4.13 Some Popular Hash Function is:

4.13.1. Division Method: Pick a number m more modest than the quantity of n of keys in k (The number m is typically picked to be an indivisible number or a number without little divisors, since this often a base number of crashes).

The hash work is:

$\boxed{h(k) = k \bmod m}$ $\boxed{h(k) = k \bmod m + 1}$

For Example: if the hash table has size m = 12 and the key is k = 100, at that point h (k) = 4. Since it requires just a solitary division activity, hashing by division is very quick.

4.13.2. Duplication Method:

The increase technique for making hash capacities works in two stages. In the first place, we increase the vital k by a steady An in the reach 0 < A < 1 and concentrate the fragmentary piece of kA. At that point, we increment this incentive by m and take the floor of the outcome.

The hash work is:

$\boxed{h(k) = \lfloor m(k\,A \bmod 1)\rfloor}$

Where "k A mod 1" means the fractional part of k A, that is, k A -⌊k A⌋.

4.13.3. Mid Square Method:

The key k is squared. Then function H is defined by

H (k) = L

Where L is obtained by deleting digits from both ends of k2. We emphasize that the same position of k2 must be used for all of the keys.

4.13.4.Folding Method:

The key k is partitioned into a number of parts k1, k2.... kn where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

H (k) = k1+ k2+.....+kn

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2- digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

(a) Division Method: Choose a Prime number m close to 99, such as m =97, Then

H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k = 3205 7148 2345
k2= 10272025 51093904 5499025
h (k) = 72 93 99
Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) Folding Method:
Divide the key k into 2 parts and adding yields the following hash address:

H (3205) = 32 + 50 = 82 H (7148) = 71 + 84 = 55
H (2345) = 23 + 45 = 68

Unit 4: Chapter 5

Linear Data Structures

5.0 Objective

This chapter would make you understand the following concepts:

- **What is mean by Stack**
- **Different Operations on stack**
- **Application of stack**
- **Link List Implementation of stack**

5.1.What is a Stack?

A Stack is a straight information structure that follows the LIFO (Last-In-First-Out) guideline. Stack has one end, though the Queue has two finishes (front and back). It contains just a single pointer top pointer highlighting the highest component of the stack. At whatever point a component is included the stack, it is added on the highest point of the stack, and the component can be erased uniquely from the stack. All in all, a stack can be characterized as a compartment in which inclusion and erasure should be possible from the one end known as the highest point of the stack.

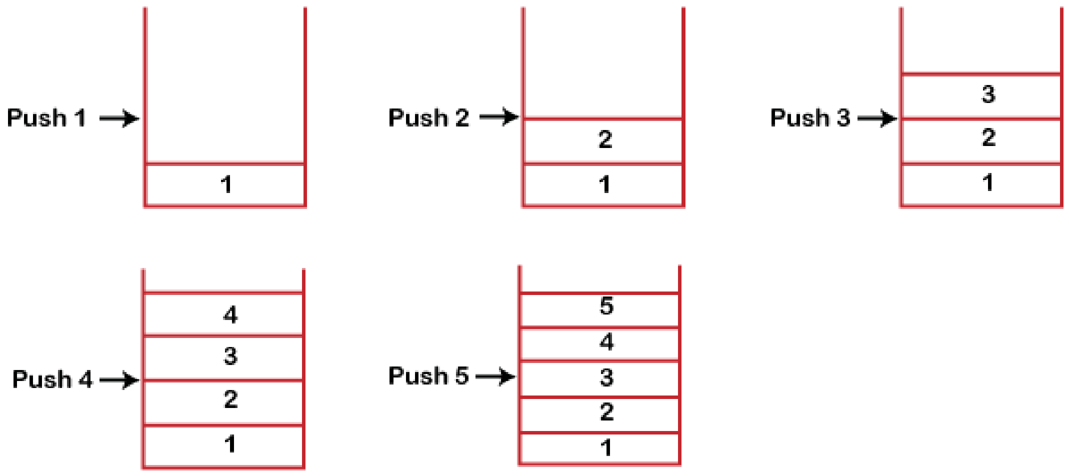
Some key points related to stack

o It is called as stack since it carries on like a certifiable stack, heaps of books, and so forth

o A Stack is a theoretical information type with a pre-characterized limit, which implies that it can store the components of a restricted size.
o It is an information structure that follows some request to embed and erase the components, and that request can be LIFO or FILO.

5.2.Working of Stack

Stack chips away at the LIFO design. As we can see in the underneath figure there are five memory blocks in the stack; along these lines, the size of the stack is 5. Assume we need to store the components in a stack and how about we expect that stack is vacant. We have taken the pile of size 5 as appeared underneath in which we are pushing the components individually until the stack turns out to be full.



Since our stack is full as the size of the stack is 5. In the above cases, we can see that it goes from the top to the base when we were entering the new component in the stack. The stack gets topped off from the base to the top.
At the point when we play out the erase procedure on the stack, there is just a single route for passage and exit as the opposite end is shut. It follows the LIFO design, which implies that the worth entered first will be eliminated last. In the above case, the worth 5 is entered first, so it will be taken out simply after the cancellation of the multitude of different components.

5.3. Standard Stack Operations

Coming up next are some basic activities actualized on the stack:

o **push():** When we embed a component in a stack then the activity is known as a push. On the off chance that the stack is full, at that point the flood condition happens.
o **pop():** When we erase a component from the stack, the activity is known as a pop. In the event that the stack is unfilled implies that no component exists in the stack, this state is known as an undercurrent state.
o **isEmpty():** It decides if the stack is unfilled or not.
o **isFull():** It decides if the stack is full or not.'
o **peek():** It restores the component at the given position.
o **count():** It restores the all out number of components accessible in a stack.
o **change():** It changes the component at the given position.

o **display():** It prints all the components accessible in the stack.

