COMP_SCI 214: Data Structures and Algorithms

Data Design

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 4 due tonight
- Homework 3 self-eval due tonight
- Homework 5 to be released tonight

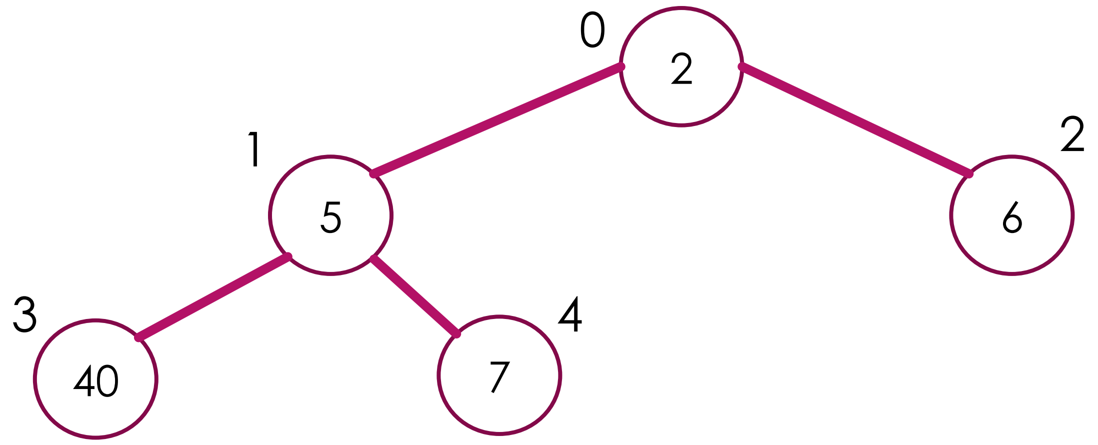
Announcements

- Project to be released next Tuesday
  - Much larger than a homework (and hence much more time)
  - More open-ended for you to make DS and algorithm decisions
- Today and Thursday’s design lectures are meant prepare you to work on your project

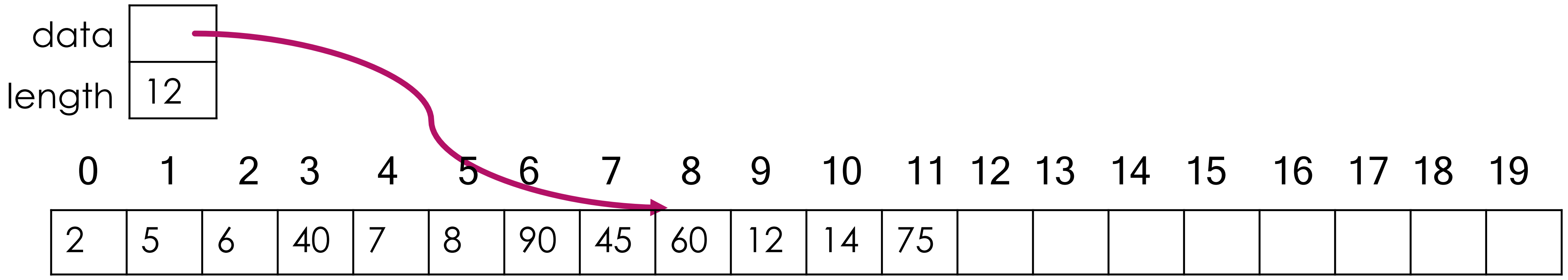
Wrap-up: Findings parents and children

- Structure is *implicit*, so can't just follow arrows as there are no real arrows
  - Structure is well-defined though, so **array indices follow a pattern**

- Given a node at index i
  - Parent index = ⌊ (i-1)/2 ⌋
    - "floor" of quotient
  - Left child index = 2*i + 1
  - Right child index = 2*i + 2



Wrap-up: Mapping operations to the array



- Where would you insert?
  - At index `length`
- Where would you remove?
  - Remove what's at index `length-1` and put it at index `0`
- Update `length` in both cases

Data design

Today and Tuesday will be more active lectures

- Throughout the class, we’ll do more neighbor discussions and sharing
- You’ll learn a lot coming up with your own solutions!

