Recitation 7
Binary Search Trees and Tree Traversal

1 Binary Search Tree (BST)

- BSTs are binary trees (every node has at most two child nodes).
- For any given root node, left sub-tree is less than its value, and right sub-tree is greater than its value.
- They can be easily searched. From a given subtree root node:
  - if query equals nodes data, done!
  - if query $<$ nodes data, recurse into left sub-tree
  - if query $>$ nodes data, recurse into right sub-tree
- Can be thought of as a branching linked list.

Figure 1 shows a binary search tree of numbers that obeys the rules above.

2 Generics

Generics are a feature of Java that allow you to build an object (such as a BST or other data structure!) without knowing the type of data it will be storing. Instead, the unknown type is given a name and treated much as a regular type. When something else uses the code, the type gets plugged in at that time (usually during compile). Example:

```
public class Thing <T> {
  private T data ;
  public Thing (T input ) {
    data = input ;
  }
  public T get Data ( ) {
    return data ;
  }
  public void setData (T input ) {
    data = input ;
  }
}
```

Then later:

```
public static void main (String [ ] args ) {
  Thing <String> variable = new Thing <String> ( " a string " ) ;
}
```

Note that many of the data structures in the Java standard library (List!) are programmed this way. This is why Eclipse tends to prompt you about generics when using them:

```
List <Integer> mylist = new List <Integer> ( ) ;
```

But there is still a problem trying to apply generics to a BST: the tree must be able to compare its elements!

```
  T thing1 = new T ( ) ;
  T thing2 = new T ( ) ;
  if (thing1 < thing2) { // Does not compile !
  ...
  }
```

This is because it does not know what type T will be, and thus cannot tell if the $<$ operator is valid for that type. The way around this is to force T to be an object implementing the comparable interface. This is one when the class is defined:

```
public class Thing <T extends Comparable <T> >{
  T thing1 = new T ( ) ;
  T thing2 = new T ( ) ;
  if (thing1.compareTo(Thing2) > 0 ) { // Compiles !
    . . . .
    }
  . . . .
  }
```

Here we are using an interface (Comparable) to promise to the compiler that type T will have a compareTo(...) method. Notice that the generic declaration says extends when it should say implements. This is the correct syntax for a generic type which implements an interface.

3 Dot

There is a program called dot on the department computers which we can use to draw our trees. It is part of the Graphviz package of software. We just have to print the tree out in the right format:

```
Digraph BST {
     5 -> 3
     3 -> 1
     3 -> 4
     5 -> 7
}
```

Save that to a file (e.g., test_data), then run dot on the file.

```
corn> dot Tps test_data o graph.ps
```

This results in a postscript file (graph.ps), which contains a picture of the graph:

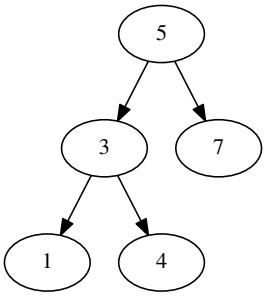

Figure 1: An example binary search tree

4 Traversing Trees

Because trees are not linear data structures, there are multiple ways to visit each of the data elements. If you think of a tree produced for an arithmetic expression (e.g., + - 12 3 4), to convert that tree to infix you would perform an "inorder" traversal of the tree:

1. Go left
2. "visit" (add the data element to a queue)
3. Go right

At each "go", it stops if there is no left or right node.

We can implement such a traversal with an `Iterator`. An `Iterator` in Java allows going over all the elements of a collection in some sequence. It provides three key methods: `hasNext`, `next` and `remove`. For now, you can ignore "remove".

To implement `next`, use a queue to order the nodes according to the type of traversal. Initialize the iterator by enqueuing all nodes in the order necessary for traversal, essentially write code that performs the inorder traversal described above. Dequeue happens for each `next` operation. (See the Prichard text 11.2 for details.)

4.1 Exercise

First, download lab7.tar, untar it, import all the java files into Eclipse.

Second, look at BinaryTreeIteratorTester.java. Use dot to draw the tree that is constructed in this file.

Third, you need to implement an inorder iterator (like producing infix from an expression tree). You have a stub file for you to use on the iterator. There is a lot of code provided for this exercise, but you should not need to look at most of it. It's all code that is taken from the textbook and modified slightly to make greater use of generics. To get started, you should write the inorder traversal which adds data elements to the queue. Then you should write the iterator methods.