

**Vellore Institute of Technology, Vellore**

**Course Name: Data Structures & Algorithms**

**Course Code: PMCA501P**

**Student Name: Barsha Routh**

**Instructor’s Name- Dr. Vijayan E**

**Submission Date: 21st October 2024**

**CycleSheet 2**

1.Linear search for Sorted Array

#include<stdio.h>

int linearSearchSorted(int arr[], int n, int data) {

    for (int i = 0; i < n; i++) {

        if (arr[i] == data) {

            printf("Element found at location %d\n", i + 1);

            return i;

        } else if (arr[i] > data) {

            break;

        }

    }

    return -1;

}

void main() {

    int n, data;

    printf("Enter the number of elements (sorted array): \n");

    scanf("%d", &n);

    int arr[n];

    printf("Enter sorted array elements: \n");

    for (int i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }

    printf("Elements are: ");

    for (int i = 0; i < n; i++) {

        printf("%d ", arr[i]);

    }

    printf("\nEnter data to search: ");

    scanf("%d", &data);

    // Call linear search for sorted array

    int result = linearSearchSorted(arr, n, data);

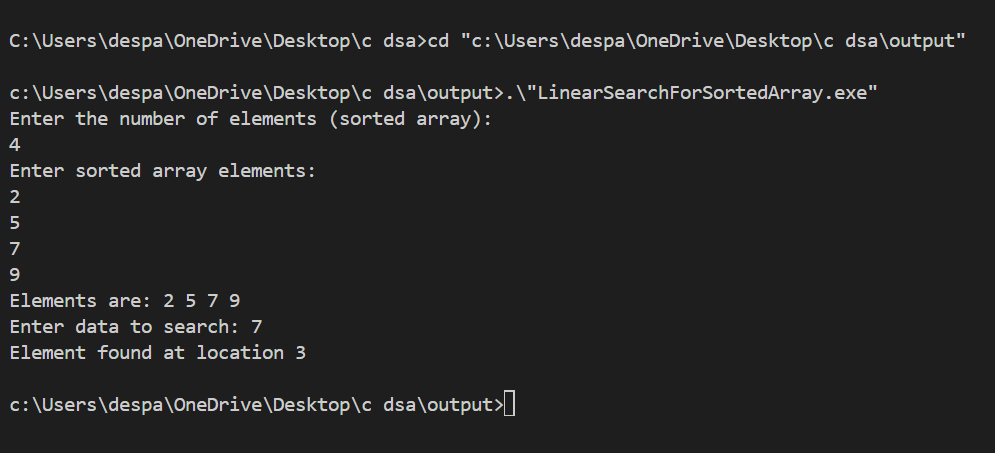
    if(result == -1) {

        printf("Element not found\n");

    }

}

Output-



2.Linear search for unsorted Array

#include<stdio.h>

int linearSearch(int arr[],int n, int data){

    int found=0;

    for (int i = 0; i < n; i++)

    {

        if(arr[i]==data){

            printf("Element found at location %d ",i+1);

            found=1;

            break;

        }

    }

    if(found ==0)

        return -1;

}

void main(){

    int n,data;

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

    scanf("%d",&n);

    int arr[n];

    printf("Enter array elements: \n");

    for (int i = 0; i < n; i++)

    {

        scanf("%d",&arr[i]);

    }

    printf("Elements are: ");

    for (int i = 0; i < n; i++)

    {

        printf("%d ",arr[i]);

    }

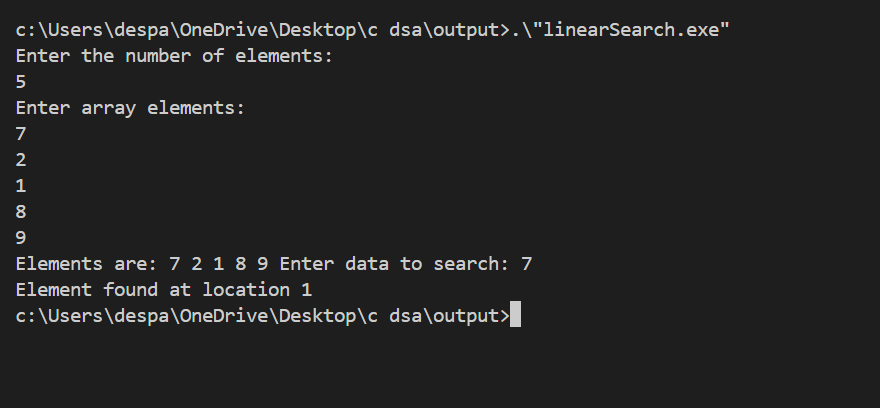
    printf("Enter data to search: ");

    scanf("%d",&data);

    linearSearch(arr,n,data);

