

**DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING**

**DATA STRUCTURES AND ALGORITHM**



**CS 104**

**LAB FILE**

**SUBMITTED TO:**

**MR. JATIN SHARMA**

**SUBMITTED BY:**

**SAMIR GUPTA 23/CS/498**

# INDEX

SNO.	OBJECTIVE	DATE	SIGN
1.	Write a program to Implement Linear Search in the C/C++ programming language.	23.02.24	
2.	Write a program to Implement Binary Search in the C/C++ programming language. Assume the list is already sorted.	23.02.24	
3.	Write a program to insert an element at the mid-position in the One-dimensional array.	23.02.24	
4.	Write a program to delete a given row in the two-dimensional array.	23.02.24	
5.	Write a program to implement a stack data structure and perform its operations.	23.02.24	
6	Write a program to implement two stacks using a single array.	23.02.24	
7.	Write a program to reverse a 5-digit number	01.03.24	
8.	Write a program to convert decimal to binary and vice versa.	01.03.24	
9.	Write a program to find the minimum element of the stack in constant time using extra space.	05.04.24	
10.	Write a program to find the minimum element of the stack in constant time without using extra space.	05.04.24	

11.	Write a program to implement Queue Data Structure.	05.04.24	
12.	Write a program to reverse the first k elements of a given Queue.	05.04.24	

13.	Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.	05.04.24	
14.	Write a Program to check whether the given tree is a Binary Search Tree or not.	05.04.24	
15.	Write a Program to check whether the given tree is a Binary Search Tree or not.	05.04.24	

## **EXPERIMENT-1**

**AIM** - Write a program to Implement Linear Search in the C/C++ programming language.

### **ALGORITHM -**

1. Select the first element of the array.
2. Compare the target element with the selected element.
3. If the target element is found, return the index.
4. If the target element is not found after iterating through the entire array, return -1.

### **CODE -**

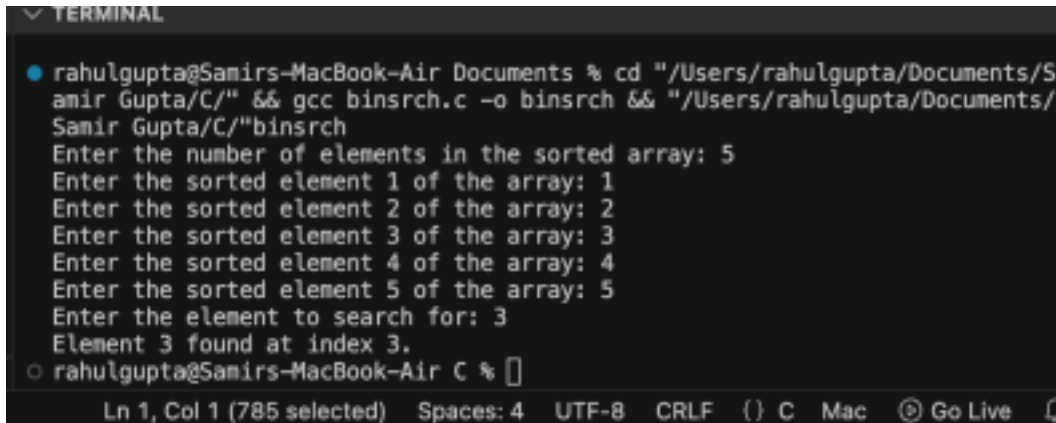
```
#include <stdio.h>
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
int main() {
    int n, target;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the element %d of the array: ", i+1);
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to search for: ");
    scanf("%d", &target);
    int result = linearSearch(arr, n, target);
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
```

```

    printf("Element %d not found in the array.\n", target);
}
return 0;
}

```

Output:



```

rahulgupta@Samirs-MacBook-Air Documents % cd "/Users/rahulgupta/Documents/S
amir Gupta/C/" && gcc binsrch.c -o binsrch && "/Users/rahulgupta/Documents/
Samir Gupta/C/"binsrch
Enter the number of elements in the sorted array: 5
Enter the sorted element 1 of the array: 1
Enter the sorted element 2 of the array: 2
Enter the sorted element 3 of the array: 3
Enter the sorted element 4 of the array: 4
Enter the sorted element 5 of the array: 5
Enter the element to search for: 3
Element 3 found at index 3.
rahulgupta@Samirs-MacBook-Air C %

```

## **EXPERIMENT-2**

**AIM** - Write a program to Implement Binary Search in the C/C++ programming language. Assume the list is already sorted.

### **ALGORITHM -**

1. Find the middle element of the sorted array.
2. If the middle element is the required element, return its index.
3. If the target is less than the middle element, repeat the search on the left half of the array.
4. If the target is greater than the middle element, repeat the search on the right half of the array.
5. Continue this process until the target is found.

