

Introduction to Data Structures

① Algorithmic Specification

An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statements in simple English language.

Ex:-

Addition of two numbers and store the result in a third variable.

Step ① : Start

Step ② : Read the first number 'is variable 'a'.

Step ③ : Read the second number 'in variable 'b'.

Step ④ : Perform the addition of both the numbers i.e. and store the result in variable 'c'.

Step ⑤ : Print the value of 'c' as a result of addition.

Step ⑥ : Stop.

Characteristics of Algorithm

- 1) Each algorithm is supplied with zero or more inputs.
- 2) Each algorithm must produce at least one output.
- 3) Each algorithm should have definiteness i.e. each instruction must be clear and unambiguous.
- 4) Each algorithm should have finiteness i.e. if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after finite number of steps.

Abstract Data Types (ADT) :-

The abstract data type is a tuple of D-Set of domains, F-Set of functions, A - Axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

In ADT, all the implementation details are hidden.

ADT = Type + function names + Behavior of each function.

ADT Data Structures :- The ADT Operations are Carried out with the help of data Structures. This part describes the structure of the data used in the ADT in an informal way. Various data Structures that can be used for ADT are Arrays, Set, Linked list, Stack, Queues and so on.

Ex:-

1) ADT for Set

If we want to write ADT for a set of integers, then we will use following method.

-Abstract Data Type Set

§ Instances : Set is a Collection of integer type of elements.

PreConditions : none

Operations :

- * Store () : This operation is for storing the integer element in a set.
- * Retrieve () : This operation is for retrieving the desired element from the given set.
- * Display () : This operation is for displaying the Contents of set.

2. ADT for Arrays

AbstractDataType Array

{

Instances : An array A of some size, index i and total number of elements in the array n .

Operations :

1. Store () : This operation is for storing the integer element in a set.
2. Retrieve () : This operation is for retrieving the desired element from the given set.
3. Display () : This operation is for displaying the contents of set.

}

① Introduction to Data Structures

The data structure can be defined as the Collection of ~~elements~~ ^{nodes} and all the possible operations which are required for those set of elements.

- * Reading
- * Printing
- * Searching

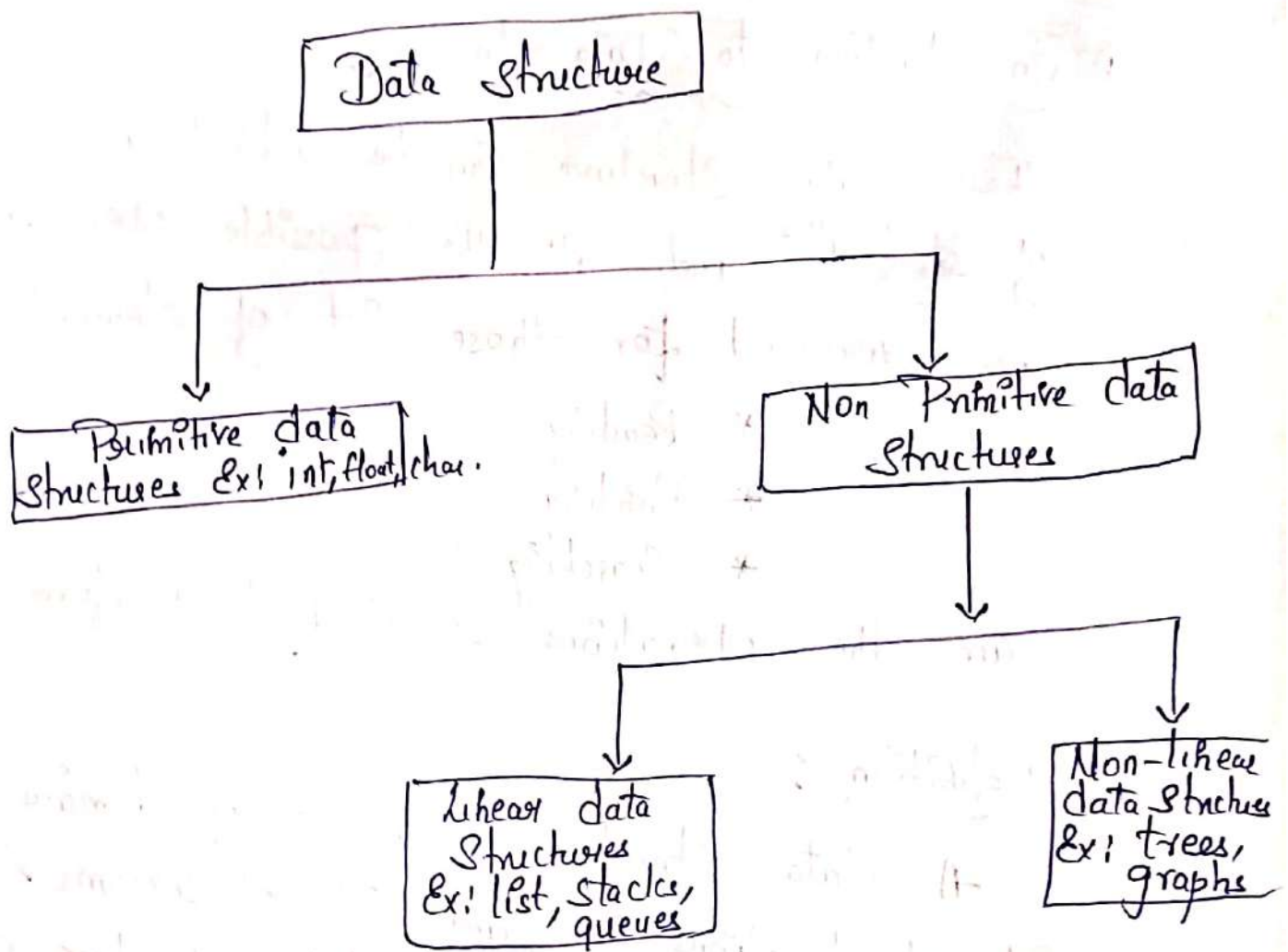
are the operations required to perform the task.

Definition :

A data structure is a set of domains D , a set of functions F and set of axioms A . This triple (D, F, A) denotes the data structure d .

Types of Data Structures

The data structures can be divided into two basic types Preliminary data structure and Secondary data structure.



* Linear data Structures are the data Structures in which data is arranged in a list or in a straight sequence.

Ex:- Arrays, list

* Non linear data Structures are the data Structures in which data may be arranged in hierarchical manner.

Ex:- Trees, Graphs.

Here, we discuss both the linear data structures such as Arrays, Linked lists, Stacks, Queues. And non-linear data structures such as Trees and Graphs.

Primitive and Non Primitive Data Structures.

- * The Primitive data types are the fundamental data types. The non primitive data types can be built using primitive data types.

Ex:- int, float, double, char.

- * The Non-Primitive data types are user defined data types.

Ex:- Structure, Union and enumerated

Linear and Non-Linear Data Structure

Linear data Structure are the data structures in which data is arranged in a list or in straight sequence.

Ex:- Arrays, list.

Non-linear data Structure are the data structures in which data may be arranged in hierarchical manner.

Ex:- Trees, Graphs.

Static and Dynamic Data Structures

The static data structures are data structures having fixed size memory.

Ex:- Arrays

Arrays in C are a static data structure. We first allocate the size of array.

The dynamic data structures are data structures having the allocation of memory as per the user requirement. If we do not want to use some block of memory, we can deallocate it.

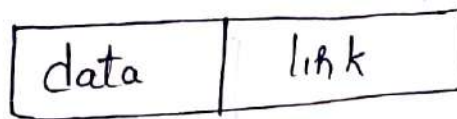
Ex:- linked list.

linked list is a collection of many nodes. Each node consists of data and pointer to next node.

Advantage of dynamic data structure over static data structure is there is no wastage of memory.

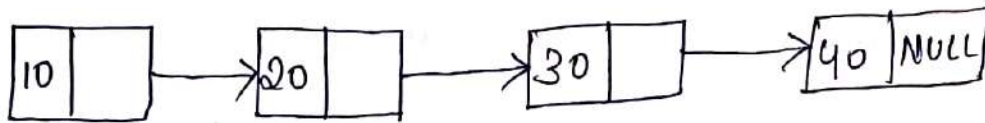
Singly linked list

A linked list is a set of nodes where each node has two fields 'data' and 'link/address'. The 'data' field stores actual piece of information and 'link' field is used to point to next node.



Structure of node

Hence, linked list of integers 10, 20, 30, 40 is



The 'link/address' field of last node consists of NULL which indicates end of list.

Representation of linked list

'C' structure

typedef struct node

{

int data; /* data field */

struct node * next; /* link field */

}

SLL;

Declaration in Structures Consists two members i.e data member And Pointer member.

Data member Can be character/integer/float.

Pointer member holds the address of next node.

Advantages & Disadvantages of linked list

Advantages	Disadvantages
1. (Linked Organization does not Support Random or direct access.) ^x	linked Organization does not Support random or direct access.
2. In linked Organization, insertion and deletion of elements can be efficiently done.	Each data field should be supported by a link field to point to next node.
3. There is no wastage of memory. The memory can be allocated and deallocated as per requirement.	

Memory Allocation and Deallocation for linked list

- * For allocating the memory in 'C' the malloc function is used. This function can be used if we include alloc.h file in our 'C' program.
- * Similarly for de-allocating the memory the free function is used. Free $\rightarrow$ function

`int *i;` $\rightarrow$ Pointer to integer variable

`float *f;` $\rightarrow$ Pointer to float variable

`char *c;` $\rightarrow$ Pointer to character type variable

`typedef struct Student`

`{`

`int roll_no;`

`char name[10];`

`} s;`

$s \rightarrow$ structure, $s1 \rightarrow$ pointer

`s * s1;`

`i = (int *) malloc (Size of (int));`

`f = (float *) malloc (Size of (float));`

`c = (char *) malloc (Size of (char));`

`s1 = (s *) malloc (Size of (s));`

`free (i)` $\rightarrow$ Memory is free/deallocated

We can also allocate the memory using calloc function.

The Syntax of calloc is

`Void * Calloc (Size_t num, Size_t size);`

where

num represents the number of elements to allocate.

Size_t $\rightarrow$ Unsigned integral type.

Size $\rightarrow$ Size of each element.

Ex:-

`int * Data;`

`Data = (int*) Calloc (n, Size of (int));`

where

n $\rightarrow$ number of items for which memory is allocated.

Size of (int) $\rightarrow$ Storing integer elements.

Difference between malloc() and calloc()

<u>malloc()</u>	<u>calloc()</u>
1. malloc takes only one element and that is size of the block to be allocated.	It takes two arguments. One is <u>no</u> of items to be allocated say " <u>n</u> ", and second argument is size of each item say <u>s</u> . Thus the function allocates $n \times s$ bytes of memory.
2. Malloc does not initialize the contents of memory that is been allocated.	Calloc initializes the memory after allocation.

linked list without using functions

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

```
struct node * n1; * n2; * n3;
```

```
n1 = (struct node *) malloc (sizeof(struct));
```

```
n1 → data = 10;
```

```
n1 → next = NULL;
```

```
n2 = (struct node *) malloc (sizeof(struct));
```

```
n2 → data = 20;
```

```
n2 → next = NULL;
```

```
n1 → next = n2;
```

```
n3 = (struct node *) malloc (sizeof(struct));
```

```
n3 → data = 30;
```

```
n3 → next = NULL;
```

```
n2 → next = n3;
```

```
n4 → data = 40;
```

```
n4 → next = NULL;
```

```
n1 → next = n4;
```

```
n4 → next = n2
```

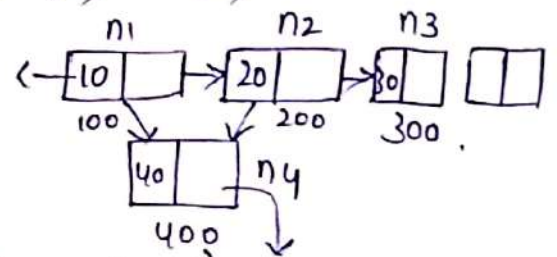
```
delete(n2) n4 → next = n3
```

```
free (n2);
```

```
Display() temp = n1;
```

```
while (temp != NULL)
```

```
{
```



[Here n1, n2, n3, n4 are nodes]

[10, 20, 30, 40 are data]

[next → link/address of next node]

[Cont...d]

[Cont--d]

8

```
printf (temp → data);  
temp = temp → next;  
printf ("Null");
```

Output :-

10 40 30.

Operations On linked list

Various operations of linked list are

- 1) Creation of linked list
- 2) Display of linked list
- 3) Insertion of any element in the linked list
- 4) Deletion of any element in the linked list
- 5) Searching of desired element in the linked list.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
```

```
struct node
{
    int data;
    struct node *next;
    * head = NULL;
} node;
```

```
struct node * Create ();
```

```
void main()
```

```
{
    int choice, val;
    char ans;
    node * head;
    void display (node *);
    node * Search (node *, int);
    node * insert (node *);
    void dele (node **);
    head = NULL;
```

```
do
```

```
{
    clrscr();
```

```
printf("\n Program to Perform Various operations  
On linked list");
```

```
Printf("\n1. Create");
```

```
Printf("\n2. Display");
```

```
Printf("\n3. Search for an item");
```

```
Printf("\n4. Insert an element in a list");
```

```
Printf("\n5. Delete an element from list");
```

```
Printf("\n6. Quit");
```

```
Printf("\n Enter Your choice (1-6)");
```

```
Scanf("%d", &choice);
```

```
Switch (choice)
```

```
{
```

```
Case 1: head = Create ();  
break;
```

```
Case 2: Display(head);  
break;
```

```
Case 3: Printf("Enter the element you want to  
search");
```

```
Scanf("%d", &val);
```

```
Search(head, val);
```

```
break;
```

```
Case 4: head = Insert(head);  
break;
```



```
Case 5: dele (&head);  
        break;
```

```
Case 6: exit (0);
```

```
default: clrscr ();
```

```
        printf ("Invalid Choice, Try again");
```

```
        getch ();
```

```
    }
```

```
    } while (choice != 6);
```

```
}
```

① → Struct node * Create ()

```
{
```

```
    node * temp, * New, * head;
```

```
    int val, flag;
```

```
    char ans = 'y';
```

```
    node * get_node ();
```

```
    temp = NULL;
```

```
    flag = True; /* flag to indicate whether  
                  a new node is created  
                  for the first time or not */
```

