Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

Traversing a binary tree

A/Prof. Pawel Dmochowski

School of Engineering and Computer Science

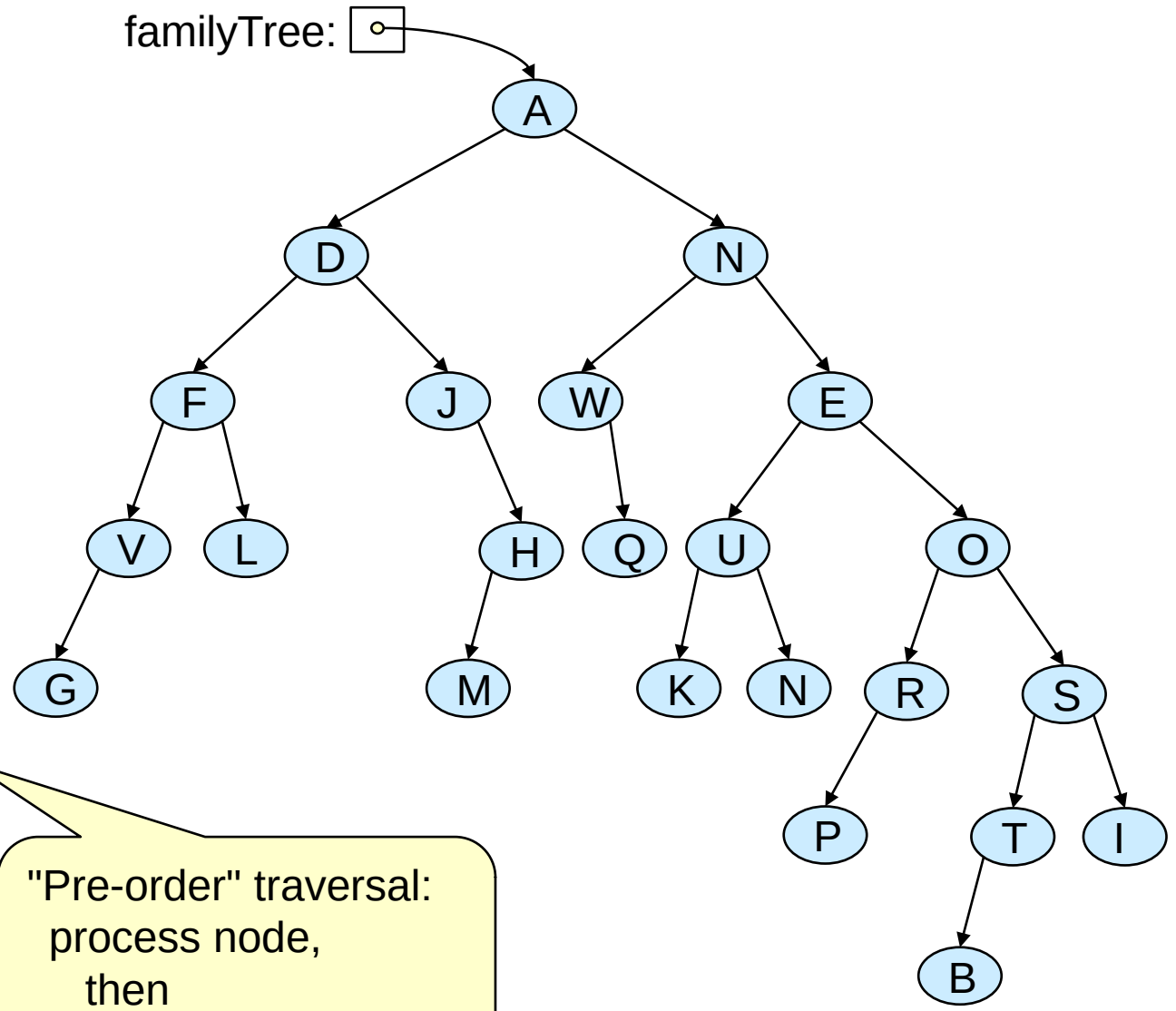
Victoria University of Wellington

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.

