
# CSE 326: Data Structures Lecture #10 Hashing I

Henry Kautz
Winter 2002

## Midterm

- Friday February 8th
- Will cover everything through hash tables
- Weiss Chapters 1 – 5
- 50 minutes, in class
- You may bring one page of notes to refer to

## 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,…,n-1 |
|---|---|---|---|---|
| insert | O(1) | O(n) | O(log n) | |
| find | O(n) | O(log n) | O(log n) | |
| delete | O(n) | O(n) | O(log n) | |

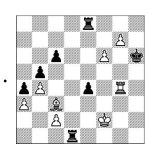
## Hash Tables: Basic Idea

- 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…

- Convert a document to a Sparse Boolean Vector?
- Create an index for a book?
- Implement a linked list?

## Properties of Good Hash Functions

- Must return number 0, …, tablesize
  - Easy: modulo arithmetic – always end in "`% 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*

```
while (*key != '\0')
    StringValue += *key++;
```

- **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_i + h<<5) % tableSize;
  }
  return h;
}
```

*What is happening here???*

- Advantages:

## How Can You Hash…

- A pointer?

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

## How Can You Hash…

- A pointer?
  ((int) p) % TableSize

- A set of values – (name, birthdate) ?
  (Hash1(name) + Hash2(birthdate)) % TableSize

## 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
  - **Open addressing (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

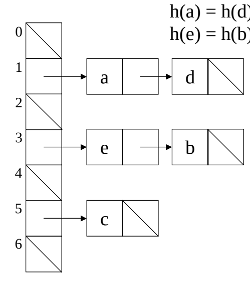
- Open addressing = Closed hashing

## 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
  - **λ can be greater than 1**

What was **λ**??

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
  - unsuccessful search:
    Whole list – average length **λ**
  - successful search:
    Half the list – average length **λ/2+1**
- Optimal load factor:
  - Zero! But between ½ and 1 is fast and makes good use of memory.

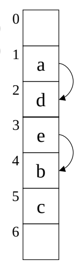
## Alternative Strategy: Open Addressing

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
  - **λ ≤ 1**
  - performance degrades with **difficulty of finding** right spot

h(a) = h(d)
h(e) = h(b)




## Question to Think About for Monday

- What is an application where it is a good idea to use open addressing and *not* do probing – you just *allow* collisions to occur?

## Collision Resolution by Open Addressing

- 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. Three possibilities:
  - Linear: F(i) = i
  - Quadratic: $F(i) = i^2$
  - Double Hashing: $F(i) = i \cdot Hash_2(X)$

## Open Addressing 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 *TableSize* (i = 0, 1, 2, …)
  - Compute hash value and increment it until a free cell is found

## Linear Probing Example

| | insert(14) 14%7 = 0 | insert(8) 8%7 = 1 | insert(21) 21%7 =0 | insert(2) 2%7 = 2 |
|---|---|---|---|---|
| 0 | 14 | 14 | 14 | 14 |
| 1 | | 8 | 8 | 8 |
| 2 | | | 21 | 12 |
| 3 | | | | 2 |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| probes: | 1 | 1 | 3 | 2 |

## 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* $\lambda < 1$, linear probing will find an empty slot
- Search cost (for large table sizes)
  - 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 $\lambda > 1/2$

## Open Addressing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by $i^2$ instead of i
- $h_i(X) = (\text{Hash}(X) + i^2)$ % *TableSize*

  h0(X) = Hash(X) % TableSize

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

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

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

## Quadratic Probing Example

| | insert(14) 14%7 = 0 | insert(7) 8%7 = 1 | insert(21) 21%7 =0 | insert(2) 2%7 = 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) 14%7 = 0 | insert(8) 8%7 = 1 | insert(21) 21%7 =0 | insert(2) 2%7 = 2 | insert(7) 7%7 = 0 |
|---|---|---|---|---|---|
| 0 | 14 | 14 | 14 | 14 | 14 |
| 1 | | 8 | 8 | 8 | 8 |
| 2 | | | | 2 | 2 |
| 3 | | | | | |
| 4 | | | 21 | 21 | 21 |
| 5 | | | | | |
| 6 | | | | | |
| probes: | 1 | 1 | 3 | 1 | ?? |

## Load Factor in Quadratic Probing

- **Theorem**: If TableSize is prime and $\lambda \le ½$, quadratic probing *will* find an empty slot; for greater $\lambda$, *might not*
- With load factors near ½ the expected number of probes is about 1.5
- Don't get clustering from *similar* keys (primary clustering), still get clustering from *identical* keys (secondary clustering)

## Monday

- Double hashing
- Deletion and rehashing
- Analysis of memory use
- Universal hash functions
- Perfect hashing
- and answer to the PUZZLER: *What is an application where it is a good idea to use open addressing and not do probing – you just allow collisions to occur?*