CS2030 Programming Methodology
Semester 1, 2020/2021

28 October 2020
Problem Set #9: Trees
Suggested Guidance

So far, we have seen only *linear* data structures, eg. lists, maps, streams, arrays. Today, we will learn some basics of a *hierarchical* data structure called *trees*, which are useful for modeling many problems and solving them. [1]

A tree is a non-linear structure made up of *nodes* (aka *vertices*) linked to one another via *edges* (aka *arcs*). Figure 1 shows a tree (ignore color for now) in which each node is drawn as a circle containing a value (in this case, a number, but it could be any datatype `T`), and each edge is drawn as a line. Edges can contain values too, but for now, we will ignore this. Trees must be *acyclic*, ie. there must not be a cycle, or loop. For instance, in the figure, if there is an edge between nodes 23 and 54, then, a cycle is created. You could walk the cycle as follows: $23 \rightarrow 54 \rightarrow 76 \rightarrow 50 \rightarrow 17 \rightarrow 23$.

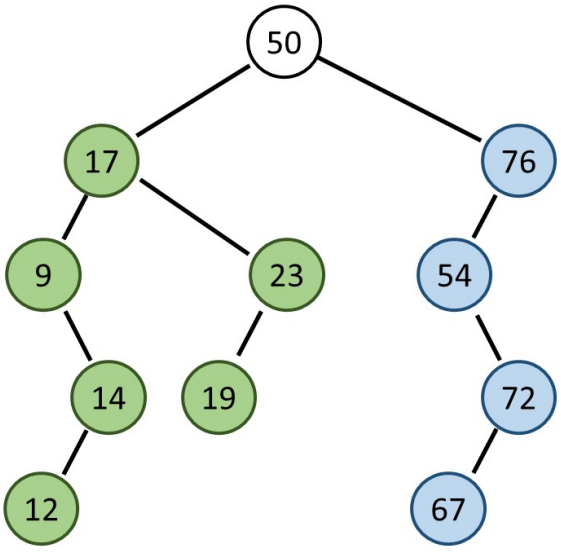


Figure 1: A binary tree.

Node 50 at the top is typically called the *root*, whose *left child* is node 17 and *right child* is node 76. In turn, node 17 has two children 9 and 23, while node 76 has only 1 child, 54, and so on. Trees are usually drawn top down, or left to right. A *binary tree* is a tree in which every node has at most 2 children. Nodes which have no children are called *leaves* or *terminal nodes*, while the others are called *internal* or *non-terminal* nodes. In this example, nodes $12, 19, 67$, are the leaves, while the rest are internal nodes.

[1] You will dive into more details in CS2040 Data Structures and Algorithms.

Notice that the green nodes and their edges also form a tree; we call this the *subtree rooted at node 17*, or alternatively, the left subtree of node 50. Likewise, the blue nodes and their edges form a subtree rooted at the right child of node 50, and is thus called its right subtree. For any non-empty tree, whether binary or not, the following are true:

(1) There is only one root node.

(2) Every node, except the root, has exactly one parent node.

(3) If the tree has n nodes, then it has exactly $n - 1$ edges, and vice versa.

(4) The tree is *connected*, ie. you can "walk" from any node to any other node.

(5) The tree is acyclic.

We may easily implement binary trees in Java as follows. By now, we prefer lazy evaluation where possible.

```
public class BinaryTree<T extends Comparable<T>> {

    private final T value;
    private final Supplier<BinaryTree<T>> leftTree;
    private final Supplier<BinaryTree<T>> rightTree;
    private final boolean amIEmpty;

    //Use a Supplier to freeze the left and right subtrees
    public BinaryTree(T value, Supplier<BinaryTree<T>> left,
                      Supplier<BinaryTree<T>> right) {
        this.value = value;
        this.leftTree = left;
        this.rightTree = right;
        this.amIEmpty = false;
    }

    public T value() {
        if (this.isEmpty())
            throw new IllegalArgumentException("value: empty tree!");
        return this.value;
    }

    public BinaryTree<T> leftTree() {
        if (this.isEmpty())
            throw new IllegalArgumentException("leftTree: empty tree!");
        return this.leftTree.get();
    }



    public BinaryTree<T> rightTree() {
```

```
        if (this.isEmpty())
            throw new IllegalArgumentException("rightTree: empty tree!");
        return this.rightTree.get();
    }
    //..
}
```

Read the rest of the code in `BinaryTree.java`. Since, we want the ability to compare node values, we declare `T extends Comparable<T>`. Notice also that the definition is recursive: a `BinaryTree` is either empty, or holds a value and has right and left children of type `BinaryTree` (albeit frozen).

