CS 261: Graduate Data Structures

Week 2: Dictionaries and hash tables

David Eppstein
University of California, Irvine

Spring Quarter, 2020

Dictionaries

What is a dictionary?

Maintain a collection of key–value pairs

- Keys are often integers or strings but can be other types
- At most one copy of each different key
- Values may be any type of data

Operations update the collection and look up the value associated with a given key

Dictionary in everyday usage: book of definitions of words.
So the keys are words and the values are their definitions

Example application

We've already seen one in week 1!

In the depth-first-search example, we represented the input graph as a dictionary with keys = vertices and values = collections of neighboring vertices

The algorithm didn't need to know what kind of object the vertices are, only that they are usable as keys in the dictionary

Dictionary API

Create new empty dictionary

Look up key k and return the associated value

(Exception if k is not included in the collection)

Add key–value pair (k, x) to the collection

(Replace old value if k is already in the collection)

Check whether key k is in the collection and return Boolean result

Enumerate pairs in the collection

Dictionaries in Java and Python

Python: dict type

- Create: `D = {key1: value1, key2: value2, ...}`
- Get value: `value = D[key]`
- Set value: `D[key] = value`
- Test membership: `if key in D:`
- List key–value pairs: `for key, value in D.items():`

Java: HashMap

Similar access methods with different syntax

Python example: Counting occurrences

```
def count_occurrences(sequence):
    D = {}              # Create dictionary
    for x in sequence:
        if x in D:      # Test membership
            D[x] += 1   # Get and then set
        else:
            D[x] = 1    # Set
    return D
```

Non-hashing dictionaries: Association list

Store unsorted collection of key–value pairs

- Very slow (each get/set must scan whole dictionary),
  $O(n)$ time per operation where n is # key–value pairs
- Can be ok when you know entire dictionary will have size $O(1)$
- We will use these as a subroutine in *hash chaining*

Non-hashing dictionaries: Direct addressing

Use key as index into an array

- Only works when keys are small integers
- Wastes space unless most keys are present
- Fast: $O(1)$ time to look up a key or change its value
- Important as motivation for hashing

Non-hashing dictionaries: Search trees

Binary search trees, B-trees, tries, and flat trees

- We'll see these in weeks 6 and 7!

  (Until then, you won't need to know much about them)
- Unnecessarily slow if you need only dictionary operations (searching for exact matches)
- But they can be useful for other kinds of query (inexact matches)

Hashing

Hash table intuition

The short version:

Use a *hash function* $h(k)$ to map keys to small integers

Use direct addressing with key $h(k)$ instead of k itself

Hash table intuition

Maintain a (dynamic) array A whose cells store key–value pairs

Construct and use a hash function $h(k)$ that "randomly" scrambles the keys, mapping them to positions in A

Store key–value pair (k, x) in cell $A[h(k)]$ and do lookups by checking whether the pair stored in that cell has the correct key

When table doubles in size, update h for the larger set of positions

All operations: $O(1)$ amortized time (assume computing h is fast)

Complication: *Collisions*. What happens when two keys k_1 and k_2 have the same hash value $h(k_1) = h(k_2)$?

Hash functions: Perfect hashing

Sometimes, you can construct a function h that maps the n keys one-to-one to the integers $0, 1, \ldots n - 1$

By definition, there are no collisions!

Works when set of keys is small, fixed, and known in advance, so can spend a lot of time searching for perfect hash function

Example: reserved words in programming languages / compilers

Use fixed (not dynamic) array A of size n, store key–value pair (k, x) in cell $A[h(k)]$ (include value so can detect invalid keys)

Hash functions: Random

Standard assumption for analysis of hashing:

The value of $h(k)$ is a random number,
independent of the values of h on all the other keys

Not actually true in most applications of hashing

Results in this model are not mathematically valid
for non-randomly-chosen hash functions

(nevertheless, analysis tends to match practical performance
because many nonrandom functions behave like random)

This assumption is valid for Java IdentityHashMap:
Each object has hash value randomly chosen at its creation

Hash functions: Cryptographic

There has been much research in *cryptographic hash functions* that map arbitrary information to large integers (e.g. 512 bits)

Could be used for hash functions in dictionaries
by taking result modulo n

Any detectable difference between the results and a random function $\Rightarrow$ the cryptographic hash is considered broken

Too slow to be practical for most purposes

Hash functions: Fast, practical, and provable

It's possible to construct hash functions that are fast and practical

... and at the same time use them in valid mathematical analysis of hashing algorithms

Details depend on the choice of hashing algorithm

We'll see more on this topic later this week!

Quick review of probability

Basic concepts

Random event: Something that might or might not happen

Probability of an event: a number, the fraction of times that an event will happen over many repetitions of an experiment

$$\Pr[X]$$

Discrete random variable: Can take one of finitely many values, with (possibly different) probabilities summing to one

Uniformly random variable: Each value is equally likely

Independent variables: Knowing the value of any subset doesn't help predict the rest (their probabilities stay the same)

Expected values

If R is any function $R(X)$ of the random choices we're making, its *expected value* is just weighted average of its values:

$$E[R] = \sum_{\text{outcome } X} \Pr[X] R(X)$$

Linearity of expectations: For all collections of functions R_i,

$$\sum_i E[R_i] = E\left[\sum_i R_i\right]$$

(It expands into a double summation, can do sums in either order)

If X can only take the values 0 or 1, then $E[X] = \Pr[X = 1]$

So $E[\text{number of } X_i \text{ that are one}] = \sum_i \Pr[X_i = 1]$

Analysis of random algorithms

We will analyze algorithms that use randomly-chosen numbers to guide their decisions

...but on inputs that are not random (usually worst-case)

The amount of time (number of steps) that such an algorithm takes is a random variable

Most common measure of time complexity: $E[\text{time}]$

Also used: "with high probability", meaning that the probability of seeing a given time bound is $1 - o(1)$

Chernoff bound: Intuition

Suppose you have a collection of random events
(independent, but possibly with different probabilities)

Define a random variable X: how many of these events happen

Main idea: X is very unlikely to be far away from $E[X]$

Example: Flip a coin n times
$X =$ how many times do you flip heads?

Very unlikely to be far away from $E[X] = n/2$

Chernoff bound (multiplicative form)

Let X be a sum of independent 0–1 random variables

Then for any $c > 1$,

$$\Pr[X \geq c\,E[X]] \leq \left(\frac{e^{c-1}}{c^c}\right)^{E[X]}$$

(where $e \approx 2.718281828\ldots$, "Euler's constant")

Similar formula for the probability that X is at most $E[X]/c$

Three special cases of Chernoff

$$\Pr[X \geq c\, E[X]] \leq \left(\frac{e^{c-1}}{c^c}\right)^{E[X]}$$

If c is constant, but $E[X]$ is large $\Rightarrow$
probability is an exponentially small function of $E[X]$

If $E[X]$ is constant, but c is large $\Rightarrow$
probability is $\leq 1/c^{\Theta(c)}$, even smaller than exponential in c

If c is close to one $(c = 1 + \delta$ for $0 < \delta < 1)$ $\Rightarrow$
probability is $\leq e^{-\delta^2 E[X]/3}$

Collisions and collision resolution

Random functions have many collisions

Suppose we map n keys randomly to a hash table with N cells.

Then the expected number of pairs of keys that collide is

$$\sum_{\text{keys } i,j} \Pr[i \text{ collides with } j] = \binom{n}{2}\frac{1}{N} \approx \frac{n^2}{2N}$$

The sum comes from linearity of expectation.
There are $\binom{n}{2}$ key pairs, each colliding with probability $1/N$.

E.g. birthday paradox: a class of 80 students has in expectation $\binom{80}{2}\frac{1}{365} \approx 8.66$ pairs with the same birthday

Conclusion: Avoiding all collisions with random functions needs hash table size $\Omega(n^2)$, way too big to be efficient

Resolving collisions: Open addressing

Instead of having a single array cell that each key can go to,
use a “probe sequence” of multiple cells

Look for the key at each position of the probe sequence until either:

You find a cell containing the key

You find an empty cell (and deduce that key is not present)

Many variations; we’ll see two later this week

Resolving collisions: Hash chaining (bucketing)

Instead of storing a single key–value pair in each array cell, store an association list (collection of key–value pairs)

To set or get value for key k, search the list in $A[h(k)]$ only

Time will be fast enough if these lists are small

(But in practice the overhead of having a multi-level data structure and keeping a list per array cell means the constant factors in time and space bounds are larger than other methods.)

First hashing method ever published: Peter Luhn, 1953

Expected analysis of hash chaining

Assumption: N cells with *load factor* $\alpha = n/N = O(1)$
(Use dynamic tables and increase the table size when $N << n$)

Then, for any key k, the time to set or get the value for k is $O(\ell_k)$, the number of key–value pairs in $A[h(k)]$.

By linearity of expectation,

$$E[\ell_k] = 1 + \sum_{\text{key } j \neq k} \Pr[j \text{ collides with } k] = \frac{n-1}{N} < \alpha = O(1).$$

So with tables of this size, expected time/operation is $O(1)$.

Expected size of the biggest chain

A random or arbitrary key has expected time per operation $O(1)$

But what if attacker chooses key whose cell has max # keys?

(Again, assuming constant load factor)

Chernoff bound: The probability that any given cell has $\geq x\alpha$ keys is at most

$$\left(\frac{e^x}{x^x}\right)^\alpha$$

(and is lower-bounded by a similar expression)

For $x = C\frac{\log n}{\log\log n}$, simplifies to $1/N^c$ for some c (depending on C)

$c > 1 \Rightarrow$ high probability of no cells bigger than $x\alpha$

$c < 1 \Rightarrow$ expect many cells bigger than $x\alpha$

Can prove: Expected size of largest cell is $\Theta(\frac{\log n}{\log\log n})$

Linear probing

What is linear probing?

The simplest possible probe sequence:
Look for key k in cells $h(k), h(k)+1, h(k)+2, \ldots \pmod{N}$

Invented by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel, 1954, and first analyzed by Donald Knuth, 1963

- Fast in practice: simple lookups and insertions, works well with cached memory
- Commonly used
- Requires a high-quality hash function (next week)
- Load factor α must be < 1 (else no room for all keys)

Lookups and insertions

With array A of length N, holding n keys, and hash function h:

To look up value for key k:

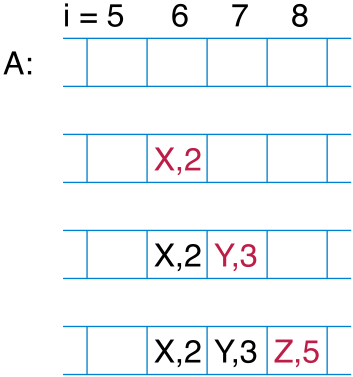
```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    raise exception
return value from A[i]
```

To set value for k to x:

```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    n += 1
    if n/N too large:
        expand A and rebuild
A[i] = k,x
```

Example

key	value	hash
X	2	6
Y	3	7
Z	5	6



	i = 5	6	7	8
A:				
		X,2		
		X,2	Y,3	
		X,2	Y,3	Z,5

Default position for Z was filled ⇒ placed at the next available cell

Have to be careful with deletion: blanking the cell for X would make the lookup algorithm think Z is also missing!

Deletion

First, find key and blank cell:

```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    raise exception
A[i] = empty
```

Then, pull other keys forward:

```
j = (i + 1) % N
while A[j] nonempty:
    k = key in A[j]
    if h(k) <= i < j (mod N):
        A[i] = A[j]
        A[j] = empty
        i = j
    j = (j + 1) % N
```

Result = As if remaining keys inserted without the deleted key

Blocks of nonempty cells

The analysis is controlled by the lengths of contiguous blocks of nonempty cells (with empty cells at both ends)

Any sequence of B cells is a block only when:

- Nothing hashes to the empty cells at both ends
- Exactly B items hash to somewhere inside
- For each i, at least i items hash to the first i cells

Our analysis will only use the middle condition:
Exactly B items hash into the block

Probability of forming a block

Suppose we have a sequence of exactly B cells

Expected number of keys hashing into it: αB

To be a block, actual number = B = $\frac{1}{\alpha}$expected

Chernoff bound:
Probability this happens is at most

$$\left(\frac{e^{1/\alpha-1}}{(1/\alpha)^{1/\alpha}}\right)^{\alpha B} = c^B$$

for some constant $c < 1$ depending on α

Long sequences of cells are exponentially unlikely to form blocks!

Linear probing analysis

Expected time per operation on key k

$$= O(\text{expected length of block containing } h(k))$$

$$= \sum_{\text{possible blocks}} \Pr[\text{it is a block}] \times \text{length}$$

$$= \sum_{\ell} (\text{number of blocks of length } \ell \text{ containing } k) \times c^{\ell} \times \ell$$

$$= \sum_{\ell} \ell \times c^{\ell} \times \ell$$

$$= O(1)$$

(because the exponentially-small c^{ℓ} overwhelms the factors of ℓ)

Expected length of the longest block

A random or arbitrary key has expected time per operation $O(1)$

But what if attacker chooses key whose block has max # length?

Same analysis $\Rightarrow$ blocks of length $C \log n$ have probability $1/N^{\Theta(1)}$ (inverse-exponential in length) of existing

Large $C \Rightarrow$ high probability of no blocks bigger than $C \log n$

Small $C \Rightarrow$ expect many blocks bigger than $C \log n$

Can prove: Expected size of largest block is $\Theta(\log n)$

Cuckoo hashing

Main ideas

Cuckoo = bird that lays eggs in other birds' nests
when it hatches, the baby pushes other eggs and chicks out

Cuckoo hashing = open addressing, only two possible cells per key

Invented by Rasmus Pagh and Flemming Friche Rodler in 2001

Lookup: always $O(1)$ time – just look in those two cells

Deletion: easy, just blank your cell

Insertion: if a key is already in your cell, push it out
(and recursively insert it into its other cell)

Comparison with other methods

When an adversary can pick a bad key, other methods can be slow:

$\Theta(\log n/\log\log n)$ for hash chaining

$\Theta(\log n)$ for linear probing

In contrast, cuckoo hash lookup is always $O(1)$

Cuckoo hashing is useful when keys can be adversarial
or when lookups must always be fast

e.g. internet packet filter

But for fast average case, linear probing may be better

Requirements and variations

Sometimes the two cells for each key are in two different arrays but it works equally well for both to be in a single array

(We will use the single-array version)

Basic method needs load factor $\alpha < 1/2$ to avoid infinite recursion

More complicated variations can handle any $\alpha < 1$

- Store a fixed number of keys per cell, not just one
- Keep a "stash" of $O(1)$ keys that don't fit

Easy operations: Lookup and delete

With array A, hash functions $h1$ and $h2$ such that $h1(k) \neq h2(k)$:

To look up value for key k:

```
if A[h1(k)] contains k:
    return value in A[h1(k)]
if A[h2(k)] contains k:
    return value in A[h2(k)]
raise exception
```

To delete key k:

```
if A[h1(k)] contains k:
    empty A[h1(k)] and return
if A[h2(k)] contains k:
    empty A[h2(k)] and return
raise exception
```

Insertion

To insert key *k* with value *x*:

```
i = h1(k)
while A[i] is non-empty:
    swap k,x with A[i]
    i = h1(k) + h2(k) - i
A[i] = k,x
```

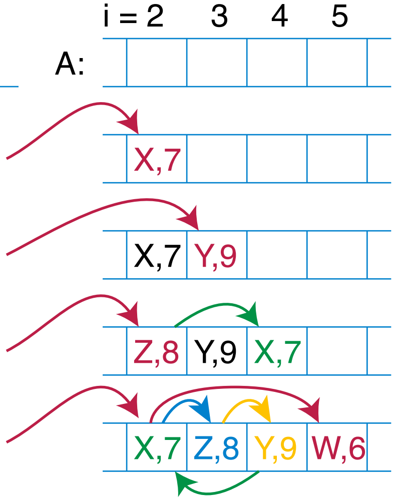
Not shown: test for infinite loop

If we ever return to a state where we are trying to re-insert the original key *k* into its original cell $h1(k)$, the insertion fails

(put *k* into the stash or rebuild the whole data structure)

Example

key	value	h1	h2
X	7	2	4
Y	9	3	4
Z	8	2	3
W	6	2	5



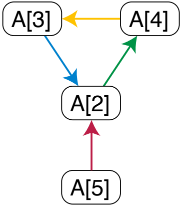
Graph view of cuckoo state

Vertices = array cells

Edges = pairs of cells used by the same key
Directed away from cell where the key is stored

key	value	h1	h2
X	7	2	4
Y	9	3	4
Z	8	2	3
W	6	2	5

	i = 2	3	4	5
A:	X,7	Z,8	Y,9	W,6



Key properties: Each vertex has at most one outgoing edge

Each component is a *pseudotree* (cycle with trees attached)

Graph view of key insertion

Add edge $h1(k) \to h2(k)$ to the graph

If $h1(k)$ already had an outgoing edge:

Follow path of outgoing edges, reversing each edge in the path

Case analysis:

- $h1(k)$ was in a tree: path stops at previous root
- $h1(k)$ was in a pseudotree and $h2(k)$ was in a tree: path loops through cycle, comes back out, reverses new edge, and stops at root of tree
- $h1(k)$ and $h2(k)$ are both in pseudotrees (or same pseudotree): fails, too many edges for all vertices to have only one outgoing edge

Summary of graph interpretation

The insertion algorithm

- Works correctly when the *undirected* graph with edges $h1(k)$—$h2(k)$ has at most one cycle per component
- Fails when any component has two cycles (equivalently: more edges than vertices)
- Takes time proportional to the component size

The graph is *random* (each two vertices equally likely to be edge)

We know a lot about random graphs!

For n vertices and αn edges, $\alpha < 1/2$:

- With high probability, all components are trees or pseudotrees
- Prob. component of v has size ℓ is exponentially small in ℓ
- Expected size of component containing v is $O(1)$
- But expected largest component size $\Theta(\log n)$
- Expected number of pseudotrees is $O(1)$

For $\alpha = 1/2$:

- There is a *giant component* with $\Theta(n^{2/3})$ vertices (with high probability)
- The rest of the graph is still small trees and pseudotrees

What random graphs imply for cuckoo hashing

For load factor $\alpha < 1/2$:

- With high probability, all keys can be inserted
- Expected time per insertion is $O(1)$
- Expected time of slowest insertion is $O(\log n)$

For load factor $\alpha \geq 1/2$:

- It doesn't work (in simplest variation)

k-independent hash functions

The problem

All analysis so far has assumed hash function is random

But that is rarely achievable in practice

- Cryptographic functions act like random but too slow
- IdentityHashMap not usable in all applications and doesn't allow changing to a new hash function

Many software libraries use ad-hoc hash functions that are arbitrary, but not random

- We can't prove anything about how well they work!

Instead, we want a function that

- Can be constructed using only a small seed of randomness
- Is fast (theoretically and in practice) to evaluate
- Can be proven to work well with hashing algorithms

k-independence

Choose function h randomly from a bigger family H of functions

If $H =$ all functions, h is uniformly random (previous assumption)

If H is smaller, h will be less random

Define H to be k-independent if every k-tuple of keys has independent outputs (every tuple of outputs is equally likely)

Bigger values of k give stronger independence guarantees

An example of a (bad) 1-independent hash function: choose one random number r and define h_r to ignore its argument and return r

So we are selecting a function randomly from the set $H = \{h_r\}$

Is k-independence enough?

Expected-time analysis of hash chaining only considers pairs of keys

(expected time is sum of probabilities that another key collides with given key)

If we use a 2-independent hash function, these probabilities are the same as for a fully independent hash function

Expected-time analysis of linear probing has been done with 5-independent hashing [Pagh, Pagh & Ružić]

But there exist 4-independent hash functions designed to make linear probing bad [Pătrașcu & Thorup]

Cuckoo hashing requires $\Theta(\log n)$-independence

Algebraic method for k-independence

From b-bit numbers (that is, $0 \leq \text{value} < 2^b$ to the range $[0, N-1]$

Choose (nonrandom) prime number $p > 2^b$

Choose k random coefficients $a_0, a_1, \ldots a_{k-1}$ (mod p)

$$h(x) = \left(\left(\sum_i a_i x^i \right) \bmod p \right) \bmod N$$

Works because, for every k-tuple of keys and every k-tuple of outputs, exactly one polynomial mod p produces that output

$O(k)$ arithmetic operations but multiplications can be slow

Tabulation hashing

Represent key x as a base-B number for an appropriate base B
$x = \sum_i x_i B^i$ with $0 \le x_i < B$

E.g. for $B = 256$, x_i can be calculated by `(x>>(i<<3))&0xff`
(using only shifts and masks, no multiplication)

Let d be number of digits ($d = 4$ for 32-bit keys and $B = 256$)

Initialize: fill a $d \times B$ table T with random numbers

$h(x) =$ bitwise exclusive or of $T[i, x_i]$ ($i = 0, 1, \ldots d - 1$)

3-independent but not 4-independent; works anyway for linear probing and static cuckoo hashing [Pătrașcu & Thorup]

Summary

Summary

- Dictionary problem: Updating and looking up key–value pairs
- In standard libraries (e.g. Java HashMap, Python dict)
- Hash tables: direct addressing + hash function
- Hash functions: perfect, cryptographic, random, k-independent (algebraic and tabulation)
- Hash chaining, expected time, expected worst case
- Linear probing, expected time, expected worst case
- Cuckoo hashing and random graphs
- Reviewed basic probability theory + Chernoff bound