Lecture 6:
Hashing

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Dictionary / Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- *Search(S, k)* – A query that, given a set S and a key k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.

- *Insert(S,x)* – A modifying operation that augments the set S with the element x.

- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key.

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.

- Logical *Precessor(S,x), Successor(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

Problem of the Day

You are given the task of reading in n numbers and then printing them out in sorted order. Suppose you have access to a balanced dictionary data structure, which supports each of the operations search, insert, delete, minimum, maximum, successor, and predecessor in $O(\log n)$ time.

- Explain how you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: minimum, successor, insert, search.

- Explain how you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: minimum, insert, delete, search.

- Explain how you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: insert and in-order traversal.

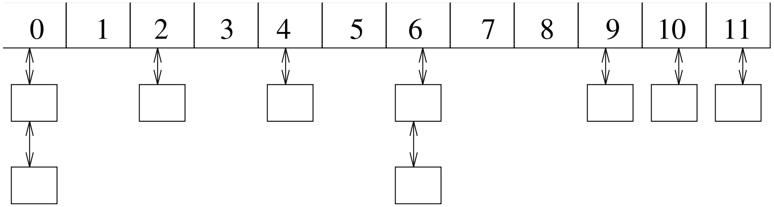
Hash Tables

Hash tables are a *very practical* way to maintain a dictionary. The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index.

A hash function is a mathematical function which maps keys to integers.

Collisions

Collisions are the set of keys mapped to the same bucket.
If the keys are uniformly distributed, then each bucket should contain very few keys!
The resulting short lists are easily searched!



Chaining is easy, but devotes a considerable amount of memory to pointers, which could be used to make the table larger.

Hash Functions

It is the job of the hash function to map keys to integers. A good hash function:

1. Is cheap to evaluate
2. Tends to use all positions from $0 \ldots M$ with uniform frequency.

The first step is usually to map the key to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

Modular Arithmetic

This large number must be reduced to an integer whose size is between 1 and the size of our hash table.

One way is by $h(k) = k \bmod M$, where M is best a large prime not too close to $2^i - 1$, which would just mask off the high bits.

This works on the same principle as a roulette wheel!

Bad Hash Functions

The first three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

Good Hash Functions

The last three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

Performance on Set Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary. However, the worst-case time is unpredictable.
The best worst-case bounds come from balanced binary trees.

Hashing, Hashing, and Hashing

Udi Manber says that the three most important algorithms at Google are hashing, hashing, and hashing.

Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.

- *Is part of this document plagerized from part of a document in a large corpus?* – Hash overlapping windows of length w in the document and the corpus. If there is a match of hash codes, there is possibly a text match.

- *How can I convince you that a file isn't changed?* – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.

Substring Pattern Matching

Input: A text string t and a pattern string p.

Problem: Does t contain the pattern p as a substring, and if so where?

E.g: Is *Skiena* in the Bible?

Brute Force Search

The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character.

This runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$.

String Matching via Hashing

Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t.

If these two strings are identical, clearly the resulting hash values will be the same.

If the two strings are different, the hash values will *almost certainly* be different.

These false positives should be so rare that we can easily spend the $O(m)$ time it take to explicitly check the identity of two strings whenever the hash values agree.

The Catch

This reduces string matching to $n - m + 2$ hash value computations (the $n - m + 1$ windows of t, plus one hash of p), plus what *should be* a very small number of $O(m)$ time verification steps.

The catch is that it takes $O(m)$ time to compute a hash function on an m-character string, and $O(n)$ such computations seems to leave us with an $O(mn)$ algorithm again.

The Trick

Look closely at our string hash function, applied to the m characters starting from the jth position of string S:

$$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

A little algebra reveals that

$$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

Thus once we know the hash value from the j position, we can find the hash value from the $(j+1)$st position for the cost of two multiplications, one addition, and one subtraction. This can be done in constant time.

Hashing as a Representation

Custom-designed hashcodes can be used to bucket items by a cannonical representation.

- Which five letters of the alphabet can make the most different words?
- Hash each word by the letters it contains: $skiena \rightarrow aeikns$! Observe that *dog* and *god* collide!

Proximity-preserving hashing techniques put similar items in the same bucket.

Use hashing for everything, except worst-case analysis!