Hashing

Hash Tables, Hash Functions, Linear Probing, Quadratic Probing, Double Hashing

What is Hashing & Why?

1. Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.

2. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list $O(n)$ time in the worst case (almost pathological) – in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

3. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

4. When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key.
Components for Storage and Retrieval with Hash Tables

1. A hash table. This is a fixed size table that stores data of a given type.
2. A hash function: This is a function that converts a piece of data into an integer. Sometimes we call this integer a hash value. The integer should be at least as big as the hash table. When we store a value in a hash table, we compute its hash value with the hash function, take that value modulo the hash table size, and that’s where we store/retrieve the data.
3. A collision resolution strategy: There are times when two pieces of data have hash values that, when taken modulo the hash table size, yield the same value. That is called a collision. We need to handle collisions. There are in general four collision resolution strategies: Separate chaining, linear probing, quadratic probing, and double hashing.

Hash Tables

Assume that, universe of the keys is very large, say the set of all possible integers (−MAX, …, MAX). We can have a huge array, initialized with some kind of a sentinel (to indicate empty) and then we can use any key as the array index – insert, delete, find is constant time – sometimes known as Direct Addressing.

Disadvantages: (a) the set K of keys actually stored may be so small relative to U (universe of all possible keys) that most of the space allocated for T would be wasted; (b) storage inefficiency; (c) problematic for storing duplicate keys.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table. With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot h(k); that is, we use a hash function h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a hash table T[0 … m-1]:

\[ h: U \rightarrow \{0, 1, 2, \ldots, m-1\} \]

where the size m of the hash table is typically much less than |U| (= size of all possible keys). We say that an element with key k hashes to slot h(k); we also say that h(k) is the hash value of key k.

The hash function reduces the range of array indices and hence the size of the array. Instead of a size of |U|, the array can have size m.
Direct-address tables

Direct addressing is a simple technique that works well when the universe \( U \) of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe \( U = \{0, 1, \ldots, m - 1\} \) where \( m \) is not too large. We assume that no two elements have the same key.

To represent the dynamic set, we use an array, or direct-address table, denoted by \( T[0\ldots m - 1] \), in which each position, or slot, corresponds to a key in the universe \( U \). Slot \( k \) points to an element in the set with key \( k \). If the set contains no element with key \( k \), then \( T[k] = \text{NIL} \).

The dictionary operations are trivial to implement:

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

- **DIRECT-ADDRESS-INSERT** \((T, x)\)
  
  \( T\{x, \text{key}\} = x \)

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

Each of these operations takes only \( O(1) \) time [obvious].

Collisions in Hashing

There is a problem when two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.

One idea is to make \( h \) appear to be “random,” thus at least minimizing collisions. The very term “to hash,” evoking images of random mixing and chopping, captures the spirit of this approach. Of course, a hash function \( h \) must be deterministic in that a given input \( k \) should always produce the same output \( h(k) \). But, avoiding collisions altogether is therefore impossible. We still need a method for resolving the collisions that do occur. One simple method is called **Chaining**.
Hashing with Chaining

In chaining, we place all the elements that hash to the same slot into the same linked list. Slot $j$ contains a pointer to the head of the list of all stored elements that hash to $j$; if there are no such elements, slot $j$ contains NIL.

Insertion, deletion, search are extremely easy. Interesting question is how long does it take to search for a key? **Worst case** answer is the length of the longest list – obviously.

**How probable is the worst case?** The average-case performance of hashing depends on how well the hash function $h$ distributes the set of keys to be stored among the $m$ slots, on the average.

Given a hash table $T$ with $m$ slots that stores $n$ elements, we define the **load factor $\alpha$** for $T$ as $\alpha = n/m$, that is, the average number of elements stored in a chain. Assuming **simple uniform hashing** (a key is equally likely to be hashed into any slot) For $j = 0, 1, \ldots, m-1$, if $n_j$, $T[j]$ denotes the length of $T[j]$, then $n = n_0 + n_1 + \ldots + n_{m-1}$ or the expected value of $n_j$ is $\alpha = n/m$. 

Hashing with Chaining (Pictorially)
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, independently of where any other key has hashed to. We typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function $h(k) = \lfloor km \rfloor$ satisfies the condition of simple uniform hashing.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.

For example, the “division method” computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.

We will consider two simple hash functions here. The best way to design a good hash function is “uniform hashing”; we may not discuss those in this course.

Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we need a way to interpret them as natural numbers.

**Interpreting keys as natural numbers**

Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we need a way to interpret them as natural numbers.

For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the character string `pt` as the pair of decimal integers (112, 116) in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \times 128) + 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In our subsequent discussions, we assume that the keys are natural numbers.

Simple Modulo Arithmetic and Integer GCD and LCM (we have seen in High School and again in 1010; refresh your memory; let us see a few examples; you should be able to write simple programs to compute those functions.

128 is the total number of the standard ASCII symbols.
**Hash Tables without Linked Lists**

- Separate chaining hashing becomes a bit slow because of the time required to allocate new cells.
- An alternative to resolving collisions with linked lists is to try alternative cells until an empty cell is found, sometimes known as open addressing. In open addressing, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL.
- More formally, cells \( h_0(x) \), \( h_1(x) \), \( h_2(x) \), . . . are tried in succession, where \( h_i(x) = (\text{hash}(x) + f(i)) \mod \text{TableSize} \), with \( f(0) = 0 \). The function, \( f \), is the collision resolution strategy.
- Because all the data go inside the table, a bigger table is needed in such a scheme than for separate chaining hashing. Generally, the load factor \( \alpha \) should be below 0.5 for a hash table that doesn’t use separate chaining. We call such tables probing hash tables.
- There are three common collision resolution strategies: **Linear Probing**, **Quadratic Probing**, and **Double Hashing**.

**Insertion & Search in Open Addressing**

- To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order \( 0,1,2,\ldots,m-1 \), the sequence of positions probed depends upon the key being inserted.
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

  \[
  h: U \times \{0,1,2,\ldots,m-1\} \rightarrow \{0,1,2,\ldots,m-1\}
  \]

- With open addressing, 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 \([0, 1, 2, \ldots, m-1]\), so that every hash-table position is eventually considered as a slot for a new key as the table fills up.
- In the pseudocode, we assume each slot contains either a key or NIL (if the slot is empty). We assume that keys are not deleted from the hash table.
**Pseudocodes**

**HASH-INSERT** (T, k)
1  \( i = 0 \)
2  \textbf{repeat} \begin{align*}
3   j &= h(k, i) \\
4   \textbf{if} T[j] = \text{NIL} \\
5   &T[j] = k \\
6   \textbf{return} j \\
7   \textbf{else} i = i + 1 \\
8   \textbf{until} i = m \\
9   \textbf{error} \, \text{“hash table overflow”}\end{align*}

**HASH-SEARCH** (T, k)
1  \( i = 0 \)
2  \textbf{repeat} \begin{align*}
3   j &= h(k, i) \\
4   \textbf{if} T[j] = k \\
5   \textbf{return} j \\
6   i = i + 1 \\
7   \textbf{until} T[j] = \text{NIL} \, \text{or} \, i = m \\
8   \textbf{return} \text{NIL}\end{align*}

---

**Linear Probing**

Given an ordinary hash function \( h1: U \rightarrow \{0, 1, \ldots, m - 1\} \), called an auxiliary hash function, the method of linear probing uses the hash function \( h(k, i) = (h1(k) + 1) \mod m \), for \( i = 0, 1, \ldots, m - 1 \).

Given key \( k \), we first probe \( T[h1(k)] \), i.e., the slot given by the auxiliary hash function. We next probe slot \( T[(h1(k)+1) \mod m] \), and so on up to slot \( T[m - 1] \). Then we wrap around to slots \( T[0], T[1], \ldots \) until we finally probe slot \( T[h1(k) - 1] \). Because the initial probe determines the entire probe sequence, there are only \( m \) distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as primary clustering. 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 \( \frac{i+1}{m} \). Long runs of occupied slots tend to get longer, and the average search time increases.
**Quadratic probing**

Quadratic probing uses a hash function of the form

\[ h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \mod m \]

Where \( h_1 \) is an auxiliary hash function, \( c_1 \) and \( c_2 \) are positive auxiliary constants, and \( i = 0, 1, \ldots, m - 1 \). The initial position probed is \( T[h_1(k)] \); later positions probed are offset by amounts that depend in a quadratic manner on the probe number \( i \).

This method works much better than linear probing, but to make full use of the hash table, the values of \( c_1 \), \( c_2 \), and \( m \) are constrained. There are ways to select these parameters (outside our scope here).

Also, if two keys have the same initial probe position, then their probe sequences are the same, since \( h(k_1,0) = h(k_2,0) \) implies \( h(k_1,i) = h(k_2,i) \). This property leads to a milder form of clustering, called secondary clustering.

As in linear probing, the initial probe determines the entire sequence, and so only \( m \) distinct probe sequences are used.

**Double Hashing**

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form

\[ h(k,i) = (h_1(k) + i.h_2(k)) \mod m \]

where both \( h_1 \) and \( h_2 \) are auxiliary hash functions. The initial probe goes to position \( T[h_1(k)] \); successive probe positions are offset from previous positions by the amount \( h_2(k) \), modulo \( m \). Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key \( k \), since the initial probe position, the offset, or both, may vary.

The value \( h_2(k) \) must be relatively prime to the hash-table size \( m \) for the entire hash table to be searched. A convenient way to ensure this condition is to let \( m \) be a power of 2 and to design \( h_2 \) so that it always produces an odd number. Another way is to let \( m \) be prime and to design \( h_2 \) so that it always returns a positive integer less than \( m \).
The Division method

In the division method for creating hash functions, we map a key $k$ into one of $m$ slots by taking the remainder of $k$ divided by $m$. That is, the hash function is $h(k) = k \mod m$. For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$; $k = 543 \Rightarrow h(k) = 3$; $k = 148 \Rightarrow h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast. Did you see the collision?

- Efficiency of the division method depends on appropriate choice of $m$.
  - Avoid certain values of $m$. For example, $m$ should not be a power of 2, since if $m = 2^p$, then $h(k)$ is always the $p$ lowest-order bits of $k$. Such values of $m$ cannot in general distribute the keys appropriately. Unless we know that all low-order $p$-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key. Can you evaluate if $m = 2^p - 1$ is a good choice? Why or why not?
  - A prime not too close to an exact power of 2 is often a good choice for $m$.

The Multiplication method

The multiplication method for creating hash functions operates in two steps. First, we multiply the key $k$ by a constant $A$ in the range $0 < A < 1$ and extract the fractional part of $kA$. Then, we multiply this value by $m$ and take the floor of the result. In short, the hash function is $h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$, where “$kA \mod 1$” means the fractional part of $kA$, that is, $kA - \lfloor kA \rfloor$.

An advantage of the multiplication method is that the value of $m$ is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer $p$), since we can then easily implement the function on most computers.

Suppose, the word size of the machine is $w$ bits and that $k$ fits into a single word. We restrict $A$ to be a fraction of the form $s/2^w$, where $s$ is an integer in the range $0 < s < 2^w$.

- We first multiply $k$ by the $w$-bit integer $s = A \cdot 2^w$. The result is a $2w$-bit value $r_12^w + r_0$, where $r_1$ is the high-order word of the product and $r_0$ is the low-order word of the product. The desired $p$-bit hash value consists of the $p$ most significant bits of $r_0$.
- While this method works for any value of $A$, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed.
- $A \approx (\sqrt{5} - 1)/2$ works very well.
Cuckoo Hashing

Cuckoo hashing is a simple hash table where
- Lookups are worst-case O(1)
- Deletions are worst-case O(1)
- Insertions are amortized, expected O(1)
- Insertions are amortized O(1) with reasonably high probability.

1. Maintain two tables, each of which has m elements.
2. We choose two hash functions \( h_1 \) and \( h_2 \) from \( U \) (universe) to \([m]\) (array of m elements).
3. Every element \( x \in U \) will either be at position \( h_1(x) \) in the first table or \( h_2(x) \) in the second.
4. Lookups take time O(1) because only two locations must be checked.
5. Deletions take time O(1) because only two locations must be checked.

To insert an element \( x \), start by inserting it into table 1.
- If \( h_1(x) \) is empty, place \( x \) there.
- Otherwise, place \( x \) there, evict the old element \( y \), and try placing \( y \) into table 2.
- Repeat this process, bouncing between tables, until all elements stabilize.

Insertions run into trouble if we run into a cycle.
- If that happens, perform a rehash by choosing a new \( h_1 \) and \( h_2 \) and inserting all elements back into the tables.
- Multiple rehashes might be necessary before this succeeds.
- It's possible for a successful insertion to revisit the same slot twice.
- Cycles only arise if we revisit the same slot with the same element to insert.
Cuckoo hashing can be tricky to analyze for a few reasons:

- Elements move around and can be in one of two different places.
- The sequence of displacements can jump chaotically over the table.
- It turns out there's a beautiful framework for analyzing cuckoo hashing.