**Matrix multiplication:**

#include<stdio.h>

#include<stdlib.h>

**int** main(){

**int** a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

system("cls");

printf("enter the number of row=");

scanf("%d",&r);

printf("enter the number of column=");

scanf("%d",&c);

printf("enter the first matrix element=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

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

}

}

printf("enter the second matrix element=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

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

}

}

printf("multiply of the matrix=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

mul[i][j]=0;

**for**(k=0;k<c;k++)

{

mul[i][j]+=a[i][k]\*b[k][j];

}

}

}

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

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

}

printf("\n");

}

**return** 0;

}

**Even/Odd:**

#include<stdio.h>

int main()

{

int num;

scanf("%d",&num);

**if**(num % **2** == **0**)

printf("Even");

**else**

printf("Odd");

**return** **0**;

}

**Factorial(without recursion):**

#include<stdio.h>

int main(){

int i=1,f=1,num;

printf("Enter a number: ");

scanf("%d",&num);

while(i<=num){

f=f\*i;

i++;

}

printf("Factorial of %d is: %d",num,f);

return 0;

}

**Factorial(with recursion):**

#include<stdio.h>

int fact(int);

int main() {

int num,f;

printf("\nEnter a number: ");

scanf("%d",&num);

f=fact(num);

printf("\nFactorial of %d is: %d",num,f);

return 0;

}

int fact(int n) {

if(n==1)

return 1; else

return(n\*fact(n-1));

}

**Fibonacci(without recursion):**

#**include**<stdio.h>

**int** **main**()

{

**int** n1=0,n2=1,n3,i,num;

printf("Number of elements:");

scanf("%d",&num);

//To print first 0, and 1.

printf("\n%d %d",n1,n2);

**for**(i=2; i < num; ++i)

{

n3=n1+n2;

printf(" %d",n3);

n1=n2;

n2=n3;

}

**return** 0;

}

**Fibonacci(with recursion):**

#**include**<stdio.h>

//Function

**void** **my\_fibonacci**(**int** n){

**static** **int** n1=0,n2=1,n3;

**if**(n>0){

n3 = n1 + n2;

n1 = n2;

n2 = n3;

printf("%d ",n3);

my\_fibonacci(n-1);

}

}

**int** **main**(){

**int** n;

printf("Number of elements: ");

scanf("%d",&n);

printf("Fibonacci Series: \n");

printf("%d %d ",0,1);

my\_fibonacci(n-2);

**return** 0;

}

**Array insertion:**

#include<stdio.h>#include<stdlib.h>

void insert(int n1, int \*a, int l, int e)

{

int i;

printf(“Array after insertion is:\n”);

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

{

printf(“%d\n”,\*(a+i));

}

printf(“%d\n”,e);

for(i=l-1;i<n1;i++)

{

printf(“%d\n”,\*(a+i));

}

}

int main()

{

int \*a,n1,i,l,e;

scanf(“%d”,&n1);

a=(int\*)malloc(n1\*sizeof(int));

for(i=0;i<n1;i++)

{

scanf(“%d”,a+i);

}

scanf(“%d”,&l);

if(l<=n1)

{

scanf(“%d”,&e);

insert(n1,a,l,e);

}

else

{

printf(“Invalid Input”);

}

return 0;

}

**Array deletion:**

#include<stdio.h>#include<stdlib.h>

void delete(int n,int \*a,int l);

int main()

{

int \*a,n,i,l;

scanf(“%d”,&n);

a=(int\*)malloc(sizeof(int)\*n);

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

{

scanf(“%d”,(a+i));

}

scanf(“%d”,&l);

delete(n,a,l);

return 0;

}

void delete(int n,int \*a,int l)

{

int i,j;

if(l<=n)

{

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

{

j=i+1;

\*(a+i)=\*(a+j);

}

printf(“Array after deletion is:\n”);

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

{

printf(“%d\n”,(\*(a+i)));

}

}

else

{

printf(“Invalid Input”);

}

}

**Linear search:**

#include <stdio.h>

**int** linearSearch(**int** a[], **int** n, **int** val) {

  // Going through array sequencially

**for** (**int** i = 0; i < n; i++)

    {

**if** (a[i] == val)

**return** i+1;

    }

**return** -1;

}

**int** main() {

**int** a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array

**int** val = 41; // value to be searched

**int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array

**int** res = linearSearch(a, n, val); // Store result

  printf("The elements of the array are - ");

