数据结构

第五讲 队列

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年10月9日

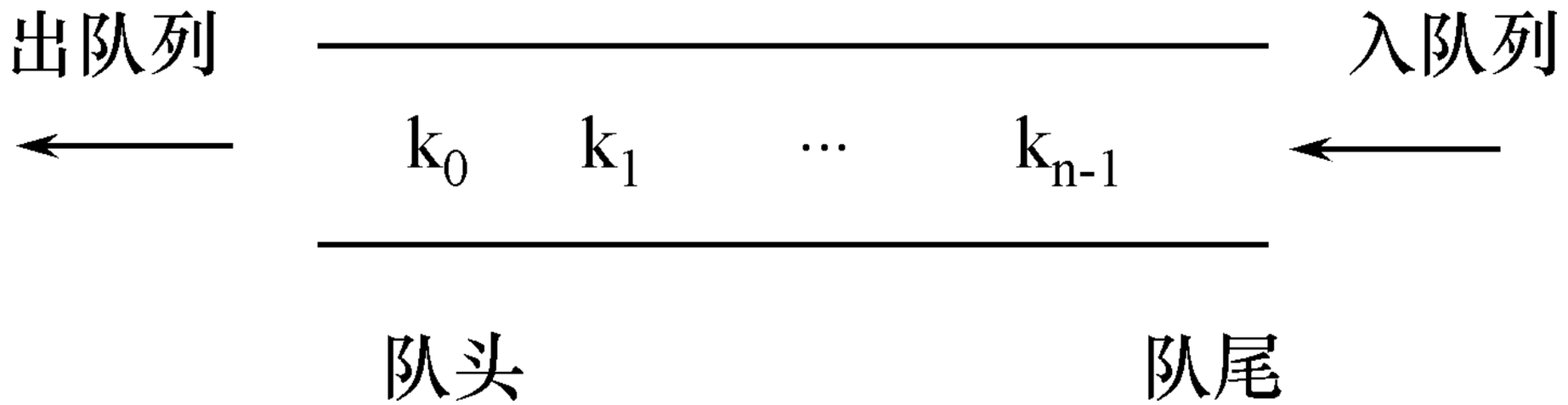
课程内容

- 队列及其抽象数据类型
- 队列的实现
- 队列的应用

队列及其抽象数据类型

- 基本概念
  - 队列是一种只允许在表的一端进行插入操作，而在另一端进行删除操作的线性表。
  - 允许进行删除的这一端叫队列的头，允许进行插入的这一端叫队列的尾。
  - 当队列中没有任何元素时，称为空队列。
  - 队列的插入操作通常称为进队列或入队列，队列的删除操作通常称为退队列或出队列。

先进先出 (FIFO) 表



抽象数据类型

ADT Queue is

operations

Queue createEmptyQueue (void)

创建一个空队列。

int isEmptyQueue (Queue qu)

判断队列qu是否为空队列。

void enQueue (Queue qu, DataType x)

往队列qu尾部插入一个值为x的元素。

void deQueue (Queue qu)

从队列qu头部删除一个元素。

DataType frontQueue (Queue qu)

求队列qu头部元素的值。

end ADT Queue

队列的实现

- 队列实现时通常采用
  - 顺序表方式实现
  - 链接表方式实现

顺序表示

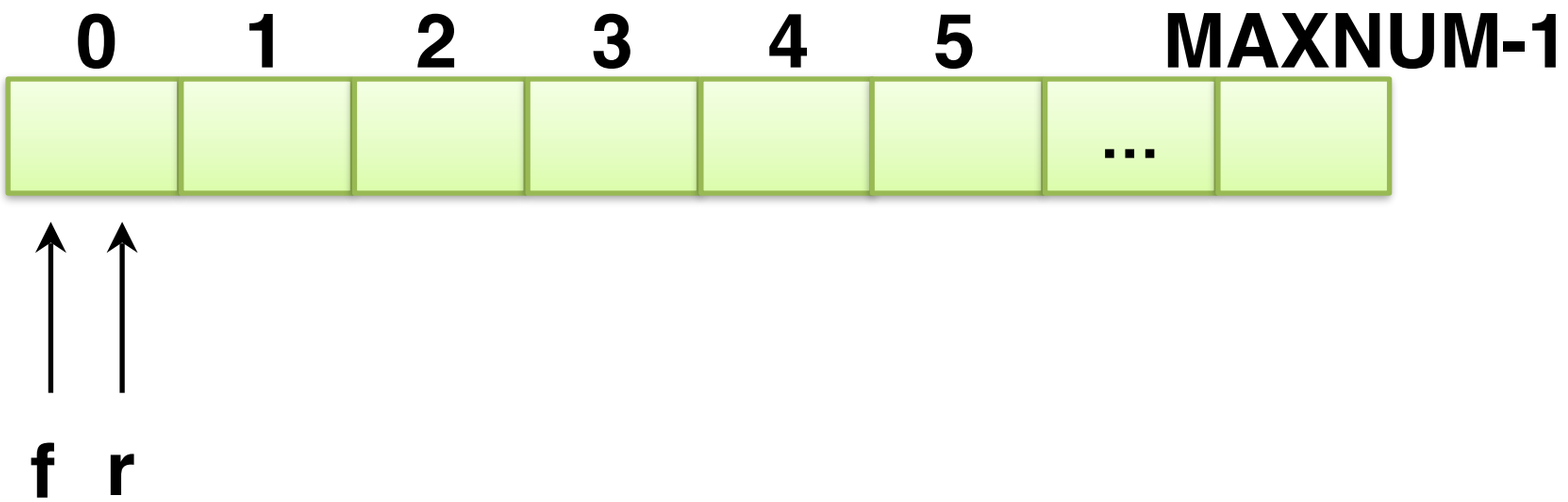
- 存储结构

```
struct  SeqQueue{        /* 顺序队列类型定义 */
    int  MAXNUM;          /* 队列中最大元素个数 */
    int  f,r;
    DataType  *q;
};
/*为了算法设计上的方便：f指出实际队头元素所在的位置，r
  指出实际队尾元素所在位置的下一个位置。*/
typedef  struct SeqQueue  *PSeqQueue;
/* 顺序队列类型的指针类型 */
```

存储结构

- 假设paqu是PSeqQueue类型的变量，
- paqu->f存放即将要被删除的元素的下标，
- paqu->r存放即将要被插入的元素的下标。
- paqu->q[paqu->f]表示当前队列头部的元素；
- paqu->q[paqu->r]表示当前队列尾部的（即将要插入的）元素。
- 初始时paqu->f = paqu->r = 0。
- 当前队列中元素的个数=paqu->r - paqu->f。
- 当paqu->r = paqu->f时，元素的个数为0，即为空队列。

队列动态示意图



空队列

队列的溢出问题

- 与顺序表类示，在队列中，同样由于数组是静态结构，而队列是动态结构，也可能出现队列溢出问题。
- 当队列满时，再作进队操作，这种现象称为上溢；
- 当队空时，作删除操作，这种现象称为下溢。

