Chapter 11: Hash Tables

Chapter 11 overview

Many applications require a dynamic set that supports only the ***dictionary operations*** INSERT, SEARCH, and DELETE. Example: a symbol table in a compiler.

A hash table is effective for implementing a dictionary.

- The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions.
- Worst-case search time is $\Theta(n)$, however.

A hash table is a generalization of an ordinary array.

- With an ordinary array, store the element whose key is k in position k of the array.
- Given a key k, to find the element whose key is k, just look in the kth position of the array. This is called ***direct addressing***.
- Direct addressing is applicable when you can afford to allocate an array with one position for every possible key.

Use a hash table when do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key k, don't just use k as the index into the array.
- Instead, compute a function of k, and use that value to index into the array. We call this function a ***hash function***.

Issues that we'll explore in hash tables:

- How to compute hash functions. We'll look at several approaches.
- What to do when the hash function maps multiple keys to the same table entry. We'll look at chaining and open addressing.

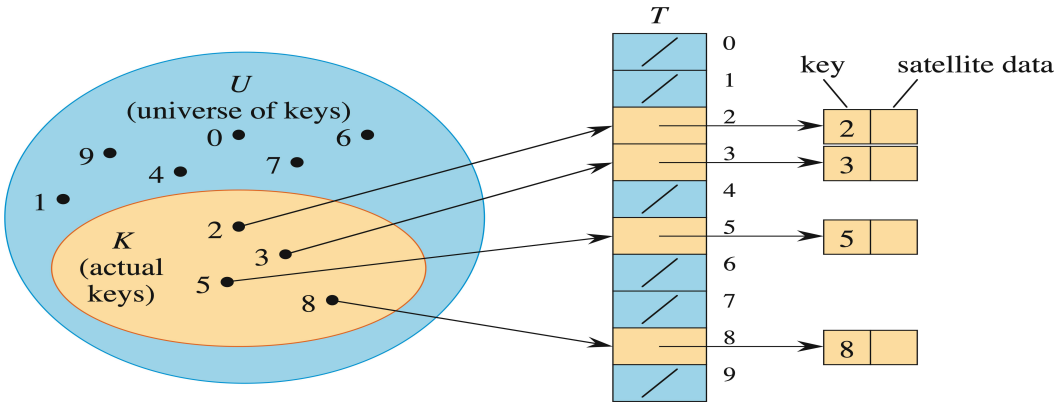
Direct-address tables

Scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a ***direct-address table***, or array, $T[0 \ldots m-1]$:

- Each ***slot***, or position, corresponds to a key in U.
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k]$ is empty, represented by NIL.



Dictionary operations are trivial and take $O(1)$ time each:

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

DIRECT-ADDRESS-INSERT(T, x)
  $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
  $T[x.key] =$ NIL

Hash tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$. Θ (N(K)) //number of elements in K
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

Idea

Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.

- We call h a ***hash function*** and T a ***hash table***.
- $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Collision

When two or more keys hash to the same slot.

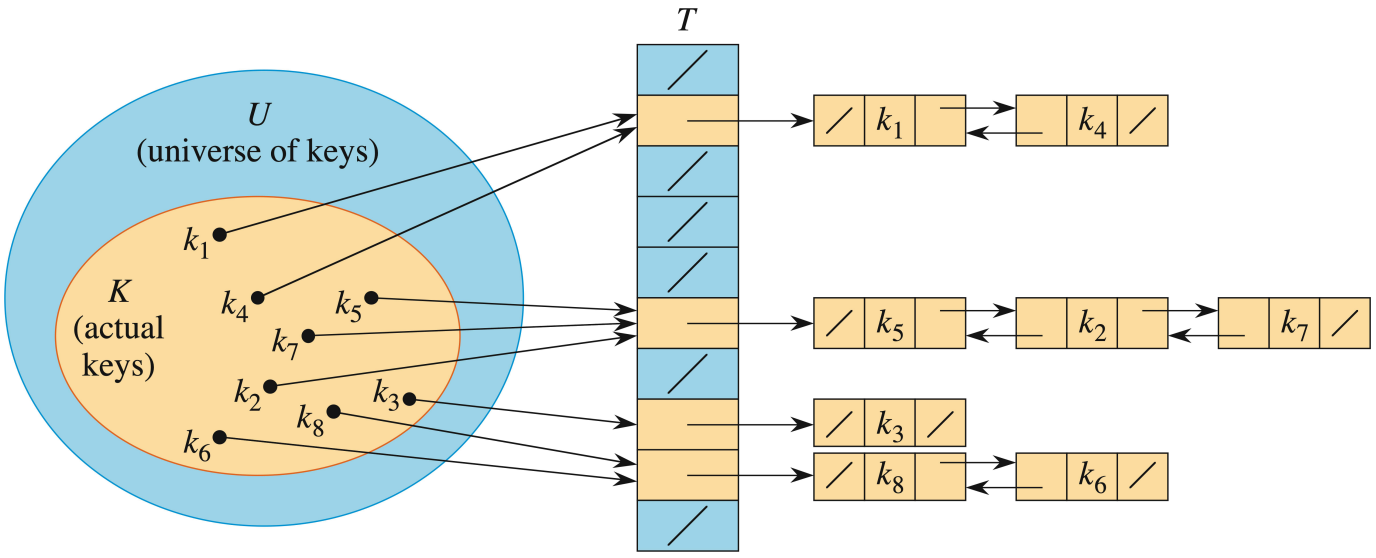
- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing. We'll examine both.

Independent uniform hashing

- Ideally, $h(k)$ would be randomly and independently chosen uniformly from $\{0, 1, \ldots, m-1\}$. Once $h(k)$ is chosen, each subsequent evalution of $h(k)$ must yield the same result.
- Such an idea hash function is an ***independent uniform hash function***, or ***random oracle***.
- It's an ideal theoretical abstraction. Cannot be reasonably implemented in practice. Use it to analyze hashing behavior, and find practical approximations to the ideal.

Collision resolution by chaining

Like a nonrecursive type of divide-and-conquer: use the hash function to divide the n elements randomly into m subsets, each with approximately $|n/m|$ elements. Manage each subset independently as a linked list.



[This figure shows singly linked lists. If needed to delete elements, it's better to use doubly linked lists.]

- Slot j contains a pointer to the head of the list of all stored elements that hash to j *[or to the sentinel if using a circular, doubly linked list with a sentinel]* ,
- If there are no such elements, slot j contains NIL.

How to implement dictionary operations with chaining:

- ***Insertion:***

  CHAINED-HASH-INSERT(T, x)
  LIST-PREPEND$(T[h(x.key)], x)$

  - Worst-case running time is $O(1)$.
  - Assumes that the element being inserted isn't already in the list.
  - It would take an additional search to check if it was already inserted.

- ***Search:***

  CHAINED-HASH-SEARCH(T, k)
  return LIST-SEARCH$(T[h(k)], k)$

  Worst-case running time is proportional to the length of the list of elements in slot $h(k)$.

- ***Deletion:***

  CHAINED-HASH-DELETE(T, x)
  LIST-DELETE$(T[h(x.key)], x)$

  - Given pointer x to the element to delete, so no search is needed to find this element.
  - Worst-case running time is $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because need to find x's predecessor in its list in order to correctly update *next* pointers.

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the ***load factor*** $\alpha = n/m$:
  - $n =$ # of elements in the table.
  - $m =$ # of slots in the table = # of (possibly empty) linked lists.
  - Load factor is 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 $\Rightarrow$ get a single list of length n $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

Focus on average-case performance of hashing with chaining.

- Assume ***independent uniform hashing***: any given element is equally likely to hash into any of the m slots, independent of where any other elements hash to.
- Independent uniform hashing is ***universal***: probability that any two distinct keys collide is $1/m$.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that the hash function takes $O(1)$ time to compute, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

We consider two cases:

- If the hash table contains no element with key k, then the search is unsuccessful.
- If the hash table does contain an element with key k, then the search is successful.

[In the theorem statements that follow, we omit the assumptions that we're resolving collisions by chaining and that independent uniform hashing applies. The theorems in the book spell out these assumptions.]

Unsuccessful search

Theorem

An unsuccessful search takes average-case time $\Theta(1 + \alpha)$.

Proof Independent uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the m slots.

To search unsuccessfully for any key k, need to search to the end of list $T[h(k)]$. This list has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α.

Adding in the time to compute the hash function, the total time required is $\Theta(1 + \alpha)$. ■

Successful search

- The average-case time for a successful search is also $\Theta(1 + \alpha)$.
- The circumstances are slightly different from an unsuccessful search.
- The probability that each list is searched is proportional to the number of elements it contains.

Theorem

A successful search takes expected time $\Theta(1 + \alpha)$.

Proof

Not covered.

Interpretation

If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that searching takes constant time on average.

Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations take $O(1)$ time on average.

Hash functions

We discuss hash-function properties and several ways to design hash functions.

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independent of any other key.
- In practice, it's not possible to satisfy this assumption, since you don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- If you know the distribution of keys, you can take advantage of it.

  Example: If keys are random real numbers independently and uniformly distributed in the half-open interval $[0, 1)$, then can use $h(k) = \lfloor km \rfloor$.
- We'll see "static hashing," which uses a single fixed hash function. And "random hashing," which chooses a hash function at random from a family of hash functions. With random hashing, don't need to know the probability distribution of the keys. Instead, the randomization is in the choice of hash function. We recommend random hashing.

Keys are integers, vectors, or strings

In practice, hash functions assume that the keys are either

- A short nonnegative integer that fits in a machine word (typically 32 or 64 bits), or
- A short vector of nonnegative integers, each of bounded size, e.g., a string of bytes.

For now, assume that keys are short nonnegative integers. We'll look at keys as vectors later.

Static hashing

A single, fixed hash function. Randomization comes only from the hoped-for distribution of the keys.

Division method

$h(k) = k \bmod m$.

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.
Advantage: Fast, since requires just one division operation.

Good choice for m: A prime not too close to an exact power of 2.

Multiplication method

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

Put another way, $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor =$ fractional part of kA.

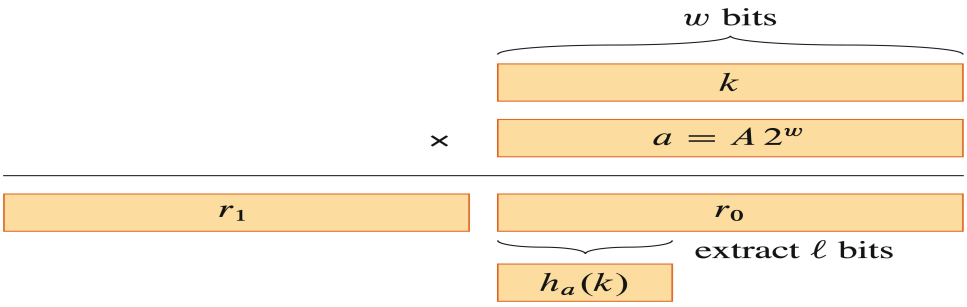
Disadvantage: Slower than division method.

Advantage: Value of m is not critical. Can choose it independently of A.

Multiply-shift method

A special case of the multiplication method.

- Let the word size of the machine be w bits.
- Set $m = 2^\ell$ for some integer $\ell \le w$.
- Assume that the key k fits into a single word. (k takes w bits.)
- Choose a fixed w-bit positive integer $a = A\,2^w$ in the range $0 < a < 2^w$.



- Multiply k by a.
- Multiplying two w-bit words $\Rightarrow$ the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product.
- Hash value is $h_a(k) = \ell$ most significant bits of r_0. Since $\ell \le w$, don't need the r_1 part of the product. Need only r_0.
- Define the operator $\ggg$ as logical right shift, so that $x \ggg b$ shifts x right by b bits, filling in the vacated positions on the left with zeros. Then $h_a(k) = (ka \bmod 2^w) \ggg (w - \ell)$. Here, $ka \bmod 2^w$ zeroes out r_1 (the high-order w bits of $ka = kA\,2^w$), and shifting right by $w - \ell$ bits moves the ℓ most significant bits of r_0 into the ℓ rightmost positions (same as dividing by $2^{w-\ell}$ and taking the floor of the result).
- Need only three machine instructions to compute $h_a(k)$: multiply, subtract, logical right shift.
- ***Example:*** $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, $w = 32$. Choose $a = 2654435759$. Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$ $\Rightarrow r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 give $h_a(k) = 67$.

Multiply-shift is fast, but doesn't guarantee good average-case performance. Can get good average-case performance by picking a as a randomly chosen odd integer.

Random hashing

Suppose that a malicious adversary, who gets to choose the keys to be hashed, has seen your hashing program and knows the hash function in advance. Then they could choose keys that all hash to the same slot, giving worst-case behavior. Any static hash function is vulnerable to this type of attack.

One way to defeat the adversary is to choose a hash function randomly independent of the keys. We describe a special case, ***universal hashing***, which can yield

provably good performance average when collisions are resolved by chaining, no matter the keys.

Consider a finite collection $\mathcal{H}$ of hash functions that map a universe U of keys into the range $\{0, 1, \ldots, m-1\}$. $\mathcal{H}$ is ***universal*** if for each pair of keys $k_1, k_2 \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k_1) = h(k_2)$ is $\leq |\mathcal{H}|/m$.
In other words, $\mathcal{H}$ is universal if, with a hash function h chosen randomly from $\mathcal{H}$, the probability of a collision between two different keys is no more than the $1/m$ chance of just choosing two slots randomly and independently.

Corollary to the previous theorem on average-case time for a successful search with chaining:

Corollary
Using chaining and universal hashing and starting with an initially empty table with m slots, the expected time is $\Theta(s)$ to handle any sequence of s INSERT, SEARCH, or DELETE operations with $n = O(m)$ INSERT operations.

Proof INSERT and DELETE take constant time (recall: DELETE has a pointer to the element to delete, so no search required as part of DELETE).

$n = O(m) \Rightarrow \alpha = O(1)$. Expected time for each SEARCH is $O(1)$, because the proof of the theorem depended only on the collision probabilities, which rely on the independent uniform hashing assumption. A universal hash function fulfills this assumption. Since each search has expected time $O(1)$, linearity of expectation gives that the expected time for any sequence of s operations is $O(s)$.

Each operation takes $\Omega(1)$ time $\Rightarrow$ entire sequence takes $\Omega(s)$ time $\Rightarrow \Theta(s)$. ■

Achievable properties of random hashing

Families of hash functions may exhibit any of several properties. Consider a family $\mathcal{H}$ of hash functions over domain U and with range $\{0, 1, \ldots, m-1\}$, keys in U, slot numbers in $\{0, 1, \ldots, m-1\}$, and a hash function h picked randomly from $\mathcal{H}$. The following properties may pertain to $\mathcal{H}$:

Uniform: The probability over picks of h that $h(k) = q$ is $1/m$.

Universal: For any distinct keys k_1, k_2, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.

ϵ-universal: For any distinct keys k_1, k_2 the probability that $h(k_1) = h(k_2)$ is at most ϵ (so that universal means $1/m$-universal).

***d*-independent:** For any distinct keys $k_1, k_2, \ldots, k_d$ and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, the probability that $h(k_i) = q_i$ is $1/m^d$.

Open addressing

An alternative to chaining for handling collisions.

Idea

- Store all elements in the hash table itself.
- Each slot contains either an element or NIL.
- The hash table can fill up, but the load factor can never be > 1.
- How to handle collisions during insertion:
  - Determine the element's "first-choice" location in the hash table.
  - If the first-choice location is unoccupied, put the element there.
  - Otherwise, determine the element's "second-choice" location. If unoccupied, put the element there.
  - Otherwise, try the "third-choice" location. And so on, until an unoccupied location is found.
  - Different elements have different preference orders.
- How to search:
  - Same idea as for insertion.
  - But upon finding an unoccupied slot in the hash table, conclude that the element being searched for is not present.
  - Open addressing avoids the pointers needed for chaining. You can use the extra space to make the hash table larger.

  More specifically, to search for key k:
  - Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a ***probe***.
  - If slot $h(k)$ contains key k, the search is successful. If this slot contains NIL, the search is unsuccessful.
  - If slot $h(k)$ contains a key that is not k, compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.).
  - Keep probing until either find key k (successful search) or find a slot holding NIL (unsuccessful search).
  - Need the sequence of slots probed to be a permutation of the slot numbers $\langle 0, 1, \ldots, m-1 \rangle$ (so that all slots are examined if necessary, and so that no slot is examined more than once).
  - Thus, the hash function is $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$.
  - The requirement that the sequence of slots be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ is equivalent to requiring that the ***probe sequence*** $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - To insert, act as though searching, and insert at the first NIL slot encountered.

Pseudocode for insertion

HASH-INSERT either returns the slot number where the new key k goes or flags an error because the table is full.

```
HASH-INSERT(T, k)
  i = 0
  repeat
      q = h(k, i)        // It is a number between 0 and m-1
      if T[q] == NIL
          T[q] = k
          return q
      else i = i + 1
  until i == m
  error "hash table overflow"
```

Pseudocode for searching

HASH-SEARCH returns either the slot number where the key k resides or NIL if key k is not in the table.

```
HASH-SEARCH(T, k)
  i = 0
  repeat
      q = h(k, i)
      if T[q] == k
          return q
      i = i + 1
  until T[q] == NIL or i == m
  return NIL
```

Deletion

Cannot just put NIL into the slot containing the key to be deleted.

- Suppose key k in slot q is inserted.
- And suppose that sometime after inserting key k, key k' was inserted into slot q', and during this insertion slot q (which contained key k) was probed.
- And suppose that key k was deleted by storing NIL into slot q.
- And then a search for key k' occurs.
- The search would probe slot q *before* probing slot q', which contains key k'.
- Thus, the search would be unsuccessful, even though key k' is in the table.

Solution: Use a special value DELETED instead of NIL when marking a slot as empty during deletion.

- Search should treat DELETED as though the slot holds a key that does not match the one being searched for.
- Insertion should treat DELETED as though the slot were empty, so that it can be reused.

The disadvantage of using DELETED is that now search time is no longer dependent on the load factor α.

A simple special case of open addressing, called linear probing, avoids having to mark slots with DELETED. *[Deletion with linear probing will be covered later in this chapter.]*

How to compute probe sequences

The ideal situation is ***independent uniform permutation hashing*** (also known as ***uniform hashing***): each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence. (This generalizes independent uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It's hard to implement true independent uniform permutation hashing. Instead, approximate it with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$. None of these techniques can produce all $m!$ probe sequences. Double hashing can generate at most m^2 probe sequences, and linear probing can generate only m. They will make use of ***auxiliary hash functions***, which map $U \to \{0, 1, \ldots, m-1\}$.

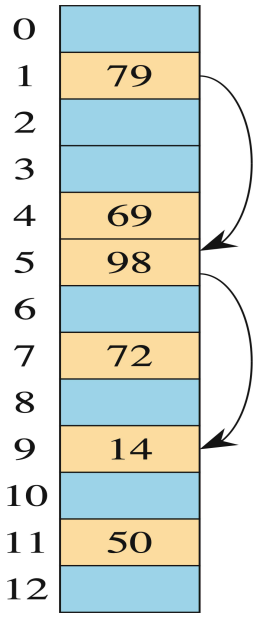
Double hashing

Uses auxiliary hash functions h_1, h_2 and probe number i:

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

The first probe goes to slot $h_1(k)$ (because i starts at 0). Successive probes are offset from previous slots by $h_2(k)$ modulo $m \Rightarrow$ the probe sequence depends on the key in two ways.

Example: $m = 13$, $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$, inserting key $k = 14$. First probe is to slot 1 ($14 \bmod 13 = 1$), which is occupied. Second probe is to slot 5 ($((14 \bmod 13)+(1+(14 \bmod 11))) \bmod 13 = (1+4) \bmod 13 = 5$), which is occupied. Third probe is to slot 9 (offset from slot 5 by 4), which is free, so key 14 goes there.



Example: Hash table size 13 with $h_1(k)= k \bmod 13$ and $h_2(k)= 1+(k \bmod 11)$
Consider k = 14. $h_1(14) = 1$ and $h_2(14) = 4$.
Therefore, we probe 1, 5, 9, …

Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

- Could choose m to be a power of 2 and h_2 to always produce an odd number.
- Could let m be prime and have $1 < h_2(k) < m$.

  Example: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$ (as in the example above, with $m = 13, m' = 11$).

Linear probing

Special case of double hashing.

Given auxiliary hash function h', the probe sequence starts at slot $h'(k)$ and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0.
Given key k and probe number i $(0 \le i < m)$, $h(k,i) = (h'(k) + i) \bmod m$.

The initial probe determines the entire sequence $\Rightarrow$ only m possible sequences.*[Will revisit linear probing later in the chapter.]*

Analysis of open-address hashing

Assumptions

- Analysis is in terms of load factor α. Assume that the table never completely fills, so always have $0 \leq n < m \Rightarrow 0 \leq \alpha < 1$.
- Assume independent uniform permutation hashing.
- No deletion.
- In a successful search, each key is equally likely to be searched for.

Theorem

The expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Intuition behind the proof: The first probe always occurs. The first probe finds an occupied slot with probability (approximately) α, so that a second probe happens. With probability (approximately) α^2, the first two slots are occupied, so that a third
probe occurs. Get the geometric series $1 + \alpha + \alpha^2 + \alpha^3 + \cdots = 1/(1-\alpha)$
(since $\alpha < 1$).

Proof: Not covered

Interpretation

If α is constant, an unsuccessful search takes $O(1)$ time.

- If $\alpha = 0.5$, then an unsuccessful search takes an average of $1/(1-0.5) = 2$ probes.
- If $\alpha = 0.9$, takes an average of $1/(1-0.9) = 10$ probes.

Corollary

The expected number of probes to insert is at most $1/(1-\alpha)$.

Proof Since there is no deletion, insertion uses the same probe sequence as an unsuccessful search. ■

Theorem

The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Proof: Not covered