27.1 Binary Search Trees

In Chapter 25, we introduced containers called a priority queues and dictionaries. A priority queue is a dispenser in which the current item is the largest item in the container with respect to some ordering: the item with “highest priority.” A dictionary, or key-value table, is a container in which the items are “key-value” pairs. Items are accessed by key. In this section, we see a binary-tree structure that can be used to efficiently implement priority queues and dictionaries.

Specifically, we address implementing a dictionary in which the keys are ordered. The application to a priority queue is straightforward.

Suppose that we have a binary tree in which each node contains a key-value pair. Suppose further that the class of possible keys is ordered by some specific ordering. The binary tree is a binary search tree if it satisfies the following property: for any node n, the keys in the left subtree of n are all smaller than the key at n, and the keys in the right subtree of n are all larger than the key at n. Figure 27.1 illustrates binary trees in which the keys are shown as integers. Assuming that the ordering is the usual integer ordering, the tree on the left is a binary search tree, but the tree on the right is not. The node with key 19 is in the left subtree of the node with key 18, violating the binary search tree condition.

![Figure 27.1 Two binary trees, one of which is a binary search tree.](image-url)
Notice that an inorder traversal of a binary search tree traverses the nodes in key order: the first inorder node contains the smallest key, etc. Also note that to locate a key, one can start at the root and traverse a path to the element by comparing the key at each node on the path with the key that is sought.

We implement a binary search tree method to locate the value associated with a given key. We first make the following assumptions.

- The items in the binary tree are of class `KeyValuePair`, which provides methods for obtaining the key and the value:
  ```java
  public Object key ();
  public Object value ();
  ```
- The keys are ordered by the `Order` order.

First we define simple auxiliary methods for extracting the key and value from the item referenced by a `PathTraverser`, and a method for locating the key with a `PathTraverser`:

```java
private Object key (PathTraverser t) {
    return ((KeyValuePair)t.currentItem).key();
}

private Object value (PathTraverser t) {
    return ((KeyValuePair)t.currentItem).value();
}

/**
 * A PathTraverser referencing the node with the specified key.
 * locate(key).done() if there is no item in this BinarySearchTree
 * with the specified key.
 * /
 private PathTraverser locate (Object key) {
     PathTraverser t = this.createPathTraverser();
     while (!t.done() && !key(t).equals(key))
         if (order.lessThan(key(t), key))
             t.advanceRight();
         else
             t.advanceLeft();
     return t;
}
```

Now we can locate the node with a `PathTraverser`, and return the associates value.

```java
/**
 * The value associated with the specified key. Returns null if
 * there is no item in this BinarySearchTree with the specified key.
 * /
 public Object valueOf (Object key) {
     PathTraverser t = this.locate(key);
     if (t.done())
         return null;
     else
         return value(t);
}
```
Inserting a new key-value pair in the binary search tree is equally straight forward. We find a node with a bigger key and empty left subtree, or a node with smaller key and empty right subtree.

```java
/**
 * Insert the specified key-value pair in the BinarySearchTree.
 */
public void add (Object key, Object value) {
    KeyValuePair p = new KeyValuePair(key, value);
    if (this.isEmpty()) // first node is special case
        this.liftLeft(p);
    else {
        PathTraverser t = this.createPathTraverser();
        boolean done = false;
        while (!done)
            if (order.lessThan(key(t), key))
                if (!t.emptyRight())
                    t.advanceRight();
                else {
                    this.addRight(t, p);
                    done = true;
                }
            else
                if (!t.emptyLeft())
                    t.advanceLeft();
                else {
                    this.addLeft(t, p);
                    done = true;
                }
    }
}
```

Recall that we defined the BinaryTree method delete in such a way as to preserve the inorder sequence of the remaining nodes. The reason for this is so that we can delete a node from a binary search tree, and end up with a binary search tree: that is, end up with a binary tree that satisfies the ordering property for a binary search tree. For instance, if we delete the node with key 5 from the binary search tree illustrated in Figure 27.1, the result is as shown below.

![Figure 27.2 Deleting an element from a binary search tree.](image)
A method for deleting a node with a specified key can be written as follows.

```java
public void delete (Object key) {
    PathTraverser t = this.locate(key);
    if (!t.done())
        this.delete(t);
}
```

Now if a binary tree with \( n \) nodes is full, the length of the longest path is bounded by \( \log_2 n \), and the algorithm to find an item will operate in \( O(\log_2 n) \) steps. But the structure of the binary tree depends on the order in which items are added. For instance, if elements are added to an empty tree in the key order 4, 2, 6, 1, 3, 5, 7, the following tree will result:

```
        4
       /\  \\
      2  6
     /\  /\  \\
    1 3 5 7
```

On the other hand, if the items are added in the order 1, 2, 3, 4, 5, 6, 7, then the tree will look like this:

```
   1
  /\  \\
 2 3
 /\  /\  \\
4 5 6 7
```

Thus worse case performance of the method to locate a node by key is \( O(n) \). If all sequences of additions is equally likely in the construction of the tree, then the average node depth can be shown to be logarithmic. The analysis is non-trivial, and beyond the scope of our discussion.

We should mention that there are a number of methods for making sure that a binary search tree remains “balanced” as items are added and removed. (See for instance, references.) Again, these algorithms are beyond the scope of the present discussion.
We conclude with a brief introduction to hash tables. Suppose that we are building a dictionary in
which keys are relatively small integers, say in the range 0 - 999. An easy way to implement this
is to use an array containing the values, with the key as index. That is, given the component defi-
nition

```java
private Object[] theDictionary = new Object[1000];
```

we retrieve the value associated with a particular key by simply indexing into the array:

```java
/**
 * The value associated with the specified key. Returns null if
 * there is no item in this Dictionary with the specified key.
 * require:
 * 0 <= key && key <= 999
 */
public Object valueOf (int key) {
    return theDictionary[key];
}
```

The advantages are obvious. Accessing an element in the dictionary requires only constant
time, rather than linear or logarithmic time required by lists or binary search trees.

The problem, of course, is that the set of possible keys is rarely limited to a small range of
integers. Keys are often character strings (names, for instance) or more general objects. When
keys are integers, the set of possible values is typically large, such as the set of social security
numbers.

We can still use the basic idea of indexing an array (in constant time) if we have an easy way
of converting a key to an array index. A method that does this is called a hash function. A hash
function takes a key as argument, and returns an array index as result. If theDictionary is the
name of the array, a hash function can be specified as follows:

```java
/**
 * Hash value for the specified key.
 * ensure:
 * 0 <= result && result < theDictionary.length
 */
int hash (Object key);
```

![Figure 27.3 A hash function maps keys to array indexes.](image)

-5-
To locate an item in the dictionary, the hash function is applied to the key to obtain the array index:

```java
public Object valueOf (Object key) {
    return theDictionary[hash(key)];
}
```

Since our goal here is efficiency, we want the hash function to be easy to compute. For example, if keys are social security numbers, represented as integers, and the array contains 1000 elements (indexed 0 - 999), the hash function might simply return the low-order three digits of the key:

```java
int hash (int key) {
    return key % 1000;
}
```

A little thought will reveal another problem that must be addressed. Since the set of possible keys is in general much larger than the number of indexes, the hash function cannot be one-to-one. That is, the hash function will map many different keys to the same index. For instance, if my social security number is 439901245 and yours is 630728245, the above function will map both to the index 245. Attempting to enter the key 630728245 in the dictionary after we have already entered 439901245 results in a collision. We first take a closer look at hash functions, and then see some approaches to handling collisions.

### 27.2.1 Hash functions.

There are several properties we would like a hash function to have. First, we have already mentioned that it should be easy to compute. What we hope to accomplish is a structure that will permit efficient access to dictionary elements. An inefficient hash function will doom this effort from the start.

Second, we want the hash function to “uniformly distribute” keys to indexes. That is, we would like the hash function to map roughly the same number of keys to each index. The goal here is to reduce the number of collisions. To achieve this goal, we generally need to know something about the expected keys. For instance, if keys are seven digit numbers and the index set is 0 - 999, it might seem that a reasonable hash function simply takes the high-order three digits:

```java
int hash (int key) {
    return key / 10000;
}
```

However, if the seven digit keys are local telephone numbers, then this is clearly not a good approach. We are in fact selecting the local exchange, of which there will be very few.

Third, we want “equal” keys to be hashed to the same index. For example, suppose the keys are objects that denote calendar dates. We may well have two distinct objects that represent the same date. The hash function applied to each of these objects should produce the same value. A hash function that, for instance, used the virtual machine address at which the object was stored would not satisfy this requirement.

The Java class `Object` defines a method `hashCode`, specified as
public int hashCode();

that can be used to build a hash function. Thus a hash function for the array theDictionary might be written:

    int hash (Object key) {
        return Math.abs(key.hashCode()) % theDictionary.length;
    }

Note that the remainder of a positive integer divide by theDictionary.length is guaranteed to be non-negative and less than theDictionary.length.

A little caution must be exercised, however, in using the default method provided at the Object level. Two Object’s are equal only if they are identical. That is, the method equals defined for the class Object is identity: a.equals(b) if and only if a == b. Thus the method hashCode as defined in the class Object is only guaranteed to produce equal values for identical arguments. That is, if a == b, then a.hashCode() == b.hashCode().

But we want “equal” keys to have the same hash code. That is, we want to ensure that if a and b are keys, a.equals(b) implies a.hashCode() == b.hashCode(). If the method equals has been overridden for the keys, the method hashCode should also be overridden.

27.2.2 Handling Collisions.

Finally, we must address the problem of collisions. There are a number of approaches; we briefly consider two of the most straightforward.

An easy to understand though not particularly efficient approach is called open chaining or linear probing. The basic idea is simple: if we attempt to insert an element in the dictionary and the location returned by the hash function is already occupied, search the array for the next unoccupied location. The insertion algorithm would look like this.

    /**
     * Add a key-value pair, given that the dictionary is not full.
     */
    public add (Object key, Object value) {
        int index = hash(key);
        // Find the next available position
        while (theDictionary[index] != null)
            index = (index+1)%theDictionary.length;
        theDictionary[index] = new KeyValuePair(key, value);
    }

Note that the search for the next available space starts at the position specified by the hash function, and proceed thought he array in a “circular” fashion: if we reach the end of the array without finding an available position, we continue the search at the beginning of the array. The algorithm assumes that there is an empty array element. If no element is null, the algorithm loops forever.

As an example, suppose the array contains ten elements, keys are three digit numbers, and the hash function simply returns the low-order digit of the key. If elements with keys 126, 128, 206, 208, 306, 776 are added in that order, the array will be as shown in Figure 27.4.
Note that the last two elements that hashed into location 6 must be placed after all the elements that hashed into location 8. It is this behavior, in which “chains” of elements hashed into different locations can merge, that causes the relatively poor performance of open chaining.

To locate a key, we start at the hash location and sequentially search the array.

```java
public Object valueOf (Object key) {
    int index = hash(key);
    while (theDictionary[index] != null &&
           !theDictionary[index].key().equals(key))
        index = (index+1)%theDictionary.length;
    if (theDictionary[index] == null)
        return null;
    else
        return theDictionary[index].value();
}
```

If the key we are searching for is not in the dictionary, we encounter an empty element and return null. Again, the algorithm assume the array is not full.

Note that this can create a problem when items are deleted form the dictionary. For instance, suppose we delete the item with key 206 from the table of Figure 27.4 by replacing it with a null, as shown in Figure 27.5. Now if we search for the key 306, we’ll stop when we examine location 7 and report that the item is not in the table. To avoid this difficulty, the deleted element can be left in the table and marked “deleted.” Of course, over time the table can become cluttered with

![Figure 27.4 An open-chained hash table after insertion of six elements.](image1)

![Figure 27.5 Item at location 7 replaced with a null.](image2)
deleted elements. At some point, the deleted elements will need to be removed, and the remaining elements repositioned in the table. This is referred to as rehashing, and is a relatively expensive operation.

A somewhat different approach, sometimes called separate chaining, implements the dictionary as an array of lists. An element is added by appending it to the list specified by the hash function. Figure 27.6 shows a list-based hash table after adding the same six elements as above. Note

![List-based hash table after insertion of six elements.](image)

that the time required to add an element will be constant, if the list append operation is constant, and the time required to locate a key depends only on the number of items hashed to the same location. Deletion also presents no particular problem.

**Complexity.**

Although a detailed analysis is beyond the scope of the text, a little thought should convince you that the more elements a table contains, the more the look up method exhibits linear behavior. We define the load factor to be the number of elements in the table divided by the array size. With open chaining, the load factor is 0 for an empty table, 1 for a completely full table. With separate chaining, the load factor can be larger than 1.

With linear probing, if the load factor is 0.5 – i.e., the table is half full – locating a key in the table requires on average examining only about 1.5 cells. Inserting an element or determining that an element is not in the table, requires only about 2.5 probes. As the load factor approaches 1, performance degenerates substantially. With a load factor of 0.9, roughly 50 cells need to be examined to locate an item in the table.

This implies that the linear probing is acceptable if the load factor remains relatively small – less than 0.5, for instance. This is easy to achieve for a stable dictionary whose size can be determined a priori. It is not so easy to accomplish with a dynamic table, in which there are many additions and deletions. In this case, we must be prepared to increase table size when the table becomes too full. However, since the hash function depends on the number of elements in the table, increasing table size requires repositioning of all elements in the table – that is, rehashing. Clearly, this is not an operation one wants to perform too often.

With separate chaining, a load factor of 1.0 is generally considered acceptable. Locating a key requires about 1.5 problems with a load factor of 1.0.
Library Classes.

The standard Java libraries provide classes and interfaces for dictionaries and hash tables. These are the interface `java.util.Map` and the class `java.util.HashMap`. We summarize their features here. Details can be found in the standard documentation.

Map is an interface that associates keys and values. (It replaces an abstract class `java.util.Dictionary` which is now obsolete.) Among the methods specified are the following.

```java
public Object put (Object key, Object value)
```

Add a key-value pair to the dictionary, or replaces the value if an entry with the specified key is already in the dictionary. Returns the value previously associated with the key, or `null` if the key was not previously in the map.

As with the Collection method `add` discussed in Section 24.5, this method can throw an `UnsupportedOperationException`, a `ClassCastException`, or an `IllegalArgumentException`.

```java
public Object get (Object key)
```

Retrieve a value associated with a specified key.

```java
public Object remove (Object key)
```

Remove an item with the specified key. Returns the value associated with the key, or `null` if the key was not in the dictionary.

HashMap is a concrete class that implements Map. The table of key-value pairs is implemented with a separate-chaining approach. The hash function is based on `hashCode`. Keys must implement `hashCode` and `equals` in a consistent way. That is, if `k1` and `k2` are keys and `k1.equals(k2)`, then it must be the case that `k1.hashCode() == k2.hashCode()`.

The class HashMap provides several constructors. One version allows specification of the initial array size (`initialCapacity`) and the maximum acceptable load factor.

```java
public HashMap (int initialCapacity, float loadFactor);
```

Performance is improved if `initialCapacity` is a prime number. The specified load factor should be between 0 and 1. If the load factor of the table is exceeded, a new array is allocated, and the elements are rehashed into the new array. The size of the new array is twice the size of the previous, plus one.

A second constructor requires only the initial capacity be specified:

```java
public HashMap (int initialCapacity);
```

A default load factor of 0.75 is used in this case.

Finally, a Hashtable with default initial capacity of 101 items can be created by using a constructor with no arguments:

```java
public HashMap ();
```