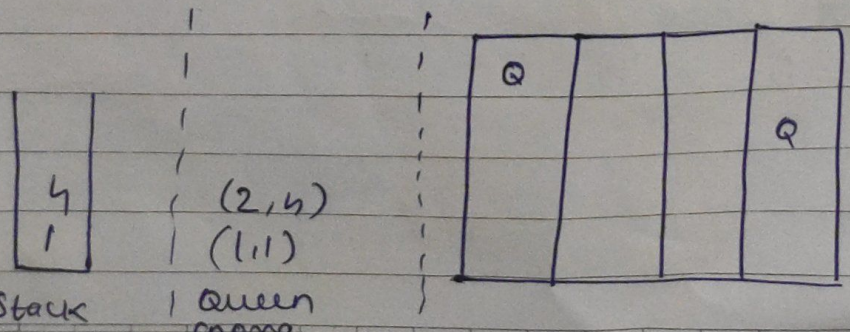
place queen 1 in the first possible position of its row ie. on 1,1.



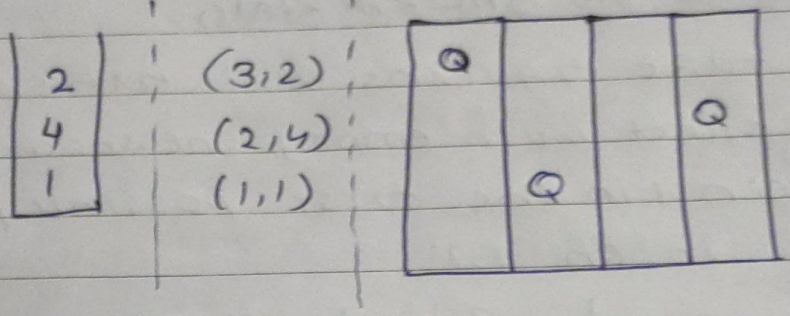
(ii) Then place queen 2 after trying unsucessful place - 1(1,2), (2,1), (2,2) at (2,3) ie. 2nd row & 3rd column.



(iii) This is dead end because at 3rd queen cannot be placed in next column, as there is no acceptable possible for queen 3. Hence algorithms backtracks and places the 2nd queen at (2,4) position

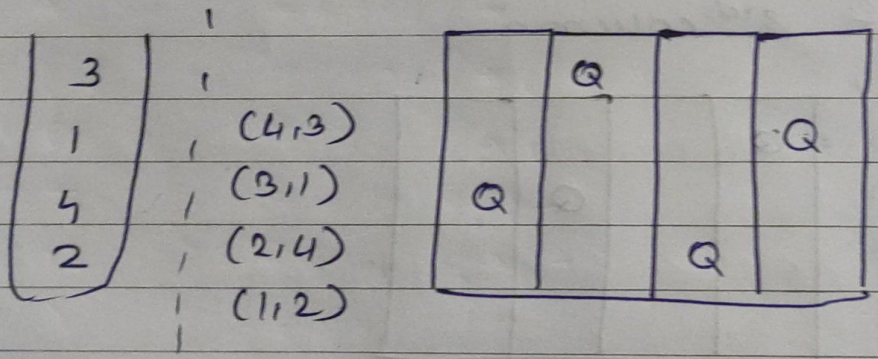


(iv) Place 3rd Queen at 2nd column



(V) Now, there is no valid place for queen 4. So we needed to backtrack all the moves by popping the column index from stack to try out other possible places.

(vi) finally the possible solution can be



7) What is circular queue. Explain advantages of circular queue over linear queue.

⇒ Circular queue:

- It is a linear data structure in which operations are carried out on the basis of FIFO (First in first out) principle and the last position is connected back to the first position to make a circle.

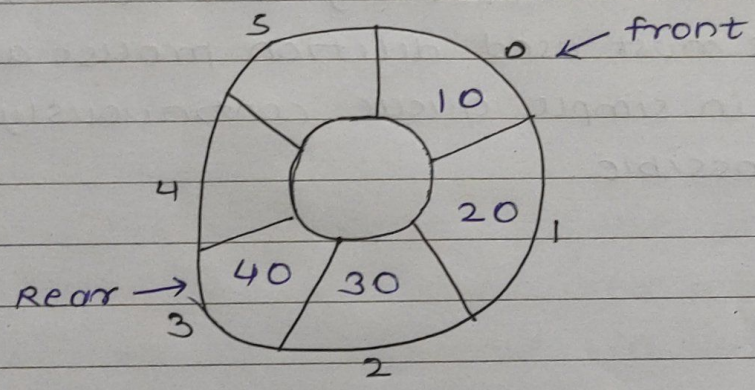


Fig. Circular Queue

- It operates much like regular queue, but with key difference: when the rear of the queue reaches the end, it wraps around to the front. This circular structure allows efficient use of space and avoids the limitations of a fixed size linear queue.
- Here when rear reaches at size-1 position, it is not considered that queue is full. Only in two situations circular queue is considered as full:
  (i) front == 0 && rear == size-1
  (ii) front > rear

* Following are some advantages of using circular queue over a linear queue:

1.> In circular queue we can insert new item to the location from where previous item is deleted.

