Data Structures and Algorithms

Lecture 4

© Fall 2025- **Dr. Hesham Sakr**
Building, office No. 12
Hesham.sakr@sut.edu.eg

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

Data Structures and Algorithms

Learning Objectives:

- Representation of Stack (or) Implementation of stack
- Stack using linked list
- Linked list implementation of stack
- Adding a node to the stack (Push operation)
- Deleting a node from the stack (POP operation)
- Display the nodes (Traversing)

DATA STRUCTURES

Representation of Stack (or) Implementation of stack

The stack should be represented in two ways:

1. Stack using array

2. Stack using linked list

2. Stack using linked list:

- We can represent a stack as a linked list.
- In a stack push and pop operations are performed at one end called top.
- We can perform similar operations at one end of list using top pointer.

The linked stack looks as shown in figure

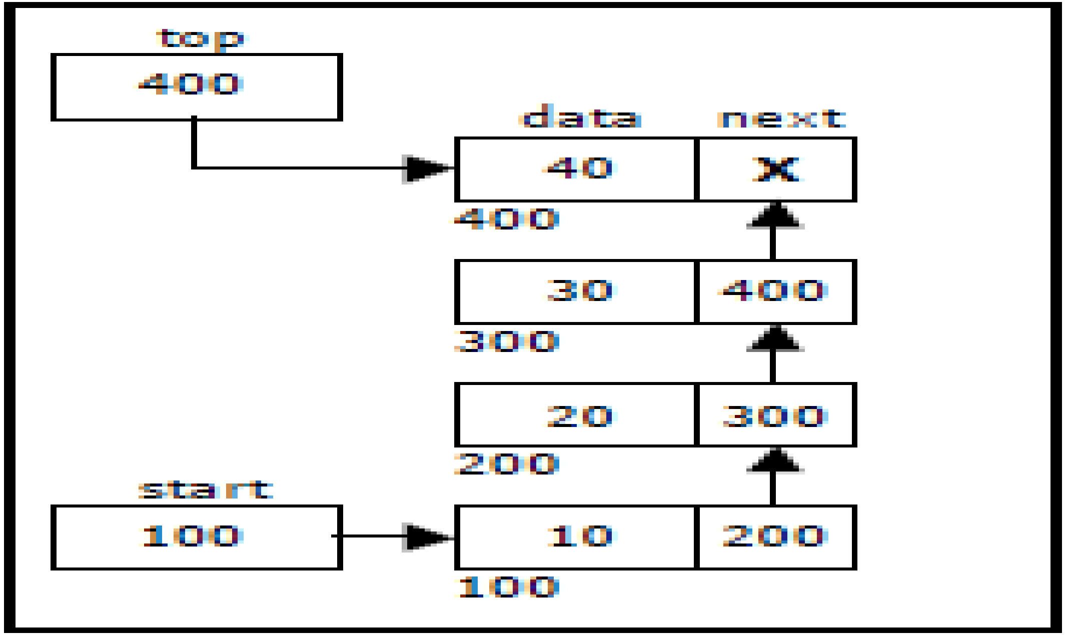


Figure Linked stack representation

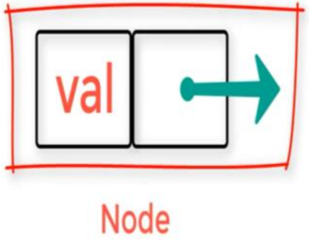
ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

