**16. Write a C program to arrange a series of numbers using Insertion Sort**

Input:

#include <stdio.h>

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

Void displayArray(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};

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

printf("Original array: ");

displayArray(arr, n);

insertionSort(arr, n);

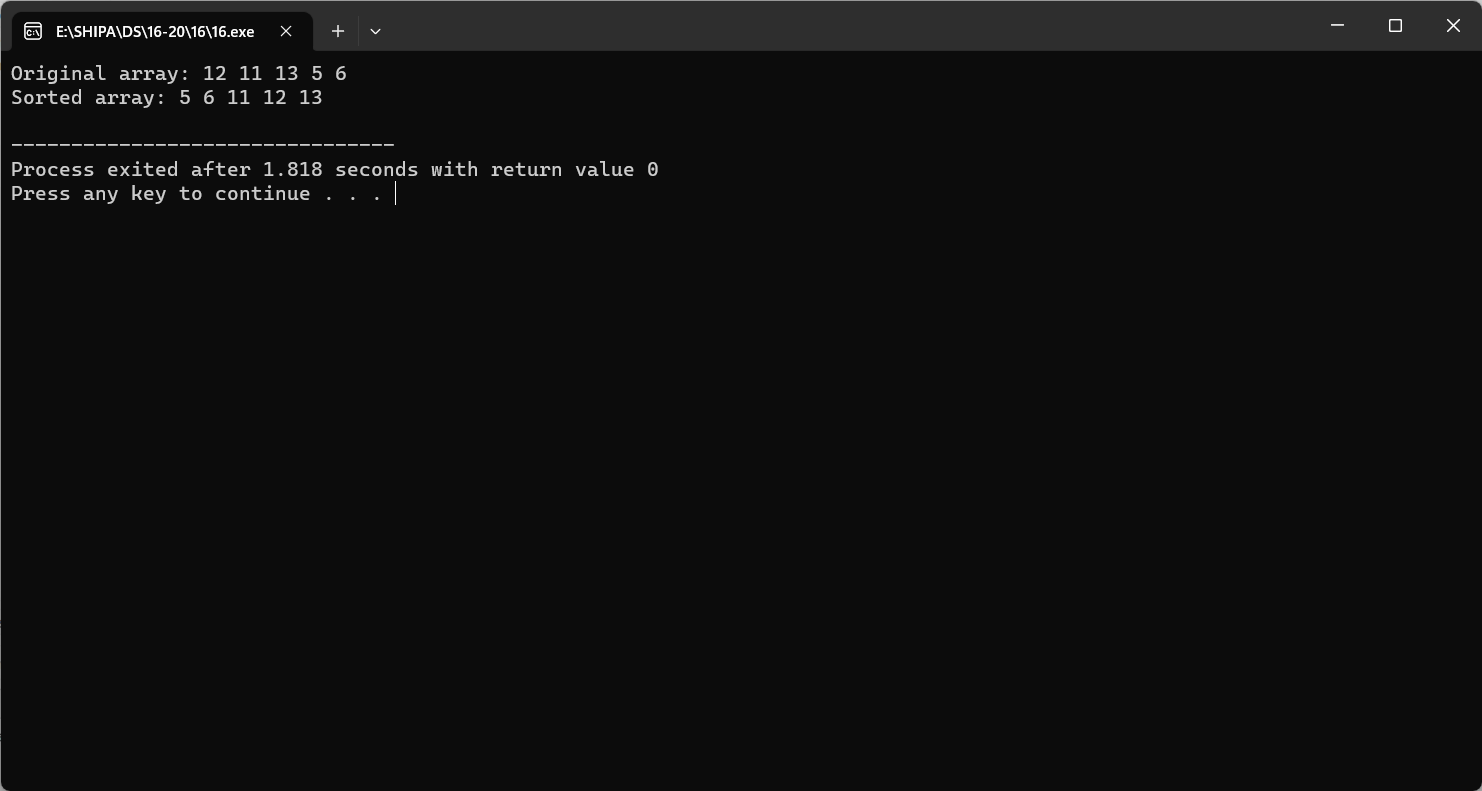
printf("Sorted array: ");

displayArray(arr, n);

return 0;

