



数据结构与算法

Data Structures and Algorithms

谢昊

xiehao@cuz.edu.cn

第一章

线性结构 Linear Structures

1. 引例

2. 线性结构

顺序存储与运算实现

链式存储与运算实现

3. 常用的线性结构

栈

队列

4. 小结

引例

引例：一元多项式的表示与计算

- 考察一元 n 次多项式

$$f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

- 如何表示 $f(x)$?
 - 多项式的阶数 n
 - 各项系数 a_k 与指数 k , 其中 $k \in \mathbb{Z} \cap [0, n]$
- 如何对 $f(x)$ 与 $g(x)$ 两个多项式做基本运算?
 - $f(x) \pm g(x)$
 - $f(x) \cdot g(x)$

解决方案甲：利用顺序存储结构直接表示

- 利用数组元素与下标分别表示多项式中对应各项的系数与指数
- 即：令数组元素 $a[k]$ 表示 x^k 前的系数 a_k
- 简单四则运算只需在数组对应元素之间运算即可

解决方案甲：利用顺序存储结构直接表示

- 利用数组元素与下标分别表示多项式中对应各项的系数与指数
- 即：令数组元素 $a[k]$ 表示 x^k 前的系数 a_k
- 简单四则运算只需在数组对应元素之间运算即可

例

- 多项式 $f(x) = 1 - 3x^2 + 4x^6$ 与 $g(x) = x + 5x^2 - 7x^4$ 可分别由数组 $a[]$ 与 $b[]$ 表示

解决方案甲：利用顺序存储结构直接表示

- 利用数组元素与下标分别表示多项式中对应各项的系数与指数
- 即：令数组元素 $a[k]$ 表示 x^k 前的系数 a_k
- 简单四则运算只需在数组对应元素之间运算即可

例

- 多项式 $f(x) = 1 - 3x^2 + 4x^6$ 与 $g(x) = x + 5x^2 - 7x^4$ 可分别由数组 $a[]$ 与 $b[]$ 表示

	0	1	2	3	4	5	6	...
a	1	0	-3	0	0	0	4	...
b	0	1	5	0	-7	0	0	...

图 1: 解决方案甲示例演示

解决方案甲：利用顺序存储结构直接表示

- 利用数组元素与下标分别表示多项式中对应各项的系数与指数
- 即：令数组元素 $a[k]$ 表示 x^k 前的系数 a_k
- 简单四则运算只需在数组对应元素之间运算即可

例

- 多项式 $f(x) = 1 - 3x^2 + 4x^6$ 与 $g(x) = x + 5x^2 - 7x^4$ 可分别由数组 $a[]$ 与 $b[]$ 表示

	0	1	2	3	4	5	6	...
a	1	0	-3	0	0	0	4	...
b	0	1	5	0	-7	0	0	...

图 1: 解决方案甲示例演示

- 思考：如何处理系数过于稀疏的情况？如： $f(x) = 1 - 3x^{100} + 2x^{1,000,000}$

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$							

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)						

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)					

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)				

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)	(-13, 6)			

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)	(-13, 6)	(3, 2)		

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)	(-13, 6)	(3, 2)	(82, 0)	

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)	(-13, 6)	(3, 2)	(82, 0)	...

图 2: 解决方案乙示例演示过程

解决方案乙：利用顺序存储结构只表示非零项

- 利用数组元素表示由各非零项的系数与指数组成的二元组 (a_k, k)
- 数组元素按非零项指数大小顺序排列
- 在运算时只需逐个比较每项的指数即可

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 相加可表示为

	0	1	2	3	4	5	...
$f(x)$	(9, 12)	(15, 8)	(3, 2)	...			
$g(x)$	(26, 19)	(-4, 8)	(-13, 6)	(82, 0)	...		
$f + g$	(26, 19)	(9, 12)	(11, 8)	(-13, 6)	(3, 2)	(82, 0)	...

图 2: 解决方案乙示例演示过程

- 于是, $f(x) + g(x) = 26x^{19} + 9x^{12} + 11x^8 - 13x^6 + 3x^2 + 82$

解决方案丙：利用链式存储结构只表示非零项

coefficient	exponent	next
-------------	----------	------

图 3: 单链表结点结构

- 利用链表结点表示各非零项的系数、指数与下一个结点的地址

解决方案丙：利用链式存储结构只表示非零项

coefficient	exponent	next
-------------	----------	------

图 3: 单链表结点结构

- 利用链表结点表示各非零项的系数、指数与下一个结点的地址

例

- 多项式 $f(x) = 9x^{12} + 15x^8 + 3x^2$ 与 $g(x) = 26x^{19} - 4x^8 - 13x^6 + 82$ 可分别表示为



图 4: 解决方案丙示例结构

一点启示

- 同一问题有不同的表示方案
- 共性：线性结构的组织与管理

线性结构

线性结构 (Linear Structure)

- 又名线性表、序列 (**sequence**), 指具有**相同**数据类型的 n 个¹数据元素的**有限**序列
- 其长度 (**length**) 指序列中数据元素的个数
- 非空序列指至少含有一个元素的序列, 可记作

$$s = (a_1, a_2, \dots, a_k, \dots, a_n),$$

其中

- a_k 表示数据元素²
- 下标 $k \in \mathbb{Z} \cap [1, n]$ 表示该元素在序列中的位置序号 (**index**) 或秩 (**rank**)
- 特别地, 空序列可记作

$$s = ()$$

¹ $n \in \mathbb{Z} \cap [0, +\infty)$

²可表示**任意**数据类型, 如无特别说明则采用简单数据类型

例：幼儿园小朋友排队放学

- 每个班均只有有限多个小朋友
- 在每个班队伍中一般不允许有不属于该班小朋友的存在
- 为点名方便，所有小朋友均须按顺序排队
 - 排头小朋友前面与排尾小朋友后面均没有其他小朋友
 - 其余每个小朋友的前后均有且只有一个其他小朋友



序列的特点

- 有穷性：序列中数据元素的个数是**有限**的
- 一致性：序列中所有数据元素**类型**均须**相同**
- 顺序性：序列中所有元素均按**顺序**排列
 - 首元素无前驱 (**predecessor**)，尾元素无后继 (**successor**)
 - 其余元素均分别有且只有一个**直接 (immediate)** 前驱与 **直接后继**



序列的抽象数据类型

ADT Sequence {

数据:

数据对象: $\mathcal{D} = \{a_k | a_k \in \text{数据元素集合}, k \in \mathbb{Z} \cap [1, n]\}$

逻辑关系: $\mathcal{R} = \{\langle a_{k-1}, a_k \rangle | k \in \mathbb{Z} \cap [2, n]\}$

操作:

create_sequence()

构造并初始化一个空序列s

get_sequence_length(s)

获取并返回序列s 中所含元素个数

get_sequence_element(s, k)

获取并返回序列s 中的第k 个元素

search_sequence_element(s, x)

查找序列s 中值为x 的元素, 返回其首次出现的序号或地址

insert_sequence_element(s, k, x)

在序列s 的第k 个位置插入值为x 的新元素, 后续元素序号与总元素个数均加 1

remove_sequence_element(s, k)

删除序列s 的第k 个元素, 后续元素序号与总元素个数均减 1

}

一些说明

- 线性结构的基本操作由实际应用而定
- 复杂操作可通过基本操作的组合而实现
- 针对不同应用，其操作接口可能略有差异

序列的存储结构

- 顺序存储结构：顺序列表
 - 元素按地址相邻存储在内存的一片连续地址空间中
 - 长度固定，可由内置数组的简单封装实现
 - 若欲实现变长结构，须采用进一步策略封装成动态数组或向量
- 链式存储结构：链式列表
 - 元素在内存中一般彼此不相邻，仅按前驱后继关系通过指针相连
 - 长度可变，可由自定义结构体实现，较灵活

