Maharashtra State Board Of Technical Education

Government Polytechnic, Gondia

DEPARTMENT OF COMPUTER ENGINEERING (2018-19)

MICRO PROJECT REPORT

ON

"REPORT ON STACK WITH ITS IMPLEMENTATION USING LINKED LIST"

SUBMITTD BY:

- **NAINA P. BANSOD**
- **BHUSHAN G. ASATI**
- **ROSHNI T. THAKUR**
- **KHUSHAL M. GANDHE**
- **TEJAS G. FASATE**

Guided by	**: Mr. K. Kumar**
Head of Department	**: Mr. J. M. Meshram**
Principal	**: Dr. R. R. Wakodkar**

Certificate

This is to certify that the

"Micro-project title''

"REPORT ON STACK WITH ITS IMPLEMENTATION USING LINKED LIST"

Carried out under Guidance of **Mr. K. Kumar Sir** lecturer Computer department and submitted to the department of Computer Engineering.

Name of Student:

- **NAINA P. BANSOD**
- **BHUSHAN G. ASATI**
- **ROSHNI T. THAKUR**
- **KHUSHAL M. GANDHE**
- **TEJAS G. FASATE**

As the Partial fulfilment of third subject Course of **"Data Structure"** During winter 2018.

Mr. K. Kumar	**Mr. J. M. Meshram**
Guide and lecturer:	Head of Department:

Department of Computer Engineering Government Polytechnic, Gondia (Winter 2018)

Declaration

We are the student of Computer engineering hereby declare that the micro Project tittle

"REPORT ON STACK WITH ITS IMPLEMENTATION USING LINKED LIST"

Submitted to the Computer engineering department GPG for the practical work of **"Data Structure"** subject **(22317)** that the micro project has not previously formed the basic of any copyright work.

Submitted by:

Name of the Student	Enrollment No	Signature
1. Naina P. Bansod	1612420071	……………
2. Bhushan G. Asati	1612420073	...…………
3. Roshni T. Thakur	1612420077	…………….
4. Khushal M. Gandhe	1612420078	…………….
5. Tejas G. Fasate	1612420084	…………….

Acknowledgement

This is completed micro project work comes as gift to us after all the efforts that has gone in to it has been a beautiful end over only because of the valuable guidance of our guide and well Wishers.…..

We wish extend our heart full gratitude of our guide **Mr. K. Kumar** for his constant guidance , encouragement ,motivation for every stage of this work made this micro project a success…..

Finally we are proud to express our gratitude and respect to each member of this group……

Place: Government Polytechnic Gondia

Date:

CONTENT

Abstract

- Now, let's think about a stack in an abstract way. I.e., it doesn't hold any particular kind of thing (like books) and we aren't restricting ourselves to any particular programming language or any particular implementation of a stack.
- Stacks hold objects, usually all of the same type. Most stacks support just the simple set of operations we introduced above; and thus, the main property of a stack is that objects go on and come off of the *top* of the stack.
- Here are the minimal operations we'd need for an abstract stack (and their typical names):
- Push: Places an object on the top of the stack.
- Pop: Removes an object from the top of the stack and produces that object.
- Is Empty: Reports whether the stack is empty or not.
- Because we think of stacks in terms of the physical analogy, we usually draw them vertically (so the **top** is really *on top*).

Introduction

- The linked list data structure is one of the fundamental data structures in computer science.
- Think of the linked list data structure as your ABCs. Without learning the ABCs, it is difficult to conceptualize words, which are made up by stringing alphabetical characters together.
- Therefore, you want to know the ins and outs of the linked list data structures.
- In this article, we will explore the linked list data structure’s key features and operations. Afterwards, we will begin by implementing our own singly linked list.
- Another separate post will be dedicated towards the doubly linked list, which is a variant of the linked list data structure.

Abstract idea of a stack

- The stack is a very common data structure used in programs. By data structure, we mean something that is meant to hold data and provides certain operations on that data.
- One way to describe how a stack data structure behaves is to look at a physical analogy, a stack of books...
- Now, a minimal set of things that we might do with the stack are the following:
- We'll consider other, more complex operations to be inappropriate for our stack. For example, pulling out the 3rd book from the top cannot be done directly because the stack might fall over.

Order Produced by a Stack

1. Stacks are linear data structures. This means that their contexts are stored in what looks like a line (although vertically). This linear property, however, is not sufficient to discriminate a stack from other linear data structures. For example, an array is a sort of linear data structure in which you can access any element directly. In contrast, in a stack, you can only access the element at its top.

   One of the distinguishing characteristics of a stack, and the thing that makes it useful, is the order in which elements come out of a stack. Let's see what order that is by looking at a stack of letters...

   Suppose we have a stack that can hold letters, call it stack. What would a particular sequence of Push and Pops do to this stack?

   We begin with stack empty:

   ```
   -----
   stack
   ```

   Now, let's perform Push(stack, A), giving:

   ```
   -----
   | A |  <-- top
   -----
   stack
   ```

   Again, another push operation, Push(stack, B), giving:

   ```
   -----
   ```

```
| B |  <-- top
-----
| A |
-----
stack
```

Now let's remove an item, letter = Pop(stack), giving:

```
-----            -----
| A |  <-- top   | B |
-----            -----
stack            letter
```

And finally, one more addition, Push(stack, C), giving:

```
-----
| C |  <-- top
-----
| A |
-----
stack
```

You'll notice that the stack enforces a certain order to the use of its contents, i.e., the Last thing In is the First thing Out. Thus, we say that a stack enforces **LIFO** order.

Now we can see one of the uses of a stack...To reverse the order of a set of objects.

2. **Implementing a stack with a linked list:**

Using a linked list is one way to implement a stack so that it can handle essentially any number of elements.

Here is what a linked list implementing a stack with 3 elements might look like:

```
list
 |
 v
--------   --------   ---------
| C | -+-->| B | -+-->| A | 0 |
--------   --------   ---------
```

Where should we consider the **top** of the stack to be, the **beginning** or **end** of the list.

3. **C implementation**

Now, think about how to actually implement this *stack of characters* in C.

It is usually convenient to put a data structure in its own module, thus, we'll want to create files stack.h and a stack.c.

The operations needed for our stack are mainly determined by the operations provided by an abstract stack, namely:

```
StackPush()
StackPop()
```

```
StackIsEmpty()
```

We may need to add a few other operations to help implement a stack. These will become apparent as we start to implement the stack.

4. **Data types for a stack**

In order to determine what data types we'll need, let's think about how someone will use our stack:

```
1. type-of-a-stack stack1, stack2;
2. StackPush(ref-to-stack1, 'A');
3. StackPush(ref-to-stack2, 'B');
4. ch = StackPop(ref-to-stack1);
```

First, stack variables must be defined (e.g., stack1 and stack2). Then, stack operations may be called. Those operations will need to know which stack to operate on.

Thus, our goal is to get some kind of type that we can use to keep track of a stack.

Let's start bottom-up from the simplest type and work our way up through types that use the simpler types...

First, we want to write a stack that is very generic. The fact that this stack holds a character is only particular to this program. It should be easy to change the type of things held in the stack.

Let's introduce a type name for the type of thing held in the stack:

```
typedef char stackElementT;
```

Now, elements of the stack are being stored in a linked list. Recall that linked lists are made up of nodes that contain both an element and a pointer to the next node.

```
typedef struct stackTag
{
  stackElementT element;
  struct stackTag *next;
 }
 stackNodeT;
```

Finally, we need something that holds all the information needed to keep track of the stack. Since elements will be held in nodes, we only need a pointer to keep track of the beginning of the list (which will be our *top* of the stack).

We suggest that this single pointer be put in a structure, so that we can give it a descriptive field name and so that we can add more fields easily in the future (if needed):

```
typedef struct
{
  stackNodeT *top;
}
 stackT;
```

5. **Filling in stack operations**

Now that we've decided on the data types for a stack, we think we'd like to add 2 extra operations:

```
StackInit()
StackDestroy()
```

They are not part of the *abstract* concept of a stack, but they are necessary for setup and cleanup when writing the stack in C.

Now, let's think about the StackInit() function. It will need to set up a stack stackT structure, so that we have an *empty stack*.

What does the prototype for StackInit() look like? What do we do in its body?

Now, what about putting an element in the stack. What should the prototype for StackPush() look like?

The steps needed to push an element onto the stack are:

1. Allocate a new node.
2. Put the element in the node.
3. Link the element into the proper place in the list.

Let's fill in the body of StackPush().

