Chapter 11
Hashing Tables

Many applications require a dynamic set that supports such dictionary operations as Insert, Search, and Delete. Check out §1.4 of Aho’s Turing lecture on the course page.

Membership list is a good example. Another one could be the symbolic table, the bedrock of a compiler, where various information of variables are kept to facilitate the translation of a program from one language to another.

When a variable is created, all the associated information such as name (as the key), type, initial value, scope, etc., are inserted into this table. When the compiler runs into a name, a search has to be carried out in the symbolic table to verify its scope, and type, also gets its current value. Finally, when something is no longer needed, e.g., 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)
$T[key[x]] <- x$

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

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

The above simple implementation of the direct-addressing table, $T[n]$, is based on the assumption that “$m$ is not too large”, when we can let $n$ be $m$. Otherwise, it would not be practical. In other words, although saving time 😊, it would lead to poor space efficiency. 😞

**Question:** Can we save both time and space?

**Answer:** Yes, we can. 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, $f(k)$ in $T[n]$, according to its key value $k$.

Hashing table is used a lot in practice.

Notice that, with the direct address table implementation when $m$ is not too large, we can let $f(k) = k$. 
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 in, 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 cannot 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 $k_1, k_2, f(k_1) \equiv f(k_2)$. In other words, multiple things are mapped to the same place.

**Question:** Why is it unavoidable?
Balls and buckets...

We can visualize the hashing problem with balls and buckets: How about randomly throw in $n$ distinguished balls into $m$ distinguishable buckets?

**Question:** How likely multiple balls will be thrown into the same bucket?

When throwing $n = 200$ balls into $m = 100$ buckets, let $P(\text{Max} = k)$ be the probability that the most filled bucket contains $k$ balls, an analysis shows the following:

<table>
<thead>
<tr>
<th>$k$</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(\text{Max} = k)$</td>
<td>$1.4 \cdot 10^{-3}$</td>
<td>0.17</td>
<td>0.46</td>
<td>0.26</td>
<td>0.07</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Thus, in 99% of the time, a most filled bucket holds between 5 through 8 balls, perhaps 6 😊. Indeed, if $m$ is proportional to $n$, i.e., $n/m = \alpha$, the *load factor*, when $n \to \infty$,

\[
\text{Min} = 0, \text{ and, } \text{Max} \approx \log n / \log \log n.
\]

It is clear that, quite likely, multiple data will be hashed into the same data cell. 😊
Birthday related questions

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

**Answer:** 367 for this year of 2024, as this is a leap year with February 29, but 366 for the next year, and all the non-leap years.

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

**Answer:** The probability that none of the $k (\in [1, 365])$ people share the same birthday, for a non-leap year with 365 days, is

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

Notice that, $\forall k \geq 366$, $Prob(365, k) = 0$: 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 \( P_S(365, k) = 1 - Prob(365, k) \).

For example, the probability that two people won’t share the same birthday, \( Prob(365, 2) \), will be simply \( \frac{365}{365} \times \frac{364}{365} = 0.99276 \), thus the probability that they do share the same birthday is \( P_S(365, 2) = 0.00724 \).

By the same token, the probability that three people won’t share the same birthday, \( Prob(365, 3) \), is \( \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} = 0.99180 \). As a result, the probability that they do share the same birthday is \( P_S(365, 3) = 0.00820 \).
We have the answer...

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.

In other words, if you drag in just 23 people, it is more than likely that they will share the same date of birth.

Isn’t this surprising? 😊
In general...

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

**Answer:** $k_r = \frac{1}{2} + \sqrt{\frac{1}{4} + 2r \log_e 2}$, e.g., $k_{365} = 22.7$, consistent with the previous one.

**Question:** How many people should we drag in so that it will be *more than likely* that at least two of them share the same birthday for those who are born on Mars, and Pluto?

There are 687 days on Mars in one year, $k_{687} = 29.8$, we need to drag in 30 people, less than 5% of the size. There are 90,520 days a year in Pluto, you need to drag in 355 people, 0.3%.
Why...

... do we care about this birthday problem when discussing the Hashing table ADT?

**Answer:** Given a table of size $r$, as we saw, we only need to bring in very few data, $\Theta(r^{\frac{1}{2}})$, when it is more than likely that at least two of these data would end up in the same slot in this table.

As another example, given a table of one million slots, if you just bring in 1,178 pieces of data, about 0.12% of the size of the table, it is more than likely some of them would take over the same spot.

*Thus, collision occurs pretty early, and we have to seriously deal with it.* 😊
A few related ones....

**Question:** What is the average (expected) number (Cf. Page 17, Chapter 5 notes) 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 on average we need to drag in about 24 people, just 6.5% of the total.

Check out Page 13 and 14 for technical details.

**Question:** What is the expected number of people we should gather so that at least three of them share the same birthday?

**Answer:** It is also not that big, 89, about 24.4% of the total of 365 birth days.
The general case

**Question:** What is the 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? (There are 687 earth days on Mars.)

**Answer:** Let \( B \) be a random variable whose value is the number of people we get when the first sharing occurs, \( B \in [2, r] \). (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 within \( k \) people.
What is $P[B > k]$?

With either or no person, no sharing can occur, thus, $P[B > 0] = P[B > 1] = 1$. For $k \geq 2$, similar to what we did on Page 6,

$$P[B > k] = \frac{r(r - 1) \cdots (r - k + 1)}{r^k}.$$ 

With at least $r + 1$ people, sharing is guaranteed. Thus, $P[B > k] = 0$, 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.
The collector’s problem

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

**Answer:** An analysis show that, to make this to happen, we need to drag in quite a few: 2,364.

**Question:** What is the expected number of people we should drag in so that all the \( r \) birthdays will be taken by at least one person?

**Answer:** A rather sophisticated analysis shows that the expected number of people we have to collect to exhaust all the \( r \) days is the following:

\[
    r \log r + \gamma r + \frac{1}{2} + O(r^{-1}),
\]

where the Euler’s constant \( \gamma \approx 0.5772156649 \).
Operations

To insert a record with its key value being $k$ into a hashing table, we calculate its location $f(k)$, where $f$ is a hashing function, 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 nothing is there, 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 (Cf. Page 5).

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

For example, to design a hashing 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 by using, e.g., a random number generator(?).
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.

**Question:** Is this one good?

It has been 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.

By the way, a prime number is a natural number that can be wholly divided by 1 and itself, e.g., 2, 3, 5, 7, 11, ... .
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 $\hat{\phi}$, the golden ratio? We played with it in Homework 3.2-6)

$$A = \hat{\phi} = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887.$$

For example, when $k = 2$, we have that

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

which will be a value between 0 and $m - 1$.

**Question:** How to implement a hashing table?
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:

If we use the following hashing function, as 11 is prime.

\[ h(k) = k \mod 11. \]

Then, e.g., \( h(36) = 36 \mod 11 = 3. \)
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 will play with in Project 5.

“Programs = algorithms+data structures.” by Niklaus Wirth, 1976.
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 $n/m$, essentially the average length of a chain. (Cf. Page 6)

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 $h(k)$. 😞

With an even distribution, 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 $\frac{n}{m} (= \alpha)$.

In other words, if the hashing function leads to an even distribution, then the length of each and every linked list will be $\Theta(\alpha)$. This would lead to a $\Theta(1)$ cost for search, as well. 😊

**Question:** Why?
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 on Page 20 in Chapter 5, we have the following:

$$E(N_j) = \sum_{i=1}^{n} E(X^j_i)$$

$$= n \times \Pr[\text{element } i \text{ goes to } l_j]$$

$$= n \times \frac{1}{m} = \frac{n}{m} = \alpha,$$

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

**Question:** Where is the equal likelihood assumption?
Finally,…

Therefore, we have the following:

**Theorem 11.1/2:** In a hashing 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 hashing 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 as made on Page 19, 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 the *Hashing table*, where all the 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 delete an element, we also look for it first.

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,

$$h(k, i) = (h(k) + i) \mod m.$$
I want to see...

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.

![Diagram](image-url)
Other approaches

Linear probing is simple, but it tends to form large cluster ([24, 80, 58, 35]), as shown on Page 24, 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.

For example, setting \( c_1 = c_2 = 1 \), for \( k = 58 \),

\[ h(58, 0) = h(58) \% 11 = 3 \% 11 = 3, \]

which leads to a large cluster.

But, we have

\[ h(58, 1) = (h(58) + 1 + 1^2) \% 11 = 5 \% 11 = 5, \]

Similarly,

\[ h(35, 1) = (h(35) + 1 + 1^2) \% 11 = 4 \% 11 = 4, \]

which also leads to a large cluster, but

\[ h(35, 2) = (h(35) + 2 + 2^2) \% 11 = 8 \% 11 = 8. \]
Is there another game?

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, if this one leads to a larger cluster, continue from that point on, using another hashing function \( h_2 \) to find an alternative position.

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

You should have really understood this process by going through the following assignment, which I have. 😊

**Homework:** Exercise 11.4-1
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. (Cf. Section 6.4 of Searching and Sorting, a classic textbook by Prof. Knuth.)