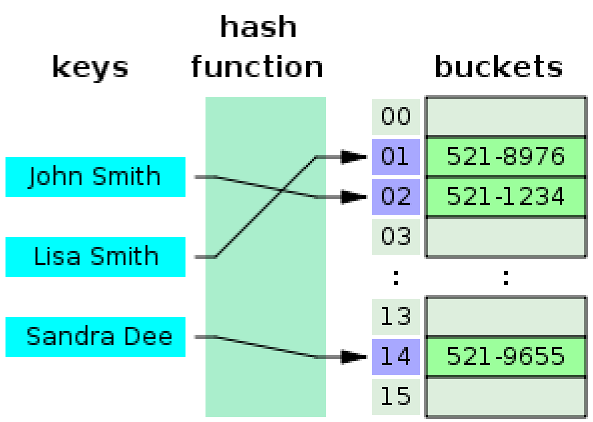
Keynotes on Data structure 2 -Hash table

Instructor: Shizhe Zhou

Course Code:00125401

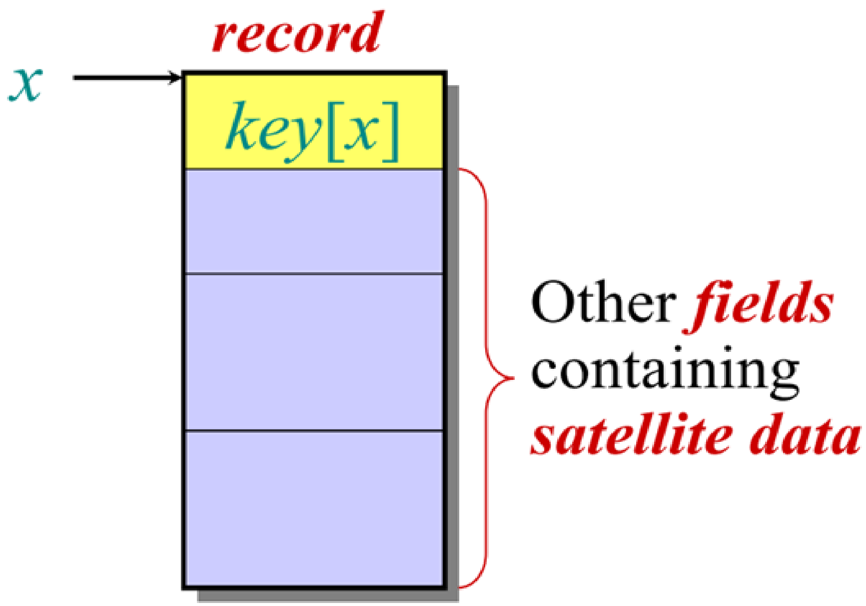
How to Implement a Dictionary?

- Sequences
  - ordered
  - unordered
- Binary Search Trees
- Direct Access Table
- Hashtables



An example : Symbol table problem

Symbol table S holding n ***records***:



Operations on S:

- INSERT(S, x)
- DELETE(S, x)
- SEARCH(S, k)

How should the data structure S be organized?

- The simplest DS: Direct Access Table
  - Simple deletion
  - Simple inserting – write over the slot in the array
  - Simplest search – indexing!!!

Direct Access Table

IDEA: Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \,.\,.\, m-1]$:

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Then, operations take $\Theta(1)$ time.

Problem: The range of keys can be large:
- 64-bit numbers (which represent

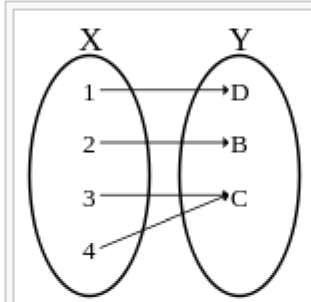
- Limitation: the keys could be drawn from a monster size set, say 2^64 integer (~ 18,446,744,073,709,551,616

[18 *Quintillion(cubic of a million)*]

Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and list implementation)

压缩映像 injection



Non-injective and surjective (surjection, or onto)

