1)

```
#include<stdio.h>
 int main(){
    int arr[]={1,2,3,4,5},i;
    int n=sizeof(arr)/sizeof(arr[0]);
    for(i=n-1;i>=0;i--){
            printf("%d ",arr[i]);
    }
    return 0;
}
```

Test Data :

Input the number of elements to store in the array :3

Input 3 number of elements in the array :

element - 0 : 2

element - 1 : 5

element - 2 : 7

Expected Output :

The values store into the array are :

2 5 7

The values store into the array in reverse are : 7 5 2

2.)

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int key;
  struct Node *left;
  struct Node *right;
  int height;
};
int max(int a, int b);
int height(struct Node *N) {
```

```
  if (N == NULL)
    return 0;
  return N->height;
}
int max(int a, int b) {
  return (a > b) ? a : b;
}
struct Node *newNode(int key) {
  struct Node *node = (struct Node *)
    malloc(sizeof(struct Node));
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct 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;
}
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct 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(struct Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}
struct Node *insertNode(struct Node *node, int key) {
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(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;
}
struct Node *minValueNode(struct Node *node) {
  struct Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}
struct Node *deleteNode(struct 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)) {
      struct Node *temp = root->left ? root->left : root->right;

      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      struct 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 printPreOrder(struct Node *root) {
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}
int main() {
  struct Node *root = NULL;
  root = insertNode(root, 2);
```

```
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);
  printPreOrder(root);
  root = deleteNode(root, 3);
  printf("\nAfter deletion: ");
  printPreOrder(root);
  return 0;
}
```

Input: 4 2 1 3 7 5 8

After deletion of “3”.

output: 4 2 1 7 5 8

3.
```
#include <stdio.h>
#include <ctype.h>
int isValidString(const char *str) {
    while (*str) {
        if (!isalpha(*str) && !isspace(*str)) {
            return 0;
        }
        str++;
    }
    return 1;
}
int main() {
    char input[100];
    printf("Enter a string: ");
```

```
    fgets(input, sizeof(input), stdin);
    size_t len = strlen(input);
    if (len > 0 && input[len - 1] == '\n') {
        input[len - 1] = '\0';
    }
    if (isValidString(input)) {
        printf("The string is valid.\n");
    } else {
        printf("The string is not valid.\n");
    }
    return 0;
}
```

Input: hello world

Output: string is valid

Input: hello123

Output: string is not valid

4.)

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

// Function to check if a sequence of operations is valid for a stack
int isValidStackSequence(const char *operations) {
    int stackSize = 0;
    for (int i = 0; operations[i] != '\0'; i++) {
        if (operations[i] == 'P') {
            stackSize++;
        } else if (operations[i] == 'O') {
            if (stackSize == 0) {
                return 0;
            }
```

```
            stackSize--;
        } else {
            return 0;
        }
    }
    return 1;
}
int main() {
    char operations[100];
    printf("Enter the sequence of stack operations (P for push, O for pop): ");
    fgets(operations, sizeof(operations), stdin);
    size_t len = strlen(operations);
    if (len > 0 && operations[len - 1] == '\n') {
        operations[len - 1] = '\0';
    }
    if (isValidStackSequence(operations)) {
        printf("The sequence is a valid stack sequence.\n");
    } else {
        printf("The sequence is not a valid stack sequence.\n");
    }

    return 0;
}
```

Input: PPP0

Output: valid stack

Input: ppo

Output: invalid stack

5.)

```
#include <stdio.h>
void mergeArrays(int arr1[], int size1, int arr2[], int size2, int merged[]) {
    int i, j;
```

```
    for (i = 0; i < size1; i++) {
        merged[i] = arr1[i];
    }
    for (j = 0; j < size2; j++) {
        merged[i + j] = arr2[j];
    }
}
int main() {
    int size1, size2;
    printf("Enter the size of the first array: ");
    scanf("%d", &size1);
    int arr1[size1];
    printf("Enter %d elements for the first array:\n", size1);
    for (int i = 0; i < size1; i++) {
        scanf("%d", &arr1[i]);
    }
    printf("Enter the size of the second array: ");
    scanf("%d", &size2);
    int arr2[size2];
    printf("Enter %d elements for the second array:\n", size2);
    for (int i = 0; i < size2; i++) {
        scanf("%d", &arr2[i]);
    }
    int merged[size1 + size2];
    mergeArrays(arr1, size1, arr2, size2, merged);
    printf("Merged array:\n");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d ", merged[i]);
    }
    printf("\n");
    return 0;
```

```
}
```

Input: array 1={1,2,3}

Array 2={4,5}

Output: merged array: {1,2,3,4,5}

6.)

```
#include <stdio.h>
#include <limits.h>
#define V 9
int minDistance(int dist[], int sptSet[]) {
   int min = INT_MAX, min_index;
   int v;
   for (v = 0; v < V; v++)
      if (sptSet[v] == 0 && dist[v] <= min)
         min = dist[v], min_index = v;

   return min_index;
}
void printSolution(int dist[], int n) {
   printf("Vertex   Distance from Source\n");
   int i;
   for (i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src) {
   int dist[V];
   int sptSet[V];
   int i, count, v;
   for (i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = 0;
   dist[src] = 0;
   for (count = 0; count < V - 1; count++) {
```

```
      int u = minDistance(dist, sptSet);
      sptSet[u] = 1;
      for (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, 0, 10, 0, 2, 0, 0},
                       {0, 0, 0, 14, 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;
}
```

Output:

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
```

5 11

6 9

7 8

8 14

7.) #include <stdio.h>

```
int countDuplicates(int arr[], int size) {
    int count = 0;
    int visited[size];
    for (int i = 0; i < size; i++) {
        visited[i] = 0;
    }
    for (int i = 0; i < size; i++) {
        if (visited[i] == 1) {
            continue;
        }
        int duplicateFound = 0;
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                visited[j] = 1;
                duplicateFound = 1;
            }
        }
        if (duplicateFound == 1) {
            count++;
        }
    }
    return count;
}
int main() {
```

```
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    int duplicateCount = countDuplicates(arr, n);
    printf("Total number of duplicate elements: %d\n", duplicateCount);
    return 0;
}
```

Input: array={1,1,2,3,4,1}

Output: no.of duplicate elements is: 1

8.)

```
#include <stdio.h>
#include <limits.h>
#define N 4
int calculateTourLength(int dist[N][N], int tour[]) {
    int totalLength = 0;
    for (int i = 0; i < N - 1; i++) {
        totalLength += dist[tour[i]][tour[i + 1]];
    }
    totalLength += dist[tour[N - 1]][tour[0]];
    return totalLength;
}
void tspBruteForce(int dist[N][N]) {
    int tour[N];
    int minLength = INT_MAX;
    int minTour[N];
    for (int i = 0; i < N; i++) {
```

```
        tour[i] = i;
    }
    do {
        int currentLength = calculateTourLength(dist, tour);
        if (currentLength < minLength) {
            minLength = currentLength;
            for (int i = 0; i < N; i++) {
                minTour[i] = tour[i];
            }
        }
    } while (nextPermutation(tour, N));
    printf("Minimum tour length: %d\n", minLength);
    printf("Tour: ");
    for (int i = 0; i < N; i++) {
        printf("%d ", minTour[i]);
    }
    printf("%d\n", minTour[0]);
}
int nextPermutation(int *arr, int size) {
    int i = size - 2;
    while (i >= 0 && arr[i] >= arr[i + 1]) {
        i--;
    }
    if (i < 0) {
        return 0;
    }
    int j = size - 1;
    while (arr[j] <= arr[i]) {
        j--;
    }
    int temp = arr[i];
```

```
    arr[i] = arr[j];
    arr[j] = temp;
    int left = i + 1;
    int right = size - 1;
    while (left < right) {
        temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
    return 1;
}
int main() {
    int dist[N][N] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    tspBruteForce(dist);
    return 0;
}
```

Input: Minimum tour length: 80

Tour: 0 1 3 2 0

9.)

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

```
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void insertEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
struct Node* mergeLists(struct Node* l1, struct Node* l2) {
    struct Node* result = NULL;
    if (l1 == NULL)
        return l2;
```

```
    if (l2 == NULL)
        return l1;
    if (l1->data <= l2->data) {
        result = l1;
        result->next = mergeLists(l1->next, l2);
    } else {
        result = l2;
        result->next = mergeLists(l1, l2->next);
    }
    return result;
}
int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;
    insertEnd(&list1, 1);
    insertEnd(&list1, 3);
    insertEnd(&list1, 5);
    insertEnd(&list2, 2);
    insertEnd(&list2, 4);
    insertEnd(&list2, 6);
    struct Node* mergedList = mergeLists(list1, list2);
    printf("Merged sorted linked list: ");
    printList(mergedList);
    return 0;
}
```

Output: Merged sorted linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL

10.)

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

```
    struct Node* left;
    struct Node* right;
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

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

    if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
```

```
    }
}
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
struct Node* findMax(struct Node* root) {
    while (root->right != NULL) {
        root = root->right;
    }
    return root;
}
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("In-order traversal of the BST: ");
```

```
    inorder(root);
    printf("\n");
    int searchValue = 40;
    struct Node* searchResult = search(root, searchValue);
    if (searchResult != NULL) {
        printf("Element %d found in the BST.\n", searchValue);
    } else {
        printf("Element %d not found in the BST.\n", searchValue);
    }
    struct Node* minNode = findMin(root);
    struct Node* maxNode = findMax(root);
    if (minNode != NULL) {
        printf("Minimum element in the BST: %d\n", minNode->data);
    }
    if (maxNode != NULL) {
        printf("Maximum element in the BST: %d\n", maxNode->data);
    }
    return 0;
}
```

Output: in-order traversal of the BST :20 30 40 50 60 70 80

Element 40 found in the BST

Minimum element in the BST :20

Maximum element in the BST: 80

11)

```
#include <stdio.h>
int searchRegistrationNumber(char regNos[][15], int size, const char* target) {
    for (int i = 0; i < size; i++) {
        if (strcmp(regNos[i], target) == 0) {
            return i;
        }
    }
```

```
    return -1;
}
int main() {
    int n;
    char regNos[100][15];
    char target[15];
    printf("Enter the number of registration numbers: ");
    scanf("%d", &n);
    printf("Enter the registration numbers:\n");
    for (int i = 0; i < n; i++) {
        printf("Registration number %d: ", i + 1);
        scanf("%s", regNos[i]);
    }
    printf("Enter the registration number to search for: ");
    scanf("%s", target);
    int index = searchRegistrationNumber(regNos, n, target);
    if (index != -1) {
        printf("Registration number '%s' found at index %d.\n", target, index);
    } else {
        printf("Registration number '%s' not found.\n", target);
    }
    return 0;
}
```

Input: number of reg no's: 3

Reg n01: 12345se

Reg no2: ertww452

Reg no3: 765efgt

Search: ertww452

Output: found at index 1

12.)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#define CHAR_COUNT 256
void countCharacters(const char* str, int* count) {
    while (*str) {
        count[(unsigned char)(*str)]++;
        str++;
    }
}
bool areAllCharactersPresent(const char* needle, const char* haystack) {
    int needleCount[CHAR_COUNT] = {0};
    int haystackCount[CHAR_COUNT] = {0};
    countCharacters(needle, needleCount);
    countCharacters(haystack, haystackCount);
    for (int i = 0; i < CHAR_COUNT; i++) {
        if (needleCount[i] > 0 && haystackCount[i] < needleCount[i]) {
            return false;
        }
    }
    return true;
}
int main() {
    char needle[100];
    char haystack[100];
    printf("Enter the needle string: ");
    scanf("%s", needle);

    printf("Enter the haystack string: ");
    scanf("%s", haystack);
```

```
    if (areAllCharactersPresent(needle, haystack)) {
        printf("All characters in needle are present in haystack.\n");
    } else {
        printf("Not all characters in needle are present in haystack.\n");
    }
    return 0;
}
```

Input: Enter the needle string: abc

Enter the haystack string: aabbcc

Output: All characters in needle are present in haystack.

13.)

```
#include <stdio.h>
#include <limits.h>
#define MAX_SIZE 100
#define RANGE 100
void countFrequency(int arr[], int size) {
    int freq[RANGE] = {0};
    for (int i = 0; i < size; i++) {
        if (arr[i] >= 0 && arr[i] < RANGE) {
            freq[arr[i]]++;
        } else {
            printf("Element %d is out of range.\n", arr[i]);
        }
    }
    printf("Element Frequencies:\n");
    for (int i = 0; i < RANGE; i++) {
        if (freq[i] > 0) {
            printf("Element %d: %d times\n", i, freq[i]);
        }
    }
}
```

```
int main() {
    int arr[MAX_SIZE];
    int size;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &size);
    if (size <= 0 || size > MAX_SIZE) {
        printf("Invalid array size.\n");
        return 1;
    }
    printf("Enter %d elements (0 to %d):\n", size, RANGE - 1);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    countFrequency(arr, size);
    return 0;
}
```

Input: Enter the number of elements in the array: 10

Enter 10 elements (0 to 99):

1 2 2 3 3 3 4 4 4 4

Output: Element Frequencies:

Element 1: 1 times

Element 2: 2 times

Element 3: 3 times

Element 4: 4 times

14.)

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX;
    int min_index;
```

```
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}
void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
```

```
            parent[v] = u;
          }
        }
    }
    printMST(parent, graph);
}
int main() {
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    primMST(graph);

    return 0;
}
```

output: Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

15.)

```
#include <stdio.h>
#define MAX_SIZE 100
void separateOddEven(int arr[], int size, int odd[], int* oddCount, int even[], int* evenCount) {
    *oddCount = 0;
    *evenCount = 0;

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

```
        if (arr[i] % 2 == 0) {
            even[*evenCount] = arr[i];
            (*evenCount)++;
        } else {
            odd[*oddCount] = arr[i];
            (*oddCount)++;
        }
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[MAX_SIZE];
    int odd[MAX_SIZE], even[MAX_SIZE];
    int size, oddCount, evenCount;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &size);
    if (size <= 0 || size > MAX_SIZE) {
        printf("Invalid array size.\n");
        return 1;
    }
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    separateOddEven(arr, size, odd, &oddCount, even, &evenCount);
    printf("Odd numbers: ");
```

```
    printArray(odd, oddCount);
    printf("Even numbers: ");
    printArray(even, evenCount);

    return 0;
}
```

Input:

Enter the number of elements in the array: 10

Enter 10 elements:

1 2 3 4 5 6 7 8 9 10

Output:

Odd numbers: 1 3 5 7 9

Even numbers: 2 4 6 8 10