

Department of Computer Science and Engineering (CSE)

Laboratory Report Cover Page Semester: Fall-2023

**Students must complete all details except the faculty use part*.***

**Experiment Number:**  **01-10**

**Course Code:**  **0611-CSE-1208**

**Course Name:** Data Structure Lab

**Submission Date: 23-12-2023**

**Md. Mizanur Rahman**

**Assistant Professor, CSE, RPSU**

**Course Teacher/Instructor**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Batch No.** | **Student Name** | **Student ID** | **Student**  **Signature** | **Date** |
| **Submitted by:** | | | | |
| 26 | Md. Siam Hossain | 23100084 23-12-2023 | | |

|  |  |
| --- | --- |
| **For faculty use only:** | **Total Marks:** **Marks Obtained:** |
| **Faculty comments** | |

|  |  |  |
| --- | --- | --- |
| Lab No: | Topics | Page No: |
| 01 | Write a program that will traverse each element of the array to find the sum and average of the elements. | 03-05 |
| 02 | Write a program in C to insert an element in an array at a particular position. | 06-08 |
| 03 | Write a program in C to delete an element at a particular position from an array. | 09-11 |
| 04 | Write a program that will find the location of a giver item in an array using Linear Search. | 12-14 |
| 05 | Write a program that will find the location of a given item in a sorted array using Binary Search algorithm. | 15-17 |
| 06 | Write a program that will sort the array elements in ascending order using Bubble Sort Algorithm. | 18-20 |
| 07 | Write a C program that will find the factorial of a number using Recursion. | 21-23 |
| 08 | Write a C program that print Fibonacci series of a given range using Recursion. | 24-26 |
| 09 | Write a C program to perform Push operation on stack using an array. | 27-29 |
| 10 | Write a C program to perform Pop operation on stack using an array. | 30-32 |

**INDEX**

**Problem No. 01**

**Name Of The Problem:** Write a program that will traverse each element of the array to find the sum and average of the elements.

**Theory:**

Traversing is a fundamental operation in data structures. Traversal involves the systematic access of each record precisely once, allowing for the examination or manipulation of specific elements within the records. This process of accessing or interacting with the records is commonly referred to as "visiting" them.

**Algorithm :**

Algorithm to Calculate Sum and Average of N Numbers in an Array

1. Start.

2. Input the value of N.

3. Create an array a of size 50 but assigned only n numbers of elements.

4. Initialize sum to 0 and i to 0.

5. Repeat Steps 6 and 7 while i < N

6. Add a[i] to sum.

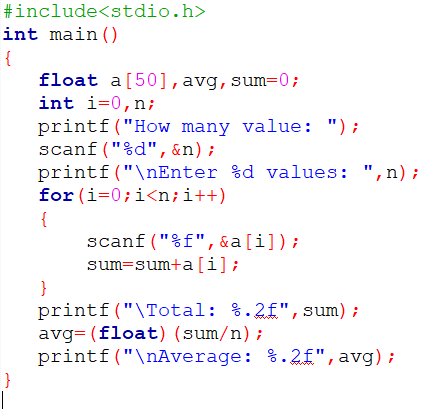
7. Increment i by 1.

8. Calculate the average as Average = sum / N.

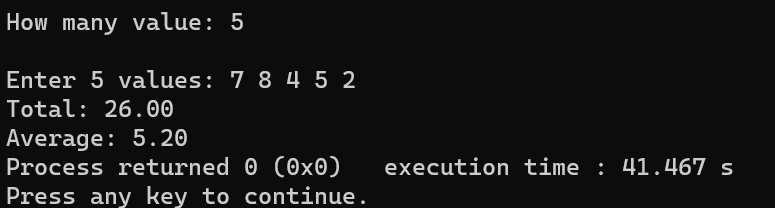
9. Output the sum and average.

10. End.

**Program:**



**Input & Output:**



**Discussion:**

Here is the modified passage for the conclusion and discussion of the C program:

"In this C program, we have implemented a basic algorithm to traverse an array, calculating both the sum and average of its elements. The program starts by allowing the user to specify the size of the array. It then utilizes a `while` loop to iterate through the array, accumulating the elements' values to calculate the sum.

