Module 5

Introduction to Data Structures

Objectives:

1. Understand the importance of using inheritance when building a hierarchical set of data structures.

2. Familiarize yourself with the hierarchy of data structures that will be used in this course (you don't need to know implementation details about each structure yet).

3. Understand the difference between a traversable collection and a non-traversable collection.

4. Understand the difference between an ordered collection and an unordered collection.

5. Become familiar with the methods provided by the `Collection`, `TraversableCollection`, `OrderedCollection`, and `TraversableOrderedCollection` interfaces.

Textbook:

Java Interlude 2 (Exceptions).

Java Interlude 3 (More about Exceptions).

Java Interlude 7 (Inheritance and Polymorphism).

Assignment:

Complete the UML class diagram assignment for Module 5. This assignment should help you with the design for your semester project.

Lab:

No lab for this module.

Quiz:

No quiz for this module.

Pre-lecture Note:

We will spend most of the remainder of the semester talking about various data structures. One thing I want to make clear before we start is that every textbook will have a somewhat different take on how data structures should be organized and implemented. Unfortunately, few textbooks seem to use a consistent organization scheme throughout all the data structures they cover, which can lead to some confusion. The textbook we are currently using is no exception, but it has the advantage of being about the only textbook that separates data structure concepts from data structure implementation, which allows the reader to focus on the concepts, and refer to whatever implementations they prefer. To add to the confusion, the data structures that come packaged with the JDK are heavyweight structures that contain more than the basic functionality of adding, removing, and modifying items.

To minimize confusion, I will primarily assign only those chapters that cover a data structure's concepts, not those that cover the implementation. You are, of course, free to read the other chapters on your own, but if you get confused trying to keep track of two sets of data structure implementations, I recommend skipping those chapters.

Introduction

Data structures are essential components in every software project. All the data that are processed by a program must be stored somehow. The simplest form of storage is a simple variable that stores a primitive value, such as an integer, or a reference to an object. Since variables can only store one value at a time, they are not suitable when large amounts of data need to be stored. Arrays can be used to store collections of items, but arrays are not always efficient for certain operations, and they require contiguous blocks of memory, which may be difficult to obtain depending on how much memory is available and how fragmented it is. There are also times when access to data must be restricted, and the random access nature of an array allows full access to any item at any time.

By the term **data structure**, we are referring to an **abstract data type (ADT)** that is used to organize data in a particular way in order to promote greater efficiency for the operations that will be performed on the data. An abstract data type is defined as a collection of data and the operations that can be performed on those data, irrespective of the implementation. For data structures, the four primary operations of interest are:

1. adding new items
2. retrieving existing items
3. removing existing items
4. modifying existing items

Operations that are of secondary interest are:

- getting the total number of items in the data structure
- removing all items from the data structure
- returning whether or not the data structure is empty
- traversing the data structure
- sorting the items in the data structure

Ultimately what is needed is a variety of data structures that will accomplish the following goals:

- allow items to be stored using contiguous blocks of memory or fragmented memory;
- provide various mechanisms of data access control;
- provide a user-friendly interface for performing operations on items;
- utilize an inheritance hierarchy to allow new data structures, or new variants of existing data structures, to be developed, in order to minimize the amount of new code that must be written, reuse existing code that has been proven robust through testing, and maintain logical relationships between similar types of structures.

The last two points deserve some additional explanation. You’re all probably familiar with the traditional syntax for traversing an array:

```
for (int i = 0; i < intArray.length; i++)
{
    process(intArray[i]);
}
```

This syntax is potentially error-prone, since it is surprisingly easy to go beyond the bounds of the array if the starting and ending positions are not carefully defined, especially when they are defined by other variables. The Java language has introduced a new syntax for traversing arrays that allows you to traverse an array completely without specifying either the starting or the ending position:

```
for (int i : array)
{
    process(i);
}
```

Unfortunately, this syntax only allows for a complete traversal from beginning to end. If you need to start or end at different positions, this syntax will not work.

Additional problems arise when adding items. Arrays in Java are statically sized, meaning that once the size of an array has been declared, it cannot be changed. If it becomes necessary to add more items than an array will hold, a new, larger array must be created and the items must be copied from the old array to the new array. It is tedious to have to do a bounds check, and possibly resize an array, every time an item is added. It would be more user-friendly to have these operations encapsulated so that a developer can simply add and remove items without worrying about array resizing and bounds checking.

The last point takes into account the fact that all collections have a minimal number of operations in common: adding items, removing items, and modifying items. It is helpful to use inheritance to abstract these operations into a common entity and force each data structure to inherit and implement them according to its own data access restrictions. If this doesn’t make sense right now, it will become apparent soon. Using inheritance will also allow for the standardization of these operations so the developer will have a consistent interface to work with.

Internal Storage Mechanisms

There are really only two ways to store a collection of items and keep them associated with each other:

1. Contiguous memory
2. Linked nodes

You already know that arrays use a block of contiguous memory in which to store items. Because all the items are contiguous, it is easy to perform random access of items, since the memory location of any item can be calculated given the memory address of the first item and the data type of the items. Since arrays cannot store mixed types, all the items will be of the same data type, so each item will occupy the same number of bytes. The primary advantage of an array is the guaranteed O(1) time random access of any item in the array. There are several disadvantages to using contiguous memory blocks. One is the fact that the size of block, and hence the number of items that can be stored in it, is limited by the amount of physical memory available, as well as the degree to which available memory has been fragmented by other software. Another is the fact that adding items may at some point involve allocating a new block of memory, and copying all the items from the old locations to new locations. A third disadvantage is the potential for wasted space. If an array is not full, the unused space cannot be used for anything else.

Linked nodes avoid some of the disadvantages of using contiguous memory. Each node is a structure that contains a single data item, along with a reference to the next node in the collection. In some cases, references are also maintained to previous nodes. There is no requirement for the nodes to be stored in contiguous memory, so a collection stored using linked nodes can be stored even in fragmented memory, and there is no problem of wasted space. Adding items will never involve allocating new blocks of memory and copying all the items from one location to another. There is a catch, however: a linked node structure is limited to sequential access, since the only way to access an arbitrary item is to start at one end and follow node references until the desired item is reached.

All higher-level data structures must use one of these mechanisms for storing its items.

Type Safety

In order for a data structure to be useful, it must be allowed to store any type of data; i.e., any object type. In Java, all objects implicitly inherit the Object class, so if a data structure is built to store an internal array of type Object, or linked nodes that store references to type Object, any object that is created can be stored in the data structure. There is a problem with this, though. Since all objects created in Java are derived from the Object class, if a data structure is constructed to store references to the Object class, it is possible to add a mixture of object types to a single data structure. For example, the same data structure may contain some Employee objects, some Student objects, and some Car objects. This can cause a problem when processing the items in the data structure, since the programmer must know beforehand which data types are stored, and cast to the appropriate data types as needed. Although this can be done, it tends to make the code clumsy and difficult to maintain.

With version 1.5, the JDK introduced parameterized data types. Parameterized data types force the storage of a single object type, which eliminates the problem of determining which data type a particular item is when retrieving it from a data structure. The data type must always be specified at the time a data structure is created; references to the generic Object class are no longer used, except in rare circumstances. An obvious question arises at this point: if the data type must be specified at the time a data structure is created, doesn't that mean it would be necessary to implement a different version of a data structure for each different data type? The answer is no, because the Java language provides for the creation of templates that use "placeholder" data types. The placeholders can be replaced with any data type at the time a data structure is declared. (*Note: for those of you knowledgeable about C++, these are not equivalent to C++ templates*). With parameterization, casting is no longer necessary when processing the items in a collection.

One final thing to note about type safety is that parameterized types can be used without specifying any data types, if desired, for those cases where a mixture of data types might need to be stored in the same data structure. Some IDE's, such as Eclipse, will show you a warning for every instance where this occurs, but will still usually compile the code unless you have set the options for the IDE to do otherwise.

Mechanisms of Data Access

I've alluded to the importance of having different mechanisms of accessing the items in a collection, but I haven't mentioned what those mechanisms are. Briefly, the primary ways in which data access can be controlled in a data structure are:

1. Sequential (Lists, Trees)
2. Random Access (Hashtables, HashMaps, and other structures that use key-based access; Heaps)
3. Last In First Out (LIFO) (Stacks, Deques)
4. First in First Out (FIFO) (Queues, Deques)
5. First Out based on some value that can change over time (Priority Queues)

Some data structures, such as the graph, are designed not to improve the efficiency of adding or removing items, but to accomplish some other purpose, such as performing searches or finding shortest paths. For some data structures, having the ability to traverse the items is important, but for others, traversal is prohibited. For example, stacks, queues, and deques are designed to only allow access at either end of the structure.

Architecture

When designing the architecture of data structures, one of the primary considerations is that the structures should be reusable. No one wants to have to reinvent the wheel every time they need a data structure, so it makes sense to design data structures in a way that they can be placed into a component library and used when needed. Also, since many data structures share certain functionality, it is reasonable to try to design a collection of data structures in a way that utilizes inheritance to abstract common functionality into a base class, and place specific functionality in derived classes. This will lead to a hierarchy of structures that minimizes repetitious code, minimizes code faults, allows for the creation of a consistent user interface, and promotes extensibility, meaning new data structures can be easily added into the existing hierarchy.

There is certainly more than one way to go about designing a hierarchy of data structures. The Java Development Kit comes prepackaged with a reasonably complete set of data structures, which do extensively use inheritance. The authors of your textbook provide their own rendition of a data structure hierarchy, although they don't follow through in all the structures they present. For this class, I will provide a third alternative, in which I try to achieve the following goals:

1. Utilize inheritance throughout all the structures; i.e., do not leave any structures orphaned from the rest.

2. Keep the overall hierarchy relatively simple, for ease of understanding.

3. Focus primarily on the algorithms for the various operations, rather than code.

I want to emphasize the last point above. If you look at 10 different data structures texts, I can almost guarantee you will see 10 different ways to implement data structures. Different authors will use inheritance in different ways, and there will certainly be some difference in the source code. However, the fundamental algorithms will be common in all cases. The point is that I want everyone to learn the characteristics of the various data structures and the algorithms behind the major operations, not memorize the source code. If I ask you to write a method for adding an item to a linked list, for example, you should understand the algorithm well enough to write the method in any programming language you know.

The Collection Hierarchy

Figure 5.1 shows the hierarchy of unordered, non-traversable data structures we will look at in this course. We will cover three data structures whose access requirements prohibit traversal: the stack, the queue, and the deque. Stacks are structures in which the next item that can be accessed is the one that was most recently added (Last In First Out, or LIFO). With queues, the next item that can be accessed is the one that was least recently added (First In First Out, or FIFO). Deques (pronounced "decks") behave as a hybrid of a stack and a queue. In a deque items can be accessed at either end.

The `Collection` interface will be described in this module. Stacks, queues, and deques will be covered in Module 7.

The `Collection` Interface

The most basic functionality we will use that is common to all data structures, is:

- add an element to the collection
- remove an element from the collection
- get an item from the collection
- empty the collection
- return whether or not the collection is empty
- return the size of the collection

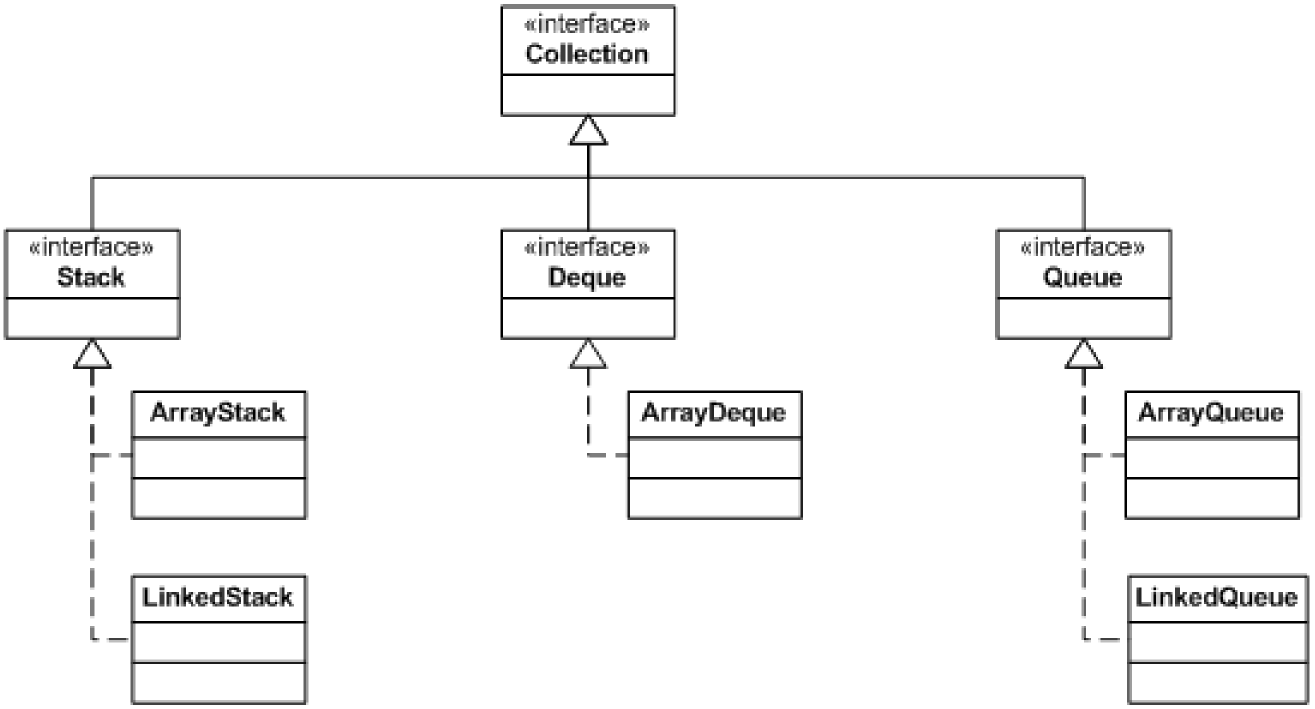

Figure 5.1. The `Collection` interface, and the data structures derived from it.

We can encapsulate this functionality in an interface, which I will call `Collection`, to follow the unofficial standard used by JDK and most authors. An interface does not contain any implementation, but it enforces a requirement on implementing classes to implement the functionality represented in the interface. Every class that is derived from `Collection`, therefore, must provide implementations for all the methods in `Collection`. Since this functionality may differ for different data structures, each derived data structure can provide its own implementation according to the requirements of that data structure.

Often you will see the use of an abstract class—often called `AbstractCollection`—that provides default functionality for a very generic collection. Since the class is abstract, it cannot be instantiated, but other data structures can be derived from it. I have chosen not to include such a class, for the following reasons:

1. It adds another level of complexity to the hierarchy, and I want to keep the hierarchy as simple as possible.
2. Usually, such general, default implementations must be overridden in order to promote efficiency, which renders the default code somewhat redundant.

The basic `Collection` interface is shown below:

```
public interface Collection
{
    public boolean add(Object obj); // add an item to the Collection

    public boolean remove(); // remove an item from the Collection

    public void clear(); // empty the Collection

    public int size(); // return the number of items currently in the Collection

    public boolean isEmpty(); // return whether or not the Collection is empty

    public Object get(); // retrieve an item from the Collection
}
```

Below, I show what the Collection interface shown above would look like if it were to be made type safe using parameterization:

```
public interface Collection<E>
{
    public boolean add(E obj); // add an item of type E to the Collection

    public boolean remove();   // remove an item from the Collection

    public void clear();       // empty the Collection

    public int size();         // return the number of items currently in the
                               // Collection

    public boolean isEmpty();  // return whether or not the Collection is empty

    public Object get();       // retrieve an item from the Collection
}
```

Note there are only two modifications: 1) the string "<E>" is appended to the interface declaration; and 2) in the add method, instead of specifying the parameter as a reference to Object, we use a reference to E. Remember that E can be a reference to anything, but once its type has been set in a variable declaration, for that particular collection, E must always be a reference to that type. For example, suppose we have an ArrayList<E> class that implements the Collection<E> interface, and we instantiate an ArrayList<E> as follows:

```
ArrayList<Student> studentList = new ArrayList<Student>();
```

For the variable studentList, the generic type E has been replaced by the type Student. Therefore, only objects of type Student may be added to studentList.

Interface **TraversableCollection**

The **TraversableCollection** interface extends **Collection** to allow two additional elements of functionality:

1. the ability to traverse the items in the collection
2. the ability to determine whether or not a given item is present in the collection

Determining whether or not an item is present requires the ability to traverse the entire collection of items, which is why this capability is included here and not in **Collection**. An Iterator provides a way to traverse the items of any data structure, regardless of how that data structure is implemented. We will have more to say on iterators when we cover lists in Module 6.

The **TraversableCollection** interface is shown below:

```
public interface TraversableCollection extends Collection
{
    public Iterator iterator();          // return a generic Iterator for the
                                         // Collection

    public boolean contains(Object obj); // return whether the Collection contains
                                         // the specified item
}
```

Here is a parameterized version:

```
public interface TraversableCollection<E> extends Collection<E>
{
    public Iterator<E> iterator();  // return a generic Iterator for the Collection

    public boolean contains(E obj); // return whether the Collection contains the
                                    // specified item of type E
}
```

Figure 5.2 shows the architecture for the traversable collections we will cover in this course.

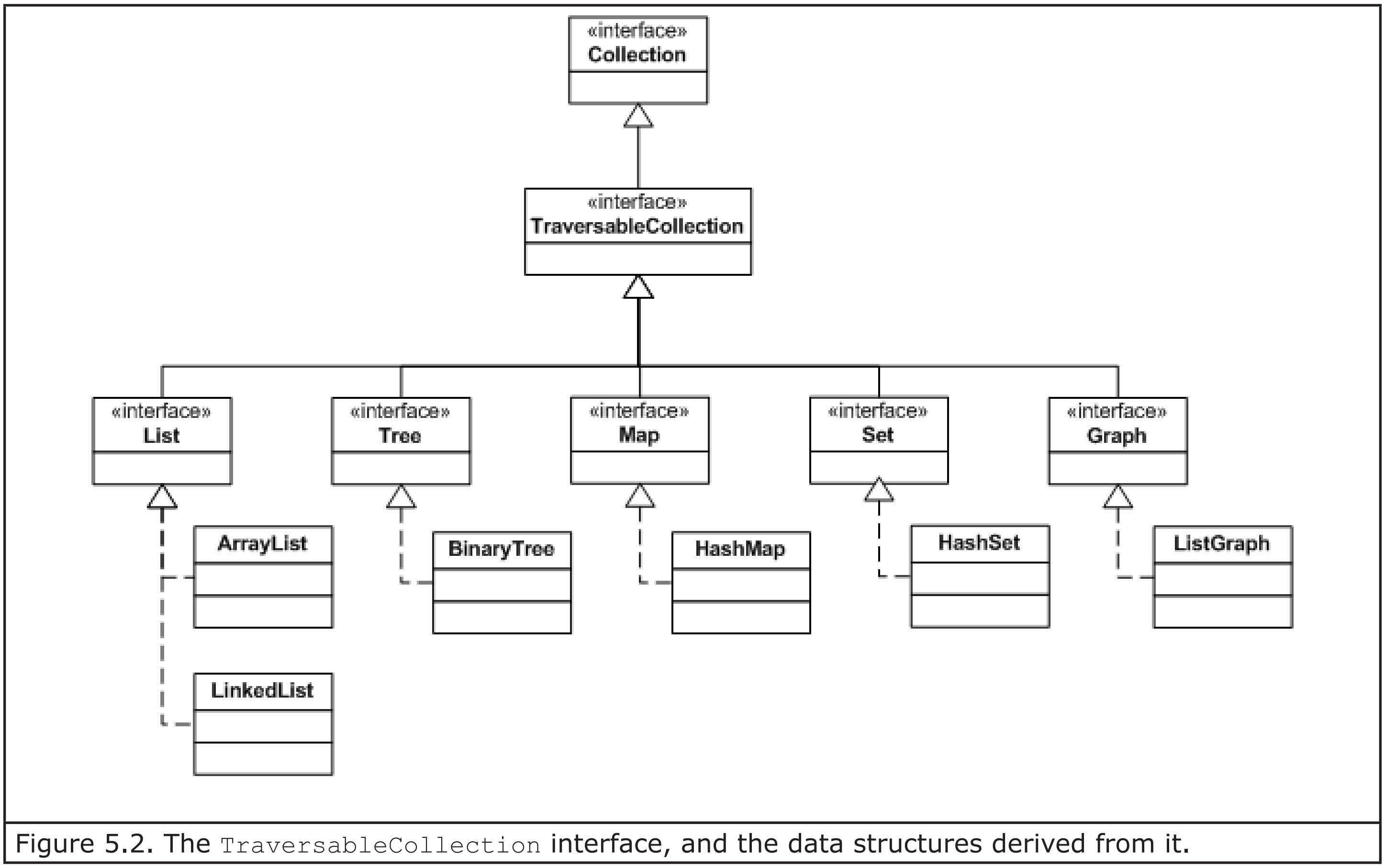

Figure 5.2. The `TraversableCollection` interface, and the data structures derived from it.

Lists, Trees, and Graphs

You are probably already familiar with lists. In a list, each item is associated with a position in the list relative to the other items. There are two common ways to implement a list: 1) using an array, and 2) using linked nodes. Trees are usually represented as a linked structure in which one node is designated as the root, and all subsequent nodes branch off the root or its children. The bottommost nodes are referred to as leaves. The linkage between nodes is usually one-way, so it is easy to travel from the root down, but it's not so easy to travel from the leaves up. Trees can be very complex structures depending on how many child nodes are allowed. Graphs also make use of an underlying linked structure, but in a graph there is no limit to how nodes can be connected, and each connection can have an associated cost. Graphs are useful in problems where the shortest path must be determined. Lists, trees, and graphs are all traversable structures.

Sets, Maps, and Hashtables

Maps and hashtables are examples of a more generalized type of data structure known as a dictionary. Items in dictionaries are associated with keys, and can only be accessed via those keys. Hashtables are unique in that the items are stored in an underlying array, and the keys are derived from the items themselves. A hashing function performs this derivation, and converts the results to non-negative integers that specify the array indexes where the items are stored. Sets have the unique characteristic of disallowing duplicate items. Sets, maps, and hashtables are all traversable structures. However, when traversing these structures, especially hashtables, there is often no guarantee as to the order in which the items will be traversed. This contrasts with structures such as lists and trees, where the items can be traversed in a defined order.

The `OrderedCollection` and `TraversableOrderedCollection` Interfaces

The data structures described above can store collections of items, but they do not provide any way to maintain the items in sorted order. Although collections may be sorted using algorithms we discussed in Module 4, sometimes it is more efficient to maintain the items in sorted order in the data structure. Whether to maintain items in an ordered structure, or to maintain them in an unordered structure and sort them as needed, depends on several factors, such as the number of items, how frequently new items are added and removed, and how frequently searches are performed.

In an ordered data structure, new items are added precisely where they belong in the ordering. Since it isn't always necessary to maintain items in an ordered fashion, a separate, top-level interface, `OrderedCollection`, is used for those structures where ordering is desired. This interface provides the same essential functionality as the `Collection` interface. The difference is in the data type of the items. For Collection, any object can be used. For `OrderedCollection`, all objects must be of type Comparable. Recall from Module 3 that Comparable objects must provide the capability to compare two objects for both equality *and* inequality. It is the ability to test for inequality that allows for items to be sorted. The `TraversableOrderedCollection` interface is analogous to the `TraversableCollection` interface. Again, the same basic functionality is provided, the only difference being that the data must be of type `Comparable`. Figure 5.3 shows a diagram of the architecture we will be using for ordered collections.

Data Structures Provided by the Java Development Kit

I should say a few words about the data structures that come packaged with the JDK, even though we will not be covering those structures specifically. The JDK is continually evolving. Early versions of the JDK library were sparse in the way of data structures. Over time, more data structures have been added. The JDK provides implementations for many of the data structures we will be discussing, plus several we will not cover. These data structures are all found in the java.util package. Figure 5.4 lists the data structures currently provided by the JDK library. These data structures follow a slightly different hierarchy in terms of organization from the hierarchy I will be presenting in this course. The implementations of the JDK data structures are considerably more complex as well, including capabilities such as serialization to facilitate the transfer of data across data streams. JDK also provides a `Collections` class and an `Arrays` class, which contain several useful static methods such as `binarySearch` and `sort`.

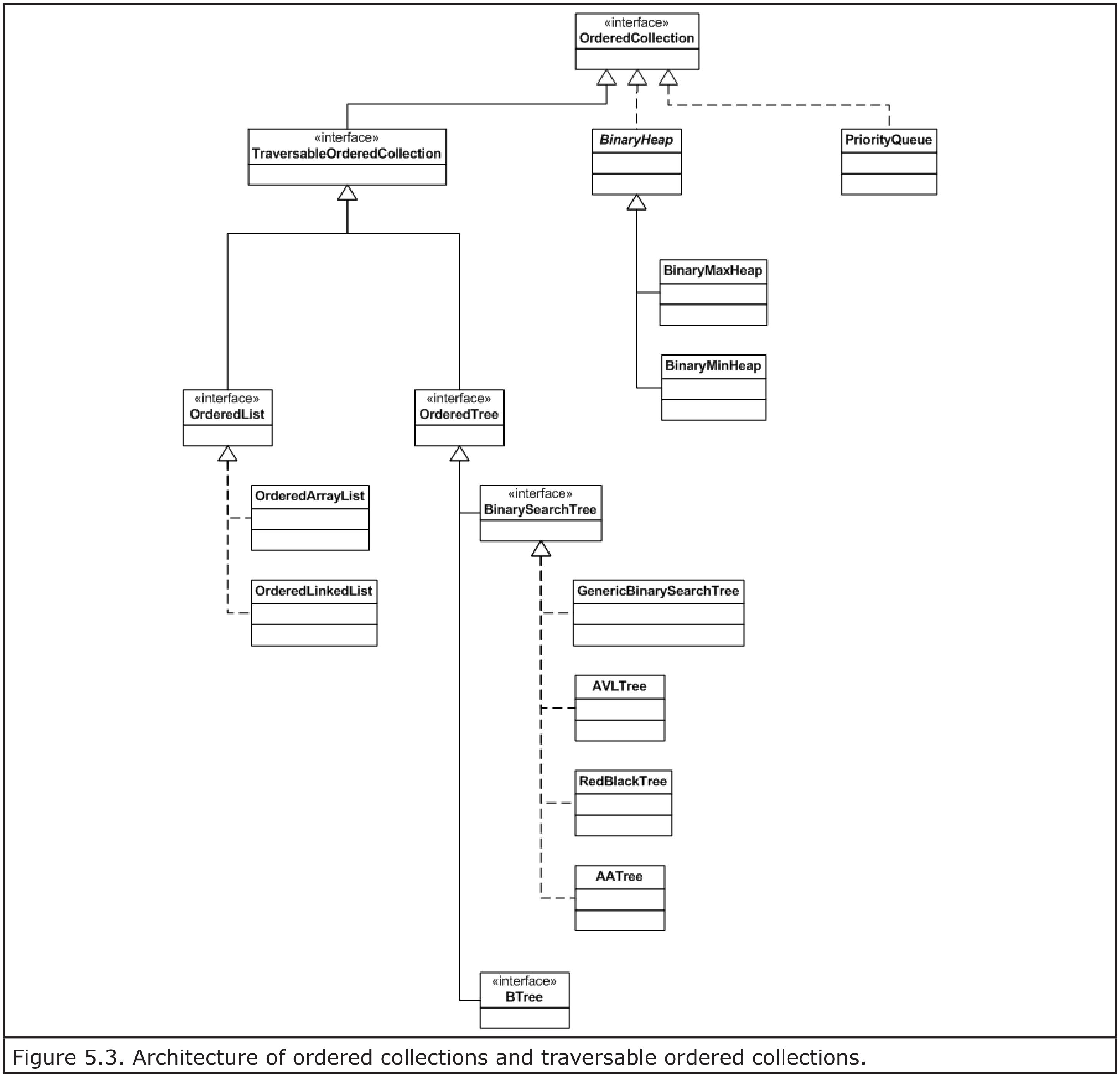

Figure 5.3. Architecture of ordered collections and traversable ordered collections.

Interfaces	Classes
Collection	AbstractCollection
Deque	AbstractList
Iterator	AbstractMap
List	AbstractQueue
ListIterator	AbstractSequentialList
Map	AbstractSet
NavigableMap	ArrayDeque
NavigableSet	ArrayList
Queue	Dictionary
Set	EnumMap
SortedMap	EnumSet
SortedSet	HashMap
	HashSet
	Hashtable
	IdentityHashMap
	LinkedHashMap
	LinkedHashSet
	LinkedList
	PriorityQueue
	Stack
	TreeMap
	TreeSet
	Vector
	WeakHashMap

Figure 5.4. The data structures provided by the JDK.