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用顺序表形式存储的过程



图 5: 顺序表存储过程演示

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用顺序表形式存储的过程



图 5: 顺序表存储过程演示

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用顺序表形式存储的过程



图 5: 顺序表存储过程演示

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用顺序表形式存储的过程

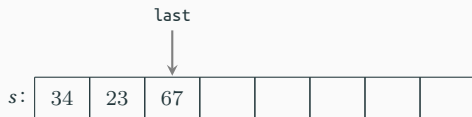


图 5: 顺序表存储过程演示

顺序列表 (Sequence List)

- 又名顺序表，用一段地址连续的存储单元，依次存储序列中的数据元素

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用顺序表形式存储的过程

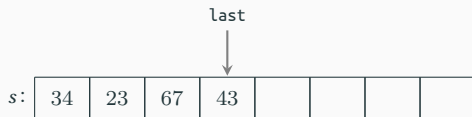


图 5: 顺序表存储过程演示

思考：用何种属性描述顺序表？

s:

34	23	67	43					4
----	----	----	----	--	--	--	--	---

图 6: 顺序表的属性描述

思考：用何种属性描述顺序表？

- 存储空间的起始位置



图 6: 顺序表的属性描述

思考：用何种属性描述顺序表？

- 存储空间的起始位置
- 容量 (capacity)：最多可容纳的元素个数



图 6: 顺序表的属性描述

思考：用何种属性描述顺序表？

- 存储空间的起始位置
- 容量 (capacity)：最多可容纳的元素个数
- 长度 (length)：实际容纳的元素个数



图 6: 顺序表的属性描述

思考：如何为顺序表分配内存？

s:

34	23	67	43					4
----	----	----	----	--	--	--	--	---

图 7: 顺序表的内存分配

思考：如何为顺序表分配内存？

- 一维数组的静态分配

s:

34	23	67	43					4
----	----	----	----	--	--	--	--	---

图 7: 顺序表的内存分配

思考：如何获取任意元素的存储地址？



图 8: 顺序表的随机访问原理

思考：如何获取任意元素的存储地址？

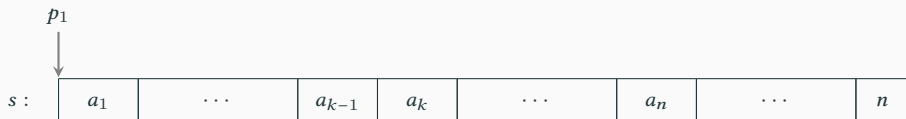


图 8: 顺序表的随机访问原理

思考：如何获取任意元素的存储地址？

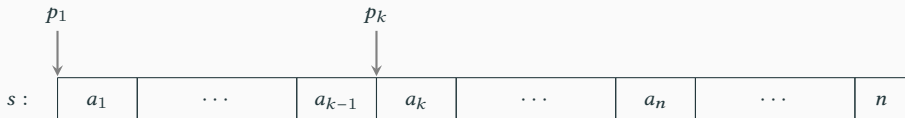


图 8: 顺序表的随机访问原理

思考：如何获取任意元素的存储地址？

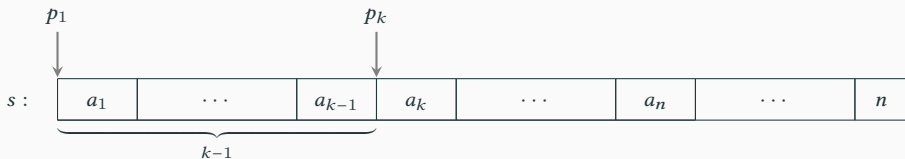


图 8: 顺序表的随机访问原理

思考：如何获取任意元素的存储地址？

- 令 p_k 为元素 a_k 的起始地址，且每个元素 a_k 所占空间为 m ，则有

$$p_k = p_1 + (k - 1)m, \quad k \in \mathbb{Z} \cap [1, n]$$

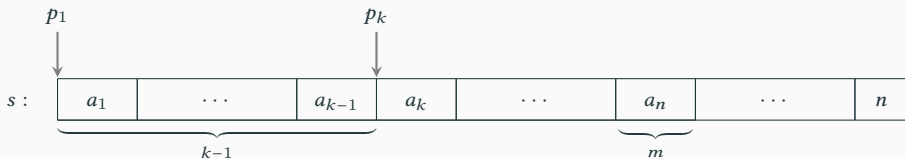


图 8: 顺序表的随机访问原理

思考：如何获取任意元素的存储地址？

- 令 p_k 为元素 a_k 的起始地址，且每个元素 a_k 所占空间为 m ，则有

$$p_k = p_1 + (k - 1)m, \quad k \in \mathbb{Z} \cap [1, n]$$

- 顺序列表中的元素可被**随机访问 (Random Access)**，时间复杂度为 $O(1)$

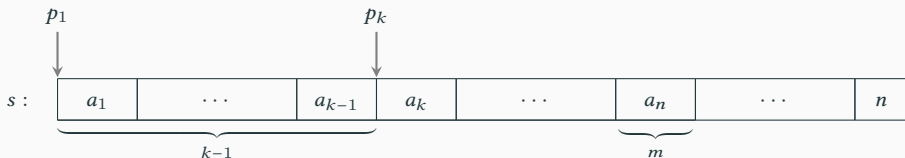


图 8: 顺序表的随机访问原理

顺序表的类型说明

- 以整型常量表示容量
- 以静态数组存储表中各元素
- 以尾元素的序号加 1 表示长度
- 特别地，空表尾元素序号为 -1

```
1  #define CAPACITY 256
2
3  typedef int DataType; // 元素类型
4
5  typedef struct {
6      DataType data[CAPACITY]; // 元素
7      int last; // 尾元素序号
8  } SequenceList;
```

顺序表的初始化

- 为空表动态分配空间
- 将尾元素序号置为 -1
- 返回指向空表的指针

```
1  SequenceList *create_sequence_list() {  
2      SequenceList *s = malloc(sizeof(SequenceList));  
3      if (s) {  
4          s->last = -1;  
5      }  
6      return s;  
7  }
```

向顺序表中插入元素

- 检测并处理非法输入³
- 依次向后移动目标位置后元素
- 在目标位置处插入新元素
- 更新尾元素序号

```
1 bool insert_sequence_list(  
2     SequenceList *s, int k, DataType d) {  
3     if (full_sequence_list(s)  
4         || wrong_insert_index(s, k)) {  
5         return false; // 检测并处理各种非法插入情况  
6     }  
7     for (int i = s->last; i >= k; --i) {  
8         s->data[i + 1] = s->data[i]; // 注意移动顺序  
9     }  
10    s->data[k] = d; // 插入新元素  
11    ++s->last; // 更新尾元素序号  
12    return true;  
13 }
```

³为表示布尔型变量，须包含 <stdbool.h>头，下同

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

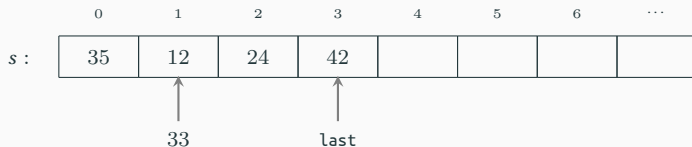


图 9: 在顺序表中插入元素过程演示

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

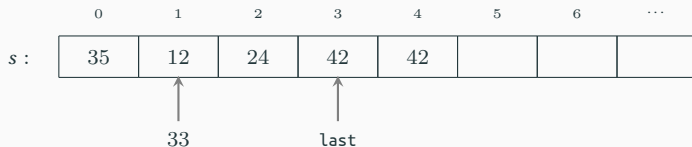


图 9: 在顺序表中插入元素过程演示

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

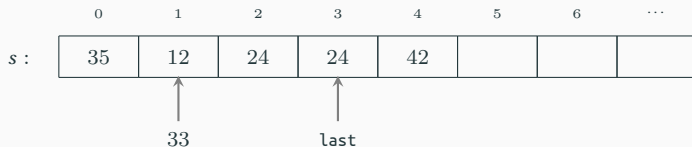


图 9: 在顺序表中插入元素过程演示

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

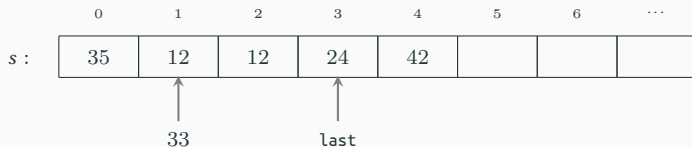


图 9: 在顺序表中插入元素过程演示

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

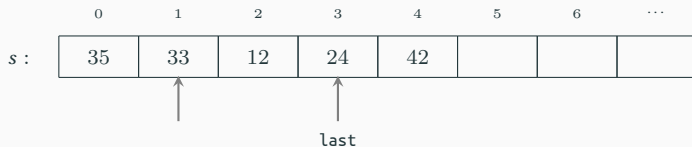


图 9: 在顺序表中插入元素过程演示

例

- 在顺序表形式的序列 $s = (35, 12, 24, 42)$ 中第 1 个位置处插入 33

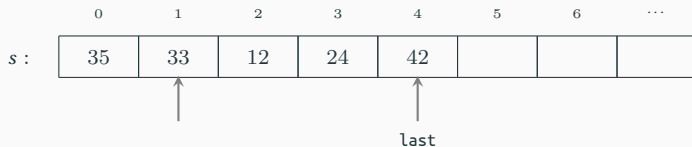


图 9: 在顺序表中插入元素过程演示

顺序表插入算法的复杂度分析

顺序表插入算法的复杂度分析

- 插入运算主要耗时于依次移动数据
 - 在序号为 k 的位置插入时须移动 $n - k$ 次, $k \in \mathbb{Z} \cap [0, n]$

顺序表插入算法的复杂度分析

- 插入运算主要耗时于依次移动数据
 - 在序号为 k 的位置插入时须移动 $n - k$ 次, $k \in \mathbb{Z} \cap [0, n]$
- 若在序号为 k 的位置插入的概率为 p_k , 则移动次数的期望为

$$\mathbb{E}_{\text{insert}} = \sum_{k=0}^n (n - k)p_k$$

顺序表插入算法的复杂度分析

- 插入运算主要耗时于依次移动数据
 - 在序号为 k 的位置插入时须移动 $n - k$ 次, $k \in \mathbb{Z} \cap [0, n]$
- 若在序号为 k 的位置插入的概率为 p_k , 则移动次数的期望为

$$\mathbb{E}_{\text{insert}} = \sum_{k=0}^n (n - k)p_k$$

- 当在所有合法位置插入的概率均等时, 即对所有合法的 k 均有 $p_k = (n + 1)^{-1}$, 则有

$$\mathbb{E}_{\text{insert}} = \frac{1}{n + 1} \sum_{k=0}^n (n - k) = \frac{n}{2}$$

顺序表插入算法的复杂度分析

- 插入运算主要耗时于依次移动数据
 - 在序号为 k 的位置插入时须移动 $n - k$ 次, $k \in \mathbb{Z} \cap [0, n]$
- 若在序号为 k 的位置插入的概率为 p_k , 则移动次数的期望为

$$\mathbb{E}_{\text{insert}} = \sum_{k=0}^n (n - k)p_k$$

- 当在所有合法位置插入的概率均等时, 即对所有合法的 k 均有 $p_k = (n + 1)^{-1}$, 则有

$$\mathbb{E}_{\text{insert}} = \frac{1}{n + 1} \sum_{k=0}^n (n - k) = \frac{n}{2}$$

- 综上, 顺序表插入操作的时间复杂度为 $O(n)$

从顺序表中删除元素

- 检测并处理非法输入
- 依次向前移动目标位置后元素
- 覆盖目标位置处的元素
- 更新尾元素序号

```
1 bool remove_sequence_list(  
2     SequenceList *s, int k, DataType *d) {  
3     if (wrong_remove_index(s, k)) {  
4         return false; // 检测并处理各种非法插入情况  
5     }  
6     if (d) {  
7         *d = s->data[k]; // 记录并返回被删的值  
8     }  
9     for (int i = k; i < s->last; ++i) {  
10        s->data[i] = s->data[i + 1]; // 注意移动顺序  
11    }  
12    --s->last; // 更新尾元素序号  
13    return true;  
14 }
```

例

- 删除顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 中第 1 个位置元素 33

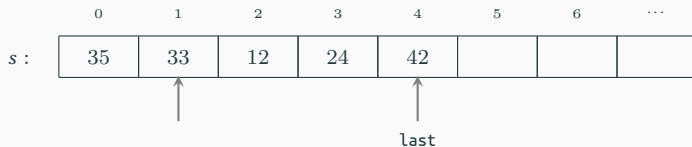


图 10: 从顺序表中删除元素过程演示

例

- 删除顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 中第 1 个位置元素 33

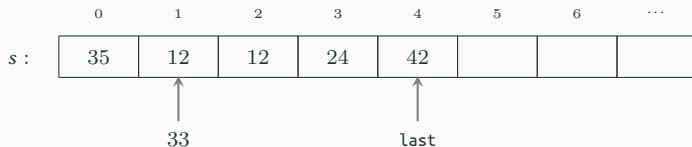


图 10: 从顺序表中删除元素过程演示

例

- 删除顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 中第 1 个位置元素 33

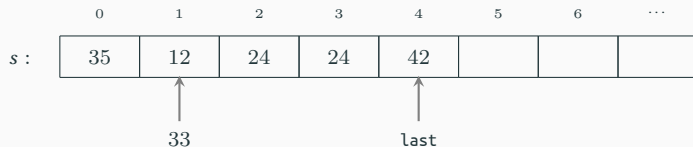


图 10: 从顺序表中删除元素过程演示

例

- 删除顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 中第 1 个位置元素 33

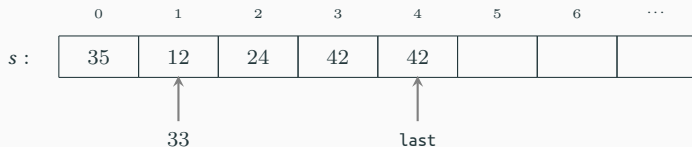


图 10: 从顺序表中删除元素过程演示

例

- 删除顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 中第 1 个位置元素 33

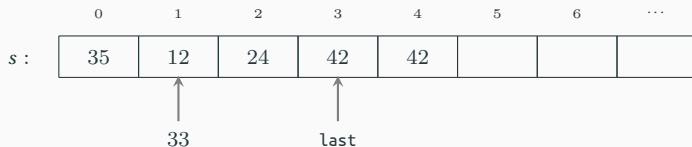


图 10: 从顺序表中删除元素过程演示

顺序表删除算法的复杂度分析

顺序表删除算法的复杂度分析

- 删除与插入运算互为逆运算，故仍主要耗时于依次移动数据

顺序表删除算法的复杂度分析

- 删除与插入运算互为逆运算，故仍主要耗时于依次移动数据
- 同理可得顺序表删除操作的时间复杂度亦为 $O(n)$

在顺序表中按值查找元素

- 从首至尾遍历表中所有元素
- 若发现匹配元素则返回其序号
- 否则返回 -1 表示查找失败

```
1 int search_sequence_list(SequenceList *s, DataType d) {  
2     int k = 0;  
3     for (; k <= s->last && s->data[k] != d; ++k)  
4         ; // 空语句，不执行任何操作  
5     return s->last < k ? -1 : k;  
6 }
```


例

- 在顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 查找值为 12 的元素的位置

	0	1	2	3	4	5	6	...
s :	35	33	12	24	42			

图 11: 在顺序表中查找元素过程演示

例

- 在顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 查找值为 12 的元素的位置

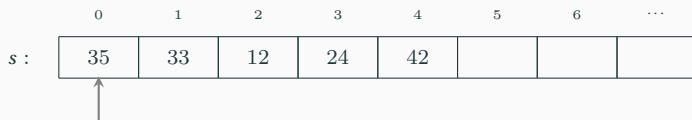


图 11: 在顺序表中查找元素过程演示

例

- 在顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 查找值为 12 的元素的位置

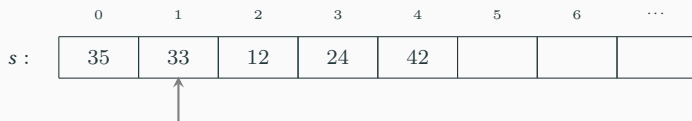


图 11: 在顺序表中查找元素过程演示

例

- 在顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 查找值为 12 的元素的位置

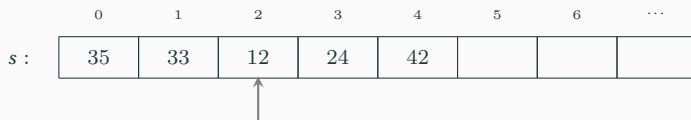


图 11: 在顺序表中查找元素过程演示

例

- 在顺序表形式的序列 $s = (35, 33, 12, 24, 42)$ 查找值为 12 的元素的位置

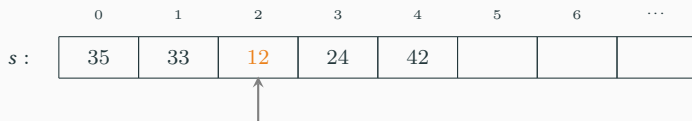


图 11: 在顺序表中查找元素过程演示

课堂思考练习

课堂思考练习

- 分析顺序表删除算法的时间复杂度

课堂思考练习

- 分析顺序表删除算法的时间复杂度
- 实现逆序遍历搜索并讨论其与顺序遍历的异同

顺序表的优势

- 节省空间：无需为表示元素间逻辑关系而增加额外存储空间
- 随机访问：可快速访问表中任意位置元素， $T(n) = O(1)$

顺序表的优势

- 节省空间：无需为表示元素间逻辑关系而增加额外存储空间
- 随机访问：可快速访问表中任意位置元素， $T(n) = O(1)$

顺序表的不足

- 容量固定：表容量事先难以确定且难以扩充
- 增减困难：插入删除元素需移动大量元素， $T(n) = O(n)$

课堂编程练习

- 输入：顺序表 s_a 与 s_b ，其元素均已按升序排列
- 输出：顺序表 s_c ，其元素为 s_a 与 s_b 中元素的合并，并以降序排列

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程

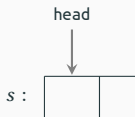


图 12: 单链表的存储过程演示

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程

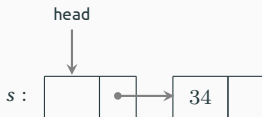


图 12: 单链表的存储过程演示

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程

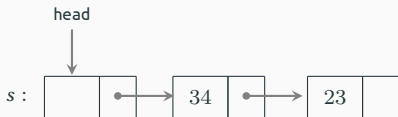


图 12: 单链表的存储过程演示

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程



图 12: 单链表的存储过程演示

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程



图 12: 单链表的存储过程演示

⁴简称单链表

链式列表 (Linked List)

- 简称链表，用一组在内存中零散分布的存储单元存储序列中的数据元素
- 按链接关系分为单向链表⁴、双向链表与循环链表等

例

- 考察序列 $s = (34, 23, 67, 43)$ 采用单链表形式存储的过程



图 12: 单链表的存储过程演示

⁴简称单链表

思考

- 用何种属性描述单链表？
- 如何为单链表分配内存？
- 如何获取任意元素的存储地址？

单链表的类型说明

- 以**结点**为基本单位
- 为结点**动态**分配内存
- 结点中包含下一个结点**地址**
- 单链表只需记录**首结点**即可

```
1 typedef int DataType; // 元素类型
2
3 typedef struct ListNode {
4     DataType data; // 数据元素
5     struct ListNode *next; // 下一个结点
6 } ListNode;
7
8 typedef struct {
9     ListNode *head; // 首结点
10 } LinkedList;
```

单链表结点的初始化

- 为结点动态分配内存
- 更新结点中各种数据

```
1  ListNode *create_linked_node(DataType d) {  
2      ListNode *n = malloc(sizeof(ListNode));  
3      if (n) {  
4          n->data = d;  
5          n->next = NULL;  
6      }  
7      return n;  
8  }
```

单链表的初始化

- 为单链表动态分配内存
- 以默认值⁵初始化首结点

```
1  LinkedList *create_linked_list() {  
2      LinkedList *s = malloc(sizeof(LinkedList));  
3      if (s) {  
4          s->head = create_linked_node(0);  
5      }  
6      return s;  
7  }
```

⁵与DataType有关，此处暂时为 0

求单链表的长度

- 从首结点开始遍历至尾结点
- 记录已遍历的结点数
- 计数时不包括首结点

```
1  int get_linked_list_length(LinkedList *s) {  
2      int length = -1; // 计数不包括首结点  
3      for (LinkedListNode *p = s->head; p != NULL;  
4          p = p->next, ++length)  
5          ; // 空语句  
6      return length;  
7  }
```

求单链表的长度

- 从首结点开始遍历至尾结点
- 记录已遍历的结点数
- 计数时不包括首结点

说明

- $T(n) = O(n)$
- 注意与顺序表比较

```
1  int get_linked_list_length(LinkedList *s) {  
2      int length = -1; // 计数不包括首结点  
3      for (LinkedListNode *p = s->head; p != NULL;  
4          p = p->next, ++length)  
5          ; // 空语句  
6      return length;  
7  }
```


在单链表中按序号查找指定元素

- 从首结点遍历指定次数即可
- 注意边界条件

```
1  ListNode *search_linked_by_index(  
2      LinkedList *s, int k) {  
3      ListNode *p = s->head;  
4      for (int i = 0; p != NULL && i < k;  
5          ++i, p = p->next)  
6          ; // 空语句  
7      return p;  
8  }
```

在单链表中按序号查找指定元素

- 从首结点遍历指定次数即可
- 注意边界条件

在单链表中按值查找指定元素

- 从首结点遍历至找到匹配值的结点
- 遍历范围不应包括首结点

```
1  ListNode *search_linked_by_index(  
2      LinkedList *s, int k) {  
3      ListNode *p = s->head;  
4      for (int i = 0; p != NULL && i < k;  
5          ++i, p = p->next)  
6          ; // 空语句  
7      return p;  
8  }
```

```
1  ListNode *search_linked_by_data(  
2      LinkedList *s, DataType d) {  
3      ListNode *p = s->head->next;  
4      for (; p != NULL && p->data != d; p = p->next)  
5          ; // 空语句  
6      return p;  
7  }
```

