ECS32B

Introduction to Data Structures

Heaps

Graphs

Lecture 25

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22			
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

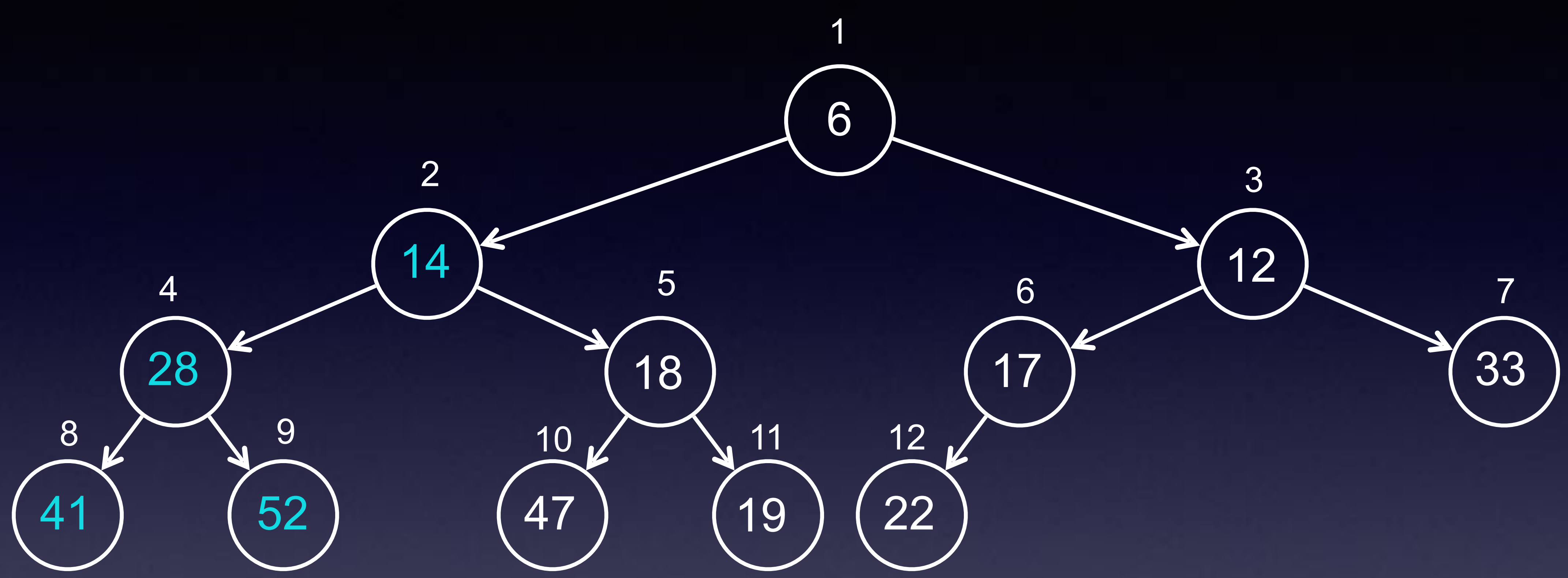
we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

If the node we're looking at is 28, then k = 4
The left child is 41 at index = 2*4 = 8
The right child is 52 at index 2*4+1 = 9
The parent is 14 at index = int(4/2) = 2

Implementing a heap



If the node we're looking at is 28, then k = 4
The left child is 41 at index = 2*4 = 8
The right child is 52 at index 2*4+1 = 9
The parent is 14 at index = int(4/2) = 2

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22	3		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22	3		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3
Less than or equal to its parent? (index = int(13/2) = 6, value = 17)

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22	3		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3

Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Implementing a heap

0	6	14	12	28	18	3	33	41	52	47	19	22	17		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Implementing a heap

0	6	14	12	28	18	3	33	41	52	47	19	22	17		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3

Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Less than or equal to its parent? (index = 3, value = 12)

Implementing a heap

0	6	14	3	28	18	12	33	41	52	47	19	22	17		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap
Less than or equal to its parent? (index = 3, value = 12) Yes, swap

Implementing a heap

0	6	14	3	28	18	12	33	41	52	47	19	22	17		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap
Less than or equal to its parent? (index = 3, value = 12) Yes, swap
Less than or equal to its parent? (index = 1, value = 6)

Implementing a heap

0	3	14	6	28	18	12	33	41	52	47	19	22	17		
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

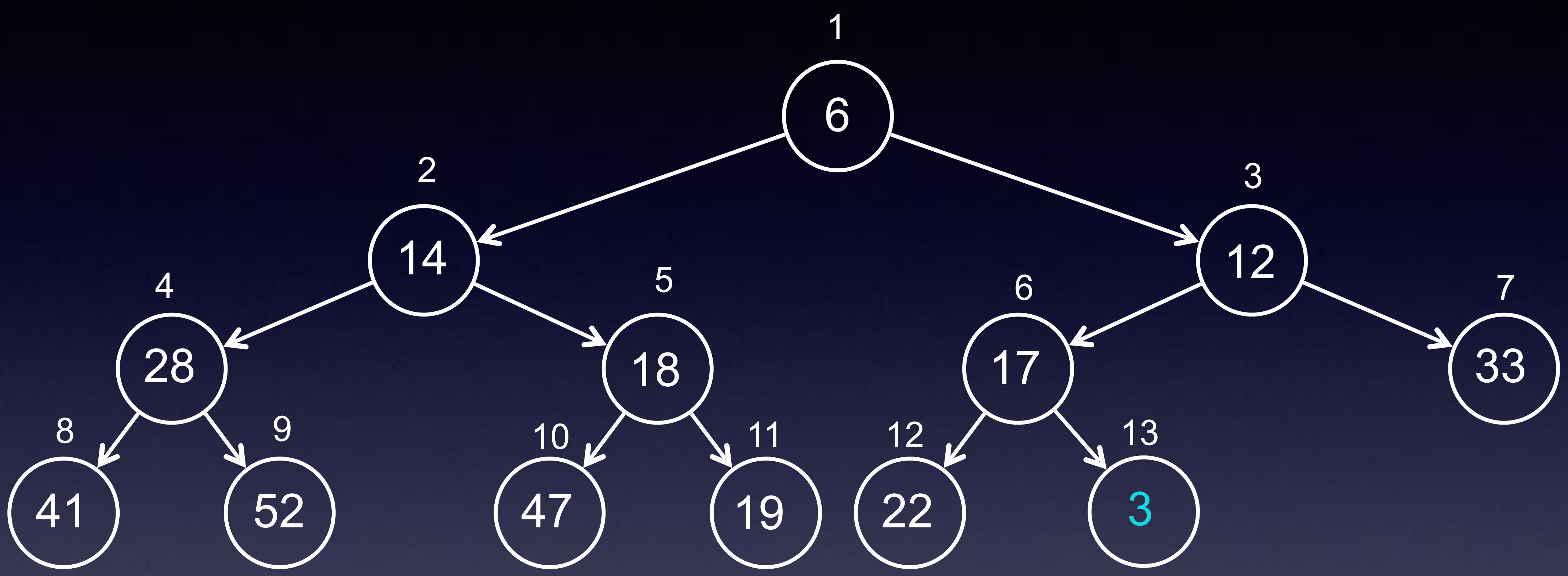
we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

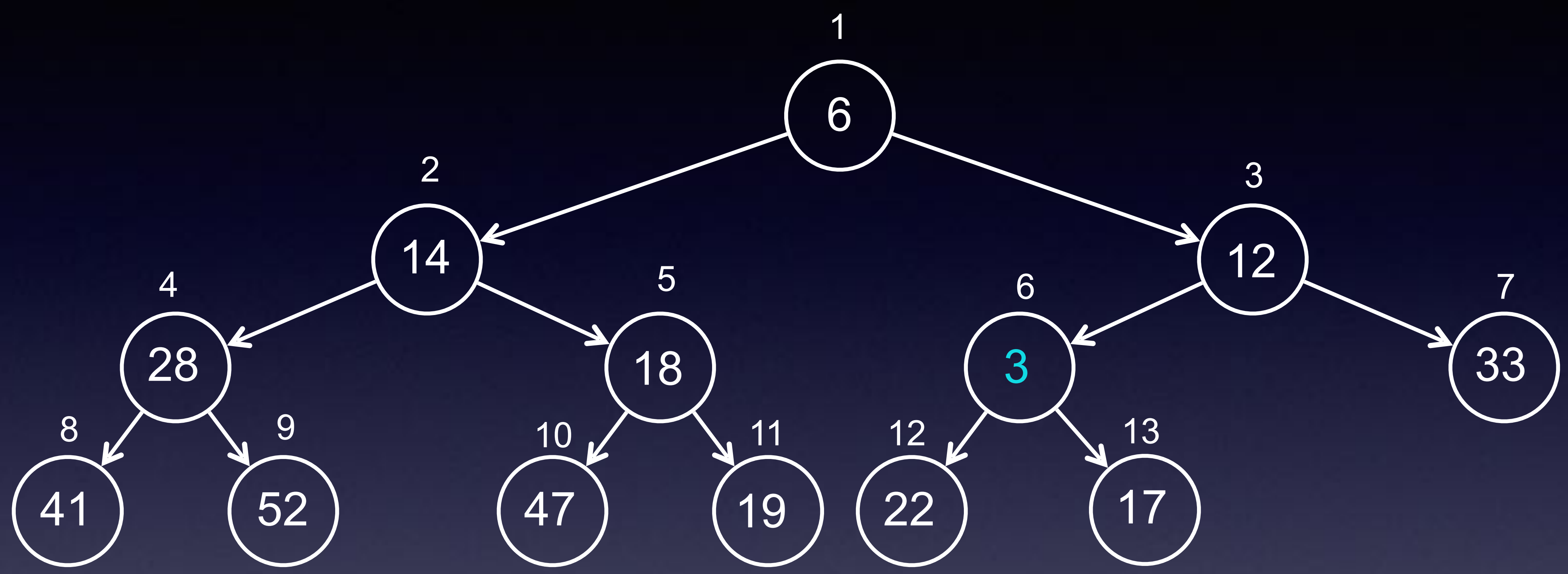
Let's insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap
Less than or equal to its parent? (index = 3, value = 12) Yes, swap
Less than or equal to its parent? (index = 1, value = 6) Yes, swap
Less than or equal to its parent? Wait, we're at top – **done!**

Implementing a heap



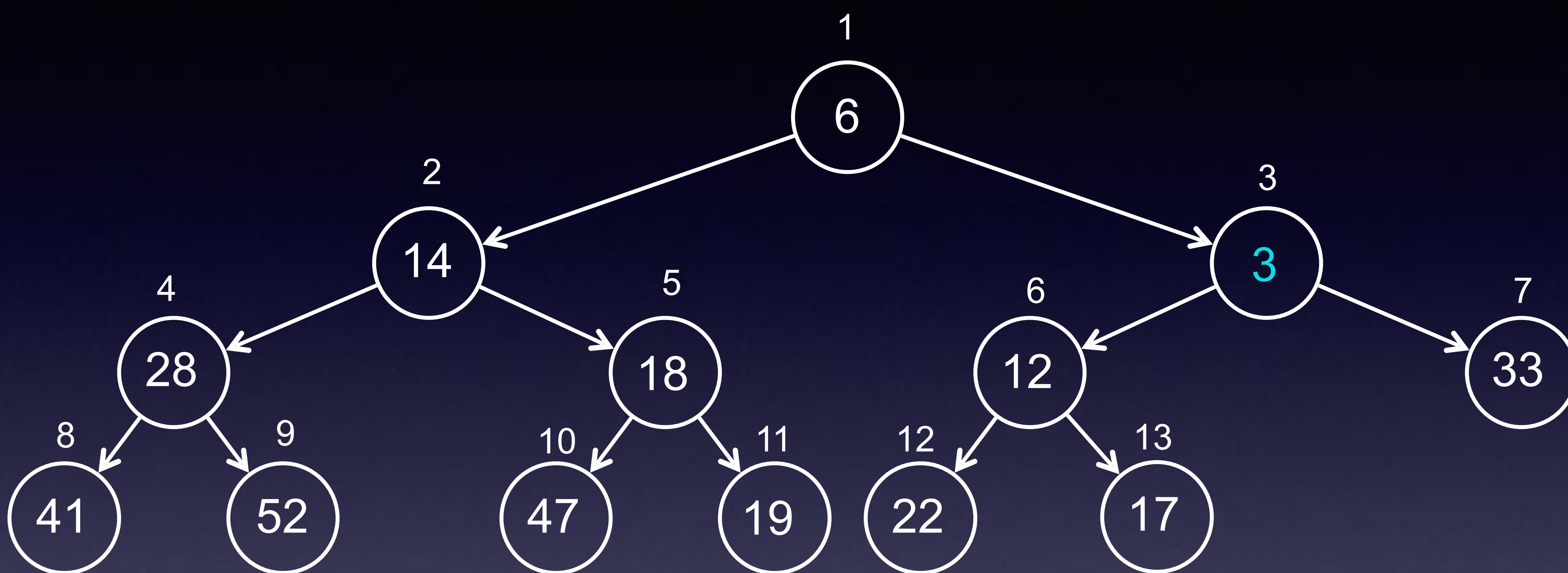
Let's insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Implementing a heap



Let’s insert 3
Less than or equal to its parent? (index = 6, value = 17) Yes, swap
Less than or equal to its parent? (index = 3, value = 12) Yes, swap

Implementing a heap



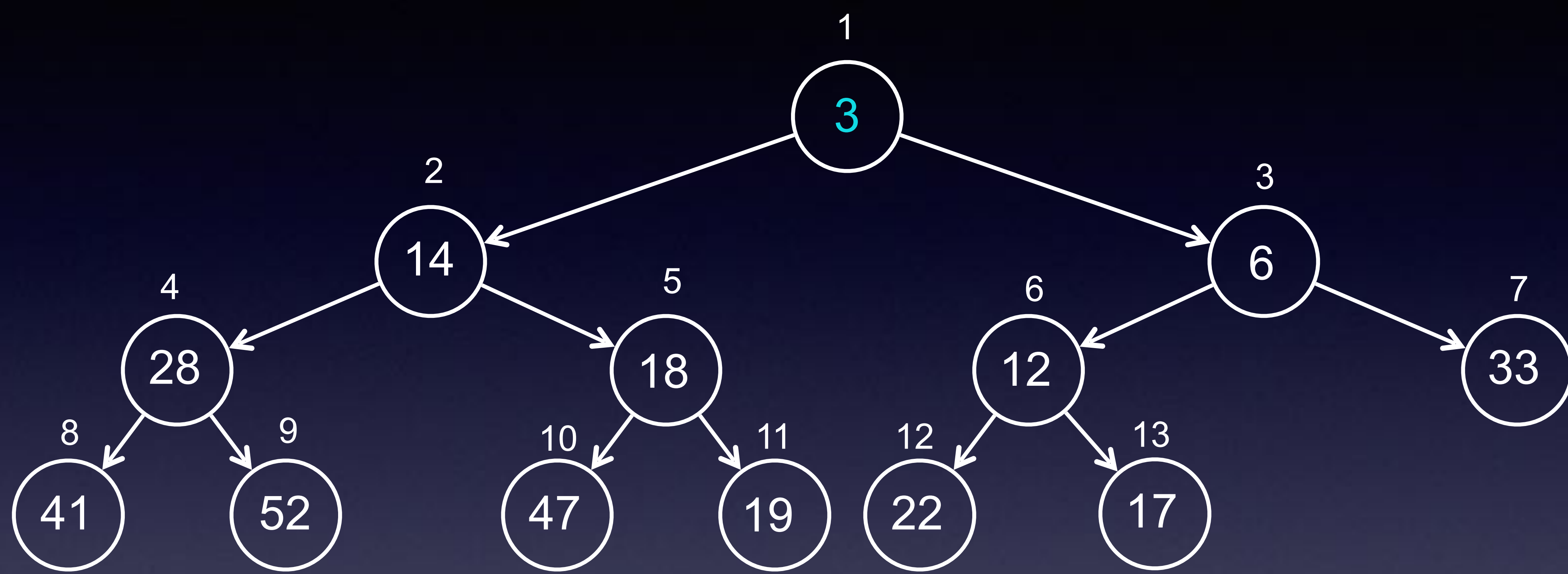
Let's insert 3

Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Less than or equal to its parent? (index = 3, value = 12) Yes, swap

