Hashing

October 19, 2016

Dictionary ADT

- Data structure with just three basic operations:
  - findItem (i): find item with key (identifier) i
  - insert (i): insert i into the dictionary
  - remove (i): delete i
  - Just like words in a Dictionary
- Where do we use them:
  - Symbol tables for compiler
  - Customer records (access by name)
  - Games (positions, configurations)
  - Spell checkers,etc.

How to Implement a Dictionary?

We can use:

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

Hashing

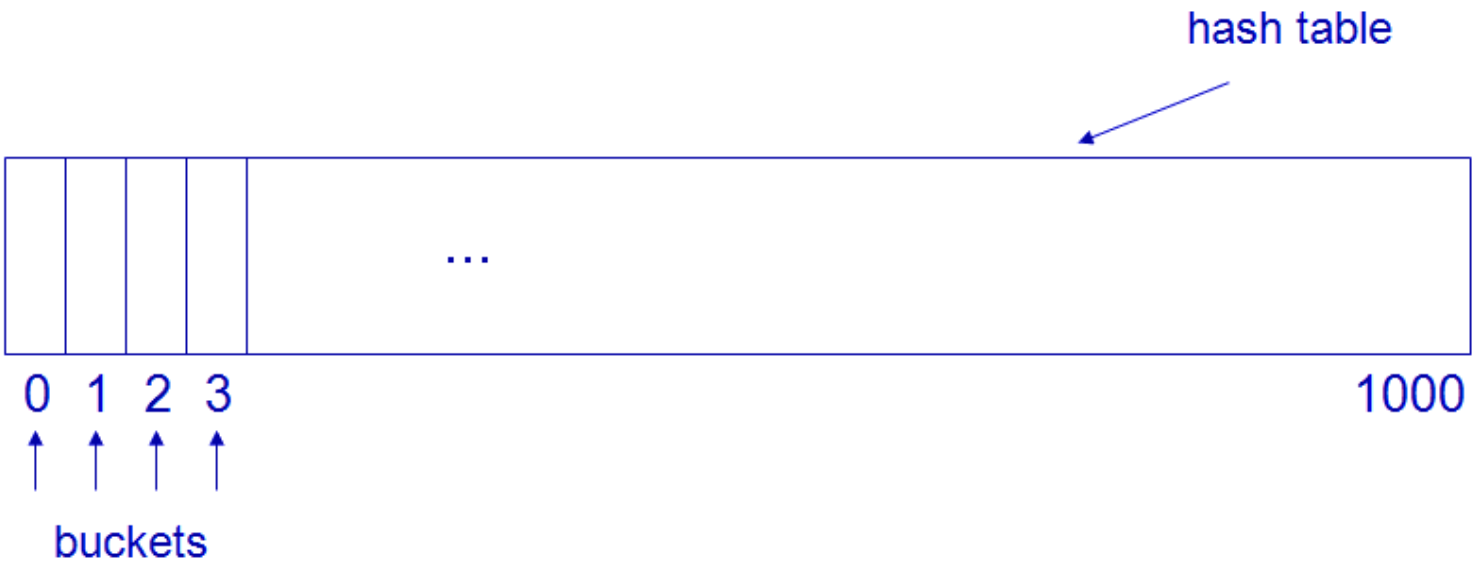
- An important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array based and linked implementation)

Basic Idea

- Use the hash function to map keys into positions in a hash table.
- Ideally
  - If element e has key k and h is the hash function, then e is stored in position $h(k)$ of the table
  - To search for e, compute $h(k)$ to locate the position. If no element has key k, dictionary does not contain e.

Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 1000 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 the hash table (b is the number of positions or buckets in the hash table)
- $O(1)$ time to perform `insert, remove, findItem`

Ideal Case is Unrealistic

- Works for implementing dictionaries, but many applications have key ranges that are too large to have one-to-one 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 the key range is too large, use a hash table with fewer buckets and a hash function which maps multiple keys to the 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 is the number of buckets in the 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

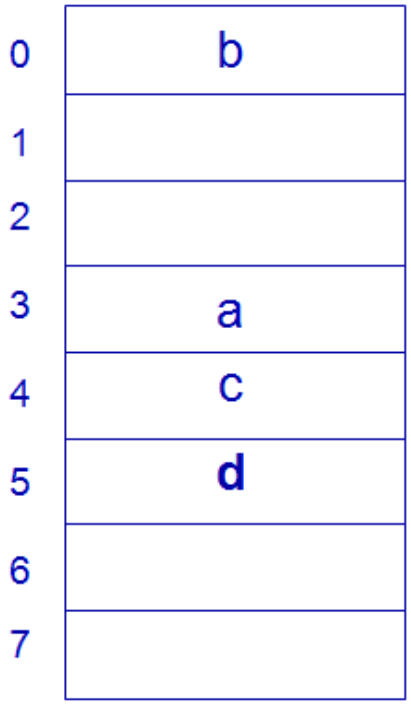
- Two approaches:
  1. Open hashing, or separate chaining
  2. Closed hashing, or open addressing
- The difference is about whether collisions are stored outside the table (open hashing) or whether collisions result in storing one of the records at another slot in the table (closed hashing)

