CMPT 225: Data Structures & Programming
– Unit 14 –
Traversals & Binary Trees

Dr. Jack Thomas

Simon Fraser University

Spring 2021

Today's Topics

- Traversals Continued
- Binary Trees
- The Binary Tree ADT
- Binary Trees in Java
- Inorder Traversals
- Euler Tour Traversals

More on Traversals

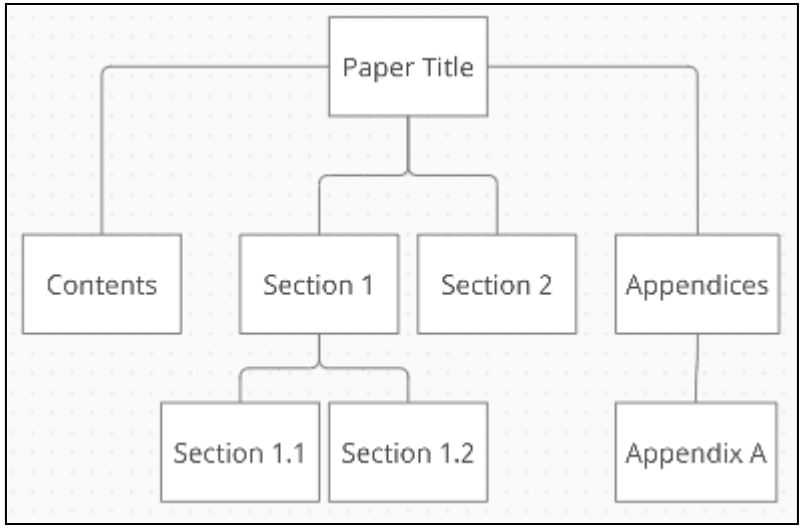
- There are several **types of traversals** that can serve as the basis for our solutions, depending on the problem.
- Traversal algorithms are typically recursive, involving a step that **accesses** the current node's element and steps that **recursively call itself** on the node's parents or children, or **returning**.
- The distinguishing feature of each type is when they **access** (or **visit**) each node's element, instead of first visiting a parent or child of that node.
- Algorithms typically allow **only one visit per node**.

Preorder Traversal

- In preorder traversals, the visit happens first, then the algorithm moves on to that node's children.

```
Algorithm preorder(T,v):
    Input: A tree T and node v.
    Output: The result of each visit, starting with
            v and moving to v's children.
    perform the "visit" action for node v
    for each child w of v in T do
            preorder(T,w)
```

Example Preorder: Printing a Tree



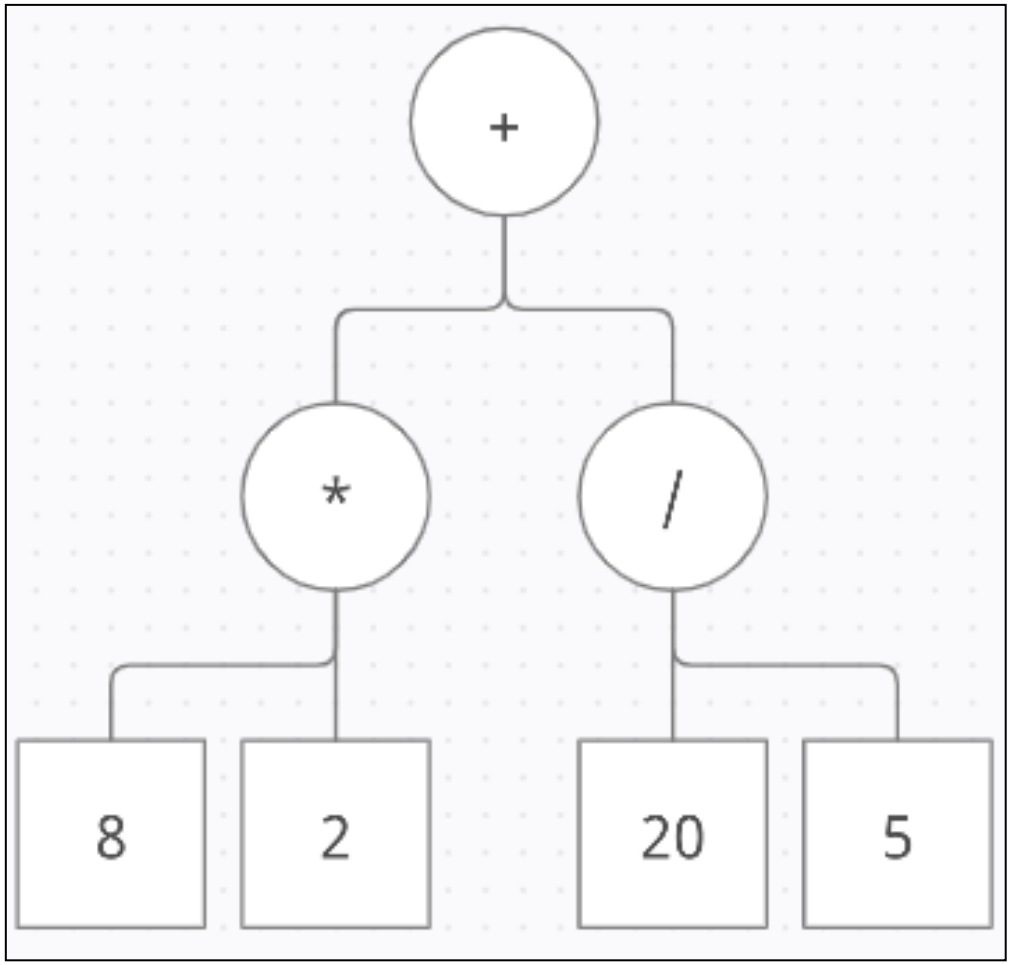
- A recursive, preorder solution would print:
  - Paper Title
  - Contents
  - Section 1
  - Section 1.1
  - Section 1.2
  - Section 2
  - Appendices
  - Appendix A

Postorder Traversal

- Postorder traversals start by visiting the children of the node, then visit the parent.

```
Algorithm postorder(T,v):
    Input: A tree T and node v.
    Output: The result of each visit, starting with
            v's children before v.
    for each child w of v in T do
        postorder(T,w)
    perform the "visit" action for node v
```

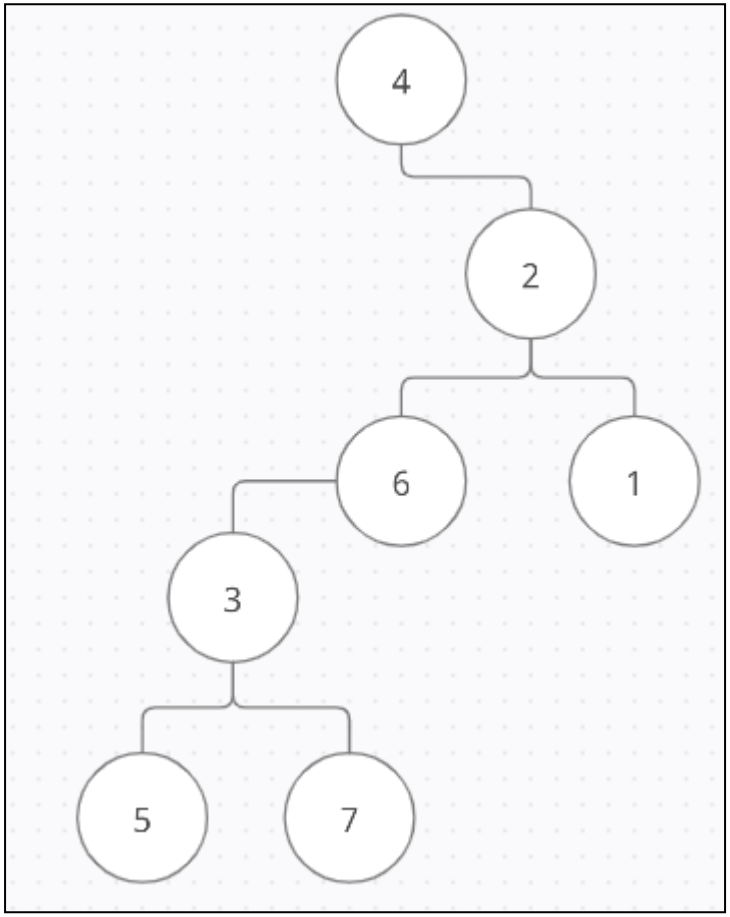
Postorder Example: Resolving an Arithmetic Expression



- A postorder recursive solution would calculate (8*2) + (20/5)
- This depends on the order of operations being correctly represented by the tree, internal nodes being operators, and external nodes being numbers.

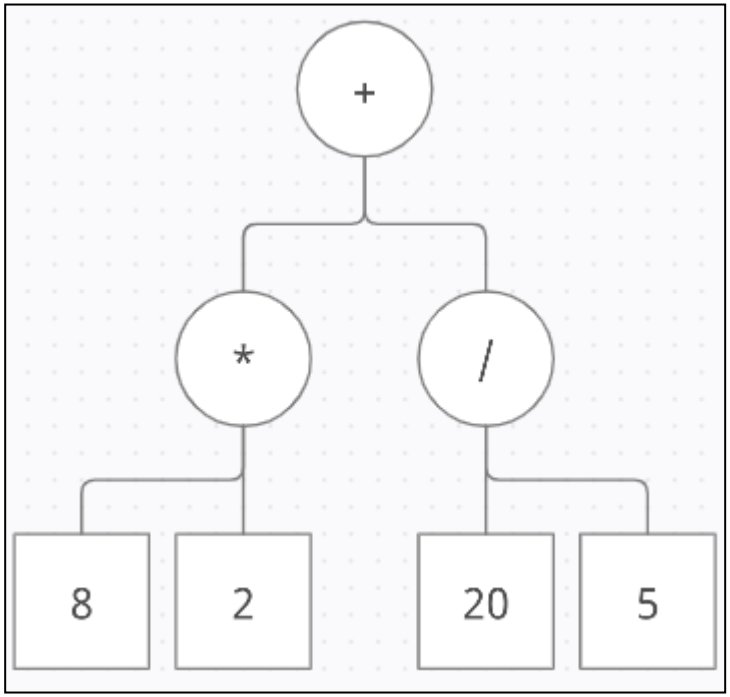
The Binary Tree

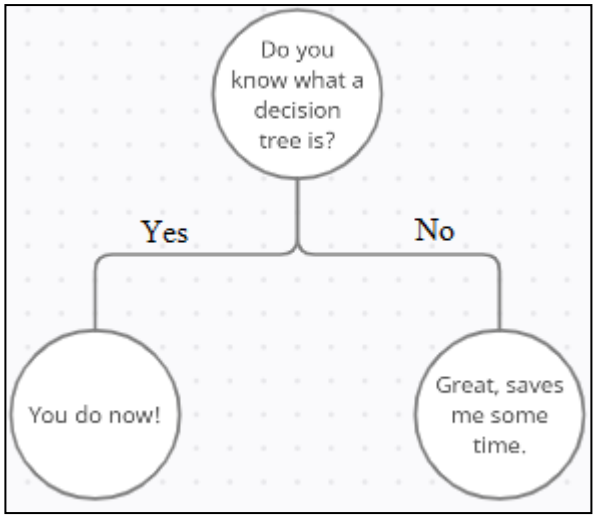
- A **Binary Tree** is a tree with these additional constraints:
  - No node has more than two children.
  - The children of each node are ordered (one is first, or left, while the other is last, or right).
- In a **proper** (or **full**) binary tree, nodes have either zero or two children.



Applications of a Binary Tree

- Arithmetic expressions.
  - Ex: 8 * 2 + 20 /5
- Decision trees





- Sorting and Searching

Properties of a Binary Tree

- The binary constraint influences the relationship between the **number (n) of nodes** in the tree, the **height (h)** of the tree, and the number **of internal (i) and external (e) nodes**.
  1. The number of nodes (**n**) will be greater than or equal to the height **(h) + 1**, but less than or equal to $\mathbf{2^{h+1} - 1}$
  2. There must always be at least one external node **e**, but there can't be more than $\mathbf{2^h}$.
  3. There must at least be as many internal nodes **i** as the height **(h)** of the tree, and no more than $\mathbf{2^h - 1}$.
  4. The height **(h)** can be between $\mathbf{\log_2(n+1)}$ and **n – 1**.
- Remember these when coming up with algorithmic solutions to problems involving Binary Trees.

Binary Tree ADT

- A **subtype of the Tree** data structure which limits nodes to a maximum of two ordered children.
- Includes all of the methods and properties of the general Tree.
- Binary Trees include the following methods:
  - **Left**: Returns the left child.
  - **Right**: Returns the right child.
  - **hasLeft**: Confirms whether there's a left child.
  - **hasRight**: Confirms whether there's a right child.

The Binary Tree in Java

- Once again, there is **no built-in Binary Tree class or interface** in Java.
- We can take the **general tree** from our previous unit and **modify it** into a Binary Tree.
- “Isn’t this a job for inheritance”? Well, it could be, depending on how we defined our Tree and Node classes.
- To make our implementations easier to follow, however, we’ll **simply rewrite** the parts we need directly to make one pure Binary Tree class.

Implementing a Binary Tree

- A ton of the methods for a Binary Tree will be either **the same or require only a trivial change** from the general tree we implemented before, including:
  - Size, isEmpty
  - isRoot, isInternal, isExternal, hasLeft, hasRight
  - Left, right, root, sibling, parent, children
- Attaching a new node, removing an old one, and collecting all the nodes (without keeping a supplementary list) is more involved.

Implementing a Binary Tree

```
class BTreeNode{
    protected String element;
    protected BTreeNode parent;
    protected BTreeNode left;
    protected BTreeNode right;

    public BTreeNode()
    {
        element = null;
        parent = null;
        left = null;
        right = null;
    }
    public BTreeNode(String input)
    {
        element = input;
        parent = null;
        left = null;
        right = null;
    }
    public String getElement() { return element; }
}
```

Implementing a Binary Tree

```
public void remove(BTreeNode node)
{
    BTreeNode leftNode = node.left;
    BTreeNode rightNode = node.right;
    if (leftNode != null && rightNode != null)
    {
        throw new RuntimeException
                ("Can't remove nodes with two children");
    }
    BTreeNode onlyChild = null;
    if(rightNode != null) {
        onlyChild = rightNode;
    }
    else if(leftNode != null)
    {
        onlyChild = leftNode;
    }
```

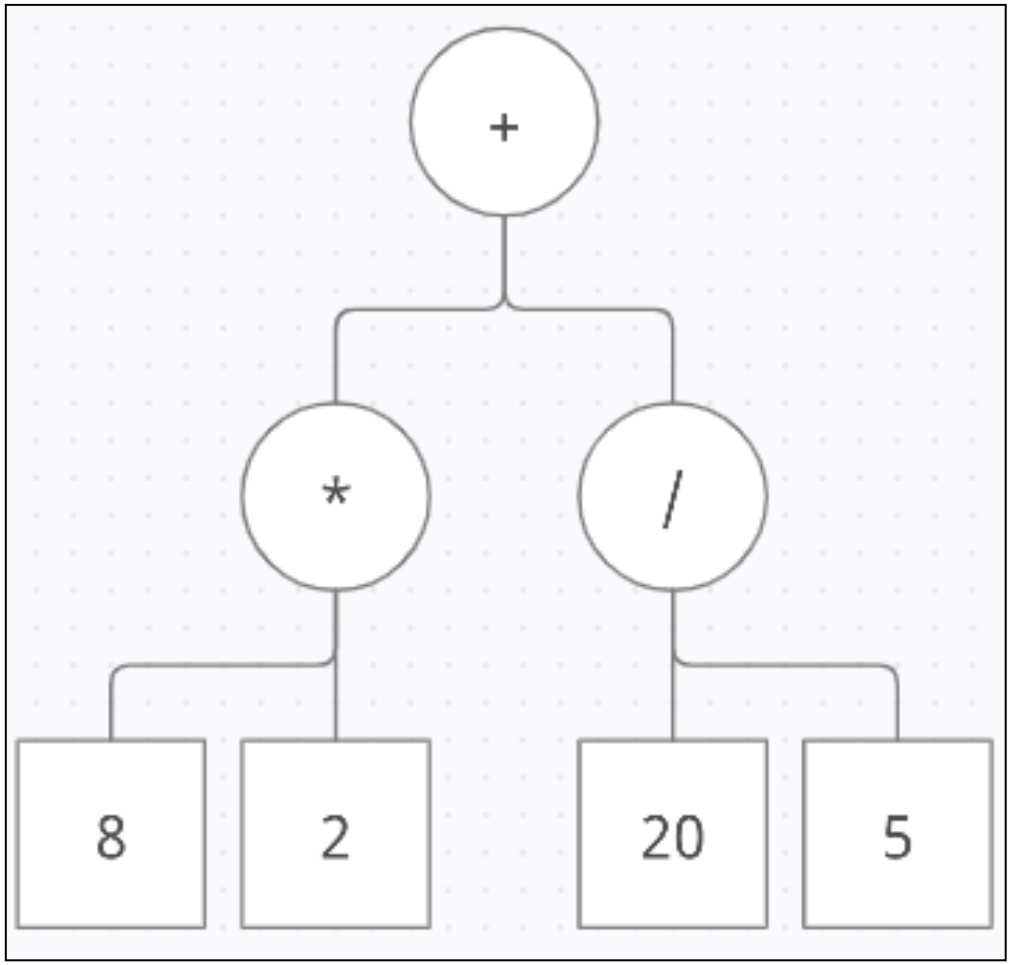
```
if (node == root)
{
    if (onlyChild != null)
    {
        onlyChild.parent = null;
    }
    root = onlyChild;
}
else {
    BTreeNode parent = node.parent;
    if(node == parent.left)
    {
        parent.left = onlyChild;
    }
    else
    {
        parent.right = onlyChild;
    }
    if(onlyChild != null)
    {

    }
    size--;
}
```

Inorder Traversals

- A node is visited after its left subtree but before its right subtree.

```
Algorithm inorder(T,v):
    Input: A tree T and node v.
    Output: The result of each visit, starting with
            v's left child before v and followed by v's right
            child.
    for the left child w of v in T do
            inorder(T,w)
    perform the "visit" action for node v
    for the right child z of v in T do
            inorder(T,z)
```

Inorder Example: Printing an Arithmetic Expression



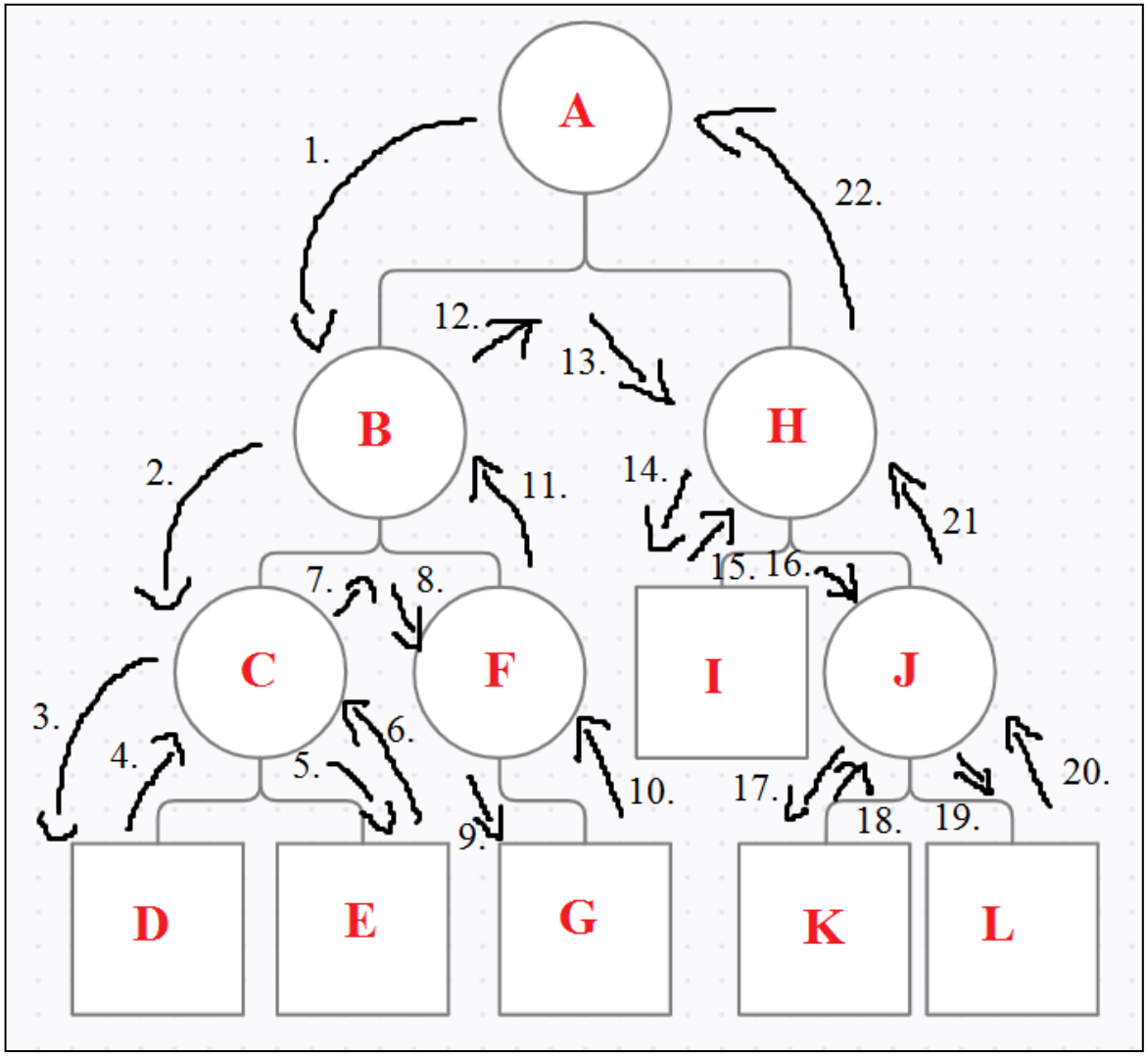
- An inorder recursive solution would print:
  - 8 * 2 + 20 / 5

Euler Tour Traversal

- By lifting the constraint that a node can only be "visited" once, we can perform a full tour around every node of a subtree.

```
Algorithm eulerTour(T,v):
    Input: A tree T and node v.
    Output: The result of each visit, starting with v's children
            before v.
    Perform the action for visiting node v on the left.
    If v has a left child u in T then
        eulerTour(T, u)
    Perform the action for visiting node v from below.
    if v has a right child w in T then
        euler tour(T, w)
    perform the action for visiting node v on the right
```

Euler Tour Traversal



1. A toLeft B
2. B toLeft C
3. C toLeft D
4. D toParent C
5. C toRight E
6. E toParentC
7. C toParentB
8. B toRight F
9. F toRight G
10. G toParent F
11. F toParent B
12. B toParent A
13. A toRight H
14. H toLeft I
15. I toParent H
16. H toRight J
17. J toLeft K
18. K toParent J
19. J toLeft L
20. L toParent J
21. J toParent H
22. H toParent A
23. A isRoot

Inorder, postorder, and preorder traversals are all sub-types of Euler Tours

Recap – Traversed to the Leaf Slide

- Algorithms for interacting with data in a tree are organized into different types of **traversals**.
- **Binary trees** are a structurally-constrained version of trees with a maximum of two ordered children per node.
- There is no default binary tree class in Java, though it's a **simple modification from a general tree**.
- They also enable the **inorder traversal**, which visits the left child before the node itself, and then the right child afterward.
- The **Euler Tour traversal** allows us to travel to and visit every node in the tree.