RECURSION HW

// Return false if the somePredicate function returns false for at

// least one of the array elements; return true otherwise.

bool allTrue(const double a[], int n)

{

if(n<=0)

{

return false;

}

if(n==1)

{

return somePredicate(a[0]);

}

return (somePredicate(\*a)&&allTrue(a+1,n-1));

}

// Return the number of elements in the array for which the

// somePredicate function returns false.

int countFalse(const double a[], int n)

{

int count=0;

if (n==0)

{

return 0;

}

if (n==1)

{

if (!somePredicate(a[n-1]))

{

count++;

}

return count;

}

if (!somePredicate(a[n-1]))

{

count++;

}

count+=countFalse(a,n-1);

return count;

}

// Return the subscript of the first element in the array for which

// the somePredicate function returns false. If there is no such

// element, return -1.

int firstFalse(const double a[], int n)

{

if (n==0)

{

return -1;

}

if (n==1)

{

if (!somePredicate(a[0]))

{

return 0;

}

return -1;

}

int check=firstFalse(a, n-1);

if (check==-1)

{

if(!somePredicate(a[n-1]))

{

return n-1;

}

else

{

return check;

}

}

return check;

}

// Return the subscript of the smallest double in the array (i.e.,

// the one whose value is <= the value of all elements). If more

// than one element has the same smallest value, return the smallest

// subscript of such an element. If the array has no elements to

// examine, return -1.

int indexOfMin(const double a[], int n)

{

if (n <= 0)

{

return -1;

}

if (n == 1)

{

return 0;

}

if (a[n - 1] < a[indexOfMin(a, n - 1)])

{

return n - 1;

}

return indexOfMin(a, n - 1);

}

// If all n2 elements of a2 appear in the n1 element array a1, in

// the same order (though not necessarily consecutively), then

// return true; otherwise (i.e., if the array a1 does not include

// a2 as a not-necessarily-contiguous subsequence), return false.

// (Of course, if a2 is empty (i.e., n2 is 0), return true.)

// For example, if a1 is the 7 element array

// 10 50 40 20 50 40 30

// then the function should return true if a2 is

// 50 20 30

// or

// 50 40 40

// and it should return false if a2 is

// 50 30 20

// or

// 10 20 20

bool includes(const double a1[], int n1, const double a2[], int n2)

{

if (n1 <= 0)

{

return false;

}

if (n2<=0)

{

return true;

}

if (a1[0]==a2[0])

{

return includes(a1+1, n1-1, a2+1, n2-1);

}

else

{

return includes(a1+1, n1-1, a2, n2);

}

}

DERIVED/BASE CLASS

#include <iostream>

#include <string>

using namespace std;

class Investment

{

public:

Investment(string name)

:m\_name(name)

{

}

virtual ~Investment()

{

}

virtual bool fungible() const

{

return false;

}

virtual string description() const = 0;

virtual int purchasePrice() const = 0;

string name() const

{

return m\_name;

}

private:

string m\_name;

};

///

class Painting: public Investment

{

public:

Painting(string name, int value)

:Investment(name), m\_value(value)

{

}

virtual bool fungible() const

{

return false;

}

virtual ~Painting()

{

cout << "Destroying " << name() << ", a painting" << endl;

}

virtual string description() const

{

return "painting";

}

virtual int purchasePrice() const

{

return m\_value;

}

private:

int m\_value;

};

///

class Stock: public Investment

{

public:

Stock(string name, int value, string ticker)

:Investment(name), m\_value(value), m\_ticker(ticker)

{

}

virtual bool fungible() const

{

return true;

}

virtual ~Stock()

{

cout << "Destroying " << name() << ", a stock holding" << endl;

}

virtual string description() const

{

string s = "stock trading as ";

s+=m\_ticker;

return s;

}

virtual int purchasePrice() const

{

return m\_value;

}

private:

int m\_value;

string m\_ticker;

};

///

class House: public Investment

{

public:

House(string name, int value)

:Investment(name), m\_value(value)

{

}

virtual bool fungible() const

{

return false;

}

virtual ~House()

{

cout<<"Destroying the house " << name() << endl;

}

virtual string description() const

{

return "house";

}

virtual int purchasePrice() const

{

return m\_value;

}

private:

int m\_value;

};

///

void display(const Investment\* inv)

{

cout << inv->name();

if (inv->fungible())

cout << " (fungible)";

cout << ": " << inv->description() << " bought for $" << inv->purchasePrice() << endl;

}

int main()

{

Investment\* portfolio[4];

portfolio[0] = new Painting("Salvator Mundi", 450300000);

// Stock holdings have a name, value, and ticker symbol

portfolio[1] = new Stock("Alphabet", 100000, "GOOGL");

portfolio[2] = new Stock("Symantec", 50000, "SYMC");

portfolio[3] = new House("4 Privet Drive", 660000);

for (int k = 0; k < 4; k++)

display(portfolio[k]);

// Clean up the investments before exiting

cout << "Cleaning up" << endl;

for (int k = 0; k < 4; k++)

delete portfolio[k];

}

/\*

Output produced:

Salvator Mundi: painting bought for $450300000

Alphabet (fungible): stock trading as GOOGL bought for $100000

Symantec (fungible): stock trading as SYMC bought for $50000

4 Privet Drive: house bought for $660000

Cleaning up

Destroying Salvator Mundi, a painting

Destroying Alphabet, a stock holding

Destroying Symantec, a stock holding

Destroying the house 4 Privet Drive

\*/

RECURSION WITH MAZE

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)

{

if(sr == er && sc == ec)

{

return true;

}

maze[sr][sc] = '$';

//Direction north

if(sr!=0 && maze[sr-1][sc]!='$' && maze[sr-1][sc]!='X')

{

if(pathExists(maze, sr-1, sc, er, ec))

{

return true;

}

}

//Direction east

if(sc!=9 && maze[sr][sc+1]!='$' && maze[sr][sc+1]!='X')

{

if(pathExists(maze, sr, sc+1, er, ec))

{

return true;

}

}

//Direction south

if(sr!=9 && maze[sr+1][sc]!='$' && maze[sr+1][sc]!='X')

{

if(pathExists(maze, sr+1, sc, er, ec))

{

return true;

}

}

//Direction west

if(sc!=0 && maze[sr][sc-1]!='$' && maze[sr][sc-1]!='X')

{

if(pathExists(maze, sr, sc-1, er, ec))

{

return true;

}

}

return false;

}

RECRURSION RANDOM FUNCTIONS

int countIncludes(const double a1[], int n1, const double a2[], int n2)

{

if (n2<=0)

{

return 1;

}

if (n2>0 && n1<=0)

{

return 0;

}

if (a1[n1-1]==a2[n2-1])

{

return countIncludes(a1, n1-1, a2, n2-1) + countIncludes(a1, n1-1, a2, n2);

}

else

{

return countIncludes(a1, n1-1, a2, n2);

}

}

void exchange(double& x, double& y)

{

double t;

t = x;

x = y;

y = t;

}

void split(double a[], int n, double splitter, int& firstNotGreater, int& firstLess)

{

int firstU = 0;

firstLess = n;

firstNotGreater = 0;

if (n<0)

{

n=0;

}

while (firstU<firstLess)

{

if (a[firstU] < splitter)

{

firstLess--;

exchange(a[firstU], a[firstLess]);

}

else

{

if (a[firstU] > splitter)

{

exchange(a[firstNotGreater], a[firstU]);

firstNotGreater++;

}

firstU++;

}

}

}

void order(double a[], int n)

{

if (n<=1)

{

return;

}

int f, fG;

split(a, n, a[n / 2], f, fG);

order(a, f);

order(a + fG, n - fG);

}

DOUBLY LINKED LIST MAP

#include <iostream>

#include "Map.h"

using namespace std;

Map::Map() // new map that is empty (size 0)

{

m\_size = 0;

head = nullptr;

tail = nullptr;

}

bool Map::empty() const

{

if(m\_size == 0)

{

return true;

}

return false;

}

int Map::size() const

{

return m\_size; //# Nodes that map is storing

}

bool Map::insert(const KeyType& key, const ValueType& value)

{

if (contains(key))

{

return false; //if contains key already, then don't do anything

}

Node\* p = new Node;

p->pair.value = value;

p->pair.key = key;

if (empty())