5.3.1. PUSH operation

The means engaged with the PUSH activity is given beneath:

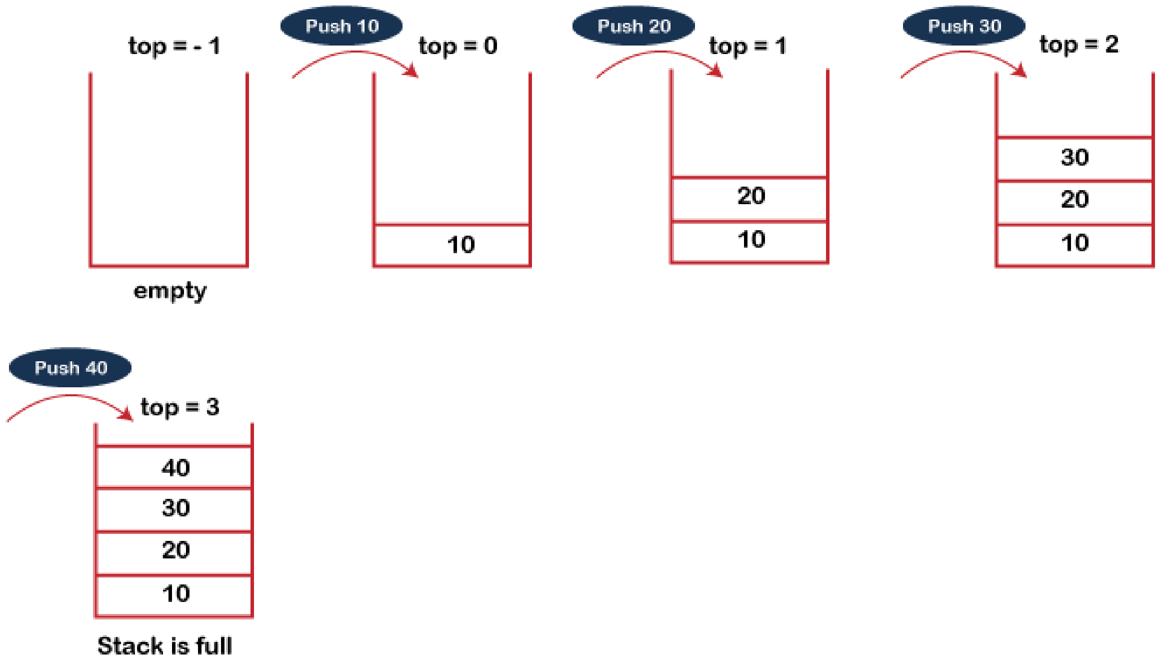
o Before embeddings a component in a stack, we check whether the stack is full.

o If we attempt to embed the component in a stack, and the stack is full, at that point the flood condition happens.

o When we introduce a stack, we set the estimation of top as - 1 to watch that the stack is unfilled.

o When the new component is pushed in a stack, first, the estimation of the top gets increased, i.e., top=top+1, and the component will be put at the new situation of the top.

o The components will be embedded until we arrive at the maximum size of the stack.



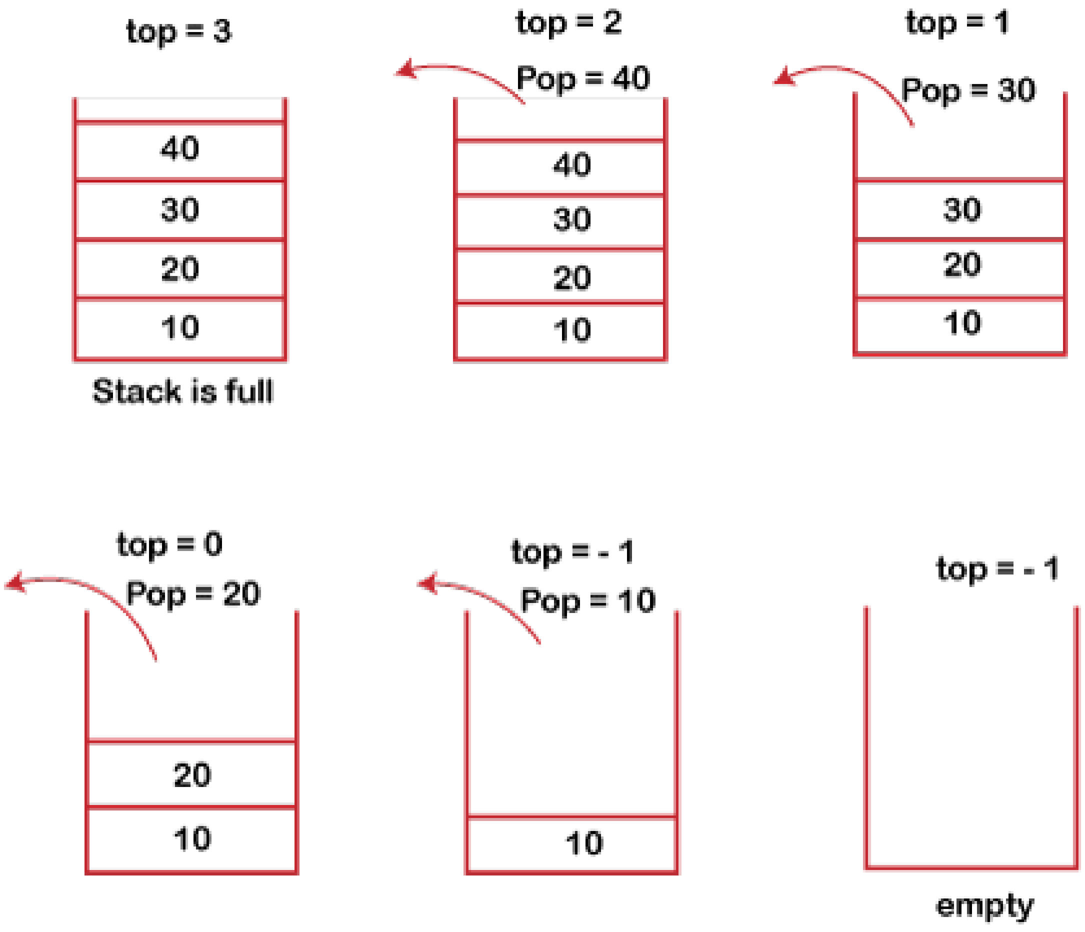
5.3.2. POP operation

The means engaged with the POP activity is given beneath:

o Before erasing the component from the stack, we check whether the stack is vacant.

o If we attempt to erase the component from the vacant stack, at that point the sub-current condition happens.

o If the stack isn't unfilled, we first access the component which is pointed by the top o Once the pop activity is played out, the top is decremented by 1, i.e., top=top-



5.4. Applications of Stack

The following are the applications of the stack:

Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
int main()
{
cout<<"Hello";
cout<<"Data Structure";
}
```

As we probably are aware, each program has an opening and shutting supports; when the initial supports come, we push the supports in a stack, and when the end supports show up, we pop the initial supports from the stack. Subsequently, the net worth comes out to be zero. In the event that any image is left in the stack, it implies that some language structure happens in a program.

- **String inversion:** Stack is additionally utilized for switching a string. For instance, we need to switch a "Information Structure" string, so we can accomplish this with the assistance of a stack.

To start with, we push all the characters of the string in a stack until we arrive at the invalid character.

In the wake of pushing all the characters, we begin taking out the character individually until we arrive at the lower part of the stack.

- **UNDO/REDO:** It can likewise be utilized for performing UNDO/REDO tasks. For instance, we have a manager where we compose 'a', at that point 'b', and afterward 'c'; hence, the content written in a supervisor is abc. Thus, there are three expresses, a, stomach muscle, and abc, which are put away in a stack. There would be two stacks in which one stack shows UNDO state, and different shows REDO state.

In the event that we need to perform UNDO activity, and need to accomplish 'stomach muscle' state, at that point we actualize pop activity.

- **Recursion:** The recursion implies that the capacity is calling itself once more. To keep up the past states, the compiler makes a framework stack in which all the past records of the capacity are kept up.

- **DFS(Depth First Search):** This hunt is actualized on a Graph, and Graph utilizes the stack information structure.

- **Backtracking:** Suppose we need to make a way to tackle a labyrinth issue. In the event that we are moving in a specific way, and we understand that we please the incorrect way. To come toward the start of the way to make another way, we need to utilize the stack information structure.

- **Expression change:** Stack can likewise be utilized for articulation transformation. This is perhaps the main utilizations of stack. The rundown of the articulation change is given underneath:Infix to prefix
Infix to postfix
Prefix to infix
Prefix to postfix
Postfix to infix

- **Memory the board**: The stack deals with the memory. The memory is alloted in the touching memory blocks. The memory is referred to as stack memory as all the factors are allocated in a capacity call stack memory. The memory size alloted to the program is known to the compiler. At the point when the capacity is made, every one of its factors are doled out in the stack memory. At the point when the capacity finished its execution, all the factors doled out in the stack are delivered.

5.4.1. Cluster execution of Stack

In cluster execution, the stack is shaped by utilizing the exhibit. All the activities with respect to the stack are performed utilizing exhibits. Lets perceive how every activity can be actualized on the stack utilizing cluster information structure.

Adding a component onto the stack (push activity)

Adding a component into the highest point of the stack is alluded to as push activity. Push activity includes following two stages.

1. Increment the variable Top with the goal that it can now refere to the following memory area.
2. Add component at the situation of increased top. This is alluded to as adding new component at the highest point of the stack.

Stack is overflown when we attempt to embed a component into a totally filled stack thusly, our primary capacity should consistently stay away from stack flood condition.

Algorithm:

```
begin
if top = n then stack full
top = top + 1
stack (top) : = item;
end
```

Time Complexity : o(1)

5.4.2. Implementation of push algorithm in C language

```
void push (intval,int n) //n is size of the stack
{
if (top == n )
printf("\n Overflow");
else
    {
top = top +1;
stack[top] = val;
    }
}
```

Algorithm :

```
begin
if top = 0 then stack empty;
item := stack(top);
top = top - 1;
end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
int pop ()
{
if(top == -1)
    {
printf("Underflow");
return 0;
    }
else
    {
return stack[top - - ];
    }
}
```

5.4.3. Visiting each element of the stack (Peek operation)

Look actKivity includes restoring the component which is available at the highest point of the stack without erasing it. Sub-current condition can happen in the event that we attempt to restore the top compo:.,nent in an all around void stack.

Algorithm :

