Binary tree
Looping down tree

Running off the end
while(p!=null)

Stepping along a path from root.

eg: Print out maternal line:

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

Stopping at the end
while(p.getMother()!=null)

Finding a leaf node:

eg: Add next maternal ancestor:

familyTree:

p:

```
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));
```

Add next maternal ancestor:

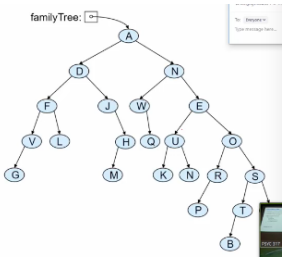
```
public Person oldestMatAnc (Person p){
    Person tmp = p;
    while (tmp.getMother()!=null){
        tmp = tmp.getMother();
    }
    return tmp;
}
:
Person p = oldestMatAnc(familyTree);
UI.println("Oldest known maternal ancestor: "+p);
String name = UI.askString("Name of her mother");
int dob =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

Traversing a tree
Depth first traversal; Pre-order
Process node, then process subtrees e.g print node before subtree (parent before child)
-ADFVGLJHMN…

- 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);
```



Loop; recursion
A>D
D>F
F>V
V>G
G>NULL breaks (no father)
V>NULL breaks (no mother)
F>L
L>NULL breaks (no mother)
F>FINISHED breaks
F>D
D>J
J>NULL breaks (no father)
J>H
H>M
M>NULL breaks

H>NULL breaks(no mother)
J>FINISHED breaks
D>FINISHED breaks
A>N
…..

Keeping track of depth

- Traversing the tree, printing generation:

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

4

A 1
D 2
F 3
v 4
G 5
L 4
J 3

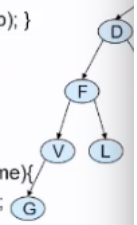
Collecting nodes in set

- Traversing the tree: find all with name.

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

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

familyTree:



String that grows with depth

- 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");
```

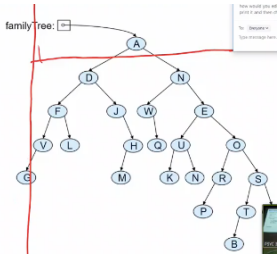
Me A
Father of me D
Father of Father of me F
….

Depth first traversal; Post-order
Process subtree, then process nodes e.g print subtree_before_node before (child before parent)
-GVLFMHJDN…..A

- "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);
```

Depth first traversal; in-order
IF we put print line statement in the middle we would get
-GVFLDJMHAWQNKUNE….
(imagine line scanning across from left to right - node on left of line will print first)



Traverse one child subtree
then visit parent node
then traverse other subtree
(only for binary)

<u>Breath First Traversal</u>
Traversing nodes by levels

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

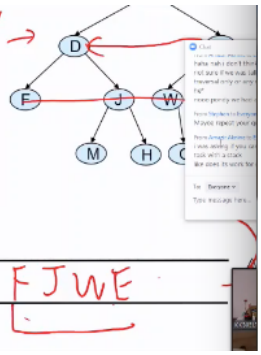
```
public void breadthFirstTraversal (Person root){
    Queue<Person> todo = new ArrayDeque<Person>();
    todo.offer(root);
    while ( ! todo.isEmpty() ){
        Person p = todo.poll();
        UI.println(p);
        if ( p.getMother() != null ){
            todo.offer(p.getMother());
        }
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

family

F

```
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());
    }
}
```

AND



> can change to priority queue to produce more complex traversal outcome

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 from recursive calls

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

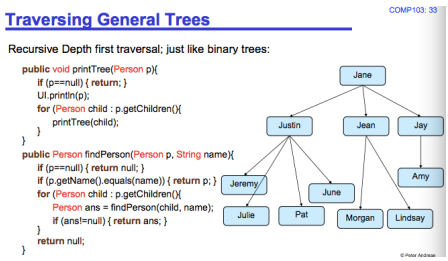
- Finding a single node or value to return:
  - In recursive traversal, must pass back the answer, all the way up the tree

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

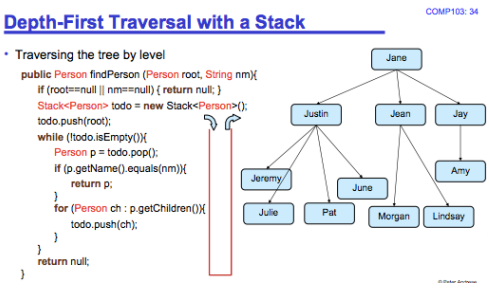
General Trees

Binary trees: at most two child nodes
Termary trees: at most three child nodes
General trees: any number of child nodes

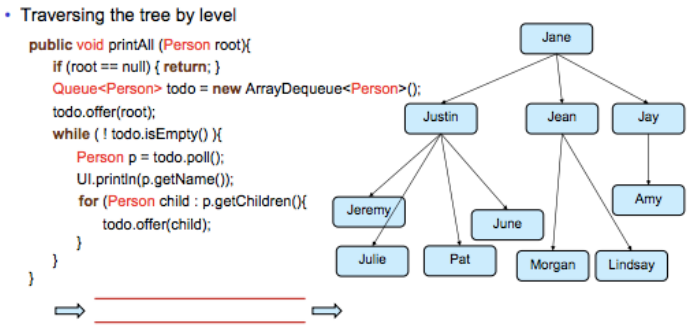
Depth first general tree



Depth-First Traversal with a Stack



Breadth-First Traversal



Adding to a node in a Tree

- Depth first traversal – need to exit out of ALL levels if add the new child.
- Return a boolean to signal success.

