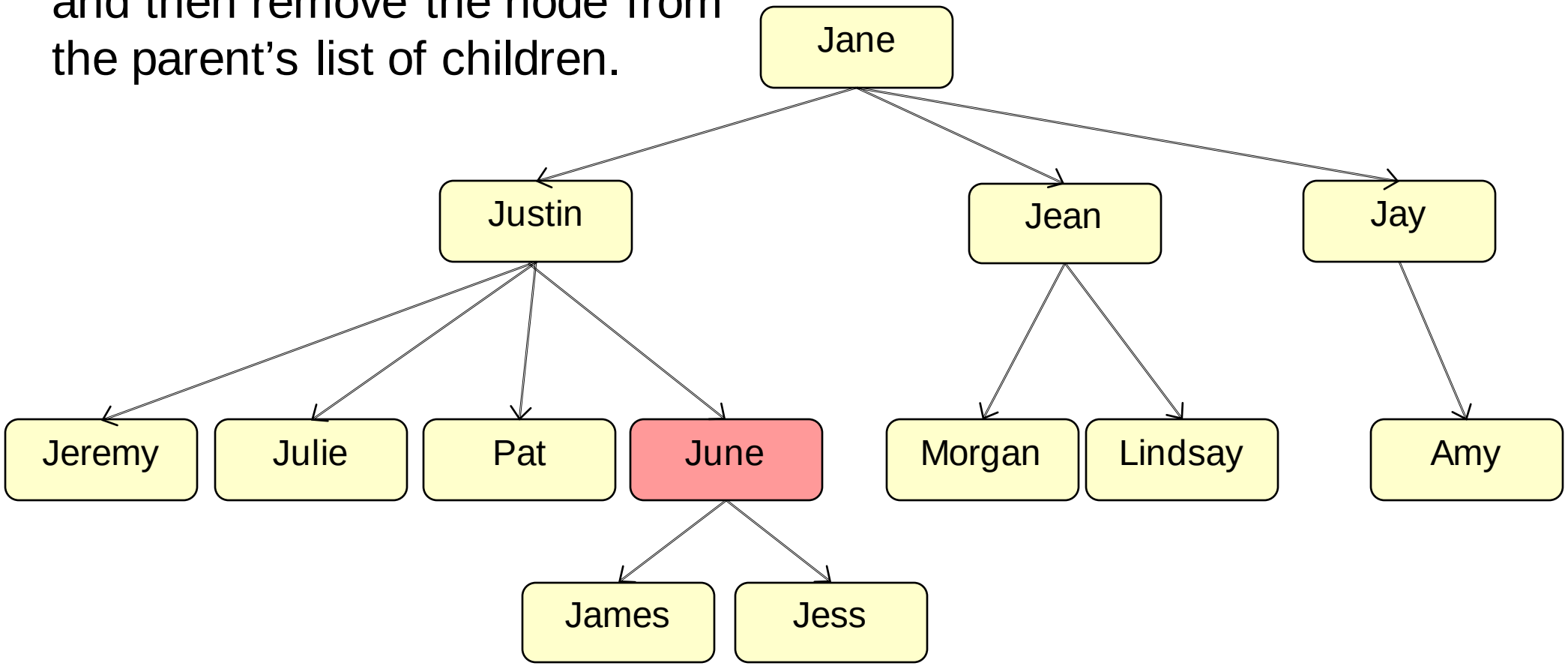
Removing a node from a Tree

- If you find a node, how do you remove it?

- Have to find the node's parent, and then remove the node from the parent's list of children.



Removing a node (and subtree) from a Tree

- Depth first traversal with “look ahead”

```
public void removePerson(Person tree, String name){
    if (tree ==null) { return; }
    for (Person child : tree.getChildren(){
        if (child.getName().equals(name)) {
            tree.removeChild(child);
            return;
        }
        else {
            removePerson(child, name);
        }
    }
}
```

How many nodes might it remove?

What if the person to remove is at the root of the tree

Not allowed to remove child from inside the foreach loop!!

Removing a node (and subtree) from a Tree

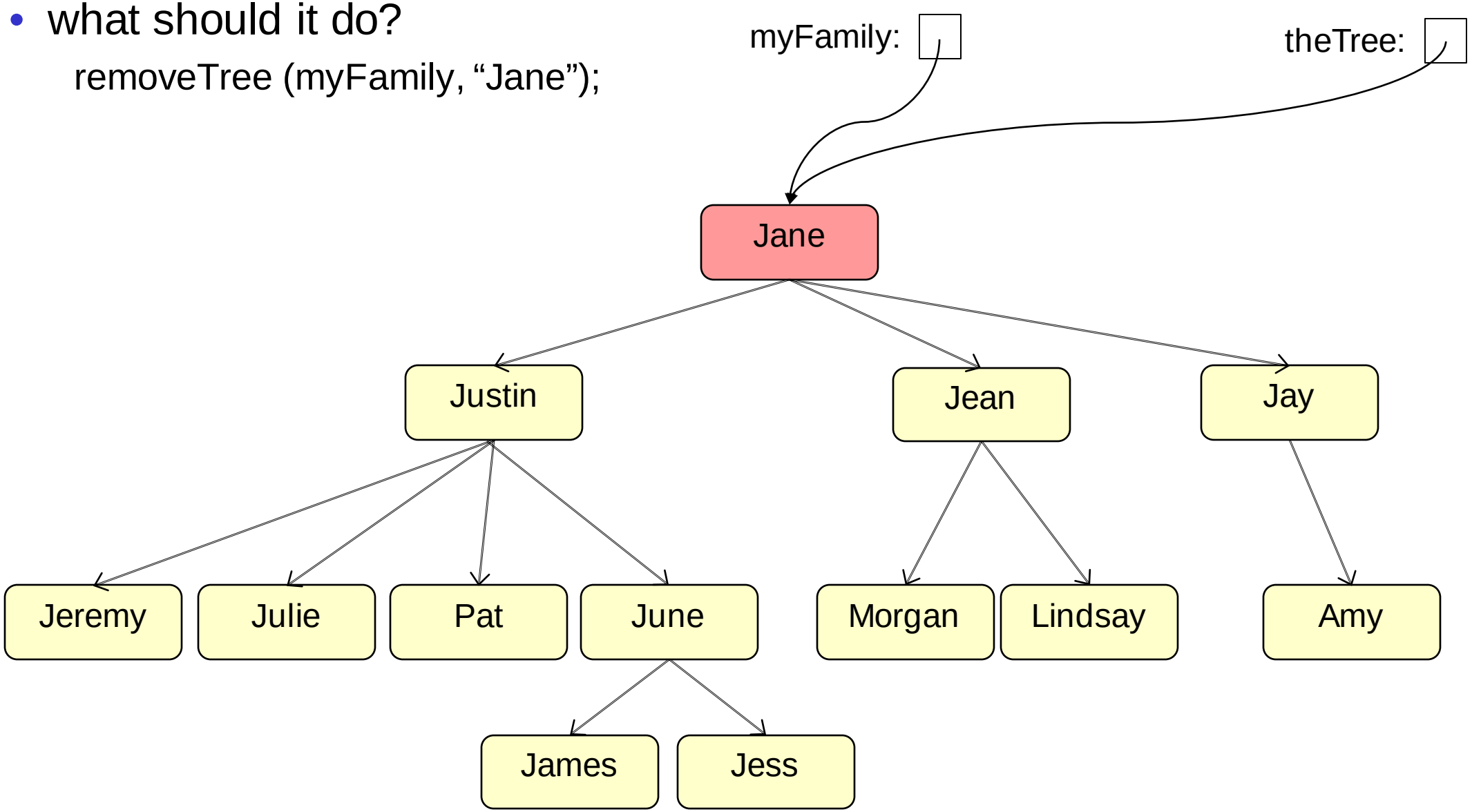
- DF traversal, "look ahead", remove after loop, return success when removed

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

Removing the root node from a Tree

- what should it do?

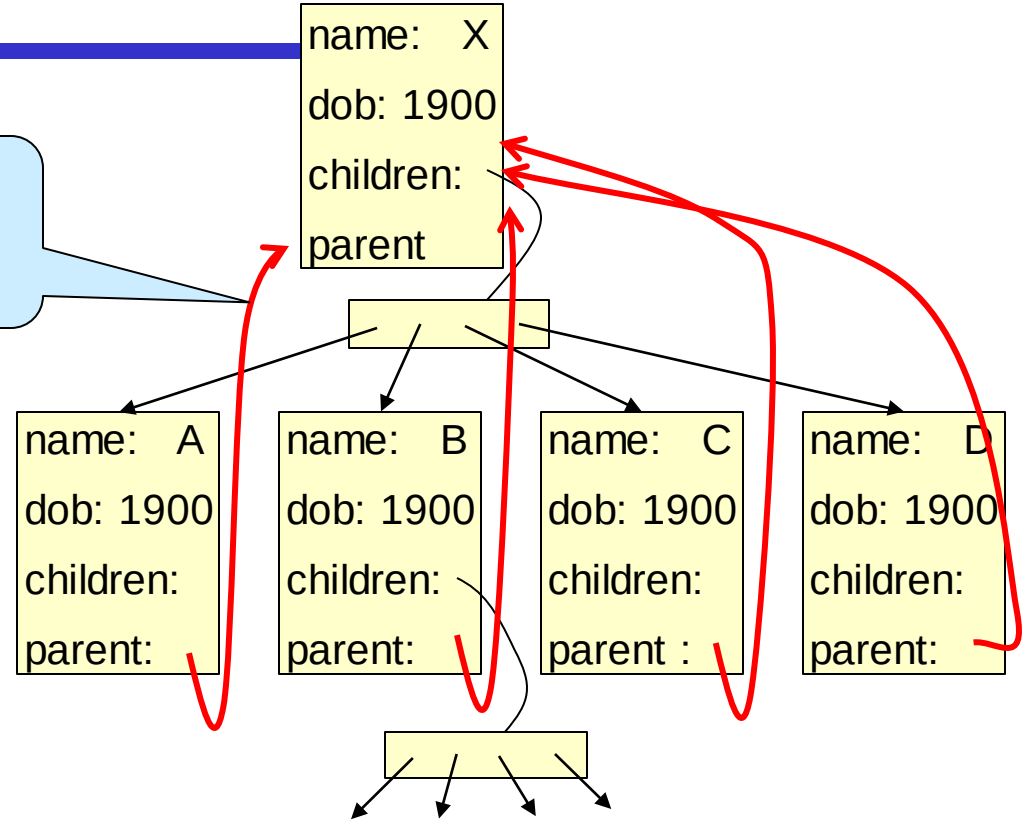
  removeTree (myFamily, “Jane”);



General Trees with parent links

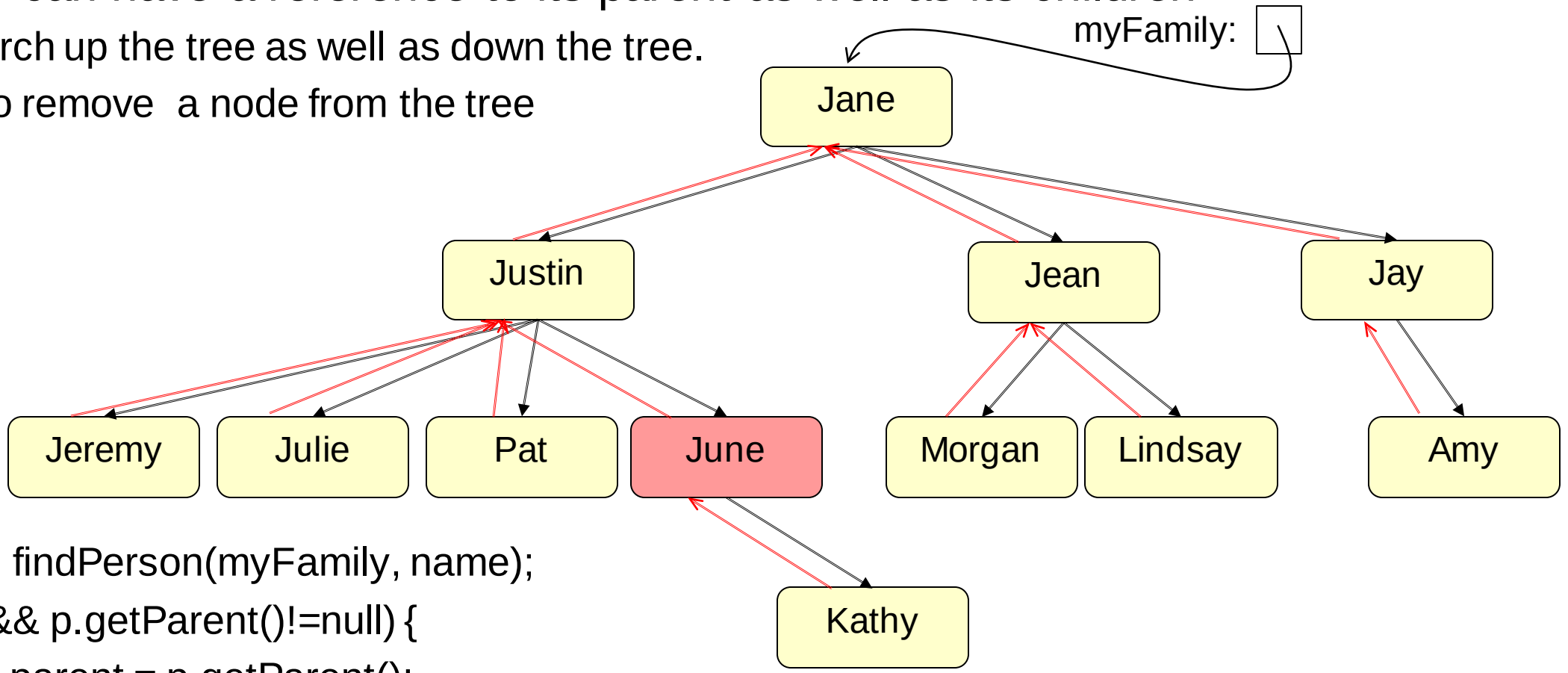
```
public class Person {
    private String name;
    private int dob;
    private Set<Person> children;
    private Person parent;

        :

    public Person getParent(){ return parent; }
    public void setParent(Person p){parent = p; }

    public Person getChildren(){ return Collections.unmodifiableSet(children);}
    public void addChild(Person ch){ children.add(ch); }
    public void removeChild(Person ch){ children.remove(ch); }
}
```

Parent links make it easy to get the parent of a node



General Trees with parent links

- Each node can have a reference to its parent as well as its children
  - ⇒ can search up the tree as well as down the tree.
  - ⇒ easier to remove a node from the tree




```
Person p = findPerson(myFamily, name);
if (p!=null && p.getParent()!=null) {
    Person parent = p.getParent();
    parent.removeChild(p);
    p.setParent(null);
}
```

Following parent links: OrganisationChart

To determine horizontal placement of Position node:

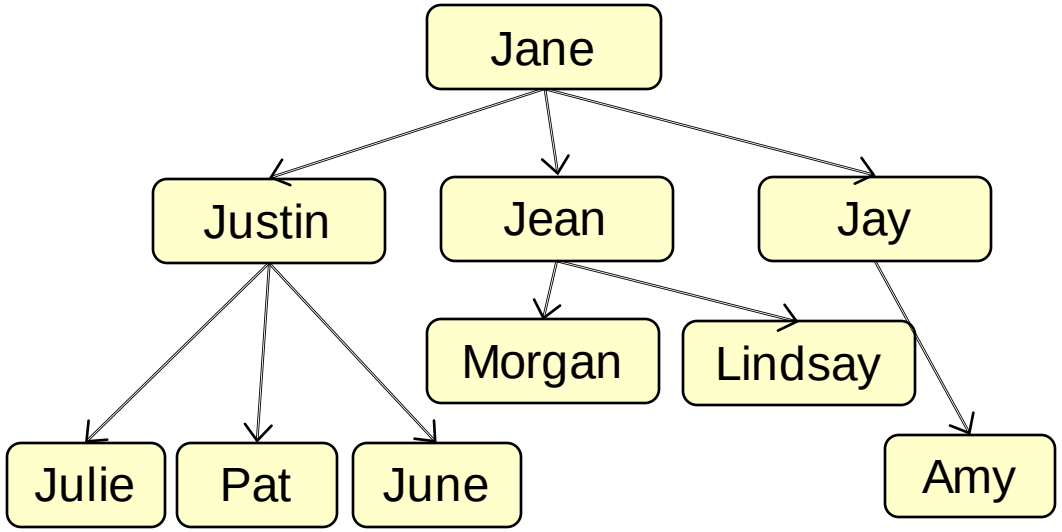
- Position contains horizontal **offset** from parent (manager)
  Must find parent's position and then add the offset:

(in Position class)

```
public int getX(){
    if (this.manager==null)  {                    // this is the root of the tree.
        return  BASE_X + this.offset;
    }
    else {                                        // relative to position of parent.
        return this.manager.getX() + this.offset;
    }
}
```

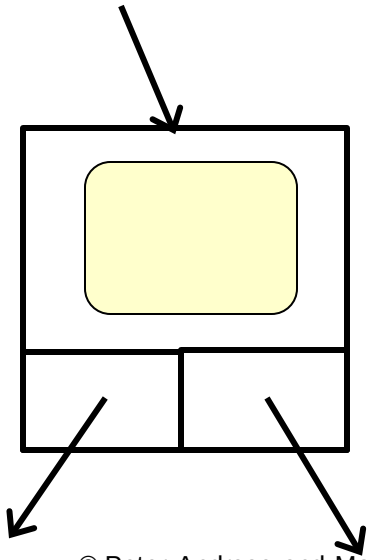
Trees of Data *vs* Trees containing Data

- 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.

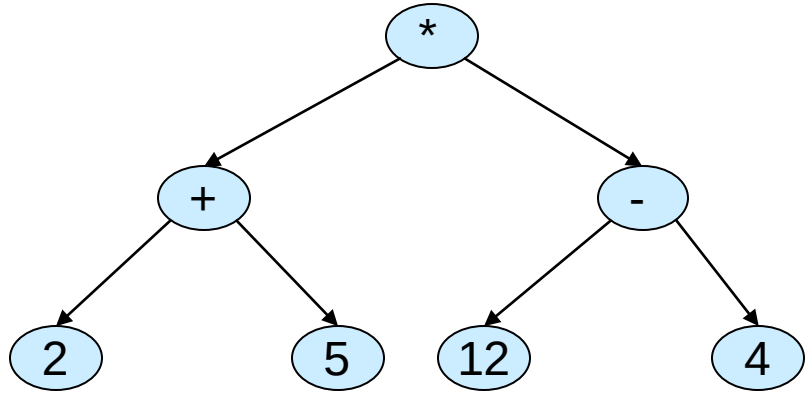
eg: Binary tree node:

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

- (2 + 5) * (12 - 4)



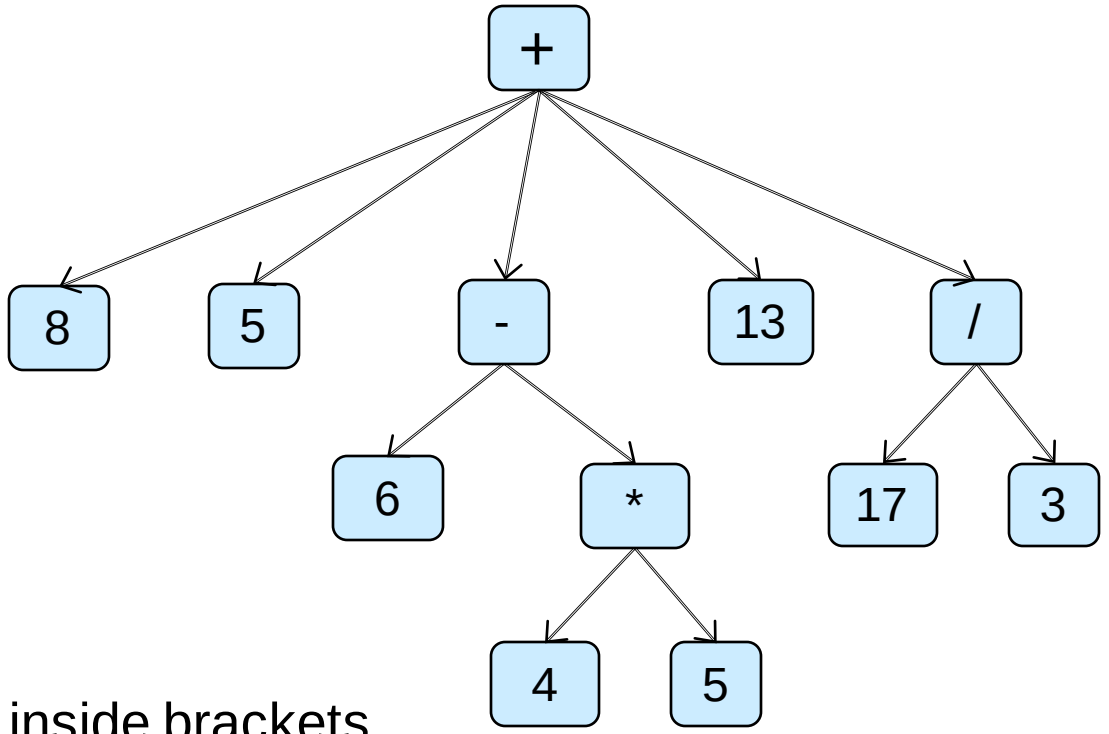
- Printing:
  - standard format with (…)
  - pre-order: * + 2 5 – 12 4 "Polish notation" - never need brackets!
  - post-order: 2 5 + 12 4 - * "Reverse Polish notation"
- Evaluating
  - recursive, post-order traversal of tree
- Reading
  - pre-order and post-order: easy
  - in order: hard! (COMP 261,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

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)

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)               { this.children.add(child); }
public void addChild(int indx, GTNode<E> child) { this.children.set(indx, child); }
public void setChild(int indx, GTNode<E> child)   { this.children.set(indx, child); }
```

- What about iterating through the children?
  - Could return child node at given index:

```
public GTNode<E> getChild(int indx) { return this.children.get(indx); }
```

  - Could enable foreach loop:

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

HOW?

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

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