Hashing

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

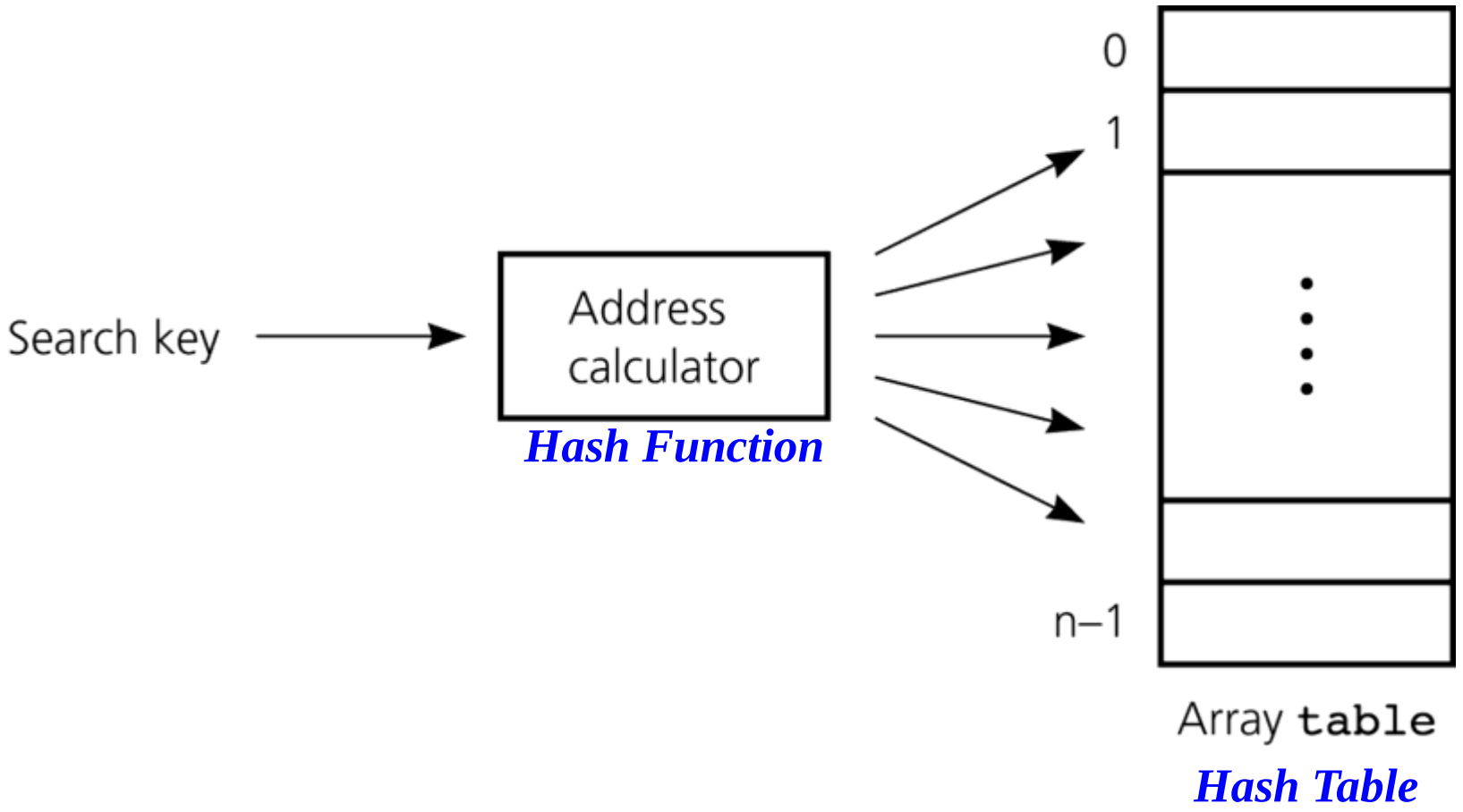
Hashing

- Using balanced search trees (2-3, 2-3-4, red-black, and AVL trees), we implement **table operations in O (logN) time**
  - retrieval, insertion, and deletion
- Can we find a data structure so that we can perform these table operations **even faster** (**e.g., in O(1) time**)?
  - **Hash Tables**

Hash Tables

- In hash tables, we have
  - **An array** (index ranges 0 … n – 1) and
    - Each array location is called a ***bucket***
  - **An address calculator** (***hash function***), which maps a search key into an array index between 0 … n – 1

Hash Function -- Address Calculator



Hashing

- A **hash function** tells us where to place an item in array called a **hash table**.
  - This method is known as **hashing**.

- Hash function **maps a search key into an integer** between 0 and n – 1.
  - We can have different hash functions.
  - Hash function depends on key type (int, string, ...)
    - E.g., **h(x) = x mod n**, where x is an integer

Collisions

- A **perfect hash function** maps each search key into a unique location of the hash table.
  - A perfect hash function is possible if we know all search keys in advance.
  - In practice we do not know all search keys, and thus, a hash function can map more than one key into the same location.

- **Collisions** occur when a hash function maps more than one item into the same array location.
  - We have to resolve the collisions using a certain mechanism.

Hash Functions

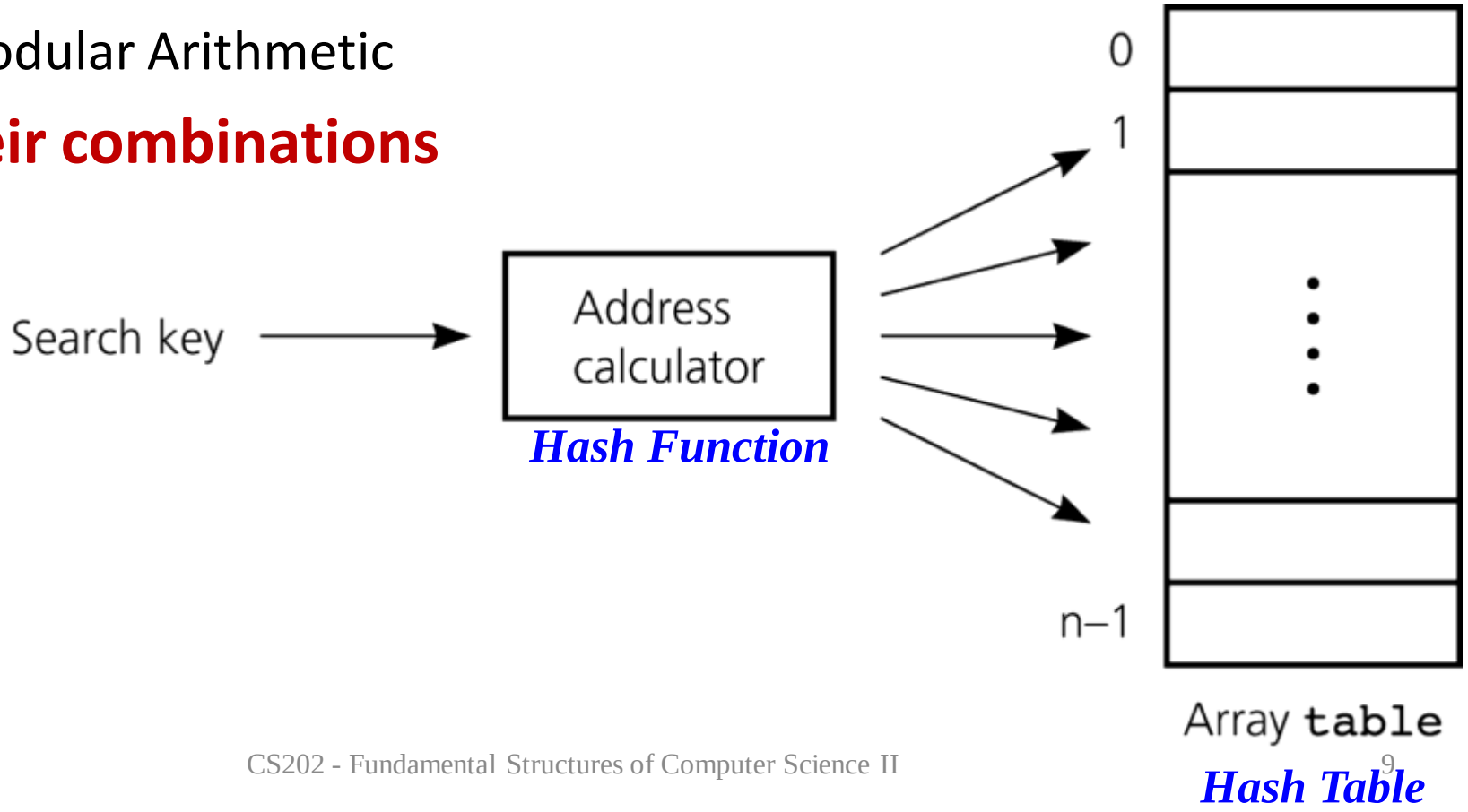
- We can design different hash functions.
- But a **good hash function** should
  - be easy and fast to compute
  - place items uniformly (evenly) throughout the hash table.
- We will consider only **integer hash functions**
  - On a computer, everything is represented with bits.
  - They can be converted into integers.
    - 10010101010010100001 10…. remember?

Everything is an Integer

- If search keys are strings, think of them as integers, and apply a hash function for integers.
- For example, strings can be encoded using ASCII codes of characters.
- Consider the string "NOTE"
  - ASCII code of **N** is 4Eh (01001110), **O** is 4Fh (01001111),
    T is 54h(01010100), **E** is 45h (01000101)
  - Concatenate four binary numbers to get a new binary number
    01001110010011110101010001000101= 4E4F5445h = **1313821765**

How to Design a Hash Function?

- **Three possibilities**
  1. Selecting digits
  2. Folding
  3. Modular Arithmetic
- **Or, their combinations**



Hash Functions -- Selecting Digits

- **Select certain digits** and combine to create the address.
- For example, suppose that we have 11-digit Turkish nationality ID's
  - Define a hash function that selects the 2nd and 5th most significant digits

    h(0**3**34**7**5678) = **37**

    h(0**2**34**5**5678) = **25**
  - Define the table size as 100
- Is this a good hash function?
  - No, since it does not place items uniformly.

Hash Functions -- Folding

- **Folding –** selecting all digits and adding them.

- For example, suppose previous nine-digit numbers
  - Define a hash function that selects all digits and adds them
    h(033475678) = **0 + 3 + 3 + 4 + 7 + 5 + 6 + 7 + 8** = 43
    h(023455678) = **0 + 2 + 3 + 4 + 5 + 5 + 6 + 7 + 8** = 40
  - Define the table size as 82

- We can select a group of digits and add the digits in this group as well.

Hash Functions -- Modular Arithmetic

- **Modular arithmetic –** provides a simple and effective hash function.

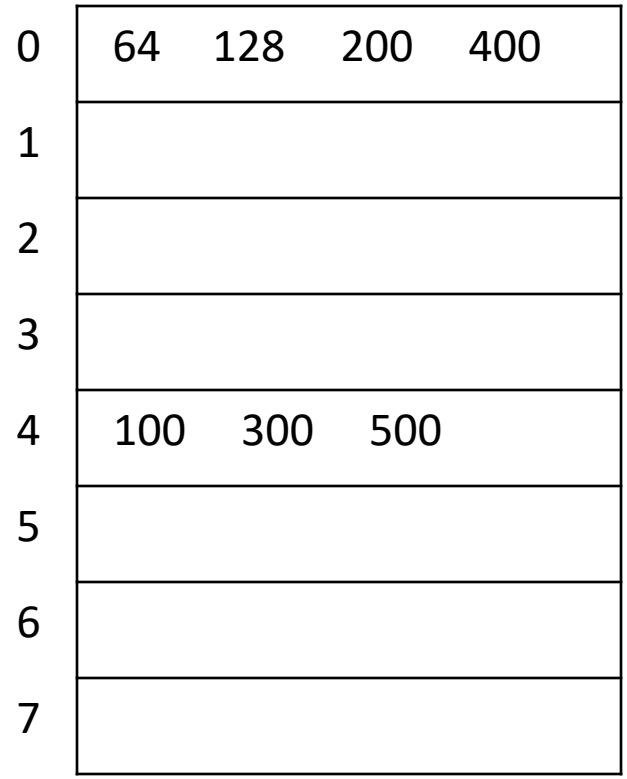
$$h(x) = x \bmod \text{tableSize}$$

- The table size should be a prime number.
  - ***Why? Think about it.***

- We will use modular arithmetic as our hash function in the rest of our discussions.

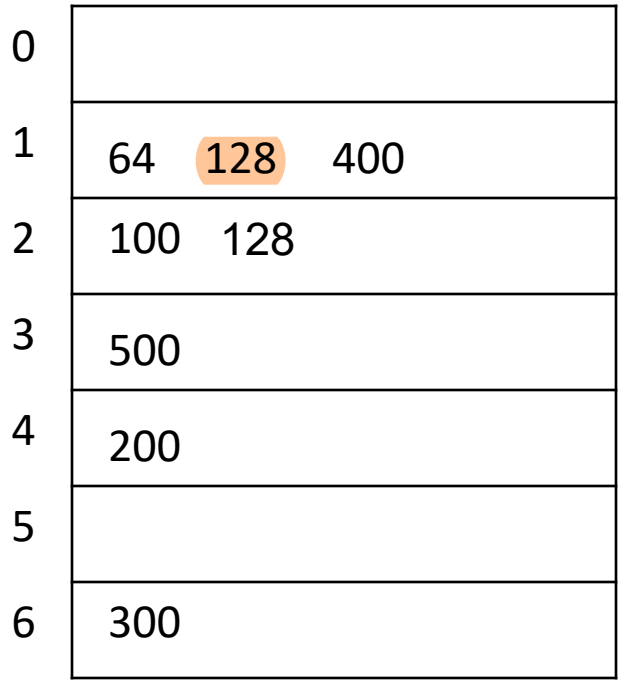
Why Primes?

- Assume you hash the following with x mod 8:
  - 64, 100, 128, 200, 300, 400, 500



Why Primes?

- Now try it with x mod 7
  - 64, 100, 128, 200, 300, 400, 500



Rationale

- If we are adding numbers a_1, a_2, a_3 … a_4 to a table of size m
  - All values will be hashed into multiples of
    gcd(a_1, a_2, a_3 … a_4 ,m)
  - For example, if we are adding 64, 100, 128, 200, 300, 400, 500 to a table of size 8, all values will be hashed to 0 or 4
    gcd(64,100,128,200,300,400,500, **8**) = 4
  - When m is a prime gcd(a_1, a_2, a_3 … a_4 ,m) = 1, all values will be hashed to anywhere
    gcd(64,100,128,200,300,400,500,**7**) = 1
    unless gcd(a_1, a_2, a_3 … a_4) = m, which is rare.

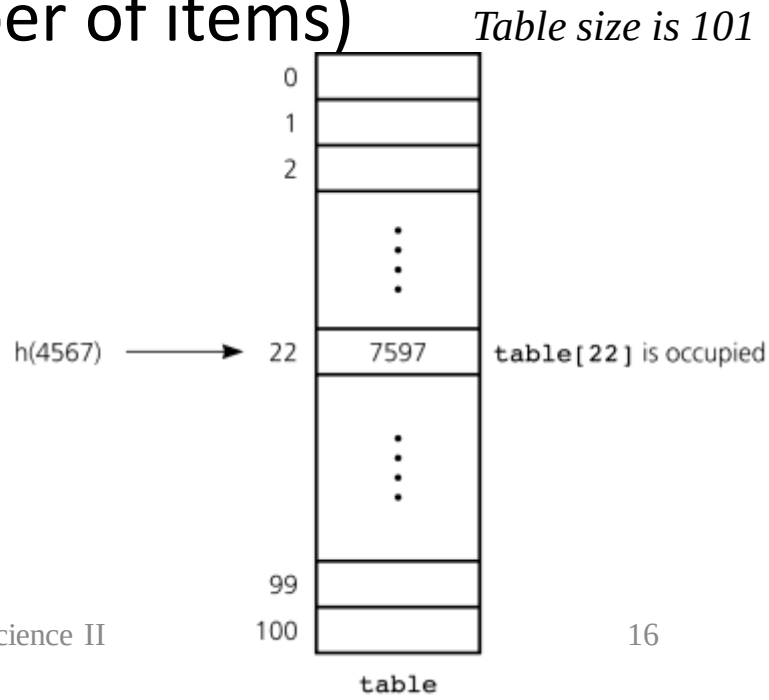
Collision Resolution

- **Collision resolution –** two general approaches
  - **Open Addressing**

    Each entry holds one item
  - **Chaining**

    Each entry can hold more than one item

    (**Buckets** – hold certain number of items)



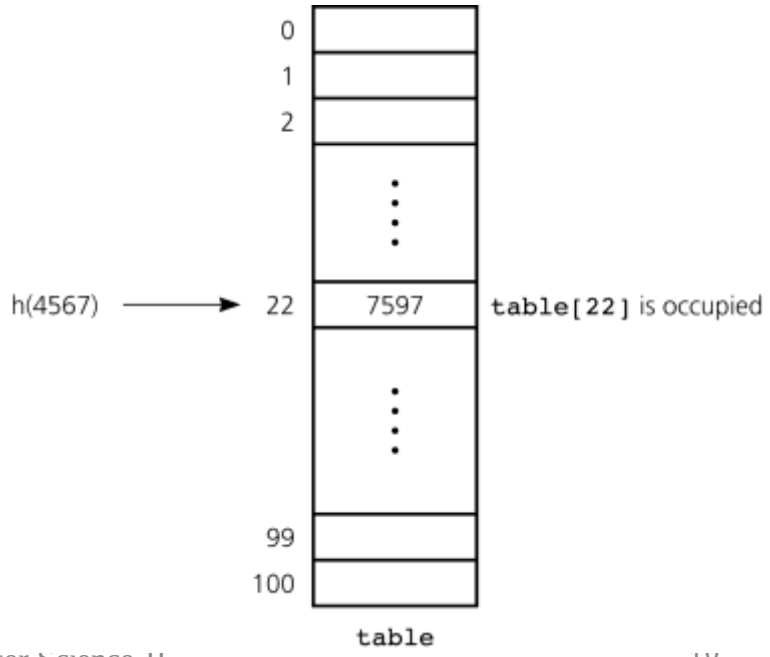
Open Addressing

- **Open addressing** – probes for some other empty location when a collision occurs.

- **Probe sequence:** sequence of examined locations. Different open-addressing schemes:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

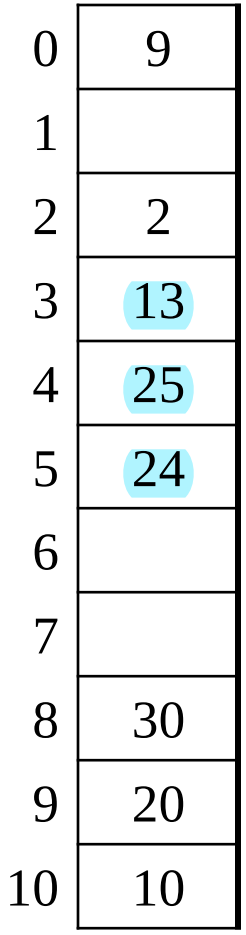
Open Addressing -- Linear Probing

- **linear probing**: search table sequentially starting from the original hash location.
  - Check next location, if location is occupied.
  - Wrap around from last to first table location



Linear Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ 2+1=3
    - 25 mod 11 = 3 ➔ 3+1=4
    - 24 mod 11 = 2 ➔ 2+1, 2+2, 2+3=5
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ 9+1, 9+2 mod 11 =0



Linear Probing -- Clustering Problem

- One of the problems with linear probing is that table items tend to **cluster** together in the hash table.
  - i.e. table contains groups of consecutively occupied locations.

