***JAGGED ARRAYS /***

***N-DIMENSION ARRAYS***

1. JAGGED ARRAYS (SIR’S CODE)

#include <iostream>

using namespace std;

int main() {

    int c[4][4];

    //int\*\* ja = new int\*[4];

    int \*\*a,\*\*b;

    a = new int\*[4];

    b = new int\*[4];

    printf("&a = %u a = %u , &b = %u , b = %u \n",&a,a,&b,b);

    // int C[4]={3,6,2,5};

    int C[4] = {4, 4, 4, 4};

    for (int i=0; i<4; i++) {

        a[i] = new int(C[i]);

        b[i] = new int(C[i]);

    }

    //  int a[4][4],b[4][4]

    for (int x=0; x<4; x++) {

        for (int y=0; y<C[x]; y++) {

            a[x][y] = b[x][y] = x \* C[x] + y;

        }

    }

    for (int x=0; x<4; x++) {

  printf("&a[%d]= %u , %u , &b[%d] = %u , %u\n", x, &a[x],a[x], x, &b[x], b[x]);

    }

    for (int x=0; x<4; x++) {

        for (int y=0; y<C[x]; y++) {

            printf("a[%d][%d] = %d @ %u\n",x,y,a[x][y],&a[x][y]);

        }

    }

    for (int x=0; x<4; x++) {

        for (int y=0; y<4; y++) {

            printf("c[%d][%d] = %d @ %u\n",x,y,c[x][y],&c[x][y]);

        }

    }

    for (int x=0; x<4; x++) {

        for (int y=0; y<C[x]; y++) {

            printf("b[%d][%d] = %d @ %u\n",x,y,b[x][y],&b[x][y]);

        }

    }

    return 0;

}

2. ROW MAJOR

#include <iostream>

using namespace std;

int main() {

    int N;

    cout << "Enter number of dimensions: ";

    cin >> N;

    int \*S = new int[N];

    for (int i = 0; i < N; i++) {

        cout << "Enter size of Dimension " << i+1 << ": ";

        cin >> S[i];

    }

    int totalValues = 1;

    for (int i = 0; i < N; i++) {

        totalValues = totalValues \* S[i];

    }

    int e;

    int\* I = new int[N];

    for (int k = 0; k < N; k++) {

        cout << "Enter Index of Dimension: " << k+1 << endl;

        cin >> e;

        I[k] = e;

    }

    int alpha = 0; // require index

    for (int i=0 ; i<N; i++) {

int s = 1;

        for (int j=i+1 ; j<N; j++) {

            s = s \* S[j];

        }

        alpha += I[i] \* s;

    }

    int \*linearArray = new int[totalValues];

    cout << "Size of Linear Array: " << totalValues << endl;

    int \*baseAddress = &linearArray[0];

    const int size\_dt = sizeof(linearArray[0]);

    int address = int(baseAddress) + (alpha \* size\_dt);

    cout << "Base Address: " << baseAddress << endl;

    cout << "Address of given indexes " << address << endl;

    delete S;

    S = nullptr;

    delete I;

    I = nullptr;

    delete linearArray;

    linearArray = nullptr;

    return 0;

}

3. COLUMN MAJOR

#include <iostream>

using namespace std;

int main() {

    int N;

    cout << "Enter number of dimensions: ";

    cin >> N;

    int \*S = new int[N];

    for (int i = 0; i < N; i++) {

        cout << "Enter size of Dimension " << i+1 << ": ";

        cin >> S[i];

    }

    int totalValues = 1;

    for (int i = 0; i < N; i++) {

        totalValues = totalValues \* S[i];

    }

    int e;

    int\* I = new int[N];

    for (int k = 0; k < N; k++) {

        cout << "Enter Index of Dimension: " << k+1 << endl;

        cin >> e;

        I[k] = e;

    }

    int s = 1;

    int alpha = 0; // require index

    for (int i=0 ; i<N; i++) {

        for (int j=0 ; j<i; j++) {

            s = s \* S[j];

        }

        alpha += I[i] \* s;

        s = 1;

    }

    int \*linearArray = new int[totalValues];

    cout << "Size of Linear Array: " << totalValues << endl;

    int \*baseAddress = &linearArray[0];

    const int size\_dt = sizeof(linearArray[0]);

    int address = int(baseAddress) + (alpha \* size\_dt);

    cout << "Base Address: " << baseAddress << endl;

    cout << "Address of given indexes " << address << endl;

    delete S;

    S = nullptr;

    delete I;

    I = nullptr;

    delete linearArray;

    linearArray = nullptr;

    return 0;

}

***LINKED LIST***

#include <iostream>

using namespace std;

struct Node {

    int key;

    Node \*next;

};

void insert(Node \*hn, int value) {

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

    newNode->key = value;

    newNode->next = hn;

    hn = newNode;

}

void deleteNode(Node \*hn, int value) {

    Node \*current;

    Node \*previous;

    while (current->next != NULL) {

        if (current->key==value) {

            previous->next = current->next;

            free(current);

            return;

        }

        previous = current;

        current = current->next;

    }

    cout << value << " not found." << endl;

}

void display(Node\* hn) {

    while (hn!=NULL) {

        cout << hn->key << "->";

        hn = hn->next;

    }

    cout << "NULL" << endl;

}

int main() {

    int values[] = {4, 7, 1, 5, 9, 0, 1, 3, 77};

    int len = sizeof(values)/sizeof(values[0]);

    Node \*HeadNode;

    for (int i=0; i<len; i++) {

        insert(HeadNode, values[i]);

        display(HeadNode);

    }

    deleteNode(HeadNode, 9);

    display(HeadNode);

    deleteNode(HeadNode, 999);

    display(HeadNode);

    return 0;

}

***QUEUE***

1. USING ARRAY

#include <iostream>

using namespace std;

int queue[20]={0}, rear = -1, front = -1, capacity = 20, count = 0;

bool isFull() {

    if (capacity == count) {

        return true;

    }

    return false;

}

bool isEmpty() {

    if (front==-1 && rear==-1) {

        return true;

    }

    return false;

}

void enqueue(int value) {

    if (isFull()) {

        cout << "Queue Overflow" << endl;

    }

    else {

        if (front == - 1) front++;

        queue[rear++] = value;

        count++;

    }

}

int dequeue() {

    int val;

    if (front == - 1 || front > rear) {

        cout << "Queue Underflow " << endl;

    }

    else {

        val = queue[front];

        queue[front] = -1;

        if(front==rear) {

            front = -1, rear = -1;

        }

        else {

            front++;

        }

        count--;

    }

    return val;

}

int top() {

    if (isEmpty()) {

        cout << "Queue is empty." << endl;

        return -1;

    }

    else {

        return queue[front];

    }

}

void display() {

    if (front == - 1) {

        cout << "Queue is empty." << endl;

    }

    else {

        cout << "Queue elements are: ";

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

            cout << queue[i] << ", ";

        }

        cout << endl;

    }

}

int main() {

    int choice;

    cout << "1) Insert an element" << endl;

    cout << "2) Delete first element" << endl;

    cout << "3) Display all the elements" << endl;

    cout << "4) Get top element" << endl;

    cout << "5) Exit" << endl;

    do {

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                int value;

                cout << "Enter value: " << endl;

                cin >> value;

                enqueue(value);

                break;

            case 2:

                dequeue();

                break;

            case 3:

                display();

                break;

            case 4:

                cout << "Top value: " << top() << endl;

                break;

            case 5:

                cout << "Exit" << endl;

                break;

            default:

                 cout << "Invalid choice" << endl;

        }

    }

    while(choice!=5);

    return 0;

}

2. USING LINKED LIST

#include <iostream>

using namespace std;

struct QNode {

    int data;

    QNode\* next;

    QNode(int d) {

        data = d;

        next = NULL;

    }

};

QNode \*front=NULL, \*rear=NULL;

void enqueue(int value) {

    QNode \*newNode = new QNode(value);

    if (rear==NULL) {

        front = newNode;

        rear = newNode;

    }

    rear->next = newNode;

    rear = newNode;

}

int dequeue() {

    if (front==NULL) {

        return 0;

    }

    QNode \*temp = front;

    front = front->next;

    if (front==NULL) {

        rear = NULL;

    }

    int value = temp->data;

    delete temp;

    return value;

}

void display() {

    QNode \*temp = front;

    while (temp!=NULL) {

        cout << temp->data << "->";

        temp = temp->next;

    }

    cout << "NULL" << endl;

}

int main() {

    enqueue(10);

    enqueue(20);

    display();

    dequeue();

    display();

    dequeue();

    display();

    enqueue(30);

    enqueue(40);

    enqueue(50);

    display();

    dequeue();

    display();

    return 0;

}

***STACK***

1. USING ARRAY

#include <iostream>

using namespace std;

int stack[20], top=0, numOfValues=0;

bool isEmpty() {

    if (numOfValues==0) {

        return true;

    }

    return false;

}

bool isFull() {

    if (numOfValues==20) {

        return true;

    }

    return false;

}

void push(int value) {

    if (isFull()) {

        cout << "Stack Overflow" << endl;

        return;

    }

    stack[top++] = value;

    numOfValues++;

}

int pop() {

    if (isEmpty()) {

        cout << "Stack Underflow" << endl;

        return -1;

    }

    int value = stack[top];

    stack[top--] = -1;

    numOfValues--;

    return value;

}

int count() {

    return numOfValues;

}

void display() {

    cout << "Stack: ";

    for (int i=top-1; i>=0; i--) {

        cout << stack[i] << "->";

    }

    cout << endl;

}

int main() {

    for (int i=0; i<20; i++) {

        stack[i] = -1;

    }

    push(4);

    push(6);

    push(5);

    display();

    pop();

    display();

    push(9);

    display();

    pop();

    display();

    pop();

    display();

    pop();

    display();

    return 0;

}

2. USING LINKED LIST

#include <iostream>

using namespace std;

int stack[20], top=0, numOfValues=0;

bool isEmpty() {

    if (numOfValues==0) {

        return true;

    }

    return false;

}

bool isFull() {

    if (numOfValues==20) {

        return true;

    }

    return false;

}

void push(int value) {

    if (isFull()) {

        cout << "Stack Overflow" << endl;

        return;

    }

    stack[top++] = value;

    numOfValues++;

}

int pop() {

    if (isEmpty()) {

        cout << "Stack Underflow" << endl;

        return -1;

    }

    int value = stack[top];

    stack[top--] = -1;

    numOfValues--;

    return value;

}

int count() {

    return numOfValues;

}

void display() {

    cout << "Stack: ";

    for (int i=top-1; i>=0; i--) {

        cout << stack[i] << "->";

    }

    cout << endl;

}

int main() {

    for (int i=0; i<20; i++) {

        stack[i] = -1;

    }

    push(4);

    push(6);

    push(5);

    display();

    pop();

    display();

    push(9);

    display();

    pop();

    display();

    pop();

    display();

    pop();

    display();

    return 0;

}

***SEARCHING ALGORITHMS***

(LINEAR & BINARY SEARCH)

#include <iostream>

using namespace std;

int liearSearch(int values[], int size, int value) {

    for (int i=0; i<size; i++) {

        if (values[i]==value) {

            return i;

        }

    }

    return -1;

}

int binarySearch(int values[], int size, int value) {

    int low=0, high=size-1;

    while (low <= high) {

        int mid = (low + high) / 2;

        if (values[mid] == value)

            return mid;

        if (values[mid] < value)

        low = mid + 1;

        else

        high = mid - 1;

    }

    return -1;

}

int main() {

    int values[] = {2, 4, 7, 9, 12, 14, 15, 22, 33};

    int numOfValues = sizeof(values)/sizeof(values[0]);

    cout << binarySearch(values, numOfValues, 2);

    cout << binarySearch(values, numOfValues, 1);

    return 0;

}

***SORTING ALGORITHMS***

1. QUICK SORT

#include <iostream>

using namespace std;

int partition(int arr[], int start, int end) {

    int pivot = arr[start];

    int count = 0;

    for (int i = start + 1; i <= end; i++) {

        if (arr[i] <= pivot)

            count++;

    }

    int pivotIndex = start + count;

    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;

    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) i++;

        while (arr[j] > pivot) j--;

        if (i < pivotIndex && j > pivotIndex) {

            swap(arr[i++], arr[j--]);

        }

    }

    return pivotIndex;

}

void quickSort(int arr[], int start, int end) {

    if (start<end) {

        int p = partition(arr, start, end);

        quickSort(arr, start, p - 1);

        quickSort(arr, p + 1, end);

    }

}

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

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    cout << "Quick Sort" << endl;

    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};

    int size = sizeof(values) / sizeof(values[0]);

    cout << "Before Sort: "; printArray(values, size);

    quickSort(values, 0, size - 1);

    cout << "After Sort: "; printArray(values, size);

    return 0;

}

2. BUBBLE SORT

#include <iostream>

using namespace std;

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

    int i, j;

    for (i=0; i < size-1; i++) {

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

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

                swap(arr[j], arr[j + 1]);

            }

        }

    }

}

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

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    cout << "Bubble Sort" << endl;

    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};

    int size = sizeof(values) / sizeof(values[0]);

    cout << "Before Sort: "; printArray(values, size);

    bubbleSort(values, size);

    cout << "After Sort: "; printArray(values, size);

    return 0;

}

3. INSERTION SORT

#include <iostream>

using namespace std;

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

    int temp, j;

    for (int i=1; i < size; i++) {

        temp = arr[i];

        j = i - 1;

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

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = temp;

    }

}

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

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    cout << "Insertion Sort" << endl;

    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};

    int size = sizeof(values) / sizeof(values[0]);

    cout << "Before Sort: "; printArray(values, size);

    insertionSort(values, size);

    cout << "After Sort: "; printArray(values, size);

    return 0;

}

4. MERGE SORT

#include <iostream>

using namespace std;

void merge(int arr[], int lower, int mid, int upper) {

    int n1 = mid - lower + 1;

    int n2 = upper - mid;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[lower + i];

    for (int j = 0; j < n2; j++)

        M[j] = arr[mid + 1 + j];

    int i, j, k;

    i = 0;

    j = 0;

    k = lower;

    while (i < n1 && j < n2) {

        if (L[i] <= M[j]) {

            arr[k] = L[i];

            i++;

        }

        else {

            arr[k] = M[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k++] = L[i++];

    }

    while (j < n2) {

        arr[k++] = M[j++];

    }

}

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

    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

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

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    cout << "Merge Sort" << endl;

    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};

    int size = sizeof(values) / sizeof(values[0]);

    cout << "Before Sort: "; printArray(values, size);

    mergeSort(values, 0, size - 1);

    cout << "After Sort: "; printArray(values, size);

    return 0;

}

5. SELECTION SORT

#include <iostream>

using namespace std;

void selectiontionSort(int arr[], int size) {

    for (int i=0; i < size-1; i++) {

        int minimumIndex = i;

        for (int j = i+1; j < size; j++) {

            if (arr[j] < arr[minimumIndex])

                minimumIndex = j;

        }

        if (minimumIndex!=i)

            swap(arr[minimumIndex], arr[i]);

    }

}

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

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    cout << "Selection Sort" << endl;

    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};

    int size = sizeof(values) / sizeof(values[0]);

    cout << "Before Sort: "; printArray(values, size);

    selectiontionSort(values, size);

    cout << "After Sort: "; printArray(values, size);

    return 0;

}

***EXPRESSION PARSING***

#include <iostream>

#include "bits/stdc++.h"

#include <string.h>

#include <stack>

#include <cmath>

#include <map>

using namespace std;

int isOperator(char op) {

    if (op=='(') return 1;

    else if (op==')') return -1;

    else if (op=='+' || op=='-') return 2;

    else if (op=='/' || op=='\*') return 3;

    else if (op=='^') return 4;

    else return 0;

}

double operation(double a, double b, char op) {

    if (op=='+') return a+b;

    else if (op=='-') return a-b;

    else if (op=='\*') return a\*b;

    else if (op=='/') return a/b;

    else return pow(a, b);

}

char\* InfixToPostfix(char\* infix) {

    char \*postfix;

    postfix = (char\*) malloc(100 \* sizeof(char));

    stack<char> s;

    int len = strlen(infix);

    int j = 0;

    for (int i=0; i<len; i++) {

        char current = infix[i];

        int op = isOperator(current);

        if (op==-1) {

            while (s.top()!='(') {

                postfix[j++] = s.top();

                s.pop();

            }

            s.pop();

            continue;

        }

        if (op==0) {

            postfix[j++] = current;

        }

        else {

            if ((s.empty()) || op==1)   s.push(current);

            else {

                while (op <= isOperator(s.top())) {

                    postfix[j++] = s.top();

                    s.pop();

                    if (s.empty()) break;

                }

                s.push(current);

            }

        }

    }

    while (!s.empty()) {

        postfix[j++] = s.top();

        s.pop();

    }

    postfix[j] = '\0';

    return postfix;

}

double postfixEvaluation(char\* postfix, map<char, double> &values) {

    stack<char> solution;

    double ans;

    int length = strlen(postfix);

    for (int k=0; k<length; k++) {

        char current = postfix[k];

        if (isOperator(current)==0) {

            solution.push(current);

        }

        else {

            double b = values[solution.top()];

            solution.pop();

            double a = values[solution.top()];

            solution.pop();

            ans = operation(a, b, current);

            // cout << a << current << b << " = " << ans << endl;

            values[char(ans)] = ans;

            solution.push(char(ans));

        }

    }

    solution.pop();

    return ans;

}

int main() {

    char infix[100] = "A+B\*C/(E-F)\*(A^(B-C/D))";

    // cout << "Enter an expression \n";

    // char infix[100];

    // cin >> infix;

    int len = strlen(infix);

    map<char, double> values;

    for (int z=0; z<len; z++) {

        char c = infix[z];

        double a;

        if (isOperator(c)==0 && values.count(c)==0) {

            cout << "Enter value of " << c << ": ";

            cin >> a;

            values[c] = a;

        }

    }

    cout << endl << "Infix: " << infix << endl << endl;

    char\* postfix = InfixToPostfix(infix);

    cout << "Postfix: " << postfix << endl << endl;

    double ans = postfixEvaluation(postfix, values);

    cout << "Solution: "<< ans << endl;

}

***TREES***

1. EXPRESSION TREE

#include <iostream>

#include "infToPos.h"

using namespace std;

struct Node {

    char data;

    Node \*left;

    Node \*right;

};

Node\* createExpressionTree(char \*postfix) {

    stack<struct Node\*> s;

    struct Node \*n, \*l, \*r;

    int len = strlen(postfix);

    for (int i=0; i<len; i++) {

        char current = postfix[i];

        n = new Node;

        n->data = current;

        n->left = NULL;

        n->right = NULL;

        if (isOperator(current)!=0) {

            r = s.top();

            s.pop();

            l = s.top();

            s.pop();

            n->right = r;

            n->left = l;

        }

        s.push(n);

    }

    struct Node \*ETRoot;

    ETRoot = s.top();

    s.pop();

    return ETRoot;

}

void traverseInOrder(struct Node \*temp) {

    if (temp!=NULL) {

        traverseInOrder(temp->left);

        cout << temp->data;

        traverseInOrder(temp->right);

    }

}

void traversePreOrder(struct Node \*temp) {

    if (temp!=NULL) {

        cout << temp->data;

        traversePreOrder(temp->left);

        traversePreOrder(temp->right);

    }

}

void traversePostOrder(struct Node \*temp) {

    if (temp!=NULL) {

        traversePostOrder(temp->left);

        traversePostOrder(temp->right);

        cout << temp->data;

    }

}

int main() {

    // char infix[100] = "a+b";

    char infix[100] = "A+B\*C/(E-F)\*(A^(B-C/D))";

    // inputExpression(infix);

    cout << infix << endl;

    char postfix[100];

    infixToPostfix(infix, postfix);

    cout << postfix << "\n\n" << endl;

    struct Node \*ETRoot = new Node;

    ETRoot = createExpressionTree(postfix);

    cout << "Pre-Order: ";  traversePreOrder(ETRoot); cout << endl;

    cout << "Post-Order: "; traversePostOrder(ETRoot); cout << endl;

    cout << "In-Order: ";   traverseInOrder(ETRoot); cout << endl;

    return 0;

}

2. BINARY SEARCH TREE

#include <iostream>

using namespace std;

struct Node {

    int key;

    Node \*left, \*right;

};

Node\* newNode(int item) {

    Node \*temp = (Node\*)malloc(sizeof(Node));

    temp->key = item;

    temp->left = temp->right = NULL;

    return temp;

}

// Inorder Traversal

void inorder(Node \*root) {

  if (root != NULL) {

    inorder(root->left);

    cout << root->key << " -> ";

    inorder(root->right);

  }

}

// Preorder Traversal

void preorder(Node \*root) {

  if (root != NULL) {

    cout << root->key << " -> ";

    inorder(root->left);

    inorder(root->right);

  }

}

// Postorder Traversal

void postorder(Node \*root) {

  if (root != NULL) {

    inorder(root->left);

    inorder(root->right);

    cout << root->key << " -> ";

  }

}

Node\* insert(Node \*Node, int value) {

    if (Node==NULL) return newNode(value);

    if (value < Node->key)

        Node->left = insert(Node->left, value);

    else

        Node->right = insert(Node->right, value);

    return Node;

}

Node\* minValueNode(Node \*HeadNode) {

    Node \*current = HeadNode;

    while (current && current->left != NULL)

        current = current->left;

    return current;

}

Node\* maxValueNode(Node \*HeadNode) {

    Node \*current = HeadNode;

    while (current && current->right != NULL)

        current = current->right;

    return current;

}

Node\* deleteNode(Node \*root, int key) {

    // Return if the tree is empty

    if (root == NULL) return root;

    // Find the Node to be deleted

    if (key < root->key) {

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

    }

    else if (key > root->key) {

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

    }

    else {  // key == root->key

        // If the Node is with only one child or no child

        if (root->left == NULL) {

            Node \*temp = root->right;

            free(root);

            return temp;

        }

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

            Node \*temp = root->left;

            free(root);

            return temp;

        }

        // If the Node has two children

        struct Node \*temp = minValueNode(root->right);

        // Place the inorder successor in position of the Node to be deleted

        root->key = temp->key;

        // Delete the inorder successor

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

// Driver code

int main() {

    struct Node \*root = NULL;

    root = insert(root, 8);

    root = insert(root, 3);

    root = insert(root, 1);

    root = insert(root, 6);

    root = insert(root, 7);

    root = insert(root, 10);

    root = insert(root, 14);

    root = insert(root, 4);

    cout << "Preorder traversal: ";

    preorder(root);

    cout << endl;

    cout << "Inorder traversal: ";

    inorder(root);

    cout << endl;

    cout << "Postorder traversal: ";

    postorder(root);

    cout << endl;

    root = deleteNode(root, 3);

    cout << endl << "After deleting 3" << endl;

    cout << "Inorder traversal: ";

    inorder(root);

    cout << endl;

}

3. BINARY HEAP / PRIORITY QUEUE

#include<iostream>

using namespace std;

class BinaryHeap {

    private:

        int\* PQ;

        int count;

        int capacity;

        int n;

    public:

        BinaryHeap() {

            capacity = 5;

            PQ = (int\*) malloc(capacity\*sizeof(int));

            count = 0;

            n = 1;

        }

        void display(){

            cout << "count = " << count << ": ";

            for (int i=1; i<count+1; i++) {

                cout << PQ[i] << " ";

            }

            cout << endl;

        }

        void heapify() {

            int i = count;

            while (i!=1) {

                int k = int(i/2);

                if (PQ[i]<PQ[k]) {

                    int temp = PQ[k];

                    PQ[k] = PQ[i];

                    PQ[i] = temp;

                    i = k;

                }

                else break;

            }

        }

        void enqueue(int data) {

            if (count == capacity) {

                capacity \*= 2;

                int \*temp = (int\*) malloc(capacity\*sizeof(int));

                for (int i=1; i<count+1; i++) {

                    temp[i] = PQ[i];

                }

                free(PQ);

                PQ = temp;

            }

            PQ[++count] = data;

            heapify();

        }

        void dequeue() {

            if ((2\*n <= count) && ((2\*n)+1 <= count)) {

                if (PQ[2\*n] < PQ[(2\*n)+1]) {

                    PQ[n] = PQ[2\*n];

                    n \*= 2;

                }

                else {

                    PQ[n] = PQ[(2\*n)+1];

                    n = (2\*n)+1;

                }

                dequeue();

            }

            else {

                PQ[n] = PQ[count];

                count--;

            }

            n = 1;

        }

};

int main() {

    BinaryHeap PriorityQueue;

    PriorityQueue.enqueue(8);

    PriorityQueue.enqueue(13);

    PriorityQueue.enqueue(5);

    PriorityQueue.enqueue(10);

    PriorityQueue.enqueue(3);

    PriorityQueue.enqueue(9);

    PriorityQueue.enqueue(2);

    PriorityQueue.enqueue(-1);

    PriorityQueue.enqueue(6);

    PriorityQueue.enqueue(4);

    PriorityQueue.display();

    PriorityQueue.dequeue();

    PriorityQueue.display();

    PriorityQueue.dequeue();

    PriorityQueue.display();

    PriorityQueue.dequeue();

    PriorityQueue.display();

    return 0;

}

***HASHING***

1. OPEN HASHING

#include <iostream>

using namespace std;

bool isPrime(int n) {

    // Corner cases

    if (n <= 1)  return false;

    if (n <= 3)  return true;

    // This is checked so that we can skip

    // middle five numbers in below loop

    if (n%2 == 0 || n%3 == 0) return false;

    for (int i=5; i\*i<=n; i=i+6)

        if (n%i == 0 || n%(i+2) == 0)

           return false;

    return true;

}

int nextPrime(int N) {

    // Base case

    if (N <= 1)

        return 2;

    int prime = N;

    bool found = false;

    // Loop continuously until isPrime returns

    // true for a number greater than n

    while (!found) {

        prime++;

        if (isPrime(prime))

            found = true;

    }

    return prime;

}

struct HNode {

    int key;

    HNode\* next;

    HNode(int x) {

        this->key = x;

        next = NULL;

    }

};

HNode\*\* openHashing(int &HS, int \*values, int totalValues) {

    HNode \*\*HT = (HNode\*\*) malloc(HS \* sizeof(HNode\*));

    int numOfValues = 0;

    float load = 0;

    for (int i=0; i<HS; i++) {

        HT[i] = NULL;

    }

    int HI;

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

        if (load >= 1) {

            int tempValues[numOfValues];

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

                tempValues[j] = values[j];

            }

            free(HT);

            HS = nextPrime(HS \* 2);

            HT = openHashing(HS, tempValues, numOfValues);

        }

        int key = values[i];

        HNode \*hn = new HNode(key);

        HI = key % HS;

        if (HT[HI]==NULL) {

            HT[HI] = hn;

        }

        else {

            HNode \*temp = HT[HI];

            HT[HI] = hn;

            HT[HI]->next = temp;

        }

        numOfValues++;

        load = float(numOfValues)/float(HS);

    }

    return HT;

}

void printHash(HNode\*\* array, int len) {

    for (int i=0; i<len; i++) {

        cout << "Index " << i << ": ";

        if (array[i]==NULL) {

            cout << "NULL" << endl;

        }

        else {

            HNode \*temp = array[i];

            while (temp!=NULL) {

                cout << temp->key << "->";

                temp = temp->next;

            }

            cout << endl;

        }

    }

}

void printList(int\* list, int len) {

    cout << "[";

    for (int i=0; i<len; i++) {

        cout << list[i] << ", ";

    }

    cout << "]" << endl;

}

bool exists(HNode \*\*HT, int HS, int val) {

    int HI = val % HS;

    HNode \*temp = HT[HI];

    while (temp!=NULL) {

        if (temp->key==val) return true;

        temp = temp->next;

    }

    return false;

}

int main() {

    int sizeOfArray = 10;

    int values[sizeOfArray] = {5, 13, 2, 44, 100, 15, 33, 66, 88, 99};

    cout << "Array: "; printList(values, sizeOfArray);

    int hs = 5;

    struct HNode \*\*ht = openHashing(hs, values, sizeOfArray);

    cout << endl << "HS: " << hs << endl; printHash(ht, hs);

    cout << endl << "searching" << endl;

    cout << exists(ht, hs, 44) << endl;

    return 1;

}

2. CLOSED HASHING

#include <iostream>

using namespace std;

bool isPrime(int n) {

    // Corner cases

    if (n <= 1)  return false;

    if (n <= 3)  return true;

    // This is checked so that we can skip

    // middle five numbers in below loop

    if (n%2 == 0 || n%3 == 0) return false;

    for (int i=5; i\*i<=n; i=i+6)

        if (n%i == 0 || n%(i+2) == 0)

           return false;

    return true;

}

int nextPrime(int N) {

    // Base case

    if (N <= 1)

        return 2;

    int prime = N;

    bool found = false;

    // Loop continuously until isPrime returns

    // true for a number greater than n

    while (!found) {

        prime++;

        if (isPrime(prime))

            found = true;

    }

    return prime;

}

void printList(int\* list, int len) {

    cout << "[";

    for (int i=0; i<len; i++) {

        cout << list[i] << ", ";

    }

    cout << "]" << endl;

}

int\* closedHashing(int &HS, int \*values, int totalValues) {

    int \*HT = (int\*) malloc(HS \* sizeof(int));

    for (int i=0; i<HS; i++) {

        HT[i] = -1;

    }

    int no\_of\_values = 0;

    float load = 0;

    int HI, key;

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

        if (load >= 0.5) {

            // rehashing

            int val[no\_of\_values];

            for (int k = 0; k < no\_of\_values; k++)

            {

                val[k] = values[k];

            }

            free(HT);

            HS = nextPrime(HS \* 2);

            HT = closedHashing(HS, val, no\_of\_values);

        }

        key = values[i];

        HI = key % HS;

        if (HT[HI] == -1) {

            HT[HI] = key;

        }

        else {

            // linear probing

            int j = HI;

            while (HT[j] != -1) {

                j = (j+1) % HS;

            }

            // quadratic probing

            // int x = 1, j;

            // while (HT[j] != -1) {

            //     j = (HI + (x\*x)) % HS;

            //     x++;

            // }

            HT[j] = key;

        }

        no\_of\_values++;

        load = float(no\_of\_values)/float(HS);

    }

    return HT;

}

bool exists(int\* HT, int HS, int val) {

    int HI = val % HS;

    // linear probing

int i = 1;

    while (HT[HI]!=-1 || i<HS) {

        if (HT[HI]==val) return true;

        HI = (HI++) % HS;

i++;

    }

    // quadratic probing

    // int x = 1, j = HI, i=1;

    // while (HT[j] != -1 || i<HS) {

    //     if (HT[j]==val) return true;

    //     j = (HI + (x\*x)) % HS;

    //     x++;

// i++;

    // }

    return false;

}

int main() {

    int sizeOfArray = 10;

    int values[sizeOfArray] = {5, 13, 2, 44, 100, 15, 33, 66, 88, 99};

    cout << "Array: "; printList(values, sizeOfArray);

    int hs = 10, \*ht;

    ht = closedHashing(hs, values, sizeOfArray);

    cout << endl << "Size of Linear Array: " << hs << endl;

    cout << "Linear Probation: "; printList(ht, hs);

    cout << "Exists" << endl;

    cout << exists(ht, hs, 55555) << endl;

    cout << exists(ht, hs, 13) << endl;

    return 0;

}

***GRAPHS***

*‘graph.txt’* FILE

7,12,1

0,2,1

0,1,2

1,4,10

1,2,3

2,6,4

2,5,8

2,4,2

2,3,2

3,0,4

3,5,5

4,6,6

6,5,1

1. IN & OUT DEGREE

*i. ADJACENCY LIST FUNCTIONS*

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

using namespace std;

// --------ADJACENCY LIST----------

struct AdjNode {

    int vertex, weight;

    AdjNode \*next;

    AdjNode(int ver, int w) {

        this->vertex = ver;

        this->weight = w;

        this->next = NULL;

    }

};

struct AdjList {

    AdjNode \*\*List;

    int vertices, edges, direction;

};

void addNode(AdjNode \*\*List, int v1, int v2, int weight, int direction) {

    AdjNode \*node = new AdjNode(v2, weight);

    if(List[v1]==NULL) {

        List[v1] = node;

    }

    else {

        AdjNode \*temp = List[v1];

        List[v1] = node;

        node->next = temp;

    }

    if (direction==0) {

        node = new AdjNode(v1, weight);

        if(List[v2]==NULL) {

            List[v2] = node;

        }

        else {

            AdjNode \*temp = List[v2];

            List[v2] = node;

            node->next = temp;

        }

    }

}

AdjList adjacencyList(string filename) {

    int vertices, edges, direction;

    ifstream file;

    file.open(filename);

    string graphInfo;

    getline(file, graphInfo);

    stringstream ved(graphInfo);

    string n;

    getline(ved, n, ',');

    vertices = stoi(n);

    getline(ved, n, ',');

    edges = stoi(n);

    getline(ved, n, ',');

    direction = stoi(n);

    AdjList info;

    info.vertices = vertices;

    info.edges = edges;

    info.direction = direction;

    AdjNode \*\*List = (AdjNode\*\*) malloc(vertices\*sizeof(AdjNode\*));

    for (int i=0; i<vertices; i++)

        List[i] = NULL;

    for (int i=0; i<edges; i++) {

        int v1, v2, weight;

        getline(file, graphInfo);

        stringstream data(graphInfo);

        string value;

        getline(data, value, ',');

        v1 = stoi(value);

        getline(data, value, ',');

        v2 = stoi(value);

        getline(data, value, ',');

        weight = stoi(value);

        addNode(List, v1, v2, weight, direction);

    }

    file.close();

    info.List = List;

    return info;

}

int outDegree(AdjNode \*\*List, int vertex) {

    int degree = 0;

    AdjNode \*temp = List[vertex];

    while (temp!=NULL) {

        degree++;

        temp = temp->next;

    }

    return degree;

}

int inDegree(AdjNode \*\*List, int vertex, int vertices) {

    int degree = 0;

    for (int i=0; i<vertices; i++) {

        AdjNode \*temp = List[i];

        while (temp!=NULL) {

            if (temp->vertex==vertex)

                degree++;

            temp = temp->next;

        }

    }

    return degree;

}

void printList(AdjNode \*\*List, int vertices) {

    cout << "Adjacency List:" << endl << endl;

    for (int i=0; i<vertices; i++) {

        cout << "Vertex " << i << ": ";

        AdjNode \*temp = List[i];

        while (temp!=NULL) {

            cout << temp->vertex << "->";

            temp = temp->next;

        }

        cout << endl;

    }

    cout << endl;

}

*ii. ADJACENCY MATRIX FUNCTIONS*

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

using namespace std;

// -------ADJACENCY MATRIX--------

int\*\* squareMatrix(int n) {

    int \*\*matrix = new int\*[n];

    for (int i=0; i<n; i++) {

        matrix[i] = new int[n];

    }

    for (int i=0; i<n; i++) {

        for (int j=0; j<n; j++) {

            matrix[i][j] = 0;

        }

    }

    return matrix;

}

void printMatrix(int \*\*matrix, int rows, int columns) {

    cout << "Adjacency Matrix" << endl << endl;

    cout << "  \t";

    for (int k=0; k<columns; k++) {

        cout << k << "\t";

    }

    cout << endl << endl;

    for (int i=0; i<rows; i++) {

        cout << i << ":\t";

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

            cout << matrix[i][j] << "\t";

        }

        cout << endl;

    }

    cout << endl;

}

int inDegree(int \*\*Matrix, int vertices, int vertex) {

    int degree = 0;

    for (int i=0; i<vertices; i++) {

        if (Matrix[i][vertex] != 0) {

            degree += 1;

        }

    }

    return degree;

}

int outDegree(int \*\*Matrix, int vertices, int vertex) {

    int degree = 0;

    for (int i=0; i<vertices; i++) {

        if (Matrix[vertex][i] != 0) {

            degree += 1;

        }

    }

    return degree;

}

struct AdjMatrix {

    int \*\*Matrix;

    int vertices, edges, direction;

};

AdjMatrix adjacencyMatrix(string filename) {

    int vertices, edges, direction;

    ifstream file;

    file.open(filename);

    string graphInfo;

    getline(file, graphInfo);

    stringstream ved(graphInfo);

    string n;

    getline(ved, n, ',');

    vertices = stoi(n);

    getline(ved, n, ',');

    edges = stoi(n);

    getline(ved, n, ',');

    direction = stoi(n);

    AdjMatrix info;

    info.vertices = vertices;

    info.edges = edges;

    info.direction = direction;

    int \*\*Matrix = squareMatrix(vertices);

    for (int i=0; i<edges; i++) {

        int v1, v2, weight;

        getline(file, graphInfo);

        stringstream data(graphInfo);

        string value;

        getline(data, value, ',');

        v1 = stoi(value);

        getline(data, value, ',');

        v2 = stoi(value);

        getline(data, value, ',');

        weight = stoi(value);

        Matrix[v1][v2] = weight;

        if (direction==0) {

            Matrix[v2][v1] = weight;

        }

    }

    file.close();

    info.Matrix = Matrix;

    return info;

}

*iii. DRIVER CODE*

#include "adjMatFunctions.h"

#include "adjListFunctions.h"

int main() {

    // ----------MATRIX------------

    int vertices, edges, direction;

    AdjMatrix info = adjacencyMatrix("graph.txt");

    int \*\*matrix = info.Matrix;

    vertices = info.vertices;

    edges = info.edges;

    direction = info.direction;

    printMatrix(matrix, vertices, vertices);

    for (int i=0; i<vertices; i++) {

        cout << "Indegree of Vertex " << i << ": " << inDegree(matrix, vertices, i) << endl;

        cout << "Outdegree of Vertex " << i << ": " << outDegree(matrix, vertices, i) << endl;

        cout << endl;

    }

    // --------LIST--------------

    int vertices1, edges1, direction1;

    AdjList info1 = adjacencyList("graph.txt");

    AdjNode \*\*list = info1.List;

    vertices1 = info1.vertices;

    edges1 = info1.edges;

    direction1 = info1.direction;

    printList(list, vertices1);

    for (int i=0; i<vertices; i++) {

        cout << "Indegree of Vertex " << i << ": " << inDegree(list, i, vertices1) << endl;

        cout << "Outdegree of Vertex " << i << ": " << outDegree(list, i) << endl;

        cout << endl;

    }

    return 0;

}

2. TOPOLOGICAL SORT

#include "adjMatFunctions.h"

bool allVisited(bool vertices[], int size) {

    for (int i=0; i<size; i++) {

        if (vertices[i]==false)

            return false;

    }

    return true;

}

int\* topologicalSort(int \*\*matrix, int vertices) {

    int ID[vertices];

    for (int i=0; i<vertices; i++)

        ID[i] = inDegree(matrix, vertices, i);

    bool visited[vertices] = {false};

    int \*topSortList = new int[vertices], k=0;

    bool allVis = false;

    while (!allVis) {

        int minVertex = 0;

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

            // finding minimum value which is not visited

            if ( ID[i]<=ID[minVertex] && visited[i]==false )

                minVertex = i;

        }

        visited[minVertex] = true;

        topSortList[k++] = minVertex;

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

            if (matrix[minVertex][j]!=0)

                ID[j]--;

        }

        allVis = allVisited(visited, vertices);

    }

    return topSortList;

}

int main() {

    int vertices, edges, direction;

    AdjMatrix info = adjacencyMatrix("graph.txt");

    int \*\*matrix = info.Matrix;

    vertices = info.vertices;

    edges = info.edges;

    direction = info.direction;

    int \*sortedVertices = topologicalSort(matrix, vertices);

    cout << "Sorted Vertices w.r.t Independence:" << endl;

    for (int i=0; i<vertices; i++) {

        cout << sortedVertices[i] << " | ";

    }

    cout << endl;

    return 0;

}

3. DIJKSTRA ALGORITHM

#include "adjMatFunctions.h"

struct Vertex {

    Vertex \*previousVertex;

    int id;

    int distance;

    bool visited;

    bool connected;

};

Vertex\* createNewVertex(int id) {

    Vertex \*newVertex = (Vertex\*) malloc(sizeof(Vertex));

    newVertex->previousVertex = NULL;

    newVertex->visited = false;

    newVertex->connected = false;

    newVertex->id = id;

    newVertex->distance = 2147483647;

    return newVertex;

}

bool allVisited(Vertex \*\*vertices , int size) {

    for (int i=0; i<size; i++) {

        if (vertices[i]->visited == false)

            return false;

    }

    return true;

}

int \*connectedVertices(int \*\*Matrix, int numOfVertices, int vertex) {

    int out\_degree = outDegree(Matrix, numOfVertices, vertex);

    int \*connected\_vertices = new int[out\_degree];

    int j=0;

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

        if (Matrix[vertex][i] != 0)

            connected\_vertices[j++] = i;

    return connected\_vertices;

}

Vertex\* nextVertex(Vertex \*\*vertices, int size) {

    Vertex\* next\_vertex = vertices[0];

    int j;

    for (j=1; j<size; j++) {

        if (!vertices[j]->visited) {

            next\_vertex = vertices[j++];

            break;

        }

    }

    if (j==size) return NULL;

    for (int i=j; i<size; i++) {

        if (!vertices[i]->visited) {

            if (vertices[i]->distance < next\_vertex->distance) {

                next\_vertex = vertices[i];

            }

        }

    }

    return next\_vertex;

}

Vertex\*\* dijkstraAlgorithm(int \*\*matrix, int vertices) {

    Vertex \*\*allVertices = (Vertex\*\*) malloc(vertices \* sizeof(Vertex\*));

    for (int i=0; i<vertices; i++)

        allVertices[i] = createNewVertex(i);

    Vertex \*current\_vertex = allVertices[0];

    current\_vertex->distance = 0;

    current\_vertex->connected = true;

    bool visit = false;

    while (!visit) {

        int \*connected\_vertices = connectedVertices(matrix, vertices, current\_vertex->id);

        int num\_of\_connected\_vertices = outDegree(matrix, vertices, current\_vertex->id);

        for (int i=0; i<num\_of\_connected\_vertices; i++) {

            int conVert = connected\_vertices[i];

            Vertex \*connected\_vertex = allVertices[conVert];

            int current\_edge\_weight = matrix[current\_vertex->id][connected\_vertex->id];

            int current\_vertex\_weight = current\_vertex->distance;

            int totalWeight = current\_vertex\_weight + current\_edge\_weight;

            if (!connected\_vertex->connected) {

                connected\_vertex->connected = true;

                connected\_vertex->previousVertex = current\_vertex;

                connected\_vertex->distance = totalWeight;

            }

            else if (totalWeight < connected\_vertex->distance) {

                connected\_vertex->distance = totalWeight;

                connected\_vertex->previousVertex = current\_vertex;

            }

        }

        current\_vertex->visited = true;

        visit = allVisited(allVertices, vertices);

        if (!visit)

            current\_vertex = nextVertex(allVertices, vertices);

    }

    return allVertices;

}

void display(Vertex \*\*vertices, int size) {

    cout << "(Vertex, Distance) <- Previous Vertex" << endl;

    for (int i=0; i<size; i++) {

        Vertex \*current = vertices[i];

        while (current != NULL) {

            cout << "(" << current->id << ", " << current->distance << ") <- ";

            current = current->previousVertex;

        } cout << endl;

    }

}

int main() {

    int vertices, edges, direction;

    AdjMatrix info = adjacencyMatrix("graph.txt");

    int \*\*matrix = info.Matrix;

    vertices = info.vertices;

    edges = info.edges;

    direction = info.direction;

    Vertex \*\*singlePathGraph = dijkstraAlgorithm(matrix, vertices);

    display(singlePathGraph, vertices);

    return 0;

}

4. PRIM’S ALGORITHM

#include "adjMatFunctions.h"

#define infinity 2147483647;

int\*\* primsAlgorithm(int \*\*Matrix, int num\_of\_vertices) {

    int \*\*newMatrix = squareMatrix(num\_of\_vertices);

    int total\_cost = 0;

    bool selected[num\_of\_vertices];

    selected[0] = true;

    int edgeNumber;

    for (edgeNumber=0; edgeNumber < num\_of\_vertices-1; edgeNumber++) {

        int minimum = infinity;

        int x, y;

        for (int i = 0; i < num\_of\_vertices; i++) {

            if (selected[i]) {

                for (int j=0; j<num\_of\_vertices; j++) {

                    if (!selected[j] && Matrix[i][j]!=0) {

                        if (Matrix[i][j]<minimum) {

                            minimum = Matrix[i][j];

                            x = i;

                            y = j;

                        }

                    }

                }

            }

        }

        newMatrix[x][y] = minimum;

        printf("%d - %d : %d\n", x, y, minimum);

        total\_cost += minimum;

        selected[y] = true;

    }

    printf("Total Cost = %d",total\_cost);

    cout << endl;

    return newMatrix;

}

int main() {

    int vertices, edges, direction;

    AdjMatrix info = adjacencyMatrix("graph.txt");

    int \*\*matrix = info.Matrix;

    vertices = info.vertices;

    edges = info.edges;

    direction = info.direction;

    int \*\*newMatrix = primsAlgorithm(matrix, vertices);

    printMatrix(newMatrix, vertices, vertices);

    return 0;

}

5. KRUSKAL’S ALGORITHM

To covert Prim’s Algo into Kruskal’s, just remove the

*“if (selected[i])” condition*  on line 17.