Problem Set 4

Due prior to lecture on Wednesday, November 17

Getting Started

To get the files that you will need for this problem set, log into `nice.harvard.edu` and enter the following command:

```
gethw 4
```

This will create a directory named `ps4` in your Harvard home directory containing the files that you need for this assignment. To change into this directory, the command is:

```
cd ~/ps4
```

The final versions of your files for Problem Set 4 should ultimately be put in this directory and submitted from there, following the directions below.

Submitting Your Work

Once you are ready to submit, make sure that all of your files are in your `ps4` directory on `nice.harvard.edu`.

To check that your code compiles, enter the following command from within the `ps4` directory: `make`

To submit your assignment, enter the following command: `make submit`

To check which files were submitted, enter the following command: `make check`

Important note: the submit commands only work if you are logged onto *`nice`*`.harvard.edu`.

I. Written Problems (50-60 points total)

1. **Uninformed state-space search** (6 points total)
You want to use state-space search to solve the following Eight Puzzle:

4	7	5
3		1
6	2	8

a. (2 points) What are the first six states to which breadth-first search (BFS) would apply the goal test? Give the states in the order in which they would be tested. *Hint:* The first state to be tested is the initial state.
b. (2 points) What are the first six states to which depth-first search (DFS) would apply the goal test if it uses a depth limit of 2? Give the states in the order in which they would be tested. See the hint above.
c. (2 points) What are the first six states to which iterative-deepening search (IDS) would apply the goal test? Give the states in the order in which they would be tested. Don't forget that IDS starts over at the initial state for each value of the depth limit. Therefore, it is possible for a given state to appear more than once among the first six states to be tested.

In answering these questions, a given state should be included more than once if it is tested more than once during the first six applications of the goal test. In addition, you should make the following assumptions:

- Each algorithm follows the approach outlined on the slide entitled "Pseudocode for Finding a Solution" (pg. 8 of the notes on state-space search; pg. 152 of the coursepack). However, previously seen states are *not* discarded.
- When generating the successors of a state, the four operators are applied in the following order: move blank left, move blank right, move blank up, move blank down.
- When adding states to the data structure used to store the yet-to-be-considered states, the states are added in the order in which they were generated. For example, if a stack is used to store the states, the successor generated by moving the blank left would be pushed onto the stack first, followed by the successor generated by moving the blank right, etc.

Please write the states as matrices of numbers, with an underscore for the blank. For example:

```
4 7 5
3 _ 1
6 2 8
```

2. **Finding the largest key in a binary tree** (12 points total; 4 points each part)
The following algorithm represents one way of finding the largest key in an instance of our `LinkedTree` class. The `maxKeyTree()` method returns the largest key in the tree/subtree whose root node is specified by the parameter of the method, and the `maxKey()` method returns the largest key in the entire tree represented by the `LinkedTree` object on which the method is invoked.

```
public int maxKey() {
    if (root == null)    // root is the root of the entire tree
        throw new IllegalStateException(“the tree is empty”);
    return maxKeyTree(root);
}
private static int maxKeyTree(Node root) {
    int max = root.key;
    if (root.left != null) {
        int maxLeft = maxKeyTree(root.left);
        if (maxLeft < max)
            max = maxLeft;
    }
    if (root.right != null) {
        int maxRight = maxKeyTree(root.right);
        if (maxRight < max)
            max = maxRight;
    }
    return max;
}
```

a. For a binary tree with n nodes, what is the time complexity of this algorithm in the best case? In the worst case? Give your answers using big-O notation, and explain them briefly.

b. 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 `maxKeyTree` 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 the maximum value. Your new method may be either recursive or iterative; the choice is up to you.

c. 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? Give your answers using big-O notation, and explain them briefly.

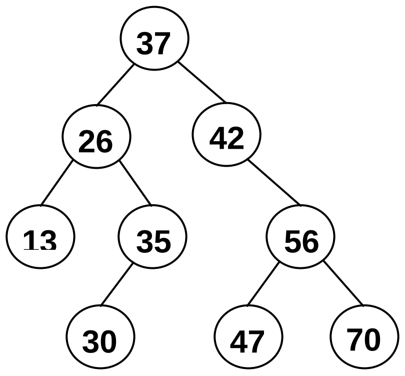
3. **Tree traversal puzzles** (10 points; 5 points each part)
   a. When a binary tree of characters (which is *not* a binary *search* tree) is listed in postorder, the result is IKHAFLMBCQ. Inorder traversal gives IAKHQCFLBM. Construct the tree.
   b. When a binary tree of characters (which is *not* a binary *search* tree) is listed in preorder, the result is BAFCIDGEHJ. Postorder traversal gives FICAHJEGDB. Construct the tree. (There is more than one possible answer in this case.)

4. **Huffman encoding** (6 points total)
   a. (4 points) Show the Huffman tree that would be constructed from following character frequencies:

Character	**Frequency**
l	8
f	10
e	17
d	20
i	40

   b. (2 points) What will be the encoding of the string *field*?

5. **Binary search trees** (8 points; 2 points each part)
   a. If a preorder traversal were used to print the keys in the nodes of the tree at right, what would be output?
   b. If a postorder traversal were used to print the keys in the nodes, what would be output?
   c. Show the tree as it will appear if 50 is inserted, followed by 10.
   d. 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.



6. **2-3 Trees and B-trees** (8 points; 4 points each part)
   Illustrate the process of inserting the sequence of keys J, E, I, H, C, F, B, G, D, A into:
   a. an initially empty 2-3 tree
   b. an initially empty B-tree of order 2
   In both cases, show the tree before and after each split, as well as the final state of the tree.

7. **Implementing a Bag using a 2-3 Tree** (10 points; *required of grad-credit students; partial extra credit for others*)
At the start of the semester, we gave you an implementation of the `Bag` interface (our `ArrayBag` class) that uses an array to store the items in the bag. In Problem Set 3, you implemented the same interface using a linked list instead of an array. Consider a third possible implementation of the `Bag` interface: one that uses a 2-3 tree to store the items, with the items themselves serving as the keys. (In other words, you should assume that the items can be compared to each other, and that such comparisons are used when inserting items into the 2-3 tree.)

   For each of the three `Bag` implementations (array-based, linked-list-based, and 2-3-tree-based), determine the worst-case time efficiencies of the `Bag` methods listed in the table below. Complete the table by filling in the appropriate big-O expressions, and provide a short explanation of expression. In addition, state clearly when (if ever) the 2-3 tree implementation is better than the other two, when (if ever) it is worse, and when (if ever) the three implementations have equivalent efficiencies.

   You may assume that the 2-3-tree-based `Bag` implements each of the three methods as efficiently as possible by making use of the following 2-3-tree operations: (1) searching for an item in the tree using the algorithm discussed in lecture; (2) inserting an item in the tree using the algorithm discussed in lecture; and (3) using an iterator to access the items in the tree one a time, in increasing order. We did not discuss iterators for 2-3 trees, but you may assume that they access the items as efficiently as possible.

worst-case time efficiencies (using big-O notation)

	method		
implementation	`addItem()`	`contains()`	`toArray()`
array-based			
linked-list-based			
2-3-tree-based			

II. Programming Problems (50 points total)

1. **Counting the number of even keys** (10 points)
   In the file `LinkedTree.java`, implement the `numEvenKeysTree()` method, which has the following header:

   ```
   private static int numEvenKeysTree(Node root)
   ```

   The method should return the number of even-numbered keys in the binary search tree or subtree whose root node is specified by the parameter. (Don't forget that you can test if a value `x` is even by checking if `x % 2 == 0`.) Make sure that your method correctly handles empty trees/subtrees – i.e., cases in which the value of the parameter `root` is `null`.

   We have given you a public method named `numEvenKeys()` that takes no parameters and that makes the initial call to `numEvenKeysTree()`, passing in the root of the entire tree. Add test code to the `main()` method that calls `numEvenKeys()` to test your implementation of `numEvenKeysTree()`.

2. **Range search** (15 points)
   In the file `LinkedTree.java`, implement the `rangeSearchTree()` method, which has the following header:

   ```
   private static void rangeSearchTree(Node root, int lower, int upper)
   ```

   This method should perform a *range search* in the binary search tree or subtree whose root node is specified by the first parameter. This means that it should search for and print all keys in the tree that fall within the specified range—greater than or equal to the value of the parameter called `lower` and less than or equal to the value of the parameter called `upper`. The relevant keys should be printed in increasing order, separated by spaces.

   Your method should use recursion to search through the tree, and it should avoid visiting nodes unnecessarily. In the same way that the search for a single item doesn't consider every node in the tree, your range search method should also avoid considering portions of the tree that couldn't fall within the specified range.

   We have given you a public method named `rangeSearch()` that performs error checking on the parameters and then makes the initial call to `rangeSearchTree()`, passing in the root of the entire tree. Add test code to the `main()` method that calls `rangeSearch()` to test your implementation of `rangeSearchTree()`.

3. **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, including ones that perform preorder, inorder, and postorder traversals. We have given you the implementation of a preorder iterator that we will go over in section (the inner class `PreorderIterator`), and you will implement a **postorder** iterator for this problem. In addition to implementing an inner class for your iterator, you should also modify the `postorderIterator()` method of the `LinkedTree` class so that it returns an instance of your new class.

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
}
```

You may assume that the tree will not be modified during the course of a given traversal. If the user calls the `next()` method when there are no remaining nodes to visit, the method should throw a `NoSuchElementException`.

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 postorder iterator class should have a space complexity of O(1).

Because you won't be using an auxiliary data structure, your `next()` method will need to follow links in the trees when advancing the position of the iterator. In order for this approach to work, it's necessary for each node to maintain a reference to its parent in the tree, in addition to the references that it already maintains to its left and right children. We have added a `parent` field to the `Node` class, but we have *not* updated the `LinkedTree` code to properly maintain this field in the nodes as the tree is updated over time. You will need to make the necessary changes to the `LinkedTree` methods as part of your work on this problem. The root of the entire tree should have a `parent` value of `null`.

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.

Here is our suggested approach to completing this problem:

a. Begin by updating the `LinkedTree` methods so that they properly maintain the `parent` field in the `Node` objects in the tree. Think about when the `parent` field should be initially set, and when (if ever) it needs to be updated.

b. Next, choose a name for the inner class that you're using for your postorder iterator, and add a skeleton for this class within the `LinkedTree` class. Include whatever private instance variables will be needed to keep track of the location of the iterator.

c. 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()`.

d. Implement the `hasNext()` method. Make sure that it executes in *O*(1) time.

e. Implement the `next()` method. Make sure 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.

f. Modify the `postorderIterator()` method in the `LinkedTree` class so that it returns an instance of your new class.

g. Test everything! One good thing to do is to find the places in the `main()` method that print an postorder traversal using the `postorderPrint()` method. After each invocation of this method, add code that prints out the keys visited during a traversal by your postorder iterator, and make sure that you get the same result from both traversals. You may also want to add code that tests your iterator's performance on trees other than the one hard-coded into the `main()` method.

We encourage you to consult our implementation of the `PreorderIterator` class during the design of your class. It can also be helpful 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.