CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

MIDTERM 2 NOTES

- Midterm 2: take during a **2 hour block this Thurs/Fri** (in any time zone, including ones you are not in)
- Let me know by TONIGHT if you will be unable to take the exam in this time frame
- Can use self-created "cheat-sheet" (double-sided), but no other notes or resources
- Please no Piazza posts related to exam review Thurs/Fri

Remind me to RECORD

PREPARING/SUBMITTING YOUR EXAM

Options:

1) Print exam, write on exam, take pictures of exam
2) Look at exam on-screen, write answers on paper, take pictures of answers
3) Look at exam on-screen, type answers in word processor (draw diagrams by hand, take pictures and combine)
4) Previous option, but create diagrams on-screen (may take too much time if you're not familiar with a similar process)

In all cases: submit as a single PDF file on Moodle within 30 minutes of your end time

EXAM: WRITE DOWN START AND END TIME

11. Sign below (or print your name on your submission) to indicate that you will **abide by the Honor Code** in taking this examination:

_____

Regardless of how you submit your work, include the following information (include timezone too):

START TIME: _____

END TIME: _____

MY OFFICE HOURS THIS WEEK

Tuesday 4:30-6pm (can stay late)

Wednesday 8:30-9:30am (cannot stay late)

Will rotate through questions in the order people join and I will use the whiteboard (similar to Midterm 1 office hours).

You are welcome to just come and listen even if you don't have questions.

APR 28 OUTLINE

- Highlight important concepts from this course and other takeaways
- Review recap based on Google form
- Review problems

APR 28 OUTLINE

- **Highlight important concepts from this course and other takeaways**
- **Review recap based on Google form**
- **Review problems**

NOTES FOR FUTURE CS CLASSES

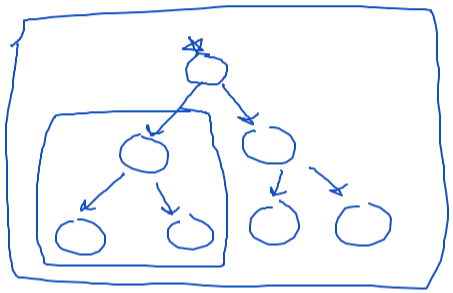
- **LinkedList Node is a recursive data structure like a BinaryTree**

- **Hash maps are dictionaries**

- **Java is an imperative language**
  - Use statements/commands to change state in a specific way

NOTES FOR FUTURE CS CLASSES

- **LinkedList Node is a recursive data structure like a BinaryTree**



- **Hash maps are dictionaries**

- **Java is an imperative language**
  - Use statements/commands to change state in a specific way

Pure Functional Programming

How do we think about computer code?

Option 1: more advanced reasoning techniques

- commands that change state in a specific way

Option 2: limit programming techniques

- don't change name/object relationships
- don't change object/value relationships
- known as "pure functional programming"

Creating Unchanging Objects

If we can't *change* an object, how can we insert in a heap/tree/etc?

Instead of "mutator" methods, create new objects

- Examples
  - heap.insert(x) → newHeap1 = heap.with(x)
  - heap.remove(x) → newHeap2 = heap.withOut(y)

TOP DOWN DESIGN

- We haven't practiced this too much, but I wanted to highlight this idea as something to continue working on in the future
- Start with "main" and pretend you have all the methods working (develop a "wish list" of methods and data structures)
- For each method or data structure you need, do the same thing!
- When you finally reach simple methods, start implementing them an work your way up
- Overall: recursive approach to design ☺

OBJECT-ORIENTED DESIGN

Goals

- **Robustness**
  - Capable of handling unexpected inputs
  - Gracefully recover from errors
  - Exception handling is a concrete way we can do this in Java
- **Adaptability**
  - Be able to evolve and respond to changing conditions/data
  - Argument passing is a concrete way to do this in Java
- **Reusability**
  - Code should work in a variety of different situations
  - Implementing an interface is a concrete way to do this in Java

See Chap 2.1 for more info

OBJECT-ORIENTED DESIGN

Principles

- **Abstraction**
  - Developing a mathematical model of a structure or algorithm
  - Creating an interface is an example in Java
- **Encapsulation**
  - Do not need to reveal inner workings – users only need to know the method signatures
  - Class structure (including public/private) aids encapsulation
- **Modularity**
  - Components can be implemented and tested separately, then work together smoothly
  - Debug data structure before using it or tying it to a specific application

See Chap 2.1 for more info

DESIGN PATTERNS

- **Recursion**
  - Binary search
  - Binary trees
  - Quick sort
  - Merge sort
- **Divide and conquer**
  - Quick sort
  - Merge sort
- **Greedy method**
  - Kruskal's algorithm

See Chap 2.1 for more info

THOUGHTS ABOUT CS AFTER GRADUATION…

Ask me after this week!

APR 28 OUTLINE

- Highlight important concepts from this course and other takeaways
- **Review recap based on Google form**
- Review problems

UNDERSTAND WELL

- Queues
- Most sorting algorithms
- Heaps and priority queues
- Binary Trees
- Graphs
- Huffman coding

NEEDS MOST REVIEW

- Abstract Data Type (ADT)
- Graphs
- Recursive aspects of binary trees
- Unbalanced trees and rotations
- Runtime
- Hash maps and different types of probing/collision handling
- Sets and Kruskal's algorithm
- Merge sort
- Radix sort

GRAPH REVIEW

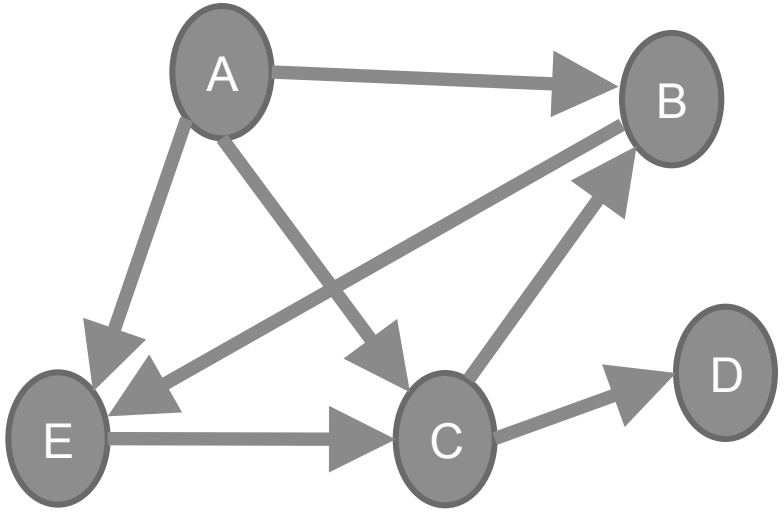
ADT (ABSTRACT DATA TYPE)

- Mathematical model of a data structure
- Says what operations should be possible
- Does not say how those operations are accomplished

- In practice: you can think of this as an interface with an understood description

ADT EXAMPLE: GRAPH

Method	Returns
`vertices()`	{A, B, C, D, E}
`numVertices()`	5
`numEdges()`	7
`outDegree(C)`	2
`inDegree(B)`	2
`outgoingEdges(A)`	{E, C, B}
`incomingEdges(B)`	{A, C}



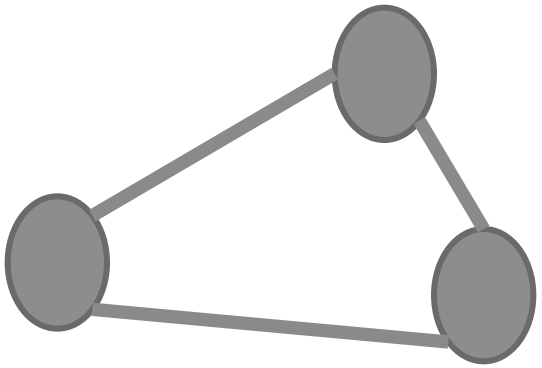
ADT EXAMPLE: GRAPH

```
/**
 * Simplified Graph interface
 */
public interface Graph {

    List<Vertex> vertices();

    int numVertices();

    Vertex insertVertex(String name);

    void insertEdge(Vertex u, Vertex v);

    boolean hasEdge(Vertex u, Vertex v);

    List<Vertex> outgoingEdges(Vertex v);

    List<Vertex> incomingEdges(Vertex v);
}
```

GRAPHS

Graphs (aka networks) represent relationships between pairs of objects.

Vertices (aka nodes) are the objects. (Singular: vertex)
Edges (aka links) are the relationships.



Note: these are graph theory graphs, not charts or plots.

NOTATION

A graph *G* is a set of vertices *V* and a set of edges *E*:

$$G = (V, E)$$

Each edge is a pair of vertices:

$$(u, v) \in V$$

The total number of vertices in a graph is denoted *n*:

$$|V| = n$$

The total number of edges in a graph is denoted *m*:

$$|E| = m$$

EDGE TERMINOLOGY

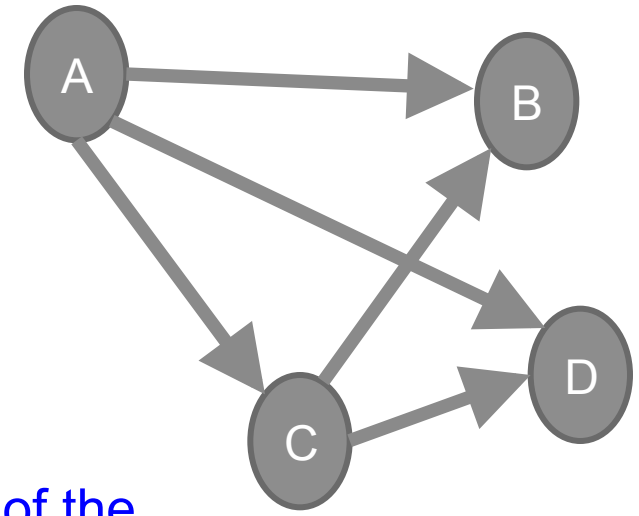
Two vertices are *adjacent* if they are joined by an edge.

