K. J. Somaiya College of Engineering, Mumbai-77
(Constituent college of Somaiya Vidya Vihar University)

| | |
|---|---|
| **Batch: E2** | **Roll No.: 16010123325** |
| **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:**

- **Browser Back and Forward Navigation**: Web browsers use stacks to manage the history of web pages you visit. When you navigate to a new page, the current page is pushed onto the back stack. When you press the back button, the page is popped from the stack and displayed. Similarly, a forward stack is used when you navigate forward.

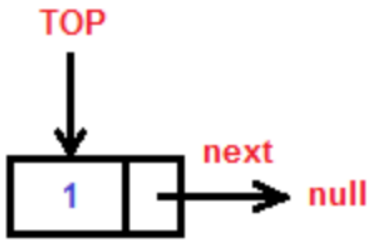
- **Undo Mechanism in Text Editors:** When you press 'Ctrl+Z' to undo your last action in a text editor, the program uses a stack to keep track of all the actions you perform. Each time you perform an action, it gets pushed onto the stack, and when you undo an action, it gets popped off the stack.

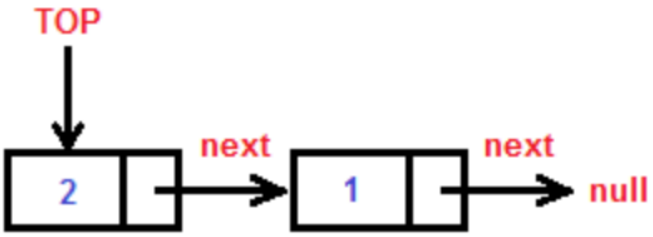
- **Expression Conversion (Infix to Postfix/Prefix):** Algorithms for converting expressions from infix notation (e.g., A + B) to postfix (e.g., AB+) or prefix notation (e.g., +AB) use stacks to manage operators and operands during the conversion process.

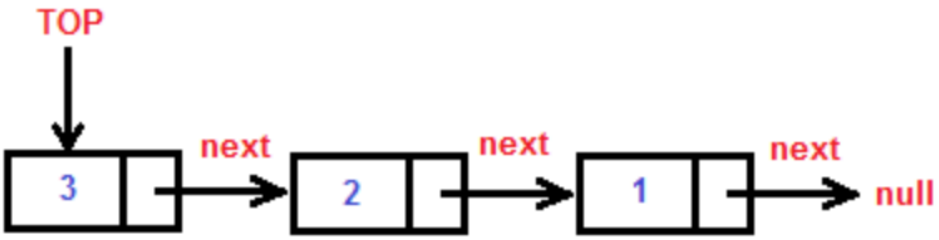
- **Recursive Algorithms:** Many algorithms, particularly those involving recursion, use stacks to keep track of recursive function calls.

- **Function Call Management in Programming Languages**: When a function calls another function, a stack is used to keep track of the return points and local variables of each function. This is known as the call stack.

**Diagram:**


TOP
next
1
null


TOP
next
next
2
1
null


TOP
next
next
next
3
2
1
null

**Explain Stack ADT:**

The Stack Abstract Data Type (ADT) is a collection of elements with two primary operations: push and pop. The stack follows the Last-In-First-Out (LIFO) principle, meaning the last element added (pushed) to the stack is the first one to be removed (popped) and does not allow random access to the elements of the stack.

**Key Operations of Stack ADT**

1. Push: Adds an element to the top of the stack
2. Pop: Removes and returns the top element of the stack
3. Peek: Returns the top element without removing it
4. isEmpty: Checks if the stack is empty, and displays a stack underflow error
5. display: Displays elements present in the stack at that moment

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

- Initialize Stack
    - Create a Stack structure with a pointer top initialized to NULL and an integer size initialized to 0.
- Assign Stack
    - Define a function createStack to allocate memory for a new stack, set its top to NULL, and its size to 0.
- Check if Empty
    - Define a function isEmpty to return if the stack's size is 0.
- Push Operation
    - Define a function insert to:

        i. Allocate memory for a new node

        ii. Set its data to the input value

        iii. Point its next to the current top

iv. Update the stack's top to this new node

v. Increment the stack's size

- Pop Operation
    - Define a function pop to:

      i. Check if the stack is empty

      ii. If not, store the data of the top node

      iii. Update the top to the next node

      iv. Free the memory of the old top node

      v. Decrement the stack's size vi. Return the popped data
- Peek Operation
    - Define a function peek to:

      i. Check if the stack is empty

      ii. Return the data of the top node
- Display Stack
    - Define a function display to:

      i. Check if the stack is empty

      ii. Traverse the stack from top to NULL, printing each node’s data
- Main Function
    - Continuously display a menu for stack operations
    - Read user input to perform push, pop, peek, display, check if empty, or exit based on choice.

