Hashing and Hash Tables

Weiss, chapter 20 (pages 773–806)
Hashing

• Implement Map
  – Association list: linear time lookup
  – Tree: logarithmic lookup
  ➢ Array: constant time lookup
    ▪ Keys are indices and values are elements of the array
    ▪ Array is called hash map

• Example
  – Map product IDs to inventory sizes
  – Problems
    • Array has fixed size
    • Array list wastes a lot of space
Hashing (cont.)

• Solution
  – Use modulo operator with the fixed size array
  – E.g., mod key by 10:
    • Values 0, 10, 20, 30, ... map to index 0
    • Values 1, 11, 21, 31, ... map to index 1
      [ null, null, v1, null, null, v2, null, null, v3, null ]
      key232 key555 key8

• Problem
  – Store an entry for key 1992 (collision)
  – Two keys use the same position in the array
Probing

• Search for an open position to store a value
  – Start from the original index of key
  – Linear probing
  – Also store key in array cell
  – Put k8 (target key)

[ null, null, k236, null, null, k555, null, null, k18, null ]
[ null, null, k236, null, null, k555, null, null, k18, k8 ]

• Cost of lookup and put operations
  – $O(1)$ without collision
  – $O(N)$ with collision
Chaining

• Probing is not efficient when hash array is dense.
• Store keys, that hash to the same index, in a list under a cell array
  – Resolve collision by chaining
• Look up through the list of a cell
• Number of keys may be much more than array size.
  – Double the array size and rehash keys
Hash Function

• Transform keys of some data type to integers
  \[ h : T \rightarrow \text{int} \]
  \[ h( c ) = 0 \]

Degrade the performance of map operations

```java
public int hash( String key, int arraySize ) {
    int hashVal = 0;
    for( int i = 0; i < key.length(); i++ ) {
        hashVal += key.charAt( i );
    }
    return hashVal % arraySize;
}
```

distribute keys ineffectively
Hash Function (cont.)

\[ h( c ) = \text{rand}( 0, 100 ) \]

- Two separate call to \( h( c ) \) returns different indices.
- Hash values are not consistent.

- Hash functions need to
  - Evenly distribute keys
  - Consistently assign hash values: \( k1 = k2 \to h( k1)=h( k2 ) \)
Hash Functions for Objects

• Compute hash values for objects of a class type
  public int hashCode( ) { }  // returns memory address of objects
• Compute hash values over the fields of objects
• A class that overrides the equals method needs to override the hashCode method.
• A strategy suggested by Joshua Bloch:
  public int hashCode( ) {
      int result = a random non-zero integer;
      for each field f in this object {
          int c = f.hashCode( );
          result = 31 * result + c;
      }
      return result;
  }
Hash Function Applications

• Compilers use symbol table to trace declared variables in programs

• Speller checkers use hash tables to detect misspellings.

• Find anagrams