- This phenomenon is called **primary clustering**.
  - Clusters can get close to one another, and merge into a larger cluster.
  - Thus, the one part of the table might be quite dense, even though another part has relatively few items.
  - Primary clustering causes long probe searches, and therefore, decreases the overall efficiency.

Open Addressing -- Quadratic Probing

- **Quadratic probing:** almost eliminates clustering problem

- Approach:
  - Start from the original hash location `i`
  - If location is occupied, check locations `i+1²`, `i+2²`, `i+3²`, `i+4²` `...`
  - Wrap around table, if necessary.

Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ $2+1^2=3$
    - 25 mod 11 = 3 ➔ $3+1^2=4$
    - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

Open Addressing -- Double Hashing

- **Double hashing** also reduces clustering.
- **Idea**: increment using a **second hash function h_2**. Should satisfy:

$$h_2(\text{key}) \neq 0$$

$$h_2 \neq h_1$$

- Probes following locations until it finds an unoccupied place

$$h_1(\text{key})$$

$$h_1(\text{key}) + h_2(\text{key})$$

$$h_1(\text{key}) + 2*h_2(\text{key}),$$

...

Double Hashing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

    $h_1(x) = x \bmod 11$

    $h_2(x) = 7 - (x \bmod 7)$
  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
    - 14 mod 11 = 3 ➔ 3+7=10
    - 91 mod 11 = 3 ➔ 3+7, 3+2*7 mod 11=6

to wrap around --> x mod(table size)

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Open Addressing -- Retrieval & Deletion

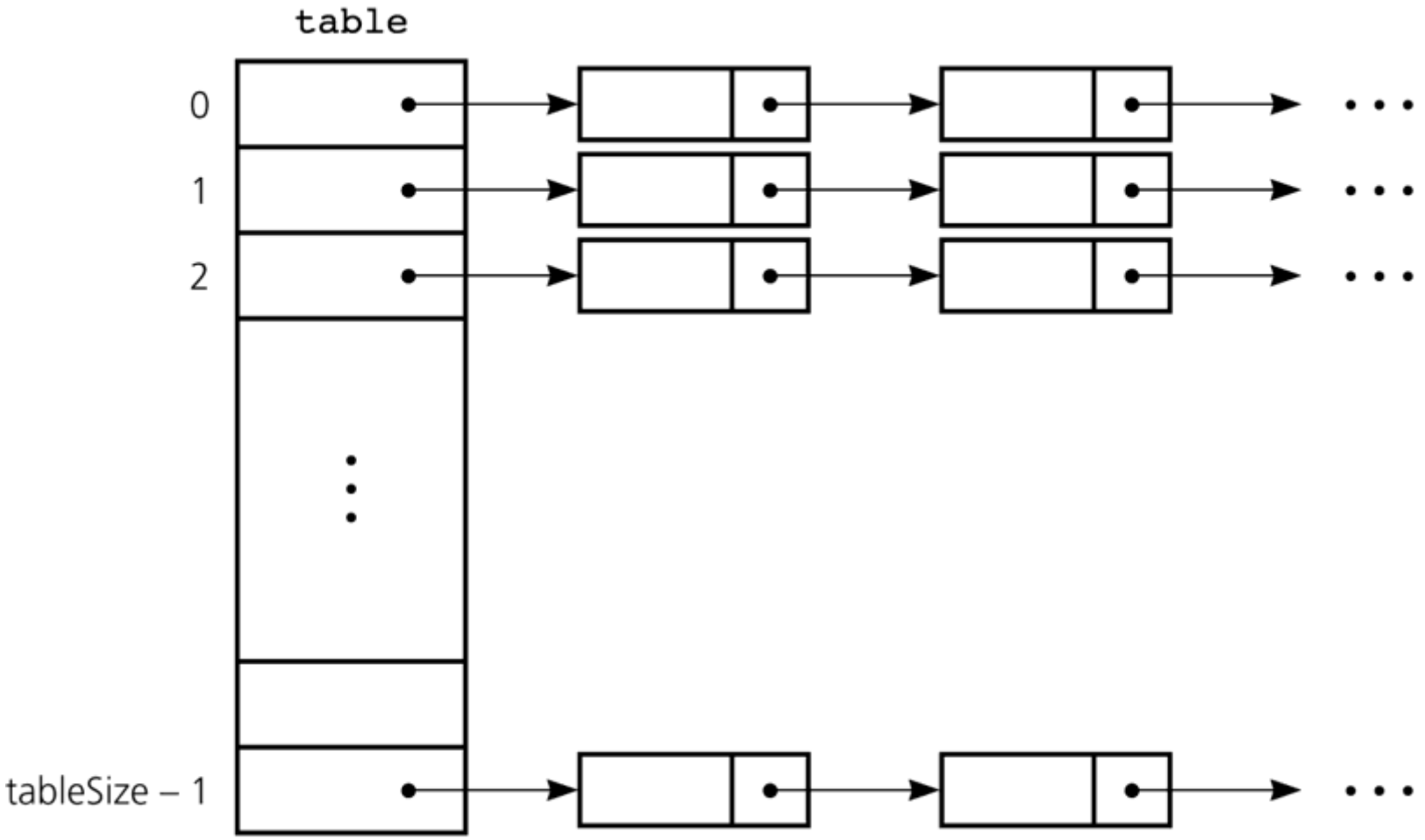
- **Retrieving an item** with a given key:
  - (same as insertion): probe the locations until we find the desired item or we reach to an empty location.

- **Deletions** in open addressing cause complications
  - We CANNOT simply delete an item from the hash table because this new empty (a deleted) location causes to stop prematurely (incorrectly) indicating a failure during a retrieval.
  - Solution: We have to have three kinds of locations in a hash table: ***Occupied, Empty, Deleted***.
  - A deleted location will be treated as an occupied location during retrieval.

Separate Chaining

- Another way to resolve collisions is to change the structure of the hash table.
  - In open-addressing, each location holds only one item.
- **Idea 1**: each location is itself an array called bucket
  - Store items that are hashed into same location in this array.
  - Problem: What will be the size of the bucket?
- **Idea 2:** each location is itself a linked list. Known as **separate-chaining**.
  - Each entry (of the hash table) is a pointer to a linked list (the chain) of the items that the hash function has mapped into that location.

Separate Chaining



Each location of the hash table contains a pointer to a linked list

Hashing Analysis

Hashing -- Analysis

- An analysis of the average-case efficiency of hashing involves the **load factor** α:

  α = (current number of items) / tableSize

- α measures how full a hash table is.
  - Hash table should not be too loaded if we want to get better performance from hashing.
- In average case analyses, we assume that the hash function uniformly distributes keys in the hash table.
- Unsuccessful searches generally require more time than successful searches.

Separate Chaining -- Analysis

- **Separate Chaining –** approximate average number of comparisons (probes) that a search requires :

$$1+\frac{\alpha}{2} \qquad \textit{for a successful search}$$

$$\alpha \qquad \textit{for an unsuccessful search}$$

- It is the most efficient collision resolution scheme.
- But it requires more storage (needs storage for pointers).
- It easily performs the deletion operation. Deletion is more difficult in open-addressing.

Linear Probing -- Analysis

- **Linear Probing –** approximate average number of comparisons (probes) that a search requires :

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right] \qquad \textit{for a successful search}$$

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] \qquad \textit{for an unsuccessful search}$$

- Insert and search cost depend on length of cluster.
  - Average length of cluster = α = N / M.
  - Worst case: all keys hash to the same cluster.

Average probes for insert/search miss

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) = \frac{(1+\alpha+2\alpha^2+3\alpha^3+4\alpha^4+\ldots)}{2}$$

Average probes for search hit

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right) = 1+(\alpha+\alpha^2+\alpha^3+\alpha^4+\ldots)/2$$

Linear Probing -- Analysis

- **Linear Probing –** approximate average number of comparisons (probes) that a search requires :

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right] \qquad \textit{for a successful search}$$

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] \qquad \textit{for an unsuccessful search}$$

- As load factor increases, number of collisions increases, causing increased search times.
- To maintain efficiency, it is important to prevent the hash table from filling up.

Linear Probing -- Analysis

Example: Find the average number of probes for a successful search and an unsuccessful search for this hash table? Use the following hash function: **h(x) = x mod 11**

Successful Search: ***Try 20, 30, 2, 13, 25, 24, 10, 9***

20: 9 **30:** 8 **2:** 2 **13:** 2,3

25: 3, 4 **24:** 2, 3, 4, 5 **10:** 10 **9:** 9, 10, 0

Avg. no of probes = (1+1+1+2+2+4+1+3)/8 = 1.9

Unsuccessful Search: ***Try 0, 1, 35, 3, 4, 5, 6, 7, 8, 31, 32***

look one more empty bucket

0: 0, 1 1: 1 35: 2, 3, 4, 5, 6

3: 3, 4, 5, 6 4: 4, 5, 6 5: 5, 6

6: 6 7: 7 8: 8, 9, 10, 0, 1

if it is empty we are done item is not in the table

31: 9, 10, 0, 1 32: 10, 0, 1

Avg. no of probes =(2+1+5+4+3+2+1+1+5+4+3)/11 = 2.8

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Quadratic Probing & Double Hashing -- Analysis

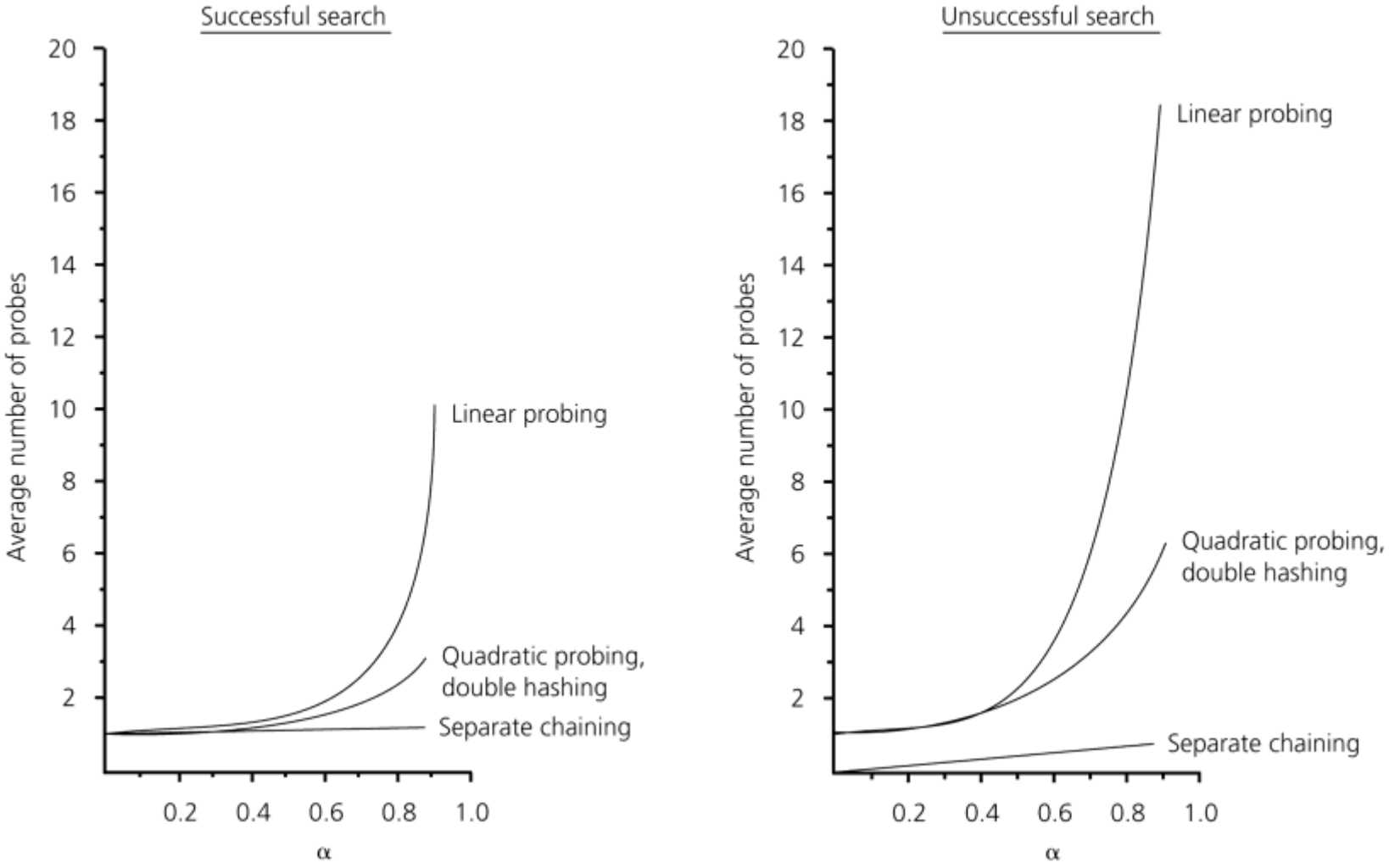
- The approximate average number of comparisons (probes) that a search requires is given as follows:

$$\left[\frac{1}{\alpha}(\log_e \frac{1}{1-\alpha})\right] = \frac{-\log_e(1-\alpha)}{\alpha}$$ *for a successful search*

$$\frac{1}{1-\alpha}$$ *for an unsuccessful search*

- On average, both methods require fewer comparisons than linear probing.

The relative efficiency of four collision-resolution methods



What Constitutes a Good Hash Function

- A hash function should be **easy** and **fast** to compute.
- A hash function should **scatter the data evenly** throughout the hash table.
  - How well does the hash function scatter random data?
  - How well does the hash function scatter non-random data?
- Two general principles :
  1. The hash function should use entire key in the calculation.
  2. If a hash function uses modulo arithmetic, the table size should be prime.

Example: Hash Functions for Strings

Hash Function 1

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
        int hasVal = 0;

        for (int i = 0; i < key.length(); i++)
                hashVal += key[i];
        return hashVal % tableSize;
}
```

- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.

last ascii value = 127

  - e.g. Table size =10000, key length <= 8, the hash function can assume values only between 0 and 1016

Hash Function 2

- Examine only the first 3 characters of the key.

```
int hash (const string &key, int tableSize)
{
      return (key[0]+27 * key[1] + 729*key[2]) % tableSize;

}
```

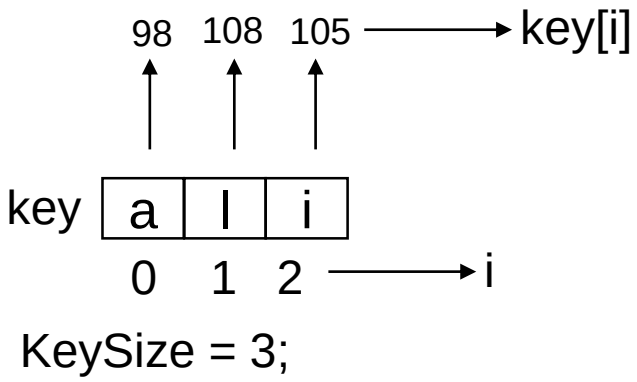
- In theory, **26 * 26 * 26 = 17576** different words can be generated. However, English is not random, only **2851** different combinations are possible.
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

Hash Function 3

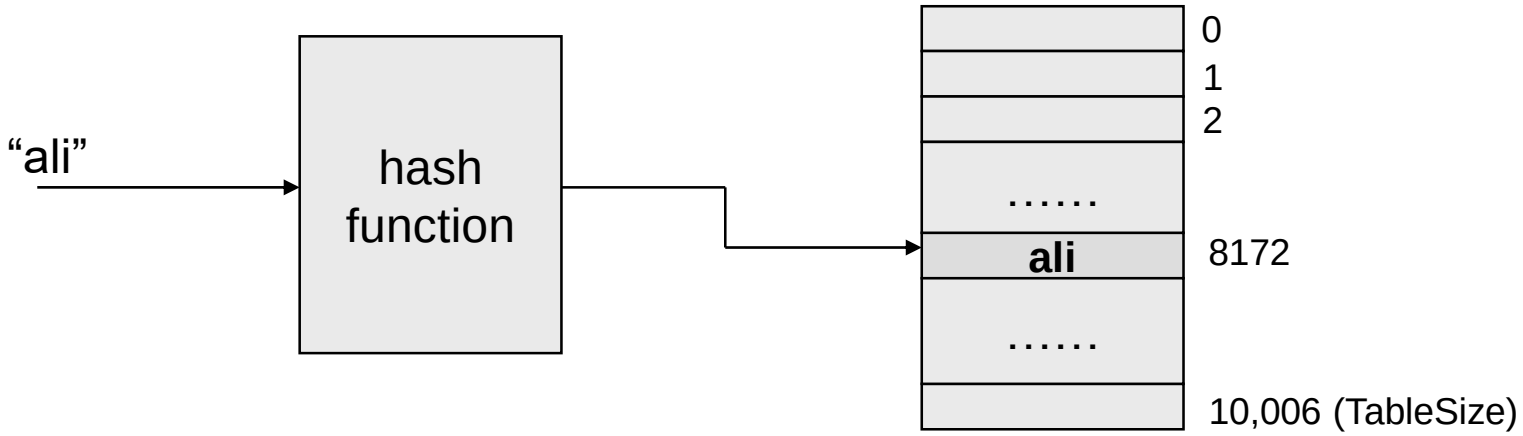
$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

```
int hash (const string &key, int tableSize)
{
   int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
       hashVal = 37 * hashVal + key[i];

   hashVal %=tableSize;
   if (hashVal < 0)   /* in case overflows occurs */
       hashVal += tableSize;

   return hashVal;
};
```

Hash function for strings:



hash("ali") = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172



Hash Table versus Search Trees

- In most of the operations, the hash table performs better than search trees.

- However, traversing the data in the hash table in a sorted order is very difficult.
  - For similar operations, the hash table will not be good choice (e.g., finding all the items in a certain range).

Performance

- With either chaining or open addressing:
  - **Search** - O(1) expected, O(n) worst case
  - **Insert** - O(1) expected, O(n) worst case
  - **Delete** - O(1) expected, O(n) worst case
  - **Min**, **Max** and **Predecessor**, **Successor** - O(n+m) expected and worst case
- Pragmatically, a hash table is often the best data structure to maintain a dictionary/table. However, the worst-case time is unpredictable.
- The best worst-case bounds come from balanced binary trees.

Other applications of hash tables

- To implement Table ADT, Dictionary ADT
- Compilers
- Spelling checkers
- Games
- Substring Pattern Matching
- Searching
- Document comparison

Applications of Hashing

- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Substring Pattern Matching

- Input: A text string t and a pattern string p.
- Problem: Does t contain the pattern p as a substring, and if so where?
- e.g: Is *Bilkent* in the news?
- **Brute Force:** search for the presence of pattern string p in text t overlays the pattern string at every position in the text. → O(mn) (m: size of pattern, n: size of text)
- **Via Hashing:** compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t. → O(n)

Slide by Steven Skiena

Hashing, Hashing, and Hashing

- Udi Manber says that the three most important algorithms at Yahoo are hashing, hashing, and hashing.
- Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.

Slide by Steven Skiena

Document Comparison

- *Is this new document different from the rest in a large database? – Hash the new document, and compare it to* the hash codes of database.

- *How can I convince you that a file isn't changed? – Check* if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.

Slide by Steven Skiena