


Lecture 41: Graphs/
Inheritance

+ Today

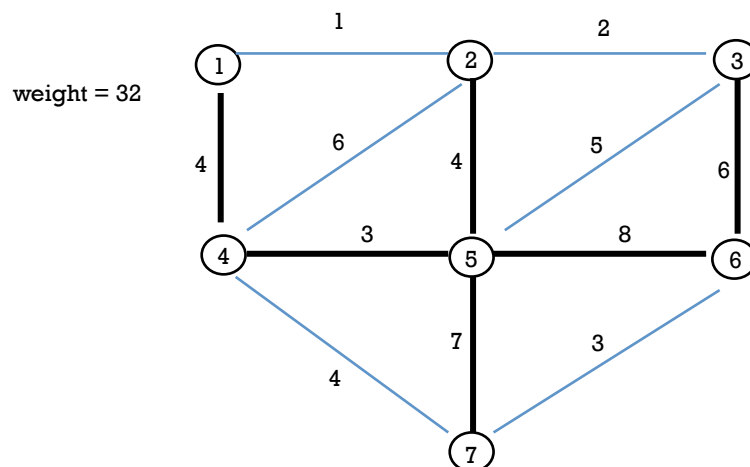
- Reading
 - JS Ch. 16
 - Weiss Ch. 6
- Objectives
 - Prim's algorithm for minimum spanning trees
 - Inheritance in C++



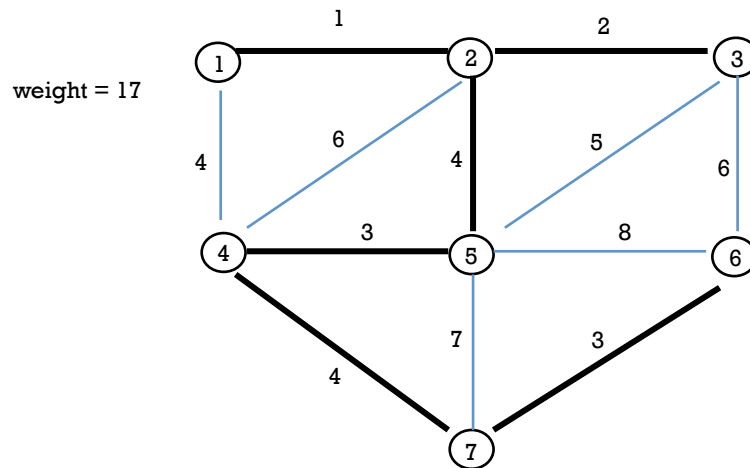
+ Minimum Spanning Trees

- $G' = (V', E')$ is a subgraph of $G=(V,E)$ if V' is a subset of V and E' is a subset of E
- A spanning tree is a subgraph of G that is a tree and connects all of the vertices together
- A minimum spanning tree is a minimum weight spanning tree
 - Weight is the sum of the weights of the edges in the MST

+ A Spanning Tree



+ The Minimum Spanning Tree



+ Prim's Algorithm

- Algorithm for finding a minimum spanning tree
- Runs on a connected, weighted (possibly negative), undirected graph
- Greedy algorithm – makes the greedy choice each time
- Basic algorithm:
 - Initialize tree with randomly chosen vertex
 - Find minimum weight edge that connects tree to vertices not yet in tree
 - Add this edge/vertex to the tree

+ Prim's Algorithm

- Data structures:
- For each vertex v , $\text{key}[v]$ is least-cost edge (found so far) joining v to tree
- For each vertex v , $\text{parent}[v]$ is vertex u in $\text{edge}(u,v)$ that added v to the tree
- Q is priority queue ordered by least-cost edges (i.e. by key)

+ Prim's Algorithm

```

prim(g) {
  // initialization
  pick start node r
  foreach(u in V-{r}) key[u] =  $\infty$ 
  key[r] = 0; parent[r] = null;
  add all vertices to Q (by key)

  // each iteration adds one node to MST
  while(!Q.empty()){
    u = min node from Q
    foreach v adjacent to u
      if v in Q and edge_weight(u,v) < key[v]
        parent[v] = u; key[v] = edge_weight(u,v)
        adjust priority of v in Q
  }
  return parent
}

```

+ Inheritance in C++

- Finish up our discussion of C++ with inheritance
- Default parameters
 - Specifies a value to use if input argument is not given
- Syntax of declaring a subclass

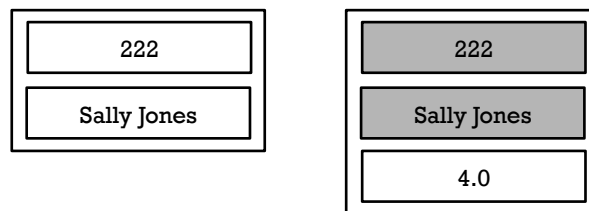
```
class Student : public Person
```
- Public inheritance implements an “isA” relationship
- Constructor for subclass must call constructor for base class

+ The virtual keyword

- Solution: determine at runtime (not compile time) which function to call
- This is known as *dynamic dispatching*
- The `virtual` keyword signals that the function uses dynamic dispatching
- Allows this function to be overwritten in subclasses
- If don't use `virtual`, the function called is based on compile-time type not runtime type

+ Slicing

- Slicing – when a derived class is copied into base class, only the instance variables from the base class are preserved
- Slicing occurs whenever objects are copied
 - For example, call-by-value



+ Slicing

- Assignment of subclass object to base class variable results in slicing – converts to base class!
- Assignment of pointers works as expected!
- Bottom line: if want subtyping, use pointers or call by reference. Copying destroys subtyping
- In particular, if you want a vector of Person or subclass,

```
vector<Person*> people;
```

+ Casting in C++

- Type casts in C++ always succeed!
- Downcasting on pointers always succeeds!
- To get checked conversions, use `dynamic_cast`
 - Returns NULL if the cast is incorrect
 - `dynamic_cast` does a compile time *and* runtime check to make sure the cast can work
 - requires that the object you're casting has polymorphic type, i.e. has at least one virtual method

+ Writing Java classes in C++

- Declare every method as virtual
- Only allocate objects with `new`
- Access all objects by pointer
- Use collections of pointers (not objects)
- *Must worry about manual garbage collection!*