1a

#include<stdio.h>

#include<conio.h>

#include<stdlib.h> // For exit()

#define MAX 20

void create();

void insert();

void deletion();

void search();

void display();

int b[MAX], n = 0, pos, e, p, i;

void main() {

int ch;

char g = 'y';

do {

printf("\nMain Menu:");

printf("\n1. Create \n2. Delete \n3. Search \n4. Insert \n5. Display \n6. Exit");

printf("\nEnter your choice: ");

scanf("%d", &ch);

switch (ch) {

case 1: create(); break;

case 2: deletion(); break;

case 3: search(); break;

case 4: insert(); break;

case 5: display(); break;

case 6: exit(0); // Exit the program

default: printf("\nEnter a valid choice!");

}

printf("\nDo you want to continue? (y/n): ");

scanf(" %c", &g); // Space before %c to handle trailing newline

} while (g == 'y' || g == 'Y');

getch();

}

void create() {

printf("\nEnter the number of elements (max %d): ", MAX);

scanf("%d", &n);

if (n > MAX) {

printf("\nExceeded maximum size! Resetting to %d.\n", MAX);

n = MAX;

}

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

printf("Enter element %d: ", i + 1);

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

}

}

void deletion() {

printf("\nEnter the position to delete (1 to %d): ", n);

scanf("%d", &pos);

if (pos < 1 || pos > n) {

printf("\nInvalid position!");

return;

}

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

b[i] = b[i + 1];

}

n--;

printf("\nThe list after deletion:");

display();

}

void search() {

int found = 0;

printf("\nEnter the element to search: ");

scanf("%d", &e);

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

if (b[i] == e) {

printf("\nElement %d found at position %d.", e, i + 1);

found = 1;

break;

}

}

if (!found) {

printf("\nElement %d not found in the list.", e);

}

}

void insert() {

if (n >= MAX) {

printf("\nList is full! Cannot insert more elements.");

return;

}

printf("\nEnter the position to insert (1 to %d): ", n + 1);

scanf("%d", &pos);

if (pos < 1 || pos > n + 1) {

printf("\nInvalid position!");

return;

}

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

scanf("%d", &p);

for (i = n; i >= pos; i--) {

b[i] = b[i - 1];

}

b[pos - 1] = p;

n++;

printf("\nThe list after insertion:");

display();

}

void display() {

if (n == 0) {

printf("\nThe list is empty!");

return;

}

printf("\nThe elements in the list are:");

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

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

}

}

o/p

Array of 10 numbers:

Input elements:

12456789

10

11

Current array: 1 2 4 5 6 7 8 9 10 11

Would you like to enter another element? [1/0]: 1

Input element: 3

Input position: 2

Current array: 1 2 3 4 5 6 7 8 9 10

1b

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void create();

void display();

void insert();

void find();

void delete();

struct node {

int data;

struct node \*next;

};

typedef struct node \*position;

position L = NULL, p, newnode;

void main() {

int choice;

clrscr();

do {

printf("1. Create\n2. Display\n3. Insert\n4. Find\n5. Delete\n6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: create(); break;

case 2: display(); break;

case 3: insert(); break;

case 4: find(); break;

case 5: delete(); break;

case 6: exit(0);

default: printf("Invalid choice. Please try again.\n");

}

} while (choice != 6);

getch();

}

void create() {

int i, n, value;

L = NULL; // Initialize list as empty

printf("\nEnter the number of nodes to be inserted: ");

scanf("%d", &n);

if (n <= 0) {

printf("Invalid number of nodes.\n");

return;

}

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

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

if (!newnode) {

printf("Memory allocation failed.\n");

return;

}

printf("Enter data for node %d: ", i);

scanf("%d", &value);

newnode->data = value;

newnode->next = NULL;

if (L == NULL) {

L = newnode;

p = L;

} else {

p->next = newnode;

p = newnode;

}

}

}

void display() {

if (L == NULL) {

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

return;

}

p = L;

printf("List: ");

while (p != NULL) {

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

p = p->next;

}

printf("NULL\n");

}

void insert() {

int ch, value, pos, i;

printf("\nEnter your choice:\n1. At Beginning\n2. At Middle\n3. At End\n");

scanf("%d", &ch);

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

if (!newnode) {

printf("Memory allocation failed.\n");

return;

}

switch (ch) {

case 1: // Insert at beginning

printf("Enter the data to insert: ");

scanf("%d", &value);

newnode->data = value;

newnode->next = L;

L = newnode;

break;

case 2: // Insert in the middle

printf("Enter the data to insert: ");

scanf("%d", &value);

printf("Enter the position to insert: ");

scanf("%d", &pos);

if (pos <= 0) {

printf("Invalid position.\n");

free(newnode);

return;

}

newnode->data = value;

newnode->next = NULL;

p = L;

for (i = 1; i < pos - 1 && p != NULL; i++) {

p = p->next;