Less than or equal to its parent? (index = 1, value = 6) Yes, swap

Implementing a heap



Let's insert 3

Less than or equal to its parent? (index = 6, value = 17) Yes, swap

Less than or equal to its parent? (index = 3, value = 12) Yes, swap

Less than or equal to its parent? (index = 1, value = 6) Yes, swap

Less than or equal to its parent? Wait, we're at top of the heap – **done!**

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6

Implementing a heap

0	22	14	12	28	18	17	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item, position 12, into position 1

Implementing a heap

0	22	14	12	28	18	17	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item into position 1
Is 22 smaller than smallest child? if yes then swap with smallest child

Implementing a heap

0	22	14	12	28	18	17	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item into position 1
Look at the children (pos 2 and 3), swap with smallest child? Yes

Implementing a heap

0	12	14	22	28	18	17	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item into position 1
Look at the children (pos 2 and 3), swap with smallest child? Yes
Look at the children (pos 6 and 7), swap with smallest child? Yes

Implementing a heap

0	12	14	17	28	18	22	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item into position 1
Look at the children (pos 2 and 3), swap with smallest child? Yes
Look at the children (pos 6 and 7), swap with smallest child? Yes

Implementing a heap

0	12	14	17	28	18	22	33	41	52	47	19				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Given a node at index k,

we can find the left child of k at index: 2k

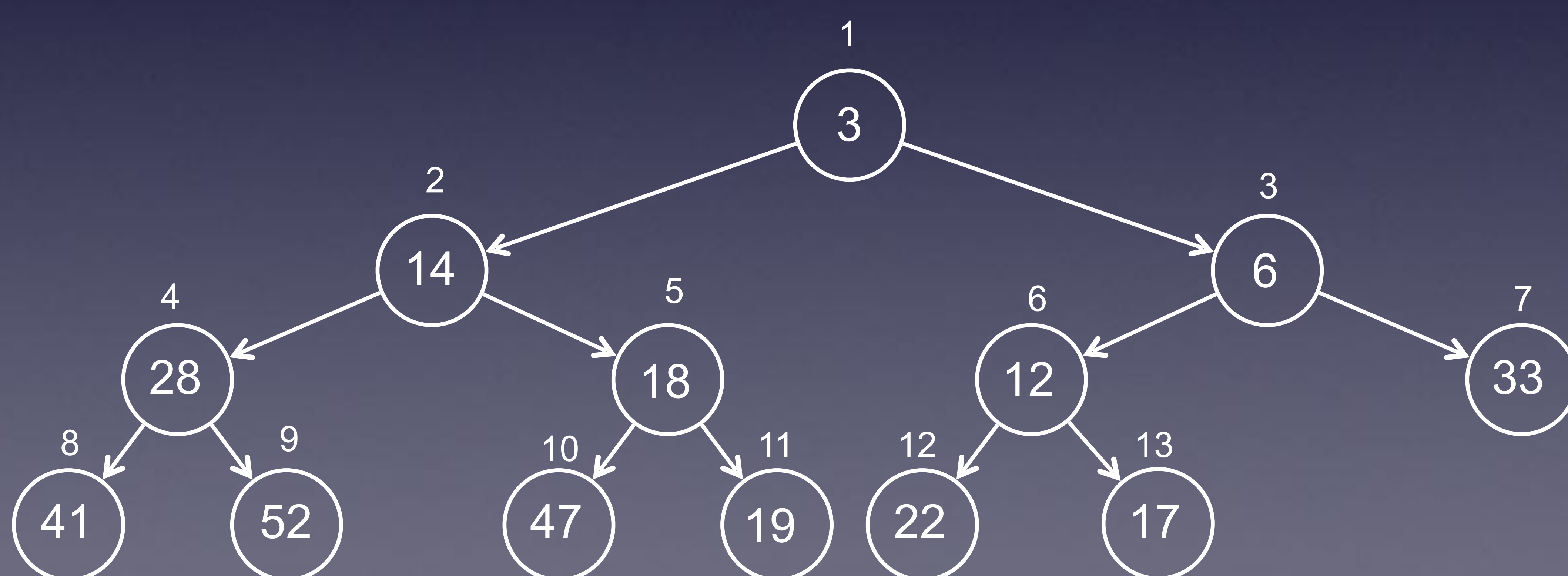
we can find the right child of k at index: 2k + 1

we can find the parent of k at index: int(k / 2) or k // 2

Let's delete and return 6
Save the item at position 1 and put the last item into position 1
Look at the children (pos 2 and 3), swap with smallest child? Yes
Look at the children (pos 6 and 7), swap with smallest child? Yes
Look at the children (pos 12 and 13). No children, so return 6.

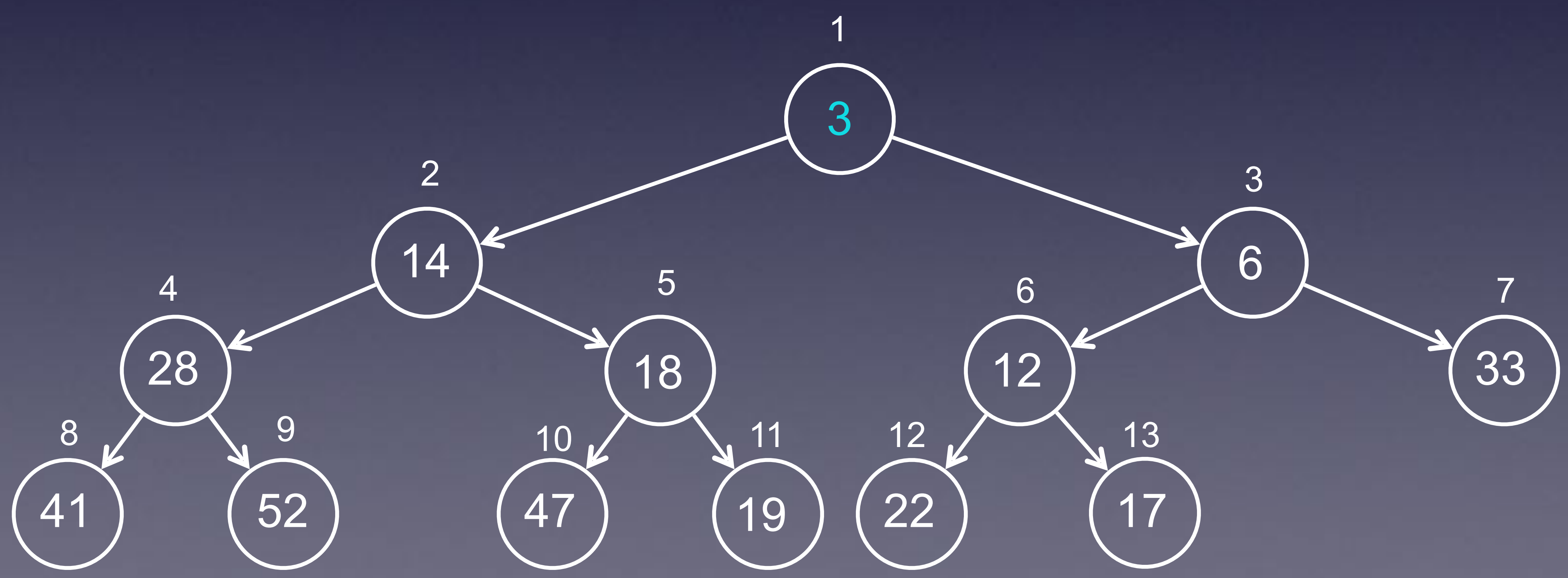
Priority queues

The heap data structure provides the basis for the priority queue abstract data type. We assume that the value shown in the heap represents the priority of a job (smaller numbers have higher priority, like number of pages in a print request), then as values are added, the smaller, higher priority values float toward the top, and lower priority values fall to the bottom.



Priority queues

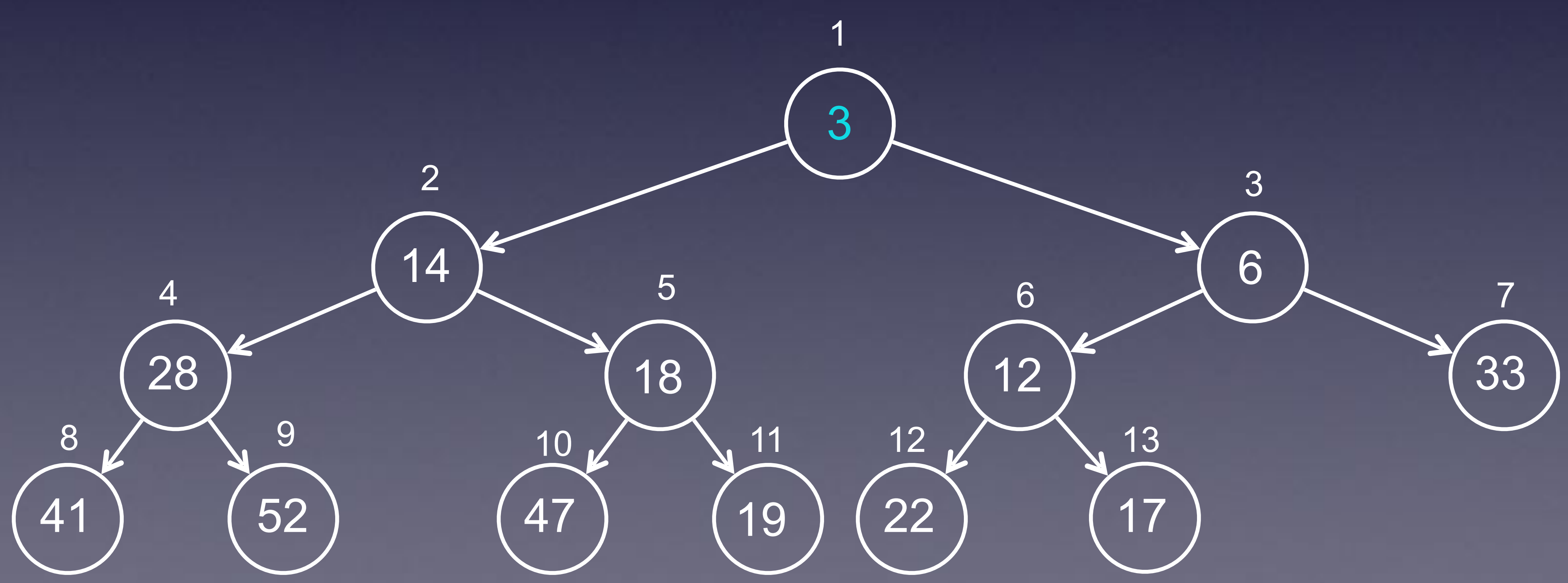
When we want to select the **highest priority job** in the queue, we **just grab the item at the top of the heap** (e.g. 3). We then move the last item in the heap to the top and perform percDown as described previously.



Priority queue analysis

Removing the minimum value (i.e., highest priority item) from the priority queue requires percDown, which takes O(log n) time.

Adding a new item to the priority queue requires percUp, which also takes O(log n) time.



Priority Queue ADT

```
class PriorityQueue:
    def __init__(self):
        self.heapArray = [(0,0)]
        self.currentSize = 0
    def add(self,k):
        self.heapArray.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
    def delMin(self):
        retval = self.heapArray[1][1]
        self.heapArray[1] = self.heapArray[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapArray.pop()
        self.percDown(1)
        return retval
    def isEmpty(self):
        if self.currentSize == 0:
            return True
        else:
            return False
```

The books implementation of a priority queue uses a tuple to combine a priority with the actual item being prioritized

Heapsort

We can also use a heap as the foundation for a very efficient sorting algorithm called heapsort.

Heapsort consists of two phases:

Heapify: build a heap using the elements to be sorted

Sort: Use the heap to sort the data

This can all be done in place in the array that holds the heap, but it’s easier to see if we draw the trees instead of the array.

Once you see how it works with trees, make sure you understand how it works in place in the array.

Heapsort

Here’s the heapify component:

for each item in the sequence to be sorted

add the item to the next available position in the complete binary tree

restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

Heapsort

Here’s the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

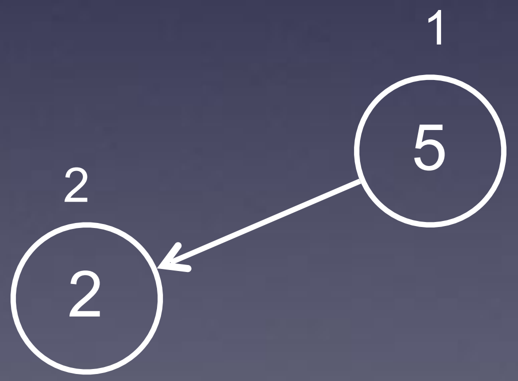
Say we want to sort the sequence 5 2 1 4 3:

Heapsort

Here’s the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

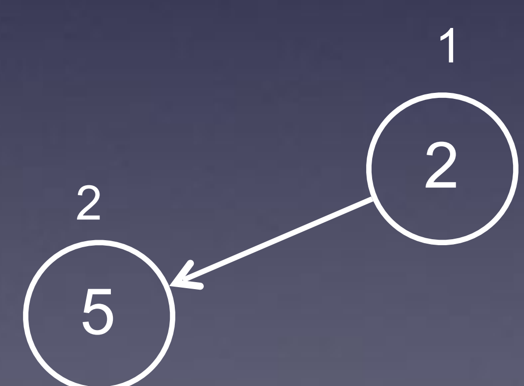


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

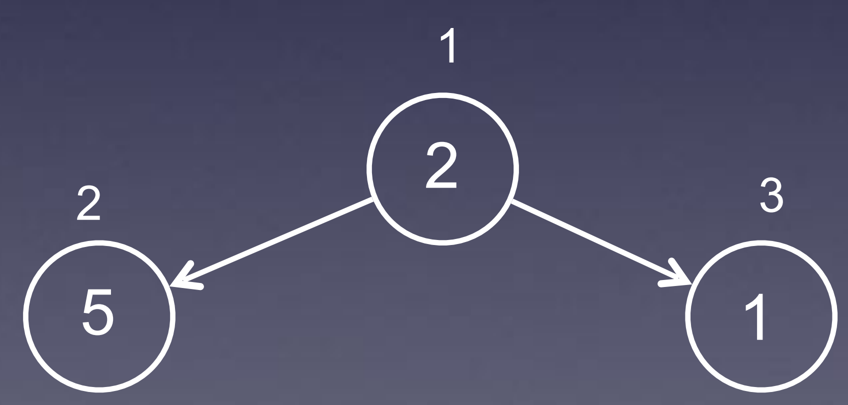


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
add the item to the next available position in the
complete binary tree
restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

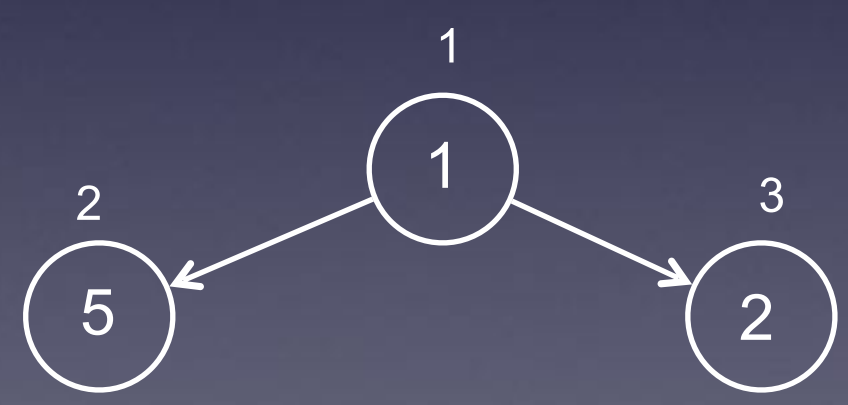


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
add the item to the next available position in the complete binary tree
restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

