CSE 373: Data Structures and Algorithms

Final Review

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Today

Practice exam posted

- Use it as reference for exam question formats, not as a study material.

TA-led review session

- Sunday 2-5pm in **Smith Hall (SMI) 120**

Office hours on Monday

- Check the course calendar tomorrow

HW5 Part 2

Exam format: Type of questions

Type of questions

1. Multiple choice questions (MCQ)
   - Select one correct choice
   - Select multiple correct choices
2. Short-answer question (SAQ)
   - Answer in one word or one sentence
3. Medium-answer question (MAQ)
   - Expected in 4-5 sentences
   - E.g., explain how you would solve so and so problem

Exam format

- Somewhat different than previous term exams that you may have seen.

- Ordered roughly in the order of difficultly

Three (or maybe four) sections

1. Warmup – mostly MCQs and SAQs

2. Basic – mostly MCQs and SAQs

3. Applied – mostly MAQs

4. … ?

Exam Tips

1. Manage your time well
2. Don’t dwell on MCQs for too long
3. (Rough guideline) 1 point worth MCQ means you should not spend more than 1 minute on it.
4. What type of questions you won’t get on the exam:
    1. Execute Primm’s algorithm on this graph and fill the Primm’s algorithm table
    2. Insert given list of values in a hash table that uses quadratic probing
    3. Insert elements in heap or AVL tree
    4. Questions asking you to write java code
    5. Find c and n0, or solve summations

Exam Tips: Medium-answer questions

1. No need to explain how the algorithms we covered in the class. You can use them as black box tools in your solution, unless you are modifying the algorithm, in which case you need to describe your modification.

2. Don't worry about constant factors unless the question explicitly asks otherwise.

3. Expectation: Answer in 4-5 sentences.

(For partial credit)

4. A brute force solution that works will get more partial credit than an efficient but incomplete answer. So, if you are stuck at finding and efficient, but know a brute force solution give that first.

5. Solving a problem may involve multiple steps. If you know how to solve one step, but not the other, write down solution to step 1 for partial credit.

Review

ADTs vs Data Structures

Data Structure

- *A way of organizing and storing related data points*
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

Algorithm

- A series of precise instructions used to perform a task
- Examples from CSE 14X: binary search, merge sort, recursive backtracking

Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

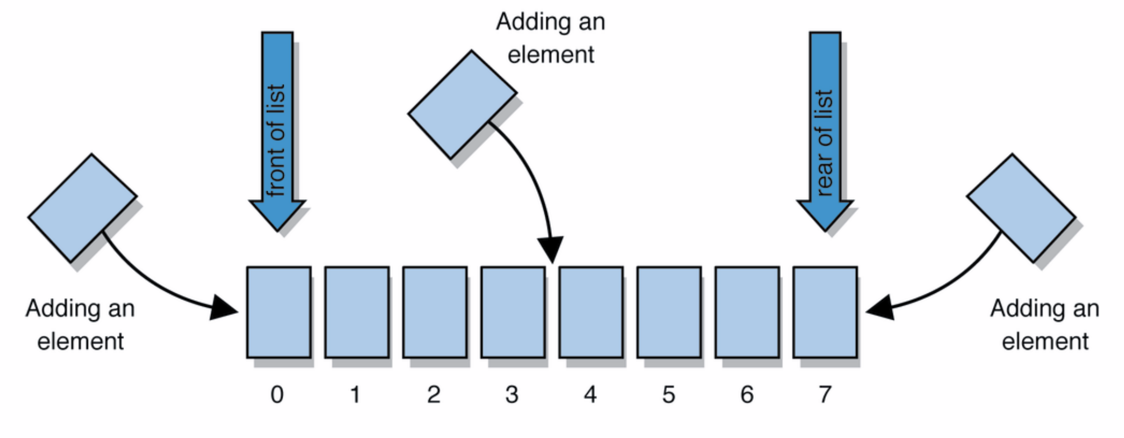
List ADT

list: stores an ordered sequence of information.

- Each item is accessible by an index.
- Lists have a variable size as items can be added and removed

Supported Operations:

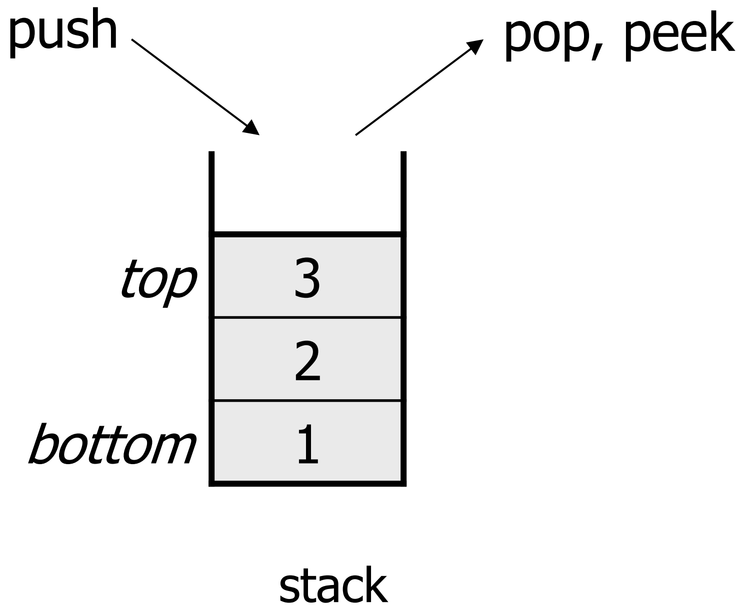
- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list



