Lecture 8: Hashing I

Lecture Overview

- Dictionaries and Python
- Motivation
- Prehashing
- Hashing
- Chaining
- Simple uniform hashing

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)

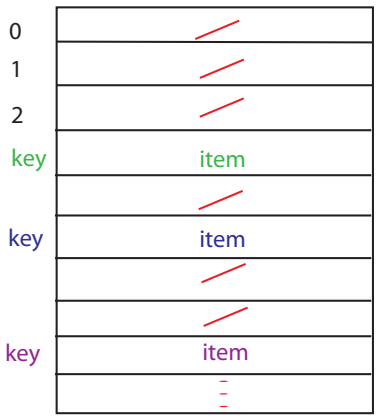

Figure 1: Direct-access table

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.

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

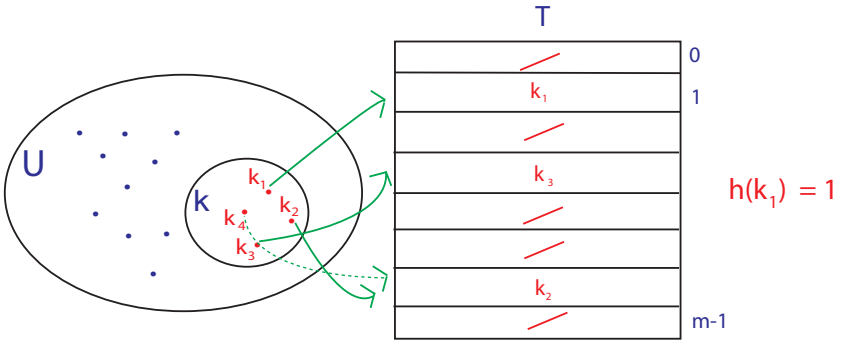

Figure 2: Mapping keys to a table

- 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

Chaining

Linked list of colliding elements in each slot of table

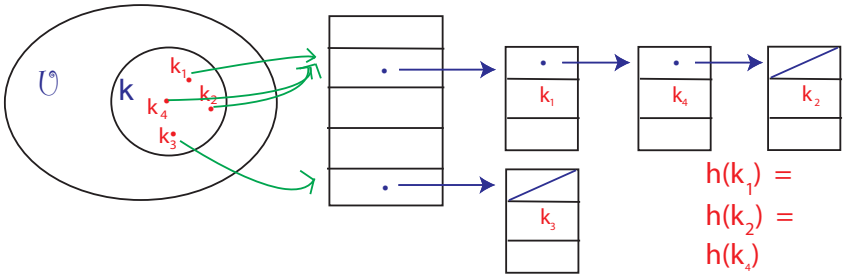

Figure 3: Chaining in a Hash Table

- 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.

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

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].