Dictionaries and Hash Tables

Outline

1. Dictionaries and Applications
2. Possible Implementations
3. Hash Tables (Chaining + variants, Linear Probing + variants, Cuckoo Hashing)

Dictionaries (aka associative arrays) maintain a set of key-value pairs

Examples: counters in Misra-Gries algorithm, words + definitions, symbol table of a compiler, etc.

Operations to support:
- insert(key)
- delete(key)
- lookup(key)

Different applications might call for different implementations that optimize some operations at the expense of others

Usually # of stored data records is much smaller than # of possible keys (data universe size). e.g. # of possible IP addresses in IPv6 is 2^{128}.

Goal: If there are n records, want to use $m = O(n)$ words of space.

a word lets you store 1 key or value ($O(\log n)$ bits)

Ideally $m = (1+\epsilon)\cdot n$ for ϵ small.

Let's call $\frac{n}{m}$ the <u>load</u>. (Loads close to 1 mean we are not wasting much space.

Possible implementations:
- self-balancing binary search tree. All operations $O(\log n)$ time worst case
- B-tree (a <u>non-binary</u> search tree, to optimize memory locality. Very well-studied/used in database community).

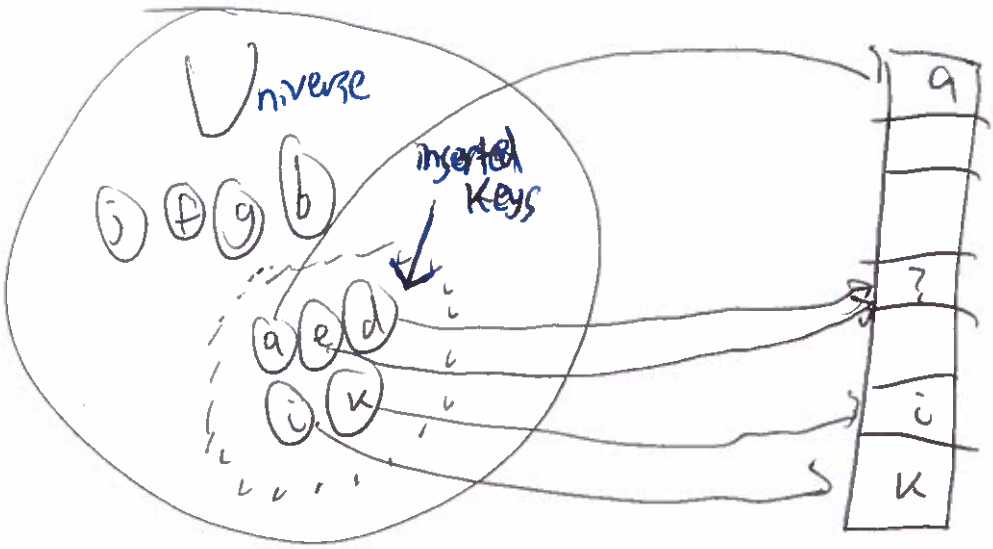
<u>Randomized</u> implementations:
- skip list (probabilistic hierarchy of linked lists)
- <u>Hash table</u>. Goal: constant time operations.

Hash Table idea: Have a table with $m = O(n)$ slots and

data universe $h: U \to [m]$. When we want to insert record (x, v),

we try to put it into slot $h(x)$. Implementations

differ on how they resolve collisions.



By birthday paradox collisions will occur with high probability if $m = o(n^2)$, i.e. to avoid collisions completely, you'd need a terabyte of space to store 1 MB of keys. How to resolve collisions?

Option 1: Chaining, A linked list is used for each slot to store all items mapped there.

Option 1': 2-choice hashing + chaining. Same as above, except each key is mapped to two slots and placed in the least loaded slot.

Option 2: Linear probing: Successively try slots $h(x)$, $h(x)+1 \pmod m$, $h(x)+2 \pmod m$ ... Other variants: Quadratic probing (on ith try, try slot $h(x) + c_1 i + c_2 i^2$). Double hashing (on ith try, try $h(x) + i \cdot h_2(x)$ where $h_2(x)$ is a second, independently chosen hash function).

Option 3: Cuckoo hashing, we'll get there

Pros and Cons:

- Chaining:
  - Pro: Can store more keys than there are slots in table
  - Cons:
    - Memory to store pointers for linked lists
    - Poor memory locality due to linked lists
    - Memory allocation/deallocation mechanism needed for linked lists

- Linear Probing: Pros:
  - <u>Great</u> memory locality. The dominant implementation in practice as a result.

  Cons:
  - Suffers from "clustering": long-ish runs of occupied cells will occur with high probability
  - Rather sensitive to choice/strength of hash function
  - Quadratic probing and Double hashing variants cannot support deletions (removing a key may make other keys undiscoverable). But they tend to mitigate clustering.

Runtime: <u>Chaining</u>: • Inserts take worst-case constant time (stick the inserted key at the front of the list for its slot).

• Lookups and deletes take <u>expected</u> constant time. But with high probability <u>some</u> operations will take $\Theta\left(\frac{\log n}{\log\log n}\right)$ time.

Reason: if you throw n balls at random into $\Theta(n)$ bins, the load of the heaviest bin will be $\Theta\left(\frac{\log n}{\log\log n}\right)$ with high probability (see first problem set).

<u>2-choice hashing + chaining</u>: • Max loaded slot will only contain $\Theta(\log\log n)$ keys with high probability.

$\boxed{\text{"The Power of Two Choices"}}$

<u>Intuition</u>: Suppose we throw n balls into n bins via the 2-choice process. Let p_i denote the fraction of balls in a bin with load $\geq i$. Let v_i denote the fraction of bins with load $\geq i$.

Note: $v_i \leq p_i$.

What is the probability ball $n+1$ has <u>both</u> bin choices of load $\geq i$? Answer: v_i^2. So we expect $p_{i+1} \leq v_i^2 \leq p_i^2$.

So if say $p_{18} \leq \frac{1}{2}$ then $p_{18+j} \leq \left(\frac{1}{2}\right)^{2^j}$.

Once $j = \log\log n$, $p_{18+j} \leq \left(\frac{1}{2}\right)^{2^{\log\log n}} = \left(\frac{1}{2}\right)^{\log n} = \frac{1}{n}$.

Linear probing: First analyzed by Knuth in 1963.

Considered to be the birth of analysis of algorithms

- Expected time for a successful search is

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

α is the load $\frac{n}{m}$

- Expected time for an unsuccessful search is

$$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right).$$

- With high probability the longest probe will have length $O(\log n)$

Hash functions: • All time bounds stated above assumed truly random hash functions. What is known about these hash table implementations' performance under 'realistic' hash functions?

- For the linear probing guarantees to hold, 5-wise independence suffice [Pagh, Pagh, Ruzic 2009]. But there are 4-wise independent hash families that do not [Patrascu + Thorup 2010].

- To get the $O\left(\frac{\log n}{\log\log n}\right)$ bound on the max-loaded bin in Chaining $O\left(\frac{\log n}{\log\log n}\right)$-wise independence suffices but this is unsatisfying. Evaluation time is $O\left(\frac{\log n}{\log\log n}\right)$ and space is $O\left(\frac{\log^2 n}{\log\log n}\right)$.

