# Graphs and Trees

# Note

- Out of town Thursday afternoon

- Willing to meet before 1pm, email me if you want to meet then so I know to be in my office

# A few extra remarks about recursion

- If you can write it recursively you can also write it iteratively, and vice versa
  - Church-Turing Thesis

- But why pick one method over the other?

# Example: Fibonacci numbers

```
int fib(int n) {
    if(n == 1 || n == 0) return 1;
    else return fib(n-1)+fib(n-2)
}
```

```
int fib(int n) {
    int current = 1; int previous = 1; int temp;
    if(n == 0 || n == 1) return 1;
    for(int i = 2; i < n; i++) {
        temp = current + previous;
        previous = current; current = temp;
    }
    return current;
}
```

# Results...

- Recursive may be easier, but may not be the best way

- Work out the best way before moving forward

# Graphs and Trees

- Represents a **lot** of things
- Formally a graph G is a pair of sets, G= (V,E)
- V - The set of vertices
- E - the set of “edges”
- Each edge connects 2 vertices together

# Terminology

- **Lots** of terminology surrounding graphs, tons of types, specific patterns, etc...
- Tip: Always read the description of graphs carefully - they describe a lot in just a few words

# Graph terminology

- Undirected - you can travel along an edge in either direction
- Directed (Digraph) - One-way travel only (but still possible to have 2 edges!)
  - Note: All undirected graphs can be converted into directed graphs, but not the other way around!

- Unweighted - Edges have no value associated with them
- Weighted - Edges have a value associated with them

- Examples on board

# Yet more terminology

- Cycle - A set of edges that forms a “loop”

- Cyclic - Graph contains a cycle
- Acyclic - No cycles allowed in the graph

- DAG - Directed Acyclic Graph

- Self-loop: An edge from a vertex to itself (yes, this counts as a cycle)

- Simple Graph - Shorthand for “Undirected Unweighted No-self-loops graph”

# Yes, there's still more

- Degree (undirected only): The number of edges connected to a vertex
- In/Out-degree (directed only): The number of edges entering/leaving a graph
- Regular - All vertices have the same degree
- (Simple) Path - A sequence of (distinct) edges going from *u* to *v*
- Connected - For any two vertices there exists a path between them
  - Directed graphs - strongly connected components

# Special graphs

- Clique ($K_n$) - A graph will all possible edges on *n* vertices
- Bipartite - A graph with two sets of vertices. Edges only allowed between sets
- Complete Bipartite ($K_{n,m}$) - A bipartite graph with all possible edges
- Empty graph - No vertices or edges
- Cycle Graph - One giant cycle, no other edges allowed
- Path Graph - All linked in a line

# And...

- Tree: Any acyclic graph with *n* vertices and *n-1* edges.

- Or.. An acyclic directed graph where there is only one path between any two vertices

- Forest - A collection of trees

# Graph representations

- Adjacency Matrix - a |V| x |V| matrix where A[*u*][*v*] represents the existence and weight of an edge from *u* to *v*
  - Good for dense graphs, easy to work with
  - $O(n^2)$ memory

- Adjacency list - A list of lists. Each list represents a vertex, and contains the vertices it is connected to
  - Good for sparse graphs, harder to work with
  - $O(n + m)$ memory

- Example on board

# Trees

- Used in a lot of more complex data structures
- Often a single vertex is designated as the root
- Root is on top. Not on bottom
- Allows a natural direction for edges

# Tree terminology

- Root - The “top” of the tree

- Children - Those connected one layer below
- Descendant - Can get to these nodes by only moving downward

- Ancestor - Node that can be reached by following parents

# Minimum Spanning Trees

- Consider a **weighted** graph G

- Find a set of edges that forms a tree (or forest, if G is not connected) within G in such a way that the sum of the edges is minimized

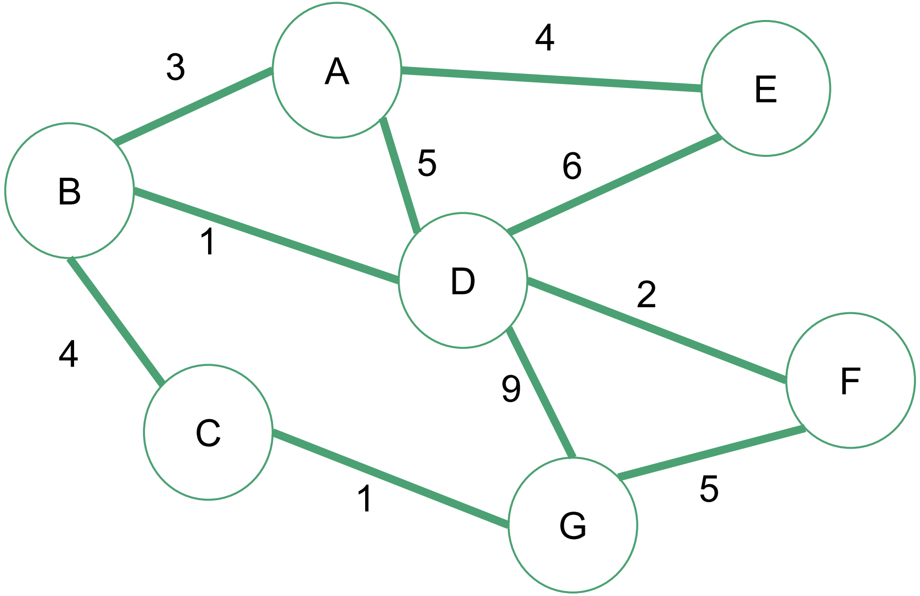
- Essentially, find the cheapest possible spanning tree that you can

