Lecture 26: More Dictionaries & Hashing

CS 62

Fall 2017

Kim Bruce & Alexandra Papoutsaki

Naïve Version

- Warning: this code is simplified!

```
public class Map<K, V> {
  protected V[] entries;

  public V get(K key) {
    int index = key.hashCode() % entries.length;
    return entries[index];
  }
  public void put(K key, V value) {
    int index = key.hashCode() % entries.length;
    entries[index] = value;
  }
}
```

Hash Collisions

- `k1.hashCode() == k2.hashCode()` but `k1 != k2`
  - May also be caused by the modulus operation
- This is inevitable (e.g., the birthday paradox)
- A “good” hash function rarely collides

Avoiding Collisions

- Two main strategies

1) Open addressing (closed hashing):
    - Linear probing
    - Quadratic probing
    - Double probing
    - Cuckoo hashing

2) Closed Addressing(open or external hashing/bucketing):
    - Separate chaining

Linear Probing

- If we collide, check next entry until one is empty
- Deletion is complicated
- Can only hold `entries.length` items
- Resizing the table requires rehashing everything
- Suffers from primary clustering

Quadratic Probing

- $h(k,i) = (h(k) + c_1 i + \ c_2 i^2)\ (mod\ n), c_2 \neq 0$
- If $c_2 = 0$ then degrades to linear probing
- E.g., $h(k,i) = (h(k) + \ i^2)\ (mod\ n)$, then $h(k), h(k) + 1, h(k) + 4, \ldots$
- Can result in case where don't try all slots
  - E.g., $n = \ 5$, and start with $h(k) \ = \ 1$.
  - Rehashings give $2, 0, 0, 2, 1, 2, 0, 0, \ldots$
  - The slots 3 and 4 will never be examined to see if they have room
- Secondary Clustering

Double Hashing

- Use second hash function on key to determine delta (interval) for next try
- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \ (mod\ n)$,
- E.g., $h_2(k) = (k\ mod\ (n\ - 2)) + 1$
- Helps with primary and secondary clustering

- Example:
  - Suppose $h_1(n) = n \% 5$
  - Then $h_1(1) = h_1(6) = h_1(11)$
  - However, $h_2(1) = 2, h_2(6) = 1, h_2(11) = 3$

Separate Chaining

- Turn each entry into a linked list (or array, etc.)
- On collision add to the bucket
- Searching list is fast if lists are small
- Deletion is simple
- Can hold more than `entries.length` items easily

Load Factor

- Performance depends on *load factor*
- Load factor is $\alpha = \frac{n}{N}$ where n = items in table and N = size of table
- Higher load factor → more collisions → slow
- Can be > 1 for external chaining
- For open addressing usually want to ensure $\alpha < 0.75$
  - Generally $\alpha > 0.75$ means resize the table (& rehash everything)

Performance

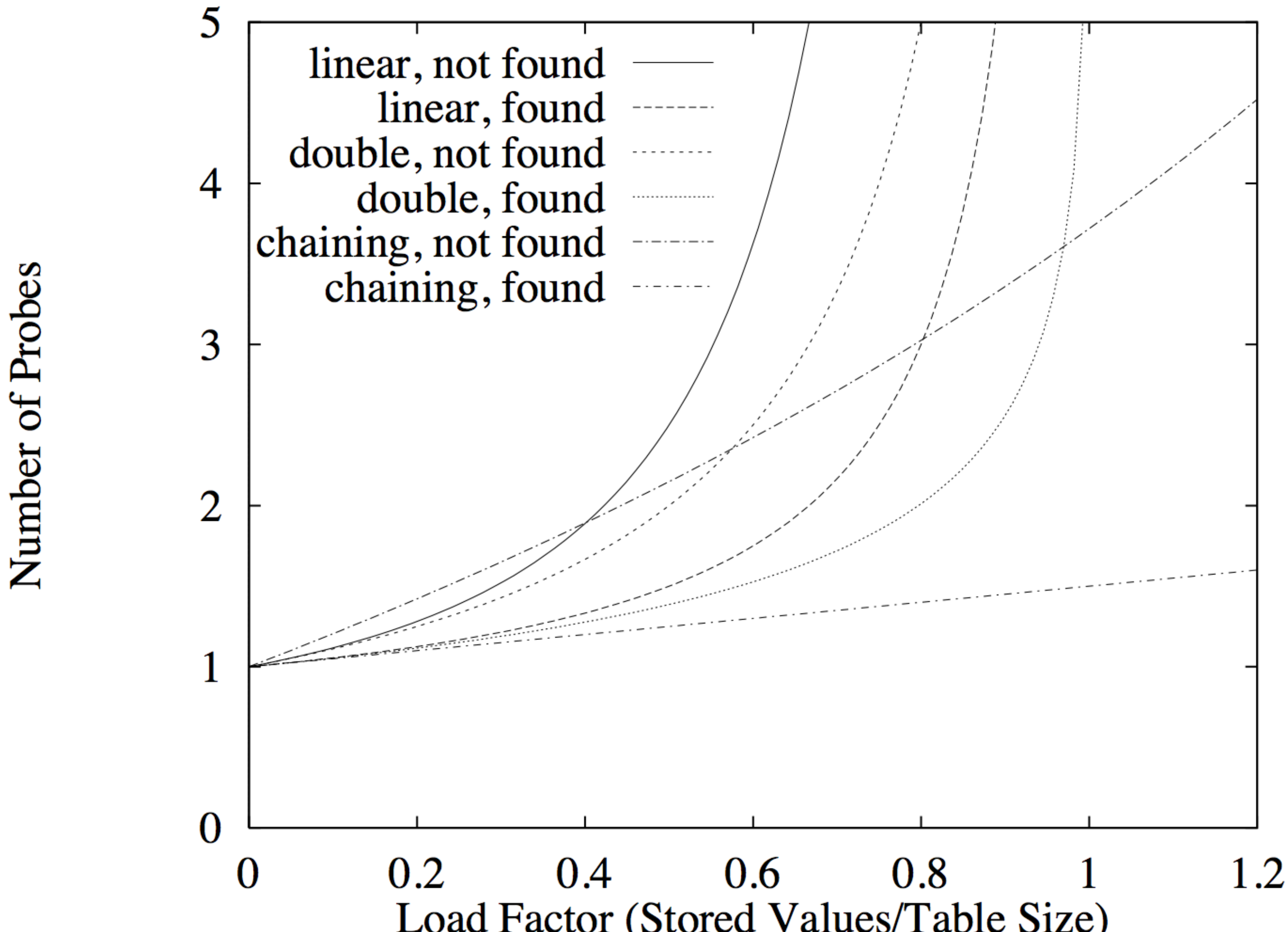
Strategy	Unsuccessful	Successful
Linear Probing	$1/2\ (1 +\ 1/(1-a)^2)$	$1/2\ (1 +\ 1/(1-a))$
Double Probing	$1/(1-a)$	$-(1/a)/log(1-a)$
External Chaining	$a + e^{-a}$	$1 + 1/2a$

Entries represent number of comparisons needed to find a specific element or demonstrate that it is not in the hash table

Performance for $a = .9$

Strategy	Unsuccessful	Successful
Linear Probing	55	5.5
Double Probing	10	~ 4
External Chaining	3	1.45

Entries represent number of comparisons needed to find a specific element or demonstrate that it is not in the hash table



Number of Probes
Load Factor (Stored Values/Table Size)
linear, not found
linear, found
double, not found
double, found
chaining, not found
chaining, found
0
1
2
3
4
5
0
0.2
0.4
0.6
0.8
1
1.2

Space requirements

- Open addressing: TableSize +n*objectsize
- External chaining: TableSize +(n*objectsize+1)

- Rule of thumb:
  - Small elements, small load factor: open addressing
  - Large elements, large load factor: external chaining