CSE 241 Class 6

Jeremy Buhler

September 16, 2015

Today: a new idea – **hashing**!

1 Collections

- **Collection**: a data structure that stores a bunch of objects (aka *records*).
- each object has a *key* k that identifies it (plus some other data maybe) [draw picture]
- The world is full of collections:
  - shopping list
  - student registration database
  - table of courses, etc.
  - **[ask students to think of some others]**

Dynamic collections have some common methods:

- `insert`(x) – insert record x into collection
- `delete`(x) – delete record x from collection
- `find`(k) – locate a record with key k in collection, or fail (e.g. return NULL) if no such record exists

Note: `insert` and `delete` take references to records. If you want to delete a record with key k, you must first `find` the record, then `delete` it.

2 Basic Collections

Some common collections are *lists* (assume doubly linked) and *arrays*. They can be sorted or unsorted.

Let n be number of elements currently in a collection.

structure	`insert`	`delete`	`find`	space
unsorted list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

Among these choices, we can trade off between fast insert/delete and fast search. **[Ask students: any other collection types?]**

3 Directly Addressed Tables

- **Suppose we know** that records can have only a small finite set of keys, e.g., $0 \ldots r-1$ for some small r.
- Let's allocate an array of r pointers (or references). These are the table's **slots**.
- If input record has key k, put it in slot k of array.
- Empty slots are null pointers.

[draw an example of size 5 with a few records in it]

What are the costs of direct table?

structure	`insert`	`delete`	`find`	space
direct table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(r)$

Conclude: used space ($r \geq n$) to save time! Space is independent of number of elements.

4 Why Hashing is Needed

What if r is big? Let $\mho$ be the universe of all keys.

- social security #'s: $|\mho| \approx 10^9$
- IP addresses: $|\mho| \approx 4 \times 10^9$ (IPv6: 10^{38})
- UNIX passwords: $|\mho| \approx 10^{15}$

In many applications, $n \ll r$. **Do you really want to spend space proportional to r for these kinds of sets?**

Want to get direct table-like performance without direct table-like space.

5 Hashing Definition

Hash tables are a variant on the idea of direct-addressed tables. They use $\Theta(n)$ space but can give $\Theta(1)$ *average* performance for all basic operations.

- Let's fix a table size m independent of # records n.
- Now define a *hash function* $h(k)$ that maps any key to numbers in the range $0 \ldots m-1$.
$$h(k) : \mho \to [0 \ldots m-1]$$
- A record with key k goes in slot $h(k)$ of the table.
- Slots with no record are "null".

[sketch a picture of table with several keys in the universe mapping to slots in it.]

- **Question**: can two distinct keys hash to the same slot?
- **Yes**: if $m < r$, it *must* be possible by Pigeonhole Principle.
- When two keys hash to the same slot, we call this a **collision**. **[add a collision to the diagram]**
- What can we do to resolve collisions?

6 Collision Resolution by Chaining

Chaining lets multiple records share a single slot.

- Every record that hashes to a given slot s gets added to an *unsorted list* whose head is linked to s.
- a slot with no records has an empty list
- T.`insert`(x): add x to head of list $h(x.\text{key})$
- T.`find`(k): checks every record in list $h(k)$ until we find one whose key equals k (success) or exhaust list (failure).
- T.`delete`(x): remove x from its list

[**Example of Building and Searching a Chained Table**]

Defn: if m is number of slots in table, and n is number of records in table, then table's **load factor** α is defined by

$$\alpha = \frac{n}{m}$$

What is load factor of example table?

In chaining table, how big can load factor get? [arbitrarily large – can handle any number of records]

7 Collision Resolution by Open Addressing

Sometimes, we don't want to maintain linked lists outside the table. We can resolve collisions internally through open addressing.

- Each key k in $\mho$ maps to a *sequence* of slots $s_0(k), s_1(k), s_2(k) \ldots s_{m-1}(k)$
- Distinct keys map to distinct (but possibly overlapping) sequences of slots (also called *probe sequences*)
- If all of slots $s_1(k) \ldots s_i(k)$ are full, try to put record in slot $s_{i+1}(k)$.
- Slots may be full, empty, or "deleted"

Slot sequences for a given key can be derived many ways, but a good one in practice is **double hashing**.

- define two *different* hash functions $h_1(k)$, $h_2(k)$
- define $s_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod m$
- ex: $s_0(k) = h_1(k)$, $s_1(k) = (h_1(k) + h_2(k)) \bmod m$, etc.
- **Note**: in real code, you should write

$$s_{i+1}(k) = (s_i(k) + [h_2(k)] \bmod m) \bmod m$$

  to avoid integer overflow problems. (Assumes $2m <$ max integer)

What do the three basic operations look like now?

- T.`insert`(x):
  1. find first slot $s^* = s_j(x.\text{key})$ in table that is not full (i.e. empty or "deleted")
  2. Put x in $T[s^*]$

  (Does a free slot always exist? **No**: we may fail if table is full! (or if slot sequence fails to cover all slots)
- T.`find`(k): check each slot $s_j(k)$ in k's slot sequence until either
  1. we find that $T[s_j(k)]$ holds a record with key k (success!)
  2. we find that $T[s_j(k)]$ is empty (failure!)

  (Cannot stop at a "deleted" cell; we'll see why shortly)
- T.`delete`(x): If x is in slot s of T, $T[s] \leftarrow$ "deleted"

  (We want deletion to take constant time, which is not possible if we have to move up to cn other elements in x's slot sequence to fill the deleted slot.)

[Example of Building and Searching an OA Table]