}

Output:



**17. Write a C program to arrange a series of numbers using Merge Sort**

Input:

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int leftArr[n1], rightArr[n2];

for (int i = 0; i < n1; i++) {

leftArr[i] = arr[left + i];

}

for (int j = 0; j < n2; j++) {

rightArr[j] = arr[mid + 1 + j];

}

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

}

else {

arr[k] = rightArr[j];

j++;

}

k++;

}

while (i < n1) {arr[k] = leftArr[i];

i++;

k++;

}

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void displayArray(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]);

printf("Original array: ");

displayArray(arr, n);

mergeSort(arr, 0, n - 1);

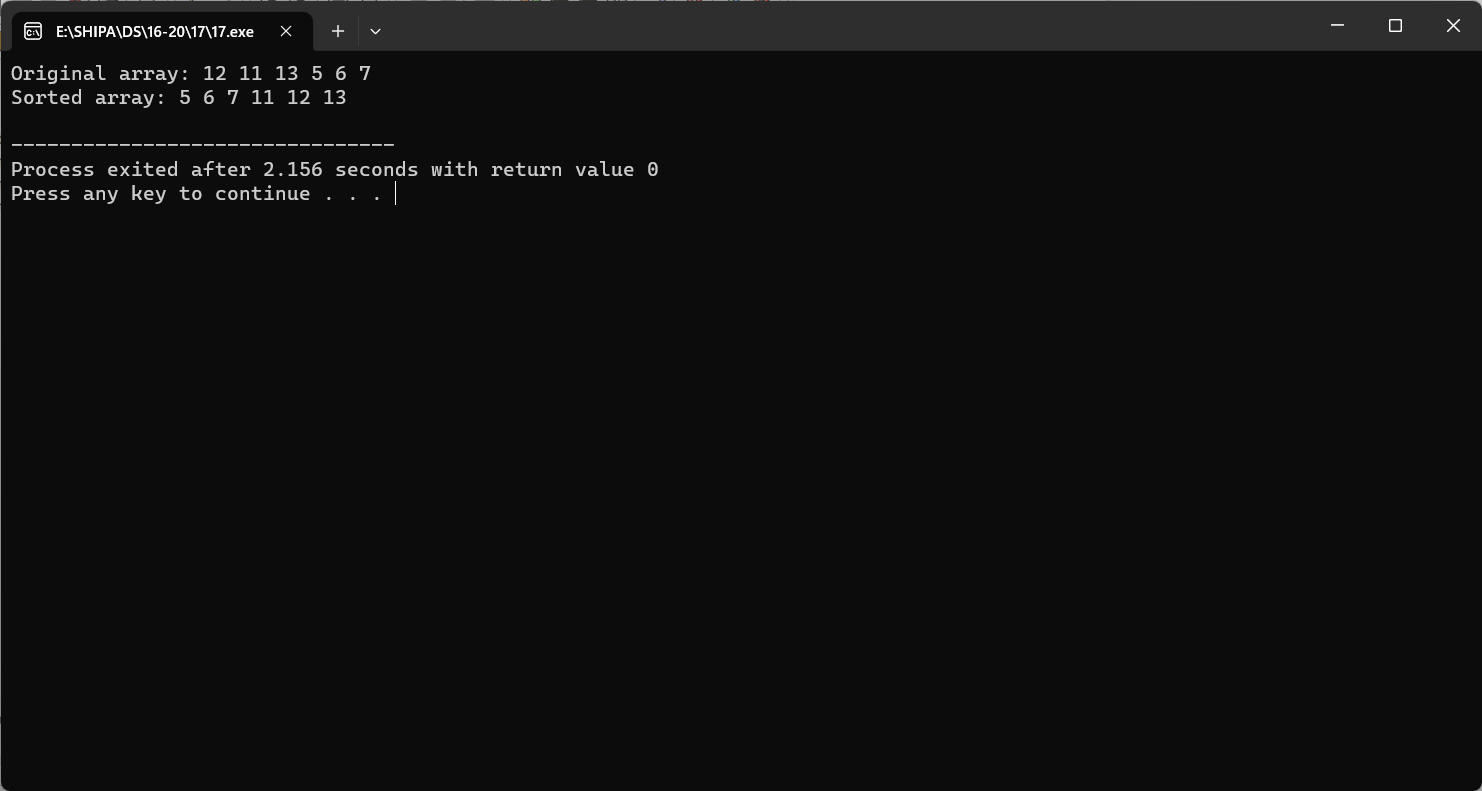
printf("Sorted array: ");

displayArray(arr, n);

return 0;

}

Output:



**18. Write a C program to arrange a series of numbers using Quick Sort**

Input:

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);}

}

void displayArray(int arr[], int n) {

for (int i = 0; i < n; i++) {

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

}

printf("\n");

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

printf("Original array: ");

displayArray(arr, n);

quickSort(arr, 0, n - 1);

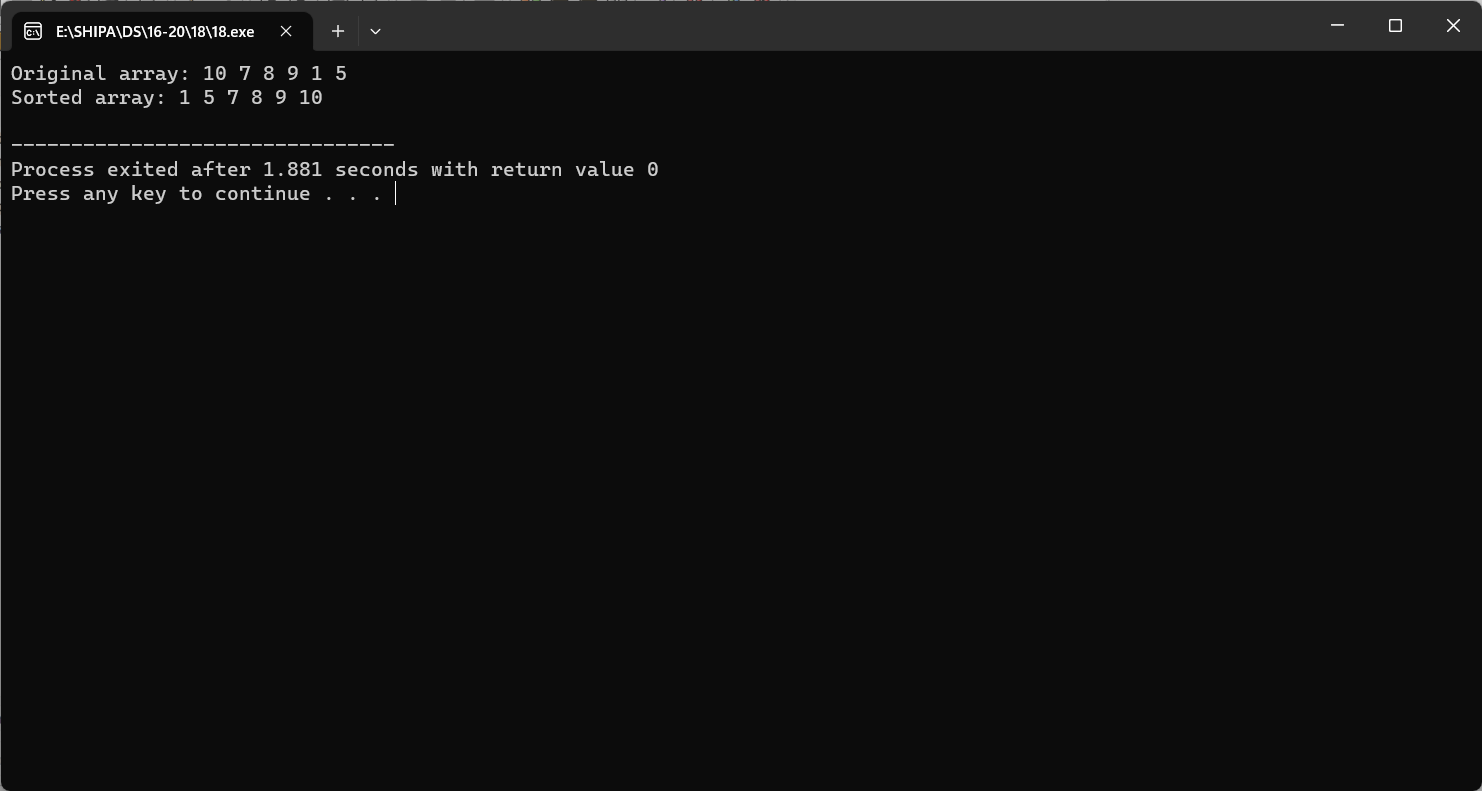
printf("Sorted array: ");