```
do
```

```
{
```

```
    printf ("\n Enter the element : ");
```

```
    scanf ("%d", &val);
```

[Cont...d]

```

/* allocate new node */
New = get_node ();
if (New == NULL)
    printf ("\n Memory is not allocated");
New → data = val;
if (flag == True) /* Executed only first time */
{
    head = New;
    temp = head; /* head is a first node */
    flag = false;
}
else
{
    /* temp keeps track of the most recently
       Created node */
    temp → next = New;
    temp = New;
}
printf ("\n Do you want to enter more elements?
        (y/n)");
ans = getch ();
while (ans == 'y')
    printf ("\n The Singly linked list is Created\n");
    getch ();

```

```
return head ;
```

```
}
```

```
node * get_node ( )
```

```
{
```

```
node * get_node ( )
```

```
{
```

```
node * temp;
```

```
temp = (node *) malloc (Size of (node));
```

```
temp → next = NULL;
```

```
return temp;
```

```
}
```

```
② → void display (node * head)
```

```
{
```

```
node * temp;
```

```
temp = head;
```

```
if (temp == NULL)
```

```
{
```

```
printf (" \n The list is Empty \n");
```

```
getch ( );
```

```
return;
```

```
}
```

```
while (temp != NULL)
```

```
{
```

```
printf ("%d →", temp → data);
```

```
temp = temp → next ;
```

```
}
```



```
Printf("NULL");  
getch();  
}
```

③ → node * Search (node * head, int key)

```
{  
    node * temp;  
    int found;  
    temp = head;  
    if (temp == NULL)
```

```
{  
    Printf("The linked list is Empty\n");  
    getch();  
    return NULL;  
}
```

```
    found = false;  
    while (temp != NULL && found == false)  
    {  
        if (temp->data != key)  
            temp = temp->next;  
        else  
            found = True;
```

```
    }  
    if (found == True)
```

```
{  
    Printf("\n The Element is present in the list\n");  
    getch();  
}
```

```

        return temp;
    }
    else
    {
        printf("The Element is not Present in the list\n");
        getch();
        return NULL;
    }
}

```

④ → node * insert (node * head)

```

{
    int choice;
    node * insert_head (node *);
    void insert_after (node *);
    void insert_last (node *);
    printf("\n 1. Insert a node as a head node");
    printf("\n 2. Insert a node as a last node");
    printf("\n 3. Insert a node at intermediate position\n");
    printf("in the linked list");
    scanf("%d", &choice);
    switch (choice)
    {
        Case 1: head = insert_head (head);
                break;
        Case 2: insert_last (head);
                break;
    }
}

```

Case 3: Insert - after (head);

```

    break;
}
return head;
}

```

[New head is returned from function insert_head hence we have to return the new head from the insert function to main]

⑤ → /* insertion of node at first position */

node * insert_head (node * head)

```

{
    node * New, * temp;
    New = get_node ();
    printf("\n Enter the element which you want to
           insert");

```

```

    scanf("%i", &New → data);

```

```

    if (head == NULL)
        head = New;

```

[No node in the linked list]

```

    else
    {

```

```


```

```

        temp = head;

```

```

        New → next = temp;

```

```

        head = New;
    }

```

```


```

```

    return head;
}

```

```

}

```


⑥ → /* insertion of node at last position */

void insert_last (node * head)

{
node * New, * temp;

New = get_node ();

Printf (" \n Enter the element which you want to
insert ");

scanf ("%d", & New → data);

if (head == NULL)

head = New;

else

{
temp = head;

while (temp → next != NULL)

temp = temp → next;

temp → next = New;

New → next = NULL;

}

}

⑦ → /* Insertion of node at intermediate position */

void insert_after (node * head)

{
int key;

node * New, * temp;

New = get_node ();

Printf (" \n Enter the element which you want to
insert ");

[Cont..d]

scanf("%d", &New → data);

if (head == NULL)

{

head = New;

}

else

{

printf("\n Enter the element after which you want to insert
the node");

scanf("%d", &key);

temp = head;

do

{

if (temp → data == key)

{

New → next = temp → next;

temp → next = New;

return;

}

else

temp = temp → next;

}

while (temp != NULL);

}

}

node * get_prev (node * head, int val)

{

node * temp, * prev;

```

        int flag ;
        temp = head;
        if (temp == NULL)
            return NULL;
        flag = false;
        prev = NULL;
        while (temp != NULL & !flag)
        {
            if (temp->data != val)
            {
                prev = temp;
                temp = temp->next;
            }
            else
                flag = True;
        }
        if (!flag) /* flag is true */
            return prev;
        else
            return NULL;
    }

```

⑧ → void dele (node **head)

```

{
    node *temp, *prev;
    int key;
    temp = *head;

```

[Cont... d]


```
if (temp == NULL)
```

```
{
```

```
    printf("\n The list is empty \n");
```

```
    getch();
```

```
    return;
```

```
}
```

```
printf("\n Enter the element you want to delete");
```

```
scanf("%i", &key);
```

```
temp = Search(*head, key);
```

```
if (temp != NULL)
```

```
{
```

```
    prev = get_prev(*head, key);
```

```
    if (prev != NULL)
```

```
{
```

```
        prev → next = temp → next;
```

```
        free(temp);
```

```
}
```

```
else
```

```
{
```

```
    *head = temp → next;
```

```
    free(temp);
```

```
}
```

```
printf("\n The element is deleted \n");
```

```
getch();
```

```
}
```

Output

Program to Perform Various operations on linked list .

1. Create
2. Display
3. Search for an item
4. Insert an Element
5. Delete an Element
6. Quit

Enter Your choice (1-6) 1

Enter element : 10

Do you want to enter more elements ? (y/n) y

Enter element : 20

Do you want to enter more elements ? (y/n) y

Enter element : 30

Do you want to enter more elements ? (y/n) n

Singly linked list is Created .

Enter your choice (1-6) 2

10 → 20 → 30

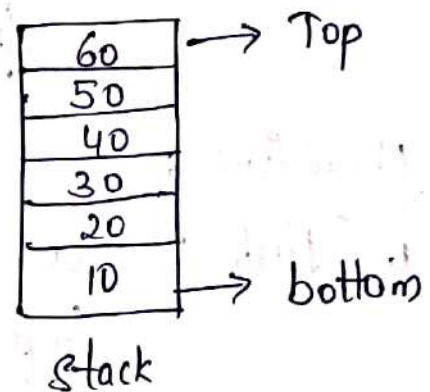
//

Stack

Stack :- A Stack is a Ordered list in which all insertions and deletions are made at one end, called top. If we have a Stack of elements 10, 20, 30, 40, 50, 60 then "10" will be the bottom most element and "60" will be top most element.

Ex:- CD Stack

Stack follows LIFO approach
(Last In First Out).



Stack ADT [Abstract Data Type]

Stack is a data structure which possesses LIFO.

The ADT for Stack is

Abstract Data Type Stack

§

Instances : Stack is a collection of elements in which insertion and deletion of elements is done by one end called top.

(Stfull $\rightarrow$ Stackfull, Stempty $\rightarrow$ Stack_{empty})

Pre conditions :

1. Stfull(): This indicates whether Stack is full or not. If stack is full we cannot insert the elements in stack.
2. Stempty(): This indicates whether stack is empty or not. If stack is empty we cannot pop/remove the elements from stack.

Operations :

1. Push : we can push elements onto the stack. before performing push check whether stack is full or not.
2. Pop : we can pop elements from the stack before performing pop check whether stack is empty or not.

Stack Operations

The Stack operations are

- 1) To Create a Stack
- 2) To insert an element on to the stack.
- 3) To delete an element from the stack.
- 4) To check which element is at the top of the Stack.
- 5) To check whether stack is empty or not.

Array Implementation in Stacks

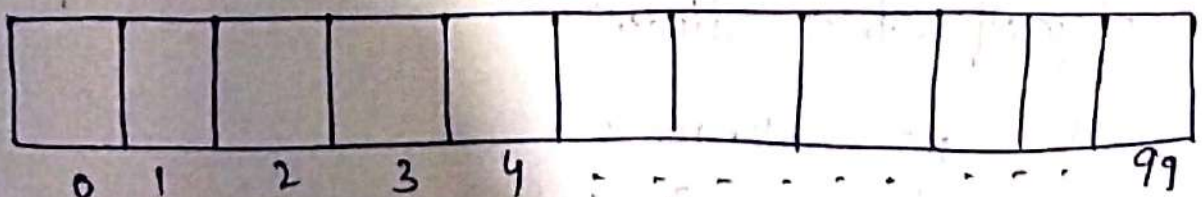
Declaration 1 :

