数据结构

第 11 周 – 基于链表的栈和队列；哈希表

Koczka Ferenc 博士 · 埃斯特哈齐·卡罗伊大学 · 2026

复习

- 用数组实现的**栈**（第 7 周）和**队列**（第 8 周）：大小固定，有“满”的状态
- **链表**（第 9–10 周）：在表头 O(1)，有 End 指针时在表尾也是 O(1)
- **抽象数据类型**：重要的是操作，而不是实现

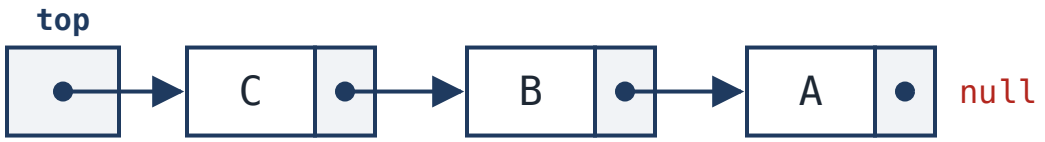
今天的问题

- 怎样用链表实现栈和队列?
- 能否在 **O(1)** 时间内找到一个元素 – 不用排序和二分查找?

用链表实现栈和队列

用链表实现栈

链栈：在表头 push 和 pop – 均为 O(1)



`push(x)` = 表头插入

`pop()` = 表头删除

永远不会“满”

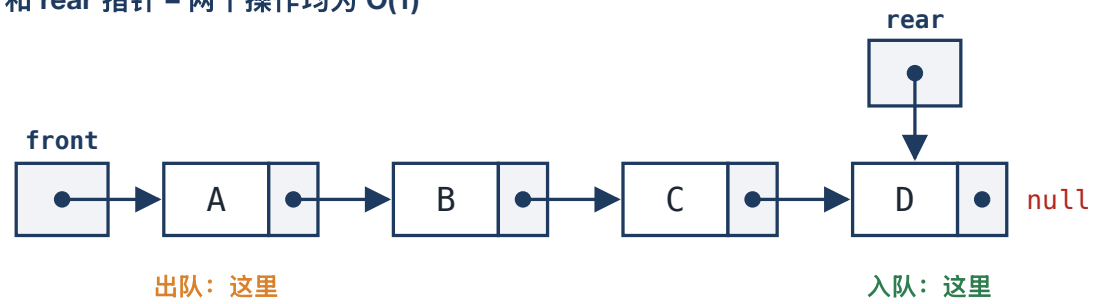
```
Procedure Push(Top, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Top
    Top ← P
  else ERROR
End procedure
```

```
Function Pop(Top)
  If Top = Nil then ERROR
  x ← Elem[Top].Value
  P ← Top
  Top ← Elem[Top].Next
  Free(P)
  Return x
End function
```

这正是第 9 周的“在表头插入”和“删除表头”。

用链表实现队列

链队列：front 和 rear 指针 – 两个操作均为 O(1)




```
Procedure Enqueue(Front, Rear, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Nil
    If Rear = Nil then Front ← P
    else Elem[Rear].Next ← P
    Rear ← P
  else ERROR
End procedure
```

```
Function Dequeue(Front, Rear)
  If Front = Nil then ERROR
  x ← Elem[Front].Value
  P ← Front
  Front ← Elem[Front].Next
  If Front = Nil then Rear ← Nil
  Free(P)
  Return x
End function
```

队列变空时，**Rear** 指针也必须设为 Nil!

一个接口，两种实现

Java

```java
interface Stack<T> {
    void push(T x);
    T pop();
    boolean isEmpty();
}

class LinkedStack<T> implements Stack<T> {
    private Node<T> top;
    public void push(T x) { top = new Node<>(x, top); }
    public T pop() {
        T x = top.value;
        top = top.next;
        return x;
    }
    public boolean isEmpty() { return top == null; }
}

// 使用者只知道接口：
String reverse(String s, Stack<Character> st)
```

Go

```go
type Stack[T any] interface {
    Push(x T)
    Pop() (T, error)
    IsEmpty() bool
}

type LinkedStack[T any] struct{ top *node[T] }

func (s *LinkedStack[T]) Push(x T) {
    s.top = &node[T]{x, s.top}
}

// 可以传入任意一种实现：
reverse("数据结构", &LinkedStack[rune]{})
reverse("数据结构", &SliceStack[rune]{})
```

数组实现还是链表实现?

	用数组	用链表
Push / Pop、Enqueue / Dequeue	O(1)	O(1)
大小	固定 – 可能出现“满”	**不受限制**（只要还有内存）
每个元素的空间	只有值	值 + **指针**
未使用的空间	数组的空闲部分	无
缓存友好	**是**（连续）	较差（分散）
每个元素的分配	无	每次 push/enqueue 都要分配

> 实际中最常用的是**动态（循环）数组**实现（Java `ArrayDeque`、Go 切片） – 快且节省内存。

哈希表

问题：快速查找

到目前为止查找一个元素的方法：

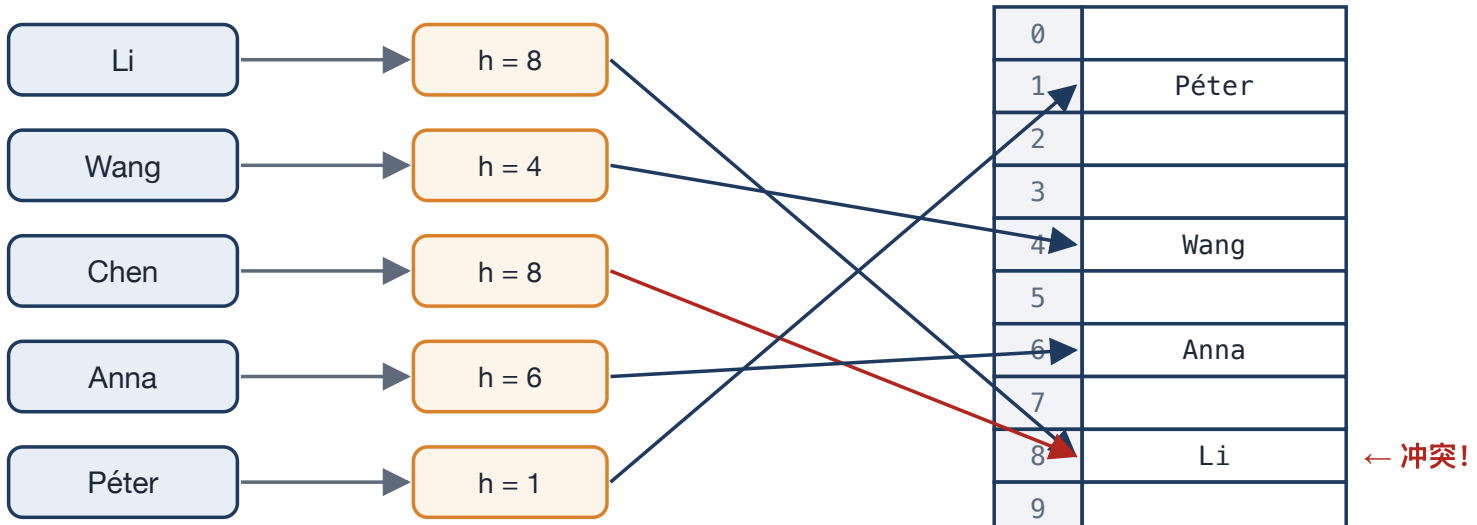
方法	查找	插入
无序数组 / 链表	O(n)	O(1)
有序数组 + 二分查找	O(log n)	O(n)
位向量（第 6 周）	**O(1)**	**O(1)**

位向量很快，但只适用于**很小且已知**的全集。

> **思路：**用一个函数把键（任意大的数或字符串）转换成一个**较小的下标**，把元素存放在数组的这个下标处。

哈希函数

哈希函数：键 → 下标（0 … m – 1），这里 m = 11



“Chen”和“Li”映射到同一个下标

好的哈希函数

哈希函数： h：键 → {0, 1, …, m – 1}

要求：

- 计算**快** – O（键的长度）
- 把键**均匀**地分布到各个桶中
- **确定性：**相同的键总是得到相同的结果

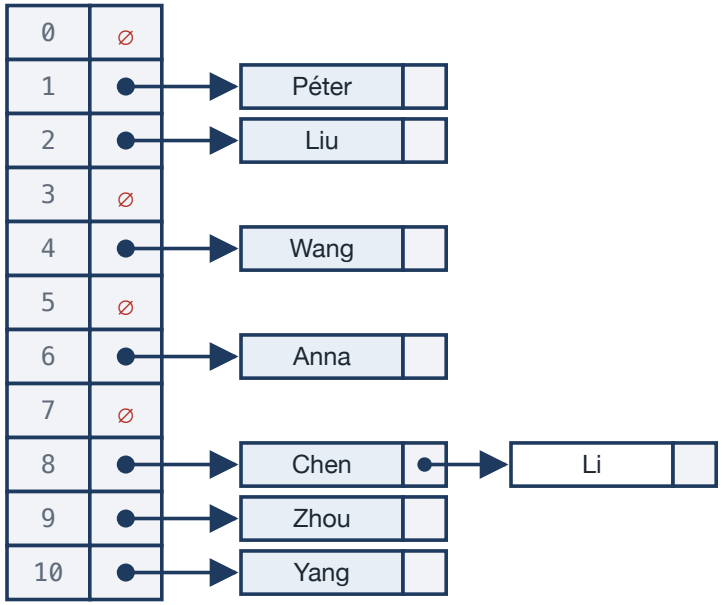
例子：

- 整数键：**h(k) = k mod m**（m 最好是素数）
- 字符串：**多项式哈希**：对每个字符 c,h = (h · 31 + c) mod m
  Java: `"Li".hashCode()` = 'L' · 31 + 'i' = 76 · 31 + 105 = **2461**

冲突：两个不同的键得到相同的哈希值。冲突是不可避免的（键比桶多） – 必须**处理**！

冲突处理 1：链地址法

链地址法：每个桶是一个链表



插入：插到链表 h(k) 的表头

查找：只在链表 h(k) 中

删除：从链表 h(k) 中删除

装填因子：α = n / m = 8 / 11 ≈ 0.73

平均链长 = α

链地址法 – 操作

伪代码

```
Procedure Insert(T, key, value)
  i ← h(key)
  P ← Search(T[i], key)        // 在链表中查找
  If P ≠ Nil then
    Elem[P].Value ← value      // 修改
  else
    InsertFirst(T[i], key, value)
  End if
End procedure

Function Find(T, key)
  Return SearchList(T[h(key)], key)
End function
```

Java

```java
void put(String key, int value) {
    int i = hash(key);
    for (Entry e = buckets[i]; e != null; e = e.next)
        if (e.key.equals(key)) { e.value = value; return; }
    buckets[i] = new Entry(key, value, buckets[i]);
    size++;
    if (size > 0.75 * buckets.length) resize();
}

Integer get(String key) {
    for (Entry e = buckets[hash(key)];
         e != null; e = e.next)
        if (e.key.equals(key)) return e.value;
    return null;
}
```

装填因子与再哈希

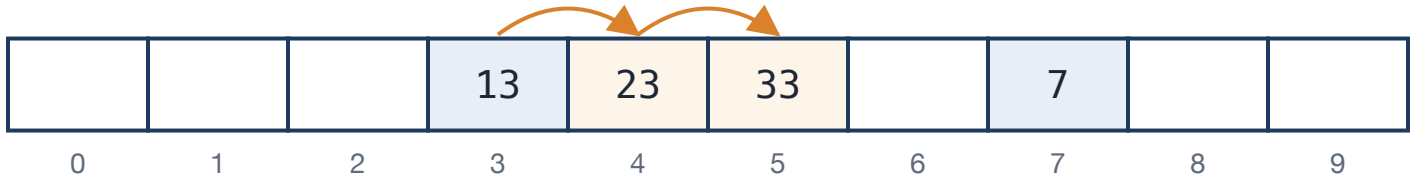
装填因子： α = n / m（元素个数 / 桶的个数）

- 用链地址法时平均链长为 **α** → 查找平均为 **O(1 + α)**
- 如果 α 太大（例如 > 0.75）： **再哈希** – 建立一个两倍大的表，**所有**元素重新放置（哈希值依赖于 **m**!）
- 与动态数组一样：一次扩容是 O(n)，但**均摊 O(1)**

例（`HashTable` 程序）： m = 11,8 个元素（α = 0.73） – 一次冲突（Li、Chen）
插入第 9 个元素后 α = 0.82 > 0.75 → m = 22,α = 0.41 – 没有冲突

冲突处理 2：开放定址法

开放定址法，线性探测：h(k) = k mod 10；冲突时取下一个空位



13、23、33 的哈希值都是 3 → 放在 3、4、5

查找：从 h(k) 开始探测，直到找到或遇到空位

- 所有元素都存放在**数组本身**中 – 没有链表
- 冲突时寻找**下一个**空位（线性探测）
- **删除：**不能简单地写成“空” – 那样会破坏查找链 → 需要“已删除”标记

开放定址法：装填因子的影响

查找成功时的平均探测次数（实测值，`LinearProbing` 程序）：

α	0.25	0.50	0.75	0.90	0.99
探测次数	1.17	1.50	2.48	5.52	**36.16**

- 装填因子较低时，几乎总是**第一次**就能找到
- 随着表被填满，被占用的位置形成**聚集** → 急剧变慢
- 开放定址法必须 $\alpha < 1$；实际中 $\alpha \le 0.5$–0.75

哈希表的效率

操作	平均情况	最坏情况
查找	**O(1)**	O(n)
插入	**O(1)**（均摊）	O(n)
删除	**O(1)**	O(n)

- 最坏情况：**所有**键都落入同一个桶（哈希函数很差，或者恶意输入）
- **没有顺序**：最小值、最大值、有序遍历分别是 O(n) / O(n log n)
- 需要有序的键时使用**二叉搜索树**（第 15 周，Java `TreeMap`）

内置的哈希表

Java – HashMap、HashSet

```java
Map<String, Integer> freq = new HashMap<>();
for (String w : text.split(" "))
    freq.merge(w, 1, Integer::sum); // freq[w]++

freq.get("fox");                // 2
freq.getOrDefault("cat", 0);    // 0
freq.containsKey("dog");        // true
freq.remove("the");

Set<String> seen = new HashSet<>();
```

键的类的 `hashCode()` 和 `equals()` 方法必须**保持一致**！

Go – map

```go
freq := map[string]int{}
for _, w := range strings.Fields(text) {
    freq[w]++           // 不存在的键:0
}

v, ok := freq["cat"]   // “comma ok”写法
delete(freq, "the")
len(freq)

seen := map[string]bool{}
```

map 的遍历顺序是**随机**的 – 这是有意设计的！

应用： 字典、频率统计、缓存、数据库索引、去重、编译器的符号表

小结

- **链栈：**在链表表头 push/pop · **链队列：**Front + Rear 指针 – 都是 O(1)，没有“满”的状态
- 同一个**接口**可以有多种实现
- **哈希表：**用哈希函数由键得到下标 → 查找、插入、删除平均 **O(1)**
- **冲突处理：**链地址法（每个桶一个链表）或开放定址法（下一个空位）
- **装填因子** α = n/m;α 太大 → 再哈希
- Java `HashMap` / `HashSet` · Go `map`

下周：递归。

复习题

1. 为什么链栈在链表的**表头**而不是表尾操作?
2. 在链队列中，如果删除最后一个元素时忘记把 Rear 指针设为 Nil，会怎样?
3. h(k) = k mod 7，链地址法。插入：10、3、17、24、5!画出哈希表!
4. 同样的键，用线性探测，m = 7 – 24 放在哪里?
5. 判断对错：哈希表的查找时间即使在最坏情况下也是 O(1)。
6. 为什么在哈希表中不能高效地找到最小的键?

实践课

✎ 示例程序

程序	Java	Go	演示内容
链栈与链队列	`LinkedStackQueue.java`	`w11/linkedstackqueue`	接口，两种实现
哈希表	`HashTable.java`	`w11/hashtable`	链地址法、冲突、再哈希
线性探测	`LinearProbing.java`	`w11/linearprobing`	开放定址法、装填因子的影响
单词频率	`WordCount.java`	`w11/wordcount`	使用 `HashMap` / `map`

✎ 练习

1. **两数之和：**数组中是否有两个元素之和等于给定的值？用哈希表 O(n)，而不是 $O(n^2)$！
2. **第一个不重复的字符：**哪个字符第一个只出现一次？（`"swiss" → 'w'`）
3. **变位词分组：**把单词分组，使互为变位词的单词在同一组！
4. **重复元素：**数组中是否有重复的元素？（平均 O(n)）
5. **括号：**第 7 周的题目，括号对存放在 map 中。
6. **电话簿：**姓名 → 电话号码，按姓名查找。

参考答案：`Exercises11.java`、`w11/exercises`

✎ 课后作业

1. 为我们自己的 `HashTable` 类增加一个返回所有键的 `keys()` 方法！需要多少步？
2. 在线性探测哈希表中用“已删除”标记实现**删除**！查找和插入怎样变化？
3. 对 10 000 个随机单词，用两种哈希函数测量你的哈希表的冲突次数：（a） 字符编码之和，（b） 多项式哈希！你观察到了什么？
4. 用哈希表和双向链表（见第 10 周的课后作业）实现 **LRU 缓存**，使 `get` 和 `put` 都是 O(1)！