This simple program showcases fundamental concepts of array manipulation and user input in C. While the program is intentionally written in an elementary C style for clarity, it can serve as a starting point for more complex applications. By understanding and mastering these basic principles, one can build more sophisticated programs to process and analyze data efficiently.

**Conclusion:**

this program demonstrates the core principles of array handling, user input, and basic arithmetic operations in C, making it a successful example of a basic C program."

**Problem No. 02**

**Name Of the Problem :** Write a program in C to insert an element in an array at a particular position.

**Theory:**

Inserting elements in data structures follows a general process: determine the position, shift if necessary, and insert. For arrays, directly assign to an index. In linked lists, create a new node and adjust pointers. Binary trees require traversal and pointer adjustments. Queues use enqueue at the rear/front based on their behavior.

**Algorithm:**

The algorithm for inserting an element into an array at a particular position can be summarized as follows:

1. Declare an array a of size 60 and variables n, k, item, and i.

2. Read the number of elements n from the user.

3. Read the values of the array elements from the user and store them in the array a

4. Read the position k at which you want to insert the new element.

5. Read the value item that you want to insert.

6. Start a loop from the end of the array (n-1) to position k-1. This loop is used to shift the elements to the right to make space for the new element.

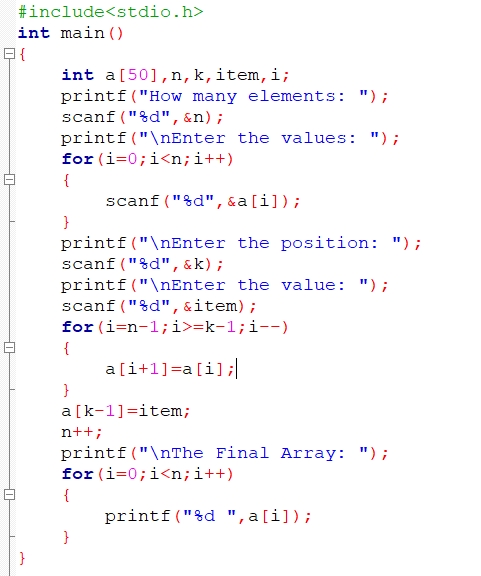
• a[i+1] = a[i] shifts the element at position i to the right by one position.

7. After the loop, assign the item to the position k-1 to insert it at the specified location.

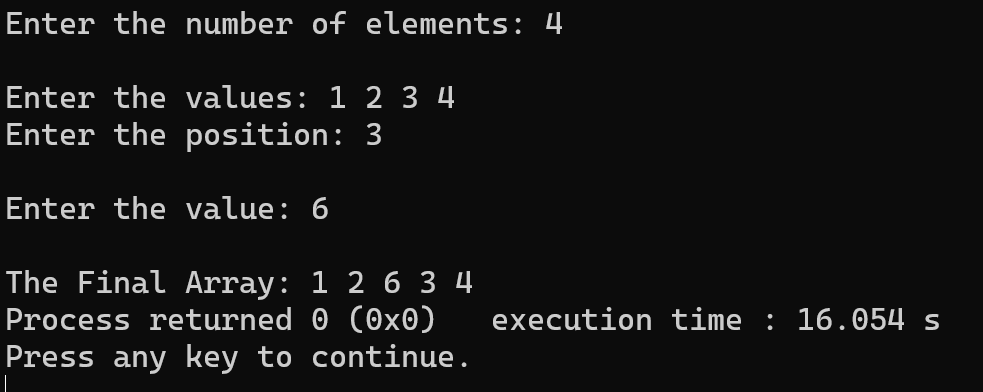
8. Increment n to reflect the new size of the array.

9. Print the final array after the insertion.

**Program:**



**Input & Output:**



**Discussion:**

In this lab, we learned how to insert an element into an array at a specific position in C. The program first takes the number of elements and the elements themselves as input. It then asks for the position at which to insert the new element and validates whether the position is within a valid range. If the position is valid, it shifts the elements to the right to make space for the new element and inserts it. Finally, it prints the updated array. This program is useful when you need to maintain the order of elements in an array while inserting new values at specific positions. It can be applied in various programming scenarios where array manipulation is required.

