CSE 373: Data Structures and Algorithms

Lecture 8: Finish Hash Table Collisions, Intro to Trees

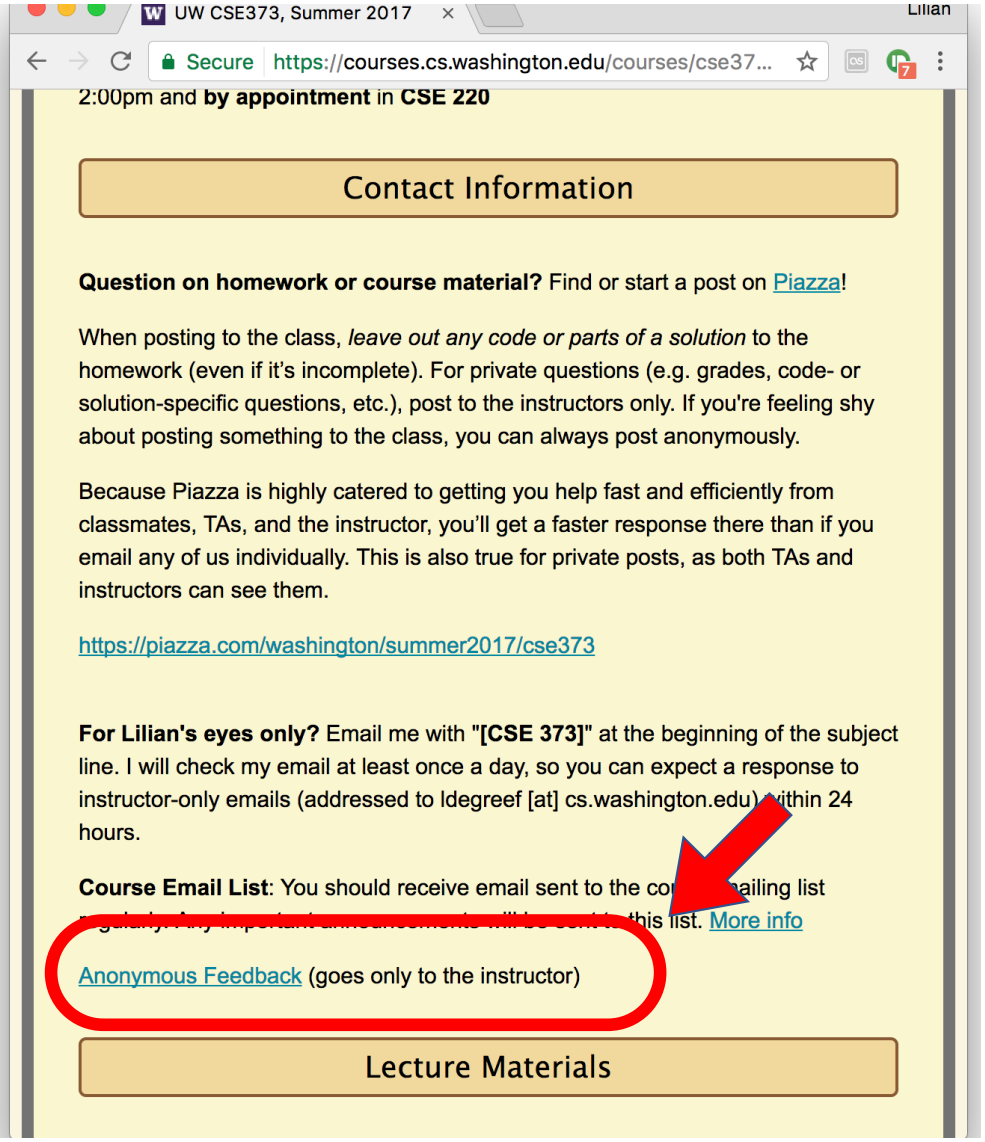
Instructor: Lilian de Greef
Quarter: Summer 2017

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) = i²`**
  - So probe sequence is:
    - 0th probe: `h(key) % TableSize`
    - 1st probe: `(h(key)+ 1) % TableSize`
    - 2nd probe: `(h(key)+ 4) % TableSize`
    - 3rd probe: `(h(key)+ 9) % TableSize`
    - …
    - i^{th} probe: `(h(key) + i²) % TableSize`

- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example #2

0	48
1	
2	5
3	55
4	
5	40
6	76

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)

Yikes! For all n, $(n^2+5)\%7$ is 0, 2, 5, or 6

↳ 47 is never placed in the table!

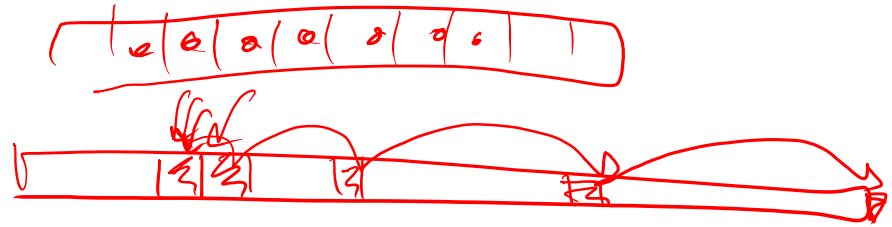
i^{th} probe: `(h(key) + i²) % TableSize`

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 ≠ 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 secondary clustering

- Can avoid secondary clustering with a probe function that depends on the key... aka double hashing!

Open Addressing: Double Hashing

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely 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: (h(key) + 2*g(key)) % TableSize
- 3rd probe: (h(key) + 3*g(key)) % tableSize
- …
- 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:

  - it must never evaluate to zero

  - must make sure all cells are probed

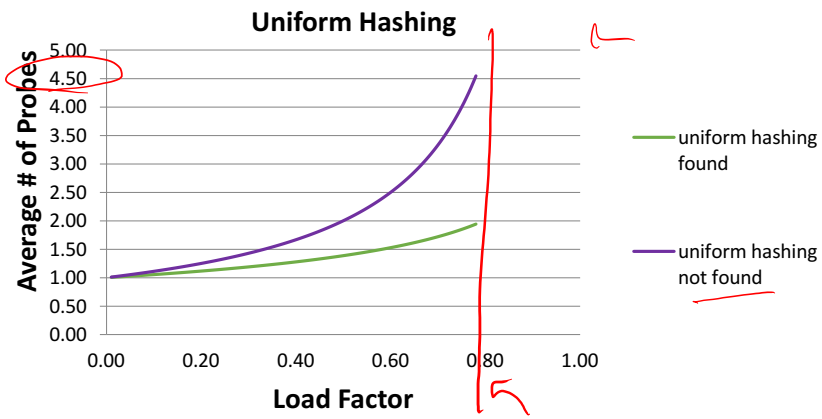
- Example of double hash function pair that works:
  - `h(key) = key % p`
  - `g(key) = q - (key % q)`
  - `2 < q < p`
  - `p` and `q` are prime

