CMPSCI 187: Programming With Data Structures

Lecture #34: Ways to Implement a Set
7 December 2011

Ways to Implement a Set

- Review: `SetADT` and the `Set` Interface
- Using Sets: Bingo
- The `ArraySet` Class
- The `LinkedSet` Class
- The `TreeSet` Class
- The HashSet Class
- Comparing the Implementations

Review: `SetADT` and the `Set` Interface

- A set is a collection that may not contain duplicate elements. Two sets are equal if every element of each is an element of the other.
- L&C's `SetADT` interface has methods `add`, `addAll`, `removeRandom`, `remove`, `union`, `contains`, `equals`, `isEmpty`, `size`, `iterator`, and `toString`. Most of these are self-explanatory -- `addAll` adds all the elements of another set to this one, and `union` returns a third set combining this one and another.
- The `removeRandom` method gives you an element at random, with each one equally likely to be the one returned. This will be useful for games.
- The `Set` interface in Collections has no `removeRandom` but has `removeAll` (set subtraction) and `retainAll` (intersection).

Using Sets: Bingo

- In the game of Bingo, each player has a card with a five-by-five array of numbers (except for the middle square which is free). The management draws balls with numbers out of a bowl, and players mark the corresponding numbers on their card if they are there, until some player marks five in a row and wins.

- We can model the bowl as a set. We create an empty `Set<BingoBall>` object, and add a ball for each of the possible numbers (1 through 75). Then each draw of a ball is a `removeRandom` operation.

- We would use a similar method to deal out cards in a card game or assign tiles in Scrabble. (If there are multiple copies of a card, as in pinochle, or a tile, as in Scrabble, we need to distinguish them in some way to keep them in a set.) This is selection **without replacement** -- to model repeated throws of a die, for example, we would need to put the element drawn back into the set.

The `ArraySet` Class

- L&C give a simple implementation of a set using a resizable array, much like that in an `ArrayList`. An element of the set might be anywhere among the non-null entries of the array, which are consecutive at the beginning.

- We have a single static `Random` object for the class, so that each object will get separate values from the random number generator.

- To add a *new* element, we put it in the first null position, after resizing if needed. To remove an element, we replace it with the element in the last non-null position, since it doesn't matter what order they are in.

- To find an element, or do a `contains` operation, we need linear search. To add or remove, we first need to know whether the element is already there.

- To removeRandom, we get a random number from 0 to size - 1 and remove the element in that position, replacing it with the last element as before.

The `LinkedSet` Class

- L&C also give a simple linked implementation of a set, using (singly-linked) `LinearNode` objects. We have a field for the first node as in a linked list.
- To add a new element to the set, we can splice it in at the beginning. But if we don't know that it is new, we must search for it first to avoid duplication.
- To remove an element, even given a pointer to it, means finding its parent which takes O(n) in the worst case.
- Finding an element or testing containment is by linear search, taking O(n).
- The removeRandom is also O(n) in the worst case, because once we decide what number element to remove we have to walk down the list to find it.

The `TreeSet` Class

- Although the set has no order on it defined by the user, if it is from an ordered base type there is no reason we can't keep it in order for our own purposes. We can use a binary search tree, a self-balancing one for best performance.

- If we use the balanced search tree to implement an ordered list, we can find out in O(log n) time whether the element is there. Then we can add it if it is not, or remove it if it is, taking another O(log n) time in the worst case to rebalance the tree (using techniques we have deferred to CMPSCI 311).

- The `removeRandom` operation requires an *indexed* balanced binary search tree if we are going to find the i'th element in the ordering without linear search. This is not hard to do if every node in the search tree knows exactly how many descendants it has -- we can then in O(1) time reduce the search for the i'th element under the parent to the search for the j'th element under one of its children, making the whole search O(log n) with another O(log n) to rebalance after the removal.

The `HashSet` Class

- We've seen how this works -- we need a good `hashcode` method for the base class, then we can place item `x` in bucket `x.hashcode( ) % c`, where `c` is the current capacity of the hash table. The class `java.util.HashSet` uses chaining, so that each bucket is a linked list.

- Adding an element means computing its hashcode, searching its bucket for a duplicate, and inserting it into the bucket if it is new. Removing an element just means removing it from the right bucket, and finding just means doing a linear search of the right bucket.

- Each of these operations takes O(1) plus O(k), where k is the size of the bucket in question. A good hash function will give *average* bucket size O(1), but a bad one could put most elements in large buckets. With a hash function that acts randomly, the largest bucket should have size O(log n).

- The Collections class does not support `removeRandom`. We could implement it using an iterator, but we would need O(n) time in the worst case.

Comparing the Implementations

- Trees are assumed to be self-balancing, and hashing assumes a good function.
- L&C's `ArraySet add` method always checks `contains` and so is O(n).

	ArraySet	LinkedSet	TreeSet	HashSet
add w/o find	O(1) if no resize	O(1)	O(log n)	O(1) sort of
remove w/o find	O(1)	O(n)	O(log n)	O(1) sort of
find/ contains	O(n)	O(n)	O(log n)	O(1) sort of
remove random	O(1)	O(n)	O(log n)	O(n)