2.> No memory wastage.

3.> In circular queue we can insert n numbers of elements continuously but condition is that we must used deletion process as well. Whereas in simple queue continuously inseretion is not possible.

q) Explain Application of different type of queue.

⇒ Applications of queue are as follows :-

(i) Requests processing on a single resource which is of shared manner, such as a printer and task scheduling of CPU, etc.

Linear queue

(ii) In real life scenario, we can consider the call centre phone systems where there is use of queue concept. People are attended in an order. First call gets first priority and others are on hold until a service representative becomes free.

Circular queue

(iii) In real-time systems, the various types of interrupts which occured are handled in the exact sequence in which they arrive ie. First come first serve.

(iv) When there is data transferred in an asynchronous way between two different types of process. For eg. IO Buffers, pipes, fifo, TIO, etc.

(v) priority queue applications include task scheduling in Operating system, Dijkstra's algorithm in graph theory, and Huffman coding in compression.

circular queue

Page No.
Date

Q4. Write pseudo code to implement ~~linked~~ circular queue using linked list.

→

1) Insert Node
2) Delete Node
3) Displaying Data

1) Insert:

```
void insert()
{
    node *newnode;
    newnode=(node*)malloc(sizeof(node));
    cout<<" Enter the node value";
    cin>>newnode->data;
    newnode->next=NULL;

if(rear==NULL)
    front=rear=newnode;
else
{
    rear->next=newnode;
    rear=newnode;
}
    rear->next=front;
}
```

2) Delete:

```
void del()
{
    temp = front;
  if (front == NULL)
    cout<<"Underflow";
  else
  {
      if (front == rear)
      {
       front = rear = NULL;
      }
      else
      {
          front = front->next;
          rear->next = front;
      }
     cout<<" Node deleted : "<<front->data;
      temp->next = NULL;
      free(temp);
     }
}
```

3] Display:

```
void display()
{
    temp = front;
    if (front == NULL)
        cout<<" Linked List is empty";
    else
    {
    for (; temp != rear; temp = temp->next)
    {
        cout << tem->data << "    ";
    }
     cout<< temp->data;
```

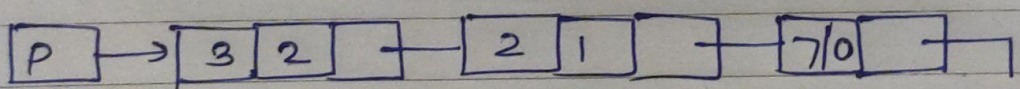
POLYNOMIAL ADDITION.

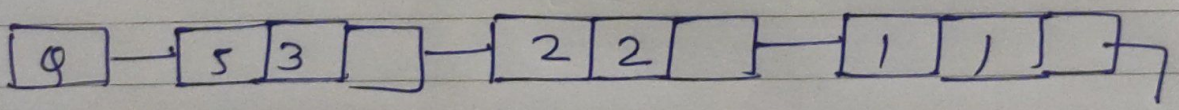
Algorithm:

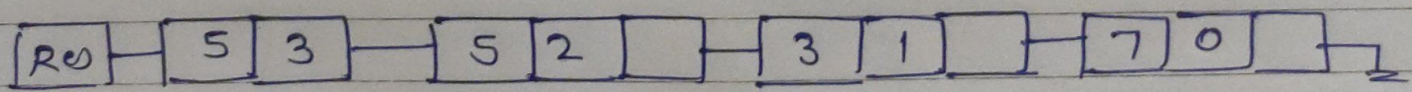
1. Read no. of terms in first polynomial 'P'.
2. Read no. of coefficient and exponent in first polynomial.
3. Read no. of terms in second polynomial 'Q'.
4. Read coefficient and exponent in second polynomial.
5. set temp pointer 'p' and 'q' to traverse the two polynomial respectively.
6. compare the exponents of two polynomials starting from first node.
   (i) If both exponents are equal then add coefficients and store in result linked list [p = p+1 & q = q+1]
   (ii) If exponent p less than exponent of q then add terms of q in result and move q to point next node
   (iii) If exponent of p > exponent of q, then add p term in the result and increase p to p+1.

