Programming Assignment 4

Due: March 12, 2018

0 Instructions

Submit your work through Canvas. You should submit a tar file containing all source files and a README for running your project. Don't submit any other files (e.g., test case or pyc files).

More precisely, submit on Canvas a tar file named lastname.tar (where lastname is your last name) that contains:

- All source files. You can choose any language that builds and runs on ix-dev.
- A file named README that contains your name and the exact commands for building and running your project on ix-dev. If the commands you provide don't work on ix-dev, then your project can't be graded and there will be a significant penalty.

Here is an example of what to submit:

```
hampton.tar
  README
  my_class1.py
  my_class2.py
  my_class3.py
  problem.py
  another_problem.py
  ...
```

```
README
  Andrew Hampton

  Problem 1: python problem1.py <input_filename>
  Problem 1: python problem1.py <input_filename>
  ...
```

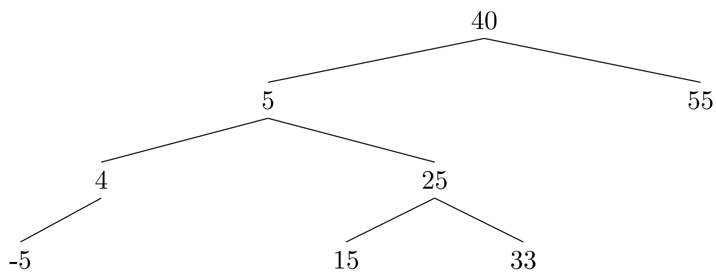
Note that Canvas might change the name of the file that you submit to something like lastname-N.tar. This is totally fine!

The grading for the project will be roughly as follows:

Task	Points
Problem 1	**10**
pass given sample test case	3
pass small grading test case	3
pass large grading test case	4
Problem 2	**10**
pass given sample test case	3
pass small grading test case	3
pass large grading test case	4
Problem 3	**30**
pass given sample test case	10
pass small grading test case	10
pass large grading test case	10
TOTAL	**50**

1 Applications

Problem 1. Best Path
Add a method to your BST class that calculates the value of the *best path* from the root to a leaf node. We define the *best path* to be the path with the highest occurrence of the digit 5. Consider the tree:



This tree has a total of four paths from root to leaf:

40 – 5 – 4 – -5: this path has 2 occurrences of the digit 5
40 – 5 – 25 – 15: this path has 3 occurrences of the digit 5
40 – 5 – 25 – 33: this path has 2 occurrences of the digit 5
40 – 55: this path has 2 occurrences of the digit 5

Therefore, the path 40 – 5 – 25 – 15 is the *best path* and its value is 3. Your new method must calculate the value of the best path in the tree:

`best_path_value()`: Returns the value of the best path. $O(n)$

Write a driver program that takes a single command-line argument, which will be a filename. The input file will contain instructions for tree operations. The first line of the input file will be an integer $0 \leq N \leq 10^6$ giving the number of instructions. Following will be N lines, each containing an instruction. The possible instructions are:

`insert K`, where $-10^5 \leq K \leq 10^5$ is an integer: insert a node with key K into the tree. There is no output.

`remove K`, where $-10^5 \leq K \leq 10^5$ is an integer: remove a node with key K from the tree. If such a node exists, there is no output. If no such node exists, output *TreeError*.

`bpv`: output the best path value. If the tree is empty, output *TreeError*.

Hint: in Python, to count the number of occurences of the digit 5 in an integer x, try `str(x).count('5')`.

Example input file:

```
11
insert 40
insert 5
insert 4
insert -5
insert 25
insert 15
insert 33
insert 55
bpv
```

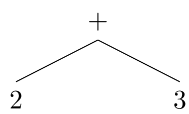
```
remove 15
bpv
```

Example output:

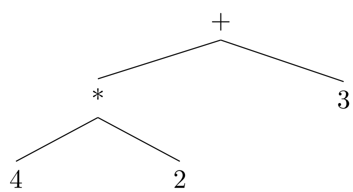
```
3
2
```

Problem 2. Syntax Tree
Extracting meaning from a sequence of symbols is a challenging task! Think about how difficult it is to learn a new language. Fortunately, formal languages like those found in programming and mathematics are designed with more structure than natural language. In this problem, we will build and analyze a syntax tree for basic arithmetic expressions.

