Ex. No: 1	TIME COMPLEXITY ANALYSIS
17 / 02 / 22	

AIM:

To analyse the time complexity of different (i) sorting algorithms to find the ‘k’th largest element from a group of ‘N’ numbers (ii) algorithms to generate legal random permutations of ‘n’ integers (legal permutations are when numbers generated are less than or equal to ‘n’ and each number occurs only once) (iii) algorithms to evaluate polynomials.

THEORY:

Time complexity is the analysis of the runtime of an algorithm irrespective of the hardware used. It depends on the input size and is designed as a function of input size. It captures the growth rate when the input size is increased. We may calculate worst-case, average-case or best-case time complexity.

Categories of algorithms depending on their time complexity:

- Constant time algorithm: running time complexity given as O(1).
- Linear time algorithm: running time complexity given as O(n).
- Logarithmic time algorithm: running time complexity given as $O(\log_2(n))$.
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$.
- Exponential time algorithm: running time complexity given as $O(2^n)$.

Program 1:

```
/*
Consider the problem of finding 'k'th largest element from a group of 'N'
numbers.
Sort the numbers in descending order and retrieve the ‘k’th element.
Add the logic to calculate running time to your code. Calculate the actual
running times for various values of 'N' and compare results obtained.
*/

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

#define BILLION 1000000000.0
#define UPPER_BOUND 1000
#define PRINT_ARR for (i = 0; i < numOfInputs; i++) {printf("%d\n",
numArr[i]);}

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main() {
    // Variable to store runtime.
    struct timespec start, end;

    // Input
    printf("*** Program to Find 'k'th Largest Element *** \n");

    int numOfInputs;
    printf("\nEnter number of inputs: ");
    scanf("%d", &numOfInputs);

    int numArr[numOfInputs];
    int i, j;

    // Generate number array (+ Progress Bar)
    printf("Generating random number array with %d elements. \n",
numOfInputs);
    printf("[");    // Progress Bar Start

    for (i = 0; i < numOfInputs; i++) {
        numArr[i] = (rand()%(UPPER_BOUND + 1));    // (100 + 1) = 'U'
(Inclusive Upper Bound) + 1 -> Range: '0' to 'U'

        // When 'i' increments by '1/30'th of input size -> Print 1 bar
        if (i % (numOfInputs / 30) == 0) {
            printf("|");
        }
    }

    printf("] - Completed. \n");

    int k;    // 'k'th largest element to search
    printf("\nEnter 'k' value: ");
    scanf("%d", &k);

    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    // Selection Sort
    int max, maxIndex;

    for (i = 0; i < numOfInputs; i++) {
        max = numArr[i];
        maxIndex = i;

        for (j = i; j < numOfInputs; j++) {
            if (numArr[j] > max) {
                max = numArr[j];
                maxIndex = j;
            }
        }
        swap(&numArr[i], &numArr[maxIndex]);
    }

    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + (end.tv_nsec -
start.tv_nsec) / BILLION;
    }

    // Note: ((float)(end.tv_nsec - start.tv_nsec)) / BILLION -> 'float'
type conversion not required if 'BILLION' is 'float' type

    // Output
    printf("'k'th largest element is %d. \n", numArr[k-1]);
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: 'Lf' -> Long
double, 'lf' -> Double
    printf("(Sort + search algorithm) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
}

/*
start.tv_sec -> Timestamp at Start [Using system clock] (in s)
start.tv_nsec -> Timestamp of nanoseconds within the current second (in ns)
end.tv_sec -> Timestamp at End [Using system clock] (in s)
end.tv_nsec -> Timestamp of nanoseconds within the current second (in ns).

E.g., If Start -> 167876541.78770000 s & End- > 167876542.12200000 s

Then,
    start.tv_sec = 167876541 s
    start.tv_nsec = 78770000 ns
    end.tv_sec = 167876542 s
    end.tv_nsec = 12200000 ns
```

```
Note: 'end.tv_nsec - start.tv_nsec' < 0

Result,
=  ((end.tv_sec  -  start.tv_sec)  -  1)  +  ((end.tv_nsec  -  start.tv_nsec)  /
BILLION)
= ((2) - 1) + ((-66570000) / BILLION)
= (1) + (-0.66570000)
= 0.33430000 s

[
Note: If 'end.tv_nsec - start.tv_nsec' >= 0,

Result,
= ((end.tv_sec - start.tv_sec) + ((end.tv_nsec - start.tv_nsec) / BILLION)
]

Therefore,
    if 'end.tv_nsec - start.tv_nsec < 0', subtract '1' from 'end.tv_sec -
start.tv_sec'.


Types of time representation:
i) Floating-Point: 0.02 s (Above representation)
ii) Integer: 20000000 ns

For Integer Representation:

if (end.tv_nsec - start.tv_nsec > 0) {
    runTime  =  (end.tv_sec  -  start.tv_sec)  *  BILLION  +  (end.tv_nsec  -
start.tv_nsec);
}
else {
    runTime = (end.tv_sec - start.tv_sec - 1) * BILLION + (end.tv_nsec -
start.tv_nsec);
}

Note: Change 's' to 'ns' in output. ('ns' -> 'nanosecond')
*/
```

Program 2:

```c
/*
Consider the problem of finding 'k'th largest element from a group of 'N'
numbers.
Place the first ‘k’ elements in an array and sort it in descending order.
Read the remaining elements one by one and compare with ‘k’th element in the
array.
If the element is smaller, ignore. Otherwise, place the element in the correct
spot replacing the existing element.
When the comparisons end, the ‘k’th element in the array will be the ‘k’th
largest element.
Add the logic to calculate running time to your code. Calculate the actual
running times for various values of 'N' and compare results obtained.
*/

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

#define BILLION 1000000000.0
#define UPPER_BOUND 1000
#define PRINT_NUMARR for (i = 0; i < numOfInputs; i++) {printf("%d\n",
numArr[i]); printf("\n");

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main() {
    // Variable to store runtime.
    struct timespec start, end;

    // Input
    printf("*** Program to Find 'k'th Largest Element *** \n");

    int numOfInputs;
    printf("\nEnter number of inputs: ");
    scanf("%d", &numOfInputs);

    int numArr[numOfInputs];
    int i, j;

    // Generate number array (+ Progress Bar)
    printf("Generating random number array with %d elements. \n",
numOfInputs);
    printf("[");    // Progress Bar Start

    for (i = 0; i < numOfInputs; i++) {
        numArr[i] = (rand()%(UPPER_BOUND+1));    // (100 + 1) = 'U'
(Inclusive Upper Bound) + 1 -> Range: '0' to 'U'

        // When 'i' increments by '1/30'th of input size -> Print 1 bar
        if (i % (numOfInputs / 30) == 0) {
            printf("|");
        }
    }

    printf("] - Completed. \n");

    int k;    // 'k'th largest element to search
    printf("\nEnter 'k' value: ");
    scanf("%d", &k);

    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

     // Placing first 'k' elements in 'arr'
     int arr[k];

     for (i = 0; i < k; i++) {
         arr[i] = numArr[i];
     }

    // Selection sort (Array with first 'k' elements) - in descending order
    int max, maxIndex;

    for (i = 0; i < k; i++) {
        max = arr[i];
        maxIndex = i;
        for (j = i; j < k; j++) {
            if (arr[j] > max) {
                max = arr[j];
                maxIndex = j;
            }
        }
        swap(&arr[i], &arr[maxIndex]);
    }

    int key;

    for (i = k; i < numOfInputs; i++) {
        if (numArr[i] < arr[k-1]) {
            continue;
        }
        else {
            key = numArr[i];
            j = k - 1;
            arr[j] = numArr[i];
            while (j >= 0 && key > arr[j-1]) {
                arr[j] = arr[j-1];
                j--;
            }
            arr[j] = key;
        }
    }

    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }

    // Output
    printf("'k'th largest element is %d. \n", arr[k-1]);
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: 'Lf' -> Long
double, 'lf' -> Double
    printf("(Place & sort first 'k' elements in an array + insertion sort
algorithm) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
}
```

Program 3:

```
/*
A set of 'n' integers need to be generated in a random permutation each time
the code is run. Examples of legal permutations are: {4,2,1,3,5} and
{2,3,1,5,4} for n=5. Each number has to occur only once and all numbers less
than or equal to ‘n’ should occur.

You may use the rand() or srand() function for random number generation.

Consider the following algorithm:
Fill the array a from a[0] to a[n-1] as follows: To fill a[i], generate
random numbers in the range until you get one that is not already in a[0],
a[1], ..., a[i-1].
*/

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

#define BILLION 1000000000.0
#define PRINT_ARR for (int i = 0; i < n; i++) printf("%d ", intArr[i]);
printf("\n");

int main() {
    // Input
    int n;    // 'n' -> Number of integers & maximum value of an integer
    printf("*** 'n' Integers Permutation Generator *** \n");
    printf("\nEnter the value of 'n': ");
    scanf("%d", &n);

    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    printf("\nGenerating a random array with a legal permutation. \n");
    printf("[");

    int intArr[n];
    int i, j;

    intArr[0] = (rand() % n) + 1;    // Generate random numbers from '1' to
'n' (upper & lower bound inclusive).

    for (i = 1; i < n; i++) {
        intArr[i] = (rand() % n) + 1;
        for (j = 0; j < i; j++) {
            if (intArr[j] == intArr[i]) {
                intArr[i] = rand()%(n+1);
                j = 0;
            }
        }
        if (i % (n/10) == 0) {
            printf("|");
        }
    }

    printf("] - Completed. \n");

    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }

    // Output
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: 'Lf' -> Long
double, 'lf' -> Double
    printf("(Search array + insert algorithm) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
}
```

Program 4:

```c
/*
A set of 'n' integers need to be generated in a random permutation each time
the code is run. Examples of legal permutations are: {4,2,1,3,5} and
{2,3,1,5,4} for n = 5. Each number has to occur only once and all numbers
less than or equal to ‘n’ should occur.

rand() or srand() function may be used for random number generation.

Consider the following algorithm:
(i) Fill the array a from a[0] to a[n-1] as follows: To fill a[i], generate
random numbers in the range until you get one that is not already in a[0],
a[1], ..., a[i-1].

(ii) Same as algorithm (i) but keep an extra array called 'used'.
When a random number is first put into the array ‘a’, set 'used[ran] = true'.
*/

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

#define BILLION 1000000000.0
#define PRINT_ARR for (int i = 0; i < n; i++) printf("%d ", intArr[i]);
printf("\n");

int main() {
    // Input
    int n;    // 'n' -> Number of integers & maximum value of an integer
    printf("*** 'n' Integers Permutation Generator *** \n");
    printf("\nEnter the value of 'n': ");
    scanf("%d", &n);

    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    printf("\nGenerating a random array with a legal permutation. \n");
    printf("[");

    int intArr[n];
    int i;

    // Extra 'used' array
    int used[n];

    for (i = 0; i < n; i++) {
        used[i] = 0;    // Set all values to '0' (i.e.) 'false'
    }

    // 'rand()' -> will generate same sequence of random numbers every time
(It assumes 'srand(1)' (i.e.) seed = 1 )
```

```
    // 'srand()' -> will generate different sequence of random numbers
(Seed for random number depends on current time)
    srand(time(0));

    for (i = 0; i < n; i++) {
        intArr[i] = (rand() % n) + 1;    // Generate random numbers from
'1' to 'n' (upper & lower bound inclusive).
        if (used[intArr[i]] == 1) {
            i--;
            continue;
        }
        used[intArr[i]] = 1;    // Set used[<random_number>] = 1 (i.e.)
'false'
        if (i % (n/10) == 0) {
            printf("|");
        }
    }

    printf("] - Completed. \n");
    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // To view array, un-comment 'PRINT_ARR' below
    // PRINT_ARR

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }

    // Output
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: 'Lf' -> Long
double, 'lf' -> Double
    printf("(Search index + insert algorithm) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
```

Program 5:

```
/*
A set of 'n' integers need to be generated in a random permutation each time
the code is run. Examples of legal permutations are: {4,2,1,3,5} and
{2,3,1,5,4} for n = 5. Each number has to occur only once and all numbers
less than or equal to n should occur.

rand() or srand() function may be used for random number generation.

Consider the following algorithm:
Fill the array such that a[i] = i+1. Then swap each number with another
number picked from a random index.
*/

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

#define BILLION 1000000000.0
#define PRINT_ARR for (int i = 0; i < n; i++) printf("%d ", intArr[i]);
printf("\n");

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main() {
    // Input
    int n;    // 'n' -> Number of integers & maximum value of an integer
    printf("*** 'n' Integers Permutation Generator *** \n");
    printf("\nEnter the value of 'n': ");
    scanf("%d", &n);

    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    printf("\nGenerating a random array with a legal permutation. \n");
    printf("[");

    int intArr[n];
    int i;

    for (i = 0; i < n; i++) {
        intArr[i] = i + 1;
    }

    // 'rand()' -> will generate same sequence of random numbers every time
(It assumes 'srand(1)' (i.e.) seed = 1 )
```

```
    // 'srand()' -> will generate different sequence of random numbers
(Seed for random number depends on current time)
    srand(time(0));

    int randInt;

    for (i = 0; i < n; i++) {
        randInt = (rand() % n) + 1;    // Random number in range '1' to 'n'
(upper & lower bound inclusive).
        swap(&intArr[i], &intArr[randInt]);

        if (i % (n/10) == 0) {
            printf("|");
        }
    }

    printf("] - Completed. \n");

    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // To view array, un-comment 'PRINT_ARR' below
    // PRINT_ARR

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec)) / BILLION;
    }

    // Output
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: '%Lf' -> Long
double, '%lf' -> Double, '%d' -> Integer
    printf("(Fill array + swap algorithm) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
}
```

