CSE 225 (MMR4) - Hashing techniques

If we are writing a program that needs a data structure which is very good at insertion and searching, we have three choices so far. Arrays can be searched quickly O(log n), but do not do very well at insertion O(n). Linked lists are good at insertion O(1) (easy case, not ordered) but not at searching O(n). And tree structures are O(log n) for both but have overhead for memory usage and complexity.

One alternative is to use an array, but in a slightly different way. Instead of thinking of the array as just a sequence of boxes that we can put data in, we can choose the indexes of the array where data is stored based on the values of our search keys. Consider a data set consisting of students with unique student numbers in the range 50000-59999, where we want fast insertion and search by number. If we build an array with 10000 elements and fill them all with null values, we can choose a position for new students with a simple calculation.

array position = student number - 50000

This mapping gives a number between 0-9999 from our student numbers: perfect for an array of size 10000. If we want to search by student number, we perform the same calculation, and look at that position in the array. If that position contains null, the student does not exist, or else we find the correct student. Insertion and search are both O(1).

Unfortunately, real-world data is rarely so regularly identified. If there are a lot fewer than 10000 students (or if the range of values is much larger, say 5000000-5999999) this oversized array will waste a lot of space. If there is a pattern that can be exploited, perhaps we can adjust our calculation. For example, if all the student numbers are even:

array position = (student number - 50000) ÷ 2

For even student numbers, this maps to a number between 0-4999, allowing us to create an array with only 5000 elements. As the patterns in our search keys get more complex, the formula we use to map them into array positions get longer.

But what about the situation where there are no patterns in the data, or if there are, the range of values in the mapping is too large to use to create an array (where most of the elements will be empty anyway)? The answer is to try to come up with a mathematical expression that makes a good attempt at mapping an arbitrary value to a limited range. Such an expression is commonly called a *hash function*.

We will discuss hash functions in more detail later; for now, we will choose a common one: the modulus operator.

array position = key % array size

Given any positive integer key, the array position this calculates is in the range *[0..array size - 1]*, a valid array index. When we use a hash function to map key values to array positions, the combination of the function and the array gives us a data structure called a *hash table*.

But there is a problem. Unlike our earlier calculations, the array position we get is no longer guaranteed unique. In other words, two keys may map to the same spot in the array. This is known as a *collision*. We do not want our insertions to fail or our searches to return incorrect results when a collision occurs. Instead, we have to come up with a strategy to resolve collisions.

There are two main types of strategies for resolving collisions in a hash table. One is to choose another position in the array when we try to insert an item at a position that is already occupied. This is known as *open addressing*, and there are three different techniques that can be used for open addressing.

The *linear probing* technique will simply keep looking sequentially for the next empty spot; if the hash function selects position x for an item to insert but position x is full, it will try $x + 1$, $x + 2$, etc., until it finds an empty spot. If it reaches the end of the array, it starts back at the beginning. This process of trying different positions is called *probing*.

And a linear probe means the probe operation looks a lot like a linear search.

Searching a hash table using linear probing requires the same operation; if you look up an item by key and its key maps to position x, but x contains a different item, it will try $x + 1$, $x + 2$, etc., until it finds a match (success) or hits an empty spot or ends up back at x (failure).

A problem with linear probing is *clustering*, where clumps of elements form. If you insert a value that maps to x, but x is already full, it may find a spot at $x + 1$. But if the next value inserted maps to $x + 1$, that spot is already filled, and it has to try probing to the next item. Things get clustered together in the array. In real-world data, items tend to be inserted with like items, resulting in clustering, and causing insertion and search efficiency to fall.

One way to avoid clustering is to move ahead, but not by one position. In the *quadratic probing* technique, if the hash function selects position x for an item to insert but position x is full, it will try adding powers instead $x + 1^1$, $x + 2^2$, $x + 3^3$, etc., until it finds an empty spot. Similarly, when searching, this same sequence of positions has to be tried until either a match or an empty element are found.

Quadratic probing has the advantage of reducing clustering because the positions being probed are not close together. But it does not help in the situation where many keys map to the same position; given a lot of keys that map to x, they will all end up trying to fit in positions x, $x + 1$, $x + 4$, $x + 9$, etc., making some of them difficult to find.

The third technique for open addressing is *double hashing*. In double hashing, the offset for the probe is based on the key value itself. It is produced by a second hash function. When a key maps to position x, but position x is full, the second hash function chooses a value y. It will try positions $x + y$, $x + 2y$, $x + 3y$, etc., until it finds a spot for insertion. The goal in choosing a second hash function is to make sure it gives valid offsets (always ≥ 1), that are different for keys that produce the same results from the primary hash function (you do not want all keys that map to position x to have the same y). A common second hash function is:

$$\textit{probe offset} = c - (\textit{key} \% c)$$

Where c is a constant, a prime number smaller than the array size.

The other major strategy for collision resolution uses a linked list (simple singly-linked) at each position in the array. Initially, the lists are all empty. When an item hashes to a position in the array, it gets added to the list at that position. Collisions are resolved by simply adding more items to a list. This strategy is known as *separate chaining*.

Within each list, items are not ordered. Searching for an item involves first finding the correct linked list by choosing an array position using the hash function, and then doing a linear search of the linked list at that position.

Separate chaining has more overhead than open addressing, because a large array means many linked lists. But it has two advantages. First, the structure never fills up, because the linked lists can be as long as they need to be. Secondly, it is much easier to remove items from a separate chained hash table, because it only requires a simple linked list removal. On the other hand, removal from a hash table using open addressing would be very difficult because not all items are where they "should" be, and there are many complexities in trying to rearrange items to make up for the one that was removed. It essentially requires rebuilding the hash table.