STOP!

Before you proceed:

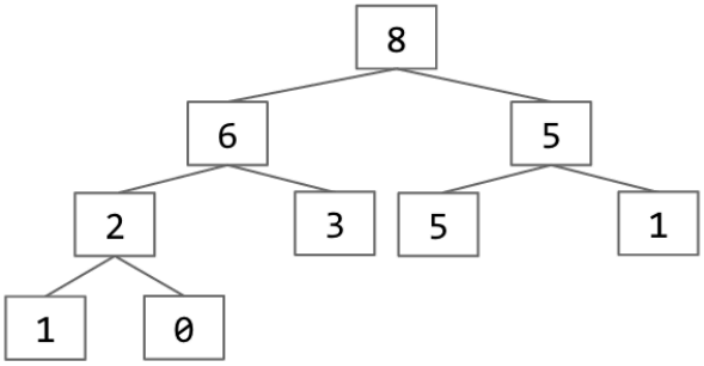
Don’t be tempted to check the solutions if you feel like you understand the solution to the problem.

Make sure you fully attempt a problem before checking its solution. This way, you can simulate an exam setting and see how you’d do if a particular problem were on the exam!

8. Balanced Trees (8 Points)

a) Suppose we have the max heap below, with array representation as shown. Show the heap after the maximum is deleted, using the procedure described in class. **Give your answer as an array.**

---	8	6	5	2	3	5	1	1	0			



Your answer:

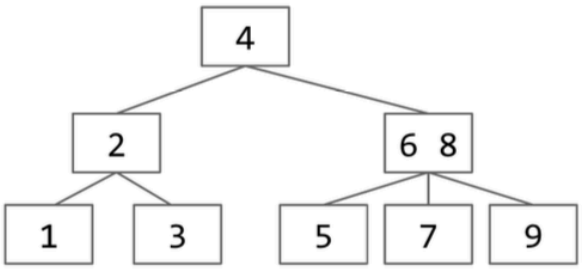
---	6	3	5	2	0	5	1	1				

We delete the node containing 0, then replace 8 with 0. We then swap, sinking the 0 node down: First we swap 0 and 6, then 0 and 3. We then write our answer as an array by doing a level-order traversal (top-to-bottom, left-to-right).

b) Consider the 2-3 tree below. What order should we insert these numbers so that we get the tree shown? There may be multiple correct answers.

Answer:

There are multiple answers, including the sequence [1, 2, 3, 4, 5, 6, 7, 8, 9].



One way to start is to recognize that the final numbers inserted could be 8 and 9. In this case, we must now figure out how to build a max-height 2-3 tree containing the numbers 1 through 7. This tree is relatively easy to obtain by guessing and checking. (Nearly half of all permutations of the numbers 1 through 7 would work.)

c) Tumelo Barttrain suggests creating a special version of a heap where each item has 4 children to improve performance. Would such a heap have better, worse, or the same asymptotic performance in Θ notation as compared to a normal binary heap? Briefly explain your reasoning.

Tumelo's heap would have the same asymptotic performance as a normal binary heap. The height of a heap where each item has 4 children is $\log_4(N)$ instead of $\log_2(N)$. Using the change of base formula for logs, $\log_4(N) = \log_2(N) / \log_2(4) = 0.5 * \log_2(N)$, so the height of the heap is still $\Theta(logN)$.

9. Heaps and JUnit (10 points).

Write a JUnit test to check a method `public void minheapify(int[] arr) {...}` that is supposed to perform bottom-up min-heap heapification. This means ensuring that the array is actually a heap, and also that the array still has all the same items. Our tests will verify only correctness, not runtime. Assume there will be no duplicates (which may make it easier to test that the array still contains the same inputs after heapification).

- **Hint: We're not leaving index 0 blank, so the left child of k is 2k + 1, and the right child is 2k + 2.**
- Hint 2: Feel free to use assertEquals(x, y), assertTrue(b), assertArrayEquals(x, y), etc.
- Hint 3: If you don't remember the exact syntax but your meaning is clear, penalties will be minimal.

```
@Test
public static void testHeapify() {
      int[] arr = generateRandomIntArray();
      int[] original = new int[arr.length]; // copy of array before being
heapified
      System.arraycopy(arr, 0, original, 0, arr.length);
      minheapify(arr);
      // Check the integrity of the result using one or two calls to helper
methods
      testIsAHeap(arr);
      testHaveSameItems(original, arr);
}
private static void testIsAHeap(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        if (2 * i + 1 < arr.length) {
            assertTrue(arr[i] < arr[2 * i + 1]);
        }
        if (2 * i + 2 < arr.length) {
            assertTrue(arr[i] < arr[2 * i + 2]);
        }
    }
} //You may not need all lines for these methods. Must include at least one
assert!
private static void testHaveSameItems(int[] original, int[] arr) {
    assertEquals(original.length, arr.length);
    for (int i = 0; i < arr.length; i++) {
        boolean exists = false;
        for (int j = 0; j < original.length; j++) {
            if (arr[i] == original[j]) {
                exists = true;
            }
        }
```

```
        assertTrue(exists);
    }
} // There is also a shorter solution that uses Arrays.sort
```

2. **Sorting (8 points).** a) **(6 pts)** Below, the leftmost column is an array of strings to be sorted. The column to the far right gives the strings in sorted order. Each of the remaining columns gives the contents of the array during some intermediate step of one of the algorithms listed below. Match each column with its corresponding algorithm. You will use each answer once. Write your answer in the blanks provided.

7777	1979	1979	7777	2001	1234	1234
2001	1234	2001	7777	8009	2001	1979
3015	1984	2015	4444	3015	3015	1981
2015	1981	2048	2015	2015	2015	1984
2048	2001	3015	7450	2016	2048	2001
8009	2015	4444	3015	2048	1981	2015
1979	2048	4500	2016	9150	1979	2016
7777	2016	7450	2001	1234	2016	2048
9150	3015	7777	1979	4444	1984	3015
4500	4500	7777	4500	7450	4500	4444
7450	4444	8009	2048	4500	7450	4500
4444	7777	9150	1981	7777	4444	7450
1234	7777	1234	1234	7777	7777	7777
1984	7450	1984	1984	1979	9150	7777
2016	8009	2016	8009	1981	7777	8009
1981	9150	1981	9150	1984	8009	9150
----	----	----	----	----	----	----
_1_	_6__	_2__	_4__	_5__	_3__	_7__

1: Unsorted, 2: Insertion, 3: Quick, 4: Heap, 5: LSD, 6: MSD, 7: Sorted

Column 2 is sorted by the thousandths place => MSD Sort
Column 3 is perfectly sorted except for the last four numbers, which are untouched => Insertion Sort
Column 4 is in heap form, with the last 2 max elements in its correct place => Heap
If you ignore the thousandths place of all the elements of column 5, it is in order => LSD
The second to last column immediately puts the first element in the correct place => strongly hints QuickSort

Notes:

- Quicksort is non-random and uses leftmost item as a pivot. The pivoting strategy is the Hoare two-pivot strategy, discussed in class.
- Insertion, Heapsort, LSD Radix Sort, and MSD Radix Sort as described in class.

b) **(1 pt)** One way to sort N items is to insert them randomly into a left leaning red black tree, and then traverse the LLRB. Which traversal should we use in order to print out the keys in sorted order? Circle one:

Preorder **Inorder** **Postorder** **Reverse-Preorder** **Reverse-Postorder**

c) **(1 pt)** $\Theta(N * \log(N))$ What is the worst case runtime of the sort described in part b? **Give your answer in Big Theta notation in terms of N**. Put your answer in the given blank.