在单链表中按序号查找指定元素

- 从首结点遍历指定次数即可
- 注意边界条件

在单链表中按值查找指定元素

- 从首结点遍历至找到匹配值的结点
- 遍历范围不应包括首结点

说明

- 二者均有: $T(n) = O(n)$
- 注意与顺序表比较

```
1  ListNode *search_linked_by_index(  
2      LinkedList *s, int k) {  
3      ListNode *p = s->head;  
4      for (int i = 0; p != NULL && i < k;  
5          ++i, p = p->next)  
6          ; // 空语句  
7      return p;  
8  }
```

```
1  ListNode *search_linked_by_data(  
2      LinkedList *s, DataType d) {  
3      ListNode *p = s->head->next;  
4      for (; p != NULL && p->data != d; p = p->next)  
5          ; // 空语句  
6      return p;  
7  }
```

在单链表中指定序号的结点后插入元素

- 按序号查找出目标结点
- 处理可能的非法查询结果
- 按指定值创建新结点
- 在目标结点后插入新结点

```
1  ListNode *attach_after_node(  
2      ListNode *p, ListNode *n) {  
3      assert(p && n);  
4      n->next = p->next;  
5      p->next = n;  
6      return n;  
7  }
```

```
1  bool insert_after_linked_by_index(  
2      LinkedList *s, int k, DataType d) {  
3      ListNode *p = search_linked_by_index(s, k);  
4      if (!p) {  
5          printf("Wrong insert index!\n");  
6          return false;  
7      }  
8      ListNode *n = create_linked_node(d);  
9      return n && attach_after_node(p, n);  
10 }
```

在单链表中指定序号的结点后插入元素

- 按序号查找出目标结点
- 处理可能的非法查询结果
- 按指定值创建新结点
- 在目标结点后插入新结点

说明

- 注意高亮代码的执行顺序
- 插入本身: $T(n) = O(1)$
- 连同查找: $T(n) = O(n)$
- 注意与顺序表比较

```
1  ListNode *attach_after_node(  
2      ListNode *p, ListNode *n) {  
3      assert(p && n);  
4      n->next = p->next;  
5      p->next = n;  
6      return n;  
7  }
```

```
1  bool insert_after_linked_by_index(  
2      LinkedList *s, int k, DataType d) {  
3      ListNode *p = search_linked_by_index(s, k);  
4      if (!p) {  
5          printf("Wrong insert index!\n");  
6          return false;  
7      }  
8      ListNode *n = create_linked_node(d);  
9      return n && attach_after_node(p, n);  
10 }
```

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

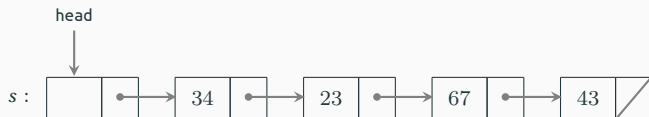


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

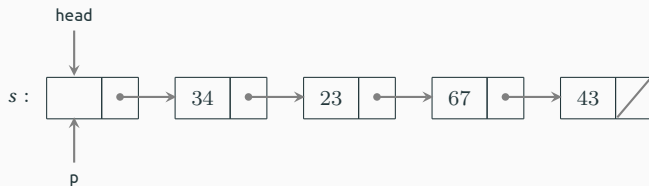


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

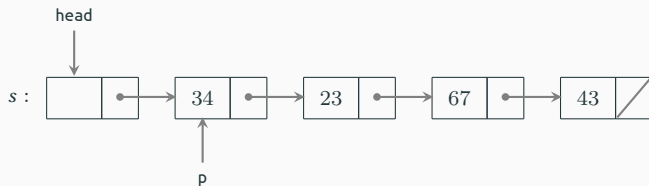


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

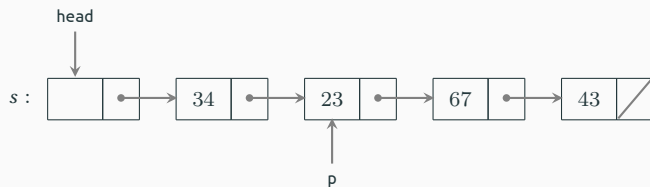


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

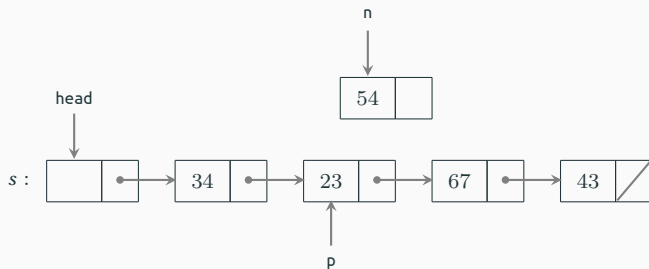


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

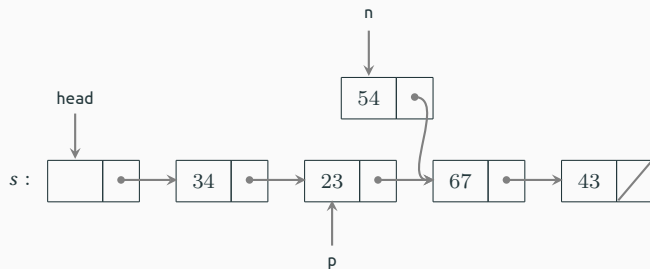


图 13: 在单链表结点后插入新结点的过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中第 2 个结点 23 后插入 54

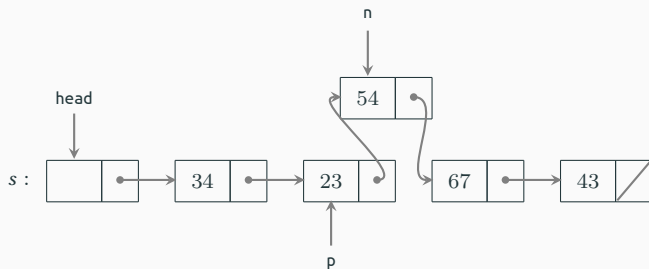


图 13: 在单链表结点后插入新结点的过程演示

思考

- 如何在单链表中指定序号的结点前插入元素？

在单链表中删除指定序号的结点

- 按序号查找出目标结点的直接前驱
- 处理可能的非法查询结果
- 更新结点的顺序关系
- 删除目标结点

```
1  ListNode *detach_after_node(ListNode *q) {  
2      assert(q && q->next);  
3      ListNode *p = q->next;  
4      q->next = p->next;  
5      return p;  
6  }
```

```
1  bool remove_linked_by_index(  
2      LinkedList *s, int k, DataType *d) {  
3      ListNode *q = search_linked_by_index(s, k - 1);  
4      if (!q || !q->next) {  
5          printf("Wrong remove index!\n");  
6          return false;  
7      }  
8      ListNode *p = detach_after_node(q);  
9      if (d) { *d = p->data; }  
10     free(p);  
11     return true;  
12 }
```

在单链表中删除指定序号的结点

- 按序号查找出目标结点的直接前驱
- 处理可能的非法查询结果
- 更新结点的顺序关系
- 删除目标结点

说明

- 注意高亮代码的执行顺序
- 删除本身: $T(n) = O(1)$
- 连同查找: $T(n) = O(n)$
- 注意与顺序表比较

```
1  ListNode *detach_after_node(ListNode *q) {  
2      assert(q && q->next);  
3      ListNode *p = q->next;  
4      q->next = p->next;  
5      return p;  
6  }
```

```
1  bool remove_linked_by_index(  
2      LinkedList *s, int k, DataType *d) {  
3      ListNode *q = search_linked_by_index(s, k - 1);  
4      if (!q || !q->next) {  
5          printf("Wrong remove index!\n");  
6          return false;  
7      }  
8      ListNode *p = detach_after_node(q);  
9      if (d) { *d = p->data; }  
10     free(p);  
11     return true;  
12 }
```

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

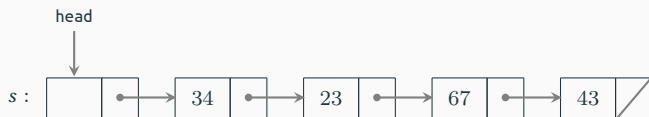


图 14: 删除单链表结点过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

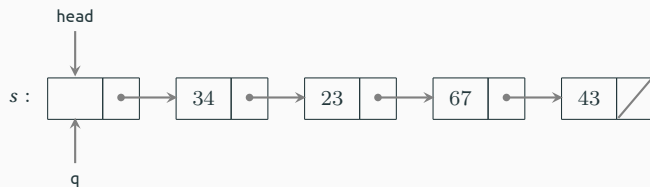


图 14: 删除单链表结点过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

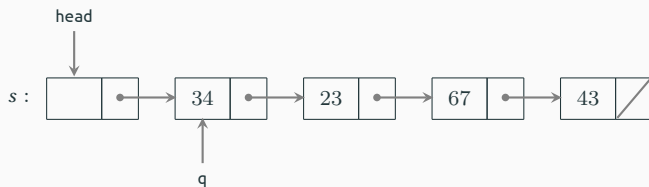


图 14: 删除单链表结点过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

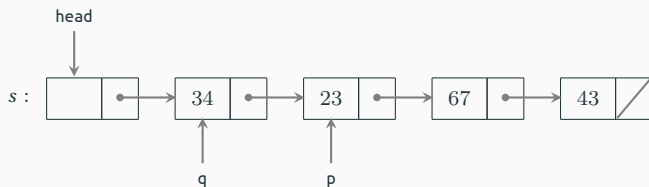


图 14: 删除单链表结点过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

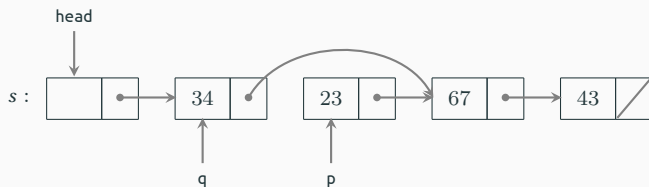


图 14: 删除单链表结点过程演示

例

- 在单链表形式的序列 $s = (34, 23, 67, 43)$ 中删除第 2 个结点 23

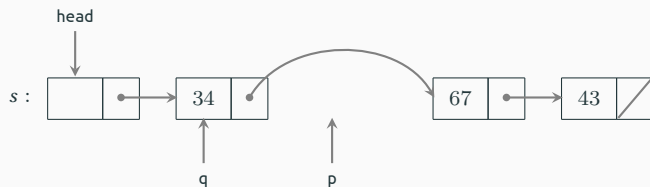


图 14: 删除单链表结点过程演示

链表的优势

- 容量可变：无需提前确定容量，可自动扩充
- 增删便捷：插入删除元素只影响局部， $T(n) = O(1)$ ⁶

⁶仅限插入删除操作本身，不包括查找

链表的优势

- 容量可变：无需提前确定容量，可自动扩充
- 增删便捷：插入删除元素只影响局部， $T(n) = O(1)$ ⁶

链表的不足

- 略占空间：每个结点均需额外存储下一结点的位置信息
- 访问不便：查找指定元素须从首结点开始遍历， $T(n) = O(n)$

⁶仅限插入删除操作本身，不包括查找

序列存储结构的选择

- 存储：当存储规模事先难以估计时，宜选用链表
- 运算：当查找操作为主时，宜选用顺序表
- 实现：当需实现简单时，宜选用顺序表

常用的线性结构

栈 (Stack)

- 一种操作受限的序列
- 仅允许在一端插入删除元素
- 可操作一端为栈顶 (top)，另一端为栈底 (bottom)
- 后进先出 (Last In First Out, LIFO)

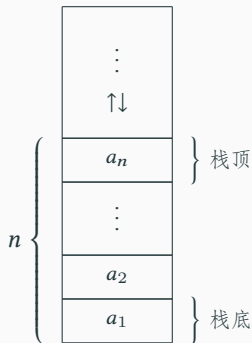


图 15: 栈的示意图

栈 (Stack)

- 一种操作受限的序列
- 仅允许在一端插入删除元素
- 可操作一端为栈顶 (top), 另一端为栈底 (bottom)
- 后进先出 (Last In First Out, LIFO)

日常生活中的栈

- 一摞碗、碟、凳子等
- 糖葫芦、牛羊肉串等
- 子弹夹
- ...

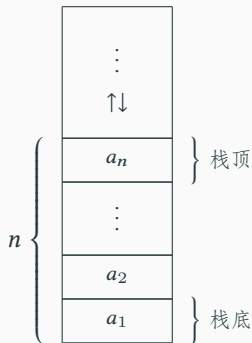


图 15: 栈的示意图

栈的抽象数据类型

ADT Stack {

数据:

数据对象: $\mathcal{D} = \{a_k | a_k \in \text{数据元素集合}, k \in \mathbb{Z} \cap [1, n]\}$

逻辑关系: $\mathcal{R} = \{\langle a_{k-1}, a_k \rangle | k \in \mathbb{Z} \cap [2, n]\}$

操作:

create_stack()

构造并初始化一个空栈s

is_empty_stack()

判断栈是否为空

push_stack(s, x)

若栈s 存在且未滿, 则在其栈顶插入值为x 的新元素

pop_stack(s)

若栈s 存在且非空, 则删除其栈顶元素

top_stack(s)

若栈s 存在且非空, 则返回其栈顶元素

}

栈的类型说明

- 可利用已有结构的再封装
- 顺序栈可采用序列
- 链式栈可采用链表

```
1 typedef struct {  
2     SequenceList *_;  
3 } SequenceStack;
```

```
1 typedef struct {  
2     LinkedList *_;  
3 } LinkedStack;
```

栈的初始化

- 可利用序列/链表已有的初始化方法
- 注意空指针的处理

```
1 SequenceStack *create_sequence_stack() {  
2     SequenceStack *s = malloc(sizeof(SequenceStack));  
3     if (s) {  
4         s->_ = create_sequence_list();  
5     }  
6     return s;  
7 }
```

```
1 LinkedStack *create_linked_stack() {  
2     LinkedStack *s = malloc(sizeof(LinkedStack));  
3     if (s) {  
4         s->_ = create_linked_list();  
5     }  
6     return s;  
7 }
```

判断栈是否为空

- 顺序栈可利用序列已有的判空方法
- 链式栈需检测首结点的下一个结点
- 此处利用了短路求值原则，下同

```
1 bool empty_sequence_stack(SequenceStack *s) {  
2     return !s || empty_sequence_list(s->_);  
3 }
```

```
1 bool empty_linked_stack(LinkedStack *s) {  
2     return !s || !s->_>head->next;  
3 }
```

入栈

- 将新元素压入栈中
- 指将新元素放入栈顶并更新栈顶
- 顺序栈栈顶序号为last
- 链式栈栈顶为首结点的直接后继

```
1 bool push_sequence_stack(  
2     SequenceStack *s, DataType d) {  
3     return s && insert_sequence_list(  
4         s->_, s->_->last + 1, d);  
5 }
```

```
1 bool push_linked_stack(  
2     LinkedStack *s, DataType d) {  
3     return s && insert_after_linked_by_index(  
4         s->_, 0, d);  
5 }
```

出栈

- 将栈顶元素从栈中弹出
- 指将栈顶元素取出并更新栈顶

```
1 bool pop_sequence_stack(  
2     SequenceStack *s, DataType *p) {  
3     return s && remove_sequence_list(  
4         s->_, s->_->last, p);  
5 }
```

```
1 bool pop_linked_stack(  
2     LinkedStack *s, DataType *p) {  
3     return s && remove_linked_by_index(  
4         s->_, 1, p);  
5 }
```

取栈顶元素

- 将栈顶元素复制一份
- 不改变栈
- 首先判断栈是否为空

```
1  bool top_sequence_stack(  
2      SequenceStack *s, DataType *p) {  
3      if (empty_sequence_stack(s) || !p) {  
4          return false;  
5      }  
6      *p = s->_->data[s->_->last];  
7      return true;  
8  }
```

```
1  bool top_linked_stack(  
2      LinkedStack *s, DataType *p) {  
3      if (empty_linked_stack(s) || !p) {  
4          return false;  
5      }  
6      *p = s->_->head->next->data;  
7      return true;  
8  }
```

栈与递归

- 递归 (**recursion**): 函数直接或间接调用自身的过程
- 递归要素: 终止条件与递归体
- 递归实现: 在栈中记录每次调用的有用信息

栈与递归

- 递归 (**recursion**): 函数直接或间接调用自身的过程
- 递归要素: 终止条件与递归体
- 递归实现: 在栈中记录每次调用的有用信息

例: 求阶乘

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & n > 0 \end{cases}$$

```
1  int factorial(int n) {  
2      return 0 == n ? 1 : n * factorial(n - 1);  
3  }
```

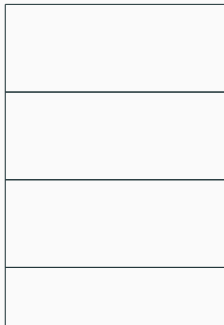


图 16: 函数调用栈:

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```



图 16: 函数调用栈: 递进调用

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }
```

```
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```

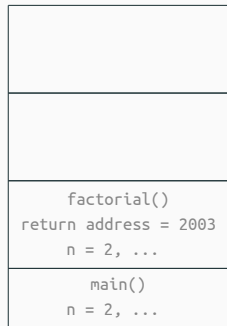


图 16: 函数调用栈: 递进调用

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }
```

```
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```

factorial() return address = 1002 n = 1, ...
factorial() return address = 2003 n = 2, ...
main() n = 2, ...

图 16: 函数调用栈: 递进调用

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```


factorial() return address = 1002 n = 0, ...
factorial() return address = 1002 n = 1, ...
factorial() return address = 2003 n = 2, ...
main() n = 2, ...

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```

图 16: 函数调用栈：递进调用

factorial() return address = 1002 n = 1, ...
factorial() return address = 2003 n = 2, ...
main() n = 2, ...

图 16: 函数调用栈: 回归求值

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```

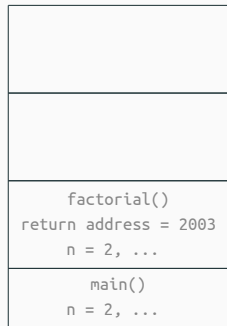


图 16: 函数调用栈: 回归求值

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```



图 16: 函数调用栈: 回归求值

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }
```

```
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```



图 16: 函数调用栈: 回归求值

```
1001 int factorial(int n) {  
1002     return 0 == n ? 1 : n * factorial(n - 1);  
1003 }  
  
2001 int main(void) {  
2002     int n = 2;  
2003     int f = factorial(n);  
2004     return 0;  
2005 }
```

递归应用：汉诺塔 (Hanoi) 问题

- 相传世界之初有钻石宝塔甲，其上有 64 金碟，由大到小依次向上摆放
- 附近另有二空塔：乙、丙，与甲类似
- 婆罗门牧师试图将甲塔金碟移至丙塔，但需借乙塔中转
- 每次只移最上一碟，且不可将大碟置于小碟之上
- 牧师完成之日便是世界末日

递归解题思路

1. 若剩余碟数 $n = 0$ ，则结束
2. 否则执行：
 - 将 $n - 1$ 碟从甲借丙移至乙
 - 将甲中剩余一碟从甲移至丙
 - 将 $n - 1$ 碟从乙借甲移至丙

```
1 void move(char a, char b) {  
2     printf("%c -> %c\n", a, b);  
3 }
```

```
1 void hanoi(int n, char a, char b, char c) {  
2     if (!n) {  
3         return; // 递归出口  
4     }  
5     hanoi(n - 1, a, c, b);  
6     move(a, c);  
7     hanoi(n - 1, b, a, c);  
8 }
```

注意

- 递归可在抽象层面帮助快速理清思路
- 递归需借助栈结构存储额外信息，故效率较迭代/循环低
- 在实现中，栈容量有限，故过多层递归极易爆栈

队列 (Queue)

- 另一种操作受限的序列
- 仅允许在一端插入且在另一端删除元素
- 可插入的一端为队尾 (**rear**)，另一端为队首 (**front**)
- 先进先出 (**First In First Out, FIFO**)

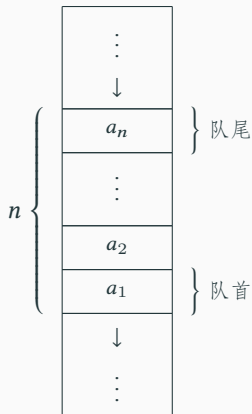


图 17: 队列的示意图

队列 (Queue)

- 另一种操作受限的序列
- 仅允许在一端插入且在另一端删除元素
- 可插入的一端为队尾 (rear)，另一端为队首 (front)
- 先进先出 (First In First Out, FIFO)

日常生活中的队列

- 排队的人（不允许插队）
- 羽毛球桶
- ...

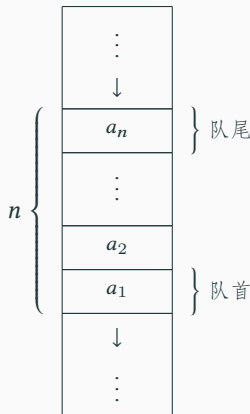


图 17: 队列的示意图

队列的抽象数据类型

ADT Queue {

数据:

数据对象: $\mathcal{D} = \{a_k | a_k \in \text{数据元素集合}, k \in \mathbb{Z} \cap [1, n]\}$

逻辑关系: $\mathcal{R} = \{\langle a_{k-1}, a_k \rangle | k \in \mathbb{Z} \cap [2, n]\}$

操作:

`create_queue()`

构造并初始化一个空队列 s

`is_empty_queue()`

判断队列是否为空

`in_queue(s, x)`

若队列 s 存在, 则在其队尾插入值为 x 的新元素

`out_queue(s)`

若队列 s 存在且非空, 则记录并删除其队首元素

}

顺序队列（版本甲）

- 可采用已有序列结构的封装
- 序列第一个元素为队首
- 序列最后一个元素为队尾
- 入队时可直接在末尾插入新元素
- 出队时可删除第一个元素

```
1 typedef struct {  
2     SequenceList *_;  
3 } SequenceQueue;
```

```
1 bool push_sequence_queue(  
2     SequenceQueue *s, DataType d) {  
3     return s && insert_sequence_list(  
4         s->_, s->_->last + 1, d);  
5 }
```

```
1 bool pop_sequence_queue(  
2     SequenceQueue *s, DataType *d) {  
3     return s && remove_sequence_list(s->_, 0, d);  
4 }
```

时间复杂度

- 入队操作：无需移动任何元素， $O(1)$
- 出队操作：需移动所有元素， $O(n)$ ，如何改进？

时间复杂度

- 入队操作：无需移动任何元素， $O(1)$
- 出队操作：需移动所有元素， $O(n)$ ，如何改进？

思路

- 与其移动元素，不如移动队首，正如队尾一样

顺序队列（版本乙）

- 总体同版本甲
- 新增队首序号成员
- 出队时仅需增加队首序号

```
1  typedef struct {  
2      SequenceList *_;  
3      int front;  
4  } SequenceQueue;  
  
1  bool pop_sequence_queue(  
2      SequenceQueue *s, DataType *d) {  
3      if (!s) {  
4          return false;  
5      }  
6      if (d) {  
7          *d = s->_->data[front];  
8      }  
9      return ++front;  
10 }
```

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

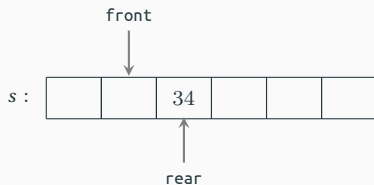


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

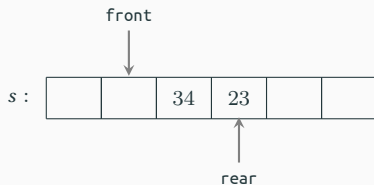


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

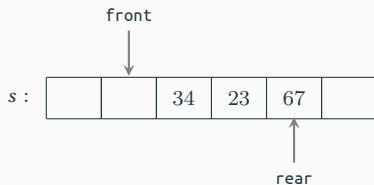


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

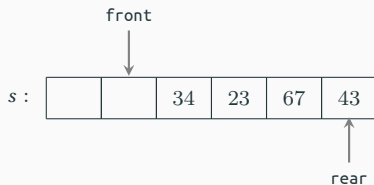


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

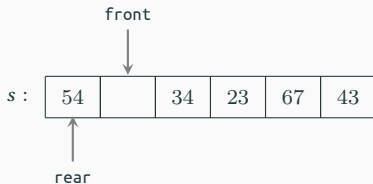


图 18: 队列的假上溢现象及其解决思路

思路

- 采用循环队列，假上溢时可将队尾移至内置数组首元素，以充分利用空间

新问题：如何判断队列已空或已满？

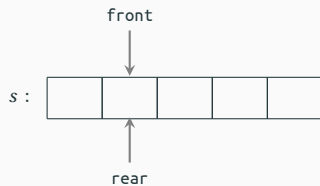


图 19: 队列已空

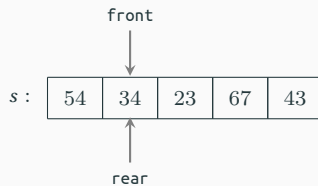


图 20: 队列已满

思路甲：设立标志位`flag`辅助判断

- 除检查队首与队尾是否重合外，尚需判断标志位（可约定：0 为队空，1 为队满）

思路甲：设立标志位`flag`辅助判断

- 除检查队首与队尾是否重合外，尚需判断标志位（可约定：0 为队空，1 为队满）

思路乙：在队首处空出一个元素

- 队空时队首队尾重合，队满时二者不重合

思路甲：设立标志位`flag`辅助判断

- 除检查队首与队尾是否重合外，尚需判断标志位（可约定：0 为队空，1 为队满）

思路乙：在队首处空出一个元素

- 队空时队首队尾重合，队满时二者不重合

思路丙：新增`size`成员记录队列长度

- 队空时长度为 0，队满时长度为容量

顺序队列（版本丙）

- 总体同版本乙
- 新增队列长度成员
- 因出入较大，故已不借用序列结构

```
1 typedef struct {  
2     DataType data[CAPACITY];  
3     int front; // 队首  
4     int rear; // 队尾  
5     int size; // 队列长度  
6 } SequenceQueue;
```

顺序队列的初始化

- 为队列分配空间
- 将队首与队尾均置于末尾
- 将队列长度置 0

```
1 SequenceQueue *create_sequence_queue() {  
2     SequenceQueue *s = malloc(sizeof(SequenceQueue));  
3     if (s) {  
4         s->front = s->rear = CAPACITY - 1;  
5         s->size = 0;  
6     }  
7     return s;  
8 }
```

入队

- 确保队列未满
- 队尾后移
 - 若已至绝境则从头再来
- 加入新元素
- 更新队列长度

```
1  bool push_sequence_queue(  
2      SequenceQueue *s, DataType d) {  
3      if (full_sequence_queue(s)) {  
4          return false;  
5      }  
6      s->rear = (s->rear + 1) % CAPACITY;  
7      s->data[s->rear] = d;  
8      return ++s->size;  
9  }
```

出队

- 确保队列非空
- 队首后移⁷
 - 若已至绝境则从头再来
- 记录已删除元素
- 更新队列长度

```
1  bool pop_sequence_queue(  
2      SequenceQueue *s, DataType *d) {  
3      if (empty_sequence_queue(s)) {  
4          return false;  
5      }  
6      s->front = (s->front + 1) % CAPACITY;  
7      if (d) {  
8          *d = s->data[s->front];  
9      }  
10     --s->size;  
11     return true;  
12 }
```

⁷注：队首一般指向首元素前一个位置

链式队列

- 可采用已有链表结构的封装
- 新增队首与队尾指针

```
1 typedef struct {  
2     LinkedList *_;  
3     ListNode *front; // 队首  
4     ListNode *rear;  // 队尾  
5 } LinkedQueue;
```

链式队列

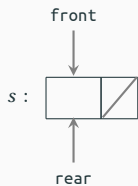


图 21: 空链式队列

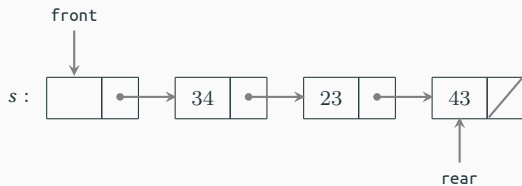


图 22: 非空链式队列

链式队列的初始化

- 调用链表初始化方法
- 将队首与队尾均指向首结点

```
1  LinkedQueue *create_linked_queue() {  
2      LinkedQueue *s = malloc(sizeof(LinkedQueue));  
3      if (s) {  
4          s->_ = create_linked_list();  
5          s->front = s->rear = s->_->head;  
6      }  
7      return s;  
8  }
```

链式队列的判空

- 队首与队尾是否重合

```
1 bool empty_linked_queue(LinkedQueue *s) {  
2     return !s || s->front == s->rear;  
3 }
```


入队

- 创建新结点
- 在链表末端插入新结点⁸
- 队尾指针后移

```
1  bool push_linked_queue(  
2      LinkedQueue *s, DataType d) {  
3      if (!s) {  
4          return false;  
5      }  
6      ListNode *n = create_linked_node(d);  
7      return n && (s->rear =  
8          attach_after_node(s->rear, n));  
9  }
```

⁸注：此时已有队尾指针记录队尾结点，故无需每次查找之

出队

- 记录链表第一结点的值
- 删除该首结点
- 当队列空时队尾指针前移

```
1  bool pop_linked_queue(  
2      LinkedQueue *s, DataType *d) {  
3      if (empty_linked_queue(s)) {  
4          return false;  
5      }  
6      ListNode *p = detach_after_node(s->front);  
7      if (d) {  
8          *d = p->data;  
9      }  
10     free(p);  
11     if (empty_linked_queue(s)) {  
12         s->rear = s->front;  
13     }  
14     return true;  
15 }
```

表 1: 一点建议

	栈	队列
应用	逆序记录	顺序缓冲
实现	动态数组	双向链表

小结

- 线性结构为其他数据结构的基础
- 线性结构特点包括**有穷性**、**一致性**与**顺序性**
- 从存储结构上包括连续的**顺序表/序列**与分散的**链表**两大类
 - 序列：容量固定⁹，增删费时，随机访问
 - 链表：容量可变，增删灵活，查找费时
- 线性结构衍生出**栈**与**队列**两类常用实例
 - 栈：后进先出，一般用于逆序记录
 - 队列：先进先出，一般用于正序缓冲

⁹动态数组/向量可弥补

问与答