```
PEEK (STACK, TOP)
Begin
if top = -1 then stack empty
item = stack[top]
return item
End
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```
int peek()
{
if (top == -1)
    {
printf("Underflow");
return 0;
    }
else
    {
return stack [top];
    }
}
```

5.4.4. Menu Driven program in C implementing all the stack operations

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

printf("Enter the number of elements in the stack ");
scanf("%d",&n);
printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
while(choice != 4)
   {
printf("Chose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n");
scanf("%d",&choice);
switch(choice)
      {
case 1:
         {
push();
break;
         }
case 2:
         {
pop();
break;
         }
case 3:
         {
show();
break;
         }
case 4:
         {
printf("Exiting....");
break;
         }
default:
         {
printf("Please Enter valid choice ");
         }
      };
   }
}
```

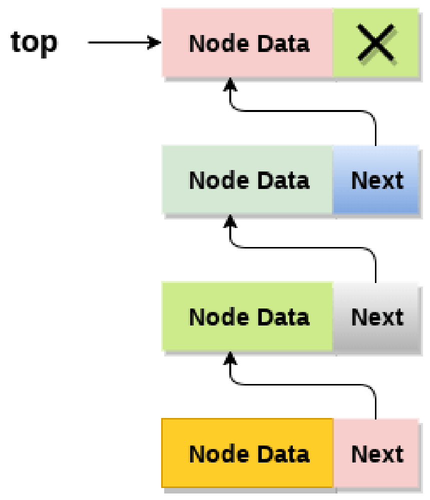
```
void push ()
{
intval;
if (top == n )
printf("\n Overflow");
else
    {
printf("Enter the value?");
scanf("%d",&val);
top = top +1;
stack[top] = val;
    }
}

void pop ()
{
if(top == -1)
printf("Underflow");
else
top = top -1;
}
void show()
{
for (i=top;i>=0;i--)
    {
printf("%d\n",stack[i]);
    }
if(top == -1)
    {
printf("Stack is empty");
    }
}
```

5.5. Linked list implementation of stack

Rather than utilizing cluster, we can likewise utilize connected rundown to execute stack. Connected rundown assigns the memory powerfully. Nonetheless, time unpredictability in both the situation is same for all the tasks for example push, pop and look.

In connected rundown execution of stack, the hubs are kept up non-coterminously in the memory. Every hub contains a pointer to its nearby replacement hub in the stack. Stack is supposed to be overflown if the space left in the memory pile isn't sufficient to make a hub.



Stack

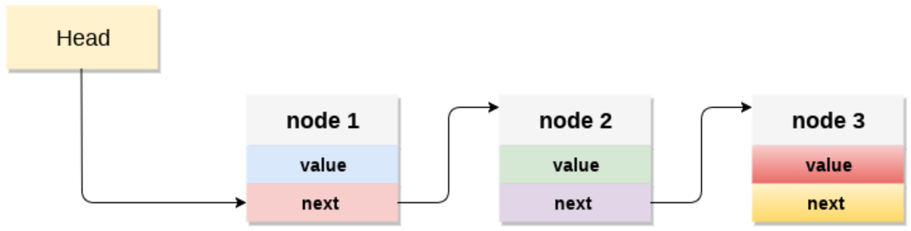
The top most hub in the stack consistently contains invalid in its location field. Lets examine the manner by which, every activity is acted in connected rundown usage of stack.

5.5.1. Adding a node to the stack (Push operation)

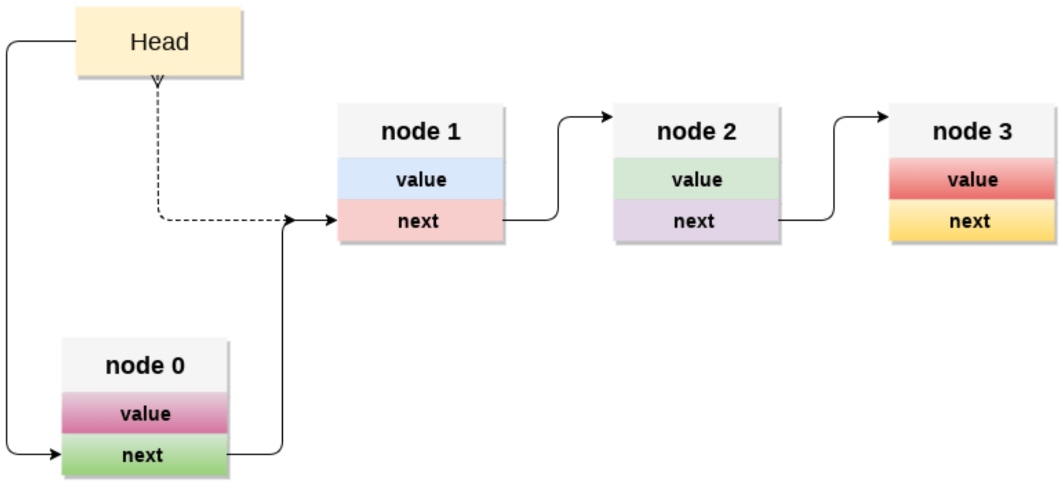
Adding a hub to the stack is alluded to as push activity. Pushing a component to a stack in connected rundown execution is not quite the same as that of a cluster usage. To push a component onto the stack, the accompanying advances are included.

1. Create a hub first and designate memory to it.
2. If the rundown is vacant then the thing is to be pushed as the beginning hub of the rundown. This incorporates doling out an incentive to the information part of the hub and allot invalid to the location part of the hub.
3. If there are a few hubs in the rundown effectively, at that point we need to add the new component in the start of the rundown (to not disregard the property of the stack). For this reason, allocate the location of the beginning component to the location field of the new hub and make the new hub, the beginning hub of the rundown.

