Computer Science For Practicing Engineers

Abstract Data Types: Hash Tables I

Anthony J. Lattanze
Senior Lecturer, Executive Education Program
Institute for Software Research
Carnegie Mellon University

Lecture Topics

- We will look at the Hash Table ADT in detail:
  - Overview
  - Implementation Strategies
  - Memory and Performance considerations
  - Hash Functions and Strategies

The Hash Function

- A hash function *h(k)* maps keys of a given type to integers in a fixed interval [0, N−1], for example:
- Assume that we have an array A of length N, and k is a key (integer, character, or string), we will use to identify some related data then:
  - $h(k) = i$ will be an integer index into an array where k and all associated data will be stored
  - if N is the size of A then i < N-1

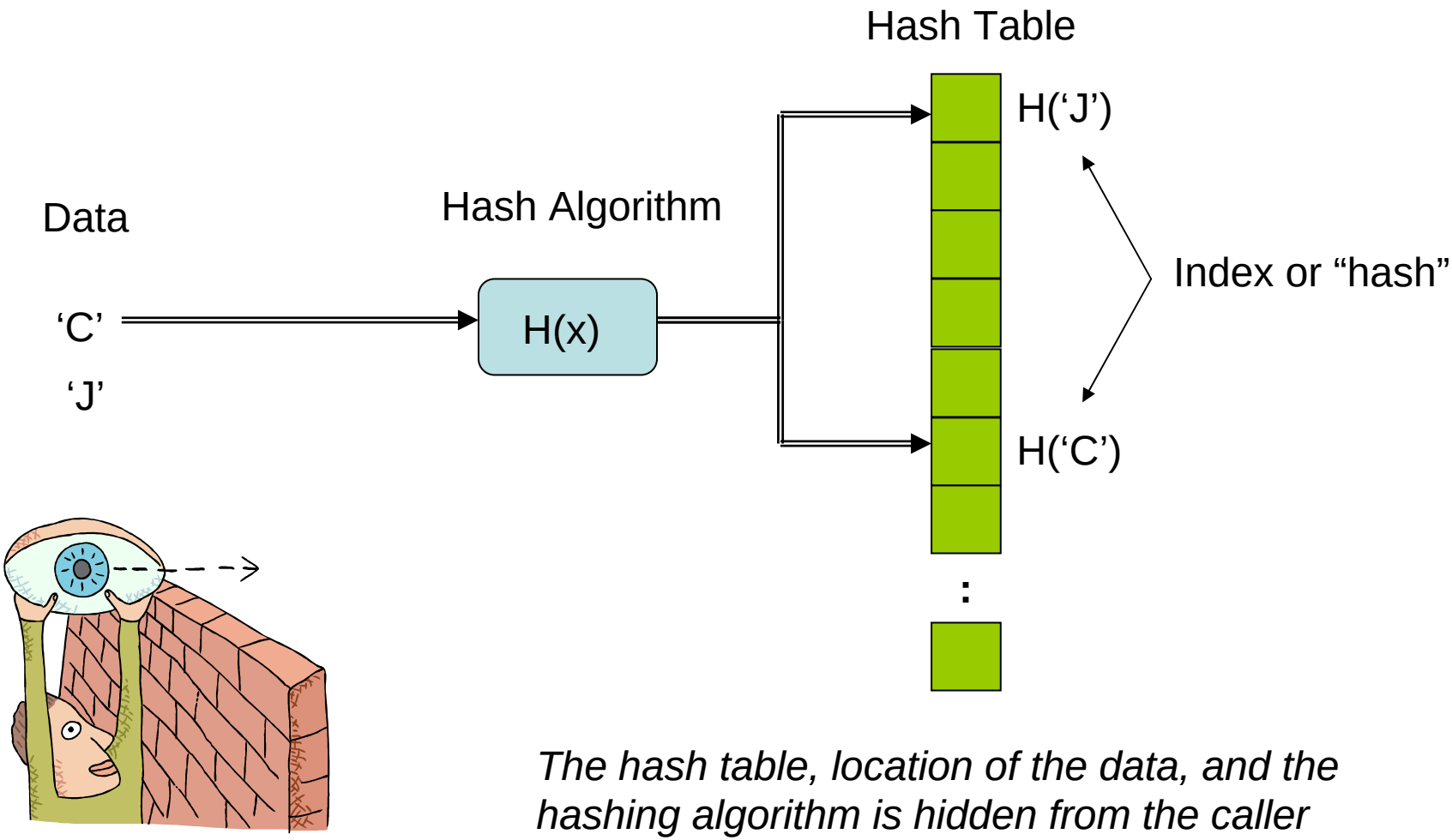
The Hash Table

- A hash table is usually implemented as a fixed size array
- There are many variations. The array can:
  - directly contain the data items
  - may be a list of pointers to one or more data items
- Hash tables can be implemented in many ways, but the important point to remember is that elements must be *directly addressable*, or *nearly directly addressable*

Hash Table Data Structure – 1

- The hash table is a non-linear data structure for storing data
  - They are often misunderstood, making them troublesome to implement
- Hash tables are composed of three parts:
  - data: the stuff you want to store
