CMSC 420: Lecture 15
Hashing

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 expect any type of comparison-based structure to do better than this in the worst case. Using binary decisions, there is a lower bound of $\Omega(\log n)$ (and more precisely, $1 + \lfloor \lg n \rfloor$) on the worst case search time.

Remarkably, there is a better method, assuming that we are willing to give up on the idea of using comparisons to locate keys. The best known method is called *hashing*. Hashing and its variants support all the dictionary operations in $O(1)$ (i.e. constant) expected time. Hashing is considered so good, that in contexts where just these operations are being performed, hashing is the *method of choice*.

Unordered vs. Ordered Dictionary: We pay a price for this speed advantage. Operations based on the totally ordering of keys *cannot* be answered efficiently. (E.g., find the kth smallest key, or find the next key larger than x, or count the number keys that lie in the interval $[a, b]$.) All of the dictionary data structures we have seen so far are examples of the *ordered dictionary ADT*. This is generally true for data structures that find keys based on comparisons. In contrast, hashing implements the more limited (unordered) *dictionary ADT*. This means we can only perform *exact* look-ups in the dictionary.

Overview: The idea behind hashing is very simple. We have a table of given size m, called the *table size*. We will assume that m is at least a small constant factor larger n. (As we shall see, when m gets close to n, the running time slows down considerably. Making m much larger than n does not improve the running time by much and wastes space.) We select a *hash function* $h(x)$, which is an easily computable function that maps a key x to a "random-like" index in the range `[0..m-1]`. We then attempt to store x (and its associated value) in index $h(x)$ in the table.

Of course, it may be that different keys are mapped to the same location. Such events are called *collisions*, and a key element in the design of a good hashing system how collisions are to be handled. If the table size is large (relative to the total number of entries) and the hashing function has been well designed, collisions should be relatively rare.

Content-Addressable Storage: 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.

For example, suppose that in a mapping application, I want to compute the travel distance between two cities. Rather than running Dijkstra's algorithm, I could just precompute and store the distances between the largest 10,000 cities in the country. While this would take a lot of space, compute times would be super fast! Note that hashing is not usually appropriate for search based on real-valued data. Very similar keys, like 3.14159 and 3.14158, may be mapped to entirely different locations by the hash function.

There are two important issues that need to be addressed in the design of any hashing system, the *hash function* and the method of *collision resolution*. Let's discuss each of these in turn.

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

- **Efficiently computable**: Ideally in constant time using simple arithmetic operations.
- **Avoids collisions**: Two different keys (even if very similar) should have a very low probability of colliding. To achieve this, the hash function should:
  - Involve *every bit* of the key (otherwise keys that differ only in these bits will collide)
  - Scatter naturally occurring *clusters* of key values.

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 very different locations in the *hash output space*. By the way, the origin of the name "hashing" is from this mixing aspect of hash functions (thinking of "hash" in food preparation as a mixture of things).

We will think of hash functions as being applied to *nonnegative integer keys*. Keys that are not integers will generally need to be converted into this form (e.g., by converting the key into a bit string, such as an ASCII or Unicode representation of a string) and then interpreting the bit string as an integer. Since the hash function's output is the range $[0..m-1]$, an obvious (but not very good) choice for a hash function is:

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

This is called *division hashing*. It satisfies our first criteria of efficiency, but consecutive keys are mapped to consecutive entries, and this is does not do a good job of breaking up clusters.

Some Common Hash Functions: Many different hash functions have been proposed. The topic is quite deep, and we will not claim to have a definitive answer for the best hash function. Here are three simple, commonly used hash functions:

Multiplicative Hash Function: Uses the hash function

$$h(x) \;=\; (ax) \bmod m,$$

where a is a *large prime number* (or at least, sharing no common factors with m).

Linear Hash Function: Enhances the multiplicative hash function with an added constant term

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

Polynomial Hash Function: We can further extend the linear hash function to a polynomial. This is often handy with keys that consist of a sequence of objects, such as strings or the coordinates of points in a multi-dimensional space.