displayArray(arr, n);

return 0;

}

Output:



**19. Write a C program to implement Heap sort**

Input:

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

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 displayArray(int arr[], int n) {

for (int i = 0; i < n; i++) {

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

}printf("\n");

}

int main() {

int arr[] = {6456,594,9,1,949, 5, 6, 7};

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

displayArray(arr, n);

heapSort(arr, n);

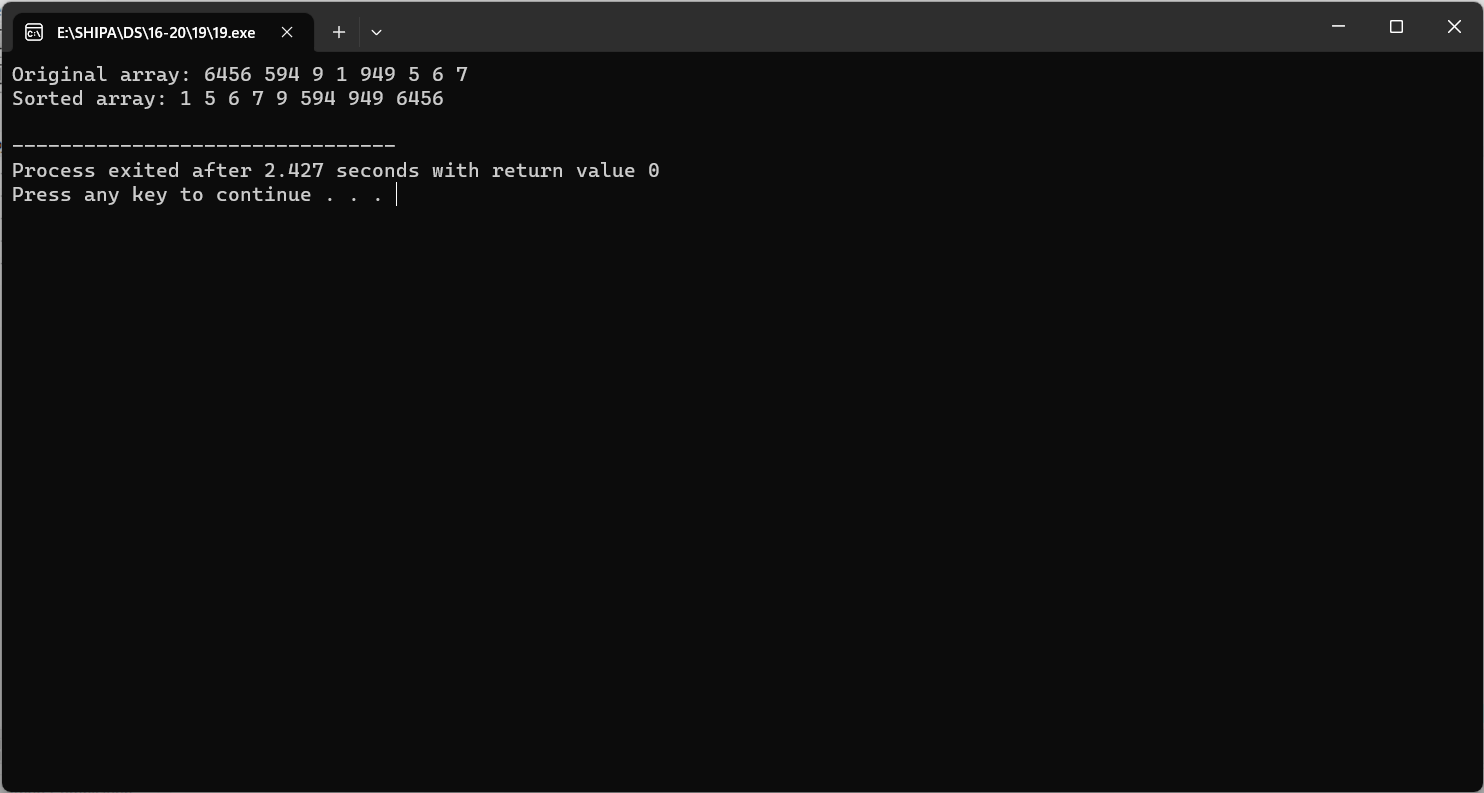
printf("Sorted array: ");

