中山大學
SUN YAT-SEN UNIVERSITY

Lecture 11
Hashing

Data Structures and Algorithms

zhangzizhen@gmail.com

QQ group: 614335192

Outline

- Introduction
- Hashing
- Analysis of Hashing
- Comparison of Methods

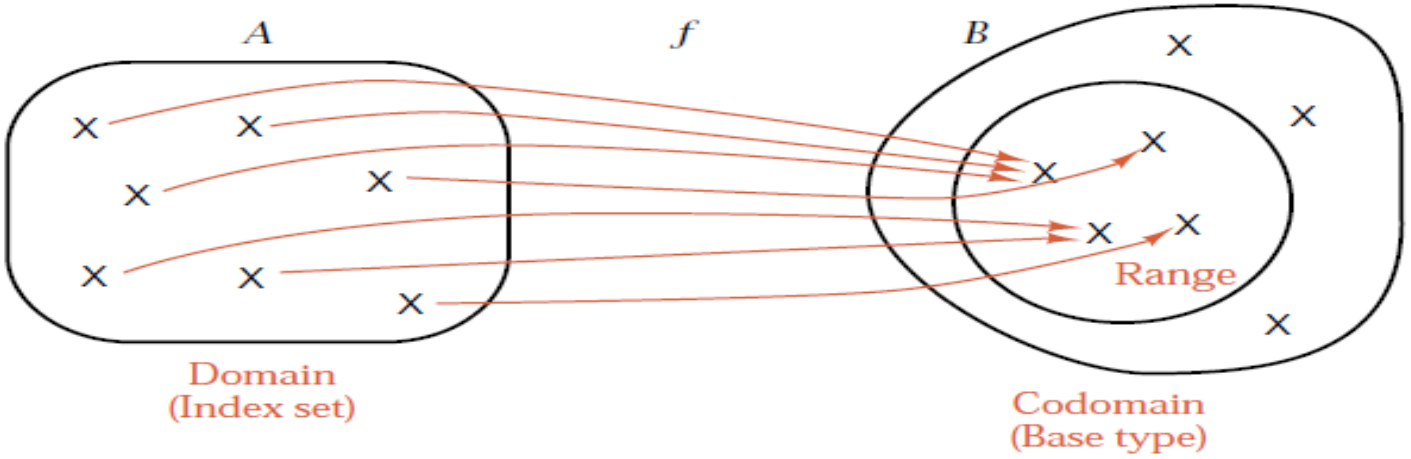
Introduction: Breaking the lg *n* Barrier

- By use of key comparisons alone, it is impossible to complete a search of *n* items in fewer than lg *n* comparisons, on average.
- Ordinary table lookup or array access requires only constant time *O(1).*
- Both table lookup and searching share the same essential purpose, that of *information retrieval. The key* used for searching and the *index* used for table lookup have the same essential purpose: one piece of information that is used to locate further information.
- Both table lookup and searching algorithms provide *functions* from a set of keys or indices to locations in a list or array.

Introduction: Breaking the lg *n* Barrier

- In this chapter we study ways to implement and access various kinds of tables in contiguous storage.
- Several steps may be needed to retrieve an entry from some kinds of tables, but the time required remains *O(1).* It is bounded by a constant that does not depend on the size of the table. Thus table lookup can be more efficient than any searching method.
- We look at methods for information retrieval using various tables, from rectangular tables to hash tables.

