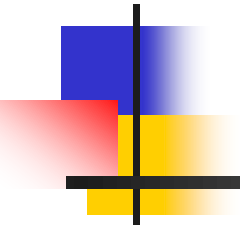
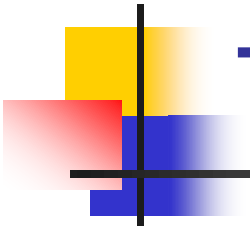
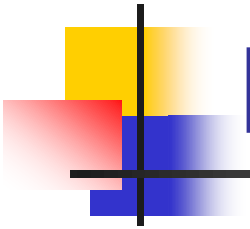
Algorithms and Data Structures Lecture VI





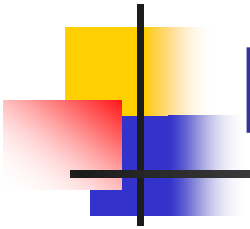
This Lecture

- Dictionaries or **how to search efficiently**
- Hashing
  - the concept
  - collision resolution
  - chooing a hash function
  - advanced collision resolution



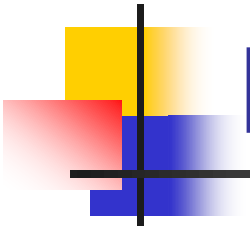
Dictionary

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



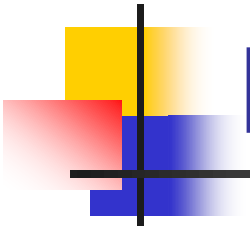
Dictionaries

- 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 wealth of additional information
    - including the current balance,
    - the name and address of the account holder, and
    - the history of deposits and withdrawals performed
  - an application wishing to operate on an account would have to provide the account number as a search **key**



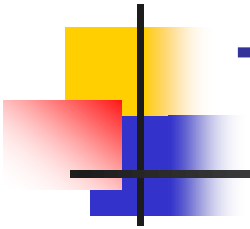
Dictionaries (2)

- Supporting order (methods *min, max, successor, predecessor*) is not required, thus it is enough that **keys are comparable for equality**



Dictionaries (3)

- Different data structures to realize dictionaries
  - arrays, linked lists (inefficient)
  - **Hash table** (used in Java...)
  - Binary trees
  - Red/Black trees
  - AVL trees
  - B-trees
- In Java:
  - java.util.Dictionary – abstract class
  - java.util.Map – interface



The Problem

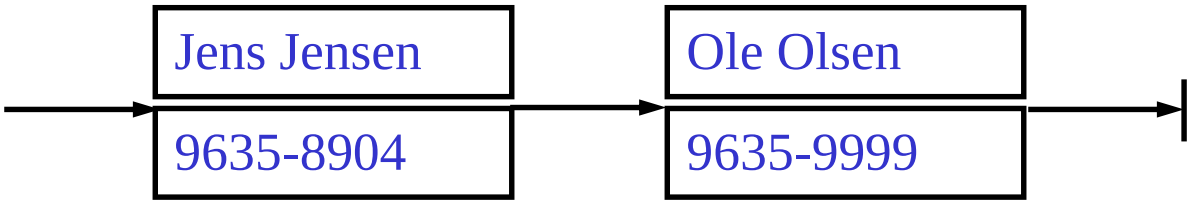
- RT&T is a large phone company, and they want to provide caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers range from 0 to r = 10^8 -1
  - want to do this as efficiently as possible

The Problem

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

(null)	(null)	Jens Jensen	(null)	(null)
0000-0000	0000-0000	9635-8904	0000-0000	0000-0000

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



Another Solution

- We can do better, with a **Hash table** -- O(1) expected time, O(*n*+*m*) space, where *m* is table size
- Like an array, but come up 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 array, and use that as an index
- Insert (9635-8904, Jens Jensen) into a hashed array with, say, five slots
  - 96358904 mod 5 = 4

