Chapter 3

Hashing

3.1 Introduction

We explore data structures that support the following operations: Given a **universe** of elements $U = \{0, 1, 2, ..., u-1\}$, and a set $S \subseteq U$ containing elements, we want to support the following operations:

- $insert(x)$—add $x \in U$ to S; i.e., $S \leftarrow S \cup \{x\}$.
- $find(x)$— determine whether element $x \in S$.
- $delete(x)$–remove element x from S; i.e., $S \leftarrow S - \{x\}$.

The elements $\in U$ are keys belonging to records.

Question. What is the best data structure to solve this problem?

Answer. Hash Tables.

Quote of Udi Member, former CTO of Yahoo:

> "The three most important data structures are hashing, hashing and hashing."

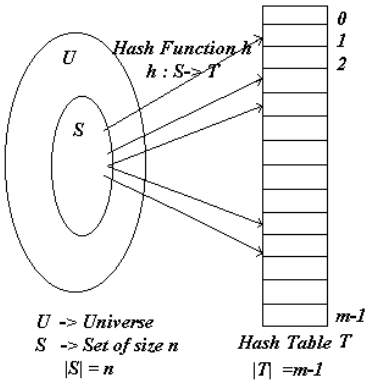


Figure 3.1: Hashing.

3.2 Main Idea of Hashing

Use the hash function h to map the universe $U = \{0, \ldots, u-1\}$ of keys to the set $T = \{0, ..., m-1\}$.

3.3 Collisions

Definition 23. *When a record x maps to an already occupied slot in T, a **collision** occurs.*

3.3.1 Avoiding Collisions by Chaining

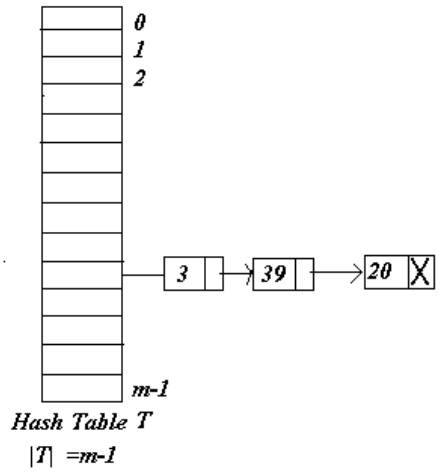


Figure 3.2: Chaining: resolving collision by chaining. Records in same slot are linked by list. In this example, $h(3) = h(39) = h(20) = j$.

3.3.2 Resolving Conflicts by Open Addressing

No storage is used outside of hashtable itself. Insertion systematically probes the table until an empty slot is found.

Main Idea: Hash function depends *both* on the key and probe number.

- $insert(k)$ - Try to insert key k into position $h(k, 0)$.
  If position $h(k, 0)$ has already been filled, then try $h(k, 1)$.
  If position $h(k, 1)$ has already been filled, then try $h(k, 2)$, etc, ...
- $delete(k)$ - Very slightly more difficult but still pretty straightforward.
- $search(k)$ - Keep searching in postions $h(k, 0)$, $h(k, 1)$, $h(k, 2)$,... until either k is found or and empty cell is found.

Linear Probing

Given an ordinary hash function $h(k)$, linear probing uses the hash function

$$\tilde{h}(k, i) = h(k) + i \bmod m$$

Disadvantage: Suffers from *clustering.* (Is this really a disadvantage?)

Double Hashing

Given two hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

$$\tilde{h}(k, i) = h_1(k) + i \cdot h_2(k) \bmod m.$$

Requirement:

1. $h_2(k)$ should be relatively prime to m.
2. Make $m = 2^\ell$ and $h_2(k)$ should be odd.

3.3.3 Myth

You probably learned in your undergraduate class that deletions are hard when there is linear probing. This is just not true.

3.3.4 Modern Memory Hierarchy

The folk wisdom says that chaining is better than double hashing, which is better than linear probing. Linear probing is considered to be the worst because it induces clustering which tends to cause long runs.

This wisdom is being turned on its head by modern memory hierarchies as explained below.

In a modern memory hierarchy, memory is divided into words, blocks, etc. Memory transfers are often the dominant cost of running time. The number of memory transfers can be reduced if data is read and written sequentially. As a result, it is better to read sequentially rather than randomly. This is achieved best by using linear probing.

The arrow shows the cells queried during linear probing. As we can see, even if all the cells indicated by the arrow are touched, it results in just two memory transfers.

$\Rightarrow$ Linear probing has locality of reference.
$\Rightarrow$ Linear probing results in fewer number of memory transfers.

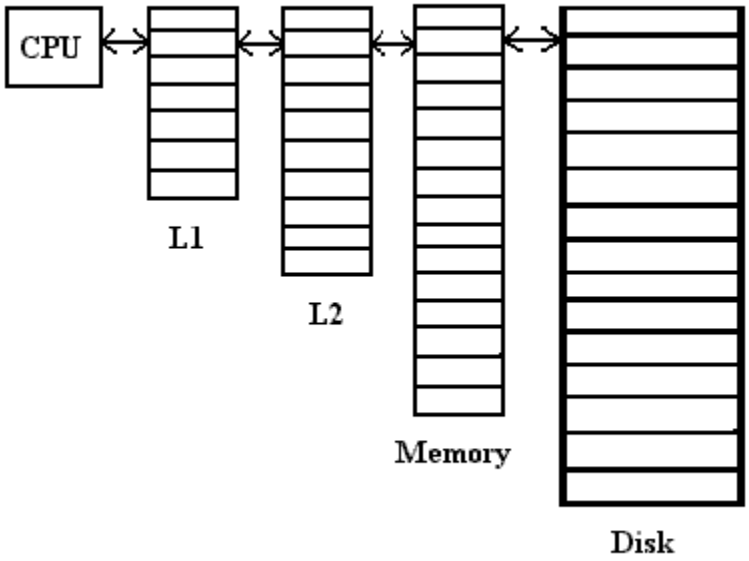


Figure 3.3: Memory hierarchy.

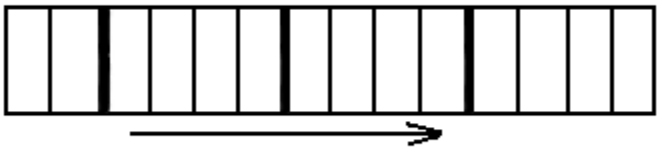


Figure 3.4: Memory is divided into contiguous fixed-sized blocks.

3.4 Universal Hashing

3.4.1 Weakness of Hashing

For any hash function mapping from a universe $U = \{0, 1, \cdots, u-1\} \longrightarrow \{0, 1, \cdots, m-1\}$, there exists a set S, $|S| = n$, such that lookups have very slow running time. (Just choose a set of keys that all hash to the same value, i.e., $S \subseteq \{x|\ h(x) = y, \text{for some } y\}$.)

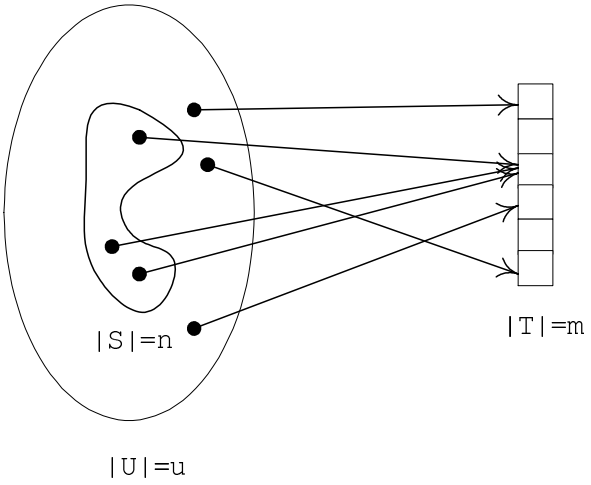


Figure 3.5: Hashing from $U = \{0, 1, \cdots, u-1\}$ to $T = \{0, 1, \cdots, m-1\}$. We choose a set $S \subseteq U$ such that all the elements of S are hashed to the same value in T. The lookups on S will have a very slow running time.

3.4.2 Universal Hash Functions

Definition 24. *We are given a set $\mathcal{H}$ of hash functions mapping from $U \to \{0, 1, \cdots, m-1\}$, such that $\forall x, y \in U$, where $x \neq y$:*

$$\left|\{h_a \in \mathcal{H} | h_a(x) = h_a(y)\}\right| = \frac{|\mathcal{H}|}{m}.$$

That is, the probability that x and y collide is $\frac{1}{m}$, if we choose h_a randomly from $\mathcal{H}$.

Claim 25. *Let h_a be a hash function chosen uniformly at random from a universal set $\mathcal{H}$ of hash functions. Suppose that h_a is used to hash n arbitary keys into m slots of the table. Then for any given key x:*

$$E[\#\ of\ collisions\ with\ x] = \frac{n-1}{m}.$$

Proof. Let C_{xy} be the random variable that indicated whether x and y collide, then

$$C_{xy} = \begin{cases} 1 & \text{with probability } \frac{1}{m} \\ 0 & \text{with probability } 1 - \frac{1}{m}. \end{cases}$$

Then,

$$E[C_{xy}] = \frac{1}{m} \cdot 1 + \left(1 - \frac{1}{m}\right) \cdot 0 = \frac{1}{m}.$$

Let C_x be the random variable denoting the number of collisions with x. Thus,

$$C_x = \sum_{y \in S - \{x\}} C_{xy}.$$

By linearity of expection we know that

$$\begin{aligned} E[C_x] &= \sum_{y \in S - \{x\}} E[C_{xy}] \\ &= \sum_{y \in S - \{x\}} \frac{1}{m} \\ &= \frac{n-1}{m}. \end{aligned}$$

□

3.4.3 Construction of a Universal Set of Hash Functions

Let m be prime. Let key k be divided into r components, for any $r = \max\{\log_m u, 2\}$ (where u is the size of universe to be hashed). That is,

$$k = (k_0, k_1, k_2, \cdots, k_{r-1}) \qquad (k_i \in \{0, 1, \cdots, m-1\}).$$

That is, k is represented in base m. Choose a random vector

$$a = (a_0, a_1, a_2, \cdots, a_{r-1}) \qquad (a_i \in \{0, 1, \cdots, m-1\}).$$

That is, a_i is also represented in base m. Define the hash function

$$h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m.$$

Next we prove that we have a universal set of hash functions.

Claim: $|\mathcal{H}| = m^r$.

Claim: The set $|\mathcal{H}| = \{h_a\}$ is universal, that is,

$$\text{Prob}(x, y \text{ collide}) = \frac{1}{m},$$

or

$$\left|\{h_a \in \mathcal{H} | h_a(x) = h_a(y)\}\right| = \frac{|\mathcal{H}|}{m}.$$

Proof. Let

$$\begin{aligned} x &= (x_0, x_1, \cdots, x_{r-1}), \\ y &= (y_0, y_1, \cdots, y_{r-1}), \qquad \text{where } x \neq y. \end{aligned}$$

Assume that:

$$h_a(x) = h_a(y).$$

That is,

$$\begin{aligned} \sum_{i=0}^{r-1} a_i x_i &= \sum_{i=0}^{r-1} a_i y_i \qquad (\mod m), \\ \sum_{i=0}^{r-1} (x_i - y_i) a_i &= 0 \qquad (\mod m). \end{aligned}$$

Since $x \neq y$, there is at least one digit that differs. Without loss of generality, say that $x_0 \neq y_0$. Then,

$$(x_0 - y_0) a_0 = -\sum_{i=1}^{r-1} (x_i - y_i) a_i \qquad (\mod m).$$

Now we use the following fact:

> **Fact:** Let m be prime, for any $z \in Z_m = \{0, 1, 2, \cdots, m-1\}$, where $z \neq 0$, there exists a unique $z^{-1} \in Z_m$, such that $z \cdot z^{-1} = 1 \ (\mod m)$.

Therefore, there exists the inverse of $x_0 - y_0$. So we have:

$$a_0 = (x_0 - y_0)^{-1} \left(\sum_{i=1}^{r-1} (y_i - x_i) a_i \right) \qquad (\mod m).$$

That's why we require the m to be prime.

Question: How many degrees of freedom do we have? i.e., how many h_a's satisfy above requirement (cause x and y to collide)?
Answer: There are **m choices** for each of **the r − 1 numbers** $a_1, a_2, \cdots, a_{r-1}$. But once these are chosen, exactly **one choice** for a_0 causes x and y to collide:

$$a_0 = (x_0 - y_0)^{-1} \left(\sum_{i=1}^{r-1} (y_i - x_i) a_i \right) \qquad (\mod m).$$

Therefore, there are m^{r-1} hash functions satisfying the above requirement.

Thus, the number of h_a's that cause x and y to collide is:

$$m^{r-1} = \frac{|\mathcal{H}|}{m}.$$

□

3.5 Perfect Hashing

Claim: If we hash n keys into a hash table of size n^2, then

$$\text{Prob(There exists collision)} < \frac{1}{2}.$$

Proof. Let random variable

$$C_{xy} = \begin{cases} 1 & \text{if } h_a(x) = h_a(y) \\ 0 & \text{otherwise.} \end{cases}$$

Because we have a universal hash function

$$C_{xy} = \begin{cases} 1 & \text{with probability } \frac{1}{m} \\ 0 & \text{with probability } 1 - \frac{1}{m}. \end{cases}$$

Therefore,

$$E[C_{xy}] = \frac{1}{m}.$$

Let C be the number of pairwise collisions:

$$C = \sum_{x,y \in S,\ x \neq y} C_{xy}.$$

By linearity of expectation,

$$\begin{aligned} E[C] &= \sum_{x,y \in S,\ x \neq y} E[C_{xy}] \\ &= \sum_{x,y \in S,\ x \neq y} \frac{1}{m} \\ &= \frac{\binom{n}{2}}{m} \\ &= \frac{n(n-1)/2}{n^2} \\ &< \frac{1}{2}. \end{aligned}$$

So the expected # of pairwise collision is less than $\frac{1}{2}$.

Using Markov's Inequality:*

Markov's Inequality: Let X be a nonnegative random variable,

$$\Pr(X \geq \alpha E[X]) \leq \frac{1}{\alpha}.$$

We obtain:

$$E[C] \leq \frac{1}{2}.$$

The previous equation means that

$$\Pr(\geq 1 \text{ collisions}) < \Pr(C \geq 2E[C]) \leq \frac{1}{2}.$$

□

3.6 Perfect Hashing Using Linear Space

Difficulty: Bins may have many balls, i.e., elements.

Fact: The fullest bin may have $O(\sqrt{n})$ balls (elements).

Great idea: Use a 2-level hash function!

*This is another thing you should remember on your deathbed.

- 1st level: A table of size n. But there will be many collisions.

  Definition 26. *Let $N(i)$ be the # of elements hashed to position i. That is,*

  $$N(i) = |\{k \in S | h(k) = i\}|$$

- 2nd level: Associated with position i is its *own* table of size $(N(i))^2$. This table has all keys such that $h(k) = i$. Each table has its *own* hash function.

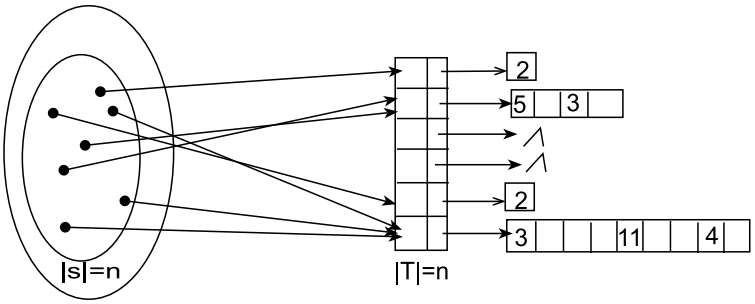


Figure 3.6: Two-level hash function.

We already know that this is a perfect hash function (from the previous section). We need to show that this construction uses linear space . This is not obvious , since the space usage is

$$\Theta(n) + \sum_{i=1}^{n} \Theta(N(i))^2,$$

and, a priori, there is no reason to assume that this equation is linear.

3.7 Analysis of Space Usage in Perfect Hashing

Amazingly, the space usage is only $\Theta(N)$ and that is what remains to prove.

Theorem 27. *The space usage for the perfect hashing construction is $\Theta(N)$.*

Main Beautiful Idea: We show that the space usage is linear by counting the # of collisions in first level in *2 ways*.

3.7.1 First way with random variables

Lemma 28. *The expected number of pairwise collisions is $\frac{n-1}{2}$.*

Proof. Let random variable C_{ij} be:

$$C_{ij} = \begin{cases} 1 & \text{if elements } i \text{ and } j \text{ land in the same bin.} \\ 0 & \text{otherwise.} \end{cases}$$

Then ,

$$C_{ij} = \begin{cases} 1 & \text{with prob } \frac{1}{n} = \frac{1}{m}. \text{ (We are letting } m = n.) \\ 0 & \text{with prob } 1 - \frac{1}{n}. \end{cases}$$

Therefore, $E[C_{ij}] = \frac{1}{n}$.

Now let C be the total # of pairwise collisions, hence $C = \sum_{i<j} C_{ij}$.

Therefore,

$$\begin{aligned} E[C] &= \sum_{i<j} E[C_{ij}] \\ &= \binom{n}{2} \cdot \frac{1}{n} \\ &= \frac{n-1}{2}. \end{aligned}$$

□

3.7.2 Second way

Theorem 29. *The expected # of collisions is*

$$\sum_{i=1}^{n} \binom{N(i)}{2} = \sum_{i=1}^{n} \Theta(N(i))^2.$$

Proof. If there are $N(i)$ keys that hash to position i then the expected # of (pairwise) collisions is the # of ways of choosing subsets of size 2, i.e., $\binom{N(i)}{2}$. □

But the space usage of the data structure is:

Fact 30. *The space usage of the data structure is* $\Theta(n) + \sum_{i=1}^{n} \Theta(N(i))^2$.

Therefore combining these two observations, we obtain

Theorem 31. *The space usage of the pefect-hashing data structure is* $\Theta(N)$.

Other refs. Cite static perfect hashing. Who did dynamic perfect hashing? What are the results?

3.8 Stronger Kinds of Universal Hashing

We're going to do dynamic hashing, and for that we need to be a little bit more careful about our hash functions. [2] So here goes:

Definition 32. *A set $\mathcal{H}$ of hash functions is said to be a* weak universal family *if for all $x, y \in U$, $x \neq y$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x) = h(y)] = \frac{O(1)}{m}.$$

In particular, it is c-universal iff

$$\Pr[h \leftarrow \mathcal{H} : h(x) = h(y)] \leq \frac{c}{m}.$$

Example: $\mathcal{H}_{p,m} = \{h_{a,b} \mid a \in \{1, 2, \ldots, p\}, b \in \{0, 1, 2, \ldots, p-1\}\}$, for some prime $p > |U|$, where $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Each function requires only $O(\log |U|)$ bits to represent, and we can evaluate it in constant time. This class of functions is 2-universal.

Definition 33. *A set $\mathcal{H}$ of hash functions is said to be a* strong universal family *if for all $x, y \in U$ such that $x \neq y$, and for all $a, b \in [m]$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x) = a \wedge h(y) = b] = \frac{O(1)}{m^2}.$$

Definition 34. *A set $\mathcal{H}$ of hash functions is said to be a k-*independent *if for all k distinct items $x_1, x_2, \ldots, x_k \in U$, and for all $a_1, a_2, \ldots, a_k \in [m]$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \ldots \wedge h(x_k) = a_k] = \frac{O(1)}{m^k}.$$

Similarly, if

$$\Pr[h \leftarrow \mathcal{H} : h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \ldots \wedge h(x_k) = a_k] \leq \frac{c}{m^k}.$$

*We refer to the family as (c, k)-*independent*.*

Example: Pick some $p > |U|$.

$\mathcal{H} = \{h \mid h(x) = (c_0 + c_1 x + \ldots + c_{k-1} x^{k-1}) \bmod m$, for some $c_0, c_1, \ldots c_{k-1} \in [p]\}$.

To see that universal hashing gives what we want, it's enough to show *weak universal family* does, suppose we pick m so that $\frac{n}{m} = O(1)$, and let h be a random element of $\mathcal{H}$. Now,

[2] From Eric Demaine's lecture notes.

an operation that involves item x has running time proportional to the length of the chain that x is in, which is equal to $\sum_{y \in S} I_y$, where I_y is the indicator random variable that is 1 if and only if $h(x) = h(y)$. So, the expected length (and the expected running time) is

$$E\left[\sum_{y \in S} I_y\right] = \sum_{y \in S} E[I_y] = 1 + \sum_{y \neq x} \Pr[h(x) = h(y)] \leq 1 + n \cdot \frac{O(1)}{m} = O(1)$$

Theorem 35 (Siegel, 1989)**.** *For any $\varepsilon > 0$, there exists a $n^{\Omega(1)}$-independent family of hash functions such that each function can be represented in n^{ε} space, and can be evaluated in $O(1)$ time.*

Theorem 36 (Pagh, Ostlin, 2003)**.** *There exists a n-independent family of hash functions such that each function takes $O(n)$ words to describe, and can be evaluated in $O(1)$ time.*

3.9 Cuckoo - Dynamic Hashing (Pagh and Rodler 2001 [?])

Theorem 37 (Cuckoo hashing performance)**.** *Cuckoo hashing achieves:*

- *insert – in $O(1)$ expected time*
- *queries/deletes – in $O(1)$ worst-case time*

Where cuckoo hashing gets its name. Cuckoo hashing is based on the nesting habit of the Cuckoo bird, which is known to place its eggs in the unattended nests of other birds. When the baby cuckoo birds are born, the surrogate mother bird feeds them. The baby cuckoo is typically born first, and pushes some of the unhatched eggs out of the nest.

Requirement for $(c, O(\log n))$-independent hash functions. Unlike static perfect hashing, cuckoo hashing make uses of $(c, O(\log n))$-independent hashing. Whether the scheme can achieve the same bound using only $O(1)$-independent hash family is still an open problem.

The scheme requires two $(c, O(\log n))$-independent hash functions, say h_1 and h_2. (We'll use $6 \log n$-independent hash functions.)

Parameter Choices. The table size m is required to be greater than $2n$. We'll use $m = 4n$. Throughout the life of the data structure, it maintains the following invariant:

Invariant 3.9.1. *An item x that has already been inserted is stored either at hash-table location $T[h_1(x)]$ or at $T[h_2(x)]$.*

Invariant 3.9.2. *A query takes at most two probes in the worst case. So does a delete.*

Before presenting the algorithm, we introduce some notation:

Definition 38. *We let $h(x)$ represent the location that most recently holds x, that is,*

$$h(x) = \begin{cases} h_1(x) & \text{if } x \text{ was most recently put in } h_1(x) \\ h_2(x) & \text{if } x \text{ was most recently put in } h_2(x) \\ \text{undefined} & \text{if } x \text{ is not yet in either location} \end{cases}$$

We let $h(x)$ represent the alternate location, that is,

$$\bar{h}(x) = \begin{cases} h_1(x) & \text{if } h(x) = h_2(x) \\ h_2(x) & \text{if } h(x) = h_1(x) \\ \text{undefined} & \text{if } x \text{ is not yet in either location} \end{cases}$$

Definition 39.

- *We say that x **bumps** y if we remove y from its location and replace it with x, that is, $h_1(x) = h(y)$ (or $h_2(x) = h(y)$), and we remove y from the table and set $h(x) = h_1(x)$ (or $h(x) = h_2(x)$).*
- *We **toggle** y if we move y from one of its possible locations to the other in the hash table, that is if we set $h(y)$ to be $\bar{h}(y)$*
- *A **toggle and bump** happens when the location we toggle to is full and a **toggle and stick** happens when the location we toggle to is empty.*

We know how to query and delete.

To insert an element x:

1. Compute $h_1(x)$. If $T[h_1(x)]$ is empty, we put x there, and we are done. Otherwise, bump whatever is in $T[h_1(x)]$ and toggle it.
2. This produces a chain reactions of toggle and bump, until one of two conditions have been met. Either there is a toggle and stick or we have bumped $6 \log n$ elements. In the latter case, pick two new hash functions and rehash the entire contents of the table.

It remains is to analyze the running time of inserting an item. Consider the process of inserting an item x_1. Let $x_1, x_2, \ldots, x_t$ be the sequence of items, with the exception of x_1, that are bumped during the process, in the order they are bumped.

There are three cases.

- **Case (a): No cycle.** The process continues, without coming back to a previously visited cell, until an empty cell is found or rehashes.
- **Case (b): One cycle.** Otherwise, the process comes back to a cell that it has already visited. That is, for some j, the other cell that x_j can be in is occupied by a previously evicted item x_i. Then x_i will be evicted a second time, and will go to the other position it can be found in–namely, back to the location of x_{i-1}. Thus, $x_i, x_{i-1}, \ldots, x_1$ will be evicted in that order, and x_1 will be sent to $T[h_2(x)]$, and the sequence continues.

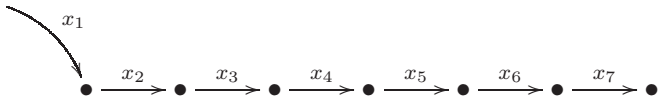
  The sequence continues until it ends at an empty cell or rehashes.
- **Case (c): Two cycles.** Otherwise the sequence continues until, once more, it runs into a previously visited cell. In this case, we have discovered an infinite loop, so you rehash. (In fact, you can stop as soon as you find 2 cycles, but this doesn't affect the run time.)

We can visualize the above behaviors by considering the cuckoo graph G, whose vertex set is $[m]$ and whose edge set contains edges of the form $(h_1(x), h_2(x))$ or $(h_2(x), h_1(x))$ for all $x \in U$. In this way, the process of inserting item x_1 can be viewed as a walk on G starting at $h(x_1)$. Visualizations of the three different behaviors are given in Figure 3.7

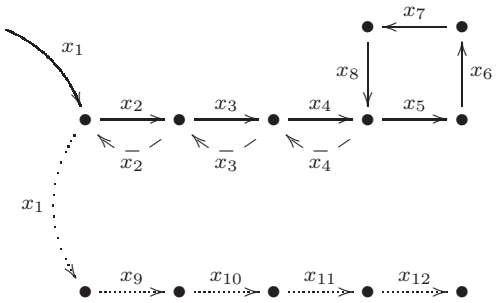
We analyze the running time in all three cases. The key observation is that when inserting a new element, we never examine more than $6 \log n$ items. Since our functions are $6 \log n$-independent, we can treat them as truly random functions.

- **Case (a): No cycle.** We calculate the probability that the insertion process evicts t items. The process carries out the first eviction if and only if $T[h_1(x_1)]$ is occupied. By the union bound,

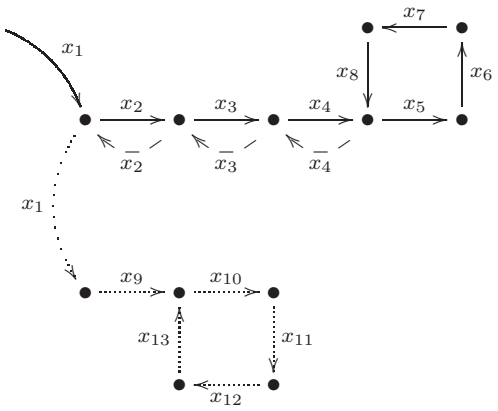
$$\begin{aligned} \Pr[\geq 1 \text{ eviction}] &\leq \sum_{x \in S, x \neq x_1} (\Pr[h_1(x) = h_1(x_1)] + \Pr[h_2(x) = h_1(x_1)]) \\ &< 2\frac{n}{4n} = \frac{1}{2}. \end{aligned}$$



(a) no cycle



(b) one cycle



(c) two cycles

Figure 3.7: Three different behaviors of the process of inserting element x_1.

Similarly,

$$\Pr[\geq 2 \text{ evictions}] \leq \frac{1}{4}.$$

and in general,

$$\Pr[\geq \text{t evictions}] \leq 2^{-t}.$$

Therefore,

$$E[\text{running time}] \leq \sum_{t=1}^{\infty} t2^{-t} = O(1).$$

Also,

$$\Pr[\text{rehash}] \leq 2^{-6\log n} = \frac{1}{n^6} < \frac{1}{n^2}.$$

- **Case (b): One cycle.**

  Claim 40. *In the sequence $x_1, x_2, \ldots, x_t$ of evicted items, there exists a consecutive subsequence distinct items of length at least $t/3$ that starts with x_1 (see Figure 3.7(b)).*

  Proof. Consider Figure 3.7(b). The sequence can be partitioned into three parts — the solid line part, the dashed line part, and the dotted line part — and one of them must contains at least $t/3$ items. □

  By the same reasoning as in the previous case, we have

  $$\Pr[\geq \text{t evictions}] \leq 2^{-t/3},$$

  and

  $$E[\text{running time}] \leq \sum_{t=1}^{\infty} t2^{-t/3} = O(1).$$

  Also,

  $$\Pr[\text{rehash}] \leq 2^{\frac{-6\log n}{3}} = \frac{1}{n^2}.$$

- **Case (c): Two cycles.** We now calculate the probability of a sequence of length t having two cycles. We use a counting argument.

  Question. How many two-cycle configurations are there?

  - The first item in the sequence is x_1.
  - There are at most $t-1$ items in the sequence, each of which is drawn from a set of size n. *Thus, the number of sequences of elements starting with x_1 is at most n^{t-1}.*

- There are at most t choices for when the first loop occurs, at most t choices for where this loop returns on the path so far, and at most t choices for when the second loop occurs. *Thus, there are t^3 choices for describing the structure of the loop.*

  Question. Why isn't it t^4? I don't know the answer, but it doesn't seem to matter.

- Additionally, this pattern can occur anywhere in our table (data structure) of hash values. While the first hash value is $h_1(x_1)$, the remaining $t-1$ values are unconstrained. *Our table is of size $m = 4n$, so there are $(4n)^{t-1}$ possibilities for where the elements hash to.*

Claim 41. *There are at most $t^3 n^{t-1}(4n)^{t-1}$ configurations.*

However, to calculate the total number of cuckoo graphs, note that each edge in the graph comes from the evaluation of two different hash functions on a single input. The hash functions have a range of size $m = 4n$, so the number of possible locations for a single edge is $\frac{(4n)^2}{2}$ (where the division by two occurs because the edge is undirected, so an edge from i to j is the same as an edge from j to i). Therefore, the total number of cuckoo graphs (of t nodes) is $\frac{(4n)^{2t}}{2^t}$. Thus, the probability that a two-cycle configuration occurs is at most

$$\frac{t^3 n^{t-1}(4n)^{t-1}2^t}{(4n)^{2t}} = \frac{t^3}{4n^2 2^t}.$$

Therefore, the probability that a two-cycle occurs at all is at most

$$\sum_{t=2}^{\infty} \frac{t^3}{4n^2 2^t} = \frac{1}{4n^2}\sum_{t=2}^{\infty}\frac{t^3}{2^t} = \frac{1}{4n^2}\cdot O(1) = O\left(\frac{1}{n^2}\right).$$

So, an insertion causes the data structure to rehash with probability $O(1/n^2)$. Therefore, n insertions can cause the data structure to rehash with probability at most $O(1/n)$. Thus, rehashing, which is basically n insertions, succeeds with probability $1-O(1/n)$, which means that it succeeds after a constant number trials in expectation. In a successful trial, every insertion must fall into the first two cases. Therefore, a successful trial takes $n \times O(1) = O(n)$ time in expectation. In an unsuccessful trial, however, the last insertion can take $O(\log n)$ time, so it takes $O(n) + O(\log n) = O(n)$ time in expectation as well. Since we are bound to be successful after a constant number of trials, the whole process of rehashing takes $O(n)$ time in expectation. Hence, the expected running time of an insertion is $O(1) + O(1/n^2) \cdot O(n) = O(1) + O(1/n) = O(1)$.

3.10 Worst-case Guarantees in Static Hashing

Still, universal hashing gives us only good performance in expectation, making it vulnerable to an adversary who always insert/query items that make the data structure perform the worst. In static hashing, however, we can establish some worst-case upper bounds.

Theorem 42 (Gonnet, 1981)**.** *Let $\mathcal{H}$ be an n-independent family of hash functions. The expected length of the longest chain is $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$.*

By this theorem, we can construct a static hash table with $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$ worst-case running time per each operation. We start by picking a random hash function from the family, hash every item in S, and see if the length of the longest chain is at most twice the expected length. If so, we stop. If not, we pick a new function and start over again. Since the probability that we pick a bad hash function is at most $\frac{1}{2}$, we will find a good hash function after a constant number of trials. The construction thus takes $O(n)$ time in expectation.

3.11 Minimum Two Choices

As we'll see later, we can do better than this. By using two hash functions, adding the inserted item to the shorter list, and searching in both lists when an item is queried, we achieve $\Theta(\lg \lg n)$ length of longest chain in expectation. (More generally, for d hash functions, the longest chain is of length $\Theta(\frac{\log \log n}{\log d})$.)