Chapter 11
Hashing Tables

Many applications require a dynamic set that supports such dictionary operations as Insert, Search, and Delete. Membership list is a good example.

Another one could be the symbolic table as managed by a compiler, in which the keys of the elements are arbitrary character strings that corresponds to identifiers used in a program.

When a new id is created, all the associated information such as name (as the key), type, initial value, etc., are inserted into this table.

Later on, whenever this name is seen in its defining scope, a search has to be carried out in the table for its type and value.

Finally, for dynamically created ids, when it is no longer needed, such as those in a linked list, it should be deleted from the table.
Direct address tables

Assume that an application needs a dynamic set in which each element has a key drawn from \( U = \{0, 1, \cdots, m - 1\} \), where \( m \) is not too large, and all the keys are distinct.

We can use a direct-address table, \( T[0..m-1] \), where we directly use a key as its position in \( T \). If position \( k \) does not contain an element yet, we set \( T[k] \) to NULL.

Below shows such a table when \( m = 10 \), which is not too large, and everything might be used.
The related operations

The operations for such a direct address table are then easy to implement, and all taking $\Theta(1)$. For example, let $x$ be an element,

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

DIRECT-ADDRESS-INSERT(T, x)
$T[\text{key}[x]] <- x$

DIRECT-ADDRESS-DELETE(T, x)
$T[\text{key}[x]] <- $NIL

**Homework:** Exercises 11.1-1(*), and 11.1-2.
Hashing tables

Recall that the above simple implementation of the direct-addressing table is based on the assumption that “\( m \) is not too large.”

It is not practical to use a direct-addressing table when \( |U| \) is “too large”. More specifically, when the number of keys actually stored is much smaller than that of the potential ones, it leads to poor space efficiency. 😞

Alternatively, a **Hashing table** uses an array whose size is *proportional* to the number of keys actually stored, but not the total number of such keys, thus saving space.

We use an efficient *hashing function*, \( f \), that maps every element to a location, \( f(k) \), according to its key value \( k \). Notice that, with the direct address table implementation, \( f(k) = k \).
Two issues

1. When using an efficient, i.e., $\Theta(1)$, hashing function, Hashing table provides expected time of $\Theta(1)$ when searching for an item, inserting an item into, and deleting an item from a list. It may reach a worst time of $\Theta(n)$.

In comparison, if we use either a sorted, or an unsorted array following a bear/corn approach, it would take $\Theta(n)$ on average.

*Even distribution* is an important issue: nothing is too short or too long; otherwise, the promised $\Theta(1)$ times can be delivered.

2. Because the number of potential records is usually much larger then the size of such a table, *collision* is unavoidable, i.e., for some $k_1, k_2, f(k_1) = f(k_2)$. In other words, multiple things are mapped to the same place. ☹️
Birthday related questions

**Question:** How many people should we gather so that it is guaranteed that at least two of them will have the same birthday?

**Answer:** 366, if we assume there are 365 days a year i.e., not a leap year.

**Question:** How many people should we gather so that it will be *more than likely* that at least two of them will have the same birthday?

**Answer:** The probability that none of the \( k \) \((\in [2, 366])\) people shares the same birthday is

\[
Prob(365, k) = \frac{365}{365} \times \frac{364}{365} \times \cdots \times \frac{365 - (k - 1)}{365}.
\]

Notice that, for all \( k \geq 366 \), \( Prob(365, k) = 0 \): When there are 366 people or more, it is guaranteed that at least two people share the same birthday by the “pigeonhole principle”.


Now what?

Since either at least two people share the same birthday, or none of them does, the probability of at least two of them sharing the same birthday is simply $P_S(365, k) = 1 - Prob(365, k)$.

It turns out that

$$k_{365} = \min_k \{k | P_S(365, k) \geq 0.5\} = 23,$$

which is about 6.3% of the total, where the associated probability is 0.507297.

**Question:** What about the case when there are $r$, instead of 365, days in a year?

**Answer:**

$$k_r = \frac{1}{2} + \sqrt{\frac{1}{4} + 2r \log_e 2}.$$

When taking $r = 365$, we have 22.7, consistent with the previous one.

When taking $r$ to be a million, we have that $k_r = 1,178$, only 0.11% of the total.
A few related ones....

**Question:** What is the average (expected) number of people we should gather so that at least two of them share the same birthday, i.e., some birthday will be shared by at least two people?

**Answer:** It turns out that its expected value is 24. Check out the following page for details.

**Question:** What is the average (expected) number of people we should gather so that all the 365 birthdays will be taken by at least one person?

**Answer:** It is actually a pretty large number: 2,364.

We will discuss a few effective techniques addressing this collision issue, since we have to deal with it.
The general case

**Question:** What is the average (expected) number of people we should gather so that at least two of them share the same birthday, if there are \( r \) days in a year?

**Answer:** Let \( B \) be a random variable whose value is the number of the people we get when the first sharing occurs, \( B \in [2, r + 1] \). (When we have either no body or just one person, no sharing will occur; and, if there are at least \( r + 1 \) people, it is guaranteed to occur.)

Let \( p_k = P[B = k] \),

\[
E(B) = \sum_{k \geq 0} kp_k = p_1 + 2p_2 + 3p_3 + \cdots
\]

\[
= (p_1 + p_2 + \cdots) + (p_2 + p_3 + \cdots) + (p_3 + p_4 + \cdots) + \cdots
\]

\[
= \sum_{k \geq 0} \left( \sum_{n > k} p_n \right) = \sum_{k \geq 0} P[B > k],
\]

where \( P[B > k] \) is the probability that no birthday sharing occurs with \( k \) people.
What is $P[B > n]$?

When $k \in \{0, 1\}$, no sharing can occur, thus, $P[B > 0] = P[B > 1] = 1$. For $k \geq 2$, it is simply

$$Prob(r, k) = \frac{r(r-1) \cdots (r-k+1)}{r^k}.$$  

Notice that $Prob(r, k) = 0$, for all $k \geq r + 1$. Then,

$$E(B) = \sum_{k \geq 0} \frac{r(r-1) \cdots (r-k+1)}{r^k} = 1 + \sum_{k=1}^{r} \frac{r(r-1) \cdots (r-k+1)}{r^k}.$$  

In particular, taking $r = 365$,

$$E(B) = \frac{12681 \cdots 06674}{5151 \cdots 0625} \approx 24.61658,$$

where the denominator consists of 864 digits.

Thus, collision occurs pretty early. 😊
Operations

To insert a record with its key value being $k$ into a hashing table, we calculate its location $f(k)$, and put it in, if there is nothing there yet. Otherwise, collision happens, and we have to think about where to put it....(?). That is, how should we address the collision issue?

To search for a record with key value, $k$, we simply calculate its possible location $f(k)$, a $\Theta(1)$ computation, and check if there is an element there. If negative, report a failure; otherwise, we have found it (Do we really?).

In the former case, we might add something into the table. In the latter case, we might delete something from the table.

What to do depends on how we will implement this hashing table: as an array, or a more sinister linked list.
Hashing functions

A good hashing function satisfies the assumption of simple uniform hashing: each key is equally likely to be hashed to any of the $m$ slots in the table, independent of the locations of other keys. The goal is even distribution, i.e., about the same number of stuff will be hashed into the same slot.

In practice, we might use some heuristic information to guide us to design a good hash function.

For example, to design a hash function for the symbolic tables, we might notice that closely related strings, such as `print` and `println` often occur in the same program, thus, we have to make sure that they will not be mapped to the same slot.
The division method

When the keys are numeric, and the size of the table is $m$, we can use the following function,

$$h(k) = k \mod m.$$

When keys are not numeric, they have to be converted to non-negative integers first, using, e.g., the ASCII table.

It is shown that, when $m$ is a prime number, the above function leads to an even distribution of elements into an hashing table of size $m$. We will see a bit later why this is preferred.
The multiplication method

This method creates a hashing function in two steps. First, we multiply a key value \(k\) by a constant \(A \in (0, 1)\), and takes the fractional part of the product. We then multiply the value by \(m\) and take the floor of the result. Thus,

\[ h(k) = \lfloor m(kA \mod 1) \rfloor. \]

It works with any constant \(A\), but it works better with some values than with others. It is suggested that we can use the following (Remember \(\phi\)?)

\[ A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887. \]

For example, when \(k = 2\), we have that

\[ h(2) = \lfloor 0.236 \times m \rfloor, \]

which will be a value between 0 and \(m - 1\).
Hashing with chains

We can use a chain structure to implement a hashing table, in which we collect all the elements whose keys are mapped to the same home bucket into a chain, i.e., a singly linked list, as shown in the following picture:
The code...

for this approach is straightforward:

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

**Question:** Why add it at the head?

`CHAINED-HASH-SEARCH(T, x)`
search for an element with x in T[h(key(x))]

**Question:** Will binary search work?

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

The worst-case running time for insertion is $O(1)$, assuming that $x$ is not in the list yet; while deletion can be done in $O(1)$, if the list is a doubly-linked one, which you with play with in Project 5.
For the searching

Assume the hashing table $T$ with $m$ slots contains $n$ elements in all the chains. We define $\alpha$, the load factor, as $n/m$.

In the worst case, all the $n$ elements go to the same slot, creating a list of length $n$. We now have the worst-case scenario for searching: it takes $O(n)$, plus the time to calculate $h(k)$.

But, on average, if every element is equally likely to be hashed into any of the list, called the simple uniform hashing, then, $n_j$, the length of any such list $l_j$, is $\frac{n}{m} (= \alpha)$.

*If the hashing function leads to an even distribution, then the length of each and every linked list will be $\Theta(\alpha)$.\*
A little calculation

Let $N_j$ be the length of $l_j$, and let

$$X^j_i = \begin{cases} 
1 & \text{if element } i \text{ is put in } l_j. \\
0 & \text{otherwise.}
\end{cases}$$

Then, we have that

$$N_j = \sum_{i=1}^{n} X^j_i,$$

and, based on the stuff as we discussed in Chapter 5, we have the following:

$$E(N_j) = \sum_{i=1}^{n} E(X^j_i) = nPr[\text{element } i \text{ goes to } l_j] = \frac{1}{m} \frac{n}{m} = \alpha,$$

which is the average length of the list $N_j$.  

18
Therefore, we have the following:

**Theorem 11.1/2:** In a hash table that solves the collision problem using chains, both successful, and unsuccessful, search takes expected time $\Theta(\alpha)$, under the simple uniform hashing assumption.

Thus, if the number of hash-table slots, $m$, is proportional to the number of elements in the table, $n$, we have $n = O(m)$, thus $\alpha = O(1)$. Then, by the above theorem and other analysis, all the hashing table operations take $O(1)$ time.

**Homework:** Exercises 11.2-2(*), and 11.2-3(*).
Linear open addressing

Open addressing provides an alternative implementation to Hashing table, where all elements are stored within the table itself. Thus, each slot contains either an element or NIL.

When searching for an element, we systematically look for it until we have exhausted the whole table.

To insert an element into this table, we successively check, or probe, the table until we find an available spot, or there is no such slot left, when we declare failure. The probing sequence depends on the key of the element being inserted.

The linear probing starts with \( h(k) \), keeps on checking all the slots one by one,

\[
h(k, i) = (h(k) + i) \mod m.
\]
An example

The following is an example of working with the linear open addressing, where a Hashing table has 11 buckets, from 0 through 10.

After adding 80, 40, and 65, when adding 58, as $58 \% 11 = 3$, and this bucket has already been occupied by 80, we have to put it into bucket 4. The insertion of other numbers are similar.

Homework: Exercises 11.4-1(*) and 11.4-2.
Other approaches

Linear probing is simple, but it tends to form large cluster, which might lead to $O(n)$. 😊

We can use other probing sequences, e.g., the quadratic probing, when

\[ h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m, \]

where both $c_1$ and $c_2 (\neq 0)$ are constants.

We can also use double hashing, when we use two hashing functions as follows:

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m. \]

We start with the position $h_1(k)$, and continue from that point on.
Other approaches

Linear probing is simple, but it tends to form large cluster ([24, 80, 58,35]), which is bad, bad, really bad.

We can also use quadratic probing, when

\[ h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m, \]

where both \( c_1 \) and \( c_2 (\neq 0) \) are constants.

We can use other stuff, such as double hashing, when we use two hashing functions as follows:

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m. \]

We start with the position \( h_1(k) \), and continue from that point on.

The aim is that the collision strategy should not run into the same place as much as possible, as it would lead to \( O(n) \).
Performance analysis

For open hashing, the worst case is $O(n)$, when we have the longest possible cluster formed in the hashing table.

On average, let $\alpha = \frac{n}{m}$ be the loading factor, we have that the expected time for an unsuccessful search is

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

and the expected time for a successful search, also delete, is

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

Finally, the expected time to insert an element into an open hashing table is also

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

This the first efficiency problem addressed through algorithm analysis, done in Summer 1962 by Donald Knuth.