HASH TABLES

Tran Thanh Tung

Introduction

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

More…

- 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

Content

- Introduction to Hashing
- Open addressing
- Separate chaining
- Hash functions
- Hashing efficiency
- Hashing and External Storage

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

Algorithm

Given an input object,

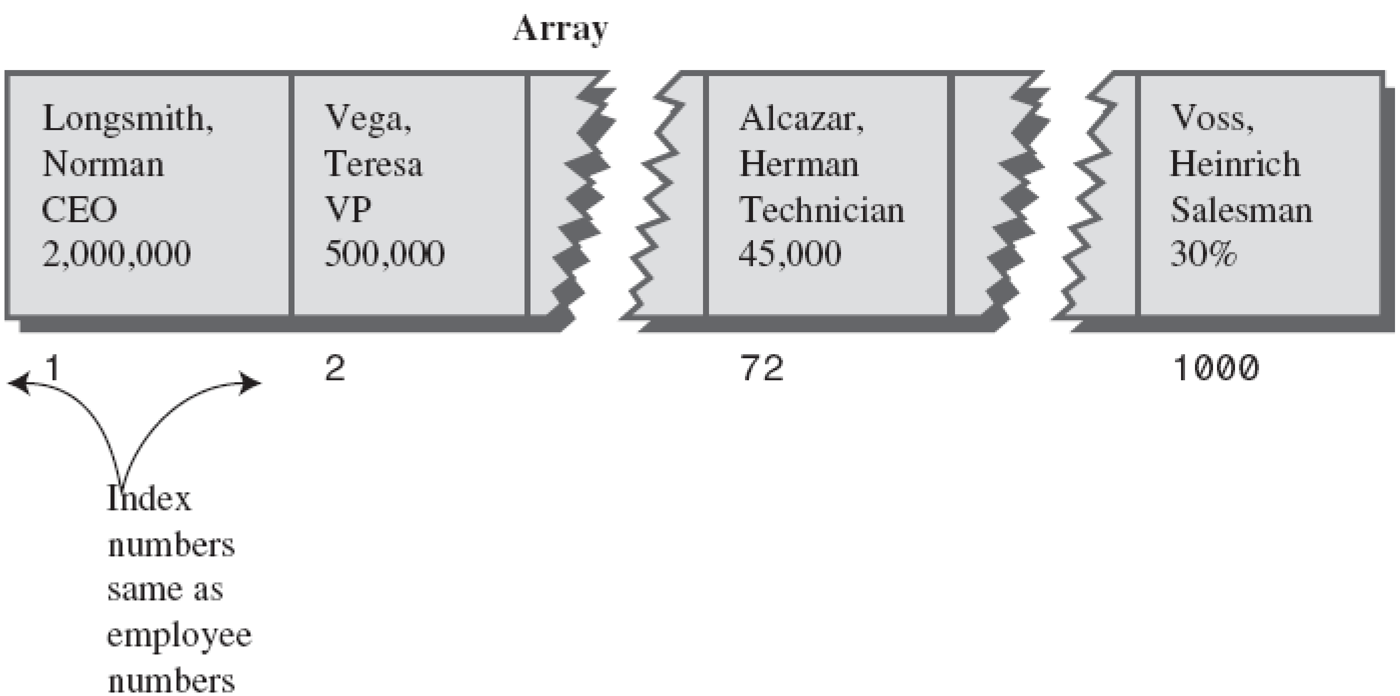
1. Access this key value,
2. Hash to a location in an array,
3. Move the object into this location (indexed by the hashed-to value)

If object have a numeric key

→ Use key as index

→ Don’t have to hash

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

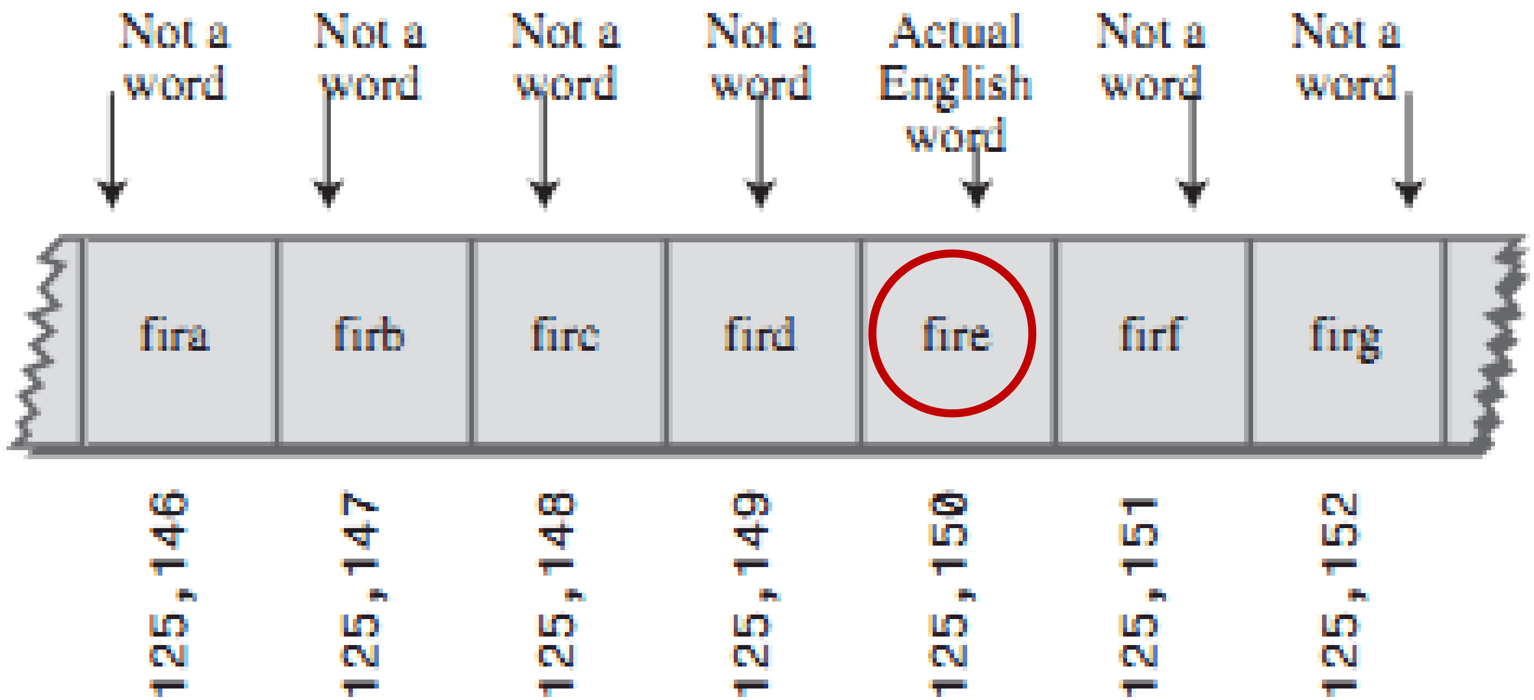
Converting Words to Numbers

- 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

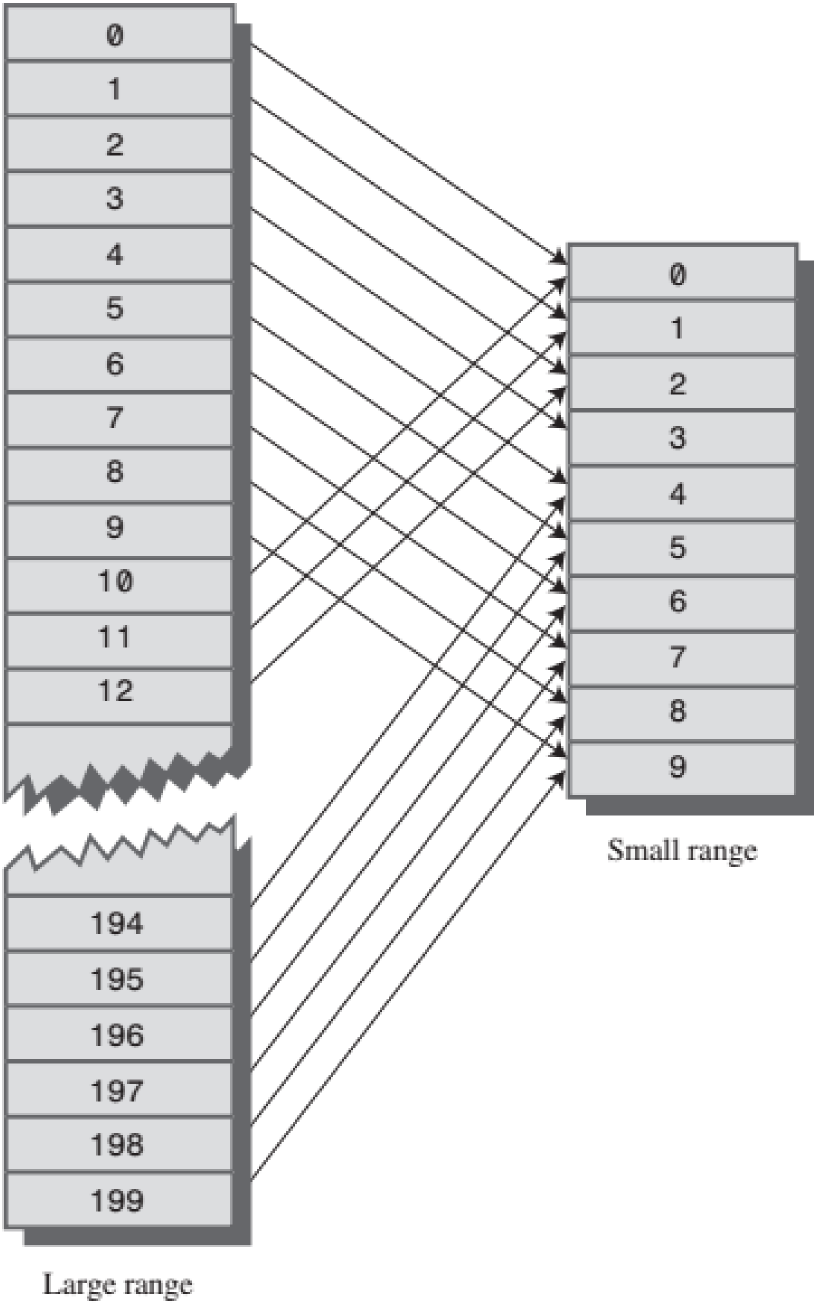
- To warranty each word have its own unique index
- Multiply by power
- Idea
  - For number
    7654 $= 7*10^3 + 6*10^2 + 5*10^1 + 4$
  - For word
    cats $= 3*27^3 + 1*27^2 + 20*27^1 + 19$
    $= 60337$

New problem: too many indices



Hash function

Hashing


0
1
2
3
4
5
6
7
8
9
10
11
12
194
195
196
197
198
199
Large range
0
1
2
3
4
5
6
7
8
9
Small range

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

Organization of Chapter 11

- Much of the remainder of this chapter is devoted to collision algorithms
- Then, much later in the chapter, your author goes back to hashing algorithms
- So, 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

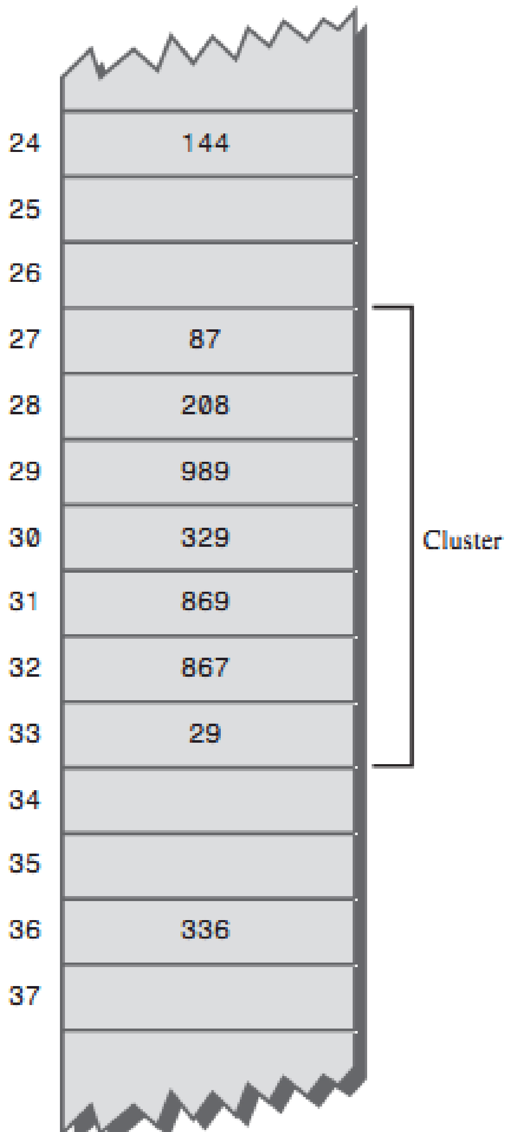
Ex:

- 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

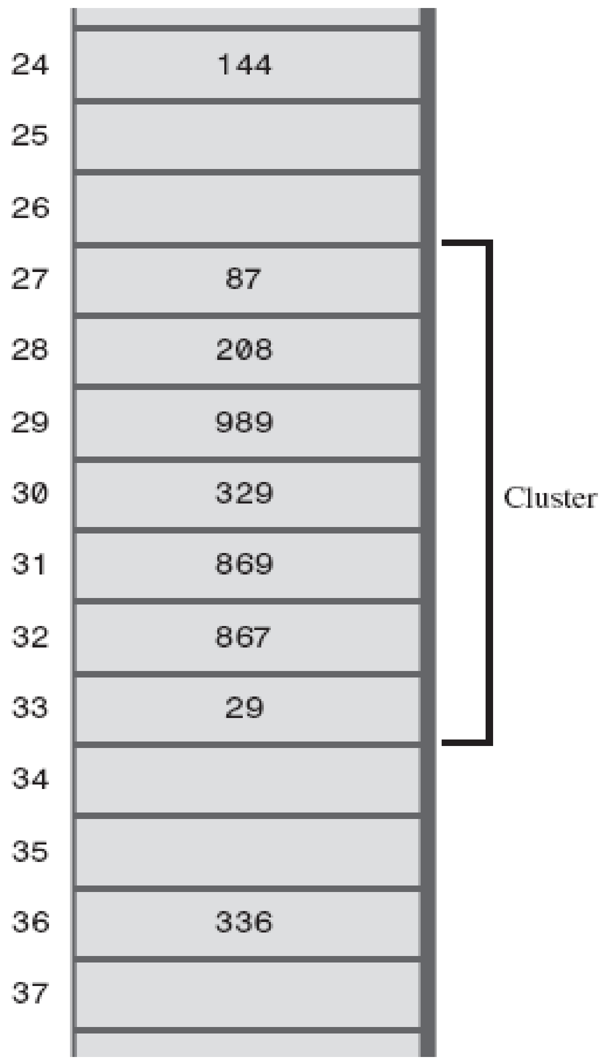
Clustering



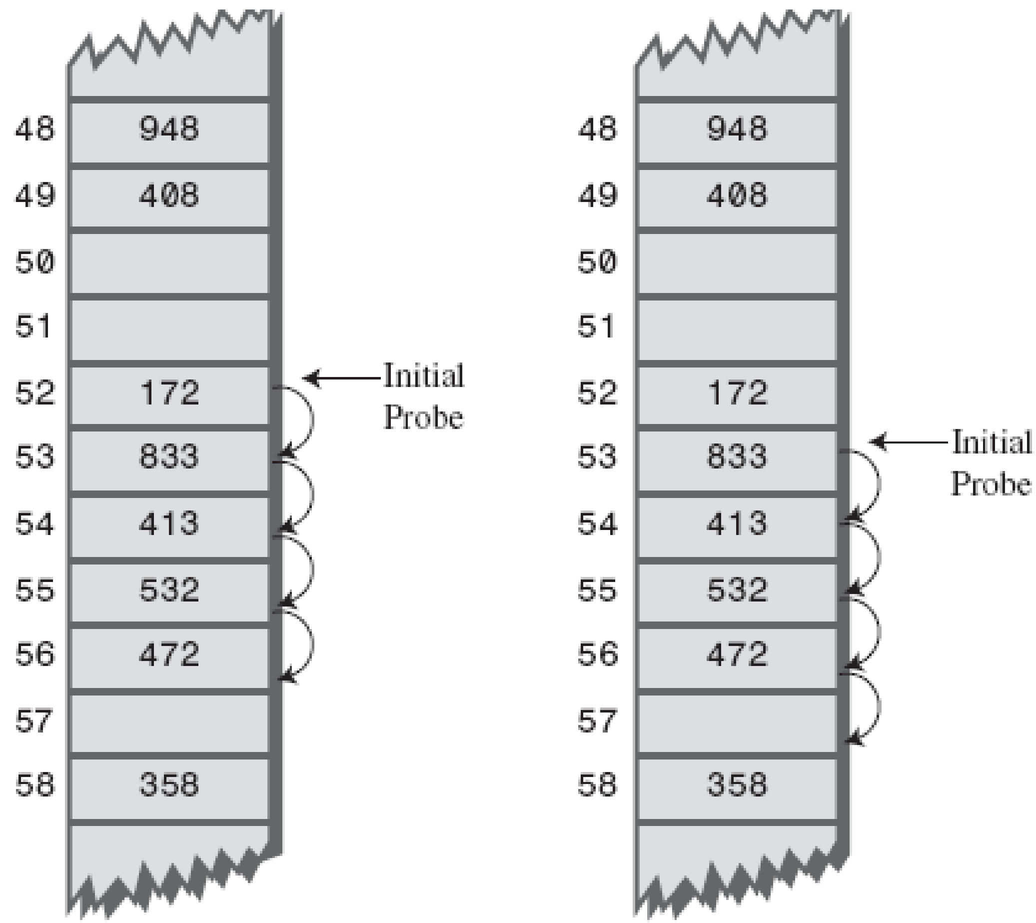
- Sequence of filled elements
- Hash table more full, the clusters grow larger
- Clustering →
  very long probe length

→ Degrade the performance

clustering.



Linear probes.



a) Successful search for 472

b) Unsuccessful search for 893

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

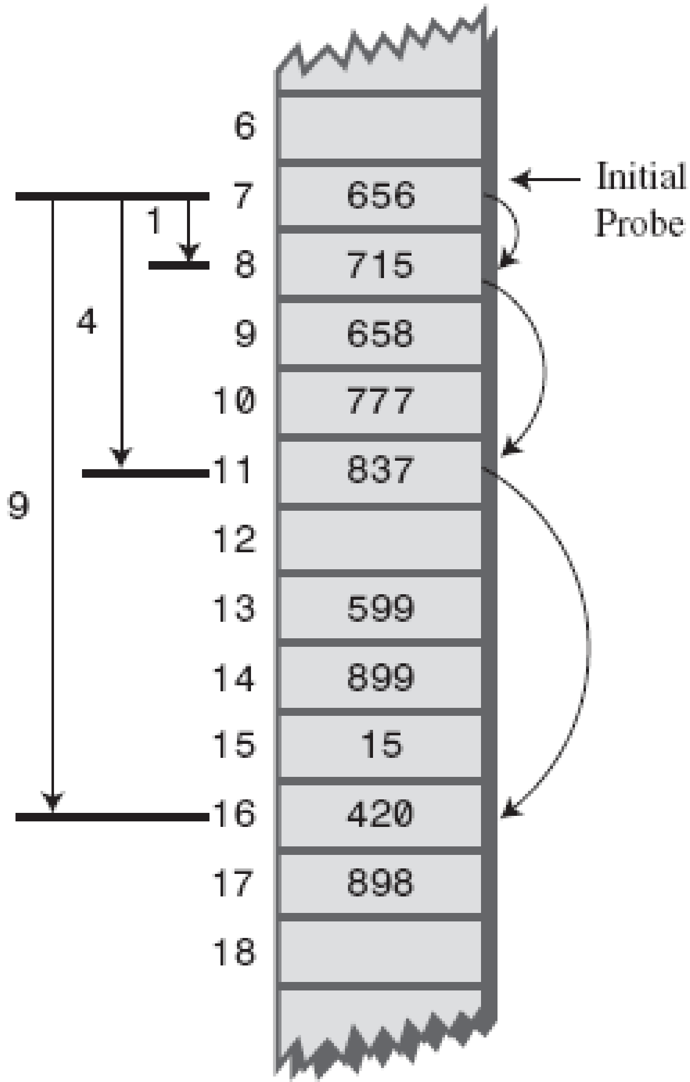
Open Addressing: Quadratic Probing

- 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…

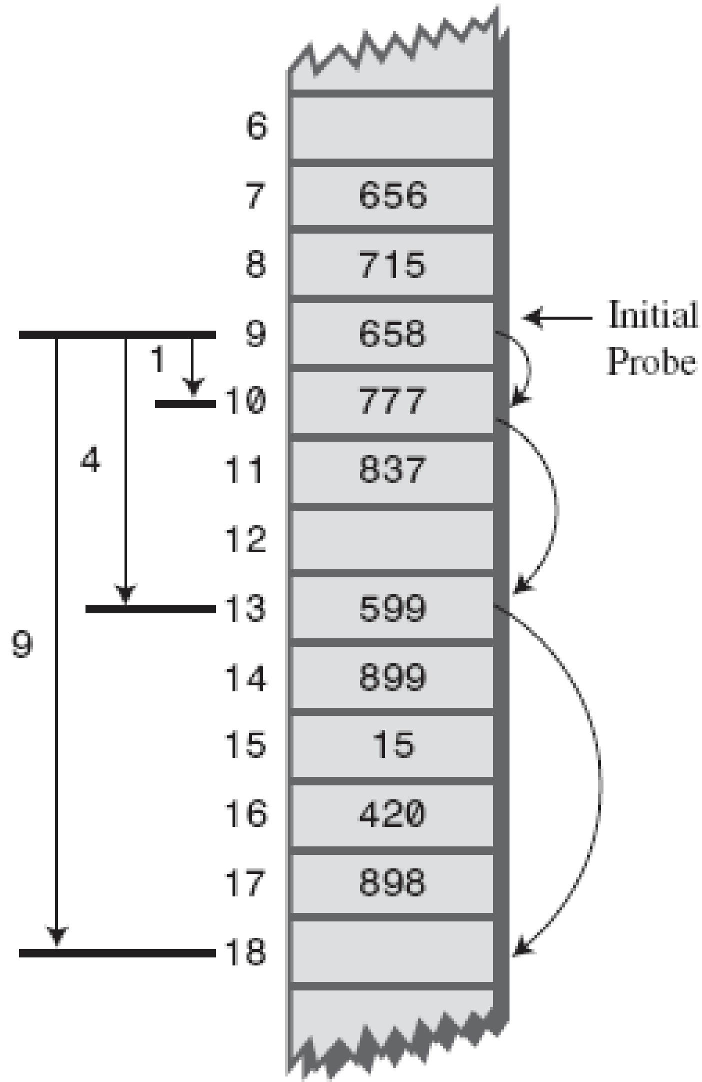
Quadratic Probing – 2

- 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



a) Successful search for 420



b) Unsuccessful search for 481

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
- 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

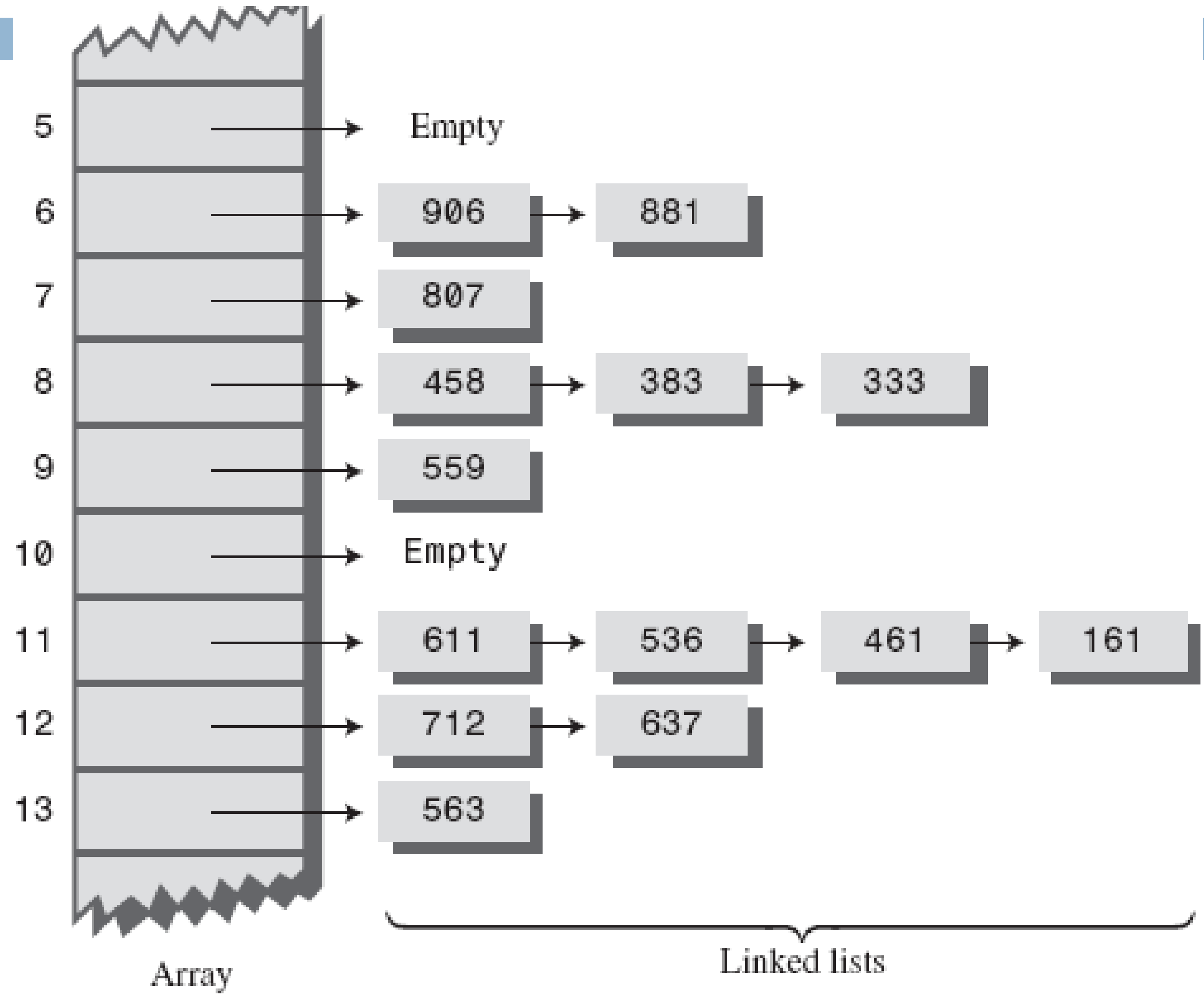
Finale

- 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 a number of 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()
```

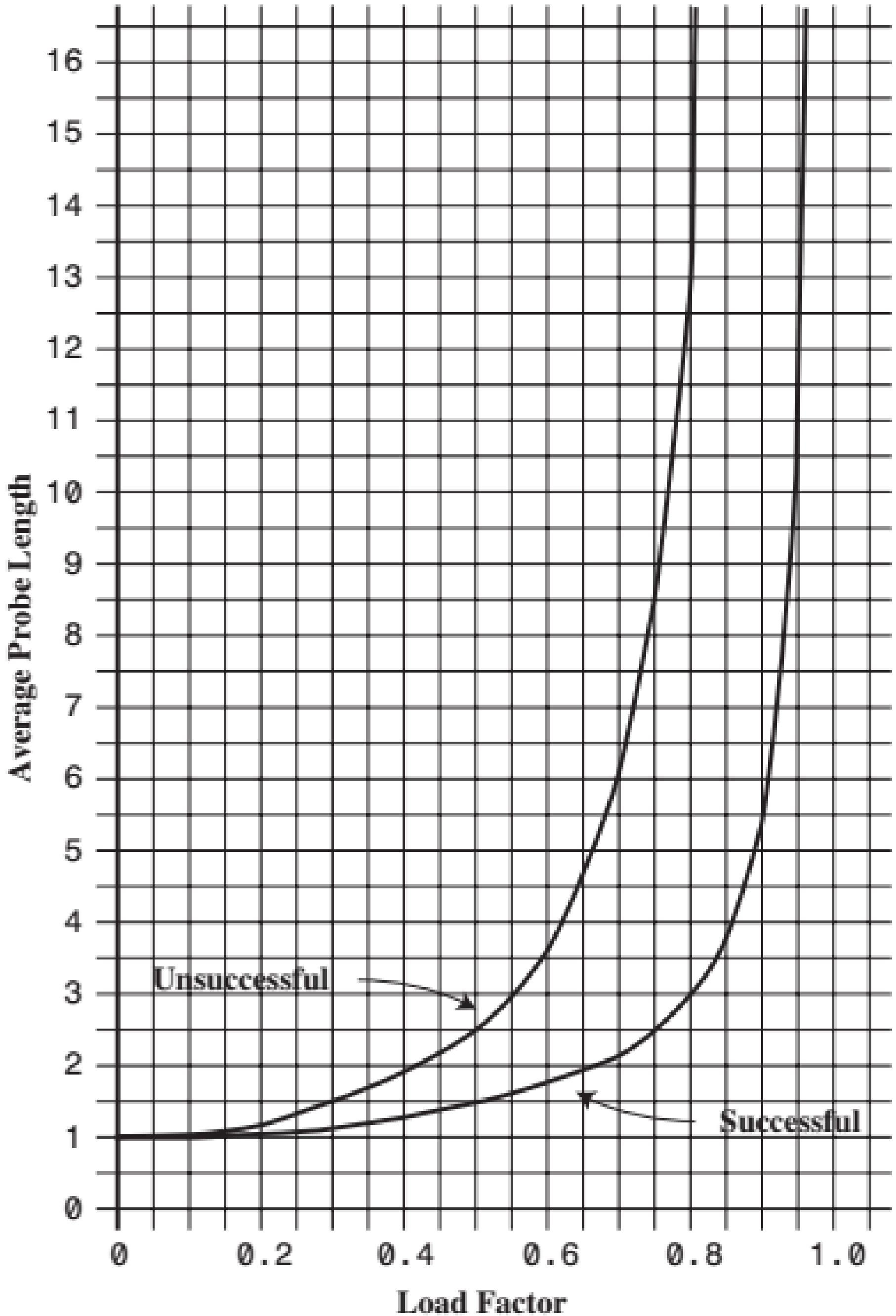
Duplicates / Deletions and Table Size in Separate Chaining

- 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!