  - the hashing algorithm or function: the formula for deciding where datum will be stored in the hash table
  - the hash table: the place where data is stored; usually implemented with linked lists or arrays

Hash Table Data Structure – 2



The hash table, location of the data, and the hashing algorithm is hidden from the caller

Simple Example – 1

- A common way to collect data in real-time is in terms of *time*, *tag*, *data*:
  - Time – refers to the arrival time of the data measurement
  - Tag – refers to a unique measurement ID
  - Data – refers to the actual data measurement
- Such data usually arrives in streams and we often must maintain a list of the most recent or current values
- An array based CVT is an easy way to do this

Simple Example – 2

One way (inefficient)

Natural Index	Tag	Time	Data
1	017	04C82983	A4384F01
2	273	04C98390	9DF21A01
3	002	04C99877	876402E3
4	543	04CA5905	F1200032
5	119	04CA7439	74625103
6	011	04CB0023	6FDE35A1
:			

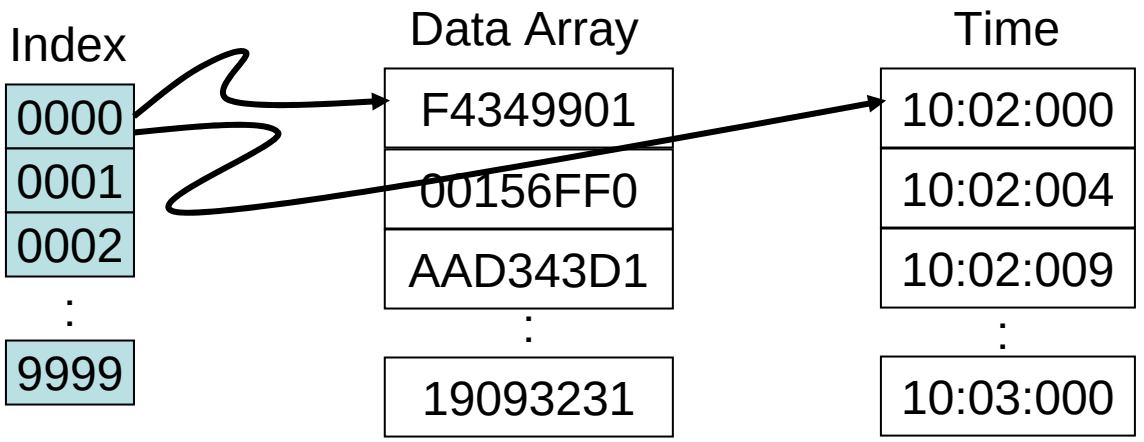
Searches by tag are O(n),…

Simple Example – 3

Assume we are collecting real-time data over time:
- *data has 4 digit ids*
- *data is the same type*
- *each datum arrives at a discrete time*

Measurement IDs	Measurement Data	Arrival Time
0000	F4349901	10:02:000
0001	00156FF0	10:02:004
0002	AAD343D1	10:02:009
:	:	:
9999	19093231	10:03:000

In this case, the array ID can be used as an index into parallel arrays that store the data and time respectively…



