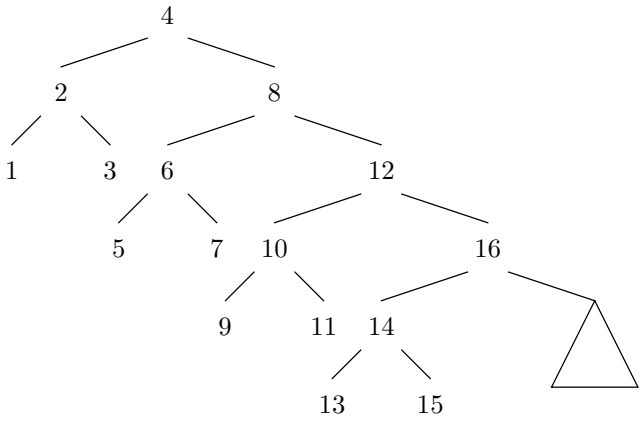
CS61B, Fall 2014 **Test #2 Solution** **P. N. Hilfinger and Josh Hug**

1. [1 point] We know that with a complete binary search tree with $n = 2^k - 1$ nodes, we can perform searches in $\Theta(\lg n)$ time. All nodes in such trees have either zero or two non-null children. In contrast, in a binary search tree where every node has either zero or one children, searches take $\Theta(n)$ time. Suppose that you have a general binary search tree in which every node has either zero or two children. What is the worst-case search time in that case? What do worst-case trees look like?

Answer: These, too, have worst-case times of $\Theta(n)$. Consider, for example,



Here, the depth of the tree is roughly $n/4$, which, of course, is $\Theta(n)$.

2. [1 point] Suppose we know that a certain type of object is going to be used as a key in a `HashMap` and looked up many times. Then one way to improve performance is to store the hash code of each such object in the object itself. Consider the following implementation of this idea for a wrapper class for the String class:

```
public class EfficientString {

    private int _hash;
    private String _val;

    public EfficientString(String val) {
        if (val == null) {
            throw new IllegalArgumentException();
        }
        _val = val;
        _hash = val.hashCode();
    }

    @Override
    public String toString() {
        return _val;
    }

    @Override
    public int hashCode() {
        return _hash;
    }

    public void setVal(String val) {
        _val = val;
    }

    @Override
    public boolean equals(Object obj) {
        return obj instanceof EfficientString && _val.equals(obj.toString());
    }
}
```

a. Unit testing shows that in some cases, the return value of `hashCode` is wrong. What is the cause of this error and how can it be fixed?

   Answer: The `setVal` method changes the value, but not the hash code.

b. Even after problem (a) is fixed (so that the hash value is always correct,) some users report problems using `EfficientStrings` in hash tables. What is the problem?

   Answer: It is still possible to change the value and hash code of an `EfficientString` *after* it has been added to the table, which can make it "disappear" from the table.

3. [3 points] In the table below, there are three pairs of sequences—a, b, and c. The first of each pair is the input to a sorting algorithm. The second is an intermediate sequence that might occur during the execution of the algorithm. Fill in each box of the table below with 'Y' or 'N' depending on whether the second sequence could or could not occur at any point during the execution of the indicated algorithms on the first sequence.

Note 1. For insertion sort, assume that items are swapped pairwise one-by-one towards the left of the array until they reach their final position, i.e. that we use the same swapping procedure that we used in lecture and in the optional HW8 solutions.

Note 2. For mergesort, assume that we merge from the "top down." For example, items 0 through 3 will be merged before items 4 and 5.

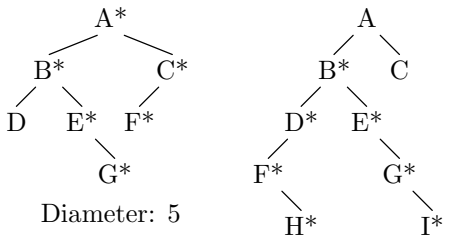
Note 3. For quicksort, we mean *any* valid partitioning method that divides the data into portions less than, equal to, and greater than a pivot, not necessarily the version from the homework (where elements stay in the same relative order after partitioning). Assume that the leftmost item is always used as the pivot.

Note 4. For heapsort, assume that the initial heapification process is performed by inserting all items in the array into a fresh (initially empty) heap. An intermediate sequence is considered valid even if it represents the state of an array in the middle of the re-heapification process after a delete (also called "sinking" or "heapify-down").

	Sequence	**Insertion sort**	**Merge sort**	**Quicksort sort**	**Heap sort**
a.	`12 17 13 7 16 23 22 24` `7 12 13 17 16 23 22 24`	Y	Y	Y	Y
b.	`23 22 16 11 15 3 4 10` `10 22 16 11 15 3 4 23`	N	N	Y	Y
c.	`23 23 21 14 1 1 21 12` `21 12 21 14 1 1 23 23`	N	N	Y	N

Put 'Y' or 'N' in each box: 'Y' for "could occur" or 'N' for "could not occur."

4. [2 points] The *diameter* of a tree (for this problem, a binary tree) is the maximum number of edges on any path connecting two nodes of the tree. For example, here are two sample trees and their diameters. In each case the nodes along a longest path are starred (there can be more than one longest path).



Diameter: 5

Diameter: 6

The diameter of an empty tree or a tree with one node is 0.

Fill in the method `diameter` to compute the diameter of a binary tree. Assume that `null` represents a missing child (empty tree), and that `.left` and `.right` give the left and right children of a node, and that the `height` method has been implemented correctly.

```
/** Return the height of T, where the height of an empty tree is -1 and
 *  that of a tree with one node is 0. */
public static int height(BinTree T) {
    if (T == null) {
        return -1;
    } else {
        return Math.max(height(T.left), height(T.right)) + 1;
    }
}

/** Return the diameter of T. */
public static int diameter(BinTree T) {

    if (T == null                                    ) {
         return 0;
    } else {


       int childDiam = Math.max(diameter(T.left), diameter(T.right));


       int nodeDiam = 2 + height(T.left) + height(T.right);


       return Math.max(childDiam, nodeDiam);

}
```

5. [2 points] The following classes represent nodes in a 2-4 tree (aka 2-3-4 tress). In order to avoid having to handle leaf nodes specially (all of which are empty in a 2-4 tree), this representation uses a single leaf object (`Node2_4.EMPTY`) to represent all leaves (instead of the usual null value). Fill in the blanks in `InnerNode2_4.contains` to look up keys in a tree.

```
/** Represents a node in a 2-4 (aka 2-3-4) tree. There is only one
 *  instance of this base class itself, Node2_4.EMPTY, representing
 *  the empty tree. */
class Node2_4 {

    /** The unique empty node. */
    final static Node2_4 EMPTY = new Node2_4();

    /** Return my Kth child (numbering from 0). */
    Node2_4 kid(int k) { /* Implementation not shown. */ }

    /** Return the number of my children (which is one more than the
     *  the number of my keys). */
    int arity() { /* Implementation not shown. */ }

    /** Return my Kth key (numbering from 0). */
    String key(int k) { /* Implementation not shown. */ }

    /** Return true iff KEY is a key in the tree rooted at me. */
    boolean contains(String key) {
        return false;
    }
}

/** Represents non-empty nodes. Exam note: Java short-circuits
    conditionals, e.g. 'if (true || b)' does not evaluate b. */
class InnerNode2_4 extends Node2_4 {
    @Override
    boolean contains(String key) {

        for (int k = 0; k < size()-1; k += 1          ) {

            if (key(k).equals(key)                          ) {
                return true;

            } else if (key(k).compareTo(key) > 0                ) {


                return kid(k             ).contains(key);
            }
        }


        return kid(size() - 1).contains(key);
    }
}
```

6. [2 points] Suppose you are given an array `A` with n elements such that every element is less than k slots away from its position in the sorted array. Assume that $k > 0$ and that k, while not constant, is much less than n ($k \ll n$).

a. Fill in the blanks such that the array `A` is sorted after execution of this method. The important operations on a `PriorityQueue` are `add(x)`, `remove()` (remove smallest), and `isEmpty()`. Your solution should be as fast as possible.

```
public static void zorkSort(int[] A, int k) {
    int i;
    int n = A.length;
    i = 0;
    PriorityQueue<Integer> pq = new PriorityQueue<>();
    while (i < k) {

        pq.add(A[i]);
        i += 1;
    }

    while (i < n                        ) {

        A[i - k] = pq.remove();

        pq.add(A[i]);
        i += 1
    }

    while (!pq.isEmpty()                ) {

        A[i - k] = pq.remove();
        i += 1;
    }
}
```

b. What is the running time of this algorithm, as a function of n and k? Justify your answer.

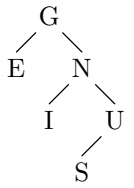
Answer: We add a total of n elements (where n is the length of `A`) to the priority queue and remove the same number. The queue never has more than k elements in it, giving a running time of $n \lg k$.

7. [1 point] During an oral dissertation defense, the great mathematician David Hilbert supposedly asked the student a single question: "In all my years, I have never seen such beautiful evening clothes; pray, who is the candidate's tailor?" Who was this PhD candidate?

Answer: John von Neumann

8. [1 point] Consider a binary search tree whose labels are each one capital letter, and assume that a level-order traversal yields the sequence "GENIUS". What is the preorder traversal of this tree? Draw the tree itself.

Answer: The preorder traversal also yields "GENIUS." The tree is



9. [2 points] The following questions are about tries.

a. How can you use a trie to do a range query (e.g., "What are all strings between "alpha" and "beta" in the dictionary?").

Answer: Keep the children of each node in order by edge label. Traverse the trie, skipping those edges that would take you to strings that are less than or greater than the bounds.

b. Henry Hacker has decided, for some reason, to replace a 2-4 tree he is using to implement a type of `SortedSet` of unbounded positive integers by a trie. (In the Java library, a `SortedSet` is an extension of `Set` whose iterator delivers all items in order—numeric order in this case.) He wants to be sure that his new implementation, like his 2-4 trees, can iterate through its values or a bounded subset of them in linear time. He says, "It's easy; I'll just treat my numbers as strings of digits." Why doesn't this work?

Answer: Unfortunately, numerals don't sort like strings. For example, $91 < 100$, but the string `100` comes before `91`.

c. If searches through a certain trie for two different strings, X and Y, traverse the same particular node in the trie that is at depth k in the trie (where the root is depth 0), what can you say about X and Y?

Answer: Their first k characters (at least) are the same.

d. Bernice Bitwiddle has successfully defined a new implementation of the Java `Set<String>` interface (the type `HashSet<String>` is an existing type that implements the same interface). Her new class uses a trie in which she stores the strings *backwards;* that is, with the top of trie corresponding to the *last,* rather than the first letter of the word. Could she easily implement an `iterator()` method for this representation that returns (in constant time) an iterator that then allows one to iterate through all the elements in sorted order in $\Theta(N)$ time? If not, why not, and if so, how?

Answer: The characters at the end of a string tell you almost nothing about its ordering, so no ordinary traveral of the trie will find keys in sorted order.

10. [2 points] Suppose we have a complete binary tree, X, that is not a heap, and we wish to heapify it. The obvious thing to do is just insert all nodes of X into a new binary heap Y. This works, but it doubles the space requirement. Suppose we'd prefer to heapify in place. Which of the following procedures will convert X into a heap (containing all of the original values)? To each, either answer "yes" or give a counter-example.

a. Sink (heapify down) all nodes in level order (first the root, then its left child, then the right child of the root, etc).

   Answer: No. [Here and below, we assume a max heap, which we present as an array.] Consider 1, 2, 3, 12, 13, 4, 5. This will bring 3 to the top, instead of 13.

b. Swim (heapify up) all nodes in level order.

   Answer: Yes. This is just what you do to add elements to the heap.

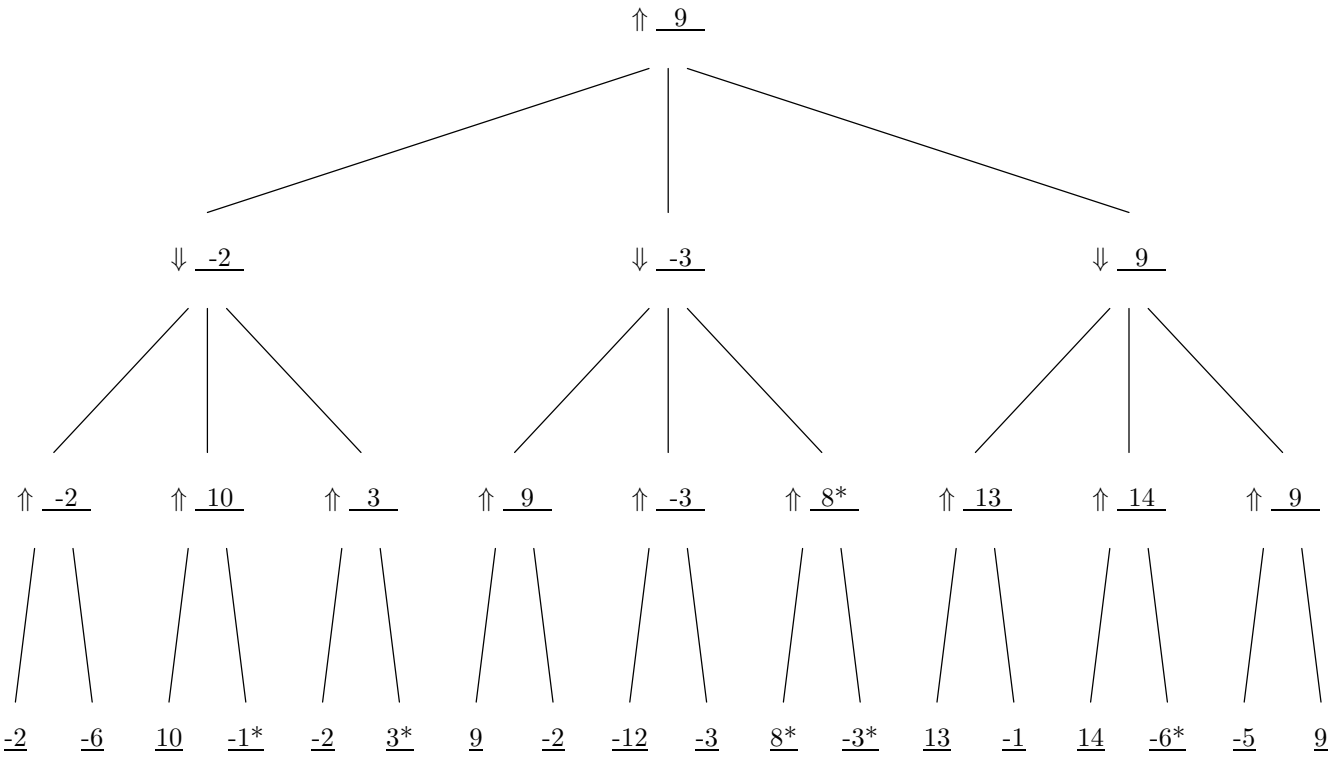
c. Swim all nodes in reverse level order.

   Answer: No. Consider 1, 6, 7, 2, 3, 4, 5. This will leave 1 above 4 and 5.

d. Sink all nodes in reverse level order.

   Answer: Yes. At each step, the current node will be the root of a subtree all of whose nodes, with the possible exception of the current node, will obey the heap property (inductive assumption). In this case, heapifying down will necessarily work, just as when you swap a (small) node for the root in the remove-first operation.

11. [1 point] Consider the following game tree (the top node represents a position in which it is the "max" player's turn).



a. Fill in the blanks in the first three rows of nodes with the values computed by the minimax algorithm.

b. Show which of these nodes can be skipped by alpha-beta pruning by crossing them out.

Answer: Nodes that never need to be visited are indicated with an asterisk

c. True/False: Assume that the min player plays sub-optimally at every turn, but MAX does not know this. Then the outcome of the game for the max player could be larger than predicted via minimax. Justify your answer.

Answer: True. Consider the min player on the far right subtree above. Suppose that after max (in the top row) moves to the last position in the second row, the min player suboptimally chooses the move with value 14 instead of that with value 9. Then max would end up in the position on the bottom row with value 14, even though he expected only to achieve a value of 9.