

A REPORT
ON
DATA STRUCTURE – STACK
PROJECT
BY

R. Vidya – AP24110011915

K. Jeevana Sarayu – AP24110011926

V. Jeeshitha Sai – AP24110011931

R. Alpana – AP24110011934

G. Chaitanya Siri – AP24110011935

Rolli Gupta – AP24110012176

Prepared in the partial fulfilment of the
Project Based Learning of Course CSE 201 - CODING SKILLS 1



SRM UNIVERSITY, AP.

NOVEMBER 2025

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to all those who supported and guided us throughout the successful completion of our project. First and foremost, we are deeply grateful to our Vice Chancellor, Prof. V. S. Rao, for providing an encouraging academic environment and the necessary resources that enabled us to carry out this work effectively.

We extend our heartfelt thanks to our Faculty Mentors, for their continuous guidance, valuable suggestions, and support during every stage of this project. His expertise and motivation have been crucial in shaping our work.

We also acknowledge the cooperation, teamwork, and collective effort of all team members— whose dedication and contribution were essential to the successful completion of this project.

Finally, we express our appreciation to all others who, directly or indirectly, supported us throughout this journey

ABSTRACT

The rapid development of digital technology has driven the need for efficient, modular, and maintainable applications. In this context, the stack data structure plays a crucial role in supporting the development of responsive and structured applications. The stack is not only theoretically relevant as a Last-In First-Out (LIFO) structure but also proven effective in implementing various application features, such as user navigation, undo- redo systems, mathematical expression processing, and graph traversal.

Stacks can be implemented using arrays or linked lists, depending on the requirements for flexibility and memory efficiency. Considering its ease of implementation and wide range of real-world applications, the stack data structure becomes an important component for developers to master. Proficiency in stack concepts and practices directly contributes to improving the quality of applications build.

INTRODUCTION

The rapid development of digital technology in recent years has brought about significant changes in how humans interact with information and computing systems. This transformation has not only impacted daily life but also revolutionized the approach to software development. Continuous digital innovations demand software that is not only efficient but also scalable, maintainable, and adaptable to dynamic user needs. In this context, the selection and implementation of appropriate data structures become a crucial component that cannot be overlooked.

A data structure defines a method of storing, organizing, and arranging data within a computer storage medium so that the data or information can be used efficiently. In programming techniques, a data structure refers to the layout of data containing data columns, whether visible to the user or not. It is stated that building the right data structure will enhance algorithmic capabilities, while developing the right algorithm will reduce the complexity of the algorithm itself. Without a good understanding of data structures, developers will find it difficult to build optimal systems, especially when faced with large-scale challenges or high system complexity. One of the fundamental yet highly influential data structures is the stack.

Objectives of the Project

- To give a clear understanding of stacks.
- To learn different stack operations like push, pop, and peek.
- To understand stack implementation using arrays and linked lists.
- To learn the real-world applications of stacks.
- To improve coding skills by writing and testing programs.
- To develop teamwork and problem-solving abilities.

Stack Data Structure

Definition - A stack is a linear structure where insertion and deletion take place only at one end called the **top**. It follows LIFO (Last In, First Out).

Example: A pile of plates—last plate placed is the first removed.

Characteristics:

- Elements are added and removed only from the top.
- Follows the LIFO rule.
- Only one pointer (top) is used.
- Operations are simple and fast.
- Memory usage depends on implementation (array or linked list).

Basic Applications of Stack:

Stacks are used in many real-life and computer tasks, such as

- Function calls (call stack in programming languages).
- Undo/Redo options in text editors.
- Backtracking, such as maze solving.
- Expression evaluation, such as converting infix to postfix.
- Memory management.
- Browser history, where pages are visited and returned backward.

Stack Operations:

Push

- Adds a new element to the top of the stack.
- If the stack is full (array), it gives **Stack Overflow**.

Pop

- Removes the topmost element.
- If stack is empty, it gives **Stack Underflow**.

Peek/Top

- Shows the value of the top element without removing it.
- Useful for checking the current item in stack.

isEmpty / isFull

- **isEmpty** checks if the stack has no elements.
- **isFull** checks if the stack is completely filled (for array-based stack)

Implementation of Stack:

Stack Using Array

- A fixed-size array is used to store elements.
- A variable `top` is used to track the index of the top element.
- Push increases `top` and inserts the value.
- Pop deletes the top element and decreases `top`.
- Overflow occurs when `top == size-1`.
- Underflow occurs when `top == -1`.

Stack Using Linked List

- A linked list allows a stack of flexible size.
- Each node contains:
 - Data
 - Pointer to next node
- The **top** pointer points to the most recent node.
- Push adds a node at the beginning.
- Pop removes the first node.

This implementation avoids overflow unless memory is full.