Adjacent vertices are also known as *neighbors*.

IN-DEGREE AND OUT-DEGREE

Identify the ***in-degree*** and ***out-degree*** for each of the nodes in the below graph:

Node	in-degree	out-degree
A	0	3
B	2	0
C	1	2
D	2	0



Q: What do you notice about the sums of the in-degrees and out-degrees?

A: They are the same and equal to the number of edges! Because each edge has one origin and one destination.

CYCLES

A ***cycle*** is a path that starts and ends at the same vertex and has at least one edge. A ***directed cycle*** is the same for a directed graph.

A directed graph is ***acyclic*** if it has no directed cycles.

(A ***tree*** is a special type of acyclic graph.)

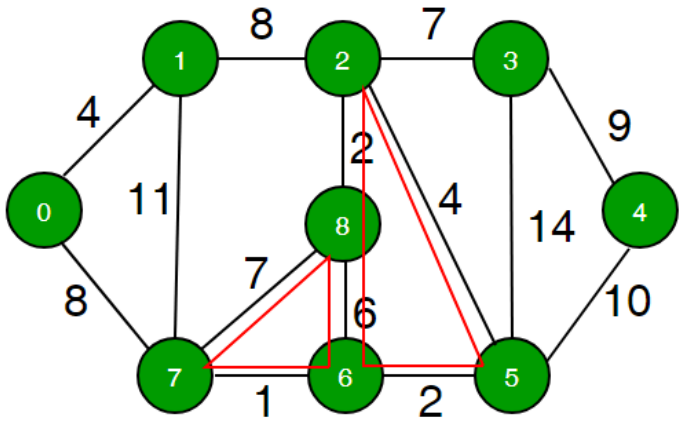
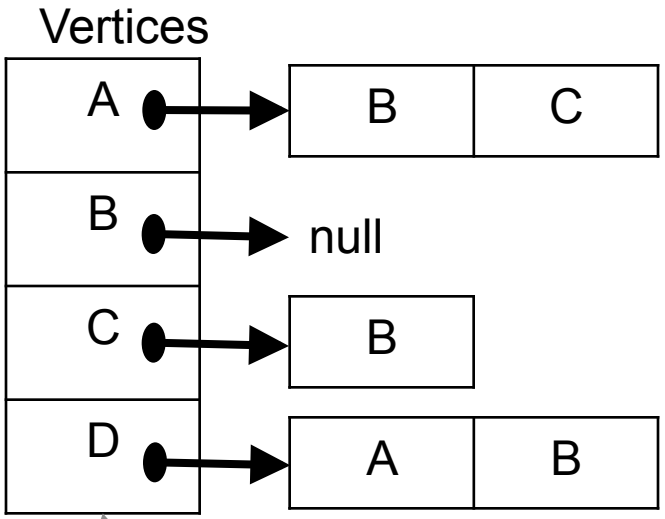
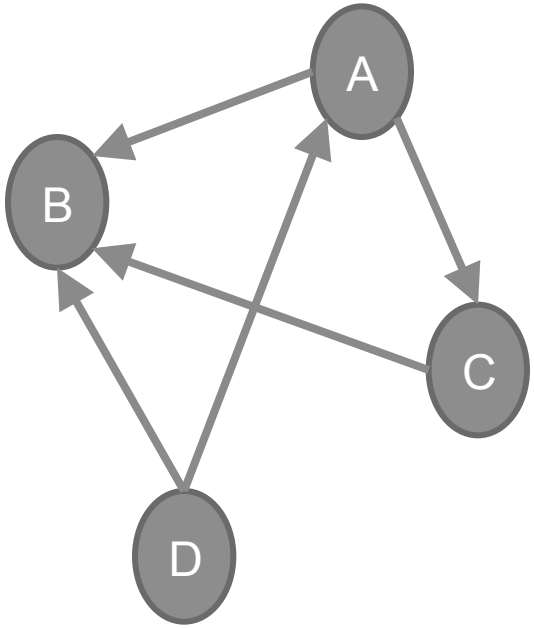


Image: GeeksforGeeks

GRAPH ADJACENCY LIST REPRESENTATION





Each Vertex contains a list of destination edges

One instance variable: list of Vertices

See section 14.2 of the book for more info!

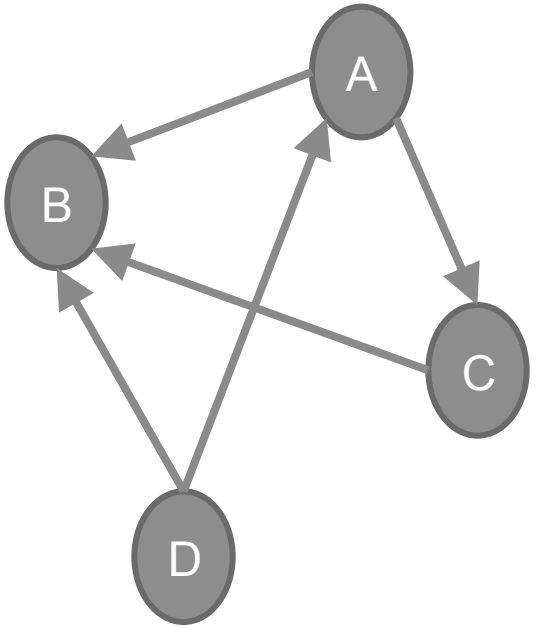
GRAPH ADJACENCY LIST RUNTIMES

Let *n* be the number of vertices.

`List<Vertex> vertices()`	O(1)
`int numVertices()`	O(1)
`Vertex insertVertex(elem)`	O(1)
`void insertEdge(u,v)`	O(1)
`boolean hasEdge(u,v)`	O(n)
`List<Vertex> outgoingEdges(v)`	O(1)
`List<Vertex> incomingEdges(v)`	$O(n^2)$

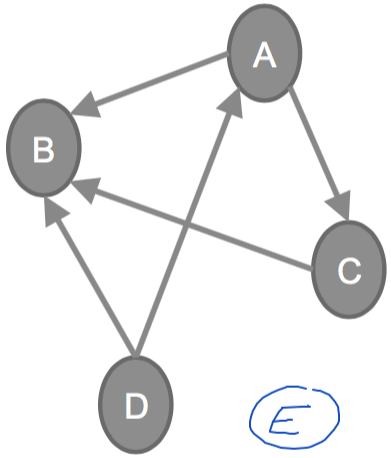
GRAPH ADJACENCY MATRIX REPRESENTATION

	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	1	0	0
D	1	1	0	0



GRAPH ADJACENCY MATRIX REPRESENTATION

	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	1	0	0
D	1	1	0	0

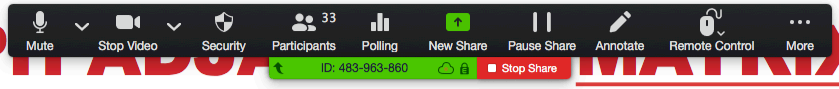


GRAPH ADJACENCY MATRIX RUNTIMES

Let *n* be the number of vertices.

`List<Vertex> vertices()`	O(1)
`int numVertices()`	O(1)
`Vertex insertVertex(elem)`	O(1)
`void insertEdge(u,v)`	O(1)
`boolean hasEdge(u,v)`	O(1)
`List<Vertex> outgoingEdges(v)`	O(n)
`List<Vertex> incomingEdges(v)`	O(n)

GRAPH ADJ MATRIX RUNTIMES



Let *n* be the number of vertices.

`List<Vertex> vertices()`	O(1)
`int numVertices()`	O(1)
`Vertex insertVertex(elem)`	O(1)
`void insertEdge(u,v)`	O(1)
`boolean hasEdge(u,v)`	O(1)
`List<Vertex> outgoingEdges(v)`	O(n)
`List<Vertex> incomingEdges(v)`	O(n)

