Data Structures and Algorithms

Werner Nutt
nutt@inf.unibz.it
http://www.inf.unibz/it/~nutt

Part 7

Academic Year 2011-2012

Acknowledgements & Copyright Notice

These slides are built on top of slides developed by Michael Boehlen**.** Moreover, some material (text, figures, examples) displayed in these slides is courtesy of **Kurt Ranalter**. Some examples displayed in these slides are taken from [**Cormen, Leiserson, Rivest and Stein**, ``Introduction to Algorithms'', MIT Press], and their copyright is detained by the authors. All the other material is copyrighted by **Roberto Sebastiani**. Every commercial use of this material is strictly forbidden by the copyright laws without the authorization of the authors. No copy of these slides can be displayed in public or be publicly distributed without containing this copyright notice.

Data Structures and Algorithms
Week 7

1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

Data Structures and Algorithms
Week 7

1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

Dictionary

- *Dictionary* – a dynamic data structure with methods:
  - **Search(S, k)** – *an access operation that returns a pointer x to an element where x.key = k*
  - **Insert(S, x)** – *a manipulation operation that adds the element pointed to by x to S*
  - **Delete(S, x)** – *a manipulation operation that removes the element pointed to by x from S*
- An element has a *key* part and a *satellite data* part.

Dictionaries

- Dictionaries store elements so that they can be located quickly using **keys.**
- A dictionary may hold bank accounts.
  - Each account is an object that is identified by an account number.
  - Each account stores a lot of additional information.
  - An application wishing to operate on an account would have to provide the account number as a search **key.**

Dictionaries/2

- If order (methods such as *min*, *max*, *successor*, *predecessor*) is not required it is enough to check for **equality**.
- Operations that require ordering are still possible but cannot use the dictionary access structure.
  - Usually all elements must be compared, which is slow.
  - Can be OK if it is rare enough

Dictionaries/3

- Different data structures to realize dictionaries
  - arrays
  - linked lists
  - **Hash tables**
  - Binary trees
  - Red/Black trees
  - B-trees
- In Java:
  - java.util.Map – interface defining Dictionary ADT

Data Structures and Algorithms
Week 7

1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

The Problem

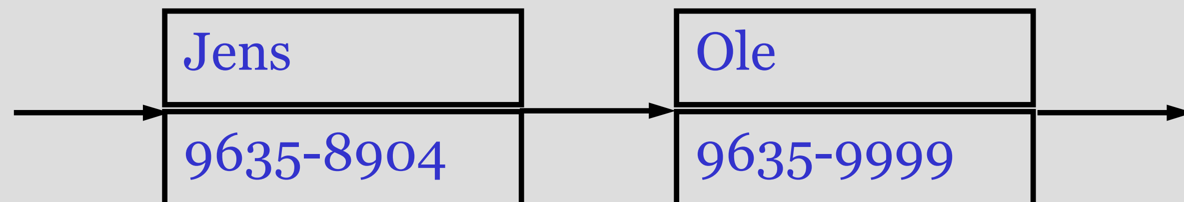
- XY Telecom, a large phone company, wants to provide a caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers range from 0 to $r = 10^8 - 1$
  - want to do this as efficiently as possible

The Problem/2

- Two suboptimal ways to design this dictionary
  - direct addressing: an array indexed by key:
    - Requires O(1) time,
    - Requires O(r) space - huge amount of wasted space

(null)	(null)	Jens	(null)	(null)
0000-0000	0000-0001	9635-8904	9635-8905	9999-9999

  - a linked list: requires O(n) time, O(n) space



Another Solution: Hashing

- We can do better, with a **Hash table** *of size m*.
- Like an array, but with a **function** to map the large range into one which we can manage.
  - e.g., take the original key, modulo the (relatively small) size of the table, and use that as an index
- Insert (9635-8904, Jens) into a hash table with, say, five slots (m = 5)
  - 96358904 mod 5 = 4

(null)	(null)	(null)	(null)	Jens
0	1	2	3	4

- O(1) expected time, O(n+m) space

Data Structures and Algorithms
Week 7

1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

Hash Functions

- Need to choose a good hash function (HF)
  - quick to compute
  - distributes keys uniformly throughout the table
- How to deal with hashing non-integer keys:
  - find some way of turning the keys into integers
    - in our example, remove the hyphen in 9635-8904 to get 96358904
    - for a string, add up the ASCII values of the characters of your string (e.g., java.lang.String.hashCode())
  - then use a standard hash function on the integers

HF: Division Method

- Use the remainder: $h(k) = k \bmod m$
  - k is the key, m the size of the table
- Need to choose m
- $m = b^e$ **(bad)**
  - if m is a power of 2, $h(k)$ gives the ***e*** least significant bits of k
  - all keys with the same ending go to the same place
- m prime **(good)**
  - helps ensure uniform distribution
  - primes not too close to exact powers of 2 are best

HF: Division Method/2

- Example 1
  - hash table for n = 2000 character strings, ok to investigate an average of three attempts/search
  - m = 701
    - a prime near 2000/3
    - but not near any power of 2
- Further examples
  - m = 13
    - $h(3) = 3$
    - $h(12) = 12$
    - $h(13) = 0$

HF: Multiplication Method

- Use $h(k) = \lfloor m\ (k\,A \bmod 1) \rfloor$
  - k is the key
  - m the size of the table
  - A is a constant $0 < A < 1$
  - (k A mod 1): the fractional part of k A
- The steps involved
  - map $0 \ldots k_{max}$ into $0 \ldots k_{max} A$
  - take the fractional part (mod 1)
  - map it into $0 \ldots m\text{-}1$

HF: Multiplication Method/2

- Choice of m and A
  - Value of m is not critical:
    typically, for some p use $m = 2^p$
  - Optimal choice of A depends on the characteristics of the data
    - Knuth says use $A = \frac{\sqrt{5}-1}{2}$ = 0.618033988

HF: Multiplication Method/3

- Assume 7-bit binary keys, $0 \leq k < 128$
- $m = 64 = 2^6$, p = 6
- $A = 89/128 = .1011001$, $k = 107 = 1101011$
- Computation of h(k):

```
       .1011001 A
        1101011 k
 1001010.0110011 kA
        .0110011 kA mod 1
  011001.1       m(kA mod 1)
```

- Thus, $h(k) = 25$

Data Structures and Algorithms
Week 7

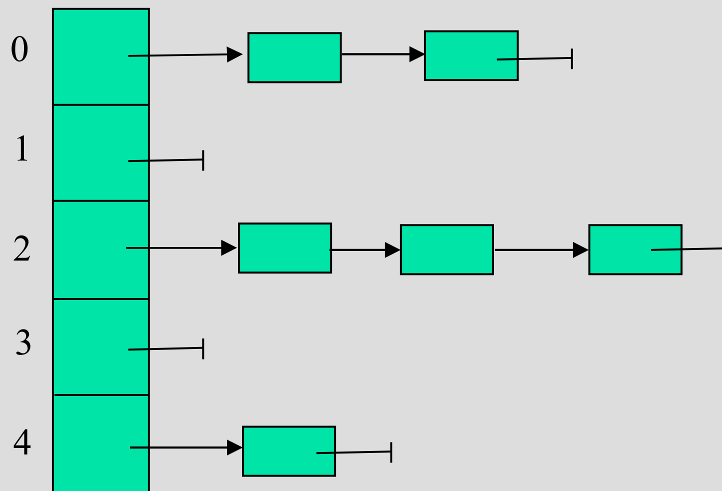
1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

Collisions

- Assume a key is mapped to an already occupied table location
  - what to do?
- Use a **collision handling** technique
- 3 techniques to deal with collisions:
  - chaining
  - open addressing/linear probing
  - open addressing/double hashing

Chaining

- **Chaining** maintains a table of links,
  indexed by the keys,
  to **lists** of items with the same key