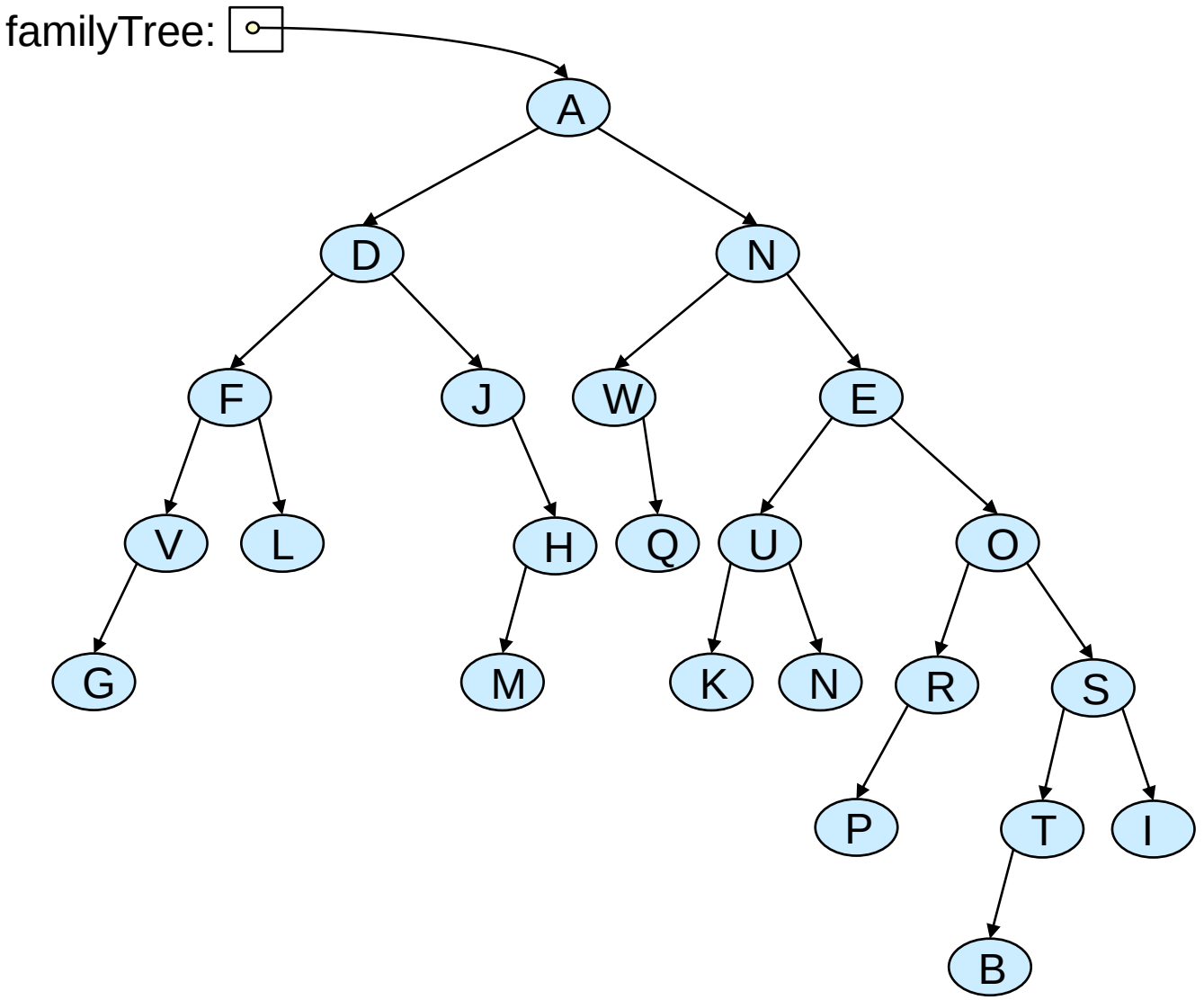
More traversals: depth-first, post-order

- "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);
```

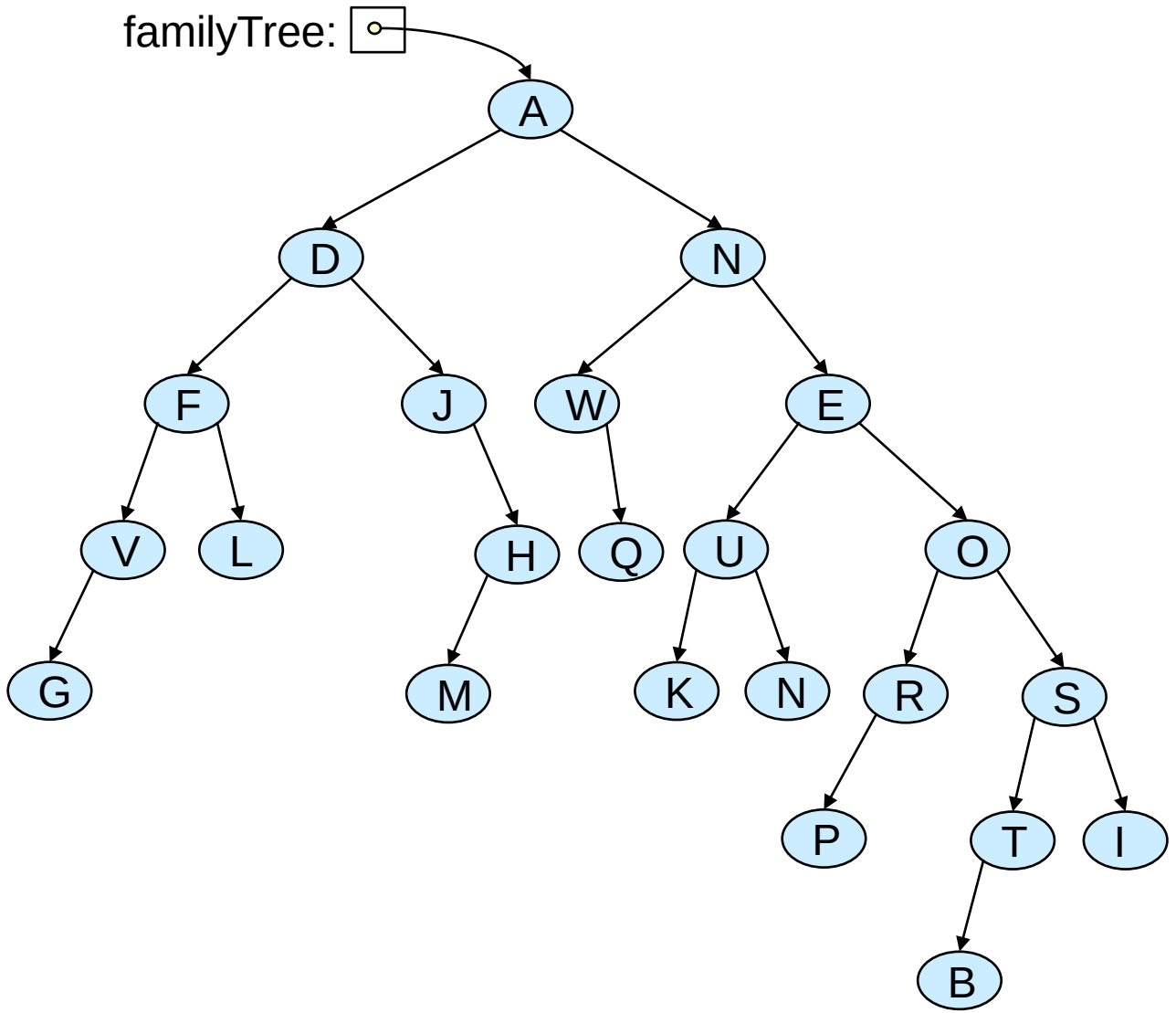


Another traversal: depth-first, in-order

- 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);
```

