Batch: D3	**Roll No.:16010123294**
Experiment / assignment / tutorial No. 4	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Dynamic implementation of Stack- Creation, Insertion, Deletion, Peek

Objective: To implement Basic Operations of Stack dynamically

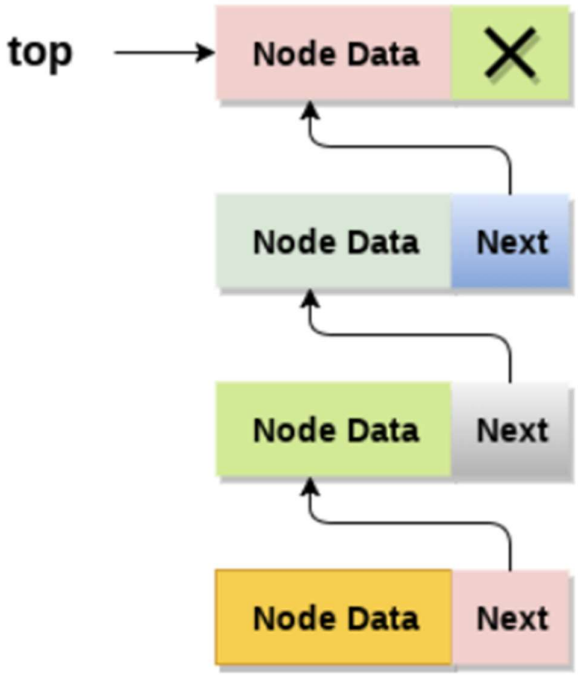
Expected Outcome of Experiment:

CO	Outcome
2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:

Introduction:

(explain with diagram linked list representation of stack)



Stack

A **Stack** is a data structure that follows the Last In, First Out (LIFO) principle, where the last element added is the first one to be removed. While a stack can be implemented using arrays, a more dynamic approach involves using a linked list. Here's how it works:

Key Concepts:

- **Node**: The basic unit of a linked list that contains two fields: data and a reference (or link) to the next node.
- **Top**: A pointer that keeps track of the last inserted node, i.e., the top of the stack.

Operations:

Push())_
Pop()

Traversal()
Peek()

Linked List ADT:

Node Structure:

- **Data**: Stores the actual value (e.g., an integer, string, etc.).
- **Link/Pointer**: Stores the address of the next node in the list.

Linked List Structure:

- **Head/Start**: A pointer that references the first node in the list. If the list is empty, the head is NULL.

Basic Operations

- **Create List**:
  - Initialize the head of the list to NULL.
- **Insert Element**:
  - **At the Start**: Insert a new node at the beginning of the list.
  - **At the End**: Insert a new node at the end of the list.
  - **At a Specific Position**: Insert a new node at a specified position in the list.
- **Delete Element**:
  - **From the Start**: Remove the first node from the list.
  - **From the End**: Remove the last node from the list.
  - **From a Specific Position**: Remove the node at a specified position in the list.
- **Traverse List**:
  - Visit each node in the list, starting from the head, and perform an operation (e.g., printing the node's data).
- **Search Element**:

- Find a node containing a specific value. If found, return its position or a reference to the node; otherwise, indicate that the element is not present.

- **Check if Empty**:
  - Deermine whether the linked list is empty (i.e., whether the head is NULL).

- **Get Length**:
  - Calculate and return the number of nodes in the list.

Key Characteristics

- **Dynamic Size**: The size of a linked list can grow or shrink dynamically, unlike arrays with fixed sizes.

- **Non-contiguous Memory Allocation**: Nodes are not stored contiguously in memory; they are linked using pointers.

- **Efficient Insertion/Deletion**: Insertion and deletion operations can be performed efficiently, particularly at the beginning or end of the list.

Types of Linked Lists

- **Singly Linked List**: Each node has a single pointer pointing to the next node.
- **Doubly Linked List**: Each node has two pointers: one pointing to the next node and one to the previous node.
- **Circular Linked List**: The last node points back to the first node, forming a circle.

Algorithm for creation, insertion, deletion, traversal and searching an element in dynamic stack using linked list:

1.**Creation of Stack**

Algorithm:

- Initialize Top pointer to NULL.
- The stack is now ready for operations.

Explanation: The stack is created by setting the Top pointer to NULL, indicating that the stack is empty.

2. Insertion (Push Operation)

Algorithm:

- Create a new node new_node.

