Searching(3)

College of Computer Science, CQU

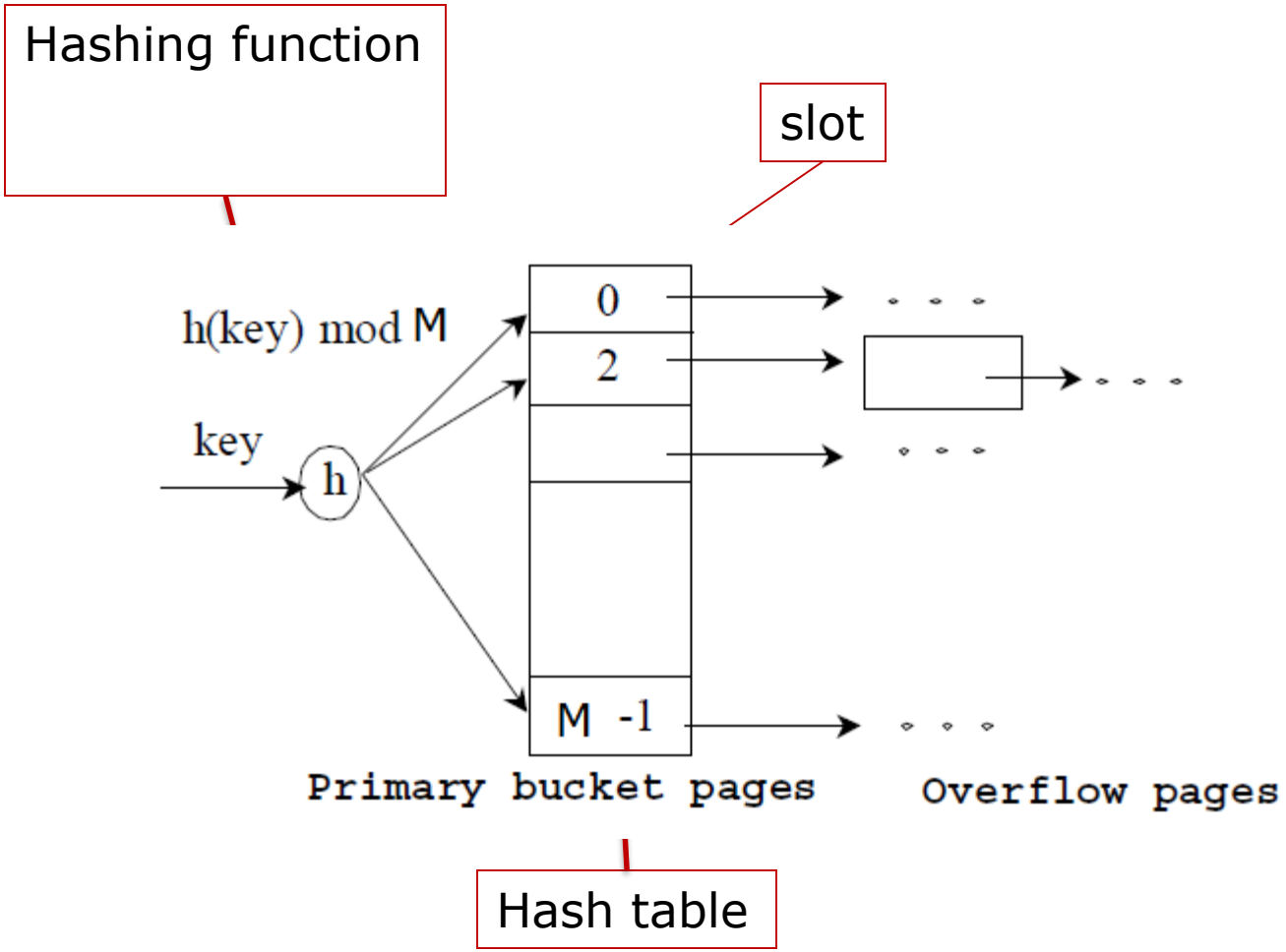
outline

- Hashing
- Hash functions
- Open hashing
- Closed hashing

Hashing

- The process of finding a record using some computation to map its key value to a position in the array is called **hashing**.
- **Hash function:** the function that maps key values to positions , denoted by ***h***
- **Hash table:** the array that holds the records , denoted by ***HT***.
- **Slot:** A position in the hash table.

Hashing


Hashing function
slot
h(key) mod M
key
h
0
2
M -1
Primary bucket pages
Overflow pages
Hash table

Basic Idea

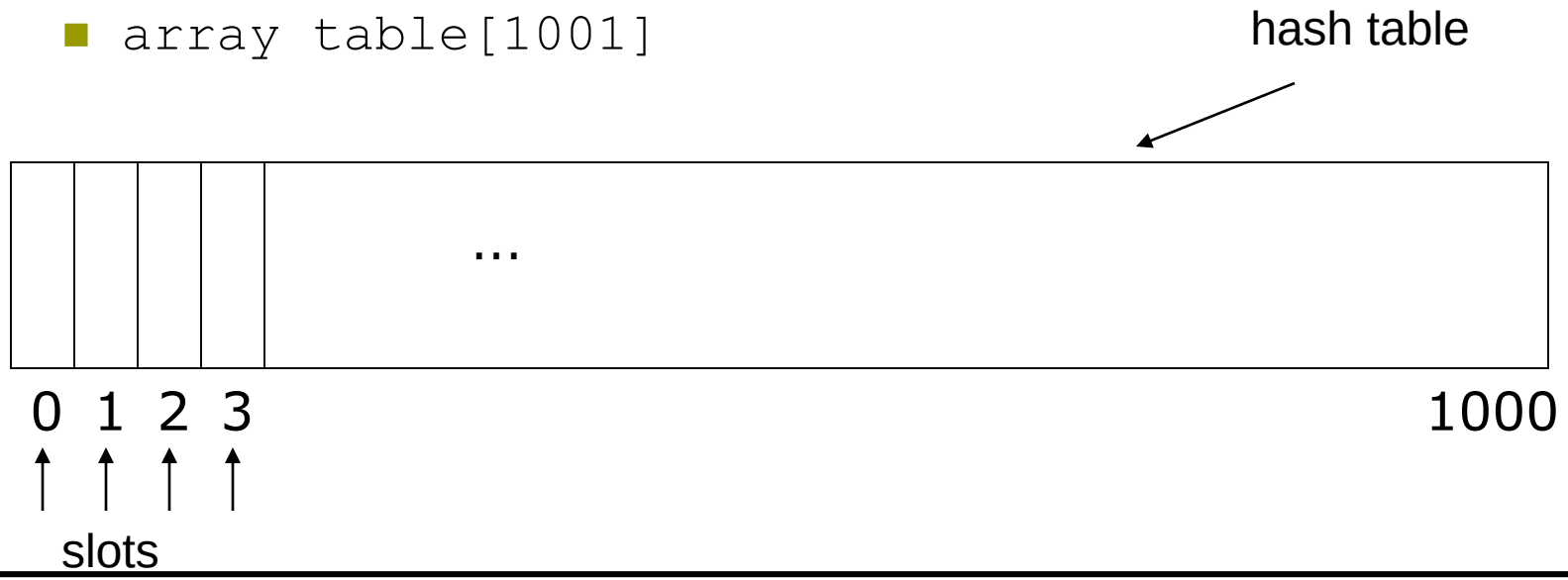
- Use *hash function* to map keys into positions in a *hash table*

Ideally

- If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table
- To search for *e*, compute *h(k)* to locate position. If no element, array does not contain *e*.

Example

- **Dictionary Student Records**
  - Keys are ID numbers (151000 - 152000), no more than 1000 students
  - Hash function: h(k) = k-151000 maps ID into distinct table positions 0-1000
  - `array table[1001]`



Analysis (Ideal Case)

- O(M) time to initialize hash table (M number of positions or slots in hash table)
- O(1) time to perform *insert, remove, search*

Ideal Case is Unrealistic

- **Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between slots and keys!**

Example:

- **Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)**
- **Expect ≈ 1,000 records at any given time**
- **Impractical to use hash table with 65,536 slots!**

Hash Functions

- **If key range too large, use hash table with fewer slots and a hash function which maps multiple keys to same slot:**

  $h(k_1) = \beta = h(k_2)$: k_1 and k_2 have collision at slot β

- **Popular hash functions: hashing by division**

  h(k) = k % M, where M number of slots in hash table

- **Example: hash table with 11 slots**

  h(k) = k%11

  80 $\rightarrow$ 3 (80%11= 3), 40 $\rightarrow$ 7, 65 $\rightarrow$ 10

  58 $\rightarrow$ 3 collision!

Hash Functions - Numerical Values

- **Consider: h(x) = x%16**
  - poor distribution, not very random
  - depends solely on least significant four bits of key
- **Better, *mid-square* method**
  - if keys are integers in range 0,1,…,K , pick integer C such that MC^2 about equal to K^2, then

$$h(x) = \lfloor x^2/C \rfloor \% M$$

  - extracts middle r bits of x^2, where 2^r=M (a base-M digit)
  - better, because most or all of bits of key contribute to result

Hash Function – Strings of Characters

- **Folding Method:**

Example 9.7 Here is a hash function for strings of characters:

```
int h(char* x) {
  int i, sum;
  for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
  return sum % M;
}
```

  - sums the ASCII values of the letters in the string
    - ASCII value for "A" =65; sum will be in range 650-900 for 10 upper-case letters; good when M around 100, for example
  - order of chars in string has no effect

Hash Function – Strings of Characters

- **Much better: Cyclic Shift**

Example 9.8 Here is a much better hash function for strings.

```
// Use folding on a string, summed 4 bytes at a time
int sfold(char* key) {
  unsigned int *lkey = (unsigned int *)key;
  int intlength = strlen(key)/4;
  unsigned int sum = 0;
  for(int i=0; i<intlength; i++)
    sum += lkey[i];

  // Now deal with the extra chars at the end
  int extra = strlen(key) - intlength*4;
  char temp[4];
  lkey = (unsigned int *)temp;
  lkey[0] = 0;
  for(int i=0; i<extra; i++)
    temp[i] = key[intlength*4+i];
  sum += lkey[0];

  return sum % M;
}
```

Collision Resolution Policies

- **Two classes:**
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- **Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)**

Open Hashing

- **Each slot in the hash table is the head of a linked list**
- **All elements that hash to a particular slot are placed on that slot's linked list**
- **Records within a slot can be ordered in several ways**
  - by order of insertion, by key value order, or by frequency of access order

Open Hashing Data Organization

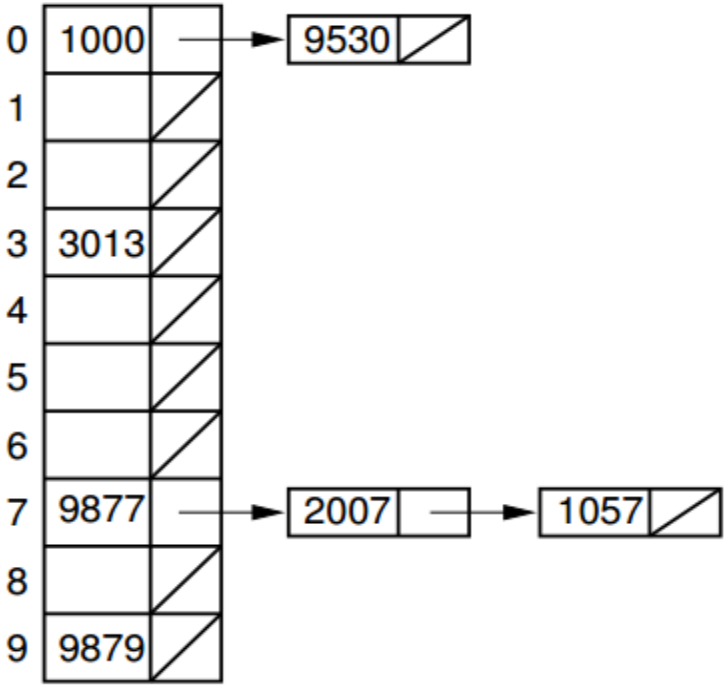
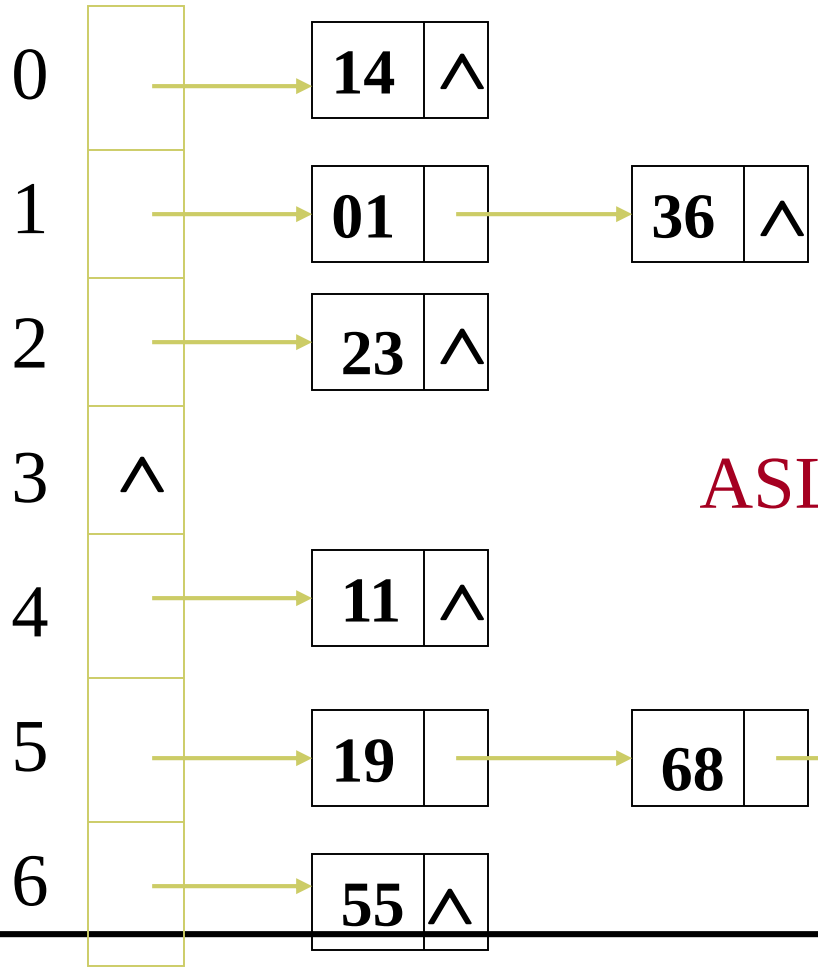


Figure 9.3 An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to slot 0, one value hashes to slot 2, three of the values hash to slot 7, and one value hashes to slot 9.

开散列法

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }



例如:同前例的关键字，哈希函数为

H(key)=key MOD 7

ASL=(6×1+2×2+3)/9=13/9

Analysis

- Open hashing is the most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list
- We hope that number of elements per slot roughly equal in size, so that the lists will be short
- If there are *N* elements in set, then each slot will have roughly *N/M*
- If we can estimate *n* and choose *M* to be roughly as large, then the average slot will have only one or two members

Analysis Cont'd

Average time per dictionary operation:

- ***M*** **slots,** ***N*** **elements in dictionary** $\Rightarrow$ **average** ***N/M*** **elements per slot**
- ***insert, search, remove*** **operation take O(1+N/M) time each**
- **If we can choose M to be about N, constant time**
- **Assuming each element is likely to be hashed to any slot, running time constant, independent of N**

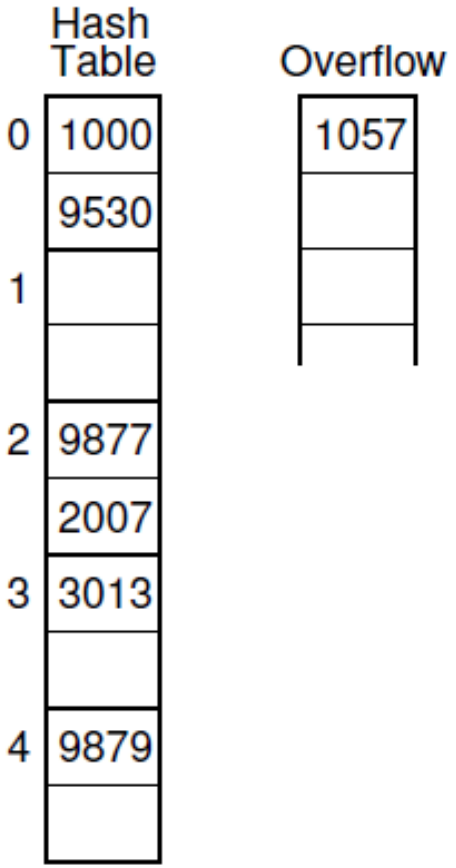
Closed hashing

- Closed hashing stores all records directly in the hash table.
- Each record R with key value k_R has a home position that is $h(k_R)$, the slot computed by the hash function.
- If R is to be inserted and another record already occupies R's home position, then R will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be.
- Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

Bucket hashing

- The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots.
- The hash function assigns each record to the first slot within one of the buckets.
- If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an *overflow bucket* of infinite capacity at the end of the table.
- ALL buckets share the same overflow bucket.
- A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

Bucket hashing: example



- seven numbers are stored in a five bucket hash table
- hash function h(K) = K mod 5.
- Each bucket contains two slots.
- three values hash to bucket 2.
- Because bucket 2 cannot hold three values, the third one ends up in the overflow bucket.

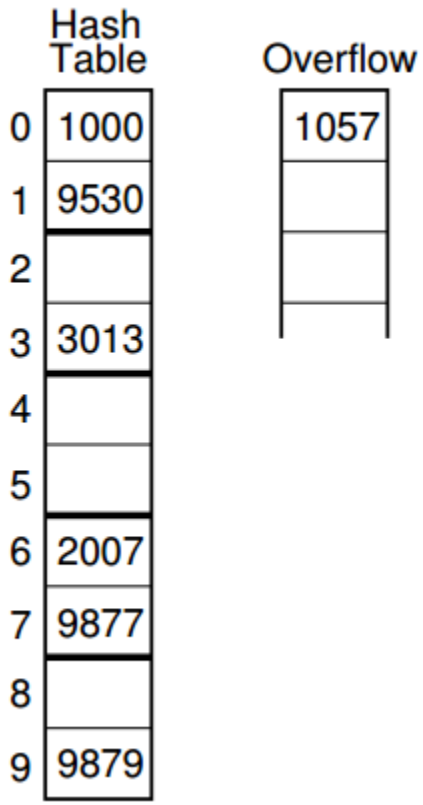
Search using bucket hashing

1. to hash the key to determine which bucket should contain the record. The records in this bucket are then searched.
2. If the desired key value is not found and the bucket still has free slots, then the search is complete.
3. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

A simple variation on bucket hashing

- Hash a key value to some slot in the hash table as though bucketing were not being used.
- If the home position is full, then the collision resolution process is to move down through the table toward the end of the bucket while searching for a free slot in which to store the record.
- If the bottom of the bucket is reached, then the collision resolution routine wraps around to the top of the bucket to continue the search for an open slot.
- If all slots in this bucket are full, then the record is assigned to the overflow bucket.
- advantage : initial collisions are reduced
  - Because any slot can be a home position rather than just the first slot in the bucket.

An variant of Bucket hashing



- hash table using the hash function **h**(*K*) = *K* mod 10. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057.

Bucket hashing: summary

- Bucket methods are good for implementing hash tables stored on disk, because the bucket size can be set to the size of a disk block.
- Whenever search or insertion occurs, the entire bucket is read into memory. Because the entire bucket is then in memory, processing an insert or search operation requires only one disk access, unless the bucket is full.
- If the bucket is full, then the overflow bucket must be retrieved from disk as well.
- Naturally, overflow should be kept small to minimize unnecessary disk accesses.

Linear probing

- In general, linear probing is to generate a sequence of hash table slots (**probe sequence**) that can hold the record using some **probe function** ; test each slot until find empty one (**probing**).

```
// Insert e into hash table HT
template <typename Key, typename E>
void hashdict<Key, E>::
hashInsert(const Key& k, const E& e) {
  int home;                          // Home position for e
  int pos = home = h(k);             // Init probe sequence
  for (int i=1; EMPTYKEY != (HT[pos]).key(); i++) {
    pos = (home + p(k, i)) % M; // probe
    Assert(k != (HT[pos]).key(), "Duplicates not allowed");
  }
  KVpair<Key,E> temp(k, e);
  HT[pos] = temp;
}
```

Figure 9.6 Insertion method for a dictionary implemented by a hash table.

- The simplest probe function : **p(k, i) = i**

HashSearch Method

- **Searching in a hash table follows the same probe sequence that was followed when inserting records. In this way, a record not in its home position can be recovered.**

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
  int home;                  // Home position for k
  int pos = home = h(k); // Initial position is home slot
  for (int i = 1; (k != (HT[pos]).key()) &&
                  (EMPTYKEY != (HT[pos]).key()); i++)
    pos = (home + p(k, i)) % M; // Next on probe sequence
  if (k == (HT[pos]).key())      // Found it
    return (HT[pos]).value();
  else return NULL;              // k not in hash table
}
```

Figure 9.7 Search method for a dictionary implemented by a hash table.

Linear probing: Example

- **D=8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3**

- Where do we insert *d*? 3 already filled
- Probe sequence using linear hashing:

  $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$

  $h_2(d) = (h(d)+2)\%8 = 5\%8 = \mathbf{5}^*$

  $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$

  etc.

  7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Linear probing: Example

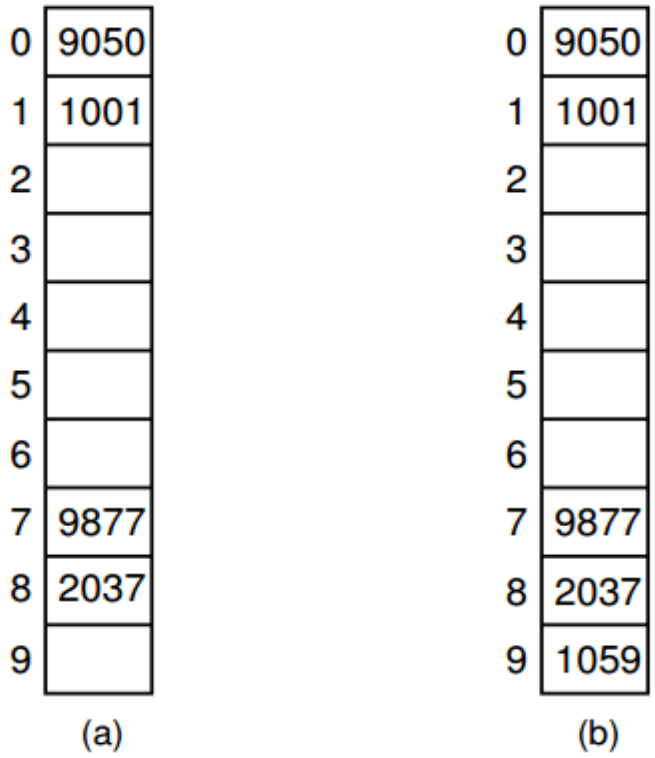


Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $\mathbf{h}(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

Problem of linear probing

- **primary clustering:**
  - by linear probing, the items has tendency to be clustered together.
- **Small clusters tend to merge into big clusters, making the problem worse.**
- **primary clustering leads to long probe sequences.**

Improved Collision Resolution

- **Linear probing: $h_i(x) = (h(x) + i) \% M$**
  - all slots in table will be candidates for inserting a new record before the probe sequence returns to home position
  - clustering of records, leads to long probing sequences
- **Linear probing with skipping: $h_i(x) = (h(x) + ci) \% M$**
  - c constant other than 1
  - records with adjacent home slot will not follow same probe sequence
- **(Pseudo)Random probing:**
  - the ith slot in the probe sequence is $(h(K) + r_i) \bmod M$ where r_i is the ith value in a random permutation of the numbers from 1 to M - 1.
  - All insertion and search operations use the same random permutation. The probe function is **p(K,i)=Perm[i-1].** where Perm is an array of length M-1 containing a random permutation of the values from 1 to M-1.

Improved Collision Resolution

- **Quadratic probing:**
  - **$P(K,i)=c_1i^2+c_2i+c_3$**
  - The simplest variation is **p(K, i) = i^2(i.e**., c1 = 1, c2 = 0, and c3 = 0.) Then the ith value in the probe sequence would be (h(K) + i^2) mod M.
  - Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on the probe sequence.
- **Double hashing:**
  - **p(K, i) = i *h_2(K).**
  - A good implementation of double hashing should ensure that all of the probe sequence constants are relatively prime to the table size M.
  - One way is to select M to be a prime number, and have h_2 return a value in the range $1 \leq h_2(K) \leq M-1$
  - Another way is to set $M = 2^m$ for some value m and have h_2 return an odd value between 1 and 2^m.

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key **MOD** 11 (表长=11)

若采用**线性探测再散列**处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	**01**	**23**	**14**	**68**	**11**	**82**	**36**	**19**		
1	1	2	1	3	6	2	5	1		

若采用**二次探测再散列**处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	**01**	**23**	**14**	**11**	**82**	**68**	**36**	**19**		
1	1	2	1	3	1	3	3	1		

练习1:已知一组关键字为（26，36，41，38，44，15，68，12，06，51，25），用线性探查法解决冲突构造这组关键字的哈希表。表长取15，哈希函数H(key)=key MOD 13。并求出等概率情况下查找成功的平均查找长度ASL.

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51

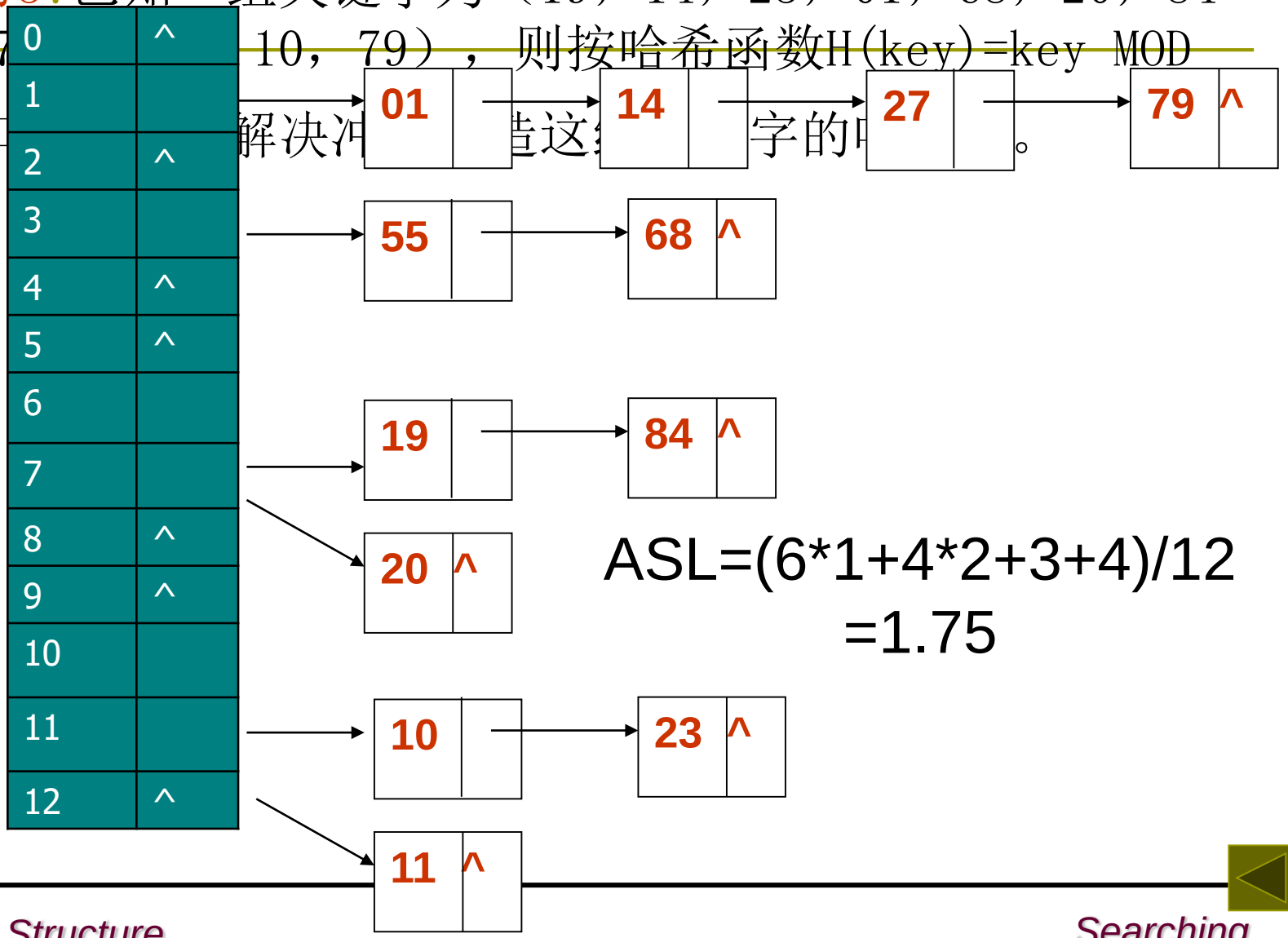
ASL=(1+1+1+1+1+2+2+2+1+3+5)/11=20/11

练习2:已知一组关键字为（26，36，41，38，44，15，68，12，06，51，25），用**二次探查法**解决冲突构造这组关键字的哈希表。表长取15，哈希函数H(key)=key MOD 13。并求出等概率情况下查找成功的平均查找长度ASL.

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	51	41	15	68	44	06	25			36		38	12	

ASL=(1+1+1+1+1+2+2+2+1+3+6)/11=21/11

练习3:已知一组关键字为（19，14，23，01，68，20，84，27，10，79），则按哈希函数H(key)=key MOD 13和解决冲突造这组字的哈希表。



ASL=(6*1+4*2+3+4)/12
=1.75

A partial implementation for the dictionary ADT using a hash table.

```
// Dictionary implemented with a hash table
template <typename Key, typename E>
class hashdict : public Dictionary<Key,E> {
private:
  KVpair<Key,E>* HT;    // The hash table
  int M;          // Size of HT
  int currcnt;    // The current number of elements in HT
  Key EMPTYKEY; // User-supplied key value for an empty slot

  int p(Key K, int i) const // Probe using linear probing
    { return i; }

  int h(int x) const { return x % M; } // Poor hash function
  int h(char* x) const { // Hash function for character keys
    int i, sum;
    for (sum=0, i=0; x[i] != '\0'; i++) sum += (int) x[i];
    return sum % M;
  }

  void hashInsert(const Key&, const E&);
  E hashSearch(const Key&) const;
```

A partial implementation for the dictionary ADT using a hash table.

```
public:
  hashdict(int sz, Key k){ // "k" defines an empty slot
    M = sz;
    EMPTYKEY = k;
    currcnt = 0;
    HT = new KVpair<Key,E>[sz]; // Make HT of size sz
    for (int i=0; i<M; i++)
      (HT[i]).setKey(EMPTYKEY); // Initialize HT
  }

  ~hashdict() { delete HT; }
```

A partial implementation for the dictionary ADT using a hash table.

```
// Find some record with key value "K"
E find(const Key& k) const
  { return hashSearch(k); }
int size() { return currcnt; } // Number stored in table

// Insert element "it" with Key "k" into the dictionary.
void insert(const Key& k, const E& it) {
  Assert(currcnt < M, "Hash table is full");
  hashInsert(k, it);
  currcnt++;
}
```

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.324-339

Thank you for listening!