**Problem-03:**

**Name of the problem:** Write a program in C to delete an element at a particular position from an array.

**Theory:**

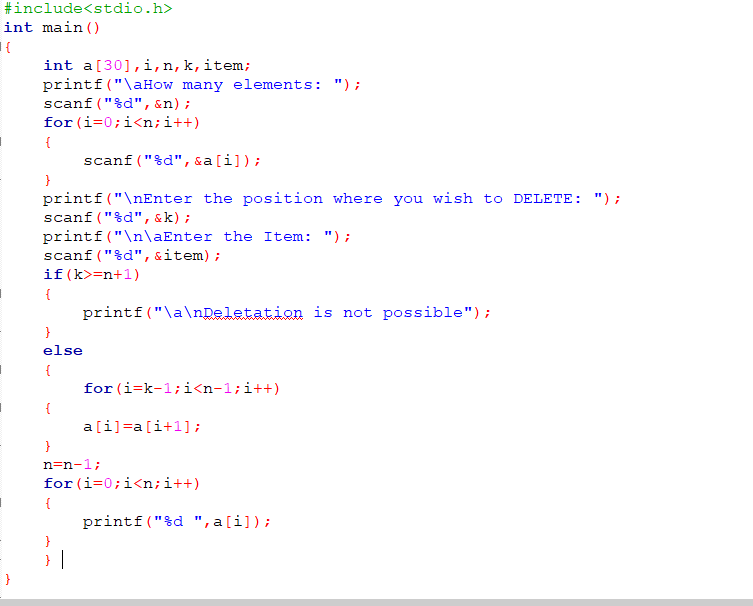
Deleting elements in data structures follows a general process: determine the position, shift if necessary, and delete. For arrays, directly delete from an index. In linked lists, create a new node and adjust pointers. Binary trees require traversal and pointer adjustments. Queues use enqueue at the rear/front based on their behavior.

**Algorithm:**

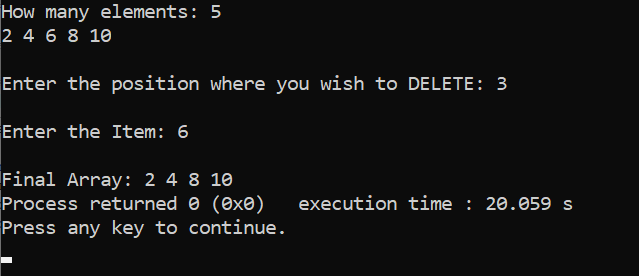
The algorithm for inserting an element into an array at a particular position can be summarized as follows:

1. It takes user input for the number of elements in the array elements.
2. It takes the position(K) where the deletion is to be performed and the item (if any) to insert at that position.
3. It checks if the position to delete(K) is valid. if it’s out of bounds, it prints an error message.
4. If the position is valid, it shifts the elements to the left starting from the specified position, effectively deleting the element at that position.
5. It reduces the size of the array by 1.
6. Finally, it pains the updated array without the deleted element.

**Program:**



**Input and Output:**



**Discussion:**

In this lab, we learned how to delete an element from an array at a specific position in C. The program first takes the position of elements and the elements themselves as input. If the position is valid, it move the elements from the right side to the specific valid position. Finally, it prints the updated array. This program is useful when you need to maintain the order of elements in an array while deleting any values from a specific positions. It can be applied in various programming scenarios where array manipulation is required.

**Problem No. 04**

**Name Of The Problem:** Write a program that will find the location of a giver item in an array using Linear Search.

**Theory:**

Linear Search is defined as a sequential search algorithm that starts at

one end and goes through each element of a list until the desired element is

found, otherwise the search continues till the end of the set.

**Algorithm:**

1. Read the number of elements **n**.
2. Create an array **a** of size **n**.
3. Read **n** elements into the array.
4. Read the item to be searched (**item**).
5. Iterate through the array to find the location (**loc**) of the item:
   * If the current element is equal to the item, set **loc** to the current index + 1 and break out of the loop.
6. Check the value of **loc**:
   * If **loc** is 0, print "Item is not found."
   * If **loc** is greater than 0, print the item and its location.

**Program:**

#include <stdio.h>

int main()

{

int n, item, i, loc = 0;

printf("How many elements: ");

scanf("%d", &n);

int a[n];

for (i = 0; i < n; i++)

{

scanf("%d", &a[i]);

}

printf("Item that you want to search: ");

scanf("%d", &item);

for (i = 0; i < n; i++)

{

if (a[i] == item)

{

loc = i + 1;

break;

}

}

if (loc == 0)

{

printf("Item is not found!\n");

}

else if (loc > 0)

{

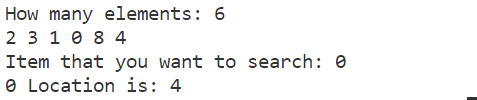
printf("%d Location is: %d\n", item, loc);

}

return 0;

}

**Input and Output:**

****

**Discussion**:

In this C program, we've successfully implemented a linear search

algorithm for finding the location of a given item in an array. The user is

prompted to input the item to search for, and the program then iterates through

the array elements sequentially, checking for a match. If a match is found, it

prints the location of the item; otherwise, it informs the user that the item is not

in the array. This concise program effectively illustrates the linear search

algorithm, offering a clear and efficient solution for the task.

**Problem No. 05**

**Name of the problem:** Write a program that will find the location of a given item in a sorted array using Binary Search algorithm.

**Theory**:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half and the correct interval to find is decided based on the searched value and the mid value of the interval.

**Algorithm**:

Step-1: Initialize beg to 0 (LB), end to n - 1 (UB), and mid to int((beg + end)/2). Step-2: Repeat Steps 3 and 4 while beg is less than or equal to end and data[mid] is not equal to item.

Step-3: If item is less than data[mid], then set end to mid -1. Otherwise, set beg to mid + 1.

Step-4: Set mid to int((beg + end)/2).

Step-5: If data[mid] is equal to item, then set Loc to mid. Otherwise, set loc to null.

Step-6: Exit

**Program**:

#include <stdio.h>

int main()

{

    int data[] = {7, 9, 13, 17, 19, 20, 23, 27};

    int loc = 0, item, n, mid;

    printf("Enter the number of items: ");

    scanf("%d", &n);

    printf("Enter the item you want to search for: ");

    scanf("%d", &item);

    int beg = 0;

    int end = n - 1;

    while (beg <= end)

    {

        mid = (beg + end) / 2;

        if (data[mid] < item)

            beg = mid + 1;

        else if (data[mid] > item)

            end = mid - 1;

        else

        {

            printf("%d found at position %d", item, mid + 1);

            break;

        }

    }

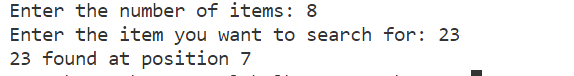
    if (beg > end)

        printf("%d does not exist in the array", item);

    return 0;

}

**Input and Output:**

****

**Discussion**:

In this C program, we have successfully implemented a binary search algorithm to find the location of a given item in a sorted array. Binary search is a highly efficient searching technique, particularly suited for sorted data. This program exemplifies how binary search works by dividing the search interval in half, which significantly reduces the number of iterations required to find the desired item.

**Problem No. 06**

**Name of the problem:** Write a program that will sort the array elements in ascending order using Bubble Sort Algorithm.

**Theory**:

The provided C program implements the Bubble Sort algorithm to sort an array of integers in ascending order. Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

**Algorithm**:

1. Input:

- Prompt the user to enter the total number of elements (n).

- Create an array `a` of size `n` to store the elements.

- Prompt the user to enter the array values.

2. Bubble Sort:

- Initialize a variable `count` to 1.

- Use a while loop to repeat the sorting process until `count` is equal to the total number of elements (`n`).

- Inside the while loop, use a nested for loop to iterate through the array elements.

- Compare adjacent elements, and if they are in the wrong order, swap them.

- Increment the `count` after each pass through the array.

3. Output:

- After sorting is completed, print the sorted array.

**Program**:

#include <stdio.h>

int main()

{

    int n;

    printf("Enter the total elements: ");

    scanf("%d", &n);

    int a[n];

    printf("Enter the Array Values: ");

    for (int i = 0; i < n; i++)

    {

        scanf("%d", &a[i]);

    }

    int count = 1;

    while (count < n)

    {

        for (int i = 0; i < n - 1; i++)

        {

            if (a[i] > a[i + 1])

            {

                int temp = a[i + 1];

                a[i + 1] = a[i];

                a[i] = temp;

            }

        }

        count++;

    }

    printf("After Sorting: ");

    for (int i = 0; i < n; i++)

    {

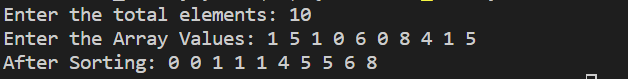
        printf("%d ", a[i]);

    }

    return 0;

}

**Input and Output:**

****

**Discussion:**

- The program uses an array to store the elements provided by the user.

