Lab 2 :

1. Write a program for implementing the following operations of stack S & also find the amortized cost if a sequence of n following operations are performed on a data structure.

(i) Push(S, x)

(ii) Pop(S)

(iii) Multipop(S,k)

stack_2.c
~/Documents/ADSANEW_LAB/lab2

```c
#include <stdio.h>
#include <stdlib.h>
int top;
int s[10];
int item, i, n;
int count = 0, value = 0, stacksize = 10;

void multipop(int n)
{
    if (top == -1)
    {
        printf("Stack is empty\n");
        return;
    }
    if (top + 1 < n)
        n = top + 1;
    for (i = 0; i < n; i++)
    {
        value++;
        count++;
        item = s[top--];
        printf("Popped element is %d\n", item);
        printf("Total cost:%d\n", value);
    }
}

void push(int item)
{
    if (top == (stacksize - 1))
    {
        printf("Stack is full\n");
        return;
    }
    top += 1;
    s[top] = item;
    count++;
    value++;
    printf("Total cost:%d\n", value);
}

void pop()
{
    if (top == -1)
    {
        printf("Stack is empty\n");
        return;
    }
    value++;
    count++;
    item = s[top--];
    printf("Popped element is %d\n", item);
    printf("Total cost:%d\n", value);
}

void display()
{
    if (top == -1)
    {
        printf("Stack is empty\n");
        return;
    }
    printf("\n STACK ELEMENTS : ");
    for (i = 0; i <= top; i++)
    {
        printf("%d ", s[i]);
    }
    printf("\n The Amortized cost for %d operation is %d\n", count, (value / count));
}
```

```
int main()
{
    int operation, n;
    top = -1;
    for (;;)
    {
        printf("\n1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit\n");
        printf("Enter the operation : ");
        scanf("%d", &operation);
        switch (operation)
        {
        case 1:
        {
            printf("Enter the element : ");
            scanf("%d", &item);
            push(item);
            break;
        }

        case 2:
            pop();
            break;

        case 3:
        {
            printf("Enter the number of items : ");
            scanf("%d", &n);
            multipop(n);
            break;
        }

        case 4:
            display();
            break;
        case 5:
            exit(0);
            break;
            break;
        }
    }
    return 0;
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB/lab2$ gcc -o stk stack_2.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB/lab2$ ./stk

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 1
Enter the element : 10
Total cost:1

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 1
Enter the element : 20
Total cost:2

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 1
Enter the element : 30
Total cost:3

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 1
Enter the element : 40
Total cost:4

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 1
Enter the element : 50
Total cost:5

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 4

 STACK ELEMENTS : 10 20 30 40 50
 The Amortized cost for 5 operation is 1
```

```
1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 2
Popped element is 50
Total cost:6

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 2
Popped element is 40
Total cost:7

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 4

 STACK ELEMENTS : 10 20 30
 The Amortized cost for 7 operation is 1
```

```
1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 3
Enter the number of items : 2
Popped element is 30
Total cost:8
Popped element is 20
Total cost:9

1:Push 2:Pop 3:Multi-Pop 4:Display 5:Exit
Enter the operation : 4

 STACK ELEMENTS : 10
 The Amortized cost for 9 operation is 1
```

Analysis

- Initially, added 5 elements (10, 20, 30, 40, 50) to the stack. This action increased the overall operation cost.
- Two elements (50 and 40) were then removed from the stack using pop operations. Each pop is added to the total operation cost.
- Later a multi-pop operation to remove 2 elements. Since only 3 elements remained, this cleared the stack, causing additional cost.

- Amortized cost per operation gives an average expense for each action. It's found by dividing the total cost by all operations performed.
- The actual amortized cost and stack content can differ based on how you carried out the steps.
- This analysis helps understand how costs change with various operations and their order, considering factors like stack size and sequence.

Time Complexity:
The time complexity of the stack operations in this code is generally constant (O(1)) for push and pop, as they involve single-element manipulation. Multi-pop and display operations take linear time (O(n)), where n is the number of elements being popped or displayed, respectively.

1. Push Operation:
   Time Complexity: O(1)
   Best Case: O(1) (when the stack has space for the new element)
   Average Case: O(1) (constant time operation)
   Worst Case: O(1) (when the stack is not full and there is available space)

2. Pop Operation:
   Time Complexity: O(1)
   Best Case: O(1) (when the stack is not empty)
   Average Case: O(1) (constant time operation)
   Worst Case: O(1) (when the stack is not empty)

3. Multi-Pop Operation:
   Time Complexity: O(n) (where n is the number of elements to be popped)
   Best Case: O(1) (when the number of elements to be popped is very small)
   Average Case: O(n) (linear time operation based on the number of elements to be popped)
   Worst Case: O(n) (when the number of elements to be popped is equal to the size of the stack)

2. Write a program to implement INCREMENT operation in a k-bit binary counter that counts upward from 0. What happens to the counter as it is incremented 16 times? Find the amortized cost of this operation if sequences of n increment operations are performed.

binary_counter.c
~/Documents/ADSANEW_LAB/lab2

```c
#include <stdio.h>
#include <stdlib.h>
#define NOP 16

int main()
{
    int j, len, i = 0, count = 0, co = 0, amor = 0;
    int A[4] = {0, 0, 0, 0};
    len = sizeof A / sizeof A[0];

    for (j = 0; j < len; j++)
    {
        printf("%d", A[j]);
    }
    printf("\n");

    while (i < len)
    {
        count = 0;

        if (A[i] == 1)
        {
            A[i] = 0;
            count++;
            i = i + 1;
        }

        if (i <= len && A[i] == 0)
        {
            A[i] = 1;
            count++;

            for (j = len - 1; j >= 0; j--)
                printf("%d", A[j]);
            printf("\n");

            if (i > 0)
                i = 0;
        }

        co = co + count;
    }

    amor = co / NOP;
    printf("Total amortized cost is: %d\n", co);
    printf("Average Amortized cost is: %d\n", amor);
}
```

```
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB/lab2$ gcc -o bc binary_counter.c
rohithsaidatta@rohithsaidatta-VirtualBox:~/Documents/ADSANEW_LAB/lab2$ ./bc
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Total amortized cost is: 61
Average Amortized cost is: 3
```

Analysis:
If there's a 1 in the list at position i, change it to 0 and move to the next position (i = i + 1).
If there's a 0 in the list at position i, change it to 1, show the reversed list, and if i is greater than 0, reset i to 0.
After that, the code calculates and shows the total and average cost based on the number of actions performed (NOP).

Time complexity:

- Displaying the initial list takes O(n) time, where n is 4 (the length of the list).
- The loop continues as long as i is less than 4. In the worst case, it loops 4 times, each time doing a fixed number of actions.
- The actions within the loop are all quick, and the loop iterates 4 times no matter the input. This makes the entire code's time complexity O(1), meaning it's really fast.

Different Cases:

1. Best Case: When the list is all 0s initially, no actions are needed, and the time complexity is O(1).
2. Worst Case: When the list starts with non-zero values, all four actions happen (changing 1s to 0s and vice versa), and the loop repeats 4 times. Even then, the time complexity is O(1).
3. Average Case: Since the code's time complexity stays constant no matter the input, the average case time complexity is also O(1).