SORTING

BUBBLE SORT

STEP 1) START

STEP 2) INPUT THE NUMBER OF TERMS (N)

STEP 3) INPUT THE ARRAY A[] USING FOR LOOP

STEP 4) FOR I TO N-1 DO{

FOR I TO N-I-1 DO{

IF A[J]>A[J+1]

SWAP THEM

}

}

STEP 5)STOP

CODE

#include <stdio.h>

int main() {

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

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

scanf("%d", &n);

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

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

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

}

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

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

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

temp = a[j];

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

a[j + 1] = temp;

}

}

}

printf("Sorted array: ");

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

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

}

return 0;

}

BFS

BFS stands for Breadth-First Search. It's a graph traversal algorithm that starts at a given node (often referred to as the "source" node) and explores all of its neighbors at the present depth level before moving on to the nodes at the next depth level.

1. Start with the given source node.

2. Enqueue the source node.

3. Mark the source node as visited.

4. While the queue is not empty:

- Dequeue a node from the queue.

- Visit the dequeued node.

- Enqueue all adjacent nodes of the dequeued node that have not been visited and mark them as visited.

5. Repeat step 4 until the queue is empty.

#include<stdio.h>

#define MAX 100

void BFS(int adj[][MAX], int visited[], int start) {

int queue[MAX], rear = -1, front = -1, i;

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

visited[i] = 0;

}

queue[++rear] = start;

visited[start] = 1;

while (front != rear) {

int current = queue[++front];

printf("%d ", current); // Print or process the current node

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

if (adj[current][i] && !visited[i]) {

queue[++rear] = i;

visited[i] = 1;

}

}

}

}

int main() {

int visited[MAX] = {0};

int adj[MAX][MAX], i, j;

printf("Enter the values of the graph:\n");

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

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

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

}

}

printf("Breadth First Traversal ");

BFS(adj, visited, 0);

printf("\n");

return 0;

}

DFS

Depth-First Search (DFS) is another fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. It traverses deeper into the graph or tree structure before visiting neighboring nodes.

1. Start with a stack data structure (or recursion) and push the root node onto it (or make the initial function call with the root node).

2. Pop a node from the stack (or obtain the current node in the recursive call) and mark it as visited.

3. Explore all unvisited adjacent nodes of the popped node.

4. Push each unvisited adjacent node onto the stack (or recursively call the function with each unvisited adjacent node).

5. Repeat steps 2-4 until the stack is empty (or until there are no unvisited nodes in the recursive calls).

#include<stdio.h>

#define MAX 100

void DFS(int adj[][MAX], int visited[], int start) {

printf("%d ", start); // Print or process the current node

visited[start] = 1;

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

if (adj[start][i] && !visited[i]) {

DFS(adj, visited, i);

}

}

}

int main() {

int visited[MAX] = {0};

int adj[MAX][MAX], i, j;

printf("Enter the values of the graph:\n");

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

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

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

}

}

printf("Depth First Traversal starting from node 0: ");

DFS(adj, visited, 0);

printf("\n");

return 0;

}

POLYNOMIAL ADDITION

#include <stdio.h>

struct poly {

int exp;

int coe;

} p1[10], p2[10], p3[10];

void read(struct poly p[], int \*t);

int add(struct poly p1[], struct poly p2[], int t1, int t2, struct poly p3[]);

void display(struct poly p[], int terms);

int main() {

int t1 = 0, t2 = 0, t3;

read(p1, &t1);

printf("First polynomial\n");

display(p1, t1);

read(p2, &t2);

printf("Second polynomial\n");

display(p2, t2);

t3 = add(p1, p2, t1, t2, p3);

printf("\n\nResult polynomial\n");

display(p3, t3);

return 0;

}

void read(struct poly p[], int \*t) {

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

scanf("%d", t);

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

for (int i = 0; i < \*t; i++) {

printf("Enter the coefficient: ");

scanf("%d", &p[i].coe);

printf("Enter the exponent: ");

scanf("%d", &p[i].exp);

}}

void display(struct poly p[], int terms) {

for (int k = 0; k < terms; k++) {

printf("%d(x^%d)", p[k].coe, p[k].exp);

if (k < terms - 1) {

printf(" + ");

} }

printf("\n");

}

int add(struct poly p1[], struct poly p2[], int t1, int t2, struct poly p3[]) {

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

while (i < t1 && j < t2) {

if (p1[i].exp == p2[j].exp) {

p3[k].coe = p1[i].coe + p2[j].coe;

p3[k].exp = p1[i].exp;

i++;

j++;

k++;

} else if (p1[i].exp > p2[j].exp) {

p3[k].coe = p1[i].coe;

p3[k].exp = p1[i].exp;

i++;

k++;

} else {

p3[k].coe = p2[j].coe;

p3[k].exp = p2[j].exp;

j++;

k++;

} }

while (i < t1) {

p3[k].coe = p1[i].coe;

p3[k].exp = p1[i].exp;

i++;

k++;

}

while (j < t2) {

p3[k].coe = p2[j].coe;

p3[k].exp = p2[j].exp;

j++;

k++;

}

return k;

}

1. Structure Definition:

struct poly {

int exp;

int coe;

};

2. Input Function - `read()

- This function reads the number of terms in a polynomial and then reads the coefficients and exponents for each term.

- It takes an array of structures (`p[]`) and a pointer to an integer (`t`) representing the number of terms.

3. Display Function - `display()

- This function displays a polynomial.

- It takes an array of structures (`p[]`) and the number of terms (`terms`) as input.

- It prints each term of the polynomial with its coefficient and exponent.

4. Addition Function - `add()

- This function adds two polynomials (`p1` and `p2`) and stores the result in `p3`.

- It iterates through both input polynomials and combines terms with the same exponent.

- If the exponent of a term in `p1` is equal to the exponent of a term in `p2`, their coefficients are added, and the result is stored in `p3`.

- If the exponents are not equal, the term with the smaller exponent is copied directly to `p3`.

- If one polynomial is exhausted before the other, the remaining terms of the other polynomial are copied to `p3`.

- It returns the number of terms in the resulting polynomial.

5. Main Function - `main()

- It declares arrays `p1`, `p2`, and `p3` to store the polynomials.

- It reads and displays the first polynomial (`p1`).

- It reads and displays the second polynomial (`p2`).

- It adds the two polynomials (`p1` and `p2`) and displays the result (`p3`).

INSERTION SORT

#include <stdio.h>

void insert(int a[], int n)

int i, j, temp;

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

{

temp = a[i];

j = i - 1;

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

{

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

j = j - 1;

}

a[j + 1] = temp;

}

}

int main()

{

int a[10], i, n;

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

scanf("%d", &n);

printf("Enter the elements: ");

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

{

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

}

insert(a, n);

printf("\nAfter sorting, array elements are:\n");

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

{

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

}

return 0;

}

1. Consider the first element of the array as sorted.
2. Pick the next element in the array.
3. Compare the picked element with the sorted subarray.
4. If the picked element is smaller than the current element in the sorted subarray, move the current element one position to the right.
5. Repeat steps 3 and 4 until you find the correct position for the picked element.
6. Insert the picked element into the sorted subarray at the correct position.
7. Repeat steps 2 to 6 until the entire array is sorted.

PREODER

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left;

struct node \*right;

}

void preorder(struct node \*root) {

if (root == NULL)

return;

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

preorder(root->left);

preorder(root->right);

}

int main() {

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

root->data = 1;

root->left = (struct node \*)malloc(sizeof(struct node));

root->left->data = 2;

root->left->left = NULL;

root->left->right = NULL;

root->right = (struct node \*)malloc(sizeof(struct node));

root->right->data = 3;

root->right->left = NULL;

root->right->right = NULL;

printf("Preorder traversal: ");

preorder(root);

printf("\n");

free(root->left);

free(root->right);

free(root);

return 0;

}

Algorithm PreorderTraversal(root):

1. If root is NULL, return.

2. Print the data of root.

3. Recursively call PreorderTraversal(root->left).

4. Recursively call PreorderTraversal(root->right).

PREFIX EVALUVATION

1. Define a stack data structure to hold operands.

2.Start scanning the expression from right to left.

3. For each character in the expression:

- If it's a digit, push it onto the stack after converting it to an integer.