```
#define Size 100
```

```
int Stack[Size], top = -1;
```

In the above declaration Stack is nothing but an array of integers. And most recent index of that array will act as a top.

Stack with one dimensional array



Initially top = -1

[Cont.. d]

The Stack is of size 100. As we insert the numbers, the top will get incremented. The elements will be placed from 0th position in the Stack. At most we can place 100 elements in Stack, then most last element will be (Size - 1) position, (ie) at index 99.

Declaration 2:

Stack using Structure

define Size 10

struct Stack

{

int s[Size];

int top;

} st;

top points

top s	
9	100
⋮	⋮
2	30
1	20
0	10

Value in Stack.

Stack using structure

Here Stack is declared as Structure.

Ex:- If we want to store marks of all the students we can declare a structure of stack as follows.

define Size 60

typedef struct Student

{

int roll. no;

char name[30];

[Cont..d]

[Cont..d]

(17)

```

float marks;
}
Stud;
Stud s1[Size];
int top = -1;

```

Roll no	name	marks
59		
40	Mita	66
30	Pita	91
20	Geeta	88.3
10	Seeta	76.5

S1

The above Stack can store the data about whole class in stack.

→ Stack Empty operation

Initially stack is empty. At that time the top should be initialized to -1 or 0. If we set top to -1 initially then stack will contain the elements from 0th position & if we set top to 0 initially the elements will be stored from 1st position, in the stack.

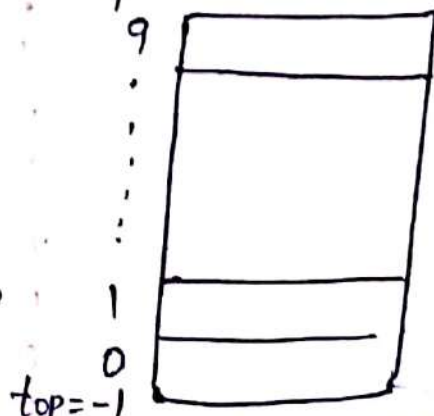
Elements may be pushed on the stack & removed from stack then stack becomes empty. So

If top reaches to -1 we say stack is empty.

```

int sempty()
{
    if (st.top == -1)
        return 1;
    else
        return 0;
}

```



Stack full operation

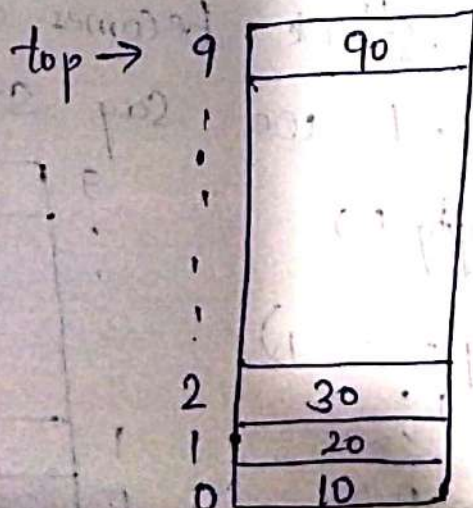
In representation of stack using arrays, size of array means size of stack. By inserting the elements the stack gets filled with elements.

So, check whether stack is full or not before inserting the elements.

Stack full condition is achieved when stack reaches to maximum size of array.

```
int stfull()
{
    if (st.top >= size - 1)
        return 1;
    else
        return 0;
}
```

Thus stfull is a boolean function of stack. If it is full it returns 1 (or) it returns 0.



Here

st.s[0] = 10

st.s[1] = 20 & so on.

And

st.top = 9

The 'Push' and 'Pop' functions

The two important functions of Stack are Push which inserts new elements at the top of the Stack.

void Push (int item)

```
{  
    st.top++; /* top pointer is set to next location */  
    st.s[st.top] = item; /* placing the element at that location */  
}
```

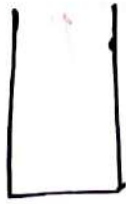
[Here the Push function takes the parameter item for inserting elements into the Stack.]

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

Pop which deletes the elements at the top of the Stack. If Stack is Empty it generates a Stack as 'underflow'.

Push operations

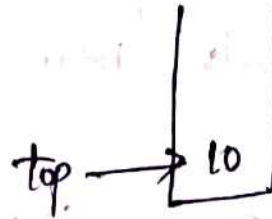
(a) Stack is Empty.



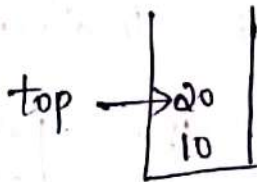
$top = -1$

(a)

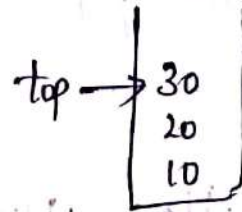
(b) Push element 10.



(c) Push element 20

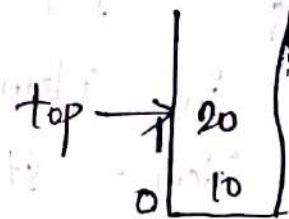
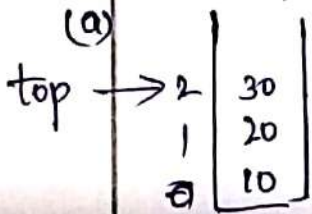


(d) Push element 30



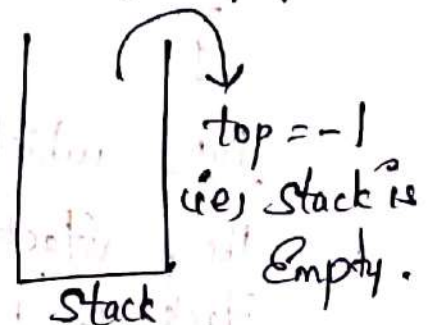
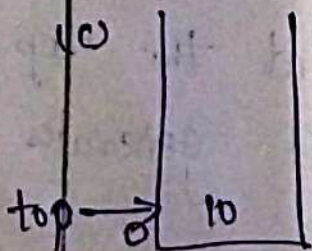
Pop operations

Popping element by $top = 2$ (b) 30 got popped.



20 got popped

(d) 10 got popped.



Program for implementing a stack using arrays.

Operations

* Push, Pop, Stack empty, Stack full, Display.

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
/* Stack Structure */
```

```
struct Stack
```

```
{
```

```
    int s[size];
```

```
    int top;
```

```
} st;
```

/* The stfull function

Input: none, Output: returns 1 or 0 for stack full or not. Called by: main /*

```
int stfull()
```

```
{
```

```
    if (st.top >= size - 1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
/*
```

/*

The Push function

Input : item which is to be pushed

Output : none- Simply Pushes the item onto the stack.

