**BUBBLE SORT**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int i, n, temp, j, arr[10];

printf("Enter the maximum elements you want to store : ");

scanf("%d", &n);

printf("Enter the elements \n");

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

{

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

}

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

{

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

{

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

{

temp = arr[j];

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

arr[j+1] = temp;

}

}

}

printf("The array sorted in ascending order is :\n");

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

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

return 0;

**INSERTION SORT**

#include<stdio.h>

void main ()

{

int i, j, k,temp;

int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

printf("\nprinting sorted elements...\n");

for(k=1; k<10; k++)

{

temp = a[k];

j= k-1;

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

{

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

j = j-1;

}

a[j+1] = temp;

}

for(i=0;i<10;i++)

{

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

} }

**LINEAR SEARCH**

#include<stdio.h>

void main ()

{

int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};

int item, i, flag;

printf("\nEnter Item which is to be searched\n");

scanf("%d",&item);

for (i = 0; i< 10; i++)

{

if(a[i] == item)

{

flag = i+1;

break;

}

else

{

flag = 0;

}

}

if(flag != 0)

{

printf("\nItem found at location %d\n",flag);

}

else

{

printf("\nItem not found\n");

}

}

**BINARY SEARCH**

#include<stdio.h>

void main()

{

int first, last, middle, size, i, key, list[100];

printf("Enter the size of the list: ");

scanf("%d",& size);

printf("Enter %d integer values in Ascending order\n", size);

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

{

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

}

printf("Enter value to be search: ");

scanf("%d", &key);

first = 0;

last = size - 1;

middle = (first+last)/2;

while (first <= last)

{

if (list[middle] <key)

{

first = middle + 1;

}

else if (list[middle] == key)

{

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

break;

}

else

{

last = middle - 1;}

middle = (first + last)/2;

}

if (first > last)

{

printf("Element Not found in the list.");

}

}

**ARRAY IMPLEMENTATION**

#include <stdio.h>

int stack[100],i,j,choice=0,n,top=-1;

void push();

void pop();

void show();

void main (){

printf("Enter the number of elements in the stack ");

scanf("%d",&n);

printf("\*\*\*Stack operations using array\*\*\*");

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

while(choice != 4){

printf("Choose one from the below options...\n");

printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");

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

scanf("%d",&choice);

switch(choice){

case 1:{

push();

break;}

case 2:{

pop();

break;}

case 3:{

show();

break;}

case 4:{

printf("Exiting....");

break;}

default:{

printf("Please Enter valid choice ");}

}

}

}

void push (){

int val;

if (top == n )

printf("\n Overflow");

else{

printf("Enter the value?");

scanf("%d",&val);

top = top +1;

stack[top] = val;

}

}

void pop (){

if(top == -1)

printf("Underflow");

else

top = top -1;

}

void show(){

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

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

}

if(top == -1){

printf("Stack is empty");

}

}

**SINGLY LINKED LIST OF ORDINARY QUEUE**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtBeginning(int value) {

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

newNode->data = value;

newNode->next = head;

head = newNode;

printf("\nNode inserted at the beginning.\n");

}

void insertAtEnd(int value) {

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

newNode->data = value;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

printf("\nNode inserted at the end.\n");

}

void insertBetween(int value, int loc1, int loc2) {

if (head == NULL) {

printf("\nList is empty. Inserting at the beginning.\n");

insertAtBeginning(value);

return;

}

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

newNode->data = value;

struct Node\* temp = head;

while (temp != NULL && (temp->data != loc1 && temp->data != loc2)) {

temp = temp->next;

}

if (temp == NULL) {

printf("\nLocation not found.\n");

} else {

newNode->next = temp->next;

temp->next = newNode;

printf("\nNode inserted between %d and %d.\n", loc1, loc2);

}

}

void removeBeginning() {

if (head == NULL) {

printf("\nThe list is empty.\n");

} else {

struct Node\* temp = head;

head = head->next;

free(temp);

printf("\nNode removed from the beginning.\n");

}

}

void removeEnd() {

if (head == NULL) {

printf("\nThe list is empty.\n");

} else if (head->next == NULL) {

free(head);

head = NULL;

printf("\nNode removed from the end.\n");

} else {

struct Node\* temp = head;

struct Node\* prev = NULL;

while (temp->next != NULL) {

prev = temp;

temp = temp->next;

}

prev->next = NULL;

free(temp);

printf("\nNode removed from the end.\n");

}

}

void removeSpecific(int value) {

if (head == NULL) {

printf("\nThe list is empty.\n");

return;

}

struct Node\* temp = head;

struct Node\* prev = NULL;

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

prev = temp;

temp = temp->next;

}

if (temp == NULL) {

printf("\nValue not found in the list.\n");

} else {

if (prev == NULL) { // Node to be deleted is the head

head = temp->next;

} else {

prev->next = temp->next;

}

free(temp);

printf("\nNode with value %d removed.\n", value);

}

}

void display() {

if (head == NULL) {

printf("\nThe list is empty.\n");

} else {

struct Node\* temp = head;

printf("\nList elements are: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

}

int main() {

int choice, value, loc1, loc2;

while (1) {

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

printf("\n1. Insert");

printf("\n2. Display");

printf("\n3. Delete");

printf("\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nEnter the value to insert: ");

scanf("%d", &value);

printf("\nWhere do you want to insert?");

printf("\n1. At the beginning");

printf("\n2. At the end");

printf("\n3. Between two nodes");

printf("\nEnter your choice: ");

scanf("%d", &loc1);

switch (loc1) {

case 1:

insertAtBeginning(value);

break;

case 2:

insertAtEnd(value);

break;

case 3:

printf("\nEnter the two values where you want to insert: ");

scanf("%d %d", &loc1, &loc2);

insertBetween(value, loc1, loc2);

break;

default:

printf("\nInvalid choice!\n");

}

break;

case 2:

display();

break;

case 3:

printf("\nHow do you want to delete?");

printf("\n1. From the beginning");

printf("\n2. From the end");

printf("\n3. Specific value");

printf("\nEnter your choice: ");

scanf("%d", &loc1);

switch (loc1) {

case 1:

removeBeginning();

break;

case 2:

removeEnd();

break;

case 3:

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

scanf("%d", &value);

removeSpecific(value);

break;

default:

printf("\nInvalid choice!\n");

}

break;

case 4:

exit(0);

default:

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

}

}

return 0;

}

**CIRCULAR LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtBeginning(int value) {

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

newNode->data = value;

if (head == NULL) {

newNode->next = newNode; // Circular link

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

newNode->next = head;

head = newNode;

temp->next = head; // Maintain circular link

}

printf("\nNode inserted at the beginning.\n");

}

void insertAtEnd(int value) {

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

newNode->data = value;

if (head == NULL) {

newNode->next = newNode; // Circular link

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Maintain circular link

}

printf("\nNode inserted at the end.\n");

}

void insertAfter(int value, int location) {

if (head == NULL) {

printf("\nThe list is empty. Cannot insert after a node.\n");

return;

}

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

newNode->data = value;

struct Node\* temp = head;

do {

if (temp->data == location) {

newNode->next = temp->next;

temp->next = newNode;

printf("\nNode inserted after %d.\n", location);

return;

}

temp = temp->next;

} while (temp != head);

printf("\nNode with value %d not found.\n", location);

}

void deleteBeginning() {

if (head == NULL) {

printf("\nThe list is empty. Cannot delete.\n");

return;

}

struct Node\* temp = head;

if (head->next == head) { // Only one node

head = NULL;

} else {

struct Node\* last = head;

while (last->next != head) {

last = last->next;

}

head = head->next;

last->next = head; // Maintain circular link

}

free(temp);

printf("\nNode deleted from the beginning.\n");

}

void deleteEnd() {

if (head == NULL) {

printf("\nThe list is empty. Cannot delete.\n");

return;

}

struct Node\* temp = head;

struct Node\* prev = NULL;

if (head->next == head) { // Only one node

head = NULL;

} else {

while (temp->next != head) {

prev = temp;

temp = temp->next;

}

prev->next = head; // Maintain circular link

}

free(temp);

printf("\nNode deleted from the end.\n");

}

void deleteSpecific(int value) {

if (head == NULL) {

printf("\nThe list is empty. Cannot delete.\n");

return;

}

struct Node\* temp = head;

struct Node\* prev = NULL;

do {

if (temp->data == value) {

if (temp == head) { // Deleting the head node

deleteBeginning();

return;

} else {

prev->next = temp->next;

free(temp);

printf("\nNode with value %d deleted.\n", value);

return;

}

}

prev = temp;

temp = temp->next;

} while (temp != head);

printf("\nNode with value %d not found.\n", value);

}

void display() {

if (head == NULL) {

printf("\nThe list is empty.\n");

return;

}

struct Node\* temp = head;

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

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

}

int main() {

int choice, value, position;

while (1) {

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

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nEnter the value to insert: ");

scanf("%d", &value);

printf("\n1. At the beginning\n2. At the end\n3. After a node\nEnter your choice: ");

scanf("%d", &position);

if (position == 1) {

insertAtBeginning(value);

} else if (position == 2) {

insertAtEnd(value);

} else if (position == 3) {

printf("\nEnter the value of the node after which to insert: ");

scanf("%d", &position);

insertAfter(value, position);

} else {

printf("\nInvalid choice!\n");

}

break;

case 2:

printf("\n1. Delete from the beginning\n2. Delete from the end\n3. Delete a specific node\nEnter your choice: ");

scanf("%d", &position);

if (position == 1) {

deleteBeginning();

} else if (position == 2) {

deleteEnd();

} else if (position == 3) {

printf("\nEnter the value of the node to delete: ");

scanf("%d", &value);

deleteSpecific(value);

} else {

printf("\nInvalid choice!\n");

}

break;

case 3:

display();

break;

case 4:

exit(0);

default:

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

}

}

return 0;

}

**DOUBLY LINKED LIST**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node

struct Node {

int data;

struct Node \*previous, \*next;

} \*head = NULL;

// Function to insert at the beginning

void insertAtBeginning(int value) {

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

newNode->data = value;

newNode->previous = NULL;

newNode->next = head;

if (head != NULL) {

head->previous = newNode;

}

head = newNode;

printf("\nNode inserted at the beginning!\n");

}

// Function to insert at the end

void insertAtEnd(int value) {

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

newNode->data = value;

newNode->next = NULL;

if (head == NULL) {

newNode->previous = NULL;

head = newNode;

} else {

struct Node \*temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->previous = temp;

}

printf("\nNode inserted at the end!\n");

}

// Function to insert after a specific node

void insertAtAfter(int value, int location) {

struct Node \*temp = head;

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

temp = temp->next;

}

if (temp == NULL) {

printf("\nThe specified node does not exist!\n");

} else {

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

newNode->data = value;

newNode->next = temp->next;

newNode->previous = temp;

if (temp->next != NULL) {

temp->next->previous = newNode;

}

temp->next = newNode;

printf("\nNode inserted after %d!\n", location);

}

}

// Function to delete the first node

void deleteBeginning() {

if (head == NULL) {

printf("\nThe list is empty! Cannot delete.\n");

return;

}

struct Node \*temp = head;

head = head->next;

if (head != NULL) {

head->previous = NULL;

}

free(temp);

printf("\nNode deleted from the beginning!\n");

}

// Function to delete the last node

void deleteEnd() {

if (head == NULL) {

printf("\nThe list is empty! Cannot delete.\n");

return;

}

struct Node \*temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

if (temp->previous != NULL) {

temp->previous->next = NULL;

} else {

head = NULL;

}

free(temp);

printf("\nNode deleted from the end!\n");

}

// Function to delete a specific node

void deleteSpecific(int value) {

if (head == NULL) {

printf("\nThe list is empty! Cannot delete.\n");

return;

}

struct Node \*temp = head;

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

temp = temp->next;

}

if (temp == NULL) {

printf("\nThe specified node does not exist!\n");

return;

}

if (temp->previous != NULL) {

temp->previous->next = temp->next;

} else {

head = temp->next;

}

if (temp->next != NULL) {

temp->next->previous = temp->previous;

}

free(temp);

printf("\nNode with value %d deleted!\n", value);

}

// Function to display the list

void display() {

if (head == NULL) {

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

return;

}

struct Node \*temp = head;

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

printf("NULL <-> ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Main function

int main() {

int choice1, choice2, value, position;

while (1) {

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

printf("\n1. Insert");

printf("\n2. Delete");

printf("\n3. Display");

printf("\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice1);

switch (choice1) {

case 1:

printf("\nEnter the value to insert: ");

scanf("%d", &value);

printf("\nSelect from the following insert options:");

printf("\n1. At the beginning");

printf("\n2. At the end");

printf("\n3. After a node");

printf("\n4. Cancel");

printf("\nEnter your choice: ");

scanf("%d", &choice2);

switch (choice2) {

case 1:

insertAtBeginning(value);

break;

case 2:

insertAtEnd(value);

break;

case 3:

printf("\nEnter the position after which you want to insert: ");

scanf("%d", &position);

insertAtAfter(value, position);

break;

case 4:

break;

default:

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

}

break;

case 2:

printf("\nSelect from the following delete options:");

printf("\n1. At the beginning");

printf("\n2. At the end");

printf("\n3. Specific node");

printf("\n4. Cancel");

printf("\nEnter your choice: ");

scanf("%d", &choice2);

switch (choice2) {

case 1:

deleteBeginning();

break;

case 2:

deleteEnd();

break;

case 3:

printf("\nEnter the value of the node to be deleted: ");

scanf("%d", &value);

deleteSpecific(value);

break;

case 4:

break;

default:

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

}

break;

case 3:

display();

break;

case 4:

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

exit(0);

default:

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

}

}

return 0;

}

**Binary search tree.**

# include <iostream>

# include <cstdlib>

using namespace std;

// Binary Search Tree Node Declaration

struct node

{

int info;

struct node \*left;

struct node \*right;

}\*root;

// Class Declaration for Binary Search Tree

class BST

{

public:

void insert(node \*,node \*) ;

void preorder(node \*);

void inorder(node \*);

void postorder(node \*);

void display(node \*, int);

BST()

{

root = NULL;

}

};

// Main Contains Menu

int main()

{

int choice, num;

BST bst;

node \*temp;

while (1)

{

// Main menu for Binary Search Tree Operations

cout<<"-----------------"<<endl;

cout<<"Operations on BST"<<endl;

cout<<"-----------------"<<endl;

cout<<"1.Insert Element "<<endl;

cout<<"2.Inorder Traversal"<<endl;

cout<<"3.Preorder Traversal"<<endl;

cout<<"4.Postorder Traversal"<<endl;

cout<<"5.Display"<<endl;

cout<<"6.Quit"<<endl;

cout<<"Enter your choice : ";

cin>>choice;

switch(choice)

{

case 1:

temp = new node;

cout<<"Enter the number to be inserted : ";

cin>>temp->info;

bst.insert(root, temp);

break;

case 2:

cout<<"Inorder Traversal of BST:"<<endl;

bst.inorder(root);

cout<<endl;

break;

case 3:

cout<<"Preorder Traversal of BST:"<<endl;

bst.preorder(root);

cout<<endl;

break;

case 4:

cout<<"Postorder Traversal of BST:"<<endl;

bst.postorder(root);

cout<<endl;

break;

case 5:

cout<<"Display BST:"<<endl;

bst.display(root,1);

cout<<endl;

break;

case 6:

exit(1);

default:

cout<<"Wrong choice"<<endl;

}

}

}

// Inserting Element into the Binary Search Tree

void BST::insert(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info;

root->left = NULL;

root->right = NULL;

cout<<"Root Node is Added"<<endl;

return;

}

if (tree->info == newnode->info)

{

cout<<"Element already in the tree"<<endl;

return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insert(tree->left, newnode);

}

else

{

tree->left = newnode;

(tree->left)->left = NULL;

(tree->left)->right = NULL;

cout<<"Node Added To Left"<<endl;

return;

}

}

else

{

if (tree->right != NULL)

{

insert(tree->right, newnode);

}

else

{

tree->right = newnode;

(tree->right)->left = NULL;

(tree->right)->right = NULL;

cout<<"Node Added To Right"<<endl;

return;

}

}

}

// Pre Order Traversal

void BST::preorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

cout<<ptr->info<<" ";

preorder(ptr->left);

preorder(ptr->right);

}

}

// In Order Traversal

void BST::inorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

inorder(ptr->left);

cout<<ptr->info<<" ";

inorder(ptr->right);

}

}

// Postorder Traversal

void BST::postorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

postorder(ptr->left);

postorder(ptr->right);

cout<<ptr->info<<" ";

}

}

// Display Binary Search Tree Structure

void BST::display(node \*ptr, int level)

{

int i;

if (ptr != NULL)

{

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

cout<<endl;

if (ptr == root)

cout<<"Root->: ";

else

{

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

cout<<" ";

}

cout<<ptr->info;

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

}

}

**Adjacency Matrix Graph**

#include<iostream>

using namespace std;

int vertArr[20][20]; //the adjacency matrix initially 0

int count = 0;

void displayMatrix(int v) {

int i, j;

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

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

cout << vertArr[i][j] << " ";

}

cout << endl;

}

}

void add\_edge(int u, int v) { //function to add edge into the matrix

vertArr[u][v] = 1;

vertArr[v][u] = 1;

}

int main() {

cout<<"Adjacency Matrix \n";

int v = 5; //there are 6 vertices in the graph

add\_edge(0, 1);

add\_edge(0, 2);

add\_edge(0, 4);

add\_edge(1, 3);

add\_edge(3, 2);

add\_edge(2, 4);

displayMatrix(v);

return 0;

}