Stack ADT

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").



basic stack operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size():** how many items are in the stack?
- **isEmpty():** true if there are 1 or more items in stack, false otherwise

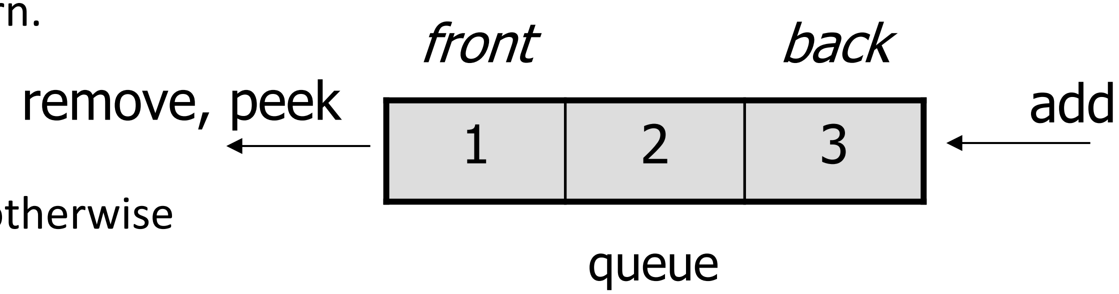
Queue ADT

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

basic queue operations:

- **add(item):** aka “enqueue” add an element to the back.
- **remove():** aka “dequeue” Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise



Map ADT (Dictionary)

map: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.

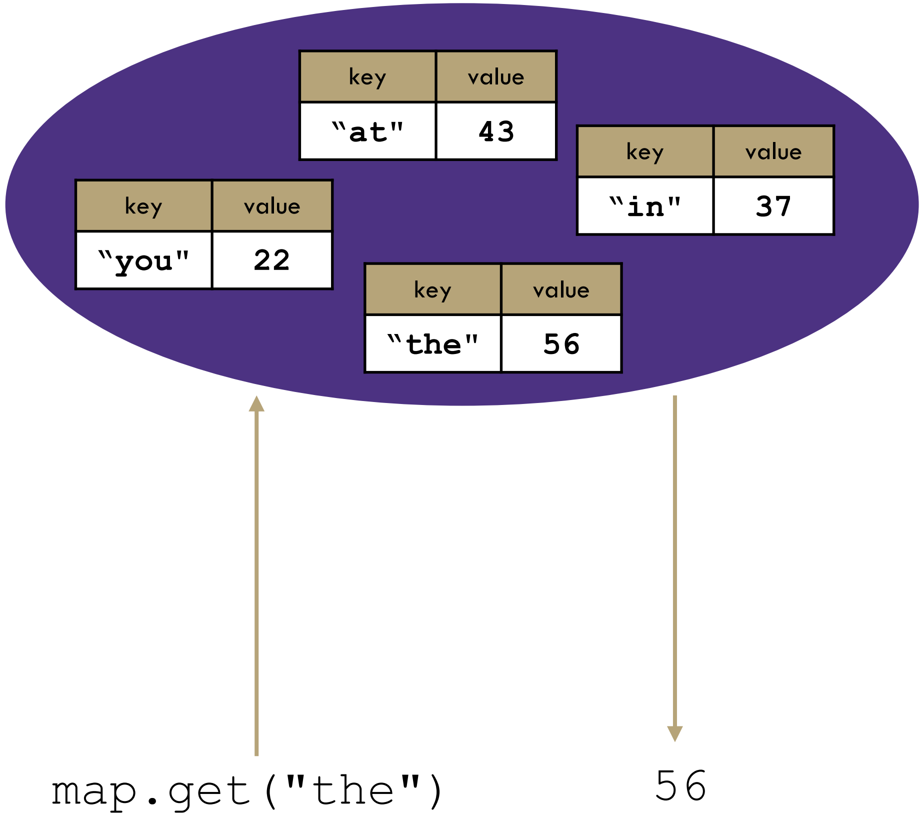
- a.k.a. "dictionary", "associative array", "hash"

operations:

- **put**(*key*, *value*): Adds a mapping from a key to a value.
- **get**(*key*): Retrieves the value mapped to the key.
- **remove**(*key*): Removes the given key and its mapped value.