# Prim’s Algorithm

- Idea: Take a greedy approach to making a MST

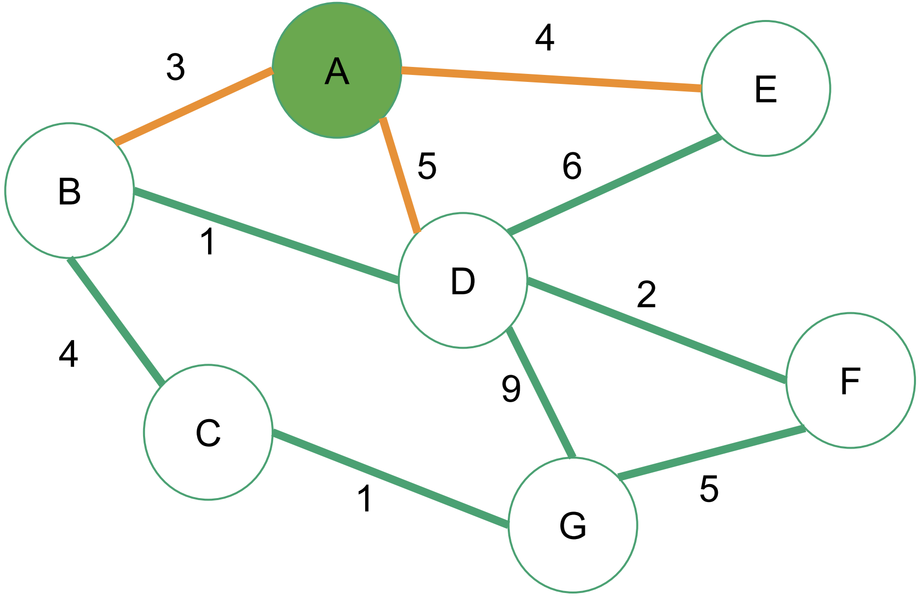
- Two versions: Adj. Matrix and Adj. List
- Matrix - $O(n^2)$
- List - $O(m \log n)$ [$O(E \log V)$]

- Gist: Pick a starting point. Repeatedly add the closest edge to the current tree to the tree until you’re done

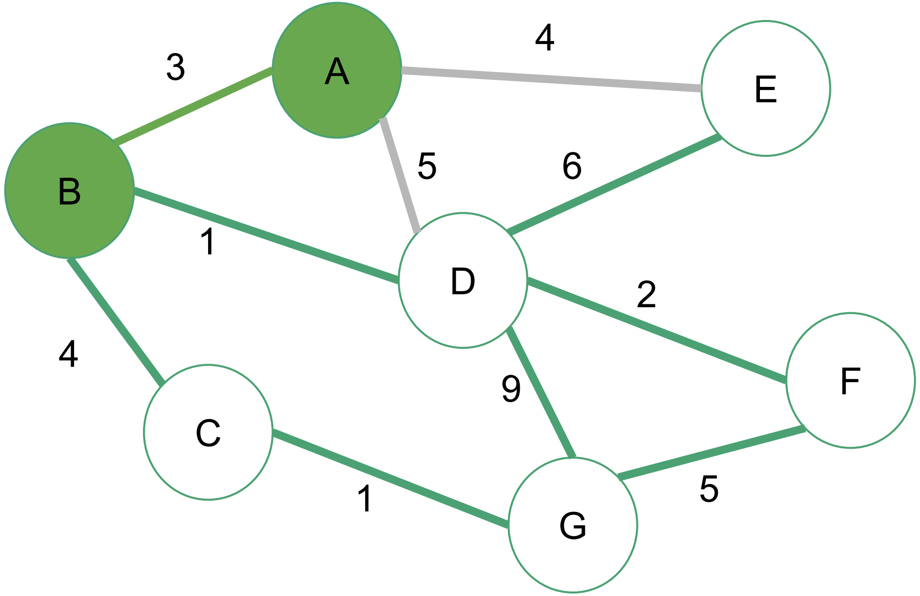


A
B
C
D
E
F
G
3
4
5
6
1
2
4
9
1
5

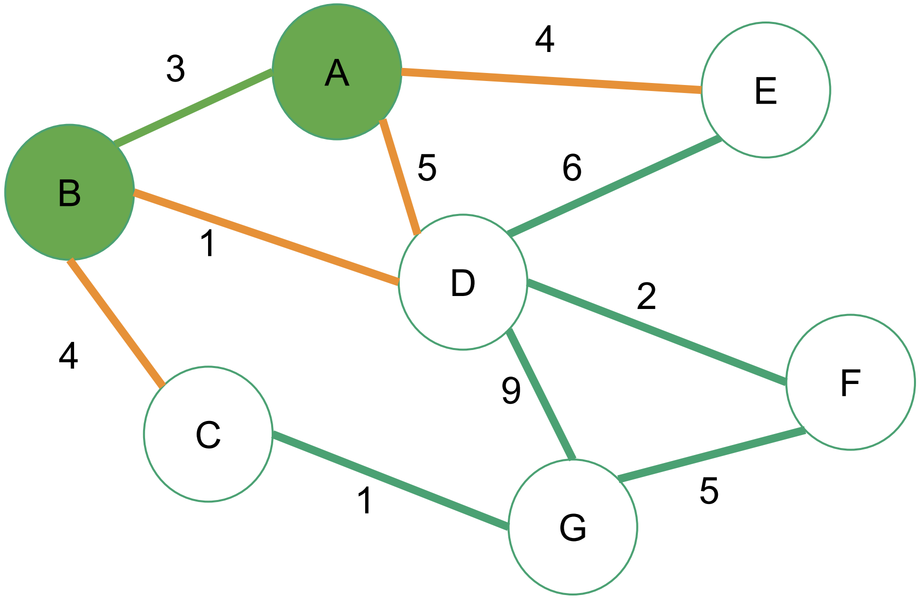
# Start with A. Find “closest” vertex to tree



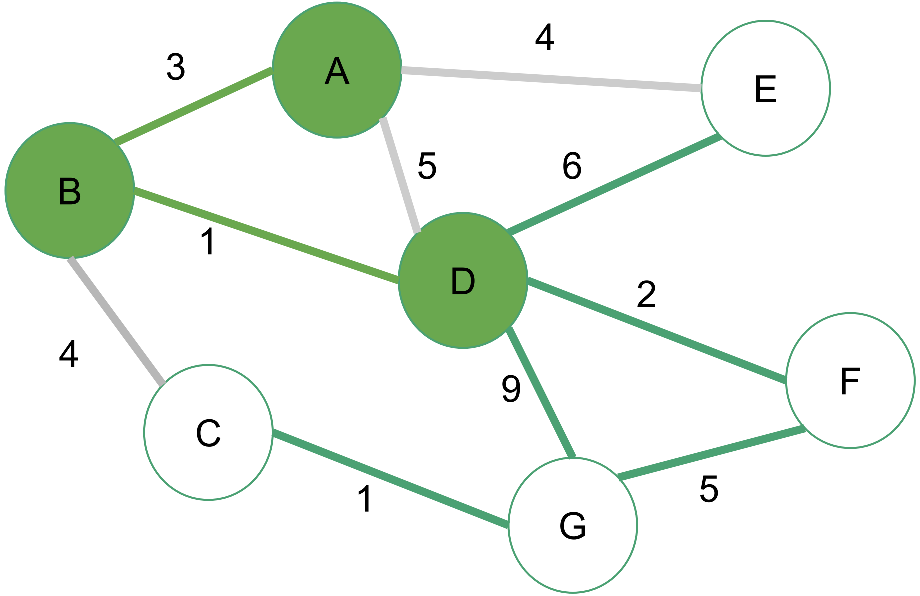
# Start with A. Find “closest” vertex to tree



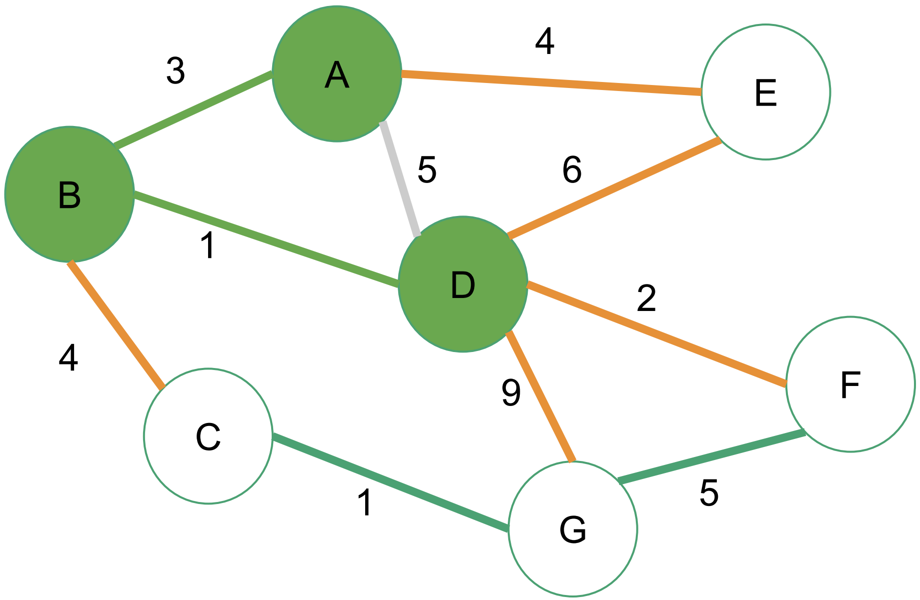
# Repeatedly find the closest vertex to the tree



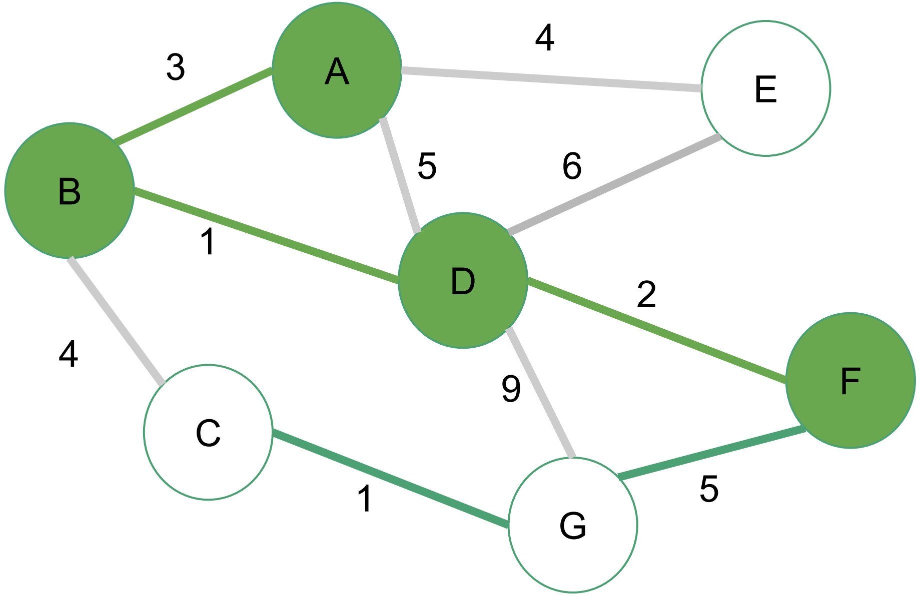
# Repeatedly find the closest vertex to the tree



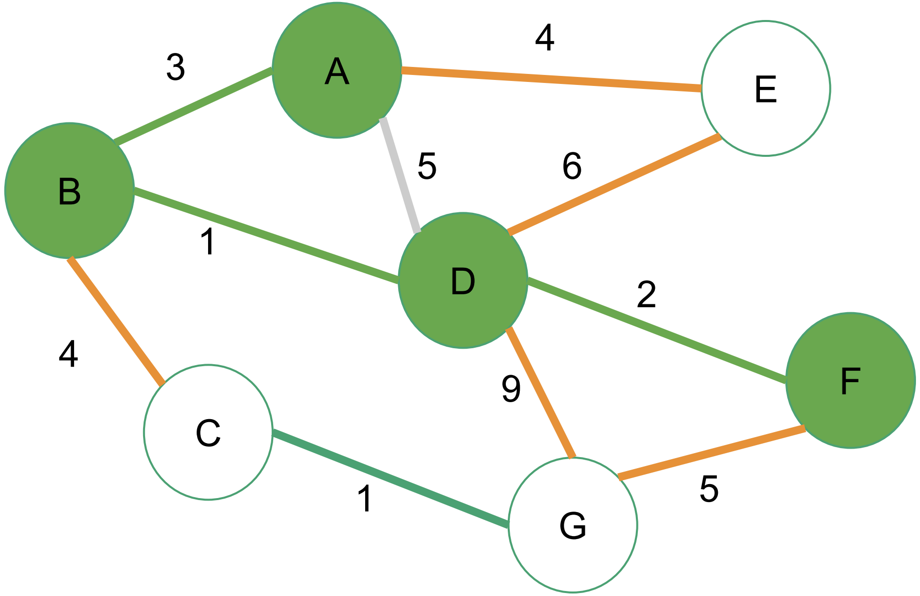
# Repeatedly find the closest vertex to the tree



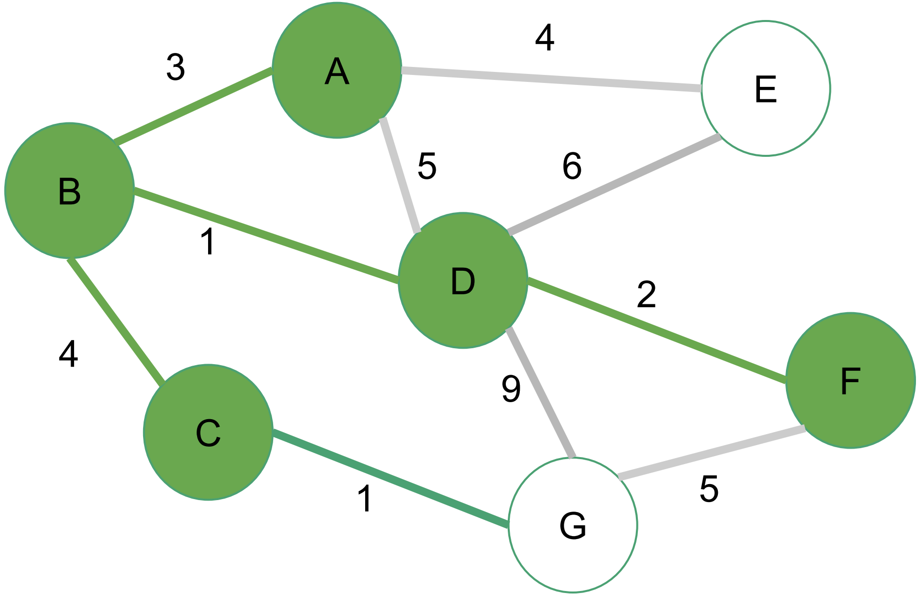
# Repeatedly find the closest vertex to the tree



# Repeatedly find the closest vertex to the tree

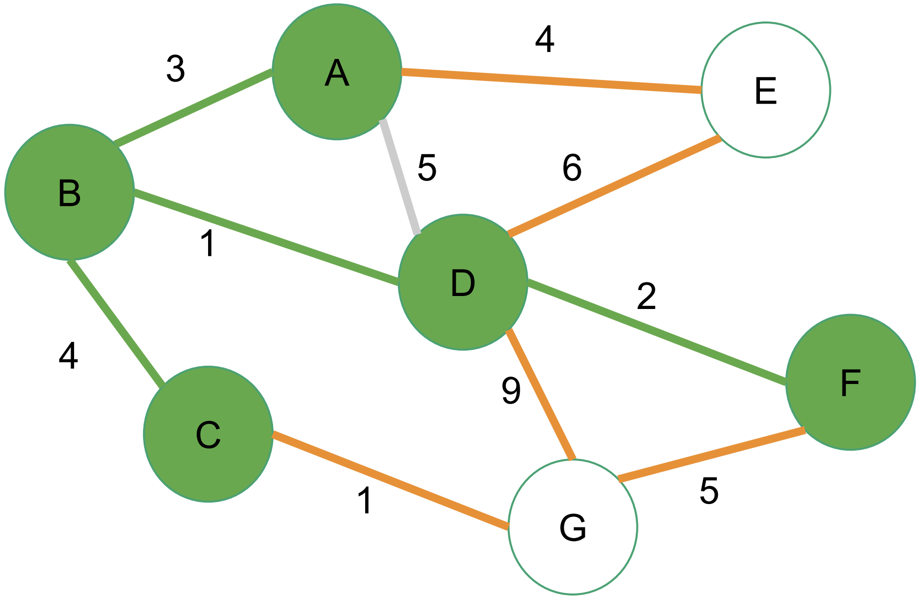


# Repeatedly find the closest vertex to the tree

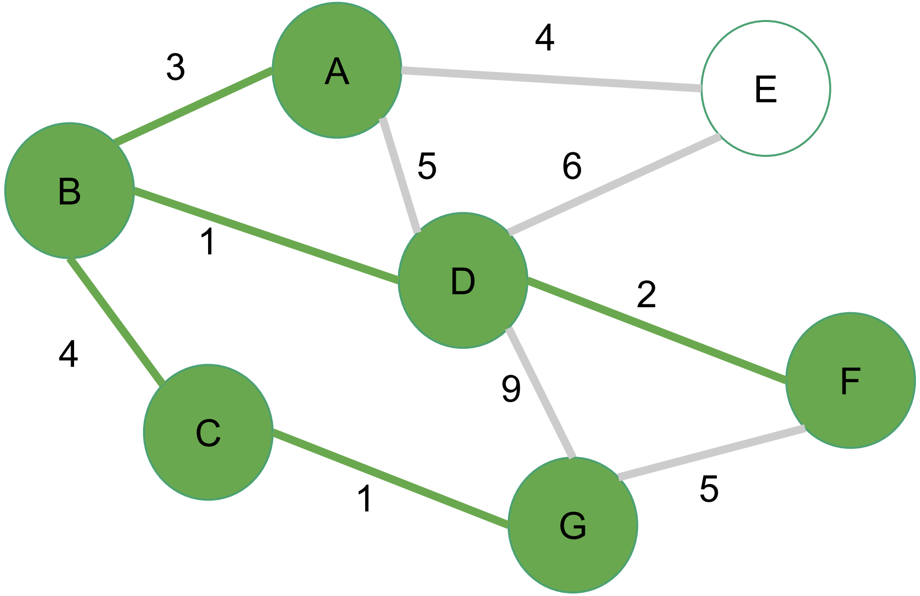



Arbitrary tiebreaker:
Alphabetical order

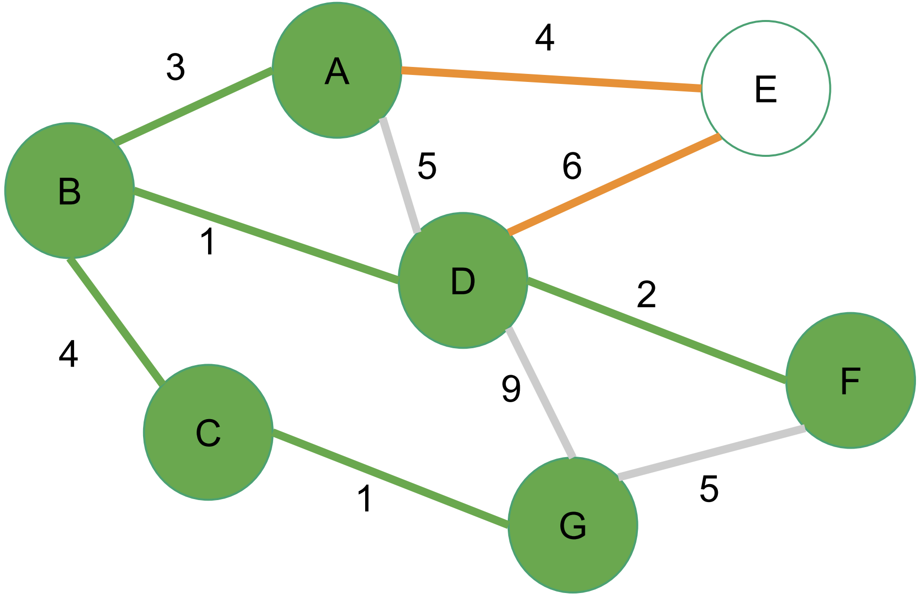
# Repeatedly find the closest vertex to the tree



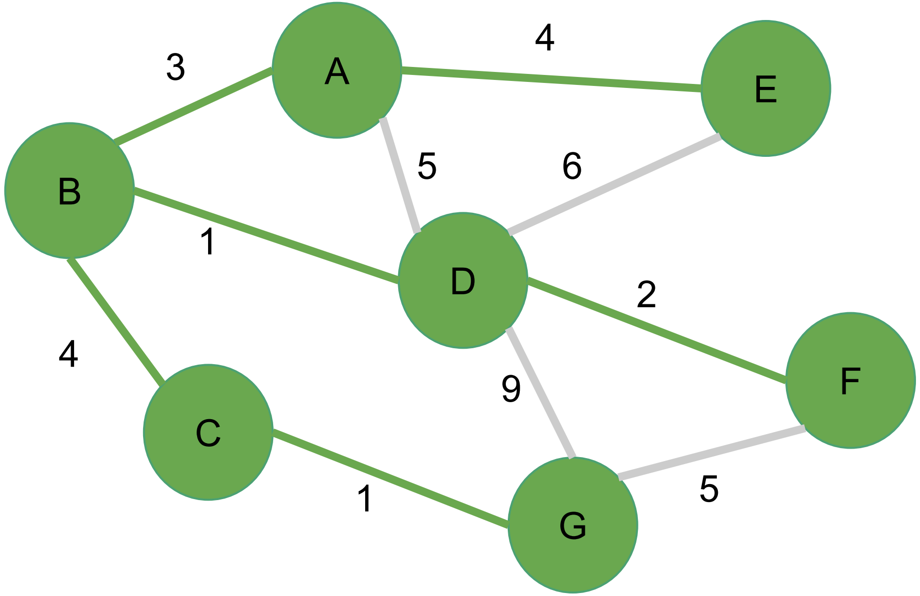
# Repeatedly find the closest vertex to the tree



# Repeatedly find the closest vertex to the tree



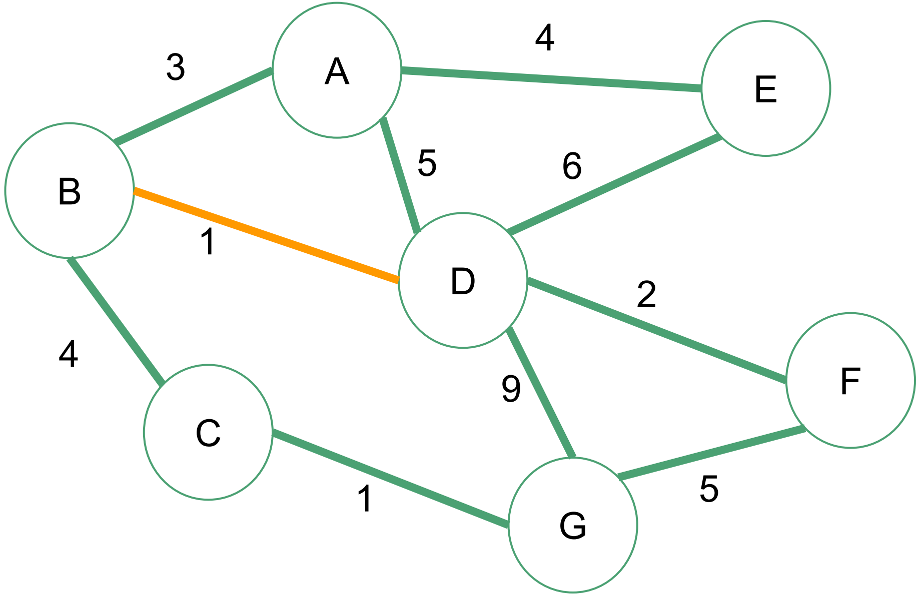
# All N nodes in the tree. Done!