We may use a binary tree for efficient search. In Figure 1, notice that all numbers on the left subtree of the root (node 50) are smaller than the root, while all numbers on the right subtree are larger. Furthermore, this property is also true *at every node!* This property makes the tree a *binary search tree*.

Let's see how we may exploit this property to search for 23 in the tree. Begin at the root, and compare its value with our search item (23). The root value (50) is larger, which means that our search item must lie in the left subtree (if it is there at all). We now go to the left child and continue our search there. Its value is 17, which is less than 23. This means we must next go to the right subtree of 17. We do so, and arrive at node 23, which equals our search item. We now stop the search and declare success.

Suppose instead we wish to search for 30. We follow the procedure as before, and at node 23, we conclude that we must next visit its right subtree (since $23 < 30$). But node 23 has no right child. This means our search item cannot be in the tree at all. We now stop and declare failure.

You now see how this may be generalized: to search for x, start at the root and check to see if its value is equal to, or less than, or greater than x. Depending on the check result, we next do one of three things, respectively: stop and declare success, or recursively search the right subtree, or recursively search the left subtree. If there are no more children to visit, we stop and declare failure. This search procedure is easily coded as follows:

```
public boolean eagerContains(T thing) {
    // Eager evaluation
    // Return true if this tree contains item, false otherwise
    if (this.isEmpty())
        return false; //nothing to search

    int compareResult = this.value().compareTo(thing);
    if (compareResult == 0)
        return true; //found item

    else if (compareResult > 0)
        //current node is larger than item, so search left subtree
```

```
        return this.leftTree().eagerContains(thing);

    else //current node is smaller, so search right subtree
        return this.rightTree().eagerContains(thing);
}
```

In our search for 23 or 30 above, we compared our search item at three nodes: $50, 17$ and 23. This was sufficient to declare success or failure. We didn't have to check every node because of the tree's property. It may be proven that *on average*, searching for any item in an n-node binary search tree takes about $\log_2(n)$ comparisons. Our tree above has 11 nodes, and $\log_2(11) \approx 3.46$ which is about the number of comparisons we used, as predicted by the theory. If we plot $\log_2(n)$ vs n, we will see that the curve increases very slowly. For large n, say, $n = 10^6, \log_2(5 \times 10^6) \approx 22$. This means that to search for an NRIC number in Singapore (population ≈ 5.64 million), only 22 comparisons are needed on average. This is efficient indeed!

Q1. Searching the tree.

(a) If `eagerContains(61)` is called using the tree in Figure 1, how many times is `compareTo` invoked?

(b) The analysis above was for the average case. Unfortunately, in the worst case, searching for an item may require checking every node in the tree. When will this happen? Illustrate with a tree.

(c) Complete the code below to implement a lazy version of `eagerContains`. This instance method in `BinaryTree.java` returns a `Lazy<Boolean>` object that captures the result of checking if the search item is in this tree. The result is computed only when `get` is called.

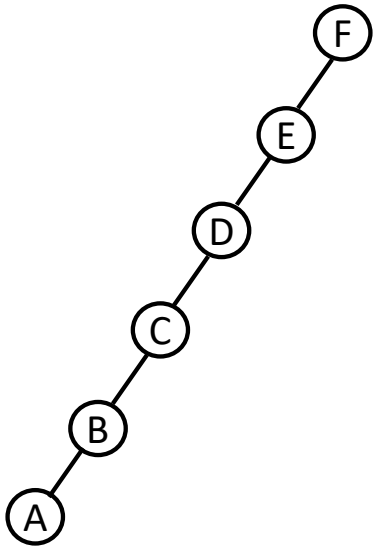
```
//Instance method in BinaryTree.java
public Lazy<Boolean> contains(T searchItem) {
    // Insert code here.
}

public class Lazy<T> {
    private final Supplier<T> value;

    public Lazy(Supplier<T> v) {
        this.value = v;
    }

    public T get() {
        return this.value.get();
    }
}
```

(d) Write an instance method `max` that lazily returns (via the `Lazy` object) the largest value in the binary search tree.

Solution:

(a) 5 times. The nodes being compared are, in order, $50, 76, 54, 72, 67$.

(b) This can happen if the tree is a "stick". Searching for `A` in the tree below will check every node.