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

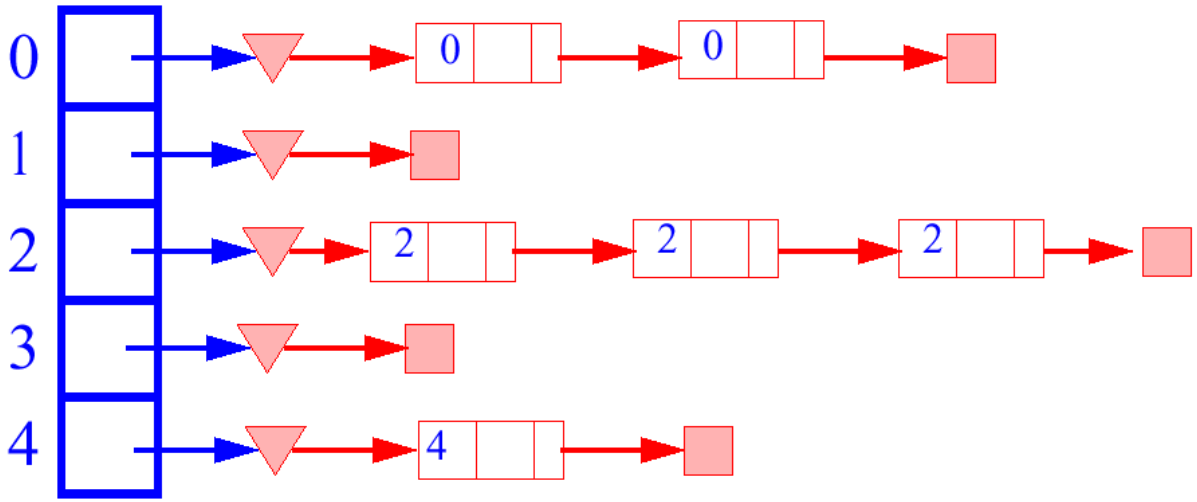
Another Solution (2)

- A lookup uses the same process: hash the query key, then check the array at that slot
- Insert (9635-8900, Leif Erikson)
- And insert (9635-8004, Knut Hermandsen).

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

Collision Resolution

- How to deal with two keys which hash to the same spot in the array?
- Use **chaining**
  - Set up an array of links (a **table**), indexed by the keys, to **lists** of items with the same key



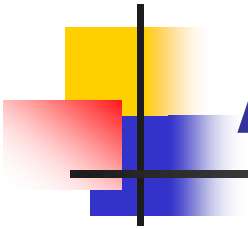
  - Most efficient (time-wise) collision resolution scheme

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 $T[0...m-1]$

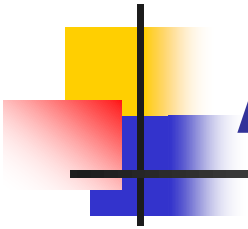
$$h : U \rightarrow \{0,1,...,m-1\}$$

- 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 $\alpha = n/m$
- Assume time to compute $h(k)$ is $\Theta(1)$



Analysis of Hashing (2)

- 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
- Unsuccessful search
  - element is not in the linked list
  - uniform hashing yields an average list length $\alpha = n/m$
  - expected number of elements to be examined α
  - search time $O(1+\alpha)$ (this includes computing the hash value)



Analysis of Hashing (3)

- Successful search
  - assume that a new element is inserted at the end of the linked list
  - upon insertion of the i-th element, the expected length of the list is (i-1)/m
  - in case of a successful search, the expected number of elements examined is 1 more that the number of elements examined when the sought-for element was inserted!

Analysis of Hashing (4)

- The expected number of elements examined is thus

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)$$

$$= 1+\frac{1}{nm}\cdot\frac{(n-1)n}{2}$$

$$= 1+\frac{n-1}{2m}$$

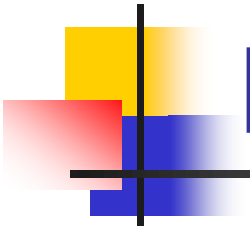
$$= 1+\frac{n}{2m}-\frac{1}{2m}$$

$$1+\frac{\alpha}{2}-\frac{1}{2m}$$

- Considering the time for computing the hash function, we obtain $\Theta(2+\alpha/2-1/2m) = \Theta(1+\alpha)$

Analysis of Hashing (5)

- 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 when the lists are doubly-linked

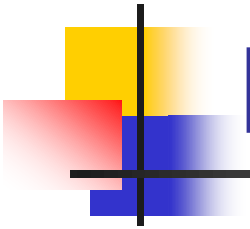


Hash Functions

- Need to choose a good hash function
  - quick to compute
  - distributes keys uniformly throughout the table
  - good hash functions are very rare – *birthday* paradox
- 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 $\boldsymbol{e}$ 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



HF: Division Method (2)

- Example 1
  - hash table for n = 2000 character strings
  - we don't mind examining 3 elements
  - m = 701
    - a prime near 2000/3
    - but not near any power of 2
- Further examples
  - m = 13
    - $h(3)$?
    - $h(12)$?
    - $h(13)$?
    - $h(47)$?

HF: Multiplication Method

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

HF: Multiplication Method(2)

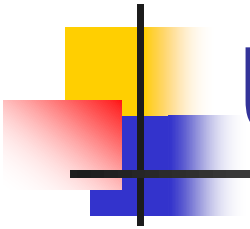
- Choice of *m* and *A*
  - value of *m* is not critical, typically use $m = 2^p$
  - optimal choice of *A* depends on the characteristics of the data
    - Knuth says use $A = \frac{\sqrt{5}-1}{2}$ (conjugate of the *golden ratio*) – *Fibonaci* hashing

HF: Multiplication Method (3)

- Assume 7-bit binary keys, $0 \leq k < 128$
- $m = 8 = 2^3$
- $A = .1011001 = 89/128$
- $k = 1101011$ (107)
- Using binary representation

$$\begin{array}{lll} & .1011001 & A \\ & 1101011 & k \\ \hline 1001010 & .0110011 & kA \\ & h(k) & \end{array}$$

- Thus, $h(k) = ?$

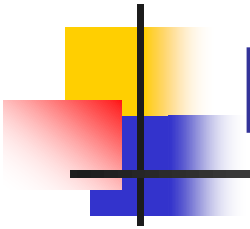


Universal Hashing

- For any choice of hash function, there exists a bad set of identifiers
- A malicious adversary could choose keys to be hashed such that all go into the same slot (bucket)
- Average retrieval time is $\Theta(n)$
- Solution
  - a random hash function
  - choose hash function independently of keys!
  - create a **set of hash functions *H***, from which *h* can be randomly selected

Universal Hashing (2)

- A collection H of hash functions is ***universal*** if for any randomly chosen f from H (and two keys k and l), $\Pr\{f(k) = f(l)\} \leq 1/m$
- Heuristics
  - learning from experience rather than analysis
    - find good Hash functions for a real-world set of keys through trial and error
  - practical rather than theoretical



More on Collisions

- A key is mapped to an already occupied table location
  - what to do?!?
- Use a collision handling technique
- We've seen *Chaining*
- Can also use *Open Addressing*
  - Probing
  - Double Hashing

Open Addressing

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

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

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

Open Adressing (2)

- Probe sequence for a given key k given by

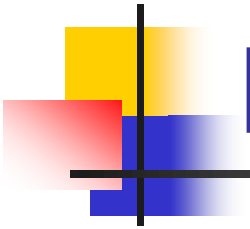
$\langle h(k,0), h(k,1), ..., h(k,m-1) \rangle$ - a permutation of $\langle 0,1,...,m-1 \rangle$

Linear Probing

- If the current location is used, try the next table location

```
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
```

- 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 walk along the table for a long time



Linear Probing

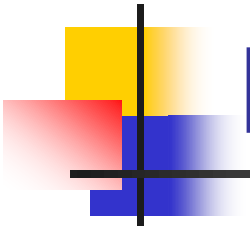
- A real pain to delete from
  - either mark the deleted slot
  - or fill in the slot by shifting some elements down
- Example
  - $h(k) = k \bmod 13$
  - insert keys: 18 41 22 44 59 32 31 73

Double Hashing

- Use two hash functions
- If *m* is prime, eventually will examine every position in the table

```
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
```

- Many of the same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing



Double Hashing (2)

