Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Hash Tables ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

10 LABS

Objectives

- Why Hashing?
- Hash Table
- Hash Functions
- Collision Resolution
- Deletion
- Perfect Hash Functions
- Hash Functions for Extendable files
- Hash code
- Maps
- Hashing in *java.util*

Why hashing?

- If data collection is sorted array, we can search for an item in O(log n) time using the binary search algorithm.
- However, with a sorted array, inserting and deleting items are done in O(n) time.
- If data collection is balanced binary search tree, then inserting, searching and deleting are done in O(log n) time.
- Is there a data structure where inserting, deleting and searching for items are more efficient?
- The answer is “Yes”, that is a Hash Table.

Why hashing?

- Very **VERY** fast way to build and access tables (and records in files too)
- Provide nearly O(1) performance
- Significantly faster than trees
  - ➔ which are O(log2n)!
- Simple to do too

Hash Tables

- We'll discuss the hash table ADT which supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called hashing.
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e., O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

Hash Tables

- Hash tables are based on arrays – which also raise disadvantages
- Difficult to expand after hash tables have been created
- Hash tables become too full
  - Performance degradation
  - → estimate the stored data
  - → move to larger table
- No convenient way to visit the items in a hash table in any kind of order

Hashing

Introduction to Hashing

Important concept

- Transform: Key values → Index values
  - Function do it: Hash function
- Example
  - Index of key = key / 10
  - Or If staff ID = 1 → store at position 1 of array

Index	**0**	**1**	**2**	**3**
A1	0	10	20	30
A2		Bill – 01	John – 02	Tom – 03

Introduction to Hashing

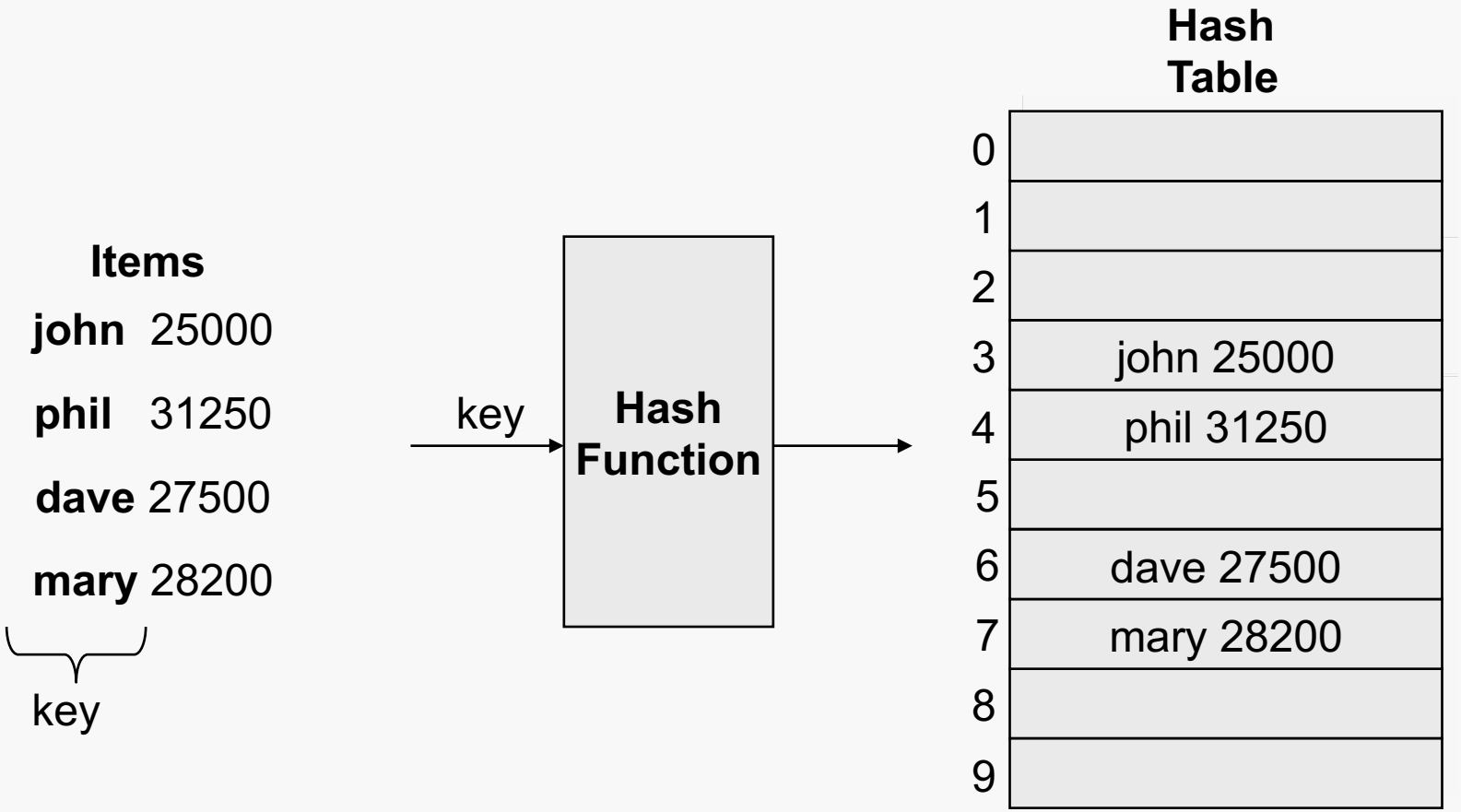
How to transform

- Key values → index values of Array ?
- Object → file location ?

- **Hash function maps**
- Key values (of these records, objects, etc.)

=> To table locations (or relative file locations)

Hash Table Example



Examples

- Take objects of type Person with ID attribute
- Take the last four digits (e.g., 1234)
  - divide by some prime number (often)
  - such that ALL objects will hash to values between, say, 0 and 47

Algorithm

Given an input object,

1. Access the key value of the object,
2. Hash to a location in an array,
3. Move the object into this location (indexed by the hashed-to value)

If object have a numeric key

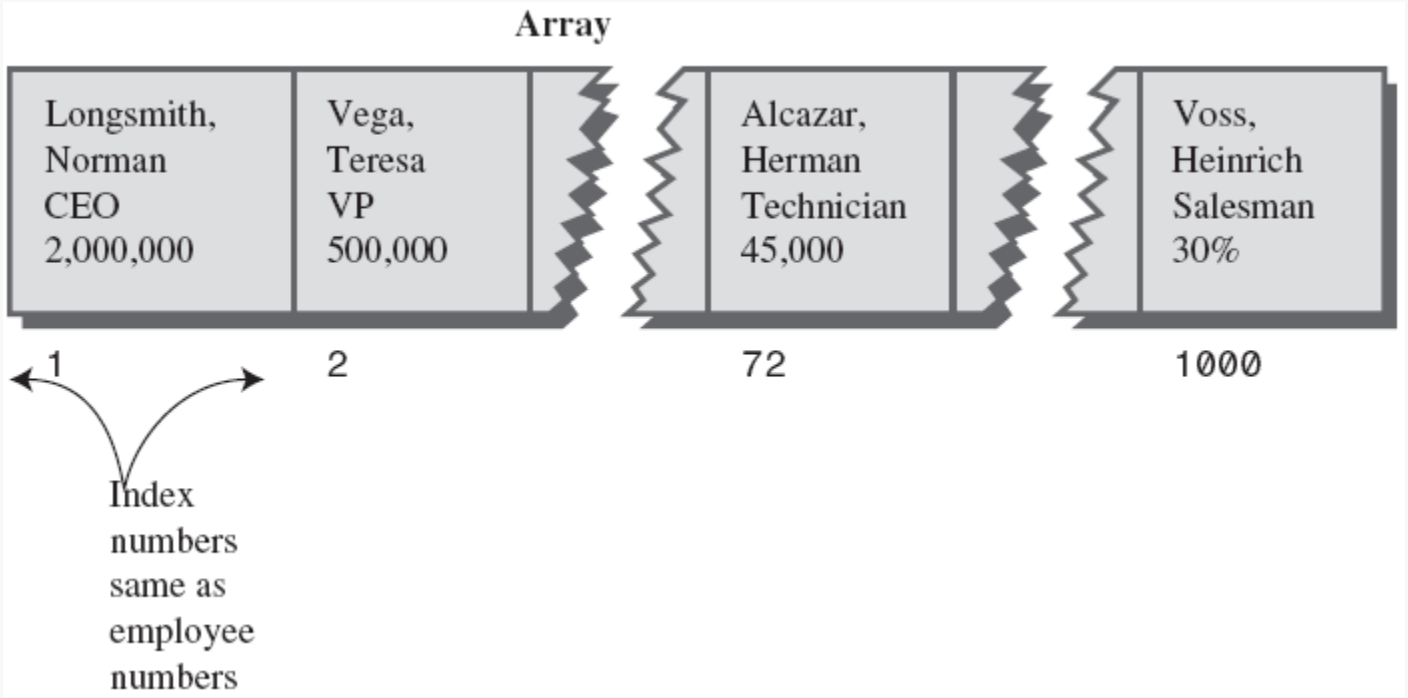
→Use key as index

→Don’t have to hash

Exception Case

- Existing Key: An attribute of an object can be used directly as an index in the hash table
- Examples:
  - Employee number
  - or an ‘experiment’ number
- In these cases,
  - Index number into the hash table = the key value
  - And we don’t have to actually ‘hash’ to a table value

Existing key



Staff ID No. As Keys

In this case

- Array-based database: very good
  - Data access: simple with high speed
- no deletions → memory-wasting, gaps don't develop
- New items added at the end of the array
- But it is not always the case.

A Dictionary

- Store 50,000-word English-language dictionary in main memory
- Every word to occupy its own cell in a 50,000-cell array
  - Access the word using an index number!

  ➔ **Fast**
  - what's the relationship of these index numbers to the words?
- A hash table is a good choice

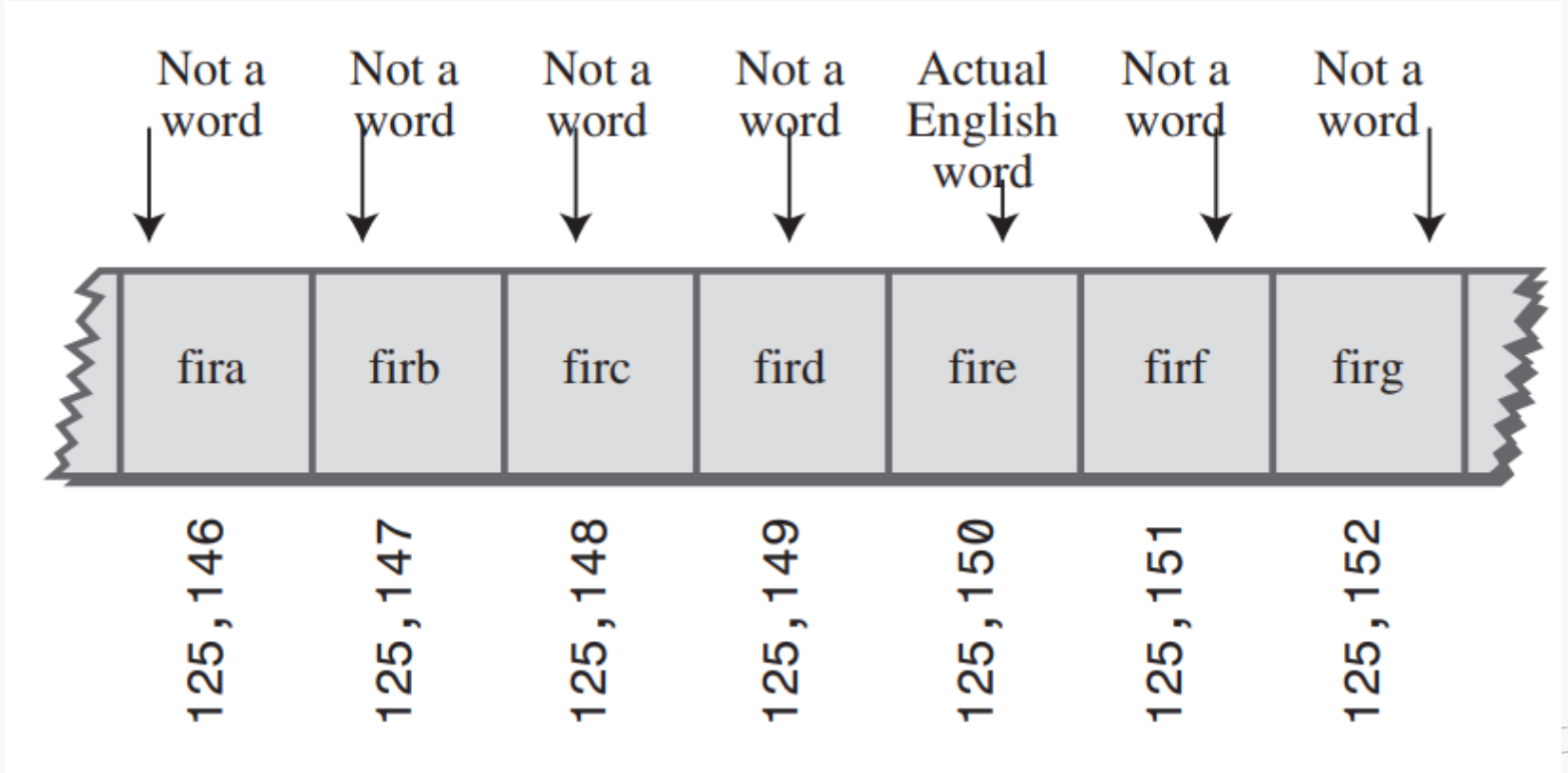
Converting Words to Numbers

- Convert letter → number → sum all letters
  - a = 1; b = 2, etc.
  - ‘cats’ → 3 + 1 + 20 + 19 = 43
- Unfortunately, many words will ‘hash’ to 43.
  - →‘synonyms’
- Our array, then, would be too small for all possible combinations.
- Ex: zzzzzzzzzz
  - → 260: Maximum index
  - → 50 000 / 260 = 192 synonyms

New converting formula

- To warranty each word have its own unique index
- Multiply by power
- Idea
  - For number
    $7654 = 7*10^3 + 6*10^2 + 5*10^1 + 4$
  - For word
    $cats = 3*27^3 + 1*27^2 + 20*27^1 + 19$
    $= 60337$

New problem: too many indices



Hash function

Hash Function

- The hash function:
  - Must be simple to compute.
  - Must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but in many cases we don’t.

Hash Function

- **Problems:**
- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

How to select Hash Function

- We want a hash function that is easy to compute and that minimizes the number of collisions.
- Hashing functions should be unbiased.
  - That is, if we randomly choose a key, x, from the key space, the probability that f(x) = i is 1/M, where M is the size of the hash table.
  - We call a hash function that satisfies unbiased property a *uniform hash function*.

Hash Function

- If the input keys are integers, then simply *Key mod TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Hashing

- To compress huge rage of number into reasonable range
  - Modulo (%): ArrayIndex = HugeNum % ArraySize
- → ***hash function***
- → ***hash table***: array to store data

Collisions & Open addressing

- Once squeeze a large number → small one
- Two words can hash to the same index

  → Collisions
- How to handle?
  - Set size of array = 2 * number of items
  - E.g.: 50 000 words → array of 100 000 elements
  - Collision → search for an empty slot → insert

  → ***Open addressing***

Collisions & Separate chaining

Another approach: ***Separate chaining***

- Array stores linked lists of words
- Then, when collision occurs

  → insert new item to linked list

Focus

- Devoted to collision algorithms
- For now, only consider the division remainder hashing algorithm (dividing by a prime number and optionally adding 1)

Collision algorithms

Collision Algorithms

- First Approach: Open Addressing
  - Search the array for an empty cell, then insert the new item there (e.g., increase the index by 1)
  - Three schemes
    - Linear probing
    - Quadratic probing, and
    - Double hashing
- Second Approach: Separate Chaining
  - An item of array consists of linked list of words
  - When a collision occurs, new item is inserted in the list at that index

Open addressing – linear probe

Ex:

- If 145th slot is occupied,
- Go to 146 and try again,
- If still occupied, increment the index and try again
- Until find empty cell

Operations on Hash table

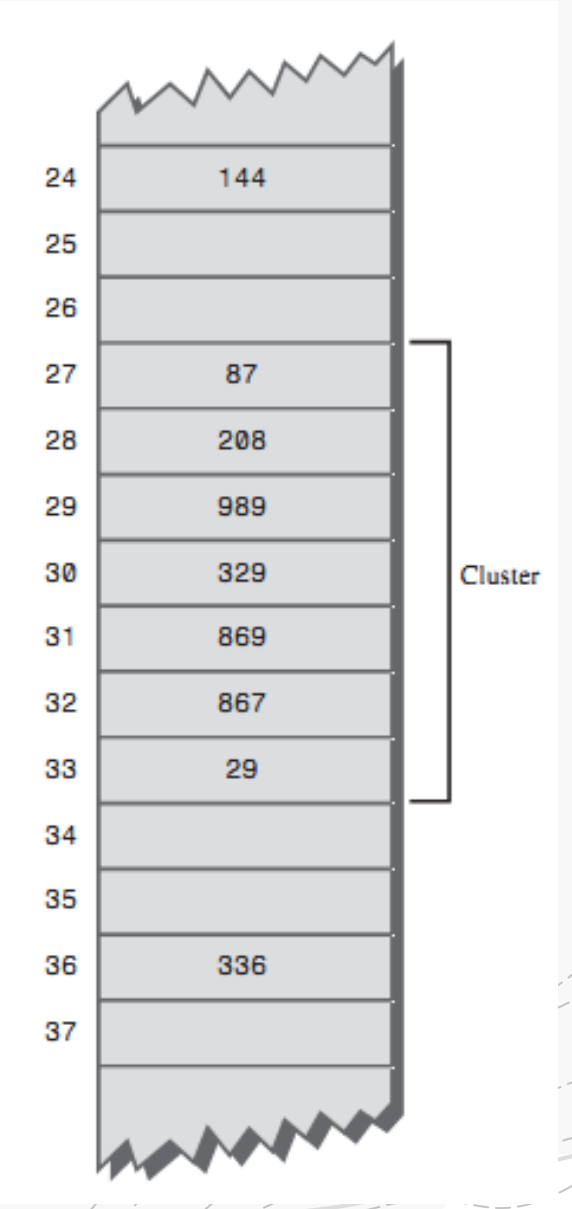
- Find: Hash the key → locate in array
  - Probe
- Insert: Hash the key → find the available cell
  - Probe length
- Delete: Find → replace
  - Don’t remove but replace with special value
- Duplication in hash table

Clustering

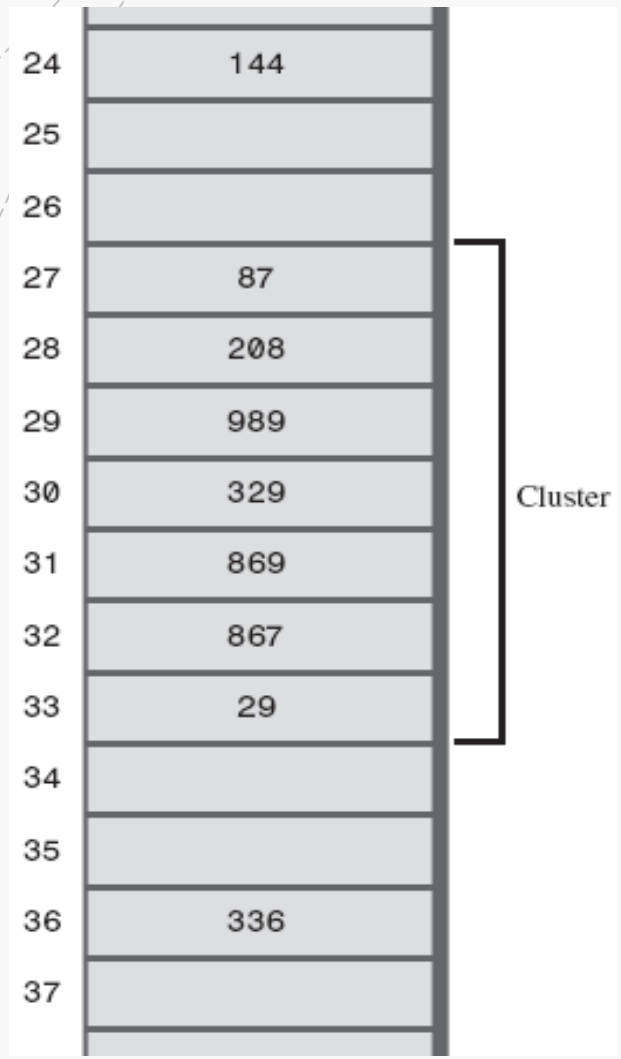
- Sequence of filled elements
- Hash table more full, the clusters grow larger
- Clustering

→ very long probe length

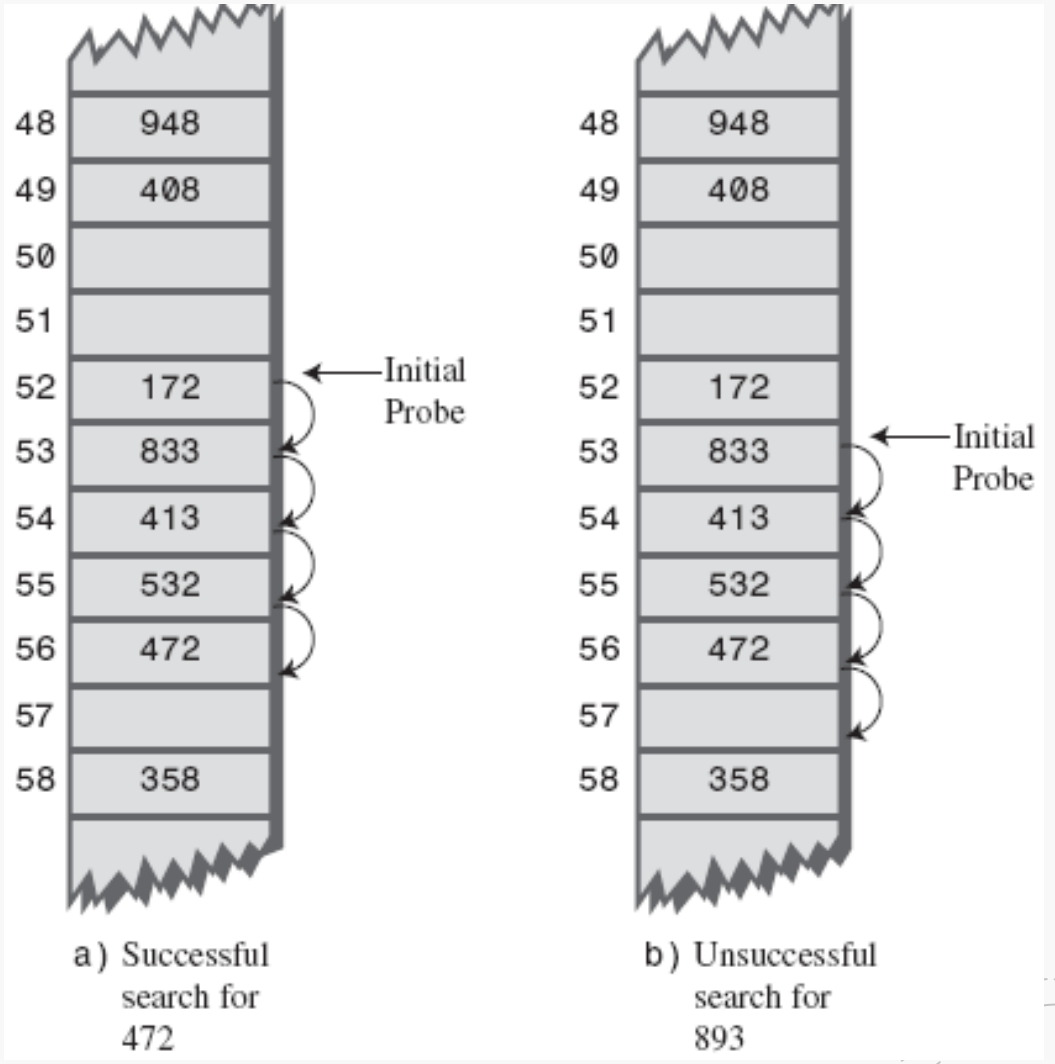
→ degrade the performance



clustering.



Linear probes.



Hash Table is Too Small?

- If hash table is full, what to do?
  - ***Load factor = nItems/arraySize***
- Can’t simply copy to new array
- Go through the old array, insert each item to new hash table by using INSERT()
  - This is called rehashing
- Since array sizes should be a prime number, the new array will be a bit larger than twice the original size

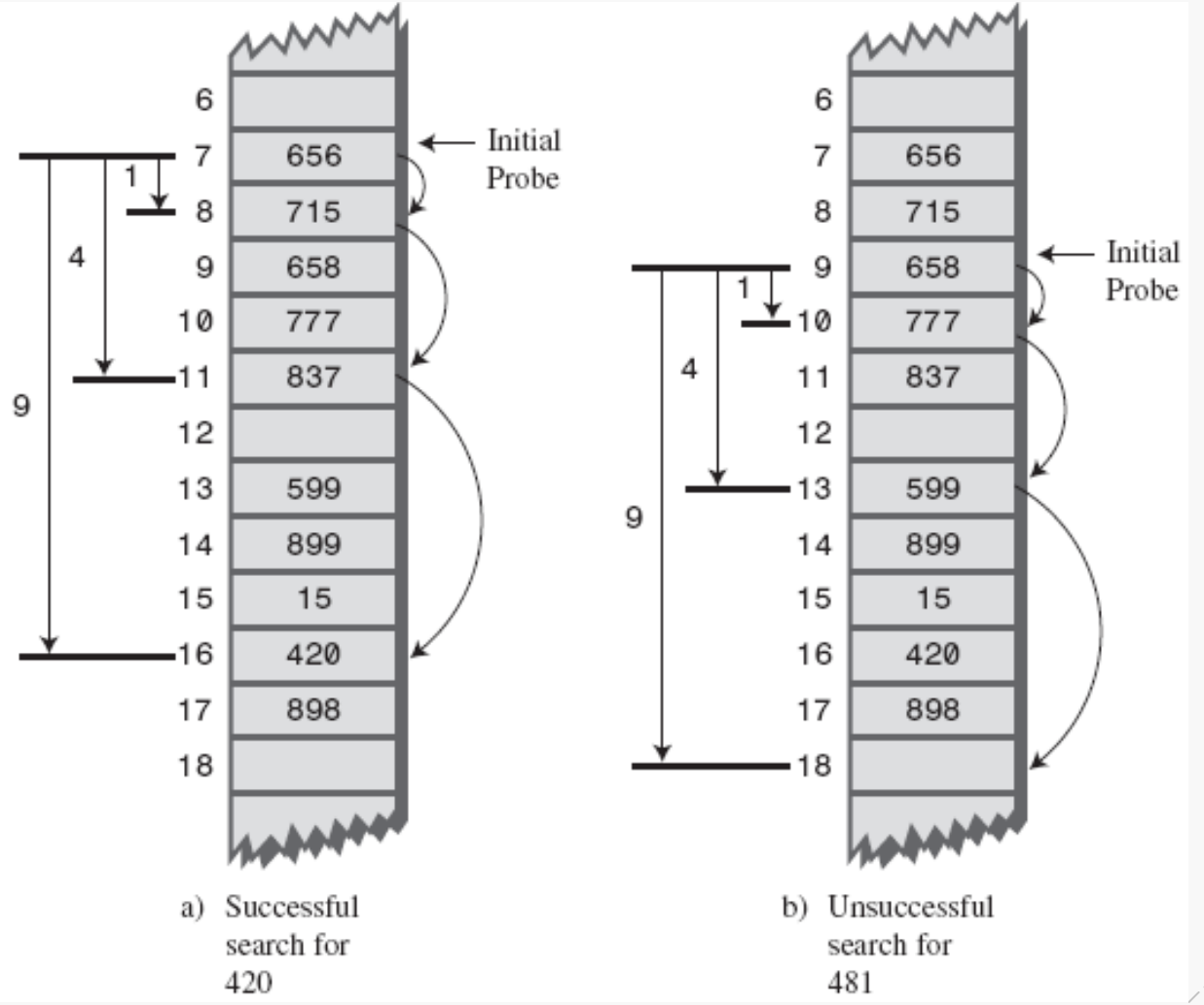
Open Addressing: Quadratic Probing

- An alternative to linear probe
- Used to address the problem of clustering
  - New values that hash to the same address or to a range near this one (and find their home addresses occupied) really increases clustering!!
- Quadratic probing stretches out the synonyms and thus reduces clustering…

Quadratic Probing – 2

- Idea is simple:
- In linear probing, addresses when from x to x+1 to x+2, etc.
- In quadratic probing, addresses go from x to x+1 to $x+2^2$ to $x+3^2$ to $x+4^2$…
  - Distance from initial probe is the square of the step number
- This approach does spread out the collisions, but can easily become wild…

Quadratic Probes



Problems with Quadratic Probe

- There is a secondary problem
  - All keys that hash to a specific address DO follow the same step in looking for an open slot in hash table

  ➔ Each additional item will require a longer probe
  - Called secondary clustering
- There are better solutions than quadratic probing

Open Addressing: Double Hashing

- We need a different collision algorithm that depends on the key value
- Our solutions is to hash a second time using a different hashing function that uses the result of the first hash
- Secondary hash functions have two rules:
  - It must NOT be the same as the primary hash function, and
  - It must NEVER output a 0 (otherwise there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop)

Double Hashing

- Here is a sequence that works well:
  - stepSize = constant – (key % constant);
  - [1..constant]
- Where ‘constant’ is
  - a prime number and
  - smaller than the array size
- We will see this algorithm ahead for the insert and the two hashing functions
- Essentially, this algorithm adjusts the step size by rehashing…

Operative Code

```
public int hashFunc1(int key)
    {
    return key % arraySize;
    }
// ----------------------------------------------------------
  public int hashFunc2(int key)
    {
    // non-zero, less than array size, different from hF1
    // array size must be relatively prime to 5, 4, 3, and 2
    return 5 - key % 5;
    }
// ----------------------------------------------------------
  public void insert(int key, DataItem item)   // insert a DataItem
   // (assumes table not full)
    {
    int hashVal = hashFunc1(key);  // hash the key
    int stepSize = hashFunc2(key); // get step size
                           // until empty cell or -1
    while(hashArray[hashVal] != null &&
                hashArray[hashVal].getKey() != -1)
      {
      hashVal += stepSize;        // add the step
      hashVal %= arraySize;       // for wraparound
      }
    hashArray[hashVal] = item;     // insert item
    }  // end insert()
```

hashing algorithms. Observe closely

Can see what is done if collision

Table Size as a Prime Number

- Double hashing requires table size to be a prime number
- If not a prime number, for example
  - Size = 15, step = 5
  - The probe sequence = 0, 5, 10, 0, 5, 10, …
- Prime numbers have many very interesting arithmetic properties
  - all entries in the hash table will be visited, if necessary and not a cycle of repeated visits – a result of having a hash table of non-prime size

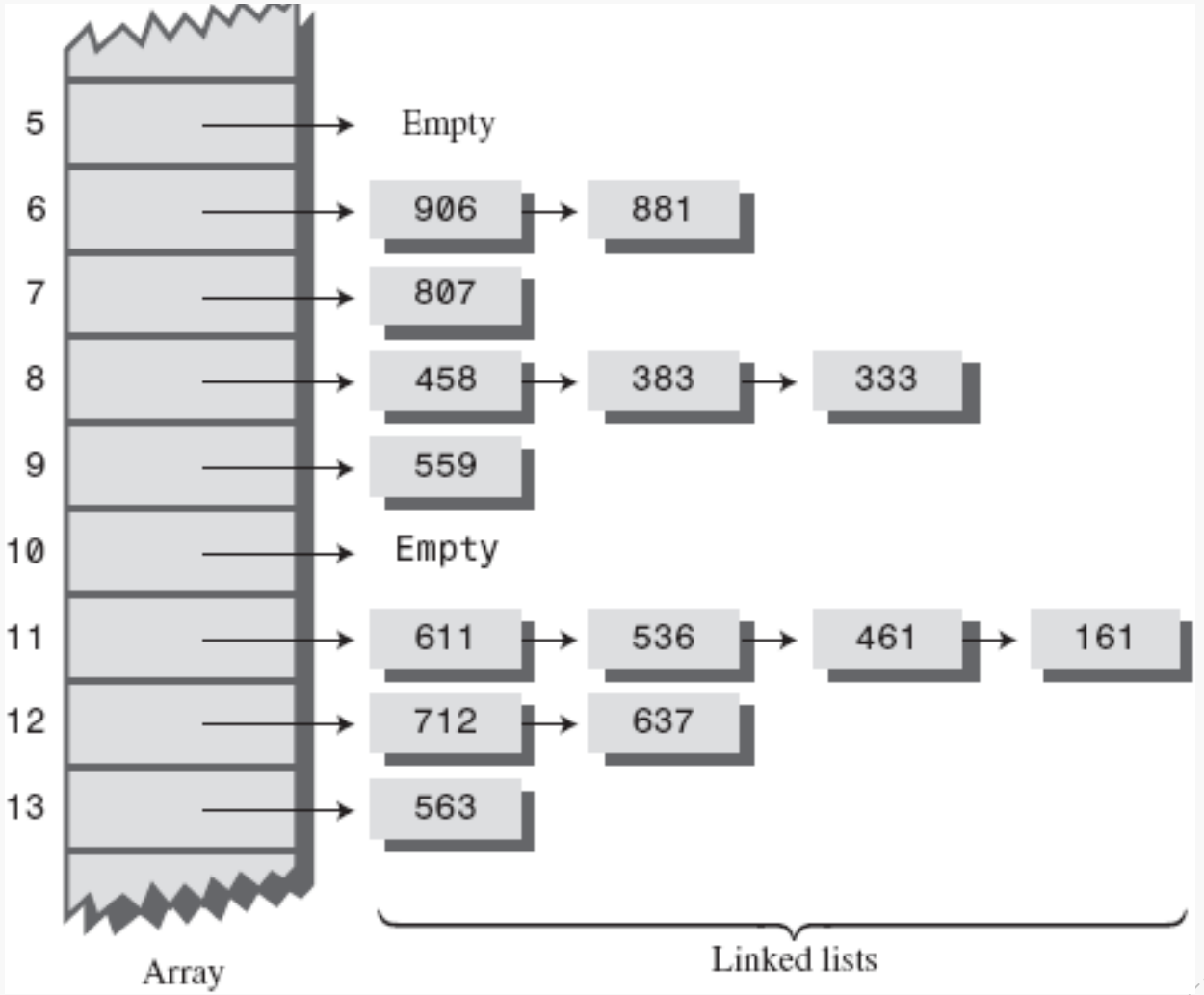
Finale

- If open addressing is used (and there are MAJOR disadvantages to Open Addressing), then double hashing provides the best performance
- Bear in mind, that these simple approaches may be quite fine for several applications!

Separate Chaining

- Problem with Open Addressing Schemes
  - Synonyms ALL occupy addresses (home addresses) of other potential keys!
- Conceptually the ‘fix’ is simple, but does require more code
  - All entries in the hash table are the first node of a linked list for that index
  - All keys that hash to an address in the hash table after the first entry are then moved into a singly-linked list with root in that home location
  - The linked list is used for synonyms (for collisions)

Separate Chaining-Example



Separate Chaining

- Initial cell takes O(1) time, which is super
- But search through linked lists takes time proportional to M, the average number of items on the list: O(M) time
- Result: we must keep linked lists short!
- Note: the linked lists have items allocated dynamically, so there is no wasted space
- The linked lists are not part of the hash table

Separate Chaining-Insert Code

```
public int hashFunc(int key)      //
   {
   return key % arraySize;
   }
-------------------------------------
public void insert(Link theLink)  //
   {
   int key = theLink.getKey();
   int hashVal = hashFunc(key);   //
   hashArray[hashVal].insert(theLink)
   }  // end insert()
```

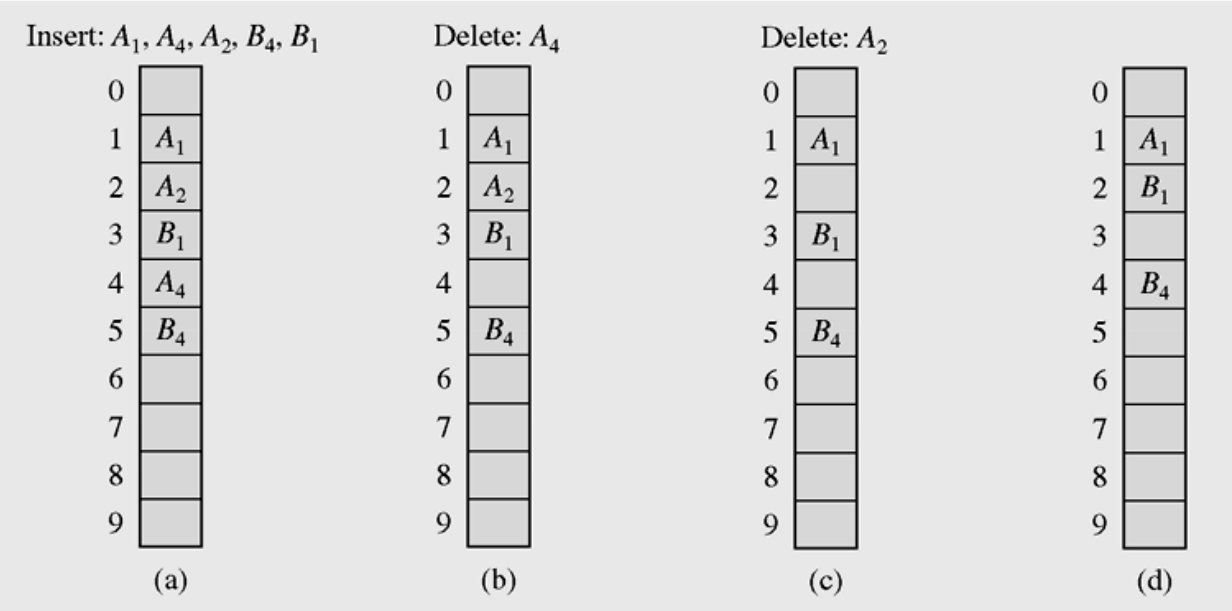
```
public void insert(Link theLink)  // insert link, in order
   {
   int key = theLink.getKey();
   Link previous = null;          // start at first
   Link current = first;
                                  // until end of list,
   while( current != null && key > current.getKey() )
      {                           // or current > key,
      previous = current;
      current = current.next;     // go to next item
      }
   if(previous==null)             // if beginning of list,
      first = theLink;            //    first --> new link
   else                           // not at beginning,
      previous.next = theLink;    //    prev --> new link
   theLink.next = current;        // new link --> current
   }  // end insert()
```

Duplicates / Deletions and Table Size in Separate Chaining

- Are allowed. No problem. You would simply traverse all the links in the linked list at that hash table's index (if allow dupes)
- Deletions: delink the node as appropriate
- Table Size: prime number not important as with quadratic probes and double hashing because we are handling collisions quite differently
- Still – a good idea to have hash table size = prime!

Deletion

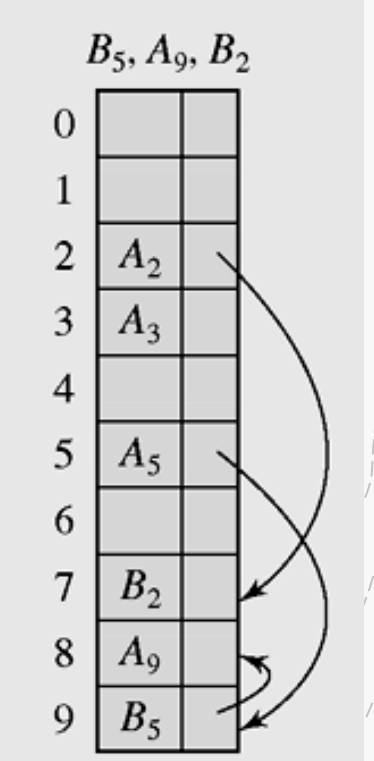
- Consider the table in which the keys are stored using linear probing.
- Suppose we delete A4 and then then try to find B4. Because when searching B we hash it to position 4 and see that this position is empty and conclude that B4 is not found (which is not true).
- To avoid this situation, we mark the deleted positions only.
- When inserting new element to this position, we update information for new element. When there too many marked deleted elements in the table, the table is refresh (d).



Linear search in the situation where both insertion and deletion of keys are permitted

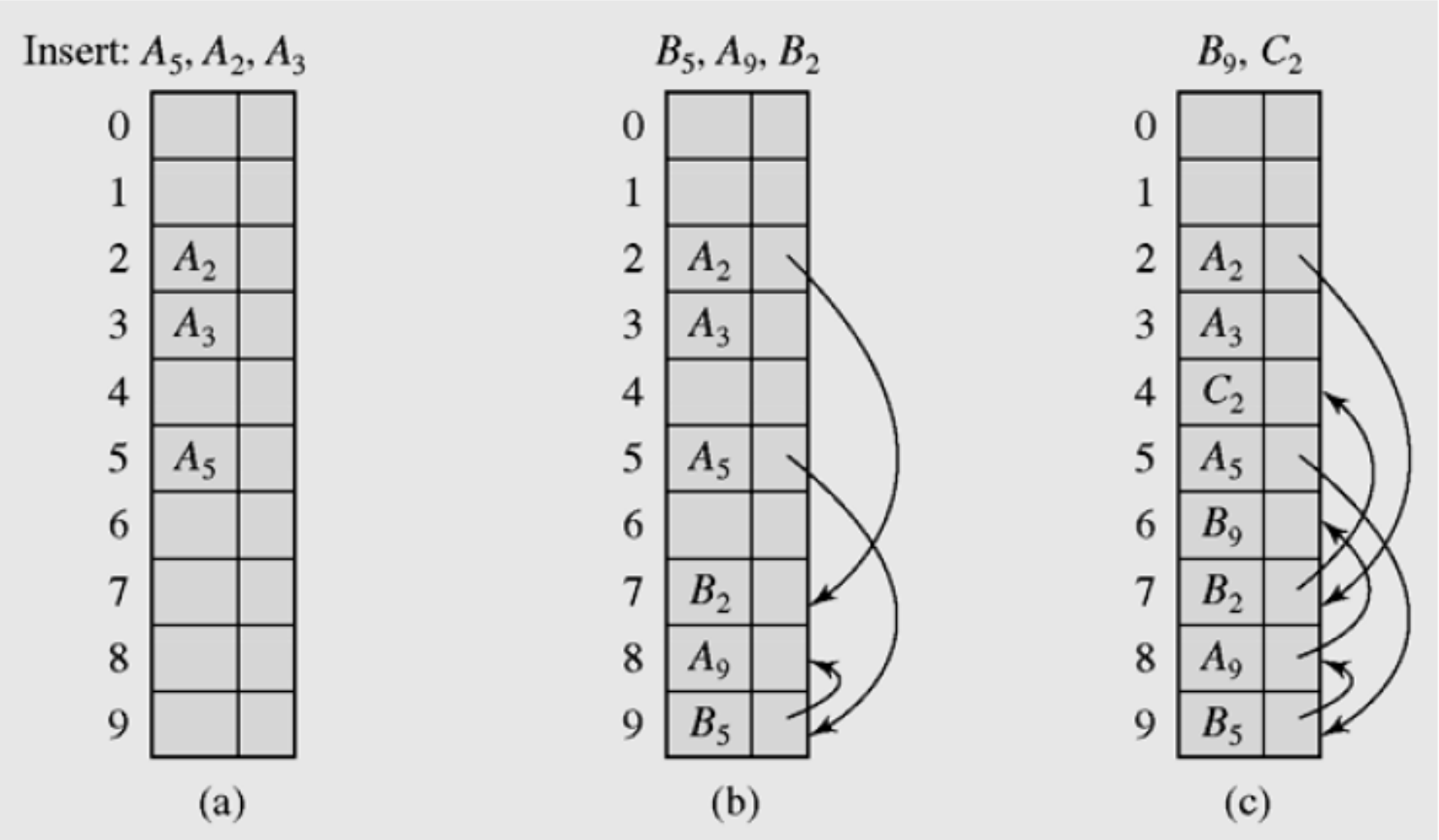
Coalesced hashing or coalesced chaining method

- A version of chaining called **coalesced hashing** (or **coalesced chaining**) combines linear probing with chaining. Each position ***pos*** in the table contains 2 fields: ***info*** and ***next***. The ***next*** field contains the index of the next key that is hashed to ***pos***. By this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list.
- An overflow area known as a ***cellar*** can be allocated to store keys for which there is no room in the table



Coalesced hashing puts a colliding key in the last available position of the table

Coalesced hashing example



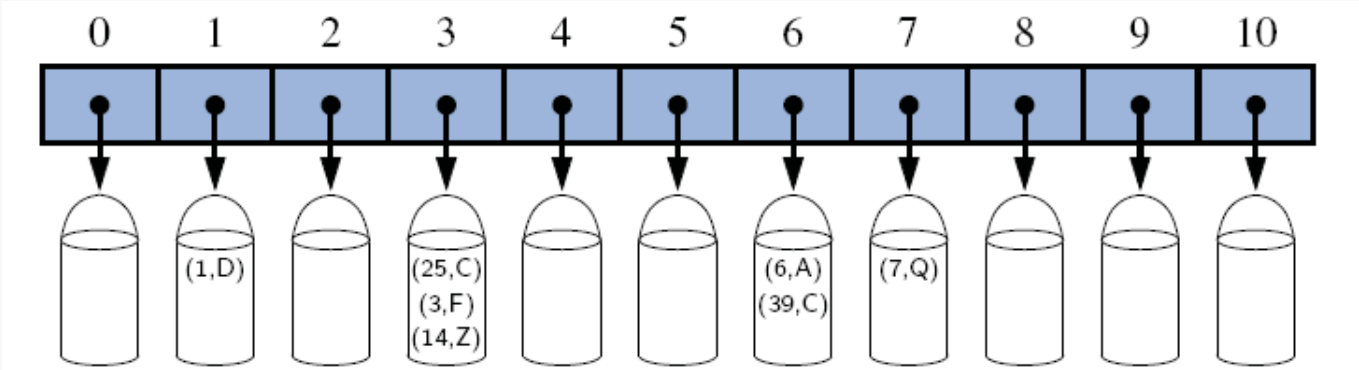
Coalesced hashing puts a colliding key in the last available position of the table

Bucket Approach

- Another approach in lieu of linked list is to **use an array** at each hash table location
- To store colliding elements in the same position in the table can be achieved by associating a bucket with each address.
- A bucket is a block of space large enough to store multiple items (a block consists of slots; each slot contains one item).
- By using buckets, the problem of collisions is not totally avoided.
- By incorporating the open addressing approach, the colliding item can be stored in the next bucket is it has an available slot when using linear probing, or it can be stored in some other bucket when, say, quadratic probing is used.
- The colliding items can also be stored in an overflow area. In this case, each bucket includes a field that indicates whether the search should be continued in this area or not.

Bucket Approach

- But must know array size first
- Can cause wasted space for unused slots or can be of insufficient size
- Thus, this approach is not as efficient as the linked list approach



Buckets of a hash table with size 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a modulo-division hash function.

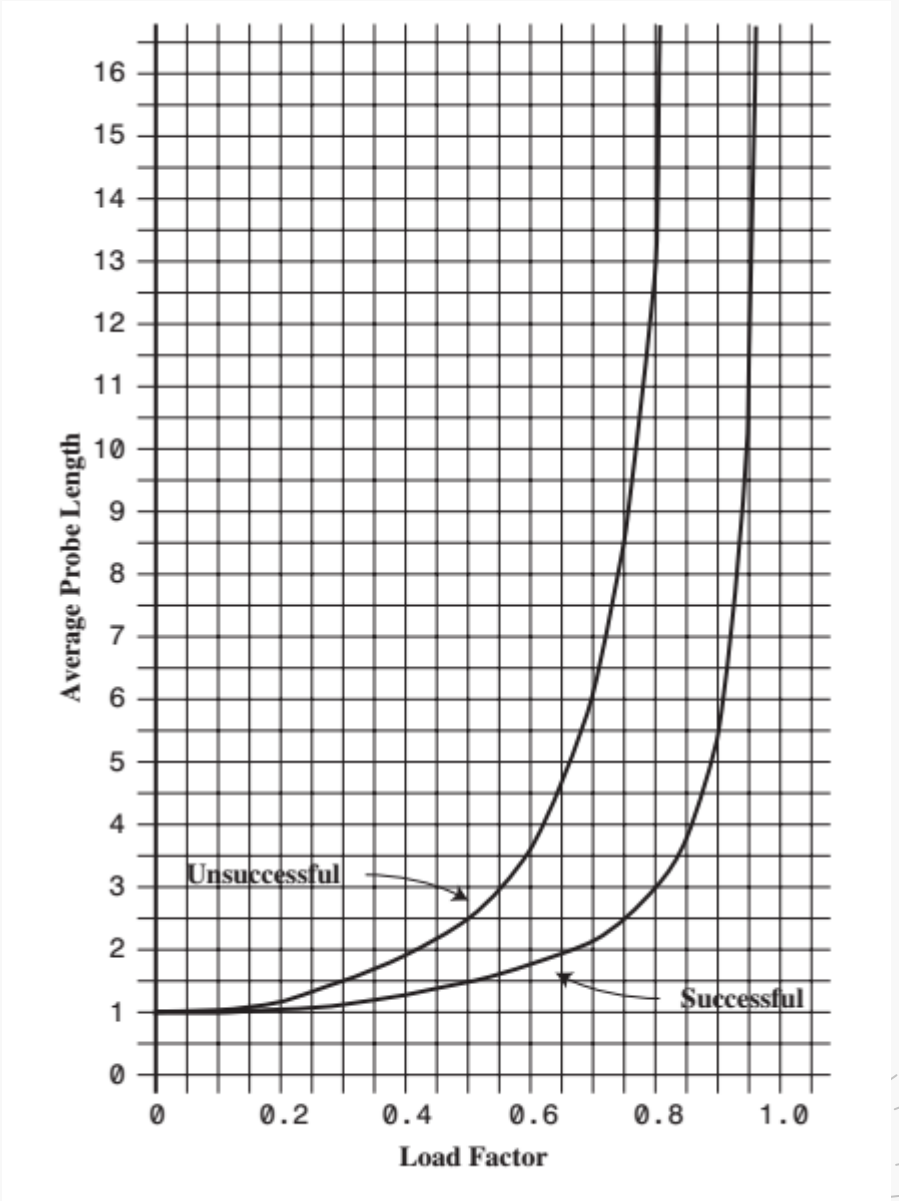
Perfect Hash Function

- If hash function **h** transforms different keys into different numbers, it is called a ***perfect hash function***.
- If a function requires only as many cells in the table as the number of data so that no empty cell remains after hashing is completed, it is called a **minimal perfect hash function**.
- **Cichelli's method** is an algorithm to construct a minimal perfect hash function

Linear probing

Successful search: $P = (1 + 1 / (1 - L)^2) / 2$

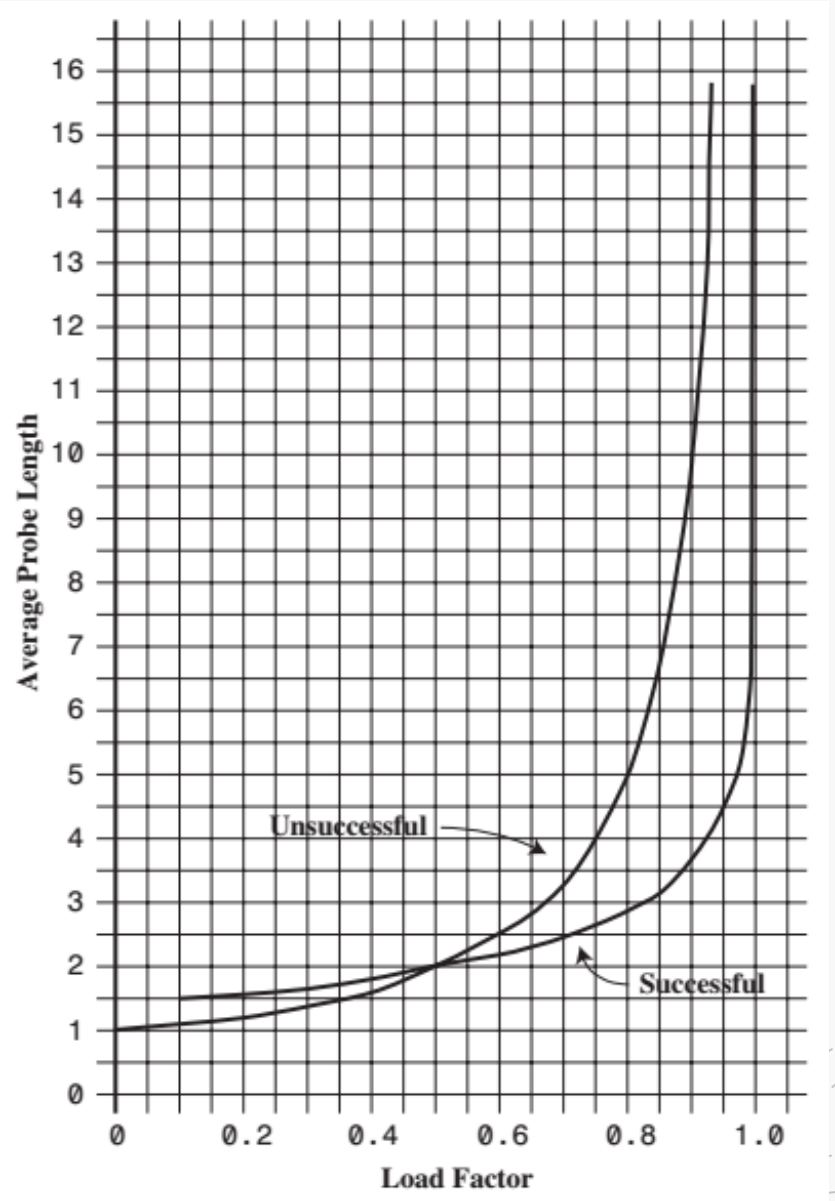
Unsuccessful search: $P = (1 + 1 / (1 - L)) / 2$



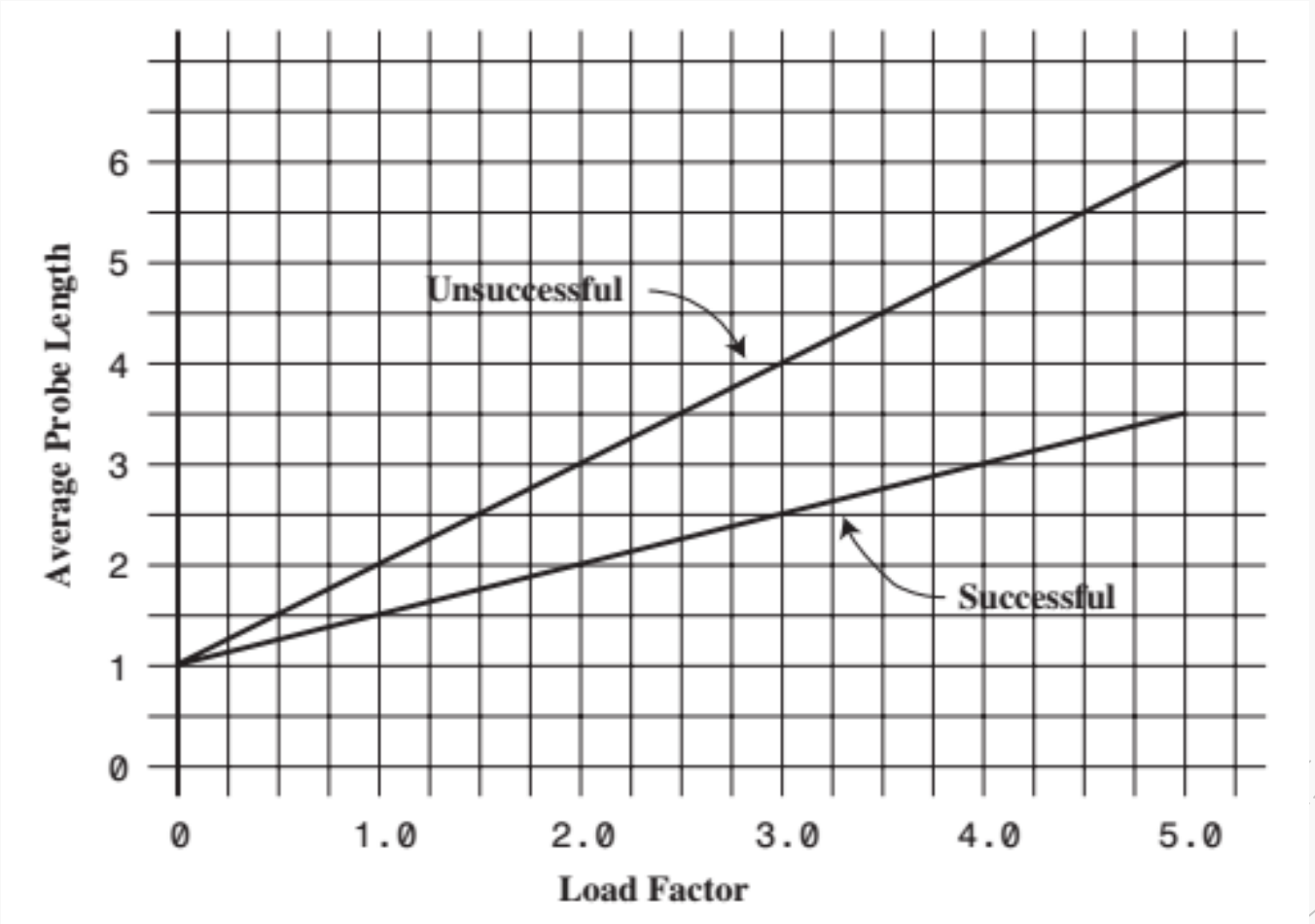
Quadratic probing and double hashing

Successful search: $-\log_2(1-\text{loadFactor}) / \text{loadFactor}$

Unsuccessful search: 1 / (1-loadFactor)



Separate chaining



HashMap class

Some main methods	
void	**clear**() Removes all mappings from this map.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Object	**get**(Object key) Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
Object	**remove**(Object key) Removes the mapping for this key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.

Applications of Hashing

- There are many areas where hashing is applicable.Here are common ones:

1. Databases: Efficient retrieval of records.
2. Compilers: Symbol tables.
3. Games: Lookup board configuration to find the move that goes with it.
4. UNIX shell: Quick command lookup.
5. IP Routing: Fast IP address lookup.

Open addressing refers to

a. keeping many of the cells in the array unoccupied.

b. keeping an open mind about which address to use.

c. probing at cell x+1, x+2, and so on until an empty cell is found.

d. looking for another location in the array when the one you want is occupied.

The best technique when the amount of data is not well known is

a. linear probing.

b. quadratic probing.

c. double hashing.

d. separate chaining.

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io