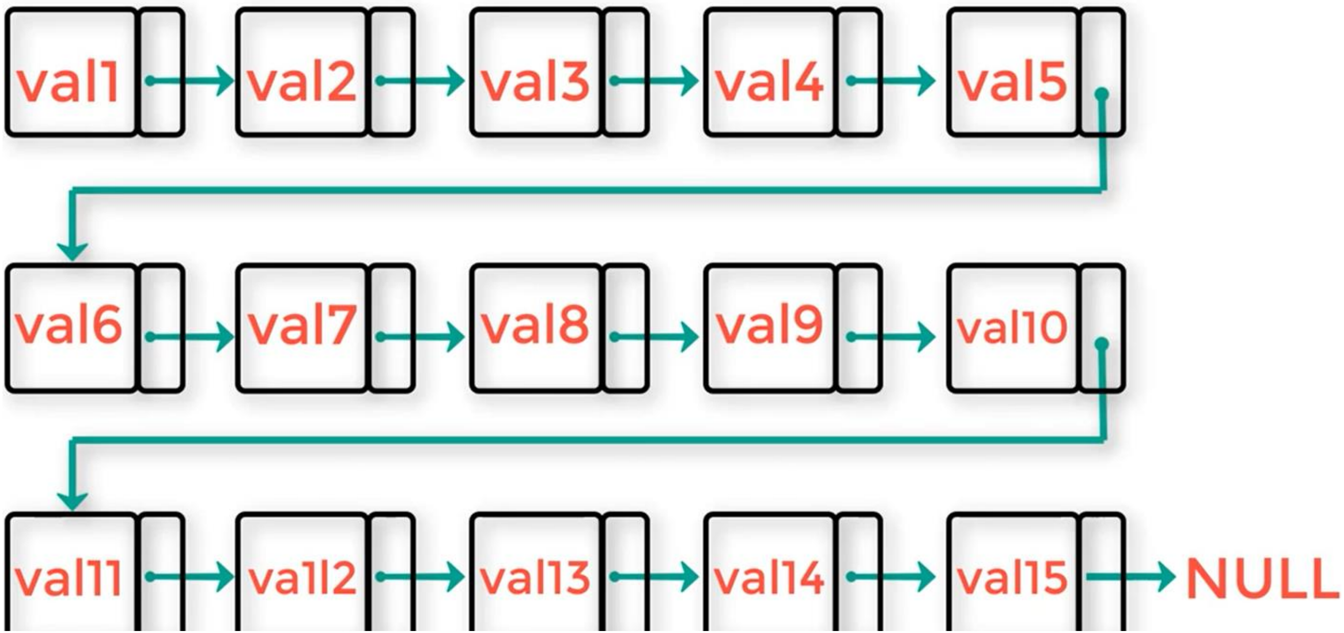
Linked list implementation of stack

Pointers in C++

- Pointers are variables that store the memory address of another variable.
- Syntax: int* ptr = &var; // ptr holds the address of var
- Dereferencing: *ptr returns the value stored at the memory address ptr points to.

Nodes


val
Node


val1
val2
val3
val4
val5
val6
val7
val8
val9
val10
val11
val12
val13
val14
val15
NULL

Basic Pointer Example

```
#include <iostream>
using namespace std;

int main() {
    int var = 10;
    int* ptr = &var;

    cout << "Value of var: " << var << endl;
    cout << "Address of var: " << &var << endl;
    cout << "Pointer ptr stores: " << ptr << endl;
    cout << "Pointer dereferenced value: " << *ptr << endl;

    return 0;
}
```

Benefits of Using Pointers in Linked List vs Arrays for Stacks

- 1. Dynamic Memory Allocation: Memory is allocated as needed in linked lists.
- 2. Efficient Memory Usage: Memory is allocated only when required.
- 3. No Size Limitation: Limited by system memory, unlike arrays.
- 4. Faster Insertion and Deletion: Only pointer updates required.
- 5. No Wasted Space: Memory is not pre-allocated, avoiding wastage.

Linked list implementation of stack:

- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically.

Linked list implementation of stack:

- However, time complexity in both the scenario is same for all the operations i.e. push, pop.

- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.

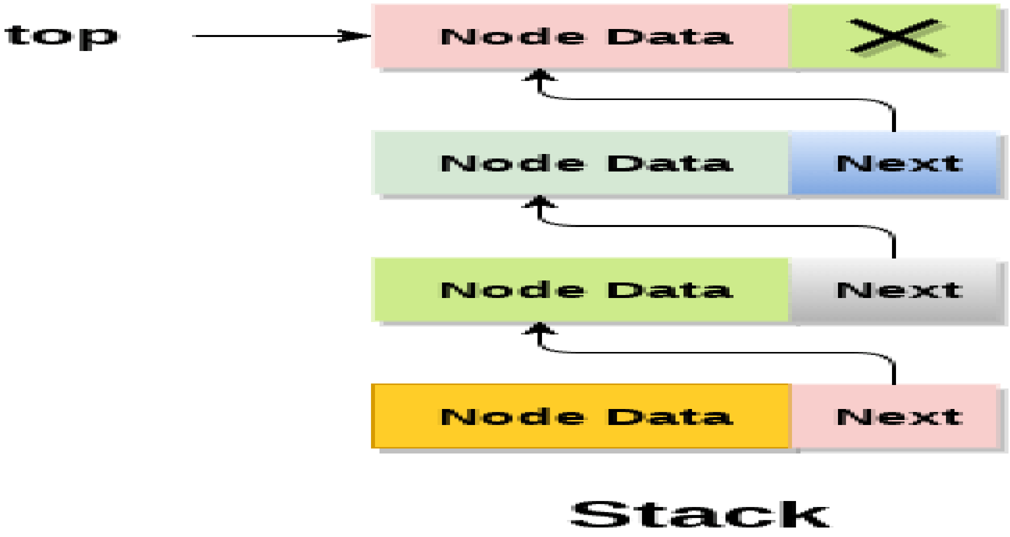
Linked list implementation of stack:

- Each node contains a pointer to its immediate successor node in the stack.

- Stack is said to be overflown in Linked list implementation if the space left in the memory heap is not enough to create a node.

Linked list implementation of stack:

- The top most node in the stack always contains null in its address field.



The linked stack looks as shown in figure

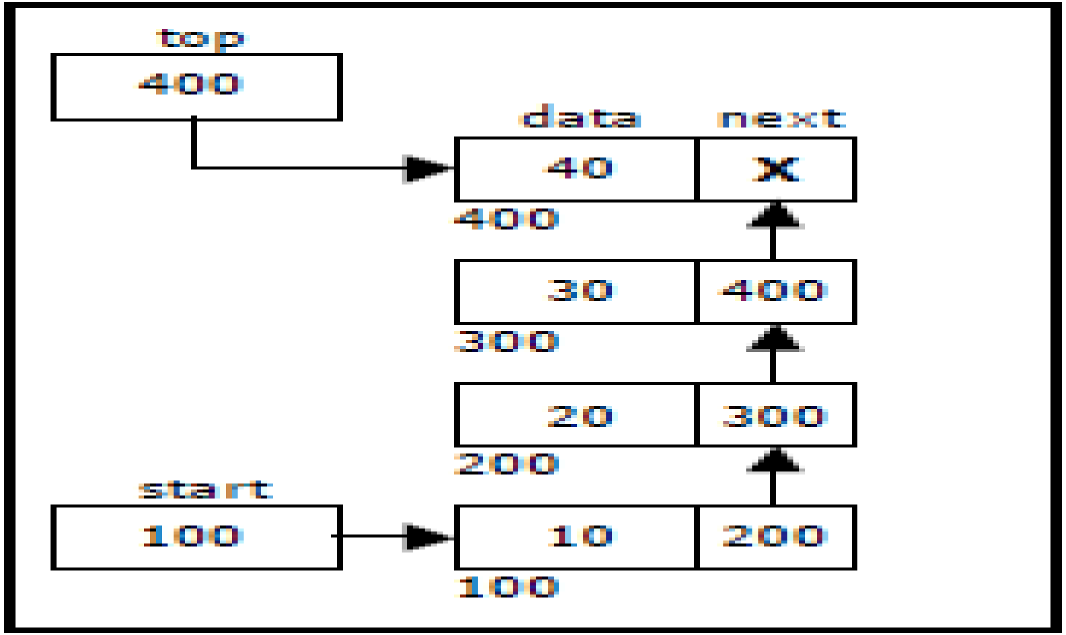


Figure Linked stack representation

Adding a node to the stack (Push operation)

Adding a node to the stack (Push operation):

- Adding a node to the stack is referred to as **push** operation.

- Pushing an element to a stack in linked list implementation is different from that of an array implementation.

- In order to push an element onto the stack, the following steps are involved.

Adding a node to the stack (Push operation):

1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list.

This includes:

- assigning value to the data part of the node, and
- assign null to the address part of the node.

