Problem Set 4

Due before the start of lecture on November 28, 2018.

See below for a summary of the policies regarding late submissions.

Preliminaries

Homework is due prior to the start of lecture. If it is submitted more than 10 minutes after the start of lecture, it will be considered a full day late. There will be a 10% deduction for late submissions that are made by 11:59 p.m. on the Sunday after the deadline, and a 20% deduction for submissions that are made after that Sunday and before the start of the next lecture (but see above for the revised policy for this assignment). ***We will not accept any homework that is more than 7 days late.*** Plan your time carefully, and don't wait until the last minute to begin an assignment. Starting early will give you ample time to ask questions and obtain assistance.

In your work on this assignment, make sure to abide by the policies on academic conduct described in the syllabus.

If you have questions while working on this assignment, please attend office hours, post them on Piazza, or email `cscie22@fas.harvard.edu`.

Part I

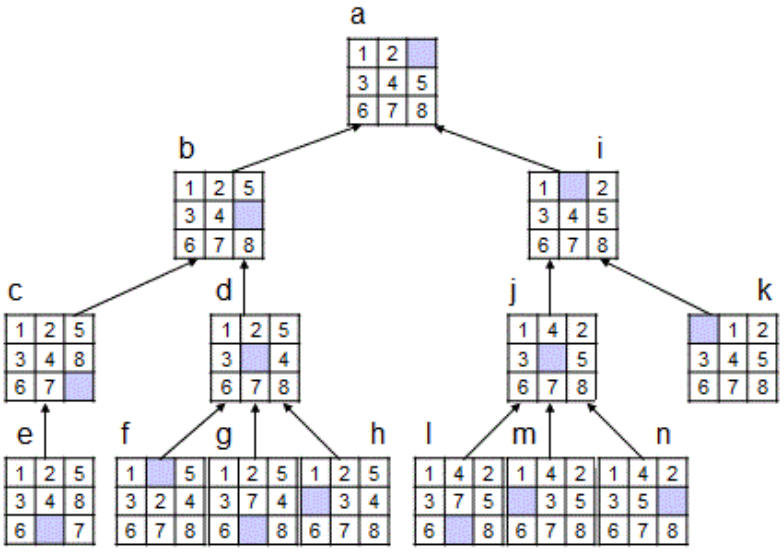
50 points total

Because we are asking you to draw diagrams, you may submit *either* a plain-text file or a PDF file. Give it the name `ps4_partI.txt` or `ps4_partI.pdf`.

Problem 1: Uninformed state-space search

6 points total; 2 points each part

Below is a portion of the state-space search tree for an Eight Puzzle whose the initial configuration is shown at the root of the tree. This tree reflects an assumption that the search algorithms will discard successor states that are already present on the path from the current state to the root of the tree.



1. What are the first six states to which breadth-first search (BFS) would apply the goal test? Please follow the guidelines given below. *Hint:* The first state to be tested is the initial state.

2. What are the first six states to which depth-first search (DFS) would apply the goal test if it uses a depth limit of 3? See the hint from part 1, and follow the guidelines given below.

3. What are the first **eight** states to which iterative-deepening search (IDS) would apply the goal test? Don't forget that IDS starts over at the initial state for each new value of the depth limit. Therefore, it is possible for a given state to appear more than once among the first eight states to be tested. Don't forget to follow the guidelines below.

Guidelines:

- In the diagram, each state is labeled with a lower-case letter that appears above the state. You should use these labels to identify the states in your answers.
- ***When an algorithm needs to choose among multiple states at the same depth, you should assume that it will select the one whose lower-case letter comes first alphabetically.*** However, you should also take into account the fact that the tree is generated gradually over the course of a given algorithm, and you shouldn't attempt to test a state that wouldn't have been generated yet at that point in the algorithm.
- Your answers should list the requested number of states in the order in which they would be tested by the algorithm.

Problem 2: Counting keys below a threshold

12 points total; 4 points each part

The code below represents one algorithm for counting the number of keys that are less than some threshold value `t` in an instance of our `LinkedTree` class. The `numSmallerInTree()` method returns the number of keys less than `t` in the tree/subtree whose root node is specified by the first parameter of the method, and the `numSmaller()` method returns the number of keys less than `t` in the entire tree represented by the `LinkedTree` object on which the method is invoked.

```
public int numSmaller(int t) {
    if (root == null) {    // root is the root of the entire tree
        return 0;
    }

    return numSmallerInTree(root, t);
}

private static int numSmallerInTree(Node root, int t) {
    int count = 0;

    if (root.left != null) {
        count += numSmallerInTree(root.left, t);
    }
    if (root.right != null) {
        count += numSmallerInTree(root.right, t);
    }

    if (root.key < t) {
        return 1 + count;
    } else {
        return count;
    }
}
```

1. For a binary tree with *n* nodes, what is the time complexity of this algorithm in the best case? In the worst case? For the worst case, give two expressions: one for when the tree is balanced, and one for when the tree is not balanced. Give your answers using big-O notation, and explain them briefly.

2. If the tree is a binary *search* tree, we can revise the algorithm to take advantage of the ways in which the keys are arranged in the tree. Write a revised version of `numSmallerInTree` that does so. Your new method should avoid visiting nodes unnecessarily. In the same way that the search for a key doesn't consider every node in the tree, your method should avoid considering subtrees that couldn't contain values less than the threshold. Like the original version of the method above, your revised method should also be recursive.

   Note: In the files that we've given you for Part II, the `LinkedTree` class includes the methods shown above. Feel free to replace the original `numSmallerInTree()` method with your new version so that you can test its correctness. However, your new version of the method should ultimately be included in the file with the rest of your work for Part I.

3. For a binary search tree with *n* nodes, what is the time complexity of your revised algorithm in the best case? In the worst case? For the worst case, give two expressions: one for when the tree is balanced, and one for when the tree is not balanced. Give your answers using big-O notation, and explain them briefly.

Problem 3: Tree traversal puzzles

8 points total; 4 points each part

1. When a binary tree of characters (which is *not* a binary *search* tree) is listed in postorder, the result is ENIMOPTFRW. Inorder traversal gives IENWOMRPFT. Construct the tree.

2. When a binary tree of characters (which is *not* a binary *search* tree) is listed in preorder, the result is TRMESBONVY. Postorder gives ESMRVYNOBT. Construct the tree. (There is more than one possible answer in this case.)

Problem 4: Huffman encoding

6 points total

Consider the following table of character frequencies:

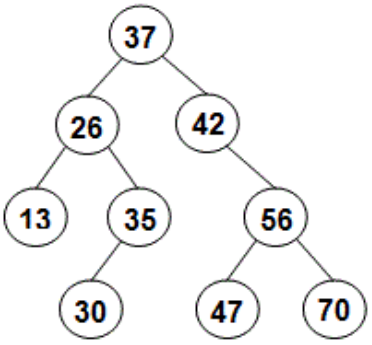
Character	Frequency
o	6
f	9
l	13
c	22
a	30

1. (4 points) Show the Huffman tree that would be constructed from these character frequencies.
2. (2 points) Using the Huffman tree from part 1, what will be the encoding of the string *focal*?

Problem 5: Binary search trees

10 points; 2 points each part

Consider the following binary search tree, in which the nodes have the specified integers as keys:



1. If a *preorder* traversal were used to print the keys, what would the output be?
2. What would be the output of a *postorder* traversal?
3. Show the tree as it will appear if 50 is inserted, followed by 10.
4. Suppose we have the original tree and that 37 is deleted and then 26 is deleted, using the algorithm from the lecture notes. Show the final tree.
5. Is the original tree balanced? Explain briefly why or why not.

Problem 6: 2-3 Trees and B-trees

8 points; 4 points each part

Illustrate the process of inserting the key sequence

```
J, E, I, H, C, F, B, G, D, A
```

into:

1. an initially empty 2-3 tree
2. an initially empty B-tree of order 2

In both cases, show the tree before and after each split, and the final tree.

Part II

50-60 points total

Preparing for Part II

Begin by downloading the following zip file: ps4.zip

Unzip this archive, and you should find a folder named `ps4` that contains all of the files that you will need for this assignment. **You should *not* move any of the files out of the `ps4` folder.** Keeping everything together in that folder will ensure that you are able to compile and run the work that you complete for the problems below.

Problem 7: Adding methods to the `LinkedTree` class

25 points

*Make sure to begin by following the instructions given above in the **Preparing for Part II** section.*

In the file `LinkedTree.java`, add code that completes the following tasks:

1. Write a non-static `sumKeysTo(int key)` method that takes takes an integer key as its only parameter and that uses ***uses iteration*** to determine and return the sum of the keys on the path from the root node to the node with the specified key, including the key itself. ***Your method should take advantage of the fact that the tree is a binary search tree***, and it should avoid considering subtrees that couldn't contain the specified key. It should return 0 if the specified key is not found in the tree.

   Note: There are two methods in the `LinkedTree` class that can facilitate your testing of this method and the other methods that you'll write:

   - The `insertKeys()` method takes an array of integer keys, and it processes the array from left to right, adding a node for each key to the tree using the `insert()` method. (The data associated with each key is a string based on the key, although our tests will focus on just the keys.)
   - The `levelOrderPrint()` method performs a level-order traversal of the tree and prints the nodes as they are visited; each level is printed on a separate line. This method doesn't show you the precise shape of the tree or the edges between nodes, but it gives you some sense of where the nodes are found.

   For example, below are some tests of `sumKeysTo()` from the Interactions Pane in DrJava. To help you visualize the tree, it's worth noting that we're using an array of keys that produces the binary search tree shown in Problem 5.

```
> LinkedTree tree = new LinkedTree();
> tree.sumKeysTo(13)
0
```

```
> int[] keys = {37, 26, 42, 13, 35, 56, 30, 47, 70};
> tree.insertKeys(keys);
> tree.sumKeysTo(13)
76
> tree.sumKeysTo(56)
135
> tree.sumKeysTo(37)
37
> tree.sumKeysTo(50)
0
```

2. Write two methods that together allow a client to determine the number of leaf nodes in the tree:

   - a private static method called `numLeafNodesInTree()` that takes a reference to a `Node` object as its only parameter; it should ***use recursion*** to find and return the number of leaf nodes in the binary search tree or subtree whose root node is specified by the parameter. Make sure that your method correctly handles empty trees/subtrees – i.e., cases in which the value of the parameter `root` is `null`.

   - a public non-static method called `numLeafNodes()` that takes no parameters and that returns the number of leaf nodes in the entire tree. This method should serve as a "wrapper" method for `numLeafNodesInTree()`. It should make the initial call to that method – passing in the root of the tree as a whole – and it should return whatever value that method returns.

   For example:

```
> LinkedTree tree = new LinkedTree();
> tree.numLeafNodes()
0
> int[] keys = {37, 26, 42, 13, 35, 56, 30, 47, 70};
> tree.insertKeys(keys);
> tree.numLeafNodes()
4
```

3. Write a non-static method `deleteSmallest()` that takes no parameters and that ***uses iteration*** to find and delete the node containing the smallest key in the tree; it should also return the value of the key whose node was deleted. If the tree is empty when the method is called, the method should return -1.

   Important: Your `deleteSmallest()` method may ***not*** call any of the other `LinkedTree` methods (including the `delete()` method), and it may not use any helper methods. Rather, this method must take all of the necessary steps on its own – including correctly handling any child that the smallest node may have.

   For example:

```
> LinkedTree tree = new LinkedTree();
> tree.deleteSmallest()
-1
> int[] keys = {37, 26, 42, 13, 35, 56, 30, 47, 70};
> tree.insertKeys(keys);
> tree.levelOrderPrint();
37
26 42
13 35 56
30 47 70
> tree.deleteSmallest()
```

```
13
> tree.levelOrderPrint();
37
26 42
35 56
30 47 70
> tree.deleteSmallest()
26
> tree.levelOrderPrint();
37
35 42
30 56
47 70
```

Note that first we delete 13, because it is the smallest key in the original tree. Next we delete 26, because it is the smallest remaining key. As a result of these deletions, 35 and 30 move up a level in the tree.

4. Writing well-formatted units tests is an extremely important part of a programmer's work. In the `main()` method of `LinkedTree.java`, we've given you an example of what such a unit test should look like.

   Update the `main()` method to include ***at least two unit tests for each of your new methods***. Your unit tests must follow the same format as our example test. In particular, the output of each of your unit tests should include:

   - a header that specifies the test number and a description of what is being tested
   - the actual return value that you get from that test
   - the expected return value
   - whether the actual return value matches the expected return value.

   Put each test in the context of a `try-catch` block so that you can handle any exceptions that are thrown. Leave a blank line between tests.

   Additional notes:

   - For part 2 (`numLeafNodesInTree()`/`numLeafNodes()`), your unit tests only need to call `numLeafNodes()`, since doing so will also call `numLeafNodesInTree()`.
   - Our model unit test can be used to test the `numSmaller()`/`numSmallerInTree()` methods from Problem 2.

Problem 8: Binary tree iterator

25 points

The traversal methods that are part of the `LinkedTree` class are limited in two significant ways: (1) they always traverse the entire tree; and (2) the only functionality that they support is printing the keys in the nodes. Ideally, we would like to allow the users of the class to traverse only a portion of the tree, and to perform different types of functionality during the traversal. For example, users might want to compute the sum of all of the keys in the tree. In this problem, you will add support for more flexible tree traversals by implementing an iterator for our `LinkedTree` class.

You should use an inner class to implement the iterator, and it should implement the following interface:

```
public interface LinkedTreeIterator {
    // Are there other nodes to see in this traversal?
    boolean hasNext();
```

```
    // Return the value of the key in the next node in the
    // traversal, and advance the position of the iterator.
    int next();
}
```

There are a number of types of binary-tree iterators that we could implement. We have given you the implementation of a *preorder* iterator (the inner class `PreorderIterator`), and you will implement a ***postorder*** iterator for this problem.

Your postorder iterator class should implement the `hasNext()` and `next()` methods so that, given a reference named tree to an arbitrary `LinkedTree` object, the following code will perform a complete postorder traversal of the corresponding tree:

```
LinkedTreeIterator iter = tree.postorderIterator();
while (iter.hasNext()) {
    int key = iter.next();

    // do something with key
}
```

Important guidelines

- In theory, one approach to implementing a tree iterator would be to perform a full recursive traversal of the tree when the iterator is first created and to insert the visited nodes in an auxiliary data structure (e.g., a list). The iterator would then iterate over that data structure to perform the traversal. **You should *not* use this approach.** One problem with using an auxiliary data structure is that it gives your iterator a space complexity of $O(n)$, where n is the number of nodes in the tree. Your iterator class should have a space complexity of $O(1)$.
- Your iterator's `hasNext()` method should have a time efficiency of $O(1)$.
- Your iterator's constructor and `next()` methods should be as efficient as possible, given the time efficiency requirement for `hasNext()` and the requirement that you use no more than $O(1)$ space.
- We encourage you to consult our implementation of the `PreorderIterator` class when designing your class. It can also help to draw diagrams of example trees and use them to figure out what you need to do to go from one node to the next.

Here are the tasks that you should perform:

0. Review the code that we've given you in the `PreorderIterator` class and the `preorderIterator()` method, and understand how that iterator works. It's worth noting that you can't actually use a `PreorderIterator` object yet, because it will only work after you have completed the next task.

1. In order for an iterator to work, it's necessary for each node to maintain a reference to its parent in the tree. These parent references will allow the iterator to work its way back up the tree.

   In the copy of `LinkedTree.java` that we've given you for this assignment, the inner `Node` class includes a field called `parent`, but the `LinkedTree` code does ***not*** actually maintain this field as the tree is updated over time. Rather, this `parent` field is assigned a value of `null` by the `Node` constructor, and its value remains `null` forever.

   Before implementing the iterator, you should make whatever changes are needed to the existing `LinkedTree` methods so that they correctly maintain the `parent` fields in the `Node` objects.

   - For example, when a `Node` object is first inserted in the tree, you should set its `parent` field to point to the appropriate parent node.

- Think about when the parent of a node can change, and update the necessary method(s) to modify the `parent` field in a `Node` object whenever its parent changes.
- The root of the entire tree should have a `parent` value of `null`.

2. Next, add a skeleton for your iterator class, which you should name `PostorderIterator` (note that only the `P` and `I` are capitalized). It should be a private inner class of the `LinkedTree` class, and it should implement the `LinkedTreeIterator` interface. Include whatever private fields will be needed to keep track of the location of the iterator. Use our `PreorderIterator` class as a model.

3. Implement the constructor for your iterator class. Make sure that it performs whatever initialization is necessary to prepare for the initial calls to `hasNext()` and `next()`.

   In the `PreorderIterator` constructor that we've given you, this initialization is easy, because the first node that a preorder iterator visits is the root of the tree as a whole. For a postorder iterator, however, the first node visited is not necessarily the root of the tree as a whole, and thus you will need to perform whatever steps are needed to find the first node that the postorder iterator should visit, and initialize the iterator's field(s) accordingly.

4. Implement the `hasNext()` method in your iterator class. Remember that it should execute in *O*(1) time.

5. Implement the `next()` method in your iterator class. Make sure that it includes support for situations in which it is necessary to follow one or more `parent` links back up the tree, as well as situations in which there are no additional nodes to visit. If the user calls the `next()` method when there are no remaining nodes to visit, the method should throw a `NoSuchElementException`.

6. Add a `postorderIterator()` method to the outer `LinkedTree` class. It should take no parameters, and it should have a return type of `LinkedTreeIterator`. It should create and return an instance of your new class.

7. Test everything! ***At a minimum, you must do the following:*** In the `main()` method, add a unit test that uses the `while`-loop template shown near the start of this problem to perform a full postorder traversal of a sample tree.

Problem 9: Recursive sifting

10 points; required for grad credit; partial extra credit for others

In `Heap.java`, the `siftDown()` method uses iteration to sift an element into place. Rewrite `siftDown()` so that it uses recursion instead. As you may recall from lecture, the existing method waits until the final location of the sifted node is determined before putting it into place. For the recursive version, we recommend swapping the sifted node with its larger child when the node needs to move down a level in the heap. (Note that this swap-based approach is the one that we illustrated in the initial diagrams for sifting in the lecture notes.) Make the swap, and then call the method recursively to continue the sifting process. You should ***not*** change the header of `siftDown()` in any way. You can use the existing test code in `main()` to test your recursive version of the method.

Submitting Your Work

You should use Canvas to submit the following files:

- **for part I:** your `ps4_partI.txt` or `ps4_partI.pdf` file.
- **for part II:** your modified `LinkedTree.java` file (and, if you completed Problem 9, your modified `Heap.java` file)

Make sure to use these exact file names for your files. If you need to change the name of a Java file so that it corresponds to the name we have specified, make sure to also change the name of your class and check that it still

compiles.

1. Go to the page for submitting assignments (logging in as needed)
2. Click on the appropriate link: either `ps4, part I` or `ps4, part II`.
3. Click on the *Submit Assignment* link near the upper-right corner of the screen. (If you have already submitted something for this assignment, click on *Re-submit Assignment* instead.)
4. Use the *Choose File* button to select a file to be submitted. If you have multiple files to submit, click *Add Another File* as needed, and repeat the process for each file.
5. Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
6. After submitting the assignment, you should check your submission carefully. In particular, you should:
   - Check to make sure that you have a green checkmark symbol labeled *Turned In!* in the upper-right corner of the submission page (where the *Submit Assignment* link used to be), along with the names of all of the files from the part of the assignment that you are submitting.
   - **Click on the link for each file to download it, and view the downloaded file so that you can ensure that you submitted the correct file.**

Warning

We will not accept any files after the fact, so please check your submission carefully following the instructions in Step 6.

Important

- You must submit all of the files for a given part of the assignment (Part I or Part II) at the same time. If you need to resubmit a file for some reason, you should also resubmit any other files from that part of the assignment.
- If you re-submit a file in Canvas, it will append a version number to the file name. You do *not* need to worry about this. Our grading scripts will remove the version number before we attempt to grade your work.
- There is a *Comments* box that accompanies each submission, but we do **not** read anything that you write in that space. If you need to inform us of something about your submission, please email `cscie22@fas.harvard.edu`.
- If you encounter problems submitting your files, close your browser and start again, or try again later if you still have time. If you are unable to submit and it is close to the deadline, email your homework before the deadline to `cscie22@fas.harvard.edu`.