Time Complexity :



o(1)



New Node

C implementation:

```
void push ()
{
intval;
struct node *ptr =(struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
    {
printf("not able to push the element");
    }
else
    {
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
        {
ptr->val = val;
ptr -> next = NULL;
head=ptr;
        }
else
        {
ptr->val = val;
ptr->next = head;
head=ptr;

        }
printf("Item pushed");

    }
}
```

5.5.2. Deleting a hub from the stack (POP activity)

Erasing a hub from the highest point of stack is alluded to as pop activity. Erasing a hub from the connected rundown usage of stack is not quite the same as that in the exhibit execution. To pop a component from the stack, we need to follow the accompanying advances :

Check for the undercurrent condition: The sub-current condition happens when we attempt to fly from a generally unfilled stack. The stack will be unfilled if the head pointer of the rundown focuses to invalid.

Change the head pointer in like manner: In stack, the components are popped uniquely from one end, thusly, the worth put away in the head pointer should be erased and the hub should be liberated. The following hub of the head hub presently turns into the head hub.

Time Complexity: o(n)

C implementation

```
void pop()
{
int item;
struct node *ptr;
if (head == NULL)
    {
printf("Underflow");
    }
else
    {
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");
      }
}
```

5.5.3. Display the nodes (Traversing)

Showing all the hubs of a stack requires navigating all the hubs of the connected rundown coordinated as stack. For this reason, we need to follow the accompanying advances.

1. Copy the head pointer into an impermanent pointer.
2. Move the brief pointer through all the hubs of the rundown and print the worth field joined to each hub.

Time Complexity: o(n)

C Implementation

```
void display()
{
inti;
struct node *ptr;
ptr=head;
if(ptr == NULL)
    {
printf("Stack is empty\n");
    }
else
    {
printf("Printing Stack elements \n");
while(ptr!=NULL)
        {
printf("%d\n",ptr->val);
ptr = ptr->next;
        }
    }
}
```

5.5.4. Menu Driven program in C implementing all the stack operations using linked list:

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
intval;
struct node *next;
};
struct node *head;
void main ()
{
int choice=0;
printf("\n*********Stack operations using linked list*********\n");
printf("\n----------------------------------------------\n");
while(choice != 4)
    {
printf("\n\nChose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n");
scanf("%d",&choice);
switch(choice)
        {
case 1:
            {
push();
break;
            }
case 2:
            {
pop();
break;
            }
case 3:
            {
display();
break;
            }
case 4:
            {
printf("Exiting....");
break;
            }
default:
            {
printf("Please Enter valid choice ");
            }
    };
}
}
void push ()
{
intval;
struct node *ptr = (struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
    {
printf("not able to push the element");
    }
else
    {
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
        {
ptr->val = val;
ptr -> next = NULL;
head=ptr;
        }
else
        {
ptr->val = val;
ptr->next = head;
```

```
head=ptr;

        }
printf("Item pushed");

    }
}

void pop()
{
int item;
struct node *ptr;
if (head == NULL)
    {
printf("Underflow");
    }
else
    {
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");