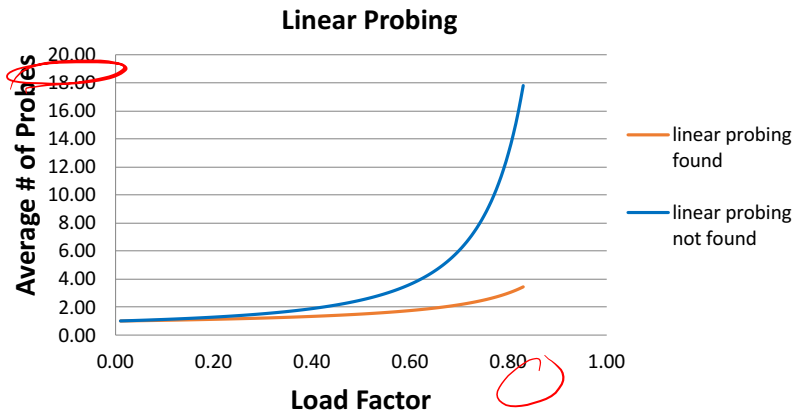
$x = 1$

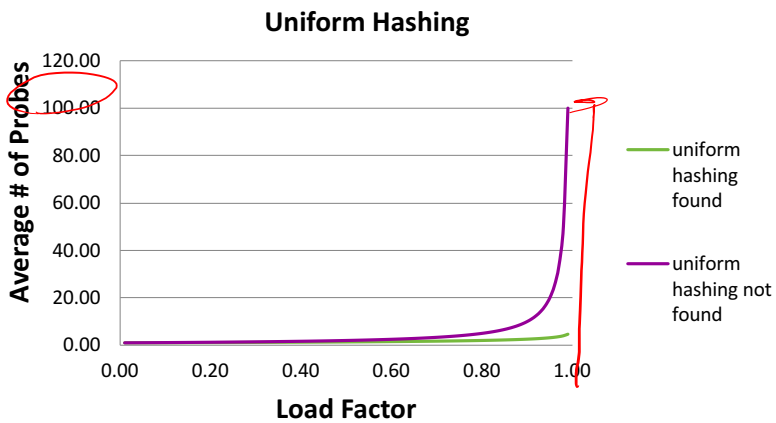
More Double Hashing Facts

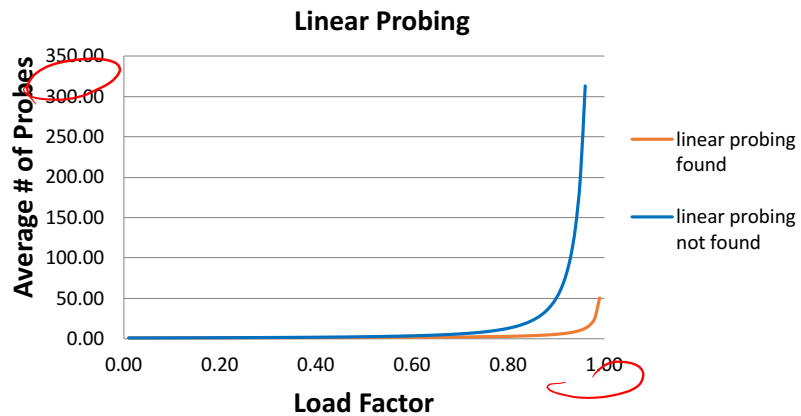
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`
- Non-trivial facts we won't prove:
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search (intuitive): $\frac{1}{1-\lambda}$
  - Successful search (less intuitive):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

index = hash value % table size

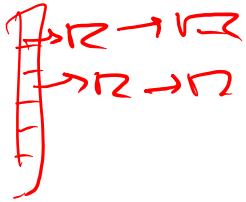
- What do we do if the table gets too full?

  Increase table size, copy elements over (like in array-based stacks/queues)

- How do we copy over elements?

  Redo hash (ie. rehash!)

Rehashing



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

0.5 ≤ $\boxed{\lambda < 1}$

we get to decide load factor (λ)
usually choose to keep low & reasoble (eg. $\lambda = 1$ or 1.5, or 2)

- "Too full" for Open Addressing / Probing

(if $\lambda \geq 1$, it breaks!)

Half-full is a good rule of thumb

Rehashing

- How big do we want to make the new table?
  - twice-as-big is a good thought, but the size won't be prime!
  - make size a prime close to twice-as big

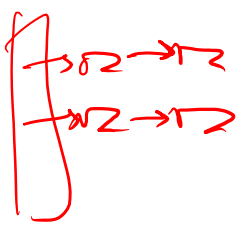
- Can keep a list of prime numbers in your code, since you likely won't grow more than 20-30 times (2^30 = 1,073,741,824)

Wrapping up Hash Tables

Trade-off btw sep. chaining & open address

speed vs space

- A hash table is a data-structure for dictionary ADT (storing (key, value) pairs)
- Some example uses of hash tables:
  - phone numbers (person's name, #)
  - passwords (encrypted pw, authentication)
  - spell checker (2 hash tables: (misspelled words, likely words) (correctly spelled words))



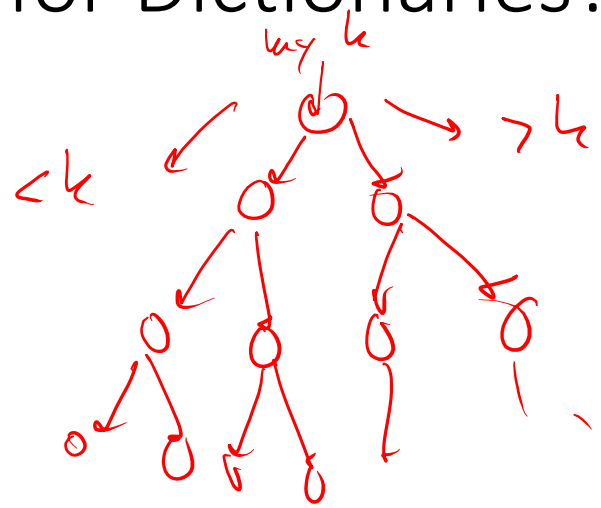
Another Data-Structure for Dictionaries?

