Dictionaries: Hashing, Amortization, and other magic

Dictionary Problem

Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

We assume items have distinct keys (or that inserting new one clobbers old).
Balanced BSTs solve in $O(\lg n)$ time per op. (in addition to inexact searches like next-largest).
Goal: $O(1)$ time per operation.
We saw in the last lecture a lower-bound of $\Omega(\lg n)$ for searching and $\Omega(n \lg n)$ for sorting *in the comparison model*. We also saw that by moving out of the comparison model and using the fact that items are (bounded-size) integers, we can sort faster, in linear time. In this and the next two lectures, we will see how hashing lets us do $O(1)$ search, overcoming the $\Omega(\lg n)$ lower bound.
A caveat: The time for search is $O(1)$ not in the worst-case, but is an average-case, high probability statement.

Python Dictionaries:

Items are (key, value) pairs e.g. $d =$ {'algorithms': 5, 'cool': 42}

d.items()	→	[('algorithms', 5),('cool',5)]
d['cool']	→	42
d[42]	→	KeyError
'cool' in d	→	True
42 in d	→	False

Python set is really dict where items are keys (no values)

Motivation

Dictionaries are perhaps the most popular data structure in CS

- built into most modern programming languages (Python, Perl, Ruby, JavaScript, Java, C++, C#, ...)
- e.g. best docdist code: word counts & inner product
- implement databases: (DB_HASH in Berkeley DB)
  - English word → definition (literal dict.)
  - English words: for spelling correction
  - word → all webpages containing that word
  - username → account object
- compilers & interpreters: names → variables
- network routers: IP address → wire
- network server: port number → socket/app.
- virtual memory: virtual address → physical

Less obvious, using hashing techniques:

- substring search (grep, Google) [L9]
- file or directory synchronization (rsync)
- cryptography: file transfer & identification [L10]

How do we solve the dictionary problem?

Simple Approach: Direct Access Table

This means items would need to be stored in an array, indexed by key (random access)

Problems:

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\implies$ large space — e.g. one key of 2^{256} is bad news.

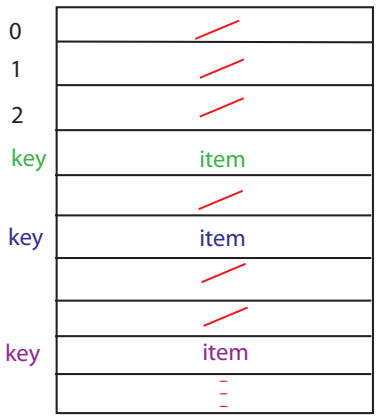

Figure 1: Direct-access table

2 Solutions:

Solution to 1: "prehash" keys to integers.

- In theory, possible because keys are finite $\implies$ set of keys is countable
- In Python: hash(object) (actually hash is misnomer should be "prehash") where object is a number, string, tuple, etc. or object implementing __hash__ (default = id = memory address)

Solution to 2: hashing (verb from French 'hache' = hatchet, & Old High German 'happja' = scythe)

- Reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n =$ # keys stored in dictionary
- hash function h: $\mathcal{U} \to \{0, 1, \ldots, m-1\}$
- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions?

We will see two ways

1. Chaining: TODAY
2. Open addressing: L10

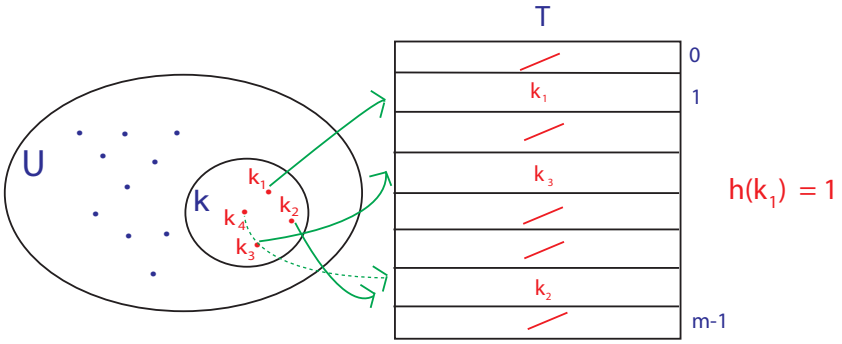

Figure 2: Mapping keys to a table

Chaining

Linked list of colliding elements in each slot of table

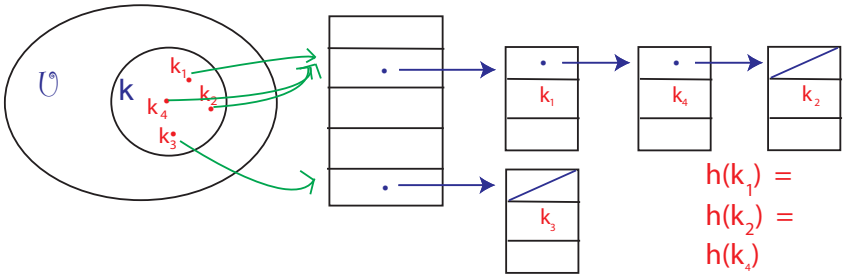

Figure 3: Chaining in a Hash Table

- Search must go through *whole* list T[h(key)]
- Worst case: all n keys hash to same slot $\implies \Theta(n)$ per operation

Simple Uniform Hashing:

An assumption (cheating): Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

$$\begin{aligned} \text{let } n &= \#\text{ keys stored in table} \\ m &= \#\text{ slots in table} \\ \underline{\text{load factor}}\ \alpha &= n/m = \text{expected } \#\text{ keys per slot} = \text{expected length of a chain} \end{aligned}$$

Performance

This implies that expected running time for search is $\Theta(1+\alpha)$ — the 1 comes from applying the hash function and random access to the slot whereas the α comes from searching the list. This is equal to $O(1)$ if $\alpha = O(1)$, i.e., $m = \Omega(n)$.

We will see three examples of hash functions, two in the recitations and one in [L9].

Hash function families:

Division Method:

$$h(k) = k \mod m$$

where m is ideally prime

Multiplication Method:

$$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$$

where a is a random odd integer between 2^{w-1} and 2^w, k is given by w bits, and m = table size = 2^r.

How Large should Table be?

- want $m = \Theta(n)$ at all times
- don't know how large n will get at creation
- m too small $\implies$ slow; m too big $\implies$ wasteful

Idea:

Start small (constant) and grow (or shrink) as necessary.

Rehashing:

To grow or shrink table hash function must change (m, r)

$\implies$ must rebuild hash table from scratch
for item in old table: $\rightarrow$ for each slot, for item in slot
insert into new table
$\implies$ $\Theta(n+m)$ time $= \Theta(n)$ if $m = \Theta(n)$

How fast to grow?

When n reaches m, say

- $m \mathrel{+}= 1$?
  $\implies$ rebuild every step
  $\implies$ n inserts cost $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$
- $m \mathrel{*}= 2$? $m = \Theta(n)$ still ($r \mathrel{+}= 1$)
  $\implies$ rebuild at insertion 2^i
  $\implies$ n inserts cost $\Theta(1 + 2 + 4 + 8 + \cdots + n)$ where n is really the next power of 2 $= \Theta(n)$
- a few inserts cost linear time, but $\Theta(1)$ "on average".

Amortized Analysis

This is a common technique in data structures — like paying rent: \$1500/month $\approx$ \$50/day

- operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time.

Back to Hashing:

Maintain $m = \Theta(n) \implies \alpha = \Theta(1) \implies$ support search in $O(1)$ expected time (assuming simple uniform or universal hashing)

Delete:

Also $O(1)$ expected as is.

- space can get big with respect to n e.g. $n\times$ insert, $n\times$ delete
- solution: when n decreases to $m/4$, shrink to half the size $\implies$ $O(1)$ amortized cost for both insert and delete — analysis is harder; see CLRS 17.4.

String Matching

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)
E.g. $s =$ '6.006' and $t =$ your entire INBOX ('grep' on UNIX)

Simple Algorithm:

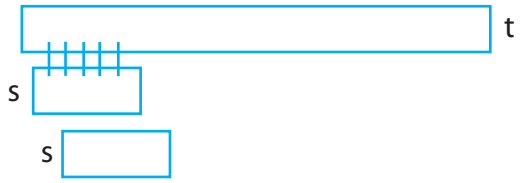

Figure 4: Illustration of Simple Algorithm for the String Matching Problem

any($s == t[i : i + \text{len(s)}]$ for i in range(len(t) − len(s)))
— $O(|s|)$ time for each substring comparison
$\implies O(|s| \cdot (|t| - |s|))$ time
$= O(|s| \cdot |t|)$ potentially quadratic

Karp-Rabin Algorithm:

- Compare $h(s) == h(t[i : i + \text{len(s)}])$
- If hash values match, likely so do strings
  - can check $s == t[i : i + \text{len(s)}]$ to be sure $\sim$ cost $O(|s|)$
  - if yes, found match — done
  - if no, happened with probability $< \frac{1}{|s|}$
    $\implies$ expected cost is $O(1)$ per i.
- need suitable hash function.
- expected time $= O(|s| + |t| \cdot \text{cost(h)})$.
  - naively $h(x)$ costs $|x|$
  - we'll achieve $O(1)$!
  - idea: $t[i : i + len(s)] \approx t[i + 1 : i + 1 + len(s)]$.

Rolling Hash ADT

(We did this informally in class. Make sure to go over the formal description of the rolling hash ADT below.)

Maintain string x subject to

- $r()$: reasonable hash function $h(x)$ on string x
- r.append(c): add letter c to end of string x
- r.skip(c): remove front letter from string x, assuming it is c

Karp-Rabin Application:

```
for c in s: rs.append(c)
for c in t[:len(s)]: rt.append(c)
if rs() == rt(): ...
```

This first block of code is $O(|s|)$

```
for i in range(len(s), len(t)):
   rt.skip(t[i-len(s)])
   rt.append(t[i])
   if rs() == rt(): ...
```

The second block of code is $O(|t|) + O(\#\text{ matches} - |s|)$ to verify.

Data Structure:

Treat string x as a multidigit number u in base a where a denotes the alphabet size, e.g., 256

- $r() = u \bmod p$ for (ideally random) prime $p \approx |s|$ or $|t|$ (division method)
- r stores $u \bmod p$ and $|x|$ (really $a^{|x|}$), not u

  $\implies$ smaller and faster to work with ($u \bmod p$ fits in one machine word)
- r.append(c): $(u \cdot a + \text{ord}(c)) \bmod p = [(u \bmod p) \cdot a + \text{ord}(c)] \bmod p$
- r.skip(c): $[u - \text{ord}(c) \cdot (a^{|u|-1} \bmod p)] \bmod p$

  $= [(u \bmod p) - \text{ord}(c) \cdot (a^{|x-1|} \bmod p)] \bmod p$

Readings

CLRS Chapter 11.4 (and 11.3.3 and 11.5 if interested)

Open Addressing

Another approach to collisions:

- no chaining; instead all items stored in table (see Fig. 5)

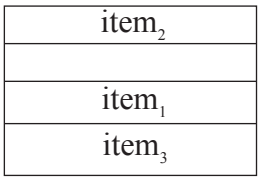

Figure 5: Open Addressing Table

- one item per slot $\implies m \geq n$
- hash function specifies *order* of slots to probe (try) for a key (for insert/search/delete), not just one slot; **in math. notation**:

  We want to design a function h, with the property that for all $k \in \mathcal{U}$:

$$h : \mathcal{U} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

universe of keys — trial count — slot in table

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

  is a permutation of $0, 1, \ldots, m-1$. i.e. if I keep trying $h(k,i)$ for increasing i, I will eventually hit all slots of the table.

Insert(k,v) : Keep probing until an empty slot is found. Insert item into that slot.

```
for i in xrange(m):
  if T[h(k,i)] is None:        # empty slot
    T[h(k,i)] = (k,v)          # store item
    return
raise 'full'
```

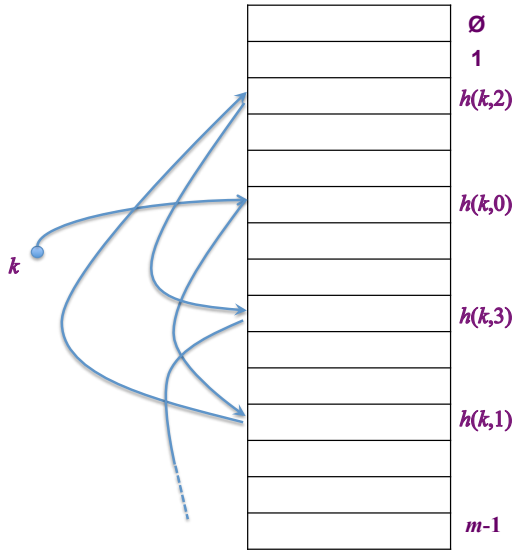

Figure 6: Order of Probes

Example: Insert $k = 496$

Search(k): As long as the slots you encounter by probing are occupied by keys $\neq k$, keep probing until you either encounter k or find an empty slot—return *success* or *failure* respectively.

```
for i in xrange(m):
    if   T[h(k,i)] is None:        # empty slot?
         return None               # end of "chain"
    elif T[h(k,i)][∅] == k:        # matching key
         return T[h(k,i)]          # return item
return None                        # exhausted table
```

Deleting Items?

- can't just find item and remove it from its slot (i.e. set $T[h(k,i)] = \text{None}$)
- *example*: delete(586) $\implies$ search(496) fails
- replace item with special flag: "DeleteMe", which Insert treats as None but Search doesn't

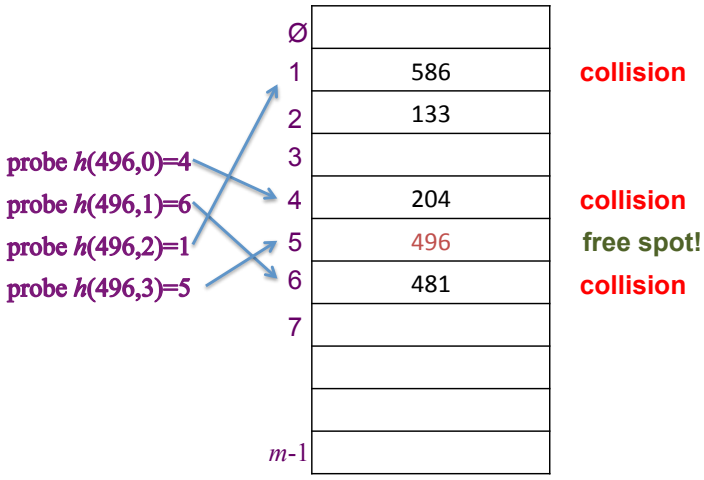

Figure 7: Insert Example

Probing Strategies

Linear Probing

$h(k,i) = (h'(k) + i) \bmod m$ where $h'(k)$ is ordinary hash function

- like street parking
- **problem?** *clustering*—cluster: consecutive group of occupied slots
  as clusters become longer, it gets *more* likely to grow further (see Fig. 8)

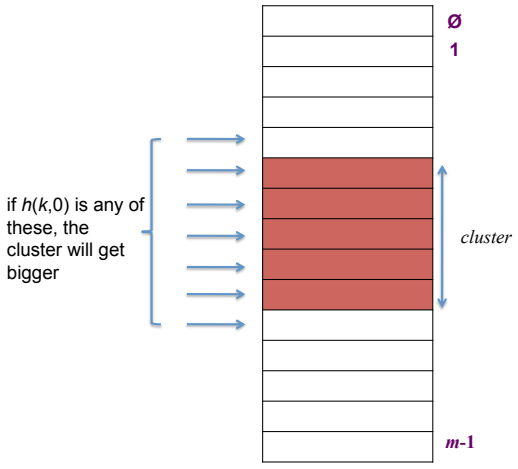

Figure 8: Primary Clustering

- can be shown that for $0.01 < \alpha < 0.99$ say, clusters of size $\Theta(\log n)$.

Double Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$ where $h_1(k)$ and $h_2(k)$ are two ordinary hash functions.

- actually hit all slots (permutation) if $h_2(k)$ is relatively prime to m for all k **why?**

$$h_1(k) + i \cdot h_2(k) \bmod m = h_1(k) + j \cdot h_2(k) \bmod m \Rightarrow m \textit{ divides } (i-j)$$

- e.g. $m = 2^r$, make $h_2(k)$ always odd

Uniform Hashing Assumption (cf. Simple Uniform Hashing Assumption)

Each key is equally likely to have any one of the $m!$ permutations as its probe sequence

- not really true
- but double hashing can come close

Analysis

Suppose we have used open addressing to insert n items into table of size m. Under the uniform hashing assumption the next operation has expected cost of $\leq \frac{1}{1-\alpha}$, where $\alpha = n/m (< 1)$.
Example: $\alpha = 90\% \implies$ 10 expected probes

Proof:

Suppose we want to insert an item with key k. Suppose that the item is not in the table.

- probability first probe successful: $\frac{m-n}{m} =: p$
  (n bad slots, m total slots, and first probe is uniformly random)
- if first probe fails, probability second probe successful: $\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$
  (one bad slot already found, $m-n$ good slots remain and the second probe is uniformly random over the $m-1$ total slots left)

- if 1st & 2nd probe fail, probability 3rd probe successful: $\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$ (since two bad slots already found, $m-n$ good slots remain and the third probe is uniformly random over the $m-2$ total slots left)
- ...

$\Rightarrow$ Every trial, success with probability at least p.
Expected Number of trials for success?

$$\frac{1}{p} = \frac{1}{1-\alpha}.$$

With a little thought it follows that search, delete take time $O(1/(1-\alpha))$. Ditto if we attempt to insert an item that is already there.■

Open Addressing vs. Chaining

Open Addressing: better cache performance (better memory usage, no pointers needed)

Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor α (OA degrades past 70% or so and in any event cannot support values larger than 1)

Cryptographic Hashing

A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (cryptographic) hash value, such that an accidental or intentional change to the data will change the hash value. The data to be encoded is often called the *message*, and the hash value is sometimes called the *message digest* or simply digest.

The ideal cryptographic hash function has the properties listed below. d is the number of bits in the output of the hash function. You can think of m as being 2^d. d is typically 160 or more. These hash functions can be used to index hash tables, but they are typically used in computer security applications.

Desirable Properties

1. **One-Way (OW)**: Infeasible, given $y \in_R \{0,1\}^d$ to find any x s.t. $h(x) = y$. This means that if you choose a random d-bit vector, it is hard to find an input to the hash that produces that vector. This involves "inverting" the hash function.

2. **Collision-resistance (CR)**: Infeasible to find x, x', s.t. $x \neq x'$ and $h(x) = h(x')$. This is a collision, two input values have the same hash.

3. **Target collision-resistance (TCR)**: Infeasible given x to find $x' = x$ s.t. $h(x) = h(x')$.

TCR is weaker than CR. If a hash function satisfies CR, it automatically satisfies TCR. There is no implication relationship between OW and CR/TCR.

Applications

1. **Password storage**: Store $h(PW)$, not PW on computer. When user inputs PW', compute $h(PW')$ and compare against $h(PW)$. The property required of the hash function is OW. The adversary does not know PW or PW' so TCR or CR is not really required. Of course, if many, many passwords have the same hash, it is a problem, but a small number of collisions doesn't really affect security.

2. **File modification detector**: For each file F, store $h(F)$ securely. Check if F is modified by recomputing $h(F)$. The property that is required is TCR, since the adversary wins if he/she is able to modify F without changing $h(F)$.

3. **Digital signatures**: In public-key cryptography, Alice has a public key PK_A and a private key SK_A. Alice can sign a message M using her private key to produce $\sigma = sign(SK_A, M)$. Anyone who knows Alice's public key PK_A and verify Alice's signature by checking that $verify(M, \sigma, PK_A)$ is true. The adversary wants to forge a signature that verifies. For large M it is easier to sign $h(M)$ rather than M, i.e., $\sigma = sign(SK_A, h(M))$. The property that we require is CR. We don't want an adversary to ask Alice to sign x and then claim that she signed x', where $h(x) = h(x')$.

Implementations

There have been many proposals for hash functions which are OW, CR and TCR. Some of these have been broken. MD-5, for example, has been shown to not be CR. There is a competition underway to determine SHA-3, which would be a Secure Hash Algorithm certified by NIST. Cryptographic hash functions are significantly more complex than those used in hash tables. You can think of a cryptographic hash as running a regular hash function many, many times with pseudo-random permutations interspersed.