

Problem Set 5

Due: Friday, October 25

Overview: In this problem set, you will get experience with trees and with mutable data structures.

Download: To begin this assignment, you should download a copy of the directory `~cs230/download/ps5`. In your local copy of this directory, you will:

- Modify `PS5IntTreeOps.java` in Problem 1.
- Modify `TreeParser.java` in Problem 2.
- Modify `DestructiveReverse.java` in Problem 3.

Submission:

- For Problem 1, your hardcopy submission should be your final version of `PS5IntTreeOps.java`.
- For Problem 2, your hardcopy submission should be your final version of `TreeParser.java`.
- For Problem 3, your hardcopy submission should be your final version of `DestructiveReverse.java`.

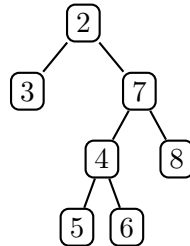
Your softcopy submission for this problem should be the entire `ps5` directory.

Remember to include a signed cover sheet (found at the end of this problem set description) at the beginning of your hardcopy submission.

Problem 1 [50]: IntTree Methods

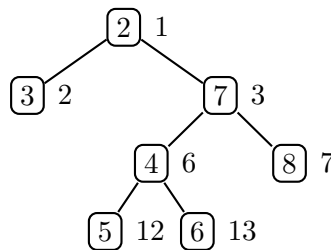
In this problem, you will define methods that manipulate integer trees. Skeletons for these methods appear in the file `PS5IntTreeOps.java`.

For this problem, it is helpful to know a number of definitions. The sample tree T below will be used as an example in many of the definitions:

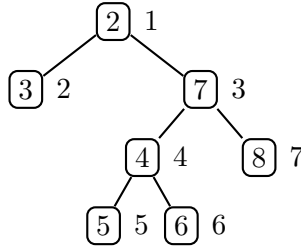


- The *height* of a tree is the longest number of edges from the root to a leaf. The height of T is 4.
- A tree is *balanced* if for every node the height of its two subtrees differ by no more than one. T is not balanced because its left subtree has height 1 and its right subtree has height 3. However, T 's right subtree is balanced.
- An integer tree is a *binary search tree (BST)* if for every node in the tree, the value of the node is \geq all values in its left subtree and \leq all values in its right subtree. T is not a BST because its left subtree contains a value (2) greater than its root. However, the right subtree of T is a BST.
- The *binary address* of a tree node is defined as follows.
 - The binary address of the root of a tree is 1.
 - If the binary address of a node is n , the binary address of its left child is $2n$ and the binary address of its right child is $2n + 1$.

For example, here is a version of T in which each node has been annotated with its binary address:



- The *pre-order address* of a tree node is an integer (starting at 1) that indicates the order in which that node would be visited in a pre-order traversal of the tree. For example, here is a version of T in which each node has been annotated with its pre-order address:



Based on the above definitions, write definitions of the following `IntTree` functions within the class `PS5IntTreeOps`.

a. [10]

public static boolean isBalanced (IntTree t);

Returns `true` if `t` is balanced and false otherwise. You may find it helpful to use the absolute value function `Math.abs` in your solution.

b. [10]

public static boolean isBST (IntTree t);

Returns `true` if `t` is a binary search tree and false otherwise. Be careful – it is *not* sufficient to simply compare the value at the root of the tree to the values at the root of its left and right subtrees.

c. [10]:

public static IntTree addBinaryAddress (IntTree t);

Returns a new tree that has the same structure as `t` where the value at every node has been incremented by the binary address of the node. *Hint*: define `addBinaryAddress` in terms of an auxiliary recursive method that takes two arguments – a tree and a binary address – and returns a tree.

d. [10]:

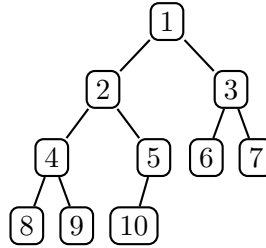
public static IntTree addPreorderAddress (IntTree t);

Returns a new tree that has the same structure as `t` where the value at every node has been incremented by the preorder address of the node. *Hint*: define `addPreorderAddress` in terms of an auxiliary recursive function that takes two arguments – a tree and a pre-order address – and returns a tree.

e. [10]:

public static IntTree breadthTree (int n);

Returns a tree with `n` nodes whose binary addresses are the values 1 through `n`. The value at each node should be its binary address. For example, `breadthTree(10)` should yield the following tree:



Hint: define `breadthTree` in terms of an auxiliary recursive function that takes two arguments.

