快速哈希表的新实现

用户 14869360

评价我: 5.00/5 (2 票)

2020 年 8 月 3 日 麻省理工学院

有序数组和哈希表的合并

一种看起来像有序数组并且行为像哈希表的数据结构。任何值都可以通过其索引或名称访问。它保留元素的顺序并使用两个内存块。因此，它具有初始大小，并且冲突不会导致新的分配。调整大小时，它会删除已删除的项目并重置其哈希基础。

下载源 - 1.2 MB

介绍

哈希表是能够使用字符串键定位值的数据结构。有些哈希表插入速度很快，有些则查找值很快。一些实现会遭受缓慢删除、内存碎片或随着项目增多而性能下降的问题。大多数哈希表不按顺序存储项目，或使用索引检索值，并且必须消耗更多内存以进行快速搜索或跟踪顺序（如果它们支持的话）。但是，这种实现可能会以更少的交易提供更高的速度。

执行

对于使用数字检索值的数据结构，它需要是一个线性数组，当它的内存块达到极限时，它会扩展到一个新的堆，并将其所有项从旧内存移到新内存中。因此，它必须是一个一维数组。虽然它不仅仅是一个数组或一个哈希表，但它是两者兼而有之。

MC++ 复制代码

```
template <typename Value_>
class HArray {
    size_t  size_{0};
    size_t  capacity_{0};
    Value_ *storage_{nullptr};
};
```

该变量`size_`保存项目数，`capacity_`用于表示堆/内存块的大小，并`storage_`指向第一个值的位置。尽管如此，项目必须存储的不仅仅是值；他们需要变量`Key`及其哈希`HArray`作为哈希表。

MC++ 复制代码

```
template <typename Key_, typename Value_>
struct HAItem {
    size_t   Hash;
    Key_     Key;
    Value_   Value;
};
```

```
template <typename Key_, typename Value_>
class HArray {
    size_t                  size_{0};
    size_t                  capacity_{0};
    HAItem<Key_, Value_>   *storage_{nullptr};
};
```

线性数组插入如下所示：

MC++ 复制代码

```
private:
void insert_(Key_ key, Value_ value, size_t hash) {
    if (size_ == capacity_) {
        // expand
    }

    HAItem<Key_, Value_> &item = storage_[size_];

    item.Hash  = hash;
    item.Key   = key;
    item.Value = value;

    ++size_;
}
```

但是，要使哈希表正常工作，它需要一种搜索技术并处理冲突。解决这个问题的一种方法是通过Next在结构中添加一个额外的变量来添加一个链表，以指向下一个项目。因此，新结构如下所示：

MC++ 复制代码

```
template <typename Key_, typename Value_>
struct HAItem {
    size_t     Hash;
    HAItem    *Next;
    Key_       Key;
    Value_     Value;
};
```

搜索技术将给定的密钥与存储的密钥进行比较。它通过将其散列值除以（容量 - 1）和余数 - 该除法点指向一个索引，即分配的堆中的一个位置来定位它。如果compare函数寻求两个键不同，则它检查指针Next以查看其项目是否持有相同的键，并继续查找直到找到它或命中null指针。但是，可能会出现无限循环。

假设capacity_是21，并且有两个项目要插入：item1具有等于 的散列值41，并将item2其散列设置为60。的位置item1是41 % (21 - 1) = 41 % 20 = 1。该算法将查找索引1并将最后一个/null Next指针设置为指向item1。对于item2，60 % 20 = 0,storage_[0].Next被设置为指向item2; 假设Next是null。现在，如果一个新项的哈希值为20, 40, 21, 41…，它将指向item1或item2，当它item1指向或时，它将看到指向item2和item2.Next指向item1，并且当它试图找到链表的末尾时，它最终会陷入无限循环。

这可以通过存储起始地址并检查每个Next变量来解决，如果它们匹配，则中断循环，然后在最后一个Next和后面的一个之间插入新项目（可能是null）。但是，还有一种更快的方法，那就是添加另一个名为 的指针Anchor。因此，最终的结构HAItem如下：

MC++ 复制代码

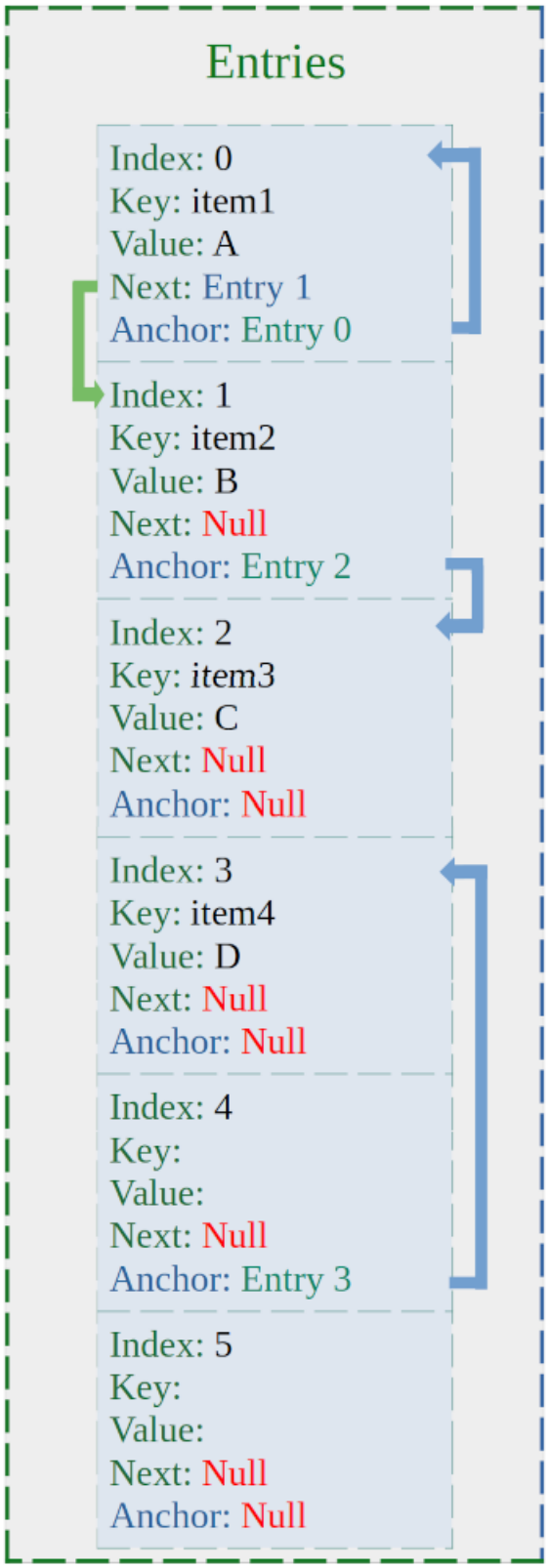
```
template <typename Key_, typename Value_>
struct HAItem {
    HAItem    *Anchor{nullptr};
    size_t     Hash{0};
    HAItem    *Next{nullptr};
    Key_       Key{};
    Value_     Value{};
};
```

现在Next仅用于碰撞，并Anchor指向项目的位置。这样，Next将指向前向元素 (1, 2, 3, 4, 5)，并CPU prefetching可能带来下一个项目，而该search函数正在将给定的键与当前项目的Key. 但是，Anchor可以指向该堆中的任何人，如果是null，则该项目不存在。

为了理解这个概念，假设我们有一个容量为 6 个元素的数组，我们需要插入以下项目：

钥匙	价值	哈希
item1	A	10
item2	B	15
item3	C	21
item4	D	24

哈希表的基是(capacity - 1)，即5。item1插入位置0- 因为 10 % 5 是 0，因此，storage_[0].Anchor= 索引 0 的指针。item2插入索引 1，但其哈希结果指向 0 (15 % 5 = 0)，搜索函数将查找索引 0 和finds item1，然后它检查其Next指针以找到具有null值的最终指针并将其设置为指向item2。item3放置在索引 3 中，其散列指向索引 1。即使索引 1 中有一个项目，它Anchor也是null，并且函数将设置storage_[1].Anchor为指向item3。item4将得到索引 3，其散列结果指向索引 4，因此，storage_[4].Anchor = &item3。数据将如下所示：



不需要重新散列、二叉树、分离的哈希表或额外的指针。此外，拥有变量`Anchor`意味着`search`函数始终可以返回对指针的引用，并且它可以引用`Anchor`or `Next`，并且该指针可以指向一个项目 or `null`。该`search`函数可以定义为：

MC++ 缩小▲ 复制代码

```
public:
Value_ *Find(const Key_ &key) {
    HAItem_T *item = *(find_(key, hash_fun_(key)));

    if (item != nullptr) {
        return &(item->Value);
    }

    return nullptr;
}
```

```
private:
HAItem_T **find_(const Key_ &key, size_t hash) const {
    if (capacity_ == 0) {
        // To prevent dividing by zero, allocate a small heap.
        capacity_ = 2;
        storage_  = new HAItem_T[capacity_];
    }

    HAItem_T **item = &(storage_[(hash % capacity_)].Anchor);

    while (((*item) != nullptr) && ((*item)->Key != key)) {
        item = &((*item)->Next);
    }

    return item;
}
```

哈希表检查每个插入的项目以查看它是否存在，并使用`find_()`返回对该项目的引用。当它存在时，它用给定的值替换它的值，如果不存在，则返回的引用指向新项应放置在哈希表中的位置；两只鸟，一块石头：

MC++ 复制代码

```
public:
Value_ &operator[](Key_ &&key) {
    const size_t hash = hash_fun_(key);
    HAItem_T **item   = find_(key, hash);

    if ((*item) == nullptr) {
        insert_(static_cast<Key_ &&>(key), hash, item);
    }

    return (*item)->Value;
}

private:
void insert_(Key_ &&key, size_t hash, HAItem_T **position) {
    if (index_ == capacity_) {
        Resize(capacity_ * 2);
        position = find_(key, hash);
    }

    (*position) = (storage_ + index_);
    (*position)->Hash    = hash;
    (*position)->Key     = static_cast<Key_ &&>(key);

    ++index_;
    ++size_;
}
```

唯一一次`find_()`调用两次，是当内存块的大小发生变化时；因为基数取决于 的值`capacity_`。

MC++ 缩小▲ 复制代码

```
public:
void Resize(size_t new_size) {
    capacity_ = ((new_size > 1) ? new_size : 2);

    if (index_ > new_size) {
        index_ = new_size; // Shrink
    }

    const size_t base = (capacity_ - 1);
    size_t       n    = 0;
    size_t       m    = 0;

    HAItem_T *src = storage_;
    storage_      = new HAItem_T[capacity_];
```

```
    while (n != index_) {
        HAItem_T &src_item = src[n];
        ++n;

        if (src_item.Hash != 0) {
            HAItem_T *des_item = (storage_ + m);
            des_item->Hash    = src_item.Hash;
            des_item->Key     = static_cast<Key_ &&>(src_item.Key);
            des_item->Value   = static_cast<Value_ &&>(src_item.Value);

            HAItem_T **position = &(storage_[(des_item->Hash % base)].Anchor);

            while (*position != nullptr) {
                position = &((*position)->Next);
            }

            *position = des_item;

            ++m;
        }
    }

    index_ = size_ = m;
    delete[] src;
}
```

由于Resize()不需要检查item是否存在，rehashing的机制很简单（4行代码）：

MC++ 复制代码

```
HAItem_T **position = &(storage_[(des_item->Hash % base)].Anchor);

while (*position != nullptr) {
    position = &((*position)->Next);
}

*position = des_item;
```

该行if (src_item.Hash != 0)删除任何已删除的项目，删除与插入相同，但它会跟踪要删除的项目之前的项目；其设置Next到Next下一个项目。此外，它会清除值和关键，但保持Next和Anchor完整; 因为他们持有散列信息。

C++ 缩小▲ 复制代码

```
public:
inline void Delete(const Key_ &key) {
    delete_(key);
}

public:
void DeleteIndex(size_t index) {
    if (index < index_) {
        const HAItem_T &item = storage_[index];

        if (item.Hash != 0) {
            delete_(item.Key, item.Hash);
        }
    }
}

private:
void delete_(const Key_ &key, size_t hash) {
    if (capacity_ != 0) {
        HAItem_T **item = &(storage_[(hash % capacity_)].Anchor);
        HAItem_T **before = item;

        while (((*item) != nullptr) && ((*item)->Key != key)) {
            before = item; // Store the previous item
```

```
            item   = &((*item)->Next);
        }

        if ((*item) != nullptr) {
            HAItem_T *c_item = *item; // Current item

            if ((*before) >= (*item)) {
                /*
                * If "before" inserted after "item"
                * (e.g., deleting items from 0 to n).
                */
                (*before) = c_item->Next;
            } else {
                /*
                * If "before" inserted before "item"
                * (e.g., deleting items from n to 0).
                */
                (*item) = c_item->Next;
            }

            c_item->Hash  = 0;
            c_item->Next  = nullptr;
            c_item->Key   = Key_();
            c_item->Value = Value_();
            --size_;
        }
    }
}
```

由于HArray它的核心是一个数组，因此可以添加一个具有初始大小的构造函数：

MC++ 复制代码

```
explicit HArray(size_t size) : capacity_(size) {
    if (size != 0) {
        storage_ = new HAItem_T[capacity_];
    }
}
```

尽管如此，也可以通过索引访问任何键或值：

C++ 缩小▲ 复制代码

```
Value_ *GetValue(size_t index) {
    if (index < index_) {
        HAItem_T &item = storage_[index];

        if (item.Hash != 0) {
            return &(item.Value);
        }
    }

    return nullptr;
}

const Key_ *GetKey(size_t index) const {
    if (index < index_) {
        const HAItem_T &item = storage_[index];

        if (item.Hash != 0) {
            return &(item.Key);
        }
    }

    return nullptr;
}

const HAItem_T *GetItem(size_t index) const {
```

```
    if (index < index_) {
        const HAItem_T &item = storage_[index];

        if (item.Hash != 0) {
            return &(item);
        }
    }

    return nullptr;
}
```

改进的实施

虽然之前的实现有效并且其性能还不错，但仍有改进的空间，从确定最弱的链开始。

看代码，最常用的变量是`Anchor`，即使没有关联的item也需要初始化，连同整个结构`HAItem`。因此，需要做两件事：一是移动`Anchor`到其分离的数据结构，二是分配未初始化的内存块；以避免将变量设置两次，或者除非需要，否则不必初始化它的任何部分。对于第二，`std::allocator`是合适的。因此，新的结构如下：

MC++ 复制代码

```
template <typename Key_, typename Value_>
struct HAItem {
    size_t  Hash;
    HAItem *Next;
    Key_    Key;
    Value_  Value;
};

struct HAItem_P {
    HAItem_T *Anchor{nullptr};
};

template <typename Key_, typename Value_, typename Hash_ = std::hash<Key_>,
          typename Allocator_ = std::allocator<HAItem<Key_, Value_>>>
    size_t       size_{0};
    size_t       index_{0};
    size_t       capacity_{0};
    HAItem_P   * hash_table_{nullptr};
    HAItem_T   * storage_{nullptr};
    Allocator_   allocator_{};
};
```

其余部分几乎相同，但由于Anchor在其单独的堆中，因此与以前相比，增加其大小不会有太大影响，并且使其更大可提供更快的查找并减少对 的调用Next；因为更少的碰撞：

MC++ 缩小▲ 复制代码

```
private:
void insert_(Key_ &&key, size_t hash, HAItem_T **&position) {
    if (index_ == capacity_) {
        Resize(capacity_ * 2);
        position = find_(key, hash);
    }

    (*position) = (storage_ + index_);
    new (*position) HAItem_T{hash, nullptr, static_cast<Key_ &&>(key), Value_{}};
```

```
    ++index_;
    ++size_;
}

private:
HAItem_T **find_(const Key_ &key, size_t hash) {
    if (capacity_ == 0) {
        // To prevent dividing by zero, allocate a small heap.
        capacity_   = 1;
        hash_table_ = new HAItem_P[(capacity_ * 2)];
        storage_    = allocator_.allocate(capacity_);
    }

    HAItem_T **item = &(hash_table_[(hash % (capacity_ * 2))].Anchor);

    while (((*item) != nullptr) && ((*item)->Key != key)) {
        item = &((*item)->Next);
    }

    return item;
}

private:
void delete_(const Key_ &key, size_t hash) {
    if (capacity_ != 0) {
        HAItem_T **item = &(hash_table_[(hash % (capacity_ * 2))].Anchor);

        ...
    }
}
```

调整大小函数可以分为两个：一个处理项目，另一个生成散列：

C++ 缩小▲ 复制代码

```
public:
void Resize(size_t new_size) {
    const size_t old_capacity = capacity_;

    capacity_ = ((new_size > 1) ? new_size : 2);

    if (index_ > new_size) {
        destruct_range_((storage_ + new_size), (storage_ + index_));
        index_ = new_size; // Shrink
    }

    size_t n = 0;
    size_t m = 0;

    delete[] hash_table_;
    HAItem_T *src = storage_;
    storage_      = allocator_.allocate(capacity_);

    while (n != index_) {
        HAItem_T &src_item = src[n];
        ++n;

        if (src_item.Hash != 0) {
            HAItem_T *des_item = (storage_ + m);

            new (des_item)
                HAItem_T{src_item.Hash, nullptr,
                        static_cast<Key_ &&>(src_item.Key),
                        static_cast<Value_ &&>(src_item.Value)};

            ++m;
        }
    }
```

```
    index_ = size_ = m;
    allocator_.deallocate(src, old_capacity);

    generate_hash_();
}

private:
void generate_hash_() {
    // hash_table_ should be deleted before this.

    const size_t base = (capacity_ * 2);

    if (base != 0) {
        hash_table_ = new HAItem_P[base];

        for (size_t i = 0; i < size_; i++) {
            HAItem_T *item = (storage_ + i);

            HAItem_T **position = &(hash_table_[(item->Hash % base)].Anchor);

            while (*position != nullptr) {
                position = &((*position)->Next);
            }

            *position = item;
        }
    }
}
```

改进的查找

现在剩下的问题是：它可以更快吗？答案是：是的，只需一个简单的更改即可；在函数%中用按位 AND替换除法为. 这是为了避免获取哈希表的大小或零。为了更深入地理解这一点，假设哈希表的大小为 4：查找将是 0 & 4 = 0、1 & 4 = 0、2 & 4 = 0、3 & 4 = 0，但 4 & 4 = 4，这不是一个有效的索引。此外，更多的零意味着更多的碰撞。然而，1 & 3 = 1, 2 & 3 = 2, 3 & 3 = 3, 4 & 3 = 0。现在所有这些都可以放置并定位在哈希表中。`&find_()(hash & (base - 1))-1`

MC++ 复制代码

```
private:
HAItem_T **find_(const Key_ &key, size_t hash) {
    if (capacity_ == 0) {
        base_       = 1;
        capacity_   = 1;
        hash_table_ = new HAItem_P[base_ + 1];
        storage_    = allocator_.allocate(capacity_);
    }

    HAItem_T **item = &(hash_table_[(hash & base_)].Anchor);

    while (((*item) != nullptr) && ((*item)->Key != key)) {
        item = &((*item)->Next);
    }

    return item;
}
```

当大小可以表示为 2 的 N 次方时，这种方法很有效`2^n`：2、4、8、16、32、64、128……因为计算机只使用二进制代码。以下表为例：

N & (7-1)	结果
0 & 6	0
1 & 6	0

N & (7-1)	结果
2 & 6	2
3 & 6	2
4 & 6	4
5 & 6	4
6 & 6	6

0 与 1、2 与 3、4 与 5 发生碰撞。现在，如果大小为 8，即 2^3：

N & (8-1)	结果
0 & 7	0
1 & 7	1
2 & 7	2
3 & 7	3
4 & 7	4
5 & 7	5
6 & 7	6

没有碰撞！因此，它是 O(1)，但并没有那么简单，因为条目有键，而值是键的散列。尽管如此，它产生的碰撞更少。

既然HArry有2个增长因子：1,2,4,8,... 那么使用bitwise AND over是没有问题的，让CPU花更多的时间计算，什么时候可以在一个操作中完成，嗯，是的，并且不。是的，如果它在没有初始大小的情况下使用，并且没有，因为它有Compress(),Resize()和SetCapacity()。如果它与另一个数组合并，则它们的元素总和成为新的大小，如果它不小于容量。为此，HArry需要一个额外的方法来对齐base_：

C++ 复制代码

```
private:
void set_base_(size_t n_size) noexcept {
    base_ = 1U;
    base_ <<= CLZL(static_cast<unsigned long>(n_size));

    if (base_ < n_size) {
        base_ <<= 1U;
    }

    --base_;
}
```

随着base_是该类别的新变量：

MC++ 复制代码

```
...
class HArray {
    size_t      base_{0};
    size_t      index_{0};
    size_t      size_{0};
    size_t      capacity_{0};
    HAItem_P *  hash_table_{nullptr};
    HAItem_T *  storage_{nullptr};
    Allocator_  allocator_{};
};
```

对于其余的：

C++ 缩小▲ 复制代码

```
public:
void Resize(size_t new_size) {
    if (index_ > new_size) {
        destruct_range_((storage_ + new_size), (storage_ + index_));
        index_ = new_size; // Shrink
    }

    set_base_(new_size);
    resize_(new_size);
}

private:
void resize_(size_t new_size) {
    HAItem_T *src = storage_;
    storage_      = allocator_.allocate(new_size);
    size_t n      = 0;
    size_t m      = 0;

    while (n != index_) {
        HAItem_T &src_item = src[n];
        ++n;

        if (src_item.Hash != 0) {
            HAItem_T *des_item = (storage_ + m);
            ++m;

            new (des_item)
                HAItem_T{src_item.Hash, nullptr,
                            static_cast<Key_ &&>(src_item.Key),
                            static_cast<Value_ &&>(
                                src_item.Value)}; // Construct *des_item
        }
    }

    index_ = size_ = m;

    delete[] hash_table_;
    allocator_.deallocate(src, capacity_);
    capacity_ = new_size;
    generate_hash_();
}

private:
void generate_hash_() {
    hash_table_ = new HAItem_P[(base_ + 1)];

    for (size_t i = 0; i < index_; i++) {
        HAItem_T * item     = (storage_ + i);
        HAItem_T **position = &(hash_table_[(item->Hash & base_)].Anchor);

        ...
    }
}

private:
void insert_(Key_ &&key, size_t hash, HAItem_T **&position) {
    if (size_ == capacity_) {
        ++base_;
        base_ *= 2;
        --base_;

        resize_(capacity_ * 2);
        position = find_(key, hash);
    }
```

```
    ...
}

private:
void delete_(const Key_ &key, size_t hash) {
    if (capacity_ != 0) {
        HAItem_T **item = &(hash_table_[(hash & base_)].Anchor);
        ...
    }
}
```

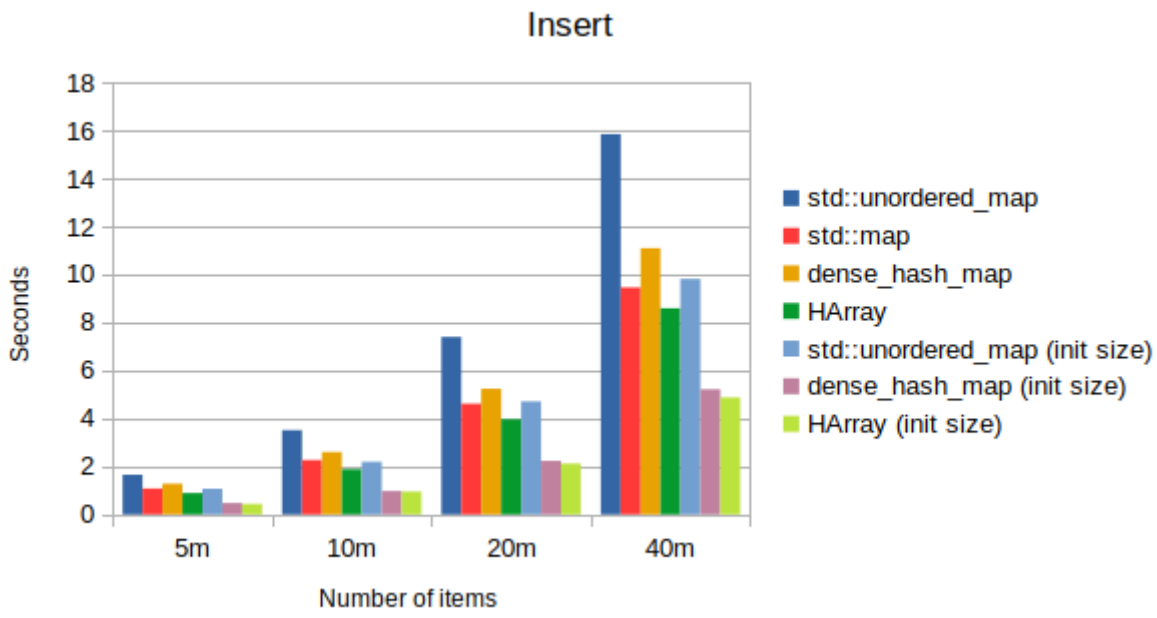
对于具有初始大小的构造函数：

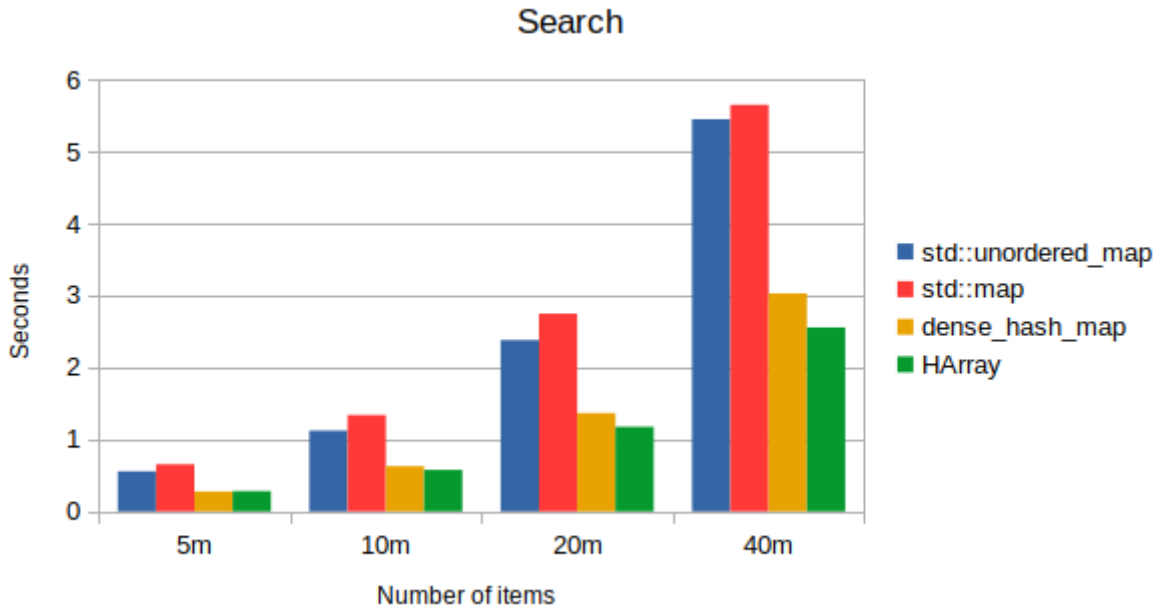
MC++ 复制代码

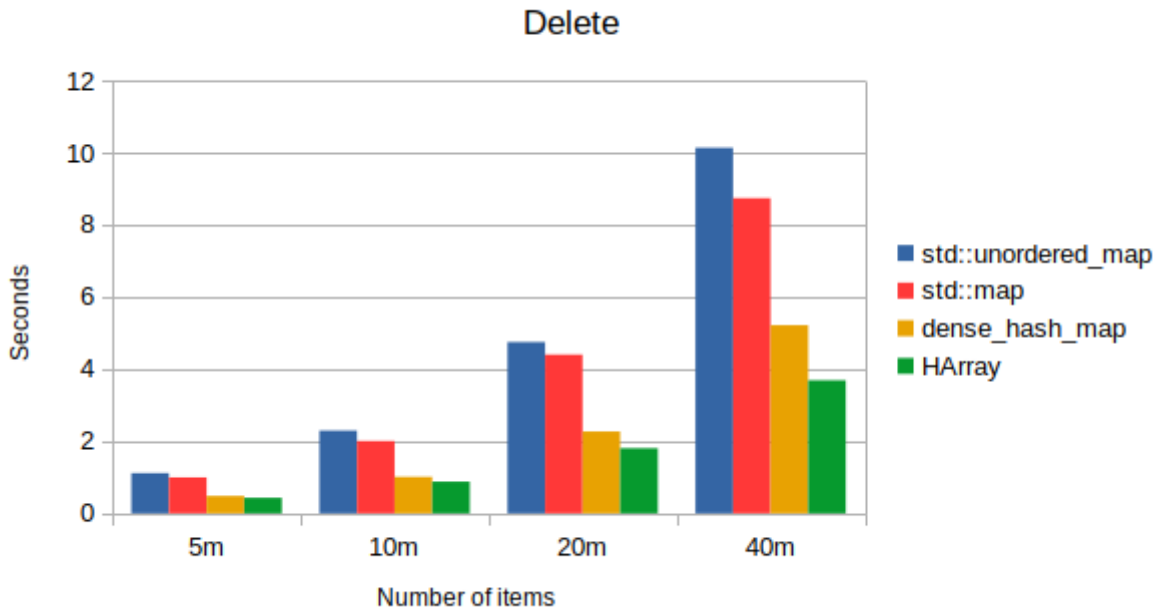
```
explicit HArray(size_t size) : capacity_{size} {
    set_base_(size);
    hash_table_ = new HAItem_P[(base_ + 1)];
    storage_    = allocator_.allocate(capacity_);
}
```

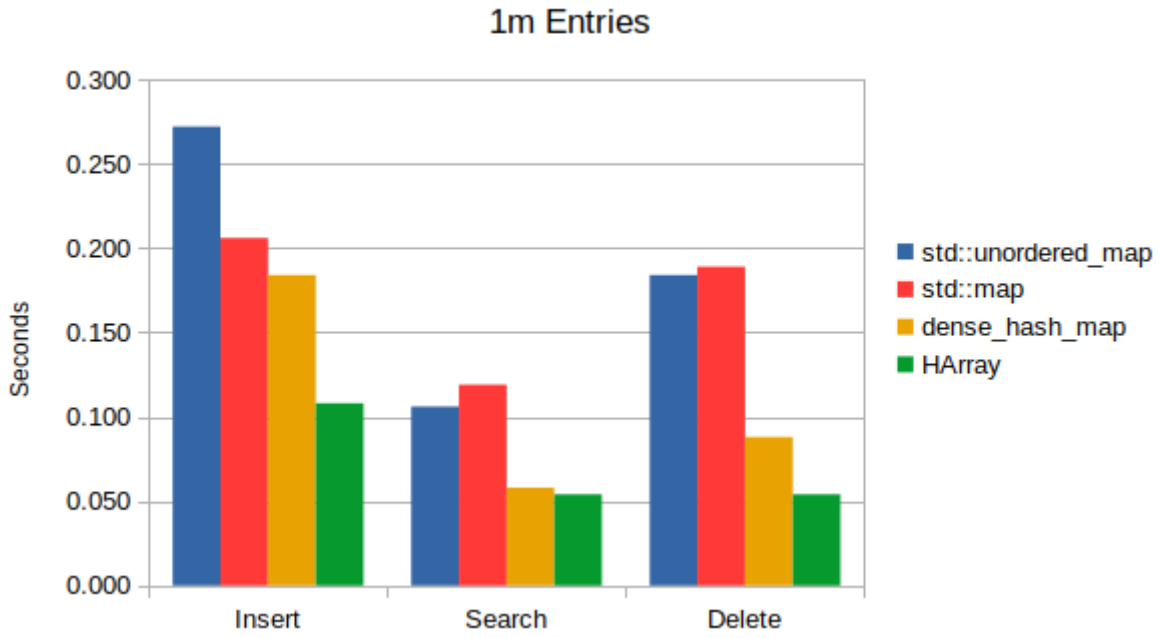
现在，它与最快的竞争。

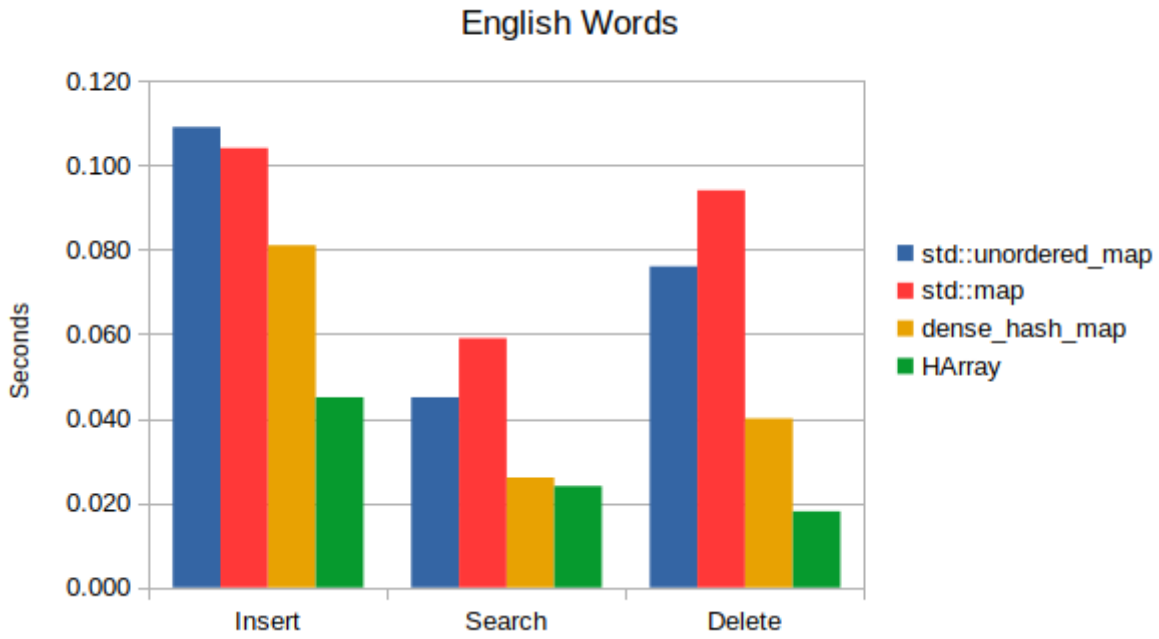
基准

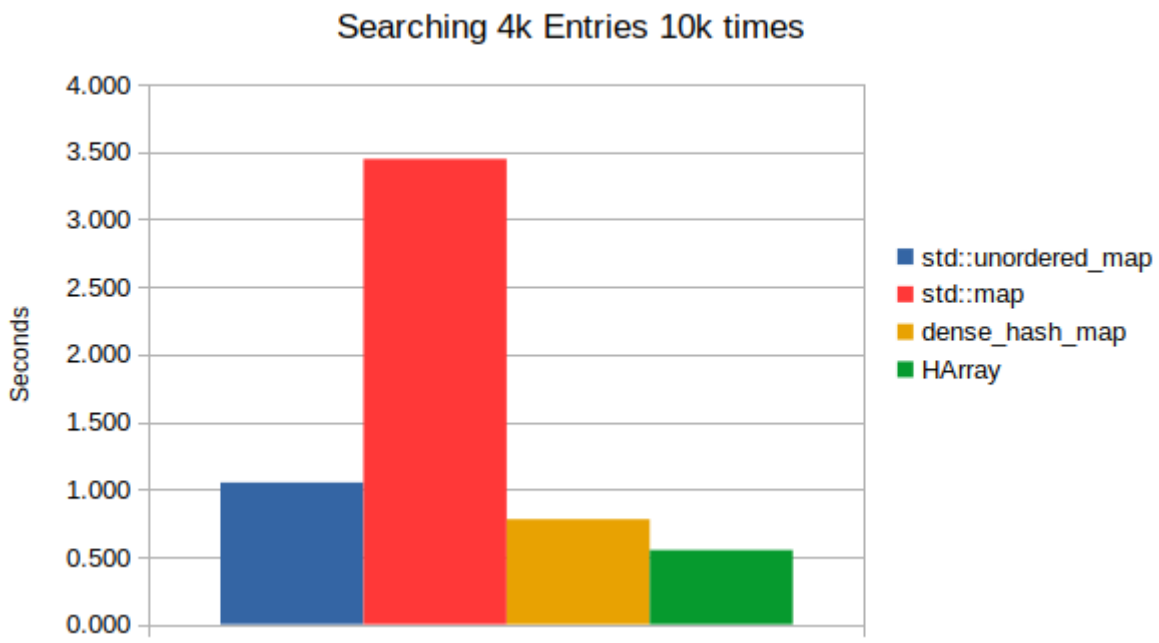



Search
Seconds
0
1
2
3
4
5
6
5m
10m
20m
40m
Number of items
std::unordered_map
std::map
dense_hash_map
HArray


Delete
Seconds
0
2
4
6
8
10
12
5m
10m
20m
40m
Number of items
std::unordered_map
std::map
dense_hash_map
HArray


1m Entries
Seconds
0.300
0.250
0.200
0.150
0.100
0.050
0.000
Insert
Search
Delete
std::unordered_map
std::map
dense_hash_map
HArray


English Words
Seconds
0.120
0.100
0.080
0.060
0.040
0.020
0.000
Insert
Search
Delete
std::unordered_map
std::map
dense_hash_map
HArray



注：英文单词来自：https : //github.com/dwyl/english-words

编译

- 制作
  - Linux

蝙蝠 复制代码

```
mkdir build
cd build
cmake -D BUILD_TESTING=0 ..
cmake --build .
./benchmark
```

- 视窗

蝙蝠 复制代码

```
mkdir build
cd build
cmake -D BUILD_TESTING=0 ..
cmake --build . --config Release
.\Release\benchmark.exe
```

- 海湾合作委员会/铿锵

蝙蝠 复制代码

```
mkdir build
c++ -O3 -std=c++11 -I ./include ./src/benchmark.cpp -o ./build/benchmark
./build/benchmark
```

结论

虽然哈希表的实现有很多，但很少有提供使用索引访问元素的方法，并且其中一些会受到内存碎片的影响。此外，按顺序存储项目可能会导致查找速度变慢。此实现采用一个有序数组并为其添加散列功能，以创建一个新的散列表，该表在约 300 行代码中更快速且更可用。

GitHub 存储库：https : //github.com/HaniAmmar/HArray

例子

C++ 缩小▲ 复制代码

```
#include <iostream>
#include <string>

#include "HArray.hpp"

using Qentem::HArray;

int main() {
    HArray<std::string, size_t> ha;
    const size_t *              val;

    // HArray<std::string, size_t> ha(10); // Initialize with the capacity of 10
    // ha.SetCapacity(10); // Same, but after initialization

    ha["a"]  = 1;
    ha["b"]  = 2;
    ha["c"]  = 30;
    ha["c"]  = 3; // Reset "c" to value of "3";
    ha["dd"] = 4;
    ha["e"]  = 5;
    ha["z"]  = 50;
    ha.Rename("dd", "d"); // Will rename "dd" to "d".

    std::cout << ha["a"] << '\n'; // Output: 1
    std::cout << ha["c"] << '\n'; // Output: 3

    std::cout << "\nSize: " << ha.Size() << "\nValues: "; // Output: 6

    for (size_t i = 0; i < ha.ArraySize(); i++) {
        val = ha.GetValue(i);

        if (val != nullptr) {
            std::cout << (*val) << ' '; // Output: 1 2 3 4 5 50
        }
    }

    std::cout << "\n\n";

    val = ha.Find("z");

    if (val != nullptr) {
        std::cout << "The value of \"z\" is: " << (*val); // Output: 50
    }

    std::cout << "\n\nKeys: ";

    for (size_t i = 0; i < ha.ArraySize(); i++) {
        const std::string *key = ha.GetKey(i);

        if (key != nullptr) {
            std::cout << *key << ' '; // Output: a b c d e z
        }
    }

    ha.Delete("z");
    std::cout << "\n\nSize: " << ha.Size() << '\n';     // Output: 5
    std::cout << "Index: " << ha.ArraySize() << '\n';   // Output: 6
    std::cout << "Capacity: " << ha.Capacity() << '\n'; // Output: 8

    ha.Compress(); // Will remove extra storage and/or remove any deleted items
    std::cout << "\nSize: " << ha.Size() << '\n';       // Output: 5
```

```
std::cout << "Index: " << ha.ArraySize() << '\n';   // Output: 5
std::cout << "Capacity: " << ha.Capacity() << '\n'; // Output: 5

ha.Delete("d");

/* The next line will resize it to 4 if 4 is bigger than the number of
 * items, and will remove any deleted items.
 *
 * If 4 is not bigger, the capacity will be set to 4.
 */
ha.Resize(4);
std::cout << "\nSize: " << ha.Size() << '\n';         // Output: 3
std::cout << "Capacity: " << ha.Capacity() << "\n\n"; // Output: 4

for (size_t i = 0; i < ha.ArraySize(); i++) {
    val = ha.GetValue(i);

    if (val != nullptr) {
        std::cout << (*val) << ' '; // Output: 1 2 3
    }
}

std::cout << "\n\n";

ha["d"] = 4; // Add "d" again

HArray<std::string, size_t> ha2;

ha2["b"] = 10;
ha2["d"] = 30;
ha2["w"] = 220;
ha2["z"] = 440;

HArray<std::string, size_t> ha3 = ha; // Copy ha to ha3

/* Add ha2's items to ha if they do not exist, and/or replace the ones that
 * exist.
 */
ha += ha2;

using my_item_T = Qentem::HAItem<std::string, size_t>;

for (size_t i = 0; i < ha.ArraySize(); i++) {
    const my_item_T *item = ha.GetItem(i);

    if (item != nullptr) {
        std::cout << item->Key << ": " << item->Value << ". ";
        //  Will output: a: 1. b: 10. c: 3. d: 30. w: 220. z: 440.
    }
}

std::cout << "\n\n";

ha2 += ha3; // Add ha3's items to ha2

for (size_t i = 0; i < ha2.ArraySize(); i++) {
    const my_item_T *item = ha2.GetItem(i);

    if (item != nullptr) {
        std::cout << item->Key << ": " << item->Value << ". ";
        //  Will output: b: 2. d: 4. w: 220. z: 440. a: 1. c: 3.
    }
}

std::cout << '\n';

ha.Clear(); // Will remove any existing items and set it to zero.
```

```
    return 0;
}
```

历史

- 2020 年 7 月30日：初始版本
- 3次 2020年8月：新增改善执行
- 2020 年 8 月6日：添加了改进的查找
- 2020 年 8 月10日：添加示例

执照

分享

关于作者

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

我的5票
Matth Moestl 7-Aug-20 5:35

回复：我的5票
User 14869360 7-Aug-20 6:19

std::unordered_map
Sergeant Kolja 2-Aug-20 17:26

回复： std::unordered_map
User 14869360 2-Aug-20 19:53

回复： std::unordered_map
Sergeant Kolja 3-Aug-20 17:13

回复： std::unordered_map

User 14869360 4-Aug-20 0:03

关于删除功能
wmjordan **1-Aug-20 14:52**

回复：关于删除功能
User 14869360 1-Aug-20 22:01

回复：关于删除功能
wmjordan 20-Aug-20 9:51

回复：关于删除功能
User 14869360 22-Aug-20 9:28

消息已关闭
30-Jul-20 16:53

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。