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 set keys are integers 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

Mod

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

Basic Hashtable Idea

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

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

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

![Hashtable Illustration](image)
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:
  - 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 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 \(\rightarrow O(N)\)
  - per insert – gets more complicated with deletions

Analysis, cont.’

- Worst case not so great
  - Recall BST set/map find() in worst case O(log_2 N)
  - O(N) much, much worse than O(log_2 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.

Analysis, cont.’

- 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)!
  \((Could\ get\ unlucky,\ of\ course...\)
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

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):

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

Example

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

\[ s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + ... + s[n-2] \cdot 31 + s[n-1] \]

hash(“apple”)

= ‘a’ × 31^4 + ‘p’ × 31^3 + ‘o’ × 31^2 + ‘o’ × 31 + ‘l’
= 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

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, \( \sqrt{5} - 1 \), etc.

Multiplication Method Illustration

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

Hashables in C++ (STL)

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

- **Wednesday, April 25**
  - Hand back, go over midterms
- **Friday, April 27**
  - Lab 11 – Analysis of Algorithms
- **Monday, April 30**
  - Graphs
  - Reading: Chapter 19
- **Wednesday, May 2**
  - Final exam review
  - Project 5 due