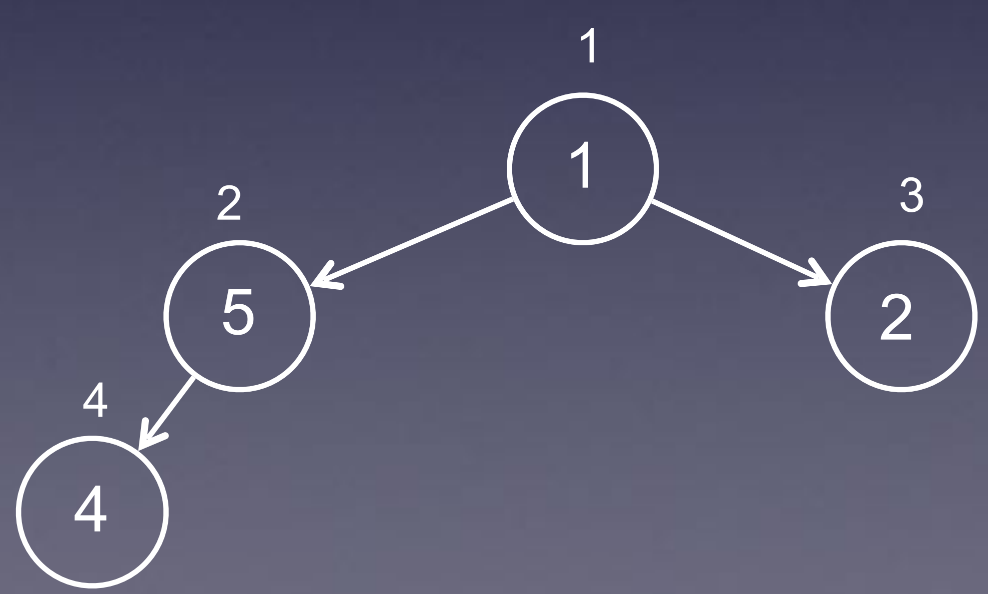


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

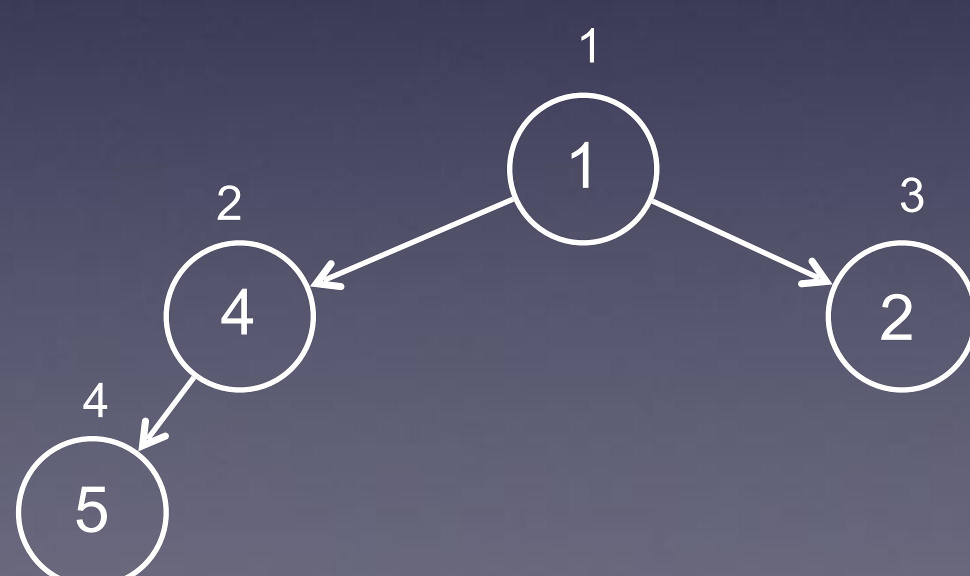


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

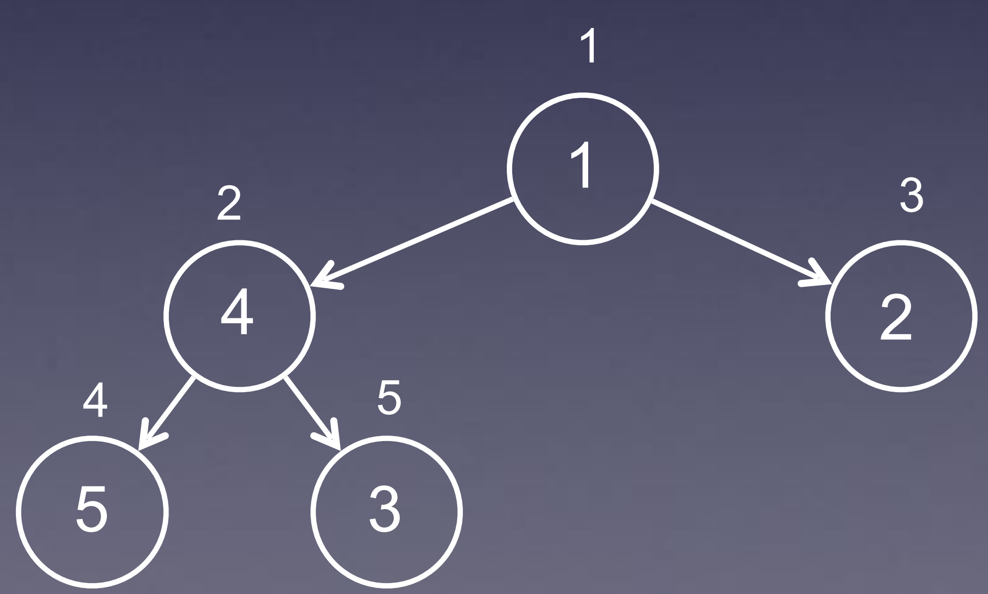


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

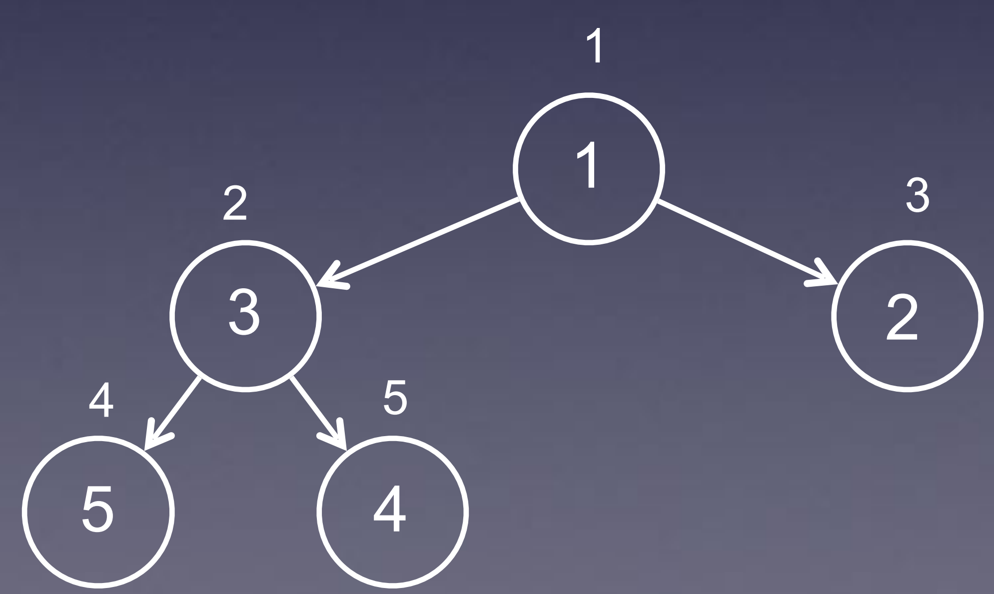


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap order property (using percUp)

Say we want to sort the sequence 5 2 1 4 3:

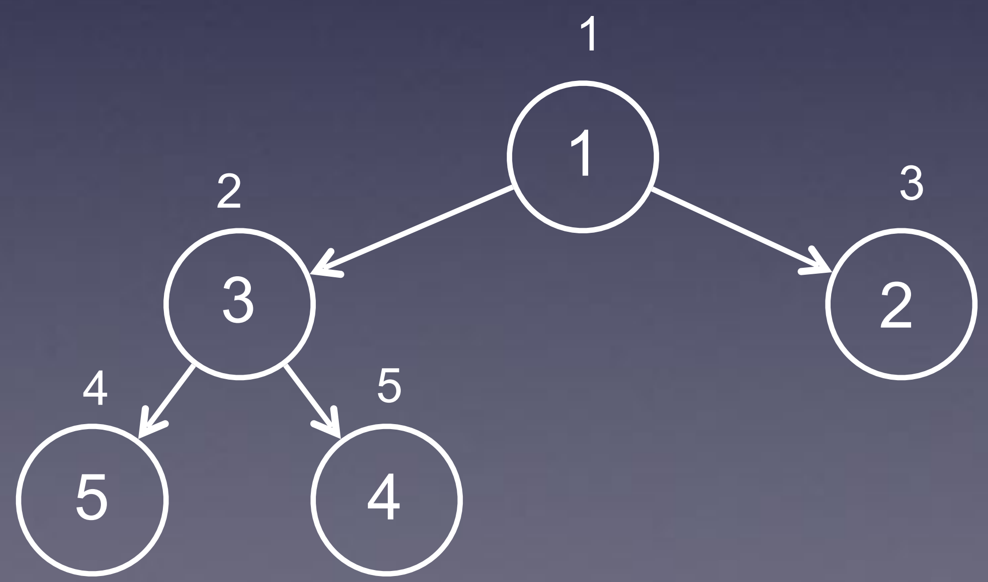


The sequence is heapified

Heapsort

Now for the sorting:

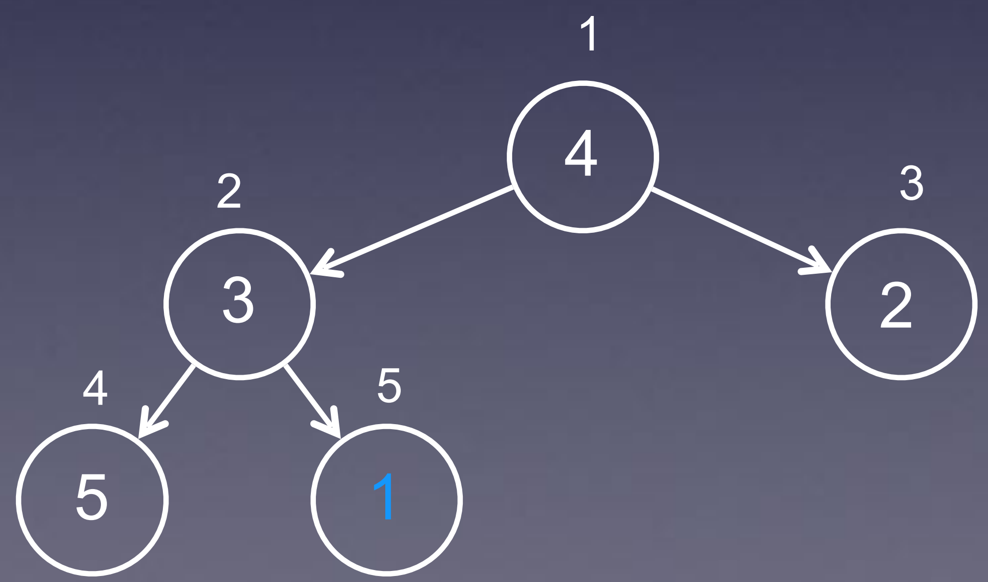
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)



Heapsort

Now for the sorting:

while the heap is not empty
remove the first item from the heap by swapping it
with the last item in the heap
reduce the size of the heap by one
restore the heap order property (using percDown)

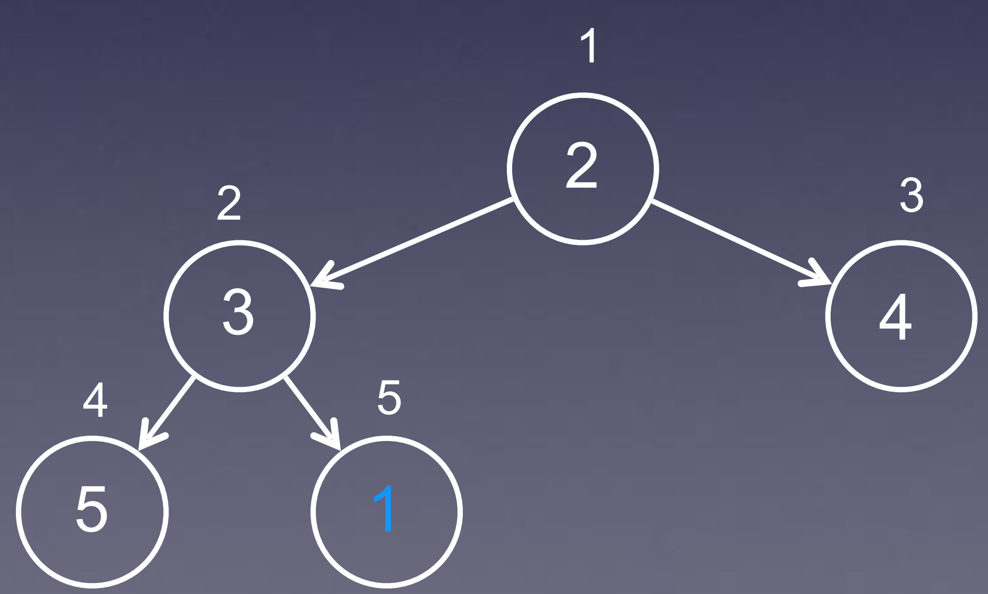


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)

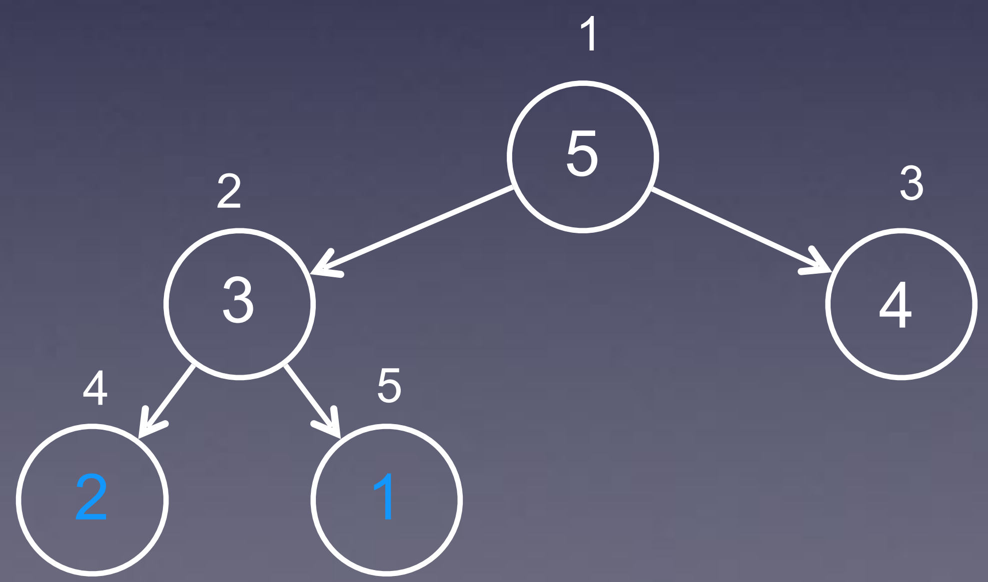


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)

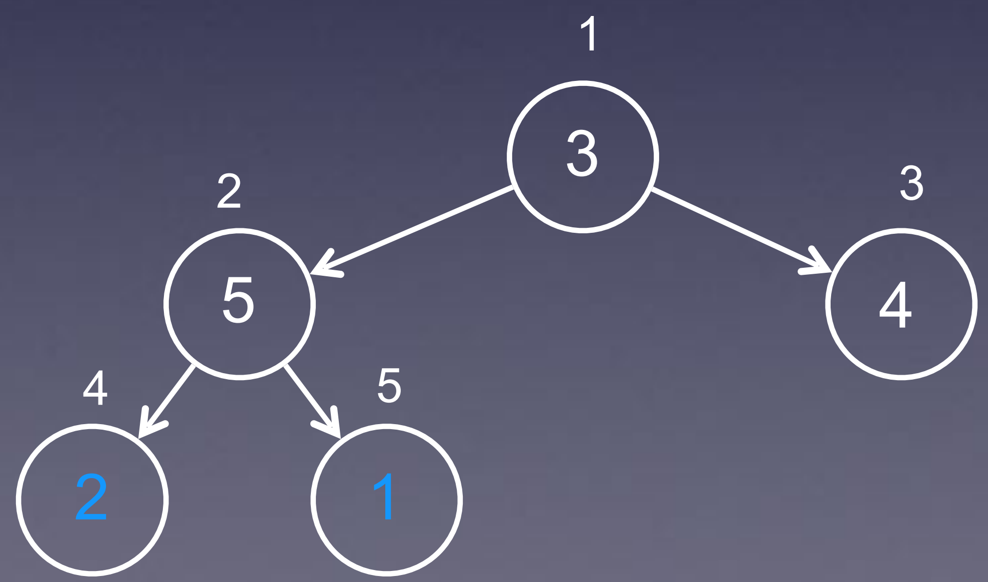


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)

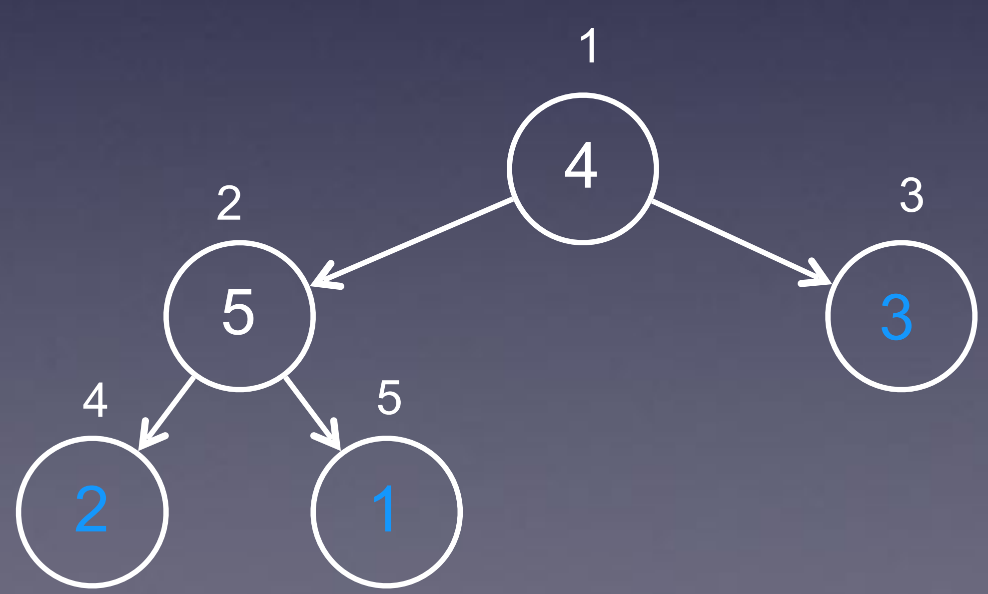


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
  remove the first item from the heap by swapping it
    with the last item in the heap
  reduce the size of the heap by one
  restore the heap order property (using percDown)

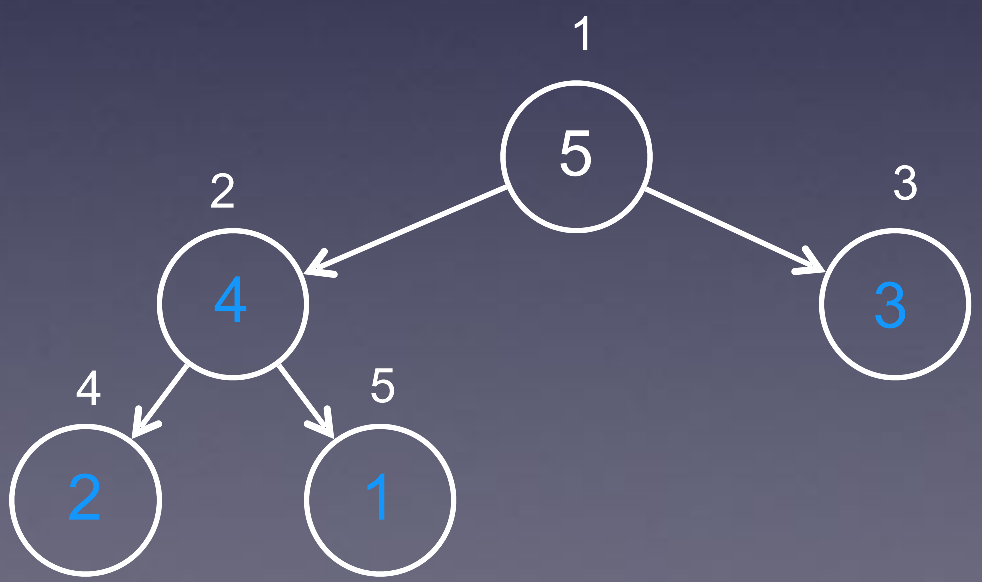


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)

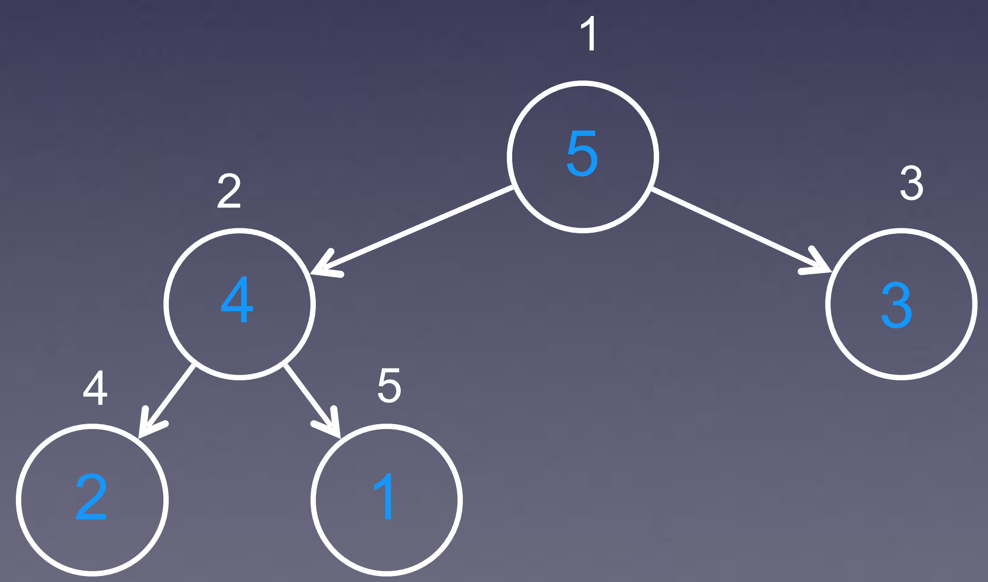


Not in the heap

Heapsort

Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap order property (using percDown)

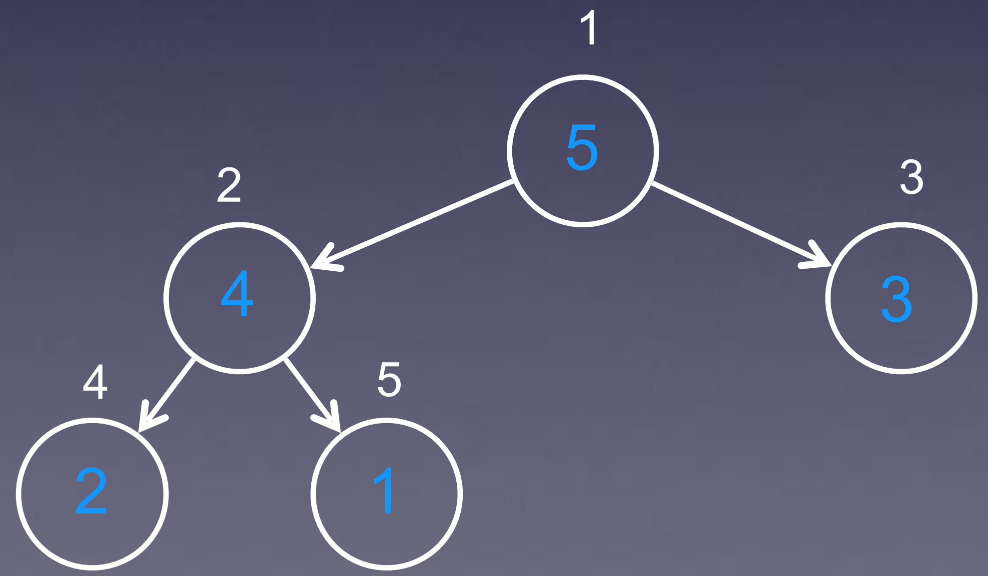


Not in the heap

Heapsort

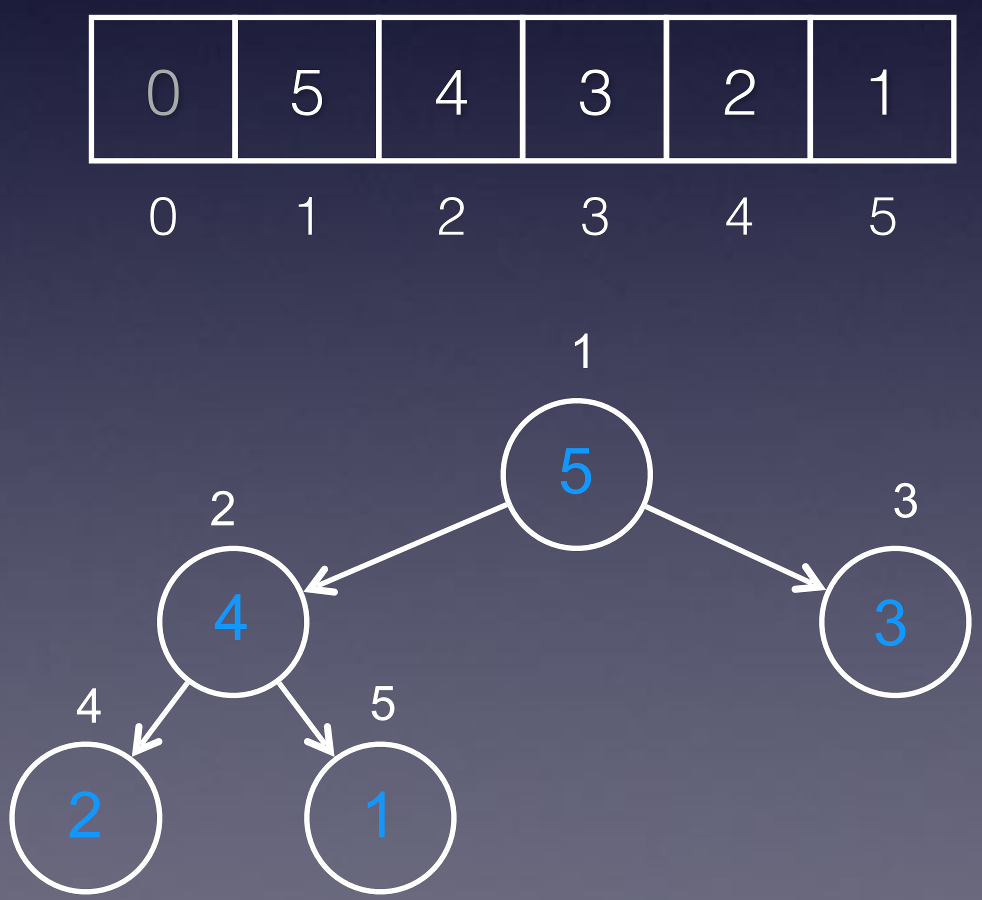
Now it's sorted.

Does it look sorted?



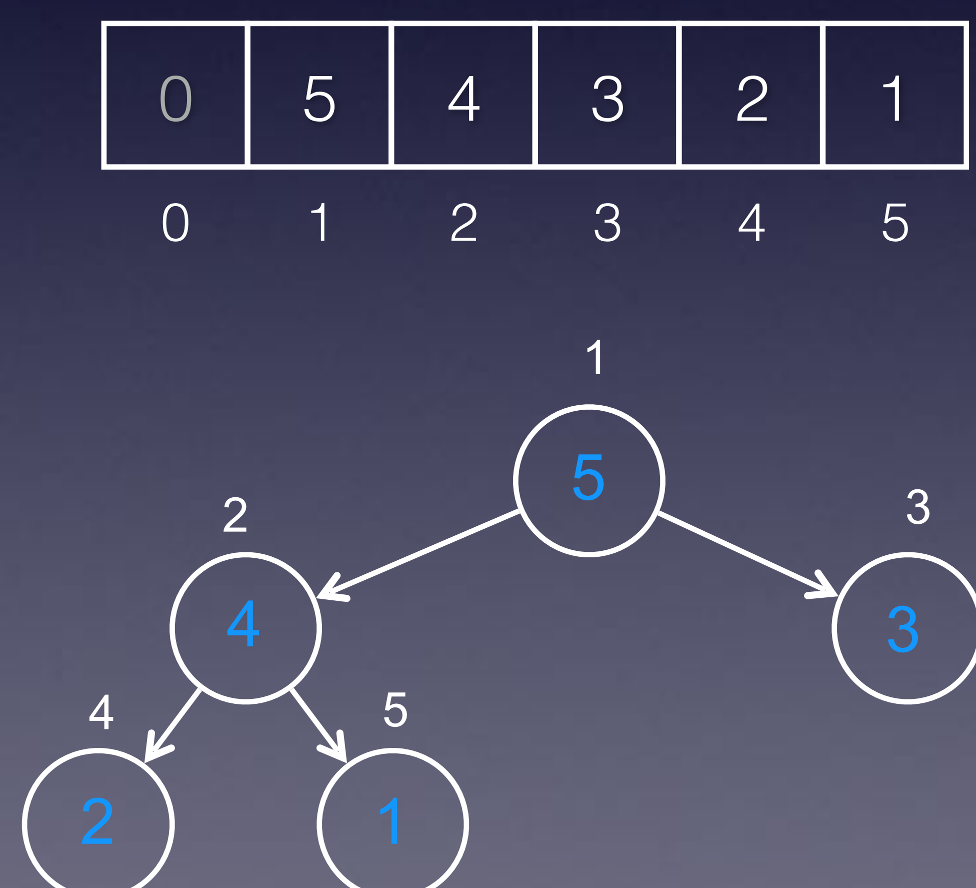
Heapsort

Look at the heap as a list:



Heapsort

We used a **min heap**, and we ended up with the sequence sorted from **largest to smallest**. A **max heap** could be used to obtain a **smallest to largest** sort order.

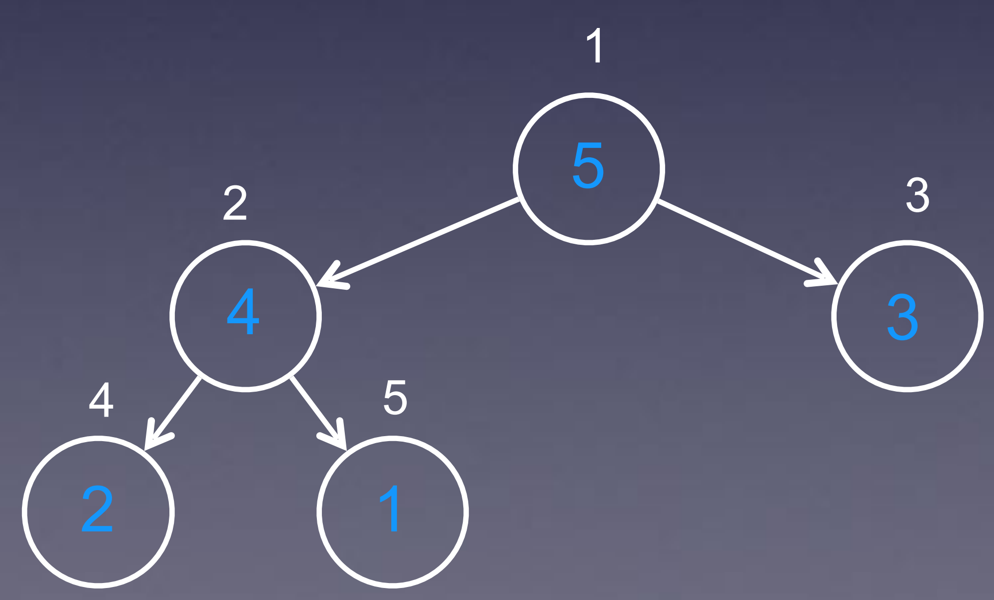


Heapsort analysis

There are n items to insert in the heap and each insert is O(logn), the time to heapify the original unsorted sequence is O(nlogn).