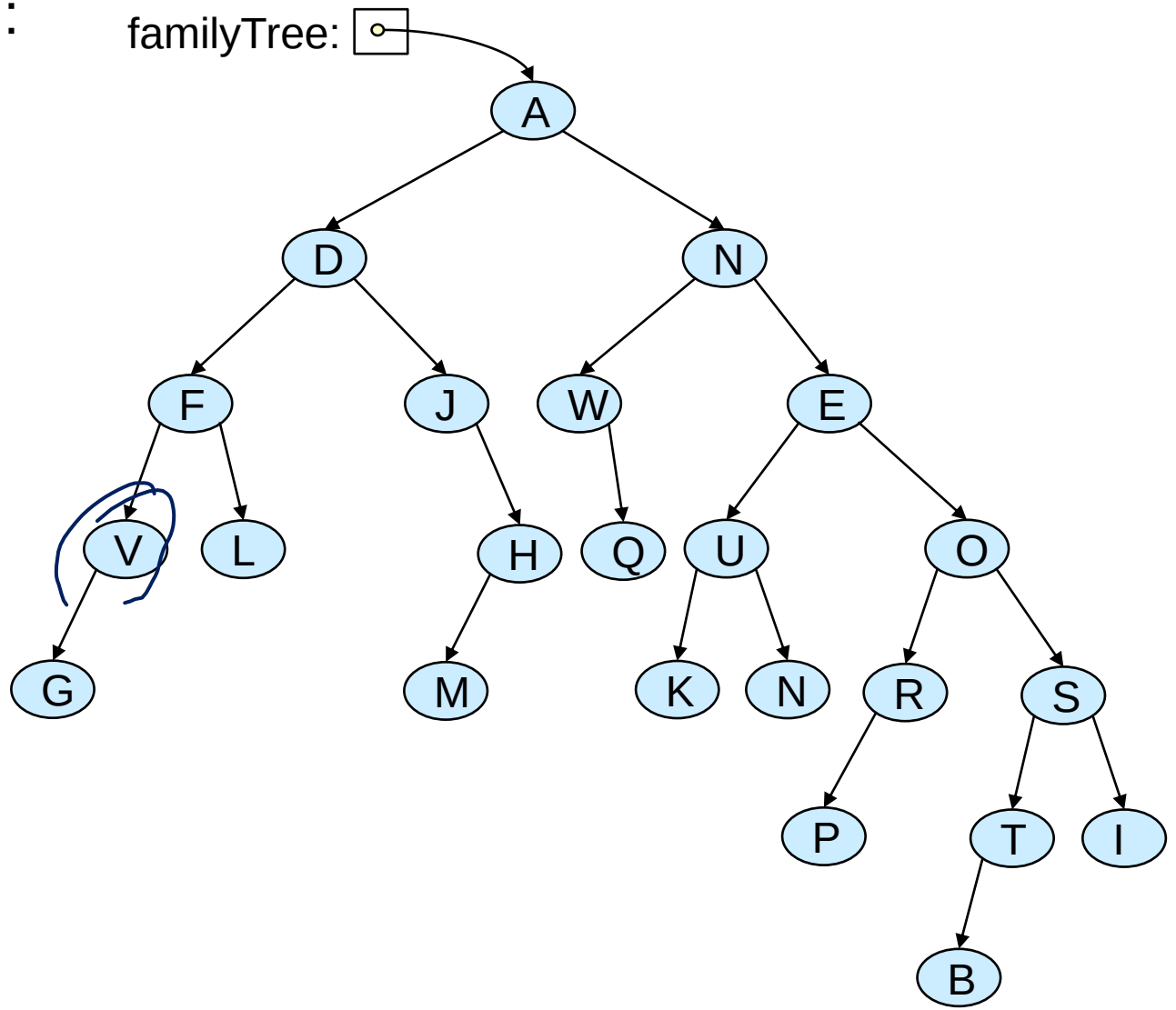


Traversing with an extra parameter

- 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);
```

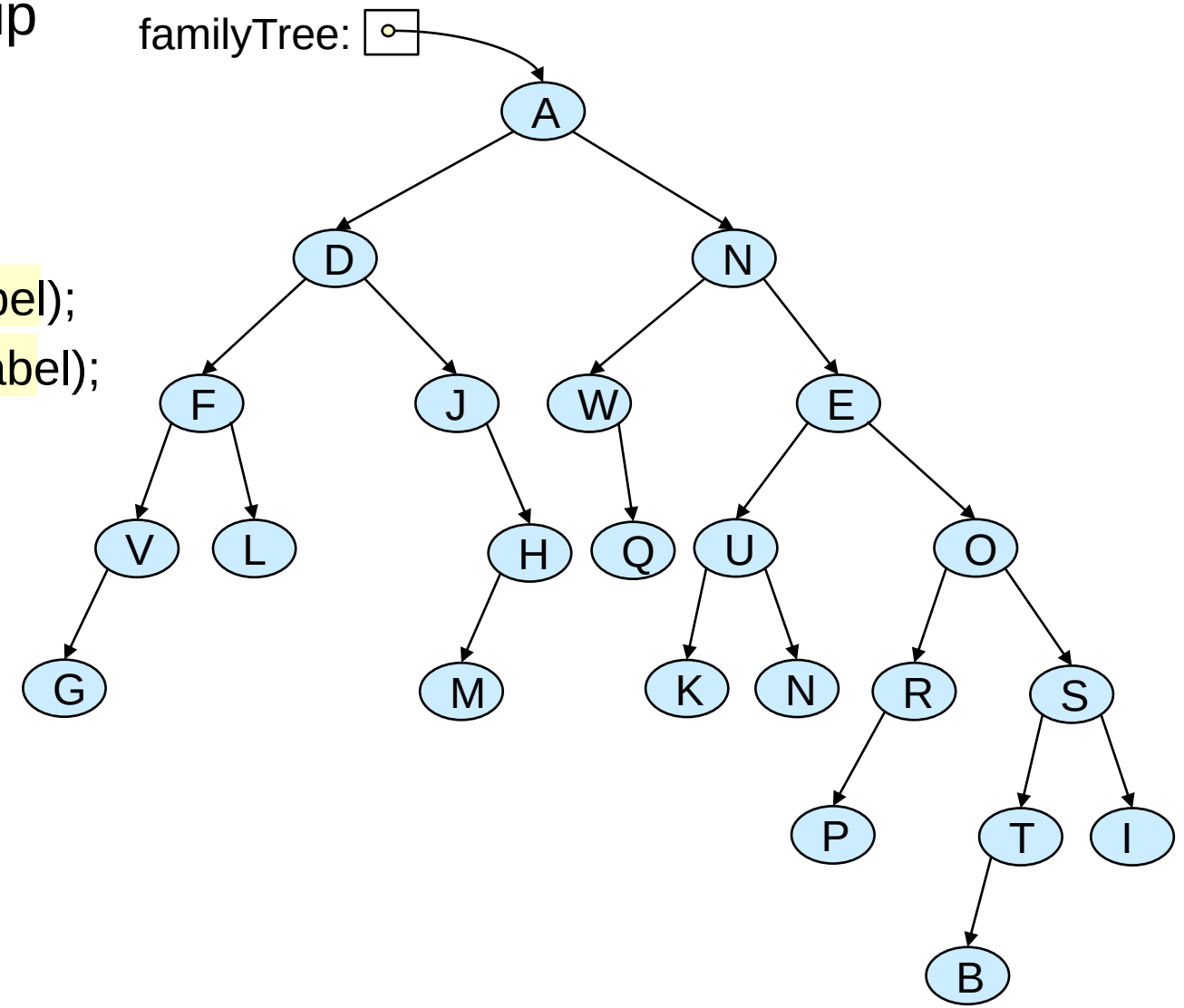
1 : A
2 : D
3 : F
4 : V
5 : G



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”);
```



