哈希表

哈希表是一种支持快速插入、查找和删除的非线性数据结构。其用一种叫做“哈希”的技术达到这个目的。

哈希

哈希（Hash），又称为**散列**，是一种将任意长度的输入数据通过特定的算法转换为固定长度输出的过程。这个输出通常是一个整数，被称为**哈希值**或**哈希码**。哈希的核心是使用**哈希函数（Hash Function）**，它接受输入数据（称为“键”），并计算出对应的哈希值。

哈希的核心概念

1. **哈希函数**：一种算法或数学函数，用于将输入数据映射到固定范围的哈希值。理想的哈希函数应具备以下特性：
    - **确定性**：相同的输入总是产生相同的输出。
    - **高效性**：计算哈希值的速度快，适合高性能需求。
    - **均匀性**：能够将输入数据均匀地映射到哈希值空间，减少冲突的发生。
    - **不可逆性（在密码学中）**：无法通过哈希值逆推出原始输入。
2. **哈希值**：由哈希函数生成的固定长度的数据，通常用作数据的唯一标识符，便于快速查找和比较。
3. **冲突（碰撞）**：当两个不同的输入数据经过哈希函数处理后得到相同的哈希值时，就发生了冲突。由于哈希函数的值域远远小于定义域，所以哈希冲突是无法完全避免的。处理冲突是哈希技术中的一个重要问题。

哈希的主要用途

哈希在算法中的最重要的用途就是实现无重复元素的“集合”结构，即上面提到的“哈希表”。使用哈希函数将键映射到数组中的索引位置，实现高效的插入、删除和查找操作，平均时间复杂度为$O(1)$。hash set只是实现集合结构，hash map则是实现一种键值对（key value pair）结构，key不允许重复。

当然哈希还有别的用途，比如怎样高效的确定下载的文件和源文件相同呢，我们可以求一下下载了的文件的哈希值，和源文件的哈希值进行比对，即“校验和”。

在算法中，一般不用很复杂的哈希函数，而用比较简单的，例如取模，等等。

哈希表的结构

下面先介绍Hash Set的实现。

哈希表按照处理哈希冲突的方式不同，分为开放地址法（open addressing）和链地址法（separate chaining）两种类型。注意，一般而言哈希表是不允许存重复元素的。我们之后默认如此。

开放地址法的哈希表被称为flat hash set，absl库里有absl::flat_hash_set和absl::flat_hash_map数据结构。而C++里的unordered_set和unordered_map都是采用的链地址法。C++标准库里暂时没有开放地址法的哈希表。

哈希表的插入原理

首先，哈希表底层都会有一个数组，这个数组每一个位置是一个“槽”，每个槽对应的哈希值是不同的。当插入元素的时候，首先会计算元素的哈希值，然后通过哈希值寻找槽的位置。接着，看槽上是否已经存在元素，如果不存在，则直接插入；如果存在，就发生了哈希冲突。例如我们考虑数组是$a=[_,_,_,_]$，简单起见，假设数组只有4个槽，而且我们插入的元素是非负int类型，哈希函数直接取$h(x)=x \mod 4$。如果我们先插入1，那么哈希表变为$a=[_,1,_,_]$。如果再插入5，由于$h(5)=h(1)=1$，这样就发生了哈希冲突。

1. 开放地址法：在发生冲突时，按照一定的探测序列寻找下一个空槽位。那么由于$a[1]$已经有元素了，所以就直接向后找下一个空的位置，由于$a[2]$是空的，那么直接插入，变为$[_,1,5,_]$。当然，“向后找下一个空的位置”可以有多种方法，这里采取的是直接向后一个个找，此外还存在平方探测法等其它方法；
2. 链地址法：每个槽位存储一个链表，所有映射到同一槽位的元素都链接在一起。此时哈希表变为$[_,1\to 5,_,_]$，即$a[1]$的地方变为了一个链表，将两个元素都存在那个槽里。注意，这里我们是将5插到了链表末尾。因为哈希表不允许重复，所以在找到槽之后，还需遍历该槽的链表，确定该元素不存在了，才能执行插入操作。但是，具体插入在链表头还是链表尾，当然是都可以的，没有区别。

这两个方法没有绝对的高下之分，但是在不同的场景之下，我们还是需要有所取舍。具体后面会提到。

一般而言，数组的长度要取一个特别大的质数，并且要离2的幂次比较远；而哈希函数直接取模即可。这样取，可以让哈希冲突的概率降低。

哈希表的查询

也是先计算元素x的哈希值然后找到槽。

1. 开放地址法：如果槽里没有元素，那说明没找到；如果槽里有了元素，并且和x相等，那么就找到了；否则继续向后探测，直到找到了x或者找不到为止。
2. 链地址法：如果槽里没有元素，也说明没找到；否则沿着该槽的链表寻找，直到找到了x或者找不到为止。

如果我们只需要不支持删除操作的哈希表，上面的实现就已经足够了。我们还需要考虑的一点是，如果哈希冲突太多，上面的插入和查询操作都会变得特别慢。我们就需要“重哈希”操作。

重哈希简单说起来，就是另外开一个更加长的数组，当然哈希函数也要跟着改变，然后将原来的元素一个个重新插到新的表里。

那我们怎么界定哈希冲突严重不严重呢？对于开放地址法来说，可以直接看有元素的槽的数量和整个数组长度的比值；对于链地址法来说，可以看链表的长度是否过长。

例题：https://blog.csdn.net/qq_46105170/article/details/113809784。这道例题里，插入操作并没有考虑重复元素的问题（由于没有删除操作，这样做是不会出错的），也没考虑重哈希的问题。一般而言算法题里不需要搞得太复杂。

哈希表的删除

哈希表的删除操作，在链地址法里是相对简单的。只需要找到槽之后，沿着链表继续向下找，直到找到元素然后删除即可。但对于开放地址法，就变得非常复杂。

对于开放地址法，我们的处理方式是这样的：对于每个槽，我们规定三个标记：

```
enum SlotStatus {
  Empty, Occupied, Deleted
};
```

Empty：该槽没有被用过；

Occupied：该槽正在被有效元素使用；

Deleted：该槽曾经被有效元素占用，但后来这个元素被删了。

删除某个元素x的时候，先找到这个元素对应的槽，沿着这个槽进行探测，如果找不到x，那当然什么也不用做；如果找到了，直接将这个槽标记为Deleted就行，不需要进行多余的操作。注意，探测的时候，只要没遇到Empty，或者是Occupied并且等于x的槽，都需要继续进行探测，否则会出错。

查找x操作也要进行相应的修改：首先还是找到它对应的槽，沿着槽进行探测，如果遇到了Occupied的槽并且元素恰好等于x，那当然就意味着找到了；如果遇到了Empty的槽，说明没有找到；但是，遇到了Occupied并且元素不等于x，或者遇到Deleted的时候，必须继续探测，直到发生上面两种情况才能停止。只有这样才能保证不出错。

插入x的操作当然也要进行相应的修改：首先还是找到它对应的槽，沿着槽进行探测。为了保证哈希表不存重复元素，要仔细考虑探测的逻辑：

1. 如果遇到了Occupied并且等于x的槽，说明插入了一个重复元素，更新或者放弃插入（这取决于具体需求）；
2. 如果遇到了Occupied且不等于x的槽，或者遇到了Deleted槽，都必须继续探测，直到遇到Occupied且等于x的槽，或者Empty槽为止（当然，如果槽的数量太少，导致探测回了原位置，那肯定是不行的，需要通过重哈希避免）；
3. 如果遇到了Empty槽，要看之前是否遇到了Deleted槽，如果有，则将x插到第一次的Deleted槽那里，否则插到Empty槽里。

读者可以自行考虑探测操作如此复杂的原因，构造各种情形予以说明。

当Deleted的槽数量特别多的时候，查询会变得特别慢。这时候就需要进行重哈希，以保证效率不降低。并且，当元素的数量特别多的时候，为了保证插入操作不失效，我们也需要提早做重哈希操作。

Hash Map

Hash Map的实现，只需利用Hash Set直接存键值对即可，查询、插入和删除都直接针对key进行操作。Hash Map还有一个更新操作，即将某个key对应的value进行修改，这可以通过查询操作实现，非常简单。

开放地址法和链地址法的比较

当元素的个数不是太多的时候（比如只有几千个），大多数情况下，都要使用开放地址法的哈希表（又叫做“平坦哈希表”）。开放地址法的优势非常明显：

1. 开放地址法的元素存储是inline的，不需要链表节点，速度极快；
2. 元素连续存储，缓存友好；
3. 元素存储紧凑，节省内存，没有指针的额外开销；
4. 元素个数少的话，重哈希情形少，有利于性能。

但是，如果需要所谓的“指针稳定性”（pointer stability）的话，就不能用平坦哈希表了，必须用链地址法哈希表。

指针稳定性

一个数据结构容器，如果存储元素之后，元素的内存地址随着容器继续进行各种操作不会发生改变（当然删除操作除外，元素都被删了，就更没必要讨论了），就称这个容器是指针稳定的。如果容器指针稳定，那么就能保证指向其里面元素的指针和引用不失效，在有的情形下是必要的。

根据这个概念，我们可以知道：

1. `std::vector`是不指针稳定的，因为其有扩容操作，扩容会将元素移动到另一片内存地址；同理，`absl::flat_hash_set`和`absl::flat_hash_map`也都不是指针稳定的，因为重哈希的时候会另外申请一片内存并且移动元素，从而改变元素地址；
2. `std::list`是指针稳定的，同理，`std::deque`也是，从而`std::queue`和`std::stack`也都是。由于`unordered_set`和`unordered_map`都是采用链地址法（槽上事实上存的不是元素，而是链表的头元素的地址的指针），即使发生了重哈希，也不会改变元素的内存地址，所以它们也是指针稳定的。

所以，如果需要指针稳定性，就必须使用链地址法哈希表。此外，如果元素数量太多，导致重哈希次数太多的话，链地址法哈希表要比平坦哈希表性能更好一些。

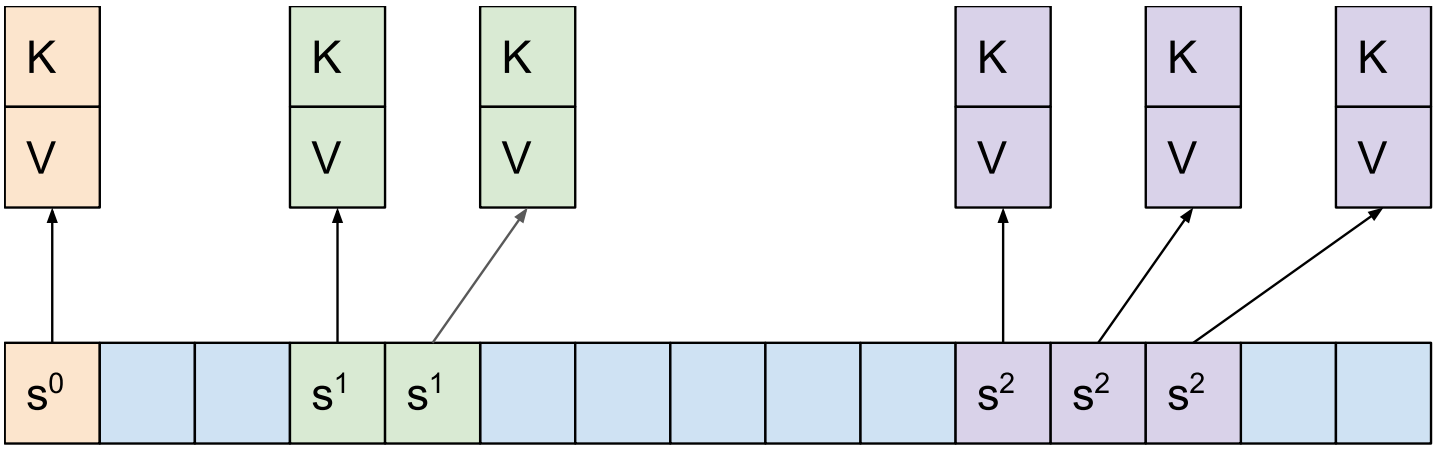
补充

传统的`absl::flat_hash_set`的存储，元素是`inline`存的（即直接存在了数组上），导致了其不指针稳定。其实可以在槽里只存元素的指针，而将元素存在别处，这样就可以保证指针稳定性了（但这样做要牺牲一部分性能）。所以说平坦哈希表并不是一定保证不了指针稳定性，而是要做适当的修改。

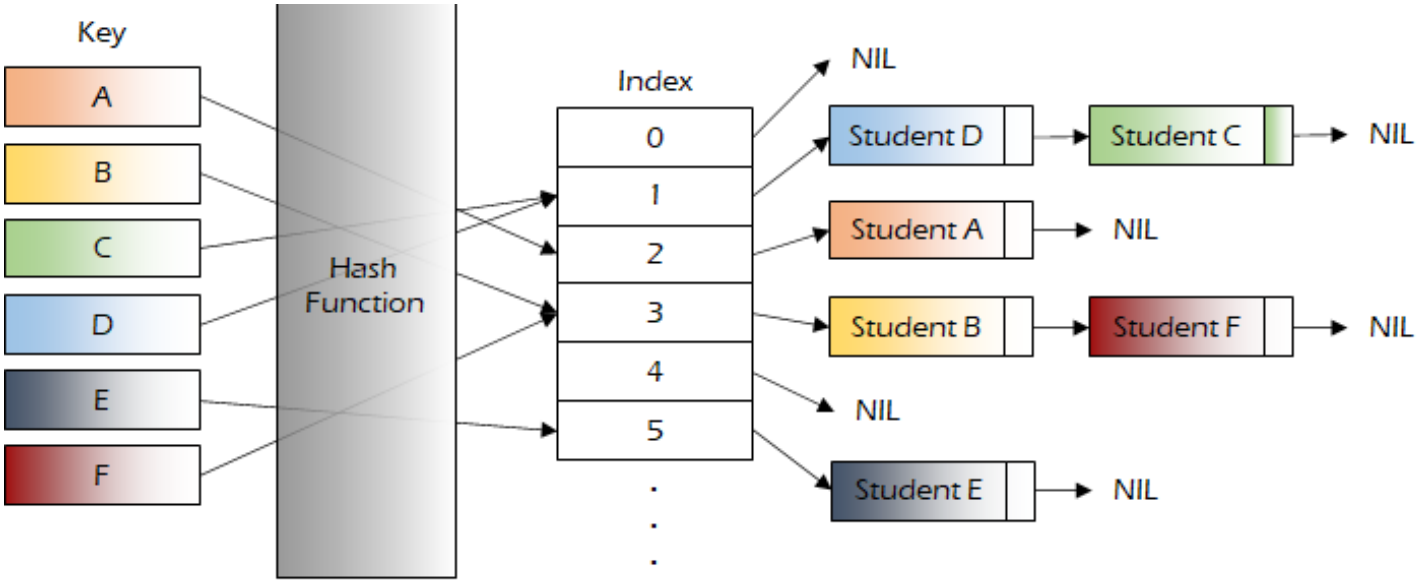
示意图如下：

`absl::flat_hash_map`：

`absl::node_hash_map`：



`std::unordered_map`：



总结说来，在大部分场景下：

1. 如果不需要指针稳定性，就用`absl::flat_hash_map`；
2. 如果需要key的指针稳定性，但不需要value的，就用`absl::flat_hash_map<Key, std::unique_ptr<Value>>`，`unique_ptr`的`std::move`操作并不会改变value的内存地址；
3. 如果key和value的稳定性都需要，就用`absl::node_hash_map`。

算法题里`ABSL`库用不了，就用`std`的。

作业

实现只支持插入和查询的哈希表，开放地址和链地址法都要实现。

实现支撑插入、查询和删除的哈希表，同样，开放地址和链地址法都要实现。

理解指针稳定性的概念，并体会哈希表的实现与其的联系。