CMSC 341

Hashing
The Basic Problem

- We have lots of data to store.

- We desire efficient – $O(1)$ – performance for insertion, deletion and searching.

- Too much (wasted) memory is required if we use an array indexed by the data’s key.

- The solution is a “hash table”.
Hash Table

- Basic Idea
  - The hash table is an array of size ‘m’
  - The storage index for an item determined by a hash function \( h(k): U \rightarrow \{0, 1, \ldots, m-1\} \)

- Desired Properties of \( h(k) \)
  - easy to compute
  - uniform distribution of keys over \( \{0, 1, \ldots, m-1\} \)
    - when \( h(k_1) = h(k_2) \) for \( k_1, k_2 \in U \), we have a collision
Division Method

- The hash function:
  \[ h(k) = k \mod m \text{ where } m \text{ is the table size.} \]
- \( m \) must be chosen to spread keys evenly.
  - Poor choice: \( m = \text{a power of 10} \)
  - Poor choice: \( m = 2^b, \ b > 1 \)
- A good choice of \( m \) is a prime number.
- Table should be no more than 80% full.
  - Choose \( m \) as smallest prime number greater than \( m_{\text{min}} \), where \( m_{\text{min}} = (\text{expected number of entries})/0.8 \)
Multiplication Method

- The hash function:
  \[ h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor \]
  where \( A \) is some real positive constant.

- A very good choice of \( A \) is the inverse of the “golden ratio.”

- Given two positive numbers \( x \) and \( y \), the ratio \( x/y \) is the “golden ratio” if \( \phi = x/y = (x+y)/x \)

- The golden ratio:
  \[ x^2 - xy - y^2 = 0 \Rightarrow \phi^2 - \phi - 1 = 0 \]
  \[ \phi = (1 + \sqrt{5})/2 \approx 1.618033989… \]
  \[ \sim= \text{Fib}_i/\text{Fib}_{i-1} \]
Because of the relationship of the golden ratio to Fibonacci numbers, this particular value of A in the multiplication method is called “Fibonacci hashing.”

Some values of

\[ h(k) = \left\lfloor m(k \phi^{-1} - \left\lfloor k \phi^{-1} \right\rfloor) \right\rfloor \]

- for \( k = 0 \), \( h(k) = 0 \)
- for \( k = 1 \), \( h(k) = 0.618m \)
- for \( k = 2 \), \( h(k) = 0.236m \)
- for \( k = 3 \), \( h(k) = 0.854m \)
- for \( k = 4 \), \( h(k) = 0.472m \)
- for \( k = 5 \), \( h(k) = 0.090m \)
- for \( k = 6 \), \( h(k) = 0.708m \)
- for \( k = 7 \), \( h(k) = 0.326m \)
- ... for \( k = 32 \), \( h(k) = 0.777m \)
Fibonacci Hashing

![Graph of Fibonacci Hashing](image)
Non-integer Keys

- In order to have a non-integer key, must first convert to a positive integer:

  \[ h(k) = g(f(k)) \] with \( f: U \rightarrow \text{integer} \)

  \[ g: I \rightarrow \{0..m-1\} \]

- Suppose the keys are strings.

- How can we convert a string (or characters) into an integer value?
static int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++)
        hashVal = 37 * hashVal + key.charAt(i);
    hashVal %= tableSize;
    if (hashVal < 0)
        hashVal += tableSize;
    return hashVal;
}
HashTable Class

public class SeparateChainingHashTable<AnyType>
{
    public SeparateChainingHashTable(){/* Later */}
    public SeparateChainingHashTable(int size){/*Later*/}
    public void insert(AnyType x){ /*Later*/ }
    public void remove(AnyType x){ /*Later*/}
    public boolean contains(AnyType x){/*Later */}
    public void makeEmpty(){ /* Later */}
    private static final int DEFAULT_TABLE_SIZE = 101;
    private List<AnyType> [ ] theLists;
    private int currentSize;
    private void rehash(){ /* Later */}
    private int myhash(AnyType x){ /* Later */}
    private static int nextPrime( int n){ /* Later */}
    private static boolean isPrime( int n){ /* Later */}
}
HashTable Ops

- boolean contains( AnyType x )
  - Returns true if x is present in the table.

- void insert (AnyType x)
  - If x already in table, do nothing.
  - Otherwise, insert it, using the appropriate hash function.

- void remove (AnyType x)
  - Remove the instance of x, if x is present.
  - Otherwise, does nothing

- void makeEmpty()
private int myhash( AnyType x ) {
    int hashVal = x.hashCode();
    hashVal %= theLists.length;
    if( hashVal < 0 )
        hashVal += theLists.length;
    return hashVal;
}
Handling Collisions

- Collisions are inevitable. How to handle them?

- Separate chaining hash tables
  - Store colliding items in a list.
  - If $m$ is large enough, list lengths are small.

- Insertion of key $k$
  - $\text{hash}( k )$ to find the proper list.
  - If $k$ is in that list, do nothing, else insert $k$ on that list.

- Asymptotic performance
  - If always inserted at head of list, and no duplicates, $\text{insert} = O(1)$: best, worst, average
Hash Class for Separate Chaining

To implement separate chaining, the private data of the hash table is a vector (array) of Lists. The hash functions are written using List functions

```
private List<AnyType> [] theLists;
```
Performance of contains()

- contains
  - Hash k to find the proper list.
  - Call contains( ) on that list which returns a boolean.

- Performance
  - best:
  - worst:
  - average
Performance of remove()

- Remove k from table
  - Hash k to find proper list.
  - Remove k from list.
- Performance
  - best
  - worst
  - average
Handling Collisions Revisited

- **Probing hash tables**
  - All elements stored in the table itself (so table should be large. Rule of thumb: \( m \geq 2N \))
  - Upon collision, item is hashed to a new (open) slot.

Hash function

\[
h( k, i ) = ( h'( k ) + f( i ) ) \mod m
\]

for some \( h' : U \rightarrow \{0, 1, \ldots, m-1\} \)

and some \( f( i ) \) such that \( f(0) = 0 \)

- Each attempt to find an open slot (i.e. calculating \( h( k, i ) \)) is called a **probe**
HashEntry Class for Probing Hash Tables

- In this case, the hash table is just an array

```java
private static class HashEntry<AnyType> {
    public AnyType element; // the element
    public boolean isActive; // false if deleted
    public HashEntry( AnyType e ) {
        this( e, true );
    }
    public HashEntry( AnyType e, boolean active ) {
        element = e; isActive = active;
    }
}
// The array of elements
private HashEntry<AnyType> [] array;
// The number of occupied cells
private int currentSize;
```
Linear Probing

- Use a linear function for $f(i)$
  \[ f(i) = c \times i \]

- Example:
  \[ h'(k) = k \mod 10 \] in a table of size 10, \[ f(i) = i \]
  So that
  \[ h(k, i) = (k \mod 10 + i) \mod 10 \]

Insert the values $U=\{89,18,49,58,69\}$ into the hash table.
Linear Probing (cont.)

- Problem: Clustering
  - When the table starts to fill up, performance $\rightarrow O(N)$

- Asymptotic Performance
  - Insertion and unsuccessful find, average
    - $\lambda$ is the “load factor” – what fraction of the table is used
    - Number of probes $\approx \left( \frac{1}{2} \right) \left( 1 + 1/(1-\lambda)^2 \right)$
    - if $\lambda \geq 1$, the denominator goes to zero and the number of probes goes to infinity
Remove

- Can’t just use the hash function(s) to find the object and remove it, because objects that were inserted after X were hashed based on X’s presence.
- Can just mark the cell as deleted so it won’t be found anymore.
  - Other elements still in right cells
  - Table can fill with lots of deleted junk
Quadratic Probing

- Use a quadratic function for $f( i )$
  \[ f( i ) = c_2 i^2 + c_1 i + c_0 \]
  The simplest quadratic function is $f( i ) = i^2$

- Example:
  Let $f( i ) = i^2$ and $m = 10$
  Let $h'( k ) = k \mod 10$

    So that
    \[ h( k, i ) = (k \mod 10 + i^2) \mod 10 \]

  Insert the value $U=\{89, 18, 49, 58, 69\}$ into an initially empty hash table
Quadratic Probing (cont.)

- **Advantage:**
  - Reduced clustering problem

- **Disadvantages:**
  - Reduced number of sequences
  - No guarantee that empty slot will be found if $\lambda \geq 0.5$, even if $m$ is prime
  - If $m$ is not prime, may not find an empty slot even if $\lambda < 0.5$
Double Hashing

- Let $f(i) = i \times h_2(k)$
  
  Then $h(k, l) = \left( h'(k) + l \times h_2(k) \right) \mod m$

  And probes are performed at distances of $h_2(k), 2 \times h_2(k), 3 \times h_2(k), 4 \times h_2(k)$, etc

- Choosing $h_2(k)$
  - Don’t allow $h_2(k) = 0$ for any $k$.
  - A good choice: $h_2(k) = R - (k \mod R)$ with $R$ a prime smaller than $m$

- Characteristics
  - No clustering problem
  - Requires a second hash function
Rehashing

- If the table gets too full, the running time of the basic operations starts to degrade.
- For hash tables with separate chaining, “too full” means more than one element per list (on average)
- For probing hash tables, “too full” is determined as an arbitrary value of the load factor.
- To rehash, make a copy of the hash table, double the table size, and insert all elements (from the copy) of the old table into the new table
- Rehashing is expensive, but occurs very infrequently.