Lecture 7

Prof. Constantinos Daskalakis

CLRS: 11.4, 17.
So Far

- Hash table as dictionary
  - Insert/Search/Delete
- Collisions by chaining
  - Build a linked list in each bucket
  - Operation time is length of list
- Simple Uniform Hashing
  - Every item to uniform random bucket
  - \( n \) items in size \( m \) table \( \rightarrow \) average length \( n/m = \alpha \)
- Signatures: fast comparison
- Rolling Hash: fast sequence of hash’s
Dynamic Dictionaries

• In substring application, inserted all at once then scanned

• More generally, arbitrary sequence of insert, delete, find

• How do we know how big the table will get?

• What if we guess wrong?
  too small $\Rightarrow$ load high, operations too slow
  too large $\Rightarrow$ high initialization cost, consumes space, potentially more cache-misses

• Want $m=\Theta(n)$ at all times
Solution: Resize

• Start table at small constant size
• When table too full, make it bigger
• When table too empty, make it smaller

• How?
  ▪ Build a whole new hash table and insert items
  ▪ Recompute all hashes
  ▪ Recreate new linked lists
  ▪ Time spent to rebuild:
    \[(\text{new-size}) + \#\text{hashes} \times (\text{HashTime})\]
When to resize?

• **Approach 1**: whenever \( n > m \), rebuild table to new size
  - Sequence of \( n \) inserts
  - Each increases \( n \) past \( m \), causes rebuild
  - Total work: \( \Theta(1 + 2 + \ldots + n) = \Theta(n^2) \)

• **Approach 2**: Whenever \( n \geq 2m \), rebuild table to new size
  - *Costly inserts*: insert \( 2^i \) for all \( i \):
    - *These cost*: \( \Theta(1 + 2 + 4 + \ldots + n) = \Theta(n) \)
  - All other inserts take \( O(1) \) time – *why?*
  - Inserting \( n \) items takes \( O(n) \) time
  - Keeps \( m \) a power of 2 --- good for mod
Amortized Analysis

• If a sequence of \( n \) operations takes time \( T \), then each operation has amortized cost \( \frac{T}{n} \)
  ▪ Like amortizing a loan: payment per month

• Rebuilding when \( n \geq 2m \) \( \rightarrow \) some ops are very slow
  ▪ \( \Theta(n) \) for insertion that causes last resize

• But on average, fast
  ▪ \( O(1) \) amortized cost per operation

• Often, only care about total runtime
  ▪ So averaging is fine
Insertions+Deletions?

• Rebuild table to new size when \( n < m \)?
  - Same as bad insert: \( O(n^2) \) work

• Rebuild when \( n < m/2 \)?
  - Makes a sequence of deletes fast
  - What about an arbitrary sequence of inserts/deletes?
    • Suppose we have just rebuilt: \( m = n \)
    • Next rebuild a grow \( \rightarrow \) at least \( m \) more inserts are needed before growing table
      - Amortized cost \( O(2m / m)) = O(1) \)
    • Next rebuild a shrink \( \rightarrow \) at least \( m/2 \) more deletes are needed before shrinking
      - Amortized cost \( O(m/2 / (m/2)) = O(1) \)
Another Approach

• Algorithm
  ▪ Keep m a power of 2 (good for mod)
  ▪ Grow (double m) when \( n \geq m \)
  ▪ Shrink (halve m) when \( n \leq m/4 \)

• Analysis
  ▪ Just after rebuild: \( n=m/2 \)
  ▪ Next rebuild a grow \( \rightarrow \) at least \( m/2 \) more inserts
    • Amortized cost \( O(2m / (m/2)) = O(1) \)
  ▪ Next rebuild a shrink \( \rightarrow \) at least \( m/4 \) more deletes
    • Amortized cost \( O(m/2 / (m/4)) = O(1) \)
Summary

• Arbitrary sequence of insert/delete/find
• $O(1)$ amortized time per operation
OPEN ADDRESSING
$\mathcal{U}$: universe of all possible keys—huge set

$K$: actual keys—small set, but not known when designing data structure

Recall Chaining...
Open Addressing

- Different technique for dealing with collisions; does not use linked list
- Instead: if bucket occupied, find other bucket \( (m \geq n) \)
- For insert: **probe** a sequence of buckets until find empty one!
- \( h(x) \) specifies probe sequence for item \( x \)
  - Ideally, sequence visits all buckets
  - \( h: U \times [1..m] \rightarrow [1..m] \)

\( U \) - Universe of keys
\( m \) - Probe number
\( n \) - Bucket

Diagram: U - Universe of keys, m - Probe number, h - Function, Bucket
Open Addressing (example)

- $k$ maps to different indexes:
  - $h(k,1)$ collision
  - $h(k,2)$ collision
  - $h(k,3)$ collision
  - Free spot at $h(k,4)$
Operations

• Insert
  ▪ Probe till find empty bucket, put item there

• Search
  ▪ Probe till find item (return with success)
  ▪ Or find empty bucket (return with failure)
    • Because if item inserted, would use that empty bucket

• Delete
  ▪ Probe till find item
  ▪ Remove, leaving empty bucket
Problem with Deletion

• Consider a sequence
  ▪ Insert x
  ▪ Insert y
    • suppose probe sequence for y passes x bucket
    • store y elsewhere
  ▪ Delete x (leaving hole)
  ▪ Search for y
    • Probe sequence hits x bucket
    • Bucket now empty
    • Conclude y not in table (else y would be there)
Solution for deletion

• When delete x
  ▪ Leave it in bucket
  ▪ But mark it deleted --- store “tombstone”

• Future search for x sees x is deleted
  ▪ Returns “x not found”

• “Insert z” probes may hit x bucket
  ▪ Since x is deleted, overwrite with z
  ▪ So keeping deleted items doesn’t waste space
What probe sequence?
Linear probing

• $h(k,i) = h'(k) + i$ for ordinary hash $h'$
• Problem: creates “clusters”, i.e. sequences of full buckets
  ▪ exactly like parking
  ▪ Big clusters are hit by lots of new items
  ▪ They get put at end of cluster
  ▪ Big cluster gets bigger: “rich get richer” phenomenon
if $h(k,1)$ is any of these, the cluster will get bigger

i.e. the bigger the cluster is, the more likely it is to grow larger, since there are more opportunities to make it larger...
Linear probing

- $h(k,i) = h'(k) + i$ for ordinary hash $h'$
- Problem: creates “clusters”, i.e. sequences of full buckets
  - exactly like parking
  - Big clusters are hit by lots of new items
  - They get put at end of cluster
  - Big cluster gets bigger: “rich get richer” phenomenon
- For $0.1 < \alpha < 0.99$, cluster size $\Theta(\log n)$
- Wrecks our constant-time operations
Double Hashing

• Two ordinary hash functions $f(k)$, $g(k)$
• Probe sequence $h(k, i) = f(k) + i \cdot g(k) \mod m$
• If $g(k)$ relatively prime to $m$, hits all buckets
  ▪ E.g., if $m=2^r$, make $g(k)$ odd
  ▪ The same bucket is hit twice if for some $i, j$:
    $f(k) + i \cdot g(k) = f(k) + j \cdot g(k) \mod m$
    $\rightarrow i \cdot g(k) = j \cdot g(k) \pmod{m}$
    $\rightarrow (i-j) \cdot g(k) = 0 \pmod{m}$
    $\rightarrow m$ and $g(k)$ not relatively prime
(otherwise $m$ should divide $i-j$, which is not possible for $i, j<m$)
Performance of Open Addressing

• Operation time is length of probe sequence
• How long is it?
• In general, hard to answer.
• Introducing…
• “Uniform Hashing Assumption” (UHA):
  ▪ Probe sequence is a uniform random permutation of [1..m]
  ▪ (N.B. this is different to the simple uniform hashing assumption (SUHA))
Analysis under UHA

• Suppose:
  ▪ a size-m table contains n items
  ▪ we are using open addressing
  ▪ we are about to insert new item

• Probability first prob successful?
\[
\frac{m - n}{m} := p
\]

Why? From UHA, probe sequence random permutation
Hence, first position probed random
m-n out of the m slots are unoccupied
Analysis (II)

• If first probe unsuccessful, probability second prob successful?

\[
\frac{m - n}{m - 1} \geq \frac{m - n}{m} = p
\]

Why?
• From UHA, probe sequence random permutation
• Hence, first probed slot is random; the second probed slot is random among the remaining slots, etc.
• Since first probe unsuccessful, it probed an occupied slot
• Hence, the second probe is choosing uniformly from m-1 slots, among which m-n are still clean
Analysis (II)

• If first two probes unsuccessful, probability third prob successful?

\[ \frac{m - n}{m - 2} \geq \frac{m - n}{m} = p \]

• …

\[ \Rightarrow \text{every trial succeeds with probability} \geq p \]

expected number of probes till success? \( \leq \frac{1}{p} = \frac{1}{1 - \alpha} \)

e.g. if \( \alpha = 90\% \), expected number of probes is at most 10
Open Addressing vs. Chaining

• Open addressing skips linked lists
  ▪ Saves space (of list pointers)
  ▪ Better locality of reference
    • Array concentrated in m space
    • So fewer main-memory accesses bring it to cache
    • Linked list can wander all of memory

• Open addressing sensitive to $\alpha$
  ▪ As $\alpha \to 1$, access time shoots up
\[
\frac{1}{1 - \alpha}
\]
Open Addressing vs. Chaining

- Open addressing skips linked lists
  - Saves space (of list pointers)
  - Better locality of reference
    - Array concentrated in m space
    - So fewer main-memory accesses bring it to cache
    - Linked list can wander all of memory

- Open addressing sensitive to $\alpha$
  - As $\alpha \to 1$, access time shoots up
  - Cannot allow $\alpha > 1$

- Open addressing needs good hash to avoid clustering
ADVANCED HASHING

covered in recitation (for those who care)
Universal Hashing

• Get rid of simple uniform hashing assumption
• Create a family of hash functions
• When you start, pick one at random
• Unless you are unlucky, few collisions
  ▪ Adversary doesn’t know what hash you will use
  ▪ So cannot pick keys that collide in it
Universal Hash Family...

...is a family (set) of hash functions such that, for any keys $x$ and $y$, if you choose a random $h$ from the family, $\Pr[h(x) = h(y)] = 1/m$

Such a family produces few expected collisions

- $E[\text{collisions with } x] = E[\text{number of } y \text{ s.t. } h(x) = h(y)]$

  $$
  = E\left[ \sum_y 1_{h(x) = h(y)} \right] \\
  = \sum_y E\left[ 1_{h(x) = h(y)} \right] \quad \text{(linearity of } E) \\
  = \sum_y \Pr[h(x) = h(y)] \\
  = n/m
  $$
Universal Families Exist!

• Suppose m is a prime p
• Define \( h_{ab}(x) = a \cdot x + b \pmod{p} \)
• If \( a \) and \( b \) are random elements in \( \{0, \ldots, p-1\} \), then \( h_{ab}(x) \) is a universal family
  ▪ mod \( p \) is field, so you can divide/substract as well
  ▪ Pick two keys \( x \) and \( y \). What is the probability (over the choice of \( a, b \)) that the hashes of \( x \) and \( y \) collide?
  ▪ It has to be that \( a \cdot x + b = q \pmod{p} \) and \( a \cdot y + b = q \pmod{p} \), for some \( q \) in \( \{0, \ldots, p-1\} \)
  ▪ This is a linear system in \( a, b \)
    ▪ Two variables, two equations
  ▪ Unique solution---unique \( h_{ab} \) makes this happen
    ▪ Probability of choosing this \( h_{ab} \) is \( \frac{1}{p^2} \)
  ▪ Collide if \( h_{ab}(x) = h_{ab}(y) = q \) for some \( q \)
    ▪ hence overall probability of collision: \( \frac{p}{p^2} = \frac{1}{p} = \frac{1}{m} \)
• Justifies multiplication hash
Even Better

• Perfect Hashing
  ▪ Hash table with zero collisions
  ▪ So don’t need linked lists
• Can’t guarantee for arbitrary keys
• But if you know keys in advance, can quickly find a hash function that works
  ▪ E.g. for a fixed dictionary
Summary

• Hashing maps a large universe to a small range
• But avoids collisions
• Result:
  ▪ Fast dictionary data structure
  ▪ Fingerprints to save comparison time

• Next week: sorting