- It employs the Bubble Sort algorithm, which has a time complexity of O(n^2) in the worst case.

- Bubble Sort is not the most efficient sorting algorithm, especially for large datasets, but it is simple to understand and implement.

- The algorithm works by repeatedly swapping adjacent elements until the entire array is sorted.

- The outer while loop ensures that the sorting process is repeated enough times to guarantee that the array is fully sorted.

**Conclusion**:

- The program successfully sorts the array elements in ascending order using the Bubble Sort algorithm.

- It demonstrates the basic concepts of array manipulation and sorting in C.

- While Bubble Sort is not the most efficient sorting algorithm for large datasets, it serves as a good educational example due to its simplicity.

- Other more efficient sorting algorithms like Quick Sort or Merge Sort are preferred for larger datasets in real-world applications.

**Problem No. 07**

**Name of the problem:** Write a C program that will find the factorial of a number using Recursion.

**Theory**:

The given C program calculates the factorial of a non-negative integer entered by the user. The factorial of a non-negative integer \(n\), denoted as \(n!\), is the product of all positive integers less than or equal to \(n\). Mathematically, \(n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1\).

**Algorithm**:

1. The program defines a function `factorial` that takes an integer \(a\) as a parameter and returns its factorial.

2. Inside the `factorial` function, there's a base case: if \(a\) is 0 or 1, the function returns 1, as \(0! = 1\) and \(1! = 1\).

3. If \(a\) is greater than 1, the function calculates the factorial recursively using the formula \(f = a \times \text{factorial}(a - 1)\).

4. The `main` function:

- Prompts the user to enter an integer between 0 and 31.

- Reads the input integer.

- Calls the `factorial` function to compute the factorial of the entered integer.

- Outputs the result.

**Program:**

#include <stdio.h>

int factorial(int a)

{

    int f;

    if (a == 0 || a == 1)

        return 1;

    f = a \* factorial(a - 1);

    return f;

}

int main()

{

    int n;

    printf("Enter any integer number between 0-31: ");

    scanf("%d", &n);

    int fact = factorial(n);

    printf("Factorial of %d is: %d\n", n, fact);

    return 0;

}

**Input and Output:**

****

**Discussion**:

1. Recursive Approach: The program uses a recursive approach to calculate the factorial. Each recursive call reduces the problem to a smaller subproblem until it reaches the base case.

2. Input Range: The program restricts the user to input integers between 0 and 31. This limitation is imposed because the factorial of larger numbers grows rapidly, and it may exceed the range representable by the data type `int`.

3. Factorial Calculation: The factorial is calculated using a recursive method, which may not be the most efficient for large values of \(n\) due to the overhead of multiple function calls.

**Conclusion:**

The program successfully computes and displays the factorial of the user-input integer using a recursive approach. While recursion provides a clear and concise solution, it may not be the most efficient method for large factorials. Additionally, the input range limitation helps prevent overflow issues for larger factorials within the constraints of the `int` data type.

**Problem No. 08**

**Name of the problem:** Write a C program that print Fibonacci series of a given range using Recursion.

**Theory**:

The program calculates and displays the Fibonacci series up to a specified range. The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. In this program, a recursive function (`fibo`) is used to calculate the Fibonacci numbers.

**Algorithm**:

1. The `fibo` function takes an integer parameter `a` representing the position in the Fibonacci sequence.

2. If `a` is 0, the function returns 0. If `a` is 1, the function returns 1.

3. For `a` greater than 1, the function recursively calls itself with `a - 1` and `a - 2`, then adds the results to calculate the Fibonacci number at position `a`.

4. The `main` function:

- Takes user input for the range `n`.

- Prints a message indicating the range.

- Iterates from 0 to `n - 1` and calls the `fibo` function for each iteration, printing the Fibonacci numbers in the specified range.