Program 6:

```
/*
C program to calculate the time complexity of the algorithm to calculate
f(x) = ∑ [i = 0 to 'n'] (ai)*(x^i).
*/

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

#define BILLION 1000000000
#define UPPER_LIMIT 10

float power(float num, int pow) {
    float prod = 1;
    int i;

    for (i = 0; i < pow; i++) {
        prod *= num;
    }

    return prod;
}

int main() {
    // Input
    int n;    // 'n' -> Order of polynomial
    printf("*** f(x) = ∑ [i = 0 to 'n'] (ai)*(x^i) Calculator *** \n");
    printf("\nEnter the value of 'n' (Degree of polynomial): ");
    scanf("%d", &n);

    int i;
    float x, sum = 0;
    float constant[n];
    printf("Enter (1) to generate random constants and (2) to enter values:
");
    int choice;
    scanf("%d", &choice);

    while (1) {
        if (choice == 1) {
            for (i = 0; i <= n ; i++) {
                constant[i] = ((rand() % UPPER_LIMIT) + 1);
            }
            break;
        }
        else if (choice == 2) {
            for (i = 0; i <= n ; i++) {
                printf("Enter co-efficient of x^%d: ", i);
                scanf("%f", &constant[i]);
            }
```

```c
            break;
        }
        else {
            printf("\nPlease enter a valid option. \n");
            printf("Enter (1) to generate random constants and (2) to enter
values: ");
            scanf("%d", &choice);
        }
    }

    printf("\nEnter value of 'x': ");
    scanf("%f", &x);

    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    for (i = 0; i <= n ; i++) {
        sum = sum + (constant[i] * power(x, i));
    }

    printf("f(%.1f) = %f \n", x, sum);
    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // To view array, un-comment 'PRINT_ARR' below
    // PRINT_ARR

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }

    // Output
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: '%Lf' -> Long
double, '%lf' -> Double, '%d' -> Integer
    printf("(With user-defined function to calculate power) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */
    return 0;
}
```

Program 7:

```c
/*
C program to calculate the time complexity of the algorithm to calculate
f(x) = ∑ [i = 0 to 'n'] (ai)*(x^i).
Consider this algorithm for evaluation of the above function:
    poly = 0;
    for(i = n; i >= 0; i--)
        poly = x * poly + a[i];
*/

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

#define BILLION 1000000000
#define UPPER_LIMIT 10

int main() {
    // Input
    int n;    // 'n' -> Order of polynomial
    printf("*** f(x) = ∑ [i = 0 to 'n'] (ai)*(x^i) Calculator *** \n");
    printf("\nEnter the value of 'n' (Degree of polynomial): ");
    scanf("%d", &n);

    int i;
    float x;
    float constant[n];

    printf("Enter (1) to generate random constants and (2) to enter values:
");

    int choice;
    scanf("%d", &choice);

    while (1) {
        if (choice == 1) {
            for (i = 0; i <= n ; i++) {
                constant[i] = ((rand() % UPPER_LIMIT) + 1);
            }
            break;
        }
        else if (choice == 2) {
            for (i = 0; i <= n ; i++) {
                printf("Enter co-efficient of x^%d: ", i);
                scanf("%f", &constant[i]);
            }
            break;
        }
        else {
            printf("\nPlease enter a valid option. \n");
            printf("Enter (1) to generate random constants and (2) to enter
values: ");
            scanf("%d", &choice);
        }
    }

    printf("\nEnter value of 'x': ");
    scanf("%f", &x);

    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);    // Start Clock

    float poly = 0;

    for (i = n; i >= 0; i--) {
        poly = x * poly + constant[i];
    }

    printf("f(%.1f) = %f \n", x, poly);

    clock_gettime(CLOCK_REALTIME, &end);    // End Clock

    // To view array, un-comment 'PRINT_ARR' below
    // PRINT_ARR

    // Calculating Runtime
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }

    // Output
    printf("\nRuntime is %Lf s. \n", runTime);    // Note: '%Lf' -> Long
double, '%lf' -> Double, '%d' -> Integer
    printf("(Without power function) \n");

    /*
    // Cross-verification of time values
    printf("'start.tv_sec' = %ld s\n", start.tv_sec);
    printf("'start.tv_nsec' = %ld ns\n", start.tv_nsec);
    printf("'end.tv_sec' = %ld s\n", end.tv_sec);
    printf("'end.tv_nsec' = %ld ns\n", end.tv_nsec);
    */

    return 0;
}
```

RESULT:

Ex. No: 2	ARRAY IMPLEMENTATION OF LISTS
24 / 02 / 22	

AIM:

To implement List Abstract Data Type (ADT) using arrays.

THEORY:

A list is an abstract data type in C with ‘n’ elements – ‘a_1, a_2, a_3, …, a_n’. The position of element ‘a_i’ in the list is ‘i’. List operations include printing, searching for an element or ‘kth’ element, insertion and deletion. Running times for insertions and deletions is slow and the list size must be known in advance when arrays are used to implement lists.

The worst-case time complexities of operations are:

- Printing – O(n).
- Searching for element – O(n).
- Searching for ‘kth’ element – O(C) (‘C’ is a constant).
- Insertion – O(n).
- Deletion – O(n).

In array implementation, all elements to the right of the deleted element must be shifted to the left once (to delete an element), and all elements to the right of specified index, including element at index, must be shifted to the right to create space for element to be inserted.

Program 1:

```
/*
Write a program to implement list operations using arrays. The list
operations to be implemented are
a. Insertion
b. Deletion
c. Search
d. Display list.
The program should be menu-driven which allows user to choose any operation
at any point in time. The user may also choose to exit the program.
*/

#include <stdio.h>

int main() {
    int option = 1;

    // Print menu
    printf("*** LIST IMPLEMENTATION *** \n");
    printf("\n** OPERATIONS MENU ** \n");
    printf("1. Insert Element. \n");    // Insert element at specified
index
    printf("2. Delete Element Using Index. \n");    // Delete element at
given index
    printf("3. Delete Element. \n");    // Delete first occurrence element
    printf("4. Search Element. \n");    // Search for first occurrence of
element
    printf("5. Display List. \n");    // Display entire list
    printf("0. Exit. \n");

    int lst[100] = {};    // Initialise all elements to zero
    int lst_size = 0;    // Set list size to '0'
    int elt, ind;
    int i, j, flag;

    while (option != 0) {
        printf("\nEnter an option: ");
        scanf("%d", &option);

        if (option == 0) {
            // Exit program
            printf("\nProgram ended. \n");
        }
        else if (option == 1) {
            // Insert element at index
            printf("Enter element to insert: ");
            scanf("%d", &elt);

            // Empty list - insert at first position
            if (lst_size == 0) {
                lst[0] = elt;
                printf("\'%d\' inserted at index - '0'. \n", elt);
            }
            else {
                printf("Enter index number: ");
                scanf("%d", &ind);

                if (ind > lst_size) {
                    printf("Invalid index value. Out of bounds. \n");
                    continue;
                }
                else {
                    // Shift elements to the right
                    for (i = lst_size - 1; i >= ind; i--) {
                        lst[i+1] = lst[i];
                    }

                    // Insert element
                    lst[ind] = elt;
                    printf("\'%d\' inserted at index - \'%d\'. \n", elt,
ind);
                }
```

```c
        }

        lst_size++;
    }
    else if (option == 2) {
        // Delete index element
        if (lst_size == 0) {
            printf("List is empty. Nothing to delete. \n");
            continue;
        }
        else {
            printf("Enter index of element to delete: ");
            scanf("%d", &ind);

            if (ind > lst_size - 1) {
                printf("Invalid index value. Out of bounds. \n");
                continue;
            }
            else {
                // Shift elements to left
                for (i = ind; i < lst_size - 1; i++) {
                    lst[i] = lst[i+1];
                }
                printf("Element deleted at index - \'%d\'. \n", ind);
            }
        }

        lst_size--;
    }
    else if (option == 3) {
        printf("Enter element to delete: ");
        scanf("%d", &elt);

        flag = 0;

        for (i = 0; i < lst_size; i++) {
            if (lst[i] == elt) {
                // Shift elements to left
                for (j = i; j < lst_size - 1; j++) {
                    lst[j] = lst[j+1];
                }
                printf("Element deleted at index - \'%d\'. \n", i);
            }
        }

        lst_size--;
    }
    else if (option == 4) {
        // Search for element
        printf("Enter element to search: ");
        scanf("%d", &elt);
```

```
            flag = 0;

            for (i = 0; i < lst_size; i++) {
                if (lst[i] == elt) {
                    printf("\'%d\' found at index - \'%d\'. \n", elt, i);
                    flag = 1;
                    break;
                }
            }

            if (flag == 0) {
                printf("Element not found. \n");
            }
        }
        else if (option == 5) {
            // Display elements of list
            if (lst_size == 0) {
                printf("[ ] \n");
            }
            else {
                printf("[");

                for (i = 0; i < lst_size - 1; i++) {
                    printf("%d, ", lst[i]);
                }

                printf("%d] \n", lst[lst_size-1]);
            }
        }
        else {
            printf("Please enter a valid option from the menu (e.g., 1).
\n");
        }
    }

    return 0;
}
```

RESULT:

Ex. No: 3	LINKED LIST IMPLEMENTATION OF LISTS
03 / 03 / 22	

AIM:

To implement List Abstract Data Type (ADT) using Linked Lists.

THEORY:

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element, and a pointer to a structure containing its successor called the next pointer. The last cell's next pointer points to 'NULL'.

The worst-case time complexities of operations are:

- Printing – O(n).
- Searching for element – O(n).
- Searching for 'kth' element – O(n).
- Insertion – O(n).
- Deletion – O(n).

Insertion and deletion operations are linear-time, although the constant is larger in array implementation of lists. Insertion and deletion do not require shifting of other elements. Deletion can be executed in 1 pointer change and insertion command requires obtaining a new cell and then executing 2 pointer manoeuvres, after finding the required element / index.

Program 1:

```
/*
Write a program to implement list operations using the concept of linked
lists. The list operations to be implemented are
a. Insertion
b. Deletion
c. Search
d. Display list.
The program should be menu-driven which allows user to choose any operation
at any point in time. The user may also choose to exit the program.
*/

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

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

```
struct node *head;

// Return list length
int listLen() {
    if (head == NULL) {
        return 0;
    }
    else {
        struct node *temp = head;
        int size = 1;

        while (temp->next != NULL) {
            temp = temp->next;
            size++;
        }

        return size;
    }
}

// Insert element in index - '0'
void startInsert(int elt) {
    if (head == NULL) {
        head = (struct node *) malloc(sizeof(struct node *));
        head->next = NULL;
        head->data = elt;
    }
    else {
        struct node *temp = (struct node *) malloc(sizeof(struct node));
        temp->data = elt;
        temp->next = head;
        head = temp;
    }

    printf("\'%d\' inserted at index \'0\'. \n", elt);
}

// Insert element in index - '<list_length> - 1'
void endInsert(int elt) {
    if (head == NULL) {
        startInsert(elt);
    }
    else {
        struct node *temp = head;

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

        temp->next = (struct node *) malloc(sizeof(struct node));
        temp->next->data = elt;
        temp->next->next = NULL;
```

```c
        printf("\'%d\' inserted at index \'%d\'. \n", elt, listLen() - 1);
    }
}

// Insert element in index - '1' to '<list_length> - 2'
void indexInsert(int elt, int ind) {
    if (ind == 0) {
        startInsert(elt);
    }
    else if (ind > listLen() - 1) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        int i = 0;
        struct node *temp = head;

        while (i < ind - 1) {
            temp = temp->next;
            i++;
        }

        struct node *temp1 = (struct node *) malloc(sizeof(struct node));
        temp1->next = temp->next;
        temp1->data = elt;

        temp->next = temp1;

        printf("\'%d\' inserted at index \'%d\'. \n", elt, ind);
    }
}

// Delete element from index - '0'
void startDelete() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        printf("\'%d\' deleted from index \'0\'. \n", head->data);
        head = head->next;
    }
}

// Delete element in index - '<list_length> - 1'
void endDelete() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        if (head->next == NULL) {
            startDelete();
        }
        else {
            struct node *temp = head;
            while (temp->next->next != NULL) {
                temp = temp->next;
            }

            printf("\'%d\' deleted from index \'%d\'. \n", temp->next-
>data, listLen() - 1);
            temp->next = NULL;
        }
    }
}

// Delete element in index - '1' to '<list_length> - 2'
void indexDelete(int ind) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (ind > listLen() - 1 || ind < 0) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        if (ind == 0) {
            startDelete();
        }
        else if (ind == listLen() - 1) {
            endDelete();
        }
        else {
            int i = 0;
            struct node *temp = head;

            while (i < ind - 1) {
                temp = temp->next;
                i++;
            }

            printf("\'%d\' deleted from index \'%d\'. \n", temp->next-
>data, ind);
            temp->next = temp->next->next;
        }
    }
}

void eltDelete(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        struct node *temp = head;

        if (temp->data == elt) {
            head = head->next;
            printf("\'%d\' deleted from index \'0\'. \n", elt);
        }
        else {
            int ind = 0;
            while (temp->next != NULL && temp->next->data != elt) {
                temp = temp->next;
                ind++;
            }

            if (temp->next != NULL) {
                temp->next = temp->next->next;
                printf("\'%d\' deleted from index \'%d\'. \n", elt, ind +
1);
            }
            else {
                printf("\'%d\' not found. \n", elt);
            }
        }
    }
}

void search(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to search. \n");
    }
    else {
        struct node *temp = head;
        int flag = 0;
        int ind = 0;

        while (temp->data != elt) {
            ind++;
            if (temp->next == NULL) {
                flag = 1;
                break;
            }
            temp = temp->next;
        }

        if (flag == 1) {
            printf("\'%d\' not found in list. \n", elt);
        }
        else {
            printf("\'%d\' found at index - \'%d\'. \n", elt, ind);
        }
    }
}

void display() {
    if (listLen() == 0) {
        printf("[] \n");
    }
    else if (listLen() == 1) {
        printf("[%d] \n", head->data);
    }
    else {
        struct node *temp = head;

        printf("[");
        while (temp->next->next != NULL) {
            printf("%d, ", temp->data);
            temp = temp->next;
        }
        printf("%d, %d] \n", temp->data, temp->next->data);
    }
}

int main() {
    int option = 1;
    int elt, ind;
    int pos;

    // Print menu
    printf("*** LIST IMPLEMENTATION *** \n");

    printf("\n** OPERATIONS MENU ** \n");
    printf("1. Insert Element. \n");    // Insert element at specified
index
    printf("2. Delete Element. \n");    // Delete element at given index
    printf("3. Search Element. \n");    // Search for first occurrence of
element
    printf("4. Display List. \n");    // Display entire list
    printf("0. Exit. \n");

    while (option != 0) {

        printf("\nEnter an option: ");
        scanf("%d", &option);

        switch (option) {
            case 0:
                printf("Program ended. \n");
                break;

            case 1:
                printf("\n* INSERT POSITION * \n");
                printf("1. Start. \n");
                printf("2. Specified index. \n");
                printf("3. End. \n");

                printf("\nEnter position to insert: ");
                scanf("%d", &pos);
                if (pos == 1) {
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);
                    startInsert(elt);
                }
                else if (pos == 2) {
                    printf("Enter index number: ");
                    scanf("%d", &ind);
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);

                    if (ind == 0) {
                        startInsert(elt);
                    } else if (ind == listLen()) {
                        endInsert(elt);
                    }
                    else {
                        indexInsert(elt, ind);
                    }
                }
                else if (pos == 3) {
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);
                    endInsert(elt);
                }
                else {
                    printf("Invalid option. \n");
                }

                break;

            case 2:
                printf("\n* DELETE POSITION * \n");
                printf("1. Start. \n");
                printf("2. Specified index. \n");
                printf("3. Specified element. \n");
                printf("4. End. \n");

                printf("\nEnter position to delete: ");
                scanf("%d", &pos);

                if (pos == 1) {
                    startDelete();
                }
                else if (pos == 2) {
                    printf("Enter index number: ");
                    scanf("%d", &ind);
                    indexDelete(ind);
                }
                else if (pos == 3) {
                    printf("Enter element to delete: ");
                    scanf("%d", &elt);
```

```
                    eltDelete(elt);
                }
                else if (pos == 4) {
                    endDelete();
                }
                else {
                    printf("Invalid option. \n");
                }

                break;

            case 3:
                printf("\nEnter element to search: ");
                scanf("%d", &elt);
                search(elt);
                break;

            case 4:
                display();
                break;

            default:
                printf("Invalid option. Please choose a valid option from
the menu. \n");
        }
    }

    return 0;
}
```

RESULT:

Ex. No: 4	DOUBLY AND CIRCULAR LINKED LISTS
10 / 03 / 22	

AIM:

To implement doubly-linked and circular linked list ADT's using Linked Lists.

THEORY:

Circular linked list is a linked list where all nodes are connected to form a circle. The last node points to the head of the list.

Advantage: Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again. It is useful for the implementation of a queue.

Doubly linked list is a list where each node points to not only the successor but to the predecessor. There are two NULLs - one at the first and another at the last node in the list.

Advantage: Given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards.

Program 1:

```
/*
 A doubly linked list is a linked list in which every node has two pointers
- one to the previous node and one to the next node. Implement the insertion
and deletion operations in a doubly linked list.
*/

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

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

struct node *head;

// Return list length
int listLen(void) {
    if (head == NULL) {
        return 0;
    }
    else {
```

```c
        struct node *temp = head;
        int size = 1;

        while (temp->next != NULL) {
            temp = temp->next;
            size++;
        }

        return size;
    }
}

// Insert element in index - '0'
void startInsert(int elt) {
    if (head == NULL) {
        head = (struct node *) malloc(sizeof(struct node *));
        head->next = NULL;
        head->prev = NULL;
        head->data = elt;
    }
    else {
        struct node *temp = (struct node *) malloc(sizeof(struct node));
        temp->prev = NULL;
        temp->next = head;
        temp->data = elt;

        head->prev = temp;
        head = temp;
    }
    printf("\'%d\' inserted at index \'0\'. \n", elt);
}

// Insert element in index - '<list_length> - 1'
void endInsert(int elt) {
    if (head == NULL) {
        startInsert(elt);
    }
    else {
        struct node *temp1 = head;

        // Go to last element
        while (temp1->next != NULL) {
            temp1 = temp1->next;
        }

        // Insert element after last element
        struct node *temp2 = (struct node *) malloc(sizeof(struct node));
        temp1->next = temp2;

        temp2->next = NULL;
        temp2->prev = temp1;
        temp2->data = elt;
        printf("\'%d\' inserted at index \'%d\'. \n", elt, listLen() - 1);
    }
}

// Insert element in index - '1' to '<list_length> - 2'
void indexInsert(int elt, int ind) {
    if (ind == 0) {
        startInsert(elt);
    }
    else if (ind > listLen() - 1) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        int i = 0;
        struct node *temp1 = head;

        while (i < ind - 1) {
            temp1 = temp1->next;
            i++;
        }

        struct node *temp2 = (struct node *) malloc(sizeof(struct node));

        temp2->next = temp1->next;
        temp2->prev = temp1;
        temp2->data = elt;
        temp1->next = temp2;

        printf("\'%d\' inserted at index \'%d\'. \n", elt, ind);
    }
}

// Delete element from index - '0'
void startDelete(void) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        printf("\'%d\' deleted from index \'0\'. \n", head->data);

        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
    }
}

// Delete element in index - '<list_length> - 1'
void endDelete(void) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (head->next == NULL) {
        printf("\'%d\' deleted from index \'%d\'. \n", head->data,
listLen() - 1);
        head = NULL;
    }
    else {
        struct node *temp1 = head, *temp2 = head->next;

        while (temp2->next != NULL) {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }

        printf("\'%d\' deleted from index \'%d\'. \n", temp2->data,
listLen() - 1);
        temp1->next = temp2->next;
    }
}

// Delete element in index - '1' to '<list_length> - 2'
void indexDelete(int ind) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (ind > listLen() - 1 || ind < 0) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        if (ind == 0) {
            startDelete();
        }
        else if (ind == listLen() - 1) {
            endDelete();
        }
        else {
            int i = 0;
            struct node *temp = head;

            while (i < ind - 1) {
                temp = temp->next;
                i++;
            }

            printf("\'%d\' deleted from index \'%d\'. \n", temp->next-
>data, ind);
            temp->next = temp->next->next;
        }
    }
}

void eltDelete(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        int ind = 0;
        struct node *temp = head;

        // Find element in list
        while (temp->data != elt && temp->next != NULL) {
            ind++;
            temp = temp->next;
        }

        // Element found
        if (temp->data == elt) {
            // Element is at start
            if (temp == head) {
                startDelete();
            }
            // Element is at end
            else if (temp->next == NULL) {
                endDelete();
            }
            else {
                // Element is in middle
                indexDelete(ind);
            }
        }

        // Element not found
        else {
            printf("\'%d\' not found. \n", elt);
        }
    }
}

void search(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to search. \n");
    }
    else {
        struct node *temp = head;
        int flag = 0;
        int ind = 0;

        while (temp->data != elt) {
            ind++;
            if (temp->next == NULL) {
                flag = 1;
                break;
            }
            temp = temp->next;
        }

        if (flag == 1) {
            printf("\'%d\' not found in list. \n", elt);
        }
        else {
            printf("\'%d\' found at index - \'%d\'. \n", elt, ind);
        }
    }
}

void display(void) {
    if (listLen() == 0) {
        printf("[] \n");
    }
    else if (listLen() == 1) {
        printf("[%d] \n", head->data);
    }
    else {
        struct node *temp = head;

        printf("[");
        while (temp->next->next != NULL) {
            printf("%d, ", temp->data);
            temp = temp->next;
        }
        printf("%d, %d] \n", temp->data, temp->next->data);
    }
}

int main() {
    int option = 1;
    int elt, ind;
    int pos;

    // Print menu
    printf("*** LIST IMPLEMENTATION *** \n");

    printf("\n** OPERATIONS MENU ** \n");
    printf("1. Insert Element. \n");    // Insert element at specified
index
    printf("2. Delete Element. \n");    // Delete element at given index
    printf("3. Search Element. \n");    // Search for first occurrence of
element
    printf("4. Display List. \n");    // Display entire list
    printf("0. Exit. \n");

    while (option != 0) {

        printf("\nEnter an option: ");
        scanf("%d", &option);
```

```c
switch (option) {
    case 0:
        printf("Program ended. \n");
        break;

    case 1:
        printf("\n* INSERT POSITION * \n");
        printf("1. Start. \n");
        printf("2. Specified index. \n");
        printf("3. End. \n");

        printf("\nEnter position to insert: ");
        scanf("%d", &pos);

        if (pos == 1) {
            printf("Enter element to insert: ");
            scanf("%d", &elt);
            startInsert(elt);
        }
        else if (pos == 2) {
            printf("Enter index number: ");
            scanf("%d", &ind);
            printf("Enter element to insert: ");
            scanf("%d", &elt);

            if (ind == 0) {
                startInsert(elt);
            } else if (ind == listLen()) {
                endInsert(elt);
            }
            else {
                indexInsert(elt, ind);
            }
        }
        else if (pos == 3) {
            printf("Enter element to insert: ");
            scanf("%d", &elt);
            endInsert(elt);
        }
        else {
            printf("Invalid option. \n");
        }

        break;

    case 2:
        printf("\n* DELETE POSITION * \n");
        printf("1. Start. \n");
        printf("2. Specified index. \n");
        printf("3. Specified element. \n");
        printf("4. End. \n");
```

```
                printf("\nEnter position to delete: ");
                scanf("%d", &pos);

                if (pos == 1) {
                    startDelete();
                }
                else if (pos == 2) {
                    printf("Enter index number: ");
                    scanf("%d", &ind);
                    indexDelete(ind);
                }
                else if (pos == 3) {
                    printf("Enter element to delete: ");
                    scanf("%d", &elt);
                    eltDelete(elt);
                }
                else if (pos == 4) {
                    endDelete();
                }
                else {
                    printf("Invalid option. \n");
                }

                break;

            case 3:
                printf("\nEnter element to search: ");
                scanf("%d", &elt);
                search(elt);
                break;

            case 4:
                display();
                break;

            default:
                printf("Invalid option. Please choose a valid option from
the menu. \n");
        }
    }

    return 0;
}
```

Program 2:

```
/*
A circular linked list is a linked list in which the last node points to the
first node. Implement the insertion and deletion operations in a circular
linked list.
*/


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

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

struct node *head;

// Return list length
int listLen(void) {
    if (head == NULL) {
        return 0;
    }
    else {
        struct node *temp = head;
        int size = 1;

        while (temp->next != head) {
            temp = temp->next;
            size++;
        }

        return size;
    }
}

// Insert element in index - '0'
void startInsert(int elt) {
    if (head == NULL) {
        head = (struct node *) malloc(sizeof(struct node *));
        head->next = head;
        head->data = elt;
    }
    else {
        struct node *temp = head;

        // Go to element before head
        while (temp->next != head) {
            temp = temp->next;
        }
```

```c
        // Insert element between last element and head
        temp->next = (struct node *) malloc(sizeof(struct node));
        temp = temp->next;

        temp->next = head;
        temp->data = elt;

        head = temp;
    }

    printf("\'%d\' inserted at index \'0\'. \n", elt);
}

// Insert element in index - '<list_length> - 1'
void endInsert(int elt) {
    if (head == NULL) {
        startInsert(elt);
    }
    else {
        struct node *temp = head;

        // Go to element before head
        while (temp->next != head) {
            temp = temp->next;
        }

        // Insert element between last element and head
        temp->next = (struct node *) malloc(sizeof(struct node));
        temp = temp->next;

        temp->next = head;
        temp->data = elt;

        printf("\'%d\' inserted at index \'%d\'. \n", elt, listLen() - 1);
    }
}

// Insert element in index - '1' to '<list_length> - 2'
void indexInsert(int elt, int ind) {
    if (ind == 0) {
        startInsert(elt);
    }
    else if (ind > listLen() - 1) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        int i = 0;
        struct node *temp = head;

        while (i < ind - 1) {
            temp = temp->next;
            i++;
        }

        struct node *temp1 = (struct node *) malloc(sizeof(struct node));
        temp1->next = temp->next;
        temp1->data = elt;

        temp->next = temp1;

        printf("\'%d\' inserted at index \'%d\'. \n", elt, ind);
    }
}

// Delete element from index - '0'
void startDelete(void) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (head->next == head) {
        printf("\'%d\' deleted from index \'0\'. \n", head->data);
        head = NULL;
    }
    else {
        printf("\'%d\' deleted from index \'0\'. \n", head->data);

        struct node *temp = head;

        // Go to element before head
        while (temp->next != head) {
            temp = temp->next;
        }

        // Delete element between last element and head
        temp->next = head->next;
        head = temp->next;
    }
}

// Delete element in index - '<list_length> - 1'
void endDelete(void) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (head->next == head) {
        printf("\'%d\' deleted from index \'%d\'. \n", head->data,
listLen() - 1);
        head = NULL;
    }
    else {
        struct node *temp1 = head, *temp2 = head->next;

        while (temp2->next != head) {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        printf("\'%d\' deleted from index \'%d\'. \n", temp2->data,
listLen() - 1);
        temp1->next = temp2->next;
    }
}

// Delete element in index - '1' to '<list_length> - 2'
void indexDelete(int ind) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else if (ind > listLen() - 1 || ind < 0) {
        printf("Invalid index. Index out of bounds. \n");
    }
    else {
        if (ind == 0) {
            startDelete();
        }
        else if (ind == listLen() - 1) {
            endDelete();
        }
        else {
            int i = 0;
            struct node *temp = head;

            while (i < ind - 1) {
                temp = temp->next;
                i++;
            }

            printf("\'%d\' deleted from index \'%d\'. \n", temp->next-
>data, ind);
            temp->next = temp->next->next;
        }
    }
}

void eltDelete(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete. \n");
    }
    else {
        int ind = 0;
        struct node *temp = head;

        // Find element in list
        while (temp->data != elt && temp->next != head) {
            ind++;
            temp = temp->next;
        }

        // Element found
        if (temp->data == elt) {
            // Element is at start
            if (temp == head) {
                startDelete();
            }
            // Element is at end
            else if (temp->next == head) {
                endDelete();
            }
            else {
                // Element is in middle
                indexDelete(ind);
            }
        }

        // Element not found
        else {
            printf("\'%d\' not found. \n", elt);
        }
    }
}

void search(int elt) {
    if (head == NULL) {
        printf("List is empty. Nothing to search. \n");
    }
    else {
        struct node *temp = head;
        int flag = 0;
        int ind = 0;

        while (temp->data != elt) {
            ind++;
            if (temp->next == head) {
                flag = 1;
                break;
            }
            temp = temp->next;
        }

        if (flag == 1) {
            printf("\'%d\' not found in list. \n", elt);
        }
        else {
            printf("\'%d\' found at index - \'%d\'. \n", elt, ind);
        }
    }
}
```

```
void display(void) {
    if (listLen() == 0) {
        printf("[] \n");
    }
    else if (listLen() == 1) {
        printf("[%d] \n", head->data);
    }
    else {
        struct node *temp = head;

        printf("[");
        while (temp->next->next != head) {
            printf("%d, ", temp->data);
            temp = temp->next;
        }
        printf("%d, %d] \n", temp->data, temp->next->data);
    }
}

int main() {
    int option = 1;
    int elt, ind;
    int pos;

    // Print menu
    printf("*** LIST IMPLEMENTATION *** \n");

    printf("\n** OPERATIONS MENU ** \n");
    printf("1. Insert Element. \n");    // Insert element at specified
index
    printf("2. Delete Element. \n");    // Delete element at given index
    printf("3. Search Element. \n");    // Search for first occurrence of
element
    printf("4. Display List. \n");    // Display entire list
    printf("0. Exit. \n");

    while (option != 0) {

        printf("\nEnter an option: ");
        scanf("%d", &option);

        switch (option) {
            case 0:
                printf("Program ended. \n");
                break;

            case 1:
                printf("\n* INSERT POSITION * \n");
                printf("1. Start. \n");
                printf("2. Specified index. \n");
                printf("3. End. \n");
                printf("\nEnter position to insert: ");
                scanf("%d", &pos);

                if (pos == 1) {
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);
                    startInsert(elt);
                }
                else if (pos == 2) {
                    printf("Enter index number: ");
                    scanf("%d", &ind);
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);

                    if (ind == 0) {
                        startInsert(elt);
                    } else if (ind == listLen()) {
                        endInsert(elt);
                    }
                    else {
                        indexInsert(elt, ind);
                    }
                }
                else if (pos == 3) {
                    printf("Enter element to insert: ");
                    scanf("%d", &elt);
                    endInsert(elt);
                }
                else {
                    printf("Invalid option. \n");
                }

                break;

            case 2:
                printf("\n* DELETE POSITION * \n");
                printf("1. Start. \n");
                printf("2. Specified index. \n");
                printf("3. Specified element. \n");
                printf("4. End. \n");

                printf("\nEnter position to delete: ");
                scanf("%d", &pos);

                if (pos == 1) {
                    startDelete();
                }
                else if (pos == 2) {
                    printf("Enter index number: ");
                    scanf("%d", &ind);
                    indexDelete(ind);
                }
```

```
                else if (pos == 3) {
                    printf("Enter element to delete: ");
                    scanf("%d", &elt);
                    eltDelete(elt);
                }
                else if (pos == 4) {
                    endDelete();
                }
                else {
                    printf("Invalid option. \n");
                }

                break;

            case 3:
                printf("\nEnter element to search: ");
                scanf("%d", &elt);
                search(elt);
                break;

            case 4:
                display();
                break;

            default:
                printf("Invalid option. Please choose a valid option from
the menu. \n");
        }
    }

    return 0;
}
```

RESULT:

Ex. No: 5	IMPLEMENTATION OF STACK DATA STRUCTURE
24 / 03 / 22	

AIM:

To implement Stack data structure using arrays and linked lists.

THEORY:

Stack is an abstract data type (ADT) that works on Last In First Out (LIFO) policy which provides operations like push, pop, etc. for the users to interact with the data. The two primary operations on stacks are – Push and pop. Pushing an element stores an element on the stack. If the stack is full, then it is said to be an Overflow condition. Removing (popping) an elements lets the user access the element from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition. Peeking into the stack provides access the top-most element without popping it.

Program 1:

```
/*
Program to implement the stack ADT using arrays. push(), pop(), top(),
isEmpty() and isFull() operations must be implemented.
*/

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

#define MAX_SIZE 10

// Declare stack globally
int stk[MAX_SIZE];
int stkSize = 0;
// Note: It is a better practice to NOT use global variables.

bool isFull() {
        if (stkSize == MAX_SIZE) {
            return true;
        }
        else {
            return false;
        }
}

bool isEmpty() {
        if (stkSize == 0) {
            return true;
        }
        else {
            return false;
        }
}

void push() {
        if (isFull()) {
            printf("Stack overflow. \n");
        }
        else {
            int elt;
            printf("Enter element: ");
            scanf("%d", &elt);

            stkSize++;
            stk[stkSize - 1] = elt;
            printf("%d pushed into stack. \n", elt);
        }
}

void pop() {
        if (isEmpty()) {
            printf("Stack underflow. \n");
        }
        else {
            printf("%d popped out of stack. \n", stk[stkSize - 1]);
            stkSize--;
        }
}

void top() {
        if (isEmpty()) {
            printf("Stack is empty. \n");
        }
        else {
            printf("%d <- Top. \n", stk[stkSize-1]);
        }
}

int main() {
        printf("*** STACK IMPLEMENTATION USING ARRAYS *** \n");

        int option;
        printf("** Options Menu ** \n");
        printf("1. Push (Insert element). \n");
        printf("2. Pop (Delete element). \n");
        printf("3. Peek (Print top element). \n");
        printf("0. Exit. \n");

        while (true) {
            printf("\nEnter option: ");
            scanf("%d", &option);

            switch (option) {
                  case 1:
                        push();
                        break;
                  case 2:
                        pop();
                        break;
                  case 3:
                        top();
                        break;
                  case 0:
                        printf("Program ended. \n");
                        exit(0);
                        break;
                  default:
                        printf("Please enter a valid option (e.g., 1) from
the menu. \n");
            }
        }

        return 0;
}
```

Program 2:

```
/*
Program to implement the stack ADT using linked lists. push(), pop(), top()
and isEmpty() operations must be implemented.
*/

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

// Linked List - Node
struct node {
    int data;
    struct node *next;
} *head;

bool isEmpty() {
    if (head == NULL) {
        return true;
    }
    else {
        return false;
    }
}

void push() {
    int elt;
    printf("Enter element to push: ");
    scanf("%d", &elt);

    printf("\'%d\' pushed into stack. \n", elt);

    if (isEmpty()) {
        head = (struct node *) malloc(sizeof(struct node));
        head->data = elt;
        head->next = NULL;
    }
    else {
        // Add a new node before head & make it head node.
        struct node *temp = (struct node *) malloc(sizeof(struct node));
        temp->data = elt;
        temp->next = head;
        head = temp;
    }
}

void pop() {
    if (isEmpty()) {
        printf("Stack underflow. \n");
    }
    else {
        printf("\'%d\' popped out of stack. \n", head->data);
        head = head->next;
    }
}

void top() {
    if (isEmpty()) {
        printf("Stack is empty. \n");
    }
    else {
        printf("\'%d\' <- Stack top. \n", head->data);
    }
}

int main() {
    printf("*** Implementation of Stack Using Linked Lists *** \n");
    printf("** Options Menu ** \n");
    printf("1. Push into stack. \n");
    printf("2. Pop out of stack. \n");
    printf("3. Peek at top. \n");
    printf("0. Exit. \n");

    int opt;

    while (true) {
        printf("\nEnter option: ");
        scanf("%d", &opt);

        switch (opt) {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                top();
                break;
            case 0:
                printf("Program ended. \n");
                exit(0);
                break;
            default:
                printf("Invalid option. Please enter a valid option. \n");
                break;
        }
    }

    return 0;
}
```

```
/*
Note: Typical linked list format is not used.

For stack implementation using linked lists:
push() -> Add a new node before 'head' and make it 'head'.
pop() -> Move to next node in list and make it 'head'.
top() -> Read data in 'head' node.
*/
```

RESULT:

Ex. No: 6	APPLICATIONS OF STACKS
31 / 03 / 22	

AIM:

To use stacks to (i) Convert infix expressions to postfix expressions, and (ii) evaluate postfix expressions.

THEORY:

In a infix expression, the operator is written in between two operands. In a postfix expression, the operator is written after the operands. A stack can be used to convert an infix expression to a postfix expression.

E.g. Infix : a + b * c + (d * e + f) * g
Postfix: a b c * + d e * f + g * +

Evaluation of normal mathematical expressions require precedence rules to get the correct answer. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. Thus, in postfix (or reverse polish) notation, precedence is in-built.

E.g. 2 + (3 * 4) is represented as 2 3 4 * +

(2 + 3) * 4 is represented as 2 3 + 4 *

Evaluation of a postfix expression can be done easily using a stack.

Program 1:

```
/*
Program to convert an infix expression into postfix expression using 'stack'
data structure.
*/

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

// Linked List Implementation of Stacks
struct node {
    char data;
    struct node *next;
} *head;
```

```
// Prototype of functions - Function Declaration
bool isEmpty(void);
char pop(void);
char top(void);

// Note: Instead, type 'void' in fxn definition if function takes no
arguments

bool isEmpty() {
    return (head == NULL);
}

void push(char elt) {
    struct node *temp = (struct node *) malloc(sizeof(struct node));
    temp->next = head;
    temp->data = elt;
    head = temp;
}

char pop() {
    char retVal = head->data;
    head = head->next;
    return retVal;
}

char top() {
    return isEmpty() ? '\0' : head->data;    // Ternary operator in C
    // Format: (condition) ? (exec. if cond. is true) : (exec. if cond. is
false)

    /*
    (or)

    if (isEmpty()) {
        return '\0';
    }
    else {
        return head->data;
    }
     */
}

// Returns precedence of the argument character
int preced(char elt) {
    if (elt == '\0') {
        return -1;
    }
    else if (elt == '(') {
        return 0;
    }
    else {
        int flag = 0;
        char sym[4] = {'+', '-', '*', '/'};
        int preced[4] = {1, 1, 2, 2};

        for (int i = 0; i < 4; i++) {
            if (sym[i] == elt) {
                flag = 1;
                return preced[i];
            }
        }
        return '\0';
    }
}

// Checks if argument character is an alphabet (a-z & A-Z) or not
bool isAlp(char alp) {
    char alpArr[52] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

    for (int i = 0; i < 52; i++) {
        if (alp == alpArr[i]) {
            return true;
        }
    }
    return false;
}

// Checks if argument character is a number (0-9) or not
bool isNum(char num) {
    char numArr[10] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

    for (int i = 0; i < 10; i++) {
        if (num == numArr[i]) {
            return true;
        }
    }
    return false;
}

// Checks if argument character is a symbol (+,-,/,*) or not
bool isSym(char sym) {
    char symArr[4] = {'+', '-', '*', '/'};

    for (int i = 0; i < 4; i++) {
        if (sym == symArr[i]) {
            return true;
        }
    }
    return false;
}
```

```
int main() {

    printf("*** Infix to Postfix Converter *** \n");

    // Input
    char infix[100];
    printf("\nEnter infix expression: ");
    scanf("%s", infix);

    char postfix[100] = {};
    int i;
    int pos = 0;

    for (i = 0; i < strlen(infix); i++) {
        // Check for number & Add to Postfix
        if (isNum(infix[i])) {
            postfix[pos++] = infix[i];
        }

        // Check for alphabet & Add to Postfix
        else if (isAlp(infix[i])) {
            postfix[pos++] = infix[i];
        }

        // Check for symbol
        else if (isSym(infix[i])) {
            while ((preced(infix[i])) <= preced(top())) {
                postfix[pos++] = pop();
            }
            push(infix[i]);
        }

        // Check for parenthesis
        else if (infix[i] == '(') {
            push(infix[i]);
        }
        else if (infix[i] == ')') {
            // Pop all elements of stack until '(' is reached
            while (top() != '(') {
                postfix[pos++] = pop();
            }
            // Pop '('
            pop();
        }
        else {
            printf("Invalid character(s) in expression. Only numbers (0-9), 
alphabets (A-Z & a-z) and \'+\', \'-\', \'/\' and \'*\' are allowed. \n");
            exit(0);
        }
    }

    // Pop remaining elements of stack
    while(!(isEmpty())) {
        postfix[pos++] = pop();
    }

    // Output
    printf("Postfix: %s \n", postfix);

    return 0;
}

/*
 i) Note:

 postfix[pos] = pop();
 pos++

 (alternative)

 postfix[pos++] = pop();

 ii) Note:

 <type> foo() {    <- Definition
    <statements>
 }

 is an old-style function definition in C without any parameters.

 If no parameters, declare function with parameter 'void' to avoid errors.
 E.g., <type> foo(void);    <- Declaration

 (or)

 Type 'void' in function definition itself if function takes no parameters.
 */
```

Program 2:

```
/*
Program to evaluate a postfix expression using 'stack' data structure.
*/

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

// Linked List Implementation of Stacks
struct node {
    int data;
    struct node *next;
} *head;

void push(float elt) {
    struct node *temp = (struct node *) malloc(sizeof(struct node));
    temp->next = head;
    temp->data = elt;
    head = temp;
}

float pop(void) {    // Note: Enter 'void' if function takes no parameters
    float retVal = head->data;
    head = head->next;
    return retVal;
}

// Checks if argument character is a number (0-9) or not
bool isNum(char num) {
    char numArr[10] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

    for (int i = 0; i < 10; i++) {
        if (num == numArr[i]) {
            return true;
        }
    }
    return false;
}

// Checks if argument character is a symbol (+,-,/,*) or not
bool isSym(char sym) {
    char symArr[4] = {'+', '-', '*', '/'};

    for (int i = 0; i < 4; i++) {
        if (sym == symArr[i]) {
            return true;
        }
    }
    return false;
}

float charToInt(char num) {
    if (num == '1') {
        return 1.0;
    }
    else if (num == '2') {
        return 2.0;
    }
    else if (num == '3') {
        return 3.0;
    }
    else if (num == '4') {
        return 4.0;
    }
    else if (num == '5') {
        return 5.0;
    }
    else if (num == '6') {
        return 6.0;
    }
    else if (num == '7') {
        return 7.0;
    }
    else if (num == '8') {
        return 8.0;
    }
    else if (num == '9') {
        return 9.0;
    }
    else {
        // If 'num' is '0'
        return 0.0;
    }
}

// Evaluates the expression
float evaluate(float a, float b, char sym) {
    if (sym == '+') {
        return a + b;
    }
    else if (sym == '-') {
        return a - b;
    }
    else if (sym == '/') {
        return a / b;
    }
    else {
        // Symbol is '*'
        return a * b;
    }
}
```

```
int main() {

    printf("*** Postfix Evaluator *** \n");

    // Input
    char postfix[100];
    printf("\nEnter postfix expression: ");
    scanf("%s", postfix);

    int a, b;
    int i;

    for (i = 0; i < strlen(postfix); i++) {
        if (isNum(postfix[i]) == true) {
            push(charToInt(postfix[i]));
        }
        else if (isSym(postfix[i]) == true) {
            b = pop();
            a = pop();
            push(evaluate(a, b, postfix[i]));
        }
        else {
            printf("Invalid character(s) in expression. Only numbers (0-9)
and \'+\', \'-\', \'/\' and \'*\' are allowed. \n");
            exit(0);
        }
    }

    // Output
    printf("Evaluated expression is %f \n", pop());

    return 0;
}

/*
Note:
        '-4 7 * 5 +' -> Update code to evaluate expression with negative
numbers.
*/
```

RESULT:

Ex. No: 7	IMPLEMENTATION OF QUEUE DATA STRUCTURE
07 / 04 / 22	

AIM:

To implement Queue data structure using arrays and linked lists.

THEORY:

In a queue, insertion is done at one end whereas deletion is performed at the other end. It is a First In First Out (FIFO) structure. Basic queue operations are:

- Enqueue: Inserts an element at the end of the list (called the rear).
- Dequeue: Deletes (and returns) an element at the start of the list (called the front).
- Peek: Gets the front element

In array implementation of queues, multiple enqueue and dequeue operations may lead to situations where no more insertions are possible even though there is free space in the array. A workaround would be to use circular queues with arrays or to use linked lists to implement queues.

Program 1:

```
/*
Write a program to implement the Queue ADT using arrays. You must implement
enqueue(), dequeue(), peek(), isEmpty() and isFull() operations. Implement
queue in a circular fashion in the array.
*/

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

// Initialising queue
int queue[MAX_SIZE] = {};
int front = -1;
int rear = -1;

int isEmpty(void) {
    return front == -1;
}

int isFull(void) {
    if (front == 0 && rear == MAX_SIZE - 1) {
        return 1;
    }
    else if (rear == front - 1) {
        return 1;
    }
    else {
        return 0;
    }
}

int enqueue(int elt) {
    /*
     Returns '\0' (NULL) if queue is full. Returns argument 'elt'
otherwise.
    */
    // Full queue
    if (isFull()) {
        return '\0';
    }
    else {
        // Empty queue
        if (isEmpty()) {
            front++;
            rear++;
            queue[rear] = elt;
        }
        else if (rear == MAX_SIZE - 1) {
            rear = 0;
            queue[rear] = elt;
        }
        else {
            rear++;
            queue[rear] = elt;
        }

        return elt;
    }
}

int dequeue(void) {
    if (isEmpty()) {
        return '\0';
    }
    else {
        int retVal = queue[front];

        if (front == rear) {
            front = -1;
            rear = -1;
        }
        else if (front == MAX_SIZE - 1) {
            front = 0;
        }
        else {
            front++;
        }
        return retVal;
    }
}

void peek(void) {
    isEmpty() ? printf("Queue is empty. Nothing to peek at. \n") :
printf("%d <- Front. \n", queue[front]);
}

void display(void) {
    if (isEmpty()) {
        printf("[] \n");
    }
    else {
        printf("[");

        for (int i = front; i != rear; i = (i + 1) % MAX_SIZE) {
            printf("%d, ", queue[i]);
        }

        printf("%d] \n", queue[rear]);
    }
}

int main() {
    // Print options menu
    printf("*** IMPLEMENTATION OF CIRCULAR QUEUE ADT USING ARRAYS *** \n");
    printf("** Options Menu ** \n");
    printf("1. Enqueue - Insert element at rear. \n");
    printf("2. Dequeue - Delete element at front. \n");
    printf("3. Peek    - View front element. \n");
    printf("4. Display - Display entire queue. \n");
    printf("0. Exit    - End program. \n");

    int opt, elt;

    while(1) {
        // Input
        printf("\nEnter option: ");
        scanf("%d", &opt);

        switch (opt) {
            case 0:
                printf("Program ended. \n");
                exit(0);
            case 1:
                printf("Enter element: ");
                scanf("%d", &elt);
                enqueue(elt) == '\0' ? printf("Queue is full. \n") :
printf("\'%d\' enqueued. \n", elt);
                break;
            case 2:
                elt = dequeue();
                elt == '\0' ? printf("Queue is empty. Nothing to dequeue.
\n") : printf("'\%d\' dequeued. \n", elt);
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            default:
                printf("Invalid option. Enter a valid option (e.g., 4).
\n");
        }
    }

    return 0;
}

/*
Ternary Operators:
<condition> ? <execute_if_condition_is_true> :
<execute_if_condition_is_false>;
*/
```

Program 2:

```
/*
Write a program to implement the Queue ADT using linked list. You must
implement enqueue(), dequeue(), peek(), isEmpty() and isFull() operations.
*/

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

struct node {
    int data;
    struct node *next;
} *front, *rear;


int isEmpty(void) {
    return front == NULL;
}

int isFull(void) {
    struct node *temp = front;
    int nodeCount = 0;

    // Count number of nodes between front and rear
    while (temp != rear) {
        temp = temp->next;
        nodeCount++;
    }

    if (nodeCount == MAX_SIZE - 1) {
        return 1;
    }
    else {
        return 0;
    }
}

int enqueue(int elt) {
    /*
     Returns '\0' (NULL) if queue is full. Returns argument 'elt'
otherwise.
    */
    // Full queue
    if (isFull()) {
        return '\0';
    }
    else if (isEmpty()) {
        rear = (struct node *) malloc(sizeof(struct node));
        rear->data = elt;
        rear->next = NULL;
        front = rear;
        return elt;
    }
    else {
        struct node *temp = (struct node *) malloc(sizeof(struct node));
        temp->data = elt;
        temp->next = NULL;
        rear->next = temp;
        rear = temp;
        return elt;
    }
}

int dequeue(void) {
    if (isEmpty()) {
        return '\0';
    }
    else if (front->next == NULL) {
        int retVal = front->data;
        front = NULL;

        return retVal;
    }
    else {
        int retVal = front->data;
        front = front->next;

        return retVal;
    }
}

void peek(void) {
    isEmpty() ? printf("Queue is empty. Nothing to peek at. \n") :
printf("\'%d\' <- Front. \n", front->data);
}

void display(void) {
    if (isEmpty()) {
        printf("[] \n");
    }
    else {
        struct node *temp = front;

        printf("[");
        while (temp != rear) {
            printf("%d, ", temp->data);
            temp = temp->next;
        }
        printf("%d] \n", temp->data);
    }
}
```

```
int main() {
    // Print options menu
    printf("*** IMPLEMENTATION OF CIRCULAR QUEUE ADT USING LINKED LISTS ***
\n");
    printf("** Options Menu ** \n");
    printf("1. Enqueue - Insert element at rear. \n");
    printf("2. Dequeue - Delete element at front. \n");
    printf("3. Peek    - View front element. \n");
    printf("4. Display - Display entire queue. \n");
    printf("0. Exit    - End program. \n");

    int opt, elt;

    while(1) {
        // Input
        printf("\nEnter option: ");
        scanf("%d", &opt);

        switch (opt) {
            case 0:
                printf("Program ended. \n");
                exit(0);
            case 1:
                printf("Enter element: ");
                scanf("%d", &elt);
                enqueue(elt) == '\0' ? printf("Queue is full. \n") :
printf("\'%d\' enqueued. \n", elt);
                break;
            case 2:
                elt = dequeue();
                elt == '\0' ? printf("Queue is empty. Nothing to dequeue.
\n") : printf("'\%d\' dequeued. \n", elt);
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            default:
                printf("Invalid option. Enter a valid option (e.g., 4).
\n");
        }
    }

    return 0;
}
```

Program 3:

```
/*
A deque is a data structure consisting of a list of items, on which the
following operations are possible:

push(x): Inserts item 'x' in the front end of the deque.
pop(): Removes the front item from the deque and returns it.
inject(x): Inserts item 'x' in the rear end of the deque.
eject(): Removes the rear item from the queue and returns it.

Write routines to support the deque that take O(1) time per operation.

Note: The deque must be circular.
*/

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

#define MAX_SIZE 5    // Maximum size of deque

// Initialise empty Deque
int dq[MAX_SIZE] = {0};
int front = -1;
int rear = -1;

int isEmpty() {
        return (front == -1);
}

int isFull() {
        if ((rear == MAX_SIZE - 1 && front == 0) || (front - 1 == rear)) {
            printf("Deque is full. \n");
            return 1;
        }
        else {
            return 0;
        }
}

void push(int elt) {
        // Deque is full
        if (isFull()) {
            return;
        }
        // Deque is not full
        else {
            // No elements
            if (isEmpty()) {
                  front++;
                  rear++;
                  dq[front] = elt;
            }
            // Only 1 element
            else if (front == 0) {
                  front = MAX_SIZE - 1;
                  dq[front] = elt;
            }
            // Normal case
            else {
                  front--;
                  dq[front] = elt;
            }

            printf("\'%d\' was pushed. \n", elt);
        }
}

int pop() {
        // Deque is empty
        if (isEmpty()) {
            printf("Deque is empty. Nothing to pop. \n");
            return '\0';
        }
        // Deque has elements
        else {
            int retElt = dq[front];
            printf("\'%d\' was popped. \n", retElt);

            // Only 1 element
            if (front == rear) {
                  front = -1;
                  rear = -1;
                  return retElt;
            }
            // 'front' at 'MAX_SIZE - 1' -> Jump to '0'
            else if (front == MAX_SIZE - 1) {
                  front = 0;
                  return retElt;
            }
            // Normal case
            else {
                  front++;
                  return retElt;
            }
        }
}

void inject(int elt) {
        // Deque is full
        if (isFull()) {
            return;
        }
        // Deque is not full
        else {
            // No elements
            if (rear == -1) {
                  front++;
                  rear++;
                  dq[rear] = elt;
            }
            // 'rear' at 'MAX_SIZE - 1' -> Jump to '0'
            else if (rear == MAX_SIZE - 1) {
                  rear = 0;
                  dq[rear] = elt;
            }
            // Normal case
            else {
                  rear++;
                  dq[rear] = elt;
            }

            printf("\'%d\' was injected. \n", elt);
        }
}

int eject() {
        // Deque is empty
        if (isEmpty()) {
            printf("Deque is empty. Nothing to eject. \n");
            return '\0';
        }
        // Deque has elements
        else {
            int retElt = dq[rear];
            printf("\'%d\' was ejected. \n", retElt);

