CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by Anna Bretscher and Albert Lai

dictionaries (again)

Recall that a dictionary is an ADT that supports the following operations on a set of elements with well-ordered key-values k-v:

1. `insert(k, v)`: insert new key-value pair `k-v`
2. `delete(k)`: delete the node with key `k`
3. `search(k)`: find the node with key `k` (or value associated with key `k`)

Q. If we know the keys are integers from 1 to K, what is a fast and simple way to represent a dictionary?

A.

This data structure is called direct addressing.

Q. What is the asymptotic worst-case time for each of the important operations?

A.

direct addressing

Q. What may be a problem with direct addressing?

A.

Example 1: Reading a text file

Suppose we want to keep track of the frequencies of each letter in a text file.

Q: Why is this a good application of direct addressing?

A. **Example 2**: Reading a data file of 32-bit integers

Suppose we want to keep track of the frequencies of each number.

Q: Is this a good or bad application of direct addressing?

A.

hashing: idea

- the range of keys is large
- but many keys are not "used"
- don't need to allocate space for all possible keys

A hash table:

- if keys come from a universe (set) U
- allocate a table (an array) of size m (where $m < |U|$)
- use a hash function $h : U \to \{0, \ldots, m-1\}$ to decide where to store the element with key x
- x gets stored in position $h(x)$ of the hash table

hashing: problem

If $m < |U|$, then there must be $k_1, k_2 \in U$ such that $k_1 \neq k_2$ and yet $h(k_1) = h(k_2)$.

This is called a collision.

How we deal with collisions is called collision resolution. When we study hashing, we mostly study collision resolution.

collision resolution: idea

Say we have a small address book and one of the letters fills up, for example, "N"s. Where do you add the next "N" entry?

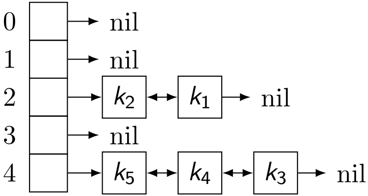
- flip to the next page
- have an overflow page at the very end
- write a little note explaining where to find rest of the "N" names

Two general collision resolution approaches:

1. Closed Addressing: Keys are always stored in the bucket they hash to — use additional data structure to store the keys in the same bucket.
2. Open Addressing: Give a general rule of where to look next (directions to another bucket).

closed addressing: chaining

Idea: store a doubly linked list at each entry in the hash table



An element with key k_1 and an element with key k_2 can both be stored at position $h(k_1) = h(k_2)$.

This is called chaining.

chaining: complexity

- Assume we can compute the hash function *h* in constant time.
- `insert(k,v)` takes:
- `delete(k)` takes:
    -
    -
- `search(k)` takes:

chaining: worst case

Q. What happens if $|U| > m * n$?

A.

Q. What is the worst case?

A.

simple uniform hashing

We assume hash function h has the simple uniform hashing property:

- any element is eqaully likely to hash into any of m buckets
  - independently of where any other element has hashed to, and
- h distributes elements of U evenly across m buckets
- formally:
  - sample space: set of elements with key-values from U
  - for any probability distribution on U
  - $$Pr(h(k) = i) = \frac{1}{m} \text{ for all } 1 \leq i \leq m, k \in U \text{ and}$$
  - $$\sum_{k \in U_i} \Pr(k) = \frac{1}{m} \text{ where } U_i = \{k \in U \mid h(k) = i\}$$

load factor

Q. If the table has n elements, how many would you expect in any one entry of the table ?

A.

- We call this ratio n/m the load factor, denoted by α.
- This simple uniform hashing assumption may or may not be accurate depending on U, h and the probability distribution for $k \in U$.

average case analysis

Calculating the average-case run time:

- Let T_k be a random variable which counts the number of elements checked when searching for key k.
- Let L_i be the length of the list at entry i in the hash table.
- Either we are searching for an item in the table or not in the table.

average case analysis: unsuccessful search

$E(T) =$

average case analysis: successful search

- Suppose we are searching for any of the n elements in the hash table, with equal probability, $1/n$.
- The number of elements examined before we reach the element x we are looking for is determined by the number of elements inserted **after** x.
- Expected number of elements examined is:

Let:

- $k_1, k_2, \ldots, k_n$: keys inserted, in order
- X_{ij} indicator variable of event that $h(k_i) = h(k_j)$
- then $E[X_{ij}] =$

average case analysis: successful search

average case analysis: successful search

$E(T) =$

average case of search — closed addressing

- So the average-case running time of `search` under simple uniform hashing with chaining is $\Theta(1 + \alpha)$.
- If the number of slots is proportional to number of elements in the table, then n is $\mathcal{O}(m)$ and so search takes constant time on average.

open addressing

- Each entry in the hash table stores a fixed number c of elements.
- This has the immediate implication that we only use it when $n \leq cm$.
- We will keep c at 1 for today's class.

To insert a new element if we get a collision:

- Find a new location to store the new element.
- We need to know where we put it: for future retrieval.
- Search a well-defined sequence of other locations in the hash table, until we find one that's not full.

This sequence is called a probe sequence.

probe sequences

Many methods for generating a probe sequence. For example:

- linear probing: try $A[(h(k) + i) \bmod m]$, $i = 0, 1, 2, \ldots$
- quadratic probing: try $A[(h(k) + c_1 i + c_2 i^2) \bmod m]$
- double hashing: try $A[(h(k) + i \cdot h'(k)) \bmod m]$
  where h' is another hash function

linear probing

For a hash table of size m, key k and hash function h, the probe sequence is calculated as:

$$s_i = (h(k) + i) \bmod m \quad \text{for } i = 0, 1, 2, \ldots$$

- $s_0 = h(k)$ is called the home location for the item
- the problem:
  -
- when we hash to a location within a group of filled locations
  -
  -

non-linear probing

Idea: the probe sequence does not involve steps of fixed size.

Example: Quadratic probing is where the probe sequence is calculated as:

$$s_i = (h(k) + c_1 i + c_2 i^2) \bmod m \quad \text{for } i = 0, 1, 2, \ldots$$

But:

double hashing

- In *double hashing* we use a different hash function $h_2(k)$ to calculate the step size.
- The probe sequence is:

$$s_i = (h(k) + i \cdot h'(k)) \bmod m \qquad \text{for } i = 0, 1, 2, \ldots$$

- Note that $h'(k)$ should not be 0 for any k.
- Also, we want to choose h' so that, if $h(k_1) = h(k_2)$ for two keys k_1, k_2, it won't be the case that $h'(k_1) = h'(k_2)$.
- That is, the two hash functions don't cause collisions on the same pairs of keys.

open addressing: complexity

- consider the complexity of `search(k)`
- worst case scenario?
-

Suppose:

- the hash table has m locations
- the hash table contains n elements and $n < m$
- we search for a random key k in the table, with probablility $\frac{1}{n}$

Consider a random probe sequence for k:

- probe sequence is equally likely to be any permutation of $\langle 0, 1, ..., m-1 \rangle$

open addressing: unsuccessful search

Let T be the number of probes performed in an **unsuccessful search**.

Then $E(T) =$

open addressing: unsuccessful search

Let A_i denote the event that the i-th probe occurs and it is to an occupied slot.

Then, $T \geq i$ iff $A_1, A_2, \ldots, A_{i-1}$ all occur.

$\Pr(T \geq i) =$

open addressing: unsuccessful search

$Pr(A_j | A_1 \cap \cdots \cap A_{j-1}) = ?$

Intuition:

- number of elements we have not yet seen:
- number of slots we have not yet seen:

Math: for $1 \leq j \leq m$:

open addressing: unsuccessful search

Now we can calculate the expected value of T, or the average-case complexity of unsuccessful `search(k)`.

$E(T) =$

open addressing: insert

To `insert` a new element:

- perform an unsuccessful search (for an available location)
- insert:

Thus, `insert(k,v)` requires at most $\frac{1}{1-\alpha}$ probes on average.

open addressing: successful search

Let T be the number of probes performed in a **successful search**.

Idea: successful `search(k)` reproduces the same probing sequence as `insert(k,v)`.

If k was the $(i+1)^{st}$ key inserted into the table, then the expected number of probes made is at most

Then, averaging over all n keys in the table:

open addressing: successful search

$E(T) =$

open addressing: successful search

$$E(T) \leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

This is pretty good!

- if the table is half full, the expected number of probes is < 1.387
- if the table is 90% full, this number is < 2.559

open addressing: delete

What about `delete`?

- with closed addressing: easy
  - first do `search` then
  - $\mathcal{O}(1)$ un-link
- with open addressing: two approaches
  - find an existing key to fill the hole
    -
  -
  - over time slows down all operations
  - `delete` is problematic under open addressing