Algorithms in Systems Engineering
ISE 172

Lecture 13

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Sections 6.1-6.36
- References
  - CLRS Chapter 21 and 22
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

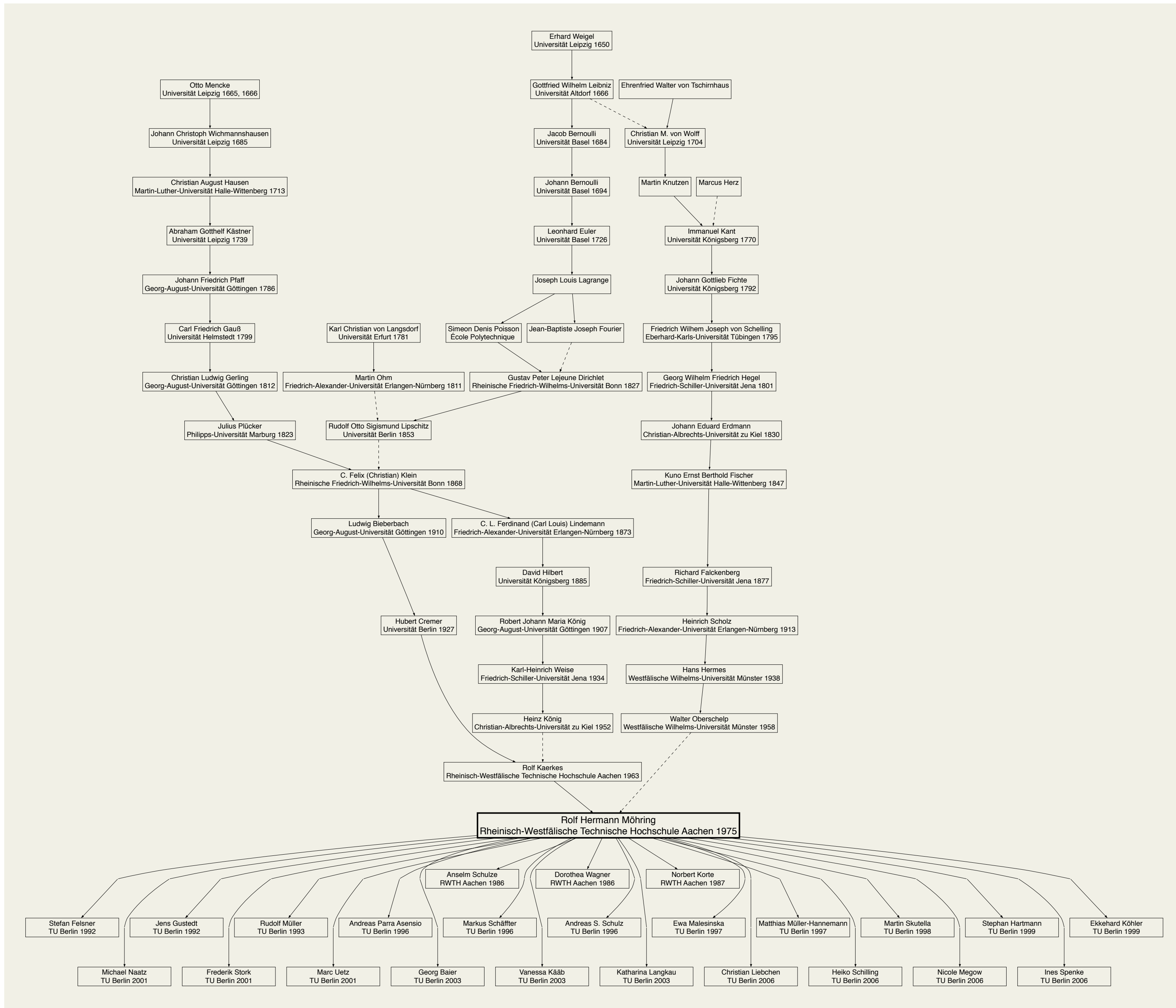
Trees

- A *tree* is a set of items organized into a hierarchical structure (think of a family tree).
- When organized in this way, we call the items *nodes*.
- Each node has a single designated *parent* and one or more *children*.
- There is a single designated node, called the *root*, with no parent.
- Any node with no children is called a *leaf*.
- Any node with children is called *internal*.
- A tree in which all nodes have 2 or fewer children is called a *binary tree*.
- Storing a list of items in a tree structure allows us to represent additional relationships among the items in the list.
- Trees occur naturally in a wide variety of applications.

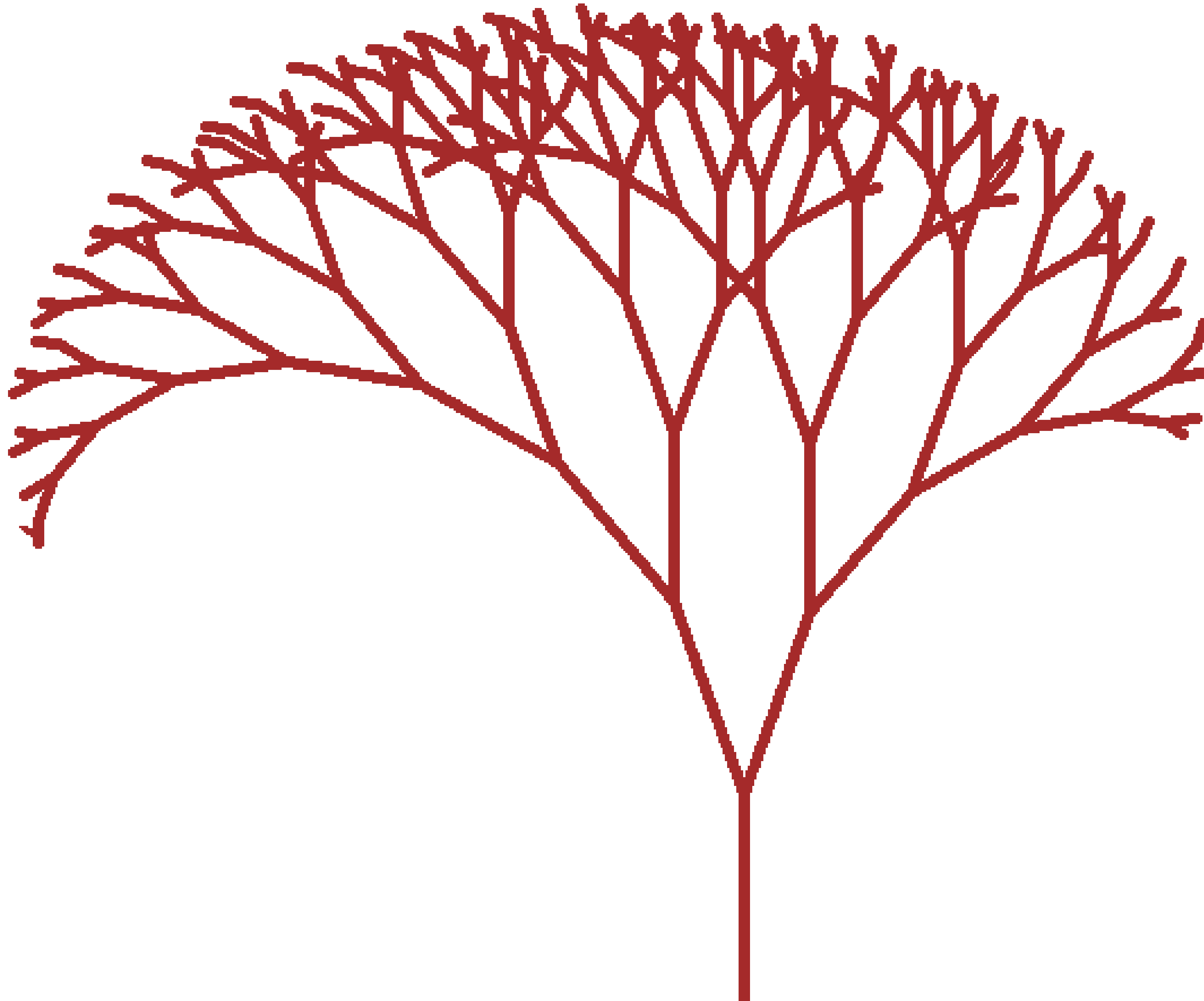
Trees in Action

- File system
- Philogenic Trees
- Family Trees
- Call Trees
- Web page

A Family Tree (Mathematics Geneology



An Tree Generated in Python



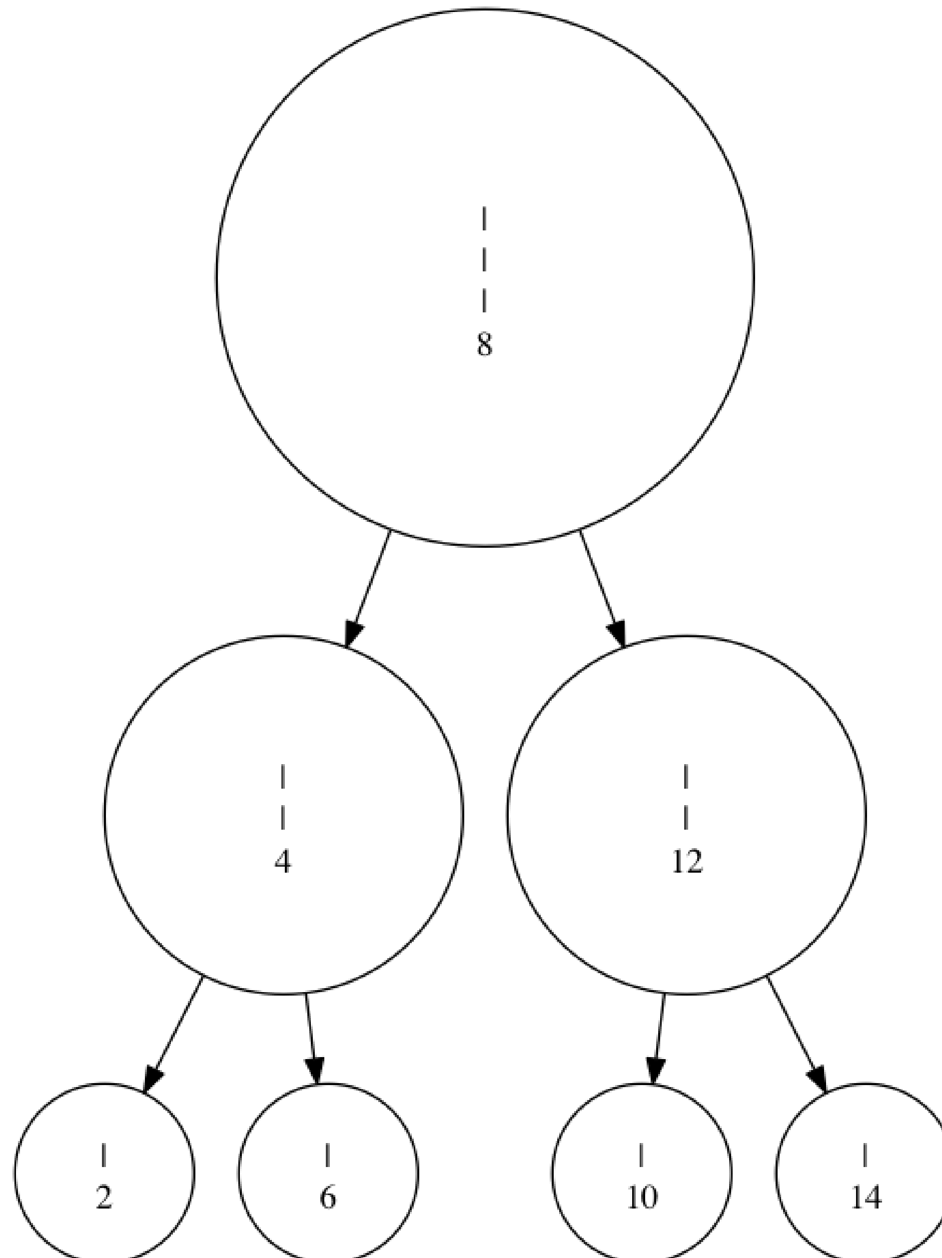
Trees and Recursion

```
def draw_r(self, branchLen):
    if branchLen > 5:
        self.turtle.forward(branchLen)
        self.turtle.right(20)
        self.draw_r(branchLen-15)
        self.turtle.left(40)
        self.draw_r(branchLen-10)
        self.turtle.right(20)
        self.turtle.backward(branchLen)
```

Non-recursive Version

```
def draw_nr_static1(self, branchLen):
    s = Stack()
    while True:
        if branchLen > 5:
            self.turtle.forward(branchLen)
            self.turtle.left(20)
            s.push((branchLen-10,
                    self.turtle.position(),
                    self.turtle.heading()))
            self.turtle.right(40)
            s.push((branchLen-15,
                    self.turtle.position(),
                    self.turtle.heading()))
            self.turtle.left(20)
        if s.isEmpty():
            break
        branchLen, p, h = s.pop()
        self.turtle.setheading(h); self.turtle.penup()
        self.turtle.goto(p);       self.turtle.pendown()
```

A Tree Representing Location of Ruler Hashmarks



The Code

```
def ruler(hash_marks, left = 0, right = 16, height = 3):
    if height > 0:
        middle = (left+right)//2
        ruler(hash_marks, left, middle, height - 1)
        hash_marks[middle] = height
        ruler(hash_marks, middle, right, height - 1)

def ruler_nr(hash_marks, left = 0, right = 16, height = 3):
    for i in range(height):
        for j in range(left+right//(2**(height-i)), right,
                       right//(2**(height-i))):
            hash_marks[j] = i+1
```

Additional Terminology

- The *level* of a node in the tree is the number of recursive calls to `parent()` needed to reach the root.
- The *depth* of the tree is the maximum level of any of its nodes.
- A *balanced tree* is one in which all leaves are at levels k or $k-1$, where k is the depth of the tree.
- Additional terms
  - Edge
  - Path
  - Siblings
  - Subtree

Tree Data Structures

- The tree ADT can be thought of as a list ADT with additional structure.
- One of the most important roles of the additional structure is to allow for the list to be traversed easily in various orders.
- We may also want to be able to be able to query the relationships of a given node to others (parent/sibling/child).

