Section 11

CSCI E-22

Will Begin Shortly

Double Hashing

Recall:

- Double Hashing is a collision resolution scheme that uses *a second* hash function
  - *h1(x)* computes the hash code
  - *h2(x)* computes the interval for probing

Double Hashing

Recall:

- Double Hashing is a collision resolution scheme that uses *a second* hash function
  - *h1(x)* computes the hash code
  - *h2(x)* computes the interval for probing
- Combines good features of linear and quadratic probing
  - Reduces clustering
  - Can be proven to *always* find an open position if there is one, so long as the length of the table is a prime number

Double Hashing

For our example, we'll use the same keys from last time, and the following hash functions

- *h1(x):* index related to the first letter of the word (a = 0, b = 1, …)
- *h2(x):* length of the word (*h2*("apple") = 5)

Double Hashing

For our example, we'll use the same keys from last time, and the following hash functions

- *h1(x):* index related to the first letter of the word (a = 0, b = 1, …)
- *h2(x):* length of the word (*h2*("apple") = 5)

Let's go through inserting elements using double hashing and count the total length of the probes.

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

0	
1	
2	
3	
4	
5	
6	

apple, cat, anvil, boy, bag, dog, cup, down

Double Hashing

Probe length: 1

h1(apple) = 0

0	apple
1	
2	
3	
4	
5	
6	

apple, cat, anvil, boy, bag, dog, cup, down

Double Hashing

Probe length: 2

h1(apple) = 0

h1(cat) = 2

0	apple
1	
2	cat
3	
4	
5	
6	

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 3

0	apple
1	
2	cat
3	
4	
5	
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 4

0	apple
1	
2	cat
3	
4	
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 5

0	apple
1	boy
2	cat
3	
4	
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 6

0	apple
1	boy
2	cat
3	
4	
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 7

0	apple
1	boy
2	cat
3	
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 8

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 9

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 10

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5

apple, cat, anvil, boy, bag, dog, cup, down

Double Hashing

Probe length: 11

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5
(h1(cup) + 2 * h2(cup)) % 7 = 1

apple, cat, anvil, boy, bag, dog, cup, down

Double Hashing

Probe length: 12

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5
(h1(cup) + 2 * h2(cup)) % 7 = 1
(h1(cup) + 3 * h2(cup)) % 7 = 4

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 13

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5
(h1(cup) + 2 * h2(cup)) % 7 = 1
(h1(cup) + 3 * h2(cup)) % 7 = 4
(h1(cup) + 4 * h2(cup)) % 7 = 0

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 14

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5
(h1(cup) + 2 * h2(cup)) % 7 = 1
(h1(cup) + 3 * h2(cup)) % 7 = 4
(h1(cup) + 4 * h2(cup)) % 7 = 0
(h1(cup) + 5 * h2(cup)) % 7 = 3

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 15

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	cup

h1(apple) = 0

h1(cat) = 2

h1(anvil) = 0
h1(anvil) + 1 * h2(anvil) = 0 + 5 = 5

h1(boy) = 1

h1(bag) = 1
h1(bag) + 1 * h2(bag) = 1 + 3 = 4

h1(dog) = 3

h1(cup) = 2
h1(cup) + 1 * h2(cup) = 5
(h1(cup) + 2 * h2(cup)) % 7 = 1
(h1(cup) + 3 * h2(cup)) % 7 = 4
(h1(cup) + 4 * h2(cup)) % 7 = 0
(h1(cup) + 5 * h2(cup)) % 7 = 3
(h1(cup) + 6 * h2(cup)) % 7 = 6

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 15

0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	cup

We cannot insert “down” because the table is full. Using double hashing, when the size of the hash table is a prime number, detecting that we have overflow is linear in the size of the hash table.

`apple, cat, anvil, boy, bag, dog, cup, down`

Double Hashing

Probe length: 22

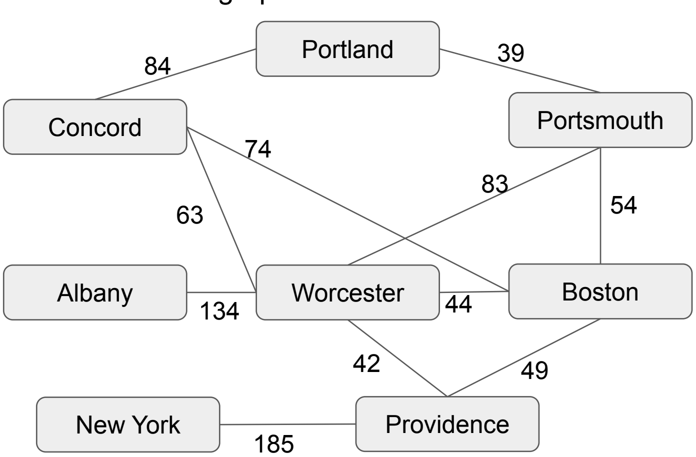
0	apple
1	boy
2	cat
3	dog
4	bag
5	anvil
6	cup

We cannot insert “down” because the table is full. Using double hashing, when the size of the hash table is a prime number, detecting that we have overflow is linear in the size of the hash table.

Total probe length = 15 + 7 (for the probe length of “down”) = **22**

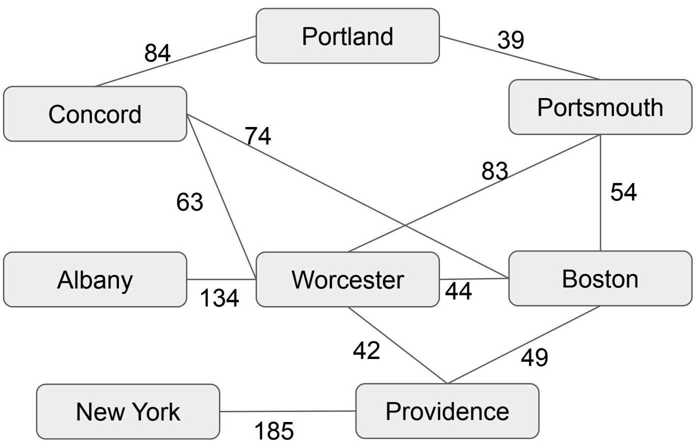
Graph Terminology and Representation

Consider the graph from lecture:



Graph Terminology and Representation

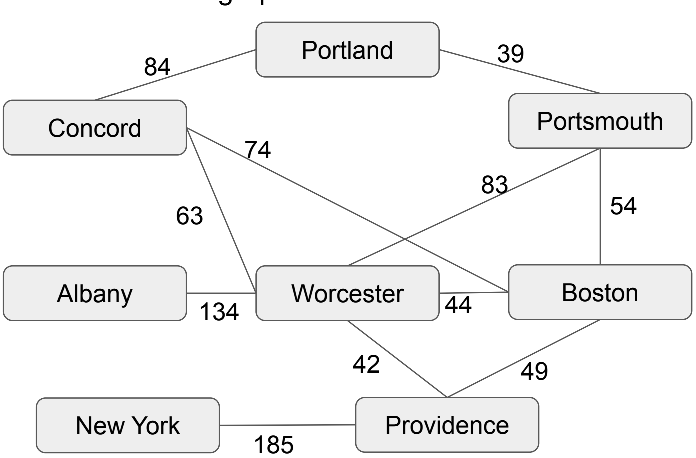
Consider the graph from lecture:



What are Worcester's neighbors in the graph?

Graph Terminology and Representation

Consider the graph from lecture:

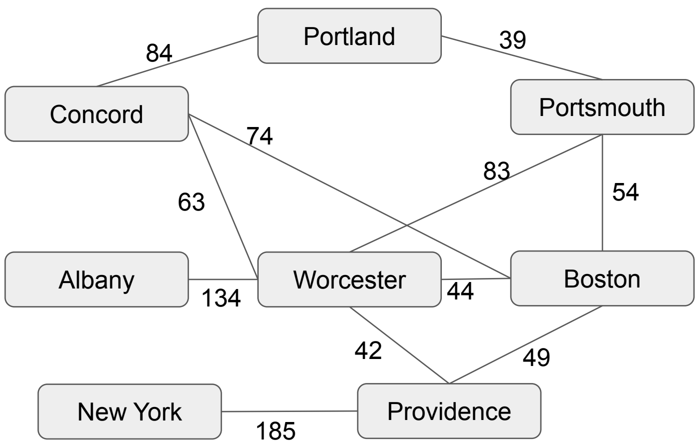


What are Worcester's neighbors in the graph?

Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Graph Terminology and Representation

Consider the graph from lecture:

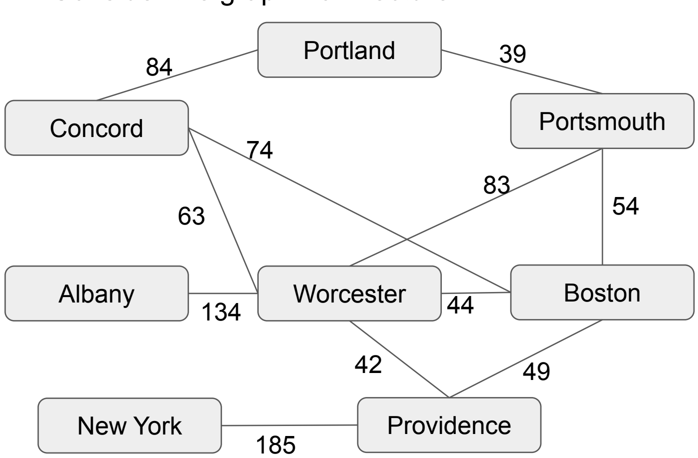


What are Worcester’s neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Is the graph connected? Why or why not?

Graph Terminology and Representation

Consider the graph from lecture:

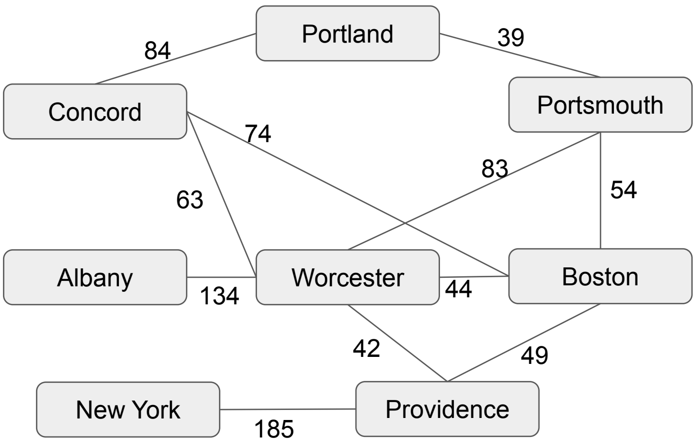


What are Worcester’s neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Is the graph connected? Why or why not?
Yes, because there is a path between every pair of vertices.

Graph Terminology and Representation

Consider the graph from lecture:



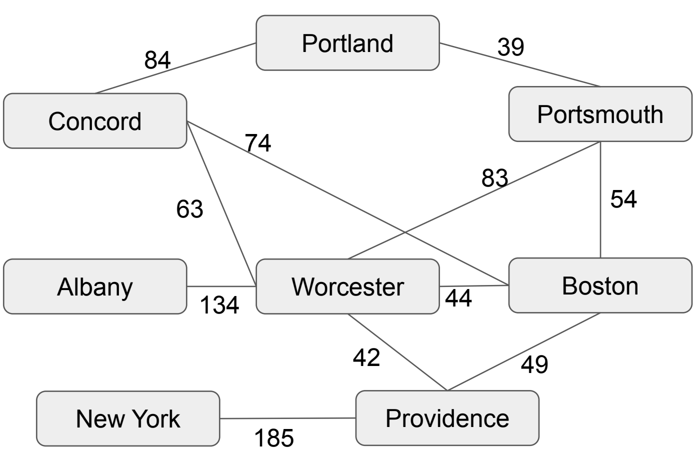
What are Worcester's neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Is the graph connected? Why or why not?
Yes, because there is a path between every pair of vertices.

Is it complete? Why or why not?

Graph Terminology and Representation

Consider the graph from lecture:



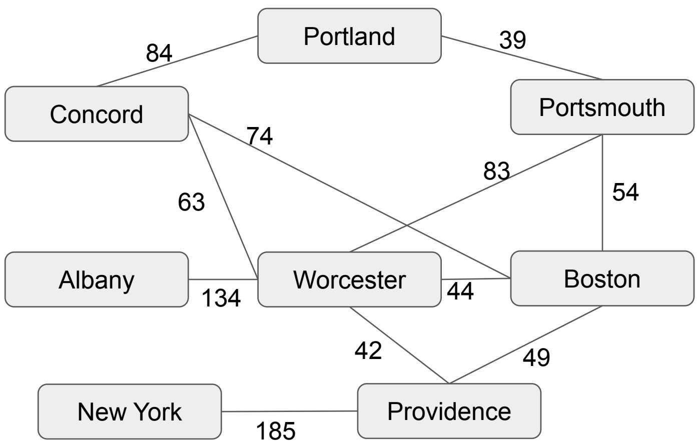
What are Worcester's neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Is the graph connected? Why or why not?
Yes, because there is a path between every pair of vertices.

Is it complete? Why or why not?
It is not, because there is not an edge between every pair of vertices.

Graph Terminology and Representation

Consider the graph from lecture:



What are Worcester's neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

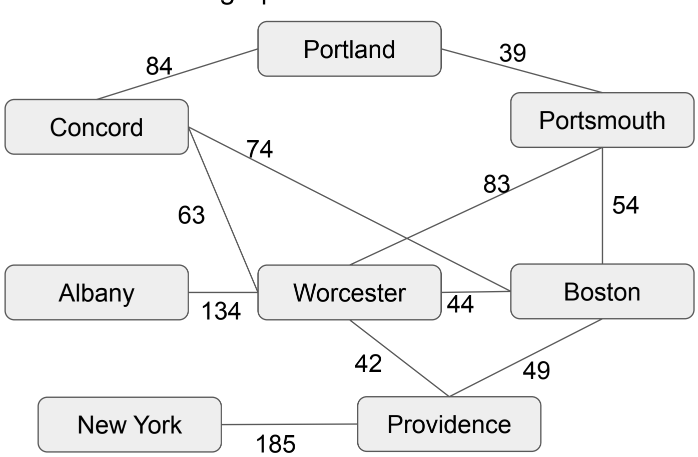
Is the graph connected? Why or why not?
Yes, because there is a path between every pair of vertices.

Is it complete? Why or why not?
It is not, because there is not an edge between every pair of vertices.

Is it acyclic? If not, give an example cycle.

Graph Terminology and Representation

Consider the graph from lecture:



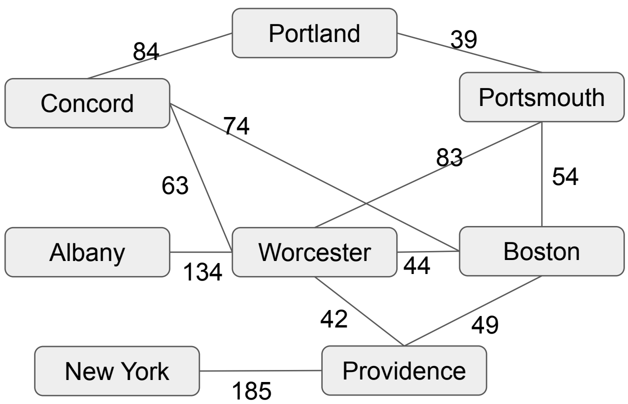
What are Worcester's neighbors in the graph?
Albany, Boston, Concord, Portsmouth, and Providence, because it is connected to each of them by a single edge.

Is the graph connected? Why or why not?
Yes, because there is a path between every pair of vertices.

Is it complete? Why or why not?
It is not, because there is not an edge between every pair of vertices.