```
public boolean addPerson(Person root, Person newChild, String parentName){
    if (root==null) { return false; }
    if (root.getName().equals(parentName)) {
        root.addChild(newChild);
        return true;
    }
    for (Person ch : root.getChildren(){
        if ( addPerson(ch, newChild, parentName) ) {
            return true;
        }
    }
    return false;
}
```

If succeeded here, return success

If succeeded in one subtree, return immediately

Removing a node from a Tree

- DF traversal, "look ahead", remove after loop, return success when removed

```
public boolean removePerson(Person tree, String name){
    if (tree == null){ return false; }
    Person chToRemove = null;
    for (Person ch : tree.getChildren()){
        if (ch.getName().equals(name)){  chToRemove = ch;  break;  }
        else {
            if (removePerson(ch, target)) { return true; }
        }
    }
    if (chToRemove==null){ return false; }
    tree.removeChild(chToRemove);
    return true;
}
```

Iterable and Iterator

To be able to iterate along an object using foreach loop, the object must be iterable:
The object class must implement iterable<??> and have a

public Iterator<??> iterator{..} //method which returns an iterator object

An Iterator must have a

public boolean hasNext()

Method, and a

public ??? next(){..}

Method

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

Graphs

-graphs are like trees:

Nodes and links (edges); (trees are a special kind of graphs)

-Nodes have neighbours

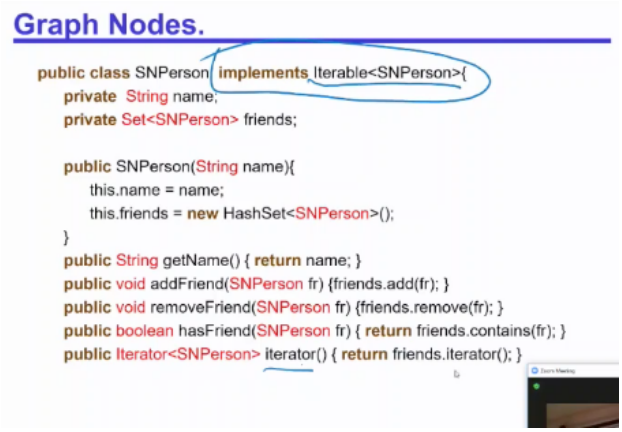
rather than children

-Graphs don't have a "root"

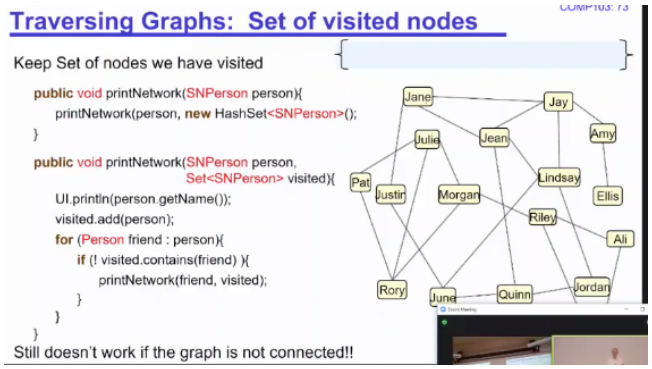
-Graphs may not be connected

-Can traverse a graph starting at node but graphs have cycles

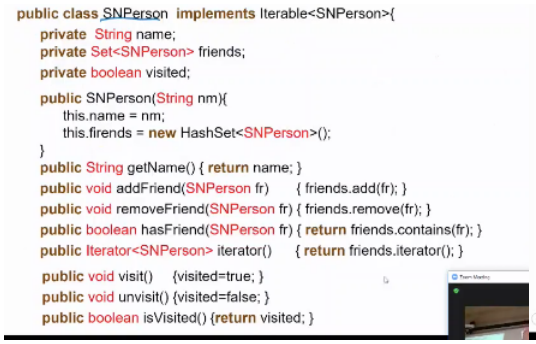
-Lots of varieties of graphs

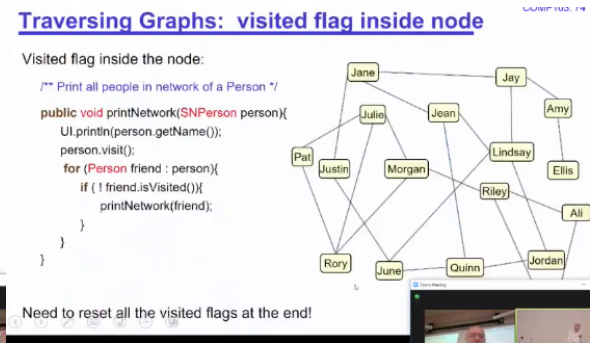


Using set



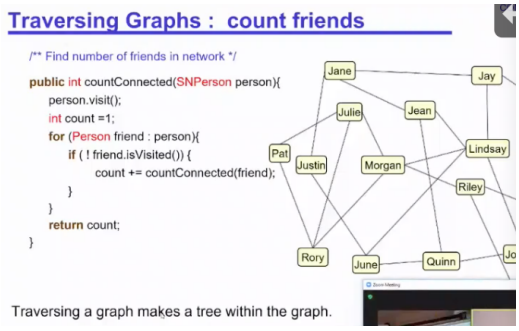
Using variable



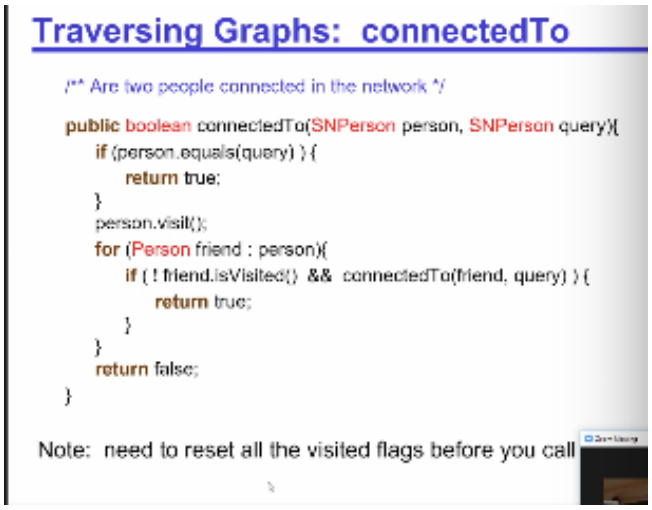


>problem needs to reset: this is simple if I have a list of nodes

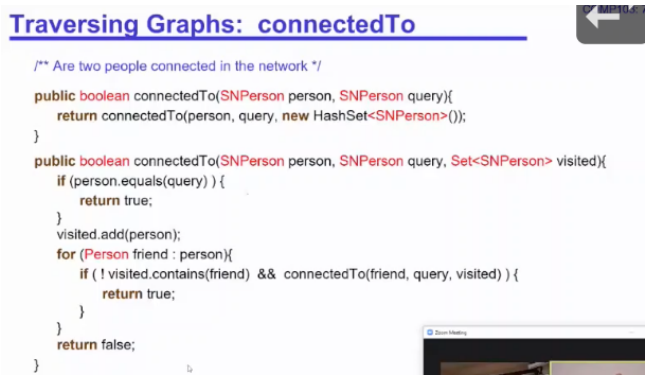
Find number of connections



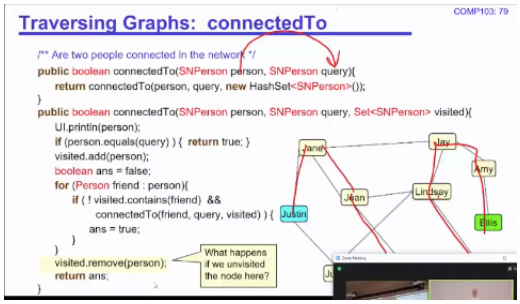
Are two people connected in the network using flag



Are two people connected in the network using set



Find all possible paths



>loops until find goal or deadens then resets and repeats