数据结构与算法

栈与队列

张晓平

武汉大学数学与统计学院

2016 年 10 月 11 日

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

基本概念和抽象数据类型

定义 (栈) 栈 (Stack) 是限定仅在表尾进行插入和删除操作的线性表。

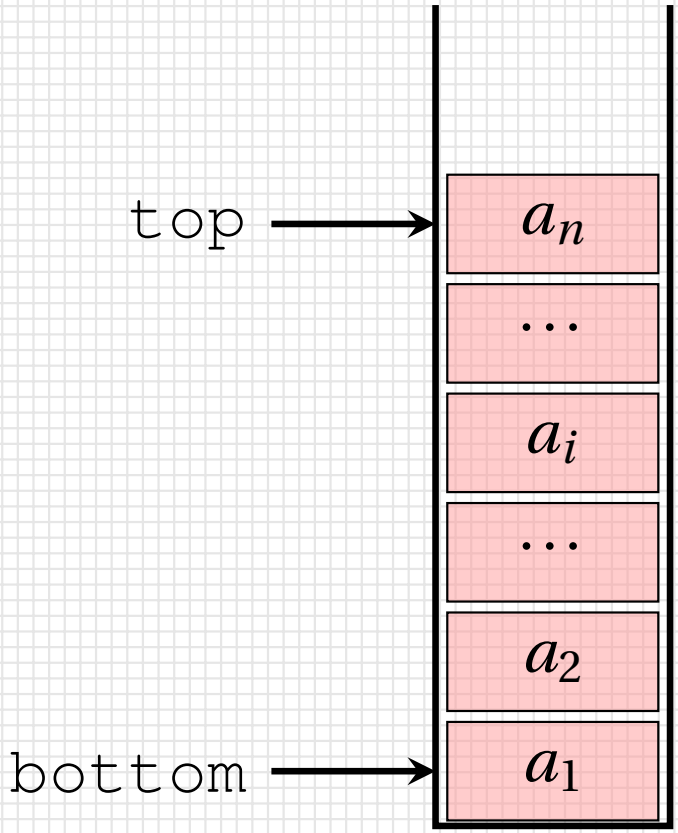
- 允许进行插入、删除操作的一端称为栈顶 (top)，另一端称为栈底 (bottom)；
- 不含任何数据元素的栈称为空栈；
- 栈又称后进先出 (Last In First out) 的线性表，简称 LIFO 结构。

基本概念和抽象数据类型

- 栈是一种特殊的线性表，仍具有前驱后继关系。
- 栈限制了插入和删除的位置，这些操作始终只在栈顶进行。于是栈底是固定的，最先进栈的只能在栈底。

基本概念和抽象数据类型

设栈 $S=(a_1,a_2,\cdots,a_n)$，称 a_1 为栈底元素，a_n 为栈顶元素。



基本概念和抽象数据类型

- 栈的插入操作叫做进栈，也称压栈、入栈。
- 栈的删除操作叫做出栈，也称弹栈。

基本概念和抽象数据类型

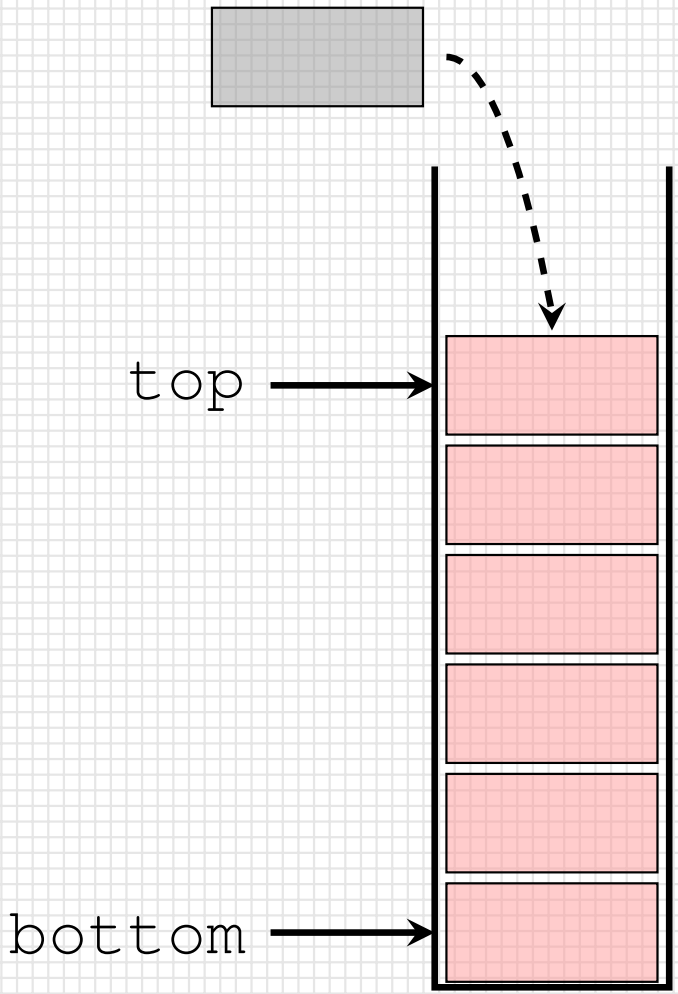


图: 进栈

基本概念和抽象数据类型

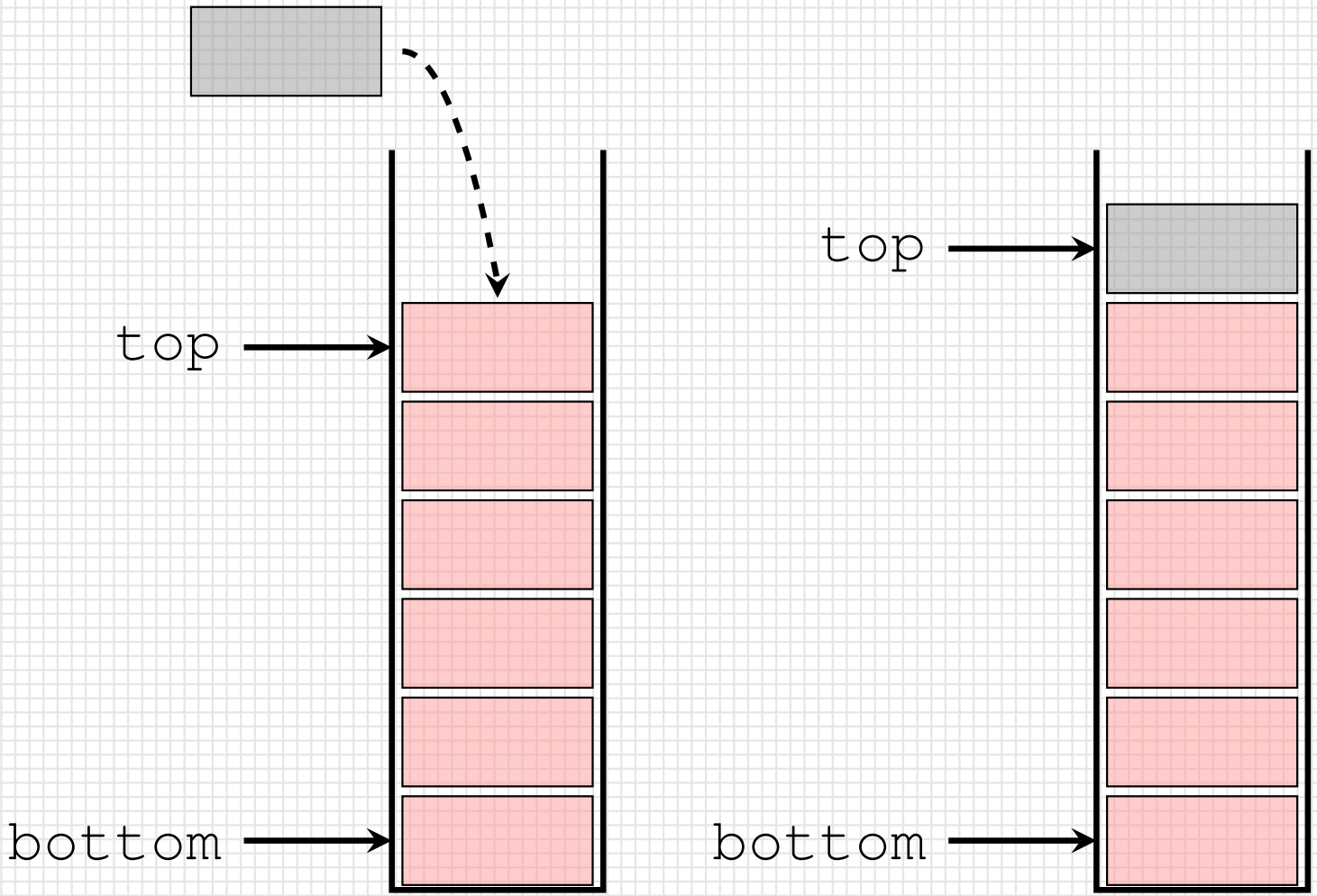


图: 进栈

基本概念和抽象数据类型

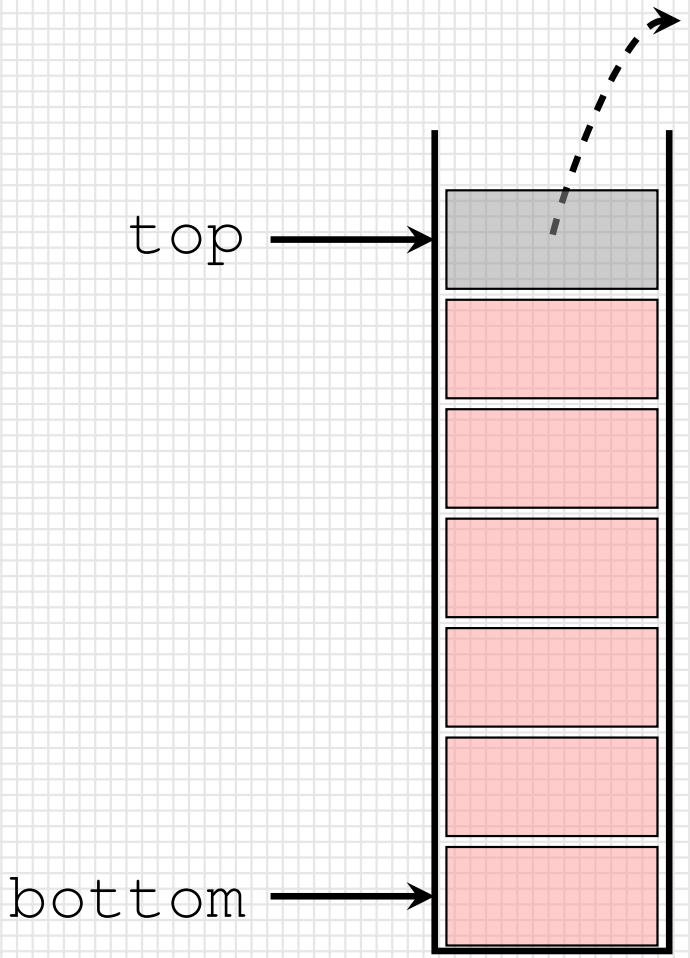


图: 出栈

基本概念和抽象数据类型

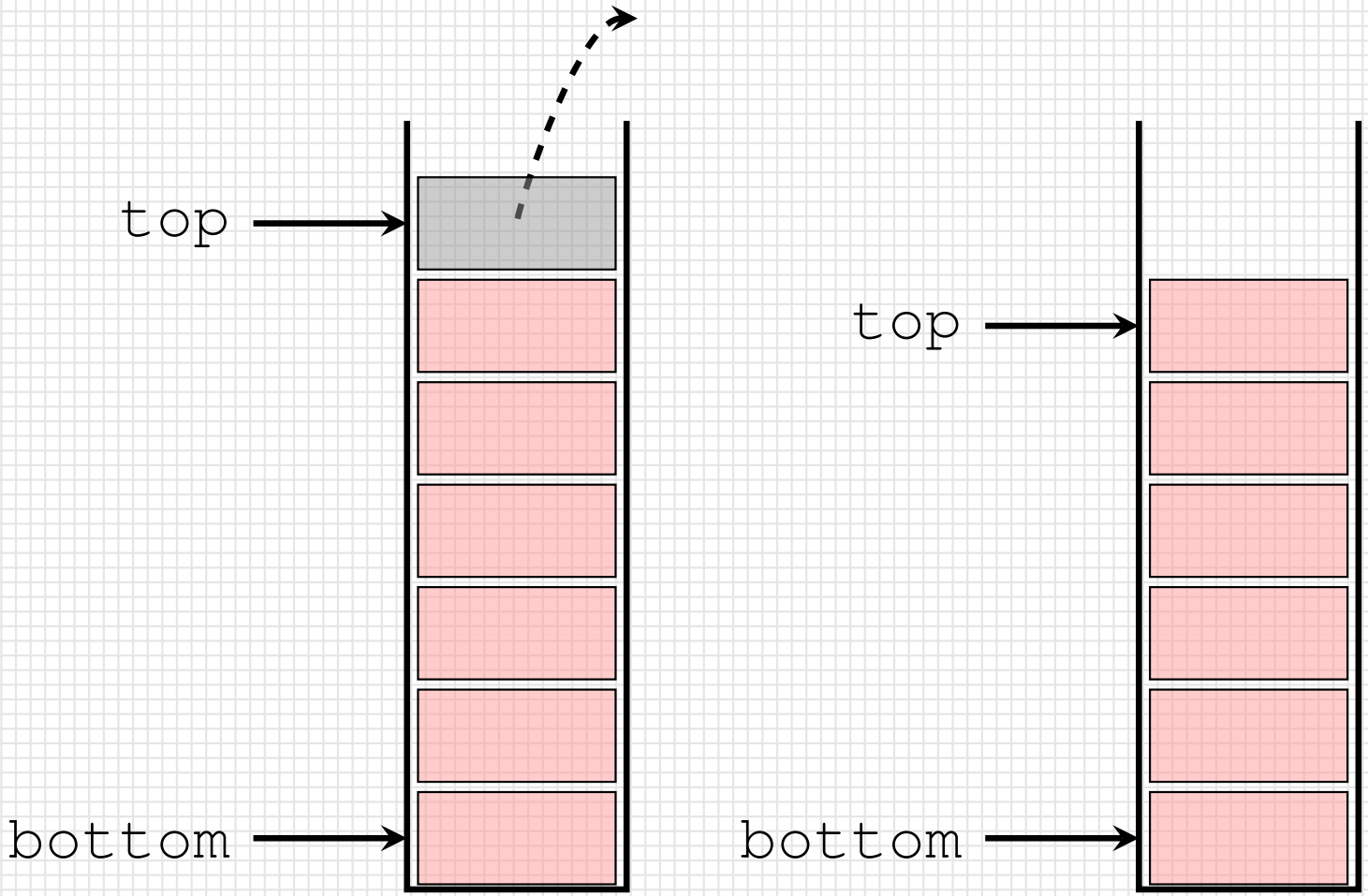


图: 出栈

基本概念和抽象数据类型

```
ADT Stack{
  Data:
  同线性表。元素具有相同的数据类型，相邻元素有前驱、后继关
  系。
  Operation:
  Init(*S):        初始化操作，创建一个空栈 S
  Destroy(*S):     若栈存在，销毁之
  Clear(*S):       将栈清空
  IsEmpty(S):      若栈为空，返回 true；否则返回 false
  GetTop(S,*e):    若栈存在且非空，用 e 返回栈顶元素
  Push(*S,e):      若栈存在，插入新元素 e 并称为栈顶元素
  Pop(*S,*e):      删除栈顶元素，并用 e 返回其值
  Length(S):     返回元素个数
} ADT Stack
```

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

顺序栈

既然栈是线性表的特例，那么栈的顺序存储其实是线性表顺序存储的简化。栈的顺序存储结构简称顺序栈，用数组来实现。

顺序栈

既然栈是线性表的特例，那么栈的顺序存储其实是线性表顺序存储的简化。栈的顺序存储结构简称顺序栈，用数组来实现。

问题　对于栈，用数组哪一端作为栈顶或栈底会比较好？

顺序栈

既然栈是线性表的特例，那么栈的顺序存储其实是线性表顺序存储的简化。栈的顺序存储结构简称顺序栈，用数组来实现。

问题　对于栈，用数组哪一端作为栈顶或栈底会比较好？

下标为 0 的一端作为栈底比较好，因为首元素都存在栈底，变化最小。

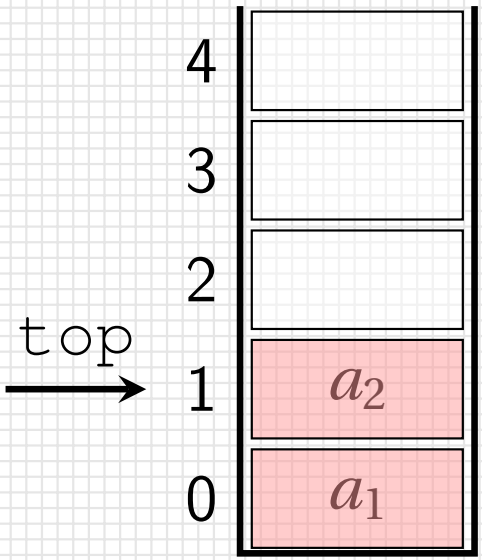
顺序栈

栈的结构定义

```
#define MAXSIZE 100
typedef int ElemType;
typedef struct SqStack{
  ElemType data[MAXSIZE];
  int top;
}SqStack;
```

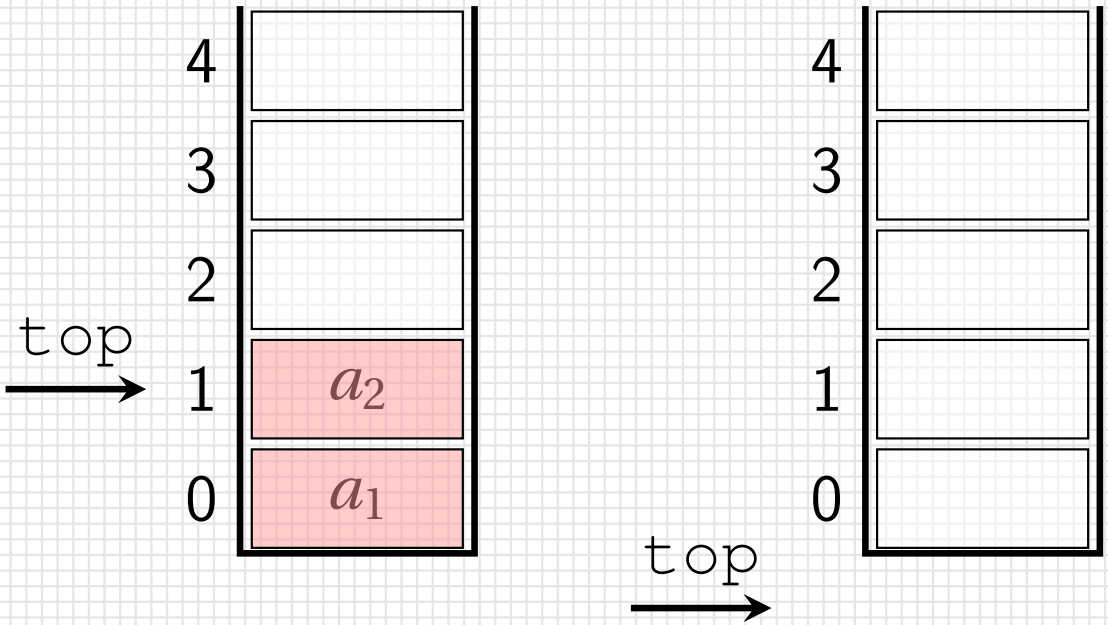
- top 用于指示栈顶元素在数组中的位置，它必须小于存储栈的长度 StackSize;
- 当栈存在一个元素时，top == 0，因此通常把空栈的判定条件定为 top == -1.

顺序栈



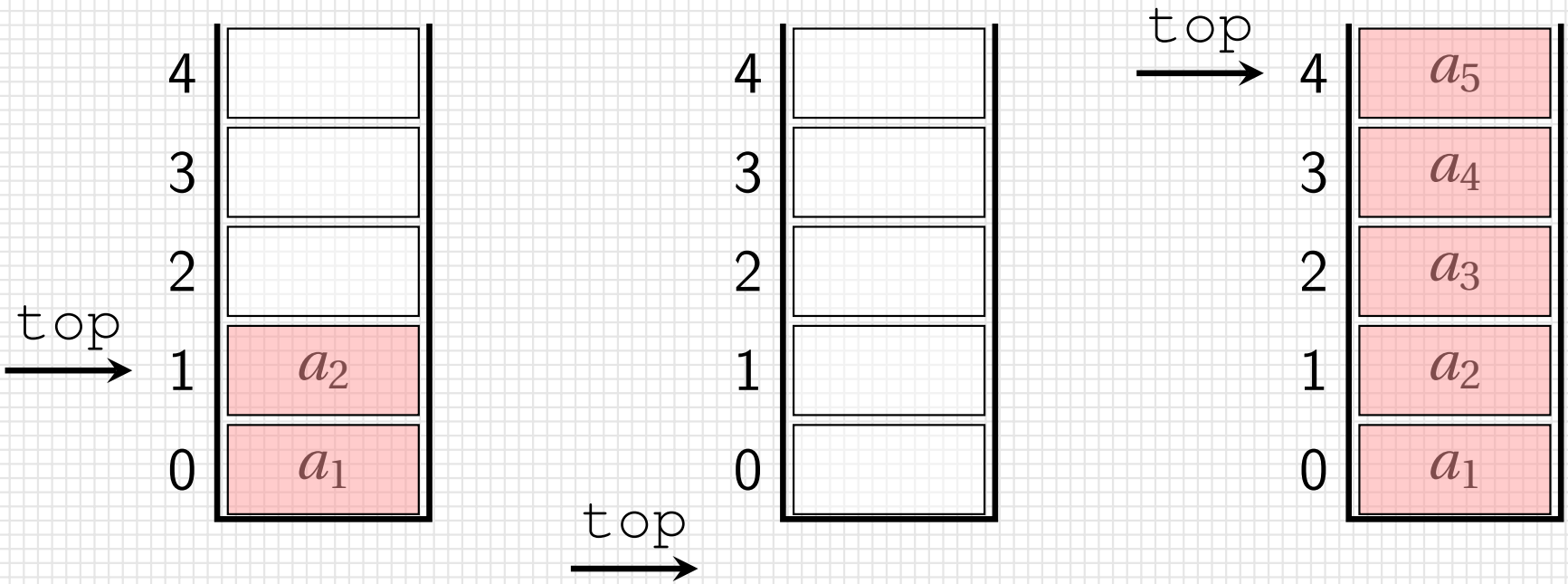
有两个元素,top=1

顺序栈



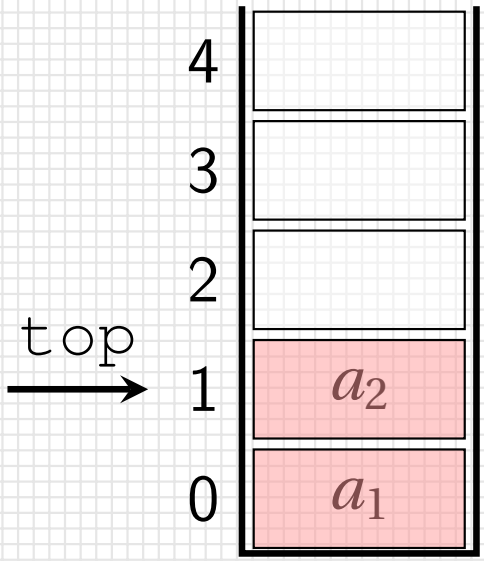
有两个元素,top=1　　空栈,top=-1

顺序栈



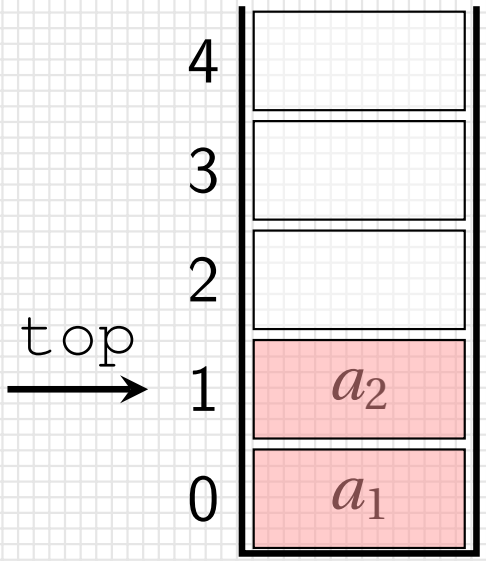
有两个元素,top=1　　**空栈**,top=-1　　**栈满**,top=4