Real programming

- We’ve learned about a bunch of ADTs and data structures till now
- But in the real world, no one will ever ask you to “write a hash table on its own” since libraries exist
  - It was necessary and important for you to reason about and work with ADTs, data structures, and invariants though
- You’ll need to reduce or map the real-world problem to DSes and algorithm problems and figure out when to use what
  - These problems will not be as clear-cut a what we ask you to do in HWs

Data design

- Before solving a problem, map out all the kinds of data you'll need
  - Build a model
  - Provides an outline for your program
  - Implementation is just filling in that outline
- Open-ended process
  - Many possible right answers
  - Also iterative as we may need to go back and change things
- **Today:** we walk through the data design process on an example problem
- **Final project:** you apply this process to a *much larger* problem

The problem: Metropolitan areas

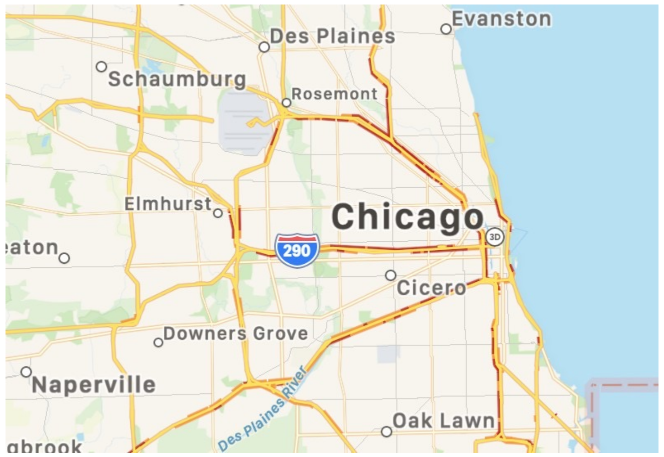
Problem of the day

- We want to group cities into metropolitan areas, then find the n largest metropolitan areas in the US by population

- From Wikipedia: A **metropolitan area** is a region consisting of a densely populated urban core and its less-populated surrounding territories
  - i..e, a group of cities connected a central one

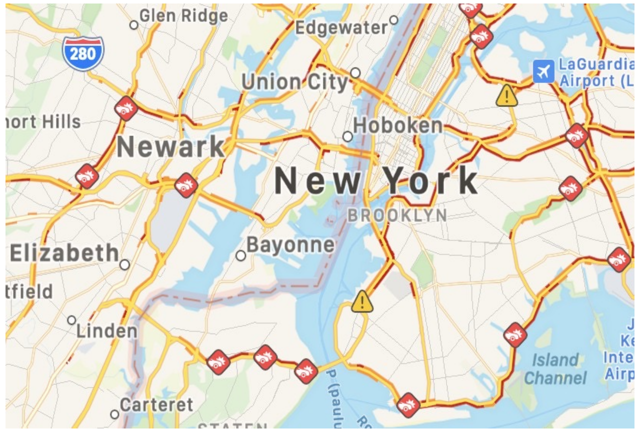
Problem of the day

- We want to group cities into metropolitan areas, then find the n largest metropolitan areas in the US population
- For example:
  - **Chicago:** Chicago, Evanston, Cicero, Des Plaines, etc.



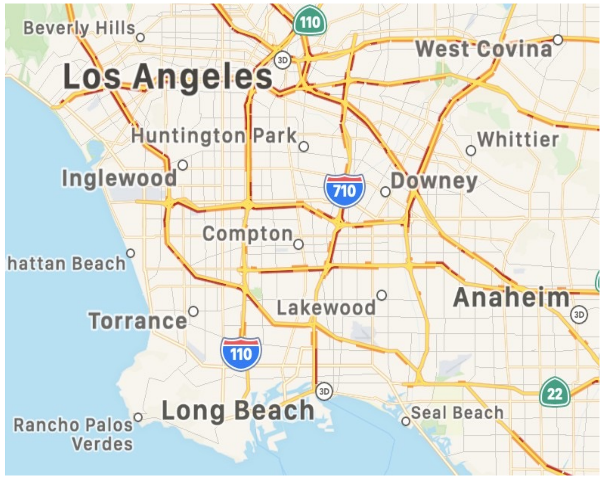
Problem of the day

- We want to group cities into metropolitan areas, then find the n largest metropolitan areas in the US population
- For example:
  - **New York:** New York City, Newark, Union City, etc.



