**DATA STRUCTURE LAB PROGRAMS**

Name **:** A.Nanda kumar reddy

Reg.No.  **: 192365034**

Subject **: Data Structure**

Course code **: CSA0386**

**Lab Programs 1 to 30**

**DATA STRUCTURE LAB PROGRAMS**

1. **Write a program in c to convert/Reversing a 32 bit signed**.

**Program:**

#include <stdio.h>

void printBinary(int num) {

unsigned int mask = 1 << 31;

for (int i = 0; i < 32; i++) {

printf("%d", (num & mask) ? 1 : 0);

mask >>= 1;

if ((i + 1) % 8 == 0) {

printf(" ");

}

}

printf("\n");

}

int main() {

int number;

printf("Enter a 32-bit signed integer: ");

scanf("%d", &number);

printf("Binary representation: ");

printBinary(number);

return 0;

}

1. **Write a c program to check for a valid string.**

**Program:**

#include <stdio.h>

#include <ctype.h>

int isValidString(const char \*str) {

while (\*str) {

if (!isalnum(\*str) && !isspace(\*str)) {

return 0;

}

str++;

}

return 1;

}

int main() {

char str[100];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin);

str[strcspn(str, "\n")] = 0;

if (isValidString(str)) {

printf("The string is valid.\n");

} else {

printf("The string is not valid.\n");

}

return 0;

}

1. **Write a c program to merge two arrays.**

**Program:**

#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 arr1[] = {1, 2, 3, 4, 5};

int arr2[] = {6, 7, 8, 9, 10};

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;

}

1. **Write a program in c to count the total number of duplicate elements in an array.**

**Programs:**

#include <stdio.h>

int countDuplicates(int arr[], int size) {

int count = 0;

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

for (int j = i + 1; j < size; j++) {

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

count++;

break;

}

}

}

return count;

}

int main() {

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

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

int duplicateCount = countDuplicates(arr, size);

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

return 0;

}

1. **Implement c program to merge two linked list.**

**Program:**

#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 appendNode(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

struct Node\* last = \*head;

if (\*head == NULL) {

\*head = newNode;

return;

}

while (last->next != NULL) {

last = last->next;

}

last->next = newNode;

}

struct Node\* mergeLists(struct Node\* head1, struct Node\* head2) {

if (head1 == NULL) return head2;

if (head2 == NULL) return head1;

struct Node\* mergedHead = NULL;

if (head1->data <= head2->data) {

mergedHead = head1;

mergedHead->next = mergeLists(head1->next, head2);

} else {

mergedHead = head2;

mergedHead->next = mergeLists(head1, head2->next);

}

return mergedHead;

}

void printList(struct Node\* head) {

while (head != NULL) {

printf("%d -> ", head->data);

head = head->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head1 = NULL;

struct Node\* head2 = NULL;

appendNode(&head1, 1);

appendNode(&head1, 3);

appendNode(&head1, 5);

appendNode(&head2, 2);

appendNode(&head2, 4);

appendNode(&head2, 6);

printf("First Linked List: ");

printList(head1);

printf("Second Linked List: ");

printList(head2);

struct Node\* mergedHead = mergeLists(head1, head2);

printf("Merged Linked List: ");

printList(mergedHead);

return 0; }

1. **Implement a C program given an array of reg number need to search for the particular reg number.**

**Program:**

#include <stdio.h>

int searchRegNumber(int arr[], int size, int regNumber) {

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

if (arr[i] == regNumber) {

return I; }

}

return -1; }

int main() {

int regNumbers[] = {1001, 1002, 1003, 1004, 1005};

int size = sizeof(regNumbers) / sizeof(regNumbers[0]);

int regNumberToFind;

printf("Enter the registration number to search: ");

scanf("%d", &regNumberToFind);

int index = searchRegNumber(regNumbers, size, regNumberToFind);

if (index != -1) {

printf("Registration number %d found at index %d.\n", regNumberToFind, index);

} else {

printf("Registration number %d not found.\n", regNumberToFind);

}

return 0;

}

1. **Write a program in C to separate odd and even integers into separate arrays.**

**Program:**

#include <stdio.h>

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)++;

}

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

int odd[size], even[size];

int oddCount, evenCount;

separateOddEven(arr, size, odd, &oddCount, even, &evenCount);

printf("Odd numbers: ");

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

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

}

printf("\n");

printf("Even numbers: ");

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

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

}

printf("\n");

return 0;

}

1. **Implement a C program to identify the location of element in given array.**

**Program:**

#include <stdio.h>

int findElement(int arr[], int size, int element) {

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

if (arr[i] == element) {

return i;

}

}

return -1;

}

int main() {

int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

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

int elementToFind;

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

scanf("%d", &elementToFind);

int index = findElement(arr, size, elementToFind);

if (index != -1) {

printf("Element %d found at index %d.\n", elementToFind, index);

} else {

printf("Element %d not found in the array.\n", elementToFind);

}

return 0;

}

1. **Implement C program to find the sum of the Fibonacci series using recursion.**

**Program:**

#include <stdio.h>

int fibonacci(int n) {

if (n <= 1) {

return n;

}

return fibonacci(n - 1) + fibonacci(n - 2);

}

int sumOfFibonacci(int n) {

if (n <= 1) {

return n;

}

return fibonacci(n) + sumOfFibonacci(n - 1); }

int main() {

int n;

printf("Enter the number of terms: ");

scanf("%d", &n);

int sum = sumOfFibonacci(n - 1); // We use n-1 because the series starts from 0

printf("Sum of the first %d Fibonacci numbers is: %d\n", n, sum);

return 0;

}

**10) Implement C program to find the factorial of n with recursion.**

**Program:**

#include <stdio.h>

int factorial(int n) {

if (n == 0) {

return 1;

}

return n \* factorial(n - 1);

}

int main() {

int n;

printf("Enter a positive integer: ");

scanf("%d", &n);

if (n < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

printf("Factorial of %d is: %d\n", n, factorial(n));

}

return 0;

}

**11) Implement a C program to insert and delete elements in AVL tree.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

int height;

};

int height(struct Node\* node) {

return node ? node->height : 0;

}

int getBalance(struct Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

struct Node\* createNode(int data) {

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

node->data = data;

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 = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

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 = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

return y;

}

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

if (!node) return createNode(data);

if (data < node->data)

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

else if (data > node->data)

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

else

return node; // Duplicate data is not allowed

node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

int balance = getBalance(node);

if (balance > 1 && data < node->left->data)

return rightRotate(node);

if (balance < -1 && data > node->right->data)

return leftRotate(node);

if (balance > 1 && data > node->left->data) {

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

return rightRotate(node);

}

if (balance < -1 && data < node->right->data) {

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 data) {

if (!root) return root;

if (data < root->data)

root->left = deleteNode(root->left, data);

else if (data > root->data)

root->right = deleteNode(root->right, data);

else {

if (!root->left) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (!root->right) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = minValueNode(root->right);

root->data = temp->data;

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

}

if (!root) return root;

root->height = 1 + (height(root->left) > height(root->right) ? 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 inOrder(struct Node\* root) {

if (root) {

inOrder(root->left);

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

inOrder(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("In-order traversal of the AVL tree: ");

inOrder(root);

printf("\n");

root = deleteNode(root, 10);

root = deleteNode(root, 30);

printf("In-order traversal after deletion: ");

inOrder(root);

printf("\n");

return 0;

}

**12) Implement a C program to check valid stack.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

int arr[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 value) {

if (isFull(s)) {

printf("Stack Overflow\n");

return;

}

s->arr[++(s->top)] = value;

}

int pop(Stack\* s) {

if (isEmpty(s)) {

printf("Stack Underflow\n");

return -1;

}

return s->arr[(s->top)--];

}

void checkStackValidity() {

Stack s;

initStack(&s);

printf("Pushing elements onto the stack...\n");

for (int i = 1; i <= 5; i++) {

push(&s, i);

printf("Pushed %d\n", i);

}

printf("\nPopping elements from the stack...\n");

for (int i = 1; i <= 5; i++) {

int value = pop(&s);

if (value != -1) {

printf("Popped %d\n", value);

} else {

printf("Error: Stack is empty\n");

}

}

printf("\nAttempting to pop from an empty stack...\n");

pop(&s);

}

int main() {

checkStackValidity();

return 0;

}

**13) Implement a C program to find the shortest path in graph.**

**Program:**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#define V 9

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

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

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

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[], int n) {

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

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

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

}

}

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

int dist[V]; // The output array dist[i] holds 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}

};

int source = 0;

dijkstra(graph, source);

return 0;

}

**14) Travelling salesman program.**

**Program:**

#include <stdio.h>

#include <limits.h>

#define V 4

int min(int a, int b) {

return (a < b) ? a : b;

}

int tsp(int graph[V][V], int dp[1 << V][V], int mask, int pos) {

if (mask == (1 << V) - 1) {

return graph[pos][0]; // Return to the starting city

}

if (dp[mask][pos] != -1) {

return dp[mask][pos];

}

int ans = INT\_MAX;

for (int city = 0; city < V; city++) {

if ((mask & (1 << city)) == 0) {

int newAns = graph[pos][city] + tsp(graph, dp, mask | (1 << city), city);

ans = min(ans, newAns);

}

}

dp[mask][pos] = ans;

return 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;

}

}

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

return 0;

}

**15) Binary search tree-search for a element ,insert element and remove element.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

Node\* insert(Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

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

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

}

return root;

}

Node\* search(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);

}

}

Node\* findMin(Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

Node\* remove(Node\* root, int data) {

if (root == NULL) {

return root;

}

if (data < root->data) {

root->left = remove(root->left, data);

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

root->right = remove(root->right, data);

} else {

if (root->left == NULL) {

Node\* temp = root->right;

free(root);

return temp;

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

Node\* temp = root->left;

free(root);

return temp;

}

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = remove(root->right, temp->data);

}

return root;

}

void inOrder(Node\* root) {

if (root != NULL) {

inOrder(root->left);

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

inOrder(root->right);

}

}

int main() {

Node\* root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

printf("In-order traversal of the BST: ");

inOrder(root);

printf("\n");

int key = 40;

Node\* foundNode = search(root, key);

if (foundNode != NULL) {

printf("Node %d found in the BST.\n", key);

} else {

printf("Node %d not found in the BST.\n", key);

}

printf("Removing node 20 from the BST.\n");

root = remove(root, 20);

printf("In-order traversal after removal: ");

inOrder(root);

printf("\n");

return 0;

}

**16) Implement a C Program for Haystack. There are two strings needle and haystack (or hay). You need to check if all the characters in the needle are present in haystack or not. If yes then return True (1) or False (0) .**

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define ASCII\_SIZE 256

bool areCharactersPresent(const char\* haystack, const char\* needle) {

int haystackCount[ASCII\_SIZE] = {0};

for (int i = 0; haystack[i]; i++) {

haystackCount[(unsigned char)haystack[i]]++;

}

for (int i = 0; needle[i]; i++) {

if (haystackCount[(unsigned char)needle[i]] == 0) {

return false;

}

}

return true;

}

int main() {

char haystack[100], needle[100];

printf("Enter the haystack string: ");

fgets(haystack, sizeof(haystack), stdin);

haystack[strcspn(haystack, "\n")] = '\0';

printf("Enter the needle string: ");

fgets(needle, sizeof(needle), stdin);

needle[strcspn(needle, "\n")] = '\0';

if (areCharactersPresent(haystack, needle)) {

printf("True (1)\n");

} else {

printf("False (0)\n");

}

return 0;

}

**17) Write a program in C to count the frequency of each element of an array.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 1000

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

int\* freq = (int\*)malloc(MAX \* sizeof(int));

if (freq == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

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

freq[i] = 0;

}

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

freq[arr[i]]++;

}

printf("Element Frequency\n");

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

if (freq[i] > 0) {

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

}

}

free(freq);

}

int main() {

int arr[100], n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

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

}

countFrequency(arr, n);

return 0;

}

**18) Implement a C Program for Given Graph convert array and print minimum edges (Prim’s Algorithm).**

**Program:**

#include <stdio.h>

#include <limits.h>

#define V 5

int minKey(int key[], bool mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void printMST(int parent[], int graph[V][V]) {

printf("Edge \t Weight\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]; // Array to store the constructed MST

int key[V]; // Key values used to pick the minimum weight edge

bool mstSet[V]; // To represent the set of vertices included in MST

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

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++) {

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

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;

}

**19) Implement a C Program to perform heap sort.**

**Program:**

#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 printArray(int arr[], int size) {

for (int i = 0; i < size; 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("Unsorted array: \n");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**20) Implement a C Program to perform quick sort.**

**Program:**

#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; 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 pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; 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("Unsorted array: \n");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**21) Implement a C Program for an Array to sort in ascending order.**

**Program:**

#include <stdio.h>

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

int i, j, temp;

for (i = 0; i < n - 1; i++)

for (j = 0; j < n - i - 1; j++)

if (arr[j] > arr[j + 1]) {

temp = arr[j];

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

arr[j + 1] = temp;

}

}

void printArray(int arr[], int size) {

int i;

for (i = 0; i < size; i++)

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

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

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

printf("Unsorted array: \n");

printArray(arr, n);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**22) Implement a C Program to Print no of nodes in the given linked list.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node \*next;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

int countNodes(Node \*head) {

int count = 0;

Node \*current = head;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

void printList(Node \*head) {

Node \*current = head;

while (current != NULL) {

printf("%d -> ", current->data);

current = current->next;

}

printf("NULL\n");

}

int main() {

Node \*head = createNode(10);

head->next = createNode(20);

head->next->next = createNode(30);

head->next->next->next = createNode(40);

printf("Linked List: ");

printList(head);

int numNodes = countNodes(head);

printf("Number of nodes in the linked list: %d\n", numNodes);

Node \*current = head;

Node \*nextNode;

while (current != NULL) {

nextNode = current->next;

free(current);

current = nextNode;

}

return 0;

}

**23) A Implement a C Program to perform search for sorted elements.**

**Program:**

#include <stdio.h>

int binarySearch(int arr[], int size, int target) {

int left = 0, right = size - 1;

while (left <= right) {

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

if (arr[mid] == target) return mid;

if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

int main() {

int arr[] = {2, 5, 7, 10, 12, 15, 18, 21, 24, 30};

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

int target = 15;

int result = binarySearch(arr, size, target);

if (result != -1) printf("Element %d found at index %d.\n", target, result);

else printf("Element %d not found in the array.\n", target);

return 0;

}

**24) Implement a C Program to Print the index of repeated characters given in an array.**

**Program:**

#include <stdio.h>

#include <string.h>

#define MAX\_CHAR 256

void printRepeatedCharIndices(const char \*str) {

int length = strlen(str);

int count[MAX\_CHAR] = {0};

int indices[MAX\_CHAR][length];

for (int i = 0; i < MAX\_CHAR; i++) {

memset(indices[i], -1, sizeof(indices[i]));

}

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

char c = str[i];

indices[(unsigned char)c][count[(unsigned char)c]] = i;

count[(unsigned char)c]++;

}

printf("Repeated characters and their indices:\n");

for (int i = 0; i < MAX\_CHAR; i++) {

if (count[i] > 1) {

printf("Character '%c': ", i);

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

printf("%d ", indices[i][j]);

}

printf("\n");

}

}

}

int main() {

char str[100];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin);

str[strcspn(str, "\n")] = '\0';

printRepeatedCharIndices(str);

return 0;

}

**25) Implement a C Program for given set of Array elements - display 5th iterated element.**

**Program:**

#include <stdio.h>

#define ARRAY\_SIZE 10

int main() {

int array[ARRAY\_SIZE] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

if (ARRAY\_SIZE >= 5) {

printf("The 5th element of the array is: %d\n", array[4]);

} else {

printf("The array does not contain 5 elements.\n");

}

return 0;

}

**26) Implement a C Program to perform Palindrome using SLL.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

typedef struct Node {

char data;

struct Node\* next;

} Node;

Node\* createNode(char data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void insertEnd(Node\*\* head, char data) {

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

Node\* reverseList(Node\* head) {

Node\* prev = NULL;

Node\* curr = head;

Node\* next = NULL;

while (curr != NULL) {

next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

return prev;

}

int areEqual(Node\* head1, Node\* head2) {

while (head1 != NULL && head2 != NULL) {

if (head1->data != head2->data) {

return 0;

}

head1 = head1->next;

head2 = head2->next;

}

return head1 == NULL && head2 == NULL;

}

int isPalindrome(char\* str) {

Node\* head = NULL;

Node\* reversed = NULL;

for (int i = 0; str[i] != '\0'; i++) {

if (isalpha(str[i])) {

insertEnd(&head, tolower(str[i]));

}

}

reversed = reverseList(head);

return areEqual(head, reversed);

}

int main() {

char str[100];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin);

str[strcspn(str, "\n")] = '\0';

if (isPalindrome(str)) {

printf("The string is a palindrome.\n");

} else {

printf("The string is not a palindrome.\n");

}

return 0;

}

**27) Implement a C Program for Given unsorted array - Display missing element.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

void findMissingElements(int arr[], int size, int range) {

int\* present = (int\*)calloc(range + 1, sizeof(int));

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

if (arr[i] > 0 && arr[i] <= range) {

present[arr[i]] = 1;

}

}

printf("Missing elements:\n");

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

if (present[i] == 0) {

printf("%d ", i);

}

}

printf("\n");

free(present);

}

int main() {

int size;

printf("Enter the number of elements in the array: ");

scanf("%d", &size);

int\* arr = (int\*)malloc(size \* sizeof(int));

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

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

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

}

int range;

printf("Enter the maximum value in the range: ");

scanf("%d", &range);

findMissingElements(arr, size, range);

free(arr);

return 0;

}

**28) Implement a C Program for a Binary tree.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

Node\* insertNode(Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

void inOrder(Node\* root) {

if (root != NULL) {

inOrder(root->left);

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

inOrder(root->right);

}

}

void preOrder(Node\* root) {

if (root != NULL) {

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

preOrder(root->left);

preOrder(root->right);

}

}

void postOrder(Node\* root) {

if (root != NULL) {

postOrder(root->left);

postOrder(root->right);

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

}

}

void freeTree(Node\* root) {

if (root != NULL) {

freeTree(root->left);

freeTree(root->right);

free(root);

}

}

int main() {

Node\* root = NULL;

int choice, data;

while (1) {

printf("\nBinary Tree Operations:\n");

printf("1. Insert node\n");

printf("2. In-order traversal\n");

printf("3. Pre-order traversal\n");

printf("4. Post-order traversal\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insertNode(root, data);

break;

case 2:

printf("In-order traversal: ");

inOrder(root);

printf("\n");

break;

case 3:

printf("Pre-order traversal: ");

preOrder(root);

printf("\n");

break;

case 4:

printf("Post-order traversal: ");

postOrder(root);

printf("\n");

break;

case 5:

freeTree(root);

exit(0);

default:

printf("Invalid choice. Please enter a number between 1 and 5.\n");

}

}

return 0;

}

**29) Implement a C Program for Array concatenation.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

void concatenateArrays(int \*arr1, int size1, int \*arr2, int size2, int \*result) {

for (int i = 0; i < size1; i++) {

result[i] = arr1[i];

}

for (int i = 0; i < size2; i++) {

result[size1 + i] = arr2[i];

}

}

void printArray(int \*arr, int size) {

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

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

}

printf("\n");

}

int main() {

int arr1[] = {1, 2, 3, 4};

int arr2[] = {5, 6, 7, 8};

int size1 = sizeof(arr1) / sizeof(arr1[0]);

int size2 = sizeof(arr2) / sizeof(arr2[0]);

int \*result = (int \*)malloc((size1 + size2) \* sizeof(int));

if (result == NULL) {

printf("Memory allocation failed\n");

return 1;

}

concatenateArrays(arr1, size1, arr2, size2, result);

printf("Concatenated Array:\n");

printArray(result, size1 + size2);

free(result);

return 0;

}

**30) Implement a C Program to perform stack using two queues.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

int front, rear, size;

int arr[MAX];

} Queue;

typedef struct {

Queue q1, q2;

} Stack;

void initQueue(Queue \*q) {

q->front = q->rear = q->size = 0;

}

int isQueueEmpty(Queue \*q) {

return q->size == 0;

}

void enqueue(Queue \*q, int value) {

if (q->size == MAX) {

printf("Queue is full\n");

return;

}

q->arr[q->rear] = value;

q->rear = (q->rear + 1) % MAX;

q->size++;

}

int dequeue(Queue \*q) {

if (isQueueEmpty(q)) {

printf("Queue is empty\n");

return -1;

}

int value = q->arr[q->front];

q->front = (q->front + 1) % MAX;

q->size--;

return value;

}

void push(Stack \*s, int value) {

enqueue(&s->q2, value);

while (!isQueueEmpty(&s->q1)) {

enqueue(&s->q2, dequeue(&s->q1)); }

Queue temp = s->q1;

s->q1 = s->q2;

s->q2 = temp;

}

int pop(Stack \*s) {

if (isQueueEmpty(&s->q1)) {

printf("Stack is empty\n");

return -1;

}

return dequeue(&s->q1);

}

void printStack(Stack \*s) {

Queue q = s->q1;

while (!isQueueEmpty(&q)) {

printf("%d ", dequeue(&q));

}

printf("\n");

}

int main() {

Stack stack;

initQueue(&stack.q1);

initQueue(&stack.q2);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

printf("Stack after pushes:\n");

printStack(&stack);

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

printf("Stack after pop:\n");

printStack(&stack);

return 0;

}