Lecture 9: Hash Tables

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Value-oriented data structures

Consider the data structures you have already met (arrays, stacks, queues, etc).

All have basic data management operations: insert, delete, search.

However, most are *position-oriented*—their operations have the form:

- insert into cell i, or at the top of the stack
- delete from cell j, or from the front of the queue
- retrieve what's in cell k, or at the top of the stack

But many simple tasks involve values, not positions; e.g. find Brendan's phone number.

You've seen one value-oriented data structure in COSC241 - the heap. Over the next 4 weeks, we'll be looking at three others: hash tables; binary search trees (BST); red-black trees (a type of BST).

A table data type

In a table every ‘object’ or ‘record’ (represented by a row) has some key that uniquely identifies it. E.g.

ALEC	8299
SHARLENE	8581
ANDY	8314
BARRY	5691
MIKE	8588
CAROL	8578
LANA	8580

Implementations?

There are a few possibilities for implementing such a thing:

Array: Simple: can store keys in sorted order and use binary search to find a key. Search is $O(\log n)$, but insert is $O(n)$.

Binary Search Tree: Implementation more difficult. Search is $O(\log n)$ and insert is $O(\log n)$.

Hash table: Use our key (first name in the previous slide) to compute an index or address. Search and insert can be $O(1)$ if the table is big enough and the hash function is a good one.

Hash functions

Suppose our keys are already numbers.

The simplest hash function, called *direct addressing*, is $h(k) = k$. If you have the keys 5, 3, 8, 9, 6 then you could insert key i into position A[i] of an array A of length 10.

But if your keys are 20 digit numbers, you'd need an array of length 10^{20}. Most likely you have far fewer than 10^{20} records. So direct addressing is usually a bad idea.

Division hashing

Use $h(k) = k\%m$ where m is the size of the array, k the key, and $k\%m$ is the remainder after dividing k by m.

0		
1		
2		
.		
25	001364825	Whatever data goes with this key
.		
97		
98		
99		

What's the problem?

Collisions

A good hash function has few collisions.

Rule of thumb: Choose the table size m to be a prime not too close to a power of 2 (e.g. 37 rather than 31).

If $m = 2^p$ and key k is written in binary then $h(k) = k \mod m$ is just the p lowest-order bits of k. Unless we know that all the low-order p-bit patterns are equally likely, we'll get a more even distribution by using a hash function that depends on all bits of the key.

We will still need a collision resolution strategy, because perfect hash functions are rare.

Keys that are strings

The key field may be of any ordinal type, including character strings. If key values are strings, we need a way to convert them to numbers first. A reasonable transformation takes into account the position of each character, say by multiplying the ASCII values by a number raised to a power that reflects the position.

E.g. if we multiply the first character by the most and the last character by the least, we could define T so that:

$$\text{T(ALAN)} = 0 \cdot 2^3 + 11 \cdot 2^2 + 0 \cdot 2^1 + 13 \cdot 2^0 = 57$$

Strings up to length about 25 would fit into a 4-byte integer. What about longer strings? Normally, we would use a BigInt or equivalent and interpret a string as a base 128 number:

$$\text{T(ALAN)} = 65 \cdot 128^3 + 76 \cdot 128^2 + 65 \cdot 128^1 + 78 \cdot 128^0 = 9811.$$

Linear Probing

Given key k, first try the home cell h(k).

If already occupied, examine h(k) + 1, h(k) + 2, etc., wrapping around the array until an empty cell is found or the whole array examined.

If you like, we're defining a new hash function

$$H(k, i) = (h(k) + i) \% m$$

where k is the key, i is the number of collisions so far, and m is the array's capacity. The home cell is given by H(k, 0).

In class example.

Limitation

Linear probing suffers from primary clustering.

We get clusters because the steps are small and all the same size and all ignore the key.

Retrieving then requires sequential search through the cluster. Also, clusters tend to coalesce (bad news).

But if we have few collisions, clusters stay small and linear probing is good enough.

Quadratic Probing

An old idea to avoid clustering is to make the step size quadratic (gets bigger with each step), e.g. use the new hash function

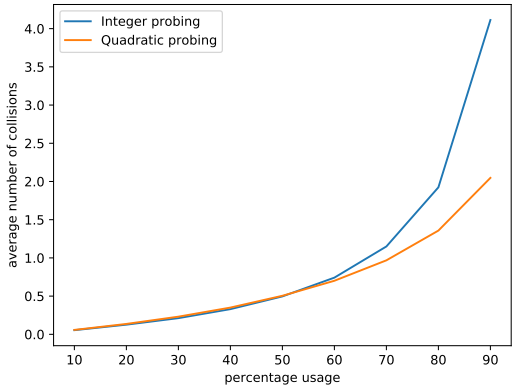
$$H(k, i) = (h(k) + i^2) \% m$$

where h is the original hash function and i the number of collisions so far. (Drawbacks: secondary clustering and space wasting.)

In class example.

Linear vs quadratic

Proofs are good, but sometimes experiments are good enough: table size is 997; random numbers from 0 to 1000000; 0 to 90% occupancy; repeat 1000 times; how many collisions on average?



Relevant parts of the textbook

Hash functions are discussed in section 11.3, with division hashing in 11.3.1.

Linear and quadratic probing are discussed in section 11.4 (called open addressing).

Exercises

1. Insert 89, 18, 49, 58, 69, 78, with h(k) = k % 10, using first linear then quadratic probing.

linear

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

quadratic