Tree ADT

```
class Tree:
    def __init__(self, root)
    def add_root(self, n, **attrs)
    def get_root(self)
    def add_child(self, name, parent, **attrs)
    def get_children(self, name) # return list of children
    def get_parent(self, name)
    def print_nodes(self, order, priority) # binary tree only
    def traverse(self, q) # print nodes in order
    def __contains__(self, name)
    def __iter__(self) # iterate over nodes in order
```

Additional Functionality

- Later, we'll want to be able to "splice" nodes into the tree at particular places.
- We'll also want to be able to do certain "rotations" in which we change the parent/child relationships in a systematic way.
- The goal of these operations will be to maintain a certain structure in the tree.
- This will make certain kinds of additions, deletions, and traversals efficient so we can implement additional operations.

Traversing a Tree

- Traversing a tree consists of visiting the nodes in a specified order, starting at the root node.
- As we encounter each node, we put all of its children on the list to be visited.
- The order in which we take nodes off this list determined the *search order*.
  - Depth-first: Last in, first out. This means that we visit the node in the list at the deepest level first.
  - Breadth-first: First in, first out. This means we visit the node in the list at the shallowest level first.

Traversing a Tree (Depth First)

Here is a recursive implementation of a depth-first search.

```
def dfs(self, root):
    print root
    for i in self.get_children(current):
        print i
        self.depth(root)
```

Traversing a Tree (Depth First)

We can also do depth-first search with a stack

```
def dfs(self):
    s = Stack()
    s.push(self.get_root)
    while s.isEmpty() != True:
        current = s.pop()
        print current
        for i in self.get_children(current):
            s.push(i)
```

Traversing a Tree (Breadth First)

To get breadth first search, we can simply replace the stack with a queue:

```
def dfs(self):
    s = Queue()
    s.enqueue(self.get_root)
    while s.isEmpty() != True:
        current = s.dequeue()
        print current
        for i in self.get_children(current):
            s.enqueue(i)
```

Breadth First Search of Generated Tree

```
def draw_nr_static1(self, branchLen):
    s = Queue()
    while True:
        if branchLen > 5:
            self.turtle.forward(branchLen)
            self.turtle.left(20)
            s.push((branchLen-10,
                    self.turtle.position(),
                    self.turtle.heading()))
            self.turtle.right(40)
            s.push((branchLen-15,
                    self.turtle.position(),
                    self.turtle.heading()))
            self.turtle.left(20)
        if s.isEmpty():
            break
        branchLen, p, h = s.pop()
        self.turtle.setheading(h); self.turtle.penup()
        self.turtle.goto(p);       self.turtle.pendown()
```

Binary Trees

- In many applications, the trees that arise are binary by nature.
- The call tree in quicksort or mergesort is an example.
- When we know that there will be at most two children of a given node, we call them the *right* and *left* children.
- We can specialize the ADT by adding methods to access the right and left children directly.
  - `get_parent(name)`: return the parent of node `index`.
  - `get_right_child(name)`: return the "right" child of node `name`.
  - `get_left_child(name)`: return the "left" child of node `name`.

Binary Tree ADT

```
class BinaryTree(Tree):

    def get_left_child(self, name):
        get_children(index)[0]

    def get_right_child(self, name):
        get_children(index)[1]
```

Traversing a Binary Tree (Pre-order)

When doing a depth first search, if we print each node before searching either of the children recursively, this produces an "pre-order" traversal.

```
def depth(self, root):
    print root
    self.depth(get_left(root))
    self.depth(get_right(root))
```

Traversing a Binary Tree (In-order)

Alternatively, if we print each node in between searching the left and right subtrees, this produces an "in-order" traversal.

```
def depth(self, root):
    self.depth(get_left(root))
    print root
    self.depth(get_right(root))
```

Traversing a Binary Tree (Post-order)

Finally, if we print each node after searching both the left and right subtrees, this produces an "in-order" traversal.

```
def depth(self, root):
    self.depth(get_left(root))
    self.depth(get_right(root))
    print root
```

Running Time of Tree Traversal

- What is the running time of these tree traversal methods?

Example: Expression Tree