

# Lab-1:

## Program1:

### Algorithm:

1. Start
2. Take array size as input.
3. Take the array elements as input from the user.
4. Calculate the max of the array using linear search.
5. Stop

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

```
int MAX;
```

```
void findMax(int size, int *arr){
```

```
    static int count = 0;
```

```
    ++ count;//counting num of times func is called
```

```
    if ( *(arr) > MAX){
```

```
        MAX = *(arr);
```

```
    }
```

```
    if (count == size ){//checking if we've reached the last element of array
```

```
        return;
```

```
    }
```

```
    findMax(size, (++arr));
```

```
}
```

```
int main(){
```

```
    int size;
```

```
    scanf("%d", &size);//taking the array size
```

```
    int arr[size];//declaring array
```

```
    for (int i = 0; i < size; i++){
```

```
        scanf("%d", &arr[i]);//reading the array elements
```

```
    }
```

```
    MAX = arr[0];
```

```
    findMax(size, arr);
```

```
    printf("Max: %d", MAX);
```

```
    return 0;
```

```
}
```

```
10      int size;
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS$ ./a.out
10
1
3
5
7
9
2
4
6
8
10
Max: 10Portkey@taruni:~/Desktop/DS$
```

## Program2:

### Algorithm:

1. Start.
2. Take the two arrays as input from the user.
3. Check if the two arrays can be multiplied.
4. If yes, multiply the two arrays (matrices) and print out the resultant matrix.
5. If not, print that multiplication is not possible.
6. Stop.

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

```
void display(int arr[100][100], int row, int col){
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

```
void multiply(int a[100][100], int r1, int c1, int b[100][100], int r2, int c2, int c[100][100]){
    static int i = 0, j = 0, k = 0;
    if ( i == r1){
        return ;
    }
    if ( j < c2){
        if ( k < c1){
```

```

        c[i][j] += a[i][k] + b[k][j];
        k++;
        multiply(a, r1, c1, b, r2, c2, c);
    }
    k = 0;
    j++;
    multiply(a, r1, c1, b, r2, c2, c);
}
j = 0;
i++;
multiply(a, r1, c1, b, r2, c2, c);
}
int main(){
    int r1, c1, r2, c2;
    printf("Enter r1, c1, r2, c2:\n");
    scanf("%d%d%d%d", &r1, &c1, &r2, &c2);
    if ( c1 != r2){
        printf("Not possible");
    }
    else {
        int a[r1][c1], b[r2][c2];
        int c[100][100] = {0};
        printf("enter elements of first matrix\n");
        for (int i = 0; i < r1; i++){
            for (int j = 0; j < c1; j++){
                scanf("%d", &a[i][j]);
            }
        }
        printf("enter elements of second matrix\n");
        for (int i = 0; i < r2; i++){
            for (int j = 0; j < c2; j++){
                scanf("%d", &b[i][j]);
            }
        }
        printf("1");
        multiply(a, r1, c1, b, r2, c2, c);
        printf("2");
        display(c, r1, c2);
        printf("3");
    }
    return 0;
}

```

## Program3:

### Algorithm:

1. Start
2. Initialize hour, minute, seconds with 0.
3. Run an infinite loop.
4. Increase second and check if it is equal to 60 then increase minute and reset second to 0.
5. Increase minute and check if it is equal to 60 then increase hour and reset minute to 0.
6. Increase the hour and check if it is equal to 24 then reset hour to 0.
7. Stop

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

```
int MAXMIN = 60, MAXSEC = 60, MAXHOUR = 24;
```

```
void print(int n) {
    if (n < 10) {
        printf(" 0%d ", n);
    }
    else {
        printf(" %d ", n);
    }
}
```

```
int main(){
    int ss = 0, mm = 0, hh = 0;//hh:mm:ss format
    while (1){
        print(hh);
        printf(":");
        print(mm);
        printf(":");
        print(ss);
        printf("\n");

        ss ++;

        if ( ss == MAXSEC) {
            mm += 1;
            ss = 0;
        }
        if ( mm == MAXMIN) {
            hh += 1;
```

```

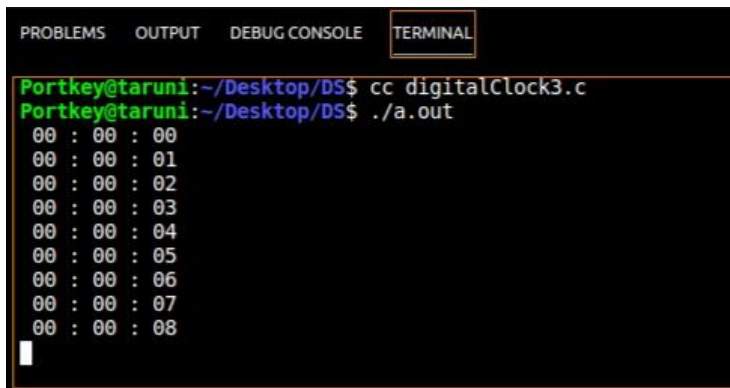
        mm = 0;
    }
    if ( hh == MAXHOUR ) {
        hh = 0;
        mm = 0;
        ss = 0;
    }

    sleep(1);

}

return 0;
}

```



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS$ cc digitalClock3.c
Portkey@taruni:~/Desktop/DS$ ./a.out
00 : 00 : 00
00 : 00 : 01
00 : 00 : 02
00 : 00 : 03
00 : 00 : 04
00 : 00 : 05
00 : 00 : 06
00 : 00 : 07
00 : 00 : 08

```

## Program4:

### Algorithm:

1. Start.
2. Take the student's name and marks from the user.
3. Store these details of each student in a struct.
4. Display the details of each student.
5. Stop

```

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

```

```

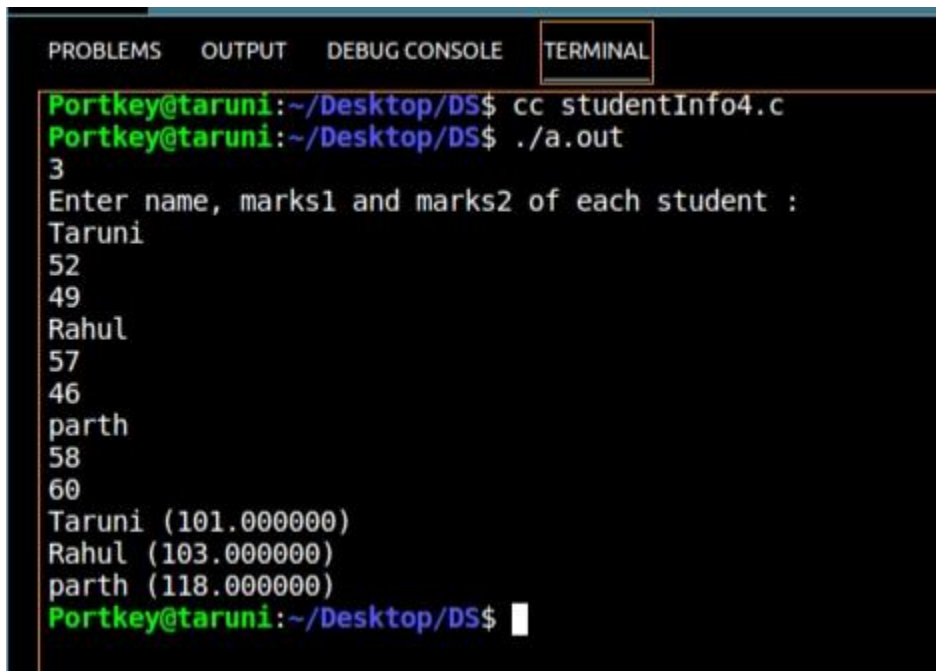
struct student{
    char name[30];
    float marks1, marks2;
};

```

```

int main(){
    struct student *arr;
    int n;
    scanf("%d", &n); //taking the number of students
    arr = (struct student *)malloc(n * sizeof(struct student)); //dynamic memory allocation
    printf("Enter name, marks1 and marks2 of each student :\n");
    for ( int i = 0; i < n; i ++){
        scanf("%s %f %f", (arr[i].name), &(arr[i].marks1), &(arr[i].marks2)); //reading info of each
student
    }
    for ( int i = 0; i < n; i ++){
        printf("%s (%f)\n", (arr[i].name), (arr[i].marks1) + (arr[i].marks2));
    }
}

```



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS$ cc studentInfo4.c
Portkey@taruni:~/Desktop/DS$ ./a.out
3
Enter name, marks1 and marks2 of each student :
Taruni
52
49
Rahul
57
46
parth
58
60
Taruni (101.000000)
Rahul (103.000000)
parth (118.000000)
Portkey@taruni:~/Desktop/DS$

```

## Program5:

### Algorithm:

1. Start.
2. Take the info of each student from the user and store them in a struct.
3. Sort these structs based on their names i.e, in an alphabetical order.
4. Display them after sorting.
5. Stop

```

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

struct student{
    char name[30];
    int rnum;
    float marks1, marks2;
};

int main(){
    struct student *arr;
    struct student temp;
    int n;
    scanf("%d", &n);
    arr = (struct student *)malloc(n * sizeof(struct student));
    printf("enter name, rnum marks1 marks2\n");
    for ( int i = 0; i < n; i ++){
        scanf("%s%d%f%f", (arr[i].name), &(arr[i].rnum), &(arr[i].marks1), &(arr[i].marks2));
    }

    for (int i = 0; i < n; i++){//sorting alphabetically
        int max = i;
        for (int j = 0; j < n; j++){
            if (arr[j].name > arr[max].name){
                max = j;
            }
        }
        temp = arr[max];
        arr[ max] = arr[i];
        arr[i] = temp;
    }
    for ( int i = 0; i < n; i ++){
        printf("%d. %s (marks-%f)\n", (i+1), (arr[i].name), ((arr[i].marks1) + (arr[i].marks2)));
    }
}

```

```

Portkey@taruni:~/Desktop/DS$ ./a.out
3
enter name, rnum marks1 marks2
ram
123
17
18
sam
124
20
16
pooja
125
20
17.5
1. pooja (marks-37.500000)
2. ram (marks-35.000000)
3. sam (marks-36.000000)
Portkey@taruni:~/Desktop/DS$ █

```

## Program6:

### Algorithm:

1. Start.
2. Take the line number and the text that has to be replaced with from the user as input.
3. Traverse through the file using a file pointer and copy the text into another file.
4. Once you arrive at the specified line, write the specified text into the new file.
5. Copy the remaining lines as it is.
6. Delete the old file and rename the new file.
7. Stop

```

#include <stdio.h>
int main(){
    FILE *fp, *fp2;
    char c, line[50];
    int lnum, count = 1;
    printf("Enter text:\n");
    fgets(line, 50, stdin);
    printf("Enter line num:\n");
    scanf("%d", &lnum);
    fp = fopen("file.txt", "r");
    fp2 = fopen("temp.txt", "w");
    while ( (c = getc(fp)) != EOF){
        if ( c == '\n'){
            ++ count;

```



```

    }
    if ( count == lnum){
        fprintf(fp2, "\n%s", line);//replacing line
        break;
    }
    else {
        putc(c, fp2);
    }
}
while ( (c = getc(fp)) != EOF){
    if ( c == '\n'){//removing original line
        break;
    }
}
while ((c = getc(fp)) != EOF) {
    putc(c, fp2);//copy the remaining lines as it is
}
fclose(fp);
remove("file.txt");

fclose(fp2);
rename("temp.txt", "file.txt");

return 0;
}

```

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Portkey@taruni:~/Desktop/DS$ cc file6.c
Portkey@taruni:~/Desktop/DS$ ./a.out
Enter text:
Replacing text
Enter line num:
2
Portkey@taruni:~/Desktop/DS$

```

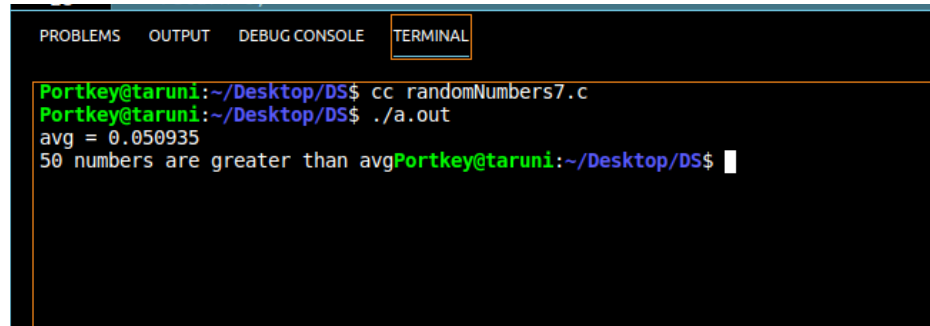
## Program7:

### Algorithm:

1. Start.
2. Open a file in write mode using file pointer.
3. Write random floating numbers using the rand() method into the file.
4. Close the file and open in read mode this time.
5. Read the numbers and calculate the average of them all.
6. Make the file pointer point to the starting of the file using rewind().
7. Now, count the number of numbers that are greater than the average,
8. Print the count.
9. Stop

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    float MAX = 0.5, MIN = -0.5;//initializing upper and lower limits respectively
    double sum = 0, number = 0, avg;
    srand(0);
    int key = 100, count = 0 ;
    FILE *fp;
    fp = fopen("file.txt", "w");
    for (int i = 0; i < key; i ++){
        float num = MIN + (float)(rand()) / ( (float) (RAND_MAX/(MAX - MIN))); //generating random
floating numbers
        fprintf(fp, "%f\n", num); //inserting into the file
    }
    fclose(fp);
    fp = fopen("file.txt", "r");
    while(!feof(fp)){
        fscanf(fp, "%lf", &number); //reading the nums from file
        sum += number; //adding those 100 numbers
    }
    avg = sum / key;
    rewind(fp);
    while(!feof(fp)) {
        fscanf(fp, "%lf", &number);
        if ( number > avg) ++ count ;
    }
```

```
}  
printf("avg = %lf", avg); //displaying the average  
printf("\n%d numbers are greater than avg", count);  
return 0;  
}
```



The image shows a terminal window with a dark background. At the top, there are four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is selected and highlighted with a yellow border. Below the tabs, the terminal displays the following text:   
Portkey@taruni:~/Desktop/DS\$ cc randomNumbers7.c  
Portkey@taruni:~/Desktop/DS\$ ./a.out  
avg = 0.050935  
50 numbers are greater than avgPortkey@taruni:~/Desktop/DS\$

# Lab-2:

## Program1:

### Algorithm:

1. Start.
2. We first create a stack by taking the size from the user.
3. The user can choose from one of the five options given: 1. Push, 2. Pop, 3. Peek, 4. Display and 5. Exit
  - 1.Push: when selected, we prompt the user for an element and push it into the stack if the stack isn't full.
  - 2.Pop: When selected, we print the top most element of the stack(i.e if the stack isn't empty) and delete it from the stack by moving the top pointer one step behind.
  - 3.Peek: When selected, we display the topmost element of stack.
  - 4.Display: When selected, we display the elements of the stack from 0th position to the top most position.
  - 5.Exit: When selected, we exit the program.
4. Stop

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

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

```
struct stack {
```

```
    int size;
```

```
    int top;
```

```
    int *a;
```

```
};
```

```
int isFull(struct stack *s) {
```

```
    return (s -> top == s -> size - 1) ? (1) : (0);
```

```
}
```

```
int isEmpty(struct stack *s) {
```

```
    return (s -> top == -1) ? (1) : (0);
```

```
}
```

```
struct stack * createStack() {
```

```
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
```

```
    printf("Enter the size of stack:\n");
```

```
    scanf("%d", & s -> size);
```

```
    s -> a = (int *)malloc (s -> size * sizeof(int));
```

```
    s -> top = -1;
```

```
    printf("Created a stack of size %d\n", s -> size);
```

```
    return s;
```

```
};
```

```
void push(struct stack *s) {
```

```

    if ( isFull(s) ) {
        printf("The stack is Full.\n");
        return ;
    }
    printf("Enter the element to be pushed: ");
    scanf("%d", &(s -> a[ ++ (s -> top)]));
    printf("Element has been pushed into the stack.\n");
    return ;
}

void pop(struct stack *s) {
    if ( isEmpty(s) ) {
        printf("The stack is Empty.\n");
        return;
    }
    printf("The popped element is: %d\n", s -> a[(s -> top) --]);
    return;
}

void peek(struct stack *s) {
    if ( isEmpty(s) ) {
        printf("The stack is Empty.\n");
        return;
    }
    printf("The top element is: %d\n", s -> a[(s -> top)]);
    return;
}

void display(struct stack *s) {
    if ( isEmpty(s) ) {
        printf("The stack is Empty.\n");
        return;
    }
    printf(" The elements of the stack are:\n");
    for (int i = 0; i <= s -> top; i ++ ) {
        printf("%d ", s -> a[i]);
    }
    return;
}

}

int main() {
    struct stack *a;
    int ch;
    a = createStack();
    while ( 1 ) {
        printf("\n1. Push an element\n2. Pop\n3. Peek\n4. Display\n5. Exit\nPlease enter your choice:\n");

```

```

scanf("%d", &ch);
switch (ch) {
    case 1 : push(a);
        break;
    case 2 : pop(a);
        break;
    case 3 : peek(a);
        break;
    case 4 : display(a);
        break;
    case 5 : printf("Exiting\n");
        break;
}
if ( ch == 5) {
    break;
}
}
return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/Lab2Stack$ cc menuDriven.c
Portkey@taruni:~/Desktop/DS/Lab2Stack$ ./a.out
Enter the size of stack:
3
Created a stack of size 3
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
2
The stack is Empty.
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter the element to be pushed: 20
Element has been pushed into the stack.
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:

```

```

Please enter your choice:
1
Enter the element to be pushed: 30
Element has been pushed into the stack.
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter the element to be pushed: 25
Element has been pushed into the stack.
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
1
The stack is Full.
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:

```

```
5. Exit
Please enter your choice:
1
The stack is Full.

1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
3
The top element is: 25

1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
4
The elements of the stack are:
20 30 25
1. Push an element
2. Pop
3. Peek
4. Display
5. Exit
Please enter your choice:
5
Exiting
Portkey@taruni:~/Desktop/DS/Lab2Stack$
```

## Program2:

### Algorithm:

1. Start.
2. First we take the expression ( a string) from the user.
3. We then create a stack of the size same as the size of the expression.
4. We traverse through the string character by character. When we encounter an opening brace/parenthesis, we push it into the stack.
5. If the character is a closing brace/parenthesis, we check if the topmost element of the stack matches with the current brace.
  - i) If it matches(i.e ')' and '(' or '}' and '{' or ']' and '['), we pop the stack.
  - ii) If not, it means the given expression is incorrect. We stop here.
6. If we reach the end of the string, and the stack is empty, it means the given expression is correct.
7. If not, it's incorrect.
8. Stop.

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

```
struct stack {
    int size;
    int top;
    char *a;
```

```

};
int isEmpty(struct stack *s) {
    return ( s -> top == -1) ? (1) : (0);
}
struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> size = size;
    s -> a = (char *)malloc (size * sizeof(char));
    s -> top = -1;
    return s;
};
void push(struct stack *s, char ch) {
    s -> a[ ++ (s -> top)] = ch;
}
void pop(struct stack *s) {
    -- (s -> top);
}
char peek(struct stack *s) {
    return s -> a[(s -> top)];
}
int isCorrect(char *exp, int size) {
    struct stack *s = createStack(size);
    for (int i = 0; i < size; i ++) {
        char ch = exp[i];
        if ( (ch == '{') || (ch == '[') || (ch == '(') ){
            push(s, ch);
        }
        else if ( isEmpty(s)) {
            return 0;
        }
        else {
            char topElement = peek(s);
            switch(ch) {
                case '}': if (topElement == '{') {
                    pop(s);
                }
                else {
                    return 0;
                }
                break;
                case ')': if (topElement == '(') {
                    pop(s);
                }
                else {

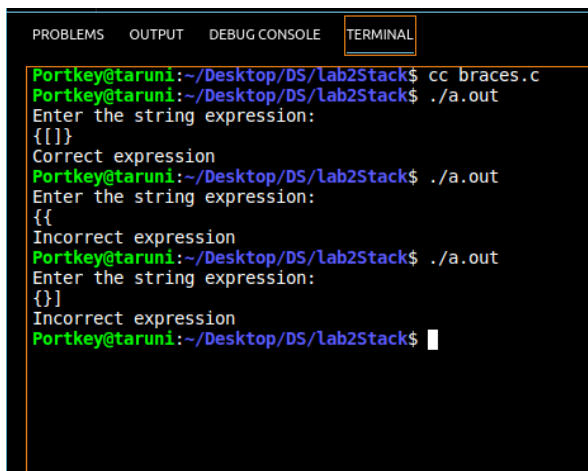
```



```

        return 0;
    }
    break;
case ']': if (topElement == '[') {
    pop(s);
}
else {
    return 0;
}
break;
default : break;
}
}
}
return ( (isEmpty(s)) ? (1) : (0));
}
int main() {
    int size;
    char exp[50];
    printf("Enter the string expression:\n");
    scanf("%s", exp);
    size = strlen(exp);
    if (isCorrect(exp, size)) {
        printf("Correct expression\n");
    }
    else {
        printf("Incorrect expression\n");
    }
    return 0;
}

```



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Portkey@taruni:~/Desktop/DS/Lab2Stack$ cc braces.c
Portkey@taruni:~/Desktop/DS/Lab2Stack$ ./a.out
Enter the string expression:
{[]}]
Correct expression
Portkey@taruni:~/Desktop/DS/Lab2Stack$ ./a.out
Enter the string expression:
{{
Incorrect expression
Portkey@taruni:~/Desktop/DS/Lab2Stack$ ./a.out
Enter the string expression:
{)}
Incorrect expression
Portkey@taruni:~/Desktop/DS/Lab2Stack$

```

## Program3:

### Algorithm:

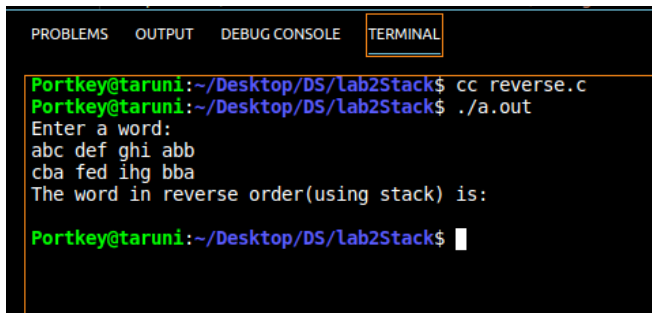
1. Start.
2. Take the input sentence(space separated words) from user
3. Create a stack with size as size of string.
4. Traverse the string char by char and push them into the stack until you encounter a space.
5. When space is encountered, pop all the elements of the stack until it becomes empty and print a space after this.
6. Repeat the steps until u reach the last character of string
7. To print the last word (since we do not encounter a space after the last word), pop all the characters of stack until it's empty and print those characters.
8. Stop.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
struct stack {
    int size;
    int top;
    char *a;
};
int isEmpty(struct stack *s) {
    return ( s -> top == -1) ? (1) : (0);
}
int isFull(struct stack *s) {
    return (s -> top == s -> size - 1) ? (1) : (0);
}
struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> size = size;
    s -> a = (char *)malloc (size * sizeof(char));
    s -> top = -1;
    return s;
}
void push(struct stack *s, char ch) {
    s -> a[ ++ (s -> top)] = ch;
}
char pop(struct stack *s) {
    return (s -> a[ (s -> top) --]);
}
int main() {
```

```

int size, i = 0;
char word[100];
printf("Enter a word:\n");
fgets(word, 100, stdin);
size = strlen(word);
struct stack *s;
s = createStack(size);
for ( int i = 0; i < size; i ++){
    char letter = word[i];
    if ( (! isspace(letter)) && (letter != '\n') ) {
        push(s, letter);
    }
    else {
        while ( !(isEmpty(s)) ) {
            printf("%c", pop(s));
        }
        printf("%c", letter);
    }
}
printf("The word in reverse order(using stack) is:\n");
while ( !(isEmpty(s)) ) {
    printf("%c", pop(s));
}
printf("\n");
return 0;
}

```



The screenshot shows a terminal window with the following content:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Portkey@taruni:~/Desktop/DS/Lab2Stack$ cc reverse.c
Portkey@taruni:~/Desktop/DS/Lab2Stack$ ./a.out
Enter a word:
abc def ghi abb
cba fed ihg bba
The word in reverse order(using stack) is:

Portkey@taruni:~/Desktop/DS/Lab2Stack$ 

```

## Pre Lab Questions:

Ans1. Stack is a data structure used to store data in such a way that the first element to enter is the last to leave. The open end of the stack is called top, along which we can push(insert) or pop(delete) elements.

Ans2. Applications of Stack:

- Web pages history(visited pages).
- HTML opening and closing tags.
- Balancing of braces and parentheses.
- Undo sequence in an editor.
- Function calls and return statements.

Ans3. No, we cannot delete any element from a stack. We can only delete the most recent element that is the element which is at the top or that entered last. This is because a stack is open only at one end.

Ans4. Empty condition is indicated if the top pointer of a stack is less than 0 (or -1). Full condition is indicated if the top pointer equals (sizeofStack - 1).

Ans5. It is possible to implement 3 stacks in one array only if the sizes of all the stacks are the same. If the sizes of all the stacks aren't the same, we cannot do so.

# Lab-3:

## Program1:

### Algorithm for infix to postfix:

This involves a stack for storing operators of the given infix expression.

1. Start.
2. Traverse through the string, character by character.
3. If the character is an operand, simply print it out ( or store it to the postfix expression string)
4. If it's an operator, check if the operator stack is empty.
  - a. If it's empty, push it into the stack.
  - b. If not, if it's an opening brace, then push it into the operator stack.
  - c. If not, if it's a closing brace, then pop the operators from the stack until you encounter an opening brace and append to the postfix exp string(or print out)
  - d. If not, check the precedence of the top element of the stack.
    - i. If the precedence of the present operator is greater than that in the stack, push it into the stack.
    - ii. If not, if its precedence is less than that of the top element of stack, pop out the stack, and then push into the stack( until the precedence is greater)
    - iii. If it's equal, and if the associativity is left to write, pop until precedence is greater or the stack becomes empty.
5. Finally, if you traverse the entire string and the operator stack isn't empty, pop out the stack until it's empty and add to the exp string or print out.
6. Stop.

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

```
struct stack {
    int size;
    int top;
    char *a;
};

int isEmpty(struct stack *s) {
    return ( s -> top == -1) ? (1) : (0);
};

struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> a = (char *)malloc (s -> size * sizeof(char));
```

```

    s -> top = -1;
    return s;
};

void push(struct stack *s, int term) {
    s -> a[ ++ (s -> top)] = term;
};

char pop(struct stack *s) {
    return s -> a[(s -> top) --];
};

char peek(struct stack *s) {
    return s -> a[(s -> top)];
};

int precedence(char ch) {
    switch(ch) {
        case '+': return 1;
        case '-': return 1;
        case '/': return 2;
        case '*': return 2;
        case '^': return 3;
        case '(': return 4;
    }
}

void infToPost(char infExp[100]) {
    int index = 0, contains = 0;
    int len = strlen(infExp);
    struct stack *operatorStack;
    operatorStack = createStack(len);
    printf("The postfix expression is:\n");
    for ( index = 0; index < len; index ++ ) {
        char ch = infExp[index];
        if ( isspace(ch)) {
            continue;
        }
        if ( isalnum(ch) ) { //if its an operand
            if ( isalpha(ch)) {
                printf("%c ", ch); //if operands are entered as variables
            }
            else {
                while ( !isspace(ch)) {
                    printf("%c", ch);
                    ch = infExp[ ++ index];
                }
                printf(" ");
            }
        }
    }
}

```

```

    }
    else { //if its an operator
        if (ch == '(' ) {
            push(operatorStack, ch);
            contains += 1;
        }
        else if (isEmpty(operatorStack)) {
            push(operatorStack, ch);
        }

        else if ( ch == ')') {
            while ( peek(operatorStack) != '('){
                printf("%c ", pop(operatorStack));
            }
            char waste = pop(operatorStack);
            contains -= 1;
        }
        else if ( (precedence( peek(operatorStack)) < precedence(ch)) || contains ) {
            push(operatorStack, ch);
        }
        else {
            while (precedence( peek(operatorStack)) >= precedence(ch)) {
                printf("%c ", pop(operatorStack));
                if (isEmpty(operatorStack)) {
                    break;
                }
            }
            push(operatorStack, ch);
        }
    }
}

while ( !isEmpty(operatorStack)) {
    printf("%c ", pop(operatorStack));
}

}

int main() {
    char infExp[100];
    printf("Enter the infix exp (with space between the each operator and operand):\n");
    fgets(infExp, 100, stdin);
    infToPost(infExp);
    return 0;
}

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/Lab3$ cc inftopost.c
Portkey@taruni:~/Desktop/DS/Lab3$ ./a.out
Enter the infix exp (with space between the each operator and operand):
a * b - ( c + d ) + e
The postfix expression is:
a b * c d + - e + Portkey@taruni:~/Desktop/DS/Lab3$
```

## Program2:

### Algorithm for evaluation of postfix:

1. Start.
2. Traverse through the given postfix expression.
3. If it's an operand, push it into the stack.
4. If its an operator, evaluate it with its operators as the top two elements of the stack(pop them) and push the result to the stack
5. When we reach the end of the string, the element/number left in the stack is the final answer of the given postfix expression.
6. Stop.

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

struct stack {
    int size;
    int top;
    int *a;
};

int isFull(struct stack *s) {
    return (s -> top == s -> size - 1) ? (1) : (0);
};

int isEmpty(struct stack *s) {
    return ( s -> top == -1) ? (1) : (0);
};

struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> a = (int *)malloc (s -> size * sizeof(int));
    s -> top = -1;
    return s;
};

void push(struct stack *s, int term) {
    s -> a[ ++ (s -> top)] = term;
```



```

};
int pop(struct stack *s) {
    return s -> a[(s -> top) --];
};
int peek(struct stack *s) {
    return s -> a[(s -> top)];
};
int operate(int a, int b, char op) {
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
        case '^': return a ^ b;
    }
}
int evaluate(char postExp[100]) {
    int index = 0;
    int len = strlen(postExp);
    struct stack *operandStack;
    operandStack = createStack(len);
    for ( index = 0; index < len; index ++ ) {
        char ch = postExp[index];
        if ( isspace(ch) ) {/// if its a space
            //printf("space\n");
            continue;
        }
        else if ( isdigit(ch) ) {///if its an operator
            int num = (int)(ch - '0');
            ch = postExp[ ++ index];
            while ( isdigit(ch) ) {
                num = num * 10 + (int)(ch - '0');
                ch = postExp[ ++ index];
            }
            push(operandStack, num);
            //printf("pushed a num\n");
        }
        else {
            int a, b;
            b = pop(operandStack);
            a = pop(operandStack);
            int res = operate(a, b, ch);
            push(operandStack, res);
            //printf("Performed an op\n");
        }
    }
}

```

```

        //printf("%d \n", res);

    }
}
return peek(operandStack);
}
int main() {
    char postExp[100];
    printf("Enter the postfix exp (with space between the each operator and operand):\n");
    fgets(postExp, 100, stdin);
    printf("Result = %d\n", evaluate(postExp));
    return 0;
}

```

```

Portkey@taruni:~/Desktop/DS/Lab3$ cc postEval.c
Portkey@taruni:~/Desktop/DS/Lab3$ ./a.out
Enter the postfix exp (with space between the each operator and operand):
100 200 + 2 / 5 * 7 +
Result = 757
Portkey@taruni:~/Desktop/DS/Lab3$

```

## Program3:

### Algorithm for infix to prefix:

1. Start.
2. Reverse the given expression and treat the opening brace as the closing brace and vice versa.
3. Then find the postfix of this expression.
4. And then finally reverse the result again.
5. This gives the prefix of the infix expression given.
6. Stop.

```

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

```

```

struct stack {
    int size;
    int top;
    char *a;
};
int isEmpty(struct stack *s) {

```

```

    return ( s -> top == -1) ? (1) : (0);
};
struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> a = (char *)malloc (s -> size * sizeof(char));
    s -> top = -1;
    return s;
};
void push(struct stack *s, int term) {
    s -> a[ ++ (s -> top)] = term;
};
char pop(struct stack *s) {
    return s -> a[(s -> top) --];
};
char peek(struct stack *s) {
    return s -> a[(s -> top)];
};
int precedence(char ch) {
    switch(ch) {
        case '+': return 1;
        case '-': return 1;
        case '/': return 2;
        case '*': return 2;
        case '^': return 3;
        case ')': return 4;
    }
}
void display(struct stack *s) {
    for (int i = s -> top ; i >= 0; i --) {
        printf("%c", s -> a[i]);
    }
    return;
}
void infToPost(char infExp[100]) {
    int index = 0, contains = 0;
    int len = strlen(infExp);
    struct stack *operatorStack, *preExp;
    operatorStack = createStack(len);
    preExp = createStack(len+1);
    printf("The prefix expression is:\n");
    for ( index = len - 2; index >= 0; index --) {
        char ch = infExp[index];
        if ( isspace(ch)) {
            continue;

```

```

    }
    if ( isalnum(ch) ) { //if its an operand
        push(preExp, ch); //if operands are entered as variables
        push(preExp, ' ');
    }
    else { //if its an operator
        if (ch == ')') {
            push(operatorStack, ch);
            contains += 1;
        }
        else if (isEmpty(operatorStack)) {
            push(operatorStack, ch);
        }
        else if ( ch == '(') {
            while ( peek(operatorStack) != ' '){
                push(preExp, pop(operatorStack));
                push(preExp, ' ');
            }
            char waste = pop(operatorStack);
            contains -= 1;
        }
        else if ( (precedence( peek(operatorStack)) <= precedence(ch)) || contains ) {
            push(operatorStack, ch);
        }
        else {
            while (precedence( peek(operatorStack)) > precedence(ch)) {
                push(preExp, pop(operatorStack));
                push(preExp, ' ');
                if (isEmpty(operatorStack)) {
                    break;
                }
            }
            push(operatorStack, ch);
        }
    }
}

while ( !isEmpty(operatorStack)) {
    push(preExp, pop(operatorStack));
    push(preExp, ' ');
}
display(preExp);
}

int main() {
    char infExp[100];

```

```

printf("Enter the infix exp (with space between the each operator and operand):\n");
fgets(infExp, 100, stdin);
//printf("%c\n", infExp[strlen(infExp) - 2]);
infToPost(infExp);

return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/lab3$ cc inftopre.c
Portkey@taruni:~/Desktop/DS/lab3$ ./a.out
Enter the infix exp (with space between the each operator and operand):
a + b + ( c * d )
The prefix expression is:
+ + a b * c dPortkey@taruni:~/Desktop/DS/lab3$

```

## Program4:

### Algorithm for infix to prefix with right to left associativity:

1. Start.
2. This is similar to evaluating infix to postfix expressions. The only change is, when the operator has right to left associativity, we can push it into the stack even if it's precedence is the same as that of the top operator in the stack.
3. Stop.

