DHEMAJI ENGINEERING COLLEGE

DSA LAB REPORT

Submitted To:
Dhantu Buragohain Sir

Submitted By:
Name: Saranga Pani Dutta
Branch: COMPUTER SCIENCE AND ENGINEERING
Semester: 3rd
Roll Number: 2481105745

Exercise 1 Date: 20-08-2025

Aim:
To write a C program to perform a Linear Search on an array of integers entered by the user.

Code:

```c
/*To write a C program to perform a Linear Search on an array of integers entered by the user.*/
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
  for (int i = 0; i < n; i++) {
    if (arr[i] == key) {
      return i;
    }
}
  return -1;
}


int main() {
  int n, key;
    printf("Enter number of elements: ");
    scanf("%d", &n);
  int arr[n];
    printf("Enter %d elements: ", n);
    for(int i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
    }

  printf("Enter element to search: ");
  scanf("%d", &key);

  int result = linearSearch(arr, n, key);

  if (result == -1) {
    printf("Element not found.\n");
  }
  else {
    printf("Element found at index: %d\n", result);
  }
  return 0;
}
```

Output:

```
Enter number of elements: 6
Enter 6 elements: 1 2 4 7 3 5
Enter element to search: 7
Element found at index: 3

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

This algorithm checks each element sequentially until it finds the desired element or reaches the end of the array. This program doesn't work for duplicate elements.

Exercise 2 **Date: 20-08-2025**

Aim:
To write a C program to perform a Linear Search in an array that may contain duplicate elements.

Code:

```c
/*To write a C program to perform a Linear Search in an array that may contain duplicate
elements.*/

#include <stdio.h>
int main() {

  int n, key;
  int count = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);
  int a[n];
  printf("Enter %d elements: ", n);
  for (int i = 0; i<n; i++) {
    scanf("%d", &a[i]);
  }
  printf("Enter element to search: ");
  scanf("%d", &key);
  for (int i = 0; i <n; i++) {
    if (a[i] == key) {
      printf("%d found at position %d\n", key, i + 1);
      count++;
    }
  }
  if (count == 0) {
    printf("Element not found.\n");
  } else {
    printf("%d found %d times.\n", key, count);
  }
  return 0;

}
```

Output:

```
Enter number of elements: 7
Enter 7 elements: 1 2 3 3 3 5 2
Enter element to search: 2
2 found at position 2
2 found at position 7
2 found 2 times.

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

This program checks all the numbers in the list and shows every position where the searched number appears, unlike the program in Exercise 1.

Exercise 3 **Date: 20-08-2025**

Aim:
To write a C program that performs a Binary Search on a sorted array entered by the user.

Code:

```c
#include<stdio.h>

int BinarySearch(int arr[], int n, int key) {
  int low = 0, high = n-1;

  while(low <= high){
    int mid = low + (high - low) / 2;

    if(arr[mid] == key){
      return mid;
    }
    else if(arr[mid] < key){
      low = mid + 1;
    }
    else{
      high = mid - 1;
    }
  }
  return -1;
}
```

```c
int main(){
  int n, i, key;

  printf("Enter the number of elements in the array: ");
  scanf("%d", &n);


  int arr[n];


  printf("Enter %d sorted elements:\n", n);
  for (i = 0; i < n; i++){
    scanf("%d", &arr[i]);
  }

  printf("Enter the element to search: ");
  scanf("%d", &key);

  int result = BinarySearch(arr, n, key);

  if(result != -1) {
    printf("Element found at index %d.\n", result);
  }
  else {
  printf("Element not found in the array.\n");
  }
  return 0;
}
```

Output:

```
Enter the number of elements in the array: 5
Enter 5 sorted elements:
2 3 4 5 7
Enter the element to search: 5
Element found at index 3.

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

This program only works when the list is already sorted in increasing order. It keeps dividing the list into halves to find the searched number. It doesn't work if any duplicate items are entered by the user.

Exercise 4 Date: 20-08-2025

Aim:
To write a C program that performs a Binary Search on a sorted array containing duplicate elements.

Code:

```c
/*To write a C program that performs a Binary Search on a sorted array containing duplicate elements.*/