    }
}
void display()
{
inti;
struct node *ptr;
ptr=head;
if(ptr == NULL)
    {
printf("Stack is empty\n");
    }
else
    {
printf("Printing Stack elements \n");
while(ptr!=NULL)
        {
printf("%d\n",ptr->val);
ptr = ptr->next;
        }
    }
}
```

Unit 4 - Chapter 6
Queue

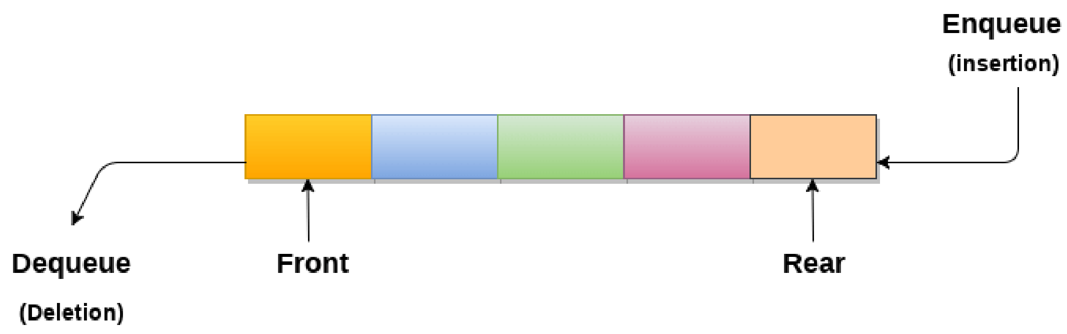
6.0.Objective

This chapter would make you understand the following concepts:

- **Queue**
- **Definition**
- **Operations, Implementation of simple**
- **queue (Array and Linked list) and applications of queue-BFS**
- **Types of queues: Circular, Double ended, Priority,**

6.1.Queue

1. A Queue can be characterized as an arranged rundown which empowers embed tasks to be performed toward one side called REAR and erase activities to be performed at another end called FRONT.
2. Queue is alluded to be as First In First Out rundown.
3. For instance, individuals sitting tight in line for a rail ticket structure a Queue



6.2.Applications of Queue

Because of the way that line performs activities on first in first out premise which is very reasonable for the requesting of activities. There are different uses of queues examined as beneath.

1. Queues are generally utilized as hanging tight records for a solitary shared asset like printer, plate, CPU.
2. Queues are utilized in offbeat exchange of information (where information isn't being moved at similar rate between two cycles) for eg. pipes, document IO, attachments.
3. Queues are utilized as cradles in the greater part of the applications like MP3 media player, CD player, and so on
4. Queue are utilized to keep up the play list in media major parts to add and eliminate the tunes from the play-list.
5. Queues are utilized in working frameworks for dealing with interferes.

Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

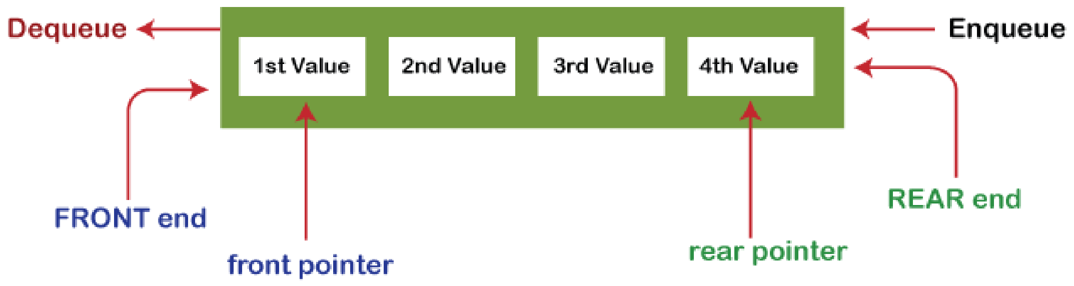
6.3.Types of Queues

Prior to understanding the sorts of queues, we first glance at 'what is Queue'.

What is the Queue?

A queue in the information construction can be viewed as like the queue in reality. A queue is an information structure in which whatever starts things out will go out first.

It follows the FIFO (First-In-First-Out) arrangement. In Queue, the inclusion is done from one end known as the backside or the tail of the queue, though the erasure is done from another end known as the front end or the top of the line. At the end of the day, it very well may be characterized as a rundown or an assortment with an imperative that the addition can be performed toward one side called as the backside or tail of the queue and cancellation is performed on another end called as the front end or the top of the queue.



6.4.Operations on Queue

o Enqueue: The enqueue activity is utilized to embed the component at the backside of the queue. It brings void back.

o Dequeue: The dequeue activity plays out the erasure from the front-finish of the queue. It additionally restores the component which has been eliminated from the front-end. It restores a number worth. The dequeue activity can likewise be intended to void.

o Peek: This is the third activity that profits the component, which is pointed by the front pointer in the queue yet doesn't erase it.

o Queue flood (isfull): When the Queue is totally full, at that point it shows the flood condition.

o Queue undercurrent (isempty): When the Queue is unfilled, i.e., no components are in the Queue then it tosses the sub-current condition.

A Queue can be addressed as a compartment opened from both the sides in which the component can be enqueued from one side and dequeued from another side as demonstrated in the beneath figure:

6.5.Implementationof Queue

There are two different ways of executing the Queue:

6.5.1.Consecutive assignment: The successive distribution in a Queue can be executed utilizing a cluster.

6.5.2.Linked list allocation: The linked list portion in a Queue can be actualized utilizing a linked list.

6.6.What are the utilization instances of Queue?

Here, we will see this present reality situations where we can utilize the Queue information structure. The Queue information structure is primarily utilized where there is a shared asset that needs to serve the different asks for however can serve a solitary solicitation at a time. In such cases, we need to utilize the Queue information structure for lining up the solicitations. The solicitation that shows up first in the line will be served first. Coming up next are this present reality situations in which the Queue idea is utilized:

o Suppose we have a printer divided among different machines in an organization, and any machine or PC in an organization can send a print solicitation to the printer. In any case, the printer can serve a solitary solicitation at a time, i.e., a printer can print a solitary archive at a time. At the point when any print demand comes from the organization, and if the printer is occupied, the printer's program will put the print demand in a line.

o If the solicitations are accessible in the Queue, the printer takes a solicitation from the front of the Queue, and serves it.

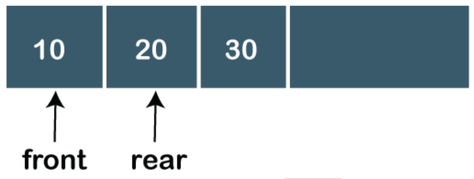
o The processor in a PC is likewise utilized as a shared asset. There are numerous solicitations that the processor should execute, however the processor can serve a solitary ask for or execute a solitary interaction at a time. Consequently, the cycles are kept in a Queue for execution.

6.7.Types of Queue

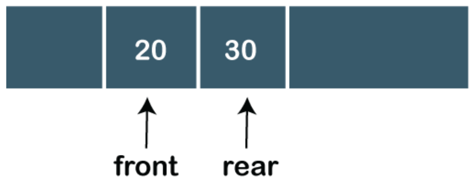
There are four kinds of Queues:

6.7.1.Linear Queue

In Linear Queue, an inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside, and the end at which the erasure happens is known as front end. It carefully keeps the FIFO rule. The straight Queue can be addressed, as demonstrated in the beneath figure:



The above figure shows that the components are embedded from the backside, and on the off chance that we embed more components in a Queue, at that point the back worth gets increased on each addition. In the event that we need to show the cancellation, at that point it tends to be addressed as:

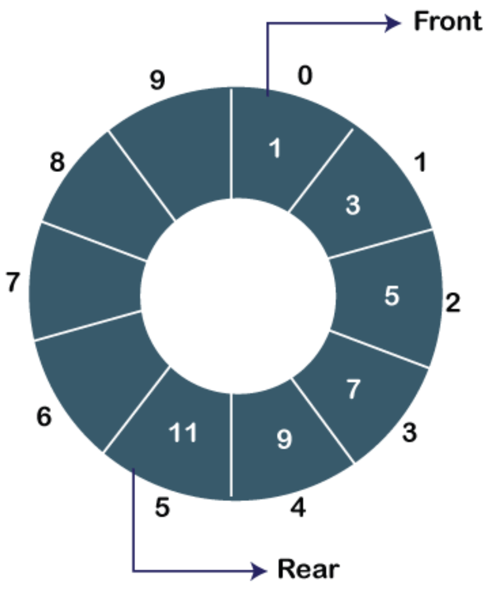


In the above figure, we can see that the front pointer focuses to the following component, and the component which was recently pointed by the front pointer was erased.

The significant disadvantage of utilizing a straight Queue is that inclusion is done distinctly from the backside. In the event that the initial three components are erased from the Queue, we can't embed more components despite the fact that the space is accessible in a Linear Queue. For this
situation, the straight Queue shows the flood condition as the back is highlighting the last component of the Queue.

6.7.2.Circular Queue

In Circular Queue, all the hubs are addressed as round. It is like the direct Queue aside from that the last component of the line is associated with the principal component. It is otherwise called Ring Buffer as all the finishes are associated with another end. The round line can be addressed as:



The disadvantage that happens in a direct line is overwhelmed by utilizing the roundabout queue. On the off chance that the unfilled space is accessible in a round line, the new component can be included a vacant space by just augmenting the estimation of back.

6.7.3.Priority Queue

A need queue is another exceptional sort of Queue information structure in which every component has some need related with it. In view of the need of the component, the components are organized in a need line. In the event that the components happen with a similar need, at that point they are served by the FIFO rule.

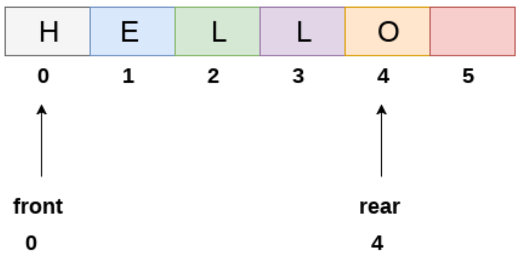
In need Queue, the inclusion happens dependent on the appearance while the cancellation happens dependent on the need. The need Queue can be appeared as:
The above figure shows that the most elevated need component starts things out and the components of a similar need are organized dependent on FIFO structure.

6.7.4.Deque

Both the Linear Queue and Deque are distinctive as the direct line follows the FIFO standard while, deque doesn't follow the FIFO rule. In Deque, the inclusion and erasure can happen from the two closures.

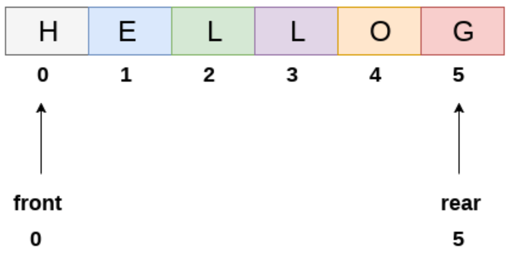
6.8.Array representation of Queue

We can without much of a stretch address queue by utilizing direct exhibits. There are two factors for example front and back, that are actualized on account of each queue. Front and back factors highlight the situation from where inclusions and cancellations are acted in a queue. At first, the estimation of front and queue is - 1 which addresses an unfilled queue. Cluster portrayal of a queue containing 5 components alongside the separate estimations of front and back, is appeared in the accompanying figure.



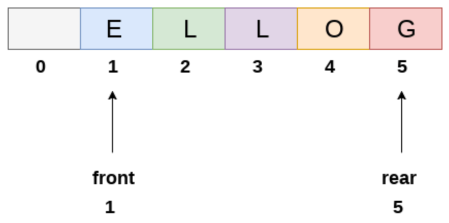
6.9.Queue

The above figure shows the queue of characters shaping the English word "Hi". Since, No cancellation is acted in the line till now, thusly the estimation of front remaining parts - 1 . Be that as it may, the estimation of back increments by one each time an addition is acted in the queue. Subsequent to embeddings a component into the queue appeared in the above figure, the queue will look something like after. The estimation of back will become 5 while the estimation of front remaining parts same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to embed any component in a line

Check if the line is as of now full by contrasting back with max - 1. assuming this is the case, at that point return a flood blunder.

In the event that the thing is to be embedded as the principal component in the rundown, all things considered set the estimation of front and back to 0 and addition the component at the backside.

In any case continue to expand the estimation of back and addition every component individually having back as the file.

6.9.1.Algorithm

```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

6.9.2.C Function

```
void insert (int queue[], int max, int front, int rear, int item)
{
if (rear + 1 == max)
   {
printf("overflow");
   }
else
   {
if(front == -1 && rear == -1)
      {
```

```
front = 0;
rear = 0;
        }
else
        {
rear = rear + 1;
        }
queue[rear]=item;
    }
}
```

Algorithm to delete an element from the queue
If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

6.9.3.Algorithm

```
Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT
```

C Function

```
int delete (int queue[], int max, int front, int rear)
{
int y;
if (front == -1 || front > rear)

    {
printf("underflow");
    }
else
    {
        y = queue[front];
if(front == rear)
        {
```

```
front = rear = -1;
else
front = front + 1;

        }
return y;
    }
}
```

6.9.4.Menu driven program to implement queue using array

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
int choice;
while(choice != 4)
    {
printf("\n*************************Main
Menu*****************************\n");
printf("\n=====================================================
==============\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",&choice);
switch(choice)
        {
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
int item;
printf("\nEnter the element\n");
scanf("\n%d",&item);
if(rear == maxsize-1)
    {
printf("\nOVERFLOW\n");
return;
    }
if(front == -1 && rear == -1)
    {
front = 0;
rear = 0;
    }
else
    {
rear = rear+1;
    }
queue[rear] = item;
printf("\nValue inserted ");

}
void delete()
{
int item;
if (front == -1 || front > rear)
    {
printf("\nUNDERFLOW\n");
return;
```

```
    }
else
    {
item = queue[front];
if(front == rear)
        {
front = -1;
rear = -1 ;
        }
else
        {
front = front + 1;
        }
printf("\nvalue deleted ");
    }


}

void display()
{
inti;
if(rear == -1)
    {
printf("\nEmpty queue\n");
    }
else
    {   printf("\nprinting values .....\n");
for(i=front;i<=rear;i++)
        {
printf("\n%d\n",queue[i]);
        }
    }
}
```

Output:

```
*************Main Menu**************
==============================================
1.insert an element
2.Delete an element
3.Display the queue
```

```
4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

*************Main Menu**************

===============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*************Main Menu**************

====================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

*************Main Menu**************
===============================================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

*************Main Menu**************

==============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

6.10.Linked List implementation of Queue

Because of the disadvantages examined in the past part of this instructional exercise, the exhibit usage can not be utilized for the huge scope applications where the queues are actualized. One of the option of cluster usage is connected rundown execution of queue.

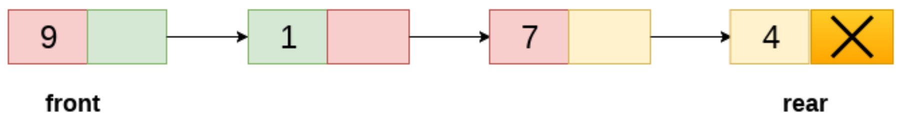
The capacity prerequisite of connected portrayal of a queue with n components is o(n) while the time necessity for tasks is o(1).

In a linked queue, every hub of the queue comprises of two sections for example information part and the connection part. Every component of the queue focuses to its nearby next component in the memory.

In the linked queue, there are two pointers kept up in the memory for example front pointer and back pointer. The front pointer contains the location of the beginning component of the queue while the back pointer contains the location of the last component of the queue.

Inclusion and erasures are performed at back and front end separately. On the off chance that front and back both are NULL, it shows that the line is vacant.

The connected portrayal of queue is appeared in the accompanying figure.



Linked Queue

6.11.Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

6.11.1.Insert operation

The addition activity attach the line by adding a component to the furthest limit of the line. The new component will be the last component of the line.

Right off the bat, assign the memory for the new hub ptr by utilizing the accompanying assertion.

Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two situation of embeddings this new hub ptr into the connected line.

In the principal situation, we embed component into an unfilled queue. For this situation, the condition front = NULL turns out to be valid. Presently, the new component will be added as the lone component of the queue and the following pointer of front and back pointer both, will highlight NULL.

```
ptr -> data = item;
if(front == NULL)
        {
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
        }
```

In the subsequent case, the queue contains more than one component. The condition front = NULL turns out to be bogus. In this situation, we need to refresh the end pointer back with the goal that the following pointer of back will highlight the new hub ptr. Since, this is a connected line, consequently we likewise need to make the back pointer highlight the recently added hub ptr. We additionally need to make the following pointer of back highlight NULL.

```
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

6.11.2.Algorithm

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
Step 4: END
```

6.11.3.C Function

```
void insert(struct node *ptr, int item; )
{
ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
   {
printf("\nOVERFLOW\n");
return;
   }
else
   {
ptr -> data = item;
if(front == NULL)
      {
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
      }
else
      {
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
        }
    }
}
```

6.12.Deletion

Cancellation activity eliminates the component that is first embedded among all the queue components. Right off the bat, we need to check either the rundown is unfilled or not. The condition front == NULL turns out to be valid if the rundown is unfilled, for this situation, we essentially compose undercurrent on the comfort and make exit. Else, we will erase the component that is pointed by the pointer front. For this reason, duplicate the hub pointed by the front pointer into the pointer ptr. Presently, move the front pointer, highlight its next hub and free the hub pointed by the hub ptr. This is finished by utilizing the accompanying assertions.

```
ptr = front;
front = front -> next;
free(ptr);
```

The algorithm and C function is given as follows.

6.12.1.Algorithm

```
Step 1: IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

6.12.2.C Function

```
void delete (struct node *ptr)
{
if(front == NULL)
    {
printf("\nUNDERFLOW\n");
return;
    }
else
    {
ptr = front;
front = front -> next;
free(ptr);
```

```
    }
}
```

6.12.3.Menu-Driven Program implementing all the operations on Linked Queue

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
int choice;
while(choice != 4)
    {
printf("\n*************************Main
Menu*****************************\n");
printf("\n========================================================
===============\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",& choice);
switch(choice)
        {
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
    {
printf("\nOVERFLOW\n");
return;
    }
else
    {
printf("\nEnter value?\n");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
        {
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
        }
else
        {
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
        }
    }
}
void delete ()
```

```
{
struct node *ptr;
if(front == NULL)
    {
printf("\nUNDERFLOW\n");
return;
    }
else
    {
ptr = front;
front = front -> next;
free(ptr);
    }
}
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
    {
printf("\nEmpty queue\n");
    }
else
    {   printf("\nprinting values .....\n");
while(ptr != NULL)
        {
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
        }
    }
}
```

Output:

```
***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
```