《数据结构与算法》课程组

重庆大学计算机学院

Data Structures & Algorithms

9

SEARCHING ALGORITHMS

Outline

9.1 Definitions

Searching

- Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set.
- The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

Searching: formal definition

- Suppose that we have a collection L of n records of the form

  $(k_1; I_1); (k_2; I_2); :::; (k_n; I_n)$

  where I_j is information associated with key k_j from record j for 1≤ j ≤ n. Given a particular key value K, the search problem is to locate a record (k_j ; I_j) in L such that $k_j = K$ (if one exists). Searching is a systematic method for locating the record (or records) with key value $k_j = K$.

Terminologies

- A **successful search** is one in which a record with key $k_j = K$ is found.
- An **unsuccessful search** is one in which no record with $k_j = K$ is found (and no such record exists).
- An **exact-match query** is a search for the record whose key value matches a specified key value.
- A **range query** is a search for all records whose key value falls within a specified range of key values.

Searching Algorithms

- We can categorize search algorithms into three general approaches:
  - 1. Sequential and list methods.
  - 2. Direct access by key value (hashing).
  - 3. Tree indexing methods.

9.2 Sequential Search

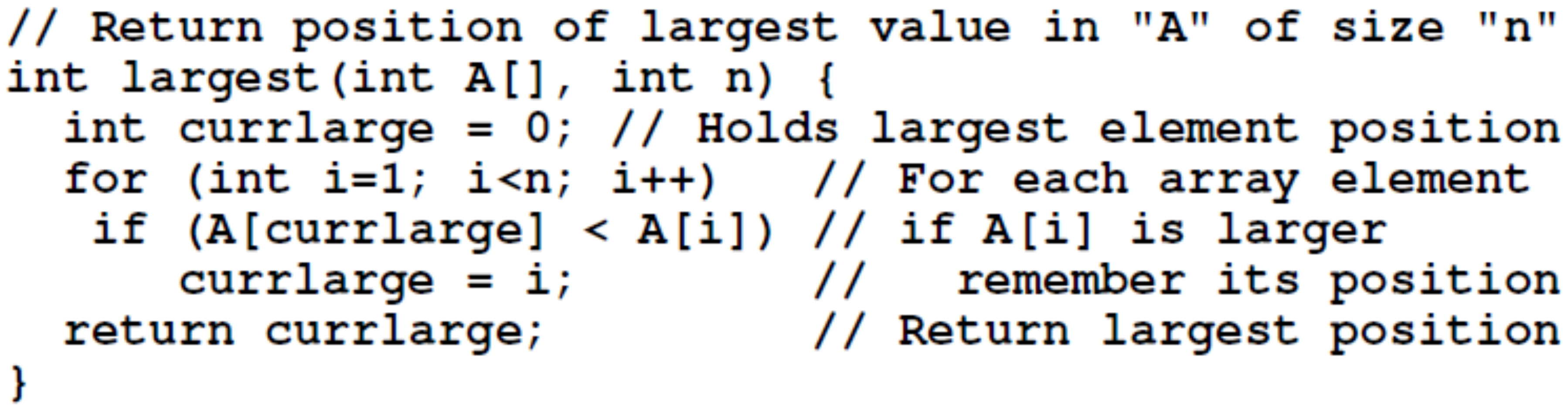
Searching on Unsorted Arrays

- **Sequential search algorithm**
  - **the simplest form of search**
  - **Sequential search on an unsorted list requires $O(n)$ time in the worst case.**
- **Basic algorithm:**

```
Get the search criterion (key)
Get the first record from the file
While ( (record != key) and (still more records) )
    Get the next record
End_while
```

- **When do we know that there wasn't a record in the file that matched the key?**

Sequential Search on unsorted arrays: implementation



```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)   // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

Comparison: n-1 times

Sequential Search on unsorted arrays: implementation

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)   // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

面试题：查找数组A[1..n]中的最大值和最小值，比较次数不超过1.5n。

Sequential Search on unsorted arrays: implementation

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)    // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;         //   remember its position
  return currlarge;          // Return largest position
}
```

面试题：查找数组A[1..n]中的最大值和最小值，比较次数不超过1.5n。

- 依次查找最大值和最小值，需要的比较次数: 2(n-1)
- 考虑同时找最大值和最小值
- 为更新当前的最大值和最小值，怎样与A元素比较？与多少个元素一起比较效率高？

Sequential Search on unsorted arrays: implementation

面试题：查找数组A[1..n]中的最大值和最小值，比较次数不超过1.5n。

- 每次与A中一个元素比较

```
void maxMinValue(E A[], int n, int& maxp, int& minp) {
        maxp = minp = 0;  //最大值和最小值的初始位置

        for(int k = 1;  k < n ;  k ++) {

              if(A[maxp] < A[k])
                    maxp = k;          //更新最大值位置
              else if(A[minp] > A[k])
                    minp = k;          //更新最小值位置
        }
}
```

比较次数（最坏情况）：2(n-1)

Sequential Search on unsorted arrays: implementation

面试题：查找数组A[1..n]中的最大值和最小值，比较次数不超过1.5n。

- 每次与A中两个元素比较

比较次数：$\frac{3}{2}n$

```
void maxMinValue(E A[], int n, int& maxp, int& minp) {
        maxp = minp = 0;  //最大值和最小值的初始位置

        for(int k = n&1;  k < n ;  k += 2) {

              if(A[k] < A[k+1]) {
                    if(A[maxp] < A[k+1])
                          maxp = k+1;     //更新最大值位置
                    if(A[minp] > A[k])
                          minp = k;           //更新最小值位置
              }else{  //A[k] > A[k+1]

                    if(A[maxp] < A[k])
                          maxp = k;          //更新最大值位置
                    if(A[minp] > A[k+1])
                          minp = k+1;       //更新最小值位置
              }
        }
}
```

