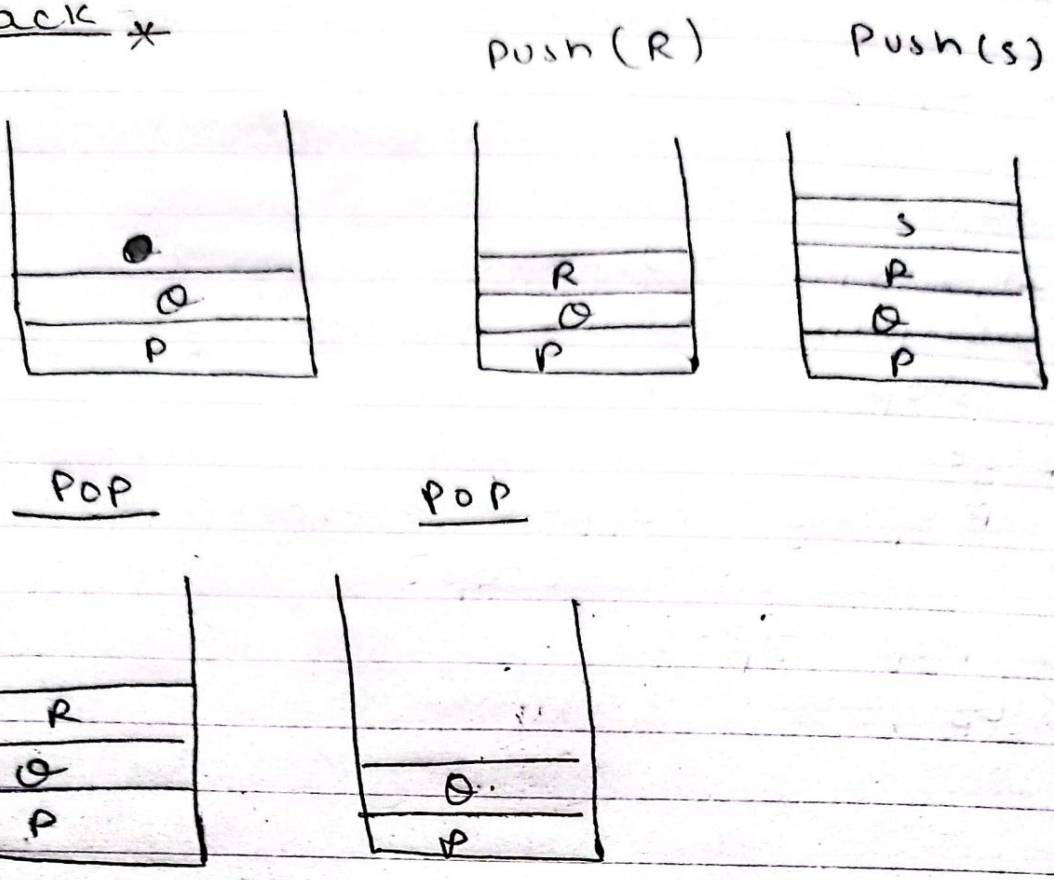
Stack

* Stack *



Algorithm to:

1) Push an element in the stack.

Step 1: Start
Step 2: IF (top == max-1)
Step 3: Display stack is ~~empty~~ full
~~Step 4~~: Exit
Step 4: ELSE
Step 5: Read an element / data.
Step 6: top ++
Step 6: set stack [top] = data
Step 7: Stop.

b) Delete an element from the stack (pop)

Step 1: Start
Step 2: If (top == -1)
    i) Display "Stack is empty"
    ii) stop
Step 3: ELSE.
    i) Display "Popped element is stack[top]
    ii) top = top - 1.
Step 4: Stop

c) Peek

Step 1: Start
Step 2: If (top == -1)
    i) Display "Stack is empty"
    ii) stop
Step 3: ELSE
    i) Display "Top element is stack[top]".
    ii) Display "Index of top is top".
Step 4: Stop.

Display all elements in the stack.

step 1: start
step 2: if (top == -1)
i) Display "stack is empty"
ii) stop
step 3: else
i) for (i = top; i >= 0; i--)
a) Display stack[i];

ii) End of loop.
step 4: stop.

Queue

Queue

→ Queue is linear data structure that follows the First In First Out Principle.

Deleting position is called front and inserting position is called rear.

Types of Queue

1) Linear Queue or simple Queue.
2) Circular Queue
3) Double ended queue
4) Priority queue.

Enqueue: Inserting an element. From rear
Dequeue: Deleting an element from front
Total no. of elements in the queue
= rear - front + 1

i) Linear Queue or simple Queue

Inserting an element

Step 1: start
Step 2: IF (rear == max-1)
    i) Display "Queue is full"
    ii) stop
Step 3: ELSE
    i) Read a data
    ii) rear ++
    iii) queue [rear] = data
Step 4: IF (front == -1)
    i) front = front + 1.
Step 5: stop.

Deleting an element

Step 1: start
Step 2: IF (rear == -1 and front == -1) or (rear<fro
    i) Display "Queue is empty"
    ~~ii) stop~~ ii) rear = -1, front = -1
    iii) stop
Step 3:
    ELSE
    i) "Delered element is queue [front]
    is displayed
    ii) front = front + 1;
Step 4: stop

2) Circular Queue

Circular queue is a linear data structure in which operations are performed based on FIFO and the last position is connected to first position to make a circle.

While inserting:

rear = (rear + 1) % max.

While deleting

front = (front + 1) % max.

a) Inserting an element.

Step 1: Start

Step 2: IF (front == 0 and rear == max - 1)

OR (rear == front - 1)

i) Display "Queue is Full"

ii) Stop

Step 3: ELSE

i) Read data

ii) rear = (rear + 1) % max

iii) queue[rear] = data

step 4: ~~Stop~~ IF (front == -1)

i) front = 0

Step 5: Stop

b) Deleting an element

Step 1: Start
Step 2: IF (front == -1 and rear == -1)
    i) Display "Underflow"
    ii) Stop
Step 3: ELSE IF (rear == front)
    i) Display "Deleted element is queue [front]".
    ii) rear = -1, front = -1.
Step 4: ELSE
    i) Display "Deleted element is queue [front]".
    ii) front = (front + 1) % max
Step 5: Stop

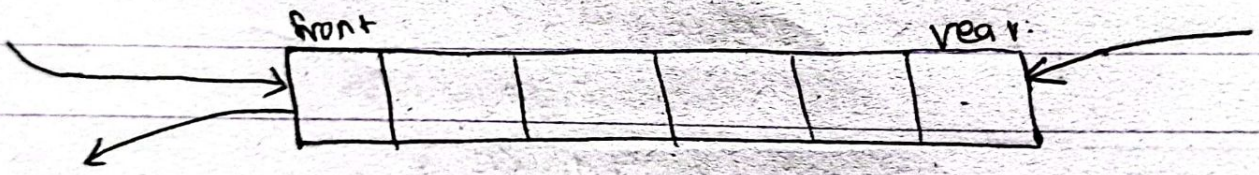
3) DEQUES (Double ended queue)

→ Insertion and deletion can be performe from both ends, Hence, it doesn't follow the FIFO rule.

Types of DEQUES

a) Input Restricted deque

b) Output Restricted deque



Possible Operation on DEQUES

a) Insertion at Front
b) Insertion at rear
c) Deletion from front
d) Deletion from rear

a) Inserting an element from rear end.

Step 1: start
Step 2: If (front == 0 and rear == max-1)
OR (front == rear + 1)
i) Display "Overflow"
ii) Stop
Step 3: ELSE IF (rear == -1 and front == -1)
i) Read data
~~ii) queue[rea~~
ii) rear = 0, front = 0
iii) queue[rear] = data
Step 4: ELSE
i) If (rear != max-1)
a) rear = rear + 1.
b) Read data
c) queue[rear] = data
d) Stop
ii) ELSE
// Shifting data of front and rear by 1
// step back.
a) for (i = front ; i <= rear; i++)
i) queue[i-1] = queue[i]
b) queue[rear] = data
c) front = front - 1
Step 5: Stop.

b) Inserting an element from front.