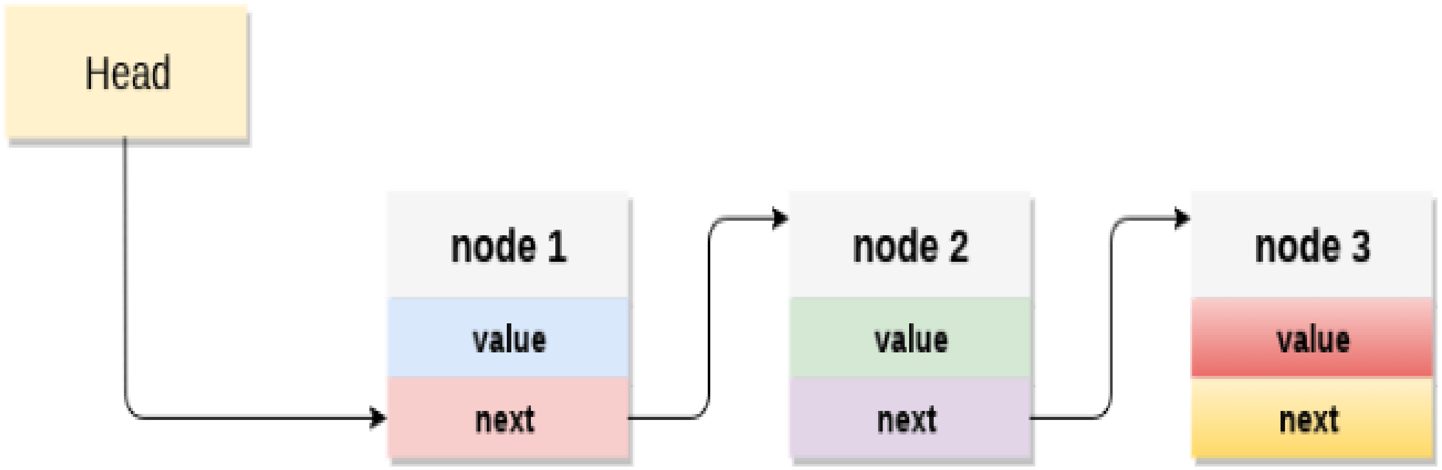
Adding a node to the stack (Push operation):

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack).