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 an alternative location $h_1(x), h_2(x)$, etc. Try each in order, if none of them is empty then the table is full,"
- $h(x)$ is called home bucket
- The 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 the 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 is already filled!
- Probe sequence using linear hashing:
  $h1(d) = (h(d) + 1)\%8 = 4\%8 = 4$
  $h2(d) = (h(d) + 2)\%8 = 5\%8 = 5*$
  $h3(d) = (h(d) + 3)\%8 = 6\%8 = 6$
  etc.
  $7, 0, 1, 2$
- Wraps around the beginning of the table!



Operations Using Linear Hashing

- Test for membership: `findItem`
- Examine $h(k), h_1(k), h_2(k), \ldots,$ until we find k or an empty bucket or home bucket
- If no deletions are possible, the strategy works!
- What if there are deletions?
- If we reach empty bucket, we cannot be sure that k is not somewhere else and the empty bucket was occupied when k was inserted
- Need a special placeholder `deleted`, to distinguish the bucket that was never used from the one that once held a value
- May need to reorganize the table after many deletions

Performance Analysis - Worst Case

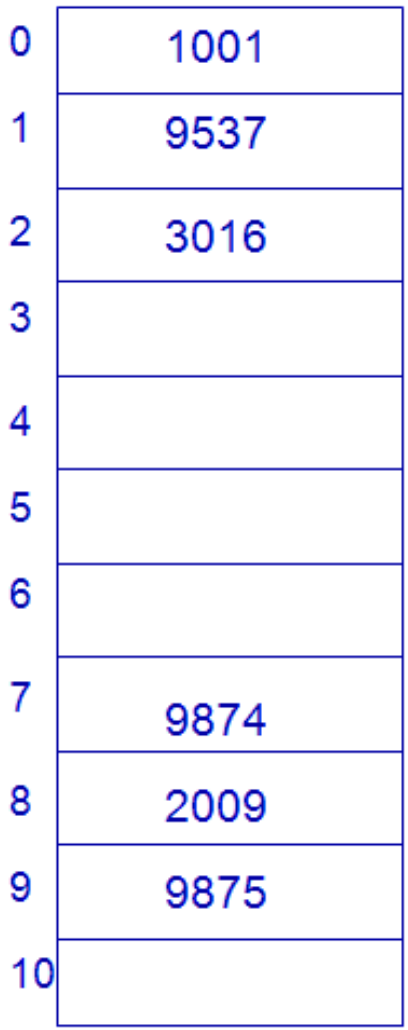
- Initialization: $O(b)$, b is the # of buckets
- Insert and search: $O(n)$, n is the number of the elements in the table; all n key values have the same home bucket
- Not better than a linear list for maintaining dictionary!

Improved Collision Resolution

- Linear probing: $h_i(x) = (h(x) + i)\%D$
  - all buckets in th etable 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 is a constant other than 1
  - records with adjacent home buckets will not follow the 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

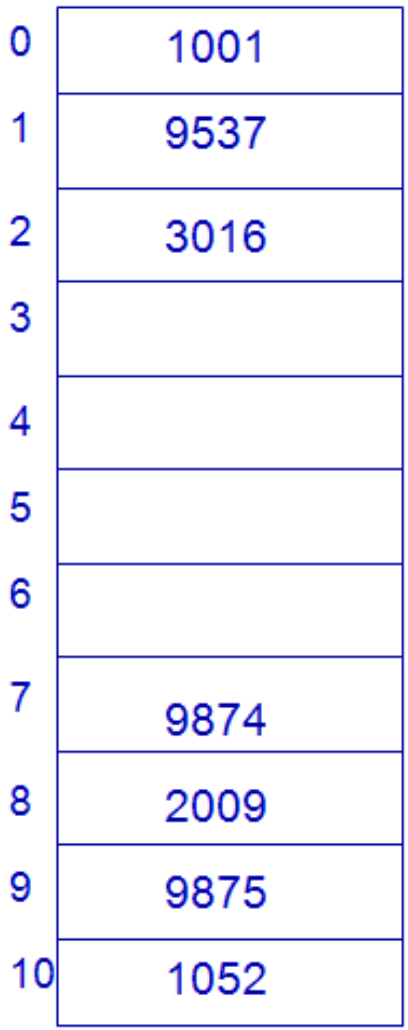
Example

1. What if the next element has home bucket 0? $\rightarrow$ 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



Example (Cont.)

- insert 1052 (home bucket: 7)
- 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 $0, 1 \ldots, K$, 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

- Polynomial hash codes

```
static long hashCode(String key, int D)
{
        int h = key.charAt(0);

        for (int i=1, i<key.length(); i++)
        {
                h = h*a;
                h += (int) key.charAt(i);
        }
        return h % D;
}
```

  - They have found that a = 33, 37, and 41 have less than 7 collisions.

Hash Function –Strings of Characters

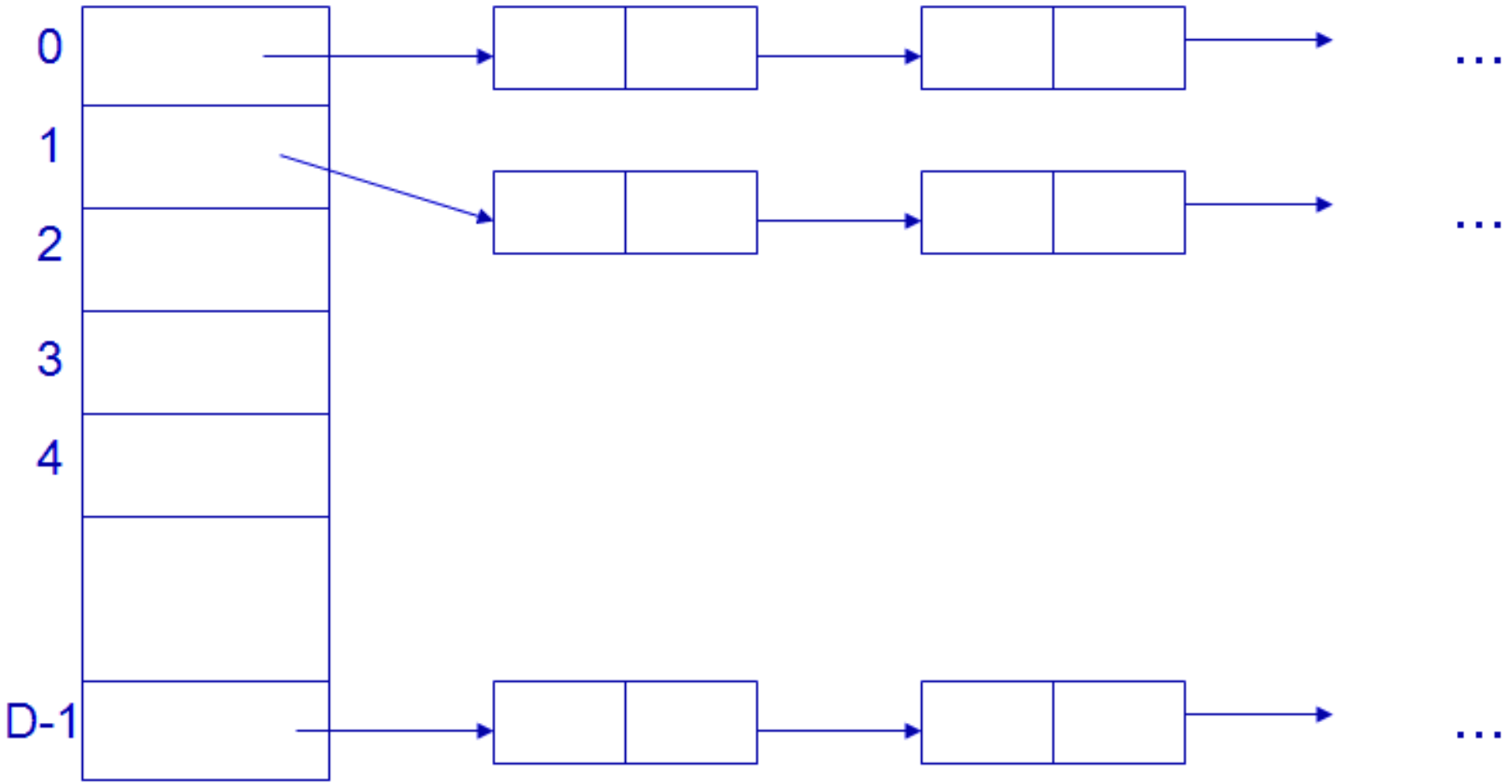
- Much better: Cyclic Shift

```
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 such as
  - 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 the main memory, implemented with a standard in-memory linked list
- We hope that the number of elements per bucket is roughly equal in size, so that the lists will be short
- If there are n elements in the set, then each bucket will have roughly n/D members
- 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.)

- Average time per dictionary operation:
  - D buckets, n elements in dictionary $\Rightarrow$ average n/D elements per bucket
  - `insert, findItem, remove` operations 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