Called by : main

*/

void Push (int item)

{

st.top ++;

st.s[st.top] = item;

}

*/

The stempty function

Input : none

Output : returns 1 or 0 for stack empty or not

Called by : main

*/

int stempty ()

{

if (st.top == -1)

return 1;

else

return 0;

}

/*

/*
The pop function

Input : none

Output : returns the item which is popped from the stack

Called by : main

/*

int pop ()

{
int item;

item = st.s[st.top];

st.top --;

return (item);

}

/*

The display function

Input : none

Output : none - displays the contents of the stack

Called by : main

*/

void display ()

{
int i;

if (st.empty ())

printf("\n Stack is Empty");

else

{

[Cont..d]

for (i = st.top; i >= 0; i--) ← (Displaying stack from top to bottom)

printf("\n %d", st.s[i]);

}

}

/* The main function

Input : none

Output : none

Called by : Os

Calls : push, pop, stempty, stfull, display

*/

void main()

{

int item, choice;

char ans;

st.top = -1;

printf("\n Implementation of stack");

do

{

printf("\n Main Menu");

printf("\n 1. Push 2. Pop 3. Display 4. Exit");

printf("\n %d", &choice);

switch (choice)

{

Case 1: Printf("\n Enter the item to be Pushed"); ②

scanf("%d", &item);

if (stfull())
Printf("\n stack is full");
else

Before Pushing.
we should check
stack full condition.

Push(item);
break;

Case 2: if (stempty())

Printf("\n Empty stack");

Before popping we
should check stack
empty condition.

else

{

item = pop();

Printf("\n The popped element is %d", item);

}

break;

Case 3: display();

break;

Case 4: exit(0);

}

Printf("\n Do you want to Continue");

getch();

}

~~while~~ while (ans == 'Y' // ans == 'y');

getch();

}

or

[if we give Capital 'Y' or
small 'y' it accepts for Main Menu]

Output

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 1

Enter the item to be Pushed 10

Do You want to Continue? y

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 1

Enter the item to be Pushed 20

Do you want to Continue? y

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 3

20

10

Do You want to Continue? y

(Cont...d)

[Cont..d]

22

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 2

The Popped element is 20

Do You want to Continue? y

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 2

The Popped element is 10

Do you want to Continue? y

Main Menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 2

Empty Stack.

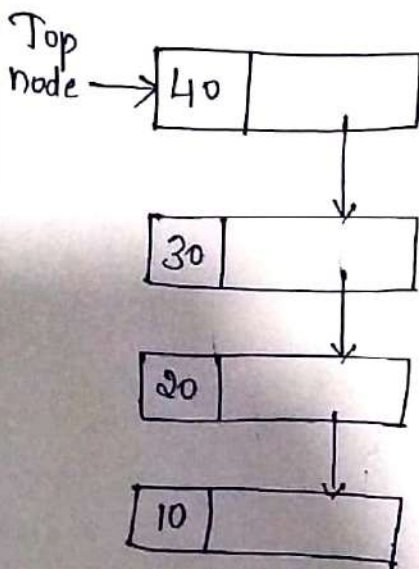
Stack with linked list Implementation

The advantage of implementing stack using linked list is we need to worry about the size of the stack. Since we are using linked list many elements we can insert as many nodes can be created. And the nodes are dynamically created.

"C structure" for linked list stack

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

Each node consists of data and next field. Such a node will be inserted in the stack.



Representing linked stack.

1. Push Operation

```
void Push (int Item, node **top)
```

```
{
    node * New;
    node * get_node (int);
    New = get_node (Item);
    New → next = *top;
    *top = New;
}
```

[The get_node() function is called to allocate the memory for New node]

Ex:-

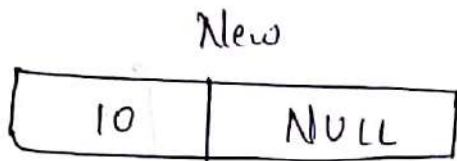
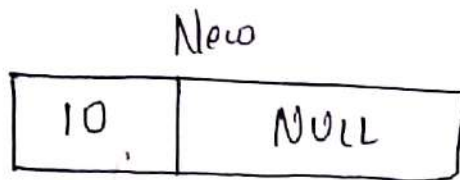
10	NULL
----	------

New/top

```
node * get_node (int item)
```

```
{
    node * temp;
    temp = (node *) malloc (Size of (node));
    if (temp == NULL)
        printf ("In Memory Cannot be allocated");
    temp → data = item;
    temp → next = NULL;
    return (temp);
}
```

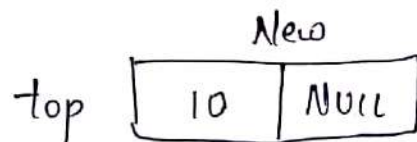
In the main function, when the item to be inserted is entered, a call to Push function is given. In Push function another function get_node is invoked to allocate the memory for a node.



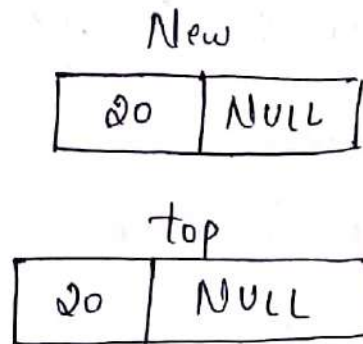
I Item = 10, then by get node function node will be allocated. Then

$*top = New$

This is our new top node. Hence stack will look



II again Push function is called for pushing the value 20 then

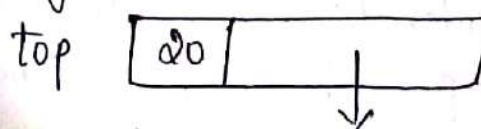


By using following statements

New $\rightarrow$ next = $*top$;

top = New;

And we will get

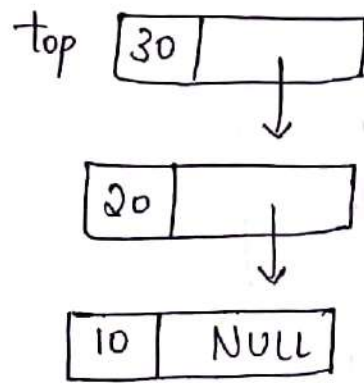


[Cont. . . d]

[cont..d]

when 30 is pushed then

(24)



Thus by pushing the items repeatedly we can create a stack using linked list.