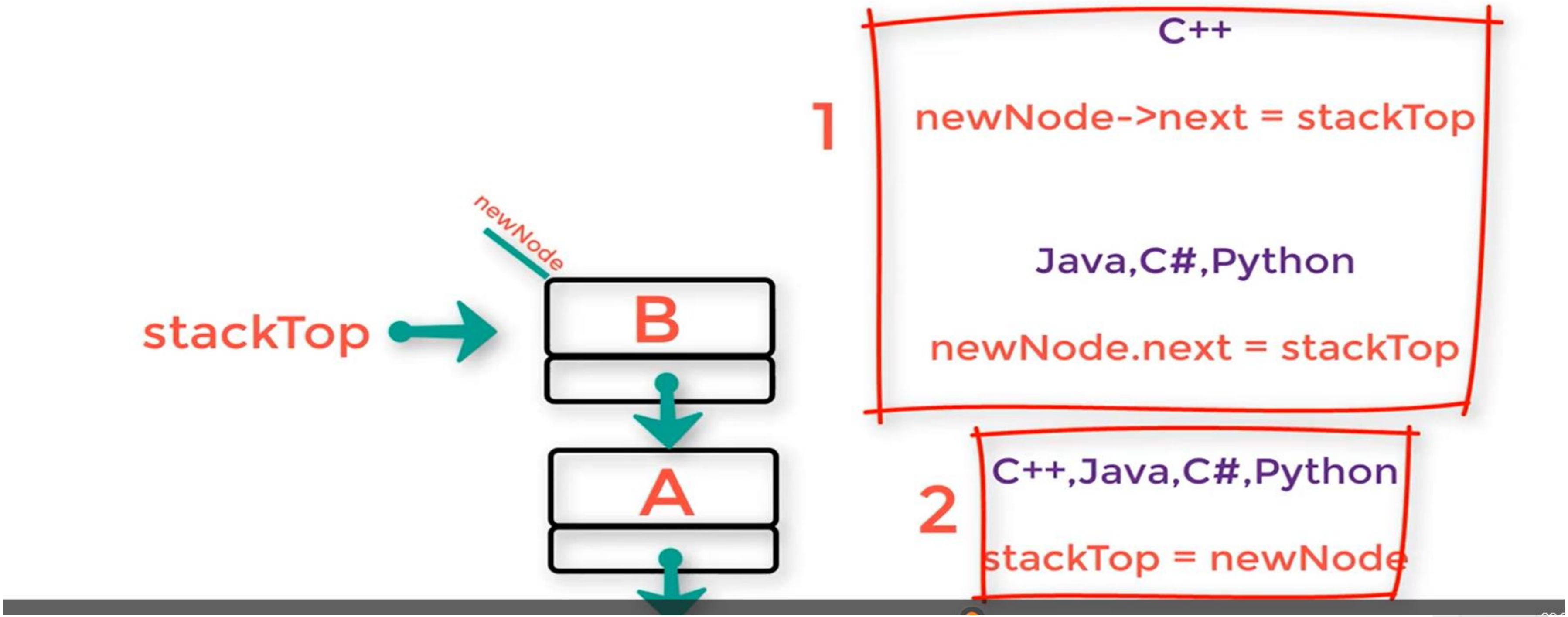
For this purpose,

- assign the address of the starting element to the address field of the new node, and
- make the new node, the starting node of the list.

Time Complexity : o(1)



Push Operation



Deleting a node from the stack (POP operation)

Deleting a node from the stack (POP operation):

- Deleting a node from the top of stack is referred to as **pop** operation.
- Deleting a node from the linked list implementation of stack is different from that in the array implementation.

Deleting a node from the stack (POP operation):

In order to pop an element from the stack, we need to follow the following steps :

- Check for the underflow condition
- Adjust the head pointer accordingly

- Time Complexity : o(n)

Deleting a node from the stack (POP operation):

Check for the underflow condition:

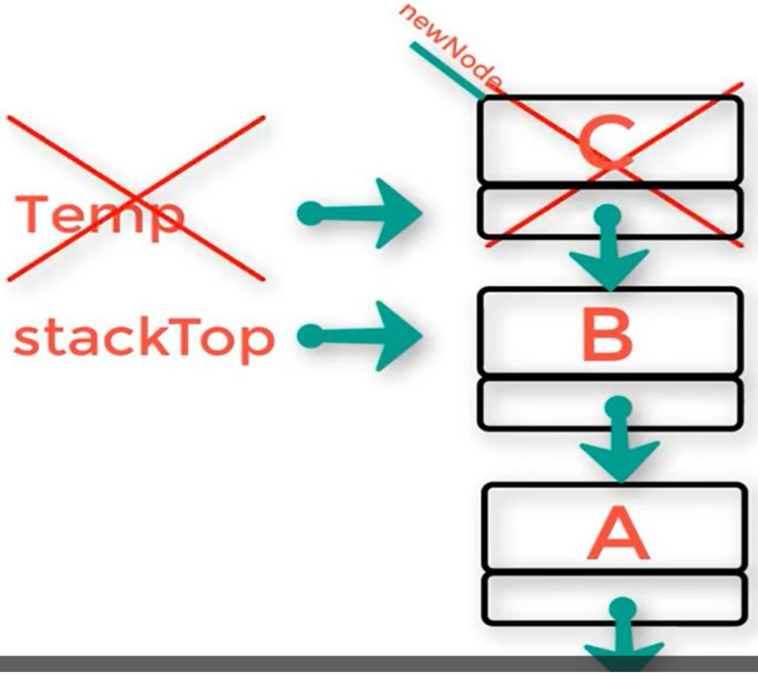
- The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.

Deleting a node from the stack (POP operation):

Adjust the head pointer accordingly:

- In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Pop Operation



c++,Java,C#,Python

temp = stackTop

c++

stackTop = stackTop->next;

Java,C#,Python

stackTop = stackTop.next

Display the nodes (Traversing)

Display the nodes (Traversing):

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack.

Traversing عبور: It is used to access each data item exactly once so that it can be processed.

Display the nodes (Traversing):

To Display the nodes follow the following steps:

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

- Time Complexity : o(n)

Linked List Node Structure

```
#include <iostream>
using namespace std;

// Node structure for Linked List
struct Node {
    int data;
    Node* next;
};

// Head of the linked list (initially empty)
Node* head = nullptr;
```

1.Push Operation (Insert element at the top of the list)

- This function adds an element to the top of the stack (beginning of the linked list).

```
void push(int value) {
    // Create a new node
    Node* newNode = new Node();
    newNode->data = value;

    // New node's next is the current head
    newNode->next = head;

    // Move the head pointer to the new node
    head = newNode;

    cout << value << " pushed onto the stack." << endl;
```

Pop Operation (Remove the element from the top of the list)

This function removes the top element from the stack (the first element in the linked list).

```
void pop() {
    if (head == nullptr) {
        cout << "Stack is empty, nothing to pop." << endl;
        return;
    }

    // Store the current head node
    Node* temp = head;

    // Move head to the next node
    head = head->next;

    cout << temp->data << " popped from the stack." << endl;

    // Free the old head node
    delete temp;
```

Main Function to Illustrate Operations

```
int main() {
    push(10);
    push(20);
    push(30);  // Stack: 30 -> 20 -> 10

    getTop();  // Should display 30

    pop();     // Removes 30; Stack: 20 -> 10
    getTop();  // Should display 20

    pop();     // Removes 20; Stack: 10
    getTop();  // Should display 10

    pop();     // Removes 10; Stack is empty
    getTop();  // Should indicate the stack is empty

    pop();     // Stack is already empty, should handle the case

    return 0;
```

Explanation

- **Push**: Each new node is inserted at the beginning, making it the new head of the list.
- **Pop**: The head of the list (top of the stack) is removed, and the second node becomes the new head.
- **GetTop**: This simply returns the value of the node at the head without removing it.

Full Program for Push operation

```
#include <iostream>
using namespace std;

// Node structure for Linked List
struct Node {
    int data;           // To store the value of the node
    Node* next;         // Pointer to the next node in the list
};

// Head pointer for the linked list (initially empty)
Node* head = nullptr;

// Function to push a new element to the top of the stack
void push(int value) {
    // Step 1: Create a new node
    Node* newNode = new Node();         // Allocate memory for a new
node
    newNode->data = value;              // Assign value to the new
node
    cout << "Created new node with value: " << value << endl;

    // Step 2: Set the new node's next pointer to point to the
current head
    newNode->next = head;               // The current head is now
the second element
    cout << "New node's next set to point to the current head."
<< endl;

    // Step 3: Update the head pointer to point to the new node
    head = newNode;                     // New node becomes the head
    cout << value << " pushed onto the stack. Head updated to
the new node." << endl;
}

// Function to display the current stack
void display() {
    Node* temp = head;                  // Start from the head node
    cout << "Stack elements: ";
    while (temp != nullptr) {           // Traverse the list until
the end
        cout << temp->data << " ";      // Print the value of each
node
        temp = temp->next;              // Move to the next node
    }
    cout << endl;
}
```

Full Program for POP Operation

```
#include <iostream>
using namespace std;

// Node structure for Linked List
struct Node {
    int data;           // To store the value of the node
    Node* next;         // Pointer to the next node in the list
};

// Head pointer for the linked list (initially empty)
Node* head = nullptr;

// Function to pop the top element from the stack
void pop() {
    // Step 1: Check if the stack is empty
    if (head == nullptr) {                          // If head is null, the
stack is empty
        cout << "Stack is empty, nothing to pop." << endl;
        return;
    }

    // Step 2: Store the current head node in a temporary pointer
    Node* temp = head;                              // Store current head in
temp

    // Step 3: Move the head to the next node
    head = head->next;                              // Update head to point to
the next node
    cout << temp->data << " popped from the stack." << endl;

    // Step 4: Free the old head node
    delete temp;                                    // Delete the node that was
removed
}

// Function to push a new element to the stack (to test pop operation)
void push(int value) {
    Node* newNode = new Node();     // Create a new node
    newNode->data = value;          // Set the value
    newNode->next = head;           // Set next to current head
    head = newNode;                 // Update head to new node
    cout << value << " pushed onto the stack." << endl;
}

// Function to display the current stack
void display() {
    Node* temp = head;              // Start from the head node
    cout << "Stack elements: ";
    while (temp != nullptr) {       // Traverse the list until the end
        cout << temp->data << " ";  // Print the value of each node
        temp = temp->next;          // Move to the next node
    }
```

2. Push Operation (Generic Version)

```
#include <iostream>
using namespace std;

// Node structure for a generic Linked List using templates
template <typename T>
struct Node {
    T data;             // To store the value of any type (int, float,
char, etc.)
    Node* next;         // Pointer to the next node in the list
};

// Head pointer for the linked list (initially empty)
template <typename T>
Node<T>* head = nullptr;

// Function to push a new element of any type to the top of the stack
template <typename T>
void push(T value) {
    // Step 1: Create a new node
    Node<T>* newNode = new Node<T>();    // Allocate memory for a new
node
    newNode->data = value;               // Assign the value to the new
node
    cout << "Created new node with value: " << value << endl;

    // Step 2: Set the new node's next pointer to point to the current
head
    newNode->next = head<T>;             // The current head is now the
second element
    cout << "New node's next set to point to the current head." <<
endl;

    // Step 3: Update the head pointer to point to the new node
    head<T> = newNode;                   // New node becomes the head
    cout << value << " pushed onto the stack. Head updated to the new
node." << endl;
}

// Function to display the current stack
template <typename T>
void display() {
    Node<T>* temp = head<T>;              // Start from the head node
    cout << "Stack elements: ";
    while (temp != nullptr) {             // Traverse the list until
the end
        cout << temp->data << " ";        // Print the value of each
node
        temp = temp->next;                // Move to the next node
    }
    cout << endl;
}
```

Explanation:

- **Template definition**: We define the node structure and functions using the template `template <typename T>`, where `T` can be any data type (`int`, `float`, `char`, `double`, etc.).
- **Push function**: The function works with any data type. We use `Node<T>` to ensure the node can store any type of data.
- **Display function**: This function also works generically for any data type.
- In the `main` function, we push and display different types of data (`int`, `float`, `char`, and `double`).

Pop Operation (Generic Version)

```
#include <iostream>
using namespace std;

// Node structure for a generic Linked List using templates
template <typename T>
struct Node {
    T data;            // To store the value of any type (int, float,
char, etc.)
    Node* next;        // Pointer to the next node in the list
};

// Head pointer for the linked list (initially empty)
template <typename T>
Node<T>* head = nullptr;

// Function to pop the top element of any type from the stack
template <typename T>
void pop() {
    // Step 1: Check if the stack is empty
    if (head<T> == nullptr) {                // If head is null, the stack
is empty
        cout << "Stack is empty, nothing to pop." << endl;
        return;
    }

    // Step 2: Store the current head node in a temporary pointer
    Node<T>* temp = head<T>;                 // Store current head in temp

    // Step 3: Move the head to the next node
    head<T> = head<T>->next;                 // Update head to point to
the next node
    cout << temp->data << " popped from the stack." << endl;

    // Step 4: Free the old head node
    delete temp;                             // Delete the node that was
removed
}

// Function to push a new element of any type to the stack
template <typename T>
void push(T value) {
    Node<T>* newNode = new Node<T>();        // Create a new node
    newNode->data = value;                   // Set the value
    newNode->next = head<T>;                 // Set next to current head
    head<T> = newNode;                       // Update head to new node
    cout << value << " pushed onto the stack." << endl;
}
```

Pop Operation (Generic Version)

```
// Function to display the current stack
template <typename T>
void display() {
    Node<T>* temp = head<T>;                  // Start from the head node
    cout << "Stack elements: ";
    while (temp != nullptr) {                 // Traverse the list until
the end
        cout << temp->data << " ";            // Print the value of each
node
        temp = temp->next;                    // Move to the next node
    }
    cout << endl;
}

int main() {
    // Push different types of data into the stack
    push<int>(10);
    push<float>(20.5f);
    push<char>('A');
    push<double>(30.99);

    display<int>();             // Display stack for integers
    display<float>();           // Display stack for floats
    display<char>();            // Display stack for chars
    display<double>();          // Display stack for doubles

    // Pop different types of data from the stack
    pop<int>();                 // Pop an integer
    pop<float>();               // Pop a float
    pop<char>();                // Pop a char
    pop<double>();              // Pop a double

    return 0;
}
```

Explanation:

- **Pop function**: Like the `push` function, the `pop` function is also generic and works with any data type.
- **Template usage**: In the `main` function, we demonstrate `pop` and `push` for different types of data (`int`, `float`, `char`, and `double`).

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

Thank You