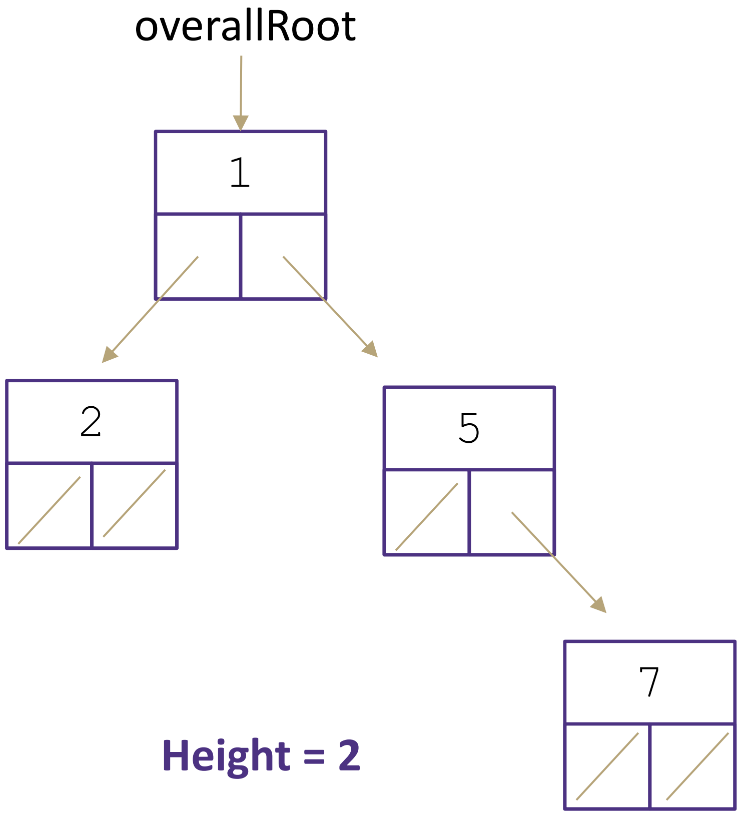
KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3

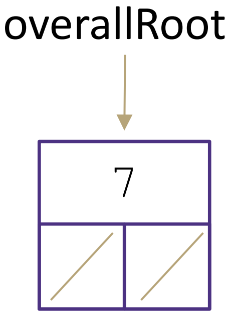


Tree Height

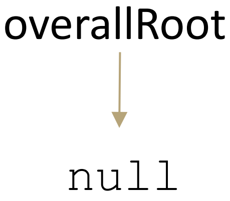
What is the height of the following trees?



Height = 2



Height = 0



Height = -1 or NA

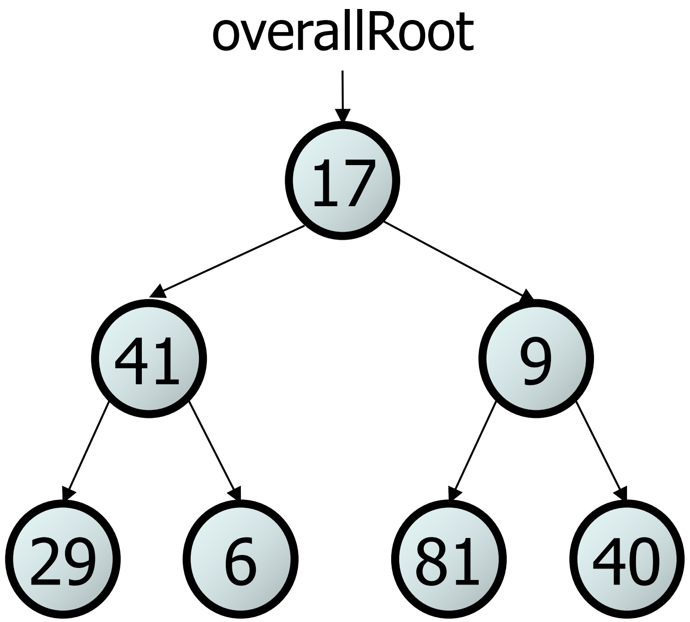
Traversals

traversal: An examination of the elements of a tree.

