Hash Details
Load Factor

- $\lambda =$ ratio of # elements in hash table to the table size.

- Load factor is a measure of how full the hash table is.
  - $\lambda = 0$ means empty.
  - $\lambda = 1$ means completely full.
How Avoid Collision?

1. Keep $\lambda$ small.
2. Have good hash function that distributes keys across entire array.
3. Good collision function to prevent clustering.

So why are these three enough to prevent bad collisions?
Open Addressing Insert (part 1)

• How small is small $\lambda$ for open addressing?

• $\lambda \leq 0.5$

• Why? Well, imagine doing an insert.
  • when $\lambda \leq 0.5$ then 50% of cells are occupied.
    – so 50% chance of finding empty cell
  • when $\lambda \leq 0.2$ then 20% of cells are occupied.
    – so 80% chance of finding empty cell
  • the probability of randomly finding an available cell is just $1-\lambda$.  

Ok, from this point, the proof gets complex. But can see how it translates into probabilities.

Find that with linear probing, the number of probes until get a place to insert (put) is

$$\# \text{probes} = \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$$
Open Addressing Insert (part 3)

- Quadratic probing.

- If $\lambda \leq 0.5$ and table size is prime, then can always insert. Otherwise may never find a place!

Why prime? Can get into a loop where don’t search all the cells.

- Imagine probing with collision function $f(i) = 2i$.
- If table size is even, then never visits half the cells!
Separate Chaining Insert

- The number of steps for a successful insert.
  \[ \# \text{steps} = 1 \]

- Wow! Just insert at beginning of linked list.
- Load factor can be larger with almost no penalties.
  - Why? Because *no probing*!
    - Conflicts are resolved by adding to a linked list.
  - Reality? Allocating new cell in linked list can be expensive.
Open Address Search/Find

A “find” will take \textit{at most} the same amount of time as an insert.
  
  Why? Because that’s the max distance have to probe.
  
  This assumes open addressing!

So the big-O run time will depend on the load factor!
  
  see earlier formulas…

But typically O(1) for insert, find
  
  see next example.
“Find” Example

• Consider the slower linear probing.

\[
\# \text{probes} = \frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2}\right)
\]

• If \(\lambda = 0.2\), then \#probes is 1.2 on average.
  • FAST!
  • And nearly independent of N as long as not near the table size (when near table size, probing gets out of hand).
  • Constant time!!!

In fact, a more complex analysis shows that the absolute worst behavior is actually O(N), but on average is constant time.
“Find” Example

• But suppose $\lambda = 0.9$. Then will take approximately 50 probes for search.
  • Much worse!
  • So keep $\lambda$ small.
Separate Chaining Search/Find

- The number of steps for a find (on average)
  \[ \# \text{steps} = 1 + \frac{\lambda}{2} \]

- Why?
  - Takes “1” step to get to the correct array index.
  - Then have to traverse the linked list.
    - On average have \( \lambda \) elements in each chain! (try example to see this!)
    - Will have to traverse half the list on average. i.e., \( \lambda/2 \)
Rehashing

• Suppose you don’t make your array large enough.
• Suppose insert lots of keys and values.
• Then $\lambda$ gets too big.

Solution? Rehash.

• Create a larger array.
• Move every element into the new array.
• Cost? $O(N)$ because have to walk through original array.
• But real cost? Is rare, so most of the time, won’t notice this expensive operation!
Rehashing

• Do this whenever $\lambda$ gets bigger than say 0.5, 0.6…

• Usually make table about twice as large (but prime if you can!).

• Cost is $O(N)$ to copy array of size $N$.
  – If do when $\lambda = 0.5$ then have already done $N/2$ inserts.
    • In that case, what’s the total cost? $N/2 + N$.
    • In that case, what’s the cost per insert?
      » total cost / # inserts = $(N/2 + N) / N/2 = 3$

  – So the amortized cost of moving all the elements is really just a small constant of 3 added onto each insert.
    • Trivial! Over the long term will be barely noticeable.
    • But there will be occasional noticeable $O(N)$ slowdowns.
Hash Summary

- Simple array implementation possible.
- Use hash function
  - want uniform distribution
  - add collision function or chaining
- Ideally:
  - Keep load factor small.
  - Keep clustering to a minimum.
  - Make the table size prime (help avoid collisions).
- Very fast for searches
  - \(O(1)\) for small load factors.
- Rehash as necessary
  - minimal cost.
Hash Advantages/Disadvantages

• Advantages:
  – Hash is faster than Binary Search Tree for searches.
    • O(1) versus O(logN)
  – Hash has no problems with ordered data.
    • Binary Search Tree gets unbalanced.
  – Hash does not require that data be comparable
    • binary search tree has to have <, > concepts.

• Disadvantages:
  – Hash can’t provide an ordering.
    • e.g. traversals make no sense. So wouldn’t want to store files systems this way!
  – Not a natural structure for some problems.