CSE 332: Data Structures & Parallelism

Lecture 11:More Hashing

Winston Jodjana

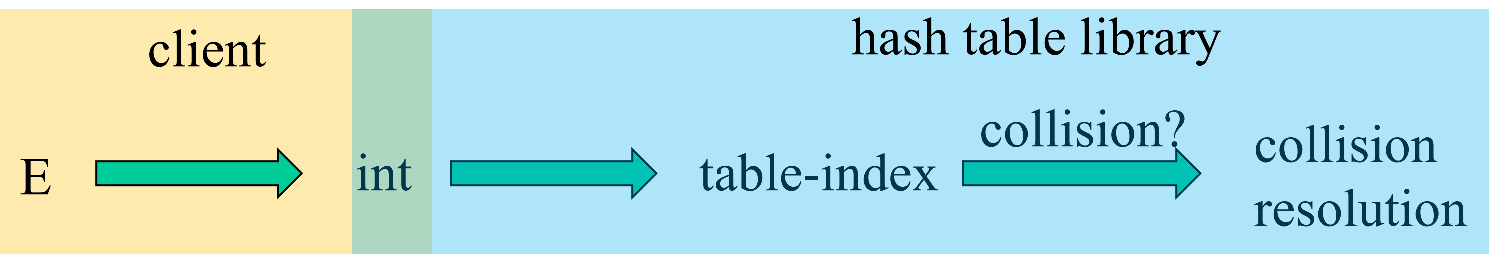
Spring 2023

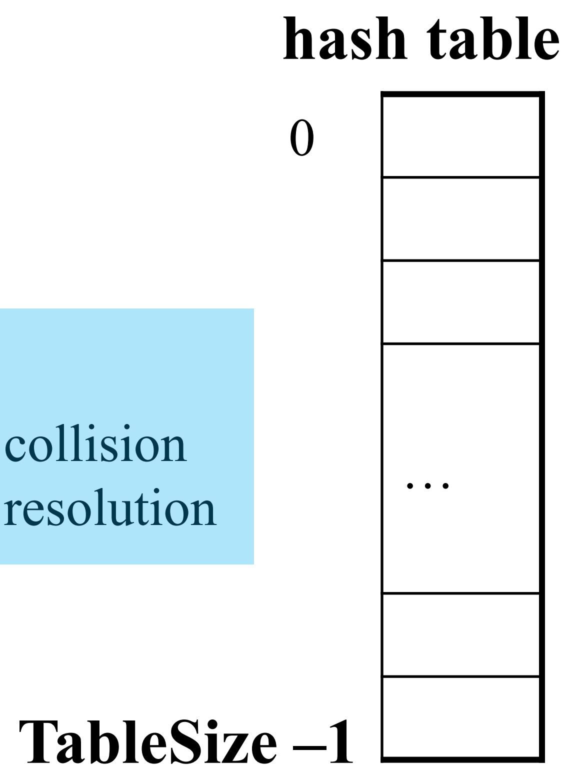
Today

- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Rehashing

Hash Tables: Review

- Aim for constant-time (i.e., *O*(1)) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size
  - But growable as we'll see





Hashing Choices

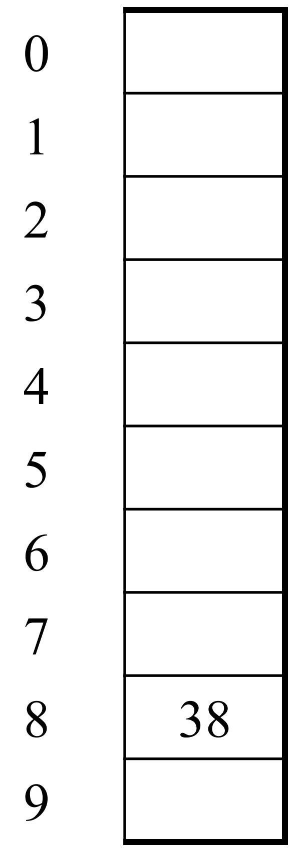
1. Choose a Hash function
   - Fast
   - Even spread
2. Choose TableSize
   - Prime Numbers
3. Choose a Collision Resolution Strategy from these:
   - Separate Chaining
   - **Open Addressing**
     - Linear Probing
     - Quadratic Probing
     - Double Hashing

- Other issues to consider:
  - What to do when the hash table gets “too full”?

Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10



Open addressing

Linear probing is *one example* of open addressing

In general, open addressing means resolving collisions by trying a sequence of other positions in the table.

Trying the *next* spot is called probing

- We just did linear probing:
  - **i**th probe: **`(h(key) + i) % TableSize`**
- In general have some probe function **`f`** and :
  - **i**th probe: **`(h(key) + f(i)) % TableSize`**

Open addressing does poorly with high load factor λ

- So want larger tables
- Too many probes means no more $O(1)$

Questions: Open Addressing: Linear Probing

How should **`find`** work? If key is in table? If not there?

Worst case scenario for **`find`**?

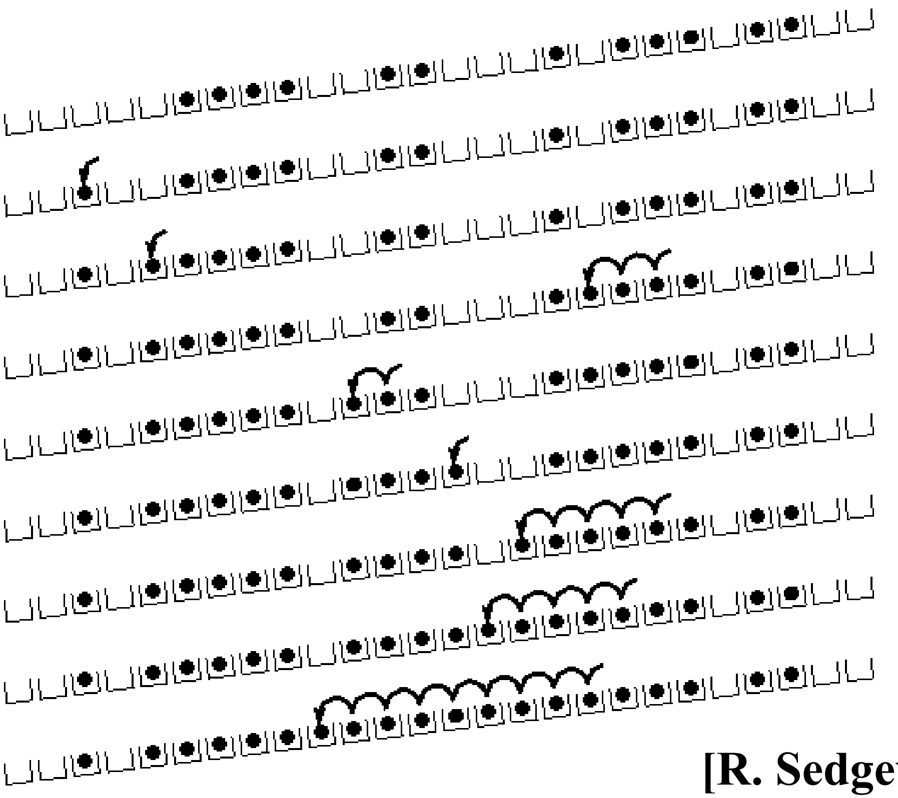
How should we implement **`delete`**?

How does **open addressing with linear probing** compare to **separate chaining**?

Primary Clustering

It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (a good thing)

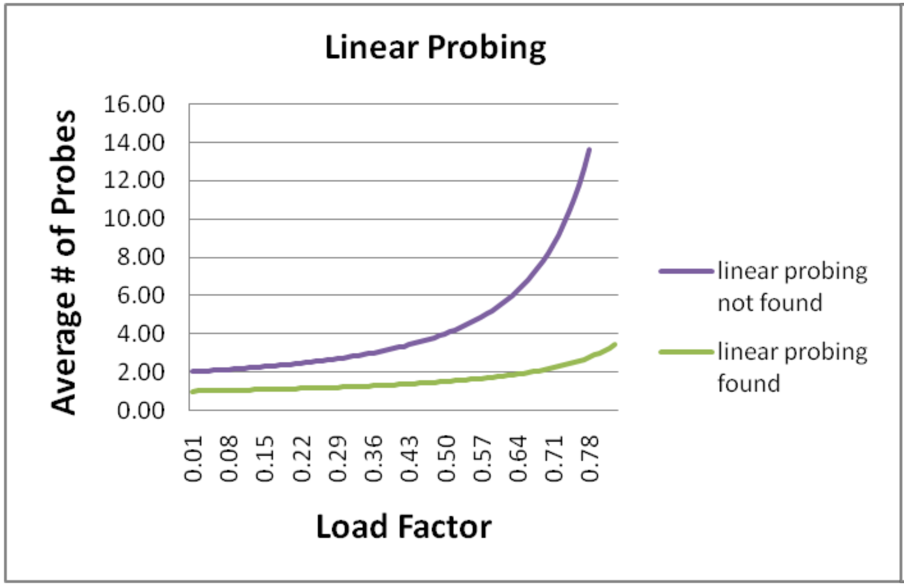
- Tends to produce *clusters*, which lead to long probe sequences
- Called primary clustering
- Saw the start of a cluster in our linear probing example

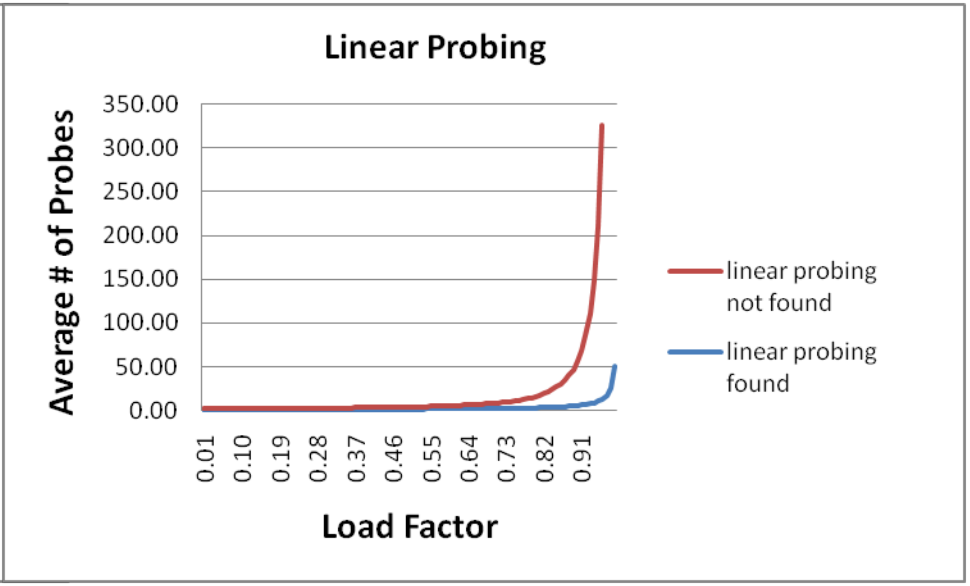


[R. Sedgewick]

Analysis in chart form

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)





- By comparison, separate chaining performance is linear in λ and has no trouble with $\lambda>1$

Open Addressing: Linear probing

`(h(key) + f(i)) % TableSize`

- For linear probing:

  `f(i) = i`

- So probe sequence is:
  - 0^{th} probe: `h(key) % TableSize`
  - 1^{st} probe: `(h(key) + 1) % TableSize`
  - 2^{nd} probe: `(h(key) + 2) % TableSize`
  - 3^{rd} probe: `(h(key) + 3) % TableSize`
  - …
  - i^{th} probe: `(h(key) + i) % TableSize`

Open Addressing: Quadratic probing

- We can avoid primary clustering by changing the probe function…

  (**`h(key) + f(i)) % TableSize`**

  - For quadratic probing:

    `f(i) = i`^{2}

  - So probe sequence is:
    - 0th probe: **`h(key) % TableSize`**
    - 1st probe: **`(h(key) + 1) % TableSize`**
    - 2nd probe: **`(h(key) + 4) % TableSize`**
    - 3rd probe: **`(h(key) + 9) % TableSize`**
    - …
    - ith probe: **`(h(key) + i`**^{2}**`) % TableSize`**

- Intuition: Probes quickly “leave the neighborhood” ith probe: (`h(key) + i`^{2}`) % TableSize`

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize=10
Insert:
89
18
49
58
79

ith probe: (`h(key)` + `i`^{2}`) % TableSize`

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76 (76 % 7 = 6)

40 (40 % 7 = 5)

48 (48 % 7 = 6)

5 (5 % 7 = 5)

55 (55 % 7 = 6)

47 (47 % 7 = 5)

(47 + 1) % 7 = 6 collision!

(47 + 4) % 7 = 2 collision!

(47 + 9) % 7 = 0 collision!

(47 + 16) % 7 = 0 collision!

(47 + 25) % 7 = 2 collision!

(47 + 36) % 7 = 6 collision!

(47 + 49) % 7 = 5 collision!

Will we ever get a 1 or 4?!?

From bad news to good news

Bad News:

- After **`TableSize`** quadratic probes, we cycle through the same indices

Good News:

- If **`TableSize`** is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
- So: If you keep $\lambda < ½$ and **`TableSize`** is *prime*, no need to detect cycles
- Proof posted in **`lecture11.txt`** (slightly less detailed proof in textbook)

For prime **`TableSize`** and $0 \leq i,j \leq \texttt{TableSize/2}$ where $i \neq j$,

$$(h(key) + i^2)\ \%\ TableSize \neq (h(key) + j^2)\ \%\ TableSize$$

That is, if **`TableSize`** is prime, the first **`TableSize`**/2 quadratic probes map to different locations (and one of those will be empty if the table is < half full).

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering:
  As we resolve collisions we are not merely growing "big blobs" by adding one more item to the end of a cluster, we are looking i^2 locations away, for the next possible spot.

- But quadratic probing does not help resolve collisions between keys that initially hash *to the same **index***
  - Any 2 keys that initially hash to the same index **will have the same series of moves after that** looking for any empty spot
  - Called secondary clustering

- Can avoid secondary clustering with *a probe function that depends on the key*: double hashing…

Open Addressing: Double hashing

Idea: Given two good hash functions *h* and *g*, and two different keys *k1* and *k2*, it is very unlikely that: **`h(k1)==h(k2)`** and **`g(k1)==g(k2)`**

`(h(key) + f(i)) % TableSize`

- For double hashing:

  `f(i) = i*g(key)`

- So probe sequence is:
  - 0th probe: **`h(key) % TableSize`**
  - 1st probe: **`(h(key) + g(key)) % TableSize`**
  - 2nd probe: **`(h(key) + 2*g(key)) % TableSize`**
  - 3rd probe: **`(h(key) + 3*g(key)) % TableSize`**
  - …
  - ith probe: **`(h(key) + i*g(key)) % TableSize`**

- Detail: Make sure **`g(key)`** can't be **`0`** ith probe: `(h(key) + i*g(key)) % TableSize`

Open Addressing: Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

T = 10 (TableSize)

Hash Functions:

h(key) = key

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

ith probe: `(h(key) + i*g(key)) % TableSize`

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	147

T = 10 (TableSize)

<u>Hash Functions</u>:

h(key) = key

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147 → g(147) = 1 + 14 mod 9 = 6

43 → g(43) = 1 + 4 mod 9 = 5

We have a problem:

3 + 0 = 3   3 + **5** = **8**   **3** + 10 = **13**

3 + **15** = **18**   3 + 20 = 23

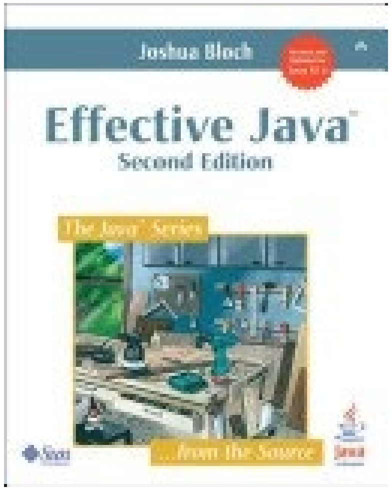
Where are we?

- Separate Chaining is easy
  - **`find, insert, delete`** proportional to load factor on average if using unsorted linked list nodes
  - If using another data structure for buckets (e.g. AVL tree) , runtime is proportional to runtime for that structure.
- Open addressing uses probing, has clustering issues as table fills Why use it:
  - Less memory allocation?
    - Some run-time overhead for allocating linked list (or whatever) nodes; open addressing could be faster
  - Easier data representation?
- Now:
  - Growing the table when it gets too full (aka “rehashing”)
  - Relation between hashing/comparing and connection to Java

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over
- With **separate chaining,** we get to decide what "too full" means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
- For **open addressing**, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except, uhm, that won't be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you probably won't grow more than 20-30 times, and then calculate after that

A Generally Good hashCode()




```
int result = 17; // start at a prime

foreach field f
   int fieldHashcode =
      boolean: (f ? 1: 0)
      byte, char, short, int: (int) f
      long: (int) (f ^ (f >>> 32))
      float: Float.floatToIntBits(f)
      double: Double.doubleToLongBits(f), then above
      Object: object.hashCode( )

      result = 31 * result + fieldHashcode;
return result;
```

Final word on hashing

- The hash table is one of the most important data structures
  - Efficient find, insert, and delete
  - Operations based on sorted order are not so efficient!
  - Useful in many, many real-world applications
  - Popular topic for job interview questions
- Important to use a good hash function
  - Good distribution, Uses enough of key's components
  - Not overly expensive to calculate (bit shifts good!)
- Important to keep hash table at a good size
  - Prime #
  - Preferable λ depends on type of table
- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums