Hash Tables (Ch. 11)

Many applications require a dynamic set that supports only Insert, Delete, and Search. E.g., a dictionary ADT.

A hash table is, usually, an array. It tries to get the benefit of array's direct addressing when we may have more than one key per address.

Definition:
- U is the "key space", the set of all possible keys
- $K \subseteq U$ is the set of keys seen

Goals:
- fast implementation of all operations -- O(1) time
- space efficient data structure -- O(n) space if n elements in dictionary

Approach 1: Linked Lists

Linked List Implementation
- Insert(x) : add x at head of list
- Search(k) : start at head and scan list
- Delete(x) : start at head, scan list, and then delete if found

Running Times: (assume n elements in list)
- Insert(x) : O(1) time
- Search(k) : worst-case -- element at end of list: n operations
  average-case -- element at middle of list: n/2 ops
  best-case -- element at head of list: 1 op
- Delete(x) : same as searching

We'd like O(1) time for all operations, we have O(n) for two.

Space Usage: O(n) space -- very space efficient, only uses what is needed to store the data at any time.

Approach 2: Direct-Addressing

Direct-Address Table Assume U = {0, 1, 2, ..., m}.
The data structure is an ARRAY T[0...m]
- Insert(x) : T[key[x]] := x
- Search(k) : return T[k]
- Delete(x) : T[key[x]] := NIL

Running Times: (assume n elements in list)
- Insert(x) : O(1) time
- Search(k) : O(1) time
- Delete(x) : O(1) time

Great running time!

Space Usage: (assume n elements to be stored in list).
- O(m) space always!
- bad if n << m

Approach 3: Hashing

Hashing
- **hash table** (an array) H[0..m], where m << |U|
  - amount of storage closer to what is really needed
- **hash function** h is a mapping of keys to indices in H
  - $h : U \rightarrow \{0, 1, ..., m\}$

Problem: there will be some ***collisions***; that is, h will map some keys to the same position in H (i.e., $h(k_1) = h(k_2)$ for $k_1 \neq k_2$).

Different methods of resolving collisions:
1. **chaining:** put all elements that hash to same location in a linked list at that location.
2. **open addressing:** each time there is a collision, a probe number (initially 1) is incremented. There are various types of probe sequences:
   - linear probing
   - quadratic probing
   - double hashing

Hash Functions

- The mapping of keys to indices of a hash table is called a ***hash function***

Purpose of hash function is to translate an extremely large key space into a reasonably small range of integers, i.e., to map each key k to a position in the hash table.

- A hash function is usually the composition of two functions, a ***hash code map*** and a ***compression map.***
  - An essential requirement of the hash function is to map equal keys to equal indices
  - A "good" hash function minimizes the probability of collisions

Choosing Hash Functions

Ideally, a hash function satisfies the ***Simple Uniform Hashing Assumption***. Unfortunately, we cannot usually achieve this...so we use **heuristics**.

Assumption: Simple Uniform Hashing
- Any key is equally likely to hash to any location (index, slot) in hash table

Hash-Code Maps

A hash function assigns each key k in our dictionary an integer value, called the **hash code** or **hash value**. This integer does not necessarily have to be in the range [0, m-1] and it can be negative.

An essential feature of a hash code is consistency, i.e., it should map all items with key k to the same integer.

Common hash code maps:

1) Component sum: for numeric types with more than 32 bits, we can add the 32-bit components, i.e., sum the high-order bits with the low-order bits. Integer result is the hash code.

Hash-Code Maps

Common hash code maps (cont.):

2) Polynomial accumulation: for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 a_1 \ldots a_{n-1}$ by viewing them as the coefficients of a polynomial:

$$a_0 + a_1 x + a_2 x^2 \ldots + a_{n-1} x^{n-1}$$

-The polynomial is computed with Horner's rule at a fixed value x (a non-zero constant):

$$a_0 + x\,(a_1 + x\,(a_2 + \ldots\, x\,(a_{n-2} + x\,a_{n-1}) \ldots))$$

-The choice x = 33, 37, 39, or 41 gives at most 6 collisions on a vocabulary of 50,000 English words

Compression Maps

Normally, the range of possible hash codes generated for a set of keys will exceed the range of the array.

So we need a way to map this integer into the range [0, m-1].

Division method: h(k) = k mod m

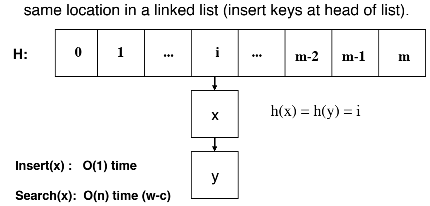
– the table size m is usually chosen as a prime number to help "spread out" the distribution of hashed values

0	1	2	3	4	5	6	7	8	9	10
22 11		2 13			5 16					

For example, each pair of keys 5 and 16, 22 and 11, 2 and 13 would hash to the same index if m = 11.

Collision Resolution by Chaining

Chaining: Use array of linked lists. Put all keys that hash to the same location in a linked list (insert keys at head of list).



Insert(x) : O(1) time

Search(x): O(n) time (w-c)

Delete(x): O(n) time (w-c)

NOTE: The idea of hashing is to get the average-case behavior down to θ(1) for all operations

Collision Resolution by Open Addressing

In this method, the hash function includes the probe number (i.e., how many attempts have been made to find a slot for this key) as an argument.

- the *probe sequence* for key *k = h(k,0), h(k,1),..., h(k,m-1)*

- In the worst case, every slot in table will be examined, so stop looking either when the item with key k is found (if searching) or an empty slot is found (if inserting)

Modifying the placement using the probe value is known as rehashing.

Linear Probing (Open Addressing)

<u>Linear Probing:</u> Simplest rehashing functions (e.g., add 1 for each probe) the ith probe (where i is initially 0) h(k, i) is

h(k, i) = (h'(k) + i) mod m

- h'(k) is ordinary hashing function, tells where to start the search.
- search sequentially through table (with wrap around) from starting point.

How many distinct probe sequences are there? m

- each starting point gives a probe sequence
- there are m starting points

plus: easy to implement
minus: leads to clustering (long run of occupied slots in H), yields bad performance if a key collides with an element in a cluster (also known as *primary clustering).*

Linear Probing Example

- h(k, i) = (h(k) + i) mod m (i is probe number, initially, i = 0)
- Insert keys: 18 41 22 44 59 32 31 73 (in that order)

31 44 32 73

How many collisions occur in this case?

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

h(k) = k mod 13
m = 13

If a collision occurs, when j = h(k), we try next at A[(j+1)mod m], then A[(j+2)mod m], and so on. When an empty position is found the item is inserted.

Each time key is compared to number in the array, there is a collision.

Quadratic Probing (Open Addressing)

<u>Quadratic Probing:</u> the ith probe h(k,i) is

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

- c_1 and c_2 are constants
- h'(k) is ordinary hash function, tells where to start the search
- later probes are offset by an amount quadratic in i (the probe number).

How many distinct probe sequences are there? m
- each starting point gives a probe sequence
- there are m starting points

plus: easy to implement
minus: leads to *secondary clustering*

Quadratic Probing

Insert keys: 18 41 22 44 59 32 31 73 (in that order)

31 44 73 31

How many collisions occur in this case?

0	1	2	3	4	5	6	7	8	9	10	11	12
73		41			18	32	59	31	22	44		

h(k) = k mod 13
m = 13
$c_1 = 2$, $c_2 = 3$

44 % 13 = 5 (collision), next try: $(5 + 2 \cdot 1 + 3 \cdot 1^2)$ % 13 = 10

31 % 13 = 5 (collision), next try: $(5 + 2 \cdot 1 + 3 \cdot 1^2)$ % 13 = 10 % 13 = 10 (collision)

next try: $(5 + 2 \cdot 2 + 3 \cdot 2^2)$ % 13 = 21 % 13 = 8

73 % 13 = 8 (collision), next try: $(8 + 2 \cdot 1 + 3 \cdot 1^2)$ % 13 = 0

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

Double Hashing (Open Addressing)

<u>Double Hashing:</u> the ith probe h(k,i) is

$$h(k, i) = (h_1(k) + h_2(k) \cdot i) \bmod m$$

- $h_1(k)$ is ordinary hash function, tells where to start the search
- $h_2(k)$ is ordinary hash function that gives offset for subsequent probes.

Note: $h_2(k)$ should be relatively prime to m.

How many distinct probe sequences are there?
- there are m starting points
- starting point and offset can vary independently

Double Hashing Example

- $h_1(K)$ = K mod m
- $h_2(K)$ = K mod (m – 1)
- The i^{th} probe is $h(k, i) = (h_1(k) + h_2(k) \cdot i) \bmod m$
- we want h_2 to be an offset to add

Insert keys: 18 41 22 44 59 32 31 73 (in that order)

31 44

How many collisions occur in this case?

0	1	2	3	4	5	6	7	8	9	10	11	12
44		41			18	32	59	73	22			31

m = 13

44 % 13 = 5 (collision), next try: (5 + (44 % 12)) % 13 = 13 % 13 = 0

31 % 13 = 5 (collision), next try: (5 + (31 % 12)) % 13 = 12 % 13 = 12

Analyzing Open Addressing

α = n/m (load factor). We need $\alpha \leq 1$ (table cannot be overfilled).

If the load $\alpha < 1$, then the expected number of probes in a successful search is

$$\leq (1/\alpha)\ln(1/(1-\alpha))$$

Thus, for example, we have:

- o if the hash table is half full, (α = .5), then the expected number of probes in a successful search is 2ln 2 < 1.386.
- o if the hash table is 90% full, (α = .9), then the average number of probes in a successful search is 1.1 ln 10 < 2.558.

If α is a constant ≤ 1, a successful search runs in O(1) time.