Step 1: Start
Step 2: IF (front == 0 and rear == max-1
OR (front == rear+1)
i) Display "overflow"
ii) stop
Step 3: ELSE IF (rear == -1 and front == -1
i) Read data
ii) rear = 0, front = 0
iii) queue [front] = data.
Step 4: ELSE
i) if (front != 0)
a) front = front - 1
b) Read data
c) queue [front] = data
ii) ELSE
a) Read data
b) // shifting rear's and front
// data ahead by 1 step
b) for (i = rear; i >= 0; i--)
i) queue [i+1] = queue [i]
c) queue [front] = data
d) rear = rear + 1.

Step 5: Stop.

c) Deleting from rear end

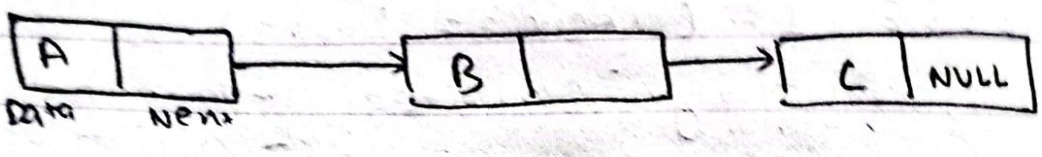
Step 1: Start
Step 2: IF (rear == -1 and front == -1)
i) Display "underflow"
ii) Stop.
Step 3: ELSE IF (rear == front)
i) Display "Deleted element is queue[rear]."
ii) Set rear = -1, front = -1.
Step 4: ELSE.
i) "Deleted element is queue[rear]" is displayed.
ii) rear = rear - 1.
Step 5: Stop.

d) Deleting from front end

Step 1: Start
Step 2: IF (rear == -1 and front == -1)
i) Display "underflow"
ii) Stop
Step 3: ELSE IF (rear == front)
i) Display "queue [front]"
ii) set rear == -1, front == -1
Step 4: ELSE
i) Display queue [front].
ii) front = front + 1.
Step 5: Stop.

List

List is an ordered set consisting of no. of elements to which addition, deletion operation can be done on the elemen



Static List

Algorithm to insert an element in List

Step 1: Start
Step 2: Read data, position (pos).
Step 3: [resize size of array] size = size + 1.
Step 4: set i = size - 1
Step 5: for i = size - 1 to i >= pos - 1
    i) avr[i + 1] = avr[i]
    ii) i = i - 1
    iii) (End of loop)
Step 6: avr[pos - 1] = data
Step 7: Stop.

Dynamic List (Linked list)

→ A linked list is a linear collection of specially designed data structure, called nodes, linked to one another by means of pointer.

→ Each node is divided into 2 parts: data and address part. The address part holds the address of next node.

Representation

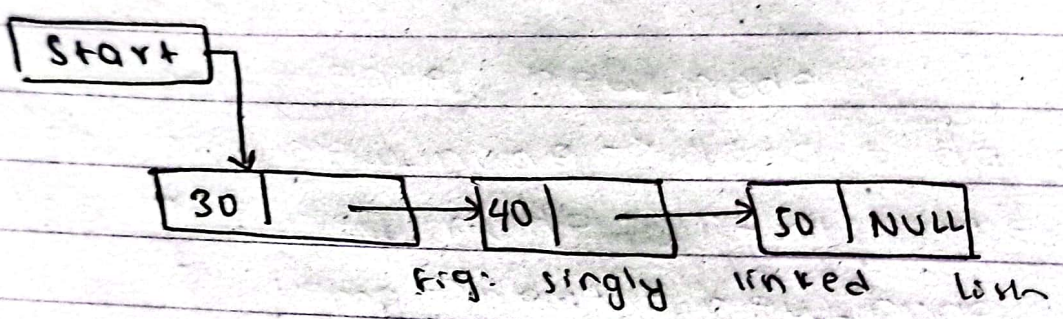
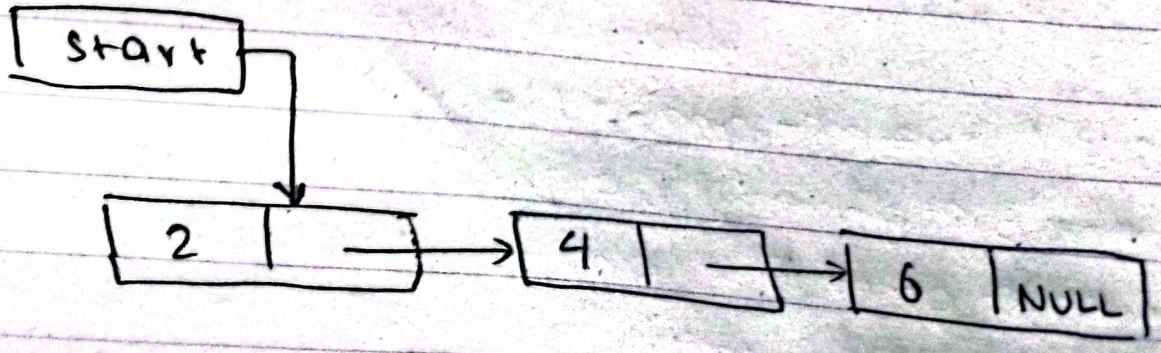


