MODULE -1

LAB EXERCISE PROGRAMS – INSERTION SORT, STRONG PASSWORD

1. **INSERTION SORT**

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>
/* Function to sort an array
using insertion sort*/
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++)
	{
		key = arr[i];
		j = i - 1;

		/* Move elements of arr[0..i-1],
		that are greater than key,
		to one position ahead of
		their current position */
		while (j >= 0 && arr[j] > key)
		{
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
// A utility function to print
// an array of size n
void printArray(int arr[], int n)
{
	int i;
	for (i = 0; i < n; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

// Driver code
int main()
{
	int arr[] = {12, 11, 13, 5, 6};
	int n = sizeof(arr) / sizeof(arr[0]);
```

```
    insertionSort(arr, n);
    printArray(arr, n);
    return 0;
}
```

OUTPUT : 5 6 11 12 13

2. **STRONG PASSWORD**

```
#include <stdio.h>
#include <string.h>
// Function to determine the minimum number of characters to add
int minimumNumber(int n, char *password) {
    int required_chars = 0;
    int has_digit = 0, has_lower = 0, has_upper = 0, has_special = 0;
    const char *special_characters = "!@#$%^&*()-+";
    // Check the existing characters in the password
    for (int i = 0; i < n; i++) {
        if (password[i] >= '0' && password[i] <= '9') has_digit = 1;
        else if (password[i] >= 'a' && password[i] <= 'z') has_lower = 1;
        else if (password[i] >= 'A' && password[i] <= 'Z') has_upper = 1;
        else if (strchr(special_characters, password[i])) has_special = 1;
    }
    // Count the missing types of characters
    if (!has_digit) required_chars++;
    if (!has_lower) required_chars++;
    if (!has_upper) required_chars++;
    if (!has_special) required_chars++;
    // Ensure the password length is at least 6 characters
    if (n + required_chars < 6) {
        required_chars += (6 - (n + required_chars));
    }
    return required_chars;
}
int main() {
    int n;
    char password[101];
    // Input the length of the password and the password itself
    scanf("%d", &n);
    scanf("%s", password);
    // Calculate and print the minimum number of characters to add
    int result = minimumNumber(n, password);
    printf("%d\n", result);
    return 0;
}
```

INPUT: 3

Ab1

OUTPUT: Number of characters to make password strong: 3

5

aB1@

OUTPUT: Number of characters to make password strong: 1

ADDITIONAL PROGRAMS – THE POWER SUM, RUNNING TIME OF ALGORITHM

3. THE POWER SUM

```
#include <stdio.h>
#include <math.h>
// Function to recursively find the power sums
int powerSumHelper(int X, int N, int num) {
  int power = pow(num, N);
  if (power > X) {
    return 0;
  } else if (power == X) {
    return 1;
  } else {
    return powerSumHelper(X - power, N, num + 1) + powerSumHelper(X, N, num + 1);
  }
}
// Main function to find the number of ways to express X as sum of N-th powers of unique natural numbers
int powerSum(int X, int N) {
  return powerSumHelper(X, N, 1);
}
int main() {
  int X, N;
  printf("Enter X: ");// Input the values of X and N
  scanf("%d", &X);
  printf("Enter N: ");
  scanf("%d", &N);
```

```
    int result = powerSum(X, N);    // Calculate and print the number of combinations
    printf("%d\n", result);
    return 0;
}
```

Input:
Enter X: 100
Enter N: 3

Output: 1

Explanation:

The only way to express 100 as the sum of cubes of unique natural numbers is: 100=43100 = 4^3100=43

4.RUNNING TIME OF ALGORITHM

```
#include <stdio.h>
// Function to perform Insertion Sort and count the number of shifts
int runningTime(int arr[], int n) {
    int shifts = 0;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Move elements of arr[0..i-1] that are greater than key
     //to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
            shifts++;
        }
        arr[j + 1] = key;
    }
    return shifts;
}
```

```
int main() {
   int n;
   // Read the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter the elements: "); // Read the array elements
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   int result = runningTime(arr, n); // Get the number of shifts and print it
   printf("Number of shifts: %d\n", result);
   return 0;
}
```

Input : Enter the number of elements: 5
Enter the elements: 2 1 3 1 2
Output : Number of shifts: 4

MODULE -2

LAB EXERCISE PROGRAMS – Sorting: Comparator , Pattern-syntax-checker

1. **Sorting Comparator**

```
import java.util.*;

class Player {
   String name;
   int score;

   Player(String name, int score) {
      this.name = name;
      this.score = score;
   }
}
class Checker implements Comparator<Player> {
   // complete this method
      public int compare(Player p1, Player p2) {
      return p1.score != p2.score ? (p2.score - p1.score) : p1.name.compareTo(p2.name);
```

```
    }
}
public class Solution {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();

        Player[] player = new Player[n];
        Checker checker = new Checker();

        for(int i = 0; i < n; i++){
            player[i] = new Player(scan.next(), scan.nextInt());
        }
        scan.close();

        Arrays.sort(player, checker);
        for(int i = 0; i < player.length; i++){
            System.out.printf("%s %s\n", player[i].name, player[i].score);
        }
    }
}
```

Sample Input

```
5
amy 100
david 100
heraldo 50
aakansha 75
aleksa 150
```

Sample Output

```
aleksa 150
amy 100
david 100
aakansha 75
heraldo 50
```

2. **Pattern-syntax-checker**

```
import java.util.Scanner;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
public class PatternSyntaxChecker {
```

```
        public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter the number of patterns to check:");
            int testCases = Integer.parseInt(scanner.nextLine());

            while (testCases > 0) {
                String pattern = scanner.nextLine();
                try {
                    Pattern.compile(pattern);
                    System.out.println("Valid");
                } catch (PatternSyntaxException e) {
                    System.out.println("Invalid");
                }
                testCases--;
            }
            scanner.close();
        }
    }
```

Sample Input

```
3
([A-Z])(.+)
[AZ[a-z](a-z)
batcatpat(nat
```

Sample Output

```
Valid
Invalid
Invalid
```

Additional Programs – JAVA SHA-256 , Java Regex 2 - Duplicate Words

3.Java SHA-256

```
import java.io.*;
import java.util.*;
import java.security.*;
public class Solution {
    public static void main(String[] args) {
        /* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class sho
uld be named Solution. */
        Scanner scanner = new Scanner(System.in);
        String key = scanner.next();
        try{
        MessageDigest md = MessageDigest.getInstance("SHA-256");
            md.update(key.getBytes());
```

```
            byte[] digest = md.digest();
            StringBuffer stringbuffer = new StringBuffer();
        for (byte b: digest)
        {  // needed to print it in hexadecimal format
           stringbuffer.append(String.format("%02x", b));
        }
            System.out.println(stringbuffer.toString());
            }
         catch (NoSuchAlgorithmException exception)
        {
           System.out.println(exception);
        }

    }
}
```

Sample Input

HelloWorld

Sample Output

872e4e50ce9990d8b041330c47c9ddd11bec6b503ae9386a99da8584e9bb12c4

4. Java Regex 2 - Duplicate Words

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class DuplicateWords {
   public static void main(String[] args) {
      String regex = "\\b(\\w+)(?:\\W+\\1\\b)+";
      Pattern p = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
       Scanner in = new Scanner(System.in);
      int numSentences = Integer.parseInt(in.nextLine());
       while (numSentences-- > 0) {
         String input = in.nextLine();
                Matcher m = p.matcher(input);
          // Check for subsequences of input that match the compiled pattern
         while (m.find()) {
            input = input.replaceAll(m.group(), m.group(1));
         }
         // Prints the modified sentence.
         System.out.println(input);
      }
            in.close();
   }
}
```

Sample Input

5
Goodbye bye bye world world world
Sam went went to to to his business
Reya is is the the best player in eye eye game
in inthe
Hello hello Ab aB

Sample Output

Goodbye bye world
Sam went to his business
Reya is the best player in eye game
in inthe
Hello Ab

MODULE -3

LAB EXERCISE PROGRAMS - Insert a node at a specific position in a linked list, Merge two sorted linked lists

1. **Insert a node at a specific position in a linked list**

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct slinklist {
   int data;
   struct slinklist *next;
};
typedef struct slinklist node;
node *start = NULL;
int menu() {
   int ch;
   printf("\n 1.Create a list ");
   printf("\n-------------------------");
   printf("\n 2.Insert a node at specified position");
   printf("\n-------------------------");
   printf("\n 3.Display");
   printf("\n-------------------------");
   printf("\n 4. Exit ");
   printf("\n\n Enter your choice: ");
   scanf("%d", &ch);
   return ch;
}
node* getnode() {
```

```
    node *newnode;
    newnode = (node *)malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;
    return newnode;
}
void createlist(int n) {
    int i;
    node *newnode;
    node *temp;
    for (i = 0; i < n; i++) {
        newnode = getnode();
        if (start == NULL) {
            start = newnode;
        } else {
            temp = start;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = newnode;
        }
    }
}

int countnode(node *ptr) {
    int count = 0;
    while (ptr != NULL) {
        count++;
        ptr = ptr->next;
    }
    return count;
}

void display() {
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): \n");
    if (start == NULL) {
        printf("\n Empty List");
        return;
    } else {
        while (temp != NULL) {
            printf("%d-->", temp->data);
            temp = temp->next;
        }
    }
    printf(" X ");
```

```
}

void insert_at_pos() {
   node *newnode, *temp, *prev;
   int pos, nodectr, ctr = 1;
   newnode = getnode();
   printf("\n Enter the position: ");
   scanf("%d", &pos);
   nodectr = countnode(start);

   if (pos < 1 || pos > nodectr + 1) {
      printf("Position %d is invalid\n", pos);
      free(newnode);
      return;
   }

   if (pos == 1) { // Insert at the beginning
      newnode->next = start;
      start = newnode;
   } else {
      temp = start;
      while (ctr < pos - 1) {
         temp = temp->next;
         ctr++;
      }
      newnode->next = temp->next;
      temp->next = newnode;
   }
}
void main(void) {
   int ch, n;
   while (1) {
      ch = menu();
      switch (ch) {
         case 1:
            if (start == NULL) {
               printf("\n Number of nodes you want to create: ");
               scanf("%d", &n);
               createlist(n);
               printf("\n List created..");
            } else
               printf("\n List is already created..");
            break;
         case 2:
            insert_at_pos();
            break;
         case 3:
```

```
            display();
            break;
         default:
            exit(0);
      }
   }
}
```

2. Merge two sorted linked lists

Create a method to merge two singly linked lists into one sorted linked list. The merged list should maintain the sorted order of elements.

To perform the above operation, both the list should be traversed, comparing the elements at each step, and adding the smaller element to the merged list. If one of the lists is exhausted before the other, we simply append the remaining elements of the other list to the merged list.

Program Code:

```
#include <stdio.h>
#include <stdlib.h>
struct slinklist {
   int data;
   struct slinklist *next;
};
typedef struct slinklist node;
node *start1 = NULL;
node *start2 = NULL;
int menu() {
   int ch;
   printf("\n 1. Create lists");
   printf("\n-------------------------");
   printf("\n 2. Insert a node into list 1");
   printf("\n-------------------------");
   printf("\n 3. Insert a node into list 2");
   printf("\n-------------------------");
   printf("\n 4. Merge two lists");
   printf("\n-------------------------");
   printf("\n 5. Exit ");
```

```
   printf("\n\n Enter your choice: ");
   scanf("%d", &ch);
   return ch;
}
node* getnode() {
   node *newnode;
   newnode = (node *)malloc(sizeof(node));
   printf("\n Enter data: ");
   scanf("%d", &newnode->data);
   newnode->next = NULL;
   return newnode;
}
void createlist(node **start, int n) {
   int i;
   node *newnode;
   node *temp;
   for (i = 0; i < n; i++) {
      newnode = getnode();
      if (*start == NULL) {
         *start = newnode;
      } else {
         temp = *start;
         while (temp->next != NULL)
            temp = temp->next;
         temp->next = newnode;
      }
   }
}
void display(node *start) {
   node *temp;
   temp = start;
   printf("\n The contents of the list (Left to Right): \n");
   if (start == NULL) {
      printf("\n Empty List");
      return;
   } else {
      while (temp != NULL) {
         printf("%d-->", temp->data);
         temp = temp->next;
      }
   }
   printf(" X ");
}
void insert_node(node **start) {
   node *newnode = getnode();
   if (*start == NULL) {
      *start = newnode;
```

```
    } else {
        node *temp = *start;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newnode;
    }
    printf("\n Node inserted successfully.\n");
    display(*start);
}
node* merge_lists(node *list1, node *list2) {
    node dummy;
    dummy.next = NULL;
    node *tail = &dummy;
    while (list1 != NULL && list2 != NULL) {
        if (list1->data <= list2->data) {
            tail->next = list1;
            list1 = list1->next;
        } else {
            tail->next = list2;
            list2 = list2->next;
        }
        tail = tail->next;
    }
    if (list1 != NULL) {
        tail->next = list1;
    } else {
        tail->next = list2;
    }
    return dummy.next;
}
void main(void) {
    int ch, n;
    while (1) {
        ch = menu();
        switch (ch) {
            case 1:
                if (start1 == NULL && start2 == NULL) {
                    printf("\n Number of nodes you want to create in list 1: ");
                    scanf("%d", &n);
                    createlist(&start1, n);
                    printf("\n List 1 created..");
                    display(start1);
                    printf("\n\n Number of nodes you want to create in list 2: ");
                    scanf("%d", &n);
                    createlist(&start2, n);
                    printf("\n List 2 created..");
                    display(start2);
```

```
            } else {
               printf("\n Lists are already created..");
            }
            break;
         case 2:
            insert_node(&start1);
            break;
         case 3:
            insert_node(&start2);
            break;
         case 4:
            if (start1 == NULL || start2 == NULL) {
               printf("\n One or both lists are empty, cannot merge\n");
            } else {
               node *merged_list = merge_lists(start1, start2);
               printf("\n Merged List: ");
               display(merged_list);
            }
            break;
         default:
            exit(0);
      }
   }
}
```

ADDITIONAL PROGRAMS - Delete the first node from the linked list , Print the linked list in reverse order

3. Delete the first node from the linked list

```
#include <stdio.h>
#include <stdlib.h>
/* Structure of a node */
struct node {
   int data;        // Data
   struct node *next; // Address
}*head;
void createList(int n);
void deleteFirstNode();
void displayList();
int main()
{
   int n, choice;

   /*
    * Create a singly linked list of n nodes
    */
   printf("Enter the total number of nodes: ");
   scanf("%d", &n);
```

```
   createList(n);
   printf("\nData in the list \n");
   displayList();
   printf("\nPress 1 to delete first node: ");
   scanf("%d", &choice);
   /* Delete first node from list */
   if(choice == 1)
      deleteFirstNode();

   printf("\nData in the list \n");
   displayList();
   return 0;
}

/*
 * Create a list of n nodes
 */
void createList(int n)
{
   struct node *newNode, *temp;
   int data, i;
   head = (struct node *)malloc(sizeof(struct node));
   /*
    * If unable to allocate memory for head node
    */
   if(head == NULL)
   {
      printf("Unable to allocate memory.");
   }
   else
   {
      /*
       * In data of node from the user
       */
      printf("Enter the data of node 1: ");
      scanf("%d", &data);
      head->data = data; // Link the data field with data
      head->next = NULL; // Link the address field to NULL
      temp = head;
      /*
       * Create n nodes and adds to linked list
       */
      for(i=2; i<=n; i++)
      {
         newNode = (struct node *)malloc(sizeof(struct node));

         /* If memory is not allocated for newNode */
```

```
            if(newNode == NULL)
            {
               printf("Unable to allocate memory.");
               break;
            }
            else
            {
               printf("Enter the data of node %d: ", i);
               scanf("%d", &data);
               newNode->data = data; // Link the data field of newNode with data
               newNode->next = NULL; // Link the address field of newNode with NULL
               temp->next = newNode; // Link previous node i.e. temp to the newNode
               temp = temp->next;
            }
         }
         printf("SINGLY LINKED LIST CREATED SUCCESSFULLY\n");
      }
}

/*
 * Deletes the first node of the linked list
 */
void deleteFirstNode()
{
   struct node *toDelete;

   if(head == NULL)
   {
      printf("List is already empty.");
   }
   else
   {
      toDelete = head;
      head = head->next;
      printf("\nData deleted = %d\n", toDelete->data);
      /* Clears the memory occupied by first node*/
      free(toDelete);

      printf("SUCCESSFULLY DELETED FIRST NODE FROM LIST\n");
   }
}
/*
 * Displays the entire list
 */
void displayList()
{
   struct node *temp;
```

```
  /*
   * If the list is empty i.e. head = NULL
   */
  if(head == NULL)
  {
     printf("List is empty.");
  }
  else
  {
     temp = head;
     while(temp != NULL)
     {
        printf("Data = %d\n", temp->data); // Print data of current node
        temp = temp->next;              // Move to next node
     }
  }
}
```

4. Print the linked list in reverse order

Given a pointer to the head of a singly-linked list, print each data value from the reversed list. If the given list is empty, do not print anything.

Function Description

Complete the *reversePrint* function in the editor below.

reversePrint has the following parameters:

- *SinglyLinkedListNode pointer head:* a reference to the head of the list

Prints

The data values of each node in the reversed list.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* next;
};

typedef struct Node Node;

// Function to create a new node
Node* create_node(int data) {
  Node* new_node = (Node*)malloc(sizeof(Node));
  new_node->data = data;
  new_node->next = NULL;
```

```
    return new_node;
}

// Function to insert a node at the end of the list
void insert_node(Node** head, int data) {
    Node* new_node = create_node(data);
    if (*head == NULL) {
        *head = new_node;
    } else {
        Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}
// Function to print the list
void print_list(Node* head) {
    Node* temp = head;
    printf("The linked list is: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Recursive function to print the list in reverse order
void print_reverse(Node* head) {
    if (head == NULL) {
        return;
    }
    print_reverse(head->next);
    printf("%d ", head->data);
}

int main() {
   Node* head = NULL;
    int n, data;

    // Reading the number of nodes
    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    // Reading the data for each node and inserting into the list
    for (int i = 0; i < n; i++) {
        printf("Enter data for node %d: ", i + 1);
```

```
        scanf("%d", &data);
        insert_node(&head, data);
    }
    // Printing the list before reversing
    print_list(head);

    // Printing the list in reverse order
    printf("The linked list in reverse order is: ");
    print_reverse(head);
    printf("\n");

    // Freeing the allocated memory
    Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
    return 0;
}
```

Output:

```
Enter the number of nodes: 4
Enter data for node 1: 34
Enter data for node 2: 12
Enter data for node 3: 67
Enter data for node 4: 22
The linked list is: 34 12 67 22
The linked list in reverse order is: 22 67 12 34
```

MODULE - 4

LAB EXERCISE PROGRAMS – POISONOUS PLANT, TRUCK TOUR

1.POISONOUS PLANT

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a stack element
typedef struct {
    int pesticide;
    int days;
} Plant;
// Function to find the number of days until no plants die
int poisonousPlants(int n, int* p) {
    Plant* stack = (Plant*)malloc(n * sizeof(Plant));
    int top = -1, max_days = 0;
```

```
    for (int i = 0; i < n; i++) {
        int days = 0;
        while (top >= 0 && stack[top].pesticide >= p[i])
     {
            days = days > stack[top].days ? days : stack[top].days;
            top--;
        }
        if (top >= 0) {
            days++;
        } else {
            days = 0;
        }
        max_days = days > max_days ? days : max_days;
        stack[++top] = (Plant){p[i], days};
    }

    free(stack);
    return max_days;
}

int main() {
    int n;
    scanf("%d", &n);
    int* p = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", &p[i]);
    }
    int result = poisonousPlants(n, p);
    printf("%d\n", result);
    free(p);
    return 0;
}
```

2. Truck Tour

```
#include <stdio.h>

int main() {

    int n;

    scanf("%d", &n);

    int petrol[n], distance[n];

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

        scanf("%d %d", &petrol[i], &distance[i]);
```

```
    }
     int start = 0;
    int deficit = 0;
    int capacity = 0;
        for (int i = 0; i < n; i++) {
        capacity += petrol[i] - distance[i];
        if (capacity < 0) {
            start = i + 1;
            deficit += capacity;
            capacity = 0;
        }
    }
        if (capacity + deficit >= 0) {
        printf("%d\n", start);
    } else {
        printf("-1\n");
    }


    return 0;
}
```

ADDITIONAL PROGRAMS – Queue using 2 Stacks, Largest Rectangle

3. Largest Rectangle

```
      #include <stdio.h>
     #include <stdlib.h>
// Function to find the maximum area of the histogram
     long largestRectangle(int* h, int n) {
     int* stack = (int*)malloc(n * sizeof(int));
      int top = -1;
      long max_area = 0;
      long area_with_top;
         int i = 0;
     while (i < n) {
         if (top == -1 || h[stack[top]] <= h[i]) {
```

```
            stack[++top] = i++;
        } else {
            int tp = stack[top--];
            area_with_top = h[tp] * (top == -1 ? i : i - stack[top] - 1);
            if (max_area < area_with_top) {
              max_area = area_with_top;
            }
        }
    }
    while (top != -1) {
          int tp = stack[top--];
        area_with_top = h[tp] * (top == -1 ? i : i - stack[top] - 1);
        if (max_area < area_with_top) {
            max_area = area_with_top;
        }
    }
    free(stack);
    return max_area;
}
int main() {
    int n;
        scanf("%d", &n);
        int* h = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            scanf("%d", &h[i]);
        }
        printf("%ld\n", largestRectangle(h, n));
        free(h);
        return 0;
    }
```

Input : 5
2 1 5 6 2 3
Output : 10

4. Queue using two stacks

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Stack {
   int top;
   unsigned capacity;
   int* array;
} Stack;
// Function to create a stack of given capacity
Stack* createStack(unsigned capacity) {
   Stack* stack = (Stack*)malloc(sizeof(Stack));
```

```
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
// Stack is full when top is equal to the last index
int isFull(Stack* stack) {
    return stack->top == stack->capacity - 1;
}
// Stack is empty when top is -1
int isEmpty(Stack* stack) {
    return stack->top == -1;
}

// Function to add an item to stack. It increases top by 1
void push(Stack* stack, int item) {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}
// Function to remove an item from stack. It decreases top by 1
int pop(Stack* stack) {
    if (isEmpty(stack))
        return -1;
    return stack->array[stack->top--];
}
// Function to implement queue using two stacks
typedef struct Queue {
    Stack* stack1;
    Stack* stack2;
} Queue;
// Function to create a queue
Queue* createQueue(unsigned capacity) {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->stack1 = createStack(capacity);
    queue->stack2 = createStack(capacity);
    return queue;
}
// Function to enqueue an item to the queue
void enqueue(Queue* queue, int item) {
    push(queue->stack1, item);
}
// Function to dequeue an item from the queue
int dequeue(Queue* queue) {
```

```
    if (isEmpty(queue->stack2)) {
        while (!isEmpty(queue->stack1)) {
            push(queue->stack2, pop(queue->stack1));
        }
    }
    return pop(queue->stack2);
}
// Function to display the queue
void displayQueue(Queue* queue) {
    if (isEmpty(queue->stack1) && isEmpty(queue->stack2)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    for (int i = 0; i <= queue->stack2->top; i++) {
        printf("%d ", queue->stack2->array[i]);
    }
    for (int i = queue->stack1->top; i >= 0; i--) {
        printf("%d ", queue->stack1->array[i]);
    }
    printf("\n");
}
int main() {
    Queue* queue = createQueue(100);
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    printf("Dequeued item is %d\n", dequeue(queue));

    displayQueue(queue);
    return 0;
}
```

OUTPUT : Dequeued item is 10
Queue: 20 30

MODULE -5

LAB EXERCISE PROGRAMS - Lowest Common Ancestor in Binary Tree, Height of a Binary Tree

1. **Lowest Common Ancestor in Binary Tree**

Algorithm :

- Create a recursive function that takes a node and the two values n1 and n2.

- If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
- If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
- If both the above cases are false, then return the current node as LCA.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data;
 struct node *left, *right;
};
struct node *lca (struct node *root, int n1, int n2)
{
 while (root != NULL)
  {
   if (root->data > n1 && root->data > n2)
        root = root->left;

   else if (root->data < n1 && root->data < n2)
        root = root->right;

   else
        break;
  }
 return root;
}
struct node *newNode (int data)
{
 struct node *node = (struct node *) malloc (sizeof (struct node));
 node->data = data;
 node->left = node->right = NULL;
 return (node);
}
int main ()
{
 struct node *root = newNode (20);
 root->left = newNode (8);
 root->right = newNode (22);
 root->left->left = newNode (4);
 root->left->right = newNode (12);
 root->left->right->left = newNode (10);
 root->left->right->right = newNode (14);
 int n1 = 10, n2 = 14;
 struct node *t = lca (root, n1, n2);
 printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
```

```
 n1 = 14, n2 = 8;
 t = lca (root, n1, n2);
 printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
 n1 = 10, n2 = 22;
 t = lca (root, n1, n2);
 printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
 getchar ();
 return 0;
}
```

OUTPUT : LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

2. **Height of a Binary Tree**

```
#include <stdio.h>

#include <stdlib.h>

struct node

{

 int data;

 struct node *left;

 struct node *right;

};

int height (struct node *node)

{

 if (node == NULL)

  return 0;

 else

  {

   int leftHeight = height (node->left);

   int rightHeight = height (node->right);

   if (leftHeight > rightHeight)

    return (leftHeight + 1);
```

```
    else
      return (rightHeight + 1);
   }
}
struct node *newNode (int data)
{
  struct node *node = (struct node *) malloc (sizeof (struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return (node);
}
int main ()
{
  struct node *root = newNode (10);
  root->left = newNode (20);
  root->right = newNode (30);
  root->left->left = newNode (40);
  root->left->right = newNode (50);
  printf ("Height of tree is %d", height (root));
  return 0;
}
```

OUTPUT : Height of tree is 3

ADDITIONAL PROGRAMS - . To check if a Tree is Binary Search Tree **,** To perform insertion in Binary Search Tree

3. Program to Check if a Tree is Binary Search Tree

```
#include <stdio.h>
#include <stdlib.h>
 struct node
{
   int data;
```

```
    struct node* left;
    struct node* right;
};
 static struct node *prev = NULL;
 /*Function to check whether the tree is BST or not*/
int is_bst(struct node* root)
{
    if (root)
    {
       if (!is_bst(root->left)) //moves towards the leftmost child of the tree and checks for the BST
          return 0;
       if (prev != NULL && root->data <= prev->data)
          return 0;
       prev = root;
       return is_bst(root->right);    //moves the corresponding right child of the tree and checks for the BST
    }
    return 1;
}
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
     return(node);
}
 int main()
{
  /*
    The input tree is as shown below
```

```
          40
          / \
        20      60
        / \     \
      10      30     80
                \
                 90

 */
  struct node *root = newNode(40);
  root->left       = newNode(20);
  root->right      = newNode(60);
  root->left->left  = newNode(10);
  root->left->right = newNode(30);
  root->right->right = newNode(80);
  root->right->right->right = newNode(90);
  if (is_bst(root))
    printf("TREE 1 Is BST");
  else
    printf("TREE 1 Not a BST");
  prev = NULL;
/*
  The input tree is as shown below
          50
          / \
        20      30
        / \

      70      80

      / \       \
```

```
   10    40    60
*/
   struct node *root1 = newNode(50);
   root1->left  = newNode(20);
   root1->right  = newNode(30);
   root1->left->left  = newNode(70);
   root1->left->right = newNode(80);
   root1->left->left->right = newNode(40);
   root1->left->left->left= newNode(90);
   if (is_bst(root1))
      printf("TREE 2 Is BST");
   else
      printf("TREE 2 Not a BST");
   return 0;
}
```

OUTPUT : TREE 1 Is BSTTREE 2 Not a BST

4. Program to perform insertion in Binary Search Tree

```
#include<stdio.h>
#include<stdlib.h>
// Basic struct of Tree
struct node
{
   int val;
   struct node *left, *right;
};
  // Function to create a new Node
struct node* newNode(int item)
{
   struct node* temp =  (struct node *)malloc(sizeof(struct node));
   temp->val = item;
   temp->left = temp->right = NULL;
```

```
    return temp;
}
    // Function print the node in inorder format, when insertion is complete
void inorder(struct node* root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->val);
        inorder(root->right);
    }
}
    // Here we are finding where to insert the new node so BST is followed
struct node* insert(struct node* node, int val)
{
    /* If the tree(subtree) is empty, return a new node by calling newNode func.*/
    if (node == NULL) return newNode(val);

    /* Else, we will do recursion down the tree to further subtrees */
    if (val < node->val)
        node->left  = insert(node->left, val);
    else if (val > node->val)
        node->right = insert(node->right, val);
     /* (Safety) return the node's pointer which is unchanged */
    return node;
}
  int main()
{
    /* Our BST will look like this
           100
         /    \
        40     140
       / \    / \
      40  80 120  160 */
```

```
  struct node* root = NULL;
  root = insert(root, 100);
  insert(root, 60);
  insert(root, 40);
  insert(root, 80);
  insert(root, 140);
  insert(root, 120);
  insert(root, 160);
      // Finally printing the tree using inorder
  inorder(root);
    return 0;
}
```

OUTPUT : **40**

60

80

100

120

140

160