Is it acyclic? If not, give an example cycle.
Not acyclic; Worcester > Boston > Providence > Worcester

Graph Terminology and Representation

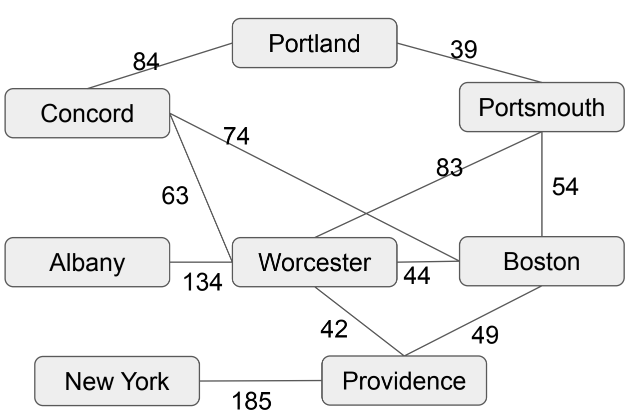


If we used an adjacency matrix to represent this graph, what would it look like? Assume that the vertices are numbered alphabetically, starting from zero:

0. Albany
1. Boston
2. Concord
3. New York
4. Portland
5. Portsmouth
6. Providence
7. Worcester

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

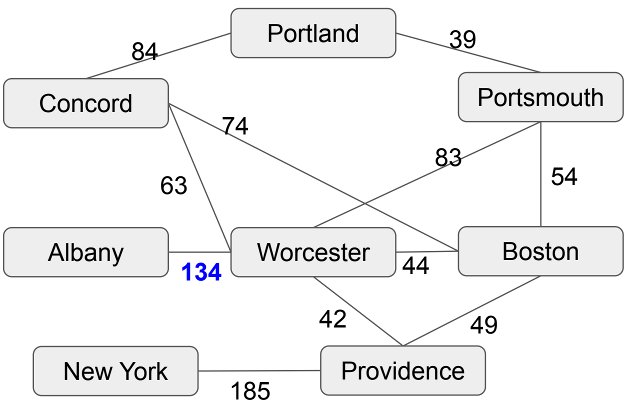
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

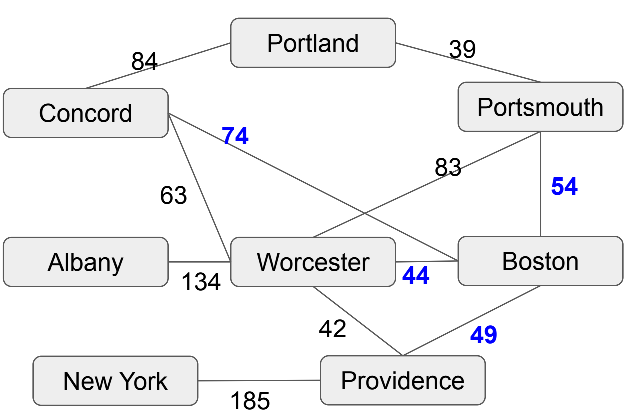
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1								
2								
3								
4								
5								
6								
7	134							

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

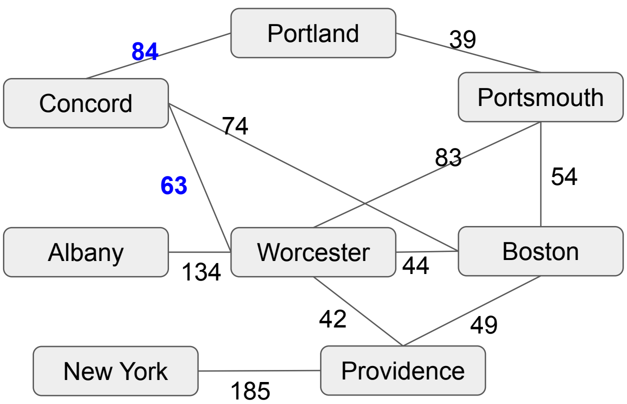
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74						
3								
4								
5		54						
6		49						
7	134	44						

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

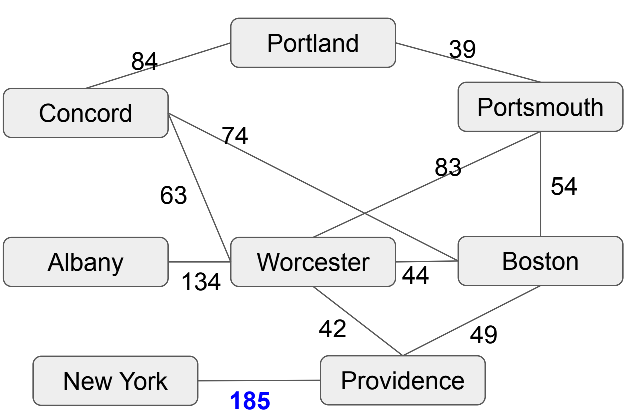
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74			84			63
3								
4			84					
5		54						
6		49						
7	134	44	63					

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

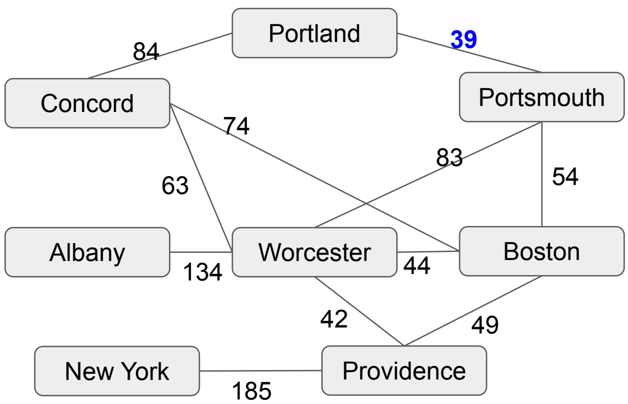
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74			84			63
3							185	
4			84					
5		54						
6		49		185				
7	134	44	63					

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

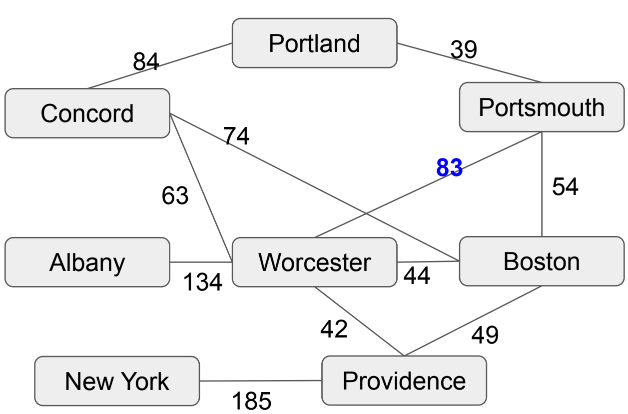
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74			84			63
3							185	
4			84			39		
5		54			39			
6		49		185				
7	134	44	63					

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

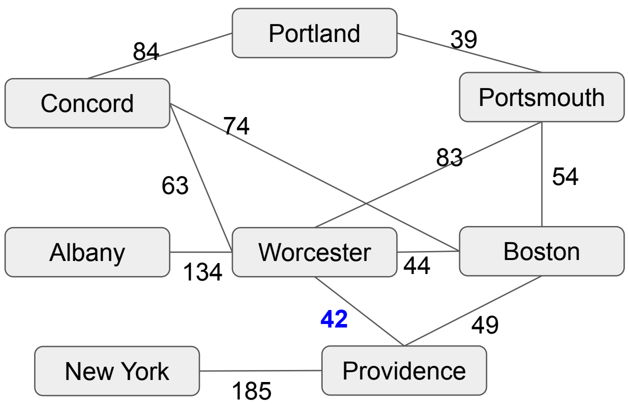
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74			84			63
3							185	
4			84			39		
5		54			39			83
6		49		185				
7	134	44	63			83		