$P = 3x^2 + 2x + 7$

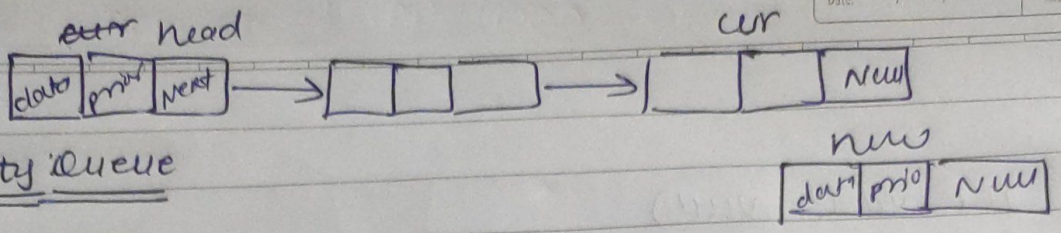
$Q = 5x^3 + 2x^2 + x$







Res: $5x^3 + 5x^2 + 3x + 7$



Priority Queue

① create:

```
struct Node
{
    int data;
    int priority;
    Node* next;
};
```

(struct Node { ... }; — structure creation)

② enqueue:

```
void enqueue(int ele, int priority)
{
    Node* new = new Node;
    new->data = ele;
    new->priority = priority;
    new->next = NULL;

    Node* cur = head;
    while (cur->next != NULL && priority <=
                                cur->next->priority)
    {
        cur = current->next;
    }
    new->next = cur->next;
    cur->next = new;
}
```

(Node* new = new Node; new->data = ele; new->priority = priority; new->next = NULL; — create new node & initialize)

(while (cur->next != NULL && priority <= cur->next->priority) { cur = current->next; } — traverse to last node)

③ dequeue :

```
void dequeue()
{
    if (head != NULL)
    {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

deletion

Polynomial Addition

Create:

```
struct Node
{
    int coe;
    int exp;
    Node* next;
}
```

function to add:

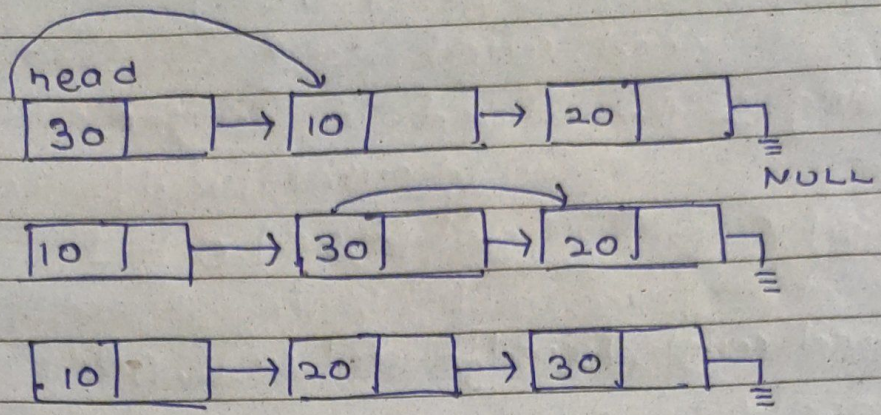
```
Node* addPoly(Node* poly1, Node* poly2)
{
    Node *ress = NULL;

    while(poly1 != nullptr || poly2 != nullptr)
    {
        Node* new = new Node;
        new->next = nullptr;

        if(poly1->exp < poly2->exp)
        {
            new->coe = poly2->coe;
            new->exp = poly2->exp;
            poly2 = poly2->next;
        } else if(poly1->exp > poly2->exp)
        {
            new->coe = poly1->coe;
            new->exp = poly1->exp;
            poly1 = poly1->next;
        }
```

```
        //Add
else
{
        new->coe = poly1->coe + poly2->coe;
        new->exp = poly1->exp;
        poly1 = poly1->next;
        poly2 = poly2->next;
}

} //attach new node to res
}

        res = new;
        cur = new;

}
}
```

void display

```
void display (Node *poly)
{
    while (poly != null)
    {
        cout << poly->coe << " x^ " << poly->exp
        cout << " + ";
        poly = poly->next;

        if (poly != null)
        {
            cout << " + ";
        }
    }
}
```

* Sort element in singly linked list:-



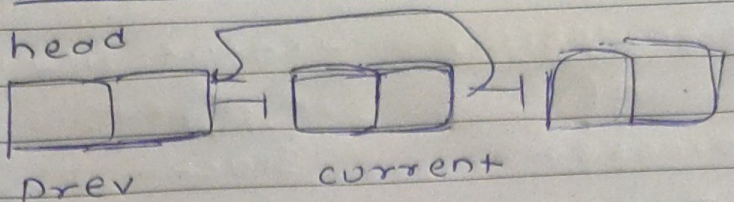
Bubble sort

```
Struct node *t1;
Struct node *t2;

For (t1 = head; t1 != NULL; t1 = t1->next)
{
   For(t2 = head; t2 != NULL; t2 -> t2 -> next)
   {
      IF (t2->data > t2->next->data)
      {

         temp = t2->data;
         t2->data = t2->data->next;
         t2->data->next = temp;

      }
   }
}
```

Rainbow

PAGE:
DATE: / /

*Reverse singly linked list




```
IF (head != NULL)
{
    head = Prev;
    head = head -> Next;
    curr = head;
    Prev -> next = NULL;
}

WHILE (head != NULL)
{
    head = head -> next;
    current -> next = Prev;
    Prev = current;
    curr = head;
}
```