}

Output-



3.Binary Search

int binarySearch(int arr[],int n ,int data){

    int low=0, high=n-1;

    while(low<=high){

        int mid=(low+high)/2;

            if(data==arr[mid]){

                return mid+1;

            }

            else if (data<arr[mid])

            {

                high=mid-1;

            }

            else{

                low=mid+1;

            }

    }

    return -1;

}

int main(){

    int n,data;

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

    scanf("%d",&n);

    int arr[n];

    printf("Enter array elements: \n");

    for (int i = 0; i < n; i++)

    {

        scanf("%d",&arr[i]);

    }

    printf("Elements are: ");

    for (int i = 0; i < n; i++)

    {

        printf("%d ",arr[i]);

    }

    printf("Enter data to search: ");

    scanf("%d",&data);

    int loc=binarySearch(arr,n,data);

    if(loc==-1){

        printf("Element not found ");

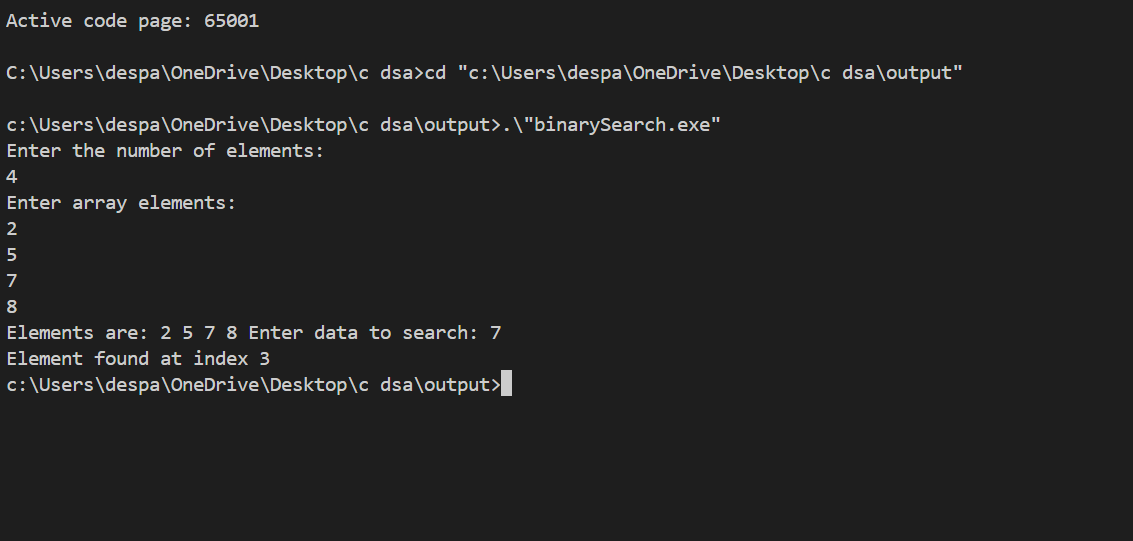
    }

    else

        printf("Element found at index %d",binarySearch(arr,n,data));

}

Output-



4.Insertion Sort

#include<stdio.h>

// Insertion Sort

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

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

int temp = arr[i];

int j = i - 1;

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

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

j--;

}

arr[j+1] = temp;

}

}

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements: ");

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

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

}

insertionSort(arr, n);

printf("After Sorting: ");

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

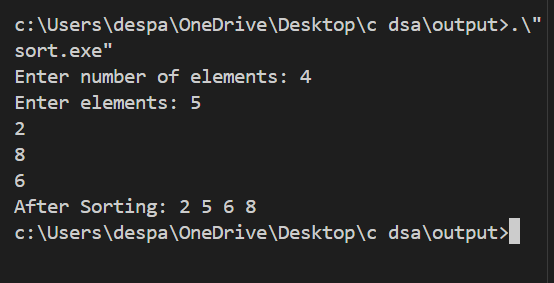
printf("%d ", arr[i]);

}

return 0;