```

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

```

```

struct stack {
    int size;
    int top;
    char *a;
};

int isEmpty(struct stack *s) {
    return ( s -> top == -1) ? (1) : (0);
};

struct stack * createStack(int size) {
    struct stack *s = (struct stack *) malloc(sizeof(struct stack));
    s -> a = (char *)malloc (s -> size * sizeof(char));
    s -> top = -1;
    return s;
}

```

```

};
void push(struct stack *s, int term) {
    s -> a[ ++ (s -> top)] = term;
};
char pop(struct stack *s) {
    return s -> a[(s -> top) --];
};
char peek(struct stack *s) {
    return s -> a[(s -> top)];
};
int precedence(char ch) {
    switch(ch) {
        case '+': return 1;
        case '-': return 1;
        case '/': return 2;
        case '*': return 2;
        case '^': return 3;
        case '(': return 4;
    }
}
void infToPost(char infExp[100]) {
    int index = 0, contains = 0;
    int len = strlen(infExp);
    struct stack *operatorStack;
    operatorStack = createStack(len);
    printf("The postfix expression is:\n");
    for ( index = 0; index < len; index ++ ) {
        char ch = infExp[index];
        if ( isspace(ch)) {
            continue;
        }
        if ( isalnum(ch) ) { //if its an operand
            if ( isalpha(ch)) {
                printf("%c ", ch); //if operands are entered as variables
            }
            else {
                while ( !isspace(ch)) {
                    printf("%c", ch);
                    ch = infExp[ ++ index];
                }
                printf(" ");
            }
        }
        else { //if its an operator

```

```

if (ch == '(' ) {
    push(operatorStack, ch);
    contains += 1;
}
else if (isEmpty(operatorStack)) {
    push(operatorStack, ch);
}

else if ( ch == ')') {
    while ( peek(operatorStack) != '('){
        printf("%c ", pop(operatorStack));
    }
    char waste = pop(operatorStack);
    contains -= 1;
}
else if ( ch == '^') {
    if ( (precedence( peek(operatorStack)) <= precedence(ch)) || contains ) {
        push(operatorStack, ch);
    }
    else {
        while (precedence( peek(operatorStack)) > precedence(ch)) {
            printf("%c ", pop(operatorStack));
            if (isEmpty(operatorStack)) {
                break;
            }
        }
        push(operatorStack, ch);
    }
}
else {
    if ( (precedence( peek(operatorStack)) < precedence(ch)) || contains ) {
        push(operatorStack, ch);
    }
    else {
        while (precedence( peek(operatorStack)) >= precedence(ch)) {
            printf("%c ", pop(operatorStack));
            if (isEmpty(operatorStack)) {
                break;
            }
        }
        push(operatorStack, ch);
    }
}
}

```

```

    }
    while ( !isEmpty(operatorStack)) {
        printf("%c ", pop(operatorStack));
    }
}

int main() {
    char infExp[100];
    printf("Enter the infix exp (with space between the each operator and operand):\n");
    fgets(infExp, 100, stdin);
    infToPost(infExp);
    return 0;
}

```

```

Portkey@taruni:~/Desktop/DS/lab3$ cc inftopost2.c
Portkey@taruni:~/Desktop/DS/lab3$ ./a.out
Enter the infix exp (with space between the each operator and operand):
( ( a + b ) * c ) - d ^ e ^ f
The postfix expression is:
a b + c * d e f ^ ^ - Portkey@taruni:~/Desktop/DS/lab3$

```

## Pre Lab questions:

1. free() function is used for de-allocating memory at runtime for stacks.
2. -isFull() - O(1)  
 -isEmpty() - O(1)  
 -createStack() - O(1)  
 -peek() - O(1)  
 -display() - O(n)  
 -pop() - O(1)  
 -push() - O(1)
3. s->arr = (int \*) malloc (2\*size\*size(int))
4. After converting infix to postfix, the order of operands does not change, only the positions change and the order of operators can change.  
 Ex: if infix = a+b\*(c^d-e)^(f+g\*h), postfix expression = abcd^e-fgh\*+^\*+
5. i) SSSXXSSXSXXX gives 325641 ii) Not possible



# Lab-4:

## Pogram1: Linear Queues

### Algorithm for Linear queue:

1. Enqueue operation: Check if the queue is full.
  - If the queue is full, print("full") and exit.
  - If the queue is not full, increment the rear pointer to point to the next empty space. Add the data element to the queue location where the rear is pointing.
2. Dequeue operation: Check if the queue is empty.
  - If the queue is empty, print("empty") and exit.
  - If the queue is not empty, access the data where the front is pointing.
  - Increment the front pointer to point to the next available data element.
3. Peek : check if the queue is empty.
  - If it's empty, print("empty")
  - Otherwise, print the element pointed by the front pointer.
4. Display: If the queue is empty, print empty.  
Else, display the elements from front pointer to rear pointer.

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

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

```
struct queue {  
    int rear, front, size;  
    int *a;  
};  
int isEmpty(struct queue *q) {  
    return q -> rear == -1 ? 1: 0;  
}  
int isFull( struct queue *q) {  
    return (q -> rear == q -> size -1) ? 1: 0;  
}  
struct queue * createQueue() {  
    struct queue* q;  
    q = (struct queue*)malloc(sizeof(struct queue));  
    q -> front = -1;  
    q -> rear = -1;  
    printf("Enter size of queue:\n");  
    scanf("%d", & ( q -> size));  
    q -> a = (int *)malloc(q -> size * sizeof(int));  
    return q;  
}
```

```

int peek(struct queue *q) {
    return q -> a[ q -> front];
}

void display(struct queue *q) {
    if ( isEmpty(q)) {
        printf("The queue is empty!");
    }
    else {
        printf("The elements of the queue are:\n");
        for ( int index = q -> front; index <= q -> rear; index ++ ) {
            printf("%d ", q -> a[index]);
        }
    }
    printf("\n");
}

void enqueue(struct queue *q, int num ) {
    if ( isEmpty(q)) {
        q -> front = 0;
    }
    q -> a[ ++ q -> rear] = num;
    printf("Enqueued %d \n", num);
}

int dequeue(struct queue * q) {
    if ( q -> front == q -> size - 1 ) {
        q -> front = -1;
        q -> rear = -1;
        return q -> a[ q -> size - 1];
    }
    return q -> a[ q -> front ++];
}

int main() {
    struct queue *q;
    int ch;
    q = createQueue();
    while ( 1 ) {
        int num;
        printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nPlease enter your choice:\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1 : if ( isFully(q)) {
                printf("The queue is full, cannot insert\n");
            }
            else {

```

```

        printf("Enter num to be inserted:\n") ;
        scanf("%d", &num);
        enqueue(q, num);
    }
    break;
case 3 : if ( isEmpty(q)) {
        printf("The queue is empty!\n");
    }
    else {
        printf("The top element in queue is: %d\n", peek(q));
    }
    break;
case 2 : if ( isEmpty(q)) {
        printf("The queue is empty!\n");
    }
    else {
        printf("The element %d in queue is deleted! \n", dequeue(q));
    }
    break;
case 4 : display(q);
    break;
case 5 : printf("Exiting\n");
    break;
}
if ( ch == 5 ) {
    break;
}
}
return 0;
}

```

```
Portkey@taruni:~/Desktop/DS/Lab4Qs$ ./a.out
Enter size of queue:
4
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
3
The queue is empty!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
2
The queue is empty!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter num to be inserted:
1
Enqueued 1
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter num to be inserted:
2
Enqueued 2
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter num to be inserted:
3
Enqueued 3
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter num to be inserted:
4
Enqueued 4
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
The queue is full, cannot insert
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
2
The element 1 in queue is deleted!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
The queue is full, cannot insert
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
4
2 3 4 1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
5
Exiting
Portkey@taruni:~/Desktop/DS/Lab4Qs$
```

## Program2 : Circular Queues

### Algorithm circular queues:

1. Enqueue: Check  $((\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} == 0) \ || \ (\text{rear} == \text{front} - 1))$ .
  - If it is full, then display Queue is full.
  - If the queue is not full then, check if  $(\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} != 0)$  if it is true then set rear to 0 and insert element.
2. Dequeue: Check whether the queue is Empty.

- If it is empty, then display Queue is empty.
- If queue is not empty then Check if (front == rear)
  - if it is true then set front and rear to -1
  - else check if (front == size - 1),
    - if it is true then set front to 0 and return the element.
    - otherwise return the front element and increment the front pointer by one

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

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

```
struct queue {
```

```
    int rear, front, size;
```

```
    int *a;
```

```
};
```

```
int isEmpty(struct queue *q) {
```

```
    if (q -> rear == -1) {
```

```
        return 1;
```

```
    }
```

```
    return 0;
```

```
}
```

```
int isFull( struct queue *q) {
```

```
    if ((q -> front == 0) && ( q -> rear == q -> size -1)) {
```

```
        return 1;
```

```
    }
```

```
    return 0;
```

```
}
```

```
struct queue * createQueue() {
```

```
    struct queue* q;
```

```
    q = (struct queue*)malloc(sizeof(struct queue));
```

```
    q -> front = -1;
```

```
    q -> rear = -1;
```

```
    printf("Enter size of queue:\n");
```

```
    scanf("%d", & ( q -> size));
```

```
    q -> a = (int *)malloc(q -> size * sizeof(int));
```

```
    return q;
```

```
}
```

```
int peek(struct queue *q) {
```

```
    return q -> a[ q -> front];
```

```
}
```

```
void display(struct queue *q) {
```

```
    if ( isEmpty(q)) {
```

```
        printf("The queue is empty!");
```

```
    }
```

```

else if ( q -> front <= q -> rear) {
    printf("The elements of the queue are:\n");
    for ( int index = q -> front; index <= q -> rear; index ++ ) {
        printf("%d ", q -> a[index]);
    }
}
else {
    printf("The elements of the queue are:\n");
    for ( int index = q -> front; index < q -> size; index ++ ) {
        printf("%d ", q -> a[index]);
    }
    for ( int index = 0; index <= q -> rear; index ++ ) {
        printf("%d ", q -> a[index]);
    }
}
printf("\n");
}

void enqueue(struct queue *q) {
    if ( isEmpty(q) ) {
        q -> front = 0;
        printf("Enter number to be enqueued: ");
        scanf("%d", & (q -> a [ ++ q -> rear ]));
        printf("Enqueued successfully!\n");
    }
    else if (isFull(q)) {
        printf("Cannot enqueue as queue is full!\n");
    }
    else if ( ( q -> front > 0 ) && ( q -> rear == q-> size - 1 ) ) {
        q -> rear = 0;
        printf("Enter number to be enqueued: ");
        scanf("%d", & (q -> a [ q -> rear ]));
        printf("Enqueued successfully!\n");
    }
    else if ( q -> rear == q -> front - 1 ) {
        printf("Cannot enqueue as the queue is full!\n");
    }
    else {
        printf("Enter number to be enqueued: ");
        scanf("%d", & (q -> a [ ++ q -> rear ]));
        printf("Enqueued successfully!\n");
    }
}

void dequeue(struct queue * q) {

```

```

if ( isEmpty(q)) {
    printf("Cannot dequeue as the queue is empty!\n");
}
else if ( (q -> rear == q -> size - 1) && (q -> front == q -> size - 1) ) {
    printf("%d is dequeued!\n", q -> a [ q -> front]);
    q -> front = -1;
    q -> rear = -1;
}
else if ( q -> front == q -> size - 1 ) {
    printf("%d is dequeued!\n", q -> a [ q -> front]);
    q -> front = 0;
}
else if ( q -> front == q -> rear ) {
    printf("%d is dequeued!\n", q -> a [ q -> front]);
    q -> front = -1;
    q -> rear = -1;
}
else {
    printf("%d is dequeued!\n", q -> a [ q -> front ++]);
}
}

int main() {
    struct queue *q;
    int ch;
    q = createQueue();
    while ( 1 ) {
        int num;
        printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nPlease enter your
choice:\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1 : enqueue(q);
                break;
            case 3 : if ( isEmpty(q)) {
                printf("The queue is empty!\n");
            }
            else {
                printf("The top element in queue is: %d\n", peek(q));
            }
            break;
            case 2 : dequeue(q);
                break;
            case 4 : display(q);
                break;

```

```

        case 5 : printf("Exiting\n");
                break;
    }
    if ( ch == 5) {
        break;
    }
}
return 0;
}

```

```

Portkey@taruni:~/Desktop/DS/lab4Qs$ cc circularQs.c
Portkey@taruni:~/Desktop/DS/lab4Qs$ ./a.out
Enter size of queue:
4
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter number to be enqueued: 1
Enqueued succesfully!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter number to be enqueued: 2
Enqueued succesfully!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1

```

```

Enter number to be enqueued: 3
Enqueued succesfully!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter number to be enqueued: 4
Enqueued succesfully!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Cannot enqueue as queue is full!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
4
The elements of the queue are:
1 2 3 4
1. Enqueue

```



```

2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
2
1 is dequeued!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
4
The elements of the queue are:
2 3 4
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
3
The top element in queue is: 2
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit

```

```

Please enter your choice:
1
Enter number to be enqueued: 10
Enqueued succesfully!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
4
The elements of the queue are:
2 3 4 10
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
5
Exiting
Portkey@taruni:~/Desktop/DS/Lab4Qs$

```

## Pre Lab questions:

1. A queue is a logical data structure which works on the principle of "First in First out" or "Last in Last out".
2. Multiprogramming, tickets at a movie theatre, waiting times of customers at call centres etc.
3. A circular queue is a linear data structure, similar to a linear queue, but differs because the end of a circular queue is connected to the front of the queue.
4. Traffic signals, CPU processes.
5. In a linear queue, even though there is space before the front pointer due to dequeuing of elements, it cannot be utilised. But in a circular queue, all the space can be utilised efficiently.

# Lab-5:

## Program1:

### Algorithm:

1. Start.
2. Insert at the beginning: if the head is null, make head as the newnode. If not, make newnode-> next as head and point head to the new node.
3. Insert at a given position: if the given position is one, call insertFirst function. If not, if the head is null, print that it cannot be inserted. If the head is not null and the position is one more than the count, call insertEnd function. If the head is not null and the position is neither null nor one more than the count.. Traverse the linked list till the count -2 position, make newnode -> next to that element's next. Then make the element's next to point to newnode.
4. Insert at the end: if the head is null, make newnode = head. If not, traverse till the last element of list (until u get p -> next = null) and make the next to point to newnode.
5. Delete from beginning: if the head is null, print that the list is empty and cannot be deleted. If the head is not null, store head-> data in a variable, make head = head -> next.
6. Delete from the end: if the head is null, print that the list is empty and cannot be deleted. If the head is not null, traverse till the count - 2 element, store the that and make that element -> next to null.
7. Delete from a given position: If the head is null, print that the list is empty and cannot be deleted. If the head is not null, if position is one, call delete first and if position is count, call the deleteLast function. If the head is not null and position is greater than count, print that the list is empty and cannot be deleted. If not, traverse till the position - 2 element, store the that and make that element -> next to null.
8. Displaying: If the head is null, print that there are no elements to display. Otherwise, take a pointer initially pointing to the head and until p becomes null, make it p -> next and display the p -> data.
9. stop

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

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

```
struct node {
```

```
    int data;
```

```
    struct node *next;
```

```
};
```

```
struct node *head;
```

```
int count = 0;
```

```
void insertFirst( int num) {
```

```
    struct node *newnode;
```

```

newnode = (struct node *)malloc(sizeof(struct node));
newnode -> data = num;
newnode -> next = NULL;
if ( head ) {
    newnode -> next = head;
}
head = newnode;
++ count;
printf("Inserted!\n");
}

void insertLast(int num) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( !head ) {
        head = newnode;
        ++ count;
    }
    else {
        p = (struct node *)malloc(sizeof(struct node));
        p = head;
        while ( p -> next != NULL ) {
            p = p -> next;
        }
        p -> next = newnode;
        ++ count;
    }
    printf("Inserted!\n");
}

void insertAt(int num, int pos) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( pos == 1 ) {
        insertFirst(num);
    }
    else if ( head == NULL ) {
        printf("The linked list is empty. Cannot insert at the given position\n");
    }
    else if ( pos == count + 1 ) {
        insertLast(num);
    }
}

```

```

else if (pos > (count + 1)) {
    printf("Cannot insert at the given position\n");
}
else {
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    int i = 1;
    while (i < pos - 1) {
        p = p -> next;
        ++ i;
    }
    newnode -> next = p -> next;
    p -> next = newnode;
    ++ count;
    printf("Inserted!\n");
}
}

int deleteFirst() {
    if (!head) {
        return -1;
    }
    int num = head -> data;
    head = head -> next;
    -- count;
    return num;
}

int deleteLast() {
    if (!head) {
        return -1;
    }
    struct node *p;
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    int i = 1;
    while (i < count - 1) {
        p = p -> next;
        ++ i;
    }
    int num = p -> next -> data;
    p -> next = NULL;
    -- count;
    return num;
}

int deleteFrom(int pos) {

```

```

struct node *p;
if ( head == NULL) {
    printf("The linked list is empty.");
    return -1;
}
else if ( pos == 1){
    return deleteFirst();
}
else if ( pos == count) {
    return deleteLast();
}
else if ( pos > count) {
    return -1;
}
else {
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    int i = 1;
    while ( i < pos - 1) {
        p = p -> next;
        ++ i;
    }
    int num = p -> next -> data;
    p -> next = p -> next -> next;
    -- count;
    return num;
}
}

void display() {
    struct node *p;
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    if ( ! p) {
        printf("No elements to display\n");
    }
    else {
        while ( p != NULL) {
            printf("%d ", p -> data);
            p = p -> next;
        }
        printf("\n");
    }
}

int main() {

```

```

head = (struct node *)malloc(sizeof(struct node));
head = NULL;
int ch;
printf("1. Insert at the beginning\n2. Insert at a given position\n3. Insert at the end\n4. Delete
from beginning\n5. Delete at a giving position\n6. Delete from the end\n7. Display the linked
list\n8. Display count\n9. Exit\nEnter your choice: ");
scanf("%d", &ch);
while ( ch != 9 ) {
    int num, pos;
    switch (ch) {
        case 1: printf("Enter number to be inserted: ");
                scanf("%d", &num);
                insertFirst(num);
                break;
        case 2: printf("Enter the number to be inserted : ");
                scanf("%d", &num);
                printf("Enter the position to be inserted at: ");
                scanf("%d", &pos);
                insertAt(num, pos);
                break;
        case 3: printf("Enter number to be inserted: ");
                scanf("%d", &num);
                insertLast(num);
                break;
        case 4: num = deleteFirst();
                if ( num == -1 ) {
                    printf("No elements present to delete\n");
                }
                else {
                    printf("%d is the deleted element\n", num);
                }
                break;
        case 5: printf("Enter the position to be deleted from: ");
                scanf("%d", &pos);
                num = deleteFrom(pos);
                if ( num == -1 ) {
                    printf("No element to be deleted at position %d\n", pos);
                }
                else {
                    printf("%d is the element deleted at %d position\n", num, pos);
                }
                break;
        case 6: num = deleteLast();
                if ( num == -1 ) {

```

```

        printf("No elements present to delete\n");
    }
    else {
        printf("%d is the deleted element\n", num);
    }
    break;
case 7: display();
    break;
case 8: printf("%d is the count\n", count);
    break;
}
printf("1. Insert at the beginning\n2. Insert at a given position\n3. Insert at the end\n4.
Delete from beginning\n5. Delete at a giving position\n6. Delete from the end\n7. Display the
linked list\n8. Display count\n9. Exit\nEnter your choice: ");
scanf("%d", &ch);
}
return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Enter your choice: 7
1 2
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 3
Enter number to be inserted: 3
Inserted!
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 3
Enter number to be inserted: 4
Inserted!
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 7

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
8. Display count
9. Exit
Enter your choice: 7
1 2 3 4
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 4
1 is the deleted element
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 6
4 is the deleted element
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 2

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Enter the number to be inserted : 999
Enter the position to be inserted at: 2
Inserted!
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 7
2 999 3
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: 5
Enter the position to be deleted from: 2
999 is the element deleted at 2 position
1. Insert at the beginning
2. Insert at a given position
3. Insert at the end
4. Delete from beginning
5. Delete at a giving position
6. Delete from the end
7. Display the linked list
8. Display count
9. Exit
Enter your choice: █
```

## Program2:

### Algorithm:

1. Start
2. Keep track of the number of elements in the linked list with help of a variable 'count'.
3. Send one number more than the entered position to the insertAt function to insert at the position.
4. Stop

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

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

```
int count = 0;
void insertLast(int num) {
    struct node *newnode, *p;
```



```

newnode = (struct node *)malloc(sizeof(struct node));
newnode -> data = num;
newnode -> next = NULL;
if ( !head ) {
    head = newnode;
    ++ count;
}
else {
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    while ( p -> next != NULL ) {
        p = p -> next;
    }
    p -> next = newnode;
    ++ count;
}
printf("Inserted!\n");
}

void insertAt(int num, int pos) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if (pos == count + 1 ) {
        insertLast(num);
    }
    else if (pos > (count + 1)) {
        printf("Cannot insert at the given position\n");
    }
    else {
        p = (struct node *)malloc(sizeof(struct node));
        p = head;
        int i = 1;
        while ( i < pos - 1 ) {
            p = p -> next;
            ++ i;
        }
        newnode -> next = p -> next;
        p -> next = newnode;
        ++ count;
        printf("Inserted!\n");
    }
}

void display() {

```

```

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

int main() {
    int size, num, pos;
    head = (struct node *)malloc(sizeof(struct node));
    head = NULL;
    printf("Enter the initial size of linked list: ");
    scanf("%d", &size);
    printf("Enter the elements of linked list:\n");
    for ( int i = 0; i < size; i ++) {
        scanf("%d", &num);
        insertLast(num);
    }
    printf("Enter a number to be inserted: ");
    scanf("%d", &num);
    printf("Enter the position to be inserted at: ");
    scanf("%d", &pos);
    insertAt(num, pos + 1);
    printf("The linked list after insertion is:\n");
    display();
    return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/Lab5LL$ cc prog2.c
Portkey@taruni:~/Desktop/DS/Lab5LL$ ./a.out
Enter the initial size of linked list: 5
Enter the elements of linked list:
1
Inserted!
2
Inserted!
3
Inserted!
4
Inserted!
5
Inserted!
Enter a number to be inserted: 99
Enter the position to be inserted at: 2
Inserted!
The linked list after insertion is:
1
2
99
3
4
5
Portkey@taruni:~/Desktop/DS/Lab5LL$

```

## Program3:

### Algorithm:

1. Start.
2. Keep track of the number of elements in the linked list with help of a variable 'count'.
3. Send one number less than the entered position to the insertAt function to insert at the position.
4. Stop

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

struct node {
    int data;
    struct node *next;
};
struct node *head;
int count = 0;
void insertFirst( int num) {
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( head ) {
        newnode -> next = head;
    }
    head = newnode;
    ++ count;
}
void insertLast(int num) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( !head ) {
        head = newnode;
        ++ count;
    }
    else {
        p = (struct node *)malloc(sizeof(struct node));
        p = head;
        while ( p -> next != NULL) {
            p = p -> next;
```

```

    }
    p -> next = newnode;
    ++ count;
}
}

void insertAt(int num, int pos) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( pos == 1 || pos == 0) {
        insertFirst(num);
    }
    else {
        p = (struct node *)malloc(sizeof(struct node));
        p = head;
        int i = 1;
        while ( i < pos - 1) {
            p = p -> next;
            ++ i;
        }
        newnode -> next = p -> next;
        p -> next = newnode;
        ++ count;
    }
}

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

int main() {
    int size, num;
    head = (struct node *)malloc(sizeof(struct node));
    head = NULL;
    printf("Enter the initial size of linked list: ");
    scanf("%d", &size);
    printf("Enter the elements of linked list:\n");
    for ( int i = 0; i < size; i ++ ) {

```

```

        scanf("%d", &num);
        insertLast(num);
    }
    printf("Enter a number to be inserted: ");
    scanf("%d", &num);
    int pos = -1, highestMin, i;
    struct node *p;
    p = (struct node *)malloc(sizeof(struct node));
    p = head;
    for ( i = 1; i <= count; i ++ ) {
        if ( p -> data < num ) {
            pos = i;
            highestMin = p -> data;
            p = p -> next;
            break;
        }
        p = p -> next;
    }
    if ( pos == -1 ) {
        insertFirst(num);
    }
    else {
        while ( p != NULL ) {
            ++ i;
            if ( p -> data < num && p -> data > highestMin ) {
                pos = i;
                highestMin = p -> data;
            }
            p = p -> next;
        }
        insertAt(num, pos);
    }
    printf("The linked list after insertion is:\n");
    display();
    return 0;
}

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/lab5LL$ cc prog3.c
Portkey@taruni:~/Desktop/DS/lab5LL$ ./a.out
Enter the initial size of linked list: 5
Enter the elements of linked list:
1
3
5
7
9
Enter a number to be inserted: 8
The linked list after insertion is:
1
3
5
8
7
9
Portkey@taruni:~/Desktop/DS/lab5LL$
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/lab5LL$ ./a.out
Enter the initial size of linked list: 3
Enter the elements of linked list:
1
3
4
Enter a number to be inserted: 2
The linked list after insertion is:
2
1
3
4
Portkey@taruni:~/Desktop/DS/lab5LL$ ./a.out
Enter the initial size of linked list: 3
Enter the elements of linked list:
4
5
6
Enter a number to be inserted: 2
The linked list after insertion is:
2
4
5
6
Portkey@taruni:~/Desktop/DS/lab5LL$
```

## Program4:

### Algorithm:

1. Start.
2. Use the insertFirst function of the linked list as the push function of a stack and deleteFirst function as the pop option of the stack.
3. The top element , i.e the peek is by printing the element at head node.
4. Stop

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

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

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

```
struct stack {
    struct node* top;
};
```

```
int count = 0;
```

```
void push( struct stack *s, int num) {
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( s -> top ) {
        newnode -> next = s -> top;
```

```

    }
    s -> top = newnode;
    ++ count;
    printf("Pushed %d into the stack!\n", num);
}

int pop(struct stack *s) {
    if (!s -> top) {
        return -1;
    }
    int num = s -> top -> data;
    s -> top = s -> top -> next;
    -- count;
    return num;
}

void display(struct stack *s) {
    if ( ! s -> top) {
        printf("No elements to display\n");
    }
    else {
        struct node *p;
        p = (struct node *)malloc(sizeof(struct node));
        p = s -> top;
        printf("The elements in the stack are:\n ");
        while ( p != NULL) {
            printf("%d ", p -> data);
            p = p -> next;
        }
        printf("\n");
    }
}

int main() {
    struct stack* s;
    s = (struct stack *)malloc(sizeof(struct stack));
    s -> top = NULL;
    printf("1. Push \n2. Pop\n3. Peek\n4. Display\n5. Exit\nEnter your choice: ");
    int ch, num;
    scanf("%d", &ch);
    while ( ch != 5) {
        switch (ch) {
            case 1: printf("Enter the number to be pushed: ");
                    scanf("%d", &num);
                    push(s, num);
                    break;
            case 2: num = pop(s);

```

```

        if ( num == -1) {
            printf("No elements to delete\n");
        }
        else {
            printf("%d is popped from the stack\n", num);
        }
        break;
    case 3: if ( s -> top == NULL) {
            printf("The stack is empty\n");
        }
        else {
            printf("%d is the top element\n", s -> top -> data);
        }
        break;
    case 4: display(s);
        break;
    case 5: printf("Exiting\n");
        break;
}
printf("1. Push\n2. Pop\n3. Peek\n4. Display \n5. Exit\nEnter your choice: ");
scanf("%d", &ch);
}
return 0;
}

```

```

Portkey@taruni:~/Desktop/DS/lab5LL$ cc prog4.c
Portkey@taruni:~/Desktop/DS/lab5LL$ ./a.out
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
No elements to display
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the number to be pushed: 1
Pushed 1 into the stack!
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the number to be pushed: 2
Pushed 2 into the stack!
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the number to be pushed: 3
Pushed 3 into the stack!
1. Push
2. Pop
3. Peek

```

```

2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter the number to be pushed: 3
Pushed 3 into the stack!
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
The elements in the stack are:
3 2 1
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
3 is the top element
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
3 is popped from the stack
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 5
Portkey@taruni:~/Desktop/DS/lab5LL$

```



## Program5:

### Algorithm:

1. Start.
2. Use two pointers front and rear.
3. Front acts as the head pointer.
4. Each time we want to enqueue(insertlast), increment the rear pointer to the next node.
5. When we want to dequeue(deleteFirst), increment the front pointer to the next node.  
Initially both front and rear are null.
6. Stop

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

struct node {
    int data;
    struct node *next;
};
struct queue {
    struct node *front, *rear;
};
int count = 0;
void enqueue(struct queue *q, int num) {
    struct node *newnode, *p;
    newnode = (struct node *)malloc(sizeof(struct node));
    newnode -> data = num;
    newnode -> next = NULL;
    if ( ! q -> rear ) {
        q -> front = newnode;
        q -> rear = newnode;
        ++ count;
    }
    else {
        q -> rear -> next = newnode;
        q -> rear = newnode;
        ++ count;
    }
    printf("Enqueued!\n");
}
int dequeue(struct queue *q) {
    if (! q -> front) {
        return -1;
    }
    int num = q -> front -> data;
```

```

    q -> front = q -> front -> next;
    -- count;
    return num;
}
int peek(struct queue *q) {
    if (! q -> front) {
        return -1;
    }
    return q -> front -> data;
}
void display(struct queue *q) {
    struct node *p;
    p = (struct node *)malloc(sizeof(struct node));
    p = q -> front;
    if ( ! p) {
        printf("No elements to display\n");
    }
    else {
        printf("The elements of the queue are:\n");
        while ( p != NULL) {
            printf("%d ", p -> data);
            p = p -> next;
        }
        printf("\n");
    }
}
int main() {
    struct queue *q;
    q = (struct queue *)malloc(sizeof(struct queue));
    q -> rear = q -> front = NULL;
    int ch, num;
    printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nPlease enter your choice:\n");
    scanf("%d", &ch);
    while ( ch != 5) {
        switch (ch) {
            case 1: printf("Enter the number to be enqueued: ");
                    scanf("%d", &num);
                    enqueue(q, num);
                    break;
            case 2: num = dequeue(q);
                    if ( num == -1) {
                        printf("No elements in the queue to delete\n");
                    }
                    else {

```

```

        printf("%d is dequeued from the queue\n", num);
    }
    break;
case 3: num = peek(q);
    if ( num == -1) {
        printf("No elements in the queue\n");
    }
    else {
        printf("%d is the top element of the queue\n", num);
    }
    break;
case 4: display(q);
    break;
case 5: printf("Exiting\n");
    break;
}
printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nPlease enter your
choice:\n");
scanf("%d", &ch);
}
return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/lab5LL$ cc prog5.c
Portkey@taruni:~/Desktop/DS/lab5LL$ ./a.out
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
4
No elements to display
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter the number to be enqueued: 1
Enqueued!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter the number to be enqueued: 2
Enqueued!
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
Enter the number to be enqueued: 3

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
5. Exit
Please enter your choice:
4
3 2 1
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
3
1 is the top element of the queue
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
2
2 is dequeued from the queue
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
1
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Please enter your choice:
5
Portkey@taruni:~/Desktop/DS/lab5LL$

```

## Pre lab questions:

1. We can keep track of the number of elements in the linked list (the num of nodes) with the help of a variable 'count'. Each time we insert, we increment the count. Each time we delete, we decrement the count. In this way we can calculate the mid element by  $\text{count}/2$  and traverse through the linked list.
2. We can have two variables 'first' and 'second'. Initially, first points to head and second points to head -> next. Assume the list is in ascending order. First check if the head is > given element. If it's greater, insert the element at first. If not, run a while loop until 'second' != null. Every time, increase first and second to their respective next. Check if 'second' is greater than element. When u find that second is greater than element, use first to insert second in that position and shift second. If this does not happen even when the second reaches null, insert element at the last.
3. We can reverse the linked list and then print from the head pointer.

# Lab-6:

## Program1:

### Algorithm:

Addition:

1. Start.
2. Loop around all values of the linked list and follow step 2 & 3.
3. If the value of a node's exponent is greater copy this node to result node and move towards the next node.
4. If the values of both node's exponent is same add the coefficients and then copy the added value with node to the result and move towards the next node in both the lists.
5. Print the resultant node.
6. Stop

Multiplication:

Step1: We will multiply the 2nd polynomial with each term of 1st polynomial.

Step2: Store the multiplied value in a new linked list.

Step3: Then we will add the coefficients of elements having the same power in resultant polynomial.

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

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

```
#include <math.h>
```

```
struct poly {  
    int coeff, power;  
    struct poly *next;
```

```
};
```

```
void insertEnd(struct poly* exp, int coeff, int power) {
```

```
    if (!exp) {
```

```
        exp -> coeff = coeff;
```

```
        exp -> power = power;
```

```
        exp -> next = NULL;
```

```
    }
```

```
    else {
```

```
        struct poly *p, *newnode;
```

```
        p = (struct poly*)malloc(sizeof(struct poly));
```

```
        newnode = (struct poly*)malloc(sizeof(struct poly));
```

```
        newnode -> coeff = coeff;
```

```
        newnode -> power = power;
```

```

    newnode -> next = NULL;
    p = exp;
    while ( p -> next != NULL) {
        p = p -> next;
    }
    p -> next = newnode;
}
}

void display(struct poly *exp) {
    struct poly *p;
    p = (struct poly*)malloc(sizeof(struct poly));
    p = exp;
    while ( p-> next != NULL) {
        printf("%dx^%d + ", p -> coeff, p -> power);
        p = p -> next;
    }
    printf("%dx^%d\n", p -> coeff, p -> power);
}

void add(struct poly *exp1, struct poly *exp2, struct poly *res) {
    struct poly *p1, *p2, *newnode, *p;
    p1 = (struct poly*)malloc(sizeof(struct poly));
    p2 = (struct poly*)malloc(sizeof(struct poly));
    p = (struct poly*)malloc(sizeof(struct poly));
    p = res;
    p1 = exp1;
    p2 = exp2;
    int count = 0;
    while ( p1 != NULL && p2 != NULL) {
        newnode = (struct poly*)malloc(sizeof(struct poly));
        if ( p1 -> power > p2 -> power) {
            newnode -> coeff = p1 -> coeff;
            newnode -> power = p1 -> power;
            newnode -> next = NULL;
            p1 = p1 -> next;
        }
        else if ( p1 -> power < p2 -> power) {
            newnode -> coeff = p2 -> coeff;
            newnode -> power = p2 -> power;
            newnode -> next = NULL;
            p2 = p2 -> next;
        }
        else {
            newnode -> coeff = p2 -> coeff + p1 -> coeff;
            newnode -> power = p1 -> power;

```

```

        newnode -> next = NULL;
        p1 = p1 -> next;
        p2 = p2 -> next;
    }
    if ( count == 0) {
        res -> coeff = newnode -> coeff;
        res -> power = newnode -> power;
        res -> next = NULL;
        ++ count;
    }
    else {
        p -> next = newnode;
        p = p -> next;
    }
}
if ( p1 != NULL) {
    while ( p1 != NULL) {
        p -> next = p1;
        p = p -> next;
        p1 = p1 -> next;
    }
}
else if ( p2 != NULL) {
    while ( p2 != NULL) {
        p -> next = p2;
        p = p -> next;
        p2 = p2 -> next;
    }
}
}

float eval(struct poly *exp) {
    struct poly *p;
    p = (struct poly*)malloc(sizeof(struct poly));
    float res = 0, x;
    printf("Enter the x value: ");
    scanf("%f", &x);
    p = exp;
    while ( p != NULL) {
        res += (p -> coeff * pow(x, p -> power));
        p = p -> next;
    }
    return res;
}

void multiply(struct poly *exp1, struct poly *exp2, struct poly *res) {

```

```

struct poly *p, *q, *r, *newterm;
p = (struct poly*)malloc(sizeof(struct poly));
q = (struct poly*)malloc(sizeof(struct poly));
r = (struct poly*)malloc(sizeof(struct poly));
res -> coeff = exp1 -> coeff * exp2 -> coeff;
res -> power = exp1 -> power + exp2 -> power;
res -> next = NULL;
q = exp1 ;
r = exp2 -> next;
while ( q != NULL) {
    p = res;
    while ( r != NULL) {
        newterm = (struct poly*)malloc(sizeof(struct poly));
        newterm -> coeff = q -> coeff * r -> coeff;
        newterm -> power = q -> power + r -> power;
        newterm -> next = NULL;
        while ( p -> next != NULL) {
            if ( p -> power == newterm -> power) {
                p -> coeff += newterm -> coeff;
                break;
            }
            p = p -> next;
        }
        if ( p -> power == newterm -> power) {
            p -> coeff += newterm -> coeff;
        }
        else {
            p -> next = newterm;
        }
        r = r -> next;
    }
    q = q -> next;
    r = exp2;
}
}

int main() {
    struct poly *exp1, *exp2, *res1, *res2;
    int n1, n2, coeff, power;
    exp1 = (struct poly*)malloc(sizeof(struct poly));
    exp2 = (struct poly*)malloc(sizeof(struct poly));
    res1 = (struct poly*)malloc(sizeof(struct poly));
    res2 = (struct poly*)malloc(sizeof(struct poly));
    printf("Enter number of terms of first expression ");
    scanf("%d", &n1);

```



```

for (int i = 0; i < n1; i++) {
    printf("Enter coeff and power of term %d:\n", (i + 1));
    scanf("%d%d", &coeff, &power);
    if (i == 0) {
        exp1 -> coeff = coeff;
        exp1 -> power = power;
        exp1 -> next = NULL;
    }
    else {
        insertEnd(exp1, coeff, power);
    }
}
printf("The first expression is: ");
display(exp1);
printf("Enter number of terms of second expression ");
scanf("%d", &n2);
for (int i = 0; i < n2; i++) {
    printf("Enter coeff and power of term %d:\n", (i + 1));
    scanf("%d%d", &coeff, &power);
    if (i == 0) {
        exp2 -> coeff = coeff;
        exp2 -> power = power;
        exp2 -> next = NULL;
    }
    else {
        insertEnd(exp2, coeff, power);
    }
}
printf("The second expression is: ");
display(exp2);
int count = 0;
while (count < 3) {
    printf("Enter 1 for addition 2 for multiplication and 3 for evaluating the expression.\n Enter
your choice: ");
    int num;
    scanf("%d", &num);
    if (num == 1) {
        add(exp1, exp2, res1);
        printf("The result after addition is:\n");
        display(res1);
    }
    else if (num == 2) {
        printf("The result after multiplication is:\n");
        multiply(exp1, exp2, res2);
    }
}

```

```

        display(res2);
    }
    else {
        printf("Expression 1 or 2?: ");
        int ch;
        scanf("%d", &ch);
        float evalRes;
        if (ch == 1) {
            evalRes = eval(exp1);
        }
        else {
            evalRes = eval(exp2);
        }
        printf("The result is: %.2f\n", evalRes);
    }
    ++ count;
}
return 0;
}

```

```

Portkey@taruni:~/Desktop/DS/lab6LL$ cc polyExp.c -lm
Portkey@taruni:~/Desktop/DS/lab6LL$ ./a.out
Enter number of terms of first expression 3
Enter coeff and power of term 1:
1
4
Enter coeff and power of term 2:
2
2
Enter coeff and power of term 3:
2
1
The first expression is: 1x^4 + 2x^2 + 2x^1
Enter number of terms of second expression 2
Enter coeff and power of term 1:
2
2
Enter coeff and power of term 2:
1
0
The second expression is: 2x^2 + 1x^0
Enter 1 for addition 2 for multiplication and 3 for evaluating the expression.
Enter your choice: 2
The result after multiplication is:
2x^6 + 5x^4 + 2x^2 + 4x^3 + 2x^1
Enter 1 for addition 2 for multiplication and 3 for evaluating the expression.
Enter your choice: 1
The result after addition is:
1x^4 + 4x^2 + 2x^1 + 1x^0
Enter 1 for addition 2 for multiplication and 3 for evaluating the expression.
Enter your choice: 3
Expression 1 or 2?: 1
Enter the x value: 1
The result is: 5.00
Portkey@taruni:~/Desktop/DS/lab6LL$

```

## Program2:

### Algorithm:

The sparse matrix used anywhere in the program is sorted according to its row values. Two elements with the same row values are further sorted according to their column values. Now to Add the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into the resultant matrix.

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

struct matrix {
    int row, col, val;
    struct matrix *next;
};

void insertEnd(struct matrix* a, int row, int col, int val) {
    if (!exp) {
        a -> row = row;
        a -> col = col;
        a -> val = val;
        a -> next = NULL;
    }
    else {
        struct matrix *p, *newnode;
        p = (struct matrix*)malloc(sizeof(struct matrix));
        newnode = (struct matrix*)malloc(sizeof(struct matrix));
        newnode -> row = row;
        newnode -> col = col;
        newnode -> val = val;
        newnode -> next = NULL;
        p = a;
        while (p -> next != NULL) {
            p = p -> next;
        }
        p -> next = newnode;
    }
}

void display(struct matrix *a) {
    struct matrix *p;
    p = (struct matrix*)malloc(sizeof(struct matrix));
```

```

p = a;
printf("ROW  COLUMN  VALUE\n");
while ( p-> next != NULL) {
    printf("%d      %d      %d\n", p -> row, p -> col, p -> val);
    p = p -> next;
}
printf("%d      %d      %d\n", p -> row, p -> col, p -> val);
}

void add(struct matrix *a, struct matrix *b, struct matrix *c) {
    struct matrix *p, *p1, *p2, *newnode;
    p = (struct matrix*)malloc(sizeof(struct matrix));
    p1 = (struct matrix*)malloc(sizeof(struct matrix));
    p2 = (struct matrix*)malloc(sizeof(struct matrix));
    p = c;
    p1 = a;
    p2 = b;
    int count = 0, row, col, val;
    while ( p1 != NULL && p2 != NULL ) {
        newnode = (struct matrix*)malloc(sizeof(struct matrix));
        if (p1 -> row < p2 -> row) {
            newnode -> row = p1 -> row;
            newnode -> col = p1 -> col;
            newnode -> val = p1 -> val;
            newnode -> next = NULL;
            p1 = p1 -> next;
        }
        else if ( p1 -> row > p2 -> row) {
            newnode -> row = p2 -> row;
            newnode -> col = p2 -> col;
            newnode -> val = p2 -> val;
            newnode -> next = NULL;
            p2 = p2 -> next;
        }
        else {
            if ( p1 -> col < p2 -> col) {
                newnode -> row = p1 -> row;
                newnode -> col = p1 -> col;
                newnode -> val = p1 -> val;
                newnode -> next = NULL;
                p1 = p1 -> next;
            }
            else if ( p1 -> col > p2 -> col ) {
                newnode -> row = p2 -> row;
                newnode -> col = p2 -> col;

```

```

        newnode -> val = p2 -> val;
        newnode -> next = NULL;
        p2 = p2 -> next;
    }
    else {
        newnode -> row = p1 -> row;
        newnode -> col = p1 -> col;
        newnode -> val = p1 -> val + p2 -> val;
        newnode -> next = NULL;
        p1 = p1 -> next;
        p2 = p2 -> next;
    }
}
if ( count == 0 ) {
    c -> row = newnode -> row;
    c -> col = newnode -> col;
    c -> val = newnode -> val;
    c -> next = NULL;
    ++ count;
}
else {
    p -> next = newnode;
    p = p -> next;
}
}
if ( p1 != NULL ) {
    while ( p1 != NULL ) {
        p -> next = p1;
        p = p -> next;
        p1 = p1 -> next;
    }
}
else if ( p2 != NULL ) {
    while ( p2 != NULL ) {
        p -> next = p2;
        p = p -> next;
        p2 = p2 -> next;
    }
}
}
int main() {
    struct matrix *a, *b, *res;
    int n1, n2, row, col, val;
    a = (struct matrix*)malloc(sizeof(struct matrix));

```

```

b = (struct matrix*)malloc(sizeof(struct matrix));
res = (struct matrix*)malloc(sizeof(struct matrix));
printf("Enter number of non-zero terms of first matrix ");
scanf("%d", &n1);
for (int i = 0; i < n1; i++) {
    printf("Enter row, col and value of term %d:\n", (i + 1));
    scanf("%d%d%d", &row, &col, &val);
    if (i == 0) {
        a->row = row;
        a->col = col;
        a->val = val;
        a->next = NULL;
    }
    else {
        insertEnd(a, row, col, val);
    }
}
printf("The first matrix is:\n");
display(a);
printf("Enter number of non-zero terms of second matrix ");
scanf("%d", &n2);
for (int i = 0; i < n2; i++) {
    printf("Enter row, col and value of term %d:\n", (i + 1));
    scanf("%d%d%d", &row, &col, &val);
    if (i == 0) {
        b->row = row;
        b->col = col;
        b->val = val;
        b->next = NULL;
    }
    else {
        insertEnd(b, row, col, val);
    }
}
printf("The second matrix is: ");
display(b);
add(a, b, res);
printf("The result after addition is:\n");
display(res);
return 0;
}

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Portkey@taruni:~/Desktop/DS/lab6LL$ ./a.out
Enter number of non-zero terms of first matrix 3
Enter row, col and value of term 1:
1
1
1
Enter row, col and value of term 2:
2
2
1
Enter row, col and value of term 3:
3
3
1
The first matrix is:
ROW    COLUMN    VALUE
1       1         1
2       2         1
3       3         1
Enter number of non-zero terms of second matrix 3
Enter row, col and value of term 1:
1
2
1
Enter row, col and value of term 2:
3
1
1
Enter row, col and value of term 3:
3
3
1
The second matrix is:
ROW    COLUMN    VALUE
1       2         1
1       3         1
3       3         1
The result after addition is:
ROW    COLUMN    VALUE
1       1         1
1       2         1
1       3         1
2       2         1
3       3         2
Portkey@taruni:~/Desktop/DS/lab6LL$
```

## Pre Lab questions:

1. Unlike linked lists, for polynomials using arrays, we should maintain 2 arrays, one for the coefficients and one for the powers of each coefficient. If the number of terms are large, to store the coefficients and powers we need large arrays. As we know arrays are contiguous memory allocation, thus, the contiguous memory may not be available. In case we have to add or subtract 2 expressions, we would be needing 6 arrays in total (including result) !!
2. Poly creation:  $O(1)$   
Poly addition:  $O(n)$   
Poly multiplication:  $O(n*m)$  where  $m$  and  $n$  are number of terms of the two expressions  
Poly evaluation:  $O(1)$

# Lab-8\_9:

## Program1:

### Algorithm:

1. Start.
2. insert: –If the tree is empty, initialize the root with new node  
–Else, Push root into Queue  
–Get the front node of the queue
  - If the left child of this front node doesn't exist, set the new node as the left child
  - Else if the right child of this front node doesn't exist, set the new node as the right child  
–If the front node has both left child and right child, Dequeue it  
–Enqueue the new node.
3. delete node: Starting at root, find the node which we want to delete  
–Find the deepest, right most node in the tree.  
–Replace the deepest right most node's data with nodes to be deleted.  
–Then delete the deepest rightmost node.
4. Delete entire tree: Make the head/ root node null.
5. Pre order: Visit the root  
–Traverse the left subtree in PreOrder  
–Traverse the right subtree in PreOrder
6. In order: Traverse the left subtree in InOrder  
– Visit the root  
–Traverse the right subtree in InOrder
7. Post order: Traverse the left subtree in PostOrder  
–Traverse the right subtree in PostOrder  
–Visit the root
8. Level order: Use a queue to store the addresses of the nodes visited. Initially the queue contains only the address of the head node. As we empty the queue, we print its value and enqueue the right and left nodes' address into the queue.
9. Search: Start with the root node.  
–Recurse down the tree,  
–Choose the left or right branch by comparing data with each node's data
10. Counting number of nodes: If the node is a null node, return 0. Otherwise return 1 and recursively call the function for the node's right and left child.
11. Height: If the node is null, return 0. Otherwise, return 1 and maximum of height of left and right nodes. (recursive)
12. Sum of nodes: Similar to the number of nodes except here we return the data of the node instead of 1 if the node is not null.



### 13. Stop

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

int size;
struct Node {
    int data;
    struct Node *rightChild, *leftChild;
};
struct Node *rootNode;

struct Queue {
    int front, rear;
    struct Node **arr;
}; //Queue to store addresses of nodes while creation

void enqueue(struct Queue *q, struct Node *node) {
    if ( q -> front > 0 && q -> rear == size - 1 ) {
        q -> arr [0] = node;
        q -> rear = 0;
    }
    else {
        q -> arr[++ q -> rear] = node;
    }
} //enqueue operation

struct Node * dequeue(struct Queue *q) {
    if ( q -> front == size -1 && q -> rear < q -> front ) {
        q -> front = 0;
        return q -> arr[size - 1];
    }
    return q -> arr[ q -> front  ++];
} //dequeue operation

int isEmpty(struct Queue *q) {
    return (q -> front == q -> rear + 1) ;
} //checking if queue is empty

void createTree(struct Queue* q, int val) {
    q -> front = 0;
    q -> rear = -1;
    q -> arr = (struct Node **)malloc(size * sizeof(struct Node*));
    rootNode = (struct Node *)malloc(sizeof(struct Node));
```

```

rootNode -> data = val;
rootNode -> rightChild = NULL;
rootNode -> leftChild = NULL;
q -> arr[ ++ q -> rear] = rootNode;
struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
ptr = rootNode;
while (!isEmpty(q)) { //reading till queue becomes empty
    ptr = dequeue(q);
    printf("Enter the leftChild of %d: ", ptr -> data);
    int val, val2;
    scanf("%d", &val); //reading leftChild of current node
    if ( val != -1) {
        struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
        newnode -> data = val;
        newnode -> rightChild = NULL;
        newnode -> leftChild = NULL;
        enqueue(q, newnode);
        ptr -> leftChild = newnode;
    }
    printf("Enter the rightChild of %d: ", ptr -> data);
    scanf("%d", &val2); //reading rightChild of current node
    if ( val2 != -1) {
        struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
        newnode -> data = val2;
        newnode -> rightChild = NULL;
        newnode -> leftChild = NULL;
        enqueue(q, newnode);
        ptr -> rightChild = newnode;
    }
}
} //creating queue by storing addreses of nodes in queue

int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(struct Node * p) {
    if (p) return 1 + max(height( p -> leftChild ), height(p -> rightChild));
    return 0;
} //function to calculate height of a tree/node

int numOfNodes(struct Node *p) {
    if (p) return 1 + numOfNodes(p -> leftChild) + numOfNodes(p -> rightChild);
    return 0;
}

```

```
} //counting total number of nodes of a tree using recursion
```

```
int sumOfNodes(struct Node *p) {  
    if (p) return p -> data + sumOfNodes(p -> leftChild) + sumOfNodes(p -> rightChild);  
    return 0;  
} //calculating sum of nodes using recursion
```

```
void preOrder(struct Node * p) {  
    if (p) {  
        printf("%d ", p -> data);  
        preOrder(p -> leftChild);  
        preOrder(p -> rightChild);  
    }  
} //preorder traversal using recursion
```

```
void postOrder(struct Node * p) {  
    if (p) {  
        postOrder(p -> leftChild);  
        postOrder(p -> rightChild);  
        printf("%d ", p -> data);  
    }  
} //postOrder traversal using recursion
```

```
void inOrder(struct Node * p) {  
    if (p) {  
        inOrder(p -> leftChild);  
        printf("%d ", p -> data);  
        inOrder(p -> rightChild);  
    }  
} //inorder traversal using recursion
```

```
void levelOrder(struct Node *rootNode) {  
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));  
    q -> front = 0;  
    q -> rear = -1;  
    q -> arr = (struct Node **)malloc(size * sizeof(struct Node*));  
    enqueue(q, rootNode);  
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));  
    printf("%d ", rootNode -> data);  
    while (!isEmpty(q)) {  
        p = dequeue(q);  
        if ( p -> leftChild) {  
            printf("%d ", p -> leftChild -> data);  
            enqueue(q, p -> leftChild);  
        }  
    }  
}
```

```

    }
    if (p -> rightChild) {
        printf("%d ", p -> rightChild -> data);
        enqueue(q, p -> rightChild);
    }
}
} //level order traversal using queues to store addresses of node

```

```

void insertNode(struct Node *rootNode, int val) {
    struct Node *p, *newnode;
    p = (struct Node *)malloc(sizeof(struct Node));
    if (!rootNode) {
        rootNode -> data = val;
        rootNode -> leftChild = NULL;
        rootNode -> rightChild = NULL;
        return;
    }
    newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode -> data = val;
    newnode -> leftChild = NULL;
    newnode -> rightChild = NULL;
    struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
    q -> front = 0;
    q -> rear = -1;
    q -> arr = (struct Node **)malloc(size * sizeof(struct Node*));
    enqueue(q, rootNode);
    while (!isEmpty(q)) {
        p = dequeue(q);
        if (p -> leftChild) {
            enqueue(q, p -> leftChild);
        }
        else {
            p -> leftChild = newnode;
            return;
        }
        if (p -> rightChild) {
            enqueue(q, p -> rightChild);
        }
        else {
            p -> rightChild = newnode;
            return;
        }
    }
}
} //traversal by level order and inserting at the first vacant position

```

```

struct Node * getDeepestNode(struct Node *rootNode) {
    struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
    q -> front = 0;
    q -> rear = -1;
    q -> arr = (struct Node **)malloc(sizeof(struct Node*));
    enqueue(q, rootNode);
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p = rootNode;
    while (!isEmpty(q)) {
        p = dequeue(q);
        if (p -> leftChild) enqueue(q, p -> leftChild);
        if (p -> rightChild) enqueue(q, p -> rightChild);
        if (isEmpty(q)) return p;
    }
} //finding deepest node of i.e the last entered node of level order traversal

```

```

void deleteNode(struct Node *rootNode, int toReplace) {
    struct Node *p, *replaceWith;
    p = (struct Node *)malloc(sizeof(struct Node));
    replaceWith = (struct Node *)malloc(sizeof(struct Node));
    replaceWith = getDeepestNode(rootNode);
    struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
    q -> arr = (struct Node **)malloc(sizeof(struct Node *));
    q -> front = 0;
    q -> rear = -1;
    enqueue(q, rootNode);
    while (!isEmpty(q)) {
        p = dequeue(q);
        if (p -> data == toReplace) p -> data = replaceWith -> data;
        if (p -> rightChild == replaceWith) p -> rightChild = NULL;
        if (p -> leftChild == replaceWith) p -> leftChild = NULL;
        if (p -> leftChild) enqueue(q, p -> leftChild);
        if (p -> rightChild) enqueue(q, p -> rightChild);
    }
} //deleting the desired node and replacing it with deepest node of tree.

```

```

int find(struct Node * p, int key) {
    if (!p) return 0;
    if (p -> data == key) return 1;
    if (find(p -> leftChild, key)) return 1;
    if (find(p -> rightChild, key)) return 1;
    return 0;
} //searching if a given vaue/key present in the tree or not

```

```

int main() {
    size = 20;
    int rootVal;
    struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
    printf("Enter root node value: ");
    scanf("%d", &rootVal);
    createTree(q, rootVal);
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    int ch;
    do {
        ptr = rootNode;
        printf("1. Insert a node\n2. Delete a node\n3. Delete entire tree\n4. PreOrder\n5.
InOrder\n6. PostOrder\n7. LevelOrder\n8. Search a node\n9. Count no. of nodes\n10. Height of
tree\n11. Sum of nodes\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1: { int num;
                printf("Enter number to be inserted: ");
                scanf("%d", &num);
                insertNode(ptr, num);
            }
            break;
            case 2: {int num;
                printf("Enter number to be deleted: ");
                scanf("%d", &num);
                deleteNode(ptr, num);
            }
            break;
            case 3: rootNode = NULL;
                ptr = NULL;
            break;
            case 4: printf("PreOrder: ");
                preOrder(ptr);
                printf("\n");
            break;
            case 5: printf("InOrder: ");
                inOrder(ptr);
                printf("\n");
            break;
            case 6: printf("PostOrder: ");
                postOrder(ptr);
                printf("\n");

```

```

        break;
    case 7: printf("Level order: ");
        levelOrder(ptr);
        printf("\n");
        break;
    case 8: printf("Enter node to be searched: ");
        int num;
        scanf("%d", &num);
        if (find(ptr, num)) printf("Found!\n");
        else printf("Not found!");
        break;
    case 9: printf("Number of nodes = %d\n", numOfNodes(ptr));
        break;
    case 10: printf("Height of tree = %d\n", height(ptr));
        break;
    case 11: printf("Sum of nodes = %d\n", sumOfNodes(ptr));
        break;
    default: printf("Exiting !\n");
        break;
    }
} while (ch <= 11);
return 0;
}

```

```
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ cc treeswithLL.c
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ ./a.out
Enter root node value: 1
Enter the leftChild of 1: 2
Enter the rightChild of 1: 3
Enter the leftChild of 2: -1
Enter the rightChild of 2: -1
Enter the leftChild of 3: -1
Enter the rightChild of 3: -1
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 4
PreOrder: 1 2 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 1
Enter number to be inserted: 4
```



```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 5
InOrder: 4 2 1 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 2
Enter number to be deleted: 1

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 6
PostOrder: 2 3 4
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 8
Enter node to be searched: 10
Not found!1. Insert a node

```

```

2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 7
Level order: 4 2 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 9
Number of nodes = 3

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 10
Height of tree = 2
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 11
Sum of nodes = 9

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 4
PreOrder:

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 12
Exiting !
portkey@Taruni:/mnt/c/User

```

## Program2:

### Algorithm:

In this we use an array to represent a binary tree. For every node  $i$ , its right child can be given by  $2 * i$  and left child can be given by  $2 * i + 1$ .

The capacity or number of nodes is indicated with a variable named mostRecentIndex.

1. Start.
2. insert: –If the tree is empty, initialize the root with new node
  - Else, Push root into Queue
  - Get the front node of the queue
    - If the left child of this front node doesn't exist, set the new node as the left child
    - Else if the right child of this front node doesn't exist, set the new node as the right child
  - If the front node has both left child and right child, Dequeue it
  - Enqueue the new node.
3. delete node: Starting at root, find the node which we want to delete
  - Find the deepest, right most node in the tree.

- Replace the deepest right most node's data with nodes to be deleted.
- Then delete the deepest rightmost node.
- 4. Delete entire tree: Make the head/ root node null.
- 5. Pre order: Visit the root
  - Traverse the left subtree in PreOrder
  - Traverse the right subtree in PreOrder
- 6. In order: Traverse the left subtree in InOrder
  - Visit the root
  - Traverse the right subtree in InOrder
- 7. Post order: Traverse the left subtree in PostOrder
  - Traverse the right subtree in PostOrder
  - Visit the root
- 8. Level order: Use a queue to store the addresses of the nodes visited. Initially the queue contains only the address of the head node. As we empty the queue, we print its value and enqueue the right and left nodes' address into the queue.
- 9. Search: Start with the root node.
  - Recurse down the tree,
  - Choose the left or right branch by comparing data with each node's data
- 10. Counting number of nodes: The most recent index gives the number of nodes
- 11. Height: If the node is null, return 0. Otherwise, return 1 and maximum of height of left and right nodes. (recursive)
- 12. Sum of nodes: Recursively call the sum function for left and right nodes and return the data of the current node. If the current node is null, return 0.
- 13. Stop

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

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

```
int mostRecentIndex = 1; //keeping track of the most recent index entered into the tree (array)
int capacity = 50; //capacity/size of array
```

```
void createTree(int completeTree[50]) {
    int num;
    printf("Enter the rootNode: ");
    scanf("%d", &num); //reading the rootNode value from the user
    completeTree[1] = num;
    int index = 1;
    do {
        printf("Enter leftChild of %d: ", completeTree[index]);
        scanf("%d", &num); //reading left child
        if (num == -1) break; //breaking if -1 is entered
        completeTree[index * 2] = num;
        ++ mostRecentIndex;
    } while (1);
}
```

```

        printf("Enter rightChild of %d: ", completeTree[index]);
        scanf("%d", &num); //reading right child
        if ( num == -1) break; //breaking if -1 is entered
        completeTree[index * 2 + 1] = num;
        ++ mostRecentIndex;
        ++ index;
    } while ( mostRecentIndex < 19); //taking an arbitrary size of tree to be taken from the user
}

void insertNode(int completeTree[50], int val) {
    if ( mostRecentIndex < capacity - 1) {
        completeTree[++ mostRecentIndex ] = val;
        //inserting at the first vacant space
        printf("Inserted\n");
    }
    else {
        printf("Cannot insert, no space!\n");
    }
}

void deleteNode(int completeTree[50], int val) {
    for ( int i = 1; i <= mostRecentIndex; i ++ ) {
        if (completeTree[i] == val ) { //searching for node to be deleted
            completeTree[i] = mostRecentIndex --; //if found, replacing it with deepest node (pointed
by mostRecentIndex)
            return;
        }
    }
    printf("Number to be deleted not found!\n");
}

void deleteTree(int completeTree[50]) {
    mostRecentIndex = 0;
    printf("Tree deleted!\n");
} //deleting entireNode

void preOrder(int completeTree[50], int index) {
    if (index <= mostRecentIndex) {
        printf("%d ", completeTree[index]);
        preOrder(completeTree, index * 2);
        preOrder(completeTree, index * 2 + 1);
    }
} //recursive preorder traversal

```

```

void inOrder(int completeTree[50], int index) {
    if (index <= mostRecentIndex) {
        inOrder(completeTree, index * 2);
        printf("%d ", completeTree[index]);
        inOrder(completeTree, index * 2 + 1);
    }
} //recursive inOrder traversal

void postOrder(int completeTree[50], int index) {
    if (index <= mostRecentIndex) {
        postOrder(completeTree, index * 2);
        postOrder(completeTree, index * 2 + 1);
        printf("%d ", completeTree[index]);
    }
} //recursive postOrder traversal

void levelOrder(int completeTree[50]) {
    for (int i = 1; i <= mostRecentIndex; i++) {
        printf("%d ", completeTree[i]);
    }
} //non-recursive (iterative) levelOrder traversal

int find(int completeTree[50], int key) {
    for (int i = 1; i <= mostRecentIndex; i++) {
        if (completeTree[i] == key) return 1; //searching for the desired node, if found returning
1
    }
    return 0; //returning 0 if not found
}

int numOfNodes(int completeTree[50]) {
    return mostRecentIndex;
} //returning the count of number of nodes, i.e the mostRecentIndex itself

int max(int a, int b) {
    return (a > b) ? a : b;
} //function to find the max of two numbers

int height(int completeTree[50], int index) {
    if (index <= mostRecentIndex) return 1 + max(height(completeTree, index * 2),
height(completeTree, index * 2 + 1) );
    return 0;
} //function to calculate height of a tree/node

```

```

int sumOfNodes(int completeTree[50], int index) {
    if ( index <= mostRecentIndex) return completeTree[index] + sumOfNodes(completeTree,
index * 2) + sumOfNodes(completeTree, index * 2 + 1);
    return 0;
} //calculating sum of nodes using recursion

```

```

int main() {
    int completeTree[50] = {0};
    createTree(completeTree);
    int ch;
    do {
        printf("1. Insert a node\n2. Delete a node\n3. Delete entire tree\n4. PreOrder\n5.
InOrder\n6. PostOrder\n7. LevelOrder\n8. Search a node\n9. Count no. of nodes\n10. Height of
tree\n11. Sum of nodes\nEnter your choice: ");
        scanf("%d", & ch);
        switch ( ch)
        {
            case 1: { int num;
                printf("Enter number to be inserted: ");
                scanf("%d", & num);
                insertNode(completeTree, num);
            }
            break;
            case 2: {int num;
                printf("Enter number to be deleted: ");
                scanf("%d", & num);
                deleteNode(completeTree, num);
            }
            break;
            case 3: deleteTree(completeTree);
            break;
            case 4: printf("PreOrder: ");
                preOrder(completeTree, 1);
                printf("\n");
            break;
            case 5: printf("InOrder: ");
                inOrder(completeTree, 1);
                printf("\n");
            break;
            case 6: printf("PostOrder: ");
                postOrder(completeTree, 1);
                printf("\n");
            break;
            case 7: printf("Level order: ");

```

```

        levelOrder(completeTree);
        printf("\n");
        break;
    case 8: printf("Enter node to be searched: ");
        int num;
        scanf("%d", &num);
        if (find(completeTree, num)) printf("Found!\n");
        else printf("Not found!\n");
        break;
    case 9: printf("Number of nodes = %d\n", numOfNodes(completeTree));
        break;
    case 10: printf("Height of tree = %d\n", height(completeTree, 1));
        break;
    case 11: printf("Sum of nodes = %d\n", sumOfNodes(completeTree, 1));
        break;
    default: printf("Exiting !\n");
        break;
    }
} while (ch <= 11);

}

```

```
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ cc treesWithArrays.c
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ ./a.out
Enter the rootNode: 1
Enter leftChild of 1: 2
Enter rightChild of 1: 3
Enter leftChild of 2: 4
Enter rightChild of 2: 5
Enter leftChild of 3: 6
Enter rightChild of 3: 7
Enter leftChild of 4: -1
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 4
PreOrder: 1 2 4 5 3 6 7
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 2
Enter number to be deleted: 4
```



```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 5
InOrder: 7 2 5 1 6 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 1
Enter number to be inserted: 10
Inserted

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 6
PostOrder: 7 5 2 6 10 3 1
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 7
Level order: 1 2 3 7 5 6 10

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 8
Enter node to be searched: 4
Not found!
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 9
Number of nodes = 7

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 10
Height of tree = 3
1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 11
Sum of nodes = 34

```

```

1. Insert a node
2. Delete a node
3. Delete entire tree
4. PreOrder
5. InOrder
6. PostOrder
7. LevelOrder
8. Search a node
9. Count no. of nodes
10. Height of tree
11. Sum of nodes
Enter your choice: 12
Exiting !

```

```
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$
```

## Progam3:

### Algorithm:

1. Start
2. PreOrder Traversal:
  - 1) Create an empty stack nodeStack and push root node to stack.
  - 2) Do following while nodeStack is not empty.
    - a) Pop an item from stack and print it.
    - b) Push right child of popped item to stack
    - c) Push left child of popped item to stackRight child is pushed before left child to make sure that left subtree is processed first.
3. Inorder Traversal:
  - 1) Create an empty stack S.
  - 2) Initialize current node as root
  - 3) Push the current node to S and set current = current->left until current is NULL
  - 4) If current is NULL and stack is not empty then
    - a) Pop the top item from stack.
    - b) Print the popped item, set current = popped\_item->right
    - c) Go to step 3.
  - 5) If current is NULL and stack is empty then we are done.
4. PostOrder traversal:
  - 1) Create an empty stack
  - 2) Do following while root is not NULL
    - a) Push root's right child and then root to stack.
    - b) Set root as root's left child.
  - 3) Pop an item from stack and set it as root.
    - a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
    - b) Else print root's data and set root as NULL.
  - 4) Repeat steps 2.1 and 2.2 while stack is not empty.
5. Stop

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

```
int size;
struct Node {
    int data;
    struct Node *rightChild, *leftChild;
};
struct Node *rootNode;
```

```

struct Queue {
    int front, rear;
    struct Node **arr;
}; //Queue to store addresses of nodes while creation

```

```

struct Stack {
    int top;
    struct Node **arr;
}; //stack to implement the recursive stack for iterative traversal by storing node addresses

```

```

void enqueue(struct Queue *q, struct Node *node) {
    if ( q -> front > 0 && q -> rear == size - 1 ) {
        q -> arr [0] = node;
        q -> rear = 0;
    }
    else {
        q -> arr[++ q -> rear] = node;
    }
} //enqueue operation

```

```

struct Node * dequeue(struct Queue *q) {
    if ( q -> front == size -1 && q -> rear < q -> front ) {
        q -> front = 0;
        return q -> arr[size - 1];
    }
    return q -> arr[ q -> front ++];
} //dequeue operation

```

```

int isEmpty(struct Queue *q) {
    return (q -> front == q -> rear + 1) ;
} //checking if queue is empty

```

```

void push(struct Stack *s, struct Node *node1) {
    s -> arr[++ s -> top ] = node1;
} //push operation

```

```

struct Node* pop( struct Stack *s) {
    return s -> arr[ s -> top --];
} //pop operation

```

```

void createTree(struct Queue* q, int val) {
    q -> front = 0;

```

```

q -> rear = -1;
q -> arr = (struct Node **)malloc(size * sizeof(struct Node*));
rootNode = (struct Node *)malloc(sizeof(struct Node));
rootNode -> data = val;
rootNode -> rightChild = NULL;
rootNode -> leftChild = NULL;
q -> arr[ ++ q -> rear] = rootNode;
struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
ptr = rootNode;
while (!isEmpty(q)) { //reading till queue becomes empty
    ptr = dequeue(q);
    printf("Enter the leftChild of %d: ", ptr -> data);
    int val, val2;
    scanf("%d", &val); //reading leftChild of current node
    if ( val != -1) {
        struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
        newnode -> data = val;
        newnode -> rightChild = NULL;
        newnode -> leftChild = NULL;
        enqueue(q, newnode);
        ptr -> leftChild = newnode;
    }
    printf("Enter the rightChild of %d: ", ptr -> data);
    scanf("%d", &val2); //reading rightChild of current node
    if ( val2 != -1) {
        struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
        newnode -> data = val2;
        newnode -> rightChild = NULL;
        newnode -> leftChild = NULL;
        enqueue(q, newnode);
        ptr -> rightChild = newnode;
    }
}
} //creating queue by storing addresses of nodes in queue

```

```

void preOrder(struct Node * p) {
    struct Stack *s = (struct Stack *)malloc(sizeof(struct Stack));
    s -> arr = (struct Node **) malloc(size * sizeof(struct Node*));
    s -> top = -1;
    while (p || s -> top != -1) {
        if ( p) {
            printf("%d ", p -> data);
            push(s, p);
            p = p -> leftChild;
        }
    }
}

```

```

    }
    else {
        p = pop(s);
        p = p -> rightChild;
    }
}
} //preorder traversal without using recursion

```

```

void inOrder (struct Node *rootNode) {
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p = rootNode;
    struct Stack *s = (struct Stack *)malloc(sizeof(struct Stack));
    s -> arr = (struct Node **) malloc(size * sizeof(struct Node*));
    s -> top = -1;
    while (p || s -> top != -1) {
        if (p) {
            push(s, p);
            p = p -> leftChild;
        }
        else {
            p = pop(s);
            printf("%d ", p -> data);
            p = p -> rightChild;
        }
    }
} //inorder traversal without using recursion

```

```

void postOrder(struct Node *rootnode) {
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p = rootNode;
    struct Node *prev = (struct Node *)malloc(sizeof(struct Node));
    prev = NULL;
    struct Stack *s = (struct Stack *)malloc(sizeof(struct Stack));
    s -> arr = (struct Node **) malloc(size * sizeof(struct Node*));
    s -> top = -1;
    do {
        while (p) {
            push(s, p);
            p = p -> leftChild;
        }
        while ( !p && s -> top != -1 ) {
            p = s -> arr[s -> top];
            if ( !p -> rightChild || p -> rightChild == prev ) {
                printf("%d ", p -> data);
            }
        }
    }
}

```

```

        pop(s);
        prev = p;
        p = NULL;
    }
    else {
        p = p -> rightChild;
    }
}
} while (s -> top != -1 );
} //postOrder traversal without using recursion

int main() {
    size = 20;
    int rootVal;
    struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
    printf("Enter root node value: ");
    scanf("%d", &rootVal);
    createTree(q, rootVal);
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    int ch1;
    do {
        printf("1. PreOrder\n2. InOrder\n3. PostOrder\nEnter your choice: ");
        scanf("%d", &ch1);
        ptr = rootNode;
        switch (ch1) {
            case 1: preOrder(ptr);
                    printf("\n");
                    break;
            case 2: inOrder(ptr);
                    printf("\n");
                    break;
            case 3: postOrder(ptr);
                    printf("\n");
                    break;
            default : printf("Exiting!\n");
                    break;
        }
    } while ( ch1 > 0 && ch1 <= 3);
    return 0;
}

```

```
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ cc treestraversals.c
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$ ./a.out
Enter root node value: 1
Enter the leftChild of 1: 2
Enter the rightChild of 1: 3
Enter the leftChild of 2: 4
Enter the rightChild of 2: 5
Enter the leftChild of 3: -1
Enter the rightChild of 3: -1
Enter the leftChild of 4: -1
Enter the rightChild of 4: -1
Enter the leftChild of 5: -1
Enter the rightChild of 5: -1
1. PreOrder
2. InOrder
3. PostOrder
Enter your choice: 1
1 2 4 5 3
1. PreOrder
2. InOrder
3. PostOrder
Enter your choice: 2
4 2 5 1 3
1. PreOrder
2. InOrder
3. PostOrder
Enter your choice: 3
4 5 2 3 1
1. PreOrder
2. InOrder
3. PostOrder
Enter your choice: 10
Exiting!
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab8_9Trees$
```

## Pre Lab Questions:

1. Traverse level by level and print the first node of each level. Keep track of a variable that increases each time we go to a next level. In this way, each time we go to a different level, we can print the first node's value of that level.
2. Traverse level by level by two pointers, such that one pointer is the parent of the other. When you encounter any of the nodes that we need to check for cousins, store its level number and its parent's address. Similarly store the level number and parent's address of another node that we need to check for cousins. These 2 are cousins only if both are at the same levels and their parent's are not the same.
3. Traverse through the tree in a recursive manner such that, whenever u encounter a leaf we should print the path. If it's not a leaf, then add that element to the path (array) and recursively call the same function for nodes -> leftChild and node -> rightChild. And if u encounter a NULL value, then return.
4. A binary node can be built if we know the inOrder and the postOrder traversal, or the inOrder and preOrder traversal. Procedure: Case 1: When inOrder and PreOrder are given: In this, the PreOrder is used to determine the root node and the inOrder is used to determine if the nodes go left or right of the root node. The first element/number in the preOrder gives us the root node of the tree. By spotting this root node in the inOrder traversal, we can say the elements on the right of this in inOrder are in the right subtree and on the left are in the left subtree. Similarly, going to the next element of the preOrder traversal, we can determine the root of the left/right subtree, depending upon whether it is on the right or left of the root node. By repeating this, we get the entire tree.  
Case2: When inOrder and PostOrder are given: The procedure is entirely the same as that of Case1, except in this, we traverse the postOrder from the last, because we know for sure that the root nodes come last.



# Lab-10:

## Program1:

### Algorithm:

- 1.Start
- 2.A menu driven choice is given to the user which includes Create a tree,Insert,Delete,Search,Inorder and Exit.
- 3.Create a structure for new node so that user can enter values.
- 4.Create a queue function.
- 5.Ask the user to enter the choice.
- 6.If the entered option is Create a tree function then ask the user to enter the root value.
  - ....6.1 Enter left child and right child values to the root node.
  - ....6.2 Now enter left child and right child values for the previous left,right child.
  - ....6.3 Continue this process until entered value is equal to -1.
- 7.If the entered choice is insert function then ask the user to enter the element to be inserted.
  - A new key is always inserted at leaf.
  - We start searching a key from root till we hit a leaf node (move left if key is smaller, and right if it is larger)
  - Once a leaf node is found, the new node is added as a left child of the leaf node if it is smaller than the leaf node / right child of the leaf node if it is larger than the leaf node.
8. If the entered choice is search for a node then ask the user to enter a element to be searched.
  - we first compare it with root, if the key is present at root, we return root.
  - If key is greater than root's key, we recur the search operation for right sub-tree of root node.
  - Otherwise we recur the search operation for left sub-tree of root.
  - If element to search is found anywhere, return true, else return false.
- 9.If the entered choice is delete then ask the user to enter a node to be deleted.
  - 1) Node to be deleted is leaf: Simply remove from the tree. Return NULL to its parent, i.e. make the corresponding child pointer NULL
  - 2) Node to be deleted has only one child: Copy the child key to the node and delete the child.
  - 3) Node to be deleted has both children: There are two options
    - a) Find the Inorder successor of the node (Minimum value in the node's right sub-tree), copy contents of the Inorder successor to the node and delete the Inorder successor.
    - b) Find the Inorder predecessor of the node (Maximum value in the node's left sub-tree), copy contents of the Inorder predecessor to the node and delete the Inorder predecessor.
- 10.If the entered choice is inorder traversal then
  - ...10.1 if tree is not null call inorder of left child,print the value of root and call inorder of right child.

- 11.If the entered choice is exit then program get finished.
- 12.If the entered choice is incorrect then print entered choice is wrong.
- 13.stop.

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

struct Node {
    int data;
    struct Node *rightChild, *leftChild;
};
struct Node *rootNode;

struct Node * createNewNode(int val) {
    struct Node *newnode = (struct Node *)malloc (sizeof(struct Node));
    newnode -> rightChild = NULL;
    newnode -> leftChild = NULL;
    newnode -> data = val;
    return newnode;
}

void insertNode (int val) {
    struct Node *newnode = createNewNode(val);
    if (!rootNode) {
        rootNode = newnode;
        printf("Inserted!\n");
        return;
    } // if rootnode is null, making newnode as the rootnode
    struct Node *p = rootNode;
    while (p) {
        if ( val > p -> data) {
            if ( p -> rightChild ) {
                p = p -> rightChild;
            }
            else {
                p -> rightChild = newnode;
                printf("Inserted!\n");
                return;
            }
        }
        else if ( val < p -> data) {
            if ( p -> leftChild ) {
                p = p -> leftChild;
            }
            else {
                p -> leftChild = newnode;
                printf("Inserted!\n");
                return;
            }
        }
    }
}
```

```

        if ( p -> leftChild ) {
            p = p -> leftChild;
        }
        else {
            p -> leftChild = newnode;
            printf("Inserted!\n");
            return;
        }
    }
}

```

```

int minKey(struct Node *node) {
    struct Node *p = node;
    while (p -> leftChild) {
        p = p -> leftChild;
    }
    return p -> data;
}

```

```

int maxKey(struct Node *node) {
    struct Node *p = node;
    while (p -> rightChild) {
        p = p -> rightChild;
    }
    return p -> data;
}

```

```

int inorderSuc(struct Node *rootNode, struct Node *p) {
    if ( p -> rightChild ) return minKey(p -> rightChild);
    struct Node *suc = NULL;
    struct Node *q = rootNode;
    while (q) {
        if ( q -> data > p -> data ) {
            suc = q;
            q = q -> leftChild;
        }
        else if ( q -> data < p -> data ) {
            q = q -> rightChild;
        }
        else break;
    }
    return suc -> data;
}

```

```

int inorderPre(struct Node *rootNode, struct Node *p) {
    if ( p -> leftChild ) return maxKey(p -> rightChild);
    struct Node *pre = NULL;
    struct Node *q = rootNode;
    while (q) {
        if ( q -> data < p -> data) {
            pre = q;
            q = q -> rightChild;
        }
        else if (q -> data > p -> data) {
            q = q -> leftChild;
        }
        else break;
    }
    return pre -> data;
}

```

```

int max(int a, int b) {
    return (a > b) ? a : b;
}

```

```

int height(struct Node *p) {
    if (p) return 1 + max(height(p -> leftChild), height(p -> rightChild));
    return 0;
}

```

```

struct Node * deletenode(struct Node *p, int val) {
    if (!p) return NULL;
    if ( !p -> leftChild && ! p -> rightChild) {
        if (p == rootNode) rootNode = NULL;
        free(p);
        return NULL;
    }
    if (val < p -> data) {
        p -> leftChild = deletenode(p -> leftChild, val);
    }
    else if (val > p -> data) {
        p -> rightChild = deletenode( p -> rightChild, val);
    }
    else {
        int temp;
        if (height(p -> leftChild) > height(p -> rightChild)) {
            temp = inorderPre(rootNode, p);

```

```

        p -> data = temp;
        p -> leftChild = deletenode(p -> leftChild, temp);
    }
    else {
        temp = inorderSuc(rootNode, p);
        p -> data = temp;
        p -> rightChild = deletenode(p -> rightChild, temp);
    }
}
return p;
}

```

```

void inOrder(struct Node *p) {
    if (p) {
        inOrder(p -> leftChild);
        printf("%d ", p -> data);
        inOrder(p -> rightChild);
    }
}

```

```

struct Node* searchNode(struct Node *rootNode, int val) {
    if (!rootNode) {
        return NULL;
    }
    if ( val < rootNode -> data) searchNode(rootNode -> leftChild, val);
    else if ( val > rootNode -> data) searchNode(rootNode -> rightChild, val);
    else return rootNode;
}

```

```

int main() {
    int rootVal;
    printf("Enter root node value: ");
    scanf("%d", &rootVal);
    rootNode = createNewNode(rootVal);
    int ch;
    do {
        printf("1. Insert\n2. Delete\n3. Search\n4. InOrder Traversal\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: { printf("Enter number to be inserted: ");
                int num;
                scanf("%d", &num);
                insertNode(num);
                break; }

```

```

    case 2: { printf("Enter number to be deleted: ");
              int num;
              scanf("%d", &num);
              deletenode(rootNode, num);
              printf("Deleted!\n");
              break; }
    case 3: { printf("Enter element to be searched: ");
              int num;
              scanf("%d", &num);
              struct Node *p = searchNode(rootNode, num);
              if (p) printf("Found!\n");
              else printf("Not found!\n");
              break; }
    case 4: { printf("Inorder traversal is: ");
              struct Node *p = rootNode;
              inOrder(p);
              printf("\n");
              break; }
    default: printf("Exiting!\n");
            break;
        }
    } while (ch < 5);
    return 0;
}

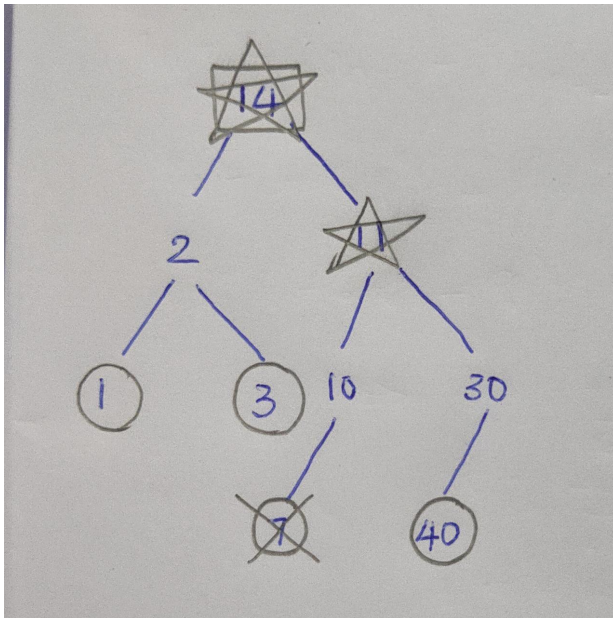
```

```
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab10_BST$ ./a.out
Enter root node value: 50
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 1
Enter number to be inserted: 24
Inserted!
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 4
Inorder traversal is: 24 50
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 1
Enter number to be inserted: 65
Inserted!
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 4
Inorder traversal is: 24 50 65
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 2
```

```
Enter number to be deleted: 50
Deleted!
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 4
Inorder traversal is: 24 65
1. Insert
2. Delete
3. Search
4. InOrder Traversal
Enter your choice: 6
Exiting!
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab10_BST$
```

## Pre Lab Questions:

1.



2.

	Inorder Predecessor	Inorder Successor
16	14	no successor
4	2	5
1	no predecessor	2
14	5	16

3. We can use a recursive approach. If the value of the current node is less than both node1 and node2, then, LCA lies in the right subtree. Now we should call the LCAfunction for the right subtree. If the value of the current node is greater than both node1 and node2, then LCA lies in the left subtree. Now we should call the recursive function for the left subtree. Otherwise, the current node is the LCA.



# Lab-11:

## Program1:

### Algorithm:

1. Start
2. Create a node structure which includes the members leftchild,rightchild,data(value of node) and height.
3. Create a structure for new node so that when user enter values memory is dynamically allocated for that particular node.
4. Define the functions height,nodeheight,balance factor,LL,LR,RR and RL rotations,min\_key,max\_key,inorderpredecessor,inordersuccessor.
5. Give a menu to the user:1)Insert 2)Delete 3)Search 4)Preorder 5)Inorder 6)Exit
6. If the user chooses choice 1 then ask the user to enter an element to insert and call the insert function by passing root and user entered value as arguments.
  1. If the root is null then make the user entered value as the root of the tree.
  2. If the user entered element is less than the root then call the insert function by passing left child of root and user entered value as arguments and assign it to left child of root.
  3. If the user entered element is greater than root value then call the insert function by passing right child of root and user entered value as arguments and assign it to right child of root.
  4. Out of these 3 conditions any of the above is true after checking those,calculate height of node using nodeheight() function.
  5. If balancefactor(p)==2 and user entered value is less than value of leftchild of the node then call LLRotation(node) to balance the tree.
  6. If balancefactor(p)==2 and user entered value is greater than value of leftchild of the node then call LRRotation(node) to balance the tree.
  - 7.If balancefactor(p)== -2 and user entered value is greater than value of rightchild of the node then call RRRotation(node) to balance the tree.
  8. If balancefactor(p)== -2 and user entered value is less than value of rightchild of the node then call RLRotation(node) to balance the tree.
7. If the user chooses choice 2 then ask the user to enter an element to delete and call the delete function by passing root and user entered value as arguments.
  1. First,search for the element entered by the user.If the entered element is not found then print entered element is not found and cannot be deleted.
  2. If the user entered element is found ,then check for following conditions
    - If the node to be deleted is leaf then delete directly and free the node.
    - If the node to be deleted has only one child then copy the child to the parent node and delete the child node.

- If the node to be deleted has two children then find height of left and right subtrees. If the height of left subtree is more then replace the node to be deleted with inorder predecessor
  - If the height of right subtree is more then replace the node to be deleted with inorder successor
  - If the heights of both subtrees are same then replace with inorder successor.
3. After deleting calculate height of node using nodeheight() function check the following conditions:
- If balancefactor(p)==2 and balance factor of leftchild of the node is 1 then call LLRotation(node) to balance the tree.
  - If balancefactor(p)==2 and balance factor of leftchild of the node is -1 then call LRRotation(node) to balance the tree.
  - If balancefactor(p)== -2 and balance factor of rightchild of the node is -1 then call RRRotation(node) to balance the tree.
  - If balancefactor(p)== -2 and balance factor of leftchild of the node is 1 then call RLRotation(node) to balance the tree.
  - If balancefactor(p)== 2 and balance factor of rightchild of the node is 0 then call LLRotation(node) to balance the tree.
  - If balancefactor(p)== -2 and balance factor of rightchild of the node is 0 then call RRRotation(node) to balance the tree.
8. If the user chooses choice 3 then ask the user to enter an element to search and call search function
- if the entered element is equal to root value, then print entered element is found.
  - if the entered element is less than the root value, then make node as leftchild and search
  - If the entered element is greater than the root value, then make node as right child and search
  - If the entered element is not found then print entered element is not found.
9. If the user chooses choice 4 print preorder of the tree (root,left,right) by calling preorder() function.
10. If the user chooses choice 5 print Inorder of the tree (left,root,right) which gives sorted order by calling Inorder() function.
11. If the user chooses choice 6 program is exited.
12. If the user other than the above mentioned choices display wrong choice.
13. Stop

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

```
struct Node {
    int data, height;
    struct Node *rightChild, *leftChild;
};
```

```
struct Node *rootNode;
```

```
struct Node *createNewNode(int val) {  
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));  
    newnode -> data = val;  
    newnode -> height = 0;  
    newnode -> rightChild = NULL;  
    newnode -> leftChild = NULL;  
    return newnode;  
}
```

```
int max(int a, int b) {  
    return (a > b) ? a : b;  
}
```

```
int findHeight(struct Node *p) {  
    if (p) {  
        return 1 + max(findHeight(p -> leftChild), findHeight(p -> rightChild));  
    }  
    return 0;  
}  
// return 1 + ( max(p -> leftChild ? p -> leftChild -> height : 0, p -> rightChild ? p ->  
rightChild -> height : 0))
```

```
int balancingFactor (struct Node *p) {  
    return findHeight(p ->leftChild) - findHeight(p ->rightChild);  
    //return hl - hr;  
}
```

```
int minKey(struct Node *node) {  
    struct Node *p = node;  
    while (p -> leftChild) {  
        p = p -> leftChild;  
    }  
    return p -> data;  
}
```

```
int maxKey(struct Node *node) {  
    struct Node *p = node;  
    while (p -> rightChild) {  
        p = p -> rightChild;  
    }  
    return p -> data;  
}
```

```

int inorderSuc(struct Node *rootNode, struct Node *p) {
    if ( p -> rightChild ) return minKey(p -> rightChild);
    struct Node *suc = NULL;
    struct Node *q = rootNode;
    while (q) {
        if ( q -> data > p -> data) {
            suc = q;
            q = q -> leftChild;
        }
        else if (q -> data < p -> data) {
            q = q -> rightChild;
        }
        else break;
    }
    return suc -> data;
}

```

```

int inorderPre(struct Node *rootNode, struct Node *p) {
    if ( p -> leftChild ) return maxKey(p -> rightChild);
    struct Node *pre = NULL;
    struct Node *q = rootNode;
    while (q) {
        if ( q -> data < p -> data) {
            pre = q;
            q = q -> rightChild;
        }
        else if (q -> data > p -> data) {
            q = q -> leftChild;
        }
        else break;
    }
    return pre -> data;
}

```

```

struct Node* LLrotation(struct Node *p) {
    struct Node * pl = p -> leftChild;
    struct Node *plr = pl -> rightChild;
    pl -> rightChild = p;
    p -> leftChild = plr;
    if (p == rootNode) rootNode = pl;
    p -> height = findHeight(p);
    pl -> height = findHeight(pl);
    return pl;
}

```

```

struct Node* LRrotation(struct Node *p) {
    struct Node *pl = p -> leftChild;
    struct Node *plr = pl -> rightChild;
    p -> leftChild = plr -> rightChild;
    pl -> rightChild = plr -> leftChild;
    plr -> rightChild = p;
    plr -> leftChild = pl;
    if ( p == rootNode) rootNode = plr;
    p -> height = findHeight(p);
    pl -> height = findHeight(pl);
    plr -> height = findHeight(plr);
    return plr;
}

```

```

struct Node* RRrotation(struct Node *p) {
    struct Node *pr = p -> rightChild;
    struct Node *prl = pr -> leftChild;
    pr -> leftChild = p;
    p -> rightChild = prl;
    if (p == rootNode) rootNode = pr;
    p -> height = findHeight(p);
    pr -> height = findHeight(pr);
    return pr;
}

```

```

struct Node* RLrotation(struct Node *p) {
    struct Node *pr = p -> rightChild;
    struct Node *prl = pr -> leftChild;
    pr -> leftChild = prl -> rightChild;
    p -> rightChild = prl -> leftChild;
    prl -> leftChild = p;
    prl -> rightChild = pr;
    if (p == rootNode) rootNode = prl;
    p -> height = findHeight(p);
    pr -> height = findHeight(pr);
    prl -> height = findHeight(prl);
    return prl;
}

```

```

struct Node* insertNode(struct Node *p, int key) {
    if (!p) return createNewNode(key);
    if ( key < p -> data) {
        p -> leftChild = insertNode( p -> leftChild, key);
    }
}

```

```

}
else {
    p -> rightChild = insertNode(p -> rightChild, key);
}
p -> height = findHeight(p);
// bf of p == 2, bf of lchild 1 -> ll rotation
if (balancingFactor(p) == 2 && balancingFactor(p -> leftChild) == 1) {
    return LLrotation(p);
}
// 2, l = -1, -> lr rotation
else if (balancingFactor(p) == 2 && balancingFactor(p -> leftChild) == -1) {
    return LRrotation(p);
}
// -2, r = -1, rr rotation
else if (balancingFactor(p) == -2 && balancingFactor(p -> rightChild) == -1) {
    return RRrotation(p);
}
// -2, r = 1 -> rl rotation
else if (balancingFactor(p) == -2 && balancingFactor(p -> rightChild) == 1) {
    return RLrotation(p);
}
//change height of the three nodes in respective rotation
//if imbalance is at root, don't forget to point rootNode to correct node at the last
return p;
}

```

```

struct Node* deleteNode(struct Node *p, int key) {
    if (!p) return NULL;
    if (!p-> leftChild && !p-> rightChild) {
        if (p == rootNode) rootNode = NULL;
        free(p);
        return NULL;
    }
    if (key < p-> data) {
        p -> leftChild = deleteNode(p -> leftChild, key);
    }
    else if (key > p-> data) {
        p -> rightChild = deleteNode(p -> rightChild, key);
    }
    else {
        int temp;
        if (findHeight(p -> leftChild) > findHeight(p -> rightChild)) {
            temp = inorderPre(rootNode, p);
            p -> data = temp;
        }
    }
}

```

```

        p -> leftChild = deleteNode(p -> leftChild, temp);
    }
    else {
        temp = inorderSuc(rootNode, p);
        p -> data = temp;
        p -> rightChild = deleteNode(p -> rightChild, temp);
    }
}
p -> height = findHeight(p);
if ( balancingFactor(p) == 2 && balancingFactor(p -> leftChild) >= 0) {
    return LLrotation(p);
}
if (balancingFactor(p) == 2 && balancingFactor(p) == -1 ) {
    return LRrotation(p);
}
if (balancingFactor(p) == -2 && balancingFactor(p -> rightChild) <= 0) {
    return RRrotation(p);
}
if (balancingFactor(p) == -1 && balancingFactor(p -> rightChild) == 1) {
    RLrotation(p);
}
return p;
}

```

```

void preOrder(struct Node * p) {
    if (p) {
        printf("%d ", p -> data);
        preOrder(p -> leftChild);
        preOrder(p -> rightChild);
    }
} //preorder traversal using recursion

```

```

void postOrder(struct Node * p) {
    if (p) {
        postOrder(p -> leftChild);
        postOrder(p -> rightChild);
        printf("%d ", p -> data);
    }
} //postOrder traversal using recursion

```

```

void inOrder(struct Node * p) {
    if (p) {
        inOrder(p -> leftChild);
        printf("%d ", p -> data);
    }
}

```

```

        inOrder(p -> rightChild);
    }
} //inorder traversal using recursion

int main() {
    int rootVal;
    printf("Enter root node value: ");
    scanf("%d", &rootVal);
    rootNode = createNewNode(rootVal);
    int ch;
    do {
        printf("1. Insert\n2. Delete\n3. PreOrder Traversal\n4. InOrder Traversal\n5. PostOrder Traversal\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: { printf("Enter number to be inserted: ");
                int num;
                scanf("%d", &num);
                insertNode(rootNode, num);
                break; }
            case 2: { printf("Enter number to be deleted: ");
                int num;
                scanf("%d", &num);
                deleteNode(rootNode, num);
                printf("Deleted!\n");
                break; }
            case 3: { printf("PreOrder traversal is: ");
                struct Node *p = rootNode;
                preOrder(p);
                printf("\n");
                break; }
            case 4: { printf("Inorder traversal is: ");
                struct Node *p = rootNode;
                inOrder(p);
                printf("\n");
                break; }
            case 5: { printf("PostOrder traversal is: ");
                struct Node *p = rootNode;
                postOrder(p);
                printf("\n");
                break; }
            default: printf("Exiting!\n");
                break;
        }
    }
}

```



```
} while (ch < 6);  
return 0;  
}
```

## Pre Lab Questions:

1. An AVL (Adelson, Velski & Landis) tree is a Binary search tree with some extra constraints. It is also called the height balanced binary search tree. In an AVL tree, if we consider a node, the height of its left and right subtree must not differ by more than 1 at any time. In case, they differ, rotation operations are performed on the tree to make the tree balanced.
2. We can first sort the set of numbers. Select the mid element as the root. And keep inserting one element from the right of mid one from the left and keep inserting until all the elements are inserted.
3. If the binary tree is not height balanced it becomes skewed to one side and all the operations like search, insert, delete would have a complexity of  $O(\text{height})$  whereas if they are height balanced, they would take only  $O(\log(n))$ .
4.  $\log(p)$
5. The AVL property can be restored by performing rotations. The order/type of rotation can be determined by observing the way we traverse from root in order to insert the new node. If we move right and right from root, we perform RR rotation. Similarly LL, LR, RL.

# Lab-12:

## Program1:

### Algorithm:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

```
#include <stdio.h>
#include<stdlib.h>
int visited[8];
struct queue
{
    int space[100],front,rare,size;
}*q;

int isFull(struct queue *que)
{
    if(que->rare==que->size-1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int isEmpty(struct queue *que)
{
    if(que->front==-1 && que->rare==-1)
        return 1;
    else
        return 0;
}

void enqueue(int n)
{
    if(!isFull(q))
    {
```

```

        if(q->rare== -1 && q->front== -1)
        {
            q->rare+=1;
            q->front+=1;
            q->space[q->rare]=n;
        }
        else
        {
            q->rare=q->rare+1;
            q->space[q->rare]=n;
        }
    }
}

```

```

}
int dequeue()
{
    int n;
    if(!isEmpty(q))
    {
        if(q->rare==q->front)
        {
            n=q->space[q->front];
            q->rare=-1;
            q->front=-1;
            return n;
        }
        else
        {
            n=q->space[q->front];
            q->front=q->front+1;
            return n;
        }
    }
    else
    {
        return -1;
    }
}

```

```

void BFS(int arr[][8],int start,int n)
{
    int i=start;int u;
    printf("%d ",i);
    visited[i]=1;
    enqueue(i);
}

```

```

while(!isEmpty(q))
{
    u=dequeue();
    for(int i=1;i<n;i++)
    {
        if(arr[u][i]==1 && visited[i]==0)
        {
            printf("%d ",i);
            visited[i]=1;
            enqueue(i);
        }
    }
}
}

```

```

int main()
{
    int n;
    printf("Enter number of vertices\n");
    scanf("%d",&n);
    int arr[n][n];
    printf("Enter the adjacency matrix\n");
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            scanf("%d",&arr[i][j]);
        }
    }
    q=(struct queue*)malloc(sizeof(struct queue));
    q->front=-1;
    q->rare=-1;
    q->size=100;
    BFS(arr,1,8);
    return 0;
}

```

```

Enter number of vertices
4
Enter the adjacency matrix
0 1 1 0
1 0 0 0
1 0 0 1
0 1 1 0\
1 3 5 6 2

```

```

Enter number of vertices
4
Enter the adjacency matrix
0 1 1 0
1 0 0 0
1 0 0 1
0 0 1 0
1 3 6

```

```

Enter number of vertices
0 1 1 0
Enter the adjacency matrix
1

```

## Program2:

### Algorithm:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

```

#include<stdio.h>
void DFS(int G[][4], int start,int n)
{
    static int visited[4]={0};
    int j;
    if(visited[start]==0)
    {
        printf("%d ",start);
        visited[start]=1;
        for(j=1;j<n;j++)
        {
            if(G[start][j]==1 && visited[j]==0)
                DFS(G,j,n);
        }
    }
}
int main()
{
    int n=4;
    //printf("Enter n (size of matrix) value\n");

```

```
//scanf("%d",&n);
int G[n][n];
int i,j;
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        printf("Enter value of %d %d \n",i,j);
        scanf("%d",&G[i][j]);
    }
    printf("\n");
}
printf("DFS of Graph\n");
DFS(G,0,n);
return 0;
}
```

```
Enter value of 0 0
0
Enter value of 0 1
1
Enter value of 0 2
1

Enter value of 1 0
1
Enter value of 1 1
0
Enter value of 1 2
0

Enter value of 2 0
1
Enter value of 2 1
0
Enter value of 2 2
0

DFS of Graph
0 1 2
```

```
Enter value of 0 0
0
Enter value of 0 1
1
Enter value of 0 2
1
Enter value of 0 3
0

Enter value of 1 0
1
Enter value of 1 1
0
Enter value of 1 2
0
Enter value of 1 3
0

Enter value of 2 0
1
Enter value of 2 1
0
Enter value of 2 2
0
Enter value of 2 3
1

Enter value of 3 0
0
Enter value of 3 1
0
Enter value of 3 2
1
Enter value of 3 3
0

DFS of Graph
0 1 2 3
```

```
Enter value of 0 0
0
Enter value of 0 1
1
Enter value of 0 2
0

Enter value of 1 0
1
Enter value of 1 1
0
Enter value of 1 2
1

Enter value of 2 0
0
Enter value of 2 1
1
Enter value of 2 2
0

DFS of Graph
0 1 2
```

## Pre Lab Questions:

1. BFS uses Queue data structure whereas DFS uses Stack data structures. In BFS, we explore all the adjacent nodes at every node and then move to another node whereas in DFS, we go from one node to another till we reach null and then explore the nodes.
2. A spanning tree is a subgraph that contains all the vertices of a graph.
3. No they don't give the same output unless the graph is a tree.
4. In Kruskal's algorithms, we go edge by edge choosing the edge of lowest cost. Its time complexity is  $O(e \log(e))$ . In Prim's algorithm, we begin with a vertex and move accordingly looking at the costs. It has  $O(n^2)$  time complexity.
5. Graphs are used to define the flow of computation. Graphs are used to represent networks of communication. Graphs are used to represent data organization. Graph transformation systems work on rule-based in-memory manipulation of graphs.



# Lab-13:

## Program1:

### Algorithm:

1. Build a max heap from the input data. (Sort in ascending order)
2. At this point, the largest item is stored at the root of the heap. Swap it with the last item of the heap followed by reducing the size of heap by 1.
3. Finally, heapify the root of tree (i.e build a max heap from remaining elements)
4. Repeat above steps while size of heap is greater than 1.
5. Stop

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

```
#include <math.h>
```

```
void swap(int heap[10], int index1, int index2) {  
    int temp = heap[index1];  
    heap[index1] = heap[index2];  
    heap[index2] = temp;  
}
```

```
void minHeapify(int heap[10], int currentNode, int n) {  
    int lChild = 2 * currentNode, rChild = 2 * currentNode + 1, smallest = currentNode;  
    if ( lChild <= n && heap[smallest] > heap[lChild] ) smallest = lChild;  
    if ( rChild <= n && heap[smallest] > heap[rChild] ) smallest = rChild;  
    if ( smallest != currentNode ) {  
        swap(heap, currentNode, smallest);  
        minHeapify(heap, smallest, n);  
    }  
}
```

```
void buildMinHeap(int heap[10], int n) {  
    for ( int i = floor( n / 2 ) ; i >= 1; i -- ) {  
        minHeapify(heap, i, n);  
    }  
}
```

```
void heapSort(int heap[10], int n) {  
    buildMinHeap(heap, n);  
    printf("The heap before sorting, i.e maxHeap, is: ");  
    for (int i = 1; i <= n; i ++ ) {  
        printf("%d ", heap[i]);  
    }
```

```

    }
    for (int i = n; i > 0; i--)
    {
        swap(heap, i, 1);
        int j = 1, index;
        do {
            index = (2 * j);
            if (heap[index] > heap[index + 1] && index < (i - 1))
                index ++;
            if (heap[j] > heap[index] && index < i)
                swap(heap, j, index);
            j = index;
        } while (index < i);
    }
}

int main() {
    int n;
    printf("Enter n value (number of nodes of minHeap): ");
    scanf("%d", &n);
    int heap[100] = {0};
    for (int i = 1; i <= n; i++) {
        scanf("%d", &heap[i]);
    }
    heapSort(heap, n);
    printf("The heap after sorting (descending order) is: ");
    for (int i = 1; i <= n; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");
    return 0;
}

```

```

portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$ cc minHeap2.c -lm
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$ ./a.out
Enter n value (number of nodes of minHeap): 8
5
12
64
1
37
90
91
97
The heap after sorting (descending order) is: 97 91 90 64 37 12 5 1
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$

```

## Program2:

### Algorithm:

6. Build a max heap from the input data. (Sort in ascending order)
7. At this point, the largest item is stored at the root of the heap. Swap it with the last item of the heap followed by reducing the size of heap by 1.
8. Finally, heapify the root of tree (i.e build a max heap from remaining elements)
9. Repeat above steps while size of heap is greater than 1.
10. Stop

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

```
void swap(int heap[10], int index1, int index2) {
    int temp = heap[index1];
    heap[index1] = heap[index2];
    heap[index2] = temp;
}
```

```
void maxHeapify(int heap[10], int currentNode, int n) {
    int lChild = 2 * currentNode, rChild = 2 * currentNode + 1, largest = currentNode;
    if ( lChild <= n && heap[largest] < heap[lChild]) largest = lChild;
    if ( rChild <= n && heap[largest] < heap[rChild]) largest = rChild;
    if ( largest != currentNode) {
        swap(heap, currentNode, largest);
        maxHeapify(heap, largest, n);
    }
}
```

```
void buildMaxHeap(int heap[10], int n) {
    for ( int i = floor( n / 2 ) ; i >= 1; i --) {
        maxHeapify(heap, i, n);
    }
}
```

```
void heapSort(int heap[10], int n) {
    buildMaxHeap(heap, n);
    printf("The heap before sorting, i.e maxHeap, is: ");
    for (int i = 1; i <= n; i ++ ) {
        printf("%d ", heap[i]);
    }
    printf("\n");
}
```

```

        for (int currentN = n; currentN >= 1; currentN --) {
            swap(heap, currentN, 1);
            maxHeapify(heap, 1, currentN);
        }
    }

int main() {
    int n;
    printf("Enter n value (number of nodes of maxHeap): ");
    scanf("%d", &n);
    int heap[100] = {0};
    for (int i = 1; i <= n; i++) {
        scanf("%d", &heap[i]);
    }
    heapSort(heap, n);
    printf("The heap after sorting (ascending order) is: ");
    for (int i = 1; i <= n; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");
    return 0;
}

```

```

portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$ cc heapSort.c -lm
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$ ./a.out
Enter n value (number of nodes of maxHeap): 8
8
6
7
1
5
3
2
4
The heap before sorting, i.e maxHeap, is: 8 6 7 1 5 3 2 4
The heap after sorting (ascending order) is: 1 2 3 4 5 6 7 8
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$ cc heapSort.c -lm
portkey@Taruni:/mnt/c/Users/Taruni/Desktop/DS/lab12_heaps$

```

## Pre lab questions

1. Both minHeaps and maxHeaps are complete binary trees with the difference being, in minHeap, every node is  $\leq$  its lChild and rChild, whereas in maxHeap, every node is  $\geq$  its lChild and rChild.
2. Heapify is a lot better than normal heap because in normal heap we make unnecessary extra comparisons and it takes more complexity. Whereas heapify is efficient and uses  $\log(n)$  complexity.
3. There are mainly 4 operations we want from a priority queue: 1. Insert  $\rightarrow$  To insert a new element in the queue. 2. Maximum/Minimum  $\rightarrow$  To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively. 3. Extract Maximum/Minimum  $\rightarrow$  To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively. 4. Increase/Decrease key  $\rightarrow$  To increase or decrease key of any element in the queue. A priority queue stores its data in a specific order according to the keys of the elements. So, inserting a new element must go in a place according to the specified order. This is what the insert operation does. The entire point of the priority queue is to get the data according to the key of the data and the Maximum/Minimum and Extract Maximum/Minimum does this for us. We may also face a situation in which we need to change the key of an element, so Increase/Decrease key is used to do that.
4. Heap sort is very useful in implementing priority queues and also used in many algorithms like Prim's algorithm, Dijkstra's algorithm.
5. Average case complexity:  $O(n(\log(n)))$  Worst case complexity:  $O(n(\log(n)))$