Last, fill in the prototypes for the rest of the stack operations:

```
void        StackInit(stackT *stackPtr);
return-type StackDestroy(parameters);
void        StackPush(stackT *stackPtr,
                    stackElement(element);
        return-type StackPop(parameters);
        return-type StackIsEmpty(parameters);
```

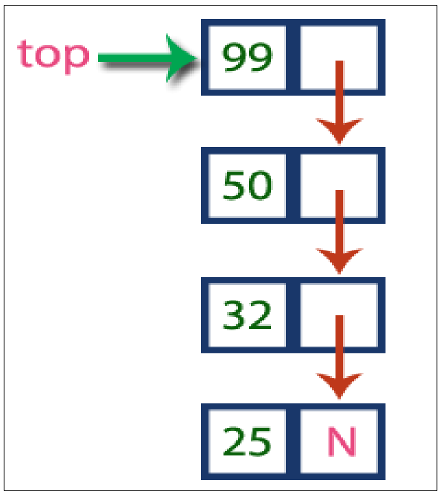
Under some circumstances, you may be able to pass a stack and not need a pointer. We prefer to always pass a pointer so that users call all stack functions in the same way.

Stack using Linked List

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Operations

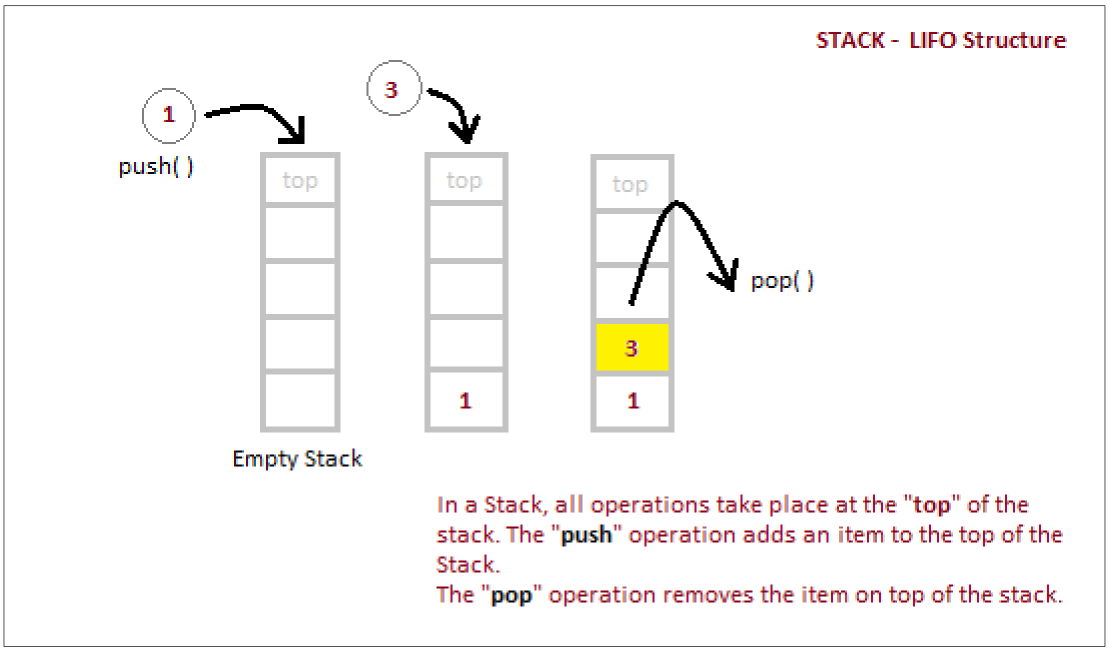
To implement stack using linked list, we need to set the following things before implementing actual operations.

- **Step 1:** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2:** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3:** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

