Indexed Search

Motivation: Wordlists
Data Structure: Hash Table
Hash Functions
Examples and Experiments
Google

Reading: SoC Chapter 8

Spell Check

- When the "spell check" option is turned on an application will indicate misspelled words as soon as you type them
- The system uses a wordlist
  - like a dictionary but it doesn't have any definitions
  - simply a list of correctly spelled words
- Each time you end a word the application searches the wordlist

A Thought Experiment

- As efficient as binary search is, it is still (theoretically) possible to do a search using fewer comparisons
- Suppose there is a special function $f$
  - the input to $f$ is any one of $n$ words
  - the output is an integer between 0 and $n-1$
  - the function has a very important property: each word maps to a different number
- If there is such a function we can use it to store data in an array
  - save word $w$ at location $f(w)$

$$f(w) \rightarrow n$$

An array for the 234,936 words from Webster's 2nd
A Thought Experiment

- We can use the array and this special function to implement a spell checker
  - put every word \( w \) at location \( f(w) \)
  - To see if a string \( s \) is a correctly spelled word just look in location \( f(s) \)
    - if the word there is \( s \) we know it is spelled correctly
      \[ f(\text{favorite}) = 161802 \]
    - if we find a different word \( w \) then \( s \) is not a word in the wordlist
      \[ f(\text{favourite}) = 90458 \]
- This “search algorithm” requires only one comparison

Hash Function

- Even though we can’t define a perfect function that assigns each word to a unique location we can come close
- A hash function is a function that maps strings to integers
  - it is possible to define a “perfect” hash function for small sets of words
  - most practical applications define a function that scatters words randomly throughout an array
  - use a large array, intentionally leave some locations empty

A Thought Experiment

- The function \( f \) does not have to order words according to any particular rule
  - it could order them alphabetically
  - the words might also be ordered according to their length
  - the order can even be completely random
- All we care about is that this function maps every word to a unique integer

\[ f(w) \rightarrow n \]

Hash Function

- The phrase “hash function” comes from the way these functions are typically implemented
  - an input word is chopped into small pieces
  - each piece is converted to an integer
  - these numbers are then reassembled to make the location for a word

\[ (5 \times 26^7 + \ldots + 19 \times 26^1 + 4) \mod n \]
### ASCII Characters

- The characters in strings are based on a numeric code
- The most common code: ASCII
  - pronounced “ass-key”
  - acronym for “American Standard Code for Information Interchange”
- Original standard had 128 entries (upper and lower case, digits, punctuation)
  - modern extended ASCII has 256 symbols (math, accented letters, etc)

From the Wikipedia page for ASCII

### Collisions

- Even though the array has extra room, it is inevitable the function will want to put more than one word in the same location
  - a collision occurs when two words hash to the same location
- Example (using the hash function explained later in these slides):
  - read words from Webster’s 2nd from a file, insert into an empty table
  - “madding” inserted in the table at location 565987
  - “pedagoguish” hashes to same location $f(\text{madding}) = f(\text{pedagoguish}) = 565987$

### ASCII Characters

- Accessing a single element of a string in Ruby returns an ASCII code
  ```ruby
  s = "hello"
  s[0]  # => 104
  s[1]  # => 101
  ```

Later we’ll see a method that converts letter codes to numbers between 0 and 25

Later we’ll see a method that converts letter codes to numbers between 0 and 25
Hash Tables

- A table is similar to an array
  - a technique for organizing a collection of data
  - access rows of a table according to an index value
  - The difference: each row in a table can have multiple values
  - Rows in a table are often called records

<table>
<thead>
<tr>
<th>Name</th>
<th>Party</th>
<th>Precinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baggins, Frodo</td>
<td>Dem</td>
<td>101317</td>
</tr>
<tr>
<td>Flintstone, Fred</td>
<td>Dem</td>
<td>101237</td>
</tr>
<tr>
<td>Freakowski, Freddy</td>
<td>Ind</td>
<td>101317</td>
</tr>
<tr>
<td>Krueger, Freddy</td>
<td>Rep</td>
<td>101317</td>
</tr>
<tr>
<td>Mercury, Freddy</td>
<td>Dem</td>
<td>101317</td>
</tr>
<tr>
<td>Mertz, Fred</td>
<td>Rep</td>
<td>101237</td>
</tr>
<tr>
<td>Weasley, Fred</td>
<td>Dem</td>
<td>101317</td>
</tr>
</tbody>
</table>

