Open Address Hashing

Adapted from the CLRS book slides

OPEN ADDRESSING

Idea

- Store all elements in the hash table itself.
- Each slot contains either an element or NIL.
- The hash table can fill up, but the load factor can never be > 1.
- How to handle collisions during insertion:
  - Determine the element’s “first-choice” location in the hash table.
  - If the first-choice location is unoccupied, put the element there.
  - Otherwise, determine the element’s “second-choice” location. If unoccupied, put the element there.
  - Otherwise, try the “third-choice” location. And so on, until an unoccupied location is found.
  - Different elements have different preference orders.

OPEN ADDRESSING (continued)

- How to search:
  - Same idea as for insertion.
  - But upon finding an unoccupied slot in the hash table, conclude that the element being searched for is not present.
- Open addressing avoids the pointers needed for chaining. You can use the extra space to make the hash table larger.

OPEN ADDRESSING (continued)

More specifically, to search for key k:

- Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a ***probe***.
- If slot $h(k)$ contains key k, the search is successful. If this slot contains NIL, the search is unsuccessful.
- If slot $h(k)$ contains a key that is not k, compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.).
- Keep probing until either find key k (successful search) or find a slot holding NIL (unsuccessful search).
- Need the sequence of slots probed to be a permutation of the slot numbers $\langle 0, 1, \ldots, m-1 \rangle$ (so that all slots are examined if necessary, and so that no slot is examined more than once).
- Thus, the hash function is $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$.
- The requirement that the sequence of slots be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ is equivalent to requiring that the ***probe sequence*** $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- To insert, act as though searching, and insert at the first NIL slot encountered.

PSEUDOCODE FOR INSERTION

- HASH-INSERT either returns the slot number where the new key k goes or flags an error because the table is full.

```
HASH-INSERT$(T, k)$
1  $i = 0$
2  repeat
3          $q = h(k, i)$
4          if $T[q] == \text{NIL}$
5              $T[q] = k$
6              return $q$
7          else $i = i + 1$
8  until $i == m$
9  error “hash table overflow”
```

PSEUDOCODE FOR SEARCHING

- HASH-SEARCH returns either the slot number where the key k resides or NIL if key k is not in the table.

```
HASH-SEARCH($T, k$)
$i = 0$
repeat
    $q = h(k, i)$
    if $T[q] == k$
        return $q$
    $i = i + 1$
until $T[q] ==$ NIL or $i == m$
return NIL
```

DELETION

Cannot just put NIL into the slot containing the key to be deleted.

- Suppose key k in slot q is inserted.
- And suppose that sometime after inserting key k, key k' was inserted into slot q', and during this insertion slot q (which contained key k) was probed.
- And suppose that key k was deleted by storing NIL into slot q.
- And then a search for key k' occurs.
- The search would probe slot q *before* probing slot q', which contains key k'.
- Thus, the search would be unsuccessful, even though key k' is in the table.

DELETION (continued)

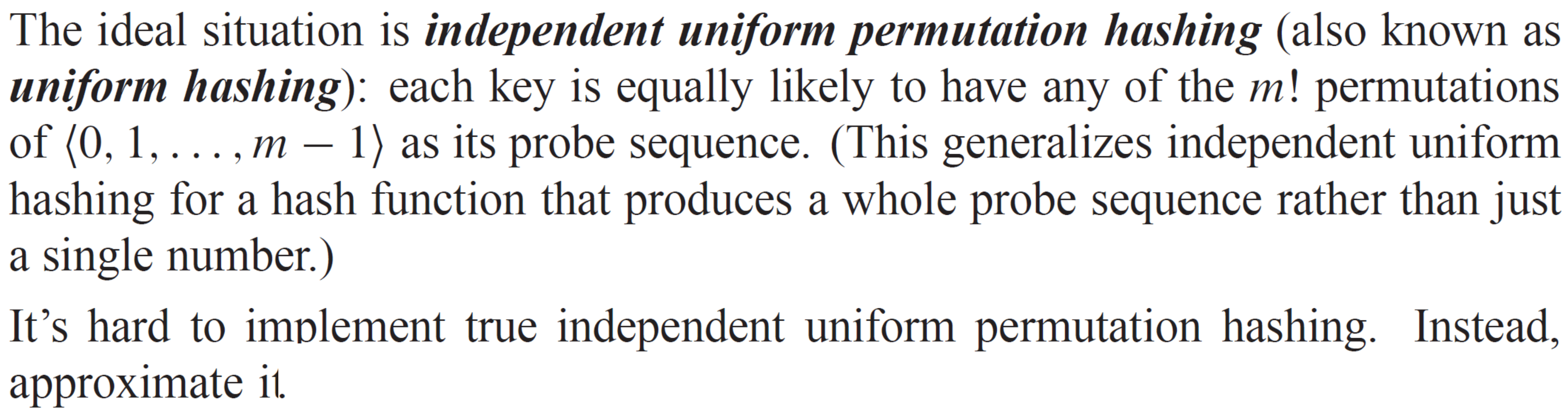
Solution: Use a special value DELETED instead of NIL when marking a slot as empty during deletion.

- Search should treat DELETED as though the slot holds a key that does not match the one being searched for.
- Insertion should treat DELETED as though the slot were empty, so that it can be reused.

The disadvantage of using DELETED is that now search time is no longer dependent on the load factor α.

A simple special case of open addressing, called linear probing, avoids having to mark slots with DELETED.

HOW TO COMPUTE PROBE SEQUENCES



The ideal situation is ***independent uniform permutation hashing*** (also known as ***uniform hashing***): each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence. (This generalizes independent uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It's hard to implement true independent uniform permutation hashing. Instead, approximate it.

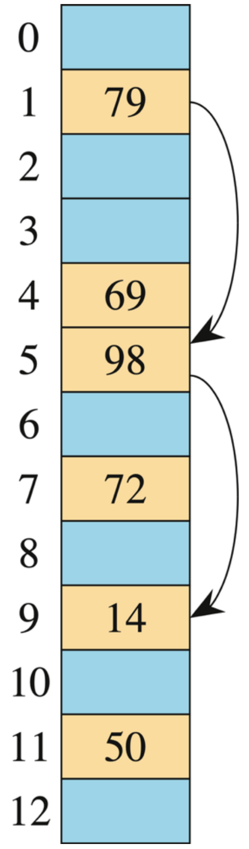
DOUBLE HASHING

Uses auxiliary hash functions h_1, h_2 and probe number i:

$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$.

The first probe goes to slot $h_1(k)$ (because i starts at 0). Successive probes are offset from previous slots by $h_2(k)$ modulo $m \Rightarrow$ the probe sequence depends on the key in two ways.

Example: $m = 13$, $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$, inserting key $k = 14$. First probe is to slot 1 ($14 \bmod 13 = 1$), which is occupied. Second probe is to slot 5 ($((14 \bmod 13) + (1 + (14 \bmod 11))) \bmod 13 = (1+4) \bmod 13 = 5$), which is occupied. Third probe is to slot 9 (offset from slot 5 by 4), which is free, so key 14 goes there.



DOUBLE HASHING (continued)

Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

- Could choose m to be a power of 2 and h_2 to always produce an odd number.
- Could let m be prime and have $1 < h_2(k) < m$.

  Example: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$ (as in the example above, with $m = 13$, $m' = 11$).

LINEAR PROBING

Special case of double hashing.

Given auxiliary hash function h', the probe sequence starts at slot $h'(k)$ and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0.

Given key k and probe number i $(0 \leq i < m)$, $h(k, i) = (h'(k) + i) \bmod m$.

The initial probe determines the entire sequence $\Rightarrow$ only m possible sequences.