Binary search trees and skip lists are dictionaries where the fundamental operation is a comparison. Similar to radix sort, we may alternatively develop a data structure based on the idea of storing an item at a location that depends on the item’s key. Here we distinguish between an item and its key, stipulating that the item contains additional information that is also stored but not used in searching. In radix sort we use the method of direct addressing, where an item with key \( x \) is stored in \( T[x] \). This is illustrated in Figure 37. The key \( x \) must be an integer, else we need a bijection \( f \) between the set of keys and the indices of the table and we store the item with key \( x \) in \( T[f(x)] \). In most situations direct addressing is impractical because the range of keys (the universe) is much larger than the number of keys to be stored.

**Hashing.** In hashing we store \( x \) at a location \( h(x) \), where \( h \) is a function that is easy to compute and spreads the keys over the table. Since the range (the universe) is usually much larger than the domain (the table), \( h \) can generally not be injective. We introduce some terminology:

- \( U \) is the universe.
- \( K \subset U \) is the set of keys, \( n = \text{card} \ K \).
- \( T[0..m - 1] \) is the hash table.
- \( h : U \to \{0, 1, \ldots, m - 1\} \) is the hash function.

The item with key \( x \) is stored in \( T[h(x)] \), unless this slot is already occupied because \( h(x) = h(y) \) for a key \( y \neq x \). Such an event is a collision. Collisions cannot be avoided if \( m < \text{card} \ U \), which is the usual case. The most straightforward method in dealing with collisions uses buckets: the elements of \( T \) store pointers to linked lists rather than the items themselves, as illustrated in Figure 38. This is called hashing with chaining. The three dictionary operations translate into array indexing plus searching in a linked list. In the context of hashing, we often refer to a search as a look-up operation.

- **LOOKUp** (\( x \)): do linear search in \( T[h(x)] \).
- **INSERTion** (\( x \)): add \( x \) to list \( T[h(x)] \).
- **DELETion** (\( x \)): remove \( x \) from list \( T[h(x)] \).

Assuming \( h(x) \) can be computed in constant time, we have time \( O(1) \) for an insertion, while the time for a look-up and a deletion depends on the length of the list.

**Average Analysis of Chaining.** The load factor of a hash table is the number of keys stored in the table divided by the size of the table, \( \alpha = \frac{n}{m} \). This ratio \( \alpha \) is also the average number of items per linked list. The analysis assumes that \( h \) takes constant time to compute and that

\[
\text{Prob}[h(x) = i] = \frac{1}{m},
\]

for each \( 0 \leq i < m \). The second condition is for example satisfied if \( h \) spreads the universe uniformly over
ing at least one pointer for a new key is appended to the appropriate list. For a successful look-up we need a different argument because long lists are searched more often. In other words, the probability of a long list to be searched is higher than the probability of a short list. Of course, searching in a long list takes longer than in a short list. We do the analysis indirectly, by pretending we change the insertion strategy such that a new key is appended to the appropriate linked list rather than added at the top (which would be easier). This is illustrated in Figure 39. Looking up \( x \) takes one more step than inserting \( x \) (assuming this new and less efficient insertion strategy). At the time we insert the \( i \)-th key, the average list length is \( \frac{i-1}{m} \). The average number of steps to insert \( n \) keys is therefore

\[
\frac{1}{n} \cdot \sum_{i=1}^{n} (1 + \frac{i-1}{m}) = 1 + \frac{1}{n \cdot m} \cdot \sum_{i=1}^{n} (i - 1) = 1 + \frac{n - 1}{2m} = 1 + \frac{\alpha - 1}{2m}.
\]

Hence, the average number of steps for a single insertion is less than \( 1 + \frac{\alpha}{2} \) and the average number of steps for a successful look-up is \( S(n, m) < 2 + \frac{\alpha}{2} \). Note that for \( \alpha > 2 \) this is less than for an unsuccessful look-up. This shows that lists significantly longer than the average are rare.

**Linear Probing and Double Hashing.** To avoid following at least one pointer for each look-up we can store the items in the table itself. For this to work we need to restrict the load factors to values \( \alpha \leq 1 \). If a collision happens we find another slot. In other words, each key goes through a sequence of probes: \( h(x) = H(x, 0), H(x, 1), \ldots, H(x, i) \). The linear probing strategy is characterized by

\[
H(x, i) = H(x, i - 1) + 1 \pmod{m}.
\]

The disadvantage of this method is primary clustering: areas of collisions grow and cause more and more collisions. Primary clustering can be avoided by double hashing:

\[
H(x, i) = H(x, i - 1) + h_2(x) \pmod{m},
\]

where \( h_2 \) is a second hash function. The probe sequence is a cycle that eventually returns to the first probe location. The cycle exhausts the entire table iff \( m \) and \( h_2(x) \) are relative prime, that is, there is no integer larger than \( 1 \) that divides both \( m \) and \( h_2(x) \) without remainder. The easiest way to guarantee \( m \) and \( h_2(x) \) are relative prime is to choose \( m \) prime and \( h_2(x) > 0 \) strictly smaller than \( m \).

We consider again the three dictionary operations. To insert the item with key \( x \) we traverse the cycle that starts at \( T[(h(x)] \) and store the item in the first empty slot. If no empty slot is found then the table must be full. To do further insertions we need to rebuild the hash table for a larger \( m \). LookUp also does a linear search in the cycle that starts at \( T[h(x)] \). If the search ends at an empty location then the key is not in the table. Otherwise, we either find the key or we traverse the entire table and eventually return to the start position. In the latter case, the search is unsuccessful and we know the table is full and should be rebuilt. To delete an item is more complicated since gaps in the probe sequences interfere with the look-up algorithm. Deletions are therefore replaced by marking an item as deleted, but the item still occupies the slot. If the number of marked items gets large relative to \( m \) then the table should be rebuilt as before. Hashing with open addressing is more difficult to analyze than hashing with chaining. We therefore summarize the performance by stating the bounds without proof. Under some simplifying assumptions the average number of steps for unsuccessful and successful look-up for linear probe sequences is

\[
U(n, m) = \frac{1}{2} + \frac{1}{2(1 - \alpha)^2},
\]

\[
S(n, m) = \frac{1}{2} + \frac{1}{2(1 - \alpha)}.
\]

The average numbers of steps for double hashing is

\[
U(n, m) = 1 + \frac{1}{1 - \alpha},
\]

\[
S(n, m) = \frac{1}{\alpha} \cdot \ln \frac{1}{1 - \alpha}.
\]
In all cases the performance deteriorates as $\alpha$ approaches 1, which corresponds to the table getting full. The deterioration is obviously milder for double hashing than for linear probe sequences.

**Hash Functions.** Until now we have avoided the question of how to construct a hash function. A popular method interprets the keys as integers and subtracts multiples of the table size,

$$h(x) = x - \left\lfloor \frac{x}{m} \right\rfloor \cdot m = x \pmod{m}.$$ 

With this method, $m$ should be a prime number. For example, if $m = 2^k$ then $h(x)$ is the integer determined by the rightmost $i$ bits of $x$. The hash function thus depends on only part of $x$, which leads to non-uniform spreading unless the $i$ bits are uniformly spread.

In general, for every fixed hash function there are sets of keys for which the function spreads poorly. We can again use randomness to make this an unlikely event. Before explaining how this works we consider modulo functions for positive integers in more detail. By definition, $x \pmod{m}$ is the remainder of the integer division by $m$, $x = Q \cdot m + R$, where $x$, $m$, $Q$, $R$ are integers and the remainder $R = x \pmod{m}$ is in the interval $[0, m)$.

Let $a$ and $b$ be positive integers and consider the sequence $b$, $b+a$, $b+2a$, \ldots all modulo $m$. Since there are only finitely many remainders there must be a smallest integer $1 \leq i \leq m$ with

$$b \pmod{m} = b + i \cdot a \pmod{m}.$$ 

After reaching $i$ the remainders repeat. If $i < m$ then there is an integer $j \geq 2$ that divides both $m$ and $a$, as illustrated in Figure 40. If on the other hand $i = m$ then

![Image of a table with 10 boxes, numbered 0 to 9, illustrating the modulo operation.](image)

Figure 40: The size of the table is $m = 10$ and the step size is $a = 4$. Since 2 is a common factor of $m$ and $a$ the table is not exhausted.

$b, b+a, b+2a, \ldots$ exhaust all $m$ remainders. This happens iff $m$ and $p = a$ are relative prime. Another way to say the same thing is as follows.

**Modulo Claim.** Two positive integers $m$ and $p$ are relatively prime iff for each $0 \leq q < m$ there is a unique $0 \leq i < m$ such that $i \cdot p \pmod{m} = q$.

**Universal Hashing.** We now return to using randomness to avoid poor spreading. Let $H$ be a finite collection of functions $U \to \{0,1,\ldots, m-1\}$. $H$ is universal if for every $x \neq y$ in the universe $U$ the number of functions $h \in H$ with $h(x) = h(y)$ is at most $\frac{\text{card } H}{m}$. The idea is to choose $h$ randomly from $H$ at run-time. For two keys $x \neq y$ the probability that they collide is therefore at most $\frac{1}{m}$ by definition. The simplifying assumptions that lead to the performance of open addressing methods mentioned above are exactly that for each pair of non-equal keys the probability of a collision is $\frac{1}{m}$. In other words, the performance estimates will be accurate if we use universal hashing.

To construct a universal collection of hash functions we choose $m$ a prime number. Decompose a key $x$ into components: $x = \langle x_0, x_1, \ldots, x_r \rangle$, with $0 \leq x_i < m$ for all $i$. Randomly choose a sequence $a = \langle a_0, a_1, \ldots, a_r \rangle$ with $0 \leq a_i < m$ for all $i$. The hash function defined by $a$ is

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod{m},$$

and $H$ is the class of hash functions defined by all possible $(r+1)$-tuples $a$. Note that $\text{card } H = m^{r+1}$. We prove that the set of hash functions thus constructed is indeed universal.

**Universality Theorem.** $H$ is universal.

**Proof.** Consider two keys $x \neq y$ in the universe and assume $x_0 \neq y_0$, where

$$x = \langle x_0, x_1, \ldots, x_r \rangle,$$

$$y = \langle y_0, y_1, \ldots, y_r \rangle.$$ 

For any fixed values of $a_0$ through $a_r$ we have the following condition for $a_0$ to cause a collision:

$$a_0(x_0 - y_0) = \sum_{i=1}^{r} a_i(y_i - x_i),$$

where both sides are taken modulo $m$. In this equation we let $a_0$ vary and consider everything else fixed. Since $m$ is prime we have $m$ and $x_0 - y_0$ relative prime. Set $p = x_0 - y_0$ and $q = \sum_{i=1}^{r} a_i(y_i - x_i)$. The Modulo Claim implies that there is a unique $a_0$ that satisfies the equation. Thus $x$ and $y$ collide for exactly $m^r$ sequences $a$. Since $\text{card } H = m^{r+1}$ we have

$$\text{Prob}[h(x) = h(y)] = \frac{1}{m}.$$
By definition the probability is the number of hash functions that cause \( x \) and \( y \) to collide divided by the total number of hash functions. In other words, the number of hash functions that cause \( x \) and \( y \) to collide is \( \frac{\text{card } H}{m} \) as required.

**Example.** Consider \( m = 3 \) and \( r = 1 \). The collection of hash functions is
\[
H = \{ h_{ij} | 0 \leq i, j \leq 2 \}.
\]

Table 2 shows the slot assignments for all possible keys and under all hash functions. Observe that indeed every pair of keys collides for exactly \( \frac{\text{card } H}{m} = \frac{1}{3} \) hash functions. Observe also that every pair has a collision for the function defined by \( a = (0, 0, \ldots, 0) \). If we remove \( h_{00} \) from \( H \), the probability of a collision drops to
\[
\frac{m^r - 1}{m^{r+1} - 1} < \frac{m^r}{m^{r+1}}.
\]
In the example the probability decreases from \( \frac{1}{3} \) to \( \frac{1}{4} \). The only reason for keeping \( h_{00} \) in \( H \) would be to test out the implementation of the collision detection algorithm.