Open & Bucket Hashing

Introduction

Hashing: a method for storing and retrieving records from a database based on some attribute value of the records.

- Generate a *hopefully unique* key for each record
- Insertion, deletion, and search is based on the key value of the record

Careful implementation of hashing allows for constant time insertion, deletion, and search on average.

Hashing for Storage: The Model

Imagine that you're searching for a book in a gigantic library.

- The books are not ordered meaningfully by title, author, ISBN, or anything
- Instead, to find a book, you use a "crystal ball" which tells you the shelf number where the book is located.

Introduction

- Appropriate for
  - applications where all search is done by exact-match queries
- Not Appropriate for
  - applications where multiple records with the same key are permitted
  - answering range searches

Java Implementations: Hashtable, HashMap, HashSet

Hashing, Hash Systems, and Hash Tables

Hashing creates some slightly overloaded terms. Some disambiguation:

- **Hashing** refers to the process of applying a hash function to a key.
- A **hash function** is a mathematical object that generates maps a key to an integer.
- A **hash table** is a data structure that stores records in an array. A record's position is determined by applying the hash function to the record's key.
- A **hash system** is a system that uses a hash table to store records & resolve collisions.

Defining a Hash System

A hash system consists of a hash table and a hash function.

- A hash table T is an array of *slots*. Depending on the hash system, the slots may contain records or auxilliary data structures.
  - We use M to denote the number of slots in a hash table.
- A hash function h maps a key K to a slot in the hash table.
  - $h(K) = i$ refers to a slot in the hash table such that $0 \leq i < M$.

In a given hash system, a record with key K is stored at $T[h(K)]$.

Collisions

Collision: when two search keys are mapped by the hash function to the same slot in the hash table

Finding a record with key K in a database organized by hashing follows a two-step procedure:

- Compute the table location $h(K)$
- Starting with slot $h(K)$, locate the record containing key K using (if necessary) a *collision resolution policy*

Hashing & Collisions: An Example

First, we hash 🐱 to find its slot in the table. `h(🐱) = 3`.

We can place 🐱 in slot 3.

0	1	2	3	4	5	6	7	8	9
			🐱						

Hashing & Collisions: An Example

Then, we hash 🐶 to find its slot in the table. `h(🐶) = 8`.

We can place 🐶 in slot 8.

0	1	2	3	4	5	6	7	8	9
			🐱					🐶	

Hashing & Collisions: An Example

Then, we hash 🐟 to find its slot in the table. `h(🐟) = 0`.

We can place 🐟 in slot `0`.

0	1	2	3	4	5	6	7	8	9
🐟			🐱					🐶	

Hashing & Collisions: An Example

Then, we hash 🐘 to find its slot in the table. `h(🐘) = 0`.

We can try to place 🐘 in slot 3. But it's filled!

This is a **collision**—where would we put 🐘 so that:

- we can find it again?
- we don't overwrite any other records?

0	1	2	3	4	5	6	7	8	9
🐟			🐱					🐶	

Collision Resolution Policies

Hashing and storing is simple, but collision resolution comes with a lot of tradeoffs!

- **Open hashing**: store multiple records in the same slot by using auxiliary data structures.
  - ➡️ Not all of the information is stored in the table itself
- **Closed hashing**: store one record per slot, and use a collision resolution policy to find a new slot for a record that collides with an existing record.
  - ➡️ All of the information is stored in the table itself

More (open) slots require more (unused) space, but allows for fewer collisions. Fewer slots requires less space, but may lead to more collisions, requiring more time to resolve.

Load Factor

Used to decide when to rehash (resize) the hash table

Load factor = $\frac{n}{M}$

- n = number of records in the hash table
- M = the number of slots in the hash table

Resize and rehash the hash table when the load factor exceeds a certain threshold to keep it as low as possible.

Open Hashing

Open Hashing

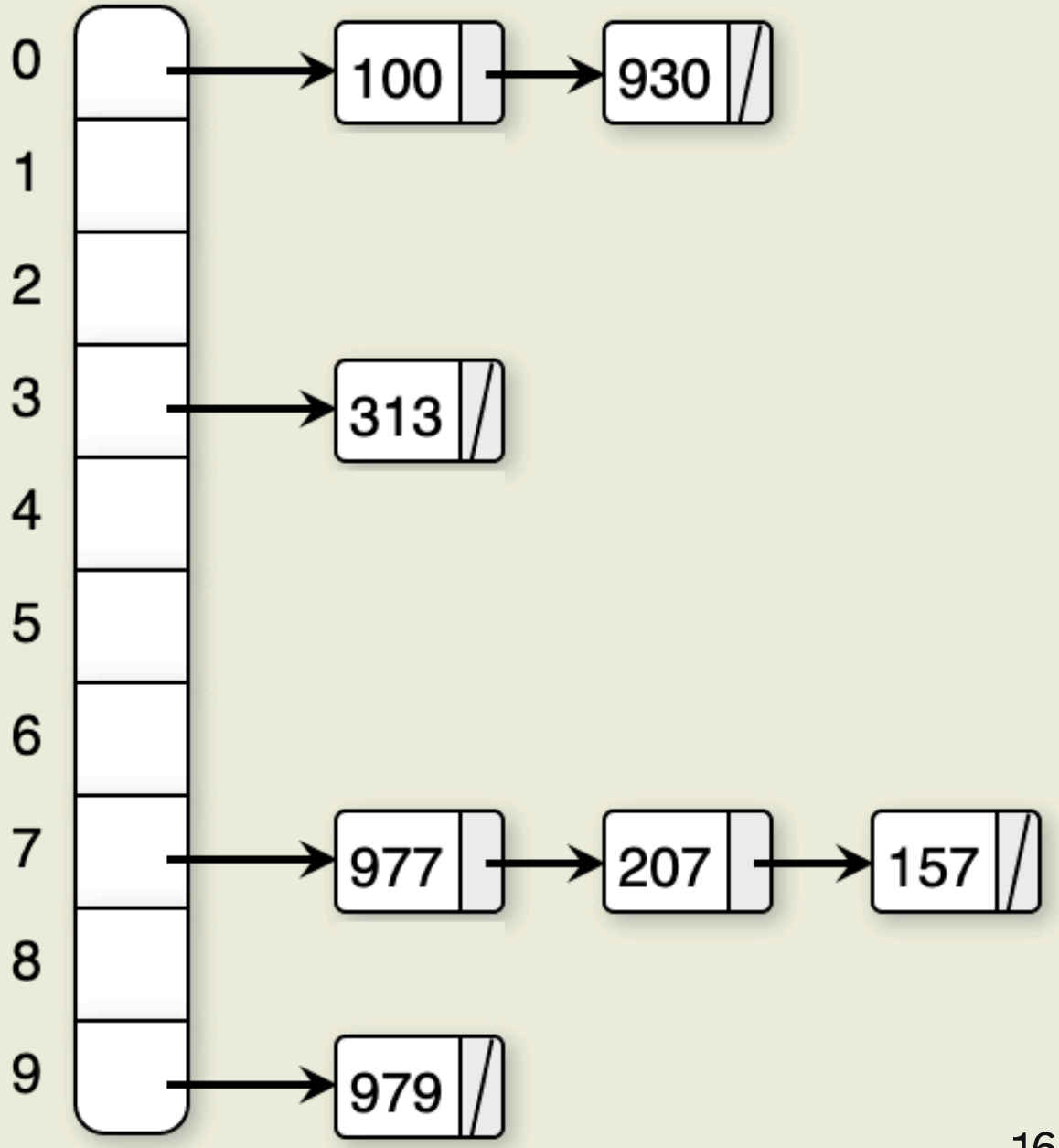
A hash system where multiple records might be associated with the same slot of a hash table.

- A linked list or other data structure is used to store the records in a slot

Example

Open Hashing, using a simple mod (%) as hash function:

- $h(K) = K\%M$, with M = the number of slots in the hash table.
- If we insert the keys: 157, 313, 930, 207, 979, 100, 977



Open Hashing

- **Insertion**:
  - Compute the hash value of the key
  - Insert the record at the head of the linked list at the hash value
- **Searching**:
  - Compute the hash value of the key
  - Search the linked list at the hash value
- **Deletion**:
  - Compute the hash value of the key
  - Delete the record from the linked list at the hash value

Runtimes??

Open Hashing

- **Insertion**: $O(\frac{n}{M})$ on average, $O(n)$ in the worst case
  - Compute the hash value of the key
  - Insert the record at the head of the linked list at the hash value
- **Searching**: $O(\frac{n}{M})$ on average, $O(n)$ in the worst case
  - Compute the hash value of the key
  - Search the linked list at the hash value
- **Deletion**: $O(\frac{n}{M})$ on average, $O(n)$ in the worst case
  - Compute the hash value of the key
  - Delete the record from the linked list at the hash value

Java HashMap

HashMap

Hash function: bitwise AND (&) operator

```
public static int findSlot(Object toHash) {
  slot = array_length & key.hashcode()
}
```

- ***Treeification*** when list size above a threshold (8)
- $O(\log n)$ in the worst case

Closed Hashing

Closed Hashing

A hash system where all records are stored in slots inside the hash table.

Implementations:

- **Closed hashing with buckets**
- **Closed hashing with no buckets**

Closed Hashing with Buckets

Bucket Hashing

Slots of the hash table are grouped into **buckets**

- If the hash table has M slots and B buckets ($M > B$), each bucket will consist of $\frac{M}{B}$ slots.
- Additionally, the table will include an **overflow bucket**: the *bucket* into which a record is placed if the bucket containing the record's *home slot* is full
    - Overflow bucket is often considered to have infinite capacity—an `ArrayList`, perhaps

Bucket Hashing: Insertion

- Hash the key to determine which bucket should contain the record
- If the bucket is not full, insert the record in the first available slot
- If the bucket is full then store the record in the first available slot in the overflow bucket

Bucket Hashing: Searching

Hash the key to determine which bucket should contain the record.

The records in this bucket are then searched.

- If the desired key value is not found and the bucket still has free slots, then the search is complete.
- If the bucket is full, then search the overflow bucket until the record is found or all records in the overflow bucket have been checked.

Expensive process if many records are in the overflow bucket!

Collision Resolution Policy

The process of finding the proper position in a hash table that contains the desired record

Used if the hash function did not return the correct position for that record due to a collision with another record

Mainly used in closed hashing systems

Closed Hashing with No Bucket

Implements a collision resolution policy

Closed Hashing with No Bucket

Hash function: simple mod (%)

Slot = key.hashcode() % array_size

Collision Resolution

Goal: find a free slot in the hash table when the home position for the record is already occupied

Uses a probe function

Collision Resolution

Probe function: function used by a *collision resolution* method to calculate where to look next in the *hash table*

Probe sequence: the series of *slots* visited by the *probe_ _function* during *collision resolution*.

Collision Resolution

Find home slot

- `int pos = home = h(K);` where h is the hash function and K is the key
  Probe sequence (iterative process)
- `pos = (home + p(k, i)) % M;`
  - Initialize `i` at 1
  - Increment `i` until the slot at `pos` is empty
    The probe function returns an offset from the original home position

Probe function

Collision Resolution Policies

Linear probing

Linear probing by steps

Pseudo-random probing

Quadratic probing

Double hashing

Linear Probing

Works by moving sequentially through the hash table from the *home slot*.

Probe function:

- `p(k, i) = i`
  If home slot is `home`, the probe sequence will be `home + 1, home + 2, home + 3, … home + (M – 1)`

Primary Clustering

- The tendency in certain collision resolution methods to create clustering in sections of the hash table
- Happens when a group of keys follow the same probe sequence during collision resolution
- primary clustering lead to empty slots in the table to not have an equal probability of receiving the next record inserted

Primary Clustering

- Linear probing leads to primary clustering
- Linear probing is one of the worst collision resolution methods

Linear Probing by Steps

Goal: avoid primary clustering / improve linear probing

Idea: skip slots by some constant `c`_ _other than 1

Probe function:

- `p(k, i) = ci`
  c must be relatively prime to *M* to generate a linear probing sequence that visits all slots in the table