During sorting, we have n items to remove from the heap, which then is also O(nlogn).

Because we can do it all in the orginal array, no extra storage is required.



Points and lines structures

We've seen lots of **trees**.

One limitation of the tree data structure is the **inability to have nodes with more than one parent**

In addition, a tree imposes an ordering on the elements within (e.g. level) which may not always be desirable.

A **graph** gets around these limitations...

Points and lines structures

A **graph** is a data structure consisting of a set of vertices (nodes in our trees) and a set of edges (the connections in our trees) that describe the relationships between the vertices.

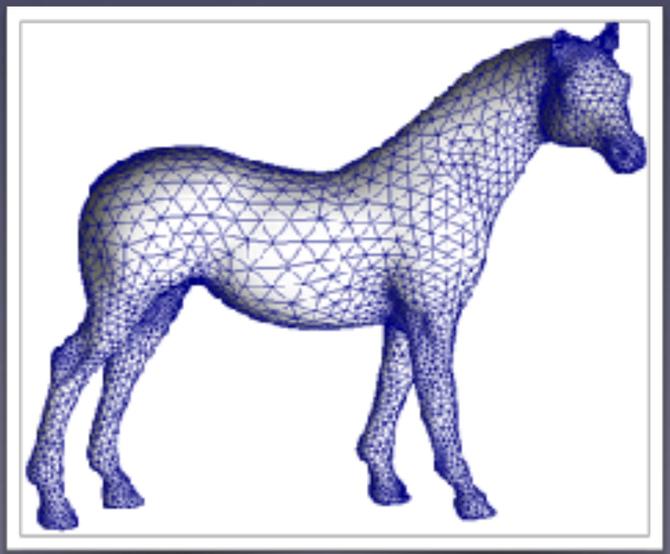
Both the vertex and the edge set may be empty (but note that you cannot have an edge without at least one vertex).

A tree is just a specialized graph.

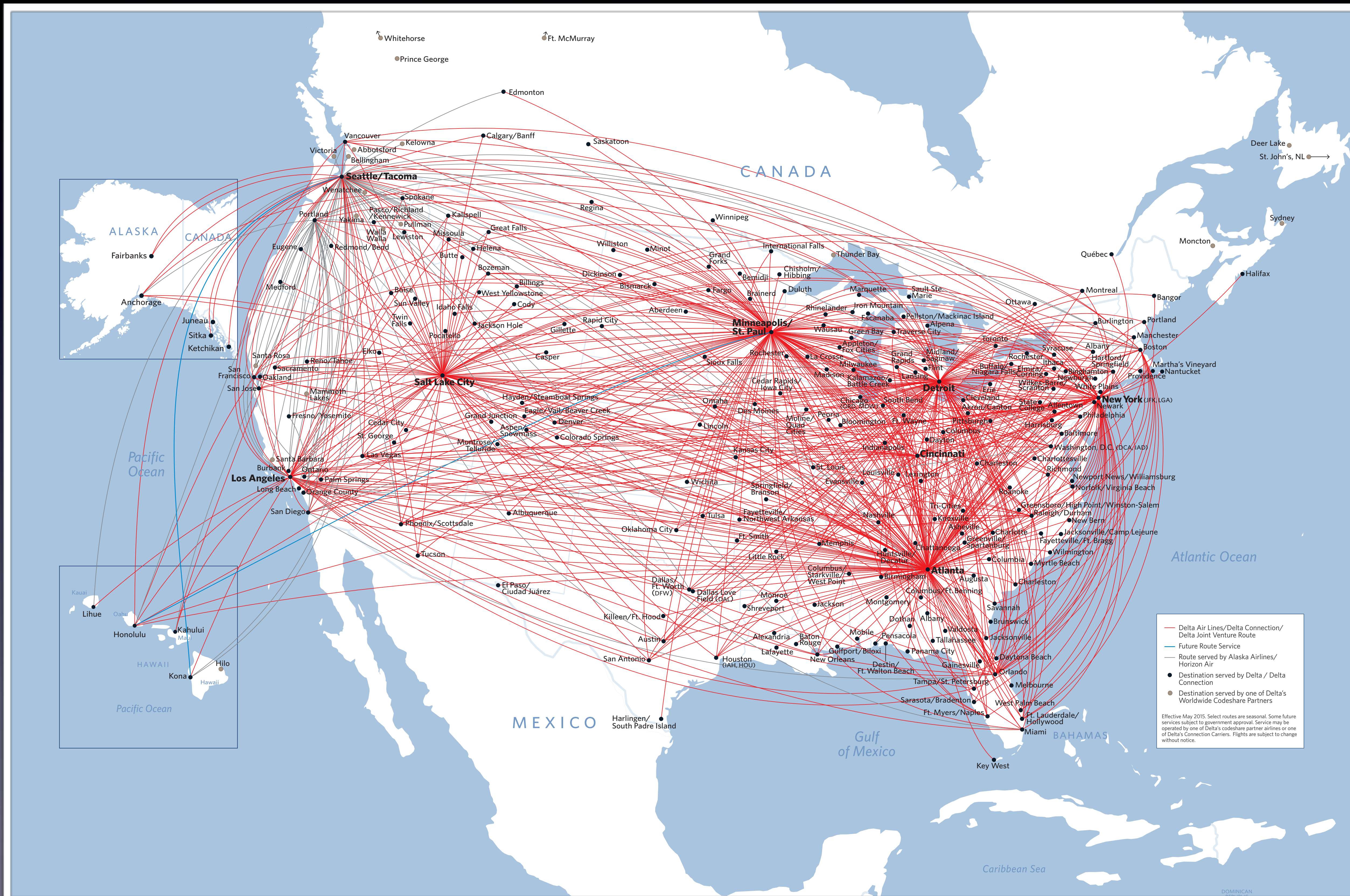
Examples of graphs in the wild

Graphs are especially useful in analyzing networks, or anything that can be represented as a collection of connected elements.

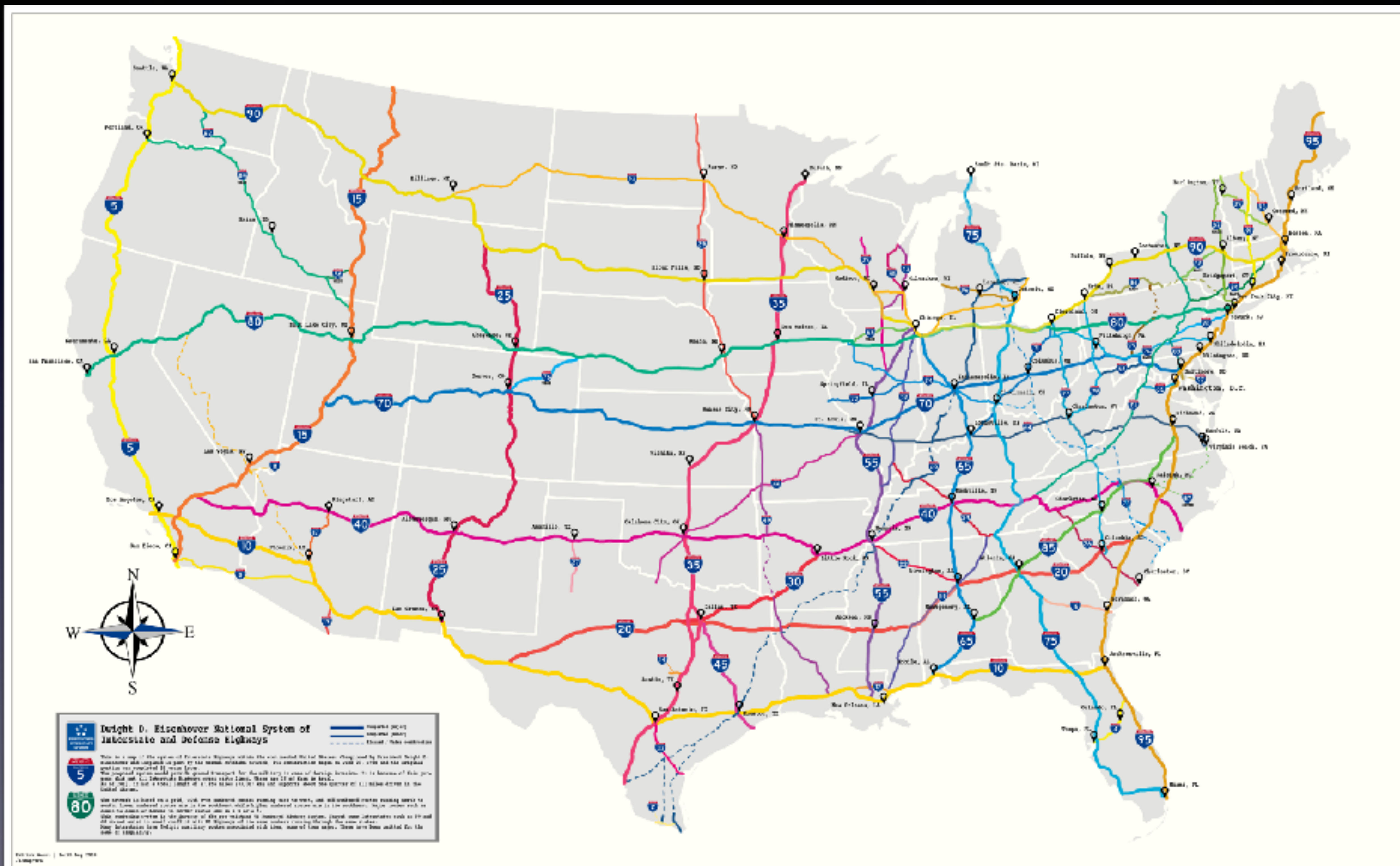
E.g. phone systems, the Internet, maps, social networks, airline routes, course prerequisites, dependency charts, silicon-chip design, plumbing, electrical circuits, meshes, Google maps, etc.



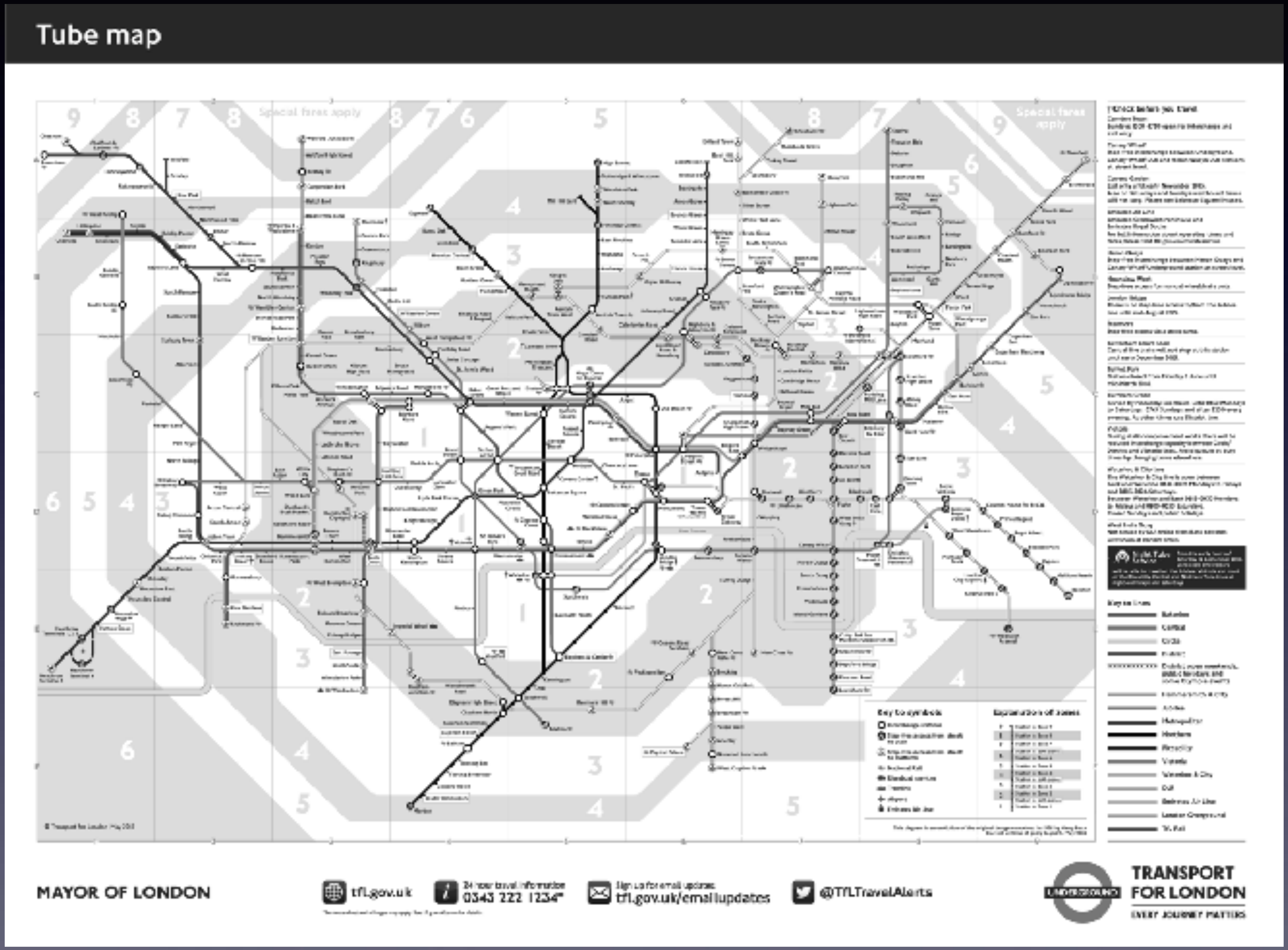
Map of Airline Routes



Map of Interstate Highways

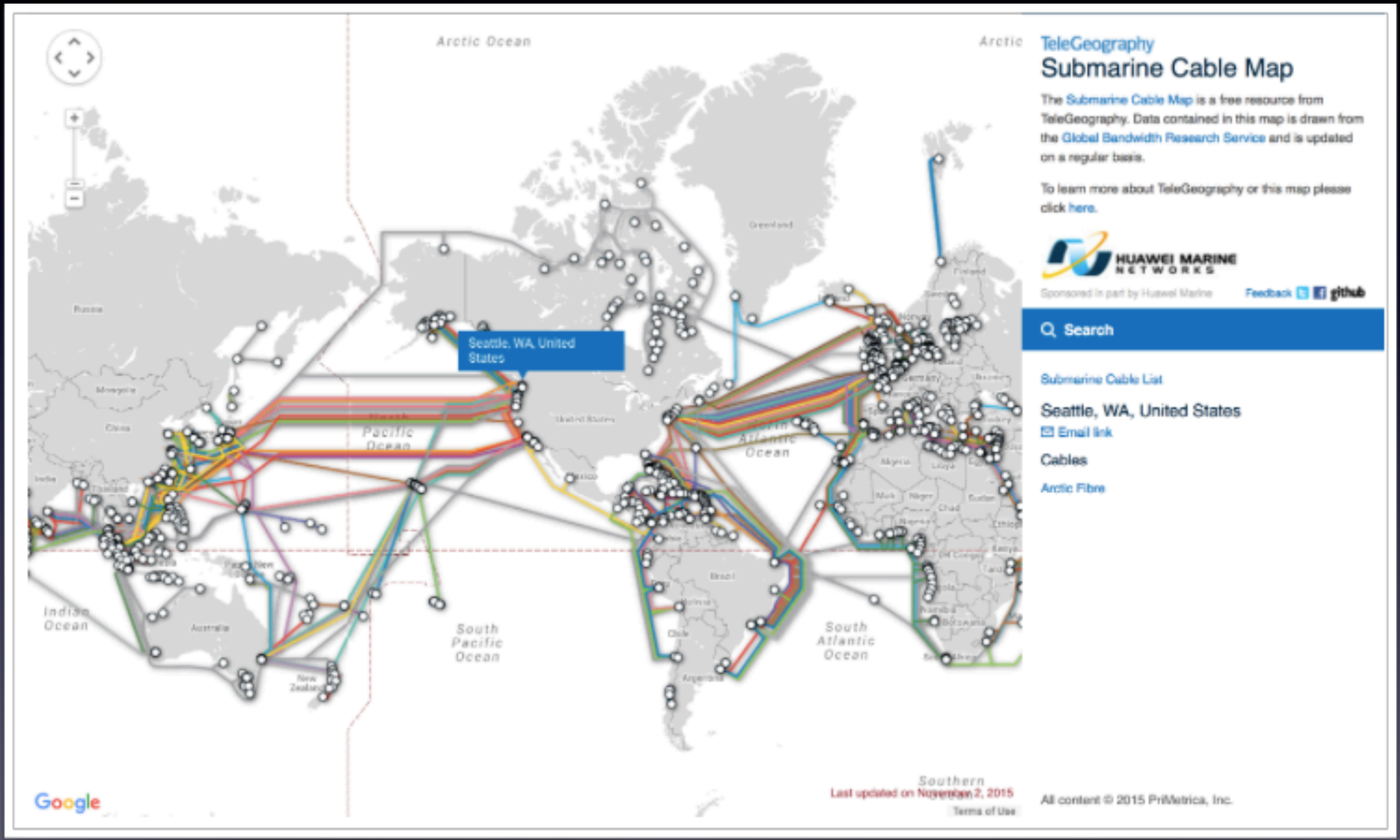


London Underground



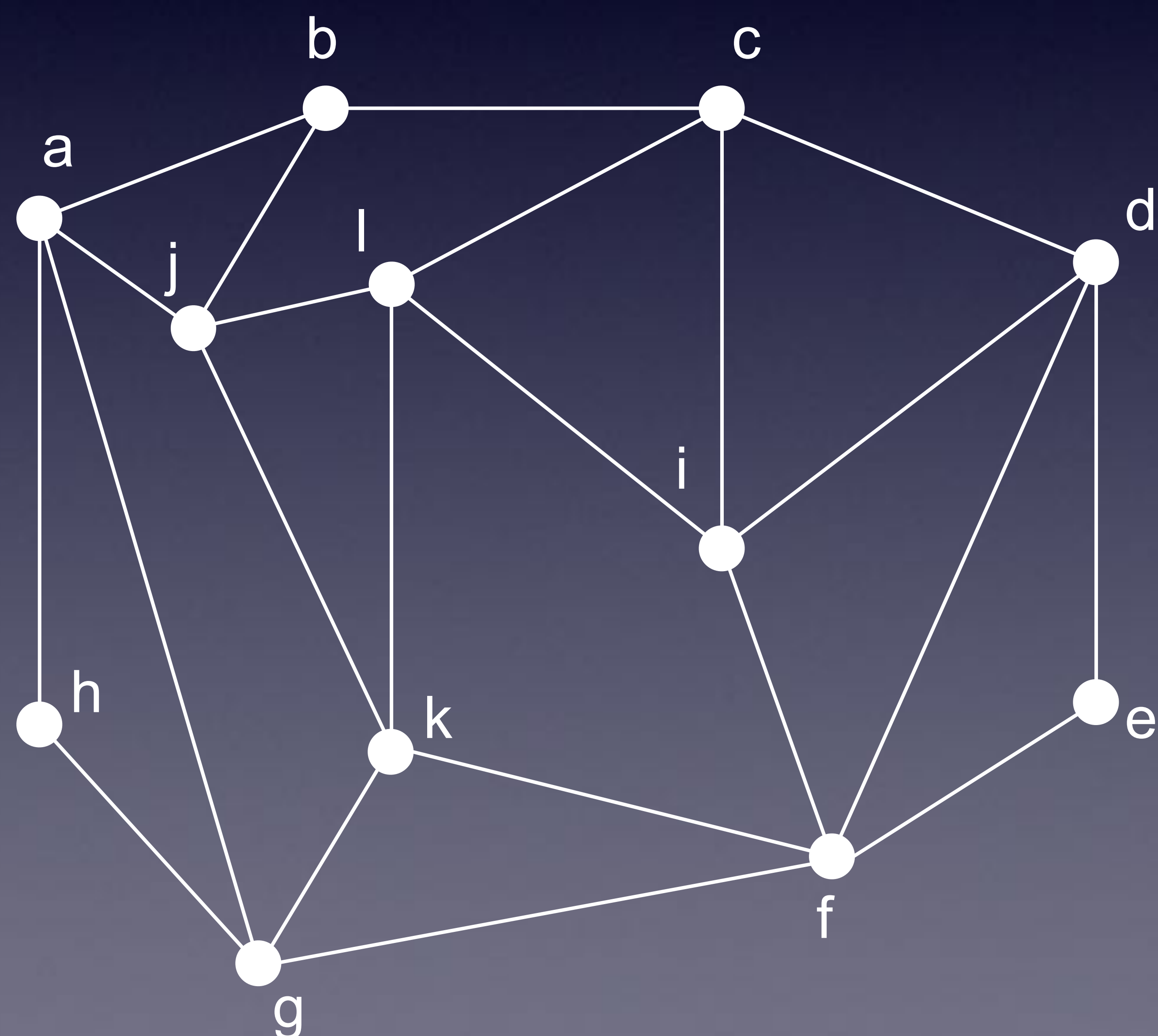
Internet and Social Networks

Internet Cable Infrastructure



Graphs, more formally

A **graph** consists of two sets
a non-empty set of **vertices** *V*, and
a set of **edges** *E* between pairs of those vertices.

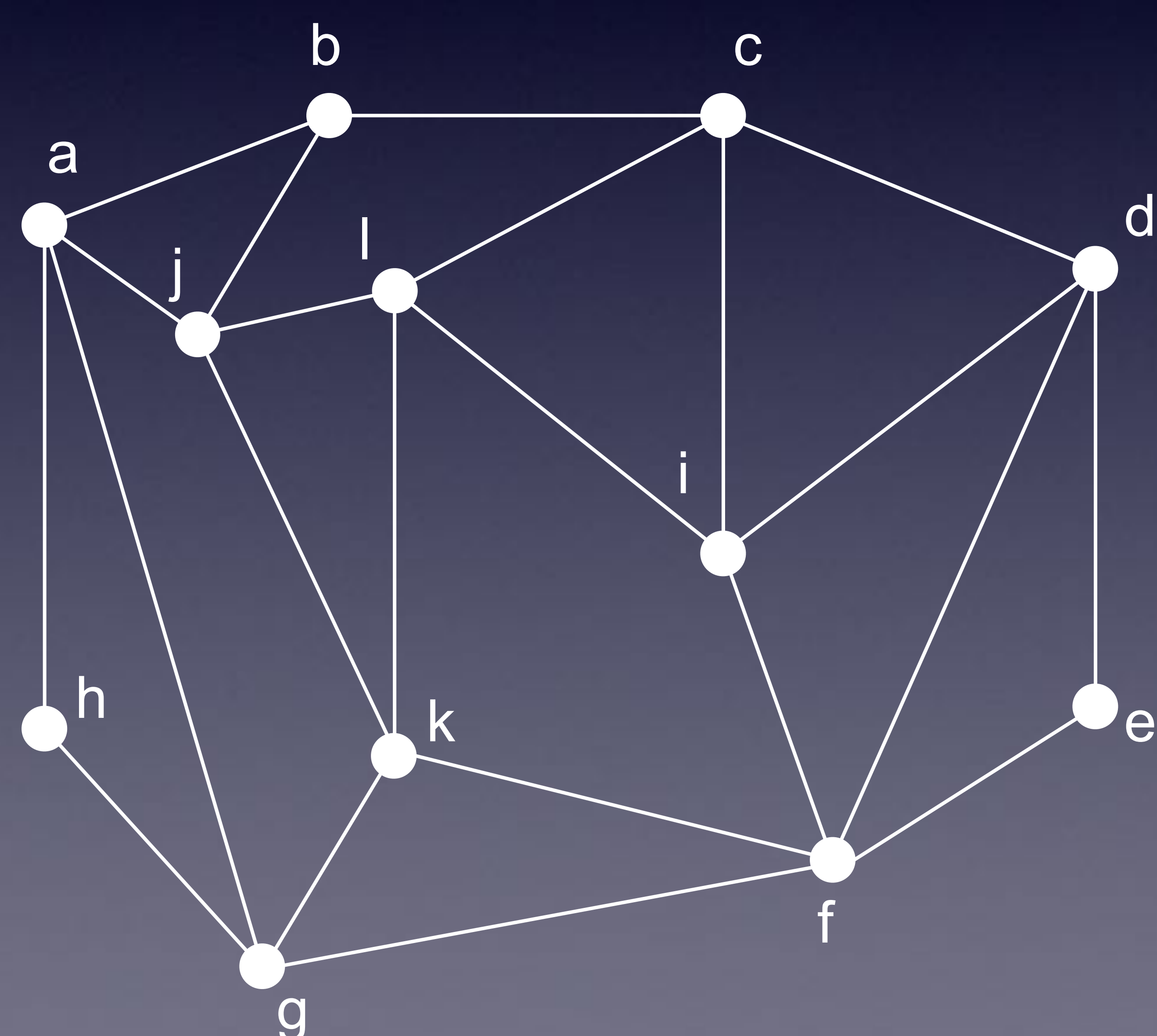


V = {a,b,c,d,e,f,g,h,i,j,k,l}

E = {(a,b),(b,c),(c,d),
(d,e),(e,f),(f,g),(g,h),(h,a),(a,j),
(a,g),(b,j),(k,f),(c,l),(c,i),(d,i),(d,f),
(f,i),(g,j),(g,k),(j,l),(j,k),(k,l),(l,i)}

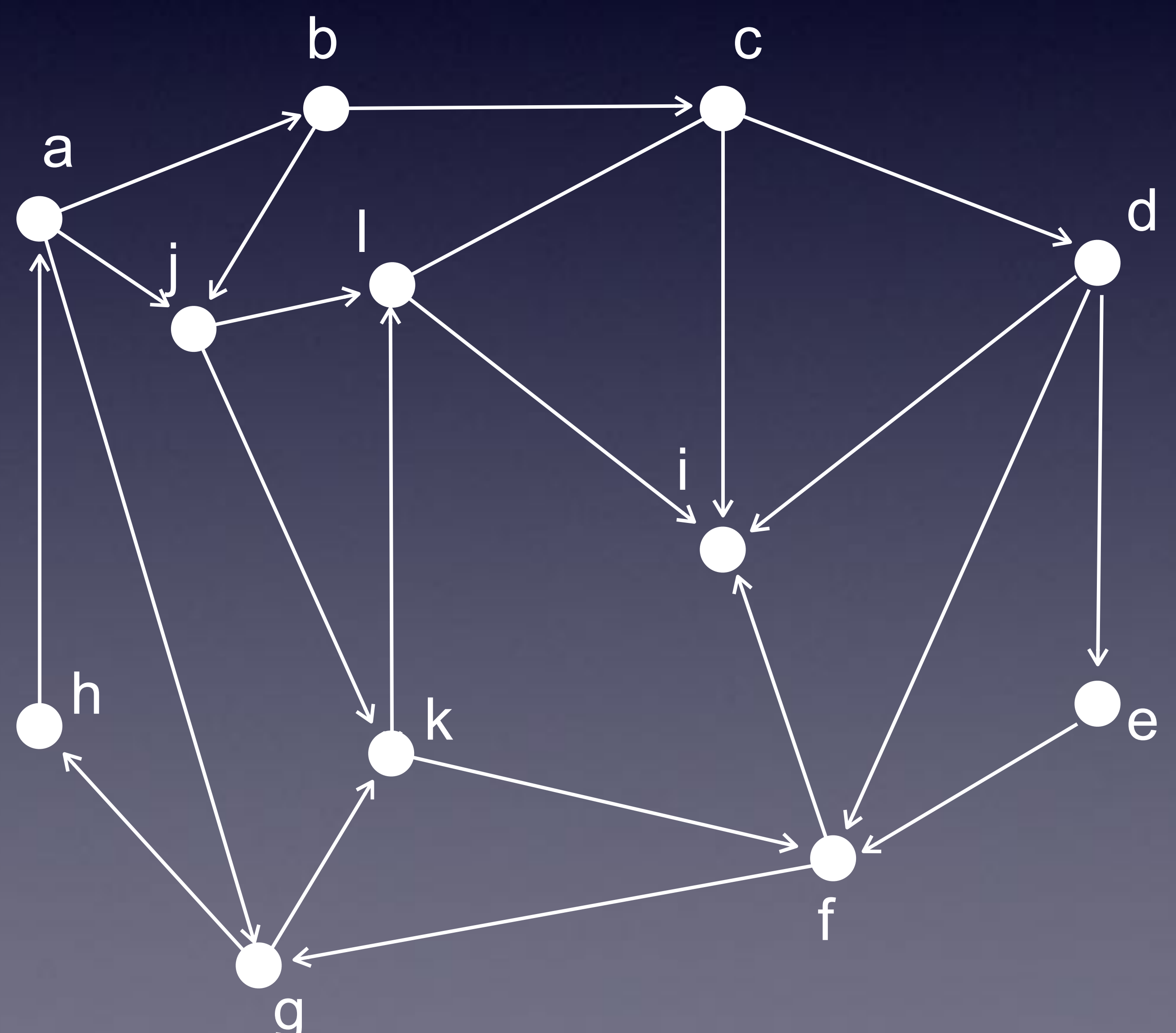
Graphs, more formally

Graphs may be **undirected**, consisting of undirected edges. For undirected edges saying (a,b) is equivalent to saying (b,a). These graphs are known as undirected graphs or just graphs.



Graphs, more formally

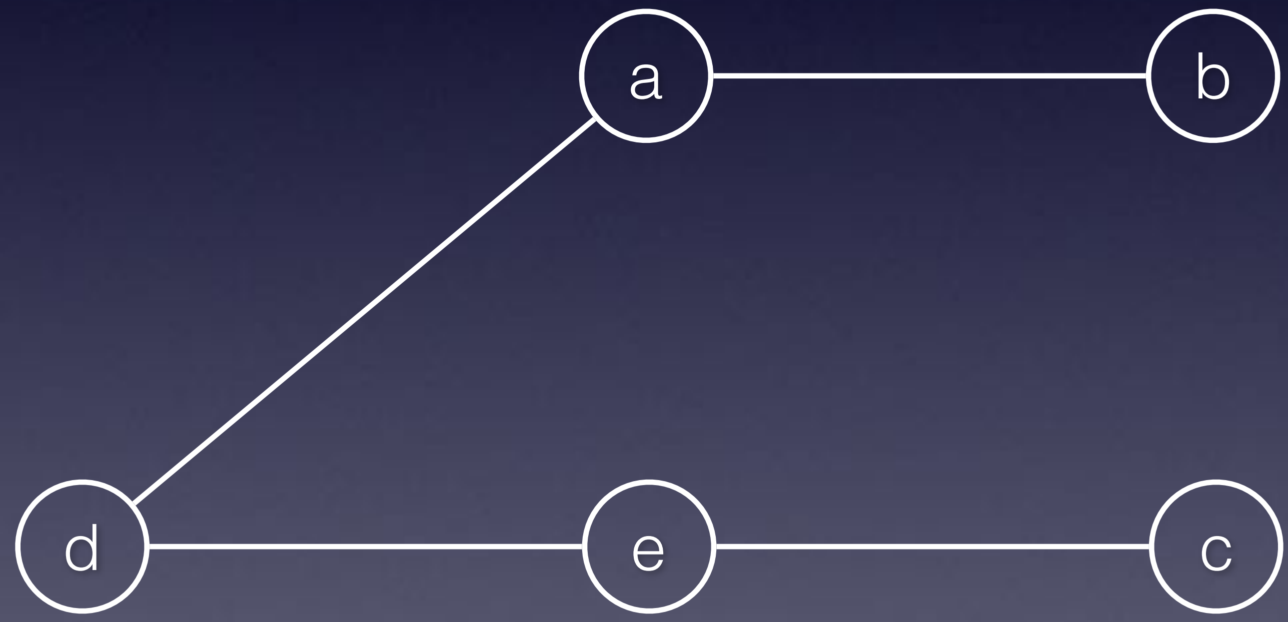
Graphs may be **directed**, consisting of directed edges. The directed edge (a,b) means from a to b. These graphs are known as directed graphs or digraphs.



V = {a,b,c,d,e,f,g,h,i,j,k,l}

E = {(a,b),(b,c),(c,d), (d,e),(e,f),(f,g),(g,h),(h,a),(a,j), (a,g),(b,j),(k,f),(c,l),(c,i),(d,i),(d,f), (f,i),(g,j),(g,k),(j,l),(j,k),(k,l),(l,i)}

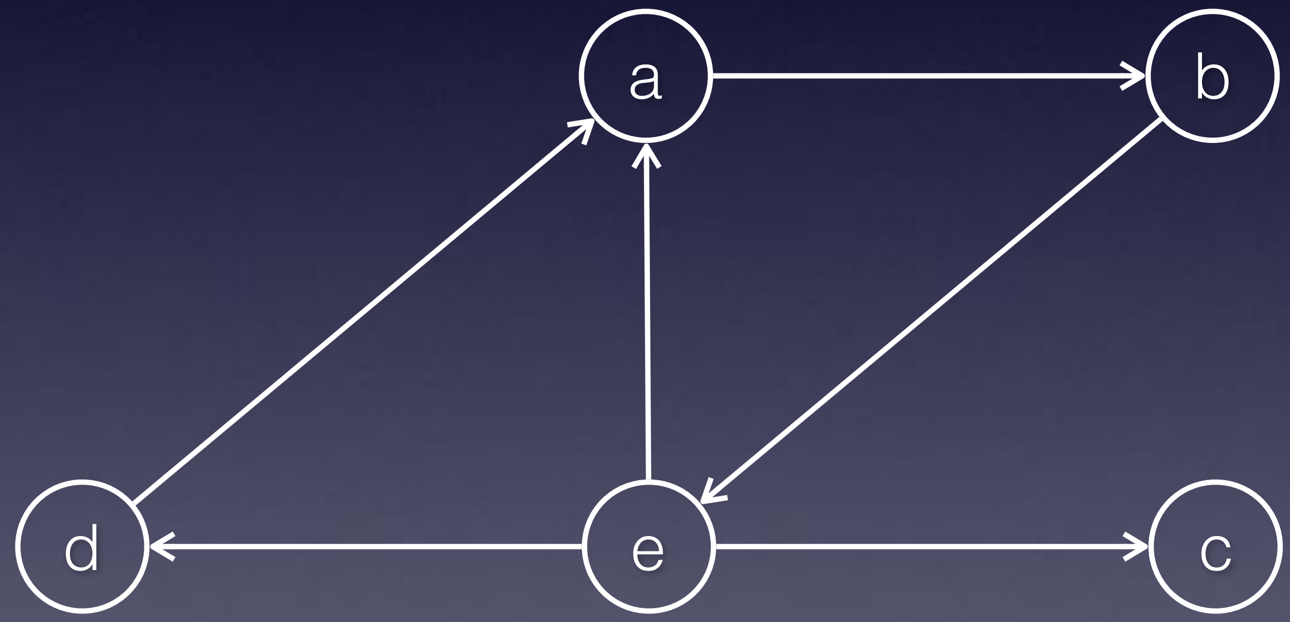
Another example



V = {a, b, c, d, e}

E = {(a, b), (a, d), (c, e), (d, e)}

Another example (directed)

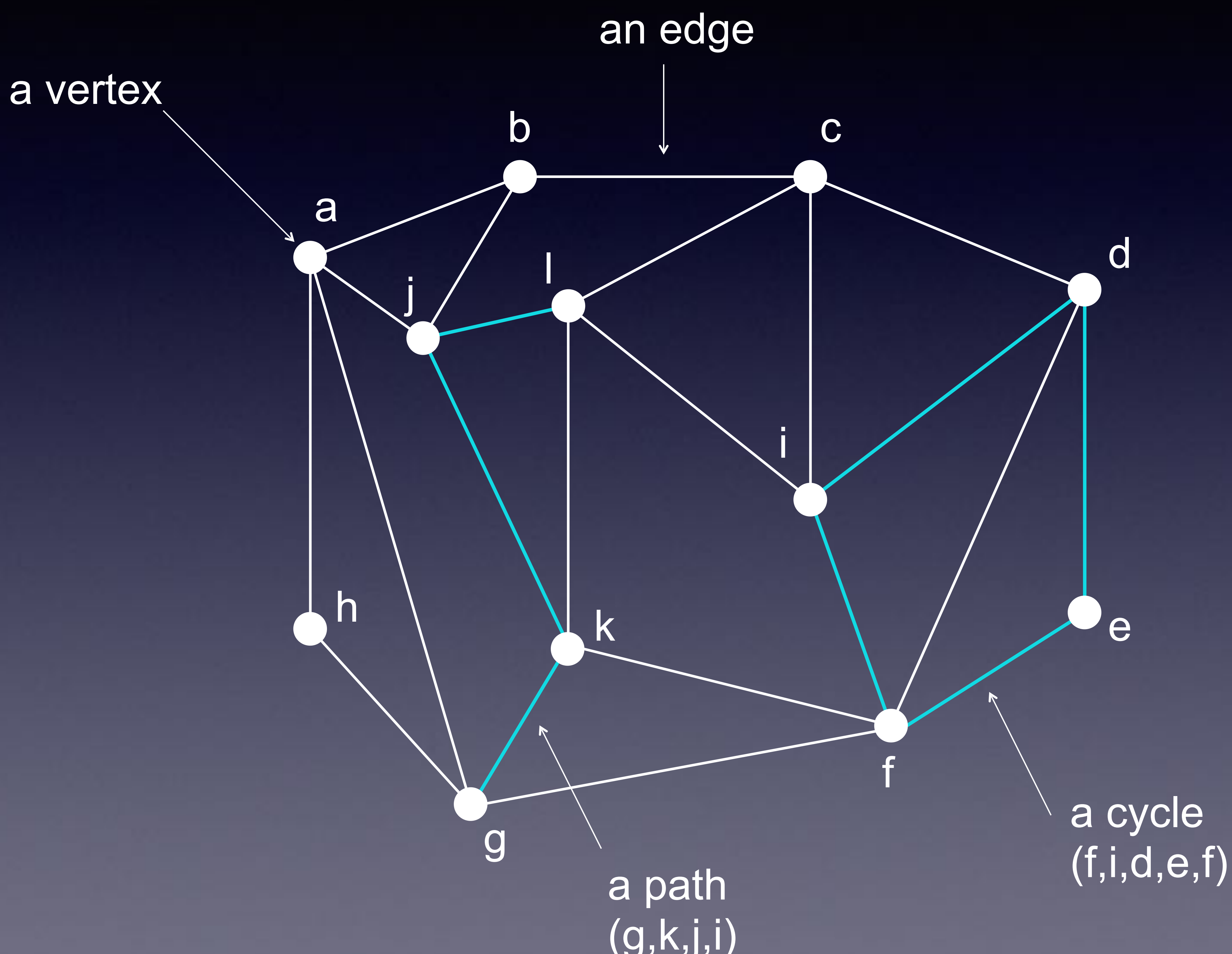


V = {a, b, c, d, e}
E = {(a, b), (b, e), (d, a), (e, a), (e, d), (e, c)}

Some Definitions

- When two vertices are connected by an edge, we say that they are **adjacent**
- A **path** is a sequence of vertices in which each successive vertex is adjacent to its predecessor. A path can be directed or undirected.
- If the first and last vertices are the same in a path, it is called a **cycle**.

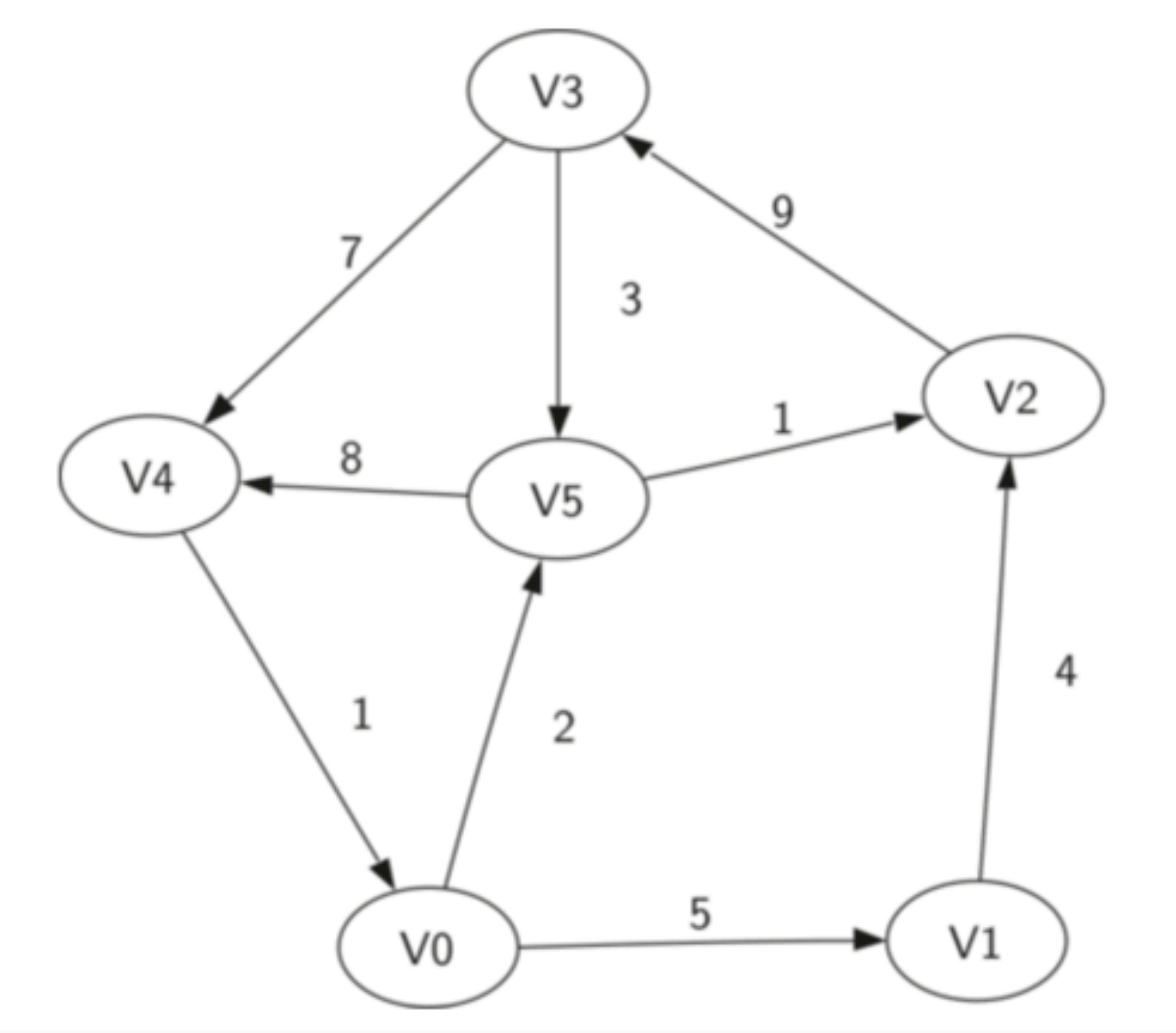
Graph anatomy



Weighted Graphs

- Edges may be weighted to show that there is a cost to go from one vertex to another.
- A weighted graph is a graph in which a value is associated with each of the edges.
- Weighted graphs may be either directed or undirected.

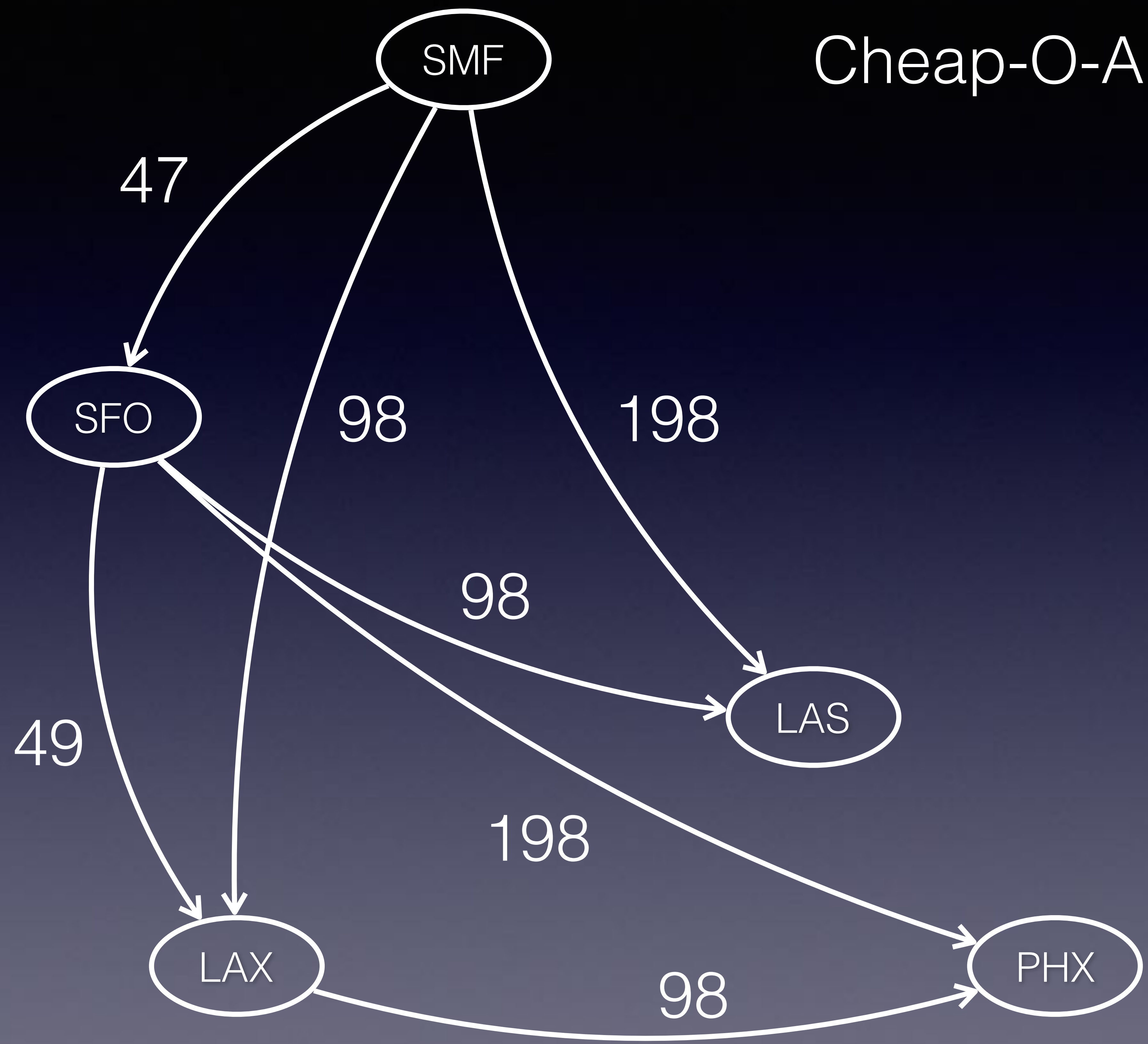
Where might a directed weighted graph like this be useful?



