CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 7

February 25, 2014

HashSet Implementation

- Chapter 20 implements a HashSet<T> by quadratic probing.
- HashSet can be a basis for HashMap with small changes.
- Also, HashSet is a useful collection class by itself.
- HashSet has an array of HashEntry, which is an internal class of hashed object type.
- Class definition:

```
public class HashSet<T> extends
AbstractCollection<T> implements Set<T> { ...
```

HashSet Partial Implementation

```
public class HashSet<AnyType> extends
AbstractCollection<AnyType> implements Set<AnyType>
{
    ...// Class implementation here
    private static final int DEFAULT_TABLE_SIZE = 101;
    private HashEntry [] array;
}
```

- The array is the actual table.
- It is private because it is an internal implementation detail external users shouldn't care about or access.
- The access is handled through API functions – contains, add, remove, findPos etc.

The Inner Data Structures

- HashEntry is an element (set element) plus a flag "isActive", so that entries can be marked deleted.
- The HashEntry class is a private static class of HashSet. Its definition is inside the HashSet class definition:

```
public class HashSet<AnyType> ...  {
   private static class HashEntry {
   ...
   }
}
```

HashEntry Class

```
// This class is defined inside HashSet
private static class HashEntry implements
Serializable
{
    public Object element; // the element
    public boolean isActive;
    // false if marked deleted
    public HashEntry( Object e ) {
        this( e, true );
    }
    public HashEntry( Object e, boolean i ) {
        element = e; isActive = i;
    }
}
```

HashSet Implementation

- The HashSet constructor allocates an array of HashEntry's.
- The **active flag** is needed to mark entries deleted when remove is called, so that probes work properly.
- Thus contains() needs to check that an entry found by findPos is actually active before returning true.
- Similarly, get() needs to check the entry's active-status before returning the value in the entry.

Intro to Inner Classes

- This kind of class with definition inside another class is called an inner class. There are two kinds of inner classes:
  1. static inner classes, AKA nested classes
  2. non-static inner classes, or plain "inner classes"
- The two kinds are quite different.
- Both kinds of inner classes can be used like ordinary classes.
- They can be hidden from client code.
- They are both used mainly for implementation details of the outer object.

Note on Inner Classes

- Static inner classes (nested classes) are very much like an outer class put later in the same file.
- Objects of nested classes have lifetimes independent of the outer class object lifetimes.
- The nesting allows us to make objects that are hidden from the client but not from the outer class code.
- Non-static inner classes are bound to an outer class objects.
- Their creation always involves an outer object, and because of the close association, they know their outer object at any time. They are good for implementing iterators, among other things.

AbstractCollection Class

- HashSet extends AbstractCollection
- This is a class in the Java Collections library that supplies some method implementations based on the general properties of a collection.
- For example, isEmpty() is implemented by calling size() and seeing if the result is 0.
- HashSet uses the implementation provided by AbstractCollection.
- That's fine, nice and fast, no need to re-implement.

AbstractCollection Class

- On the other hand, contains() is implemented in AbstractCollection by searching through the collection using an iterator.
- The whole point of a HashSet is to do a lookup in $O(1)$, so this method definitely needs an *override* in the HashSet class.
- The overriding contains method calls findPos for most of the work, which we expect to be $O(1)$.

Note on Overriding vs. Overloading

- If methods in the base and derived classes have the same signature then the derived-class method overrides the base-class method.
- The overriding method must have the same return type, or a subclass of it.
- Base class AbstractCollection has "public boolean contains (Object x)", and so does the derived class, HashSet, so the subclass version overrides the base class version for uses of the subclass.
- If the methods the base and derived classes have the same name, but different type params, then the derived-class method just overloads the base-class method, i.e. provides a different method with the same name.

Finding the Next Probe

```
int findPos( Hashable x )
{
    int collisionNum = 0;
    // Calculate hash function
    int currentPos = x.hash( array.length );
    while( array[ currentPos ] != null &&
    !array[currentPos].element.equals(x) ) {
        // Compute i^th probe
        currentPos += 2 * ++collisionNum - 1;
        if( currentPos >= array.length )
            // Implement the mod
            currentPos -= array.length;
    }
    return currentPos;
}
```

Another findPos Implementation

```
private int findPos( Object x ) {
    int offset = 1;
    // Calculate hash code and get array entry
    int currentPos = (x==null) ?  0 :
        Math.abs(x.hashCode() % array.length);
    while( array[currentPos] != null ) {
        if( x == null )
        if( array[ currentPos ].element == null ) break;
        // Element is already in the hash table
        else if( x.equals(array[currentPos].element) ) break;
        currentPos += offset; // Compute ith probe
        offset += 2;
        if(currentPos >= array.length) // Implement the mod
            currentPos -= array.length;
    }
    return currentPos;
}
```

HashSet Implementation

- In **findPos**, we see the actual hashing code.
- x.hashCode() is evaluated to get an int that represents the object for lookup, and then a mod and abs is done to get a value in [0, M-1], where here M = array.length.
- The result is currentPos, the actual bucket number. All this is (statistically average) $O(1)$ if we keep the table less than half full.
- The while loop is probing the array until either a match occurs, with x.equals(array[currentPos].element) being true, or the array element is null.
- The probing follows the quadratic recurrence relation.

Next Probe Calculation

Efficient method to calculate the next probe:

$$b_i = b_0 + i^2 (mod M)$$
$$b_{i-1} = b_0 + (i-1)^2 = b_0 + i^2 - 2i + 1 = b_i - 2i + 1 (mod M)$$
$$b_i = b_{i-1} + 2i - 1 (mod M)$$

Next Probe Calculation

- Mod calculation is quite expensive. As a further possible speedup, we note that $(b + 2 * i - 1)$ must be less than $2M$, since $b < M$ and $i < M/2$, so the %M can be done by subtraction:

```
b = b + 2*i -1;
if (b >= M)
    b -= M;
```

Comments

- Note the use of both **x.equals** and **x.hashCode** in this function.
- They must be consistent for hashing to work, that is, if x and y are keys that make equals true, then they must have the same hashCode.
- If contains() is called for an element that has been deleted, findPos will locate the entry marked deleted, and the element there will pass the equals test, so that position will be returned from findPos.
- But contains() checks it out by calling isActive() and returns false, the correct answer.

```
public boolean contains( Object x )
{
    return isActive( array, findPos( x ) );
}
```

Comments

- Note: I'd say to ignore pg. 796, HashMap.java.
- This is a way to build a HashMap on top of a Set implementation via a special class "MapImpl" Weiss has presented back on pg. 749.
- It depends on a special method getMatch() (see pg. 742) he also added to his set implementations, not in the standard Java Collections.
- Let's try to live with the Java Collections API.
- It's not that hard to morph the HashSet implementation to a HashMap implementation – see pa2.

Deletion by Quadratic Probing

- Consider deleting a middle element of a collision chain.
- We need to keep the collision chain working to allow access to elements *after* the deleted one, yet recognize that this one is no longer in the set.
- Example from the book: There we see a collision chain of length 3: key 89 originally hashed to b $= 9$ and claimed slot 9, then key 49 also hashed to 9, but since that slot was full, the next slot, $(9 + 1)\%10 = 0$, wrapping around to 0, was used, so key 49 shows up in slot 0.
- Then key 9 also hashed to 9, but that and the $+1$ spot was in use, so the $+4$ slot was used, and we see 9 in slot $(9+4)\%10 = 3$.

Deletion by Quadratic Probing

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

Deletion by Quadratic Probing

- The size of this HashSet data structure is 4*10 = 40 bytes for the array, plus 5 bytes for each HashEntry object (assuming one byte for boolean), for 5*5 = 25 bytes, and 4 bytes for each of 5 Integers, for 20 bytes, a total of 40+25+20 = 85 bytes.
- Suppose the middle element of the collision chain, key 49 is now deleted.
- It calls findPos, which scans the collision chain looking at keys until it matches, and returns 0, the position. item Then remove changes bucket 0's isActive flag to false. Note that the key 49 is still there! It's marked not-active, of course.

Deletion by Quadratic Probing

- Now look up 9.
- Contains(9) first calls findPos, which scans the collision chain by its key values, accessing the 49 and not matching it, going on to the 9, which does match, and returning 3, the position of the 9.
- Then contains checks isActive for slot 3 and finds it true, and returns true.
- Similarly check contains(49) and find it at slot 0 with findPos, but slot 0's isActive is false, so contains returns false. Correct.
- Obviously for a full check we need to do other cases, like reinserting 49.

Deletion by Quadratic Probing

We see that a hash bucket here has 3 states:

1. **Null HashEntry:** The only case that stops a collision chain scan
2. **non-null HashEntry, isActive = true:** The only case that holds a set element
3. **non-null HashEntry, isActive = false:** Doesn't hold a set element, doesn't stop collision chain scan (because something was there and was deleted. Important to keep the reference not to lose anything further down).

Performance of Separate Chaining

- Assuming a good hashing function:
- **Question:** For N keys and M buckets, what is the average run time for lookup?
  - Don't only answer, explain your calculations.
  - Use what you know about the relationship between M and N.

Rehashing: Resizing the Hash Table

- It is important to keep hash tables less than half full for good performance.
- This can be quite critical for performance.
- With Java, its particularly easy to resize a hash table because you dont have to carefully dismantle the old one – you can point to a new table and just leave the old table to the garbage collector to clean up.

Rehashing

```
private void rehash( )
{
    HashEntry [ ] oldArray = array;
    // Create a new, empty table
    allocateArray( nextPrime( 4 * size( ) ) );
    currentSize = 0;
    occupied = 0;
    // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( isActive( oldArray, i ) )
            add( (AnyType) oldArray[ i ].element );
}
```

Performance of Rehashing

- Typically, we rehash every time the number of entries exceeds 2n, starting at the original half-full point.
- Consider doing a sequence of inserts to an empty hash table of size 64+, prime (M = first prime over 64, that would be 67):
- insert 1, 2, 3, ..., 31 – still less than half full.
- insert 32 – first rehash, to M=128+, prime
- insert 33, 34, 35, ..., 63 – still less than half full ($32 = 2^5$ ops)
- insert 64: second rehash, to M=256+, prime ($64 = 2^6$ ops)

Performance of Rehashing

- In general, during inserts 2n to 2n+1, we do 2n simple inserts, each $O(1)$, plus 2n+1 relocations into a new hash table, each also $O(1)$.
- This totals $O(n)$ inserts.
- If we amortize this over 2n inserts, that is, average the total time over the total number of inserts between rehashes, 2n, we get O(1) per insert.
- Naturally there's a "hiccup" when the expansion happens, and this can be unacceptable if it exceeds some threshold time such as human-perceivable time in interactive programs.
- There are more sophisticated extensible hashing methods that don't hiccup so badly.

Hashing in Memory and on Disk

- The hash table may be located in memory, supporting fast lookup to records on disk, or even on disk, supporting fast access to further disk.
- In fact, a disk-resident hash table that is in frequent use ends up being in memory because of the memory "caching" of disk pages in the file system.

keys	hash table	Data records	Example
memory	memory	memory	typical HashMap apps
memory	memory	disk	use HashMap to hold disk record locations as values
memory	disk	disk	hashed files, some database tables

Hashing Packages in Programming Language libraries

- The C library sticks to basics and has no collection support.
- We've looked at the Java Collections HashMap and HashSet.
- The STL (Standard Template Library) of C++ provides Maps, but their performance (HP implementation) is logarithmic, and thus indicates they are implemented with trees, not hash tables.
- Microsoft's .NET/C# provides collection classes: see `http://msdn.microsoft.com/en-us/library/0ytkdh4s(VS.85).aspx`.