Department of Computer Science

Hash Tables

or, "What do you mean, that's as fast as it goes?"

or, "This one goes to O(11)...."

Review

- We want to store a collection of data. We want to add to, delete from, and search in the collection
- What is the average case complexity of add, delete, and search if:
  - The collection is stored as an unsorted array
  - The collection is stored as a sorted array
  - The collection is in a binary search tree

We want better performance!

- An alternative collection type is the "hash table"
  - Very good average case behavior (potentially as good as O(1)!)

Sample problem

- Suppose:
  - You want to keep track of students using their ID number
  - If IDs range from 0 to 99, you've got reasonably compact (dense) data
  - If you're using the SSN, things change:
    - This type of value is known as "sparse data"
    - The SSN becomes a "key" for retrieving the rest of the information

Mapping students into an array

- Let's say that we have an array that can hold 10 Student objects
- We're going to use SSNs to decide where the student records should go
  - We need a function that will map SSNs into valid array indices (0-9)
  - This type of function is called a "hash function"
  - One example might be: hash(ssn) = ssn % 10
  - Another might be: hash(ssn) = 1st digit in ssn

Choosing a hash function

- You want to choose a function that will provide an even distribution of the keys across the supported range of values
  - Referred to as a "uniform hashing"
- If you use a division-based hash function (i.e., remainder after division), you should use a prime number.
  - Studies have suggested that it's even better to use a number that is prime, **and** is of the form (4k+3)

Department of Computer Science

Storing data in a hashed array

- In the simplest case, it's easy:
```
index = hash(theData.theKey);
array[index] = theData;
```
- What's the complexity (in the simplest case) for:
  - adding data?
  - finding data?
- [In-class example]

Department of Computer Science

One small problem....

- The simplest case may not apply
  - What if you have more than one value that hashes to the same spot?
    - This is called a "collision", and it's a real problem....
  - Example:
    - hashFcn(ssn) = ssn % 17;
    - But any two SSNs that are a multiple of 17 apart from each other will generate the same hash index

Department of Computer Science

Some solutions to collisions

- Linear probing
- Double hashing
- Chained hashing

Linear probing

- The concept is simple:
  - If a collision occurs during addition, just find the next empty spot and put it there
- But consider this:
  - What patterns begin to form?
  - What happens when we're searching for data later?
  - What happens when we want to delete data from the hash table?

Problems with linear probing

- Some implications:
  - Data tends to build "clusters" of data, where keys collide (or nearly collide)
  - When searching:
    - If you don't find what you're looking for at the hash point, you need to keep walking forward until you do find it (or until you run out of data)
  - When deleting data:
    - What about if you remove something in the middle of a cluster? In this case, searching can't stop looking until you've examined <u>everything</u> (unless you rehash all 'at risk' data)....

Linear probing pros&cons

- Advantages:
  - Easy to implement
- Disadvantages:
  - Performance is somewhat poor
  - Clustering reduces efficiency of the hash table

Double hashing

- Also sometimes called "rehashing"
- If there's a collision, generate a second hash value, using a different function
  - This value is used to calculate "jumps" through the table, while we look for an open spot
  - Reduces clustering; improves overall performance
- [In-class example]

Department of Computer Science

Chained hashing

- Each element in the table holds a list of values, rather than a single value
- When adding to the table:
  - Hash the key
  - Put the object into the list in array[hash(key)]

Department of Computer Science

Time analysis

- The "load factor" of a hash table is defined as:

$$\alpha = \frac{\text{number of elements in table}}{\text{size of the table's array}}$$

Searching with linear probing

- In a non-full hash-table with no removals, and using uniform hashing, the average number of table elements examined in a successful search is approximately:

$$\frac{1}{2}(1+\frac{1}{1-\alpha})$$

Searching with double hashing

- In a non-full hash-table with no removals, and using uniform hashing, the average number of table elements examined in a successful search is approximately:

$$\frac{-\ln(1-\alpha)}{\alpha}$$

Searching with chained hashing

- In a non-full hash-table, using uniform hashing, the average number of table elements examined in a successful search is approximately:

$$1+\frac{\alpha}{2}$$

Average # of elements examined while searching

Load Factor	Linear Probing	Double Hashing	Chained Hashing
0.5	1.5	1.39	1.25
0.6	1.75	1.53	1.3
0.7	2.17	1.72	1.35
0.8	3.0	2.01	1.4
0.9	5.5	2.56	1.45
1.0	N/A	N/A	1.5

Hash tables and Java

- Remember that hashing is *really* useful
  - O(1) operations are the Holy Grail of computing
- Java recognizes this in two important ways
  - The JDK includes a variety of hash-related classes class in the java.util package
    - Hashtable
    - HashMap
    - HashSet
  - The Object class includes a getHashCode() method!

With great power....

- There is a relationship ("contract") defined in java.lang.Object between `equals()` and `getHashCode()`
  - If you override one, you should (must!) override the other, or unpredictable results may occur
  - The contract is described in the JavaDocs for `getHashCode()`

When to use what for storage (part 3)

- Hash tables are good when:
  - You need frequent capacity changes in the data structure
  - High-speed manipulation (especially searches) is important
  - You're concerned with key values, rather than positional data
  - You can deal with the risk of rehashing when the table changes

R·I·T
Department of Computer Science

Any questions?