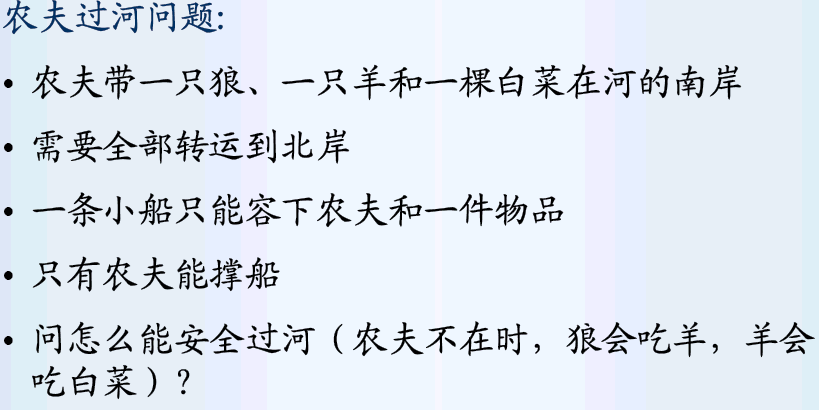
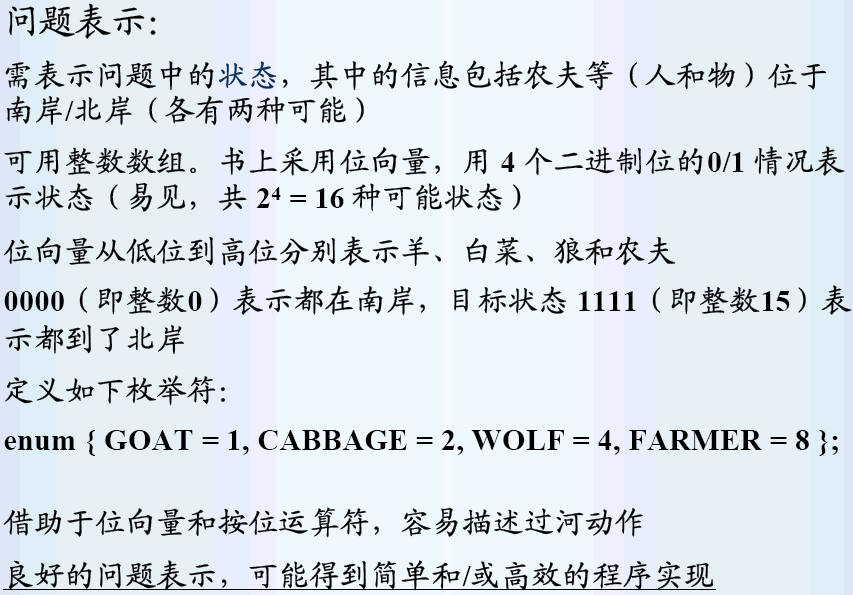
应用队列和位运算相关知识解决农夫过河相关问题

问题描述：



（解决代码使用了广度优先搜索）

#include<iostream>

#include<cstring>

#include<utility>

#include<stack>

using namespace std;

const int MAXSIZE = 100;

template<class ElemType>

class SqQueue {

private:

ElemType\* elem; // 存储空间基址

int front; // 队头指针，注意队头指针和队尾指针都不是指针类型，它只是一个记录对应标号的int型

int rear; // 队尾指针

int maxSize; //注意：maxSize-1才是允许的最大存储容量，以后用这个ADT的时候应当考虑到这一点

public:

//初始化顺序队列

SqQueue(int ms = 20) {

maxSize = ms;

elem = new ElemType[maxSize];

front = rear = 0;

}

//拷贝构造函数

SqQueue(const SqQueue<ElemType>& Q);

//拷贝赋值运算符

SqQueue<ElemType>& operator=(const SqQueue& Q);

//删除顺序队列

~SqQueue() { QueueDestroy(); /\*cout << "调用了析构函数" << endl; \*/ }

//将顺序队列置为空

bool QueueClear();

