1. Write a program to search for an element in an array using binary and linear search.

#include <stdio.h>

// Function for Linear Search

int linearSearch(int arr[], int length, int target) {

int i;

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

if (arr[i] == target) {

return i; // Return the index if found

}

}

return -1; // Return -1 if not found

}

// Function for Binary Search

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

int low = 0;

int high = length - 1;

while (low <= high) {

int mid = low + (high - low) / 2; // Calculate mid index

if (arr[mid] == target) {

return mid; // Return the index if found

} else if (arr[mid] < target) {

low = mid + 1; // Search in the right half

} else {

high = mid - 1; // Search in the left half

}

}

return -1; // Return -1 if not found

}

int main() {

int search\_list[] = {3, 1, 9, 8, 7, 12, 56, 23, 89};

int length = sizeof(search\_list) / sizeof(search\_list[0]);

int target,i,j;

int linearResult, binaryResult;

// Linear Search

printf("Enter a value to search using Linear Search: ");

scanf("%d", &target);

linearResult = linearSearch(search\_list, length, target);

if (linearResult != -1) {

printf("Linear Search: Element found at index %d\n", linearResult);

} else {

printf("Linear Search: Element not found\n");

}

// Sorting the array for Binary Search

// Here we use a simple bubble sort for demonstration purposes

for ( i = 0; i < length - 1; i++) {

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

if (search\_list[j] > search\_list[j + 1]) {

int temp = search\_list[j];

search\_list[j] = search\_list[j + 1];

search\_list[j + 1] = temp;

}

}

}

printf("Sorted Array for Binary Search: ");

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

printf("%d ", search\_list[i]);

}

printf("\n");

// Binary Search

printf("Enter a value to search using Binary Search: ");

scanf("%d", &target);

binaryResult = binarySearch(search\_list, length, target);

if (binaryResult != -1) {

printf("Binary Search: Element found at index %d\n", binaryResult);

} else {

printf("Binary Search: Element not found\n");

}

return 0;

}

2. Write a program to sort list of n numbers using Bubble Sort algorithms.

#include <stdio.h>

// Function to perform Bubble Sort

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

// Outer loop for each pass

int i,j;

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

// Inner loop for comparing adjacent elements

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

// Swap if the element found is greater than the next element

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

// Function to print the array

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

int i;

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

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

}

printf("\n");

}

int main() {

int n,i;

int arr[100];

// Input number of elements

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

scanf("%d", &n);

// Input elements of the array

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

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

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

}

// Display original array

printf("Original array: ");

printArray(arr, n);

// Perform Bubble Sort

bubbleSort(arr, n);

// Display sorted array

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

3. Perform the Insertion and Selection Sort on the input {75,8,1,16,48,3,7,0} and display the

output in descending order.

#include <stdio.h>

// Function to perform Insertion Sort

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

int i,j,key;

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

key = arr[i];

j = i - 1;

// Move elements of arr[0..i-1], that are less than key,

// to one position ahead of their current position

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

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

j--;

}

arr[j + 1] = key;

}

}

// Function to perform Selection Sort

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

int i,j,maxIndex,temp;

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

// Find the maximum element in the unsorted array

maxIndex = i;

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

if (arr[j] > arr[maxIndex]) {

maxIndex = j;

}

}

// Swap the found maximum element with the first element

temp = arr[maxIndex];

arr[maxIndex] = arr[i];

arr[i] = temp;

}

}

// Function to print the array

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

int i;

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

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

}

printf("\n");

}

int main() {

int arr[] = {75, 8, 1, 16, 48, 3, 7, 0};

int arr2[] = {75, 8, 1, 16, 48, 3, 7, 0};

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

printf("Original array: ");

printArray(arr, n);

// Perform Insertion Sort

insertionSort(arr, n);

printf("Sorted array using Insertion Sort in descending order: ");

printArray(arr, n);

// Resetting the array for Selection Sort

// Perform Selection Sort

selectionSort(arr2, n);

printf("Sorted array using Selection Sort in descending order: ");

printArray(arr2, n);

return 0;

}

4. Write a program to insert the elements {61,16,8,27} into singly linked list and delete 8,61,27

from the list. Display your list after each insertion and deletion.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to insert a node at the end of the linked list

void insertEnd(struct Node\*\* head, int data) {

struct Node\* temp;

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode; // If the list is empty, set the new node as head

return;

} temp = \*head;

while (temp->next != NULL) {

temp = temp->next; // Traverse to the last node

}

temp->next = newNode; // Link the new node at the end

}

// Function to delete a node with a specific value

void deleteNode(struct Node\*\* head, int key) {

struct Node\* temp = \*head;

struct Node\* prev = NULL;

// If head node itself holds the key to be deleted

if (temp != NULL && temp->data == key) {

\*head = temp->next; // Change head

free(temp); // Free old head

return;

}

// Search for the key to be deleted, keep track of the previous node

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

// If key was not present in linked list

if (temp == NULL) return;

// Unlink the node from linked list

prev->next = temp->next;

free(temp); // Free memory

}

// Function to print the linked list

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Inserting elements into the linked list

int elements[] = {61, 16, 8, 27};

int i;

int keysToDelete[] = {8, 61, 27};

for (i = 0; i < 4; i++) {

insertEnd(&head, elements[i]);

printf("List after inserting %d: ", elements[i]);

printList(head);

}

// Deleting elements from the linked list

for ( i = 0; i < 3; i++) {

deleteNode(&head, keysToDelete[i]);

printf("List after deleting %d: ", keysToDelete[i]);

printList(head);

}

return 0;

}

5. Write a program to insert the elements {45, 34, 10, 63,3} into linear queue and delete three

elements from the list. Display your list after each insertion and deletion.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Maximum size of the queue

// Structure to represent a linear queue

struct Queue {

int items[MAX];

int front;

int rear;

};

// Function to create an empty queue

void createQueue(struct Queue\* q) {

q->front = -1;

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* q) {

return (q->rear == MAX - 1);

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

return (q->front == -1 || q->front > q->rear);

}

// Function to insert an element into the queue

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

if (isFull(q)) {

printf("Queue is full. Cannot enqueue %d\n", value);

return;

}

if (q->front == -1) {

q->front = 0; // Set front to 0 if it's the first element

}

q->rear++;

q->items[q->rear] = value; // Insert the element

}

// Function to delete an element from the queue

int dequeue(struct Queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty. Cannot dequeue\n");

return -1; // Return -1 if the queue is empty

}

item = q->items[q->front]; // Get the front item

q->front++; // Move front ahead

// Reset front and rear if the queue becomes empty

if (isEmpty(q)) {

q->front = -1;

q->rear = -1;

}

return item; // Return the dequeued item

}

// Function to print the current state of the queue

void printQueue(struct Queue\* q) {

int i;

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Queue: ");

for (i = q->front; i <= q->rear; i++) {

printf("%d ", q->items[i]);

}

printf("\n");

}

int main() {

struct Queue q;

int elements[] = {45, 34, 10, 63, 3};

int i;

createQueue(&q);

// Inserting elements into the queue

for ( i = 0; i < 5; i++) {

enqueue(&q, elements[i]);

printf("After enqueuing %d: ", elements[i]);

printQueue(&q);

}

// Deleting three elements from the queue

for ( i = 0; i < 3; i++) {

int dequeuedValue = dequeue(&q);

if (dequeuedValue != -1) { // Check if dequeue was successful

printf("After dequeuing %d: ", dequeuedValue);

printQueue(&q);

}

}

return 0;

}

6. Write a program to simulate the working of Circular queue using an array.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 5 // Define the maximum size of the circular queue

// Structure to represent the circular queue

struct CircularQueue {

int items[MAX\_SIZE];

int front;

int rear;

};

// Function to create and initialize the circular queue

void createQueue(struct CircularQueue\* q) {

q->front = -1;

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct CircularQueue\* q) {

return (q->rear + 1) % MAX\_SIZE == q->front;

}

// Function to check if the queue is empty

int isEmpty(struct CircularQueue\* q) {

return q->front == -1;

}

// Function to add an element to the circular queue

void enqueue(struct CircularQueue\* q, int value) {

if (isFull(q)) {

printf("Queue is full! Cannot enqueue %d\n", value);

return;

}

if (q->front == -1) { // If queue is empty

q->front = 0; // Set front to 0

}

q->rear = (q->rear + 1) % MAX\_SIZE; // Move rear forward in a circular manner

q->items[q->rear] = value; // Insert the new element

printf("Enqueued: %d\n", value);

}

// Function to remove an element from the circular queue

int dequeue(struct CircularQueue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty! Cannot dequeue\n");

return -1; // Return -1 if the queue is empty

}

item = q->items[q->front]; // Get the front item

if (q->front == q->rear) { // If the queue becomes empty after this dequeue

q->front = -1; // Reset front and rear

q->rear = -1;

} else {

q->front = (q->front + 1) % MAX\_SIZE; // Move front forward in a circular manner

}

printf("Dequeued: %d\n", item);

return item; // Return the dequeued item

}

// Function to display the current state of the circular queue

void displayQueue(struct CircularQueue\* q) {

int i;

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Circular Queue: ");

i = q->front;

while (1) {

printf("%d ", q->items[i]);

if (i == q->rear) break; // Stop when we reach the rear

i = (i + 1) % MAX\_SIZE; // Move to next index in a circular manner

}

printf("\n");

}

int main() {

struct CircularQueue q;

int elements[] = {45, 34, 10, 63, 3};

int i;

createQueue(&q);

// Inserting elements into the circular queue

for (i = 0; i < 5; i++) {

enqueue(&q, elements[i]);

displayQueue(&q);

}

// Deleting three elements from the circular queue

for (i = 0; i < 3; i++) {

dequeue(&q);

displayQueue(&q);

}

return 0;

}

7. Write a program to insert the elements {61,16,8,27} into ordered singly linked list and delete

8,61,27 from the list. Display your list after each insertion and deletion.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to insert a node in an ordered manner

void insertOrdered(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

struct Node\* current;

// Case for inserting at the beginning or as the first element

if (\*head == NULL || (\*head)->data >= data) {

newNode->next = \*head;

\*head = newNode;

return;

}

// Locate the node before the point of insertion

current = \*head;

while (current->next != NULL && current->next->data < data) {

current = current->next;

}

// Insert the new node

newNode->next = current->next;

current->next = newNode;

}

// Function to delete a node with a specific value

void deleteNode(struct Node\*\* head, int key) {

struct Node\* temp = \*head;

struct Node\* prev = NULL;

// If head node itself holds the key to be deleted

if (temp != NULL && temp->data == key) {

\*head = temp->next; // Change head

free(temp); // Free old head

return;

}

// Search for the key to be deleted, keep track of the previous node

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

// If key was not present in linked list

if (temp == NULL) return;

// Unlink the node from linked list

prev->next = temp->next;

free(temp); // Free memory

}

// Function to print the linked list

void printList(struct Node\* node) {

if (node == NULL) {

printf("List is empty\n");

return;

}

printf("List: ");

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Inserting elements into the ordered linked list

int elements[] = {61, 16, 8, 27};

int keysToDelete[] = {8, 61, 27};

int i;

for ( i = 0; i < 4; i++) {

insertOrdered(&head, elements[i]);

printf("After inserting %d: ", elements[i]);

printList(head);

}

// Deleting elements from the linked list

for ( i = 0; i < 3; i++) {

deleteNode(&head, keysToDelete[i]);

printf("After deleting %d: ", keysToDelete[i]);

printList(head);

}

return 0;

}

8. Write a program for Tower of Honoi problem using recursion.

#include <stdio.h>

// Function to perform the Tower of Hanoi

void towerOfHanoi(int n, char source, char target, char auxiliary) {

// Base case: if there is only one disk, move it directly from source to target

if (n == 1) {

printf("Move disk 1 from rod %c to rod %c\n", source, target);

return;

}

// Move n-1 disks from source to auxiliary, using target as a temporary rod

towerOfHanoi(n - 1, source, auxiliary, target);

// Move the nth disk from source to target

printf("Move disk %d from rod %c to rod %c\n", n, source, target);

// Move the n-1 disks from auxiliary to target, using source as a temporary rod

towerOfHanoi(n - 1, auxiliary, target, source);

}

int main() {

int n = 4; // Number of disks

printf("Solution for Tower of Hanoi with %d disks:\n", n);

towerOfHanoi(n, 'A', 'C', 'B'); // A is the source rod, C is the target rod, B is the auxiliary rod

return 0;

}

9. Write recursive program to find GCD of 3 numbers.

#include <stdio.h>

// Function to calculate GCD of two numbers using recursion

int gcd(int a, int b) {

if (b == 0) {

return a; // Base case: if b is 0, return a

}

return gcd(b, a % b); // Recursive call

}

// Function to calculate GCD of three numbers

int gcdOfThree(int a, int b, int c) {

return gcd(gcd(a, b), c); // GCD of a and b, then with c

}

int main() {

int num1, num2, num3,result;

// Taking input from the user

printf("Enter three positive integers: ");

scanf("%d %d %d", &num1, &num2, &num3);

// Calculating GCD

result = gcdOfThree(num1, num2, num3);

// Displaying the result

printf("G.C.D of %d, %d and %d is %d\n", num1, num2, num3, result);

return 0;

}

10. Write a program to demonstrate working of stack using linked list.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the stack

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to push an element onto the stack

void push(struct Node\*\* top, int data) {

struct Node\* newNode = createNode(data);

newNode->next = \*top; // Link the old top to the new node

\*top = newNode; // Update the top to point to the new node

printf("Pushed %d onto the stack.\n", data);

}

// Function to pop an element from the stack

int pop(struct Node\*\* top) {

struct Node\* temp;

int poppedValue;

if (\*top == NULL) {

printf("Stack is empty! Cannot pop.\n");

return -1; // Return -1 if stack is empty

}

temp = \*top; // Store the current top node

poppedValue = temp->data; // Get the data from the top node

\*top = (\*top)->next; // Move the top pointer to the next node

free(temp); // Free memory of the popped node

printf("Popped %d from the stack.\n", poppedValue);

return poppedValue; // Return the popped value

}

// Function to display the elements in the stack

void display(struct Node\* top) {

struct Node\* temp;

if (top == NULL) {

printf("Stack is empty.\n");

return;

}

printf("Stack elements: ");

temp = top; // Temporary pointer for traversal

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node\* stack = NULL; // Initialize an empty stack

// Perform stack operations

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

display(stack); // Display current stack

pop(&stack); // Pop an element from the stack

display(stack); // Display current stack after pop

pop(&stack); // Pop another element from the stack

display(stack); // Display current stack after pop

pop(&stack); // Pop last element from the stack

display(stack); // Display current stack after pop

pop(&stack); // Attempt to pop from an empty stack

return 0;

}

11. Write a program to convert an infix expression x^y/(5\*z)+2 to its postfix expression

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX 100 // Maximum size of the stack

// Stack structure

struct Stack {

char items[MAX];

int top;

};

// Function to create an empty stack

void initStack(struct Stack\* s) {

s->top = -1;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* s) {

return s->top == -1;

}

// Function to push an item onto the stack

void push(struct Stack\* s, char item) {

if (s->top < MAX - 1) {

s->items[++(s->top)] = item;

}

}

// Function to pop an item from the stack

char pop(struct Stack\* s) {

if (!isEmpty(s)) {

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

}

return '\0'; // Return null character if stack is empty

}

// Function to peek at the top item of the stack

char peek(struct Stack\* s) {

if (!isEmpty(s)) {

return s->items[s->top];

}

return '\0';

}

// Function to check operator precedence

int precedence(char op) {

switch (op) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

default:

return 0;

}

}

// Function to convert infix expression to postfix expression

void infixToPostfix(const char\* infix, char\* postfix) {

struct Stack s;

int i, j;

initStack(&s);

j = 0; // Postfix index

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

char token = infix[i];

// If the token is an operand, add it to the output

if (isalnum(token)) {

postfix[j++] = token; // Add operand to postfix

}

// If the token is '(', push it onto the stack

else if (token == '(') {

push(&s, token);

}

// If the token is ')', pop from stack until '(' is found

else if (token == ')') {

while (!isEmpty(&s) && peek(&s) != '(') {

postfix[j++] = pop(&s);

}

pop(&s); // Pop '(' from the stack

}

// If the token is an operator

else {

while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(token)) {

postfix[j++] = pop(&s);

}

push(&s, token); // Push current operator onto the stack

}

}

// Pop all remaining operators from the stack

while (!isEmpty(&s)) {

postfix[j++] = pop(&s);

}

postfix[j] = '\0'; // Null-terminate the postfix expression

}

int main() {

const char\* infix = "x^y/(5\*z)+2";

char postfix[MAX];

infixToPostfix(infix, postfix);

printf("Infix Expression: %s\n", infix);

printf("Postfix Expression: %s\n", postfix);

return 0;

}

12. Write a program to evaluate a postfix expression 5 3+8 2 - \*.

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 100 // Maximum size of the stack

// Stack structure

int stack[MAX];

int top = -1;

// Function to push an element onto the stack

void push(int value) {

if (top >= MAX - 1) {

printf("Stack Overflow\n");

return;

}

stack[++top] = value;

}

// Function to pop an element from the stack

int pop() {

if (top < 0) {

printf("Stack Underflow\n");

exit(1);

}

return stack[top--];

}

// Function to evaluate the postfix expression

int evaluatePostfix(char\* expression) {

int i;

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

char token = expression[i];

// If the token is a space, skip it

if (token == ' ') continue;

// If the token is a number, push it onto the stack

if (isdigit(token)) {

push(token - '0'); // Convert char to int

} else { // The token is an operator

int operand2 = pop(); // Pop top two elements

int operand1 = pop();

switch (token) {

case '+':

push(operand1 + operand2);

break;

case '-':

push(operand1 - operand2);

break;

case '\*':

push(operand1 \* operand2);

break;

case '/':

push(operand1 / operand2);

break;

default:

printf("Invalid operator: %c\n", token);

exit(1);

}

}

}

return pop(); // The result will be on top of the stack

}

int main() {

char postfix[] = "5 3 + 8 2 - \*"; // Postfix expression

int result = evaluatePostfix(postfix); // Evaluate the expression

printf("The result of the postfix expression '%s' is: %d\n", postfix, result);

return 0;

}

13. Write a program to create a binary tree with the elements {18,15,40,50,30,17,41} after creation

insert 45 and 19 into tree and delete 15,17 and 41 from tree. Display the tree on each insertion

and deletion operation.

#include<stdio.h>

#include <stdlib.h>

// Defin e the structure for a node in the binary search tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

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;

}

// Function to insert a node into the BST

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 if (data > root->data) {

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

}

return root;

}

// Function to find the minimum value node in a subtree

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

while (root && root->left != NULL) {

root = root->left;

}

return root;

}

// Function to delete a node from the BST

struct Node\* deleteNode(struct Node\* root, int data) {

struct Node\* temp;

if (root == NULL) return root;

if (data < root->data) {

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

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

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

} else {

// Node with only one child or no child

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

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

struct Node\* temp = root->left;

free(root);

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

temp = findMin(root->right);

root->data = temp->data; // Copy the inorder successor's content to this node

root->right = deleteNode(root->right, temp->data); // Delete the inorder successor

}

return root;

}

// Function for in-order traversal of the BST

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

// Main function to demonstrate insertion and deletion

int main() {

struct Node\* root = NULL;

// Initial elements to create the BST

int i;

int initialElements[] = {18, 15, 40, 50, 30, 17, 41};

int newElements[] = {45, 19};

// Delete elements from the BST

int deleteElements[] = {15, 17, 41};

// Insert initial elements into the BST

for ( i = 0; i < sizeof(initialElements)/sizeof(initialElements[0]); i++) {

root = insert(root, initialElements[i]);

}

printf("Initial BST (Inorder): ");

inorder(root);

printf("\n");

// Insert new elements into the BST

for (i = 0; i < sizeof(newElements)/sizeof(newElements[0]); i++) {

printf("Inserting %d:\n", newElements[i]);

root = insert(root, newElements[i]);

printf("BST after inserting %d (Inorder): ", newElements[i]);

inorder(root);

printf("\n");

}

for ( i = 0; i < sizeof(deleteElements)/sizeof(deleteElements[0]); i++) {

printf("Deleting %d:\n", deleteElements[i]);

root = deleteNode(root, deleteElements[i]);

printf("BST after deleting %d (Inorder): ", deleteElements[i]);

inorder(root);

printf("\n");

}

return 0;

}

14. Write a program to create binary search tree with the elements {2,5,1,3,9,0,6} and perform

inorder, preorder and post order traversal.

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the binary search tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

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;

}

// Function to insert a node into the BST

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 if (data > root->data) {

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

}

return root;

}

// In-order traversal: Left -> Root -> Right

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

// Pre-order traversal: Root -> Left -> Right

void preorder(struct Node\* root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

// Post-order traversal: Left -> Right -> Root

void postorder(struct Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

// Main function to demonstrate BST operations

int main() {

struct Node\* root = NULL;

// Insert elements into the BST

int elements[] = {2, 5, 1, 3, 9, 0, 6};

int i;

for ( i = 0; i < sizeof(elements)/sizeof(elements[0]); i++) {

root = insert(root, elements[i]);

}

// Display traversals

printf("In-order Traversal: ");

inorder(root);

printf("\n");

printf("Pre-order Traversal: ");

preorder(root);

printf("\n");

printf("Post-order Traversal: ");

postorder(root);

printf("\n");

return 0;

}

15. Write a program to Sort the following elements using heap sort {9.16,32,8,4,1,5,8,0}.

#include <stdio.h>

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

float temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted with node i which is an index in arr[]

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

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left = 2\*i + 1

int right = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

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

largest = left;

// If right child is larger than largest so far

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

largest = right;

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]); // Swap root and largest

heapify(arr, n, largest); // Recursively heapify the affected subtree

}

}

// Main function to perform heap sort

void heapSort(float arr[], int n) {

// Build heap (rearrange array)

int i;

for ( i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

for ( i = n - 1; i > 0; i--) {

swap(&arr[0], &arr[i]); // Move current root to end

heapify(arr, i, 0); // Call max heapify on the reduced heap

}

}

// Function to print the array

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

int i;

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

printf("%.2f ", arr[i]);

printf("\n");

}

int main() {

float arr[] = {9.16, 32, 8, 4, 1, 5, 8, 0};

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

printf("Original array: ");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

16. Given S1={“Flowers”} ; S2={“are beautiful”} I. Find the length of S1 II. Concatenate S1 and

S2 III. Extract the substring “low” from S1 IV. Find “are” in S2 and replace it with “is” .

#include <stdio.h>

#include <string.h>

int main() {

// Initialize the strings

char S1[] = "Flowers";

char S2[] = "are beautiful";

char result[100]; // To store concatenated string

char \*substr; // To store extracted substring

char \*pos;

// I. Find the length of S1

int length\_S1 = strlen(S1);

printf("Length of S1: %d\n", length\_S1);

// II. Concatenate S1 and S2

strcpy(result, S1); // Copy S1 to result

strcat(result, " "); // Add a space between the two strings

strcat(result, S2); // Concatenate S2 to result

printf("Concatenated string (S1 + S2): %s\n", result);

// III. Extract the substring "low" from S1

substr = strstr(S1, "low"); // Find substring "low"

if (substr) {

printf("Extracted substring: %.\*s\n", 3, substr); // Print first 3 characters of found substring

} else {

printf("Substring 'low' not found in S1.\n");

}

// IV. Find "are" in S2 and replace it with "is"

pos = strstr(S2, "are"); // Find position of "are"

if (pos) {

strncpy(pos, "is", 2); // Replace "are" with "is"

strcpy(pos + 2, pos + 4); // Shift remaining part of string left

printf("Modified string S2: %s\n", S2);

} else {

printf("\"are\" not found in S2.\n");

}

return 0;

}

17. Write a program to implement adjacency matrix of a graph.

#include <stdio.h>

#include <stdlib.h>

#define MAX 20 // Maximum number of vertices

int adj[MAX][MAX]; // Adjacency matrix

int n; // Number of vertices

// Function to create the adjacency matrix

void createGraph() {

int origin, destin, max\_edges;

int i,j;

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

scanf("%d", &n);

max\_edges = n \* (n - 1); // Maximum edges in a directed graph

// Initialize adjacency matrix to 0

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

for ( j = 0; j < n; j++) {

adj[i][j] = 0;

}

}

// Input edges

for ( i = 0; i < max\_edges; i++) {

printf("Enter edge %d (or -1 -1 to stop): ", i + 1);

scanf("%d %d", &origin, &destin);

if (origin == -1 && destin == -1) {

break; // Exit if input is -1 -1

}

if (origin >= n || destin >= n || origin < 0 || destin < 0) {

printf("Invalid edge!\n");

i--; // Decrement i to repeat this iteration

} else {

adj[origin][destin] = 1; // Add edge to adjacency matrix

}

}

}

// Function to display the adjacency matrix

void displayGraph() {

int i,j;

printf("Adjacency Matrix:\n");

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

for ( j = 0; j < n; j++) {

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

}

printf("\n");

}

}

int main() {

createGraph(); // Create the graph

displayGraph(); // Display the adjacency matrix

return 0;

}

18. Write a program to insert/retrieve an entry into hash/ from a hash table with open addressing

using linear probing

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10 // Define the size of the hash table

// Structure t o represent an entry in the hash table

typedef struct {

int key;

int isOccupied; // Flag to indicate if the slot is occupied

} HashEntry;

// Function to create a new hash table

void initializeTable(HashEntry\* table) {

int i;

for ( i = 0; i < TABLE\_SIZE; i++) {

table[i].isOccupied = 0; // Mark all slots as empty

}

}

// Hash function to calculate the index

int hashFunction(int key) {

return key % TABLE\_SIZE; // Simple modulo hash function

}

// Function to insert a key into the hash table

void insert(HashEntry\* table, int key) {

int index = hashFunction(key);

int originalIndex = index; // Store original index for probing

while (table[index].isOccupied) { // While the slot is occupied

index = (index + 1) % TABLE\_SIZE; // Linear probing

if (index == originalIndex) { // If we have looped through the table

printf("Hash table is full, cannot insert %d\n", key);

return;

}

}

table[index].key = key; // Insert the key

table[index].isOccupied = 1; // Mark slot as occupied

printf("Inserted %d at index %d\n", key, index);

}

// Function to retrieve a key from the hash table

int retrieve(HashEntry\* table, int key) {

int index = hashFunction(key);

int originalIndex = index; // Store original index for probing

while (table[index].isOccupied) { // While we find an occupied slot

if (table[index].key == key) { // Key found

return index;

}

index = (index + 1) % TABLE\_SIZE; // Linear probing

if (index == originalIndex) { // If we have looped through the table

break;

}

}

return -1; // Key not found

}

// Function to display the hash table

void displayTable(HashEntry\* table) {

int i;

printf("Hash Table:\n");

for ( i = 0; i < TABLE\_SIZE; i++) {

if (table[i].isOccupied) {

printf("Index %d: %d\n", i, table[i].key);

} else {

printf("Index %d: Empty\n", i);

}

}

}

int main() {

HashEntry\* hashTable = (HashEntry\*)malloc(TABLE\_SIZE \* sizeof(HashEntry));

int searchKey = 32;

int index;

initializeTable(hashTable); // Initialize the hash table

// Insert elements into the hash table

insert(hashTable, 9);

insert(hashTable, 16);

insert(hashTable, 32);

insert(hashTable, 8);

insert(hashTable, 4);

insert(hashTable, 1);

insert(hashTable, 5);

insert(hashTable, 8); // Duplicate entry for testing purposes

insert(hashTable, 0);

displayTable(hashTable); // Display current state of the hash table

// Retrieve elements from the hash table

index = retrieve(hashTable, searchKey);

if (index != -1) {

printf("Key %d found at index %d\n", searchKey, index);

} else {

printf("Key %d not found in the hash table\n", searchKey);

}

free(hashTable); // Free allocated memory

return 0;

}