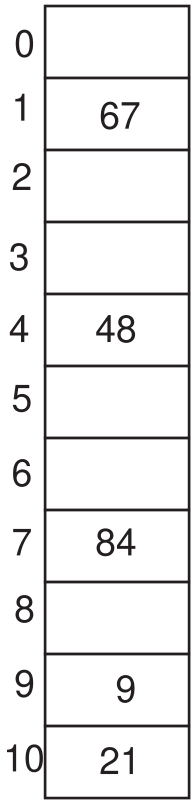
(2) Open Addressing.

Idea: How can we avoid extra space for pointers? hash directly into the table and resolve conflict by some scheme to look "elsewhere" in the hash table.



Most obvious: linear probing - if a conflict on insert, scan ahead to next open spot (with wrapping).

h(x) = x mod 11

Example. Assume locations 1,4,7,9 in the table are occupied.
Insert 29: $h(29) = 7$ conflict, next spot is 8.
Insert 18: $h(18) = 7$, conflict ,..., insert at 0.

To Search: map to primary location, then scan till element found or an empty spot is encountered. So when the search is unsuccessful, it is very bad to meet a big block of occupied cells.

Secondary clustering (also called "primary clustering" in some books): A bigger block of occupied locations has greater probability on increasing in length as a result of insertion. (I.e. bad situations are more likely getting worse).

Expected Cost for
Successful search $\approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$. It is $\Theta(\sqrt{m})$ when table is almost full.
Unsuccessful search or insert $\approx \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$. $\Theta(m)$ when table is almost full.

Here $n = \#$ elements, $m =$table size, $\alpha = n/m$.

Deletions.

(a) Mark as deleted, reuse on later inserts. (Unsuccessful searches and future insertions will become $\Theta(n)$ even for small α).

(b) Scan ahead, move elements back until a (true) empty spot is found.

- Costly
- Need to be careful - pretty confusing.

Avoiding Secondary Clustering

Try:

- Going i steps forward on i-th probe?
  - Does not guarantee we hit all positions quickly.
  - Not that effective in separating
  - In fact, still has the same problem.
- Ideal

key $\Longrightarrow$ one of the m! permutations of the table locations, uniformly. (so, we first use 1st location, if collide, use 2nd ,...).

Nice, but how?

Double hashing. An effective way to achieve a permutation similar to a "random" one:

- We use 2 hash functions
    - $h_1(key)$ produces the initial probe location.
    - $h_2(key)$ produces step size.

Example.
$h_1(key) = \text{key mod11} = c_1$
$h_2(key) = (\text{key mod10}) + 1 = c_2$

So we probe at locations, $c_1, c_1+c_2, c_1+2c_2, ...,$

Problem: Could we miss an empty spot in table?

- If $h_2(key)$ has a common factor with m, we won't look at all locations in trying to insert. That is, even table is not full, it could be full at locaitons, $c_1, c_1 + kc_2$, for all k. So we will not be able to insert to a non-empty table!

But if we make sure $h_2(key)$ and m are relatively prime (this is easy if we make m prime), then the above problem will be solved. Why? Think about it.

$h_2(key) = (\text{key mod } (m-2)) + 1$ works well for h_2.

- Obviously the permutations generated by double hashing are not the $m!$ "random" ones. (Since we go forward at fix step $h_2(key)$.) But the effects seem to be very similar.

- For random probing (proved) and double hashing (experiments), the average costs are:

Successful Search $\approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \xrightarrow{\text{full}} \Theta(\ln m)$

Unsuccessful Search $\approx \frac{1}{1-\alpha} \xrightarrow{\text{full}} \Theta(m)$

- An extra trick: Remember the <u>longest</u> successful search <u>length</u> in table. Thus if no success in that many probes, stop.