Fig: singly linked list

Types of Linked List

a) Singly Linked List
b) Doubly linked list
c) Circular linked list

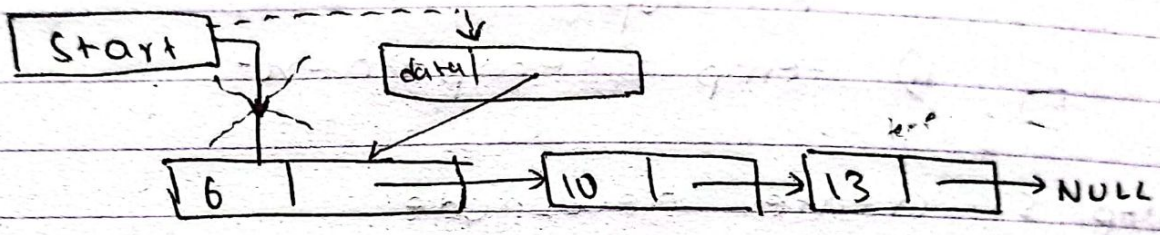
a) Singly linked list



Algorithm to

1) Insert an element at the beginning

Step 1: Start
Step 2: create new_node
Step 3: new_node -> data = data
Step 4: new_node -> next = start
Step 5: Start = new_node
Step 6: Exit



2) Insert an element at the end

Step 1: start
Step 2: create new_node
Step 3: new_node -> data = data
Step 4: new_node -> next = null
Step 5: if (start == null)
i) start = new_node
ii) Stop
Step 6: ELSE
i) temp = start
ii) while (temp -> next != null)
{ a) temp = temp -> next
}
iii) temp -> next = new_node
Step 7: Stop

3) Inserting at a specified position:

Step 1: start
Step 2: Read pos, data
Step 3: ere temp = start, i = 1
Step 4: for (i = 1; i < pos-1; i++)
{
~~i) temp = temp -> next~~
i) if (temp -> next == null)
a) Display "position not found"
b) stop
ii) temp = temp -> next
}
Step 5: Create new_node
Step 6: new_node -> data = data
Step 7: new_node -> next = temp -> next
Step 8: temp -> next = new_node
Step 9: Stop

4) Algorithm to display all nodes

Step 1: start
Step 2: if (start == NULL)
i) Display "empty" ii) stop
Step 3: temp = start
Step 4: while (temp -> next != null)
i) Display temp -> data
ii) temp = temp -> next
Step 5: Display temp -> data
Step 6: stop

Algorithm to delete an element from

a) Beginning

Step 1: Start
Step 2: If (start == NULL)
i) Display "List is Empty".
ii) Stop
Step 3: temp = start
Step 4: start = temp -> next
Step 5: temp -> next = null
Step 6: free (temp)
Step 7: Stop

b) Last

Step 1: Start
Step 2: If (start == NULL)
i) Display "Empty"
ii) Stop
Step 3: temp = start
Step 4: while (temp -> next != null)
{
i) temp2 = temp
ii) temp = temp -> next
}
Step 5: temp2 -> next = null
Step 6: Free (temp)
Step 7: Stop

c) Specified position.

```
Step 1: start
Step 2: Read position (pos
Step 3: temp = start
Step 4: ~~while (temp -> next~~
        for(i=1; i< pos ; i++)
        {
            i) temp2 = temp
            ii) temp = temp -> next.
            iii) if(temp2 -> next = NULL)
                {a) Display " Pos not found
                 b) stop
                }
        }
Step 5: temp2 -> next = temp -> next
Step 6: temp -> next = NULL
Step 7: free (temp)
Step 8: Stop
```

