数据结构

夏天

xiat@ruc.edu.cn

中国人民大学信息资源管理学院

栈和队列

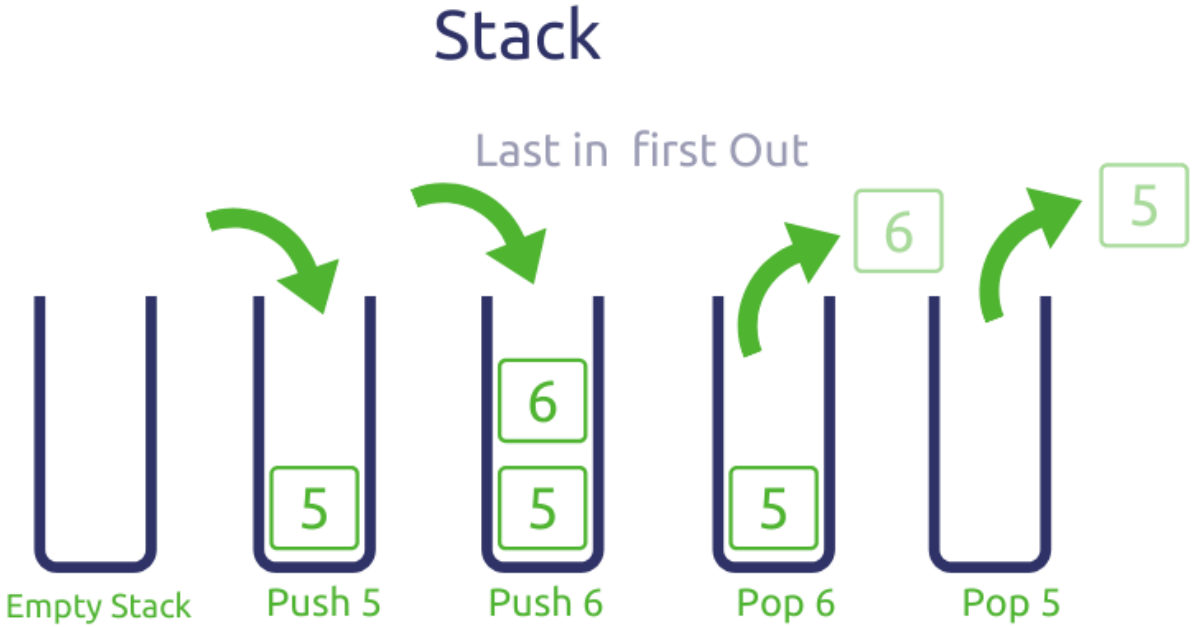
栈和队列：两种特殊的线性表

栈/Stack

- 限制仅在表的一端进行插入和删除。
- 通常称插入、删除的一端为栈顶 (Top)，另一端为栈底 (Bottom)
- LIFO：Last In First Out

队/Queue

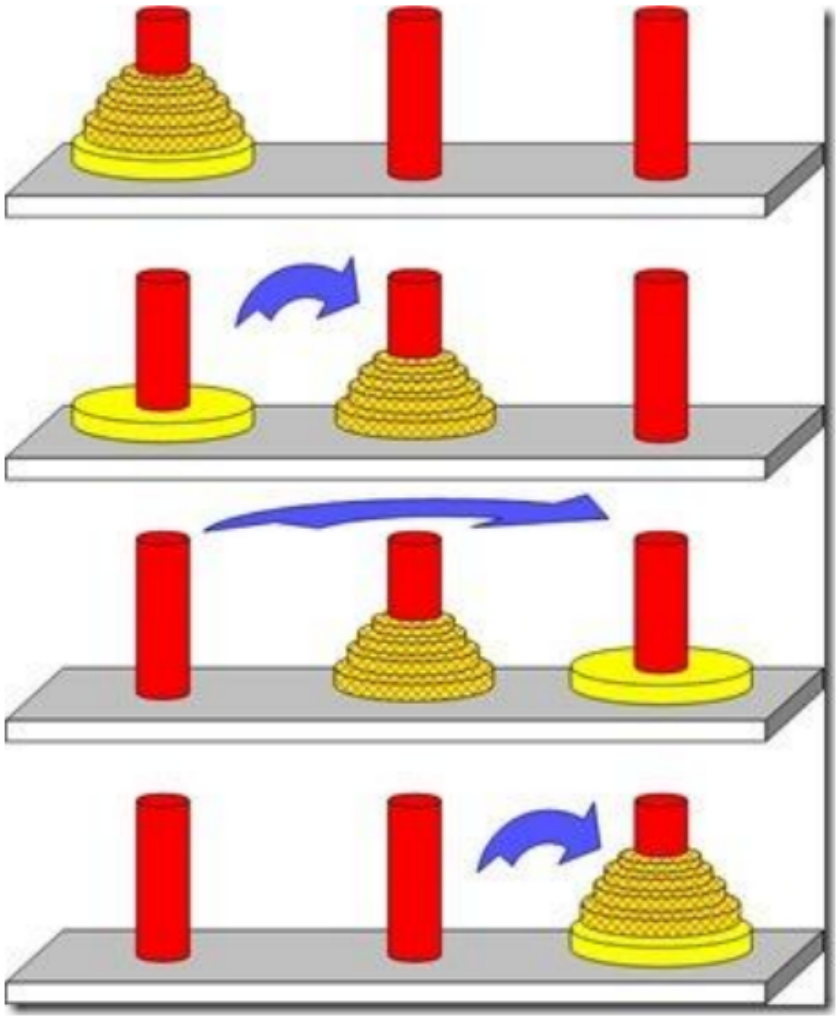
- 限制仅在表的一端进行插入、在另一端进行删除。
- 允许插入的一端称队尾 (rear)，允许删除的一段称为队头 (front)。
- FIFO：First In First Out



- 操作系统中的栈
  - 由编译器自动分配释放，存放函数的参数值，局部变量的值等。栈使用的是一级缓存，被调用时处于存储空间中，调用完毕立即释放。
- 数据结构中的栈
  - 一种后进先出的数据结构

为什么设计栈、研究栈？

- 栈的一个重要应用是在程序设计中实现递归，从而使许多实际问题大大简化。

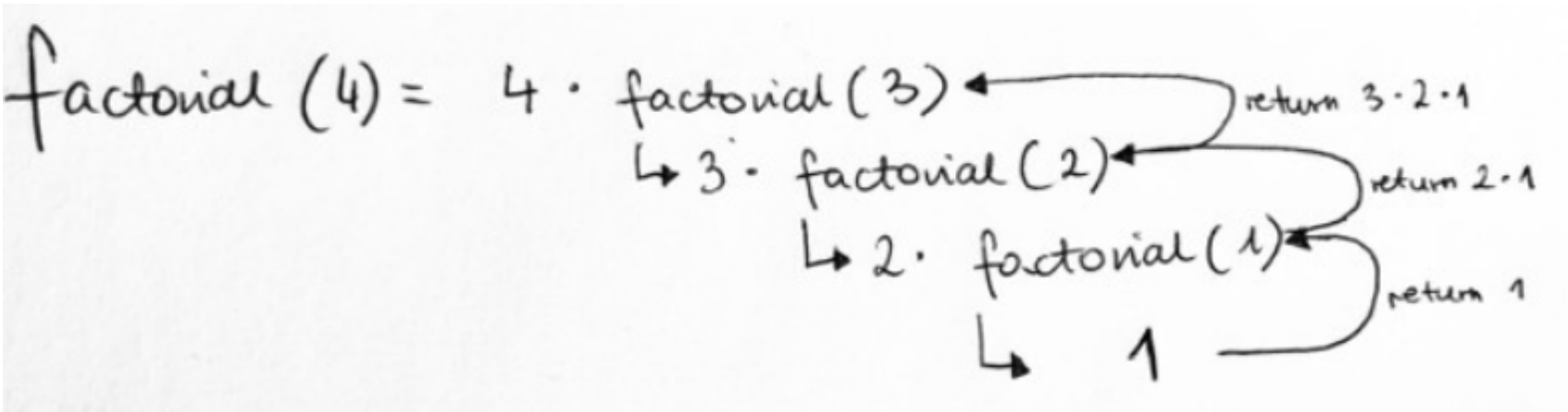


- 上帝创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按大小顺序摞着 64 片黄金圆盘。上帝命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上，并且规定一次只能移动一个圆盘，在小圆盘上不能放大圆盘。
- 有预言说，这件事完成时宇宙会在一瞬间闪电式毁灭。也有人相信婆罗门至今还在一刻不停地搬动着圆盘。
- 18,446,744,073,709,551,615 次搬动才能挪完 64 片金盘！

举例：计算 n 的阶乘

```
int factorial (int n) {
 int  f;
 if (n==1) f=1;
 else f=n*fact (n-1);
 return f;
}
```

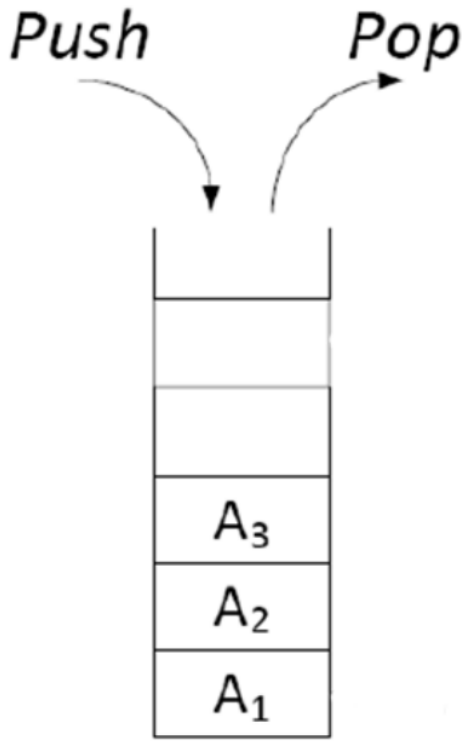
1. 将调用函数的现场（各寄存器的值，中断时的程序地址等）入栈，转入被调函数；
2. 执行被调函数，如又调用其它函数，则执行上述步骤；
3. 被调函数执行完，取栈顶的值，恢复调用函数时的现场，根据现场中的指令地址，恢复调用函数在中断处继续执行。



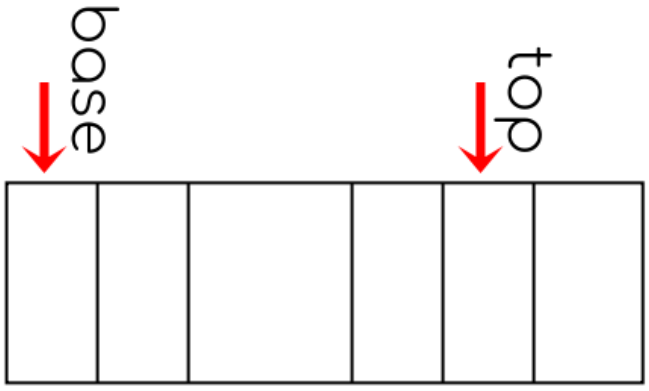
栈的存储表示方法

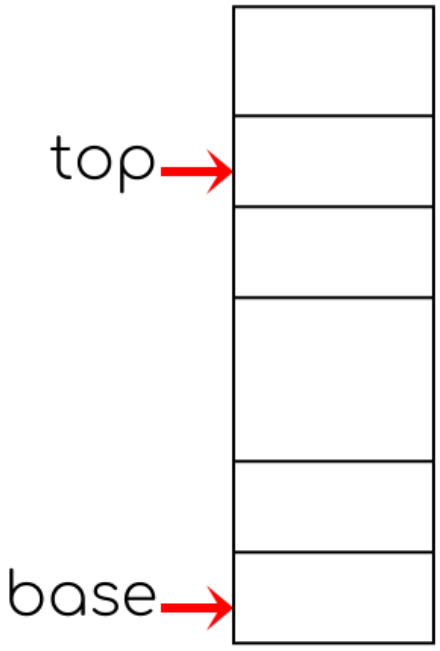
请考虑其常用操作。试想选用顺序存储还是链式存储？

- 特点：后进先出
  - 1、经常性的在栈顶插入新元素，以及取栈顶元素；
  - 2、无须访问非栈顶元素。



1. 顺序栈



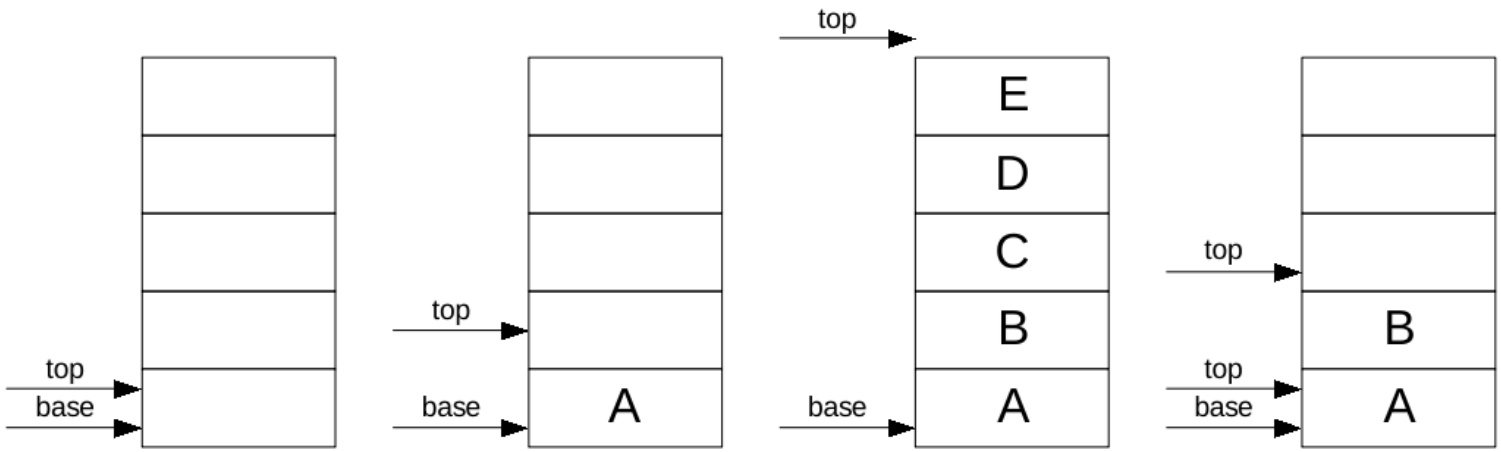


1. 顺序栈中元素用地址连续的存储单元依次存放；
2. 栈底位置固定不变；
3. 栈顶位置 top 随着进栈和出栈的操作而变化。

顺序栈类型定义：

```
class stack<Elem>{
  Object[] data;
  int top;
  int maxSize;
} SeqStack;
```

- 动态分配
  - 先为栈分配一个初始容量，在栈的空间不够使用时再逐段扩大。
- 指针 base
  - 始终指向栈底位置，如果 base 为 NULL 表示栈不存在；
- 指针 top
  - 初值指向栈底，即 top = =base，表示栈空；(不唯一)
  - 每当插入新的栈顶元素，指针 top++；
  - 每当删除栈顶元素，指针 top--；
  - 非空栈中的栈顶指针始终在栈顶元素的下一个位置上。



对于顺序栈，入栈需要判栈是否满，是则需要中止、或重新分配空间，否则出现空间溢出。

```
public Boolean push (Elem e) {
 if ( top==maxSize) {
   print(" 栈已满");
   return false;
 }
 data[top++] = e;
 return true;
}
```

对于顺序栈，入栈需要判栈是否满，是则需要中止、或重新分配空间，否则出现空间溢出。

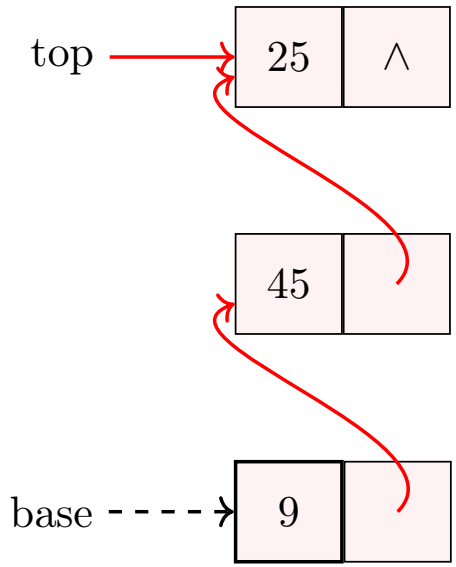
```
public Boolean push (Elem e) {
 if ( top==maxSize) {
   print(" 栈已满");
   return false;
 }
 data[top++] = e;
 return true;
}
```

出栈首先要判断栈是否为空；否则栈空时进行操作将出现下溢错误。

```
public Elem pop() {
 if(top==base) {
   return null;
 } else {
   top = top-1
   return data[top];
 }
}
```

2. 链栈

- 链栈有无栈满溢出问题？
- 指针如何指？
  - 从栈顶依次向后指，因为操作主要是在栈顶插入、删除，经常需要根据栈顶元素找次顶元素。
- 是否要加头结点？
  - No. 因为头部插入不会出现处理不一致的问题.



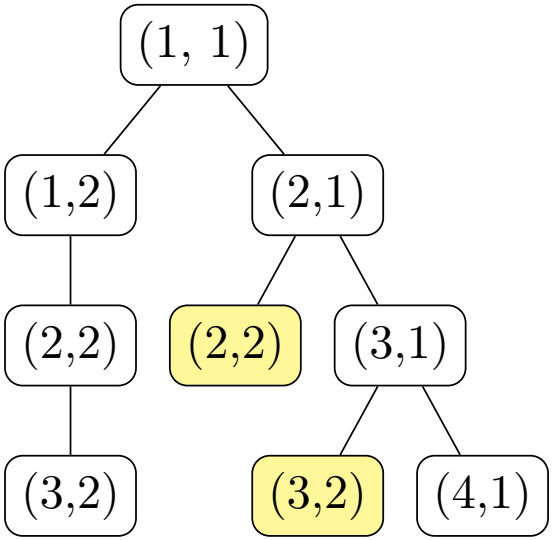
写出链栈的类型定义:

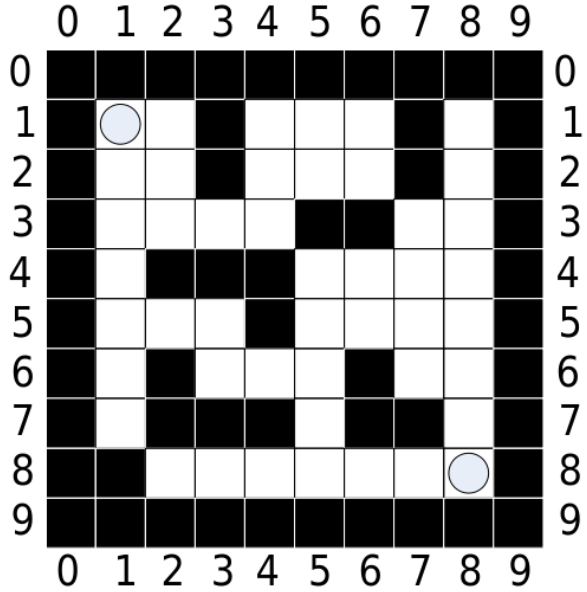
```
class Node{
 Elem data;
 Node next;
}

Node top;//链栈
```

请尝试写出入栈出栈的算法

```
public push (Elem e) {
  //是否还要判断栈是否满?
  top=new Node(e, top);
  return OK;
}

public pop( ){
  //是否还要判断栈是否空?
  e=top;
  top=top.next;
  e.next=null;
  return e.data;
}
```

栈的应用 — 迷宫求解





- 处理思路：
  - 对可通行路径的空间的深度优先搜索
  - 如果当前位置是出口，找到结果
  - 否则，保存当前位置，进入到下一个可以走的位置，探索路径
  - 如果上一步没有成功，继续探索下一个可以走的位置。

队/Queue

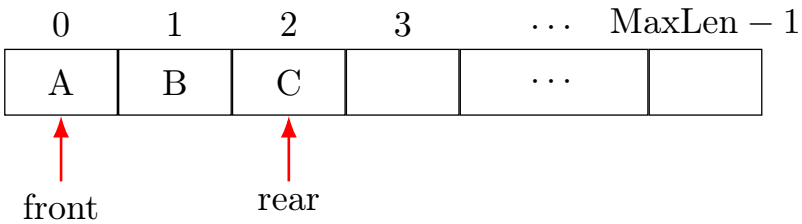
队/Queue

- 限制仅在表的一端进行插入、在另一端进行删除。
- 允许插入的一端称队尾 (rear)，允许删除的一段称为队头 (front)。
- FIFO：First In First Out

- 比如生活中排队购物、操作系统中的作业排队等。

队的存储表示方法

- 队也有两种存储表示方法: 顺序队、链队
- 顺序队：利用地址连续的存储单元依次存放数据元素。



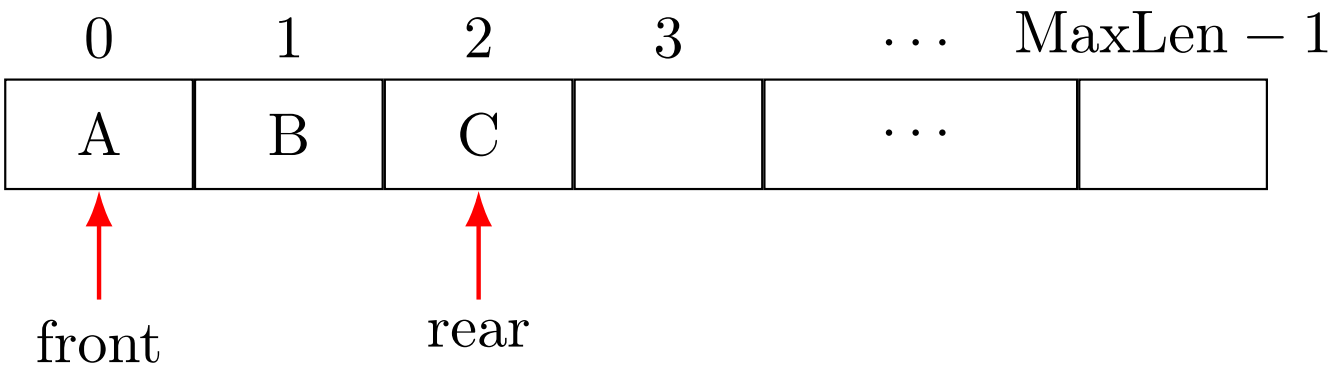
请写出顺序队的类型定义：

```
class SeqQueue{
  ElemType  data[ ];
  int maxsize;
  int rear; //队尾, 以此计算队长
}
```

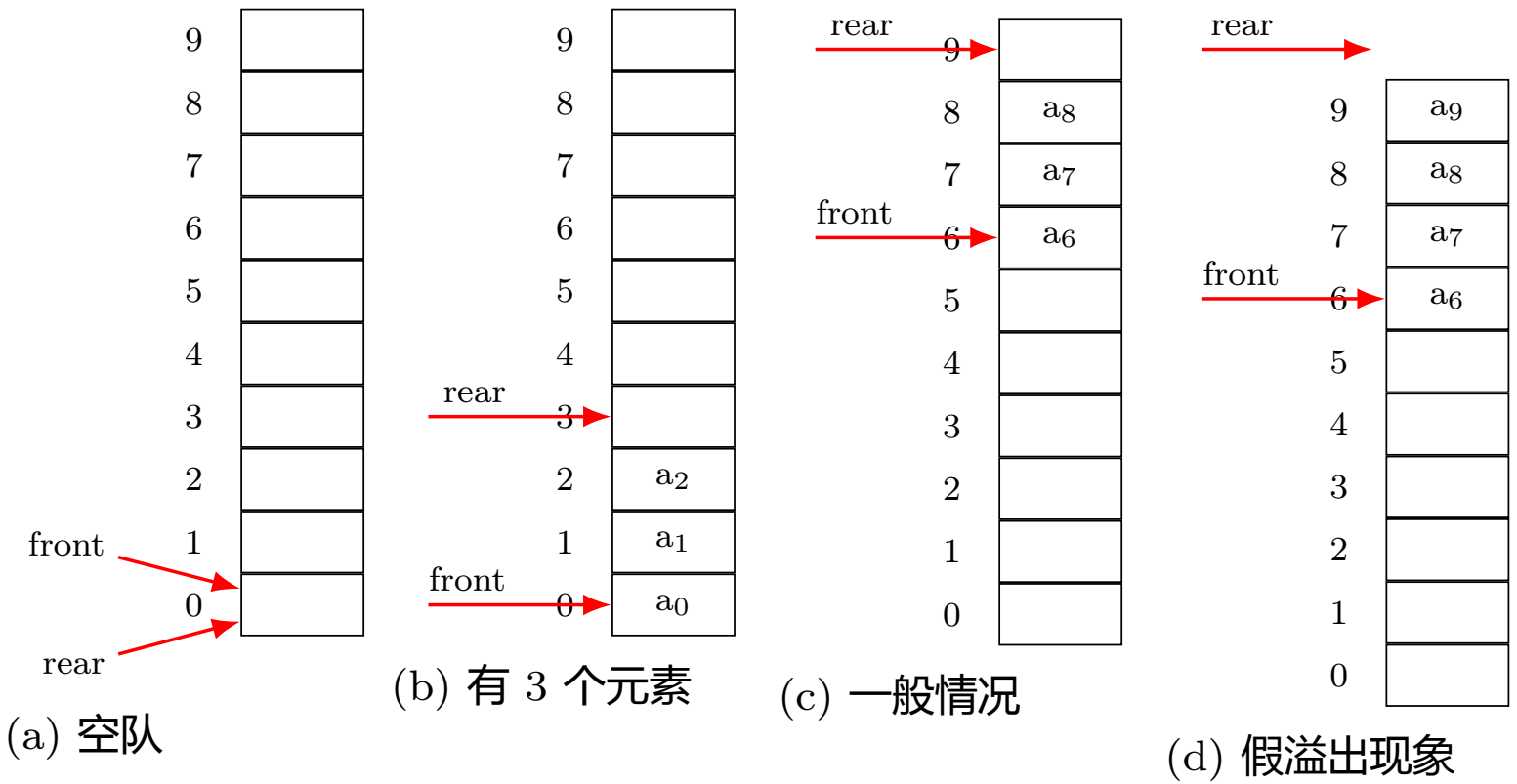
可否??

顺序队的基本操作

- 入队：如有空间，元素 x 入队后队尾指针加 1
  - data[rear++]=x;
- 出队：如有元素，队头指针加 1，表明队头元素出队。
  - x=sq.data[sq.front++];

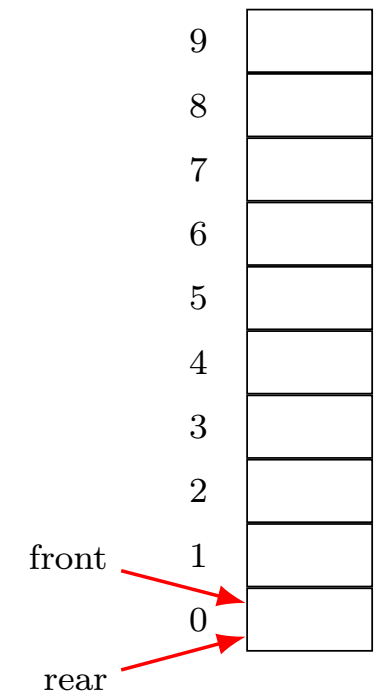


- 设 maxsize=10。请你写出如下状态或执行某操作后的顺序队元素。
  1. 空队；
  2. a0, a1, a2 依次入队；
  3. a3, a4, a5, a6, a7, a8 依次入队，a0, a1, a2, a3, a4, a5 依次出队;
  4. a9, a10 入队。

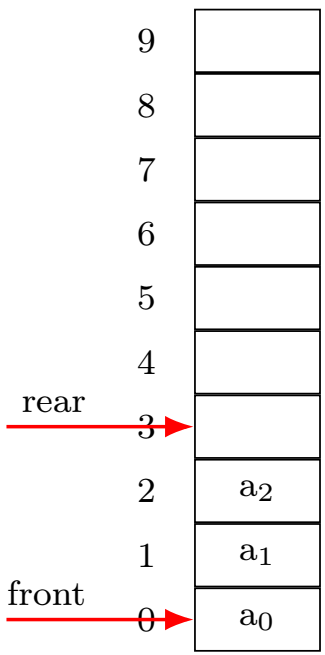


(a) 空队　(b) 有 3 个元素　(c) 一般情况　(d) 假溢出现象

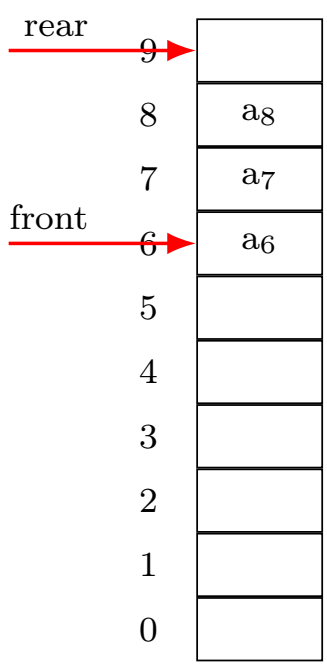
- 设 maxsize=10。请你写出如下状态或执行某操作后的顺序队元素。
  1. 空队；
  2. a0, a1, a2 依次入队；
  3. a3, a4, a5, a6, a7, a8 依次入队，a0, a1, a2, a3, a4, a5 依次出队;
  4. a9, a10 入队。



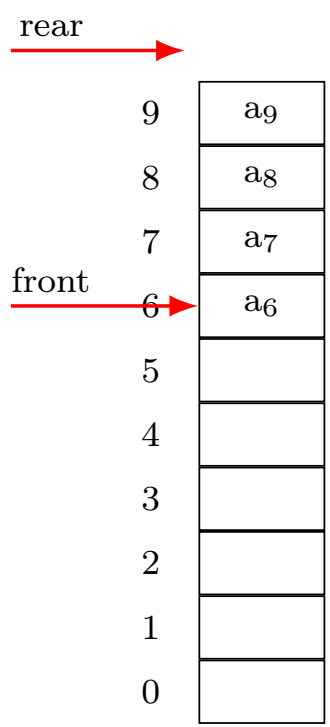
(a) 空队



(b) 有 3 个元素



(c) 一般情况



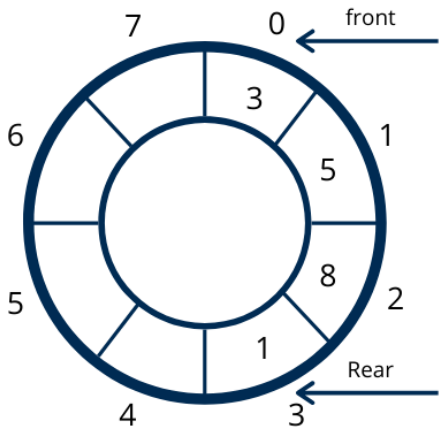
(d) 假溢出现象

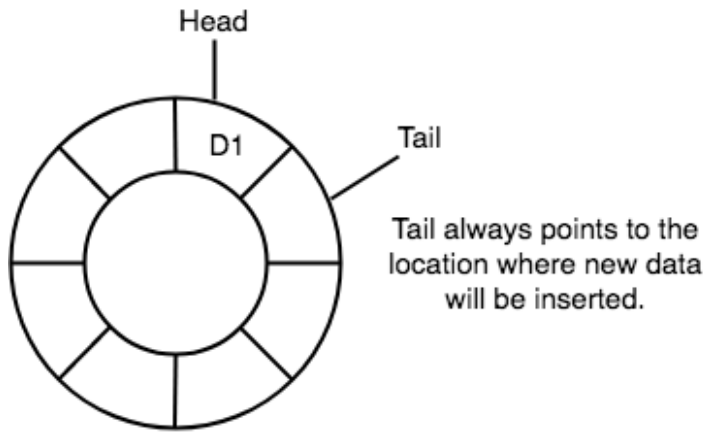
- 入队出队会使整个队列整体后移
- “假溢出”：队尾指针已经移到了最后，不能再执行入队操作，但此时队中并未真的“满员”。如何解决这个问题？

- Linear Queue 入队出队会使整个队列整体后移
- “假溢出”：队尾指针已经移到了最后, 不能再执行入队操作，但此时队中并未真的“满员”。如何解决这个问题？
  1. 平移元素；
  2. 循环队列 (Circular Queue)：将顺序队列的存储区假想为环状空间。在假溢出时，将新元素插入到第一个位置上，这样做，虽物理上队尾在队首之前，但逻辑上队首仍然在前。入列和出列仍按“先进先出”的原则进行。
     - ★ 入队时的队尾指针操作：
       rear=(rear+1) % maxsize;
     - ★ 出队时的队头指针操作：
       front=(front+1) % maxsize;

循环队列 (Circular Queue)[1]

- head/front points to the first (oldest) used element — the next element to be read
- tail/rear points to the first (oldest) unused element — the next element to be written
- rear: 多数情况下指向尾部元素，tail: 多数情况下指向尾部的下一个元素。教材中的 rear 等同于 tail



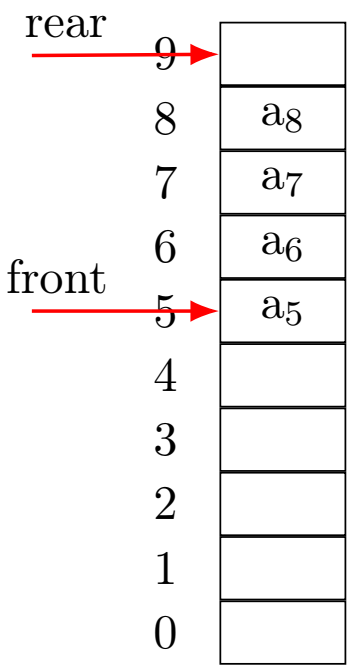


[1] 延伸阅读：https://en.wikipedia.org/wiki/Circular_buffer

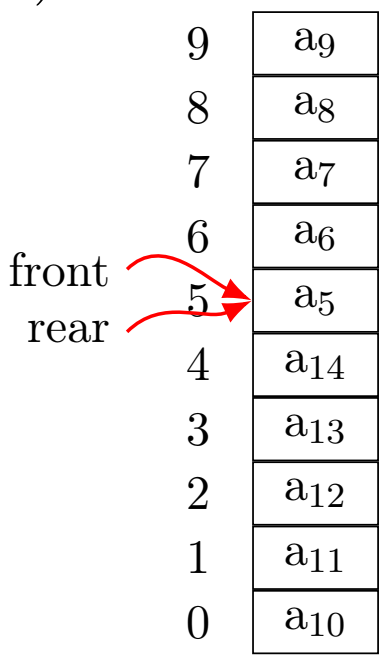
- 设 maxsize=10。队列状态如 (a) 所示。请写出分别执行如下操作后的 front 和 rear 值：
  - 情况 1: a9, a10, a11, a12, a13, a14 依次入队
  - 情况 2: a5, a6, a7, a8 依次出队

- 设 maxsize=10。队列状态如 (a) 所示。请写出分别执行如下操作后的 front 和 rear 值：
  - 情况 1: a9, a10, a11, a12, a13, a14 依次入队
  - 情况 2: a5, a6, a7, a8 依次出队

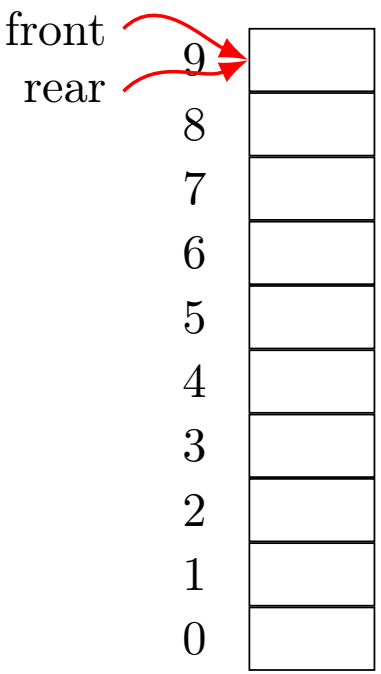
(a) 有 4 个元素



(b) 情况 1: 队满



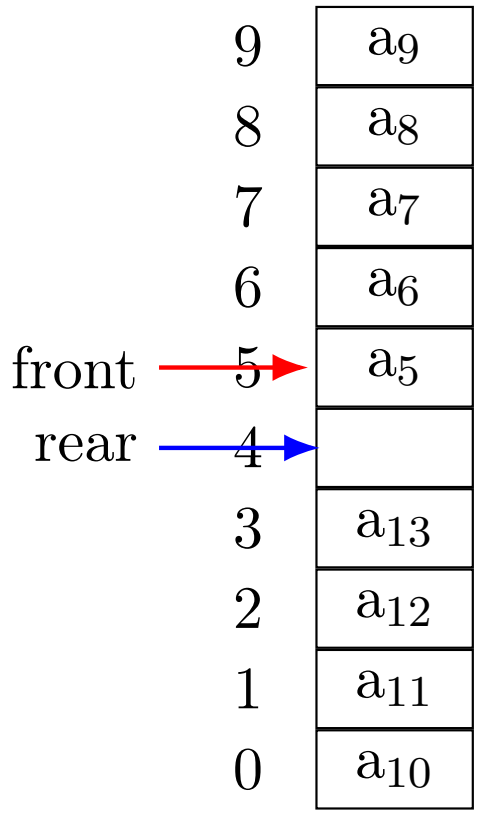
(c) 情况 2: 队空



当 front=rear, 队满还是队空？如何判断队满？如何判断队空？

循环队的队满判别

- 方法一
  - 附设一个存储队中元素个数的变量如 num，当 num=0 时队空，当 num=maxsize 时为队满。
- 方法二
  - 少用一个元素空间，把右图所示情况视为队满，此时的状态是队尾指针加 1 就会从后面赶上队头指针：(rear+1) % maxsize==front，从而和空队区别开来。



循环队的基本操作

```
class SeqQueue  {
 ElemType data[];
 int maxsize;
 int front, rear; /*队头队尾指针*/
 int num;       /*队中元素的个数*/
}

//入队
int InQueue (ElemType  x){
 if  (num==maxsize) {
  printf(" 队满"); return 0;
 } else {
  data[rear]=x;
  rear=(rear+1) % maxsize;
  num++;
  return 1;   /*入队完成*/
 }
}
```

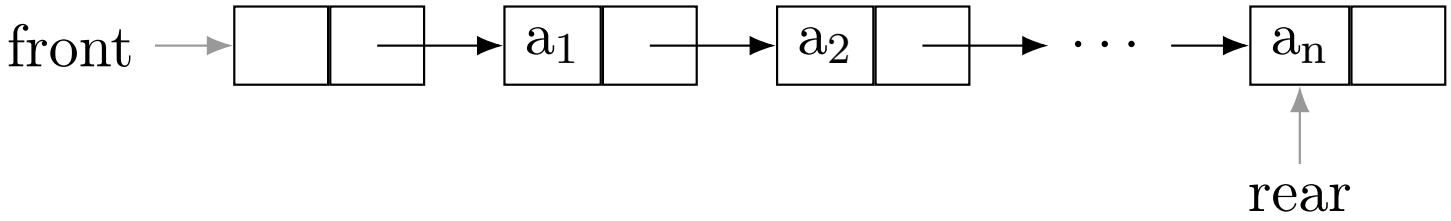
循环队的基本操作

```
//出队
int OutQueue (ElemType  x) {
 if  (num==0) {
   printf(" 队空");
   return -1;
 } else {
   x=data[front]; /* 读出队头元素 */
   front=(front+1) % maxsize;
   num--;
   return 1;  /* 出队完成 */
 }
}
```

Python:

```
class SeqQueue:
    def __init__(self, max_size=10):
        self.max_size = max_size + 1 #实际多放一个空间，区分对空、队满
        self.front = 0
        self.rear = 0
        self.elements = [None]*max_size

    def enqueue(self, element) -> bool:
        if (self.rear + 1) % self.max_size == self.front:
            print('Queue is Full!')
            return False
        else:
            self.elements[self.rear] = element
            self.rear = (self.rear + 1) % self.max_size
            return True

    def dequeue(self):
        if self.front == self.rear:
            print('Queue is Empty!')
            return None
```

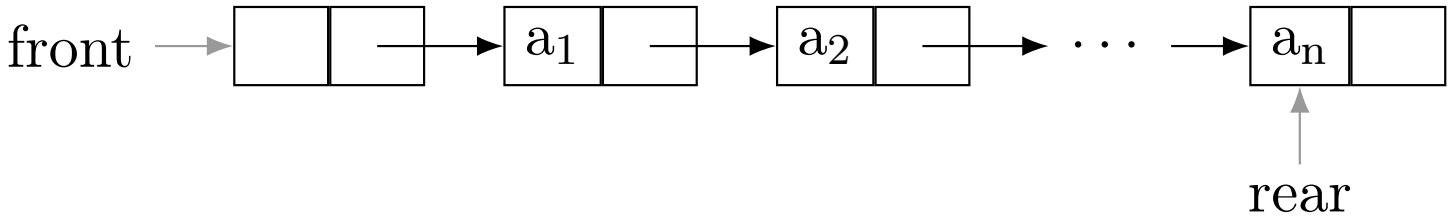
链队



- 带头结点吗？
- 尾指针 rear 指向队尾元素，提高插入操作效率
- 请写出链队的类型定义

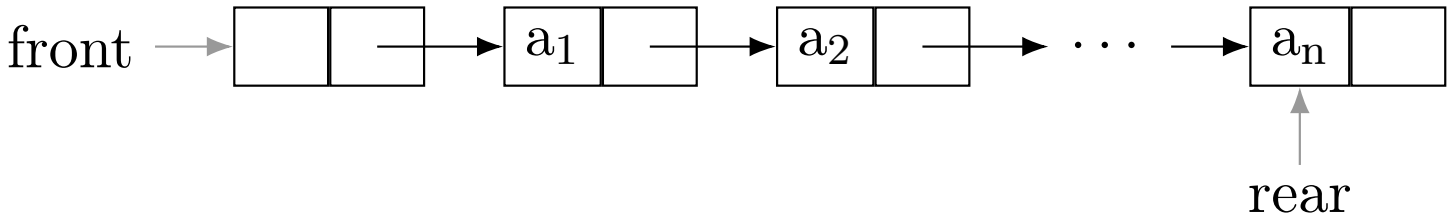
```
class QNode{
  ElemType  data;
  QNode next;
}
class LQueue{
  Node head, rear;
}
```

链队的基本操作 — 入队




```
void InQueue(ElemType  x) {
  rear.next=new QNode(x,null);
  rear=rear.next;
}
```

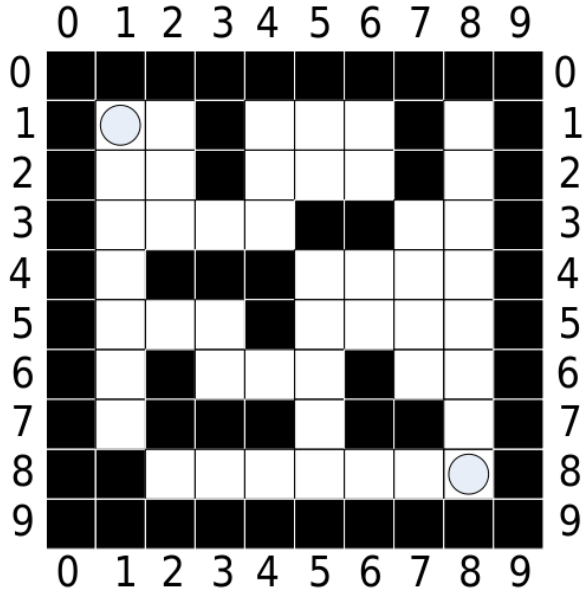
链队的基本操作 — 出队



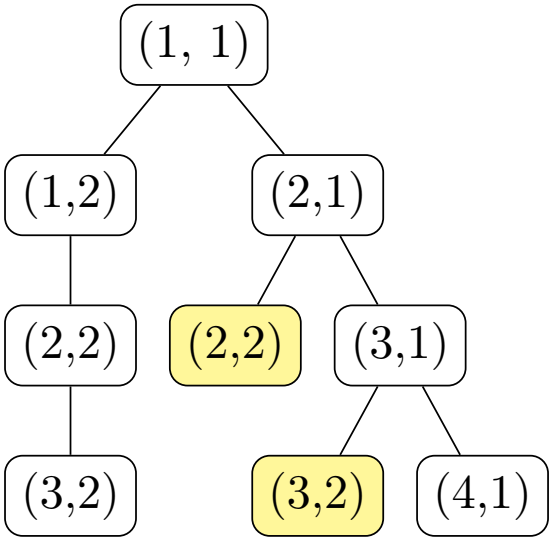

```
ElemType OutQueue() {
  QNode p;
  if (front==rear) {
    printf (" 队空");
    return null;
  } else {
    p=front.next;   //把队的第一个结点取出
    front.next=p.next;  //把队头直接指向原第二个结点
    e=p.data;     /* 队头元素放 e 中 */
    if (front.next==NULL)
```

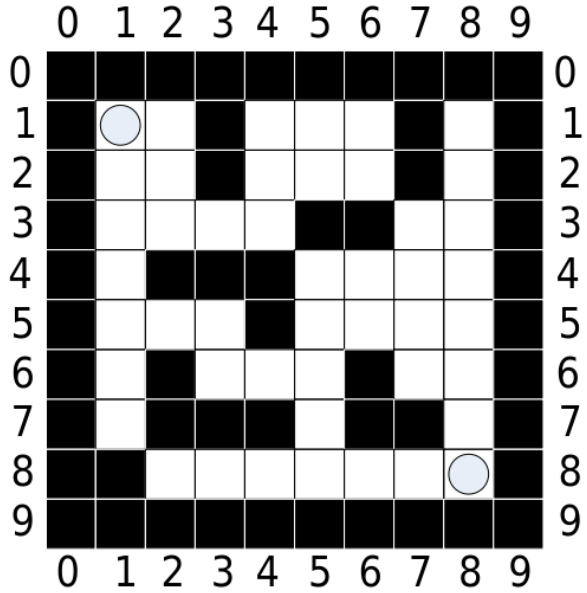
队列应用 — 迷宫寻路

- 从迷宫入口点 (1,1) 出发，向四周搜索，记下所有一步能到达的坐标点；然后依次再从这些点出发，再记下所有一步能到达的坐标点，...，依此类推，直到到达迷宫的出口点 (8,8) 为止，然后从出口点沿搜索路径回溯直至入口。这样就找到了一条迷宫的最短路径，否则迷宫无路径。
- 与基于栈的迷宫求解的异同？



解法图示

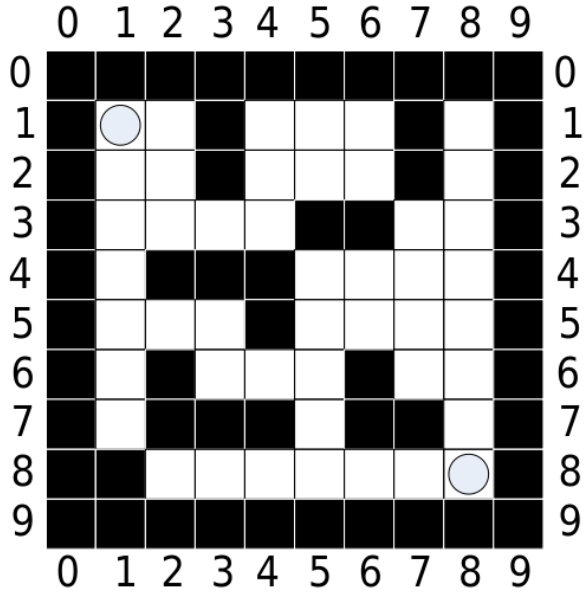




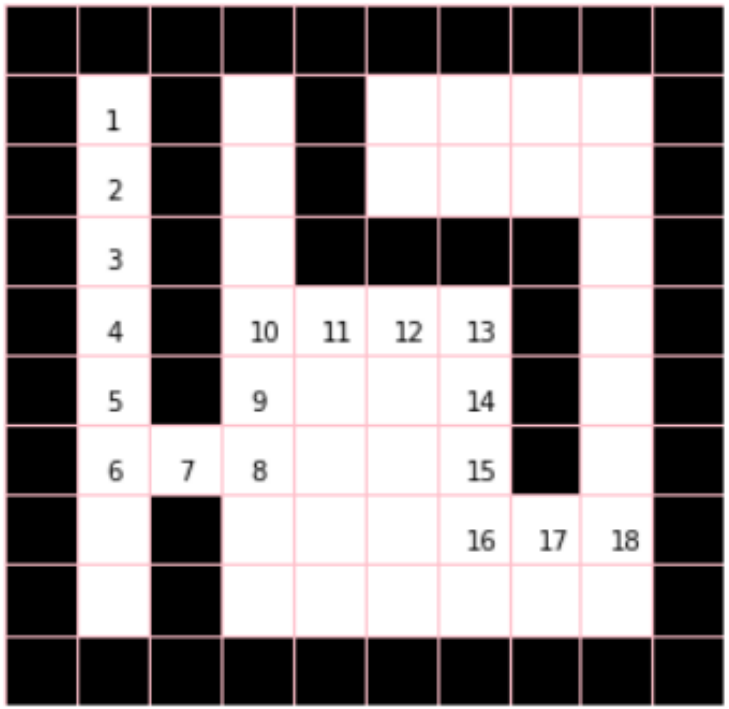
- 上述处理的实质：
  - 对可通行路径的空间的广度优先搜索
- 如何向前摸索？
- 到达出口后，如何回溯？

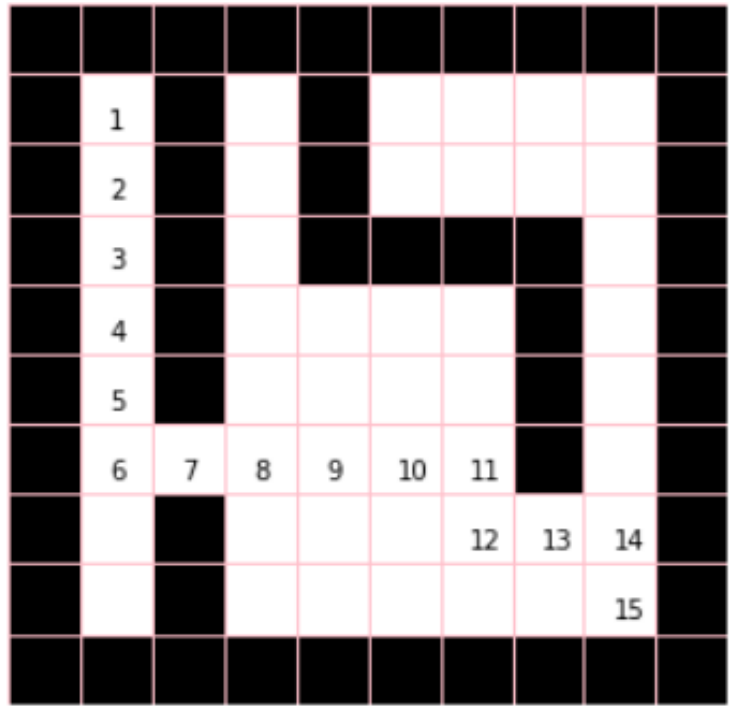
解法图示

- 如何向出口摸索：搜索路径的存储 1
  - 在搜索过程中必须记下每一个可到达的坐标点，以便从这些点出发继续向四周搜索，使用“FIFO”的队列保存已到达的点即可.
- 如何向入口回溯：搜索路径的存储 2
  - 为了能够从出口点沿搜索路径回溯直至入口，对于每一点，记下坐标点的同时，还要记下到达该点的前驱点或者从前驱点来的方向（8 个方向之一）。



夏老师实现结果的对比





实现代码见 ipynb/stack.ipynb 和 ipynb/queue.ipynb