# Assignment 4

Due: Sunday, April 2, 2017 before 11:59 pm

## *Objectives*

- Introduction to the Binary Search Tree ADT
- More practice with generics
- Introduction to tree traversals
- Introduction to the Map ADT.
- Exposure to performance measurement

## *Introduction*

In this assignment you will implement a binary search tree that is slightly different from the simple example we've been discussing in lecture. In this assignment, each node in the binary search tree will contain both a key and a value associated with that key. The elements in the tree will be ordered by the key stored at each node.

After you've completed implementing the binary search tree, we will use that implementation to implement the Map ADT.

Finally, after completing the Map implementation, you will do some experiments to contrast two different implementations of the Map interface: a linked list implementation (which have provided for you) and your own Binary Search Tree implementation.

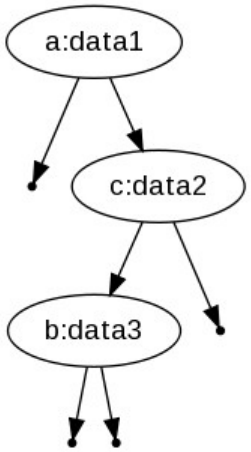
## *Quick Start*

1. Read this entire document first once through!
2. Read the comments in `BinarySearchTree.java` carefully and implement the methods until all tests pass in `BinarySearchTreeTester.java`
3. Read `Map.java` carefully and implement the methods in `BSTMap.java` until all tests pass in `MapTester.java`
4. Update your `BinarySearchTree.java` and `BSTMap.java` to count loop iterations as described on page 6 of this document.
5. Perform the experiments described on page 7 of this document and record the results in `performance.txt`

## *Part A, step 1: Binary Search Trees*

Your binary search tree implementation will use generics to allow users of the tree to specify the type for both the key and the value. For example, the following code produces the tree shown below:

```
BinarySearchTree<String, String>t1;
t1 = new BinarySearchTree<String,String>();
t1.insert("a", "data1");
t1.insert("c", "data2");
t1.insert("b", "data3");
```




When testing your code, you may find it useful to look at the trees the created and used by `BinarySearchTreeTester.java`. (A visual representation of each of the four test trees is provided on the last pages of this assignment description.)

Since it is helpful to be able to visualize trees, you've been provided with a `TreeView` class which takes an instance of a binary search tree as constructor's parameter. For example, if after the code shown above you use the following code:

```
TreeView<String,String> v1 = new TreeView(t1);
v1.dotPrint();
```

then the result will be the contents of a `dot` file[1] printed to the standard output. You can copy and paste that output text into the following web page to get a picture of your tree:

http://sandbox.kidstrythisathome.com/erdos/

1 The .dot files are used as input into a program called dot which is part of the graphviz project. `dot` performs automatic layout of directed graphs. Trees are restricted versions of directed graphs, so we can use the tool to draw our trees.

## *Part A, step 2: Entry*

It is often helpful to be able to get a list of all the key/value pairs stored in the binary search tree. In your assignment, you must implement two additional methods in `BinarySearchTree.java`:

```
    List<Entry<K,V>> entryList()
    List<Entry<K,V>> entryList(int order)
```

which returns a list of all the key/value pairs stored in tree instance. The first method returns the entries as they would be visited in level-order. The second method allows the caller to specify if they want the entries in pre, post, or in-order with the constants: `BST_PREORDER`, `BST_POSTORDER` and `BST_INORDER` respectively.

Notice that these methods are returning an instance of `List` from the built-in Java ADTs. However, the `List` can only store objects of a single type, but we need to return key/value pairs.

To get around this restriction, a common solution is to use a simple data structure called `Entry` whose only purpose is to store two items. We have provided you with such a class (i.e., `Entry.java`). The following code shows how to declare and create instances of `Entry`:

```
Entry<String,Integer> e1 = new Entry<String, Integer>("Hello", 10);
```

## *Part A, step 2 (continued): Java Collections*

As mentioned above, we are going to make use of the built-in Java ADTs. Specifically, we will use the `List` interface and the `LinkedList` implementation of the `List` interface.

In your Binary Search Tree, you can create a List of Entry with the following code:

```
List<Entry<K, V>> l = new LinkedList<Entry<K,V> >();
```

You can insert elements to the end of this particular list with:

```
l.add(new Entry<K,V>(n.key, n.value));
```

When implementing the level order traversal in the `entryList()` method, you will want to make a list of `BSTNode`, like:

```
LinkedList<BSTNode<K,V>>   nodes = new LinkedList<BSTNode<K,V> >();
```

and then you can use the methods: `removeFirst()` and `addLast()`:

```
n = nodes.removeFirst();
nodes.addLast(n.left);
```

You can read the documentation for the `List` interface and the `LinkedList` implementation at the URLs below, however, everything you need to know is covered in the above examples.