}

Output-



5.Merge Sort

#include<stdio.h>

// Merge function

void merge(int arr[], int lb, int mid, int ub, int n) {

int brr[n];

int i = lb, j = mid + 1, k = lb;

while (i <= mid && j <= ub) {

if (arr[i] <= arr[j]) {

brr[k++] = arr[i++];

} else {

brr[k++] = arr[j++];

}

}

while (i <= mid) brr[k++] = arr[i++];

while (j <= ub) brr[k++] = arr[j++];

for (int i = lb; i <= ub; i++) arr[i] = brr[i];

}

// Merge Sort function

void mergeSort(int arr[], int lb, int ub, int n) {

if (lb < ub) {

int mid = (lb + ub) / 2;

mergeSort(arr, lb, mid, n);

mergeSort(arr, mid+1, ub, n);

merge(arr, lb, mid, ub, n);

}

}

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements: ");

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

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

}

mergeSort(arr, 0, n-1, n);

printf("After Sorting: ");

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

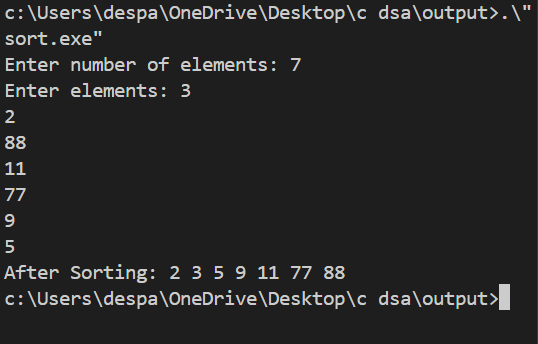
printf("%d ", arr[i]);

}

return 0;

}

Output-



6. Bucket Sort

#include <stdio.h>

#include <stdlib.h>

// Function to perform insertion sort on each bucket

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

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

int temp = arr[i];

int j = i - 1;

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

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

j--;

}

arr[j + 1] = temp;

}

}

// Bucket Sort function

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

// Find the maximum value in the array to determine bucket range

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

// Create buckets (each bucket is a dynamic array)

int bucketCount = 10; // For simplicity, using 10 buckets

int \*\*buckets = (int \*\*)malloc(bucketCount \* sizeof(int \*));

int \*bucketSizes = (int \*)malloc(bucketCount \* sizeof(int)); // To keep track of bucket sizes

// Initialize buckets and sizes

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

buckets[i] = (int \*)malloc(n \* sizeof(int));

bucketSizes[i] = 0;

}

// Distribute elements into buckets

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

int bucketIndex = (arr[i] \* bucketCount) / (max + 1); // Normalize the value to a bucket index

buckets[bucketIndex][bucketSizes[bucketIndex]++] = arr[i];

}

// Sort each bucket and concatenate the result

int index = 0;

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

insertionSort(buckets[i], bucketSizes[i]); // Sort the bucket

for (int j = 0; j < bucketSizes[i]; j++) {

arr[index++] = buckets[i][j]; // Concatenate sorted buckets into the original array

}

}

// Free allocated memory for buckets

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

free(buckets[i]);

}

free(buckets);

free(bucketSizes);

}

// Main function to test bucket sort

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements: ");

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

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

}

printf("Before Sorting: ");

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

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

}

bucketSort(arr, n);

printf("\nAfter Sorting: ");

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

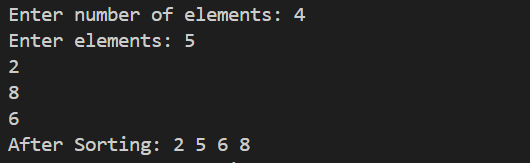
printf("%d ", arr[i]);

}

return 0;

}

Output-



7. Shell Sort

#include<stdio.h>

// Shell Sort

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

for (int gap = n / 2; gap >= 1; gap /= 2) {

for (int j = gap; j < n; j++) {

for (int i = j-gap; i >= 0; i -= gap) {

if (arr[i+gap] > arr[i]) {

break;

} else {

int temp = arr[i+gap];

arr[i+gap] = arr[i];

arr[i] = temp;

}

}

}

}

}

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements: ");

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

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

}

shellSort(arr, n);

printf("After Sorting: ");