{

head = p;

tail = p;

}

p->previous = tail; //node added to start of map

tail->next = p;

tail = p;

p->next = nullptr;

m\_size++;

return true;

}

bool Map::update(const KeyType& key, const ValueType& value)

{

Node\* p = head;

while (p != nullptr) //iterate through the entire list

{

if (p->pair.key == key) //if such a key exists

{

p->pair.value = value; //update its value

return true;

}

p = p->next;

}

return false;

}

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

{

if (update(key, value))

{

return true;

}

else

{

return (insert(key, value)); //this will always return true

}

}

bool Map::erase(const KeyType& key)

{

if (!contains(key)) //don't do anything if key doesn't exist

{

return false;

}

if (head == nullptr) //if list is empty don't do anything

{

return false;

}

Node\* p = head;

if (m\_size == 1) //if there is one Node

{

m\_size--;

delete head;

head = nullptr;

tail = nullptr;

return true;

}

while (p != nullptr) //if # Nodes greater than 1

{

if (p->pair.key == key)

{

if (p == tail)

{

tail = p->previous;

tail->next = nullptr;

delete p;

}

else if (p == head) //if it reaches the head pointer

{

head = p->next;

head->previous = nullptr; //change prev to nullptr

delete p;

}

else

{

Node\* t2 = p->next;

Node\* t1 = p->previous;

t1->next = t2; //sets to node after

t2->previous = t1; //sets to node before

delete p;

}

m\_size--;

return true;

}

p = p->next; //move to next node in the list

}

return false;

}

bool Map::contains(const KeyType& key) const

{

Node\* p = head;

while (p != nullptr) //iterate through entire list

{

if (p->pair.key == key) //if the key is present return true

{

return true;

}

p = p->next; //move to next Node in the list

}

return false;

/\*

Node\* ptr = keyExists(key);

if (ptr == nullptr)

return false;

return true;

\*/

}

bool Map::get(const KeyType& key, ValueType& value) const

{

Node\* p = head;

while (p != nullptr) //iterate through entire list

{

if (p->pair.key == key)

{

value = p->pair.value; //sets the value to corresponding key

return true;

}

p = p->next; //move to next Node in the list

}

return false; //returns false if no such key exists

}

bool Map::get(int i, KeyType& key, ValueType& value) const

{

Node\* p = head;

if (i >= size() || i < 0) //check if i is in bounds

{

return false;

}

for (int j = 0; j < i; j++) //moves to correct position of the list

{

p = p->next;

}

value = p->pair.value;

key = p->pair.key; //changes respective values of key and value

return true;

}

void Map::swap(Map& other)

{

head = other.head; //switches the head and tail pointers

tail = other.tail;

Node\* tHead = head; //makes temporary pointers for storage

Node\* tTail = tail;

int tSize = size();

other.head = tHead;

other.tail = tTail;

m\_size = other.size(); //switches the respective sizes

other.m\_size = tSize;

}

Map::~Map()

{

Node\* p = head;

while (p != nullptr) //iterate through the array until the end (nullptr)

{

Node\* n = p->next; //another new pointer to store the next value of p

delete p; //deletes every instance of a node

p = n; //original pointer equals node of temp

}

}

Map::Map(const Map &src)

{

head=nullptr;

tail=nullptr;

m\_size=0;

for (int i = 0; i < src.size(); i++)

{

KeyType key;

ValueType val;

//for each value, call the get member function and then insert

src.get(i, key, val);

insert(key, val);

}

}

Map& Map::operator=(const Map &src) //fixes alliasing

{

if (&src == this) // if equal, then do nothing

{

return \*this;

}

Map temp(src); //constructs a temp Map identical to src

swap(temp);

return \*this;

/\*

if (this != &rhs) // Aliasing check

{

Map temp(rhs);

swap(temp); // Might as well use the swap function with the copy constructor

}

return \*this;

\*/

}

bool combine(const Map& m1, const Map& m2, Map& result)

{

Map temp1 = m1;

Map temp2 = m2;

temp1.swap(result); //makes swap have all nodes of m1

bool comb = true;

for (int i = 0; i < temp2.size(); i++)

{

KeyType key;

ValueType val;

temp2.get(i, key, val);

if (!result.contains(key)) //if m1 does not contain node of m2

result.insert(key, val); //add node

else

{

ValueType val2;

result.get(key, val2);

if (val != val2) //if m1 and m2 have same key but different values

{

result.erase(key); //removes from result Map

comb = false; //sets bool comb to false

}

}

}

return comb; //after making result Map, not returns true/false

}

void subtract(const Map& m1, const Map& m2, Map& result)

{

Map temp1 = m1;

Map temp2 = m2;

Map empty;

empty.swap(result); //swaps result with an empty Map

for (int i = 0; i < temp1.size(); i++)

{

KeyType key;

ValueType val;

temp1.get(i, key, val); //gets key and value

if (!temp2.contains(key)) //if m2 does not have key of m1

result.insert(key, val); //insert into result

}

}

#ifndef Map\_h

#define Map\_h

#include <string>

typedef std::string KeyType;

typedef double ValueType;

//typedef int KeyType;

//typedef std::string ValueType;

class Map

{

public:

Map();

bool empty() const;

int size() const;

bool insert(const KeyType& key, const ValueType& value);

bool update(const KeyType& key, const ValueType& value);

bool insertOrUpdate(const KeyType& key, const ValueType& value);

bool erase(const KeyType& key);

bool contains(const KeyType& key) const;

bool get(const KeyType& key, ValueType& value) const;

bool get(int i, KeyType& key, ValueType& value) const;

void swap(Map& other);

~Map();

Map(const Map &src);

Map& operator=(const Map &src);

private:

struct data

{

KeyType key;

ValueType value;

};

struct Node

{

data pair;

Node\* next;

Node\* previous;

};

Node \*head;

Node \*tail;

int m\_size;

};

bool combine(const Map& m1, const Map& m2, Map& result);

void subtract(const Map& m1, const Map& m2, Map& result);

#endif /\* Map\_h \*/

CARMAP TEST

#include "CarMap.h"

#include <iostream>

CarMap::CarMap()

{

}

bool CarMap::addCar(std::string license)

{

if(m\_license.insert(license, 0))

{

return true;

}

return false;

}

double CarMap::gas(std::string license) const

{

double gasCount;

return (m\_license.get(license, gasCount) ? gasCount : -1);

// double gasCount=0;

// if(m\_license.contains(license))

// {

// m\_license.get(license, gasCount);

// return true;

// }

// return -1;

}

//// If a car with the given license plate is in the map, return the

//// number of gallons of gas in its tank; otherwise, return -1.

bool CarMap::addGas(std::string license, double gallons)

{

double current;

if (!m\_license.get(license, current) || gallons < 0)

{

return false;

}

current+=gallons;

return m\_license.update(license, current);

// double current=0;

// if(m\_license.contains(license) && gallons>=0)

// {

// current = (gas(license)+gallons);

// m\_license.update(license, current);

// return true;

// }

// return false;

}

// If no car with the given license plate is in the map or if

// gallons is negative, make no change to the map and return

// false. Otherwise, increase the number of gallons of gas in the

// indicated car by the gallons parameter and return true.

bool CarMap::useGas(std::string license, double gallons)

{

double current;

if (!m\_license.get(license, current) || gallons < 0 || gas(license)<gallons)

return false;

current-=gallons;

return m\_license.update(license, current);

}

// If no car with the given license plate is in the map or if

// gallons is negative or if gallons > gas(), make no change to the

// map and return false. Otherwise, decrease the number of gallons

// of gas in the indicated car by the gallons parameter and return

// true.

int CarMap::fleetSize() const

{

return m\_license.size();

}// Return the number of cars in the CarMap.

void CarMap::print() const

{

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

{

KeyType j;

ValueType k;

m\_license.get(i, j, k);

std::cout<<j<<" "<<k<< std::endl;

}

}

// Write to cout one line for every car in the map. Each line

// consists of the car's license plate, followed by one space,

// followed by the number of gallons in that car's tank. Write

// no other text.

GOOD RECURSION EXAMPLES

An example to print numbers counting down:

void print(int p) { if (p==0) return; cout<<p; print(p-1); return; }

An example to print counting up:

void print(int p) {

if (p==0)

return;

print(p-1);

cout<<p;

return; }

An example to produce the fibonacci number for a given index in the series: int Fibonacci(int n) {

if (n==0)

return 0;

if (n==1)

return 1;

return( Fibonacci(n-2) + Fibonacci(n-1) );

}

A recursive function to determine if an input is prime:

bool isPrime(int p, int i=2) {

if (i\*i>p)

return 1;

if (p%i == 0)

return 0;

return isPrime (p, i+1);

}

// two versions of recursive solution to adding up numbers from 1 to any given number.

// the second example is tail recusion because once the total is found, the function returns and

// does not need to unravel previous recursive steps

int sum (int num) {

if (num==0)

return 0;

return (sum(num-1)+(num)); }

int sum (int num, int total=0) {

if (num<=0)

return total;

sum( num-1, sum );

}

MAZE QUEUE

#include <iostream>

#include <queue>

#include <cassert>

using namespace std;

class Coord

{

public:

Coord(int rr, int cc) : m\_r(rr), m\_c(cc) {}

int r() const { return m\_r; }

int c() const { return m\_c; }

private:

int m\_r;

int m\_c;

};

bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int ec)

{

queue<Coord> coordQueue;

Coord start(sr, sc);

coordQueue.push(start);

maze[sr][sc] = '$';

while (!coordQueue.empty())

{

Coord temp=coordQueue.front();

int r=temp.r();

int c=temp.c();

cout << "(" << r << "," << c << ")" << endl;

coordQueue.pop();

if (r==er && c==ec)

return true;

if (maze[r-1][c] == '.')

{

coordQueue.push(Coord(r - 1, c));

maze[r-1][c] = '$';

}

if (maze[r][c+1] == '.')

{

coordQueue.push(Coord(r, c + 1));

maze[r][c+1] = '$';

}

if (maze[r+1][c] == '.')

{

coordQueue.push(Coord(r + 1, c));

maze[r+1][c] = '$';

}

if (maze[r][c-1] == '.')

{

coordQueue.push(Coord(r, c - 1));

maze[r][c-1] = '$';

}

}

return false;

}

int main()

{

string maze[10] = {

"XXXXXXXXXX",

"X...X..X.X",

"X..XX....X",

"X.X.XXXX.X",

"XXX......X",

"X...X.XX.X",

"X.X.X..X.X",

"X.XXXX.X.X",

"X..X...X.X",

"XXXXXXXXXX"

};

if (pathExists(maze, 10,10, 4,3, 1,8))

cout << "Solvable!" << endl;

else

cout << "Out of luck!" << endl;

}

MAZE STACK

#include <iostream>

#include <stack>

#include <cassert>

using namespace std;

class Coord

{

public:

Coord(int rr, int cc) : m\_r(rr), m\_c(cc) {}

int r() const { return m\_r; }

int c() const { return m\_c; }

private:

int m\_r;

int m\_c;

};

bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int ec)

{

stack<Coord> coordStack;

Coord start(sr, sc);

coordStack.push(start);

maze[sr][sc] = '$';

while (!coordStack.empty())

{

Coord temp=coordStack.top();

int r=temp.r();

int c=temp.c();

coordStack.pop();

if (r==er && c==ec)

return true;

if (maze[r-1][c] == '.')

{

coordStack.push(Coord(r - 1, c));

maze[r-1][c] = '$';

}

if (maze[r][c+1] == '.')

{

coordStack.push(Coord(r, c + 1));

maze[r][c+1] = '$';

}

if (maze[r+1][c] == '.')

{

coordStack.push(Coord(r + 1, c));

maze[r+1][c] = '$';

}

if (maze[r][c-1] == '.')

{

coordStack.push(Coord(r, c - 1));

maze[r][c-1] = '$';

}

}

return false;

}

int main()

{

string maze[10] = {

"XXXXXXXXXX",

"X...X..X.X",

"X..XX....X",

"X.X.XXXX.X",

"XXX......X",

"X...X.XX.X",

"X.X.X..X.X",

"X.XXXX.X.X",

"X..X...X.X",

"XXXXXXXXXX"

};

if (pathExists(maze, 10,10, 4,3, 1,8))

cout << "Solvable!" << endl;

else

cout << "Out of luck!" << endl;

}