# CSE 326: Data Structures
# Part 5
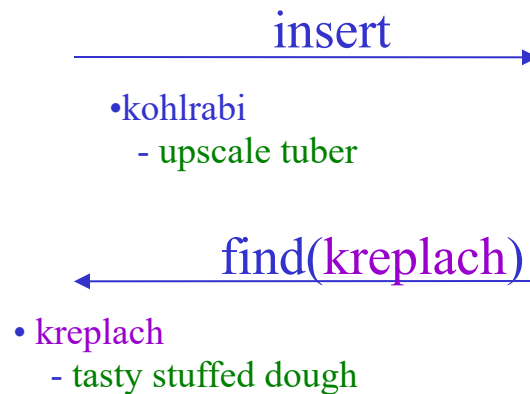# Hashing

Henry Kautz

Autumn 2002

# Dictionary & Search ADTs

- Operations
  - create
  - destroy
  - insert
  - find
  - delete

insert →

- kohlrabi
  - upscale tuber

← find(kreplach)

- kreplach
  - tasty stuffed dough

- kim chi
  - spicy cabbage
- kreplach
  - tasty stuffed dough
- kiwi
  - Australian fruit

- Dictionary:  Stores *values* associated with user-specified *keys*
  - keys may be any (homogenous) comparable type
  - values may be any (homogenous) type
  - implementation: data field is a struct with two parts
- Search ADT:  keys = values

# Implementations So Far

|  | unsorted list | sorted array | Trees<br>BST – average<br>AVL – worst case<br>splay – amortized | Array of size n where keys are $0,\ldots,n\text{-}1$ |
|---|---|---|---|---|
| insert | find+$\theta(1)$ | $\theta(n)$ | $\theta(\log n)$ | |
| find | $\theta(n)$ | $\theta(\log n)$ | $\theta(\log n)$ | |
| delete | find+$\theta(1)$ | $\theta(n)$ | $\theta(\log n)$ | |

# Hash Tables: Basic Idea

- Use a key (arbitrary string or number) to index directly into an array – O(1) time to access records
  - A["kreplach"] = "tasty stuffed dough"
  - Need a *hash function* to convert the key to an integer

|   | Key | Data |
|---|-----|------|
| 0 | kim chi | spicy cabbage |
| 1 | kreplach | tasty stuffed dough |
| 2 | kiwi | Australian fruit |

# Applications

- When log(*n*) is just too big…
    - Symbol tables in interpreters
    - **Real-time databases (in core or on disk)**
        - air traffic control
        - packet routing
- When associative memory is needed…
    - Dynamic programming
        - cache results of previous computation

        **f(x) →if ( Find(x) ) then Find(x) else f(x)**

        - Chess endgames
    - Many text processing applications – e.g. Web

        **$Status{$LastURL} = "visited";**
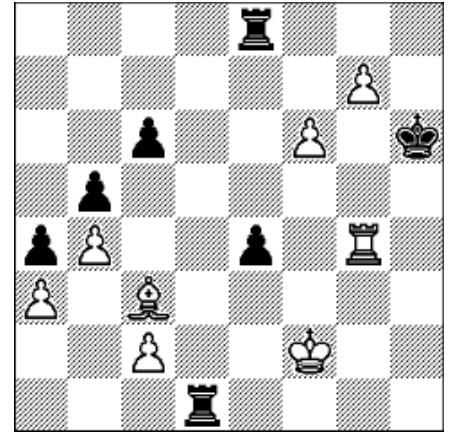
# How could you use hash tables to…

- Implement a linked list of unique elements?

- Create an index for a book?

- Convert a document to a Sparse Boolean Vector (where each index represents a different word)?

# Properties of Good Hash Functions

- Must return number 0, …, tablesize
- **Should be efficiently computable** – O(1) time
- Should not waste space unnecessarily
  - For every index, there is at least one key that hashes to it
  - Load factor lambda $\lambda$ = (number of keys / TableSize)
- Should minimize collisions
  = different keys hashing to same index

# Integer Keys

- Hash(x) = x % TableSize

- Good idea to make TableSize *prime*.  Why?

# Integer Keys

- Hash(x) = x % TableSize

- Good idea to make TableSize *prime*.  Why?
  - Because keys are typically not randomly distributed, but usually have some *pattern*
    - mostly even
    - mostly multiples of 10
    - in general: mostly multiples of some k
  - If k is a factor of TableSize, then only (TableSize/k) slots will ever be used!
  - Since the only factor of a prime number is itself, this phenomena only hurts in the (rare) case where k=TableSize

# Strings as Keys

- If keys are strings, can get an integer by adding up ASCII values of characters in *key*

```
for (i=0;i<key.length();i++)
        hashVal += key.charAt(i);
```

- **Problem 1**: What if *TableSize* is 10,000 and all keys are 8 or less characters long?

- **Problem 2**: What if keys often contain the same characters ("abc", "bca", etc.)?

# Hashing Strings

- Basic idea: consider string to be a integer (base 128):

  Hash("abc") = ('a'*$128^2$ + 'b'*$128^1$ + 'c') % TableSize

- Range of hash large, anagrams get different values

- **Problem:** although a char can hold 128 values (8 bits), only a subset of these values are commonly used (26 letters plus some special characters)

  - So just use a smaller "base"

  - Hash("abc") = ('a'*$32^2$ + 'b'*$32^1$ + 'c') % TableSize

# Making the String Hash Easy to Compute

- Horner's Rule

```
int hash(String s) {
  h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (s.keyAt(i) + h<<5) % tableSize;
  }
  return h;
}
```

*What is happening here???*

- Advantages:

# How Can You Hash…

- A set of values – (name, birthdate) ?

- An arbitrary pointer in C?

- An arbitrary reference to an object in Java?
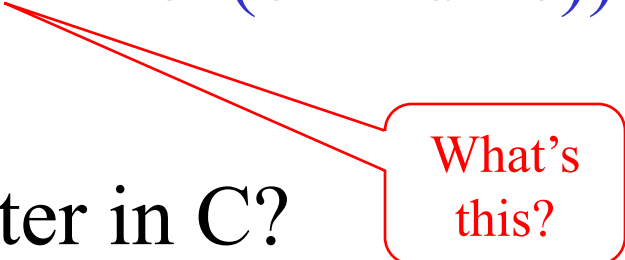
# How Can You Hash…

- A set of values – (name, birthdate) ?

    (Hash(name) ^ Hash(birthdate))% tablesize

    What's this?

- An arbitrary pointer in C?

    ((int)p) % tablesize

- An arbitrary reference to an object in Java?

    Hash(obj.toString())

    or just obj.hashCode() % tablesize

14

# Optimal Hash Function

- The best hash function would distribute keys as evenly as possible in the hash table
- "Simple uniform hashing"
  - Maps each key to a (fixed) random number
  - Idealized gold standard
  - Simple to analyze
  - Can be closely approximated by best hash functions

# Collisions and their Resolution

- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
- Two different methods for collision resolution:
  - **Separate Chaining:** Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot
  - **Closed Hashing (or *probing*):** search for empty slots using a second function and store item in first empty slot that is found

# A Rose by Any Other Name…

- Separate chaining = Open hashing

- Closed hashing = Open addressing

# Hashing with Separate Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered linked list (chain)

- Properties
  - performance degrades with length of chains
  - **$\lambda$ can be greater than 1**

What was $\lambda$??

$h(a) = h(d)$
$h(e) = h(b)$
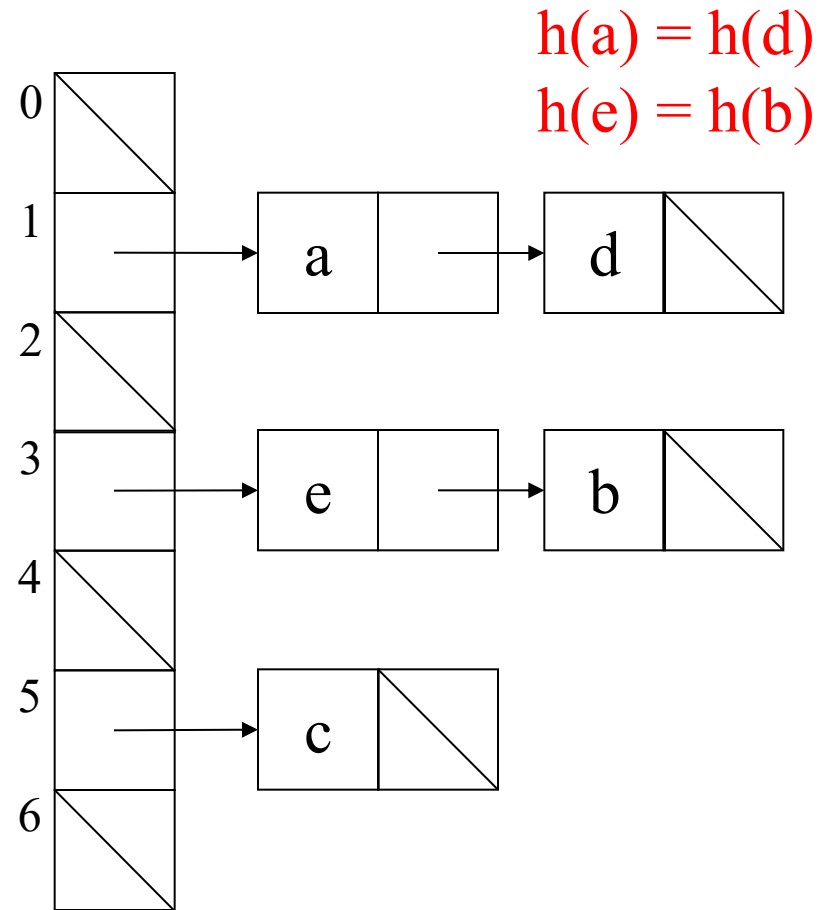
# Load Factor with Separate Chaining