Sequential Search on unsorted arrays: implementation

问题：查找数组A[1..n]中的最大值和最小值，比较次数不超过1.5n。

- 每次与A中三个元素比较

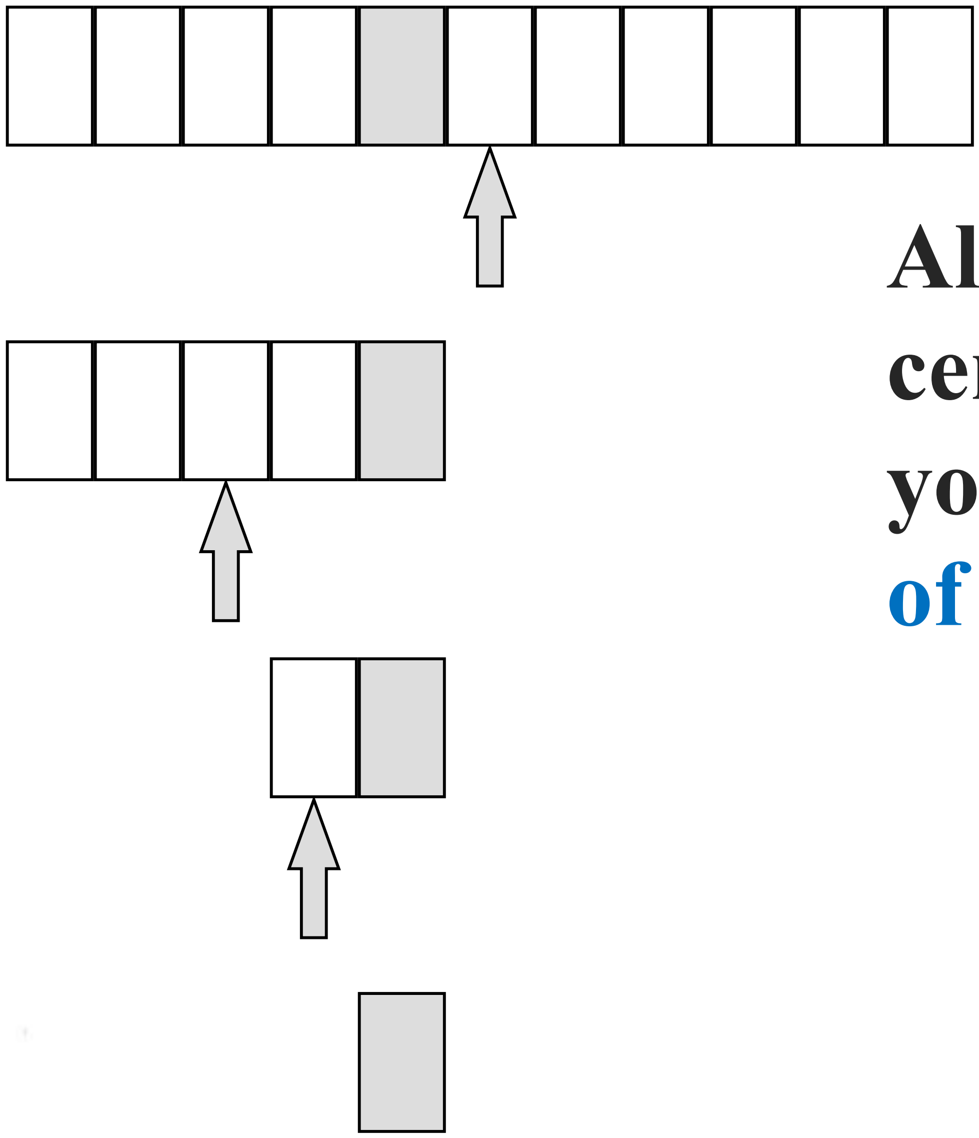
（分析）从三个元素中找最大值与最小值需要3次比较（最坏情况！），更新A当前的最大值与最小值需要比较2次。因此，总共的比较次数：

$$\frac{n}{3}(3+2)=\frac{5}{3}n。$$

Searching on sorted Arrays

- If we have an ordered list and we know how many things are in the list (i.e., number of records in a file), we can use a different strategy.
- The binary search gets its name because the algorithm continually divides the list into two parts.

How a Binary Search Works



Always look at the center value. Each time you get to **discard half of the remaining list**.

Is this fast ?

Binary search: Implementation

```
// Return the position of an element in sorted array "A" of
// size "n" with value "K".  If "K" is not in "A", return
// the value "n".
int binary(int A[], int n, int K) {
  int l = -1;
  int r = n;            // l and r are beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Check middle of remaining subarray
    if (K < A[i]) r = i;     // In left half
    if (K == A[i]) return i; // Found it
    if (K > A[i]) l = i;     // In right half
  }
  return n; // Search value not in A
}
```

How Fast is a Binary Search?

- Worst case: 11 items in the list took 4 tries
- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item

A Very Fast Algorithm!

- **How long (worst case) will it take to find an item in a list 30,000 items long?**

$2^{10} = 1024$	$2^{13} = 8192$
$2^{11} = 2048$	$2^{14} = 16384$
$2^{12} = 4096$	$2^{15} = 32768$

- **So, it will take only 15 tries!**
- **Time complexity:** $\Theta(\log n)$

Binary search: Application

Range query: Given an array A[1…n] of size n that satisfies

$$A[1] \le A[2] \le \cdots \le A[i] \le \cdots \le A[n],$$

find all elements in range [a, b)

思路：查找A中$\ge a$的最小值位置和$< b$的最大值位置

```
int binary(E A[], int n, int K)  {
      int l = 0;           //指向< K的元素
      int r = n+1;      //指向≥ K的元素
      while (l+1 < r) {
           int m = (l + r) / 2;
           if (A[m] < K)
                   l = m;
           else
                   r = m;
      }
      return r;   //返回≥ K的最小值位置
}
```

```
int binary(E A[], int n, int K)  {
    int l = 0;
    int r = n+1;
    while(l+1 < r)  {
         int m = (l+r) / 2;
         if(A[m] < K)
             l = m;
         else
             r = m;
     }
     return l;  //返回<K的最大值位置
}
```

Binary search: Application

不似“二分”，恰是二分

快速求幂 a^n （$a, n > 0$）

- 直接迭代：$a^n = a^{n-1} * a$　　　时间复杂度（相乘次数）$\Theta(n)$
- 二分递归：$\boxed{a^n = a^{n\%2} * a^{\frac{n}{2}} * a^{\frac{n}{2}}}$

时间 $\Theta(\log(n))$

9.3 Dictionary

Dictionary

- **Dictionary is a collection of data records, an** efficient ways to organize collections of data records so that they can be stored and retrieved quickly.
- Key: a number which can be used for searching records
- comparable

ADT for a simple dictionary

```
// The Dictionary abstract class.
template <typename Key, typename E>
class Dictionary {
private:
        void operator =(const Dictionary&) {}
        Dictionary(const Dictionary&) {}
public:
        Dictionary() {} // Default constructor
        virtual ˜Dictionary() {} // Base destructor
          // Reinitialize dictionary
        virtual void clear() = 0;
```

The ADT for a simple dictionary

```
// Insert a record
// k: The key for the record being inserted.
// e: The record being inserted.
virtual void insert(const Key& k, const E& e) = 0;
// Remove and return a record.
// k: The key of the record to be removed.
// Return: A maching record. If multiple records match
// "k", remove an arbitrary one. Return NULL if no record
// with key "k" exists.
virtual E remove(const Key& k) = 0;
// Remove and return an arbitrary record from dictionary.
// Return: The record removed, or NULL if none exists.
virtual E removeAny() = 0;
```

The ADT for a simple dictionary

```
  // Return: A record matching "k" (NULL if none
exists).
 // If multiple records match, return an arbitrary one.
 // k: The key of the record to find
 virtual E find(const Key& k) const = 0;
// Return the number of records in the dictionary.
 virtual int size() = 0;
};
```

Example: A payroll record Implementation

```
// A simple payroll entry with ID, name, address fields
class Payroll {
private:
   int ID;
   string name;
   string address;
public:
// Constructor
   Payroll(int inID, string inname, string inaddr) {
        ID = inID;
        name = inname;
        address = inaddr;
    }
    ~Payroll() {} // Destructor
  // Local data member access functions
     int getID() { return ID; }
     string getname() { return name; }
     string getaddr() { return address; }
};
```

A dictionary search example of payroll record

```
int main() {
    UALdict<int, Payroll*> IDdict; // IDdict organizes Payroll records by ID
    UALdict<string, Payroll*> namedict; // namedict organizes Payroll records by name
    Payroll *foo1, *foo2, *findfoo1, *findfoo2;
    foo1 = new Payroll(5, "Joe", "Anytown");
    foo2 = new Payroll(10, "John", "Mytown");
    IDdict.insert(foo1->getID(), foo1);
    IDdict.insert(foo2->getID(), foo2);
    namedict.insert(foo1->getname(), foo1);
    namedict.insert(foo2->getname(), foo2);
    findfoo1 = IDdict.find(5);
    if (findfoo1 != NULL)
            cout << findfoo1;
    else
            cout << "NULL ";
    findfoo2 = namedict.find("John");
    if (findfoo2 != NULL)
            cout << findfoo2;
    else
            cout << "NULL ";
}
```

Here, payroll records are stored in two dictionaries, one organized by ID and the other organized by name. Both dictionaries are implemented with an unsorted array-based list.

Mechanism for extracting keys

- One approach: to require all record types to support some particular method that returns the key value.
- Problem: this approach does not work when the same record type is meant to be stored in multiple dictionaries, each keyed by a different field of the record.

Mechanism for extracting keys

- Another approach: to supply a class whose job is to extract the key from the record.
- Problem: this solution also does not work in all situations, because there are record types for which it is not possible to write a key extraction method.
- The fundamental issue is that the key value for a record is not an intrinsic property of the record's class, or of any field within the class. The key for a record is actually a property of the context in which the record is used.

Mechanism for extracting keys

- Solution: key-value pairs
- to explicitly store the key associated with a given
- record, as a separate field in the dictionary.
- That is, each entry in the dictionary will contain both a record and its associated key. Such entries are known as key-value pairs.

Implementation for a class representing a key-value pair

```
template <typename Key, typename E>
class KVpair {                          // Container for a key-value pair
private:
   Key k;
   E e;
public:
  KVpair() {}                           // Default Constructors
  KVpair(Key kval, E eval)             // Constructors
  { k = kval; e = eval; }
  KVpair(const KVpair& o)              // Copy constructor
  { k = o.k; e = o.e; }
  void operator =(const KVpair& o) // Assignment operator
  { k = o.k; e = o.e; }
                                  // Data member access functions
  Key key() { return k; }
  void setKey(Key ink) { k = ink; }
  E value() { return e; }
};
```

Ways to implement dictionary :implemented with an unsorted array-based list

```
// Dictionary implemented with an unsorted array-based list
template <typename Key, typename E>
class UALdict : public Dictionary<Key, E> {
private:
    AList<KVpair<Key,E> >* list;
public:
   UALdict(int size=defaultSize) // Constructor
   { list = new AList<KVpair<Key,E> >(size); }
   ˜UALdict() { delete list; } // Destructor

    void clear() {  list->clear();  } // Reinitialize
      // Insert an element: append to list
    void insert(const Key&k, const E& e) {
         KVpair<Key,E> temp(k, e);
         list->append(temp);
    }
```

A dictionary implemented with an unsorted array-based list

```
// Use sequential search to find the element to remove
E remove(const Key& k) {
        E temp = find(k); // "find" will set list position
        if(temp != NULL) list->remove();
        return temp;
}
E removeAny() { // Remove the last element
        Assert(size() != 0, "Dictionary is empty");
        list->moveToEnd();
        list->prev();
        KVpair<Key,E> e = list->remove();
        return e.value();
}
```

A dictionary implemented with an unsorted array-based list

```
// Find "k" using sequential search
    E find(const Key& k) const {
            for(list->moveToStart();
                    list->currPos() < list->length(); list->next()) {
                            KVpair<Key,E> temp = list->getValue();
                            if (k == temp.key())
                                    return temp.value();
            }
            return NULL; // "k" does not appear in dictionary
    }
    int size()  {   // Return list size
            return list->length();
    }
};
```

Ways to implement dictionary : An implementation for a sorted array-based list

```
// Sorted array-based list
// Inherit from AList as a protected base class
template <typename Key, typename E>
class SAList: protected AList<KVpair<Key,E> > {
public:
   SAList(int size=defaultSize) : AList<KVpair<Key,E> >(size) {     }
   ˜SAList() {} // Destructor
// Redefine insert function to keep values sorted
    void insert(KVpair<Key,E>& it) { // Insert at right
            KVpair<Key,E> curr;
            for (moveToStart(); currPos() < length(); next()) {
                    curr = getValue();
                   if(curr.key() > it.key())  break;
            }
            AList<KVpair<Key,E> >::insert(it); // Do AList insert
}
```

An implementation for a sorted array-based list: Inherit from AList

```
// With the exception of append, all remaining methods are
// exposed from AList.  Append is not available to SAList
// class users since it has not been explicitly exposed.
    AList<KVpair<Key,E> >::clear;
    AList<KVpair<Key,E> >::remove;
    AList<KVpair<Key,E> >::moveToStart;
    AList<KVpair<Key,E> >::moveToEnd;
    AList<KVpair<Key,E> >::prev;
    AList<KVpair<Key,E> >::next;
    AList<KVpair<Key,E> >::length;
    AList<KVpair<Key,E> >::currPos;
    AList<KVpair<Key,E> >::moveToPos;
    AList<KVpair<Key,E> >::getValue;
};
```

An implementation for a sorted array-based list

```
// Dictionary implemented with a sorted array-based list
template <typename Key, typename E>
class SALdict : public Dictionary<Key, E> {
private:
     SAList<Key,E>* list;
public:
    SALdict(int size=defaultSize) // Constructor
   { list = new SAList<Key,E>(size); }
   ~SALdict() { delete list; } // Destructor
    void clear() { list->clear(); } // Reinitialize
    // Insert an element: Keep elements sorted
    void insert(const Key&k, const E& e) {
              KVpair<Key,E> temp(k, e);
              list->insert(temp);
    }
```

An implementation for a sorted array-based list

```
// Use sequential search to find the element to remove
E remove(const Key& k) {
        E temp = find(k);
        if (temp != NULL) list->remove();
        return temp;
}
E removeAny() { // Remove the last element
        Assert(size() != 0, "Dictionary is empty");
        list->moveToEnd();
        list->prev();
        KVpair<Key,E> e = list->remove();
        return e.value();
}
```

An implementation for a sorted array-based list

```
// Find "K" using binary search
E find(const Key& k) const {
    int l = -1;
    int r = list->length();
    while (l+1 != r) { // Stop when l and r meet
        int i = (l+r)/2; // Check middle of remaining subarray
        list->moveToPos(i);
        KVpair<Key,E> temp = list->getValue();
        if (k < temp.key()) r = i; // In left
        if (k == temp.key()) return temp.value(); // Found it
        if (k > temp.key()) l = i; // In right
    }
    return NULL; // "k" does not appear in dictionary
}
```

