CSE 373: Hash functions and open addressing

Michael Lee

Wednesday, Jan 24, 2018

Warmup

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a linked list where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a), (5, b), (11, a), (7, d), (12, e), (17, f), (1, g), (25, h)

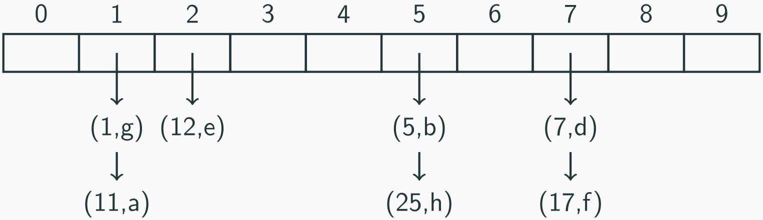
0	1	2	3	4	5	6	7	8	9

Warmup

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a linked list where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a), (5, b), (11, a), (7, d), (12, e), (17, f), (1, g), (25, h)



Announcements

Written HW 1 due tonight at 11:30pm

PSA:

- For questions involving math, make sure it's easy for us to follow your work
  - Don't just spit out equations without context, add some text to (briefly) explain what you're doing
  - Neatly label or circle your final answer
- Make sure you're submitting to the right place on Canvas

Announcements

Project 2 released, due Wed Jan 24

- Partner selection due Thursday

  Can work with same partner or different one

Announcements

Project 2 released, due Wed Jan 24

- Partner selection due Thursday

  Can work with same partner or different one
- About project
  - Bulk of project is spent implementing a hash table, using separate chaining
  - Will need to add an iterator to ArrayDictionary and your hash table
  - Implementing iterator for hash table may be tricky, don't leave it to the last moment

Midterm

Core details

Midterm

Core details

Times:

- Midterm on Friday, Feb 2, in-class
- Will last 80 minutes (3:30 to 4:50)

Midterm

Core details

Times:

- Midterm on Friday, Feb 2, in-class
- Will last 80 minutes (3:30 to 4:50)

Review sessions

- Monday, Jan 29: Gowen 201, 4:30 to 6:30
- Tuesday, Jan 30: Gowen 201, 4:30 to 6:30

Midterm

Midterm topics

Midterm

Midterm topics

Full list of topics available on course website now. Summary:

- Basic data structures (stacks, queues, list)
- Asymptotic analysis, modeling code
- Trees (BSTs and AVL trees)
- Hash tables
- Systems and B-Trees (on a high-level)

Midterm

Midterm topics

Full list of topics available on course website now. Summary:

- Basic data structures (stacks, queues, list)
- Asymptotic analysis, modeling code
- Trees (BSTs and AVL trees)
- Hash tables
- Systems and B-Trees (on a high-level)

Topics NOT covered on the midterm

- Finding the closed form of summations or recurrences
- Sorting
- Heaps
- Anything about Java (generics, interfaces, junit, eclipse, etc)

Midterm

Practice

- Past CSE 373 midterms available on course website
- Past sections
- Questions on written homework 1 are representative of what will appear on midterm

Hash functions

Hash function

A hash function is a mapping from the key set U to an integer.

Hash functions

Hash function

A hash function is a mapping from the key set U to an integer.

Or, in other words, a function that turns the input into an integer in some way.

How do we use a hash function?

1. We receive a key

How do we use a hash function?

1. We receive a key
2. We run the hash function to get some integer

How do we use a hash function?

1. We receive a key
2. We run the hash function to get some integer
3. We do the same thing we did for IntegerDictionary

Analyzing hash function

Exercise: let's convert a string into an integer.

Analyzing hash function

Exercise: let's convert a string into an integer.

What we have:

```
public class OurString {
    char[] chars;
    int size;

    // etc...
}
```

Analyzing hash function

Exercise: let's convert a string into an integer.

What we have:

```
public class OurString {
    char[] chars;
    int size;

    // etc...
}
```

Our goal:

```
int hashCode(str)
    // What goes here?
```

Analyzing hash functions

In math: $h(s) = 1$

In pseudocode:

```
int hashCode(str)
    return 1
```

Analyzing hash functions

In math: $h(s) = 1$

In pseudocode:

```
int hashCode(str)
    return 1
```

Good idea? Bad idea?

Analyzing hash functions

In math: $h(s) = 1$

In pseudocode:

```
int hashCode(str)
    return 1
```

Bad idea: Every string has same hash code! Everything collides!

(But hey, at least it's fast...)

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} s_i$

In pseudocode:

```
int hashCode(str)
    int out = 0
    for (char c : str.chars) {
        // Use ASCII value of char
        out += c
    return out
```

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} s_i$

In pseudocode:

```
int hashCode(str)
    int out = 0
    for (char c : str.chars) {
        // Use ASCII value of char
        out += c
    return out
```

Good idea? Bad idea?

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} s_i$

In pseudocode:

```
int hashCode(str)
    int out = 0
    for (char c : str.chars) {
        // Use ASCII value of char
        out += c
    return out
```

Better but not ideal: Still too many collisions! Ex: "baker" and "brake", and "break" all have same hash code!

Runtime: still pretty decent, relatively speaking

Insight: can we use character positions somehow?

Analyzing hash functions

In math: $h(s) = 2^{s_0} \cdot 3^{s_1} \cdot 5^{s_2} \cdot 7^{s_3} \cdot 11^{s_4} \cdots$

In pseudocode:

```
int hashCode(str)
    int out = 1
    for (char c : str.chars)
        int nextPrime = get next prime number
        out *= Math.pow(nextPrime, c)
    return out
```

Analyzing hash functions

In math: $h(s) = 2^{s_0} \cdot 3^{s_1} \cdot 5^{s_2} \cdot 7^{s_3} \cdot 11^{s_4} \cdots$

In pseudocode:

```
int hashCode(str)
    int out = 1
    for (char c : str.chars)
        int nextPrime = get next prime number
        out *= Math.pow(nextPrime, c)
    return out
```

Good idea? Bad idea?

Analyzing hash functions

In math: $h(s) = 2^{s_0} \cdot 3^{s_1} \cdot 5^{s_2} \cdot 7^{s_3} \cdot 11^{s_4} \cdots$

In pseudocode:

```
int hashCode(str)
    int out = 1
    for (char c : str.chars)
        int nextPrime = get next prime number
        out *= Math.pow(nextPrime, c)
    return out
```

Not ideal: Hideously expensive, creates gigantic integers

(But hey, at least every string maps to a unique int!)

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} 31^i \cdot s_i$

In code:

```
int hashCode(str)
    int accum = 1
    int out = 0
    for (char c : s.chars)
        out += accum * c
        accum *= 31
    return out
```

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} 31^i \cdot s_i$

In code:

```
int hashCode(str)
    int accum = 1
    int out = 0
    for (char c : s.chars)
        out += accum * c
        accum *= 31
    return out
```

Good idea? Bad idea?

Analyzing hash functions

In math: $h(s) = \sum_{i=0}^{|s|-1} 31^i \cdot s_i$

In code:

```
int hashCode(str)
    int accum = 1
    int out = 0
    for (char c : s.chars)
        out += accum * c
        accum *= 31
    return out
```

Good idea: Uses both character values and positions.

Strikes good balance between efficiency and reducing collisions.

(Why use 31? People tried a bunch of different strategies, and this one seemed to work well "in practice")

Hash functions

So, what does a good hash function look like?

Using hash functions inside dictionaries: useful properties

A hash function that is intended to be used for a dictionary should ideally have the following properties:

- **Low collision rate:**
  The hash of two different inputs should usually be different.
  We want to *minimize collisions* as much as possible.

Hash functions

So, what does a good hash function look like?