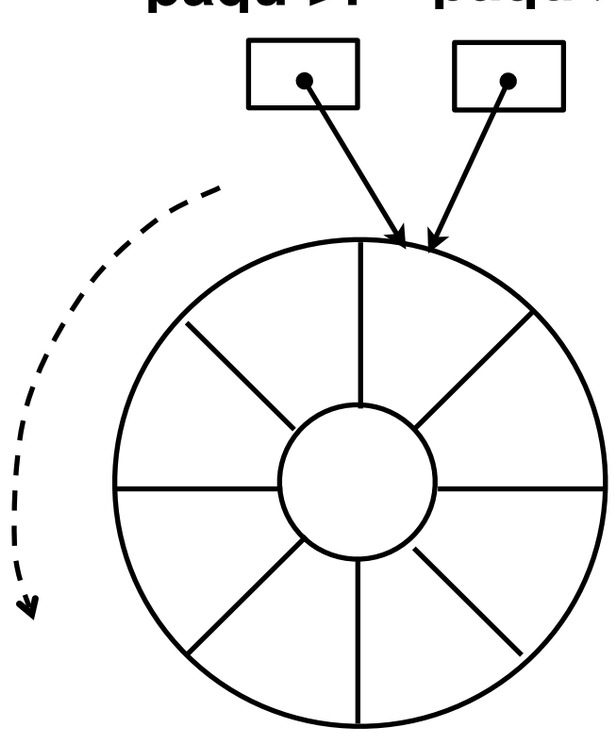
假溢出

- 由于队列中经常要执行插入和删除运算，而每运行一次插入或删除，paqu->r或paqu->f就增加1，使得队列中的元素被删除后，其空间就永远使用不到了。
- 当paqu->r = MAXNUM时，再作插入运算就会产生溢出，而实际上这时队列的前端可能还有许多空的（可用的）位置，因此，这种现象称为假溢出。

环形队列

- 解决假溢出通常采用的方法是环形队列。

“环形队列”（把数组看成环形）



队列空的情况

实现中的不变关系（不变式）：

paqu->r 是最后元素之后空位的下标

paqu->f 是首元素的下标

[paqu->f, paqu->r) 是队列中所有元素（看作按照环形排列）

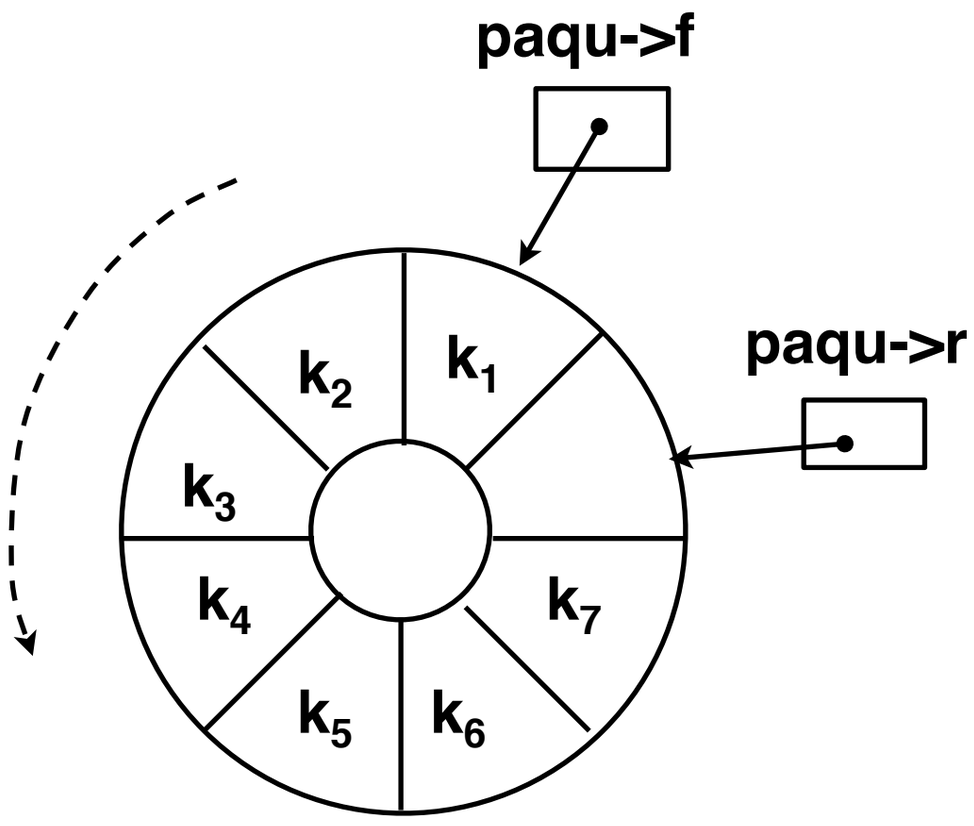
入队时，先存入，后移位

当 paqu->f == paqu->r 时队列空

队列满如何判断?

条件不能与队列空判断相同

环形队列



完全可以采用其他设计，例如：

- 用 f 域记录队头元素位置，num 记录队中元素个数
- 基于这两个变量实现操作，可以不空闲单元

一种方案，牺牲队列中的一个结点，当队列中已有MAXNUM-1个结点时，就称队列已满。再插入就会发生溢出。

基本运算的实现

- 创建一个空队列

  PSeqQueue createEmptyQueue_seq(int m)

- 具体实现与创建空表**PSeqList createNullList_seq(int m)**类似，需要为队列申请空间，不同之处是需要对变量**head**和**rear**均赋值为**0**。请自己给出。

- 判断队列是否为空

  int isEmptyQueue_seq(PSeqQueue paqu)

- 当**paqu->f==paqu->r**时，则返回**1**，否则返回**0**

进队运算

```
void  enQueue_seq( PSeqQueue paqu, DataType x ) {
        /* 在队尾插入元素x */
   if( (paqu->r + 1) % MAXNUM == paqu->f  )
      printf( "Full queue.\n" );
   else {
      paqu->q[paqu->r] = x;
      paqu->r = (paqu->r + 1) % MAXNUM;
   }
}
```

出队运算

```
void  deQueue_seq( PSeqQueue paqu ) {
      /* 删除队列头部元素 */
    if( paqu->f == paqu->r )  printf( "Empty Queue.\n" );
    else  paqu->f = (paqu->f + 1) % MAXNUM;
}
```

取队列头部元素运算

```
DataType  frontQueue_seq( PSeqQueue paqu ) {
   if( paqu->f == paqu->r )  printf( "Empty Queue.\n" );
   else  return (paqu->q[paqu->f]);
}
```