Suppose that the key being hashed involves a sequence of numbers $x = (c_0, c_1, \ldots, c_{k-1})$. We map them to a single number by computing a polynomial function whose coefficients are these values.

$$h(x) \;=\; h(c_0, \ldots, c_{k-1}) \;=\; \left(\sum_{i=0}^{k-1} c_i p^i\right) \bmod m$$

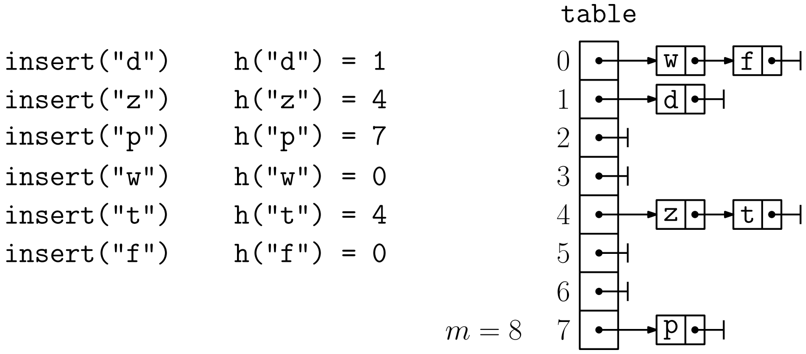
For example, if $k = 4$ and $p = 37$, and x is the sequence $x = (c_0, \ldots, c_3)$, the associated polynomial would be $c_0 + c_1 37 + c_2 37^2 + c_3 37^3$.

Collision Resolution: We have discussed how to design a hash function in order to achieve good scattering properties. But, given even the best hash function, it is possible that distinct keys can map to the same location, that is, $h(x) = h(y)$, even though $x \neq y$. Such events are called *collisions*, and a fundamental aspect in the design of a good hashing system how collisions are handled. We focus on this aspect of hashing in this lecture, called *collision resolution.*

Separate Chaining: If we have additional memory at our disposal, a simple approach to collision resolution, called *separate chaining*, is to store the colliding entries in a separate linked list, one for each table entry. More formally, each table entry stores a reference to a list data structure that contains all the dictionary entries that hash to this location.

To make this more concrete, let h be the hash function, and let `table[]` be an m-element array, such that each element `table[i]` is a linked list containing the key-value pairs (x, v), such that $h(x) = i$. We will set the value of m so that each linked list is expected to contain just a constant number of entries, so there is no need to be clever by trying to sort the elements of the list. The dictionary operations reduce to applying the associated linked-list operation on the appropriate entry of the hash table.

- `insert(x,v)`: Compute `i = h(x)`, and then invoke `table[i].insert(x,v)` to insert (x, v) into the associated linked list. If x is already in the list, signal a duplicate-key error (see Fig. 1).



```
insert("d")    h("d") = 1
insert("z")    h("z") = 4
insert("p")    h("p") = 7
insert("w")    h("w") = 0
insert("t")    h("t") = 4
insert("f")    h("f") = 0
```

Fig. 1: Collision resolution by separate chaining.

- `delete(x)`: Compute `i = h(x)`, and then invoke `table[i].delete(x)` to remove x's entry from the associated linked list. If x is not in the list, signal a missing-key error.
- `find(x)`: Compute `i = h(x)`, and then invoke `table[i].find(x)` to determine (by simple brute-force search) whether x is in the list.

Clearly, the running time of this procedure depends on the number of entries that are stored in the given table entry. To get a handle on this, consider a hash table of size m containing n keys. Define its *load factor* to be $\lambda = n/m$. If we assume that our hash function has done a good job of scattering keys uniformly about the table entries, it follows that the expected number of entries in each list is λ.

Performance: We say that a search `find(x)` is *successful* if x is in the table, and otherwise it is *unsuccessful.* Assuming that the entries appear in each linked list in random order, we would expect that we need to search roughly half the list before finding the item being sought after. It follows that the expected running time of a successful search with separate chaining is roughly $1 + \lambda/2$. (The initial "+1" accounts for the fact that we need to check one more entry than the list contains, if just to check the `null` pointer at the end of the list.) On the other hand, if the search is unsuccessful, we need to enumerate the entire list, and so the expected running time of an unsuccessful search with separate chaining is roughly $1 + \lambda$. In summary, the successful and unsuccessful search times for separate chaining are:

$$S_{\text{SC}} \;=\; 1 + \frac{\lambda}{2} \qquad U_{\text{SC}} \;=\; 1 + \lambda,$$

Observe that both are $O(1)$ under our assumption that λ is $O(1)$. Since we can insert and delete into a linked list in constant time, it follows that the expected time for all dictionary operations is $O(1 + \lambda)$.

