CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- Make use of chat!
    - Can ask/answer questions publicly or privately
- Exam regraded question: submit today if possible
    - Thanks to those who have submitted already
- TA hours: moving away from queue (can join zoom meeting directly)
    - May start out in the "Waiting Room"
    - Or be directed to a "Breakout Room"

Remind me to RECORD

REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-midnight (Steve)

Today/Tomorrow

Thursday 11:30-12:30pm (Lizzie)

Thursday 9-11pm (Will)

Friday 8-10pm (Gareth)

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

MAR 24 OUTLINE

- Recap Graphs
- Motivation for Trees: Binary Search
- Tree data structure and theory
- Tree traversals

MAR 24 OUTLINE

- **Recap Graphs**
- Motivation for Trees: Binary Search
- Tree data structure and theory
- Tree traversals

THE GRAPH ADT

The designation of the graph as undirected or directed happens at construction time.

```
numVertices()
vertices()
numEdges()
edges()
getEdge(u,v)
endpoints(e)
removeVertex(v)
```

```
outDegree(v)
inDegree(v)
outgoingEdges(v)
incomingEdges(v)
insertVertex(elem)
insertEdge(u,v,elem)
removeEdge(e)
```

Note: there are many ways to implement a Graph!

VERTEX CLASS (ADJACENCY LIST)

```
public class Vertex {

    private String name; // names should be unique
    private List<Vertex> edges;

    public Vertex(String initName) {
        name = initName;
        edges = new ArrayList<Vertex>();
    }

    public String getName() {
        return name;
    }

    public List<Vertex> getEdges() {
        return edges;
    }

    public void addEdge(Vertex destination) {
        edges.add(destination);
    }
```

VERTEX CLASS (ADJACENCY LIST)

```
public boolean hasEdge(Vertex destination) {
    // return true if there is an edge
    // between this Vertex and destination
    for (Vertex v : edges) { // worst-case: O(n)
        if (v.equals(destination)) {
            return true;
        }
    }
    return false;
}
```

ADJACENCY GRAPH CLASS
(ADJACENCY LIST)

```
public boolean hasEdge(Vertex u, Vertex v) {
    return u.hasEdge(v); // runtime???
}

public List<Vertex> outgoingEdges(Vertex v) {
    return v.getEdges(); // already outgoing edges
}


// O(n^2) !!!
public List<Vertex> incomingEdges(Vertex v) {

    List<Vertex> incoming = new ArrayList<Vertex>();

    // num vertices is n
    for (Vertex origin : vertices) { // O(n)
        if (hasEdge(origin, v)) {
            incoming.add(origin);
        }
    }
    return incoming;
}
```

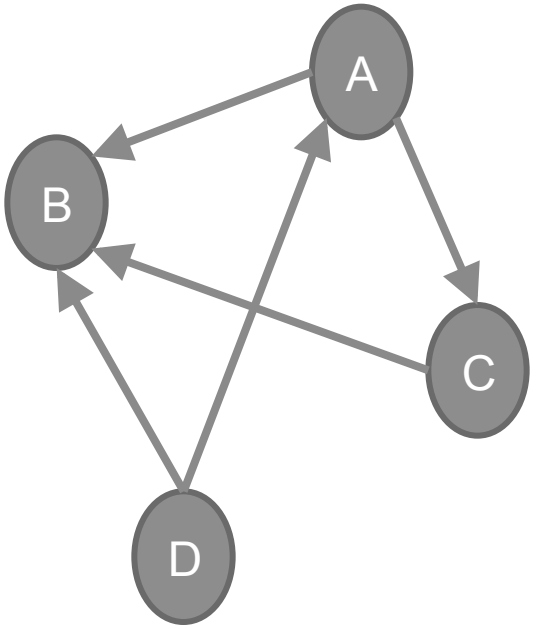
GRAPH ADJACENCY LIST RUNTIMES (LAST TIME)

Let n be the number of vertices. Fill in the runtime for each method below.

`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)`	$O(n^2)$

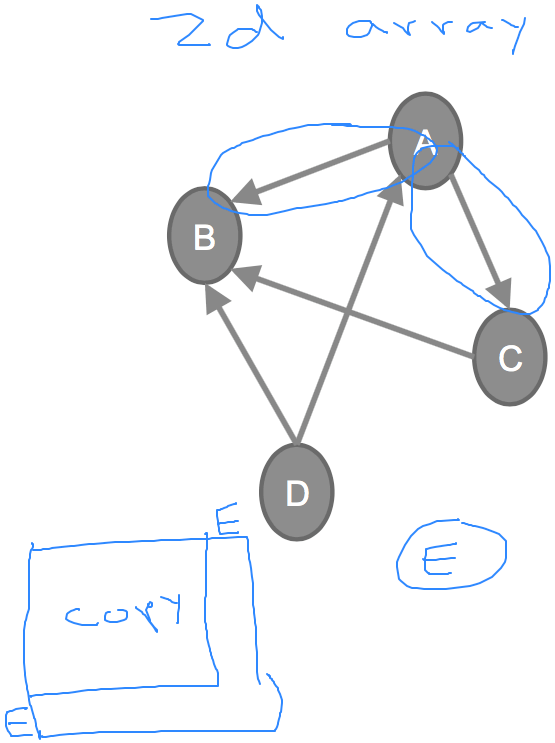
ADJACENCY MATRIX

	A (0)	B (1)	C (2)	D (3)
A (0)				
B (1)				
C (2)				
D (3)				



ADJACENCY MATRIX

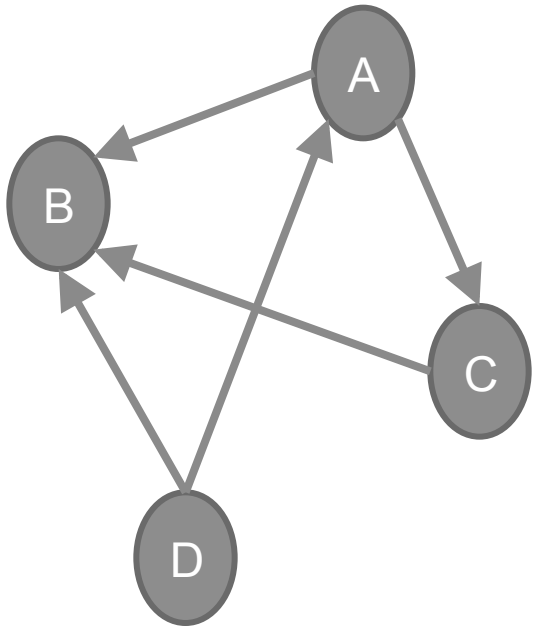
	A (0)	B (1)	C (2)	D (3)
A (0)				
B (1)				
C (2)				
D (3)				



Chat question: “What’s the time complexity of copying?”
Answer: usually the **size** of the whole data structure (so $O(n^2)$ here)

ADJACENCY MATRIX

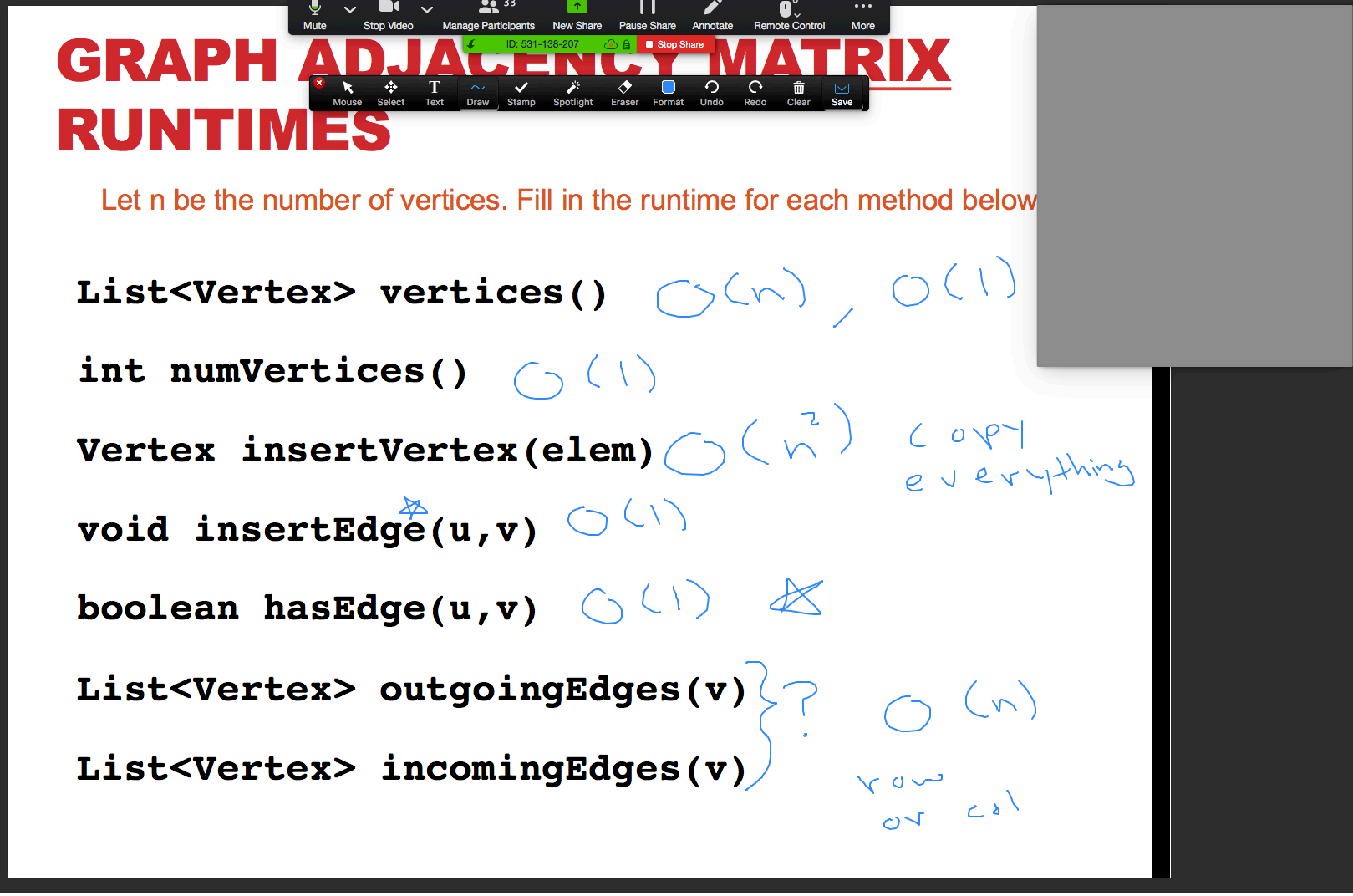
	A (0)	B (1)	C (2)	D (3)
A (0)	0	1	1	0
B (1)	0	0	0	0
C (2)	0	1	0	0
D (3)	1	1	0	0



GRAPH ADJACENCY MATRIX RUNTIMES

Let n be the number of vertices. Fill in the runtime for each method below.

```
List<Vertex> vertices()

int numVertices()

Vertex insertVertex(elem)

void insertEdge(u,v)

boolean hasEdge(u,v)

List<Vertex> outgoingEdges(v)

List<Vertex> incomingEdges(v)
```



Chat answer for **vertices** method: “Loop through the row or column of the matrix and add the vertices to a list” -> O(n)

If we maintained a list of vertices when we added them, could return this list in O(1)

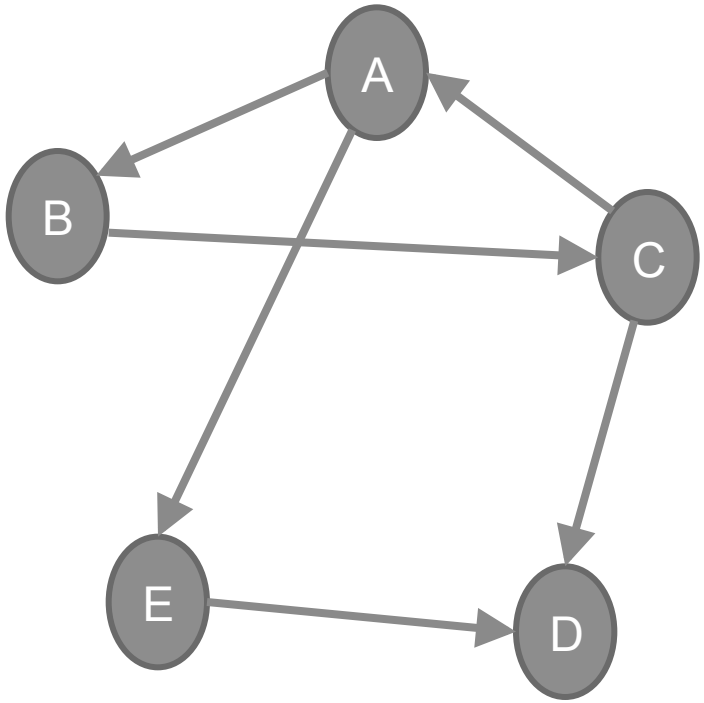
GRAPH ADJACENCY MATRIX RUNTIMES

Let n be the number of vertices. Fill in the runtime for each method below.

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

EXERCISES (AFTER CLASS)

What's the adjacency matrix for this graph?



Takeaways

Pros:

* fast to access any edge
* good for densely connected graphs where vertices are fixed

Cons:

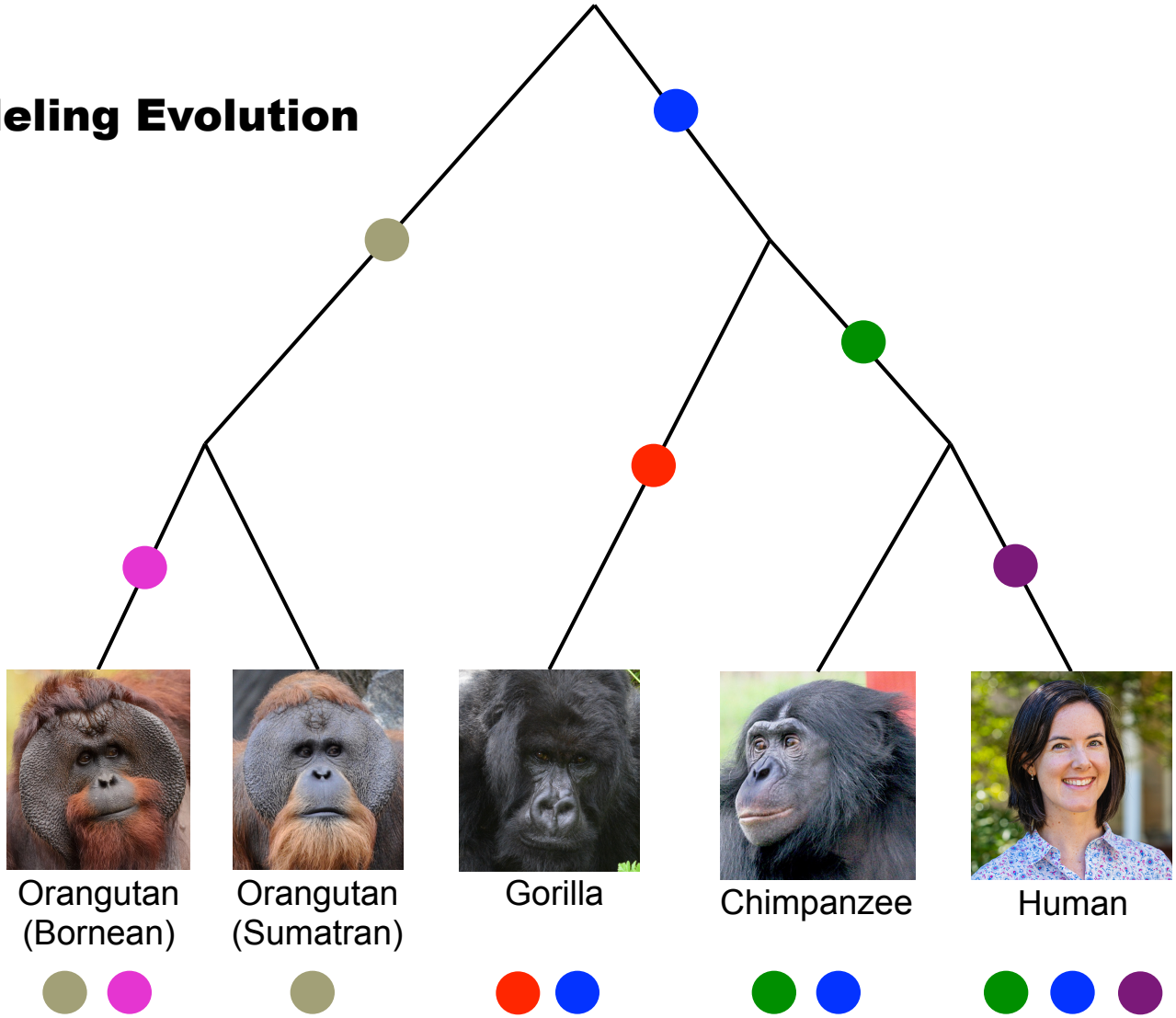
* slow to insert and delete vertices
* storage space $O(n^2)$ is large

MAR 24 OUTLINE

- Recap Graphs
- **Motivation for Trees: Binary Search**
- **Tree data structure and theory**
- Tree traversals

EXAMPLES OF TREES

1) Modeling Evolution



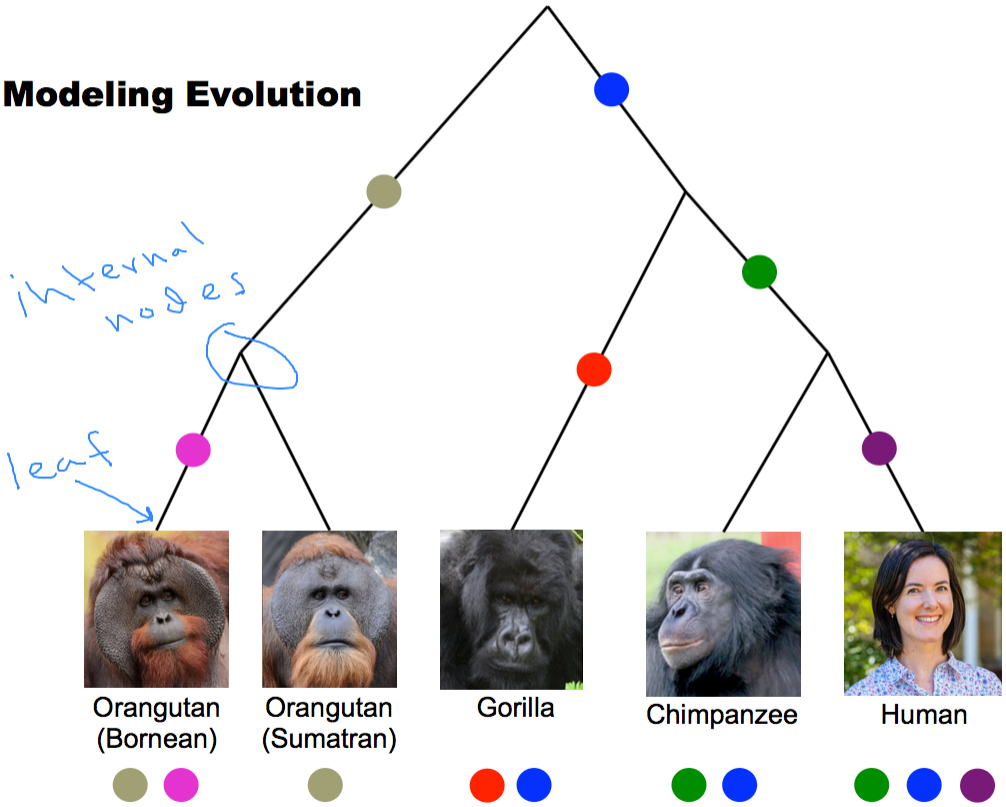
Mute | Stop Video | 33 Manage Participants | New Share | Pause Share | Annotate | Remote Control | More

ID: 531-138-207 | Stop Share

Mouse | Select | Text | Draw | Stamp | Spotlight | Eraser | Format | Undo | Redo | Clear | Save

EXAMPLES OF TREES

1) Modeling Evolution



EXAMPLES OF TREES

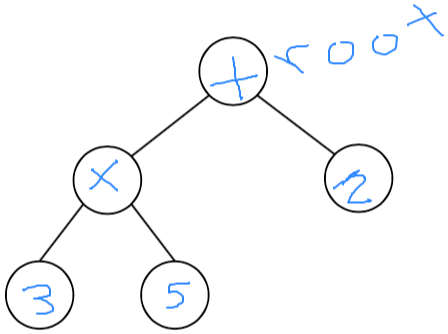
EXAMPLES OF TREES

Mute | Stop Video | Manage Participants 33 | New Share | Pause Share | Annotate | Remote Control | More

ID: 531-138-207 | Stop Share

Mouse | Select | Text | Draw | Stamp | Spotlight | Eraser | Format | Undo | Redo | Clear | Save

2) Arithmetic Expressions (binary operators)



compute(root) = compute(left) "op" compute(right)

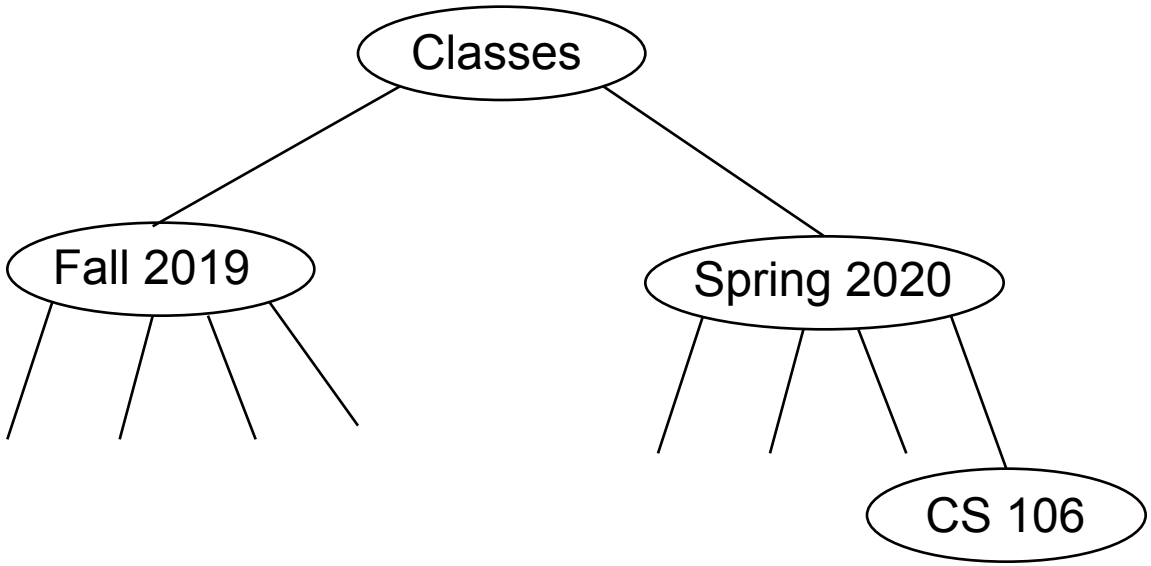
(3 x 5) + 2

15 + 2

17

EXAMPLES OF TREES

3) File systems (i.e. folders)

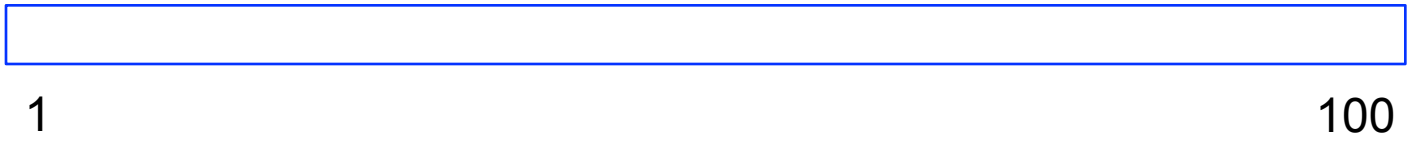


MOTIVATION: BINARY SEARCH

I'm thinking of a number between 1 and 100

You can only ask questions of the form: "is it greater than x?"

What is your first question?



MOTIVATION: BINARY SEARCH

I'm thinking of a number between 1 and 100

You can only ask questions of the form: "is it greater than x?"

What is your first question?

① ? > 50 yes

② ? > 75 no

③ ? > 62 yes

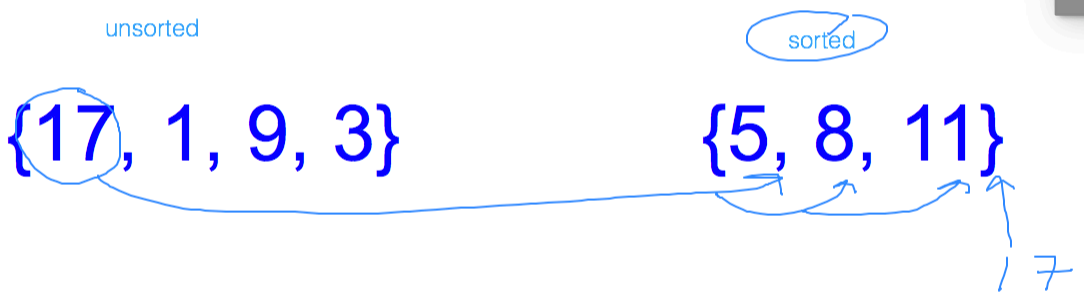
69

6 questions!

MOTIVATION: INSERTION SORT

MOTIVATION: INSERTION SORT

Back to Lab 3, when we were trying to iteratively insert names create a sorted list of names.



BINARY SEARCH PSEUDOCODE

Input: query (q) and sorted array (arr)

Output: index of q in arr (-1 if not in arr)

```
def bsearch(q, arr, l, r):
```

Input: q = 11, arr = {5, 8, 11}
Output: 2 (index)

BINARY SEARCH PSEUDOCODE

Input: query (q) and sorted array (arr)

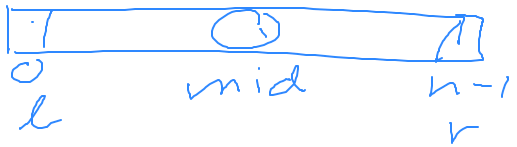
Output: index of q in arr (-1 if not in arr)

Input: q = 11, arr = {5,

Output: 2 (index)

```
def bsearch(q, arr, l, r):
    mid = (l+r)/2
    base case
    if arr[mid] == q:
        return mid
    *not found case
    recursive case
    if arr[mid] > q:
        return bsearch(q, arr, l, mid-1)
    if arr[mid] < q:
        return bsearch(q, arr, mid+1, r)
```

// l = 0
r = n-1

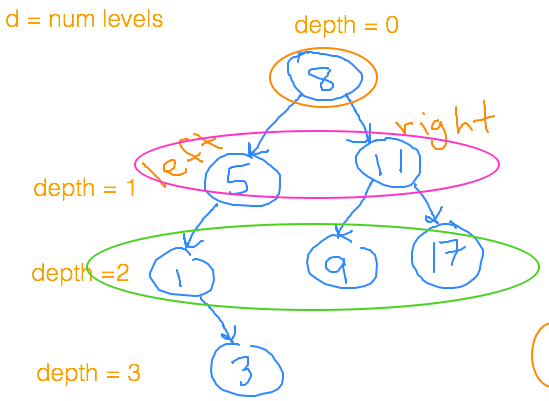


INSERTION SORT WITH TREES

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

len = n

n=7



runtime

insertions = n

each insertion:

$1 + 2 + 2^2 + 2^3 + \cdots 2^{d-1} = n$

$2 + 2^2 + \cdots 2^{d-1} + 2^d = 2n$

$2^d - 1 = n$

$d = \log(n)$

$1 + 2 = 3$

$2^0 + 2^1 = 3$

right

left

TOTAL = O(n * log(n))

worst case!

- No duplicates (for now)
- Everything to the right of each root is less than root data
- Everything to the left of each root is greater than root data

INSERTION SORT WITH TREES

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

Chat Q&A:

Q: Why does the first row end at 2^(d-1) instead of 2^d?
A: Let d be the number of levels (number of comparisons), so depth = d-1

Q: Could you explain the expression (1+2+…) again using the tree example?
A: See the color coding on the previous slide (1 for the root, 2 for its children, etc)

Q: Why is the runtime O(nlog(n)) instead of O(log(n))?
A: We have n nodes, and each of them need to be inserted up to the depth

Q: Shouldn't n be 7 for our example instead of 1+2+4+8 which is what we'd get using the expression?
A: Right – think about this runtime analysis as the worst-case when the tree is "full"

- No duplicates (for now)
- Everything to the left of each root is less than root data
- Everything to the right of each root is greater than root data

MAR 24 OUTLINE

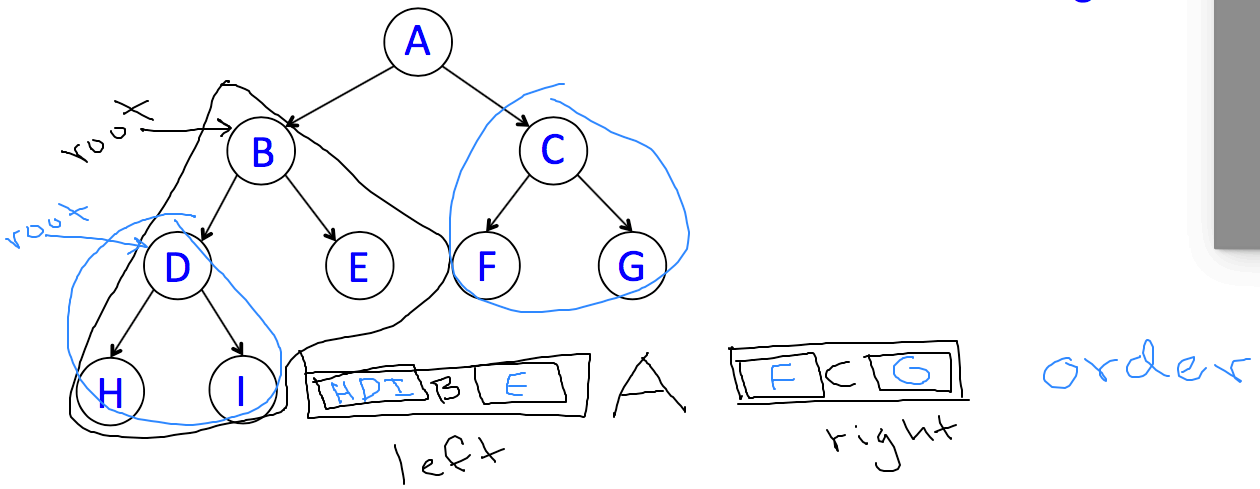
- Recap Graphs
- Motivation for Trees: Binary Search
- Tree data structure and theory
- **Tree traversals**

TREE TRAVERSALS: IN-ORDER

left – root – right

TREE TRAVERSALS: IN-ORDER

left – root – right

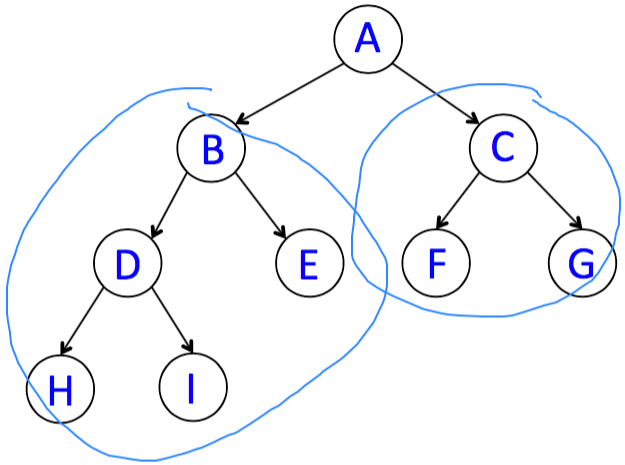


TREE TRAVERSALS: PRE-ORDER

root − left − right

TREE TRAVERSALS: PRE-ORDER

root − left − right

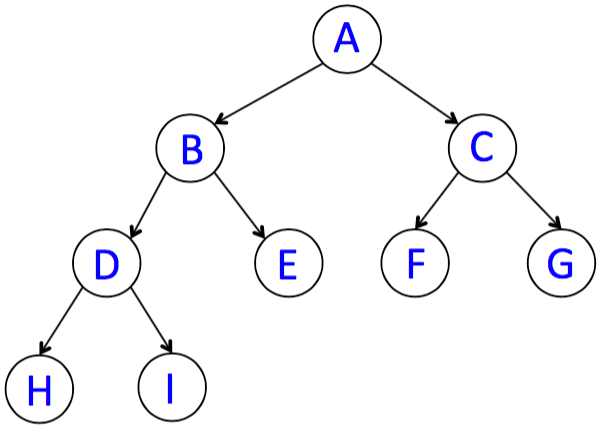


TREE TRAVERSALS: POST-ORDER

left – right – root

TREE TRAVERSALS: POST-ORDER

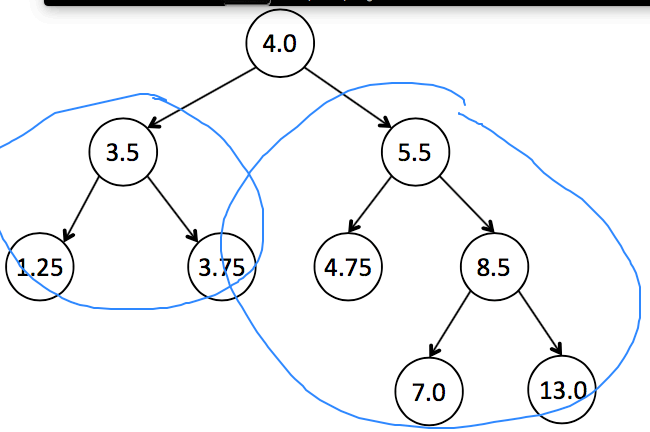
left – right – root



HANDOUT (PAIR EXERCISE)

Next time!

HANDOUT (PAIR EXERCISE)



3.5 4.0 5.5

sorted!

inorder

TODAY!

11:30 - 12:30

4:30 - 6