Introduction to Hashing

Hash tables
Hash functions
And the journey ends...

Abstract data types

- Stack
- Queue
- Dictionary
- Priority queue

Tools

- Pointers
- Dynamic memory allocation
- Asymptotic analysis
- Recursion
- Recurrence relations

Data structures

- Linked list
- Circular array
- Array
- Binary heap
- (Binary) (search) tree
- Hash table

Algorithms

- Heapsort
- Sorting: Selection, Insertion, Bubble
- Divide and conquer paradigm (Merge sort and Quicksort)

November 22, 2017

Hassan Khosravi / Geoffrey Tien
Review: dictionary ADT

- Stores **values** associated with user-specified **keys**
  - Values may be any (homogenous) type
  - Keys may be any (homogenous) comparable type

- Dictionary operations
  - Create
  - Destroy
  - Insert
  - Find
  - Remove

  ![Diagram of dictionary operations]

  **Find(Z125 Pro)**
  - Z125 Pro
    - Fun in the sun!

  **Insert**
  - Feet
    - Useful for something, presumably

  **Stumpjumper**
  - The favourite baby of VanCity planners
  - Z125 Pro
    - Fun in the sun!
  - GL1800
    - Quiet comfort
Implementing a dictionary

Using some of the data structures we have seen so far

- **Worst case complexities for Insert, Remove, Find**

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Remove</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Unordered list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Let's go back to basics – arrays!

What if we are allowed to leave gaps in our array, and we know the index of the element we want to access?
A simple example

- Suppose a company has 300 numbered lockers in its office building, and assigns a locker to every employee it hires
  - Suppose also, the company currently has 250 employees, and maintains information such as locker (employee) number, name, position, salary, etc.
  - every month when computing payroll, the company system must access an employee record, given a locker number

<table>
<thead>
<tr>
<th>Locker number</th>
<th>Employee name</th>
<th>Position, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>Jorge Lorenzo</td>
<td>...</td>
</tr>
<tr>
<td>92</td>
<td>Scott Redding</td>
<td>...</td>
</tr>
<tr>
<td>225</td>
<td>Dani Pedrosa</td>
<td>...</td>
</tr>
<tr>
<td>46</td>
<td>Katsuyuki Nakasuga</td>
<td>...</td>
</tr>
<tr>
<td>171</td>
<td>Valentino Rossi</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why not simply use an array with 300 (plus one) elements? We can do all insert/remove/find in $O(1)$ time

That's even better than a balanced BST, but how does it scale?
A larger example

• Suppose we want to store some census data on Canadians with telephone numbers
  – telephone number in the format (123) 456-7890, can be conveniently converted to a number between 0 and 9,999,999,999, let's use this as an array index
  – But, consider that the entire population of Canada is approximately 36,297,241 (as of 2016
    • Over 99.6% of the array will be empty
    • And it probably will not fit in your computer's RAM anyway

So the full-ranged array doesn't scale well, but let's hold onto that idea. What if the data we want to store don't have a convenient integer field?
• What if we had to store data by name?
  – We would need to convert strings to integer indices

• Here is one way to encode strings as integers
  – Assign a value between 1 and 26 to each letter
    – $a = 1$, $z = 26$ (regardless of case)
    – Sum the letter values in the string

"dog" = 4 + 15 + 7 = 26
loves to eat Milk-Bone
pees on the rug

"god" = 7 + 15 + 4 = 26

Insert("dog", description)  

Find("god")

We found an entry, return true!
Finding unique string values

• Ideally we would like to have a unique integer for each possible string
  – The "sum the letters" encoding scheme does not achieve this

• There is a simple method to achieve this goal
  – As before, assign each letter a value between 1 and 26
  – Treat the string as a base 26 number
  – Multiply the letter's value by \(26^i\), where \(i\) is the position of the letter in the word:
    • "dog" = 4*26^2 + 15*26^1 + 7*26^0 = 3,101
    • "god" = 7*26^2 + 15*26^1 + 4*26^0 = 5,126
  – But, suppose we store strings of length 10, there are \(26^{10}\) possible combinations
    • most of which are meaningless, e.g. "achcyertxa"
A different approach

- Don't determine the array size by the maximum possible number of keys
- Fix the array size based on the amount of data to be stored
  - Map the key value (phone number or name or some other data) to an array element
  - We still need to convert the key value to an integer index using a hash function
- This is the basic idea behind hash tables
Hash tables

• A hash table consists of an array to store data
  – Data often consists of complex types, or pointers to such objects
  – One attribute of the object is designated as the table's key

• A hash function maps a key to an array index in 2 steps
  – The key should be converted to an integer
  – And then that integer mapped to an array index using some function
    (often the modulo function)
Collisions

• A hash function may map two different keys to the same index
  – Referred to as a collision
  – Consider mapping phone numbers to an array of size 1,000 where \( h = \text{phone mod 1,000} \)
    • Both 604-555-1987 and 512-555-7987 map to the same index
      \( (6,045,551,987 \mod 1,000 = 987) \)

• A good hash function can significantly reduce the number of collisions

• It is still necessary to have a policy to deal with any collisions that may occur
  – Collisions are actually unavoidable due to pigeonhole principle
Collisions

• **Pigeonhole principle (informally)**
  – Try to fit $k + 1$ pigeons into $k$ pigeon-sized holes
  – Try to get 33 CPSC 221 students into a lab section with 32 seats
  – Try to hash without collision $m$ keys into $n$ array indices with $m > n$

• **Formally:**
  Let $X$ and $Y$ be finite sets where $|X| > |Y|$
  If $f : X \rightarrow Y$, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x_1 \neq x_2$
Pigeonhole principle

Formally

• Let $X$ and $Y$ be finite sets with $|X| = n$, $|Y| = m$, and $k = \lceil n/m \rceil$

• If $f : X \to Y$, then $\exists k$ values $x_1, x_2, \ldots, x_k \in X$ such that $f(x_1) = f(x_2) = \cdots = f(x_k)$
  
  – Informally, if $n$ pigeons fly into $m$ holes, at least 1 hole contains at least $k = \lceil n/m \rceil$ pigeons

• Proof (by contradiction):
  
  – Assume there is no such hole. Then, there are at most $(\lceil n/m \rceil - 1) \times m$ pigeons in all the holes combined.
  
  – This is fewer than $(n/m + 1 - 1) \times m = n/m \times m = n$, which is a contradiction
Exercise

Pigeonhole principle examples

1. Suppose we have 5 colours of Hallowe'en candy in a bag.
   a. How many pieces of candy do we have to take from the bag if we want to be certain to get 2 of the same colour?
   b. If there are 1000 pieces of candy of each colour, how many pieces do we need to take to guarantee that we will get two red pieces of candy (assuming that red is one of the 5 colours)

2. If 5 points are placed in a 6cm × 8cm rectangle, argue that there are two points that are not more than 5cm apart

3. For $a, b \in \mathbb{Z}$, we write $a$ divides $b$ as $a \mid b$, meaning $\exists c \in \mathbb{Z}$ such that $b = ac$
   - Consider $n + 1$ distinct positive integers, each $\leq 2n$. Show that one of them must divide one of the others.
   - Hint: any integer can be written as $2^k \cdot q$ where $k$ is an integer and $q$ is odd. e.g., $129 = 2^0 \cdot 129; 60 = 2^2 \cdot 15$
Readings for this lesson

• Thareja
  – Chapter 15.1 – 15.2 (Introduction to hash tables)

• Next class
  – Thareja Chapter 15.3 (Hash functions)
  – Chapter 15.5.1 (Open addressing)