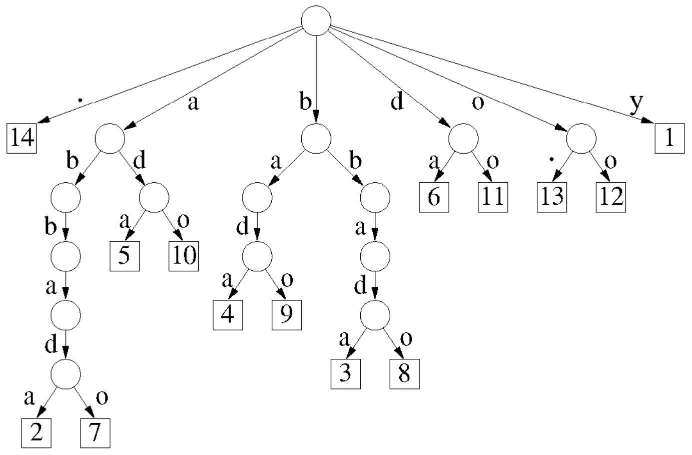

Figure 79: A suffix tree for the string "yabbadabbadoo.".

Lecture 26: Hashing

Reading: Chapter 5 in Weiss and Chapter 6 in Samet's notes.

Hashing: We have seen various data structures (e.g., binary trees, AVL trees, splay trees, skip lists) that can perform the dictionary operations `insert()`, `delete()` and `find()`. We know that these data structures provide $O(\log n)$ time access. It is unreasonable to ask any sort of tree-based structure to do better than this, since to store n keys in a binary tree requires at least $\Omega(\log n)$ height. Thus one is inclined to think that it is impossible to do better. Remarkably, there is a better method, at least if one is willing to consider expected case rather than worst case performance.

Hashing is a method that performs all the dictionary operations in $O(1)$ (i.e. constant) expected time, under some assumptions about the hashing function being used. Hashing is considered so good, that in contexts where just these operations are being performed, hashing is the method of choice (e.g. symbol tables for compilers are almost always implemented using hashing). Tree-based data structures are generally prefered in the following situations:

- When storing data on *secondary storage* (e.g. using B-trees),
- When knowledge of the relative *order* of elements is important (e.g. if a `find()` fails, I may want to know the nearest key. Hashing cannot help us with this.)

The idea behind hashing is very simple. We have a table containing m entries. We select a *hash function* $h(x)$, which is an easily computable function that maps a key x to a "virtually random" index in the range `[0..m-1]`. We will then attempt to store the key in index $h(x)$ in the table. Of course, it may be that different keys are mapped to the same location. This is called a *collision.* We need to consider how collisions are to be handled, but observe that if the hashing function does a good job of scattering the keys around the table, then the chances of a collision occuring at any index of the table are about the same. As long as the table size is at least as large as the number of keys, then we would expect that the number of keys that are map to the same cell should be small.

[1]Copyright, David M. Mount, 2001

Hashing is quite a versatile technique. One way to think about hashing is as a means of implementing a *content-addressable array.* We know that arrays can be addressed by an integer index. But it is often convenient to have a look-up table in which the elements are addressed by a key value which may be of any discrete type, strings for example or integers that are over such a large range of values that devising an array of this size would be impractical. Note that hashing is not usually used for continuous data, such as floating point values, because similar keys 3.14159 and 3.14158 may be mapped to entirely different locations.

There are two important issues that need to be addressed in the design of any hashing system. The first is how to select a hashing function and the second is how to resolve collisions.

Hash Functions: A good hashing function should have the following properties.

- It should be simple to compute (using simple arithmetic operations ideally).
- It should produce few collisions. In turn the following are good rules of thumb in the selection of a hashing function.
  - It should be a function of every bit of the key.
  - It should break up naturally occuring clusters of keys.