(c) The easiest solution is to wrap the call in a `Lazy` object:

```
public Lazy<Boolean> contains(T searchItem) {
    return new Lazy<>(()-> eagerContains(searchItem));
}
```

(d) First, create a `private` method to eagerly find the maximum; then wrap the call in a `Lazy` object as before. Note that in a binary search tree, the maximum value is at the righmost leaf, if it exists, or the rightmost parent.

```
public Lazy<T> max() {
    return new Lazy<>(()-> this.eagerMax());
}

private T eagerMax() {
    if (this.isEmpty())
        return null;

    else if (this.rightTree().isEmpty())
        return this.value();

    else
        return this.rightTree().eagerMax();
}
```

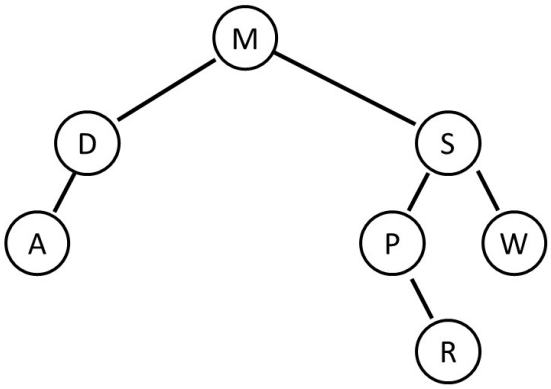


Figure 2: A `BinaryTree<String>`.

Q2. To eagerly get values out of the tree, we need to *traverse* it. One type of traversal is called *pre-order depth first traversal*. Starting at the root, the root value is "consumed" by an action, then the left subtree is recursively traversed, followed by the right subtree. The instance method to do so is straightforward:

```
public void preOrder(Consumer<T> action) {
    if (!this.isEmpty()) {
        action.accept(this.value());
        this.leftTree().preOrder(action);
        this.rightTree().preOrder(action);
    }
}
```

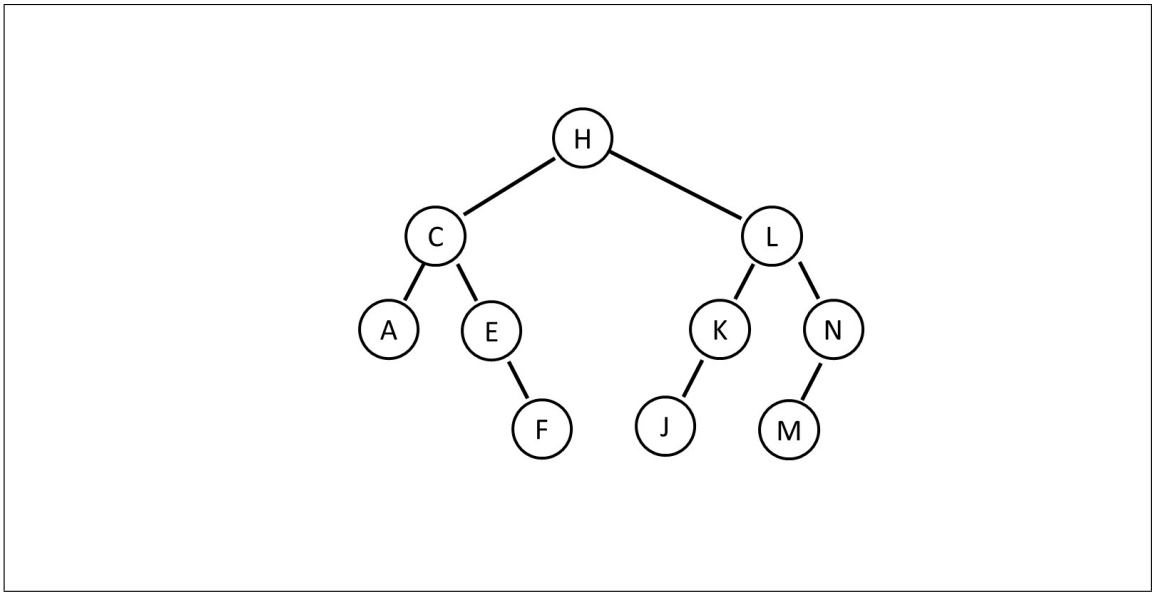
(a) Using the tree in Figure 2, determine the output of:
`preorder(x -> System.out.print(x + ", "));`

(b) Is there a binary search tree that will produce the following `preorder` output? If so, draw it; otherwise explain why not. `H, C, A, E, F, L, K, J, N, M,`

Solution:

(a) `M, D, A, S, P, R, W,`

(b) Yes, this tree:



Q3. Adding to, and mapping over, a tree.