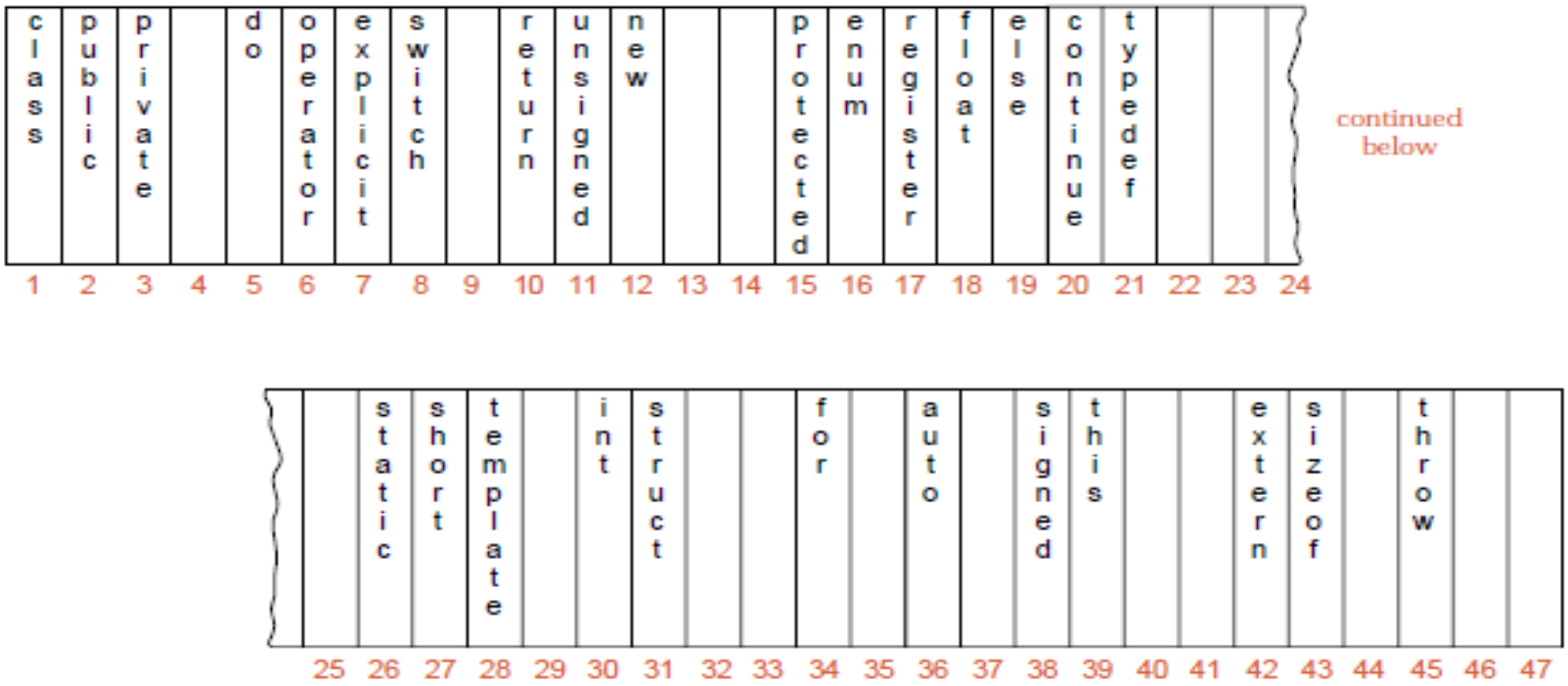
The ADT Table



DEFINITION A ***table*** with index set I and base type T is a function from I to T together with the following operations.

1. *Table access*: Evaluate the function at any index in I.
2. *Table assignment*: Modify the function by changing its value at a specified index in I to the new value specified in the assignment.
3. *Creation*: Set up a new function.
4. *Clearing*: Remove all elements from the index set I, so there is no remaining domain.
5. *Insertion*: Adjoin a new element x to the index set I and define a corresponding value of the function at x.
6. *Deletion*: Delete an element x from the index set I and restrict the function to the resulting smaller domain.

Hash Tables（哈希表/散列表）



- Idea of Hashing:
  - A table can be indexed by a key.
  - If there is a function ***f*: *K* → *N***, where *K* is the set of possible keys and *N* is the set of indices of an array, then we can store an entry with key *K* at the location *f(K)* and hence access a table entry becomes easy.

Hash Tables

- The function *f* is called ***hash function***(散列函数).
- Hash tables (散列表): a table that is stored using hashing, a structure that can map ***keys*** to ***values***.
- We start with an *array* that holds the hash table.
- Use a *hash function* to take a key and map it to some index in the array. This function will generally map several different keys to the same index.
- If the desired record is in the location given by the index, then we are finished; otherwise we must use some method to resolve the ***collision*** that may have occurred between two records wanting to go to the same location.
- To use hashing we must
  - (a) find good hash functions
  - (b) determine how to resolve collisions.