Note the "in expectation" condition is not based on any assumptions about the insertion or deletion order. It depends simply on the assumption that the hash function uniformly scatters the keys. (There are methods, such as universal hashing, which can guarantee this with high probability.) It has been borne out through many empirical studies that hashing is indeed very efficient.

The principal drawback of separate chaining is that additional storage is required for linked-list pointers. It would be nice to avoid this additional wasted space. The remaining methods that we will discuss have this property. Before discussing them, we should discuss the issue of controlling the load factor.

Controlling the Load Factor and Rehashing: Recall that the load factor of a hashing scheme is $\lambda = n/m$, and the expected running time of hashing operations using separate chaining is $O(1 + \lambda)$. We will see below that other popular collision-resolution methods have running times that grow as $O(\lambda/(1 - \lambda))$. Clearly, we would like λ to be small and in fact strictly smaller than 1. Making λ too small is wasteful, however, since it means that our table size is significantly larger than the number of keys. This suggests that we define two constants $0 < \lambda_{\min} < \lambda_{\max} < 1$, and maintain the invariant that $\lambda_{\min} \le \lambda \le \lambda_{\max}$. This is equivalent to saying that $n \le \lambda_{\max} m$ (that is, the table is never too close to being full) and $m \le n/\lambda_{\min}$ (that is, the table size is not significantly larger than the number of entries). Define the *ideal load factor* to be the mean of these two, $\lambda_0 = (\lambda_{\min} + \lambda_{\max})/2$.

Now, as we insert new entries, if the load factor ever exceeds $\lambda_{\max}$ (that is, $n > \lambda_{\max} m$), we replace the hash table with a larger one, devise a new hash function (suited to the larger size), and then insert the elements from the old table into the new one, using the new hash function. This is called *rehashing* (see Fig. 2). More formally:

- Allocate a new hash table of size $m' = \lceil n/\lambda_0 \rceil$
- Generate a new hash function h' based on the new table size
- For each entry (x, v) in the old hash table, insert it into the new table using h'
- Remove the old table

Observe that after rehashing the new load factor is roughly $n/m' \approx \lambda_0$, thus we have restored the table to the ideal load factor. (The ceiling is a bit of an algebraic inconvenience. Throughout, we will assume that n is sufficiently large that floors and ceilings are not significant.)

Symmetrically, as we delete entries, if the load factor ever falls below $\lambda_{\min}$ (that is, $n < \lambda_{\min} m$), we replace the hash table with a smaller one of size $\lceil n/\lambda_0 \rceil$, generate a new hash function for this table, and we rehash entries into this new table. Note that in both cases

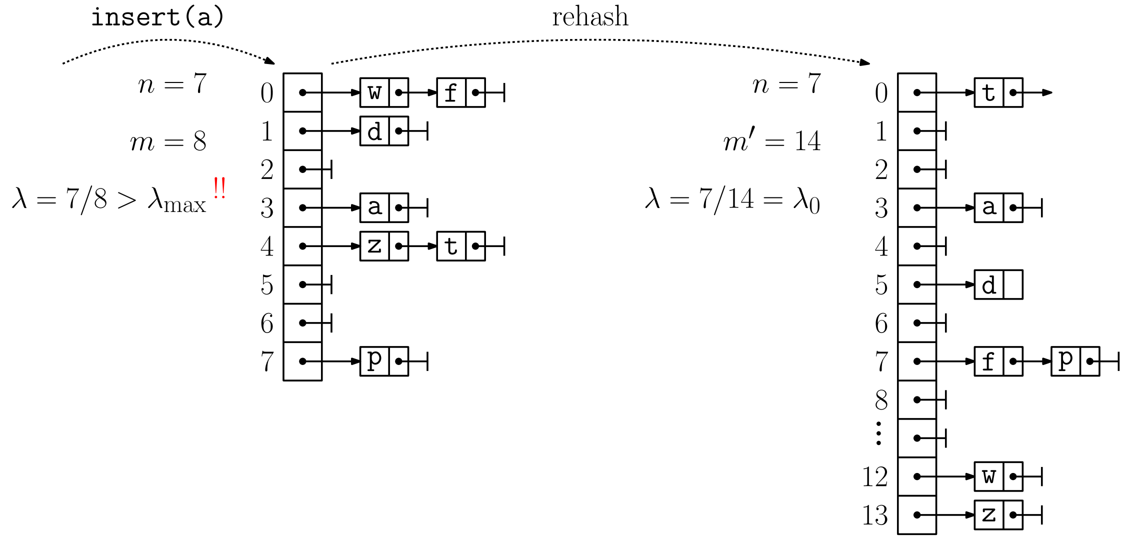


Fig. 2: Controlling the load factor by rehashing, where $\lambda_{\min} = 0.25$, $\lambda_{\max} = 0.75$, and $\lambda_0 = 0.5$. When the table size changes, we create an entirely different hash function.

