数据结构与算法

第 10 讲：散列表

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

10.1 问题引入

10.1.1 二叉查找树的不足

上一讲学习了二叉查找树，自平衡二叉查找树（如 AVL 树、红黑树等）能够以 $O(\log n)$ 的时间复杂度进行查找、插入、删除等操作，并维护元素的顺序。

在很多应用场景中，维护元素的顺序并不重要（如词频统计），关键是尽可能快地完成查找、插入、删除等操作。有什么可能的改进思路?

使用基于比较的方法，每次比较除相等外，只可能获得两种不同的结果（小于和大于），这是限制所在。计算机系统的存储器支持随机访问，能否利用这一特性设计更加高效的查找操作?

10.1.2 关键字与存储位置的映射

考虑在关键字的值和元素的存储位置之间建立直接的映射关系。

例如，元素的关键字取值范围是连续整数区间 $[0, n)$ 的情况。假设元素的关键字各不相同。如何针对这种情况设计高效的动态查找表?

可以利用计算机存储器的随机访问能力，设置一个长度为 n 的数组。此时，查找关键字为 k 的元素可以直接访问数组中下标为 k 的元素，插入和删除也可以通过添加标志位来表示关键字为 k 的元素是否存在。这三种操作均可以在 $O(1)$ 时间复杂度完成。

10.1.3 进一步扩展

如果要将上述方法扩展到更加一般的情况，会面临一些问题：

1. 关键字的取值不是连续的整数区间，甚至不是整数。
2. 有多个关键字相同的元素。

要使上述方法能够支持更加一般的情况，需要解决这些问题。本讲学习一种新的动态查找表——散列表。

10.2 散列表的概念

10.2.1 散列表的相关定义

定义 1 散列表（也叫哈希表，hash table）将关键字映射到表中的一个存储位置，进行查找和修改操作。从关键字到存储位置的映射函数称为**散列函数**（也叫哈希函数，hash function）。由散列函数计算得到的存储位置称为**散列地址**。

形式化地，设地址空间 H 是长度为 n 的表，K 是含有 m 个元素的关键字集合，散列查找的核心就是在 K 和 H 之间建立一种函数关系 h，使得对 K 中任意一个关键字 k，均有 $0 \leq h(k) < n$。

10.2.2 散列表示例

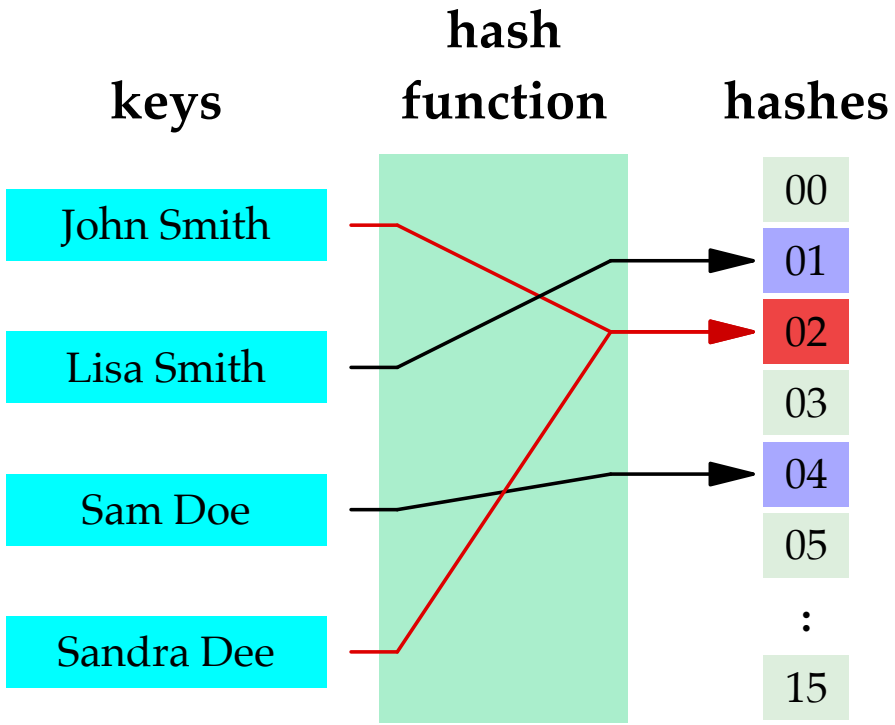

图 10.1　一个通讯录的散列表

10.2.3 散列函数的性质

散列函数不一定是单射函数（一对一）。例如，当 $m > n$ 时，由抽屉原理可知至少存在两个关键字 $k_1 \neq k_2$，使得 $h(k_1) = h(k_2)$。

这种对不同关键字得到同一地址的现象，称为**冲突**（也叫碰撞，collision）。即使 $m < n$，冲突的情况也有可能发生。

例. 已知 $n = 10$，存储空间的地址分别为 $0, 1, 2, ..., 9$，$K = \{1, 6, 11, 16\}$，如果取散列函数 $h(k) = k \bmod n$，则 $h(1) = h(11), h(6) = h(16)$。

一般来说，要找到一个散列函数对所有元素都不冲突是很困难的。

10.2.4 散列表的主要技术问题

在使用散列表时，需要解决两个主要技术问题：

1. 散列函数的构造方法：散列函数构造得好，能够尽可能地减少冲突。
2. 解决冲突的方法：解决冲突的方法选得好，能在尽可能短的时间里完成查找。

10.3 散列表的实现

10.3.1 散列函数的构造

什么是“好”的散列函数?

1. 计算简单容易：由于对散列表的每次操作都要计算散列函数，散列函数的时间复杂度一般不能高于后续的查找操作。
2. 很少有冲突：散列函数应该尽可能均匀地把关键字集合中的关键字映射到地址空间中。

常用的散列函数构造方法有：

- **直接地址存储法**
- **除留取余法**
- **折叠法**
- 数字分析法
- 平方取中法
- 随机数法

10.3.2 直接地址存储法

直接地址存储法取关键字本身或关键字的某个线性函数值作为散列地址，即

$$h(k) = ak + b$$

其中，a, b 为常数。

例. 有一份 1949 年以来的出生人口调查表，每条数据都包含出生年份、出生人数等数据项。以出生年份为关键字，则散列函数可取为 $h(k) = k + (-1949)$。

10.3.3 除留取余法

直接地址存储法得到的散列地址的值域范围可能会很大，为了将散列地址控制在一定范围内，除留取余法取关键字被某个不超过散列表长度 n 的数 p 除后的余数作为散列地址，即

$$h(k) = k \bmod p$$

其中 $p \leq n$。 可以将除留取余法与直接地址存储法结合起来，即

$$h(k) = (ak + b) \bmod p$$

在选择 p 的取值时，一般会选取素数，让关键字更加均匀地分布。

10.3.4 折叠法

当关键字位数较长时（例如字符串），可以将关键字看成一个多位数，给每位赋一定的权值，把结果相加，即

$$h(k)=\sum_{i=0}^{l-1}k_i w^i$$

其中 l 是 k 的长度，k_i 是 k 的第 i 位，w 是权重，一般取像 $17,31$ 等较小的素数。

折叠法也可以和除留取余法结合，将结果控制在一定范围内。

10.3.5 折叠法示例：字符串散列函数

伪码 10.1: 字符串散列函数示例

```
StringHash(S, w, p):
1   h ← 0
2   l ← Length(S)
3   for i ← 0 to l − 1
4       h ← (w · h + S[i]) mod p
5   return h
```

10.3.6 散列冲突的解决方法

当关键字得到的散列地址上已经有其他数据时，发生散列冲突。当散列表的项数一定时，冲突无法避免。

解决散列冲突的基本思想是要给冲突元素找到可以存放的空间，寻找的方式可以是在散列表内找（开放地址法、再散列法等），也可以扩展散列表中的项（链地址法、公共溢出区法等），课内主要介绍链地址法。

10.3.7 链地址法

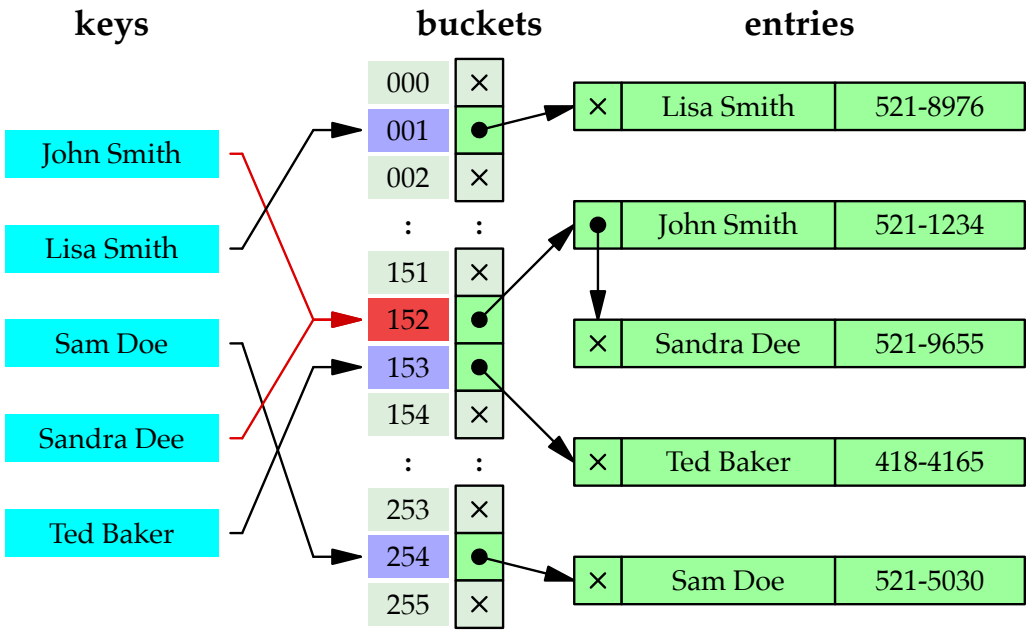

图 10.2　链地址法示意

链地址法将关键字冲突的元素存储在同一个链表中，此时散列表中的每个项不再是单个元素，可以存储一组元素，称为一个**桶**（bucket）。

在查找元素时，先根据关键字计算散列地址，按散列地址在对应的链表（桶）中查找元素。

伪码 10.2: 散列表的查找（链地址法）

HashSearch(H, x):

1 $n \leftarrow \text{Length}(H)$ # 散列表的桶数

2 $i \leftarrow \text{Hash}(x) \bmod n$

3 **return** SequentialSearch$(H[i], x)$

10.3.9 散列表的构造举例

例. 给出序列 $19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79$，散列表的长度为 13，散列函数为 $h(k) = k \bmod 13$，采用链地址法解决冲突，求依次插入这些元素后散列表的情况。

10.3.10 散列表的效率与装填因子

散列表的查找效率取决于三个因素：散列函数、冲突处理的方法和散列表的装填因子。散列表的装填因子定义为

$$\alpha = \frac{\text{填入的元素个数}}{\text{散列表的长度（也就是桶的个数）}}$$

α 表示散列表装满的程度。α 越小，散列表中填入的元素越少，发生冲突的可能性就越小；反之，α 越大，散列表中填入的元素越多，发生冲突的可能性就越大。

使用链地址法解决冲突时，查找成功的平均查找长度为 $1 + \frac{\alpha}{2}$，查找失败的平均查找长度为 $\alpha + e^{\alpha}$。注意与散列表的长度 n 无关。

10.3.11 C++ 中的散列表

C++ 对散列表进行了封装，提供 unordered_set、unordered_multiset、unordered_map 和 unordered_multimap 这四个类，提供了与 map 等类似的接口，也同样属于关联容器。

与 map 等不同的地方是，unordered_map 不需要元素类型支持比较大小运算，而是需要支持散列函数（通过 std::hash<T>）和比较相等运算。对于 C++ 内置数据类型以及 std::string 和 std::string_view 等常用类型，都已经实现了 std::hash。

C++ 标准库中这些容器的实现一般采用链地址法解决冲突，桶的个数会根据填入元素的个数自动调整以维持合适的装填因子。

10.4 散列表的应用

10.4.1 词频统计：使用散列表实现

C++

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <unordered_map>  // 修改 1
#include <vector>

const std::size_t kNumTopWords = 10;

int main() {
  std::unordered_map<std::string, std::size_t> count_map;  // 修改 2
  std::string word;

  // 依次输入每个单词，统计出现次数
  while (std::cin >> word) {
```

10.4.1 词频统计：使用散列表实现

```
15      ++count_map[word];
16    }
17
18    // 对统计结果按出现次数排序，出现次数相同按字典序排序
19    std::vector<std::pair<std::string, std::size_t>> count_vec(
20        count_map.begin(), count_map.end());
21    std::sort(count_vec.begin(), count_vec.end(),
22              [](const auto& a, const auto& b) {
23                return a.second > b.second ||
24                       (a.second == b.second && a.first < b.first);
25              });
26
27    // 输出部分频率最高的单词
28    auto end = count_vec.begin() + kNumTopWords;
```

10.4.1 词频统计：使用散列表实现

```
29    for (auto it = count_vec.begin(); it != end; ++it) {
30      std::cout << it->first << ": " << it->second << '\n';
31    }
32    std::cout << "不同的单词数量: " << count_map.size() << '\n';
33  }
```

10.4.2 使用自定义类型

std::unordered_map 的声明如下：

```cpp
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
> class unordered_map;
```

对于自定义类型，可以通过特化实现 std::hash 的方式来定义散列函数，也可以在定义散列表时在类的模板参数中指定散列函数。

10.4.3 自定义类型：实现

C++

```cpp
#include <cassert>
#include <print>
#include <set>
#include <string>
#include <unordered_set>

struct Point {
    int x;
    int y;
    // 自动生成比较运算符 (C++20)
    auto operator<=>(const Point&) const = default;
};

// 为 Point 类定义散列函数
```

10.4.3 自定义类型：实现

```
15  template <>
16  struct std::hash<Point> {
17    size_t operator()(const Point& p) const { return p.x + p.y; }
18    // 这里如果使用 p.x - p.y 或 p.x ^ p.y 会使性能变得非常差，思考原因？
19  };
20
21  template <class Container>
22  void Bench(Container& points, int n) {
23    for (int i = 0; i < n; i++) {
24      points.insert(Point{i, i});
25    }
26    for (int i = 0; i < n; i++) {
27      assert(points.contains(Point{i, i}));
28    }
```

10.4.3 自定义类型：实现

```
29    assert(!points.contains(Point{n, n}));
30  }
31
32  int main(int argc, char* argv[]) {
33    // hash: 使用散列表
34    // bst: 使用二叉搜索树（平衡二叉树）
35    std::string method = argc > 1 ? argv[1] : "hash";
36    int n = argc > 2 ? std::stoi(argv[2]) : 1000000;
37    std::println("Method: {}, n: {}", method, n);  // C++23
38    if (method == "bst") {
39      std::set<Point> points;
40      Bench(points, n);
41      std::println("Size: {}", points.size());
42    } else if (method == "hash") {
```

10.4.3 自定义类型：实现

```
43      std::unordered_set<Point> points;
44      Bench(points, n);
45      std::println("Size: {}", points.size());
46      std::println("Bucket count: {}", points.bucket_count());
47    } else {
48      std::println("Invalid argument");
49    }
50  }
```

10.4.4 散列函数的应用

除了散列表，散列函数在计算机科学中还有很多应用场景：

- 数据校验
  - MD 系列，如 MD5（128 位）：
    d41d8cd98f00b204e9800998ecf8427e
  - SHA 系列，如 SHA-1（160 位）：
    da39a3ee5e6b4b0d3255bfef95601890afd80709
- 密码存储：不存储密码原文，而存储密码的散列值
- 不可变对象的标识：版本控制系统、区块链

10.5 小结

10.5.1 本讲小结

- 散列表的核心思想：把关键字和存储位置建立映射
- 散列表的主要技术问题：散列函数、解决冲突

数据结构	需求	元素顺序	查找	插入	删除
二叉查找树	大小比较	有序	$O(\log n)$	$O(\log n)$	$O(\log n)$
散列表	散列函数 相等比较	无序	$O(1)$	$O(1)$	$O(1)$

以上时间复杂度均为平均情况，对于自平衡二叉查找树也是最坏情况。