A dictionary is a data structure that stores (key, value) pairs and supports the operations Insert, Search, and Delete. So far we have seen a couple ways to implement dictionaries. We can use a balanced binary search tree, which supports all these operations in $O(\log n)$ time. We can also use a doubly linked list, which supports Insert and Delete in $O(1)$ time but Search in $O(n)$ time. Now we will learn how to use a hash table to support all of these operations in less time.

0.1 Direct-address tables

Inspired by counting sort, we can come up with a way to do this that is kind of naive, and relies on strong assumptions about the input.

Suppose each element in the data structure has a key drawn from $\{0, 1, \ldots, m-1\}$, where m is not too large, and no two elements have the same key.

Then we can simply build an array `T[0..m-1]`, where the element with key k is placed at `T[k]`, and `T[i] = NIL` if there is no element in the data structure with key i. (Depending on the application, we may either place the value itself at `T(k)`, or just a pointer to the value.) This structure is called a **direct-address table**, and Search, Insert, and Delete all run in $O(1)$, since they simply correspond to accessing the elements in the array.

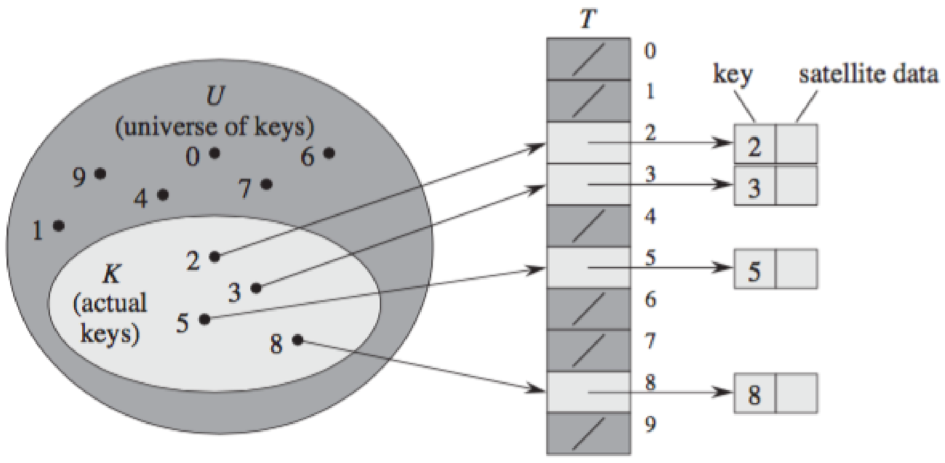


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

1 Hashing with chaining

Direct-address tables are impractical when the number of possible keys is large, or when it far exceeds the number of keys that are actually stored. Instead, we use hash tables. With hash tables, instead of storing the element with key k in slot k, we store it in slot $h(k)$. Here $h : U \to \{0, 1, \dots, m-1\}$ is a **hash function** that maps all possible keys to the slots of an array `T[0..m - 1]`.

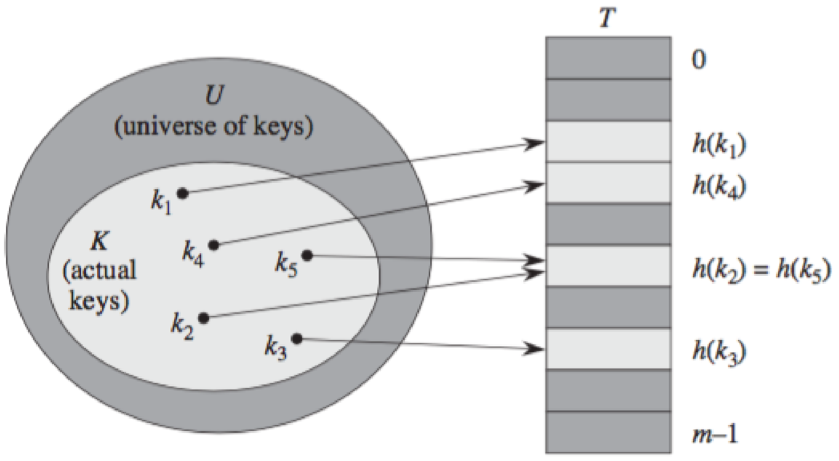


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

h is not a one-to-one function (or hashing would add no value over direct addressing), so elements with different keys may be hashed to the same slot in the array (producing a **hash collision**). One way to resolve this is by having a linked list in each nonempty array slot that contains all of the elements whose keys map to that slot. (More precisely, the array slot contains either a pointer to the head of the list, or `NIL` if there are no elements currently in the table whose keys map to that slot.)

