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.

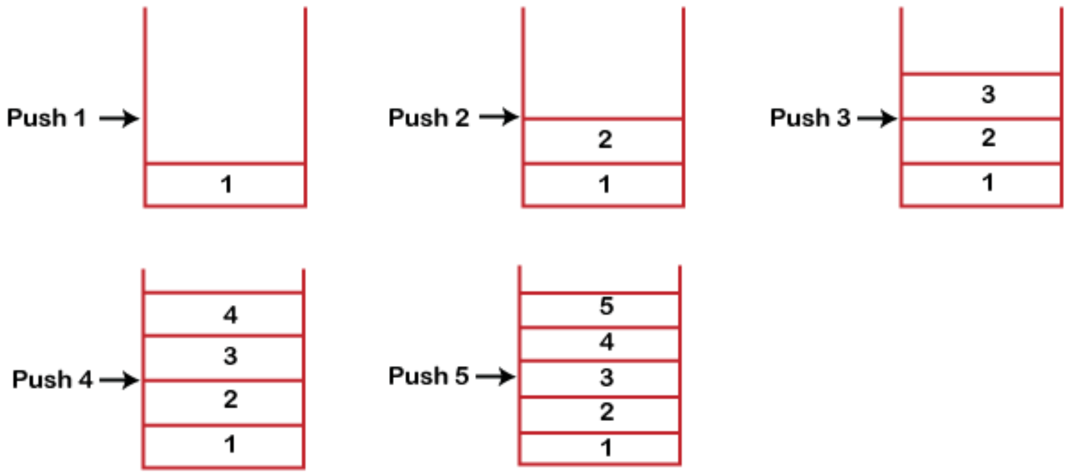
Some key points related to stack

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:

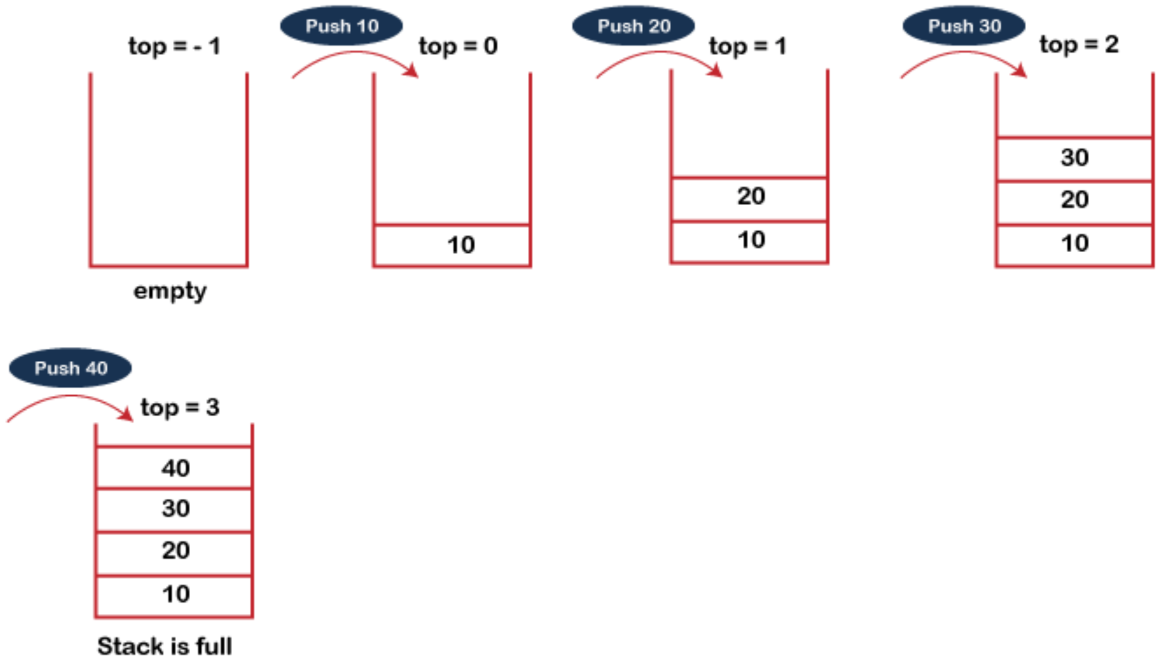
o Before embeddings a component in a stack, we check whether the stack is full.

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.



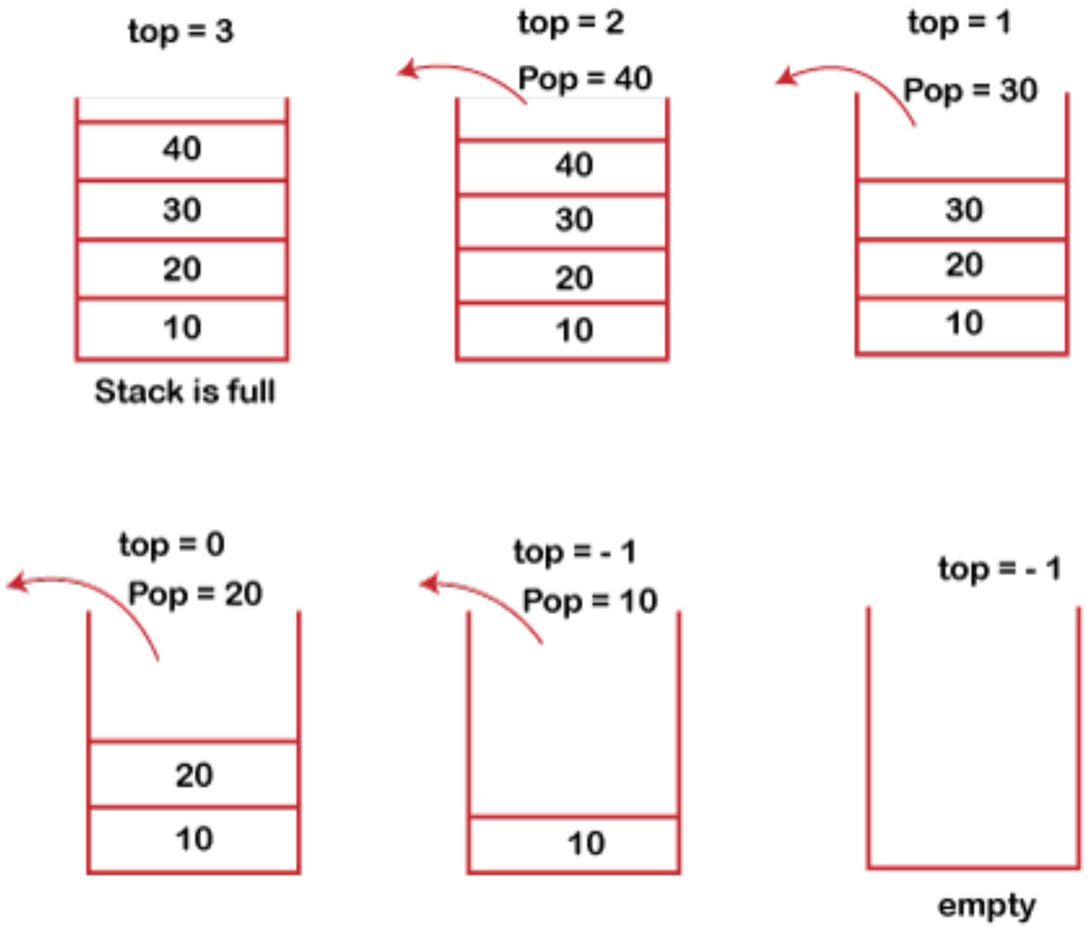
5.3.2. POP operation

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 ");
         }
      };
   }
}
```

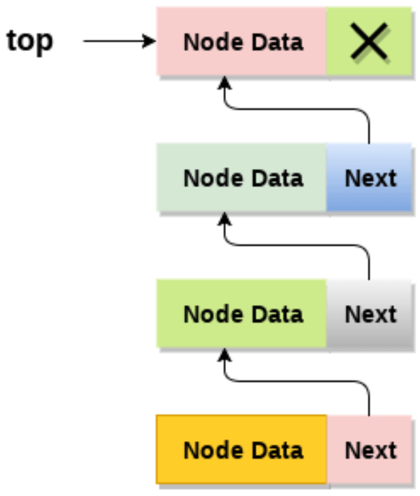
```
void push ()
{
intval;
if (top == n )
printf("\n Overflow");
else
    {
printf("Enter the value?");
scanf("%d",&val);
top = top +1;
stack[top] = val;
    }
}

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

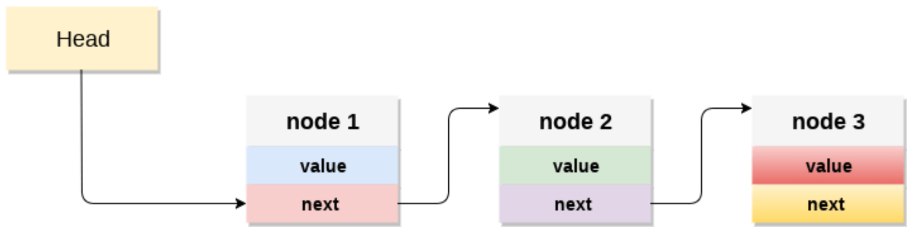
The top most hub in the stack consistently contains invalid in its location field. Lets examine the manner by which, every activity is acted in connected rundown usage of stack.

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

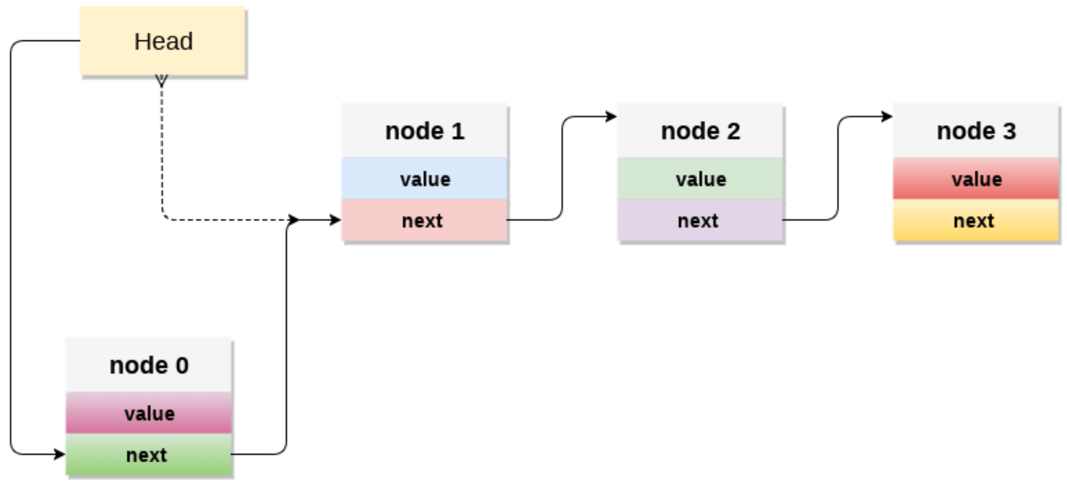
Adding a hub to the stack is alluded to as push activity. Pushing a component to a stack in connected rundown execution is not quite the same as that of a cluster usage. To push a component onto the stack, the accompanying advances are included.

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;
        }
    }
}
```