Dictionaries and Hashing

Outline

- Dictionary ADT
- Containers with look-up by keys
  - Bucket-based data structures
- Hash Functions
  - Hash Code
  - Compression Map
- Collision Resolution

Search

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst-case O(N) for either searching or inserting

Need data structure with optimal efficiency for searching and inserting
Review: Dictionary ADT

Def'n: abstract data structure of items with keys that supports two basic operations: insert a new item, and return an item with a given key

- Insert a new item
- Search for an item (items) having a given key
- Remove a specified item
- Sort the symbol table
- Select the kth largest item in a symbol table
- Join two symbol tables

Also may want construct, test if empty, destroy, copy...

Review: Dictionary ADTs

Types
- Log File
- Ordered Dictionary
- Hash Table
- Skip List
What if the set of keys is large?

- Example: calendar for 1..N days
  - N could be 365, 3x365 or greater
  - Can look up a particular day in O(1) time
- Every day is represented by a bucket, i.e., some container
- If we have a range of integers that fits into memory, everything is easy
  - What if we don’t?

Intuition for a more general case

- Store student records by last name
  - Idea: make a table, one bucket per last name
  - How many buckets do we need?
  - Need to account for newly added last names
- How should last names be ordered?
  - We just want fast look-ups for now
- Idea: recompute last names into integers (table addresses)
  - Potential problem: collisions

Log Files

- Good for archiving structured data
  - some databases, bank transactions, record of logs over Internet
  - backup if something goes wrong
- Keys have no influence on arrangement
- Cheap Insertion: O( )
- Expensive Search, Removal: O( )
- Space: O( )
Hashing

- Reference items in a table by keys
  - Use arithmetic operations to transform keys into table addresses (<em>buckets</em>)

Need:
- Hash function: transforms the search key into a table address
- Collision resolution: dealing with search keys that hash to same table address

Dictionary ADT & Hashing

Hashing is an efficient implementation of:
- <em>Insert</em> a new item
- <em>Search</em> for an item (or items) having a given key
- <em>Remove</em> a specified item

Hashing is an inefficient implementation of:
- <em>Select</em> the <em>k</em><sup>th</sup> largest item in a symbol table
- <em>Sort</em> the symbol table

Hash Function

Two Parts
Hash Code: <em>t(key) ⇒ intmap</em>
- Maps the key into an integer
- Note that the key can a string, float, w-bit integer

Compression Map: <em>c(intmap) ⇒ address</em>
- Maps the integer into the bucket range [0, M-1]

Given key:
<em>h(key) ⇒ c(t(key)) ⇒ address</em>
Good hash functions

- Benefits of hash tables depend on having good hash functions
- Must be easy to compute
  - will compute a hash for every key
  - will compute same hash for same key
- Should distribute keys evenly in table
  - will minimize collisions
    - collision: two keys map to same address
    - trivial, poor hash function: \( h(key) \) { return 0; }
      - easy to compute, maximizes collisions

Hash Function:
Floats in fixed range

- key between 0 and 1
  \[ h(key) = \lfloor key \times M \rfloor \]

- key between \( s \) and \( t \)
  \[ h(key) = \lfloor ( (key - s)/(t - s) ) \times M \rfloor \]

Hash Function:
\( w \)-bit integers

- To get between 0 and 1
  - divide by \( 2^w \) (shift right \( w \) bits)
- To determine which bin
  - multiply by \( M \)
- That is:
  \[ h(key) = \lfloor (key \gg w) \times M \rfloor \]
- Not very convenient for integers
Hash Function:
* $w$-bit integers

Modular hash function
$$h(key) = key \mod M$$

- Great if keys randomly distributed
- Often, keys are not randomly distributed
- Example: midterm 1 scores cluster on 80
- Don’t want to pick a bad $M$, where bad:
  - $M$ and $key$ have common factors
- Don’t know key pattern in advance
  - Pick a prime number for $M$

Hash Function:
* $w$-bit integers

- Combination modular and multiply
  $$h(key) = \lfloor key \times \alpha \rfloor \mod M$$
  - Say $\alpha = 0.618033$
  - And $M$ is prime

Hash Code:
* Strings

- Consider the following strings:
  - stop, tops, pots, spot
- ASCII sum of each is equivalent
- All will map to same hash table address
  - I.e., will cause collision
- Position is important
Hash Code:
strings: \((x_0, x_1, \ldots, x_{k-1})\)

Polynomial Hash Code (p. 374 text)
- \(x_0a^{k-1} + x_1a^{k-2} + \ldots + x_{k-2}a + x_{k-1}\)

if \(a = 33\) then
- \(t(‘tops’) = \)
- \(t(‘pots’) = \)

Note: Experiments have shown that good values for ‘a’ are 33, 37, 39, or 41 producing few collisions.

Hash Code:
strings: \((x_0, x_1, \ldots, x_{k-1})\)

Cyclic Shift Hash Code (p. 375 text)
- Cyclic shift of partial sum of characters by a certain number of bits
- Cyclic shift of zero reverts to the case that simply sums all the characters in a string
- Using a shift of 5 or 6 has been shown to be good for English words (see Table 8.1 text)

Compression Mapping

Why needed? The range of possible hash codes for our keys will typically exceed the range of legal indices for our bucket \([0, M-1]\)

\(c(\text{intmap}) \Rightarrow \text{range } [0, M-1]\)
- \(\text{intmap} \) may be less than 0 or greater than \(M-1\)

Division Method
\[\text{intmap} \mod M, \text{ where } M \text{ is prime}\]

MAD (multiply and divide) Method
\[|a \cdot \text{intmap} + b| \mod M, \text{ where } M \text{ is prime and } a \text{ and } b \text{ are non-negative integers}\]
Complexity of Hashing

For simplicity, assume perfect hashing (no collisions)

- What is cost of insertion? \( O(1) \)
- What is cost of search? \( O(1) \)
- What is cost of removal? \( O(1) \)

Wouldn't it be nice to live in a perfect world?

Summary: Hash Functions

- Efficient ADT for insert, search, and remove
- Hash Function \( h(key) \Rightarrow addr \)
  - Maps key to table address
  - Hash code \( t(key) \Rightarrow intmap \)
    - Translates key into integer
  - Compression map \( c(intmap) \Rightarrow addr \)
    - Maps integer into range of 0 to \( M-1 \) addresses
- Therefore, \( h(key) = c(t(key)) \)