Constructing a Hash table

- Initialization: declare an array that will hold the hash table.
- Insertion: insert a record with key *k* into the hash table using hash function *h*,
  - The hash function for the key *h*(*k*) is computed.
  - If *h*(*k*) is empty, then the record is inserted; Otherwise,
  - If *h*(*k*) is not empty and the keys are different (a collision encountered), then find a new location for the record using the collision resolving method.
- Retrieval:
  - The hash function for the key *h*(*k*) is computed.
  - If the location *h*(*k*) is empty, then the retrieval is unsuccessful;
  - If the corresponding location has the desired key, then the retrieval is successful, otherwise,
  - Follow the collision resolving method until an empty location is reached or the location has the desired key.

Hash Table Specifications

```
const int hash_size = 997;          //   a prime number of appropriate size
class Hash_table {
public:
   Hash_table( );
   void clear( );
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

Hash_table :: Hash_table( );

Post: The hash table has been created and initialized to be empty.

void Hash_table :: clear( );

Post: The hash table has been cleared and is empty.

Error_code Hash_table :: retrieve(const Key &target, Record &found) const;

Post: If an entry in the hash table has key equal to target, then found takes on the value of such an entry, and success is returned. Otherwise, not_present is returned.

Choosing a Hash Function

- A hash function should be easy and quick to compute.
- A hash function should achieve an even distribution of the keys that actually occur across the range of indices.
- The usual way to make a hash function is to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that will be uniformly distributed over the range of indices.
- Note that there is nothing random about a hash function. If the function is evaluated more than once on the same key, then it must give the same result every time, so the key can be retrieved without fail.

Hash Function Construction

常用的构造哈希函数的方法：

1. 直接定址法
2. 除留余数法 (Modular arithmetic) √
3. 平方取中法
4. 折叠法 (Folding)
5. 数字分析法
6. 随机数法

Modular Arithmetic Method

- Assume all keys are integers, and define

  $$h(k) = k \bmod m.$$

- Warning: Don't pick an m that has a small divisor d. A preponderance(多数) of keys that are congruent(重叠) modulo d can adversely affect uniformity.
- Example 1：给定一组关键字为:12, 39, 18, 24, 33, 21，若取 $m = 9$, 则他们对应的哈希函数值将为:3, 3, 0, 6, 6, 3。可见，若p中含质因子3，则所有含质因子 3的关键字均映射到“3的倍数”的地址上，从而增加了“冲突”的可能。
- Example 2: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:
  - If $k = 1011000111\underbrace{011010}_{h(k)}{}_2$ and $r = 6$, then $h(k) = 011010_2$.

Modular Arithmetic Method

- Hash Function: *h(k) = k mod m*.
- Pick *m* to be a prime not too close to a power of 2 or 10.
- Sometimes, when keys are not integers, try to convert them to integers. For example, how to convert the permutation (2,4,1,3) to a number? （使用康托展开，参见 http://baike.baidu.com/view/437641.htm ）

```
int PermutationToNumber(int permutation[]， int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = i + 1; j < n; j++) {
            if (permutation[j] < permutation[i])  count++;
        }
        // factorials[j]保存着j！
        result += count * factorials[n - i - 1];
    }
    return result;
}
```

Folding Method

- We may partition the key into several parts and combine the parts in a convenient way.

- **例：**关键码为 key=25346358705，设哈希表长为三位数，则可对关键码每三位一部分来分割。

- 对于位数很多的关键码，且每一位上符号分布较均匀时，可采用此方法求得哈希地址 。

253
463
587
＋ 05
1308

Hash(key)=308

Hash Functions for Strings

- 关键字为字符串时常用的hash函数

(1) 折叠法: 把所有字符的ASCII码相加

(2) 采用ELFhash函数

