# Data Structures and Algorithms

## XMUT-COMP 103 - 2026 T1

## Tree Nodes

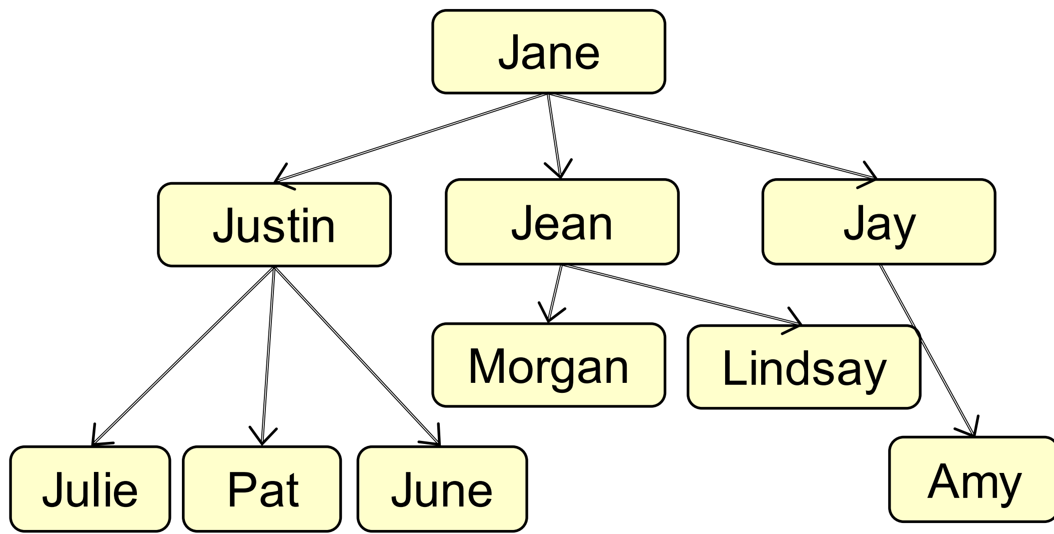
Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

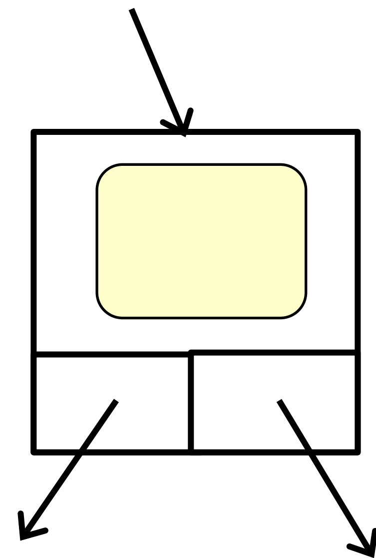
# Trees of Data *vs* Trees containing Data

- Person, Position,…
  - The data object contains the links that make the tree.
  - Person: father, mother, children,…
  - Employee: manager, team




- Typical Collection (Set, Map, Queue…)
  - The collection has the structure,
  - The data objects sit inside the structure

- Tree data structures:
  - Node object that has fields for the data item, and for the links.



# Tree Node types

- There is no single way of defining tree structures
  - Need to make your own.

eg: Binary tree node:

Type variable: parameter of the ***type***
specify when you make a new node:

```
public class BTNode <E> {
    private E item;
    private BTNode<E>  left;
    private BTNode<E>  right;

    public BTNode(E item){  this.item = item;  }

    public E getItem()                {  return item; }
    public void setItem(E item)   { this.item = item; }

    public BTNode<E> getLeft()              { return left;}
    public void setLeft(BTNode<E> left)     { this.left = left;}

    public BTNode<E> getRight()             { return right;}
    public void setRight(BTNode<E> right)  { this.right = right;}
}
```

# Using BTNode: Expressions

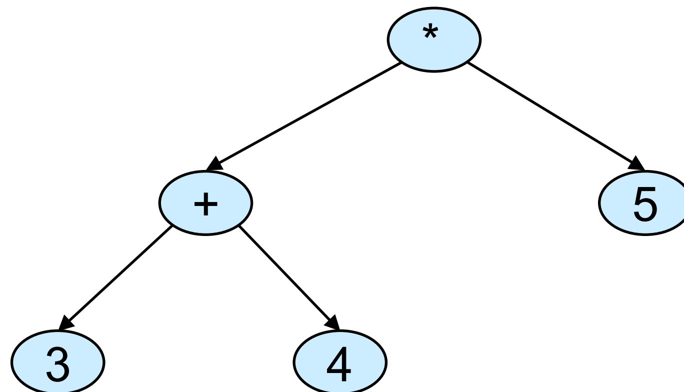
**Expression Trees**
An **expression tree** is a special type of **binary tree** used to represent mathematical expressions.

**Main Idea**
In an expression tree:
•**Operators** are stored in parent nodes
•**numbers/variables** are stored in leaf nodes

| Node | Meaning |
|---|---|
| × | root operator |
| + | sub-expression |
| 3,4,5 | operands (leaf nodes) |

Example

$(3 + 4) \times 5$



# Using BTNode: Expressions

**Expression Trees**

An **expression tree** is a special type of **binary tree** used to represent mathematical expressions.

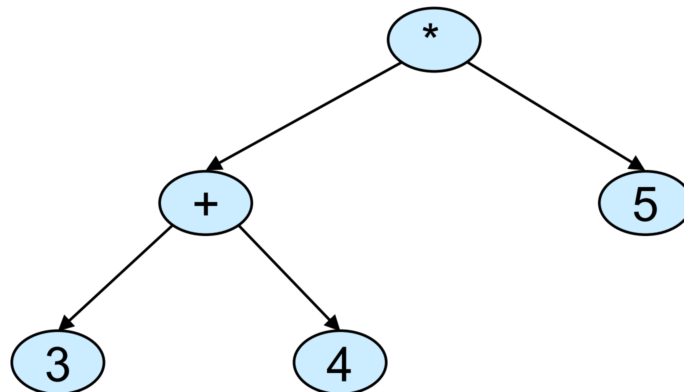
**Why Use Expression Trees?**

Expression trees help computers:

- evaluate expressions
- convert between infix/prefix/postfix
- build compilers
- parse mathematical formulas

Example

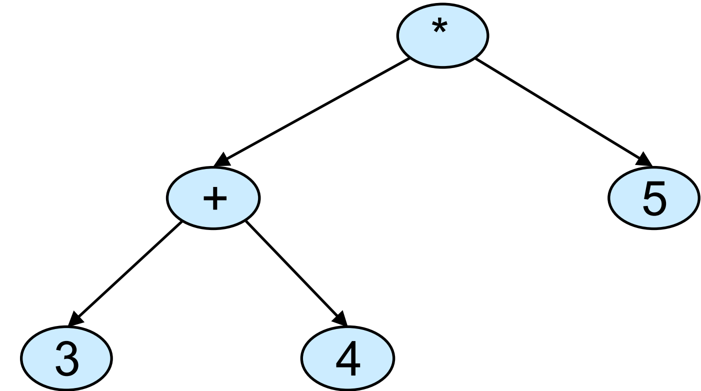
(3 + 4) × 5



# Using BTNode: Expressions

**Expression Types from Tree Traversals**
Different traversals produce different notations.

Tree:




**1. Inorder Traversal → Infix**

Rule:

`Left → Root → Right`

Result:

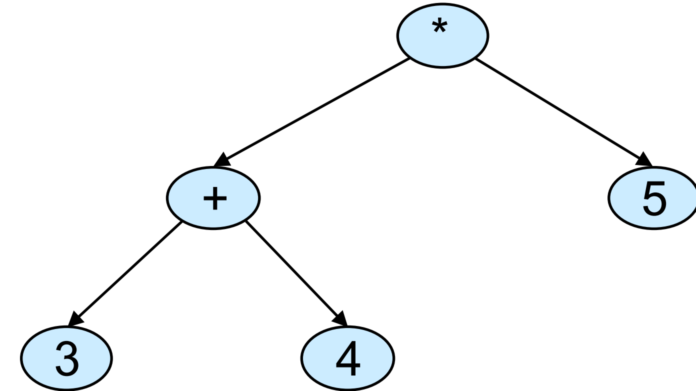
`(3 + 4) × 5`

This is **normal** math notation.