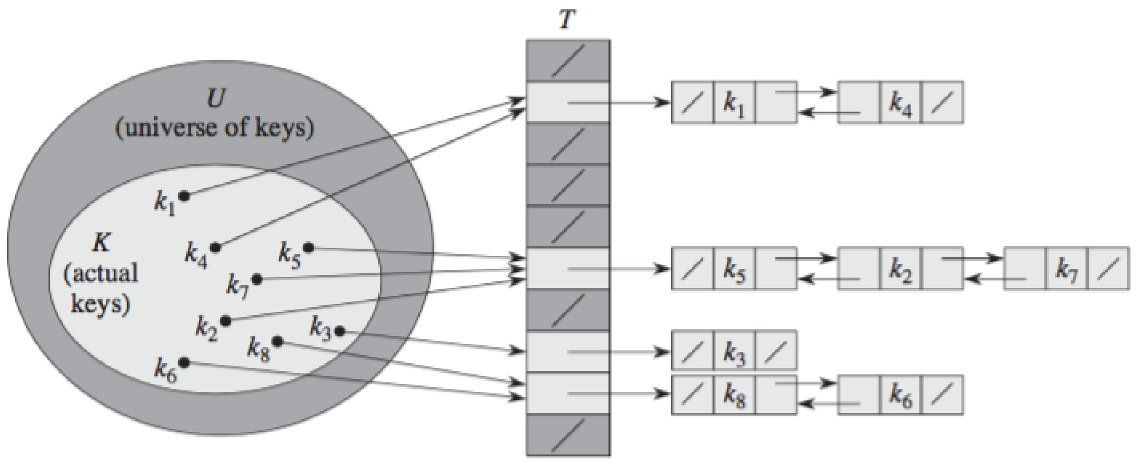


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

The Insert, Search, and Delete operations reduce to Insert, Search, and Delete on a linked list:

In the worst-case scenario, all of the keys hash to the same array slot, so Search takes $O(n)$, while Insert and Delete still take $O(1)$. This is in addition to the time taken to compute the hash function (which we usually assume is $O(1)$). However, we will primarily be concerned with the average case behavior. It turns out that in the average case, Search takes $\Theta(1 + \alpha)$ time, where $\alpha = n/m$ is the **load factor** (n being the number of elements, and m being the number of slots).

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

1.1 Average case runtime analysis

The average case runtime depends on how well the hash function h distributes the set of keys to be stored among the m slots. There is an art to choosing good hash functions, which we will describe later. For our runtime analysis, we will assume **simple uniform hashing**, which means any given element is equally likely to hash into any of the m slots.

1.1.1 Time for an unsuccessful search

Theorem 11.1: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof: Let n_j be the number of elements in the list $T[j]$. Then $n = n_0 + n_1 + \cdots + n_{m-1}$, so $E[n] = E[n_0] + \cdots + E[n_{m-1}]$, and since all the lists have the same expected value, $E[n_j] = n/m = \alpha$.

Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length α. Thus the expected number of elements examined in an unsuccessful search is α, and since we need $O(1)$ time to compute the hash function, we get a runtime of $\Theta(1 + \alpha)$.

1.1.2 Time for a successful search

Theorem 11.2: In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof: We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is one more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x were all inserted *after* x was inserted. To find the expected number of elements examined, we take the average, over the n elements in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

Let x_i denote the ith element added to the table, and let

$$X_{ij} = \begin{cases} 1 & \text{if the keys of } x_i \text{ and } x_j \text{ hash to the same table slot} \\ 0 & \text{otherwise} \end{cases}$$

Then if we want to count the number of keys that hash to the same list as element i after i is added to the list, that is just

$$\sum_{j=i+1}^{n} X_{ij}$$

We are starting the sum at $i+1$ because we only want to count the elements that are inserted after i.

Because of simple uniform hashing, the probability that two keys hash to the same table slot is $1/m$, so $E[X_{ij}] = 1/m$. So the expected number of elements examined in a successful search is

$$\begin{aligned}
& E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
= & \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \\
= & \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
= & \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) \\
= & 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
= & 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
= & 1+\frac{n-1}{2m} \\
= & 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}$$

which is $\Theta(1+\alpha)$.

2 Hash functions

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. But we can frequently get good results by attempting to derive the hash value in a way that we expect to be independent of any patterns that might exist in the data.

Here we assume that the keys are natural numbers. If they're not, we can often interpret them as natural numbers. For example, if we map any ASCII character to a number between 0 and 255, and specifically if `h` gets mapped to 104 and `i` gets mapped to 105, we can map the string `hi` to $104 \cdot 256 + 105$.

These methods are heuristics, and as far as I know, they are not guaranteed to produce good performance.

2.0.1 Division method

Let $h(k) = k \bmod m$ for some value of m.

For instance, if $m = 12$ and $k = 100$, then $h(k) = 4$.

Some values of m are better than others. For instance, you don't want to choose a power of 2, because then $h(k)$ is just the lowest-order bits of k, and generally not all low-order bit patterns are equally likely. Often you want to choose a prime number that is not too close to an exact power of 2.

2.0.2 Multiplication method

1. Multiply the key k by some number $0 < A < 1$.
2. Extract the fractional part of kA.
3. Multiply it by m.
4. Take the floor of the results.

In other words, let $h(k) = \lfloor m(kA \bmod 1) \rfloor$.

Here the value of m is not critical, but some values of A work better than others. Knuth suggests that $A = (\sqrt{5} - 1)/2$ is likely to work reasonably well.

3 Open addressing

Another implementation of hash tables uses open addressing. In open addressing, we don't have linked lists, and every entry of the hash table contains either a single element or `NIL`.

To insert an element, we first try to insert it into one slot, and if there is no room in that slot, we try to insert it into another slot, and we keep going until we discover all the slots are full. The order in which we examine the slots depends on which key is inserted (if it didn't, Search would take $\Theta(n)$, which would be bad). To search for an element, we examine the slots in the same order until we either find the element or reach NIL.

To account for the order in which the slots are examined, we extend the hash function to take an extra parameter (namely, where we are in the ordering). So now we have $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$. We require that for every key k, the **probe sequence** $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ be a permutation of $0, 1, \ldots, m-1$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Deletion from an open-address table is difficult. You can't just replace the key with `NIL`, because it will cause the Search function to terminate early. So we can replace the key with `DELETED` instead, and modify the Insert function so that it treats `DELETED` slots as empty. Unfortunately, this is still not a good solution, because if we delete lots of elements from the table, search times no longer depend on the load factor of the table (they may be much larger than would be predicted by the number of elements in the table). So usually if we need to delete keys we would use chaining instead of open addressing.

3.1 Ways to create hash functions

These two-dimensional hash functions rely on the existence of a one-dimensional "auxiliary hash function" $h' : U \to \{0, 1, \ldots, m-1\}$.

Linear probing: Let $h(k, i) = (h'(k) + i) \bmod m$ for $i = 0, 1, \ldots, m-1$. In other words, first we examine the slot $T[h'(k)]$, i.e. the slot given by the auxiliary hash function. Then we probe the slot next to it, and then the slot next to that slot, etc, wrapping around to the beginning of the array if necessary.

This is easy to implement, but can be problematic because long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. On the other hand, it is a fairly cache friendly algorithm, since you are accessing elements that are very close to each other.

Quadratic probing: Let $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$. This is a better method than linear probing, but we may have to select the constants carefully. Also, if two keys have the same initial probe position, then their probe sequences are the same, which means we still get clustering, but to a lesser extent.

Double hashing: Here we use two hash functions, h_1 and h_2, and let $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. This is a great method and the permutations produced have many of the characteristics of randomly chosen permutations.

Note that $h_2(k)$ must be relatively prime to the hash table size m for the entire hash table to be searched. One way to do this is to let m be a power of 2 and design h_2 so that it always produces an odd number. Another way is to let m be prime and design h_2 so it always returns a positive integer less than m.

For example, we can let $h_1(k) = k \bmod m$ and $h_2(k) = 1 + (k \bmod m')$, where m' is chosen to be slightly less than m.

3.2 Runtime analysis

In this analysis, we assume no keys are deleted. We also assume uniform hashing, which means the probe sequence of each key is equally likely to be any of the $m!$ permutations of $0, 1, \dots, m-1$.

3.2.1 Unsuccessful search

Theorem 11.6: Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

(Note that in our scheme the load factor has to be less than or equal to 1 because there is at most one element in each slot.)

This means if the hash table is half full, we have to make 2 probes on average for each unsuccessful search, and if the hash table is 90% full, we have to make 10 probes on average.

Let X be the number of probes made in an unsuccessful search, and let A_i be the event that an ith probe occurs and it is to an occupied slot. Then $X \geq i$ when the event $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$ occurs.

Thus

$$\begin{aligned} P[X \geq i] &= P[A_1 \cap A_2 \cap \cdots \cap A_{i-1}] \\ &= P[A_1] \cdot P[A_2 | A_1] \cdot P[A_3 | A_1 \cap A_2] \dots P[A_{i-1} | A_1 \cap \cdots \cap A_{i-2}] \end{aligned}$$

$P[A_1] = n/m$ because there are n elements and m slots. For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$ because we are finding one of the remaining $(n-(j-1))$ elements in one of the $(m-(j-1))$ unexamined slots, and we have uniform hashing so each of the slots is equally likely to be chosen next.

Because $n < m$, we have

$$\frac{n-j}{m-j} \leq \frac{n}{m}$$

for all $0 \leq j < m$. Thus

$$P[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \dots \cdot \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

Now

$$E[X] = \sum_{i=1}^{\infty} P[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

by the sum of an infinite geometric series.

The reason why $E[X] = \sum_{i=1}^{\infty} P[X \geq i]$ is because for all i, $P[X \geq i] = P[X = i] + P[X = i+1] + \dots$. So if we sum up all the terms, the term $P[X = 1]$ appears once, the term $P[X = 2]$ appears twice, the term $P[X = 3]$ appears three times, etc. This makes the sum equivalent to $\sum_{i=1}^{\infty} i \cdot P[X = i]$.

Note that we can interpret the bound of $1 + \alpha + \alpha^2 + \dots$ as follows. We always make the first probe. With probability approximately α, the first probe finds an occupied slot, and we need to probe a second time. With probability approximately α^2, the first two slots are occupied so we make a third probe, etc.

3.2.2 Insertion

Corollary 11.7: Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.

Proof: An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1-\alpha)$.

3.2.3 Successful search

Theorem 11.8: Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

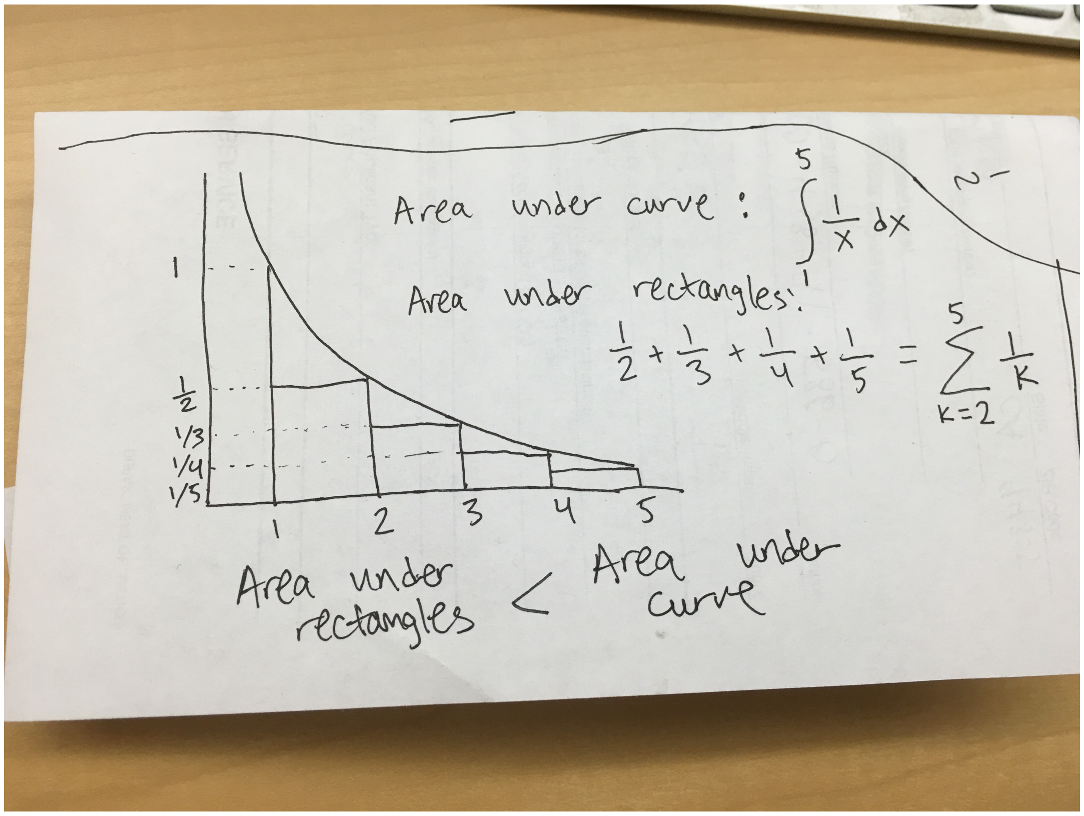
This means that if the hash table is half full, we have to make < 1.387 probes on an average successful search, and if the hash table is 90% full, we have to make < 2.559 probes on an average successful search.

Proof: The key insight is that a search for a key k reproduces the same probe sequence as when the element with key k was inserted. So we just need to count how many probes were required to insert the element into the table, back when there were fewer elements in the table.

By Corollary 11.7, if k was the $(i+1)$st key inserted into the hash table, there were only i keys in the table at the time, so the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table, we get

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \qquad \text{(reversing the sum, letting } k = m-i\text{)} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)dx \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned}$$

Here is an explanation of why the sum is bounded by an integral:


Area under curve : $\int_1^5 \frac{1}{x}dx$
Area under rectangles:
$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5} = \sum_{k=2}^{5}\frac{1}{k}$
1
1/2
1/3
1/4
1/5
1
2
3
4
5
Area under rectangles < Area under curve