Notes

- Flesh out the skeletons for the methods in the class `PS5IntTreeOps`.
- You can use the methods of the `IntTree` contract prefixed by `IT`: `IT.leaf`, `IT.isLeaf`, `IT.node`, `IT.value`, `IT.left`, and `IT.right`.
- In addition to the standard `IntTree` methods, the following helpful methods can also be accessed by prefixing them with `IT`:

public static int height (`IntTree t`);

Returns the height of `t`.

public static int size (`IntTree t`);

Returns the number of nodes in `t`.

public static int BSTMin (`IntTree t`);

If `t` is a non-empty binary search tree, returns the least integer in `t`. If `t` is an empty tree, returns `Integer.MAX_VALUE`. The behavior of `BSTMin` is unspecified if `t` is not a binary search tree.

public static int BSTMax (`IntTree t`);

If `t` is a non-empty binary search tree, returns the greatest integer in `t`. If `t` is an empty tree, returns `Integer.MIN_VALUE`. The behavior of `BSTMax` is unspecified if `t` is not a binary search tree.

- The invocation `java PS5IntTreeOps method-name` will run some standard test cases for the method named `method-name`. The invocation `java PS5IntTreeOps` will run standard test cases for all five methods in the problem. You are encouraged to read and understand the testing code and to add any test cases you think are helpful.

Problem 2 [30]: Parsing Strings Into Trees

Background

It is easy to convert a tree into a string representation via the following `toString` method:

```
public String toString (IntTree t) {
    if (IT.isLeaf(t)) {
        return "*";
    } else {
        return "(" + IT.left(t) + " " + IT.value(t) + " " + IT.right(t) + ")";
    }
}
```

This problem is about a `fromString` method that inverts this process – i.e., converts a string representation `S` of an integer tree into an `IntTree` instance `T` that has `S` as its string representation. This process is called "parsing" the string into a tree. The goal of this problem is to implement the method.

If the string parameter to `fromString` is a well-formed tree representation for an integer tree, then `fromString` should return the appropriate instance of `IntTree`. However, if the string parameter to `fromString` is ill-formed, then `fromString` should throw an exception indicating this fact.

For example, here are the results of `fromString` on some sample input strings. First some examples of well-formed trees:

```
fromString("*") =
*
```

```
fromString("( * 17 * )") =
( * 17 * )
```

```
fromString("(( * 54 * ) 255 ( * 3 * ))")=
(( * 54 * ) 255 ( * 3 * ))
```

```
fromString("((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))") =
((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))
```

Fig. 1 shows some examples involving strings that are not interpretable as trees. In these cases, the exceptions thrown by `fromString` have been caught by the testing program and an error message has been printed out.

A parsing process is usually decomposed into two separate passes. The first pass breaks the input string up into so-called tokens that represent the primitive units being parsed. In the case of tree parsing, the tokens are an open parenthesis "(", a close parenthesis ")", and any contiguous sequence of non-whitespace characters, such as "*", "253", "foo", and "\$a-b_c!#?73".

For example, the string

```
(( * 54 * ) 255 ( * 3 * ))
```

consists of the following tokens:

```
"(", "(", "*", "54", "*", ")", "255", "(", "*", "3", "*", ")", ")"
```

```

fromString("") =
IntTree.fromString: expected ( but got )

fromString("17") =
IntTree.fromString: expected ( but got 17

fromString("(") =
IntTree.fromString: too few tokens!

fromString("* ") =
IntTree.fromString: too few tokens!

fromString("* 1") =
IntTree.fromString: too few tokens!

fromString("* 1 *") =
IntTree.fromString: too few tokens!

fromString("* 1 * *") =
IntTree.fromString: expected ) but got *

fromString("1 2 3") =
IntTree.fromString: expected ( but got 1

fromString("* 2 3") =
IntTree.fromString: expected ( but got 3

fromString("(* 1 *) 2 (3 4 5)") =
IntTree.fromString: expected ( but got 3

fromString("(* 1 *) 2 (* 4 5)") =
IntTree.fromString: expected ( but got 5

fromString("(* 1 *) 2 (* 4 *)") =
IntTree.fromString: too few tokens!

fromString("(* 1 *) 2 (* 3 *) 4)") =
IntTree.fromString: expected ) but got 4

fromString("(* 1 *) 2 (* 3 *) 4") =
IntTree.fromString: superfluous token 4

fromString("* 1 *) 2 (* 3 *)") =
IntTree.fromString: superfluous token 2

fromString("* foo *)") =
IntTree.fromString: expected an integer but got foo

fromString("(* bar *) 2 (* 3 *)") =
IntTree.fromString: expected an integer but got bar

fromString("(* 1 *) baz (* 3 *)") =
IntTree.fromString: expected an integer but got baz

fromString("(* 1 *) 2 (* quux *)") =
IntTree.fromString: expected an integer but got quux

```

Figure 1: Examples of tree parsing errors encountered by `fromString`.

Whitespace is ignore in the tokenizing process, so "(* 1 *)" has exactly the same tokens as "(* 1 *)".

The second pass builds a tree from the token sequence according to rules for tree formation. In the case of `IntTree`, the rules are:

- a leaf is represented by "*"
- a node is represented by "(*left-subtree-tokens* *integer-token* *right-subtree-tokens*)".

The first pass of splitting the string into tokens is tricky, so this process has already been encapsulated for you in the `StringTreeTokens` class. The class method invocation

```
new StringTreeTokens(string)
```

creates an enumeration whose elements are the tokens in *string* for the tree parsing process.

The second pass is captured by a `fromTokens` method that takes a stream of tokens (represented as an instance of `StringTreeTokens` and consumes all the tokens that it needs to to form a tree. It leaves behind any tokens not consumed by this process. In particular, when parsing an integer tree:

- If `fromTokens` encounters a "*" token, it should return a leaf.
- Otherwise, `fromTokens` “expects” that the tree should begin with a "(" token. If not, it throws an exception. If so, it should consume all the tokens up to and including the matching ")" token, and then create an appropriate node. Of course, `fromTokens` will need to be invoked recursively to parse the left and right hand subtrees of this node.

Here is the definition of `fromString` based on this approach:

```
public static IntTree fromString (String s) {
    // Parses a string into an integer tree.
    // Either an IntTree, or throws an exception.
    Enumeration tokens = new StringTreeTokens(s);
    IntTree t = fromTokens(tokens);
    if (tokens.hasMoreElements()) {
        throw new TreeException("IntTree.fromString: superfluous token "
                                + (tokens.nextElement()));
    } else {
        return t;
    }
}
```

Your Task

Your problem is to flesh out the `fromTokens` method:

```
public static IntTree fromTokens (Enumeration tokens);
Parses the given enumeration of tokens into an integer tree. Either returns an IntTree, or
throws an exception indicating the tokens were not parsable into an IntTree.
```

So that you don't have to throw any exceptions directly, the following methods have been provided to abstract over the exception-throwing parts of the code:

```

public static void check (String expected, String actual) {
    if (! expected.equals(actual)) {
        throw new TreeException("IntTree.fromString: expected " + expected
            + " but got " + actual);
    }
}

public static int parseInt (String s) {
    try {
        return Integer.parseInt(s);
    } catch (NumberFormatException e) {
        throw new TreeException("IntTree.fromString: expected an integer but got " + s);
    }
}

public static String nextToken (Enumeration tokens) {
    try {
        return (String) tokens.nextElement();
    } catch (RuntimeException e) { // no more tokens
        throw new TreeException("IntTree.fromString: too few tokens!");
    }
}

```

If you use the above auxiliary methods, then you should not have to explicitly throw any exception in your definition of `fromString`.

Notes

- The skeleton for `fromTokens` can be found in the file `TreeParser.java`, which you should flesh out for this problem.
- The amount of code you have to write is rather small – `fromTokens` can be expressed in under 15 lines. However, you need to think carefully. The most important part of the solution strategy is *wishful thinking* – believing that when `fromTokens` is called recursively, it will correctly return the appropriate subtree after consuming all the tokens for that subtree.
- You can test your code by invoking:

```
java TreeParser tree-strings.txt
```

This invokes `fromString` on each of the lines in the test file `tree-strings.txt`. Feel free to add additional test cases to `tree-strings.txt`.

Problem 3 [20]: Destructive Reverse

Background

Recall that there are many ways to return a new list that is the reverse of a given list. For instance, here is a method encapsulating the classical divide/conquer/glue strategy for this problem:

```
public static IntList reverseRec (IntList L) {
    if (IL.isEmpty(L)) {
        return IL.empty();
    } else {
        return IL.postpend(reverseRec(IL.tail(L), IL.head(L)));
    }
}
```

Here is a more efficient iterative solution:

```
public static IntList reverseIter (IntList L) {
    IntList result = IL.empty();
    while (! IL.isEmpty(L)) {
        result = IL.prepend(IL.head(L), result);
        L = IL.tail(L);
    }
    return result;
}
```

Both of the above methods are *non-destructive* in the sense that they return a new list rather than modifying the given list. Indeed, all methods that operate on objects of type `IntList` are necessarily non-destructive because `IntList` is a class denoting *immutable* integer lists – there are no operations for changing the head or tail of a given list.

In class, we have seen another class – `IntMList` – that denotes *mutable* integer lists. The `IntMList` class supports all of the static methods of `IntList` but additionally has the following two methods:

```
public static int setHead (IntMList L, int newHead);
```

Changes the head of the list node `L` to be the integer `newHead`. Returns the integer previous stored in the head of `L`.

```
public static IntList setTail (IntMList L, IntMList newTail);
```

Changes the tail of the list node `L` to be the mutable integer list `newTail`. Returns the mutable integer list previous stored in the tail of `L`.

Your Task

In this problem, you are to implement the following *destructive* version of list reversal:

```
public static IntMList reverseD (IntMList L);
```

Destructively rearranges the list nodes of `L` so that they are in reverse order. If `L` is non-empty, returns the first node of the reversed list (which is the last node of the list in its original order); otherwise returns the empty list.

To complete this problem you should (1) flesh out the skeleton for `reverseD` in the class `DestructiveReverse` and (2) implement and execute testing code that shows that your `reverseD` method works as expected.

Notes

- `DestructiveReverse.java` has been configured so that the `IntegerMList` operations `empty`, `isEmpty`, `prepend`, `head`, `tail`, `setHead`, `setTail`, `toString`, and `fromString` can all be invoked with a `IML` prefix.
- Your `reverseD` method should only rearrange pointers in the the given list, not create any new list nodes. Thus, your solution should not contain any calls to `prepend`.
- The testing code in this problem is just as important as, if not more important than, your code for `reverseD`. Think carefully about how to convince someone else that your `reverseD` method works as specified, and implement test cases that are “convincing” in this way.

*Problem Set Header Page
Please make this the first page of your hardcopy submission.*

CS230 Problem Set 5

Due Friday, October 25

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

By signing below, I attest that I have followed the collaboration policy as specified in the Course Information handout.

Signature:

*In the **Time** column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

| Part | Time | Score |
|-----------------|-------------|--------------|
| General Reading | | |
| Problem 1 [50] | | |
| Problem 2 [30] | | |
| Problem 3 [20] | | |
| Total | | |