- https://docs.oracle.com/javase/8/docs/api/java/util/List.html
- https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

## *Part B: Map*

Once you've finished implementing the Binary Search Tree, you will use this to implement the Map ADT in the file `BSTMap.java`. The Map interface is specified in `Map.java`, and provides two main operations:

- `get(key)` – return the value stored at key
- `put(key,value)` – insert the key/value pair into the map. If the key already exists, update the value stored at key to be the new value.

You have to write very little code to accomplish this – simply create an instance of your `BinarySearchTree` in the `BSTMap` constructor and then use methods you've already completed in the `BinarySearchTree` to provide the services required by the Map[2].

(In your instructor's sample solution, all methods except `containsKey` are only one line. `containsKey` contains seven lines.)

2 This is a common practice. You can read about it here: https://en.wikipedia.org/wiki/Adapter_pattern

## *Part C, step 1: Experimenting with efficiency*

Your instructor has provided you with a Map implementation using linked lists (called `LinkedMap.java`). In class, we will be discussing how it is that we expect the binary search tree implementation of Map to perform less computation for the same tasks which use a list.

In this assignment we will use a very simple approach to comparing the two implementations: we will count the number of times loops execute in the get and `put` methods.

Look at the implementation of `LinkedMap.java` to see how your instructor counted the loop iterations. In particular, look at how the variables `getLoops` and `putLoops` are used.

Change your `BinarySearchTree.java` to count the loops for `insert` and `find` and then change your `BSTMap.java` to return these counts for `getPutLoopCount()` and `getGetLoopCount()` respectively.

Once you've done that, compile `Performance.java` and run it as follows:

```
java Performance linked 0
```

You should see something resembling:

```
Doing initial tests:
Your solution should match exactly for linked and be comparable for BST.
-- Instructor's solution:
[128 linked] put loop count: 499500
[128 linked] get loop count: 1000000
[128 bst   ] put loop count: 11079
[128 bst   ] get loop count: 12952
--
--Your solution:
[128 linked] put loop count: 499500
[128 linked] get loop count: 1000000
[128 bst   ] put loop count: 11079
[128 bst   ] get loop count: 12952
```

It is possible that your BinarySearchTree implementation will have slightly more or slightly fewer loops, but you should be within, say, 2% of your instructor's solution.

Once you are satisfied your loop counting is correct, perform some experiments described in the next section to compare the two implementations and document the results in `performance.txt`

### *Part C, step 2: Using Performance.java*

If you run:

```
java Performance linked 1000
```

it will gather information about the linked list implementation over 1000 put and gets. If you run:

```
java Performance tree 1000
```

it will gather information about the binary search tree implementation over 1000 `puts` and `gets`.

In the supplied `performance.txt` file, you must copy and paste the results you obtain when running the various tests described in the file.

- You need to run the linked implementation lists of size 10, 100, 1000, 10000, 100000 and 300000
- You need to run the binary search tree implementation for trees of size 10, 100, 1000, 10000, 100000, 300000 and 1000000.

Be sure to answer the questions at the bottom of the `performance.txt` file.

For example, here is the output produced by one run of the instructor's solution where there are 10000 insertions (or `puts`) of random values into the BST, and then 10000 searches (or `gets`) of random values from the BST:

```
> java Performance tree 10000
 bst   map over 10000 iterations.
[1489713840577 bst   ] put loop count: 146936
[1489713840577 bst   ] get loop count: 166935
```

The very large numbers represent the random seed used to generate random values for insertion into, and lookup from, the BST. The counts are the number of loop iterations performed in BST methods (i.e., `insert` and `find`) in order to implement the `put` and `get` in the Map (in this case a Map based on tree – i.e., the BST).

## *Submission*

Submit your `BinarySearchTree.java`, `BSTMap.java` and `performance.txt` files using connex.

A reminder that it is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

We will use plagiarism detection software on your assignment submissions.

## *Grading*

If you submit something that does not compile, you will receive a grade of 0 for the assignment. It is your responsibility to make sure you submit the correct files.

**Part A**

| Requirement | Marks |
|---|---|
| Your code follows the published coding standards | 1 |
| Your code compiles | 1 |
| Your code passes all the test cases in `BinarySearchTreeTester.java` | 10 |

**Part B**

| Requirement | Marks |
|---|---|
| Your code passes all the test cases in `MapTester.java` | 4 |

**Part C**

| Requirement | Marks |
|---|---|
| Your `performance.txt` file contains the required information. | 4 |

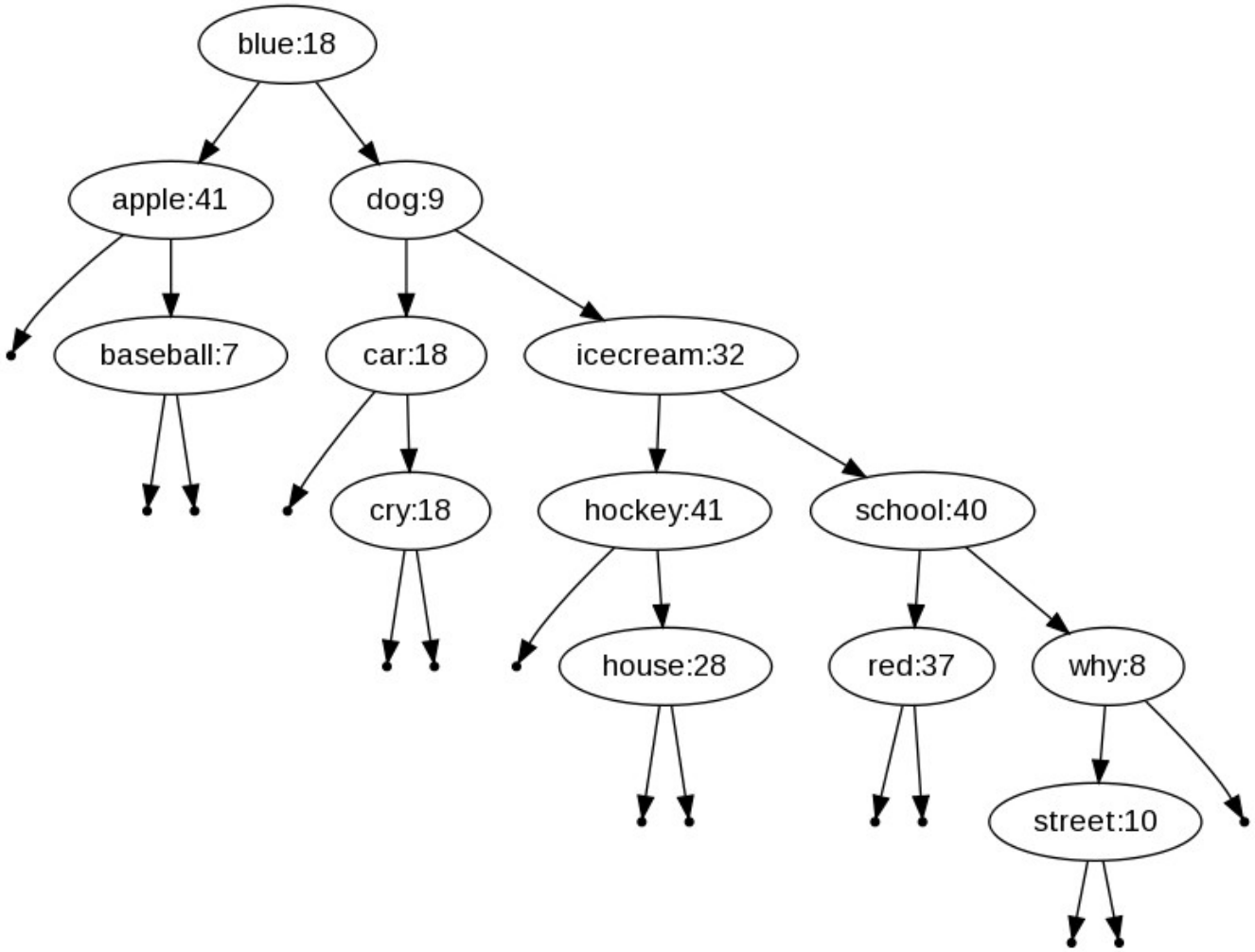
**Total 20**

**NOTE:**

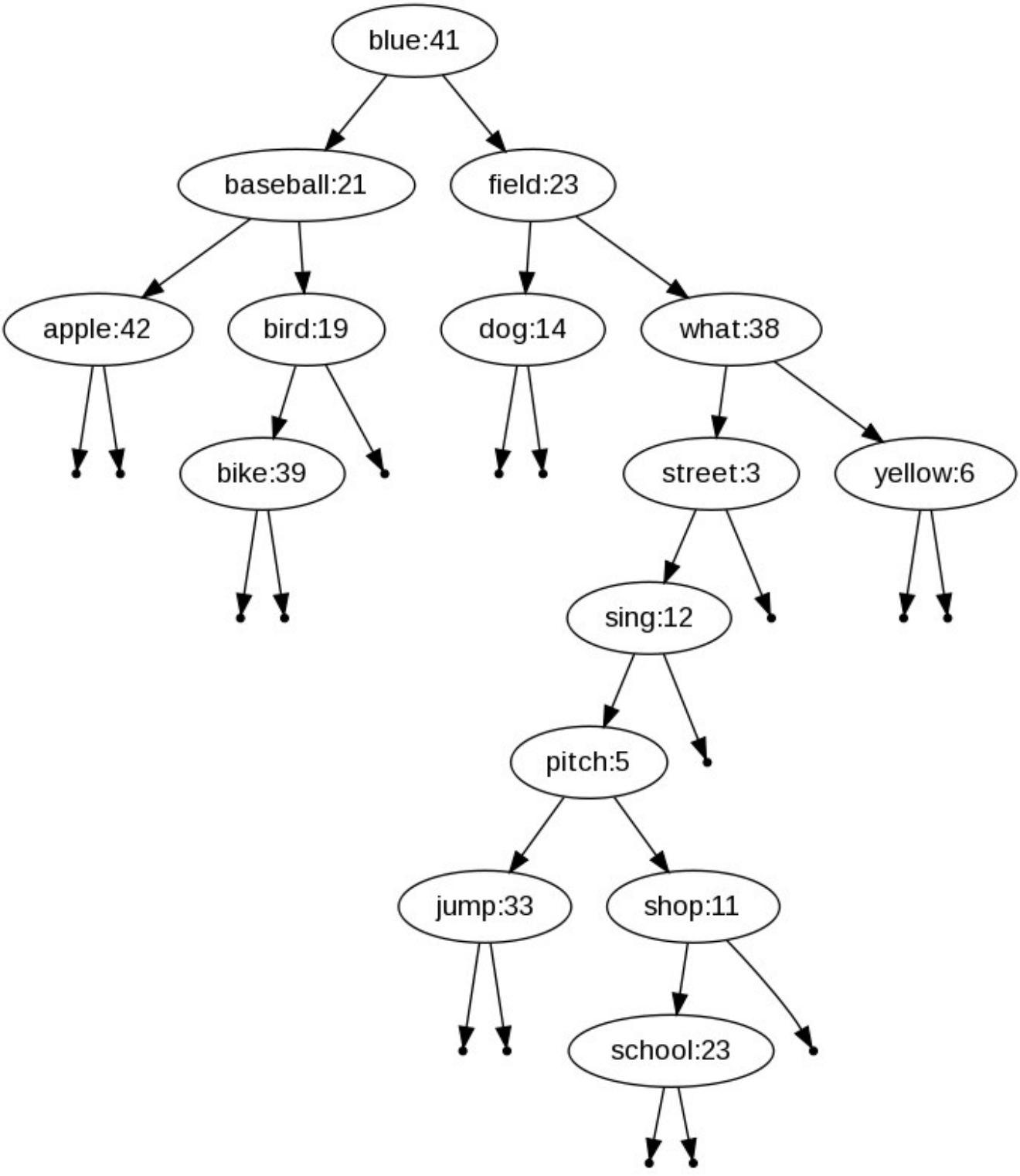
**Your submission for Part C *must be a text file*. If you submit anything else (i.e. a PDF, Word document, Open Office document, etc.) we will not mark it.**