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

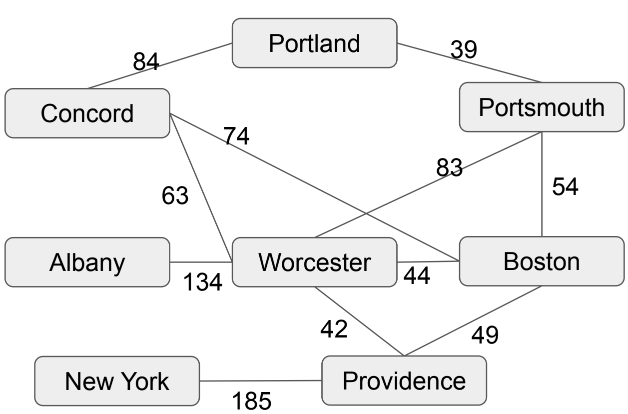
Graph Terminology and Representation



	0	1	2	3	4	5	6	7
0								134
1			74			54	49	44
2		74			84			63
3							185	
4			84			39		
5		54			39			83
6		49		185				42
7	134	44	63			83	42	

0: Albany, 1: Boston, 2: Concord, 3: NY, 4: Portland, 5: Portsmouth, 6: Providence, 7: Worcester

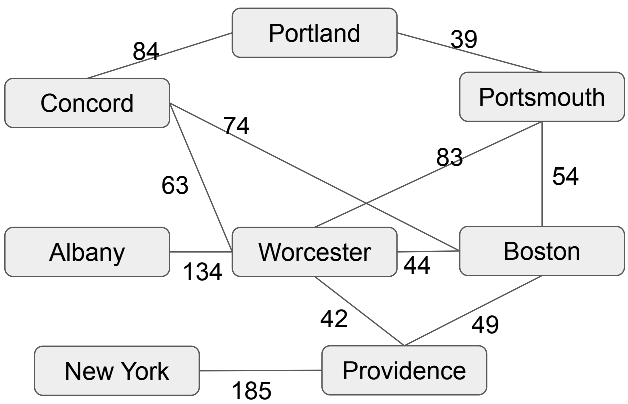
Graph Terminology and Representation



	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0								**134**
1			**74**			**54**	**49**	**44**
2		**74**			**84**			**63**
3							**185**	
4			**84**			**39**		
5		**54**			**39**			**83**
6		**49**		**185**				**42**
7	**134**	**44**	**63**			**83**	**42**	

Graph Traversals

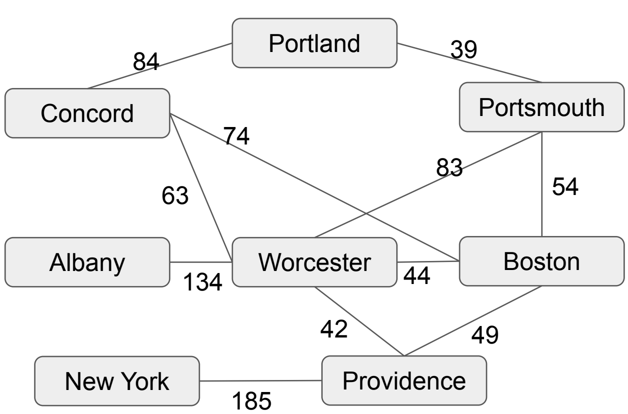
In what order would the cities be visited if we performed a **depth-first traversal** from Boston? Draw the resulting spanning tree.



Graph Traversals

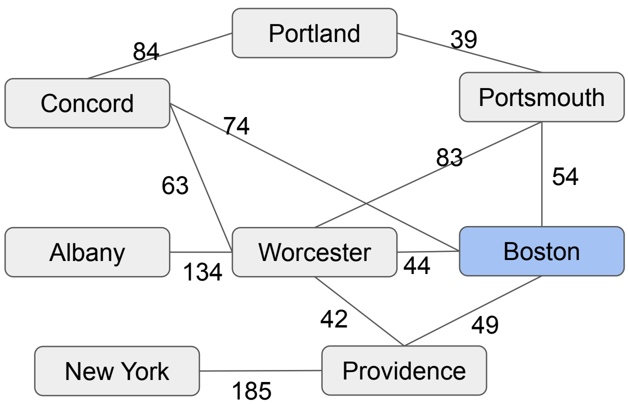
In what order would the cities be visited if we performed a **depth-first traversal** from Boston? Draw the resulting spanning tree.

Remember that we assume adjacency lists sort edges in order of increasing weight!



Boston,

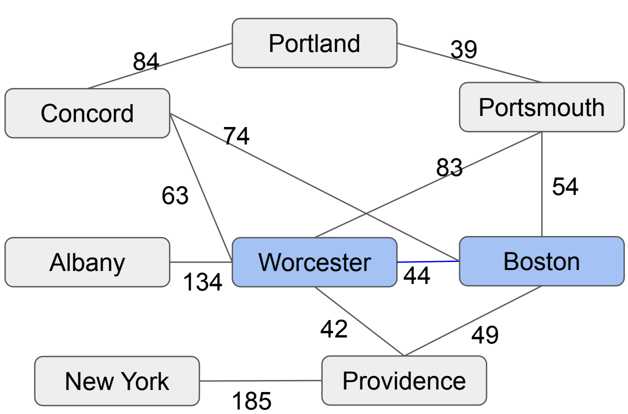
Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

Boston, Worcester,

Graph Traversals

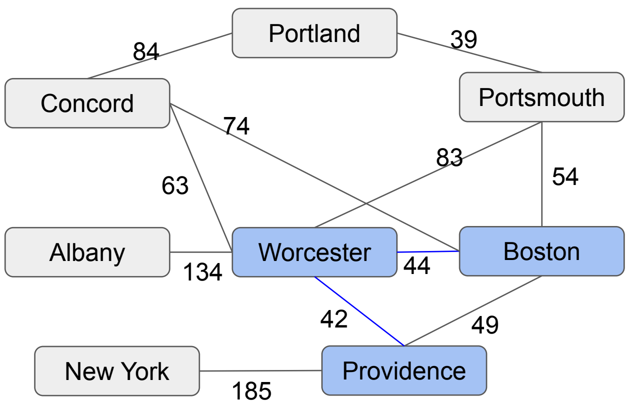


dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): visit Worcester, set its parent reference to Boston, make a recursive call on closest neighbor, Providence

Boston, Worcester, Providence,

Graph Traversals



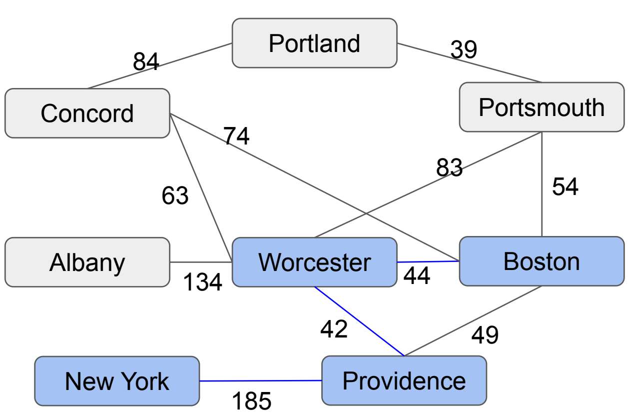
dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): visit Worcester, set its parent reference to Boston, make a recursive call on closest neighbor, Providence

dfTrav(Providence, Worcester): visit Providence, set its parent reference to Worcester, recurse to nearest *unvisited* neighbor, New York

Boston, Worcester, Providence, NY,

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

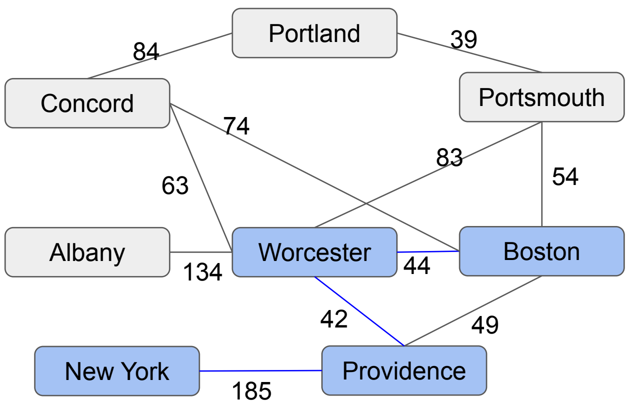
dfTrav(Worcester, Boston): visit Worcester, set its parent reference to Boston, make a recursive call on closest neighbor, Providence

dfTrav(Providence, Worcester): visit Providence, set its parent reference to Worcester, recurse to nearest *unvisited* neighbor, New York

dfTrav(New York, Providence): visit NY, set its parent reference to Providence. No unvisited neighbors, so **return.**

Boston, Worcester, Providence, NY,

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

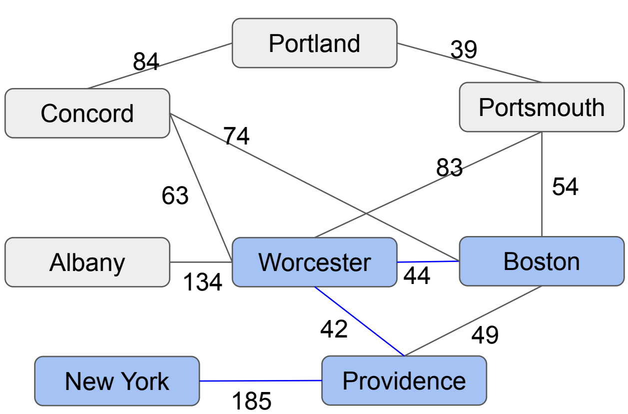
dfTrav(Worcester, Boston): visit Worcester, set its parent reference to Boston, make a recursive call on closest neighbor, Providence

dfTrav(Providence, Worcester): visit Providence, set its parent reference to Worcester, recurse to nearest *unvisited* neighbor, New York

- Providence has no unvisited neighbors, so **return**

Boston, Worcester, Providence, NY,

Graph Traversals

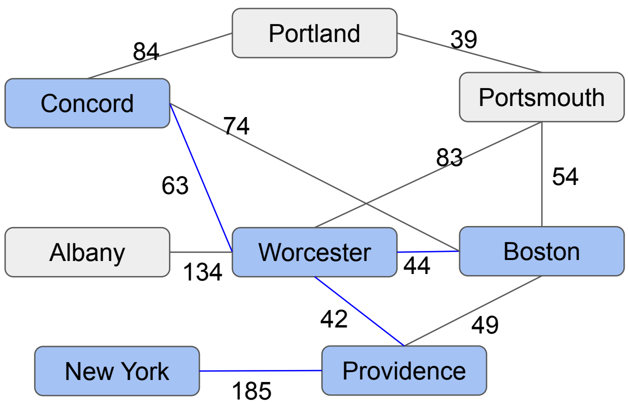


dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

Boston, Worcester, Providence, NY, Concord

Graph Traversals



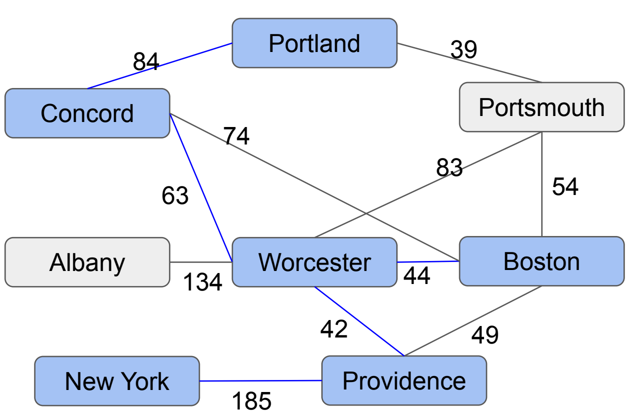
dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

dfTrav(Concord, Worcester): visit Concord, set its parent reference to Worcester, and recurse on nearest *unvisited* neighbor, **Portland**

Boston, Worcester, Providence, NY, Concord, Portland

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

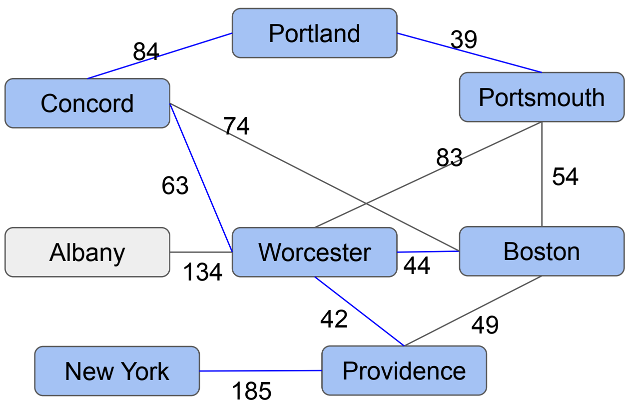
dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

dfTrav(Concord, Worcester): visit Concord, set its parent reference to Worcester, and recurse on nearest *unvisited* neighbor, **Portland**

dfTrav(Portland, Concord): visit Portland, set its parent reference to Concord, and recurse on Portsmouth

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth,

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

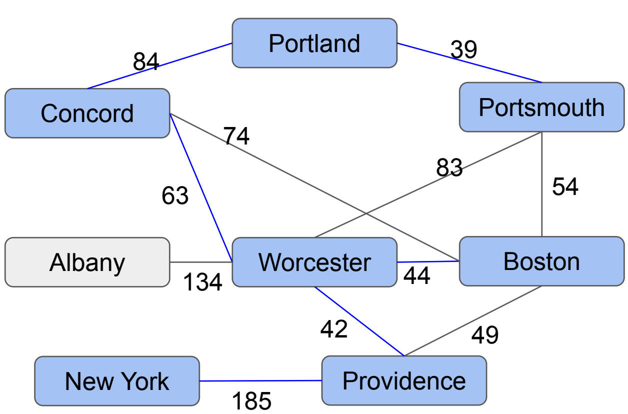
dfTrav(Concord, Worcester): visit Concord, set its parent reference to Worcester, and recurse on nearest *unvisited* neighbor, **Portland**

dfTrav(Portland, Concord): visit Portland, set its parent reference to Concord, and recurse on Portsmouth

dfTrav(Portsmouth, Portland): visit Portsmouth, set its parent reference to Portland. No unvisited neighbors, so **return**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth,

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

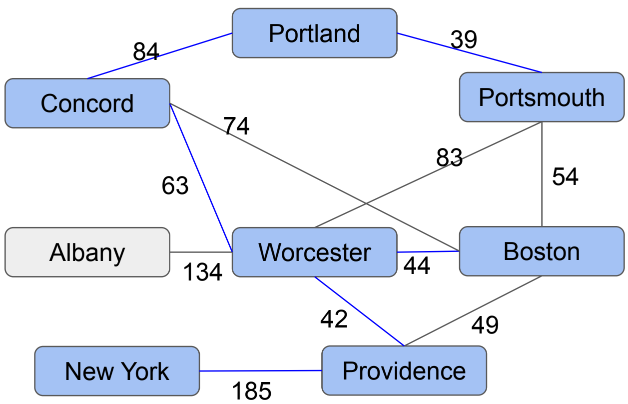
dfTrav(Concord, Worcester): visit Concord, set its parent reference to Worcester, and recurse on nearest *unvisited* neighbor, **Portland**

dfTrav(Portland, Concord): visit Portland, set its parent reference to Concord, and recurse on Portsmouth

- Portland has no unvisited neighbors, so **return**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth,

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

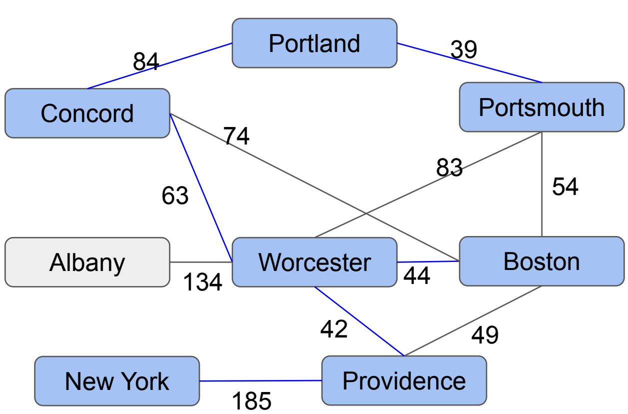
dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Concord**

dfTrav(Concord, Worcester): visit Concord, set its parent reference to Worcester, and recurse on nearest *unvisited* neighbor, **Portland**

- Concord has no unvisited neighbors, so **return**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth,

Graph Traversals

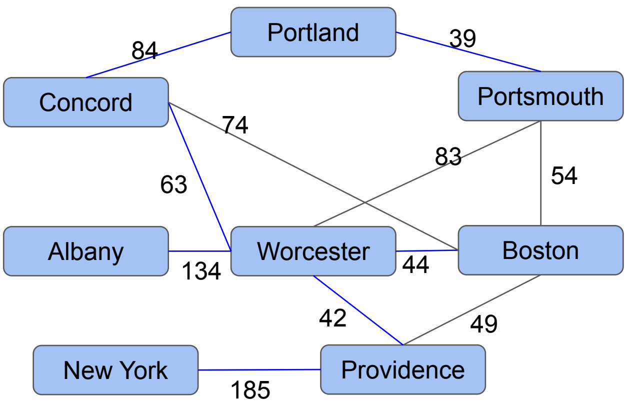


dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Albany**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

Graph Traversals



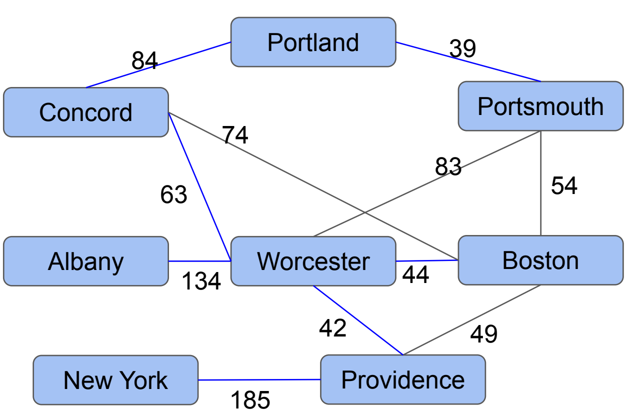
dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Albany**

dfTrav(Albany, Worcester): visit Albany, set its parent reference to Worcester. No unvisited neighbors, so **return**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

Graph Traversals



dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

dfTrav(Worcester, Boston): Worcester still has unvisited neighbors, so we recurse on the next closest one: **Albany**

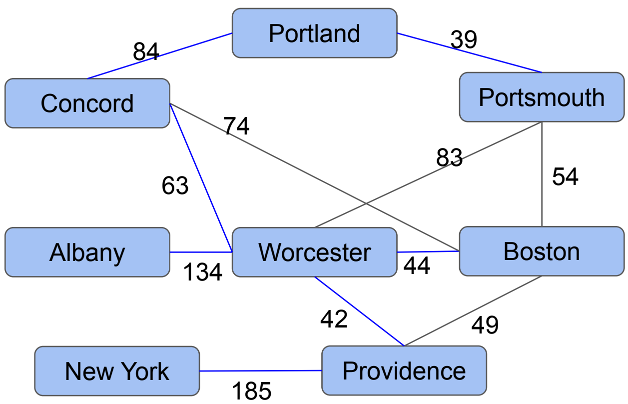
- Worcester has no unvisited neighbors, so **return**

Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

Graph Traversals

dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

- Boston has no unvisited neighbors, so we return, and complete the DF traversal!

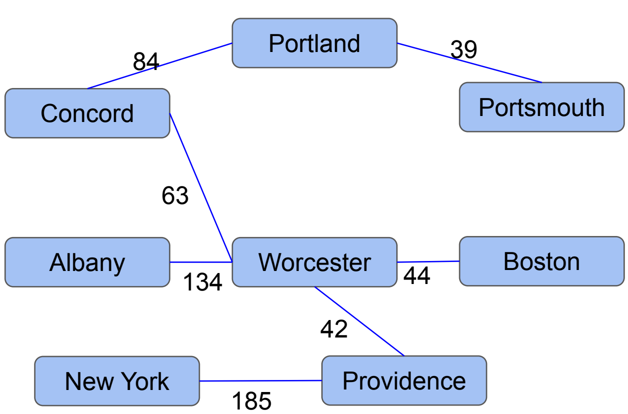


Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

Graph Traversals

dfTrav(Boston, null): visit Boston, set its parent reference to null, and make a recursive call on closest neighbor, Worcester

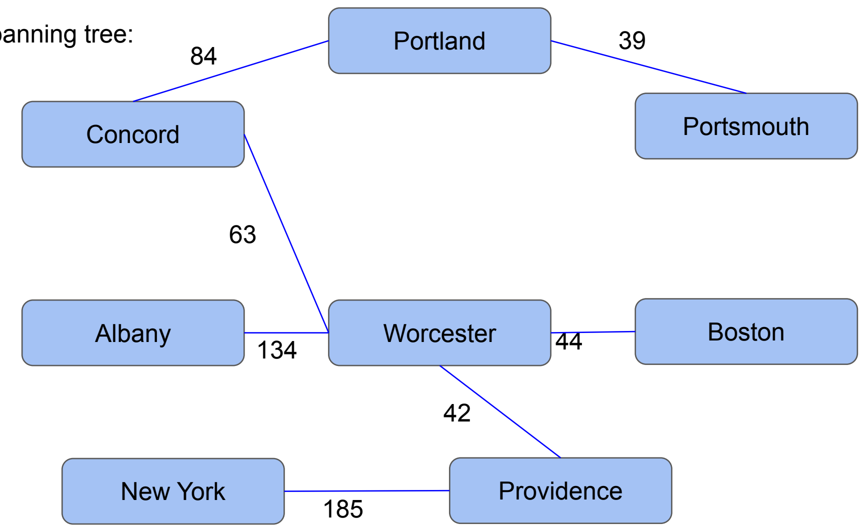
- Boston has no unvisited neighbors, so we return, and complete the DF traversal!



Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

Graph Traversals

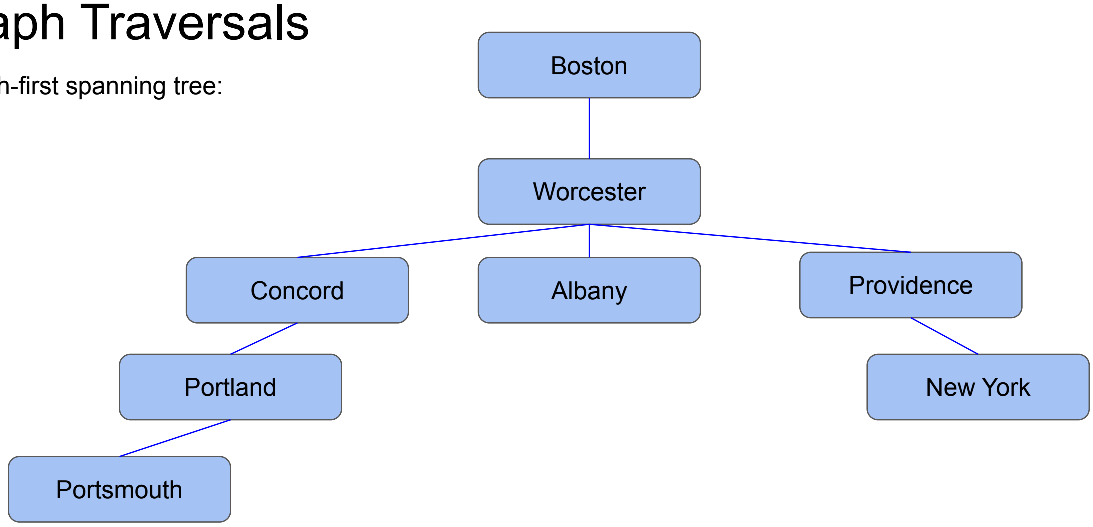
Our depth-first spanning tree:



Boston, Worcester, Providence, NY, Concord, Portland, Portsmouth, Albany

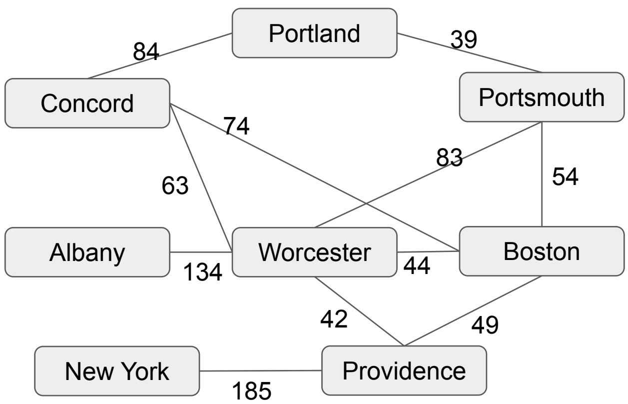
Graph Traversals

Our depth-first spanning tree:



Graph Traversals

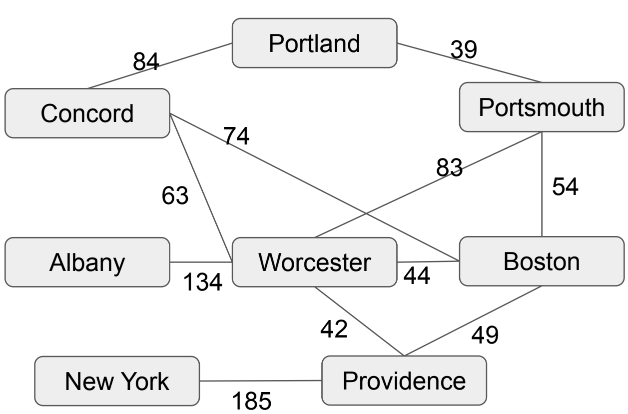
In what order would the cities be visited if we performed a **breadth-first traversal** from Boston? Draw the resulting spanning tree.



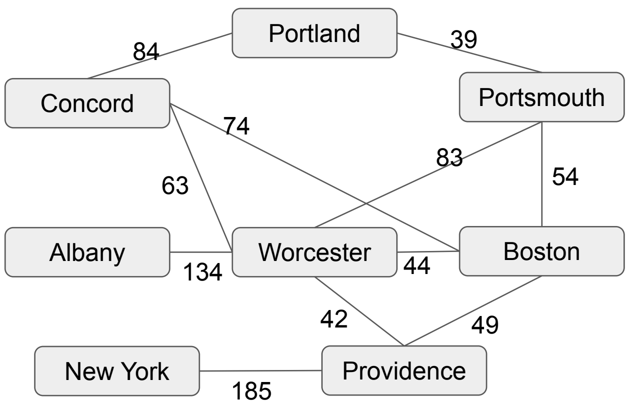
Graph Traversals

In what order would the cities be visited if we performed a **breadth-first traversal** from Boston? Draw the resulting spanning tree.

Remember, with breadth-first traversal, we mark a node as “encountered” *before* it’s put into the queue, and visit it after it’s dequeued.



Graph Traversals

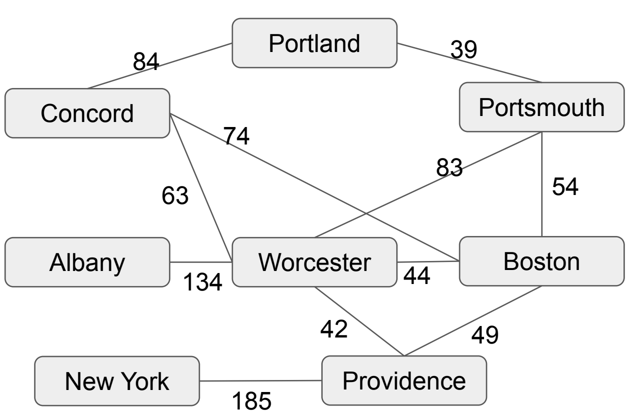


In what order would the cities be visited if we performed a **breadth-first traversal** from Boston? Draw the resulting spanning tree.

Remember, with breadth-first traversal, we mark a node as "encountered" *before* it's put into the queue, and visit it after it's dequeued.

We set parent references when we "encounter" a node.

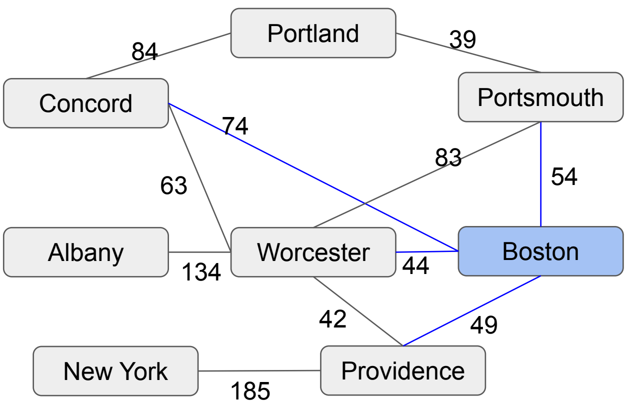
Graph Traversals



remove	**insert**	**queue contents**
	Boston	Boston

Boston,

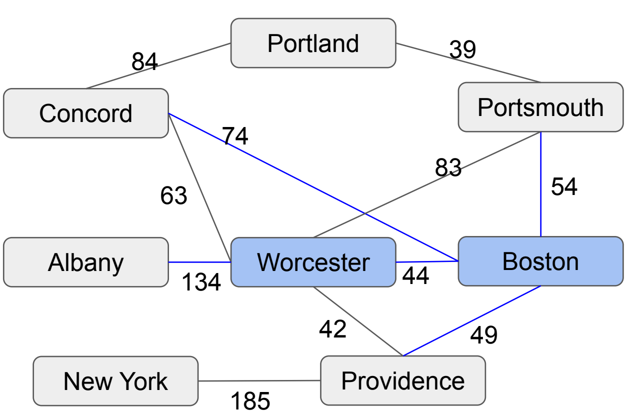
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord

Boston, Worcester,

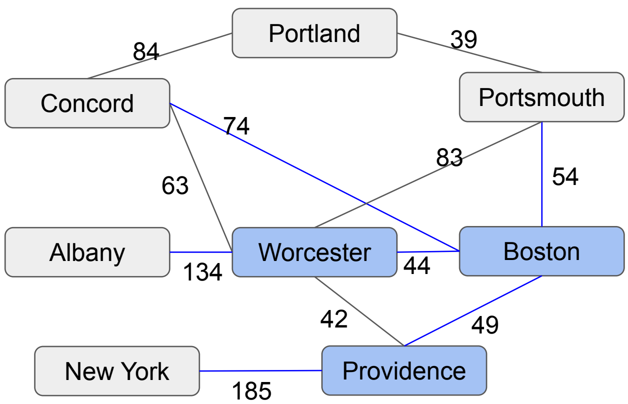
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany

Boston, Worcester, Providence

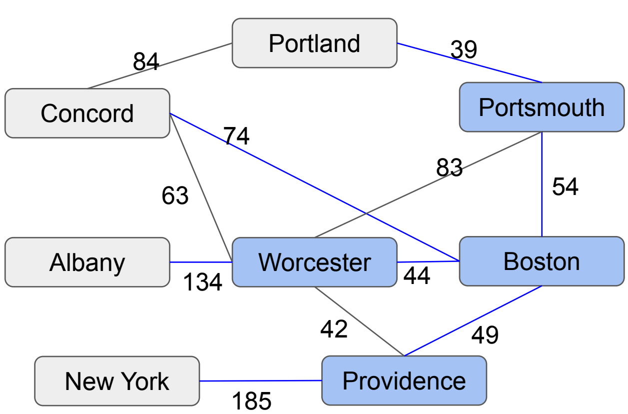
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY

Boston, Worcester, Providence, Portsmouth

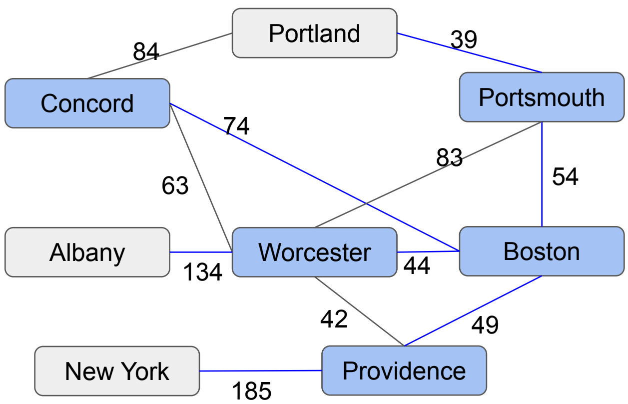
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland

Boston, Worcester, Providence, Portsmouth, Concord,

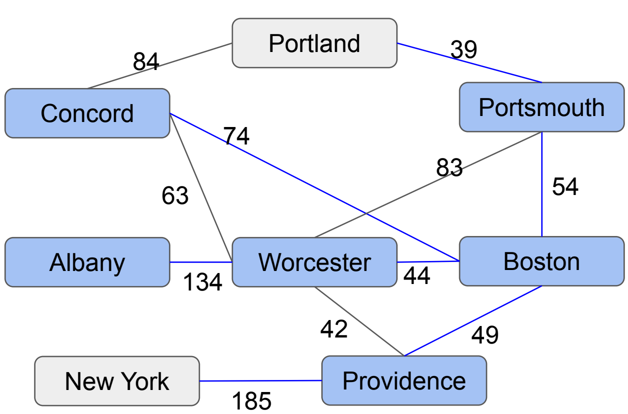
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland
Concord	*none*	Albany, NY, Portland

Boston, Worcester, Providence, Portsmouth, Concord, Albany,

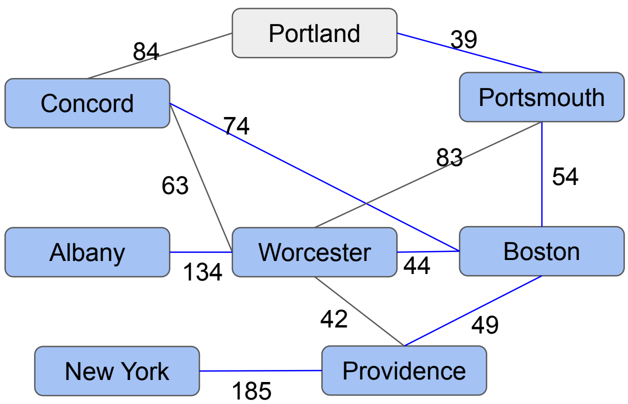
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland
Concord	*none*	Albany, NY, Portland
Albany	*none*	NY, Portland

Boston, Worcester, Providence, Portsmouth, Concord, Albany, NY,

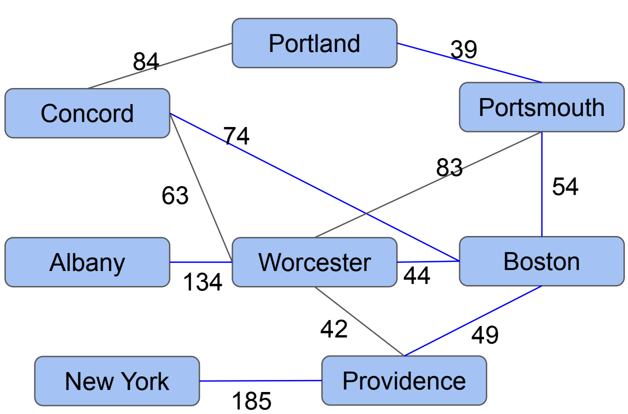
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland
Concord	*none*	Albany, NY, Portland
Albany	*none*	NY, Portland
NY	*none*	Portland

Boston, Worcester, Providence, Portsmouth, Concord, Albany, NY, Portland

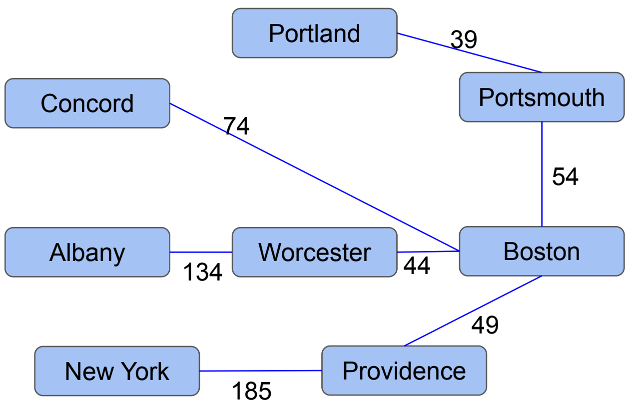
Graph Traversals



remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland
Concord	*none*	Albany, NY, Portland
Albany	*none*	NY, Portland
NY	*none*	Portland
Portland	*none*	*empty*

Boston, Worcester, Providence, Portsmouth, Concord, Albany, NY, Portland

Graph Traversals

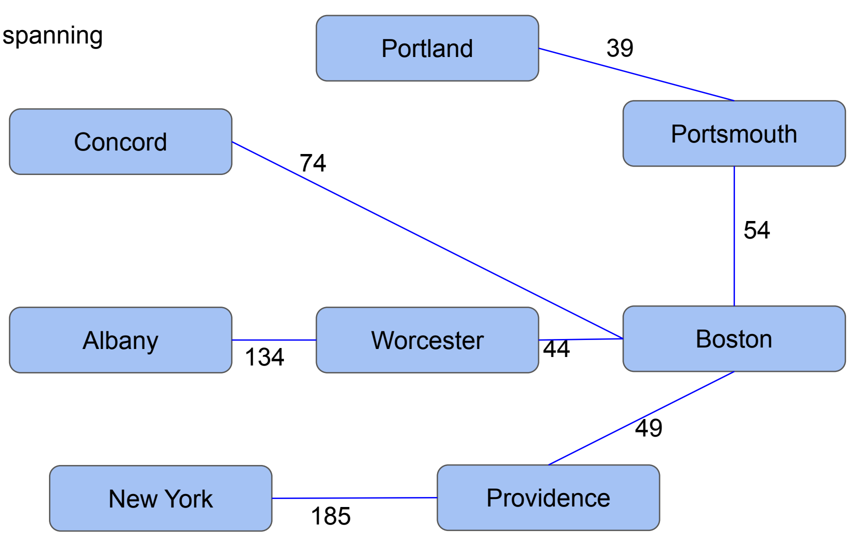


remove	insert	queue contents
	Boston	Boston
Boston	Worcester, Providence, Portsmouth, Concord	Worcester, Providence, Portsmouth, Concord
Worcester	Albany	Providence, Portsmouth, Concord, Albany
Providence	NY	Portsmouth, Concord, Albany, NY
Portsmouth	Portland	Concord, Albany, NY, Portland
Concord	*none*	Albany, NY, Portland
Albany	*none*	NY, Portland
NY	*none*	Portland
Portland	*none*	*empty*

Boston, Worcester, Providence, Portsmouth, Concord, Albany, NY, Portland

Graph Traversals

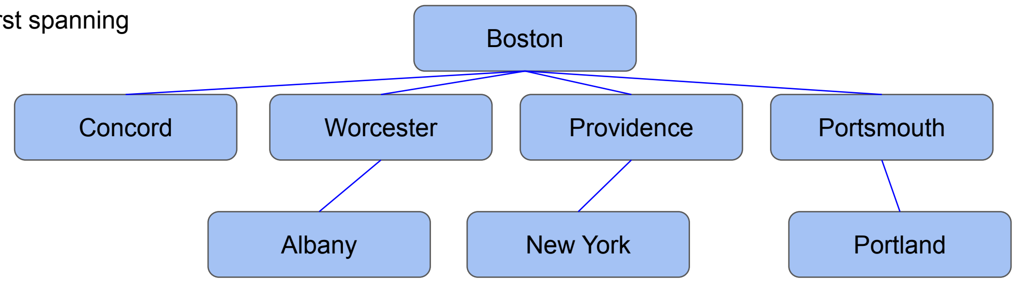
Our breadth-first spanning tree:



Boston, Worcester, Providence, Portsmouth, Concord, Albany, NY, Portland

Graph Traversals

Our breadth-first spanning tree:



End of section.

Questions?

Lecture 13

CSCI E-22

Will Begin Shortly