# COMP122/20 - Data Structures and Algorithms

# 17 Hash Tables

*Instructor* : Ke Wei 〖柯韋〗

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

April 3, 2020



## Outline



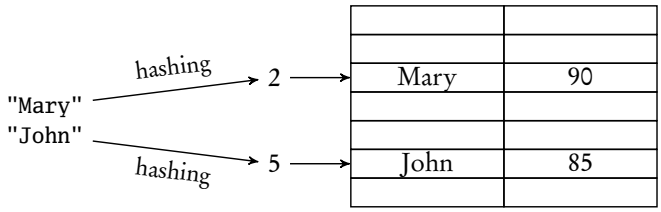
👁 *Textbook §10.2 – 10.3.*





## Hash Tables

- A hash table is another implementation of a associative arrays, or maps.
- A hash table is an array of size *ts*, if there's a function to map each key *k* to an array index *i*, then we may store, delete and find the key *k* in the array cell located by the index *i*. This operation costs constant time ideally.
- To translate a key of a much larger domain to an index of a smaller index domain (such as integers ranging from 0 to $ts-1$) is called *hashing*.
- The function that performs this mapping is called a *hash function*.



## Hash Functions

A hash function $h$ maps keys of a given type to integers in a fixed range, $0 \ldots ts-1$. For example,

$$h(k) = k \,\%\, ts$$

is a hash function for integer keys. The integer $h(k)$ is called the *hash value* of key $k$.

- A hash function is usually specified as the composition of two functions:

$$\text{(Hash code) } h_1 : \text{keys} \to \text{integers},$$
$$\text{(Compression function) } h_2 : \text{integers} \to 0 \ldots ts-1.$$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(k) = h_2(h_1(k)).$$

- The goal of the hash function is to "disperse" the keys in an apparently random way.





## Ideas of Hash Codes

Memory address:

- We reinterpret the memory address of the key object as an integer.
- Good in general, except for numeric and string keys, which have value semantics.

Integer cast:

- We reinterpret the bits of the key as an integer.
- Suitable for keys of length less than or equal to the CPU word length.

Component sum:

- We partition the bits of the key into components of fixed length (e.g., 32 or 64 bits) and we sum the components (ignoring overflows).
- Suitable for numeric keys of fixed length greater than or equal to the CPU word length.





## Hash Codes — Polynomial Accumulation

Polynomial accumulation:

- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16, 32 or 64 bits): $a_0 a_1 \ldots a_{n-1}$;
- We evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$ on a fixed value $z$, ignoring overflows, that is, modulo $2^32$ or $2^64$ in Python.
- Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words).

Polynomial $p(z)$ can be evaluated in $\mathcal{O}(n)$ time using Horner's rule:

- The following polynomials are successively computed, each from the previous one in $\mathcal{O}(1)$ time,

$$p_0(z) = a_{n-1}$$
$$p_i(z) = a_{n-i-1} + z \cdot p_{i-1}(z) \qquad \text{for } i = 1, 2, \ldots, n-1.$$

- Then, we have $p(z) = p_{n-1}(z)$.

## Compression Functions

- Division:
$$h_2(y) = y \,\%\, ts,$$
where the size $ts$ of the hash table is usually chosen to be a prime.
- Multiply, Add and Divide (MAD):
$$h_2(y) = (ay + b) \,\%\, ts,$$
where $a$ and $b$ are nonnegative integers such that $a \,\%\, ts \neq 0$, otherwise, every integer would map to the same value $b$.





## A Simple Hash Function

```
1  def hash_z(s, ts, z):
2      h = 0
3
4      for c in s:
5          h = h*z + ord(c)
6
7      return h % ts
8
9  def hash_ts(s, ts):
10     return hash_z(s, ts, 33)
```





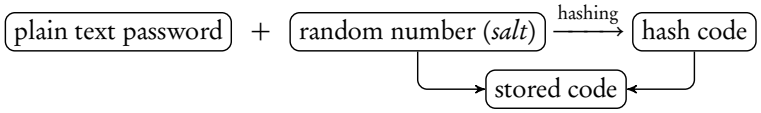
## Hash Codes in Python

- The default *__hash__(self)* method in a Python class maps each object to an integer that is derived from its memory address. In Python 3.8, we still have `hash(x) == id(x)//16` in a 64-bit machine. Because a 64-bit word takes up 16 bytes in the address space.
- The *__hash__(self)* for built-in tuples implements something similar to the *hash_z* function, with *c* replaced by the hash code of each component of the tuple.
- Each built-in type has this similar implementation.
- If we intend to use certain objects as keys in a map, that is, as immutables with value semantics, then we should reimplement the *__hash__(self)* method by listing all the attributes of the object as the components of a tuple, and use the hash code of the tuple as the result. For example, an object containing three attributes can implement its hash function like this.

```
def __hash__(self):
    return hash((self.attr_a, self.attr_b, self.attr_c))
```

# Other Usages of Hash Functions

- *Checksums*. Simple component-sum hash codes can be used to verify data integrity.
- *Cyclic redundancy checks (CRC)*. A CRC is the remainder of a binary division with no bit carry (XOR), of the message bit stream, by a predefined (short) bit stream, which represents the coefficients of a polynomial.
- *Cryptographic hash functions*. These functions have certain additional security properties. They can be used in authentication and message integrity checking. The hash codes produced are sometimes called *message digests* or digital fingerprints (e.g., SHA1, SHA256, MD5, etc.)




When we check passwords, we first hash the input with the salt, and compare the hash code with the one stored in the database.





# Collision Resolution

- There must exist some keys that have the same hash value unless the mapping is in effect injective. Since we are mapping a larger domain to a smaller range, an injective mapping is unlikely.
- If two keys have the same hash value, we call this a *collision*. For hash tables, the main programming effort is on collision resolution.
- We define the ratio of the number of elements ($n$) in the hash table to the table size ($ts$) as the *load factor*

$$\alpha = \frac{n}{ts}.$$

When the load factor $\alpha$ is greater, there should be more collisions.

- If the key we are inserting collides with an existing key, we need to resolve it. There are two simple methods: *separate chaining* and *open addressing*.





# Separate Chaining

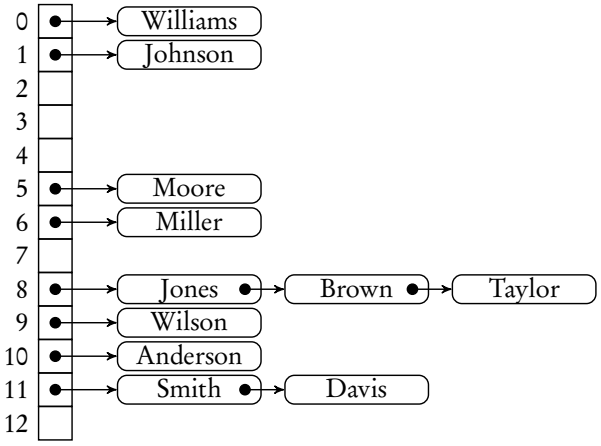
The first strategy is to keep a list of all elements that hash to the same value. The hash table becomes an array of pointers that point to linked lists.

- To find a key:
  - First compute its hash value $h$;
  - Traverse the $h^{\text{th}}$ list and search for the node with the given key.
- To insert a key:
  - Simply append it to the tail of the $h^{\text{th}}$ list.
- To remove a key:
  - First find the node with the key;
  - Remove it from the list.

Separate chaining require pointers and dynamic memory allocations, this slows down the algorithm. And the link fields in the nodes are a kind of waste.

# Separate Chaining — Illustrated

| *key* | *h* |
|---|---|
| Smith | 11 |
| Johnson | 1 |
| Williams | 0 |
| Jones | 8 |
| Brown | 8 |
| Davis | 11 |
| Miller | 6 |
| Wilson | 9 |
| Moore | 5 |
| Taylor | 8 |
| Anderson | 10 |

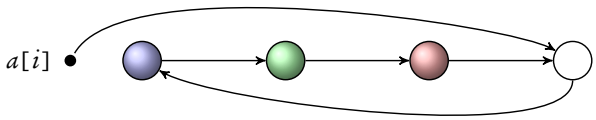







# Separate Chaining — Data Structures

```
1  class Node:
2      def __init__(self, key = None, value = None):
3          self.key, self.value = key, value
4          self.nxt = self
5
6  class HashTable:
7      def __init__(self, ts):
8          self.a = []
9          for _ in range(ts):
10             self.a.append(Node())   # dummy nodes
```








# Separate Chaining — Insertion

```
1      def __setitem__(self, key, value):
2          h = hash_ts(key, len(self.a))
3          dummy = self.a[h]
4          p = dummy.nxt   # first node
5
6          while p is not dummy and key != p.key:
7              p = p.nxt
8
9          if p is dummy:
10             self.a[h] = q = Node()
11             q.nxt = p.nxt
12             p.nxt = q
13             p.key = key
14         p.value = value
```

## Separate Chaining — Removal

```
1   def __delitem__(self, key):
2       h = hash_ts(key, len(self.a))
3       dummy = self.a[h]
4       p = dummy.nxt  # first node
5
6       while p is not dummy and key != p.key:
7           p = p.nxt
8
9       if p is not dummy:
10          q = p.nxt
11          p.key, p.value, p.nxt = q.key, q.value, q.nxt
12          if q is dummy:
13              self.a[h] = p
14      else:
15          raise KeyError
```





## Open Addressing

When a collision occurs, alternative cells are tried for key $k$ until an empty cell is found. Formally speaking, the cells (also called the collision chain)

$$g_0(k), g_1(k), g_2(k), \ldots$$

are tried in succession, where

$$g_i(k) = (h(k) + f(i,k)) \mod ts, \quad \text{with } f(0) = 0.$$

The function $f$ is called a probing function.

- Usually, the table load factor should below 0.5 to use open addressing. Otherwise, the number of tries may be too large to be efficient.
- Different probing functions result different resolution strategies.
- We must use lazy-removal not to break the collision chain.





## Some Probing Functions

- Linear probing:
$$f(i,k) = i.$$
- Quadratic probing:
$$f(i,k) = i^2.$$
- Double hashing:
$$f(i,k) = i \times h'(k).$$

## Probing and Lazy Removal

$h'(k) = 7 - (h(k) \% 7)$

| *key* | $h$ | $h'$ |
|---|---|---|
| Smith | 11 | 3 |
| Johnson | 1 | 6 |
| Williams | 0 | 7 |
| Jones | 8 | 6 |
| Brown | 8 | 6 |
| Davis | 11 | 3 |
| Miller | 6 | 1 |
| Wilson | 9 | 5 |
| Moore | 5 | 2 |
| Taylor | 8 | 6 |
| Anderson | 10 | 4 |

$f(i,k) = i$

| | | | |
|---|---|---|---|
| 0 | ✓ | Williams | 0 |
| 1 | ✗ | Johnson | 0 |
| 2 | ✓ | Taylor | 7 |
| 3 | ✓ | Anderson | 6 |
| 4 | ○ | | |
| 5 | ✓ | Moore | 0 |
| 6 | ✓ | Miller | 0 |
| 7 | ○ | | |
| 8 | ✓ | Jones | 0 |
| 9 | ✓ | Brown | 1 |
| 10 | ✓ | Wilson | 1 |
| 11 | ✗ | Smith | 0 |
| 12 | ✓ | Davis | 1 |

$f(i,k) = i \times h'(k)$

| | | | |
|---|---|---|---|
| 0 | | Williams | 0 |
| 1 | | Johnson | 0 |
| 2 | | | |
| 3 | | | |
| 4 | | Davis | 2 |
| 5 | | Moore | 0 |
| 6 | | Miller | 0 |
| 7 | | Brown | 2 |
| 8 | | Jones | 0 |
| 9 | | Wilson | 0 |
| 10 | | Anderson | 0 |
| 11 | | Smith | 0 |
| 12 | | Taylor | 5 |





## Open Addressing — Data Structures

```
1  class Entry:
2      EMPTY, REMOVED, VALID = object(), object(), object() # enums
3      def __init__(self):
4          self.info = Entry.EMPTY
5          self.key = self.value = None
6
7  class HashTable:
8      def __init__(self, ts):
9          self.a = []
10         for _ in range(ts):
11             self.a.append(Entry())
```





## Open Addressing — Retrieval

- The function *find* returns the entry containing the *key* in the hash table if the *key* is found.
- If the *key* is not found, it returns a removed entry or an empty entry that is ready to use.
- If neither is found, the function returns `None`. This means the table is full, all the items must be rehashed to a larger table to extend the size.

```
1      def find(self, key):
2          ts = len(self.a)
3          h = hash_ts(key, ts)
4          hh = 11-h%11     # probing increment
5          d = -1           # first removed key if any
```

## Open Addressing — Retrieval (2)

```
6           for i in range(ts):
7               j = (h+i*hh)%ts
8               if self.a[j].info is Entry.EMPTY:
9                   return self.a[j] if d < 0 else self.a[d]
10              elif self.a[j].info is Entry.REMOVED:
11                  if d < 0:
12                      d = j
13              else:
14                  if key == self.a[j].key:
15                      return self.a[j]
16          return None if d < 0 else self.a[d]
```





## Open Addressing — Insertion

```
1       def __contains__(self, key):
2           p = self.find(key)
3           return p is not None and p.info is Entry.VALID
4
5       def __setitem__(self, key, value):
6           p = self.find(key)
7           if p is None:
8               self.rehash()
9               p = self.find(key)
10          if p.info is not Entry.VALID:
11              p.key = key
12              p.info = Entry.VALID
13          p.value = value
```





## Analysis

- In the worst case, searches, insertions and removals on a hash table take $\mathcal{O}(n)$ time.
- The worst case occurs when all the keys inserted into the hash table collide.
- The load factor $\alpha = \frac{n}{ts}$ affects the performance of a hash table.
- Assume that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$\frac{1}{1-\alpha}.$$

- The expected running time of all the map operations in a hash table is $\mathcal{O}(1)$.
- In practice, hashing is very fast provided that the load factor is not close to 100%.