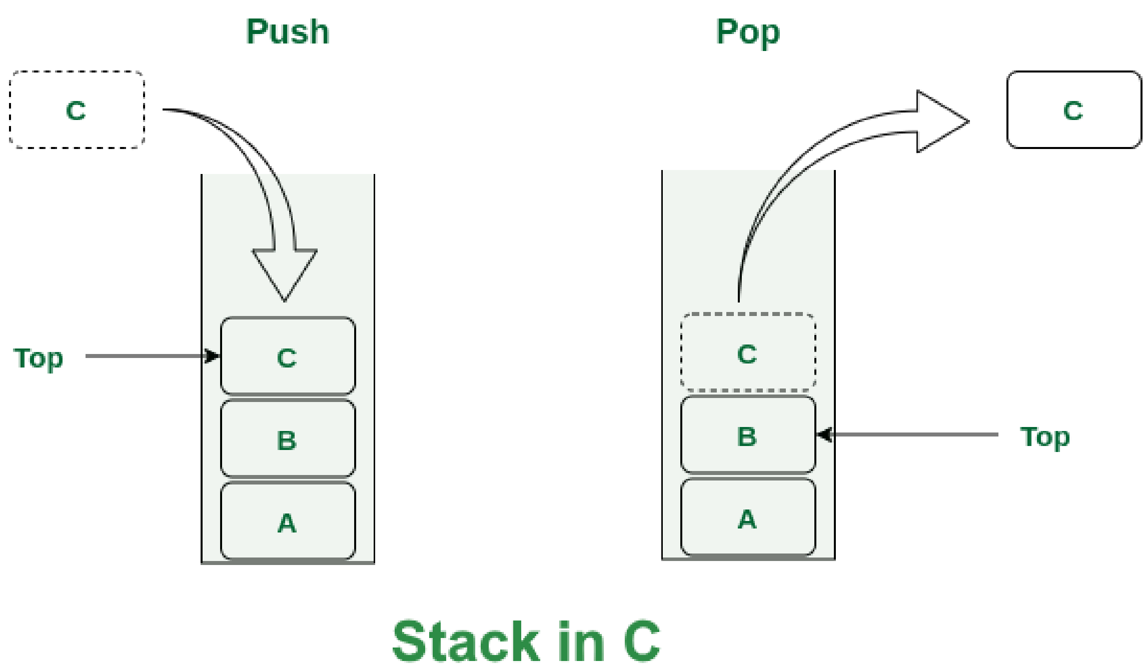
Stack Using Linked List

Stack is a linear data structure that follows the Last-In-First-Out (LIFO) order of operations. This means the last element added to the stack will be the first one to be removed. There are different ways using which we can implement stack data structure in C.

In this article, we will learn how to implement a stack using a linked list in C, its basic operation along with their time and space complexity analysis.

Implementation of Stack using Linked List in C

Stack is generally implemented using an array but the limitation of this kind of stack is that the memory occupied by the array is fixed no matter what are the number of elements in the stack. In the stack implemented using linked list implementation, the size occupied by the linked list will be equal to the number of elements in the stack. Moreover, its size is dynamic. It means that the size is gonna change automatically according to the elements present.

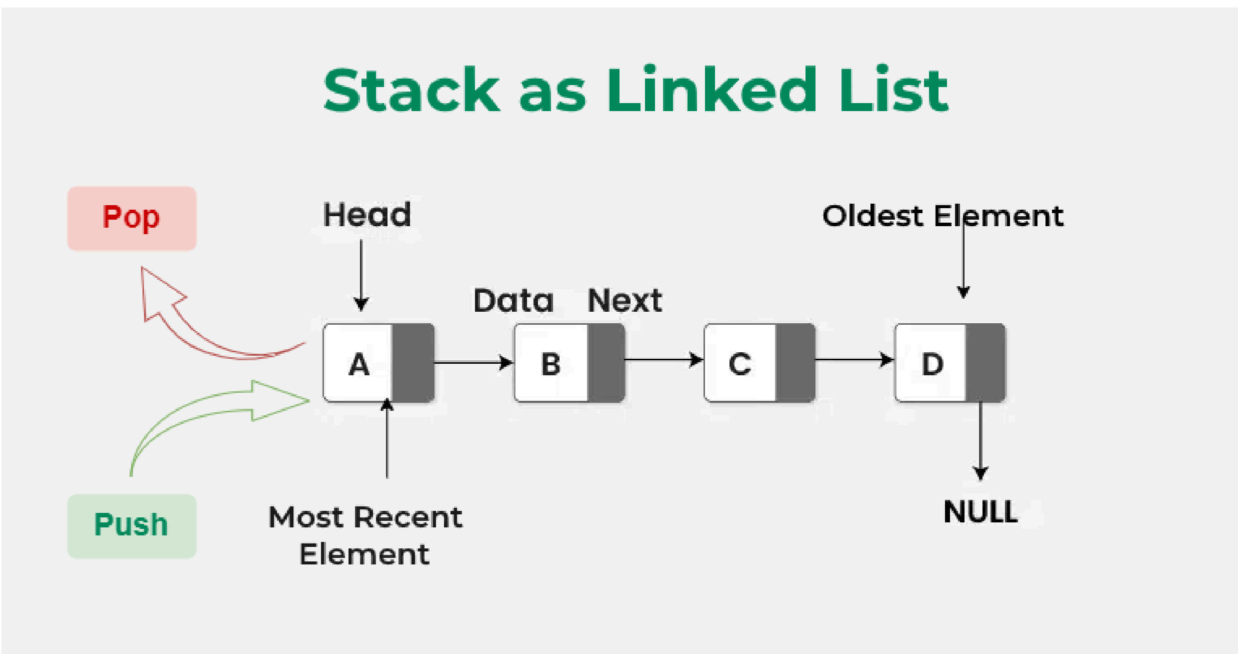


Representation of Linked Stack in C

In C, the stack that is implemented using a linked list can be represented by the pointer to the head node of the linked list. Each node in that linked list represents the element of the stack. The type of linked list here is a singly linked list in which each node consists of a data field and the next pointer.

```
struct Node {
      type data;
      Node* next;
}
```

The type of data can be defined according to the requirement.




```
// C program to implement a stack using linked list
#include <stdio.h>
#include <stdlib.h>

// ________LINKED LIST UTILITY FUNCITON____________

// Define the structure for a node of the linked list
typedef struct Node {
```

```c
    int data;
    struct Node* next;
} node;

// linked list utility function
node* createNode(int data)
{
    // allocating memory
    node* newNode = (node*)malloc(sizeof(node));

    // if memory allocation is failed
    if (newNode == NULL)
        return NULL;

    // putting data in the node
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// fuction to insert data before the head node
int insertBeforeHead(node** head, int data)
{
    // creating new node
    node* newNode = createNode(data);
    // if malloc fail, return error code
    if (!newNode)
        return -1;
```

```
    // if the linked list is empty
    if (*head == NULL) {
        *head = newNode;
        return 0;
    }

    newNode->next = *head;
    *head = newNode;
    return 0;
}

// deleting head node
int deleteHead(node** head)
{
    // no need to check for empty stack as it is already
    // being checked in the caller function
    node* temp = *head;
    *head = (*head)->next;
    free(temp);
    return 0;
}

// __________STACK IMPLEMENTATION STARTS HERE__________

// Function to check if the stack is empty or not
int isEmpty(node** stack) { return *stack == NULL; }

// Function to push elements to the stack
void push(node** stack, int data)
```

```
{
    // inserting the data at the beginning of the linked
    // list stack
    // if the insertion function returns the non - zero
    // value, it is the case of stack overflow
    if (insertBeforeHead(stack, data)) {
        printf("Stack Overflow!\n");
    }
}

// Function to pop an element from  the stack
int pop(node** stack)
{
    // checking underflow condition
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }

    // deleting the head.
    deleteHead(stack);
}

// Function to return the topmost element of the stack
int peek(node** stack)
{
    // check for empty stack
    if (!isEmpty(stack))
        return (*stack)->data;
    else
```

```c
        return -1;
}

// Function to print the Stack
void printStack(node** stack)
{
    node* temp = *stack;
    while (temp != NULL) {
        printf("%d-> ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// driver code
int main()
{
    // Initialize a new stack top pointer
    node* stack = NULL;

    // Push elements into the stack
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40);
    push(&stack, 50);

    // Print the stack
    printf("Stack: ");
```

```
    printStack(&stack);
    // Pop elements from the stack
    pop(&stack);
    pop(&stack);
    // Print the stack after deletion of elements
    printf("\nStack: ");
    printStack(&stack);

    return 0;
}
```

Output

```
Stack: 50-> 40-> 30-> 20-> 10->


Stack: 30-> 20-> 10->
```

Benifits of Linked List Stack in C

The following are the major benifits of the linked list implementation over the array implementation:

1. The dynamic memory management of linked list provide dynamic size to the stack that changes with the change in the number of elements.
2. Rarely reaches the condition of the stack overflow.