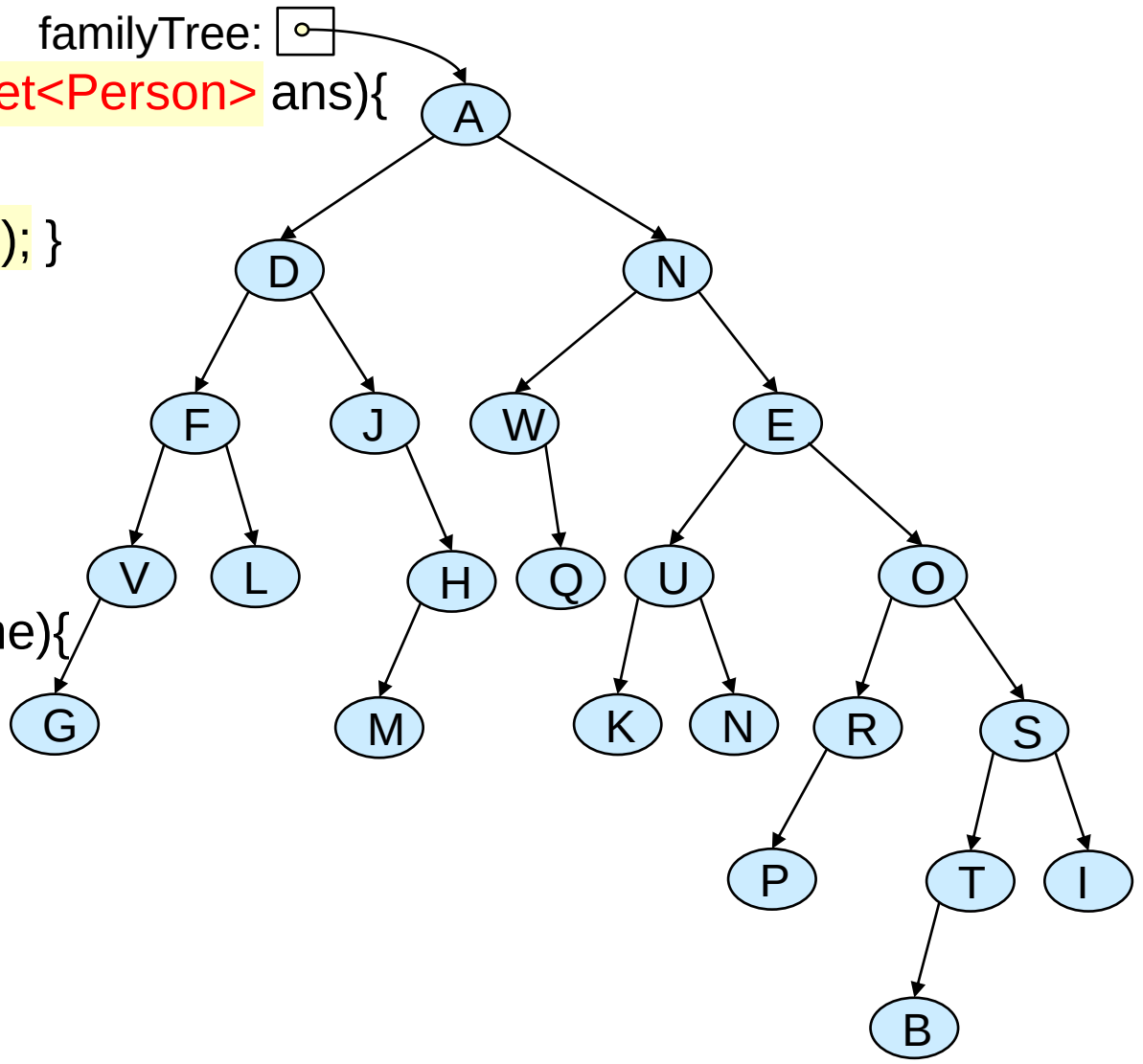
Traversing and collecting an answer

- 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.

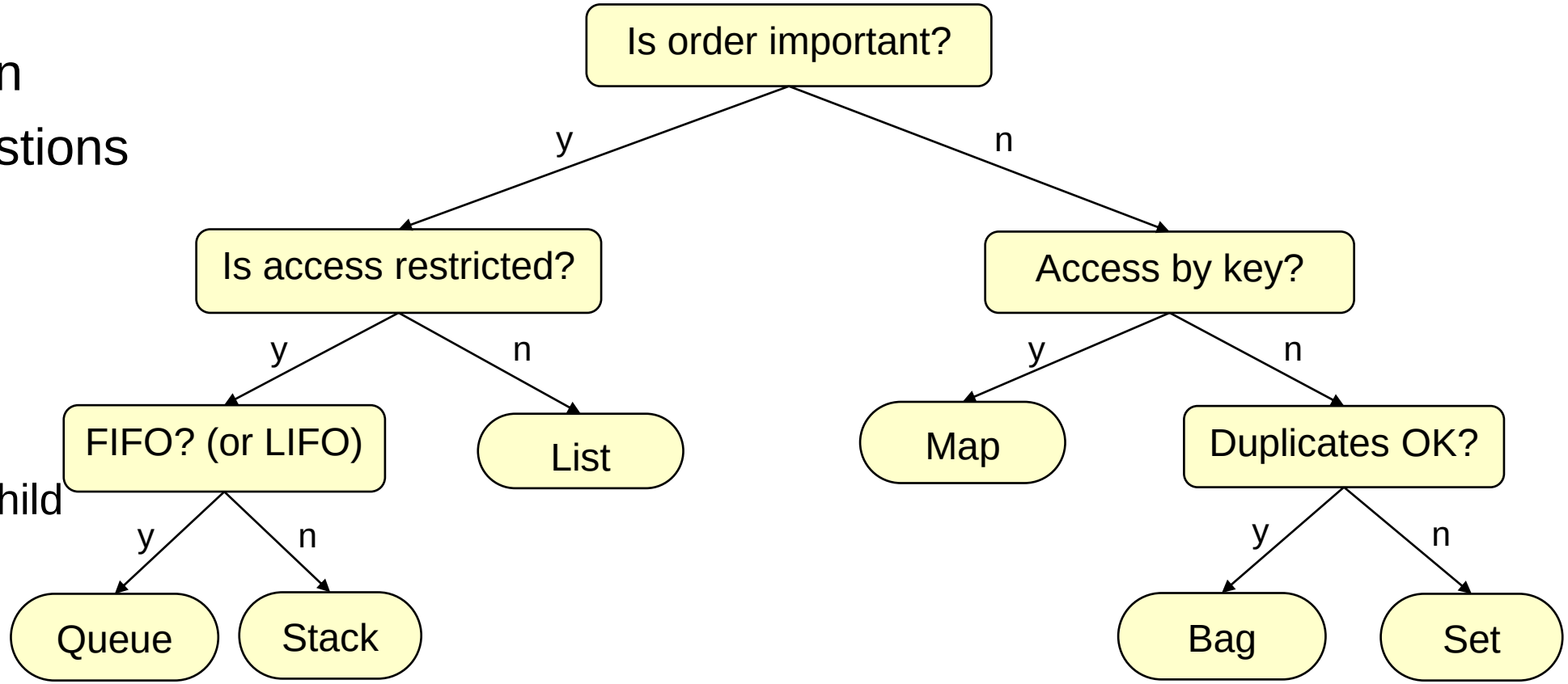
- For Binary trees (two children per node)
  - pre-order: visit parent node then traverse child subtrees,
  - post-order: traverse child subtrees then visit parent node
  - in-order: traverse one child subtree
then visit parent node
then traverse other subtree
    - (binary trees only)

Decision Trees

- Ask questions until get to a decision node (leaf)

- Path depends on answers to questions in nodes

- Loop down tree
  - Ask question
  - Choose y or n child

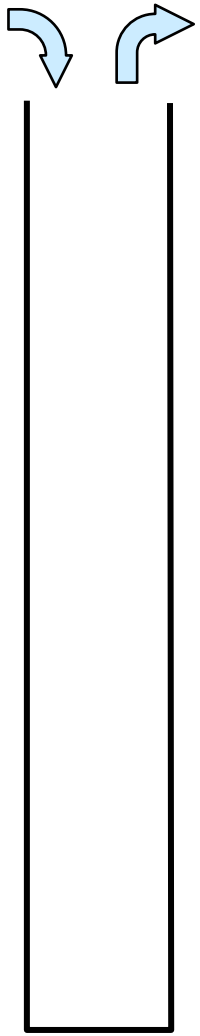


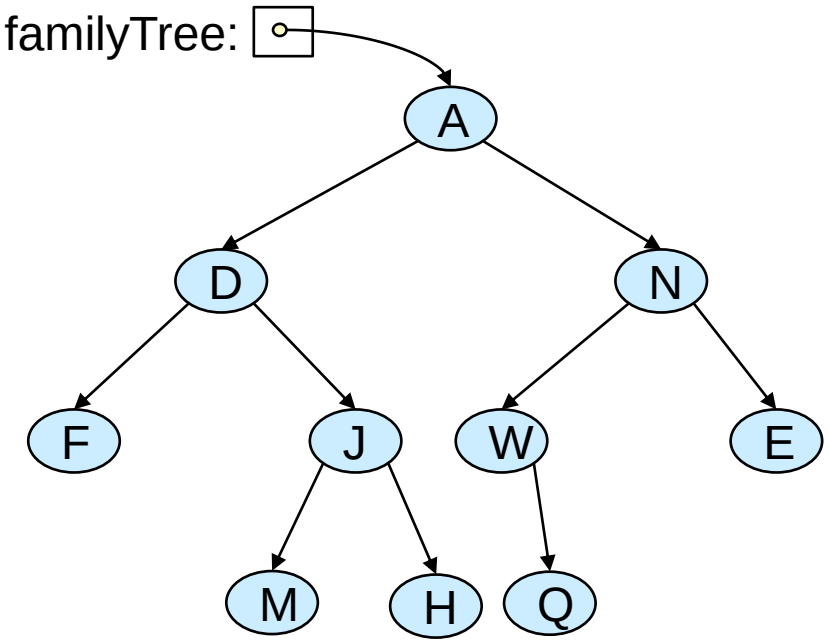
- Extending tree
  - If answer is wrong, turn into a question ode
  - Add child nodes (old and new answers)

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());
        }
    }
}
```



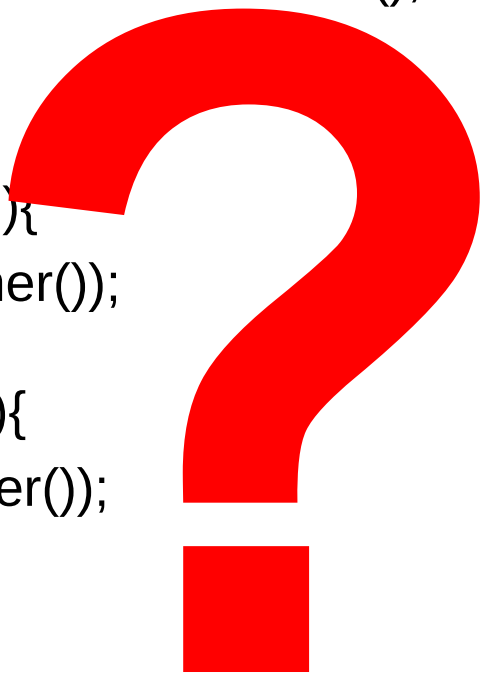


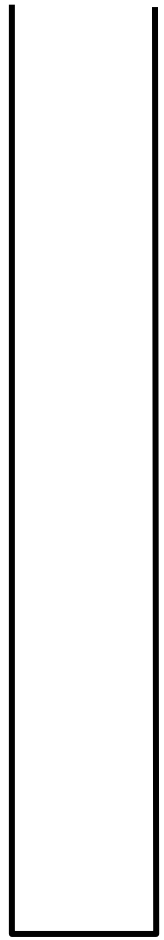
A D F J H M N W Q E

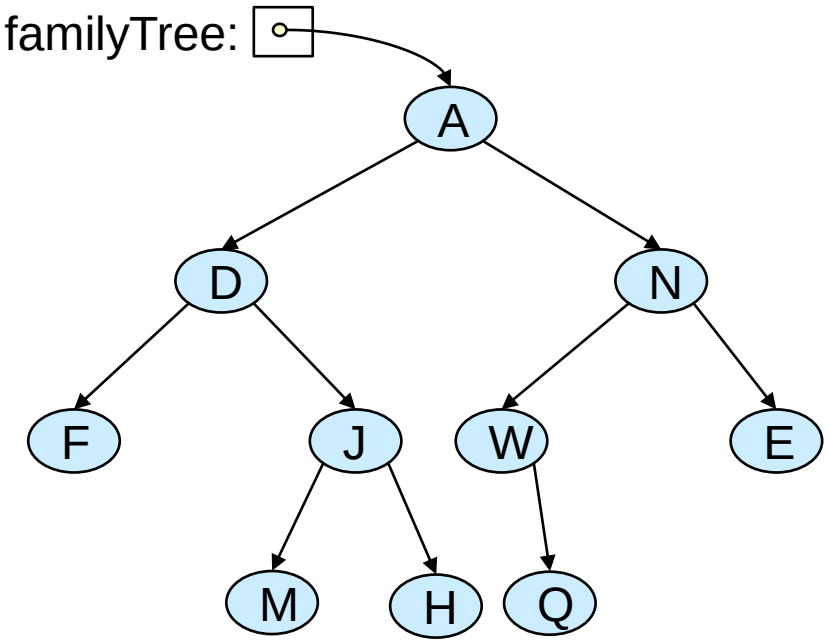
Depth first traversal without recursion

- How do we do post-order?

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



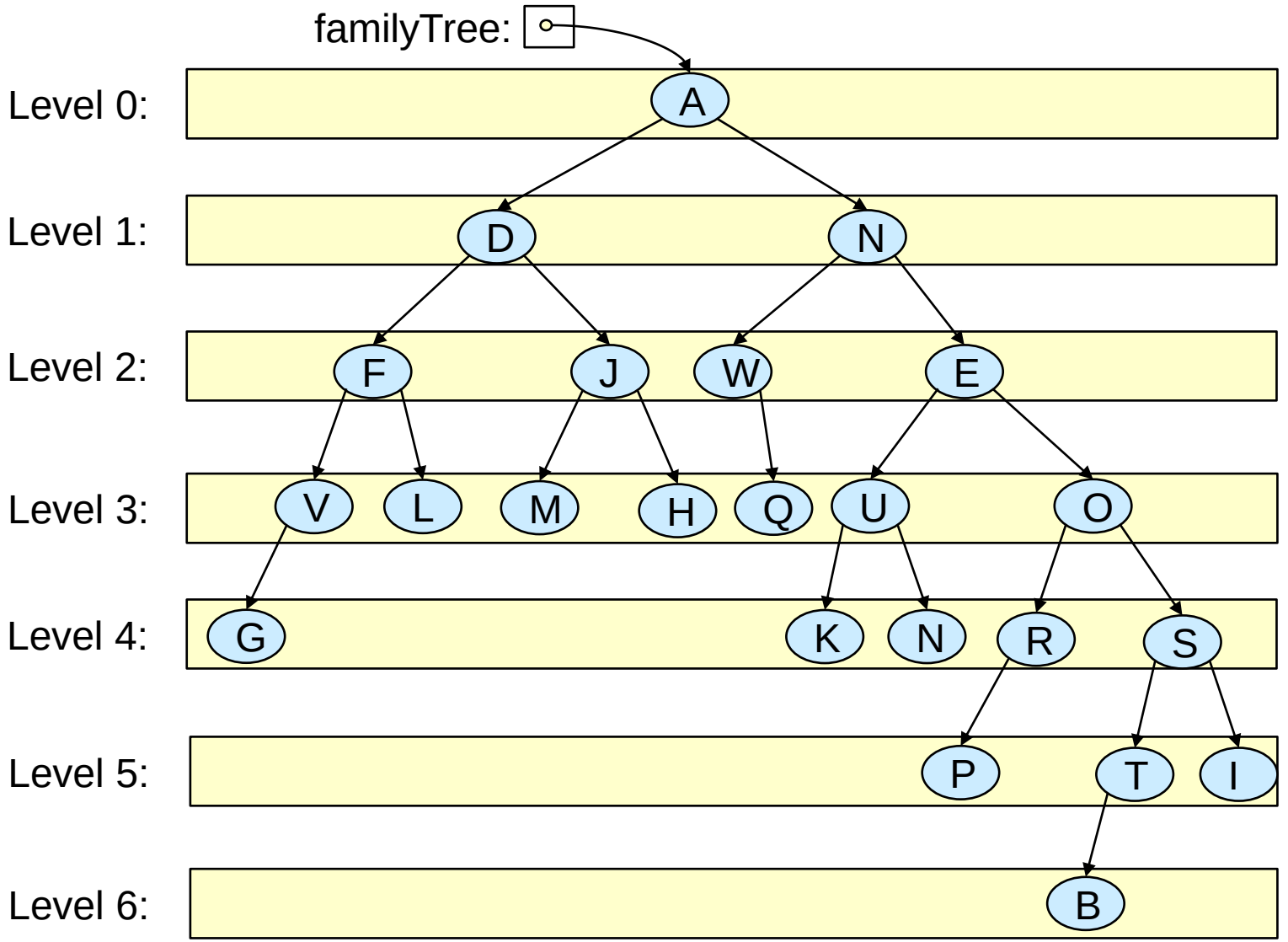




Breadth First Traversal

- Traversing nodes by level = "breadth first"

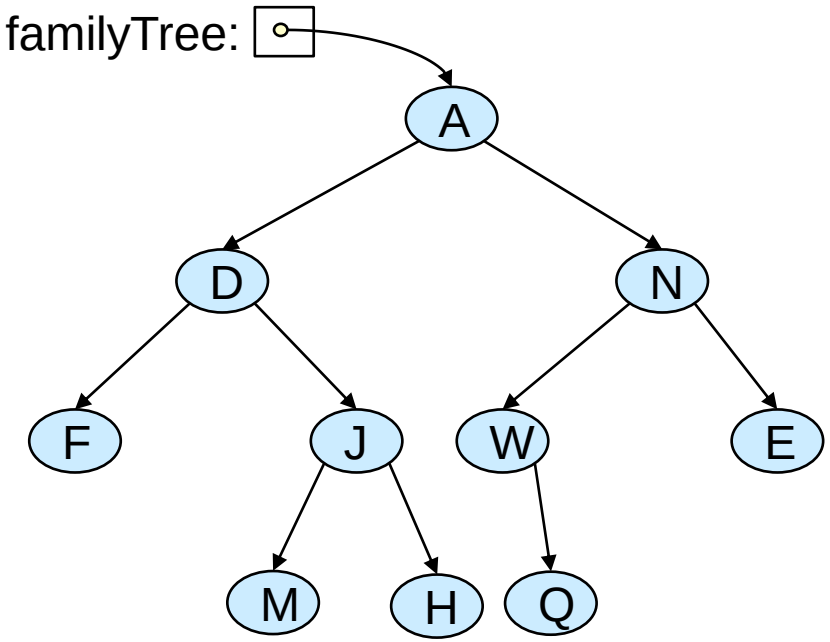
- Level-order traversal of a tree visits the nodes level-by-level, starting with level 0 (i.e. the root), then level 1, then level 2, etc. and within each level from left to right



Breadth first

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