(a) Read the code for the `fromList` and `add` methods. Notice that `add` lazily adds each item to the tree.

```
public static <T extends Comparable<T>> BinaryTree<T>
    fromList(List<T> list) {
    BinaryTree<T> result = BinaryTree.makeEmpty();
    for (T item : list)
        result = result.add(item);
    return result;
}

public BinaryTree<T> add(T thing) {
    if (this.isEmpty())
        return new BinaryTree<>(thing,
                                ()-> makeEmpty(),
                                ()-> makeEmpty());

    int compareResult = this.value().compareTo(thing);
    if (compareResult == 0)
        //already in tree
        return this;

    else if (compareResult > 0)
        //add to left tree
        return new BinaryTree<>(this.value(),
                                ()-> this.leftTree().add(thing),
                                ()-> this.rightTree());
    else //add to right tree
        return new BinaryTree<>(this.value(),
                                ()-> this.leftTree(),
```

```
                                    ()-> this.rightTree().add(thing));
}
```

Draw the resulting tree from running:

```
bt = BinaryTree.fromList(Arrays.asList(30, 8, 25, 6, 45, 0, 60, 80));
```

Draw the tree fully, as if it was constructed eagerly.

(b) We may lazily map over trees. The code is easy enough:

```
public <U extends Comparable<U>> BinaryTree<U> map(Function<T,U> f) {
    if (this.isEmpty())
        return BinaryTree.makeEmpty();
    else
        return new BinaryTree<>(f.apply(this.value),
                                ()-> this.leftTree().map(f),
                                ()-> this.rightTree().map(f));
}
```

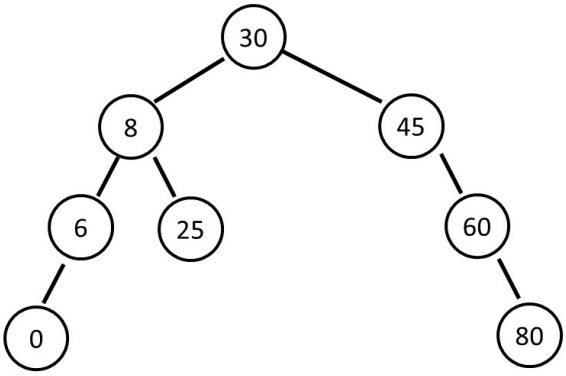
Eagerly draw the resulting tree from running:

```
bt.map(x -> 100 - x);
```

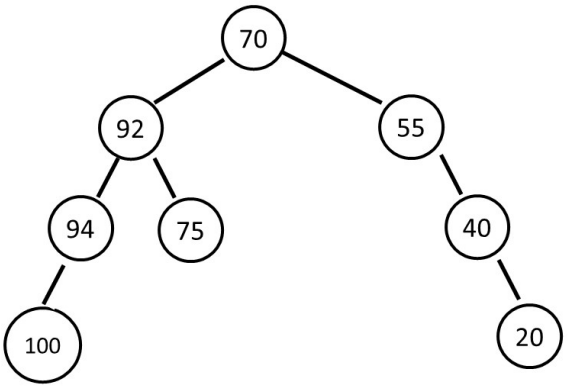
(c) Does this tree still have the Binary Search Tree property? How should you recode `map` to correct this problem?

Solution:

(a) Answer:



(b) Answer:



(c) Clearly, the Binary Search Tree (BST) property is broken. To correct this, we need to restore the property after mapping. Let's relegate our original `map` to a private helper method, and recode our `map` to call the helper method, followed by restore:

```
public <U extends Comparable<U>> BinaryTree<U>
           map(Function<T,U> f) {
    return this.mapHelper(f).restore();
}

private <U extends Comparable<U>> BinaryTree<U>
           mapHelper(Function<T,U> f) {
    if (this.isEmpty())
        return BinaryTree.makeEmpty();
    else
        return new BinaryTree<>(f.apply(this.value()),
                                ()-> this.leftTree().mapHelper(f),
                                ()-> this.rightTree().mapHelper(f));
}
```

To restore a tree, an easy way is to retrieve all its elements via `preOrder` into a list, and then build a new tree using `fromList`:

```
private BinaryTree<T> restore() {
    List<T> list = new ArrayList<>();
    this.preOrder(list::add);
    return BinaryTree.fromList(list);
}
```