- Assign the data to new_node.
- Set new_node->next to Top.
- Update Top to point to new_node.
- The new node is now the top of the stack.

Explanation: A new node is created and inserted at the top of the stack, with its next pointer referencing the previous top node.

3. Deletion (Pop Operation)

Algorithm:

- Check if the stack is empty (Top == NULL). If yes, return an error or a message indicating the stack is empty.
- Create a temporary node temp and assign it to Top.
- Update Top to Top->next.
- Free the memory of the temp node.
- The top node is now removed, and the next node in the stack becomes the new top.

Explanation: The top node is removed by moving the Top pointer to the next node, and the previous top node's memory is deallocated.

4. Traversal

Algorithm:

- Initialize a temporary pointer temp to Top.
- If Top is NULL, print "Stack is empty" and exit.
- While temp is not NULL:
- Print the data of the temp node.
- Move temp to temp->next.
- End the loop when temp becomes NULL.

Explanation: This algorithm visits each node in the stack, starting from the top, and prints the data stored in each node.

5. Searching for an Element

Algorithm:

- Initialize a temporary pointer temp to Top.
- If Top is NULL, print "Stack is empty" and exit.
- While temp is not NULL:
- Compare temp->data with the target element.
- If they are equal, print "Element found" and exit.

Otherwise, move temp to temp->next.

f the loop ends and the element is not found, print "Element not found".

Explanation: This algorithm searches for a specific element by traversing the stack from the top to the bottom, checking each node's data.

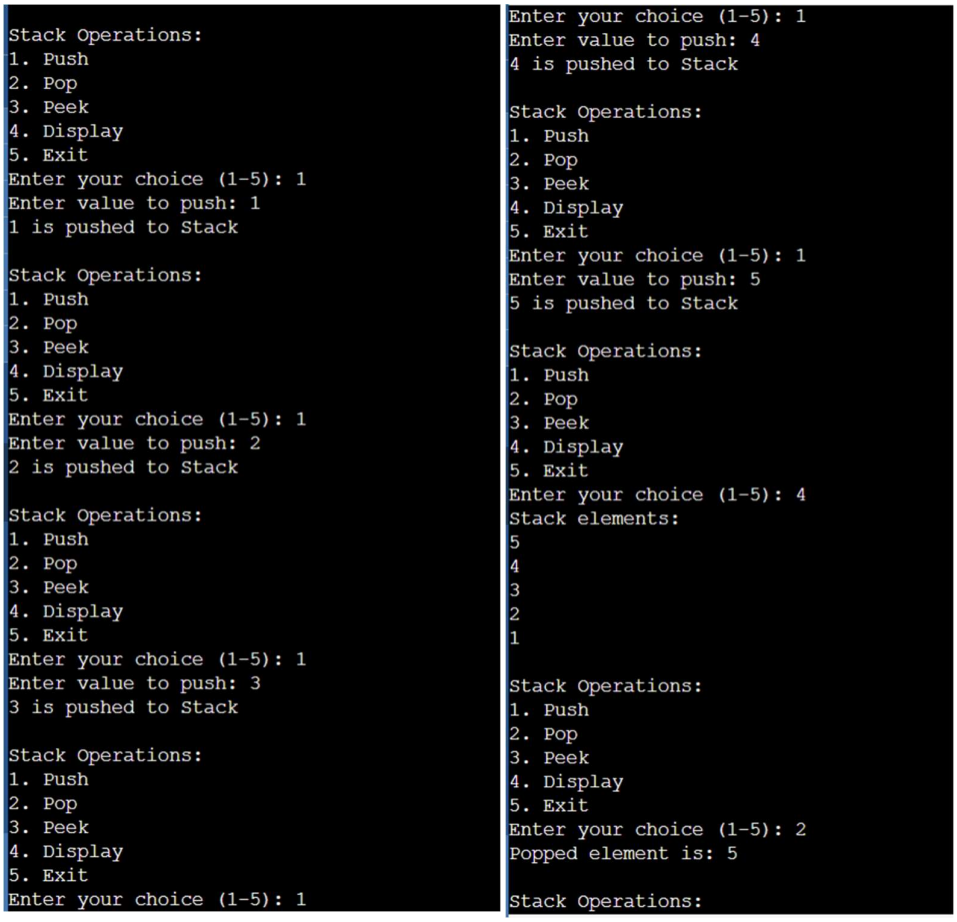
Program source code:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node* next;
} Node;

Node* top = NULL;

int isEmpty() {
   return top == NULL;
}

void push(int num) {

   Node* newNode = (Node*)malloc(sizeof(Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      return;
   }

   newNode->data = num;
   newNode->next = top;

   top = newNode;
   printf("%d is pushed to Stack\n", num);
}

void pop() {
   if (isEmpty()) {
      printf("Stack Underflow\n");
   } else {
      Node* temp = top;
```

```
        printf("Popped element is: %d\n", temp->data);
        top = top->next;
        free(temp);
    }
}

void display() {
    if (isEmpty()) {
        printf("Stack is empty\n");
    } else {
        Node* temp = top;
        printf("Stack elements:\n");
        while (temp != NULL) {
            printf("%d\n", temp->data);
            temp = temp->next;
        }
    }
}

void peek() {
    if (isEmpty()) {
        printf("Stack is empty\n");
    } else {
        printf("Top element is %d\n", top->data);
    }
}

int main() {
    int choice, num;

    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice (1-5): ");
        scanf("%d", &choice);
```

```
    switch (choice) {
      case 1:
        printf("Enter value to push: ");
        scanf("%d", &num);
        push(num);
        break;
      case 2:
        pop();
        break;
      case 3:
        peek();
        break;
      case 4:
        display();
        break;
      case 5:
        printf("Exiting the Program\n");

        while (!isEmpty()) {
          pop();
        }
        exit(0);
      default:
        printf("Invalid choice! Please enter a number between 1 and 5.\n");
    }
  }

  return 0;
}
```

Output Screenshots:



```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice (1-5): 4
Stack elements:
4
3
2
1

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice (1-5): 3
Top element is 4

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice (1-5): 5
Exiting the Program
Popped element is: 4
Popped element is: 3
Popped element is: 2
Popped element is: 1
```

Conclusion:-

Learned Implementation of stack with single linked list and applied it in C programming for Push pop peek and display options of Stack.

Post lab questions:

Explain how Stacks can be used in Backtracking algorithms with examples.

Backtracking is a problem-solving technique that involves exploring possible solutions incrementally and abandoning ("backtracking") a path as soon as it is determined that it cannot lead to a valid solution. Stacks play a crucial role in implementing backtracking algorithms by keeping track of the choices made at each step, enabling the algorithm to revert to a previous state when needed.

How Stacks Are Used in Backtracking:

- **Choice Storage**: As the algorithm explores different paths, each choice (or state) is pushed onto the stack.
- **Backtracking**: If a path is found to be invalid or a dead end, the last choice (state) is popped from the stack, effectively reverting the algorithm to the previous state.
- **Exploration**: The algorithm then tries the next possible choice from the last valid state.

Example : N-Queens Problem

In the N-Queens problem, the goal is to place N queens on an NxN chessboard such that no two queens attack each other.

- **Stack Use**: Each time a queen is placed on the board, its position is pushed onto the stack.
- **Backtracking**: If placing a queen leads to a conflict, the last queen's position is popped from the stack, and the algorithm tries the next possible position for that queen

2. Discuss the advantages and disadvantages of using static & dynamic memory allocation for implementing a stack. How does the fixed size of the stack impact its performance and usability?

Static Memory Allocation

Advantages:

Predictable Performance: Memory is allocated once, leading to consistent performance.

Simplicity: Easier to implement with no runtime memory management.

Lower Overhead: No need for dynamic allocation or deallocation.

Disadvantages:

Fixed Size: Limited to a predetermined size, leading to potential stack overflow if exceeded.

Inflexibility: Cannot adjust size based on runtime needs.

Potential Wastage: May waste memory if the allocated size is too large.

Dynamic Memory Allocation

Advantages:

Flexibility: Can grow or shrink as needed during runtime.

Efficient Memory Use: Allocates just enough memory based on actual usage.

Adaptability: Adjusts to varying workload requirements.

Disadvantages:

Overhead: Involves runtime costs for allocation and deallocation.

Complexity: Requires careful management to avoid issues like memory leaks.

Fragmentation: Can lead to inefficient memory use and performance issues over time.

Impact of Fixed Stack Size

Performance:

Static Stack: Predictable and fast but risks overflow if size is exceeded.

Dynamic Stack: More flexible and adaptable but can suffer from overhead and potential fragmentation.

Usability:

Static Stack: Simple but rigid; unsuitable for varying workloads.

Dynamic Stack: More versatile; adjusts to needs but requires careful management.