}

if (p == NULL) {

printf("Position out of bounds.\n");

free(newnode);

} else {

newnode->next = p->next;

p->next = newnode;

}

break;

case 3: // Insert at end

printf("Enter the data to insert: ");

scanf("%d", &value);

newnode->data = value;

newnode->next = NULL;

if (L == NULL) {

L = newnode;

} else {

p = L;

while (p->next != NULL) {

p = p->next;

}

p->next = newnode;

}

break;

default:

printf("Invalid choice.\n");

free(newnode);

}

}

void find() {

int search, found = 0;

printf("\nEnter the element to find: ");

scanf("%d", &search);

p = L;

while (p != NULL) {

if (p->data == search) {

found = 1;

break;

}

p = p->next;

}

if (found)

printf("Element %d is present in the list.\n", search);

else

printf("Element %d is not found in the list.\n", search);

}

void delete() {

int x;

position temp;

if (L == NULL) {

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

return;

}

printf("\nEnter the data to delete: ");

scanf("%d", &x);

if (L->data == x) { // Deleting the first node

temp = L;

L = L->next;

free(temp);

printf("Element %d deleted.\n", x);

display();

return;

}

p = L;

while (p->next != NULL && p->next->data != x) {

p = p->next;

}

if (p->next == NULL) {

printf("Element %d not found.\n", x);

} else {

temp = p->next;

p->next = p->next->next;

free(temp);

printf("Element %d deleted.\n", x);

}

display();

}

o/p

1.create

2.display

3.insert

4.find

5.delete

Enter your choice

1

Enter the number of nodes to be inserted

5

Enter the data

12345

1.create

2.display

3.insert

4.find

5.delete

Enter your choice 2

1 -> 2 -> 3 -> 4 -> 5 ->

Null

---------------------------------------------------

1.create

2.display

3.insert

4.find

5.delete

Enter your choice 3

Enter ur choice

1.first

2.middle

3.end

1

Enter the data to be inserted

77

-> 1 -> 2 -> 3 -> 4 ->5-.

Null

1c

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

// Function Prototypes

void insert();

void deletion();

void display();

void find();

typedef struct node \*position;

struct node {

int data;

position next;

position prev;

};

position L = NULL, P, temp, newnode;

void main() {

int choice;

clrscr();

do {

printf("\n1. INSERT");

printf("\n2. DELETE");

printf("\n3. DISPLAY");

printf("\n4. FIND");

printf("\n5. EXIT");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: insert(); break;

case 2: deletion(); break;

case 3: display(); break;

case 4: find(); break;

case 5: exit(0);

default: printf("\nInvalid choice! Try again.\n");

}

} while (choice != 5);

getch();

}

// Function to Insert a Node

void insert() {

int pos, i;

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

if (!newnode) {

printf("\nMemory allocation failed!");

return;

}

printf("\nEnter the data to be inserted: ");

scanf("%d", &newnode->data);

if (L == NULL) {

newnode->next = NULL;

newnode->prev = NULL;

L = newnode;

} else {

printf("\nEnter the position where the data is to be inserted: ");

scanf("%d", &pos);

if (pos == 1) {

newnode->next = L;

newnode->prev = NULL;

L->prev = newnode;

L = newnode;

} else {

P = L;

for (i = 1; i < pos - 1 && P->next != NULL; i++) {

P = P->next;

}

newnode->next = P->next;

if (P->next != NULL) {

P->next->prev = newnode;

}

newnode->prev = P;

P->next = newnode;

}

}

printf("\nNode inserted successfully.\n");

}

// Function to Delete a Node

void deletion() {

int pos, i;

if (L == NULL) {

printf("\nThe list is empty!");

return;

}

printf("\nEnter the position of the data to be deleted: ");

scanf("%d", &pos);

if (pos == 1) {

temp = L;

L = L->next;

if (L != NULL) {

L->prev = NULL;

}

printf("\nThe deleted element is %d", temp->data);

free(temp);

} else {

P = L;

for (i = 1; i < pos - 1 && P->next != NULL; i++) {

P = P->next;

}

if (P->next == NULL) {

printf("\nInvalid position!");

return;

}

temp = P->next;

P->next = temp->next;

if (temp->next != NULL) {

temp->next->prev = P;

}

printf("\nThe deleted element is %d", temp->data);

free(temp);

}

}

// Function to Display the List

void display() {

if (L == NULL) {

printf("\nNo elements in the list!");

return;

}

printf("\nThe elements in the list are:\n");

for (P = L; P != NULL; P = P->next) {

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

}

printf("\n");

}

// Function to Find an Element

void find() {

int a, flag = 0, count = 0;

if (L == NULL) {

printf("\nThe list is empty!");

return;

}

printf("\nEnter the element to be searched: ");

scanf("%d", &a);

for (P = L; P != NULL; P = P->next) {

count++;

if (P->data == a) {

flag = 1;

printf("\nThe element is found at position %d", count);

break;

