UCF Stands For Opportunity

COP 3502

Slides 11/21

SCHOOL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE

Topics

- **Hash Tables**
- **(Recommended reading: Chapter 11)**

Hash Tables

- Goal: Fast insertion and fast lookup
- The approach: Have a big array, and put elements into it at indices based on their values
- Performance (if things go well):
  - O(1) insertion
  - O(1) lookup
  - O(1) deletion

The Hash Function

- The hash function is how the hash table decides where to put elements and where to find elements when looking them up
- Basic procedure for inserting element x into a hash table (h is the hash function):
  - Compute h(x)
  - Insert x at position h(x) in the array
  - This has some problems that we'll talk about in a minute (hash collision)

The Hash Function

- **Required properties of the hash function:**
  - **The domain of the hash function must be the entire set of possible elements that might go into the hash table**
  - **The range of the hash function must be integers representing valid indices in the array**
  - **The output of the hash function must be deterministic (always the same for a given value)**

The Hash Function

- Desirable properties of the hash function
  - Output should appear to be random
    - Note that it can't actually be random because we need elements to map to the same position every single time
  - Output should be evenly distributed over the range of indices
  - The hash function should easily generalize to arrays of arbitrary size

The Hash Function

- Examples of bad hash function for strings
  - Use the ASCII value of the first character
  - Sum the ASCII values of all the characters
- Not necessarily a great approach, but it's a whole lot better
  - Suppose string $s = c_0c_1 \ldots c_n$
  - $h(s) = (c_0 * 128^0 + c_1 * 128^1 + \ldots + c_n * 128^n) \bmod tablesize$
  - There are some issues with calculating that hash function because intermediate results could get pretty big, but with the right approach, the numbers can stay small

Hash Collisions

- No matter how well designed a hash function is, it faces the problem of <u>hash collision</u>, in which more than one value in the domain maps to the same position
  - Why? Because the domain is a whole lot bigger than the range
- Hash collisions must have a way of being resolved

Hash Collision Resolution

- **Very bad: Just overwrite values**
- **Linear probing**
  - **Keep looking through the array linearly until you find a free spot and insert at the first free spot**
  - **When doing lookup, you need to look for an empty spot in order to determine that the element isn't there**

Hash Collision Resolution

- **Linear probing**
  - **Problem: Once the array starts getting too full, you might have to look for a while to find a free spot**
  - **Problem: Clustering- values have a nasty tendency to clump together into clusters once the array starts filling up**

Hash Collision Resolution

- **Quadratic probing**
  - **Instead of looking at position+1, position+2, position+3, etc., look at position+1^2, position+2^2, position+3^2.**
  - **Significantly reduces effects of clustering**
  - **Problem: If the array gets too full, it might be impossible to find a spot, even if free spots are available**

Deletion from Hash Tables

- **You can't just take the element out if you're resolving collisions with probing**
- **Lazy deletion- Take the element out, but leave a mark indicating that something was there**
- **Stuff can be added to a spot that was lazily deleted, but when doing lookup, a lazily deleted spot doesn't count as empty.**

Rehashing

- If the hash table gets too full, you can expand it:
  - Make a new table of roughly double the size
  - Remove each element from the old table and put it into the new table based on the hash function for the new table

Hash Collision Resolution

- **Separate Chaining**
  - **Instead of only having one element at each array index, have a linked list**
  - **When inserting an element, insert it at the head of the list it maps to**