- Search cost
  - unsuccessful search:


  - successful search:



- Optimal load factor:

# Load Factor with Separate Chaining

- Search cost (assuming simple uniform hashing)
  - unsuccessful search:

    Whole list – average length $\lambda$

  - successful search:

    Half the list – average length $\lambda/2+1$

- Optimal load factor:
  - Zero!  But between ½ and 1 is fast and makes good use of memory.

# Alternative Strategy: Closed Hashing

Problem with separate chaining:

**Memory consumed by pointers –**

**32 (or 64) bits per key!**

What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties
  - $\lambda \leq 1$
  - performance degrades with **difficulty of finding** right spot

h(a) = h(d)

h(e) = h(b)

| | |
|---|---|
| 0 | |
| 1 | a |
| 2 | d |
| 3 | e |
| 4 | b |
| 5 | c |
| 6 | |
| 21 | |
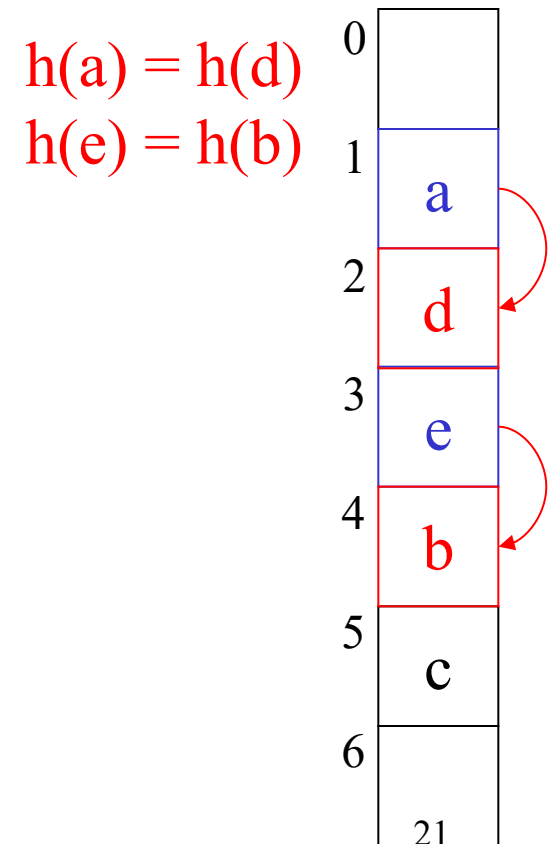
# Collision Resolution by Closed Hashing

- Given an item X, try cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$
- $h_i(X) = (Hash(X) + F(i))$ mod *TableSize*
  - Define $F(0) = 0$
- F is the *collision resolution* function. Some possibilities:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double Hashing: $F(i) = i * Hash_2(X)$

# Closed Hashing I: Linear Probing

- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
    - $h_i(X) = (Hash(X) + i) \mod \textit{TableSize}$   $(i = 0, 1, 2, \ldots)$
    - Compute hash value and increment it until a free cell is found

# Linear Probing Example

| insert(14) | insert(8) | insert(21) | insert(2) |
|:---:|:---:|:---:|:---:|
| $14\%7 = 0$ | $8\%7 = 1$ | $21\%7 = 0$ | $2\%7 = 2$ |

| | | | | | | |
|:---:|:---:|:---:|:---:|:---:|:---:|:---:|
| 0 | 14 | | | | | |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |

| | |
|:---:|:---:|
| 0 | 14 |
| 1 | 8 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

| | |
|:---:|:---:|
| 0 | 14 |
| 1 | 8 |
| 2 | 21 |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

| | |
|:---:|:---:|
| 0 | 14 |
| 1 | 8 |
| 2 | 21 |
| 3 | 2 |
| 4 | |
| 5 | |
| 6 | |

| 1 | 1 | 3 | 2 |
|:---:|:---:|:---:|:---:|

probes:

24

# Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)

- Very prone to cluster formation (as in our example)
  - If a key hashes *anywhere* into a cluster, finding a free cell involves going through the entire cluster – and making it grow!
  - *Primary clustering – clusters grow when keys hash to values close to each other*

- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of *distributing keys uniformly*

# Load Factor in Linear Probing

- For *any* λ < 1, linear probing will find an empty slot
- Search cost (assuming simple uniform hashing)
  - successful search:
  $$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$$

  - unsuccessful search:
  $$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

- Performance quickly degrades for λ > 1/2

# Optimal vs Linear

| load factor | successful | | unsuccessful | |
| --- | --- | --- | --- | --- |
| | optimal | linear | optimal | linear |
| 0.1 | 1.05 | 1.06 | 1.11 | 1.12 |
| 0.2 | 1.12 | 1.13 | 1.25 | 1.28 |
| 0.3 | 1.19 | 1.21 | 1.43 | 1.52 |
| 0.4 | 1.28 | 1.33 | 1.67 | 1.89 |
| 0.5 | 1.39 | 1.50 | 2.00 | 2.50 |
| 0.6 | 1.53 | 1.75 | 2.50 | 3.63 |
| 0.7 | 1.72 | 2.17 | 3.33 | 6.06 |
| 0.8 | 2.01 | 3.00 | 5.00 | 13.00 |
| 0.9 | 2.56 | 5.50 | 10.00 | 50.50 |

# Closed Hashing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by $i^2$ instead of $i$

- $h_i(X) = (Hash(X) + i^2)$ % *TableSize*

  h0(X) = Hash(X) % TableSize

  h1(X) = Hash(X) + 1 % TableSize

  h2(X) = Hash(X) + 4 % TableSize

  h3(X) = Hash(X) + 9 % TableSize

28

# Quadratic Probing Example

insert(14)    insert(8)    insert(21)    insert(2)
14%7 = 0      8%7 = 1      21%7 =0       2%7 = 2

| | insert(14) | insert(8) | insert(21) | insert(2) |
|---|---|---|---|---|
| 0 | 14 | 14 | 14 | 14 |
| 1 |  | 8 | 8 | 8 |
| 2 |  |  |  | 2 |
| 3 |  |  |  |  |
| 4 |  |  | 21 | 21 |
| 5 |  |  |  |  |
| 6 |  |  |  |  |

probes:    1         1         3         1

# Problem With Quadratic Probing

| insert(14) | insert(8) | insert(21) | insert(2) | insert(7) |
|:---:|:---:|:---:|:---:|:---:|
| $14\%7 = 0$ | $8\%7 = 1$ | $21\%7 = 0$ | $2\%7 = 2$ | $7\%7 = 0$ |

| | | | | |
|:---:|:---:|:---:|:---:|:---:|
| 0 **14** | 0 **14** | 0 **14** | 0 **14** | 0 **14** |
| 1 | 1 **8** | 1 **8** | 1 **8** | 1 **8** |
| 2 | 2 | 2 | 2 **2** | 2 **2** |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 **21** | 4 **21** | 4 **21** |
| 5 | 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 | 6 |

|  | 1 | 1 | 3 | 1 | ?? |
|:---|:---:|:---:|:---:|:---:|:---:|
| probes: | | | | | |

30

# Load Factor in Quadratic Probing

- **Theorem**: If TableSize is prime and $\lambda \leq \frac{1}{2}$, quadratic probing *will* find an empty slot; for greater $\lambda$, *might not*

- With load factors near $\frac{1}{2}$ the expected number of probes is empirically near *optimal* – no exact analysis known

- Don't get clustering from *similar* keys (**primary** clustering), still get clustering from *identical* keys (**secondary** clustering)

