Problem Set 4

Due prior to lecture on Wednesday, November 18

Preliminaries

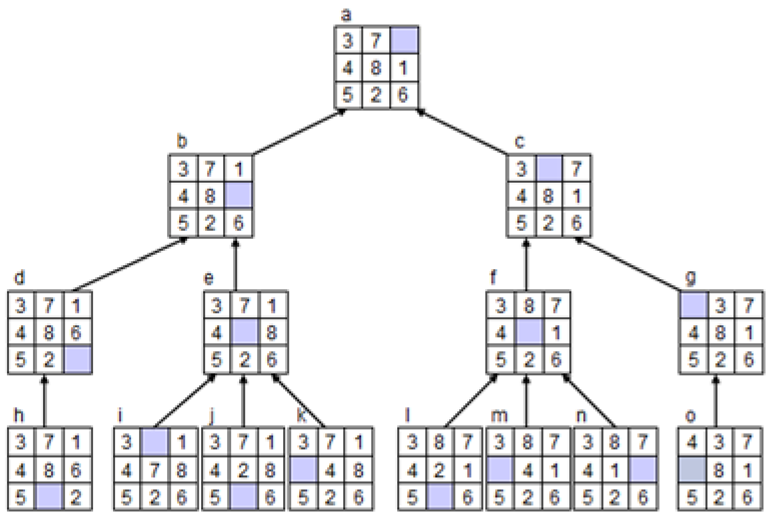
In your work on this assignment, make sure to abide by the policies on academic conduct described in the syllabus. If you have questions while working on this assignment, please come to office hours, post them on Piazza, or email `cscie22@fas.harvard.edu`.

Part I: Short-Answer ("Written") Problems (55 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`*, and put your name and email address at the top of the file.*

1. **Uninformed state-space search** (6 points total)
   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.

   The states have been labeled with lower-case letters to allow you to name them. In your answers to the questions below, you should make the following assumption: ***when the algorithms need to choose among multiple states at the same depth, they will select the one whose label 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.



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 3? Give the states in the order in which they would be tested. See the hint from part a.
c. (2 points) 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. Give the states in the order in which they would be tested.

2. **Determining the depth of a node** (12 points total; 4 points each part)
The following algorithm represents one way of finding the depth of a node in an instance of our `LinkedTree` class. The recursive `depthInTree()` method takes two parameters – a key and the root of a tree/subtree – and it returns the depth in that tree/subtree of the node with the specified key. The separate `depth()` method returns the depth in the entire tree of the node with the specified key. If the key is not found, both methods return `-1`.

```
public int depth(int key) {
    if (root == null)    // root is the root of the entire tree
        throw new IllegalStateException("the tree is empty");
    return depthInTree(key, root);
}
private static int depthInTree(int key, Node root) {
    if (key == root.key)
        return 0;
    if (root.left != null) {
        int depthInLeft = depthInTree(key, root.left);
        if (depthInLeft != -1)
            return depthInLeft + 1;
    }
    if (root.right != null) {
        int depthInRight = depthInTree(key, root.right);
        if (depthInRight != -1)
            return depthInRight + 1;
    }
    return -1;
}
```

a. 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.
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 `depthInTree` that does so. Your new method should avoid considering subtrees that couldn't contain the specified key. Like the original version of the method above, your revised method should also be recursive.

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? 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.

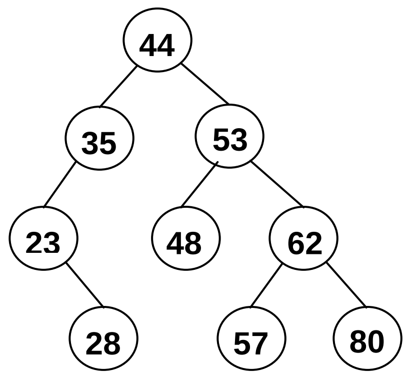
3. **Tree traversal puzzles** (10 points; 5 points each part)
   a. When a binary tree of characters (which is *not* a binary *search* tree) is listed in inorder, the result is SKBPCJRDME. Preorder traversal gives PSBKRJCMDE. Construct the tree.
   b. When a binary tree of characters (which is *not* a binary *search* tree) is listed in postorder, the result is IBGOZKHNSL. Preorder traversal gives LOGIBSHKZN. Construct the tree. (There is more than one possible answer in this case.)

4. **Huffman encoding** (7 points total)
   a. (5 points) Show the Huffman tree that would be constructed from following character frequencies:

Character	Frequency
F	8
C	14
O	17
I	20
E	40

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

5. **Binary search trees** (10 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 25 is inserted, followed by 51.
   d. Suppose we have the original tree and that 53 is deleted and then 35 is deleted, using the algorithm from the lecture notes. Show the final tree.
   e. Is the original tree balanced? Explain briefly why or why not.



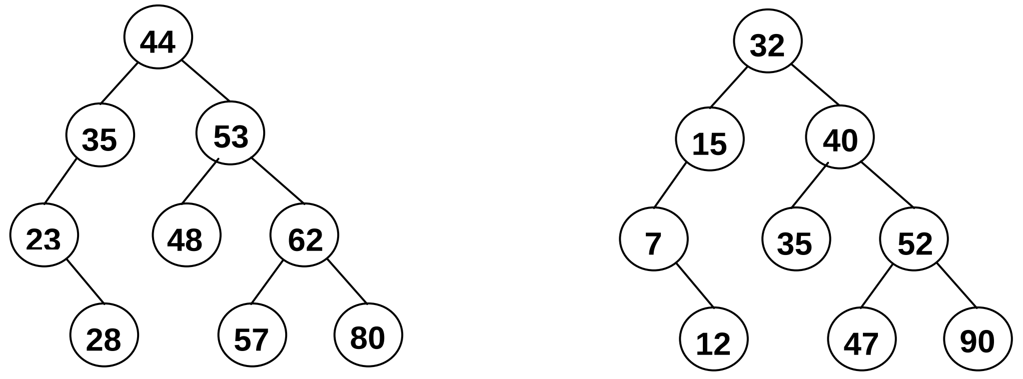
6. **2-3 Trees and B-trees** (10 points; 5 points each part)
   Illustrate the process of inserting the sequence of keys A, D, G, B, F, C, H, I, E, J into:
   a. an initially empty 2-3 tree
   b. an initially empty B-tree of order 2

   In both cases, show the tree before and after each split, as well as the final state of the tree.

Part II: Programming Problems (45-55 points total)

1. **Determining depth using iteration** (10 points)
   In written problem 2, we gave you one recursive algorithm for determining the depth of the node with a specified key, and you were asked to write a second recursive algorithm for this same task. In the file `LinkedTree.java`, implement a `depth()` method that uses **iteration** instead of recursion to determine the depth of the node with a specified key. We have given you a template for the method in `LinkedTree.java`; it currently contains just a return statement, which we have included to satisfy the compiler. You should replace that return statement with your implementation of the method. Like the recursive method that you wrote for written problem 2, your iterative method should take advantage of the fact that the tree is a binary search tree, and it should return `-1` if the specified key is not found in the tree. Add test code for your method to the `main()` method.

2. **Testing for isomorphism** (10 points)
   Two binary trees are considered to be *isomorphic* if they have the same shape. For example, the two trees below are isomorphic:



   All that matters for isomorphism is the structure of the trees; the keys and other values associated with the nodes are irrelevant. Two empty trees are isomorphic.

   In the file `LinkedTree.java`, implement the private method with this header:

```
private static boolean isomorphic(Node root1, Node root2)
```

   The method should determine if the trees whose root nodes are specified by `root1` and `root2` are isomorphic, returning `true` if they are isomorphic and `false` if they are not. We leave it to you to decide whether to use recursion or iteration. Here again, we have given you a template for the method that you should complete.

We have also given you a non-static method named `isomorphicTo()` (note the slight difference in method name). Once your method is implemented, this method will determine if the `LinkedTree` on which the method is invoked is isomorphic to the `LinkedTree` that is passed in as a parameter. It validates the parameter and then calls the private `isomorphic()` method, passing in the roots of the two trees. In the `main()` method, use calls to the `isomorphicTo()` method to test your implementation of the private `isomorphic()` method.

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 an **inorder** iterator for this problem. In addition to implementing an inner class for your iterator, you should also modify the `inorderIterator()` method of the `LinkedTree` class so that it returns an instance of your new class.

Your inorder 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 inorder traversal of the corresponding tree:

```
LinkedTreeIterator iter = tree.inorderIterator();
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 inorder 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 inorder 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 `inorderIterator()` 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 inorder traversal using the `inorderPrint()` method. After each invocation of this method, add code that prints out the keys visited during a traversal by your inorder 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.

4. **Constructing an initially balanced binary search tree** (10 points; *required of grad-credit students; partial extra credit for others*)
In the file `LinkedTree.java`, implement a constructor with the following header:

```
public LinkedTree(int[] keys, Object[] dataItems)
```

This constructor should create a `LinkedTree` containing the specified keys and data items; each element of the keys array, `keys[i]`, should be paired with the corresponding element of the data-items array, `dataItems[i]`. For full credit, the resulting tree should be a *balanced* binary *search* tree. You may assume that there are no duplicates – i.e., no repeated keys.

Hints:

- You should begin by sorting the array of keys, making sure that each change to the array of keys is accompanied by a corresponding change to the array of data items. We have given you the code needed for this in a class called `SortHelper`. You should call its `quickSort` method, passing it the array of keys and the array of data items. Because it is a static method in another class, you will need to prepend the class name when you call it.
- You may also find it useful to create a helper method that your constructor can call to add the key, data-item pairs to the tree in the appropriate order. Note that the helper method can simply call the existing `insert()` method to perform the individual insertions.

Add test code to the `main()` method that tests your implementation of the new constructor. There is a `levelOrderPrint()` method in the `LinkedTree` class that you may find helpful in this regard. It prints each level of the tree on a separate line. Although this method does *not* attempt to display the shape of the tree, seeing the keys in each level should allow you to determine if the tree is balanced.

Submitting Your Work

You should use Canvas to submit the following files:

- for Part I: your `ps4_partI.txt` file.
- for Part II: your `LinkedTree.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.

Here are the steps you should take to submit your work:

- Go to the page for submitting assignments (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Click on the appropriate link: either **Problem Set 4, Part I** or **Problem Set 4, Part II**.
- Click on the *Submit Assignment* link near the upper-right corner of the screen. (If you have already submitted something for this part of the assignment, click on *Re-submit Assignment* instead.)
- Use the *Choose File* button to select the file to be submitted.
- Once you have chosen the file that you need to submit, click on the *Submit Assignment* button.
- 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 name of the file 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.

  We will not accept any files after the fact, so please check your submission carefully!

Note: 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`