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 int-ition 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 corre-
sponding 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:

```java
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:

```java
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 \ Donetsk
```dog
``` to map. This amounts to adding the pair to the list:

```
[Donetsk
```dog
``` , 5]
```

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

```
[Donetsk
```dog
``` , 5], (Donetsk
```doghouse
``` , 8), (Donetsk
```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:

```
[Donetsk
```doghouse
``` , 8], (Donetsk
```cat
``` , 3]
```

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

119
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 \texttt{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 \( O(n) \) time (where \( n \) is the number of entries in the map) and the removal takes \( O(n) \) time—both are linear traversals of the underlying list. The addition is either \( O(1) \) or \( O(n) \) 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 \texttt{put} is the sum of complexity of these three operations, or \( O(n) \). The complexity of all the other map operations is also linear, \( O(n) \), 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 \textit{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:

```
("dog", 5)
```

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

```
("dog", 5)
   /   \
("apple", 5)  ("cat", 3)
```

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:

```
("dog", 3)
   /   \
("apple", 5)  ("cat", 3)
```

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 $O(n)$ insert and lookup, the BST has $O(n)$ 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 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.

\[0, 0, 0, 0, 0, \ldots\]

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:

\[0, 0, 5, 0, 0, \ldots\]

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:

\[0, 10, 5, 0, 0, \ldots\]

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:

\[0, 10, 2, 0, 0, \ldots\]

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

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

\[null, null, null, null, \ldots\]

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

\[null, null, "J&J Produce", null, \ldots\]
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-\(O(n)\) 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} \cap \{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.
11.6. THE SET ADT

- **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 is 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 class Unit that does nothing and acts as a placeholder value:

```java
public class Unit {
    private Unit() { }
    public static final Unit instance = 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.instance) 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 $O(\log n)$ 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 $O(\log n)$ time (again, assuming that the tree stays balanced) for an overall runtime of $O(n \log n)$. 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 $O(\log n)$ operations per element—two contains checks and an insertion—which results in an overall runtime of $O(n \log n)$. 
Chapter 12

Hashing

So far, we have explored a pair of implementations for our map ADT. Association lists gave us \( O(N) \) lookup, and tree maps gave us \( O(\log N) \) lookup. Can we do better than this? Recall that an integer map is a special case of a map where the keys are drawn from a finite set of integers. In this situation, we can use an array to efficiently implement the map ADT where keys are indices and values are elements of the array.

In this reading, we’ll breaking apart the assumption of the integer map to generalize the structure to handle a wider variety of domains. By doing so, we’ll derive a data structure called a hash map which is a very efficient implementation of the map ADT. Along the way, we’ll also develop a technique called hashing which is one of the most important techniques in computer science with applications towards fingerprinting, compression, and cryptography, among others.

12.1 Revisiting Integer Maps

The first assumption we made when using an integer map was that the space of keys was finite and known. Recall that our running example of an integer map is an inventory where we map product IDs (pids) to inventory sizes. For this example, we assumed that our pids were drawn from the range 0–999. What happens if we don’t know what values the space of keys ranges over? For example, we may want to support a number of products beyond 1000.

This seems to pose a problem for our integer map strategy. By definition, an array can only hold a fixed number of elements. With a variable number of possible keys, it seems like we cannot use an array to hold them. This seems like a job for an array list which can hold a variable number of elements, but a deeper problem remains. Consider the following map where we only hold an entry for key value 10000 (and we use null as our value indicating that this key has no entry in our map):

\[
[\text{null, null, null,} \ldots, \text{v}]
\]

\[\text{^ index 10000}\]

Because we map keys directly onto indices, our array list would need to contain entries for 0–9999 even though only one entry is in the map for key 10000. This is an extreme waste of
space—the size of the backing array or array list must be as large as the largest key that we need to support. However, if the space of keys that the map actually needs to store is sparse, i.e., very few keys are required from the range, then we end up wasting lots of space. Ideally, we’d like to avoid using a variable-size structure and instead use a fixed-size structure so that our space usage is not proportional to the size of the key space that we might support (which is very large in practice).

How can we support a variable number of keys using a fixed-sized array? One trick we can try is using the modulo operator to fit the keys into the available space of the array. Suppose that we only allocate an array of size 10 to hold the entries of our map. Then we can simply mod our key by 10 to get an index that fits in the array, e.g.,

- The values 0, 10, 20, 30, . . ., all map to index 0.
- The values 1, 11, 21, 31, . . ., all map to index 1.
- ...
- The values 9, 19, 29, 39, . . ., all map to index 9.

If our map contains entries for keys 232, 11196, 555, and 8, then our backing array would look like this:

[null, null, v1, null, null, v2, v3, null, v4, null]

Where v1 is the value associated with key 232, v2 is the value associated with key 555, v3 is the value associated with key 11196, and v4 is the value associated with key 8. Accessing the entry for key k means that we simply look up index k % 10 in the array.

Of course, the issue here is that if also want to store an entry for key 1992, we have a problem because there is an entry in index 2 already for key 232. This is known as a collision—two keys want to use the same position in the array. There are two primary methods of resolving collisions: probing and chaining.

### 12.1.1 Probing

When a key’s preferred index has already been taken by another key, we can probe the array for another position (presumably empty) to store the key. Probing means searching the array in a systematic manner starting from a key’s preferred index to find an open position to store its value. A simple probing strategy is to simply search successive elements in the array until we find an empty spot, a technique called linear probing.

To do this, we must augment what our arrays carry as values. Rather than carrying the values of our map directly, each array index will carry both a key and its corresponding value. This is because with our probing strategy, a cell may no longer correspond to a key’s preferred index—some other key might have already taken that spot. By carrying the key, we can ensure that we found the appropriate value by comparing the key we found in the array to our target key.

In the example above, consider starting with an array of five elements. For simplicity’s sake, we’ll just note the key in our diagram, but keep in mind that we are storing an entire key-value pair in each array slot.
Suppose that we are adding the key $k_6$ that has a value of 2. The key $k_1$ is already in that slot, so we search to the right until we find an empty position and place $k_6$ there.

When we go to look up $k_6$ in this map, we start at index 2 and scan to the right until we find the entry for $k_6$. Now, what happens if we add a key $k_7$ that has id 8? Index 8 in the array is already taken by $k_4$, so we search index 9. However, that is taken by $k_5$. We proceed by wrapping around the array and starting our first at the index 0. This index is empty, so we place $k_7$ there:

In general, during put and lookup, our linear search wraps around the array, terminating when we reach the preferred index of the key in question.

Using a probing scheme, we achieve $O(1)$ complexity for our fundamental map operations—put and lookup—as long as there are no collisions with our keys. However, if there are collisions, we’ll need to search part of the array to find our key. In the worst case, all keys have the same preferred index, so we’ll need to perform a linear scan of the array which takes $O(n)$ time. Note that discussing the "average case" here is very difficult because it depends on the likelihood of collisions with our keys which is dependent entirely on the specific keys we add into the map.

### 12.1.2 Chaining

With probing, we maximize the space in our backing array by checking successive indices for open positions. When our map is sparse, this is fine because there’s plenty of empty spaces between elements for collisions to be placed. However, if the map is dense, i.e., contains many elements, then probing will need to traverse significant portions of the array to find the key of interest. Rather than doing this, we can elect to store all of the keys of a particular id at their associated index. To do this, we store a list of key-value pairs at each index of the array rather than a single such pair. For example, we may have the following map:

```
[ ][ ][ ][ ][ ][ ]
| | | | |--> [k7]
| | | --> [k5, k6]
| --> []
--> [k4]
```

Keys $k_1$, $k_2$, and $k_3$ all have id 0, so they map onto index 0 in the array. Index 0 contains a list that holds these three key-value pairs. In contrast, there are no stored keys with id 2, so index 2 holds an empty list.

This method of collision resolution is called chaining, named as such because these lists look like chains hung off of each array index. Alternatively, we can call each index a bucket and the
effect of chaining is to store all the key-value pairs of a certain id in their corresponding bucket. In essence, these lists function like the association lists we studied earlier except that all the keys contained in a list have the same id.

By doing this, we no longer have to perform a linear scan of the array. Instead, we perform a linear scan of a bucket’s association list. A bucket fills up only with colliding keys, so the amount of keys that we need to traverse is proportional to the number of keys that collide for a particular bucket. In the best case, a bucket contains exactly one key-value pair corresponding to the desired key (in the absence of collisions) for $O(1)$ lookup. In the worst case, everything has the same key, so a single bucket contains all the entries in the map. In this situation, lookup takes $O(n)$ time. Like probing the average case is dependent on the nature of the collisions of the keys which depends on the specific keys stored in the map.

At first glance, it seems like we do not have resizing issues with chaining like we do with probing. With lists holding each bucket, we never run out of room for the keys. However, because our arrays have finite size, if we store many more keys than we have buckets, the pigeonhole principle tells us we will have collisions. In other words, eventually our map will look as follows:

```
[ ][ ][ ][ ][ ]
| | | | |---> [ ... ]
| | | |---> [ ... ]
| | |---> [ ... ]
| |---> [ ... ]
|->[ ... ]
```

where each of the buckets contain many keys. In this situation, it is prudent to resize the array, creating more buckets and subsequently more opportunities to spread out keys among the different buckets.

Like probing, we simply cannot create an array of double the size and copy over the elements blindly—we will not respect the new preferred indices of the keys if we do this. Instead, we must create an array with double the size and then rehash the key-value pairs back into the newly created array.

When we choose to resize the array with chaining is different than probing. With probing, we can simply resize the array when it is full. With chaining, we never truly “run out of space”; instead, we must choose an appropriate load factor that consists of the ratio of the size of the number of entries $n$ to the number of buckets $k$: $n/k$. A load factor of 0 indicates that the map is empty. In contrast, a load factor of 1 indicates the map is “full” in the sense that adding any more elements guarantees a collision. With a low load factor, the backing array is sparsely populated meaning that we are likely wasting space with empty array indices without positively affecting our lookup times. A high load factor means that there will likely be many collisions pushing our lookup time to $O(N)$.

12.1.3 Removal

One final operation that we have yet to discuss is removal of key-value pairs. In the case of chaining, removal amounts to removing a key-value pair from an association list which is easy to
do. Like an array list, we can choose to leave the backing array sparsely populated or contract it to increase its load factor, saving space in the process.

With probing, removal becomes a more nuanced affair. Consider the following array:

\[ [k1, k2, k3, k4] \]

Where \( k2 \) and \( k3 \) both have id 1, and suppose we now remove \( k2 \). Because an array must contain a value in each of its indices, we can elect to null out index 1.

\[ [k1, \text{null}, k3, k4] \]

However, what happens when we lookup \( k3 \) now? We’ll start at its preferred index, 1, and note that no element is there. We could pass over index 1 and find \( k3 \) at index 2. But now consider this scenario where the map is empty:

\[ [\text{null}, \text{null}, \text{null}, \text{null}] \]

Here, when we look up at index 1, we note that no element is there. Rather than skipping over to the next element, we want to terminate the search right now so we don’t spend \( \mathcal{O}(n) \) time trying to find an element in this empty map.

Thus, we need to differentiate between an empty index due to a deletion and an empty index due to no key-value pairs being added there yet. In the former case, we want to move onto the next index while performing lookup; in the latter case, we do not. To obtain this behavior in Java, we may use the Java 8 `Optional<T>` class. Instances of `Optional<T>` either contain a value of type T or nothing. We can use this class as follows:

- A `null` value in the array means there is no corresponding entry in the map.
- An empty `Optional<T>` value corresponds to a deleted value.
- A non-empty `Optional<T>` value corresponds to an actual entry in the map.

### 12.2 Transforming Values to Integers

Finally, we need to lift the restriction on our integer maps that our keys must be values. To do this, we develop the fundamental idea of hashing, transforming an arbitrary value into an integer. The resulting data structure that we obtain, the hash map, gives us the constant-time map operations that were seeking over any data type for which we can write a hash function.

#### 12.2.1 The Hash Function

A hash function is simply a function \( h : T \rightarrow \text{int} \) that converts a value of type \( T \) into an integer with some restrictions. To see what these are, let’s consider writing a hash function for type char (so \( T = \text{char} \)). Here is a simple example of a hash function:

\[ h(c) = 0 \]

which transforms every character \( c \) into the integer 0. This is not a very good example of a hash function because it maps every character into the same index! For example, if we used a chaining
scheme and our keys were characters, using this hash function to convert characters into indices nets us the following structure:

```
[ ][ ][ ][ ][ ][ ]
| --> [c1, c2, c3, ...]
```

Where all the keys sit in the zero bucket! We say that this hash function does not distribute its keys among the space of possible indices effectively.

On the other hand, the following hash function:

\[ h(c) = \text{rand}(0, 100) \]

effectively distributes its keys—it indeed chooses a random index in the range 0 through 100 on each invocation. However, the resulting hash value is not consistent meaning that two separate calls to \( h(c) \) will likely result in two different indices. This makes lookup with such a hash function impossible!

In summary, we need a hash function that:

1. Distributes its keys as evenly as possible among the space of possible integers.
2. Consistently assigns keys so that if \( k_1 = k_2 \) then \( h(k_1) = h(k_2) \).

Note that the second requirement says that equal keys produce equal hash values. The converse is not necessarily true! A valid hash function may have \( h(k_1) = h(k_2) \) but \( h_1 \neq h_2 \). This is precisely a key collision that we discussed previously and have methods for resolving after-the-fact. The first hash function we examined—the constant hash function—is valid, but it produces too many collisions.

For characters in Java, the hash function is quite simple: take the character value of the char!

```java
public static int hashChar(char ch) {
    return (int) ch;
}
```

Note that the way that we represent characters in a computer program is by assigning a unique integer to every character (known as the Unicode character encoding standard). Thus, we can simply use this integer as an index into our backing array for keys that are characters.

The function above is certainly consistent as it always returns the same value for, e.g., ’a’. It is very well distributed because by definition of the Unicode standard, every character has a unique integer value. We call such a hash function that provably never produces any collisions between elements a perfect hash function.

### 12.2.2 Hash Functions for Primitives

We can apply similar logic to the different primitive types of Java in order to obtain hash functions for each. Recall that the primitives types in Java are:

- int
• char
• boolean
• float
• long
• double

For integers, the hash function can simply be the identity function:

$$h_{\text{int}}(n) = n$$

As discussed, the hash function for characters simply takes their character value as the hash:

$$h_{\text{char}}(c) = (\text{int}) c$$

For booleans, there are only two possible values—true and false—we can simply assign two integers to them.

$$h_{\text{bool}}(b) = \text{if } b \text{ then } 1 \text{ else } 0$$

Floats are trickier to hash. Recall that floats and doubles are floating-point numbers—decimals of finite length—represented using the IEEE floating point standard. As a first attempt, we might try to simply truncate the float to an integer via a cast. However, for certain sets of keys, this is a very bad hash function. For example, if all our keys are drawn from the rationals in the range \((0, 1.0)\), then this function will hash them all of them to the zero index in our map.

A better hash function comes from the realization that a float is the same size as an integer—32 bits. Because of this, there is a one-to-one correspondence between floats and integers (although it is not a natural correspondence because a float is represented differently than an integer). In Java we can take advantage of this correspondence by using the static method \text{Float.floatToIntBits} to convert a float into an integer by reinterpreting the 32 bits of the float as an integer. For the float 3.14 this yields the integer 1078523331. In contrast, if we simply truncate-cast the float to an int, we receive 3.

Like char, this hash function for floats is also perfect because it maps every possible float to a unique integer. In contrast, we cannot do the same thing for longs. This is because a long is 64 bits whereas an int is 32 bits. Because there are many more longs than ints, there will be collisions with any hash function that we can design.

There are a variety of possible hash functions we can use, e.g.,

- \(h_1(f) = (\text{int}) (f \text{ \{\text{int}\}_\text{MAX\_INT}}): \text{mod the long by the maximum integer.}\)
- \(h_2(f) = (\text{int}) (f \gg 32): \text{use the most significant 32 bits of the long.}\)
- \(h_3(f) = (\text{int}) (f << 32 \gg 32): \text{use the least significant 32 bits of the long—shift the least significant bits to the top, then shift back.}\)

Each of these hash functions are consistent, so they are valid hash functions. However, there are potential problems with each:

- With \(h_1\), the modulo operator is a costly operation which we would like to avoid with our hash function if possible.
• With \( h_2 \), Shifting is cheap, however, we ignore the bottom 32 bits of the \texttt{long}. If all our keys only differed in the bottom 32 bits, then this hash function would send every key to the same hash value, zero.

• Likewise with \( h_3 \), if the all our keys differed in the top 32 bits, then this hash function would also send every key to the same hash value.

To avoid these difficulties, Joshua Bloch in his book, \textit{Effective Java}, recommends the following expression for its hash function:

\[
h(f) = (\text{int} \ (f \ ^\wedge \ (f \gg 32)))
\]

This is a variation of using the most significant bits of the \texttt{long} that takes into account the least significant bits as well. It does this by performing a bitwise or with the top 32 bits (the result of the shift) and the bottom 32 bits. This is also what the Java standard library performs for its hash function for \texttt{long}s. Again, note that this is not a perfect hash function, but it performs well in practice.

Finally, to hash a \texttt{double}, we can simply combine our strategies for a \texttt{float} and \texttt{long}:

1. Convert the \texttt{double} to a \texttt{long} by reinterpreting its 64 bits (using \texttt{Double.doubleToLongBits}).
2. Perform the \texttt{long} hash function described above to produce an \texttt{int}.

### 12.2.3 Hash Functions for Objects

Of course, we are unlikely to need to only produce hash values for primitives. More likely, we will need to hash object values of a particular class type that we have designed that has arbitrary fields. In Java, the \texttt{Object} class provides a method:

```java
/** @return a hash value appropriate for this object */
public int hashCode() { /* ... */ }
```

That any class can override to provide appropriate hashing behavior for their instances. On top of requiring that an implementor’s \texttt{hashCode} function must return an integer, Java requires several other properties of such an implementation:

• Like the hash functions discussed previously, \texttt{hashCode} must be \textit{consistent}—it must report the same hash value for any particular object during the execution of the program.

• If two objects are deemed equal (via the \texttt{equals()} method), then their hash values must be identical.

Like hash functions, \texttt{hashCode} is free to return the same value for un-equal objects—this is a hash collision and ideally avoided as much as possible.

The default implementation of \texttt{hashCode()} on most versions of Java returns the \textit{memory address} of the object. Note that this is a unique representation of the object as only one object can sit at that address in memory. This is also the default implementation of \texttt{equals()}—using \textit{object identity} as the definition of equality. However, we frequently want to override the definition of equality as \textit{structural} over the fields of the object. Likewise, to meet condition (2) described above, we must implement \texttt{hashCode} to perform similarly. This is an important enough rule to codify below:
If a subclass overrides the equals method, then it must also override the hashCode as well.

As an example, consider the humble Point class:

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public boolean equals(Object o) {
        if (o instanceof Point) {
            Point rhs = (Point) o;
            return x == rhs.x && y == rhs.y;
        }
    }
}
```

Equality over points is defined to be equality over their x- and y-coordinates. How do we override hashCode to mimic this behavior? There are two integer fields for Point—how do we combine them into a single hash value? One solution is to simply add the two fields together:

```java
public int hashCode() {
    return x + y;
}
```

However, this means that the two points (1,0) and (0,1) will hash to the same value: 1. We would like to avoid permutation collisions of this nature since they are common for a variety of objects.

Joshua Bloch recommends the following general strategy for implementing hashCode for classes taking into account their fields:

```java
public int hashCode() {
    int result = /* a random non-zero integer */;
    for (/* each field f in this object */) {
        int c = /* the result of hashing the field f */;
        result = 31 * result + f;
    }
    return result;
}
```

Start with a random non-zero integer. Then for each field of the object, add its hash code to the result after multiplying the current value of result by 31. This approach is essentially taking the linear combination of the hash codes of the fields of the object with the exception of multiplying the accumulated value by 31. According to Bloch, the value 31 is chosen because:

1. 31 is an odd prime. An odd prime is chosen to minimize the probability that the multiplication does not result in the hash value and the number of buckets having a similar prime factorization. If this is the case, then the hash values will collide much more frequently.
2. $31$ is specifically chosen because the compiler can turn operations over this number into series of shifts and subtractions, specifically $31 \times i = (i \ll 5) - i$. Although, any odd prime of sufficient size is fine.

For our point class above, we would adopt this strategy into its `hashCode` method as follows:

```java
public int hashCode() {
    int result = 1091;
    result = 31 * result + x;
    result = 31 * result + y;
    return result;
}
```