CSCI 340 Data Structures & Algorithm Analysis

Hashing

Thanks to Dr. Rada Mihalcea of the University of North Texas for sharing her slides – Modified by Minmei Hou.

Dictionary ADT

- Dictionary: abstract data type, a set with operations with searching, insertion, and deletion.
  - Key : used to identify the entity
  - e.g. records (book records, student records)
  - e.g. symbol table in a compiler

How to Implement a Dictionary?

- Sequences
  - ordered
  - unordered
- Binary Search Trees
  - Height balanced b.s.t.
  - STL map

Hashing

- Hashing – Another important and widely useful technique for implementing dictionaries
  - Distributing n keys among a one-dimensional array $H[0, 1, \ldots, m-1]$ ~ hash table (with size m)
  - 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)

Basic Idea

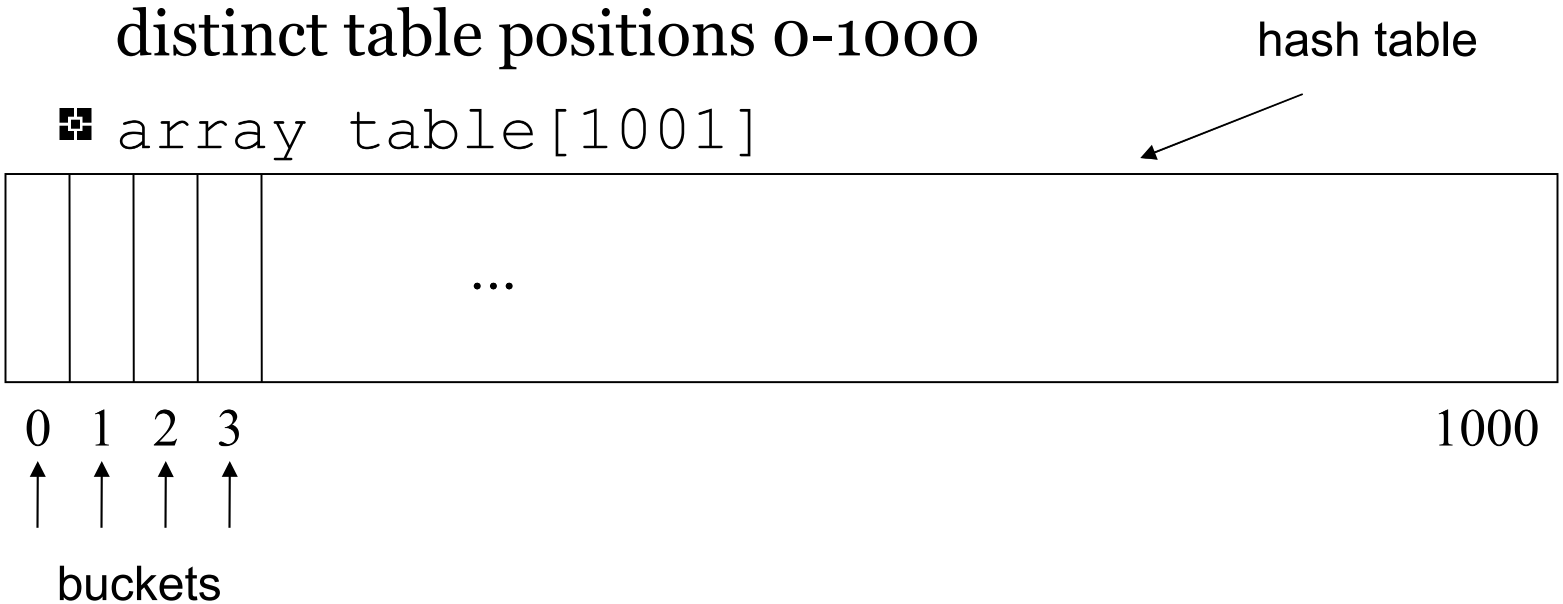
- Use *hash function* to map keys into positions in a *hash table*
  - The function assigns each key an integer between [0, m-1] (hash address)

<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(m) time to initialize hash table
  - m is the number of positions or buckets in hash table
- O(1) time to perform *insert*, *remove*, *search*

Ideal Case is Unrealistic

- 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) and expect ≈ 1,000 records at any given time
  - Might become 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

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

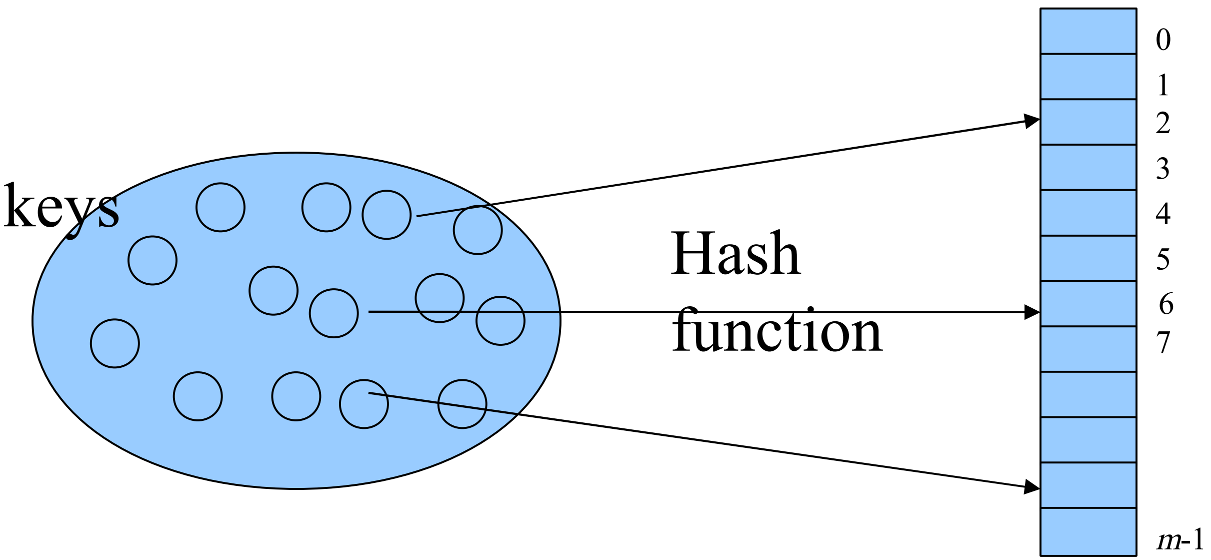
- Example: hash table with 11 buckets

  $h(k) = k\%11$

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

  $58 \rightarrow 3$: a collision happens!

Hashing



Examples:

Keys (K)	Hash function
Nonnegative integers	$h(K) = K \bmod m$
Letters of some alphabet	$h(K) = h(\ ord(K)\ ) = ord(K) \bmod m$
Strings ($c_0c_1 \ldots c_{s-1}$)	$h(K) = sum(\ ord(C_i)\ ) \bmod m$

Examples of hashing

Keys are integers:

Key K	Hash functions		
	K mod 7	K mod 11	K mod 5
7	0	7	2
11	4	0	1
5	5	5	0
14	0	3	4

Examples of hashing

Keys are chars:

Key K	Ord(K)	Hash function	
		Ord(K) mod 5	Ord(K) mod 7
a	1	1	1
b	2	2	2
c	3	3	3
d	4	4	4
e	5	0	5
f	6	1	6
g	7	2	0
h	8	3	1
i	9	4	2
j	10	0	3
k	11	1	4
l	12	2	5
m	13	3	6
...			

Examples of hashing

Keys are strings

Key	Sum (Ord(C_i))	Hash function	
		Sum mod 7	Sum mod 13
a	1	1	1
and	1+14+4 = 19	5	6
are	1+18+5 = 24	3	11
soon	19+15+15+14 = 63	0	11
money	13+15+14+5+25 = 72	2	7

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
q	17
r	18
s	19
t	20
u	21
v	22
w	23
x	24
y	25
z	26

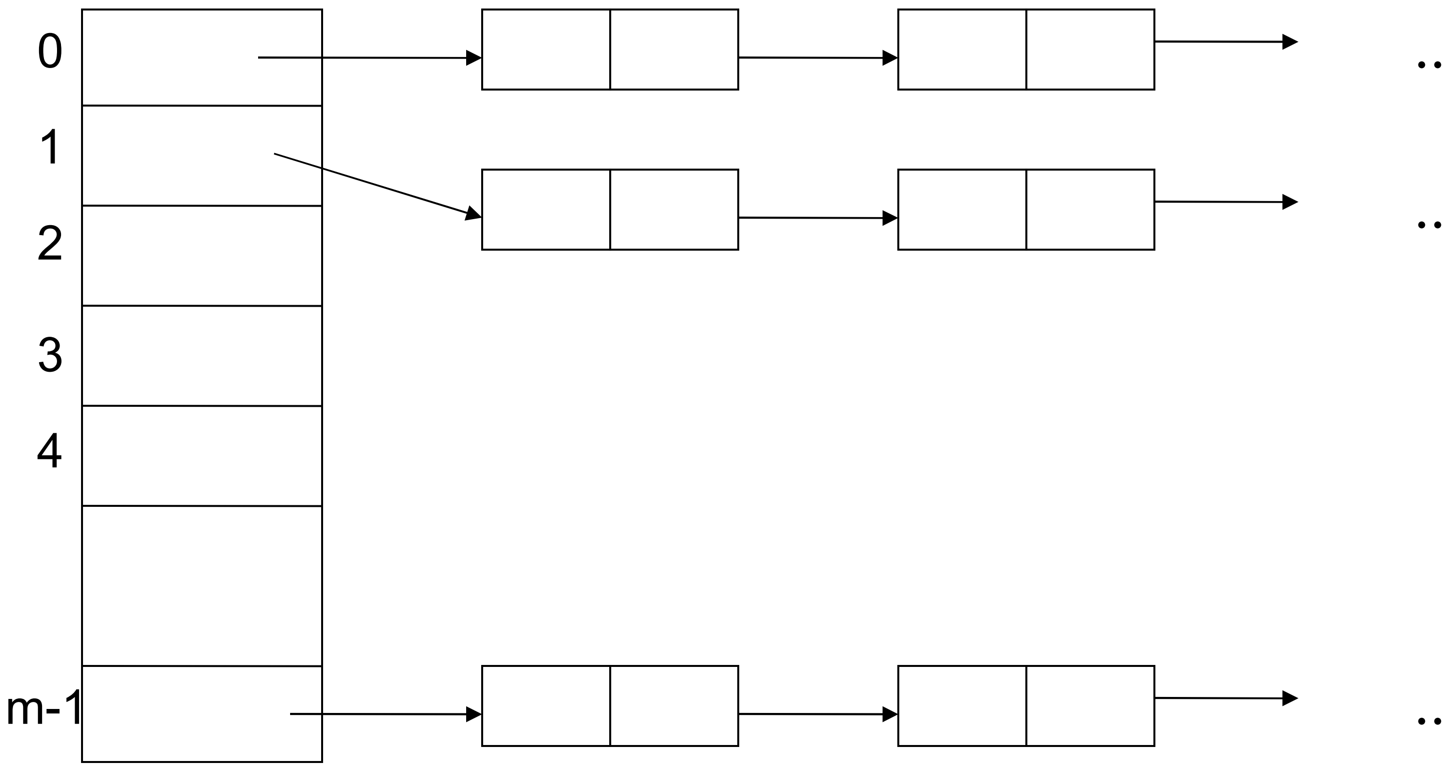
Collision Resolution Policies

- Collision: two keys being hashed into the same hash address.
  - If $m < n$, there must be collision(s)
  - Regardless m and n, in the worst case, all keys could be mapped to the same address (When table size and hash function are appropriately chose, it's rare)
- Two methods to solve collision:
  - Open hashing, a.k.a. separate chaining
  - 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,
  - by frequency of access order

Open Hashing Data Organization



Keys are stored in linked list attached to buckets of a hash table. Each list contains all the keys hashed to this bucket.

Open Hashing

Example: using a hash table m=13, store the following string:

a fool and his money are soon parted

1 9 6 10 7 11 11 12

Open Hashing operations

- Search
  - Compute $h(k)$, locate bucket in hash table
  - Search in the linked list of the bucket
- Insertion
  - Compute $h(k)$, locate bucket in hash table
  - Insert in a linked list
    - Insert at head
    - Insert in a sorted list
- Deletion
  - Compute $h(k)$, locate bucket in hash table
  - Remove the item from the linked list at the bucket

Analysis

- The efficiency of all operations depends on the size of the linked list at a bucket, which depends on table size, dictionary property, and quality of hash function
  - 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/m items (α: load factor)
    - Average number of comparisons in successful search: $\alpha/2$
    - Average number of comparisons in unsuccessful search: α
  - If we can estimate n and choose m to be roughly the same, then by average a bucket will have only one or two members
    - α is too large: inefficient in search
    - α is too small: waste lots of empty list, inefficient in space

Analysis Cont'd

<u>Average time per dictionary operation:</u>

- *m* buckets, *n* elements in dictionary ⇒ average *n*/*m* elements per bucket
- *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 bucket, running time constant

Closed Hashing

- No chain. All keys are stored in hash table.
  – Table size m >= number of keys n
- 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 is empty, table is full,"

- $h(x)$ is called *home bucket*
- Simplest rehash strategy is called *linear probing*

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

- 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

- Suppose *m*=8. Suppose 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:

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

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

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

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

etc.

7, 0, 1, 2

- Wraps around the beginning of the table! – table is a circular array

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

Operations Using Linear Hashing

- Search based on key *k*
  - Examine h(*k*), $h_1(k)$, $h_2(k)$, …, until we find *k* or an empty bucket (assuming there's no deletion)
  - If no deletions possible, strategy works!
  - Example:

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

    a, b, c, d are inserted

    Search a:

    Search b:

    Search c:

    Search d:
  - What if *c* is deleted?

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

Operations Using Linear Hashing

- Search based on key *k* (cont.)
  - What if there are deletions?
    - If we reach empty bucket, cannot be sure that *k* is not somewhere else and empty bucket was occupied when *k* was inserted
    - 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

Operations Using Linear Hashing

- Search
  - Compute $h(k)$, find the bucket
  - Follow linear probing until an item is found or until a bucket that is empty which is not caused by deletion
  - If an bucket is empty from a deletion, the search shall not stop here
- Insertion
  - Compute h(k), *find* the bucket
  - Follow linear probing until an empty bucket regardless it's caused by deletion or not
  - Place the item in the bucket
- Deletion
  - Search for the item
  - Remove the item and label the bucket by "deleted"

Performance Analysis - Worst Case

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

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 $\alpha = n/m$
- It has been shown that average costs under linear hashing (probing) are:
  - Insertion: $1/2 * (1 + 1/(1 - \alpha)^2)$
  - Deletion: $1/2 * (1 + 1/(1 - \alpha))$

An Example of potential problem

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?

   $\rightarrow$ go to bucket 3

   * Same for elements with home bucket 1 or 2!

   * A record with home position 3 will stay.

   $\Rightarrow$ p = 4/11 that next record will go to bucket 3

2. Similarly, records hashing to 7,8,9,10 will end up in 10 ==> p = 4/11

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

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

Improved Collision Resolution

- (Pseudo)Random probing: $\mathrm{h}_i(x) = (\mathrm{h}(x) + r_i) \% m$
  - 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

Hash Functions - Numerical Values

Consider: $h(x) = x\%16$

- depends solely on least significant four bits of key
- poor distribution, not very random

- 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:

```
int h(String x, int m) {
    int i, sum;
    for (sum=0, i=0; i<x.length(); i++)
        sum+= (int)x.charAt(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 – mix the components of a string

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

Comparison open and closed Hashing

- Worst case performance is O(n) for both

- Number of operations for hashing
  - 23 6 8 10 24 5 12 4 9 19
  - m=9
  - h(x) = x % m