### **CODE -**

```

#include <stdio.h>
int binarySearch(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {

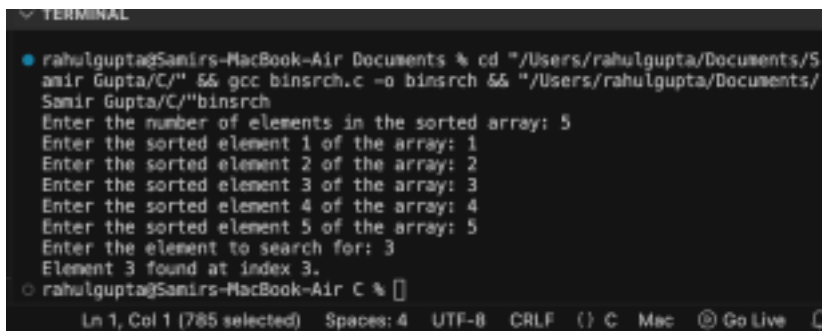
```

```

        return mid;
    }
    if (arr[mid] > target) {
        return binarySearch(arr, left, mid - 1, target);
    }
    return binarySearch(arr, mid + 1, right, target);
}
return -1;
}
int main() {
    int n, target;
    printf("Enter the number of elements in the sorted array: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the sorted element %d of the array: ", i+1);
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to search for: ");
    scanf("%d", &target);
    int result = binarySearch(arr, 0, n - 1, target);
    if (result != -1) {
        printf("Element %d found at index %d.\n", target,
result+1); } else {
        printf("Element %d not found in the array.\n", target);
    }
    return 0;
}

```

Output:



```

rahulgupta@Sanirs-MacBook-Air Documents % cd "/Users/rahulgupta/Documents/S
amir Gupta/C/" && gcc binsrch.c -o binsrch && "/Users/rahulgupta/Documents/
Samir Gupta/C/"binsrch
Enter the number of elements in the sorted array: 5
Enter the sorted element 1 of the array: 1
Enter the sorted element 2 of the array: 2
Enter the sorted element 3 of the array: 3
Enter the sorted element 4 of the array: 4
Enter the sorted element 5 of the array: 5
Enter the element to search for: 3
Element 3 found at index 3.
rahulgupta@Sanirs-MacBook-Air C % 

```

## EXPERIMENT-3

**AIM** - Write a program to insert an element at the mid-position in the One-dimensional array.

## ALGORITHM -

1. Calculate the mid-position of the array.
2. Shift elements from the mid-position to the end of the array one position to the right.
3. Insert the new element at the mid-position.

## CODE -

```
#include<stdio.h>
void display(int arr[],int n){
    for(int i = 0; i < n;i++){
        printf("%d\n",arr[i]);
    }
    printf("\n");
}

int indInsertion(int arr[],int size, int element, int capacity, int index){
    if(size>=capacity){
        return -1;
    }
    for(int i = size-1;i>=index;i--){
        arr[i+1]=arr[i];
    }
    arr[index]=element;
    return 1;
}

int main(){
    int arr[100]={7, 8, 12, 17, 88};
    int size = 5, element = 45, index=4;
    display(arr , size);
    indInsertion(arr , size, element, 100, index);
    size+=1;
    display(arr, size);

    return 0;
}
```

## OUTPUT -

```
pta/Documents/Samir Gupta/DSA(2nd sem)/programs
rayInsertoin.c -o ArrayInsertoin && "/Users/ra
ments/Samir Gupta/DSA(2nd sem)/programs/"ArrayI
7
8
12
17
88

7
8
12
17
45
88
```

## EXPERIMENT-4

**AIM** - Write a program to delete a given row in the two-dimensional array.

## ALGORITHM -

1. Move all rows below the deleted row one position up.
2. Decrement the total number of rows.

## CODE -

```
#include<stdio.h>
#define ROW 3
```



```

#define COL 4
void display(int arr[][COL],int rows){
    for(int i = 0; i < rows; i++){
        for(int j=0; j < COL; j++){
            printf("%d\t",arr[i][j]);
        }
        printf("\n");
    }
}
void deleteRow(int arr[][COL],int *rows, int index){
    if(index<0 || index >= *rows){
        printf("Invalid row index\n");
        return;
    }
    for(int i = index; i<*rows-1; i++){
        for(int j=0; j<COL; j++){
            arr[i][j] = arr[i+1][j];
        }
    }
    (*rows)--;
}
int main(){
    int rows = ROW;
    int arr[ROW][COL]= {{1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}};
    printf("Original Array:\n");
    display(arr, rows);
    deleteRow(arr, &rows, 1);
    printf("\nAfter deleting row 1:\n");
    display(arr, rows);
    return 0;
}

```

**OUTPUT -**

```

e.c -o rowDelete && "/Users/rahulgupta/Documents/SA
a/D
SA(2nd sem)/programs/"rowDelete
Original Array:
1      2      3      4
5      6      7      8
9      10     11     12

After deleting row 1:
1      2      3      4
9      10     11     12

```

## **EXPERIMENT-5**

**AIM** - Write a program to implement a stack data structure and perform its operations.

### **ALGORITHM -**

1. Initialize a stack data structure.
2. Implement push operation to insert elements onto the stack.
3. Implement pop operation to remove elements from the stack.
4. Implement peek operation to view the top element of the stack.
5. Implement isEmpty operation to check if the stack is empty.
6. Implement isFull operation to check if the stack is full.

### **CODE -**

```

#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100
typedef struct {

```

```

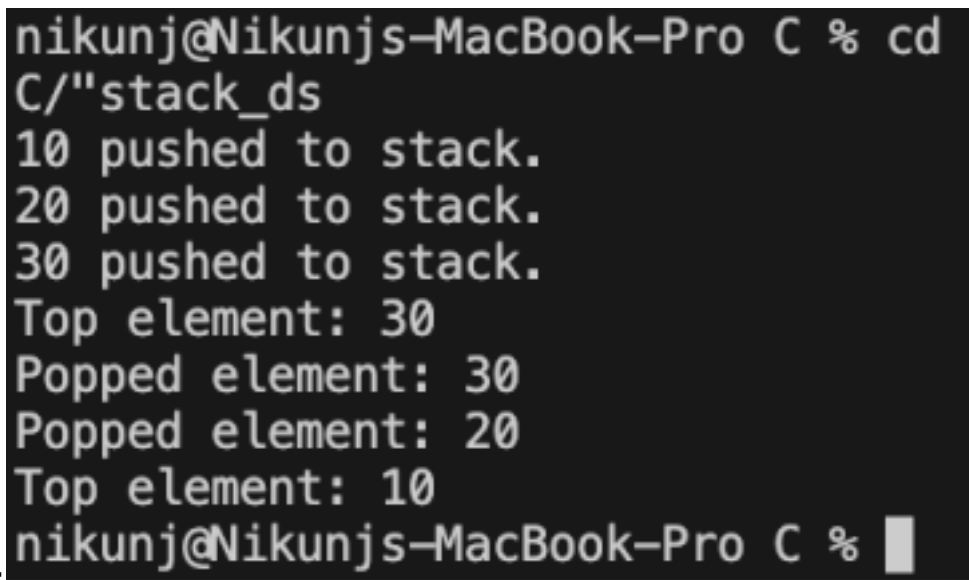
    int data[MAX_SIZE];
    int top;
} Stack;
void initStack(Stack *s) {
    s->top = -1;}
bool isEmpty(Stack *s) {
    return s->top == -1;}
bool isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;}
void push(Stack *s, int element) {
    if (!isFull(s)) {
        s->data[++s->top] = element;
        printf("%d pushed to stack.\n", element);
    } else {
        printf("Stack overflow! Unable to push %d\n", element);
    }
}
int pop(Stack *s) {
    if (!isEmpty(s)) {
        return s->data[s->top--];
    } else {
        printf("Stack underflow! Unable to pop.\n");
        return -1;
    }
}
int peek(Stack *s) {
    if (!isEmpty(s)) {
        return s->data[s->top];
    } else {
        printf("Stack is empty!\n");
        return -1;
    }
}

```

```

int main() {
    Stack s;
    initStack(&s);
    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    printf("Top element: %d\n", peek(&s));
    printf("Popped element: %d\n", pop(&s));
    printf("Popped element: %d\n", pop(&s));
    printf("Top element: %d\n", peek(&s));
    return 0;
}

```



```

nikunj@Nikunjs-MacBook-Pro C % cd '
C/"stack_ds
10 pushed to stack.
20 pushed to stack.
30 pushed to stack.
Top element: 30
Popped element: 30
Popped element: 20
Top element: 10
nikunj@Nikunjs-MacBook-Pro C % █

```

**OUTPUT -**

### **EXPERIMENT-6**

**AIM** - Write a program to implement two stacks using a single array.

**ALGORITHM** -

1. Divide the array into two halves to represent two stacks.
2. Implement push and pop operations for each stack separately.
3. Keep track of the top index of each stack

## CODE -

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100
typedef struct {
    int data[MAX_SIZE];
    int top1;
    int top2;
} TwoStacks;

void initTwoStacks(TwoStacks *ts) {
    ts->top1 = -1; // Top of stack 1
    ts->top2 = MAX_SIZE; // Top of stack 2
}

bool isFull(TwoStacks *ts) {
    return ts->top1 == ts->top2 - 1;
}

bool isEmpty1(TwoStacks *ts) {
    return ts->top1 == -1;
}

bool isEmpty2(TwoStacks *ts) {
    return ts->top2 == MAX_SIZE;
}

void push1(TwoStacks *ts, int element) {
    if (!isFull(ts)) {
        ts->data[++ts->top1] = element;
        printf("%d pushed to stack 1.\n", element);
    } else {
        printf("Stack 1 overflow! Unable to push %d\n", element);
    }
}

void push2(TwoStacks *ts, int element) {
```

```

    if (!isFull(ts)) {
        ts->data[--ts->top2] = element;
        printf("%d pushed to stack 2.\n", element);
    } else {
        printf("Stack 2 overflow! Unable to push %d\n", element);
    }
}
}
int pop1(TwoStacks *ts) {
    if (!isEmpty1(ts)) {
        return ts->data[ts->top1--];
    } else {
        printf("Stack 1 underflow! Unable to pop.\n");
        return -1;
    }
}
int pop2(TwoStacks *ts) {
    if (!isEmpty2(ts)) {
        return ts->data[ts->top2++];
    } else {
        printf("Stack 2 underflow! Unable to pop.\n");
        return -1;
    }
}
int main() {
    TwoStacks ts;
    initTwoStacks(&ts);
    push1(&ts, 10);
    push1(&ts, 20);
    push2(&ts, 30);
    printf("Popped element from stack 1: %d\n", pop1(&ts));
    printf("Popped element from stack 2: %d\n", pop2(&ts));
    return 0;
}

```

```

nikunj@Nikunjs-MacBook-Pro C % cd
ents/Coding/VS code/C/"tempCodeRun
10 pushed to stack 1.
20 pushed to stack 1.
30 pushed to stack 2.
Popped element from stack 1: 20
Popped element from stack 2: 30
nikunj@Nikunjs-MacBook-Pro C % █

```

**OUTPUT -**

## **EXPERIMENT-7**

**AIM -** Write a program to reverse a 5-digit number

**ALGORITHM -**

1. Extract each digit of the 5-digit number iteratively.
2. Reconstruct the reversed number by appending the digits in reverse order.
3. Display the reversed number.

**CODE -**

```

#include<stdio.h>
int reverse(int num){
    int rev=0;
    while(num!=0){
        rev=rev*10+num%10;
        num=num/10;
    }
    return rev;
}
int main(){
    int num,rev;
    printf("Enter a 5 digit number: ");
    scanf("%d",&num);
    rev=reverse(num);
    printf("%d",rev);
    return 0;
}

```

}

## OUTPUT -

```
rahulgupta@Samirs-MacBook-Air programs % cd "/Users/rahulgupta/Documents/Samir Gupta/DSA(2nd sem)/programs"
rahulgupta@Samirs-MacBook-Air programs % gcc p7.c -o p7 && "/Users/rahulgupta/Documents/Samir Gupta/DSA(2nd sem)/programs/"p7
Enter a 5 digit number: 12345
54321%
```

## EXPERIMENT-8

**AIM** - Write a program to convert decimal to binary and vice versa.

## ALGORITHM -

- Algorithm (Decimal to Binary):

1. Initialize variables to hold binary number and remainder.
2. Perform repeated division of the decimal number by 2.
3. Record the remainders to obtain the binary equivalent.
4. Reverse the binary equivalent to get the final binary number.
5. Display the binary number.

- Algorithm (Binary to Decimal):

1. Initialize variables to hold decimal number, base, and remainder.
2. Perform the conversion by multiplying each binary digit by powers of 2.
3. Sum the results to obtain the decimal equivalent.
4. Display the decimal number.

## CODE -



## ● Decimal to Binary

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

```
long decimalToBinary(int decimal) {  
    long binary = 0;  
    int remainder, base = 1;  
    while (decimal > 0) {  
        remainder = decimal % 2;  
        binary += remainder * base;  
        decimal /= 2;  
        base *= 10;  
    }  
  
    return binary;  
}
```

```
int main() {  
    int decimalNumber;  
    printf("Enter a decimal number: ");  
    scanf("%d", &decimalNumber);  
  
    long binaryNumber = decimalToBinary(decimalNumber);  
    printf("Binary equivalent: %ld\n", binaryNumber);  
  
    return 0;  
}
```

## ● Binary to Decimal

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

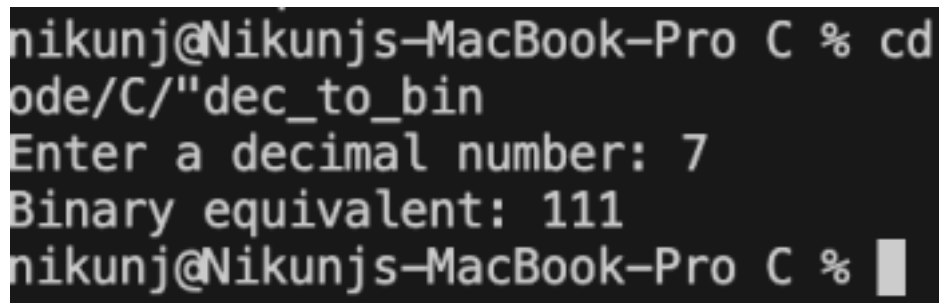
```
int binaryToDecimal(long binary) {
```

```

int decimal = 0, base = 1, remainder;
while (binary > 0) {
    remainder = binary % 10;
    decimal += remainder * base;
    binary /= 10;
    base *= 2;
}
return decimal;
}
int main() {
    long binaryInput;
    printf("Enter a binary number: ");
    scanf("%ld", &binaryInput);
    int decimalResult = binaryToDecimal(binaryInput);
    printf("Decimal equivalent: %d\n", decimalResult);
    return 0;
}

```

## OUTPUT -

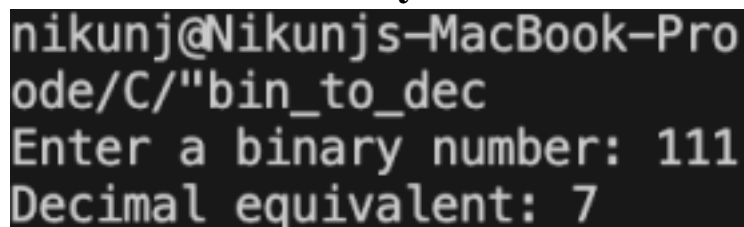


```

nikunj@Nikunjs-MacBook-Pro C % cd
ode/C/"dec_to_bin
Enter a decimal number: 7
Binary equivalent: 111
nikunj@Nikunjs-MacBook-Pro C % █

```

### • Decimal to Binary



```

nikunj@Nikunjs-MacBook-Pro
ode/C/"bin_to_dec
Enter a binary number: 111
Decimal equivalent: 7

```

## Binary to Decimal

## EXPERIMENT-9

**AIM** - Write a program to find the minimum element of the stack in constant time using extra space.

### **ALGORITHM** -

1. Maintain an auxiliary stack to keep track of the minimum elements.
2. Push each element onto the auxiliary stack while maintaining the minimum element.
3. For each push operation, if the element is smaller than or equal to the current minimum, push it onto the auxiliary stack.
4. For each pop operation, if the popped element is equal to the current minimum, pop from the auxiliary stack as well.
5. The top of the auxiliary stack always contains the minimum element of the stack.

### **CODE** -

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

#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int top;
    int minStack[MAX_SIZE];
    int minTop;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
    s->minTop = -1;
}

void push(Stack *s, int element) {
    if (s->top >= MAX_SIZE - 1) {
```

```

printf("Stack overflow!\n");
return;
}
s->top++;
s->data[s->top] = element;
if (s->minTop == -1 || element <= s->minStack[s->minTop]) {
    s->minTop++;
s->minStack[s->minTop] = element;
}
}

```

```

int pop(Stack *s) {
    if (s->top < 0) {
printf("Stack underflow!\n");
return -1;
}
int popped = s->data[s->top];
s->top--;
if (popped == s->minStack[s->minTop])
s->minTop--;
return popped;
}

