

Data Structures G5029

Lecture 8

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

Lecture 8

- Hash tables ...

Hash tables

- We saw in lectures 5,6 & 7 that binary trees and heaps offer a way to organise large amounts of data for indexing and searching
- They do this by imposing structure on the data. The structure determines how efficiently we can index and search
- $O(\log_2 N)$ for binary search
- The aim of hash tables is to reduce the complexity to constant time

Hash tables

- Suppose all our data items are stored in some array (probably quite big)
- Suppose it was possible to compute, directly or indirectly, the address where an item would be stored – independently of the number of items being stored
- We could then use this procedure to:
 - test for the presence of an item (calculate where it should be and look for it)
 - put the item in our array (calculate where it should be and put it there)

Hash tables

- If we can do this, then indexing and searching will always take a constant time
- The function to compute where the item should be stored is called the **hashing function**
- And the array of data is called a **hash table**
- Not the same as “making a hash of it”. That’s the result of bad programming practices ...

Hash tables - applications

- Anywhere where we have a lot of data to be stored and indexed, especially if the data is not structured
 - Electronic dictionaries
 - Indexes (e.g. looking up telephone numbers from address queries)
 - Search engine databases (e.g. recovering web sites addresses from key words)
 - The list goes on ...

Hash functions

- Unordered data in an array of length N ...



- If we want to see if “ox” is in the table:
 - Calculate $h(\text{“ox”})$
 - $h(\text{“ox”}) == 2$ by some (as yet unspecified) method
 - If $\text{array}[2] == \text{“ox”}$, it’s present
 - If $\text{array}[2] != \text{“ox”}$, it’s not in array

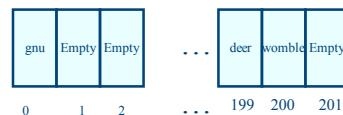
Seems easy – what’s the catch?

- Different items should hash to different integers (e.g. if “ox” and “womble” hashed to the same integer x , then if “ox” was in array then we can’t store “womble” if we wanted to)
- Unique integers is unrealistic
- Suppose want to store strings up to 16 chars in length. Even assuming just 26 lower case characters there are still $26^{16} = 43,608,742,899,428,874,059,776$ possible strings to be stored. That’s a rather large array ...

How do we solve this ?

- So we need to develop a practical system
- Need a method for dealing with cases where two objects in the hash table map to the same array index
- We call this collision resolution
 - Linear probing
 - Quadratic probing
- Will look at computing the hash function later

Linear probing



"gnu" and "deer" are already in the array.

Now we want to add "buffalo" but $h(\text{"deer"}) == h(\text{"buffalo"}) = 199$.

Can't put buffalo in that slot so we look at next slot 200. But "womble" is in that slot.

We find that the next slot again is empty (201) so we put "buffalo" there.

Linear probing

- Using this technique we are always bound to find a space to store something in the array provided that it is not completely full
- If # of entries == table size, we need to grow the table
- Does introduce one practical issue: how do we know when to stop probing?

Linear probing

- The number of probes we will need to make will depend on how full the table is
- The proportion of the has table that is occupied is called the Load Factor
- Common default for Load Factor is 0.5
- It is should be intuitive that, using probing, we will obtain clusters of sequentially occupied locations in the array. This problem is called primary clustering
- The larger a cluster becomes, the more comparisons we shall have to make on average to reach its end

Quadratic probing

- An alternative to linear probing is quadratic probing
- We still explore a sequence of locations until an empty one is found, but instead of exploring $h, h+1, h+2, h+3, h+4 \dots$ we explore $h, h+1, h+4, h+9, h+16 \dots$ i.e. $1^2, 2^2, 3^2, 4^2 \dots$
- Increments are taken modulo array length as before

Quadratic probing

- Guaranteed insertion provided that:
 - The length of the array is a prime number
 - The load factor is less than 0.5
- Why does the array length need to be a prime number?
- Free from primary clustering, but still liable to secondary clustering
- If two keys “collide” the same probe sequence will be followed for both

Hashing function

- A function $h(n)$ that turns your object n (e.g. a word to be looked up in a dictionary or search item) into an integer value
- An ideal hash function has an equal probability of $h(n)$ taking any of the values $0 \rightarrow$ array length (uniform probability distribution)
- Hash function should be computable as rapidly as possible

Hashing function

- For example, the hash code for the string hello could use the ASCII values of its chars

h	e	l	l	o
104	101	108	108	111

$$99162322 = 104 \cdot 31^4 + 101 \cdot 31^3 + 108 \cdot 31^2 + 108 \cdot 31 + 111$$
$$h(\text{"hello"}) = 99162322$$

Hashing function

- Generally use 32 bit modulo arithmetic ignoring overflow
- For 32 bits, max signed value we can represent is $2^{16} - 1 = 2147483647$
- Add 1 to this value and we get -2147483648

Double hashing

- An alternative to linear probing. Now we explore a sequence of probes $h, h+k, h+2k, h+3k \dots$
- hash functions h and k both depend on the item we want to insert
- Even if two items hash to the same h , they will not produce the same k so that different probe sequences will be followed
- Solves the secondary clustering problem

Next time ...

- Graphs ...