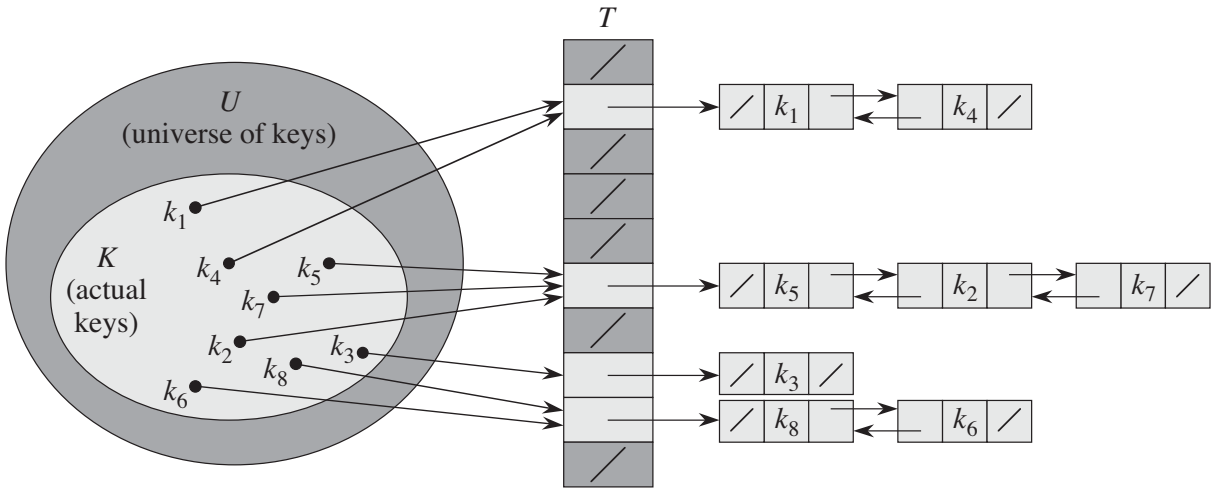
Data Structures and Algorithm Analysis (CSC317)

Hash tables (part2)

Hash table

- We have elements with key and satellite data
- Operations performed: Insert, Delete, Search/lookup
- We ***don't*** maintain order information
- We'll see that all operations **on average O(1)**
- **But worse case can be O(n)**

Review: Collision resolution by chaining



Hash table analyses

- Worst case: all n elements map to one slot (one big linked list…).

  O(n)

Hash table analyses

- Average case: Define:

m = number of slots

n = number elements (keys) in hash table

What was n in previous diagram? (answer: 8)

alpha = load factor: $\alpha = \frac{n}{m}$

Intuitively, alpha is average number elements per linked list.

Hash table analyses

Define:

- Unsuccessful search: new key searched for doesn't exist in hash table (we are searching for a new friend, Sarah, who is not yet in hash table)

- Successful search: key we are searching for already exists in hash table (we are searching for Tom, who we have already stored in hash table)

Hash table analyses

Theorem:

- Assuming uniform hashing, search takes on average O(alpha + 1)
  Here the actual search time is O(alpha) and the added 1 is the constant time to compute a hash function on the key that is being searched

Note: we'll prove for unsuccessful search, but successful search cannot be worse.

Hash table analyses:

Theorem interpretation:

- n=m

  $\Theta(1+1) = \Theta(1)$

- n=2m

  $\Theta(2+1) = \Theta(1)$

- n=m^3

  $\Theta(m^2+1) \neq \Theta(1)$

- Summary: we say constant time on average when n and m similar order, but not generally guaranteed

Hash table analyses

Theorem:

- Intuitively: Search for key k, hash function will map onto slot h(k). We need to search through linked list in the slot mapped to, up until the end of the list (because key is not found = unsuccessful search). For n=2m, on average linked list is length 2. More generally, on average length is alpha, our load factor.

Hash table analyses

Proof with indicator random variables:

- **Consider keys j in hash table (n of them), and key k not in hash table that we are searching for. For each j:**

$$X_j = \begin{array}{l} 1 \text{ if key x hashes to same slot as key j} \\ 0 \text{ otherwise} \end{array}$$

Hash table analyses

Proof with indicator random variables:

- **Consider keys j in hash table (n of them), and key k not in hash table that we are searching for. For each j:**

$$X_j = \begin{array}{l} 1 \text{ if } h(x) = h(j) \text{ (same as before, just as equation)} \\ 0 \text{ otherwise} \end{array}$$

Hash table analyses

Proof with indicator random variables:

- **As with indicator (binary) random variables:**

$$E[X_j] = 1\Pr(X_j = 1) + 0\Pr(X_j = 0) = Pr(X_j = 1)$$

- **By our definition of the random variable:**

$$= \Pr(h(x) = h(j))$$

- **Since we assume uniform hashing:**

$$= \frac{1}{m}$$

Hash table analyses

Proof with indicator random variables:

- **We want to consider key x with regards to every possible key j in hash table:**

$$E[\sum_{j=1}^{n} X_j] =$$

- **Linearity of expectations:**

$$= \sum_{j=1}^{n} E[X_j] = \sum_{j=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$$

Hash table analyses

- We've proved that average search time is O(alpha) for unsuccessful search and uniform hashing. It is O(alpha+1) if we also count the constant time of mapping a hash function for a key

Hash table analyses

Theorem:

- Assuming uniform hashing, **successful** search takes on average O(alpha + 1)
  Here the actual search time is O(alpha) and the added 1 is the constant time to compute a hash function on the key that is being searched

- Intuitively, successful search should take less than unsuccessful. Proof in book with indicator random variables; more involved than before; we won't prove here.

What makes a good hash function?

- Uniform hashing: Each key equally likely to hash to each of m slots (1/m probability)
- Could be hard in practice: we don't always know the distributions of keys we will encounter and want to store – could be biased
- We would like to choose hash functions that do a good job in practice at distributing keys evenly

Example 1 for potential bias

- Key = phone number

Key1 = 305 6215985
Key2 = 305 7621088
Key3 = 786 7447721

- Bad hash function: First three digits of phone number (eg, what if all friends in Miami, and in any case likely to have patterns of regularity)

- Better although could still have patterns: last 3 digits of phone number

Example 2 for potential bias

- Hash function: h(k) = k mod 100
  with k the key

- Keys happen to be all even numbers

- Key1 = 1986
  Key1 mod 100 = 86
  Key2 = 2014
  key2 mod 100 = 14

- Pattern: keys all map to even values.
  All odd slots of hash table unused!

How do we determine the hash function?

We'll discuss:

- Division method – simple, fast, implementation

Division method

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

  m = number of slots
  k = key

- Example: m=20; k=91
  $h(k) = k \bmod m = 11$

Division method: pros

- Any key will indeed map to one of m slots (as we want from a hash function, mapping a key to one of m slots)
- Fast and simple

Division method: cons and important to pay attention to…

- Need to avoid certain values of m to avoid bias (as in the even number example)
- Good to do:
  Often chosen in practice: m that is a prime number and not too close to base of 2 or 10

Resolution to collisions

- We discussed resolution by chaining
- Another approach: open addressing

Open addressing

- Only one object per slot allowed
- As a result: $\alpha = \frac{n}{m} \leq 1$

$(n \leq m)$

(as before alpha is load factor, n number keys in Table, and m number slots)

- No pointers or linked list
- Good approach if space is important
- Deleting keys turns out more tricky than chaining

Open addressing

- Only one object per slot allowed
- As a result: $\alpha = \frac{n}{m} \leq 1$

$(n \leq m)$

(as before alpha is load factor, n number keys in Table, and m number slots)

Main approach: If collision, keep probing hash table until empty slot is found. Hash function is now a sequence…

Open addressing

We'll discuss two types:

- Linear probing
- Double hashing (next class)

Linear probing

When inserting into hash table (also when searching)-

- If hash function results in collision, try the next available slot; if that results in collision try the next slot after that, and so on (if slot 3 is full, try slot 4, then slot 5, then slot 6, and so on)

Linear probing

More formally:

h(k,0) hash function for key k and first probe (= first time try to put in table)

h(k,1) hash function for key k and second probe (= second time try to put in table)

h(k,2) hash function for key k and 3rd probe (= third time try to put in table)

Linear probing

More formally: $h(k,i) = (h(k) + i) \bmod m$

$h(k)$ is a regular hash function, like before

i indicates number slots to skip for probe i
(e.g., attempt to insert in the i time)

$\bmod m$ is to make sure it is mapped overall
to one of m slots in the hash table

Linear probing

Example:

$$h(k,i) = (h(k) + i) \bmod m$$
$$h(k) = k \bmod 7$$

- Insert keys 2; 6; 9; 16
- On the board …

(keys 9 and 16 should result in collision and choosing next available slots)

Linear probing

- Pro: easy to implement

- Con: can result in primary clustering
  keys can cluster by taking adjacent slots in hash table, since each time searching for next available slot when there is collision

  = longer search time

Linear probing

- Pro: easy to implement

- Con: can result in primary clustering
  keys can cluster by taking adjacent slots in hash table, since each time searching for next available slot when there is collision

  = longer search time

What about if we insert 2 or 3 slots away instead of 1 slot away?

Linear probing

What about if we insert 2 or 3 slots away instead of 1 slot away?

Answer: still have problem that if two keys initially mapped to same hash slot, they have identical probe sequences, since offset of next slot to check doesn't depend on key

Linear probing

What about if we insert 2 or 3 slots away instead of 1 slot away?

Answer: still have problem that if two keys initially mapped to same hash slot, they have identical probe sequences, since offset of next slot to check doesn't depend on key

What to do? Make offset determined by another key function – double hashing!