```
public void breadthFirstTraversal (Person root){
    Queue<Person> todo = new ArrayDeque<Person>();
    todo.offer(root);
    while ( ! todo.isEmpty() ){
        Person p = todo.poll();
        UI.println(p);
        if ( p.getMother() != null ){
            todo.offer(p.getMother());
        }
        if ( p.getFather() != null ){
            todo.offer(p.getFather());
    }
}
```



Traversing and returning

- Collecting up nodes in a list/set to return:
  - Straightforward using the iterative (Stack or Queue based) traversal:

```
/** Find all Persons in tree born before a given year */
  public Set<Person> dfFindOldStack(Person root, int year){
    Set<Person> ans = new HashSet<Person>();
    Stack<Person> todo = new Stack<Person>();
    todo.push(root);
    while ( ! todo.isEmpty() ){
        Person p = todo.pop();
        if (p.getYoB()< year)      { ans.add(p); }
        if ( p.getMother() != null ){ todo.push(p.getMother()); }
        if ( p.getFather() != null ) { todo.push(p.getFather());  }
    }
    return ans;
  }
```

Traversing and returning

- Collecting up nodes in a list/set to return:
  - In recursive traversal, pass in List/Set; method just adds values to List/Set;
  - No need to return list/set from recursive calls

```
/** Find all Persons in tree born before a given year */
public Set<Person> dfFindOldRec(Person p, int year){
        Set<Person> setOfOld = new HashSet<Person>();
        dfFindOldRecHelper(p, year, setOfOld);
        return setOfOld;
}
public void  dfFindOldRecHelper(Person p, int year, Set<Person> setOfOld){
        if (p!=null){
            if (p.getYoB()< year) {setOfOld.add(p); }
            dfFindOldRecHelper(p.getFather(), year, setOfOld);
            dfFindOldRecHelper(p.getMother(), year, setOfOld);
        }
}
```

Traversing and returning

- Finding a single node or value to return:
  - Straightforward using the iterative (Stack or Queue based) traversal:

```
/** Find a Person in tree with a given name */
  public Person dfFindNameStack(Person root, String name){
     Stack<Person> todo = new Stack<Person>();
     todo.push(root);
     while ( ! todo.isEmpty() ){
        Person p = todo.pop();
        if (p.getName().equals(name))  { return p; }
        if ( p.getMother() != null ){ todo.push(p.getMother()); }
        if ( p.getFather() != null ) { todo.push(p.getFather());  }
     }
    return null;
  }
```

Traversing and returning

- Finding a single node or value to return:
  - In recursive traversal, must pass back the answer, all the way up the tree

```
/** Find a Person in tree with a given name */
public Person dfFindNameRec(Person p, String name){
    if (p==null)                          { return null;  }
    if (p.getName().equals(name)) { return p; }
    Person ans = dfFindNameRec(p.getFather(), name);
    if (ans !=null)                       { return ans; }
    return  dfFindNameRec(p.getMother(), name);
}
```