- If it's an operator (+, -, \*, /), pop two operands from the stack.

- Perform the operation on the two operands according to the operator encountered.

- Push the result back onto the stack.

4. Continue this process until the entire expression is scanned.

5. After scanning, the final result will be stored at the top of the stack.

6. Pop this result from the stack and return it as the final result of the evaluation.

CODE

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX\_SIZE 100

struct Stack {

int top;

int items[MAX\_SIZE];

};

void push(struct Stack \*s, int value);

int pop(struct Stack \*s);

int evaluatePrefix(char \*expression);

int main() {

char expression[MAX\_SIZE];

printf("Enter the prefix expression: ");

fgets(expression, sizeof(expression), stdin);

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

int result = evaluatePrefix(expression);

printf("Result: %d\n", result);

return 0;

}

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

if (s->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

exit(EXIT\_FAILURE);

} else {

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

}

}

int pop(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(EXIT\_FAILURE);

} else {

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

}

}

int evaluatePrefix(char \*expression) {

struct Stack s;

s.top = -1;

int length = 0;

while (expression[length] != '\0')

length++;

for (int i = length - 1; i >= 0; i--) {

if (isdigit(expression[i])) {

push(&s, expression[i] - '0');

} else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '\*' || expression[i] == '/') {

int operand1 = pop(&s);

int operand2 = pop(&s);

int result;

switch (expression[i]) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

default:

printf("Invalid operator\n");

exit(EXIT\_FAILURE);

}

push(&s, result);

}

}

return pop(&s);

}

LINEAR SERACH

Algorithm LinearSearch(arr, n, target):

1. Initialize an index variable to store the position of the target element.

2. Iterate over each element of the array from index 0 to n-1:

a. If the current element is equal to the target:

i. Store the index of the current element.

ii. Exit the loop.

3. If the target element is found:

a. Return the index where the target element was found.

4. If the target element is not found:

a. Return -1.

CODE

#include <stdio.h>

// Function to perform linear search

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

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

if (arr[i] == target) {

return i; // Return the index where target is found

}

}

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

}

int main() {

int arr[] = {2, 5, 7, 1, 9, 4};

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

int target = 7;

int index = linearSearch(arr, n, target);

if (index != -1) {

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

} else {

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

}

return 0;

}

**STACK**

Algorithm for Stack Operations:

1. Start

2. Initialize stack and top variable.

3. Define push function:

a. If the stack is full, print "Stack is full".

b. Otherwise, increment top by 1.

c. Set stack[top] to the given value.

d. Print "Pushed <value> onto the stack".

4. Define pop function:

a. If the stack is empty, print "Stack is empty".

b. Otherwise, print "Popped <value> from the stack" and decrement top by 1.

5. Define display function:

a. If the stack is empty, print "Stack is empty".

b. Otherwise, print "Stack elements: " followed by all elements in the stack from index 0 to top.

6. In the main function:

a. Take input for the size of the stack.

b. Enter a loop to repeatedly prompt for choices until the user chooses to exit.

c. Display a menu with options for push, pop, display, and exit.

d. Based on the user's choice, call the corresponding function: push, pop, display, or exit.

7. End

#include <stdio.h>

#define MAX\_SIZE 50

int stack[MAX\_SIZE];

int top = -1;

void push(int val) {

if (top == MAX\_SIZE - 1) {

printf("Stack is full\n");

} else {

top++;

stack[top] = val;

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

}}

void pop() {

if (top == -1) {

printf("Stack is empty\n");

} else {

printf("Popped %d from the stack\n", stack[top]);

top--;

}}

void display() {

if (top == -1) {

printf("Stack is empty\n");

} else {

printf("Stack elements: ");

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

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

}

printf("\n");

}}

int main() {

int choice, value;

printf("Enter the size of the stack (up to %d): ", MAX\_SIZE);

scanf("%d", &top);

while (1) {

printf("Enter your choice:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display\n");

printf("4. Exit\n");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to push: ");

scanf("%d", &value);

push(value);

break;

case 2:

pop();

break;

case 3:

display();

break;

case 4:

printf("Exiting the program\n");

return 0;

default:

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

}

}

return 0;

}