```
//1.write a program in c to convert /reverseing a 32bit singed integer?
#include<stdio.h>
int reverse(int n,int rev){
	if (n==0){
		return rev;
	}
	rev = rev*10+n%10;
	return reverse(n/10,rev);
}
int main(){
	int x;
	printf("Enter the number:");
	scanf("%d",&x);

	printf("Reverse:%d\n",reverse(x,0));
	return 0;
}

//2.write a cprogram to cheak for a valid string
#include<stdio.h>
int is_valid(char s[]){
	int i =0 ;
	while (s[i]!='/0'){
		if (s[i]>='0' || s[i]<='9' && s[i]<='a' || s[i]>='z'){
			return 1;
		}
		i++;
	}
	return 0;
}
int main(){
```

```
        char s[100];
        printf("Enter the string:");
        scanf("%c",&s);
        if (is_valid(s)){
                printf("String is valid");
        }
        else{
                printf("String is invalid");
        }
        return 0;
}


//3.implement a c program to merge two arrays.
#include <stdio.h>

void mergeArrays(int arr1[], int size1, int arr2[], int size2, int merged[]) {
   int i = 0, j = 0, k = 0;

   while (i < size1 && j < size2) {
      if (arr1[i] <= arr2[j]) {
         merged[k++] = arr1[i++];
      } else {
         merged[k++] = arr2[j++];
      }
   }

   while (i < size1) {
      merged[k++] = arr1[i++];
   }
```

```
    while (j < size2) {
        merged[k++] = arr2[j++];
    }
}

int main() {
    int arr1[] = {1, 3, 5, 7};
    int arr2[] = {2, 4, 6, 8};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    int merged[size1 + size2];

    mergeArrays(arr1, size1, arr2, size2, merged);

    printf("Merged array: ");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d ", merged[i]);
    }
    printf("\n");

    return 0;
}

//4. write a program in c to count the total number of duplicate elements in an array.
#include <stdio.h>

int countDuplicates(int arr[], int size) {
    int count = 0;
    int duplicateFlag[size]; // Array to keep track of duplicates
```

```
    for (int i = 0; i < size; i++) {
        duplicateFlag[i] = 0;
    }

    for (int i = 0; i < size; i++) {
        if (duplicateFlag[i] == 1) {
            continue;
        }

        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                if (duplicateFlag[j] == 0) {
                    duplicateFlag[j] = 1;
                    count++;
                }
                if (duplicateFlag[i] == 0) {
                    duplicateFlag[i] = 1 ;
                }
            }
        }
    }
    return count;
}
int main() {
    int arr[] = {1, 2, 3, 2, 3, 4, 5, 6, 4};
    int size = sizeof(arr) / sizeof(arr[0]);

    int totalDuplicates = countDuplicates(arr, size);

    printf("Total number of duplicate elements: %d\n", totalDuplicates);
```

```
    return 0;
}

//5. write a c program to merging of list.
#include <stdio.h>

void mergeLists(int list1[], int size1, int list2[], int size2, int mergedList[]) {
    int i = 0, j = 0, k = 0;

    while (i < size1 && j < size2) {
        if (list1[i] <= list2[j]) {
            mergedList[k++] = list1[i++];
        } else {
            mergedList[k++] = list2[j++];
        }
    }

    while (i < size1) {
        mergedList[k++] = list1[i++];
    }
    while (j < size2) {
        mergedList[k++] = list2[j++];
    }
}

int main() {
    int list1[] = {1, 3, 5, 7};
    int list2[] = {2, 4, 6, 8};
    int size1 = sizeof(list1) / sizeof(list1[0]);
    int size2 = sizeof(list2) / sizeof(list2[0]);
```

```
    int mergedList[size1 + size2];

    mergeLists(list1, size1, list2, size2, mergedList);

    printf("Merged list: ");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d ", mergedList[i]);
    }
    printf("\n");

    return 0;
}



//6.implement a c program given an array of  reg.no need to search for particular reg.no
#include <stdio.h>
#include <stdbool.h>

bool linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return true;
        }
    }
    return false;
}

int main() {
    int regNos[] = {1001, 1002, 1003, 1004, 1005};
    int size = sizeof(regNos) / sizeof(regNos[0]);
    int target = 1003;
```

```
    if (linearSearch(regNos, size, target)) {
        printf("Registration number %d found.\n", target);
    } else {
        printf("Registration number %d not found.\n", target);
    }

    return 0;
}
//7. Identify location of element in given array
#include <stdio.h>
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
int main() {
    int arr[] = {20, 30, 40, 50, 60};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 40;
    int index = linearSearch(arr, size, target);
    if (index != -1) {
        printf("Element %d found at index %d.\n", target, index);
    } else {
        printf("Element %d not found in the array.\n", target);
    }
    return 0;
}
```

```c
//8. write a program in c to separate odd and even integers into separate arrays.
#include <stdio.h>


void separateEvenOdd(int arr[], int size, int evenArr[], int *evenSize, int oddArr[], int *oddSize) {
    *evenSize = 0;
    *oddSize = 0;

    for (int i = 0; i < size; i++) {
        if (arr[i] % 2 == 0) {
            evenArr[(*evenSize)++] = arr[i];
        } else {
            oddArr[(*oddSize)++] = arr[i];
        }
    }
}


int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(arr) / sizeof(arr[0]);
    int evenArr[size];
    int oddArr[size];
    int evenSize, oddSize;

    separateEvenOdd(arr, size, evenArr, &evenSize, oddArr, &oddSize);

    printf("Even numbers: ");
    for (int i = 0; i < evenSize; i++) {
        printf("%d ", evenArr[i]);
    }
```

```
    printf("\n");

    printf("Odd numbers: ");
    for (int i = 0; i < oddSize; i++) {
        printf("%d ", oddArr[i]);
    }
    printf("\n");

    return 0;
}

 //9.Write a c program to find the sum of Fibonacci Series
#include <stdio.h>
long long int sumOfFibonacci(int n) {
    long long int a = 0, b = 1, sum = a + b;
    for (int i = 2; i < n; i++) {
        long long int next = a + b;
        sum += next;
        a = b;
        b = next;
    }
    return sum;
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("The number of terms must be positive.\n");
```

```
    } else if (n == 1) {
        printf("Sum of the first %d Fibonacci number is: %d\n", n, 0);
    } else if (n == 2) {
        printf("Sum of the first %d Fibonacci numbers is: %d\n", n, 1);
    } else {
        long long int sum = sumOfFibonacci(n);
        printf("Sum of the first %d Fibonacci numbers is: %lld\n", n, sum);
    }

    return 0;
}
//10.write a c program  to find the factorial of a number.
#include <stdio.h>

unsigned long long factorial(int n) {
    unsigned long long fact = 1;
    for (int i = 1; i <= n; ++i) {
        fact *= i;
    }
    return fact;
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Factorial of a negative number doesn't exist.\n");
    } else {
        printf("Factorial of %d = %llu\n", num, factorial(num));
    }
```

```
    return 0;
}

//11.write a programm AVL tree:

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

typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

int height(Node *node) {
    if (node == NULL)
        return 0;
    return node->height;
}

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

Node *newNode(int key) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
```

```
    return node;
}

Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

int getBalance(Node *node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

Node *insert(Node *node, int key) {
    if (node == NULL)
```

```
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

Node minValueNode(Node node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

```
Node *deleteNode(Node *root, int key) {
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            Node *temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
            free(temp);
        } else {
            Node *temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
    if (root == NULL)
        return root;
    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
```

```
    }
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

void PrintTree(Node* root, int space) {
    int count = 10;
    if (root == NULL)
        return;
    space += count;
    PrintTree(root->right, space);
    printf("\n");
    for (int i = count; i < space; i++)
        printf(" ");
    printf("%d\n", root->key);
    PrintTree(root->left, space);
}

int main() {
    Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
```

```
    printf("Tree structure:\n");
    PrintTree(root, 0);

    root = deleteNode(root, 40);
    printf("\nTree structure after deletion:\n");
    PrintTree(root, 0);

    return 0;
}

//12.implement a c program whether it is a valid stack:



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

#define MAX 100 // Maximum size of the stack

typedef struct {
    int items[MAX];
    int top;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == MAX - 1;
}
```

```
int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, int newItem) {
    if (isFull(s)) {
        printf("Stack is full. Cannot push %d\n", newItem);
    } else {
        s->items[++(s->top)] = newItem;
        printf("%d pushed to stack\n", newItem);
    }
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
    } else {
        return s->items[(s->top)--];
    }
}

void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
    } else {
        printf("Stack elements: ");
        for (int i = 0; i <= s->top; i++) {
            printf("%d ", s->items[i]);
        }
        printf("\n");
```

```
    }
}

int main() {
    Stack s;
    initStack(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    display(&s);

    printf("Popped element: %d\n", pop(&s));
    printf("Popped element: %d\n", pop(&s));

    display(&s);

    return 0;
}

//i13.mplement a c program for graph to identify shortest path:

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

#define V 9  // Number of vertices in the graph

int minDistance(int dist[], bool sptSet[]) {
    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;
}

void printSolution(int dist[], int n) {
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < n; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

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 the shortest path tree

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; 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];
    }

    printSolution(dist, V);
```

```
}

int main() {
   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;
}
```

//14.implement a c program travelling salesman problem to identify shortest path given a set of cities and distances between every pair of cities ,the problem is to fin dthe shortest path that visits every city exactly once and returns to the starting point:

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

#define V 4
#define INF INT_MAX

int tsp(int graph[V][V], int dp[1 << V][V], int mask, int pos) {
   if (mask == (1 << V) - 1) {
      return graph[pos][0];
   }
   if (dp[mask][pos] != -1) {
```

```
        return dp[mask][pos];
    }
    int ans = INF;
    for (int city = 0; city < V; city++) {
        if ((mask & (1 << city)) == 0) {
            int newAns = graph[pos][city] + tsp(graph, dp, mask | (1 << city), city);
            if (newAns < ans) {
                ans = newAns;
            }
        }
    }
    return dp[mask][pos] = ans;
}

int main() {
    int graph[V][V] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int dp[1 << V][V];
    for (int i = 0; i < (1 << V); i++) {
        for (int j = 0; j < V; j++) {
            dp[i][j] = -1;
        }
    }

    int result = tsp(graph, dp, 1, 0);
    printf("The minimum cost of visiting all cities is %d\n", result);
```

```
    return 0;
}

//15.implement a c program for binary search tree_search for a element ,min element and max element
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(int item) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

struct Node* insert(struct Node* node, int data) {
    if (node == NULL) return newNode(data);
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    return node;
}

struct Node* search(struct Node* root, int key) {
```

```
    if (root == NULL || root->data == key)
        return root;
    if (root->data < key)
        return search(root->right, key);
    return search(root->left, key);
}

struct Node* findMin(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}
struct Node* findMax(struct Node* node) {
    struct Node* current = node;
    while (current && current->right != NULL)
        current = current->right;
    return current;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
```

```
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal of the BST: ");
    inorder(root);
    printf("\n");

    int key = 40;
    if (search(root, key) != NULL)
        printf("Element %d found in the BST.\n", key);
    else
        printf("Element %d not found in the BST.\n", key);

    struct Node* minNode = findMin(root);
    if (minNode != NULL)
        printf("Minimum element in the BST is %d.\n", minNode->data);

    struct Node* maxNode = findMax(root);
    if (maxNode != NULL)
        printf("Maximum element in the BST is %d.\n", maxNode->data);

    return 0;
}
```