Problem of the day

- We want to group cities into metropolitan areas, then find the n largest metropolitan areas in the US population
- For example:
  - **Los Angeles:** Los Angeles, Anaheim, Long Beach, etc.



Problem of the day

- We want to group cities into metropolitan areas, then find the n largest metropolitan areas in the US population
- The source data we have available is:
  - A collection of information about all US cities (name, population, surface area, etc.)
  - Road network data: which cities are connected by road and road length
  - Census data reporting demographic information, median income, educational attainment, etc.
- From this, we'll need to discover which cities are part of which metro areas, compute their total population, etc.

Before you start

- Make sure you understand the whole problem and requirements in full

- **Never** jump straight into programming!
  - **A very bad idea**
  - You may end up writing a bunch of bad and incorrect code from the start and then can't bring yourself to let go of it once you realize it's wrong
  - **Always** develop a plan before you program!
  - Will save you a lot of programming time and make you program **intentionally** and **intelligently**

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
    1. Our *entities*
3. Determine how these entities connect with each other
    1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
    1. Our *entities*
3. Determine how these entities connect with each other
    1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 1: The problem

- We first want to make sure we understand the question fully
- Think of any clarifying questions you have about the problem and specification
  - Anything that is still not clear or specifically stated, you use your discretion intelligently

Step 1: The problem

- I might ask:

1. How do we determine if a city is in metro area X?
   - **Answer:** The city must be connected by road to city X (center)
   - And it must be smaller (less population) than city X
2. Should the metro area include the city itself?
   - **Answer:** Yes!
3. Can metro areas overlap?
   - **Answer:** In reality, no, but for simplicity, we'll allow it here

Step 1: The problem

- Distilled goals:

1. Given cities' information and the road connections between cities, identify the metro area corresponding to each city based on its neighboring cities with lower populations

2. Given the above set of metro areas identified above, determine the n most populated metro areas by the total population across all the cities in the metro area

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
    1. Our *entities*
3. Determine how these entities connect with each other
    1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 2: Our entities

- Activity: Take 5 minutes to discuss with your neighbor what kinds of data you think we'll need to represent to solve our problem
  - Think: "nouns" , "things" relevant to the problem domain (i.e., metro areas)
  - **We don't want to think of ADTs or data structures yet!** Just "things"

Step 2: Our entities + In-class exercise (5 minutes)

- Activity: Take 5 minutes to discuss with your neighbor what kinds of data you think we'll need to represent to solve our problem
  - Think: "nouns" , "things" relevant to the problem domain (i.e., metro areas)
  - **We don't want to think of ADTs or data structures yet!** Just "things"

- List all pieces of data you came up with in the form
  - There should be more than one
- Then I'd like to hear from you all

Step 2: Our entities

- Here’s what I came up with:
  - Cities
  - Roads
  - Metropolitan areas

- 1 city: name, population, surface area
- 1 road: between two cities, road length
- 1 metro area: info about its member cities, the name of the city center, total population of the metro across cities

What about the demographic data?

- Just because you have data doesn't mean you should use it!
  - **Obvious reason:** it may not be useful for your problem (like here)
  - **Subtle reason:** some data can introduce bias into your program
    - Biased data in, biased outcome out
    - E.g., using historical mortgage data to create a system that evaluates mortgage applications
  - **Deeper reason:** should you even have that data in the first place?
- See also: "CS 312: Data Privacy" and "CS 396: Computing Ethics & Society"