- A pattern used in many tree algorithms and methods

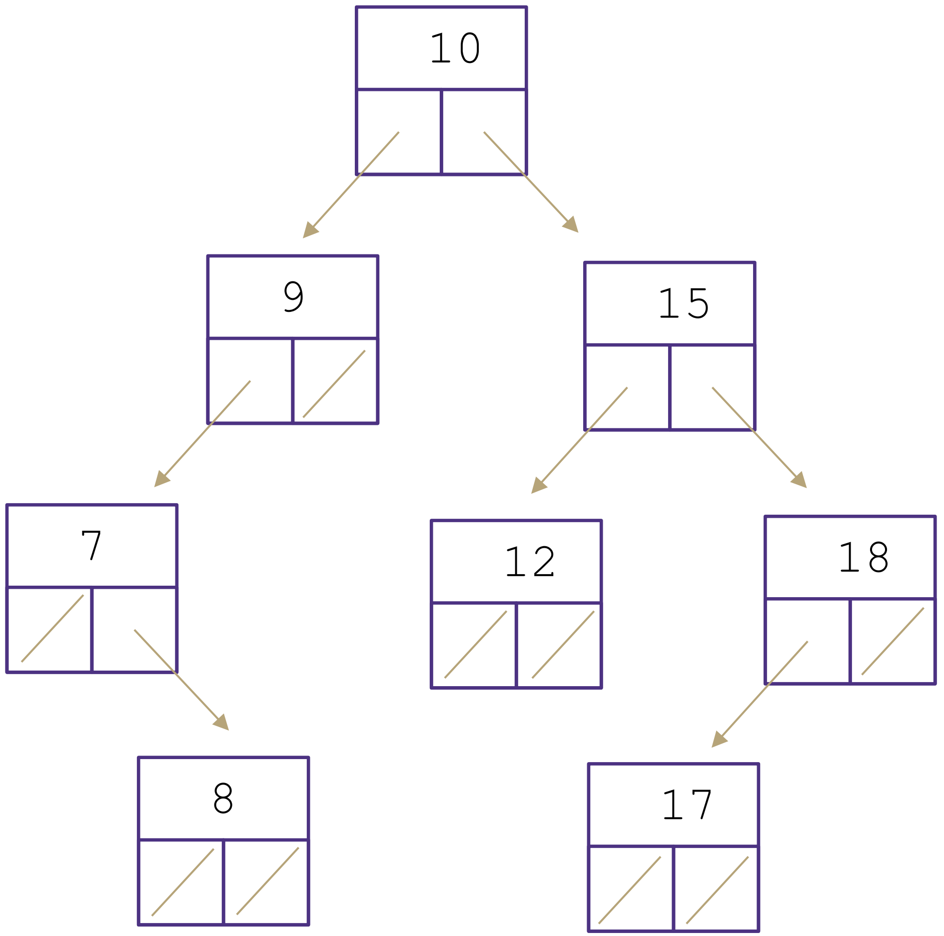
Common orderings for traversals:

- **pre-order**: process root node, then its left/right subtrees
  - `17 41 29 6 9 81 40`
- **in-order**: process left subtree, then root node, then right
  - `29 41 6 17 81 9 40`
- **post-order**: process left/right subtrees, then root node
  - `29 6 41 81 40 9 17`



Binary Search Trees

A **binary search tree** is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.



AVL trees: Balanced BSTs

AVL Trees must satisfy the following properties:

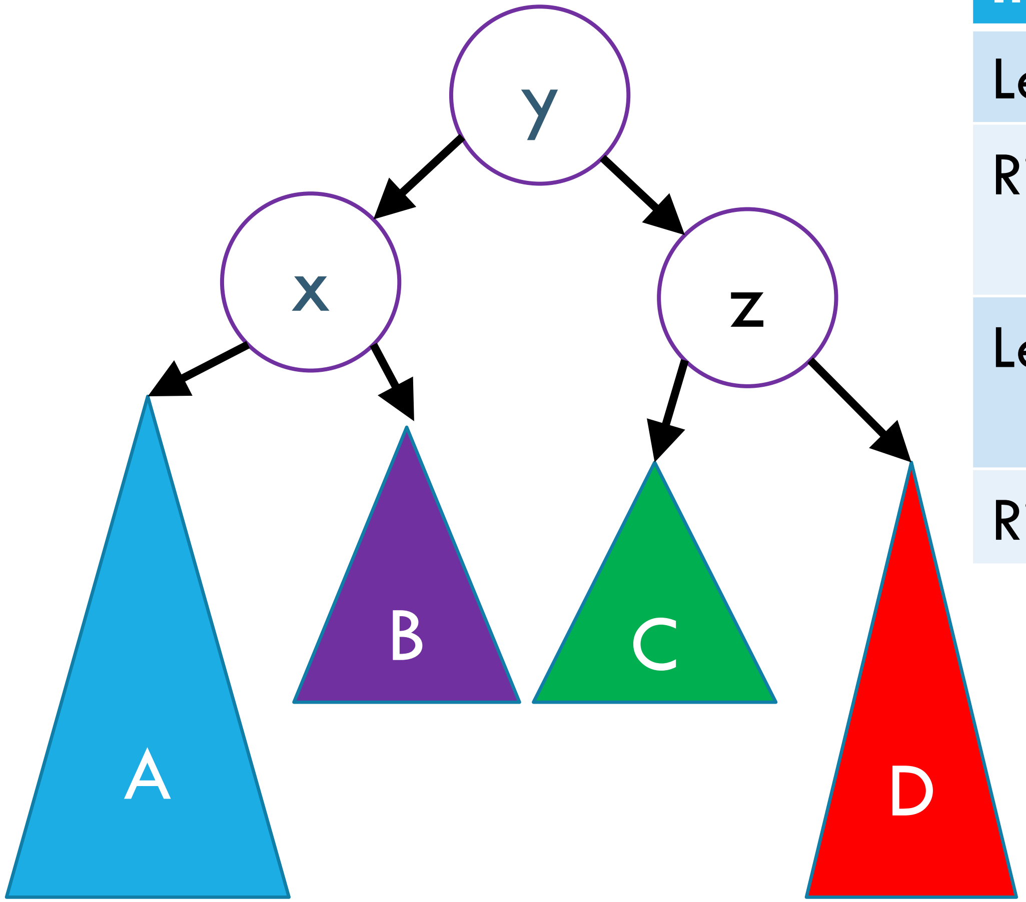
- binary trees: every node must have between 0 and 2 children
- binary search tree (BST property): for every node, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- Balanced (AVL property): for every node, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

The AVL property:

1. ensures depth is always $O(\log n)$ – Yes!
2. is easy to maintain – Yes! (using single and double rotations)

Four cases to consider



Insert location	Solution
Left subtree of left child of y	Single right rotation
Right subtree of left child of y	Double (left-right) rotation
Left subtree of right child of y	Double (right-left) rotation
Right subtree of right child of y	Single left rotation

How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

- Just go back up the tree from where we inserted.

How many rotations might we have to do?

- Just a single or double rotation on the lowest unbalanced node.
- A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion.

Hash tables: Motivation

- data = (key, value)
- operations: put(key, value);   get(key);   remove(key)

- O(n) with Arrays and Linked List
- O(log n) with BST and AVL trees.

- Can we do better? Can we do this in O(1) ?

Strategies to handle hash collision

There are multiple strategies. In this class, we'll cover the following three:

1. Separate chaining
2. Open addressing
   - Linear probing
   - Quadratic probing
3. Double hashing

Hash tables review

Hash Tables:

- Efficient find, insert, delete **on average, under some assumptions**
- Items not in sorted order

Resizing:

- Always make the table size a prime number.
- λ determines when to resize, but depends on collision resolution strategy.

Things to know:

- How different collision strategies work
- Advantages and disadvantages of different strategies
- How insert, find, delete works (or should not be implemented)

Heap review

Heap is a tree-based data structure that satisfies

- (a) structure property: it's a complete tree

- (b) heap property, which states:
  - for min-heap: $parent \leq children$
  - for max-heap: $parent \geq children$

- Operations of interest:
  - removeMin()
  - peekMin()
  - insert()

- Applications: priority queue, sorting, ..

Desired properties in a sorting algorithm

Review

Stable
- In the output, equal elements (i.e., elements with equal keys) appear in their original order

In-place
- Algorithm uses a constant additional space, $O(1)$ extra space

Adaptive
- Performs better when input is almost sorted or nearly sorted
- (Likely different big-O for best-case and worst-case)

Fast. $O\ (n \log n)$

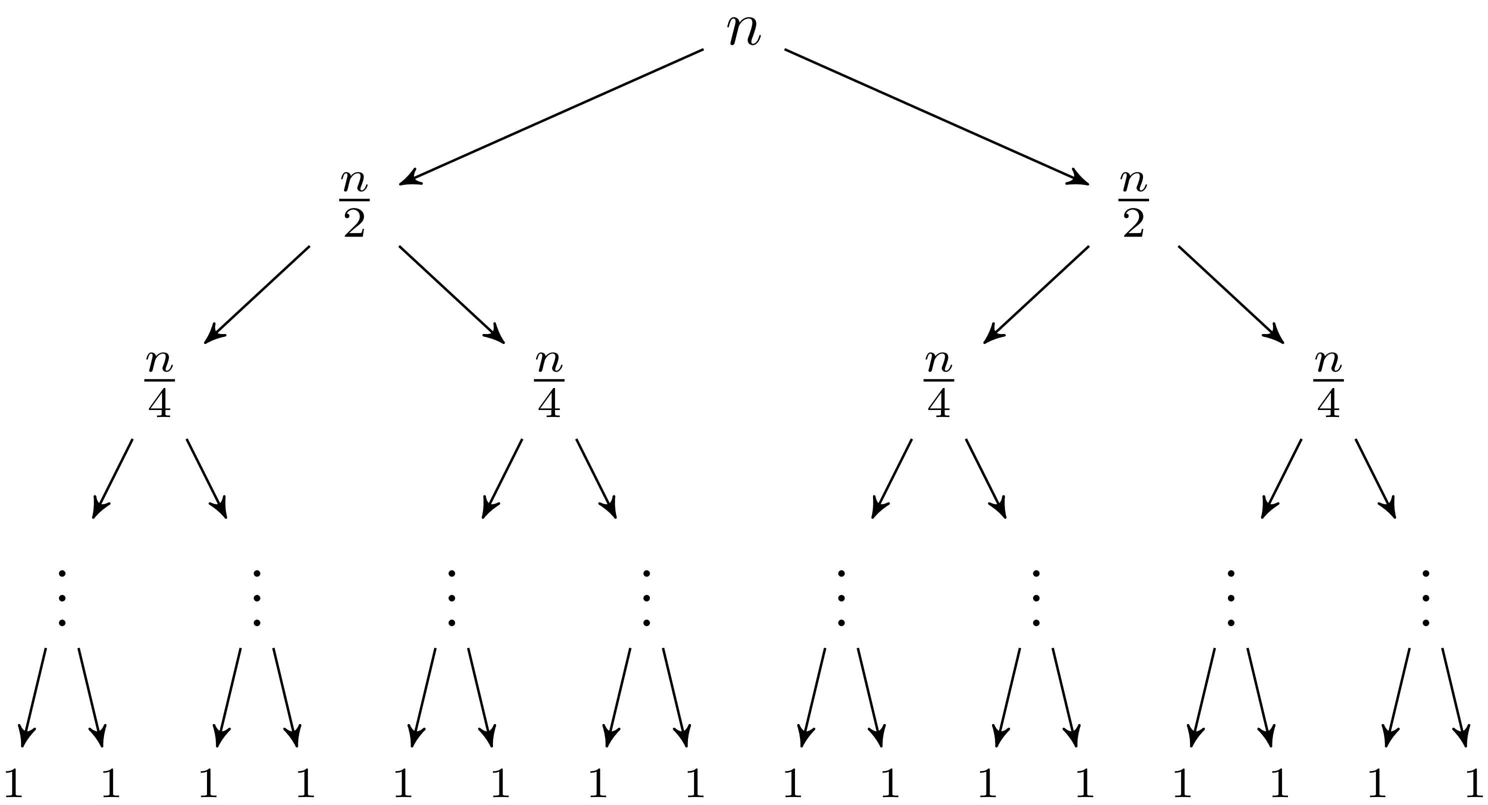
No algorithm has all of these properties. So choice of algorithm depends on the situation.

Sorting algorithms – High-level view

- $O(n^2)$
  - Insertion sort
  - Selection sort
  - Quick sort (worst)
- $O(n \log n)$
  - Merge sort
  - Heap sort
  - Quick sort (avg)
- $\Omega(n \log n)$ -- lower bound on comparison sorts
- $O(n)$ – non-comparison sorts
  - Bucket sort (avg)

Tree method

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$



Level	Number of Nodes at level	Work per Node	Work per Level
0			
1			
2			
i			
base			

Last recursive level:

Design technique: Divide-and-conquer

Very important technique in algorithm to attack problems

Three steps:

1. Divide: Split the original problem into smaller parts
2. Conquer: Solve individual parts independently (think recursion)
3. Combine: Put together individual solved parts to produce an overall solution

Merge sort and Quick sort are classic examples of sorting algorithms that use this technique

Graph Review

Graph Definitions/Vocabulary
- Vertices, Edges
- Directed/undirected
- Weighted
- Etc…

Graph Traversals
- Breadth First Search
- Depth First Search

Finding Shortest Path
- Dijkstra's

Topological Sort, Strongly connected components

Minimum Spanning Trees
- Primm's
- Kruskal's

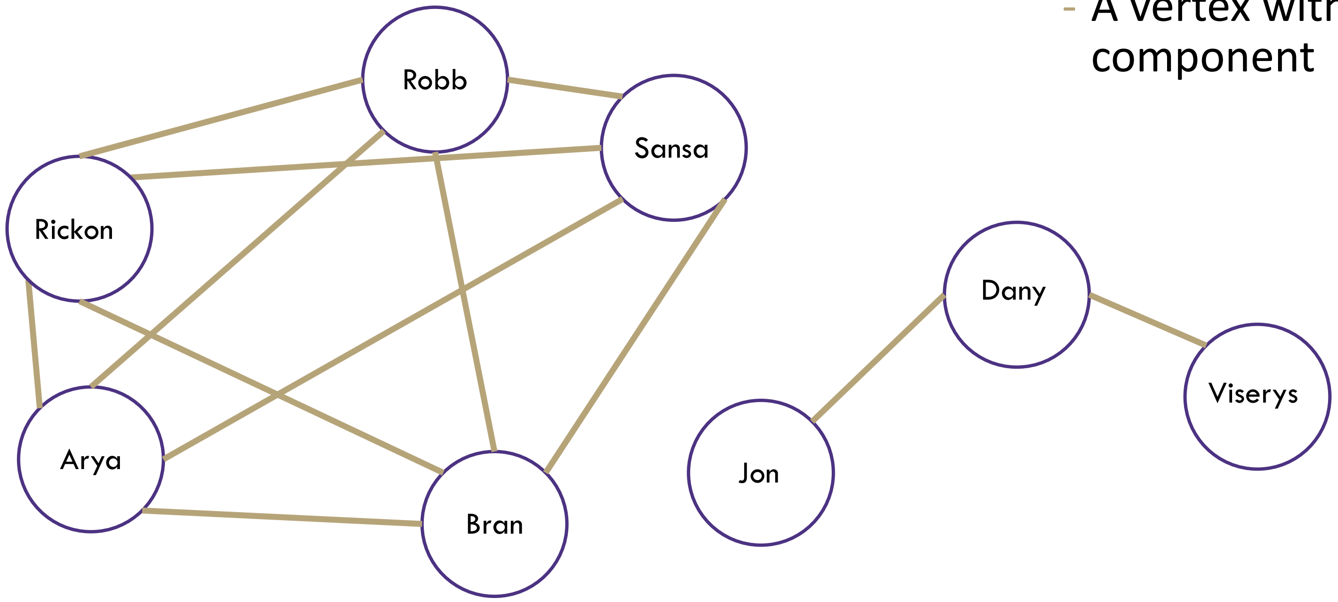
Disjoint Sets
- Implementing Kruskal's

Strongly Connected Components

Connected [Undirected] Graphs

Connected graph – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex



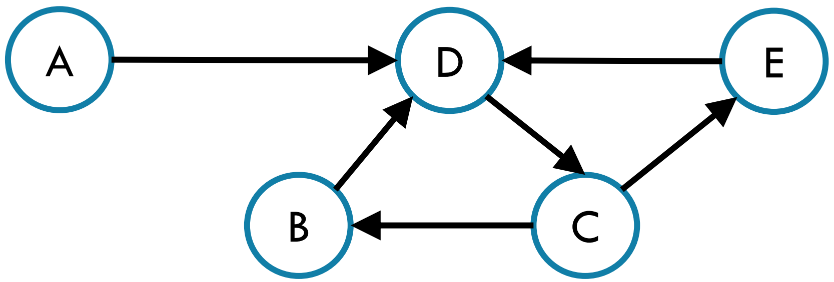
Connected Component – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component

Strongly Connected Components

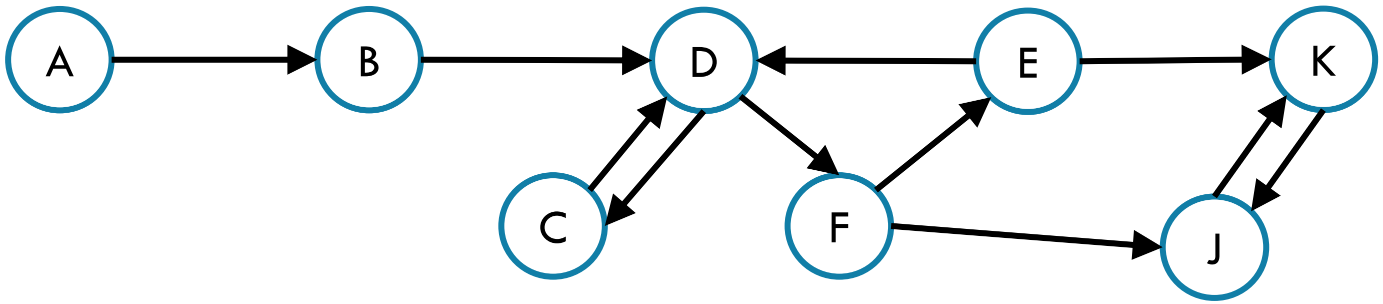
Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.



Note: the direction of the edges matters!

Strongly Connected Components Problem



{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Components Problem

Given: A directed graph G

Find: The strongly connected components of G

SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a [B/D]FS from every vertex,
- For each vertex record what other vertices it can get to
- and figure it out from there.

But you can do better. There's actually an O(|V|+|E|) algorithm!

I only want you to remember two things about the algorithm:

- It is an application of depth first search.
- It runs in linear time

The problem with running a [B/D]FS from every vertex is you recompute a lot of information.

The time you are popped off the stack in DFS contains a "smart" ordering to do a second DFS where you don't need to recompute that information.

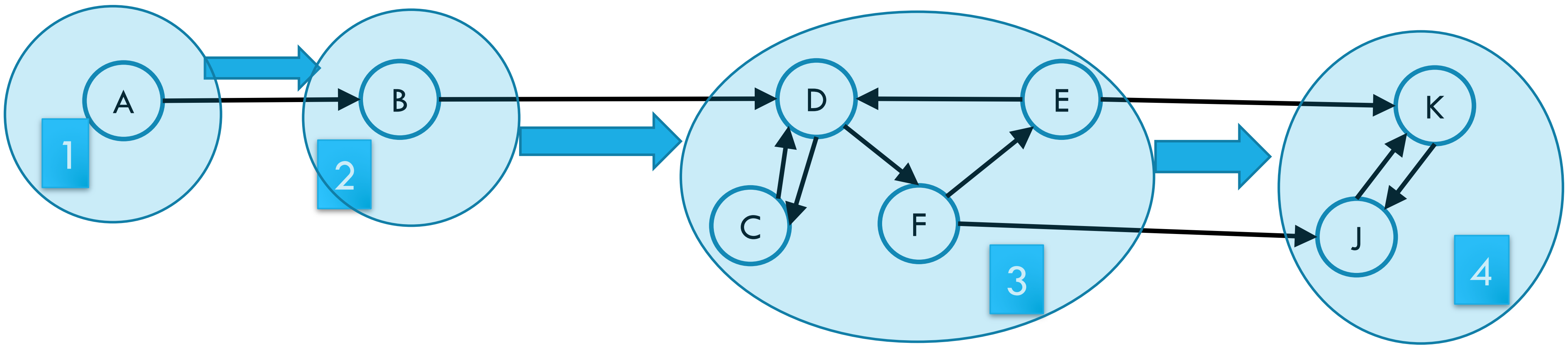
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

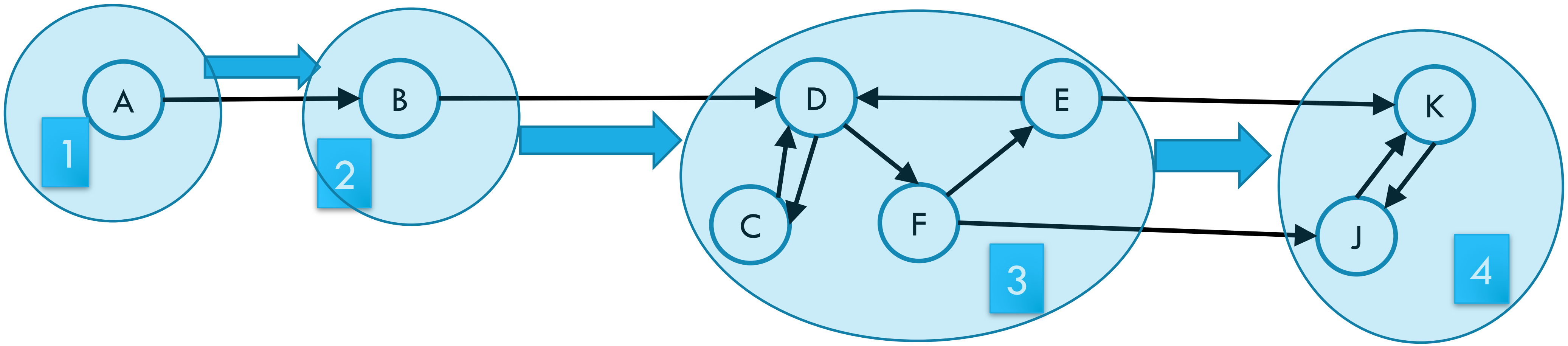
We've found the strongly connected components of G.

Let's build a new graph out of them! Call it H

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.



Why Find SCCs?



That's awful meta. Why?

This new graph summarizes reachability information of the original graph.

- I can get from A (of G) in 1 to F (of G) in 3 if and only if I can get from 1 to 3 in H.

Why Must H Be a DAG?

H is always a DAG (do you see why?).

Takeaways

Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in linear time.

Just about everything you could want to do with your graph will take at least as long.

You should think of these as **“almost free” preprocessing** of your graph.

- Your other graph algorithms only need to work on
  - topologically sorted graphs and
  - strongly connected graphs.

Practice exam Q5

Recall that in insertion sort, the algorithm does a linear search to find the position in the sorted subarray where the current element should be inserted. Suppose you use a binary search instead of the linear search to find that position, what would be the worst-case tight big-O time complexity of the sort? Briefly justify your answer.

Practice exam Q5

Recall that in insertion sort, the algorithm does a linear search to find the position in the sorted subarray where the current element should be inserted. Suppose you use a binary search instead of the linear search to find that position, what would be the worst-case tight big-O time complexity of the sort? Briefly justify your answer.

Solution: O(n^2)Because even if we find the insertion index quickly (in O (log n) time), we still need to shift all the elements, which takes O (n) time, so the time complexity still remains as O(n^2).

Practice exam Q6

Suppose you are given a source code and you need to figure out the order in which to compile the files. Explain how you would solve this problem? State the runtime of your solution.

Practice exam Q6

Suppose you are given a source code and you need to figure out the order in which to compile the files. Explain how you would solve this problem? State the runtime of your solution.

Solution: Represent the source code files as a directed unweighted graph, where each vertex is a file and an edge represents dependency, i.e., if file A imports file B, there is an edge from B to A.

Run topological sort, and use one of the topological orderings to compile files.

Practice exam Q7

Frodo and Sam are on their way to Mordor to destroy the ring, and along their path they have to pass through several human villages on their way, some of which are now deserted and likely Orc territory. They learn that there are some paths that are being heavily guarded by Orcs, and they want to avoid those paths at all costs. Friendly spies tell Frodo and Sam that they should avoid the road between two deserted villages, as it is more likely be monitored by Orcs.

Practice exam Q7

Frodo and Sam are on their way to Mordor to destroy the ring, and along their path they have to pass through several human villages on their way, some of which are now deserted and likely Orc territory. They learn that there are some paths that are being heavily guarded by Orcs, and they want to avoid those paths at all costs. Friendly spies tell Frodo and Sam that they should avoid the road between two deserted villages, as it is more likely be monitored by Orcs.

Solution:

Run BFS, identify all unsafe edges, and then remove them.

Run Dijkstra’s to find the shortest path.

P vs. NP review slides

Decision Problems

Let’s go back to dividing problems into solvable/not solvable.
For today, we’re going to talk about **decision problems**.

Problems that have a “yes” or “no” answer.

Why?

Theory reasons (ask me later).

But it’s not too bad

- most problems can be rephrased as very similar decision problems.

E.g. instead of “find the shortest path from s to t” ask
Is there a path from s to t of length at most k?

P

P (stands for "Polynomial")

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k.

The decision version of all problems we've solved in this class are in P.

P is an example of a "complexity class"

A set of problems that can be solved under some limitations (e.g. with some amount of memory or in some amount of time).

P vs. NP

P (stands for “Polynomial”)

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k.

NP (stands for “nondeterministic polynomial”)

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

Claim: P is a subset of NP, i.e. every problem in P is also in NP

(do you see why?)

NP-Complete

Let’s say we want to prove that some problem in NP needs exponential time (i.e. that P is not equal to NP).

Ideally we’d start with a really hard problem in NP.

What is the hardest problem in NP?

What does it mean to be a hard problem?

NP-complete

We say that a problem B is “NP-complete” if B is in NP and for all problems A in NP, A reduces to B.