CMSC 341

Hashing

Readings: Chapter 5
Announcements

- Midterm II on Nov 7
- Review out Oct 29
- HW 5 due Thursday
- Project due Nov 5
- Midterm II review posted on Tuesday
Motivations

- 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”.

CMSC 341 Hashing
Hash Table

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

Example: insert 89, 18, 49, 58, 69 to a table size of 10.

Hash function: \( h(k) = k \mod m \) where \( m \) is the table size.

```java
Public static int hash(String key, int tableSize)
{
    hashVal %= tableSize;

    return hashVal;
}
```

What is the problem here? How to resolve it?

Hints:
(1) How should we choose \( m \)?
(2) How to pick a hashing function?

Getting a better hash function; make a table (instead we make a linked list); pick a better table size (prime number)
Hashing function: $F(i) = i$

Example: $h'(k) = k \mod 10$ in a table of size 10 (not prime, but easy to calculate)

$U = \{89, 18, 49, 58, 69\}$

$f(I) = I$

1. 89 hashes to 9
2. 18 hashes to 8
3. 49 hashes to 9, collides with 89
   \[ h(k,1) = (49 \% 10 + 1) \% 10 = 0 \]
4. 58 hashes to 8, collides with 18
   \[ h(k,1) = (58 \% 10 + 1) \% 10 = 9, \text{ collides with } 89 \]
   \[ h(k,2) = (58 \% 10 + 2) \% 10 = 0, \text{ collides with } 49 \]
   \[ h(k,3) = (58 \% 10 + 3) \% 10 = 1 \]
5. 69 hashes to 9, collides with 89
   \[ h(69,1) = (h'(69) + f(1)) \mod 10 = 0, \text{ collides with } 49 \]
   \[ h(69,2) = (h'(69) + f(2)) \mod 10 = 0, \text{ collides with } 58 \]
   \[ h(69,3) = (h'(69) + f(3)) \mod 10 = 2 \]
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:
  \[ h(k) = k \mod m \] where \( m \) is the table size.

- \( m \) must be chosen to spread keys evenly.
  - Poor choice: \( m = \) 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
    \[ m_{\text{min}} = \frac{\text{(expected number of entries)}}{0.8} \]
Handle Non-integer Keys

- In order to have a non-integer key, must first convert to a positive integer:
  \[ h(k) = g(f(k)) \text{ with } f: U \to \text{integer} \]
  \[ g: l \to \{0 .. m-1\} \]

- Suppose the keys are strings.

- How can we convert a string (or characters) into an integer value?
Horner’s Rule

```java
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;
}
```
Exercise: Hash Function

Which hashFunction is better, when tableSize = 10,007?

**Method 1:**

Public static int hash(String key, int tableSize)
{
    int hashVal = 0;
    for(int i=0; i<key.length(); i++)
        hashVal += key.charAt (i);
    return hashVal % tableSize
}  // not good: waste a lot of memory

**Method 2: Assuming three letters**

Public static int hash(String key, int tableSize)
{ return (key.charAt(0)+27*key.charAt(1)+27^2*key.charAt(2)) %
    tableSize; }

**Method 3:**

Public 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;
}
HashTable Class

```java
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
  - 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, insert = O(1) for best, worst and average cases
Hash Class for Separate Chaining

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

```java
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: selected list is empty or key is first -> \(O(1)\)
  - worst: let \(N\) be the number of elements in the hash table. All \(N\) elements are in one list (all have the same hash value) and key not there -> \(O(N)\)
  - Average: suppose there are \(M\) buckets and \(N\) elements in the table. Then expected list length = \(N/M\) -> \(O(N/M) = O(N)\) if \(M\) is small. = \(O(1)\) if \(M\) is large.

  Here \(\lambda = N/M\) is called the load factor of the table. It is important to keep the load factor from getting too large. If \(N <= M\), \(\lambda <=1\) and \(O(N/M) -> O(1)\) where \(N/M\) is constant.
Performance of `remove()`

- Remove k from table
  - Hash k to find proper list.
  - Remove k from list.

Performance

- Best: K is the 1st element on list, or list is empty: O(1)
- Worst: all elements on one list: O(n)
- Average: O(N/M) -> O(1) for \( \lambda \leq 1 \). So what is the big deal? Performance for hash table and list are the same best and worst… But average performance for a well-designed hash table is much better: O(1).
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**
  \[
  h: \ U \times \{0,1,2,\ldots\} \rightarrow \{0,1,\ldots,m-1\}
  \]
  \[
  h( k, i ) = ( h'( k ) + f( i ) ) \mod m
  \]
  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

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:
  \[ h'(k) = k \mod 10 \text{ in a table of size 10}, f(i) = i \]
  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 $\to O(N)$

- **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 $\lambda \approx 1$, the denominator goes to zero and the number of probes goes to infinity
Linear Probing (cont.)

- 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 $f(i)$ use another hash function
  
  $$f(i) = i * h_2(k)$$

  Then $h(k, l) = (h'(k) + i * h_2(k)) \mod m$

  And probes are performed at distances of
  
  $h_2(k), 2 * h_2(k), 3 * h_2(k), 4 * h_2(k), \text{ etc}$

- 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.
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:
  \[ x^2 - xy - y^2 = 0 \quad \Rightarrow \quad \phi^2 - \phi - 1 = 0 \]
  \[ \phi = (1 + \sqrt{5})/2 \quad = \quad 1.618033989\ldots \]
  \[ \sim = \text{Fib}_i/\text{Fib}_{i-1} \]

CMSC 341 Hashing
Multiplication Method (cont.)

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

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

- \( h(k) = 0 \) for \( k = 0 \)
- \( h(k) = 0.618m \) for \( k = 1 \) (\( \phi^{-1} = 1/1.618\ldots = 0.618\ldots \))
- \( h(k) = 0.236m \) for \( k = 2 \)
- \( h(k) = 0.854m \) for \( k = 3 \)
- \( h(k) = 0.472m \) for \( k = 4 \)
- \( h(k) = 0.090m \) for \( k = 5 \)
- \( h(k) = 0.708m \) for \( k = 6 \)
- \( h(k) = 0.326m \) for \( k = 7 \)
- \( h(k) = \ldots \)
- \( h(k) = 0.777m \) for \( k = 32 \)
Fibonacci Hashing

![Graph of Fibonacci Hashing](image)