Hash Tables

Observation: Trees have logarithmic access costs

- **Can we do better?**

Idea: Buckets

- **Partition the data according to a simple, predictable, deterministic pattern**
- **Summary Idea: Assume an f(x) that gives you a number between 1 and N**
  - e.g., "first letter" or "first k bits"0
  - Allocate N pages, use f(key) to figure out which page a record is supposed to live on
- **Pros**
  - Fast: O(1) page acesses (ideally)
- **Cons**
  - Need to pick N correctly
  - Clustering: Data is generally not uniformly distributed
    - Class names: “X”, “S” common letters: “W” completely empty

Idea: Pick a Deterministic “Reshuffling”

- **Hash Functions: h(x) -> Transform any x into a pseudo-random value**
  - **Pseudo-Random**: Statistically unpredictable output between 0 and $2^{\{\text{\# of hash bits}\}}$-1
  - **Deterministic**: h(x) is always the same
- **Adaptation: Modulus Operator Makes #s between 1 and N**
  - % = Modulus = Remainder after Division
    - 5 % 2 = 1
    - 5 % 3 = 2

- 6 % 3 = 0
- 7 % 3 = 1
- 8 % 3 = 2

- If h(x) gives you a number between 0 and [Some arbitrarily big number]
  - h(x) % N gives you a number between 0 and N-1
  - As long as N << [Some arbitrarily big number], the result is still "random enough"
    - Deviation from uniform random capped at N / [Some arbitrarily big number]
    - Unless [Some arbitrarily big number] % N = 0… then randomness perfectly preserved

- **Overall Solution:**
  - Allocate N pages
  - h(key) % N tells you on which page the record with 'key' lives
  - Use "overflow pages" to handle cases where you need to put too much data in one page.
- **Pros**
  - Fast: O(1) page acesses (ideally)
  - Data is distributed **more** uniformly
- **Cons**
  - Only supports == tests
  - We still don't know how to pick N… and what if the "best" N changes?

Idea: "Dynamic" Hashing

- **Problem: Changing N requires re-hashing <u>everything</u>**
  - Example:

    ```
    def h(x):
      return x; # Bad, but easy "hashing" fn
    ```

  - Data: 1, 2, 5, 8, 9, 11

    - Now: N = 5
        - 1 -> 1, 2 -> 2, 5 -> 0, 8 -> 3, 9 -> 4, 11 -> 1
    - Change: N to 6
        - 1 -> 1, 2 -> 2, 5 -> 5, 8 -> 2, 9 -> 3, 11 -> 5
- **Observation: Jumping between multiples of N make reshuffling easier**
    - If h(x) % 5 = 4
    - Then h(x) % 10 = Either 4 or 9
- **Decide how to split on a bit-by-bit basis:**
    - Use 1 bit (2 pages), 2 bits (4 pages), 3 bits (8 pages), etc...
    - But make the decision on a page-by-page basis
    - Use an "index" that tracks which pages correspond to which hash buckets
- **If you need to split a page**
    - Check to see if you need to double the number of hash buckets
        - If so, clone the index: Buckets N to 2N-1 start off pointing to the same pages as Buckets 1 to N-1
    - Allocate a new page
    - Re-hash the contents of the page, using one more bit than before.
        - Records that have a 1 for the extra bit go to the new page, records with a 0 stay in place
    - Point the appropriate index entry(ies) at the new page
- **The same happens in reverse to merge two pages together**
- **To pull this off, you need to track...**
    - The number of buckets in the index
    - Which pages have been allocated
    - For each allocated page, how many bits of hash are being used for records on that page.