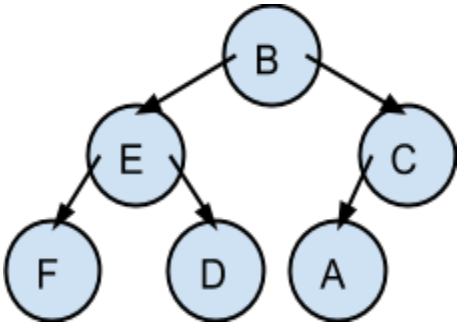
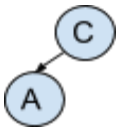
Caspar Lant
Data Structures - Project 6

Determining Tree Ordering:

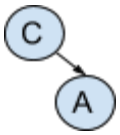


We know that ‘B’ is the root of this tree because it is the first term in output of the preorder traversal. Nodes ‘F’, ‘E’, ‘D’ must be in the left hand subtree and ‘A’ and ‘C’ must be in the right subtree because of their respective positions in the inorder traversal. Because “F” precedes ‘E’ and ‘D’ in the preorder traversal, we can surmise that ‘E’ is the “root” of the left subtree. By this logic, ‘C’ must be the root of the right subtree. ‘A’ precedes ‘C’ in the inorder traversal and is therefore its lefthand child.

At least two traversals are required to determine the ordering of an unsorted binary tree structure. Assuming that the tree is complete, any two traversals will determine the ordering. Completeness is imperative because the positioning of a leaf and its parent is ambiguous if the leaf does not have a sibling. This claim can be tested by traversing such a structure (see figures below). Trees need not be complete if an inorder traversal is provided. To determine the shape of a sorted tree only a preorder or postorder traversal is required. A single inorder traversal is not enough to determine the shape of a (sorted) binary search tree.




```
inorder:   AC
preorder:  CA
postorder: AC
```




```
inorder:   CA
preorder:  CA
postorder: AC
```

As you can see, the results of the preorder and postorder traversals for each tree are identical

Analysis of Java’s PriorityQueue Class:

A. PriorityQueue.java contains a whopping seven Constructors, but five of them call `PriorityQueue(int, Comparator<? super E>)`. I imagine ‘`?`’ is some superset of `E`, a generic object. The implementation of so many constructors allows flexible use of the class.

B. The array data field in this class is called queue and is of type `Object[]`. The queue is a linear representation of a balanced binary heap. The default initial capacity of the array is 11, but it can be explicitly be set when the array is instantiated. Multiple constructors allow for this feature. It may not have a capacity of less than one, as verified in the constructor with signature `PriorityQueue(int, Comparator<? super E>)`. The `grow()` method uses the conditional ternary operator to determine an optimal change in array size. If the capacity of the array is less than 64, the size is increased by a factor of 2, plus 2 (in other words, the size of the array is doubled, and this new capacity is increased by two. If the size of the array is greater than 64, the infrequently used signed right shift operator comes into play. This unusual operator shifts the bits in a given number to right right by some declared value. calling `>> 1` on any integer value is nearly identical to dividing it in half. My suspicion is that this operation is slightly faster than calling the division operator. This value is then added to the current length of the array, effectively increasing it by 1.5. The use of this `(arraysize < 64)` conditional prevents quadratic growth of the array.

C. The `transient` keyword marks an object that is not to be serialized. Serialization is a method through which objects are converted to bytes and written to a machine's local storage, so as not to be lost when memory is reallocated. De-serialization is the process of retrieving this data. It can be said that these transient objects are "intentionally lost", hence the name transient. Per the inline comments, the queue array is marked as `transient`, rather than `private`, to allow for nested class access.

D. Methods involved in adding to the queue:
   - `add(E)`
   - `offer(E)`
   - `grow(int) → Arrays.copyOf(E[])`
   - `siftUp(int, E)`
   - `siftUpUsingComparator(int, E) → Comparator.compare(E, E)`
   - `siftUpComparable(int, E)`

E. Methods involved in polling the queue:
   - `poll()`
   - `siftDown(int, E)`
   - `siftDownUsingComparator(int, E)`
   - `siftUpComparable(int, E)`

F. New to Me:

```
108 : private final Comparator<? super E> comparator
```

When implementing a generic class or method, it is often necessary to use the syntax `<myObject extends Comparable<E> >`. This states that `myObject` is a subclass of `E`, and ensures that `E`'s methods (namely `compareTo(E)`) are inherited. In this context, the keyword `super` sets the opposite relationship between `?` and `E`. That is, `?` is a superclass of `E`. This is useful in instances in which values of type `E` must be interpreted by a collection of type `?`, or when `E` need not be comparable to itself, but only one of its `supertypes`. Usage of this functionality can be seen on line 192 in `PriorityQueue.java`, where objects of type `SortedSet<? extends E>` are type-casted to `Comparator<? super E>`. To ensure compatibility with `PriorityQueue`'s `comparator` field. In this instance, if `Comparator<E>` would be too restrictive. We want to be able to fill a priority queue with various children of the same ancestor.

```
28 : import java.util.function.Consumer
```

According to the Java documentations on Oracle's website, `Consumer.java` is part of the `funtional interfaces` package new to Java SE 8. `Consumers` represent a function that accepts a single argument of arbitrary type and produce no result[1]. Another class, `Suppliers`, act in conjunction with `Consumers`. `Suppliers` represent a function that accepts no arguments and produce a result of some arbitrary type. No invocations of `Supplier` can be found in the `PriorityQueue` Class, because the "arguments of arbitrary type" that `Supplier` would otherwise provide are produced by other methods.

Analysis of Java's Array.Sort Class:

The sorting algorithm implemented in Java's native Arrays class goes by the name `DualPivotQuickSort`. This algorithm, in turn, implements four additional sorts, dependent on what I imagine are experimentally-derived thresholds of array length. Various implementations of the same algorithm are included to allow for differing datatypes.

[1] Excerpted from http://www.byteslounge.com/tutorials/java-8-consumer-and-supplier

It is interesting that this (seemingly redundant implementation) is done rather than allowing for generics. My feeling is that it is done to mitigate the time lost by calling a generic class' `compareTo()` method repeatedly. The sort works with data types `int, long, short, char, byte, float,` and `double`.

Of the sorts we learned in class this semester, quicksort is arguably the quickest. More intuitive insertion sort is faster in some circumstances (namely in the case of small arrays), which this implementation of dual pivot quicksort accounts for.
Like classical quicksort, dual pivot quicksort employs the "divide and conquer" approach to sorting. Because more divides are made to each subarray, it can be concluded that the "conquering time" is diminished, producing significant time savings. Simply put, two pivots are better than one.
Interestingly, dual pivot quicksort works very well on nearly-sorted arrays. Through my own experimental observations, the computation times for quicksort vs. the dual-pivot strategy can vary by a factor of two! In both average and worst-case situations, the time complexity for both classical and dual-pivot implementations is O(nlogn), but their coefficients differ significantly. The obvious disadvantage of dual pivot quicksort is its complexity; it is very difficult to conceptualize, and even harder to translate into parsable pseudocode!

Below is the pseudocode for the **recursive** implementation of this algorithm, it appears iteratively in the java library, but I felt that the recursive approach would be more effective in conveying the structure of the algorithm (especially how they differ from those in classical quicksort).

```
void dualPivotQuicksort(array,  leftPivotIndex, rightPivotIndex,  numDivisions) {

    length = rightPivotIndex - leftPivotIndex
    if (length < maximum_threshold_for_quicksort) { // insertion sort for tiny array
        insertionSort(array)
        return;
    }

     oneDivison = length / numDivisons

     median1 = leftPivotIndex + third
     median2 = rightPivotIndex - third

    if (median1 <= leftPivotIndex)
        median1 = leftPivotIndex + 1

    if (median2 >= rightPivotIndex)
```

```
        median2 = rightPivotIndex - 1

    if (array[median1] < array[median2]) {
        swap(array, median1, leftPivotIndex)
        swap(array, median2, rightPivotIndex)
    }

    else {
        swap(array, median1, rightPivotIndex)
        swap(array, median2, leftPivotIndex)
    }

     pivot1 = array[leftPivotIndex]
     pivot2 = array[rightPivotIndex]

     firstElementOfMiddlePartitionIndex = leftPivotIndex + 1;
     lastElementOfMiddlePartitionIndex = rightPivotIndex - 1;

    //Sorting, finally
    for ( currentIndex = firstElementOfMiddlePartitionIndex; currentIndex <=
lastElementOfMiddlePartitionIndex; currentIndex++) {
        if (array[currentIndex] < pivot1){
            swap(array, currentIndex, firstElementOfMiddlePartitionIndex)
            increment firstElementOfMiddlePartitionIndex
        }

        if (array[currentIndex] > pivot2) {
            while (currentIndex < lastElementOfMiddlePartitionIndex &&
array[lastElementOfMiddlePartitionIndex] > pivot2) {
                decrement lastElementOfMiddlePartitionIndex
            }

            swap(array, currentIndex, lastElementOfMiddlePartitionIndex)
            decrement lastElementOfMiddlePartitionIndex

            if (array[currentIndex] < pivot1){
                swap(array, currentIndex, firstElementOfMiddlePartitionIndex)
                firstElementOfMiddlePartitionIndex
            }
        }
    }

    if (lastElementOfMiddlePartitionIndex - firstElementOfMiddlePartitionIndex < 13) {
        increment numDivisions
    }

    swap(array, firstElementOfMiddlePartitionIndex - 1, leftPivotIndex)
    swap(array, lastElementOfMiddlePartitionIndex + 1, rightPivotIndex)

    dualPivotQuicksort(array, leftPivotIndex, firstElementOfMiddlePartitionIndex - 2,
numDivisions)
    dualPivotQuicksort(array, lastElementOfMiddlePartitionIndex + 2, rightPivotIndex,
numDivisions)


    if (dist > len - 13 && pivot1 != pivot2) { //elements are equal
        for ( currentIndex = firstElementOfMiddlePartitionIndex; currentIndex <=
lastElementOfMiddlePartitionIndex; currentIndex++) {
            if (array[currentIndex] == pivot1) {
                swap(array, currentIndex, firstElementOfMiddlePartitionIndex++)
            }
```

```
            if (array[currentIndex] == pivot2) {
                swap(array, currentIndex, lastElementOfMiddlePartitionIndex--)
                if (array[currentIndex] == pivot1) {
                    swap(array, currentIndex, firstElementOfMiddlePartitionIndex++)
                }
            }
        }
    }

    if (pivot1 < pivot2) {
        dualPivotQuicksort(array, firstElementOfMiddlePartitionIndex,
lastElementOfMiddlePartitionIndex, numDivisions)
    }
}
```