```
int ELFhash(char *key)
{
  unsigned long h = 0;
  while(*key){
      h=(h<<4) + *key++;
      unsigned long g = h & 0xf0000000L;
      if (g) h^= g >> 24;
      h &= ~g;
  }
  return h % m;
}
```

Choosing a Proper Hash Function

- 在实际应用中，应根据具体情况，灵活采用不同的方法，并用实际数据测试它的性能，以便做出正确判定。 通常应考虑以下五个因素：
  - 计算哈希函数所需时间
  - 关键字长度
  - 哈希表长度（哈希地址范围）
  - 关键字分布情况
  - 记录的查找频率

Collision Resolution

- Different keys may be mapped to the same index according to the hash function, incurring hash collisions.
- To deal with collisions, there are generally two solutions:

  (1) Open addressing（开发地址）
  - Linear Probing (线性探查)
  - Quadratic probing (二次探查)

  (2) Chaining （链地址法或拉链法）

Collision Resolution with Open Addressing

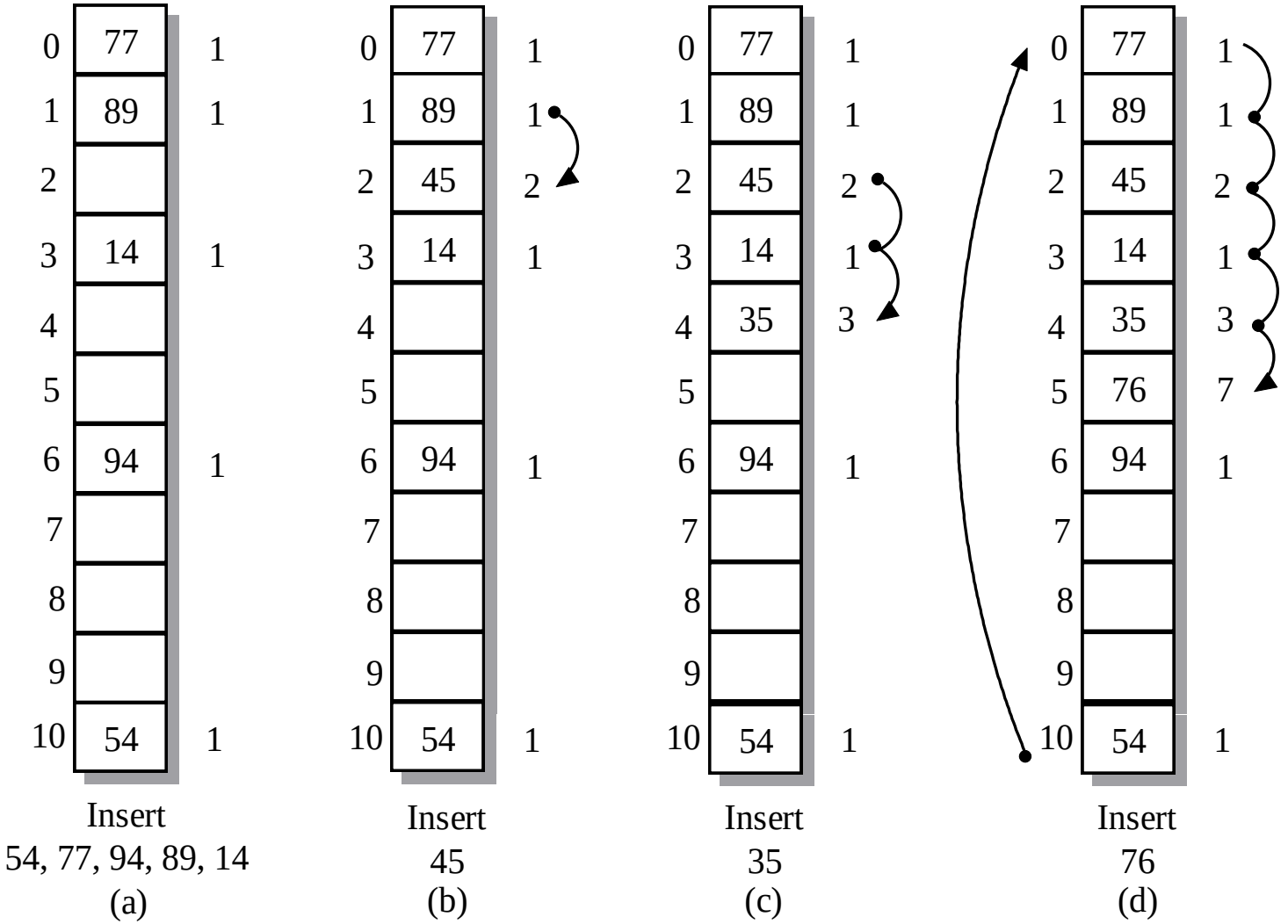
- When a collision occurs, form a sequence of addresses (探查序列) d_1, d_2, … using some method:
  - Insertion: search d1, d2, … one by one until an open address (empty unit) is found, the entry is inserted in the open address;
  - Retrieval: search d1, d2, … one by one until the desired record is found or an open address is found, the search is unsuccessful.

Linear Probing

