CMPSCI 187: Programming With Data Structures

Lecture #32: Introduction to Hashing
2 December 2011

- The Hashing Idea
- Collisions
- Some Hashing Functions
- Collisions Resolved by Chaining
- Collisions Resolved by Open Addressing
- Variants of Open Addressing

The Hashing Idea

- If I have a multivolume encyclopedia on my shelf, it is very easy to find volume 7 because it is always in the same place on the shelf and I put it back there when I am done. Each volume has its own address space.

- If I can afford to have an address for every item that *might ever be* in my collection, I can insert, delete, and find any item in O(1) time. Box numbers in a post office work this way -- it is why we assign small numbers to data.

- The problem, of course, is that the **address space** is normally much larger than the **storage space** we have to allocate.

- A **hash table** is a small address space, of the same size as our available storage, with a **hash function** that maps each address in the actual address space to an address in the hash table. The hash function is not **one-to-one**.

A Hashing Example

- Let the address space be names, strings over the 26-letter alphabet.
- A simple hash table would have 26 addresses, one for each letter. We could let the hash function of a name be its first letter. This works well until I have two names with the same first letter -- a **collision** in the hash function. I can convert each name to a hash address, but if I have a hash address I don't know which real address was mapped to it.
- What if I take the first three letters as my hash address? I may still get collisions -- this class contains several pairs of students with the same first three letters and in fact two pairs of students with the same name. And though there are 26^3 = 17576 three-letter sequences, most of them will never be used (and some like SMI or JON in this country will be very common).
- A better hash function might avoid collisions entirely or make them rare.

Collisions

- Is it possible to avoid collisions entirely? Yes, if the set of addresses to be hashed never changes and is smaller than our set of hash addresses -- for example, the set of reserved words in a programming language. *If* we can find a function that never maps two of these addresses to the same place, we get what is called **perfect hashing**.

- But in general, all we know in advance is that our addresses will come from some large set, and we have to decide on our function before we know what they are.

- We would like our function to behave “randomly” as much as possible -- if the hash codes of our addresses were each chosen randomly from the hash addresses, we could calculate the probability of a given number of collisions. You’ll do this in CMPSCI 311. The initial-letter function was far from random, and therefore was likely to have more collisions.

Some Hashing Functions

- Here is a good way to get a hash function that behaves "randomly". Take the datum and generate a large `int` value x. Then choose a number p to be the size of your hash table, and let the hash address of the datum be `x % p`.

- It is best to have p be a **prime number**, for reasons we'll explore more in CMPSCI 250. For a taste of this, suppose that p were even and most of the x values were odd. Then most of the numbers x % p would be odd -- they would not be random in that sense.

- How to get x? Since the content of any register is ultimately an integer value, there are many ways to do it. We can **extract** an `int` value by taking some of the bits of the datum, as long as we are extracting from a part that varies for different data in the address space. We can use arithmetic operations -- multiplying one part of the datum by another, for example.

- The key goal is to get a value that has *nothing to do with* the datum's meaning.

Resolving Collisions by Chaining

- What can we do when more than one address goes to the same hash value? The simplest method is called **chaining** and consists of keeping all the addresses for a given hash code in a linked list.

- If we used our first-letter hash function and were storing names of dogs, how would we decide whether “Balto” was being stored? We would look at the list for “B” -- if it is empty, Balto is not there, and if not, we search the list. On average, we will be searching a list only 1/26 the size of our entire set.

- If we could keep all the lists to O(1) size, we would be able to insert and delete in O(1) time, better than the O(log n) for binary search trees. Even if one or two lists are large, the *average* size of a list is the number of addresses stored divided by the number of hash addresses. We usually choose the size of the hash table so that this **load factor** is less than one.

- A nice trick is to use an **overflow area** and implicit pointers to store the lists.

Resolving Collisions by Open Addressing

- If our load factor is less than one, there are open spaces all over the hash table if we can just find one. If entries x and y both map to hash value h, we could store one of them at h and the other in h+1, if the latter position is vacant. If it is not, we could use h+2, or h+3, until we find an available place.

- When we are looking for an entry, we compute its hash value h, then look at locations h, h+1, h+2, and so forth until either (a) we find the item or (b) we find an open position and can give up. In effect we are simulating a linear list in the table, and are making a linear search for our item.

- There is an open location somewhere unless the hash table is full, and if the hash function behaves randomly and the table is only half full, we would expect most of our simulated lists to be small, and their average size would be O(1). But this **linear probing** has a problem -- different lists that overlap effectively merge to make even larger lists.

Variants of Open Addressing

- Linear probing places the i'th address mapped to hash code h in location h + i, a linear function of i. We could use a quadratic function, such as mapping it to h + i^2, instead. This will reduce clustering by reducing interaction between different lists. If h + i^2 happens to be equal to h' + j^2, the lists for h and h' have intersected. But the next location in the h list is h + (i+1)^2, and the next location in the other list is h' + (j+1)^2, a different number. This is called **quadratic probing**.

- A similar system is **double hashing**, where we choose a different linear function for each item using a second hash function. If x maps to h and location h is occupied, we try locations h + s, h + 2s, h+3s, and so forth, where s is the value of the second hash function on x.

- Note that open addressing does not allow us to delete elements because the implicit linear list would be broken and elements in it would be stranded.