Fill out the Brown Computer Science Survey you got in your email!

Only takes 5 min!

All multiple choice!

If you didn’t receive the survey, email litofish@cs.brown.edu
Brown alumna and New York Times reporter Natasha Singer will talk about a new movement -- "responsible computing"
Sets, Dictionaries & Hash Tables

CS16: Introduction to Data Structures & Algorithms
Spring 2020
Q: how would you build a (basic) search engine?
What’s so Hard about Search Engines?

"The **Google** Search index contains hundreds of billions of webpages and is well over 100,000,000 gigabytes in size."

How Google Search Works | Crawling & Indexing
https://www.google.com › search › crawl...
Search Through Each Page?

- Assume Google indexes 200 billion pages
- If we scan 1 page in 1 microsecond
  - each search would take 55 hours
- How can we improve search time?
Outline

- Sets
- Dictionaries
- Hash Tables
- Ex: Search engine
Dictionary

- Collection of key/value pairs
  - distinct and unordered keys
- Supports value lookup by key
- Also known as a *map*
  - “maps” keys to values
- examples
  - name → address
  - word → definition
Dictionary ADT

- **add(key, value):**
  - adds key/value pair to dict.

- **object get(key):**
  - returns value mapped to key

- **remove(key):**
  - removes key/value pair

- **int size():**
  - returns number key/value pairs

- **boolean isEmpty():**
  - returns TRUE if dict. is empty; FALSE otherwise
Q: how can we implement a dictionary?
Array-based Dictionary

- Can we use an expandable array $A$?
- $\text{add}(k,v)$:
  - store $(k,v)$ in first empty cell of $A$
  - takes $O(1)$ if you keep track of first empty cell
- $\text{get}(k)$:
  - scan $A$ to find value with key $\text{key}=k$
  - takes $O(n)$
- $\text{remove}(k)$:
  - scan $A$ to find pair with $\text{key}=k$ & remove
  - takes $O(n)$

Is $O(n)$ good enough? What if our dictionary stores 200B key/value pairs?
Q: can we do better?
Yes! with a Hash Table

- Hash tables are composed of
  - an array $A$
  - and a “hash” function $h: X \rightarrow Y$

\[ \text{Array } A \quad \& \quad h(x) \]
Dictionary vs. Hash Table

- A dictionary (or map) is an abstract data type
  - can be implemented using many different data structures
- A hash table is a dictionary data structure
  - one specific way to implement a dictionary
Yes! with a Hash Table

- A hash function is function $h: X \rightarrow Y$ that
  - *shrinks*: maps elements from a large input space to a *smaller* output space
  - *well spread*: $h$ spreads elements of $X$ over $Y$
Building a Dictionary w/ a Hash Table

- Choose a hash function $h : X \rightarrow Y$ with
  - $X = \text{“universe of keys”}$ and $Y = \text{“indices of array”}$
  - **add**($k, v$)
    - set $A[h(k)] = v$ which is $O(1)$
  - **get**($k$)
    - return $v = A[h(k)]$ which is $O(1)$
  - **remove**($k$)
    - delete $A[h(k)]$ which is $O(1)$
Hash Table — Add

keys: banner IDs
values: names

00943855
Kaila Jeter

00745911
Chantal Toupin

00238494
Alejandro Molina

00472885
David Laidlaw

00943855
Kaila Jeter

00238494
Alejandro Molina

00745911
Chantal Toupin
Q: What is the problem with this?

Remember that $|Y| < |X|$

(here $|X|$ denotes size of $X$)

...so some keys in $X$ will be hashed to the same location!

this is called the pigeonhole principle

there just isn’t enough room in $Y$ to fit all of $X$

...therefore some values in array will be overwritten

this is called a collision
Overcoming Collisions

- Hash Table with Chaining
  - store *multiple* values at each array location
  - each array cell stores a “bucket” of pairs
    - can implement bucket as a list or expandable array or …

\[ \text{buckets:} \]

\[ A \]

\[ \& \ h(x) \]

**FYI:** there are many other approaches e.g., linear probing, quadratic probing, cuckoo hashing,…
### Hash Table

| table: array |
| h: hash function |

**function add(k, v):**
- index = h(k)
- table[index].append(k, v)

**function get(k):**
- index = h(k)
- for (key, val) in table[index]:
  - if key = k:
    - return val
  - error("key not found")

- \(O(1)\) if computing hash function is \(O(1)\)
- runtime depends on bucket size
Let’s do another example but with Chaining!

We’ll use the following hash function

\[ h(banner\_id) = banner\_id \mod 7 \]
Hash Table — Add

**keys:** banner IDs
**values:** names

```
00943855
Kaila Jeter

00745911
Chantal Toupin

00238494
Alejandro Molina

00472885
David Laidlaw

00231924
Lauren Ho

00472885
David Laidlaw

00238494
Alejandro Molina

00745911
Chantal Toupin

00543163
Surbhi Madan
```

$h(key) = key \% 7$

Array of buckets w/ key/value pairs:
Hash Table — Get

keys: banner IDs
values: names

h(key) = key % 7

Array of buckets w/ key/value pairs

What is the worst-case run time of Get?
Hash Table with Chaining

- What is the worst-case runtime of Get?
  - $\approx$ size of largest bucket

- What is the size of largest bucket?
  - assume we have $n$ students and a table of size $m$
  - if $h$ “spreads” keys roughly evenly then
    - each bucket has size $\approx \frac{n}{m}$
    - ex: if $n=150$ and $m=7$ each bucket has size $\approx \frac{150}{7} = 21$

- But what is the size of the largest bucket asymptotically?
  - assume $m$ is a constant (i.e., it does not grow as a function of $n$)
    - each bucket has size $\approx \frac{n}{m} = \frac{n}{c} = O(n)$ 😞
Q: Can we do better than $O(n)$?
Beating $O(n)$ — Idea #1

- **Idea:** use large table
- Banner IDs have 8 digits so max ID is $99,999,999$
- Use table of size $m=100,000,000$
  - w/ hash function $h(key) = key$
- Are there any collisions in this case?
  - no collisions because every pair gets its own cell
  - What is run time of Get?
    - $O(1)$ since we don’t need to scan buckets
- What is the problem with this approach?
  - what if we only store 150 students? we’re wasting $99,999,850$ cells
Beating $O(n)$ — Idea #2

- **Idea**: use a table of size equal to the number of students + “good” hash function
  - set the table size to $m = n$
  - use a hash function $h$ that spreads keys well
- No wasted space since $n = m$
  - in other words, “table size” = “number of students”
- If $h$ spreads keys roughly evenly then each bucket has size
  - $\approx \frac{n}{m} = \frac{n}{n} = 1 = O(1)$
- What hash function should we use?
  - Suppose $n = 150$ (i.e., we want to insert 150 students)
  - should we use the hash function $h(key) = key \% 150$?
Banner ID Hashing

Form groups of 10

Activity #1
Banner ID Hashing

Activity #1

5 min
Banner ID Hashing

Activity #1

4 min
Banner ID Hashing

3 min

Activity #1
Banner ID Hashing

Activity #1

2 min
Banner ID Hashing

Activity #1

1 min
Banner ID Hashing

Activity #1
Beating $O(n)$ — Idea #2

- Idea #2 relied on an assumption:
  - if $h$ spreads keys roughly evenly then each bucket has size
  - $\approx \frac{n}{m} = \frac{n}{n} = 1 = O(1)$

- Will $h(ID) = ID \% 11$ spread banner IDs evenly?
  - it depends on the banner IDs…
  - if banner IDs are chosen randomly then Yes
  - But what if next year all banner IDs are multiples of 11?
  - Then all banner IDs will map to 0!
  - So there will be one bucket with all IDs
  - so worst-case runtime of Get will be $O(n)$
Since keys are not necessarily random, we make the hash function random
Universal Hash Functions

- Special "families" of hash functions
  - \( UHF = \{ h_1, h_2, \ldots, h_q \} \)
  - designed so that if we pick a function from the family at random and use it on a set of keys, then it is very likely that the function will “spread” the keys (roughly) evenly
Universal Classes of Hash Functions

J. Lawrence Carter and Mark N. Wegman

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received August 8, 1977; revised August 10, 1978

This paper gives an input independent average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions. Given any sequence of inputs the expected time (averaging over all functions in the class) to store and retrieve elements is linear in the length of the sequence. The number of references to the data base required by the algorithm for any input is extremely close to the theoretical minimum for any possible hash function with randomly distributed inputs. We present three suitable classes of hash functions which also can be evaluated rapidly. The ability to analyze the cost of storage and retrieval without worrying about the distribution of the input allows as corollaries improvements on the bounds of several algorithms.
Example of Universal Hash Functions

- Setup to store \( n \) key/value pairs
  - choose prime \( p \) larger than \( n \)
  - choose 4 numbers \( a_1, a_2, a_3, a_4 \) at random between 0 and \( p-1 \)
- Hashing a key \( k \)
  - break \( k \) into 4 parts
    - \( k_1, k_2, k_3, k_4 \)
  - output \( h(k) = \sum_{i=1}^{4} a_i \cdot k_i \mod p \)

- Setup to store 150 students
  - choose \( p=151 \)
  - choose \( a_1=12, \ a_2=43, \ a_3=105, \ a_4=83 \)
  - Hashing a key \( k=00238918 \)
    - break \( k \) into \( k_1=00, \ k_2=23, \ k_3=89, \ k_4=18 \)
    - output \( h(00238918) = 50 \)
Hash Table with UHFs

- Hash table w/ chaining using a universal hash function family
  - *Worst-case* runtime of Get is $O(n)$ 😞
  - But UHFs guarantee that worst-case happens very rarely
  - We can “expect” that Get will have runtime $O(1)$
- What do we mean by expect?
  - remember that with UHFs we picked one function from family at random
    - in example we picked the values $(a_1, a_2, a_3, a_4)$ at random
  - but for some functions in the family, keys will be well-spread & for others keys may be clustered
  - but if we were to compute the runtime of Hash Table with $h$ a million times, where each time we sample a hash function at random from the family…
  - …then the average of those runtimes would be $O(1)$
  - This is called “expected running time”
Hash Table with UHFs

- Hash table w/ chaining using a universal hash function family
  - We can “expect” that Get will have runtime $O(1)$
- What do we mean by expect?
  - remember that with UHFs we picked one function from family at random
    - in the example we picked the values $(a_1, a_2, a_3, a_4)$ at random
  - for some functions in the family, keys will be well-spread…
  - …while for others keys will be poorly spread, e.g., all mapped to same value
  - but if we were to compute the runtime of Hash Table with a million times, where each time we sample a hash function at random from the family…
  - …then the average of those runtimes would be $O(1)$
  - This is called “expected running time”
Why does Universal Hashing Work?

- See Chapter 1.5.2 in Dasgupta et al.
  - and/or read the proof in the following slides
  - You do not need to know the proof!
Proof of Universal Hashing
Inverses

‣ What is the inverse of a fraction $x/y$?
  ‣ $y/x$ because $(x/y)(y/x)=1$
  ‣ inverse is whatever we need to multiply it by to get 1

‣ What is the inverse of an int $x$ (not 1)?
  ‣ $1/x$ because $(x)(1/x)=1$

‣ What is the “integer” inverse of an int $x$ (not 1)?
  ‣ there is none…
  ‣ you can’t multiply an int w/ another int to get 1 (unless 1)
Modular Arithmetic

- If working modulo some number
  - Integers can have integer inverses!
- ex: let’s work mod 7
  - inverse of 2 mod 7 is 4 because 2*4 mod 7 = 1
  - inverse of 5 mod 7 is 3 because 5*3 mod 7 = 1
- Is this always true?
  - ex: does 2 have an inverse mod 4?
    - 2*0 mod 4 = 0; 2*1 mod 4 = 2
    - 2*2 mod 4 = 0; 2*3 mod 4 = 2
    - No!
- But it is true when we work modulo a prime number
  - mod a prime, every number except 0 has a unique inverse
Analysis

- Prime $p$ is the size of array
- $x_1, x_2, x_3, x_4$ are a banner ID in chunks
- $y_1, y_2, y_3, y_4$ are another banner ID in chunks
- If IDs are different, at least 1 of the chunks are diff
- Let's assume (wlog) it is the last one so
  - $x_4 \neq y_4$
- What is the probability that
  - $h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4)$
Analysis

‣ What is the probability that
  ‣ \( h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4) \)

‣ Step #1:
  ‣ find equivalent formulation of event
  ‣ that makes the randomness explicit
  ‣ what is the randomness here?

‣ Step #2:
  ‣ what is probability of equivalent formulation?
Step 1: Equivalent Formulation

\[ h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4) \]

by definition

\[ a_1 x_1 + \cdots + a_4 x_4 \equiv a_1 y_1 + \cdots + a_4 y_4 \pmod{p} \]

move things

\[ a_4 x_4 - a_4 y_4 \equiv (a_1 y_1 + a_2 y_2 + a_3 y_3) - (a_1 x_1 + a_2 x_2 + a_3 x_3) \pmod{p} \]

different

just some number; let’s call it \( c \)

\[ a_4 \cdot (x_4 - y_4) \equiv c \pmod{p} \]

\[ a_4 \equiv c \cdot (x_4 - y_4)^{-1} \pmod{p} \]
Step 2: Probability of Equiv. Formulation

- So hashes are equal when
  \[ a_4 \equiv c \cdot (x_4 - y_4)^{-1} \quad (\text{mod } p) \]
- But
  - \( x_4 \) and \( y_4 \) are different so \( x_4 - y_4 \neq 0 \)
  - and \( p \) is prime
  - so \((x_4 - y_4)\) has unique inverse \(\text{mod } p\)
- So \( c(x_4 - y_4)^{-1} \) can only take on one value
  - therefore \( a_4 \) can only take on one value
- What is the probability \( a_4 \) takes on that value?
  - \( a_4 \) is randomly chosen from \( p \) possible values so probability is \( \frac{1}{p} \)
Putting it all Together

- Prob. that some ID will collide w/ another ID
  - \( \frac{1}{p} = \frac{1}{151} \)
- For some ID,
  - expected # of collisions w/ all other IDs is
  - \( \frac{149}{151} = 0.986\ldots \)
- Expected size of an ID’s bucket is
  - \( 1 + 0.986\ldots = 1.986\ldots = O(1) \)
End of Universal Hashing Proof
Summary

- Array-based Dictionaries
  - Add is \( \text{worst-case } O(n) \)
  - Get is \( \text{worst-case } O(n) \)
- Hash Table-based Dictionaries with UHFs
  - Add is
    - \( \text{worst-case } O(n) \) but expected \( O(1) \)
  - Get is
    - \( \text{worst-case } O(n) \) but expected \( O(1) \)
Q: what can we build from dictionaries?
A (Basic) Search Engine

- Build a dictionary that maps keywords to URLs
  - query dictionary on keyword to retrieve URLs
- In context of search engines
  - the dictionary is often called an Index
A (Basic) Search Engine

- For each keyword `word` with a list of relevant URLs `url_1, ..., url_m`
  - store the pairs `(word|1, url_1), ..., (word|m, url_m)` in a dict `Index`
  - where “|” is string concatenation
  - Store the pair `(word, m)` in an auxiliary dictionary `Counts`

- To search for a keyword `Brown`
  - retrieve the count for `Brown` by querying `Count.get(Brown)`
  - to recover URLs, query `Index` on keys `Brown|1, ..., Brown|m`
    - `Index.get(word|1), ..., Index.get(word|m)`

Idea from Cash et al., NDSS ‘14
function build_index(page_list):
    index = dict()
    counts = dict()
    for page in page_list:
        for word in page:
            try:
                count = counts.get(word)
            except KeyError:
                counts.put(word, 0)
            count = counts.get(word)
            counts.put(word, counts[word] + 1)
            key = word + str(counts.get(word))
            index.put(key, page.url)
    return index

- build_index is \(O(nm)\) time
  - where \(n\) is number of pages and \(m\) is maximum number of words per page

Idea from Cash et al., NDSS '14
Search Index

```python
def search_index(index, word):
    output_list = list()
    count = 1
    while True:
        try:
            url = index.get(word + str(count))
            count = count + 1
        except KeyError:
            break
        output_list.append(url)
    return output_list
```

- If dictionary is implemented with hash table
  - search_index is expected $O(1)$ time
  - fast no matter how many pages and words
A (Basic) Search Engine

- What’s missing from our “search engine”?
  - No ranking!
  - But we’ll learn how to rank later in the course
    - ...after we learn about graphs
Sets

- Collection of elements that are
  - distinct and unordered
  - ...unlike lists and arrays
Set ADT

- **add**(object):
  - adds object to set if not there
- **remove**(object):
  - removes object from set if there
- **boolean contains**(object):
  - checks if object is in set
- **int size():**
  - returns number objects in set
- **boolean isEmpty():**
  - returns TRUE if set is empty; FALSE otherwise
- **list enumerate():**
  - returns list of objects in set (in arbitrary order)
Set Data Structure

- How can we implement a Set?
- Using an expandable array
  - add: $O(1)$
  - contains: $O(n)$ (scan array)
  - remove: $O(n)$ (find & compress)
- Can we do better?
Sets from Hash Tables

- We can implement sets with a hash table
- Sometimes called a Hash Set

```python
function add(object):
    index = h(object)
    table[index].append(object)

function contains(object):
    index = h(object)
    for elt in table[index]:
        if elt == object:
            return true
    return false
```

Expected $O(1)$
HashMap vs. HashSet

- HashMap
  - Hash table implementation of a dictionary
- HashSet
  - Hash table implementation of a set