**// Binary tree node class**

template <class Elem>

class BinNodePtr : public BinNode<Elem> {

private:

Elem it; // The node's value

BinNodePtr\* lc; // Pointer to left child

BinNodePtr\* rc; // Pointer to right child

public:

// Two constructors -- with and without initial values

BinNodePtr() { lc = rc = NULL; }

BinNodePtr(Elem e, BinNodePtr\* l =NULL,

BinNodePtr\* r =NULL)

{ it = e; lc = l; rc = r; }

~BinNodePtr() {} // Destructor

Elem& val() { return it; }

void setVal(const Elem& e) { it = e; }

inline BinNode<Elem>\* left() const { return lc; }

void setLeft(BinNode<Elem>\* b) { lc = (BinNodePtr\*)b; } //conversion must be explicit

inline BinNode<Elem>\* right() const { return rc; }

void setRight(BinNode<Elem>\* b) { rc = (BinNodePtr\*)b; }

bool isLeaf() { return (lc == NULL) && (rc == NULL); }

};

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

**//IMPORTANT for BT**

<class Elem>

void preorder(BinNode<Elem>\* subroot)

{

if (subroot == NULL) return; // Empty

visit(subroot); // Perform some action

preorder(subroot->left()); // Even if empty child

preorder(subroot->right());

}

template <class Elem>

void inorder(BinNode<Elem>\* subroot)

{ if (subroot==NULL) return;

inorder(subroot->left());

cout<<subroot->val()<<endl;

inorder(subroot->right());

}

template <class Elem>

void inordernonrec(BinNode<Elem>\* subroot)

{

int size=count(subroot);

AStack<BinNode<Elem>\*> stack(size);

BinNode <Elem>\* current;

current=subroot;

while(current!=NULL || (!stack.length()==0))

{

if(current!=NULL)

{

stack.push(current);

current=current->left();

}

else

{

stack.pop(current);

cout<<current->val()<<" ";

current=current->right();

}

}

}

template <class Elem>

int count(BinNode<Elem>\* subroot)

{

if (subroot == NULL)

return 0; // Nothing to count

return 1 + count(subroot->left())

+ count(subroot->right());

}

template <class Elem>

int sum(BinNode<Elem>\* subroot)

{

if (subroot == NULL)

return 0; // Nothing to count

return subroot->val() + sum(subroot->left())

+ sum(subroot->right());

}

template <class Elem>

int nnulls(BinNode<Elem>\*subroot)

{ if (subroot==NULL)

return 1;

return nnulls(subroot->left())

+ nnulls(subroot->right());

}

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

**//BST IMPLEMENTATION**

#include "dictionary.h"

#include "binnode.h"

// Binary Search Tree implementation for the Dictionary ADT

template <class Key, class Elem, class KEComp, class EEComp>

class BST : public Dictionary<Key, Elem, KEComp, EEComp> {

private:

BinNode<Elem>\* root; // Root of the BST

int nodecount; // Number of nodes in the BST

// Private "helper" functions

void clearhelp(BinNode<Elem>\*);

BinNode<Elem>\* inserthelp(BinNode<Elem>\*, const Elem&);

BinNode<Elem>\* deletemin(BinNode<Elem>\*, BinNode<Elem>\*&);

BinNode<Elem>\* removehelp(BinNode<Elem>\*, const Key&,

BinNode<Elem>\*&);

bool findhelp(BinNode<Elem>\*, const Key&, Elem&) const;

void printhelp(BinNode<Elem>\*, int) const;

public:

BST() { root = NULL; nodecount = 0; } // Constructor

~BST() { clearhelp(root); } // Destructor

void clear()

{ clearhelp(root); root = NULL; nodecount = 0; }

bool insert(const Elem& e) {

root = inserthelp(root, e);

nodecount++;

return true;

}

bool remove(const Key& K, Elem& e) {

BinNode<Elem>\* t = NULL;

root = removehelp(root, K, t);

if (t == NULL) return false; // Nothing found

e = t->val();

nodecount--;

delete t;

return true;

}

bool removeAny(Elem& e) { // Delete min value

if (root == NULL) return false; // Empty tree

BinNode<Elem>\* t;

root = deletemin(root, t);

e = t->val();

delete t;

nodecount--;

return true;

}

bool find(const Key& K, Elem& e) const

{ return findhelp(root, K, e); }

int size() { return nodecount; }

void print() const {

if (root == NULL) cout << "The BST is empty.\n";

else printhelp(root, 0);

}

};

template <class Key, class Elem, class KEComp, class EEComp>

void BST<Key, Elem, KEComp, EEComp>::

clearhelp(BinNode<Elem>\* subroot) {

if (subroot == NULL) return;

clearhelp(subroot->left());

clearhelp(subroot->right());

delete subroot;

}

template <class Key, class Elem, class KEComp, class EEComp>

BinNode<Elem>\* BST<Key, Elem, KEComp, EEComp>::

inserthelp(BinNode<Elem>\* subroot, const Elem& val) {

if (subroot == NULL) // Empty tree: create node

return (new BinNodePtr<Elem>(val, NULL, NULL));

if (EEComp::lt(val, subroot->val())) // Insert on left

subroot->setLeft(inserthelp(subroot->left(), val));

else subroot->setRight(inserthelp(subroot->right(), val));

return subroot; // Return subtree with node inserted

}

template <class Key, class Elem, class KEComp, class EEComp>

BinNode<Elem>\* BST<Key, Elem, KEComp, EEComp>::

deletemin(BinNode<Elem>\* subroot, BinNode<Elem>\*& min) {

if (subroot->left() == NULL) { // Found min

min = subroot;

return subroot->right();

}

else { // Continue left

subroot->setLeft(deletemin(subroot->left(), min));

return subroot;

}

}

template <class Key, class Elem, class KEComp, class EEComp>

BinNode<Elem>\* BST<Key, Elem, KEComp, EEComp>::

removehelp(BinNode<Elem>\* subroot, const Key& K,

BinNode<Elem>\*& t) {

if (subroot == NULL) return NULL; // Val is not in tree

else if (KEComp::lt(K, subroot->val())) // Check left

subroot->setLeft(removehelp(subroot->left(), K, t));

else if (KEComp::gt(K, subroot->val())) // Check right

subroot->setRight(removehelp(subroot->right(), K, t));

else { // Found it: remove it

BinNode<Elem>\* temp;

t = subroot;

if (subroot->left() == NULL) // Only a right child

subroot = subroot->right(); // so point to right

else if (subroot->right() == NULL) // Only a left child

subroot = subroot->left(); // so point to left

else { // Both children are non-empty

subroot->setRight(deletemin(subroot->right(), temp)); //temp:inorder successor

// returns a pointer to subroot->right()

Elem te = subroot->val(); // remember value of node to remove

subroot->setVal(temp->val());

temp->setVal(te);

t = temp;

}

}

return subroot;

}

template <class Key, class Elem, class KEComp, class EEComp>

bool BST<Key, Elem, KEComp, EEComp>:: findhelp(

BinNode<Elem>\* subroot, const Key& K, Elem& e) const {

if (subroot == NULL) return false; // Empty tree

else if (KEComp::lt(K, subroot->val())) // Check left

return findhelp(subroot->left(), K, e);

else if (KEComp::gt(K, subroot->val())) // Check right

return findhelp(subroot->right(), K, e);

else { e = subroot->val(); return true; } // Found it

}

template <class Key, class Elem, class KEComp, class EEComp>

void BST<Key, Elem, KEComp, EEComp>::

printhelp(BinNode<Elem>\* subroot, int level) const {

if (subroot == NULL) return; // Empty tree

printhelp(subroot->left(), level+1); // Do left subtree

for (int i=0; i<level; i++) // Indent to level

cout << " ";

cout << subroot->val() << "\n"; // Print node value

printhelp(subroot->right(), level+1); // Do right subtree

}

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

**// Some definitions for Comparator classes**

class intintCompare {

public:

static bool lt(int x, int y) { return x < y; }

static bool eq(int x, int y) { return x == y; }

static bool gt(int x, int y) { return x > y; }

};

class IntIntCompare {

public:

static bool lt(Int x, Int y) { return x.key() < y.key(); }

static bool eq(Int x, Int y) { return x.key() == y.key(); }

static bool gt(Int x, Int y) { return x.key() > y.key(); }

};

class intIntCompare {

public:

static bool lt(int x, Int y) { return x < y.key(); }

static bool eq(int x, Int y) { return x == y.key(); }

static bool gt(int x, Int y) { return x > y.key(); }

};

class intIntsCompare {

public:

static bool lt(int x, Int\* y) { return x < y->key(); }

static bool eq(int x, Int\* y) { return x == y->key(); }

static bool gt(int x, Int\* y) { return x > y->key(); }

};

class IntsIntsCompare {

public:

static bool lt(Int\* x, Int\* y) { return x->key() < y->key(); }

static bool eq(Int\* x, Int\* y) { return x->key() == y->key(); }

static bool gt(Int\* x, Int\* y) { return x->key() > y->key(); }

};

class CCCompare {

public:

static bool lt(char\* x, char\* y) { return strcmp(x, y) < 0; }

static bool eq(char\* x, char\* y) { return strcmp(x, y) == 0; }

static bool gt(char\* x, char\* y) { return strcmp(x, y) > 0; }

};

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

**// The Dictionary abstract** class. KEComp compares a key

// and an element. EEComp compares two elements.

template <class Key, class Elem, class KEComp, class EEComp>

class Dictionary {

public:

// Reinitialize dictionary

virtual void clear() = 0;

// Insert an element. Return true if insert is

// successful, false otherwise.

virtual bool insert(const Elem&) = 0;

// Remove some element matching Key. Return true if such

// exists, false otherwise. If multiple entries match

// Key, an arbitrary one is removed.

virtual bool remove(const Key&, Elem&) = 0;

// Remove and return an arbitrary element from dictionary.

// Return true if some element is found, false otherwise.

virtual bool removeAny(Elem&) = 0;

// Return a copy of some Elem matching Key. Return true

// if such exists, false otherwise. If multiple elements

// match Key, return an arbitrary one.

virtual bool find(const Key&, Elem&) const = 0;

// Return the number of elements in the dictionary.

virtual int size() = 0;

};

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

**///////////Cost for BST:**

**Traversing: *O* (*n*)**

**Find: *O* (log *n*) (balanced tree) to *O* (*n*) (worse)**

**Delete: *O* (log *n*) (balanced tree) to *O* (*n*)**

**involves finding the node and then replacing it with the node with minimum in the right**

**sub-tree**

**Insert n nodes: *O* (nlog *n*) if nodes arrive randomly**

***O* (*n*2) if nodes arrive in increasing value**

**Ideally, want: a balanced tree.**

**---------------------------------------------------------**

**//BOOK**

#include <time.h> // Used by timing functions

#include <iostream>

using namespace std;

// Utility functions and macros

// Return true iff x is even

inline bool EVEN(int x) { return (x % 2) == 0; }

// Return true iff x is odd

inline bool ODD(int x) { return (x & 1) != 0; }

const int DefaultListSize = 10; // Lists, etc. default size

// Assert: If boolean expression is false, print a message

// and terminate the program

void Assert(bool val, char\* string) {

if (!val) { // Assertion failed -- close the program

cout << "Assertion Failed: " << string << endl;

exit(-1);

}

}

// Random number generator functions

inline void Randomize() // Seed the generator

{ srand(1); }

// Return a random value in range 0 to n-1

inline int Random(int n)

{ return rand() % (n); }

// Swap two elements in a generic array

template<class Elem>

inline void swap(Elem A[], int i, int j) {

Elem temp = A[i];

A[i] = A[j];

A[j] = temp;

}

// Swap two objects passed by reference

template<class Elem>

inline void swap(Elem &e1, Elem &e2) {

Elem temp = e1;

e1 = e2;

e2 = temp;

}

// Timing variables and functions

clock\_t tstart = 0; // Time at beginning of timed section

// Initialize the program timer

void Settime()

{ tstart = clock(); }

// Return the elapsed time since the last call to Settime

double Gettime()

{ return (double)((double)clock() - (double)tstart)/

(double)CLOCKS\_PER\_SEC; }

// Your basic int type as an object.

class Int {

private:

int val;

public:

Int(int input=0) { val = input; }

// The following is for those times when we actually

// need to get a value, rather than compare objects.

int key() const { return val; }

// Overload = to support Int foo = 5 syntax

int operator=(int input) { val = input; }

};

// Let us print out Ints easily

ostream& operator<<(ostream& s, const Int& i)

{ return s << i.key(); }

ostream& operator<<(ostream& s, const Int\* i)

{ return s << i->key(); }

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

**//BIN NODE MAIN**

void main()

{ BinNode<int> \*mytree;

BinNode<int>\* p;

int i;

int item;

cout<<" enter an item to add to the tree:";

cin>>item;

mytree= new BinNodePtr<int>(item);

cout<<" enter an item to add to the tree:";

cin>>item;

p= new BinNodePtr<int>(item);

mytree->setLeft(p);

cout<<" enter an item to add to the tree:";

cin>>item;

p= new BinNodePtr<int>(item);

mytree->setRight(p);

p=mytree;

cout<<" Elements are:\n"

<<p->val()<<" "<<p->left()->val()<<" "

<<p->right()->val()<<endl;

}

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

**//Dictionary MAIN**

#include <iostream>

#include <string>

#include "BST.h"

using namespace std;

class Employee

{

public:

int ID;

string name;

Employee(int i=0, string n="undefined"){ID=i;name=n;}

};

ostream& operator<< (ostream& osObject, const Employee& c)

{osObject<<"("<<c.ID<<", " <<c.name<<")";

return osObject;}

class IDCompare

{

public:

// static functions so they can be used without

// having an instantiated object

static bool lt(const Employee& x, const Employee& y)

{ return x.ID < y.ID; }

static bool gt(const Employee& x,const Employee& y)

{ return x.ID > y.ID; }

static bool eq(const Employee& x, const Employee& y)

{ return x.ID == y.ID; }

};

class NameCompare

{

public:

static bool lt(const Employee& x, const Employee& y)

{ return x.name< y.name; }

static bool gt(const Employee& x, const Employee& y)

{ return x.name> y.name; }

static bool eq(const Employee& x, const Employee& y)

{ return x.name== y.name; }

};

class IDEmpCompare

{

public:

// static functions so they can be used without

// having an instantiated object

static bool lt(int i,Employee& x )

{ return i<x.ID ; }

static bool gt(int i,Employee& x )

{ return i> x.ID; }

static bool eq(int i,Employee& x )

{ return i==x.ID ; }

};

void main()

{Employee e;

BST <int,Employee,IDEmpCompare, IDCompare> EmpDict1;

EmpDict1.insert(Employee(44,"e1"));

EmpDict1.insert(Employee(82,"e2"));

EmpDict1.insert(Employee(33,"e3"));

EmpDict1.insert(Employee(24,"e4"));

EmpDict1.insert(Employee(14,"e5"));

EmpDict1.insert(Employee(35,"e6"));

EmpDict1.insert(Employee(63,"e7"));

EmpDict1.insert(Employee(48,"e8"));

i

cout<<EmpDict1.size()<<endl;

EmpDict1.find(14,e);

EmpDict1.print();

cout<<e.name<<endl;

EmpDict1.remove(33,e);

cout<<e.name<<endl;

cout<<EmpDict1.size()<<endl;

EmpDict1.removeAny(e);

cout<<e.name<<endl;

EmpDict1.print();

cout<<EmpDict1.size()<<endl;

EmpDict1.clear();

cout<<EmpDict1.size()<<endl;

}

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

**//BST MAIN**

{ BinNode<int> \*mytree;

BinNode<int>\* p,\*q;

int number;

cout<<" Enter First number:";

cin>>number;

mytree= new BinNodePtr<int>(number);

cout<<" Enter other numbers, one per line, any character to stop:\n";

while(cin>>number)

{ p=q=mytree;

while((number!=p->val())&&(q!=NULL)){

p=q;

if(number<p->val())

q=p->left();

else

q=p->right();}

if(number<p->val())

p->setLeft(new BinNodePtr<int>(number));

else

p->setRight(new BinNodePtr<int>(number));

}

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

**//BST MAIN 2**

{ BinNode<int> \*mytree;

int number;

cout<<" Enter First number:";

cin>>number;

mytree= new BinNodePtr<int>(number);

cout<<" Enter other numbers, one per line, any character to stop:\n";

while(cin>>number)

{insert(mytree,number);

}

cout<<"Number of nodes:"<<count(mytree)<<endl;

inorder(mytree);

}

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

**//UNION IMPLEMENTATION**

enum Nodetype {leaf, internal};

class VarBinNode

{ // Generic node class

public:

Nodetype mytype; // Store type for node

union

{

struct // internal node

{

VarBinNode\* left; // Left child

VarBinNode\* right; // Right child

Operator opx; // Value. Operator is a char

} intl;

Operand var; // Leaf: Value only. Operand is a pointer to char

};

//Additional member functions for VarBinNode class

}

//Constructor differentiates between leaf and Intl node

**VarBinNode\* temp1;**

**VarBinNode\* temp2;**

**VarBinNode\* root;**

**temp1 = new VarBinNode(“A”); //Leaf Node**

**temp2 = new VarBinNode(“B”); //Leaf Node**

**root = new VarBinNode('+', temp1, temp2); //Internal Node and root**

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

**//Inheritance Implementation – More efficient**

class VarBinNode { // Node abstract base class

public:

virtual bool isLeaf() = 0;

};

class LeafNode : public VarBinNode { // Leaf node

private:

Operand var; // Operand value

public:

LeafNode(const Operand& val) { var = val; } // Constructor

bool isLeaf() { return true; } // Version for LeafNode

Operand value() { return var; } // Return node value

};

class IntlNode : public VarBinNode { // Internal node

private:

VarBinNode\* left; // Left child

VarBinNode\* right; // Right child

Operator opx; // Operator value

public:

IntlNode(const Operator& op, VarBinNode\* l, VarBinNode\* r)

{ opx = op; left = l; right = r; } // Constructor

bool isLeaf() { return false; } // Version for IntlNode

VarBinNode\* leftchild() { return left; } // Left child

VarBinNode\* rightchild() { return right; } // Right child

Operator value() { return opx; } // Value

};

void traverse(VarBinNode \*subroot) { // Preorder traversal

if (subroot == NULL) return; // Nothing to visit

if (subroot->isLeaf()) // Do leaf node

cout << "Leaf: "

<< ((LeafNode \*)subroot)->value() << endl;

else { // Do internal node

cout << "Internal: "

<< ((IntlNode \*)subroot)->value() << endl;

traverse(((IntlNode \*)subroot)->leftchild());

traverse(((IntlNode \*)subroot)->rightchild());

}

}

**////////// Main test routine**

int main()

{

VarBinNode\* temp1;

VarBinNode\* temp2;

VarBinNode\* root;

char \*string1 = "Hello1";

char \*string2 = "Another string";

temp1 = new LeafNode(string1); //explicit distinction

//temp1 = new VarBinNode(string1); //explicit distinction

temp2 = new LeafNode(string2);

root = new IntlNode('+', temp1, temp2); //explicit distinction

traverse(root);

return(0);

}

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

**//Space Overhead**

For a BT with n nodes, total space = n\*(2\*pointer\_space + data\_space).

Overhead = Space required to maintain a data structure = Space not storing data = 2pn.

From the Full Binary Tree Theorem: Half of the pointers are null (due to leafs).

If leaves store only data, then overhead is reduced.

How can we optimize the storage?

1)Eliminate pointers from leaf nodes. Overhead fraction = (n/2)(2p)/[(n/2)(2p) + nd] = ½ (if p = d)

2)Have only leaves store data, then Overhead fraction = (n/2) (2p+d)/[(n/2) (2p+d) + (n/2)d] = (2p+d)/(2p+2d) = ¾ (if p = d)

3)Use variable nodes, with only leaves storing data. Overhead fraction = 2/3, but total space is lower

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

/**////// Max-heap class**

template <class Elem, class Comp> class maxheap {

private:

Elem\* Heap; // Pointer to the heap array

int size; // Maximum size of the heap

int n; // Number of elements now in the heap

void siftdown(int); // Put element in its correct place

public:

maxheap(Elem\* h, int num, int max) // Constructor

{ Heap = h; n = num; size = max; buildHeap(); }

int heapsize() const // Return current heap size

{ return n; }

bool isLeaf(int pos) const // TRUE if pos is a leaf

{ return (pos >= n/2) && (pos < n); }

int leftchild(int pos) const

{ return 2\*pos + 1; } // Return leftchild position

int rightchild(int pos) const

{ return 2\*pos + 2; } // Return rightchild position

int parent(int pos) const // Return parent position

{ return (pos-1)/2; }

bool insert(const Elem&); // Insert value into heap

bool removemax(Elem&); // Remove maximum value

bool remove(int, Elem&); // Remove from given position

void buildHeap() // Heapify contents of Heap

{ for (int i=n/2-1; i>=0; i--) siftdown(i); }

};

template <class Elem, class Comp> // Utility function

void maxheap<Elem, Comp>::siftdown(int pos) {

while (!isLeaf(pos)) { // Stop if pos is a leaf

int j = leftchild(pos); int rc = rightchild(pos);

if ((rc < n) && Comp::lt(Heap[j], Heap[rc]))

j = rc; // Set j to greater child's value

if (!Comp::lt(Heap[pos], Heap[j])) return; // Done

swap(Heap, pos, j);

pos = j; // Move down

}

}

template <class Elem, class Comp> // Insert element

bool maxheap<Elem, Comp>::insert(const Elem& val) {

if (n >= size) return false; // Heap is full

int curr = n++;

Heap[curr] = val; // Start at end of heap

// Now sift up until curr's parent > curr

while ((curr!=0) &&

(Comp::gt(Heap[curr], Heap[parent(curr)]))) {

swap(Heap, curr, parent(curr));

curr = parent(curr);

}

return true;

}

template <class Elem, class Comp> // Remove max value

bool maxheap<Elem, Comp>::removemax(Elem& it) {

if (n == 0) return false; // Heap is empty

swap(Heap, 0, --n); // Swap max with last value

if (n != 0) siftdown(0); // Siftdown new root val

it = Heap[n]; // Return deleted value

return true;

}

// Remove value at specified position

template <class Elem, class Comp>

bool maxheap<Elem, Comp>::remove(int pos, Elem& it) {

if ((pos < 0) || (pos >= n)) return false; // Bad pos

swap(Heap, pos, --n); // Swap with last value

while ((pos != 0) &&

(Comp::gt(Heap[pos], Heap[parent(pos)]))){

cout << "swapping" << Heap[pos] << "with" << Heap[parent(pos)] << endl;

swap(Heap, pos, parent(pos)); // Push up if large key

}

siftdown(pos); // Push down if small key

it = Heap[n];

return true;

}

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

**/////MAXHEAP MAIN**

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

int i, j;

int n;

Int\* A[20];

Int\* B[20];

Int C[10] = {73, 6, 57, 88, 60, 34, 83, 72, 48, 85};

maxheap<Int\*, IntsIntsCompare> BH(B, 0, 20);

maxheap<Int, IntIntCompare> Test(C, 10, 10);

}

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

**//Array implementation of a complete CBT**

For a node at index r, index for relatives can be found:

Parent (*r*) = **⎣(r-1)/2⎦** if r ≠ 0

Leftchild(*r*) = **2r+1** if 2r+1 < n

Rightchild(*r*) = **2r+2** if 2r+2 < n

Leftsibling(*r*) = **r-1** if r is even

Rightsibling(*r*) = **r+1** if r is odd and r+1 **<** n

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

**//Key Heaps Operations**

**Insert** an element: Append (Put it at the end) and sift up to preserve Heap property

**Remove the maximum**: Interchange root with last element, reduce size, and siftdown (subtrees must be heaps) to preserve Heap

**Remove** an element: Interchange it with last element, reduce size, siftup, and siftdown.

**Build a heap** from an array: Siftdown all nonleaf nodes starting with the highest numbered ones

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

**//Buildheap Cost**

Cost for heap construction:

n nodes, logn levels - Why?

Level logn: n/2 nodes, 0 op. per node

Level logn -1: n/4 nodes, 1 op. per node

Level logn-2: n/8 nodes, 2 ops per node

…

Last Level : 1 node, logn-1 ops per node

log *n*

∑ (*i* - 1) *n*/2*i* ≈ *n*.

*i*=1

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

**Graph Notations:**

A graph with relatively few edges is called **sparse**

A graph with many edges is called **dense.**

A graph containing all possible edges is called **complete**

A graph with edges directed from one vertex to another is a **directed** graph

A graph with edges not directed is an **undirected** graph

A graph with labels that are associated with its vertices is called a **labeled** graph

A graph with weights assigned to its edges is said to be **weighted**

Two **vertices are adjacent** if they are joined by an edge (also called neighbors).

An edge connecting vertices *u* and *v* is written **(*u*, *v*)**. This edge is said to be **incident** on vertices *u* and *v*.

A sequence of vertices *v*1, *v*2, …, *vn* forms a **path** of length *n*-1 if there is an edge from *vi* to *vi*+1 for 1≤*i*<*n.*

A path is **simple** if all vertices on the path are distinct.

A **cycle** is a path of length of 3 or more that connects some vertex *vi* to itself.

A **cycle is simple** if the path is simple, except for the first and last vertices (which are the same).

An undirected graph is **connected** if there is at least one path from any vertex to another.🡪 e.g. Lebanon Cities

A graph without cycles is called **acyclic**

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

**Space Efficiency:**

Adjacency list contains overhead

due to the use of pointers

For a sparse matrix, adjacency list is more space-efficient

For a dense matrix, adjacency matrix is more space-efficient

**Time Efficiency:**

If adjacent vertices are to be visited in an algorithm, adjacency list is more time-efficient

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

**/////// GRAPH ADT**

class Graph { // Graph abstract class

public:

virtual int n() =0; // # of vertices for whole graph

virtual int e() =0; // # of edges for whole graph

// Return index of first neighbor for a given vertex

virtual int first(int) =0;

// Return index of next neighbor (get vertex1’s neighbor after vertex2).

virtual int next(int, int) =0;

// Store new edge, identified by two vertices, 3rd param is weight

virtual void setEdge(int, int, int) =0;

// Delete edge defined by two vertices

virtual void delEdge(int, int) =0;

// Weight of edge connecting two vertices

virtual int weight(int, int) =0;

// The mark functions are used for traversal (VISITED or not)

virtual int getMark(int) =0;

virtual void setMark(int, int) =0;};

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

**////GRAPH MATRIX IMPLEMENTATION**

// Used by the mark array

#define UNVISITED 0

#define VISITED 1

#include "graph.h"

class Graphm : public Graph { // Implement adjacency matrix

private:

int numVertex, numEdge; // Store number of vertices, edges

int \*\*matrix; // Pointer to adjacency matrix

int \*mark; // Pointer to mark array

public:

Graphm(int numVert) { // Make graph w/ numVert vertices

int i, j;

numVertex = numVert;

numEdge = 0;

mark = new int[numVert]; // Initialize mark array

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

mark[i] = UNVISITED;

matrix = (int\*\*) new int\*[numVertex]; // Make matrix

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

matrix[i] = new int[numVertex];

for (i=0; i< numVertex; i++) // Edges start w/ 0 weight

for (int j=0; j<numVertex; j++) matrix[i][j] = 0;

}

~Graphm() { // Destructor

delete [] mark; // Return dynamically allocated memory

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

delete [] matrix[i];

delete [] matrix;

}

int n() { return numVertex; } // Number of vertices

int e() { return numEdge; } // Number of edges

int first(int v) { // Return v's first neighbor

int i;

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

if (matrix[v][i] != 0) return i;

return i; // Return n if none

}

int next(int v1, int v2) { // Get v1's neighbor after v2

int i;

for(i=v2+1; i<numVertex; i++)

if (matrix[v1][i] != 0) return i;

return i;

}

// Set edge (v1, v2) to wgt

void setEdge(int v1, int v2, int wgt) {

Assert(wgt>0, "Illegal weight value");

if (matrix[v1][v2] == 0) numEdge++;

matrix[v1][v2] = wgt;

}

void delEdge(int v1, int v2) { // Delete edge (v1, v2)

if (matrix[v1][v2] != 0) numEdge--;

matrix[v1][v2] = 0;

}

int weight(int v1, int v2) { return matrix[v1][v2]; }

int getMark(int v) { return mark[v]; }

void setMark(int v, int val) { mark[v] = val; }

};

//#include "graphutil.cpp"

// Functions for creating and printing graphs

#define LINELEN 80

void Gprint(Graph\* G) {

int i, j;

cout << "Number of vertices is " << G->n() << "\n";

cout << "Number of edges is " << G->e() << "\n";

cout << "Matrix is:\n";

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

for(j=0; j<G->n(); j++)

cout << G->weight(i, j) << " ";

cout << "\n";

}

}

char\* getl(char\* buffer, int n, FILE\* fid) {

char\* ptr;

ptr = fgets(buffer, n, fid);

while ((ptr != NULL) && (buffer[0] == '#'))

ptr = fgets(buffer, n, fid);

return ptr;

}

// Create a graph from file fid

template <class GType>

Graph\* createGraph(FILE\* fid) {

char buffer[LINELEN+1]; // Line buffer for reading

bool undirected; // true if graph is undirected, false if directed

int i;

int v1, v2, dist;

if (getl(buffer, LINELEN, fid) == NULL) // Unable to get number of vertices

{ cout << "Unable to read number of vertices\n";

return NULL;

}

Graph\* G = new GType(atoi(buffer));

if (getl(buffer, LINELEN, fid) == NULL) // Unable to get graph type

{ cout << "Unable to read graph type\n";

return NULL ;

}

if (buffer[0] == 'U')

undirected = true;

else if (buffer[0] == 'D')

undirected = false;

else {

cout << "Bad graph type: |" << buffer << "|\n";

return NULL;

}

// Read in edges

while (getl(buffer, LINELEN, fid) != NULL) {

v1 = atoi(buffer);

i = 0;

while (isdigit(buffer[i])) i++;

while (buffer[i] == ' ') i++;

v2 = atoi(&buffer[i]);

while (isdigit(buffer[i])) i++;

if (buffer[i] == ' ') { // There is a distance

while (buffer[i] == ' ') i++;

dist = atoi(&buffer[i]);

}

else dist = 1;

G->setEdge(v1, v2, dist);

if (undirected) // Put in edge in other direction

G->setEdge(v2, v1, dist);

}

return G;

}

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

**////GRAPH LIST IMPLEMENTATION**

// Used by the mark array

#define UNVISITED 0

#define VISITED 1

#include "link.h"#include"llist.h"#include "graph.h"

class Edge {

public:

int vertex, weight;

Edge() { vertex = -1; weight = -1; }

Edge(int v, int w) { vertex = v; weight = w; }

};

// Overload for the Edge << operator

ostream& operator << (ostream& s, Edge e)

{ return(s << "(" << e.vertex << ", " << e.weight << ")"); }

class Graphl : public Graph { // Implement adjacency list

private:

int numVertex, numEdge; // Number of vertices, edges

List<Edge>\*\* vertex; // List headers

int \*mark; // Pointer to mark array

public:

Graphl(int numVert) { // Make graph with numVert vertices

int i, j;

numVertex = numVert; numEdge = 0;

mark = new int[numVert]; // Initialize mark array

for (i=0; i<numVertex; i++) mark[i] = UNVISITED;

// Create and initialize adjacency lists

vertex = (List<Edge>\*\*) new List<Edge>\*[numVertex];

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

vertex[i] = new LList<Edge>();

}

~Graphl() { // Destructor

delete [] mark; // Return dynamically allocated memory

for (int i=0; i<numVertex; i++) delete [] vertex[i];

delete [] vertex;

}

int n() { return numVertex; } // Number of vertices

int e() { return numEdge; } // Number of edges

int first(int v) { // Return first neighbor of v

Edge it;

vertex[v]->setStart();

if (vertex[v]->getValue(it)) return it.vertex;

else return numVertex; // Return n if none

}

int next(int v1, int v2) { // Gete v1's neighbor after v2

Edge it;

vertex[v1]->getValue(it);

if (it.vertex == v2) vertex[v1]->next();

else { // Start from beginning of list

vertex[v1]->setStart();

while (vertex[v1]->getValue(it) && (it.vertex <= v2))

vertex[v1]->next();

}

if (vertex[v1]->getValue(it)) return it.vertex;

else return numVertex; // Return n if none

}

// Set edge (v1, v2) to wgt

void setEdge(int v1, int v2, int wgt) {

Assert(wgt>0, "Illegal weight value");

Edge it(v2, wgt);

Edge curr;

vertex[v1]->getValue(curr);

if (curr.vertex != v2) // If not already there, search

for (vertex[v1]->setStart();

vertex[v1]->getValue(curr); vertex[v1]->next())

if (curr.vertex >= v2) break;

if (curr.vertex == v2) // Clear out the current one

vertex[v1]->remove(curr);

else numEdge++;

vertex[v1]->insert(it);

}

void delEdge(int v1, int v2) { // Delete edge (v1, v2)

Edge curr;

vertex[v1]->getValue(curr);

if (curr.vertex != v2) // If not already there, search

for (vertex[v1]->setStart();

vertex[v1]->getValue(curr); vertex[v1]->next())

if (curr.vertex >= v2) break;

if (curr.vertex == v2) { // If not, then there is none

vertex[v1]->remove(curr);

numEdge--;

}

}

int weight(int v1, int v2) { // Return weight of (v1, v2)

Edge curr;

vertex[v1]->getValue(curr);

if (curr.vertex != v2) // If not already there, search

for (vertex[v1]->setStart();

vertex[v1]->getValue(curr); vertex[v1]->next())

if (curr.vertex >= v2) break;

if (curr.vertex == v2)

return curr.weight;

else

return 0; // No such edge

}

int getMark(int v) { return mark[v]; }

void setMark(int v, int val) { mark[v] = val; }

};

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

**//// Graph MAIN**

void main()

{

cout<<"Welcome to the directed graph with weight 1 presentation.\n";

cout<<"Enter first the number of vertices,\nthen the edge connecting them, beginning from 0.\n";

cout<<"An edge to itself is accepted,\na number larger than number of vertices-1 is rejected.\n\n";

myGraph \*G;

G=(myGraph\*)getGraph( );

cout<<"The adjacency matrix is:"<<endl;

Gprint (G);

cout << "\n\n";

G->path();

cout<<endl;

}

Graph \*getGraph( )

{Graph \*G;

int nbOfVertices;

cout<<"Enter the number of vertices"<<endl;

cin>>nbOfVertices;

G=new myGraph(nbOfVertices);

int x , y;

cout<<"Enter the positions of the 2 vertices connected (-1 to end)"<<endl;

cin>>x;

while (x!=-1)

{

cin>>y;

if ( x>=0 && y>=0 && x<nbOfVertices && y<nbOfVertices)

G->setEdge(x , y , 1);

else

cout<<"invalid input"<<endl;

cout<<"Enter the positions of the 2 vertices connected (-1 to end)"<<endl;

cin>>x;

}

cout<<"\nOUTPUT:\n\n";

return G;

}

void Gprint( Graph \*G )

{

int i, j;

cout << "Number of vertices is " << G->n() << "\n";

cout << "Number of edges is " << G->e() << "\n";

cout << "Matrix is:\n";

for (i=0; i<G->n(); i++)

{

for(j=0; j<G->n(); j++)

cout << G->weight(i, j) << " ";

cout << "\n";}}

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

**///// GRAPH TRAVERSALS**

To insure visiting all vertices:

void graphTraverse(const Graph\* G) {

for (v=0; v < G->n(); v++)

G->setMark(v, UNVISITED); //Initialize mark array

for (v=0; v <G-> n(); v++)

if (G->getMark(v) == UNVISITED)

// doTraverse is implemented using one of many

// well-known algorithms

doTraverse(G, v); // Traverse the graph starting from v, and ses the mark bits to VISITED }

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

**/////////////DFS**

void DFS(Graph\* G, int v) {

PreVisit(G, v);

G->setMark(v, VISITED);

for (int w=G->first(v); w<G->n();

w = G->next(v,w))

if (G->getMark(w) == UNVISITED)

DFS(G, w);

PostVisit(G, v);

}

DFS( A): pre(A), A(visited), Row of A: C(not visited)⇒

DFS(C): pre(C), C(visited), Row of C: A(visited), B(not visited) ⇒

DFS(B): pre(B), B(visited), Row of B: C(visited), F(not visited) ⇒

DFS(F): pre(F), F(visited), Row of F: B(visited), C(visited), D(not visited) ⇒

DFS(D): pre(D), D(visited), Row of D: C(visited), F(visited), █ ⇒

post(D), return from this recursive call (back to row F)

E(not visisted) ⇒

DFS(E): pre(E), E(visited), Row of E: A(visited), F(visited), █ ⇒

post(E), return from this recursive call (back to row F)

█ ⇒ post(F), return from this recursive call (back to row B)

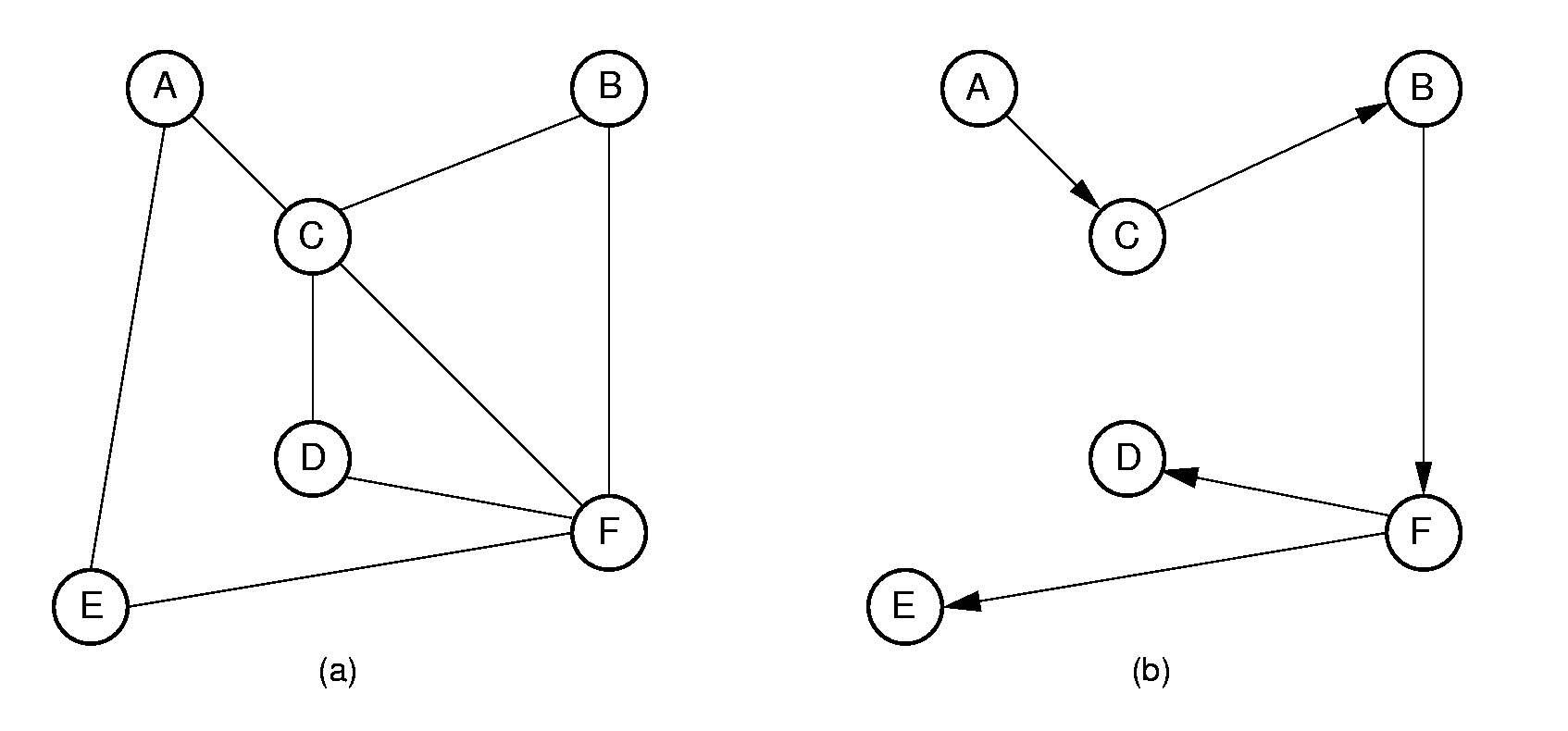
█ ⇒ post(B), return from this recursive call (back to row C)

D(visited), F(visited), █ ⇒ post(C), return from this recursive call (back to row A)

E(visited), █ ⇒ post(A), return from this recursive call

(**End of Function**)

**Cost V+E**

****

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

**//////BFS**

void BFS(Graph\* G, int start, Queue<int>\*Q) {

int v, w;

Q->enqueue(start);

G->setMark(start, VISITED);

while (Q->length() != 0) {

Q->dequeue(v);

PreVisit(G, v);

for(w=G->first(v);w<G->n();w=G->next(v,w))

if (G->getMark(w) == UNVISITED) {

G->setMark(w, VISITED);

Q->enqueue(w);

}

PostVisit(G, v);

}

}

BFS(A):

Q=[A], A(visited),

Q= [ ], pre(A), *Row of A*:

C(not visited) ⇒C(visited), Q=[C], E(not visited) ⇒E(visited), Q=[E,C], post(A)

Q= [E], pre(C), *Row of C*:

A(visited), B(not visited) ⇒B(visited), Q=[B,E], D(not visited) ⇒D(visited), Q=[D,B,E],

F(not visited) ⇒F(visited), Q=[F,D,B,E], post(C)

Q= [F,D,B], pre(E), *Row of E*:

A(visited), F(visited), post(E)

Q= [F,D], pre(B), *Row of B*:

C(visited), F(visited), post(B)

Q= [F], pre(D), *Row of D*:

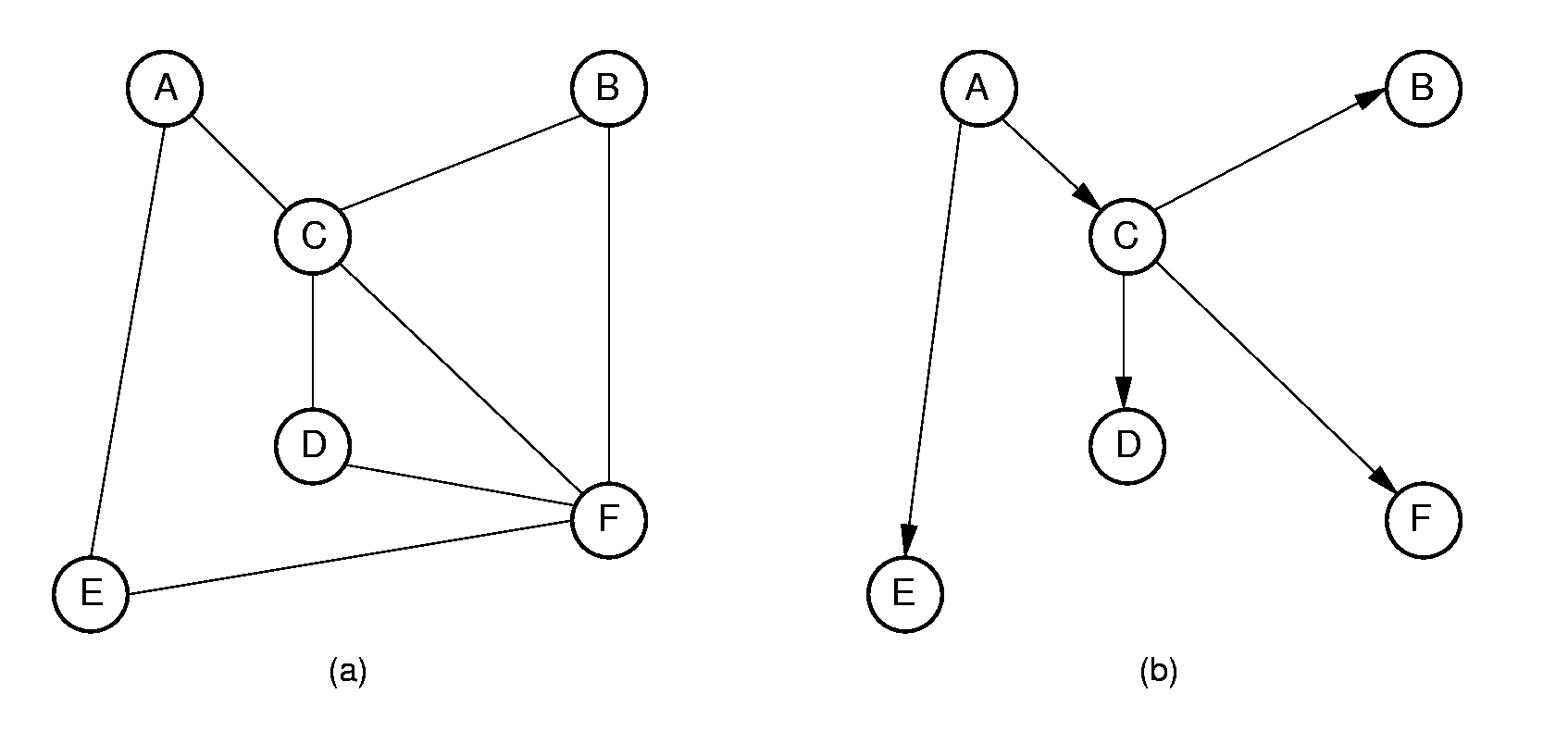
C(visited), F(visited), post(D)

Q= [ ], pre(F), *Row of F*:

B(visited), C(visited), D(visited), E(visited), post(F)

(**End of Function**)

**COST V+E**

****

**---------------------------------------------------------------**

**Topological sort implemented in DFS**

void **topsort**(Graph\* G) { // This particular implementation uses the DFS algorithm

**// Topological sort: *outputs vertices in reverse order***

int i;

for (i=0; i<G->n(); i++) // Initialize vertices to UNVISITED

G->setMark(i, UNVISITED);

// Process vertices: we use a for loop to take care of

// disconnected vertices or sub-graphs

// DFS will visit ALL vertices as long as they are connected

// to the path emanating from the input vertex v

for (i=0; i<G->n(); i++)

if (G->getMark(i) == UNVISITED)

tophelp(G, i); // Call helper function (DFS algorithm)

}

// tophelp implements the **DFS algorithm**

void tophelp(Graph\* G, int v) {

G->setMark(v, VISITED); // note that we took out the pre-visit function

for (int w=G->first(v); w<G->n(); w = G->next(v,w))

if (G->getMark(w) == UNVISITED)

tophelp(G, w);

printout(v); // post-visit for vertex v: executes @ end of each recursion

}

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

**Topological sort implemented in Queue**

void **topsort**(Graph\* G, Queue<int>\* Q) {

int Count[G->n()];

int v, w;

for (v=0; v<G->n(); v++) Count[v] = 0; // Initialize

for (v=0; v<G->n(); v++) // Process edges

for (w=G->first(v); w<G->n(); w = G->next(v,w))

Count[w]++; // Compute the # edges into each vertex

// e.g., count[J4] is incremented when J2 is processed and when J3 is processed

for (v=0; v<G->n(); v++)

if (Count[v] == 0)

Q->enqueue(v); // **Add the vertices that have no prerequisites**

while (Q->length() != 0) {

Q->dequeue(v); // Process each vertex in the Q (has no prereq.)

printout(v); // PreVisit for v

// Since we processed v, treat it as if it did not exist ⇒ **decrement count for**

**// all vertices that are connected to v** (i.e., are in the row for v in the matrix)

for (w=G->first(v); w<G->n(); w = G->next(v,w)) {

Count[w]--; // One less prereq for each vertex connected to v

if (Count[w] == 0) **// Put each free vertex (no prereq.) in Q**

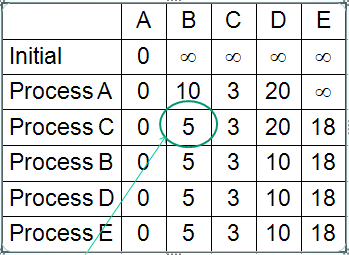
Q->enqueue(w);

}

}

}

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



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

**/////Dijkstra**

**// Compute shortest path distances from s, return them in D**

**// D[j] reflects the distance from vertex s (assume = 0) to vertex j**

**// Initially, D[0]=0 (source), D[i] = ∞ for i>0**

void Dijkstra(Graph\* G, int\* D, int s) {

int i, v, w;

for (i=0; i<G->n(); i++) { // Process the vertices

v = minVertex(G, D);

if (D[v] == INFINITY) return; // Unreachable vertices

G->setMark(v, VISITED);

for (w=G->first(v); w<G->n(); w = G->next(v,w))

if (D[w] > (D[v] + G->weight(v, w)))

D[w] = D[v] + G->weight(v, w);

}

}

int minVertex(Graph\* G, int\* D) { // Find min cost vertex

int i, v;

for (i=0; i<G->n(); i++) // Set v to an unvisited vertex

if (G->getMark(i) == UNVISITED) { v = i; break; }

for (i++; i<G->n(); i++) // Now find smallest D value

if ((G->getMark(i) == UNVISITED) && (D[i] < D[v]))

v = i;

return v;

}

// Test Depth First Search:

// Version for Adjancency Matrix representation

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

Graph\* G;

FILE \*fid;

if (argc != 2) {

cout << "Usage: grdijk1m <file>\n";

exit(-1);

}

if ((fid = fopen(argv[1], "rt")) == NULL) {

cout << "Unable to open file |" << argv[1] << "|\n";

exit(-1);

}

cout << "Now Creating Graph" << "\n";

cin.get();

G = createGraph<Graphm>(fid);

if (G == NULL) {

cout << "Unable to create graph\n";

exit(-1);

}

cout << "Graph Created" << "\n";

cin.get();

int \* D;

D = new int[G->n()];

cout << "Distance array Created, size =" << G->n() << "\n";

cin.get();

// int D[G->n()];

for (int i=0; i<G->n(); i++) // Initialize

D[i] = INFINITY;

D[0] = 0;

cout << "Initial Distances"<< "\n";

for (int i=0; i<G->n(); i++) // Initialize

cout << D[i] << "\n";

Dijkstra(G, D, 0);

for(int k=0; k<G->n(); k++)

cout << D[k] << " ";

cout << endl;

cin.get();

delete D;

return 0;

}

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

**///////////////DIJIKISTRA EXAMPLE STEP BY STEP**

**i=0**; minVertex {v=0; D[j>0]= ∞ ⇒ return 0}

*V0*: visited; consider *V1* and *V2*:

D[1]=∞ > (D[0]+5) ⇒ D[1]=5

D[2]=∞ > (D[0]+10) ⇒ D[2]=10

**i=1**; minVertex {v=1; j=2; D[2]=10, D[v]=5 ⇒ return 1}

*V1*: visited; consider *V3* and *V4*:

D[3]=∞ > (D[1]+7)⇒**D[3]**=5+7**=12**

D[4]=∞ > (D[1]+9)⇒**D[4]**=5+9**=14**

**i=2**; minVertex {v=2; j=3; D[3]<D[2], D[4]<D[2] ⇒ return 2}

*V2*: visited; consider V1, *V3* and *V4*:

D[1] = 5 < (D[2] + 2) ⇒ no change for D[1]

D[3]=12 > (D[2]+1) ⇒ **D[3]**=D[2]+1**=11**

D[4]=14 > (D[2]+3) ⇒ **D[4]**=D[2]+3**=13**

**i=3**; minVertex {v=3; j=4; D[4]<D[3], D[5]<D[3] ⇒ return 3}

*V3*: visited; consider *V4* and *V5*:

D[4]=13 > (D[3]+1) ⇒ **D[4]**=D[3]+1=11+1=**12**

D[5]=∞ > (D[3]+3) ⇒ **D[5]**=D[3]+3=11+3**=14**

**i=4**; minVertex {v=4; j=5; D[5]<D[4] ⇒ return 4}

*V4*: visited; consider *V5*:

D[5]=14 > (D[4]+4)

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