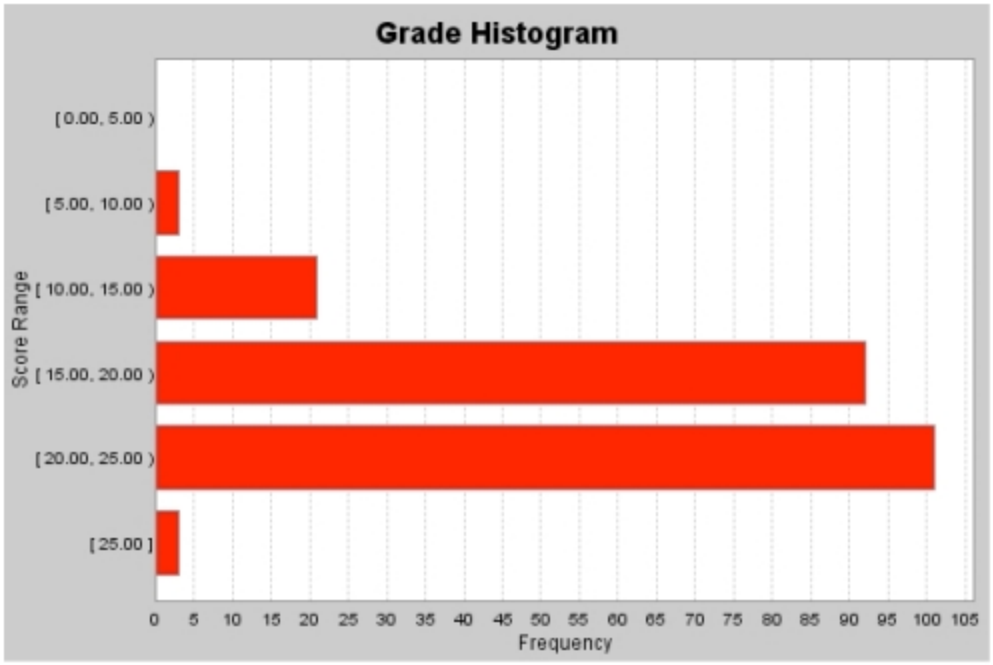
CS 173: Discrete Structures

Eric Shaffer
Office Hour: Wed. 12-1, 2215 SC
shaffer1@illinois.edu

Announcements

- Quiz 2 grading is done and in compass
  - Median of 19.5/25



- Mid-term 2 in class Wednesday April 8
  - Topic list will be up tonight
  - Discussion sections next week will be review sessions

Patches to HW 7

- Look on the HW webpage for patches and hints
- Changes to Question 2 and 3

- For Question 2:
  Use a base case of $T(n) = \theta(1)$ for $n \leq 1$

Homework 7 clarifications
Problem 3

- As written, the code doesn't do anything useful
- It can be fixed to be useful by moving swap()
- You can answer the question using either version
- STATE WHICH VERSION YOU ARE ANALYZING

Useless(?) version

```
procedure select( a1,..., an, k)
for i := 1 to k
begin
   min := i
   for j := i + 1 to n
      if aj < amin then
      begin
        min := j
        swap(ai, amin)
      end
end
selected := ak
```

Useful version

```
procedure select( a1,..., an, k)
for i := 1 to k
begin
   min := i
   for j := i + 1 to n
      if aj < amin then
      begin
        min := j
      end
   swap(ai, amin)
end
selected := ak
```

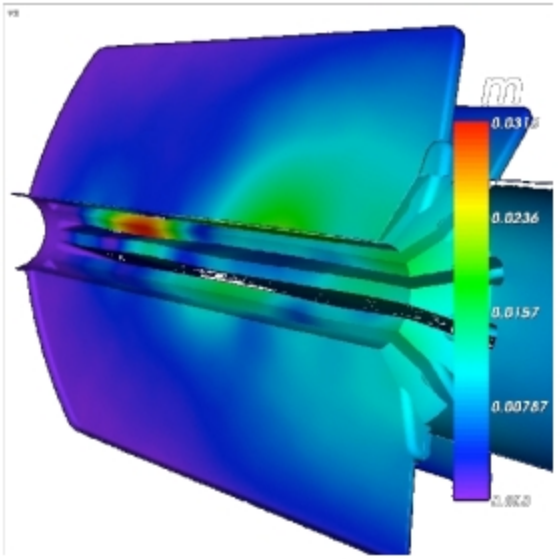
inner loop

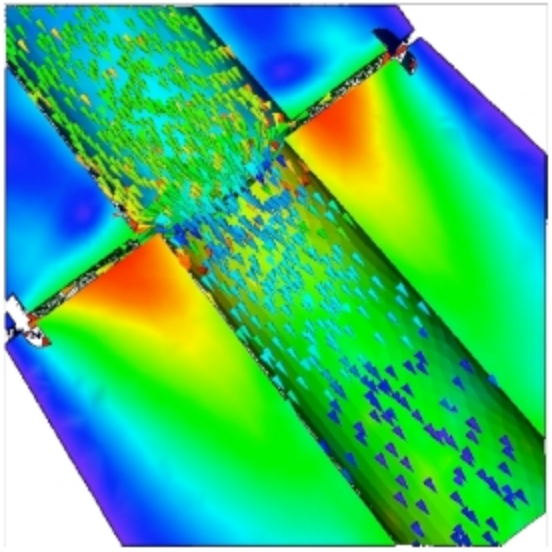
Homework 7 clarifications
Problem 3

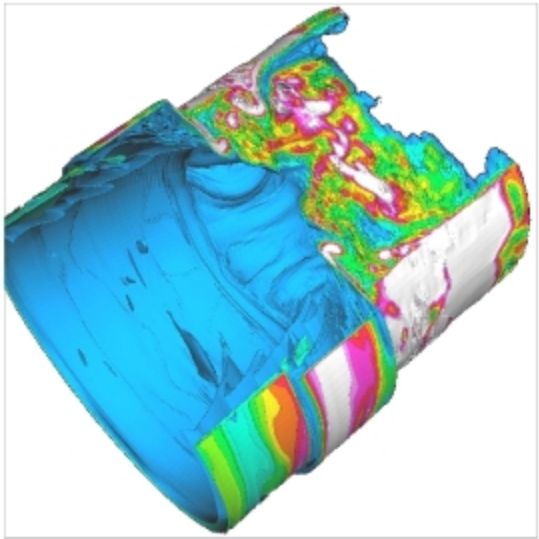
- This only affects the answer to part a
- **The answer to part b is the same for either version**

Why you should talk in class

- Class will get done sooner if we eliminate the long pauses in which no one answers my questions
- The Center for Simulation of Advanced Rockets hires undergrad hourly research assistants







When is recursion inefficient?

Recursive function for computing the nth Fibonacci number:

$T(n)$ = # ops to find F_n

$T(1 \text{ or } 2) = \Theta(1)$

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

$= \boxed{O(2^n)}$

```
fib(positive integer N)
  if ((N = 1) or (N = 2))
     return 1
  else
     return (fib(N-1) + fib(N-2))
```

Iterative function for computing the nth Fibonacci number:

```
fib (positive integer N)
  x:=0          Θ(1)
  y:=1
  for j: = 0 to n-1      → Θ(n) times through loop
     z := x +y
     x:=y       Θ(1)
     y:=z
  return y
```

