12 Sets and Maps

A *map* is a data type designed for efficiently storing and retrieving a *value* associated with each unique identifier known as a *search key*. The map data type turns out to be one of the most significant and most versatile data types. The map data type resembles a mathematical function $f : D \to R$, mapping a key from a domain D to a value in the range R. That is to say, $f(k)$ retrieves the value associated with the key k. Depending on the programming language used and the context in which the discussion arises, maps are also variously known as associate arrays, dictionaries, and tables.

Maps vs. Associate Arrays. A map is sometimes called an associate array because it generalizes a normal array to allow the indices into the array to be a "richer" unique key, as opposed to a number between 0 and $n-1$, where n is the size of the array.

The search keys have to be unique whereas the values need not be. Together, a pair of key and its associated value—a key-value pair—is known as an *entry* in the map. Below are some application scenarios that fit the map abstraction:

- A regular dictionary is a mapping from words to their definitions.
- A book's index is a mapping from terms to the page numbers where the terms appear.
- A university's information is, in part, a mapping from student IDs to student records.
- The domain name system (DNS) is, in part, a mapping from domain names (e.g., `muic.io`) to IP addresses (e.g., `202.17.99.187`).

The table below is an example of what can be represented using a map data type. The keys have to be unique because they are used to look up a value. The values can have repeats.

Key	Value
Thai	Baht
Japan	Yen
German	Euro
Greece	Euro
USA	Dollar

12.1 A `Map` Data Type

What operations are commonly performed on a map? The main activities on the map data type include: adding a new entry, updating an entry, retrieving a value corresponding to a key, and removing an entry. In the view of an abstract data type (ADT), a *map* keeps a collection of key-value entries while supporting the following operations:

- `get(k)` — returns the value associated with a key `k`.
- `put(k, v)` — associates the value `v` with a key `k`.
- `containsKey(k)` — returns a Boolean indicating whether key `k` is present in the collection.
- `remove(k)` — removes the key `k` and its associated value.

Apart from these defining operations, the following are convenience functions often provided by the map data type:

Code 12.1: A minimal `Map` interface in Java.

```java
public interface Map<K, V> {
    // retrieve the value associated with a key key
    V get(K key);

    // associate value val with a key key
    void put(K key, V val);

    // does the map have this key?
    boolean containsKey(K key);

    // remove an entry for this key
    void remove(K key);

    // the number of entries
    int size();

    // is it empty?
    boolean isEmpty();
}
```

- `size()` — return the number of keys in the collection.
- `isEmpty()` — return whether the collection is empty.

This ADT translates to an interface in Java as shown in Code 12.1. The interface uses a generic type `K` as a placeholder for the type of the keys and `V`, for the values. Notice that the type of the keys and the type of the values are often not the same.

Set

A `Set` is a close cousin of the `Map` data type. Some applications are only interested in maintaining membership, i.e., a collection of keys. They have no interest in associating each key with a value. In these applications, mathematical sets are an appropriate abstraction. We define a basic *set* abstract data type with the following operations:

- `contains(k)` — returns whether `k` is present in the set.
- `add(k)` — adds `k` to the set. If the element already exists, it has no effects.
- `remove(k)` — removes `k` from the set.

Apart from these, convenience functions such as `size` and `isEmpty` are usually defined. This leads to an interface in Java as shown in Code 12.2.

Simple Use Case

Before we begin thinking about implementing the data types, let us take a quick look at an example that uses such a data type as part of a larger program.

Code 12.2: A minimal Set interface in Java.

```java
public interface Set<E> {
    // does the set have this element?
    boolean contains(E k);

    // add this key to the set
    void add(E k);

    // remove an entry for this key
    void remove(E k);

    // the number of entries
    int size();

    // is it empty?
    boolean isEmpty();
}
```

Example. Consider computing the histogram of a sequence of numbers—that is to say, find for each number, the number of occurrences of that number. As a concrete example, for the sequence {3, 1, 2, 1, 1, 2, 5}, we are to produce a table with the following information:

Number	Frequency
1	3
2	2
3	1
5	1

This can be conveniently computed, thanks to the map data type, where we go over each element and keep the counts so far in a map:

```java
Map<Integer, Integer> makeHistogram(List<Integer> seq) {
    Map<Integer, Integer> histogram = ... // use a suitable Map
    for (int elt: seq) {
        if (histogram.containsKey(elt))
            // add 1 to the corresponding count
            histogram.put(elt, histogram.get(elt) + 1);
        else
            // a new number, so has a count of 1
            histogram.put(elt, 1);
    }
    return histogram;
}
```

12.2 Basic Implementation Strategies

As with other data types we have considered, we are interested in supporting the operations of the data type as fast as possible using as little space as we can. There are two basic strategies that we have considered in the past for keeping a collection.

We could keep the collection as a list of key-value pairs. This strategy appears to be low maintenance but looks to be expensive for operations that require looking up by key. While adding a new key is inexpensive, searching for or updating a key amounts to performing linear search on the collection.

The look-up performance can be improved by keeping the list sorted by key. While this strategy leads to a marked improvement in the lookup performance, it is a high-maintenance option. While searching, thanks to binary search, is now fast, keeping the list always sorted is expensive.

Crucially, in both strategies, the operations `put` and `remove` require that we first locate the key in the list. If the list is sorted, we could use binary search to locate the key; otherwise, it seems that linear search is about the best thing one can do. Following that, we either update the value or delete that entry altogether. This leads to the following cost table, where we use n to denote the number of keys kept in the table:

Operation	Unsorted List	Sorted List
`get(k)`	$O(n)$	$O(\log n)$
`containsKey(k)`	$O(n)$	$O(\log n)$
`put(k, v)`	$O(n)$	$O(\log n)$ or $O(n)$ if k is new
`remove(k)`	$O(n)$	$O(n)$
Space Usage	$O(n)$	$O(n)$

Table 12.1: Running time of basic implementation strategies for `Map`.

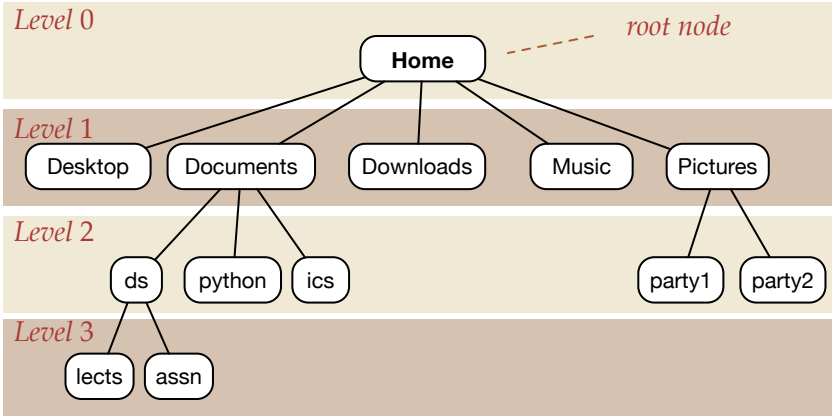
In the rest of this chapter, we present two more advanced techniques for maintaining the `Map` data type more efficiently.

12.3 Introducing Trees and Binary Trees

The data structures we have seen so far share a common property: they are linear—there is a sense of an absolute position on a line from left to right. Often, this confines our thinking and limits what we can achieve. It is possible, however, to break this pattern. One of the most important "nonlinear" structures is the tree structure. We will begin by developing an intuitive understanding of what trees are.

Instead of organizing data linearly, a *tree* is a hierarchical structure with a few notable properties. Each item in a tree is generically called a *node*. There's a special top element known as the *root*. Every node has zero or more *children*. When a node doesn't have any children, it is called a *leaf*. In Computer Science,

we often draw trees with its root node at the top and their descendants in subsequent layers below that. An example of a tree structure is given below.

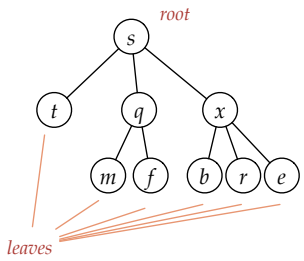


More formally, we can define a tree as follows:

Definition 12.1. A *tree* T is an ordered pair (N, p), where N is a set of nodes that keep the tree elements and $p : N \to N \cup \{\perp\}$ is a function storing a child-parent relationship satisfying the following properties:

- If the tree is nonempty, there is a *unique* node $r \in N$, called the root, such that $p(r) = \perp$, that is, it has no parent.
- Every node other than r has a parent in N, that is, $p(v) \neq \perp$ for all $v \neq r$. The children of a node w are those v such that $p(v) = w$.
- The parent chain of every vertex other than the root itself eventually reaches the root r. In other words, for every $v \in N$, $v \neq r$, $p(p(\cdots p(v))) = r$.

As an example, consider the tree diagram on the right and its corresponding formal description. In this particular example, the set of nodes N is $\{s, t, q, x, m, f, b, r, e\}$, and the parent-child relationship p is as follows:



Node	*Parent*
s	$p(s) = \perp$
t	$p(t) = s$
q	$p(q) = s$
x	$p(x) = s$
m	$p(m) = q$

Node	*Parent*
f	$p(s) = q$
b	$p(t) = x$
r	$p(q) = x$
e	$p(x) = x$

Trees are used both to "physically" represent data and to conceptually express ideas (help shape our thought, though a real tree is perhaps never stored anywhere). In the former case, we need to be able to materialize them—we need to represent them in our program. *How to represent a tree structure in a computer program?*

Representing A Generic Tree. With this definition in mind, we could represent a tree by storing this p function, for example, as a `Map`—provided we already have a good `Map` implementation. This allows for efficiently looking up the parent of a given node, but it would be hard to determine the children of a particular node. In code, this means, for example, storing the function p as a `HashMap`. The goal here would be if we have `Map<Integer, Integer> p` that represents p, then `p.get(u)` should return the identity of the parent of u.

Java has two main implementations of the `Map` data type: `HashMap` and `TreeMap`. They are based on different data structuring techniques and offer different performance tradeoffs.

This particular representation is unlikely practical especially when we are trying to use trees to implement a `Map` in the first place. Let us take a look at a specific variant of trees and another representation that directly remembers the children of each node.

Binary Trees

Of particular interest to us is the so-called *binary trees*. These are tree structures where each node can have at most *two* children. We will see how to take advantage of such a structure to obtain a fast implementation of many data structures in this chapter and the chapters that follow.

We will now turn our focus to binary trees as they are what we will mainly encounter in the rest of this book.

There are many ways to encode a tree structure in a program you write. Indeed, modern languages such as Python, Java, C/C++, etc. provide various mechanisms for expressing them. Here, we will look at a general representation: a representation using a node class and references.

