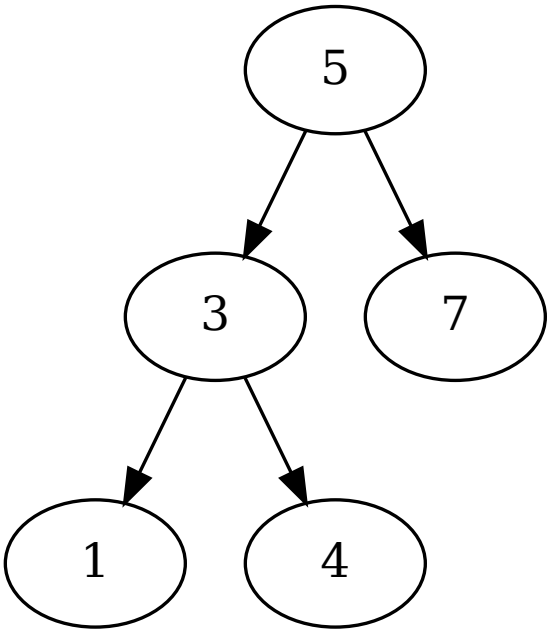
BINARY SEARCH TREES

CS 200 RECITATION 7

Binary Search Tree (BST)

- Are binary trees (every node has at most two child nodes)
- For any given node, left sub-tree is less than it's value, and right sub-tree is greater than it's value.



- They can be easily searched. From an given node:
  - if query equals node's data, done!
  - if query < node's data, recurse into left sub-tree
  - if query > node's data, recurse into right sub-tree
- Can be thought of as a branching linked list.

Generics

Are a feature of Java allowing 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 getData(){
    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 it's 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.

Given Files

- **mainClass.java** - Contains the main method and acts as a driver for our trees.
- **tree.java** - Encapsulates the tree. Keeps track of the root node, implements insert(...) and search(...), prints the tree, and can format the tree for drawing with dot.
  - insert(T input) – inserts the input into a new node in the correct position in the tree
  - search(T query) – searches for query in the tree. If it finds a matching node, it returns that node.
- **Node.java** - is a node in the tree. It should store at least one type T object, and have references to two other nodes (left and right). It implements the interface 'cs200_BST_node' to ensure that it has methods expected by tree.
- **cs200_BST_node.java** - interface implemented by Node.java. Has methods expected by by classes that use Node, like tree.

Tip about passing variables

Be careful when reassigning objects passed to a function:

```
public void foo(Object thing){
  thing = new Object(); // only has LOCAL affect, original unaffected
}

public void foo(Object thing){
  thing.setData(); // effects original object
}
```

When Java passes an object to a function, it sends a copy of that object's reference. If the function reassigns that reference to a different object, the original is not affected. Calling methods on the reference instead of reassigning it will effect the original object, since the reference still points to it.

Dot

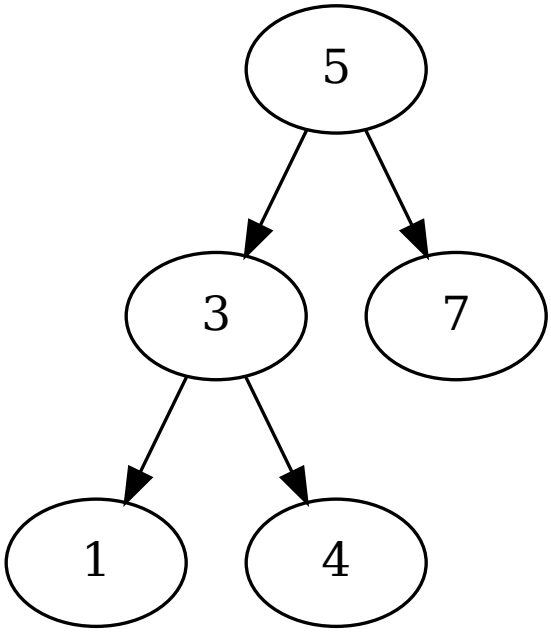
There is a program called "dot" on the department computers which we can use to draw our trees. It is part of the *Graphviz* package of software. We just have to print the tree out in the right format:

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

Save that to a file (I called it 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:



The lab machines should be able to open this file. But if not, change the "-Tps" command line flag to be "-Tpng" to generate a .png image.

The method "print_dot()" in tree.java is designed to print our BSTs in the correct format, I've tried to do the work for you on this one.

Exercise

Need to convert the skeleton files I gave into a working Binary Search Tree.

- Node.java - needs to be implemented. Look in the interface (cs200_BST_node) for a list of methods that it needs. Have it store one piece of data as type T, and have links to it's left and right children.
- In tree.java - implement the insert(...) and search(...) methods. These are very similar to each other, only differing in what they do at the end.
  - Search() moves through the tree looking for a particular node, when it finds that node it returns it.
  - Insert() moves through the tree in the same way, but it creates a new node when it finds the right place. Duplicate nodes are not allowed, so just return if the value you are trying to insert already exists.

Grading

- Sign the attendance sheet
- Checkin a .tar file containing your Binary Search Tree code
  - Here are checkin commands for each recit:
    - Monday: ~cs200/bin/checkin R7L01 R7L01.tar
    - Tuesday: ~cs200/bin/checkin R7L02 R7L02.tar
    - Wednesday: ~cs200/bin/checkin R7L03 R7L03.tar
    - Thursday: ~cs200/bin/checkin R7L04 R7L04.tar