Algorithm to search node

```
Step 1: Start
Step 2: temp = start, pos = 1,
Step 3: Read data.
Step 4: while (temp -> next != NULL)
        {
            i) if(temp -> data == data)
            {
```

a) Display "Data is at pos".
b) Stop

}

ii) temp = temp → next.

iii) pos = pos + 1

}

Step 5: If (temp == NULL)

{

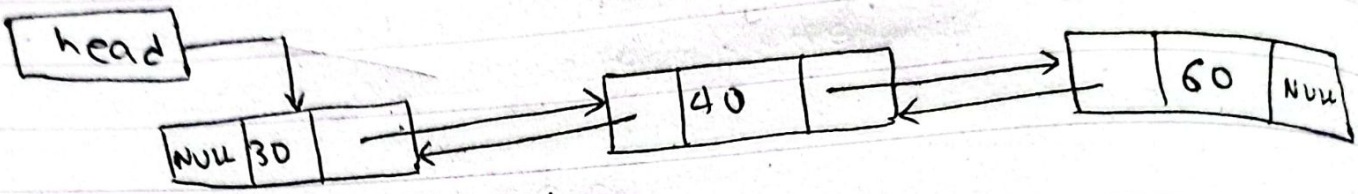
i) Display "~~Position~~ Data not found".

}

Step 6: Stop

b) Doubly linked List

A doubly linked list is one in which all the nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list.



Algorithm to add an element

a) At the beginning

Step 1: Start
Step 2: create new_node
Step 3: new_node -> data = data,
new_node -> prev = NULL
Step 4: If (head == NULL)
{ i) new_node -> next = NULL
ii) head = new_node
}
Step 5: ELSE
i) new_node -> next = head
ii) head -> prev = new_node
iii) head = new_node.
Step 6: Stop.

b) At the end

Step 1: Start
step 2: create new_node
step 3: new_node -> next = NULL
new_node -> data = data.
step 4: if (head == NULL)
i) new_node -> prev = NULL
ii) head = new_node
Step 5: ELSE
i) temp = head
ii) while (temp -> next != NULL)
a) temp = temp -> next.
(End of loop)
iii) temp -> next = new_node
iv) new_node -> prev = temp
step 6: Stop

At specified position

Step 1: Start
step 2: Read pos, data
step 3: create new_node
step 4: new_node -> data = data, i = 1
step 5: temp = head
step 6: while (i < pos-1)
i) temp = temp -> next., i++
ii) IF (temp -> next == NULL)
a) Display "Pos not found".
b) stop:

iii) End of Loop.

Step 7: new_node -> next = temp -> next
new_node -> prev = temp
(temp -> next) -> prev = new_node
(temp -> next) = new_node

Step 8 Stop

Algorithm to Delete a node

a) From the beginning

Step 1: Start
Step 2: If (head == NULL)
i) Display "Underflow"
ii) Stop.
Step 3: temp = head.
Step 4: (temp -> next) -> prev = NULL
Step 5: head = temp -> next
Step 6: temp -> next = NULL
Step 7: free temp
Step 8: Stop.

b) From the end

Step 1: Start
Step 2: If (head == NULL)
i) Display "Underflow"
ii) Stop

Step 3: temp = head
Step 4: while (temp -> next != NULL)
    i) temp = temp -> next
    (End of loop)
Step 5: (temp -> prev) -> next = NULL
Step 6: temp -> prev = NULL
Step 7: Free temp
Step 8: Stop.

~~Algo~~:

c) From specified position.
Step 1: Start.
Step 2: Read pos
Step 3: If (head == NULL)
    i) Display "Underflow".
    ii) Stop.
Step 4: temp = head, i = 1
Step 5: while (i < pos)
    i) temp = temp -> next
    ii) i++;
    iii) if (temp -> next == NULL)
        a) Display "Pos not found".
        b) Stop.
    (End of loop)
Step 6: (temp -> prev) -> next = temp -> next
Step 7: (temp -> next) -> prev = temp -> prev.
Step 8: free temp
Step 9: Stop.

3) Circular linked list

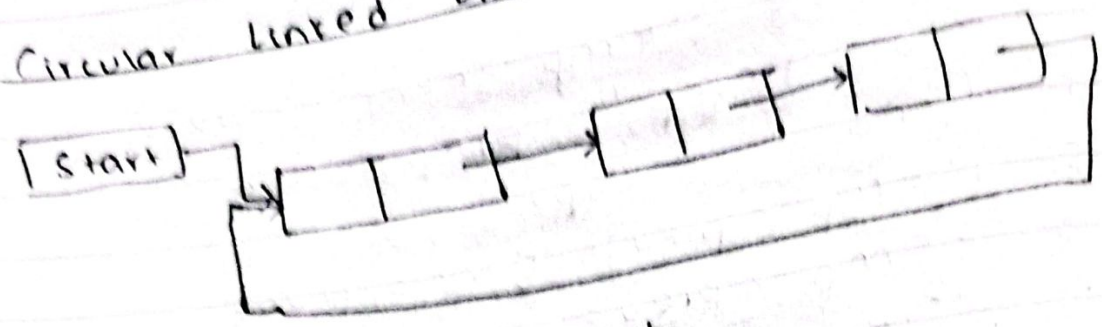


Fig: circular linked list.

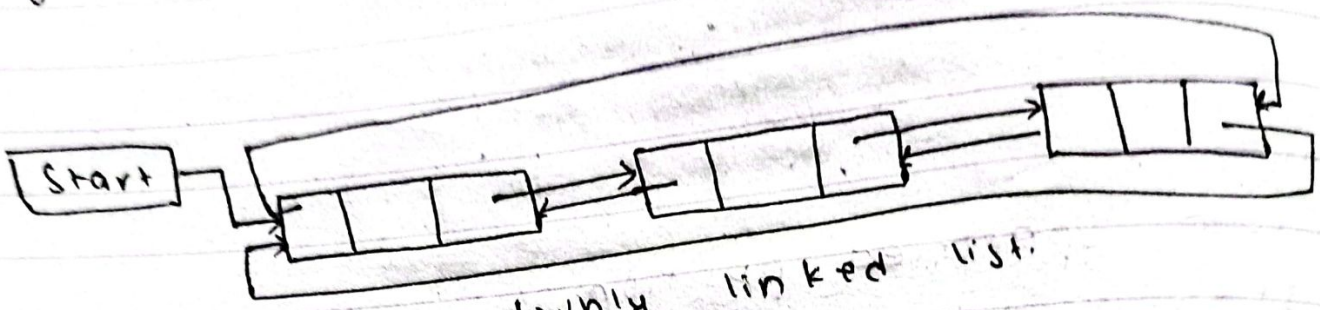


Fig: circular doubly linked list.

Circular linked list Algorithm

a) Inserting a node at beginning.

Step 1: Start
Step 2: If (start == null)
    i) new_node -> data = data
    ii) new_node -> next = new_node
    iii) start = new_node
    iv) last = new_node
    v) stop
Step 3: new_node -> data = data
Step 4: new_node -> next = start
Step 5: start = new_node
Step 6: last -> next = new_node.
Step 7: Stop.

b) Inserting at the End

step 1: Start
step 2: create new_node
step 3: If (start == null)
  i) new_node → data = data
  ii) new_node → next = new_node
  iii) start = new_node
  iv) last = new_node
  v) stop
step 4: ELSE
  i) new_node → data = data
  ii) last → next = new_node
  iii) last = new_node
  iv) last → next = start.
step 5: Stop.

c) Deleting from beginning

step 1: Start.
step 2: If (start == null)
  i) Display "Empty".
  ii) Stop.
step 3: temp = start
step 4: start = start → next.
step 5: last → next = start.
step 6: free (temp)
step 7: Stop.

d) Deleting from the end

Step 1: start

Step 2: if (start == null)

i) Display "empty"

ii) stop.

Step 3: ELSE

i) temp = start

~~ii) while(ptr!=~~

ii) while (temp != = last)

a) temp = temp -> next

iii) temp -> next = start.

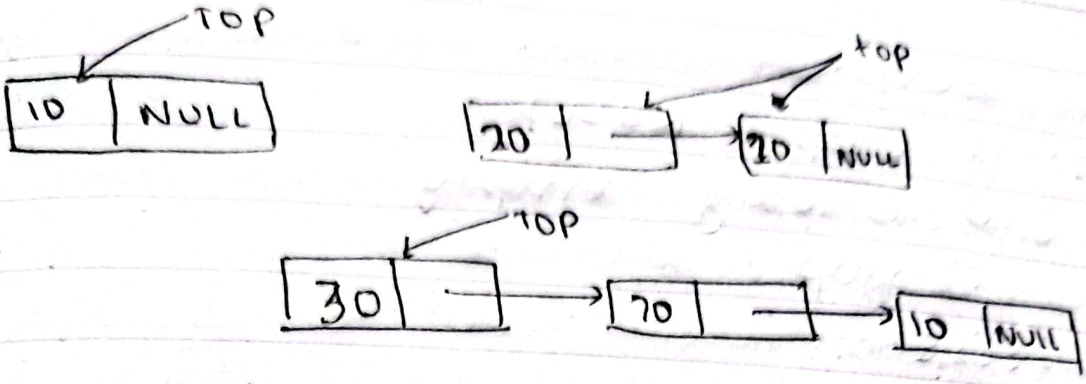
iv) ptr = last

v) last = temp

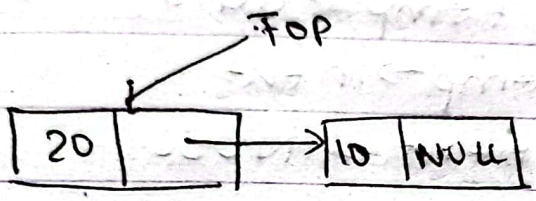
vi) free (ptr).

Step 4: stop.

Stack using Linked list



push



popped element is 30.

Algorithm to Push

step 1: start

step 2: create new_node

step 3: new_node → data = data

step 4: if (top == NULL)

i) top = new_node

ii) new_node → next = NULL

step 5: ELSE

i) new_node → next = top

ii) top = new_node

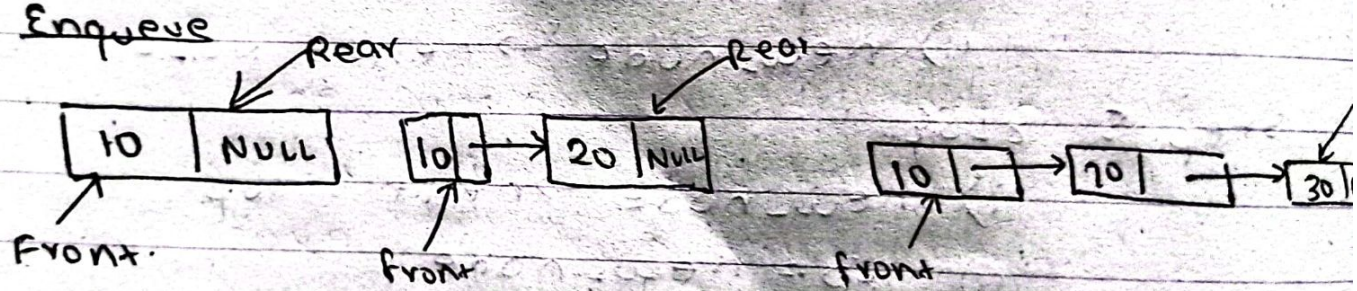
step 6: stop.

Algorithm to POP

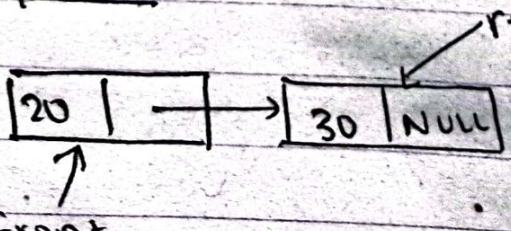
Step 1: Start
Step 2: If (top == NULL)
  i) Display "empty"
  ii) Stop
Step 3: ELSE
  i) temp = top
  ii) Display temp → data
  iii) top = temp → next
  iv) temp → next = NULL
  v) Free (temp)
Step 4: Stop.

Queue Using Linked List

Enqueue



Dequeue



dequeued element is

Algorithm to Enqueue

Step 1: start
Step 2: create new_node, new_node->next = NULL
Step 3: new_node -> data = data
Step 4: If (rear == NULL && front == NULL)
    i) rear = new_node
    ii) front = new_node
Step 6: ELSE If (rear != NULL)
    i) rear -> next = new_node
~~Step 7:~~ ii) rear = new_node
Step 7: Stop

Algorithm to Dequeue

Step 1: Start
Step 2: If (front==null && rear==null)
    i) Display "Empty".
    ii) Stop
Step 3: ELSE
    i) temp = front
    ii) front = temp -> next
    iii) temp -> next = null
    iv) free temp
Step 4: Stop

Stack

Algorithm to convert from infix to postfix:

1) Add a unique symbol # into stack and at end of array infix.
2) Scan the symbol from left to right of array infix.
3) If the symbol is parenthesis '(' then add into the stack
4) If symbol is operand then add into array post fix.
5) i) If symbol is operator then pop the operator which have same precedence or higher precedence than the operator which occurred.
   ii) Add the popped operator to array post fix.
   iii) Add the scanned symbol operator into stack.
6) If the symbol is ')' then pop all operators from stack untill '(' is occurred.
   ii) Remove '(' from stack.

If symbol is # then pop all the symbol from stack and then add to array post fix except #.

Do the same process until # comes in scanning array infix.

1.Q) Convert A + B - C / D * E into postfix exp.

⇒ A + B - C / D * E #

Scanned Symbol	Stack	Array Postfix
1) A	#	A
2) +	# +	A
3) B	# +	AB
4) -	# -	AB +
5) C	# -	AB + C
6) /	# - /	AB + C
7) D	# - /	AB + CD
8) *	# - *	AB + CD /
9) E	# - *	AB + CD / E
10 #	—	AB + CD / E * -

∴ Required postfix exp. is AB + CD / E * -

(Q.2) K + L - M * N + (O ^ P) * W / U / V * T + Q. #

	Scanned Symbol	Stack	Array Postfix
1)	K	#	K
2)	+	#+	K
3)	L	#+	KL
4)	-	#-	KL+
5)	M	#-	KL+M
6)	*	#-*	KL+M
7)	N	#-*	KL+MN
8)	+	#+	KL+MN*-
9)	(	#+(	KL+MN*-
10)	O	#+(	KL+MN*-O
11)	^	#+(^	KL+MN*-O
12)	P	#+(^	KL+MN*-OP
13)	)	#+	KL+MN*-OP^
14	*	#+*	KL+MN*-OP^
15	W	#+*	KL+MN*-OP^W
16	/	#+/	KL+MN*-OP^W*
17	U	#+/	KL+MN*-OP^W*U
18	/	#+/	KL+MN*-OP^W*U/
19	V	#+/	KL+MN*-OP^W*U/V
20	*	#+*	KL+MN*-OP^W*U/V/
21	T	#+*	KL+MN*-OP^W*U/V/T
22	+	#+	KL+MN*-OP^W*U/V/T*+
23	Q	#+	KL+MN*-OP^W*U/V/T*+
24	#	—	KL+MN*-OP^W*U/V/T*+

Conversion from infix to Prefix

Q.) Convert A + B * C + D into prefix.

Reverse = D + C * B + A #

Scanned Symbol	Stack	expression
i) D	#	D
ii) +	# +	D
iii) C	# +	DC
iv) *	# + *	DC
v) B	# + *	DCB
vi) +	# + +	DCB *
vii) A	# + +	DCB * A
viii) #	———	DCB * A + +

∴ Prefix expression = + + A * B C D

Q)2. (A+B) + (- (D-E) ^ F.

-)

Reverse : F ^) E - D (- C +) B + A (#

Scanned Symbol	Stack	Expression
1) F	#	F
2) ^	#^	F
3))	#^)	F
4) E	#^)	FE
5) -	#^)-	FE
6) D	#^)-	FED
7) (	#^	FED-
8) -	#-	FED-^
9) C	#-	FED-^C
10) +	#-+	FED-^C
11))	#-+)	FED-^C
12) B	#-+)	FED-^CB
13) +	#-+)+	FED-^CB
14) A	#-+)+	FED-^CBA
15) (	#-+	FED-^CBA+
16) #	—	FED-^CBA++-

∴ Prefix expression = - ++ ABC ^ - DEF

Checked