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

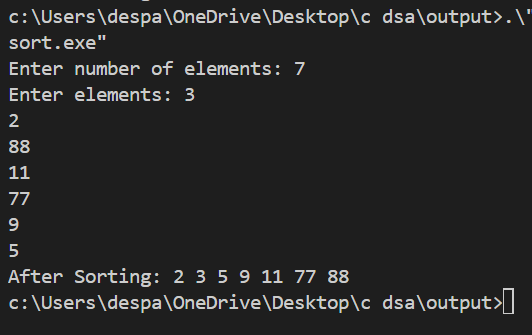
printf("%d ", arr[i]);

}

return 0;

}

Output-



8. Binary Search Tree

#include <stdio.h>

#include <stdlib.h>

struct node {

    int data;

    struct node \*left;

    struct node \*right;

};

struct node \*root;

void inorder(struct node \*root) {

    if (root == NULL) {

        return;

    }

    inorder(root->left);

    printf("%d ", root->data);

    inorder(root->right);

}

// Function to find the minimum value node in a subtree (used for in-order successor)

struct node\* findMin(struct node\* parent) {

    while (parent->left != NULL) {

        parent = parent->left;

    }

    return parent;

}

// Function to delete a node from the BST

struct node\* deleteNode(struct node\* parent, int d) {

    if (parent == NULL) {

        return parent;  // Base case: Node to be deleted is not found

    }

    // Traverse the tree to find the node to delete

    if (d < parent->data) {

        parent->left = deleteNode(parent->left, d);

    } else if (d > parent->data) {

        parent->right = deleteNode(parent->right, d);

    } else {

        // Node to be deleted found (parent->data == d)

        // Case 1: No child (Leaf node)

        if (parent->left == NULL && parent->right == NULL) {

            free(parent);

            return NULL;

        }

        // Case 2: One child

        else if (parent->left == NULL) {

            struct node\* temp = parent->right;

            free(parent);

            return temp;

        } else if (parent->right == NULL) {

            struct node\* temp = parent->left;

            free(parent);

            return temp;

        }

        // Case 3: Two children

        else {

            struct node\* temp = findMin(parent->right);  // Find in-order successor (minimum in right subtree)

            parent->data = temp->data;  // Replace data with in-order successor's data

            parent->right = deleteNode(parent->right, temp->data);  // Delete the in-order successor

        }

    }

    return parent;

}

struct node\* add(struct node\* parent, int d) {

    if (parent == NULL) {

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

        parent->data = d;

        parent->left = NULL;

        parent->right = NULL;

    } else if (d < parent->data) {

        parent->left = add(parent->left, d);

    } else {

        parent->right = add(parent->right, d);

    }

    return parent;

}

int main() {

    int n, d;

    root = NULL;

    printf("Enter number of elements you want to insert: \n");

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {

        printf("Enter data to be inserted: ");

        scanf("%d", &d);

        root = add(root, d);

    }

    printf("Inorder traversal before deletion:\n");

    inorder(root);

    printf("\n");

    printf("Enter the element to delete: ");

    scanf("%d", &d);

    root = deleteNode(root, d);

    printf("Inorder traversal after deletion:\n");

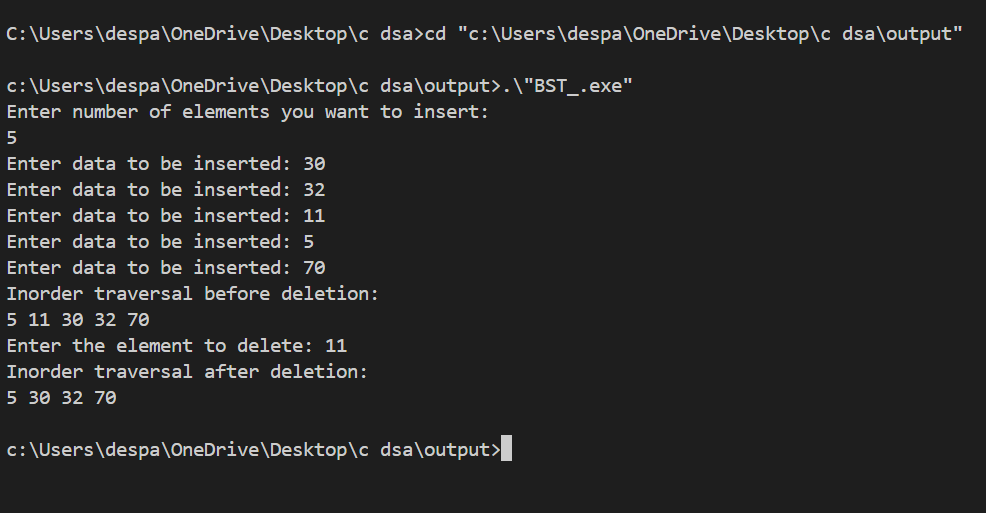
    inorder(root);

    printf("\n");

    return 0;