CODE IMPLEMENTATION

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>    // for isdigit()

#define MAX 100      // maximum size of stack

/* ----- STACK IMPLEMENTATION ----- */

typedef struct {
    int arr[MAX];
    int top;
} Stack;

// Initialize stack
void initStack(Stack *s) {
    s->top = -1;
}

// Check if stack is empty
int isEmpty(Stack *s) {
    return (s->top == -1);
}
```

```

// Check if stack is full
int isFull(Stack *s) {
    return (s->top == MAX - 1);
}

// Push an element onto stack
void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    s->arr[++(s->top)] = value;
    printf("%d pushed onto stack.\n", value);
}

// Pop an element from stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow! Nothing to pop.\n");
        return -1; // sentinel value
    }
    return s->arr[(s->top)--];
}

```

```

// Peek top element of stack
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. No top element.\n");
        return -1;
    }
    return s->arr[s->top];
}

// Display stack elements
void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements (top to bottom): ");
    for (int i = s->top; i >= 0; i--) {
        printf("%d ", s->arr[i]);
    }
    printf("\n");
}

```

```

/* ----- APPLICATION: POSTFIX EVALUATION ----- */

int applyOperator(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
            if (b == 0) {
                printf("Error: Division by zero!\n");
                exit(1);
            }
            return a / b;
        default:
            printf("Error: Unknown operator %c\n", op);
            exit(1);
    }
}

// Evaluate postfix expression using stack
int evaluatePostfix(const char *expr) {
    Stack s;
    initStack(&s);

```

```

int i = 0;
char ch;

while ((ch = expr[i]) != '\0' && ch != '\n') {
    if (ch == ' ') {
        // ignore spaces
        i++;
        continue;
    }

    if (isdigit((unsigned char)ch)) {
        // if operand is a single-digit number
        int value = ch - '0';
        push(&s, value);
    } else {
        // operator: pop two operands
        int b = pop(&s);
        int a = pop(&s);
        if (a == -1 || b == -1) {
            printf("Error: Invalid postfix expression.\n");
            return -1;
        }
        int result = applyOperator(a, b, ch);
        push(&s, result);
    }
}

```

```

    }
    i++;
}

int finalResult = pop(&s);
if (!isEmpty(&s)) {
    printf("Warning: Extra operands in expression.\n");
}
return finalResult;
}

/* ----- MAIN MENU ----- */

int main() {
    Stack s;
    initStack(&s);

    int choice, value;
    char expr[200];

```

```

while (1) {

    printf("\n==== STACK MENU =====\n");
    printf("1. Push\n");
    printf("2. Pop\n");
    printf("3. Peek (Top Element)\n");
    printf("4. Display Stack\n");
    printf("5. Evaluate Postfix Expression (Application)\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(&s, value);
            break;

        case 2: {
            int popped = pop(&s);
            if (popped != -1)
                printf("Popped element: %d\n", popped);
            break;
        }
    }
}

```

```

case 3: {
    int topVal = peek(&s);
    if (topVal != -1)
        printf("Top element: %d\n", topVal);
    break;
}

case 4:
    display(&s);
    break;

case 5:
    // flush leftover newline from previous scanf
    getchar();
    printf("Enter postfix expression (single-digit operands, no
        spaces or with spaces):\n");
    fgets(expr, sizeof(expr), stdin);
    {
        int result = evaluatePostfix(expr);
        if (result != -1)
            printf("Result of postfix expression: %d\n", result);
    }
    break;

```

```

case 6:
    printf("Exiting...\n");
    exit(0);

default:
    printf("Invalid choice! Please try again.\n");
}

return 0;
}

```

OUTPUT

```
===== STACK MENU =====
```

1. Push
2. Pop
3. Peek (Top Element)
4. Display Stack
5. Evaluate Postfix Expression (Application)
6. Exit

Enter your choice: 5

Enter postfix expression (single-digit operands, no spaces or with spaces):

3 1 5 * +

3 pushed onto stack.

1 pushed onto stack.

5 pushed onto stack.

5 pushed onto stack.

8 pushed onto stack.

Result of postfix expression: 8

```
===== STACK MENU =====
```

1. Push
2. Pop
3. Peek (Top Element)
4. Display Stack
5. Evaluate Postfix Expression (Application)
6. Exit

```
Enter your choice: 1
Enter value to push: 2
2 pushed onto stack.