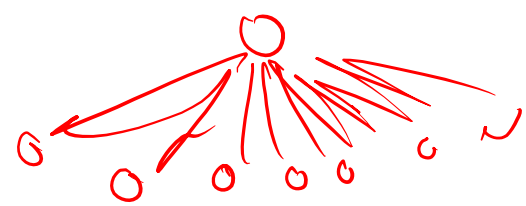
Dictionary meaning:

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

Dictionary operations:

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



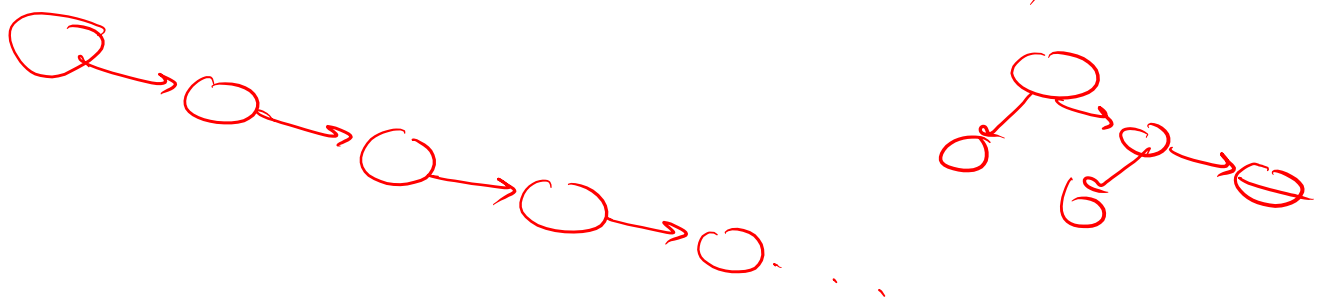


Trees!

Trees

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

Linked-list is actually a kind of tree! (I like think of it as "stick")



Tree terms

Root (tree) A

Leaves (tree) D E F I J K L M N

(For Node G:)

Children (node) H I

Parent (node) C

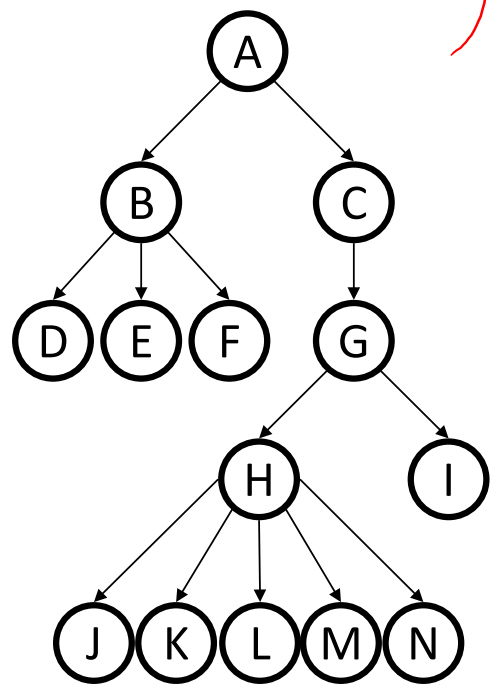
Siblings (node) (for node E): D F

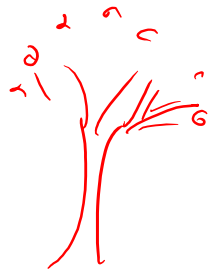
Ancestors (node) (for Node G): C A

Descendents (node) H I J K L M N

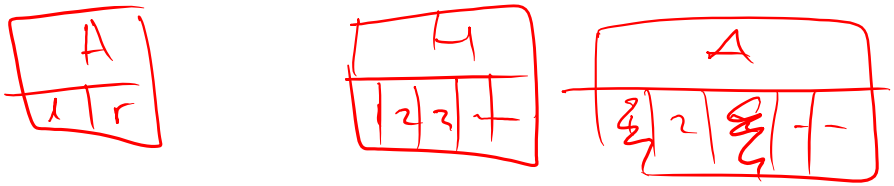
Subtree (node) left H (root) right I (root)

Tree **T**

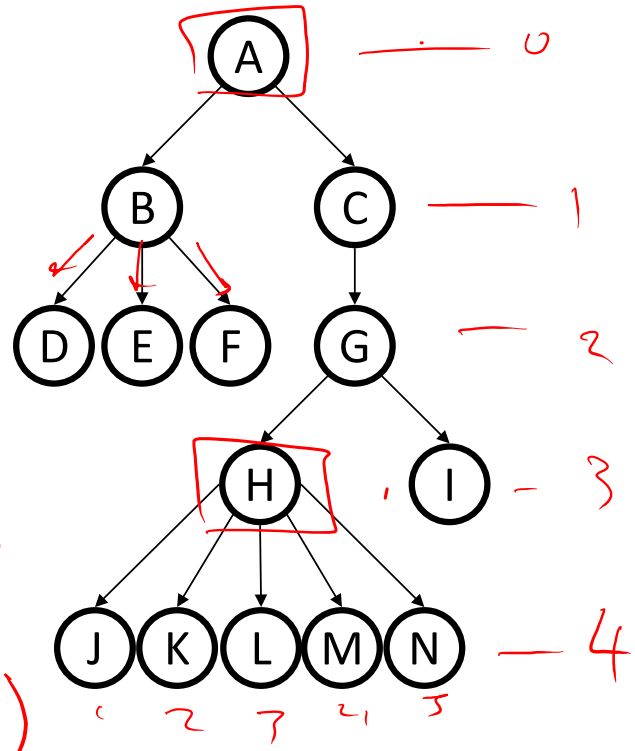




Tree terms



Tree **T**



Depth (node) For node G: 2 (# edges to root)

Height (tree) 4 (max # edges from root to leaf)

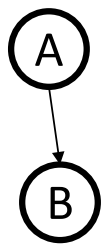
Degree (node) For node B: 3 (# children)

Branching factor (tree) 5 (max # of children at any node)

Practice with Height and Depth

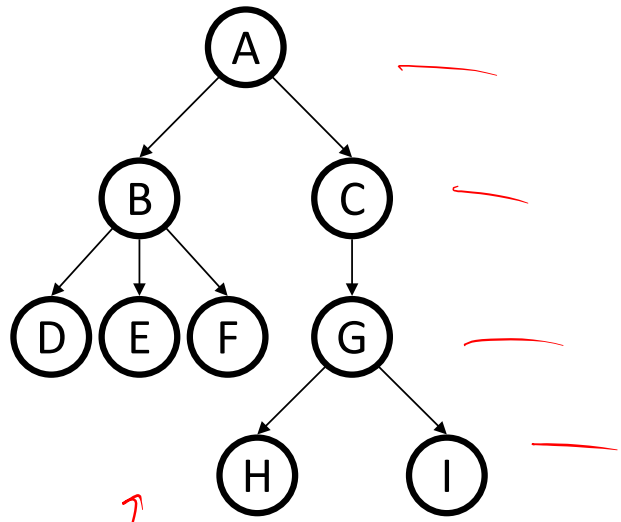
height = 0

depth(A): 0



height = 1

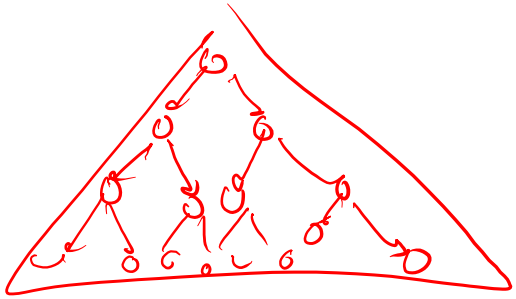
depth(B) = 1



height = 3

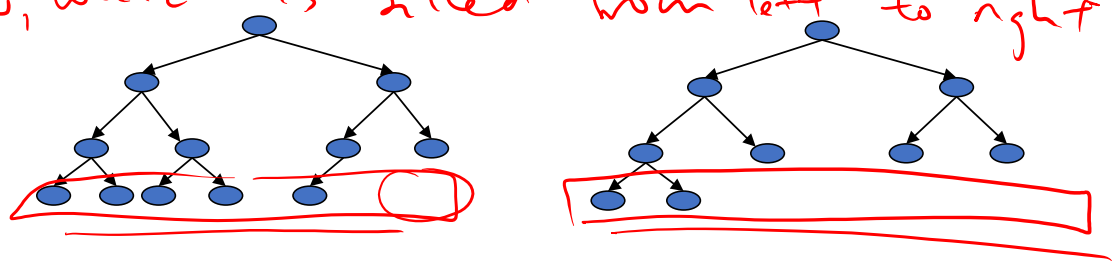
depth(F) = 2

Kinds of Trees



Certain terms define trees with specific structure

- **Binary tree**: Each node has at most 2 children (branching factor 2)
- ***n*-ary tree**: Each node has at most n children (branching factor n)
- **Perfect tree**: Each row is completely full
- **Complete tree**: Each row completely full except maybe the bottom row, which is filled from left to right



What is the height of a perfect binary tree with n nodes? $\log_2 n$

A complete 14-ary tree? $\log_{14} n$

More Tree Terms

- There are many kinds of trees

  binary tree, linked lists, ...

- There are many kinds of binary trees

  binary search tree, binary heaps

- A tree can be balanced or not
  - A balanced tree with *n* nodes has a height of $O(\log n)$
  - Different kinds of trees use different "balance conditions" to achieve this

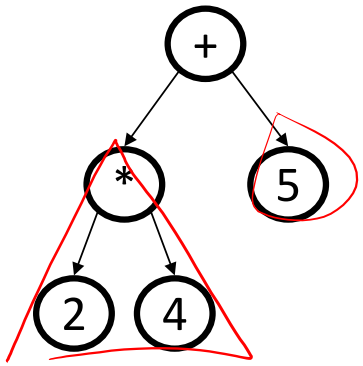
(Bonus Material) Cool Uses & Kinds of Trees!

Binary Search Tree - dictionaries and more

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

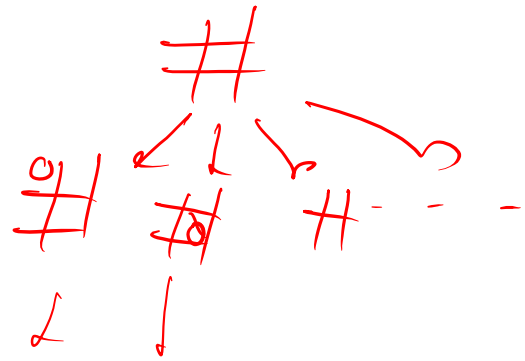
Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.

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!

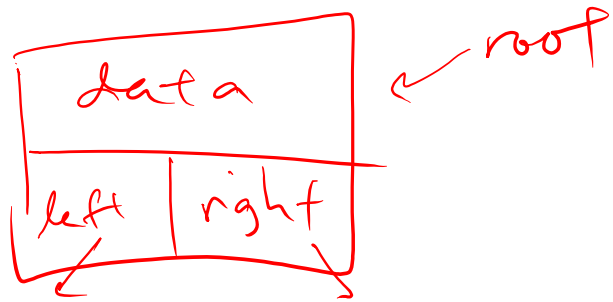
For now, focusing on generic and binary search trees (don't worry about the other ones listed here -- I just think they're cool and want to share!)

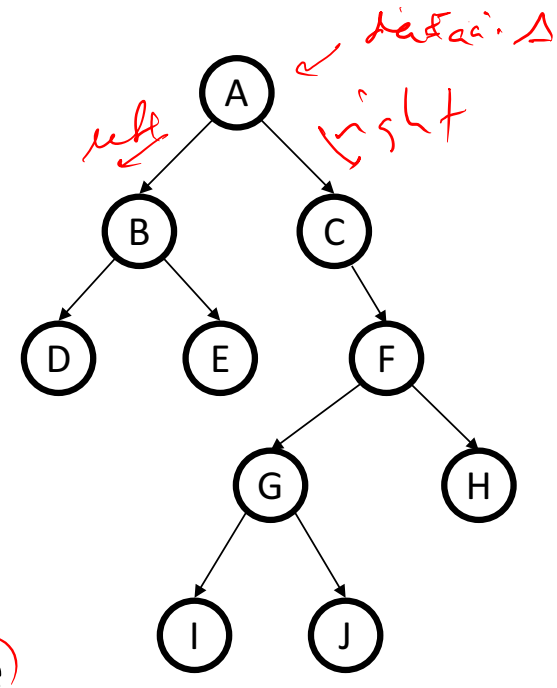
Binary Trees

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

- Binary tree is (member variables)
  - data
  - left subtree
  - right subtree

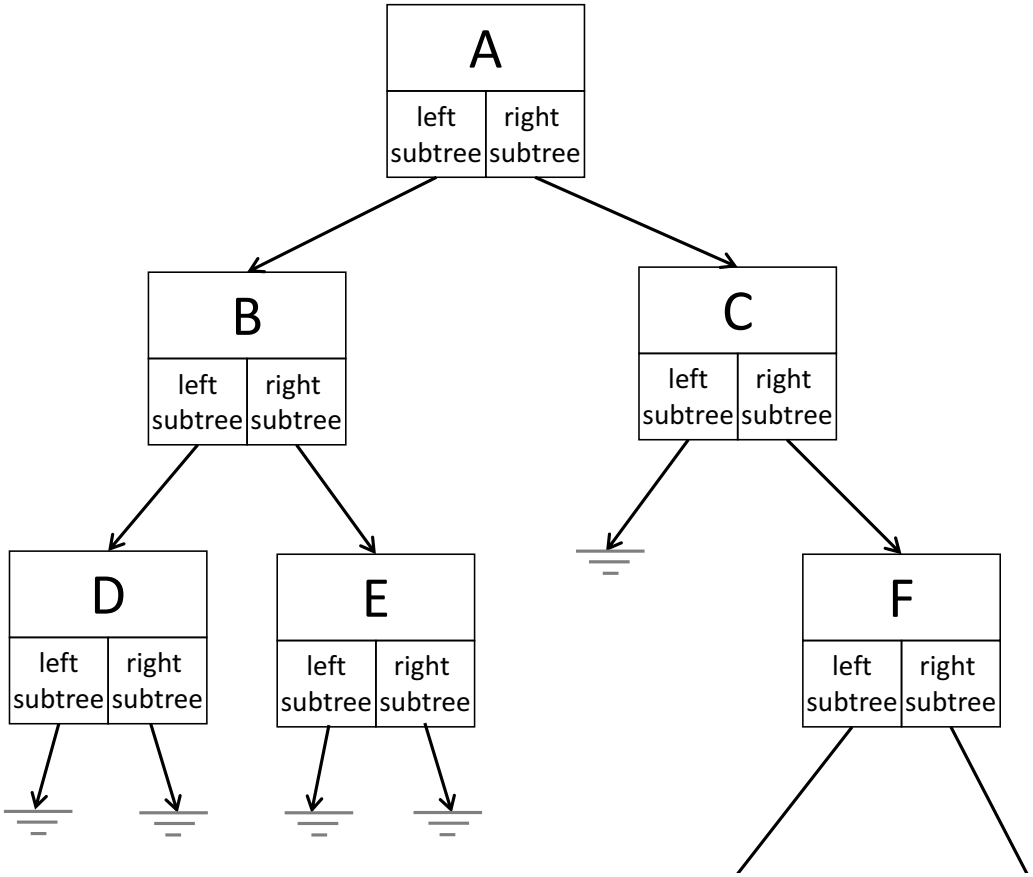
- Representation:

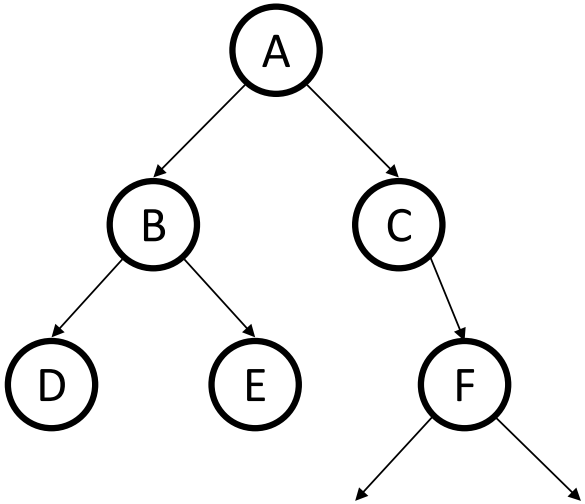




- For a dictionary, data will include a (key and a value)

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);
}
```

(Java method)

max(x, y)

= if x > y,
return x

if y > x
return y

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

B. It calculates the depth of the nodes.

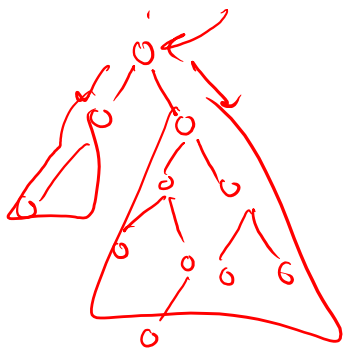
C. It calculates the height of the tree.

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

Practice time! What does the following method do?

when it gets to a leaf!

```
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. (can be sub-tree)

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