数据结构与算法

Data Structures and Algorithms

谢昊

xiehao@cuz.edu.cn

线性结构
Linear Structures

大纲

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

引例：一元多项式的表示与计算

- 考察一元 n 次多项式

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$$

- 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?
  - $f(x) \pm g(x)$
  - $f(x) \cdot g(x)$

引例

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

引例

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	$\cdots$
a	1	0	−3	0	0	0	4	$\cdots$
b	0	1	5	0	−7	0	0	$\cdots$

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	$\cdots$
a	1	0	-3	0	0	0	4	$\cdots$
b	0	1	5	0	-7	0	0	$\cdots$

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：$f(x) = 1 - 3x^{100} + 2x^{1,000,000}$

引例

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

引例

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

图 2: 解决方案乙示例演示过程

引例

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

图 2: 解决方案乙示例演示过程

引例

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

图 2: 解决方案乙示例演示过程

引例

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

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

图 2: 解决方案乙示例演示过程

引例

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

图 2: 解决方案乙示例演示过程

引例

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

图 2: 解决方案乙示例演示过程

引例

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

图 2: 解决方案乙示例演示过程

引例

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

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

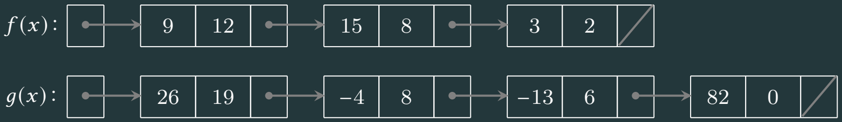


图 4: 解决方案丙示例结构

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

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

  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

顺序存储与运算实现

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

$s:$								

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

顺序存储与运算实现

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

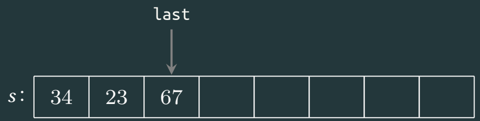


图 5: 顺序表存储过程演示

顺序存储与运算实现

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

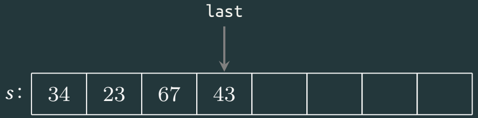


图 5: 顺序表存储过程演示

顺序存储与运算实现

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

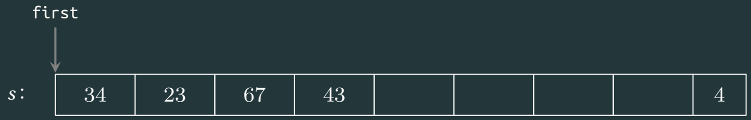


图 6: 顺序表的属性描述

顺序存储与运算实现

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

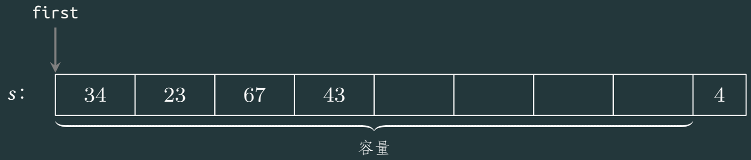


图 6: 顺序表的属性描述

顺序存储与运算实现

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

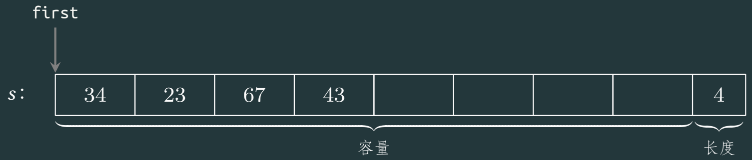


图 6: 顺序表的属性描述

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

顺序存储与运算实现

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?

$s:$

a_1	$\cdots$	a_{k-1}	a_k	$\cdots$	a_n	$\cdots$	n

图 8: 顺序表的随机访问原理

顺序存储与运算实现

思考：如何获取任意元素的存储地址?

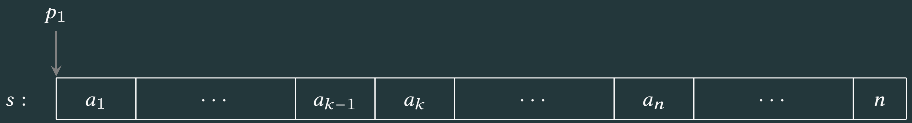


图 8: 顺序表的随机访问原理

顺序存储与运算实现

思考：如何获取任意元素的存储地址?

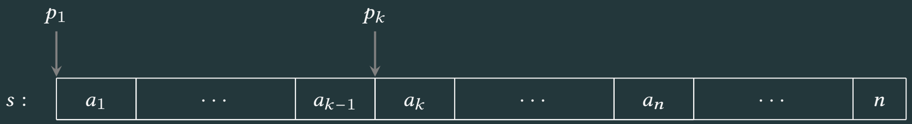


图 8: 顺序表的随机访问原理

顺序存储与运算实现

思考：如何获取任意元素的存储地址?

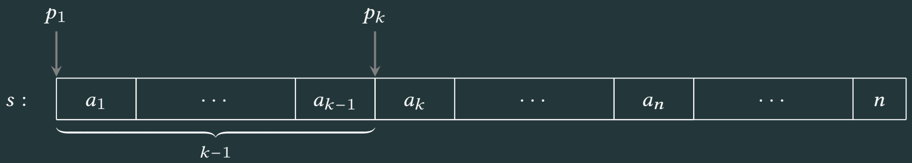


图 8: 顺序表的随机访问原理

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

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

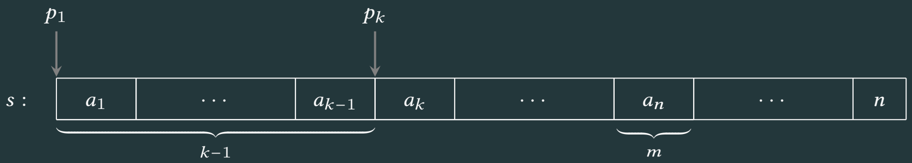


图 8: 顺序表的随机访问原理

顺序存储与运算实现

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

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

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

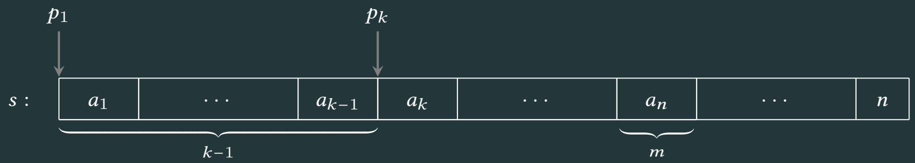


图 8: 顺序表的随机访问原理

顺序存储与运算实现

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

```c
#define CAPACITY 256

typedef int DataType; // 元素类型

typedef struct {
    DataType data[CAPACITY]; // 元素
    int last; // 尾元素序号
} SequenceList;
```

顺序存储与运算实现

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

```c
SequenceList *create_sequence_list() {
    SequenceList *s = malloc(sizeof(SequenceList));
    if (s) {
        s->last = -1;
    }
    return s;
}
```

顺序存储与运算实现

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

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

[3]为表示布尔型变量，须包含<stdbool.h>头，下同

顺序存储与运算实现

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

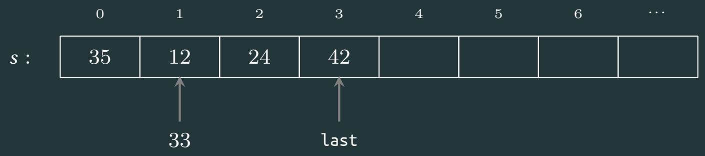


图 9: 在顺序表中插入元素过程演示

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

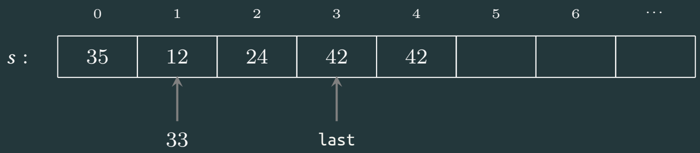


图 9: 在顺序表中插入元素过程演示

顺序存储与运算实现

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

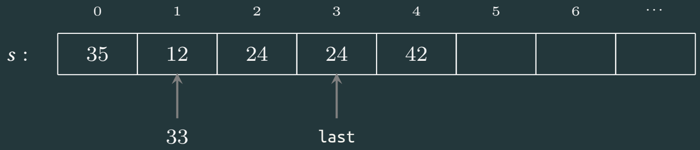


图 9: 在顺序表中插入元素过程演示

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

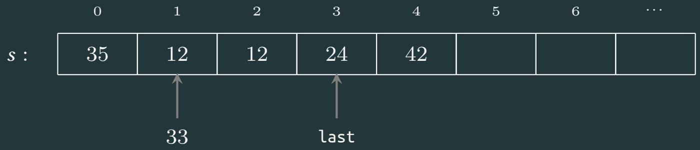


图 9: 在顺序表中插入元素过程演示

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

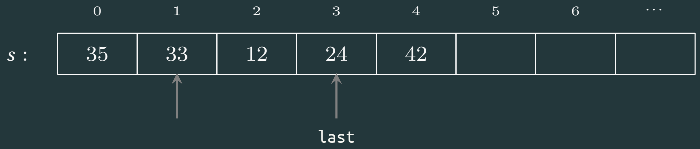


图 9: 在顺序表中插入元素过程演示

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

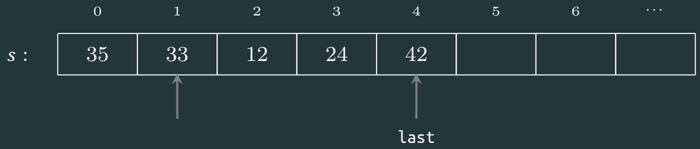


图 9: 在顺序表中插入元素过程演示

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

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

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

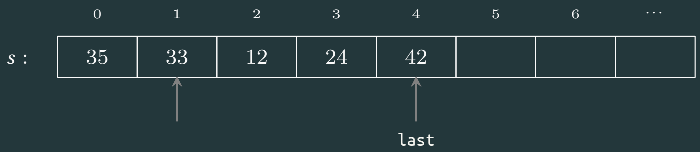


图 10: 从顺序表中删除元素过程演示

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

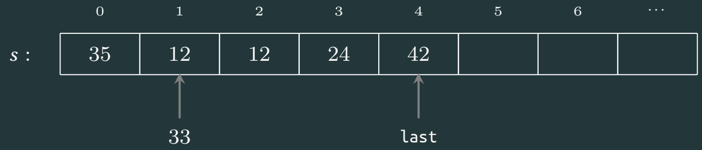


图 10: 从顺序表中删除元素过程演示

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

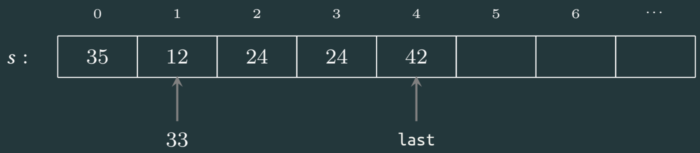


图 10: 从顺序表中删除元素过程演示

顺序存储与运算实现

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

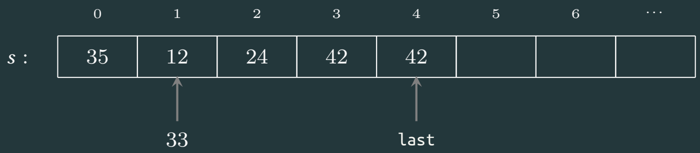


图 10: 从顺序表中删除元素过程演示

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

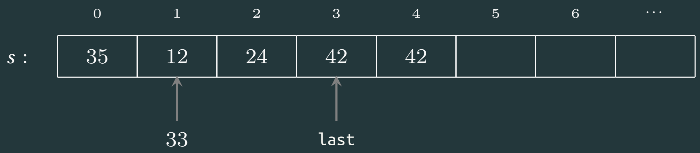


图 10: 从顺序表中删除元素过程演示

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

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

顺序存储与运算实现

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	$\cdots$
s :	35	33	12	24	42			

图 11: 在顺序表中查找元素过程演示

顺序存储与运算实现

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	$\cdots$
s :	35	33	12	24	42			

图 11: 在顺序表中查找元素过程演示

顺序存储与运算实现

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	$\cdots$
$s:$	35	33	12	24	42			

图 11: 在顺序表中查找元素过程演示

顺序存储与运算实现

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	$\cdots$
s :	35	33	12	24	42			

图 11: 在顺序表中查找元素过程演示

顺序存储与运算实现

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

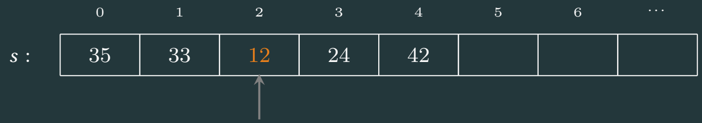


图 11: 在顺序表中查找元素过程演示

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，$T(n) = O(1)$

顺序存储与运算实现

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

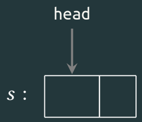


图 12: 单链表的存储过程演示

[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

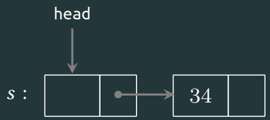


图 12: 单链表的存储过程演示

[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

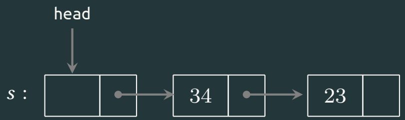


图 12: 单链表的存储过程演示

[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

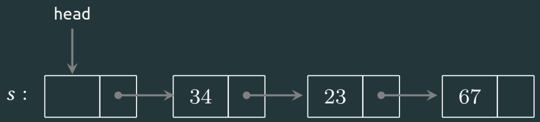


图 12: 单链表的存储过程演示

[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

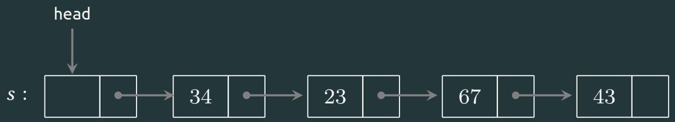


图 12: 单链表的存储过程演示

[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

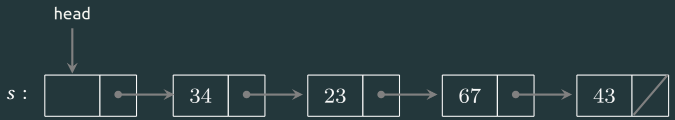


图 12: 单链表的存储过程演示

[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?

链式存储与运算实现

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

```c
typedef int DataType; // 元素类型

typedef struct LinkedNode {
    DataType data; // 数据元素
    struct LinkedNode *next; // 下一个结点
} LinkedNode;

typedef struct {
    LinkedNode *head; // 首结点
} 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

```c
LinkedNode *create_linked_node(DataType d) {
    LinkedNode *n = malloc(sizeof(LinkedNode));
    if (n) {
        n->data = d;
        n->next = NULL;
    }
    return n;
}
```

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

```
LinkedList *create_linked_list() {
    LinkedList *s = malloc(sizeof(LinkedList));
    if (s) {
        s->head = create_linked_node(0);
    }
    return s;
}
```

[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

```c
int get_linked_list_length(LinkedList *s) {
    int length = -1; // 计数不包括首结点
    for (LinkedNode *p = s->head; p != NULL;
            p = p->next, ++length)
        ; // 空语句
    return length;
}
```

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

```c
int get_linked_list_length(LinkedList *s) {
    int length = -1; // 计数不包括首结点
    for (LinkedNode *p = s->head; p != NULL;
            p = p->next, ++length)
        ; // 空语句
    return length;
}
```

说明

- $T(n) = O(n)$
- 注意与顺序表比较

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

```c
LinkedNode *search_linked_by_index(
        LinkedList *s, int k) {
    LinkedNode *p = s->head;
    for (int i = 0; p != NULL && i < k;
            ++i, p = p->next)
        ; // 空语句
    return p;
}
```

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

```c
LinkedNode *search_linked_by_index(
        LinkedList *s, int k) {
    LinkedNode *p = s->head;
    for (int i = 0; p != NULL && i < k;
            ++i, p = p->next)
        ; // 空语句
    return p;
}
```

```c
LinkedNode *search_linked_by_data(
        LinkedList *s, DataType d) {
    LinkedNode *p = s->head->next;
    for (; p != NULL && p->data != d; p = p->next)
        ; // 空语句
    return p;
}
```

链式存储与运算实现

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

```c
LinkedNode *search_linked_by_index(
        LinkedList *s, int k) {
    LinkedNode *p = s->head;
    for (int i = 0; p != NULL && i < k;
            ++i, p = p->next)
        ; // 空语句
    return p;
}
```

```c
LinkedNode *search_linked_by_data(
        LinkedList *s, DataType d) {
    LinkedNode *p = s->head->next;
    for (; p != NULL && p->data != d; p = p->next)
        ; // 空语句
    return p;
}
```

链式存储与运算实现

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

```c
LinkedNode *attach_after_node(
        LinkedNode *p, LinkedNode *n) {
    assert(p && n);
    n->next = p->next;
    p->next = n;
    return n;
}
```

```c
bool insert_after_linked_by_index(
        LinkedList *s, int k, DataType d) {
    LinkedNode *p = search_linked_by_index(s, k);
    if (!p) {
        printf("Wrong insert index!\n");
        return false;
    }
    LinkedNode *n = create_linked_node(d);
    return n && attach_after_node(p, n);
}
```

链式存储与运算实现

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

```
LinkedNode *attach_after_node(
        LinkedNode *p, LinkedNode *n) {
    assert(p && n);
    n->next = p->next;
    p->next = n;
    return n;
}
```

```
bool insert_after_linked_by_index(
        LinkedList *s, int k, DataType d) {
    LinkedNode *p = search_linked_by_index(s, k);
    if (!p) {
        printf("Wrong insert index!\n");
        return false;
    }
    LinkedNode *n = create_linked_node(d);
    return n && attach_after_node(p, n);
}
```

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

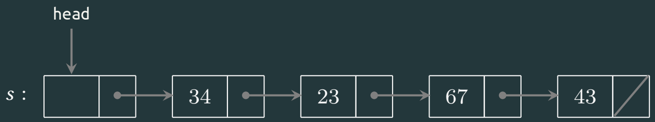


图 13: 在单链表结点后插入新结点的过程演示

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

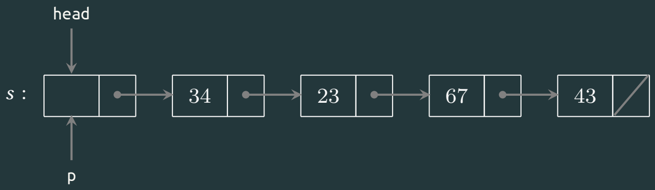


图 13: 在单链表结点后插入新结点的过程演示

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

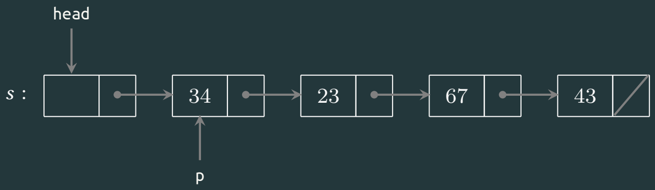


图 13: 在单链表结点后插入新结点的过程演示

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

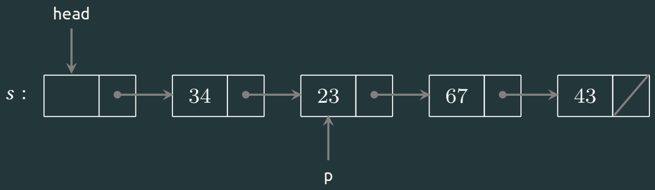


图 13: 在单链表结点后插入新结点的过程演示

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

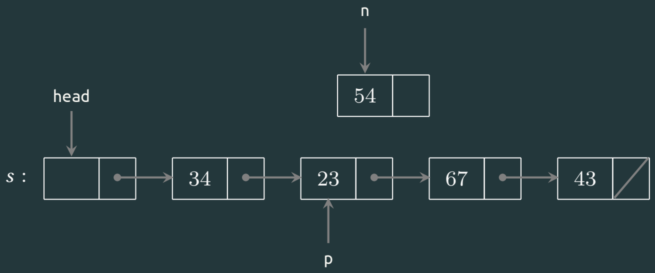


图 13: 在单链表结点后插入新结点的过程演示

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

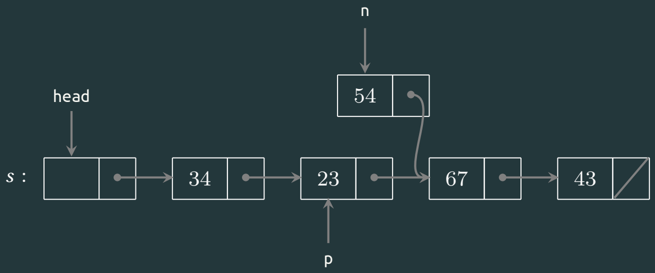


图 13: 在单链表结点后插入新结点的过程演示

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

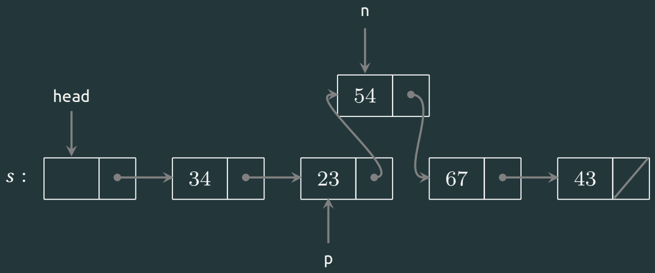


图 13: 在单链表结点后插入新结点的过程演示

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

```c
LinkedNode *detach_after_node(LinkedNode *q) {
    assert(q && q->next);
    LinkedNode *p = q->next;
    q->next = p->next;
    return p;
}
```

```c
bool remove_linked_by_index(
        LinkedList *s, int k, DataType *d) {
    LinkedNode *q = search_linked_by_index(s, k - 1);
    if (!q || !q->next) {
        printf("Wrong remove index!\n");
        return false;
    }
    LinkedNode *p = detach_after_node(q);
    if (d) { *d = p->data; }
    free(p);
    return true;
}
```

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

```c
LinkedNode *detach_after_node(LinkedNode *q) {
    assert(q && q->next);
    LinkedNode *p = q->next;
    q->next = p->next;
    return p;
}
```

```c
bool remove_linked_by_index(
        LinkedList *s, int k, DataType *d) {
    LinkedNode *q = search_linked_by_index(s, k - 1);
    if (!q || !q->next) {
        printf("Wrong remove index!\n");
        return false;
    }
    LinkedNode *p = detach_after_node(q);
    if (d) { *d = p->data; }
    free(p);
    return true;
}
```

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

例

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

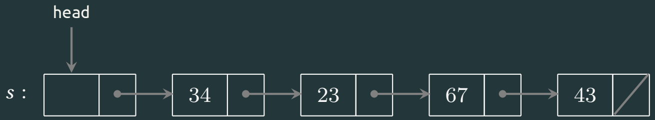


图 14: 删除单链表结点过程演示

例

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

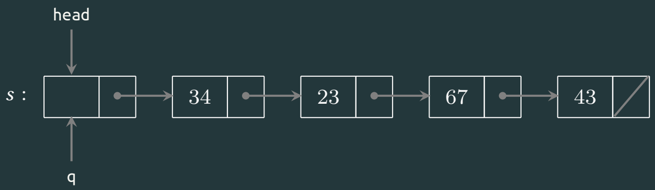


图 14: 删除单链表结点过程演示

例

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

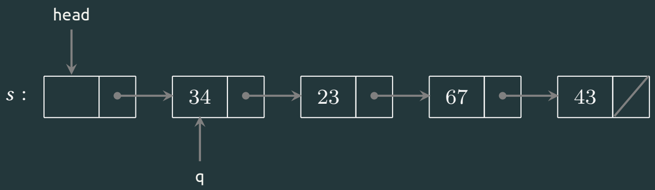


图 14: 删除单链表结点过程演示

例

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

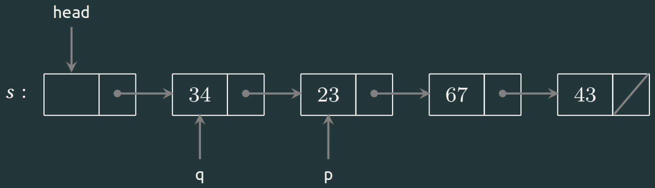


图 14: 删除单链表结点过程演示

例

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

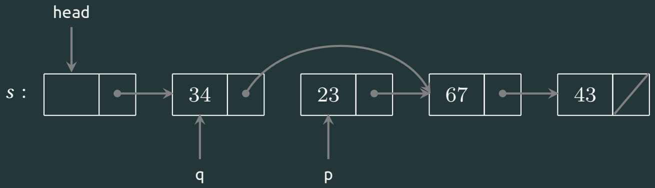


图 14: 删除单链表结点过程演示

例

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

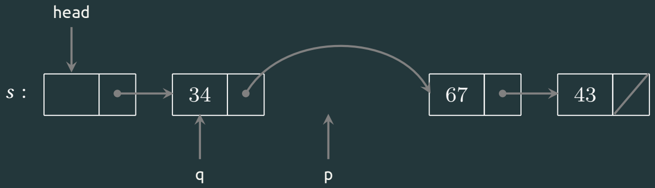


图 14: 删除单链表结点过程演示

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

链式存储与运算实现

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)**

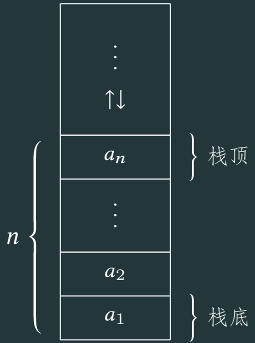


图 15: 栈的示意图

栈

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
- ...

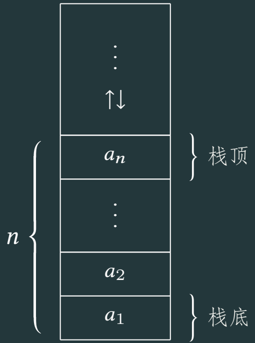


图 15: 栈的示意图

栈的抽象数据类型

```
ADT Stack {
数据:
    数据对象: D = {a_k|a_k ∈ 数据元素集合, k ∈ ℤ∩[1,n]}
    逻辑关系: R = {⟨a_{k-1}, a_k⟩|k ∈ ℤ∩[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存在且未满，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

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

```c
typedef struct {
    SequenceList *_;
} SequenceStack;
```

```c
typedef struct {
    LinkedList *_;
} 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

```c
SequenceStack *create_sequence_stack() {
    SequenceStack *s = malloc(sizeof(SequenceStack));
    if (s) {
        s->_ = create_sequence_list();
    }
    return s;
}
```

```c
LinkedStack *create_linked_stack() {
    LinkedStack *s = malloc(sizeof(LinkedStack));
    if (s) {
        s->_ = create_linked_list();
    }
    return s;
}
```

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

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

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

栈

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

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

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

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

```c
bool pop_sequence_stack(
        SequenceStack *s, DataType *p) {
    return s && remove_sequence_list(
            s->_, s->_->last, p);
}
```

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

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

```c
bool top_sequence_stack(
        SequenceStack *s, DataType *p) {
    if (empty_sequence_stack(s) || !p) {
        return false;
    }
    *p = s->_->data[s->_->last];
    return true;
}
```

```c
bool top_linked_stack(
        LinkedStack *s, DataType *p) {
    if (empty_linked_stack(s) || !p) {
        return false;
    }
    *p = s->_->head->next->data;
    return true;
}
```

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

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

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

栈

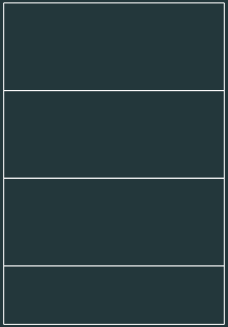


图 16: 函数调用栈：

```
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  }
```

栈

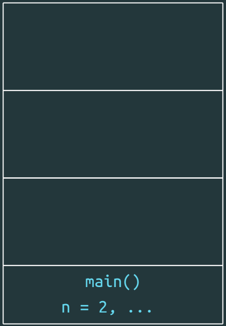


图 16: 函数调用栈：递进调用

```c
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  }
```

栈

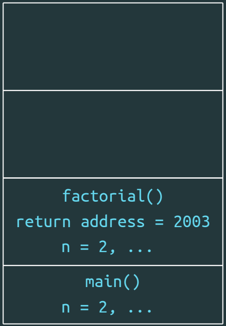


图 16: 函数调用栈：递进调用

```
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  }
```

栈

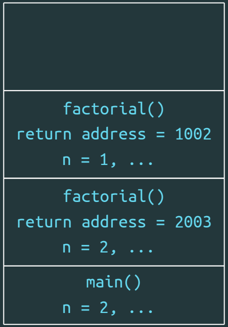


图 16: 函数调用栈：递进调用

```
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 }
```

栈

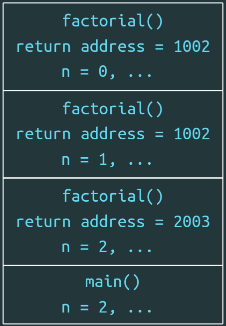


图 16: 函数调用栈：递进调用

```
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  }
```

栈

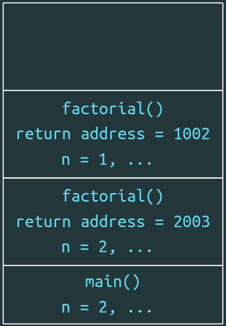


图 16: 函数调用栈：回归求值

```
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  }
```

栈

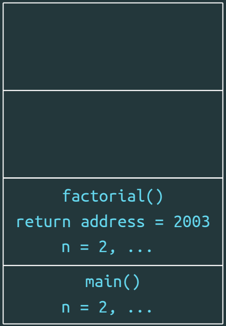


图 16: 函数调用栈：回归求值

```
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 }
```

栈

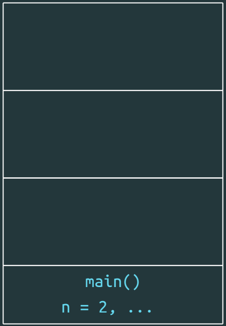
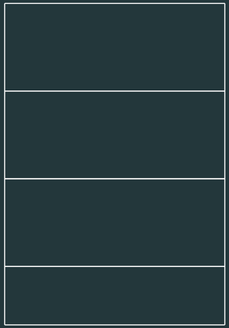


图 16: 函数调用栈：回归求值

```
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  }
```

栈



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

```c
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

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

栈

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

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

```c
void hanoi(int n, char a, char b, char c) {
    if (!n) {
        return; // 递归出口
    }
    hanoi(n - 1, a, c, b);
    move(a, c);
    hanoi(n - 1, b, a, c);
}
```

栈

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)**

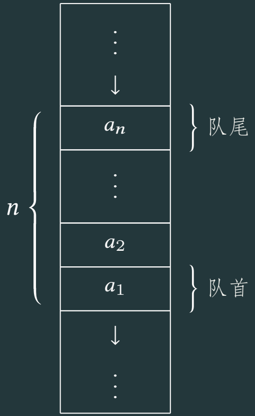


图 17: 队列的示意图

队列

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
- . . .

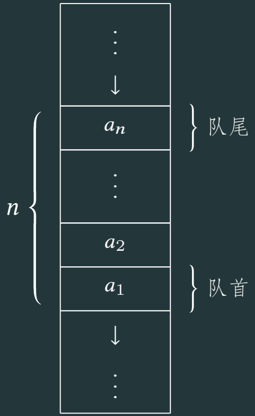


图 17: 队列的示意图

队列的抽象数据类型

ADT Queue {

数据:

  数据对象: $\mathcal{D} = \{a_k | a_k \in$ 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

}

队列

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

```
typedef struct {
    SequenceList *_;
} SequenceQueue;
```

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

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

队列

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

队列

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

```c
typedef struct {
    SequenceList *_;
    int front;
} SequenceQueue;
```

```c
bool pop_sequence_queue(
        SequenceQueue *s, DataType *d) {
    if (!s) {
        return false;
    }
    if (d) {
        *d = s->_->data[front];
    }
    return ++front;
}
```

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

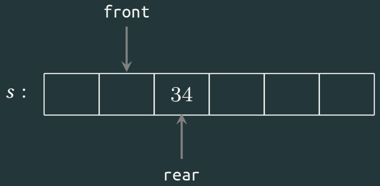


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

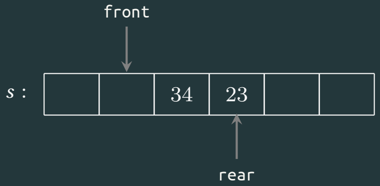


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

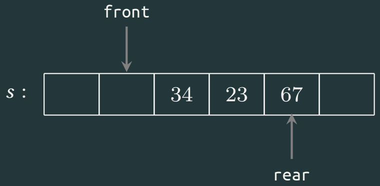


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

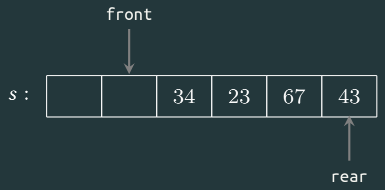


图 18: 队列的假上溢现象及其解决思路

假上溢现象

- 队首前尚有可用空间，但队尾已至序列尽头

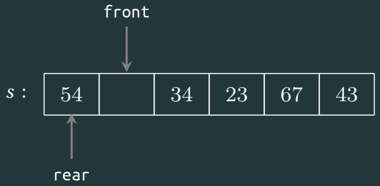


图 18: 队列的假上溢现象及其解决思路

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

队列

新问题：如何判断队列已空或已满?

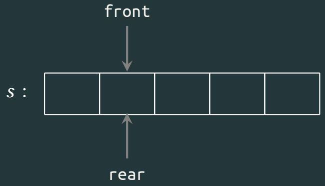


图 19: 队列已空

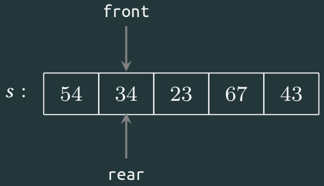


图 20: 队列已满

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）

队列

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

```
typedef struct {
    DataType data[CAPACITY];
    int front; // 队首
    int rear; // 队尾
    int size; // 队列长度
} 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

```
SequenceQueue *create_sequence_queue() {
    SequenceQueue *s = malloc(sizeof(SequenceQueue));
    if (s) {
        s->front = s->rear = CAPACITY - 1;
        s->size = 0;
    }
    return s;
}
```

队列

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

```c
bool push_sequence_queue(
        SequenceQueue *s, DataType d) {
    if (full_sequence_queue(s)) {
        return false;
    }
    s->rear = (s->rear + 1) % CAPACITY;
    s->data[s->rear] = d;
    return ++s->size;
}
```

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

```
bool pop_sequence_queue(
        SequenceQueue *s, DataType *d) {
    if (empty_sequence_queue(s)) {
        return false;
    }
    s->front = (s->front + 1) % CAPACITY;
    if (d) {
        *d = s->data[s->front];
    }
    --s->size;
    return true;
}
```

[7]注：队首一般指向首元素前一个位置

队列

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

```c
typedef struct {
    LinkedList *_;
    LinkedNode *front; // 队首
    LinkedNode *rear; // 队尾
} LinkedQueue;
```

队列

链式队列

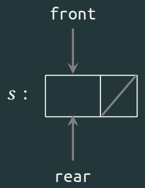


图 21: 空链式队列

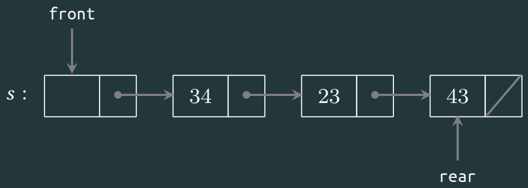


图 22: 非空链式队列

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

```c
LinkedQueue *create_linked_queue() {
    LinkedQueue *s = malloc(sizeof(LinkedQueue));
    if (s) {
        s->_ = create_linked_list();
        s->front = s->rear = s->_->head;
    }
    return s;
}
```

链式队列的判空

- 队首与队尾是否重合

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

入队

- 创建新结点
- 在链表末端插入新结点[8]
- 队尾指针后移

```
bool push_linked_queue(
        LinkedQueue *s, DataType d) {
    if (!s) {
        return false;
    }
    LinkedNode *n = create_linked_node(d);
    return n && (s->rear =
        attach_after_node(s->rear, n));
}
```

[8]注：此时已有队尾指针记录队尾结点，故无需每次查找之

队列

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

```c
bool pop_linked_queue(
        LinkedQueue *s, DataType *d) {
    if (empty_linked_queue(s)) {
        return false;
    }
    LinkedNode *p = detach_after_node(s->front);
    if (d) {
        *d = p->data;
    }
    free(p);
    if (empty_linked_queue(s)) {
        s->rear = s->front;
    }
    return true;
}
```

常用的线性结构

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

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