Consider the arithmetic expression $2 + 3$. We can represent it as a *syntax tree* like this:



For simple expressions like this, the syntax tree doesn't help much. But consider the more complicated expression $4 * 2 + 3$. The correct order of operations is clear from the syntax tree:



To *evaluate* an arithmetic expression, we find the equivalent value using the usual mathematical order of operations. The expression $2 + 3$ evaluates to 5. The expression $4 * 2 + 3$ evaluates to 11.

We say that the expression is *fully parenthesized* if parentheses are used to make the order of operations unambiguous. The fully parenthesized version of $2+3$ is $(2+3)$. The fully parenthesized version of $4*2+3$ is $((4 * 2) + 3)$.

For this problem, you will evaluate a syntax tree representing an arithmetic expression. Your program should take a single command-line argument, which will be a filename. The input file will contain exactly two lines. The first line of the input file will be an integer $0 \leq N \leq 10^5$ giving the number of nodes in the syntax tree. The second line will be a space-separated array representation of the syntax tree (using the standard representation from the textbook, where the array is one-indexed and the element at position i represents the node with children at positions $2i$ and $2i + 1$).

The syntax tree can contain integers $-10^3 \leq X \leq 10^3$, as well as the symbols $+$, $-$, and $*$.

You need to output two things. First, a fully parenthesized mathematical expression. Second, a single integer, the result of evaluating the syntax tree. This output should be separated by a newline. See sample output below.

The input tree will always represent valid mathematical syntax.

The runtime should be linear in the number of nodes.

Hint: read the input and construct a binary tree. To evaluate the expression, perform a post-order traversal. To create the fully parenthesized expression, consider an in-order traversal.

Getting started: code (*problem2_starter.py*) is provided for building a syntax tree from the input list. Read this code to understand what it's doing (a preorder traversal) and incorporate it into your solution if you want. This is a common use case for the preorder traversal.

Example input 1:

```
5
+ * 3 4 2
```

Example output 1:

```
((4*2)+3)
11
```

This input file represents the syntax tree shown above.

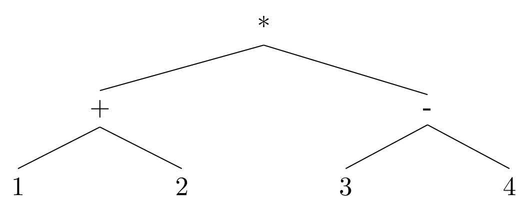
Example input 2:

```
7
* + - 1 2 3 4
```

Example output 2:

```
((1+2)*(3-4))
-3
```

This input file represents the following syntax tree:



2 Implementation

Problem 3. Red-Black Tree
For this problem, you will implement a red-black tree with integer keys. Do not use any builtin tree structures that your language might have. Since you have already implemented a binary search tree, you should augment that structure such that it satisfies the red-black properties as described in Chapter 13 of the textbook.

Your data structure should implement the same methods as the BST from Project 3. The only difference (but it is a big difference!) is that any method in the BST that had runtime complexity $O(h)$ must in the red-black tree be $O(\log n)$, where n is the number of nodes in the tree.

Write a driver program that takes a single command-line argument, which will be a filename. The input file will contain instructions for tree operations. The first line of the input file will be an integer $0 \leq N \leq 10^6$ giving the number of instructions. Following will be N lines, each containing an instruction. The possible instructions are:

`insert K`, where $-10^5 \leq K \leq 10^5$ is an integer: insert a node with key K into the tree. There is no output.

`remove K`, where $-10^5 \leq K \leq 10^5$ is an integer: remove a node with key K from the tree. If such a node exists, there is no output. If no such node exists, output *TreeError*.

`search K`, where $-10^5 \leq K \leq 10^5$ is an integer: output *Found* if a node exists with key K. If no such node exists, output *NotFound*.

`max`: output the maximum key in the tree. If the tree is empty, output *Empty*.

`min`: output the minimum key in the tree. If the tree is empty, output *Empty*.

`inprint`: print the keys of the tree according to an in-order traversal, separated by a single space. If the tree is empty, output *Empty*.

Example input file:

```
18
inprint
remove 2
max
search 5
insert 1
insert 2
search 1
search 2
insert 3
inprint
insert 10
insert 5
inprint
search 5
remove 2
inprint
max
min
```

Example output:

```
Empty
TreeError
Empty
NotFound
Found
Found
1 2 3
1 2 3 5 10
Found
1 3 5 10
10
1
```