Hash Tables

- A hash table is a table where the items are stored using a hash function
  - choose a column to be the key
  - key values should be unique (different in each record)
  - store records in the row determined by \( f(key) \)

Example: use voter name as the key

\[ f(mertz) = 5 \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Party</th>
<th>Precinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weasley, Fred</td>
<td>Dem</td>
<td>101317</td>
</tr>
<tr>
<td>Freakowski, Freddy</td>
<td>Ind</td>
<td>101317</td>
</tr>
<tr>
<td>Mercury, Freddy</td>
<td>Dem</td>
<td>101237</td>
</tr>
<tr>
<td>Mertz, Fred</td>
<td>Rep</td>
<td>101237</td>
</tr>
<tr>
<td>Krueger, Freddy</td>
<td>Rep</td>
<td>101317</td>
</tr>
</tbody>
</table>

Indexes

- A common type of table in the “real world” is the index for a book
  - records are words or phrases and an associated set of page numbers
  - to find something in the book look it up in the index, then go to the indicated page
  - we don’t scan the book page by page to find information

If the book is a PDF just click the page number....

Go to page 40 to read about creating an array

Indexing the Web

- Google uses a huge index to report results of web searches
Project Outline

- This week’s project will be on hash functions
  - techniques for mapping strings to integers
  - properties of different types of functions
- To simplify things we won’t consider tables and how to manage the extra information
  - we’ll just look at how to organize an “index” and how to resolve collisions
- We’ll run some experiments on large wordlists
- We’ll also introduce some important new programming language concepts that are used in the implementation

Types of Data in Ruby

- The programs I put together for the lab this week define a new data type
- To run an experiment make a hash table object:
  - `load "HashTable.rb"`
  - `true`
  - `t = HashTable.new(100)`
  - `#<HashTable:0x65360>`
- Then do things with this object:
  - `t.insert("apple")`
  - `"apple"`
  - `t.print`
    - `70: ["apple"]`

Data Types in Ruby

- In our previous sessions with IRB when we need an object we just write an expression that describes it
  - `s = "hello"`
    - `"hello"`
  - `a = [1, 1, 2, 3, 5, 8]`
    - `[1, 1, 2, 3, 5, 8]`
- Another way to make an object is to call a method named new
  - `s = String.new`
    - `""`
  - `a = Array.new`
    - `[]`

Note that the names of the data types start with upper case letters

Classes

- In object-oriented programming languages, a data type is called a class
- Individual pieces of data are called instances of a class
  - we also say an object “belongs to” a class
- In Ruby, class names start with upper case letters
  - String, Array, Fixnum, Float, ...
- We can always find out what type of object is stored in a variable by calling a method named class
  - given the assignments on the previous slide:
    - `s.class`
    - `String`
    - `a.class`
    - `Array`

Every object belongs to a class, and the name of the class is kept with the object in the object store
Constructors

- Every class has a special method named `new`
  - this method is a **constructor** used to create new instances of the class
- When an expression has a call to `new`
  - Ruby figures out which class the new object will belong to
  - it allocates space for the object in the object store
  - it calls `new` to initialize the object with default values
- Examples:
  ```ruby
  >> a = Array.new
  => []
  >> a = Array.new(3)
  => [nil, nil, nil]
  >> a = Array.new(10, 0)
  => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ```

Why Make a New Class?

- An array seems like a perfectly good choice for an application that needs to build an index
  ```ruby
  >> a = Array.new(999599). nil
  => nil
  >> a.length
  => 999599
  >> i = b("madding", 999599)
  => 565987
  >> a[i] = "madding"
  => "madding"
  ```

- The problem: we can do too many other things with arrays
  - Arrays have a method named `first` that will return the first item in the array
  - doesn’t make sense for a table
  - should it return the contents of the first non-empty location?
  - There are methods that will rearrange the table
    - calling `sort!` or `reverse!` would be a disaster
    - the array could no longer be used as the index for a hash table

Why Make a New Class?

- If we define a new class we can restrict the set of operations
  - define a constructor to initialize new tables
  - define methods for inserting a string or looking up a string
  - a few other operations like printing a table or gathering statistics
  - There will be an array “inside” a table object
    - it will be used to manage the storage
    - but it will be hidden so programs don’t mistakenly mess it up
Class Definitions

- A class definition in Ruby starts with the keyword `class` and continues to a matching `end`.
- Methods that perform operations on objects of the class are defined inside the class.
- A special method named `initialize` will be called whenever a new object is created.

```ruby
class HashTable
  def initialize(n)
    @tbl = Array.new(n)
    @size = n
  end
  def find(s)
  end
  def insert(s)
  end
end
```

HashTable Class

- As you work on the tutorial project in Chapter 8 you will be filling in the outline of the `HashTable` class.
  - download `HashTable.rb` from the web site.
  - use a text editor to add new Ruby code.
  - save the file, load it into IRB, call the methods.
- You do not have to understand the fine details (instance variables, etc).
- The main ideas you do need to understand:
  - what we can do with an object is defined by its class.
  - programmers often define new classes.
  - the objects used for experiments in this week’s project belong to a new class named `HashTable`.
  - for these experiments you will make new `HashTable` objects, insert strings, and gather statistics.

A Trivial Hash Function: \( h_0 \)

- The `HashTable.rb` file defines a method named `ord` that converts a letter to a number between 0 and 25.
  ```ruby
  >> t = HashTable.new(10)
  #=> #<HashTable:0x8cb04>
  >> s = "hello"
  #=> "hello"
  >> s[0].ord
  #=> 7
  >> s[1].ord
  #=> 4
  ```
  - `ord` is short for “ordinal value”.
- We can use this method to create a trivial hash function:
  - put words starting with “a” in row 0
  - starting with “b” in row 1
  - ...
- But this function works only if the table has exactly 26 rows.
A Trivial Hash Function: $h_0$

- To get it to work for any number of rows $n$, find the remainder after dividing $s[0]$ by $n$
  - this works because $x \mod n$ is always a number between 0 and $n - 1$
- The definition in Ruby:

```
def h0(s, n)
    return s[0].ord % n
end
```

Example: a table with $n = 10$ rows

<table>
<thead>
<tr>
<th>Words</th>
<th>Block</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mango</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>orange</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>strawberry</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Testing this method in an IRB session:

```
> load "HashTable.rb"
true
> h0(“apple”, 10)
=> 0
> h0(“banana”, 10)
=> 1
> h0(“mango”, 10)
=> 2
> h0(“strawberry”, 10)
=> 8
```

A Better Hash Function: $h_1$

- Here is one way to use two letters from a string for tables with $26 \times 26 = 676$ rows
  - imagine the table is a set of blocks
  - each block has 26 rows
  - first block is for words starting with “a”, second for words starting “b”, ...
- Use the second letter to find a row within a block
  - “aardvark” goes in block 0, row 0
  - “cnidarian” would go in block 2, row 12
  - “zymurgy” in block 25, row 24

The addresses of the first row in each block are 0, 26, 52, ... 624

These numbers have a very useful pattern:
- block $i$ starts at address $26 \times i$
- The second letter determines how far past the first row we go

```
def h1(s, n)
    (s[0].ord * 26 + s[1].ord) % n
end
```

block number determined by first letter
row within the block determined by the second letter
Hash Function $h_1$

- Example in IRB:
  ```ruby
  def h1(s,n)
    (s[0].ord * 26 + s[1].ord) % n
  end
  >> h1("aardvark",1000)
  => 0
  >> h1("abcissa",1000)
  => 1
  >> h1("cnidarian",1000)
  => 65
  >> h1("zany",1000)
  => 650
  >> h1("zymurgy",1000)
  => 674
  ```

Hash Function in HashTable.rb

- The hash function that uses this radix-26 function is named $h_n$
  ```ruby
  def h1(s,n)
    (s[0].ord * 26 + s[1].ord) % n
  end
  ```
  ```ruby
  >> h1("aardvark",1000)
  => 0
  >> h1("abcissa",1000)
  => 1
  >> h1("cnidarian",1000)
  => 65
  >> h1("zany",1000)
  => 650
  >> h1("zymurgy",1000)
  => 674
  ```

Radix-26

- This formula can be extended to any number of letters
  ```ruby
  h(s) = s_0 \times 26^4 + s_1 \times 26^3 + s_2 \times 26^2 + s_3 \times 26^1 + s_4 \times 26^0
  ```

- This is the same method used to figure out the value of a string of digits in a positional number system
  - e.g. the number "723" in octal (base 8) is
    ```ruby
    7 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 = 467
    ```

- The function that weights each letter by a power of 26 is called radix-26
  ```ruby
  f \times 26^7 + a \times 26^6 + v \times 26^5 + o \times 26^4 + r \times 26^3 + i \times 26^2 + t \times 26^1 + o \times 26^0
  ```

- Question:
  - In math $x \times y = y \times x$
  - This hash function computes the product of the codes of the first two letters
  - Does that mean $h(\text{"ap"}) = h(\text{"pa"})$?
Summary

- A trivial hash function \( h_0 \) uses the first letter of a word \( s[0].ord \)
  - range of values = 0..25
- A slightly better function \( h_1 \) uses the first two letters \( s[0].ord \times 26 + h[1].ord \)
  - range = 0..675
- A general-purpose function \( h_n \) uses all the letters, e.g.
  \[ s[0].ord \times 26^7 + s[1].ord \times 26^6 + \ldots \]
  - After computing the product of the letter values find the remainder mod \( n \)
  - \( n \) is the number of rows in the table
  - the result is between 0 and \( n-1 \)

Collisions

- The general-purpose hash function does a pretty good job of "scattering" words around a table

```
>> fruits.each { |x| puts x; puts h(x,100) }

apple
70
banana
2
grape
42
kiwi
48
```

- A good hash function assigns words to random locations

```
2
2
2
2
2
2
2
2
2
2
```

Birthday Paradox

- Even a perfectly random assignment can’t avoid all collisions
- Can’t we just make a bigger table?
- That helps, but the odds are still quite high
- Example: odds of a collision when inserting 100 words into a table of size \( n \):

<table>
<thead>
<tr>
<th>( n )</th>
<th>( p(\text{collision}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>99.29%</td>
</tr>
<tr>
<td>10,000</td>
<td>39.04%</td>
</tr>
<tr>
<td>100,000</td>
<td>4.83%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.49%</td>
</tr>
</tbody>
</table>

- 1,000 empty rows for every full one, but still almost 5% chance of a collision!
- A famous paradox from probability theory explains why the odds of a collision are so high
- Assume all birthdays are equally likely
- In a room with \( m \) people, what are the odds that two or more people have the same birthday?
  - Intuition might tell you its around \( m / 365 \)
  - that is the right probability for two people at random having the same birthday
- But the real question: the probability that every pair in the room have different birthdays

<table>
<thead>
<tr>
<th>( n )</th>
<th>odds of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12%</td>
</tr>
<tr>
<td>20</td>
<td>41%</td>
</tr>
<tr>
<td>30</td>
<td>70%</td>
</tr>
<tr>
<td>50</td>
<td>97%</td>
</tr>
<tr>
<td>100</td>
<td>99.99996%</td>
</tr>
</tbody>
</table>

Search for “birthday paradox” at Wikipedia
Birthday Paradox

- The statistics behind the birthday paradox also apply to hash tables:
  - birthdays: 365 choices
  - hash tables: \( n \) choices (\( n \) is the size of the table)
- Two people having the same birthday is the same as two words hashing to the same location in a table
- To try this at a party:
  - start with a blank calendar (an empty table)
  - as you ask people their birthdays, put a check mark next to a day on the calendar (that row in the table has been filled)
  - if someone’s birthday falls on a day that has been checked already, you have found a collision