Push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

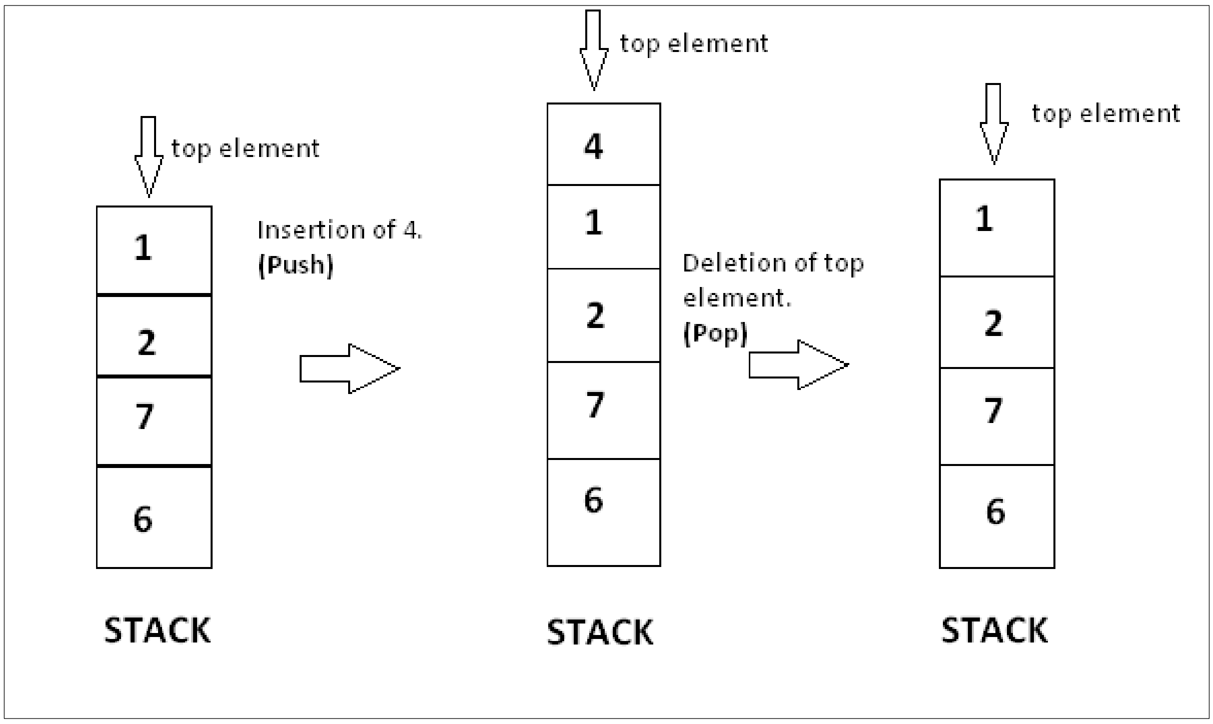
- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3:** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4:** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5:** Finally, set **top** = **newNode**.


STACK - LIFO Structure
1
push( )
3
top
top
top
1
3
1
pop( )
Empty Stack
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

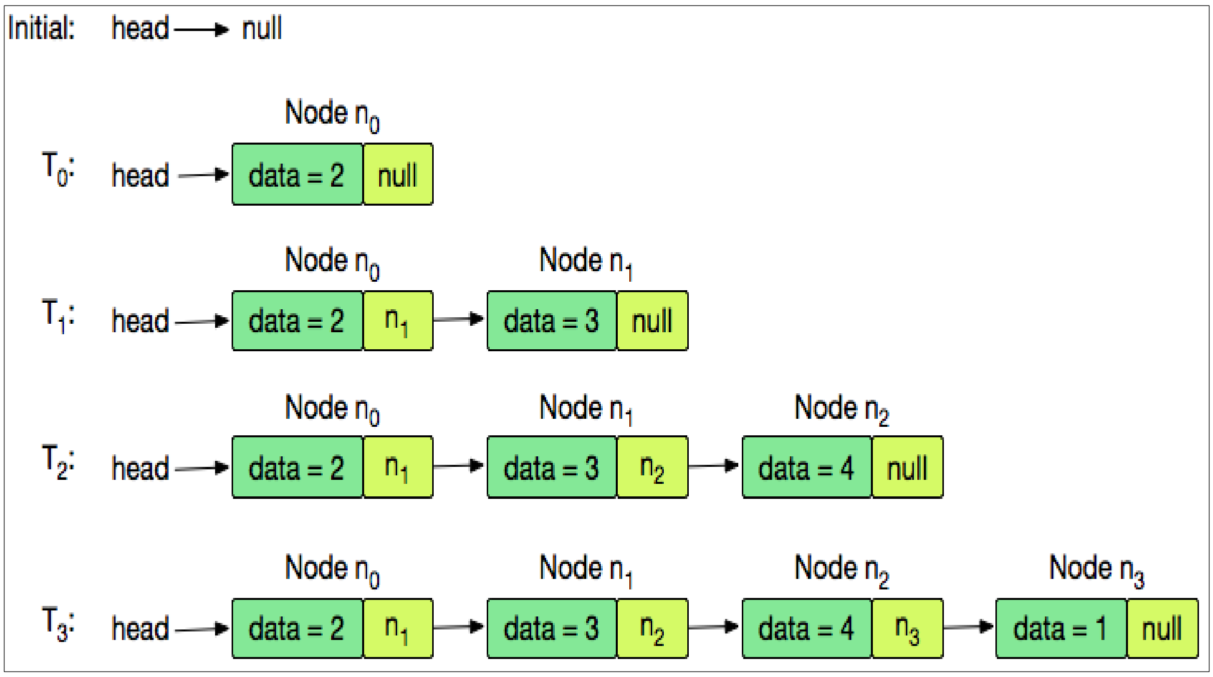
- **Step 1:** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2:** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3:** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4:** Then set '**top** = **top → next**'.
- **Step 7:** Finally, delete '**temp**' (**free(temp)**).