#include<stdio.h>

int Occurance_1(int arr[], int n, int key){
  int low =0, high = n -1;
  int result = -1;

  while(low <= high){
    int mid = low + (high - low)/2;
    if(arr[mid] == key){
      result = mid;
      high = mid - 1;
    }
    else if(arr[mid]<key){
      low = mid + 1;
    }
    else{
      high = mid - 1;
    }
  }
  return result;
}
```

```c
int Occurance_2(int arr[], int n, int key){
  int low =0, high = n -1;
  int result = -1;

  while(low <= high){
    int mid = low + (high - low)/2;
    if(arr[mid] == key){
      result = mid;
      low = mid + 1;
    }
    else if(arr[mid]<key){
      low = mid + 1;
    }
    else{
      high = mid - 1;
    }
  }
    return result;
}

 int main(){
   int n, key;

   printf("Enter the number of elements in the array: ");
   scanf("%d", &n);


   int arr[n];


   printf("Enter %d sorted elements: ", n);
   for (int i = 0; i < n; i++) {
   scanf("%d", &arr[i]);
 }

   printf("Enter the element to search: ");
   scanf("%d", &key);

   int first = Occurance_1(arr, n, key);
   int last = Occurance_2(arr, n, key);
   int count = last - first + 1;


   if(first != -1 && last != -1) {
     printf("Element found from index %d to %d.\n", first, last);
     printf("Total count of element %d is: %d\n", key, count);
   }
   else {
     printf("Element not found in the array.\n");
   }
   return 0;
 }
```

Output:

```
Enter the number of elements in the array: 7
Enter 7 sorted elements: 1 2 2 3 3 3 4
Enter the element to search: 3
Element found from index 3 to 5.
Total count of element 3 is: 3

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

This version (unlike the program in Exercise 3) finds the first and last position of a number if it appears multiple times in a sorted list.

Exercise 5 Date: 19-09-2025

Aim:
To write a C program to demonstrate a stack using an array, with push and pop functions.

Code:
```c
/*To write a C program to demonstrate a stack using an array, with push and pop functions.*/
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 100

struct stack{
  int arr[MAXSIZE];
  int top;
};

void push(struct stack *s, int item){
  if(s->top == MAXSIZE -1){
    printf("Overflow\n");
  }
  else{
    s->top++;
    s->arr[(s->top)] = item;
  }
}

int pop(struct stack *s){
  if(s->top == -1){
    printf("Underflow\n");
    return -1;
  }
  else{
    int popped_item = s->arr[(s->top)];
    s->top--;
    printf("Popped item: %d\n", popped_item);
  }
}
```

```c
int peek(struct stack *s){
  if(s->top == -1){
    printf("stack empty\n");
  }
  else{
    printf("Top item: %d\n", s->arr[s->top]);
  }
}

void PrintStack(struct stack *s){
  if(s->top == -1){
    printf("Stack is empty\n");
  }
  else{
    printf("Stack elements: ");
    for(int i = s->top; i>=0; i--){
      printf("%d ", s->arr[i]);
    }
    printf("\n");
  }

}

int main(){
  struct stack s;
  s.top = -1;

  push(&s, 10);
  push(&s, 20);
  pop(&s);
  push(&s, 50);
  push(&s, 3);
  pop(&s);
  peek(&s);
  PrintStack(&s);
}
```

Output:

```
Popped item: 20
Popped item: 3
Top item: 50
Stack elements: 50 10

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The stack operates on a Last-In-First-Out (LIFO) principle. The last number put in the stack comes out first when popped. When we try to add/remove elements to/from the stack when the stack is full/empty it shows appropriate messages.

Exercise 6 Date: 24-09-2025

Aim:
To write a C program demonstrating the implementation of a Queue data structure with functions to enqueue (insert) and dequeue (remove) elements.

Code:
```c
/*To write a C program demonstrating the implementation of a Queue data structure with
functions to enqueue (insert) and dequeue (remove) elements.*/

#include<stdio.h>
#include<stdlib.h>
#define MAX 5

int queue[MAX];
int front = -1;
int rear =-1;

void display() {
  if (front == -1 || front > rear) {
    printf("Queue is empty.\n");
    return;
  }
  printf("Queue elements: ");
    for (int i = front; i <= rear; i++){
      printf("%d ", queue[i]);
  }
  printf("\n");
}

void enqueue (int element) {
  if (rear == MAX - 1) {
    printf("Queue Overflow! Cannot enqueue %d\n", element);
    return;
  }
  if (front == -1)
    front = 0;
  rear++;
  queue[rear] = element;
  printf("Enqueued: %d\n", element);
}
```

```c
void dequeue() {
  if (front == -1 || front > rear) {
    printf("Queue Underflow! Cannot dequeue.\n");
    return;
}
  printf("Dequeued: %d\n", queue [front]);
  front++;
  if (front > rear)
    front = rear = -1;
}

int main() {
  enqueue (3);
  enqueue (2);
  display();

  dequeue();
  display();

  dequeue();
  dequeue(); // underflow

  enqueue (5);
  enqueue (4);
  enqueue (6);
  enqueue (9);
  enqueue (7);
  enqueue (12); // overflow
  display();

  return 0;
}
```

Output:

```
Enqueued: 3
Enqueued: 2
Queue elements: 3 2
Dequeued: 3
Queue elements: 2
Dequeued: 2
Queue Underflow! Cannot dequeue.
Enqueued: 5
Enqueued: 4
Enqueued: 6
Enqueued: 9
Enqueued: 7
Queue Overflow! Cannot enqueue 12
Queue elements: 5 4 6 9 7

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The program uses a simple array-based queue with front and rear pointers to manage enqueue and dequeue operations. Underflow is displayed when attempting to dequeue from an empty queue, and overflow occurs when trying to enqueue beyond MAX.

Exercise 7 Date: 08-10-2025

Aim:

To write a C program to create a singly linked list and implement insertion operations at the beginning, end and at a specific position of the list.

Code:

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

struct node {
    int data;
    struct node* next;
};

// create a new node
struct node* newNode(int item){
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = item;
    new_node->next = NULL;
    return new_node;
}

// insert at beginning
struct node* InsertAtBegin(struct node* head, int item){
    struct node* new_node = newNode(item);

    // if list is empty
    if(head == NULL){
        head = new_node;
        return head;
    }

    new_node->next = head;
    head = new_node;
    return head;
}
```

```c
// insert at end
struct node* InsertAtEnd(struct node* head, int item){
    struct node* new_node = newNode(item);

    if(head == NULL){
        head = new_node;
        return head;
    }

    struct node* curr = head;
    while(curr->next != NULL){
        curr = curr->next;
    }

    curr->next = new_node;
    return head;
}

// insert at specific position
struct node* InsertAtSpfcPos(struct node* head, int item, int position){
    struct node* new_node = newNode(item);

    if(head == NULL){
        head = new_node;
        return head;
    }

    struct node* temp = head;

    for(int i = 1; i < position - 1 && temp != NULL; i++){
        temp = temp->next;
    }

    if(temp == NULL){
        printf("Position is out of bound\n");
        return head;
    }

    new_node->next = temp->next;
    temp->next = new_node;
    return head;
}
```

```c
// display list
void display(struct node* head){
    if(head == NULL){
        printf("list is empty.\n");
        return;
    }

    struct node* curr = head;
    while(curr != NULL){
        printf("%d ->", curr->data);
        curr = curr->next;
    }
    printf(" NULL\n");
}

// main function
int main(){
    struct node* head = NULL;

    head = InsertAtBegin(head, 1);
    head = InsertAtBegin(head, 2);
    head = InsertAtBegin(head, 3);
    head = InsertAtEnd(head, 56);
    display(head);

    head = InsertAtSpfcPos(head, 7, 3);
    display(head);

    return 0;
}
```

Output:

```
3 ->2 ->1 ->56 -> NULL
3 ->2 ->7 ->1 ->56 -> NULL

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The function insertAtBeginning() inserts a new node at the start of the list by changing the head pointer. The function insertAtEnd() traverses the linked list to the last node and adds the new node at the end.

Exercise 8 Date: 15-10-2025

Aim:

To write a C program to implement a singly linked list that supports the following operations: insertion at the beginning, insertion at the end, insertion at a specific position, deletion from the beginning, deletion from the end, and deletion from a specific position.

Code:

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

struct node {
    int data;
    struct node* next;
};

// create a new node
struct node* newNode(int item){
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = item;
    new_node->next = NULL;
    return new_node;
}

// insert at beginning
struct node* InsertAtBegin(struct node* head, int item){
    struct node* new_node = newNode(item);

    // if list is empty
    if(head == NULL){
        head = new_node;
        return head;
    }

    new_node->next = head;
    head = new_node;
    return head;
}
```

```c
// insert at end
struct node* InsertAtEnd(struct node* head, int item){
    struct node* new_node = newNode(item);

    if(head == NULL){
        head = new_node;
        return head;
    }

    struct node* curr = head;
    while(curr->next != NULL){
        curr = curr->next;
    }

    curr->next = new_node;
    return head;
}
```

```c
// insert at specific position
struct node* InsertAtSpfcPos(struct node* head, int item, int position){
    struct node* new_node = newNode(item);

    if(head == NULL){
        head = new_node;
        return head;
    }

    struct node* temp = head;

    for(int i = 1; i < position - 1 && temp != NULL; i++){
        temp = temp->next;
    }

    if(temp == NULL){
        printf("Position is out of bound\n");
        return head;
    }

    new_node->next = temp->next;
    temp->next = new_node;
    return head;
}

// delete at beginning
struct node* DeleteAtBegin(struct node* head){
    if(head == NULL){
        printf("List is empty, nothing to delete.\n");
        return head;
    }

    struct node* temp = head;
    head = head->next;
    free(temp);

    return head;
}
```

```c
// delete at end
struct node* DeleteAtEnd(struct node* head){
    if(head == NULL){
        printf("List is empty, nothing to delete.\n");
        return head;
    }

    // if only one node exists
    if(head->next == NULL){
        free(head);
        head = NULL;
        return head;
    }

    struct node* curr = head;
    struct node* prev = NULL;

    while(curr->next != NULL){
        prev = curr;
        curr = curr->next;
    }

    prev->next = NULL;
    free(curr);

    return head;
}
```

```c
// delete at specific position
struct node* DeleteAtPosition(struct node* head, int position){
    if(head == NULL){
        printf("List is empty, nothing to delete.\n");
        return head;
    }

    // delete first node
    if(position == 1){
        struct node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    struct node* curr = head;
    struct node* prev = NULL;

    for(int i = 1; i < position && curr != NULL; i++){
        prev = curr;
        curr = curr->next;
    }

    if(curr == NULL){
        printf("Position out of range.\n");
        return head;
    }

    prev->next = curr->next;
    free(curr);

    return head;
}
```

```c
// display list
void display(struct node* head){
    if(head == NULL){
        printf("list is empty.\n");
        return;
    }

    struct node* curr = head;
    while(curr != NULL){
        printf("%d ->", curr->data);
        curr = curr->next;
    }
    printf(" NULL\n");
}

// main function
int main(){
    struct node* head = NULL;

    head = InsertAtBegin(head, 1);
    head = InsertAtBegin(head, 2);
    head = InsertAtBegin(head, 3);
    head = InsertAtEnd(head, 23);
    head = InsertAtSpfcPos(head, 7, 3);
    display(head);

    head = DeleteAtBegin(head);
    display(head);

    head = DeleteAtEnd(head);
    display(head);

    head = DeleteAtPosition(head, 3);
    display(head);

    return 0;
}
```

Output:

```
3 ->2 ->7 ->1 ->23 -> NULL
2 ->7 ->1 ->23 -> NULL
2 ->7 ->1 -> NULL
2 ->7 -> NULL

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

To delete a node at a particular position, we have to go through the list one by one until we reach that position.

Exercise 9 **Date: 17-10-2025**

Aim:

To perform different traversals on a binary tree, use the diagram given below (Figure 1) and write the code to display the in-order, pre-order, and post-order traversal sequences. Also, implement a function to find and print the height of the tree.

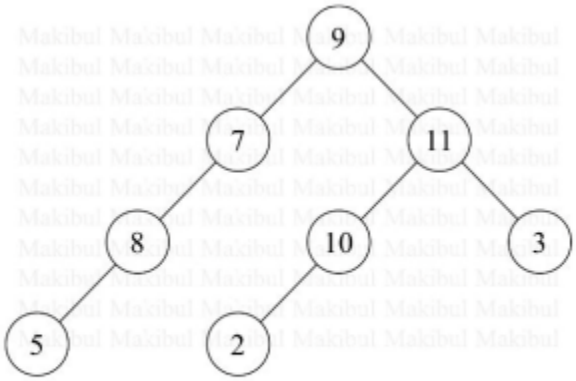


Figure 1

Code:

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

struct node {
    int data;
    struct node* left;
    struct node* right;
};

struct node* newTreeNode(int item) {
    struct node* newN = (struct node*) malloc(sizeof(struct node));
    newN->data = item;
    newN->left = NULL;
    newN->right = NULL;
    return newN;
}

void inOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}

void preOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
    printf("%d ", root->data);
    preOrder(root->left);
    preOrder(root->right);
}
```

```c
void postOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}

int CalcHeight(struct node* root) {
    if (root == NULL) {
        return -1;
    }
    int leftHeight = CalcHeight(root->left);
    int rightHeight = CalcHeight(root->right);

    if (leftHeight > rightHeight)
        return leftHeight + 1;
    else
        return rightHeight + 1;
}

int main() {
    struct node* root = newTreeNode(9);

    root->left = newTreeNode(7);
    root->left->left = newTreeNode(8);
    root->left->left->left = newTreeNode(5);

    root->right = newTreeNode(11);
    root->right->left = newTreeNode(10);
    root->right->left->left = newTreeNode(2);
    root->right->right = newTreeNode(3);

    printf("In-order: ");
    inOrder(root);

    printf("\nPre-order: ");
    preOrder(root);

    printf("\nPost-order: ");
    postOrder(root);

    printf("\nHeight of the tree is: %d\n", CalcHeight(root));

    return 0;
}
```

Output:

```
In-order: 5 8 7 9 2 10 11 3
Pre-order: 9 7 8 5 11 10 2 3
Post-order: 5 8 7 2 10 3 11 9
Height of the tree is: 3

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

This program builds a binary tree with several nodes and shows different ways to visit every node, called traversals (in-order, pre-order, post-order). The function getHeight calculates how many steps the longest path from the root to a leaf takes, which is called the height of the tree.

Exercise 10 Date: 24-10-2025

Aim:
To write a C program to create a binary search tree, display its nodes using in-order traversal, and provide an option for the user to search for a value in the tree.

Code:

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

struct node {
    int data;
    struct node* left;
    struct node* right;
};

struct node* newNode(int value) {
    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void inOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}
```

```c
struct node* Insert(struct node* root, int value) {
    if (root == NULL) {
        return newNode(value);
    }

    if (value < root->data) {
        root->left = Insert(root->left, value);
    } else if (value > root->data) {
        root->right = Insert(root->right, value);
    }

    return root;
}

int Search(struct node* root, int key) {
    if (root == NULL)
        return 0;

    if (key == root->data)
        return 1;

    if (key < root->data)
        return Search(root->left, key);
    else
        return Search(root->right, key);
}

int main() {
    struct node* root = NULL;

    // Insert values (similar approach as Makibul PDF)
    root = Insert(root, 2);
    root = Insert(root, 3);
    root = Insert(root, 9);
    root = Insert(root, 1);
    root = Insert(root, 8);

    printf("In-order: ");
    inOrder(root);

    int key;
    printf("\n\nEnter an item to search: ");
    scanf("%d", &key);

    if (Search(root, key)) {
        printf("\nItem %d is found.\n", key);
    } else {
        printf("\nItem %d is not found.\n", key);
    }

    return 0;
}
```

Output 1:

```
In-order: 1 2 3 8 9

Enter an item to search: 8

Item 8 is found.

c:\Users\sanju\Documents\CODING\DSA ( Saranga )
```

Output 2:

```
In-order: 1 2 3 8 9

Enter an item to search: 4

Item 4 is not found.

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

In-order traversal is used to display the tree nodes, which ensures the output is sorted in ascending order. The search operation allows the user to check if a particular value exists within the tree.