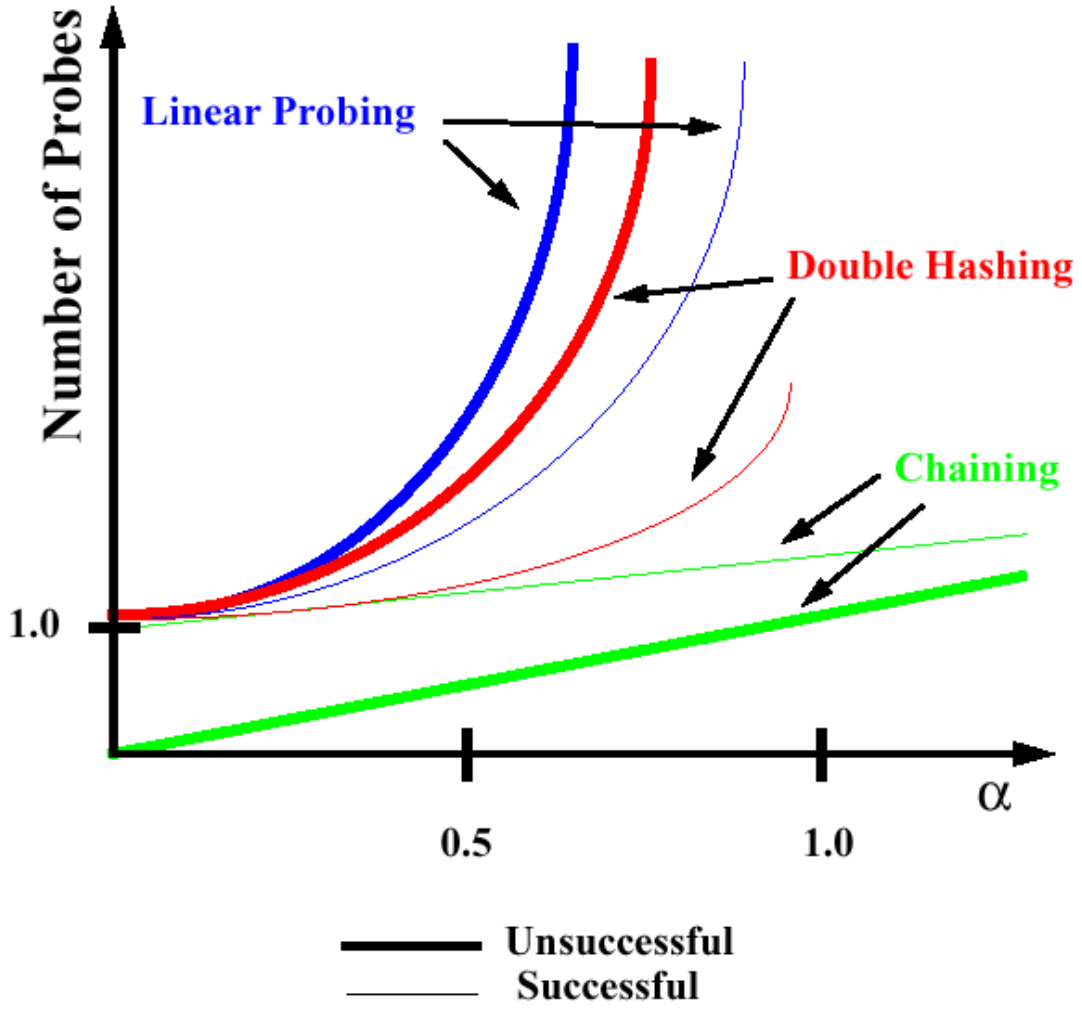
- *h2(k)* must be relative prime to *m*
- Example
  - $h1(k) = k \bmod 13$
  - $h2(k) = 8 - k \bmod 8$
  - insert keys: 18 41 22 44 59 32 31 73

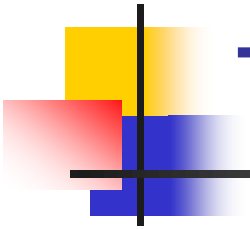
Expected Number of Probes

- Load factor $\alpha < 1$ for probing
- Analysis of probing uses *uniform hashing* assumpion – any permutation is equally likely
  - What about linear probing and double hashing?

	unsuccessful	successful
chaining	$O(1+\alpha)$	$O(1+\alpha)$
probing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

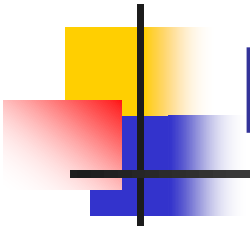
Expected Number of Probes (2)





The Hashing Class in Java

- java.util.Hashtable
- Constructor
  - public **Hashtable**(int size, float load);
- Accessor methods
  - public Object **put**(Object key, Object value):
    - maps the specified key to the specified value in this hashtable



Next lecture

- Binary Search Trees