6 Week 6: Forest for the Trees

Code snippet for generating the random list:

```
from random import randint, seed
# Generate random list (Seed specified for repeatability)
seed(3.14)
l = []
for ri in range(15):
    l.append(randint(0,60))

print("Random List: ", l)
```

Output:

```
Random List:  [31, 48, 3, 23, 17, 1, 48, 47, 41, 9, 54, 29, 8, 24, 30]
```

6.1 Heap generated by adding each element one by one:

Code snippet, exluding the class definition for BinHeap. I had extended the BinHeap class with the printList() and printTree() methods to make it easier to see the different representations.

```
# Problem #1
print("1. Inserted item by item:")
heap = BinHeap()
for item in l:
    heap.insert(item)
print("Heap List:    ", end='')
heap.printList()
print("Tree View:")
heap.printTree()
```

Output:

```
1. Inserted item by item:
Heap List:    [1, 9, 3, 41, 17, 8, 24, 48, 47, 23, 54, 31, 29, 48, 30]
Tree View:
|                       1                       |
|           9           |           3           |
|     41    |     17    |     8     |     24    |
|  48 |  47 |  23 |  54 |  31 |  29 |  48 |  30 |
```

6.2 Heap generated by passing the list to the buildHeap() method:

Code snippet, exluding the class definition for BinHeap.

```
# Problem #2
print("2. Built from list:")
heap.buildHeap(l)
print("Heap List:    ", end='')
heap.printList()
print("Tree View:")
heap.printTree()
```

Output:

```
2. Built from list:
Heap List:    [1, 9, 3, 23, 17, 8, 24, 47, 41, 48, 54, 29, 31, 48, 30]
Tree View:
|                      1                       |
|           9          |           3           |
|     23    |     17   |     8     |     24    |
|  47 |  41 |  48 |  54 |  29 |  31 |  48 |  30 |
```

Note that while still meeting the requirements in our definition of a binary heap, the items in the list are arranged differently from when they were added one-by-one.

6.3 [Bonus] Extending buildHeapTree() to handle expressions without spaces:

Code snippet of the addition to buildHeapTree() function to accomodate operators without spaces:

```
    # Adds spaces around operators and parentheses [Problem #3]
    for o in operators + parentheses:
        fpexp = fpexp.replace(o, ' '+o+' ')

    fplist = fpexp.split()
```

Elsewhere I had refactored the code to save the list of operators instead of repeatedly creating lists of string literals.

Example usage:

```
Example 2 (no spaces):
Original Expression:   ((10+5)*3)
Expression from Tree:  ( ( 10 + 5 ) * 3 )
```

6.4 [Bonus] Extending buildHeapTree() to handle Boolean Statements:

A few changes were necessary for this, first adding boolean operators to the list. I chose to include both the symbolic symbols and keywords.

```
    binary_operators = ['+', '-', '*', '/', '&', '|', "AND", "OR"]
    unary_operators = ['~', "NOT"]
    parentheses = ['(',')']
    operators = binary_operators + unary_operators
```

The unary operators are stored separately because they will require further evaluation when building the tree, as shown here in the section for parsing the closing parenthesis:

```
elif i == ')':
    currentTree = pStack.pop()

    # Climb back up stack without ) through unary operators [Problem #4]
    while currentTree.getRootVal() in unary_operators:
        currentTree = pStack.pop()
```

The same process must also be used after handling operands to the tree, as I chose to allow the NOT operator to be used without parentheses. Additionally, to allow the keywords "True" and "False" to be used in the expression I added some logic to catch those before parsing the string as an integer:

```
else:
        try:
            # Additional code for parsing logical keywords [Problem #4]
            if i.capitalize() == 'True':
                currentTree.setRootVal(True)
            elif i.capitalize() == 'False':
                currentTree.setRootVal(False)
            else: # Not True/False, so try parsing as int
                currentTree.setRootVal(int(i))
```

```
            parent = pStack.pop()
            currentTree = parent

            # Climb back up stack without ) through unary operators [Problem #4]
            while currentTree.getRootVal() in unary_operators:
                currentTree = pStack.pop()
```

Example Usage:

```
Example 3 — Logical Symbols:
Original Expression:   ( ( 1 & 0 ) | ~1 )
Expression from Tree:  ( ( 1 & 0 ) | ~1 )
Post—Order Tree Traversal:
1
0
&
1
~
|

Example 4 — Logical Keywords:
Original Expression:   ( ( 1 AND 0 ) OR NOT 1 )
Expression from Tree:  ( ( 1 AND 0 ) OR NOT 1 )

Example 5 — Logical Keywords II:
Original Expression:   ( ( False AND True ) OR NOT ( NOT False OR NOT True ) )
Expression from Tree:  ( ( False AND True ) OR NOT ( NOT False OR NOT True ) )
```