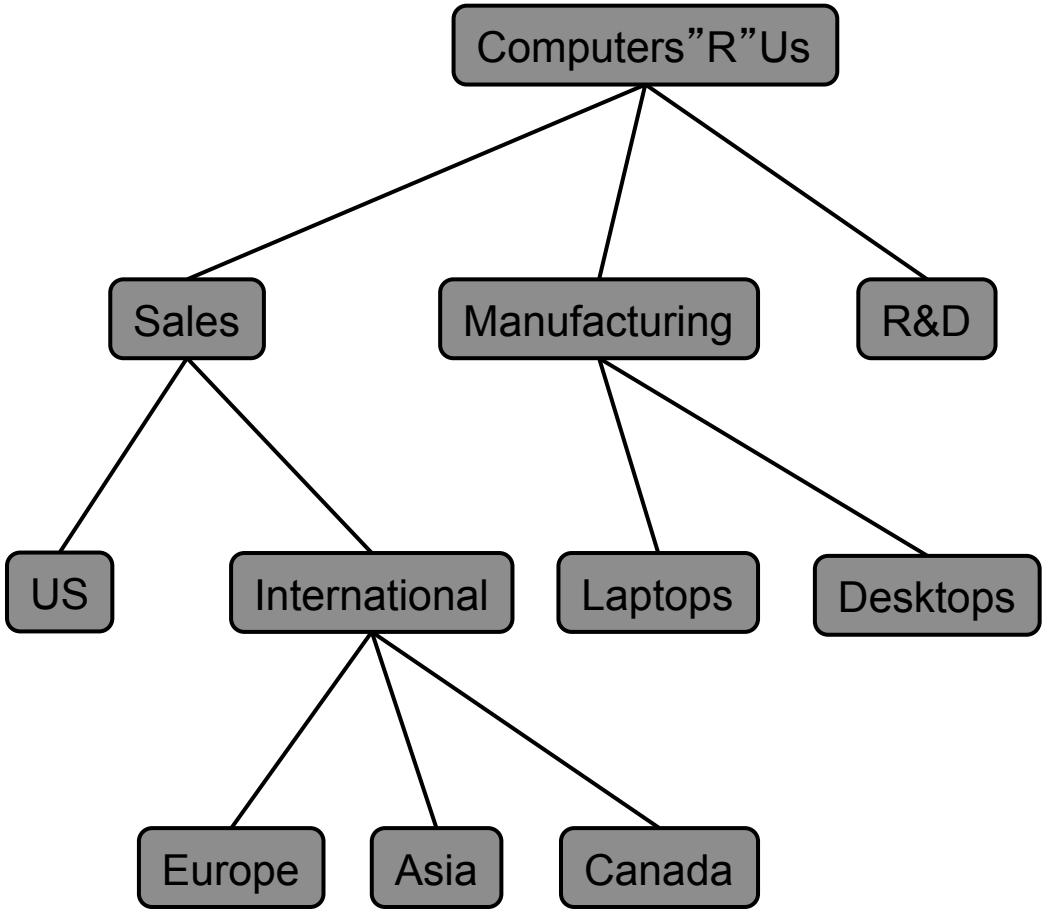
arraylist

array: $O(n^2)$

TREE REVIEW

TREE DATA STRUCTURE

- **Trees are acyclic graphs**
- **Nodes/vertices have a parent-child relationship**



TERMINOLOGY

Root: node with no parent (caveat, all nodes are roots of their subtree)

- A

Leaf node: node with no children:

- E, I, J, K, G, H, D

Internal node: node with at least one child

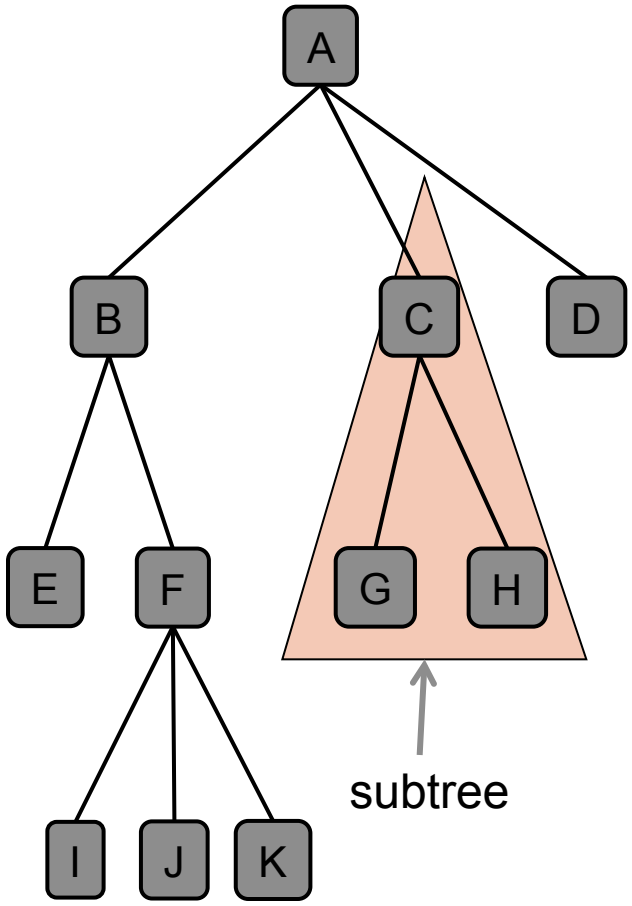
- A, B, C, F

Parent / Child relationships: two nodes connected by an edge. The node closer to the root is the parent.

- E.g., B (parent) and F (child)

Ancestor / Descendent relationships: ancestors of node X lie on the path from the root to X

- E.g., B (ancestor) and J (descendent)



TERMINOLOGY

Depth of a node: length of the path (num edges) from the root to that node

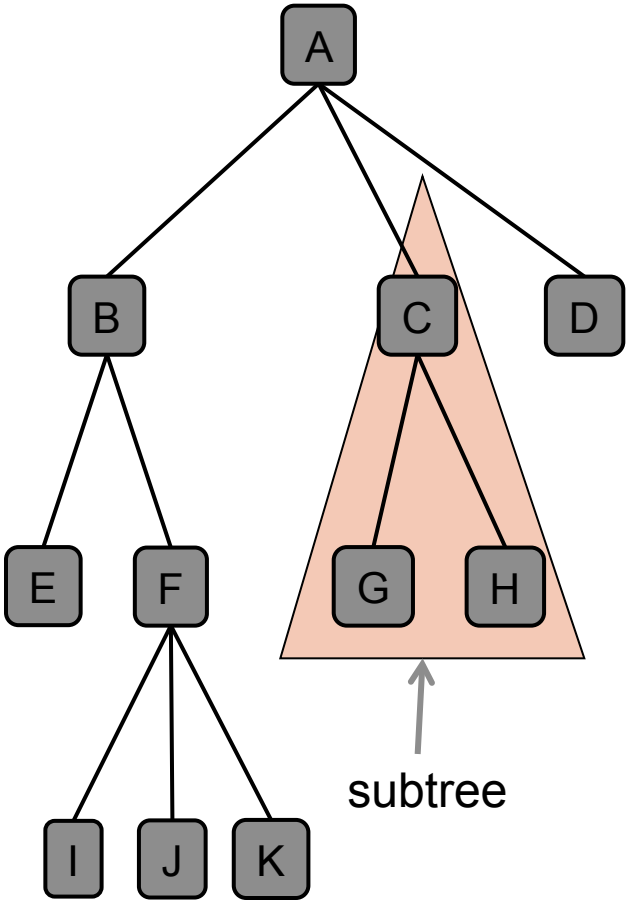
- e.g., depth of F = 2

Height: the maximum depth in the tree

- the height is 3

Subtree: a tree consisting of a node and its descendants

- the orange subtree with root C

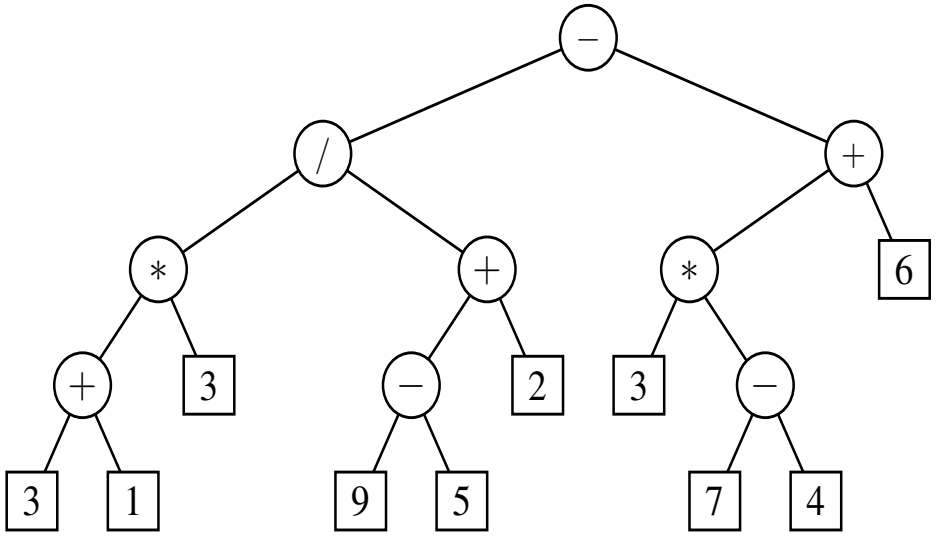


BINARY TREE

An ordered tree with every node having at most two children – left and right

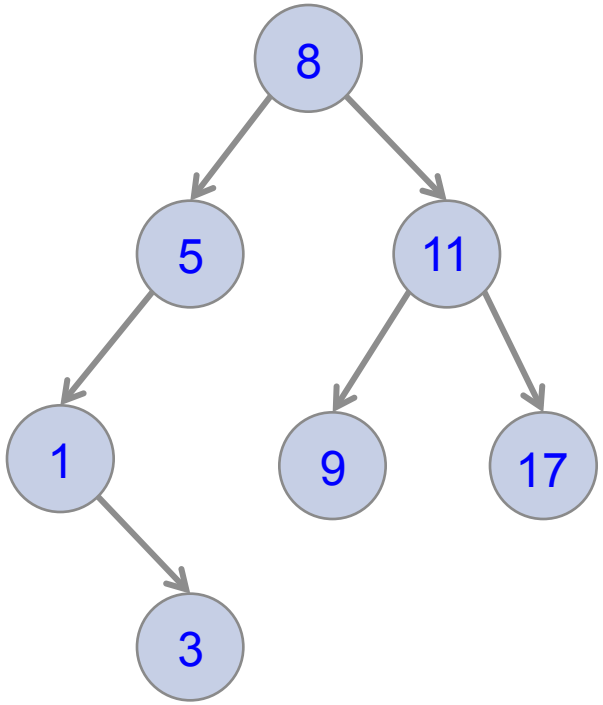
Recursive definition:

- base case: empty tree
- recursion: root with two subtrees



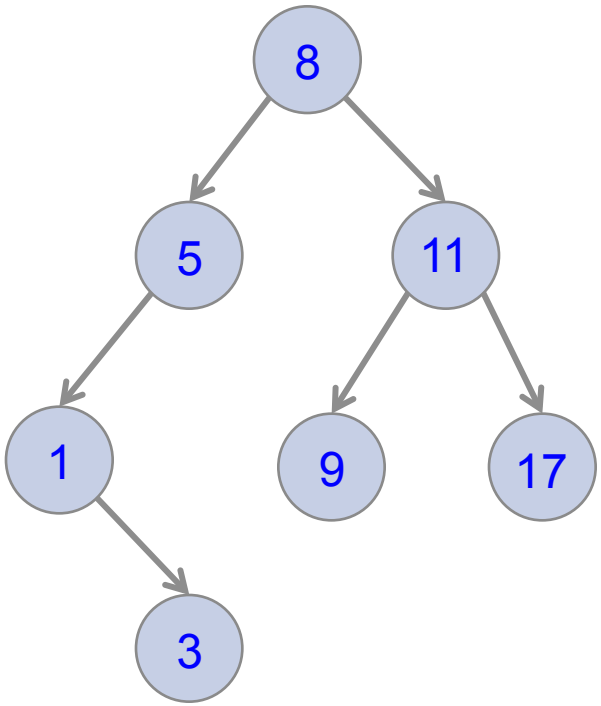
BINARY SEARCH TREE (BST)

- **Left child less than parent**
- **Right child greater than parent**
- **Typically no duplicates (alternative: store number of occurrences of each value with the node)**



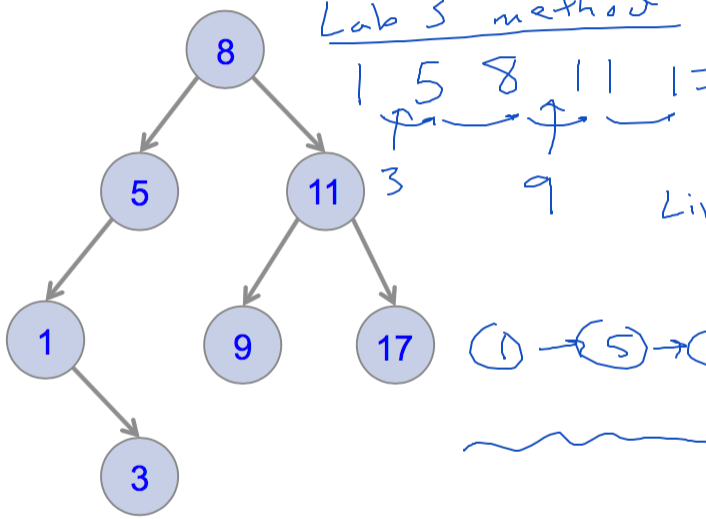
INSERTION SORT WITH TREES

Input: {8, 11, 5, 17, 1, 9, 3}



INSERTION SORT WITH TREES

Input: {8, 11, 5, 17, 1, 9, 3}



Lab 3 method

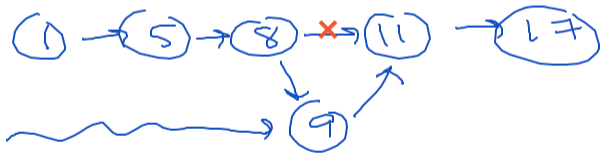
1 5 8 11 17

3 9

out of place

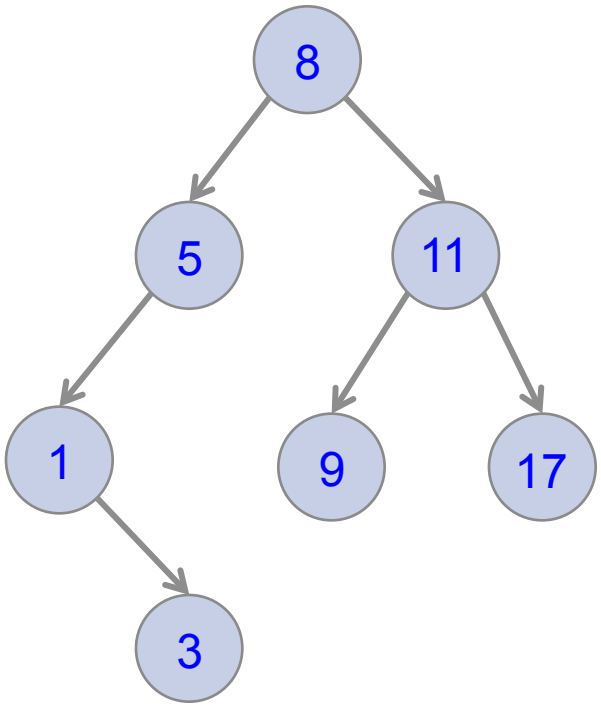
$O(n^2)$

Linked list



INSERTION SORT WITH TREES

Input: {8, 11, 5, 17, 1, 9, 3}



d = number of levels or number of comparisons

$$d \approx \log(n)$$

Expected runtime of insertion sort with binary tree:

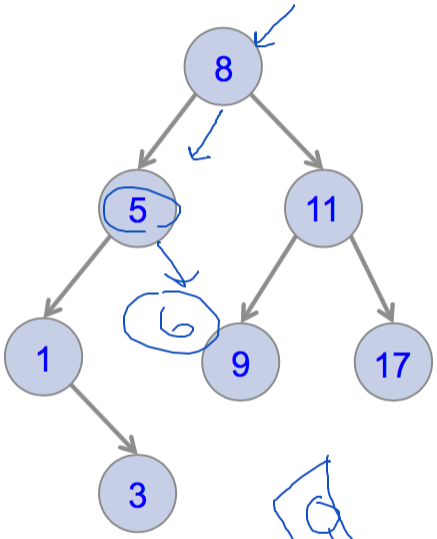
$$O(n \log n)$$

Worst case?

- Note: we sill need to run an “in-order” traversal afterward

INSERTION SORT WITH TREES

Input: {8, 11, 5, 17, 1, 9, 3}



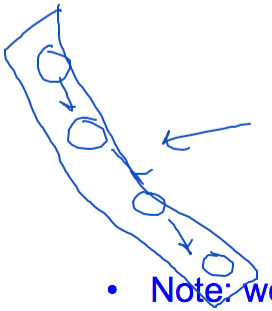
d = number of levels or number of comparisons

$$d \approx \log(n)$$

insert 6

Expected runtime of insertion sort with binary tree:

$$O(n \log n)$$



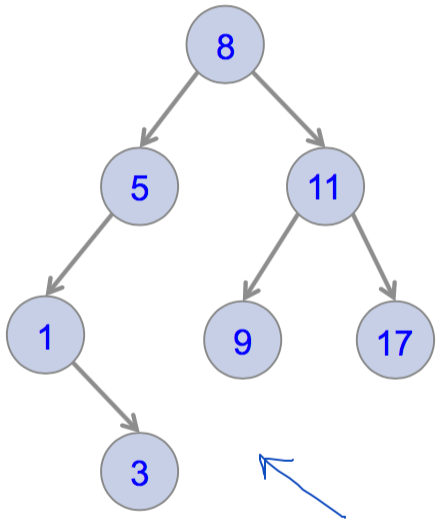
Worst case?

$O(n^2)$

- Note: we sill need to run an "in-order" traversal afterward

INSERTION SORT WITH TREES

Input: {8, 11, 5, 17, 1, 9, 3}



d = number of levels or number of comparisons

$$d \approx \log(n)$$

Expected runtime of insertion sort with binary tree:

$$O(n \log n) + O(n)$$

Worst case?

- Note: we sill need to run an “in-order” traversal afterward

REMOVE

```
boolean remove(E element);
```

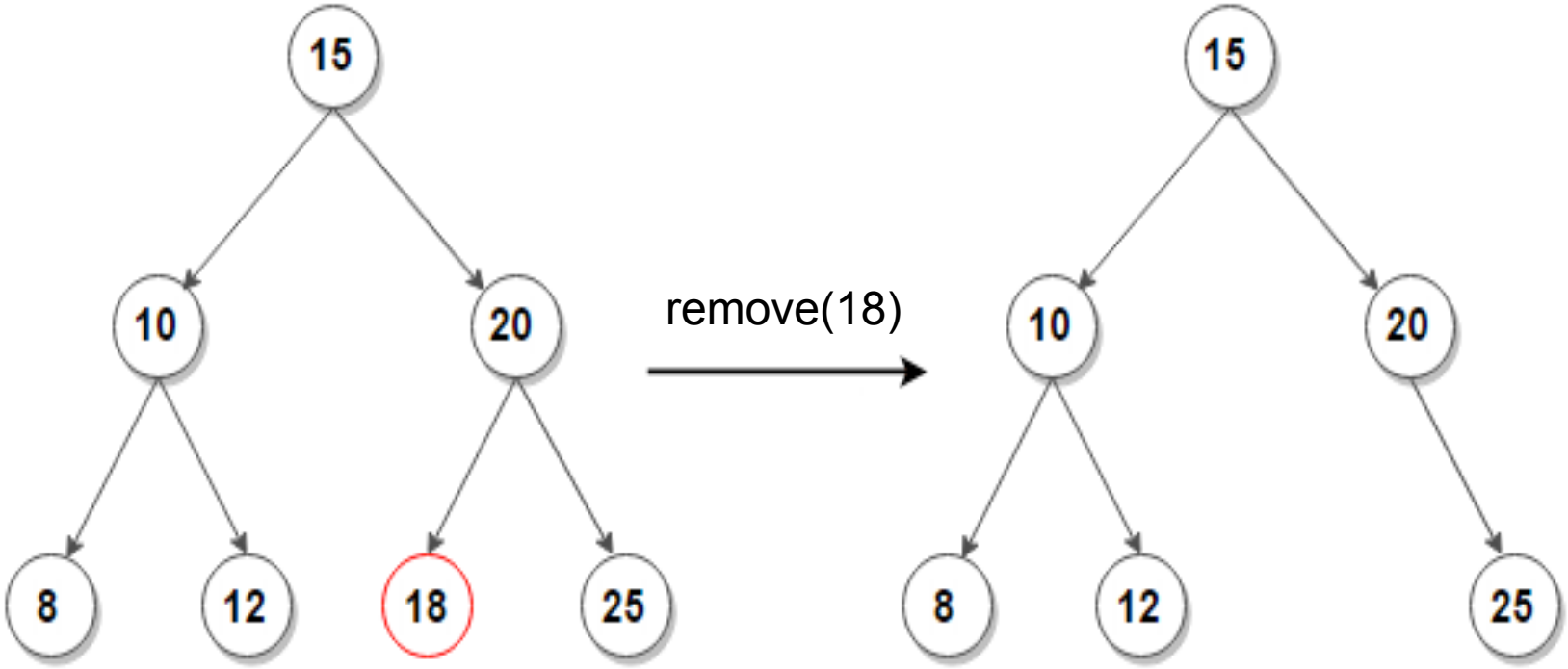
returns true if element existed and was removed and false otherwise

