CSCI 262
Data Structures

13 – 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

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?

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

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...

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!
**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)

```
4  0
3  3
2  1
1  16
0  31
```

Insert 49
Insert 3
Insert 16
Insert 31

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)

```
4  0
3  3
2  1
1  16
0  31... uh oh
```

Insert 49
Insert 3
Insert 16
Insert 31

**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

```
4  0
3  3
2  1
1  16
0  31
```

Insert 49
Insert 3
Insert 16
Insert 31

**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 \( \alpha = \frac{N}{\text{table size}} \)
    - Expected cost of find() is \( 1 + \alpha/2 = O(1) \)
    - Typically choose table size so \( \alpha \leq 0.75 \) or so.

 hash("apple")
 = 'a' \times 31 + 'p' \times 31^2 + 'o' \times 31^3 + 'l' \times 31 + 'e' 
 = 97 \times 923,521 + 112 \times 29,791 + 112 \times 961 + 108 \times 31 + 101 
 = 93,029,210

If the array size was 100, then
- index = hash \% array size
- index = 10

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

  \[s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \ldots + s[n-2] \times 31 + s[n-1]\]

Hashing Integers

- Division method:
  - \( \text{hash}(k) = k \mod \text{table size} \)
  - Avoid e.g., table size = \( 2^n \) → else \( \text{hash}(k) \) 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 \( kA \)
  - \( \text{hash}(k) = \lfloor \text{table size}(kA \mod 1) \rfloor \)
  - 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 $A = \frac{\sqrt{s} - 1}{2}$
- Insert 1, 2, 3, 4, 5

  E.g., insert 3:
  
  $\lfloor 5 \times (3 \mod 1) \rfloor$
  
  $= \lfloor 5 \times 1.85410 \mod 1 \rfloor$
  
  $= \lfloor 5 \times 0.85410 \rfloor$
  
  $= 4.2705$
  
  $= 4$

**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