//设置顺序栈的长度

bool setLength(int len);

//判断顺序队列是否为空

bool isEmpty() const { return front == rear; }

//判断顺序队列是否为满

bool isFull() const;

//返回队列中现有元素的个数

int getLength()const;

//返回队列中最多可以容纳的元素个数

int capacity() const {

return maxSize;

}

//用e返回某一位置的元素:

//注意：不建议在用户调用，只是用于实现内部的拷贝功能时需要用到，否则的话会破坏队列的FIFO的特性

//注意：使用这个函数必须确保队列中的值不是未定义的，否则会引发很严重的后果

bool getElem(int i, ElemType& e)const;

//用e返回队头元素

bool getFront(ElemType& e);

//入队

bool enQueue(const ElemType& e);

//出队

bool deQueue(ElemType& e);

//销毁顺序队列

bool QueueDestroy();

//顺序队列最大存储空间加倍

bool doubleSpace();

//队列遍历函数

void traverse();

//测试用：必须确保每一个操作之后的front和rear都指到正确的位置，否则可能会引发严重后果

/\*int getRear() {

return rear;

}

int getFront() {

return front;

}\*/

};

//拷贝构造函数

template<class ElemType>

SqQueue<ElemType>::SqQueue(const SqQueue<ElemType>& Q) {

maxSize = Q.capacity();

elem = new ElemType[maxSize];

front = rear = 0;

if (!Q.isEmpty()) {

const int L = Q.getLength();

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

ElemType e;

if (Q.getElem(i + 1, e)) {

\*(elem + i) = e;//注意虽然从总体上来说我们用到了循环数组的思想，但是从本质上来说

//数组依然是笔直排列且非循环的，而这里的拷贝只是使得其中的元素以及元素的相对位置一样，

//但是每个两个队列相同元素在各自数组中所处的相对位置确是不同的

//这样拷贝所得的队列的front一定为0，其rear值一定为Q.getLength()

}

}

rear = L;

}

}

//拷贝赋值运算符

template<class ElemType>

SqQueue<ElemType>& SqQueue<ElemType>::operator=(const SqQueue& Q) {

maxSize = Q.capacity();

elem = new ElemType[maxSize];

front = rear = 0;

if (!Q.isEmpty()) {

const int L = Q.getLength();

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

ElemType e;

if (Q.getElem(i + 1, e)) {

\*(elem + i) = e;//注意虽然从总体上来说我们用到了循环数组的思想，但是从本质上来说

//数组依然是笔直排列且非循环的，而这里的拷贝只是使得其中的元素以及元素的相对位置一样，

//但是每个两个队列相同元素在各自数组中所处的相对位置确是不同的

//这样拷贝所得的队列的front一定为0，其rear值一定为Q.getLength()

}

}

rear = L;

}

//cout << "调用了拷贝赋值运算符函数" << endl;

return \*this;

}

//将顺序队列置为空

template<class ElemType>

bool SqQueue<ElemType>::QueueClear() {

front = rear = 0;//实际上只是改变了队头和队尾指针的指向，并没有真正地将队列中不同位置的值删除，要想真正地删除

return true;//必须对值进行一定的覆盖

}

//设置顺序栈的长度

//注意：使用这个函数的时机应当是在队列中还没有元素的时候，我们对其长度进行一个提前规划

template<class ElemType>

bool SqQueue<ElemType>::setLength(int len) {

delete[]elem;

maxSize = len;

elem = new ElemType[maxSize];

front = rear = 0;

if (elem == NULL) return false;

else return true;

}

//判断顺序队列是否为满：（这里采用的策略是少使用顺序表的一个位置去存储元素，使得当rear的后一位为front时即为队列满的标志）

template<class ElemType>

bool SqQueue<ElemType>::isFull() const {

if ((rear + 1) % maxSize == front) {

return true;

}

else {

return false;

}

}

//返回队列中现有元素的个数

template<class ElemType>

int SqQueue<ElemType>::getLength()const {

return (rear - front + maxSize) % maxSize;

}

//用e返回某一位置的元素:

//注意：不建议在用户调用，只是用于实现内部的拷贝功能时需要用到，否则的话会破坏队列的FIFO的特性

//注意：使用这个函数必须确保队列中的值不是未定义的，否则会引发很严重的后果

template<class ElemType>

bool SqQueue<ElemType>::getElem(int i, ElemType& e)const {

if (this->isEmpty()) {

return false;

}

else if (i - 1 == rear || i > this->getLength() || i <= 0) {

return false;

}

else {

e = \*(elem + (front + i - 1) % maxSize);//一定要时刻注意这个循环

return true;

}

}

//用e返回队头元素

template<class ElemType>

bool SqQueue<ElemType>::getFront(ElemType& e) {

if (this->isEmpty()) {

return false;

}

else {

e = \*(elem + front);

return true;

}

}

//入队

template<class ElemType>

bool SqQueue<ElemType>::enQueue(const ElemType& e) {

if (this->isFull()) { return false; }

else {

\*(elem + rear) = e;

rear = (rear + 1) % maxSize;

//cout << "入完队后的front和rear的值为:" << front << ' ' << rear << endl;

return true;

}

}

//出队

template<class ElemType>

bool SqQueue<ElemType>::deQueue(ElemType& e) {

if (this->isEmpty()) {

return false;

}

else {

e = \*(elem + front);

front = (front + 1) % maxSize;

//cout << "出完队后front和rear的值分别为:" << front << ' ' << rear << endl;

return true;

}

}

//销毁顺序队列

template<class ElemType>

bool SqQueue<ElemType>::QueueDestroy() {

if (!elem) {

return false;

}

else {

delete[] elem;

return true;

}

}

//顺序队列最大存储空间加倍，可以将其应用于队列中已经有元素的场合

template<class ElemType>

bool SqQueue<ElemType>::doubleSpace() {

if (!elem) {

return false;

}

else {

int L = this->getLength();

if (L == 0) {

delete[] elem;

maxSize = 2 \* maxSize;

elem = new ElemType[maxSize];

front = rear = 0;

}

else {//表明原来这个队列中有元素，要先对元素进行一个拷贝

ElemType\* tmp = elem;

maxSize = 2 \* maxSize;

elem = new ElemType[maxSize];

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

\*(elem + i) = \*(tmp + front + i);

}

delete[] tmp;//这里一定要记得把原来的空间delete掉

}

if (!elem) return true;

}

}

//队列遍历函数:

template<class ElemType>

void SqQueue<ElemType>::traverse() {

if (this->isEmpty()) { return; }

else {

const int L = this->getLength();

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

cout << \*(elem + (front + i) % maxSize) << ' ';

}

//cout << endl;

//cout << "这个队列的长度为:" << L;

}

}

//初始状态为0000（都在南岸），目标状态为1111（都在北岸），且最多每次只能改变这四个数中的两个数，且每次改变的状态不与

//前面的重复，每一次过一次河后到了对岸就算实现了一个状态，注意这个状态指的是对于每个主体（羊、白菜、狼、农夫）

//此时所在哪个岸的情况（在南岸：0，在北岸：1），故可以知道的是总共有十六种状态，正确的路径就是者十六种状态的一个

//有序子集

//判断此种状态是否会有危险：即在农夫不在场的情况下，狼羊遇到了一起，或者是羊、白菜遇到了一起

//（01x1，10x0），x=0或1

//安全状态：人在的情况下，狼、羊和羊、白菜遇到了一起，或者是狼和白菜在一起，其中没有羊：

//（11x1，00x0，x110，x001），x=0或1

enum {goat=1,cabbage=2,wolf=4,farmer=8};

//求人或物当前的状态的函数（0：南岸，1：北岸）

int Goat(int st) { return (st & goat) != 0; }//思考：位上面这个式子就能正确反映当前的状态，st表示当前的四种事物的

int Cabbage(int st) { return (st & cabbage) != 0; }//所处的一个综合状态（即反映什么东西在什么地方），而我们所定义的enum类型的四个量

int Wolf(int st) { return(st & wolf) != 0; }//其值的二进制表示，恰恰反映了初始状态时的每个事物的位置的反面，而且在表示自己的位上恰好不为0

int Farmer(int st) { return (st & farmer) != 0; }//如farmer=8，二进制表示位1000，每次st与其做位与运算时，后三位必定为0，如果st的二进制表示的

//第一位（总共4位）不为0，那么st&farmer！=0，函数返回1，因为farmer1000初始是在南岸的，而且根据题意，对应位为1时

//就是在北岸

int safe(int st){//安全性判断函数

if (Goat(st) == Wolf(st) && Goat(st) != Farmer(st)) {

return 0;//狼在人不在的情况下吃了羊

}

if (Cabbage(st) == Goat(st) && Farmer(st) != Goat(st)) {

return 0;//羊在人不在的情况下吃了白菜

}

else {

return 1;//安全状态

}

}

//将一个十进制数转为四位二进制数输出

void print(int& num) {//为了准确的用二进制显示四种事物综合的状态

int a, b, c, d;

a = num % 2;

b = num / 2 % 2;

c = num / 2 / 2 % 2;

d = num / 2 / 2 / 2 % 2;

cout << d << c << b << a;

if (num != 0) {

cout << endl;

}

}

//农夫过河问题解决函数：

void farmerProblem() {

SqQueue<int>Q;//用于存放可到达但尚未向前试探的路径

int route[16];

int mover, i, st, st\_new;

//注意st，new\_st表示的是每一个状态对应的四位二进制数，而我们看到的是则是这些四位二进制数所表示的十进制数

//例如：0000（开始）表示0，1001表示9，1111（目的）等于15，因此每一个状态对应的二进制数所呈现出来的十进制数都是

//独一无二的

for (i = 0; i < 16; ++i) {

route[i] = -1;//每一个状态都在初始状态时记录为-1，因为-1永远不可能通过过程中的位运算得到

}

route[0] = 0;//初始状态总是最特殊的，设为0，同时下标也表示了0000的状态，注意：此时下标即为不同状态的

//十进制表示

Q.enQueue(0);

while (!Q.isEmpty()) {

Q.deQueue(st);

for (mover = 1; mover <= 8; mover <<= 1) {//mover为参与运动的四个事物（按次序从前到后为羊、白菜、狼、农夫）

if (Farmer(st) == (0 != (st & mover))) {//是否与农夫同岸，只有同岸才可以移动

//两个十进制数参与位与运算，要将其转换成二进制形式来做，虽然int有四个字节（16位），但是因为这里只有

//人、狼、白菜、羊四种事物，且二进制的每一个位代表其中一个事物的情况，故，我们只关心st二进制表示的后四位

//事实上因为规模太小，st的高位在本题中也绝对是0

st\_new = st ^ (farmer | mover);//过河后的新状态

//farmer|mover的结果的二进制表示无非也就是3种，即：1001，1010，1100，1000，然后采用异或运算

//对应了每过一次河，必改变过河的主体的状态，如果前后状态，而且过河前后参与过河的两种事物的

//状态必须不一样

if (route[st\_new] == -1 && safe(st\_new)) {//状态未到过且安全，记录路径并加入队列

route[st\_new] = st;//每一个数组元素的值都是对应了上一个状态的十进制表示，以这样的方式有利于寻找路径

Q.enQueue(st\_new);

}

}

}

}

i = 15;//从目的出发直至找到起点

while (i != 0) {

print(i);//下标代表的才是不同状态的十进制表示

i = route[i];

}

print(i);

}

int main()

{

farmerProblem();

return 0;

}