Cases

- element not in tree
- element is a leaf
- element has one child
- element has two children

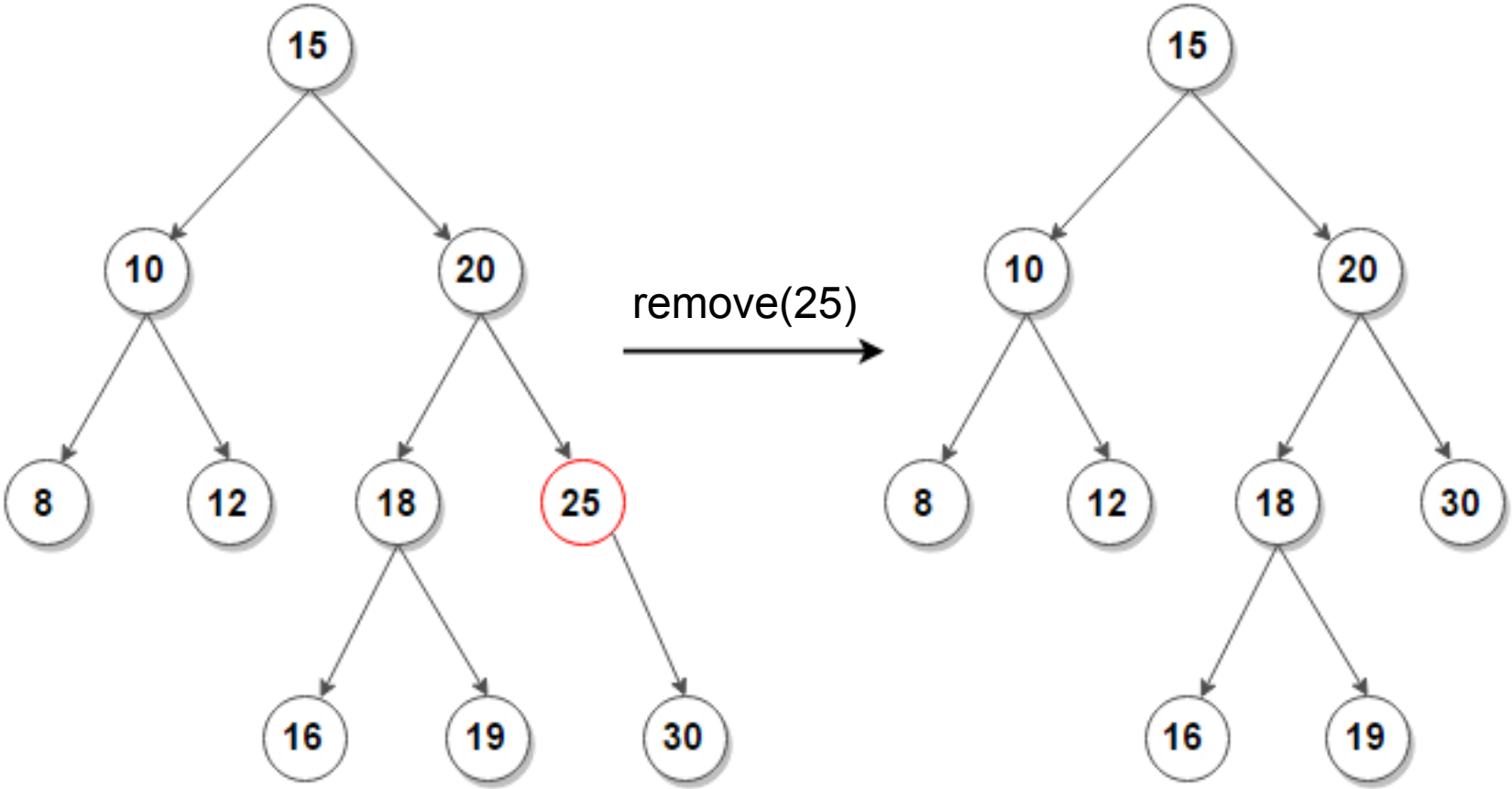
LEAF

Just delete



ONE CHILD

Replace with child



TWO CHILDREN

Replace with in-order predecessor or in-order successor

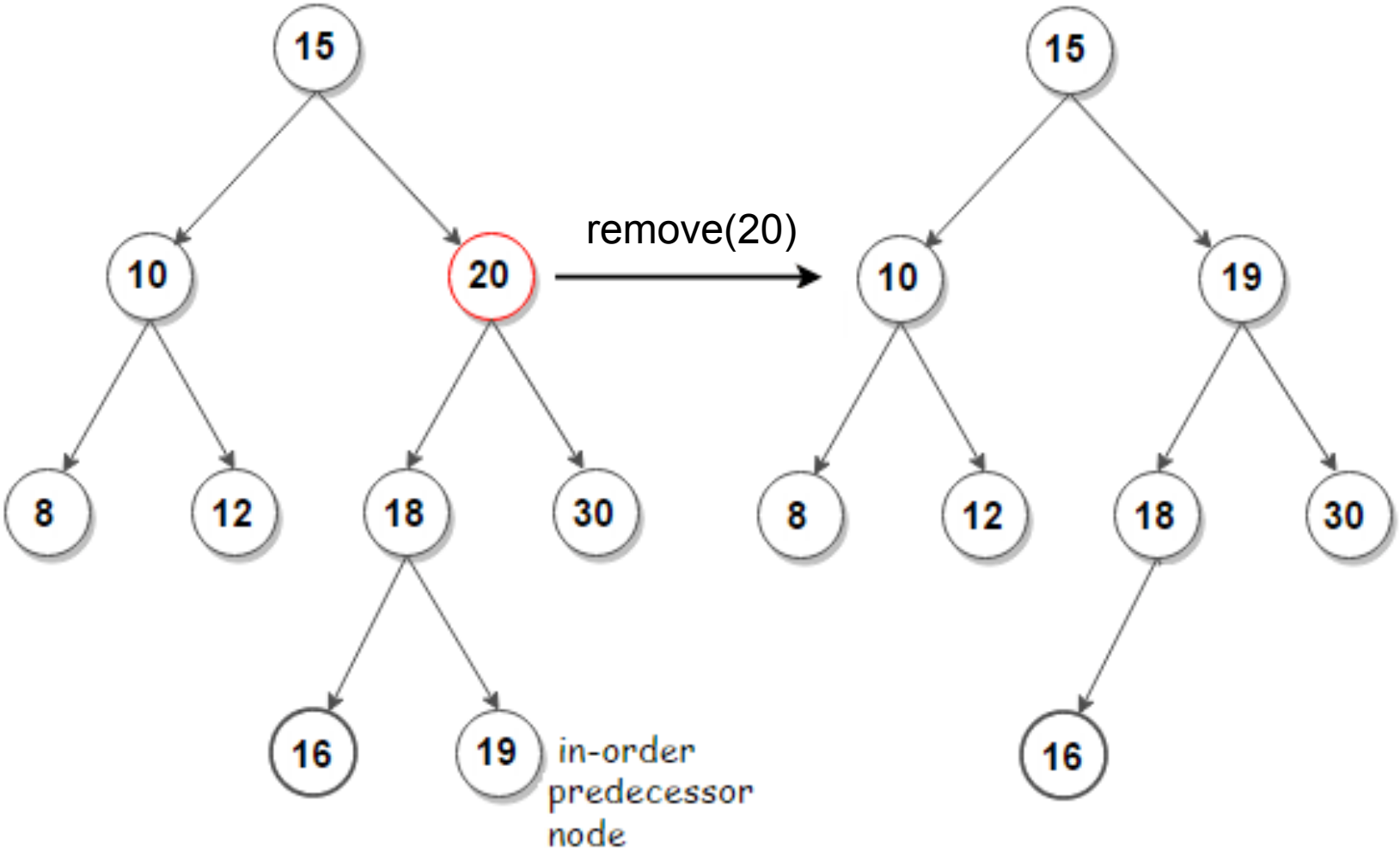
in-order predecessor

- rightmost child in left subtree
- max-value child in left subtree

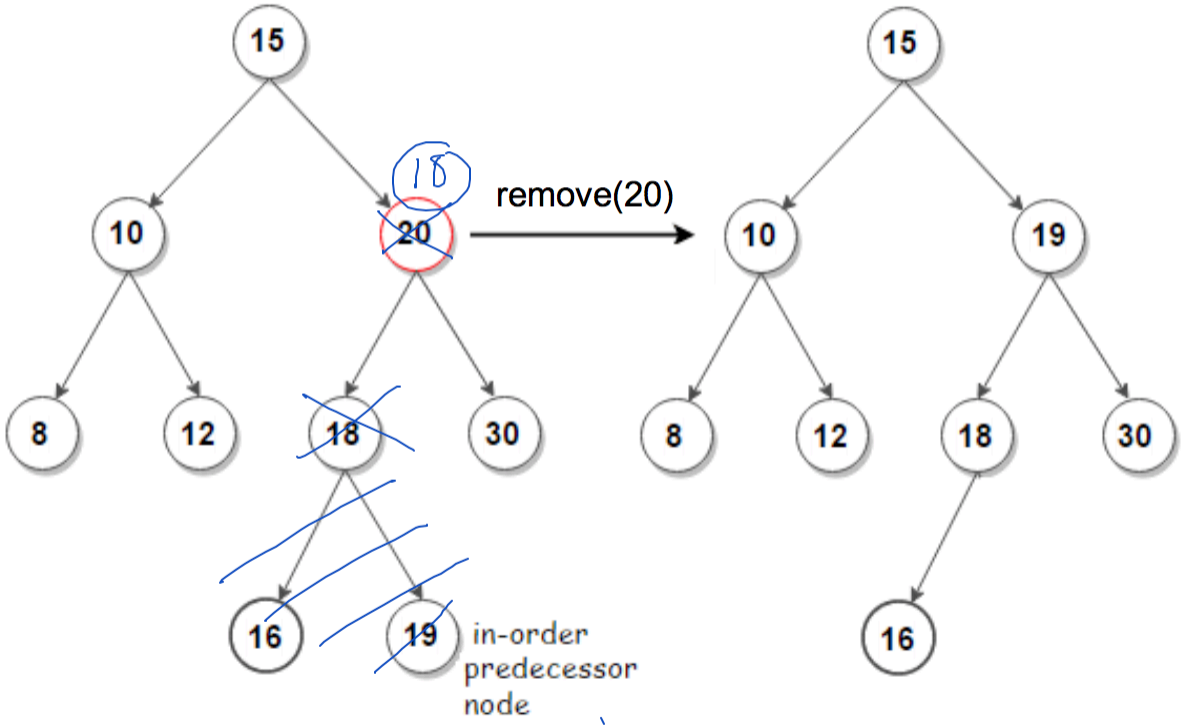
in-order successor

- leftmost child in right subtree
- min-value child in right subtree

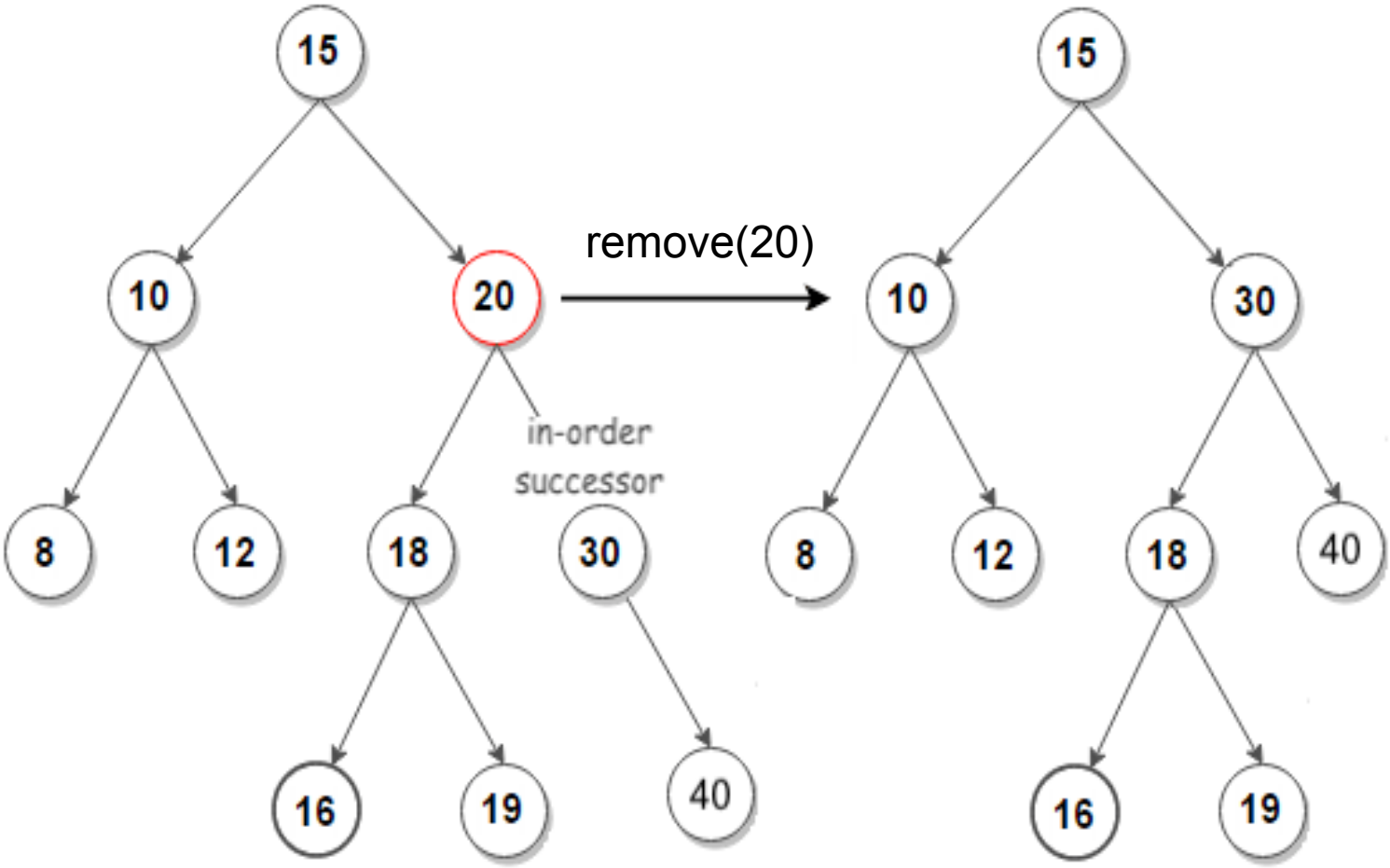
REPLACE WITH PREDECESSOR



REPLACE WITH PREDECESSOR

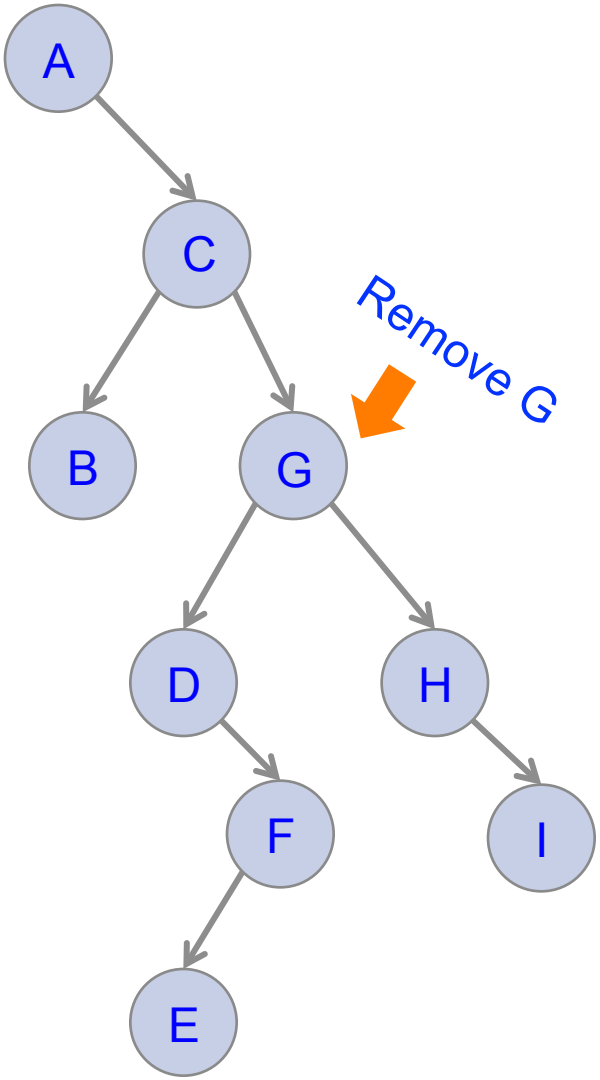


REPLACE WITH SUCCESSOR



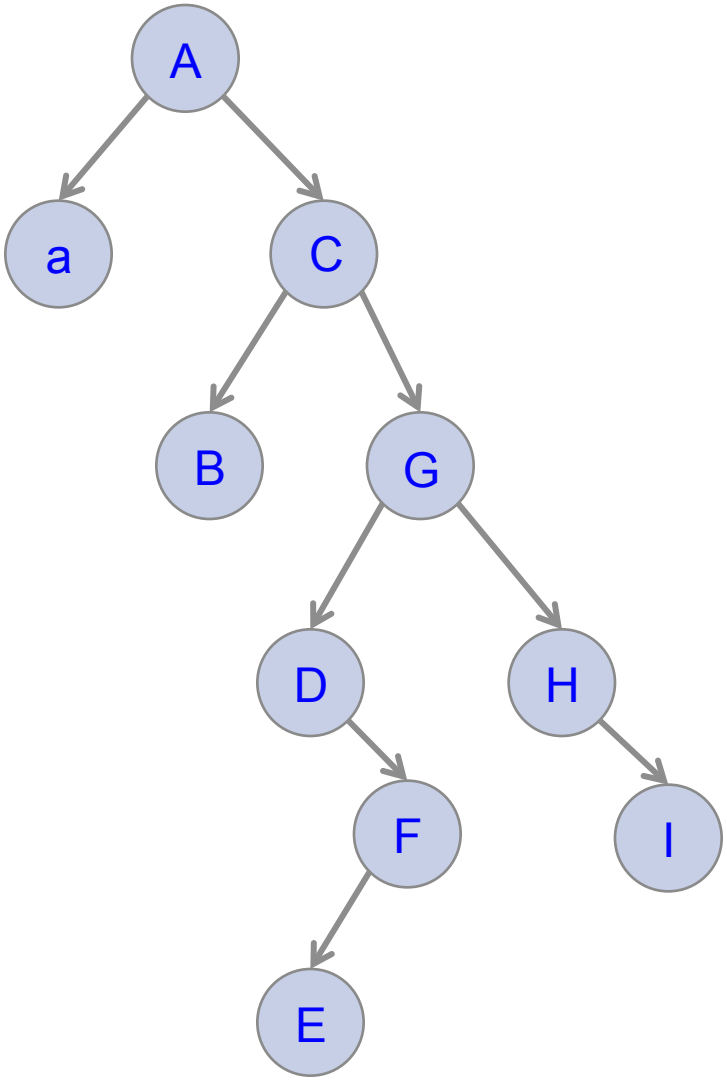
Note: this is a special case when successor is also child (replace entire node, just just data)

DELETION EXAMPLE: G

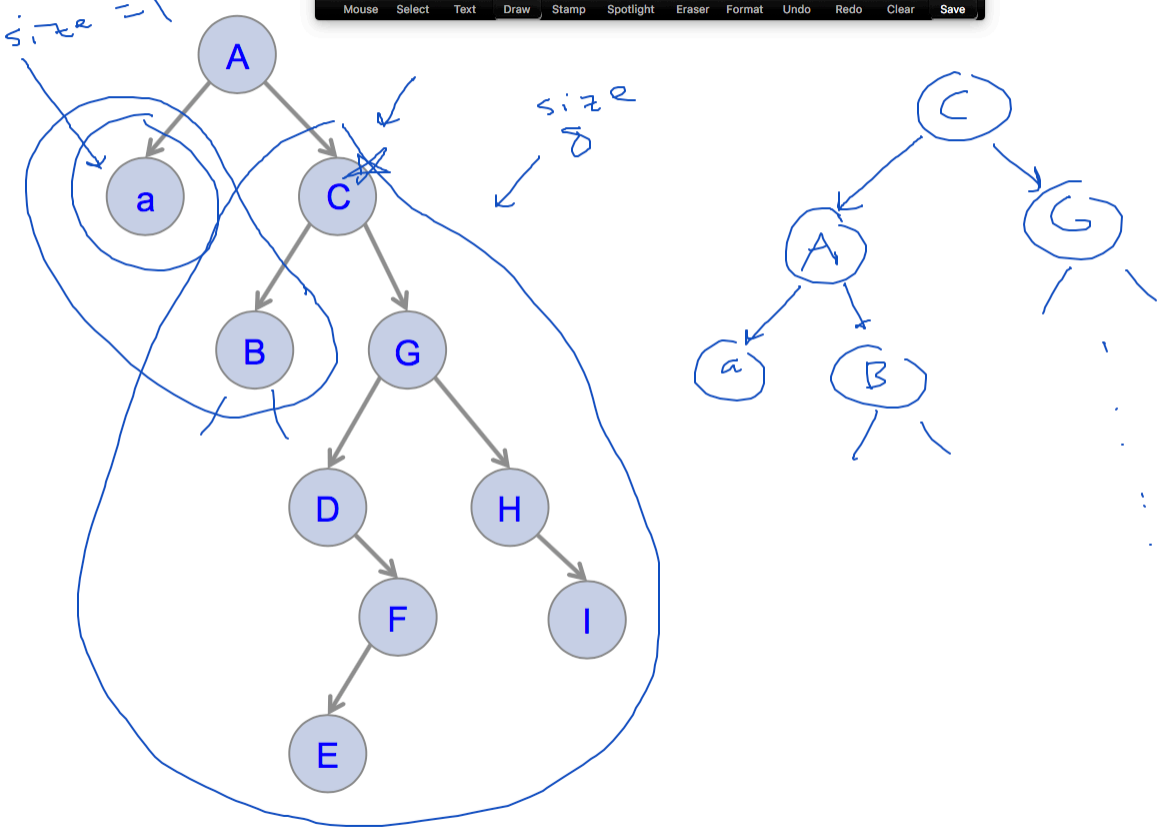


Try as an exercise!

REBALANCING EXAMPLE



REBALANCING AVL



HASH TABLE REVIEW

MAP

A searchable collection of key-value pairs

Multiple entries with the same key are not allowed

Also known as dictionaries, hash tables, etc

MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

`insert(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`

`remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null

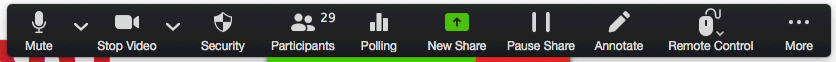
`size()`, **`isEmpty()`**

`entrySet()`: return an iterable collection of the entries in `M`

`keySet()`: return an iterable collection of the keys in `M`

`values()`: return an iterable collection of the values in `M`

MAP ADT (MANY WAYS OF DEFINING)



get

lookup(k): if the map M has an entry with key k, return its associated value; else, return null

put

insert(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, replace old value with v and return old value associated with k

remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null

size(), **isEmpty()**

entrySet(): return an iterable collection of the entries in M

keySet(): return an iterable collection of the keys in M

values(): return an iterable collection of the values in M

EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
private class TableRow {
    int key;
    String value;
    TableRow(int key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
/**
 *  Return the default position (index)
 *  where this key is stored
 */
private int hash(int key) {
    return key % rows.length;
}

/**
 *  Locates the position (index) where the
 *  specified key can be found, or where it
 *  should be inserted if it is not already
 *  in the table
 */
private int locate(int key) {
    int pos = hash(key);