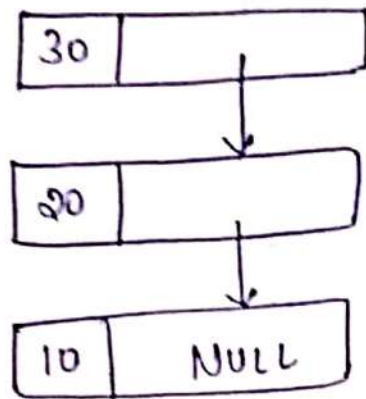
2. Pop Operation

```
int Pop (node **top)
{
    int item;
    node *temp;
    item = (*top) -> data;
    temp = *top;
    *top = (*top) -> next;
    free(temp);
    return(item);
}
```

[We want to delete topnode. hence to remember the data being deleted it is already stored as item.]

← [Taking the address of top in temp node and assigning new top to next node]

If stack is as follows



If we write

$\text{item} = * (\text{top}) \rightarrow \text{data}$

$\text{temp} = * \text{top}$

$* \text{top} = * \text{top} \rightarrow \text{next}$

$\text{free}(\text{temp});$

then $\text{item} = 30$

now $\text{top} = 20$

deallocating memory of node 30. That means node 30 is deleted.

Program for Stack using linked list.

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
/* Declaration for data structure of linked stack */
```

```
typedef struct stack
```

```
{
```

```
    int data;
```

```
    struct stack *next;
```

```
} node;
```

```
void main()
```

```
{
```

```
    /* local declarations */
```

```
    node *top;
```

```
    int data, item, choice;
```

```
    char ans, ch;
```

```
    void Push(int, node **);
```

```
    void Display(node **);
```

```
    int Pop(node **);
```

```
    int Empty(node *);
```

```

top = NULL;
Printf("\n Stack Stack using linked list");
do
{
    Printf("\n The main menu");
    Printf("\n 1. Push 2. Pop 3. Display 4. Exit");
    Printf("\n Enter your choice");
    Scanf("%d", &choice);
    Switch(choice)
    {
        Case 1: Printf("\n Enter the data");
                Scanf("%d", &data);
                Push(data, &top);
                break;
        Case 2: if (IsEmpty(top)) /* before popping the
                                data whether stack is empty or
                                not is checked */
                Printf("\n Stack underflow!");
            else
            {
                item = Pop(&top);
                Printf("\n The popped node is %d", item);
            }
            break;
    }
}

```


Case 3 : Display (&top);

break;

/* displays stack from top to bottom */

Case 4 : printf("\n Do you want to Quit? (y/n)");

ch = getch();

if (ch == 'y')

exit(0);

else

break;

}

printf("\n Do you want to Continue? ");

ans = getch();

getch();

}

while (ans == 'y' // ans == 'y');

}

void Push (int Item, node *top)

{

node *New;

node *get_node (int);

New = get_node (Item);

New → next = *top;

*top = New;

}

```
node * get_node (int item)
```

```
{
```

```
    node * temp;
```

```
    temp = (node *) malloc (sizeof (node));
```

```
    if (temp == NULL)
```

```
        printf("Memory cannot be allocated\n");
```

```
    temp->data = item;
```

```
    temp->next = NULL;
```

```
    return (temp);
```

```
}
```

```
int isempty (node * temp)
```

```
{
```

```
    if (temp == NULL)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

```
int Pop (node * top)
```

```
{
```

```
    int item;
```

```
    node * temp;
```

```
    item = (*top)->data;
```

```
    temp = *top;
```

```
    *top = (*top)->next;
```

```
    free (temp);
```

```
    return (item);
```

```
}
```

```

void Display (node * head)
{
    node * temp;
    temp = * head;
    if (Semphy (temp))
        Printf("In The Stack is Empty!");
    else
    {
        while (temp != NULL)
        {
            Printf("%d", temp->data);
            temp = temp->next;
        }
    }
    getch();
}

```

Output

1. Push
2. Pop
3. Display
4. Exit

Enter your choice |

Enter the data 10

Do you want to Continue? y

The main menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 1

Enter the data 20

Do you want to Continue? y

The main menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 1

Enter the data 30

Do you want to Continue? y

The main menu

1. Push
2. Pop
3. Display
4. Exit

Enter Your choice 3

30

20

10

Do you want to Continue? y

The main menu

1. Push
2. Pop
3. Display
4. Exit

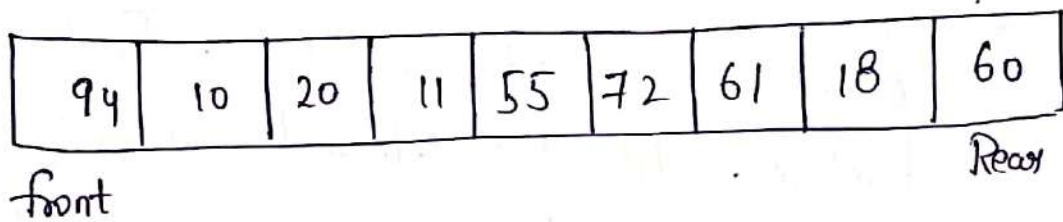
Queue

(28)

Queue :- The Queue is defined as Ordered Collection of elements that has two ends named as front and rear. from front end one can delete the elements and from rear end one can insert the elements.

Ex:-

Queue of people who are waiting for a city bus at bus stop.



Queue ADT

The Queue is a FIFO data structure in which the elements insert first will be first removed. The ADT for queue will be as follows.

AbstractDataType Queue

{

Instance : The queue is Collection of elements in which the element can be inserted by one end called rear and elements get deleted from the end called front.

Operations:

1. `que_full()` - Checks whether queue is full or not.
2. `que_empty()` - Checks whether queue is empty or not.
3. `Q_insert()` - Inserts the element in queue from rear end.
4. `Q_delete()` - Deletes the element from queue by front end.

}

Queue Operations

The Queue is called as FIFO. First in first Out. data Structure. All the elements in the queue is stored sequentially. The various operations on the queue are

- 1) Queue Overflow
- 2) Insertion of the element into the queue
- 3) Queue underflow
- 4) Deletion of element from the queue
- 5) Display of the Queue.

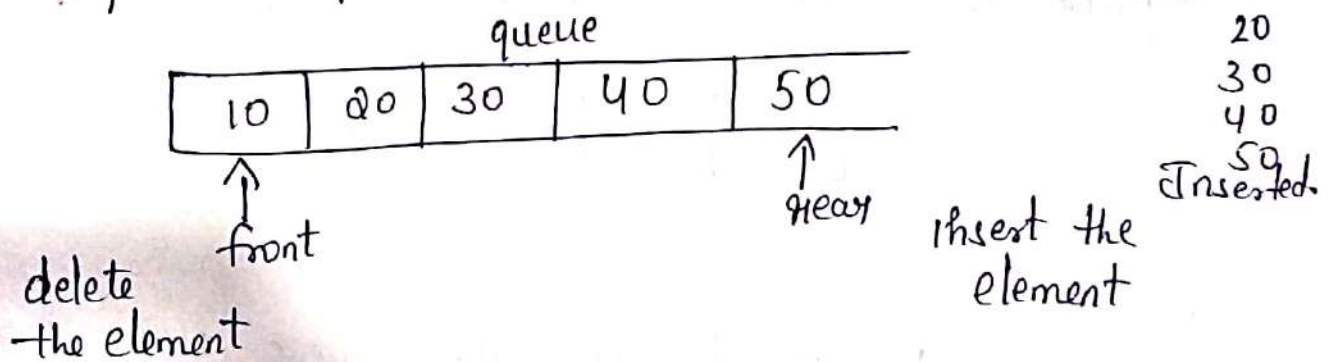
Representation of Queue.

Struct queue

```
{
    int que[Size];
    int front;
    int rear;
} Q;
```

1. Insertion of element into the Queue

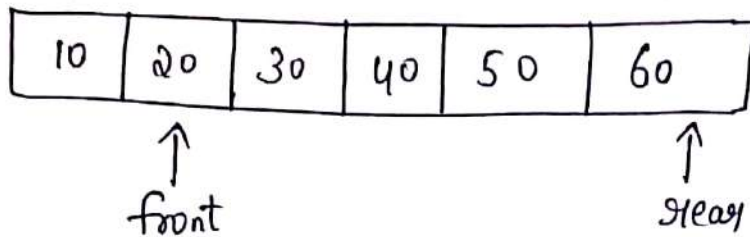
The insertion of any element in the queue will always take place from the rear end.



Before performing insert operation you must check whether the queue is full or not. If the rear pointer is going beyond the maximum size of the queue then the queue overflow occurs.

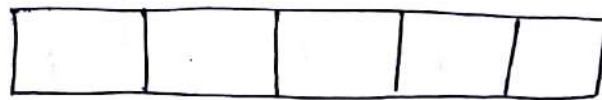
2. Deletion of element from the Queue

The deletion of any element in the queue takes place by the front end



The number 10 gets deleted. Now front element is 20.

Before performing any delete operation one must check whether the queue is empty or not. If queue is empty also we delete an element from the empty queue is called Queue underflow condition.



Program for Implementing the Queue using arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct queue
{
    int que[size];
    int front, rear;
} Q;
```

/*
① The Qfull function

Input : none

Output : 1 or 0 for q full or not

Called By : main

Calls : none

*/

Qfull ()

{
if (Q.rear $\geq$ Size - 1)

return 1;

else

return 0;

}

/*

② The insert function

Input : Item which is to be inserted
in the Q

Output : Rear Value

Called By : main

Calls : none

/*

int insert (int item)

{

if (Q.front == -1)

→ (If Queue exceeds the maximum size of the array then it returns 1 means queue full is true or return 0 means queue full is false)


```

    Q. front ++ ;
    Q. que [ ++ Q. rear ] = item ;
    return Q. rear ;
}

```

(This Condition will occur initially when queue is Empty when single element will be present then both front & rear points to the same.)

③ int Q empty ()

```

{
    if ((Q. front == -1) // (Q. front > Q. rear))
        return 1 ;
}

```

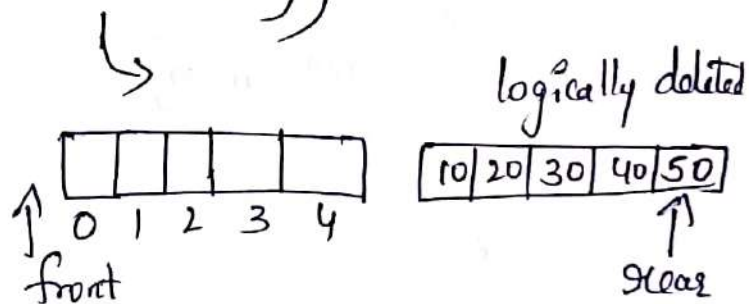
else

```

    return 0 ;
}

```

→ (Always increment the rear pointer & place the element in the queue)



Means Queue Empty .

④ /* The delete function

Input : none

Output : front value

Called By : main

Calls : none

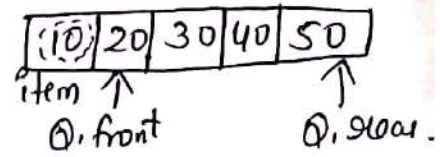
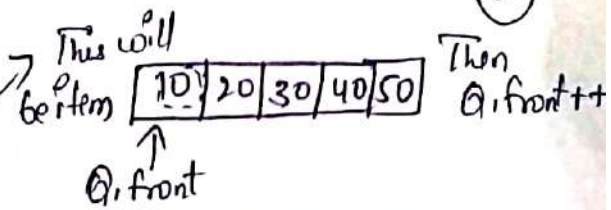
*/

[Cont...d]

```

int delet ()
{
    int item;
    item = Q.que[Q.front];
    Q.front++;
    printf("\n The deleted item is %d", item);
    return Q.front;
}

```



∴ "The deleted item is 10".
finally return Q.front.

⑤ /*
The display function
Input : none
Output : none
Called By : main
Calls : none
*/

```

void display ()
{

```

```

    int i;

```

```

    for (i = Q.front; i <= Q.rear; i++) {
        printf("%d", Q.que[i]);
    }

```

} Printing the queue from front to rear.

```

}

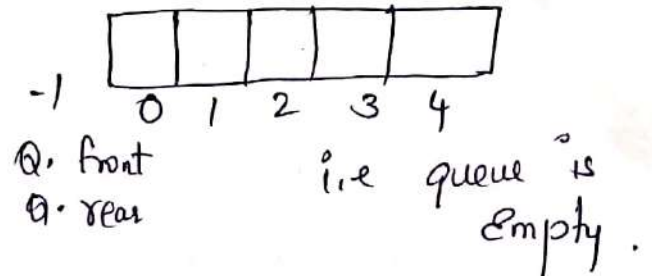
```

```
void main ( )
```

```
{  
    int choice, item;
```

```
    char ans;
```

```
    Q.front = -1;  
    Q.rear = -1; } Initially
```



```
do
```

```
{  
    printf("\n Main Menu");
```

```
    printf("\n 1. Insert 2. Delete 3. Display");
```

```
    printf("\n Enter Your choice");
```

```
    scanf("%d", &choice);
```

```
    switch (choice)
```

```
{
```

```
    case 1: if (Q.full()) // checking for Queue Overflow
```

```
        printf("\n Cannot insert the element");
```

```
    else
```

```
{
```

```
    printf("\n Enter the number to be inserted");
```

```
    scanf("%d", &item);
```

```
    insert (item);
```

```
}
```

```
    break;
```


Case 2: if (Q empty())

Printf("\n Queue Underflow!!");

else

delete();

break;

Case 3: if (Q Empty())

Printf("\n Queue is Empty!");

else

display();

break;

default: Printf("\n Wrong choice!");

break;

}

Printf("\n Do you want to Continue?");

ans = getch();

}

while (ans == 'Y' || ans == 'y');

}

Output;

Main Menu

1. Insert
2. Delete
3. Display

Enter Your choice 1

Enter the number to be inserted 10

Do you want to Continue? y

Main Menu

1. Insert
2. Delete
3. Display

Enter Your choice 1

Enter the number to be inserted 20

Do you want to Continue? y

Main Menu

1. Insert
2. Delete
3. Display

Enter Your choice 3

10 20

Do you want to Continue? y

Main Menu

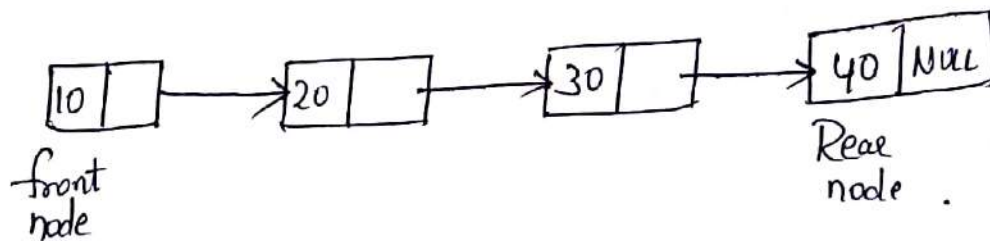
1. Insert
2. Delete
3. Display

Enter Your choice 2

The deleted item is 10.

Queue with linked list Implementation

The queue using linked list will be very much similar to a linked list. The only difference between the two is in queue the leftmost node is called front node and the rightmost node is called the rear node. Main advantage of linked list representation is need not to worry about size of the queue.



The typical node structure

```
typedef struct node
```

```
{
```

```
    int data;
```

```
    struct node next;
```

```
} Q;
```


Implementation of Queue Datastructure using Linked List - C Programming

```
#include<stdio.h>
#include<conio.h>

struct Node
{
    int data;
    struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
    int choice, value;
    clrscr();
    printf("\n:: Queue Implementation using Linked List ::\n");
    while(1){
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d", &value);
                    insert(value);
                    break;
            case 2: delete(); break;
            case 3: display(); break;
            case 4: exit(0);
```

```

        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}

void insert(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode -> next = NULL;
    if(front == NULL)
        front = rear = newNode;
    else{
        rear -> next = newNode;
        rear = newNode;
    }
    printf("\nInsertion is Success!!!\n");
}

void delet()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        front = front -> next;
        printf("\nDeleted element: %d\n", temp->data);
        free(temp);
    }
}

void display()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{

```

```

struct Node *temp = front;
while(temp->next != NULL){
    printf("%d--->",temp->data);
    temp = temp -> next;
}
printf("%d--->NULL\n",temp->data);
}
}

```

Stack Applications

Expressions

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

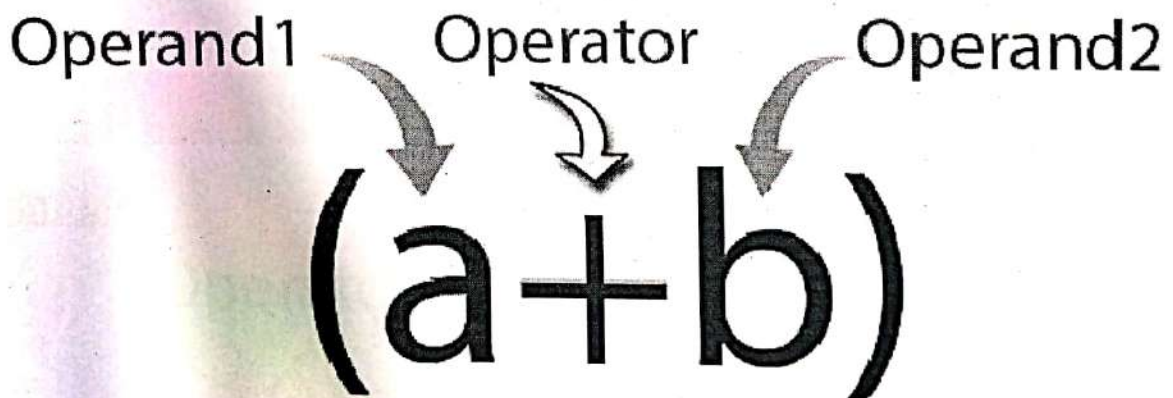
Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example

Operand1 Operand2 Operator

The diagram shows the postfix expression "ab+". Above the expression, the labels "Operand1", "Operand2", and "Operator" are positioned. Three curved arrows point from these labels to the corresponding parts of the expression: from "Operand1" to "a", from "Operand2" to "b", and from "Operator" to "+".

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example

Operator Operand1 Operand2

The diagram shows the prefix expression "+ab". Above the expression, the labels "Operator", "Operand1", and "Operand2" are positioned. Three curved arrows point from these labels to the corresponding parts of the expression: from "Operator" to "+", from "Operand1" to "a", and from "Operand2" to "b".

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

Infix to Postfix Conversion

Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example

Consider the following Infix Expression to be converted into Postfix Expression...

$$\underline{D = A + B * C}$$

- Step 1 - The Operators in the given Infix Expression : = , + , *
- Step 2 - The Order of Operators according to their preference : * , + , =
- Step 3 - Now, convert the first operator * ---- $D = A + B C *$

- Step 4 - Convert the next operator + ----- D = A BC* +

- Step 5 - Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

$$\underline{D A B C * + =}$$

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

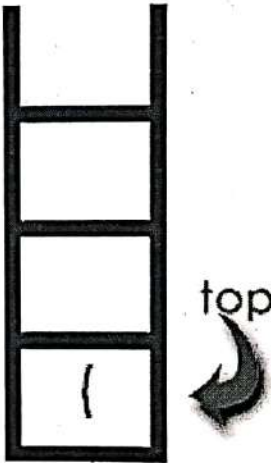
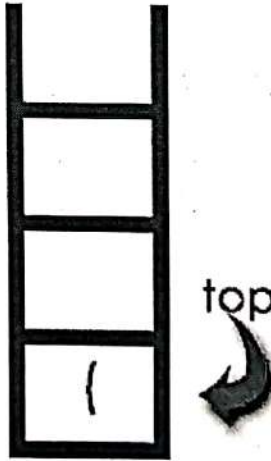
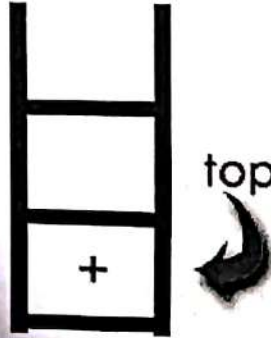
1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
5. If the reading symbol is operator (+, -, *, / etc.), then Push it on to the Stack.
However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

$$\underline{(A + B) * (C - D)}$$

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY 	EMPTY
(	Push '(' 	EMPTY
A	No operation Since 'A' is OPERAND 	A
+	'+' has low priority than '(' so, PUSH '+' 	A

The final Postfix Expression is as follows...

$$\underline{A B + C D - *}$$

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example

Operand1 Operand2 Operator



a b +

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+, -, *, / etc.), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

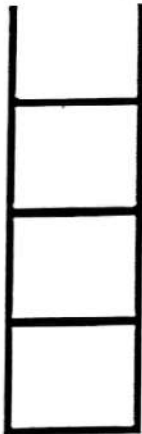
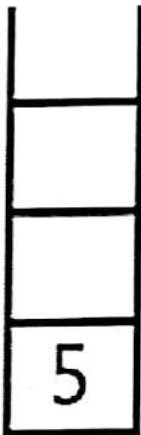
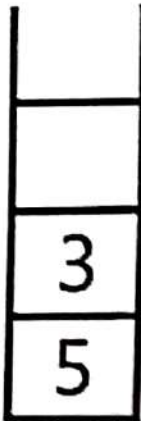
Example

Consider the following Expression...

Infix Expression $(5 + 3) * (8 - 2)$

Postfix Expression $5\ 3\ +\ 8\ 2\ -\ *$

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing