ANSWERS

A4.7 Tree traversals

Trees and graphs · A level · OCR H446 2.3.1, AQA 7517 4.3.2.1, Eduqas A500QS 1.1 · about 25 min

BugBotLab

What this lesson is about

Pre-order, in-order and post-order, the outline method, expression trees and what each traversal is for.

- Three orders
- The outline method
- What each traversal is for
- Expression trees
- Breadth-first on a tree

Questions

6 marks in all

1. Which traversal outputs the keys of a binary search tree in ascending order? [1 mark]

- ☐ **A** In-order
- ☐ **B** Pre-order
- ☐ **C** Post-order
- ☐ **D** Breadth-first

> **Answer:** A. Left subtree (smaller), then the node, then the right subtree (larger), at every node.

2. Which traversal of an expression tree produces Reverse Polish notation? [1 mark]

- ☐ **A** Post-order
- ☐ **B** Pre-order
- ☐ **C** In-order
- ☐ **D** Breadth-first

> **Answer:** A. Both operands are written before their operator, which is postfix.

3. Put the items of the expression tree for (2 + 3) * 4 in post-order. [1 mark]

Number the lines 1 to 5 to put them in the right order.

___ `*`
___ `4`
___ `2`
___ `+`
___ `3`

> **Answer:**
>
> ```
> 2
> 3
> +
> 4
> *
> ```
>
> Post-order visits each operator after both of its subtrees: 2 3 + 4 *.

4. Which traversal is used to copy a tree, so that inserting the keys into a new tree rebuilds the same shape? [1 mark]

- ☐ **A** Pre-order
- ☐ **B** In-order
- ☐ **C** Post-order
- ☐ **D** Any of them gives the same shape

> **Answer:** A. Pre-order inserts each node before its children, so every parent is in place first. In-order inserts sorted keys and builds a degenerate tree.

5. What does this program print? [1 mark]

```
class Node:
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

def walk(node):
    if node is None:
        return ""
    return node.item + walk(node.left) + walk(node.right)

tree = Node("M", Node("F", Node("B"), Node("H")), Node("T", None, Node("W")))
print(walk(tree))
```

> **Answer:**
>
> ```
> MFBHTW
> ```
>
> The node comes before both subtrees, so this is a pre-order traversal.

6. A traversal is written: traverse left, traverse right, then output the node. Which traversal is it, and what is it used for? [1 mark]

- ☐ **A** Post-order, used to empty a tree
- ☐ **B** Pre-order, used to copy a tree
- ☐ **C** In-order, used to output a binary search tree in order
- ☐ **D** Breadth-first, used to find shortest paths

> **Answer:** A. The output comes after both recursive calls. Each node is dealt with after its children, so a tree can be deleted from the leaves up.

The task: three ways round a tree

`TREE` holds the expression (5 - 1) * (2 + 3 * 4) as a binary tree of `Node` objects. Each node's `item` is an operator (`+`, `-` or `*`) or a single digit, both as strings, and a leaf has `left` and `right` set to `None`. Write recursive functions `preorder(node)`, `inorder(node)` and `postorder(node)` that each return a list of the items in that order, and an empty list for `None`. Print `pre-order: `, `in-order: ` and `post-order: `, each followed by that list's items separated by single spaces, one per line in that order. Then work out the value of the expression from the tree, without `eval`, and print `value: ` followed by the whole number.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

class Node:
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

# (5 - 1) * (2 + 3 * 4)
TREE = Node("*", Node("-", Node("5"), Node("1")), Node("+", Node("2"), Node("*", Node("3"),
Node("4"))))

def preorder(node):
    return []
```

The hint students can ask for: Each traversal visits the left subtree, the right subtree and the node itself; only the moment the node is written down changes. To work out the value, each operator node needs the values of both its subtrees first, which is the post-order pattern.

A solution

```
from bugbot import *
connect()

class Node:
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

# (5 - 1) * (2 + 3 * 4)
TREE = Node("*", Node("-", Node("5"), Node("1")), Node("+", Node("2"), Node("*", Node("3"),
Node("4"))))

def preorder(node):
    if node is None:
        return []
    return [node.item] + preorder(node.left) + preorder(node.right)

def inorder(node):
    if node is None:
        return []
    return inorder(node.left) + [node.item] + inorder(node.right)

def postorder(node):
    if node is None:
```

```
        return []
    return postorder(node.left) + postorder(node.right) + [node.item]

def value(node):
    if node.left is None:
        return int(node.item)
    a = value(node.left)
    b = value(node.right)
    if node.item == "+":
        return a + b
    if node.item == "-":
        return a - b
    return a * b

print("pre-order:", " ".join(preorder(TREE)))
print("in-order:", " ".join(inorder(TREE)))
print("post-order:", " ".join(postorder(TREE)))
print("value:", value(TREE))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.