(expanding and contracting) the hash table changes by a constant fraction of its current size. This is significant in the analysis.

Rehashing: The advantage of open addressing is that we do not have to worry about pointers and storage allocation. However, if the table becomes full, or just too close to full so that performance starts degrading (e.g. the load factor exceeds some threshold in the range from 80% to 90%). The simplest scheme, is to allocate a new array of size a constant factor larger (e.g. twice) the previous array. Then create a new hash function for this array. Finally go through the old array, and hash all the old keys into the new table, and then delete the old array.

You may think that in the worst case this could lead to *lots* of recopying of elements, but notice that if the last time you rehashed you had 1000 items in the array, the next time you may have 2000 elements, which means that you performed at least 1000 insertions in the mean time. Thus the time to copy the 2000 elements is *amortized* by the previous 1000 insertions leading up to this situation.

Amortized Analysis of Rehashing: (Optional.) Here we show that if a hash table is maintained using rehashing when it gets too full (as described earlier), the amortized cost per operation is a constant. The proof is quite similar to the proof given earlier in the semester for the dynamically expanding stack.

Observe that whenever we rehash, the running time is proportional to the number of keys n. If n is large, rehashing clearly takes a lot of time. But once we have rehashed, we will need to do a significant number of insertions or deletions before we need to rehash again.

To make this concrete, let's consider a specific example. Suppose that $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, and hence $\lambda_0 = 1/2$. Also suppose that the current table size is $m = 1000$. Suppose the most recent insertion caused the load factor to exceed our upper bound, that is $n > \lambda_{\max} m = 750$. We allocate a new table of size $m' = n/\lambda_0 = 2n = 1500$, and rehash all the old elements into this new table. In order to overflow this new table, we will need for n

to increase to some higher value n' such that $n'/m' > \lambda_{\max}$, that is $n' > (3/4)1500 = 1125$. In order to grow from the current 750 keys to 1125 keys, we needed to have at least 375 more insertions (and perhaps many more operations if finds and deletions were included as well). This means that we can *amortize* the (expensive) cost of rehashing 1125 keys against the 375 (cheap) insertions. (If you do the calculations, with 4 tokens per insertion per the example, you get $(1 + 3)375 = 375 + 1125$ so you've accumulated the desired 1125 tokens to pay for the cost of rehashing.)

Recall that the *amortized cost* of a series of operations is the total cost divided by the number of operations.

Theorem: Assuming that individual hashing operations take $O(1)$ time each, if we start with an empty hash table, the amortized complexity of hashing using the above rehashing method with minimum and maximum load factors of $\lambda_{\min}$ and $\lambda_{\max}$, respectively, is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$.

Proof: Our proof is based on the same *token-based argument* that we used in the earlier lecture. Let us assume that each standard hashing operation takes exactly 1 unit of time, and rehashing takes time n, where n is the number of entries currently in the table. Whenever we perform a hashing operation, we assess 1 unit to the actual operation, and save $2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min})$ *tokens* to pay for future rehashings. (Where did this "magic" number of tokens come from? The answer is to work through the analysis below treating the number of tokens as an unknown quantity x. Then figure out what value x needs to be to make the inequalities work out.)

There are two ways to trigger rehashing: expansion due to insertion, and contraction due to deletion. Let us consider insertion first. Suppose that our most recent insertion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\max} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that n is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we inserted at least $n - n' = (\lambda_{\max} - \lambda_0)m$ entries. Therefore, the number of tokens we have accumulated since then is at least

$$\begin{aligned}(\lambda_{\max} - \lambda_0)m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\lambda_{\max} - \frac{\lambda_{\max} + \lambda_{\min}}{2}\right)m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\ &= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right)m\frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\ &= \lambda_{\max} m \;\approx\; n,\end{aligned}$$

which implies that we have accumulated enough tokens to pay the cost of n to rehash.