Display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1:** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2:** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4:** Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp → next** != **NULL**).
- **Step 4:** Finally! Display '**temp → data ---> NULL**'.


Initial: head → null
Node n0
T0: head → data = 2 | null
Node n0 Node n1
T1: head → data = 2 | n1 → data = 3 | null
Node n0 Node n1 Node n2
T2: head → data = 2 | n1 → data = 3 | n2 → data = 4 | null
Node n0 Node n1 Node n2 Node n3
T3: head → data = 2 | n1 → data = 3 | n2 → data = 4 | n3 → data = 1 | null

Source Code

```
#include <stdio.h>

#include <stdlib.h>

struct node

{

int info;

struct node *ptr;

}*top,*top1,*temp;

int topelement();

void push(int data);

void pop();

void empty();

void display();

void destroy();

void stack_count();

void create();

int count = 0;

void main()

{
```

```
int no, ch, e;

printf("\n 1 - Push");

printf("\n 2 - Pop");

printf("\n 3 - Top");

printf("\n 4 - Empty");

printf("\n 5 - Exit");

printf("\n 6 - Dipslay");

printf("\n 7 - Stack Count");

printf("\n 8 - Destroy stack");

create();

while (1)

{

printf("\n Enter choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter data : ");

scanf("%d", &no);

push(no);
```

```
break;

case 2:

pop();

break;

case 3:

if (top == NULL)

printf("No elements in stack");

else

{

e = topelement();

printf("\n Top element : %d", e);

}

break;

case 4:

empty();

break;

case 5:

exit(0);

case 6:

display();
```

