CSE 332
Data Structures & Parallelism

Hashing 2

Melissa Winstanley
Spring 2024

Today

Last time:

- Hash tables
- Hash functions
- Separate chaining

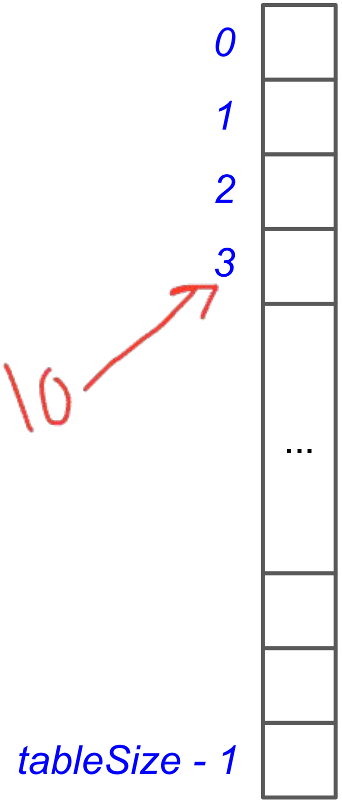
Today:

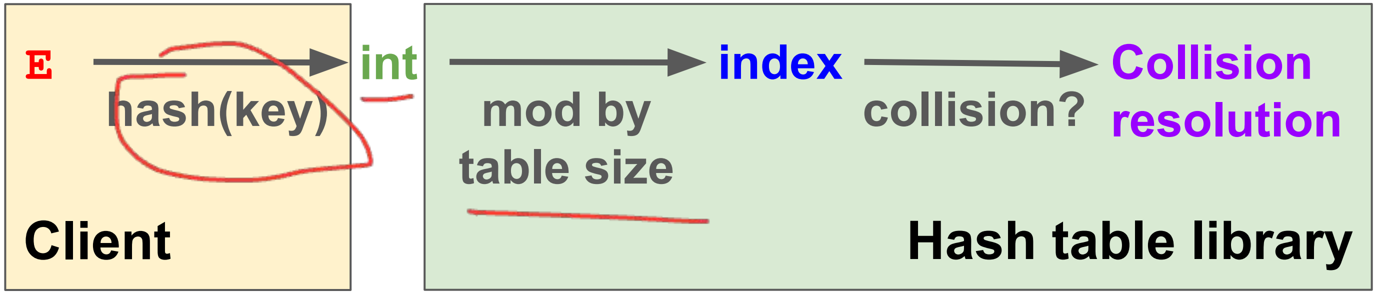
- Open Addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
- Rehashing / resizing

Hash Tables: Review

- Dictionary implementation
- Aim for constant-time (i.e., $\boxed{O(1)}$) **`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

3 % 10





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"? rehash

Open Addressing: Linear Probing (simplest)

- 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

0	8	8_2 109_1 10_0
1	109	109_2 10_1
2	10	10_2
3		
4		
5		
6		
7		
8	38	8_0
9	19	8_1 109_0

T=10

Open addressing

Linear probing is one example of **open addressing**

- 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`:
  - `i`th probe: `(h(key) + f(i,key)) % TableSize`

Open addressing does poorly with high load factor λ

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

$\lambda = 0.5$

Questions: Open Addressing: Linear Probing

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

find(10)

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

O(n) 48

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

tombstone/lazy 109

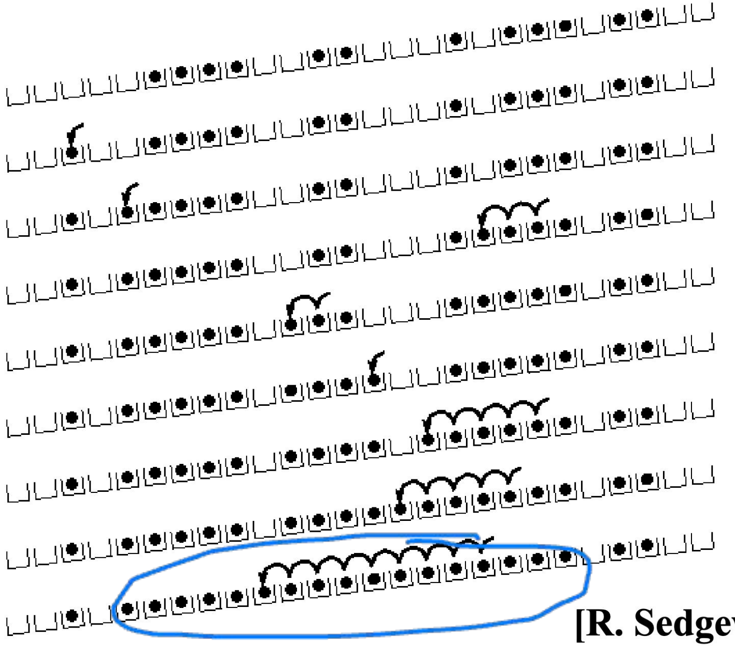
How does **open addressing with linear probing** compare to **separate chaining**?

0	8
1	48 ~~109~~
2	10
3	
4	
5	
6	
7	
8	38
9	19

Primary Clustering

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

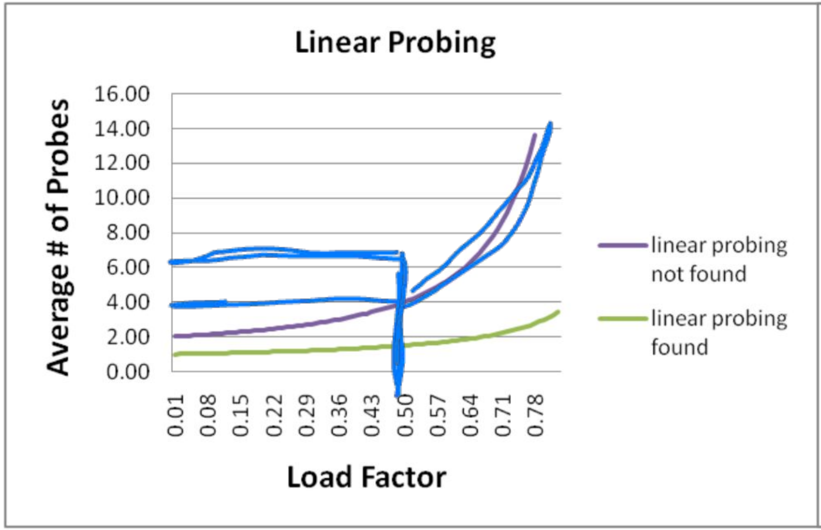
- Tends to produce clusters, which lead to long probe sequences
- Called primary clustering
- Saw the start of a cluster in our linear probing example

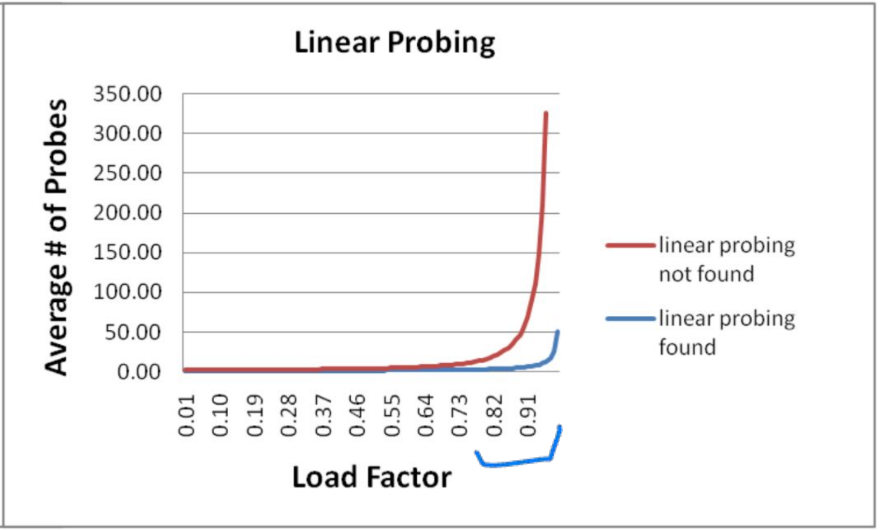


[R. Sedgewick]

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 λ>1





Open Addressing: Linear probing

$$\texttt{index}_i = (\texttt{h(key)} + \texttt{f(i, key)}) \ \% \ \texttt{TableSize}$$

- For linear probing:

$$\texttt{f(i, key)} = \texttt{i}$$

- 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…

  `index`$_i$ `= (h(key) + f(i, key)) % TableSize`

- For quadratic probing:

  `f(i, key) =` $\boxed{i^2}$

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

- Intuition: Probes quickly “leave the neighborhood”

i^{th} probe: **(h(key) + i²) % TableSize**

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

49_1 79_1

58_2 79_2

58_0

49_0 58_1 79_0

TableSize=10

Insert:

~~89~~

~~18~~

~~49~~

~~58~~

79

i^{th} probe: `(h(key) + i²) % TableSize`

Another Quadratic Probing Example

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

48 5 55

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)

i^{th} probe: `(h(key) + i²) % TableSize`

Another Quadratic Probing Example

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

Insert 47

(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 a 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 λ < ½, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
- So: If you keep λ < ½ and **`TableSize`** is *prime*, no need to detect cycles
- Proof posted in **`lecture12.txt`** (slightly less detailed proof in textbook)
  For prime **`TableSize`** and **`0 ≤ i,j ≤ TableSize/2`** where **`i ≠ j`**,
  `(h(key) + i2) % TableSize ≠ (h(key) + j2) % 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).

Primary 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 $\mathtt{i}^2$ 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)`

$$\texttt{index}_i = \texttt{(h(key) + f(i, key)) \% TableSize}$$

- For double hashing:

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

- So probe sequence is:
  - 0^{th} probe: `h(key) % TableSize`
  - 1^{st} probe: `(h(key) + g(key)) % TableSize`
  - 2^{nd} probe: `(h(key) + 2*g(key)) % TableSize`
  - 3^{rd} probe: `(h(key) + 3*g(key)) % TableSize`
  - …
  - i^{th} probe: `(h(key) + i*g(key)) % TableSize`
- Detail: Make sure `g(key)` can't be 0

i^{th} probe: **`(h(key) + i*g(key)) % TableSize`**

Double Hashing Example

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

33, 43

147

TableSize T=10

`h(key) = key`

`g(key) = 1 + (key/T) % (T-1)`

Insert:

~~13~~
~~28~~
~~33~~
147 --> g(147) = 1 + 14%9 = 6
43 --> $\boxed{\text{g(43) = 1 + 4\%9 = 5}}$

1+(3 % 9) = 4

Oh no!

Where are we?

- <u>Separate Chaining</u> is easy
  - **`find`**, **`insert`**, **`delete`** proportional to load factor on average if using unsorted linked list nodes

Java

  - If using another data structure for buckets (e.g. AVL tree), runtime is proportional to runtime for that structure.
- <u>Open addressing</u> uses probing, has clustering issues as table fills. Why use it:
  - Less memory allocation?

Python

    - 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 (resizing)

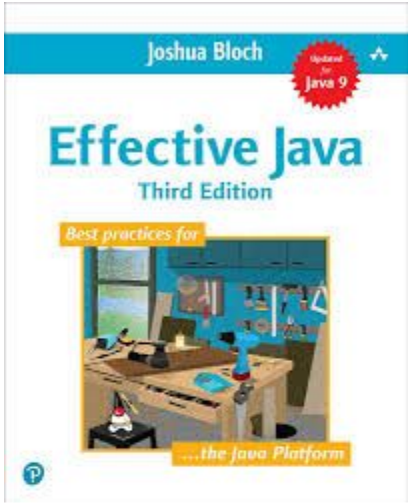
- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over
- With **separate chaining**, we get to decide what “too full” means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?

insert()

- 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
- How do we actually do the resizing? We can’t copy elements into same index!

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;
```



Even better? Use randomization (chosen on startup)

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