- ***Linear probing*** starts with the hash address and searches sequentially for the target key or an empty position.
- Let h(key) = d, then the sequence of probing (探查) is d+1, d+2, …
- The array should be considered as circular, so that when the last location is reached, the search proceeds to the first location of the array.
- Drawback: may cause clustering(堆积), that is, two probing sequences for two different keys becomes one long probing sequence. Thus the sequential searches to find an empty position become longer and longer.

Linear Probing Example

- *h(k) = k mod 11*



Quadratic Probing

- Quadratic probing: if there is a collision at hash address h, probes the table at locations $h + 1$, $h + 2^2$, $h + 3^2$, …, that is, at locations $h + i^2$ (mod hashsize) for $i = 1,2,\ldots$
- The array should be considered circular.
- Example: Inserting keys {89, 18, 49, 58, 69}, hash function $h(k) = k$ mod 10.

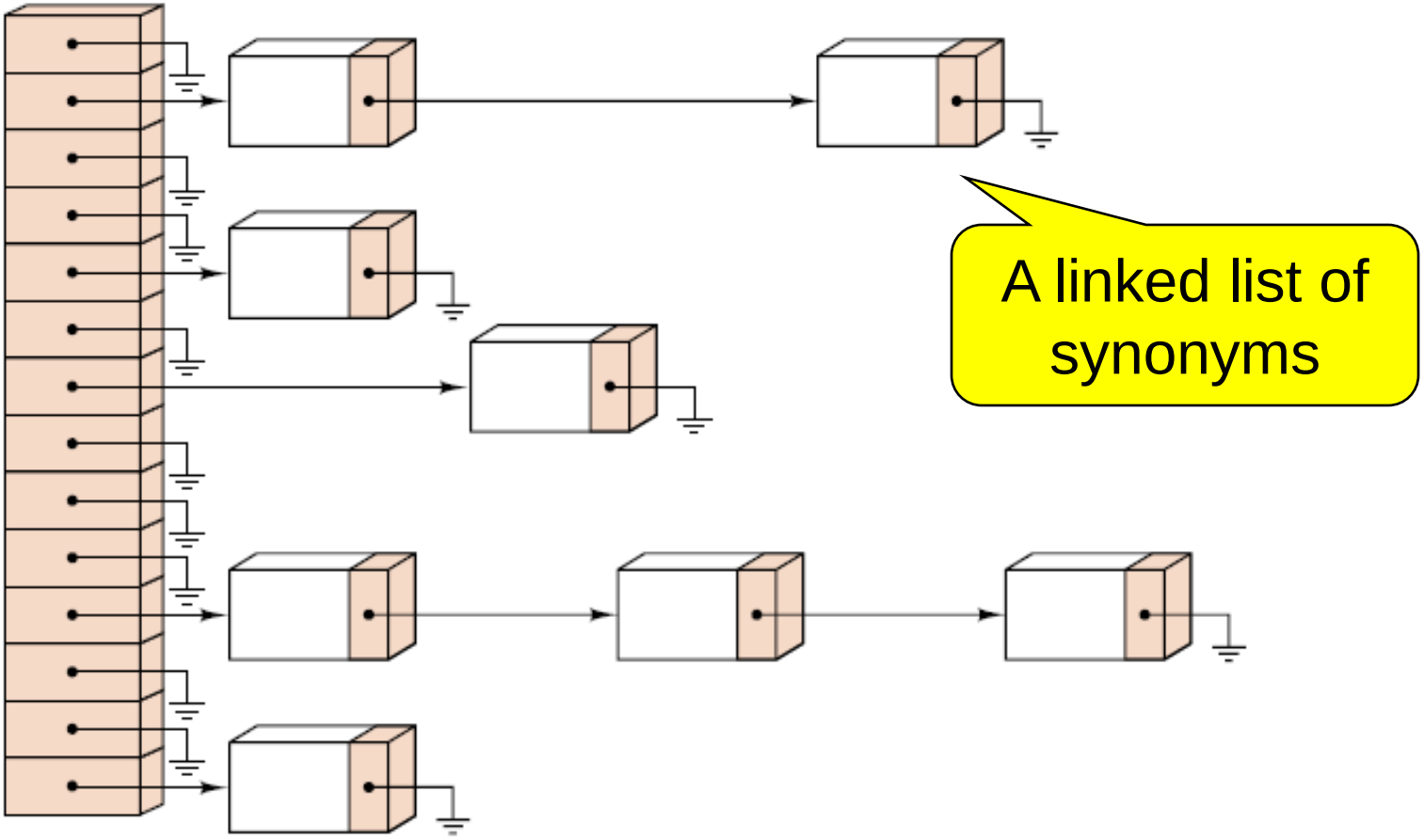
	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Deletion

- How to delete entries from a hash table? Just making the deleted location empty?
- Suppose we want to retrieve a key, we will look at the hash address and the probe sequence until the key is found or an empty location is reached. In the later case, the retrieval is unsuccessful.
- So just mark a deleted location empty will not work.
- One solution is to invent another special key that signals a deleted position, and this is empty for insertion but “not empty” for retrieval.

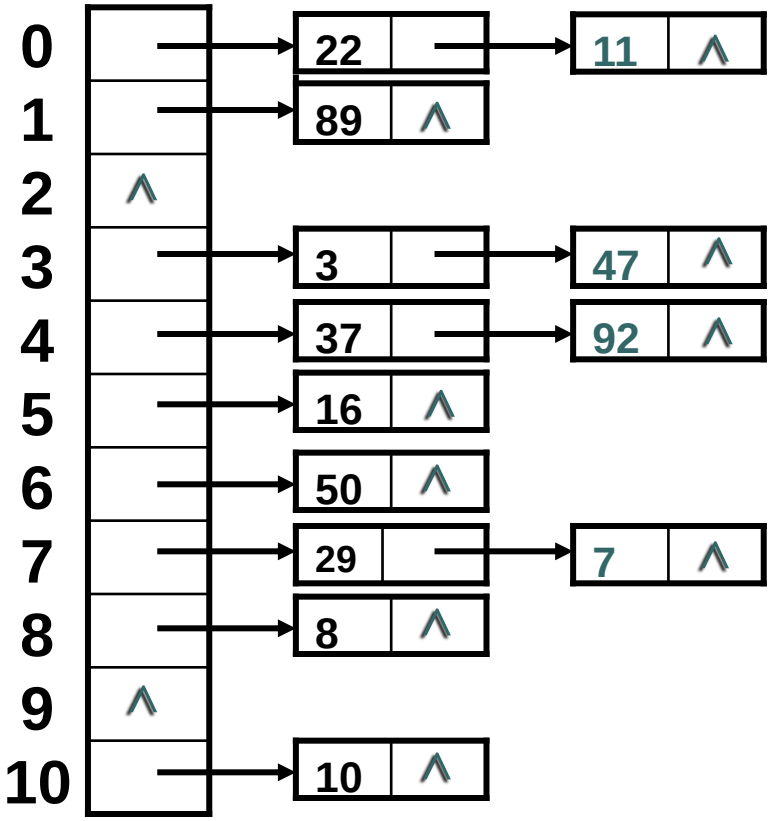
Chaining

- Idea: When there is a collision, the synonym (同义) is put into a linked list of synonyms.



Chaining Example

- 基本思想：将具有相同哈希地址的记录链成一个单链表，m个哈希地址就设m个单链表，然后用一个数组将m个单链表的表头指针存储起来，形成一个动态的结构。
- 例：设{ 47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 10 }的哈希函数为：
- Hash(key)= key mod 11,
- 注：有冲突的元素可以插在表尾,也可以插在表头。



Performance of Chaining

- If the records are large, a chained hash table can save space.
- Collision resolution with chaining is simple; clustering is no problem.
- The hash table itself can be smaller than the number of records; overflow is no problem.
- Deletion is quick and easy in a chained hash table.
- If the records are very small and the table nearly full, chaining may take more space.

```
class Hash_table {
public:
   //     Specify methods here.
private:
   List<Record> table[hash_size];
};
```

The Birthday Surprise

How many randomly chosen people need to be in a room before it becomes likely that two people will have the same birthday (month and day)?

- The probability that *m* people all have different birthdays is

$$\frac{364}{365} * \frac{363}{365} * \frac{362}{365} * \cdots * \frac{365 - m + 1}{365}.$$

- This expression becomes less than 0.5 whenever *m*>=23.
- For hashing, the birthday surprise says that for any problem of reasonable size, collisions will almost certainly occur.

Analysis of Hashing

- We count the number of probes(探查): one comparison of a key with the target.
- This number will depends on how full the table is, the load factor λ (负载因子).
- The load factor of a table is $\lambda = n/t$, n is the number of entries in the table and t the size of the array.
- λ can be seen as the average length of the chains.
- Open addressing: $\lambda \leq 1$
- Chaining: there is no limit for λ.

Analysis of Hashing

Retrieval from a chained hash table with load factor λ requires approximately $1 + \frac{1}{2}\lambda$ probes in the successful case and λ probes in the unsuccessful case.

Retrieval from a hash table with open addressing, linear probing, and load factor λ requires approximately

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

probes in the successful case and in the unsuccessful case, respectively.

Analysis of Hashing

- Theoretical comparisons:

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random probes	1.05	1.4	2.0	2.6	4.6	—
Open, Linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search, expected number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random probes	1.1	2.0	5.0	10.0	100.	—
Open, Linear probes	1.12	2.5	13.	50.	5000.	—

- Empirical comparisons:

Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search, average number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Open, Linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search, average number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Quadratic probes	1.13	2.2	5.2	11.9	126.	—
Open, Linear probes	1.13	2.7	15.4	59.8	430.	—

Conclusions: Comparison of Methods

- Sequential search is *O(n)*

  Sequential search is the most flexible method. The data may be stored in any order, with either contiguous or linked representation.

- Binary search is *O(log n)*

  Binary search demands more, but is faster: The keys must be in order, and the data must be in random-access representation (contiguous storage).

- Hash-table retrieval is O(λ)=*O(1).*

  Hashing requires the most structure, a peculiar ordering of the keys well suited to retrieval from the hash table, but generally useless for any other purpose.

Project: Empirical Analysis of Hashing

- Given a word dictionary “dict.txt” (大学英语四级单词表), you have to finish the following tasks:
  - Completing the translation: given a word list “test.in”, you need to translate all the words in the list and output their meanings (see “test.out”).
  - Analyzing the successful search using different hash techniques: you may need to implement different hash methods, e.g., chaining, open address (linear, quadratic), with different load factors and different hash functions. Then report the average number of probes by different methods. You can make tests using “successful.txt” or your self-generated cases.
  - Analyzing the unsuccessful search using different hashing techniques: You can make tests using “unsuccessful.txt” or your self-generated cases.
  - Comparing your results with the results in the textbook. (p414-415)

Project: Empirical Analysis of Hashing

- Write an article (学号姓名.doc) and submit it with all the related files (inputs, outputs, codes) in *.zip* or *.rar* to ftp://172.18.57.223/ with login account seit and password student001.
- Deadline: May. 31, 2017.
- 建议同学们自己生成测试数据，报告以论文的形式提交，把你的哈希实现方法和数据生成方式写清楚。可以尝试实现课本未提及的其他哈希技术。

Thank you!