顺序栈

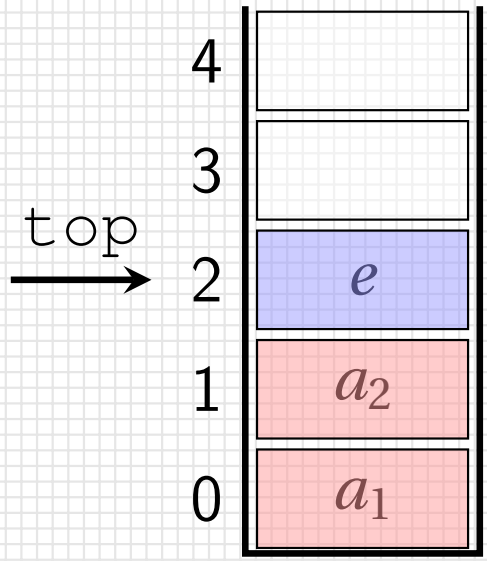


数组 data

顺序栈



数组 data



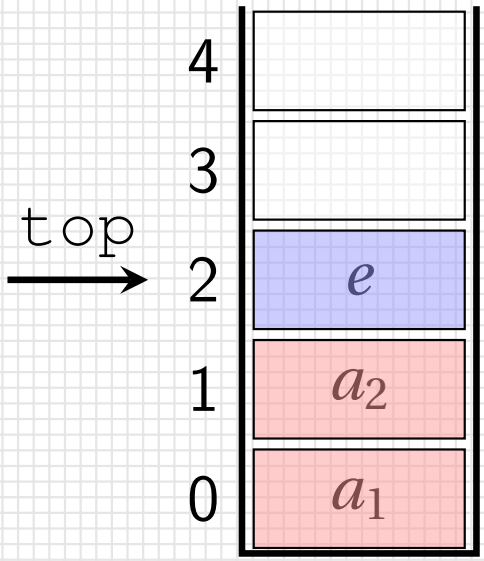
元素 e 进栈

顺序栈

Push.c

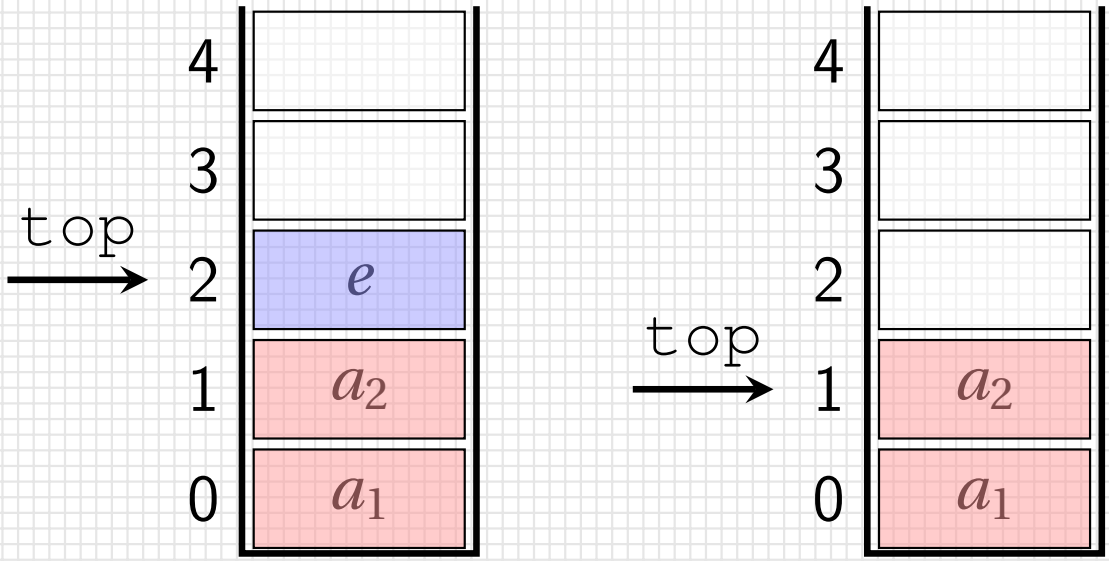
```
#include "SqStack.h"
Status Push(SqStack * S, ElemType e)
{
  if (S->top == MAXSIZE-1)
    return ERROR;
  S->top++;
  S->data[S->top] = e;
  return OK;
}
```

顺序栈



数组 data

顺序栈



数组 data　　元素 e 出栈

顺序栈

Pop.c

```
#include "SqStack.h"
ElemType Pop(SqStack * S)
{
  ElemType  e;
  if (S->top == -1)
    printf("Stack is Empty, Cannot Pop!");
  e = S->data[S->top];
  S->top--;
  return e;
}
```

顺序栈之完整程序

SqStack.h I

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 100
#define ERROR 0
#define OK 1

typedef int Status;
typedef int ElemType;
typedef struct SqStack
{
  ElemType * data;
  int top;
} SqStack;
```

SqStack.h II

```
ElemType Pop   (SqStack * S);
Status Push    (SqStack * S, ElemType e);
Status Init    (SqStack * S);
Status Clear   (SqStack * S);
Status Destroy(SqStack * S);
void PrintS    (SqStack * S);
```

Init.c

```
#include "SqStack.h"
Status Init(SqStack *S)
{
  S->data = (ElemType *) malloc(MAXSIZE * sizeof(
  ElemType));
  if(!S->data){
    printf("Malloc Failed!\n");
    return ERROR;
  }
  S->top = -1;
  return OK;
}
```

PrintS.c

```
#include "SqStack.h"
void PrintS(SqStack * S)
{
  int i;
  if(S->top == -1)
  {
    printf("Stack␣is␣empty!\n\n");
    return;
  }
  for(i = 0; i <= S->top; i++)
    printf("%3d␣", S->data[i]);
  printf("\n\n");
}
```

Push.c

```
#include "SqStack.h"
Status Push(SqStack * S, ElemType e)
{
  if (S->top == MAXSIZE-1)
    return ERROR;
  S->top++;
  S->data[S->top] = e;
  return OK;
}
```

Pop.c

```
#include "SqStack.h"
ElemType Pop(SqStack * S)
{
  ElemType  e;
  if (S->top == -1)
    printf("Stack is Empty, Cannot Pop!");
  e = S->data[S->top];
  S->top--;
  return e;
}
```

Clear.c

```
#include "SqStack.h"
Status Clear(SqStack * S)
{
  ElemType e;
  while(S->top > -1)
    e = Pop(S);
  return OK;
}
```

Destroy.c

```c
#include "SqStack.h"
Status Destroy(SqStack * S)
{
  Clear(S);
  free(S->data);
  return OK;
}
```

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

两栈共享空间

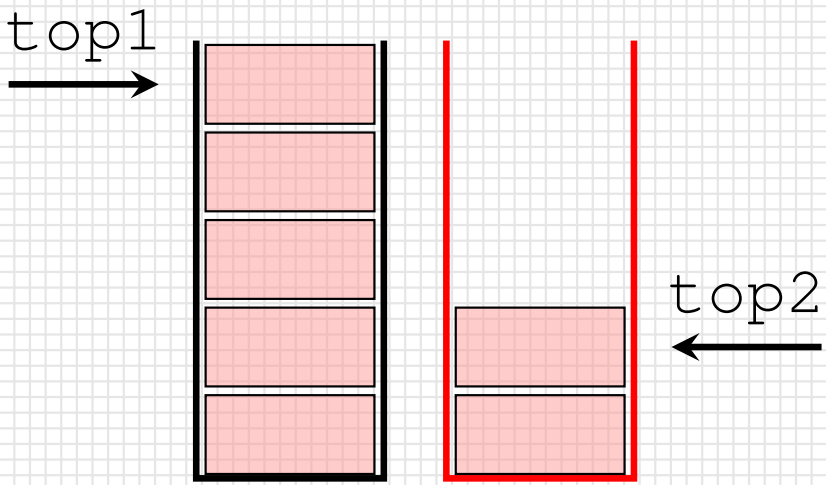
问题　设有两个相同类型的栈，为它们各自开辟数组空间，极有可能第一个栈满，而另一个栈还有很多空闲空间。请问如何更有效地利用存储空间？

两栈共享空间

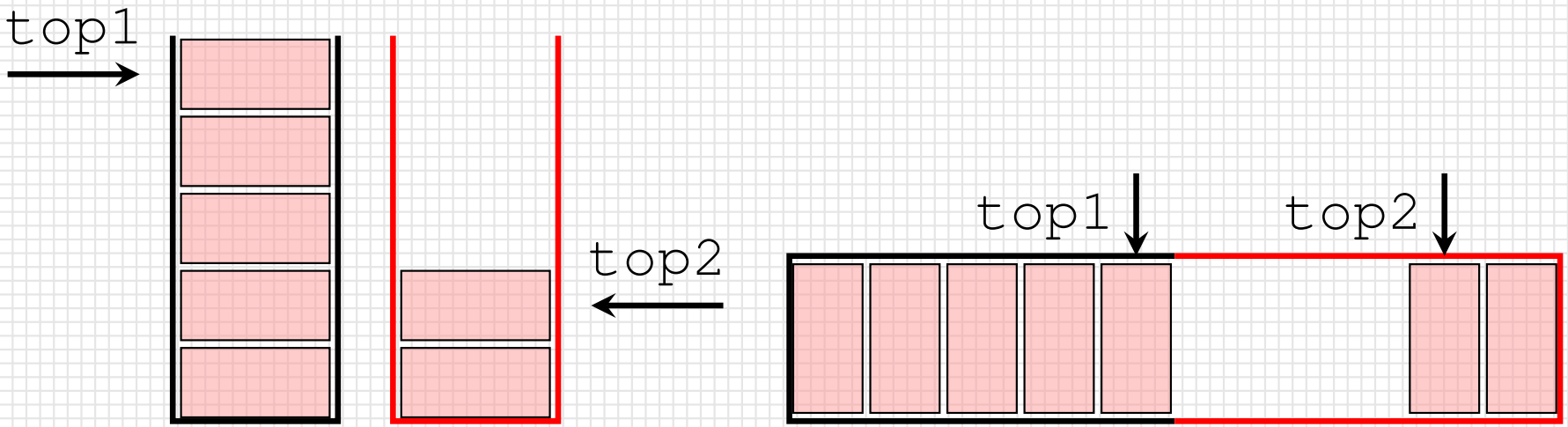
问题　设有两个相同类型的栈，为它们各自开辟数组空间，极有可能第一个栈满，而另一个栈还有很多空闲空间。请问如何更有效地利用存储空间？

可以用一个数组来存储两个栈，实现两栈共享空间。

两栈共享空间



两栈共享空间



两栈共享空间

- 让一个栈的栈底为数组的始端，另一个栈的栈底为数组的末端。这样，两个栈如果增加元素，就是两端点向中间延伸。
- 设 `top1` 和 `top2` 分别是栈 1 和栈 2 的栈顶指针，只要它们俩不碰面，两个栈就可以一直使用。

两栈共享空间

栈满的情形：

- 栈 1 为空 (`top1 == -1`)，栈 1 为满 (`top2 == 0`)
- 栈 2 为空 (`top2 == n`)，栈 1 为满 (`top1 == n-1`)
- 一般情形：两个栈见面即栈满，亦即 `top1+1 == top2`。

两栈共享空间：Share_SqStack.h

```
typedef struct {
  ElemType data[MAXSIZE];
  int top1;
  int top2;
} Share_SqStack;
```

两栈共享空间：Push.c

```
Status Push(SqDoubleStack * S, ElemType e, int
stackNumber)
{
  if(S->top1+1 == S->top2)
    return ERROR;
  if(stackNumber == 1)
    S->data[++S->top1] = e;
  if(stackNumber == 2)
    S->data[--S->top2] = e;
  return OK;
}
```

两栈共享空间：Pop.c

```
Status Pop(SqDoubleStack * S, ElemType * e, int
stackNumber)
{
  if(stackNumber == 1)
  {
    if(S->top1 == -1) return ERROR;
    *e = S->data[S->top1--];
  }
  if(stackNumber == 2)
    if(S->top2 == MAXSIZE) return ERROR;
    *e = S->data[S->top2++];
  return OK;
}
```

1. 栈

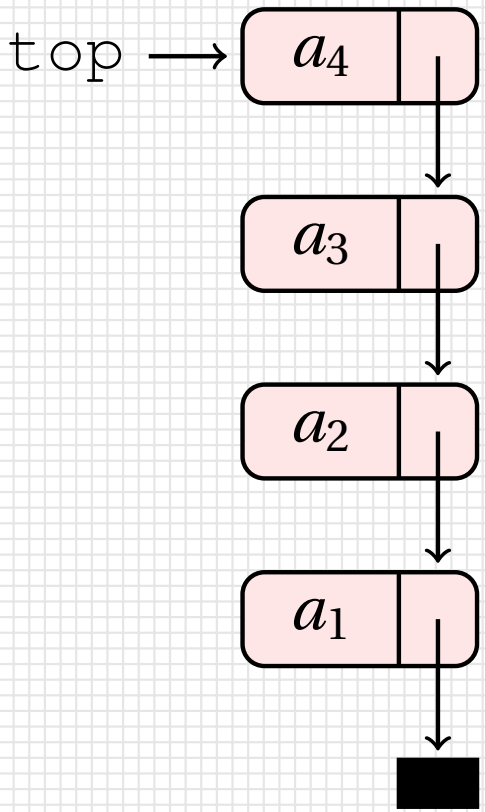
- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

链栈

定义 栈的链式存储结构称为链栈，是操作受限的单链表，其插入和删除操作只能在表头位置上进行。

链栈



非空栈

链栈

- 对于链栈，通常不需要头结点；
- 对于链栈，基本不存在栈满的情况；
- 对于空栈，通常指的是 `top == NULL`.

链栈之完整程序

linkstack.h

```
#include<stdio.h>
#include<stdlib.h>
#define ERROR 0
#define OK 1

typedef int Status;
typedef int ElemType;
typedef struct StackNode
{
  ElemType data;
  struct StackNode * next;
} StackNode, * LinkStackPtr;
```

linkstack.h II

```
typedef struct LinkStack
{
  LinkStackPtr top;
  int count;
} LinkStack;

LinkStack * Init(void);
Status Push  (LinkStack * S, ElemType e);
Status Pop   (LinkStack * S, ElemType * e);
Status PrintS(LinkStack * S);
```

Init.c

```
#include "linkstack.h"
LinkStack * Init(void)
{
  LinkStack * S;
  S = (LinkStack *) malloc(sizeof(LinkStack));
  S->top = NULL;
  return S;
}
```

Push.c

```
#include"linkstack.h"
Status Push(LinkStack * S, ElemType e)
{
  LinkStackPtr s = (LinkStackPtr) malloc(sizeof(
  StackNode));
  if(!s) return ERROR;
  s->data = e;
  s->next = S->top;
  S->top  = s;
  S->count++;
  return OK;
}
```

Pop.c

```
#include "linkstack.h"
Status Pop(LinkStack * S, ElemType * e){
  LinkStackPtr p;
  if (S->top == NULL)
    return ERROR;      //栈空，返回错误标志
  *e = S->top->data;
  p  = S->top;
  S->top = S->top->next;
  free(p);
  S->count--;
  return OK;
}
```

PrintS.c

```
#include "linkstack.h"
Status PrintS(LinkStack * S)
{
  LinkStackPtr p;
  p = S->top;
  while(p != NULL){
    printf("%3d␣", p->data);
    p = p->next;
  }
  printf("\n");
  return OK;
}
```

main.c

```
#include "linkstack.h"
int main(void)
{
  LinkStack *S; ElemType e;

  S=Init();                    PrintS(S);

  Push(S,1);  Push(S,3);
  Push(S,5);  Push(S,7);  PrintS(S);

  Pop(S,&e);  Pop(S,&e);  PrintS(S);

  return 1;
}
```

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

栈的应用

由于栈“后进先出”的固有特性，故栈是程序设计中常用的工具和数据结构。

应用 1：数制转换

十进制整数 n 向其它进制数 $d(=2,8,16)$ 的转换是计算机实现计算的基本问题。

转换法则：

$$n = (n \div d) \times d + r$$

其中 r 为余数，且 $0 \le r < d$。

应用 1：数制转换

例　$(1348)_{10} = (2504)_8$，其运算过程如下：

n	$n \div 8$	r
1348	168	4
168	21	0
21	2	5
2	0	2

应用 1：数制转换（程序）I

```
#include "SqStack.h"
void Convert(int n, int d)
{
  SqStack S;
  int k, * e;
  Init(&S);
  while(n > 0)
  {
    k = n % d;
    Push(&S, k);
    n /= d;
  }
  while (S.top > -1)
```

应用 1：数制转换（程序）II

```
    {
      Pop(&S, e);
      printf("%1d", *e);
    }
  printf("\n");
}
```

应用 2：括号匹配

在文字处理软件或编写程序时，常常需要检查一个字符串或一个表达式终端括号是否相匹配？

应用 2：括号匹配

在文字处理软件或编写程序时，常常需要检查一个字符串或一个表达式终端括号是否相匹配？

匹配思想：从左至右扫描一个字符串（或表达式），则每个右括号将于最近遇到的那个左括号相匹配。可以在从左到右扫描过程中把所遇到的左括号存放到堆栈中，每当遇到一个右括号时，就将它与栈顶的左括号（如果存在）相匹配，同时从栈顶删除该左括号。

应用 2：括号匹配

算法思想：设置一个栈，当读到左括号时，左括号进栈。当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；否则匹配失败，返回 FALSE.

应用 2：括号匹配（程序）I

```
int MatchBrackets (SqStack * S)
{
  char ch, x;
  scanf("%c", &ch);
  while (asc(ch) != 13)
  {
    if ( ( ch == '(' ) || ( ch == '[' ) )
      Push(S, ch);
    else if ( ch == ']' )
    {
      x = Pop(S);
      if ( x != '[' )
      {
```

应用 2：括号匹配（程序）II

```
        printf("[ not matching!\n");
        return FALSE;
      }
    }
    else if ( ch == ')' )
    {
      x = pop(S);
      if( x != '(' )
      {
        printf("( not matching!\n");
        return FALSE;
      }
    }
  }
```

应用 2：括号匹配（程序） III

```
  if (S->top != 0)
  {
    printf("括号数量不匹配");
    return FALSE;
  }
  else return TRUE;
}
```

前缀、中缀、后缀表达式

它们都是对表达式的记法，其区别在于运算符相对操作数的位置不同：

- 前缀表达式：运算符位于其相关操作数的前面；
- 中缀表达式：运算符位于其相关操作数的中间；
- 后缀表达式：运算符位于其相关操作数的后面。

前缀、中缀、后缀表达式

例如：

- 前缀表达式：$- \times + 3\ 4\ 5\ 6$
- 中缀表达式：$(3 + 4) \times 5 - 6$
- 后缀表达式：$3\ 4 + 5 \times 6 -$

前缀、中缀、后缀表达式

中缀表达式： 人们常用的算术表示方法。

中缀表达式很容易被人的大脑理解和分析，但对计算机来说却是很复杂的。因此计算表达式的值时，通常需要将中缀表达式转换为前缀或后缀表达式，然后进行求值。

对计算机来说，计算前缀或后缀表达式的值非常简单。

前缀、中缀、后缀表达式

前缀表达式： 也称“前缀记法、波兰式”，其运算符位于操作数之前。

后缀表达式： 也称“后缀记法、逆波兰式”，其运算符位于操作数之后。

应用 3：后缀表达式的计算机求值

从左到右扫描表达式，

(1) 遇到操作数，将数字入栈；

(2) 遇到运算符，弹出栈顶的两个数；

(3) 将这两个数用运算符做相应计算（次顶元素 op 栈顶元素），并将结果入栈；

(4) 重复 (1)-(3) 直到表达式右端，最后运算出的值即为表达式的结果。

应用 3：后缀表达式的计算机求值

例　以后缀表达式

$$3\ 4\ +\ 5\ \times\ 6\ -$$

为例，详解整个过程。

应用 3：后缀表达式的计算机求值

1. 将 3 和 4 入栈；
2. 遇到 +，弹出 4 和 3，计算 3+4=7，再将 7 入栈；
3. 将 5 入栈；
4. 遇到 *，弹出 5 和 7，计算 7*5=3，再将 35 入栈；
5. 将 6 入栈；
6. 遇到−，弹出 6 和 35，计算 35−6=29，即为最终结果。

应用 4：将中缀表达式转换为后缀表达式

1. 初始化两个栈：运算符栈 `S1` 和存储中间结果的栈 `S2`；
2. 从左到右扫描中缀表达式；
3. 遇到操作数时，将其压入 `S2`；

应用 4：将中缀表达式转换为后缀表达式

4 遇到运算符，比较其与 S1 栈顶运算符的优先级：

4.1 若 S1 为空，或栈顶运算符为“(”，则压入 S1；

4.2 否则，若其优先级高于栈顶运算符，则压入 S1；

4.3 否则，将 S1 的栈顶运算符弹出并压入到 S2 中，再次转到 4.1 与 S1 中新的栈顶运算符做比较；

应用 4：将中缀表达式转换为后缀表达式

5 遇到括号时，

5.1 如果是“(”，则压入 `S1`；

5.2 如果是“)”，则依次弹出 `S1` 的栈顶元素，并压入 S2，直到遇到“(”为止，此时这一对括号丢弃；

应用 4：将中缀表达式转换为后缀表达式

6 重复步骤 2-5，直到表达式的最右端；

7 将 S1 中剩余的运算符依次弹出并压入 S2；

8 依次弹出 S2 中的元素并输出，结果的逆序即为对应的后缀表达式。

应用 4：将中缀表达式转换为后缀表达式

表: 中缀表达式 1 + ((2 + 3) + 4) * 5

扫描到的字符	S2(栈底-> 栈顶)	S1(栈底-> 栈顶)
1	1	
+	1	+
(	1	+(
(	1	+((
2	12	+((
+	12	+((+
3	123	+((+
)	123+	+(
+	123+	+(+
4	123+4	+(+
)	123+4+	+
*	123+4	+*
5	123+45	+*
	123+45*+	

应用 4：将中缀表达式转换为后缀表达式

表: 中缀表达式 1 + ((2 + 3) * 4) - 5

扫描到的字符	S2(栈底-> 栈顶)	S1(栈底-> 栈顶)
1	1	
+	1	+
(	1	+(
(	1	+((
2	12	+((
+	12	+((+
3	123	+((+
)	123+	+(
*	123+	+(*
4	123+4	+(+
)	123+4*	+
-	123+4+	-
5	123+4+5	-
	123+4+5-	

1. 栈

- 基本概念和抽象数据类型
- 顺序栈
- 两栈共享空间
- 链栈
- 栈的应用

2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

1. 栈

2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

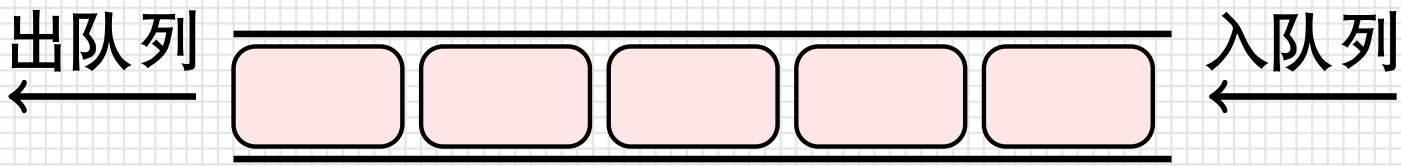
基本概念和抽象数据类型

定义 队列 (Queue) 是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

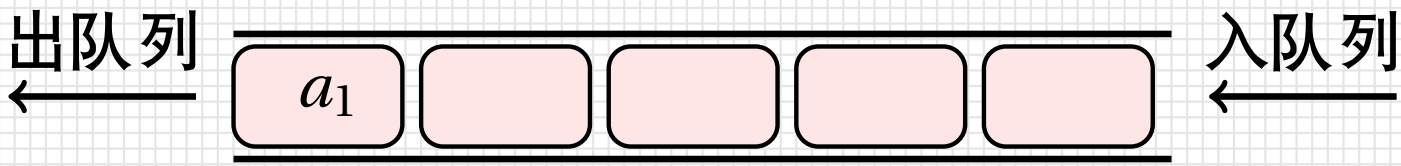
基本概念和抽象数据类型

- 队列是一种先进先出 (First in First Out, FIFO) 的线性表。
- 允许插入的一端叫队尾，允许删除的一端叫队头。

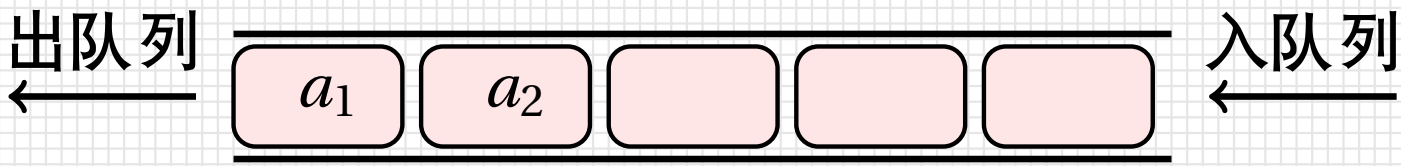
基本概念和抽象数据类型



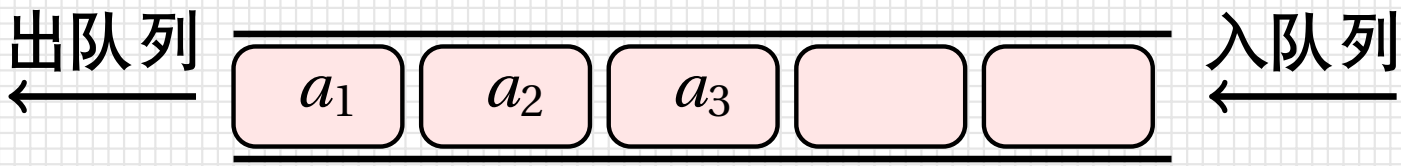
基本概念和抽象数据类型



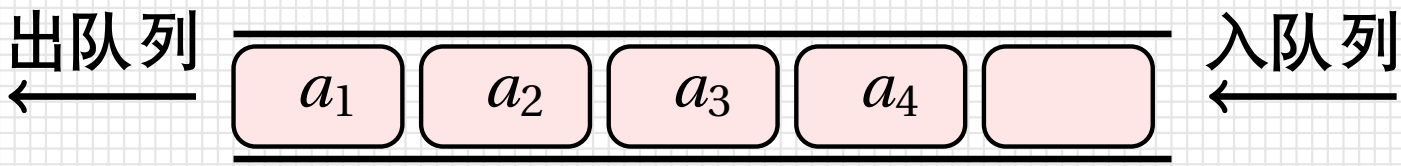
基本概念和抽象数据类型



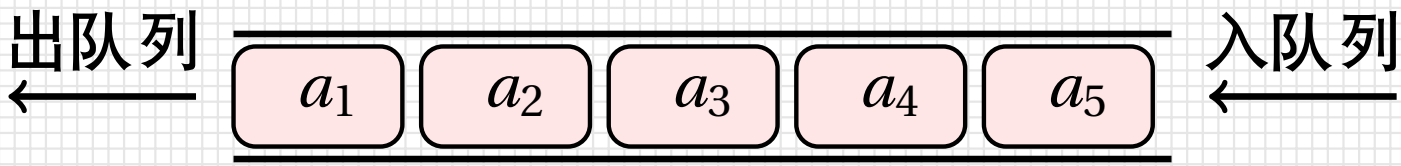
基本概念和抽象数据类型



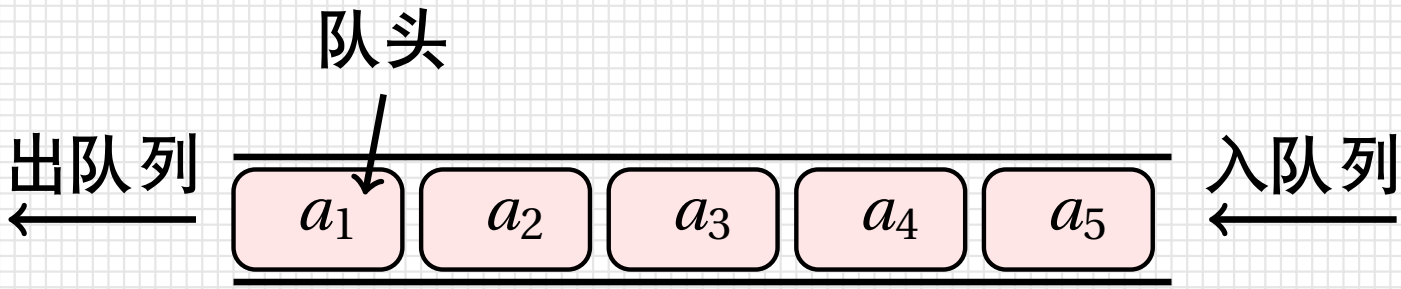
基本概念和抽象数据类型



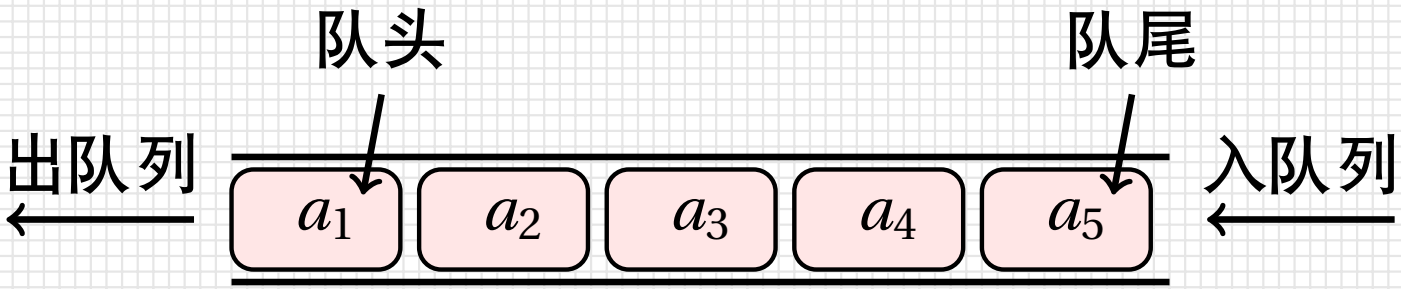
基本概念和抽象数据类型



基本概念和抽象数据类型



基本概念和抽象数据类型



基本概念和抽象数据类型

ADT 队列(Queue)

Data

同线性表。元素具有相同的类型，相邻元素具有前驱和后继关系

Operation:

Init(*Q):

Destroy(*Q):

Clear(*Q):

IsEmpty(Q):

GetHead(Q, *e):

Enter(*Q, e):

Delete(*Q, *e):

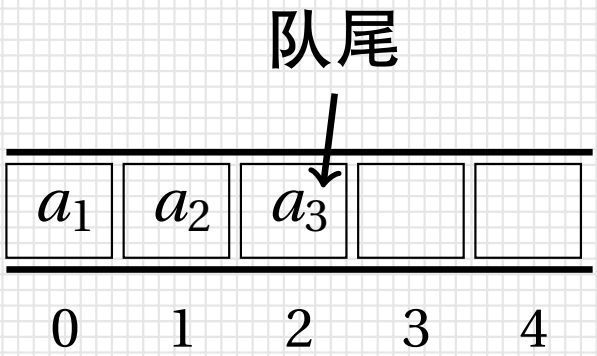
Length(Q):

1. 栈

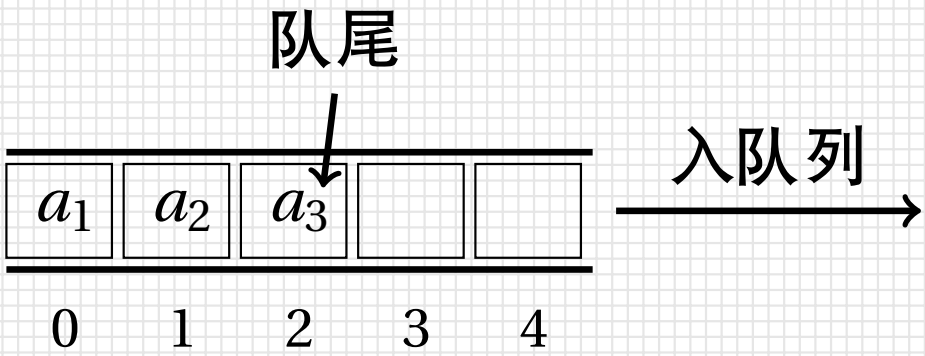
2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

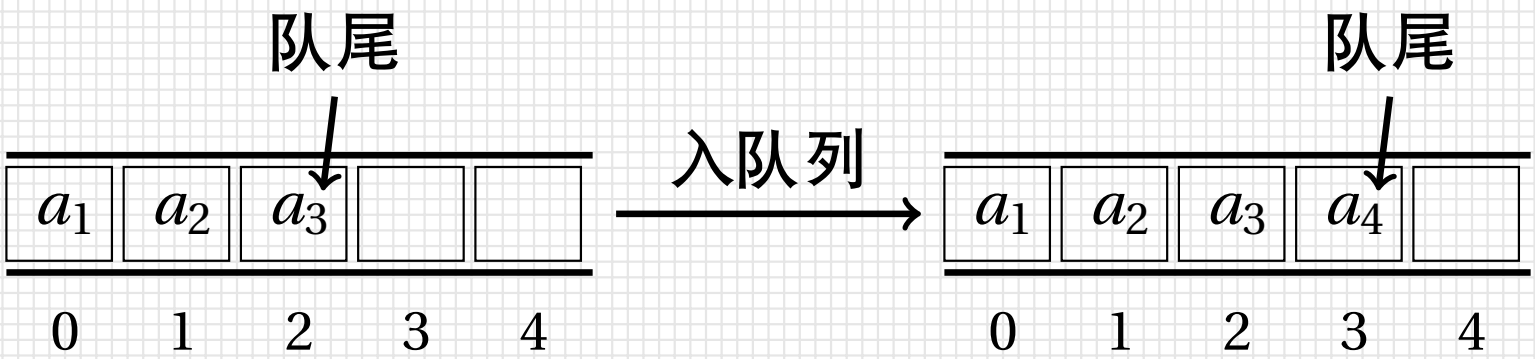
循环队列



循环队列



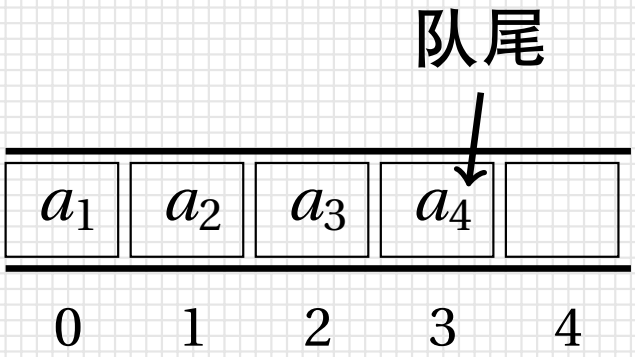
循环队列



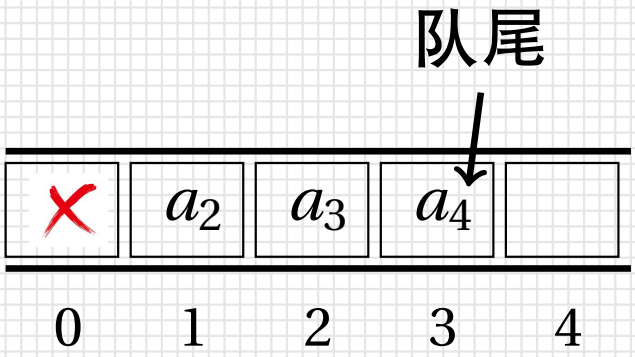
循环队列

注　入队列操作，就是在队尾追加一个元素，不需要移动任何元素，时间复杂度为 $O(1)$.

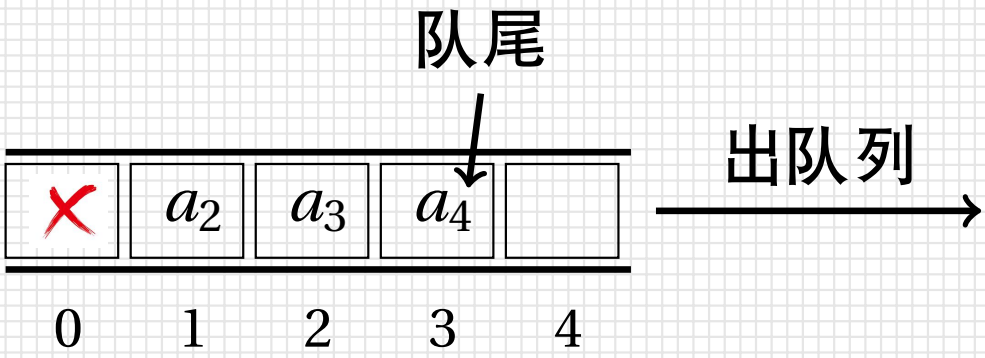
循环队列



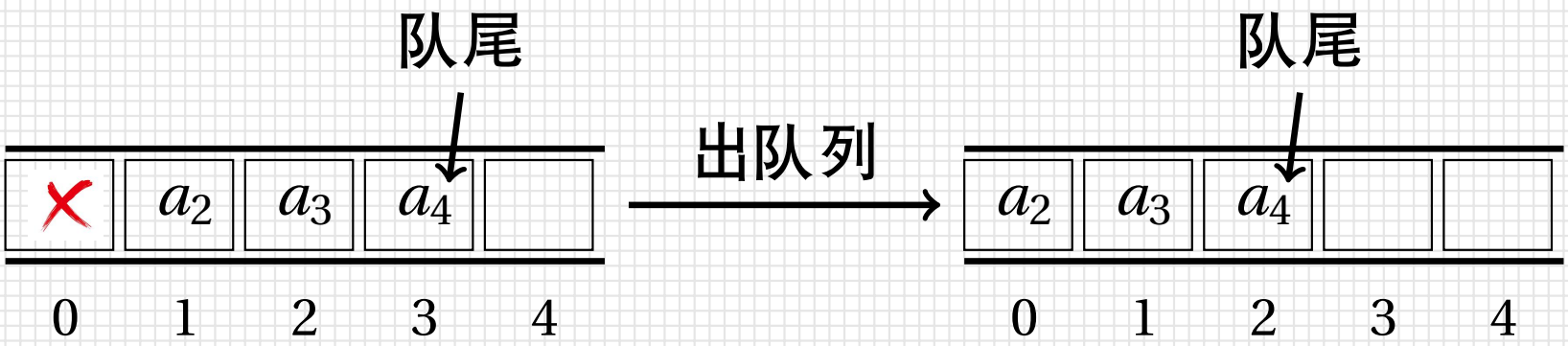
循环队列



循环队列



循环队列



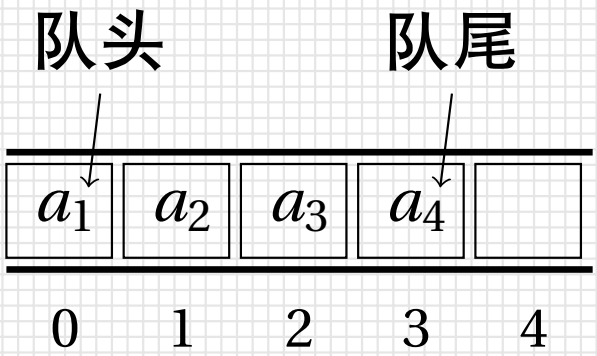
循环队列

注　出队列操作，就是删除队头元素，同时其他元素向前移动，时间复杂度为 $O(n)$.

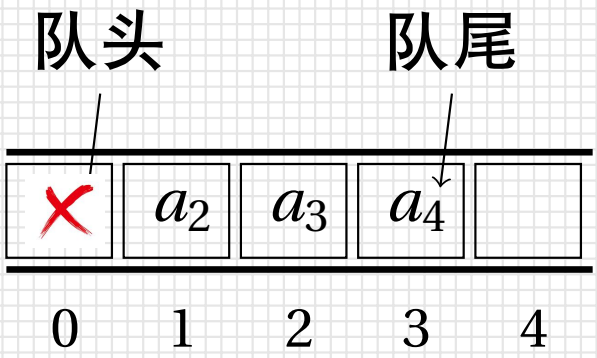
循环队列

若不想移动其他元素，可不限制队列元素必须存储在数组的前 n 个位置这一条件，即队头不一定要在下标为 0 的位置。

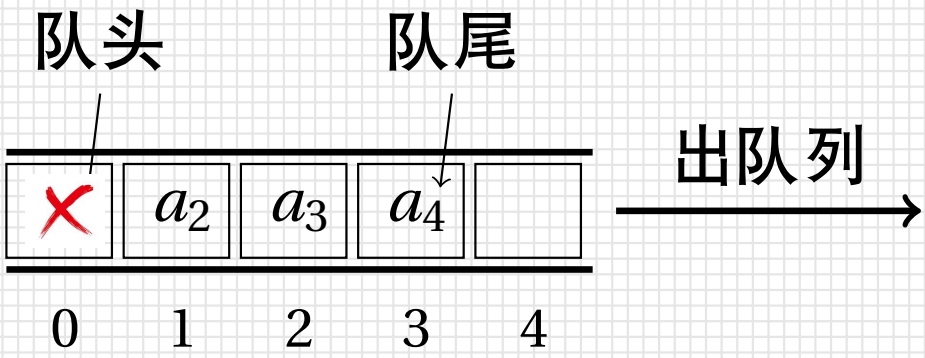
循环队列



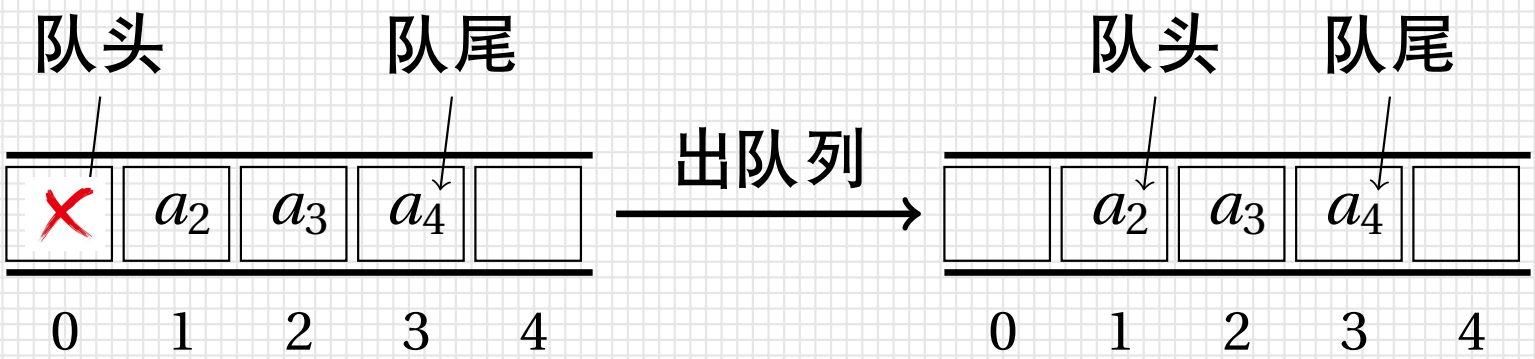
循环队列



循环队列



循环队列



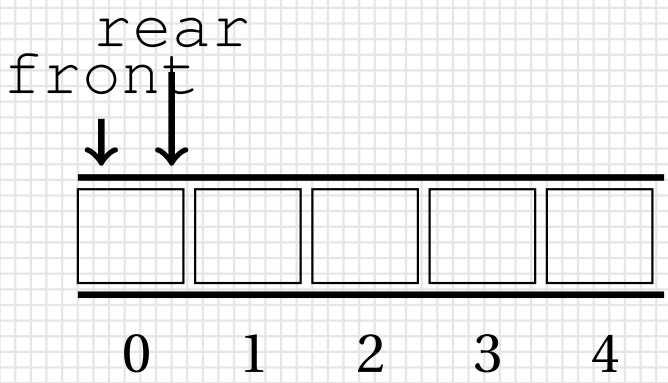
循环队列

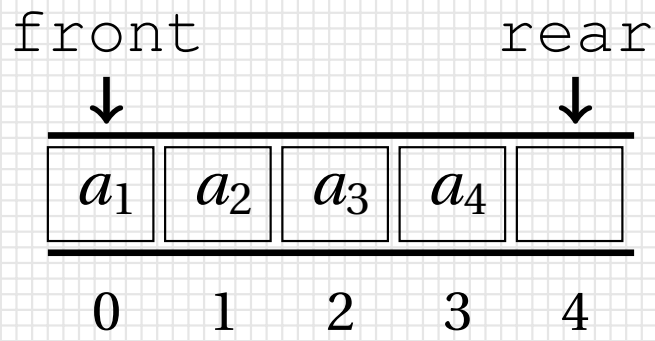
为避免只有一个元素时，队头和队尾重合使处理变得麻烦，可引入两个指针 `front` 和 `rear`，其中

- `front` 指向队头元素，
- `rear` 指向队尾元素的下一个位置，

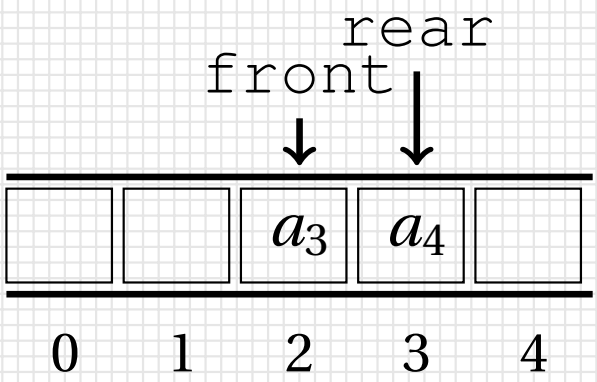
这样当 `front == rear` 时，为空队列。

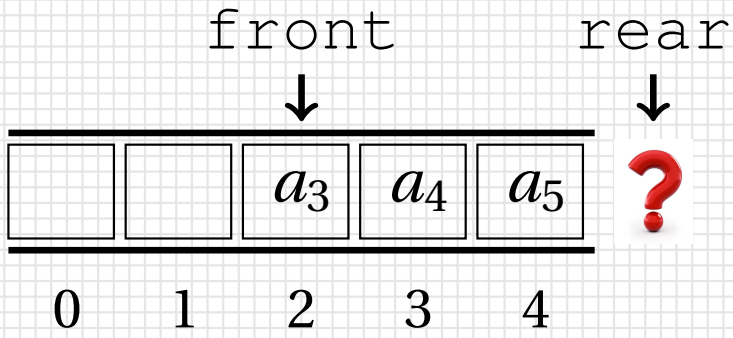
循环队列



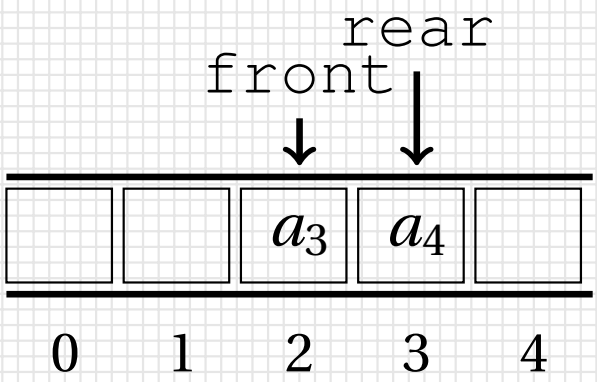


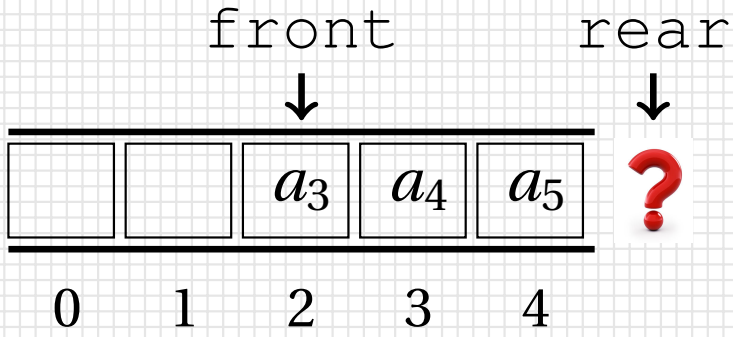
循环队列





循环队列





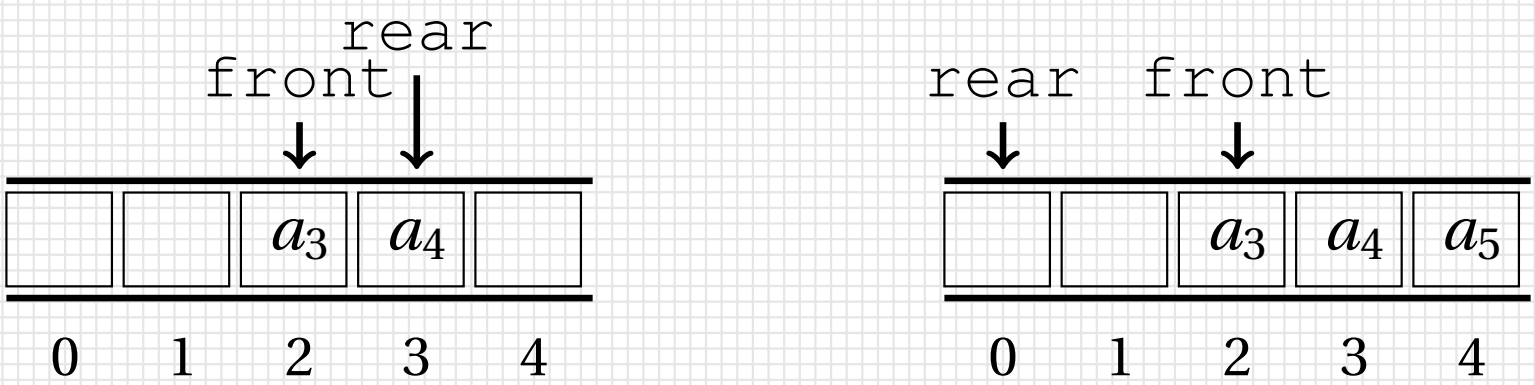
- a_1, a_2 出队，`front` 指向下标为 2 的位置，`rear` 不变；
- 接着 a_5 入队，`front` 不变，`rear` 移动到数组之外；
- 若接着入队，会产生数组越界的错误。可实际上，前两个位置还空闲，这种现象称为“假溢出”。

循环队列

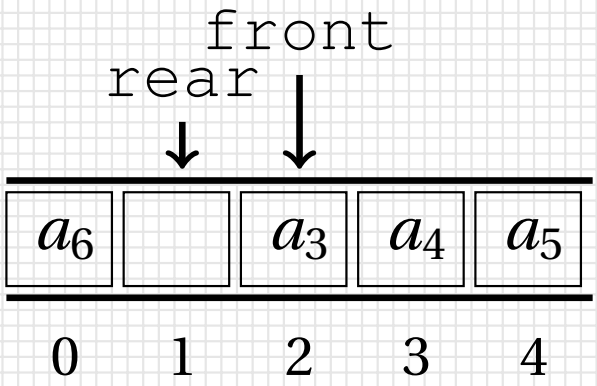
定义 头尾相连的队列顺序存储结构称为循环队列。

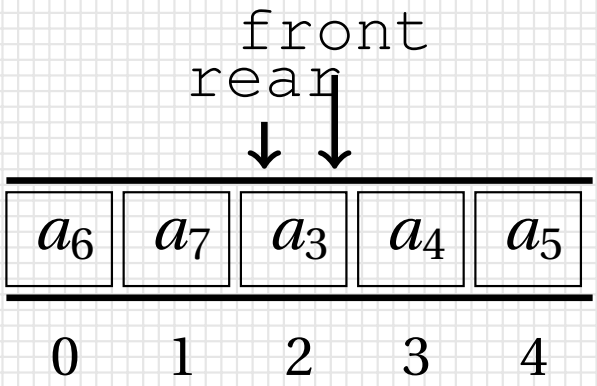
循环队列

将上图中的 `rear` 指向下标为 0 的位置，就不会造成指针指向不明的问题。

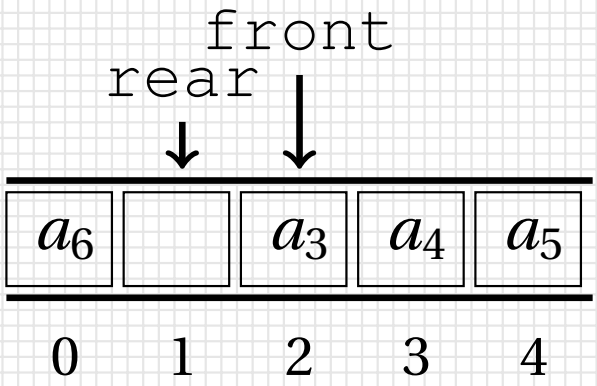


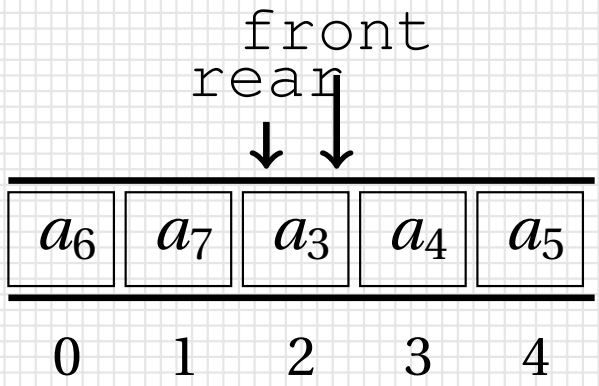
循环队列





循环队列





- 接着 a_6 入队，放置于下标为 0 处，而 `rear` 指向下标为 1 处。
- 再让 a_7 入队，则 `rear` 与 `front` 重合。

循环队列

问题 空队列时，`front==rear`，现在队列已满，仍然有`front==rear`，如何判断队列是空是满？

循环队列

问题　空队列时，front==rear，现在队列已满，仍然有front==rear，如何判断队列是空是满？

(1) 设置一个标志变量 flag，当 front == rear && flag == 0 时队列为空；而当 front == rear && flag==1 时队列为满。

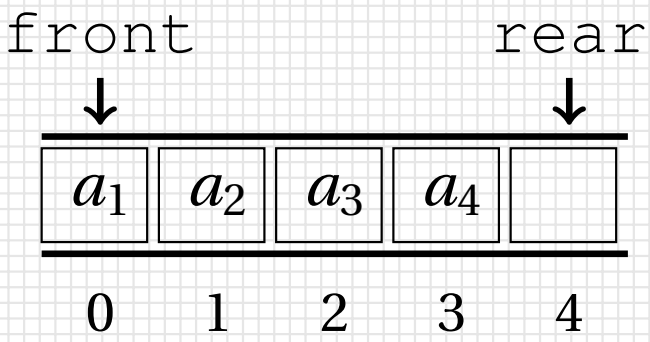
循环队列

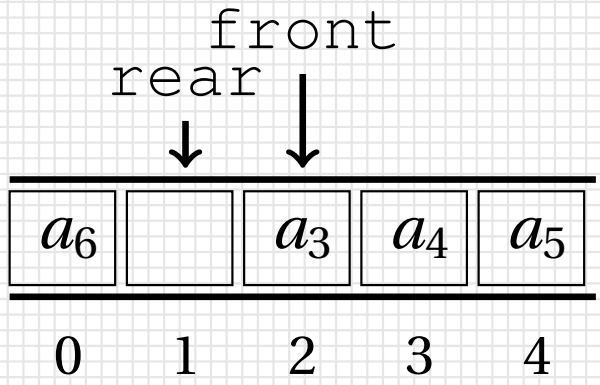
问题　空队列时，front==rear，现在队列已满，仍然有front==rear，如何判断队列是空是满？

(1) 设置一个标志变量 flag，当 front == rear && flag == 0 时队列为空；而当 front == rear && flag==1 时队列为满。

(2) 当队列为空时，条件就为 front == rear；当队列满时，修改其条件，让数组中始终保留一个空闲单元。

循环队列





循环队列

问题　对于第二种办法，由于 `rear` 可能比 `front` 大，也可能比 `front` 小，如何判定队列是否为满？

循环队列

问题　对于第二种办法，由于 `rear` 可能比 `front` 大，也可能比 `front` 小，如何判定队列是否为满？

设队列的最大尺寸为 `MAXSIZE`，则队列满的条件是

```
(rear + 1) % MAXSIZE == front
```

循环队列

问题　如何计算队列的长度？

循环队列

问题 如何计算队列的长度？

当 `rear > front` 时，队列长度为 `rear - front`；

循环队列

问题 如何计算队列的长度？

当 rear > front 时，队列长度为 rear - front；

当 rear < front 时，队列长度分为两段，一段为 MAXSIZE - front，另一段为 rear - 0，合计为 rear - front + MAXSIZE。

循环队列

问题 如何计算队列的长度？

当 rear > front 时，队列长度为 rear - front；

当 rear < front 时，队列长度分为两段，一段为 MAXSIZE - front，另一段为 rear - 0，合计为 rear - front + MAXSIZE。

故通用的队列长度计算公式为

```
(rear - front + MAXSIZE) / MAXSIZE
```

循环队列

循环队列之完整程序

SqQueue.h

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 100
#define ERROR 0
#define OK 1

typedef int Status;
typedef int ElemType;
typedef struct
{
  ElemType data[MAXSIZE];
  int front;
  int rear;
} SqQueue;
```

Init.c

```
#include "SqQueue.h"
Status Init(SqQueue *Q)
{
  Q->front = 0;
  Q->rear  = 0;
  return OK;
}
```

Length.c

```
#include "SqQueue.h"
int Length(SqQueue *Q)
{
  int len = (Q->rear - Q->front + MAXSIZE) % MAXSIZE;
  printf("Length of Queue is %3d.\n", len);
  return len;
}
```

Enter.c

```
#include "SqQueue.h"
int Enter(SqQueue * Q, ElemType e)
{
  if( (Q->rear + 1) % MAXSIZE == Q->front )
    return ERROR;
  Q->data[Q->rear] = e;
  Q->rear = (Q->rear+1) % MAXSIZE;
  return OK;
}
```

Exit.c

```
#include "SqQueue.h"
int Exit(SqQueue * Q, ElemType * e)
{
  if(Q->front == Q->rear)
    return ERROR;
  *e = Q->data[Q->front];
  Q->front = (Q->front+1) % MAXSIZE;
  return OK;
}
```

1. 栈

2. 队列

- 基本概念和抽象数据类型
- 循环队列
- 链队列

链队列

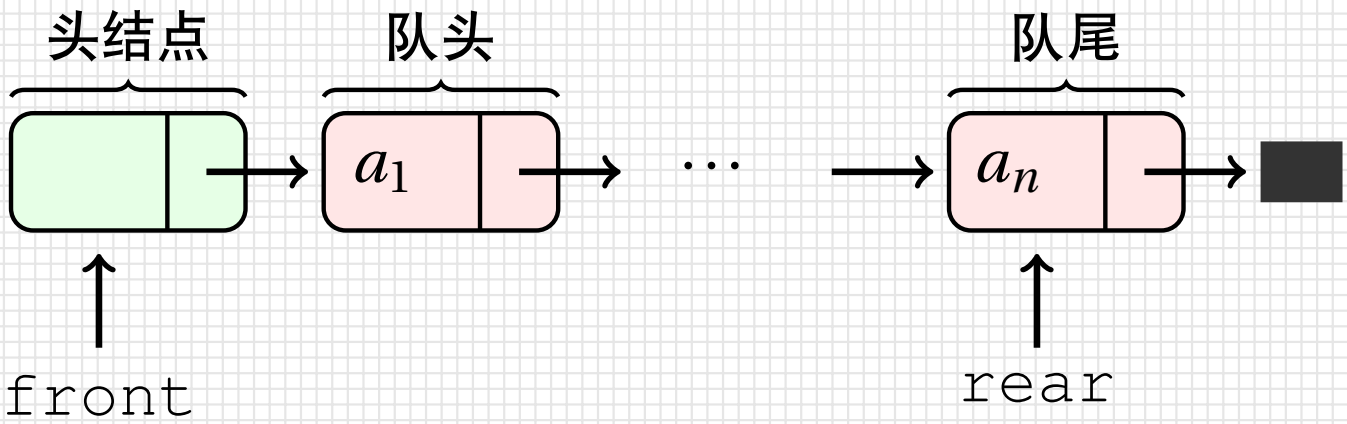
定义 队列的链式存储结构，就是线性表的单链表，只不过它只能尾进头出，简称为链队列。

链队列

注　为操作方便，通常将队头指针指向链队列的头结点，而队尾指针指向终端结点。

链队列

注 为操作方便，通常将队头指针指向链队列的头结点，而队尾指针指向终端结点。

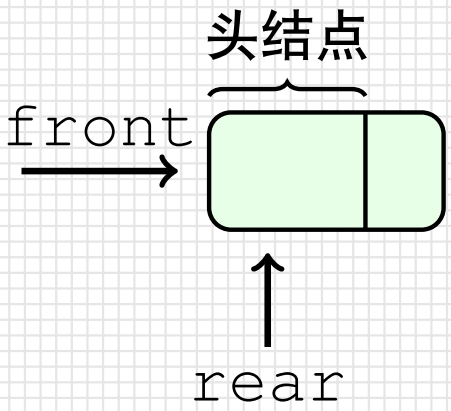


链队列

注 队列为空时，`front` 和 `rear` 都指向头结点。

链队列

注 队列为空时，`front` 和 `rear` 都指向头结点。



LinkQueue.h I

Listing:

```
#include<stdio.h>
#include<stdlib.h>
#define OK 1
#define ERROR 0

typedef int ElemType;
typedef struct QNode
{
  ElemType data;
  struct QNode * next;
} QNode, * QueuePtr;
```

LinkQueue.h II

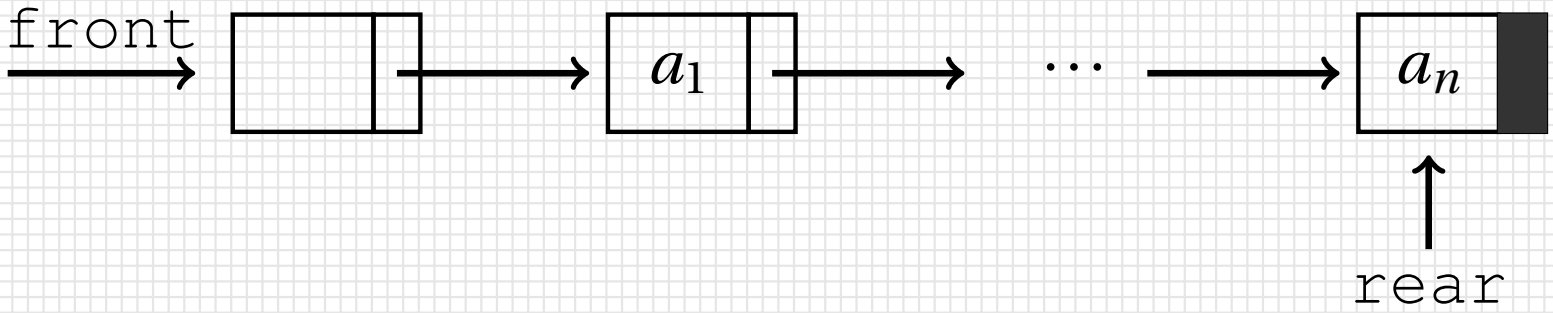
```
typedef struct LinkQueue
{
  QueuePtr front, rear;
} LinkQueue;
```

链队列

入队操作，就是在链表尾部插入结点。

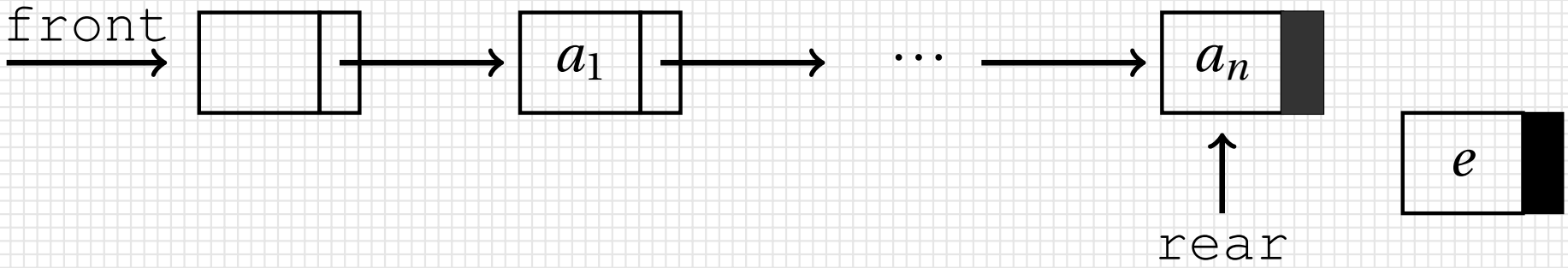
链队列

入队操作，就是在链表尾部插入结点。



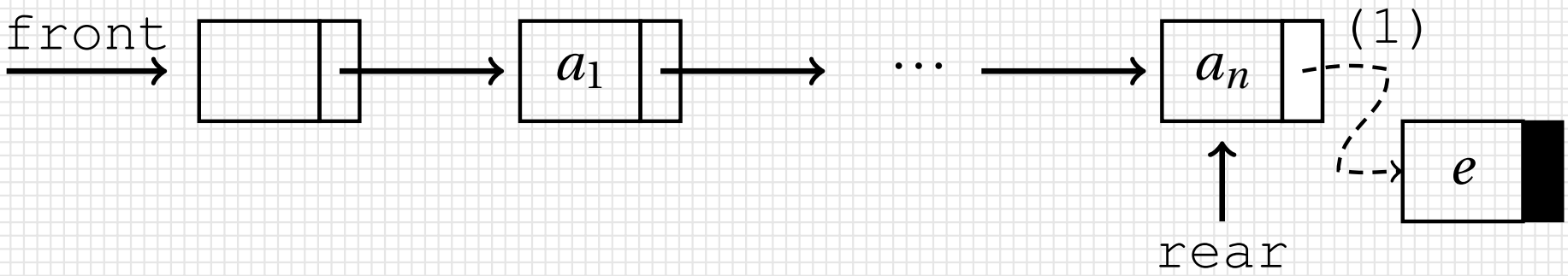
链队列

入队操作，就是在链表尾部插入结点。



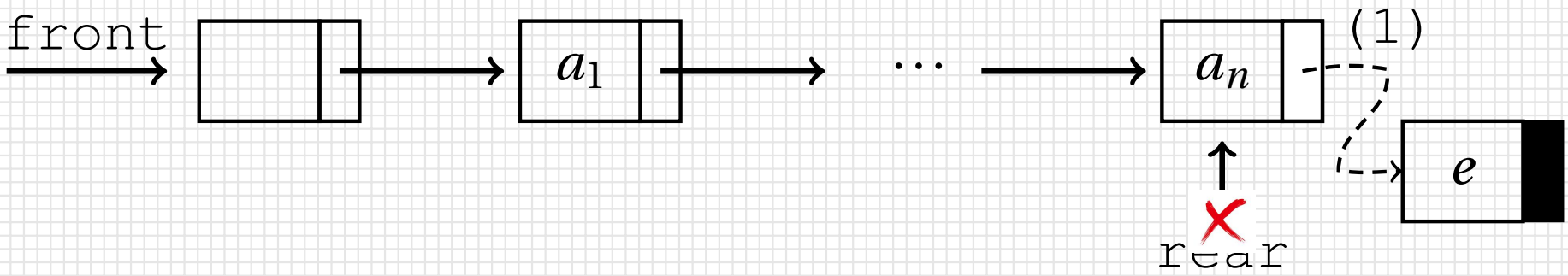
链队列

入队操作，就是在链表尾部插入结点。



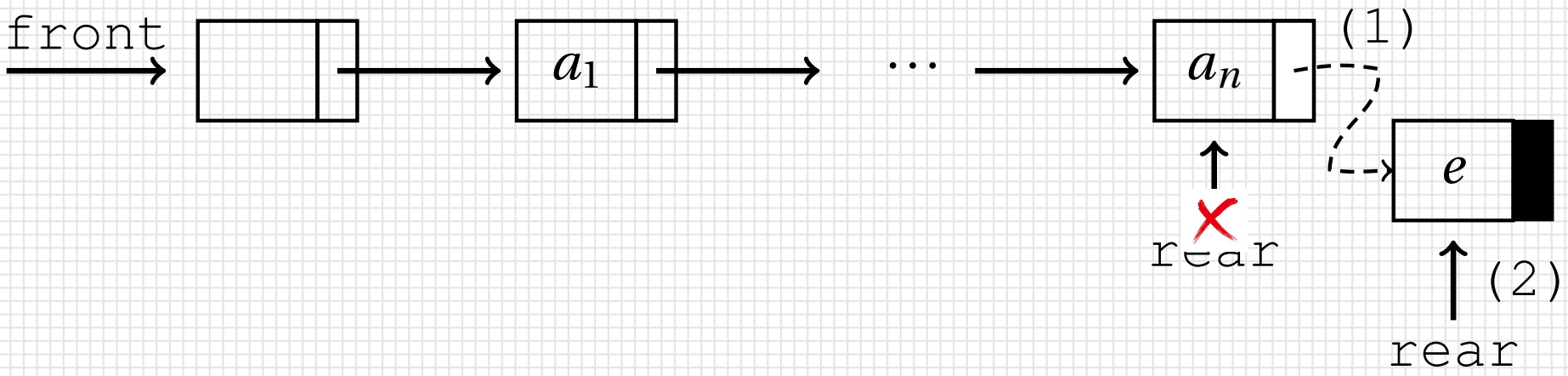
链队列

入队操作，就是在链表尾部插入结点。



链队列

入队操作，就是在链表尾部插入结点。



Enter.c

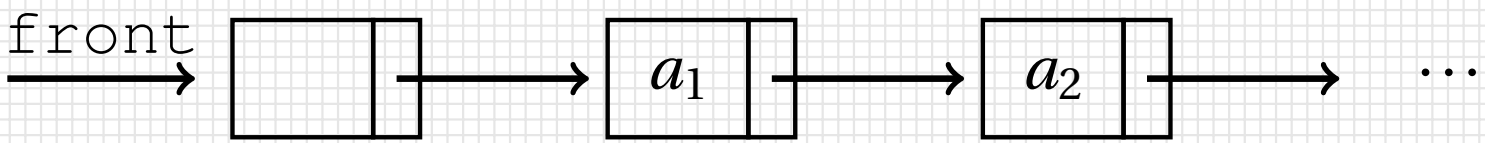
```
#include "LinkQueue.h"
Status Enter(LinkQueue * Q, ELemType e)
{
  QueuePtr s = (QueuePtr) malloc(sizeof(QNode));
  if(!s) return ERROR;    // 入队列
  s->data = e;
  s->next = NULL;
  Q->rear->next = s;
  Q->rear = s;
  return OK;
}
```

链队列

出队操作，就是头结点的后继结点出队，将头结点的后继改为它后面的结点。

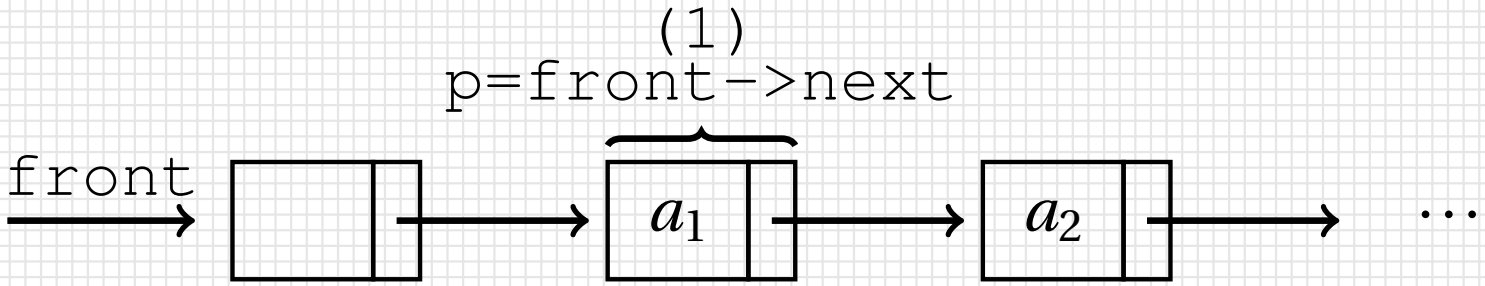
链队列

出队操作，就是头结点的后继结点出队，将头结点的后继改为它后面的结点。



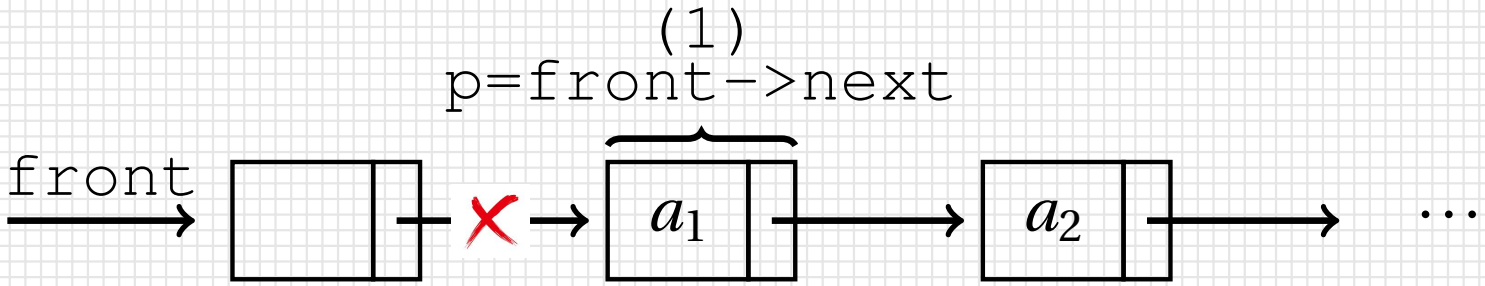
链队列

出队操作，就是头结点的后继结点出队，将头结点的后继改为它后面的结点。



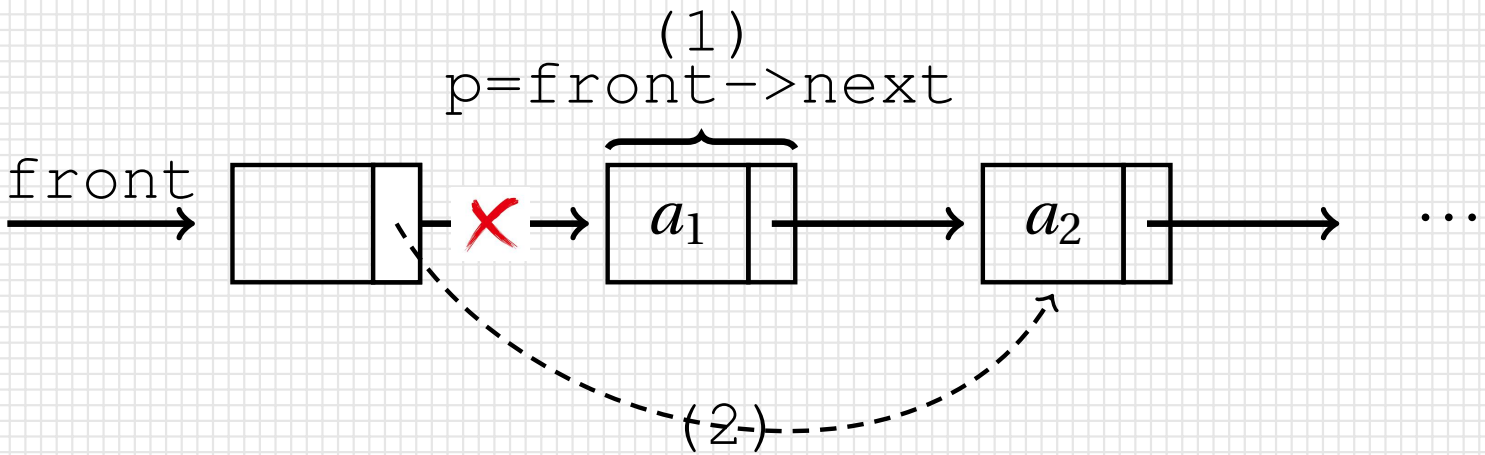
链队列

出队操作，就是头结点的后继结点出队，将头结点的后继改为它后面的结点。



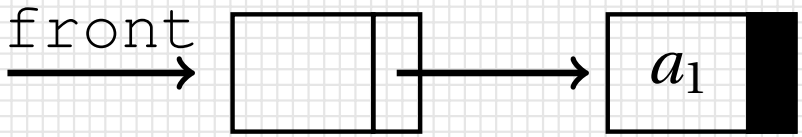
链队列

出队操作，就是头结点的后继结点出队，将头结点的后继改为它后面的结点。



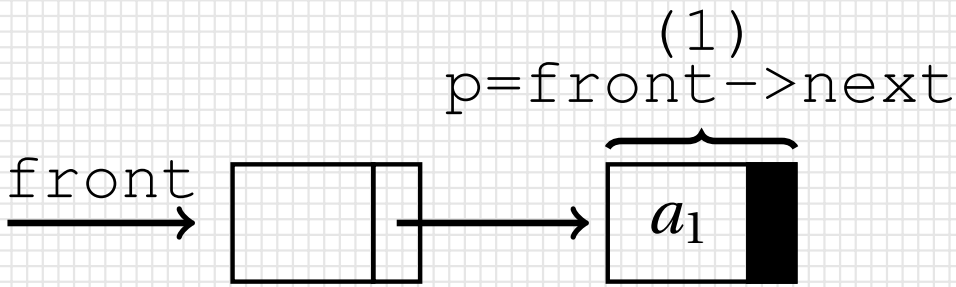
链队列

若链表处头结点外只剩一个元素时，则需将 `rear` 指向头结点。



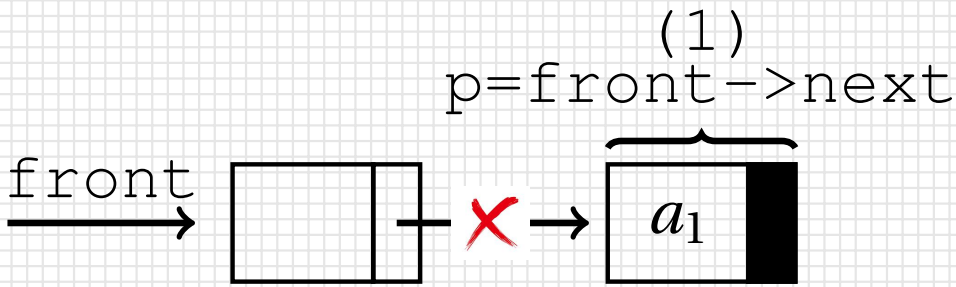
链队列

若链表处头结点外只剩一个元素时，则需将 `rear` 指向头结点。



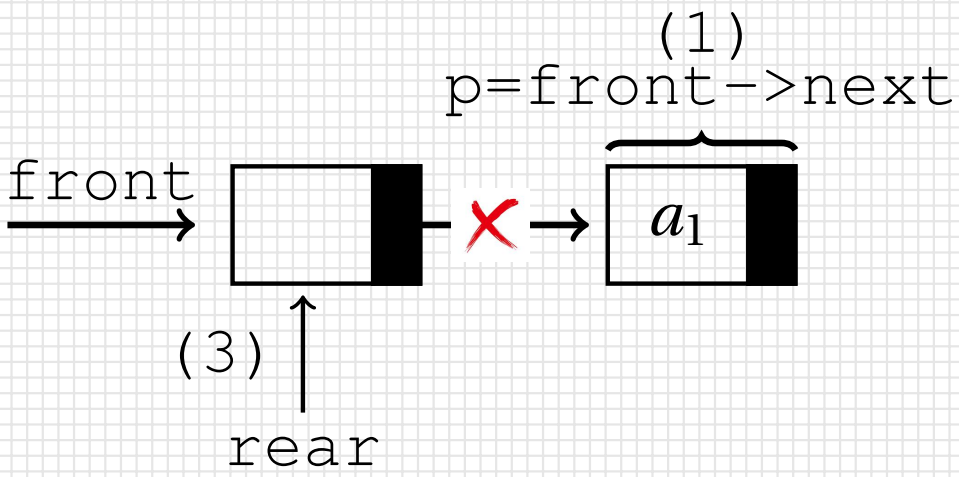
链队列

若链表处头结点外只剩一个元素时，则需将 `rear` 指向头结点。



链队列

若链表处头结点外只剩一个元素时，则需将 `rear` 指向头结点。



Exit.c

```
#include "LinkQueue.h"
Status Exit(LinkQueue * Q, ELemType * e)
{
  QueuePtr p;
  if(Q->front == Q->rear)
    return ERROR;
  p = Q->front->next;
  *e = p->data;
  Q->front->next = p->next;
  if(Q->rear == p)
    Q->rear = Q->front;
  free(p);
  return OK;
}
```

链队列

循环队列和链队列的比较：

- 从时间上看，其基本操作的复杂度均为 $O(1)$，不过循环队列事先申请好空间，使用期间不释放，而对于链队列，每次申请和释放结点会存在一些时间开销。若入队出队频繁，两者会有细微差异。
- 从空间上看，循环队列须有一个固定长度，故存在存储元素个数和空间浪费的问题。而链队列不存在该问题，尽管它需要一个指针域，会产生一些空间上的开销，但也可以接受。所以在空间上，链表更加灵活。

链队列

- 在可以确定队列长度最大值的情况下，建议使用循环队列；
- 若无法预估队列长度，请使用链队列。