# Chapter 11

# Mapping Structures

Some data exhibit a sequential relationship between elements. Other data exhibit a hierarchical relationship between elements. And yet, some data exhibit a *mapping relationship* between elements. For example:

1. A language dictionary maps words to definitions.
2. A bank account database maps account numbers to balances.
3. The Dewey Decimal Classification system maps Dewey Decimals to books.
4. In an American grade school, students learn how to map states to their capitals.
5. An array maps indices to values.

For these data, we have *mapping data structures* that allow us to efficiently query these elements based on their relationships to each other.

## 11.1 An Example: Language Dictionaries

To get an intuitive feel for the sorts of operations we might ask of our mapping types, let's consider a concrete example: *language dictionaries*. What might we do with such a dictionary? Initially, we would start with an *empty* dictionary, so it would be prudent to put entries into it. For example, we might want to *add* the definition for cat: "a small domesticated carnivorous mammal with soft fur, a short snout, and retractile claws". If our definition changes as language is an amorphous beast, we might fix it up by putting a new, *updated* definition of "cat". We should also be able to *remove* the definition of "cat" if the powers-that-be decide that cats are no longer a thing. Finally, after adding a number of definitions, it makes sense to ask for the *size* of the dictionary which is the number of entries it contains.

The most important operation we can perform on a dictionary is *lookup* a word for its corresponding definition—alternatively get the definition for a given word. When doing so, we'd like to avoid searching the whole dictionary, *e.g.*, by organizing the dictionary in lexicographical order of the words, we can quickly find where "cat" sits in the dictionary. Our dictionary structure ought to support fast lookup of these words. Note that if the word is not in the dictionary, then we should signal an error that the word was not found. Correspondingly, we would like to be able to check to see if our dictionary *contains* an entry for the word before we try to perform a lookup.

Finally, we can think of our dictionary as a collection of entries where each entry is a pair of a word and its definition. But alternatively, we can think of it as two separate collections of words and definitions. We may want to get this collection of words and collection of definitions separately to analyze them.

## 11.2 The Map ADT

Each of the operations we'd like to perform on a language dictionary translates directly into a corresponding operation on the map ADT:

Let's turn our intuition of how a language dictionary works into a general abstract data type capturing the essence of a mapping data structure. The map abstract data type captures a mapping from keys of one type `K` to values of another type `V`. We'll call each key-value pair, `(k, v)`, an *entry* in the map. In our concrete example above, our keys are words (strings) and our values are definitions (strings). Note that the type of keys and values can be distinct. For example, an account number might be a string, but the value that an account number maps to, the balance, is an integer.

- `void put(K k, V v)`: put an entry for key `k` (of type `K`) in the map, associating it with value `v` (of type `K`). If an entry for `k` already exists in the map, we overwrite the old value with this value `v`.
- `V remove(K k)`: removes the entry for key `k` from the map if it exists, returning its corresponding value.
- `int size()`: returns the number of entries in the map.
- `boolean containsKey(K k)`: returns `true` if the map contains an entry for key `k`.
- `V get(K k)`: returns the value `v` associated with the key `k`; throws an error if `k` is not present in the map.
- `List<K> keys()`: returns a list of the keys of this map.
- `List<V> values()`: returns a list of the values of this map.

In Java, we might define the following interface to capture these operations:

```
public interface Map<K, V> {
    public void put(K k, V v);
    public V remove(K k);
    public int size();
    public boolean containsKey(K k);
    public V get(K k);
    public List<K> keys();
    public List<V> values();
}
```

The Java standard library defines an interface `java.util.Map` that captures this abstract data type. It contains these operations (with some slight differences in signatures of methods) along with other methods.