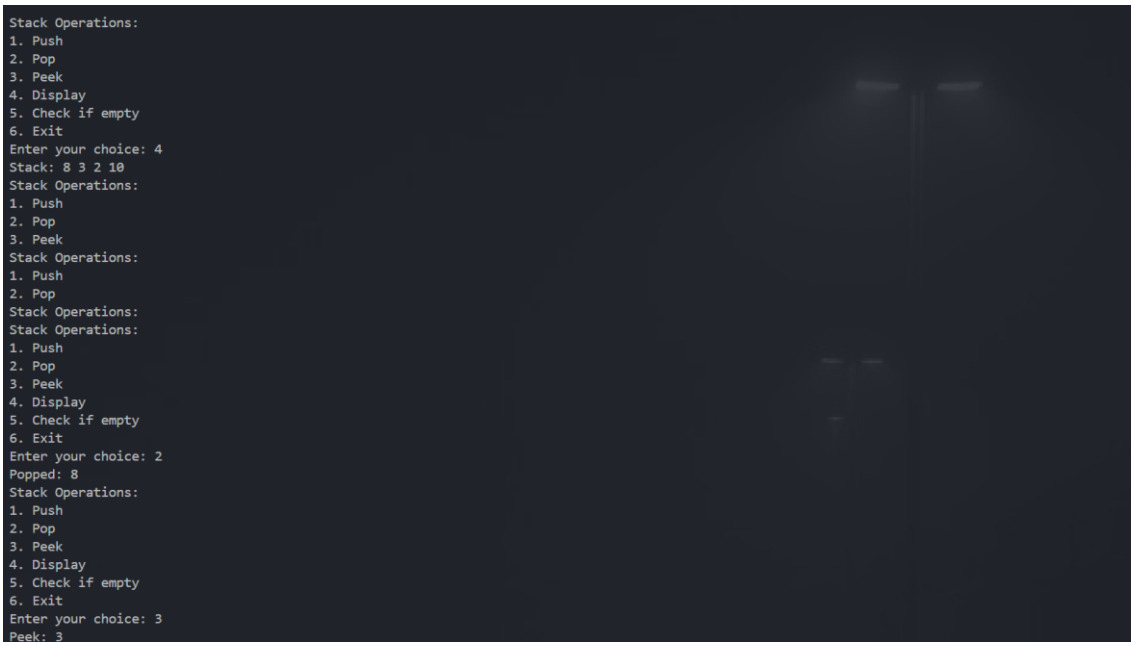
**Program source code:**

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

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

typedef struct Stack {
    StackEl* top;
    int size;
} Stack;

// Function to create a new stack element
Stack* createStack() {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->top = NULL;
    stack->size = 0;
    return stack;
}

// Function to insert an element at the top of the stack
void insert(Stack* stack, int data) {
    StackEl* element = (StackEl*)malloc(sizeof(StackEl));
    element->data = data;
    element->next = stack->top;
    stack->top = element;
    stack->size++;
}

// Function to remove an element from the top of the stack
int pop(Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    int data = stack->top->data;
    StackEl* temp = stack->top;
    stack->top = stack->top->next;
    free(temp);
    stack->size--;
```

```
    return data;
}

// Function to peek at the top element of the stack
int peek(Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack->top->data;
}

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

// Function to check if stack is empty
int isEmpty(Stack* stack) {
    return stack->size == 0;
}

int main() {
    Stack* stack = createStack();
    int choice, data;

    while (1) {
        printf("Stack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Check if empty\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to push: ");
                scanf("%d", &data);
```

```
                insert(stack, data);
                break;
            case 2:
                data = pop(stack);
                if (data != -1) {
                    printf("Popped: %d\n", data);
                }
                break;
            case 3:
                data = peek(stack);
                if (data != -1) {
                    printf("Peek: %d\n", data);
                }
                break;
            case 4:
                printf("Stack: ");
                display(stack);
                break;
            case 5:
                printf("Is stack empty? %s\n", isEmpty(stack) ? "Yes" :
"No");
                break;
            case 6:
                exit(0);
                break;
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

## Output Screenshots:

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Check if empty
6. Exit
Enter your choice: 4
Stack: 3 2 10
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Check if empty
6. Exit
Enter your choice: 5
Is stack empty? No
```

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Check if empty
6. Exit
Enter your choice: 6
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS\Programs>
```

## Conclusion:-

The above program highlights the dynamic implementation of stack data structure using linked list and dynamic memory allocation.

**PostLab Questions:**

1) A bracket is considered to be any one of the following characters: (, ), {, }, [, or ].

   Two brackets are considered to be a *matched pair* if the an opening bracket (i.e., (, [, or {) occurs to the left of a closing bracket (i.e., ), ], or }) *of the exact same type*. There are three types of matched pairs of brackets: [], {}, and ().

   A matching pair of brackets is *not balanced* if the set of brackets it encloses are not matched. For example, {[(])} is not balanced because the contents in between { and } are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, (, and the pair of parentheses encloses a single, unbalanced closing square bracket, ].

   By this logic, we say a sequence of brackets is *balanced* if the following conditions are met:

   - It contains no unmatched brackets.
   - The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

   Given strings of brackets, **Implement a program** to determine whether each sequence of brackets is balanced. If a string is balanced, return YES. Otherwise, return NO.