Open Addressing

- All elements are stored in the hash table (can fill up), i.e., $n \le m$
- Each table entry contains
  either an element or null
- When searching for an element,
  systematically probe table slots
- Modify hash function to take probe number i as second parameter

$$h: U \times \{ 0, 1, \ldots, m-1 \} \rightarrow \{ 0, 1, \ldots, m-1 \}$$

Open Addressing/2

- Hash function, *h*, determines the sequence of slots examined for a given key
- Probe sequence for a given key *k* given by

$$(h(k,0), h(k,1), \ldots, h(k,m-1))$$

a permutation of $(0, 1, \ldots, m-1)$

Linear Probing

```
LinearProbingInsert(k)
    01 if (table is full) error
    02 probe = h(k)
    03 while (table[probe] occupied)
    04    probe = (probe+1) mod m
    05 table[probe] = k
```

- If the current location is used, try the next table location: $h(key,i) = (h_1(key)+i) \bmod m$
- Lookups walk along the table until the key or an empty slot is found
- Uses less memory than chaining
  - one does not have to store all those links
- Slower than chaining
  - one might have to probe the table for a long time

Linear Probing/2

- Problem "primary clustering":
  long lines of occupied slots
  - A slot preceded by i full slots has a high probability of getting filled: (i+1)/m
- Alternatives: (quadratic probing,)
  double hashing
- Example:
  - $h(k) = k \bmod 13$
  - insert keys: 18 41 22 44 59 32 31 73

Double Hashing

- Use two hash functions:
  $h(key,i) = (h1(key) + i*h2(key)) \bmod m$, i=0,1,...

```
DoubleHashingInsert(k)
01 if (table is full) error
02 probe = h1(k)
03 offset = h2(k)
03 while (table[probe] occupied)
04    probe = (probe + offset) mod m
05 table[probe] = k
```

- Distributes keys much more uniformly than linear probing.

Double Hashing/2

- $h2(k)$ must be relative prime to m to search the entire hash table
  - Suppose $h2(k) = k*a$ and $m = w*a$, $a > 1$
- Two ways to ensure this:
  - m is power of 2, h2(k) is odd
  - m: prime, $h2(k)$: positive integer < m
- Example
  - $h1(k) = k \bmod 13$, $h2(k) = 8 - (k \bmod 8)$
  - insert keys: 18 41 22 44 59 32 31 73

Open addressing: delete

- Complex to delete from
  - A slot may be reached from different points
    - We cannot simply store “NIL”: we'd loose the information necessary to retrieve other keys
  - Possible solution: mark the deleted slot as “deleted”, insert also on “deleted”
    - Drawback: retrieval time no more depending on load factor: potentially lots of “jumps” on “deleted” slots
- When deletion admitted/frequent, chaining preferred

Data Structures and Algorithms
Week 7

1. Dictionaries
2. Hashing
3. Hash Functions
4. Collisions
5. Performance Analysis

Analysis of Hashing:

- An element with key k is stored in slot $h(k)$ (instead of slot k without hashing)
- The hash function h maps the universe U of keys into the slots of hash table $T[0...m-1]$

  $h: U \rightarrow \{ 0, 1, ..., m-1 \}$
- Assumption: Each key is equally likely to be hashed into any slot (bucket):

  simple uniform hashing
- Given hash table T with m slots holding n elements, the **load factor** is defined as $\alpha = n/m$

Analysis of Hashing/2

- Assume time to compute $h(k)$ is $\Theta(1)$
- To find an element
  - using h, look up its position in table T
  - search for the element in the linked list of the hashed slot
  - *uniform* hashing yields an average list length $\alpha = n/m$
  - expected number of elements to be examined α
  - search time $O(1+\alpha)$

Analysis of Hashing/3

- Assuming the number of hash table slots is proportional to
the number of elements in the table

  $n = O(m)$

  $\alpha = n/m = O(m)/m = O(1)$

  – searching takes constant time on average
  – insertion takes $O(1)$ worst-case time
  – deletion takes $O(1)$ worst-case time (pass the element not key, lists are doubly-linked)

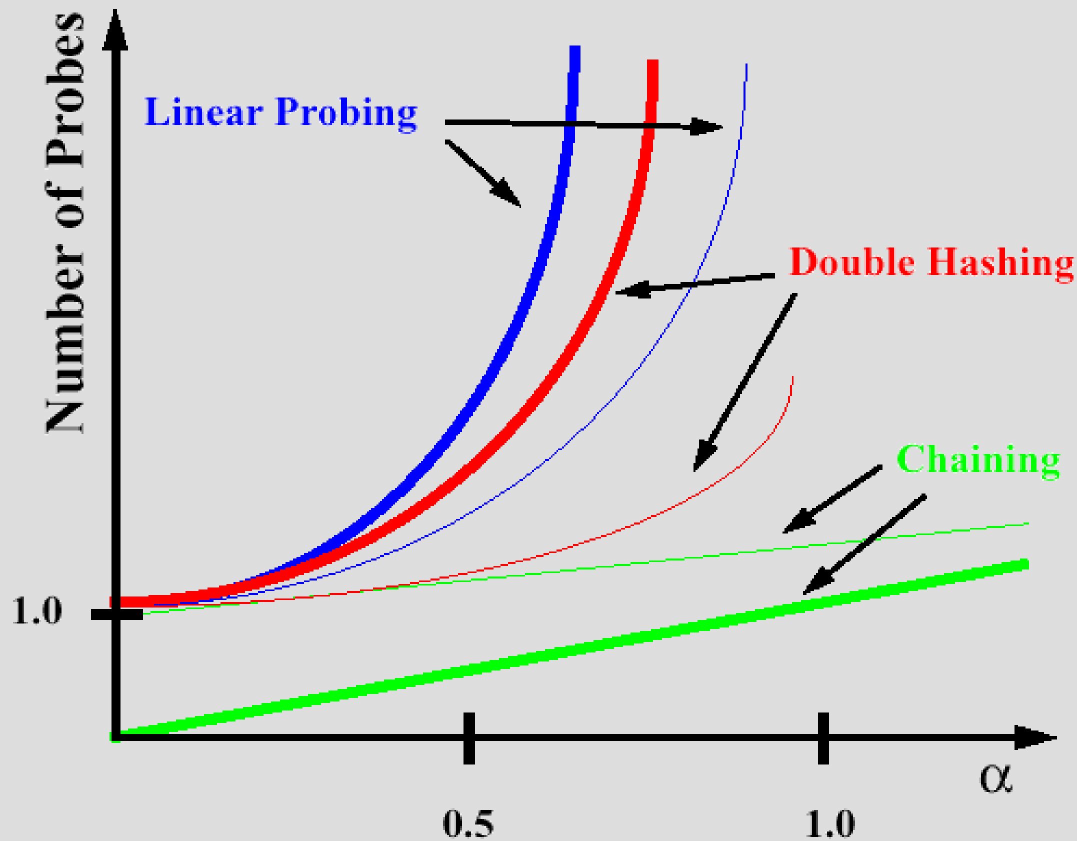
Expected Number of Probes

- Load factor α < 1 for probing
- Analysis of probing uses *uniform hashing* assumption – any permutation is equally likely

	Unsuccessful	Successful
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Probing	$O(\frac{1}{1-\alpha})$	$O(\frac{1}{\alpha}\ln\frac{1}{1-\alpha})$

- Chaining: 1 (α=0%), 1.5 (α=50%), 2 (α=100%), n (α=n)
- Probing, unsucc: 1.25 (α=20%), 2 (α=50%), 5 (α=80%), 10 (α=90%)
- Probing, succ: 0.28 (α=20%), 1.39 (α=50%), 2.01 (α=80%), 2.56 (α=90%)

Expected Number of Probes/2



Summary

- Hashing is very efficient
  (not obvious, probability theory).
- Its functionality is limited (printing elements sorted according to key is not supported).
- The size of the hash table
  may not be easy to determine.
- A hash table is not really
  a dynamic data structure.

Suggested exercises

- Implement a Hash Table with the different techniques
- With paper & pencil, draw the evolution of a hash table when inserting, deleting and searching for new element, with the different techniques
- See also exercises of CLRS

Next Part

- Graphs:
  - Representation in memory
  - Breadth-first search
  - Depth-first search
  - Topological sort