Using hash functions inside dictionaries: useful properties

A hash function that is intended to be used for a dictionary should ideally have the following properties:

- **Low collision rate:**
  The hash of two different inputs should usually be different.
  We want to *minimize collisions* as much as possible.
- **Uniform distribution of outputs:**
  In Java, there are 2^{32} 32-bit ints. So, the probability that the hash function returns any individual int should be $\frac{1}{2^{32}}$.

Hash functions

So, what does a good hash function look like?

Using hash functions inside dictionaries: useful properties

A hash function that is intended to be used for a dictionary should ideally have the following properties:

- **Low collision rate:**
  The hash of two different inputs should usually be different.
  We want to *minimize collisions* as much as possible.
- **Uniform distribution of outputs:**
  In Java, there are 2^{32} 32-bit ints. So, the probability that the hash function returns any individual int should be $\frac{1}{2^{32}}$.
- **Low computational cost:**
  We will be computing the hash function a lot, so we need it to be very easy to compute.

Client vs implementor

Who implements the hash function? The client, or the dictionary?

Client vs implementor

Who implements the hash function? The client, or the dictionary?

Client responsibilities

- Responsible for implementing a "good" hash function.
- The hash function avoids "wasting" information in the key or the output bits while still being "fast".

Client vs implementor

Who implements the hash function? The client, or the dictionary?

Client responsibilities

- Responsible for implementing a "good" hash function.
- The hash function avoids "wasting" information in the key or the output bits while still being "fast".

Dictionary/implementor responsibilities

- Responsible for calling the hash function
- Responsible for managing the internal array
- Responsible for keeping track of collisions

A Java interlude...

So, how does this work in Java?

A Java interlude...

So, how does this work in Java?

Every object has a default `equals` and `hashCode` implementation.

Override these two methods.

A Java interlude...

So, how does this work in Java?

Every object has a default `equals` and hashCode implementation.

Override these two methods.

Important invariants

When implementing hashCode, you MUST respect these invariants!

- IF you implement `hashCode(...)`,
  THEN you MUST also implement `equals(...)`
- IF `a.equals(b)`,
  THEN you MUST make sure that `a.hashCode() == b.hashCode()`

Handling multiple fields

What if an object has multiple fields?

Handling multiple fields

What if an object has multiple fields?

General considerations:

- Trade-off: hashing time vs collision avoidance
- Are some fields redundant? Do you need to hash all of them?

Handling multiple fields

What if an object has multiple fields?

General considerations:

- Trade-off: hashing time vs collision avoidance
- Are some fields redundant? Do you need to hash all of them?

Tips for creating hashes

- Use all 32 bits (including negative numbers!)
- Use different overlapping bits for different parts of the hash
- If keys are known ahead of time, choose a perfect hash
- Use expertise of others: consult books, have your IDE auto-generate a hash function...

Handling collisions

Insight:

The majority of our time is spent handling collisions

Handling collisions

Insight:

The majority of our time is spent handling collisions

Our strategy so far:

- Design a good hash function to minimize chance of collision
- If we do have a collision, store both in a “bucket”

Handling collisions

Insight:

The majority of our time is spent handling collisions

Our strategy so far:

- Design a good hash function to minimize chance of collision
- If we do have a collision, store both in a "bucket"

Are there other strategies for storing collisions?

Yes: something called **open addressing**

Open addressing

Open addressing

Open addressing is a kind of collision resolution strategy that resolves collisions by chosing a different location when the natural choice is full.

Open addressing: linear probing

Exercise: assume internal capacity of 10, insert the following keys:

1, 5, 11, 7, 12, 17, 6, 25

0	1	2	3	4	5	6	7	8	9

Open addressing: linear probing

Exercise: assume internal capacity of 10, insert the following keys:

1, 5, 11, 7, 12, 17, 6, 25

0	1	2	3	4	5	6	7	8	9
	1	11	12		5	6	7	17	25

Open addressing: linear probing

Strategy: Linear probing

If we collide, checking each next element until we find an open slot.

So, $h'(k, i) = (h(k) + i) \bmod T$, where T is the table size

```
i = 0
while (index in use)
    try (hash(key) + i) % array.length
    i += 1
```