displayArray(arr, n);

return 0;

}

Output:



**20. Write a program to perform the following operations:**

**a) Insert an element into a AVL tree**

**b) Delete an element from a AVL tree**

**c) Search for a key element in a AVL tree**

Input:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node \*left;

struct Node \*right;int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int getHeight(struct Node \*node) {

if (node == NULL) {

return 0;

}

return node->height;

}

int getBalance(struct Node \*node) {

if (node == NULL) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

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(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(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(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

return y;

}

struct Node \*insert(struct Node \*root, int key) {

if (root == NULL) {

return newNode(key);

}

if (key < root->key) {

root->left = insert(root->left, key);

} else if (key > root->key) {

root->right = insert(root->right, key);

} else {

return root; // Duplicate keys not allowed

}

root->height = 1 + max(getHeight(root->left), getHeight(root->right));

int balance = getBalance(root);

if (balance > 1 && key < root->left->key) {

return rightRotate(root);

}

if (balance < -1 && key > root->right->key) {

return leftRotate(root);

}

if (balance > 1 && key > root->left->key) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && key < root->right->key) {

root->right = rightRotate(root->right);

return leftRotate(root);

}return root;

}

struct Node \*findMin(struct Node \*root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

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 = findMin(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

}

if (root == NULL) {

return root;

}

root->height = 1 + max(getHeight(root->left), getHeight(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;

}

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

if (root == NULL || root->key == key) {

return root;

}

if (key < root->key) {

return search(root->left, key);

}

return search(root->right, key);

}

void inorderTraversal(struct Node \*root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->key);

inorderTraversal(root->right);

}

}

int main() {

struct 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("Inorder traversal after insertion: ");

inorderTraversal(root);

printf("\n");

root = deleteNode(root, 30);

printf("Inorder traversal after deletion: ");

inorderTraversal(root);

printf("\n");

int keyToSearch = 40;

struct Node \*searchResult = search(root, keyToSearch);

if (searchResult) {

printf("Key %d found in the tree.\n", keyToSearch);

} else {

printf("Key %d not found in the tree.\n", keyToSearch);

}

return 0;

}

Output:

