Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Traversing trees

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Traversing

What is Traversing?

- Traversing a tree data structure is the process of visiting each node in the tree exactly once, following a specific order. It's like exploring every room in a building, systematically.
- In the context of binary trees (where each node has at most two children), traversal algorithms determine the order in which you visit the nodes.

Traversing

There are three fundamental types of tree traversal:

- Preorder Traversal:
  - Visit the root node first.
  - Then, recursively traverse the left subtree.
  - Finally, recursively traverse the right subtree.
- Inorder Traversal:
  - Recursively traverse the left subtree.
  - Visit the root node next.
  - Finally, recursively traverse the right subtree.
- Postorder Traversal:
  - Recursively traverse the left subtree.
  - Finally, recursively traverse the right subtree.
  - Then, visit the root node last.

Review

Differences Between the Types:

Traversal Type	Order of Node Visits	Description
Preorder	Root, Left, Right	Processes the root node before its children
Inorder	Left, Root, Right	Processes the left subtree before the root node
Postorder	Left, Right, Root	Processes the children before the root node

Recap: Tree-structured data

Tree-structured data is a hierarchical data organisation where data is arranged in a parent-child relationship.

Here's a breakdown:

- Root Node: The topmost node in the tree, with no parent.
- Parent Node: A node that has one or more child nodes.
- Child Node: A node that is directly below a parent node.
- Leaf Node: A node that has no children.

Traversal

How Binary Traversal Fits In:

- Binary traversal specifically refers to traversing a binary tree (a tree where each node has at most two children). The preorder, inorder, and postorder traversal
- Algorithms are designed to work with binary trees. The key difference is that the algorithms are tailored to handle the two-child constraint, ensuring that all nodes are visited exactly once.

summary:

Traversal is a systematic way to visit every node in a tree. The different traversal types offer different orders of node visits, each suited for specific tasks. Tree-structured data is a hierarchical organisation, and binary traversal is a specialised technique for navigating binary trees.

Traversal

How Binary Traversal Fits In:

- Binary traversal specifically refers to traversing a binary tree (a tree where each node has at most two children). The preorder, inorder, and postorder traversal
- Algorithms are designed to work with binary trees. The key difference is that the algorithms are tailored to handle the two-child constraint, ensuring that all nodes are visited exactly once.

summary:

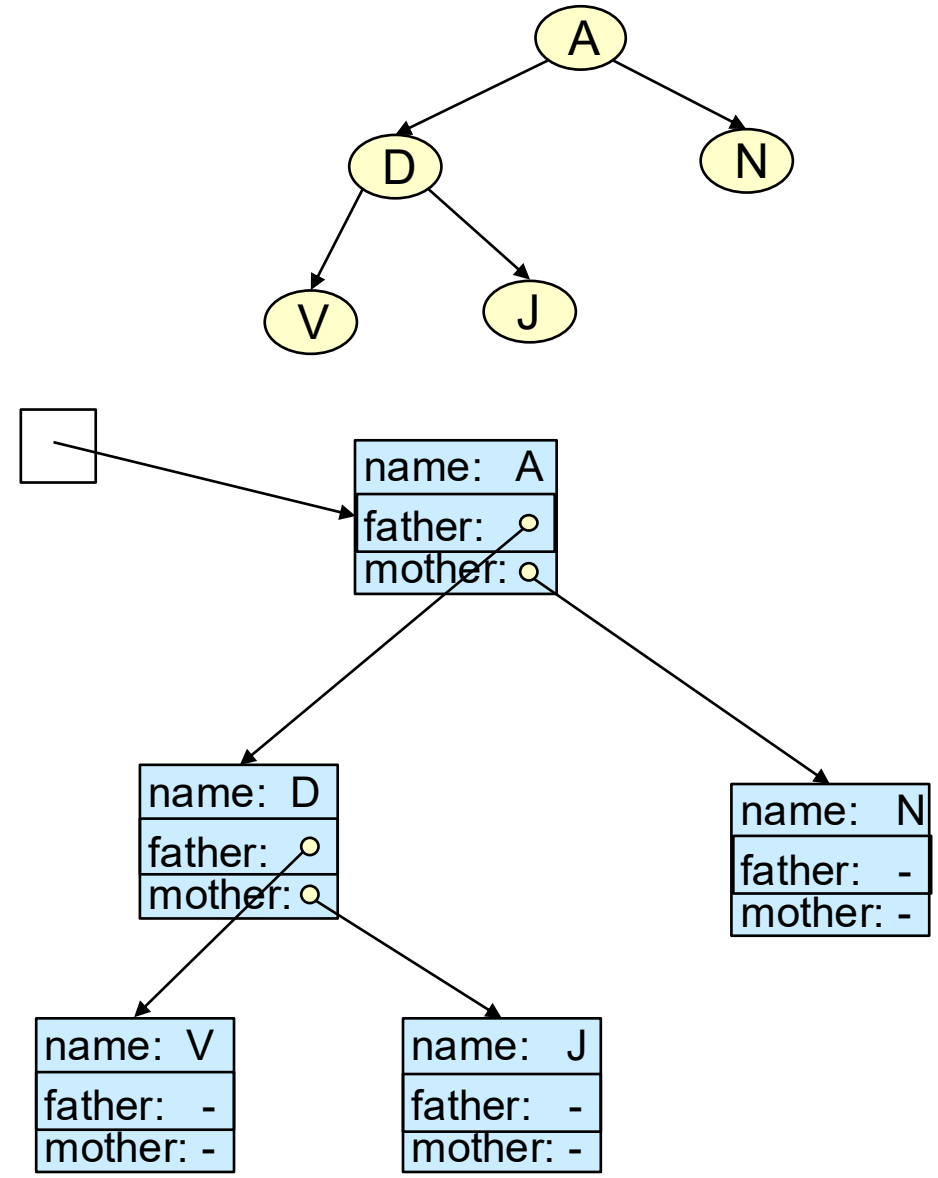
Traversal is a systematic way to visit every node in a tree. The different traversal types offer different orders of node visits, each suited for specific tasks. Tree-structured data is a hierarchical organisation, and binary traversal is a specialised technique for navigating binary trees.

Data Structures for Tree Structured data

- But why do we have to go via a key or an index?
- "Linked Structure"

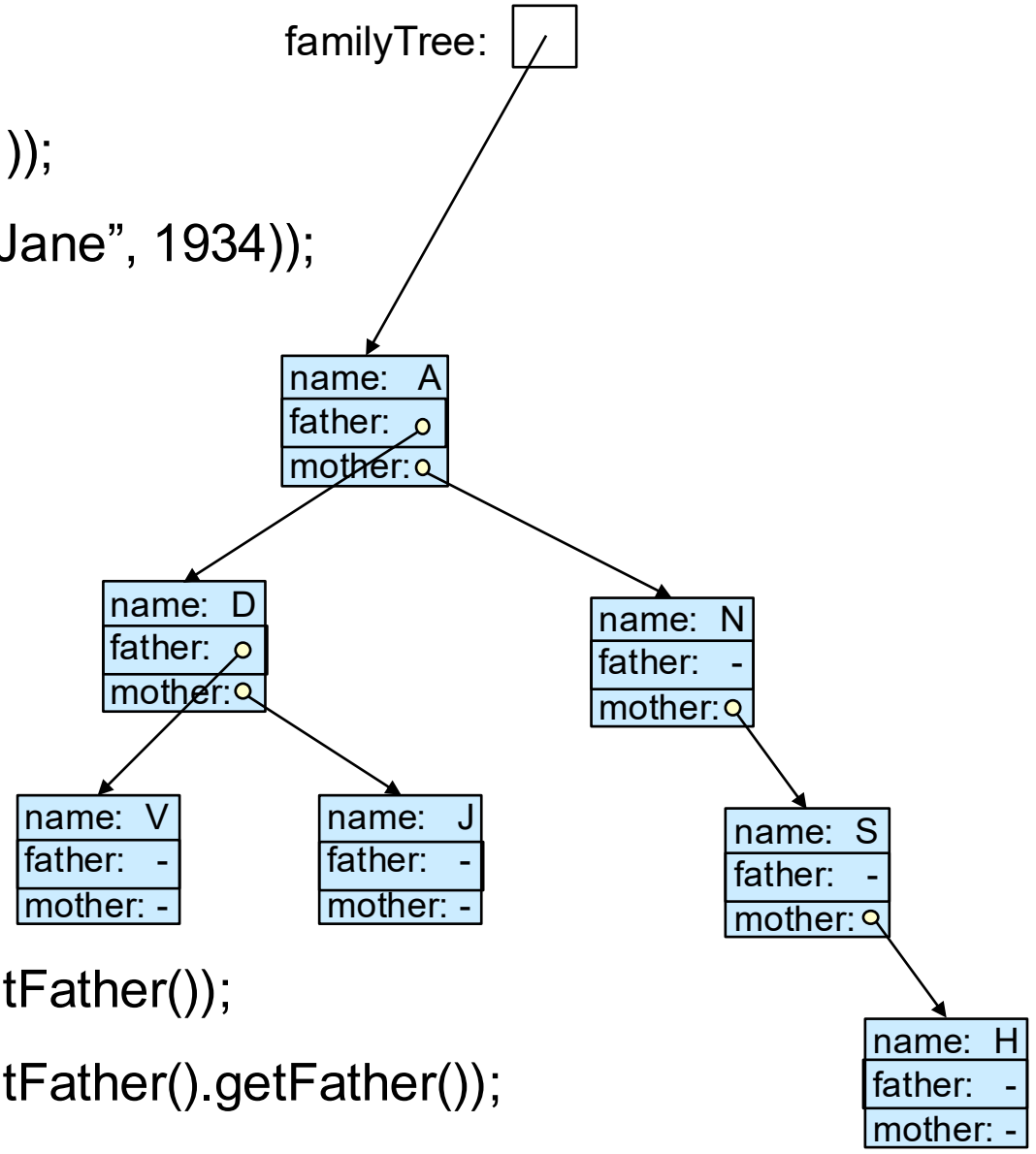
```
public class Person {
    private String name;
    private int dob;
    private Person father;
    private Person mother;
    public Person(String n, int d){ name = n; dob = d; }
        :
    public Person getMother(){ return mother; }
    public Person getFather(){ return father; }
    public void setMother(Person p){ mother = p; }
    public void setFather(Person p){ father = p; }
    public void toString(){ return name +"("+dob+")"; }
}
```

Person
Recursive Data
Structure!



Using “linked” tree structures

```
familyTree  = new Person(“Alice”, 2000);
familyTree.setFather(new Person(“David”, 1971));
familyTree.getFather().setMother(new Person(“Jane”, 1934));
```




```
UI.println(familyTree.getMother());
UI.println(familyTree.getFather().getMother());
UI.println(familyTree.getFather().getMother().getFather());
UI.println(familyTree.getFather().getMother().getFather().getFather());
```

Using "linked" structures: looping down tree

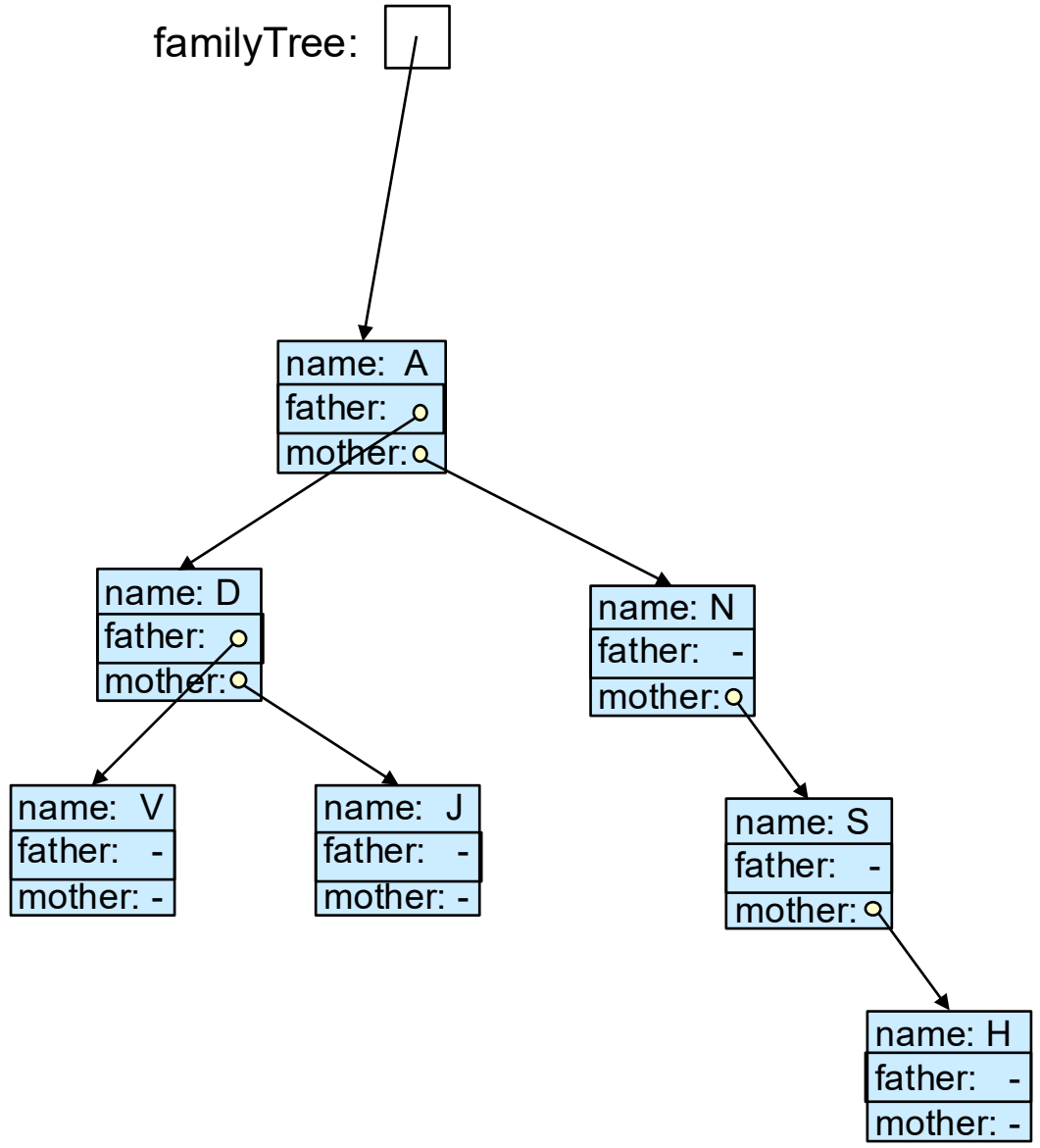
Stepping along a path from root.

eg: Print out maternal line:

```
Person p = familyTree;
while (p != null){
    UI.println(p);
    p = p.getMother();
}
```

p:

runs off the end



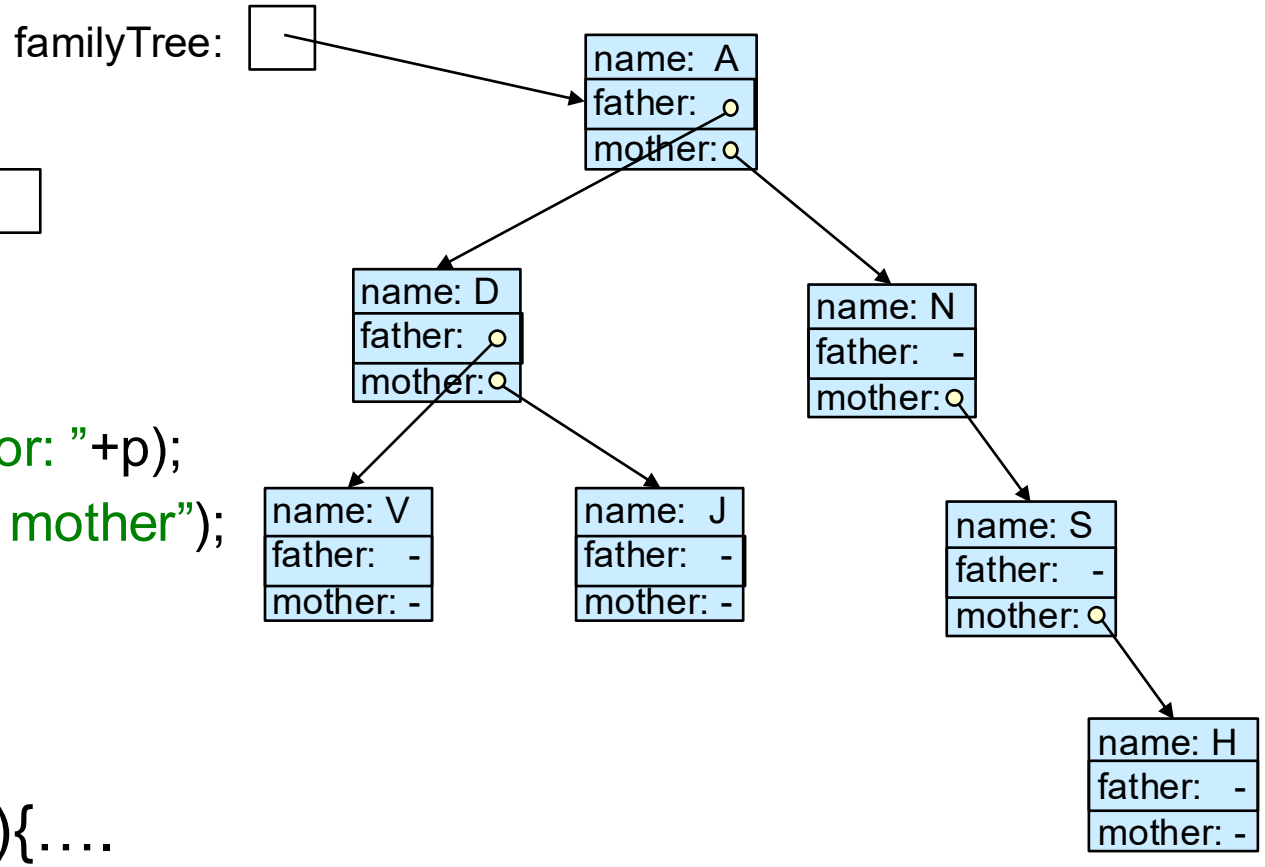
Using "linked" structures: looping down tree

Finding a leaf node:

eg: Add next maternal ancestor:

```
Person p = familyTree;
while (p.getMother() != null){
      p = p.getMother();
}
UI.println("Oldest known maternal ancestor: "+p);
String name = UI.askString("Name of her mother");
int dob = UI.askInt("year of birth");
p.setMother(new Person(name, dob));
```

Why?



Running off the end: **while** (p != null){….

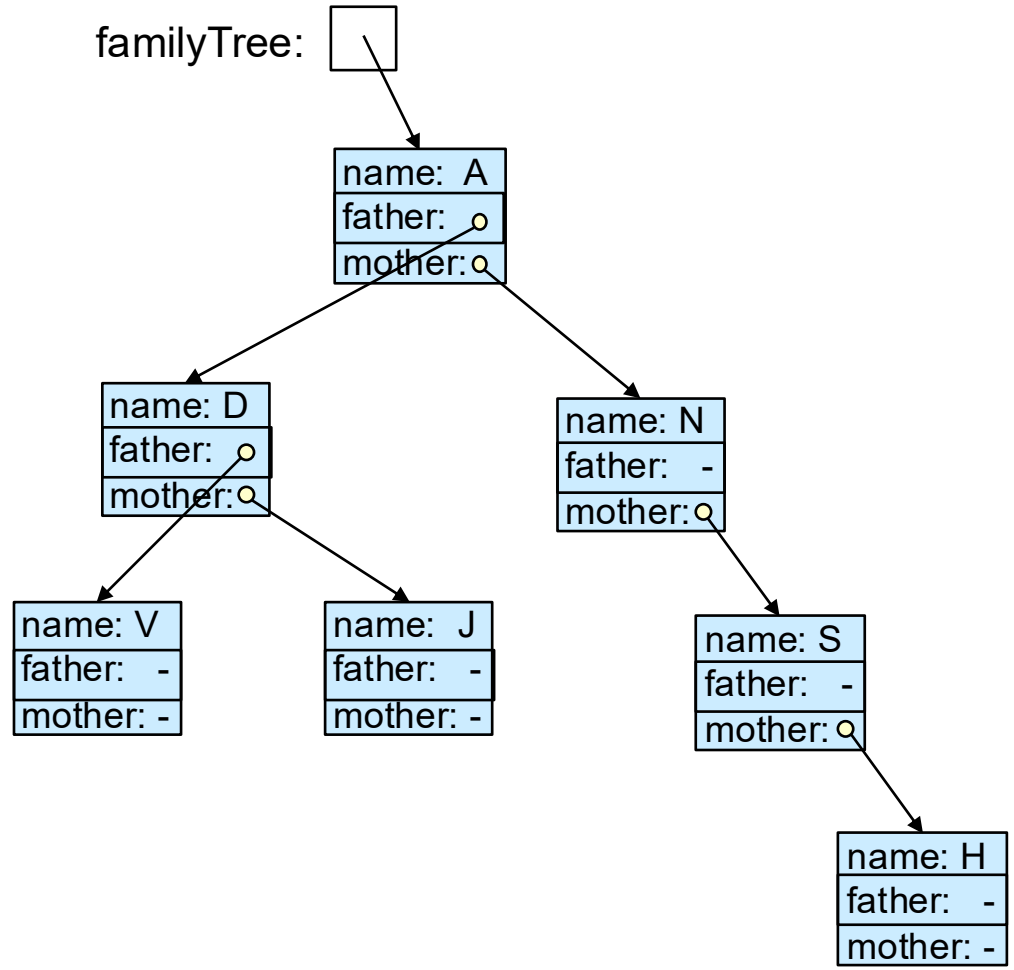
Stopping at the end: **while** (p.getMother() != null){….

Using “linked” tree structures:

Add next maternal ancestor:

```
public Person oldestMatAnc (Person p){
      Person tmp = p;
      while (tmp.getMother()!=null){
            tmp = tmp.getMother();
      }
      return tmp;
}
```

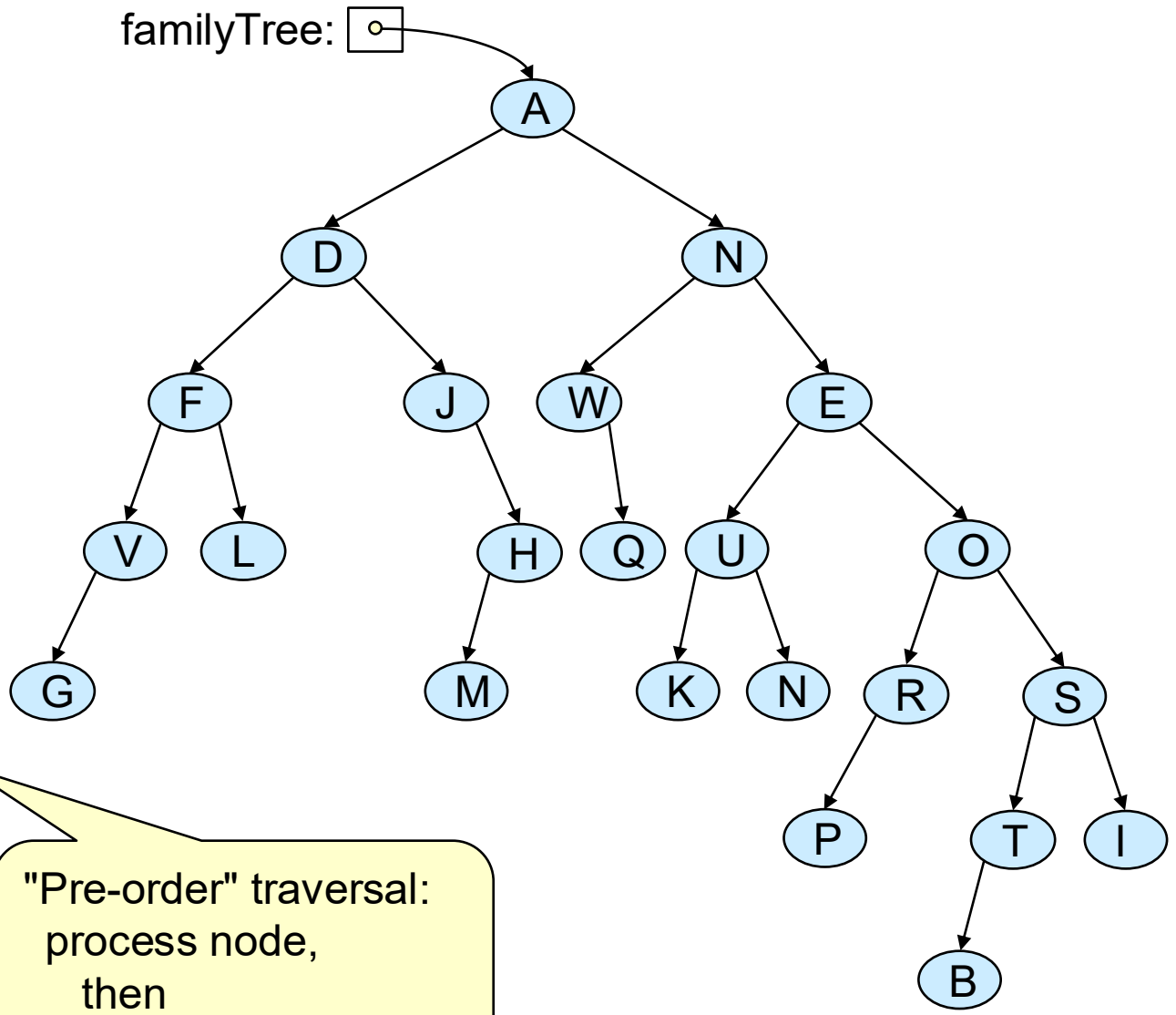
```
      :
Person p = oldestMatAnc(familyTree);
UI.println(“Oldest known maternal ancestor: ”+p);
String name = UI.askString(“Name of her mother”);
int dob = UI.askInt(“year of birth”);
p.setMother(new Person(name, dob));
```



Traversing a tree

- Traversing => processing every node
- Traversing is harder with a loop; much easier to use recursion.

```
public void printAll (Person p){
    if (p != null){
        UI.println(p);
        printAll(p.getFather());
        printAll(p.getMother());
    }
}

printAll(familyTree);
```



"Depth First" traversal:
process whole subtree
before next child

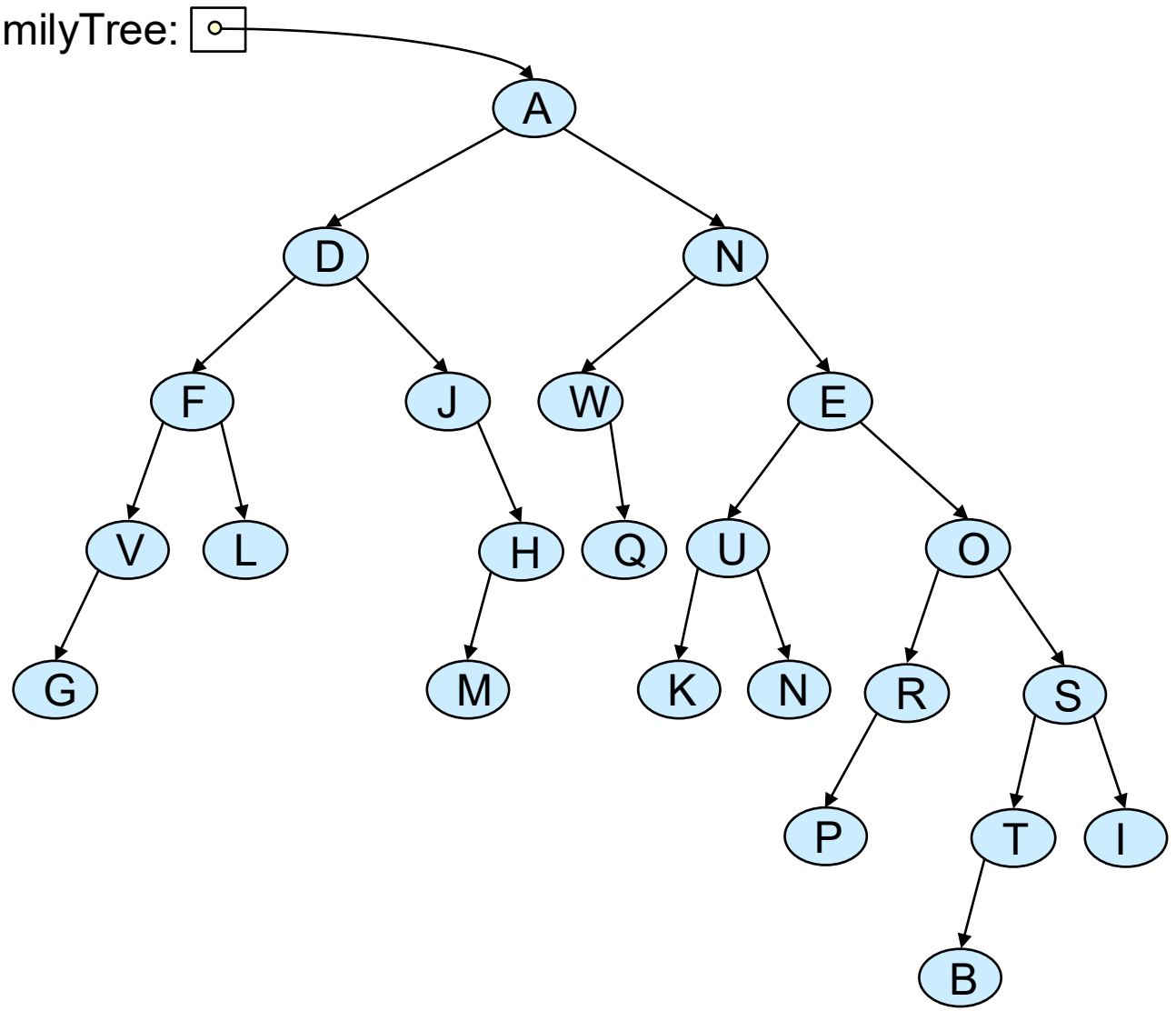
"Pre-order" traversal:
process node,
then
process subtrees.

More traversals: depth-first, post-order

- "Depth-first, Post-order" traversal:

process subtrees
then
process node:

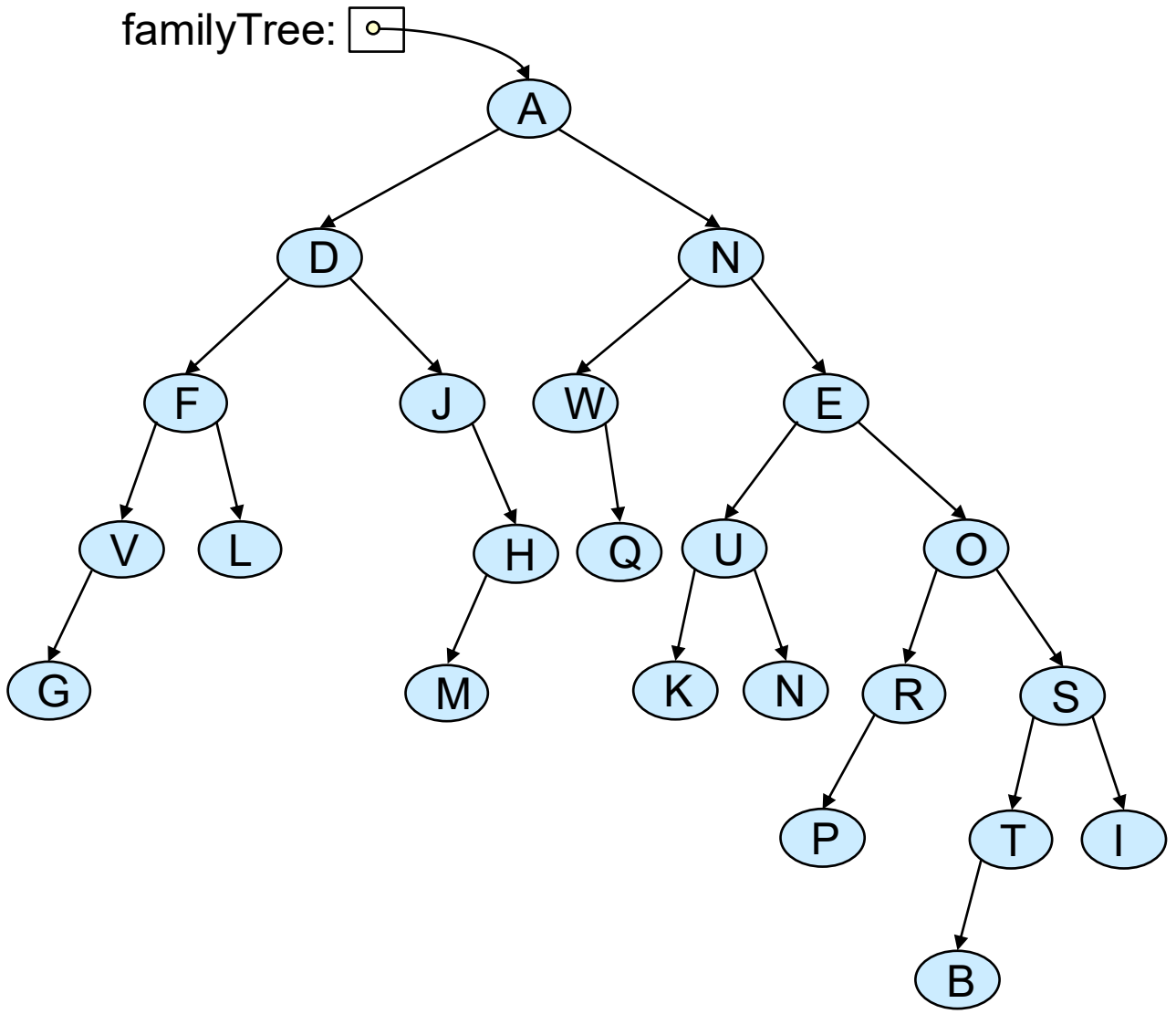
```
public void printAll (Person p){
    if (p!=null){
        printAll(p.getFather());
        printAll(p.getMother());
        UI.println(p);
    }
}

printAll(me);
```



Another traversal: depth-first, in-order

- Depth-first, in-order traversal

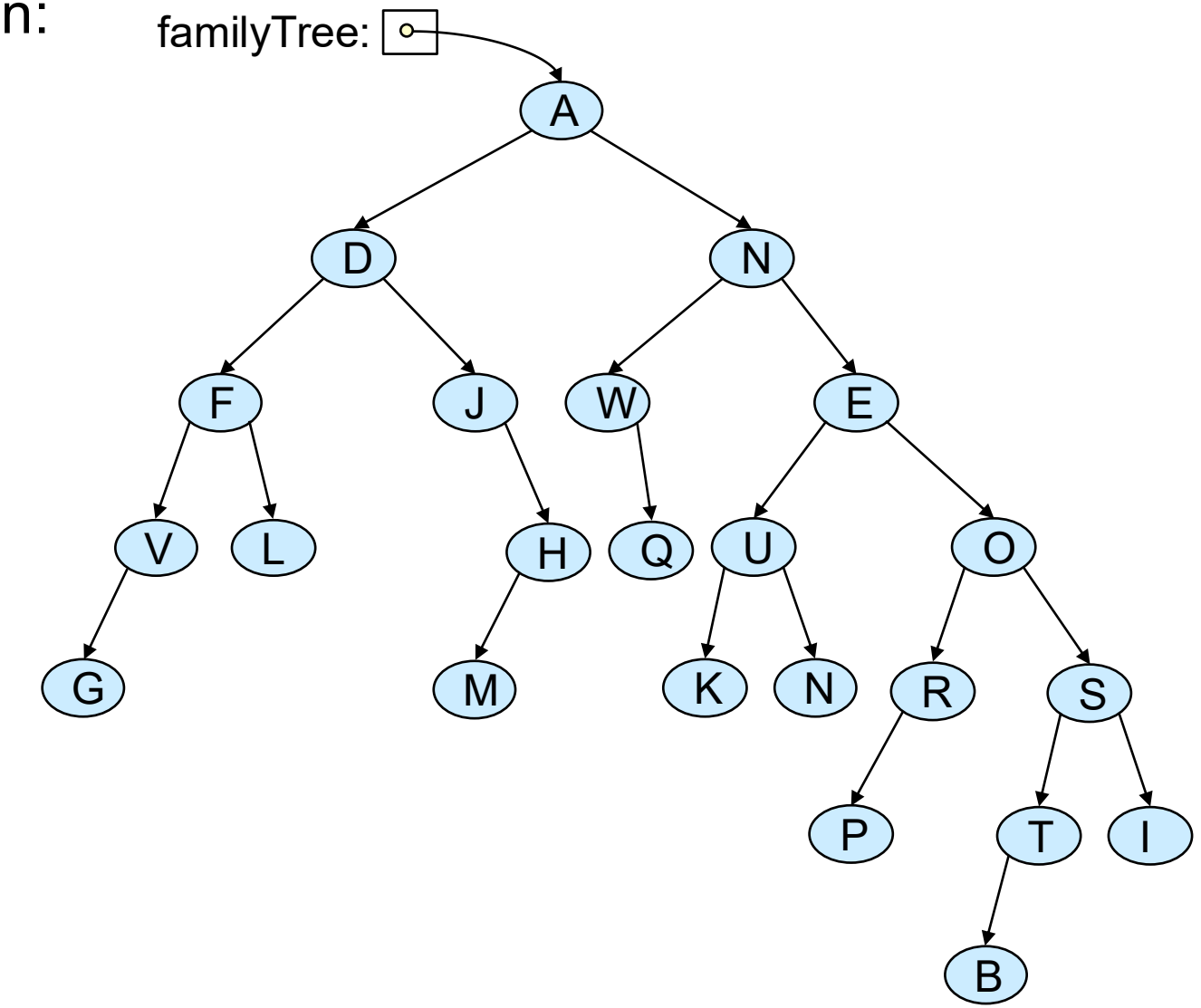
```
public void printAll (Person p){
    if (p != null){
        // traverse left child subtree
        printAll(p.getFather());
        // process node p
        UI.println(" < " + p + " > ");
        // traverse right child subtree
        printAll(p.getMother());
    }
}

printAll(familyTree);
```



Traversing with an extra parameter

- Traversing the tree, printing generation:

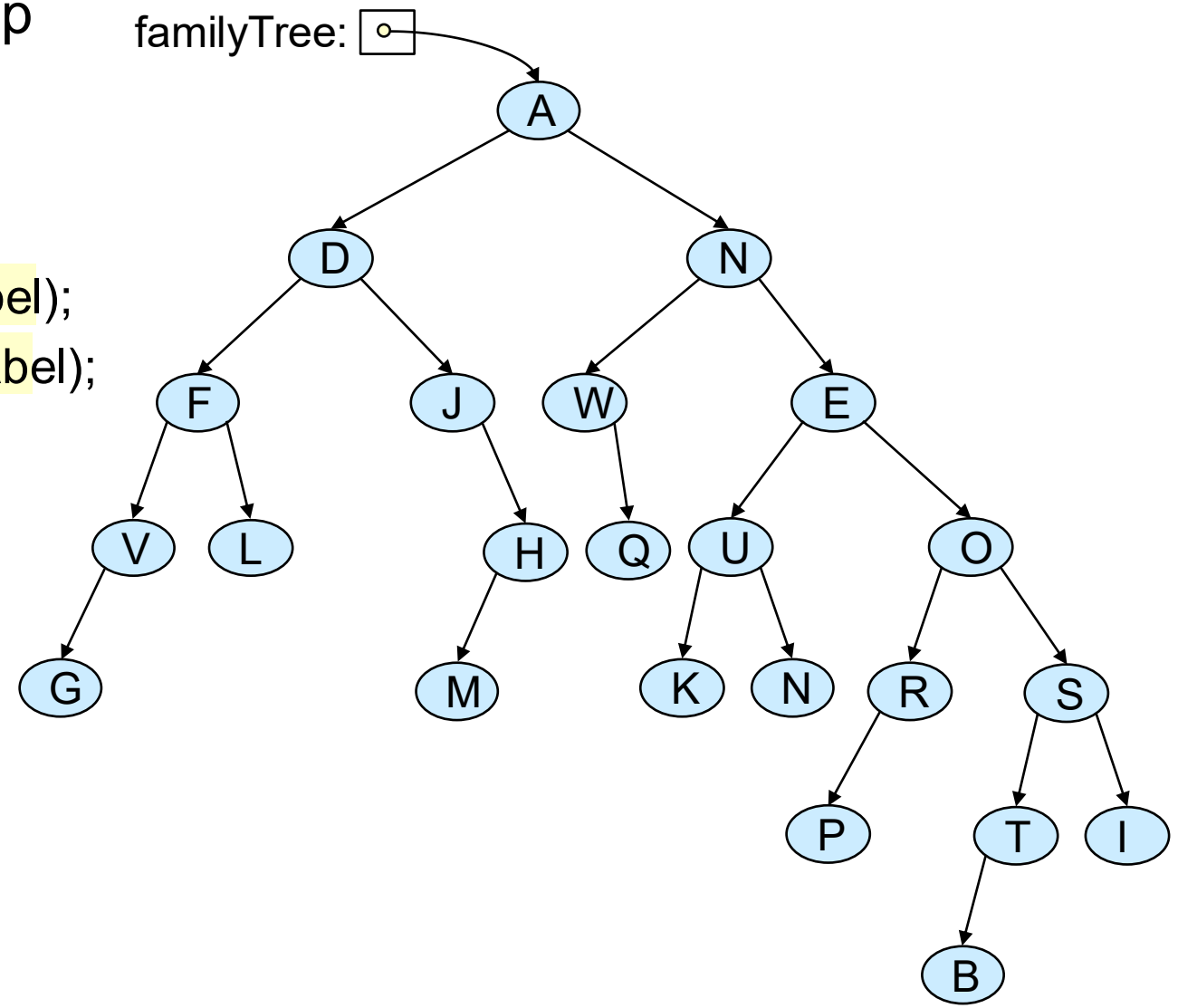
```
public void printAll (Person p, int gen){
    if (p!=null){
        UI.println(gen + ": " + p);
        printAll(p.getFather(), gen+1);
        printAll(p.getMother(), gen+1);
    }
}
printAll(familyTree, 1);
```



Traversing with an extra parameter

- Traversing the tree: printing relationship

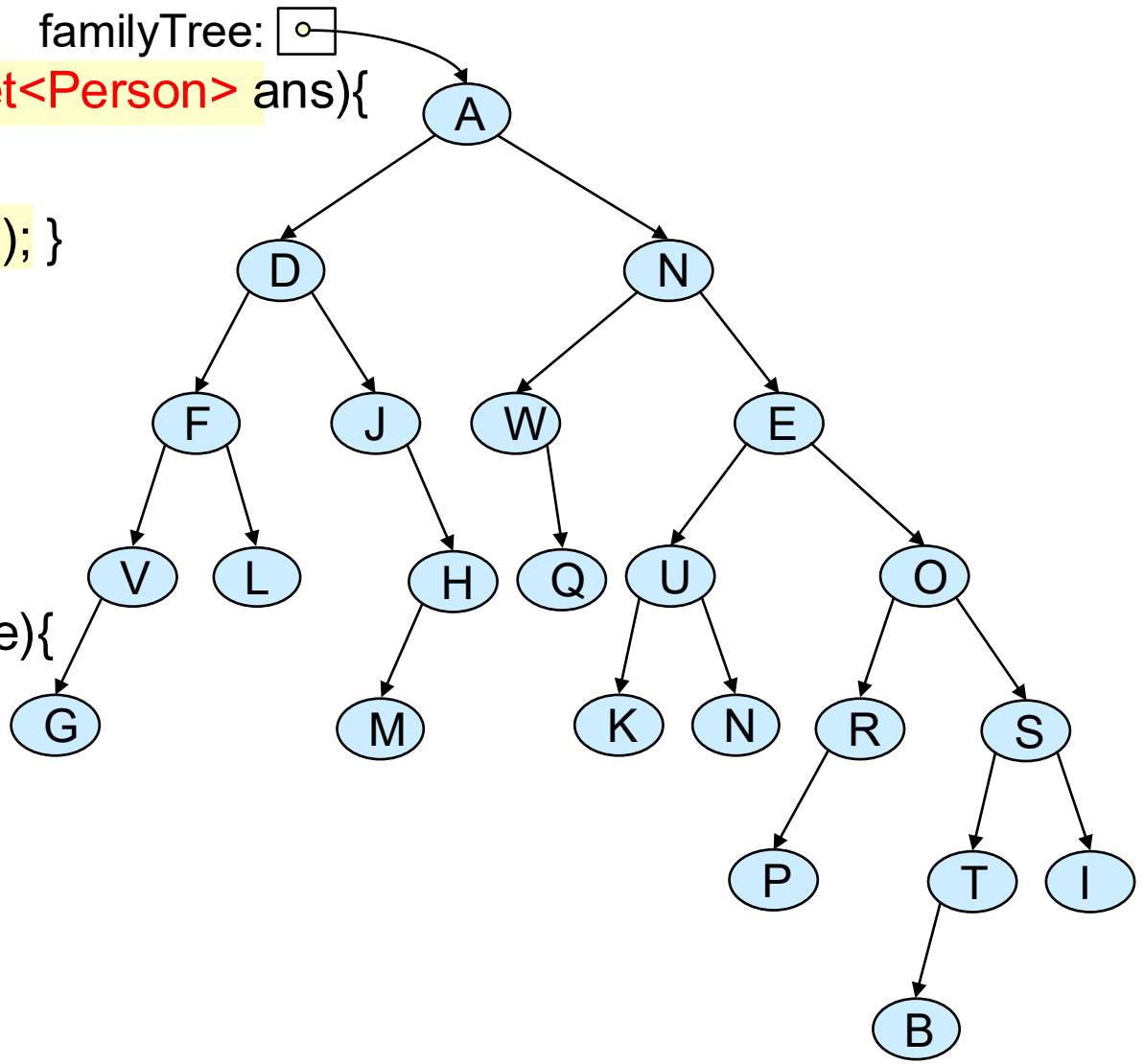
```
public void printAll (Person p, String label){
    if (p!=null){
        UI.println(label + ": " + p);
        printAll(p.getFather(), "father of " + label);
        printAll(p.getMother(), "mother of "+ label);
    }
}
printAll(familyTree, "me");
```



Traversing and collecting an answer

- Traversing the tree: find all with name.

```
public void findAll_rec (Person p, String name, Set<Person> ans){
    if (p!=null){
        if (p.getName().equals(name)){  ans.add(p); }
        findAll_rec(p.getFather(), name, ans);
        findAll_rec(p.getMother(), name, ans);
    }
}

public Set<Person> findAll (Person p, String name){
    Set<Person> ans = new HashSet<Person>();
    findAll_rec(p, name, ans);
    return ans;
}

findAll(familyTree, "Jane");
```



Tree Traversal

- Traversing a tree = visiting every node in the tree

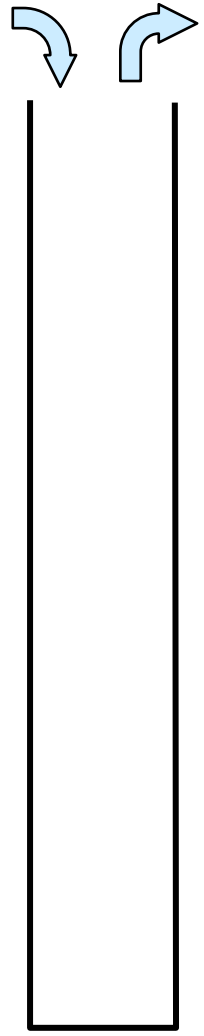
- Depth-first traversal:
  - Follows all the way down one subtree before starting other subtree(s)
  - Easy to do with recursion.

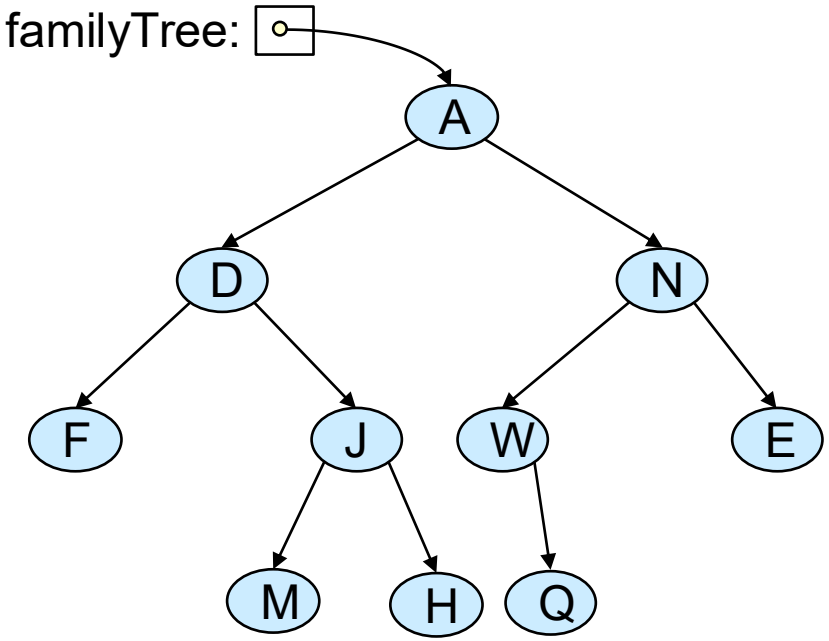
- For Binary trees (two children per node)
  - pre-order: visit parent node then traverse child subtrees,
  - post-order: traverse child subtrees then visit parent node
  - in-order: traverse one child subtree
    then visit parent node
    then traverse other subtree
    - (binary trees only)

Depth first traversal without recursion

- Use a stack to store the nodes that need to be worked on

```
public void preOrderDF (Person root){
    Stack<Person> todo = new Stack<Person>();
    todo.push(root);
    while ( ! todo.isEmpty() ){
        Person p = todo.pop()
        UI.println(p);
        if ( p.getMother() != null ){
            todo.push(p.getMother());
        }
        if ( p.getFather() != null ){
            todo.push(p.getFather());
        }
    }
}
```

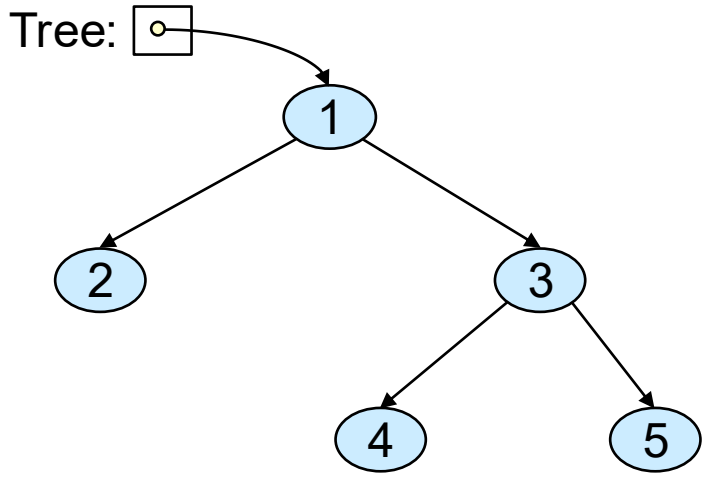


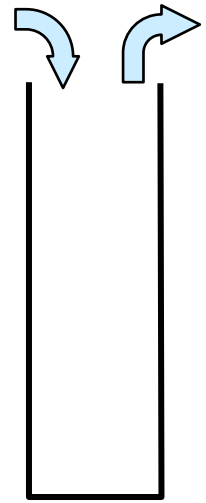


A D F J H M N W Q E

Depth first traversal without recursion

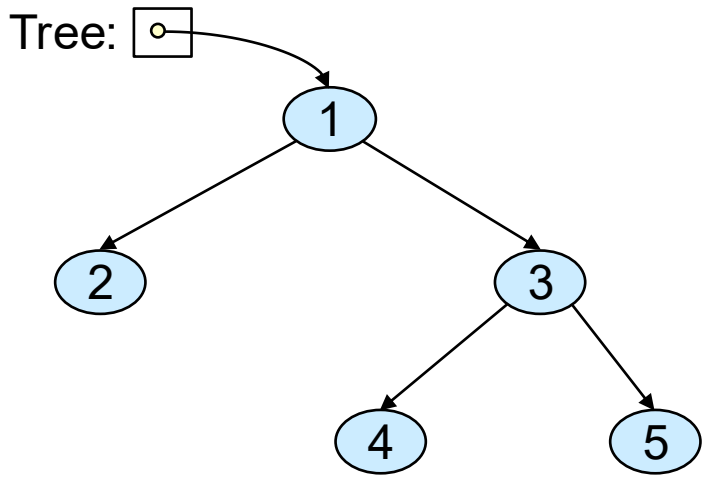
```
import java.util.Stack;

class Node {
    int data;
    Node left, right;

    Node(int value) {
        data = value;
        left = right = null;
    }
}
```

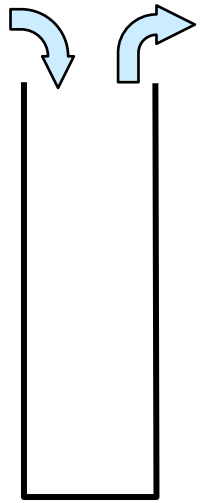




Depth first traversal without recursion

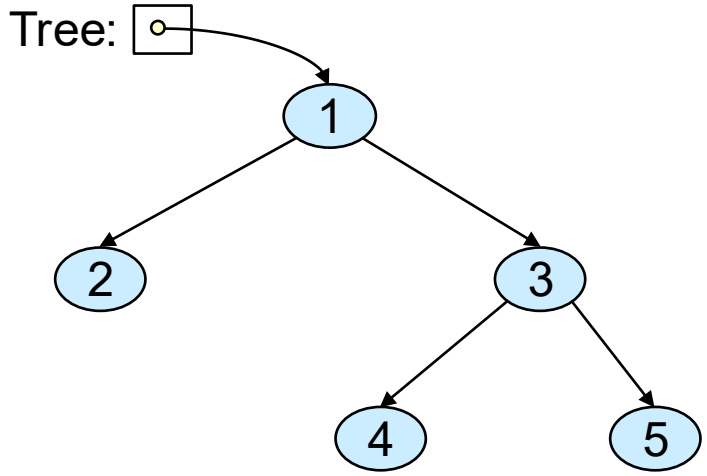
```
public class IterativeDFS {
    static void dfs(Node root) { // Preorder DFS without recursion
        if (root == null)
            return;

        Stack<Node> stack = new Stack<>();

        stack.push(root); // push root node

        while (!stack.isEmpty()) {

            Node current = stack.pop(); // pop top node
            System.out.print(current. data + " "); // visit node


            if (current.right != null) {.    // push right first
                stack.push(current. right);
            }

            if (current.left != null) {. // push left second
                stack.push(current. left);
            }
        }
    }
```



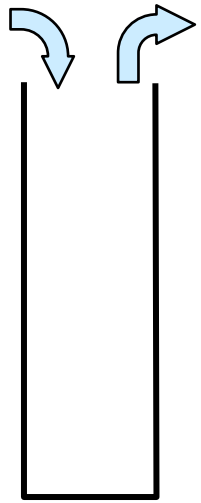


Depth first traversal without recursion

```
    public static void main(String[] args) {

        Node root = new Node(1);

        root.left = new Node(2);
        root.right = new Node(3);

        root.left.left = new Node(4);
        root.left.right = new Node(5);

        System.out.println("DFS Traversal:");

        dfs(root);
    }
}
```



DFS Traversal:
1 2 4 5 3



Depth first traversal without recursion

A stack is **Last In First Out (LIFO)**.

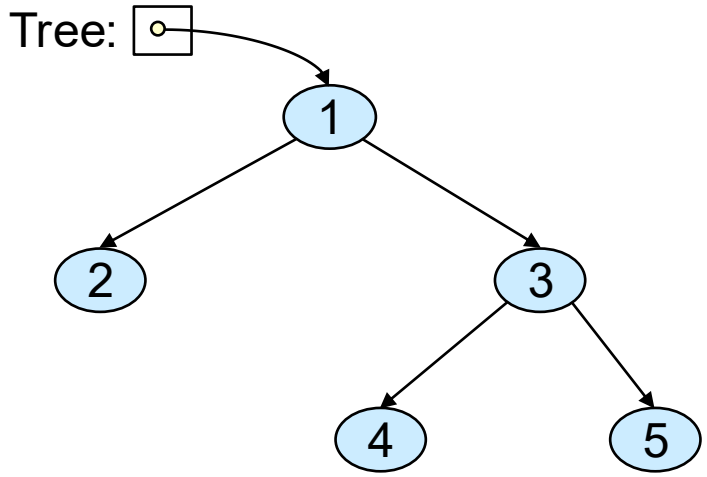
Therefore:

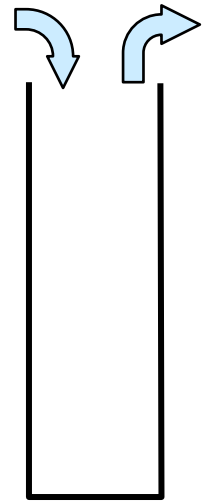
right child first, then left child second

Then the left child gets processed first.

That preserves normal preorder order:

Root → Left → Right





Depth first traversal without recursion

Step-by-Step Stack Movement

Start:

Stack: [1]

Pop 1, visit it.

Push 3 then 2:

Stack: [3, 2]

Pop 2, visit it.

Push 5 then 4:

Stack: [3, 5, 4]

Pop 4:

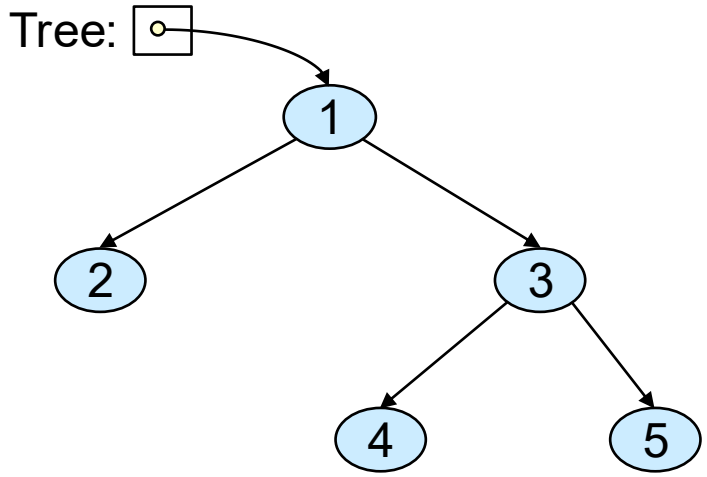
Stack: [3, 5]

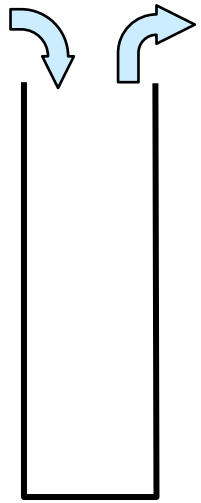
Pop 5:

Stack: [3]

Pop 3:

Stack: []





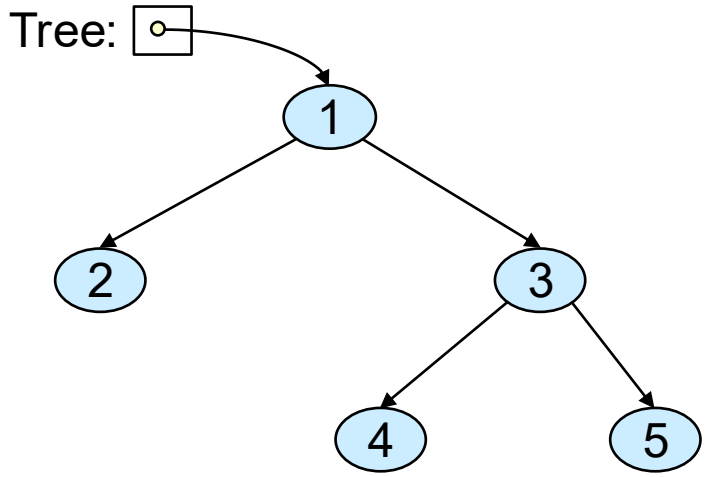
Time and Space Complexity

- Time: $O(n)$
- Space: $O(n)$ worst case

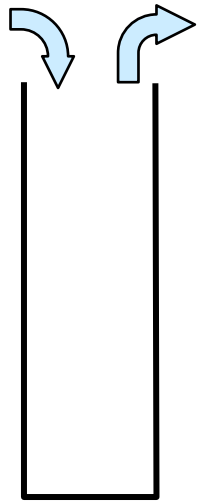
because every node is visited once.

Depth first traversal without recursion

```
    public static void main(String[] args) {

        Node root = new Node(1);

        root.left = new Node(2);
        root.right = new Node(3);

        root.left.left = new Node(4);
        root.left.right = new Node(5);

        System.out.println("DFS Traversal:");

        dfs(root);
    }
}
```



DFS Traversal:
1 2 4 5 3

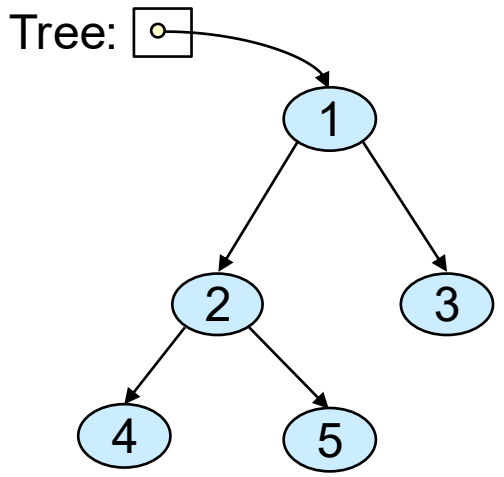


Depth first traversal without recursion

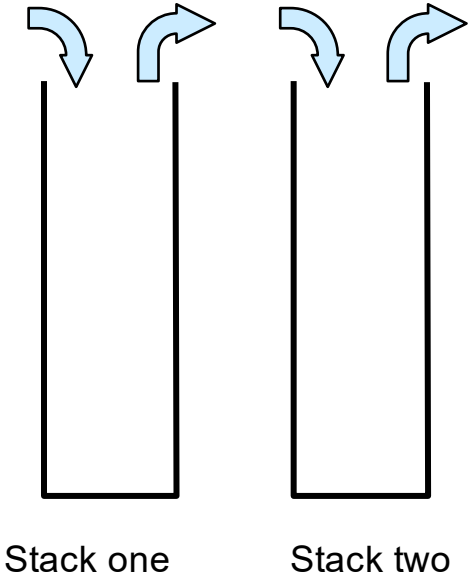
How do we do post-order?

Postorder traversal without recursion is a little trickier because the order is:
Left → Right → Root

The challenge is that the root must be processed **after** both children.
A common iterative solution uses **two stacks**.



stack1 → used for processing
stack2 → used to reverse the order



Postorder output:
4 5 2 3 1

Depth first traversal without recursion

```
static void postOrder(Node root) {
        if (root == null)
            return;

        Stack<Node> stack1 = new Stack<>();
        Stack<Node> stack2 = new Stack<>();

        stack1.push(root);                      // Start with root

        while (!stack1.isEmpty()) {
            Node current = stack1.pop();
            stack2.push(current);               // Put node into second stack

            if (current.left != null) {         // Push left and right children
                stack1.push(current.left);
            }

            if (current.right != null) {
                stack1.push(current.right);
            }
        }
        while (!stack2.isEmpty()) {// Print nodes from second stack
            UI.print(stack2.pop().data + " ");
        }
}
```

Depth first traversal without recursion

Step-by-Step Stack Movement

Start:

Stack1: [1]
Stack2: []

```
current = stack1.pop();
```

Pop 1: Push (1) into stack2, Push left(2), right(3) into

```
stack1.push(2);
stack1.push(3);
```

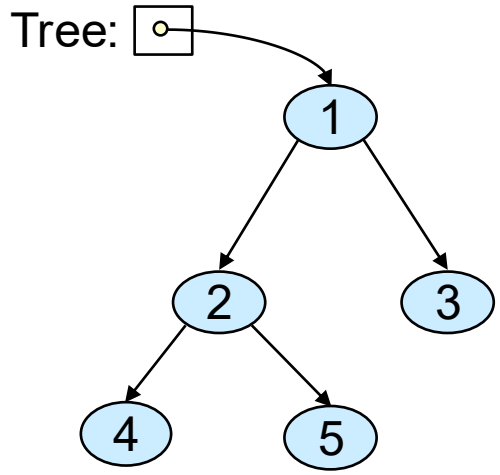
Stack1: [2,3]
Stack2: [1]

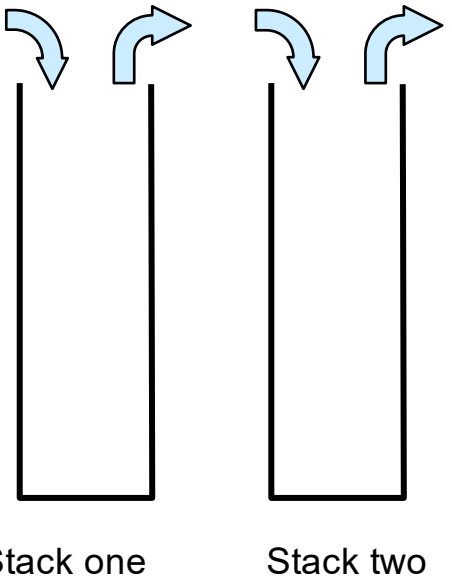
Pop (3):

Stack1: [2]
Stack2: [1,3]

Pop (2): Push children 4 and 5

Stack1: [4,5]
Stack2: [1,3,2]





Depth first traversal without recursion

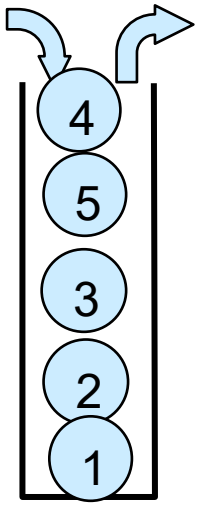
Step-by-Step Stack Movement

Pop (5)

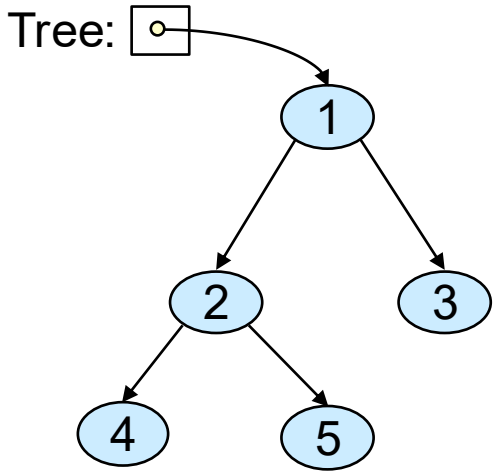
Stack1: [4]
Stack2: [1,3,2,5]

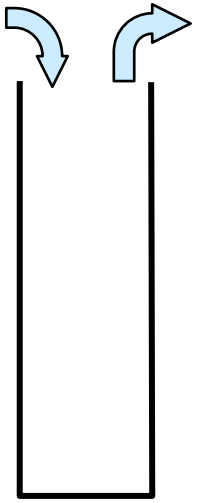
Pop (4)

Stack1: [ ]
Stack2: [1,3 ,2, 5, 4]



Stack two





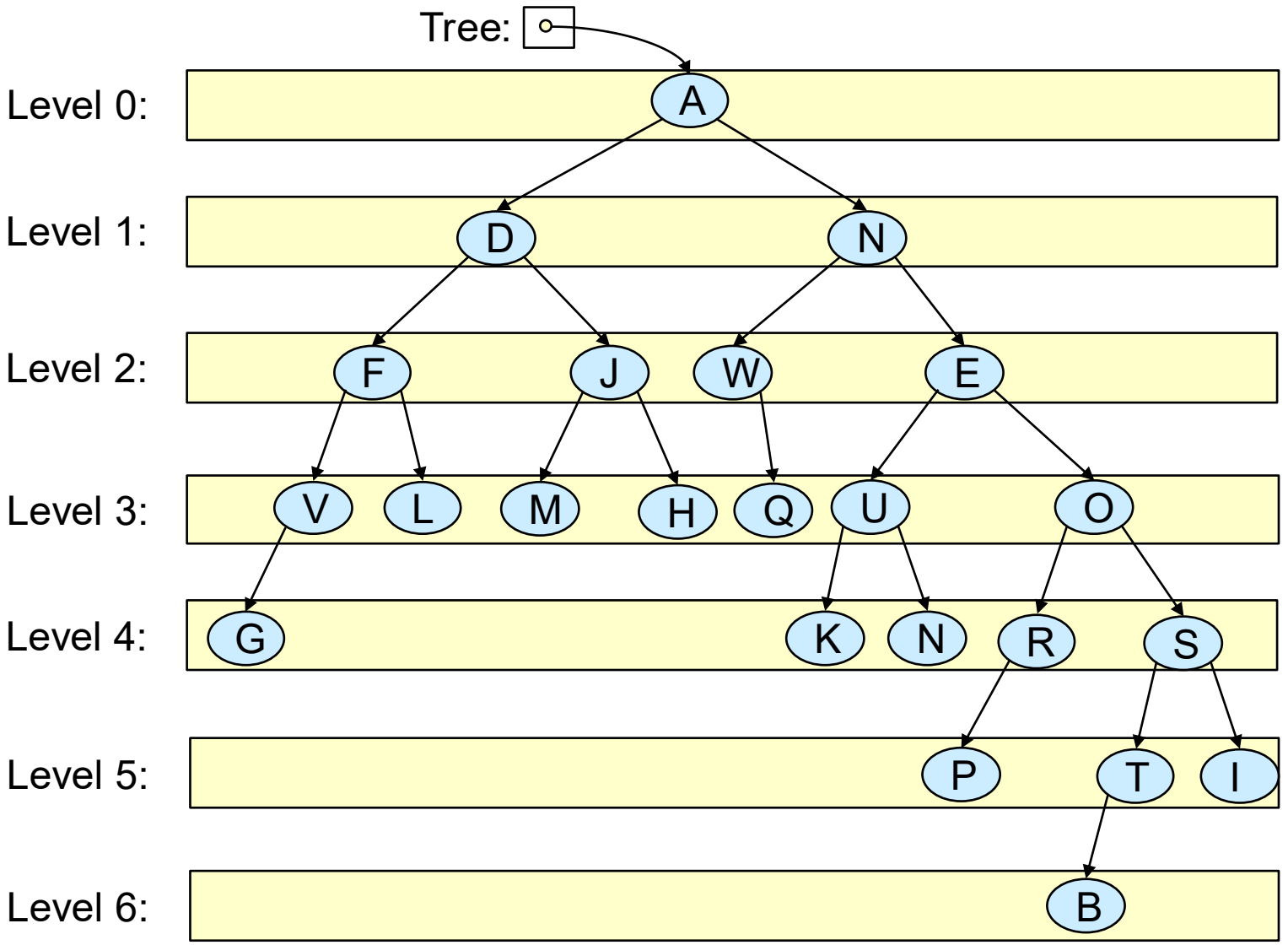
Stack one

Result: Print from stack2

4,5,2,3,1

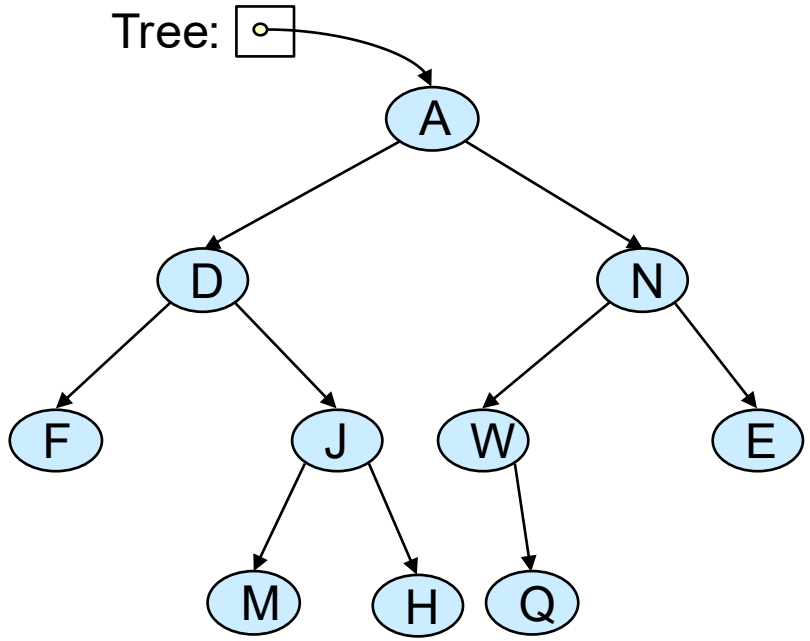
Breadth First Traversal

- Traversing nodes by level = "breadth first"
- Level-order traversal of a tree visits the nodes level-by-level, starting with level 0 (i.e. the root), then level 1, then level 2, etc. and within each level from left to right



Breadth first

- Use a **queue** to store the nodes that need to be worked on



Example 1: Breadth first

```
public class Person {
  String name;
  int yoB; // Year of Birth
  Person left;
  Person right;

  public Person(String name, int yoB) {
            this.name = name;
            this.yoB = yoB;
            this.left = null;
            this.right = null;
  }

  public Person getLeft() {
            return left;
  }

  public Person getRight() {
            return right;
  }

  public int getYoB(){
            return yoB;
  }
```

Breadth first: Recursive

```
void levelOrderRec(Person root, int level, ArrayList<ArrayList<Integer>> tree){
  //Base
  if (root == null)
                return;

  if (tree.size() <= level) //add level if needed
                tree.add(new ArrayList<>());


   tree.get(level).add(root.yoB); //Add current person node's data to its corresponding level

   levelOrderRec(root.left, level + 1, tree); // Recursive for left and right children
   levelOrderRec(root.right, level + 1, tree);

}

ArrayList<ArrayList<Integer>> levelOrder(Person root) { // Function to perform level order traversal
   // Stores the result level by level
   ArrayList<ArrayList<Integer>> tree = new ArrayList<>();
   levelOrderRec(root, 0, tree);
   return tree;
}
```

Breadth first: Queue

- Use a **queue** to store the nodes that need to be worked on

- A queue is FIFO = First In First Out
- Nodes are processed level by level.

Breadth first: Queue (Iterative)

```
//O(n) time and O(n) space
public static ArrayList<ArrayList<Integer>> levelOrder1(Person root){
  if (root == null)
            return new ArrayList<>();


  Queue<Person> q = new LinkedList<>();            // Create an empty queue for level order traversal
  ArrayList<ArrayList<Integer>> list = new ArrayList<>();

  q.offer(root);            // Enqueue Root
  int currLevel = 0;


  while (!q.isEmpty()) {
            int len = q.size();
            list.add(new ArrayList<>());

            for (int i = 0; i < len; i++) {
              // Add front of queue and remove it from queue
              Person person = q.poll();
              list.get(currLevel).add(person.yoB);

              if (person.left != null)    // Enqueue left child
              q.offer(person.left);

              if (person.right != null)  // Enqueue right child

              q.offer(person.right);
            }
            currLevel++;
  }
  return list;
}
```

Example: Breadth First

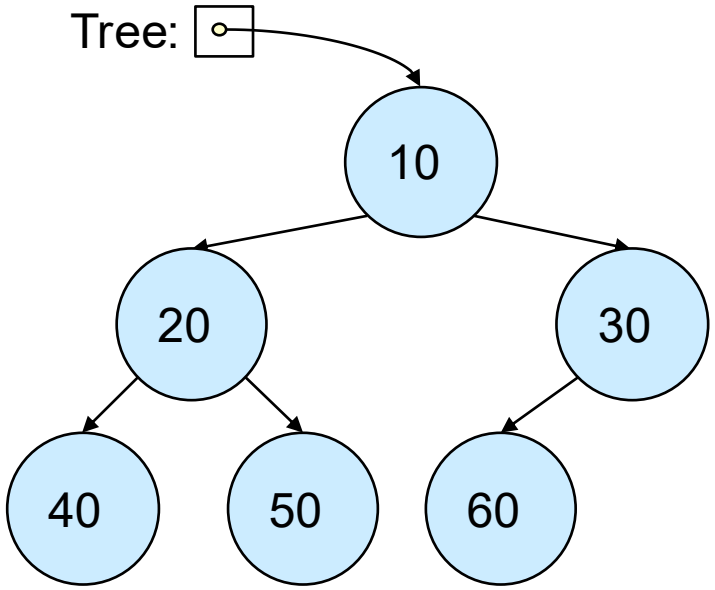
Structure of tree

Suppose:

```
[
 [10],
 [20,30],
 [40,50,60]
]
```

Meaning:

- tree.get(0) → first level
- tree.get(1) → second level
- tree.get(2) → third level



Example: Breadth First

Setup

```
Queue<Person> q = new LinkedList<>();
```

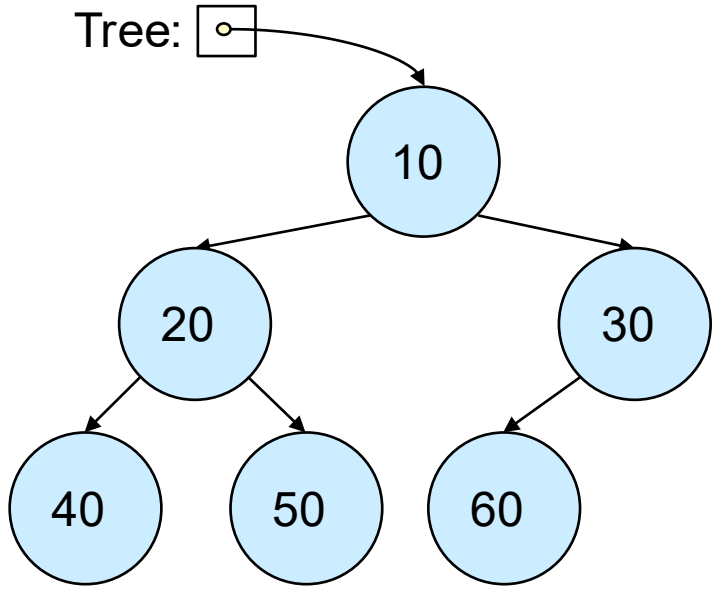
Add Root

```
q.offer(root);
```

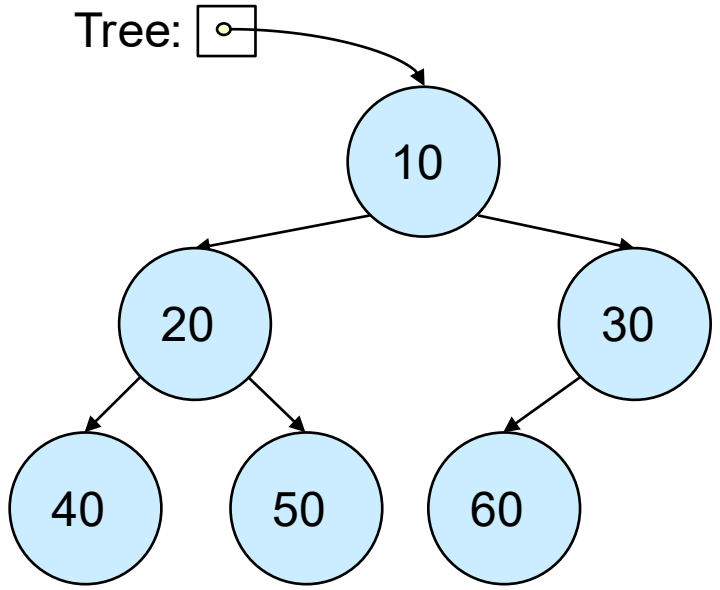
Queue = [10]

Main loop: till all nodes are processed

```
while (!q.isEmpty())
```



Example: Breadth First



Queue = [10]

Process 10.
Add children:

Queue = [20,30] len = 2

Process 20, 30
Add children:

Queue = [40,50, 60] len = 3

Breadth First

Recursive	**Queue**
Uses DFS internally	Uses BFS naturally
Tracks level using parameter	Tracks level using queue size
Elegant	More standard
Uses recursion stack	Uses explicit queue