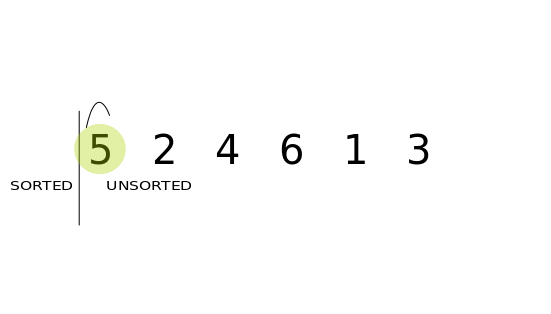
NAME: RITIK KADOLE

SCHOLAR NO: 202120130

ROLL NO: 85

**INSERTION SORT**



*// PROGRAM FOR INSERTIONT SORT BY FUNCTION*

#include <stdio.h>

#include <conio.h>

void sort(int a[], int l);

void main()

{

*// in insertion sort we start from the second element considering first as sorted*

*// we put elemnt from second to their right position*

*// making an array*

  int l, a[] = {1, 7, 5, 3, 9, 6, 3, 6, 4, 8, 0, 4, 5, 3,5,2,34,45,54,5454,54,6,6};

*// calculating the size of array*

  l = sizeof(a) / sizeof(a[0]); *// l=size*

*// printing unsorted array*

    printf("\n\n\*\*\* THIS IS THE UNSORTED ARRAY \*\*\* \n");

  for (int i = 0; i < l; i++)

  {

    printf(" %d ", a[i]);

  }

  printf(" \n\a");

*// we will call a funtioon \*accepts and returns=array pointer*

  sort(a,l);

*//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*// printing the sorted array after calling the function*

  printf("\*\*\* SORTED by INSERTION SORT Algorithm \*\*\* \n");

  for (int i = 0; i < l; i++)

  {

    printf(" %d ", a[i]);

  }

    printf("\n\n\n");

  getch();

}

*//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

void sort(int a[], int l)

{

*// we have perform the insertion sort here*

*// total size of the array is l here 1 up than the index*

  int temp;

  for (int i = 1; i < l; i++)

  {

     temp = a[i];

     int count=0;

    for (int j = i; j >= 0; j--)

    {

       if (a[j] > temp)

      {

*// printf("\n %d <--> %d ", i, j);*

        a[j+1]=a[j];

       count++;

      }

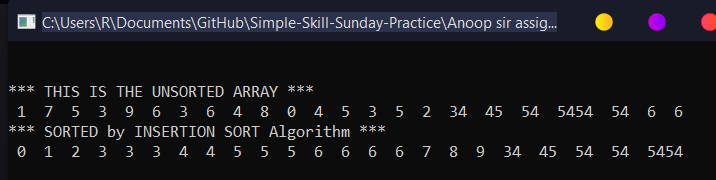
    }

    a[i-count]=temp;

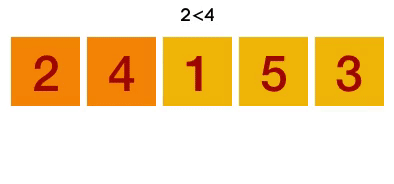
*// printf("\n\_\_\_\_\_\_\n");*

  }

}



**BUBBLE SORT**



*// SORTING THE ARRAY WITH THE BUBBLE SORT ALGORITHM*

#include <stdio.h>

#include <conio.h>

void sort(int a[],int l);

void main()

{

    int a[] = {3, 8, 0, 7, 6, 5, 6, 7, 6, 4, 6, 7, 9, 6};

    int l = sizeof(a) / sizeof(a[0]);

    printf("\n\n\*\*\* THIS IS THE UNSORTED ARRAY \*\*\*\n\n");

    for (int i = 0; i < l; i++)

    {

        printf(" %d ", a[i]);

    }

sort(a,l);  *// CLALLING THE SORT FUNCTION*

    printf("\n\n\*\*\* SORTED THIS ARRAY WITH BUBBLE SORT ALGORITHM \*\*\*\n\n");

    for (int i = 0; i < l; i++)

    {

        printf(" %d ", a[i]);

    }

    getch();

}

void sort(int  a[],int  l)

{

int temp;

    for (int i = 0; i <= l; i++)

    {

        for (int j = 0; j <l; j++)

        {

            if (a[j] > a[j + 1])

            {

                temp = a[j];

                a[j] = a[j + 1];

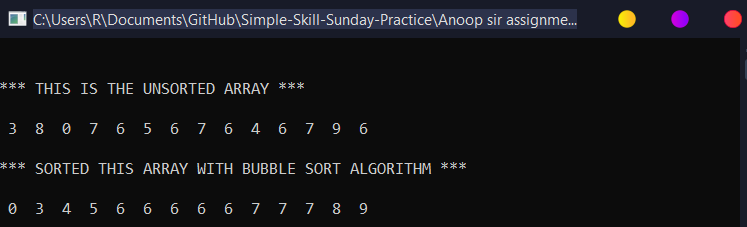
                a[j + 1] = temp;

            }

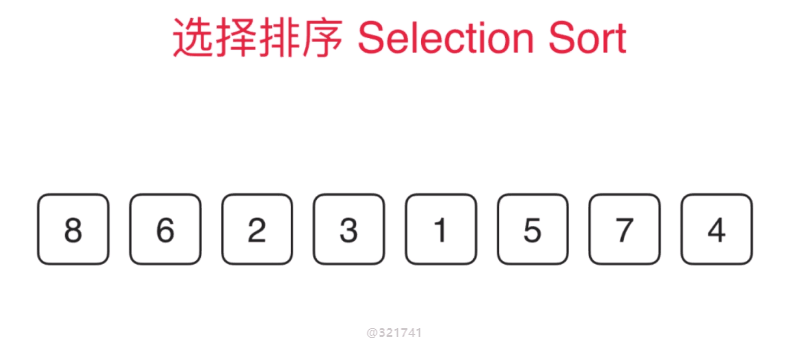
        }

    }

}



**SELECTION SORT**



*// PROGRAM TO IMPLEMENT THE SELECTION SORT*

#include <stdio.h>

#include <conio.h>

void sort(int a[], int l);

void main()

{

*// Making an array*

    int a[] = {2, 7, 4, 6, 0, 4, 6, 1, 5, 9, 3};

*// calculating the size of the array*

    int l = sizeof(a) / sizeof(a[0]);

*// l = size & l-1= last index*

    printf("\n");

    for (int i = 0; i < l; i++)

    {

        printf(" %d ", a[i]);

    }

    sort(a, l);

    printf("\n");

    for (int i = 0; i < l; i++)

    {

        printf(" %d ", a[i]);

    }

    getch();

}

void sort(int a[], int l)

{

    int temp = 0;

    for (int i = 0; i < l; i++)

    {

*// temp=a[i+1];*

        for (int j = i; j < l; j++)

        {

            if (a[j] < a[i])

            {

                temp = a[j];

                a[j] = a[i];

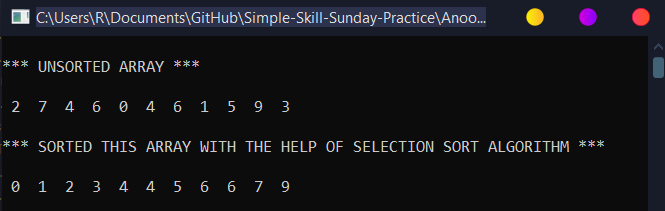
                a[i] = temp;

            }

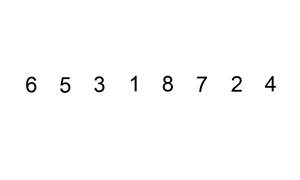
        }

    }

}



**SHELL SORT**



*// PROGRAM TO IMPLEMENTATION OF LINEAR AND BINARY SEARCH*

#include <stdio.h>

#include <conio.h>

void main()

{

    int a[] = {3, 8, 4, 6, 16, 17, 8, 16, 19, 10, 8, 9};

    int n=0, l = sizeof(a) / sizeof(int);

   printf("\n");

    for ( int i=0 ; i<l;i++)

{

    printf(" %d ",a[i]);

}

int i, j, increment, tmp;

    for(increment = l/2; increment > 0; increment /= 2)

    {

        for(i = increment; i < l; i++)

        {

            tmp = a[i];

            for(j = i; j >= increment; j -= increment)

            {

                if(tmp < a[j-increment])

                    a[j] = a[j-increment];

                else

                    break;

            }

            a[j] = tmp;

        }

    }

printf("\n\*\*\* AFTER SHELL SORT \*\*\*\n");

    for ( int i=0 ; i<l;i++)

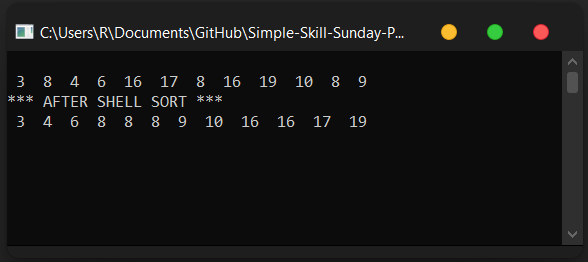
{

    printf(" %d ",a[i]);

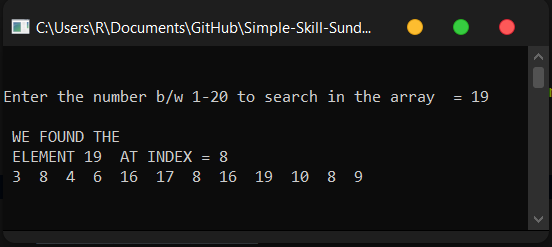
}

    getch();

}



**BINARY SEARCH and SORT**

****

*// PROGRAM TO IMPLEMENTATION OF LINEAR AND BINARY SEARCH*

#include <stdio.h>

#include <conio.h>

void main()

{

    int a[] = {3, 8, 4, 6, 16, 17, 8, 16, 19, 10, 8, 9};

    int n=0, l = sizeof(a) / sizeof(int);

   printf("\n\nEnter the number b/w 1-20 to search in the array  = ");

   scanf("%d", &n);

    int flag = 0;

*// LINEAR SEARCH*

    for (int i = 0; i < l; i++)

    {

        if (a[i] == n)

        {

            printf("\n WE FOUND THE \n ELEMENT %d  AT INDEX = %d \a", a[i], i);

            flag = 1;

            break;

        }

    }

    if (flag == 0)

    {

        printf("\n SORRY!!! Element %d not found", n);

    }

   printf("\n");

    for ( int i=0 ; i<l;i++)

{

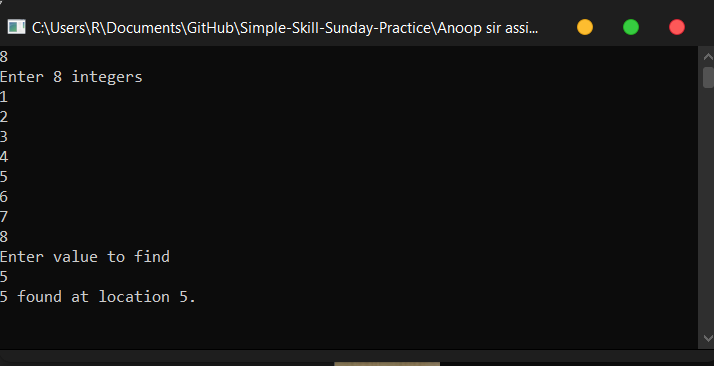
    printf(" %d ",a[i]);

}

    getch();

}

**BINARY SEARCH SORT**

****

#include <stdio.h>

#include <conio.h>

int main()

{

    int c, first, last, middle, n, search, array[100];

    printf("Enter number of elements\n");

    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)

        scanf("%d", &array[c]);

    printf("Enter value to find\n");

    scanf("%d", &search);

    first = 0;

    last = n - 1;

    middle = (first + last) / 2;

    while (first <= last)

    {

        if (array[middle] < search)

            first = middle + 1;

        else if (array[middle] == search)

        {

            printf("%d found at location %d.\n", search, middle + 1);

            break;

        }

        else

            last = middle - 1;

        middle = (first + last) / 2;

    }

    if (first > last)

        printf("Not found! %d isn't present in the list.\n", search);

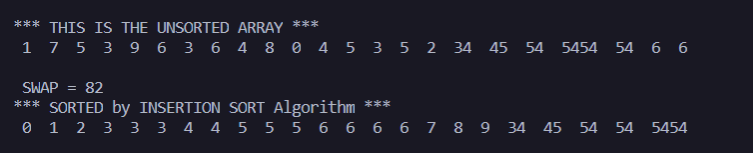
    getch();

    return 0;

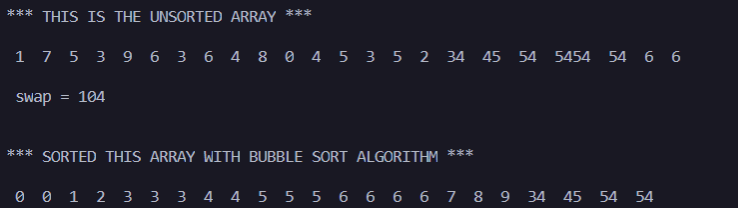
}

**Que 01 No of swapping**

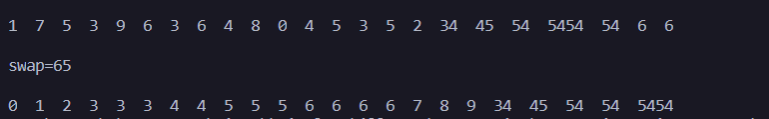
**Insertion sort = 82 swaps**

****

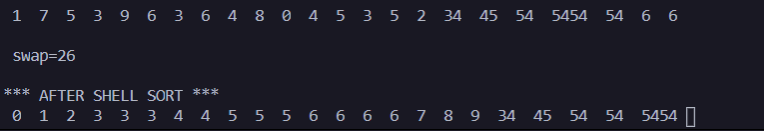
**Bubble sort = 104 swaps**

****

**Selection sort = 65 swaps**



**Shell sort = 26 swaps**



**Que 02 push and pop operation in stack**



#include<stdio.h>

void push(char element, char stack[], int \*top, int stackSize){

 if(\*top == -1){

  stack[stackSize - 1] = element;

  \*top = stackSize - 1;

 }

 else if(\*top == 0){

  printf("The stack is already full. \n");

 }

 else{

  stack[(\*top) - 1] = element;

  (\*top)--;

 }

}

void pop(char stack[], int \*top, int stackSize){

 if(\*top == -1){

   printf("The stack is empty. \n");

 }

 else{

  printf("Element popped: %c \n", stack[(\*top)]);

*// If the element popped was the last element in the stack*

*// then set top to -1 to show that the stack is empty*

  if((\*top) == stackSize - 1){

    (\*top) = -1;

  }

  else{

    (\*top)++;

  }

 }

}

int main() {

  int stackSize = 4;

  char stack[stackSize];

*// A negative index shows that the stack is empty*

  int top = -1;

  push('a', stack, &top, stackSize);

  printf("Element on top: %c\n", stack[top]);

  push('b',stack, &top, stackSize);

  printf("Element on top: %c\n", stack[top]);

  pop(stack, &top, stackSize);

  printf("Element on top: %c\n", stack[top]);

  pop(stack, &top, stackSize);

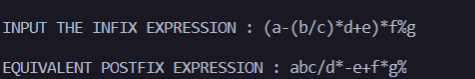
  printf("Top: %d\n", top);

  pop(stack, &top, stackSize);

  return 0;

}

**Que 03 program for infix to post fix**



**90 50 2 / \* - 1 + 2 \* 5 %**

**A = 90, B = 50, C = 2, D = 3, E = 1, F = 2, G = 5**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

*/\* Variable declarations \*/*

char infix\_string[20], postfix\_string[20];

int top;

int stack[20];

int pop();

int precedence(char symbol);

int isEmpty();

void infix\_to\_postfix();

int check\_space(char symbol);

void push(long int symbol);

int main()

{

    int count, length;

    char temp;

    top = -1;

    printf("\nINPUT THE INFIX EXPRESSION : ");

    scanf("%s", infix\_string);

    infix\_to\_postfix();

    printf("\nEQUIVALENT POSTFIX EXPRESSION : %s\n", postfix\_string);

    return 0;

}

*/\* Function to convert from infix to postfix \*/*

void infix\_to\_postfix()

{

    unsigned int count, temp = 0;

    char next;

    char symbol;

    for (count = 0; count < strlen(infix\_string); count++)

    {

        symbol = infix\_string[count]; *// Scanning the input expression*

        if (!check\_space(symbol))

        {

            switch (symbol)

            {

            case '(':

                push(symbol);

                break;

            case ')':

                while ((next = pop()) != '(') *// pop until '(' is encountered*

                {

                    postfix\_string[temp++] = next;

                }

                break;

            case '+':

            case '-':

            case '\*':

            case '/':

            case '%':

            case '^':

                while (!isEmpty() && precedence(stack[top]) >= precedence(symbol)) *// Check precedence and push the higher one*

                    postfix\_string[temp++] = pop();

                push(symbol);

                break;

            default:

                postfix\_string[temp++] = symbol;

            }

        }

    }

    while (!isEmpty())

    {

        postfix\_string[temp++] = pop();

    }

    postfix\_string[temp] = '\0';

}

*/\* Function to check precedence of operators \*/*

int precedence(char symbol)

{

    switch (symbol)

    {

    case '(':

        return 0;

    case '+':

    case '-':

        return 1;

    case '\*':

    case '/':

    case '%':

        return 2;

    case '^':

        return 3;

    default:

        return 0;

    }

}

int check\_space(char symbol)

{

    if (symbol == '\t' || symbol == ' ')

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void push(long int symbol)

{

    if (top > 20)

    {

        printf("Stack Overflow\n");

        exit(1);

    }

    top = top + 1;

    stack[top] = symbol; *// Push the symbol and make it as TOP*

}

int isEmpty()

{

    if (top == -1)

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int pop()

{

    if (isEmpty())

    {

        printf("Stack is Empty\n");

        exit(1);

    }

    return (stack[top--]); *// Pop the symbol and decrement TOP*

}

**Que 03 (c) program for infix to post fix**



*//Evaluation of Postfix expression by stack in C*

#include <stdio.h>

#include <ctype.h>

#define SIZE 50 */\* Size of Stack \*/*

int s[SIZE];

int top=-1; */\* Global declarations \*/*

int flag=0;

int pop()

{                      */\* Function for POP operation \*/*

  return(s[top--]);

}

int push(int elem)

{ */\* Function for PUSH operation \*/*

  if(flag==1){

    int num;

    num=pop();

    s[++top]=elem+10\*num;

  }

  else if(flag==0){

    s[++top]=elem;

    flag=1;

  }

}

int main()

{                         */\* Main Program \*/*

  char pofx[50],ch;

  int i=0,op1,op2;

  printf("Enter the Postfix Expression:");

  fgets(pofx,100,stdin);

  while( (ch=pofx[i++]) != '\n')

  {

    if(isdigit(ch)) push(ch-'0'); */\* Push the operand \*/*

    else if(ch==' ')

      flag=0;

    else

    {        */\* Operator,pop two  operands \*/*

      flag=0;

      op2=pop();

      op1=pop();

      switch(ch)

      {

        case '+':push(op1+op2);break;

        case '-':push(op1-op2);break;

        case '\*':push(op1\*op2);break;

        case '/':push(op1/op2);break;

        case '%':push(op1%op2);break;

        default:

                 printf("Input invalid ... give proper input\n");

                 return 0;

      }

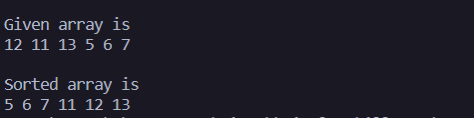
    }

  }

  printf("Result: %d\n",s[top]);

}

**Que 04 (a) program for infix to post fix**

****

#include <stdio.h>

#include <stdlib.h>

*// Merges two subarrays of arr[].*

*// First subarray is arr[l..m]*

*// Second subarray is arr[m+1..r]*

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;

*/\* create temp arrays \*/*

    int L[n1], R[n2];

*/\* Copy data to temp arrays L[] and R[] \*/*

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

*/\* Merge the temp arrays back into arr[l..r]\*/*

    i = 0; *// Initial index of first subarray*

    j = 0; *// Initial index of second subarray*

    k = l; *// Initial index of merged subarray*

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        }

        else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

*/\* Copy the remaining elements of L[], if there*

*are any \*/*

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

*/\* Copy the remaining elements of R[], if there*

*are any \*/*

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

*/\* l is for left index and r is right index of the*

*sub-array of arr to be sorted \*/*

void mergeSort(int arr[], int l, int r)

{

    if (l < r) {

*// Same as (l+r)/2, but avoids overflow for*

*// large l and h*

        int m = l + (r - l) / 2;

*// Sort first and second halves*

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

*/\* UTILITY FUNCTIONS \*/*

*/\* Function to print an array \*/*

void printArray(int A[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

*/\* Driver code \*/*

int main()

{

    int arr[] = { 12, 11, 13, 5, 6, 7 };

    int arr\_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

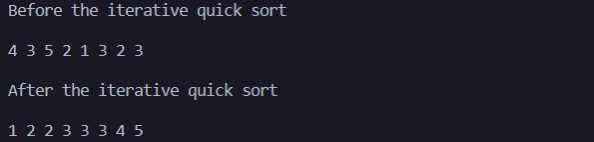
    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

    return 0;

}

**Que 04 (b)-01 recursive quick sort**

****

#include <stdio.h>

*// A utility function to swap two elements*

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

*/\* This function is same in both iterative and recursive\*/*

int partition(int arr[], int l, int h)

{

    int x = arr[h];

    int i = (l - 1);

    for (int j = l; j <= h - 1; j++) {

        if (arr[j] <= x) {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[h]);

    return (i + 1);

}

*/\* A[] --> Array to be sorted,*

*l  --> Starting index,*

*h  --> Ending index \*/*

void quickSortIterative(int arr[], int l, int h)

{

*// Create an auxiliary stack*

    int stack[h - l + 1];

*// initialize top of stack*

    int top = -1;

*// push initial values of l and h to stack*

    stack[++top] = l;

    stack[++top] = h;

*// Keep popping from stack while is not empty*

    while (top >= 0) {

*// Pop h and l*

        h = stack[top--];

        l = stack[top--];

*// Set pivot element at its correct position*

*// in sorted array*

        int p = partition(arr, l, h);

*// If there are elements on left side of pivot,*

*// then push left side to stack*

        if (p - 1 > l) {

            stack[++top] = l;

            stack[++top] = p - 1;

        }

*// If there are elements on right side of pivot,*

*// then push right side to stack*

        if (p + 1 < h) {

            stack[++top] = p + 1;

            stack[++top] = h;

        }

    }

}

*// A utility function to print contents of arr*

void printArr(int arr[], int n)

{

    int i;

    for (i = 0; i < n; ++i)

        printf("%d ", arr[i]);

}

*// Driver program to test above functions*

int main()

{

    int arr[] = { 4, 3, 5, 2, 1, 3, 2, 3 };

    int n = sizeof(arr) / sizeof(\*arr);

printf("Before the iterative quick sort");

printf("\n\n");

for(int i=0;i<n;i++)

{

printf("%d ",arr[i]);

}

printf("\n\n");

printf("After the iterative quick sort");

printf("\n\n");

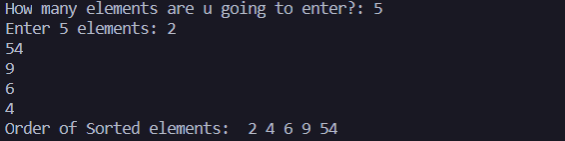
    quickSortIterative(arr, 0, n - 1);

    printArr(arr, n);

    return 0;

}

**Que 04 (b)-02 non-recursive quick sort**

****

#include<stdio.h>

void quicksort(int number[25],int first,int last){

   int i, j, pivot, temp;

   if(first<last){

      pivot=first;

      i=first;

      j=last;

      while(i<j){

         while(number[i]<=number[pivot]&&i<last)

            i++;

         while(number[j]>number[pivot])

            j--;

         if(i<j){

            temp=number[i];

            number[i]=number[j];

            number[j]=temp;

         }

      }

      temp=number[pivot];

      number[pivot]=number[j];

      number[j]=temp;

      quicksort(number,first,j-1);

      quicksort(number,j+1,last);

   }

}

int main(){

   int i, count, number[25];

   printf("How many elements are u going to enter?: ");

   scanf("%d",&count);

   printf("Enter %d elements: ", count);

   for(i=0;i<count;i++)

      scanf("%d",&number[i]);

   quicksort(number,0,count-1);

   printf("Order of Sorted elements: ");

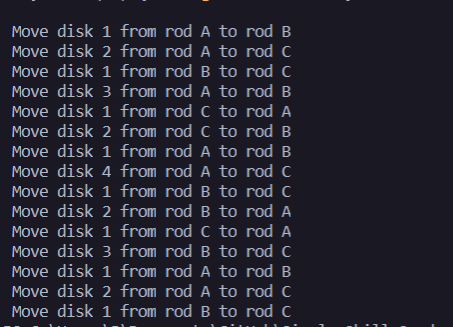
   for(i=0;i<count;i++)

      printf(" %d",number[i]);

   return 0;

}

**Que 04 (c)-01 recursive quick sort**



*/\* C program for Tower of Hanoi\*/*

*/\*Application of Recursive function\*/*

#include <stdio.h>

void hanoifun(int n, char fr, char tr, char ar)*//fr=from rod,tr =to rod, ar=aux rod*

{

    if (n == 1)

    {

        printf("\n Move disk 1 from rod %c to rod %c", fr, tr);

        return;

    }

    hanoifun(n-1, fr, ar, tr);

    printf("\n Move disk %d from rod %c to rod %c", n, fr, tr);

    hanoifun(n-1, ar, tr, fr);

}

int main()

{

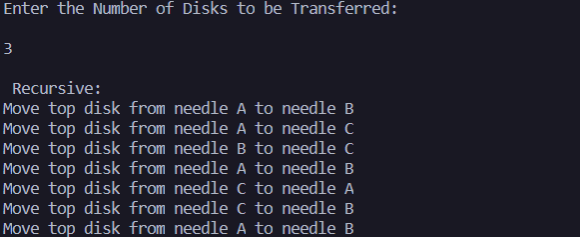
    int n = 4; *// n immplies the number of discs*

    hanoifun(n, 'A', 'C', 'B');  *// A, B and C are the name of rod*

    return 0;

}

**Que 04 (c)-02 non-recursive quick sort**



#include <stdio.h>

#include <stdlib.h>

void hanoirecursion(int num,char ndl1, char ndl2, char ndl3)

{

    if(num == 1)

    {

    printf("Move top disk from needle %c to needle %c\n",ndl1,ndl2);

    return;

    }

hanoirecursion(num-1,ndl1,ndl3,ndl2);

printf("Move top disk from needle %c to needle %c\n",ndl1,ndl2);

hanoirecursion(num-1,ndl3,ndl2,ndl1);

}

int main()

{

    int no;

    printf("Enter the Number of Disks to be Transferred:\n\n");

    scanf("%d",&no);

if(no<1)

{

    printf("\n There is nothing to move:\n");

}

else

{

    printf("\n Recursive:\n");

    hanoirecursion(no,'A','B','C');

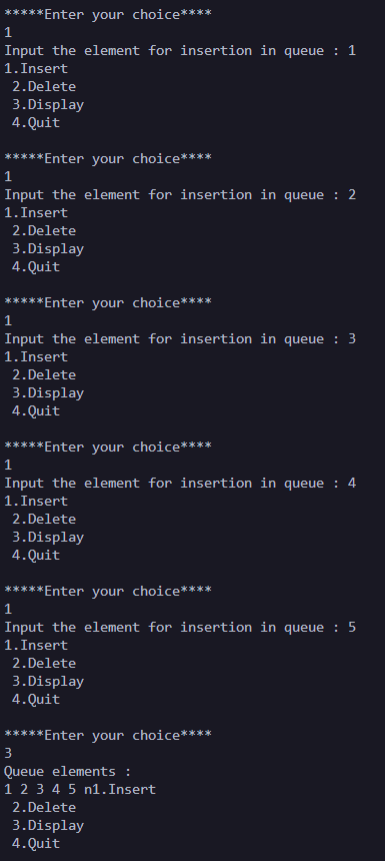
}

getch();

return 0;

}

**Q 5. Write a program using functions for implementation of circular Queue.**



#include <stdio.h>

#define MAX 5

int cqueue\_arr[MAX];

int front = -1;

int rear = -1;

void insert(int item)

{

    if ((front == 0 && rear == MAX - 1) || (front == rear + 1))

    {

        printf("Queue Overflow \n");

        return;

    }

    if (front == -1)

    {

        front = 0;

        rear = 0;

    }

    else

    {

        if (rear == MAX - 1)

            rear = 0;

        else

            rear = rear + 1;

    }

    cqueue\_arr[rear] = item;

}

void deletion()

{

    if (front == -1)

    {

        printf("Queue Underflown");

        return;

    }

    printf("Element deleted from queue is : %d\n", cqueue\_arr[front]);

    if (front == rear)

    {

        front = -1;

        rear = -1;

    }

    else

    {

        if (front == MAX - 1)

            front = 0;

        else

            front = front + 1;

    }

}

void display()

{

    int front\_pos = front, rear\_pos = rear;

    if (front == -1)

    {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements :\n");

    if (front\_pos <= rear\_pos)

        while (front\_pos <= rear\_pos)

        {

            printf("%d ", cqueue\_arr[front\_pos]);

            front\_pos++;

        }

    else

    {

        while (front\_pos <= MAX - 1)

        {

            printf("%d ", cqueue\_arr[front\_pos]);

                front\_pos++;

        }

        front\_pos = 0;

        while (front\_pos <= rear\_pos)

        {

            printf("%d ", cqueue\_arr[front\_pos]);

            front\_pos++;

        }

    }

    printf("n");

}

int main()

{

    int choice, item;

    do

    {

        printf("1.Insert \n ");

        printf("2.Delete \n ");

        printf("3.Display \n ");

        printf("4.Quit \n ");

        printf("\n\*\*\*\*\*Enter your choice\*\*\*\*\n");

        scanf("%d", &choice);

        switch (choice)

        {

        case 1:

            printf("Input the element for insertion in queue : ");

            scanf("%d", &item);

            insert(item);

            break;

        case 2:

            deletion();

            break;

        case 3:

            display();

            break;

        case 4:

            break;

        default:

            printf("Wrong choicen ->>> \n");

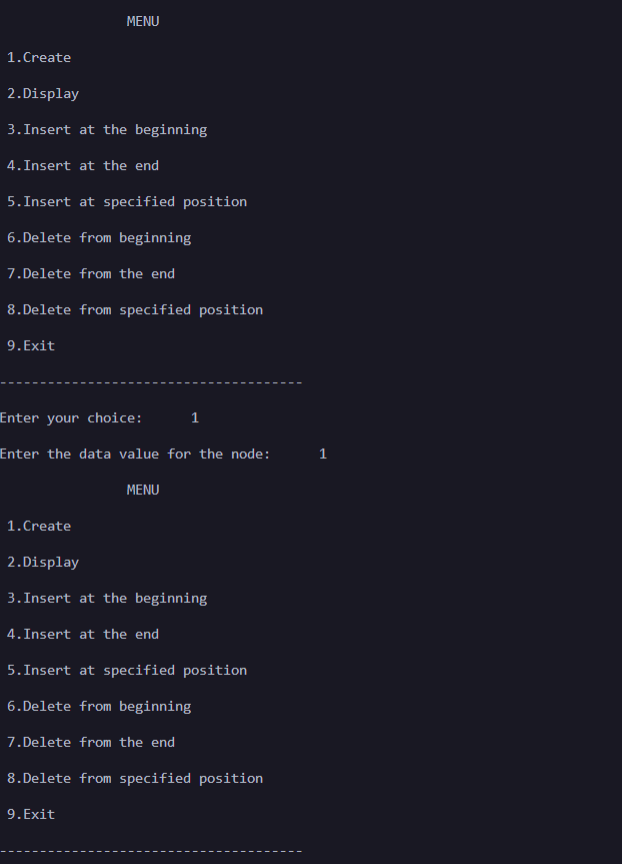
        }

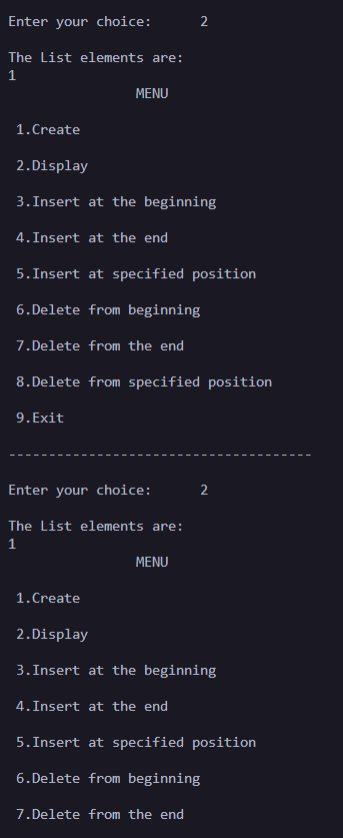
    } while (choice != 4);

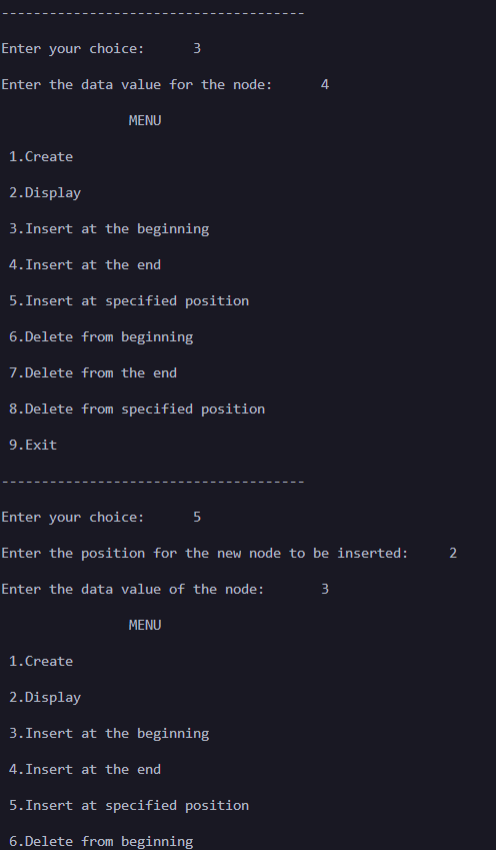
    return 0;

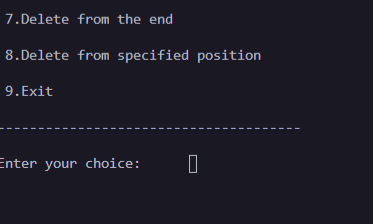
}

**Q 6.(a+b) Whole Program to implement the whole Linked-List**









#include<stdlib.h>

#include <stdio.h>

void create();

void display();

void insert\_begin();

void insert\_end();

void insert\_pos();

void delete\_begin();

void delete\_end();

void delete\_pos();

struct node

{

        int info;

        struct node \*next;

};

struct node \*start=NULL;

int main()

{

        int choice;

        while(1){

                printf("\n                MENU                             \n");

                printf("\n 1.Create     \n");

                printf("\n 2.Display    \n");

                printf("\n 3.Insert at the beginning    \n");

                printf("\n 4.Insert at the end  \n");

                printf("\n 5.Insert at specified position       \n");

                printf("\n 6.Delete from beginning      \n");

                printf("\n 7.Delete from the end        \n");

                printf("\n 8.Delete from specified position     \n");

                printf("\n 9.Exit       \n");

                printf("\n--------------------------------------\n");

                printf("\nEnter your choice:\t");

                scanf("%d",&choice);

                switch(choice)

                {

                        case 1:

                                        create();

                                        break;

                        case 2:

                                        display();

                                        break;

                        case 3:

                                        insert\_begin();

                                        break;

                        case 4:

                                        insert\_end();

                                        break;

                        case 5:

                                        insert\_pos();

                                        break;

                        case 6:

                                        delete\_begin();

                                        break;

                        case 7:

                                        delete\_end();

                                        break;

                        case 8:

                                        delete\_pos();

                                        break;

                        case 9:

                                        exit(0);

                                        break;

                        default:

                                        printf("\n Wrong Choice:\n");

                                        break;

                }

        }

        return 0;

}

void create()

{

        struct node \*temp,\*ptr;

        temp=(struct node \*)malloc(sizeof(struct node));

        if(temp==NULL)

        {

                printf("\nOut of Memory Space:\n");

                exit(0);

        }

        printf("\nEnter the data value for the node:\t");

        scanf("%d",&temp->info);

        temp->next=NULL;

        if(start==NULL)

        {

                start=temp;

        }

        else

        {

                ptr=start;

                while(ptr->next!=NULL)

                {

                        ptr=ptr->next;

                }

                ptr->next=temp;

        }

}

void display()

{

        struct node \*ptr;

        if(start==NULL)

        {

                printf("\nList is empty:\n");

                return;

        }

        else

        {

                ptr=start;

                printf("\nThe List elements are:\n");

                while(ptr!=NULL)

                {

                        printf("%d\t",ptr->info );

                        ptr=ptr->next ;

                }

        }

}

void insert\_begin()

{

        struct node \*temp;

        temp=(struct node \*)malloc(sizeof(struct node));

        if(temp==NULL)

        {

                printf("\nOut of Memory Space:\n");

                return;

        }

        printf("\nEnter the data value for the node:\t" );

        scanf("%d",&temp->info);

        temp->next =NULL;

        if(start==NULL)

        {

                start=temp;

        }

        else

        {

                temp->next=start;

                start=temp;

        }

}

void insert\_end()

{

        struct node \*temp,\*ptr;

        temp=(struct node \*)malloc(sizeof(struct node));

        if(temp==NULL)

        {

                printf("\nOut of Memory Space:\n");

                return;

        }

        printf("\nEnter the data value for the node:\t" );

        scanf("%d",&temp->info );

        temp->next =NULL;

        if(start==NULL)

        {

                start=temp;

        }

        else

        {

                ptr=start;

                while(ptr->next !=NULL)

                {

                        ptr=ptr->next ;

                }

                ptr->next =temp;

        }

}

void insert\_pos()

{

        struct node \*ptr,\*temp;

        int i,pos;

        temp=(struct node \*)malloc(sizeof(struct node));

        if(temp==NULL)

        {

                printf("\nOut of Memory Space:\n");

                return;

        }

        printf("\nEnter the position for the new node to be inserted:\t");

        scanf("%d",&pos);

        printf("\nEnter the data value of the node:\t");

        scanf("%d",&temp->info) ;

        temp->next=NULL;

        if(pos==0)

        {

                temp->next=start;

                start=temp;

        }

        else

        {

                for(i=0,ptr=start;i<pos-1;i++) { ptr=ptr->next;

                        if(ptr==NULL)

                        {

                                printf("\nPosition not found:[Handle with care]\n");

                                return;

                        }

                }

                temp->next =ptr->next ;

                ptr->next=temp;

        }

}

void delete\_begin()

{

        struct node \*ptr;

        if(ptr==NULL)

        {

                printf("\nList is Empty:\n");

                return;

        }

        else

        {

                ptr=start;

                start=start->next ;

                printf("\nThe deleted element is :%d\t",ptr->info);

                free(ptr);

        }

}

void delete\_end()

{

        struct node \*temp,\*ptr;

        if(start==NULL)

        {

                printf("\nList is Empty:");

                exit(0);

        }

        else if(start->next ==NULL)

        {

                ptr=start;

                start=NULL;

                printf("\nThe deleted element is:%d\t",ptr->info);

                free(ptr);

        }

        else

        {

                ptr=start;

                while(ptr->next!=NULL)

                {

                        temp=ptr;

                        ptr=ptr->next;

                }

                temp->next=NULL;

                printf("\nThe deleted element is:%d\t",ptr->info);

                free(ptr);

        }

}

void delete\_pos()

{

        int i,pos;

        struct node \*temp,\*ptr;

        if(start==NULL)

        {

                printf("\nThe List is Empty:\n");

                exit(0);

        }

        else

        {

                printf("\nEnter the position of the node to be deleted:\t");

                scanf("%d",&pos);

                if(pos==0)

                {

                        ptr=start;

                        start=start->next ;

                        printf("\nThe deleted element is:%d\t",ptr->info  );

                        free(ptr);

                }

                else

                {

                        ptr=start;

                        for(i=0;i<pos;i++) { temp=ptr; ptr=ptr->next ;

                                if(ptr==NULL)

                                {

                                        printf("\nPosition not Found:\n");

                                        return;

                                }

                        }

                        temp->next =ptr->next ;

                        printf("\nThe deleted element is:%d\t",ptr->info );

                        free(ptr);

                }

        }

}

**Que 09**

**Given a list, split it into two sublists — one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So FrontBackSplit() on the list {2, 3, 5, 7, 11} should yield the two lists {2, 3, 5} and {7, 11}.**



#include <stdio.h>

#include <stdlib.h>

*// A Linked List Node*

struct Node

{

    int data;

    struct Node \*next;

};

*// Helper function to print a given linked list*

void printList(struct Node \*head)

{

    struct Node \*ptr = head;

    while (ptr)

    {

        printf("%d -> ", ptr->data);

        ptr = ptr->next;

    }

    printf("null");

}

*// Helper function to insert a new node at the beginning of the linked list*

void push(struct Node \*\*head, int data)

{

    struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->next = \*head;

    \*head = newNode;

}

*// Return the total number of nodes in a list*

int findLength(struct Node \*head)

{

    int count = 0;

    struct Node \*current = head;

    while (current != NULL)

    {

        count++;

        current = current->next;

    }

    return count;

}

*/\**

*Split the given list's nodes into front and back halves*

*and return the two lists using the reference parameters.*

*If the length is odd, the extra node should go in the front list.*

*\*/*

void frontBackSplit(struct Node \*source, struct Node \*\*frontRef,

                    struct Node \*\*backRef)

{

    int len = findLength(source);

    if (len < 2)

    {

        \*frontRef = source;

        \*backRef = NULL;

        return;

    }

    struct Node \*current = source;

    int hopCount = (len - 1) / 2; *// (figured these with a few drawings)*

    for (int i = 0; i < hopCount; i++)

    {

        current = current->next;

    }

*// Now cut at current*

    \*frontRef = source;

    \*backRef = current->next;

    current->next = NULL;

}

int main(void)

{

*// input keys*

    int keys[] = {2,3,5,7,11};

    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++)

    {

        if (i != n - 1)

        {

            printf("%d -> ", keys[i]);

        }

        else

        {

            printf("null");

        }

    }

    printf("\n\n");

*// points to the head node of the linked list*

    struct Node \*head = NULL;

*// construct a linked list*

    for (int i = n - 1; i >= 0; i--)

    {

        push(&head, keys[i]);

    }

    struct Node \*a = NULL, \*b = NULL;

    frontBackSplit(head, &a, &b);

*// print linked list*

    printf("Front List: ");

    printList(a);

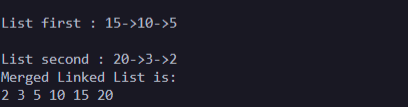
    printf("\nBack List: ");

    printList(b);

    return 0;

}

**Que 06 : (d) Given two one-way-link lists A and B. Sort them independently and then Merge as list C.**



#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

struct Node

{

    int data;

    struct Node\* next;

};

void MoveNode(struct Node\*\* destRef, struct Node\*\* sourceRef);

struct Node\* SortedMerge(struct Node\* a, struct Node\* b)

{

    struct Node dummy;

    struct Node\* tail = &dummy;

    dummy.next = NULL;

    while (1)

    {

        if (a == NULL)

        {

            tail->next = b;

            break;

        }

        else if (b == NULL)

        {

            tail->next = a;

            break;

        }

        if (a->data <= b->data)

            MoveNode(&(tail->next), &a);

        else

            MoveNode(&(tail->next), &b);

        tail = tail->next;

    }

    return(dummy.next);

}

void MoveNode(struct Node\*\* destRef, struct Node\*\* sourceRef)

{

    struct Node\* newNode = \*sourceRef;

    assert(newNode != NULL);

    \*sourceRef = newNode->next;

    newNode->next = \*destRef;

    \*destRef = newNode;

}

void push(struct Node\*\* head\_ref, int new\_data)

{

    struct Node\* new\_node =

        (struct Node\*) malloc(sizeof(struct Node));

    new\_node->data  = new\_data;

    new\_node->next = (\*head\_ref);

    (\*head\_ref)    = new\_node;

}

void printList(struct Node \*node)

{

    while (node!=NULL)

    {

        printf("%d ", node->data);

        node = node->next;

    }

}

int main()

{

    struct Node\* res = NULL;

    struct Node\* a = NULL;

    struct Node\* b = NULL;

printf("\nList first : 15->10->5\n");

printf("\nList second : 20->3->2\n");

    push(&a, 15);

    push(&a, 10);

    push(&a, 5);

    push(&b, 20);

    push(&b, 3);

    push(&b, 2);

    res = SortedMerge(a, b);

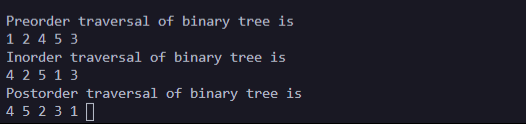
    printf("Merged Linked List is: \n");

    printList(res);

    return 0;

}

**Tree traversal**



#include <stdio.h>

#include <stdlib.h>

struct node

{

     int data;

     struct node\* left;

     struct node\* right;

};

struct node\* newNode(int data)

{

     struct node\* node = (struct node\*)

                                  malloc(sizeof(struct node));

     node->data = data;

     node->left = NULL;

     node->right = NULL;

     return(node);

}

void printPostorder(struct node\* node)

{

     if (node == NULL)

        return;

     printPostorder(node->left);

     printPostorder(node->right);

     printf("%d ", node->data);

}

void printInorder(struct node\* node)

{

     if (node == NULL)

          return;

     printInorder(node->left);

     printf("%d ", node->data);

     printInorder(node->right);

}

void printPreorder(struct node\* node)

{

     if (node == NULL)

          return;

     printf("%d ", node->data);

     printPreorder(node->left);

     printPreorder(node->right);

}

int main()

{

     struct node \*root  = newNode(1);

     root->left             = newNode(2);

     root->right           = newNode(3);

     root->left->left     = newNode(4);

     root->left->right   = newNode(5);

     printf("\nPreorder traversal of binary tree is \n");

     printPreorder(root);

     printf("\nInorder traversal of binary tree is \n");

     printInorder(root);

     printf("\nPostorder traversal of binary tree is \n");

     printPostorder(root);

     getchar();

     return 0;

}

**HEAP SORT**



*// C++ program for implementation of Heap Sort*

#include <stdio.h>

void swap(int i, int j);

void swap(int i, int j)

{

    int temp;

    temp = i;

    i = j;

    j = temp;

}

void heapify(int arr[], int n, int i)

{

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

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

        largest = l;

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

        largest = r;

    if (largest != i)

    {

        swap(arr[i], arr[largest]);

        heapify(arr, n, largest);

    }

}

void heapSort(int arr[], int n)

{

    for (int i = n / 2 - 1; i >= 0; i--)

        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--)

    {

        swap(arr[0], arr[i]);

        heapify(arr, i, 0);

    }

}

void printArray(int arr[], int n)

{

    for (int i = 0; i < n; ++i)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main()

{

    int arr[] = {12, 11, 13, 5, 6, 7};

    int n = sizeof(arr) / sizeof(arr[0]);

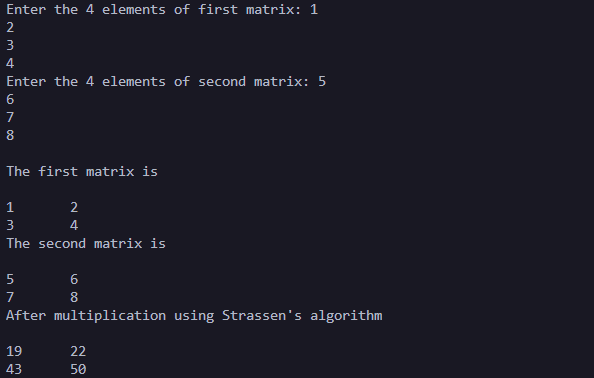
    heapSort(arr, n);

    printf("Sorted array is  : \n");

    printArray(arr, n);

}

**Q 10. Implement Strassen Matrix Multiplication problem using Divide & Conquer approach.**



#include <stdio.h>

int main()

{

    int a[2][2], b[2][2], c[2][2], i, j;

    int m1, m2, m3, m4, m5, m6, m7;

    printf("Enter the 4 elements of first matrix: ");

    for (i = 0; i < 2; i++)

        for (j = 0; j < 2; j++)

            scanf("%d", &a[i][j]);

    printf("Enter the 4 elements of second matrix: ");

    for (i = 0; i < 2; i++)

        for (j = 0; j < 2; j++)

            scanf("%d", &b[i][j]);

    printf("\nThe first matrix is\n");

    for (i = 0; i < 2; i++)

    {

        printf("\n");

        for (j = 0; j < 2; j++)

            printf("%d\t", a[i][j]);

    }

    printf("\nThe second matrix is\n");

    for (i = 0; i < 2; i++)

    {

        printf("\n");

        for (j = 0; j < 2; j++)

            printf("%d\t", b[i][j]);

    }

    m1 = (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]);

    m2 = (a[1][0] + a[1][1]) \* b[0][0];

    m3 = a[0][0] \* (b[0][1] - b[1][1]);

    m4 = a[1][1] \* (b[1][0] - b[0][0]);

    m5 = (a[0][0] + a[0][1]) \* b[1][1];

    m6 = (a[1][0] - a[0][0]) \* (b[0][0] + b[0][1]);

    m7 = (a[0][1] - a[1][1]) \* (b[1][0] + b[1][1]);

    c[0][0] = m1 + m4 - m5 + m7;

    c[0][1] = m3 + m5;

    c[1][0] = m2 + m4;

    c[1][1] = m1 - m2 + m3 + m6;

    printf("\nAfter multiplication using Strassen's algorithm \n");

    for (i = 0; i < 2; i++)

    {

        printf("\n");

        for (j = 0; j < 2; j++)

            printf("%d\t", c[i][j]);

    }

    return 0;

#include <stdio.h>

    int main()

    {

        int a[2][2], b[2][2], c[2][2], i, j;

        int m1, m2, m3, m4, m5, m6, m7;

        printf("Enter the 4 elements of first matrix: ");

        for (i = 0; i < 2; i++)

            for (j = 0; j < 2; j++)

                scanf("%d", &a[i][j]);

        printf("Enter the 4 elements of second matrix: ");

        for (i = 0; i < 2; i++)

            for (j = 0; j < 2; j++)

                scanf("%d", &b[i][j]);

        printf("\nThe first matrix is\n");

        for (i = 0; i < 2; i++)

        {

            printf("\n");

            for (j = 0; j < 2; j++)

                printf("%d\t", a[i][j]);

        }

        printf("\nThe second matrix is\n");

        for (i = 0; i < 2; i++)

        {

            printf("\n");

            for (j = 0; j < 2; j++)

                printf("%d\t", b[i][j]);

        }

        m1 = (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]);

        m2 = (a[1][0] + a[1][1]) \* b[0][0];

        m3 = a[0][0] \* (b[0][1] - b[1][1]);

        m4 = a[1][1] \* (b[1][0] - b[0][0]);

        m5 = (a[0][0] + a[0][1]) \* b[1][1];

        m6 = (a[1][0] - a[0][0]) \* (b[0][0] + b[0][1]);

        m7 = (a[0][1] - a[1][1]) \* (b[1][0] + b[1][1]);

        c[0][0] = m1 + m4 - m5 + m7;

        c[0][1] = m3 + m5;

        c[1][0] = m2 + m4;

        c[1][1] = m1 - m2 + m3 + m6;

        printf("\nAfter multiplication using Strassen's algorithm \n");

        for (i = 0; i < 2; i++)

        {

            printf("\n");

            for (j = 0; j < 2; j++)

                printf("%d\t", c[i][j]);

        }

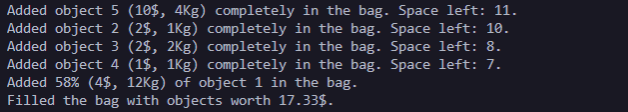
        return 0;

    }

}

**Q 11. Implement following problems using Greedy Approach:**

**Partial Knapsack**



#include <stdio.h>

int n = 5; */\* The number of objects \*/*

int c[10] = {12, 1, 2, 1, 4}; */\* c[i] is the \*COST\* of the ith object; i.e. what*

*YOU PAY to take the object \*/*

int v[10] = {4, 2, 2, 1, 10}; */\* v[i] is the \*VALUE\* of the ith object; i.e.*

*what YOU GET for taking the object \*/*

int W = 15; */\* The maximum weight you can take \*/*

void simple\_fill() {

    int cur\_w;

    float tot\_v;

    int i, maxi;

    int used[10];

    for (i = 0; i < n; ++i)

        used[i] = 0; */\* I have not used the ith object yet \*/*

    cur\_w = W;

    while (cur\_w > 0) { */\* while there's still room\*/*

*/\* Find the best object \*/*

        maxi = -1;

        for (i = 0; i < n; ++i)

            if ((used[i] == 0) &&

                ((maxi == -1) || ((float)v[i]/c[i] > (float)v[maxi]/c[maxi])))

                maxi = i;

        used[maxi] = 1; */\* mark the maxi-th object as used \*/*

        cur\_w -= c[maxi]; */\* with the object in the bag, I can carry less \*/*

        tot\_v += v[maxi];

        if (cur\_w >= 0)

            printf("Added object %d (%d$, %dKg) completely in the bag. Space left: %d.\n", maxi + 1, v[maxi], c[maxi], cur\_w);

        else {

            printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n", (int)((1 + (float)cur\_w/c[maxi]) \* 100), v[maxi], c[maxi], maxi + 1);

            tot\_v -= v[maxi];

            tot\_v += (1 + (float)cur\_w/c[maxi]) \* v[maxi];

        }

    }

    printf("Filled the bag with objects worth %.2f$.\n", tot\_v);

}

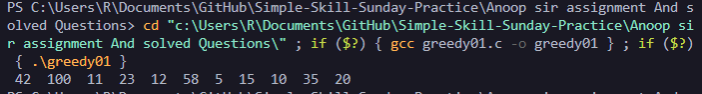
int main(int argc, char \*argv[]) {

    simple\_fill();

    return 0;

}

**b) Huffman compression**



#include <stdio.h>

#include <stdlib.h>

*// node*

*/\**

*\_\_\_\_\_*

*| data|*

*| freq|*

*|\_\_\_\_\_|*

*left ch./       \right chlid address*

*\_\_\_\_\_/         \ \_\_\_\_\_*

*| data|          | data|*

*| freq|          | freq|*

*|\_\_\_\_\_|          |\_\_\_\_\_|*

*\*/*

typedef struct node

{

  int frequency;

  char data;

  struct node \*left;

  struct node \*right;

}node;

int heap\_array\_size = 100; *// size of array storing heap*

int heap\_size = 0;

const int INF = 100000;

*//function to swap nodes*

void swap( node \*a, node \*b ) {

  node t;

  t = \*a;

  \*a = \*b;

  \*b = t;

}

*/\**

*function to print tree*

*https://www.codesdope.com/blog/article/binary-tree-in-c-linked-representation-traversals/*

*\*/*

void inorder(struct node \*root)

{

  if(root!=NULL) *// checking if the root is not null*

  {

    inorder(root->left); *// visiting left child*

    printf(" %d ", root->frequency); *// printing data at root*

    inorder(root->right);*// visiting right child*

  }

}

*/\**

*function for new node*

*\*/*

node\* new\_node(char data, int freq) {

  node \*p;

  p = malloc(sizeof(struct node));

  p->data = data;

  p->frequency = freq;

  p->left = NULL;

  p->right = NULL;

  return p;

}

*//function to get right child of a node of a tree*

int get\_right\_child(int index) {

  if((((2\*index)+1) <= heap\_size) && (index >= 1))

    return (2\*index)+1;

  return -1;

}

*//function to get left child of a node of a tree*

int get\_left\_child(int index) {

    if(((2\*index) <= heap\_size) && (index >= 1))

        return 2\*index;

    return -1;

}

*//function to get the parent of a node of a tree*

int get\_parent(int index) {

  if ((index > 1) && (index <= heap\_size)) {

    return index/2;

  }

  return -1;

}

*/\**

*Functions taken from minimum priority queue*

*https://www.codesdope.com/blog/article/priority-queue-using-heap/*

*https://www.codesdope.com/blog/article/heap-binary-heap/*

*\*/*

void insert(node A[], node\* a, int key) {

  heap\_size++;

  A[heap\_size] = \*a;

  int index = heap\_size;

  while((index>1) && (A[get\_parent(index)].frequency > a->frequency)) {

    swap(&A[index], &A[get\_parent(index)]);

    index = get\_parent(index);

  }

}

node\* build\_queue(node c[], int size) {

  node\* a = malloc(sizeof(node)\*heap\_array\_size); *// a is the array to store heap*

  int i;

  for(i=0; i<size; i++) {

    insert(a, &c[i], c[i].frequency); *// inserting node in array a(min-queue)*

  }

  return a;

}

void min\_heapify(node A[], int index) {

  int left\_child\_index = get\_left\_child(index);

  int right\_child\_index = get\_right\_child(index);

*// finding smallest among index, left child and right child*

  int smallest = index;

  if ((left\_child\_index <= heap\_size) && (left\_child\_index>0)) {

    if (A[left\_child\_index].frequency < A[smallest].frequency) {

      smallest = left\_child\_index;

    }

  }

  if ((right\_child\_index <= heap\_size && (right\_child\_index>0))) {

    if (A[right\_child\_index].frequency < A[smallest].frequency) {

      smallest = right\_child\_index;

    }

  }

*// smallest is not the node, node is not a heap*

  if (smallest != index) {

    swap(&A[index], &A[smallest]);

    min\_heapify(A, smallest);

  }

}

node\* extract\_min(node A[]) {

  node minm = A[1];

  A[1] = A[heap\_size];

  heap\_size--;

  min\_heapify(A, 1);

  node \*z;

*// copying minimum element*

  z = malloc(sizeof(struct node));

  z->data = minm.data;

  z->frequency = minm.frequency;

  z->left = minm.left;

  z->right = minm.right;

  return z; *//returning minimum element*

}

*// Huffman code*

node\* greedy\_huffman\_code(node C[]) {

  node \*min\_queue = build\_queue(C, 6); *// making min-queue*

  while(heap\_size > 1) {

    node \*h = extract\_min(min\_queue);

    node \*i = extract\_min(min\_queue);

    node \*z;

    z = malloc(sizeof(node));

    z->data = '\0';

    z->left = h;

    z->right = i;

    z->frequency = z->left->frequency + z->right->frequency;

    insert(min\_queue, z, z->frequency);

  }

  return extract\_min(min\_queue);

}

int main() {

  node \*a = new\_node('a', 42);

  node \*b = new\_node('b', 20);

  node \*c = new\_node('c', 5);

  node \*d = new\_node('d', 10);

  node \*e = new\_node('e', 11);

  node \*f = new\_node('f', 12);

  node C[] = {\*a, \*b, \*c, \*d, \*e , \*f};

  node\* z = greedy\_huffman\_code(C);

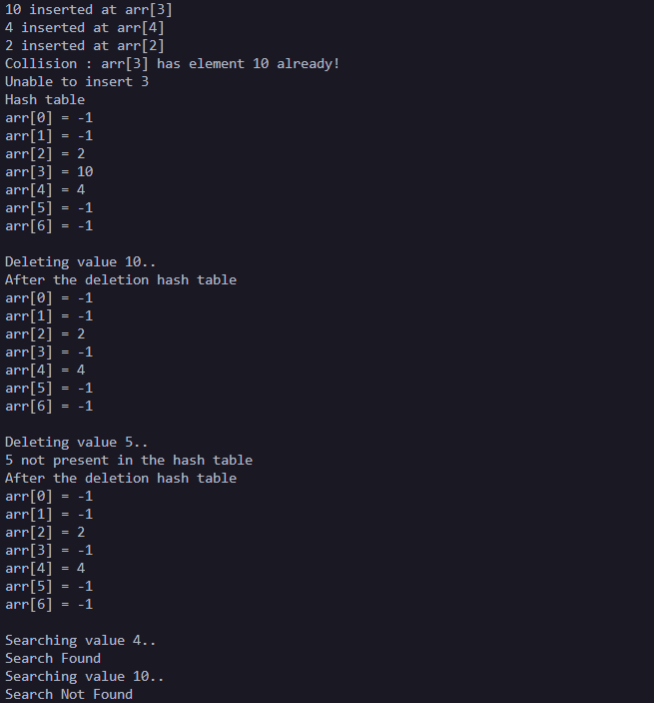
  inorder(z); *//printing tree*

  printf("\n");

  return 0;

}

**Q12. Write a program to implement concept of hashing. Design a menu based interface to call different functions for collision handling techniques.**



#include<stdio.h>

#define size 7

int arr[size];

void init()

{

    int i;

    for(i = 0; i < size; i++)

        arr[i] = -1;

}

void insert(int value)

{

    int key = value % size;

    if(arr[key] == -1)

    {

        arr[key] = value;

        printf("%d inserted at arr[%d]\n", value,key);

    }

    else

    {

        printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);

        printf("Unable to insert %d\n",value);

    }

}

void del(int value)

{

    int key = value % size;

    if(arr[key] == value)

        arr[key] = -1;

    else

        printf("%d not present in the hash table\n",value);

}

void search(int value)

{

    int key = value % size;

    if(arr[key] == value)

        printf("Search Found\n");

    else

        printf("Search Not Found\n");

}

void print()

{

    int i;

    for(i = 0; i < size; i++)

        printf("arr[%d] = %d\n",i,arr[i]);

}

int main()

{

    init();

    insert(10); *//key = 10 % 7 ==> 3*

    insert(4);  *//key = 4 % 7  ==> 4*

    insert(2);  *//key = 2 % 7  ==> 2*

    insert(3);  *//key = 3 % 7  ==> 3 (collision)*

    printf("Hash table\n");

    print();

    printf("\n");

    printf("Deleting value 10..\n");

    del(10);

    printf("After the deletion hash table\n");

    print();

    printf("\n");

    printf("Deleting value 5..\n");

    del(5);

    printf("After the deletion hash table\n");

    print();

    printf("\n");

    printf("Searching value 4..\n");

    search(4);

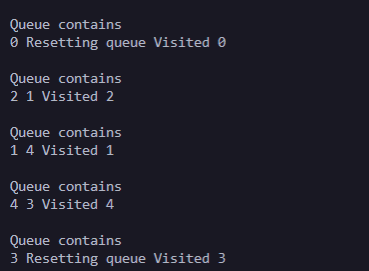
    printf("Searching value 10..\n");

    search(10);

    return 0;

}

**Q13. Write a program to find approachable nodes from a given source of a given graph using queue as an intermediate data structure (BFS).**



*// BFS algorithm in C*

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

  int items[SIZE];

  int front;

  int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

  int vertex;

  struct node\* next;

};

struct node\* createNode(int);

struct Graph {

  int numVertices;

  struct node\*\* adjLists;

  int\* visited;

};

*// BFS algorithm*

void bfs(struct Graph\* graph, int startVertex) {

  struct queue\* q = createQueue();

  graph->visited[startVertex] = 1;

  enqueue(q, startVertex);

  while (!isEmpty(q)) {

    printQueue(q);

    int currentVertex = dequeue(q);

    printf("Visited %d\n", currentVertex);

    struct node\* temp = graph->adjLists[currentVertex];

    while (temp) {

      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {

        graph->visited[adjVertex] = 1;

        enqueue(q, adjVertex);

      }

      temp = temp->next;

    }

  }

}

*// Creating a node*

struct node\* createNode(int v) {

  struct node\* newNode = malloc(sizeof(struct node));

  newNode->vertex = v;

  newNode->next = NULL;

  return newNode;

}

*// Creating a graph*

struct Graph\* createGraph(int vertices) {

  struct Graph\* graph = malloc(sizeof(struct Graph));

  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices \* sizeof(struct node\*));

  graph->visited = malloc(vertices \* sizeof(int));

  int i;

  for (i = 0; i < vertices; i++) {

    graph->adjLists[i] = NULL;

    graph->visited[i] = 0;

  }

  return graph;

}

*// Add edge*

void addEdge(struct Graph\* graph, int src, int dest) {

*// Add edge from src to dest*

  struct node\* newNode = createNode(dest);

  newNode->next = graph->adjLists[src];

  graph->adjLists[src] = newNode;

*// Add edge from dest to src*

  newNode = createNode(src);

  newNode->next = graph->adjLists[dest];

  graph->adjLists[dest] = newNode;

}

*// Create a queue*

struct queue\* createQueue() {

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

  q->front = -1;

  q->rear = -1;

  return q;

}

*// Check if the queue is empty*

int isEmpty(struct queue\* q) {

  if (q->rear == -1)

    return 1;

  else

    return 0;

}

*// Adding elements into queue*

void enqueue(struct queue\* q, int value) {

  if (q->rear == SIZE - 1)

    printf("\nQueue is Full!!");

  else {

    if (q->front == -1)

      q->front = 0;

    q->rear++;

    q->items[q->rear] = value;

  }

}

*// Removing elements from queue*

int dequeue(struct queue\* q) {

  int item;

  if (isEmpty(q)) {

    printf("Queue is empty");

    item = -1;

  } else {

    item = q->items[q->front];

    q->front++;

    if (q->front > q->rear) {

      printf("Resetting queue ");

      q->front = q->rear = -1;

    }

  }

  return item;

}

*// Print the queue*

void printQueue(struct queue\* q) {

  int i = q->front;

  if (isEmpty(q)) {

    printf("Queue is empty");

  } else {

    printf("\nQueue contains \n");

    for (i = q->front; i < q->rear + 1; i++) {

      printf("%d ", q->items[i]);

    }

  }

}

int main() {

  struct Graph\* graph = createGraph(6);

  addEdge(graph, 0, 1);

  addEdge(graph, 0, 2);

  addEdge(graph, 1, 2);

  addEdge(graph, 1, 4);

  addEdge(graph, 1, 3);

  addEdge(graph, 2, 4);

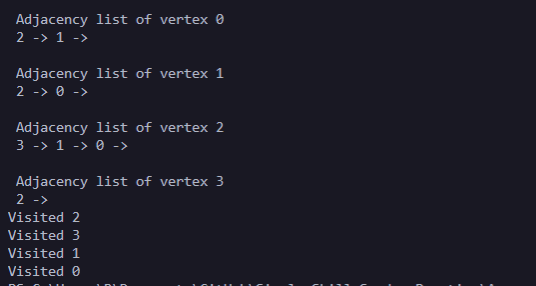
  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;

}

**Q14. Write a program to traverse various nodes of a given graph using stack as an intermediate data structure (DFS).**



*// DFS algorithm in C*

#include <stdio.h>

#include <stdlib.h>

struct node {

  int vertex;

  struct node\* next;

};

struct node\* createNode(int v);

struct Graph {

  int numVertices;

  int\* visited;

*// We need int\*\* to store a two dimensional array.*

*// Similary, we need struct node\*\* to store an array of Linked lists*

  struct node\*\* adjLists;

};

*// DFS algo*

void DFS(struct Graph\* graph, int vertex) {

  struct node\* adjList = graph->adjLists[vertex];

  struct node\* temp = adjList;

  graph->visited[vertex] = 1;

  printf("Visited %d \n", vertex);

  while (temp != NULL) {

    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {

      DFS(graph, connectedVertex);

    }

    temp = temp->next;

  }

}

*// Create a node*

struct node\* createNode(int v) {

  struct node\* newNode = malloc(sizeof(struct node));

  newNode->vertex = v;

  newNode->next = NULL;

  return newNode;

}

*// Create graph*

struct Graph\* createGraph(int vertices) {

  struct Graph\* graph = malloc(sizeof(struct Graph));

  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices \* sizeof(struct node\*));

  graph->visited = malloc(vertices \* sizeof(int));

  int i;

  for (i = 0; i < vertices; i++) {

    graph->adjLists[i] = NULL;

    graph->visited[i] = 0;

  }

  return graph;

}

*// Add edge*

void addEdge(struct Graph\* graph, int src, int dest) {

*// Add edge from src to dest*

  struct node\* newNode = createNode(dest);

  newNode->next = graph->adjLists[src];

  graph->adjLists[src] = newNode;

*// Add edge from dest to src*

  newNode = createNode(src);

  newNode->next = graph->adjLists[dest];

  graph->adjLists[dest] = newNode;

}

*// Print the graph*

void printGraph(struct Graph\* graph) {

  int v;

  for (v = 0; v < graph->numVertices; v++) {

    struct node\* temp = graph->adjLists[v];

    printf("\n Adjacency list of vertex %d\n ", v);

    while (temp) {

      printf("%d -> ", temp->vertex);

      temp = temp->next;

    }

    printf("\n");

  }

}

int main() {

  struct Graph\* graph = createGraph(4);

  addEdge(graph, 0, 1);

  addEdge(graph, 0, 2);

  addEdge(graph, 1, 2);

  addEdge(graph, 2, 3);

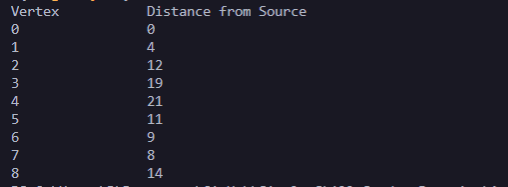
  printGraph(graph);

  DFS(graph, 2);

  return 0;

}

**Q15. Write a program to find shortest path from a given source to all the approachable nodes (Single source shortest path Dijkstra’s algorithm).**



#include <limits.h>

#include <stdio.h>

#include <stdbool.h>

*// Number of vertices in the graph*

#define V 9

*// A utility function to find the vertex with minimum distance value, from*

*// the set of vertices not yet included in shortest path tree*

int minDistance(int dist[], bool sptSet[])

{

*// Initialize min value*

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

        if (sptSet[v] == false && dist[v] <= min)

            min = dist[v], min\_index = v;

    return min\_index;

}

*// A utility function to print the constructed distance array*

void printSolution(int dist[])

{

    printf("Vertex \t\t Distance from Source\n");

    for (int i = 0; i < V; i++)

        printf("%d \t\t %d\n", i, dist[i]);

}

*// Function that implements Dijkstra's single source shortest path algorithm*

*// for a graph represented using adjacency matrix representation*

void dijkstra(int graph[V][V], int src)

{

    int dist[V]; *// The output array.  dist[i] will hold the shortest*

*// distance from src to i*

    bool sptSet[V]; *// sptSet[i] will be true if vertex i is included in shortest*

*// path tree or shortest distance from src to i is finalized*

*// Initialize all distances as INFINITE and stpSet[] as false*

    for (int i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = false;

*// Distance of source vertex from itself is always 0*

    dist[src] = 0;

*// Find shortest path for all vertices*

    for (int count = 0; count < V - 1; count++) {

*// Pick the minimum distance vertex from the set of vertices not*

*// yet processed. u is always equal to src in the first iteration.*

        int u = minDistance(dist, sptSet);

*// Mark the picked vertex as processed*

        sptSet[u] = true;

*// Update dist value of the adjacent vertices of the picked vertex.*

        for (int v = 0; v < V; v++)

*// Update dist[v] only if is not in sptSet, there is an edge from*

*// u to v, and total weight of path from src to  v through u is*

*// smaller than current value of dist[v]*

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

                && dist[u] + graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];

    }

*// print the constructed distance array*

    printSolution(dist);

}

*// driver program to test above function*

int main()

{

*/\* Let us create the example graph discussed above \*/*

    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },

                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },

                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },

                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },

                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },

                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },

                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },

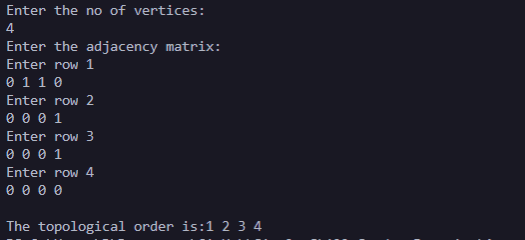
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;

}

**Q17. Write a program to arrange all the nodes of a given graph (Topological sort).**



#include <stdio.h>

int main(){

    int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;

    printf("Enter the no of vertices:\n");

    scanf("%d",&n);

    printf("Enter the adjacency matrix:\n");

    for(i=0;i<n;i++){

        printf("Enter row %d\n",i+1);

        for(j=0;j<n;j++)

            scanf("%d",&a[i][j]);

    }

    for(i=0;i<n;i++){

        indeg[i]=0;

        flag[i]=0;

    }

    for(i=0;i<n;i++)

        for(j=0;j<n;j++)

            indeg[i]=indeg[i]+a[j][i];

    printf("\nThe topological order is:");

    while(count<n){

        for(k=0;k<n;k++){

            if((indeg[k]==0) && (flag[k]==0)){

                printf("%d ",(k+1));

                flag [k]=1;

            }

            for(i=0;i<n;i++){

                if(a[i][k]==1)

                    indeg[k]--;

            }

        }

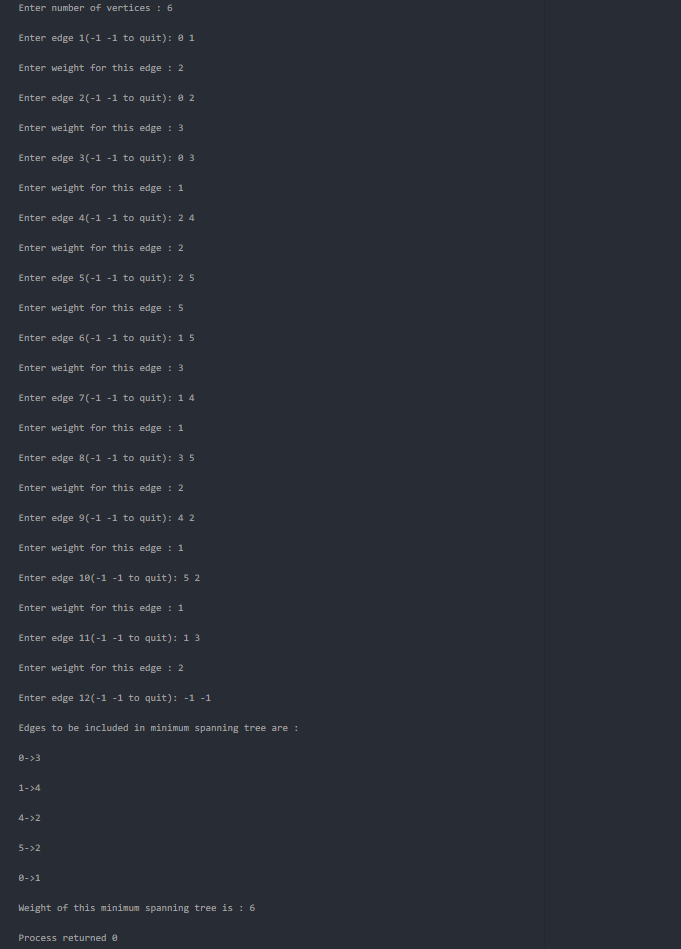
        count++;

    }

    return 0;

}

**Q18. Write a program to find Minimal spanning tree of a graph using Kruskal’s algorithm.**



*/\*  C Program for Minimum Spanning Tree using Kruskal's Algorithm Example \*/*

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

#define NIL -1

struct edge

{

        int u;

        int v;

        int weight;

        struct edge \*link;

}\*front = NULL;

void make\_tree(struct edge tree[]);

void insert\_pque(int i,int j,int wt);

struct edge \*del\_pque();

int isEmpty\_pque( );

void create\_graph();

int n;   */\*Total number of vertices in the graph \*/*

int main()

{

        int i;

        struct edge tree[MAX]; */\* Will contain the edges of spanning tree \*/*

        int wt\_tree = 0; */\* Weight of the spanning tree \*/*

        create\_graph();

        make\_tree(tree);

        printf("\nEdges to be included in minimum spanning tree are :\n");

        for(i=1; i<=n-1; i++)

        {

                printf("\n%d->",tree[i].u);

                printf("%d\n",tree[i].v);

                wt\_tree += tree[i].weight;

        }

        printf("\nWeight of this minimum spanning tree is : %d\n", wt\_tree);

        return 0;

}*/\*End of main()\*/*

void make\_tree(struct edge tree[])

{

        struct edge \*tmp;

        int v1,v2,root\_v1,root\_v2;

        int father[MAX]; */\*Holds father of each vertex \*/*

        int i,count = 0;    */\* Denotes number of edges included in the tree \*/*

        for(i=0; i<n; i++)

                father[i] = NIL;

*/\*Loop till queue becomes empty or*

*till n-1 edges have been inserted in the tree\*/*

        while( !isEmpty\_pque( ) && count < n-1 )

        {

                tmp = del\_pque();

                v1 = tmp->u;

                v2 = tmp->v;

                while( v1 !=NIL )

                {

                        root\_v1 = v1;

                        v1 = father[v1];

                }

                while( v2 != NIL  )

                {

                        root\_v2 = v2;

                        v2 = father[v2];

                }

                if( root\_v1 != root\_v2 )*/\*Insert the edge (v1, v2)\*/*

                {

                    count++;

                        tree[count].u = tmp->u;

                        tree[count].v = tmp->v;

                        tree[count].weight = tmp->weight;

                        father[root\_v2]=root\_v1;

                }

        }

        if(count < n-1)

        {

                printf("\nGraph is not connected, no spanning tree possible\n");

                exit(1);

        }

}*/\*End of make\_tree()\*/*

*/\*Inserting edges in the linked priority queue \*/*

void insert\_pque(int i,int j,int wt)

{

        struct edge \*tmp,\*q;

        tmp = (struct edge \*)malloc(sizeof(struct edge));

        tmp->u = i;

        tmp->v = j;

        tmp->weight = wt;

*/\*Queue is empty or edge to be added has weight less than first edge\*/*

        if( front == NULL || tmp->weight < front->weight )

        {

                tmp->link = front;

                front = tmp;

        }

        else

        {

                q = front;

                while( q->link != NULL && q->link->weight <= tmp->weight )

                        q = q->link;

                tmp->link = q->link;

                q->link = tmp;

                if(q->link == NULL)  */\*Edge to be added at the end\*/*

                        tmp->link = NULL;

        }

}*/\*End of insert\_pque()\*/*

*/\*Deleting an edge from the linked priority queue\*/*

struct edge \*del\_pque()

{

        struct edge \*tmp;

        tmp = front;

        front = front->link;

        return tmp;

}*/\*End of del\_pque()\*/*

int isEmpty\_pque( )

{

        if ( front == NULL )

                return 1;

        else

                return 0;

}*/\*End of isEmpty\_pque()\*/*

void create\_graph()

{

        int i,wt,max\_edges,origin,destin;

        printf("\nEnter number of vertices : ");

        scanf("%d",&n);

        max\_edges = n\*(n-1)/2;

        for(i=1; i<=max\_edges; i++)

        {

                printf("\nEnter edge %d(-1 -1 to quit): ",i);

                scanf("%d %d",&origin,&destin);

                if( (origin == -1) && (destin == -1) )

                        break;

                printf("\nEnter weight for this edge : ");

                scanf("%d",&wt);

                if( origin >= n || destin >= n || origin<0 || destin<0)

                {

                        printf("\nInvalid edge!\n");

                        i--;

                }

                else

                        insert\_pque(origin,destin,wt);

        }

}