Data Structures

Hash table: Hash functions

- Hash codes:
  - Summation hash code: not a good choice, if keys are either strings or other multiple-length objects viewed as k-tuple $(x_0, x_1, \ldots, x_{k-1})$, where order of x_i's is significant
  - "temp01", "temp10" collide
  - temp01 (116, 101, 109, 112, 48, 49)
  - "spot", "pots", "stop", "tops" collide
  - spot (115, 112, 111, 116)

Hash table: Hash functions

- Hash codes:
  - Integer representation of key x is $(x_0, x_1, . . ., x_{k-1})$
  - Polynomial hash code: Choose a nonzero constant "a" and use as a hash code the value

    $$x_0a^{k-1} + x_1 a^{k-2} + . . . + x_{k-2}a + x_{k-1},$$

    which can be written as:

    $$x_{k-1} + a(x_{k-2} + a(x_{k-3} + . . . + a(x_2 + a(x_1 + ax_0)). . .))$$
  - This hash code uses the components of key "x" as coefficients of the polynomial in "a" (polynomial hash code)
  - Taking "a" to be 33, 37, 39, and 41 produced less than 7 collisions on a list of 50, 000 English words

Hash table: Hash functions

- Second action (compression map):
- Once key object is converted into hash code, it has to be converted into an integer in the range [0, N-1]
- A simple compression map method (division method) is:

$$h(k) = |k| \bmod N$$

- {20, 25, 30, 35, 40, 45, 50}, assume that N =10
- Consider N =11
- If we choose N as a prime, then this method spreads out the distribution of hashed values
- If N is 2^p and then h(k) is p lower-order bits of k

Hash table: Hash functions

- If key values are of the form iN + j for several different i's, then there will be collisions though N is a prime
- The MAD method:
  - Multiply, add, and divide
  - Hash function is defined as: $h(k) = \{ak + b\} \bmod N$; N is prime, a and b are nonnegative integers randomly chosen, $a \bmod N \neq 0$
  - To get close to a good hash function such that the probability of two keys getting hashed to the same bucket is at most 1/N

Hash table: Hash functions

- The multiplication method
  - Multiply key k by a constant A in the range 0 < A < 1
  - Extract the fractional portion, f, of the result
  - Multiply f by m and take the floor of the result (m is hash table capacity)
  - h(k) = $\lfloor m(kA\ mod\ 1) \rfloor$
  - kA mod 1 = kA - $\lfloor kA \rfloor$
  - Value of m is not critical here
  - Can choose m as a power of 2 (say 2^p) for easy implementation of hash function
  - Assume that the word size of machine is "w" bits and k fits in a word
  - Restrict A to be of form $s/2^w$, where $0 < s < 2^w$
  - Multiply k by w-bit integer s = A. 2^w
  - The result is a 2w-bit value $r_1\ 2^w + r_0$
  - The most significant p bits of r_0 is the hash value of k

Hash table: Hash functions

- Assume that the word size of the machine 8-bits
- Select m = 2^3 and A as 0.25
- Consider key k = 51
- h(k) = $\lfloor m(kA \mod 1) \rfloor$ = $\lfloor 8(0.75) \rfloor$ = 6
- s = A. 2^w = 64
- ks = 51*64 = 3264
- 12*2^8 + 192
- 192: 1100 0000

Collision handling schemes

- Consider two items (k_1, e_1) and (k_2, e_2)
- If $h(k_1) = h(k_2)$, then we have a collision
- Which operations get affected?
  - insertItem() and findItem()
- A simple and efficient way is to have each bucket A[i] to store a reference to an unordered sequence (list), S_i, that stores all items that are mapped to bucket A[i]
- Each bucket is a miniature dictionary
- This way of collision resolution is called separate chaining
- Assume that each nonempty bucket is implemented as a list

Separate chaining

```
Algorithm findElement(k)
    B ← A[h(k)]
    if B is empty then
        return NO_SUCH_KEY
    else
        {search for key in the list for this bucket}
        return B.findElement(k)
```

Separate chaining

```
Algorithm insertItem(k, e)
        if A[h(k)] is empty then
                Create a new list B, which is initially empty
                A[h(k)] ← B
        else
                B ← A[h(k)]
        B.insertItem(k,e)
```

Separate chaining

```
Algorithm removeElement(k)
    B ← A[h(k)]
    If B is empty then
        return NO_SUCH_Key
    else
        return B.removeElement(k)
```

Separate chaining

- Insertion runs in constant time (item is not present in the table)
- In the worst-case the time to search an item is $\theta(n)$
- Consider a hash table T of capacity m that stores n items
- Average-case performance depends on how well the hash function distributes keys among m buckets on average
- Assume that any given item is equally likely to hash to any of the m buckets independent of where any other item has hashed to (simple uniform hashing)
- The load factor of T, α, is defined as n/m, that is, average number of elements stored in each list/chain
- α can be less than, equal to, or greater then 1

Separate chaining

- n_j is the length of the list pointed by T[j], where j = 0, 1, . . ., m-1
- $n = n_0 + n_1 + . . . + n_{m-1}$
- $E[n_j] = \alpha = n/m$
- Assume that the time to compute hash function is O(1)
- Time required to search for an item with key k is linearly dependent on the length $n_{h(k)}$ of the list referred by T[h(k)]
- Analyse the expected number of items examined by the search algorithm, that is, the number of items in the list referred by T[h(k)]
- Consider two cases: unsuccessful search and successful search

Separate chaining

Theorem: In a hash table, if collisions are resolved by chaining, an unsuccessful search takes average-case time $\theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof:

Any key k which is not present in table T is equally likely to hash to any of the m buckets

The expected time to perform an unsuccessful search is the expected time to search to the end of the list of T[h(k)]

The expected length of the list of T[h(k)] is $E[n_{h(k)}]$ is α

The expected number of items examined in an unsuccessful search is α

Total required for an unsuccessful search is $\theta(1 + \alpha)$

Separate chaining

Theorem: In a hash table if collisions are resolved by chaining, a successful search takes average-case time $\theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof:

Item to be searched is equally likely to be any of n items stored in the table

The number of items searched in a successful search for an item x is one more than the number of items that precede x in x's list (why?)

Find the number of items that were inserted after x was inserted in x's list

Separate chaining

Proof (contd):

Let x_i be the ith item inserted into the table, for i =1, 2, . . ., n and let $k_i = x_i$.key

For keys k_i and k_j define a random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$

Under the assumption of simple uniform hashing, $Pr\{h(k_i) = h(k_j)\} = 1/m$, so $E[X_{ij}] = 1/m$

The number of items that were searched in a successful search for x_i is: $\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)$

The expected number of items searched in a successful search is:

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

Separate chaining

Proof (Contd):

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

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

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

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

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

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

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

Separate chaining

Proof (Contd):

Thus the total time required for a successful search is: $\theta(2 + \alpha/2 - \alpha/2n)$, which is $\theta(1 + \alpha)$

- If hash table capacity is at least proportional to the number of items in the table, then n is O(m)
- $\alpha = n/m$ is $O(m)/m$, which is $O(1)$
- Thus searching takes constant time
- If the lists are maintained using doubly linked lists, then removal also takes constant time