CSE 373: Data Structures and Algorithms

Lecture 8: Finish Hash Table Collisions, Intro to Trees

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

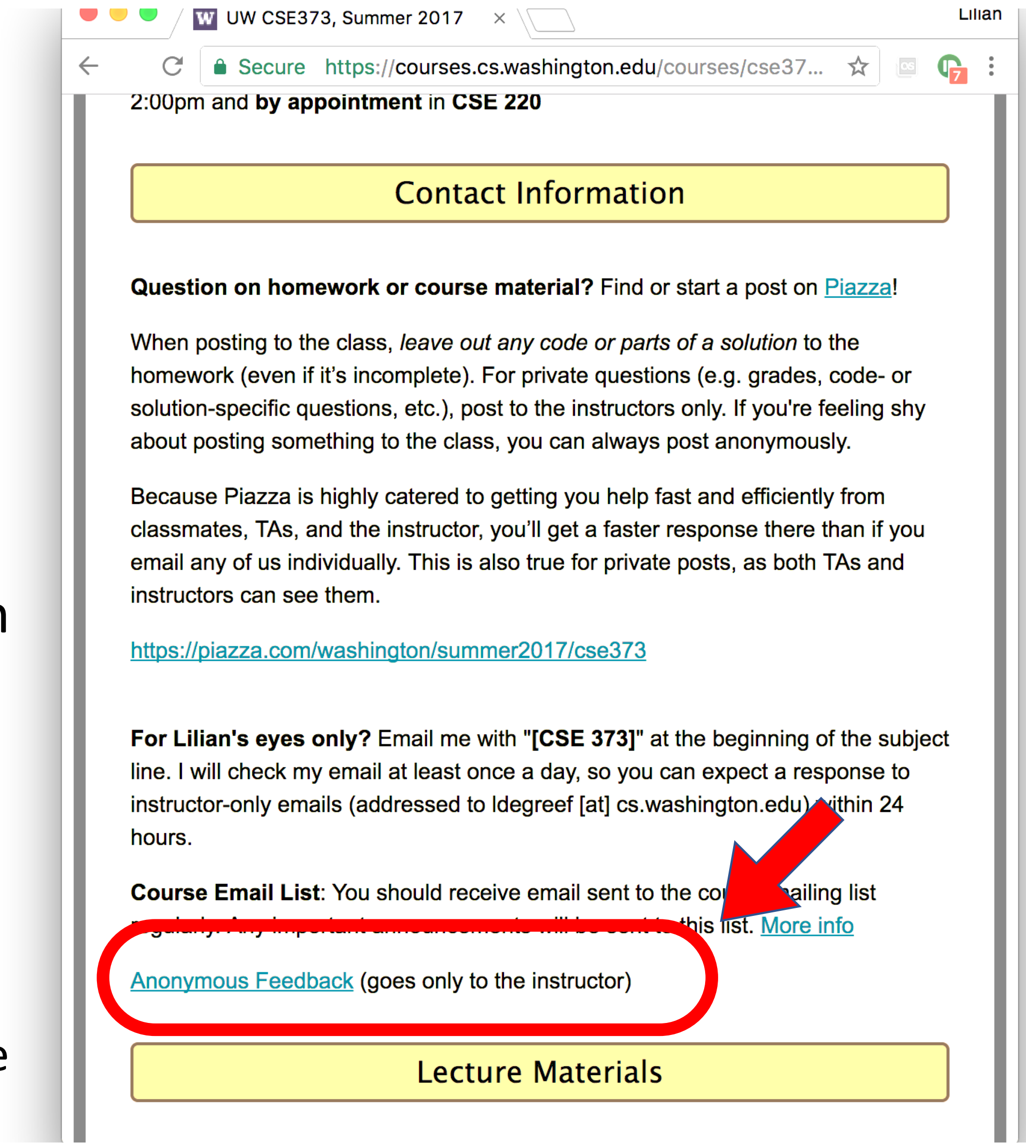
- Announcements
- Wrap up Hash Table Collisions
  - Open Addressing: Quadratic Probing
  - Open Addressing: Double Hashing
  - Rehashing
- Introduce Trees
  - Generic Trees
  - Binary Trees

Announcements

- Homework 3 is out
  - Pair-programming opportunity!
  - Start early

- Anonymous feedback mechanism available on website

- Homework from long weekend
  - Forgot to ask for it last lecture
  - Pile on top of slide print-outs on your way out
  - Ungraded, but am interested to see



Hash Table Collisions

Continued -- Part 2!

Finishing up Open Addressing

Collision resolution that uses the empty space in the table

Open Addressing: Quadratic Probing

- We can avoid primary clustering by changing the probe function

  `(h(key) + f(i)) % TableSize`

- A common technique is quadratic probing: **`f(i) = i`^{2}**
  - So probe sequence is:
    - 0^{th} probe: `h(key) % TableSize`
    - 1^{st} probe: `(h(key)+ 1) % TableSize`
    - 2^{nd} probe: `(h(key)+ 4) % TableSize`
    - 3^{rd} probe: `(h(key)+ 9) % TableSize`
    - …
    - i^{th} probe: `(h(key) + i`^{2}`) % TableSize`

- Intuition: Probes quickly “leave the neighborhood”

Quadratic Probing Example #2

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

i^{th} probe: `(h(key) + i`^{2}`) % TableSize`

Quadratic Probing: Bad News, Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full

- Good news:
  - If `TableSize` is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most `TableSize/2` probes
  - So: If you keep $\lambda < ½$ and `TableSize` is *prime*, no need to detect cycles

  - Proof is posted online next to lecture slides
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime `T` and $0 < i, j < T/2$ where $i \neq j$,
      $(k + i^2) \% T \neq (k + j^2) \% T$ (i.e. no index repeat)

Clustering Part 2

- Quadratic probing does not suffer from primary clustering:
  no problem with keys initially hashing to the same neighborhood

- But it's no help if keys initially hash to the same index:

  This is called

- Can avoid secondary clustering

Open Addressing: Double Hashing

Idea:

- Given two good hash functions *h* and *g*, it is <u>very unlikely</u> that for some *key*, `h(key) == g(key)`

- So make the probe function **`f(i) = i*g(key)`**

Probe sequence:

- 0th probe: `h(key) % TableSize`
- 1st probe: `(h(key) + g(key))    % TableSize`
- 2nd probe:
- 3rd probe:
- …
- ith probe: `(h(key) + i*g(key)) % TableSize`

Double Hashing Analysis

- Intuition: Because each probe is “jumping” by `g(key)` each time, we “leave the neighborhood” *and* “go different places from other initial collisions”

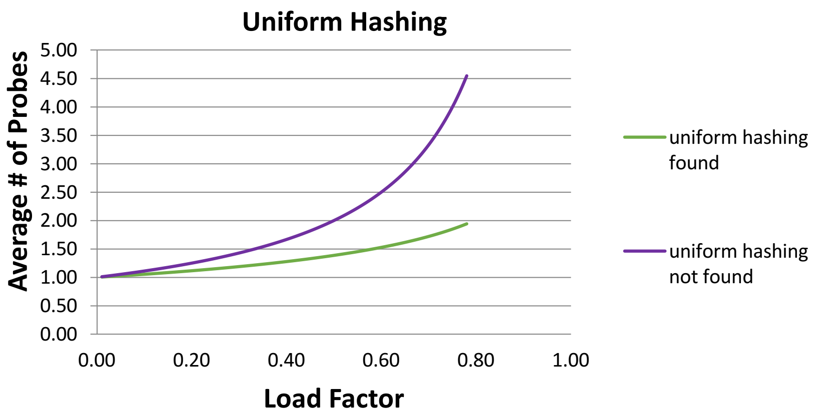
- Requirements for second hash function:

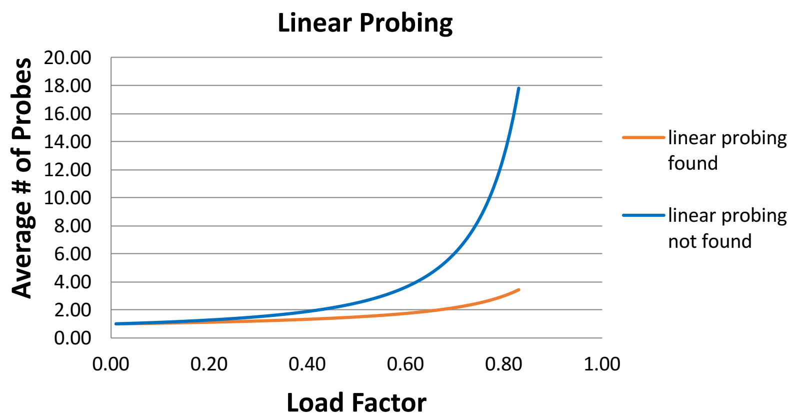
- Example of double hash function pair that works:
  - `h(key) = key % p`
  - `g(key) = q - (key % q)`
  - `2 < q < p`
  - `p` and `q` are prime

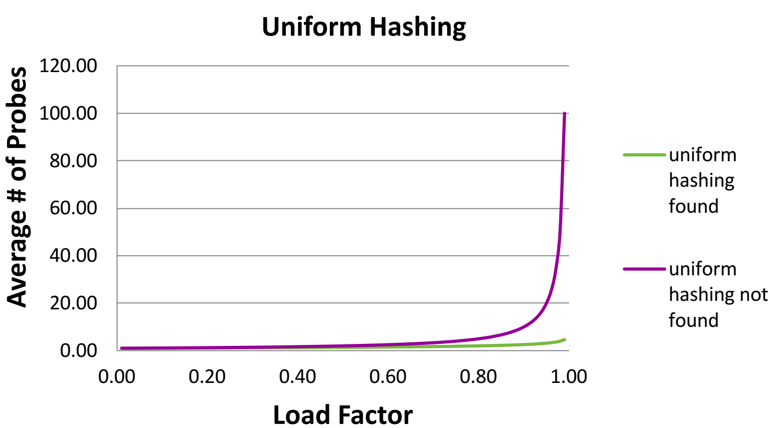
More Double Hashing Facts

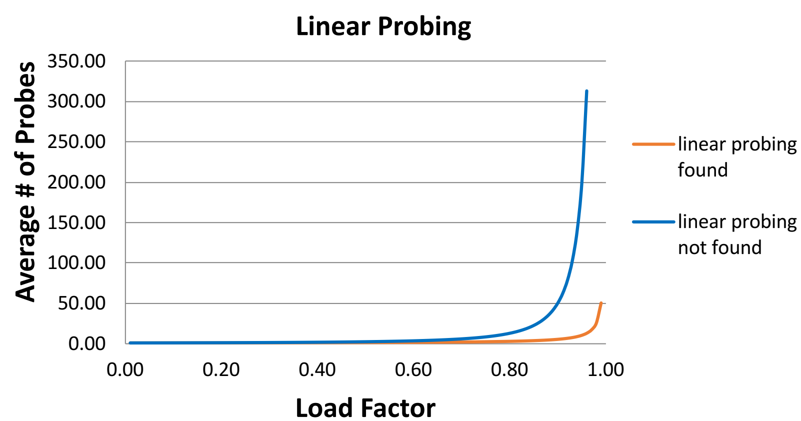
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`
- Non-trivial facts we won't prove:
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search (intuitive): $\dfrac{1}{1-\lambda}$
  - Successful search (less intuitive): $\dfrac{1}{\lambda}\log_e\left(\dfrac{1}{1-\lambda}\right)$
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

Charts


Uniform Hashing
Average # of Probes
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Linear Probing
Average # of Probes
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found


Uniform Hashing
Average # of Probes
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Linear Probing
Average # of Probes
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found

Rehashing

Rehashing

- What do we do if the table gets too full?

- How do we copy over elements?

Rehashing

- What's "too full" in Separate Chaining?

- "Too full" for Open Addressing / Probing

Rehashing

- How big do we want to make the new table?

- Can keep a list of prime numbers in your code, since you likely won't grow more than 20-30 times ($2^{30} = 1,073,741,824$)

Wrapping up Hash Tables

- A hash table is a data-structure for

- Some example uses of hash tables:

Another Data-Structure for Dictionaries?

Dictionary meaning:

- Set of (key, value) pairs
- Can compare keys

Dictionary operations:

- `insert` (key, value)
- `delete` (key)
- `find` (key)

Trees!

Trees

Are like linked-lists, but can have more than one “next”

Tree terms

Root (tree)

Leaves (tree)

Children (node)

Parent (node)

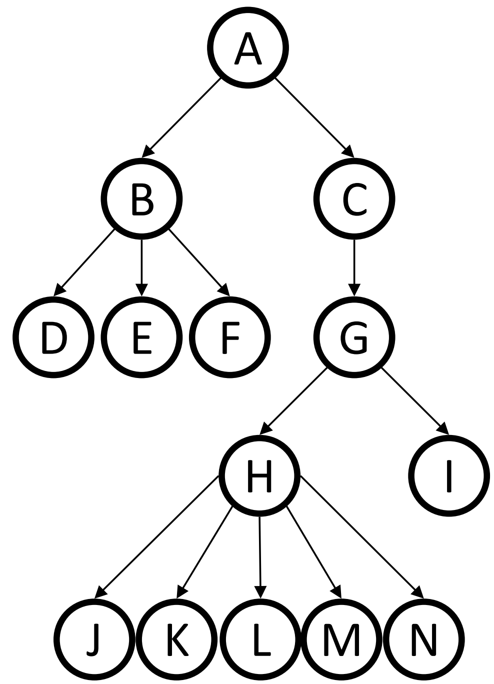
Siblings (node)

Ancestors (node)

Descendents (node)

Subtree (node)

Tree **T**



Tree terms

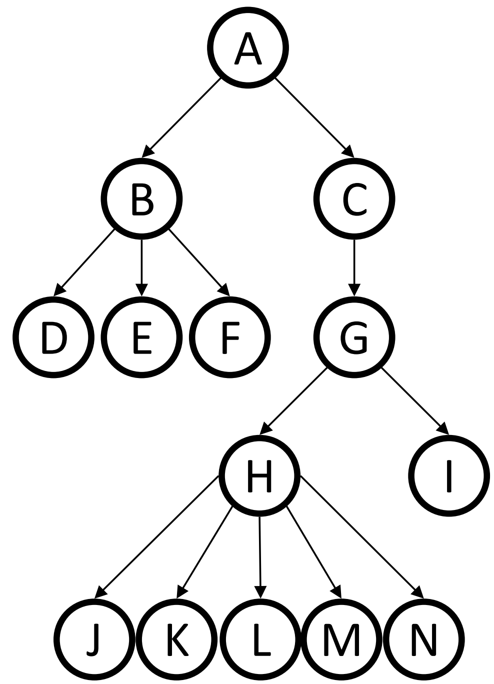
Depth (node)

Height (tree)

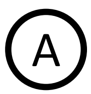
Degree (node)

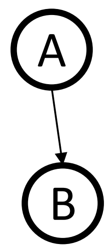
Branching factor (tree)

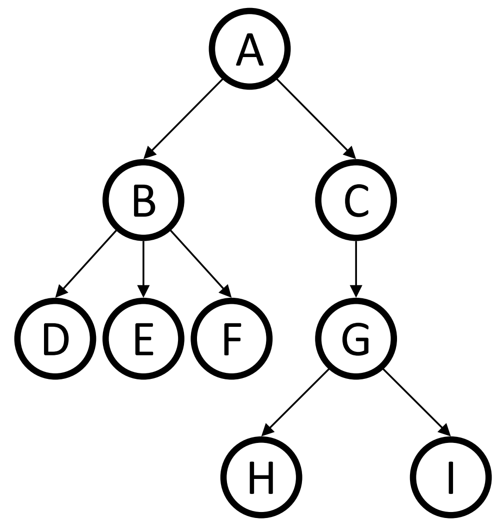
Tree **T**



Practice with Height and Depth



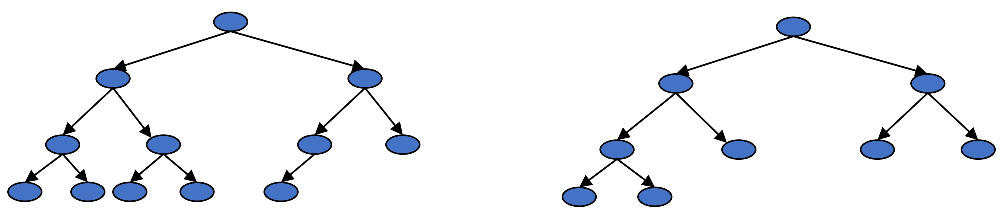




Kinds of Trees

Certain terms define trees with specific structure

- **Binary tree**: Each node has at most
- ***n*-ary tree**: Each node has at most
- **Perfect tree**: Each row
- **Complete tree**: Each row completely full except



What is the height of a perfect binary tree with n nodes?
A complete 14-ary tree?

More Tree Terms

- There are many kinds of trees

- There are many kinds of binary trees

- A tree can be balanced or not
  - A balanced tree with *n* nodes has a height of
  - Different kinds of trees use different “balance conditions” to achieve this

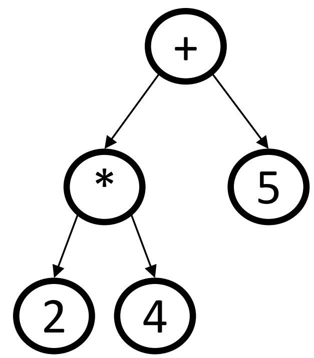
(Bonus Material) Cool Uses & Kinds of Trees!

Binary Search Tree - dictionaries and more

Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expression

Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.

Binary Tries - Used in almost every high-bandwidth router for storing router-tables.



For now, focusing on generic and binary search trees (don't worry about the other ones listed here -- I just think they're cool and want to share!)

(Bonus Material) Cool Uses & Kinds of Trees!

Game Tree - Used in computer chess and other game AIs

GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.

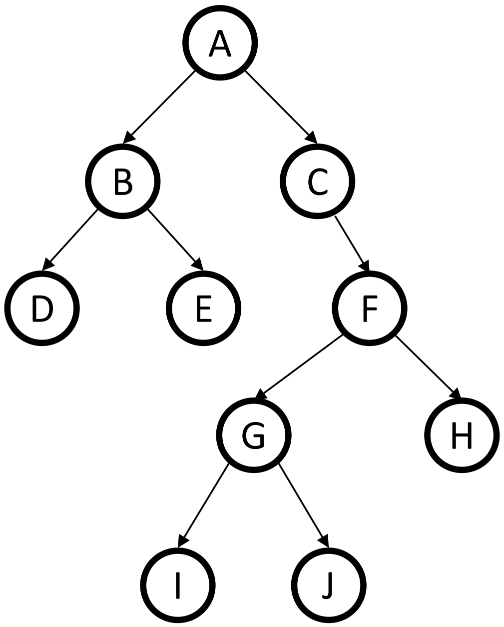
Vantage-Point Trees - Used in bioinformatics to store huge databases of genomic data records

… and many more kinds and uses of trees!

For now, focusing on generic and binary search trees (don't worry about the other ones listed here -- I just think they're cool and want to share!)

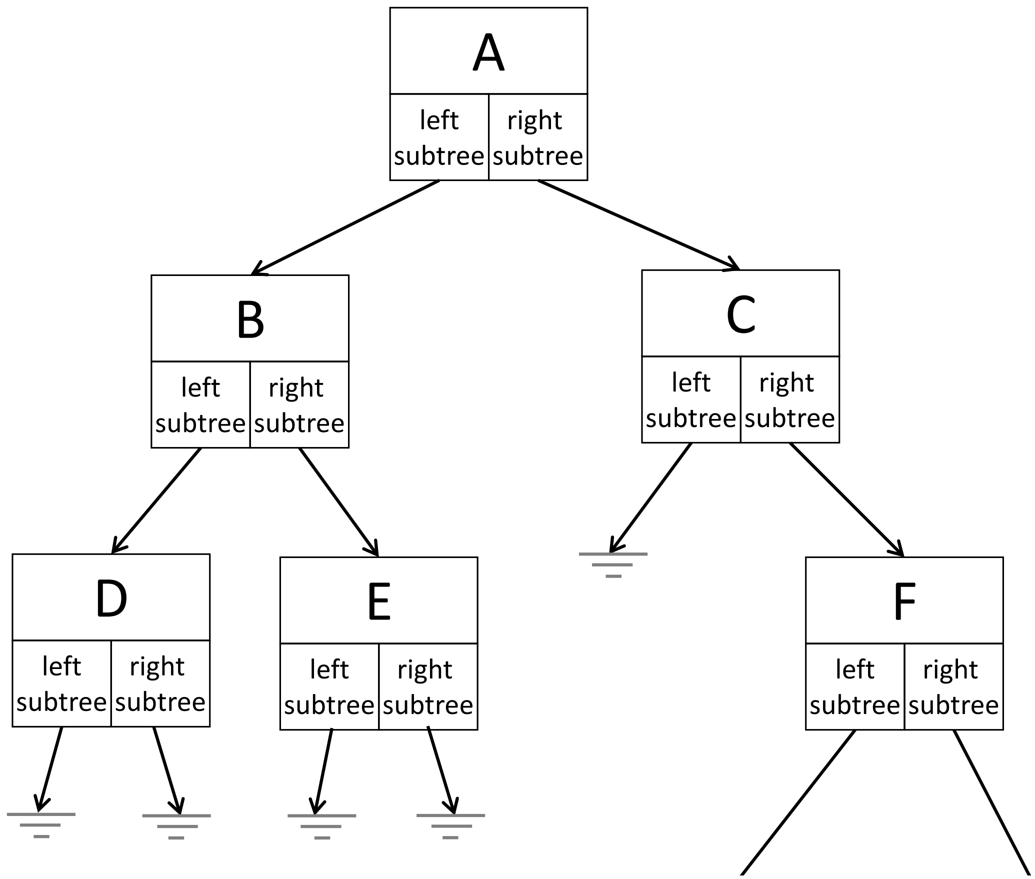
Binary Trees

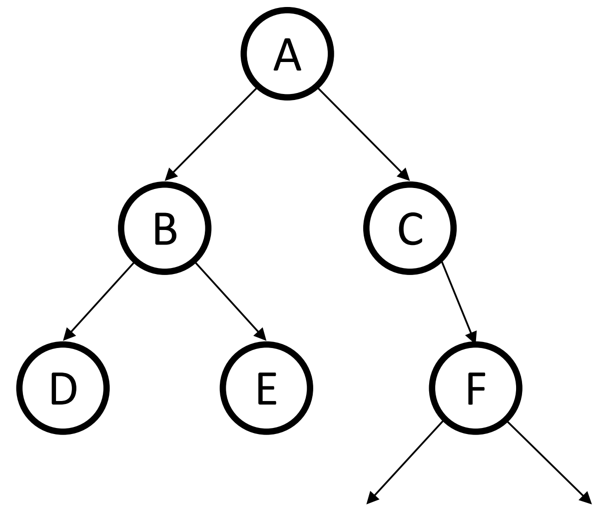
- **Binary tree**: Each node has at most 2 children (branching factor 2)
- Binary tree is



- Representation:
- For a dictionary, data will include a key and a value

Binary Tree Representation





Practice time! What does the following method do?

```
int mystery(Node node){
  if (node == null),
      return -1;
  return 1 + max(mystery(node.left),
                 mystery(node.right);
}
```

A. It calculates the number of nodes in the tree.

B. It calculates the depth of the nodes.

C. It calculates the height of the tree.

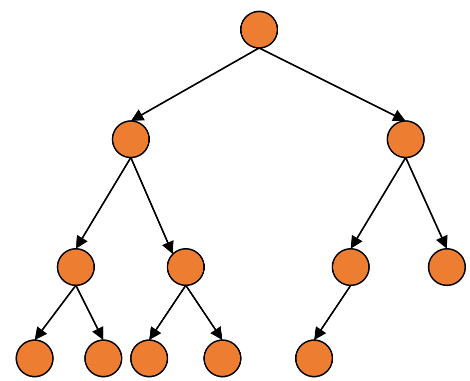
D. It calculates the number of leaves in the tree.

Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height *h*:

- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:



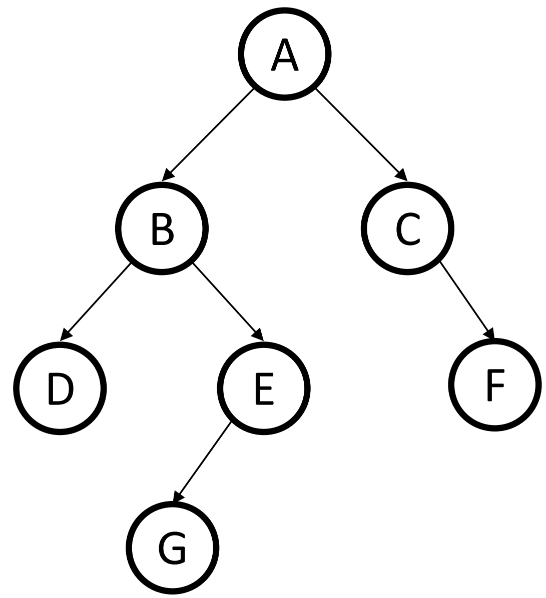
For *n* nodes, the min height (best-case) is

the max height (worst-case) is

Tree Traversals

A **traversal** is an order for visiting all the nodes of a tree

- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root



Tree Traversals: Practice

Which one makes sense for evaluating this *expression tree?*

- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root