# Hashing

We again consider the dictionary operations: Search, Delete, Insert.

- $\log_2 n$ comparisons are necessary to do a search for an element if our method uses only comparisons as operations on data.

- If we are allowed to perform *some other operations* on data value. We hope to do search faster.

**Example.** If values in range $1, ..., n$, we can use data also as address in $A[1...n]$ So we can do insert, delete, search all in $O(1)$ time.

We could apply the method (Hashing) of above example, when $n$ is small. What if range of data value is large? This lead us to hashing. We can try:

- map keys uniformly to available space
- do something about "collision" (2 keys mapped to same place).

When we perfomr mapping, here is a little guideline:

- use a "random" (looking) hash function
- use all bits of data value

**Example.** Some good hash functions.

- First half $\vee$ second half. (as bit string)
- Exclusive or several parts (as bit string)
- key modulo table size (as integer) (making table size a <u>prime</u> is a good choice.)

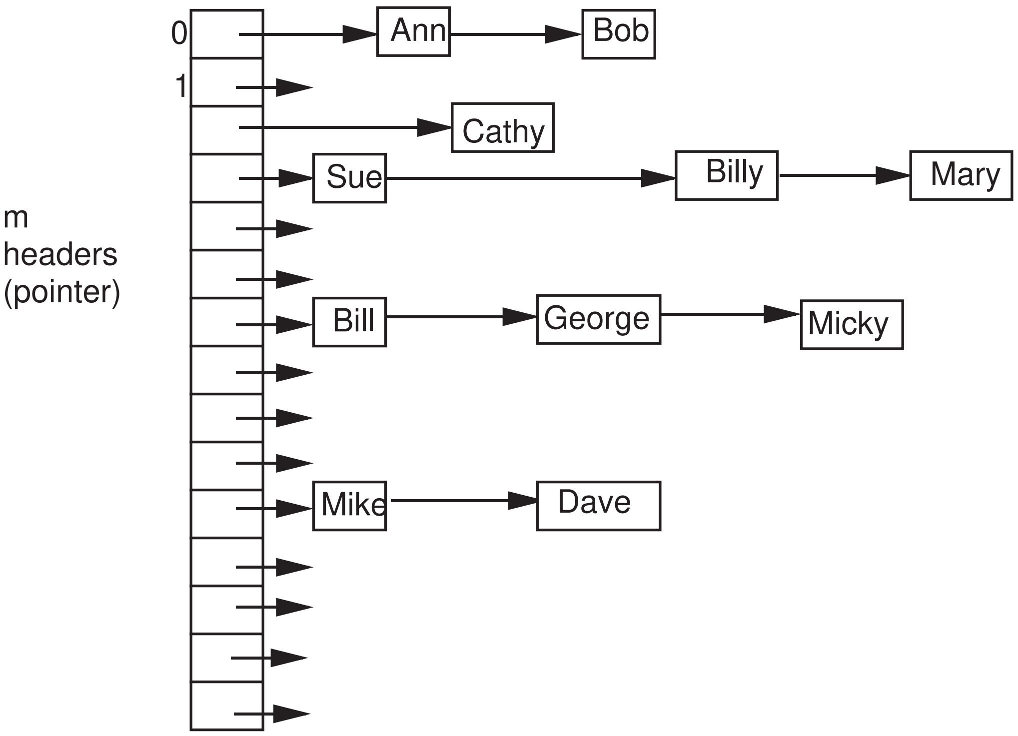
**Perfect Hash Functions**: 1 key maps to 1 place in table, without collision (a 1-— correspondence).

For a static tabla, such a perfect hash function is possible, but very costly.

The main issue then: assume a uniform mapping by (imperfect) hash function and look at *conflict resolution*: (1) chaining and (2) open addressing.

# Conflict Resolution

## (1) Chaining




Separate Chaining

1A. Separate Chaining

- Table entries are pointers to a linked list of nodes

- Advantage: makes table size less crucial than a scheme under which all must fit in a fixed size table - too many entries makes operations costly.

**Worst Case Cost:** $O(n)$, like a linked list (when all collide at the same spot.)

**Average Case Cost:** Let m = size of table, n = # elemenets in table, $\alpha = \frac{n}{m}$ (load factor). Next page.

- Successful Search (or Deletion)

$$S = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$$

(Explanation:, 1 step is needed to find the header. On average, out of the other $n-1$ elements, $n-1/m$ elements are expected to be in same list as the target element, and 1/2 are ahead of the target element.)

- Unsuccessful Search (or insertion) (count 1 if list is empty).

$$U = \frac{n}{m} + \left(1 - \frac{1}{m}\right)^n \approx \alpha + e^{-\alpha}$$

expected # elements in a list

probability $\left(1-\frac{1}{m}\right)$ is prob. 1 element does not fall into this list

Examples: $\alpha = \frac{1}{4}, U = 1.03$.
$\alpha = \frac{3}{4}, U = 1.22$.
$\alpha = 1, U = 1.37$.

## Getting Slightly Smarter

- Keep lists in sorted order so the unsuccessful search is faster

$$U = \left(1 - \frac{1}{m}\right)^n + \frac{n}{2m}$$

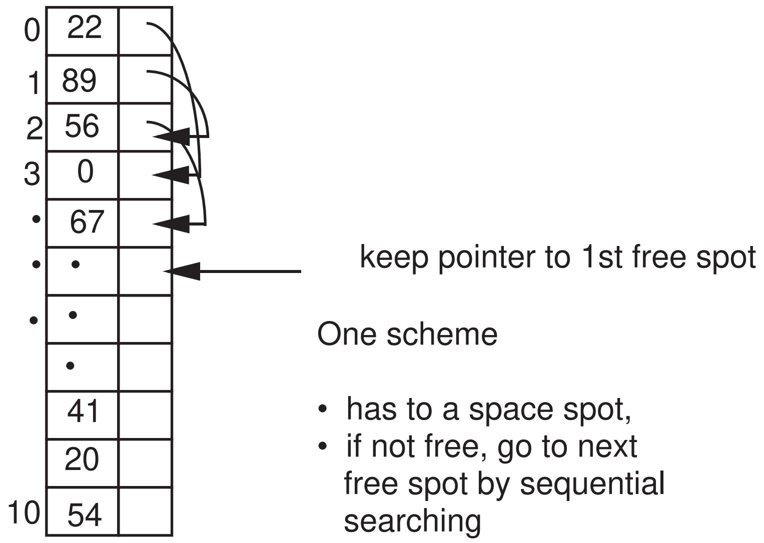
empty list as before

go half way

- Use self-organizing lists (move to front)

- Use balanced binary search tree ...

1B. Coalesced Chaining. Use no extra space.




• We can achieve search cost of separate chaining in a table at cost of complication in insertion scheme

• Deletions are painful.