V={V0,V1,V2,V3,V4,V5}

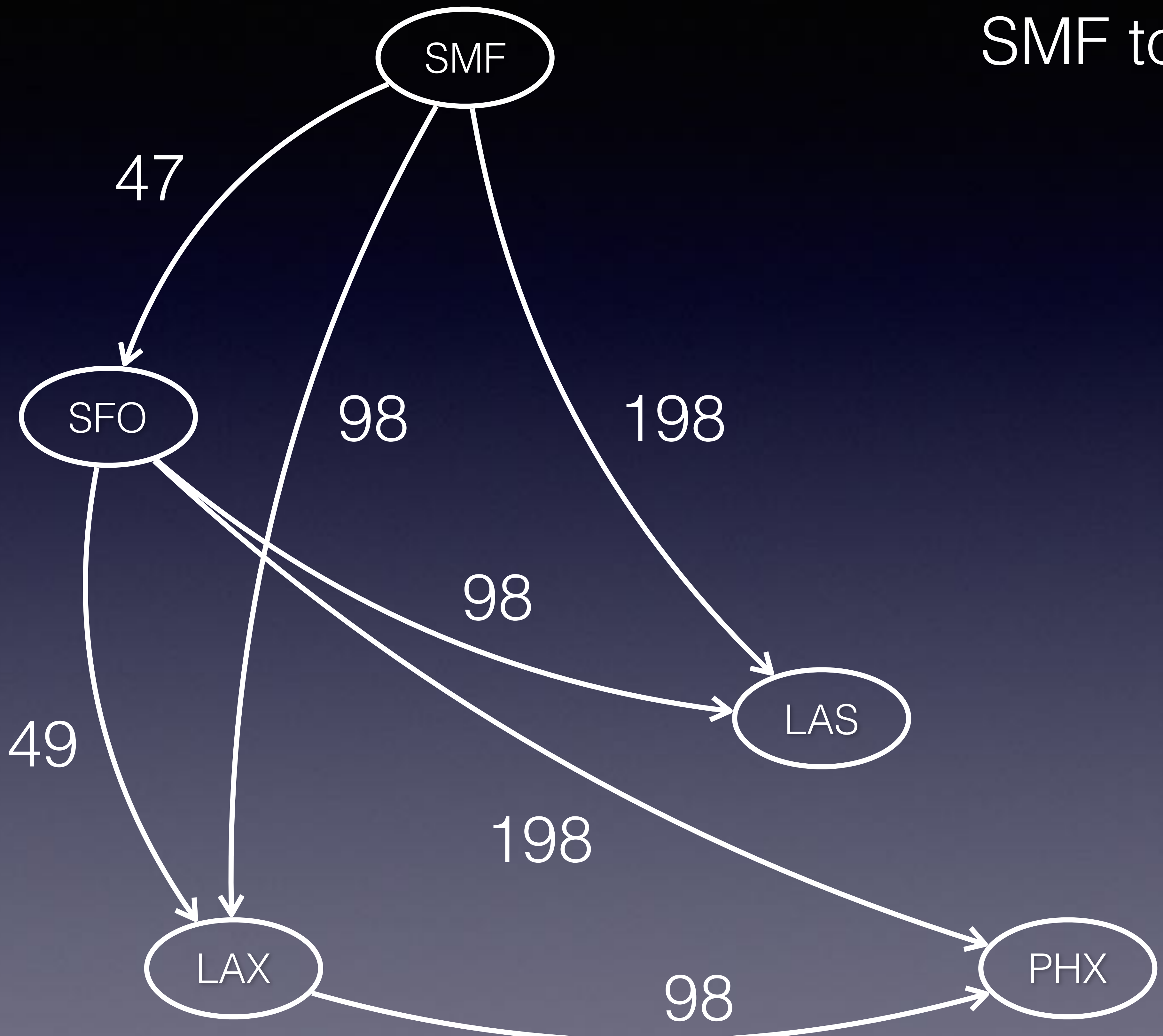
E={(V0,V1,5),(V1,V2,4),(V2,V3,9),(V3,V4,7),(V4,V0,1),

(V0,V5,2),(V5,V4,8),(V3,V5,3),(V5,V2,1)}



Cheap-O-Air?

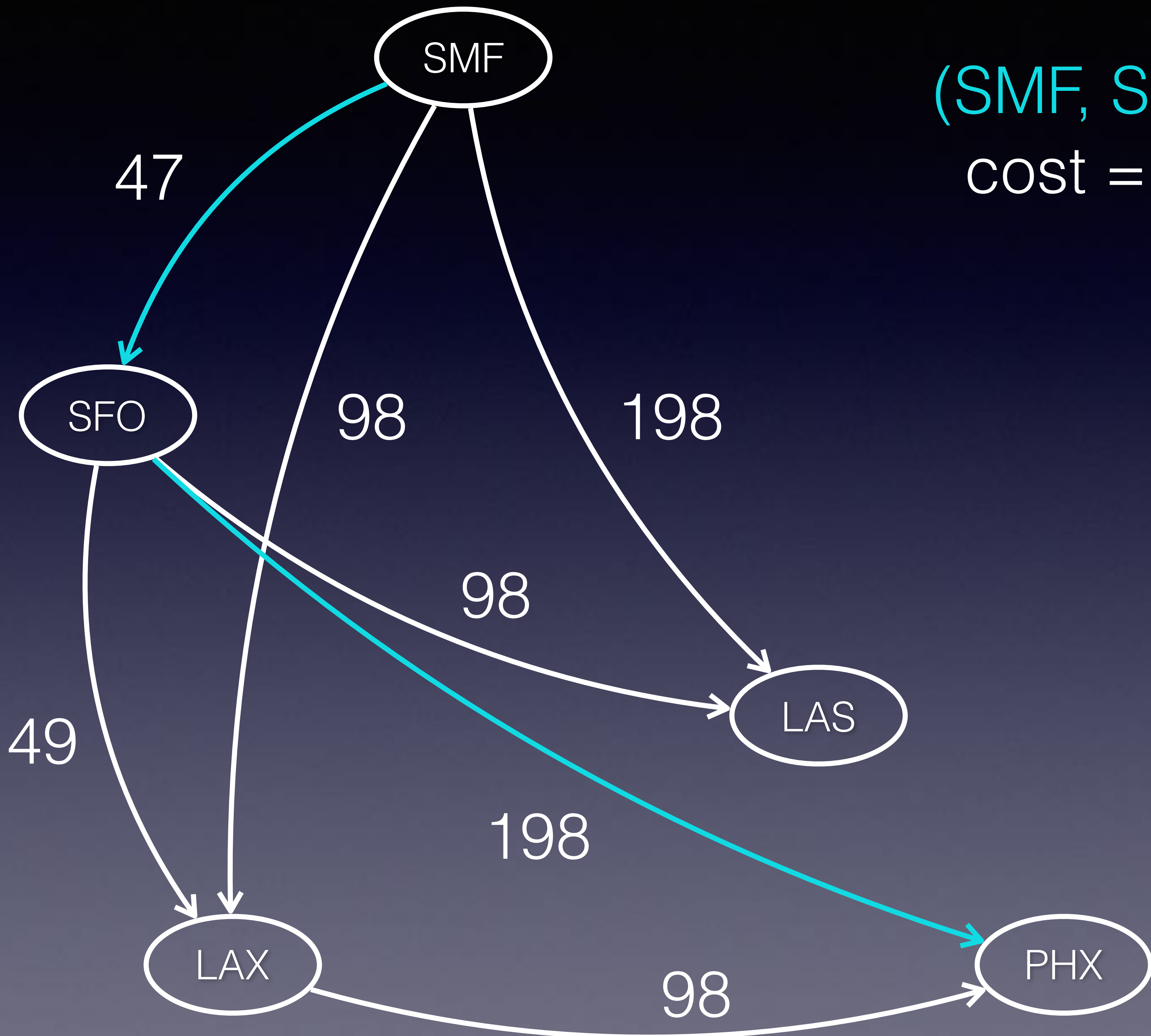
How many paths from SMF to PHX?



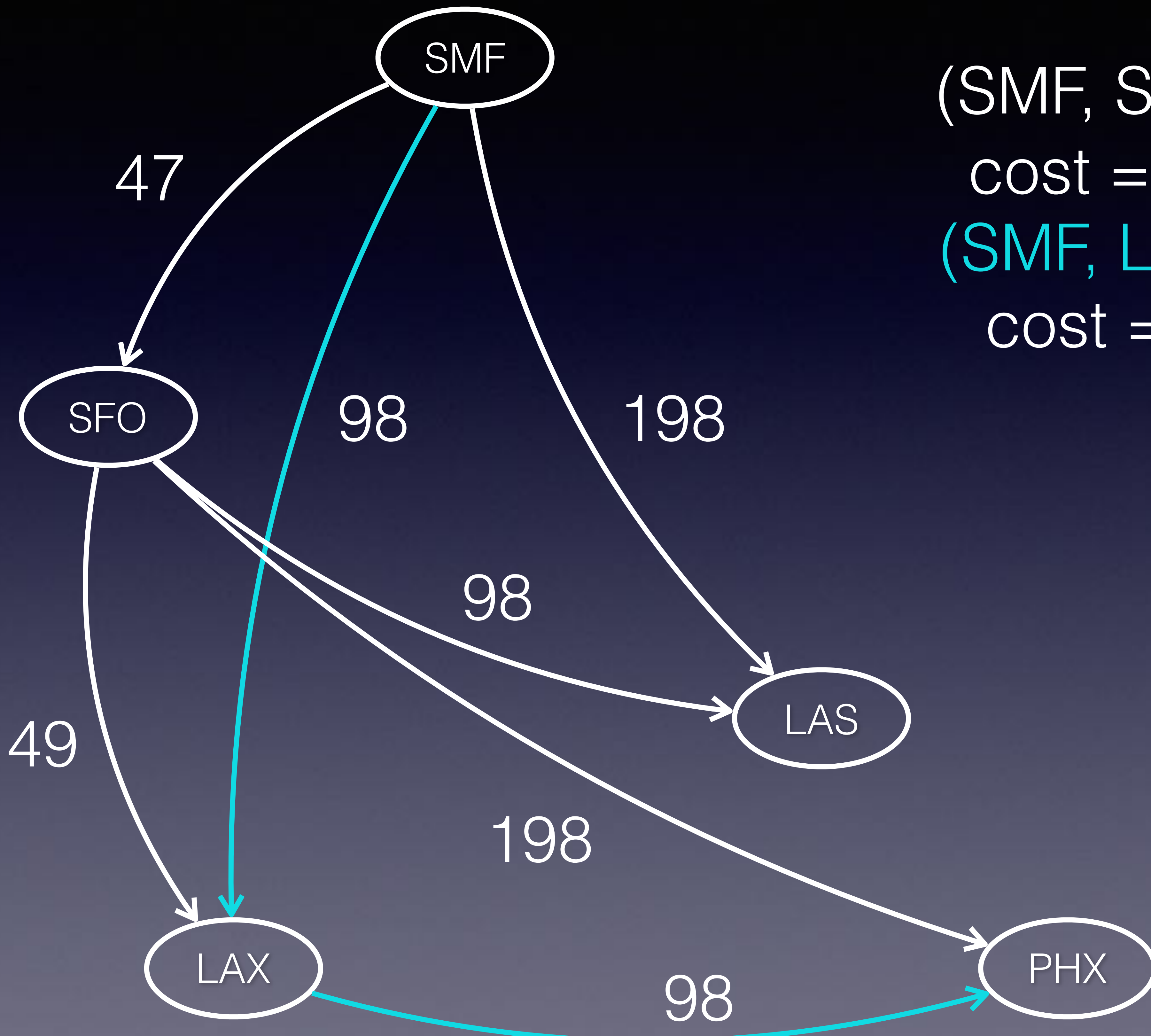
Paths from SMF to PHX

(SMF, SFO, PHX)

cost = 47+198

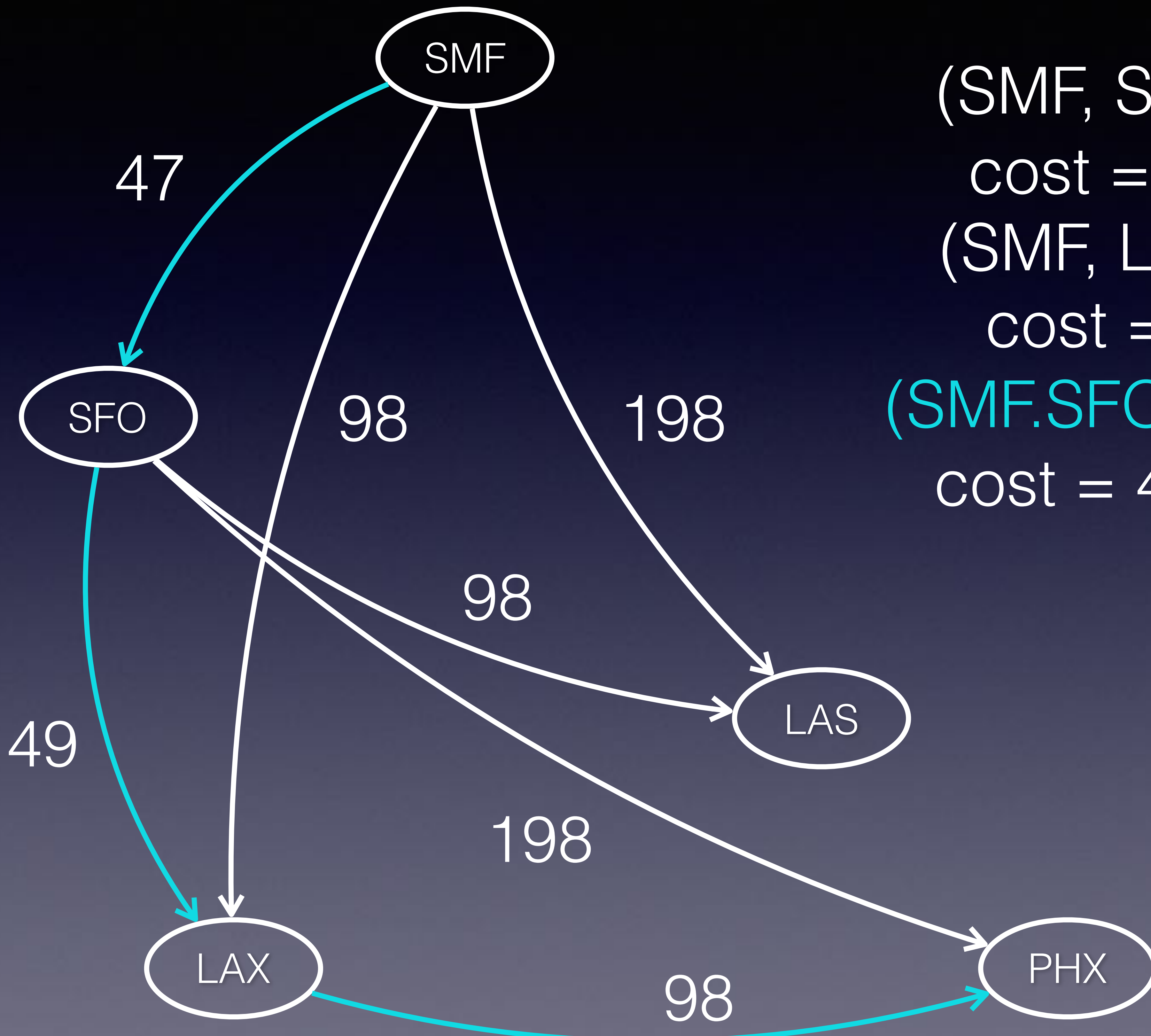


Paths from SMF to PHX



(SMF, SFO, PHX)
cost = 47+198
(SMF, LAX, PHX)
cost = 98+98

Paths from SMF to PHX



(SMF, SFO, PHX)
cost = 47+198
(SMF, LAX, PHX)
cost = 98+98
(SMF.SFO,LAX,PHX)
cost = 47+49+98