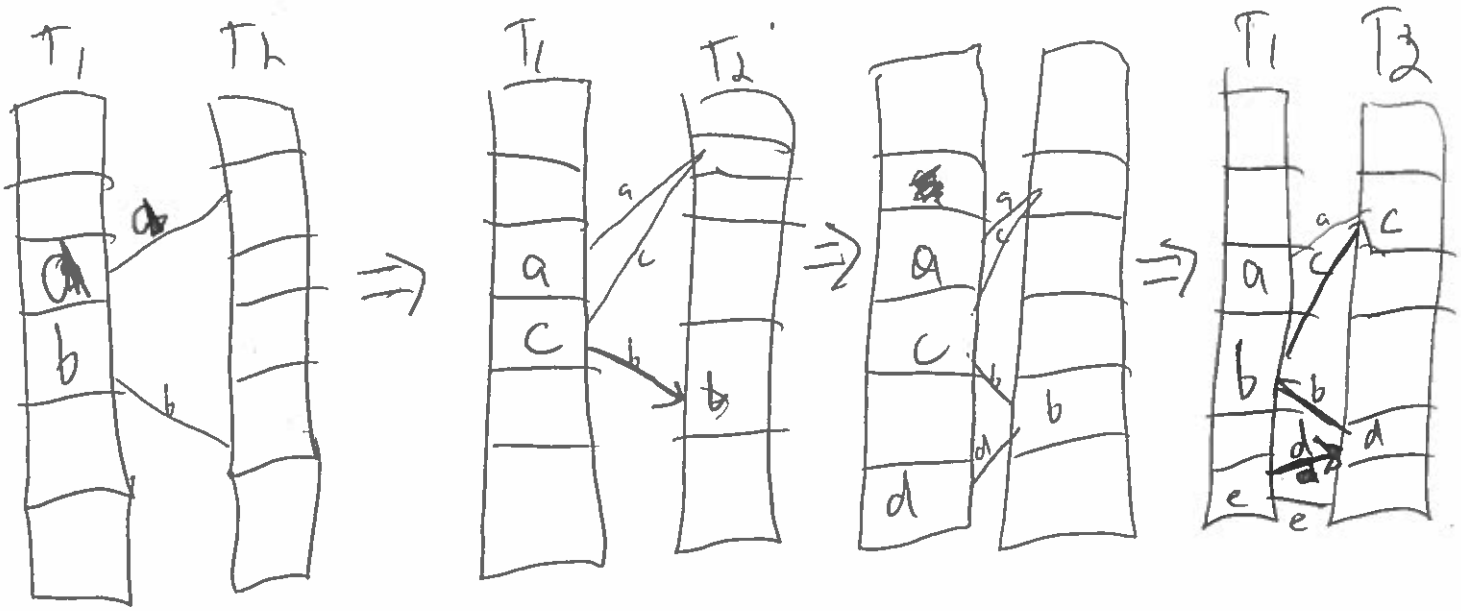
[CRSW 2011]: A hash family achieving this with $O(\sqrt{\log n})$ evaluation time and $O(\log^{3/2} n)$ bits of space.

Cuckoo Hashing: • Introduced by Pagh and Rodler in 2001.

Main pro: <u>Worst-case</u> constant-time lookups and deletions. Shifts complexity to insertions.

Idea is similar to 2-choice-hashing: Use two hash functions h_1, h_2 and two tables T_1, T_2 each w/ m slots. At all times, x will either be stored in $T_1[h_1(x)]$ or $T_2[h_2(x)]$. This is why lookups and deletes take $O(1)$ time.

Insertion Procedure: Try to place x in $T_1[h_1(x)]$. If this slot is empty, we are done. Otherwise, there is some key y occupying the cell and $h_1(x) = h_1(y)$. We try to move y to $T_2(h_2(y))$. If $T_2[h_2(y)]$ is empty we're done. Otherwise there is a z occupying $T_2(h_2[y])$. We kick z out & try to place it into $T_1(h_1(z))$. We continue until we succeed, or give up & rehash whole table. An insertion <u>fails</u> if it enters an infinite loop.



Cuckoo Hashing time bounds:

- Lookups, deletes take ≤ 2 timesteps, and allow parallelization.
- Insertions take $O(1)$-expected time and $O(\log n)$ worst case time w.h.p.

Intuition: Exactly $\frac{1}{2}$ of the $2n$ slots that exist between the two tables are full. So each time you go to move an item y from $T_1(h_1(y))$ to $T_2(h_2(y))$ or vice versa, there is probability $\frac{1}{2}$ that the slot is free. So after $c\log n$ moves, you'll have found an open slot with probability $1 - \frac{1}{2^{c\log n}} = 1 - \frac{1}{n^c}$.

Problem: The items you try to move in any sequence of insertions are not independent! Will provide a detailed proof sketch in a few minutes.

Variants of cuckoo hashing:

There are 3 major downsides to standard cuckoo hashing,

1) Need space $2n$ to store n items to ensure low failure probability (load just $\frac{1}{2}$).
2) Failure probability a bit high: $\frac{1}{n}$.
3) Although inserts take expected constant time, some will take $O(\log n)$ time w.h.p.

Ways to fix 1:

- d-ary cuckoo hashing. Use $d > 2$ hash functions & d tables. Now lookups and deletes take time d, but we can increase the load. For $d=3$, get load up to .91, for $d=4$ get up to 0.97. (Analysis is more complicated)
- Cuckoo hashing with buckets. Use two hash functions, but let each slot in both tables hold multiple items. Can handle loads closer to 1, but still not completely understood.

• Ways to fix 2). Cuckoo hashing with a stash (Kirsch, Mitzenmacher, Wieder 2009)

If an insertion fails, throw the item in a stash of size s.

Now lookups & deletes take time $2+s$, since you have to look at the whole stash.

But you can keep the stash in fast memory.

Reduces the failure probability to about $1-\frac{1}{n^s}$.

Intuition: each insertion fails with probability $O(\frac{1}{n^2})$ so if they were all independent you'd expect $\geq s$ to fail with probability about

$$\sum_{t \geq s} \binom{n}{t} \frac{1}{n^{2t}} \leq \sum_{t \geq s} \frac{1}{n^t} = O\left(\frac{1}{n^s}\right).$$

Things aren't independent but the analysis works out anyway

Ways to fix 3) Have a queue of insertions or deletions. When a new insertion or deletions of item x occurs, stick x in back of queue. Spend $O(1)$ time steps working on whatever operations are at the front of the queue. Argue that whp queue never grows larger than $O(\log n)$ operations, which allows us to ensure fast lookups into items still in the queue.

Proof sketch that cuckoo hashing inserts will all succeed with probability at least $1-\frac{1}{n}$.

- First, what is the "most obvious way" an insert can fail? Answer: 3 items x, y, z vying for the same two slots. What is the probability this happens?

$$\binom{n}{3} \left[n^2\right] \left(\frac{1}{n^2}\right) = \Theta\left(n^3 \cdot \frac{1}{n^2} \cdot \frac{1}{n^2}\right) = \Theta\left(\frac{1}{n}\right).$$

\# of ways to choose x, y, z

$\Pr(h_1(x) = h_1(y)$ and $h_2(x) = h_2(y))$

$\Pr(h_1(x) = h_1(z)$ and $h_2(x) = h_2(z))$.

So cuckoo hashing <u>will fail</u> with probability at least $\Theta\left(\frac{1}{n}\right)$. It turns out this event dominates the analysis.

- Model the hash table as a bipartite graph, called the cuckoo graph. There is a left node for each slot in Table 1 & a right node for each slot in Table 2. If a key x is in the table, we include an edge between $T_1[h_1(x)]$ and $T_2[h_2(x)]$ and label each edge with the key x it corresponds to.

An insert fails <u>iff</u> $|K|+1$ keys w.r.t. for k memory cells.
In the cuckoo graph this corresponds to a connected component with two or more cycles.

Failure probability for insertion of item x is thus

A: [# of bipartite multi-graphs for which x completes a second cycle in a component]

B: [# bipartite multi-graphs with n edges and n left-nodes and n right nodes]

$2^{2n \cdot \log n}$ as such a graph corresponds to a 1:1 to for $2n$ random numbers in n

prove any such graph can be encoded with $2n\log n - 2\log n$ bits, so must be at most

$\frac{B}{n^2}$ of them

union bound over all n inserts