Module 12

Binary Heaps and the Priority Queue

Objectives:

1. Understand what the difference is between a full binary tree and a complete binary tree.
2. Understand how a complete binary tree can be implemented using a simple array.
3. Understand what a binary heap is, and know the difference between a min heap and a max heap.
4. Understand what a priority queue is, how it can be useful, and how it can be efficiently implemented using a binary heap.

Reading:

Chapter 27 (Heap Implementation).

Chapters 7, 8 (only sections on priority queues)

Assignment:

No assignment for this module.

Quiz:

Take the quiz for Module 12.

Lab:

No lab for this module.

Full Binary Trees and Complete Binary Trees

Recall that in a binary tree every node can have a maximum of two child nodes, but there is no requirement that any node in the tree must have two child nodes. If every node in a binary tree, except for the leaf nodes, has two child nodes, the tree is known as a **full tree**. A **complete tree** results when every level is full with the possible exception of the level containing the leaf nodes, and the leaf nodes are filled from left to right. Figure 12.1 illustrates each type of binary tree.

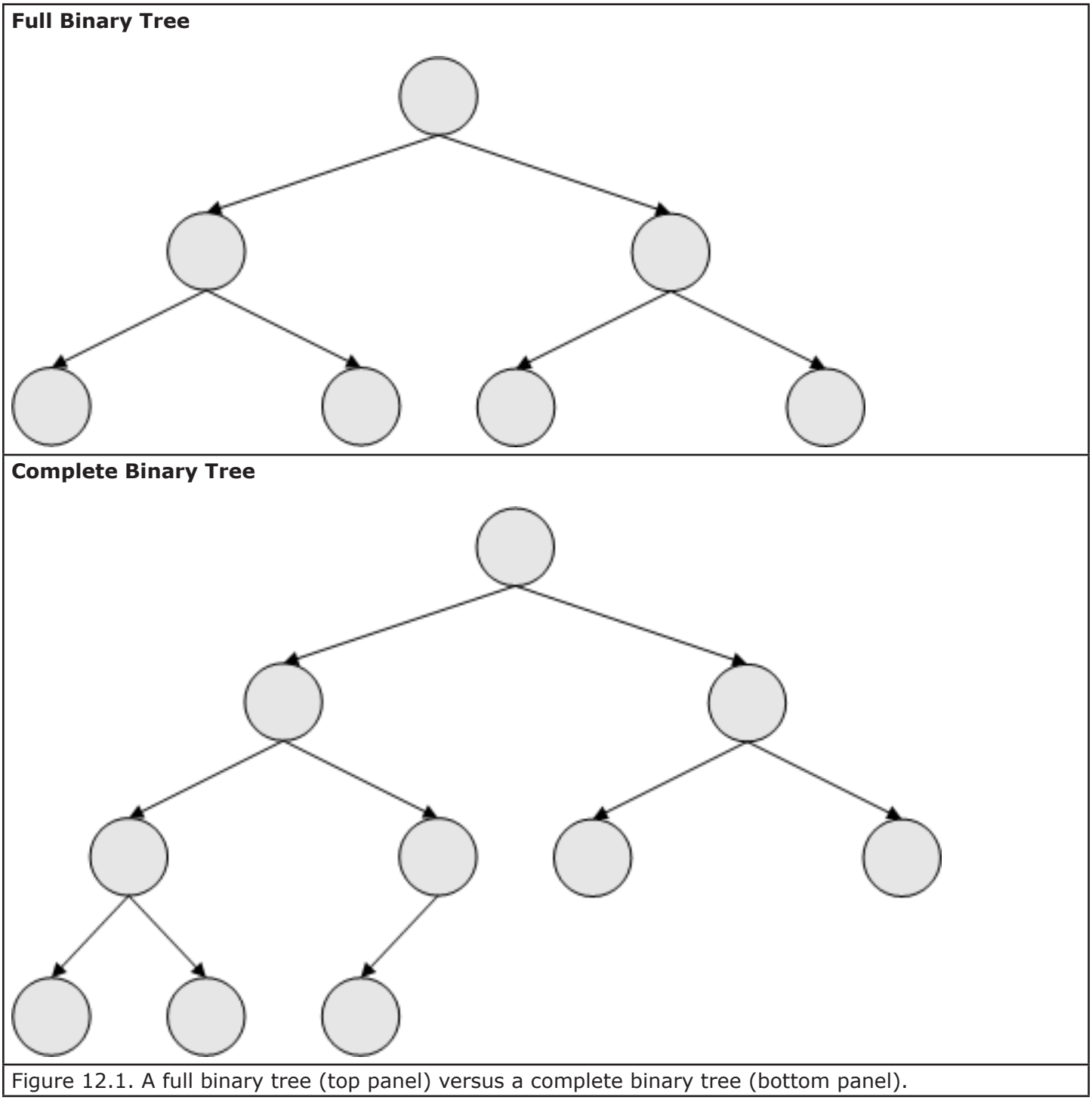

Figure 12.1. A full binary tree (top panel) versus a complete binary tree (bottom panel).

An interesting property of a complete binary tree is that it can be implemented using a simple array, without any need for each node to contain references to its child nodes. The index of any given node in the tree can be computed mathematically. Figure 12.2 shows a complete binary tree in which each node has been assigned an array index. The indexes were assigned by traversing the tree in a level-order fashion. Figure 12.3 shows what the array representing this tree would look like. For any given node, X, in the tree, the index of the node's left child is given by the formula, *index(left child) = index(X) * 2*. The index

of the right child is given by the formula, *index(right child) = index(X) * 2 + 1*. In order for these formulae to work it is essential that index 0 not be used to permanently store any items in the tree. It is also required that the root node occupy the array cell at index 1. Because an array is used as the underlying implementation, any item can be accessed in O(1) time via its array index.

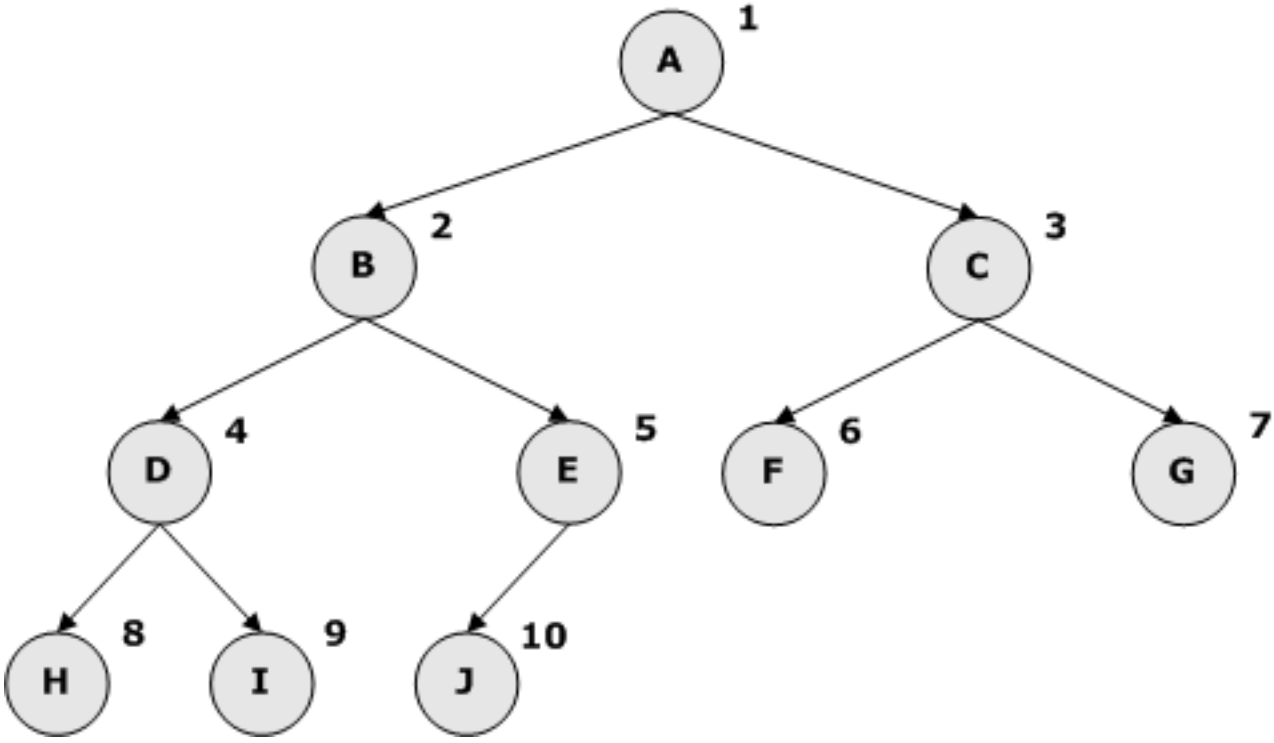

Figure 12.2. A complete binary tree, with nodes indexed.

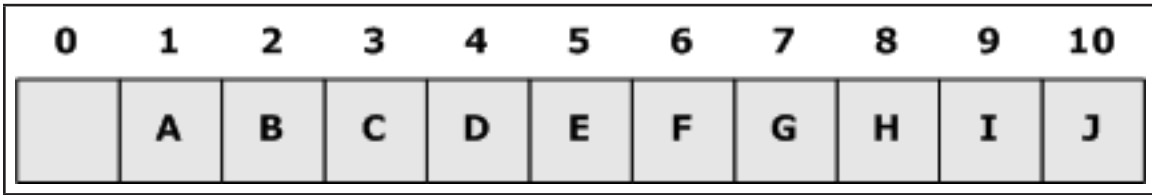

Figure 12.2. A complete binary tree, with nodes indexed.

Binary Heaps

A binary heap is a complete binary tree on which an ordering on the items is imposed. There are two possible orderings that can be used:

1. The item in any given node is less than or equal to the item in either child node.
2. The item in any given node is greater than or equal to the item in either child node.

If the first ordering is used the result is known as a **min heap**; if the second ordering is used, the result is a **max heap**. Based on these rules we can observe that finding the minimum item in a min heap is a O(1) operation, and that the minimum item must be in the root node (array index 1). Likewise, finding the maximum item in a max heap can be done in O(1) time, and it follows that the maximum item must be in the root node.

If an efficient implementation is used, adding an item to the heap can be accomplished in O(1) average time, with a worst case of O(log N). Removal and retrieval of items in a min heap or max heap is usually restricted to the minimum item or maximum item, respectively. Therefore, retrieval can be done in O(1)

time. Removal takes O(log N) time in the worst case, since once the item in the root node is removed some work must be done to ensure the next smallest or largest item is placed in the root node, and that the heap still obeys the structure property of a complete binary tree.

Binary heaps are commonly used in the implementation of the heapsort sorting algorithm. Heapsort has a worst case running time of O(N log N), which makes it a rival for quicksort in terms of speed. Another common use for binary heaps is in the implementation of priority queues, which are discussed later.

Implementation of an Abstract Class, `BinaryHeap`

Since min heaps and max heaps are for all practical purposes the same, except for their ordering properties, we can capture the commonality of a binary heap in an abstract class, and derive each type of heap from the abstract class. The abstract class, `BinaryHeap`, that I present, implements the interface `OrderedCollection`. The source code for this class, as well as the other classes discussed in this module, can be found in the Source Code section of Blackboard.

Attributes

We have two attributes: the underlying array of Comparable objects, theItems, and theSize, which stores the current size of the heap. There are also two constants: `DEFAULT_CAPACITY` and `NEXT_ITEM`. `DEFAULT_CAPACITY` stores the default array size for newly created heaps. `NEXT_ITEM` stores the index of the next item to be returned. Since this will always be either the largest or the smallest value in the heap, it will always be stored in array index 1.

Abstract Methods

There are two abstract methods: `boolean add(Comparable obj)` and `boolean remove()`. The implementations of these methods depend on the ordering used for the heap.

Other Methods

The other methods of `BinaryHeap` are fairly straightforward. Since the implementations of these methods are identical for each type of heap, the implementation is provided in the abstract class. Since retrieving an item will always return the value in the root node, the `get()` method can be implemented in this class. It simply returns the item located at the index, `NEXT_ITEM`. One other thing to note is the implementation of the `doubleArray()` method — copying items from the old array to the new array begins at index 1 instead of index 0, since index 0 is not used for permanent storage.

Implementation of the `BinaryMinHeap` Class

The `BinaryMinHeap` class that I present represents a min heap.

Adding an Item Using the `add(Comparable obj)` Method

The only tricky part about adding an item to the heap is maintaining the ordering of the items such that the ordering rule for a min heap is obeyed. First, of course, a check must be made to see if the array needs to be resized. The value of `theSize` is incremented, and an index (appropriately named index) is initialized at the bottom of the heap. The item to be added is placed in temporary storage at array index 0. A loop is executed in which the value of index is repeatedly halved. As long as the newly added item is less than the item at `index/2`, the item at `index/2` is shifted down to occupy the location marked by the current value of index. When the loop ends, index will contain the correct location for the new item. Although some shifting of items is required when adding an item to a heap, due to the structure of the array, at most log N items will need to be shifted, so the worst case running time for adding an item is O(log N). As with all the array-based data structures we've discussed, the running time required for

resizing the array is amortized over time, and does not factor significantly into adding an item to a large array.

Removing an Item Using the `remove()` Method

Removal is restricted to the minimum item. The minimum item is logically removed by replacing it with one of the larger values in the heap. The next smallest item in the heap is then shifted up into the root node. This procedure is complex enough to warrant placing it in its own method, `downshift(int index)`. The average running time for removing an item is O(log N), and is dominated by the running time of the `downshift` method.

Implementation of the `BinaryMaxHeap` Class

This class represents a max heap. The only thing that distinguishes this class from **`BinaryMinHeap`** is the ordering imposed on the items. As far as the implementation is concerned, this is accomplished by simply changing all instances of the '<' operator to '>'.

The Priority Queue

Suppose we are writing a module for an operating system that is responsible for determining when each of the processes that are currently active gets to use the CPU. We could use a standard queue, and allow the processes to cycle in a round-robin fashion, with each process given the same amount of CPU time before it must be placed at the back of the queue. In many cases this would be acceptable. But suppose we have a process that must be allowed to run at times that may not coincide with the time intervals permitted by the queue. We need a way to ensure this process will always have access to the CPU as needed. Essentially we need a way of telling the operating system that this particular process should have priority over other processes. In order to accomplish this we need to be able to associate each process with a priority value. Priority can be associated with either higher values or lower values. Higher values are usually used when an object's priority is artificially generated. Lower values may be used when the value of a particular attribute of an object is used as the object's priority value, and lower values for that attribute correspond to higher priority. Either way, we need a data structure that is capable of storing items such that the item with the maximum (or minimum) value can always be efficiently retrieved. Later in the course we will encounter another situation where a priority queue is used: Dijkstra's shortest path algorithm.

In all the data structures we have discussed throughout this course, only two provide an efficient way of returning the maximum (or minimum) value: the balanced binary search tree, which requires O(log N) time, and the binary heap, which requires O(1) time. Clearly, if we want the fastest runtime, the binary heap is the data structure of choice. (Although hashtables also offer O(1) retrieval times, on average, they do not have any way of guaranteeing that the item returned will be the maximum or minimum item). We can construct a new class, PriorityQueue, that uses a binary heap as its underlying implementation. We can use either a min heap or a max heap. The choice is irrelevant so long as we use the one that is appropriate for our definition of priority. If we define highest priority as being the maximum value, we would use a max heap. If we define highest priority as having the minimum value, we would use a min heap. Historically, it seems as though there is a bias towards using min heaps, but the reasons for this preference are not clear.

The functionality required for a priority queue is relatively minimal. In fact, we could just use a binary heap itself as a priority queue, but it would probably be beneficial to have a separate PriorityQueue class, constructed in such a way that it can accept either a min heap or a max heap as the underlying structure, to provide versatility.

Implementation of the `PriorityQueue` Class

The implementation of the `PriorityQueue` class that I have provided is extremely simple, and is barely more than a wrapper class for a binary heap. It only accepts items of type `Comparable`, and the priorities of the items are based on the values used for the natural ordering of the items as defined in the

`compareTo` method. This is fine for illustrating how a priority queue works, but in practice it would be better to have a way of determining priority that does not depend on the compareTo method. There are a couple of straightforward ways this could be done:

1. Items to be stored in a priority queue could be required to implement an interface (e.g., `Prioritizable`) specifically used for assigning and comparing items based on a given priority value.

2. Items could be wrapped in an object that contains functionality for assigning and comparing priority values.

The first alternative is more efficient when using Java, since Java allows unlimited extension via interfaces. The second alternative involves an additional layer of overhead, since wrapper methods must be used to retrieve the contents of the actual data items.

Attributes

There is one attribute, `theQueue`, which stores a reference to a `BinaryHeap`. This can be either a `BinaryMinHeap` or a `BinaryMaxHeap`. There are also two publicly accessible constants, `PRIORITY_ASCENDING` and `PRIORITY_DESCENDING`, which can be used in the constructor to specify the ordering of items in the heap.

Constructor

By default, a `BinaryMaxHeap` is used as the underlying structure. However, using one of the constants mentioned above, it is possible to specify the ordering of the items in the heap. If we define highest priority as being the priority with the minimum value, we need the underlying structure to be a min heap, whereas if we define the highest priority as being the priority with the maximum value we need the underlying structure to be a max heap.

Methods

Implementation of the methods is accomplished by simply calling the appropriate method of the underlying binary heap.