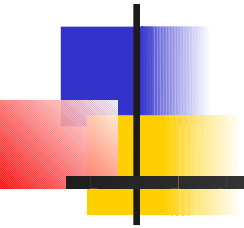
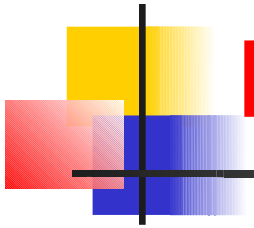


Fast Searching and Memoization

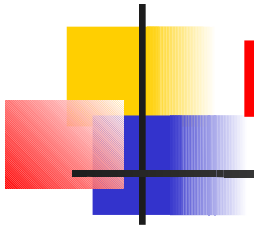
Corky Cartwright
Department of Computer Science
Rice University



Hashing: Motivation

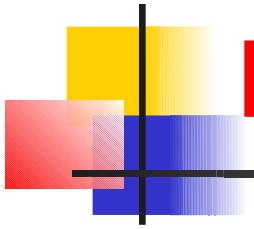
- Consider the problem of counting the numbers of each kind of char in a file.
  - If chars are represented by 8-bit bytes (the usual perspective of pre-Java langages), then we can index a 256 entry table by character code.
    Idea: convert chars to unique table indices.
  - If chars are represented by 16-bit Unicode (UTF-16), we can still use this approach. It only requires an array of counters of size 64K (2**16).
  - But what it we want to generalize our application to process characters encoded using 32-bit unicode (UTF-32). No longer practical to use direct mapping of a char to its binary representation for a table index.

How can we handle UTF-32?.



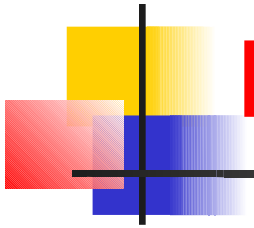
Hashing: Motivation cont.

- Consider the similar but more interesting problem of counting the number of occurrences of each word in a huge text file. We can easily parse the input stream into words (assuming we can agree on the set of delimiter characters). But how we represent and manage the table recording the words we have seen.
- Key idea: using a “scrambling” (*hash*) function to map large chars (UTF-32) or strings (words) to indices in the range [0, N-1] where N is approximately equal to the size of the longest file (measured in the items we are counting) we expect to process.



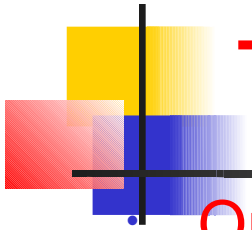
Hashing Functions

- Standard practice: hash functions yield an unsigned binary number in same format as machine addresses (formerly 32-bit binary but morphing to 64-bit binary).
- Address-sized hash codes are easily mapped to indices in the range [0, N-1] by the remainder operation (a by-product of machine division). If N is a power of 2, then remainders can by computed by shift operations (which are extremely fast), but there are theoretical advantages to using a prime number for N. (Perhaps not worth it in practice.)



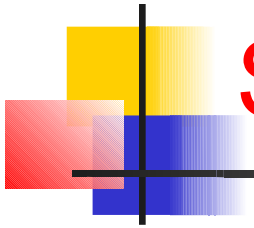
Hashing Functions

- Devising good hash functions is an art (lots of pages on the web and *some* of them are technically sound)
- Rules of thumb:
  - The hash code for an object must be consistent with equality. (Equal objects *never* hash to different codes.)
  - Hashing mutable objects is insane unless you are using object identity as the definition of object equality. (`defaultHashCode()` in Java has this property. Can also use `IdentityHashMap`.
  - The hash code for an object should depend on *all* of the fields of the object.
  - Exclusive-or is a good way to combine hash codes because it directly depends on all bits, yet is very cheap.
  - Computation of hash code should be cheap, although some extra expense is justifiable if the hash code is cached with the object.
  - Achilles heel of hash functions: aliasing (unequal objects mapped to same code).



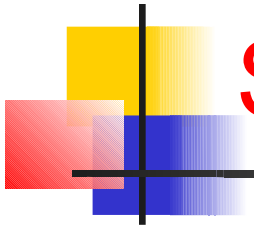
Two basic approaches to hashing

- Open addressing: all counters are stored directly in table. Collisions force reprobing which must be deterministic. Simple scheme is linear probing. But in practice, forget open addressing. No significant advantage over direct chaining except in unusual situations.
- Direct chaining ("bucket hashing") Table consists of an array (block) of linked list headers. There is a linked list for each hash code value. Actual hash entries are stored as separate objects in an auxiliary area (usually the heap). Only significant weakness is less locality because object locations are scattered across the heap. (Can be mitigated by allocating objects within array blocks stored in heap, but this is a big book-keeping hassle.)



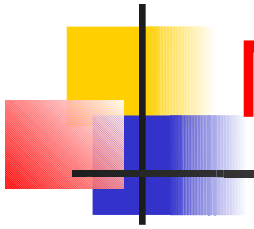
Sample Hash Table Code

- The **`MyHashMap`** class uses a direct-chained hash table to implement exactly the same **`MapI`** interface as **`OOTreeMap`**. It is straightforward but ugly; it is classic procedural code encapsulated as a Java class to hide the procedural details. From the client's perspective, there is no way to detect that the implementation is procedural. The linked list **`Node`** class is a private nested class.
- Why did I use procedural coding in writing **`MyHashMap`**? For a simple data structure like **`MyHashMap`**, the procedural code is tractable and significantly more efficient that OO code (which is important in a library). In Java software development, almost nobody writes hash table implementations anymore! Everybody uses **`HashSet`**, **`HashMap`**, **`ConcurrentHashMap`**, and **`IdentityHashMap`**. (**`HashTable`** is obsolescent because all of its methods are **`synchronized`**). If procedural code is easily encapsulated, significantly more efficient, and important to an applications overall efficiency, then I have no objection to writing procedural code. But note that the conjunction of these criteria doesn't arise very often.



Sample Hash Table Code II

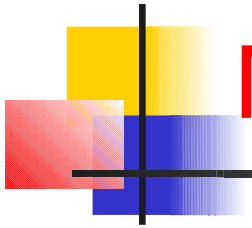
- Each bucket is a singly linked list. Within a bucket, linear searching is necessary. (If the keys are ordered, buckets can, in principle, be search trees of some form, but why bother? Simply make the hash table bigger to reduce the average size of each bucket.)
- Optimization trick in cases where load factor is high: move last referenced item to front of list on each access. (I did not bother with this optimization, because it only makes sense when buckets get large, which this implementation prevents.)
- Large load factors should be avoided if possible. **`MyHashMap`** never lets it get above 1.0. In an application written in a high-level language (not C/C++), it is almost always possible. Why?
- When the table gets full, double the table size and rehash! **`MyHashMap`** does this and it only takes about 10 lines of code. The asymptotic cost is zero! (Why? The sum of 2^k, k = 0, … N-1 = 2^N-1.)



Memoization

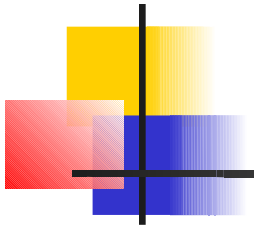
- Key idea: avoid recomputing the solution to a subproblem that has already been solved.
- Key technique: brute force. Keep a hash table mapping subproblem descriptions to subproblem answers.
- Simple illustration: naïve Fibonacci function.

```
class MyMath {
  static long fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
  }
}
```



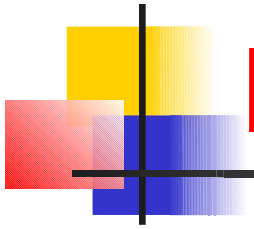
Memoization cont.

```
import java.util.HashMap;
class BetterMath {
  static HashMap<Integer, Long> Fib =
    new HashMap<Integer, Long>();
  static long fib(int n) {
  if (n <= 1) return 1;
  else {
    Long cachedAnswer = Fib.get(n);
    if (cachedAnswer != null) return cachedAnswer;
    else {
      long newAnswer = fib(n-1) + fib(n-2);
      Fib.put(n, newAnswer);
      return newAnswer;
    }
  }
}
```



More Challenging Examples

- The best known solutions to many standard computational problems can be formulated as the memoization of naïve solutions.
- Memoized algorithms correspond to a problem solving technique called *dynamic programming*.
- Examples:
  - parsing CFGs (CYK algorithm),
  - optimizing the multiplication of a chain of matrices
  - shortest path between two nodes in a graph, ...
  - many string algorithms
- Lots of information on the web on dynamic programming



For Next Class

- Exam II over OO material will be given at scheduled site on Friday, April 30.
- Parallel sudoku homework due Friday. **Go to lab.** Have fun.
- Wedneday's lecture will discuss tradeoffs in designing parallel implementations.
- Friday's lecture will review the Java portion of the course to help you study for Exam II.