CSE 143

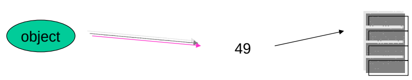
Hashing

Review

- Want to implement Sets of objects
  - Want fast contains( ), add( )
- One strategy: a sorted list
  - OK contains( ): use binary search
  - Slow add( ): have to maintain list in sorted order
- Another strategy: a binary search tree
  - OK contains( ): use binary search through tree
  - OK add( ): use binary search to find right place to insert

A Magical Strategy

- What if... we had a magic method that could *convert each possible element value into its own unique integer?*
  - Takes an element, returns an integer (called a *hash code*)
  - Called a *perfect hash function*
- Then we could store the set elements in an array, with each element stored at an index equal to its hash code



- Array access is constant time – very fast: O(1)
- If computing the hash value is also O(1), lookup is O(1)
  - Beats O(log n), which is the best we've seen so far

Hash Function Example

- Suppose we wanted to hash on a person's last name
- Use the individual characters of the name to compute a number
  - Example: cast each char to its int value, add all the int values
- Use the integer as an index into an array
- Drawbacks?
  - Array would be very large
  - "Soto" and "Soot" hash to the same value
    - Called a "collision"
- There are better string hash functions

If Only We Had A Perfect Hash...

- *A Perfect* hash function is one which has no collisions
  - two different objects never have the same hash code

How fast is contains( )?

- Would just test whether value at the hash location index was non-null
- Fast!

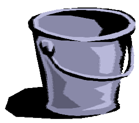
- How fast is add( )?
  - would just set the index to contain the element
  - Fast!

Perfect vs. Imperfect Hash Functions

- *Perfect* hash functions are practical to implement only in limited cases
  - When the set of possible elements is small and known in advance
- But *"imperfect"* hash functions are practical
- An imperfect (or regular) hash function can produce collisions
- Imperfect hash functions compromise the promise of fast performance
  - How?
  - Can we salvage the design?

Solution: Buckets

- Instead of each array position containing the set elements directly...
  - it can contain a *list* of elements that all share the same hash code
  - This list is called a *bucket*
  - Unlike ordinary buckets, this kind can never be full!
- To test whether an element is in the set:
  - Use the hash code to find the correct bucket
  - Search that bucket's list for the element
- Add works similarly



More about Buckets

- If hash function is good, then most elements will be in different buckets, and each bucket will be short
  - Most of the time, contains( ) and add( ) will be fast!
- There will probably be unused buckets – particularly at first
  - No data value happens to hash to a particular bucket
- Tradeoff:
  - more buckets: shorter linked lists, more unused space
  - fewer buckets: longer linked lists, less unused space
- Footnote: This design is *open hashing*; there is a variation called *closed hashing* too.

Object Hash Codes in Java

- Class Object defines a method hashCode( ) which returns a an integer code for an object
- Strives to be different for different objects, but might not always be
  - Generally, you should assume the default hashCode in Java is very imperfect
- Subclasses can override this if a more suitable hash function is appropriate for instances

Hash Codes in Your Own Classes

- Subclasses should override hashCode( ) if a more suitable hash function is appropriate for instances
- Key rule: if o1 and o2 are different objects, then if

  o1.equals(o2) == true

  it must also be true that

  o1.hashCode( ) == o2.hashCode( )
- Corollary: If you override either of hashCode( ) or equals(...) in a class, you probably should override the other one to be consistent
- Danger: The Java system cannot enforce these rules. A well-designed ("proper") class will follow them as a matter of good practice

HashCode for Complex Objects in Java

- Key idea: calculate a hash code value using the fields that are considered in method equals
- Hash codes for individual fields

  Boolean: 0 or 1; int, char: cast to int; float, double, long: get the bits (see ref.)

  Object reference: assuming this field implements equals by recursively calling equals on its parts, call get the hashCode for the fields
- Combining the field hash codes – one possibility

  result = 17;

  for each hash code *c* for some part of the object, set result = 37*result+*c*;

  return result
  - Source: *Effective Java* by Joshua Bloch (A-W, 2001)

    [Great Java book!]

HashMap: Java Library Dictionary Class

- The java.util.HashMap implements a dictionary using a hash table
  - Uses the objects hashCode( ) method to compute bucket #
- Key operations (interface Map)

```
public interface Map {
    // associate the given key with the given value
    public Object put(Object key, Object value);
    // Return the value associated with the key, or null if no such value
    public Object get(Object key);
    // Remove the key and its associated object from the map
    public Object remove(Object key);
}
```

Computing the Bucket Number

- Algorithm:
  - Compute the object's hash code
  - Convert it into a legal index into the buckets array: something in the range 0..buckets.length-1

```
/** Return the index in buckets where the elem would be found, if it's in the set */
private int bucketNum(Object elem) {
    return elem.hashCode( ) % buckets.length;
}
```


Adding a New Element

```
public boolean add(Object elem) {
    int i = bucketNum(elem);
    List bucket = buckets[i];
    if (buckets == null) {
        // this is the first element in this bucket; create the bucket list first
        bucket = new ArrayList();
        buckets[i] = bucket;
    } else {  // return false if elem is already contained in the set
        if (bucket.contains(elem)) { return false; }
    }  // otherwise add element to bucket's list
    bucket.add(elem);
    return true;
}
```

- Note that this (and following) code relies on fact that array elements are null when an array is first created

Checking Whether an Element is In the Set

```
public boolean contains(Object elem) {
    int i = bucketNum(elem);
    List bucket = buckets[i];
    if (bucket == null) {
        // empty bucket
        return false;
    } else {
        // look for element in non-empty bucket
        return bucket.contains(elem);
    }
}
```


How Efficient is HashSet?

- Parameters
  - *n* number of items stored in the HashSet
  - *b* number of buckets
- Load factor: *n*/*b* – ratio of # entries to # buckets
- Cost of contains( ) and add( ) is roughly constant, independent of the size of the set, provided that:
  - Hash function is good – distributes keys evenly throughout buckets
    Ensures that buckets are all about the same size; no really long buckets
  - Load factor is small
    Don't have to search too far in any bucket
- In the average case, the fastest set implementation!
  - In the worst case, the slowest…

Some Issues

- Interesting issues for data structures courses
  - How do you pick a good hash function?
    Needs to be O(1) and produce few duplicates
  - How do you keep the load factor small?
    One answer: Grow the buckets array and rehash all the elements if the load factor gets too large
- Take CSE373 or CSE326 to learn more!

Summary

- Hash functions "guess" the right index to look for an element
  - Can do it faster than binary search can
- If most buckets are short (e.g. <= 3 elements), then works very well
- To keep bucks small, need:
  - good hash functions and
  - the ability to grow the buckets array

Comparing Data Structures

- We now have several implementations of data structures in which we can store and search for objects
  - Array-based lists
  - Linked lists
  - Trees
    - Binary search trees, in particular
  - Hash sets
- Each offers various tradeoffs of performance for common operations
  - Add, remove, contains, iterate (either in random or sequential order)
- Which one is best?

It Depends