9.4 Hashing

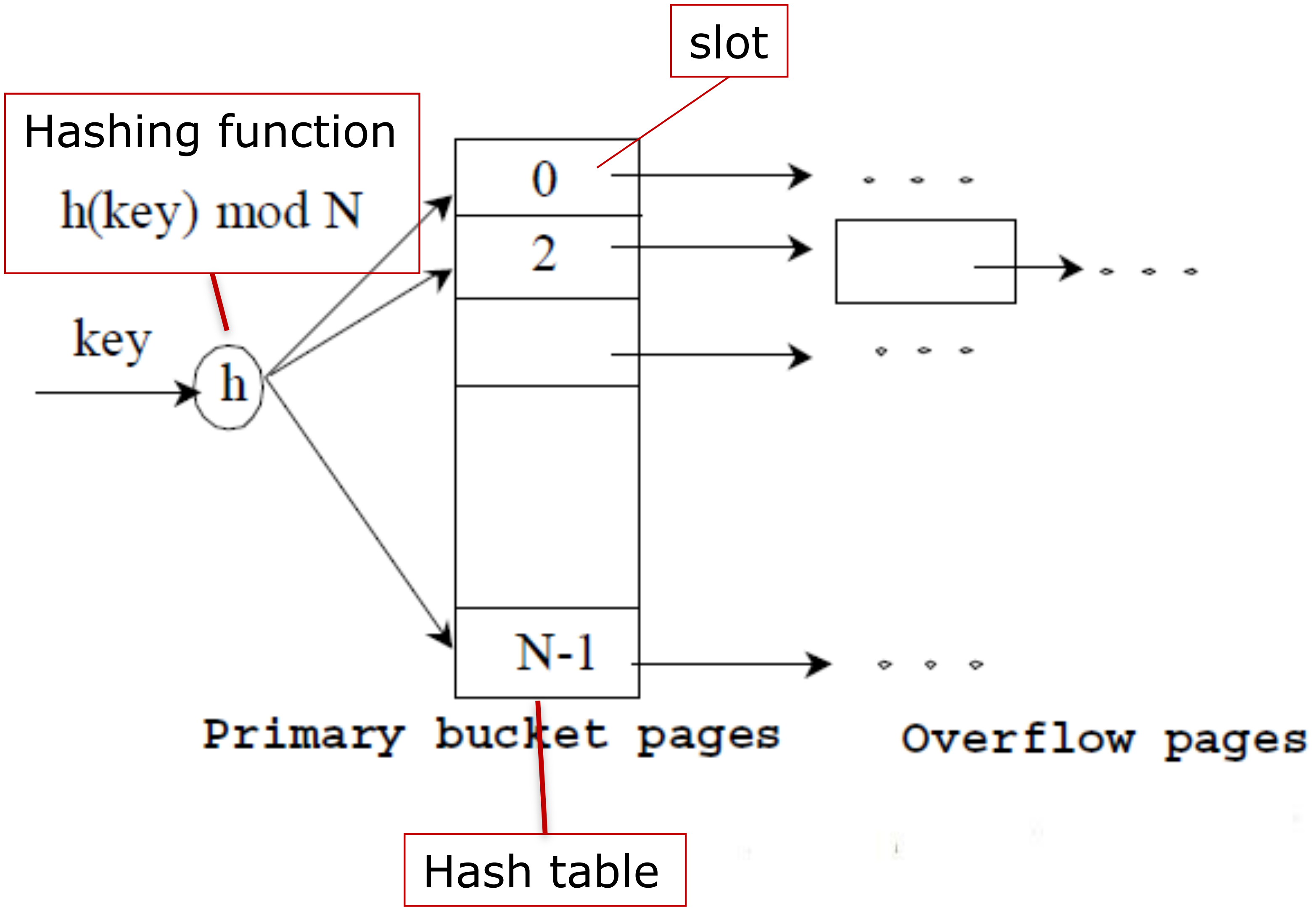
outline

- 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


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

Basic Idea

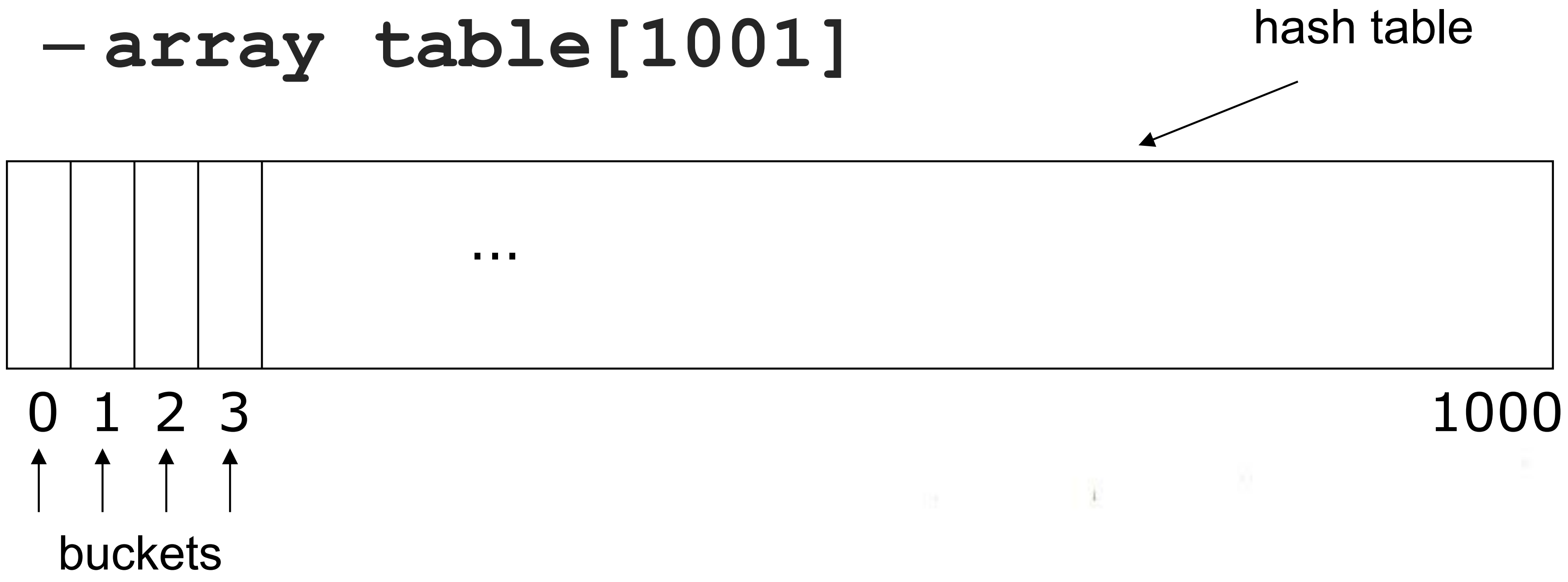
- Use ***hash function*** to map keys into positions in **a *hash table***

<u>Ideally</u>

- 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 (951000 - 952000), no more than 100 students**
  - **Hash function: h(k) = k-951000 maps ID into distinct table positions 0-1000**
  - `array table[1001]`



Analysis (Ideal Case)

- **O(b) time to initialize hash table (b number of positions or buckets 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 buckets and keys!

Example:

- Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
- Expect $\approx$ 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 buckets and a hash function which maps multiple keys to same bucket:

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

- Popular hash functions: hashing by division

  $h(k) = k \% D$, where D number of buckets in hash table

- Example: hash table with 11 buckets

  $h(k) = k \% 11$

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

  $58 \rightarrow 3$ 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 DC^2 about equal to K^2, then

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

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

Hash Function – Strings of Characters

- **Folding Method:**

```
int h(String x, int D) {
int i, sum;
for (sum=0, i=0; i<x.length(); i++)
   sum+= (int)x.charAt(i);
return (sum%D);
}
```

  – 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 D around 100, for example

  – order of chars in string has no effect

Hash Function – Strings of Characters

- **Much better: Cyclic Shift**

```
static long hashCode(String key, int D)
 {
  int h=0;
  for (int i=0, i<key.length(); i++){
   h = (h << 4) | ( h >> 27);
   h += (int) key.charAt(i);
   }
  return h%D;
}
```

Well-used Hash Functions

1. 直接寻址法: 取关键字或关键字的某个线性函数值为散列地址, 即h(key)=key或h(key) = a·key + b，其中a和b为常数（这种散列函数叫做自身函数）

2. 数字分析法: 分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址

Well-used Hash Functions

3.平方取中法: 取关键字平方后的中间几位作为散列地址

4.折叠法: 将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址

5. 随机数法: 选择一随机函数，取关键字作为随机函数的种子生成随机值作为散列地址，通常用于关键字长度不同的场合

6. 除留余数法: 取关键字被某个不大于散列表长度m的数p除后所得的余数为散列地址。即 $h(key) = key \% p$, $p<=m$。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取质数或m，若p选的不好，容易产生碰撞

Well-known Hash Algorithms

1. MD4（ RFC 1320 ）是 MIT 的 Rivest 在 1990 年设计的，其输出为 128 位。MD4 已证明不够安全
2. MD5（RFC 1321）对 MD4 的改进版本。它对输入仍以 512 位分组，其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备”强抗碰撞性”
3. SHA （Secure Hash Algorithm）是一个 Hash 函数族，由 NIST于 1993 年发布第一个算法。目前知名的 SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，因此抗穷举性更好。SHA-1 设计时基于和 MD4 相同原理，并且模仿了该算法。SHA-1 已被证明不具”强抗碰撞性”。

Well-known Hash Algorithms

- **MD5的碰撞案例**

```
import hashlib

#  两段HEX字节串，注意它们有细微差别
a =
bytearray.fromhex("0e306561559aa787d00bc6f70bbdfe3404cf03659e704f8534c00ffb659c4c8
740cc942feb2da115a3f4155cbb8607497386656d7d1f34a42059d78f5a8dd1ef")

b =
bytearray.fromhex("0e306561559aa787d00bc6f70bbdfe3404cf03659e744f8534c00ffb659c4c8
740cc942feb2da115a3f415dcbb8607497386656d7d1f34a42059d78f5a8dd1ef")

#  输出MD5，它们的结果一致
print(hashlib.md5(a).hexdigest())
print(hashlib.md5(b).hexdigest())

### a和b输出结果都为：
cee9a457e790cf20d4bdaa6d69f01e41
cee9a457e790cf20d4bdaa6d69f01e41
```

Well-known Hash Algorithms

- **SHA以及SHA1碰撞**

- SHA与MD5算法本质上是类似的，但安全性要领先很多——这种领先型更多的表现在碰撞攻击的时间开销更大，当然计算时间慢
- SHA有SHA0、SHA1、SHA256、SHA384等等，它们的计算方式和计算速度都有差别。其中**SHA1是现在用途最广泛的一种算法**。包括GitHub在内的众多版本控制工具以及各种云同步服务都是用SHA1来区别文件，很多安全证书或是签名也使用SHA1来保证唯一性。长期以来，人们都认为SHA1是十分安全的，至少大家还没有找到一次碰撞案例。
- 但这一事实在2017年2月破灭了。CWI和Google的研究人员们成功找到了一例SHA1碰撞，而且很厉害的是，发生碰撞的是两个真实的、可阅读的PDF文件。这两个PDF文件内容不相同，但SHA1值完全一样。
- 所以，对于一些大的商业机构来说， MD5 和 SHA1 已经不够安全，推荐至少使用 SHA2-256 算法。

Collision Resolution Policies

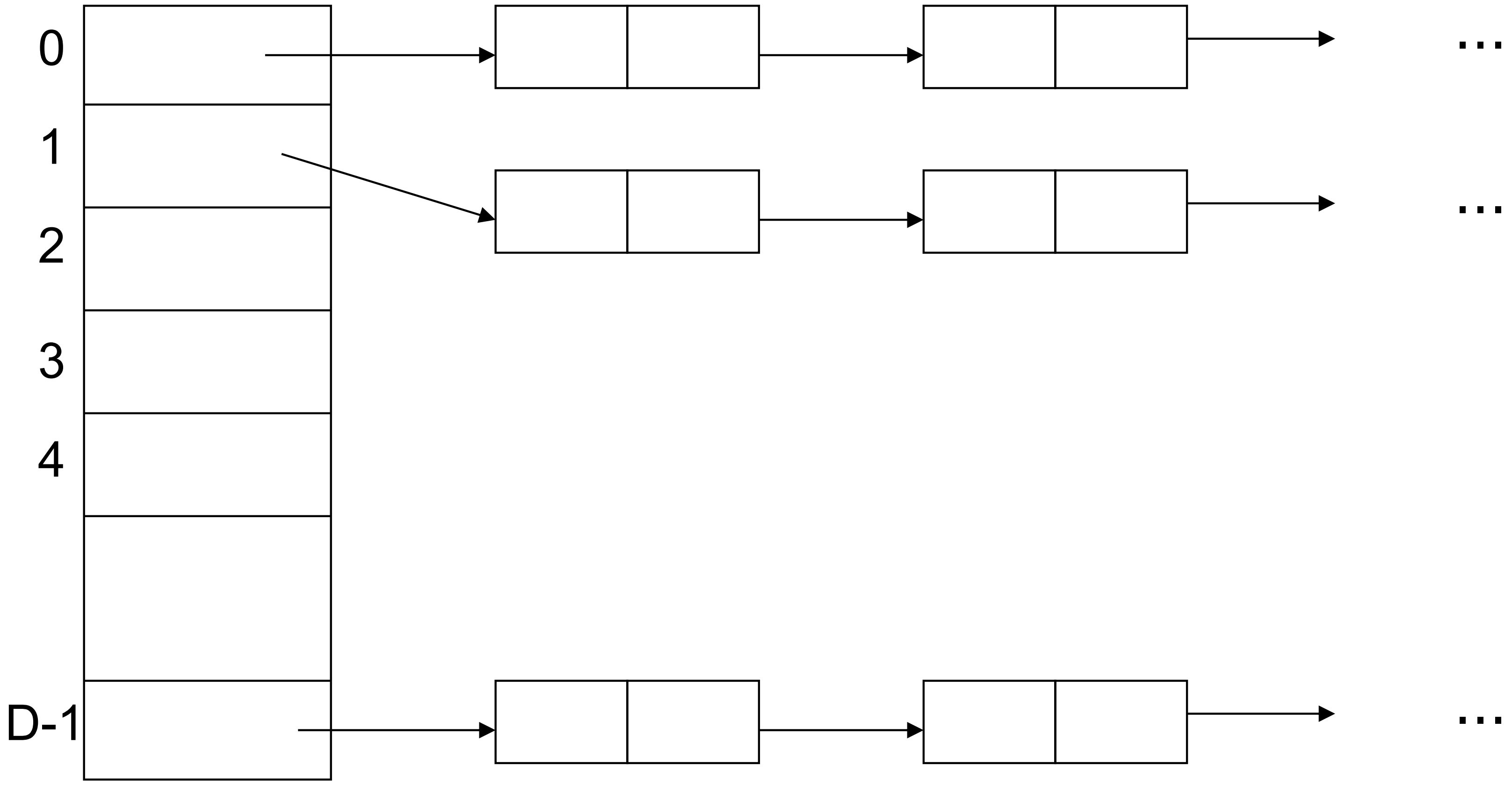
- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing

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

Open Hashing Data Organization



Analysis

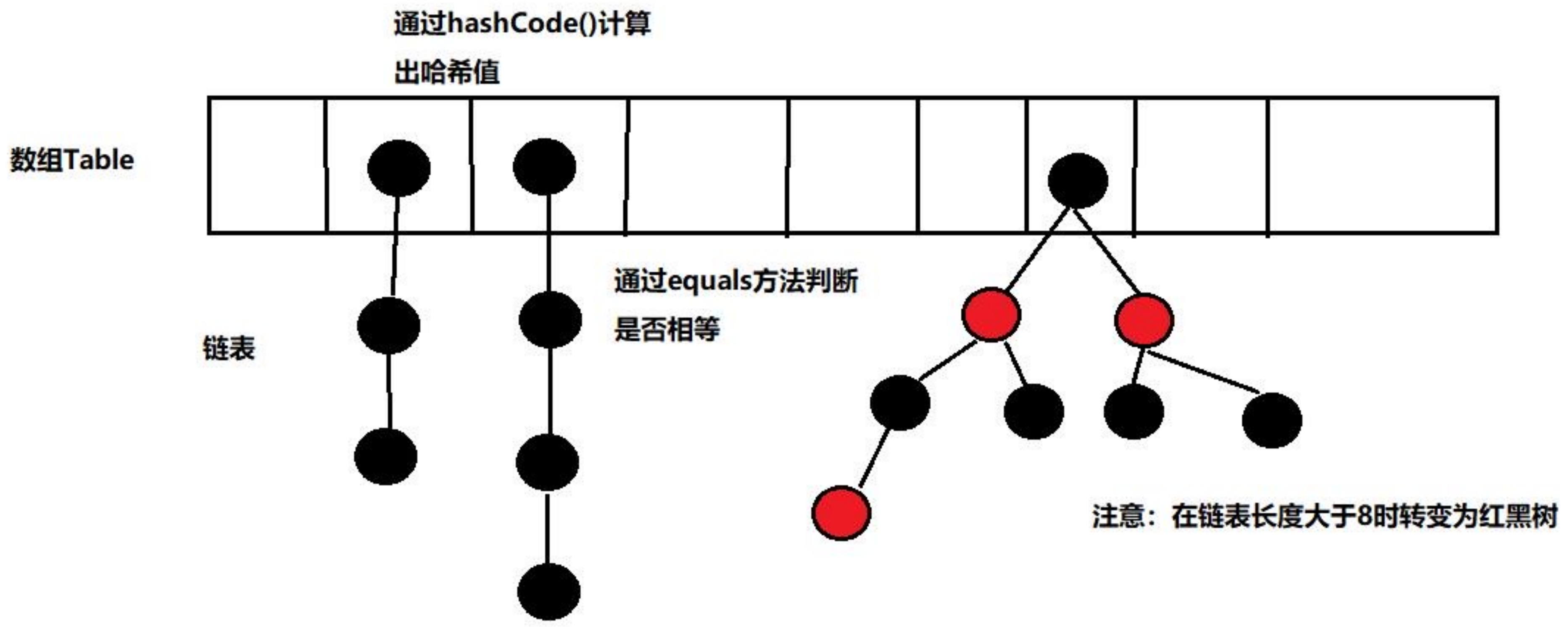
- Open hashing is the most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list
- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are *n* elements in set, then each bucket will have roughly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

Analysis Cont’d

Average time per dictionary operation:

- ***D*** **buckets,** ***n*** **elements in dictionary ⇒ average** ***n/D*** **elements per bucket**
- ***insert*****,** ***search*****,** ***remove*** **operation take O(1+n/D) time each**
- **If we can choose D to be about n, constant time**
- **Assuming each element is likely to be hashed to any bucket, running time constant, independent of n**

Open Hashing Application: Hashmap (JDK1.8)



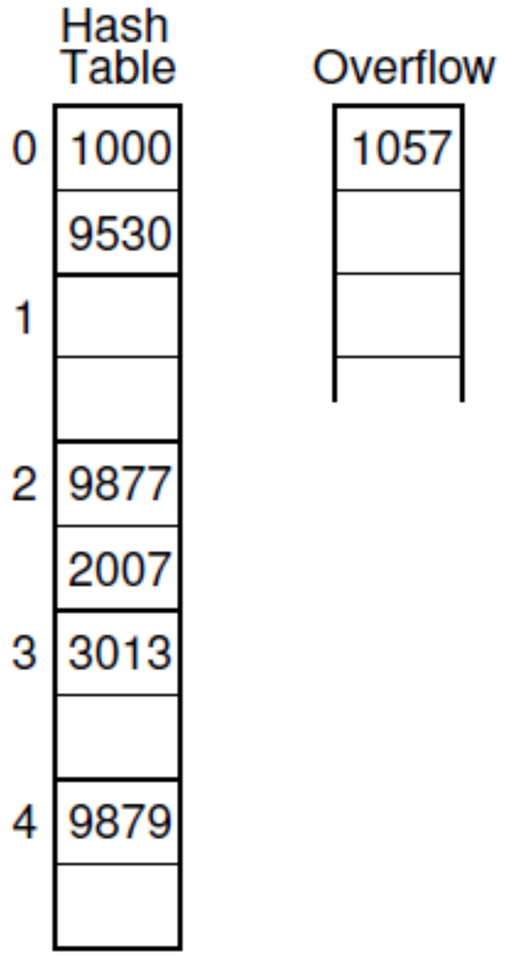
Closed hashing

- 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

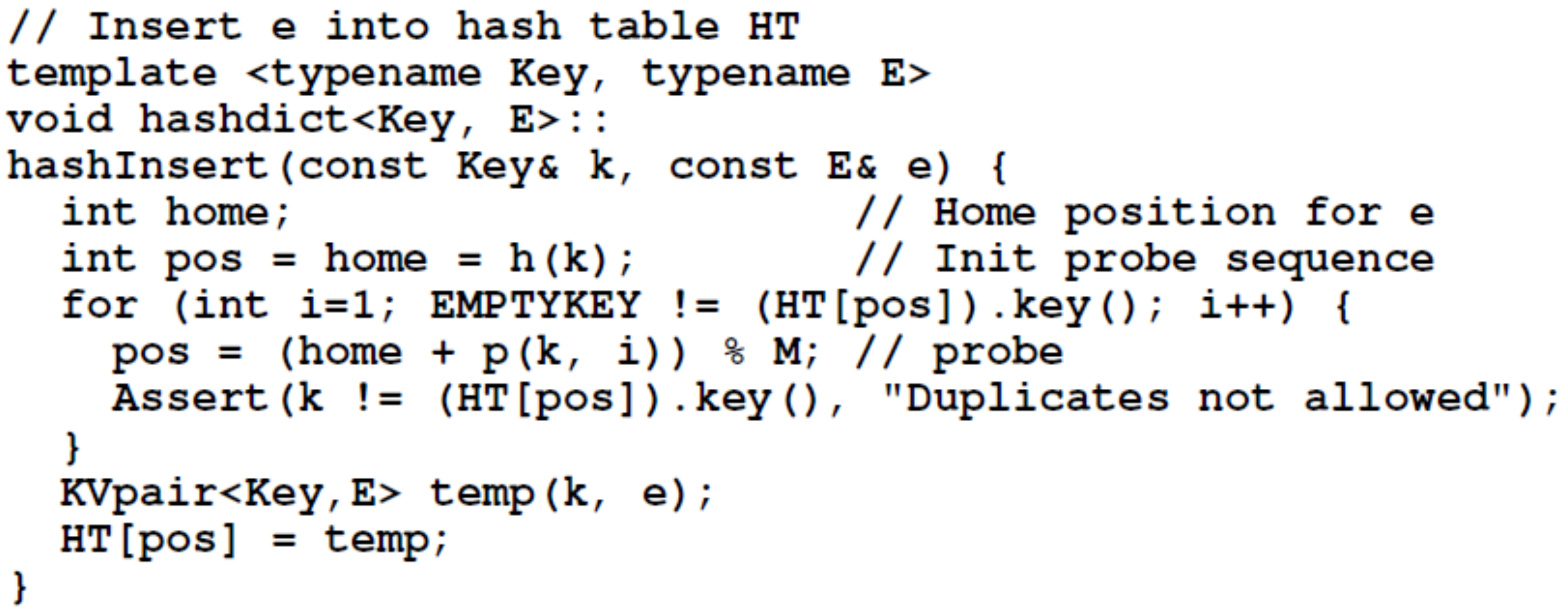
- Hash a key value to some slot in the hash table as though bucketing were not being used.
- If the home position is full, then the collision resolution process is to move down through the table toward the end of the bucket while searching for a free slot in which to store the record.
- If the bottom 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.**

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**

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	

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 buckets 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: $h_i(x) = (h(x) + ic) \% M$
  - c： constant other than 1
  - records with adjacent home buckets will not follow same probe sequence
- (Pseudo) Random probing: $h_i(x) = (h(x) + r_i) \% M$
  - r_i is the i^{th} value in a random permutation of numbers from 1 to M-1
  - insertions and searches use the *same* sequence of “random” numbers

《数据结构与算法》课程组
重庆大学计算机学院

End of Chapter