**Program:**

#include <stdio.h>

int fibo(int a)

{

    if (a == 0)

        return 0;

    else if (a == 1)

        return 1;

    else

    {

        int sum = fibo(a - 1) + fibo(a - 2);

        return sum;

    }

}

int main()

{

    int n;

    printf("Enter the Range: ");

    scanf("%d", &n);

    printf("Fibonacci Series with the Range of %d is: ", n);

    for (int i = 0; i < n; i++)

    {

        printf("%d ", fibo(i));

    }

    return 0;

}

**Input and Output:**



**Discussion**:

1. The program uses recursion to calculate Fibonacci numbers. While recursion is a straightforward way to implement the Fibonacci sequence, it can be inefficient for large values of `n` due to repeated calculations.

2. The time complexity of the program is exponential, O(2^n), because each Fibonacci number is calculated by recursively calling the function twice.

3. The program lacks error handling for negative input values or non-integer inputs.

**Conclusion:**

The program successfully generates and displays the Fibonacci series within the specified range using a recursive approach. However, it has limitations in terms of efficiency, especially for larger values of `n`. To improve performance, alternative methods such as dynamic programming or iterative approaches could be considered. Additionally, incorporating input validation for non-integer and negative values would enhance the program's robustness.

**Problem No. 09**

**Name of the problem:** Write a C program to perform Push operation on stack using an array.

**Theory**:

This C program implements a simple stack using an array. A stack is a data structure that follows the Last In, First Out (LIFO) principle. The stack operations supported in this program are push, which adds an element to the top of the stack, and exit, which terminates the program.

**Algorithm**:

1. Initialize a global variable `top` to -1 and an array `inp\_array` to store stack elements.

2. Implement the `push` function to add elements to the stack.

- Check if the stack is full (`top == SIZE - 1`), and if so, print "Stack is Overflow."

- If not full, prompt the user to enter an element (`x`), increment `top`, and store the element in the array.

- Display the current state of the stack after the push operation.

3. In the `main` function, use a `while` loop to repeatedly display a menu for stack operations.

- The user can choose to push an element onto the stack or exit the program.

- Use a `switch` statement to perform the chosen operation.

- If the user chooses to push, call the `push` function.

- If the user chooses to exit, terminate the program using `exit(0)`.

- Handle any invalid choices with a default case.

**Program**:

#include <stdio.h>

#include <stdlib.h>

#define SIZE 3

int top = -1;

int inp\_array[SIZE];

void push()

{

    int x, i;

    if (top == SIZE - 1)

    {

        printf("Stack is Overflow\n");

    }

    else

    {

        printf("Enter your Elements: ");

        scanf("%d", &x);

        top += 1;

        inp\_array[top] = x;

        printf("Present situatation of the stack: ");

        for (i = 0; i <= top; i++)

        {

            printf("%d ", inp\_array[i]);

        }

        printf("\n");

    }

}

int main()

{

    int choice;

    while (1)

    {

        printf("Perform operation on the stack: \n");

        printf("1.Push\n2.Pop\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice)

        {

        case 1:

            push();

            break;

        case 2:

            exit(0);

        default:

            printf("Invalid Choice\n");

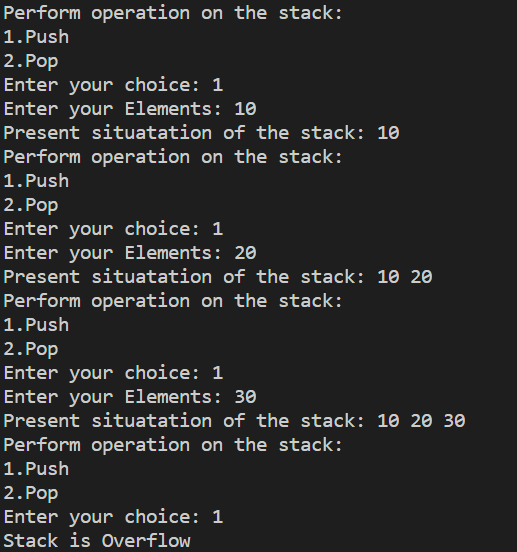
        }

    }

    return 0;

}

**Input and Output:**



**Discussion**:

- The program uses a simple menu-driven approach to interact with the user for stack operations.

