ICS 311 #6: Hash Tables

Outline

Readings and Screencasts

- CLRS Sections 11.1-11.4. (Exclude 11.5)
- Screencasts 6A, 6B, 6C, 6D (also in Laulima and iTunesU)

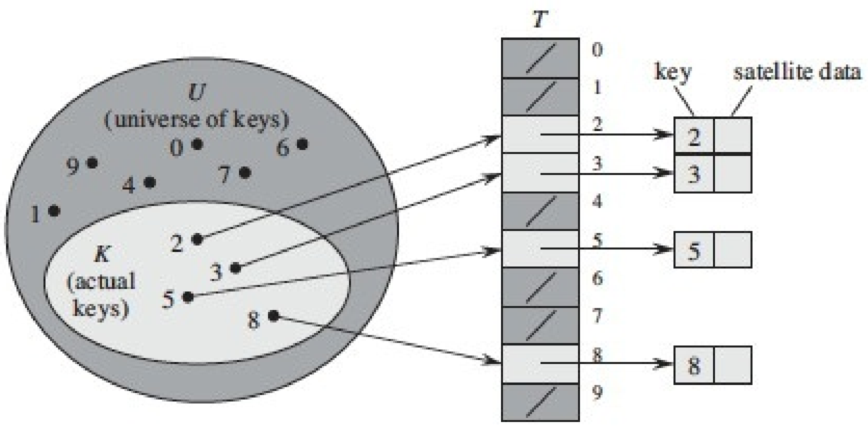
Motivations and Introduction

Many applications only need the insert, search and delete operations of a dynamic set. Example: symbol table in a compiler.

Hash tables are an effective approach. Under reasonable assumptions, they have O(1) operations, but they can be Θ(n) worst case

Direct Addressing

Hash tables generalize arrays. Let's look at the idea with arrays first. Given a key *k* from a universe *U* of possible keys, a **direct address table** stores and retrieves the element in position *k* of the array.



Direct addressing is applicable when we can allocate an array with one element for every key (i.e., of size |*U*|). It is trivial to implement:

```
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]] = NIL
```

However, often the space of possible keys is much larger than the number of actual keys we expect, so it would be wasteful of space (and sometimes not possible) to allocate an array of size $|U|$.

Hash Tables and Functions

Hash tables are also arrays, but typically of size proportional to the number of keys expected to be stored (rather than to the number of keys).

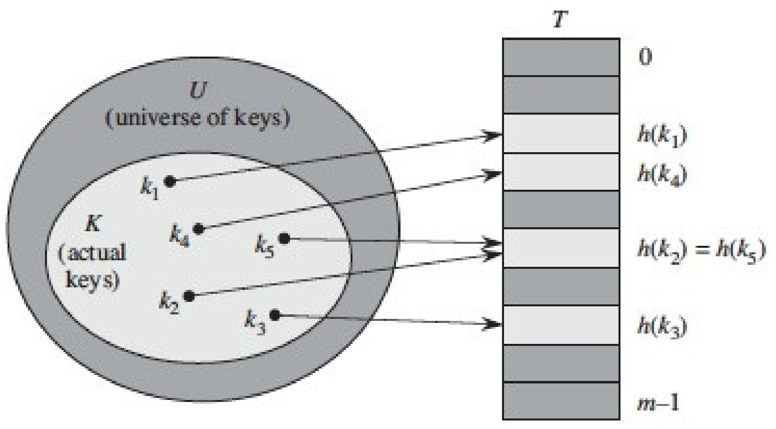
If the expected keys $K \subset U$, the Universe of keys, and $|K|$ is substantially smaller than $|U|$, then hash tables can reduce storage requirements to $\Theta(|K|)$.

A **hash function** $h(k)$ maps the larger universe U of external keys to indices into the array. Given a table of size m with zero-based indexing (we shall see why this is useful):

- $h : U \rightarrow \{0, 1, ..., m\text{-}1\}$.
- We say that k **hashes** to slot $h(k)$.

Collisions

The major issue to deal with in designing and implementing hash tables is what to do when the hash function maps multiple keys to the same table entry.



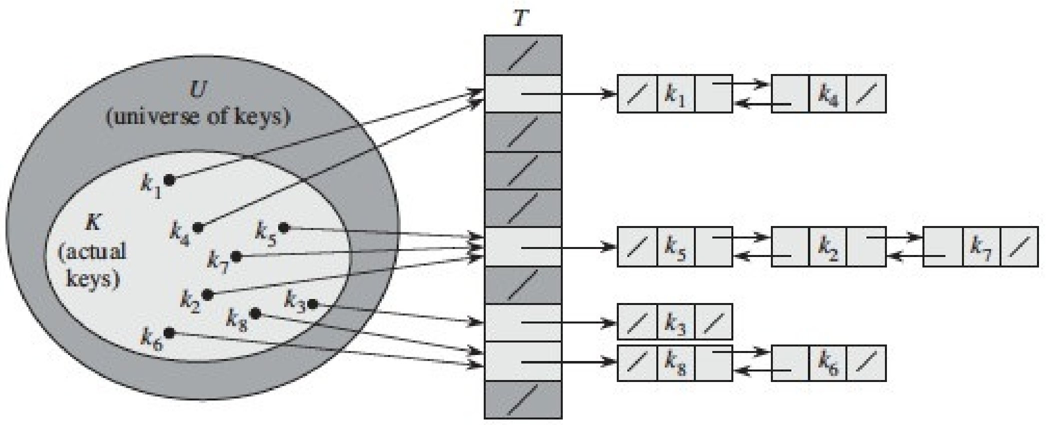
Collisions may or may not happen when $|K| \leq m$, but definitely happens when $|K| > m$. *(Is there any way to avoid this?)*

There are two major approaches: Chaining (the preferred method) and Open Addressing. We'll look at these and also hash function design.

Hash Tables with Chaining

A simple resolution: Put elements that hash to the same slot into a linked list. This is called *chaining* because we chain elements off the slot of the hash table.

- Slot j points to the head of a list of all stored elements that hash to j, or to NIL if there are no such elements.
- Doubly linked lists may be used when deletions are expected to be frequent.
- Sentinels can also be used to simplify the code.



Pseudocode for Chaining

Implementation is simple if you already have implemented linked lists:

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

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

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

What are the running times for these algorithms? Which can we state directly, and what do we need to know to determine the others?

Analysis of Hashing with Chaining

How long does it take to find an element with a given key, or to determine that there is no such element?

- Analysis is in terms of the **load factor $\alpha = n/m$**, where
  - n = number of elements in the table
  - m = number of slots in the table = number of (possibly empty) linked lists
- The load factor α is the average number of elements per linked list.
- Can have $\alpha < 1$; $\alpha = 1$; or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot.
  Why? What happens? $\Theta($______$?)$
- Average case depends on how well the hash function distributes the keys among the slots.

Let's analyze averge-case performance under the assumption of **simple uniform hashing:** any given element is equally likely to hash into any of the m slots:

- For $j = 0, 1, \ldots, m-1$, denote the length of list T[j] by n_j.
- Then $n = n_0 + n_1 + \ldots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assuming $h(k)$ computed in O(1), so time to search for k depends on length $n_{h(k)}$ of the list T[$h(k)$].

Consider two cases: Unsuccessful and Successful search. The former analysis is simpler because you always search to the end, but for successful search it depends on where in T[$h(k)$] the element with key k will be found.

Unsuccessful Search

Simple uniform hashing means that any key not in the table is equally likely to hash to any of the m slots.

We need to search to end of the list T[$h(k)$]. It has expected length $E[n_{h(k)}] = \alpha = n/m$.

Adding the time to compute the hash function gives $\mathbf{\Theta(1 + \alpha)}$. (We leave in the "1" term for the initial computation of h since α can be 0, and we don't want to say that the computation takes $\Theta(0)$ time).

Successful Search

We assume that the element x being searched for is equally likely to be any of the n elements stored in the table.

The number of elements examined during a successful search for x is 1 more than the number of elements that appear before x in x's list (because we have to search them, and then examine x).

These are the elements inserted *after* x was inserted (because we insert at the head of the list).

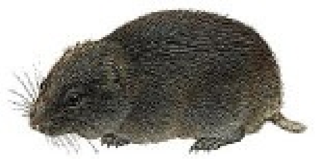
Need to find on average, over the n elements x in the table, how many elements were inserted into x's list after x was inserted. *Lucky we just studied indicator random variables!*

For $i = 1, 2, \ldots, n$, let x_i be the ith element inserted into the table, and let $k_i = key[x_i]$.

For all i and j, define the indicator random variable:

$X_{ij} = I\{h(k_i) = h(k_j)\}$. *(The event that keys k_i and k_j hash to the same slot.)*

Simple uniform hashing implies that $\Pr\{h(k_i) = h(k_j)\} = 1/m$ *(Why?)*



Therefore, $E[X_{ij}] = 1/m$ by Lemma 1 (Topic #5).

The expected number of elements examined in a successful search is those elements j that are inserted after the element i of interest *and* that end up in the same linked list (X_{ij}):

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

- The innermost summation is adding up, for all j inserted after i ($j=i+1$), those that are in the same hash table (when $X_{ij} = 1$).
- The outermost summation runs this over all n of the keys inserted (indexed by i), and finds the average by dividing by n.

I fill in some of the implicit steps in the rest of the text's analysis. First, by linearity of expectation we can move the E in:

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]\right)$$

That is the crucial move: instead of analyzing the probability of complex events, use indicator random variables to break them down into simple events that we know the probabilities for. In this case we know E[$X_{i,j}$] (if *you* don't know, ask the lemming above):

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \frac{1}{m}\right)$$

Multiplying 1/*n* by the terms inside the summation,

- For the first term, we get $\Sigma_{i=1,n}1/n$, which is just *n*/*n* or 1
- Move 1/*m* outside the summation of the second term to get 1/*nm*. This leaves $\Sigma_{i=1,n}(\Sigma_{j=i+1,n}1)$, which simplifies as shown below (if you added 1 *n* times, you would overshoot by *i*).

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

Splitting the two terms being summed, the first is clearly n^2, and the second is the familiar sum of the first *n* numbers:

$$= 1+\frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right)$$
$$= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

Distributing the 1/*nm*, we get $1 + (n^2/nm - n(n+1)/2nm = 1 + n/m - (n+1)/2m = 1 + 2n/2m - (n+1)/2m$, and now we can combine the two fractions:

$$= 1+\frac{n-1}{2m}$$

Now we can turn two instances of *n*/*m* into α with this preparation: $1 + (n - 1)/2m = 1 + n/2m - 1/2m = 1 + \alpha/2 - \mathrm{n}/2mn =$

$$= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.$$

Adding the time (1) for computing the hash function, the expected total time for a successful search is:

$\Theta(2 + \alpha/2 - \alpha/2n) =$ **$\Theta(1 + \alpha)$.**

since the third term vanishes in significance as *n* grows, and the constants 2 and 1/2 have Θ(1) growth rate.

Thus, **search is an average of $\Theta(1 + \alpha)$ in either case.**

If the number of elements stored n is bounded within a constant factor of the number of slots m, i.e., $n = O(m)$, then α is a constant, and search is O(1) on average.

Since insertion takes O(1) worst case and deletion takes O(1) worst case when doubly linked lists are used, all three operations for hash tables are O(1) on average.

(I went through that analysis in detail to show again the utility of indicator random variables and to demonstrate what is possibly the most crucial fact of this chapter, but we won't do the other analyses in detail. With perserverence you can similarly unpack the other analyses.)

Hash Functions and Universal Hashing

Ideally a hash function satisfies the assumptions of simple uniform hashing.

This is not possible in practice, since we don't know in advance the probability distribution of the keys, and they may not be drawn independently.

Instead, we use heuristics based on what we know about the domain of the keys to create a hash function that performs well.

Keys as natural numbers

Hash functions assume that the keys are natural numbers. When they are not, a conversion is needed. Some options:

- Floating point numbers: If an integer is required, sum the mantissa and exponent, treating them as integers.
- Character string: Sum the ASCII or Unicode values of the characters of the string.
- Character string: Interpret the string as an integer expressed in some radix notation. (This gives very large integers.)

Division method

A common hash function: **$h(k) = k \bmod m$.**
(Why does this potentially produce all legal values, and only legal values?)

Advantage: Fast, since just one division operation required.

Disadvantage: Need to avoid certain values of m, for example:

- Powers of 2. If $m = 2^p$ for integer p then $h(k)$ is the least significant p bits of k.
  (There may be a domain pattern that makes the keys clump together).
- If character strings are interpreted in radix 2^p then $m = 2^p - 1$ is a bad choice: permutations of characters hash the same.

A prime number not too close to an exact power of 2 is a good choice for m.

Multiplication method

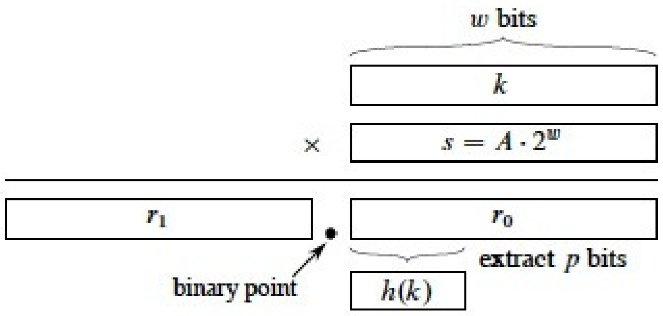
$h(k) = \text{Floor}(m(k\,A \bmod 1))$, where $k\,A \bmod 1$ = fractional part of kA.

1. Choose a constant A in range 0 < A < 1.
2. Multiply *k* by A
3. Extract the fractional part of *k*A
4. Multiply the fractional part by *m*
5. Take the floor of the result.

Disadvantage: Slower than division.

Advantage: The value of *m* is not critical.

The book discusses an implementation that we won't get into ...



Universal Hashing

Our malicious adversary is back! He's choosing keys that all hash to the same slot, giving worst case behavior and gumming up our servers! What to do?

Random algorithms to the rescue: randomly choose a different hash function each time you construct and use a new hash table.

But it has to be a good one. Can we define a family of good candidates?

Consider a finite collection ***H*** of hash functions that map universe U of keys into {0, 1, ..., *m*-1}.

H is **universal** if for each pair of keys *k, l* ∈ U, where $k \neq l$, the number of hash functions $h \in H$ for which *h(k)* = *h(l)* is less than or equal to |***H***|/*m* (that's the size of ***H*** divided by *m*).

In other words, with a hash function *h* chosen randomly from ***H***, the probability of collision between two different keys is no more than *1/m*, the chance of a collision when choosing two slots randomly and independently.

Universal hash functions are good because (proven as Theorem 11.3 in text):

- If *k* is not in the table, the expected length E[$n_{h(k)}$] of the list that *k* hashes to is less than or equal to α.
- If *k* is in the table, the expected length E[$n_{h(k)}$] of the list that holds *k* is less than or equal to 1 + α.

Therefore, the expected time for search is O(1).

One candidate for a collection ***H*** of hash functions is:

> ***H*** = {$h_{ab}(k)$: $\boldsymbol{h_{ab}(k) = ((ak + b) \bmod p) \bmod m)}$}, where $a \in \{1, 2, ..., p-1\}$ and $b \in \{0, 1, ..., p-1\}$, where *p* is prime and larger than the largest key.

Details in text, including proof that this provides a universal set of hash functions. Java built in hash functions take care of much of this for you: read the Java documentation for details.

Open Addressing Strategies

Open Addressing seeks to avoid the extra storage of linked lists by putting all the keys in the hash table itself.

Of course, we need a way to deal with collisions. If a slot is already occupied we will apply a systematic strategy for searching for alternative slots. This same strategy is used in both insertion and search.

Probes and $h(k,i)$

Examining a slot is called a **probe**. We need to extend the hash function h to take the probe number as a second argument, so that h can try something different on subsequent probes. We count probes from 0 to m-1 (you'll see why later), so the second argument takes on the same values as the result of the function:

$$h : U \times \{0, 1, \ldots m\text{-}1\} \rightarrow \{0, 1, \ldots m\text{-}1\}$$

We require that the **probe sequence**

$$\langle\, h(k,0),\ \ h(k,1)\ \ \ldots\ \ h(k,m\text{-}1)\,\rangle$$

be a permutation of ⟨ 0, 1, ... m-1 ⟩. Another way to state this requirement is that all the positions are visited.

There are three possible outcomes to a probe: k is in the slot probed (successful search); the slot contains NIL (unsuccessful search); or some other key is in the slot (need to continue search).

The strategy for this continuation is the crux of the problem, but first let's look at the general pseudocode.

Pseudocode

Insertion returns the index of the slot it put the element in k, or throws an error if the table is full:

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Search returns either the index of the slot containing element of key k, or NIL if the search is unsuccessful:

```
HASH-SEARCH($T, k$)
1  $i = 0$
2  repeat
3      $j = h(k, i)$
4      if $T[j] == k$
5          return $j$
6      $i = i + 1$
7  until $T[j] ==$ NIL or $i == m$
8  return NIL
```

Deletion is a bit complicated. We can't just write NIL into the slot we want to delete. *(Why?)*

Instead, we write a special value DELETED. During search, we treat it as if it were a non-matching key, but insertion treats it as empty and reuses the slot.

Problem: the search time is no longer dependent on α. *(Why?)*

The ideal is to have **uniform hashing**, where each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots m\text{-}1 \rangle$ as its probe sequence. But this is hard to implement: we try to guarantee that the probe sequence is *some* permutation of $\langle 0, 1, \ldots m\text{-}1 \rangle$.

We will define the hash functions in terms of **auxiliary hash functions** that do the initial mapping, and define the primary function in terms of its ith iterations, where $0 \le i < m$.

Linear Probing

Given an **auxiliary hash function *h'***, the probe sequence starts at $h'(k)$, and continues sequentially through the table:

$$h(k,i) = (h'(k) + i) \bmod m$$

Problem: **primary clustering**: sequences of keys with the same h' value build up long runs of occupied sequences.

Quadratic Probing

Quadratic probing is attempt to fix this ... instead of reprobing linearly, QP "jumps" around the table according to a quadratic function of the probe, for example:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

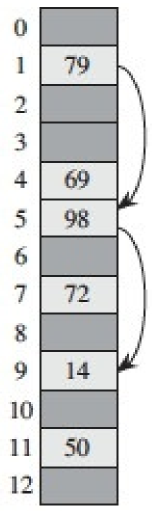
where c_1 and c_2 are constants.

Problem: **secondary clustering**: although primary clusters across sequential runs of table positions don't occur, two keys with the same h' may still have the same probe sequence, creating clusters that are broken across the same sequence of "jumps".

Double Hashing

A better approach: use two auxiliary hash functions h_1 and h_2, where h_1 gives the initial probe and h_2 gives the remaining probes (here you can see that having i=0 initially drops out the second hash until it is needed):

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$.



$h_2(k)$ must be relatively prime to m (relatively prime means they have no factors in common other than 1) to guarantee that the probe sequence is a full permutation of ⟨0, 1, ... m-1⟩. Two approaches:

- Choose m to be a power of 2 and h_2 to always produce an odd number > 1.
- Let m be prime and have $1 < h_2(k) < m$.
  (The example figure is $h_1(k) = k \bmod 13$, and $h_2(k) = 1 + (k \bmod 11)$.)

There are $\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence. This is an improvement over linear or quadratic hashing.

Analysis of Open Addressing

The textbook develops two theorems you will use to compute the expected number of probes for unsuccessful and successful search. (These theorems require $\alpha < 1$ because an expression $1/1-\alpha$ is derived and we don't want to divide by 0.)

> **Theorem 11.6:** Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an ***unsuccessful*** search is at most $\mathbf{1/(1 - \alpha)}$, assuming uniform hashing.
>
> **Theorem 11.8:** Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a ***successful*** search is at most $\mathbf{(1/\alpha) \ln (1/(1 - \alpha))}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

We leave the proofs for the textbook, but note particularly the "intuitive interpretation" in the proof of 11.6 of the ***expected number of probes*** on page 275:

$$E[X] = 1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

We always make the first probe (1). With probability $\alpha < 1$, the first probe finds an occupied slot, so we need to probe a second time (α). With probability α^2, the first two slots are occupied, so we need to make a third probe ...

Dan Suthers
Last modified: Sun Feb 16 02:14:59 HST 2014
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.