Exercise 11 **Date: 29-10-2025**

Aim:
To write a C++ program for the Bubble Sort algorithm.

Code:

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void printArray(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void bubbleSort(int arr[], int n) {
    for(int i = 0; i < n - 1; i++) {
      for(int j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

```c
int main() {
    int arr[] = {7, 2, 5, 1, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Original array: 7 2 5 1 9
Sorted array: 1 2 5 7 9

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The algorithm checks each pair of adjacent elements and swaps them if needed, thus pushing the largest element to the end in each pass. The use of the isSwap flag provides an optimization by terminating early if the array becomes sorted before completing all passes.

Exercise 12 Date: 29-10-2025

Aim:
To write a C++ function to implement the Selection Sort algorithm to sort an integer array.

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

void selectionSort(int arr[], int s){
    for(int i = 0; i < s - 1; i++){
        int minIndex = i;

        for(int j = i + 1; j < s; j++){
            if(arr[j] < arr[minIndex]){
                minIndex = j;
            }
        }

        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

int main(){
    int arr[] = {2, 3, 1, 7, 4, 9, 21, 6};
    int s = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    selectionSort(arr, s);

    printf("\nSorted array (Selection Sort): ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output:

```
Original array: 2 3 1 7 4 9 21 6
Sorted array (Selection Sort): 1 2 3 4 6 7 9 21
c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:
The algorithm always finds the smallest element from the unsorted part and places it at the beginning.

Exercise 13 Date: 29-10-2025

Aim:
To implement the Insertion Sort algorithm in C++ to sort an integer array by sequentially inserting each element into its correct position.

Code:

```c
#include <stdio.h>

void insertionSort(int arr[], int s){
    for(int i = 1; i < s; i++){
        int key = arr[i];
        int j = i - 1;

        while(j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

int main(){
    int arr[] = {5, 2, 6, 9, 1, 3, 17, 7};
    int s = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    insertionSort(arr, s);

    printf("\nSorted array (Insertion Sort): ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output:

```
Original array: 5 2 6 9 1 3 17 7
Sorted array (Insertion Sort): 1 2 3 5 6 7 9 17
c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The element is shifted until it reaches its sorted position in the left part of the array.

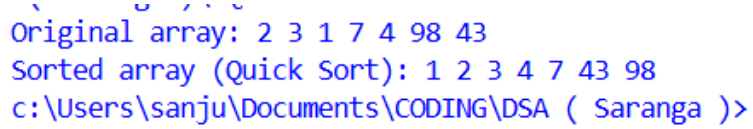
Exercise 14 Date: 31-10-2025

Aim:
To write a C program to implement the Quick Sort algorithm using the partition method and display the sorted array.

Code:
```c
#include <stdio.h>

void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high){
    int pivot = arr[low];
    int i = low;

    for(int j = low + 1; j <= high; j++){
        if(arr[j] < pivot){
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i], &arr[low]);
    return i;
}

void quickSort(int arr[], int low, int high){
    if(low < high){
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```c
int main(){
    int arr[] = {2, 3, 1, 7, 4, 98, 43};
    int s = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    quickSort(arr, 0, s - 1);

    printf("\nSorted array (Quick Sort): ");
    for(int i = 0; i < s; i++){
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output:



Observation:

The program uses the last element of the array as the pivot during partitioning. The QuickSort function applies recursion to sort the subarrays divided by the pivot index.

Exercise 15

Date: 07-11-2025

Aim:

To write a C program that creates the adjacency matrix of the graph shown in Figure 2 and displays the matrix after adding all edges as per the structure of the graph.

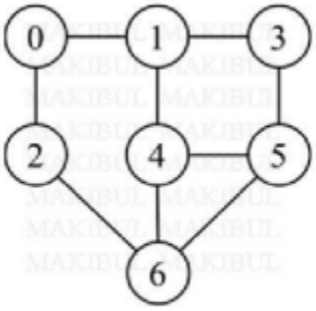


Figure 2

Code:

```c
#include <stdio.h>
#define MAX 10

int adjMatrix[MAX][MAX];

void initGraph(int vertices) {
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            adjMatrix[i][j] = 0;
        }
    }
}

void addEdge(int src, int dest) {
    adjMatrix[src][dest] = 1;
    adjMatrix[dest][src] = 1;
}

void displayGraph(int vertices) {
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            printf("%d ", adjMatrix[i][j]);
        }
        printf("\n");
    }
}


int main() {
    int vertices = 7;
    initGraph(vertices);

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(1, 4);
    addEdge(2, 6);
    addEdge(3, 5);
    addEdge(4, 5);
    addEdge(4, 6);
    addEdge(5, 6);

    displayGraph(vertices);

    return 0;
}
```

Output:

```
0 1 1 0 0 0 0
1 0 0 1 1 0 0
1 0 0 0 0 0 1
0 1 0 0 0 1 0
0 1 0 0 0 1 1
0 0 0 1 1 0 1
0 0 1 0 1 1 0

c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The adjacency matrix represents the undirected graph in Figure 2, with each edge being reflected symmetrically in the matrix.

Exercise 16

Date: 07-11-2025

Aim:

To write a C program that creates the adjacency list of the graph shown in Figure 3 and prints the list representation after adding all edges as per the figure.

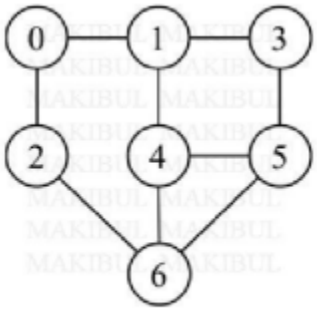


Figure 3

Code:

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

struct Gnode {
    int vertex;
    struct Gnode *next;
};

struct Gnode *adjList[10];

void initGraph(int vertices) {
    for (int i = 0; i < vertices; i++) {
        adjList[i] = NULL;
    }
}

struct Gnode* createNode(int vert) {
    struct Gnode *new_node = (struct Gnode*)malloc(sizeof(struct Gnode));
    new_node->vertex = vert;
    new_node->next = NULL;
    return new_node;
}

void addEdge(int src, int dest) {
    struct Gnode *new_node = createNode(dest);
    new_node->next = adjList[src];
    adjList[src] = new_node;

    struct Gnode *new_node2 = createNode(src);
    new_node2->next = adjList[dest];
    adjList[dest] = new_node2;
}
```

```c
void displayGraph(int vertices) {
    for (int i = 0; i < vertices; i++) {
        struct Gnode *temp = adjList[i];
        printf("Vertex %d: ", i);
        while (temp != NULL) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
    printf("\n");
}

int main() {
    int vertices = 7;
    initGraph(vertices);

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(1, 4);
    addEdge(2, 6);
    addEdge(3, 5);
    addEdge(4, 5);
    addEdge(4, 6);
    addEdge(5, 6);

    displayGraph(vertices);

    return 0;
}
```

Output:

```
Vertex 0: 2 -> 1 -> NULL
Vertex 1: 4 -> 3 -> 0 -> NULL
Vertex 2: 6 -> 0 -> NULL
Vertex 3: 5 -> 1 -> NULL
Vertex 4: 6 -> 5 -> 1 -> NULL
Vertex 5: 6 -> 4 -> 3 -> NULL
Vertex 6: 5 -> 4 -> 2 -> NULL


c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

Each undirected edge is stored twice: once for each direction, reflecting the structure of Figure 3.

Exercise 17 Date: 14-11-2025

Aim:

To write a C program to implement the Heap Sort using the **heapify** function. The program should sort an array in ascending order.

Code:

```c
#include <stdio.h>

void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        heapify(arr, n, largest);
    }
}
```

```
void heapSort(int arr[], int n)
{
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--)
    {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr, i, 0);
    }
}

int main()
{
    int arr[] = {9, 4, 3, 8, 10, 2, 5};
    int n = 7;

    heapSort(arr, n);

    printf("Sorted heap: ");
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
Sorted heap: 2 3 4 5 8 9 10
c:\Users\sanju\Documents\CODING\DSA ( Saranga )>
```

Observation:

The program builds a max heap from the given array and repeatedly applies heapify, resulting in the array being sorted in ascending order.