- It checks for overflow condition before pushing an element onto the stack.

- The current state of the stack is displayed after each push operation.

**Conclusion**:

This program provides a basic implementation of a stack using an array in C. It allows the user to push elements onto the stack and exit the program. The push operation is protected against stack overflow. The program demonstrates a simple use case of a stack data structure. Further enhancements could be made, such as adding a pop operation, checking for stack underflow, and handling more stack operations.

**Problem No. 10**

**Name of the problem:** Write a C program to perform Pop operation on stack using an array.

**Theory**:

The program implements a basic stack data structure using an array in C. A stack is a Last In First Out (LIFO) data structure, where elements are added and removed from the same end called the "top." The program provides two primary operations: push (to add elements to the stack) and pop (to remove elements from the stack). The maximum size of the stack is defined by the constant `SIZE`.

**Algorithm**:

1. **pop Function:**
   * Input: STACK (array), TOP (top of the stack), ITEM (item to be popped)
   * Check if the stack is empty (TOP == -1). If it is, print "Underflow: STACK is empty" and return.
   * Decrease TOP by 1 to point to the next element in the stack.
   * Print the updated stack after the POP operation.
   * Print the popped item.
2. **main Function:**
   * Initialize variables TOP, ITEM, and i.
   * Create an array STACK with initial values {12, 23, 34, 56, 67}.
   * Set TOP to 4 (since the array has 5 elements, and the top index is 4).
   * Display the original stack using a loop and print statements.
   * Set ITEM to the last element in the stack (STACK[TOP]).
   * Call the pop function with the STACK, TOP, and ITEM as arguments.
   * Return 0 to indicate successful execution.

**Program**:

#include <stdio.h>

void pop(int STACK[], int Top, int ITEM);

int main()

{

    int TOP, ITEM, i;

    int STACK[] = {12, 23, 34, 56, 67};

    TOP = 5;

    printf("STACK before POP operation:");

    for (i = TOP-1; i >= 0; i--)

        printf("\n----\n|%d|", STACK[i]);

    ITEM = STACK[TOP-1];

    pop(STACK, TOP, ITEM);

    return 0;

}

void pop(int STACK[], int TOP, int ITEM)

{

    int i;

    if (TOP == -1)

    {

        printf("Underflow: STACK is empty.");

        return;

    }

    TOP = TOP - 1;

    printf("\n\nSTACK after POP operation:");

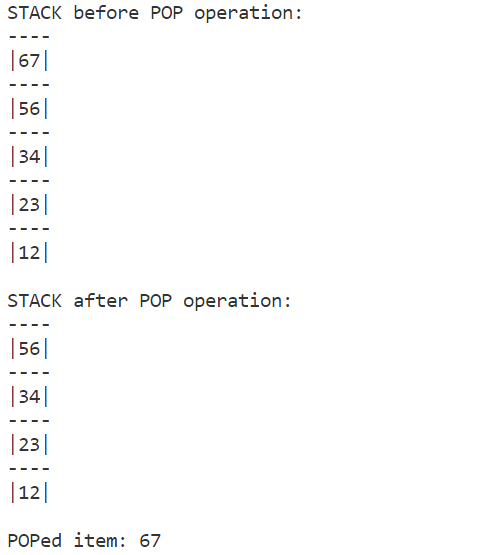
    for (i = TOP-1; i >= 0; i--)

        printf("\n----\n|%d|", STACK[i]);

    printf("\n\nPOPed item: %d\n", ITEM);

}

**Input and Output:**



**Discussion:**

* The code defines a simple stack operation - pop.
* The stack is implemented as an array named STACK with a variable TOP representing the top of the stack.
* The main function initializes the stack, displays its contents, and then performs a pop operation.
* The pop function checks for underflow (empty stack) before attempting to pop an element.
* After the pop operation, the updated stack and the popped item are displayed.

**Conclusion**:

* The code demonstrates the basic functionality of a stack and the pop operation.
* It handles underflow (empty stack) gracefully by checking before popping.
* The algorithm efficiently updates the stack and prints the popped item.
* However, the code assumes a fixed-size stack and does not include dynamic memory allocation for a more flexible implementation.

**Thank You!**