## *Additional Diagrams (BSTs created in BinarySearchTreeTester.java)*

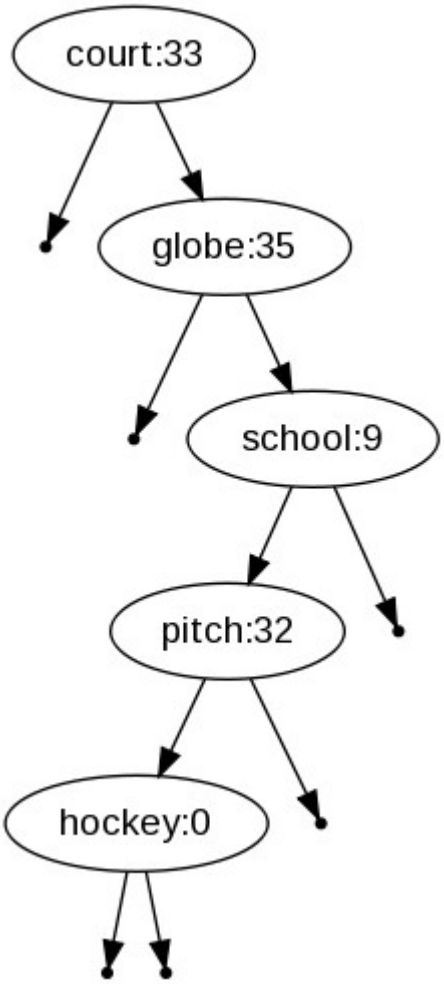
```
tree1
```



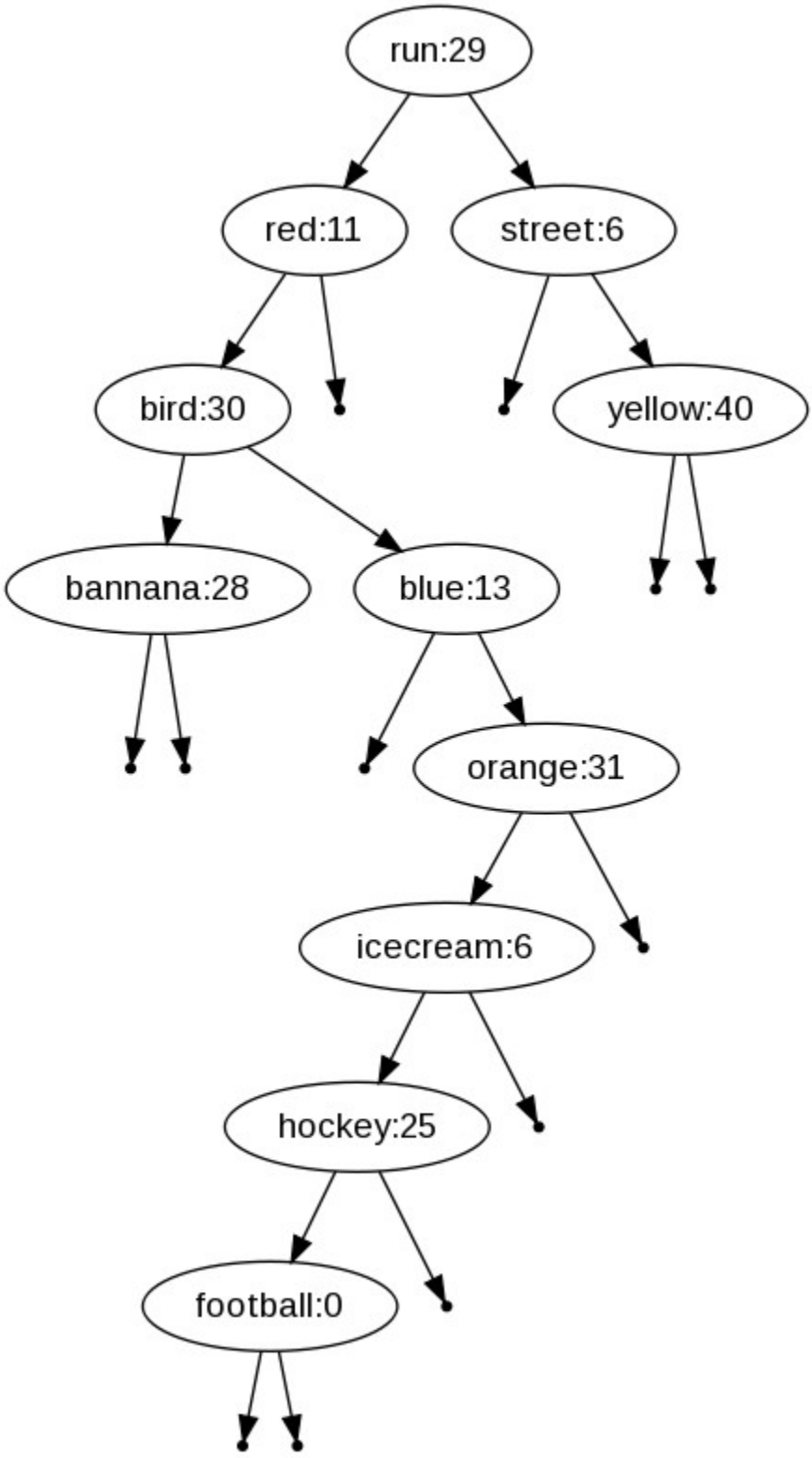
```
tree2
```


blue:41
baseball:21
field:23
apple:42
bird:19
dog:14
what:38
bike:39
street:3
yellow:6
sing:12
pitch:5
jump:33
shop:11
school:23

```
tree3
```



```
tree4
```


run:29
red:11
street:6
bird:30
yellow:40
bannana:28
blue:13
orange:31
icecream:6
hockey:25
football:0