CSC 143 Java

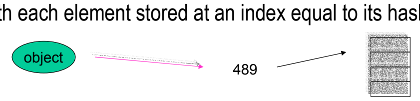
Hashing
Set Implementation via Hashing

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 very fast: O(1)
- An old and still useful idea

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"
- Improved String hash functions can be imagined

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(…)*?

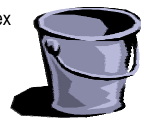
  - would just test whether value at the hash location index was non-null
  - Fast!
- How fast is *add(…)*?
  - would just set the index to contain the element
  - Fast!

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 to implement
- An imperfect hash function allows "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:
  - search the bucket list stored at the hash code index
  - *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!
- Sometimes there will be unused buckets
  - 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 can 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.

HashSet Class

- HashSet: an implementation of Set using hashing

```
public class HashSet implements Set {
    private List[ ] buckets;      // buckets[k] is a list of elements that satisfy
                                  //   elem.hashCode( ) % nBuckets == k
                                  // buckets[k]==null if no elems have hashcode k

    private static final nBuckets =101;     // default # of buckets

    public HashSet( ) {
      buckets = new List[nBuckets];        // each elem initialized to null
    }

    …
```


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 {
        // check if bucket list already contains the element
        if (bucketContains(bucket, elem)) { return false; }        // already there
    }
    bucket.add(elem);           // add the new element
    return true;
}
```


Checking Whether an Element is In the Set

```
public boolean contains(Object elem) {
    int i = bucketNum(elem);
    List bucket = buckets[i];
    if (buckets == null) {
        return false;              // no elements at this position
    } else {
        return bucketContains(bucket, elem);        // search the bucket list
    }
}
```


Searching a Bucket List

```
private boolean bucketContains(List bucket, Object elem) {
    Iterator iter = bucket.iterator();
    while (iter.hasNext()) {
        Object existingElem = iter.next();
        if (elem.equals(existingElem)) {
            // element already present
            return true;
        }
    }
    // element not found
    return false;
}
```


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 table gets 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