As our running example, we'll consider the following trees:

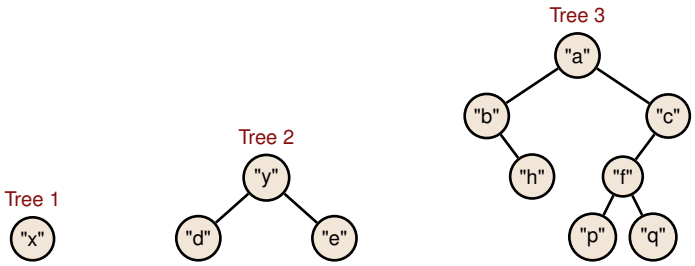


Figure 12.1: Example binary trees of different sizes

Using nodes and references. To motivate this representation, consider the second tree (above). We would like for each node to store its data, together with its two children—and we distinguish between the left child and the right child. Hence, we can define a class that represents each node individually. We will then link them up to form a tree. Specifically, our class will store information about this very node itself, for instance, the key and value associated with this node. It will store what the left subtree and the right subtree are (as references). Our preliminary cut will look as follows:

```java
class TreeNode<E> {
    E key;
    TreeNode<E> left;
    TreeNode<E> right;

    TreeNode(E key, TreeNode<E> left, TreeNode<E> right) {
        this.key=key; this.left=left; this.right=right;
    }

    TreeNode(E key) { // Call the constructor above
        this(key, null, null);
    }
}
```

In this representation, we can therefore represent the trees in our examples as follows:

```java
TreeNode<String> tree1 = new TreeNode<>("x");
TreeNode<String> tree2 =
      new TreeNode<>("y",
          new TreeNode<>("d"),
          new TreeNode<>("e"));
TreeNode<String> tree3 =
      new TreeNode<>("a",
          new TreeNode<>("b", null, new TreeNode<>("h")),
          new TreeNode<>("c",
              new TreeNode<>("f", new TreeNode<>("p"),
                  new TreeNode<>("q")),
                  null));
```

Before we move on, let us write a function that directly works with this tree representation. We are to write a function `depth` that returns the depth of the tree (i.e., the length of the longest path in the tree).

```java
int depth(TreeNode<E> u) {
    if (u==null)
        return 0;
    else
        return 1 + Math.max(depth(u.left), depth(u.right));
}
```

Now that we know how to represent binary trees, we will put that knowledge to use. Two important ideas are in order: (1) the ability to systematically traverse the tree in different order, and (2) orderly data organization strategies that will make navigating trees easier.

Binary Tree Traversal

We will study a systematic way of visiting all nodes in a tree. Given a tree T, a traversal of T prescribes an ordering in which nodes of T are visited.

When a node is visited, the specific action taken at that node depends on the particular application. This could be simply printing out the data at that node, incrementing a counter, or some complex computation.

For binary trees, there are *three* common traversal patterns that we will consider in turn: (1) preorder traversal, (2) inorder traversal, and (3) postorder traversal. All these patterns are best described recursively.

Preorder. Visit the node itself—then visit both children recursively and return:

```
def preorder(x):
    visit(x)
    preorder(x.left)
    preorder(x.right)
```

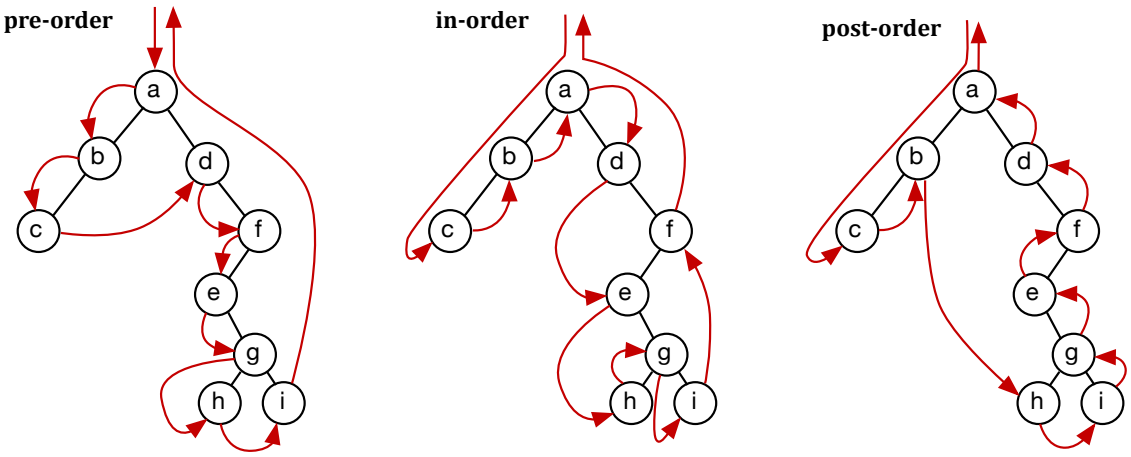
Inorder. Visit the left recursively—then visit the node, and visit the right recursively.

```
def inorder(x):
    inorder(x.left)
    visit(x)
    inorder(x.right)
```

Postorder. Visit both children recursively—then visit the node itself and return:

```
def post(x):
    post(x.left)
    post(x.right)
    visit(x)
```

Example. The figure below shows the order in which nodes are visited when these common traversal patterns are employed.

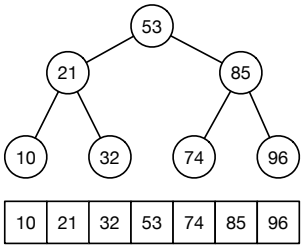


How are these used in practice? We have seen numerous applications of the for-each construct on lists. It is best to think of these patterns as for-each constructs for trees, but since trees are not flat, there are many patterns to use. Below is a short list of common applications.

- Example 1: An arithmetic expression can be represented as a tree. Then, we need to traverse that tree to evaluate it or to manipulate it in some way.
- Example 2: The table of contents of a book can be seen as a tree. Printing this out involves tree traversal.
- Example 3: An HTML page is represented in the browser as a DOM (document-object model) tree. If we want a program to go over these objects (e.g., to update color or other property), that is a tree traversal.

12.4 Binary Search Trees

To motivate this structure, on right, consider the sorted array and a tree that we superimpose on. The tree is a perfect binary tree on 7 nodes. These nodes coincidentally—or perhaps not—are labeled according to numbers from the sorted array below the tree. In many ways, this should remind us of binary search. When we look up a key using binary search, the walk from the root to that key in the tree is exactly the comparisons we make in binary search.

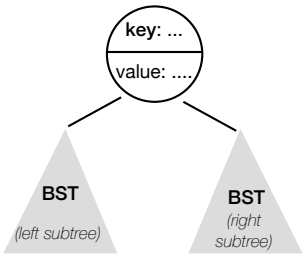


Remember that a perfect binary tree (as shown) is one in which all internal nodes have two children and all leaves are at the same level.

In the general sense of a `Map`, the collection is not fixed and our goal is to be as efficient as binary search in the face of new items getting added, existing items getting removed, etc. *How can we maintain, perhaps implicitly, a sorted collection while supporting insertion, deletion, and search efficiently?*

Remember that a binary tree is a tree in which every node in the tree has at most two children. Binary search trees (BSTs) are a binary-tree-based data structure that can be used to store and search for items that satisfy a total order. For a bit of history, BSTs date back to around 1960, usually credited to Windley, Booth, Colin, and Hibbard.

We generally work with the assumption that the keys are unique. More precisely, we define the binary search tree data structure as follows:



A Binary Search Tree with a root, and left and right (potentially empty) subtrees.

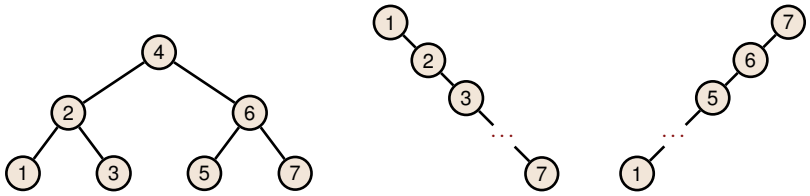
Definition 12.2. A *binary search tree* (BST) is defined recursively as
(a) an empty tree; or
(b) a node storing a key-value pair $(key, value)$, together with two BSTs, known as the left and right subtrees, where the left subtree contains only keys that are smaller than *key* and the right subtree contains only keys that are larger than *key*.

Therefore, a binary search tree satisfies the following invariant:

> For any node v, all of the left subtree is smaller than the key at v, which is smaller still than all of the right subtree.

In a standard implementation, one keeps the following attributes for each node: `key`, `value`, `left`, and `right`, which represent, respectively, the key, the value, the (reference to the) left child, and the (reference to the) right child.

Example. We give a few examples of binary search trees on the keys $1, 2, \ldots, 7$, omitting their values. Notice that some of these trees may be lopsided, some completely balanced, as the BST definition doesn't prescribe any exact shape.



Representing BST Nodes

Let's first work with BST nodes assuming that both the key and the value are integers. In this case, we'll just need a class that keeps two integers—let's call them `key` and `value`—as well as the two children.

```
public class BST {
    int key, value;
    BST left, right;
}
```

In many cases, we want our implementation to be more general and support a wide variety of types. In this case, the class will be declared as `BST<K, V>` with the intent that `K` is the key type and `V` is the value type. It is necessary that the key type is `Comparable` because otherwise we won't be able to make comparisons and navigate the tree. We have:

For `Comparable` objects, we cannot use the usual <, >, <=, etc. operators to compare them. Use `.compareTo` instead.

```
public class BST<K extends Comparable<? super K>, V> {
  K key;
  V value;
  BST<K,V> left, right;
}
```

Notice that this `BST` class is a bare-bones implementation of the internal nodes. Encapsulation and other packaging techniques from earlier chapters can be used here to improve upon usability and convenience. To illustrate clearly how binary search trees work, the rest of this chapter, however, will work directly with this low-level implementation.

Working Directly With Binary Search Trees

We consider performing two simple tasks on binary search trees. Despite their simplicity, these examples will help acquaint us with properties of the BST.

Searching For a Given Key. In this routine task, we're given a key k and we are asked to retrieve the value associated with that key or report that the key doesn't exist. Because a binary search tree maintains strict ordering of keys, when we compare k with the key at a node, we know right away which branch—left or right—to take next. For example, suppose v.key $> k$ at a node v. Then, we know that if k exists in the tree rooted at v, k must be in the subtree v.left since all keys smaller than v.key belong in the left subtree. This reasoning yields the following algorithm:

```
// not real Java code. need to use compareTo
public V search(K k) {
    if      (k==this.key)              return this.value;
    else if (k>this.key && right!=null) return right.search(k);
    else if (k<this.key && left!=null)  return left.search(k);
    else                               return null;
}
```

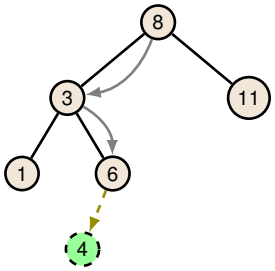
Finding the Largest Key. In Java, this is known as the `lastKey()` method. Let's pause for a moment and think about how one might locate the largest key in the tree. A moment's thought shows that the largest key lies at the bottom-right tip of the tree. This is because if there's anything bigger than the root, it must be in the root's right subtree. And inside that subtree, if there's anything bigger than its root, it must be in that root's right subtree. But if at any point, that subtree no longer has a right subtree (it's empty), the root of that subtree itself is the biggest key in that subtree. Following this line of reasoning, we arrive at the following algorithm (once again assuming integer keys and values):

```
public K lastKey() {
  if (right!=null) return right.lastKey();
  else             return key;
}
```

Greatest Key Less Than or Equal to a Given Key. In Java, this is known as the `floorKey()` method. How do we implement such a method?

```
// this is not real Java code. Use compareTo
public K floorKey(K k) {
    if      (k == key) return this.key;
    else if (k < key && left !=null) return left.floorKey(k);
    else if (k > key) {
       K rightFloor = (right!=null)?right.floorKey(k):null;
       return (rightFloor==null)?this.key:rightFloor;
    }
    else return null;
}
```

Adding and Removing a Key. *How can we add a new element to the tree? How about deleting an element?* To add, just do a search, it will lead us to where need to insert it. Try adding 4.



Deleting is more complicated. If it is a leaf, we can just let that node go; otherwise, we will have to find a replacement. The details would be beyond the scope of this book.

Remarks. A common pattern so far—and one that will be recurring throughout the discussion of BSTs—is that the performance of operations on a BST depends largely on the height of the tree. In both the `search` and `lastKey`

algorithms, the running time is proportional to the length of the path that the algorithm traverses, which is never longer than the tree's height. Therefore, we strive to keep the height small. It is not hard to convince ourselves that a tree with n keys has height at least $\log_2 n$. More precisely, we have the following lemma:

Lemma 12.3. An n-node binary tree has height at least $\log_2(n+1)$.

Proof. First, consider that the largest binary tree with height h (i.e. the tree with the most number of nodes) has exactly $2^h - 1$ nodes. This is the tree where all h levels are full, so the total number of nodes is $2^0 + 2^1 + 2^2 + \ldots 2^{h-1} = 2^h - 1$, by the geometric sum formula. Therefore, if an n-node tree has height h, then $n \leqslant 2^h - 1$, which means $h \geqslant \log_2(n+1)$, proving the lemma. □

From the intuition developed so far, if binary search trees are kept "balanced" in some way, then their heights will be small and they can usually be used to get good bounds. We refer to such trees as *balanced search trees*. If trees are never updated but only used for searching, then balancing is easy—it needs only be done once. What makes balanced trees interesting is their ability to efficiently maintain balance even when updated. To allow for efficient updates, balanced search trees do not require that the trees be strictly balanced, but rather that they are approximately balanced in some way. In fact, it is impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

Running time of a `Map` implemented using balanced binary search tree (BBST).

Operation	BBST
`get(k)`	$O(\log n)$
`containsKey(k)`	$O(\log n)$
`put(k, v)`	$O(\log n)$
`remove(k)`	$O(\log n)$

Since any balanced binary search tree data structure will be about $O(\log n)$ deep, the `Map` operations (e.g., looking up a key, inserting a new item, or deleting an existing one) can all be implemented in at most $O(\log n)$ time.

Dozens of balanced search trees have been suggested over the years, dating back to at least AVL trees in 1962. These trees mostly differ in how they maintain balance. We briefly mention two of them:

1. *AVL trees.* Invented in 1962 by two Russians G. M. Adelson-Velskii and E. M. Landis, these are binary search trees in which for any node, the heights of the two child subtrees can differ by at most 1. It has been proved that the height of an AVL tree is at most

$$\log_\varphi(\sqrt{5}(n+2)) - 2 \leqslant 1.44 \log_2(n+2),$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the Golden ratio, which is approximately $1.6180339887\cdots$.

2. *Red-Black trees.* More popular in practice than AVL trees, red-black trees are binary search trees with a somewhat looser height balance criteria. The basic idea is to label each node either red or black (requiring one extra bit of storage) and impose certain criteria about which nodes can be red/black. Overall, this leads to a tree with height at most $2\log_2(n+1)$, which is larger than the height of an AVL tree.

Sorted vs. Unordered Maps

The basic function of a map is to store a mapping between keys and values. More precisely, it remembers for each key of interest, the value the key corresponds to. Hence, the basic operations supported by a map are `get`, `put`, and `remove`, as discussed earlier.

Java comes with two standard `Map` implementations: `HashMap` and `TreeMap`. Both the `HashMap` and the `TreeMap` support these basic operations. But:

- `HashMap` supports them in $O(1)$ time for reasons the next section will detail.
- `TreeMap`, implemented as a red-black tree, supports them in $O(\log n)$ time, where n is the number of keys in the collection.

This brings up the question, *why would one choose to use a* `TreeMap` *over a* `HashMap`*?* Of course, if all we care about are `get` and `put`, then the `HashMap` will be the implementation of choice. However, it turns out there are operations that can be efficiently performed on a sorted collection—and equally efficiently on a balanced binary search tree—that will otherwise take a long time to support if the collection is unordered. Below are some of such useful operations:

- `lowerKey(key)` — returns the greatest key strictly less than the given key.
- `floorKey(key)` — returns the greatest key less than or equal to the given key.
- `ceilingKey(key)` — returns the least key greater than or equal to the given key.
- `higherKey(key)` — returns the least key strictly greater than the given key.

Hence, at a high level, readers should know the following:

- The `HashMap` does not care about the relative ordering of elements; it focuses exclusively on the task of storing and retrieving a particular key.
- The `TreeMap` understands the relative ordering of the elements, thereby being able to answer order-related queries (such as a nearby element).

In Java, there are two interfaces that capture the notion that the collection is kept sorted: `NavigableMap` and `SortedMap`*. In addition to what has been mentioned, readers can find out about other operations on such collections by browsing the official documentation.

12.5 Hashing and Hash Tables

The $O(\log n)$ running time is fast, but we want even better performance. Earlier, we learned that it takes only $O(1)$ to retrieve an item in an array. Is it possible to match this efficiency by perhaps taking advantage of the array's fast access to speed up our `Map`?

*Incidentally, `NavigableMap` "extends" the `SortedMap` interface, adding a few things for navigating the map.

Tackle An Easier Problem: Keys Are Small Integers

Suppose for a moment that keys are all small integers. Then, we could just use an array to keep the map. The main idea is for the i-th index in the list to store the value for the key i. With this scheme, if the keys are numbers between 0 and $H-1$, we will allocate an array of length H, filled with `null` initially. This is to indicate that none of the keys are associated with a value. Following that, we can support `get(k)` by looking at the k-th index and `set(k, v)` by setting the k-th index to v.

As an example, the code below implements a map from $\{0, 1, \ldots, H-1\}$ to `Strings`, storing the mapping inside an array called `storageArray`.

```
class SimpleIntStringMap {
    private String[] storageArray;

    SimpleIntStringMap(int H) { storageArray = new String[H]; }
    String get(int k) { return storageArray[k]; }
    void put(int k, String v) { storageArray[k] = v; }
}
```

The state of `storageArray` after the example code finishes:

0	null
1	"nap"
2	"bugs"
3	null
4	"bunny"

Suppose we have instantiate it with $H = 5$ and use it as follows:

```
SimpleIntStringMap myMap = new SimpleIntStringMap(5);
myMap.put(2, "bugs");
myMap.put(4, "bunny");
myMap.put(1, "nap");
System.out.println(myMap.get(4)); // ==> bunny
```

It is clear that accessing such a `Map` takes $O(1)$ time, as summarized by the following theorem:

> **Theorem 12.4** (Direct Hashing)**.** If keys are integers between 0 and $H-1$, inclusive, it is possible to keep a `Map` in $O(H)$ space supporting all operations in $O(1)$ time, except for constructing the `Map` initially, which takes $O(H)$ time.

For the general case, the obvious challenge is: *what if the range of numbers is far larger than the number of entries we intend to store (i.e., $H \gg n$) or we simply have noninteger keys?*

The first concern means using this scheme, our space usage will be about H which is significantly larger than the number of entries n, a situation that we wish to avoid. The second concern is equally important; there are many applications where the keys are, for example, strings.

We will address these concerns next using an idea known as hashing.

Hashing Explained

To overcome the challenges outlined earlier, we will come up with a function that takes an arbitrary key k and maps it to a small range, say $[0, N-1]$, where N is not much larger than the total number of keys n. More specifically, a hash function h maps each key k to an integer in the range $[0, N-1]$, where N, hopefully $N = O(n)$, is the capacity of the bucket array for a hash table.

Once we have such a hash function, we can resort to the method we just developed: use the hash value $h(k)$ as an index into our list A, instead of using the real key k directly, which may be unsuitable.

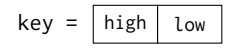
It helps to conceptually break down the hash function into two components: a component that generates hash codes and a component that compresses a hash code into the desired range. As the hash table's size may need to change over time, this conceptual view is desirable as it decouples hash code generation from the capacity of the hash table being used. They work together as illustrated in the diagram below:

To illustrate this process, we will consider a key $k =$"déjà vu". The function that generates a hash code may produce a number denoting this key. For example, this could result in the hash code `0x10010abcdeadbeef`, a 64-bit number. This number is clearly too large to be a valid range of any reasonable array. Hence, we use a compression function to convert this hash code to a number in the desired range. If, say, we have an array of size $N = 1001$, then the compression function may give us 507.

Hash Codes

To generate the hash codes, we wish to design a function that takes an arbitrary key k and computes an integer—the so-called hash code—for k. An important goal for hash codes is that if two keys are different, they should lead to different hash codes. In other words, we wish to minimize the likelihood of two different keys getting assigned the same hash code. Below are some common techniques for creating hash codes:

Mix High With Low. We can view our key as a large number; after all, it is a series of bytes. Once interpreted in this way, this simple idea mixes the higher-order bits with the lower-order ones. For example, suppose we have a 64-bit number and we want to generate a 32-bit hash code. One way to mix them is to use the exclusive OR operator (the `^` operator in Java).



Then,
`hashCode = high ^ low`

This symmetry means "evil" and "live" would result in the same hash code.

We could repeat this process to reduce the number bits as many times as we wish. While this is convenient, this method has a glaring deficiency: the high bits and the low bits are symmetrical in the sense that if we were to swap them, we would still have the same hash code, which is not ideal.

Polynomial Hash Code. The above scheme is not a good option when the order of the elements is important. A better hash code for this type of data should take into account the positions of the data element. We can fix this by using a polynomial. Let $x = x_0x_1 \dots x_{n-1}$ be a sequence of length $n \geqslant 1$. Pick an $a \neq 0$. To compute the hash code for x, we calculate

$$x_0 \cdot a^{n-1} + x_1 \cdot a^{n-2} + \cdots + x_{n-2} \cdot a + x_{n-1} = \sum_{k=0}^{n-1} x_k \cdot a^{n-k-1}.$$

At first glance, this may seem like we need to compute powers of a. An optimization is possible: Keep a running sum and as we encounter a new data item, we multiply the running run by a and add the new value in:

```
int h = x[0]; // running hash code
for (int i=1;i<n;i++) {
    h = h*a + x[i];
}
```

Note: According to Goodrich et al. [GTG14], experimental studies have shown that 33, 37, 39, and 41 are particularly good choices for a when working with English words. In a list of over 50,000 English words, they found using these numbers result in less than 7 hash-code collisions in each case.

Cyclic-Shift Code. Another popular function for hashing strings is the cyclic-shift code. Instead of multiplying by a each time, it performs a cyclic shift of the running sum. Below is an example implementation of cyclic-shift code for a string s that shifts by 5 bits:

```
int h = 0; // running hash code
int mask = 0xffffffff;  // this is 32 1's
for (int i=0;i<s.length();i++) {
    h = (h << 5 & mask) | (h >> 27); // cyclic shift by 5 bits
    h = h + (int) s.charAt(i);       // add in the next symbol
}
```

Java Built-in Hash. For built-in data types, Java provides a hash function. Every object o has `o.hashCode()`. The language specification requires that if `x.equals(y)`, then `x.hashCode() == y.hashCode()`. This is an extra requirement that we have to comply when writing a new class.

Compression

The hash code for a key k may not be readily usable as an index into the list we're keeping. Often, this is because the hash codes are intentionally on a large space (e.g., 64-bit or 128-bit numbers) to minimize the chance that two different keys yield the same hash code. As outlined before, we will use a compression function to map such a hash code to an integer between 0 and $N-1$ (inclusive), where N is the target storage array size. Below, we look at two common compression functions.

Modulo N. The simplest, yet usable, compression function is to map an integer h to $h \mod N$. A bit of number theory shows that if N is taken to be a prime number, this compression function will help nicely spread out the hash values. If N is not prime, we run a greater risk of collisions.

Linear Congruential Compression: A more sophisticated function can yield better results. In this case, we map an integer h to

$$[(a \cdot h + b) \mod p] \mod N$$

where p is a prime number larger than N, and $a \neq 0$,b are numbers chosen arbitrarily between 0 and $p-1$ (inclusive).

Collision Resolution

Our plan is to keep an array of size about $O(n)$ and use the hash value of each key as the index into this array. But we aren't quite finished.

Despite the efforts we put into crafting hash functions that will not send two keys to the same index, it is inevitable that collisions will happen. This is because we are mapping a (much) larger domain into a smaller range. Our new challenge is what to do when two or more keys, say, `B` and `N`, hash to same number and hence would occupy the same spot in the array?

Separate Chaining. The simplest solution is known as separate chaining. If multiple keys happen to hash to the same index, no worries! In separate chaining, we will build a list (a bucket) for that index and keep every entry that is sent there. If too many entries end up at the same index, searching and management will be difficult. The hope is therefore that there are only a few entries per index. Below is an illustration with $N = 7$ and the keys, as well as their corresponding hash codes and values, as shown on the left table.

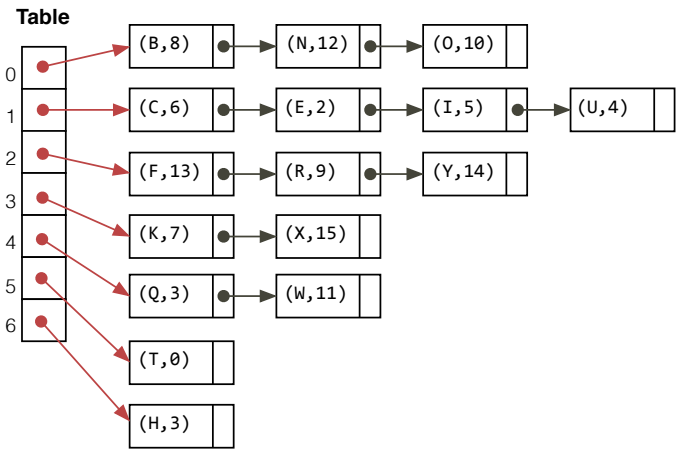


Figure 12.2: Separate chaining example, which compresses hash codes using the simple modulo N method.

Key	hashCode	Value
T	5	0
H	6	1
E	1	2
Q	4	3
U	8	4
I	8	5
C	8	6
K	3	7
B	0	8
R	9	9
O	0	10
W	11	11
N	7	12
F	9	13
Y	2	14
X	3	15

This list can be kept as an actual linked list, and as long as the size is small, almost any kind of collection will suffice. Notice that the `get` and `put` operations relies on first deriving the index (via hashing) and taking time proportional to the size of the chain/collection at that index.

Would the chains become too long, thus hampering performance? To begin analyzing the performance implications, we will first discuss how good hash

functions behave. As a standard assumption in the literature, which can be justified both theoretically and experimentally, good hash functions satisfy the following property:

> **Definition 12.5** (SUHA: Simple Uniform Hashing Assumption)**.** A hash function h uniformly distributes keys among the integer values between 0 and $N-1$. Mathematically, for any key k in the domain and $i \in [0, N-1]$, $\mathbf{Pr}[h(k) = i] = 1/N$.

Now consider an index $i \in [0, N-1]$. Let X_i denote the number of entries that hash to this index. Hence, X_i represents the length of the (linked-list) chain. Suppose the keys in this `Map` are $k_1, k_2, \ldots, k_n$. If $X_{i,j}$ indicates whether key k_j hashes to index i, then we can break down X_i as

$$X_i = X_{i,1} + X_{i,2} + X_{i,3} + \cdots + X_{i,n}.$$

By linearity of expectations, we have

$$\mathbf{E}[X_i] = \mathbf{E}\left[X_{i,1}\right] + \mathbf{E}\left[X_{i,2}\right] + \mathbf{E}\left[X_{i,3}\right] + \cdots + \mathbf{E}\left[X_{i,n}\right].$$

If the hash function is good (i.e., satisfy SUHA), then the probability that a key k_j hashes to index i is $1/N$. Thus, for every $j \in [n]$, we have $\mathbf{E}\left[X_{i,j}\right] = \frac{1}{N}$ and so

$$\mathbf{E}[X_i] = \frac{n}{N}.$$

This means that as long as $N = \Theta(n)$, the expected length of a chain X_i is $\Theta(1)$. In summary, we can use $N = \Theta(n)$ space to keep such a map and expect the operations to take $O(1)$ time.

Exercises

Exercise 12.1. Consider the following two scenarios in the `makeHistogram` example at the beginning of the chapter. What is the overall running time of the function if a `HashMap` used for the `Map`? What if a `TreeMap` is used instead?

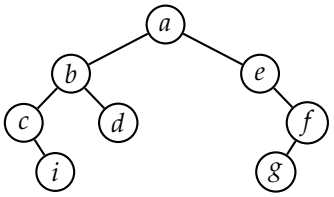
Exercise 12.2. Implement a class `UnsortedListMap` that implements the basic `Map` interface by keeping internally an unordered sequence as discussed in 12.2.

Exercise 12.3. Implement a class `SortedListMap` that implements the basic `Map` interface by keeping internally a fully-sorted sequence as discussed in 12.2.

Exercise 12.4. Extend the `TreeNode` class to support an arbitrary number of children. (*Hint:* Instead of fixed named attributes, keep a list of references.)

Exercise 12.5. By working directly with the `TreeNode<String>` class, write a function `longestKey` that finds the longest key in the tree.

Exercise 12.6. Consider the following tree. Describe the order in which pre-, in-, post- order traversal will visit the nodes.



Exercise 12.7. Write a function to construct a balanced binary *search* tree from a list of keys—and analyze its running time. In particular, write a function

```
BST buildBST(int[] keys)
```

that takes an array of integer keys and returns a BST represented using the previously described class.

Importantly, if n is the length of `keys`, the algorithm should take at most $O(n \log n)$ time and construct a BST that is no deeper than $1 + \log_2 n$ levels.

Finally, analyze the running of your implementation, as well as the height of the resulting tree, to show that it meets the requirements.

Exercise 12.8. Remember our discussion about tree representations? The parent mapping representation, which falls out directly from our definition of a tree, keeps for each node, the parent of that node in a map. For this exercise, we will assume the keys are integers and are unique. In this context, then, parent mapping keeps a map `Map<Integer, Integer>`, where a node is referred to by its key—and looking up a node in this map would result in the key of its parent node.

Your Task: Write functions to change to and from this representation the more traditional representation using a `TreeNode` class.

Subtask I: Implement a function

```
HashMap<Integer, Integer> treeToParentMap(TreeNode T)
```

that takes a binary tree `T`, though not necessarily a BST, and returns a `HashMap` representing the same tree using the parent-mapping representation.

Subtask II: Implement a function

```
TreeNode parentMapToTree(Map<Integer, Integer> map)
```

that takes a parent-mapping map and returns a binary tree encoded as `TreeNode`. Notice that the parent-mapping representation has no notion of left vs. right. The code is free to choose which is left and which is right. Moreover, we guarantee that the tree encoded in the map is a legit binary tree, though not necessarily a BST.

Exercise 12.9. Write a function `secondKey` that returns the second-smallest key in a binary search tree. In the worst case, the function must run in time proportional to the height of the tree.

Exercise 12.10. Johnny proposes to use the following simple implementation of `hashCode` for his awesome class:

```
public int hashCode() { return 42; }
```

Is this a legit hash code function? Is it a good idea? Why or why not?

Exercise 12.11. The `hashCode` below was the implementation for the `String` class in early versions of Java. Scrutinize it and explain why it is not exactly a good hash code function, which is the reason it was replaced in later versions of Java.

```
public int hashCode() {
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = (hash * 37) + charAt(i);
   return hash;
}
```

Chapter Notes

Maps and sets are typically implemented using a (balanced) tree or a hash table as the backing data structure. Binary trees and binary search trees have been around since at least the 1960s. Goodrich et al. [GTG14] describe other applications of tree traversals in addition to what this chapter covers. Binary search trees (BSTs) are usually kept balanced for good performance. Popular balanced BSTs include AVL trees, red-black trees, and treaps. Cormen et al. [Cor+09], and Sedgewick and Wayne [SW11] are wonderful references for the theoretical and practice sides of BSTs. To lower tree heights, trees with a higher fan-out (i.e., nodes have more children) are also used. For example, B-trees [BM72; Cor+09] are popular among disk-resident data such as a database index.

The first mention of hash tables dates back to 1953. Knuth [Knu98] attributes it to H. P. Luhn, who is also credited for inventing the chaining collision-resolution method. Other than chaining, open addressing is a common collision-resolution strategy. In open addressing, all entries are kept in the same hash table and a systematic order of examining the table, known as a probe sequence, is needed. Linear probing, quadratic probing, and double hashing are among popular probing strategies. Some excellent references for good hashing functions and collision-resolution protocols are Knuth [Knu98], Cormen et al. [Cor+09], and Goodrich et al. [GTG14]. More recent advances include Cuckoo hashing, first described by Pagh and Rodler in 2001 [PR01].