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

We use a hash table when we 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 the multiplication and division methods.
- What to do when the hash function maps multiple keys to the same table entry. We'll look at chaining and open addressing.

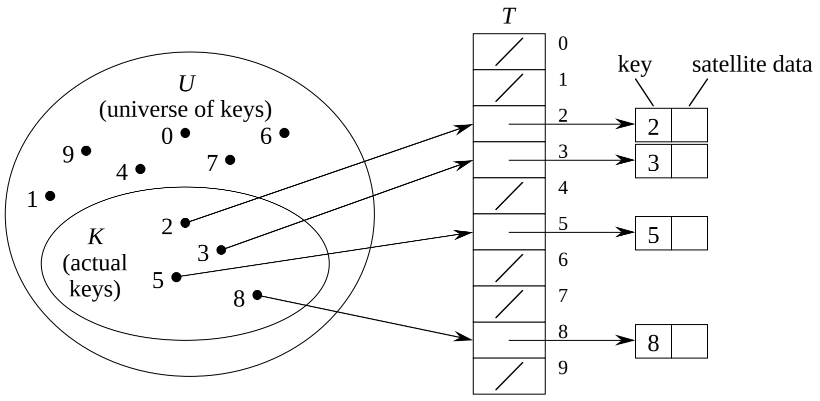
Direct-address tables

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[key[x]] = x

DIRECT-ADDRESS-DELETE(T, x)
  T[key[x]] = 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 $\Theta(|K|)$.
- 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***.
- $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

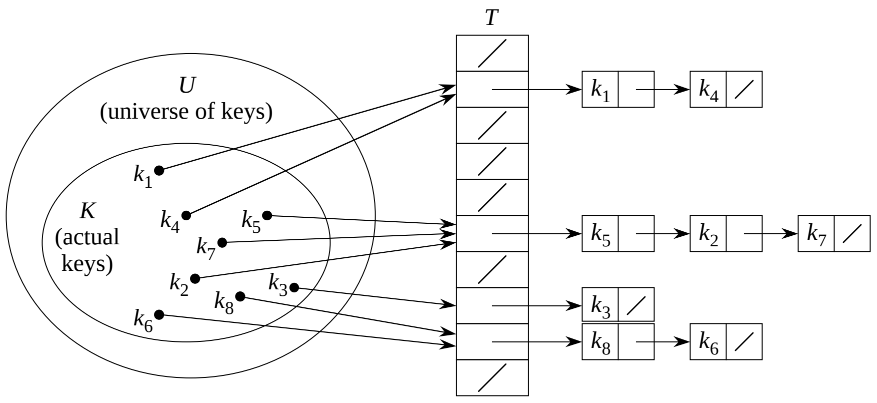
Collisions

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 $|K| \le m$, 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.
- Chaining is usually better than open addressing. We'll examine both.

Collision resolution by chaining

Put all elements that hash to the same slot into a linked list.



[This figure shows singly linked lists. If we want 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$(T, x)$
    insert $x$ at the head of list $T[h(key[x])]$
  ```

  - 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$(T, k)$
    search for an element with key $k$ in list $T[h(k)]$
  ```

  Running time is proportional to the length of the list of elements in slot $h(k)$.

- ***Deletion:***

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

  - 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 we must 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.

We focus on average-case performance of hashing with chaining.

- Assume ***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 $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, 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 simple uniform hashing applies.]

Unsuccessful search

Theorem
An unsuccessful search takes expected time $\Theta(1+\alpha)$.

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

So we need to find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

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

For all i and j, define indicator random variable $X_{ij} = \text{I}\{h(k_i) = h(k_j)\}$.

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m \Rightarrow \text{E}[X_{ij}] = 1/m$ (by Lemma 5.1).

Expected number of elements examined in a successful search is

$$\begin{aligned}
&\text{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\text{E}[X_{ij}]\right) \quad \text{(linearity of expectation)} \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad = 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(equation (A.1))} \\
&\quad = 1+\frac{n-1}{2m} \\
&\quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\ .
\end{aligned}$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$.

Alternative analysis, using indicator random variables even more

For each slot l and for each pair of keys k_i and k_j, define the indicator random variable $X_{ijl} = \text{I}\{\text{the search is for } x_i,\ h(k_i) = l, \text{ and } h(k_j) = l\}$. $X_{ijl} = 1$ when keys k_i and k_j collide at slot l and when we are searching for x_i.

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = l\} = 1/m$ and $\Pr\{h(k_j) = l\} = 1/m$. Also have $\Pr\{\text{the search is for } x_i\} = 1/n$. These events are all independent $\Rightarrow \Pr\{X_{ijl} = 1\} = 1/nm^2 \Rightarrow \text{E}[X_{ijl}] = 1/nm^2$ (by Lemma 5.1).

Define, for each element x_j, the indicator random variable

$$Y_j = \text{I}\{x_j \text{ appears in a list prior to the element being searched for}\}\ .$$

$Y_j = 1$ if and only if there is some slot l that has both elements x_i and x_j in its list, and also $i < j$ (so that x_i appears after x_j in the list). Therefore,

$$Y_j = \sum_{i=1}^{j-1}\sum_{l=0}^{m-1}X_{ijl}\ .$$

One final random variable: Z, which counts how many elements appear in the list prior to the element being searched for: $Z = \sum_{j=1}^{n} Y_j$. We must count the element being searched for as well as all those preceding it in its list $\Rightarrow$ compute $\mathrm{E}[Z+1]$:

$$\begin{aligned}
\mathrm{E}[Z+1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\
&= 1 + \mathrm{E}\left[\sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1} X_{ijl}\right] \quad \text{(linearity of expectation)} \\
&= 1 + \sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1} \mathrm{E}[X_{ijl}] \quad \text{(linearity of expectation)} \\
&= 1 + \sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1} \frac{1}{nm^2} \\
&= 1 + \binom{n}{2} \cdot m \cdot \frac{1}{nm^2} \\
&= 1 + \frac{n(n-1)}{2} \cdot \frac{1}{nm} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} .
\end{aligned}$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1+\alpha)$. ■

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 some issues regarding hash-function design and present schemes for hash function creation.

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.

- In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on 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're not, have to interpret them as natural numbers.
- ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is `CLRS`:
  - ASCII values: `C` $= 67$, `L` $= 76$, `R` $= 82$, `S` $= 83$.
  - There are 128 basic ASCII values.
  - So interpret `CLRS` as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141{,}764{,}947$.

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.

Disadvantage: Have to avoid certain values of m:

- Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- If k is a character string interpreted in radix 2^p (as in `CLRS` example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).

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.

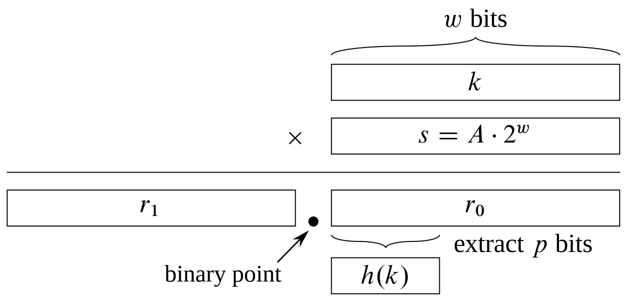
Put another way, $h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA \rfloor =$ fractional part of kA.

Disadvantage: Slower than division method.

Advantage: Value of m is not critical.

(Relatively) easy implementation:

- Choose $m = 2^p$ for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let s be an integer in the range $0 < s < 2^w$. (s takes w bits.)
- Restrict A to be of the form $s/2^w$.



- Multiply k by s.
- Since we're multiplying two w-bit words, the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($k A \bmod 1 = kA - \lfloor kA \rfloor$). Think of the "binary point" (analog of decimal point, but for binary representation) as being between r_1 and r_0. Since we don't care about the integer part of kA, we can forget about r_1 and just use r_0.
- Since we want $\lfloor m\,(k\,A \bmod 1) \rfloor$, we could get that value by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.
- We don't need to shift. The p bits that would have been shifted to the left of the binary point are the p most significant bits of r_0. So we can just take these bits after having formed r_0 by multiplying k by s.
- ***Example:*** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. Must have $0 < s < 2^5$; choose $s = 13 \Rightarrow A = 13/32$.
  - Using just the formula to compute $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32}$ $\Rightarrow k\,A \bmod 1 = 17/32 \Rightarrow m\,(k\,A \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow$ $\lfloor m\,(k\,A \bmod 1) \rfloor = 4$, so that $h(k) = 4$.
  - Using the implementation: $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$. Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.

How to choose A:

- The multiplication method works with any legal value of A.
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Universal hashing

[We just touch on universal hashing in these notes. See the book for a full treatment.]

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 he could choose keys that all hash to the same slot, giving worst-case behavior.

One way to defeat the adversary is to use a different hash function each time. You choose one at random at the beginning of your program. Unless the adversary knows how you'll be randomly choosing which hash function to use, he cannot intentionally defeat you.

Just because we choose a hash function randomly, that doesn't mean it's a good hash function. What we want is to randomly choose a single hash function from a set of good candidates.

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 $k, l \in U$, where $k \neq l$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is $\leq |\mathcal{H}|/m$.

Put another way, $\mathcal{H}$ 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 than $1/m$ chance of just choosing two slots randomly and independently.

Why are universal hash functions good?

- They give good hashing behavior:

  Theorem
  Using chaining and universal hashing on key k:

  - If k is not in the table, the expected length $\mathrm{E}[n_{h(k)}]$ of the list that k hashes to is $\leq \alpha$.
  - If k is in the table, the expected length $\mathrm{E}[n_{h(k)}]$ of the list that holds k is $\leq 1 + \alpha$.

  Corollary
  Using chaining and universal hashing, the expected time for each SEARCH operation is $O(1)$.

- They are easy to design.

[See book for details of behavior and design of a universal class of hash functions.]

Open addressing

An alternative to chaining for handling collisions.

Idea

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- 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.
  - There's a third possibility: slot $h(k)$ contains a key that is not k. We compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on.
  - Keep probing until we either find key k (successful search) or we find a slot holding NIL (unsuccessful search).
- We need the sequence of slots probed to be a permutation of the slot numbers $\langle 0, 1, \ldots, m-1 \rangle$ (so that we examine all slots if we have to, and so that we don't examine any slot more than once).
- Thus, the hash function is $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$.
- 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 we're searching, and insert at the first NIL slot we find.

Pseudocode for searching

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

HASH-SEARCH returns the index of a slot containing key k, or NIL if the search is unsuccessful.

Pseudocode for insertion

```
HASH-INSERT(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == NIL
          T[j] = k
          return j
      else i = i + 1
  until i == m
  error "hash table overflow"
```

HASH-INSERT returns the number of the slot that gets key k, or it flags a "hash table overflow" error if there is no empty slot in which to put key k.

Deletion

Cannot just put NIL into the slot containing the key we want to delete.

- Suppose we want to delete key k in slot j.
- And suppose that sometime after inserting key k, we were inserting key k', and during this insertion we had probed slot j (which contained key k).
- And suppose we then deleted key k by storing NIL into slot j.
- And then we search for key k'.
- During the search, we would probe slot j *before* probing the slot into which key k' was eventually stored.
- 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.

How to compute probe sequences

The ideal situation is ***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 simple uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It's hard to implement true uniform hashing, so we 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. They will make use of ***auxiliary hash functions***, which map $U \rightarrow \{0, 1, \ldots, m-1\}$.

Linear probing

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.

Linear probing suffers from ***primary clustering***: long runs of occupied sequences build up. And long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. Result is that the average search and insertion times increase.

Quadratic probing

As in linear probing, the probe sequence starts at $h'(k)$. Unlike linear probing, it jumps around in the table according to a quadratic function of the probe number: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $c_1, c_2 \neq 0$ are constants.

Must constrain c_1, c_2, and m in order to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$. (Problem 11-3 explores one way to implement quadratic probing.)

Can get ***secondary clustering***: if two distinct keys have the same h' value, then they have the same probe sequence.

Double hashing

Use two auxiliary hash functions, h_1 and h_2. h_1 gives the initial probe, and h_2 gives the remaining probes: $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$.

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 > 1.
- Could let m be prime and have $1 < h_2(k) < m$.

$\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence.

Analysis of open-address hashing

Assumptions

- Analysis is in terms of load factor α. We will assume that the table never completely fills, so we always have $0 \le n < m \Rightarrow 0 \le \alpha < 1$.
- Assume uniform 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 $1/(1-\alpha)$.

Proof Since the search is unsuccessful, every probe is to an occupied slot, except for the last probe, which is to an empty slot.

Define random variable X = # of probes made in an unsuccessful search.

Define events A_i, for $i = 1, 2, \ldots$, to be the event that there is an ith probe and that it's to an occupied slot.

$X \geq i$ if and only if probes $1, 2, \ldots, i-1$ are made and are to occupied slots $\Rightarrow$ $\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$.

By Exercise C.2-5,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .\end{aligned}$$

Claim

$\Pr\{A_j \mid A_1 \cap A_2 \cap \cdots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$. Boundary case: $j = 1$ $\Rightarrow \Pr\{A_1\} = n/m$.

Proof For the boundary case $j = 1$, there are n stored keys and m slots, so the probability that the first probe is to an occupied slot is n/m.

Given that $j-1$ probes were made, all to occupied slots, the assumption of uniform hashing says that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, which in turn implies that the next probe is to a slot that we have not yet probed. There are $m - j + 1$ slots remaining, $n - j + 1$ of which are occupied. Thus, the probability that the jth probe is to an occupied slot is $(n - j + 1)/(m - j + 1)$. ■ (claim)

Using this claim,

$$\Pr\{X \geq i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}} \ .$$

$n < m \Rightarrow (n - j)/(m - j) \leq n/m$ for $j \geq 0$, which implies

$$\begin{aligned}\Pr\{X \geq i\} &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \ .\end{aligned}$$

By equation (C.25),

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \qquad \text{(equation (A.6))} \ .\end{aligned}$$

■ (theorem)

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 $\dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$.

Proof A successful search for key k follows the same probe sequence as when key k was inserted.

By the previous corollary, if k was the $(i + 1)$st key inserted, then α equaled i/m at the time. Thus, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$.

That was assuming that k was the $(i + 1)$st key inserted. We need to average over all n keys:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \qquad \text{(by inequality (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \qquad \blacksquare \text{ (theorem)}
\end{aligned}$$