Computer Science 1MD3

Lab 7 – Hashing

Often we store things like employees, dates or other things in arrays. However, up until now the data we were using has had a one to one correspondence with its array position. That is, employee one would be at array position one or December would be at array position 12. In reality we don't have this situation; instead we are given things like names, or id numbers which don't have a natural ordering. To deal with this problem we employ hashing which essentially creates this one-to-one correspondence by converting things like names to integers that will fall within the array bounds.

THE MAGIC FUNCTION

Let's consider the following names: Paul, Saeed, Dai, Tina, Kevin, and Jeff. Also suppose that we have a function which takes any of these names and returns an array position in this fashion: `f(Paul)=0`, `f(Saeed)=1`, `f(Dai)=2`, `f(Tina)=3`, `f(Kevin)=4`, `f(Jeff)=5`. In some sense, this function is "magical" because it seems to know some way to have these names map to a specific array element. So say I wanted some information out of this array about Tina, I could say: `A[f(Tina)]`, which is very handy.

Now, seeing that we are not magicians, we will actually have to find a way to model a function so that `f(Tina)` will be a number between zero and five, and thus cause Tina to fall in the array. First of all to guarantee that we will always have a number between zero and five, we will take the modulus of 6 to any result of our function. Any number modulus six will always be a number between zero and five, respectively representing their remainder after division by six.

The task of converting a name to a number is a fairly easy one since all characters in C can be converted into their corresponding ASCII values. For simplicities sake we will only take the ASCII value of the first character in each person's name: `P=80`, `S=83`, `D=68`, `T=84`, `K=75`, `J=74`. Where:

Hash function

```
f(name)=ASCII of first character % 6
```

which corresponds to `f(Paul)=2`, `f(Saaed)=5`, `f(Dai)=2`, `f(Tina)=0`, `f(Kevin)=3`, `f(Jeff)=2`.

You may have noticed that several of these names under this particular hash function map to the same array position, this is called a *collision*.

COLLISION

A collision is when two or more entities hash to the same value in the array. There are many ways to deal with these collisions; in fact this is a broad subject of study. Some typical resolutions are:

Resolving Collisions

Linear – If a collision occurs go to the next available empty array position.

Double Hash – Apply the hash function, or a new hash function, again to produce a different array position.

Linked List – Create a linked list at every array position so that if a collision occurs the value is just added to the linked list

We will only cover *double hashing* in this lab because it will be your responsibility to implement the other resolutions on your next assignment.

DOUBLE HASHING

Double hashing resolves collisions by "re-hashing" the value, or, in other words, taking the result of the first hash and hashing it again.

Going back to our example, our secondary hash function may be identical to the one we used earlier except we will use the second character instead of the first. Our `f2` values may look like this: `f2(Paul)=4`, `f2(Dai)=4`, `f2(Jeff)=5`, which in fact results in even more collisions. So the claim that we don't have a very good hash function would be fair.

OPTIMAL HASH FUNCTION

It is most desirable to have a hash function which distributes array position evenly in a one-to-one correspondence. This would reduce the amount of collisions we would have to deal with by quite a bit. Investigating the nature of this problem, we come to some interesting conclusions.

Consider our initial example. We immediately limited our array to length five. So statistically, it is very likely that we will have collisions. If instead we had used an array of length ten with corresponding hash function `f(name)=ASCII of first character % 10` and `f2(name)=ASCII of second character % 10` we would instead have the following hash values: `f(Paul)=0`, `f(Saeed)=3`, `f(Dai)=8`, `f(Tina)=4`, `f(Kevin)=5`, `f(Jeff)=`**`4`**`->f2(Jeff)=2`. This is now a completely resolved hash table.

Conclusion 1

The ratio of the number of elements to array size is directly proportional to the amount of collisions. Where a number much less then one implies few collisions and a number close to one implies many collisions.

i.e. if we have ten elements and we are hashing them into an array of length one thousand (`10/1000=0.01`) we are not very likely to get collisions. Conversely, if we are hashing five elements into an array of length five (`5/5=1`) we are very likely to get collisions.

Consider a different example, where we would like to hash everyone's student number by taking the first two digits and applying some function. Even if we had a perfect hash function we would receive a tremendous amount of collision due to the fact we will always be hashing a value like `02`, or `03`. A better approach would to take the *last* two digits of everyone's student number because they are much more likely to be different.

Conclusion 2

The distribution of the input data is correlated to the amount of collision. Where sparsely distributed data produces few collisions.

So when designing a hash function you must consider what a reasonable amount of memory would be to hold your data, and how the data is represented. It is then the task to find a hash function which will best distribute this data in your array.

SELF TEST PROBLEMS

1. Why would `f(x)=x%2` be a bad hash function?
2. What are some causes of collision?
3. What is an example of good input data?
4. What is an example of bad input data?
5. Why do we use hash functions?
6. What is a collision?
7. What are some ways to resolve collisions?
8. Implement a hash table to index the first two sentences in this lab.