            // Only 1 element
            if (rear == front) {
                  front = -1;
                  rear = -1;
                  return retElt;
            }
            // 'rear' at '0' -> Jump to 'MAX_SIZE - 1'
            else if (rear == 0) {
                  rear = MAX_SIZE - 1;
                  return retElt;
            }
            // Normal case
            else {
                  rear--;
                  return retElt;
            }
        }
}
```

```
void display() {
        // Deque is empty
        if (isEmpty()) {
            printf("Deque is empty. Nothing to display. \n");
            return;
        }
        // Deque has elements
        else {
            int i;
            printf("[");

            for (i = front; i != rear; i = (i + 1) % MAX_SIZE) {
                  printf("%d, ", dq[i]);
            }
            printf("%d] \n", dq[rear]);
            }
}

int main() {
        // Print menu
        printf("*** CIRCULAR DEQUE IMPLEMENTATION *** \n");
        printf("*** Options Menu *** \n");
        printf("1. Push    - Insert at front of deque. \n");
        printf("2. Pop     - Remove front element of deque. \n");
        printf("3. Inject  - Insert at rear of deque. \n");
        printf("4. Eject   - Remove rear element of deque. \n");
        printf("5. Display - Display elements of deque. \n");
    printf("0. Exit    - End program. \n");

        int opt, elt;

        while(1) {
            // Input
            printf("\nEnter option: ");
            scanf("%d", &opt);

            switch(opt) {
                  case 0:
                        // Exit
                        printf("Program ended. \n");
                        exit(0);
                  case 1:
                        // Push
                        printf("Enter element: ");
                        scanf("%d", &elt);

                        push(elt);
                        break;
                  case 2:
                        // Pop
                        pop();
                        break;
```

```
                case 3:
                      // Inject
                      printf("Enter element: ");
                      scanf("%d", &elt);

                      inject(elt);
                      break;
                case 4:
                      // Eject
                      eject();
                      break;
                case 5:
                      // Display
                      display();
                      break;
                default:
                      printf("Invalid option. \n");
            }
        }

        return 0;
}

/*
Note:
(1)
        if (frontIsEmpty() == 1)

        *is same as*

        if (frontIsEmpty())

        -> 'frontIsEmpty()' will return '1'/'0'. 'if(1)' will
        execute statements under it & 'if(0)' will not.

(2)
        1 = true
        0 = false

(3)
        if (front == 1) {
            return 1;
        } else {
            return 0;
        }

        *is same as*

        return (front == 1);

        -> 'front == 1' will return '1'/'0' directly.
*/
```

RESULT:

Ex. No: 8	IMPLEMENTATION OF BINARY SEARCH TREE
21 / 04 / 22	

AIM:

To implement Binary Search Tree data structure.

THEORY:

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes. The topmost node in the tree is called the root node. It is a specially designated node which has no parent. The node which is the immediate successor of a node is called the child node of that node. The node which is the immediate predecessor of any node is called a parent node. Nodes with no child node is called a leaf node. Nodes with the same parent are sibling nodes. In a binary search tree, the maximum number of children for a node is 2 and the following condition is true for every node:

Left child node element < Root node element < Right child node element.

Program 1:

```
/*
 1. To a BST (Binary Search Tree) implementation with insert(), find() and
display() functions, add functions for deletion of nodes and 3 types of
traversals (inorder, preorder and postorder).
 2. Write a function for level order traversal of a tree (Level order
traversal is a traversal method in which nodes are visited level by level
from left to right).
 (Hint: Keep enqueuing the child nodes of every node that is visited starting
from root. Visit nodes in the order of the queue)
 3. Write a program to check if a binary tree is balanced. A binary tree is
balanced if the modulus of height difference 'h' between the right subtree
and left subtree is lesser than 2 (|h| < 2).
 */

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

// Binary Search Tree [BST] Node
struct node {
    int data;
    struct node *left, *right;
};

// Declare BST root node
struct node *root;
```

```
// Queue - for level order traversal
struct node *queue[100];
int front = -1;
int rear = -1;

void enqueue(struct node *temp) {
    // Queue is empty
    if (front == -1) {
        front = 0;
        rear = 0;
        queue[rear] = temp;
    }
    else {
        rear++;
        queue[rear] = temp;
    }
}

struct node * dequeue(void) {
    if (front == -1) {
        return NULL;
    }
    else if (front == 0 && rear == 0) {
        front = -1;
        rear = -1;
        return queue[front+1];
    }
    else {
        front++;
        return queue[front-1];
    }
}

struct node * insert(int elt, struct node *temp) {
    // Node is empty
    if (temp == NULL) {
        temp = (struct node *) malloc(sizeof(struct node));
        temp->data = elt;
        temp->left = NULL;
        temp->right = NULL;

        // Root node is empty
        if (root == NULL) {
            root = temp;
        }

        // Send 'temp' node as previous node's left (or) right
        return temp;
    }
    // Element smaller than node's element (node's element = temp->data)
    else if (elt < temp->data) {
```

```c
        temp->left = insert(elt, temp->left);
    }
    // Element larger than node's element
    else {
        // Condition: (elt > temp->data)
        temp->right = insert(elt, temp->right);
    }

    // Return argument node
    return temp;
}

int find(int elt, struct node *temp) {
    // Empty node
    if (temp == NULL) {
        return 0;
    }
    // Argument element smaller than node's element
    else if (elt < temp->data) {
        return find(elt, temp->left);
    }
    // Argument element larger than node's element
    else if (elt > temp->data) {
        return find(elt, temp->right);
    }
    // Argument element = node's element (Element found)
    else {
        return 1;
    }
}

void dispTree(struct node *temp, int space) {
    // Node is empty
    if (temp == NULL) {
        return;
    }

    space += COUNT;

    // Print right sub-tree
    dispTree(temp->right, space);

    // Print current node's data
    printf("%*d \n", space, temp->data);

    // Print left sub-tree
    dispTree(temp->left, space);
}

struct node * smallest(struct node *temp) {
    // Node is leftmost node
    if (temp->left == NULL) {
        return temp;
    }
    // Traverse to the leftmost node
    else {
        return smallest(temp->left);
    }
}

struct node * delete(int elt, struct node *temp) {
    // Node is empty
    if (temp == NULL) {
        return NULL;
    }
    // Arg element smaller than node's element
    else if (elt < temp->data) {
        temp->left = delete(elt, temp->left);
    }
    // Arg element larger than node's element
    else if (elt > temp->data) {
        temp->right = delete(elt, temp->right);
    }
    // Arg element = Node's element
    else {
        // Node has no children (leaf node)
        if (temp->left == NULL && temp->right == NULL) {
            return NULL;
        }
        // Node has only left child
        else if (temp->left != NULL && temp->right == NULL) {
            return temp->left;
        }
        // Node has only right child
        else if (temp->left == NULL && temp->right != NULL) {
            return temp->right;
        }
        // Node has 2 children
        else {
            // Select smallest element from right child
            int min = smallest(temp->right)->data;
            // Delete the smallest element
            delete(min, root);
            // Replace node with smallest element
            temp->data = min;
        }
    }

    // Return argument node to restore existing connections
    return temp;
}

// Pre-order traversal -> root, (left, right)
void preOrdTrav(struct node *temp) {
```

```c
    if (temp != NULL) {
        printf("%d ", temp->data);
        preOrdTrav(temp->left);
        preOrdTrav(temp->right);
    }
}

// In-order traversal -> left, root, right
void inOrdTrav(struct node *temp) {
    if (temp == NULL) {
        return;
    }
    else {
        inOrdTrav(temp->left);
        printf("%d ", temp->data);
        inOrdTrav(temp->right);
    }
}

// Post-order traversal -> (Left, Right), Root
void postOrdTrav(struct node *temp) {
    if (temp != NULL) {
        preOrdTrav(temp->left);
        preOrdTrav(temp->right);
        printf("%d ", temp->data);
    }
}

int subTreeHt(struct node *temp) {
    if (temp == NULL) {
        return -1;
    }
    else {
        int lHt = subTreeHt(temp->left);
        int rHt = subTreeHt(temp->right);

        // Use greater height
        if (lHt > rHt) {
            return lHt + 1;
        }
        else {
            return rHt + 1;
        }
    }
}

// Level order traversal
void lvlOrdTrav(struct node *temp) {
    if (temp != NULL) {
        // Visit node
        printf("%d ", temp->data);
        // Enqueue left child
        if (temp->left != NULL) {
            enqueue(temp->left);
        }
        // Enqueue right child
        if (temp->right != NULL) {
            enqueue(temp->right);
        }

        // Visit nodes in the queue order
        lvlOrdTrav(dequeue());
    }
}

int isBalanced(struct node *temp) {
    // If tree is empty
    if (temp == NULL) {
        return 1;
    }

    // Balance factor = Height of left subtree - Height of right subtree
    int balFac = subTreeHt(temp->left) - subTreeHt(temp->right);

    // Current node is balanced & child nodes are balanced
    if (balFac <= 1 && balFac >= -1 && isBalanced(temp->left) &&
isBalanced(temp->right)) {
        return 1;
    }

    // Not balanced
    return 0;
}

int main() {
    // Print menu
    printf("*** Implementation of BST Using Linked Lists *** \n");
    printf("** Options Menu ** \n");
    printf("01. Insert. \n");
    printf("02. Delete. \n");
    printf("03. Find. \n");
    printf("04. Pre-order traversal. \n");
    printf("05. In-order traversal. \n");
    printf("06. Post-order traversal. \n");
    printf("07. Level-order traversal. \n");
    printf("08. Height of Tree. \n");
    printf("09. Display BST. \n");
    printf("10. Check if BST is balanced. \n");
    printf("0. Exit. \n");

    int opt, elt;

    while (1) {
        // Input
        printf("\nEnter an option: ");
        scanf("%d", &opt);

        switch(opt) {
            case 0:
                exit(0);
            case 1:
                // Insert
                printf("Enter element: ");
                scanf("%d", &elt);
                if (find(elt, root) == 1) {
                    printf("Invalid input. Element exists in BST. \n");
                }
                else {
                    insert(elt, root);
                }
                break;
            case 2:
                // Delete
                printf("Enter element: ");
                scanf("%d", &elt);
                root = delete(elt, root);
                break;
            case 3:
                // Find element
                printf("Enter element: ");
                scanf("%d", &elt);
                find(elt, root) == 1 ? printf("\'%d\' found. \n", elt) :
printf("Element not found. \n");
                break;
            case 4:
                printf("Pre-order traversal: ");
                preOrdTrav(root);
                printf("\n");
                break;
            case 5:
                printf("In-order traversal: ");
                inOrdTrav(root);
                printf("\n");
                break;
            case 6:
                printf("Post-order traversal: ");
                postOrdTrav(root);
                printf("\n");
                break;
            case 7:
                printf("Level-order traversal: ");
                lvlOrdTrav(root);
                printf("\n");
                break;
            case 8:
                printf("Height of tree is %d. \n", subTreeHt(root));
                break;
            case 9:
                dispTree(root, 0);
                break;
            case 10:
                isBalanced(root) == 1 ? printf("The tree is balanced. \n")
: printf("The tree is not balanced. \n");
                break;
            default:
                printf("Invalid option. Please enter a valid option (e.g.,
4). \n");
        }
    }

    return 0;
}

/*
Alternate Level Order Traversal function:

void currLevelTraversal(int lvl, struct node *temp) {
    // Level greater than sub-tree height -> Invalid
    if (lvl > subTreeHt(temp)) {
        return;
    }
    // Tree is empty
    if (root == NULL) {
        return;
    }
    // Level reached (or) Root Node = Level 0
    if (lvl == 0) {
        printf("%d ", temp->data);
    }
    if (lvl > 0) {
        currLevelTraversal(lvl - 1, temp->left);
        currLevelTraversal(lvl - 1, temp->right);
    }
 }

 // Level order traversal
 void lvlOrdTrav(struct node *temp) {
    for (int i = 0; i <= subTreeHt(temp); i++) {
        printf("Level %d: ", i);
        currLevelTraversal(i, temp);
        printf("\n");
    }
 }

*/
```

RESULT:

Ex. No: 9	IMPLEMENTATION OF LINEAR SEARCH AND BINARY SEARCH
28 / 04 / 22	

AIM:

To implement Linear search and Binary search algorithms.

THEORY:

Linear search – iterate through the array until the required element is found. It runs in linear time – O(N).

Binary search – search a sorted array starting from the middle element and eliminating half the array in each iteration (i.e.) If ‘x’ is the element to be found, check if ‘x’ is the middle element. If so, the answer is at hand. If ‘x’ is smaller than the middle element, apply the same strategy to the sorted subarray to the left of the middle element; likewise, if ‘x’ is larger than the middle element, look to the right half. This algorithm takes advantage of the fact that the list is sorted and its time complexity is O(log$_2$(N)).

Program 1:

```
/*
Implementation of Linear Search Algorithm
*/

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

// For runtime calculation
#define BILLION 1000000000
// For random number upper limit
#define UPPER_LIMIT 10000

int linearSearch(int num, int *ptrArr, int arrSize) {
    int i;

    for (i = 0; i < arrSize; i++) {
        // Element found
        if (*(ptrArr + i) == num) {
            return 1;
        }
    }

    // Element not found
    return 0;
}
```

```c
void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

float calcRunTime(struct timespec start, struct timespec end) {
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1 + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION));
    }

    return runTime;
}

int main() {
    int num, arrSize;
    int i;

    // Input
    printf("Enter number array size: ");
    scanf("%d", &arrSize);

    printf("Enter a number: ");
    scanf("%d", &num);

    // Generate a random array with elements from 1 - 10000
    int arr[arrSize];
    int *ptrArr = arr;

    printf("\n* Generating random array with %d elements * ", arrSize);

    for (i = 0; i < arrSize; i++) {
        arr[i] = random() % 10000 + 1;    // '%' > '+' - Precedence
    }

    printf("- Done. \n");

    // Record clock time (start)
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Search for number in array + Output
    linearSearch(num, ptrArr, arrSize) == 0 ? printf("\nNumber not found in
array. \n") : printf("\n\'%d\' found in array. \n", num);
```

```
    // Record clock time (end)
    clock_gettime(CLOCK_REALTIME, &end);

    // Execution time = Start - End time
    float runTime = calcRunTime(start, end);

    // Output
    printf("\nLinear search time taken is %f s. \n", runTime);

    return 0;
}
```

Program 2:

```c
/*
Implementation of Binary Search Algorithm
*/

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

// For runtime calculation
#define BILLION 1000000000
// For random number upper limit
#define UPPER_LIMIT 10000

int binarySearch(int elt, int *ptrArr, int low, int up) {
    int mid;

    // Find middle element
    if ((low + up) % 2 == 0) {
        mid = (low + up) / 2;
    }
    else {
        mid = (low + up + 1) / 2;
    }

    /*
    // Print Values
    printf("Low = %d, Mid = %d, Up = %d: ", low, mid, up);

    for (int i = low; i < up + 1; i++) {
    printf("%d ", *(ptrArr + i));
    }

    printf("\n");

    // Note: If values are printed, runtime may be more than actual
runtime.
    */

    // Array completely searched. Element not found.
    if (low > up) {
        return 0;
    }
    // Element found at middle
    if (*(ptrArr + mid) == elt) {
        return 1;
    }
    else {
        // Element is smaller than mid element, find element in left half
of array
        if (elt < *(ptrArr + mid)) {
            up = mid - 1;
            return binarySearch(elt, ptrArr,  low, up);
        }
        // Element is larger than mid element, find element in right half
of array
        else {
            low = mid + 1;
            return binarySearch(elt, ptrArr,  low, up);
        }
    }

    // Element not found
    return 0;
}

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

float calcRunTime(struct timespec start, struct timespec end) {
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1 + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION));
    }

    return runTime;
}

int main() {
    int num, arrSize;
    int i, j;

    // Input
    printf("Enter number array size: ");
    scanf("%d", &arrSize);

    printf("Enter a number: ");
    scanf("%d", &num);

    // Generate a random array with elements from 1 - 10000
    int arr[arrSize];
    int *ptrArr = arr;    // Pointer to array

    printf("\n* Generating random array with %d elements * ", arrSize);
```

```c
    for (i = 0; i < arrSize; i++) {
        arr[i] = random() % 10000 + 1;    // '%' > '+' - Precedence
    }

    printf("- Done. \n");

    // Lower & Upper bounds of Array
    int low = 0, up = arrSize - 1;

    // Sort array
    printf("* Sorting array * ");

    for (i = 0; i < arrSize; i++) {

        for (j = 0; j < arrSize - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }

    printf("- Done. \n");

    // Record clock time (start)
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Search for number in array + Output
    binarySearch(num, ptrArr, low, up) == 0 ? printf("\nNumber not found in
array. \n") : printf("\n\'%d\' found in array. \n", num);

    // Record clock time (end)
    clock_gettime(CLOCK_REALTIME, &end);

    // Execution time = Start - End time
    float runTime = calcRunTime(start, end);

    // Output
    printf("\nBinary search time taken is %f s. \n", runTime);

    return 0;
}

/*
 In Bubble Sort,

 Every iteration, it will sort 1 element to the right end of the array.
 So, do not check
 last element after 1st iteration,
 last 2 elements after 2nd iteration,
 and so on...
 'j < arrSize - 1' -> Will check sorted part of array too = More time
complexity.
 'j < arrSize - i - 1' -> Will avoid checking last 'i' elements in array =
Better.

Note:
- Bubble sort is not considered in Binary Search.
- Time Complexity of Binary Search is 'O(log N)'.
*/
```

RESULT:

Ex. No: 10	IMPLEMENATION OF BUBBLE SORT, INSERTION SORT AND SELECTION SORT
28 / 04 / 22	

AIM:

To implement Bubble sort, Insertion sort and Selection sort algorithms.

THEORY:

Bubble sort (also known as sinking sort) is a simple sorting algorithm that repeatedly iterates through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. In every complete pass through the array, the smallest / largest element (in the unsorted part) is pushed to one end of the array (sorted part).

Insertion sort inserts an element from the unsorted part of the array, into the right place in the sorted part of the array.

Selection sort selects the smallest / largest element in the array and swaps it with the element in the first index position of the array. It then selects the second smallest / largest element and swaps it with the second index position of array, and so on.

Program 1:

```
/*
Implementation of Bubble Sort Algorithm
*/

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

// For runtime calculation
#define BILLION 1000000000
// For random number upper limit
#define UPPER_LIMIT 10000

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

void bubbleSort(int *ptrArr, int arrSize) {
    int i, j;

    for (i = 0; i < arrSize; i++) {
        for (j = 0; j < arrSize - i - 1; j++) {
            if (*(ptrArr + j) > *(ptrArr + j + 1)) {
                swap(ptrArr + j, ptrArr + j + 1);
            }
        }
    }
}

float calcRunTime(struct timespec start, struct timespec end) {
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1 + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION));
    }

    return runTime;
}

int main() {
    int arrSize;
    int i;

    // Input
    printf("Enter number array size: ");
    scanf("%d", &arrSize);

    // Generate a random array with elements from 1 - 10000
    int arr[arrSize];
    int *ptrArr = arr;    // Pointer to array

    printf("\n* Generating random array with %d elements * ", arrSize);

    for (i = 0; i < arrSize; i++) {
        arr[i] = random() % 10000 + 1;    // '%' > '+' - Precedence
    }

    printf("- Done. \n");

    printf("* Sorting array * ");

    // Record clock time (start)
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Bubble sort
    bubbleSort(ptrArr, arrSize);
    // Record clock time (end)
    clock_gettime(CLOCK_REALTIME, &end);

    printf("- Done. \n");

    // Execution time = Start - End time
    float runTime = calcRunTime(start, end);

    // Output
    printf("\nBubble sort time taken is %f s. \n", runTime);

    return 0;
}

/*
Note:
- *ptrArr = arr[0]
- *(ptrArr + j) = arr[j]
*/
```

Program 2:

```c
/*
Implementation of Insertion Sort Algorithm
*/

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

// For runtime calculation
#define BILLION 1000000000
// For random number upper limit
#define UPPER_LIMIT 10000

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

void insertionSort(int *ptrArr, int arrSize) {
    int i, j, num;

    for (i = 1; i < arrSize; i++) {
        j = i;
        num = *(ptrArr + j);

        /*
        For ascending order sorting:
         - Insert 'num' if element to the left (in sorted array) is smaller
           or if it has reached the left end of sorted list (i.e.) it is
the smallest number.
        */

        while (num < *(ptrArr + j - 1) && j != -1) {
            *(ptrArr + j) = *(ptrArr + j - 1);
            j--;
        }

        *(ptrArr + j) = num;
    }
}

float calcRunTime(struct timespec start, struct timespec end) {
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1 + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION));
    }

    return runTime;
}

int main() {
    int arrSize;
    int i;

    // Input
    printf("Enter number array size: ");
    scanf("%d", &arrSize);

    // Generate a random array with elements from 1 - 10000
    int arr[arrSize];
    int *ptrArr = arr;    // Pointer to array

    printf("\n* Generating random array with %d elements * ", arrSize);

    for (i = 0; i < arrSize; i++) {
        arr[i] = random() % 10000 + 1;    // '%' > '+' - Precedence
    }

    printf("- Done. \n");

    printf("* Sorting array * ");

    // Record clock time (start)
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Insertion sort
    insertionSort(ptrArr, arrSize);

    // Record clock time (end)
    clock_gettime(CLOCK_REALTIME, &end);

    printf("- Done. \n");

    // Execution time = Start - End time
    float runTime = calcRunTime(start, end);

    // Output
    printf("\nInsertion sort time taken is %f s. \n", runTime);

    return 0;
}

/*
Note:
```

```
- Insertion sort is a natural sorting algorithm used by humans.
 Procedure:
 i)   Pick up a number from unsorted array,
 ii)  Insert it into the right place in your sorted array,
 iii) Repeat.
*/
```

Program 3:

```c
/*
Implementation of Selection Sort Algorithm
*/

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

// For runtime calculation
#define BILLION 1000000000
// For random number upper limit
#define UPPER_LIMIT 10000

void swap(int *ptr1, int *ptr2) {
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

void selectionSort(int *ptrArr, int arrSize) {
    int i, j;
    int smallest, indSmallest;

    for (i = 0; i < arrSize; i++) {
        indSmallest = i;
        smallest = *(ptrArr + i);

        // Find smallest element and its index
        for (j = i; j < arrSize; j++) {
            if (*(ptrArr + j) < smallest) {
                indSmallest = j;
                smallest = *(ptrArr + j);
            }
        }

        swap(ptrArr + i, ptrArr + indSmallest);
    }
}

float calcRunTime(struct timespec start, struct timespec end) {
    long double runTime;

    if (end.tv_nsec - start.tv_nsec >= 0) {
        runTime = (end.tv_sec - start.tv_sec) + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION);
    }
    else {
        runTime = (end.tv_sec - start.tv_sec - 1 + ((float)(end.tv_nsec -
start.tv_nsec) / BILLION));
    }
    return runTime;
}

int main() {
    int arrSize;
    int i;

    // Input
    printf("Enter number array size: ");
    scanf("%d", &arrSize);

    // Generate a random array with elements from 1 - 10000
    int arr[arrSize];
    int *ptrArr = arr;    // Pointer to array

    printf("\n* Generating random array with %d elements * ", arrSize);

    for (i = 0; i < arrSize; i++) {
        arr[i] = random() % 10000 + 1;    // '%' > '+' - Precedence
    }

    printf("- Done. \n");

    printf("* Sorting array * ");

    // Record clock time (start)
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Selection sort
    selectionSort(ptrArr, arrSize);

    // Record clock time (end)
    clock_gettime(CLOCK_REALTIME, &end);

    printf("- Done. \n");

    // Execution time = Start - End time
    float runTime = calcRunTime(start, end);

    // Output
    printf("\nSelection sort time taken is %f s. \n", runTime);

    return 0;
}
```

RESULT:

Ex. No: 11	GRAPH TRAVERSALS
05 / 05 / 22	

AIM:

To implement depth-first search (DFS) and breadth-first search (BFS) algorithms.

THEORY:

In the breadth-first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level (descendants). In the depth-first traversal, we process all the descendants of a vertex before we move to the adjacent vertices. Both algorithms print all the elements in the graph. Queue data structure is used for BFS.

Program 1:

```
/*
 Adjacency list representation of graph and depth-first search.
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define NEW_LINE printf("\n");

typedef struct Node {
    int vertex;
    // Pointer to next node
    struct Node *next;    // (same as) node *next;
} node;

typedef struct Graph {
    // Adjacency list - Array of pointers to node
    node **adjList;
    int *visited;
    int numOfVertices;
} graph;

graph *create_graph(int numOfVertices) {
    // Allocate space for graph
    graph *g = (graph *) malloc(sizeof(graph));

    // Allocate space for an array of linked lists (adjacency list)
    g->adjList = malloc(numOfVertices * sizeof(node *));
    // Allocate space for a 'visited' array
    g->visited = malloc(numOfVertices * sizeof(int));
    // Initialise values to 'NULL' & 'not visited' respectively
    for (int i = 0; i < numOfVertices; i++) {
        g->adjList[i] = NULL;
        g->visited[i] = 0;
    }

    g->numOfVertices = numOfVertices;

    return g;
}

// Check if all vertices in graph have been visited
int all_nodes_visited(graph *g) {
    for (int i = 0; i < g->numOfVertices; i++) {
        // If 'not visited' vertex found
        if (g->visited[i] == 0) {
            return 0;
        }
    }

    return 1;
}

// Set all vertices to 'not visited'
void unvisit_all_nodes(graph *g) {
    for (int i = 0; i < g->numOfVertices; i++) {
        g->visited[i] = 0;
    }
}

void print_adjacency_list(graph *g) {
    printf("[ \n");

    for (int i = 0; i < g->numOfVertices; i++) {
        node *temp = g->adjList[i];

        // Print vertex
        if (temp == NULL) {
            printf("|%d| -> / \n", i);
        }
        else {
            printf("|%d| -> ", i);
            // Print adjacent vertices
            while (temp->next != NULL) {
                printf("%d -> ", temp->vertex);
                temp = temp->next;
            }
            printf("%d -> \\ \n", temp->vertex);
        }
    }

    unvisit_all_nodes(g);
    printf("] \n");
}

void add_edge(graph *g, int src, int dst) {
    // If edge already exists
    if (g->adjList[src] != NULL) {
        node *temp = g->adjList[src];
        while (temp != NULL) {
            if (temp->vertex == dst) {
                printf("Edge already exists. \n");
                return;
            }
            temp = temp->next;
        }
    }
    // If source edge has no existing edges
    if (g->adjList[src] == NULL) {
        g->adjList[src] = (node *) malloc(sizeof(node));
        g->adjList[src]->vertex = dst;
        g->adjList[src]->next = NULL;
    }
    // If source edge has edges
    else {
        node *temp = g->adjList[src];

        // Move to last vertex in adjacency list
        while (temp->next != NULL) {
            temp = temp->next;
        }

        // Add vertex at end
        temp->next = (node *) malloc(sizeof(node));
        temp = temp->next;
        temp->vertex = dst;
        temp->next = NULL;

    }
}

void depth_first_search(graph *g, int start) {
    // If all vertices are visited, un-visit them and quit
    if (all_nodes_visited(g) == 1) {
        unvisit_all_nodes(g);
        return;
    }

    // Assign pointer to first adjacent vertex
    node *temp = g->adjList[start];

    // Mark as visited
    g->visited[start] = 1;
    printf("%d ", start);
```

```c
    while (temp->next != NULL) {
        // If adjacent vertex is not visited
        if (g->visited[temp->vertex] == 0) {
            // DFS form that vertex
            depth_first_search(g, temp->vertex);
        }
        // Move to next vertex
        temp = temp->next;
    }

    // Last adjacent vertex
    if (g->visited[temp->vertex] == 0) {
        // DFS form that vertex
        depth_first_search(g, temp->vertex);
    }
}

int main() {
    printf("*** GRAPH IMPLEMENTATION USING ADJACENCY LIST *** \n");

    // Input
    int numOfVertices;
    printf("\nEnter number of vertices: ");
    scanf("%d", &numOfVertices);

    // Adjacency Matrix representation of Graphs
    graph *g = create_graph(numOfVertices);

    // Options menu
    printf("\n** Options Menu ** \n");
    printf("1. Add edge. \n");
    printf("2. Depth-first search. \n");
    printf("3. Display graph. \n");
    printf("0. Exit. \n");

    int opt;
    int src, dst;
    int start;

    while (1) {
        printf("\nEnter option: ");
        scanf("%d", &opt);

        switch (opt) {
                // Free allocated memory & end program
            case 0:
                free(g);
                printf("Program ended. \n");
                exit(0);
                // Add edge
            case 1:
                printf("Enter source: ");
                scanf("%d", &src);
                printf("Enter destination: ");
                scanf("%d", &dst);

                if (src >= g->numOfVertices || dst > g->numOfVertices ||
src < 0 || dst < 0) {
                    printf("Invalid vertex/vertices. \n");
                }
                else {
                    add_edge(g, src, dst);
                }

                break;
                // DFS
            case 2:
                printf("Enter starting vertex: ");
                scanf("%d", &start);

                if (start >= g->numOfVertices || start < 0) {
                    printf("Invalid vertex. \n");
                }
                else {
                    printf("DFS: ");
                    depth_first_search(g, start);
                    unvisit_all_nodes(g);
                    NEW_LINE
                }
                break;
                // Print adjacency list
            case 3:
                print_adjacency_list(g);
                break;
                // Invalid input
            default:
                printf("Invalid input. Please enter a valid input. \n");
        }
    }

    return 0;
}

/*
 Simple directed connected graph:
 add_edge(g, 1, 2);
 add_edge(g, 2, 1);
 add_edge(g, 1, 3);
 add_edge(g, 3, 0);
 add_edge(g, 0, 1);
 */
```

Program 2:

```
/*
Adjacency matrix representation of graph and breadth-first search.
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define QUEUE_MAX 10
#define NEW_LINE printf("\n");
#define PRINT_QUEUE printf("["); for (int i = 0; i < QUEUE_MAX; i++) printf("%d ",
queue[i]); printf("]\n");

// Queue
int queue[QUEUE_MAX];
int front = -1;
int rear = -1;

void enqueue(int vertex) {
   if (rear == QUEUE_MAX - 1) {
      printf("Queue full. \n");
      return;
   }
   else if (front == -1) {
      front = 0;
      rear = 0;
   }
   else {
      rear++;
   }

   queue[rear] = vertex;
}

int dequeue(void) {
   if (front == -1) {
      printf("Queue empty. \n");
      return '\0';
   }
   else if (front == rear) {
      int retVal = queue[front];
      front = -1;
      rear = -1;

      return retVal;
   }
   else {
        int retVal = queue[front];
        front++;

        return retVal;
    }
}

int isEmpty(void) {
    if (front == -1) {
        return 1;
    }
    else {
        return 0;
    }
}

typedef struct Graph {
    int numOfVertices;
    // Double pointer acts as an array
    int **vertex;
    // To check if a node is visited or not
    int *visited;
} graph;

graph *create_graph(int numOfVertices) {
    // Allocate space for graph
    graph *g = (graph *) malloc(sizeof(graph));

    // Allocate space for an array of arrays ('vertex' array)
    g->vertex = calloc(numOfVertices, sizeof(int *));
    // 'calloc(<number_of_blocks>, <size_of_blocks>)' initialises all memory locations to '0'.

    // Assign a pointer to an array in every memory location in 'vertex' array
    for (int i = 0; i < numOfVertices; i++) {
        g->vertex[i] = (int *)calloc(numOfVertices, sizeof(int));
    }

    // Initialise array with '0's
    g->visited = calloc(numOfVertices, sizeof(int));

    g->numOfVertices = numOfVertices;

    return g;
}

void print_adjacency_matrix(graph *g) {
    printf("[ \n");
    printf(" \t");
    // Print column index
    for (int i = 0; i < g->numOfVertices; i++) {
        printf("%d \t", i);
    }

    NEW_LINE

    for (int i = 0; i < g->numOfVertices; i++) {
        // Print row index
        printf("%d |\t", i);

        for (int j = 0; j < g-> numOfVertices; j++) {
            // Print values
            printf("%d \t", g->vertex[i][j]);
        }
        printf("\n");
    }

    printf("] \n");
}

void add_edge(graph *g, int prev, int next) {
    g->vertex[prev][next] = 1;
}

void remove_edge(graph *g, int prev, int next) {
    g->vertex[prev][next] = 0;
}

// Check if all nodes in graph have been visited
int all_nodes_visited(graph *g) {
    for (int i = 0; i < g->numOfVertices; i++) {
        if (g->visited[i] == 0) {
            return 0;
        }
    }
    return 1;
}

// Set all nodes to 'not visited'
void unvisit_all_nodes(graph *g) {
    for (int i = 0; i < g->numOfVertices; i++) {
        g->visited[i] = 0;
    }
}

void breadth_first_search(graph *g, int start) {
    printf("BFS: ");

    // Mark vertex as visited
    g->visited[start] = 1;
    // Enqueue vertex
    enqueue(start);

    // Until all vertices are visited
    while (!all_nodes_visited(g)) {
        // Enqueue all unvisited ajacent vertices & mark them as 'visited'
        for (int i = 0; i < g->numOfVertices; i++) {
            if (g->vertex[start][i] == 1) {
                if (g->visited[i] == 0) {
                    g->visited[i] = 1;
                    enqueue(i);
                }
            }
        }
        // Repeat the same for front vertex in queue
        start = dequeue();
        printf("%d ", start);
    }

    unvisit_all_nodes(g);

    // Print remaining vertices in queue
    while (!isEmpty()) {
        int i = dequeue();
        printf("%d ", i);
    }

    printf("\n");
}

int main() {
    printf("*** GRAPH IMPLEMENTATION USING ADJACENCY MATRIX *** \n");

    // Input
    int numOfVertices;
    printf("\nEnter number of vertices: ");
    scanf("%d", &numOfVertices);

    // Adjacency Matrix representation of Graphs
    graph *g = create_graph(numOfVertices);

    // Options menu
    printf("\n** Options Menu ** \n");
    printf("1. Add edge. \n");
printf("2. Remove edge. \n");
printf("3. Breadth-first search. \n");
printf("4. Display graph. \n");
printf("0. Exit. \n");

int opt;
int src, dst;
int start;

while (1) {
   printf("\nEnter option: ");
   scanf("%d", &opt);

   switch (opt) {
      case 0:
         // Free allocated memory
         free(g);
         printf("Program ended. \n");
         exit(0);
      case 1:
         printf("Enter source: ");
         scanf("%d", &src);
         printf("Enter destination: ");
         scanf("%d", &dst);

         if (src > g->numOfVertices || dst > numOfVertices) {
            printf("Invalid vertex/vertices. \n");
         }
         else {
            add_edge(g, src, dst);
         }

         break;
      case 2:
         printf("Enter source: ");
         scanf("%d", &src);
         printf("Enter destination: ");
         scanf("%d", &dst);

         if (src > g->numOfVertices || dst > g->numOfVertices) {
            printf("Invalid vertex/vertices. \n");
         }
         else {
            remove_edge(g, src, dst);
         }

         break;
      case 3:
            printf("Enter starting vertex: ");
            scanf("%d", &start);

            if (start > g->numOfVertices) {
               printf("Invalid vertex. \n");
            }
            else {
               breadth_first_search(g, start);
            }

            break;
         case 4:
            print_adjacency_matrix(g);
            break;
         default:
            printf("Invalid input. Please enter a valid input. \n");
      }
   }

   return 0;
}

/*
Simple directed connected graph:
 add_edge(g, 1, 2);
 add_edge(g, 2, 1);
 add_edge(g, 1, 3);
 add_edge(g, 3, 0);
 add_edge(g, 0, 1);

Assumption:
   - To use BFS (breadth-first search), the graph must be connected (i.e.) there must be NO
disconnected vertex.

TYPEDEF
   Syntax: typedef <data_type/data_structure> <new_name>;
   E.g. typedef long long int lint; ('lint', the last word, is the new name)
   So, instead of 'long long int multiple;' use 'lint multiple;'
*/
```

RESULT:

Ex. No: 12	MINIMUM SPANNING TREE
12 / 05 / 22	

AIM:

To implement Prim's and Kruskal's minimum spanning tree algorithms.

THEORY:

A spanning tree is a linked and undirected subgraph that connects all vertices. Many spanning trees can exist in a graph. The minimum spanning tree (MST) has the least weight of all other spanning trees. Costs are assigned to each edge of spanning trees and the sum is the cost assigned the tree. A minimum spanning tree has 'V – 1' edges ('V' – number of vertices).

Prim's algorithm:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum cost edge from found edges and add it to the tree.
3. Keep repeating step 2 until the minimum spanning tree is formed.

Kruskal's algorithm:

1. All of the edges should be arranged in a non-descending sequence of weight.
2. Choose the smallest edge. This edge is included if a cycle is not formed.
3. Step 2 should be performed until the spanning tree has (V-1) edges.

Program 1:

```
/*
Prim's Algorithm Implementation
*/

#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define INF 9999999
// Number of vertices in graph
#define V 5

// Create a 2D array of size - 5 x 5
// Adjacency matrix to represent graph
int G[V][V] = {
   {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};

int main() {
  int numOfEdges;  // Number of edges

  // Create a array to track selected vertex
  // Selected will become true otherwise false
  int selected[V];

  // Set selected false initially
  memset(selected, false, sizeof(selected));

  // Set number of edge to 0
  numOfEdges = 0;

  // The number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in the
  // graph

  // Choose '0'th vertex and make it true
  selected[0] = true;

  int x;  //  Row number
  int y;  //  Col number

  // Print for edge and weight
  printf("Edge : Weight\n");

  while (numOfEdges < V - 1) {
    // For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // If the vertex is already in the set S, discard it otherwise
    // choose another vertex nearest to selected vertex at step 1.

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {  // Not in selected and there is an edge
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
```

```
                    y = j;
                  }
                }
              }
            }
          }

          printf("%d - %d : %d\n", x, y, G[x][y]);
          selected[y] = true;
          numOfEdges++;
        }

        return 0;
      }
```

Program 2:

```
/*
 Kruskal's Algorithm
*/

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E){
    struct Graph* graph = (struct Graph*)(malloc(sizeof(struct Graph)));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(sizeof( struct Edge)*E);
    return graph;
}

struct subset {
    int parent;
    int rank;
};

int find(struct subset subsets[], int i){
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y){
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else{
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
int myComp(const void* a, const void* b){
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;

    return a1->weight > b1->weight;
}

void kruskalMST(struct Graph* graph){
    int V = graph->V;
    struct Edge
    result[V];
    int e = 0;
    int i = 0;

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);
    struct subset* subsets = (struct subset*)malloc(V * sizeof(struct
subset));

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edge[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST: \n");
    int minimumCost = 0;
    printf("\n");

    for (i = 0; i < e; ++i){
        printf("%d -- %d == %d\n", result[i].src,
               result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }

    printf("Minimum Cost Spanning tree: %d", minimumCost);
    return;
}
int main(){
    int V = 24;
    int E = 25;
    struct Graph* graph = createGraph(V, E);
    graph->edge[0].src = 20;
    graph->edge[0].dest = 21;
    graph->edge[0].weight = 30;
    graph->edge[1].src = 20;
    graph->edge[1].dest = 22;
    graph->edge[1].weight = 26;
    graph->edge[2].src = 20;
    graph->edge[2].dest = 23;
    graph->edge[2].weight = 25;
    graph->edge[3].src = 21;
    graph->edge[3].dest = 23;
    graph->edge[3].weight = 35;
    graph->edge[4].src = 22;
    graph->edge[4].dest = 23;
    graph->edge[4].weight = 24;

    kruskalMST(graph);
    printf("\n");
    return 0;
}
```

RESULT: