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

Collision resolution

Collision:
When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution
- Ideas?

Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list
(a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10

Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list
(a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10

Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list
(a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10

Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list
(a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10
Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10

Thoughts on chaining

- Worst-case time for find?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)

- Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  - Linked list vs. array vs. chunked list (lists should be short!)
  - Move-to-front (cf. Project 2)
  - Better idea: Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
  - A time-space trade-off...

Time vs. space (constant factors only here)

Definition: The load factor, $\lambda$, of a hash table is

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

Under chaining, the average number of elements per bucket is ____

So if some inserts are followed by random finds, then on average:

- Each unsuccessful find compares against ____ items
- Each successful find compares against ____ items

More Rigorous Chaining Analysis
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

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

$\lambda$蔺 number of elements

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:
- Each unsuccessful find compares against $\lambda$ items
- Each successful find compares against $\lambda/2$ items

So we like to keep $\lambda$ fairly low (e.g., 1 or 1.5 or 2) for chaining

Alternative: Use empty space in the table

- Another simple idea: 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

Alternative: Use empty space in the table

- Another simple idea: 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

Alternative: Use empty space in the table

- Another simple idea: 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
**Open addressing**

This 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 was $(h(key) + i) \% TableSize$
- In general have some **probe function** $f$ and use $h(key) + f(i) \% TableSize$

Open addressing does poorly with high load factor $\lambda$

- 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”

(If it makes you feel any better, most trees in CS grow upside-down 🌳)

---

**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”
  - Note: **delete** with chaining is plain-old list-remove

---

**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 $\lambda$ (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)

---

**In a chart**

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

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

- We can avoid primary clustering by changing the probe function
  \( h(key) + f(i) \% \text{TableSize} \)
- A common technique is quadratic probing:
  \( f(i) = i^2 \)
  - So probe sequence is:
    - 0th probe: \( h(key) \% \text{TableSize} \)
    - 1st probe: \( (h(key) + 1) \% \text{TableSize} \)
    - 2nd probe: \( (h(key) + 4) \% \text{TableSize} \)
    - 3rd probe: \( (h(key) + 9) \% \text{TableSize} \)
    - ...
    - \( i^{th} \) probe: \( (h(key) + i^2) \% \text{TableSize} \)
- Intuition: Probes quickly “leave the neighborhood”
Quadratic Probing Example

Table Size = 10
Insert:
0
1 49
2 58
3 79
4
5
6
7
8 18
9 89

Another Quadratic Probing Example

Table Size = 7
Insert:
0
1
2 76 (76 % 7 = 6)
3 40 (40 % 7 = 5)
4 48 (48 % 7 = 6)
5 5 ( 5 % 7 = 5)
6 55 (55 % 7 = 6)
7 47 (47 % 7 = 5)

Another Quadratic Probing Example

Table Size = 7
Insert:
0
1
2 76 (76 % 7 = 6)
3 40 (40 % 7 = 5)
4 48 (48 % 7 = 6)
5 5 ( 5 % 7 = 5)
6 55 (55 % 7 = 6)
7 47 (47 % 7 = 5)

Another Quadratic Probing Example

Table Size = 7
Insert:
0 48
1
2
3
4
5 40
6 76

Another Quadratic Probing Example

Table Size = 7
Insert:
0 48
1
2 5
3
4
5 40
6 76
Another Quadratic Probing Example

Table Size = 7

Doh!: For all \( n \), \((n*n) + 5) \mod 7\) is 0, 2, 5, or 6
- Excel shows takes “at least” 50 probes and a pattern
- Proof uses induction and \((n^2+5) \mod 7 = ((n-7)^2+5) \mod 7\)
- In fact, for all \( c \) and \( k\), \((n^2+c) \mod k = ((n-k)^2+c) \mod k\)

From Bad News to Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- 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 is posted in lecture11.txt
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime \( T \) and \( 0 < i, j < T/2 \) where \( i \neq j\),
      \((k + i^2) \mod T \neq (k + j^2) \mod T \) (i.e., no index repeat)

Double hashing

Idea:
- Given two good hash functions \( h \) and \( g \), it is very unlikely that for some key, \( h(\text{key}) == g(\text{key}) \)
- So make the probe function \( f(i) = i*g(\text{key}) \)

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

Detail: Make sure \( g(\text{key}) \) cannot be 0

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering:
  - no problem with keys initially hashing to the same neighborhood
- But it’s no help if keys initially hash to the same index
  - Called secondary clustering
- Can avoid secondary clustering with a probe function that depends on the key: double hashing...

Double-hashing analysis

- Intuition: Because each probe is “jumping” by \( g(\text{key}) \) each time, we “leave the neighborhood” and “go different places from other initial collisions”
- But we could still have a problem like in quadratic probing where we are not “safe” (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - \( h(\text{key}) = \text{key} \mod p \)
    - \( g(\text{key}) = q - (\text{key} \mod q) \)
    - \( 2 < q < p \)
    - \( p \) and \( q \) are prime
More double-hashing facts

- Assume "uniform hashing"
  - Means probability of \( g(\text{key}_1) \% p == g(\text{key}_2) \% p \) is \( 1/p \)
- Non-trivial facts we won’t prove:
  - 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?

- **Chaining** is easy
  - `find`, `delete` proportional to load factor on average
  - `insert` can be constant if just push on front of list
- **Open addressing** uses probing, has clustering issues as table fills
  - Why use it:
    - Less memory allocation?
    - Easier data representation?
- Now:
  - Growing the table when it gets too full ("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
- With 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 won’t grow more than 20-30 times

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
  - Run-time?
    - `O(n)`: Iterate through old table
  - Resize is an `O(n)` operation, involving `n` calls to the hash function
  - Is there some way to avoid all those hash function calls?
    - Space/time tradeoff: Could store `h(key)` with each data item
    - Growing the table is still `O(n)`: only helps by a constant factor

Hashing and comparing

- Need to emphasize a critical detail:
  - We initially `hash E` to get a table index
  - While chaining or probing we `compare to E`
    - Just need equality testing (i.e., "is it what I want")
- So a hash table needs a hash function and a comparator
  - In Project 2, you will use two function objects
  - The Java library uses a more object-oriented approach: each object has an `equals` method and a `hashCode` method

```java
class Object {
    boolean equals(Object o) { ... }
    int hashCode() { ... }
}
```
Equal Objects Must Hash the Same

- The Java library (and your project hash table) make a very important assumption that clients must satisfy...
- Object-oriented way of saying it:
  
  ```java
  if (a.equals(b), then we must require
  a.hashCode() == b.hashCode()  
  ```

- Function-object way of saying it:
  
  ```java
  if (c.compare(a,b) == 0, then we must require
  h.hash(a) == h.hash(b)
  ```

- Why is this essential?

Java bottom line

- Lots of Java libraries use hash tables, perhaps without your knowledge
- So: If you ever override `equals`, you need to override `hashCode` also in a consistent way
  
  ```java
  see CoreJava book, Chapter 5 for other "gotchas" with equals
  ```

Bad Example

- Think about using a hash table holding points

```
class PolarPoint {
  double r = 0.0;
  double theta = 0.0;
  void addToAngle(double theta2) { theta+=theta2; }
  ...
  boolean equals(Object otherObject) {
    if(this==otherObject) return true;
    if(otherObject==null) return false;
    if(getClass()!=other.getClass()) return false;
    PolarPoint other = (PolarPoint)otherObject;
    double angleDiff = (theta - other.theta) % (2*Math.PI);
    double rDiff = r - other.r;
    return Math.abs(angleDiff) < 0.0001 &&
           Math.abs(rDiff) < 0.0001;  
    } // wrong: must override hashCode!
  }
```

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`

Final word on hashing

- The hash table is one of the most important data structures
  - Supports only `find`, `insert`, and `delete` efficiently
- Important to use a good hash function
- Important to keep hash table at a good size
- What we skipped: Perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing
- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums