Hash Tables (cont.); Comparison Sorts

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Which of the following techniques can be used for collision *avoidance*?

A. Choosing a prime table size

B. Choosing a good hash function

C. ~~Using separate chaining~~

D. Ensuring a small λ (eg, resizing the table when λ too large)

E. Choosing a differentiating, but not too differentiating, set of input fields to hash

F. ~~All of the above~~

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Announcements

- Quiz 2 released tomorrow!
  - Due Thursday ***morning***!!!

Lecture Outline

- Hash Tables
  - **Review**
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - Intro

Hash Table Components

```
HashTable h;
h.add("cat", 100);
h.add("snake", 50);
h.add("dog", 200);
```

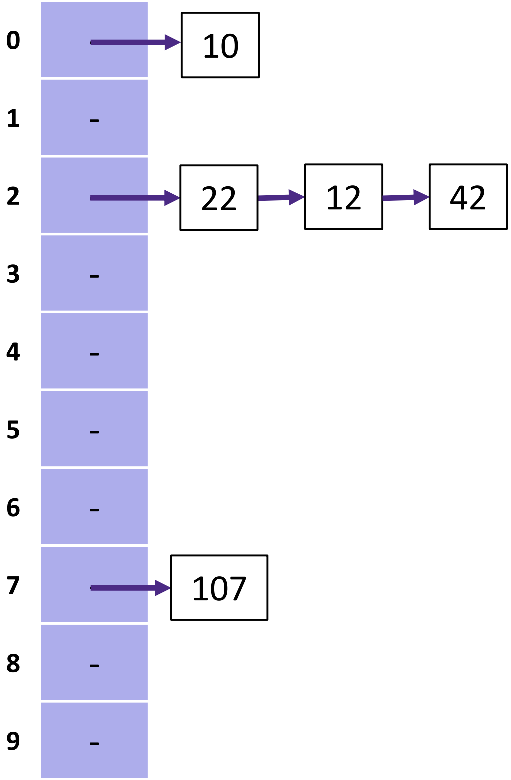
```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("snake") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```

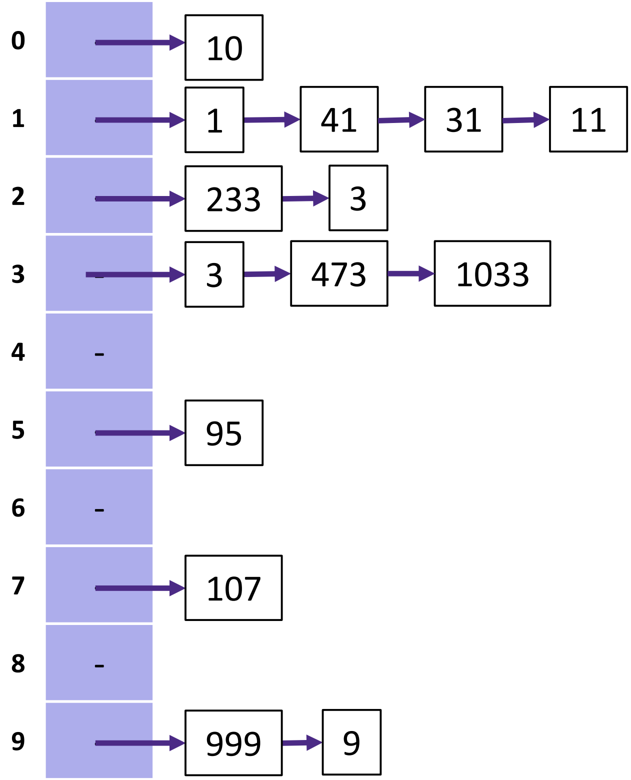
	K	V
0	-	-
1	-	-
2	snake	100
3	bee	50
4	-	-

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Separate Chaining and Load Factor

$$\lambda = \frac{N}{TableSize}$$





Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - **Intro**
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - Intro

Open Addressing Idea

- 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)%TableSize`** is already full, …

```
HashTable h;
h.add(100);
h.add(50);
h.add(200);
```

```
hashFunction(100) == 2;
2 % 5 == 2
hashFunction(50) == 2525393088;
2525393088 % 5 == 3
hashFunction(200) == 9752423;
9752423 % 5 == 3
```

0	-
1	-
2	100
3	50
4	200

Linear Probing: Add Example

- Our first option for resolving this collision is *linear probing*
- 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: add 38, 19, 8, 109, 10

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

Open Addressing

- **Open addressing** resolves 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
  - Typically want larger tables
  - Too many probes means no more *O*(1) 😭😭😭

Linear Probing: find

- You can figure this one out too ☺
  - Must use same probe function to "retrace the trail" for the item
  - Unsuccessful search when reach empty position

- What is `find`'s runtime …
  - If key is NOT there?
  - Worst case?
  - If key is in table?

gradescope

gradescope.com/courses/256241

- What is `find`'s runtime in a open addressing hash table:
  - If key is NOT there? $O(N)$
  - Worst case $O(N)$

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

Linear Probing: Remove

- remove(19)

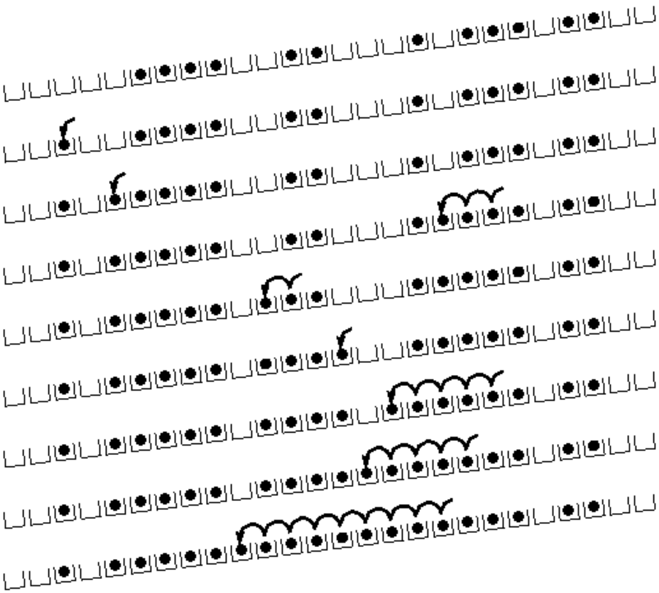
- ***Must*** use "lazy deletion"
  - Marker/tombstone indicates "no item here, but don't stop probing"
  - Without lazy deletion, find() of an existing value is incorrect; with lazy deletion, find() runs in $O(n)$

- As with lazy deletion on other data structures, spots marked "deleted" can be filled in during subsequent adds

0	8	8
1	109	109
2	10	10
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	38	38
9	19	☠

Linear Probing: 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]

Linear Probing: Analysis (1 of 2)

- **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

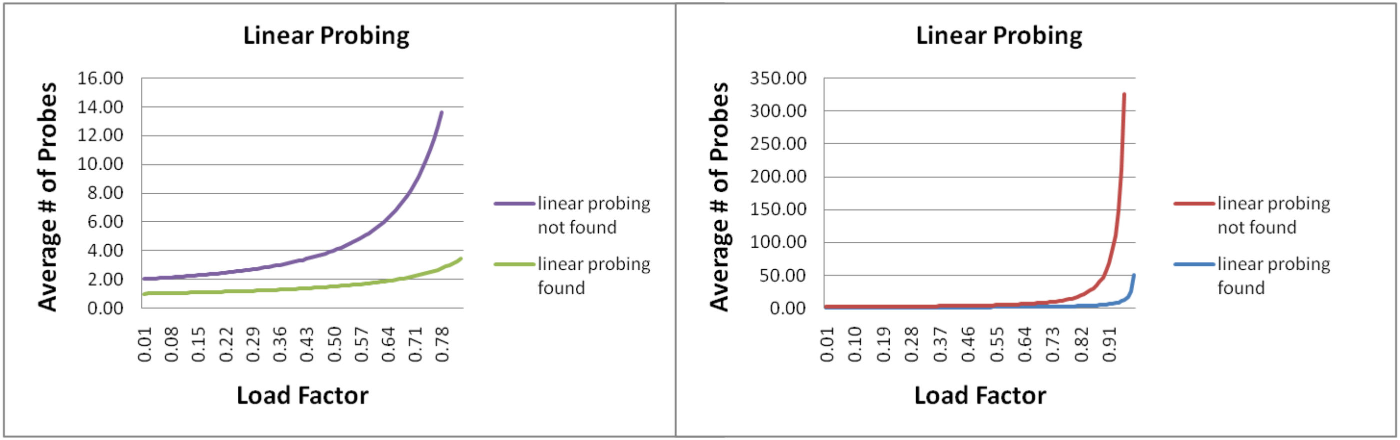
- **Non-trivial facts** we won’t prove:

  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

Linear Probing: Analysis (2 of 2)

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)
  - With open addressing, a "good" λ to aim for is 0.5



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

Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - **Quadratic Probing**
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - Intro

Quadratic Probing

- Avoid primary clustering by changing the probe function:
  - **i**th probe: **(h(key) + i²) % TableSize**
  - Probe sequence becomes:
    - 0th probe: **h(key) % TableSize**
    - 1st probe: **(h(key) + 1) % TableSize**
    - 2nd probe: **(h(key) + 4) % TableSize**
    - 3rd probe: **(h(key) + 9) % TableSize**
- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing: Add Example

- Example: add 89, 18, 49, 58, 79
  - Let hashFunction(x) = x
  - Let `TableSize` = 10

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

Quadratic Probing: Another Add Example (1 of 3)

- Example: add 76, 40, 48, 5, 55, 47
  - Let hashFunction(x) = x
  - Let `TableSize` = 7

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

Quadratic Probing: Another Add Example (2 of 3)

- Example: add ~~76, 40, 48, 5, 55,~~ 47
  - Let hashFunction(x) = x
  - Let `TableSize` = 7
  - (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!
  - **Will we ever get a 1 or 4?!?**

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

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Quadratic Probing: Another Add Example (3 of 3)

- Example: add 76, 40, 48, 5, 55, 47
- Will we ever get a 1 or 4?!?
  - add(47) will *always* fail here. Why?
  - For all i, $(5 + i^2) \% 7$ is 0, 2, 5, or 6
  - Proof uses induction and
    - $(5 + i^2) \% 7 = (5 + (i - 7)^2) \% 7$
- In fact, for all c and k,
  - $(c + i^2) \% k = (c + (i - k)^2) \% k$

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

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Quadratic Probing: Bad News / Good News

- Bad News:
  - After `TableSize` 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 online after lecture
  - Textbook also has proof, but it’s slightly less detailed

Skipped During Lecture

Quadratic Probing: Success Guarantee (1 of 2)

If `TableSize` is prime and $\lambda < ½$, then quadratic probing will find an empty bucket in `TableSize/2` probes or fewer

- Intuition: if the table is less than half full, then probing `TableSize/2` distinct buckets must find an empty one
  - Therefore, prove the first `TableSize/2` probes are distinct

Any i^{th} and any j^{th} probe results in a distinct bucket

- Theorem: for all **0** $\leq$ **i,j** $\leq$ **TableSize/2**, and **i** $\neq$ **j**

  (h(x) + i²) % TableSize ≠ (h(x) + j²) % TableSize

Skipped

Quadratic Probing: Success Guarantee (2 of 2)

- Proof, by contradiction: suppose that for some i ≠ j:

  (h(x) + i²) % TableSize = (h(x) + j²) % TableSize

  ⇒ **i² % TableSize = j² % TableSize**

  ⇒ **(i² - j²) % TableSize = 0**

  ⇒ **[(i + j)(i - j)] % TableSize = 0**

  ⇒ **[(i + j)(i - j)] = k*TableSize** for some k ≥1

  or

  [(i + j)(i - j)] = 0

  CONTRADICTION!

- How can **i+j = 0** or **i+j = k*TableSize** when:

  0 ≤ i,j and **i≠j** and **i,j ≤ TableSize/2**?

- How can **i-j = 0** or **i-j = k*TableSize** when

  i ≠ j and **i,j ≤ TableSize/2** ?

Quadratic Probing: Secondary Clustering

- Quadratic probing does not suffer from primary clustering!
  - We don't grow "big blobs" by adding to the end of a cluster

- Quadratic probing does not resolve collisions between different keys that hash *to the same **index***
  - These keys **have the same series of moves** looking for an empty spot
  - Called **secondary clustering** ☹

- Since the problem occurs when we have the different keys hashing to the same initial index, can we avoid secondary clustering with *a probe function that also incorporates the key*?
  - Known as **double hashing**

Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - **Double Hashing**
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion
- Comparison Sorting
  - Intro

Double Hashing

- Double hashing:
  - **i**th probe: **(h(key) + i*g(key)) % TableSize**
  - Probe sequence becomes:
    - 0th probe: **h(key) % TableSize**
    - 1st probe: **(h(key) + g(key)) % TableSize**
    - 2nd probe: **(h(key) + 2*g(key)) % TableSize**
    - …
- Idea:
  - **g(key)** lets us “go different places from initial collisions”
    - It is very unlikely that for some key, **h(key) == g(key)**
    - (assuming good hash functions h and g)
  - **i*g(key)** lets us “leave the neighborhood”
- Detail: Ensure **g(key)** can’t generate 0

Double Hashing: Add Example (1 of 3)

- Example: add 13, 28, 33, 147, 43
  - Remember: (`h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10

Effectively x/TableSize but constructed to never return 0

① h(13) = 3

② h(28) = 8

③ h(33) = 3 X
g(33) = 1 + 3 % 9 = 4 } (3+4) % 10 = 7

④ h(147) = 7

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

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Double Hashing: Add Example (2 or 3)

- Example: add ~~13, 28, 33, 147,~~ 43
  - Remember: (`h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10

  - h(43) = 3 and g(43) = 1 + (4%9) = 5
  - 3 + 0*5 = 3 collision!
  - 3 + 1*5 = 8 collision!
  - 3 + 2*5 = 13 collision
  - **Will we ever get anything else?!?**

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

Double Hashing: Add Example (3 of 3)

- Example: add ~~13, 28, 33, 147,~~ 43
  - Remember: (`h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10

  - Will we ever get anything else?!?
    - No. add(43) will always fail here. Why?

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

Double Hashing: Considerations (1 of 2)

- Our example implies the possibility of infinite probe sequences ☹
  - But we can be avoid infinite probes if our functions are:
    - h(key) = hash1(key) % p
    - g(key) = q – (hash2(key) % q)
  - And p and q are primes, with 2 < q < p

Double Hashing: Considerations (2 of 2)

- Double hashing:
  - **i**th probe: **`(h(key) + i*g(key)) % TableSize`**
- Assume **`g(key)`** divides `TableSize`
  - That is, there exists some integer `x` such that `x*g(key)=TableSize`
  - Therefore: after x probes, we'll "loop through" the same indices as before
  - Example:
    - `TableSize=50`
    - `g(key)=25`
    - Probe sequence:
      - i=0: h(key)
      - i=1: h(key)+25
      - i=2: h(key)+50 = h(key)
      - i=3: h(key)+75 = h(key)+25
      - …
- Bottom line: don't let **`g(key)`** divide `TableSize`
  - That is, choose a prime `TableSize` when using double hashing

Double Hashing: Performance

- Assume g() distributes its keys uniformly over its range
  - That is: probability of `g(key1)%p == g(key2)%p` is `1/p`
- We won't prove the following:
  - Average # of probes (in the limit as TableSize →∞), **unsuccessful** find:

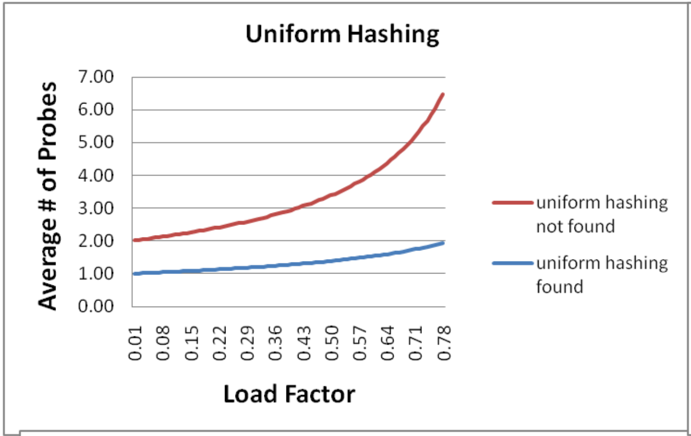
$$\frac{1}{1-\lambda}$$

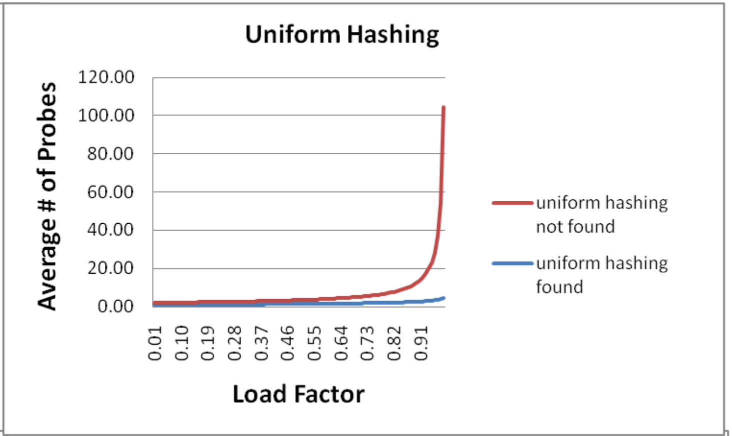
  - Average # of probes (in the limit as TableSize →∞), **successful** find:

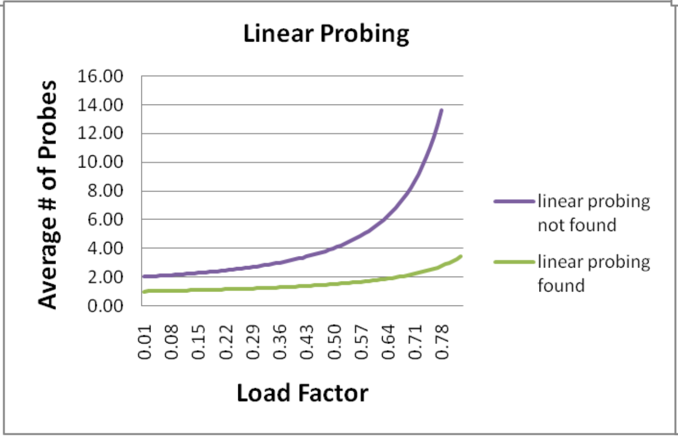
$$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$

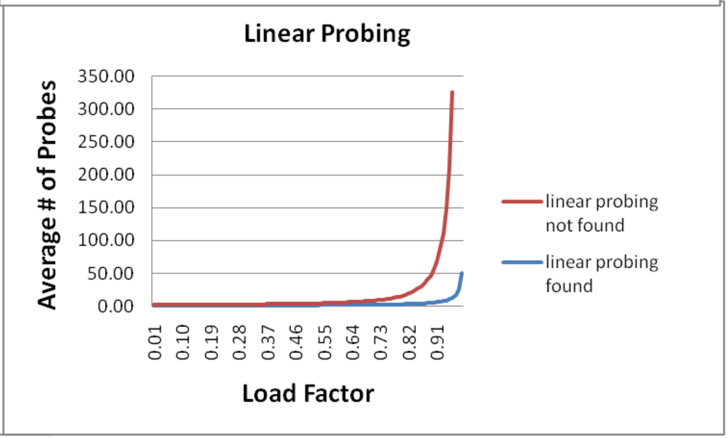
- Bottom line:
  - Performance of unsuccessful finds degrades with λ (but not as quickly as linear probing degrades)
  - Performance of successful finds degrades not nearly as quickly

Double Hashing vs Linear Probing Performance









Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - **Collision Avoidance: Rehashing**
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - Intro

Separate Chaining vs Open Addressing

- Separate Chaining
  - **`find, add, remove`** proportional to λ if using unsorted LL
  - If using another data structure for buckets (e.g. AVL tree) , runtime is proportional to runtime for that structure

- Open addressing: has clustering issues as table fills ($\lambda > 1/2$)
  - Why use it:
    - Some runtime for allocating nodes; open addressing could be faster?
    - Easier data representation?

Rehashing (1 of 3)

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and "copy" everything over
- With separate chaining, we decide what "too full" means
  - Keep load factor reasonable (e.g., < 2)?
  - Consider average or max size of non-empty chains?
- For open addressing, half-full is a good rule of thumb

Rehashing (2 of 3)

- Can't actually copy to the same indices in the new table
  - We'd calculated the index based on `TableSize`

- For each key/value in old table, must `add` into new table
  - Iterate over old table: O(n)
  - n calls to the hash function: $n \cdot O(1) = O(n)$

- Can we avoid all those hash function calls?
  - Space/time tradeoff: Could store `h(key)` with each item
  - Iterating over the table is still O(n); saving `h(key)` only helps by a constant factor

Rehashing (3 of 3)

- New table size
  - Twice-as-big is a good idea, except … ummm … that won't be prime!
  - So go *about* twice-as-big
    - Hard-coded list of primes (you probably won't grow more than 20-30 times)
    - Calculate primes after that

Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)* ← Skipped during lecture
  - Conclusion
- Comparison Sorting
  - Intro

Hashing and Equality Testing

Skipped

- Our examples use an `int` key, which overlooks a critical detail:
  - We hash **`K`** to get a table index
  - While chaining or probing, we need to test whether the current **`K'`** is equal to the **`K`** we're looking for
- So a Java hash table needs a hash *and* an equality function
  - Fortunately, in Java every object defines an **`equals`** and a **`hashCode`** method

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

Skipped!

Overriding equals()? Override hashCode() too

- The Java library (and your project's hash table) make a very important assumption that *all* clients must satisfy:
  - Object-oriented way of saying it:

    If **`a.equals(b)`**, then **`a.hashCode() == b.hashCode()`**
  - Functor way of saying it:

    If **`c.compare(a,b) == 0`**, then

    `h.hashCode(a) == h.hashCode(b)`

- In other words, if you ever override equals:
  - You must also override hashCode() in a consistent way
  - See <u>Core Java</u> book, Ch. 5, for other "gotchas" with equals()

compareTo() rules

Skipped :)

- Java also makes assumptions about `compareTo()` that affect:
  - All our dictionaries
  - Sorting (next major topic)

- Comparison must impose a consistent, total ordering:
  - For all **a**, **b**, and **c**,
    - If `a.compareTo(b) < 0`, then `b.compareTo(a) > 0`
    - If `a.compareTo(b) == 0`, then `b.compareTo(a) == 0`
    - If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c) < 0`

Skipped ♡

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 conversion to int
    Object: object.hashCode()
  result = 31 * result + fieldHashcode;

return result;
```

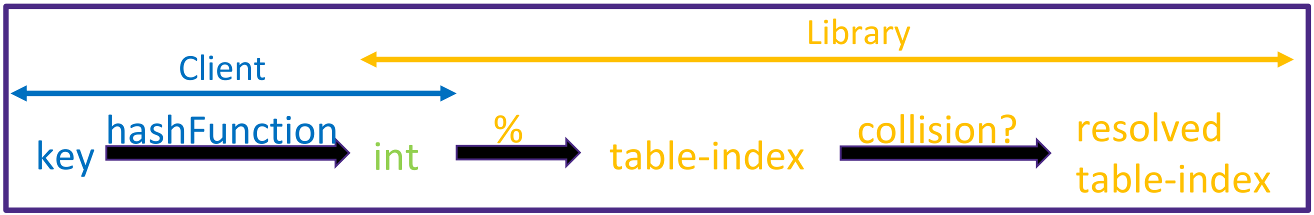
Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - **Conclusion**

- Comparison Sorting
  - Intro

Who Hashes What?

- When used as a library, hash tables generally have two roles: client vs library



- We learned both, but you'll spend more time as clients
  - Both roles must contribute to minimizing collisions
  - Client should aim for different ints for expected keys
    - Avoid "wasting" any part of K or the int's bits
  - Library should aim for putting "similar" ints in different indices
    - Conversion to index is almost always "mod table-size"
    - Using prime numbers for table-size is common

Summary: Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just **`add`**, **`find`**, **`remove`**, hash tables and balanced trees are just different data structures
  - Hash tables *O*(1) on average (assuming few collisions)
  - Balanced trees *O*(**`log`** *n*) worst-case

- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (must avoid collisions)
  - Yes, but what if we want to **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`**, **`printSorted?`**
    - Hash tables are not designed to efficiently implement these operations
    - Your textbook considers hash tables to be a different ADT; not so important to argue over the definitions

Summary: Hash Table (1 of 2)

- Hash tables are categorized by collision *resolution* strategy:
  - **Separate chaining**: use an auxiliary data structure so that colliding keys can both use the same index
    - Simple is best (eg, linked list, or LL + an extra key/value slot)
    - λ can be > 1, but recommend keeping it "smallish"
  - **Open addressing**: look elsewhere in the array if keys collide. $\lambda \leq 1$
    - **Linear probing**: finds a slot if $\lambda < 1$, but primary clustering severely impacts performance (secondary clustering is also a consideration)
    - **Quadratic probing**: finds a slot if $\lambda < 0.5$. No primary clustering but secondary clustering is possible
    - **Double hashing**: finds depending on how h(x) and g(x) are constructed.

Summary: Hash Table (2 of 2)

- Collision *avoidance* applicable to both types of hash table
  - **Crucial** to use a good hash function: deterministic, fast, uniform
  - Which fields to hash is **important**: need “just enough” differentiation
  - Array size is **important**:
    - Choose a prime size
    - “Preferred λ” depends on type of table; resize (rehash) to maintain

- What we skipped:
  - Perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing

- The hash table is one of the most important data structures
  - Useful in many, many, many real-world applications

key —hashFunction→ int —%→ table-index —collision?→ resolved table-index

Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - **Intro**

Introduction to Sorting (1 of 2)

- Stacks, queues, priority queues, and dictionaries/sets all provide one element at a time

- But often we want "all the items" in some order
  - Alphabetical list of people
  - Population list of countries
  - Search engine results by relevance

- Different sorting algorithms have different asymptotic and constant-factor trade-offs
  - Knowing one way to sort just isn't enough; no single "best sort"
  - **Sorting is an excellent case-study in making trade-offs!**

Introduction to Sorting (2 of 2)

- *Preprocessing* (e.g. sorting) data to make subsequent operations faster is a general technique in computing!
  - Example: Sort the items so that you can:
    - Find the **`k`**th largest in constant time for any **`k`**
    - Perform binary search to find an item in logarithmic time
  - Whether preprocessing is beneficial depends on
    - How often the items will change
    - How many items there are

- Preprocessing's benefits depend on how often the items will change and how many items there are
  - **Sorting is an excellent case-study in making trade-offs!**

Comparison Sorting: Definition

- Problem: We have *n* comparable items in an array, and we want to rearrange them to be in increasing order

- Input:
  - An array A of (key, value) pairs
  - A comparison function (consistent and total)
    - Given keys a & b, what is their relative ordering? <, =, >?
    - Ex: keys that implement Comparable or have a Comparator

- Output/Side-Effect:
  - Reorganize the elements of A such that for any index i and j,

    if $i < j$ then $A[i] \leq A[j]$
  - [Usually unspoken] A must have all the same items it started with
  - Could also sort in reverse order, of course

Comparison Sort: Variations (1 of 2)

1. Maybe elements are in a linked list
   - Could convert to array and back in linear time, but some algorithms can still "work" on linked lists
2. Maybe if there are ties we should preserve the original ordering
   - Sorts that do this naturally are called **stable sorts**
3. Maybe we must not use more than O(1) "auxiliary space"
   - These are called **in-place sorts**
   - Not allowed to allocate memory proportional to input (i.e., O(n)), but can allocate O(1) # of variables
   - Work is done by swapping around in the array

Comparison Sort: Variations (2 of 2)

4. Maybe we can do more with elements than just compare
   - Comparison sorts assume a binary 'compare' operator
   - In special cases we can sometimes get faster algorithms

5. Maybe we have too many items to fit in memory
   - Use an **external sorting** algorithm

Sorting: The Big Picture

- Simple comparison-based algorithms: $O(n^2)$
  - InsertionSort, SelectionSort
  - *BubbleSort, ShellSort*

- Fancier comparison-based algorithms: O(n log n)
  - HeapSort, MergeSort, QuickSort (randomized)

- Comparison-based sorting's lower bound: Ω(n log n)

- Specialized algorithms: O(n)
  - BucketSort, RadixSort

- Handling huge data sets:
  - External sorting

Lecture Outline

- Hash Tables
  - Review
  - Collision Resolution: Open Addressing
    - Intro
    - Quadratic Probing
    - Double Hashing
  - Collision Avoidance: Rehashing
  - *(Java-specific Hash Table Concerns)*
  - Conclusion

- Comparison Sorting
  - Intro