cis112-week09: Trees

v2025-04-20

Content

Introduction

This week we cover

- A nonlinear data structure `Tree`.
- A new technique for debugging.
- Extending a given library.

Web resources

- Data Structures
- Abstract Data Type (ADT)
- Tree
- Graph Theory

Debugging

- Debug / Standard build of Java application

Nonlinear Data Structures

Trees are mathematical objects in Graph Theory. That is, a tree is a special graph.

We will focus on trees as data structures. So far, we work on `arrays` and `linked lists`. They both are *linear* structures. That is, for each item there are 2 related items: a previous item and a next item.

Tree is the first *nonlinear* data structures that we encountered. In `Binary Tree`, for each item there is a *parent*, a *left subtree* and a *right subtree*, that is, there are exactly 3 related items. In N `-ary Tree`, for each item, there may be $1 + N$ related items.

When we cover `Graphs`, we see that an item can be related to much more items.

Binary Tree

A Binary Tree composed of `Node`s. A `Node` should support the methods in `NodeBinaryInterface` for navigation and tree manipulations.

NodeBinaryInterface

In many cases, operations in tree are independent of the `data` field. Therefore, it is useful to have an interface that has just the necessary methods for this. Any object type that implements the interface can use these operations.

```java
public interface NodeBinaryInterface<T> {
    NodeBinaryInterface<T> left();
    NodeBinaryInterface<T> right();
    T data();
    T canonical();
}
```

Library `LibTree`

The common operations that are independent of `data` are defined in library `LibTree`.

- `plot` method is one of the most useful operations, which visualizes the tree. Of course, it is not very helpful for large trees but it is very useful for small trees.
- `canonical` method produces a *canonical representation*, that is, unique representation of the tree. It is used in jUnit tests.
- `height` method calculates the height of the node or the tree itself.
- There are a few tree traverse algorithms are also implemented in `LibTree`.

Note. Note that thanks to interface `NodeBinaryInterface`, methods in `LibTree` can be used in any tree that are based on this interface. So you do not need to rewrite any code. You *reuse* the code written in the library.

Node

A `Node<T>` has three fields:

1. `data` is any type of object `T`
2. `left` refers to the root node of the left subtree if it exists; `null` otherwise.
3. `right` refers to the root node of the right subtree if it exists; `null` otherwise.

```java
public class MyNode<T> implements NodeBinaryInterface<T> {

    T data;
    MyNode<T> left;
    MyNode<T> right;

    public MyNode() {
        this(null);
    }

    public MyNode(T data) {
        this.data = data;
        left = null;
        right = null;
    }

    public NodeBinaryInterface<T> left() {...}
    public NodeBinaryInterface<T> right() {...}
    public T data() {...}
    public T canonical() {...}
    public String toString() {...}
}
```

Tree

Tree has 3 constructors.

```java
public class MyBinaryTree<T> {

    private MyNode<T> root;

    public MyBinaryTree() {...}

    public MyBinaryTree(T data) {...}

    public MyBinaryTree(T data//
            , MyBinaryTree<T> left//
            , MyBinaryTree<T> right//
    ) {...}
    ...
}
```

Tree Construction

A tree is constructed using `MyNode(T data)` and `MyBinaryTree(T data, MyBinaryTree<T> left, MyBinaryTree<T> right)` constructors as in the following example.

```java
public class MyBinaryTreeConstructor {
    ...
    public static MyBinaryTree<String> constructBT_S_t_l() {
        System.out.println("\n-" + StackWalker.getInstance().walk(s -> s.skip(0).findFir

        MyBinaryTree<String> tree;
        tree = new MyBinaryTree<>("t" //
                , new MyBinaryTree<>("l") //
                , null//
        );
        return tree;
    }
}
```

A new technique for debugging

1. In `MyBinaryTreeConstructor` in package `theory`, there is a two level debugging control by `DEBUG` and `DEBUG2` flags.

   ```java
   private static final boolean DEBUG = false;
   private static final boolean DEBUG2 = false;
   ```

   For example, if `DEBUG` is `true` then tree is plotted in `treeInfo` method of `MyBinaryTreeConstructor`.

   ```java
   if (DEBUG) {
       tree.plot();
   }
   ```

   The good thing is that, if `DEBUG` is `false`, then the Java compiler is smart enough not compile this part of the code. So, the compile code does not have debugging lines that you have during development.

   Remark.

   - Note that every class has its own `DEBUG`. This allows you to debug the class that you are currently working on by setting `DEBUG` to `false`, while already debugged classes have `DEBUG` is set to `false`.
   - If necessary, have a third level debugging by defining `DEBUG3`.

2. Use this technique in your developments in this lab.

Extending a given library

Consider `theory`. There is `MyBinaryTreeConstructor`, which constructs a number of trees. Similarly, we have `MyBinaryTree` and `MyBinaryTree_Test`.

- We consider `theory` as an external library developed by somebody. Therefore, we are not allowed to change.
- In `lab`, we develope our own class called `MyBinaryTreeConstructorExtended`, which extends `MyBinaryTreeConstructor`.

```java
public class MyBinaryTreeConstructorExtended extends MyBinaryTreeConstructor {...}
```

Hence, all public methods of `MyBinaryTreeConstructor` can be accessable by `MyBinaryTreeConstructorExtended`. In addition to that, we will develop two new methods in `MyBinaryTreeConstructorExtended`, namely, `constructBT_S_Full_Level3` and `constructBT_S_ExpressionQuadratic`.

- Similarly, `LibTreeExtended` extends `LibTree`. We will develop two new methods `size` and `find` in `LibTreeExtended`.

Development and Testing Convention

We have been applying the same naming convention.

- Suppose develop class `X`. Usually, there is no `main` method in `X`. So we cannot run it directly.
- To test it during development, we use `X_Test` class, which has a `main` method. Hence, it can run as an application.
- On the other hand, to test whether our development meets the specification or not, we use `X_jUnit`, which is a unit test battery. It does not have a `main` method. It runs as jUnit test.

  The recommended software engineering practice is first preparing the jUnits based on the specification. Then as the software is developed, run the unit test on the developed system.

In this lab, we have

- in package `theory`
  - `MyBinaryTree`
  - `MyBinaryTree_Test`
  - `MyBinaryTreeConstructor`
  - `MyBinaryTreeConstructor_Test`
- in package `lab`
  - `LibTreeExtended`
  - `LibTreeExtended_jUnit`
  - `LibTreeExtended_Test`
  - `MyBinaryTreeConstructorExtended`
  - `MyBinaryTreeConstructorExtended_jUnit`
  - `MyBinaryTreeConstructorExtended_Test`

Goal

G0. Fill `StudentInfo`

1. Fill your data in `StudentInfo`.

G1. Constructing Trees

1. Consider `MyBinaryTreeConstructorExtended` which extends `MyBinaryTreeConstructor`

in package `theory`.

```java
public class MyBinaryTreeConstructorExtended extends MyBinaryTreeConstructor {...}
```

2. In `MyBinaryTreeConstructorExtended`, complete `constructBT_Full_Level3` method so that the following tree is constructed.

   Use `MyBinaryTreeConstructorExtended_Test` during development.

```
      /15
    /7
      \14
  /3
      /13
    \6
      \12
>1
      /11
    /5
      \10
  \2
      /9
    \4
      \8
```

3. Make sure that you pass related tests in `MyBinaryTreeConstructorExtended_jUnit`.

G2. Extending Tree Library

Definition. Number of nodes in a tree is called *size*.

1. Note that there is no `size` method in `LibTree` in package `theory`.

   Consider `LibTreeExtended`, which extends `LibTree`, is a new library that provides missing functionality, such as `size`, in `LibTree`.

```java
public class LibTreeExtended<T> extends LibTree{...}
```

2. In `LibTreeExtended`, complete `size` method.

   Use `LibTreeExtended_Test` during development.

3. Make sure that you pass related tests in `LibTreeExtended_jUnit`.

Challenge

C1. Constructing Trees

1. In `MyBinaryTreeConstructorExtended`, complete `constructBT_ExpressionQuadratic` method so that the following tree is constructed for expression

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Use `MyBinaryTreeConstructorExtended_Test` during development.

Test your code with `expressionForQuadratic_Test` method.

```
    /a
  /*
    \2
>/
    /sqrt
          /c
        /*
            /a
          \*
            \4
      \-
        \sq
          \b
  \+
    \neg
      \b
```

Hint.

- Start construction of the tree from the lowest leaves.
- Use `neg`, `sq` and `sqrt` for negation, square and square root, respectively.

Remark. The expression

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

is the solution of quadratic equation

$$ax^2 + bx + c = 0.$$

Use `LibTreeExtended_Test` during development.

2. Make sure that you pass related tests in `MyBinaryTreeConstructorExtended_jUnit`.

C2. Extending Tree Library

Searching is an important operation in Computer Science. In the coming weeks we will be dealing with searching. We will organize data in the tree is a way in such a smart way that searching will be

Note that there is no `find` method in `LibTree` in package `theory`.

1. In `LibTreeExtended`, complete `find` method.

   Use `LibTreeExtended_Test` during development.

2. Make sure that you pass related tests in `LibTreeExtended_jUnit`.