The Cost of Searching

Given a collection of N equally-likely data values, any search algorithm that proceeds by comparing data values to each other must, on average, perform at least $\Theta(\log N)$ comparisons in carrying out a search.

There are several simple ways to achieve $\Theta(\log N)$ in the worst case as well, including:

- binary search on a sorted array
- search in a balanced binary search tree
- search in a skip list

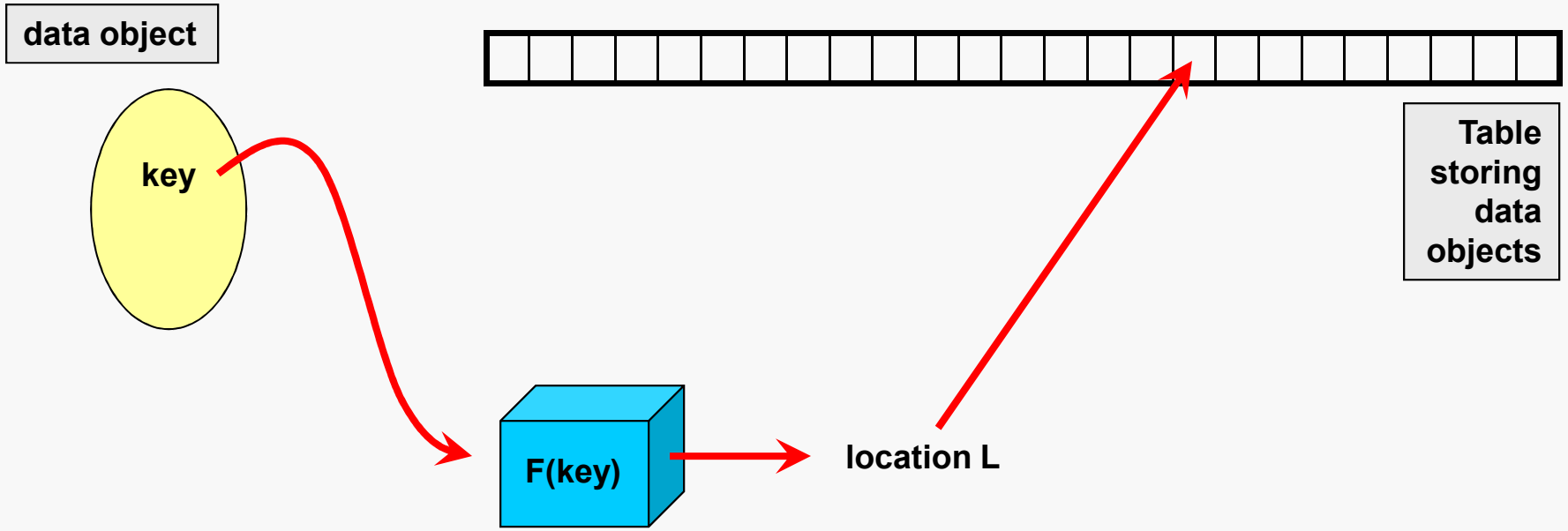
But, is there some way to "beat" the limit in the theoretical statement above?

There seem to be two possible openings:

- what if the data values are not equally-likely to be the target of a random search?
- what if the search process does not compare data elements?

In either case, the theorem would not apply…

Hash Table Concept



If:

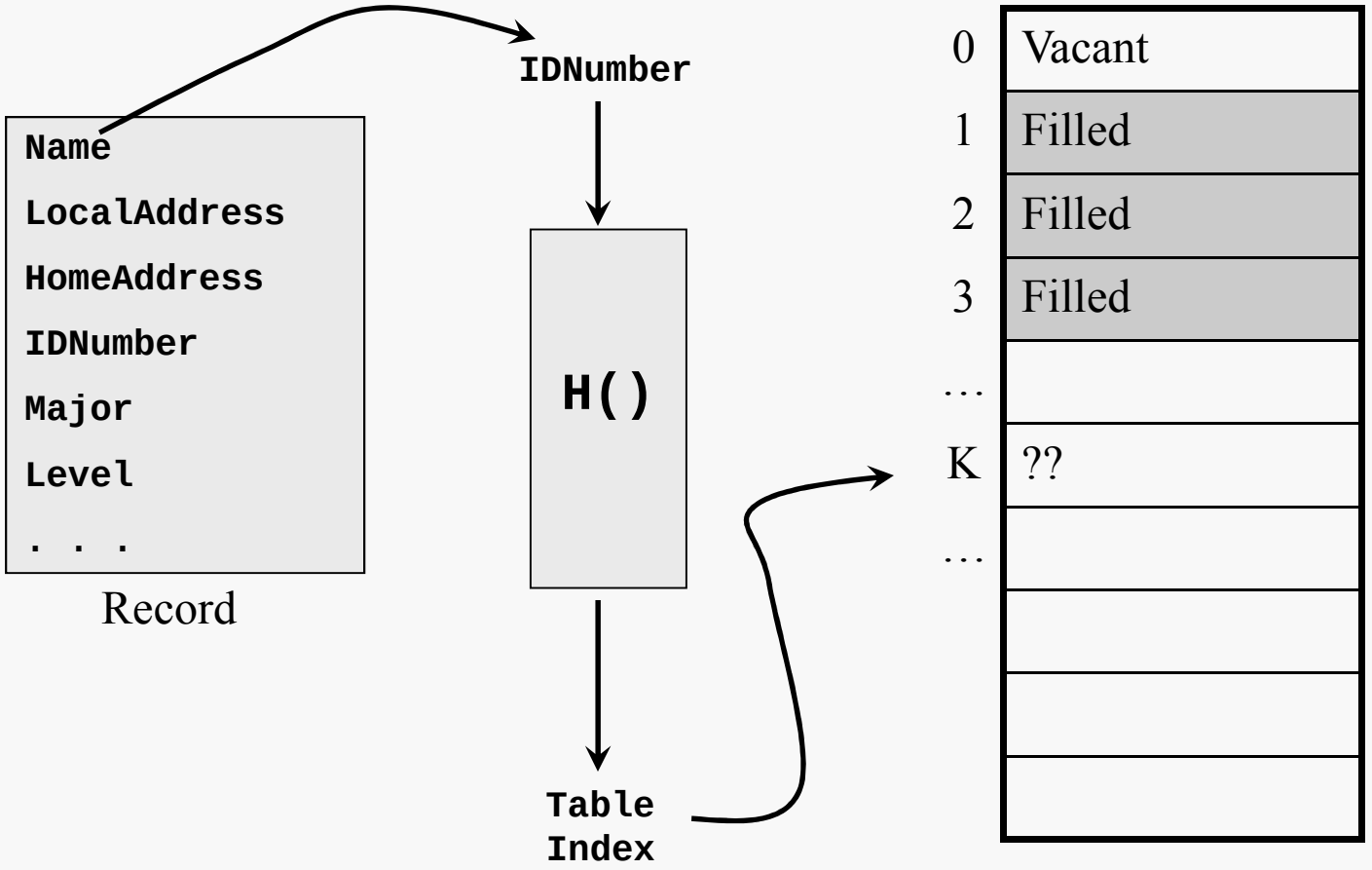
the function that computes the location has $\Theta(1)$ cost,

and the container storing the collection supports random access with $\Theta(1)$ cost,

then we would have a total search cost of $\Theta(1)$.

Hash Table Insertion

Simple insertion of an entry to a hash table involves two phases:



The appropriate record key value must be hashed to yield a table index. If that slot is vacant, the record is inserted there. If not, we have a *collision*…

Resolving Collisions

When collisions occur, the hash table implementation must provide some mechanism to resolve the collision:

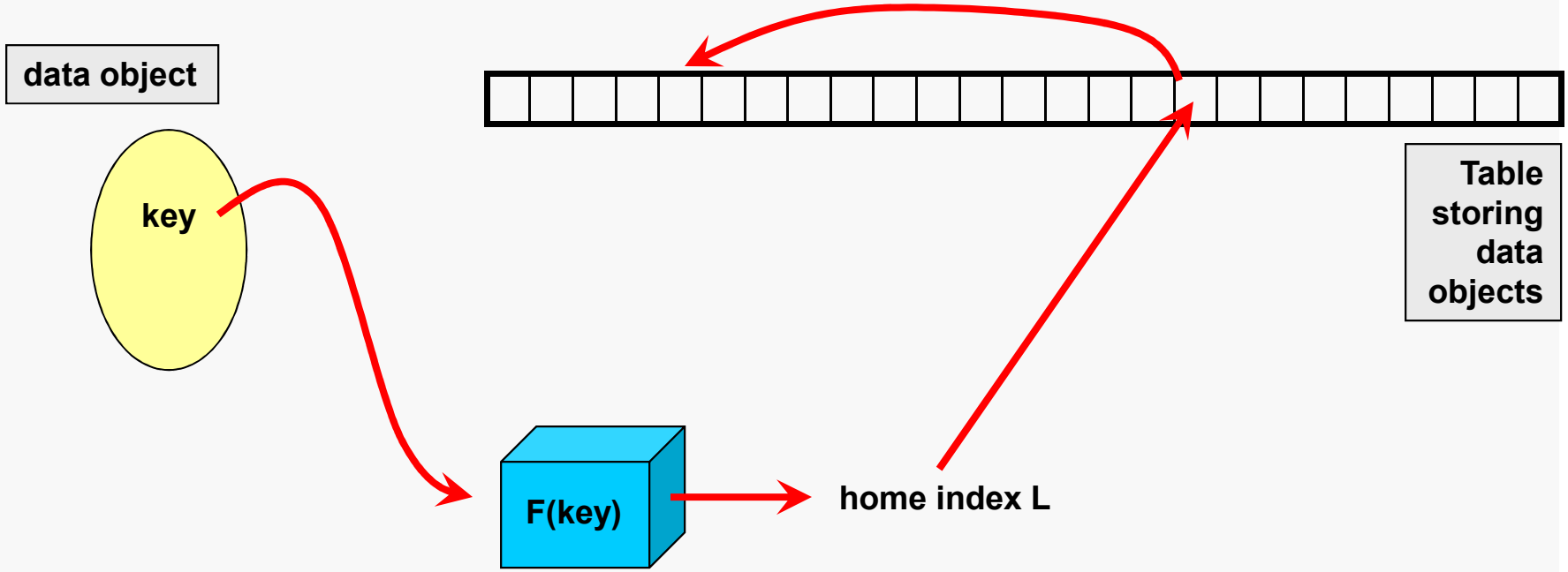
- *no strategy*: just reject the insertion. Unacceptable.

- *open hashing*: place the record somewhere other than its home slot
  - requires some method for finding the alternate location
  - method must be reproducible
  - method must be efficient
  - aka *hashing with probing*

- *chaining*: view each slot as a container, storing all records that collide there
  - requires an appropriate, efficient container for each table slot
  - overhead is a concern (e.g., pointers needed by container, search cost in slot)
  - aka open chaining

Open Hashing

If the home slot for the record that is being inserted is already occupied, then simply chose a different location within the table:

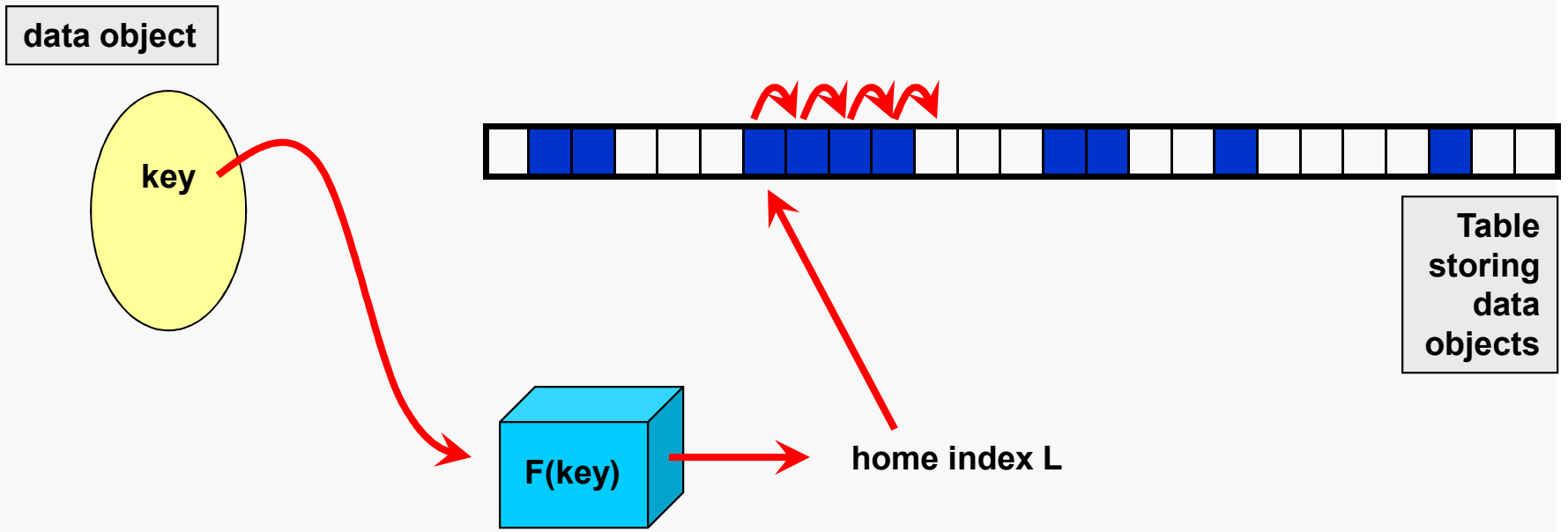


But… how do we choose this alternate location?

The technique must be reproducible, and on average be cheap.

Linear Probing

Linear probing involves simply walking down the table until an empty slot is found:

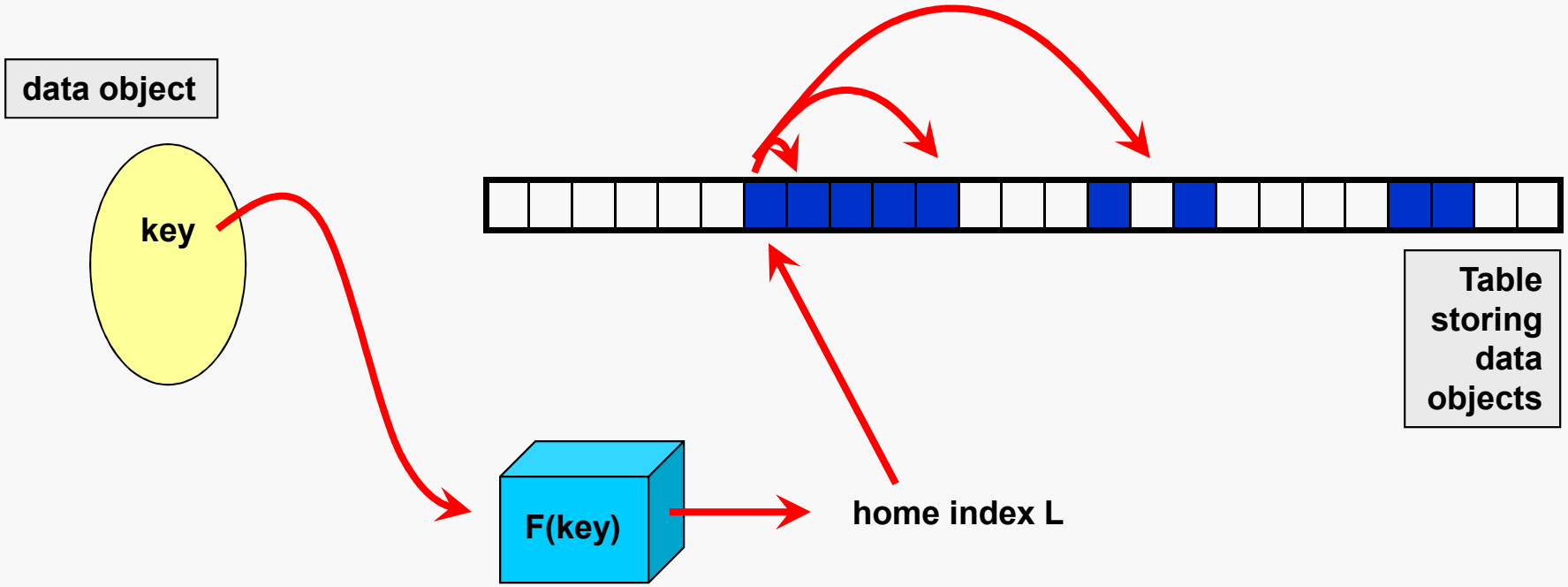


Mathematically, we iterate the following formula until we find an available slot:

$$Next(k) = (F(key) + k)\%TableSize$$

Quadratic Probing

Quadratic probing uses a formula that produces more "scattering":

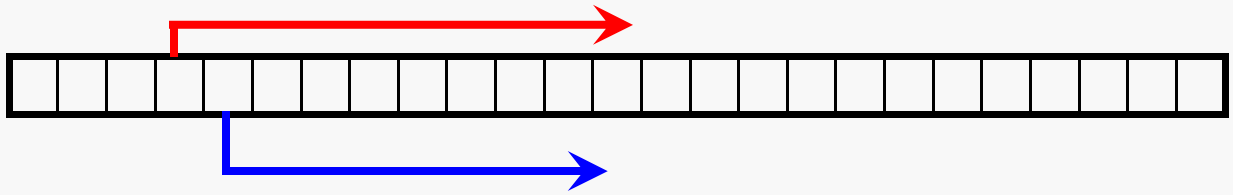


Mathematically, we iterate the following formula until we find an available slot:

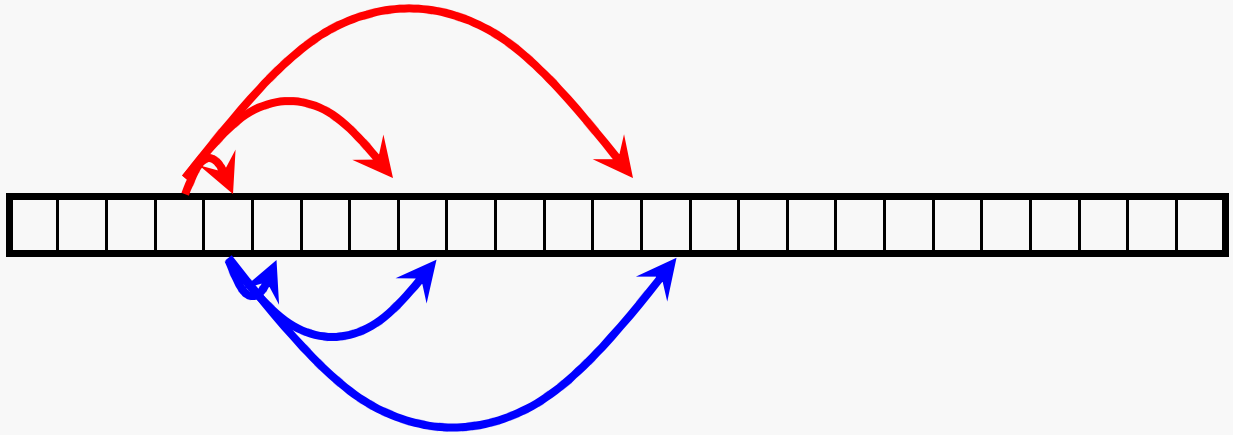
$$Next(k) = (F(key) + k^2)\%TableSize$$

Comparison

Linear probing tends to produce "runs" of adjacent filled cells when collisions occur:



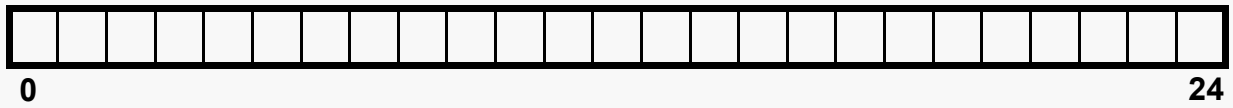
This is known as *clustering*.



Quadratic probing doesn't promote clustering… but…

Comparison

Quadratic probing doesn't visit all the slots in the table:



Suppose that we have a home index of 17, then quadratic probing will generate the following sequence of slot indices:

```
17 +   1      18
   +   4      21
   +   9       1
   +  16       8
   +  25      17
   +  36       3
   +  49      16
   +  64       6
   +  81      23
   +  100     17
   +  121     13
   +  144     11
   +  169     11
   +  196     13
   +  225     17
   . . .
```

Quadratic Probing Revisited

Using the basic formula: $Next(k) = (F(key) + k^2)\%TableSize$

If the number of slots in the table is prime, we can prove that the first TableSize/2 locations examined are unique…

If the number of slots in the table is a prime of the form 4k + 3 and we use alternating signs for the steps then every slot in the table will be reached, if necessary…

Double Hashing

Another way to resolve collisions is to pick a second hash function, say G() and then generate a sequence of probe slot index values by:

$$Next2(k) = (k \cdot G(key))\%TableSize$$

Take the earlier example for quadratic probing, and assume that G(key) is 7; we would then get the probe sequence:

```
17 (home)
24
 6
13
20
 2
 9
16
23
 5
12
19
 1
 8
15
. . .
```

Good: we only have to evaluate G() once and then do a single multiplication and mod to find the next probe slot.

Bad:

- if G(key) is 0
- how to guarantee this won't suffer from same defect as pure quadratic probing?

Deletion with Probing

Deleting a record poses a special problem: what if there has been a collision with the record being deleted? In that case, we must be careful to ensure that future searches will not be disrupted.

Solution: replace the deleted record with a "tombstone" entry that indicates the cell is available for an insertion, but that it was once filled so that a search will proceed past it if necessary.

Problem: this increases the average search cost since probe sequences will be longer that strictly necessary.

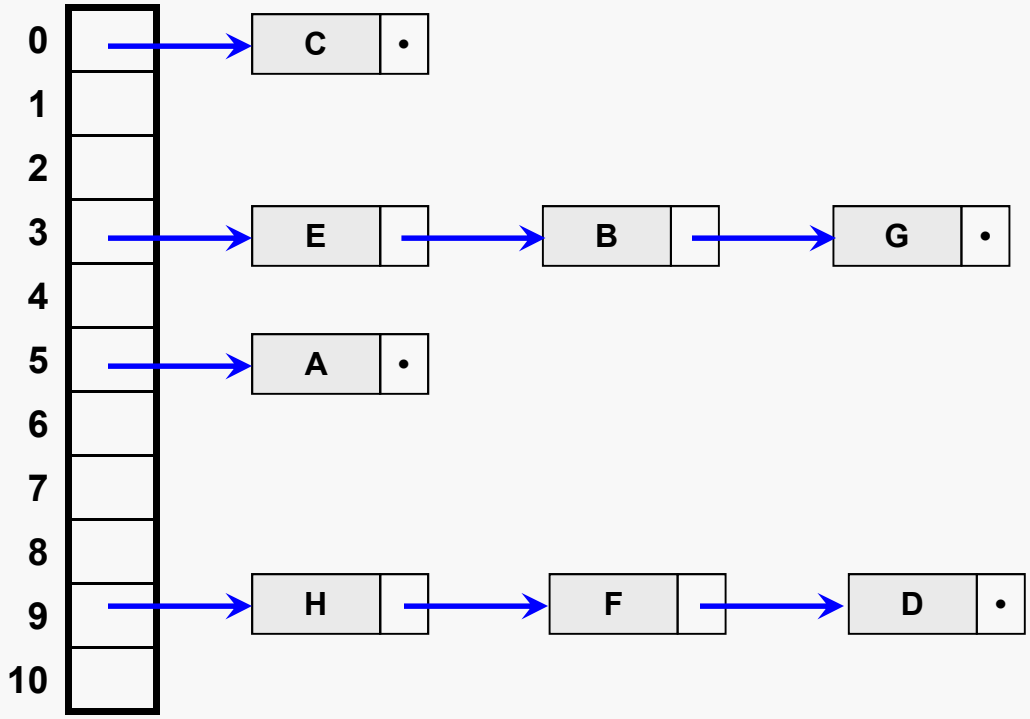
We could periodically re-hash the table or use some other reorganization scheme.

Question: how to tombstones affect the logic of hash table searching.

Question: can tombstones be "recycled" when new elements are inserted?

Chaining

Design the table so that each slot is actually a container that can hold multiple records.



Here, the “chains" are linked lists which could hold any number of colliding records. Alternatively each table slot could be large enough to store several records directly… in that case the slot may overflow, requiring a fallback…