## 11.3 Association Lists

How might we implement this abstract data type? One way to do it is to realize our map as a list of key-value pairs. Assuming that we have a way of representing pairs of data, for example, a `Pair` class:

```
public class Pair<T, U> {
    public T fst;
    public U snd;
    public Pair(T fst, U snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```

We can implement the map abstract data type by using a list of pairs. Each pair `(k, v)` is an entry in the map where the first component is a key and the second component is its corresponding value.

Let's consider an example of using an association list to get a feel for how we would implement these operations. The empty map is represented by the empty list:

```
[]
```

Let's consider creating a map from strings to integers, where the integer is the string's size. First, let's add `("dog", 5)` to map. This amounts to adding the pair to the list:

```
[("dog", 5)]
```

When we insert new key-value pairs, we simply continue to add pairs to the list in-order:

```
[("dog", 5), ("doghouse", 8), ("cat", 3)]
```

However, in general, if we add a key that already exists, we must first remove the old key-value pair and then re-add it with the new value. For example, consider fixing our entry for "dog" so that it is correct:

```
    [("doghouse" 8), ("cat", 3)]
--> [("doghouse" 8), ("cat", 3), ("dog", 3)]
```

In general, when inserting into an association list, we must see if the key already exists in the map. If so, we delete its corresponding key-value pair, and then we perform the addition like normal.

The remainder of the operations are straightforward to implement with association lists. The size of the map corresponding to the size of the list, *e.g.*, there are currently three entries in the map above. To see if a list contains a given key `k`, we must perform a traversal of the list to see if one of the pairs has key `k` as its first component. Similarly, if we want to get the value for key `k`, we perform a similar traversal, returning the corresponding value. Deletion is like `get` except we remove the key-value pair from the list. Finally, generating a list of only the keys or only the values also requires a traversal of the list, adding either the keys or values to a new list to be returned.

Note that the description so far hasn't taken the sortedness of the keys into account. We might choose to sort the key-value pairs by the key so that we can perform more efficient lookup, *e.g.*, by using binary search. However, maintaining this sortedness on every insertion makes the insertion operation less efficient and complicates our implementation. For now, we'll choose to leave the keys unsorted when talking about the complexity of association lists, leaving sorting to our BST-based implementation that we discuss next.

The complexity of all the map operations using association lists—because the structure is list-based—correspond precisely to the complexity of the structure's underlying list operations. For example, to put a key-value pair into the association list, we must check to see if `k` is already in the list. In the worst case, we must then remove that old key-value pair in the list and then add the new key-value pair to the list. The check takes

$\mathcal{O}(n)$ time (where $n$ is the number of entries in the map) and the removal takes $\mathcal{O}(n)$ time—both are linear traversals of the underlying list. The addition is either $\mathcal{O}(1)$ or $\mathcal{O}(n)$ depending on our underlying list data structure and where we add onto the beginning of the list or its end. Note that it does not matter where the key-value entry goes. Overall, this means that the time complexity of `put` is the sum of complexity of these three operations, or $\mathcal{O}(n)$. The complexity of all the other map operations is also linear, $\mathcal{O}(n)$, because they all require linear traversals of the underlying list.

## 11.4 Tree Maps

Instead of a list-like structure, we can require that our keys be comparable. By doing this, we can sort our map "by key" to perform more efficient lookup. However, we must maintain the sortedness of our map on every insertion. A binary search tree allows us to do this easily, so we can alternatively implement a map using a binary search tree, typically called a *tree map*.

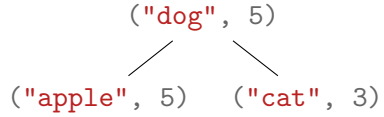
The elements of a binary search tree are not just single values, but key-value pairs like with association lists. By using a binary search tree, we require that the keys are comparable so that we can use them to determine the placement of key-value pairs into the BST. Using the example above, we can compare strings by considering their lexicographical ordering. For example:

```
"apple" < "banana" -- "a" appears before "b", "a" < "b", in the alphabet
"acorn" < "apple"  -- the 1st letters are the same, but "c" < "p" (2nd letters)
"dog"   < "dogs"   -- the 1st three characters are the same, but "dogs" is longer
                      (the empty letter "comes before" any other letter)
```

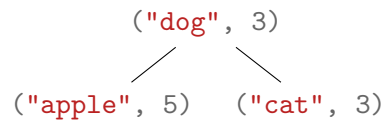
We compare both words, letter by letter. If the letters are the same, we move onto the next letter. If they are different, we declare the "smaller" word to be the one that has the "smaller" letter where "smaller" for letters is defined by their underlying character values (which coincides with alphabet order in the case the characters are letters). The `String` class implements the `Comparable` interface so that strings can be compared in this manner.

We begin with the empty tree. Inserting the first entry `("dog", 5)` gives us the following tree:


After inserting `("apple", 5)` and `("cat", 3)`, we get the following structure (because `"apple" < "dog"` and `"dog" < "cat"`):




Note that deleting from the BST is undesirable because it will cause the tree to become more unbalanced. So, to update a key's entry, we simply search for that key in the tree and if we find it, we update the corresponding entry:




Otherwise, like the association list, the remaining operations on maps are implemented in terms of the corresponding operations on BSTs. In particular, when we generate the list of keys of the map, if we traverse the BST in-order to generate the list, we receive a sorted list of the keys which may be convenient for a number of applications.

The complexity of the tree map operations comes directly from the complexity of underlying BST operations used to implement them. Critically, unlike the association list which has $\mathcal{O}(n)$ insert and lookup, the BST has $\mathcal{O}(n)$ time complexity (for balanced trees). In practice, the BST is implemented with one of the balanced tree structures discussed briefly earlier—AVL trees, red-black trees, or B-trees. The `java.util.TreeMap` class implements the `Map` interface in Java's standard library and uses a red-black tree structure to maintain balancing.

## 11.5 Integer Maps

Consider a special case of our map data structure where the keys are integers. For example, our map may be recording an inventory where we are mapping product identification numbers (pids) to the count of such items in the inventory. Let's make the following assumptions:

- The range of valid pids is finite and known, *e.g.*, say we only anticipate at most 1000 products in our stock, so our pids range from 0–999.
- The pids are unique, that is, any two products have distinct pids.

With these two assumptions, we can use an array to implement our map. Our keys, here pids, serve as indices into the array. We say that the *i*th element of the array contains the count of the item with pid *i*. Because we know the range of pids is 0–999, it suffices to allocate an array of size 1000 (with indices 0–999) to store these key value pairs.

For example, consider an initially empty inventory. Our backing array would consist of an array of 1000 cells, each containing zero. Note that we obtain this behavior by simply initializing the array as the "zero" initialization value for an int is 0.

```
[0, 0, 0, 0, 0, ...]
```

Next, let's add some values to our inventory. For example, suppose we have five copies of an item with pid 2 in stock. Then we can update the second index in the array:

```
[0, 0, 5, 0, 0, ...]
```

If we want to look up the number of copies of the item with pid 2, we simply look at index 2 in the array. We can update another item, *e.g.*, say the item with pid 1 has 10 copies in stock:

```
[0, 10, 5, 0, 0, ...]
```

If we want to update one of items, *e.g.*, we sold two copies of the item with pid 2, we simply update the corresponding position in the array:

```
[0, 10, 2, 0, 0, ...]
```

With the setup, our key-map operations are very easy to implement:

- `lookup`: a single array read at index `k`—$\mathcal{O}(1)$ time.
- `put(k, v)`: a single array write at index `k` with value `v`—$\mathcal{O}(1)$ time.

Because we rely on array indexing, we gain extremely efficient run times for `lookup` and `put`. However, we encounter a slight technical hiccup with `size`. Technically in the above inventory, even though we haven't explicitly put a count for the item pid 0, it has the default value zero. This is likely correct given our interpretation of the data structure—if we have not updated the map for a particular item, then we do not have any of that item in stock. Thus we could consider the size of the map to be 1000, corresponding to the 1000 pids that we are accurately capturing with this map.

In general, though, this may not be the case. For example, we may map a pid to its manufacturer which we represent as a string. The default value for string (because it is an object), is null, so our initial array looks as follows:

```
[null, null, null, null, ...]
```

Updating a pid, say 2, with its manufacturer updates the backing array as follows:

```
[null, null, "J&J Produce", null, ...]
```

The key thing to note is that the entries for pid = 0, 1, 3, ..., do not contain valid entries. We can get around this by assigning a sentinel value in the range of the map, *i.e.*, setting aside a value to be the "there no value" value. For example, in the above map, `null` could serve as the "no entry" value for a given key. However, we know that this doesn't work in all cases, *e.g.*, if it is possible to use all the values of a given type in the map.

To get around this, we can use the Java 8 `Optional` type which encodes whether we have a value of type `T`. You can think of an `Optional` as a 1-element cell that is either empty or non-empty. For example, our array above would contain:

```
[Empty, Empty, Optional("J&J Produce"), Empty, ...]
```

Where `Empty` corresponds to an empty optional value (its `isPresent()` method returns false), and `Optional(v)` corresponds to an optional value that contains some underlying value (`isPresent()` returns `true` and `get()` produces that value).

## 11.6 The Set ADT

A mathematical set is a collection of elements without duplicates. The set abstract data type is the realization of this concept in a program—it, too, is a collection of elements without duplicates. For example, we may start with an empty set of numbers:

$$\{\}$$

We can add elements to this set:

$$\{1, 4, 2, 9, 8\}$$

Trying to add a duplicate, *e.g.*, 4 to this set does not change the set because 4 is already in it.

A set acts similarly to a list in that we can add and remove elements as well as iterate over its contents. Unlike a list, a set is not a sequence so the elements do not have indices that we can refer to them by. Importantly, a set allows us to query for elements more efficiently than with a list—we ought to be able to check to see if an element is contained by the set in better-than-$\mathcal{O}(n)$ time.

In addition, we'd like to perform similar operations over our set data type that we would perform over a mathematical set. In particular, we ought to be able to take the union of two sets. The union of two sets (written $A \cup B$ in formal mathematical notation) is simply the combination of all the elements from both sets, respecting duplicates:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\} = \{1, 3, 4, 2, 9, 8, 6\}$$

The intersection of two sets (written $A \cap B$ in formal mathematical notation) is the set of elements found in both sets:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\} = \{4, 2\}$$

We can summarize these operations as follows:

- `void add(T v)`: adds `v` (of type `T`) to set—does nothing if `v` is already in the set.
- `boolean remove(T v)`: removes `v` from the set, returning `true` if the removal succeeds, and `false` otherwise.
- `boolean contains(T v)`: returns `true` if `v` is in the set and `false` otherwise.
- `int size()`: returns the size of the set.
- `Set<T> union(Set<T> s)`: returns a new set that is the result of taking the union of this set with the other set `s`.
- `Set<T> intersect(Set<T> s)`: returns a new set that is the result of taking the intersection of this set with the other set `s`.

Sets are seemingly unrelated to the mapping structures discussed previously. However, note that the "no-duplicates" rule of a set is precisely the restriction that map places on its keys—a map may only have one entry per key. Therefore, we can implement a set in terms of a map: a set is simply a map where the values don't matter!

To capture this idea that the "values don't matter", in Java, we may define a type `Unit` that does nothing and acts as a placeholder value:

```
public class Unit {
    private Unit() { }
    public static final Unit value = new Unit();
}
```

Because `Unit` has no fields or behavior, every value of unit is equivalent to itself. So rather than creating many such empty `Unit` values (via `new`), we hide the constructor to `Unit` by marking it `private` and then provide a single `Unit` value to clients of the class via a static field. This pattern of providing a single instance of a class while disallowing others from creating instances of the class is called the *Singleton Pattern*.

With our `Unit` type in place, we can define a set over type `T` to simply be a map from `T` to `Unit`. Adding an element `v` into the set is equivalent to putting the key `(v, Unit.value)` into the underlying map. All of the other basic set operations act similarly to their map counterparts. The complexity of these operations depend on the complexity of the underlying set. In particular, Java provides the `Set` interface and the

`TreeSet` class which is a red-black tree implementation of the interface. Tree sets (really, tree maps) provide $\mathcal{O}(\log n)$ for insertion, removal, and contains checks.

Union and intersection require a little bit more work. We can implement union by repeated insertions. For example, in the below situation:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\}$$

We can simply create a new, empty set and then insert each of the elements from the left-hand and right-hand sets into this new set. Of the $n$ elements, each insert takes $\mathcal{O}(\log n)$ time (again, assuming that the tree stays balanced) for an overall runtime of $\mathcal{O}(n \log n)$. We can do a similar thing with intersection, but we only insert an element if both sets contain the given element. This approach requires three $\mathcal{O}(\log n)$ operations per element—two contains checks and an insertion—which results in an overall runtime of $\mathcal{O}(n \log n)$.