1. **Binary trees**

Consider the following binary tree (which is *not* a binary search tree):

![Binary tree diagram]

What are the ancestors of node h?
- nodes e, b, and a

What are the descendants of node c?
- nodes f, g, and i

What is the depth of node i?
- 3, because the path from i to the root has 3 edges.

What is the height of the tree?
- 4 – the maximum depth of any of the nodes.

You want to traverse the tree and print the keys in each of the nodes. What will be the output if you use each of the following types of traversal?

a) **preorder:** a b d e h j c f i g
b) **postorder:** d j h e b i f g c a
c) **inorder:** d b j h e a f i c g
d) **level-order:** a b c d e f g h i j
2. Implementing an iterator for a binary tree
In Problem Set 4, you will implement an inorder iterator for the LinkedTree class. In this section, we will implement a preorder iterator to give you a sense of the types of issues that you will need to consider.

In each case, the Java class for the iterator should implement the following interface:

```java
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();
}
```

If we create a preorder iterator object and repeatedly invoke its next() method, we should visit the nodes of the tree in the order taken by a preorder traversal. If we create an inorder iterator object and repeatedly invoke its next() method, we should visit the nodes of the tree in the order taken by an inorder traversal.

Each type of iterator will be implemented as a private inner class of the LinkedTree class. If we name the class for our preorder iterator PreorderIterator, the header of the class will look like this:

```java
public class LinkedTree {
    ...

    private class PreorderIterator implements LinkedTreeIterator {
        ...
    }
```

Before we implement this class, we need to mention a related change to the LinkedTree class that is needed to implement the iterators. We need to add a parent field to each node in the tree; it will be used to hold a reference to the node’s parent. Below is an example of a tree in which these references have been added. They are shown as arrows that have dotted lines.
In the assignment, you will need to modify the LinkedTree code so that these parent references are correctly maintained. For the sake of this exercise, we will assume that these references are already fully supported.

What instance variable or variables do we need in our PreorderIterator class?

We only need a single instance variable, a reference to the next node to be visited:

```java
private class PreorderIterator implements LinkedTreeIterator {
    private Node nextNode;
...
```

What should the constructor for the class do?

It needs to make nextNode refer to the first node to be visited by the iterator. Since a preorder traversal begins by visiting the root of the tree, the constructor is very simple:

```java
private PreorderIterator() {
    nextNode = root;
}
```

Note that root is an instance variable of the LinkedTree class. Because PreorderIterator is an inner class, it has access to the instance variables of the tree object that was used to create it. (We’ll say more about how the iterator is created in a moment.)

Note also that the constructor for your inorder iterator will be more complicated, because an inorder traversal doesn’t necessarily begin with the root of the entire tree.

What should the hasNext() method look like?

It’s also quite simple. There are more nodes to visit so long as nextNode is non-null.

```java
public boolean hasNext() {
    return (nextNode != null);
}
```

The next() method must do two things:

1) it returns the value of the next node to be visited (in this case, the value of the node’s key)
2) it advances the iterator so that it is ready for the next call to this method

If there are no additional nodes to visit, what should we do?

throw an exception:

```java
public int next() {
    if (nextNode == null)
        throw new NoSuchElementException();
    ...
The second task involves updating the state of the iterator. What variable(s) need to be updated?

nextNode

In order to accomplish the first task, what do we need to do before updating the iterator’s state?

We need to store the key of the node currently pointed to by nextNode in a local variable:

```java
public int next() {
    if (nextNode == null)
        throw new NoSuchElementException();
    int key = nextNode.key;
    ...
```

When advancing the iterator, there are several possible cases to consider, depending on the number and types of children of the node whose key we are about to return (which we will call the “current node”).

Looking at our earlier example (shown again below), what are these cases, and how would you advance the iterator in each case?

![Diagram of a binary tree](image)

**Case 1: the current node has a left child** (examples: if the current node is the 15, 12, or 10 nodes)  
A preorder traversal visits the root, then it recursively traverses the left subtree, then it recursively traverses the right subtree.

As a result, if the current node has a left child, then the next node to be visited is the left child, since it is the root of the left subtree. We can code this as follows:

```java
public int next() {
    ...
    if (nextNode.left != null)
        nextNode = nextNode.left;
```
Case 2: the current node does not have a left child, but it does have a right child
(examples: if the current node is the 6 node or 18 node above)
In this case, because there is no left subtree, the traversal goes next to the right subtree, and thus the
next node to be visited is the current node’s right child, since it is the root of the right subtree:

```java
public int next() {
    ...
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
}
```

Case 3: the current node is a leaf node (examples: if the current node is 9, 14, or 21)
In this case, we need to go back up the tree, looking for nodes that have yet to be visited.

Because a preorder traversal visits a node as soon as it encounters it (i.e., the root is visited first) and
then visits the left subtree, the only unvisited nodes are in right subtrees of nodes that we’ve already
visited. Thus, we can trace back up using parent references until we find a node with a right child
on a different path from the one that we took to get to the currentNode.

The following code almost works:

```java
public int next() {
    ...
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
    else {
        // XXX: doesn’t work correctly in all cases
        Node parent = nextNode.parent;
        // look for a node with a right child
        while (parent != null &&
            parent.right == null) {
            parent = parent.parent;
        }
        if (parent == null)
            nextNode = null;    // the iteration is complete
        else
            nextNode = parent.right;
    }
    ...
```

However, this code fails to distinguish between right children that are on the path from the root to
the current node (which have already been visited) and right children that are not on that path (and
have yet to be visited). For example, if the current node is
For example, if the current node is the 9 node, the above code would tell us to revisit the 10 node, since 10 is the right child of its parent. Instead, we want the code to tell us to visit the 14 node next. We can get around this problem by using two different references as we trace back up the tree: one to perform the traversal, and one to stay one node “behind” the first reference. That will allow us to find previously unvisited right children:

```java
public int next() {
    ...
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
    else {
        Node parent = nextNode.parent;
        Node child = nextNode;
        // look for a node with an unvisited right child
        while (parent != null &&
            (parent.right == null || parent.right == child)) {
            child = parent;
            parent = parent.parent;
        }
        if (parent == null)
            nextNode = null;    // the iteration is complete
        else
            nextNode = parent.right;
    }
    ...
}
```

Once the iterator has been advanced, what final thing needs to be done?

We need to return the previously stored key:

```java
public int next() {
    ...
    int key = nextNode.key;
    ...
    return key;
}
```

By the way, the full implementation of the preorder iterator – including the LinkedTree method that is used to create one of these iterators – is available in the version of the LinkedTree class that we’ve given you in the ps4 directory.
3. Huffman Encoding
You are given a document in which all characters are drawn from a set of 6 characters. The frequencies of these characters are given below:

<table>
<thead>
<tr>
<th>character</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>45</td>
</tr>
<tr>
<td>a</td>
<td>33</td>
</tr>
<tr>
<td>r</td>
<td>20</td>
</tr>
<tr>
<td>i</td>
<td>18</td>
</tr>
<tr>
<td>n</td>
<td>15</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
</tbody>
</table>

a) Create the Huffman tree for this document.

We begin by creating a sorted list of nodes containing character-frequency pairs:

```
  d  10
  n  15
  i  18
  r  20
  a  33
  e  45
```

We then merge the two nodes with the lowest frequencies and add their parent back to the list. The parent has a frequency that is the sum of the frequencies of its children.

```
  i  18
  r  20
    - 25
    d  10
    n  15
  a  33
  e  45
```

We continue merging the lowest-frequency pair of nodes and reinserting their parent in the list, until we are left with the root of the Huffman tree:

```
  - 25
  a 33
  - 38
  e 45
  d 10
  n 15
  i 18
  r 20
```
In the final Huffman tree above, we include the bit values associated with each branch in the tree – left branches are labeled with a 0, and right branches with a 1.

b) Use the Huffman tree to decode the following sequence of bits from the compressed document:

```
001 | 01 | 000 | 100 | 101 = nadir
```