CSCI 262
Data Structures

13 – Hashtables

CS@Mines

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

CS@Mines

You Design It

- Suppose set keys are integers in range 0-99:
  - What is easiest way to store keys?
  - What is the “big-O” complexity of find() in your scheme?

CS@Mines

O(1) Table Lookups

- My solution: `vector<bool> table(100)`
  - If `table[i] == true` then `i` is a key in the set
  - O(1) cost
- 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

CS@Mines

Mod

- With the range of our keys being so large (infinitely large?) how do we fit into a vector?
- We could just mod key’s value by vector size to get index…

CS@Mines

Basic Hashtable Idea

- Create an array, vector, or similar of some size
- For each key you want to store:
  - Convert key to an integer (called a *hash* code)
  - Index equals hash code *mod* array size
  - Store key at resulting index in array

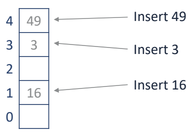
It’s that easy, except for collisions!

CS@Mines

Very Simple Illustration

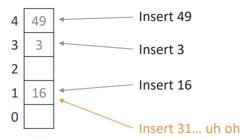
- Suppose keys are non-negative integers
- Suppose table size is 5
- Use key itself as hash code (i.e., hash(x) = x)

Note when finding keys in the hashtable, we need to test the value stored to see if it equals the key – so we don't use bool values as in previous slides.



Very Simple Illustration

- Suppose keys are non-negative integers
- Suppose table size is 5
- Use key itself as hash code (i.e., hash(x) = x)



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:

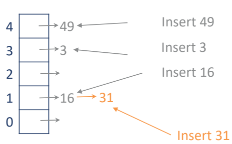
- Chaining
- Open addressing (not covered today)

Chaining

- Basic idea: store *linked list* at each index
- When finding:
  - 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 into 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 (N entries, table size >= N):
  - 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() is thus O(N)
  - Also insert/delete O(N) since find() is first step

    Inserts really average 1 + … + N = $O(N^2)$ over N inserts → O(N) per insert – gets more complicated with deletions

Analysis, con't.

- Worst case not so great
- 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 α = N / table size
    - *Expected* cost of find() is 1 + α/2 = O(1)
    - Typically choose table size so α ≤ 0.75 or so.

CS@Mines

Analysis, con't.

If "uniform hashing" assumption holds:

- find() is O(1) expected
- insert() is O(1) plus O(1) for linked list insert = O(1)
- erase() is O(1) plus O(1) for linked list erase = O(1)

All operations are expected O(1)!

(*Could* get unlucky, of course…)

CS@Mines

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

CS@Mines

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 (string of length *n*):

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

s[0] · $31^{(n-1)}$ + s[1] · $31^{(n-2)}$ + … s[n – 2] · 31 + s[n - 1]

CS@Mines

Example

What is the index for the string "apple" with an array size of 100?

s[0] · $31^{(n-1)}$ + s[1] · $31^{(n-2)}$ + … s[n – 2] · 31 + s[n - 1]

```
hash("apple")
  = 'a' × 31⁴ + 'p' × 31³ + 'p' × 31² + 'l' × 31 + 'e'
  = 97 × 923,521 + 112 × 29,791 + 112 × 961 + 108 × 31 + 101
  = 93,029,210
```

If the array size was 100, then

- index = hash % array_size
- index = 10

CS@Mines

Hashing Integers

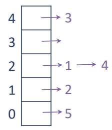
- Division method:
  - hash(k) = k mod table size
  - Avoid e.g., table size = 2^p → else hash(k) just low order bits of k!
  - Good choice: prime not too close to exact power of 2
  - Note this method dictates size of hashtable
- Multiplication method:
  - Multiply k by real constant A: 0 < A < 1
  - Extract *fractional* part of kA
  - hash(k) = ⌊(table size)(kA mod 1)⌋
  - Advantage: size of table doesn't matter!
  - Good choices for A: transcendental numbers, $\frac{\sqrt{5}-1}{2}$, etc.

CS@Mines

Multiplication Method Illustration

- Suppose keys are non-negative integers
- Suppose table size is 5
- Use A = $\frac{\sqrt{5}-1}{2}$
- Insert 1,2,3,4,5

E.g., insert 3:
$\lfloor 5(3A \bmod 1) \rfloor$
$= \lfloor 5(1.85410 \bmod 1) \rfloor$
$= \lfloor 5(.85410) \rfloor$
$= \lfloor 4.2705 \rfloor$
$= 4$



Hashtables in C++ (STL)

- C++ 11 and later:
  - unordered_set
  - unordered_map
- C++ provides a hash function for many types
- However, for user-defined key types, must provide a hash function
  - Quick-and-dirty choice: convert object to a string representation, then use the string hash function.

Up Next

- Wednesday, March 6
  - Return and go over midterms
  - Maybe some lecture as well
- Friday, March 8
  - Lab 8 - Hashtable