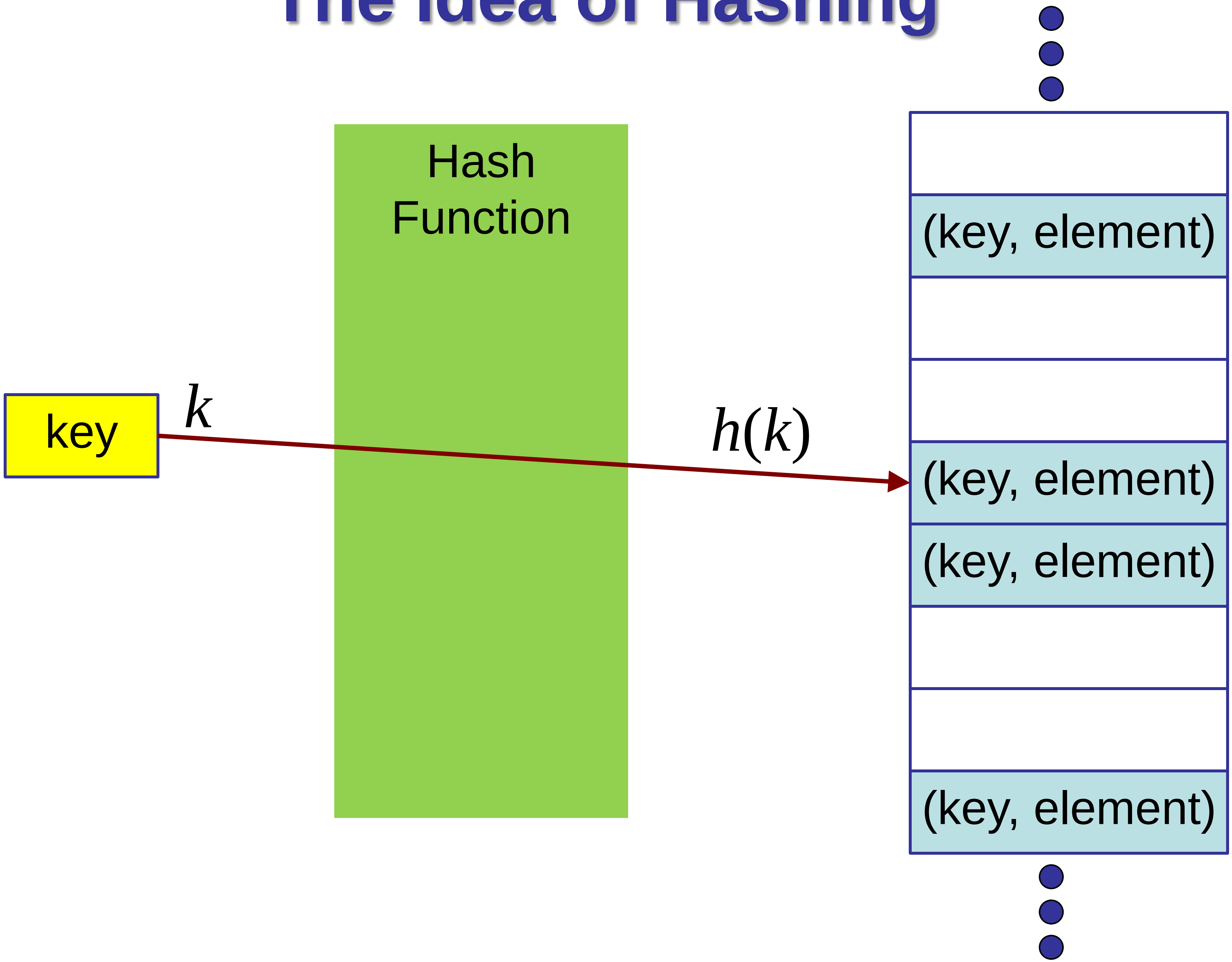
Hashing
(chapter 8)

- Hash Tables
- Hash Functions
- Handling Overflow

Motivation

- Assume that we have a collection of key-element pairs (a **dictionary**), and we need to do frequent **query** (**search**) operations. What is the data structure of choice?
- Will you like a data structure where the search time is O(1)?

The Idea of Hashing



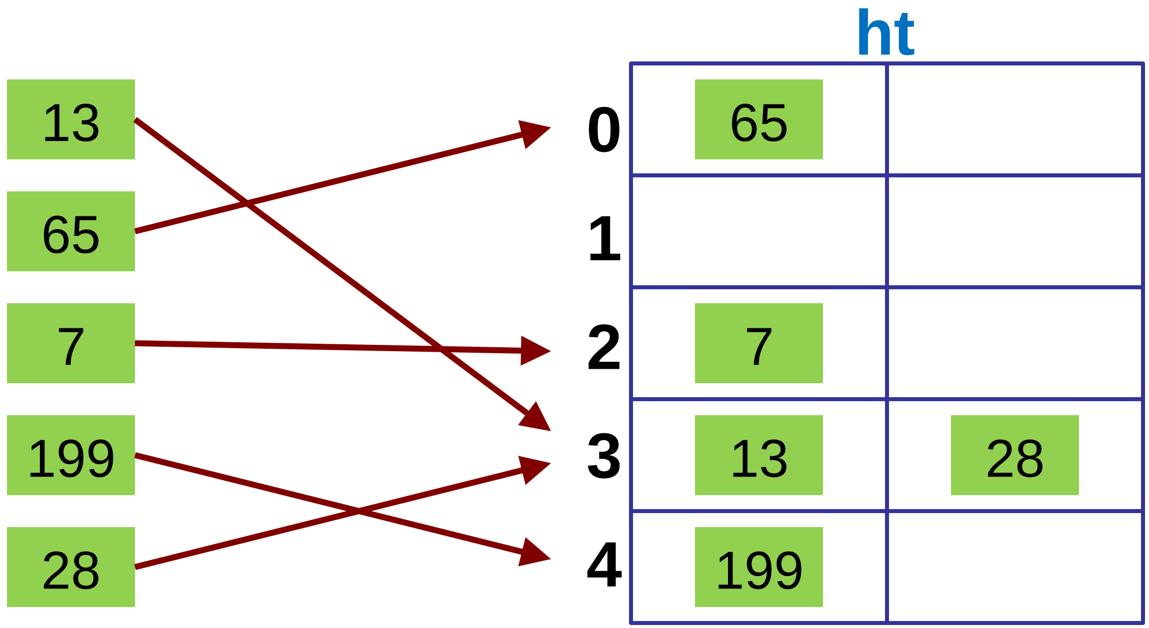
Concepts of Hash Tables

- A **hash table** stores a set of **<key,element>** pairs in sequential memory spaces.
  - Each space can store the whole record if it is small.
  - A space only stores a pointer to the record if the record is large.
- The hash table has **b** buckets, and each bucket has **s** slots. Each slot can hold a record.
- The bucket to put a record is determined by the key via a **hash function h**. Ideally, a record with key **k** is to be placed at bucket **h(k)**, which is the **hash address** of **k**.
- Time complexity becomes independent of **n**.

Hash Table: A Simple Example

- Consider a basic hash table with **s=2**.
- Let's experiment with a very simple hash function:

h(x) = x % 5



- Some waste of space is inevitable.
- What problem may arise?

Hash Table Terminology

- **Key density**: **n/T**, where **n** is the number of records and **T** is the number of possible keys.
- When two different keys **k_1** and **k_2** have **$h(k_1)=h(k_2)$**, they are called **synonyms** with respect to **h**.
- **Collision**: When a record is to be inserted into a non-empty bucket. There may or may not be slots available.
- **Overflow**: When we want to insert a record, but the bucket at its hash address is full.
- **Loading factor**: α**=n/(sb)**. This is the "density" of records in the table. The hash table is full when α**=1**.
  - We can expect collision and overflow to be more frequent for larger α.

Main Issues of Hash Tables

- The choice of the hash function:
  - Fast computation.
  - As few collisions as possible.
  - Unbiased. (For the expected set of keys, all the buckets have similar probabilities of assignment.)
- When handling overflow, we need to consider:
  - Efficiency during insertion (when overflow occurs).
  - Efficiency during subsequent searches.
- What to do if the hash table is full?

Types of Hash Functions

- Here we focus on hash functions used for hash tables. (There are other uses of hash functions. For example, see textbook section 8.2.3.)
- Common choices for integer keys:
  - Division
  - Mid-square
  - Folding

Hash Function: Division

- Simple idea: **$h(k) = k \% D$**
- The divisor **D** is also the number of buckets.
- Unbiased for uniformly distributed random keys.
- For some intuitive keys (such as power of two), may be very biased for some set of keys used in real applications.
- Best choice of the divisor **D**: An integer whose smallest prime factor is not too small (<20).
  - For example, if we want approximately 5000 buckets, using **$D=71^2$** is not bad.
- When **D** needs to be set or changed at run time, such as when we need to grow the hash table:
  - Require that **D** is odd, and grow it using **$D \leftarrow 2D+1$**.

Hash Function: Mid-Square

- Simple idea: **$h(k)$ = a set of middle bits of k^2**
- Example:

  $k = 10 \rightarrow k^2 = 100 = 0001100100_2 \rightarrow h(k) = 12$

  $k = 11 \rightarrow k^2 = 121 = 0001111001_2 \rightarrow h(k) = 15$

  $k = 12 \rightarrow k^2 = 144 = 0100100000_2 \rightarrow h(k) = 4$

  $k = 24 \rightarrow k^2 = 576 = 1001000000_2 \rightarrow h(k) = 8$

- Number of buckets is 2^r when r bits are used.

Hash Function: Folding

- Useful for sparse, long keys
- Example:
  - Shift-folding:

    $k=10235590276 \rightarrow h(k)=102+355+902+76=1435$

    $k=35122401210 \rightarrow h(k)=351+224+12+10=597$

  - Folding at the boundaries:

    $k=10235590276 \rightarrow h(k)=102+553+902+67=1624$

    $k=35122401210 \rightarrow h(k)=351+422+12+1=786$

Hash Function for Arbitrary Keys

- This includes variable-length keys, such as strings.
- We hash such keys by converting them to non-negative integers. Then hash functions for non-negative integers can be used.
- Example: Treat every pair of characters as a two-byte non-negative integer and take the sum (probably keep only the last 16 bits).
  - k="DATA" $\rightarrow$ h(k)=(68*256+65)+(84*256+65)=39042
  - k="STRUCTURE" $\rightarrow$ h(k)=(83*256+84)+(82*256+85)

    +(67*256+84)+(85*256+82)+69

    =81556 $\rightarrow$ 16020 (after %65536)

Handling Overflow

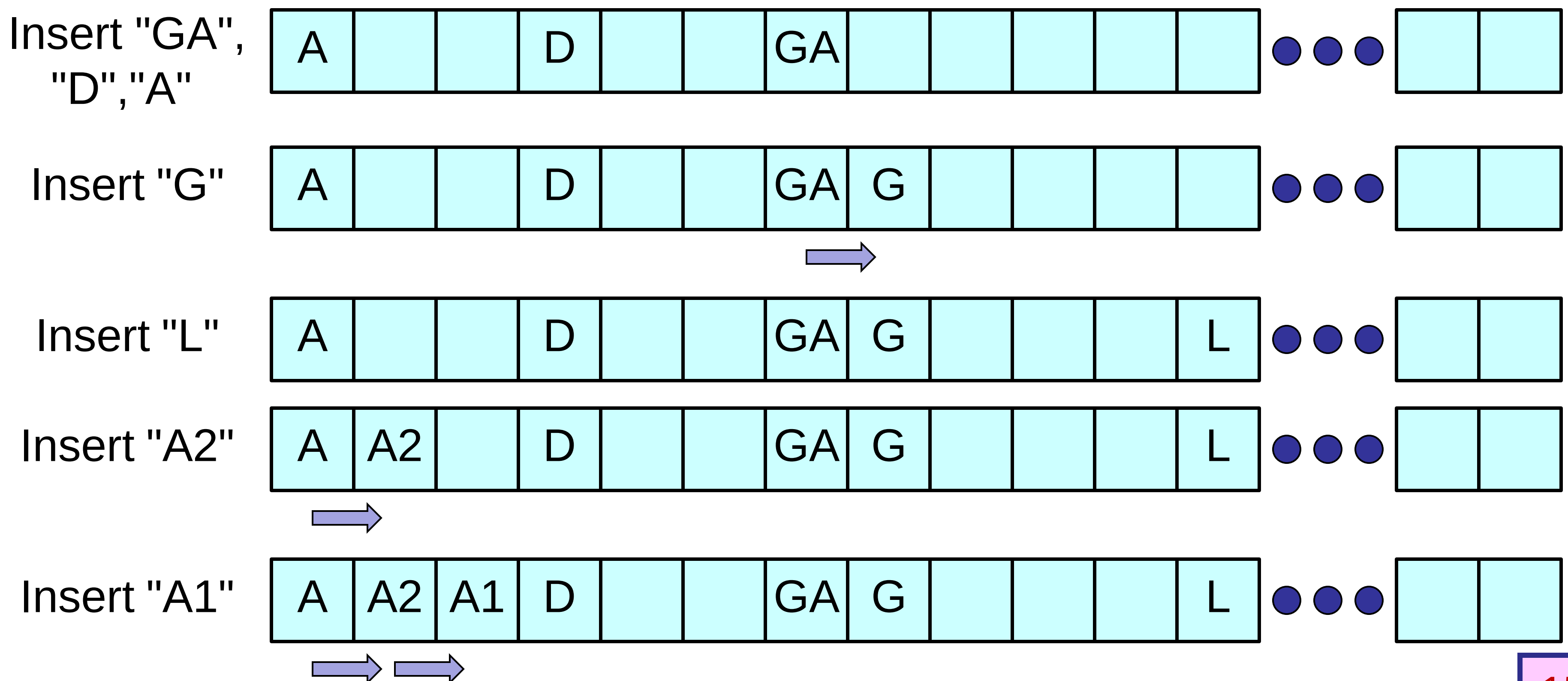
- Overflow happens when we want to insert a record into a bucket that is already full.
- Two main approaches to handle it:
  - **Open Addressing**: Use other buckets that still have spaces.
  - **Chaining**: Let each bucket have a linked list instead of a fixed number of slots. (As a result, overflow will not occur.)

Handling Overflow: Open Addressing

- Starting from the hash address (the bucket identified by **h(k)**), the algorithm has to examine a series of other buckets.
- The order of buckets to be searched has to follow a fixed rule because future queries have to follow the same rule to locate a record.
- Different open addressing schemes differ by the rule used to locate the bucket with the available slot. We will discuss:
  - **Linear Probing**
  - **Quadratic Probing**
  - **Rehashing**

Open-Addressing: Linear Probing

- The buckets are searched sequentially.
- The **i**th bucket checked is **(h(k)+i) % b**.
- Example (**b**=26, **s**=1, hashed by first letter (A-Z)):



Open-Addressing: Linear Probing

Insert "A3"	A	A2	A1	D	A3		GA	G				L	●●●		
Insert "A4"	A	A2	A1	D	A3	A4	GA	G				L	●●●		
Insert "Z"	A	A2	A1	D	A3	A4	GA	G				L	●●●		Z
Insert "ZA"	A	A2	A1	D	A3	A4	GA	G	ZA			L	●●●		Z
Insert "E"	A	A2	A1	D	A3	A4	GA	G	ZA	E		L	●●●		Z

Open-Addressing: Linear Probing

- A lot of time is used in finding the bucket with vacancy.
- Average number of comparisons per insertion/search is **$(2-\alpha)/(2-2\alpha)$**. Here α is the **loading factor**.
  - Getting worse when the hash table is quite full.
- The phenomenon of clustering is a problem.

Open-Addressing: Quadratic Probing

- The **i**th bucket checked is

  $(h(k)+i^2)$ % b for odd **i** and **$(h(k)–i^2)$ % b**. for even **i**.

- The main advantage over linear probing is reduced clustering.

- To ensure that a bucket with vacancy is found if the table is not completely full, set **b** to be a prime number in the form of **4j+3** (e.g., 3, 7, 11, 19, 23, 31, …).

- Example: h(k)=3 and b=7. The order of buckets checked is: 3, 4, 6, 5, 1, 0, 2

- Other quadratic functions can be used, too.

Open-Addressing: Rehashing

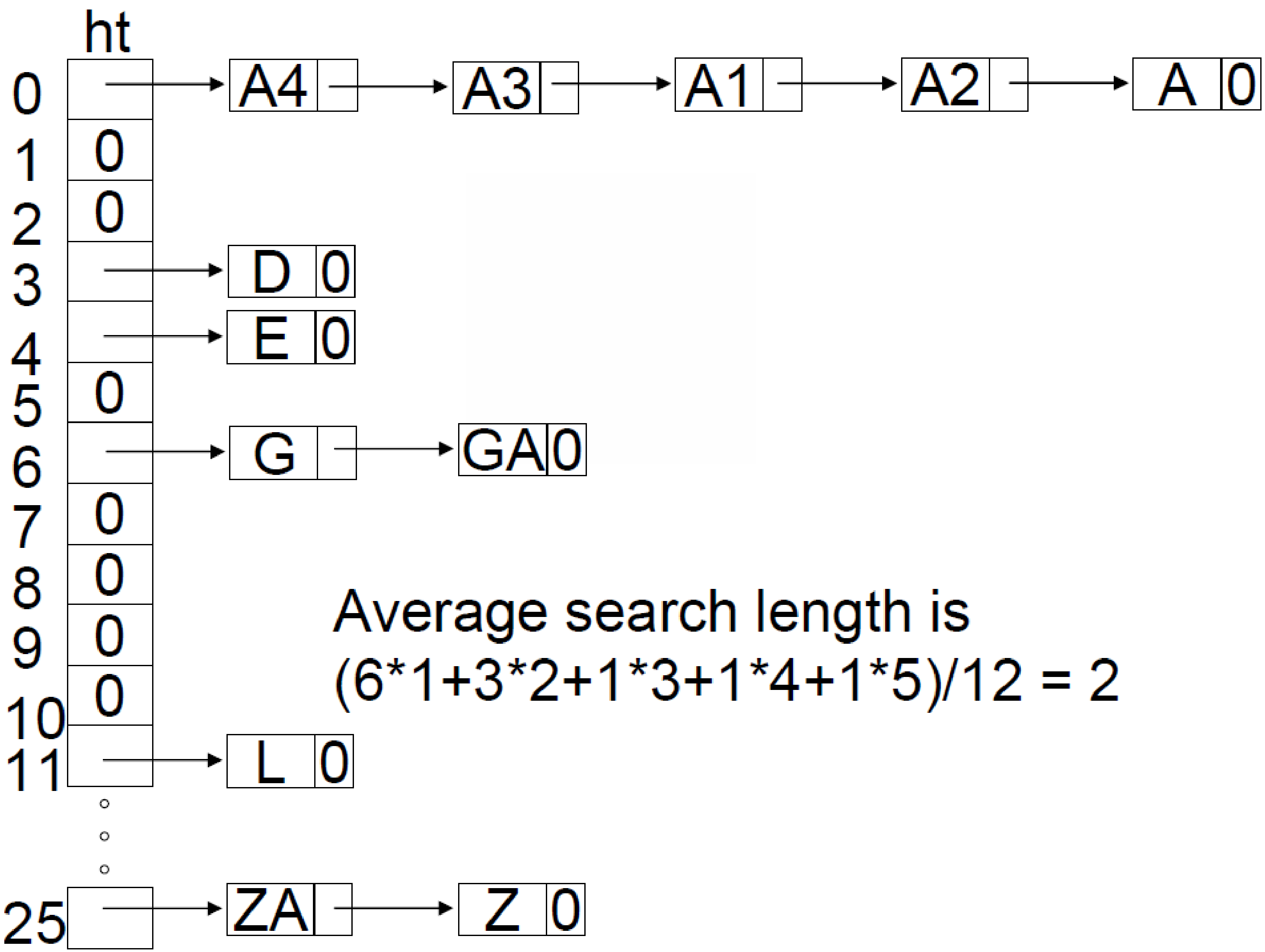
- The i^{th} bucket checked is $h_i(k)$, so we need a series of different hash functions.

Handling Overflow: Chaining

- This allows a bucket to hold (almost) unlimited number of records. ➔ The loading factor α can be larger than one.
- The data structures used to hold the records in a bucket:
  - Linked lists (the most common)
  - Self-balancing trees (only useful when a bucket might contain many records)
  - Dynamic arrays
  - etc.
- Using linked lists, the average number of comparisons per insertion/search is about $\mathbf{1+\alpha/2}$ for a uniform hash function.

Handling Overflow: Chaining

An example:



Average search length is
(6*1+3*2+1*3+1*4+1*5)/12 = 2

Open-Addressing vs. Chaining

- In general situations, chaining is faster especially when the loading factor is expected to be at 0.5 or higher.
- Open addressing is best when:
  - The loading factor is expected to be small.
  - We don't want to use dynamic memory allocation.
  - The total number of slots is fixed.

Hash Tables vs. Balanced Trees

- In average, hash table operations have O(1) time complexity, compared to O(log n) for balanced trees.
- However, the worst-case time complexity for hash table operations is O(n).
- Sometimes it is difficult to design a hash function that is both good (few collisions) and fast to compute. We have to factor in the time for computing hash functions.
- Hash tables are particularly useful when the sizes are known and fixed in advance, such as for real dictionaries.
  - Growing hash tables is much more difficult than growing balanced trees.
- We can not traverse the records in a hash table by keys.

Resizing Hash Tables

- What if a hash table is full and we need to insert more records? This is more difficult than handling overflows.
- Usually we need to allocate a new hash table with a new hash function, and move all the records over.
- This is very time consuming for large tables.
- In time-critical systems, there are methods for gradually moving over while handling operations at the same time. Such techniques are called **dynamic hashing**.

Other Applications of Hashing

- Finding duplicate or similar records.
- Data protection / authentication / digital signatures, etc. These applications require **collision-resistant** hash functions.
- More …