

11. Hashing

Hash Tables, Pre-Hashing, Hashing, Resolving Collisions using Chaining, Simple Uniform Hashing, Popular Hash Functions, Table-Doubling, Open Addressing: Probing [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

Motivating Example

Goal: Efficient management of a table of all n ETH-students of
Possible Requirement: fast access (insertion, removal, find) of a dataset
by name

Dictionary

Abstract Data Type (ADT) D to manage items¹² i with keys $k \in \mathcal{K}$ with operations

- **D.insert**(i): Insert or replace i in the dictionary D .
- **D.delete**(i): Delete i from the dictionary D . Not existing \Rightarrow error message.
- **D.search**(k): Returns item with key k if it exists.

¹²Key-value pairs (k, v) , in the following we consider mainly the keys

Dictionaries in Python

dictionary →

```
fruits = {  
    "banana": 2.95, "kiwi": 0.70,  
    "pear": 4.20, "apple": 3.95  
}
```

insert →

```
fruits["melon"] = 3.95
```

update →

```
fruits["banana"] = 1.90
```

find →

```
print("banana", fruits["banana"])  
print("melon in fruits", "melon" in  
fruits)print("onion in fruits"  
, "onion" in fruits)
```

remove →

```
del fruits["strawberry"]
```

iterate →

```
for name,price in fruits.items():  
    print(name,"->",price)
```

Dictionaries in Java

dictionary → `Map<String,Double> fruits =
new HashMap<String,Double>();`

insert → `fruits.put("banana", 2.95);
fruits.put("kiwi", 0.70);
fruits.put("strawberry", 9.95);
fruits.put("pear", 4.20);
fruits.put("apple", 3.95);`

update → `fruits.put("banana", 2.90);`

find → `Out.println("banana " + fruits.get("banana"));`

remove → `fruits.remove("banana");`

iterate → `for (String s: fruits.keySet())
Out.println(s+ " " + fruits.get(s));`

Motivation / Use

Perhaps **the** most popular data structure.

- Supported in many programming languages (C++, Java, Python, Ruby, Javascript, C# ...)
- Obvious use
 - Databases, Spreadsheets
 - Symbol tables in compilers and interpreters
- Less obvious
 - Substrin Search (Google, grep)
 - String commonalities (Document distance, DNA)
 - File Synchronisation
 - Cryptography: File-transfer and identification

1. Idea: Direct Access Table (Array)

Index	Item
0	-
1	-
2	-
3	[3,value(3)]
4	-
5	-
⋮	⋮
k	[k,value(k)]
⋮	⋮

Problems

1. Idea: Direct Access Table (Array)

Index	Item
0	-
1	-
2	-
3	[3,value(3)]
4	-
5	-
⋮	⋮
k	[k,value(k)]
⋮	⋮

Problems

1. Keys must be non-negative integers

1. Idea: Direct Access Table (Array)

Index	Item
0	-
1	-
2	-
3	[3,value(3)]
4	-
5	-
⋮	⋮
k	[k,value(k)]
⋮	⋮

Problems

1. Keys must be non-negative integers
2. Large key-range \Rightarrow large array

Solution to the first problem: Pre-hashing

Prehashing: Map keys to positive integers using a function $ph : \mathcal{K} \rightarrow \mathbb{N}$

- Theoretically always possible because each key is stored as a bit-sequence in the computer
- Theoretically also: $x = y \Leftrightarrow ph(x) = ph(y)$
- Practically: APIs offer functions for pre-hashing. (Java: `object.hashCode()`, C++: `std::hash<>`, Python: `hash(object)`)
- APIs map the key from the key set to an integer with a restricted size.¹³

¹³Therefore the implication $ph(x) = ph(y) \Rightarrow x = y$ does **not** hold any more for all x, y .

Prehashing Example : String

Mapping Name $s = s_1s_2 \dots s_{l_s}$ to key

$$ph(s) = \left(\sum_{i=0}^{l_s-1} s_{l_s-i} \cdot b^i \right) \bmod 2^w$$

b so that different names map to different keys as far as possible.

w Word-size of the system (e.g. 32 or 64)

Example (Java) with $b = 31$, $w = 32$. Ascii-Values s_i .

Anna \mapsto 2045632

Jacqueline \mapsto 2042089953442505 $\bmod 2^{32} = 507919049$

Implementation Prehashing (String) in Java

$$ph_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_{l-i+1} \cdot b^i \right) \bmod m$$

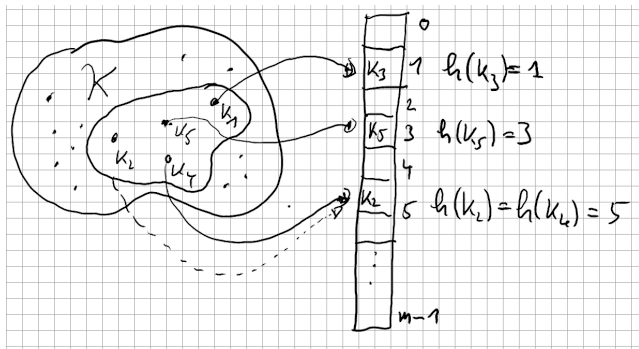
With $b = 31$ and $m = 2^{32}$ we get in Java¹⁴

```
int prehash(String s){
    int h = 0;
    for (int k = 0; k < s.length(); ++k){
        h = h * b + s.charAt(k);
    }
    return h;
}
```

¹⁴Try to understand why this works

Lösung zum zweiten Problem: Hashing

Reduce the universe. Map (hash-function) $h : \mathcal{K} \rightarrow \{0, \dots, m - 1\}$ ($m \approx n =$ number entries of the table)



Collision: $h(k_i) = h(k_j)$.

Nomenclature

Hash function h : Mapping from the set of keys \mathcal{K} to the index set $\{0, 1, \dots, m - 1\}$ of an array (**hash table**).

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Normally $|\mathcal{K}| \gg m$. There are $k_1, k_2 \in \mathcal{K}$ with $h(k_1) = h(k_2)$ (**collision**).

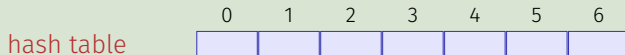
A hash function should map the set of keys as uniformly as possible to the hash table.

Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12

Direct Chaining of the Colliding entries



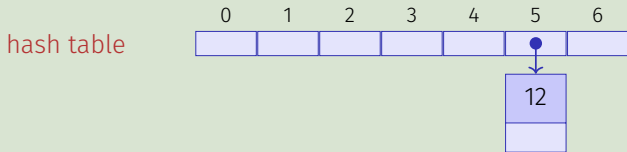
Colliding entries

Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55

Direct Chaining of the Colliding entries



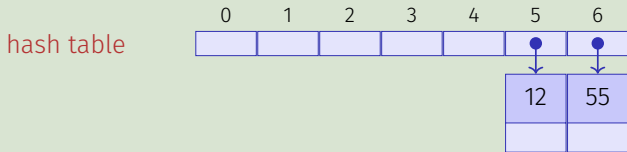
Colliding entries

Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5

Direct Chaining of the Colliding entries



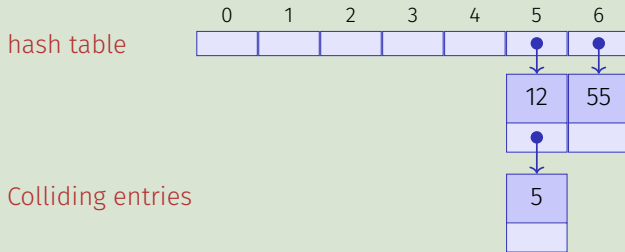
Colliding entries

Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5, 15

Direct Chaining of the Colliding entries

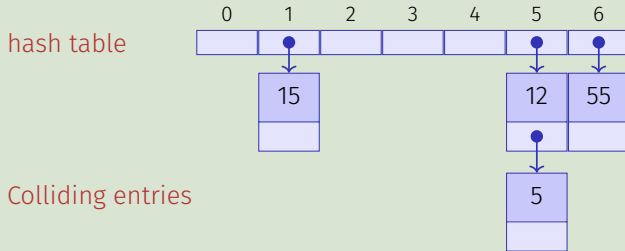


Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5, 15, 2

Direct Chaining of the Colliding entries

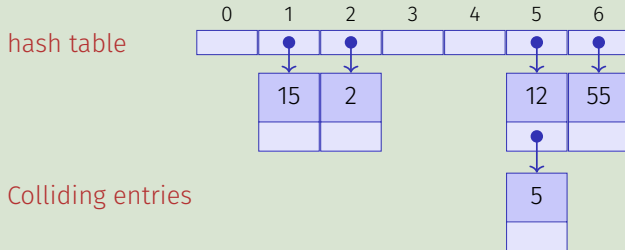


Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5, 15, 2, 19

Direct Chaining of the Colliding entries

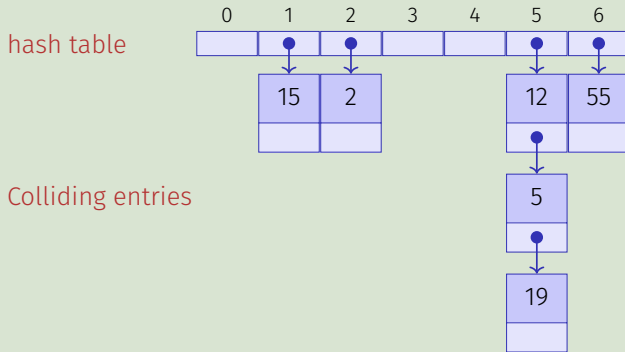


Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5, 15, 2, 19, 43

Direct Chaining of the Colliding entries

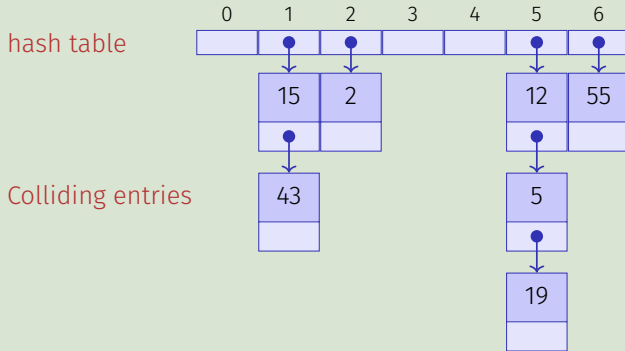


Resolving Collisions: Chaining

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Keys 12, 55, 5, 15, 2, 19, 43

Direct Chaining of the Colliding entries



Algorithm for Hashing with Chaining

- **insert**(i) Check if key k of item i is in list at position $h(k)$. If no, then append i to the end of the list. Otherwise replace element by i .
- **find**(k) Check if key k is in list at position $h(k)$. If yes, return the data associated to key k , otherwise return empty element **null**.
- **delete**(k) Search the list at position $h(k)$ for k . If successful, remove the list element.

Worst-case Analysis

Worst-case: all keys are mapped to the same index.

$\Rightarrow \Theta(n)$ per operation in the worst case. 😞

Simple Uniform Hashing

Strong Assumptions: Each key will be mapped to one of the m available slots

- with equal probability (Uniformity)
- and independent of where other keys are hashed (Independence).

Simple Uniform Hashing

Under the assumption of simple uniform hashing:

Expected length of a chain when n elements are inserted into a hash table with m elements

$$\begin{aligned}\mathbb{E}(\text{Länge Kette } j) &= \mathbb{E}\left(\sum_{i=0}^{n-1} \mathbb{1}(k_i = j)\right) = \sum_{i=0}^{n-1} \mathbb{P}(k_i = j) \\ &= \sum_{i=1}^n \frac{1}{m} = \frac{n}{m}\end{aligned}$$

$\alpha = n/m$ is called **load factor** of the hash table.

Simple Uniform Hashing

Theorem 5

Let a hash table with chaining be filled with load-factor $\alpha = \frac{n}{m} < 1$. Under the assumption of simple uniform hashing, the next operation has expected costs of $\leq 1 + \alpha$.

Consequence: if the number slots m of the hash table is always at least proportional to the number of elements n of the hash table, $n \in \mathcal{O}(m) \Rightarrow$ Expected Running time of Insertion, Search and Deletion is $\mathcal{O}(1)$.

Advantages and Disadvantages of Chaining

Advantages

- Possible to overcommit: $\alpha > 1$ allowed
- Easy to remove keys.

Disadvantages

- Memory consumption of the chains-

An Example of a popular Hash Function

Division method

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

Ideal: m prime, not too close to powers of 2 or 10

But often: $m = 2^k - 1$ ($k \in \mathbb{N}$)

Other method: multiplication method (cf. Cormen et al, Kap. 11.3).

Table size increase

- We do not know beforehand how large n will be
- Require $m = \Theta(n)$ at all times.

Table size needs to be adapted. Hash-Function changes \Rightarrow **rehashing**

- Allocate array A' with size $m' > m$
- Insert each entry of A into A' (with re-hashing the keys)
- Set $A \leftarrow A'$.
- Costs $\mathcal{O}(n + m + m')$.

How to choose m' ?

Table size increase

- 1. Idea $n = m \Rightarrow m' \leftarrow m + 1$

Increase for each insertion: Costs $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$ 😞

- 2. Idea $n = m \Rightarrow m' \leftarrow 2m$ Increase only if $m = 2^i$:

$\Theta(1 + 2 + 4 + 8 + \dots + n) = \Theta(n)$

Few insertions cost linear time but on average we have $\Theta(1)$ 😊

Jede Operation vom Hashing mit Verketteten hat erwartet amortisierte Kosten $\Theta(1)$.

(\Rightarrow Amortized Analysis)

Amortisierte Analyse

General procedure for dynamic arrays (e.g. Java: **ArrayList**, Python: **List**)

- The data structure provides, besides the data array, two numbers: size of the array (capacity m) and the number of used entries (size n)
- Double the size and copy entries when the list is full $n = m \Rightarrow m \leftarrow 2n$. Kosten $\Theta(m)$.
- Runtime costs for $n = 2^k$ insertion operations:
 $\Theta(1 + 2 + 4 + 8 + \dots + 2^k) = \Theta(2^{k+1} - 1) = \Theta(n)$.

Costs per operation **averaged over all operations** = **amortized costs** = $\Theta(1)$
per insertion operation

Open Addressing

Store the colliding entries directly in the hash table using a **probing function** $s : \mathcal{K} \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$

Key table position along a **probing sequence**

$$S(k) := (s(k, 0), s(k, 1), \dots, s(k, m - 1)) \pmod{m}$$

Probing sequence must for each $k \in \mathcal{K}$ be a permutation of $\{0, 1, \dots, m - 1\}$

Notational clarification: this method uses **open addressing** (meaning that the positions in the hashtable are not fixed) but it is a **closed hashing** procedure (because the entries stay in the hashtable)

Algorithms for open addressing

- **insert**(i) Search for key k of i in the table according to $S(k)$. If k is not present, insert k at the first free position in the probing sequence. Otherwise error message.
- **find**(k) Traverse table entries according to $S(k)$. If k is found, return data associated to k . Otherwise return an empty element **null**.
- **delete**(k) Search k in the table according to $S(k)$. If k is found, replace it with a special key **removed**.

Linear Probing

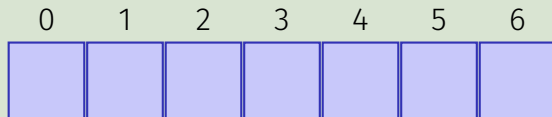
$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12

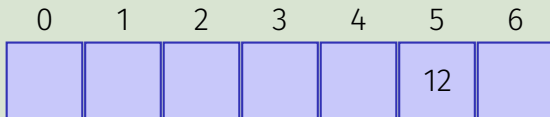


Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55



Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55, 5

0	1	2	3	4	5	6
					12	55

Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55, 5, 15

0	1	2	3	4	5	6
5					12	55

Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55, 5, 15, 2

0	1	2	3	4	5	6
5	15				12	55

Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2			12	55

Linear Probing

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Key 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

Discussion

Discussion

Example $\alpha = 0.95$

The unsuccessful search considers 200 table entries on average! (here without derivation).

Discussion

Example $\alpha = 0.95$

The unsuccessful search considers 200 table entries on average! (here without derivation).

Disadvantage of the method?

Discussion

Example $\alpha = 0.95$

The unsuccessful search considers 200 table entries on average! (here without derivation).

Disadvantage of the method?

Primary clustering: similar hash addresses have similar probing sequences
⇒ long contiguous areas of used entries.

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

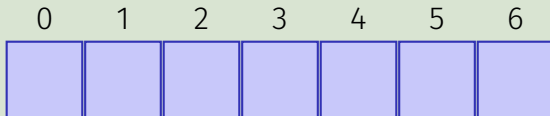
Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12



Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55

0	1	2	3	4	5	6
					12	

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55, 5

0	1	2	3	4	5	6
					12	55

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55, 5, 15

0	1	2	3	4	5	6
				5	12	55

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55, 5, 15, 2

0	1	2	3	4	5	6
	15			5	12	55

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
	15	2		5	12	55

Quadratic Probing

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Keys 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
19	15	2		5	12	55

Discussion

Example $\alpha = 0.95$

Unsuccessfully search considers 22 entries on average (here without derivation)

Discussion

Example $\alpha = 0.95$

Unsuccessfully search considers 22 entries on average (here without derivation)

Problems of this method?

Discussion

Example $\alpha = 0.95$

Unsuccessfully search considers 22 entries on average (here without derivation)

Problems of this method?

Secondary clustering: Synonyms k and k' (with $h(k) = h(k')$) traverses the same probing sequence.

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

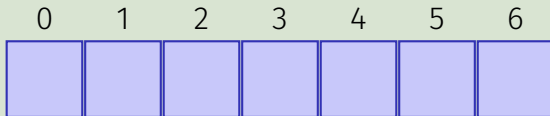
Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12



Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55

0	1	2	3	4	5	6
					12	

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55, 5

0	1	2	3	4	5	6
					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55, 5, 15

0	1	2	3	4	5	6
5					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55, 5, 15, 2

0	1	2	3	4	5	6
5	15				12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2			12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Keys 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

Double Hashing

- Probing sequence must permute all hash addresses. Thus $h'(k) \neq 0$ and $h'(k)$ may not divide m , for example guaranteed with m prime.
- h' should be as independent of h as possible (to avoid secondary clustering)

Independence largely fulfilled by $h(k) = k \bmod m$ and $h'(k) = 1 + k \bmod (m - 2)$ (m prime).

Uniform Hashing

Strong assumption: the probing sequence $S(k)$ of a key l is equally likely to be any of the $m!$ permutations of $\{0, 1, \dots, m - 1\}$

(Double hashing is reasonably close)

Analysis of Uniform Hashing with Open Addressing

Theorem 6

Let an open-addressing hash table be filled with load-factor $\alpha = \frac{n}{m} < 1$. Under the assumption of uniform hashing, the next operation has expected costs of $\leq \frac{1}{1-\alpha}$.

Without Proof, cf. e.g. Cormen et al, Kap. 11.4