链接表示

- **存储结构：**

  队列的链接表示就是用一个单链表来表示队列，队列中的每个元素对应链表中的一个结点，结点的结构与单链表中结点的结构一样。

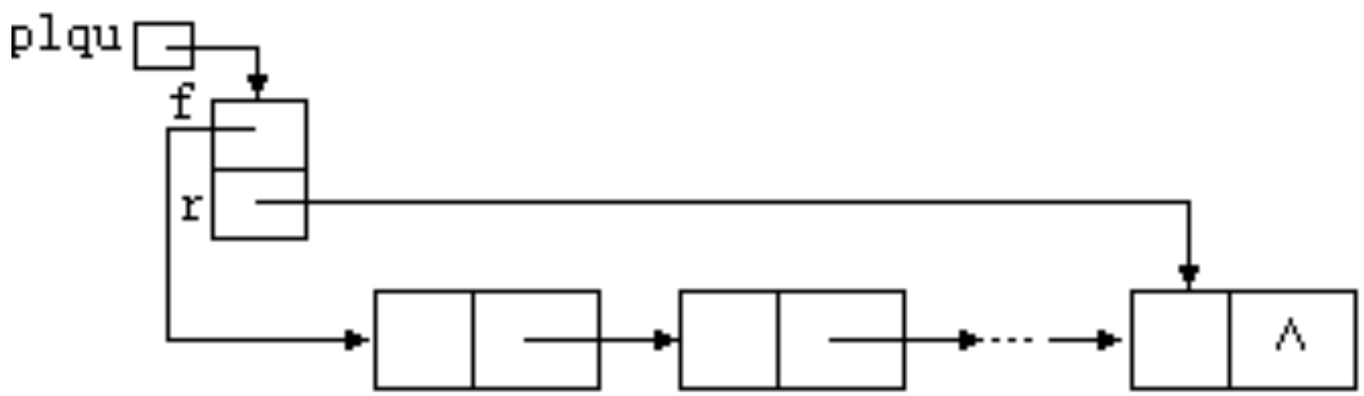
- **为了强调队头和队尾都是队列的属性，这里对队列增加了一层封装，引入LinkQueue结构的定义。这样存储的队列简称链接队列。**

存储结构

```
struct  Node;
typedef  struct  Node  *PNode;
struct  Node {                          /* 结点结构 */
    DataType   info;
    PNode       link;
};
struct  LinkQueue {                     /* 链接队列类型定义 */
    PNode  f;                           /* 头指针 */
    PNode  r;                           /* 尾指针 */
};
typedef  struct LinkQueue  *PLinkQueue;
        /*链接队列类型的指针类型*/
```

存储结构

- 假设plqu是PLinkQueue类型的变量。
- plqu->f为队列的头指针，指向队列中第一个结点。
- plqu->r是队列尾指针，指向队列中最后一个结点。
  （注意：这一点与顺序队列不同！）
- 当plqu->f 或plqu->r为NULL时队列为空。



创建空队列

```
PLinkQueue  createEmptyQueue_link( void ) {
   PLinkQueue plqu;
   plqu = (PLinkQueue )malloc(sizeof(struct LinkQueue));
   if (plqu!=NULL) {
        plqu->f = NULL;
        plqu->r = NULL;
   }
   else printf("Out of space!! \n");
   return plqu ;
}
```

判断队列是否为空、取队头元素

```
int  isEmptyQueue_link( PLinkQueue plqu ) {
   return (plqu->f == NULL);
}
```

```
Datatype  frontQueue_link( PLinkQueue plqu ) {
   if( plqu->f == NULL ) printf("Empty queue.\n");
   else return (plqu->f->info);
}
```

进队运算

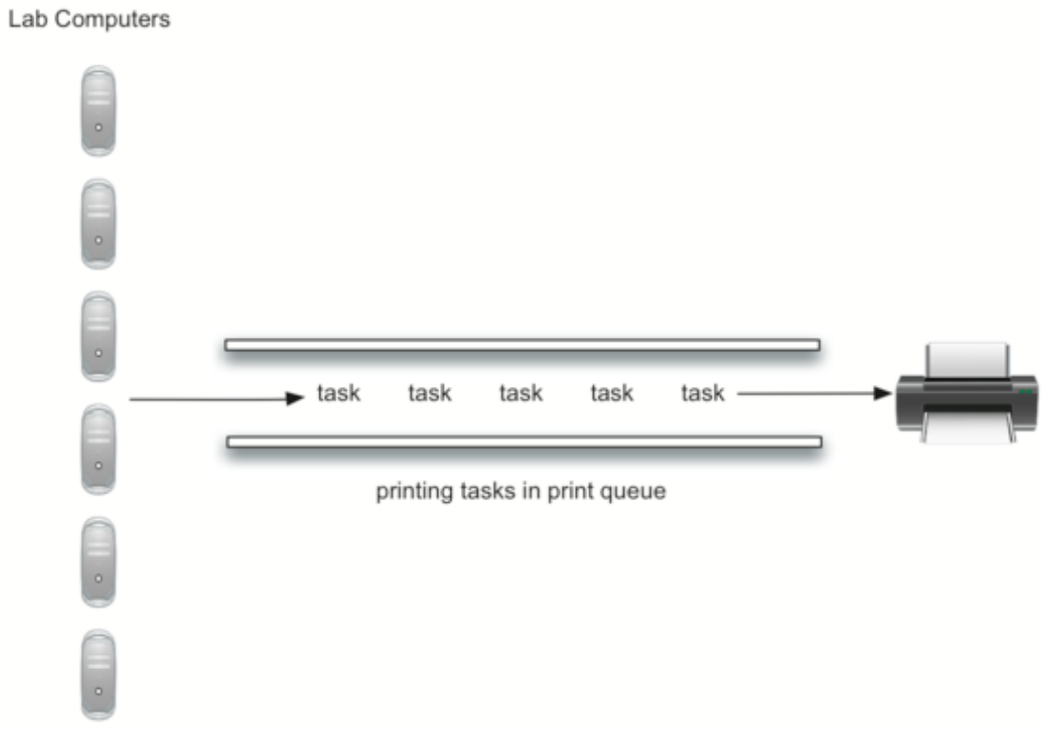
```
void  enQueue_link( PLinkQueue plqu, Datatype x ) {
  PNode  p;
  p = (PNode )malloc( sizeof( struct Node ) );   /*申请新结点空间*/
  if ( p == NULL )printf("Out of space!");        /*申请新结点失败*/
  else {
    p->info = x;                                  /*填写新结点信息*/
    p->link = NULL;
    if (plqu->f == NULL) plqu->f = p;             /*插入前是空队列*/
    else plqu->r->link = p;                       /*将新结点插入*/
    plqu->r = p;                                  /*修改队尾指针*/
  }
}
```

出队运算

```
void  deQueue_link( PLinkQueue plqu ) {
   PNode  p;
   if(plqu->f == NULL)              /*队列已空*/
     printf( "Empty queue.\n " );
   else{
      p = plqu->f;
      plqu->f = p ->link;           /*修改队头指针*/
      free(p);                      /*释放已经删除结点空间*/
   }
}
```

队列的应用

- 队列在各种计算机程序和软件里使用广泛，看几个例子
- 计算机可能连着一台打印机，可以把一些文件送去打印。打印机速度有限，可能出现这样的情况：它正在打印一个文件的过程中，人们又送去一个或几个文件。对多人共享的网络打印机，很可能出现接到了很多打印任务但不能立即处理的情况。
- 打印机管理程序管理着一个缓存打印任务的队列。接到新打印任务时，如果打印机忙，该任务就被放入队列。一旦打印机完成了当前工作，管理程序就查看队列，取出最早的任务送给打印机。



模拟过程

1. 创建一个打印任务队列。队列中每个任务到达的时候会加上一个时间标记，队列初始值为空。
2. 对每一秒（currentSecond）：
   - 是否有新的打印任务出现？如果有，则把该任务加入队列，并将currentSecond作为其时间标记。
   - 若打印机空闲且有打印任务在队列中等待：
     - 将打印任务队列中第一个任务移除，并将其发送给打印机；
     - 用currentSecond减去该任务的时间标记，计算该任务等待时间；
     - 将该任务等待时间放入另一表中，用于后续处理；
     - 根据打印任务的页数，计算该任务需要的时间。
   - 打印机花一定的时间进行打印工作，每打印一秒，从该任务所需要的时间中减去一秒。
   - 若打印任务完成，即该任务需要时间的值达到0，打印机到达空闲状态。
3. 模拟过程完成后，对等待时间表计算平均等待时间。

离散事件模拟

- 离散事件系统模拟是队列的一种常见应用
- 离散事件系统的典型情况
  - 被模拟的（真实世界的）系统的行为表现为一些活动，各种活动进行中又可能产生新的活动。这些活动都需要处理，最简单最常见的处理方式是采用“先发生先处理”的工作原则
  - 由于事件的产生和处理之间存在着速度上的波动和差异，因此系统里可能存在一些已经在等待处理的活动。模拟这种系统，就需要用队列记录正在等待的序列
- 离散事件系统的实例
  - 银行等待服务的顾客、服务席位和服务时间
  - 高速公路收费站通道和服务安排
  - 大楼电梯系统设计和安排
  - 计算机网络中的各种服务系统

离散事件模拟

- 做系统模拟，是希望通过计算机程序的运行模拟真实系统的活动，以理解真实系统运行时的行为，或在未实现系统之前做出一些设计决策。
- 在实现这种模拟系统时，一般而言，需要
  - 通过调查或设计，选择一批模拟参数（例如，顾客抵达的频率）；
  - 引入一些随机因素，反映真实世界中的各种非确定性情况（例如，确定顾客到达约为每2分钟一人，正负1分钟，有随机性）；
  - 用一个或一批队列保存各种待处理活动（例如正等待的顾客）。
- 在生成的活动中保存一些反映真实世界情况的信息（例如，顾客到达的时间，接受服务的开始时间，离开的时间等），以便所实现的模拟系统最后能完成一些统计工作，得到有参考价值的模拟结果（例如：平均等待时间等），用以指导系统设计。

队列的应用

- 考虑网络上的一台 Web 服务器
  - 服务器会不断接到来自网络的网页请求，这时它应该设法找到或做出所需要的页面，发送给提出请求的网络客户
  - 来自网络的请求在速率上会有很大波动，如果瞬时请求很多，服务器就会把来不及处理的请求放入一个队列。一个处理器（或处理线程）完成当时的工作后，就会到队列里取走一个未处理请求
- 计算机里不同处理器或处理进程（线程）之间的通讯也可能需要消息队列作为缓冲（这种方式称为异步通讯）
  - 通讯服务软件把发给一个进程的消息放入该进程的消息队列
  - 进程需要消息时查看自己的消息队列，根据情况取出处理或等待

队列的应用——农夫过河

- 一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸。河中只有一条小船，小船只能容下农夫和一件物品，只有农夫能撑船。
- 请问农夫该采取什么方案才能将所有的东西全部安全运到北岸。

算法选择

- 求解这个问题的最简单的方法是：一步一步进行试探，每一步都搜索当前可能的选择，对当前合适的选择再考虑下一步的各种方案。

状态的表示

- 要模拟农夫过河问题，首先需要选择一个对问题中每个角色的位置进行描述的方法。
- 一个很方便的办法是用四位二进制数顺序分别表示农夫、狼、白菜和羊的位置。
- 例如用0表示农夫或者某东西在河的南岸，1表示在河的北岸。因此整数5(其二进制表示为0101) 表示农夫和白菜在河的南岸，而狼和羊在北岸。

确定每个角色位置的函数

- 用整数location表示上述四位二进制描述的状态，
- 用下面的四个函数从上述状态中得到每个角色所在位置的代码。

```
int farmer(int location) {
    return (0 != (location & 0x08));
}
int wolf(int location) {
    return (0 != (location & 0x04));
}
int cabbage(int location) {
    return (0 != (location & 0x02));
}
int goat(int location) {
    return (0 !=(location & 0x01));
}
```

- 函数返回值为真表示所考察的人或物在河的北岸，否则在南岸。

算法实现中需要的位运算

- 假设x和y都是8位的字符，其值分别是：

  x = 01010111

  y = 11011010。

各种字位运算，得到的结果如下：

- ~x　　10101000(求补)
- x & y　　01010010
- x ^ y　　10001101(按位加/异或)
- x | y　　11011111
- x << 3　　10111000
- y >> 5　　00000110

判断状态是否安全

- 还应该分析问题中所有角色的各种可能位置构成的状态，确定其中哪些是安全的哪些是不安全的。
- 根据原题的描述我们知道，单独留下白菜和羊，或单独留下狼和羊在某一岸的状态是不安全的。
- 由此可以编一个函数，通过位置分布的代码来判断状态是否安全。

```
int safe(int location) {              /* 若状态安全则返回1,否则0. * /
      if ((goat(location) == cabbage(location)) &&
          (goat(location) != farmer(location)))
         return (0);
      if ((goat(location) == wolf(location)) &&
          (goat(location) != farmer(location)))
         return (0);
       return (1);                    /* 其他状态是安全的 * /
}
```

问题描述

- 从初始状态二进制0000(全部在河的南岸) 出发，寻找一种全部由安全状态构成的状态序列，它以二进制1111(全部到达河的北岸) 为最终目标，并且在序列中的每一个状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达。

广度优先

- 在搜索过程中总是首先搜索下面一步的所有可能状态，然后再进一步考虑更后面的各种情况。
- 为避免不必要的代价，要求在序列中不应该出现重复的状态。

农夫、狼、白菜和羊的状态变化

0000→1000
1001 → 0000
1010 0001 →1001
1100 1011 → 0001
0010 →1010
0011 1011
1110 → 0010
0110 → 1110
1111
0100

1101 →0001
0100 →1100
0101 1101
1110 → 0010
0110 → 1110
1111
0100

队列的设计与使用

- 为了实现广度优先搜索，算法中需要使用一个队列moveTo，它的每个元素表示一个可以安全到达的中间状态。
- 程序执行中,把下一步所有可能达到的安全状态都列举出来，放在这个队列中，然后顺序取出来分别进行处理，处理过程中把再下一步的安全状态放在队列里……。
- 由于队列的操作遵循先进先出的原则，在这个处理过程中，只有在前一步的所有情况都处理完后，才能开始后面一步各情况的处理。

记录解的数据结构

- 由于需要列举的所有状态（二进制0000~1111）一共16种，所以构造一个包含16个元素的数组route来记录问题的解。
- route的每个分量初始化值均为-1，每当我们在队列中加入一个新状态时，就把数组中以该状态作下标的元素的值改为达到这个状态的路径上前一状态的（下标）值。
- route的第i个元素记录状态i是否已被访问过，若已被访问过，则在这个数组元素中记入前驱状态值。
- 最后我们可以利用route元素的值建立起正确的状态路径（问题的解）。

农夫过河问题的求解

```
void farmerProblem( ) {
int i, movers, location, newlocation, route[16];
PSeqQueue moveTo;
moveTo=createEmptyQueue_seq( );
enQueue_seq(moveTo,0x00);
for(i=0;i<16;i++)  route[i]=-1;
route[0]=0;
while(!isEmptyQueue_seq(moveTo)&&(route[15]==-1)) {
location=frontQueue_seq(moveTo);
deQueue_seq(moveTo);
for(movers=1;movers<=8;movers<<=1)
   if ((0!=(location & 0x08))==(0!=(location & movers))) {
    newLocation=location^(0x08|movers);
    if(safe(newLocation)&&(route[newLocation]==-1)) {
        route[newLocation]=location;
        enQueue_seq(moveTo,newlocation);        }           }
}
if(route[15]!=-1) {
  printf("The reverse path is : \n");
  for(location=15;location>=0;location=route[location]) {
    printf("The location is : %d\n",location);
    if (location==0)  exit(0);                                    }
}
else printf("No solution.\n");
}
```

执行结果

The reverse path is :

The location is : 15

The location is : 6

The location is : 14

The location is : 2

The location is : 11

The location is : 1

The location is : 9

The location is : 0

队列与栈

- 对队列来说，它的插入运算在表的一端进行，而删除运算却在表的另一端进行。根据队列的这一特点，在使用顺序存储结构时，用了环形队列，这样可解决假溢出问题，但要特别注意队列满和队列空的条件及描述。

- 对于栈来说，它的插入和删除都是在表的同一端进行的，用顺序存储结构时，要注意栈满、栈空的条件。

队列与栈：状态空间搜索

- 一般性认识：对一个需要用计算的方法处理的问题
  - 如有全局性系统性的认识，就可能做出一个专门的算法解决问题
  - 如果只有对问题空间的局部性认识，无法做出直接求解问题的算法，还是有可能将其转化为一个状态空间搜索问题。搜索法是一种通用问题求解方法（general problem solving）
- 搜索过程进展中的情况：
  - 已经探查了从初始状态可达的一些中间状态
  - 已经探查的某些中间状态存在着尚未探查的相邻状态
  - 显然，对任何从初始状态可达的中间状态，与它相邻的状态也是可达的。因此，从一个中间状态向前探查可能得到新的可达状态
  - 若新确定的可达状态是结束状态，就么找到了初始状态到结束状态的路径；否则，新可达状态应加入已探查的中间状态集

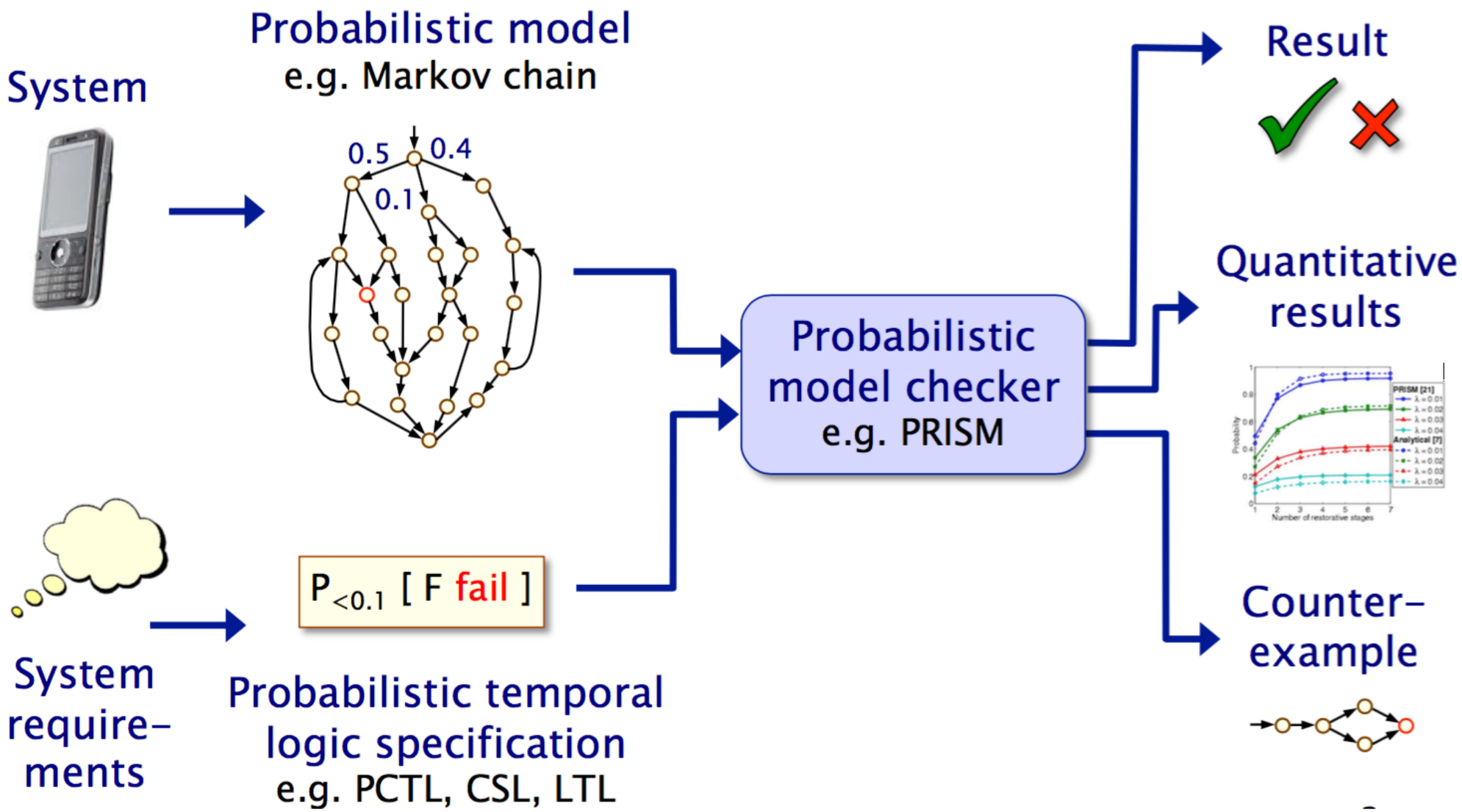
队列与栈：状态空间搜索

- 显然，搜索中需要记录存在未探查邻居的中间状态，以备后面使用。
- 要记录存在尚未完全探索后继状态的中间状态，需要用缓存结构。原则上说栈和队列都可用，但结构的选择将对搜索进展方式产生重大影响。

队列与栈：状态空间搜索

- 上堂课的迷宫算法里用的是栈，栈的特点是后进先出
  - “后进”的状态是在搜索过程中较晚遇到的状态，即是与开始状态距离较远的状态。“后进先出”意味着从最后遇到的状态考虑继续向前探索，尽可能向前检查，尽可能向远处探索
  - 只有后来的状态已经无法继续前进时，才会退到前面最近保存的状态，换一种可能性继续，后退并考虑其他可能性的动作就是回溯
- 如果用队列，队列的特点是先进先出
  - “先进”的状态是在搜索过程中较早遇到的状态，即与开始状态距离较近的状态。“先进先出”要求先考虑距离近的状态，从它们那里向外扩展，实际上是一种从各种可能性“齐头并进”式的搜索
  - 在这种搜索过程中没有回溯，是一种逐步扩张的过程。

队列与栈：状态空间搜索


System
Probabilistic model
e.g. Markov chain
0.5
0.4
0.1
Result
Quantitative results
Probabilistic model checker
e.g. PRISM
PRISM [21]
Analytical [7]
Number of restorative stages
Counter-example
System require-ments
P<0.1 [F fail]
Probabilistic temporal logic specification
e.g. PCTL, CSL, LTL

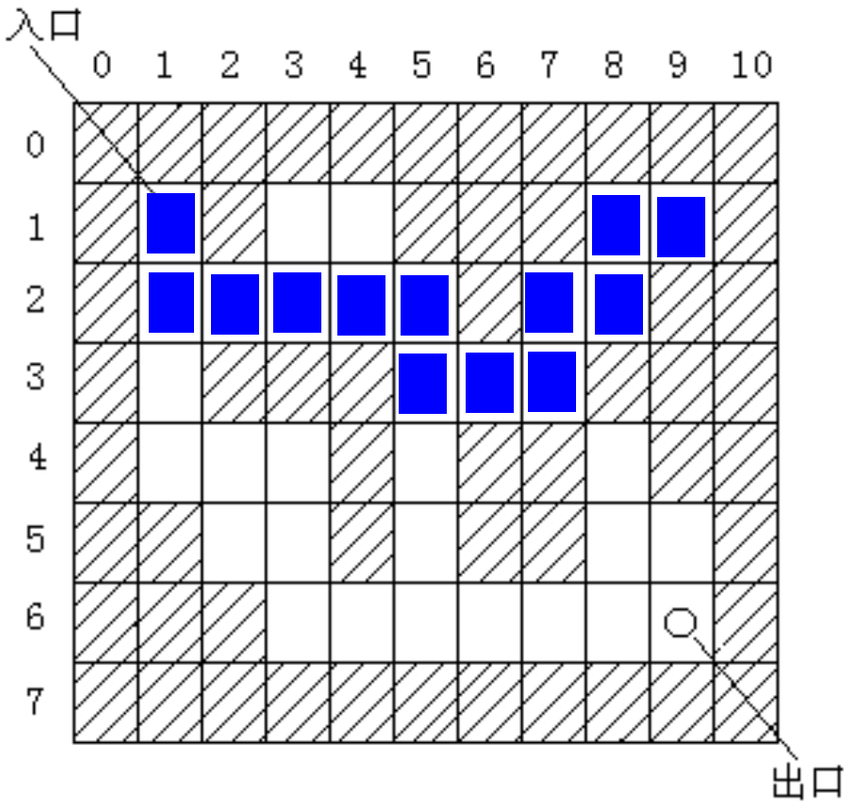
基于队列的迷宫求解

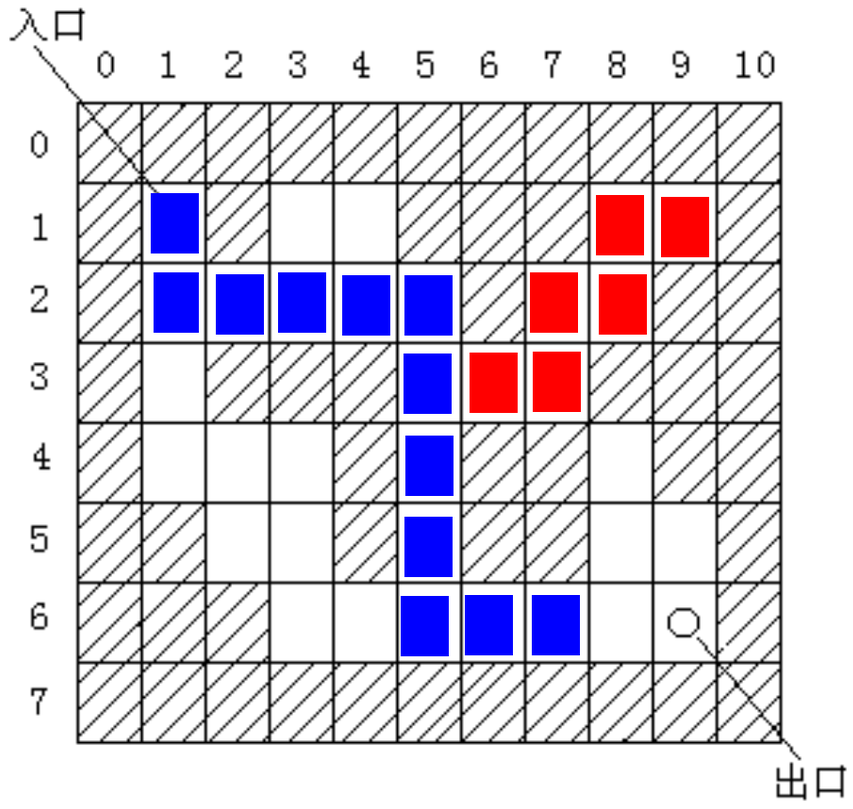
- 要实现一个基于队列的迷宫求解算法，可以重用前面的基本设计
  - 迷宫的矩阵表示和矩阵元素的取值方式
  - 标记函数 mark 和位置检查函数 passable
  - 表示方向的表 direction 和对方向的循环处理方式
- 这里遇到的最重要问题是如何获得所希望的路径
  - 前面的栈算法里，栈里每个点下面是到达它路径上的前一个点
    - 找到出口时，当时的栈里正好保存着到从入口到出口一条路径
  - 对于队列算法，栈中保存的位置点的顺序与路径无关
    - 例如，如果一个点有几个“下一探查点”，它们将顺序进队列
- 结论：在找到出口时，无法基于当时的信息追溯出一条成功路径
  - 要想在算法结束时得到路径，必须在搜索中另行记录有关信息
- 为搜索到达出口时能获得相关路径，遇到新位置时就需要记住其前驱
  - 可以用一个字典记录搜索中得到的这方面信息
  - 到达出口后反向追溯，就可以得到从迷宫入口到出口的路径了

状态空间搜索：栈和队列

- **基于栈搜索迷宫的两个场景：**

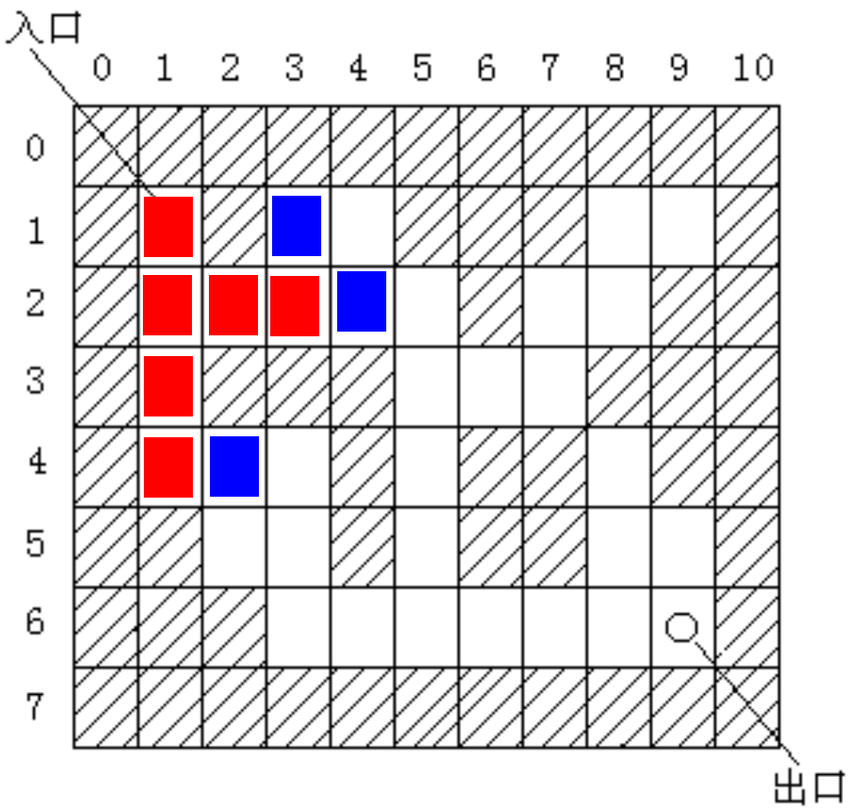
  蓝色表示栈里记录的位置，红色表示已搜索过不会再去检查的位置

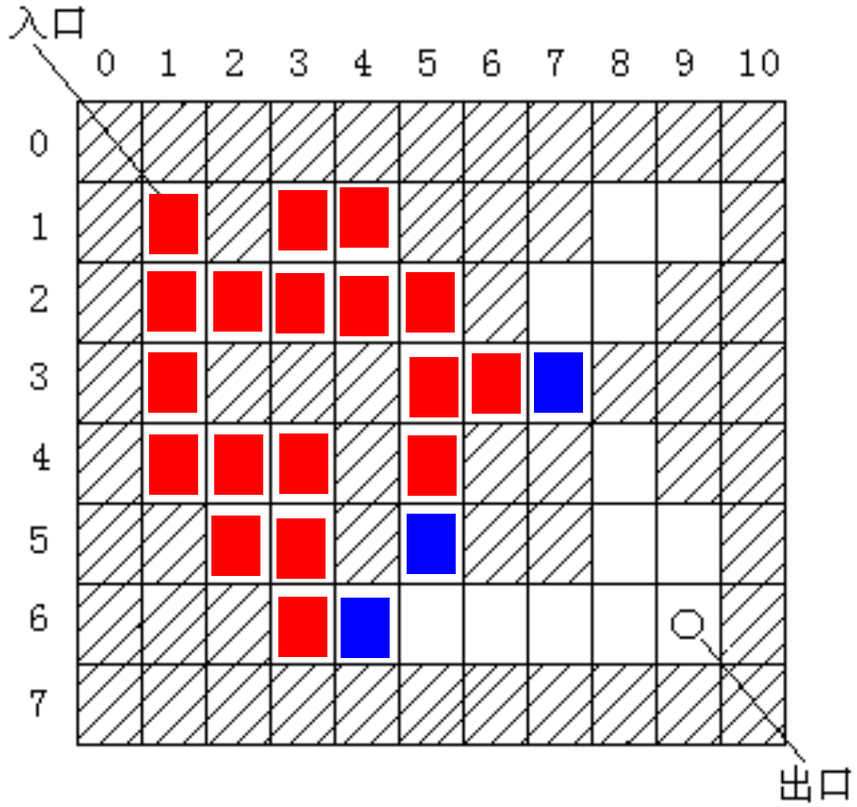




状态空间搜索：栈和队列

- **基于队列搜索迷宫的两个场景：**
  蓝色表示队列里记录的位置





状态空间搜索：栈和队列

基于栈的搜索过程的一些特点：

- **搜索比较“冒进”，可能在一条路走很远，“勇往直前/不撞南墙不回头”**
- **有可能在探查了不多的状态就找到了解；也可能陷入很深的无解区域。如果陷入的无解子区域包含无穷个状态，这种搜索方法就找不到解了**

基于队列搜索的一些特点：

- **是稳扎稳打的搜索，只有同样距离的状态都检查完后才更多前进一步**
- **队列里保存的是已搜索区域的前沿状态**
- **如果有解（通过有穷长路径可以到达结束状态），这种搜索过程一定能找到解，而且最先找到的必定是最短的路径（最近的解）**

状态空间搜索：栈和队列

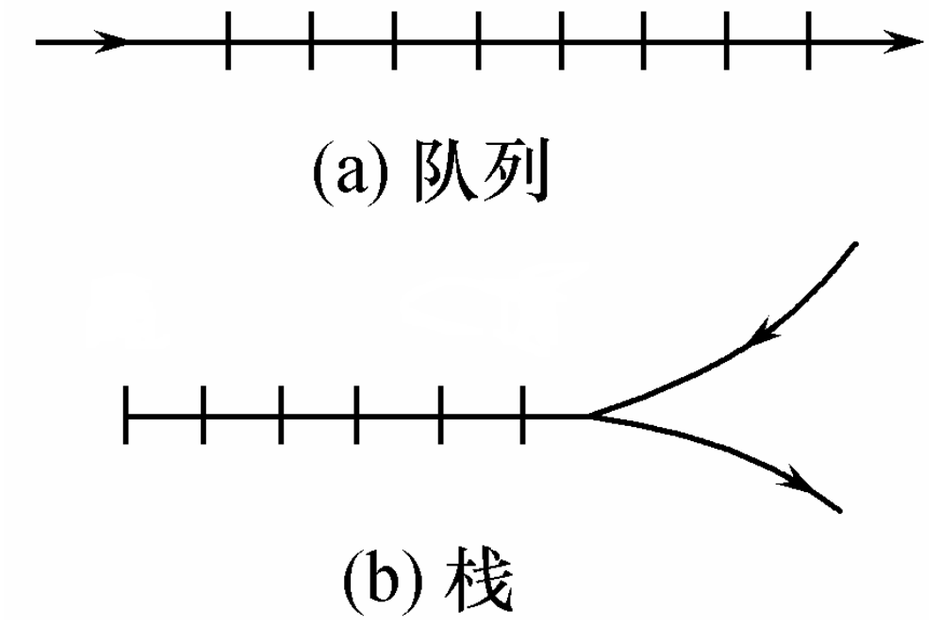
- 由于这两种搜索的性质，
  - 基于栈的搜索被称为“深度优先搜索”（depth-first search）
  - 基于队列的搜索被称为“宽度优先搜索”（width-first search）
- 如果找到了解，如何得到有关的路径：
  - 基于栈的搜索，可以让栈里保存所找到路径上历经的状态序列
  - 基于队列的搜索，需要用其他方法记录经过的路径。一种方式是在到达每个状态时记录其前一状态，最后追溯这种“前一状态”链，就可以确定路径上的状态（相当于对每个状态形成一个栈）
- 搜索所有可能的解和最优解
  - 基于栈搜索，找到一个解之后继续回溯，有可能找到下一个解。遍历完整个状态空间就能找到所有的解，从中可以确定最优解
  - 基于队列搜索，找到一个解之后继续也有可能找到其他解，但是到各个解的路径将越来越长，第一个解就是最优解

状态空间搜索：栈和队列

- 空间开销：搜索状态空间需要保存中间状态，空间开销就是搜索过程中栈或队列里的最大元素项数。两种搜索在这方面的表现也很不同
  - 基于栈的深度优先搜索，所需的栈空间由找到一个解（或所有解）之前遇到的最长搜索路径确定，这种路径越长，需要的存储量就越大
  - 基于队列的宽度优先搜索，所需的队列空间由搜索过程中最多的可能路径分支确定。可能分支越多，需要的存储量就越大。此外，为能得到路径还需要另外的存储，其存储量与搜索区域的大小线性相关
- 总结：如果一个问题可以看作空间搜索问题，栈和队列都可以使用，具体用什么要看实际问题的各方面性质

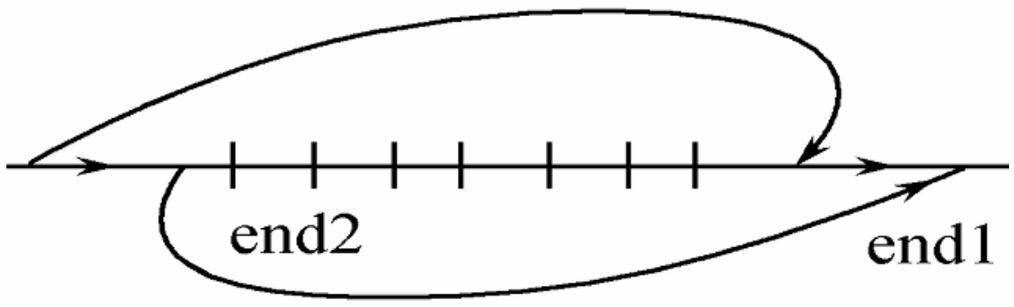
限制存取点的表

- 栈和队列的运算都限制在它们的端点上进行，所以也称为限制存取点的表。除栈和队列以外，实用的限制存取点的表还有多种。

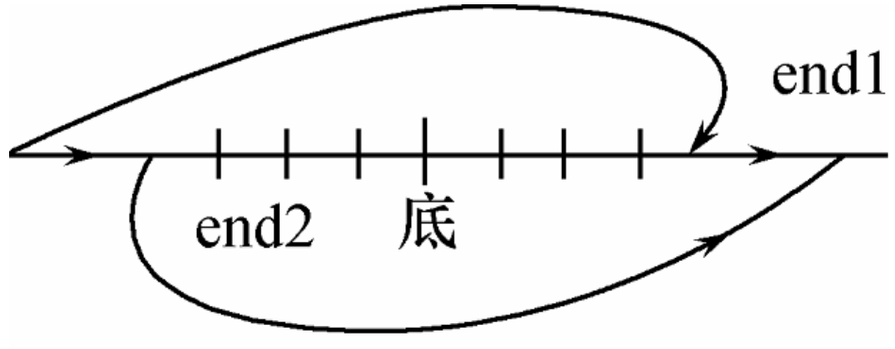


双端队列和双栈

- 双端队列是一种特殊的线性表，对它所有的插入和删除都限制在表的两端进行。这两个端点分别记作end1和end2。

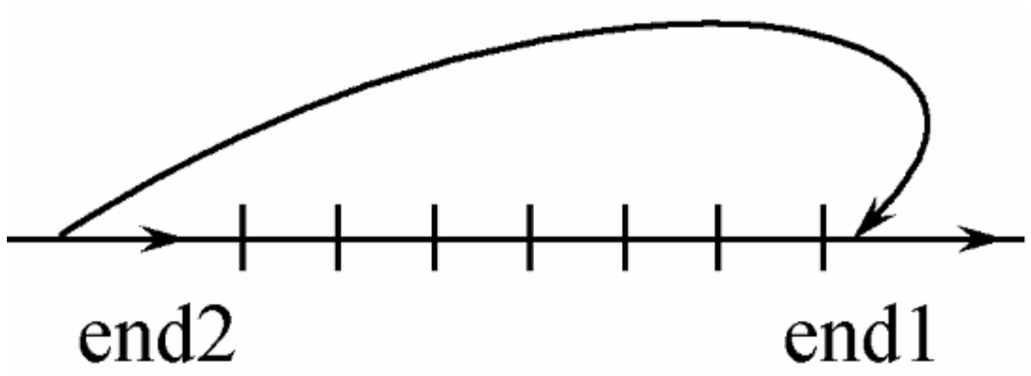


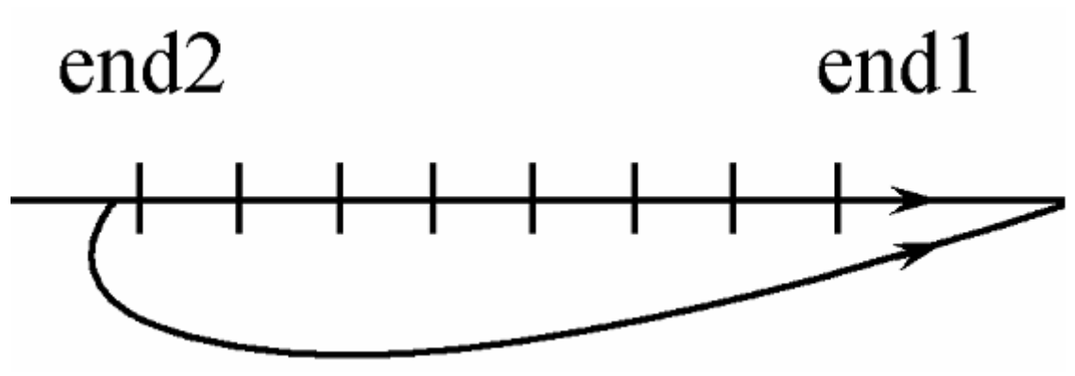
- 双栈是一种加限制的双端队列，它规定从end1插入的元素只能从end1端删除，而从end2插入的元素只能从end2端删除。它就好象两个底部相连的栈。



超队列和超栈

- 超队列是一种输出受限的双端队列，即删除限制在一端(例如end1)进行，而插入仍允许在两端进行。
- 它好象一种特殊的队列，允许有的最新插入的元素最先删除。
- 超栈是一种输入受限的双端队列，即插入限制在一端（例如end2）进行，而删除仍允许在两端进行。
- 它可以看成对栈溢出时的一种特殊的处理，即当栈溢出时，可将栈中保存最久（即end1端）的元素删除。





本讲重点

- 队列的ADT，存储表示和算法的实现
- 提高使用计算机完成问题求解的能力

两种不同的搜索策略：

- 一种是（使用队列实现的）广度优先(breadth_first) 搜索
- 另一种是（使用栈实现的）深度优先(depth_first) 搜索。

本讲内容非常重要！！