# Hash tables

1. **Background and terminology**
2. **Resolving collisions**
3. **Hash function**
4. **Rehashing**
5. **Runtime efficiency and amortized analysis**

## 1. Background and terminology

At start: we have many pairs: $(k_1, r_1)$, $(k_2, r_2)$, $(k_3, r_3)$, $\ldots$ $(k_n, r_n)$

In each pair: $k_i$ is unique key and $r_i$ is a record

Goal: maintain all records in some data structure that allows efficient **insert**, **delete**, and **search** operations

### Example

A database of cars

$k_i$ = license plate

$r_i$ = (car make, car model, motor type, color, etc..) $\square$

Consider vector (or array):

| $A[1]$ | $A[2]$ | $A[3]$ | $\ldots$ | $A[m]$ |
|---|---|---|---|---|

Given index $i$, we can very quickly

- find the contents of $A[i]$ (search)
- overwrite the contents of $A[i]$ with NIL (delete)
- put a record into $A[i]$ (insert)

A hash table tries to mimic the efficiency of an array, but only for **insert**, **delete**, and **search** operations.

A hash table is of little use when something having to do with **order** is needed:

- find the largest or smallest key
- given a key $k_i$, find the next largest or the next smallest key
- given a key $k_i$, find if there are any keys larger than $k_i$

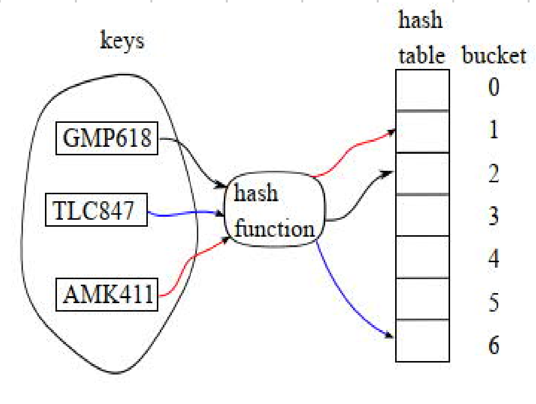
Terminology

- key: a unique key associated with a record
- record: all the data we want to store about one item
- hash table: an array of buckets
- bucket (or slot): one location in the hash table
- hash function (or hash map): a function that computes a location (the bucket) in the hash table given a key
- collision: when two or more records are assigned to the same bucket

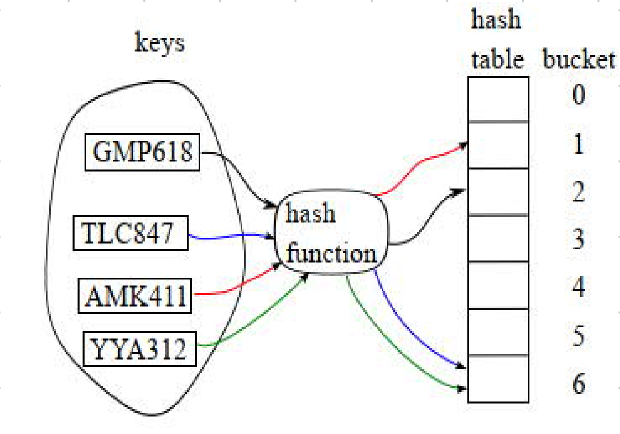
$n$ = number of records stored

$m$ = number of buckets

**Example**




If two (or more) keys get mapped to same bucket ... Collision!




In C++ STL unordered_set and unordered_map are hash tables.

From https://en.cppreference.com/w/cpp/container/unordered_map

# std::unordered_map

Defined in header <unordered_map>

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,                                    (1)  (since C++11)
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

```
namespace pmr {
    template <class Key,
              class T,
              class Hash = std::hash<Key>,                          (2)  (since C++17)
              class Pred = std::equal_to<Key>>
    using unordered_map = std::unordered_map<Key, T, Hash, Pred,
                              std::pmr::polymorphic_allocator<std::pair<const Key,T>>>;
}
```

Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket. This allows fast access to individual elements, since once the hash is computed, it refers to the exact bucket the element is placed into.

**Modifiers**

| | |
|---|---|
| clear (C++11) | clears the contents<br>(public member function) |
| insert (C++11) | inserts elements or nodes (since C++17)<br>(public member function) |
| insert_or_assign (C++17) | inserts an element or assigns to the current element if the key already exists<br>(public member function) |
| emplace (C++11) | constructs element in-place<br>(public member function) |
| emplace_hint (C++11) | constructs elements in-place using a hint<br>(public member function) |
| try_emplace (C++17) | inserts in-place if the key does not exist, does nothing if the key exists<br>(public member function) |
| erase (C++11) | erases elements<br>(public member function) |
| swap (C++11) | swaps the contents<br>(public member function) |
| extract (C++17) | extracts nodes from the container<br>(public member function) |
| merge (C++17) | splices nodes from another container<br>(public member function) |

**Lookup**

| | |
|---|---|
| at (C++11) | access specified element with bounds checking<br>(public member function) |
| operator[] (C++11) | access or insert specified element<br>(public member function) |
| count (C++11) | returns the number of elements matching specific key<br>(public member function) |
| find (C++11) | finds element with specific key<br>(public member function) |
| contains (C++20) | checks if the container contains element with specific key<br>(public member function) |
| equal_range (C++11) | returns range of elements matching a specific key<br>(public member function) |

## 2. Resolving collisions

Two strategies:

- open hashing or chaining
- closed hashing

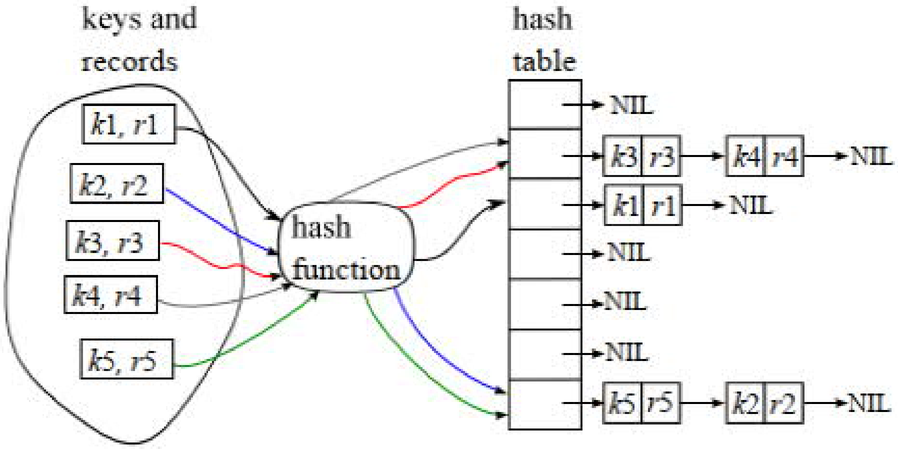
**Open hashing**

Buckets contain pointers to start of linked list.

**insert** : The hash function gives the correct bucket. The (key, record)-pair is stored at start of linked list.

**search** : The hash function gives the correct bucket, if the record exists. We must search through the linked list to find the key.

**delete** : The hash function gives the correct bucket, if the record exists. We must search through the linked list to find the key and then delete the (key, record)-pair.



From `https://en.cppreference.com/w/cpp/container/unordered_map`

**Bucket interface**

| | |
|---|---|
| **begin**(size_type)<br>**cbegin**(size_type) (C++11) | returns an iterator to the beginning of the specified bucket<br>(public member function) |
| **end**(size_type)<br>**cend**(size_type) (C++11) | returns an iterator to the end of the specified bucket<br>(public member function) |
| **bucket_count** (C++11) | returns the number of buckets<br>(public member function) |
| **max_bucket_count** (C++11) | returns the maximum number of buckets<br>(public member function) |
| **bucket_size** (C++11) | returns the number of elements in specific bucket<br>(public member function) |
| **bucket** (C++11) | returns the bucket for specific key<br>(public member function) |

## Closed hashing

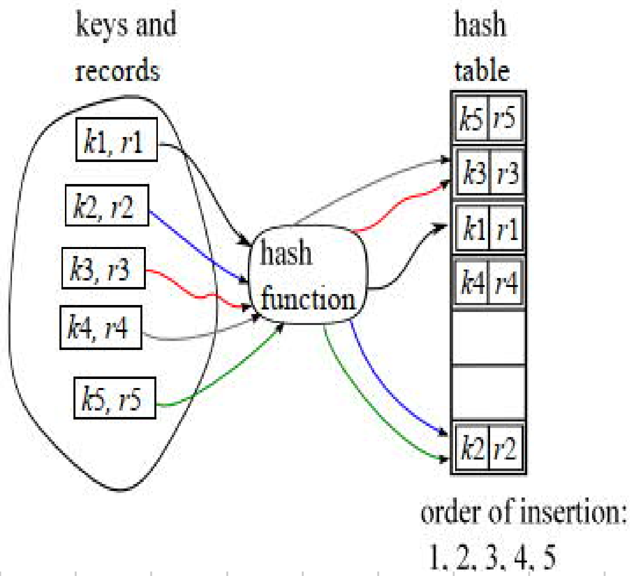
All data is stored in the buckets.

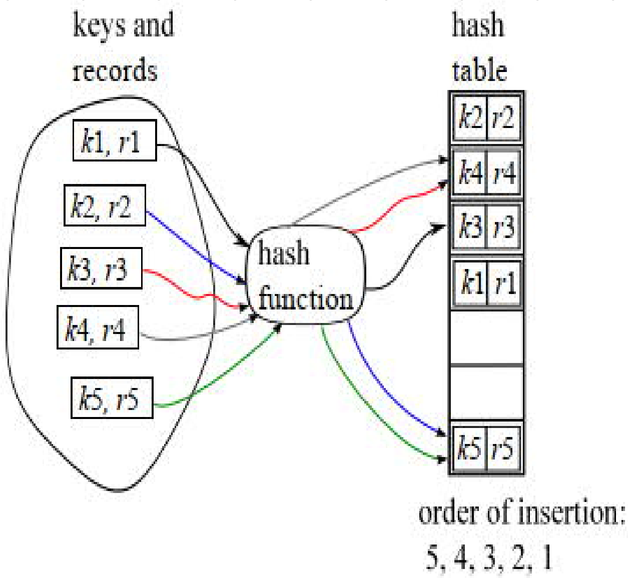
When collisions occur (or have occured), the bucket from the hash function may not be the bucket that meets our requirements. A special *probing function* is used to provide us with a sequence of buckets that we should check.

**insert** : The hash function gives the a bucket. If the bucket is not occupied, then store the (key, record)-pair in it. If the bucket is occupied, use probing function to find an unoccupied bucket.

**search** : The hash function gives a bucket. If the bucket contains the key, then we have found the (key, record)-pair. If the bucket does not contain the key, then we must use the probing function until we have checked all occupied buckets.

**delete** : The hash function gives a bucket. If the bucket contains the key, then we have found the (key, record)-pair. If the bucket does not contain the key, then we must use the probing function until we have checked all occupied buckets.




## 3. Hash functions

Properties of a **good** hash function $h(k)$:

- uniform distribution of keys amongst all buckets (minimize collisions)
- quick to compute ($O(1)$ in practice)
- deterministic: for a given key $k$ always produces same bucket
- should use all information of key

  Example: key is birthdate
  Bad choices: $dd.mm$, $dd.mm.yy$ Better choice: $dd.mm.yyyy$

- should be able to avoid assigning keys with regularities to same bucket(s)

  Example: key is integer with digits $x_1x_2 \ldots x_n$ 4556, 4679
  regularity: $x_i \leq x_{i+1}$

**Example**

Assume key $k$ is integer and $m$ is number of buckets.

Simple hash function: $h(k) = \text{mod}(k, m)$

mod(27, 5) = 2

When might this $h(k)$ be bad?

- suppose $m$ even and all keys are even
  consequence: half of buckets are always empty

- suppose $m = 10^r$ and key is integer with $s > r$ digits

$$k = \alpha_0 10^0 + \alpha_1 10^1 + \ \ldots \ + \alpha_s 10^s$$

m = 1000

k = 76487

  consequence: data $\alpha_r$, $\alpha_{r+1}$, ... $\alpha_s$ is not used

□

**Sensible policy: use existing (default) hash functions and focus on providing good keys.**

Hash functions and STL:

- declaring

```
std::unordered_map<std::string, myData> myMap;
```

simply uses STL's default hash function when key is string

- if key is number (`short`, `int`, `long`, `double` etc.) or pointer or string, best to use STL's own default hash and focus on forming keys
- `unordered_map` will allow user to input own hash function
- if key is `struct`, it might be possible to use some attribute as key

```
struct Car
{
  int yearMade;
  std::string licensePlate; // Best possible key.
  std::string color;
  std::string MotorType;
};
```

STL will not accept a struct as a key. In such cases it may be necessary to form a hash function.

Two examples of forming a hash function from a struct.

## Example I

```
struct Key {
  std::string first;
  std::string second;
};

struct KeyHash {
  std::size_t operator()(const Key& k) const
  {
    return std::hash<std::string>()(k.first) ^
    (std::hash<std::string>()(k.second) << 1);
  }
};

struct KeyEqual {
  bool operator()(const Key& lhs, const Key& rhs) const
  {
    return lhs.first == rhs.first && lhs.second == rhs.second;
  }
};

int main()
{
  // Define the KeyHash and KeyEqual structs and use them in the template
  std::unordered_map<Key, std::string, KeyHash, KeyEqual> newMap = {
    { {"John", "Doe"}, "example"}, { {"Mary", "Sue"}, "another"} };

}
```

^ XOR

## Example II

```
// Type for a coordinate (x, y)
struct Coord
{
    int x = NO_VALUE;
    int y = NO_VALUE;
};

// Example: Defining == and hash function for Coord so that it can be used
// as key for std::unordered_map/set, if needed
inline bool operator==(Coord c1, Coord c2) {return c1.x == c2.x && c1.y == c2.y; }
inline bool operator!=(Coord c1, Coord c2) {return !(c1==c2); } // Not strictly necessary

struct CoordHash
{
    std::size_t operator()(Coord xy) const
    {
        auto hasher = std::hash<int>();
        auto xhash = hasher(xy.x);
        auto yhash = hasher(xy.y);
        // Combine hash values (magic!)
        return xhash ^ (yhash + 0x9e3779b9 + (xhash << 6) + (xhash >> 2));
    }
};
```

**If your key is really composed of two or more attributes (elements), then search (Google) for help to obtain hash function.**

## 4. Rehashing

**Q**: What is a hash table's load factor $\alpha$?

**A**: The load factor is the average number of records per bucket or $\alpha = n/m$.

**Q**: What is a done in rehashing?

**A**: There are two parts:

- an acceptable limit $\alpha_{lim}$ is set for the load factor
- if $\alpha > \alpha_{lim}$, then a new hash table is created with $m_{new} > m$ buckets
- all records from old hash table are rehashed and restored in new hash table

Decisions:

- value for $\alpha_{lim}$?
- value for $m_{new}$?

The value for $\alpha_{lim}$ is a compromise:

- large $\alpha_{lim}$: fewer buckets and hence memory needed, but more records per bucket and hence longer average **search** and **delete** times
- small $\alpha_{lim}$: more buckets and hence memory needed, but fewer records per bucket and hence shorter average **search** and **delete** times

Choosing value for ratio $m_{new}/m$ is a compromise:

- smaller $m_{new}/m$: more frequent rehashes, less memory used when rehashing
- larger $m_{new}/m$: less frequent rehashes, more memory used when rehashing

STL's `unordered_set` and `unordered_map` have automatic rehashing using default values of $\alpha_{lim}$ and $m_{new}$.

However, user can set these.

From `https://en.cppreference.com/w/cpp/container/unordered_map`

**Hash policy**

| | |
|---|---|
| `load_factor` (C++11) | returns average number of elements per bucket (public member function) |
| `max_load_factor` (C++11) | manages maximum average number of elements per bucket (public member function) |
| `rehash` (C++11) | reserves at least the specified number of buckets and regenerates the hash table (public member function) |
| `reserve` (C++11) | reserves space for at least the specified number of elements and regenerates the hash table (public member function) |

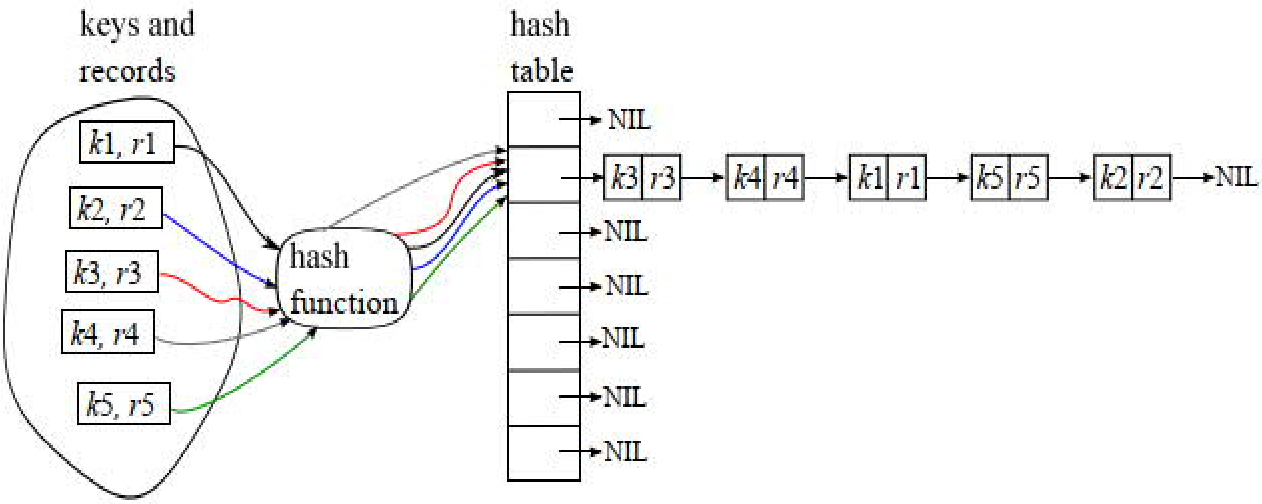
## 5. Runtime efficiency and amortized analysis

Assumptions

- use chaining (open hashing) for collisions.
- hash function computation is $O(1)$

### Worst case

All $n$ records are in the same bucket.




Operation efficiencies:

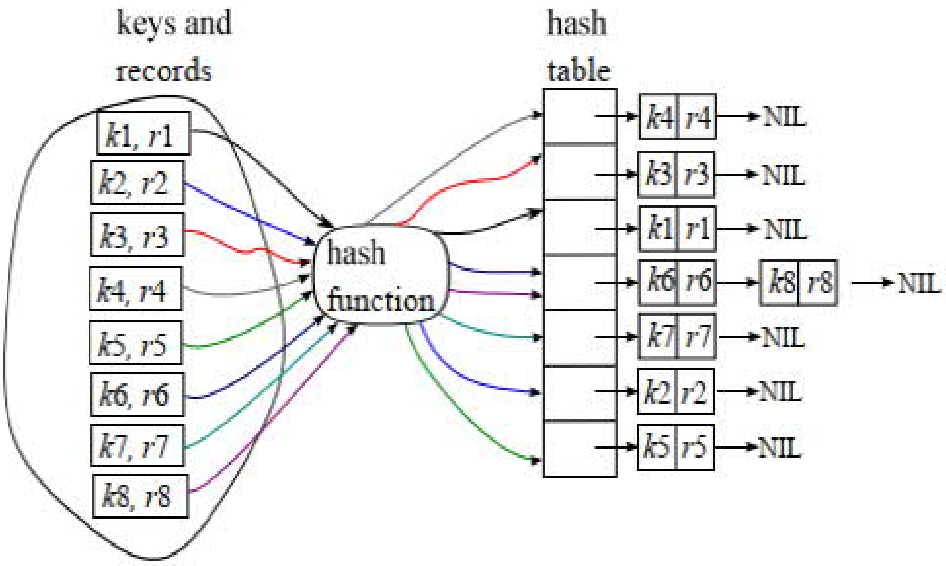
**search** $O(n)$ **delete** $O(n)$ **insert** $O(n)$

**Q** Why is **insert** $O(n)$?

**A** Have to check that record with same key does not already exist.

**Best case**

All buckets have (almost) same number of records: $\lceil n/m \rceil$.




Operation efficiencies:

**search** $\Theta(n/m)$ **delete** $\Theta(n/m)$ **insert** $\Theta(n/m)$

If number of buckets is proportional to number of records: $m = \beta n$ for some constant $\beta$:

**search** $\Theta(1/\beta) = \Theta(1)$ **delete** $\Theta(1/\beta) = \Theta(1)$ **insert** $\Theta(1/\beta) = \Theta(1)$

Note: $\beta = 1/\alpha$.

Two requirements to obtain runtime efficiencies that are constant time:

- $m$ is proportional to $n$
- **good hash function!** (does not have to be perfect)

**Q**: What to do when $n$ grows?
**A**: Rehash.

**Q**: Do we lose the $\Theta(1)$ efficiency from rehashing?
**A**: Not if we consider amortized analysis.

**Q**: What is amortized analysis?
**A**: Instead of just considering the runtime efficiency for an individual procedure or operation, we consider the average runtime efficiency for a sequence of operations.

**Example**

Assumptions:

- perform a sequence of **insert** operations and **rehash** operations
- **insert** is $\Theta(1)$
- **rehash** done when load factor is $\alpha >= 1$
- when hash table has $n$ records, **rehash** is just a sequence of $n$ **insert**s, hence $\Theta(n)$

Operation sequence:

**stage 1** Set hash table size to $m_0 = 1$. Do one **insert**. Set $i = 1$.

**stage 2** Set new hash table size to $m_i = 2m_{i-1}$. Perform a **rehash**.

**stage 3** Do $m_i - m_{i-1}$ **insert** operations. Set $i = i + 1$.

**stage 4** If $i = r$, then stop. If $i < r$, then repeat stages 2 and 3.

| $i$ | $m_{i-1}$ | $m_i$ | stage 2 operations counts | stage 3 operations counts |
|---|---|---|---|---|
| 1 | 1 | 2 | 1 | 1 |
| 2 | 2 | 4 | 2 | 2 |
| 3 | 4 | 8 | 4 | 4 |
| $\vdots$ | $\vdots$ | | | |
| $r$ | $2^{r-1}$ | $2^r$ | $2^{r-1}$ | $2^{r-1}$ |

Total number of elements in hash table:

$$n = \text{stage 1 } \textbf{insert} + \text{sum of stage 3 } \textbf{inserts}$$
$$= 1 + 1 + 2 + 4 + \ldots + 2^{r-1} = 2^r$$

Total operations count:

$$T = \text{stage 1 count} + \text{sum of stage 2 counts} + \text{sum of stage 3 counts}$$
$$= 1 + 2^r - 1 + 2^r - 1 = 2^{r+1} - 1$$

Amortized efficiency per operation:

$$\frac{T}{n} = \frac{2^{r+1} - 1}{2^r} \leq 2 = O(1)$$

Critical part: in stage 2, new hash table size is $m_i = fm_{i-1}$ for some $f > 1$.

□