Sets, Maps, and Implementation

Henry Z. Lo

June 3, 2014

1 Set and Map ADTs

Sets and maps are abstract data types (ADTs), which means that they are defined by their behaviour, not their implementation. Since run-time depends on implementation, we will not analyse how fast operations are on sets and maps until later.

1.1 Sets

A *set* is defined as a collection of distinct objects. This corresponds to the mathematical definition of a finite set.

Since each element is distinct, this means that there cannot be duplicates. `{"dog","cat","rabbit"}` is a set, but `{"dog","cat","rabbit","cat"}` is not.

Note that the definition says nothing about ordering. This can lead to odd behavior, e.g. creating the same set twice may yield different element orders.

1.2 Maps

A *map* is defined as a set of pairs. The two elements of the pair are usually referred to as *key* and *value*. Keys must be unique.

As a set, maps don't need to define ordering *between* pairs; however, the ordering *within* a pair is important, i.e. the pair `(a,b)` is not equal to `(b,a)`.

Example of a map: `{("name","Charles"), ("job","Cook")}`. This is not a map: `{("name","Charles"), ("job","Cook"), ("name","Judith")}`

2 Interfaces

2.1 Set interface

Remember the Java Collection interface? It defines basic operations required of collections, the important ones being:

- `boolean add(E e)`, which puts `e` into the collection. `E` is the parameterized class, i.e. the type of object the collection contains.

	Map	Collection
Inserting	`put(K key, V value)`	`add(E e)`

Checking	`containsKey(Object k)`
	`containsValue(Object v)`
		`contains(E e)`

Retrieving	`get(Object k)`	not needed
Removing	`remove(Object k)`	`remove(Object o)`

Iterating	`entrySet().iterator()`
	`keySet().iterator()`
	`values().iterator()`
		`iterator()`

Table 1: A comparison of some functions of the map and collection interfaces.

- `boolean contains(E e)`.
- `boolean remove(E e)`.
- `int size()`, which counts the number of elements in the collection.
- `Iterator<E> iterator()`, which returns an iterator over the collection elements.

There are other methods in the interface, but these are the core. The Collection interface already defines everything a set can do. Thus, we can define a Set interface as such:

```
public interface Set<E> extends Collection<E> {
}
```

However, we need to keep in mind that any class that implements `Set` cannot allow duplicate entries.

2.2 Map interface

The map interface in Java shares some commonalities with collection, but are not otherwise related. Table 1 compares the two interfaces side by side.

Even though a map can be thought of as a type of set, in Java they are not related. This is because maps are not merely sets of pairs; they are used in very specific ways, i.e. keys serve as an index, and values serve as the data being stored.

This is evident in the retrieval and removal functions in maps. There is no removal function like this in Collection.

Note also that there are two parameterized classes, `K` and `V` in map, whereas collection has one.

3 Implementations

Both sets and maps can be implemented using hash tables or trees. Both hash tables and trees require an index. In the case of a set, this is the object being

	Hash implementation	Tree implementation
Insert	$O(1)$	$O(\log n)$
Retrieve	$O(1)$	$O(\log n)$
Remove	$O(1)$	$O(\log n)$
Iterate	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n)$
First/last	$O(n)$	$O(\log n)$

Table 2: Complexity of basic operations on TreeSets/TreeMaps and HashSets/HashMaps.

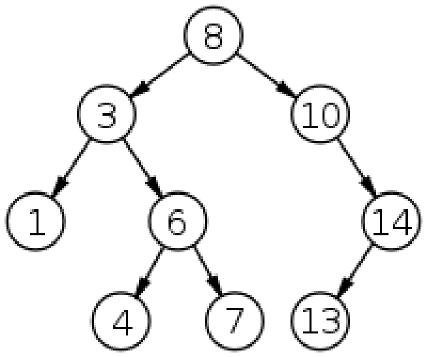

Figure 1: A binary search tree, a common tree implementation.

stored. In the case of a map, it is the key. Otherwise, the way trees and hash tables are used to implement maps and sets are very similar.

3.1 Time complexity

Time complexity of basic operations for both hash and tree implementations is given in Table 2. As discussed in the previous lecture, most basic operations can be done in constant time in hash implementations. You should have learned from CS210 that tree implementations generally take $O(\log n)$ time.

3.2 Ordering

The main benefit of tree implementations is that they sort data. This yields several benefits:

- Getting first and last elements is done simply by finding the leftmost and rightmost child, respectively (see Figure 1).
- Getting elements sorted can be done with a left-to-right depth first search.

For hash tables, there is no inherent ordering - thus, finding ordered elements requires iterating over all elements, and returning a sorted set requires running a sorting algorithm.

3.3 Other considerations

Trees need to be balanced periodically. As hash tables grow, they must be rehashed.

Trees are more space efficient than hash tables, which must contain empty spots to reduce the probability of a collision.

Hashes contain some parameters, such as the initial capacity (how many slots) and load factor (what percentage of elements before rehashing). These may need to be varied in order to optimize performance. Trees don't have such parameters.

For the same data, hash tables may have different orderings; orderings may also change as hash tables get rehashed. In other words, hash tables are less stable. Trees are usually constructed deterministically.

Problem 1. *You are programming software on a wristwatch. You need to store lots of data, but do not have much memory. Which data structure implementation would you use?*

Hash implementations are not memory efficient - they can take up to twice as much space as the number of elements you need to store. Use a tree implementation.

4 HashMap

Let's actually implement HashMap using chaining to resolve collisions.

4.1 Attributes

There only needs to be the data structure, which can be an array (or ArrayList) of LinkedLists of Entries. Each entry is a key / value pair of the generic types `K` and `V`. Let's call the data structure `entries`.

```
ArrayList<LinkedList<Entry<K,V>>> entries;
```

The Entry class should be very bare, containing only the key and value attributes, a constructor, and getter methods.

Technically, if you don't care about storing keys, you can just have an ArrayList of LinkedList of generic types. In this case you would not be able to implement `keySet`, `containsKey`, etc. We continue assuming the first implementation.

4.2 Constructor

The constructor needs to take as a parameter a default size. If we plan to rehash the table when it grows too large, then there also needs to be a load factor threshold.

4.3 Methods

Assume we have a `hash` function, which returns a valid index. For a map we need to implement the following methods:

- `put(K key, V value)`: first get the appropriate list from the key, then add the new Entry object created from key and value.
    - `LinkedList<Entry<K,V>> myList = entries.get(hash(key));`
    - `myList.add(new Entry(key, value));`
- `containsKey(Object k)`: calculate `myList`; use `contains` instead of `add`.
- `containsValue(Object v)`: this requires iterating over the whole table, because we do not hash values.
- `get(Object k)`: calculate `myList`, iterate.
- `remove(Object k)`: calculate `myList`, then use `remove`. Note that this essentially iterates over the list.

4.4 Iterator

For `entrySet()`, `keySet()`, `values()`, you need to return a single iterator. However, if we just call iterator, we will just get an iterator for LinkedLists. We need an iterator for items.

Given an iterator of collections called `inputs`, and an inner iterator `inner`:

```
public boolean hasNext() {
  while (!inner.hasNext() && inputs.hasNext()) {
    inner = inputs.next().iterator();
  }
  return inner.HasNext();
}

public T next() {
  if (!hasNext())
     throw new NoSuchElementException();
  return inner.next();
}
```