Next, suppose that our most recent deletion has triggered rehashing. This implies that the current table contains roughly $n \approx \lambda_{\min} m$ entries. (Again, to avoid worrying about floors and ceilings, let's assume that n is quite large.) The last time the table was rehashed, the table contained $n' = \lambda_0 m$ entries immediately after the rehashing finished. This implies that we deleted at least $n' - n = (\lambda_0 - \lambda_{\min})m$ entries. Therefore, the number of tokens we have accumulated since then is at least

$$\begin{aligned}(\lambda_0 - \lambda_{\min}) m \frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} &= \left(\frac{\lambda_{\max} + \lambda_{\min}}{2} - \lambda_{\min}\right) m \frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\ &= \left(\frac{\lambda_{\max} - \lambda_{\min}}{2}\right) m \frac{2\lambda_{\max}}{\lambda_{\max} - \lambda_{\min}} \\ &= \lambda_{\max} m \;\geq\; \lambda_{\min} m \;\approx\; n,\end{aligned}$$

again implying that we have accumulated enough tokens to pay the cost of n to rehash.

To make this a bit more concrete, suppose that we set $\lambda_{\min} = 1/4$ and $\lambda_{\max} = 3/4$, so that $\lambda_0 = 1/2$ (see Fig. 2). Then the amortized cost of each hashing operation is at most $1 + 2\lambda_{\max}/(\lambda_{\max} - \lambda_{\min}) = 1 + 2(3/4)/(1/2) = 4$. Thus, we pay just additional factor of four due to rehashing. Of course, this is a worst case bound. When the number of insertions and deletions is relatively well balanced, we do not need rehash very often, and the amortized cost is even smaller.

Open Addressing: Let us return to the question of collision-resolution methods that do not require additional storage. Our objective is to store all the keys within the hash table. (Therefore, we will need to assume that the load factor is never greater than 1.) To know which table entries store a value and which do not, we will store a special value, called `empty`, in the empty table entries. The value of `empty` must be such that it matches no valid key.

Whenever we attempt to insert a new entry and find that its position is already occupied, we will begin probing other table entries until we discover an empty location where we can place the new key. In it most general form, an open addressing system involves a secondary search function, f. If we discover 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. (To make this a bit more elegant, let us assume that $f(0) = 0$, so even the first probe fits within the general pattern.) This 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. In other words, the probe function is $f(i) = i$. Although this approach is very simple, it only works well for fairly small load factor. As the table starts to get full, and the load factor approaches 1, the performance of linear probing becomes very bad.

To see what is happening consider the example shown in Fig 3. Suppose that we insert four keys, two that hash to `table[0]` and two that hash to `table[2]`. Because of the collisions, we will fill the table entries `table[1]` and `table[3]` as well. Now, suppose that the fifth key ("`t`") hashes to location `table[1]`. This is the first key to arrive at this entry, and so it is not involved any collisions. However, because of the previous collisions, it needs to slide down three positions to be entered into `table[4]`.

This phenomenon is called *primary clustering*. Primary clustering occurs when the collision resolution algorithm causes keys that hash to nearby locations to form into clumps. Linear probing is especially susceptible to primary clustering. As the load factor approaches 1, primary clustering becomes more and more pronounced, and probe sequences may become unacceptably long.

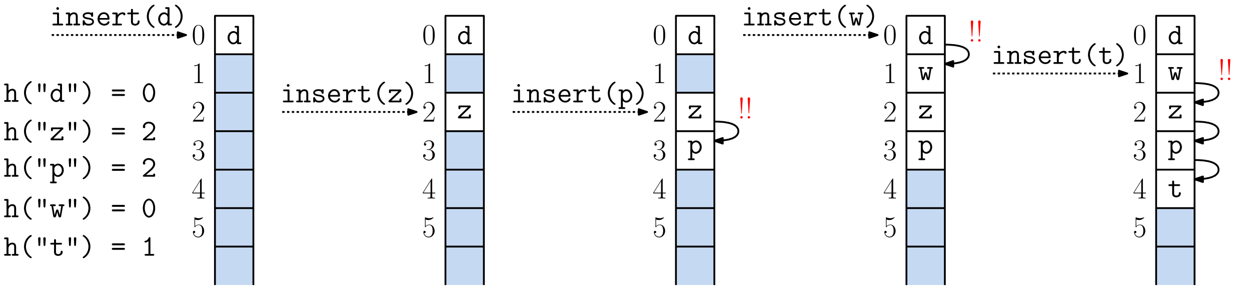


Fig. 3: Linear probing.

While we will not present it, a careful analysis shows that the expected costs for successful and unsuccessful searches using linear probing are, respectively:

$$S_{\text{LP}} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \qquad U_{\text{LP}} = \frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right).$$

The proof is quite sophisticated, and we will skip it. Observe, however, that in the limit as $\lambda \to 1$ (a full table) the running times (especially for unsuccessful searches) rapidly grows to infinity. A rule of thumb is that as long as the table remains less than 75% full, linear probing performs fairly well. Nonetheless, the issue of primary clustering is a major shortcoming, and the methods given below do significantly better in this regard.

Quadratic Probing: To avoid primary 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 $f(i) = i^2$. **Alert:** Note that the quadratic terms are added to the *initial* hash location, not to the previously probed location.

The `find` function is shown in the following code block. Rather than computing $h(x)+i^2$, we use a cute trick to update the probe location. Observe that $i^2 = (i-1)^2 + 2i - 1$. Thus, we can advance to the next position in the probe sequence (i^2) by incrementing the old position ($(i-1)^2$) by the value $2i - 1$. We assume that each table entry `table[i]` contains two elements, `table[i].key` and `table[i].value`. If found, the function returns the associated value, and otherwise it returns `null`.

Find Operation with Quadratic Probing

```
Value find(Key x) {                         // find x
    int c = h(x)                            // initial probe location
    int i = 0                               // probe offset
    while (table[c].key != empty) && (table[c].key != x) {
        c += 2*(++i) - 1                    // next position
        c = c % m                           // wrap around if needed
    }
    return table[c].value                   // return associated value (or null if empty)
}
```

Experience shows that this succeeds in breaking up the clusters that arise from linear probing, but it is far from perfect. Quadractic probing is also clumping, called *secondary clustering.*

This refers to clumping that occurs far away from the initial hash location. The problem with quadratic probing is that all keys use the same probe sequence. So, keys that hash to nearby locations have probe sequences that generate nearby locations as well.

In addition, quadratic probing suffers from a rather knotty problem. Unlike linear probing, which is guaranteed to try every entry in your table, quadratic probing bounces around less predictably. Might it miss some entries? The answer, unfortunately, is yes! To see why, consider the rather trivial case where $m = 4$. Suppose that $h(x) = 0$ and your table has empty slots at `table[1]` and `table[3]`. The quadratic probe sequence will inspect the following indices:

$$1^2 \bmod 4 = 1 \qquad 2^2 \bmod 4 = 0 \qquad 3^2 \bmod 4 = 1 \qquad 4^2 \bmod 4 = 0 \ldots$$

It can be shown that it will only check table entries 0 and 1. This means that you cannot find a slot to insert this key, even though your table is only half full!

The following lemma shows that, if you choose your table size m to be a prime number, then quadratic probing is guaranteed to visit at least half of the table entries before repeating. This means that it will succeed in finding an empty slot, provided that m is prime and your load factor is smaller than 1/2.

Theorem: If quadratic probing is used, and the table size m is a prime number, the first $\lfloor m/2 \rfloor$ probe sequences are distinct.

Proof: Suppose by way of contradiction that for $0 \le i < j \le \lfloor m/2 \rfloor$, both $h(x) + i^2$ and $h(x) + j^2$ are equivalent modulo m. Then the following equivalencies hold modulo m:

$$i^2 \equiv j^2 \iff i^2 - j^2 \equiv 0 \iff (i-j)(i+j) \equiv 0 \pmod{m}$$

This means that the quantity $(i-j)(i+j)$ is a multiple of m. But this cannot be, since m is prime and both $i-j$ and $i+j$ are nonzero and strictly smaller than m. (The fact that $i < j \le \lfloor m/2 \rfloor$ implies that their sum is strictly smaller than m.) Thus, we have the desired contradiction.

This is a rather weak result, however, since people usually want their hash tables to be more than half full. You can do better by being more careful in the choice of the table size and/or the quadratic increment. Here are two examples, which I will present without proof.

- If the table size m is a prime number of the form $4k + 3$, then *alternating quadratic* probe sequence $(-1)^i i^2$ (that is, $h(x) + \langle 0, -1, +4, -9, +16, \ldots \rangle$) succeeds in probing all entries.
- If the table size m is a power of two, and the increment is chosen to be $\frac{1}{2}(i^2 + i)$ (that is, $h(x) + \langle 0, +1, +3, +6, +10, \ldots \rangle$) succeeds in probing all entries.

The question of performance of quadratic probing is quite mathematically deep, and is closely related to the topic of *quadratic residues* from number theory.

Double Hashing: Both linear probing and quadratic probing have shortcomings. Our final method overcomes both of these limitations. Recall that in any open-addressing scheme, we are accessing the probe sequence $h(x) + f(1)$, $h(x) + f(2)$, and so on. How about if we make the increment function $f(i)$ a function of the search key? Indeed, to make it as random as possible, let's use another hash function! This leads to the concept of *double hashing*.

More formally, we define two hash functions $h(x)$ and $g(x)$. We use $h(x)$ to determine the first probe location. If this entry is occupied, we then try:

$$h(x) + g(x), \quad h(x) + 2g(x), \quad h(x) + 3g(x), \quad \ldots$$

More formally, the probe sequence is defined by the function $f(i) = i \cdot g(x)$. In order to be sure that we do not cycle, it should be the case that m and $g(x)$ are *relatively prime*, that is, they share no common factors. There are lots of ways to achieve this. For example, select $g(x)$ to be a prime that is strictly larger than m or the product of primes that are larger than m. Another approach would be to set m to be a power of 2, and then to generate $g(x)$ as the product of prime numbers other than 2. In short, we should be careful in the design of a double-hashing scheme, but there is a lot of room for adjustment.

Fig. 4 provides an illustration of how the various open-addressing probing methods work.

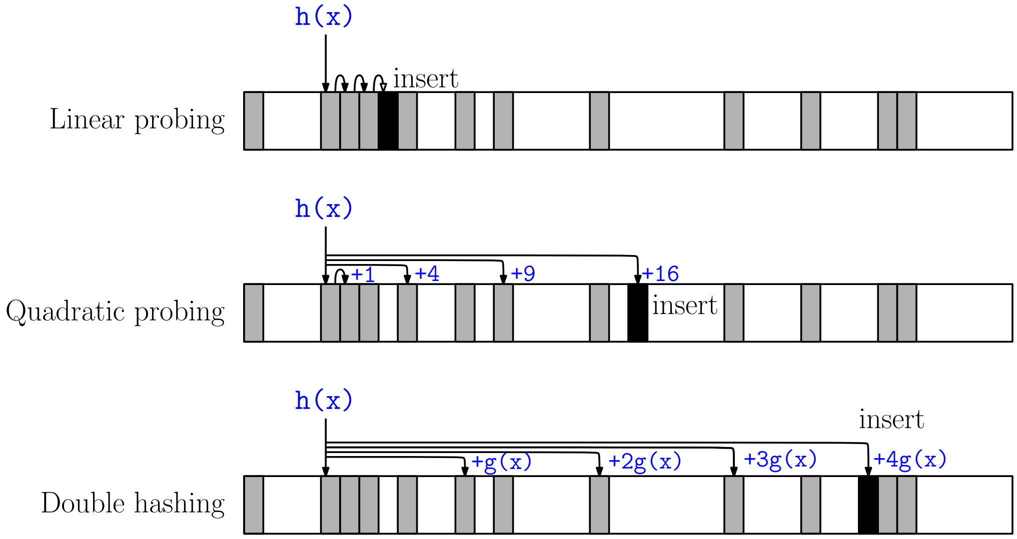


Fig. 4: Various open-addressing systems. (Shaded squares are occupied and the black square indicates where the key is inserted.)

Theoretical running-time analysis shows that double hashing is the most efficient among the open-addressing methods of hashing, and it is competitive with separate chaining. The running times of successful and unsuccessful searches for open addressing using double hashing are

$$S_{\text{DH}} = \frac{1}{\lambda} \ln \frac{1}{1-\lambda} \qquad U_{\text{DH}} = \frac{1}{1-\lambda}.$$

To get some 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

Note that, unlike tree-based search structures where the search time grows with n, these search times depend only on the load factor. For example, if you were storing 100,000 items in your data structure, the above search times (except for the very highest load factors) are superior to a running time of $O(\log n)$.

Deletions: Deletions are a bit tricky with open-addressing schemes. Can you see why?

The issue is illustrated Fig. 5. When we insert "a", an existing key "f" was on the probe path, and we inserted "a" beyond "f". Then we delete "f" and then search for "a". The problem is that with "f" no longer on the probe path, we arrive at the empty slot and take this to mean that "a" is not in the dictionary, which is not correct.

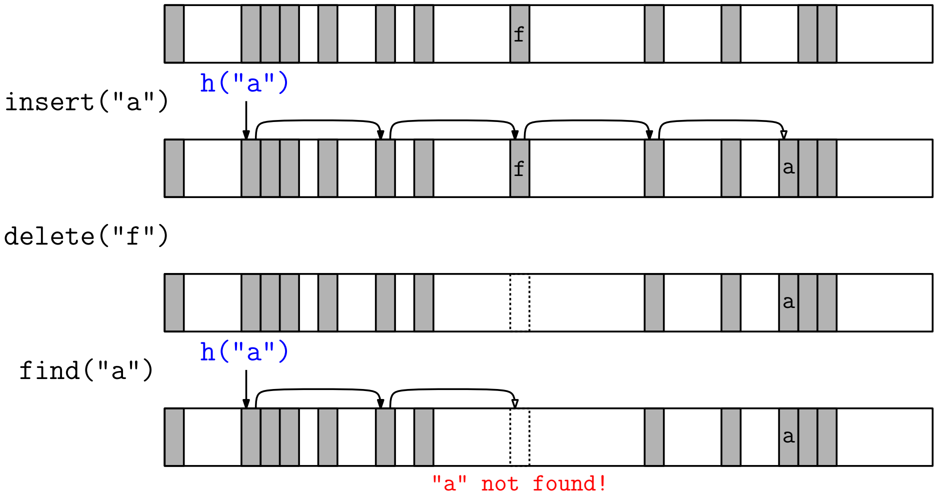


Fig. 5: The problem with deletion in open addressing systems.

To handle this we create a new special value (in addition to empty) for cells whose keys have been deleted, called, say "deleted". If the entry is marked deleted this means that the slot is available for future insertions, but if the find function comes across such an entry, it should keep searching. The searching stops when it either finds the key or arrives at an cell marked "empty" (key not found).

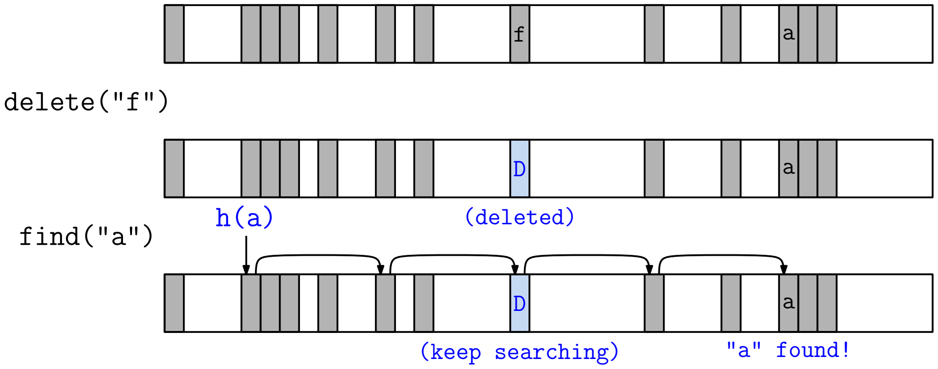


Fig. 6: Deleting in open-addressing by using special *empty* entry.

Using the "deleted" entry is a rather quick-and-dirty fix. It suffers from the shortcoming that as keys are deleted, the search paths are unnaturally long. (The load factor has come down, but the search paths are just as long as before.) A more clever solution would involve moving keys that that were pushed down in the probe sequence up to fill the vacated entries. Doing this, however make deletion times longer.

Further refinements: Hashing is a very well studied topic. We have hit the major points, but there are a number of interesting refinements that can be applied. One example is a technique called *Brent's method*. This approach is used to reduce the search times when double hashing is used. It exploits the fact that any given cell of the table may lie at the intersection of two or more probe sequences. If one of these probe sequences is significantly longer than the other, we can reduce the average search time by changing which key is placed at this point of overlap. Brent's algorithm optimizes this selection of which keys occupy these locations in the hash table.