**for** (**int** i = 0; i < n; i++)

  printf("%d ", a[i]);

  printf("\nElement to be searched is - %d", val);

**if** (res == -1)

  printf("\nElement is not present in the array");

**else**

  printf("\nElement is present at %d position of array", res);

**return** 0;

}

**Binary search:**

#include<stdio.h>

int main()

{

int c, first, last, middle, n, search, array[100];

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

scanf("%d",&n);

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

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

scanf("%d",&array[c]);

printf("Enter value to find\n");

scanf("%d",&search);

first = 0;

last = n - 1;

middle = (first+last)/2;

while( first <= last )

{

if ( array[middle] < search )

first = middle + 1;

else if ( array[middle] == search )

{

printf("%d found at location %d.\n", search, middle+1);

break;

}

else

last = middle - 1;

middle = (first + last)/2;

}

if ( first > last )

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

return 0;

}

**Linked list:**

// Linked list operations in C

#include <stdio.h>

#include <stdlib.h>

// Create a node

struct Node {

int data;

struct Node\* next;

};

// Insert at the beginning

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

// Allocate memory to a node

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

// insert the data

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

// Move head to new node

(\*head\_ref) = new\_node;

}

// Insert a node after a node

void insertAfter(struct Node\* prev\_node, int new\_data) {

if (prev\_node == NULL) {

printf("the given previous node cannot be NULL");

return;

}

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = prev\_node->next;

prev\_node->next = new\_node;

}

// Insert the the end

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* last = \*head\_ref; /\* used in step 5\*/

new\_node->data = new\_data;

new\_node->next = NULL;

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

while (last->next != NULL) last = last->next;

last->next = new\_node;

return;

}

// Delete a node

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

struct Node \*temp = \*head\_ref, \*prev;

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

\*head\_ref = temp->next;

free(temp);

return;

}

// Find the key to be deleted

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

prev = temp;

temp = temp->next;

}

// If the key is not present

if (temp == NULL) return;

// Remove the node

prev->next = temp->next;

free(temp);

}

// Search a node

int searchNode(struct Node\*\* head\_ref, int key) {

struct Node\* current = \*head\_ref;

while (current != NULL) {

if (current->data == key) return 1;

current = current->next;

}

return 0;

}

// Sort the linked list

void sortLinkedList(struct Node\*\* head\_ref) {

struct Node \*current = \*head\_ref, \*index = NULL;

int temp;

if (head\_ref == NULL) {

return;

} else {

while (current != NULL) {

// index points to the node next to current

index = current->next;

while (index != NULL) {

if (current->data > index->data) {

temp = current->data;

current->data = index->data;

index->data = temp;

}

index = index->next;

}

current = current->next;

}

}

}

// Print the linked list

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

}

// Driver program

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 1);

insertAtBeginning(&head, 2);

insertAtBeginning(&head, 3);

insertAtEnd(&head, 4);

insertAfter(head->next, 5);

printf("Linked list: ");

printList(head);

printf("\nAfter deleting an element: ");

deleteNode(&head, 3);

printList(head);

int item\_to\_find = 3;

if (searchNode(&head, item\_to\_find)) {

printf("\n%d is found", item\_to\_find);

} else {

printf("\n%d is not found", item\_to\_find);

}

sortLinkedList(&head);

printf("\nSorted List: ");

printList(head);

}

**Stack:**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;

void push(int st[], int val);

int pop(int st[]);

int peek(int st[]);

void display(int st[]);

int main(int argc, char \*argv[]) {

int val, option;

do

{

printf("\n \*\*\*\*\*MAIN MENU\*\*\*\*\*");

printf("\n 1. PUSH");

printf("\n 2. POP");

printf("\n 3. PEEK");

printf("\n 4. DISPLAY");

printf("\n 5. EXIT");

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

scanf("%d", &option);

switch(option)

{

case 1:

printf("\n Enter the number to be pushed on stack: ");

scanf("%d", &val);

push(st, val);

break;

case 2:

val = pop(st);

if(val != -1)

printf("\n The value deleted from stack is: %d", val);

break;

case 3:

val = peek(st);

if(val != -1)

printf("\n The value stored at top of stack is: %d", val);

break;

case 4:

display(st);

break;

} // end of switch

}while(option != 5);

return 0;

} // end of main ()

void push(int st[], int val)

{

if(top == MAX-1)

{

printf("\n STACK OVERFLOW");

}

else

{

top++;

st[top] = val;

}

}

int pop(int st[])

{

int val;

if(top == -1)

{

printf("\n STACK UNDERFLOW");

return -1;

}

else

{

val = st[top];

top--;

return val;

}

}

void display(int st[])

{

int i;

if(top == -1)

{printf("\n STACK IS EMPTY");}

else

{

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

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

printf("\n"); // Added for formatting purposes

}

}

int peek(int st[])

{

if(top == -1)

{

printf("\n STACK IS EMPTY");

return -1;

}

else

return (st[top]);

}

**Stack (notations):**

#include<stdio.h>

void push(char element, char stack[], int \*top, int stackSize){

if(\*top == -1){

stack[stackSize - 1] = element;

\*top = stackSize - 1;

}

else if(\*top == 0){

printf("The stack is already full. \n");

}

else{

stack[(\*top) - 1] = element;

(\*top)--;

}

}

void pop(char stack[], int \*top, int stackSize){

if(\*top == -1){

printf("The stack is empty. \n");

}

else{

printf("Element popped: %c \n", stack[(\*top)]);

// If the element popped was the last element in the stack

// then set top to -1 to show that the stack is empty

if((\*top) == stackSize - 1){

(\*top) = -1;

}

else{

(\*top)++;

}

}

}

int main() {

int stackSize = 4;

char stack[stackSize];

// A negative index shows that the stack is empty

int top = -1;

push('a', stack, &top, stackSize);

printf("Element on top: %c\n", stack[top]);

push('b',stack, &top, stackSize);

printf("Element on top: %c\n", stack[top]);

pop(stack, &top, stackSize);

printf("Element on top: %c\n", stack[top]);

pop(stack, &top, stackSize);

printf("Top: %d\n", top);

pop(stack, &top, stackSize);

return 0;

}

**Queue:**

/\*\*

\* Queue implementation using array.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Queue capacity

#define CAPACITY 100

/\*\*

\* Global queue declaration.

\*/

int queue[CAPACITY];

unsigned int size = 0;

unsigned int rear = CAPACITY - 1; // Initally assumed that rear is at end

unsigned int front = 0;

/\* Function declaration for various operations on queue \*/

int enqueue(int data);

int dequeue();

int isFull();

int isEmpty();

int getRear();

int getFront();

/\* Driver function \*/

int main()

{

int ch, data;

/\* Run indefinitely until user manually terminates \*/

while (1)

{

/\* Queue menu \*/

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

printf(" QUEUE ARRAY IMPLEMENTATION PROGRAM \n");

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

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Size\n");

printf("4. Get Rear\n");

printf("5. Get Front\n");

printf("0. Exit\n");

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

printf("Select an option: ");

scanf("%d", &ch);

/\* Menu control switch \*/

switch (ch)

{

case 1:

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

scanf("%d", &data);

// Enqueue function returns 1 on success

// otherwise 0

if (enqueue(data))

printf("Element added to queue.");

else

printf("Queue is full.");

break;

case 2:

data = dequeue();

// on success dequeue returns element removed

// otherwise returns INT\_MIN

if (data == INT\_MIN)

printf("Queue is empty.");

else

printf("Data => %d", data);

break;

case 3:

// isEmpty() function returns 1 if queue is emtpy

// otherwise returns 0

if (isEmpty())

printf("Queue is empty.");

else

printf("Queue size => %d", size);

break;

case 4:

if (isEmpty())

printf("Queue is empty.");

else

printf("Rear => %d", getRear());

break;

case 5:

if (isEmpty())

printf("Queue is empty.");

else

printf("Front => %d", getFront());

break;

case 0:

printf("Exiting from app.\n");

exit(0);

default:

printf("Invalid choice, please input number between (0-5).");

break;

}

printf("\n\n");

}

}

/\*\*

\* Enqueue/Insert an element to the queue.

\*/

int enqueue(int data)

{

// Queue is full throw Queue out of capacity error.

if (isFull())

{

return 0;

}

// Ensure rear never crosses array bounds

rear = (rear + 1) % CAPACITY;

// Increment queue size

size++;

// Enqueue new element to queue

queue[rear] = data;

// Successfully enqueued element to queue

return 1;

}

/\*\*

\* Dequeue/Remove an element from the queue.

\*/

int dequeue()

{

int data = INT\_MIN;

// Queue is empty, throw Queue underflow error

if (isEmpty())

{

return INT\_MIN;

}

// Dequeue element from queue

data = queue[front];

// Ensure front never crosses array bounds

front = (front + 1) % CAPACITY;

// Decrease queue size

size--;

return data;

}

/\*\*

\* Checks if queue is full or not. It returns 1 if queue is full,

\* overwise returns 0.

\*/

int isFull()

{

return (size == CAPACITY);

}

/\*\*

\* Checks if queue is empty or not. It returns 1 if queue is empty,

\* otherwise returns 0.

\*/

int isEmpty()

{

return (size == 0);

}

/\*\*

\* Gets, front of the queue. If queue is empty return INT\_MAX otherwise

\* returns front of queue.

\*/

int getFront()

{

return (isEmpty())

? INT\_MIN

: queue[front];

}

/\*\*

\* Gets, rear of the queue. If queue is empty return INT\_MAX otherwise

\* returns rear of queue.

\*/

int getRear()

{

return (isEmpty())

? INT\_MIN

: queue[rear];

}

**Tree traversals:**

// Tree traversal in C

#include <stdio.h>

#include <stdlib.h>

struct node {

int item;

struct node\* left;

struct node\* right;

};

// Inorder traversal

void inorderTraversal(struct node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

// preorderTraversal traversal

void preorderTraversal(struct node\* root) {

if (root == NULL) return;

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// postorderTraversal traversal

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(value) {

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

newNode->item = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Insert on the left of the node

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

root->left = createNode(value);

return root->left;

}

// Insert on the right of the node

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

root->right = createNode(value);

return root->right;

}

int main() {

struct node\* root = createNode(1);

insertLeft(root, 12);

insertRight(root, 9);

insertLeft(root->left, 5);

insertRight(root->left, 6);

printf("Inorder traversal \n");

inorderTraversal(root);

printf("\nPreorder traversal \n");

preorderTraversal(root);

printf("\nPostorder traversal \n");

postorderTraversal(root);

}

**Hashing using linear probing:**

#include <stdio.h>  
#include<stdlib.h>  
#define TABLE\_SIZE 10  
  
int h[TABLE\_SIZE]={NULL};  
  
void insert()  
{  
  
 int key,index,i,flag=0,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] == NULL)  
     {  
        h[index]=key;  
         break;  
     }  
  
    }  
  
    if(i == TABLE\_SIZE)  
  
     printf("\nelement cannot be inserted\n");  
}  
void search()  
{  
  
 int key,index,i,flag=0,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)  
    {  
      printf("value is found at index %d",index);  
      break;  
    }  
  }  
  if(i == TABLE\_SIZE)  
    printf("\n value is not found\n");  
}  
void display()  
{  
  
  int i;  
  
  printf("\nelements in the hash table are \n");  
  
  for(i=0;i< TABLE\_SIZE; i++)  
  
  printf("\nat index %d \t value =  %d",i,h[i]);  
  
}  
main()  
{  
    int opt,i;  
    while(1)  
    {  
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");  
        scanf("%d",&opt);  
        switch(opt)  
        {  
            case 1:  
                insert();  
                break;  
            case 2:  
                display();  
                break;  
            case 3:  
                search();  
                break;  
            case 4:exit(0);  
        }  
    }  
}

**Insertion sort:**

#include <stdio.h>

**void** insert(**int** a[], **int** n) /\* function to sort an aay with insertion sort \*/

{

**int** i, j, temp;

**for** (i = 1; i < n; i++) {

        temp = a[i];

        j = i - 1;

**while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/

        {

            a[j+1] = a[j];

            j = j-1;

        }

        a[j+1] = temp;

    }

}

**void** printArr(**int** a[], **int** n) /\* function to print the array \*/

{

**int** i;

**for** (i = 0; i < n; i++)

        printf("%d ", a[i]);

}

**int** main()

{

**int** a[] = { 12, 31, 25, 8, 32, 17 };

**int** n = **sizeof**(a) / **sizeof**(a[0]);

    printf("Before sorting array elements are - \n");

    printArr(a, n);

    insert(a, n);

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

    printArr(a, n);

**return** 0;

}

**Merge sort:**

// C program for Merge Sort

#include <stdio.h>

#include <stdlib.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

// Create temp arrays

int L[n1], R[n2];

// Copy data to temp arrays

// L[] and R[]

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

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

// Merge the temp arrays back

// into arr[l..r]

// Initial index of first subarray

i = 0;

// Initial index of second subarray

j = 0;

// Initial index of merged subarray

k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements

// of L[], if there are any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of

// R[], if there are any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// l is for left index and r is

// right index of the sub-array

// of arr to be sorted

void mergeSort(int arr[], int l, int r)

{

if (l < r) {

// Same as (l+r)/2, but avoids

// overflow for large l and h

int m = l + (r - l) / 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

// UTILITY FUNCTIONS

// Function to print an array

void printArray(int A[], int size)

{

int i;

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

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

printf("\n");

}

// Driver code

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}

**Quick sort:**

#include <stdio.h>

// Function to swap two elements

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

**int** t = \*a;

    \*a = \*b;

    \*b = t;

}

**int** partition(**int** arr[], **int** low, **int** high) {

**int** pivot = arr[high];

**int** i = (low - 1);

**for** (**int** j = low; j <= high - 1; 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);

    }

}

// Function to print the array

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

**int** i;

**for** (i = 0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

**int** main() {

**int** arr[] = { 12, 17, 6, 25, 1, 5 };

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

    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");

    printArray(arr, n);

**return** 0;

}

**Heap sort:**

#include **<stdio.h>**

/\* function to heapify a subtree. Here 'i' is the

index of root node in array a[], and 'n' is the size of heap. \*/

void heapify(int a[], int n, int i)

{

    int largest = i; // Initialize largest as root

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

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

    // If left child is larger than root

    if (left **<** **n** && a[left] **>** a[largest])

        largest = left;

    // If right child is larger than root

    if (right **<** **n** && a[right] **>** a[largest])

        largest = right;

    // If root is not largest

    if (largest != i) {

        // swap a[i] with a[largest]

        int temp = a[i];

        a[i] = a[largest];

        a[largest] = temp;

        heapify(a, n, largest);

    }

}

/\*Function to implement the heap sort\*/

void heapSort(int a[], int n)

{

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

        heapify(a, n, i);

    // One by one extract an element from heap

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

        /\* Move current root element to end\*/

        // swap a[0] with a[i]

        int temp = a[0];

        a[0] = a[i];

        a[i] = temp;

        heapify(a, i, 0);

    }

}

/\* function to print the array elements \*/

void printArr(int arr[], int n)

{

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

    {

        printf("%d", arr[i]);

        printf(" ");

    }

}

int main()

{

    int a[] = {48, 10, 23, 43, 28, 26, 1};

    int n = sizeof(a) / sizeof(a[0]);

    printf("Before sorting array elements are - \n");

    printArr(a, n);

    heapSort(a, n);

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

    printArr(a, n);

    return 0;

}

**AVL tree:**

1. #include<stdio.h>
2. #include<stdlib.h>
4. // structure of the tree node
5. **struct** node
6. {
7. **int** data;
8. **struct** node\* left;
9. **struct** node\* right;
10. **int** ht;
11. };
13. // global initialization of root node
14. **struct** node\* root = NULL;
16. // function prototyping
17. **struct** node\* create(**int**);
18. **struct** node\* insert(**struct** node\*, **int**);
19. **struct** node\* **delete**(**struct** node\*, **int**);
20. **struct** node\* search(**struct** node\*, **int**);
21. **struct** node\* rotate\_left(**struct** node\*);
22. **struct** node\* rotate\_right(**struct** node\*);
23. **int** balance\_factor(**struct** node\*);
24. **int** height(**struct** node\*);
25. **void** inorder(**struct** node\*);
26. **void** preorder(**struct** node\*);
27. **void** postorder(**struct** node\*);
29. **int** main()
30. {
31. **int** user\_choice, data;
32. **char** user\_continue = 'y';
33. **struct** node\* result = NULL;
35. **while** (user\_continue == 'y' || user\_continue == 'Y')
36. {
37. printf("\n\n------- AVL TREE --------\n");
38. printf("\n1. Insert");
39. printf("\n2. Delete");
40. printf("\n3. Search");
41. printf("\n4. Inorder");
42. printf("\n5. Preorder");
43. printf("\n6. Postorder");
44. printf("\n7. EXIT");
46. printf("\n\nEnter Your Choice: ");
47. scanf("%d", &user\_choice);
49. **switch**(user\_choice)
50. {
51. **case** 1:
52. printf("\nEnter data: ");
53. scanf("%d", &data);
54. root = insert(root, data);
55. **break**;
57. **case** 2:
58. printf("\nEnter data: ");
59. scanf("%d", &data);
60. root = **delete**(root, data);
61. **break**;
63. **case** 3:
64. printf("\nEnter data: ");
65. scanf("%d", &data);
66. result = search(root, data);
67. **if** (result == NULL)
68. {
69. printf("\nNode not found!");
70. }
71. **else**
72. {
73. printf("\n Node found");
74. }
75. **break**;
76. **case** 4:
77. inorder(root);
78. **break**;
80. **case** 5:
81. preorder(root);
82. **break**;
84. **case** 6:
85. postorder(root);
86. **break**;
88. **case** 7:
89. printf("\n\tProgram Terminated\n");
90. **return** 1;
92. **default**:
93. printf("\n\tInvalid Choice\n");
94. }
96. printf("\n\nDo you want to continue? ");
97. scanf(" %c", &user\_continue);
98. }
100. **return** 0;
101. }
103. // creates a new tree node
104. **struct** node\* create(**int** data)
105. {
106. **struct** node\* new\_node = (**struct** node\*) malloc (**sizeof**(**struct** node));
108. // if a memory error has occurred
109. **if** (new\_node == NULL)
110. {
111. printf("\nMemory can't be allocated\n");
112. **return** NULL;
113. }
114. new\_node->data = data;
115. new\_node->left = NULL;
116. new\_node->right = NULL;
117. **return** new\_node;
118. }
120. // rotates to the left
121. **struct** node\* rotate\_left(**struct** node\* root)
122. {
123. **struct** node\* right\_child = root->right;
124. root->right = right\_child->left;
125. right\_child->left = root;
127. // update the heights of the nodes
128. root->ht = height(root);
129. right\_child->ht = height(right\_child);
131. // return the new node after rotation
132. **return** right\_child;
133. }
135. // rotates to the right
136. **struct** node\* rotate\_right(**struct** node\* root)
137. {
138. **struct** node\* left\_child = root->left;
139. root->left = left\_child->right;
140. left\_child->right = root;
142. // update the heights of the nodes
143. root->ht = height(root);
144. left\_child->ht = height(left\_child);
146. // return the new node after rotation
147. **return** left\_child;
148. }
150. // calculates the balance factor of a node
151. **int** balance\_factor(**struct** node\* root)
152. {
153. **int** lh, rh;
154. **if** (root == NULL)
155. **return** 0;
156. **if** (root->left == NULL)
157. lh = 0;
158. **else**
159. lh = 1 + root->left->ht;
160. **if** (root->right == NULL)
161. rh = 0;
162. **else**
163. rh = 1 + root->right->ht;
164. **return** lh - rh;
165. }
167. // calculate the height of the node
168. **int** height(**struct** node\* root)
169. {
170. **int** lh, rh;
171. **if** (root == NULL)
172. {
173. **return** 0;
174. }
175. **if** (root->left == NULL)
176. lh = 0;
177. **else**
178. lh = 1 + root->left->ht;
179. **if** (root->right == NULL)
180. rh = 0;
181. **else**
182. rh = 1 + root->right->ht;
184. **if** (lh > rh)
185. **return** (lh);
186. **return** (rh);
187. }
189. // inserts a new node in the AVL tree
190. **struct** node\* insert(**struct** node\* root, **int** data)
191. {
192. **if** (root == NULL)
193. {
194. **struct** node\* new\_node = create(data);
195. **if** (new\_node == NULL)
196. {
197. **return** NULL;
198. }
199. root = new\_node;
200. }
201. **else** **if** (data > root->data)
202. {
203. // insert the new node to the right
204. root->right = insert(root->right, data);
206. // tree is unbalanced, then rotate it
207. **if** (balance\_factor(root) == -2)
208. {
209. **if** (data > root->right->data)
210. {
211. root = rotate\_left(root);
212. }
213. **else**
214. {
215. root->right = rotate\_right(root->right);
216. root = rotate\_left(root);
217. }
218. }
219. }
220. **else**
221. {
222. // insert the new node to the left
223. root->left = insert(root->left, data);
225. // tree is unbalanced, then rotate it
226. **if** (balance\_factor(root) == 2)
227. {
228. **if** (data < root->left->data)
229. {
230. root = rotate\_right(root);
231. }
232. **else**
233. {
234. root->left = rotate\_left(root->left);
235. root = rotate\_right(root);
236. }
237. }
238. }
239. // update the heights of the nodes
240. root->ht = height(root);
241. **return** root;
242. }
244. // deletes a node from the AVL tree
245. **struct** node \* **delete**(**struct** node \*root, **int** x)
246. {
247. **struct** node \* temp = NULL;
249. **if** (root == NULL)
250. {
251. **return** NULL;
252. }
254. **if** (x > root->data)
255. {
256. root->right = **delete**(root->right, x);
257. **if** (balance\_factor(root) == 2)
258. {
259. **if** (balance\_factor(root->left) >= 0)
260. {
261. root = rotate\_right(root);
262. }
263. **else**
264. {
265. root->left = rotate\_left(root->left);
266. root = rotate\_right(root);
267. }
268. }
269. }
270. **else** **if** (x < root->data)
271. {
272. root->left = **delete**(root->left, x);
273. **if** (balance\_factor(root) == -2)
274. {
275. **if** (balance\_factor(root->right) <= 0)
276. {
277. root = rotate\_left(root);
278. }
279. **else**
280. {
281. root->right = rotate\_right(root->right);
282. root = rotate\_left(root);
283. }
284. }
285. }
286. **else**
287. {
288. **if** (root->right != NULL)
289. {
290. temp = root->right;
291. **while** (temp->left != NULL)
292. temp = temp->left;
294. root->data = temp->data;
295. root->right = **delete**(root->right, temp->data);
296. **if** (balance\_factor(root) == 2)
297. {
298. **if** (balance\_factor(root->left) >= 0)
299. {
300. root = rotate\_right(root);
301. }
302. **else**
303. {
304. root->left = rotate\_left(root->left);
305. root = rotate\_right(root);
306. }
307. }
308. }
309. **else**
310. {
311. **return** (root->left);
312. }
313. }
314. root->ht = height(root);
315. **return** (root);
316. }
318. // search a node in the AVL tree
319. **struct** node\* search(**struct** node\* root, **int** key)
320. {
321. **if** (root == NULL)
322. {
323. **return** NULL;
324. }
326. **if**(root->data == key)
327. {
328. **return** root;
329. }
331. **if**(key > root->data)
332. {
333. search(root->right, key);
334. }
335. **else**
336. {
337. search(root->left, key);
338. }
339. }
341. // inorder traversal of the tree
342. **void** inorder(**struct** node\* root)
343. {
344. **if** (root == NULL)
345. {
346. **return**;
347. }
349. inorder(root->left);
350. printf("%d ", root->data);
351. inorder(root->right);
352. }
354. // preorder traversal of the tree
355. **void** preorder(**struct** node\* root)
356. {
357. **if** (root == NULL)
358. {
359. **return**;
360. }
362. printf("%d ", root->data);
363. preorder(root->left);
364. preorder(root->right);
365. }
367. // postorder traversal of the tree
368. **void** postorder(**struct** node\* root)
369. {
370. **if** (root == NULL)
371. {
372. **return**;
373. }
375. postorder(root->left);
376. postorder(root->right);
377. printf("%d ", root->data);
378. }

