

CSE 332: Data Structures & Parallelism

Lecture 11: More Hashing

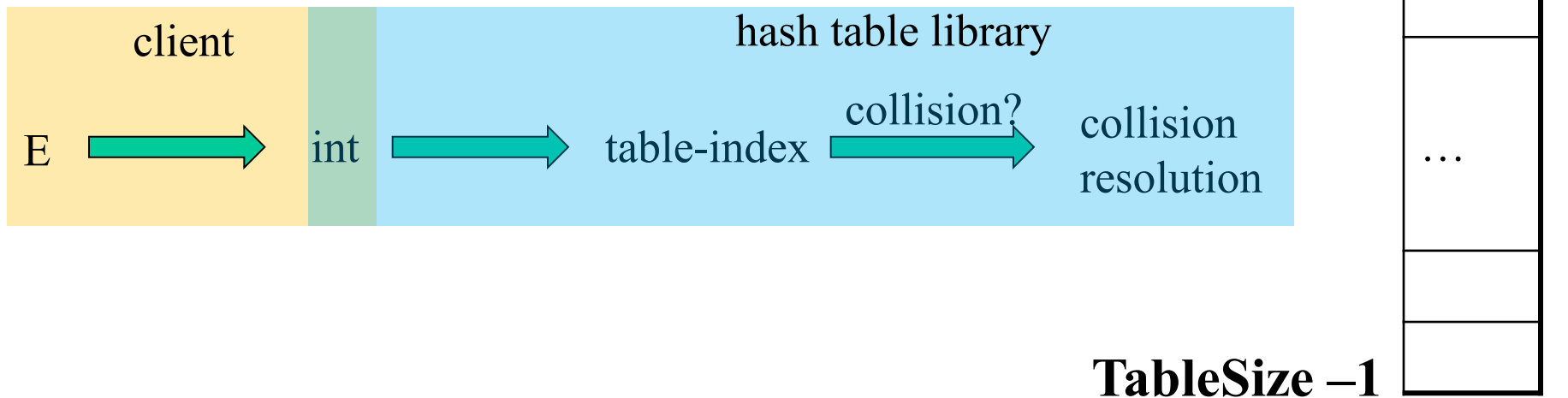
Ruth Anderson
Winter 2025

Today

- Dictionaries
 - Hashing

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
 2. Choose TableSize
 3. Choose a Collision Resolution Strategy from these:
 - Separate Chaining
 - Open Addressing
 - Linear Probing
 - Quadratic Probing
 - Double Hashing
- Other issues to consider:
 - Deletion?
 - 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 $h(\text{key})$ is already full,
 - try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,
 - try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,
 - try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...
- Example: insert 38, 19, 8, 109, 10

0	
1	
2	
3	
4	
5	
6	
7	
8	38
9	

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: $(h(\text{key}) + i) \% \text{TableSize}$

– In general have some **probe function f** and :

• i^{th} probe: $(h(\text{key}) + f(i)) \% \text{TableSize}$

Open addressing does poorly with high load factor λ

– So want larger tables

– Too many probes means no more $O(1)$

Terminology

We and the book use the terms

- “chaining” or “separate chaining”
- “open addressing”

Very confusingly,

- “open hashing” is a synonym for “chaining”
- “closed hashing” is a synonym for “open addressing”

Questions: Open Addressing: Linear Probing

How should `find` work? If value 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**?

Open Addressing: Other Operations

insert finds an open table position using a probe function

What about **find**?

- Must use same probe function to “retrace the trail” for the data
- Unsuccessful search when reach empty position

What about **delete**?

- **Must** use “lazy” deletion. Why?
 - Marker indicates “no data here, but don’t stop probing”

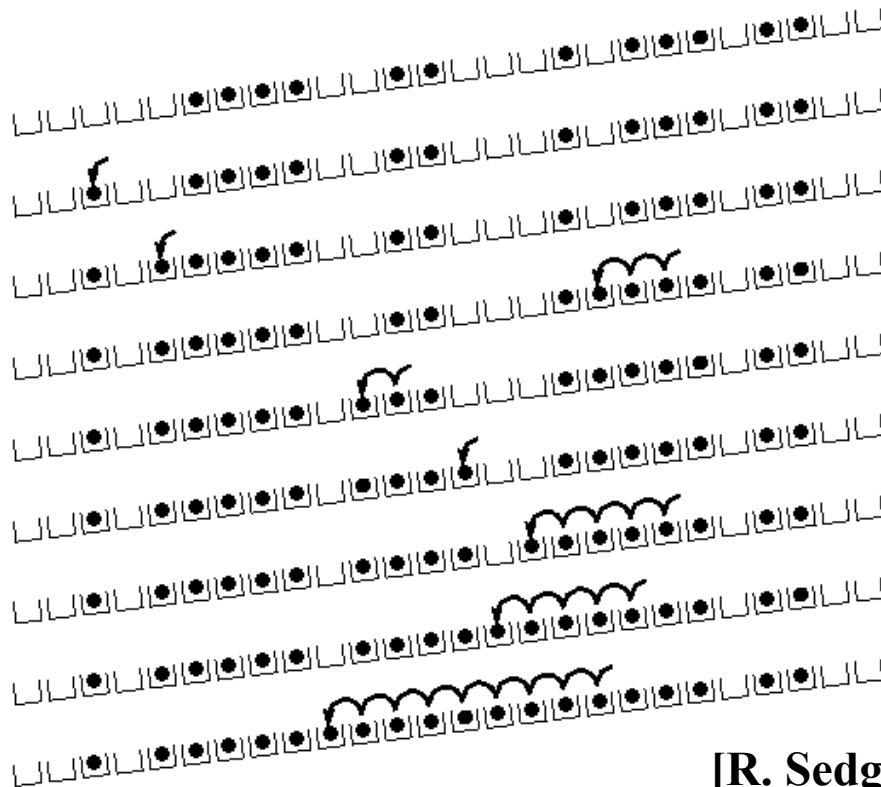
10	x	/	23	/	/	16	x	26
----	---	---	----	---	---	----	---	----

- As with lazy deletion on other data structures, on insert, spots marked “deleted” can be filled in.
- Note: **delete** with chaining is just calling delete on the bucket (e.g. linked list)

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



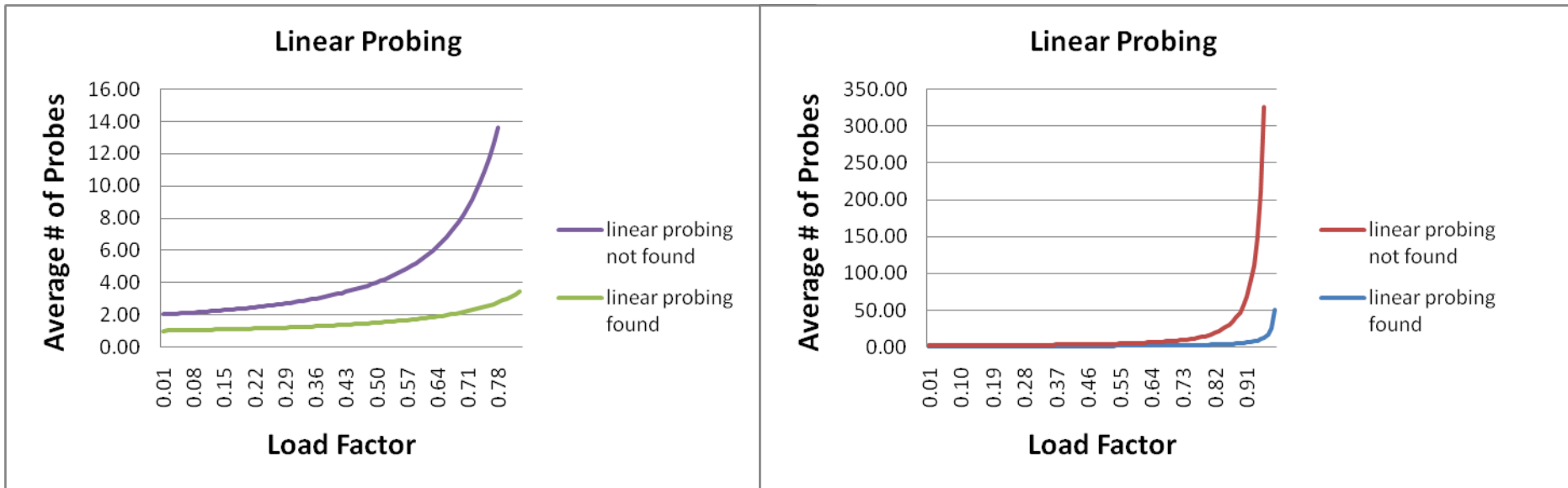
[R. Sedgewick]

Analysis of Linear Probing

- **Trivial fact:** For any $\lambda < 1$, linear probing will find an empty slot
 - It is “safe” in this sense: no infinite loop unless table is full
- **Non-trivial facts** we won't prove:
Average # of probes given λ (in the limit as **TableSize** $\rightarrow \infty$)
 - Unsuccessful search: $\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2} \right)$
 - Successful search: $\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)} \right)$
- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)

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

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

- For linear probing:

$$f(i) = i$$

- So probe sequence is:

- 0th probe: $h(\text{key}) \% \text{TableSize}$
- 1st probe: $(h(\text{key}) + 1) \% \text{TableSize}$
- 2nd probe: $(h(\text{key}) + 2) \% \text{TableSize}$
- 3rd probe: $(h(\text{key}) + 3) \% \text{TableSize}$
- ...
- i^{th} probe: $(h(\text{key}) + i) \% \text{TableSize}$

Open Addressing: Quadratic probing

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

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

- For quadratic probing:

$$f(i) = i^2$$

- So probe sequence is:

- 0th probe: $h(\text{key}) \% \text{TableSize}$
- 1st probe: $(h(\text{key}) + 1) \% \text{TableSize}$
- 2nd probe: $(h(\text{key}) + 4) \% \text{TableSize}$
- 3rd probe: $(h(\text{key}) + 9) \% \text{TableSize}$
- ...
- i^{th} probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

- Intuition: Probes quickly “leave the neighborhood”

ith probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize=10

Insert:

89

18

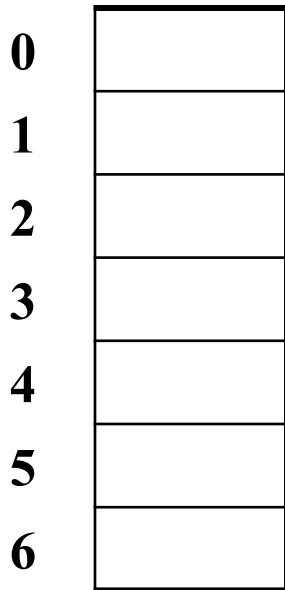
49

58

79

ith probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

Another Quadratic Probing Example



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)**

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 $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in at most `TableSize/2` probes
- So: If you keep $\lambda < \frac{1}{2}$ and `TableSize` is *prime*, no need to detect cycles
- Proof posted in `lecture11.txt` (slightly less detailed proof in textbook)

For prime `TableSize` and $0 \leq i, j \leq \text{TableSize}/2$ where $i \neq j$,
 $(h(\text{key}) + i^2) \% \text{TableSize} \neq (h(\text{key}) + j^2) \% \text{TableSize}$

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).

Quadratic Probing:

Success guarantee for $\lambda < 1/2$

First size/2 probes distinct.
If $<$ half full, one is empty.

- If size is prime and $\lambda < 1/2$, then quadratic probing will find an empty slot in size/2 probes or fewer.
 - show for all $0 \leq i, j \leq \text{size}/2$ and $i \neq j$ (ith probe and jth probe map to distinct locations)

$$(h(x) + i^2) \bmod \text{size} \neq (h(x) + j^2) \bmod \text{size}$$

- by contradiction: suppose that for some $i \neq j$:

$$(h(x) + i^2) \bmod \text{size} = (h(x) + j^2) \bmod \text{size}$$

$$\Rightarrow i^2 \bmod \text{size} = j^2 \bmod \text{size}$$

$$\Rightarrow (i^2 - j^2) \bmod \text{size} = 0$$

$$\Rightarrow [(i + j)(i - j)] \bmod \text{size} = 0$$

BUT size does not divide $(i - j)$ or $(i + j)$

How can $i + j = 0$ or $i + j = \text{size}$ when:

$$i \neq j \quad \text{and} \quad 0 \leq i, j \leq \text{size}/2?$$

Similarly how can $i - j = 0$ or $i - j = \text{size}$?

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 k_1 and k_2 , it is very unlikely that: $h(k_1) == h(k_2)$ and $g(k_1) == g(k_2)$

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

– For double hashing:

$$f(i) = i * g(\text{key})$$

– So probe sequence is:

- 0th probe: $h(\text{key}) \% \text{TableSize}$
- 1st probe: $(h(\text{key}) + g(\text{key})) \% \text{TableSize}$
- 2nd probe: $(h(\text{key}) + 2 * g(\text{key})) \% \text{TableSize}$
- 3rd probe: $(h(\text{key}) + 3 * g(\text{key})) \% \text{TableSize}$
- ...
- i^{th} probe: $(h(\text{key}) + i * g(\text{key})) \% \text{TableSize}$

• Detail: Make sure $g(\text{key})$ can't be 0

ith probe: $(h(\text{key}) + i * g(\text{key})) \% \text{TableSize}$

Open Addressing: Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

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

13

28

33

147

43

Double-hashing analysis

Intuition: Since each probe is “jumping” by $g(\text{key})$ each time, we “leave the neighborhood” *and* “go different places from other initial collisions”

But, as in quadratic probing, we could still have a problem where we are not "safe" due to an infinite loop despite room in table:

- No guarantee that $i * g(\text{key})$ will let us try all/most indices
- It is known that this cannot happen in at least one case:

For primes p and q such that $2 < q < p$

$$h(\text{key}) = \text{key} \% p$$

$$g(\text{key}) = q - (\text{key} \% q)$$

More double-hashing facts

- Assume “uniform hashing”
 - Means probability of $g(\text{key1}) \% p == g(\text{key2}) \% p$ is $1/p$
- Non-trivial facts we won't prove:
Average # of probes given λ (in the limit as **TableSize** $\rightarrow \infty$)
 - Unsuccessful search (intuitive): $\frac{1}{1-\lambda}$
 - Successful search (less intuitive): $\frac{1}{\lambda} \log_e \left(\frac{1}{1-\lambda} \right)$
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

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

More on rehashing

- What if we copy all data to the same indices in the new table?
 - Will not work; we calculated the index based on **TableSize**
- Go through table, do standard insert for each into new table
 - Iterate over old table: $O(n)$
 - n inserts / calls to the hash function: $n \cdot O(1) = O(n)$
- Is there some way to avoid all those hash function calls?
 - Space/time tradeoff: Could store **$h(\text{key})$** with each data item
 - Growing the table is still $O(n)$; saving **$h(\text{key})$** only helps by a constant factor

Hashing and comparing

- Our use of int key can lead to us overlooking a critical detail:
 - We initially *hash* **E** to get a table index
 - While chaining or probing we need to determine if this is the **E** that I am looking for. Just need equality testing.
- So a hash table needs a hash function and a equality testing
 - In the Java library each object has an **equals** method and a **hashCode** method

```
class Object {  
    boolean equals(Object o) {...}  
    int hashCode() {...}  
    ...  
}
```

Equal objects must hash the same

- The Java library (and your hash tables) make a very important assumption that clients must satisfy...
- Object-oriented way of saying it:
 If `a.equals(b)`, then we must require
 `a.hashCode() == b.hashCode()`
- Function object way of saying it:
 If `c.compare(a,b) == 0`, then we must require
 `h.hash(a) == h.hash(b)`
- If you ever override equals
 - You need to override hashCode also in a consistent way
 - See CoreJava book, Chapter 5 for other "gotchas" with equals

By the way: comparison has rules too

We have not emphasized important “rules” about comparison for:

- All our dictionaries
- Sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all **a**, **b**, and **c**,

- If **compare(a,b) < 0**, then **compare(b,a) > 0**
- If **compare(a,b) == 0**, then **compare(b,a) == 0**
- If **compare(a,b) < 0** and **compare(b,c) < 0**,
then **compare(a,c) < 0**

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