Direct Address Table – 1

- In this case, we have a collection of *N* elements stored in parallel arrays
  - each element possesses a unique key
  - the measurement ID, which are unique integers, is the key (0..m) where m >= *N*
  - this data structure is a *direct address* table (T[0..m])
  - any element of T[0..m], Ti is either empty (NULL) or contains a data item
- This is the simplest kind of hash table

Direct Address Table – 2

- Searching a direct address table is clearly an O(1) operation
  - For any key, k, we access T[k] directly. No search, No traversal, No hash function to speak of :
    - if it contains a data item, return it,
    - if it doesn't then return NULL
- There are two constraints for direct address tables:
  - the keys must be unique, and
  - the range of the key must be bounded

Direct Address Table – 3

- The range of the key determines the size of the hash table
  - For some applications the range may be too large to be practical
  - For example, it would be impossible to use a direct address table to store elements which have arbitrary 32-bit integers as their keys
    - In this case we would need a different hashing strategy or *hash function*
    - More on hash functions later…

Perfect Hashing

- Direct addressing requires that the hash function, h(k), is a one-to-one mapping from each k to integers in T[0..m]
- Such a function is known as a *perfect hashing function*
  - each key is mapped to a distinct integer within some manageable range
  - enables us to trivially build an *O*(1) search time table

Not So Perfect Hashing

- Unfortunately, finding a perfect hashing function is not always possible
  - Let's say that we can find a hash function, h(k),…
    - which maps *most* of the keys onto unique integers,
    - but maps a small number of keys on to the same integer
  - This hash function might map two different values of k onto the same key value in T[0..N]
  - This is a called a *collision*

Managing Collisions

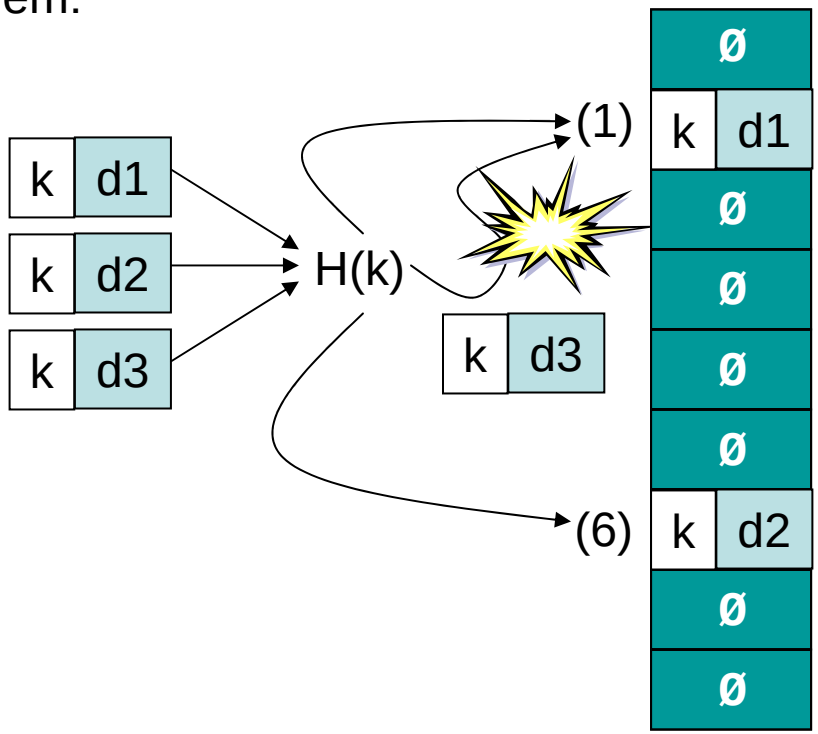
- If the number of *collisions*, is sufficiently small, then hash tables can work well enough
  - performance is good *we get O*(1) *search times*
- However, we need to handle collisions, there are several strategies:
  - chaining
  - overflow areas
  - re-hashing
  - linear probing
  - quadratic probing

Chaining – 1

- One simple scheme is to chain all collisions in lists attached to the appropriate slot
- This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection
- The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and time

Chaining – 2

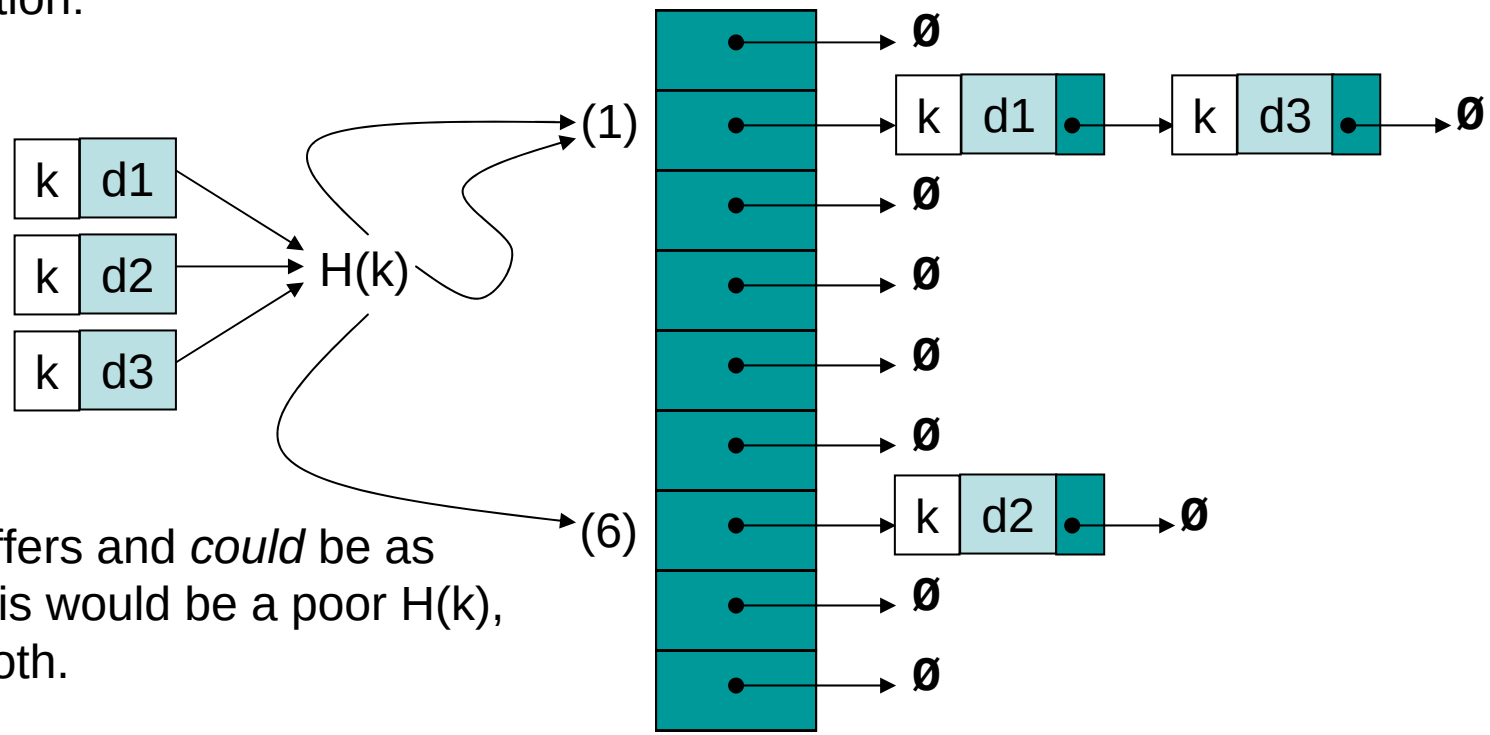
Problem:



Now what?!

Chaining – 3

Solution:



Performance suffers and *could* be as bad as *O*(n) – this would be a poor H(k), small table, or both.

Minimizing collisions in H(k) will dramatically improve index searches.

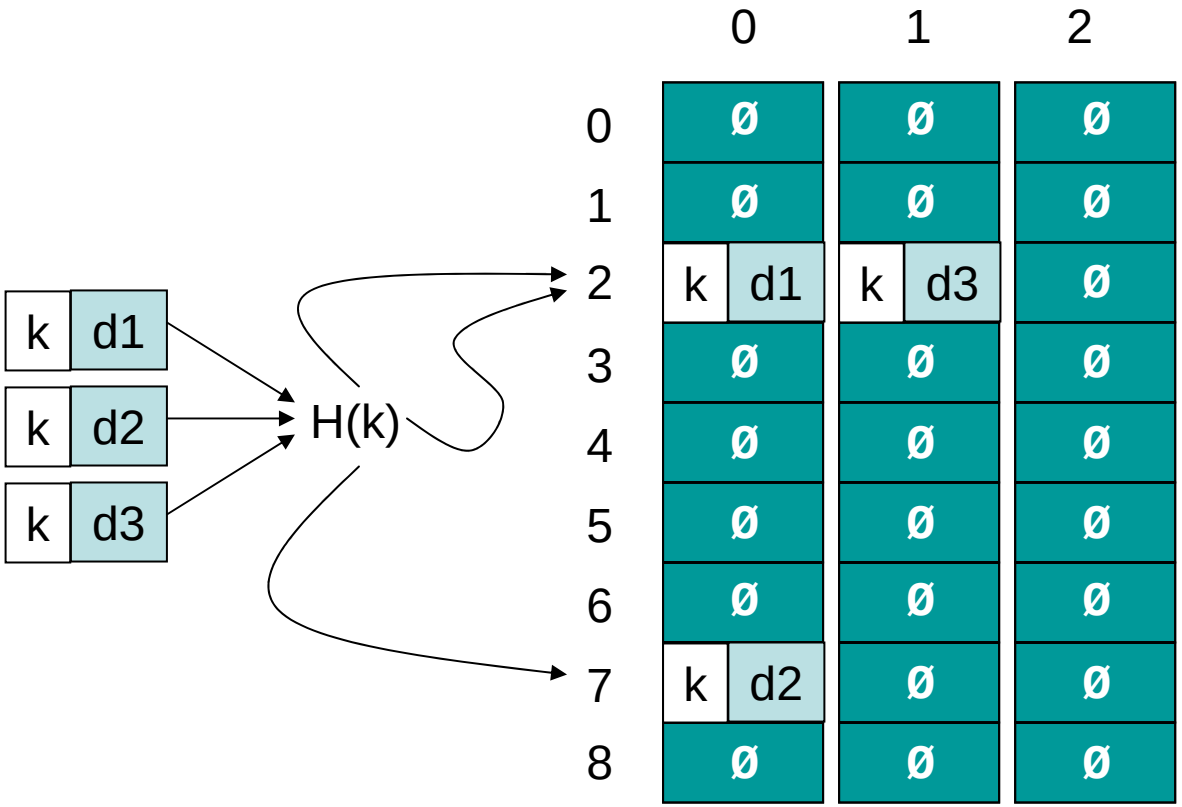
Overflow Areas – 1

- Another scheme will divide the hash table into two sections:
  - the *primary area* to which keys are mapped
  - a secondary area for collisions, normally termed the *overflow area*
- When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system

Overflow Areas – 2

- Overflowing is similar to chaining, except that the overflow area is pre-allocated and generally faster to access
- The maximum number of elements must be known in advance
  - In this approach we must be determine or otherwise estimate in the worst possible case size of the primary hash table and overflow area (or areas)