$\Theta(1) + \Theta(n)\Theta(1)$

$= \boxed{\Theta(n)}$

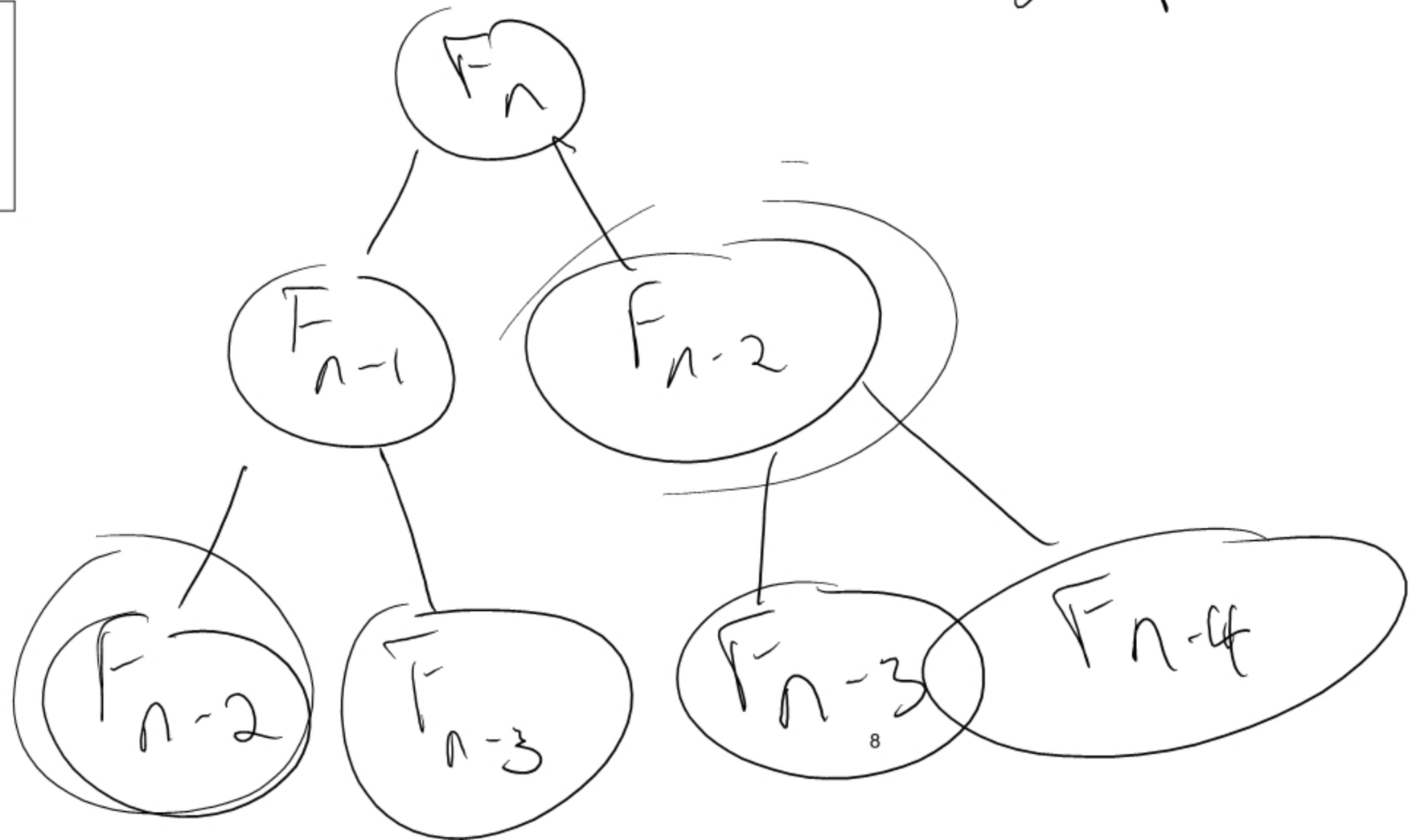
Which of these is better?

When is recursion inefficient?

when computation is duplicated

Recursive function for computing the nth Fibonacci number:

```
fib(positive integer N)
  if ((N = 1) or (N = 2))
     return 1
  else
     return (fib(N-1) + fib(N-2))
```



Recursion versus iteration

Iteration (loops) and recursion both perform an action repeatedly

Tail-recursive functions are directly equivalent to iteration

e.g. return recursive_call();

Most direct, linear recursive calls can be rewritten as tail recursion

e.g. return n(fatorial(n-1))

Any problem solved recursively can be solved iteratively*

(*=with appropriate data structures)

Recursion in programming practice

Advantages:

- Can allow simple solution for complex problems
- Results in fewer lines of code, fewer bored programmers

Disadvantages:

- Reputation as slow in practice (more function calls)
- Reputation as using more memory in practice
  - NOTE: NOT true for simple recursive functions (compiler optimizes the code)

To sum things up…

- What you should know:
  - Sequential search
  - Binary search
  - Insertion sort
  - Bubble sort
  - Merge sort
- Be able to analyze algorithmic complexity
  - Worst case and average case
- Use a recurrence relation to describe algorithmic complexity
  - Solve simple recurrences via "unrolling"

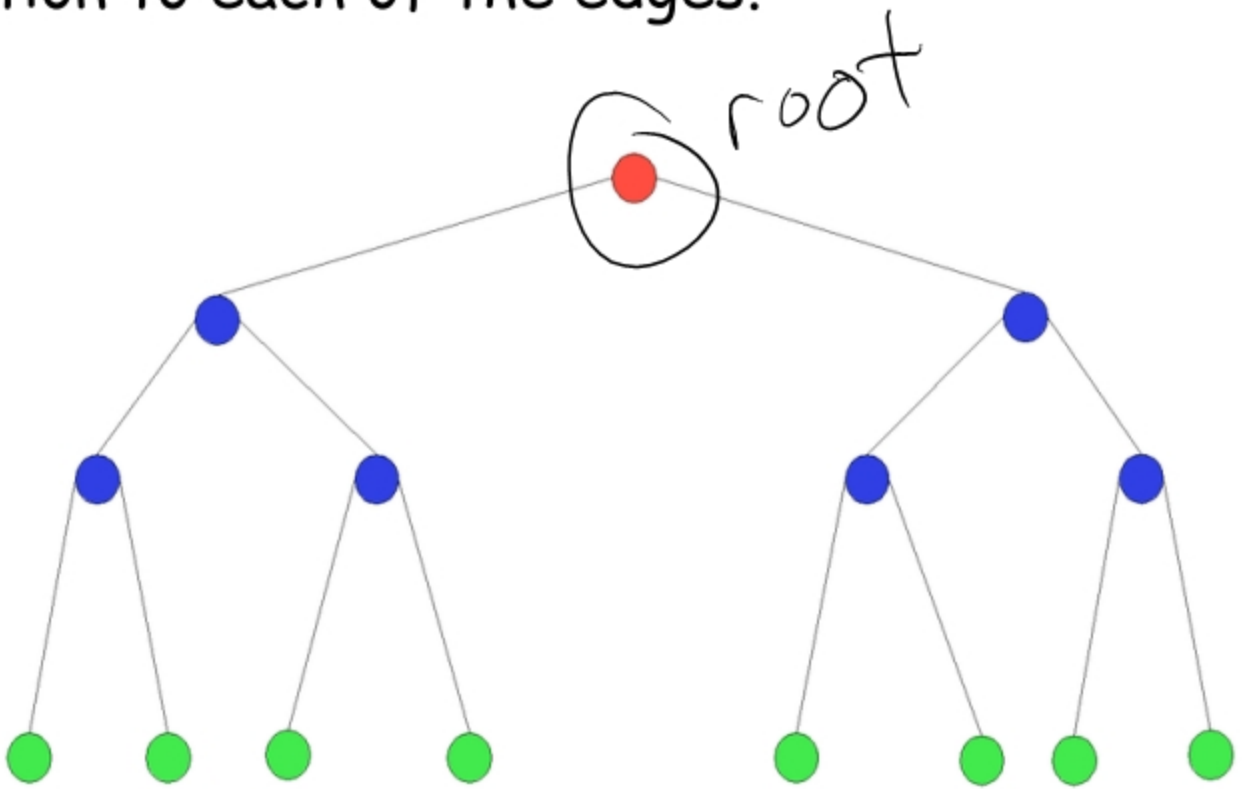
Trees

- A **Tree** is a mathematical object
(just like number or a set or…some other mathematical object)|

- It has a set V of vertices $v_1, v_2, v_3, \ldots, v_n$
- It has a set E of edges $e_1, e_2, \ldots e_{n-1}$
  - Each edge connects a pair of vertices for example: $e_k = (v_i, v_j)$
- It has 2 important properties:
  - It is **connected** (each vertex is reachable from any other)
  - It is **acyclic** (only one path between 2 vertices)
    - Equivalent condition: a tree with n vertices has n-1 edges

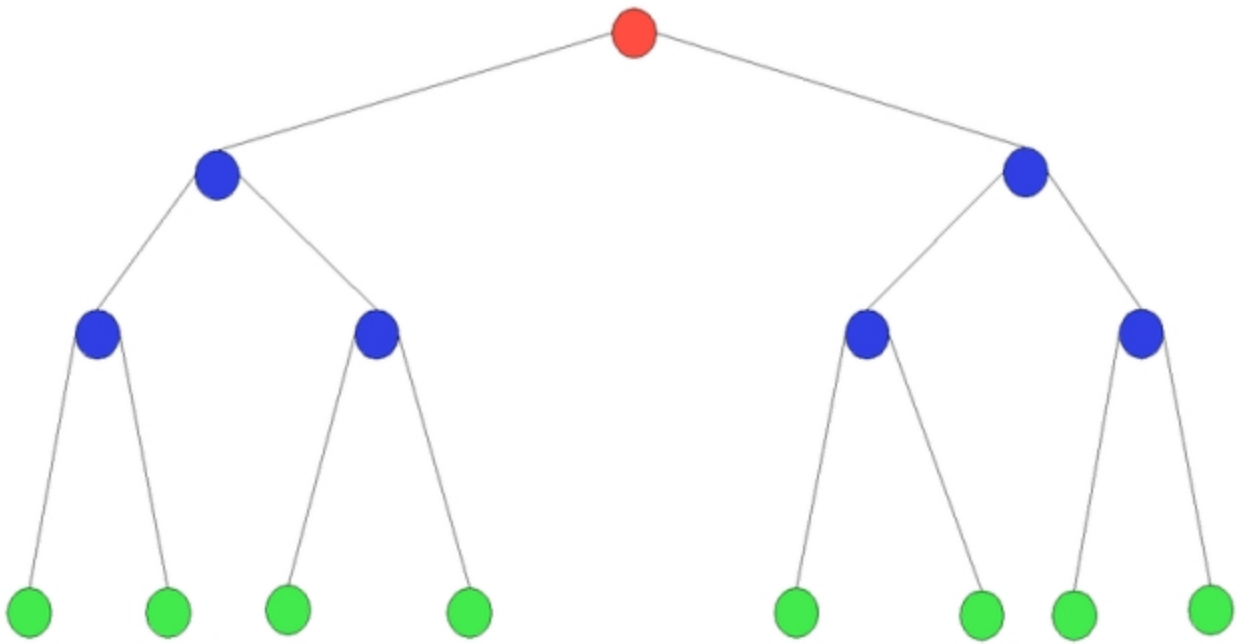
Rooted Trees

Once a vertex of a tree has been designated as the *root* of the tree, it is possible to assign direction to each of the edges.

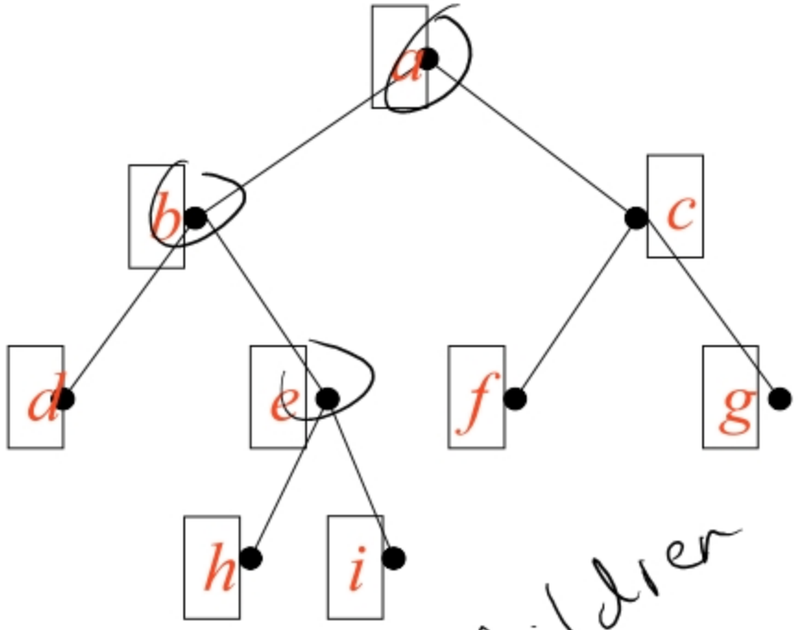


Subtrees

A subtree S is subset of the vertices and edges of a tree T with S also being a tree.



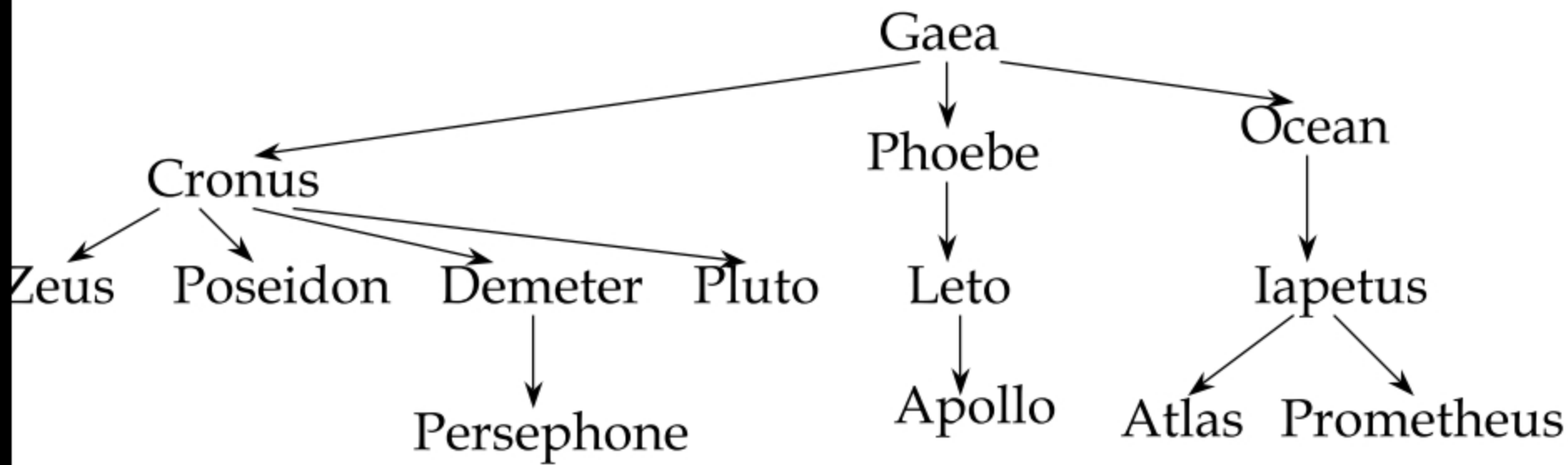
Rooted tree terminology



- Root: start of the tree, gives it a direction
- Child c is a child of a
- Parent a is the parent of c
- Sibling 2 vertices are sibs if have same parent
- Ancestor
  - e, b, a ancestors of h
- Leaf has no children
- Internal vertex has child(ren)

A Family Tree

Much of tree terminology derives from family trees.



m-ary trees

A rooted tree is called an *m-ary tree*

- if every internal vertex has no more than *m* children.

The tree is called a *full m-ary tree*

- if every internal vertex has exactly *m* children

An *m*-ary tree with *m*=2 is called a *binary tree*.

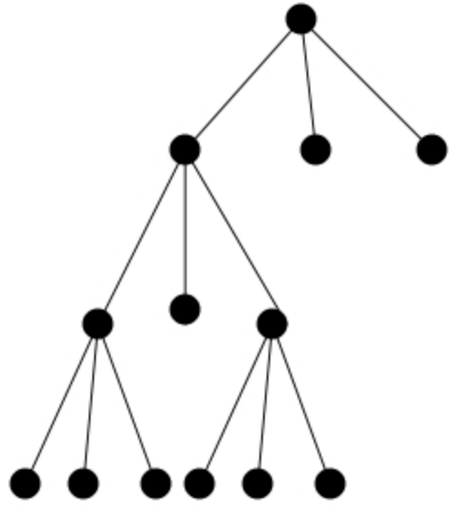
Properties of Trees

$V = V_c \cup V_R$

A full m-ary tree with i internal vertices contains $n = \underline{mi+1}$ vertices.

$V_R = \{$ vertices w/no parent $\}$

$V_c = \{$ vertices w/ parent $\}$

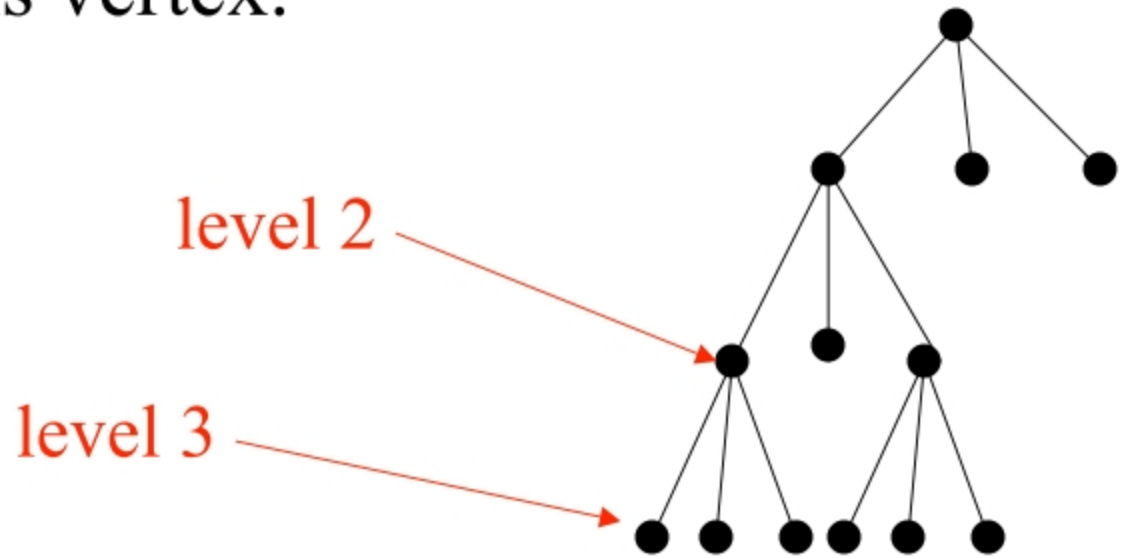


$|V_R| = 1$

$|V_c| = mi$

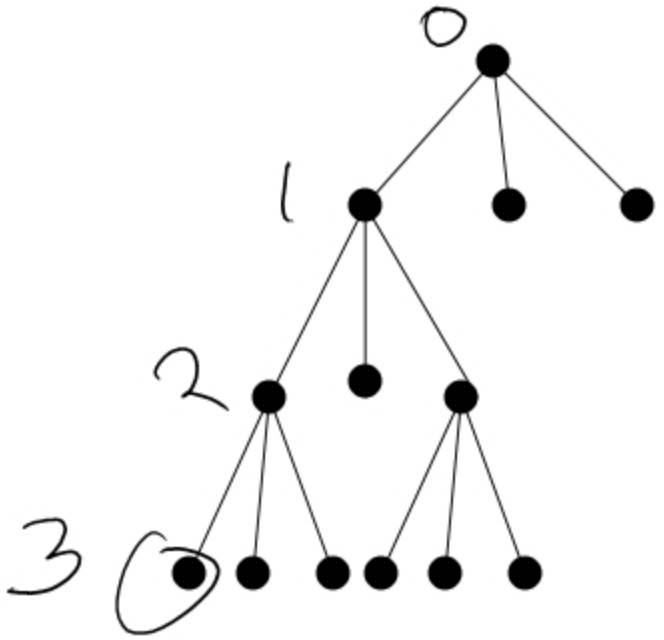
Properties of Trees

The ***level of a vertex v*** in a rooted tree is the length of the unique path from the root to this vertex.



Properties of Trees

The ***height*** of a rooted tree is the maximum of the levels of vertices.

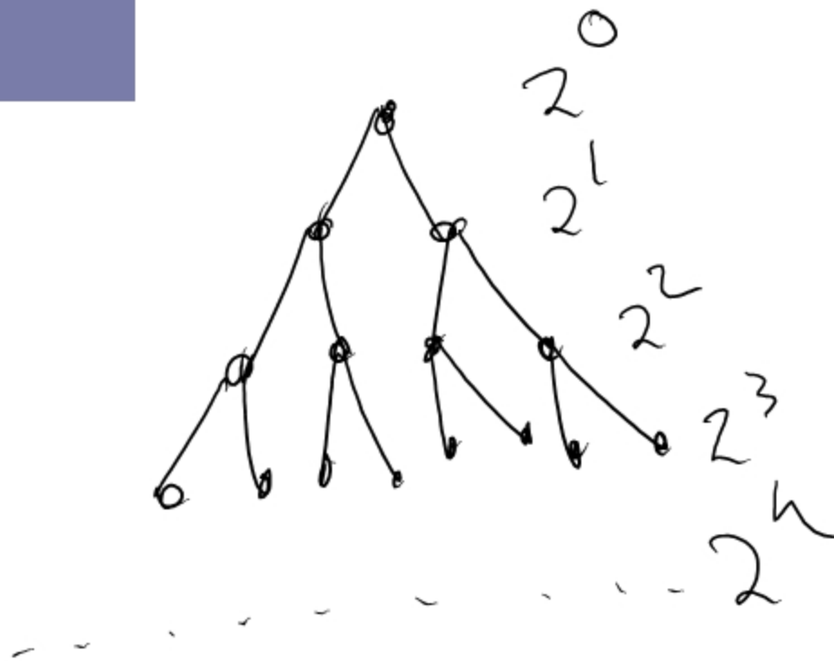


$h = 3$

Binary Trees

- In a binary tree of height h
  - What is the maximum number of vertices?

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$



  - What is the minimum number of vertices?

$h+1$

Binary Trees

A binary tree of height h is balanced iff
all leaves are at height h or h-1.

Theorem: ∃ at most 2^h leaves in binary tree of height h.

Prove using induction: Induction on h

Base case: h=0, 1 leaf (the root)
leaves = 2^0

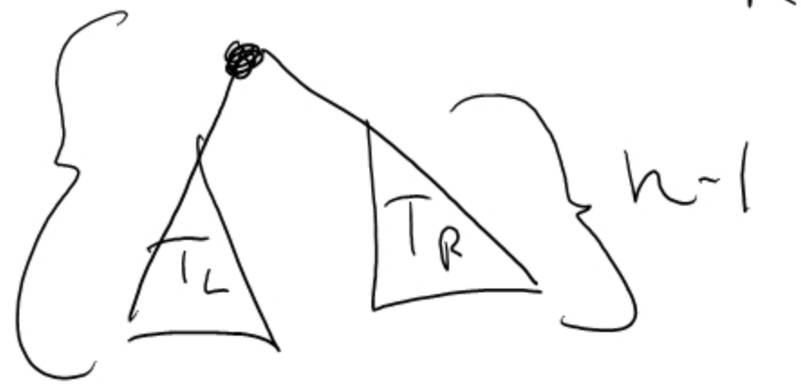
Let L = # leaves in a tree of height h
Let T be a binary tree of height h

Binary Trees

A binary tree of height h is balanced iff all leaves are at height h or h-1.

Inductive Step:

T has 2 subtrees T_L, T_R



Theorem: ∃ at most 2^h leaves in binary tree of height h.

$L = L_L + L_R$ with L_L = # leaves in T_L

L_R = # leaves in T_R

Assume a tree of height h-1 has at most 2^{h-1} leaves

$$L = L_L + L_R \leq 2^{h-1} + 2^{h-1} = 2 \cdot 2^{h-1} = 2^h$$