CPSC 223
Algorithms & Data Abstract Structures

Lecture 24:
Hash Tables

Today …

• Hash Tables  [Ch 12: 686-706]

• Reminders:
  – Project presentations Thursday …
  – Guest lecture next Tuesday
  – Next week: (re)read “The data-structure canon”
B-Trees versus Arrays

- What are advantages of balanced search trees over arrays for storing collections of data items?
  - Output (traversal) in sorted order
  - Faster retrieve (and lookup) …
  - $O(n)$ for arrays, $O(\log n)$ for balanced search trees

Can we improve search time for arrays?

Yes!

- Using Hash Tables …

Hash Tables

Basic Idea

- Define a “hash function” $h$
- $h : \text{Key} \rightarrow \text{Index}$
- Make $h$ fast (e.g., constant time)

- This makes retrieve $O(1)$!
- … which is even faster than in BSTs
Hash Tables

A “Hash Table” since

• Keys can refer to more complex structures
• As in the case for our Entries
  – (word, definition)
  – where word is the “key”
• Or consider an Employee record
  – (ssn, fname, lname, bdate, dept, age, sal, …)
  – where ssn is the “key”
• Here the array represents a “table” of such records

Some challenges

• How do we define a hash function?
• What makes for a “good” hash function?
• How do we deal with the fixed-size nature of arrays when combined with a hash function?
Hash Functions

“Perfect Hash Functions”
- Map each key to a unique array index
- Hard if you do not know all search key values to expect
- Note you may also have more keys than indexes

• Most Hash Functions
- Map two or more keys to the same index
- This results in “collisions”
- We have to deal with collisions (more later) …
- … but we also want hash functions that minimize collisions

Examples of Hash Functions (from textbook)

Assumptions
- keys are positive integers
- we have a hash table (array) of 100 elements (0 .. 99)

“Selecting digits”
• Select digits of the key to use as the hash value
• Lets say keys are 9-digit employee numbers
  - \( h(k) = 4^{th} \) and \( 9^{th} \) digit
  - For example: \( h(001364825) = 35 \)
  - Here we store (retrieve) entry with key 001364825 at table[35]
• This is a fast and simple approach, but
  - May not evenly distribute data
Examples of Hash Functions (from textbook)

“Folding”

- Add digits instead
- Lets say keys are 9-digit employee numbers
  - \( h(k) = i_1 + i_2 + \ldots + i_9 \) where \( k = i_1i_2\ldots i_9 \)
  - For example: \( h(001364825) = 29 \)
  - Store (retrieve) entry with key 001364825 at table[29]
- This is also fast, but
  - Also may not evenly distribute data
  - In this example, only hits ranges from 0 to 81
  - Can pick different schemes (like \( i_1i_2i_3 + i_4i_5i_6 + i_7i_8i_9 \))

Examples of Hash Functions (from textbook)

“Modular Arithmetic”

- Sometimes we end up with indexes outside of the range of table indexes
- We can use the modulo operator (%) to map values to valid table indexes
  - \( h(k) = \text{i mod table size} \)
  - In our example we can use the key directly …
    \( h(001364825) = 1,364,825 \text{ mod } 100 = 25 \)
  - Key values used may require carefully chosen table sizes
    - E.g., \( 110 \text{ mod } 100, 210 \text{ mod } 100, 310 \text{ mod } 100, \text{etc} \)
    - Convention to more evenly distribute values is to use a prime number (e.g., 101 in this case)
Examples of Hash Functions (from textbook)

**String-valued keys**

- What if we have strings as keys?
- Convert strings to integers … then apply \( \text{mod} \)
  - For instance add up each character’s ASCII value
  - “NOTE” = 78 + 79 + 84 + 69 = 310
- Problems with this?
  - Anagrams (same letters different word, e.g., TONE)
  - Different words can also have the same value
- Better approaches possible (see textbook)
  - Generally, weight value by position

Desiderata for Hash Functions

**Hash functions should …**

- Be functions!
  - The same key should always result in the same index
- Be fast to compute
  - Adds overhead in search or insertion
  - Modulo is a simple division
  - Summing string values depends on string size
- Scatter data evenly throughout the hash table
  - Minimize collisions if possible
  - Beware “skewed” key distributions
Resolving Collisions (insert)

- Two general approaches

  **Open Addressing**
  - If location occupied, then find another location

  **Restructuring the Hash Table**
  - Add more room to the Hash Table to store collisions

---

**Approach 1: Open Addressing**

If a location is taken, "probe" (search) array for the next "open" (available) index

- **Linear probing**
  - Search for next available sequentially
  - Take the next free index
  - If at the end, start at position 0
  - Search works similarly
    - Deletion tricky
    - Mark indexes as "deleted" so we don’t throw off search

Linear probing can create large “primary” clusters
Approach 1: Open Addressing

If a location is taken, “probe” (search) array for the next “open” (available) index

- **Quadratic probing**
  - Helps eliminate “primary” clusters
  - Instead of sequentially probing
  - Probe “quadratic” sequences
    - \( i + 1^2, i + 2^2, i + 3^2, i + 4^2, \ldots \)
  - Creates “secondary” clusters since collisions use same sequences

- **Double hashing**
  - Further reduces clustering
  - Use a second hash function \( h_2 \) to determine the size of sequence steps
  - Note steps depend on key value
Approach 1: Open Addressing

If a location is taken, “probe” (search) array for the next “open” (available) index

- **Increase array size**
  - As hash table fills, collisions increase
  - Dynamically resize array
    - Change hash function too!
    - Use new array size for modulo
  - But there is a catch
    - You can’t use new hash function on old data
    - You have to re-hash data instead of copy into new array

Approach 2: Restructure Array

Change the structure of the hash table to hold *multiple items* in the same position

- **Buckets**
  - Define each location as an array of “buckets”
  - If static arrays, have to determine bucket size in advance
  - Can use dynamically sized arrays for buckets
  - Can waste space if only a few collisions
Approach 2: Restructure Array

Change the structure of the hash table to hold *multiple items* in the same position

- **Separate Chaining** (HW10)
  - Instead of using a static arrays, use *linked lists* (*chains*)
  - We end up with “Chain Nodes” holding entries

![Diagram of Separate Chaining]
The Cost of Hashing

• Ideally
  – Insert, Delete, and Retrieve are $O(1)$
  – Traversal is $O(n)$ … but the result is not sorted

• In practice
  – Collisions increase the cost
  – Cost depends on the “load factor” … how full the table is
    \[ \alpha = \frac{\text{# items in table}}{\text{table size}} \]
  – As the table fills, the chances of collisions increase
  – Thus hashing efficiency decreases as load factor increases

  *Note that $\alpha > 1$ if more items than array positions*

The Cost of Hashing

*Cost of Separate Chaining*

• Insertion is still $O(1)$
  – New items added to the front of the linked list

• Deletion, retrieval may require searching entire linked list (chain)

• So again, cost depends on collisions

• Here $\alpha$ is the average length of each linked list
  (assuming a “good” hash function)
  – But since $\alpha = n/constant$, search is worst-case $O(n)$
  – In practice, hash tables are efficient at searching though!
Assignment 10

Assignment 10 – Chain Nodes

ChainNode
— Stores a keyword and a (linked) List of Entries
— Each Entry in the List has the keyword in their definition
— Collisions occur when more than one keyword hashes to the same table index …

Two ChainNodes

C1 : ChainNode
  keyword = “device”
  L1 : List<Entry>
    e1 : Entry
    e2 : Entry

C2 : ChainNode
  keyword = “contrivance”
  L2 : List<Entry>
    e1 : Entry
Assignment 10 – Hash Table

HashTable
- Maps each keyword to a table index (hash function)
- Each table index contains a (linked) List of ChainNodes
- In Dictionary
  - insert involves adding (keyword, Entry) pairs
  - remove involves removing Entry’s (and possibly ChainNodes)
  - search (new operation) finds and returns Entries given a keyword

If \( h(device) = 2 \) and \( h(contrivance) = 2 \)