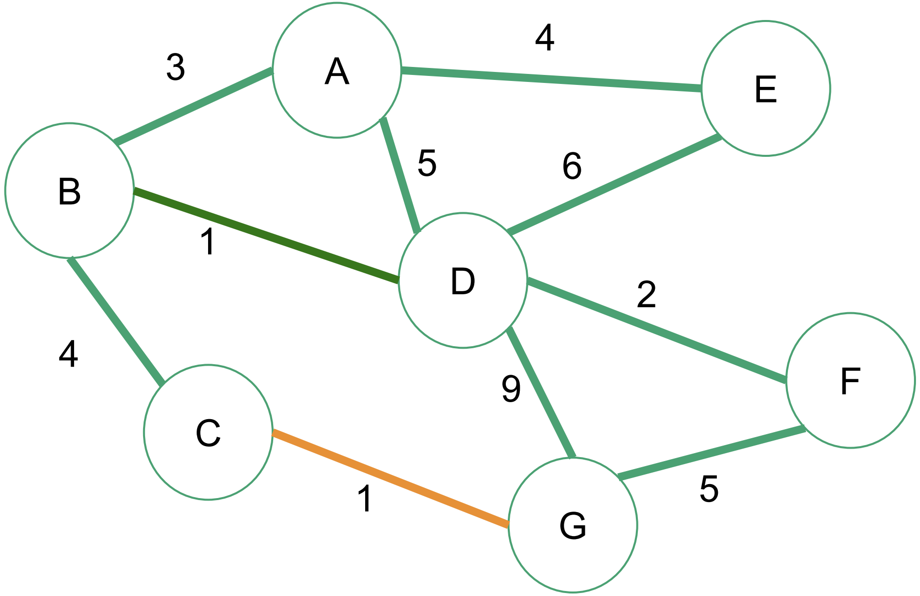
# Kruskals algorithm

- Repeatedly pick the smallest edge that has not been visited
- If it does not make a cycle, add it to the tree
- Initialized as a forest of singletons
- Tree is stored as a Union Find. Gives us the runtimes we want

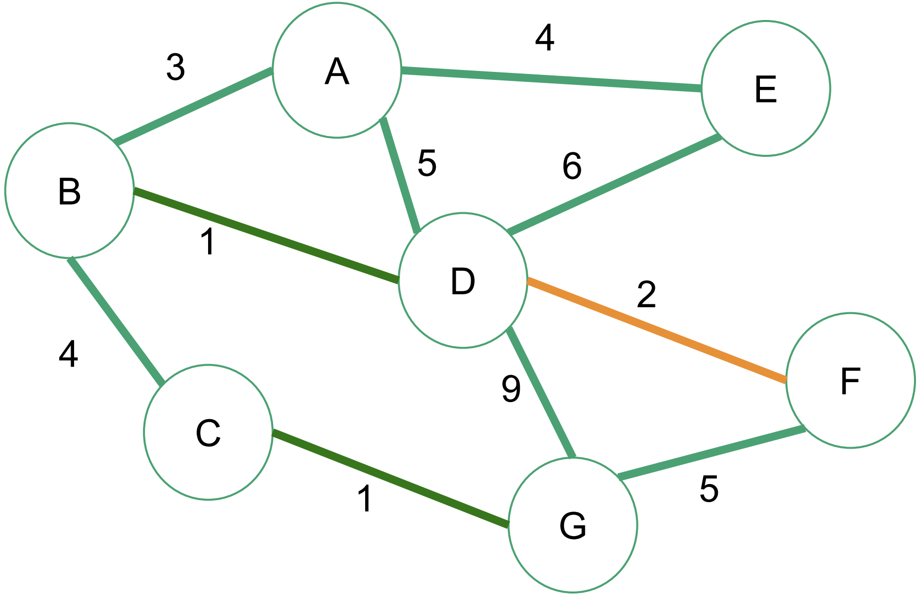
# Pick edges, if there is no cycle, add to tree



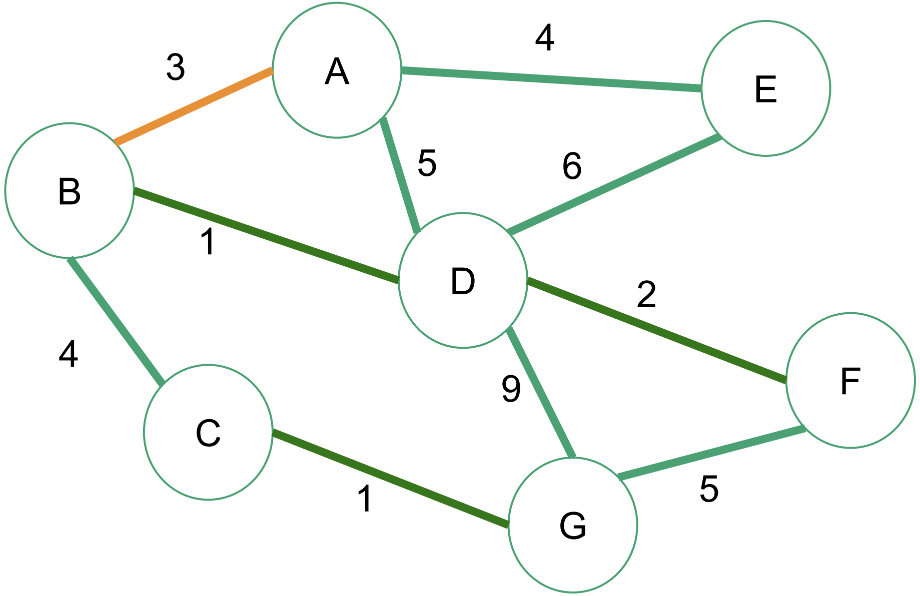
# Pick edges, if there is no cycle, add to tree



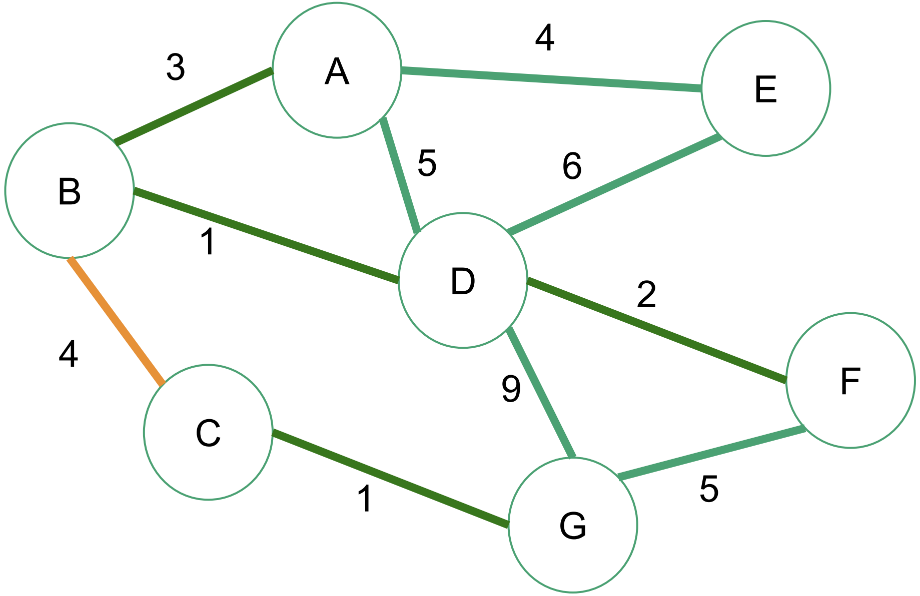
# Pick edges, if there is no cycle, add to tree



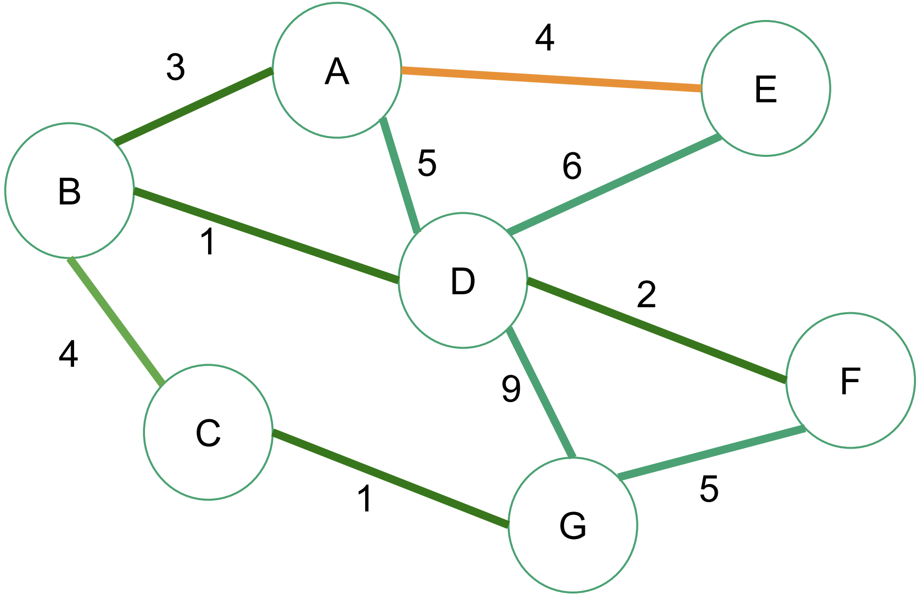
# Pick edges, if there is no cycle, add to tree



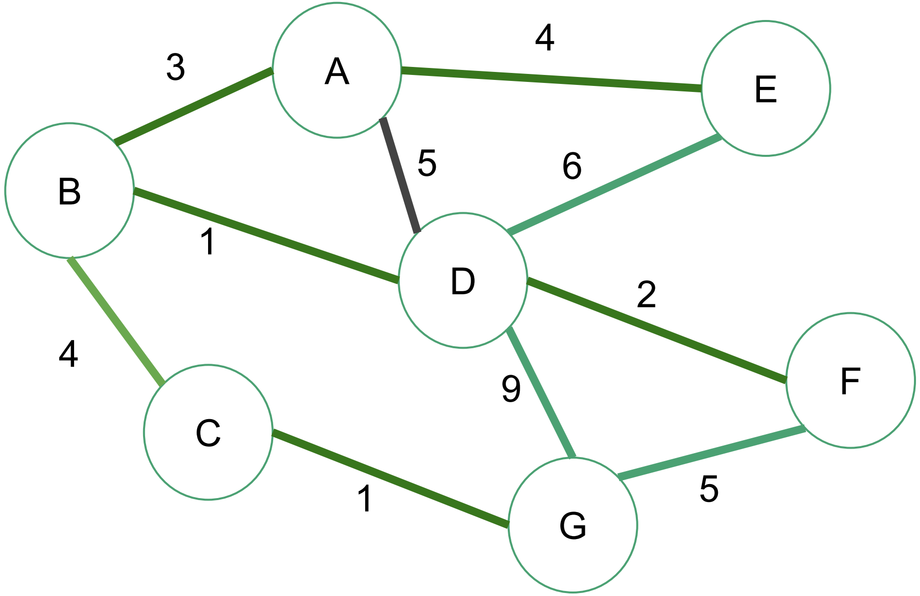
# Pick edges, if there is no cycle, add to tree



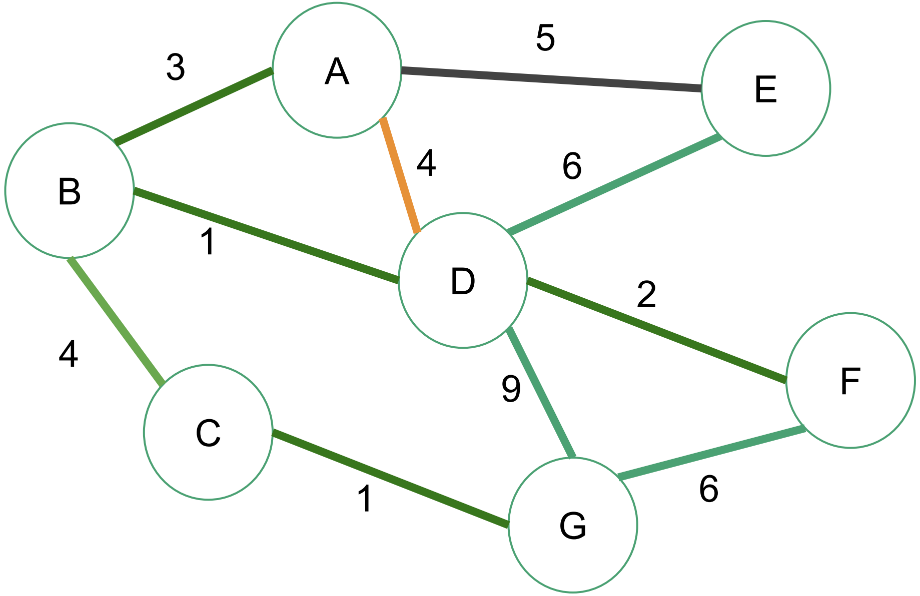
# Pick edges, if there is no cycle, add to tree



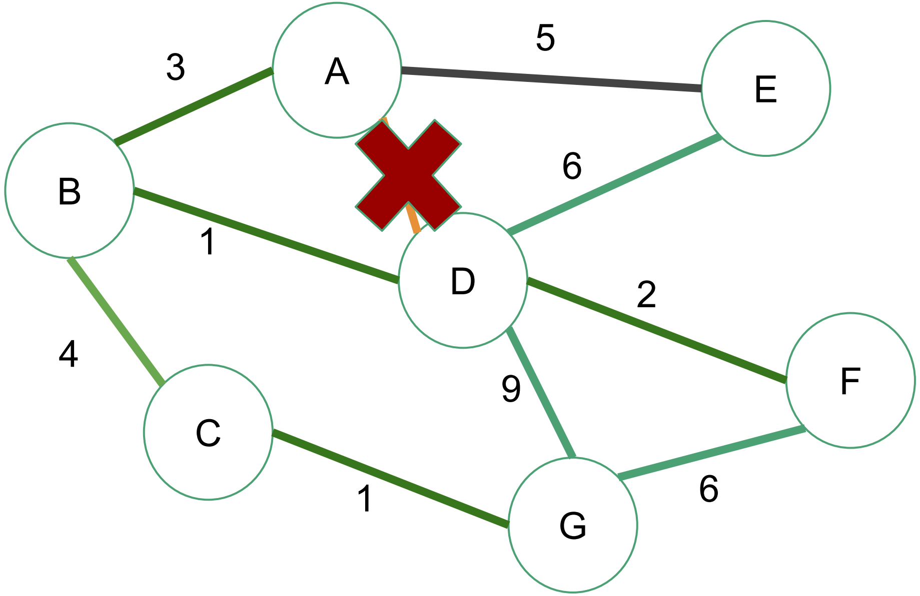
# All vertices connected. Done



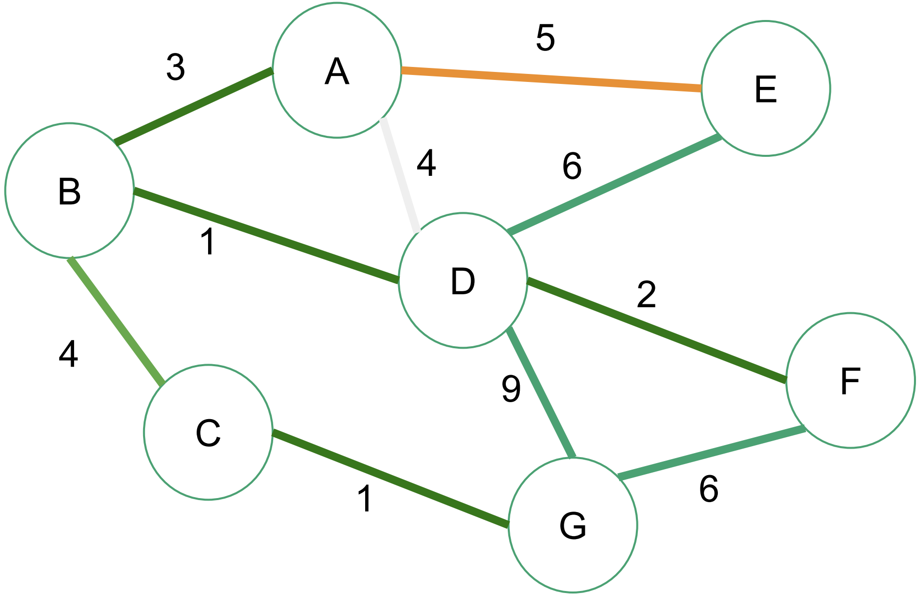
# Cycle case



# Cycle case



# Cycle case



# Cycle case