Basic Idea

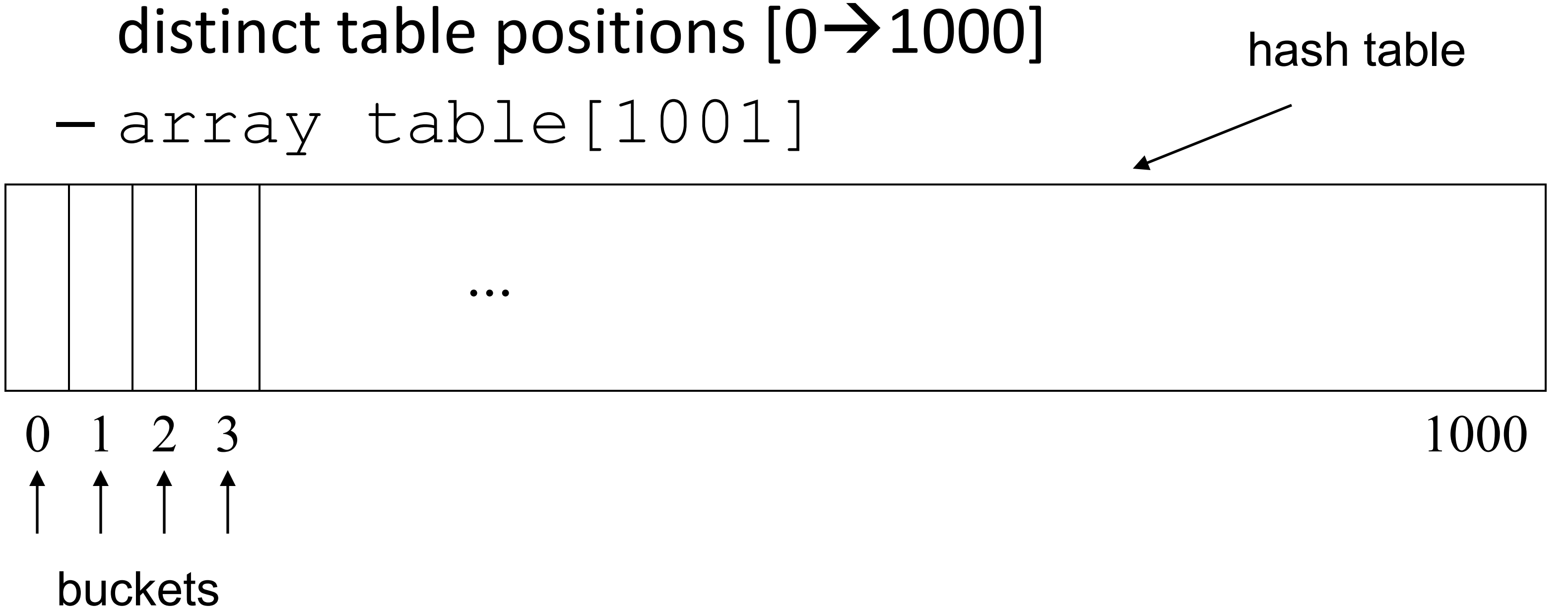
- Use *hash function* to map keys into positions in a *hash table*

<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, dictionary 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*

与map的区别: 不关心元素之间的顺序，如果希望元素按某种顺序排列，map比hash_map更合适. 是高效的关联容器。Even better than map!

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 ≈ 1,000 records at any given time
- Impractical to use hash table with 65,536 slots!

Hash Functions, collisions

- 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!

Collision Resolution Policies

- Two classes:
  - (1) Open hashing, a.k.a. **separate chaining [p.57]**
  - (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)

Closed Hashing

- Associated with closed hashing is a *rehash strategy*:

  "If we try to place *x* in bucket *h(x)* and find it occupied, find alternative location $h_1(x)$, $h_2(x)$, etc. Try each in order, if none empty table is full,"
- *h(x)* is called *home bucket*
- Simplest rehash strategy is called *linear hashing*

$$h_i(x) = (h(x) + i) \% D$$

- In general, our collision resolution strategy is to generate a sequence of hash table slots (probe sequence) that can hold the record; test each slot until find empty one (probing)

Example Linear (Closed) Hashing

- 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	

Operations Using Linear Hashing

- Test for membership: *findItem*
- Examine h(k), h_1(k), h_2(k), …, until we find k or an empty bucket or home bucket
- If no deletions at all, strategy works!
- What if deletions?
- If we reach empty bucket, cannot be sure that k is not somewhere else and empty bucket was occupied when k was inserted. So no way to know to go on or to delete just the bucket???
- Need special placeholder *deleted*, to distinguish bucket that was never used from one that once held a value
- May need to reorganize table after many deletions

Performance Analysis - Worst Case

- Initialization: O(b), b# of buckets *(or slots)*
- Insert and search: O(n), n number of elements in table; all n key values have same home bucket
- No better than linear list for maintaining dictionary!

Hash table

Type	Unsorted associative array	
Invented	1953	
Time complexity in big O notation		
	Average	Worst case
Space	$O(n)$[1]	$O(n)$
Search	$O(1 + n/k)$ k: number of buckets	$O(n)$
Insert	O(1)	O(n)
Delete	$O(1 + n/k)$	$O(n)$

Performance Analysis - Avg Case

- Distinguish between successful and unsuccessful searches
  - Delete = successful search for record to be deleted
  - Insert = unsuccessful search along its probe sequence
- Expected cost of hashing is a function of how full the table is: load factor $\alpha = n/b$
- It has been shown that average costs under linear hashing (probing) are:
  - Insertion: $1/2(1 + 1/(1 - \alpha)^2)$
  - Deletion: $1/2(1 + 1/(1 - \alpha))$

Improved Collision Resolution

- Linear probing: $h_i(x) = (h(x) + i) \% D$
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  - *{bad}* clustering of records, leads to long probing sequences
- Linear probing with skipping: $h_i(x) = (h(x) + ic) \% D$ 跳越搜索
  - c constant other than 1
  - records with adjacent home buckets will not follow same probe sequence
- (Pseudo)Random probing: $h_i(x) = (h(x) + r_i) \% D$
  - r_i is the i^{th} value in a random permutation of numbers from 1 to D-1
  - insertions and searches use the *same* sequence of "random" numbers

Example

I

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	

$h(k) = k\%11$

1. What if next element has home bucket 0?

   → go to bucket 3

   Same for elements with home bucket 1 or 2!

   Only a record with home position 3 will stay(**at home**).

   ⇒(几率) p = 4/11 that next record will go to bucket 3

2. Similarly, records hashing to 7,8,9 will end up in 10
3. Only records hashing to 4 will end up in 4 (p=1/11); same for 5 and 6

II

insert 1052 (h.b. 7)

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	1052

next element in bucket 3 with p = **8**/11

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

Division method

Assume all keys are integers, and define

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

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: 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 $h(k) = 011010_2$.

Division method (continued)

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

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.

Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where rsh is the "bit-wise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

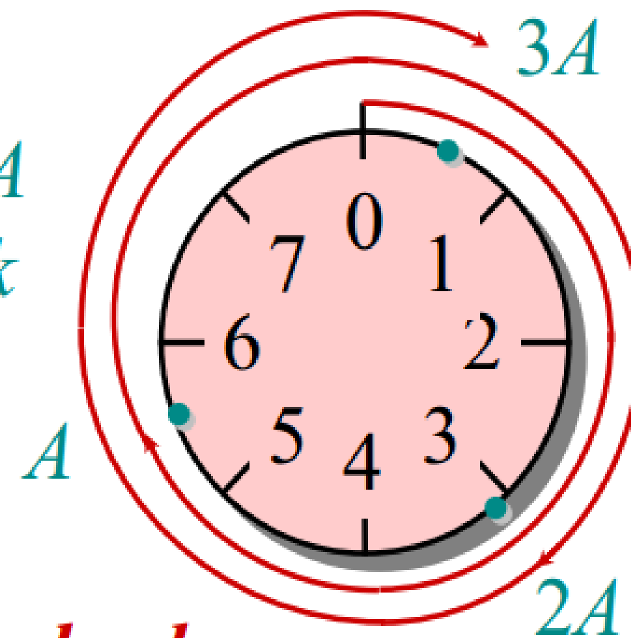
- Don't pick A too close to 2^w.
- Multiplication modulo 2^w is fast.
- The rsh operator is fast.

Multiplication method example

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$\begin{array}{rr}
 & 1\,0\,1\,1\,0\,0\,1 = A \\
\times & 1\,1\,0\,1\,0\,1\,1 = k \\
\hline
 & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1
\end{array}$$

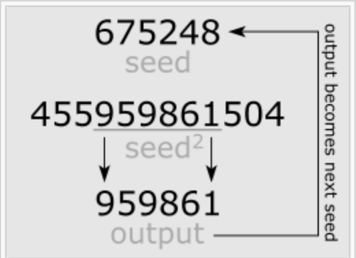


Modular wheel

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(**equivalent to the middle digits**) of x^2, where 2^r=D (a base-D digit)

  - better, because most or all of bits of key contribute to result**(随机性:随机分布的key得到随机hash address)**

exceptions:0100, 2500, 3792, and 7600

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

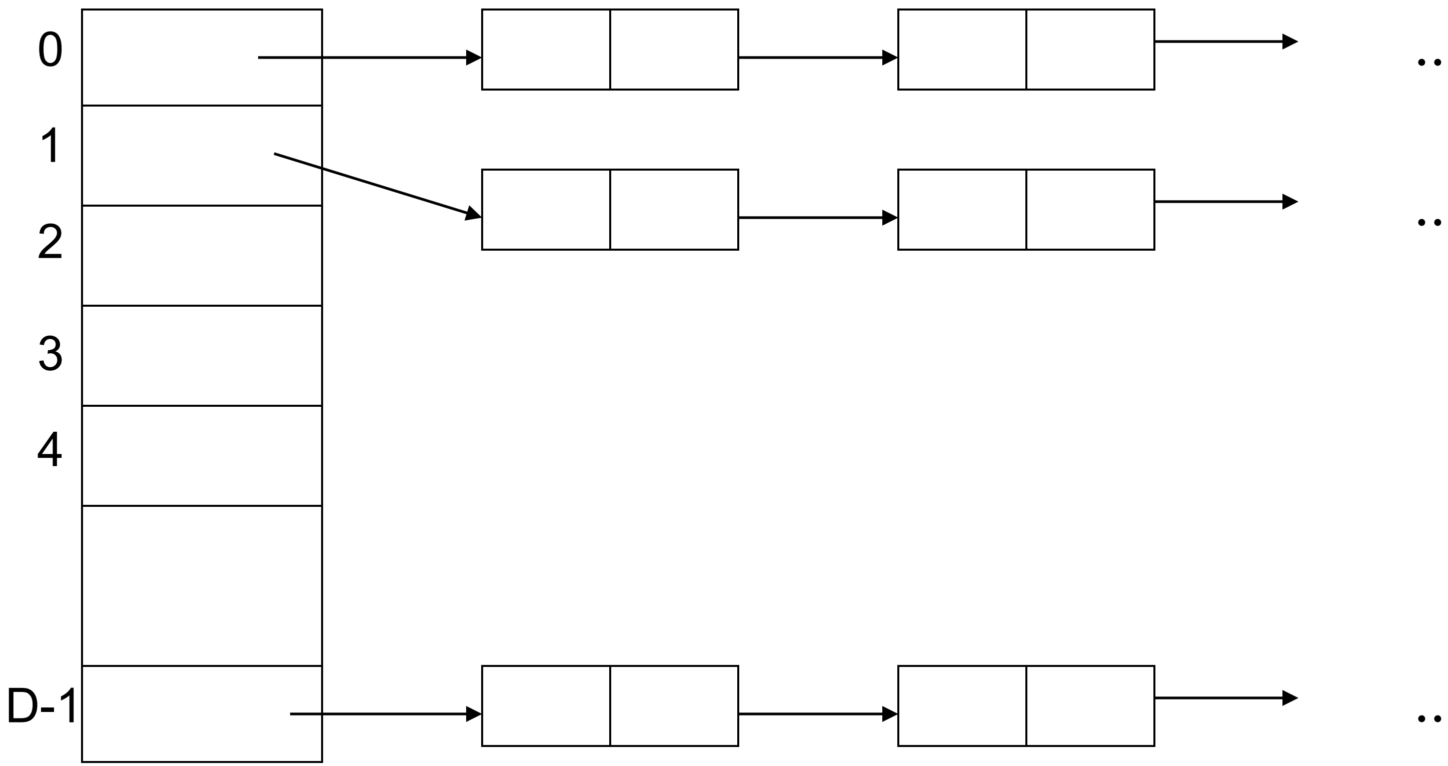
- Much better: Cyclic Shift

```
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;
}
```

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 for Open hash

- Open hashing is 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 $\Rightarrow$ 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

Comparison with Closed Hashing

- Worst case performance is O(n) for both

- Number of operations for hashing
  - 23 6 8 10 23 5 12 4 9 19
  - D=9
  - h(x) = x % D

Hashing Problem

- Draw the 11 entry hashtable for hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the function (2i+5) mod 11, closed hashing, linear probing
- Pseudo-code for listing all identifiers in a hashtable in lexicographic order, using open hashing, the hash function h(x) = first character of x. What is the running time?