Hashing

Bruce

How to Implement a Dictionary?

- Sequences
  - ordered
  - unordered
- Binary Search Trees
- Hashtables

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 chain implementation)

Basic Idea

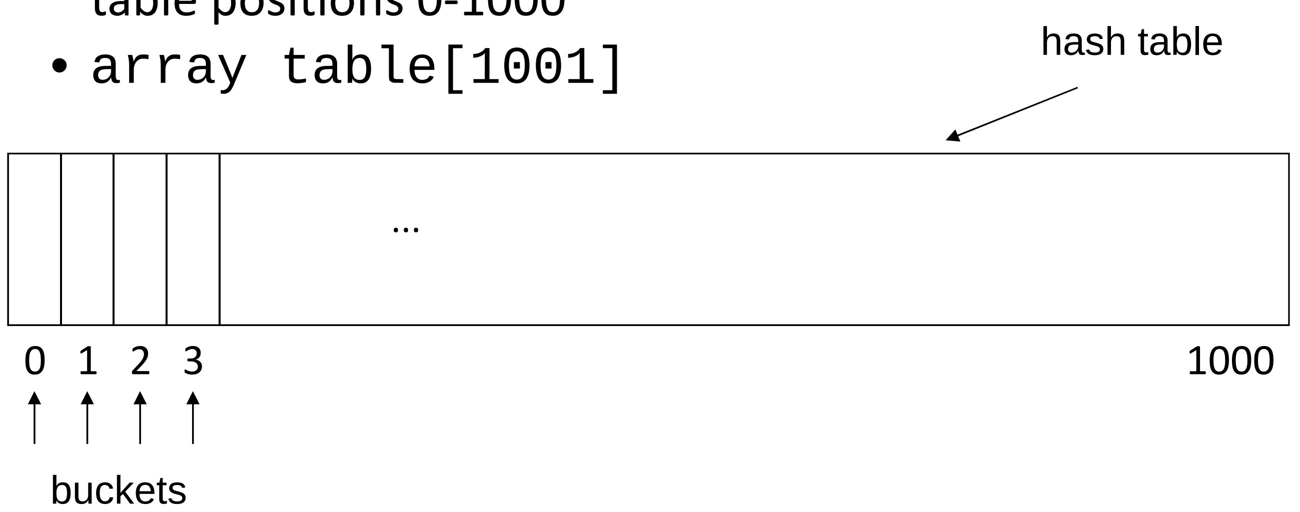
- Use *hash function* to map keys into positions in a *hash table*

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, 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*

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!

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)

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 possible, 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
- 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
- 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!

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
  - clustering of records, leads to long probing sequences
- Linear probing with skipping: $h_i(x) = (h(x) + ic) \% D$
  - 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	

1. What if next element has home bucket 0?

$h(k) = k\%11$

$\rightarrow$ go to bucket 3

Same for elements with home bucket 1 or 2!

Only 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 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 $p = 8/11$

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

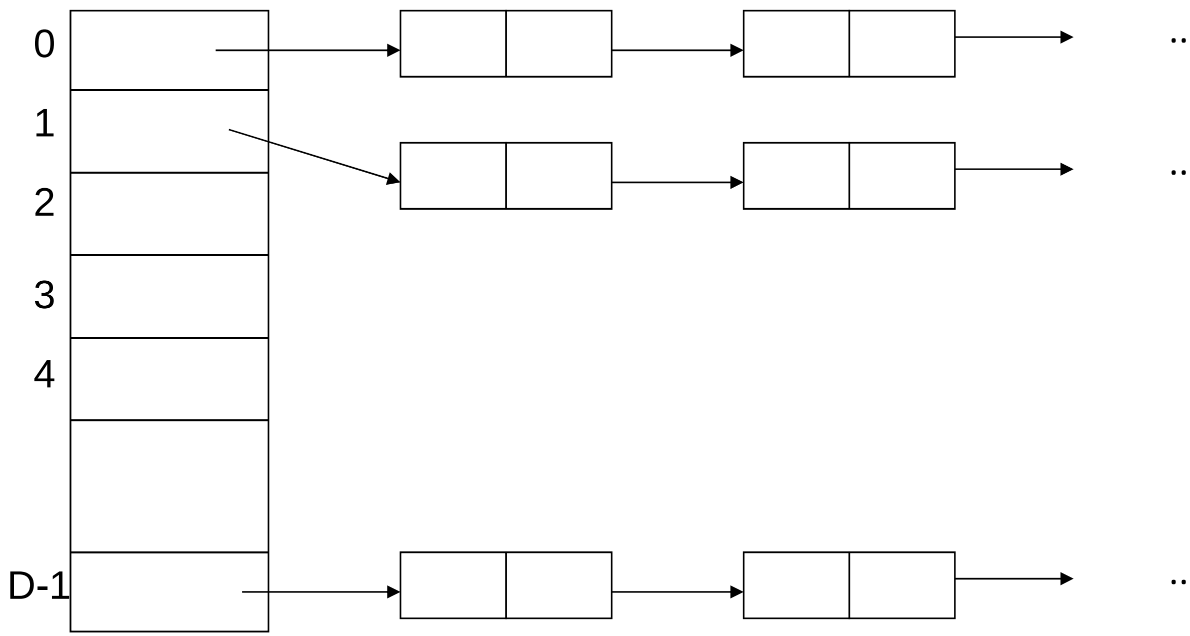
- Much better: Cyclic Shift

```
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;
}
```

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 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