As an example of the last rule, observe that in writing programs it is not uncommon to use very similar variables names, `temp1`, `temp2`, and `temp3`. It is important such similar names be mapped to entirely different locations.

We will assume that our hash functions are being applied to integer keys. Of course, keys need not be integers generally. But noninteger data, such as strings, can be thought of as a sequence of integers assuming their ASCII or UNICODE encoding. Once our key has been converted into an integer, we can think of hash functions on integers. One very simple hashing function is the function

$$h(x) = x \bmod m.$$

This certainly maps each key into the range $[0..m-1]$, and it is certainly fast. Unfortunately this function is not a good choice when it comes to collision properties, since it does not break up clusters.

A more robust strategy is to first multiply the key value by some large integer constant a and then take the mod. For this to have good scattering properties either m should be chosen to be a prime number, or a should be prime relative to m (i.e. share no common divisors other than 1).

$$h(x) = (a \cdot x) \bmod m.$$

An even better approach is to both add and multiply. Let a and b be two large integer constants. Ideally a is prime relative to m.

$$h(x) = (ax + b) \bmod m.$$

By selecting a and b at random, it can be shown such a scheme produces a good performance in the sense that for any two keys the probability of them being mapped to the same address is roughly $1/m$. In our examples, we will simplify things by taking the last digit as the hash value, although in practice this is a very bad choice.

Collision Resolution: Once a hash function has been selected, the next element that has to be solved is how to handle collisions. If two elements are hashed to the same address, then we need a way to resolve the situation.

Separate Chaining: The simplest approach is a method called *separate chaining.* The idea is that we think of each of the m locations of the hash table as simple head pointers to m linked lists. The link list `table[i]` holds all keys that hash to location i.

To insert a key x we simply compute $h(x)$ and insert the new element into the linked list `table[h(x)]`. (We should first search the list to make sure it is not a duplicate.) To find a key we just search this linked list. To delete a key we delete it from this linked list. An example is shown below, where we just use the last digit as the hash function (a very bad choice normally).

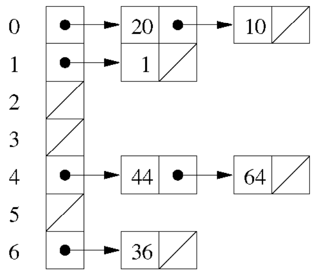

Figure 80: Collision resolution by separate Chaining.

The running time of this procedure will depend on the length of the linked list to which the key has been hashed. If n denotes the number of keys stored in the table currently, then the ratio $\lambda = n/m$ indicates the *load factor* of the hash table. If we assume that keys are being hashed roughly randomly (so that clustering does not occur), then it follows that each linked list is expected to contain λ elements. As mentioned before, we select m to be with a constant factor of n, so this ratio is a constant.

Thus, it follows from a straightforward analysis that the expected running time of a successful search is roughly

$$S_{ch} = 1 + \frac{\lambda}{2} = O(1).$$

since about half of an average list needs be searched. The running time of an unsuccessful search is roughly

$$U_{ch} = 1 + \lambda = O(1).$$

Thus as long as the load factor is a constant separate chaining provide expected $O(1)$ time for insertion and deletion.

The problem with separate chaining is that we require separate storage for pointers and the new nodes of the linked list. This also creates additional overhead for memory allocation. It would be nice to have a method that does not require the use of any pointers.

Open Addressing: To avoid the use of extra pointers we will simply store all the keys in the hash table, and use a special value (different from every key) called `EMPTY`, to determine which entries have keys and which do not. But we will need some way of finding out which entry to go to next when a collision occurs. Open addressing consists of a collection of different strategies for finding this location. In it most general form, an open addressing system involves a secondary search function, f, and if we find that location $h(x)$ is occupied, we next try locations

$$(h(x) + f(1)) \bmod m, \ (h(x) + f(2)) \bmod m, \ (h(x) + f(3)) \bmod m, \ldots.$$

until finding an open location. This sequence is called a *probe sequence*, and ideally it should be capable of searching the entire list. How is this function f chosen? There are a number of alternatives, which we consider below.

Linear Probing: The simplest idea is to simply search sequential locations until finding one that is open. Thus $f(i) = i$. Although this approach is very simple, as the table starts to get full its performance becomes very bad (much worse than chaining).

To see what is happening let's consider an example. Suppose that we insert the following 4 keys into the hash table (using the last digit rule as given earlier): 10, 50, 42, 92. Observe that the first 4 locations of the hash table are filled. Now, suppose we want to add the key 31. With chaining it would normally be the case that since no other key has been hashed to location 1, the insertion can be performed right away. But the bunching of lists implies that we have to search through 4 cells before finding an available slot.

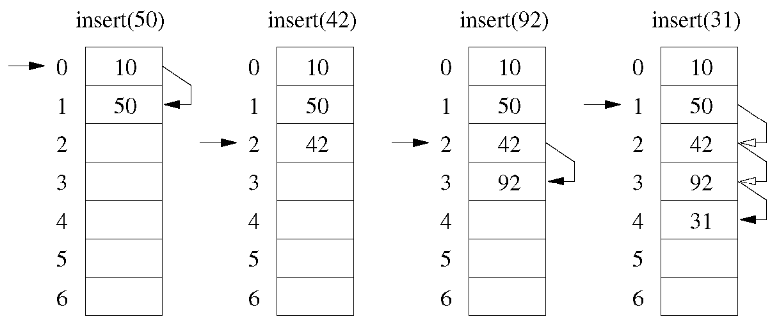

Figure 81: Linear probing.

This phenomenon is called *secondary clustering.* Primary clustering happens when the table contains many names with the same hash value (presumably implying a poor choice for the hashing function). Secondary clustering happens when keys with different hash values have nearly the same probe sequence. Note that this does not occur in chaining because the lists for separate hash locations are kept separate from each other, but in open addressing they can interfere with each other.

As the table becomes denser, this affect becomes more and more pronounced, and it becomes harder and harder to find empty spaces in the table.

Recall that $\lambda = n/m$ is the load factor for the table. For open addressing we require that $\lambda \leq 1$, because the table cannot hold more than m entries. It can be shown that the expected running times of a successful and unsuccessful searches using linear probing are

$$\begin{aligned} S_{lp} &= \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \\ U_{lp} &= \frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right). \end{aligned}$$

This is quite hard to prove. Observe that as λ approaches 1 (a full table) this grows to infinity. A rule of thumb is that as long as the table remains less than 75% full, linear probing performs fairly well.

Quadratic Probing: To avoid secondary clustering, one idea is to use a nonlinear probing function which scatters subsequent probes around more effectively. One such method is called *quadratic probing*, which works as follows. If the index hashed to $h(x)$ is full, then we consider next $h(x) + 1, h(x) + 4, h(x) + 9, \ldots$ (again taking indices mod m). Thus the probing function is $f(i) = i^2$.

Here is the search algorithm (insertion is similar). Note that there is a clever trick to compute i^2 without using multiplication. It is based on the observation that $i^2 = (i-1)^2 + 2i - 1$. Therefore, $f(i) = f(i-1) + 2i - 1$. Since we already know $f(i-1)$ we only have to add in the $2i - 1$. Since multiplication by 2 can be performed by shifting this is more efficient. The argument x is the key to find, T is the table, and m is the size of the table. We will just assume we are storing integers, but the extension to other types is straightforward.

Hashing with Quadratic Probing

```
int find(int x, int T[m]) {
    i = 0
    c = h(x)                                    // first position
    while (T[c] != EMPTY) && (T[c] != x) {      // found key or empty slot
        c += 2*(++i) - 1                        // next position
        c = c % m                               // wrap around using mod
    }
    return c
}
```

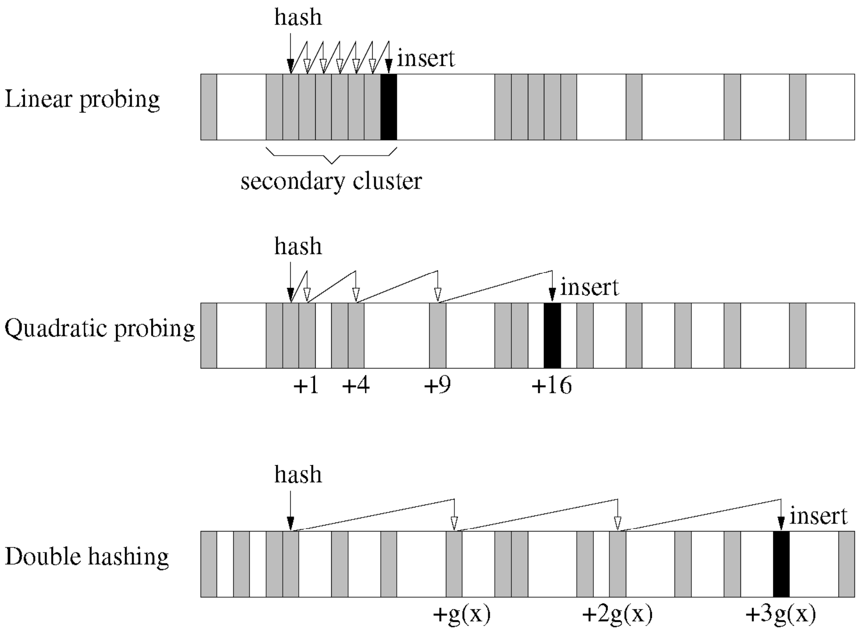

Figure 82: Various open-addressing systems.

The above procedure is not quite complete, since it loops infinitely when the table is full. This is easy to fix, by adding a variable counting the number of entries being used in the table.

Experience shows that this succeeds in breaking up the secondary clusters that arise from linear probing, but there are some tricky questions to consider. With linear probing we were assured that as long as there is one free location in the array, we will eventually find it without repeating any probe locations. How do we know if we perform quadratic probing that this will be the case? It might be that we keep hitting the same index of the table over and over (because of the fact that we take the index mod m).

It turns out (fortunately) that quadratic probing does do a pretty good job of visiting different locations of the array without repeating. It can even be formally proved that if m is prime, then the first $m/2$ locations that quadratic probing hits will be distinct. See Weiss for a proof.

Double Hashing: As we saw, the problem with linear probing is that we may see clustering or piling up arise in the table. Quadratic probing was an attractive way to avoid this by scattering the successive probe sequences around. If you really want to scatter things around for probing, then why don't you just use another hashing function to do this?

The idea is use $h(x)$ to find the first location at which to start searching, and then let $f(i) = i \cdot g(x)$ be the probing sequence where $g(x)$ is another hashing function. Some care needs to be taken in the choice of $g(x)$ (e.g. $g(x) = 0$ would be a disaster). As long as the table size m is prime and $(g(x) \bmod m \neq 0$ we are assured of visiting all cells before repeating. Note that the second hash function does not tell us *where* to put the object, it gives us an *increment* to use in cycling around the table.

Performance: Performance analysis shows that as long as primary clustering is avoided, then open addressing using is an efficient alternative to chaining. The running times of successful and unsuccessful searches for open addressing using double hashing are

$$\begin{aligned} S_{dh} &= \frac{1}{\lambda} \ln \frac{1}{1-\lambda} \\ U_{dh} &= \frac{1}{1-\lambda}. \end{aligned}$$

To give some sort of feeling for what these quantities mean, consider the following table.

λ	0.50	0.75	0.90	0.95	0.99
$U(\lambda)$	2.00	4.00	10.0	20.0	100.
$S(\lambda)$	1.39	1.89	2.56	3.15	4.65