QEEE DSA05
DATA STRUCTURES AND ALGORITHMS

G VENKATESH AND MADHAVAN MUKUND

LECTURE 9, 5 SEPTEMBER 2014

Lets take stock first …

Example Problems

- Airline routes
- Job scheduling
- Document similarity

Lets take stock first …

Example Problems

- …

Complexity analysis

- O notation – asymptotic complexity

Lets take stock first …

Example Problems

- …

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort

Lets take stock first …

Example Problems

- …

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort

Data structures

- Arrays
- Linked lists

Lets take stock first …

Example Problems

- …

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort

Data structures

- Arrays
- Linked lists

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- …

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays
- Linked lists

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- Airline routes

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays
- Linked lists

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays
- Linked lists

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays – adjacency matrix
- Linked lists – adjacency list

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays – adjacency matrix
- Linked lists – adjacency list
- Queues – Breadth first
- Stacks – Depth first

Graph Traversal

Algorithmic techniques

- Divide and conquer

Lets take stock first …

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays – adjacency matrix
- Linked lists – adjacency list
- Queues – Breadth first
- Stacks – Depth first

Algorithmic techniques

- Divide and conquer
- Greedy – Dijkstra’s algorithm

Lets take stock first …

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays – adjacency matrix
- Linked lists – adjacency list
- Queues – Breadth first
- Stacks – Depth first

Algorithmic techniques

- Divide and conquer
- Greedy – Dijkstra's algorithm

$O(n^2)$

What's left to do …

Example Problems

- Airline routes
- Job scheduling
- Document similarity

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays, Linked lists
- Queues, Stacks
- Heaps
- Trees

Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic programming

Today's class

Example Problems

- Airline routes

Graphs

Complexity analysis

- O notation – asymptotic complexity
- Methods to search and sort
- Needs to be correct !

Data structures

- Arrays – adjacency matrix
- Linked lists – adjacency list
- Heaps

Algorithmic techniques

- Divide and conquer
- Greedy – Dijkstra's algorithm

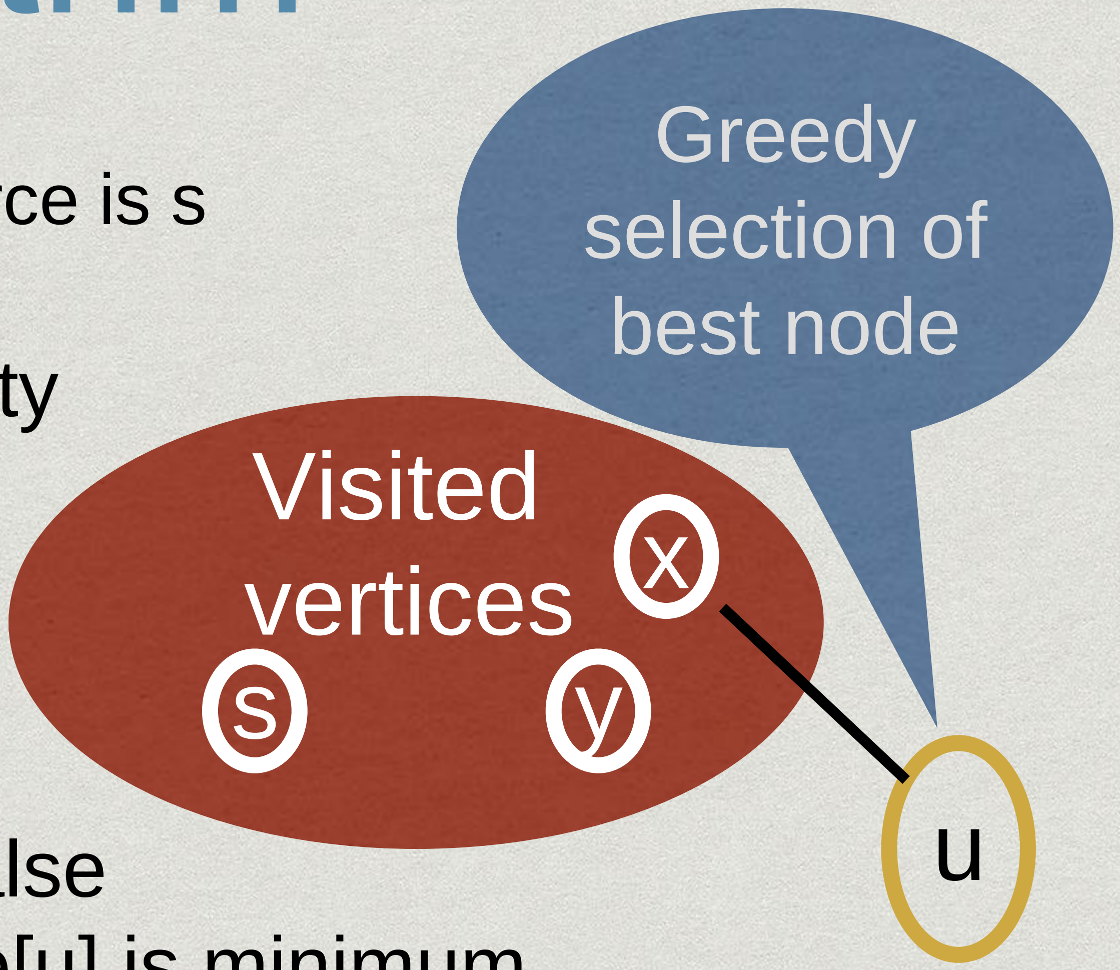
$$O((n+m)\log n)$$

Dijkstra’s algorithm

```
function ShortestPaths(s){ // assume source is s
   for i = 1 to n
      Visited[i] = False; Distance[i] = infinity

   Distance[s] = 0

   for i = 1 to n
      Choose u such that Visited[u] == False
                                  and Distance[u] is minimum
      Visited[u] = True
      for each edge (u,v) with Visited[v] == False
         if Distance[v] > Distance[u] + weight(u,v)
            Distance[v] = Distance[u] + weight(u,v)
```

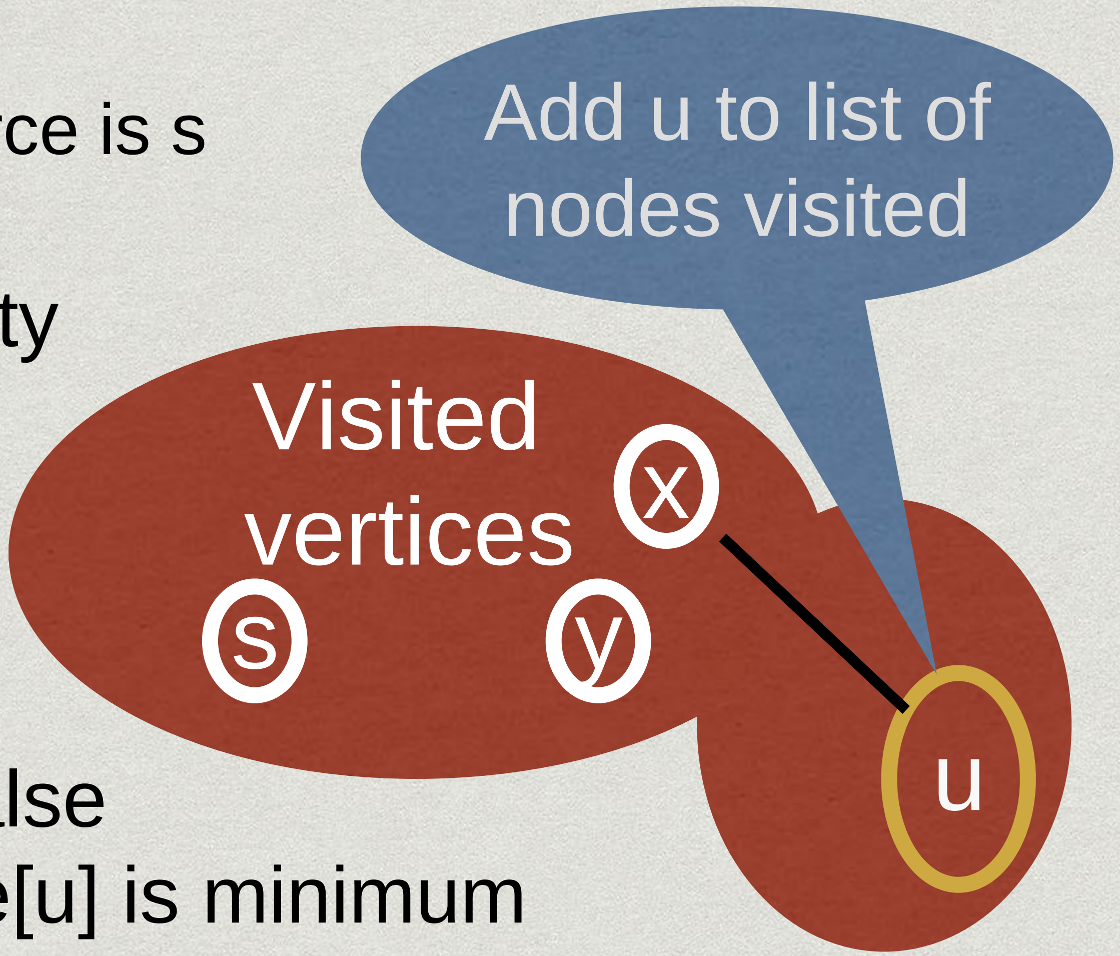
Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
   for i = 1 to n
      Visited[i] = False; Distance[i] = infinity