}

Output-



9. Binary Search Tree traversals

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a tree node

struct Node {

int data;

struct Node \*left;

struct Node \*right;

};

// Function to create a new tree node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a new node into the BST

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) {

return createNode(data);

}

if (data < node->data) {

node->left = insert(node->left, data);

} else {

node->right = insert(node->right, data);

}

return node;

}

// Pre-order Traversal (Root, Left, Right)

void preOrderTraversal(struct Node\* root) {

if (root != NULL) {

printf("%d ", root->data); // Visit the root

preOrderTraversal(root->left); // Traverse the left subtree

preOrderTraversal(root->right); // Traverse the right subtree

}

}

// In-order Traversal (Left, Root, Right)

void inOrderTraversal(struct Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left); // Traverse the left subtree

printf("%d ", root->data); // Visit the root

inOrderTraversal(root->right); // Traverse the right subtree

}

}

// Post-order Traversal (Left, Right, Root)

void postOrderTraversal(struct Node\* root) {

if (root != NULL) {

postOrderTraversal(root->left); // Traverse the left subtree

postOrderTraversal(root->right); // Traverse the right subtree

printf("%d ", root->data); // Visit the root

}

}

// Main function to test the BST and traversals

int main() {

struct Node\* root = NULL;

// Insert elements into the BST

root = insert(root, 10);

insert(root, 5);

insert(root, 20);

insert(root, 3);

insert(root, 7);

insert(root, 15);

insert(root, 30);

// Traversals

printf("Pre-order Traversal: ");

preOrderTraversal(root);

printf("\n");

printf("In-order Traversal: ");

inOrderTraversal(root);

printf("\n");

printf("Post-order Traversal: ");

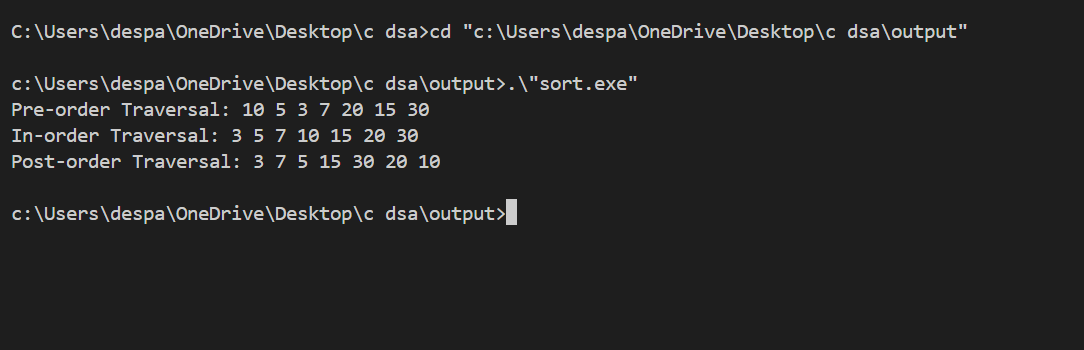
postOrderTraversal(root);

printf("\n");

return 0;

}

Output-



10.Binary Sort

#include <stdio.h>

// Function to perform binary search

int binarySearch(int arr[], int item, int low, int high) {

if (high <= low)

return (item > arr[low]) ? (low + 1) : low;

int mid = (low + high) / 2;

if (item == arr[mid])

return mid + 1;

if (item > arr[mid])

return binarySearch(arr, item, mid + 1, high);

return binarySearch(arr, item, low, mid - 1);

}

// Function to sort array using Binary Insertion Sort

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

int i, j, selected, location;

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

selected = arr[i];

// Find location where selected element should be inserted

location = binarySearch(arr, selected, 0, i - 1);

// Move all elements after location to create space

for (j = i - 1; j >= location; --j)

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

arr[location] = selected;

}

}

// Main function to test the Binary Insertion Sort

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter elements: ");

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

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

}

printf("Before Sorting: ");

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

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

}

binaryInsertionSort(arr, n);

printf("\nAfter Sorting: ");

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

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

}

return 0;

}

Output-

