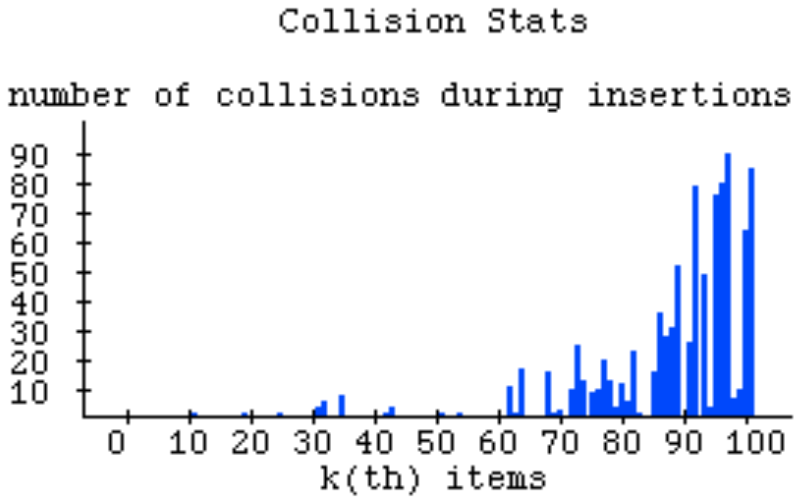
CSSE 230
Hash Table Analysis

When do hash tables degrade in performance?
How should we set the maximum load factor?

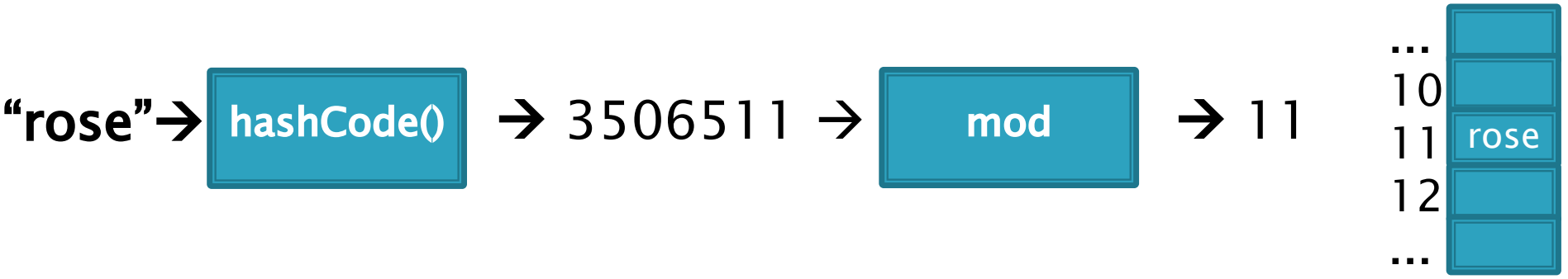


Motivation for Analysis

"It is especially important to know the average behavior of a hashing method, because we are committed to trusting in the laws of probability whenever we hash. The worst case of these algorithms is almost unthinkably bad, so we need to be reassured that the average is very good."

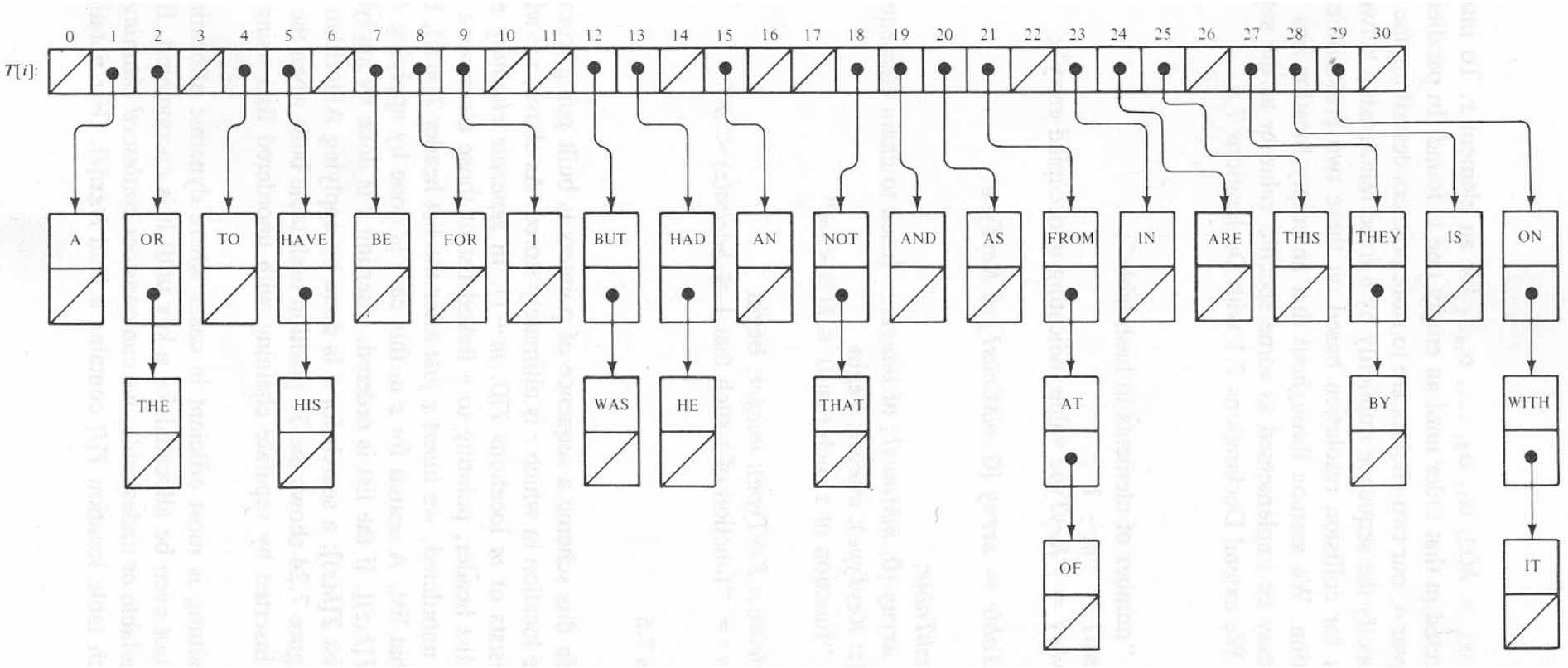
—Donald Knuth,
The Art of Computer Programming, Vol 3:
Searching and Sorting

Hash Tables Overview



- [Last time] Designing appropriate hashCode functions
  - Should "scatter" similar objects
  - E.g., for Strings: x = 31x + y pattern
    - "Interpret string as a number base 31"
- [Continued today] Collision resolution: two basic strategies
  - Separate chaining
  - Probing (open addressing)

Separate chaining: an array of linked lists

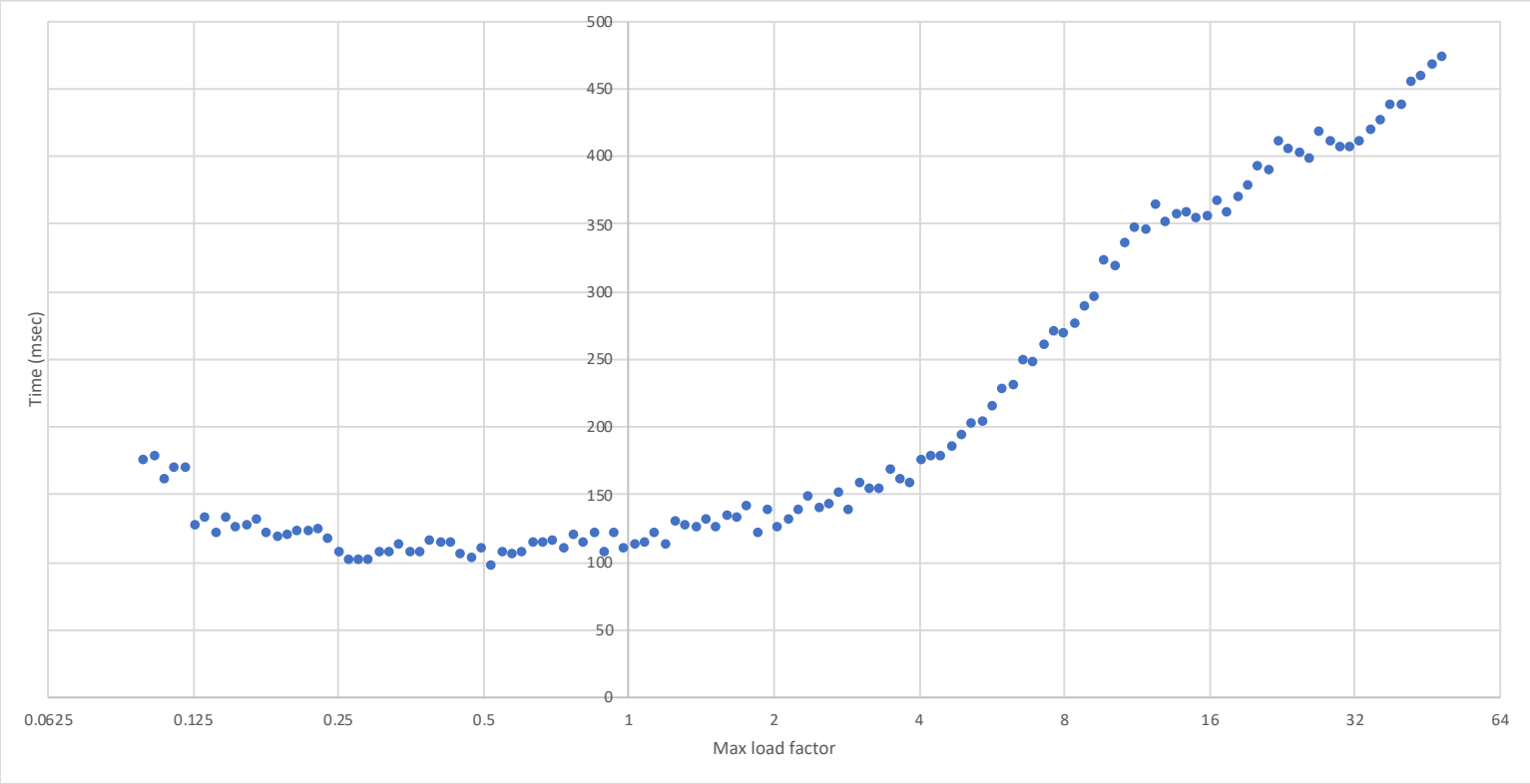


Reminder: to avoid O(n) performance, set a maximum load factor ($\lambda = n/m$) where we double the array and re-hash.

Default for Java HashMap: 0.75

Under "normal circumstances", this achieves O(1) search and amortized O(1) insert/delete.

Impact of Max Load Factor Values on Efficiency of Separate Chaining



- At each value for max load factor, ran 32 experiments
  - Each added a random number $<2^{16}$ of items to an initially empty HashSet

Alternative: Store collisions in other array slots.

- No need to grow in second direction
- No memory required for pointers
  - Historically, this was important!
  - Still is for some data…
- Will still need an appropriate max load factor or else collisions degrade performance
  - We'll grow the array again

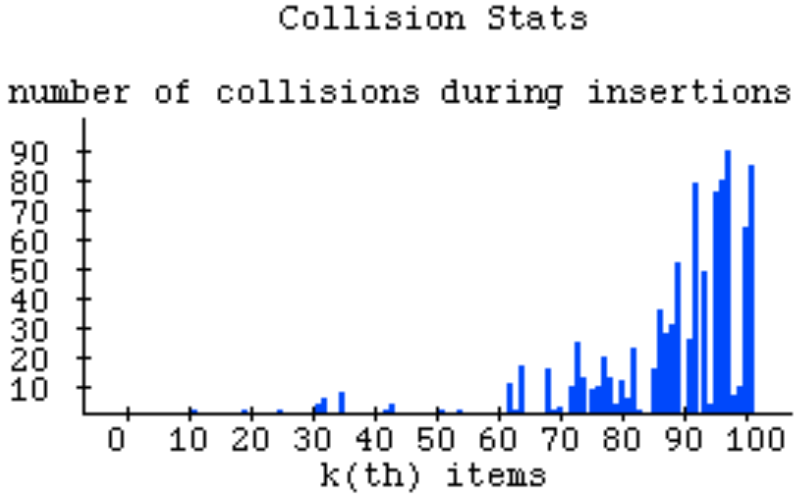
Collision Resolution: Linear Probing

- Probe H (see if it causes a collision)
- Collision? Also probe the next available space:
  - Try H, H+1, H+2, H+3, …
  - Wraparound at the end of the array
- Example on board: .add() and .get()

- Problem: Clustering

- Animation:
  - http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html
  - Applet deprecated on most browsers
  - Moodle has a video captured from there
  - Or see next slide for a few freeze-frames.

Clustering Example



Linear probing efficiency also depends on load factor, $\lambda = n/m$

- For probing to work, $0 \leq \lambda \leq 1$.

- For a given λ, what is the expected number of probes before an empty location is found?

Rough Analysis of Linear Probing

- Assume all locations are equally likely to be occupied, and hashed to.
- λ is the probability that a given cell is full, 1–λ the probability a given cell is empty.
- What's the expected number of probes to find an open location?

$$\sum_{p=1}^{\infty} \lambda^{p-1}(1-\lambda)p = \frac{1}{1-\lambda}$$

If $\lambda = 0.5$

Then $\frac{1}{1-0.5} = 2$

From https://en.wikipedia.org/wiki/List_of_mathematical_series:

$$\sum_{k=1}^{n} kz^k = z\frac{1-(n+1)z^n+nz^{n+1}}{(1-z)^2}$$

Better Analysis of Linear Probing

- **Clustering!** Blocks of neighboring occupied cells
  - Much more likely to insert adjacent to a cluster
  - Clusters tend to grow together (avalanche effect)
- Actual average number of probes for large λ:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

For a proof, see Knuth, The Art of Computer Programming, Vol 3: Searching and Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998. (1st edition = 1968)

Why consider linear probing?

- Easy to implement
- Works well when load factor is low
  - In practice, once $\lambda > 0.5$, we usually **double the size of the array** and rehash
  - This is more efficient than letting the load factor get high
- Works well with caching

To reduce clustering, probe farther apart

- Reminder: Linear probing:
  - Collision at H? Try H, H+1, H+2, H+3,...

- New: **Quadratic** probing:
  - Collision at H? Try H, H+1^2. H+2^2, H+3^2, ...
  - Eliminates primary clustering. "Secondary clustering" isn't as problematic

  - But, new problem: are we guaranteed to find open cells?
  - Try with
    - m=16, H=6.
    - m=17, H=6.

Quadratic Probing works with low λ and prime m

- **Claim. If *m* is prime, then the following are unique:**
  $$H + i^2 \pmod{m} \text{ for } i = 0,1,2,\ldots,\lfloor m/2 \rfloor$$
- **Implication.** Using prime table size *m*, and $\lambda \leq 0.5$, then quadratic probing guarantees
  - Insertion within $\lfloor m/2 \rfloor + 1$ non-repeated probes
  - Unsuccessful search within $\lfloor m/2 \rfloor + 1$ non-repeated probes
- E.g. m=17, H=6: works as long as $\lambda \leq 0.5$ ($n \leq 8$)

For a proof, see Theorem 20.4:
Suppose the table size is prime, and that we repeat a probe before trying more than half the slots in the table
See that this leads to a contradiction

Quadratic Probing runs quickly if we implement it correctly & cleverly

Use an algebraic tricks to calculate next index

- Difference between successive probes yields:
  - Probe i location, $H_i = (H_{i-1} + 2i - 1) \% M$
- Just use bit shift to multiply i by 2
  - probeLoc= probeLoc + (i << 1) - 1;

  …faster than multiplication
- Since i is at most M/2, can just check:
  - if (probeLoc >= M)
    probeLoc -= M;

  …faster than mod

When growing array, can't double!

- Can use, e.g., BigInteger.nextProbablePrime()

Quadratic probing analysis

- No one has been able to analyze it!
- Experimental data shows that it works well
  - Provided that the array size is prime, and $\lambda < 0.5$

Designers choose

- We have been presenting Java's implementation
- In Python's implementation, the designers made some different choices
  - Uses probing, but not linear or quadratic: instead, uses a variant of a linear congruential generator using the recurrence relation

$$H = 5H+1 << \text{perturb}$$

    Implementation, Explanation, Wikipedia on LCGs
  - Also uses 1000003 (also prime) instead of 31 for the String hash function

Summary:

Hash tables are fast for some operations

Structure	insert	Find value	Find max value
Unsorted array			
Sorted array			
Balanced BST			
Hash table			

- Finish the quiz.
- Then check your answers with the next slide

Answers:

Structure	insert	Find value	Find max value
Unsorted array	Amortized $\theta(1)$ Worst $\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted array	$\theta(n)$	$\theta(\log n)$	$\theta(1)$
Balanced BST	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$
Hash table	Amortized $\theta(1)$ Worst $\theta(n)$	$\theta(1)$	$\theta(n)$

In practice

- Constants matter!

- 727MB data, ~190M elements
  - Many inserts, followed by many finds
  - Microsoft's C++ STL

Structure	build (seconds)	Size (MB)	100k finds (seconds)
Hash map	22	6,150	24
Tree map	114	3,500	127
Sorted array	17	727	25

- Why?
- Sorted arrays are nice **if** they don't have to be updated frequently!
- Trees still nice when interleaved insert/find

Review: discuss with a partner

- Why use 31 and not 256 as a base in the String hash function?
- Consider chaining, linear probing, and quadratic probing.
  - What is the purpose of all of these?
  - For which can the load factor go over 1?
  - For which should the table size be prime to avoid probing the same cell twice?
  - For which is the table size a power of 2?
  - For which is clustering a major problem?
  - For which must we grow the array and rehash every element when the load factor is high?

Today's worktime

…is a great time to start StringHashSet while it's fresh

…is acceptable to use for EditorTrees Milestone 2 group worktime, especially if you have questions for me