CSCI 262
Data Structures

26 – Hashtables

Review: Sets and Maps

• Data structures for holding unique keys
• Sets just hold keys
• Maps associate keys with values
• Principal operations:
  – find() - lookup key/value in set/map
  – insert() - put a new key/value into set/map
  – erase() - remove a key/value from set/map

O(1) Table Lookups

• Suppose keys are known to be in range 0-99:
  – What is easiest way to store keys?
  – What is the “big-O” complexity of find()?

• Arguably, all keys in a computer are numbers!
  – However, range may be very large (too large!)
  – Also, have to ensure uniqueness of number conversion for different keys

Basic Hashtable Idea

• Convert key to an integer (called a hash code)
• Take hash code, mod table size
• Store key at resulting index

It’s that easy, except for collisions!

Very Simple Illustration

• Suppose keys are non-negative integers
• Suppose table size is 5
• Use key as hash code

Collision Resolution

Collisions:
  – Table size typically << size of universe of keys
  – Many keys will hash to same index!
  – Collisions are inevitable (see Birthday paradox)

Different schemes for dealing with collisions:
  – Chaining
  – Open addressing (not covered today)

Sometimes “mod table size” is implicit in the term “hash code”, but typically the computations are separate.
Chaining

• Basic idea: store linked list at each index
• When finding:
  — If null pointer at index, return NOT FOUND
  — Else, search every node in linked list for item
• When inserting:
  — First do a find() – if item is in linked list, do nothing
  — If not present in list, insert new item at head of list
• When erasing:
  — Find item
  — If found, remove from linked list

Updated Illustration

• Suppose keys are non-negative integers
• Suppose table size is 5
• Use key as hash code

Analysis of Hashing with Chaining

• Best Case:
  — Every entry occupies a unique location
  — Linked lists are all empty or have a single node
  — All operations thus O(1)
• Worst case?
  — N entries occupying same location
  — find() O(N)
  — Also insert/delete O(N) since find() is first step
  — Inserts really average 1 + … + N = O(N^2) over N inserts \(\Rightarrow\) O(N)
  — per insert – gets more complicated with deletions

Analysis, con’t.

• Worst case not so great
  — Recall BST set/map find() in worst case O(log N)
  — O(N) much, much worse than O(log N)
• However, we will likely use hashtable many times:
  — Q: what is expected (average) cost of find()?
  — Probabilistic analysis sketch:
    • Assume every hash code equally probable
    • Expected occupancy in any slot is \(\alpha = N / \text{table size}\)
    • Expected cost of find() is \(1 + \alpha/2 = O(1)\)
    • Typically choose table size so \(\alpha \leq 0.75\) or so.

Hash Functions

• First defense against collisions is a good hash function
• For example: hashing strings
  — Could just take first four bytes, cast to int
    • Easy and fast to compute
    • Can’t distinguish “football”, “footrace”, “foot”, …
  — Could just add up ascii codes
    • Almost as easy and fast to compute
    • Can’t distinguish “saw” from “was”, though

Analysis, con’t.

If “uniform hashing” assumption holds:
  — find() is O(1) expected
  — insert() is O(1) plus O(1) for insert at head = O(1)
  — erase() is O(1) plus O(1) for erase from linked list = O(1)

All operations are expected O(1)!
(Could get unlucky, of course...)
Designing a Good Hash Function

- A good hash function:
  - Fast to compute
  - Uses entire object
  - Separates similar objects widely
  - "Random-like"
- Java’s String hash function:
  \[ h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i} \]

Hashtables in C++ (Stanford CPP Lib)

- HashSet
- HashMap
- Same interfaces as Set, Map
  - StanfordCPPLib provides `hashCode()` for base types, strings
  - Must override: `int hashCode(<object_type> val)`
  - Must override: `bool operator!=(<object_type> val)`
  - Remember looping over keys go in no particular order!

Hashtables in C++ (STL)

- C++ 11 and later:
  - unordered_set
  - unordered_map
- Same interfaces as set, map
  - C++ provides a default hash for many types
  - However, for user-defined key types, non-trivial!