**BFS:**

#include <stdio.h>

int n, i, j, visited[10], queue[10], front = -1, rear = -1;

int adj[10][10];

void bfs(int v)

{

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

if (adj[v][i] && !visited[i])

queue[++rear] = i;

if (front <= rear)

{

visited[queue[front]] = 1;

bfs(queue[front++]);

}

}

void main()

{

int v;

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

scanf("%d", &n);

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

{

queue[i] = 0;

visited[i] = 0;

}

printf("Enter graph data in matrix form: **\n**");

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

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

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

printf("Enter the starting vertex: ");

scanf("%d", &v);

bfs(v);

printf("The node which are reachable are: **\n**");

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

if (visited[i])

printf("%d**\t**", i);

else

printf("BFS is not possible. Not all nodes are reachable");

return 0;

}

**DFS:**

#include<stdio.h>

#include<conio.h>

int a[20][20],reach[20],n;

void dfs(int v) {

int i;

reach[v]=1;

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

if(a[v][i] && !reach[i]) {

printf("\n %d->%d",v,i);

dfs(i);

}

}

void main() {

int i,j,count=0;

clrscr();

printf("\n Enter number of vertices:");

scanf("%d",&n);

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

reach[i]=0;

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

a[i][j]=0;

}

printf("\n Enter the adjacency matrix:\n");

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

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

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

dfs(1);

printf("\n");

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

if(reach[i])

count++;

}

if(count==n)

printf("\n Graph is connected"); else

printf("\n Graph is not connected");

getch();

**Dijkstra:**

// C program for Dijkstra's single source shortest path

// algorithm. The program is for adjacency matrix

// representation of the graph

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum

// distance value, from the set of vertices not yet included

// in shortest path tree

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

{

// Initialize min value

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;

}

// A utility function to print the constructed distance

// array

void printSolution(int dist[])

{

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

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

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

}

// Function that implements Dijkstra's single source

// shortest path algorithm for a graph represented using

// adjacency matrix representation

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

{

int dist[V]; // The output array. dist[i] will hold the

// shortest

// distance from src to i

bool sptSet[V]; // sptSet[i] will be true if vertex i is

// included in shortest

// path tree or shortest distance from src to i is

// finalized

// Initialize all distances as INFINITE and stpSet[] as

// false

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

dist[i] = INT\_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

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

// Pick the minimum distance vertex from the set of

// vertices not yet processed. u is always equal to

// src in the first iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the

// picked vertex.

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

// Update dist[v] only if is not in sptSet,

// there is an edge from u to v, and total

// weight of path from src to v through u is

// smaller than current value of dist[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 constructed distance array

printSolution(dist);

}

// driver's code

int main()

{

/\* Let us create the example graph discussed above \*/

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 } };

// Function call

dijkstra(graph, 0);

return 0;

}

**Prim’s:**

// Prim's Algorithm in C

#include<stdio.h>

#include<stdbool.h>

#define INF 9999999

// number of vertices in graph

#define V 5

// create a 2d array of size 5x5

//for adjacency matrix to represent graph

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}};

int main() {

int no\_edge; // number of edge

// create a array to track selected vertex

// selected will become true otherwise false

int selected[V];

// set selected false initially

memset(selected, false, sizeof(selected));

// set number of edge to 0

no\_edge = 0;

// the number of egde in minimum spanning tree will be

// always less than (V -1), where V is number of vertices in

//graph

// choose 0th vertex and make it true

selected[0] = true;

int x; // row number

int y; // col number

// print for edge and weight

printf("Edge : Weight\n");

while (no\_edge < V - 1) {

//For every vertex in the set S, find the all adjacent vertices

// , calculate the distance from the vertex selected at step 1.

// if the vertex is already in the set S, discard it otherwise

//choose another vertex nearest to selected vertex at step 1.

int min = INF;

x = 0;

y = 0;

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

if (selected[i]) {

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

if (!selected[j] && G[i][j]) { // not in selected and there is an edge

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

printf("%d - %d : %d\n", x, y, G[x][y]);

selected[y] = true;

no\_edge++;

}

return 0;

}

**Kruskal's:**

#include <stdio.h>

#define MAX 30

typedef struct edge {

int u, v, w;

} edge;

typedef struct edge\_list {

edge data[MAX];

int n;

} edge\_list;

edge\_list elist;

int Graph[MAX][MAX], n;

edge\_list spanlist;

void kruskalAlgo();

int find(int belongs[], int vertexno);

void applyUnion(int belongs[], int c1, int c2);

void sort();

void print();

// Applying Krushkal Algo

void kruskalAlgo() {

int belongs[MAX], i, j, cno1, cno2;

elist.n = 0;

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

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

if (Graph[i][j] != 0) {

elist.data[elist.n].u = i;

elist.data[elist.n].v = j;

elist.data[elist.n].w = Graph[i][j];

elist.n++;

}

}

sort();

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

belongs[i] = i;

spanlist.n = 0;

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

cno1 = find(belongs, elist.data[i].u);

cno2 = find(belongs, elist.data[i].v);

if (cno1 != cno2) {

spanlist.data[spanlist.n] = elist.data[i];

spanlist.n = spanlist.n + 1;

applyUnion(belongs, cno1, cno2);

}

}

}

int find(int belongs[], int vertexno) {

return (belongs[vertexno]);

}

void applyUnion(int belongs[], int c1, int c2) {

int i;

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

if (belongs[i] == c2)

belongs[i] = c1;

}

// Sorting algo

void sort() {

int i, j;

edge temp;

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

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

if (elist.data[j].w > elist.data[j + 1].w) {

temp = elist.data[j];

elist.data[j] = elist.data[j + 1];

elist.data[j + 1] = temp;

}

}

// Printing the result

void print() {

int i, cost = 0;

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

printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);

cost = cost + spanlist.data[i].w;

}

printf("\nSpanning tree cost: %d", cost);

}

int main() {

int i, j, total\_cost;

n = 6;

Graph[0][0] = 0;

Graph[0][1] = 4;

Graph[0][2] = 4;

Graph[0][3] = 0;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 4;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;

Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;

Graph[5][2] = 2;

Graph[5][3] = 0;

Graph[5][4] = 3;

Graph[5][5] = 0;

Graph[5][6] = 0;

kruskalAlgo();

print();

}