# Using BTNode: Expressions

**Expression Types from Tree Traversals**
Different traversals produce different notations.

Tree:




**2. Preorder Traversal → Prefix (Polish)**
Rule:
`Root → Left → Right`

Result:
`× + 3 4 5`

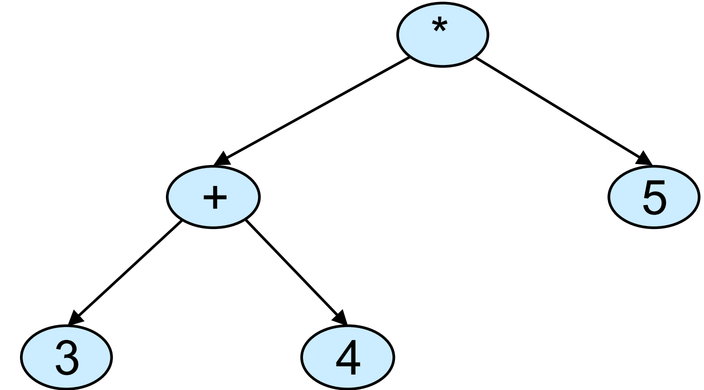
**3. Postorder Traversal → Postfix (Reverse Polish)**
Rule:
`Left → Right → Root`

Result:
`3 4 + 5 ×`

# Using BTNode: Expressions

**Expression Types from Tree Traversals**
Different traversals produce different notations.

Tree:




**2. Preorder Traversal → Prefix (Polish)**
Rule:
`Root → Left → Right`

Result:
`× + 3 4 5`

**3. Postorder Traversal → Postfix (Reverse Polish)**
Rule:
`Left → Right → Root`

Result:
`3 4 + 5 ×`

# Using BTNode: Expressions

**Preorder Traversal → Prefix (Polish Notation)**

Rule:

`Root → Left → Right`

Traversal:

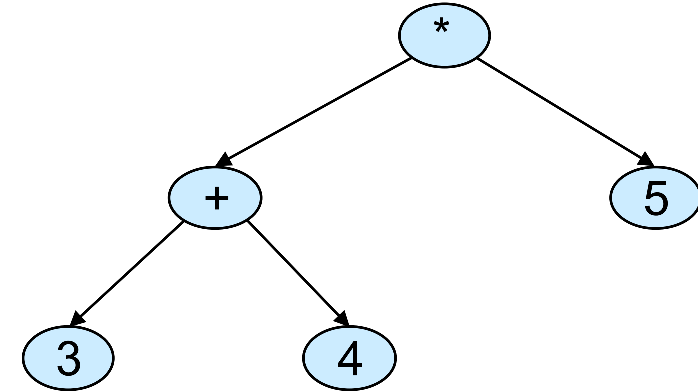
`× + 3 4 5`

This is called:

`Prefix notation`

because operators come BEFORE operands.

Tree:




**What Can We Do with Prefix?**

**A. Easy Parsing**

Computers can understand the structure immediately without parentheses.

Example:

`× + 3 4 5`

means:

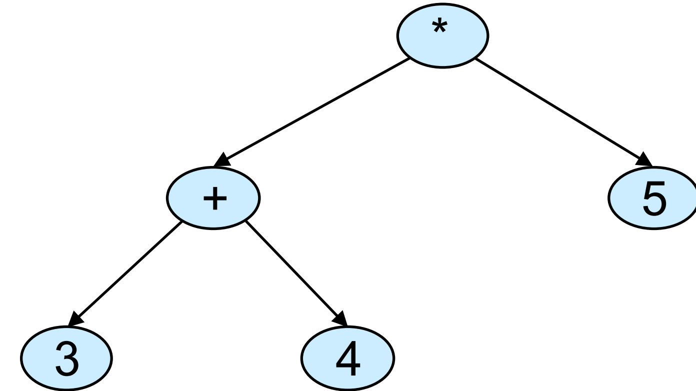
`(3 + 4) × 5`

No ambiguity.

# Using BTNode: Expressions

**Preorder Traversal → Prefix (Polish Notation)**

Tree:




**B. Used in Compilers**
Programming language compilers often convert expressions into tree or prefix-like structures internally.

**C. Recursive Evaluation**
Prefix works naturally with recursion.
Example:
`× (+ 3 4) 5`
Evaluate inside recursively.

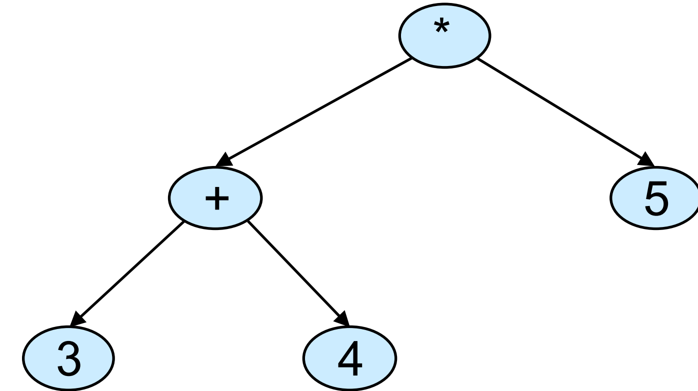
# Using BTNode: Expressions

**Postorder Traversal → Postfix (Reverse Polish)**

Rule:

`Left → Right → Root`

Tree:




Traversal:

`3 4 + 5 ×`

This is:

`Reverse Polish Notation (RPN)`

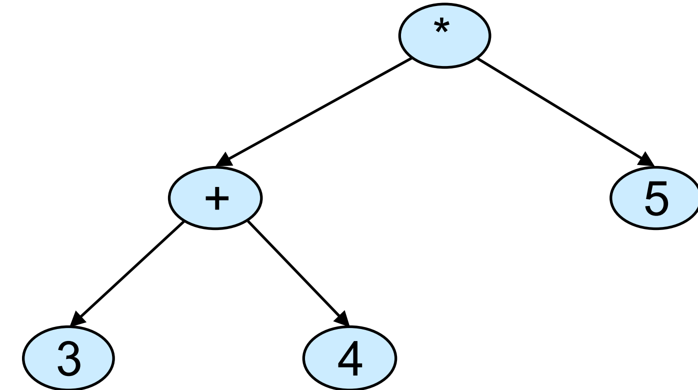
Operators come AFTER operands.

# Using BTNode: Expressions

## Postorder Traversal → Postfix (Reverse Polish)

**Evaluate Expressions Using a Stack**

Tree:




**Example Evaluation**
Expression:
`3 4 + 5 ×`

**Step 1 — Read 3**
Push onto stack: `[3]`

**Step 2 — Read 4**
Push. `[3, 4]`

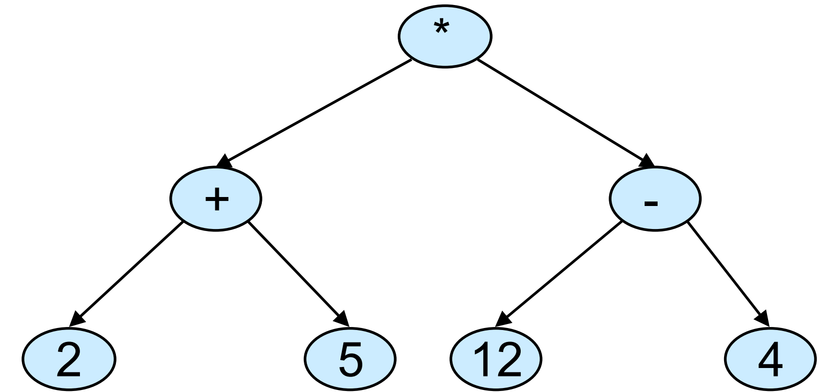
**Step 3 — Read +**
Pop:
`3 and 4`
Compute:
`3 + 4 = 7`
Push result.
`[7]`

**Step 4 — Read 5**
Push.
`[7, 5]`

**Step 5 — Read ×**
Pop:
`7 and 5`
Compute:
`7 × 5 = 35`
Push result.
`[35]`

Final answer:
`35`

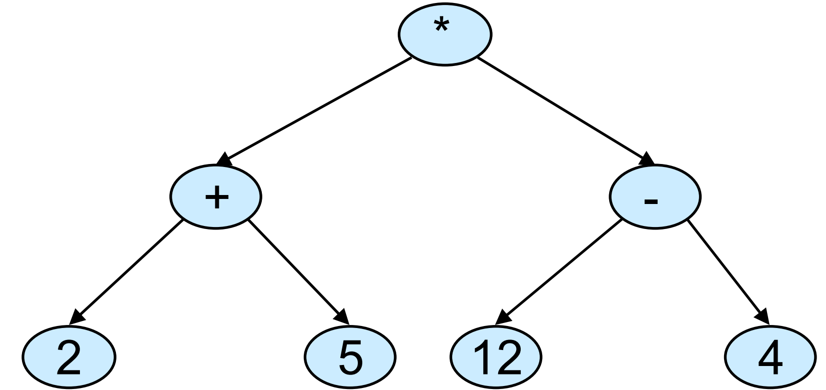
# Using BTNode: Expressions




- (2 + 5) * (12 - 4)
- Printing:
  - standard format with (…)
  - pre-order: * + 2 5 – 12 4 "Polish notation" - never need brackets!
  - post-order: 2 5 + 12 4 - * "Reverse Polish notation"
- Evaluating
  - recursive, post-order traversal of tree
- Reading
  - pre-order and post-order: easy
  - in order: hard! (COMP 261, simple parsing algorithms)

Same tree, different print format.
- Polish and Reverse Polish are easier to parse
- Require fewer keystrokes on a calculator.
- Reverse Polish can be evaluated while reading

# Using BTNode: Expressions



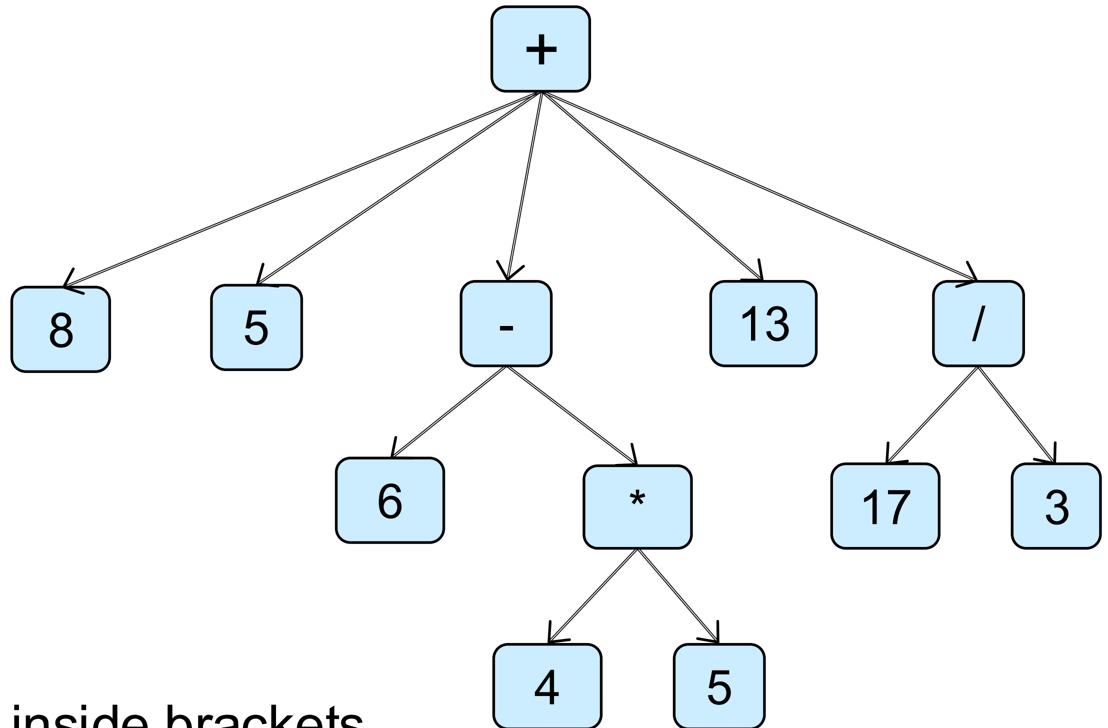

- pre-order: (Polish notation)
  * + 2 5 – 12 4
- post-order: (Reverse Polish)
  2 5 + 12 4 - *
- in-order: (infix)
  (2 + 5) * (12 – 4)

Same tree, different print format.

- Polish and Reverse Polish are easier to parse
- Require fewer keystrokes on a calculator.
- Reverse Polish can be evaluated while reading without storing the operators

# Cambridge Polish notation

- Expressions with operators with variable number of arguments
- => general tree.




- Writing such expressions:
  - Normal functional notation:
    Pre-order, with brackets and commas
    operator before brackets
    eg: +(8, 5, -(6, *(4, 5)), 13. /(17 3 ) )
  - Cambridge Polish Notation (Lisp)
    Pre-order, in brackets, no commas, operators inside brackets
    eg: ( + 8 5 ( - 6 ( * 4 5 ) ) 13 ( / 17 3 ) )

Same tree, different print format.
CPN is easier to parse

- Assignment: Make a calculator for this:
  - read (given), evaluate, print (REPL)

# Expression Tree Summary

| Traversal | Produces | Main Use |
| --- | --- | --- |
| Preorder | Prefix | Easy parsing |
| Postorder | Postfix | Stack evaluation |
| Inorder | Infix | Human-readable math |

# General Tree Nodes.

```
public class GTNode <E>  {
    private E item;
    private List<GTNode<E>> children;              // List, therefore children kept in order.

    /**Constructor for objects of class GTNode    */
    public GTNode(E item){
        this.item = item;
        this.children = new ArrayList<GTNode<E>>();
    }
    /** Getters and Setters     */
    public E getItem()              { return item; }
    public void setItem(E item) { this.item = item; }
```

Good design:
Protect the inner structure of the objects from modification by the rest of the program!

- What about the children?

```
public List<GTNode<E>> getChildren() { return Collections.unmodifiableList(children); }
```

# General Tree Node: Alternative design

- Keep the List of children internal to the class.
- Provide methods to add and remove children

```
/** Getters and Setters     */
public GTNode<E> getChild(int indx)                      { return children.get(indx); }
public void addChild(GTNode<E> child)                    { children.add(child); }
public void addChild(int indx, GTNode<E> child) { children.add(indx, child); }
public GTNode<E> removeChild(int indx)                   {return children.remove(indx); }
public void setChild(int indx, GTNode<E> child)    { children.set(indx, child); }
public int numChildren()                                 {return children.size(); }
```

- What about iterating through the children?
  - Could use a counted for loop:

```
for (int i=0; i<node.numChildren(); i++){ GTNode<String> child =node.getChild(i); ... }
```

  - Could enable foreach loop:

```
for (GTNode<String> child : node) { …. }
```

HOW?

# Iterable and Iterators

Example of a foreach loop:

```
for (String s : names) {
    UI.println(s);
}
```

A foreach loop needs a way to:

1. move through items one by one
2. Check if more items exist
3. Get the next item

For Java to use for-each on an object, that object must be:

Iterable

# Iterable and Iterators

- To be able to iterate along an object using foreach loop, the object must be Iterable:
- The object's class must implement Iterable<??> and have a
  ```
  public Iterator<??> iterator(){…}
  ```
  method which returns an Iterator object
- An Iterator object must have a
  ```
  public boolean hasNext() {…}
  ```
  method, and a
  ```
  public ??? next() {…}
  ```
  method

# Iterable and Iterators

**Why This Is Important in Trees**

- Tree traversals often return iterators.

Example:

```
for (Person p : familyTree)
```

The iterator may internally perform:

- preorder traversal
- BFS traversal
- DFS traversal
- while foreach looks simple

# General Tree Nodes.

```
public class GTNode <E> implements Iterable <GTNode<E>>  {
    private E item;
    private List<GTNode<E>> children;                    // List, therefore children kept in order.

    /**Constructor for objects of class GTNode    */
    public GTNode(E item){
        this.item = item;
        this.children = new ArrayList<GTNode<E>>();
    }
    /** Getters and Setters     */
    public E getItem()              { return item; }
    public void setItem(E item) { this.item = item; }
      :
      :
    public Iterator<GTNode<E>> iterator() { return children.iterator(); }
}
```

# Using GTNode with iterator

- look for a node in a tree with a particular item (recursive depth-first traversal)

```
public GTNode<String> findNode(GTNode<String> root,  String label){
    if (root == null){ return null;}
    if (root.getItem().equals(label) ) {
        return root;
    }
    for (GTNode<String> child : root) {
        GTNode<String> ans = findNode(child, label);
        if (ans != null) {
            return ans;
        }
    }
    return null;
}
```

# Using GTNode with iterator

- Evaluate an expression tree with + operator (recursive depth-first traversal)

```
public double evaluate(GTNode<ExpElem> root){
    if( root == null) { return Double.NaN; }
    ExpElem elem = root.getItem();
    if (elem.getItem().equals("#") ) { return elem.value; }
    double sum = 0;
    for (GTNode< ExpElem > child : root) {
        sum = sum + evaluate(child);
    }
    return sum;
}
```

# Tree Traversal Patterns

- General pattern for recursive depth-first traversing General Trees:

```
to traverseDF(node):
    process node
    foreach child of node
        traverseDF (child)
```

Need to modify to:
- stop early
- return values
- use the depth

…

- General pattern for iterative traversing General Trees:

```
to traverseBF(node)
    put node on queue
    while (queue is not empty)
        dequeue node from queue
        process node
        foreach child of node
            put child on queue
```

```
to traverseDF(node)
    put node on stack
    while (stack is not empty)
        pop node from stack
        process node
        foreach child of node
            push child on stack
```

# Tree Traversal Patterns

- General pattern for recursive depth-first traversing General Trees: passing depth and other information **down** the tree.

```
to traverseDF(node)
    traverseDF(node, 0, info)

to traverseDF(node, depth, info):
    process node at depth with info
    for each child of node
        traverseDF (child, depth+1, addto(info))
```

Entry method
Sets up the recursion

Helper method
Does the recursion

Note: The info might be a collection object that is passed through all the recursion and values get added to it.

# Tree Traversal Patterns

- General pattern for recursive depth-first traversing General Trees: passing information back **up** the tree.

```
to traverseDF(node):
    ans = …. process node
    foreach child of node
        childAns = traverseDF (child)
        combine childAns into ans
    return ans
```

# Review

Create a tree **(Node-based binary tree data structure)** to choose a university major: Engineering, Medical, Art, Accounting, Law, Business.