Batch: A1	**Roll No.: 16010123012**
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:

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan
4. https://www.cprogramming.com/tutorial/computersciencetheory/stack.html
5. https://www.geeksforgeeks.org/stack-data-structure-introduction-program/
6. https://www.thecrazyprogrammer.com/2013/12/c-program-for-array-representation-of-stack-push-pop-display.html

Abstract:

A Stack is an ordered collection of elements , but it has a special feature that deletion and insertion of elements can be done only from one end, called the top of the stack(TOP). The order may be LIFO(Last In First Out) or FILO(First In Last Out).
Students need to first try and understand the implementation of using arrays. Once comfortable with the concept, they can further implement stacks using linked list as well.

Related Theory: -

Stack is a linear data structure which follows a particular order in which the operations are performed. It works on the mechanism of Last in First out (LIFO).

List 5 Real Life Examples:

1. **Web Browser Navigation**
   When you navigate through web pages, your browser keeps track of the pages you visit in a stack. Clicking the "Back" button takes you to the previous page by popping the current page off the stack, and the "Forward" button pushes the page back onto the stack if you navigate back.

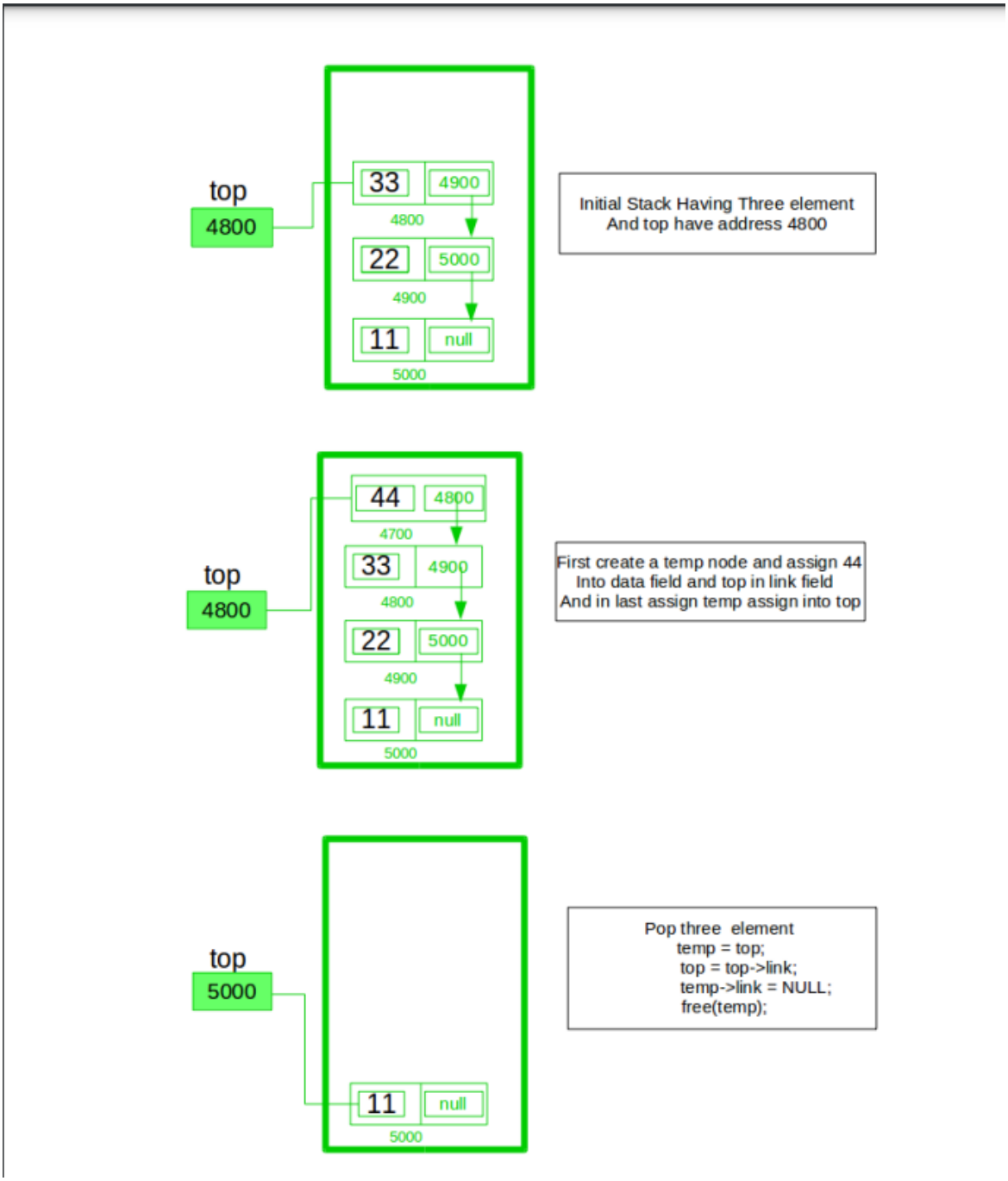
2. **Navigation History in Applications:**
   Many applications keep track of user navigation history in a stack. For example, when you navigate through directories in a file explorer, each directory is pushed onto a stack. Pressing the "Back" button navigates to the previous directory by popping from the stack.

3. **Undo Mechanism in Text Editors**
   When you perform actions in a text editor each action is pushed onto a stack. The "Undo" command pops the most recent action off the stack and reverses it. If you "Redo" an action, it is pushed back onto the stack.

4. **Call Stack in Programming:**
   When a function is called in a program, the current state (including variables, return address, etc.) is pushed onto the call stack. When the function returns, the state is popped off the stack to resume execution.

5. **Plates or Books in a Pile:**
   Imagine a stack of plates in a cafeteria. Plates are added (pushed) to the top of the stack and removed (popped) from the top. This is a direct physical analogy of the stack data structure where only the top element is accessible for removal or inspection.

Diagram:


top
4800
33
4900
4800
22
5000
4900
11
null
5000
Initial Stack Having Three element
And top have address 4800
top
4800
44
4800
4700
33
4900
4800
22
5000
4900
11
null
5000
First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top
top
5000
11
null
5000
Pop three element
temp = top;
top = top->link;
temp->link = NULL;
free(temp);

Explain Stack ADT:

The Stack ADT is a collection of elements that follows the Last In, First Out (LIFO) principle, where the last element added to the stack is the first one to be removed.

Basic Operations

The primary operations of a stack include:

- Push: Adds an element to the top of the stack.
- Pop: Removes and returns the top element of the stack.
- Peek: Returns the top element without removing it.
- isEmpty: Checks if the stack is empty.
-Size(): Returns the number of elements in the stack.

-Use Cases: Stacks are widely used in various applications, including:

- Function call management in programming (call stack).
- Undo mechanisms in text editors.
- Parsing expressions in compilers.

Algorithm for creation, insertion, deletion, displaying an element in stack:
Initialize Stack:

1. Start with top pointing to NULL.

2. Push Operation:
   Create a new node.
   Assign the data to the new node.
   Set the next pointer of the new node to the current top.
   Update top to point to the new node.
   If memory allocation fails, print "Stack Overflow".

3. Pop Operation:
   Check if top is NULL. If yes, print "Stack is Empty" and return top.
   Otherwise, save the current top node in a temporary variable temp.
   Print the data of the temp node as the popped element.
   Update top to point to the next node of temp.

Free the memory allocated to temp.
Return the updated top.

4. Peek Operation:
   Check if top is NULL. If yes, print "Stack is empty!" and return -1.
   Otherwise, return the data of the top node.

5. Display Operation:
   Check if top is NULL. If yes, print "Stack is empty!".
   Otherwise, traverse the linked list from top to the end, printing the data of each node.
   Check if Stack is Empty:
   Return 1 if top is NULL, otherwise return 0.

6. Main Menu:
   Display the menu with options:
   Read the user's choice and perform the corresponding stack operation.
   Repeat until the user chooses to exit

Program source code:

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int data;

   struct Node* next;

};

struct Node *top = NULL;

struct Node* push(struct Node* top, int data) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   if (newNode == NULL) {

      printf("Stack Overflow\n");

      return top;

   }

   newNode->data = data;
```

```
    newNode->next = top;
    top = newNode;
    return top;
}
struct Node* pop(struct Node** top) {
    struct Node *temp;
    if (*top == NULL) {
        printf("Stack is Empty\n");
    } else {
        temp = *top;
        printf("Popped element is %d\n", temp->data);
        *top = (*top)->next;
        free(temp);
    }
    return *top;
}
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return -1;
    }
    return top->data;
}

void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
```

```
        return;
    }
    struct Node *temp = top;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int isEmpty(struct Node* top) {
    return (top == NULL);
}
int main() {
    int choice, data;
    while (1) {
        printf("Make a Choice:\n1) Push\n2) Pop\n3) Peek\n4) Display\n5) Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the data to push: ");
                scanf("%d", &data);
                top = push(top, data);
                break;
            case 2:
                top = pop(&top);
                break;
```

```
            case 3:
                data = peek(top);
                if (data != -1) {
                    printf("Top element is: %d\n", data);
                }
                break;
            case 4:
                display(top);
                break;
            case 5:
                printf("Exiting.....");
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Output Screenshots:

```
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 1
Enter the data to push: 12
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 1
Enter the data to push: 23
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 1
Enter the data to push: 45
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 2
Popped element is 45
```

```
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 3
Top element is: 23
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 1
Enter the data to push: 78
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 4
78 23 12
Make a Choice:
1) Push
2) Pop
3) Peek
4) Display
5) Exit
Enter your choice: 5
Exiting.....

=== Code Execution Successful ===
```

Conclusion:-

From this experiment, I gained hands-on experience with implementing a dynamic stack using a linked data structure. I learned how to perform essential stack operations like push, pop, and peek. Applying the Last In, First Out (LIFO) principle in a practical context helped me better understand how data structures work and their real-world uses, such as managing function calls and implementing undo mechanisms. Overall, the experiment was a valuable learning experience, enhancing my grasp of these important concepts.

PostLab Questions:

1) Explain how Stacks can be used in Backtracking algorithms with example.

A stack can be used to keep track of the current state of the problem being solved. Each time a decision is made, the current state is pushed onto the stack.

If a decision leads to a dead end or invalid solution, the most recent state can be popped off the stack and the algorithm can backtrack to the previous state.

Backtracking algorithms can be recursive, with each level of recursion representing a different decision point in the problem-solving process. The stack can be used to keep track of the state at each level of recursion.

Backtracking can be used to solve a variety of problems, such as finding all possible solutions to a puzzle, finding the shortest path through a maze, or generating permutations of a set of elements.

Backtracking algorithms can be optimized by using heuristics to reduce the number of decision points that need to be explored.

For example, in the case of finding the shortest path through a maze, a heuristic might be to prioritize paths that are closer to the destination.

Backtracking algorithms can also be optimized by using techniques such as pruning or memorization to avoid exploring the same state multiple times.

Example:

In the N-Queens problem, the goal is to place N queens on an N×N chessboard such that no two queens threaten each other. A stack can be used to track the placement of queens while exploring potential solutions.

1. Start with an empty stack and begin with the first row.
2. Attempt to place a queen in the first column of the first row and push the position onto the stack.
3. Move to the next row and try placing a queen in a column where it is not under attack (no other queen in the same column, row, or diagonal). Push this position onto the stack.
4. If you reach a row where no valid position is available, backtrack by popping the last position from the stack. This removes the last placed queen, and you try the next possible column in the previous row.
5. Continue placing queens, pushing valid positions onto the stack, and popping when necessary until all queens are placed successfully or all possibilities are exhausted.
6. If the stack contains N positions (one for each queen), a solution is found. If the stack is empty after exploring all possibilities, no solution exists for the current configuration.

By the end, if you have successfully placed all queens on the board without conflicts, the stack contains the positions of the queens for a valid solution. If not, the algorithm will backtrack and try different configurations until all possibilities are explored.

2) Illustrate the concept of Call stack in Recursion.

Using example of factorial, let's illustrate the concept of a call stack in recursion using the example of calculating the factorial of a number nnn (denoted as n!n!n!).

The factorial of a number nnn is defined as:

- n!=n×(n−1)×(n−2)×…×1n! = n \times (n-1) \times (n-2) \times \ldots \times 1n!=n×(n−1)×(n−2)×…×1

- 0!=10! = 10!=1 (base case)

Code of the recursive function :

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Now for example , we want to compute 3! then :

Step1: The initial call is factorial(3). The function checks if n=0n = 0n=0. Since n=3n = 3n=3, it proceeds to the else branch. The function calls itself with factorial(2) and waits for the result before multiplying it by 3.

Call Stack:

- factorial(3)

Step 2: The second call is factorial(2). The function checks if n=0n = 0n=0. Since n=2n = 2n=2, it proceeds to the else branch. The function calls itself with factorial(1) and waits for the result before multiplying it by 2.

Call Stack:

- factorial(3)
- factorial(2)

Step 3: The third call is factorial(1). The function checks if n=0n = 0n=0. Since n=1n = 1n=1, it proceeds to the else branch. The function calls itself with factorial(0) and waits for the result before multiplying it by 1.

Call Stack:

- factorial(3)
- factorial(2)
- factorial(1)

Step 4: The base case is factorial(0). The function checks if n=0n = 0n=0. Since n=0n = 0n=0, it returns 1 as the base case.

Call Stack:

- factorial(3)

- factorial(2)
- factorial(1)
- factorial(0)

Step 5: The return value of factorial(0) (which is 1) is passed back to factorial(1). factorial(1) multiplies 1 by 1 (result from factorial(0)) and returns 1 to factorial(2). factorial(2) multiplies 2 by 1 (result from factorial(1)) and returns 2 to factorial(3). factorial(3) multiplies 3 by 2 (result from factorial(2)) and returns 6 as the final result.

Final Call Stack (after unwinding):

- Each call is removed from the stack as its execution completes and returns a result.

This process shows how the call stack works with recursion: As the function makes recursive calls, each call is added to the call stack and waits for the result of the next call. Once the base case is reached, the stack begins to unwind. Each call then completes and is removed from the stack one by one, starting with the most recent call and moving backward through the stack.