```

```

int getMin(Stack *s) {
    if (s->minTop >= 0)
return s->minStack[s->minTop];
else {
printf("Stack is empty!\n");
return -1;
}
}

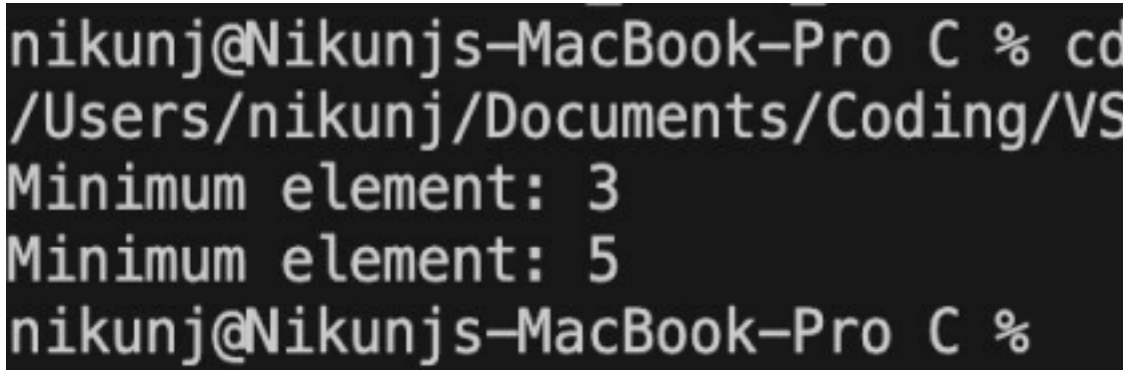
```

```

int main() {
Stack s;
initStack(&s);
push(&s, 10);
push(&s, 5);
push(&s, 15);
push(&s, 3);

```

```
printf("Minimum element: %d\n", getMin(&s)); // Should print 3
pop(&s); // Pop the top element
pop(&s); // Pop the top element
printf("Minimum element: %d\n", getMin(&s)); // Should print 5
```



```
nikunj@Nikunjs-MacBook-Pro C % cd
/Users/nikunj/Documents/Coding/VS
Minimum element: 3
Minimum element: 5
nikunj@Nikunjs-MacBook-Pro C %
```

```
return 0;}
```

## OUTPUT -

### EXPERIMENT-10

**AIM** - Write a program to find the minimum element of the stack in constant time without using extra space.

## ALGORITHM -

1. Maintain a variable to store the current minimum element.
2. For each push operation, compare the incoming element with the current minimum.
3. If the incoming element is smaller than the current minimum, update the current minimum.
4. For each pop operation, if the popped element is equal to the current minimum, update the current minimum by traversing the stack.

## CODE -

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

```
#define MAX_SIZE 1000
```

```
// Structure to represent the stack
```

```

typedef struct {
    int stack[MAX_SIZE];
    int min[MAX_SIZE]; // Stack to keep track of minimum elements
    int top;
} MinStack;

// Function to initialize the stack
MinStack* create() {
    MinStack* obj = (MinStack*)malloc(sizeof(MinStack));
    obj->top = -1;
    return obj;
}

// Function to push element onto the stack
void push(MinStack* obj, int x) {
    obj->top++;
    obj->stack[obj->top] = x;
    obj->min[obj->top] = (obj->top == 0 || x <= obj->min[obj->top - 1]) ? x : obj->min[obj->top - 1];
}

// Function to pop element from the stack
void pop(MinStack* obj) {
    if (obj->top >= 0) {
        obj->top--;
    }
}

// Function to get the top element of the stack
int top(MinStack* obj) {
    if (obj->top >= 0) {
        return obj->stack[obj->top];
    }
    return -1;
}

// Function to get the minimum element of the stack
int getMin(MinStack* obj) {
    if (obj->top >= 0) {
        return obj->min[obj->top];
    }
    return -1;
}

// Function to free the memory allocated for the stack
void freeStack(MinStack* obj) {
    free(obj);
}

int main() {
    MinStack* obj = create();

```

```

push(obj, -2);
push(obj, 0);
push(obj, -3);
printf("Minimum element in the stack: %d\n", getMin(obj)); // Output: -3

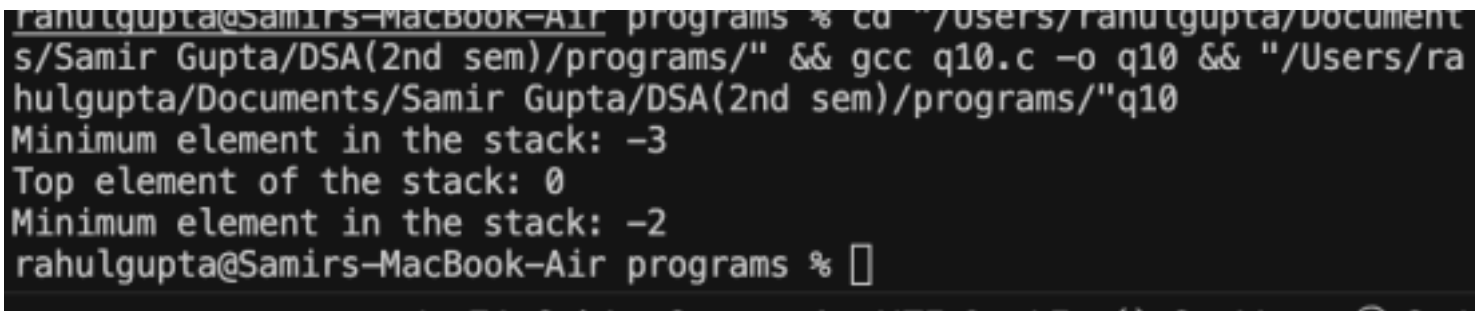
pop(obj);
printf("Top element of the stack: %d\n", top(obj)); // Output: 0
printf("Minimum element in the stack: %d\n", getMin(obj)); // Output: -2

freeStack(obj);

return 0;
}

