CS61B, Fall 2015 **Test #2 Solution** **P. N. Hilfinger**

1. [2 point] For each of the methods below, give the best case and worst case runtime in $\Theta(\cdot)$ notation, as a function of n. Your answers should be as simple as possible, excluding unnecessary constants and lower-order terms. Assume there is no limit on the size of an **int** (otherwise all run times are technically constant). The answer may be infinity.

a.
```
public static void first(int n) {
    for (int outer = 1; outer < n; outer *= 2) {
        int inner;
        inner = 0;
        while (inner < n) {
            inner++;
        }
    }
}
```

Best case: $\Theta(n \lg n)$; Worst case: $\Theta(n \lg n)$

b.
```
public static boolean second(int[] arr) {
    int n = arr.length;
    for (int index1 = 0; index1 < n; index1 += 1) {
        int index2 = index1 + 1;
        while (index2 < n) {
           if (arr[index1] + arr[index2] == n) {
                return true;
           }
           index2 += 1;
        }
    }
    return false;
}
```

Best case: $\Theta(1)$; Worst case: $\Theta(n^2)$

Problem continues on the next page.

c.
```
public static int third(int n) {
    return third(n, n);
}

public static int third(int x, int n) {
    if (x == 1) {
        return x;
    } else {
       int sum = 0;
       for (int i = 0; i < n; i++) {
            sum += third(x - 1, n);
       }
       return sum;
    }
}
```

Best case: $\Theta(n^{n-1})$; Worst case: $\Theta(n^{n-1})$

d. Ignore the cost of resizing the heap.

```
public static void fourth(PriorityQueue<Integer> heap, int[] in) {
    int n = in.length;
    for(int x : in) {
        heap.add(x);
    }
}
```

Best case: $\Theta(n)$; Worst case: $\Theta(n \lg n)$

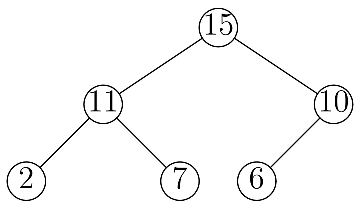
---

2. [1 point] Suppose that `A` is an array of n **int**s sorted into non-descending order, and suppose that `B` is formed from `A` by adding random integers between -3 and 3 to each element (so that `A = [1, 4, 5, 7]` might become `[2, 1, 8, 4]`). Give best and worst-case times for insertion sort to sort `B` back into non-descending order. **Justify** your answer.

Best case: $\Theta(n)$; Worst case: $\Theta(n^2)$

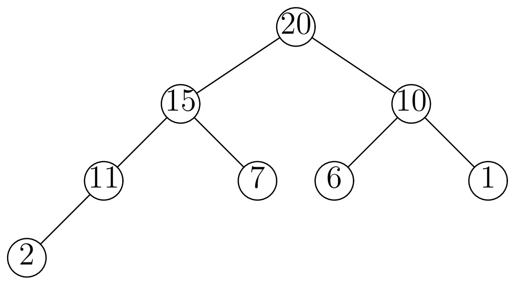
If `A` is in ascending order, all elements differ by at least 1, so any elements that are more than 6 places apart cannot change their positions relative to each other in `B`. It follows that the total number of inversions in `B`, I_B, must be proportional to n. Since insertion sort takes time $n + I_B \in O(n)$. Since insertion sort takes at least n time, the time is also $\Omega(n)$, and hence $\Theta(n)$. If, however, the elements are identical, then this perturbation can result in $\Theta(n^2)$ inversions in the worst case, requiring $\Theta(n^2)$ time.

3. [2 points] Given the following (max-) heap,



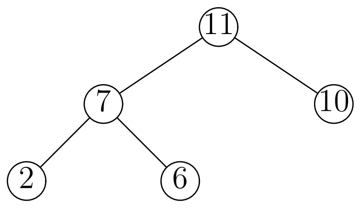
a. Show what the heap looks like after inserting 1 and then 20 (just show the final heap; you don't need to show it just after inserting 1).

Solution:



b. Starting from the original heap (shown at the top), show what the heap looks like after removing 15.

Solution:



4. [1 point] Between 1 January and 1 June, a certain star seems to shift by 1/36000 degrees against the background of very distant stars. About how far away is it?

1/36000 degree is 0.1 arcsec. Over the period of 6 months, the earth moves 1 AU in one dimension. That means the star is at most about 10 parsec (about 33 light years) from earth, if its direction is perpendicular to the direction of earth's motion on 1 January—and closer otherwise.

5. [2 points] For parts (a) and (b), suppose we have a HashMap implementation that has an initial capacity of 1, and a load factor of 1. In this hashmap implementation, each resizing of the hashmap increases the number of buckets by 1. Assume we are using a good hash function—one that (miraculously) always divides our data sets evenly between the buckets. Give a $\Theta(\cdot)$ worst-case bound for the running times of the following operations. **Justify** your answers.

a. Inserting n key-value pairs (assume distinct keys).

$\Theta(n^2)$. Each insertion requires resizing by 1, giving a total time proportional to $1 + 2 + 3 + \ldots + n \in \Theta(n^2)$.

b. Looking up a key after the n insertions from part (a).

$\Theta(1)$: since we assume the hash function is miraculously good and the load factor is 1, each bucket will contain one item.

For parts (c) and (d) now suppose we have the same HashMap implementation, except resizing doubles the capacity, and we use a hash function that maps all keys to 0. Give a $\Theta(\cdot)$ bound for the average runtime of the following operations. **Justify** your answers.

c. Inserting n key-value pairs (assume distinct keys).

$\Theta(n^2)$. Bucket 0 will contain all keys. Each insertion must check that the item is is not in the bucket first, meaning that total insertion time for n items will be proportional to $1 + 2 + \ldots + n \in \Theta(n^2)$.

d. Looking up a key after the n insertions from part (c).

Again, one is searching a single unordered linked list of size n. Hence, the time will be $\Theta(n)$.

6. [2 points] Suppose the instance methods `hashCode1()` and `hashCode2()` in class `A` are both "good" hash functions. For each of the following computations, answer whether or not it is also guaranteed to yield a good hash value and explain why or give an example of a situation when it wouldn't be "good."

a. `hashCode1() ^ hashCode2()`.

   No. If, for example, `hashCode()` and `hashCode2()` were the same, the result would always be 0. Or if they differed in just one bit, both would still be good, but the combination would have only a few possible values.

b. `hashCode1()` if `hashCode2()` is 0 and `hashCode2()` otherwise.

   Yes. This is simply `hashCode2()` most of the time, which is assumed to be good. When it isn't, it is `hashCode1()`, also assumed to be good.

c. Generate a random number. If that number is odd, yield `hashCode1()` and otherwise `hashCode2()`

   No. It's fine as far as producing a good distribution of values, but the values are nondeterministic, and so cannot be used as a hash function.

d. `-hashCode1()` if `hashCode1()` is even, and otherwise `hashCode1()`. Yes. Values are still distributed as evenly as for `hashCode1()`, just differently.

7. [3 points] For each of the parts below, the first lines gives the (same) initial array, and the remaining lines show the state of the array at major steps during a sort (not necessarily evenly spaced steps). For each item, indicate which algorithm can generate it: heap sort, quicksort, LSD radix sort, MSD radix sort, or insertion sort. Assume that quicksort always chooses the first element in a subsequence as the pivot; the particular partitioning algorithm might not be the stable one used in homework 8.

a.
```
658 613 095 997 783 226 754 442 333 572 628 421 291 093 598 257
257 613 095 226 442 333 572 628 421 291 093 598 658 783 997 754
093 095 226 257 442 333 572 628 421 291 613 598 658 783 997 754
093 095 226 257 291 333 421 442 572 628 613 598 658 783 997 754
```

Quicksort. The three steps shown give the results of partitioning on 658, 257, and 442.

b.
```
658 613 095 997 783 226 754 442 333 572 628 421 291 093 598 257
997 783 754 613 658 421 598 442 333 572 628 226 291 093 095 257
783 658 754 613 628 421 598 442 333 572 257 226 291 093 095 997
658 628 598 613 572 421 095 442 333 093 257 226 291 754 783 997
```

Heapsort. The second line is the result of heapification. The third shows the next step: moving 997 off the end of the heap and re-heapifying. The fourth shows the result after two more reheapifications.

c.
```
658 613 095 997 783 226 754 442 333 572 628 421 291 093 598 257
421 291 442 572 613 783 333 093 754 095 226 997 257 658 628 598
613 421 226 628 333 442 754 257 658 572 783 291 093 095 997 598
```

LSD radix sort, showing the results of sorting on the last and then middle digits.

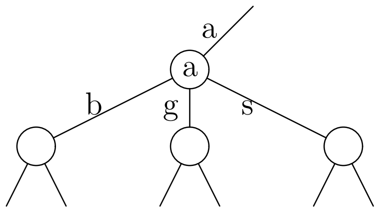
d.
```
658 613 095 997 783 226 754 442 333 572 628 421 291 093 598 257
613 658 095 997 783 226 754 442 333 572 628 421 291 093 598 257
095 226 613 658 783 997 754 442 333 572 628 421 291 093 598 257
```

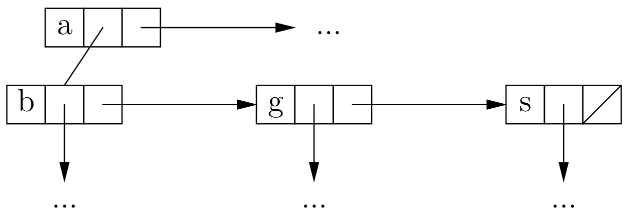
Insertion sort. The second line shows the result of the first iteration of the outer loop. The third shows the result of four more iterations of the outer loop.

e.
```
658 613 095 997 783 226 754 442 333 572 628 421 291 093 598 257
095 093 226 291 257 333 442 421 572 598 658 613 628 783 754 997
095 093 226 257 291 333 421 442 572 598 613 628 658 754 783 997
```

MSD radix sort, showing the results of sorting on the first and then second digits.

8. [3 points] A certain `TrieSet` class represents a node such as that shown on the left by the linked structure shown on the right.





That is, internal nodes contain the character on the edge leading from their parent (ignored for the root node), plus a pointer to their first child, plus a pointer to the next sibling node to their right. Leaf nodes also contain the entire string that they represent (their child pointers are null). Sibling nodes are linked in order of the characters on the edges leading from their parents. For simplicity, assume that the end-of-string "character", which we've represented as □ in lecture and in DSIJ, is represented by the ASCII NUL character (in Java, `'\0'`), is always the last character of a key, and only appears at the end. (Usually in Java, unlike C or C++, there is no explicit end-of-string character actually present, but for this problem, we'll pretend that there is).

Below and on the next page is a small part of the Java rendition of this class. You are to fill in the two private `.contains()` methods. Each of them takes two parameters: the key being searched for and the node's level in the trie (0 for the root).

```
public class TrieSet implements Set<String> {
    TrieSet() {
        _root = null;
    }


    ...
    @Override
    public boolean contains(String key) {
        if (_root == null) {
            return false;
        } else {
            return _root.contains(key, 0);
        }
    }

    private Node _root;
```

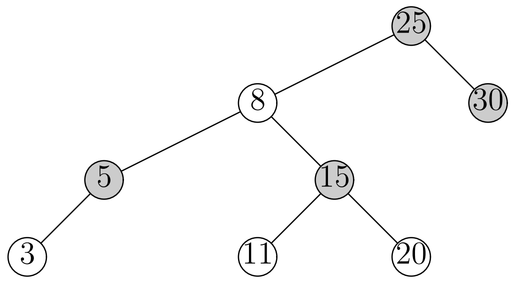
Continues on next page

Continuation of TrieSet *class*

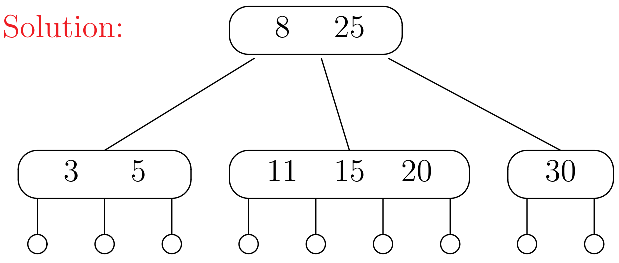
```
    private static class Node {
        ...
        /** Return true iff the subtree rooted at me contains KEY,
         *  assuming that I appear at level K of this trie, and that
         *  the length of KEY is at least K+1. */
        boolean contains(String key, int k) { // FILL IN


            char c = key.charAt(k);
            for (Node n = _firstChild; n != null; n = n._nextSibling) {
                if (c == n._c) {
                    return n.contains(key, k + 1);
                }
            }
            return false;


         }

        private char _c;
        private Node _firstChild;
        private Node _nextSibling;
    }
    private class LeafNode extends Node {
        ...
        @Override
        boolean contains(String key, int k) { // FILL IN

            return key.equals(_value);
        }
        /** The (single) String represented by this leaf node. */
        private String _value;
    }
}
```

9. [2 points]

a. Show the 2-4 tree corresponding to the following red-black tree (shaded nodes are black):

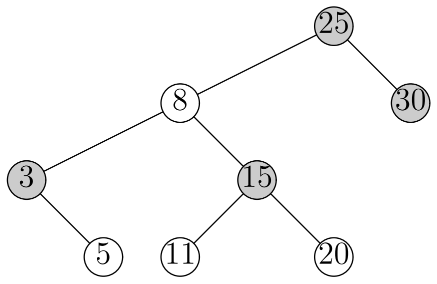


Solution:



b. Show an alternative red-black tree that corresponds to the same 2-4 tree as in part (a).

Solution:



c. If a certain 2-4 tree has a height of H (that is, has $H+1$ levels) what are the maximum and minimum heights of the corresponding red-black trees? Do not count the empty (null) nodes at the bottom of any of these trees in computing heights (so a 2-4 tree with just a root node has one level and height 0). Your answers should ***not*** use $\Theta(\cdot)$ notation.

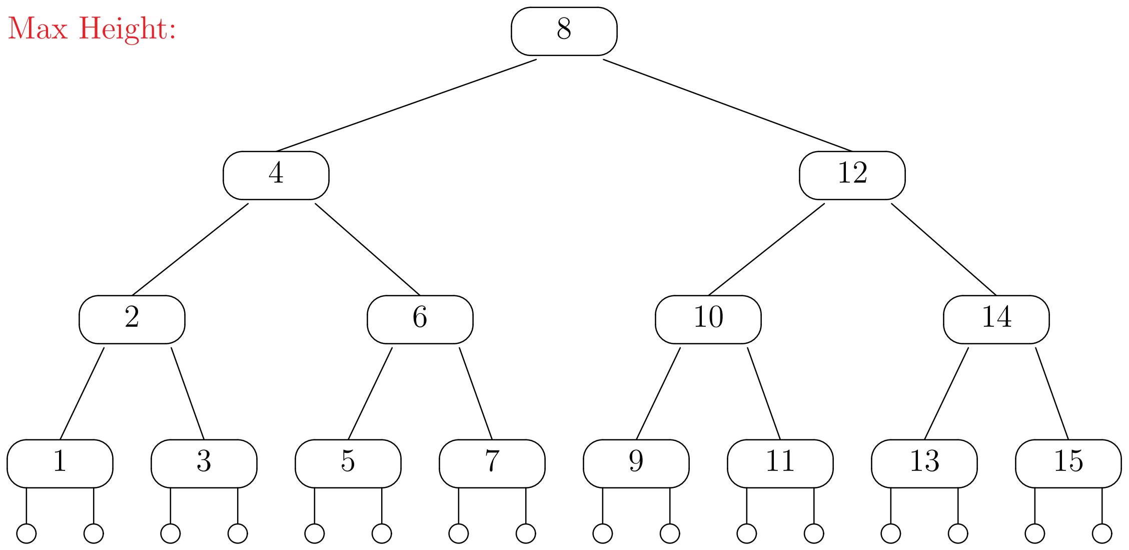
Maximum: $2H + 1$; Minimum: H

Continued

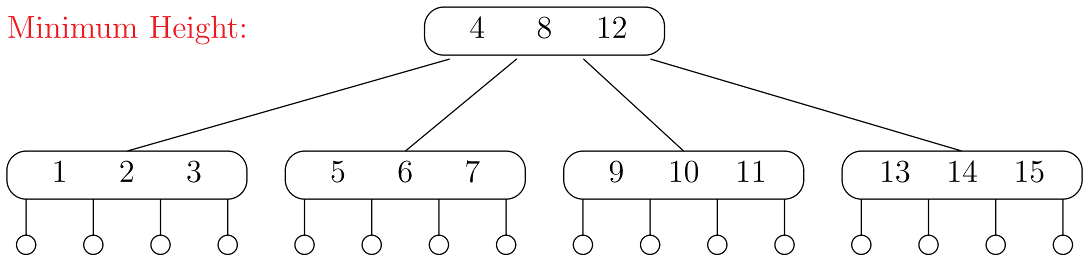
d. Show two 2-4 trees containing values 1–15 and having maximum and minimum depths.

Solution:

Max Height:



Minimum Height:


4 8 12
1 2 3
5 6 7
9 10 11
13 14 15