Hash Tables (Ch. 11)

Very efficient way to implement dictionaries. A dictionary is an ADT with the operations search, insert, and delete.

Most often used to store records, a non-trivial amount of information, each piece of which is accessed via a unique **key**.

Based on idea of distributing keys within a one-dimensional array [0...m-1] called a **hash table**
- distribution is done by computing the value of some predefined function *h* called a **hash function**
- The hash function assigns an integer between 0 and m-1, the **hash address**, to a key.

Hash functions often assign keys that are nonnegative integers using a simple function such as h(K) = K mod m, where m is the size of the array.

Hashing

Requirements for a hash function:
1. Every equal key hashes to the same position in the array.
2. The function must be easy to compute (fast).

Desirable feature of the hash function is that it distributes keys among the cells of the hash table as evenly as possible.

Definition:
- U is the "key space", the set of all possible keys
- K ⊆ 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(k) : add k at head of list
- Search(k) : start at head and scan list
- Delete(k) : start at head, scan list, and then delete if found

Running Times: (assume n elements in list)
- Insert(k) : O(1) time
- Search(k) : worst-case -- element at end of list: n operations
- Delete(k) : 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 memory needed to store the data at any time.

Approach 2: Direct-Addressing

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

Running Times: (assume n elements in list)
- Insert(k) : O(1) time
- Search(k) : O(1) time
- Delete(k) : 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 → {0, 1, ..., m}

Problem: since m << |U|, there will be possible ***collisions***; that is, h will map some keys to the same position in H

(i.e., $h(k_1) = h(k_2)$ for $k_1 \neq k_2$).

Collision Resolution

Different methods of resolving collisions:

1. **open hashing (separate chaining):** put all elements that hash to same location in a linked list at that location. Uses an array of singly-linked lists.

2. **closed hashing (open addressing):** Single element per array position...each time there is a collision, a probe number (initially 1) is incremented. There are various types of probe sequences (we will look at 3):
   - linear probing
   - quadratic probing
   - double hashing

Open Hashing (Separate Chaining)

Keys are stored in an array of linked lists (the hash table).

To distribute keys as evenly as possible, choose a prime number for m.

Example: A very simple hash function for strings of letters is to sum all the positions of a word's letters in the alphabet and compute the sum's remainder after division by 13, a randomly-chosen prime number.

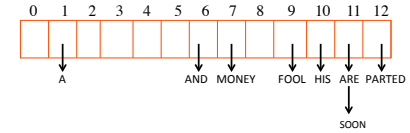
Consider the following list of words:

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 5 10 15 20 25

Open Hashing (Separate Chaining)

h(A) = 1%13 = 1, h(FOOL) = 48%13 = 9, h(AND) = 19%13 = 6,
h(HIS) = 36%13 = 10, h(MONEY) = 72%13 = 7, h(ARE) = 24%13 = 11,
h(SOON) = 63%13 = 11, h(PARTED) = 64%13 = 12



A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 5 10 15 20 25

Load Factor

If the hash function distributes n keys among m cells about evenly, each list will be about n/m keys long. The ratio α = n/m is called the *load factor*.

In practice, it is best to keep the load factor of a separate chaining implementation at about 1, providing the length of most chains is small.

Separate chaining has the advantage that deletions are not complicated, an item is just removed from the list with a couple of link changes.

Hash Functions

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

The purpose of a 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 ensure this...so we use **heuristics**.

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

Collision Resolution by Open Addressing

Using open addressing, there is only 1 key per position in the hash table.

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 or searching)

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

Deleting an element from a hash table using open addressing is not as easy as it is with separate chaining.

Linear Probing (Open Addressing)

Linear Probing: Simplest rehashing function (e.g., add 1 for each probe); the ith probe (where i is initially 0) h(k, i) is

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

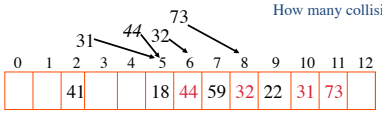
- h'(k) is ordinary hashing function, tells where to start the search.
- search sequentially through table (with wrap around) from starting point.

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)

How many collisions occur in this case?



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)

Quadratic Probing: the ith probe h(k,i) is

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

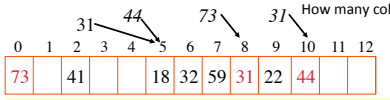
- c_1 and c_2 are constants
- h'(k) is ordinary hash function, tells where to start the search
- later probes are offset by an amount quadratic in i (the probe number).

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

Quadratic Probing

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

How many collisions occur in this case?



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)

Double Hashing: 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.

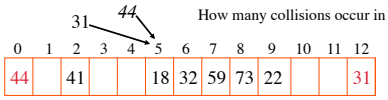
Note: $h_2(k)$ should be relatively prime to m.

Double Hashing Example

- $h_1(K) = K \bmod m$
- $h_2(K) = K \bmod (m - 1)$
- 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)

How many collisions occur in this case?



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:

- if the hash table is half full, ($\alpha = .5$), then the expected number of probes in a successful search is $2 \ln 2 < 1.386$.
- if the hash table is 90% full, ($\alpha = .9$), then the expected 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.