```

## OUTPUT -



```

rahulgupta@Samirs-MacBook-Air programs % cd "/Users/rahulgupta/Documents/Samir Gupta/DSA(2nd sem)/programs/" && gcc q10.c -o q10 && "/Users/rahulgupta/Documents/Samir Gupta/DSA(2nd sem)/programs/"q10
Minimum element in the stack: -3
Top element of the stack: 0
Minimum element in the stack: -2
rahulgupta@Samirs-MacBook-Air programs %

```

## EXPERIMENT-11

**AIM** - Write a program to implement Queue Data Structure.

## ALGORITHM -

1. Maintain a fixed-size array to store the elements of the queue.
2. Maintain two pointers, 'front' and 'rear', to keep track of the front and rear of the queue.
3. Implement enqueue operation to insert elements at the rear of the queue.
4. Implement dequeue operation to remove elements from the front of the queue.
5. Implement isEmpty operation to check if the queue is empty.
6. Implement isFull operation to check if the queue is full.

## CODE -

```

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

struct node* f = NULL;
struct node* r = NULL;
struct node{
    int data;
    struct node * next;
};

void traversal(struct node *ptr){
    printf("Printing the elements of this linked list\n");
    while(ptr!=NULL){
        printf("%d\n",ptr->data);
        ptr = ptr->next;
    }
}

void enqueue(int val){
    struct node* n = (struct node*)malloc(sizeof(struct node));
    if(n==NULL){
        printf("Queue is Full");
    }
    else{
        n->data = val;
        n->next = NULL;
        if(f==NULL){
            f=r=n;
        }
        else{
            r->next = n;
            r=n;
        }
    }
}

int dequeue(){
    int val = -1;
    struct node* ptr = f;
    if(f==NULL){
        printf("Queue is empty");
    }else{
        f = f->next;
        val = ptr->data;
        free(ptr);
    }
}

```

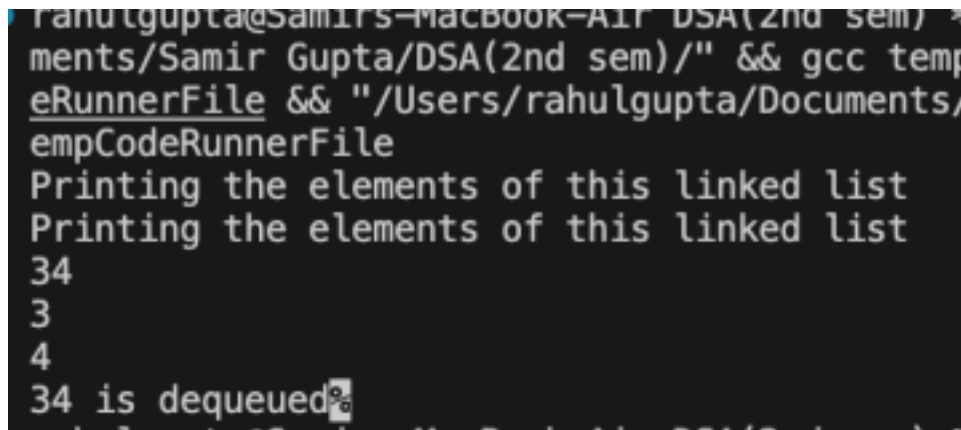


```

}
return val;
}
int main(){
traversal(f);
enqueue(34);
enqueue(3);
enqueue(4);
traversal(f);
printf("%d is dequeued", dequeue());
return 0;
}

```

## OUTPUT -



```

rahulgupta@SamirS-MacBook-Air:~/DSA(2nd sem) % gcc tempCodeRunnerFile.c -o tempCodeRunnerFile
Printing the elements of this linked list
Printing the elements of this linked list
34
3
4
34 is dequeued%

```

## EXPERIMENT-12

**AIM** - Write a program to reverse the first k elements of a given Queue.

## **ALGORITHM -**

1. Initialize two pointers, current and prev, and set them to the front of the queue. 2. Traverse the queue while keeping track of the count of nodes visited (count). Stop when count equals k or when current becomes NULL.
3. During the traversal, reverse the pointers of each node: set the next pointer of the current node to point to the previous node (prev), and update prev and current accordingly.
4. After reversing the first k nodes, update the pointers of the queue: a. Set the next pointer of the front of the queue to point to the node following the last

node of the reversed segment.

b. Update the front pointer of the queue to point to the last node of the reversed segment.

c. If the next node after the last node of the reversed segment is NULL, update the rear pointer of the queue to point to the last node.

5. The first k elements of the queue are now reversed.

## CODE -

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

// Structure to represent a node in the queue
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Structure to represent the queue
typedef struct {
    Node* front;
    Node* rear;
} Queue;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize the queue
void initQueue(Queue* q) {
    q->front = NULL;
    q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(Queue* q) {
    return q->front == NULL;
}
```

// Function to enqueue an element into the queue

```
void enqueue(Queue* q, int val) {
    Node* newNode = createNode(val);
    if (isEmpty(q)) {
        q->front = newNode;
    } else {
        q->rear->next = newNode;
    }
    q->rear =
newNode; }
```

// Function to dequeue an element from the queue

```
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        exit(EXIT_FAILURE);
    }
    Node* temp = q->front;
    int val = temp->data;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);

    }
    return val;
}
```

// Function to reverse a linked list

```
Node* reverseList(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

// Function to reverse the first k elements of the queue

```
void reverseFirstK(Queue* q, int k) {
    if (isEmpty(q) || k <= 0) {
        printf("Invalid operation!\n");
        return;
    }

    Node* current = q->front;
```

```

Node* prev = NULL;
Node* next = NULL;
int count = 0;

// Traverse to the kth node
while (current != NULL && count < k) {
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;

    ++;
    count
}

// Update pointers
q->front->next = current;
q->front = prev;

// Update rear pointer if needed
if (current == NULL) {
    q->rear =
}
    q->front;
}

// Function to display the queue
void displayQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    Node* temp = q->front;
    while (temp != NULL) {
        printf("%d ",
            temp->data); temp =
            temp->next;

        "\n");
    }
    }printf(

int main() {
    Queue q;
    initQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);

```

```

printf("Queue before reversing first 3 elements:
"); displayQueue(&q);

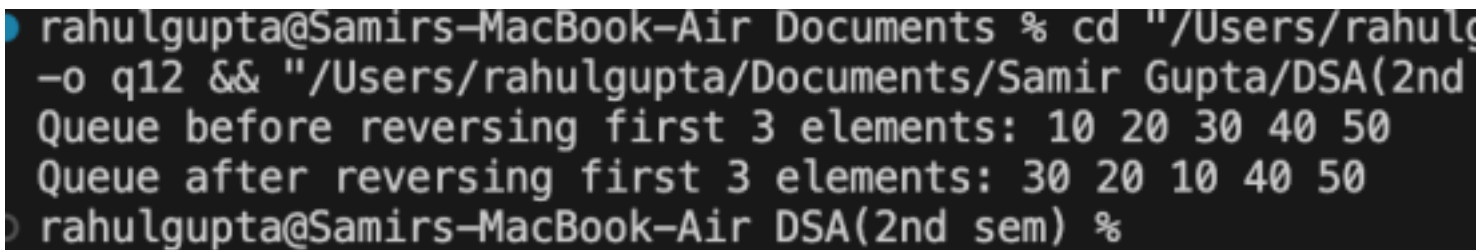
// Reverse the first 3 elements
reverseFirstK(&q, 3);

printf("Queue after reversing first 3 elements:
"); displayQueue(&q);

return 0;
}

```

## OUTPUT -



```

rahulgupta@Samirs-MacBook-Air Documents % cd "/Users/rahulg
-o q12 && "/Users/rahulgupta/Documents/Samir Gupta/DSA(2nd
Queue before reversing first 3 elements: 10 20 30 40 50
Queue after reversing first 3 elements: 30 20 10 40 50
rahulgupta@Samirs-MacBook-Air DSA(2nd sem) %

```

## EXPERIMENT-13

**AIM** - Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.

## ALGORITHM -

1. Define the Node structure:
  - a. The Node structure should have two members: data to store the data of the node, and next to point to the next node in the linked list.
2. Define traversal function:
  - a. This function takes the head pointer of the linked list as input.
  - b. Traverse the linked list from the head node to the last node.
  - c. Print the data of each node as you traverse.
3. Define insertion functions:
  - a. Each insertion function should take the head pointer of the linked list and necessary parameters (such as data and index) as input.
  - b. Allocate memory for the new node using malloc.
  - c. Set the data of the new node.
  - d. Perform the insertion operation according to the specified requirement: i.

For insertion at the beginning, update the next pointer of the new node to point to the current head node, and update the head pointer to point to the new node.

- ii. For insertion at a specified index, traverse the linked list to the node at the (index - 1) position, update the next pointers accordingly, and insert the new node.
- iii. For insertion at the end, traverse the linked list to the last node, update the next pointer of the last node to point to the new node, and update the new node's next pointer to NULL.
- iv. For insertion after a specified node, update the next pointers of the new node and the specified node accordingly to insert the new node after the specified node.

4. Define the main function:

- a. Create nodes head, second, and third and link them together to form an initial linked list.
- b. Call the traversal function to display the elements of the linked list before any insertions.
- c. Perform insertion operations using the defined functions.
- d. Call the traversal function again to display the elements of the linked list after all insertions.

5. Memory management:

- a. Free dynamically allocated memory for nodes to avoid memory leaks.  
However, the provided code does not include the deallocation of memory.

## CODE -

```
#include<stdio.h>
#include<stdlib.h>
struct Node{
    int data;
    struct Node * next;
};
void traversal(struct Node * ptr){
    printf("Elements are: \n");
    while(ptr!=NULL){
        printf("%d\n", ptr->data);
        ptr=ptr->next;
    }
}
```

```

}
struct Node * insertAtFirst(struct Node*head, int data){
    struct Node * ptr = (struct Node *)malloc(sizeof(struct Node));
    ptr->next = head;
    ptr->data = data;
    return ptr;
}

```

```

struct Node * insertAtIndex(struct Node*head, int data, int index){
    struct Node * ptr = (struct Node *)malloc(sizeof(struct Node));
    struct Node * p= head;
    int i = 0;

    while(i!=index-1){
        p = p->next;
        i++;
    }
    ptr->data = data;
    ptr->next = p->next;
    p->next = ptr;
    return head;
}

```

```

struct Node * insertAtEnd(struct Node * head, int data){
    struct Node * ptr = (struct Node *)malloc(sizeof(struct Node));
    struct Node * p = (struct Node*)malloc(sizeof(struct Node));
    ptr->data = data;
    while(p->next!=NULL){
        p = p->next;
    }
    p->next = ptr;
    ptr->next = NULL;
    return head;
}

```

```

struct Node * insertAfterNode(struct Node * head, struct Node * prevNode, int data){
    struct Node * ptr = (struct Node *)malloc(sizeof(struct Node));
    ptr->data = data;
    ptr->next = prevNode->next;
    prevNode->next = ptr;
    return head;
}

```

```

int main(){
    struct Node * head;
    struct Node * second;

```

```
struct Node * third;
//Allocate memory for nodes in the linked list in heap
head =(struct Node*)malloc(sizeof(struct Node));
second =(struct Node*)malloc(sizeof(struct Node));
third =(struct Node*)malloc(sizeof(struct Node));

//Link first and second
head->data = 7;
head->next = second;
//Link second and third
second->data = 17;
second->next = third;
//Terminate at third
third->data = 70;
third->next = NULL;

traversal(head);
head = insertAtFirst(head, 56);
head = insertAtIndex(head, 42, 1);
head = insertAtEnd(head, 99);
head = insertAfterNode(head, second, 80);
traversal(head);
return 0;
}
```

**OUTPUT -**



```

rahulgupta@Samirs-MacBook-Air DSA(2nd sem) % cd "/User
& gcc LinkedListInsertion.c -o LinkedListInsertion &&
ms/"LinkedListInsertion
Elements are:
7
17
70
Elements are:
56
42
7
17
80
70
rahulgupta@Samirs-MacBook-Air programs %

```

## **EXPERIMENT-14**

**AIM** - Write a Program to check whether the given tree is a Binary Search Tree or not.

### **ALGORITHM -**

1. Start with the root of the tree.
2. Initialize an empty stack.
3. Repeat the following steps until the current node is NULL and the stack is empty:
  - a. Traverse the left subtree of the current node by pushing all nodes encountered onto the stack.
  - b. Pop a node from the stack and set it as the current node.
  - c. Check if the current node's value is less than or equal to the previously visited node's value. If so, return `false` as the tree is not a BST.
  - d. Update the previously visited node to the current node.
  - e. Traverse the right subtree of the current node.
4. If the traversal completes without encountering any violation of the BST property, return `true` indicating that the tree is a BST.

### **1. CODE -**

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
struct TreeNode {
    int value;
    struct TreeNode* left;
    struct TreeNode* right;
};
struct StackNode {
    struct TreeNode* node;
    struct StackNode* next;
};
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
void push(struct StackNode** top, struct TreeNode* node) {
    struct StackNode* stackNode = (struct StackNode*)malloc(sizeof(struct StackNode));
    stackNode->node = node;
    stackNode->next = *top;
    *top = stackNode;
}
struct TreeNode* pop(struct StackNode** top) {
    if (*top == NULL)
        return NULL;
    struct StackNode* temp = *top;
    struct TreeNode* popped = temp->node;
    *top = temp->next;
    free(temp);
    return popped;
}
// Function to check if the tree is a Binary Search Tree
(BST) bool isBST(struct TreeNode* root) {
    struct StackNode* stack = NULL;
    struct TreeNode* prev = NULL;
    while (root != NULL || stack != NULL) {
        // Reach the leftmost node of the current subtree
        while (root != NULL) {
            push(&stack, root);
            root = root->left;
        }
        root = pop(&stack);
        if (prev != NULL && root->value <= prev->value)
            return false;
        prev = root;
        root = root->right;
    }
    return true;
}

```

```

}int main() {
    struct TreeNode* root = createNode(5);
    root->left = createNode(3);
    root->right = createNode(8);
    root->left->left = createNode(2);
    root->left->right = createNode(4);
    root->right->left = createNode(7);
    root->right->right = createNode(9);

    if (isBST(root))
        printf("The given tree is a Binary Search Tree.\n");
    else
        printf("The given tree is not a Binary Search
                Tree.\n");

    return
    0;
}

```

## OUTPUT -



## EXPERIMENT-15

**AIM** - Write an Program to count the number of leaf nodes in an AVL tree.

## ALGORITHM -

1. Start with the root of the AVL tree.
2. If the root is NULL, return 0.
3. If both the left and right children of the root are NULL, return 1 (as it is a leaf node).
4. Recursively count the number of leaf nodes in the left subtree:
  - Set leaf\_count\_left = countLeafNodes(root->left).
5. Recursively count the number of leaf nodes in the right subtree:
  - Set leaf\_count\_right = countLeafNodes(root->right).
6. Add the counts of leaf nodes in the left and right subtrees:
  - Set total\_leaf\_count = leaf\_count\_left + leaf\_count\_right.
- 7.

Return the total\_leaf\_count as the result.

## CODE -

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

// Define the structure for a tree node
struct TreeNode {
    int value;
    struct TreeNode* left;
    struct TreeNode* right;
    int height;
};

// Function to create a new tree node
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1;
    return newNode;
}

// Function to count the number of leaf nodes in an AVL tree
int countLeafNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    // If the node is a leaf node, return 1
    if (root->left == NULL && root->right == NULL)
        return 1;

    // Recursively count leaf nodes in the left and right subtrees
    return countLeafNodes(root->left) + countLeafNodes(root->right);
}

int main() {
    // Construct an AVL tree
    struct TreeNode* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(15);
    root->left->left = createNode(3);
    root->left->right = createNode(7);
    root->right->left = createNode(12);
    root->right->right = createNode(20);

    // Count the number of leaf nodes
```

```
int leafCount = countLeafNodes(root);  
printf("Number of leaf nodes in the AVL tree: %d\n", leafCount);  
  
return 0;  
}
```

## OUTPUT -