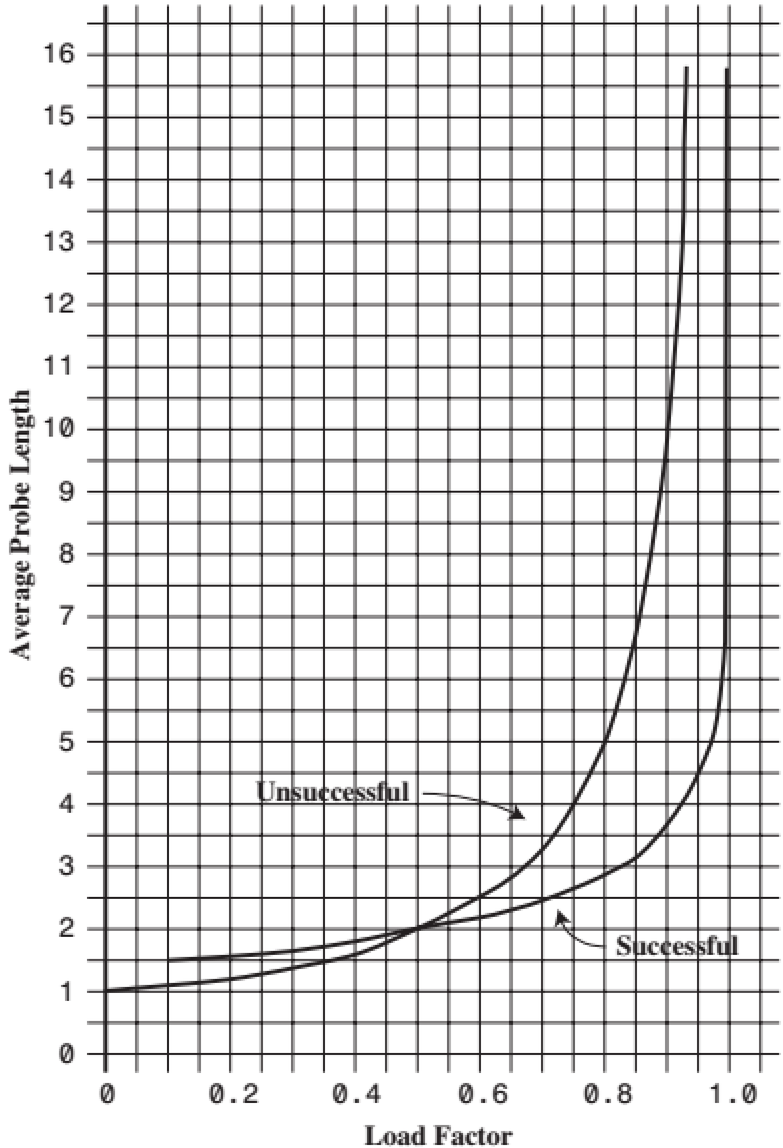
Bucket Approach

- Another approach in lieu of linked list is to use an array at each hash table location
- Called buckets
- 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

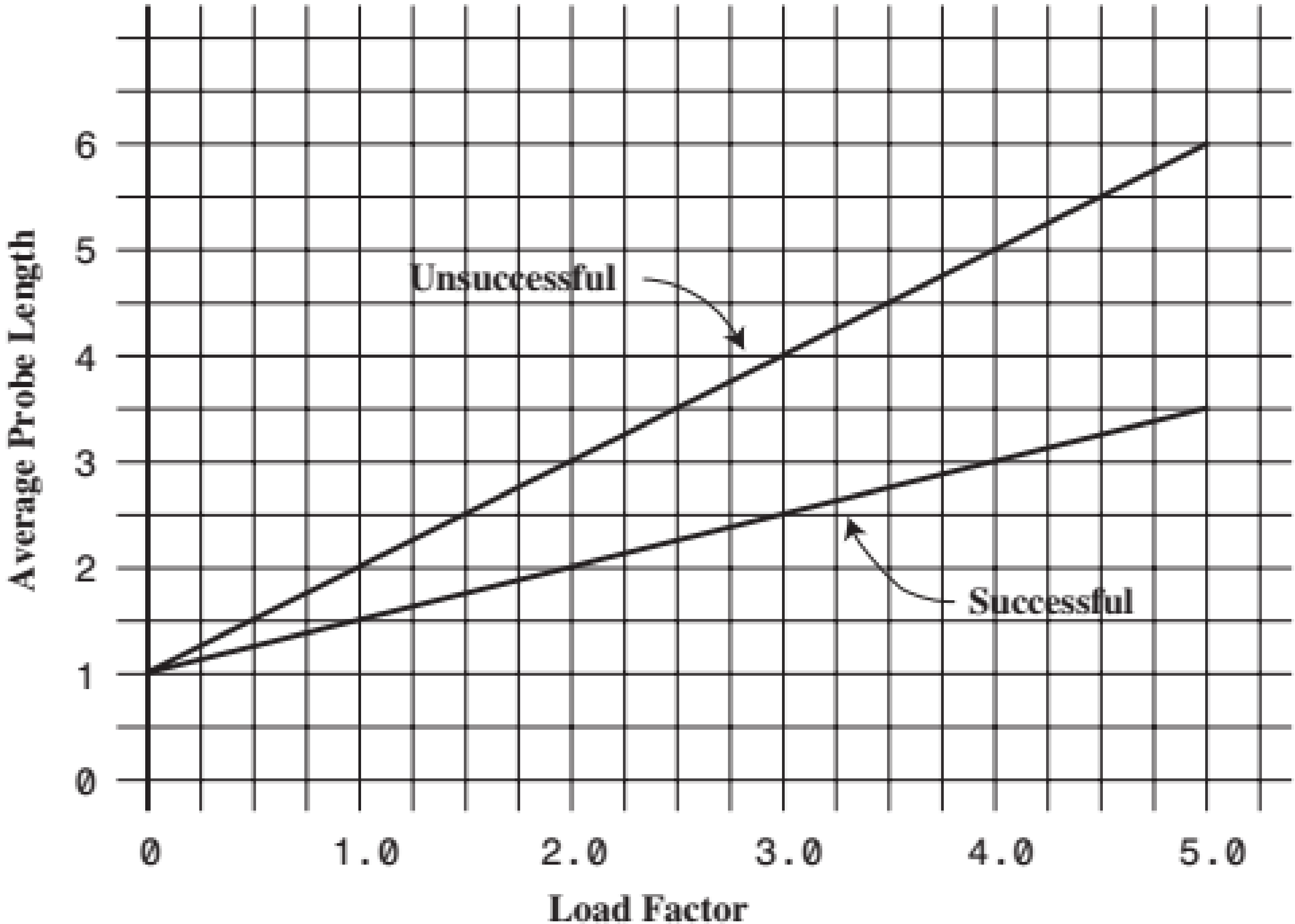
Linear probing



Quadratic probing and double hashing



Separate chaining



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.

Secondary clustering occurs because

a. many keys hash to the same location.

b. the sequence of step lengths is always the same.

c. too many items with the same key are inserted.

d. the hash function is not perfect.

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.