    // this is the linear probing part
    while (rows[pos] != null && rows[pos].key != key) {
        pos = (pos + 1) %rows.length;
    }
    return pos;
}
```

HASH TABLE RUNTIMES

Let *n* be the number of elements in the hash table

	Hash Expected	Hash Worst
lookup	$O(1)$	$O(n)$
insert	$O(1)$	$O(n)$
remove	$O(1)$	$O(n)$
min/max	$O(n)$	$O(n)$

PROBING DISTANCE

Given a hash value $h(x)$, linear probing generates $h(x)$, $h(x)+1$, $h(x)+2$, ...

Primary clustering – the bigger the cluster gets, the faster it grows

Quadratic probing – $h(x)$, $h(x)+1$, $h(x)+4$, $h(x)+9$, ...

Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

DOUBLE HASHING

Interval between probes is fixed but computed by a second hash function

Use a secondary hash function $d(k)$ to handle collisions by placing an item in the first available cell of the series
$i+jd(k)\%N, 0\leq j\leq N-1$

$d(k)\neq 0$

N must be prime

$d(k)=q-k\%q, q<N, q$ **is prime**

Extra info: great talk on dictionaries in Python!

https://www.youtube.com/watch?v=npw4s1QTmPg

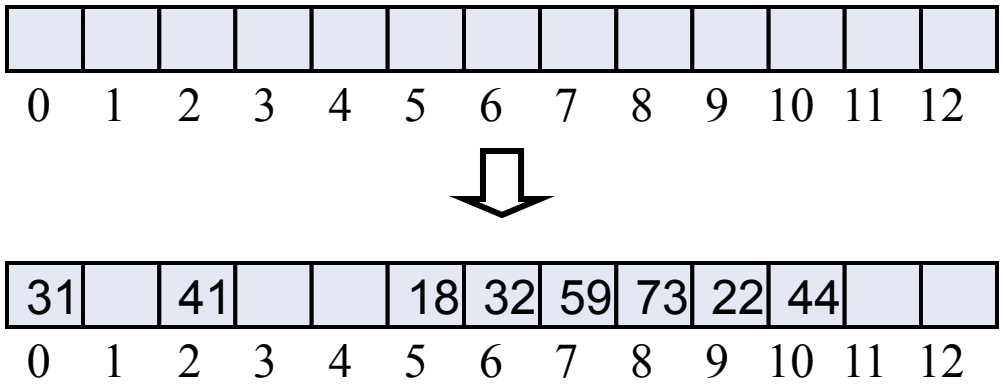
DOUBLE HASHING EXAMPLE

Double hashing:

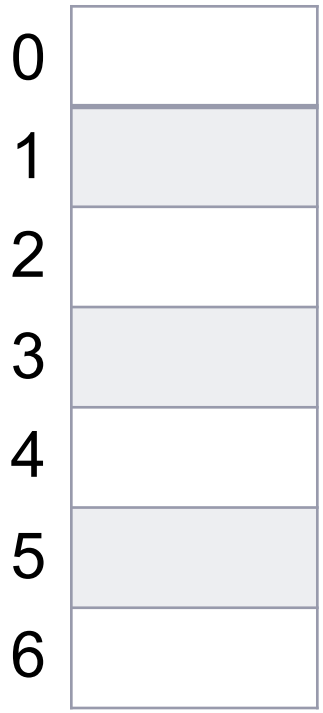
- N = 13
- h(k) = k % 13
- d(k) = 7 – k % 7

Insert: 18, 41, 22, 44, 59, 32, 31, 73

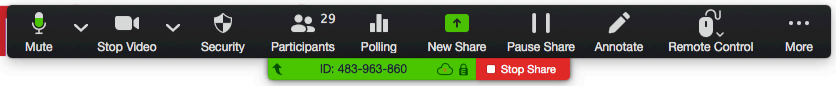
k	$h(k)$	$d(k)$	Probe Indices
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5, 10
59	7	4	7
32	6	3	6
31	5	4	5, 9, 0
73	8	4	8



PRACTICE PROBLEM EXAMPLE: LINEAR



PRACTICE: LINEAR



35, -10, 21, 7, 0, 2

		# probes	load factor
0	35	1	1/7
1	21	2	
2	7	3	
3	0	4	
4	-10	1	2/7
5	2	4	
6			

6/7

$35 \% 7 = 0$

$-10 \% 7 = 4$

$+14$

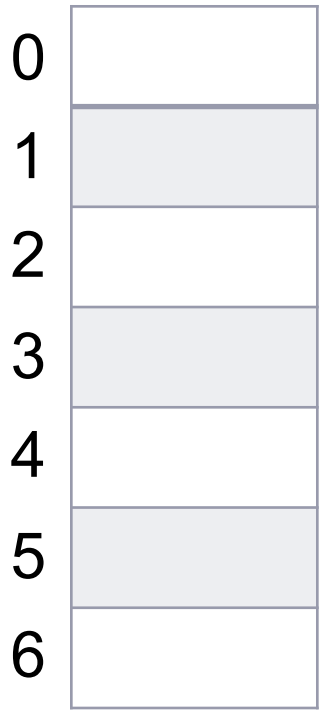
$avg = \frac{15}{6} = 2.5$

$h(k) = k \% 7$

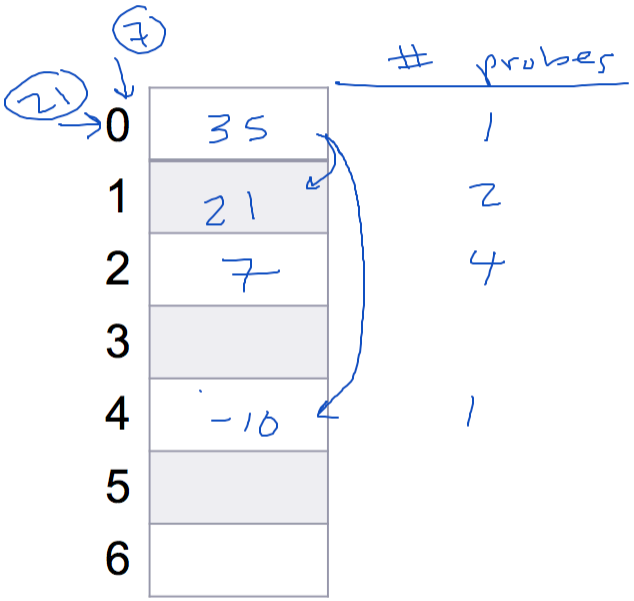
$O(1)$

$\alpha = \frac{n}{N}$

PRACTICE PROBLEM EXAMPLE: QUADRATIC



PRACT LE: QUADRATIC



35, -10, 21, 7, 0, 2

0 % 7 = 0

0 + 1 % 7 = 1

0 + 4 % 7 = 4

0 + 9 % 7 = 2

0 % 7 = 0

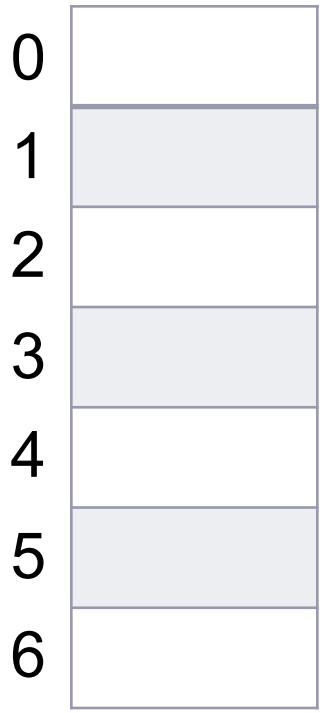
0 + 1 % 7 = 1

0 + 4 = 4

0 + 9 = 2

0 + 16 % 7 = 2

PRACTICE PROBLEM EXAMPLE: DOUBLE HASHING



PRACTICE: DOUBLE HASHING

$d(k) = p - (k \bmod p)$ 35, -10, 21, 7, 0, 2

0	35
1	
2	7
3	21
4	-10
5	2
6	0

$d(21) = 3 - (21 \bmod 3)$ (0)

$= 3$

$d(7) = 3 - 7 \bmod 3$ (1)

$= 2$

$d(0) = 3 - 0 \bmod 3$ (0)

$= 3$

avg # probes = 2.16

APR 28 OUTLINE

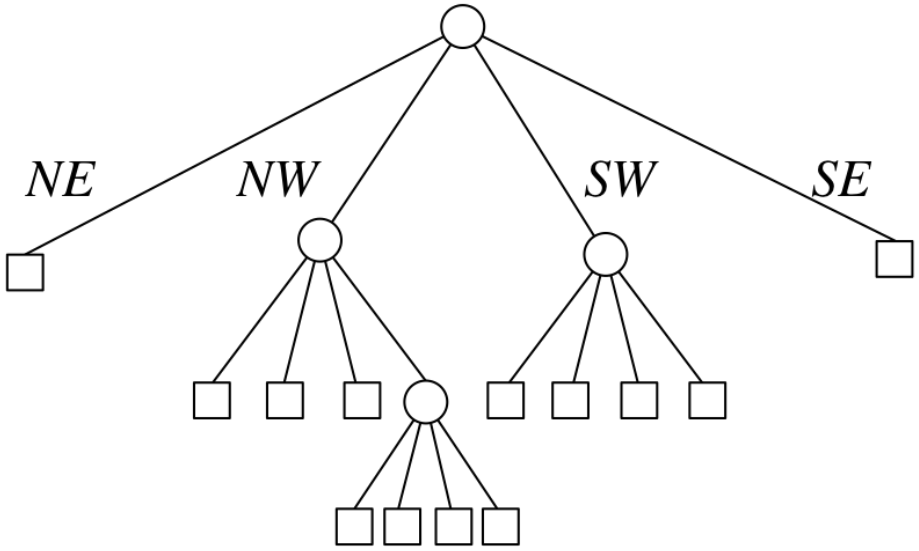
- Highlight important concepts from this course and other takeaways
- Review recap based on Google form
- **Review problems**

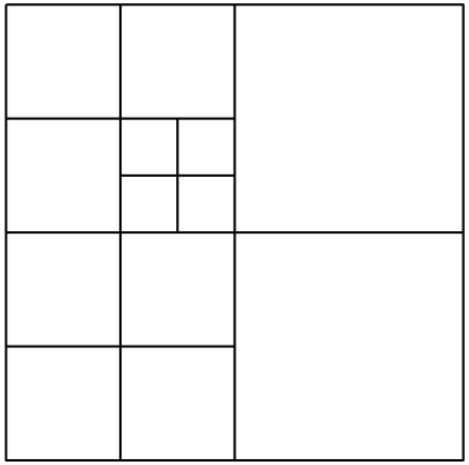
QUADTREE INTRO

QUADTREE

A tree with four children at each node

Typically used for recursive space subdivision

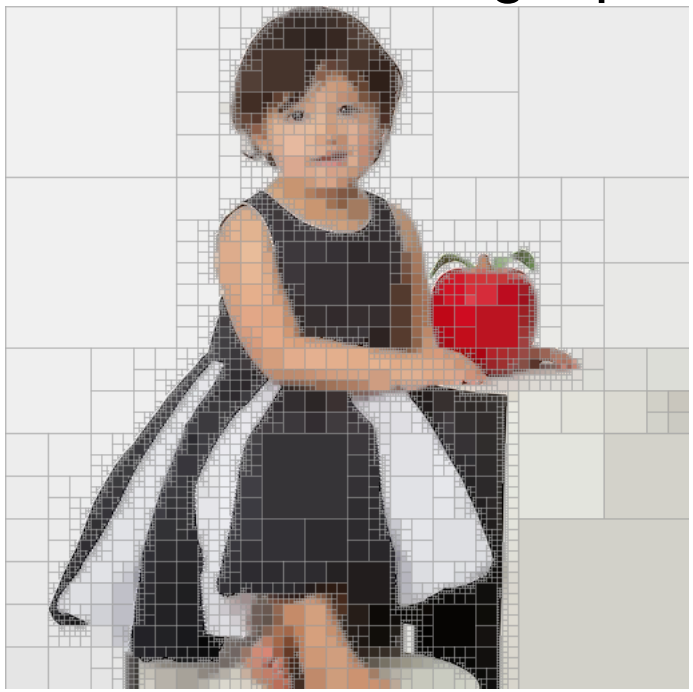




QUADTREE AND IMAGE PROCESSING

Subdivide into four sub-images

- stop when some criteria are met
- or down to a single pixel



QUADTREES