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

An Alternative to Separate Chaining: Open Addressing

• 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

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”?
An Alternative to Separate Chaining: Open Addressing

- Another simple idea: If \( h(key) \) is already full,
  - try \((h(key) + 1) \mod \text{TableSize}. \) If full,
  - try \((h(key) + 2) \mod \text{TableSize}. \) If full,
  - try \((h(key) + 3) \mod \text{TableSize}. \) If full...

- Example: insert 38, 19, 8, 109, 10

Open addressing

This is one example of open addressing
- More generally, we just need to describe where to check next when one attempt fails (cell already in use)
- Each version of open addressing involves specifying a sequence of indices to try

Trying the next spot is called probing
- Our \( i^{th} \) probe was: \( (h(key) + i) \mod \text{TableSize} \)
- This is called linear probing
- In general have some probe function \( f \) and use:
  \[ (h(key) + f(i)) \mod \text{TableSize} \]
  for the \( i^{th} \) probe (start at \( i=0 \))
- For linear probing, \( f(i)=i \)

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”
Open Addressing: Linear Probing

What about find? If value is in table? If not there? Worst case?

What about delete?

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

Other operations

Okay, so insert finds an open table position using a probe function

What about find?

– Must use same probe function to “retrace the trail” and find the data
– Unsuccessful search when reach empty position

What about delete?

– Must use “lazy” deletion. Why?
– But here just means “no data here, but don’t stop probing”
– Note: delete with chaining is plain-old list-remove

(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 probing sequences
• Called primary clustering
• Saw this starting in our 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 $\lambda$ (in the limit as TableSize $\rightarrow \infty$)
  – Unsuccessful search:
    \[
    \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)
    \]
  – 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 $\lambda > 1$

Open Addressing: linear probing

\[
\text{in linear probing}
\]

– For linear probing:
  \[
  d(i) = i
  \]

– So probe sequence is:
  • \(O\) probe \(h(i)\) \& TableSize
  • If probe \(h(i)\) \& TableSize
    \[
    d(i) = i + 3 \times \text{TableSize}
    \]
  • If probe \(h(i)\) \& TableSize
    \[
    d(i) = i + 3 \times \text{TableSize}
    \]
  • ...
Open Addressing: Quadratic probing

- We can avoid primary clustering by changing the probe function:
  \[ (h(key) + i^2) \mod \text{TableSize} \]
- For quadratic probing:
  \[ f(i) = i^2 \]
- Its probe sequence is:
  - 0th probe: \( h(key) \mod \text{TableSize} \)
  - 1st probe: \( h(key) + 1^2 \mod \text{TableSize} \)
  - 2nd probe: \( h(key) + 2^2 \mod \text{TableSize} \)
  - 3rd probe: \( h(key) + 3^2 \mod \text{TableSize} \)
  - ...
- Intuition: Probes quickly "leave the neighborhood"
Quadratic Probing Example

TableSize=10
Insert:

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

Another Quadratic Probing Example

TableSize = 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)
8
9

Another Quadratic Probing Example

TableSize = 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)
8
9

Another Quadratic Probing Example

TableSize = 7
Insert:

0 48
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)
8
9

Another Quadratic Probing Example

TableSize = 7
Insert:

0 48
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)
8
9

ith probe: \( h(key) + i^2 \) % TableSize
Another Quadratic Probing Example

TableSize = 7

Insert:
- 76 (76 % 7 = 6)
- 40 (40 % 7 = 5)
- 48 (55 % 7 = 6)
- 5 ( 5 % 7 = 5)
- 55 (55 % 7 = 6)
- 47 (47 % 7 = 5)
- 40

0
48
1
5
2
55
3
40
4
47
5
5
6
76

TableSize = 7

Insert:
- 76 (76 % 7 = 6)
- 40 (40 % 7 = 5)
- 48 (55 % 7 = 6)
- 5 ( 5 % 7 = 5)
- 55 (55 % 7 = 6)
- 47 (47 % 7 = 5)
- 40

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

Quadratic Probing: Success guarantee for \( \lambda < \frac{1}{2} \)

- If size is prime and \( \lambda < \frac{1}{2} \), 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 \( i \neq j \):
    - \( (h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size} \)
  - by contradiction: suppose that for some \( i \neq j \):
    - \( (h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size} \)
    - \( i^2 \mod \text{size} = j^2 \mod \text{size} \)
    - \( (i^2 - j^2) \mod \text{size} = 0 \)
    - \( (i + j)(i - j) \mod \text{size} = 0 \)
    - BUT size does not divide \( i+j \) or \( i-j \)

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: quadratic nature quickly escapes the neighborhood
- But it’s no help if keys initially hash to the same index
- Called secondary clustering
  - Any 2 keys that hash to the same value will have the same series of moves after that
- 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 \( g \), it is very unlikely that for some key \( x \), \( h(x) = g(x) \)

\( g(x) + \lfloor L(\frac{g(x)}{2}) \rfloor \mod \text{TableSize} \)

- For double hashing:
  - \( f(x) = L-g(x) \)
- \( s \)-th probe sequence is:
  - \( i^s \mod \text{TableSize} \)
  - \( g(x) \mod \text{TableSize} \)
  - \( 2^s g(x) \mod \text{TableSize} \)
  - \( 3^s g(x) \mod \text{TableSize} \)
  - \( \cdots \)
  - \( \cdots \)

Detail: \( g(x) \mod 2^n \) can’t be 0
### Resolving Collisions with Double Hashing

| Insert these values into the hash table in this order. Resolve any collisions with double hashing: |
|---|---|---|---|---|
| 13 | 28 | 33 | 147 | 43 |

\[ T = 10 \text{ (TableSize)} \]

Hash Functions:

\[ h(key) = key \mod T \]

\[ g(key) = 1 + ((key/T) \mod (T-1)) \]

### Double-hashing analysis

- **Intuition:** Since each probe is "jumping" by \( g(key) \) 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:
    - \( h(key) = key \mod p \)
    - \( g(key) = q - (key \mod q) \)
    - \( 2 < q < p \)
    - \( p \) and \( q \) are prime

### Yet another reason to use a prime TableSize

- So, for double hashing

  \( P \) probe: \( h(key) + i \cdot g(key) \mod \text{TableSize} \)

- Say \( g(key) \) divides \( \text{TableSize} \)

  - That is, there is some integer \( x \) such that \( xg(key) = \text{TableSize} \)

- After \( x \) probes, we'll be back to trying the same indices as before

- **Ex:**

  - \( \text{TableSize} = 50 \)
  - \( g(key) = 25 \)
  - Probing sequence:

    - \( h(key) \)
    - \( h(key) + 25 \)
    - \( h(key) + 50 = h(key) \)
    - \( h(key) + 75 = h(key) + 25 \)

- Only 1 & itself divide a prime

### More double-hashing facts

- Assume "uniform hashing"

  - Means probability of \( g(key1) \mod p = g(key2) \mod p \) is \( 1/p \)

- Non-trivial facts we won’t prove:

  - Average \# of probes given \( \lambda \) (in the limit as \( \text{TableSize} \to \infty \))
    - Unsuccessful search (intuitive): \[ \frac{1}{1-\lambda} \]
    - Successful search (less intuitive): \[ \frac{1}{\lambda} \log \left( \frac{1}{1-\lambda} \right) \]

- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

### Charts

- **Separate Chaining** is easy

  - **insert, find, delete** proportion to load factor on average
  
  (insert can be constant if just push on front of list)

- **Open addressing** uses probe functions, has clustering issues as table gets full

  - 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

• Like with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over
• Especially 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, and then calculate after that

More on rehashing

• What if we copy all data to the same indices in the new table?
  – Not going to work; calculated index based on TableSize – we may not be able to find it later
• Go through current table, do standard insert for each into new table; run-time?
  – \(O(n)\): Iterate through table
• But resize is an \(O(n)\) operation, involving \(n\) calls to the hash function (1 for each insert in the new table)
  – Is there some way to avoid all those hash function calls again?
• Space/time tradeoff: Could store \(h(key)\) with each data item, but since rehashing is rare, this is probably a poor use of space
  • And growing the table is still \(O(n)\); only helps by a constant factor

Hashing and comparing

• For insert/find, as we go through the chain or keep probing, we have to compare each item we see to the key we’re looking for
  – We need to have a comparator (or key’s type needs to be comparable)
  – Don’t actually need < & >; just =
• So a hash table needs a hash function and a comparator
  – In Project 2, you’ll use two function objects
  – The Java standard library uses a more OO approach where 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:
  • OO 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
  – See CoreJava book, Chapter 5 for other “gotchas” with \(equals\)

Bad Example

```java
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;
    }
}
```

• Think about using a hash table holding points

```java
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;
    }
}```
Aside: Comparable/Comparator have rules too

We didn’t emphasize some important “rules” about comparison functions for:
– all our dictionaries
– sorting (next major topic)

Comparison must impose a consistent, total ordering:
For all a, b, and c,
– If \( \text{compare}(a, b) < 0 \), then \( \text{compare}(b, a) > 0 \)
– If \( \text{compare}(a, b) == 0 \), then \( \text{compare}(b, a) == 0 \)
– If \( \text{compare}(a, b) < 0 \) and \( \text{compare}(b, c) < 0 \), then \( \text{compare}(a, c) < 0 \)

Some final arguments for a prime table size

If TableSize is 60 and…
– Lots of data items are multiples of 5, wasting 80% of table
– Lots of data items are multiples of 10, wasting 90% of table
– Lots of data items are multiples of 2, wasting 50% of table

If TableSize is 61…
– Collisions can still happen, but 5, 10, 15, 20, … will fill table
– Collisions can still happen but 10, 20, 30, 40, … will fill table
– Collisions can still happen but 2, 4, 6, 8, … will fill table

In general, if \( x \) and \( y \) are “co-prime” (means \( \gcd(x, y) == 1 \)), then
\[
(a \cdot x) \mod y == (b \cdot x) \mod y \quad \text{if and only if} \quad a \mod y == b \mod y
\]
– So, given table size \( y \) and keys as multiples of \( x \), we’ll get a decent distribution if \( x \) & \( y \) are co-prime
– Good to have a TableSize that has no common factors with any “likely pattern” \( x \)

Final word on hashing

• The hash table is one of the most important data structures
  – Supports only find, insert, and delete efficiently
  – \( \text{FindMin, FindMax, predecessor, etc.} \): not so efficiently
  – Most likely data-structure to be asked about in interviews; many real-world applications
• Important to use a good hash function
  – Good distribution
  – Uses enough of key’s values
• Important to keep hash table at a good size
  – Prime #
  – Preferable \( \lambda \) depends on type of table
• Side-comment: hash functions have uses beyond hash tables
  – Examples: Cryptography, check-sums