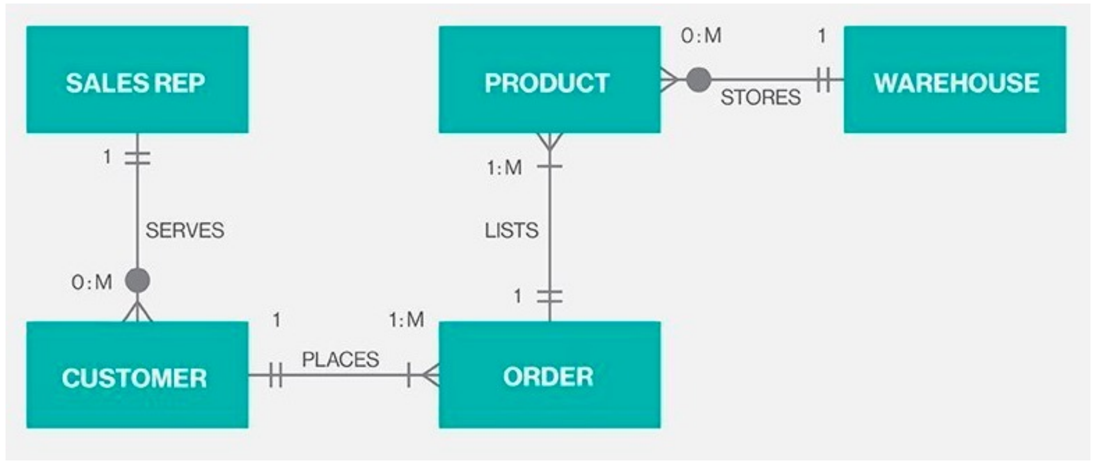
Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
   1. Our *entities*
3. Determine how these entities connect with each other
   1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 3: Our relations

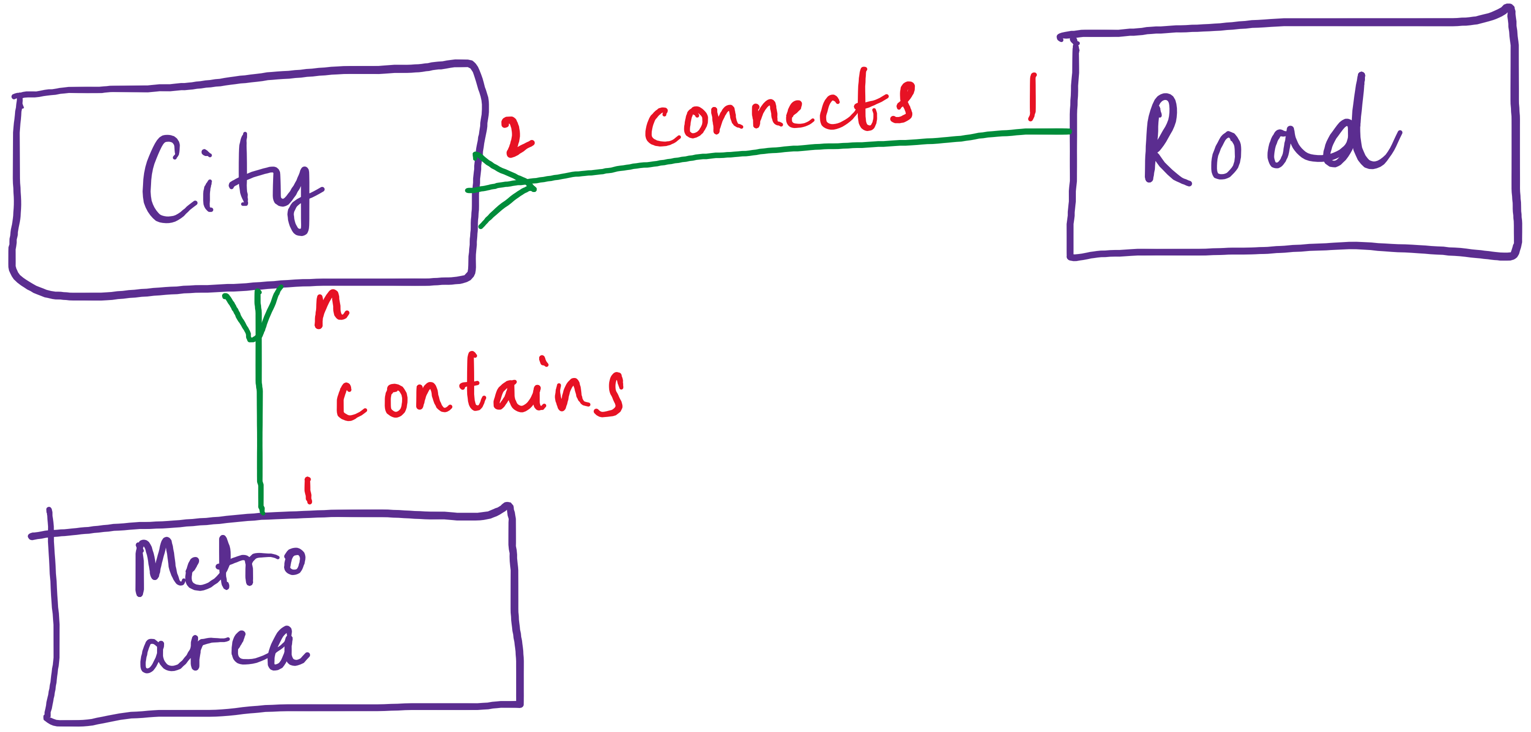
- Let’s use a rough Entity-Relation Diagram (ERD)
  - Maps out kind of data we’re working with (*entities*) and how they relate to each other (*relations*)

- **Note**: some people get very serious about diagrams, exact symbols used, rules for what can go where, etc.
  - I prefer a more informal approach



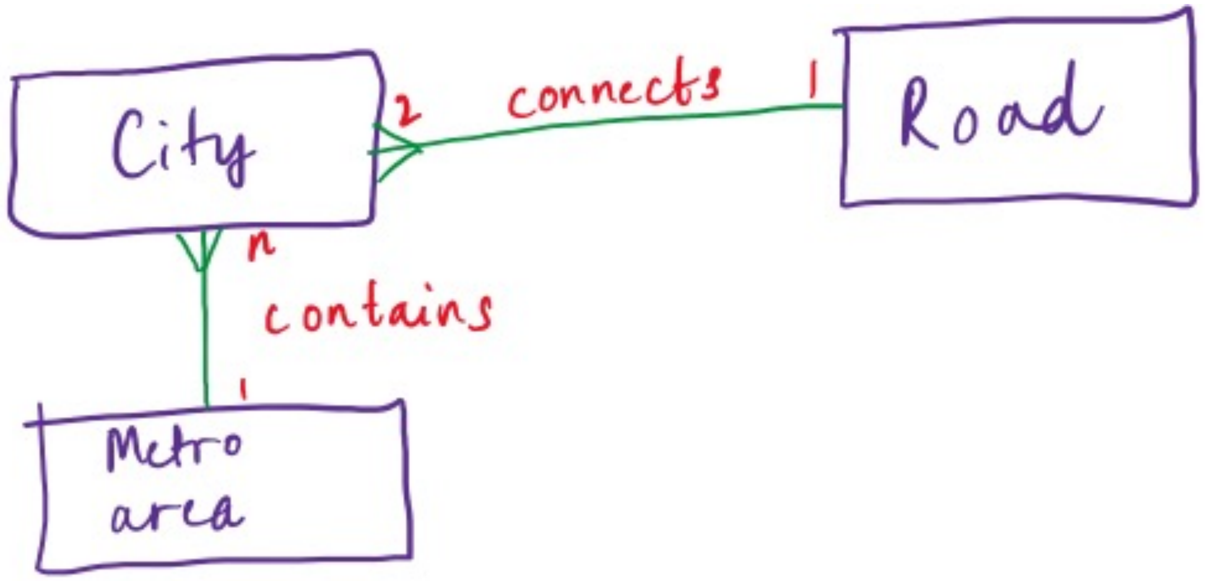
Step 3: Our entities

- An ERD for our entities



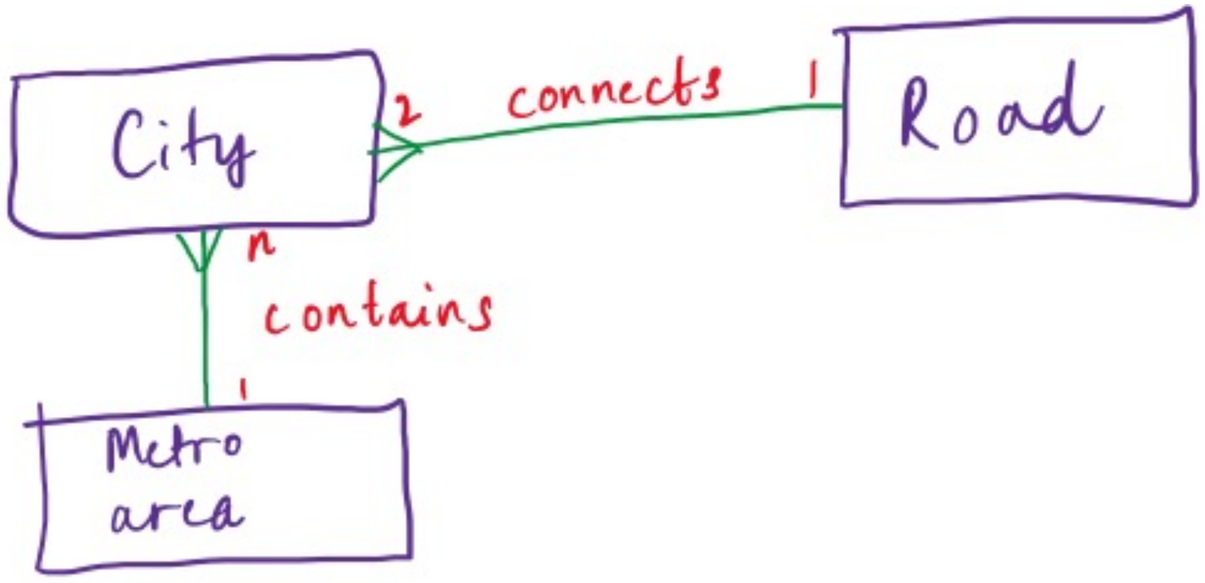
Step 3: Our relations

- Entity-Relation Diagram (ERD) maps out:
  - Kind of data we're working with (*entities*)
  - How they relate to each other (*relations*)
- Each entity box represents a *kind* of data
- We could have multiple *instances* of each!
  - Multiple cities, etc.



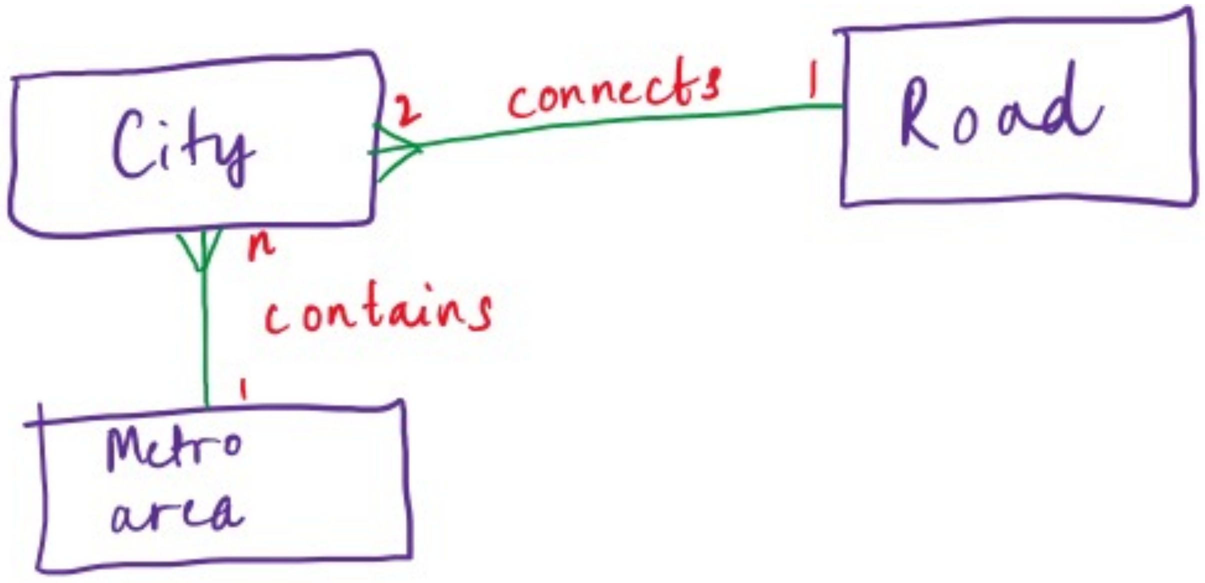
Step 3: Our relations

- Entity-Relation Diagram (ERD) maps out:
  - Kind of data we're working with (*entities*)
  - How they relate to each other (*relations*)
- Each relation has a *cardinality*
- How many instances of each does it relate:
  - 1-1, 1-N, N-M, etc.



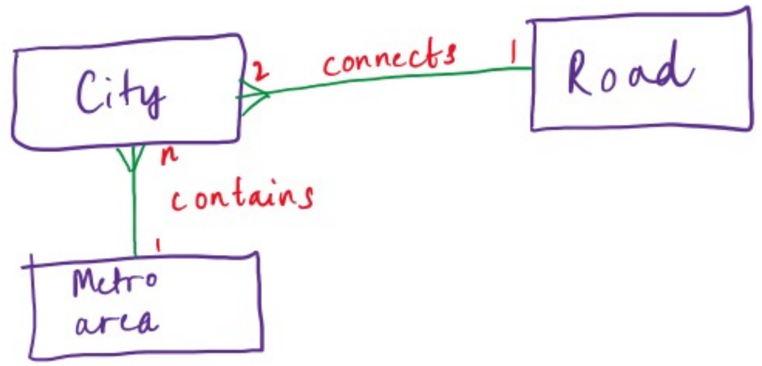
Step 3: Our relations

- Entity-Relation Diagram (ERD) maps out:
  - Kind of data we're working with (*entities*)
  - How they relate to each other (*relations*)
- Each road *connects* two cities: **1-to-2** relation
- Each metro area *contains* n cities: **1-to-n** relation



Step 3: Our relations

- Good first guesses for kinds of data we'll need in our program
- We may discover more as we go along
- Or some may turn out unnecessary



- Heads up: You'll have to draw one for your project
- Refer to:
  - Notation cheat sheet on canvas
  - Also: www2.cs.uregina.ca/~bernatja/crowsfoot.html

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
    1. Our *entities*
3. Determine how these entities connect with each other
    1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 4: Our ADTs

- How can we represent one of each entity below?
  - ADTs or building blocks (structs/vectors)?

- **City**
  - struct with city name, population, etc.
- **Road**
  - struct with city1 name, city2 name, length
- **Metro area**
  - struct containing: set of above city structs, name of the center, total population

Step 4: Our ADTs

- Let's go a level above that. We need to represent multiple of cities, roads, and metro areas, and relations

1. **Info about all cities**
2. **Info about all roads**
3. **Connections between cities**
4. **Info about all metro areas**
5. **All cities within a metro area**

Step 4: Our ADTs

- Take a couple minutes with your neighbor and think about how we can represent the following with **ADTs** relevant to our program?

	ADT
Info about all cities	Dictionary (name -> city struct) Set ADT (item: city struct)
~~**Info about all roads**~~	
Connections between cities	WU graph (nodes: cities, edges: roads, weights: distances)
Info about all metro areas	Dictionary (name -> metro struct) Set ADT (item: metro struct)
All cities within a metro area	Set ADT (item: city struct)

ADT cheat sheet

- A few hints for when to reach for various ADTs
  - Collections of many things → **set** (arrays, linked lists, sorted array)
  - Mapping from one thing to another → **dictionary**
  - Things connected to other things → some **graph**
  - Todo-lists → worklist
    - First-in first-out → **queue**
    - Last-in first-out → **stack**
    - Most-urgent-out → **priority queue**

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
   1. Our *entities*
3. Determine how these entities connect with each other
   1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 5: Our data structures

- Finally, let's pick concrete data structures

	ADT	Concrete DS	Reason?
Info about all cities	Dictionary Set ADT	Hash table Array	We may have a lot of data (all cities)
Connections between cities	WU graph	Adjacency list	Sparse graph, space complexity matters
Info about all metro areas	Dictionary Set ADT	Hash table Array Linked list	Lot of data (# cities) Fixed amount of data
All cities within a metro area	Set ADT	Array or linked list	Flexibility of linked list

How to pick concrete data structures?

- Efficiency and suitability for purpose
  - Complexities on their own don’t tell us much; relate them to the problem at hand
  - Optimize for the frequent operations
  - When you know there will be few or huge numbers of elements in your DS, use that domain knowledge
- Availability
  - Do I have a high quality implementation that I can trust?
  - Standard library > 3rd-party package > GitHub repository > StackOverflow answer > random forum post from 2005

How to pick concrete data structures?

- When in doubt, keep it simple
  - If it's sufficient, great! Low effort, working solution to start with
- Later, *if and only if* it's not good enough…
  - …*and* it's actually the problem
  - …*then* replace it with something more clever

Recipe for data design

1. Understand the *problem*, and break it up into sub-goals
2. Identify the data we need to solve the problem
    1. Our *entities*
3. Determine how these entities connect with each other
    1. Our *relations*
4. Map each entity and relation to an *ADT* or kind of box
5. For each ADT, choose a *data structure* to represent it
6. Start writing your *implementation*, using the above
7. Go back to steps 1-5 if something doesn't work out

Step 6: Our implementation

- Two parts:

1. Using the information from steps 1-5, write out in words and/or pseudocode how we can achieve our goals

2. *Then* we'll go program!

Step 6: Our implementation

1. Given cities’ information and the road connections between cities, identify the metro area corresponding to each city based on its neighboring cities with lower populations
    1. Read in raw city data and put them into structs and then store each city struct in the dictionary against its name OR store all structs in an array
    2. Read in raw road city, build a WU graph, set all the edges according to the road data and add weights according to the length
    3. For every city in our dataset, maintain a metro struct. Find all that city’s neighbors in the graph, identify if their population is lower, and if so, store as part of the member cities for that metro struct
    4. For every member city in a metro, compute total population

Step 6: Our implementation

2. Given the above set of metro areas identified above, for a given n, determine the n most populated metro areas by the total population across all the cities in the metro area

- Two ways:
  - Sort the all_metros array according to total_population and take the first n
  - Put all metro structs into a priority queue, and then find and remove_min n times
- Picking a sort: selection, merge, quick, heap sort O (n log n)

Step 6: Our implementation + in-class exercise (5 minutes)

2. Given the above set of metro areas identified above, for a given n, determine the n most populated metro areas by the total population across all the cities in the metro area

In the form, briefly write out an approach in words (a few sentences) to describe how you would achieve the above

- Use any ADT or data structure we've learned so far
- Remember each metro area struct has: center city name, member cities, total population across members

Let’s go program now!

- I’ll start implementing inside `metro-starter.rkt`
  - As we program, we may still figure things out we need to decide or change in our data design (Step 7)
  - We’ll work with a small subset of cities in tri-state area for now
- I’ll upload final code after Thursday
- Programming along with me may be distracting for you
  - You may miss the actual process by trying to keep up
  - Instead, keep starter code open and make notes if you’d like but observe and process what we’re doing live

Let’s shake things up

- What if we:
  - Want to get n metro areas with the largest number of cities
  - Want to get n metro areas by total surface area
  - Want the n largest metro areas in a specific state
  - Want the n largest metro areas in each state
- What changes would be required for each?
  - Code changes only?
  - DS changes (within the same ADT)?
  - Data design adjustments?
  - Complete redesign?

Beware exclusionary data design

- Data design defines what data is allowed within a system
  - Flip side: everything else is disallowed
- Example: representing people’s names

```
struct person:
  let first_name
  let last_name
```

- Boom. Done. Right?
  - Well, what about middle names? Or people with three first names?