}

}

if (!flag) {

printf("\nThe element is not found in the list.");

}

}

o/p

Enter the data to be inserted10

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter your option 1

Enter the data to be inserted 20

Enter the position where the data is to be inserted 2

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 1

Enter the data to be inserted 30

Enter the position where the data is to be inserted 3

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 3

The elements in the list are

10 20 30

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 2

Enter the position of the data to be deleted 2

The deleted element is 20

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 3

The elements in the list are

10 30

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 4

Enter the elements to be searched 20

The element is not found

1. INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option 4

Enter the elements to be searched 30

The element is found The position is 2 1.INSERT

2. DELETE

3. DISPLAY

4. FIND

5. EXIT

Enter ur option5

Press any key to continue …

Enter the elements to be searched 30

The element is found The position is 2

2a

#include <stdio.h>

#include <conio.h>

#define MAX 5 // Define maximum stack size

static int stack[MAX];

int top = -1; // Initialize stack pointer

// Push function to insert an element onto the stack

void push(int x) {

stack[++top] = x;

}

// Pop function to remove and return the top element of the stack

int pop() {

return stack[top--];

}

// View function to display all elements in the stack

void view() {

int i;

if (top < 0) {

printf("\nStack Empty\n");

} else {

printf("\nStack Elements:\n");

printf("Top --> ");

for (i = top; i >= 0; i--) {

printf("%4d", stack[i]);

}

printf("\n");

}

}

// Main function for stack operations

int main() {

int ch = 0, val;

clrscr(); // Clear the screen

while (ch != 4) {

printf("\nSTACK OPERATIONS\n");

printf("1. PUSH\n");

printf("2. POP\n");

printf("3. VIEW\n");

printf("4. QUIT\n");

printf("Enter your choice: ");

scanf("%d", &ch);

switch (ch) {

case 1: // PUSH operation

if (top < MAX - 1) {

printf("\nEnter Stack Element: ");

scanf("%d", &val);

push(val);

} else {

printf("\nStack Overflow\n");

}

break;

case 2: // POP operation

if (top < 0) {

printf("\nStack Underflow\n");

} else {

val = pop();

printf("\nPopped Element: %d\n", val);

}

break;

case 3: // VIEW operation

view();

break;

case 4: // Exit

printf("\nExiting...\n");

break;

default: // Invalid choice

printf("\nInvalid Choice! Please try again.\n");

}

}

getch(); // Wait for a key press before exiting

return 0;

}

o/p

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 1

Enter Stack element : 12

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 1

Enter Stack element : 23

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 1

Enter Stack element : 34

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 1

Enter Stack element : 45

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 3

Top--> 45 34 23 12

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 2

Popped element is 45

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 3

Top--> 34 23 12

STACK OPERATION

1.PUSH 2.POP 3.VIEW 4.QUIT

Enter Choice : 4

2b

#include <stdio.h>

#include <conio.h>

#define MAX 5 // Define maximum queue size

static int queue[MAX];

int front = -1; // Initialize front pointer

int rear = -1; // Initialize rear pointer

// Insert function to add an element to the queue

void insert(int x) {

if (rear == MAX - 1) {

printf("\nQueue Overflow\n");

return;

}

queue[++rear] = x;

if (front == -1) {

front = 0; // Initialize front when the first element is inserted

}

}

// Remove function to delete an element from the queue

int remove() {

int val = queue[front];

if (front == rear) {

// Reset queue if it's now empty

front = rear = -1;

} else {

front++;

}

return val;

}

// View function to display the queue elements

void view() {

int i;

if (front == -1) {

printf("\nQueue Empty\n");

} else {

printf("\nQueue Elements:\nFront --> ");

for (i = front; i <= rear; i++) {

printf("%4d", queue[i]);

}

printf(" <-- Rear\n");

}

}

// Main function for queue operations

int main() {

int ch = 0, val;

clrscr(); // Clear the screen

while (ch != 4) {

printf("\nQUEUE OPERATIONS\n");

printf("1. INSERT\n");

printf("2. DELETE\n");

printf("3. VIEW\n");

printf("4. QUIT\n");

printf("Enter your choice: ");

scanf("%d", &ch);

switch (ch) {

case 1: // INSERT operation

if (rear < MAX - 1) {

printf("\nEnter element to be inserted: ");

scanf("%d", &val);

insert(val);

} else {

printf("\nQueue Full\n");

}

break;

case 2: // DELETE operation

if (front == -1) {

printf("\nQueue Empty\n");

} else {

val = remove();

printf("\nElement deleted: %d\n", val);

}

break;

case 3: // VIEW operation

view();

break;

case 4: // Exit

printf("\nExiting...\n");

break;

default: // Invalid choice

printf("\nInvalid Choice! Please try again.\n");

}

}

getch(); // Wait for a key press before exiting

return 0;

}

o/p

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Enter element to be inserted : 12

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Enter element to be inserted : 23

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Enter element to be inserted : 34

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Enter element to be inserted : 45

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Enter element to be inserted : 56

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 1

Queue Full

QUEUE OPERATION

1.INSERT 2.DELETE 3.VIEW 4.QUIT

Enter Choice : 3

Front--> 12 23 34 45 56 <--Rear

3

#include <stdio.h>

#include <conio.h>

#include <string.h>

#define MAX 20

int top = -1;

char stack[MAX];

// Function prototypes

void push(char item);

char pop();

int prcd(char symbol);

int isoperator(char symbol);

void convertip(char infix[], char postfix[]);

int prcd(char symbol) {

switch (symbol) {

case '+':

case '-':

return 2;

case '\*':

case '/':

return 4;

case '^':

case '$':

return 6;

case '(':

case ')':

case '#':

return 1;

default:

return 0;

}

}

int isoperator(char symbol) {

switch (symbol) {

case '+':

case '-':

case '\*':

case '/':

case '^':

case '$':

case '(':

case ')':

return 1;

default:

return 0;

}

}

void convertip(char infix[], char postfix[]) {

int i, j = 0;

char symbol;

stack[++top] = '#'; // Push sentinel onto stack

for (i = 0; i < strlen(infix); i++) {

symbol = infix[i];

if (isoperator(symbol) == 0) {

postfix[j++] = symbol; // Append operands to postfix expression

} else {

if (symbol == '(') {

push(symbol);

} else if (symbol == ')') {

while (stack[top] != '(') {

postfix[j++] = pop();

}

pop(); // Pop out '('

} else {

while (prcd(symbol) <= prcd(stack[top])) {

postfix[j++] = pop();

}

push(symbol);

}

}

}

// Pop remaining operators in the stack

while (stack[top] != '#') {

postfix[j++] = pop();

}

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

}

void push(char item) {

if (top >= MAX - 1) {

printf("\nStack Overflow\n");

return;

}

stack[++top] = item;

}

char pop() {

if (top < 0) {

printf("\nStack Underflow\n");

return '#';

}

return stack[top--];

}

int main() {

char infix[20], postfix[20];

clrscr(); // Clear screen for Turbo C compatibility

printf("Enter a valid infix expression: ");

fgets(infix, sizeof(infix), stdin);

infix[strcspn(infix, "\n")] = '\0'; // Remove trailing newline from fgets

convertip(infix, postfix);

printf("\nThe corresponding postfix expression is: ");

puts(postfix);

getch(); // Wait for keypress before exiting

return 0;

}

o/p

Enter the valid infix string: (a+b\*c)/(d$e)

The corresponding postfix string is: abc\*+de$/

Enter the valid infix string: a\*b+c\*d/e

The corresponding postfix string is: ab\*cd\*e/+

Enter the valid infix string: a+b\*c+(d\*e+f)\*g

The corresponding postfix string is: abc\*+de\*f+g\*+

4

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

// Define the structure for a binary search tree node

struct searchtree {

int element;

struct searchtree \*left, \*right;

} \*root;

typedef struct searchtree \*node;

typedef int ElementType;

// Function prototypes

node insert(ElementType, node);

node delete(ElementType, node);

void makeempty();

node findmin(node);

node findmax(node);

node find(ElementType, node);

void display(node, int);

void main() {

int ch;

ElementType a;

node temp;

makeempty(); // Initialize the tree as empty

while (1) {

printf("\n1. Insert\n2. Delete\n3. Find\n4. Find min\n5. Find max\n6. Display\n7. Exit\nEnter Your Choice: ");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter an element: ");

scanf("%d", &a);

root = insert(a, root);

break;

case 2:

printf("\nEnter the element to delete: ");

scanf("%d", &a);

root = delete(a, root);

break;

case 3:

printf("\nEnter the element to search: ");

scanf("%d", &a);

temp = find(a, root);

if (temp != NULL)

printf("Element found");

else

printf("Element not found");

break;

case 4:

temp = findmin(root);

if (temp == NULL)

printf("\nEmpty tree");

else

printf("\nMinimum element: %d", temp->element);

break;

case 5:

temp = findmax(root);

if (temp == NULL)

printf("\nEmpty tree");

else

printf("\nMaximum element: %d", temp->element);

break;

case 6:

if (root == NULL)

printf("\nEmpty tree");

else

display(root, 1);

break;

case 7:

exit(0);

default:

printf("Invalid Choice");

}

}

}

node insert(ElementType x, node t) {

if (t == NULL) {

t = (node)malloc(sizeof(struct searchtree));

t->element = x;

t->left = t->right = NULL;

} else {

if (x < t->element)

t->left = insert(x, t->left);

else if (x > t->element)

t->right = insert(x, t->right);

}

return t;

}

node delete(ElementType x, node t) {

node temp;

if (t == NULL)

printf("\nElement not found");

else {

if (x < t->element)

t->left = delete(x, t->left);

else if (x > t->element)

t->right = delete(x, t->right);

else {

if (t->left && t->right) {

temp = findmin(t->right);

t->element = temp->element;

t->right = delete(t->element, t->right);