31

# Closed Hashing III: Double Hashing

- **Idea**: Spread out the search for an empty slot by using a second hash function
  - *No primary or secondary clustering*
- $h_i(X) = (Hash_1(X) + i*Hash_2(X)) \bmod \textit{TableSize}$
  for $i = 0, 1, 2, \ldots$
- Good choice of $Hash_2(X)$ can guarantee does not get "stuck" as long as $\lambda < 1$
  - Integer keys:
    $Hash_2(X) = R - (X \bmod R)$
    where R is a prime smaller than *TableSize*

# Double Hashing Example

insert(14)    insert(8)    insert(21)    insert(2)    insert(7)

$14\%7 = 0$    $8\%7 = 1$    $21\%7 = 0$    $2\%7 = 2$    $7\%7 = 0$

    $5-(21\%5)=4$           $5-(21\%5)=4$

| | Table 1 | | Table 2 | | Table 3 | | Table 4 | | Table 5 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 14 | 0 | 14 | 0 | 14 | 0 | 14 | 0 | 14 |
| 1 |    | 1 | 8  | 1 | 8  | 1 | 8  | 1 | 8  |
| 2 |    | 2 |    | 2 |    | 2 | 2  | 2 | 2  |
| 3 |    | 3 |    | 3 |    | 3 |    | 3 |    |
| 4 |    | 4 |    | 4 | 21 | 4 | 21 | 4 | 21 |
| 5 |    | 5 |    | 5 |    | 5 |    | 5 |    |
| 6 |    | 6 |    | 6 |    | 6 |    | 6 |    |

probes:     1        1        2        1        ??

33

# Double Hashing Example

insert(14)     insert(8)     insert(21)     insert(2)     insert(7)

$14\%7 = 0$   $8\%7 = 1$   $21\%7 = 0$   $2\%7 = 2$   $7\%7 = 0$

$5-(21\%5)=4$                          $5-(21\%5)=4$

| | insert(14) | insert(8) | insert(21) | insert(2) | insert(7) |
|---|---|---|---|---|---|
| 0 | 14 | 14 | 14 | 14 | 14 |
| 1 |  | 8 | 8 | 8 | 8 |
| 2 |  |  |  | 2 | 2 |
| 3 |  |  |  |  |  |
| 4 |  |  | 21 | 21 | 21 |
| 5 |  |  |  |  | 7 |
| 6 |  |  |  |  |  |

probes:         1             1             2             1             4

# Load Factor in Double Hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)

- Search cost approaches optimal (random re-hash):
  - successful search:

  $$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

  - unsuccessful search:

  $$\frac{1}{1-\lambda}$$

  **Note natural logarithm!**

- No primary clustering and no secondary clustering
- Still becomes costly as $\lambda$ nears 1.

# Deletion with Separate Chaining

*Why is this slide blank?*

36

# Deletion in Closed Hashing

delete(2)     find(7)

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 7 |
| 4 | |
| 5 | |
| 6 | |

Where is it?!

## What should we do instead?

# Lazy Deletion

delete(2)    find(7)

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 7 |
| 4 | |
| 5 | |
| 6 | |

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | # |
| 3 | 7 |
| 4 | |
| 5 | |
| 6 | |

Indicates deleted value:
if you find it, probe again

But *now* what is the problem?

# The Squished Pigeon Principle

- An insert using Closed Hashing *cannot* work with a load factor of 1 or more.
  - Quadratic probing can *fail* if $\lambda > \frac{1}{2}$
  - Linear probing and double hashing *slow* if $\lambda > \frac{1}{2}$
  - Lazy deletion never frees space
- Separate chaining becomes slow once $\lambda > 1$
  - Eventually becomes a linear search of long chains
- How can we relieve the pressure on the pigeons?

REHASH!

# Rehashing Example

Separate chaining

$h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
|   |   |   |   |   |

$\lambda=1$

25      37    83

52    98

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|----|
|   |   |   |   |   |   |   |   |   |   |    |

$\lambda=5/11$

25   37     83     52     98

# Rehashing without Stretching

- Suppose input is a mix of inserts and deletes
  - Never more than TableSize/2 active keys
  - Rehash when $\lambda=1$ (half the table must be deletions)
- Worst-case sequence:
  - T/2 inserts, T/2 deletes, T/2 inserts, Rehash,
    T/2 deletes, T/2 inserts, Rehash, …
- Rehashing at most doubles the amount of work – still O(1)