Overflow Areas – 3



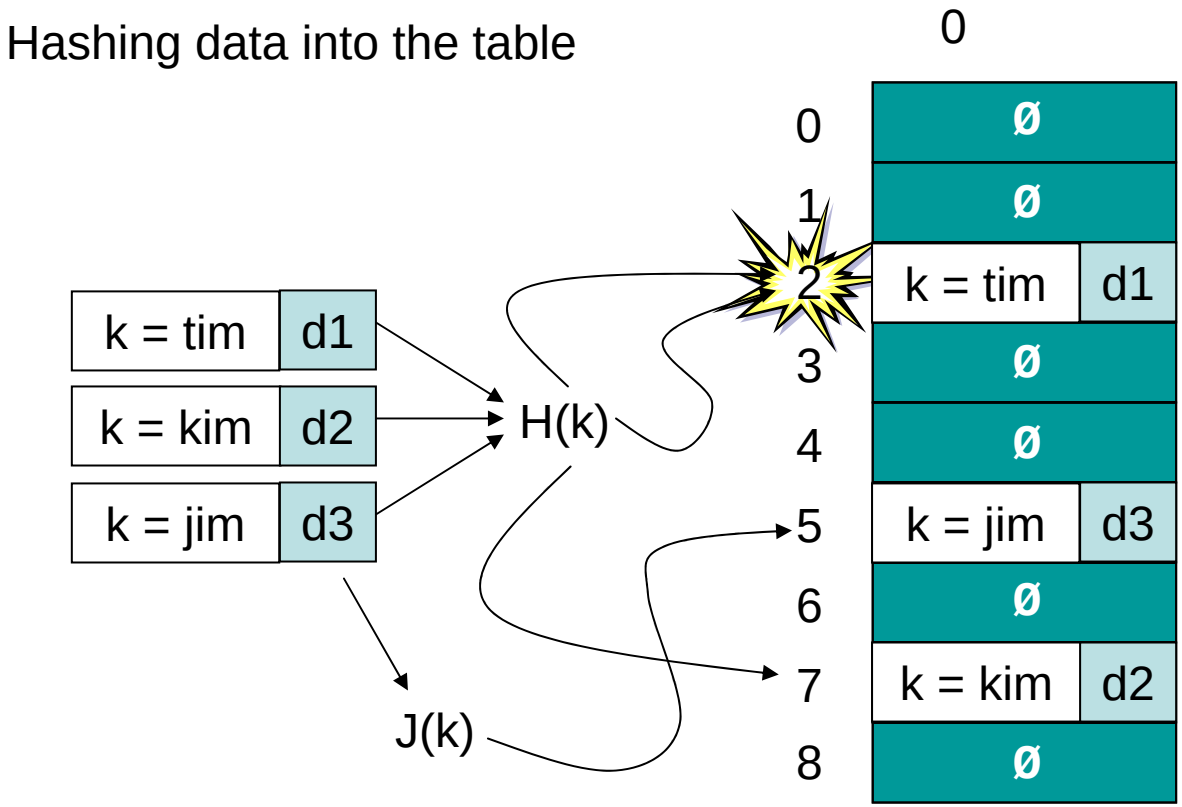
Overflow Areas – 4

- Of course, it is possible to design hash table overflow mechanisms with
  - single tables with overflow areas
  - multiple overflow tables
  - using pointers, but this starts to look like chaining and we loose some performance by not using direct array indexing
- These options provide flexibility without losing the advantages of the overflow strategy

Re-Hashing – 1

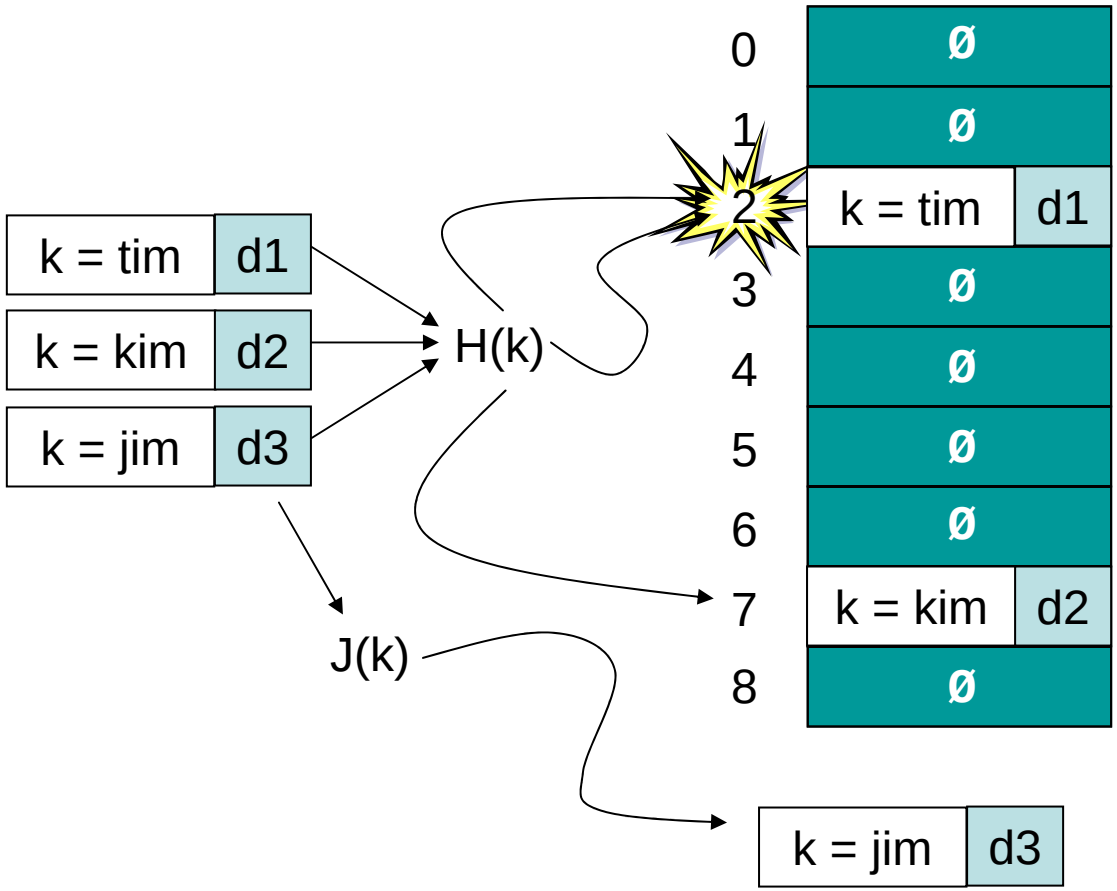
- Re-hashing schemes use a second hashing operation when there is a collision
  - If the second hash results in another collision, we *re-hash* until we find and empty "slot" in the table
  - The re-hashing function is typically a different function
  - As long as the hash functions are applied to a key in the same way, then a key can always be located

Re-Hashing – 2



Re-Hashing – 3

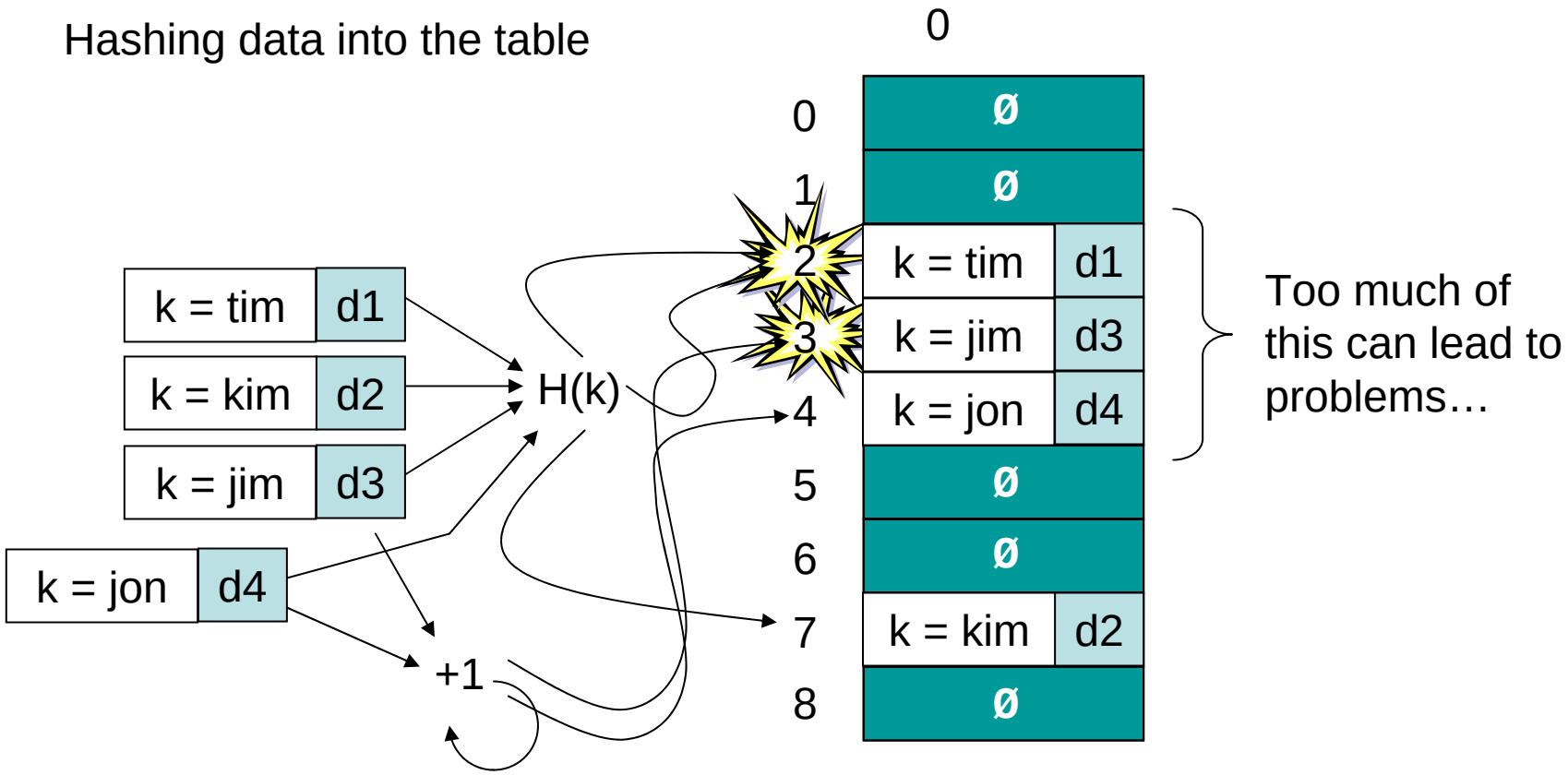
Hashing data out of the table



Linear Probing – 1

- Linear Probing is a technique of using a neighboring cell in the hash table (+1,-1) rather than the actual hash index
  - This is one of the simplest re-hashing functions
  - It calculates the new address very fast
  - Can be extremely efficient on a modern processor due to efficient cache utilization
- Linear Probing rehash policy must be consistently applied so keys can be found and recovered

Linear Probing – 2



Linear Probing – 3

- Linear probing can result in a “clustering” phenomenon where:
  - Re-hashes of keys point to the same location, causing a block of slots to be occupied in the table
  - The cluster can "grow" towards other slots in which other keys hash to
  - Clustering exacerbates collision problems and undermines any performance gains realized by utilizing linear probing

Quadratic Probing – 1

- An alternative to linear probing is quadratic probing
- Here we still explore a sequence of locations until an empty one is found:
  - linear probe: h, h+1, h+2, h+3, h+4, …
  - quadratic probe: h, h+1, h+4, h+9, h+16, ...
- Instead of incrementing the offset by 1 each time, we increment it by 1,3,5,7,..., i.e. the differences between successive squares

Quadratic Probing – 2

- Quadratic probes must be in increments equal to modulo table.length
- Naturally functions other than the quadratic could be chosen
  - these approaches have negligible overhead compared with linear probing
  - and guarantees successful insertion, provided table bounds are not exceeded

Secondary Clustering – 1

- Quadratic probing is relatively free from primary clustering, but it is still liable to *secondary clustering*
  - Secondary clustering refers to the fact that if two keys collide, the same probe sequence will be followed for both
  - If we think of this sequence as a "tail" then, as more items are inserted, insertions will take longer but because the "tails" tend to become longer

Secondary Clustering – 2

- However, secondary clustering is not nearly as severe as the clustering shown by linear probes (primary clustering)
  - Probing schemes use the originally allocated table space and avoid linked list overhead, but we must know number of items to be stored (worst case)
  - As collision elements are stored in slots to which other key values map to, the potential for multiple collisions increases as the table becomes full
    - The number of collisions can grow non-linearly…

References

- *Algorithmics: The Spirit of Computing (3rd edition)*, David Harel, Yishai Feldman
- *Data Structures and Algorithms*, Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft
- *Introduction to Algorithms (2nd edition)*, Thomas H. Cormen et al.