} else {

temp = t;

if (t->left == NULL)

t = t->right;

else

t = t->left;

free(temp);

}

}

}

return t;

}

void makeempty() {

root = NULL;

}

node findmin(node temp) {

if (temp == NULL)

return NULL;

while (temp->left != NULL)

temp = temp->left;

return temp;

}

node findmax(node temp) {

if (temp == NULL)

return NULL;

while (temp->right != NULL)

temp = temp->right;

return temp;

}

node find(ElementType x, node t) {

if (t == NULL)

return NULL;

if (x < t->element)

return find(x, t->left);

if (x > t->element)

return find(x, t->right);

return t;

}

void display(node t, int level) {

int i;

if (t != NULL) {

display(t->right, level + 1);

printf("\n");

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

printf(" ");

printf("%d", t->element);

display(t->left, level + 1);

}

}

o/p

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 1

Enter an element : 10

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 1

Enter an element : 20

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 1

Enter an element : 5

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 4

The smallest Number is 5

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 3

Enter an element : 100

Element not Found

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 2

Enter an element : 20

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 6

20

10

1. Insert

2. Delete

3. Find

4. Find Min

5. Find Max

6. Display

7. Exit

Enter your Choice : 7

5

#include <stdio.h>

#include <stdlib.h>

// Define a binary tree node structure

struct node {

int item;

struct node\* left;

struct node\* right;

};

// Inorder traversal: Left -> Root -> Right

void inorderTraversal(struct node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

// Preorder traversal: Root -> Left -> Right

void preorderTraversal(struct node\* root) {

if (root == NULL) return;

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// Postorder traversal: Left -> Right -> Root

void postorderTraversal(struct node\* root) {

if (root == NULL) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

// Create a new node

struct node\* createNode(int value) {

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

newNode->item = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Insert a node on the left of the given root

struct node\* insertLeft(struct node\* root, int value) {

root->left = createNode(value);

return root->left;

}

// Insert a node on the right of the given root

struct node\* insertRight(struct node\* root, int value) {

root->right = createNode(value);

return root->right;

}

int main() {

// Create the root node

struct node\* root = createNode(1);

// Build the binary tree

insertLeft(root, 12);

insertRight(root, 9);

insertLeft(root->left, 5);

insertRight(root->left, 6);

// Perform tree traversals

printf("Inorder traversal: \n");

inorderTraversal(root);

printf("\n");

printf("Preorder traversal: \n");

preorderTraversal(root);

printf("\n");

printf("Postorder traversal: \n");

postorderTraversal(root);

printf("\n");

return 0;

}

o/p

Inorder:

5->12->6->1->9->

Preorder:

1->12->5->6->9->

Postorder:

5->6->12->9->1->

6

#include <stdio.h>

#include <stdlib.h>

int tree\_array\_size = 20; // Maximum size of the heap

int heap\_size = 0; // Current size of the heap

const int INF = 100000; // Large value for initializing keys

// Function to swap two integers

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

int t = \*a;

\*a = \*b;

\*b = t;

}

// Function to get the right child of a node

int get\_right\_child(int index) {

if (((2 \* index) + 1 < tree\_array\_size) && (index >= 1))

return (2 \* index) + 1;

return -1;

}

// Function to get the left child of a node

int get\_left\_child(int index) {

if ((2 \* index < tree\_array\_size) && (index >= 1))

return 2 \* index;

return -1;

}

// Function to get the parent of a node

int get\_parent(int index) {

if ((index > 1) && (index < tree\_array\_size))

return index / 2;

return -1;

}

// Function to maintain the max-heap property

void max\_heapify(int A[], int index) {

int left\_child\_index = get\_left\_child(index);

int right\_child\_index = get\_right\_child(index);

int largest = index;

// Find the largest among index, left child, and right child

if ((left\_child\_index <= heap\_size) && (left\_child\_index > 0)) {

if (A[left\_child\_index] > A[largest]) {

largest = left\_child\_index;

}

}

if ((right\_child\_index <= heap\_size) && (right\_child\_index > 0)) {

if (A[right\_child\_index] > A[largest]) {

largest = right\_child\_index;

}

}

// Swap and continue heapifying if root is not the largest

if (largest != index) {

swap(&A[index], &A[largest]);

max\_heapify(A, largest);

}

}

// Function to build a max heap

void build\_max\_heap(int A[]) {

for (int i = heap\_size / 2; i >= 1; i--) {

max\_heapify(A, i);

}

}

// Function to get the maximum element

int maximum(int A[]) {

if (heap\_size >= 1)

return A[1];

return -INF; // Return a very small value if heap is empty

}

// Function to extract the maximum element

int extract\_max(int A[]) {

if (heap\_size < 1) {

printf("Heap underflow\n");

return -INF;

}

int maxm = A[1];

A[1] = A[heap\_size];

heap\_size--;

max\_heapify(A, 1);

return maxm;

}

// Function to increase the key of a node

void increase\_key(int A[], int index, int key) {

if (key < A[index]) {

printf("New key is smaller than current key\n");

return;

}

A[index] = key;

while ((index > 1) && (A[get\_parent(index)] < A[index])) {

swap(&A[index], &A[get\_parent(index)]);

index = get\_parent(index);

}

}

// Function to decrease the key of a node

void decrease\_key(int A[], int index, int key) {

if (key > A[index]) {

printf("New key is larger than current key\n");

return;

}

A[index] = key;

max\_heapify(A, index);

}

// Function to insert a new key into the heap

void insert(int A[], int key) {

heap\_size++;

if (heap\_size >= tree\_array\_size) {

printf("Heap overflow\n");

heap\_size--;

return;

}

A[heap\_size] = -INF; // Assign a very small value initially

increase\_key(A, heap\_size, key);

}

// Function to print the heap

void print\_heap(int A[]) {

for (int i = 1; i <= heap\_size; i++) {

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

}

printf("\n");

}

// Main function

int main() {

int A[tree\_array\_size]; // Heap array

// Insert elements into the heap

insert(A, 20);

insert(A, 15);

insert(A, 8);

insert(A, 10);

insert(A, 5);

insert(A, 7);

insert(A, 6);

insert(A, 2);

insert(A, 9);

insert(A, 1);

// Print the heap

printf("Initial Heap:\n");

print\_heap(A);

// Increase key at index 5

increase\_key(A, 5, 22);

printf("After increasing key at index 5 to 22:\n");

print\_heap(A);

// Decrease key at index 1

decrease\_key(A, 1, 13);

printf("After decreasing key at index 1 to 13:\n");

print\_heap(A);

// Perform maximum and extract-max operations

printf("Maximum element: %d\n", maximum(A));

printf("Extracting maximum element: %d\n", extract\_max(A));

print\_heap(A);

// Extract remaining elements

while (heap\_size > 0) {

printf("Extracting: %d\n", extract\_max(A));

}

return 0;

}

o/p

Priority Queue using Heap:

Enter the number of elements: 7

Enter elements: 234

6543

12234

64

53

6

876

77

PQueue: 64 77 234 536 876 6543 12234

7a

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#define V 9 // Number of vertices in the graph

// Function to find the vertex with the minimum distance that hasn't been processed yet

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

int min = INT\_MAX, min\_index;

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

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

min = dist[v], min\_index = v;

}

}

return min\_index;

}

// Function to print the solution

void printSolution(int dist[]) {

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

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

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

}

}

// Dijkstra's algorithm implementation to find the shortest path from the source

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

int dist[V]; // Array to store the shortest distance from source to each vertex

bool sptSet[V]; // Boolean array to track the vertices included in the shortest path tree

// Initialize all distances as INFINITE and sptSet as false

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

dist[i] = INT\_MAX;

sptSet[i] = false;

}

dist[src] = 0; // Distance to the source vertex is always 0

// Find the shortest path for all vertices

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

// Select the minimum distance vertex from the set of vertices not yet processed

int u = minDistance(dist, sptSet);

// Mark the selected vertex as processed

sptSet[u] = true;

// Update dist[] values of the adjacent vertices of the selected vertex

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

// Update dist[v] if the current vertex u can be a better path to 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];

}

}

}

// Print the final distances

printSolution(dist);

}

int main() {

// Example graph represented as an adjacency matrix

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 }

};

// Run Dijkstra's algorithm starting from vertex 0

dijkstra(graph, 0);

return 0;

}

o/p

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

7b

#include <limits.h>

#include <stdbool.h>

#include <stdio.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] == false && key[v] < min) {

min = key[v], min\_index = v;

}

}

return min\_index;

}

int 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];

bool mstSet[V];

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] == false && 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;

}

o/p

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

7c

#include <stdio.h>

#include <stdlib.h>

// Comparator function to sort edges by weight

int comparator(const void\* p1, const void\* p2)

{

const int (\*x)[3] = p1;

const int (\*y)[3] = p2;

return (\*x)[2] - (\*y)[2];

}

// Initialize the parent and rank arrays for union-find

void makeSet(int parent[], int rank[], int n)

{

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

parent[i] = i;

rank[i] = 0;

}

}

// Find the representative of a component (with path compression)

int findParent(int parent[], int component)

{

if (parent[component] == component)

return component;

return parent[component] = findParent(parent, parent[component]);

}

// Union the two components (with union by rank)

void unionSet(int u, int v, int parent[], int rank[], int n)

{

u = findParent(parent, u);

v = findParent(parent, v);

if (rank[u] < rank[v]) {

parent[u] = v;

} else if (rank[u] > rank[v]) {

parent[v] = u;

} else {

parent[v] = u;

rank[u]++;

}

}

// Kruskal's Algorithm to find the MST

void kruskalAlgo(int n, int edge[n][3])

{

// Sort the edges by weight

qsort(edge, n, sizeof(edge[0]), comparator);

int parent[n];

int rank[n];

makeSet(parent, rank, n);

int minCost = 0;

printf("Following are the edges in the constructed MST\n");

// Process each edge in sorted order

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

{

int v1 = findParent(parent, edge[i][0]);

int v2 = findParent(parent, edge[i][1]);

int wt = edge[i][2];

// If v1 and v2 are not in the same component, add the edge to the MST

if (v1 != v2)

{

unionSet(v1, v2, parent, rank, n);

minCost += wt;

printf("%d -- %d == %d\n", edge[i][0], edge[i][1], wt);

}

}

printf("Minimum Cost Spanning Tree: %d\n", minCost);

}

int main()

{

// Define edges in the graph: {vertex1, vertex2, weight}

int edge[5][3] = { { 0, 1, 10 },

{ 0, 2, 6 },

{ 0, 3, 5 },

{ 1, 3, 15 },

{ 2, 3, 4 } };

// Call Kruskal's Algorithm

kruskalAlgo(5, edge);

return 0;

}

o/p

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

8a

/\* Linear search on a sorted array \*/

#include <stdio.h>

#include <conio.h>

main()

{

int a[50],i, n, val, found;

clrscr();

printf("Enter number of elements : ");

scanf("%d", &n);

printf("Enter Array Elements : \n");

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

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

printf("Enter element to locate : ");

scanf("%d", &val);

found = 0;

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

{

if (a[i] == val)

{

printf("Element found at position %d", i);

found = 1;

break;

}}

if (found == 0)

printf("\n Element not found");

getch();

}

o/p

Enter number of elements : 7

Enter Array Elements :

23 6 12 5 0 32 10

Enter element to locate : 5

Element found at position 3

8b

/\* Binary Search on a sorted array \*/

#include <stdio.h>

void main()

{

int a[50],i, n, upper, lower, mid, val, found, att=0;

printf("Enter array size : ");

scanf("%d", &n);

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

a[i] = 2 \* i;

printf("\n Elements in Sorted Order \n");

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

printf("%4d", a[i]);

printf("\n Enter element to locate : ");

scanf("%d", &val);

upper = n;

lower = 0;

found = -1;

while (lower <= upper)

{

mid = (upper + lower)/2;

att++;

if (a[mid] == val)

{

Printf(“found at index %d in %d attempts”,mid,att);

found = 1;

break;

}

else if(a[mid] > val)

upper = mid - 1;

else

lower = mid + 1;

}

if (found == -1)

printf("Element not found");

}

o/p

Enter array size : 10

Elements in Sorted Order

0 2 4 6 8 10 12 14 16 18

Enter element to locate : 16

Found at index 8 in 2 attempts

9a

#include <stdio.h>

void main()

{

int i, j, k, n, temp, a[20], p=0;

printf("Enter total elements: ");

scanf("%d",&n);

printf("Enter array elements: ");

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

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

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

{

temp = a[i];

j = i - 1;

while((temp<a[j]) && (j>=0))

{

a[j+1] = a[j];

j = j - 1;

}

a[j+1] = temp;

p++;

printf("\n After Pass %d: ", p);

for(k=0; k<n; k++)

printf(" %d", a[k]);

}

printf("\n Sorted List : ");

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

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

}

o/p

Enter total elements: 6

Enter array elements: 34 8 64 51 32 21

After Pass 1: 8 34 64 51 32 21

After Pass 2: 8 34 64 51 32 21

After Pass 3: 8 34 51 64 32 21

After Pass 4: 8 32 34 51 64 21

After Pass 5: 8 21 32 34 51 64

Sorted List : 8 21 32 34 51 64

9b

#include <stdio.h>

void main()

{

int a[100], n, i, j, position, temp;

// Prompt user for number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

// Prompt user for the elements of the array

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

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

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

// Selection sort algorithm

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

{

position = i;

// Find the smallest element in the remaining unsorted part

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

{

if (a[position] > a[j])

position = j;

}

// Swap the found minimum element with the first element

if (position != i)

{

temp = a[i];

a[i] = a[position];

a[position] = temp;

}

}

// Print the sorted list

printf("Sorted list in ascending order:\n");

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

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

}

o/p

Enter number of elements 5

Enter 5 integers

8 3 9 5 1

Sorted list in ascending order:

13589

9c

#include<stdio.h>

void qsort(int arr[], int fst, int last);

int main()

{

int arr[30];

int i, size;

// Input size and elements

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

scanf("%d", &size);

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

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

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

// Quick sort call

qsort(arr, 0, size - 1);

// Display sorted array

printf("\nQuick sorted elements:\n");

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

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

return 0;

}

// Quick Sort function

void qsort(int arr[], int fst, int last)

{

int i, j, pivot, tmp;

if(fst < last) {

// Setting pivot as the first element

pivot = fst;

i = fst;

j = last;

// Partitioning

while(i < j) {

while(arr[i] <= arr[pivot] && i < last)

i++;

while(arr[j] > arr[pivot])

j--;

// Swap elements

if(i < j) {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}

}

// Swap pivot element with j

tmp = arr[pivot];

arr[pivot] = arr[j];

arr[j] = tmp;

// Recursively sort the subarrays

qsort(arr, fst, j - 1);

qsort(arr, j + 1, last);

}

}

o/p

Enter total no. of the elements : 8

Enter total 8 elements :

127

-1

04

-2

3

Quick sorted elements

-2 -1 0 1 2 3 4 7

9d

#include <stdio.h>

void merge(int arr[], int min, int mid, int max);

void part(int arr[], int min, int max);

int size;

int main()

{

int i, arr[30];

// Input size and array elements

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

scanf("%d", &size);

printf("Enter array elements: ");

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

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

// Call the merge sort function

part(arr, 0, size - 1);

// Output the sorted array

printf("\nMerge sorted list: ");

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

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

return 0;

}

// Function to split the array

void part(int arr[], int min, int max)

{

int mid;

if (min < max) {

mid = (min + max) / 2;

// Recursively split the array

part(arr, min, mid);

part(arr, mid + 1, max);

// Merge the two halves

merge(arr, min, mid, max);

// Print the half-sorted list

if (max - min == (size / 2) - 1) {

printf("\nHalf sorted list: ");

for (int i = min; i <= max; i++)

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

}

}

}

// Function to merge two halves of the array

void merge(int arr[], int min, int mid, int max)

{

int tmp[30]; // Temporary array for merged values

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

// Merging two sorted subarrays

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

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

tmp[k] = arr[i];

i++;

} else {

tmp[k] = arr[j];

j++;

}

k++;

}

// Copy remaining elements of left subarray

while (i <= mid) {

tmp[k] = arr[i];

i++;

k++;

}

// Copy remaining elements of right subarray

while (j <= max) {

tmp[k] = arr[j];

j++;

k++;

}

// Copy merged subarray into original array

for (i = min; i <= max; i++)

arr[i] = tmp[i];

}

o/p

Enter total no. of elements : 8

Enter array elements : 24 13 26 1 2 27 38 15

Half sorted list : 1 13 24 26

Half sorted list : 2 15 27 38

Merge sorted list : 1 2 13 15 24 26 27 38

10

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

int h[TABLE\_SIZE] = {-1}; // Using -1 to represent empty slots instead of NULL

void insert()

{

int key, index, i, hkey;

printf("\nEnter a value to insert into hash table:\n");

scanf("%d", &key);

hkey = key % TABLE\_SIZE;

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

{

index = (hkey + i) % TABLE\_SIZE;

if (h[index] == -1) // Check for empty slot

{

h[index] = key;

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

return;

}

}

printf("\nElement cannot be inserted, hash table is full.\n");

}

void search()

{

int key, index, i, hkey;

printf("\nEnter search element:\n");

scanf("%d", &key);

hkey = key % TABLE\_SIZE;

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

{

index = (hkey + i) % TABLE\_SIZE;

if (h[index] == key) // Key found

{

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

return;

}

else if (h[index] == -1) // Empty slot, stop searching

{

break;

}

}

printf("\nValue %d not found.\n", key);

}

void display()

{

int i;

printf("\nElements in the hash table are:\n");

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

{

if (h[i] != -1) // Only display slots that contain valid data

printf("At index %d\t value = %d\n", i, h[i]);

}

}

int main()

{

int opt;

while(1)

{

printf("\nPress 1. Insert\t 2. Display\t 3. Search\t 4. Exit\n");

scanf("%d", &opt);

switch(opt)

{

case 1:

insert();

break;

case 2:

display();

break;

case 3:

search();

break;

case 4:

exit(0);

default:

printf("\nInvalid choice. Please try again.\n");

}

}

return 0;

}

o/p

Press 1. Insert 2. Display 3. Search 4.Exit

1

enter a value to insert into hash table

12

Press 1. Insert 2. Display 3. Search 4.Exit

1

enter a value to insert into hash table

13

Press 1. Insert 2. Display 3. Search 4.Exit

1

enter a value to insert into hash table

22

Press 1. Insert 2. Display 3. Search 4.Exit

2

elements in the hash table are

at index 0 value = 0

at index 1 value = 0

at index 2 value = 12

at index 3 value = 13

at index 4 value = 22

at index 5 value = 0

at index 6 value = 0

at index 7 value = 0

at index 8 value = 0

at index 9 value = 0

Press 1. Insert 2. Display 3. Search 4.Exit

3

enter search element

12

value is found at index 2

Press 1. Insert 2. Display 3. Search 4.Exit

2 3

enter search element

23

value is not found

Press 1. Insert 2. Display 3. Search 4.Exit

4

nt