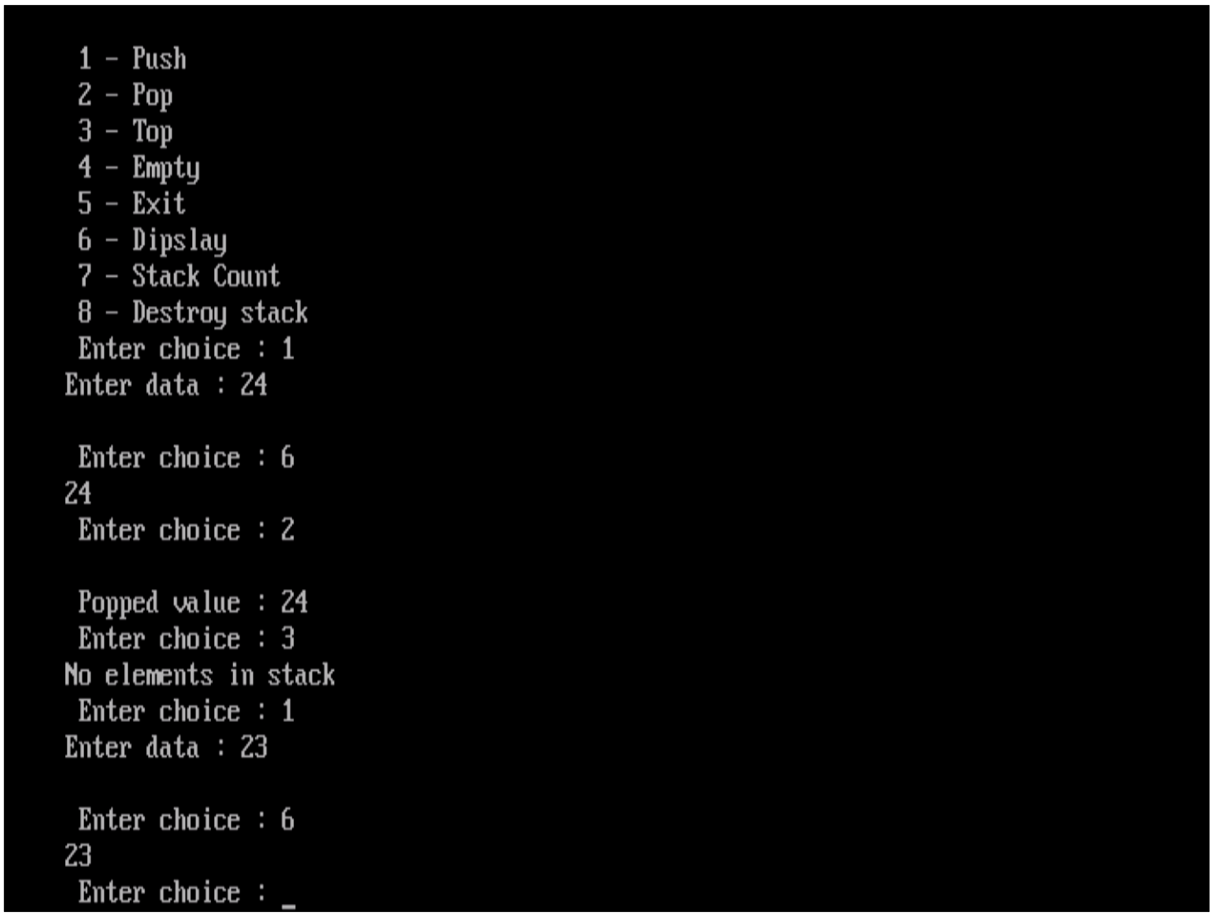
```
break;

case 7:

stack_count();

break;

case 8:

destroy();

break;

default :

printf(" Wrong choice, Please enter correct choice  ");

break;

}

}

}

void create()

{

top = NULL;

}

void stack_count()

{

printf("\n No. of elements in stack : %d", count);
```

```
}

void push(int data)

{

if (top == NULL)

{

top =(struct node *)malloc(1*sizeof(struct node));

top->ptr = NULL;

top->info = data;

}

else

{

temp =(struct node *)malloc(1*sizeof(struct node));

temp->ptr = top;

temp->info = data;

top = temp;

}

count++;

}

void display()

{
}

top1 = top;

if (top1 == NULL)

{

printf("Stack is empty");

return;

}

while (top1 != NULL)

{

printf("%d ", top1->info);

top1 = top1->ptr;

}

}

void pop()

{

top1 = top;

if (top1 == NULL)

{

printf("\n Error : Trying to pop from empty stack");

return;

}
```

```
else
top1 = top1->ptr;
printf("\n Popped value : %d", top->info);
free(top);
top = top1;
count--;
}
int topelement()
{
return(top->info);
}
void empty()
{
if (top == NULL)
printf("\n Stack is empty");
else
printf("\n Stack is not empty with %d elements", count);
}
void destroy()
{
```

```
top1 = top;

while (top1 != NULL)

{

top1 = top->ptr;

free(top);

top = top1;

top1 = top1->ptr;

}

free(top1);

top = NULL;

printf("\n All stack elements destroyed");

count = 0;

}
```

Output



```
  1 - Push
  2 - Pop
  3 - Top
  4 - Empty
  5 - Exit
  6 - Dipslay
  7 - Stack Count
  8 - Destroy stack
  Enter choice : 1
 Enter data : 24

  Enter choice : 6
 24
  Enter choice : 2

  Popped value : 24
  Enter choice : 3
 No elements in stack
  Enter choice : 1
 Enter data : 23

  Enter choice : 6
 23
  Enter choice : _
```

Conclusion

- In this lesson we learned how to traverse, append, prepend, insert and delete nodes from a linked list. These operations are quite useful in many practical applications.
- Suppose you are the programmer responsible for an application like Microsoft PowerPoint. We can think of a PowerPoint presentation as a list, whose nodes are the individual slides. In the process of creating and managing slides we can use the concepts such as inserting and deleting nodes learned in this lesson.
- Many other applications may require you to think of an abstraction like a list, where list can be easily maintained. In the next lesson we will learn more advanced list operations such as reversing a list, swapping nodes and sorting a list etc.
- Linked lists remain one of the widely used concepts in programming applications. More advanced linked list structures like an array of linked lists, or linked list of linked lists or multilinked lists can be used to implement applications that require complex data structures.

Reference

- http://btechsmartclass.com/downloads.html
- https://www.programmingsimplified.com/c/data-structures/c-program-implement-linked-list
- https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/
- https://www.geeksforgeeks.org/linked-list-set-1-introduction/
- https://www.geeksforgeeks.org/data-structures/linked-list/