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, word+definition, 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. 

Ideally $m=\lceil \log_2 n \rceil$ for $n$ small.

Let's call $\frac{n}{m}$ the load. ( 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 non-binary search tree, to optimize memory locality. Very well-studied/used in database community).

Randomized implementations:
- Skip list (probably hierarchy of linked lists)
Hash Table idea! Have a table with \( m = \Theta(n) \) slots and 
\( h : U \rightarrow [m] \). 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 2: 2-choice hashing + chaining. Same as above, except each key is mapped to two slots and placed in the least loaded slot.

Option 3: Linear probing. Successively try slots \( h(x), h(x) + 1 (\text{mod } m), h(x) + 2 (\text{mod } m), \ldots \); other variants: quadratic probing (with try try slot \( h(x) + (i^2 + 2i) \)), double hashing (with \( h(x) + i \cdot h_2(x) \) where \( h_2(x) \) is a second, independently chosen hash function).

Option 4: cuckoo hashing, we'll get there
Pros and Cons:

**Chaining:**
- **Pros:** 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:** Great 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 of hash function
  - Quadratic probing and double hashing variants cannot support deletions (removing a key may make other keys unfindable), but they tend to mitigate clustering.
Runtime: hashing - Inverse take worst-case constant time (stick the inverted key at the front of the list for its slot).

- Lookups and deletes take expected constant time. But with high probability some operators 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).

2-choice hashing + chaining: Max loaded slot will only contain \( \Theta(\log \log n) \) keys with high probability, via the power of two choices.

Intuition: Suppose we throw \( n \) balls into \( n \) bins. Let \( P_i \) denote the fraction of balls in a bin with load \( \geq \varepsilon \). Let \( V_i \) denote the fraction of bins with load \( \geq \varepsilon \).

Note: \( V_i \leq P_i \).

What is the probability ball \( n+1 \) has both bin choice choices of load \( \geq \varepsilon \)? Answer: \( V_i^2 \), so we expect \( P_{i+1} \leq \frac{V_i^2}{\varepsilon^2} \).

So if \( P_i \leq \frac{1}{2} \) then \( P_{i+j} \leq \left( \frac{1}{2} \right)^{2^j} \).

Once \( j = \log \log n \), \( P_{i+j} \leq \left( \frac{1}{2} \right)^{2^j} = \left( \frac{1}{2} \right)^{\log \log n} = \frac{1}{n} \).
Linear probing: First analyzed by Knuth in 1963, considered to be the birth of analysis of algorithms.

- Expected time for a successful search is
  \[ \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \]
  \( \alpha \) 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 suffices [Pagh, Pagh, Ruzic 2009]. But there are 4-wise independent hash families that do not [Patarasuk, Thorup 2012].

- To get the \( O(\log \frac{1}{\log \log n}) \) bound on the max-loaded bin in Chany, \( O(\frac{\log h}{\log \log n}) \)-wise independence suffices but this is unsatisfactory.
  Evaluation time is \( O(\frac{\log h}{\log \log n}) \) and space is \( O(\frac{\log h}{\log \log n}) \).
  [CRSW 2011]. A hash family achieving this with \( O(\log h) \) evaluation time and \( o(\log n) \) bits of space.

Main Property: **Worst-case** constant-time lookups and deletions, shift 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 with \( m \) slots. At all times, \( x \) will either be stored in \( T_1[h_1(x)] \) or \( T_2[h_2(x)] \), that is why looking and deleting 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 \( y \) 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 move \( z \) out and try to place \( y \) into \( T_1[h_1(y)] \).

We continue until we succeed or give up and restart whole state.

An insert fails if it enters an infinite loop.
Cuckoo Hashing time bounds:

- Lookups, deletes take $\leq 2$ timesteps, and allow parallelization.
- Inserts take $O(1)$ expected time and $O(\log n)$ worst case time.

Intuition: Exactly $\frac{1}{4}$ of the $\frac{m}{2}$ slots that exist between the two tables are filled. So each time you try to move an item $y$ from $T_i(h_i(y))$ to $T_{i+1}(h_{i+1}(y))$ or $\text{inversly}$, there is probability $\frac{1}{4}$ that the slot is free. So after $\log n$ moves, you'll have found an open slot with probability $1 - \left(1 - \frac{1}{4}\right)^{\log n} = 1 - \frac{1}{n}^\frac{\log n}{2}$.

Problem: The items you try to move in any sequence aren't independent. Will provide a more refined analysis few minutes.

Variants of Cuckoo Hashing:

There are 3 major downsides to standard cuckoo hashing:

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

Ways to fix it:
- d-ary cuckoo hashing: Use $d > 2$ hash functions for table.
  - Now lookups and deletes take $\log_d n$ steps, but we can increase the load.
  - For $d=3$, get load up to 0.91, for $d=4$ get up to 0.97. Analyzing 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, 2008)

If an insertion fails, throw the item in a stash of sizes.
Now lookups and deletes take times, since you have to look at the whole stash.
But you can keep the stash in fast memory.

Reduce the failure probability to about $1 - \frac{1}{k}$.

In general, each insertion fails with probability $O(\frac{1}{h^2})$, so if they were all independent, you'd expect $\frac{1}{k}$ to fail with probability

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

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 deletion 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 the 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 succeed with probability at least $1 - \frac{1}{n}$.

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

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

So cuckoo hashing will fail 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 $T_1[x]$ and $T_2[x]$. And label each edge with the key $x$ it corresponds to.
An matrix  \( f \) is a non-negative \( n \times n \) matrix if and only if for every \( k \) memory cells in the cycle or graph, the corresponding component with two or more cycles.

Failure probability for insertion of item \( x \) is thus

\[
A = \left[ \text{# of bipartite multi-graphs for which } x \text{ complete as second cycle in a component} \right]
\]

\[
B = \left[ \text{# bipartite multi-graphs with } n \text{ edges and } n \text{ left nodes and } n \text{ right nodes} \right]
\]

prove such such graph can be encoded with \( 2 \) size-2 log bits, so must be at most

\[
\frac{B}{n^d}
\]

union bound over all \( m \) such

\[
\log_{n} \text{ high as such a graph corresponds to a list of } \]