   SampleInput:

```
{[()]}
{[(])}
{{[[(())]]}}
```

   Sample Output:

```
YES
NO
YES
```

**Code:**

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

typedef struct StackEl {
    char bracket;
    struct StackEl* next;
} StackEl;

typedef struct Stack {
    StackEl* top;
} Stack;

// Function to create a new stack element
StackEl* createStackEl(char bracket) {
    StackEl* element = (StackEl*)malloc(sizeof(StackEl));
    element->bracket = bracket;
    element->next = NULL;
    return element;
}

// Function to push an element onto the stack
void push(Stack* stack, char bracket) {
    StackEl* element = createStackEl(bracket);
    element->next = stack->top;
    stack->top = element;
}

// Function to pop an element from the stack
char pop(Stack* stack) {
    if (stack->top == NULL) {
        return '\0';
    }
    char bracket = stack->top->bracket;
    StackEl* temp = stack->top;
    stack->top = stack->top->next;
    free(temp);
    return bracket;
}

// Function to check if the stack is empty
int isEmpty(Stack* stack) {
    return stack->top == NULL;
}
```

```c
// Function to check if a bracket is an opening bracket
int isOpeningBracket(char bracket) {
    return bracket == '(' || bracket == '[' || bracket == '{';
}

// Function to check if a bracket is a closing bracket
int isClosingBracket(char bracket) {
    return bracket == ')' || bracket == ']' || bracket == '}';
}

// Function to check if a closing bracket matches an opening bracket
int matches(char opening, char closing) {
    return (opening == '(' && closing == ')') ||
           (opening == '[' && closing == ']') ||
           (opening == '{' && closing == '}');
}

// Function to check if a sequence of brackets is balanced
int isBalanced(char* brackets) {
    Stack stack;
    stack.top = NULL;

    for (int i = 0; i < strlen(brackets); i++) {
        char bracket = brackets[i];
        if (isOpeningBracket(bracket)) {
            push(&stack, bracket);
        } else if (isClosingBracket(bracket)) {
            if (isEmpty(&stack)) {
                return 0; // Unmatched closing bracket
            }
            char opening = pop(&stack);
            if (!matches(opening, bracket)) {
                return 0; // Mismatched brackets
            }
        }
    }

    return isEmpty(&stack); // If the stack is empty, the brackets are
balanced
}

int main() {
    int t;
    scanf("%d", &t);
    while (t--) {
```

```
        char brackets[100];
        fgets(brackets, 100, stdin);
        brackets[strcspn(brackets, "\n")] = 0; // Remove newline
character

        if (isBalanced(brackets)) {
            printf("YES\n");
        } else {
            printf("NO\n");
        }
    }
    return 0;
}
```

```
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS> cd "c:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS\Programs\" ; if ($?) { gcc bal-brack.c
 -o bal-brack } ; if ($?) { .\bal-brack }
{[()]}
YES
{[(])}
NO
{{[[(())]]}}
YES
```

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

**Backtracking** is a problem-solving technique that involves exploring all possible options to solve a problem and then backtracking (reversing steps) when an option doesn't lead to a solution. Stacks are perfect for backtracking because they follow a Last In, First Out (LIFO) order, making it easy to undo the last step taken and try a different path.

**Example**- Maze solving is a problem that can be solved using a backtracking algorithm with a stack. In this problem, we need to find a path from the start to the end of a maze. The algorithm starts at the start point and recursively explores adjacent cells. If a dead-end is reached, the algorithm backtracks and tries a different path.

**Explanation:**

1. Use a stack to keep track of your path: Start at the entrance, push the starting position onto the stack.
2. Move to the next position: If it's valid (not a wall and not visited), push it onto the stack.

3. Dead-end or blocked path: If you can't move forward, pop the last position from the stack and backtrack to try a different direction.
4. Continue this process until you either find the exit or the stack is empty (no path found).

3) Illustrate the concept of Call stack in Recursion.

In recursion, the call stack stores each recursive function call. Each call adds a stack frame with the function's current state (parameters and local variables). When a base case is reached, the stack frames are popped one by one as the function returns, unwinding back to the original call. This process allows the function to remember its position in the sequence of calls, ensuring each step completes correctly.

**Example:-**

```
int factorial(int n) {
    if (n == 0) return 1; // Base case
    return n * factorial(n - 1); // Recursive call
}
```

**Explanation :-**

- factorial(3) is called and added to the stack
- factorial(3) calls factorial(2), which is added to the stack
- factorial(2) calls factorial(1), which is added to the stack
- factorial(1) calls factorial(0), which is added to the stack
- factorial(0) returns 1 (base case), and the stack begins to unwind:
  - factorial(1) returns 1 * 1 = 1
  - factorial(2) returns 2 * 1 = 2
  - factorial(3) returns 3 * 2 = 6

The stack is emptied as each function returns, producing the final result. The call stack helps manage these nested function calls and returns in the correct order