===== STACK MENU =====
1. Push
2. Pop
3. Peek (Top Element)
4. Display Stack
5. Evaluate Postfix Expression (Application)
6. Exit
Enter your choice: 4
Stack elements (top to bottom): 2

===== STACK MENU =====
1. Push
2. Pop
3. Peek (Top Element)
4. Display Stack
5. Evaluate Postfix Expression (Application)
6. Exit
Enter your choice: 2
Popped element: 2
```

```
===== STACK MENU =====
1. Push
2. Pop
3. Peek (Top Element)
4. Display Stack
5. Evaluate Postfix Expression (Application)
6. Exit
Enter your choice: 6
Exiting...
```

APPLICATIONS

A stack is a linear data structure that operates on the principle of Last In, First Out (LIFO).

This means the last element added to the stack is the first one to be removed.

While this principle seems simple, it's very powerful and has many practical applications in programming. A stack provides two main operations: push, which adds an element to the top of the stack, and pop, which removes an element from the top of the stack. Despite having only these two core operations, stacks are a crucial component in various algorithms and computer system mechanisms.

In the real world, the stack data structure is a crucial part of many systems and applications, even though its use is often hidden from the end-user. For instance, in the function call systems of modern programming languages, whenever a function is called, essential information like function parameters, local variables, and return addresses are saved onto a stack. This mechanism allows functions to be called in a nested or recursive manner while maintaining data consistency and program flow. In this context, the stack functions as a call stack, which is vital for managing program execution flow.

Beyond the technical backend, stacks are also applied in various user interface features that are very familiar to everyday users. The most common example is the undo and redo features in word processors or graphic editors. Stacks are also used in web page navigation, where browsers store the history of visited pages in two stacks: one for the "back" stack and another for the "forward" stack. When a user presses the "back" button, the browser retrieves the last page from the back stack and adds it to the forward stack.

Stacks can even be used in music queuing. In mobile app development, modern frameworks like React Native also rely on the stack concept to manage screen navigation. Whenever a user moves from one screen to another, the previous screen is saved onto a stack. This allows users to easily return to previous screens. This concept is known as a navigation stack. Despite its importance, many software developers still don't fully optimize their use of the stack data structure. This can stem from various factors, such as a limited understanding of data structure theory, insufficient practical implementation experience in real-world projects, or the dominance of frameworks that explicitly hide data structure usage. As a result, many developers tend to rely heavily on automatic framework features without understanding the underlying logical processes.

RESULTS AND DISCUSSION

The stack data structure plays a vital role in various application development scenarios due to its flexible yet structured nature. The Last In First Out (LIFO) principle underlying the stack makes it an elegant solution for temporarily storing data and managing program execution flow hierarchically. This structure can be integrated using both arrays and linked lists, providing developers with the flexibility to choose an approach that suits performance and system complexity requirements.

Stacks can accelerate the process of temporary data transfer and storage, and reduce the number of variables used in the code. This proves that stacks not only function as a theoretical structure but also have a direct impact on algorithmic efficiency in the field. Stacks are used to navigate data based on input and revision times. With this approach, users can easily track the latest changes made in the system, similar to the undo feature in modern editors.

In large-scale software development, stacks are also used to evaluate expressions and develop compilers. Stacks can be used for infix to postfix expression conversion as well as executing these expressions with high accuracy. This is important in the context of language interpretation systems or systems that require repetitive mathematical expression processing.

Array-based implementations tend to be faster but are less flexible if the stack size varies, while linked lists are more flexible but require additional pointer allocation. In addition to these practical uses, stacks also underpin many graph traversal algorithms, such as Depth First Search (DFS). This approach is used in optimal pathfinding, recursive backtracking, and backtracking algorithms widely used in game engines, AI systems, and decision tree processing.

The stack data structure demonstrates high adaptability across various programming languages. Stacks also play an important role in algorithm optimization. Stacks accelerate the selection sort process by reducing temporary variables and increasing speed compared to recursive methods, although for large datasets, the risk of stack overflow must be anticipated with iterative implementation.

It can be concluded that the stack is not only a basic data structure but also a vital foundation for building responsive, modular, and scalable software systems. Its application spans education, information systems, expression processing, and dynamic data analysis, making the stack a must-have tool for every application developer.

CONCLUSION

The stack data structure holds a highly strategic position in modern software development. From a theoretical perspective, the stack offers strong fundamental principles in data management and execution control. From a practical perspective, its application has proven to be widespread across various contexts, ranging from data sorting, user navigation systems, expression processing, to search algorithms. Stacks can be implemented efficiently using both arrays and linked lists, depending on the application's space and flexibility requirements. Several studies indicate that the use of stacks can simplify application logic flow, reduce the number of temporary variables, and enhance code modularity.

Furthermore, the flexibility of stacks in supporting undo-redo features, data sorting, and graph traversal proves its usefulness in complex real-world situations. Considering its ease of implementation, time and memory efficiency, and relevance in various use cases, the stack data structure deserves to be included in the primary toolkit of contemporary software developers. Developers, whether working with Python, Java, or C++, are advised to master and explore various forms of stack application to optimize the performance and readability of their applications.

REFERENCES

- McMaster, S., C Memon, A. (2008). Call-stack coverage for GUI test suite reduction. IEEE Transactions on Software Engineering. IEEE.
- Alharbi, O., C Kurfess, F. (2014). Visualization of data structures and algorithms. International Conference on Information Technology.
- W3Schools.