# Case Study

- Spelling dictionary
  - 50,000 words
  - static
  - arbitrary(ish) preprocessing time
- Goals
  - fast spell checking
  - minimal storage

- Practical notes
  - almost all searches are successful      Why?
  - words average about 8 characters in length
  - 50,000 words at 8 bytes/word is 400K
  - pointers are 4 bytes
  - there are *many* regularities in the structure of English words

43

# Solutions

- Solutions
  - sorted array + binary search
  - separate chaining
  - open addressing + linear probing

# Storage

- Assume words are strings and entries are pointers to strings
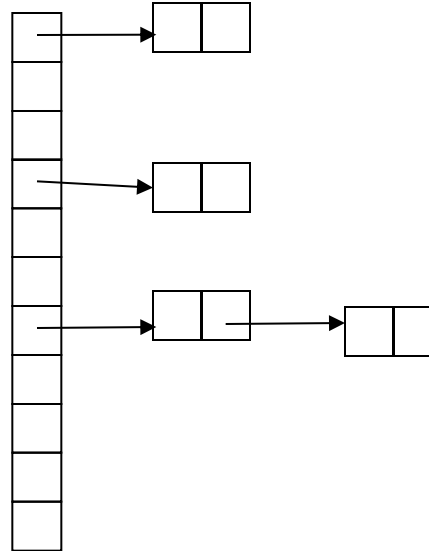
Array +
binary search

Separate chaining

Closed hashing

n pointers

table size + 2n pointers =
$n/\lambda + 2n$

$n/\lambda$ pointers 45

# Analysis

- Binary search
  - storage:  n pointers + words = 200K+400K = 600K
  - time:      $\log_2 n \leq 16$ probes per access, worst case
- Separate chaining - with $\lambda = 1$
  - storage:   $n/\lambda$ + 2n pointers + words = 200K+400K+400K = 1GB
  - time:      $1 + \lambda/2$ probes per access on average = 1.5
- Closed hashing - with $\lambda = 0.5$
  - storage:  $n/\lambda$ pointers + words = 400K + 400K = 800K
  - time: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ probes per access on average = 1.5

# Approximate Hashing

- Suppose we want to reduce the space requirements for a spelling checker, by accepting the risk of once in a while overlooking a misspelled word

- Ideas?

# A Random Hash…

- **Extensible hashing**
  - Hash tables for disk-based databases – minimizes number disk accesses

- Minimal perfect hash function
  - Hash a given set of $n$ keys into a table of size $n$ with no collisions
  - Might have to search large space of parameterized hash functions to find
  - Application: compilers

- One way hash functions
  - Used in cryptography
  - Hard (intractable) to *invert*: given just the hash value, recover the key

# Approximate Hashing

Strategy:

- – Do not store keys, just a bit indicating cell is in use

- – Keep $\lambda$ low so that it is unlikely that a misspelled word hashes to a cell that is in use

# Example

- 50,000 English words
- Table of 500,000 cells, each 1 bit
  - 8 bits per byte
- Total memory: 500K/8 = 62.5 K
  - versus 800 K separate chaining, 600 K open addressing
- Correctly spelled words will always hash to a used cell
- What is probability a misspelled word hashes to a used cell?

# Rough Error Calculation

- Suppose hash function is optimal - hash is a random number

- Load factor $\lambda \leq 0.1$
  - Lower if several correctly spelled words hash to the same cell

- So probability that a misspelled word hashes to a used cell is $\leq 10\%$

# Exact Error Calculation

- What is expected load factor?

$$\frac{\text{used cells}}{\text{table size}} = \frac{(\text{Probability a cell is used})(\text{table size})}{\text{table size}}$$

$= \text{Probability a cell is used} = 1 - (\text{Prob. cell not used})$

$=1-(\text{Prob. 1st word doesn't use cell})...(\text{Prob. last word doesn't use cell})$

$=1-((\text{table size - 1})/\text{table size})^{\text{number words}}$

$$=1-\left(\frac{499,999}{500,000}\right)^{50,000} \approx 0.095$$

# Puzzler

- Suppose you have a HUGE hash table, that you often need to re-initialize to "empty". How can you do this in small constant time, *regardless* of the size of the table?

# Databases

- A database is a set of records, each a tuple of values
  - E.g.: [ name, ss#, dept., salary ]
- How can we speed up queries that ask for all employees in a given department?
- How can we speed up queries that ask for all employees whose salary falls in a given range?