CSC207.01 2014S, Class 42: Implementing Dictionaries with Hash Tables

Overview

- Preliminaries.
  - Upcoming work.
  - Admin.
  - Questions.
- An introduction to hash tables.
- Hash functions.
- An exercise.
- Handling collisions.
- Hashing in Java.
- Handling removal.

Preliminaries

- The answer is 42. (The question is "What class is it?")

Upcoming Work

- Reading for tomorrow: Hash Tables
- No writeup today. (No lab today.)
- Part 1 of the project is due Wednesday night.
  - I’ll take questions after the introductory stuff.

Admin

- Happy spring!
- Review session tonight - JSON.
- I planned today well. No need to use MathLAN.

Extra Credit

- Convo Wednesday: Philip Deloria, Professor of History and Native American Studies, University of Michigan.
- CS extra Thursday: Charlie Eddy on Kinect.
- CS Pub Night Thursday.
- CS table Friday: Big Data (Stone leads).
- Get and wear one of the 1 in 4 shirts.

- Iowater project April 19 - Tag drains. Mail iowater@grinnell.edu for details.
- http://www.strikingly.com/pioneerweekend

Questions

Do I have to worry about whitespace characters, such as tabs and newline?

> Nope.

Is the empty string the empty string?

> Yes.

Can we crash on invalid strings, such as `"\"`?

> Yes.

Do we have to assume that JSON represents an array or string?

> I'd prefer that you assume that JSON can represent any type. But if you can find it in the ECMA standard, you can assume it represents only an array or string.

An introduction to hash tables

- One of the most popular implementations of dictionaries.
- Observation: Arrays are fast - O(1)
- Conceptually, dictionaries should be just as fast.
- Strategy: Use arrays, write function that turns keys into indices in the array.
  - If we're lucky, no two keys will have the same index.
  - We call this function a "hash" function.
- To put an object, put the key value pair at index hash(key) % table-size

  ```
  this.values[hash(key) % this.values.size] = new KeyValuePair(key,value);
  ```

- To get an object, get the key/value pair at index hash(key) % table-size

  ```
  return this.values[hash(key) % this.values.size].value;
  ```

- Note: We typically put a lot of blank space in the table in order to achieve efficiency.

Hash function

- Map keys to values
- Given the same key, should give the same value
- Given different keys, should give different values
- Impossible to achieve the second in general: There are generally more values of any type than there

are Java integers.

- Design hash function so that different keys are *unlikely* to have the same value.
- The hard parts of hash tables
    - Writing good hash functions
    - Dealing with duplicate hash values

An exercise

Letter values

```
    A: 1   F: 6   K: 11  P: 16  U: 21  Z: 26
    B: 2   G: 7   L: 12  Q: 17  V: 22
    C: 3   H: 8   M: 13  R: 18  W: 23
    D: 4   I: 9   N: 14  S: 19  X: 24
    E: 5   J: 10  O: 15  T: 20  Y: 25
```

Hash table

```
    0:              10:             20:
    1:              11:             21:
    2: erin (32)    12: vasi (42)   22:
    3: sam (33)     13:             23:
    4: shen (32)    14:             24:
    5:              15:             25: helen (25)
    6:              16:             26: graeme (26)
    7:              17: nora (37)   27:
    8:              18: alex (18)   28:
    9:              19:             29:

    Additional: shen (32), fengyuan (25), madeleine (18)
```

```
sam: 19 + 1 + 13 = 33, goes in cell 3
```

Handling collisions

- Strategy 1: Instead of putting single key/value pairs in each cell, make each cell a bucket that holds multiple key/value pairs. Chaining. Most frequently with association lists.
- Strategy 2: rehash - Find another location for the value.
    - Use another function
    - Look in the next cell = hash(key) + 1
    - What if that’s full. Look in the next cell.
    - The "add 1" is a form of what is called "linear probing"
    - Most linear probing uses some other offset, which should be relatively prime to the size of the table.
    - There’s also quadratic probing n -> n + 1 -> n + 1 + 4 -> n + 1 + 4 + 9 -> n + 1 + 4 + 9 + 16
    - Some people use a computed offset

Question: How do you implement get when you use the "rehash" approach?

- Hash the key and look at the given spot.
    - Keys match. Done.
    - Keys don’t match: Follow the rehashing steps until you find a matching key or find an empty space.

Question: Is this still constant?

- It’s *expected* constant.
- We can rebuild the table if the ratio of values to size gets too large, or if we get a chain that’s too large or ...
- If we expand the table, we typically need to move everything into a different place in the new table (and have to compute the table).

Hashing in Java

- java.util.Hastable
- java.util.HashMap
- Also a language decision: Hash functions are expected for every class.
    - That is, implement `int hashCode()`

- Sample hash function

```
public class Rational
{
  BigInteger numerator;
  BitInteger denominator;


  public int hashCode()
  {
    return numerator.hashCode() * 2 + denominator.hashCode() * 3;
  } // hashCode()
```

Handling removal

How do we get rid of sam?

```
0:              10:             20:
1:              11:             21:
2: erin (32)    12: vasi (42)   22:
3: shen (32)    13:             23:
4: leon (32)    14:             24:
5:              15:             25: helen (25)
6:              16:             26: graeme (26)
7:              17:             27: helena (25)
8:              18: alex (18)   28:
9:              19:             29:
```

- Just clearing the cell doesn’t work.
- Mark it as deleted, but don’t actually delete it.
    - Does that change how you do `put`? If you hit one of these "deleted" cells, use the cell.
- Probe further and move backwards. And then recurse
    - And you have to do this carefully. For example, deleting helen does not mean that we have to move graeme (but we do have to move helena)