   Distance[s] = 0

   for i = 1 to n
      Choose u such that Visited[u] == False
                                         and Distance[u] is minimum
      Visited[u] = True
      for each edge (u,v) with Visited[v] == False
         if Distance[v] > Distance[u] + weight(u,v)
            Distance[v] = Distance[u] + weight(u,v)
```



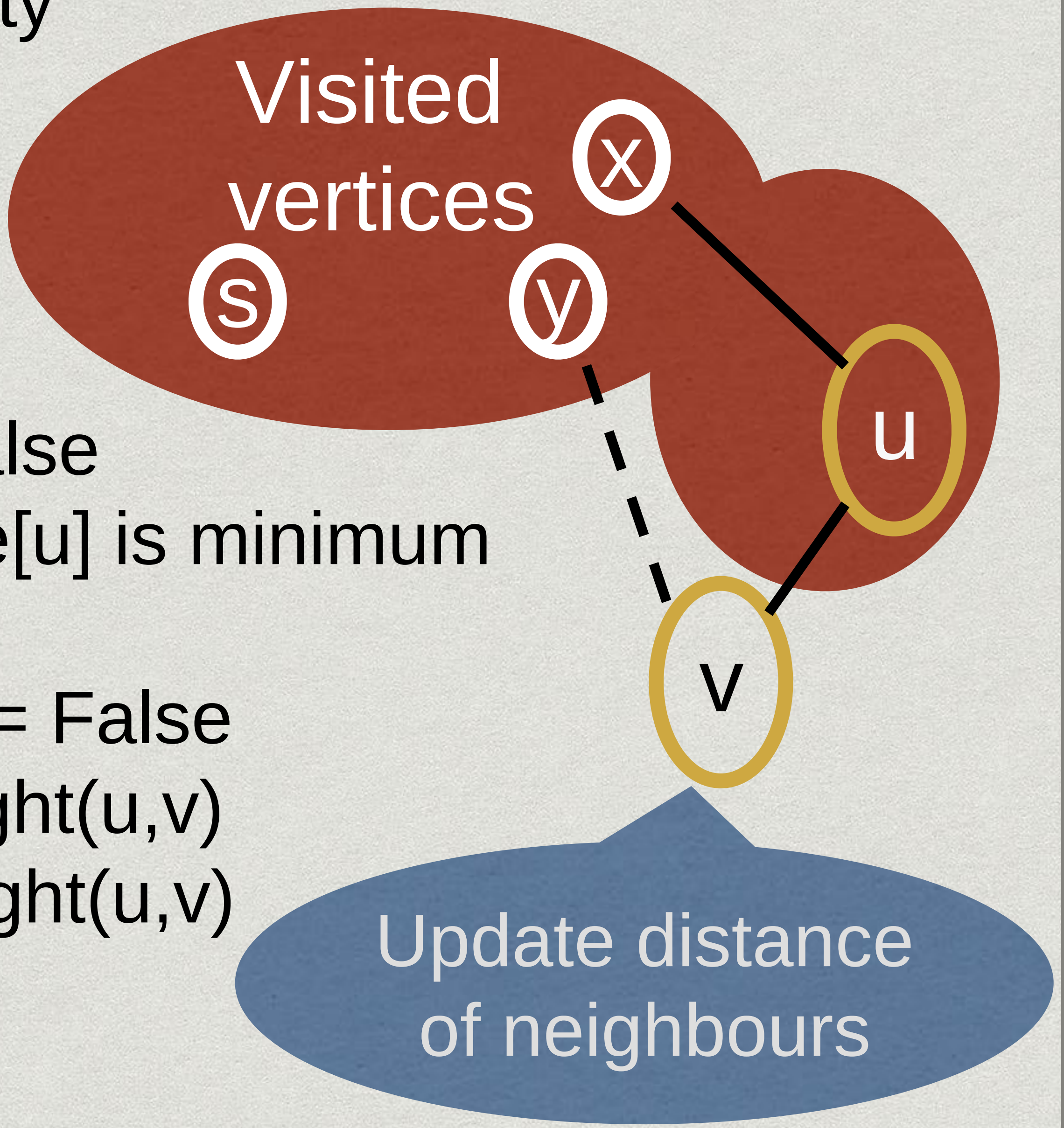
Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    Visited[i] = False; Distance[i] = infinity

  Distance[s] = 0

  for i = 1 to n
    Choose u such that Visited[u] == False
                                          and Distance[u] is minimum
    Visited[u] = True
    for each edge (u,v) with Visited[v] == False
      if Distance[v] > Distance[u] + weight(u,v)
        Distance[v] = Distance[u] + weight(u,v)
```



Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
   for i = 1 to n
      Visited[i] = False; Distance[i] = infinity

   Distance[s] = 0

   for i = 1 to n
      Choose u such that Visited[u] == False
                                  and Distance[u] is minimum
      Visited[u] = True
      for each edge (u,v) with Visited[v] == False
         if Distance[v] > Distance[u] + weight(u,v)
            Distance[v] = Distance[u] + weight(u,v)
```



Improving the algorithm

Key steps in the algorithm:

- Select an unvisited node u with the least Distance value
- Remove u from the list of unvisited nodes
- Access all neighbours of u and update distance
  - in particular, this step will change the distance value of some nodes, so will affect the selection step

Improving the algorithm

Key steps in the algorithm:

- Select an unvisited node u with the least Distance value
  - can this be done in $O(\log n)$ time rather than $O(n)$?

- Remove u from the list of unvisited nodes
  - can this be done in $O(\log n)$ time rather than $O(n)$?

- Access all neighbours of u and update distance
  - can this update be done in $O(\log n)$ time per edge without affecting the complexity of the first step

Priority queue

A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value $key(v)$ that denotes the priority of the element v.

Priority queue

A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value $key(v)$ that denotes the priority of the element v.

Smaller keys represent higher priorities.

Priority queue

A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value $key(v)$ that denotes the priority of the element v.

Smaller keys represent higher priorities.

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

Priority queue

A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value $key(v)$ that denotes the priority of the element v.

Smaller keys represent higher priorities.

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

We need one more operation: change the value of the key
But this can be simulated using the deletion and addition operations

Examples of priority queues

OS needs to execute a set of processes on a computer, the processes have different levels of priority and don’t arrive in order of priority

Flights are arriving into an airport. The Air traffic controller needs to reserve runway time for each aircraft. Requests for landing time can come anytime during the flight. Flights nearer to the airport need to be given priority.

Priority queue

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

Can we do each of the operations of the priority queue in $O(\log n)$ time?

Priority queue

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

Can we do each of the operations of the priority queue in $O(\log n)$ time?

Note: $\log n$ is the best possible time for these operations
Why?

Priority queue

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

$\log n$ is the lower bound for at least one of these operations

Consider the sorting of n numbers. We construct a priority queue into which we insert numbers one by one. We then extract the numbers (select the lowest and delete). Numbers will come out in sorted order.

Priority queue

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

$\log n$ is the lower bound for at least one of these operations

Consider the sorting of n numbers. We construct a priority queue into which we insert numbers one by one. We then extract the numbers (select the lowest and delete). Numbers will come out in sorted order.

We are doing n inserts, selects and deletes here.

Priority queue

Priority queue allows the following operations:

- Addition of an element to the set
- Deletion of an element from the set
- Selection of an element with the smallest key

$\log n$ is the lower bound for at least one of these operations

Use a priority queue to sort the numbers.
We are doing n inserts, selects and deletes here.

Sorting has a lower bound of $n \log n$ - which means that at least one of the steps - insert, select or delete should have a lower bound of $\log n$.

Implementing a priority queue

We know two ways to implement: arrays and linked lists

Implementing a priority queue

We know two ways to implement: arrays and linked lists

Lets try an Array first

Implementing a priority queue

We know two ways to implement: arrays and linked lists

Lets try an Array first

- We should maintain the elements in sorted order of keys

Implementing a priority queue

We know two ways to implement: arrays and linked lists

Lets try an Array first

- We should maintain the elements in sorted order of keys
- Selection of min key element is easy – pick the first one

Implementing a priority queue

We know two ways to implement: arrays and linked lists

Lets try an Array first

- We should maintain the elements in sorted order of keys
- Selection of min key element is easy – pick the first one
- Insertion: we can do binary search to locate the index where the element needs to be inserted. However, insertion requires shifting all the higher elements to the right – which can take $O(n)$ time

Implementing a priority queue

We know two ways to implement: arrays and linked lists

What about linked lists?

Implementing a priority queue

We know two ways to implement: arrays and linked lists

What about linked lists?

- Maintain the list in ascending order?

Implementing a priority queue

We know two ways to implement: arrays and linked lists

What about linked lists?

- Maintain the list in ascending order?
- Selecting min key is easy – pick the head of the list
- Insertion: searching for the right place to insert can take $O(n)$ time; once located, the insertion takes constant time

Implementing a priority queue

We know two ways to implement: arrays and linked lists

What about linked lists?

- Maintain the list in any order, but with a pointer to the min

Implementing a priority queue

We know two ways to implement: arrays and linked lists

What about linked lists?

- Maintain the list in any order, but with a pointer to the min
- Selecting min key is easy – use the min pointer
- Insertion: insert at the beginning of the list
  But how do we know if the min has changed?
  This requires list to be scanned - $O(n)$ time

Heap data structure

Combines the benefits of a sorted array and a list

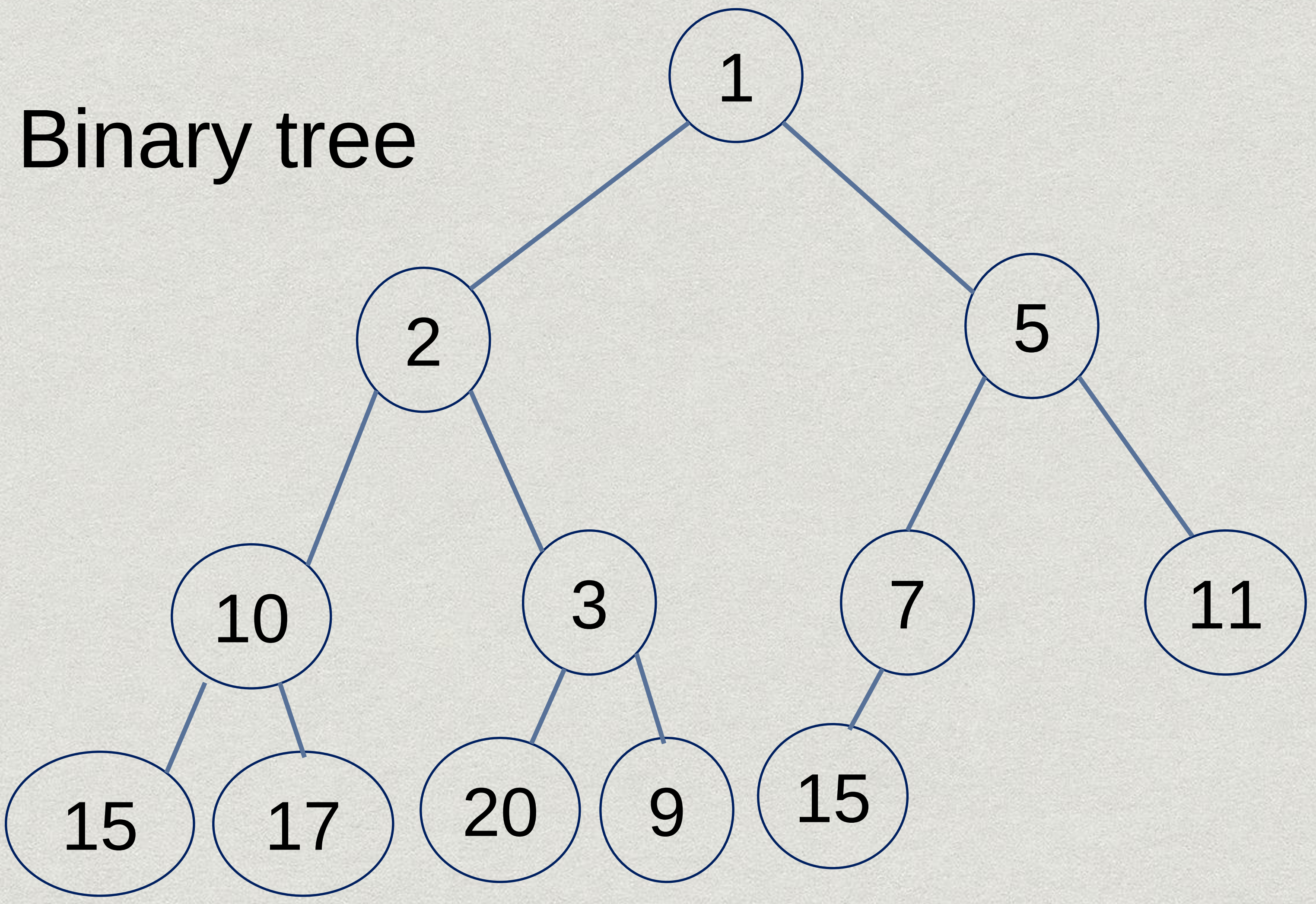
- List is good for insert, delete
- Sorted array is good for finding the min

Heap data structure

Combines the benefits of a sorted array and a list

- List is good for insert, delete
- Sorted array is good for finding the min

Binary tree

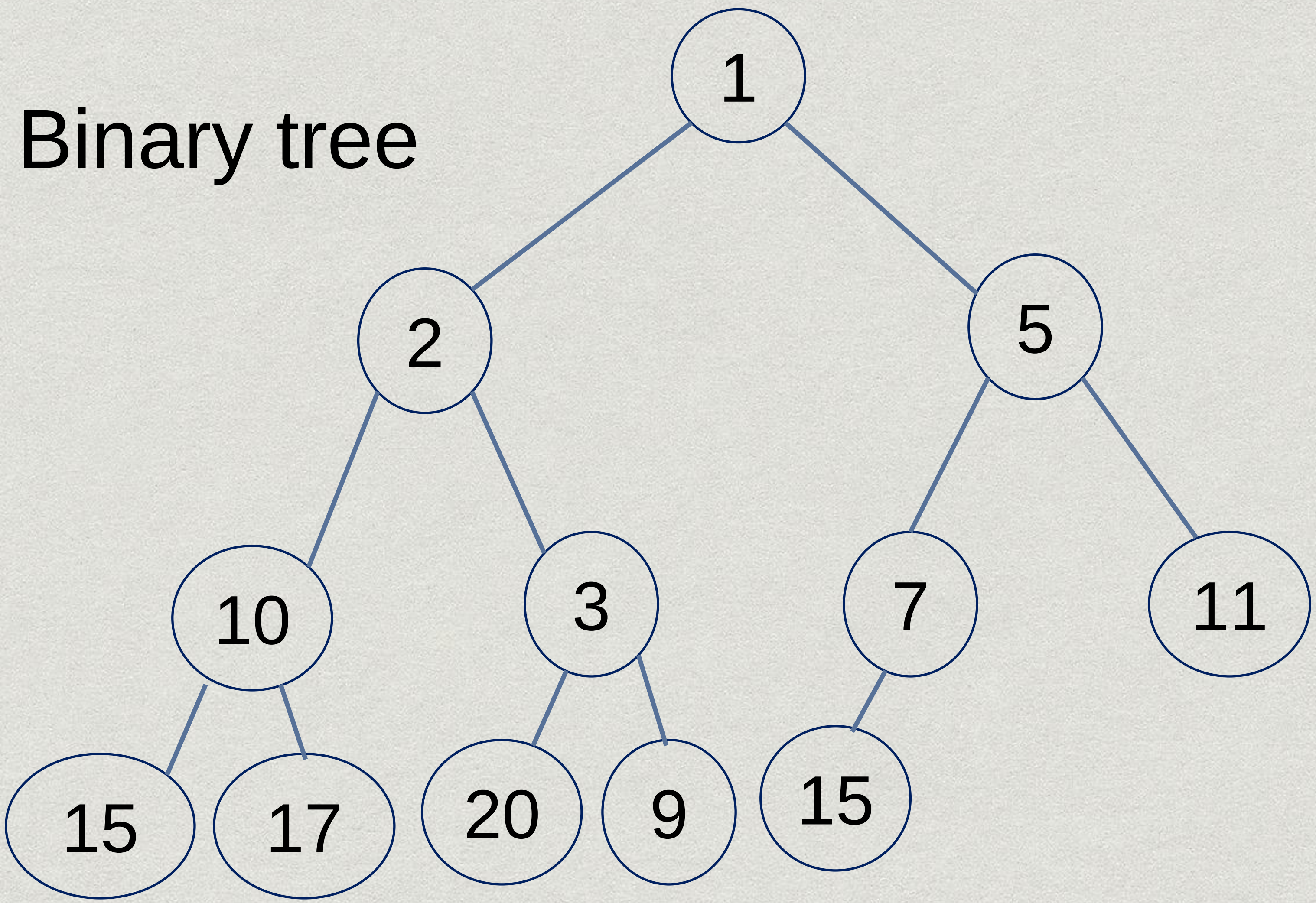


Heap data structure

Combines the benefits of a sorted array and a list

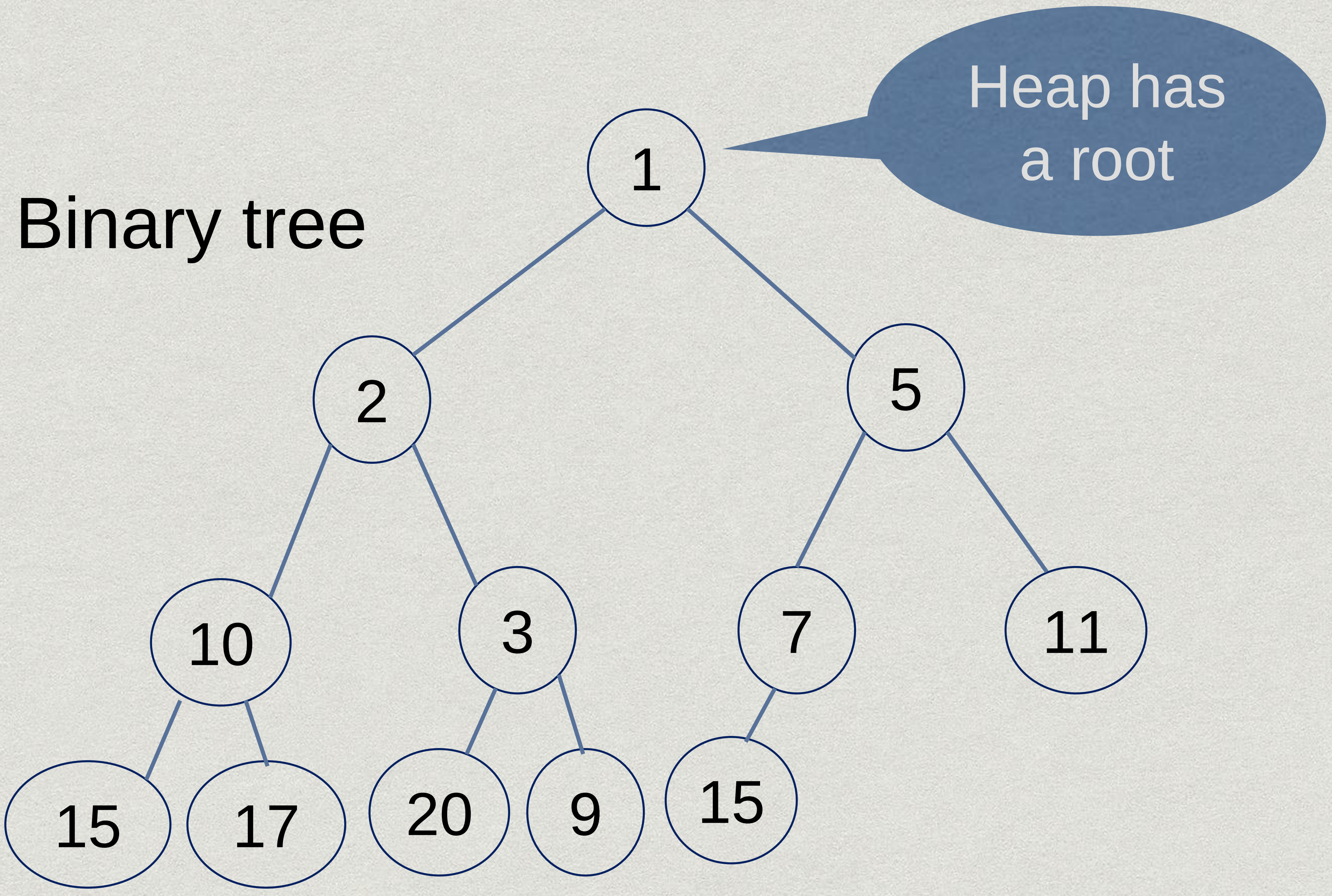
- List is good for insert, delete
- Sorted array is good for finding the min

Binary tree

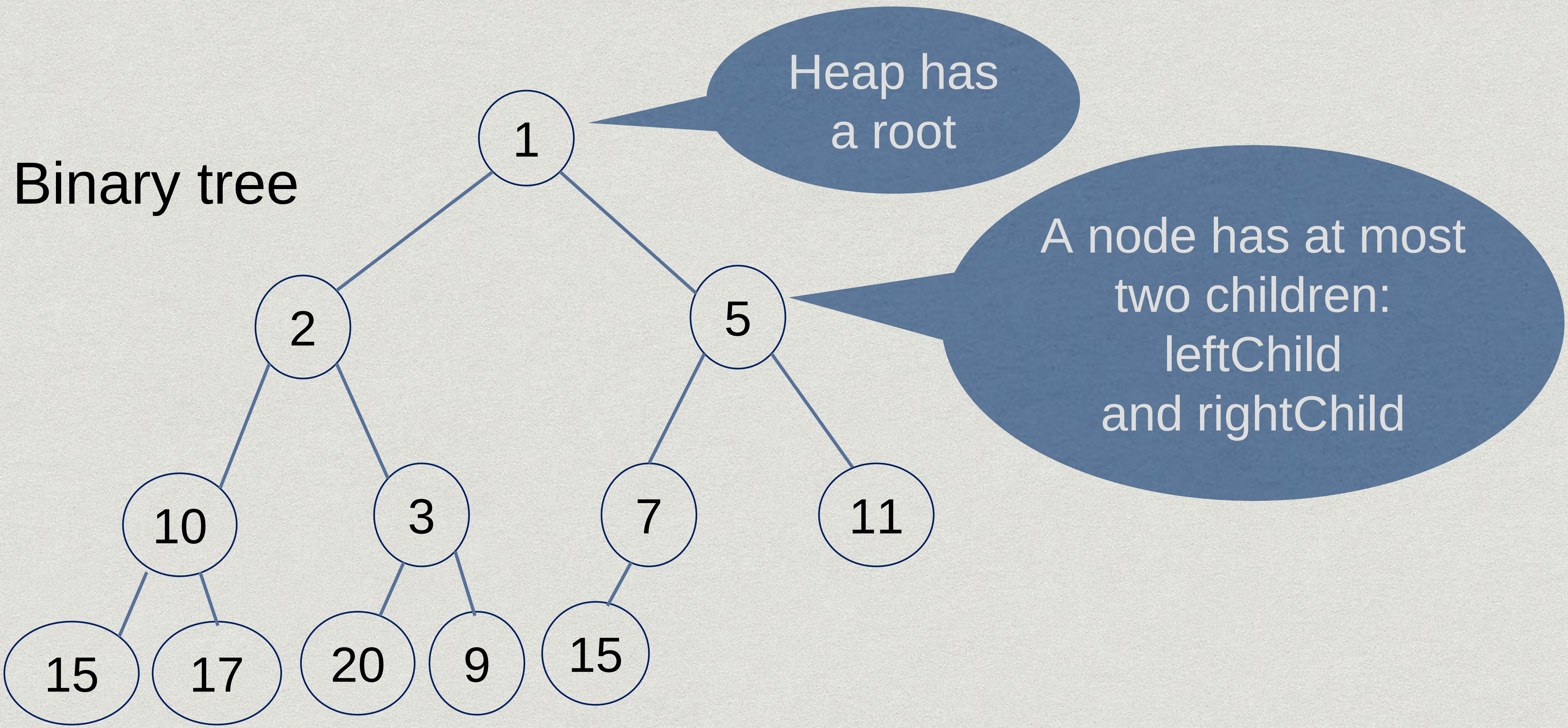


Heap data structure

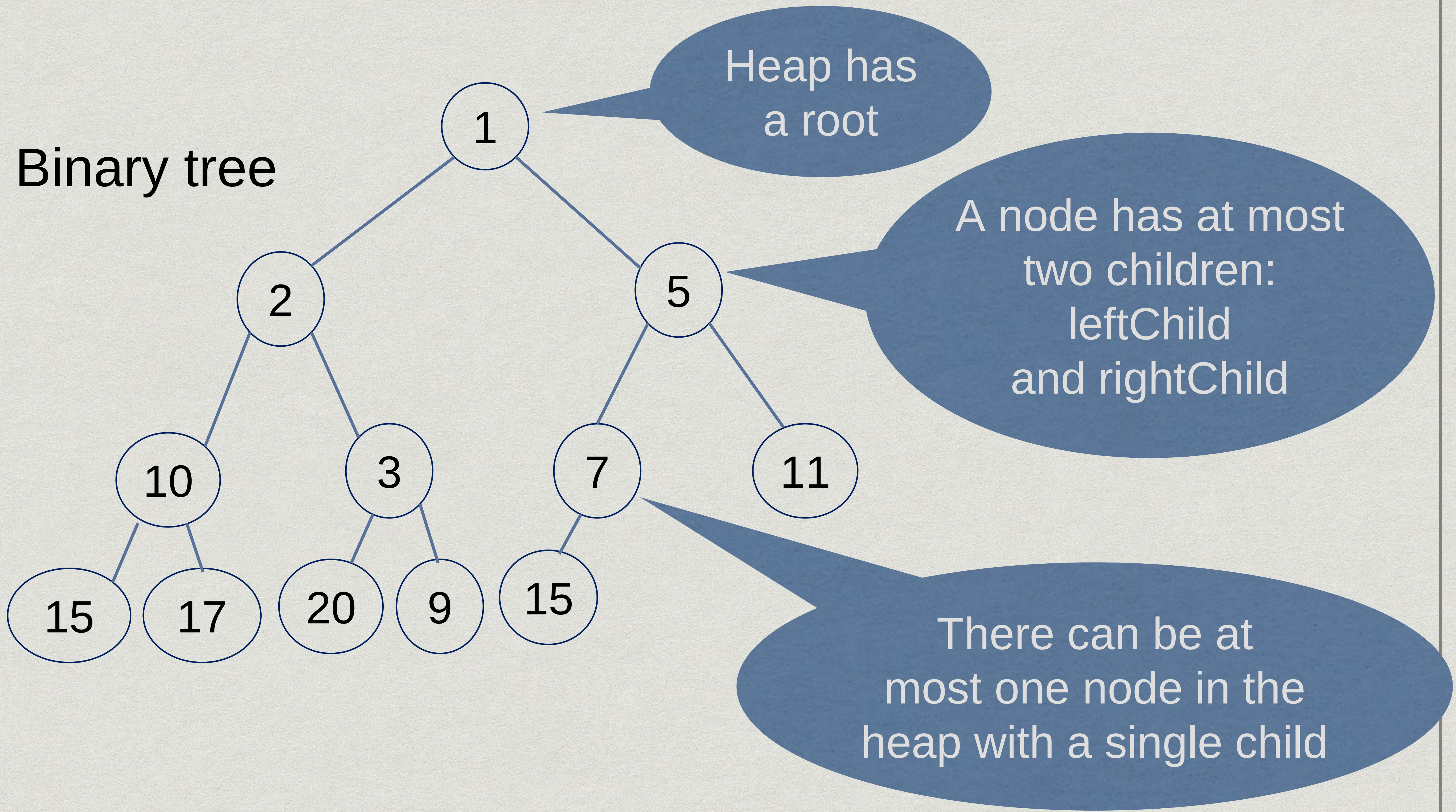
Binary tree



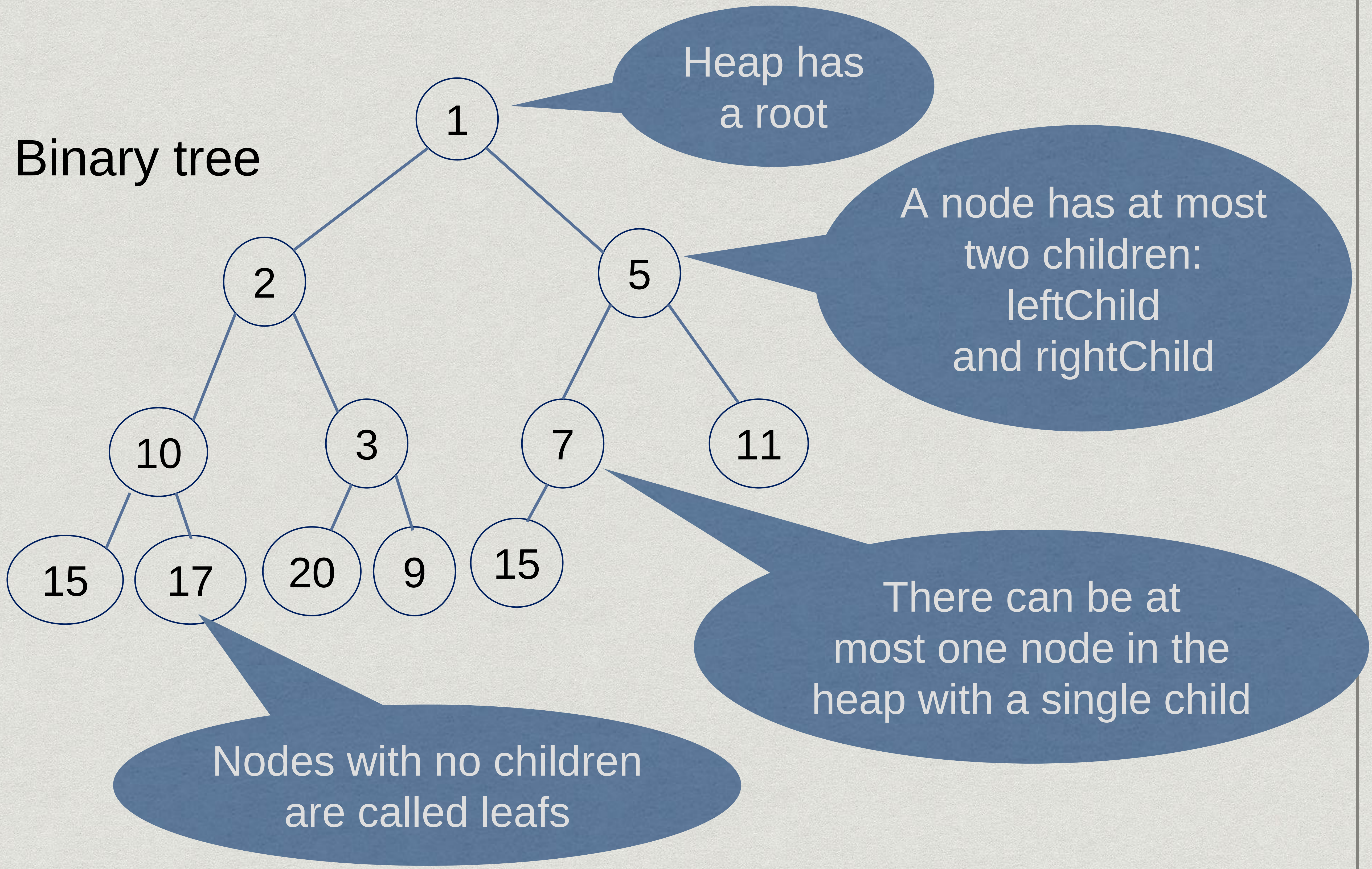
Heap data structure



Heap data structure

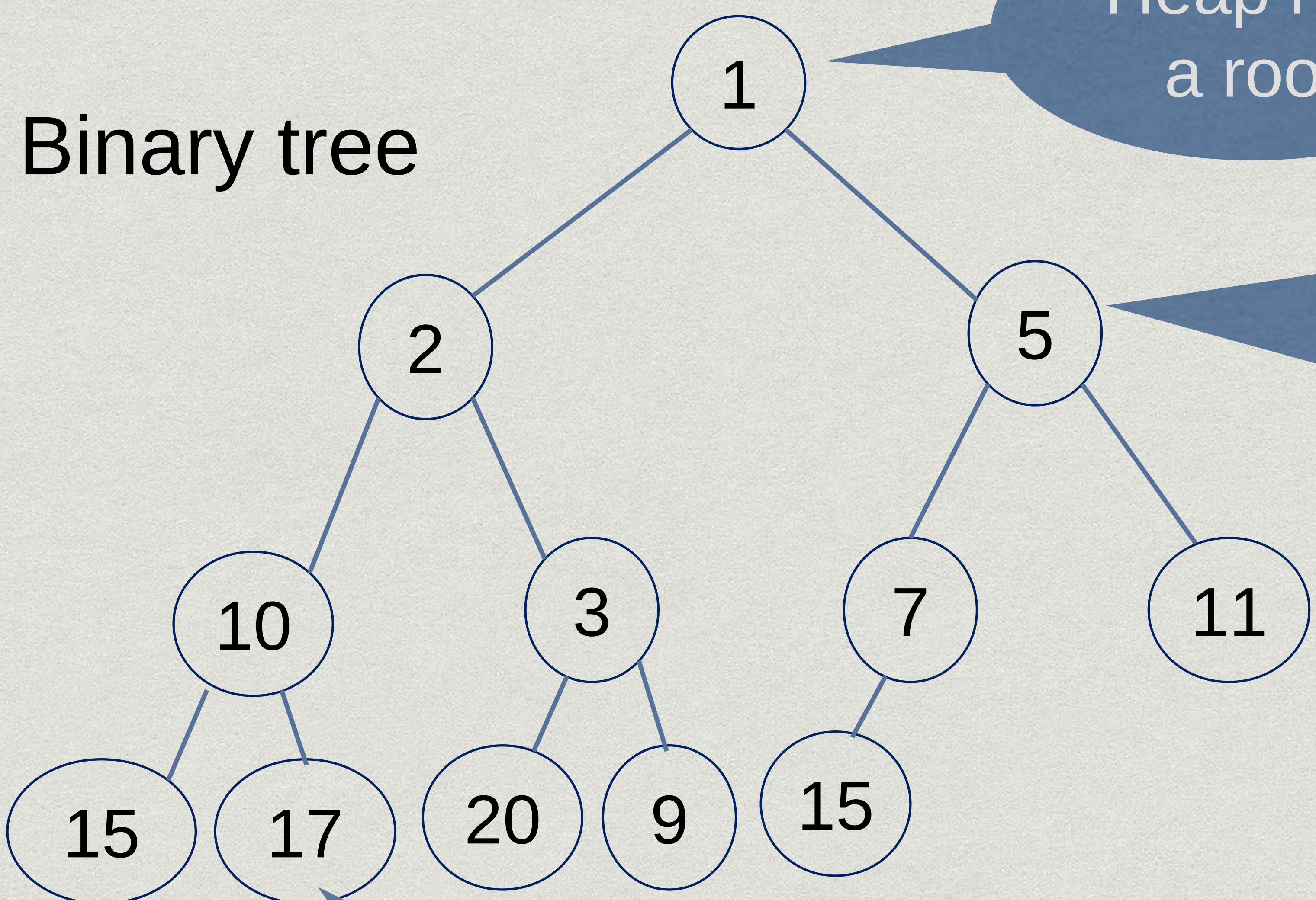


Heap data structure



Heap data structure

Binary tree



Heap has a root

A node has at most two children: leftChild and rightChild

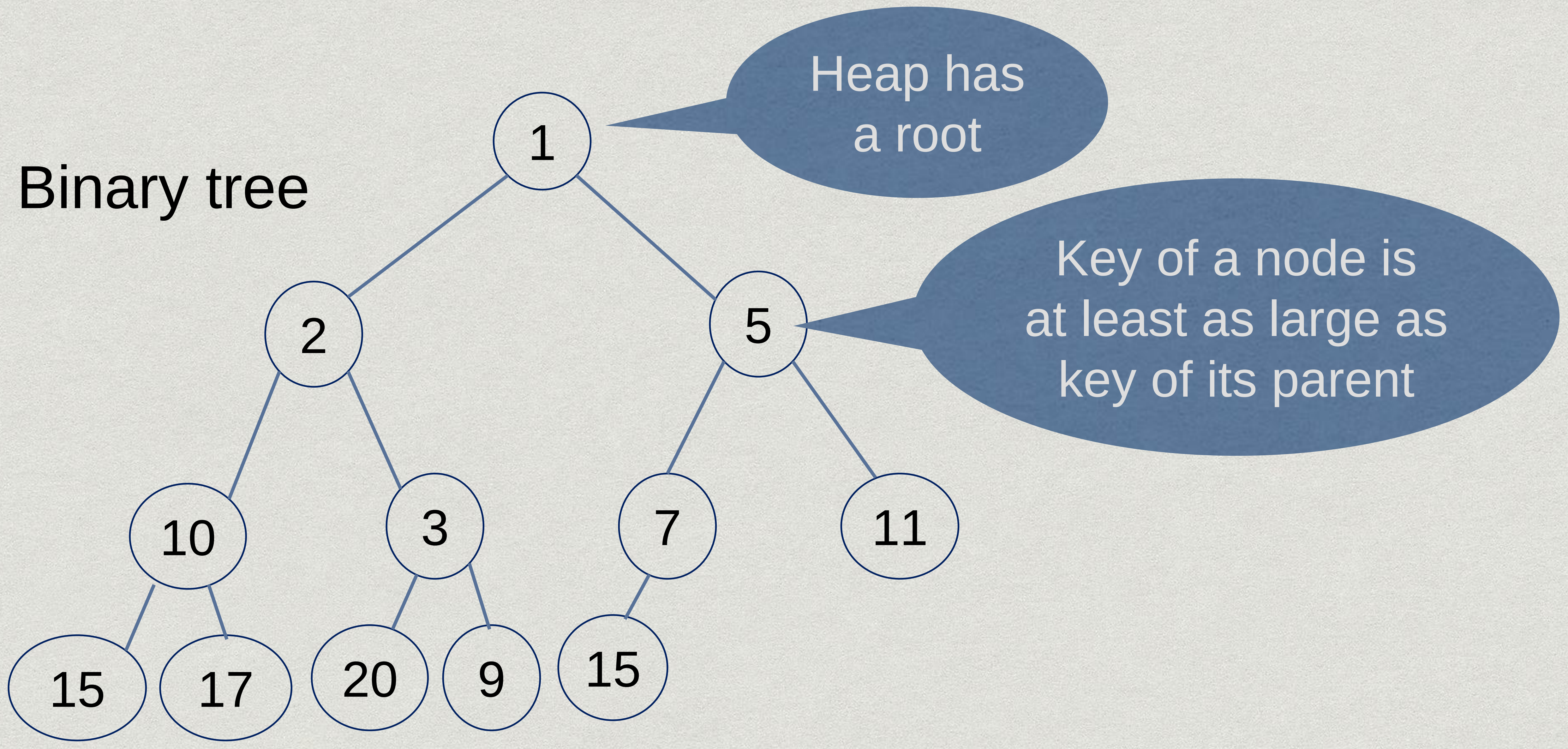
Nodes with no children are called leafs

The number of nodes in the heap is called the length of the heap

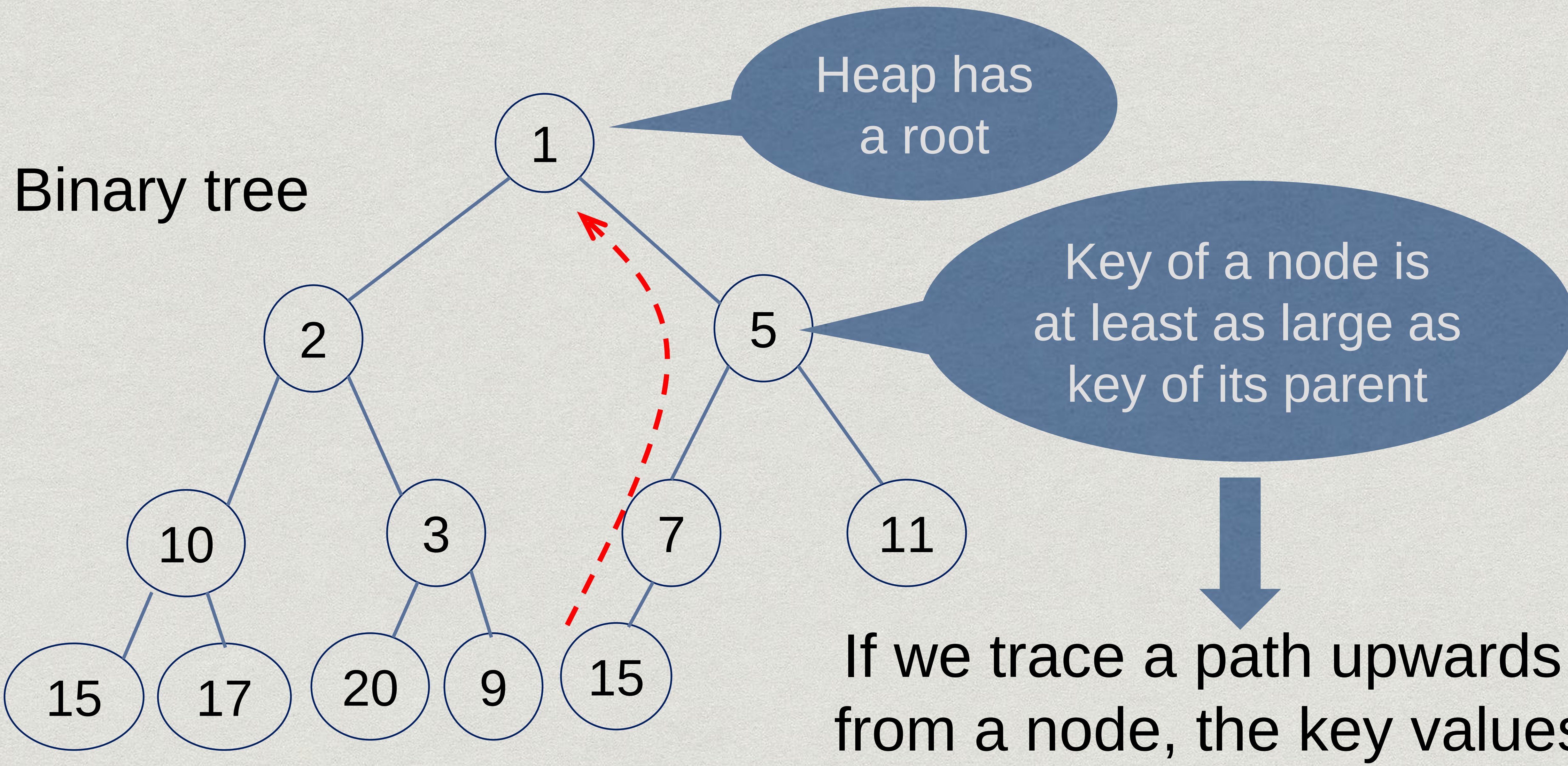
$n = 12$

Heap data structure

Binary tree



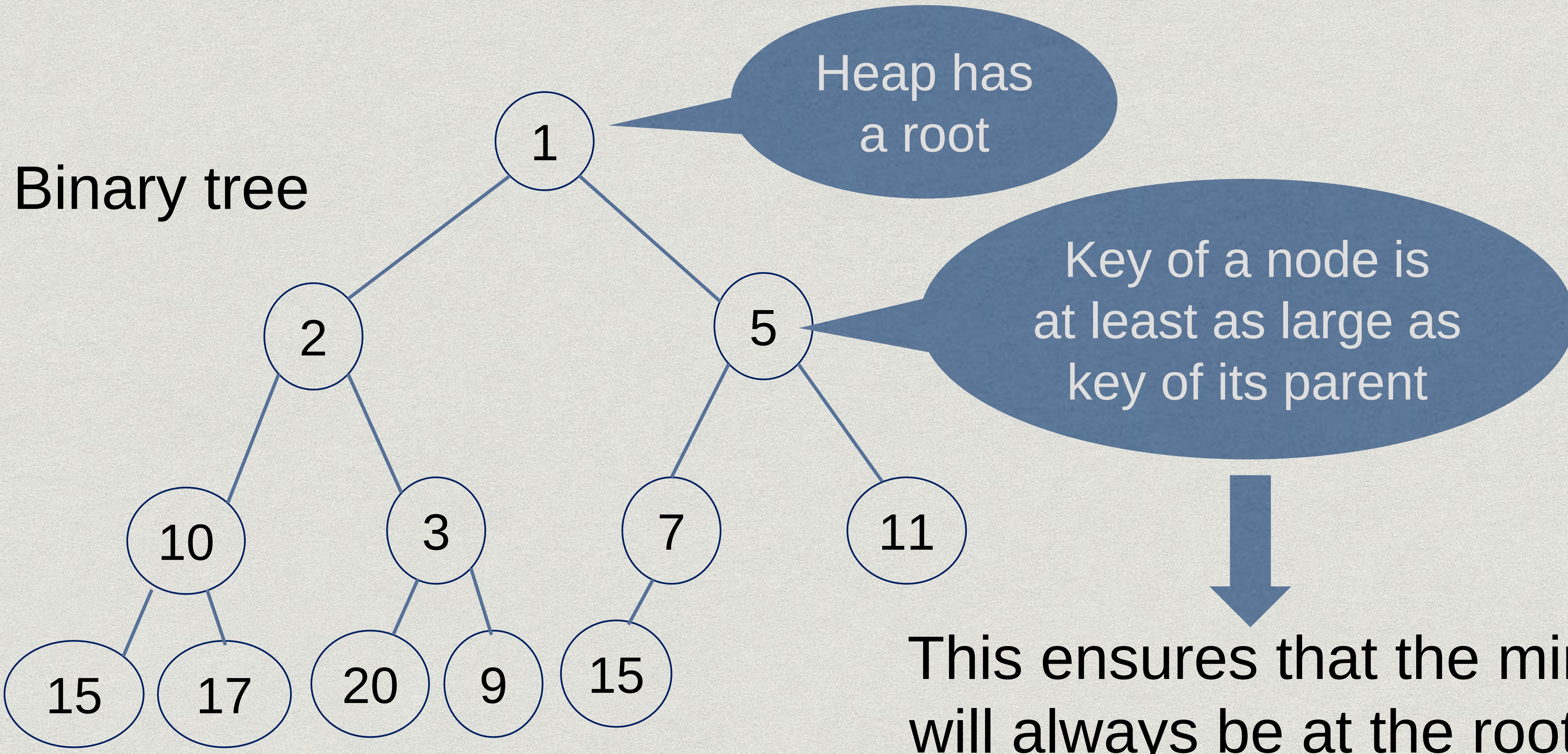
Heap data structure



If we trace a path upwards from a node, the key values will always decrease along it

Heap data structure

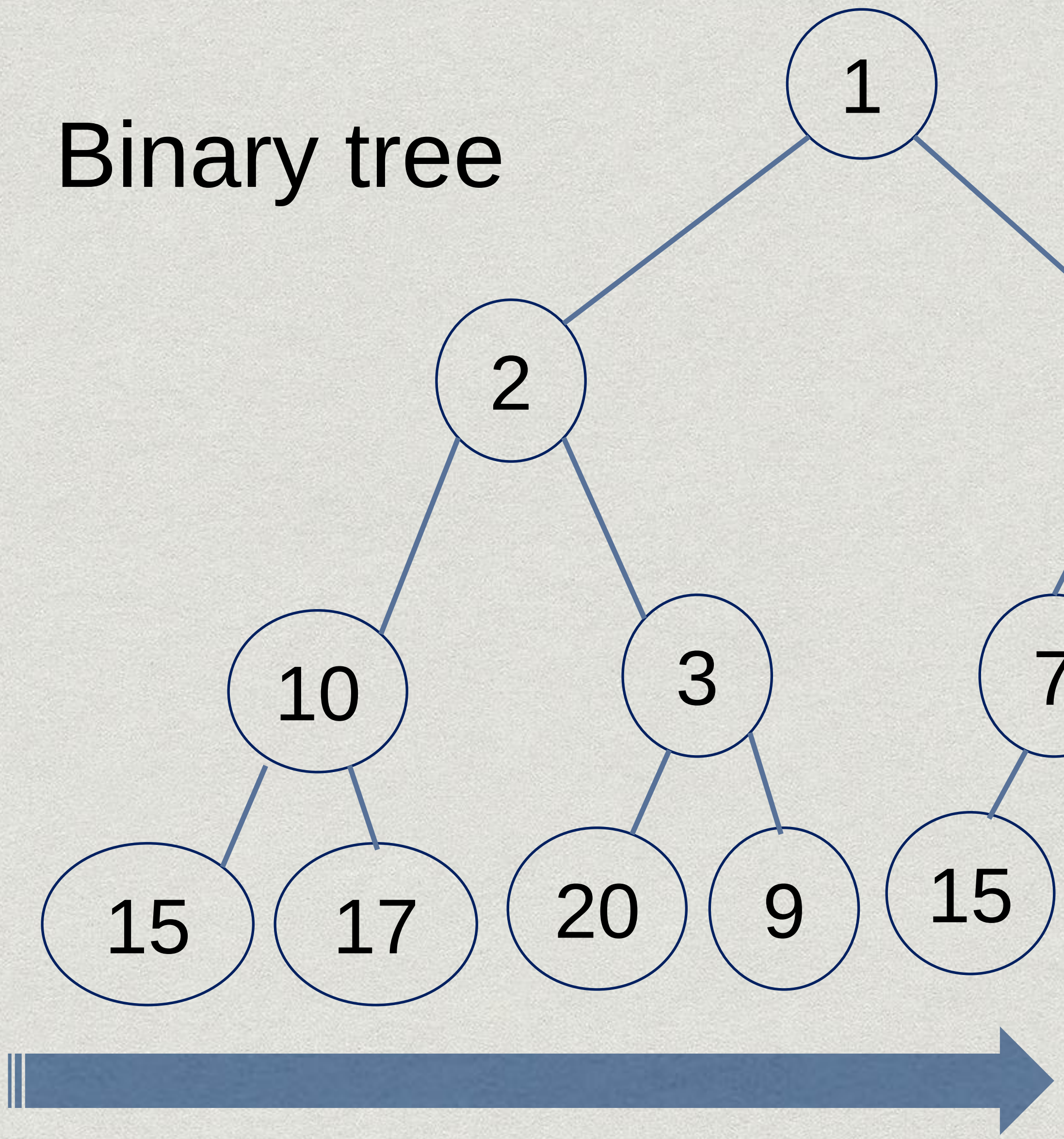
Binary tree



This ensures that the min will always be at the root !

Heap data structure

Binary tree

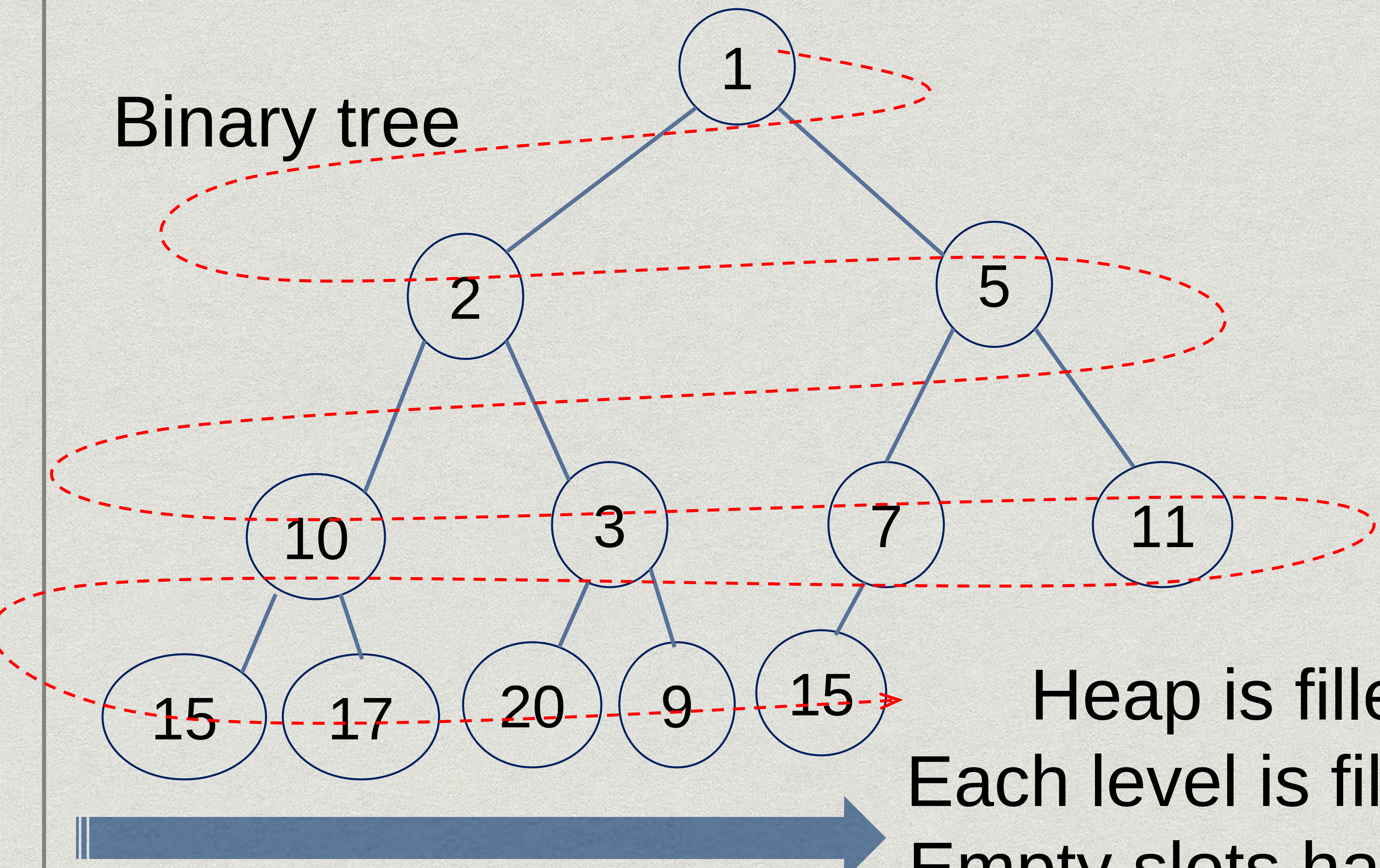


Heap is filled level by level

Each level is filled from left to right

Empty slots have to be at the right

Heap data structure



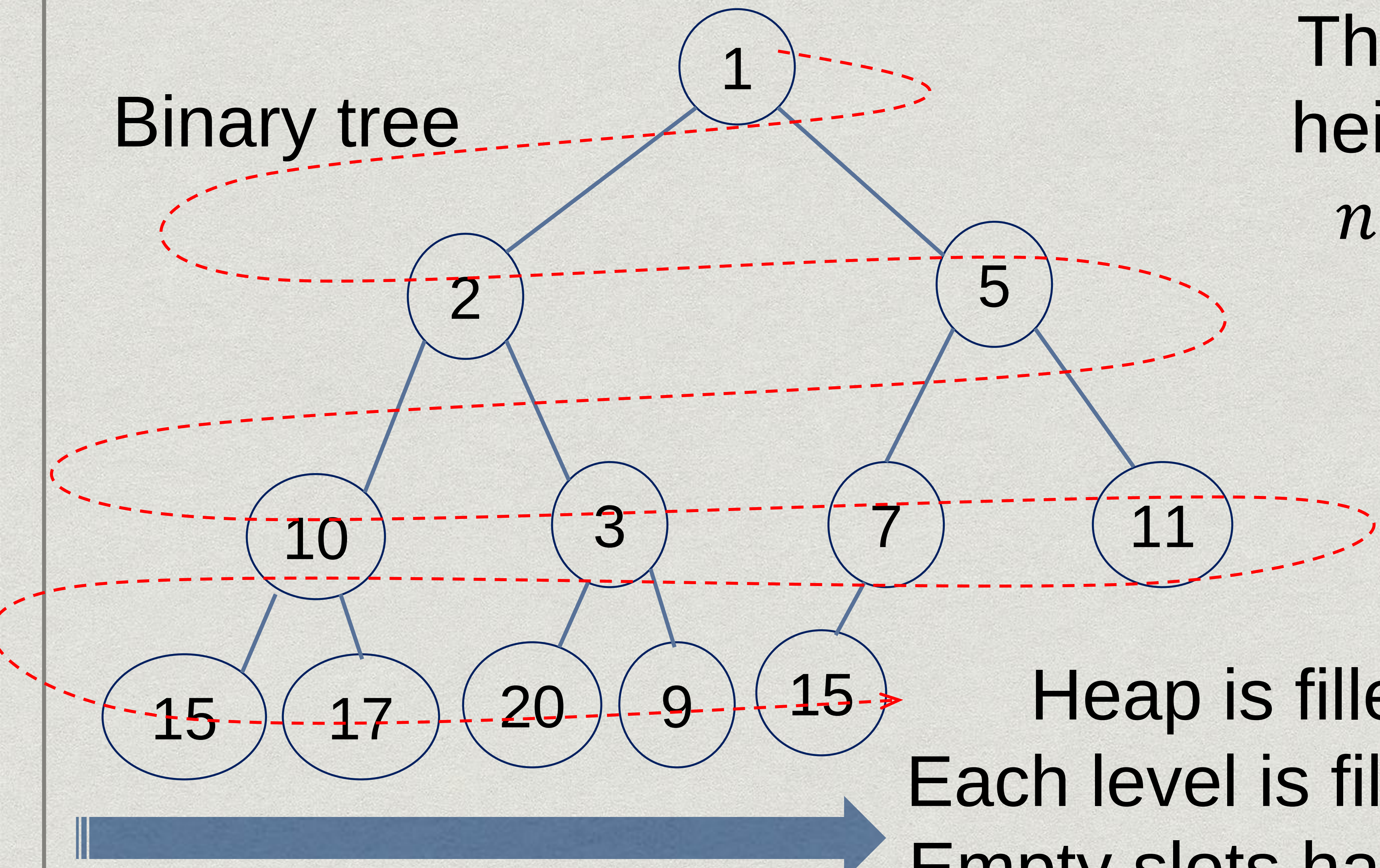
Heap is filled level by level

Each level is filled from left to right

Empty slots have to be at the right

Heap data structure

Binary tree



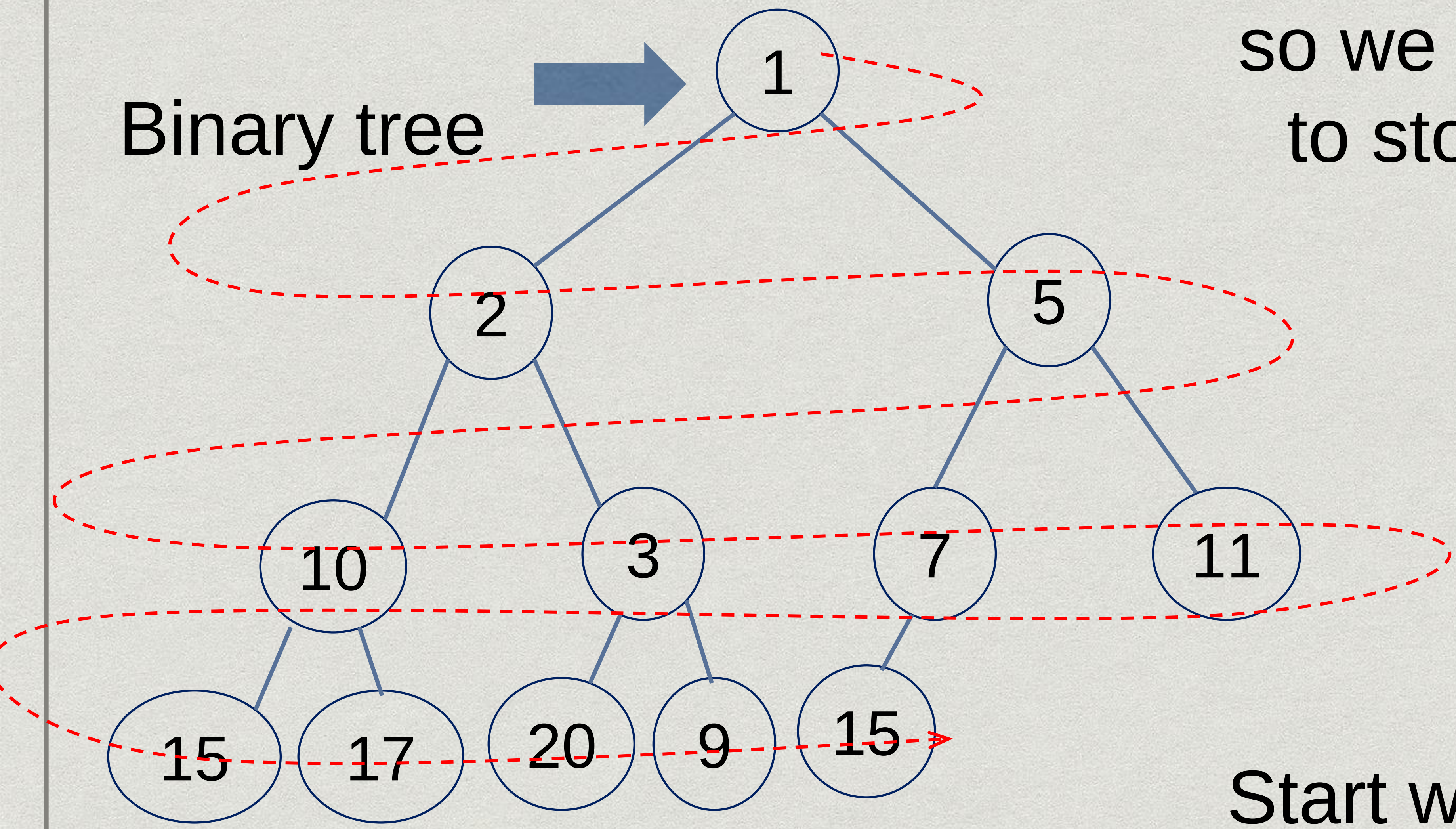
This ensures that the height of a heap with n elements will be $O(\log n)$

Heap is filled level by level
Each level is filled from left to right
Empty slots have to be at the right

Height is the length of the longest path from a leaf to the root

Heap data structure

Binary tree



Elements are contiguous, so we can use an array to store the elements

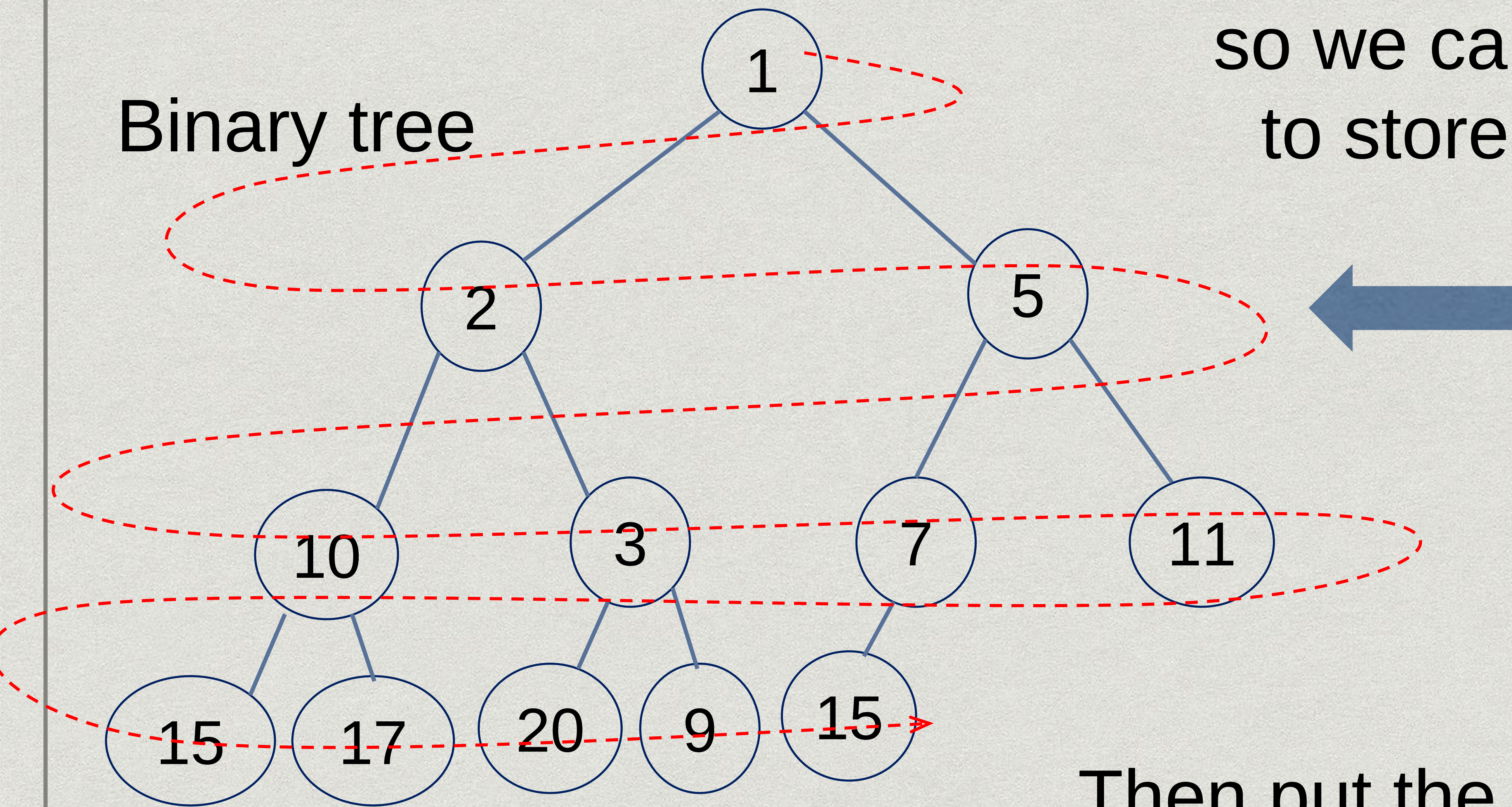
Start with the root

1													

$n = 12$

Heap data structure

Binary tree



Elements are contiguous, so we can use an array to store the elements

Then put the children of root

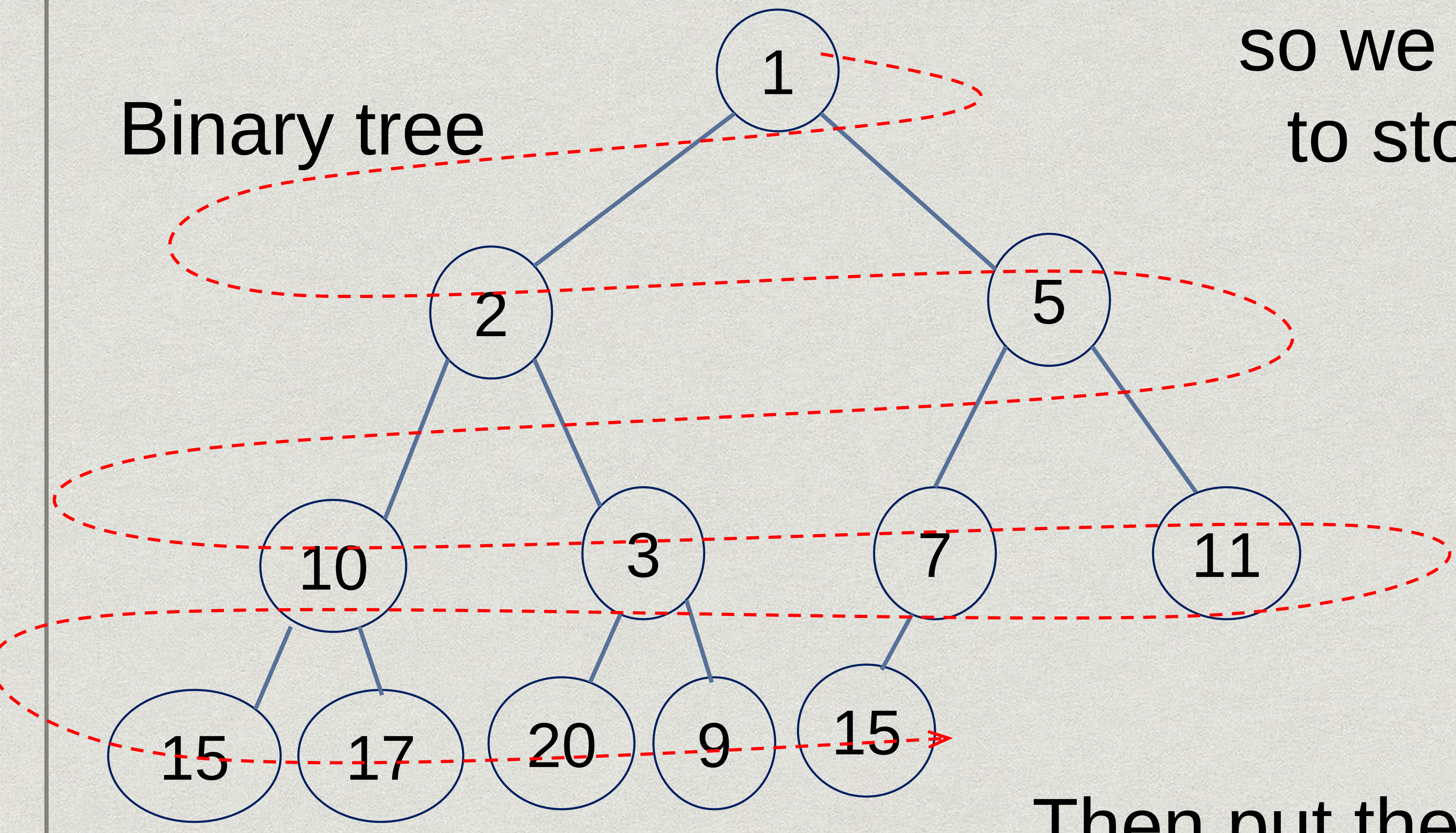
1	2	5											

$n = 12$

Heap data structure

Elements are contiguous, so we can use an array to store the elements

Binary tree

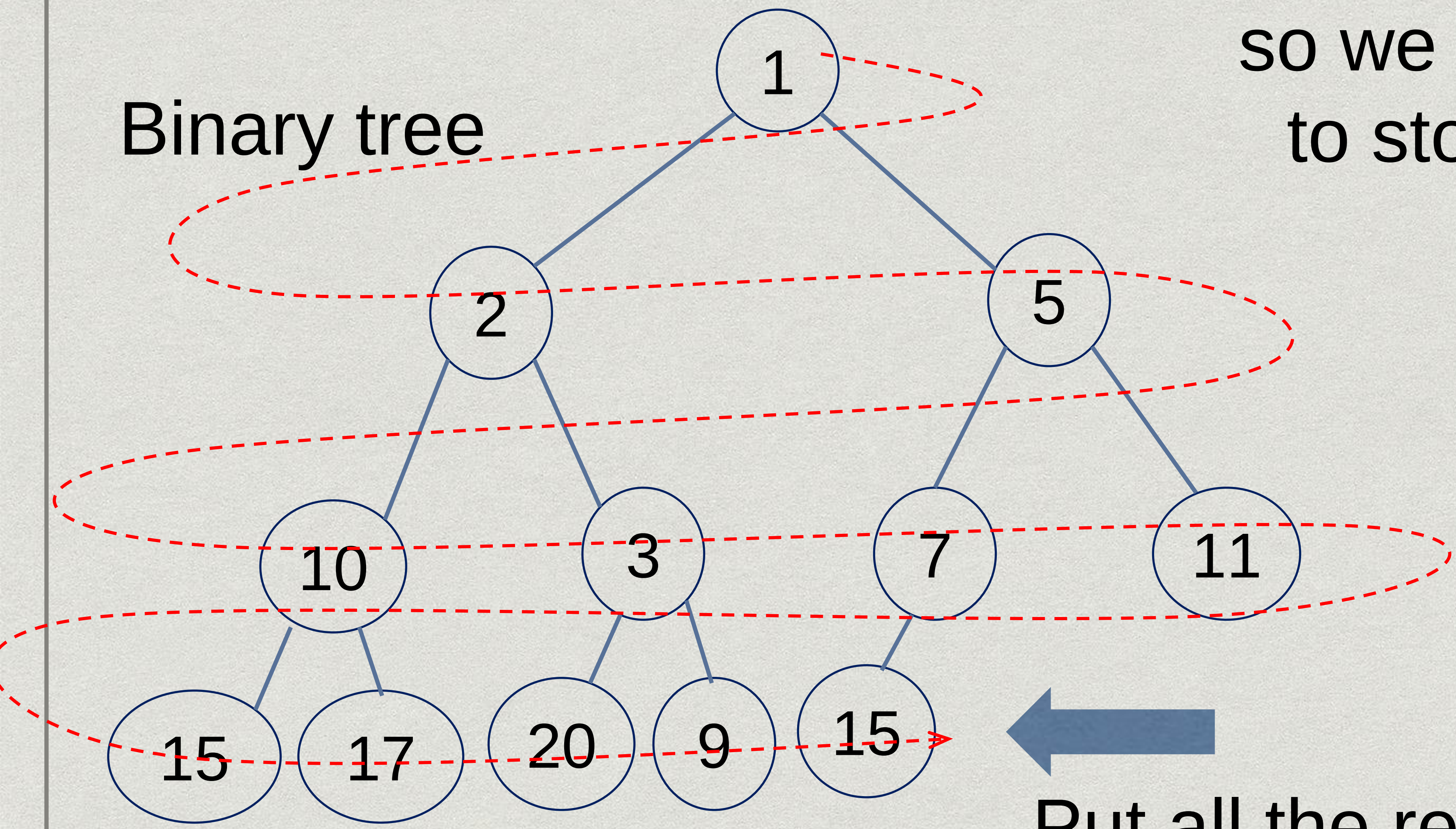


Then put the next level children

1	2	5	10	3	7	11							

$n = 12$

Heap data structure



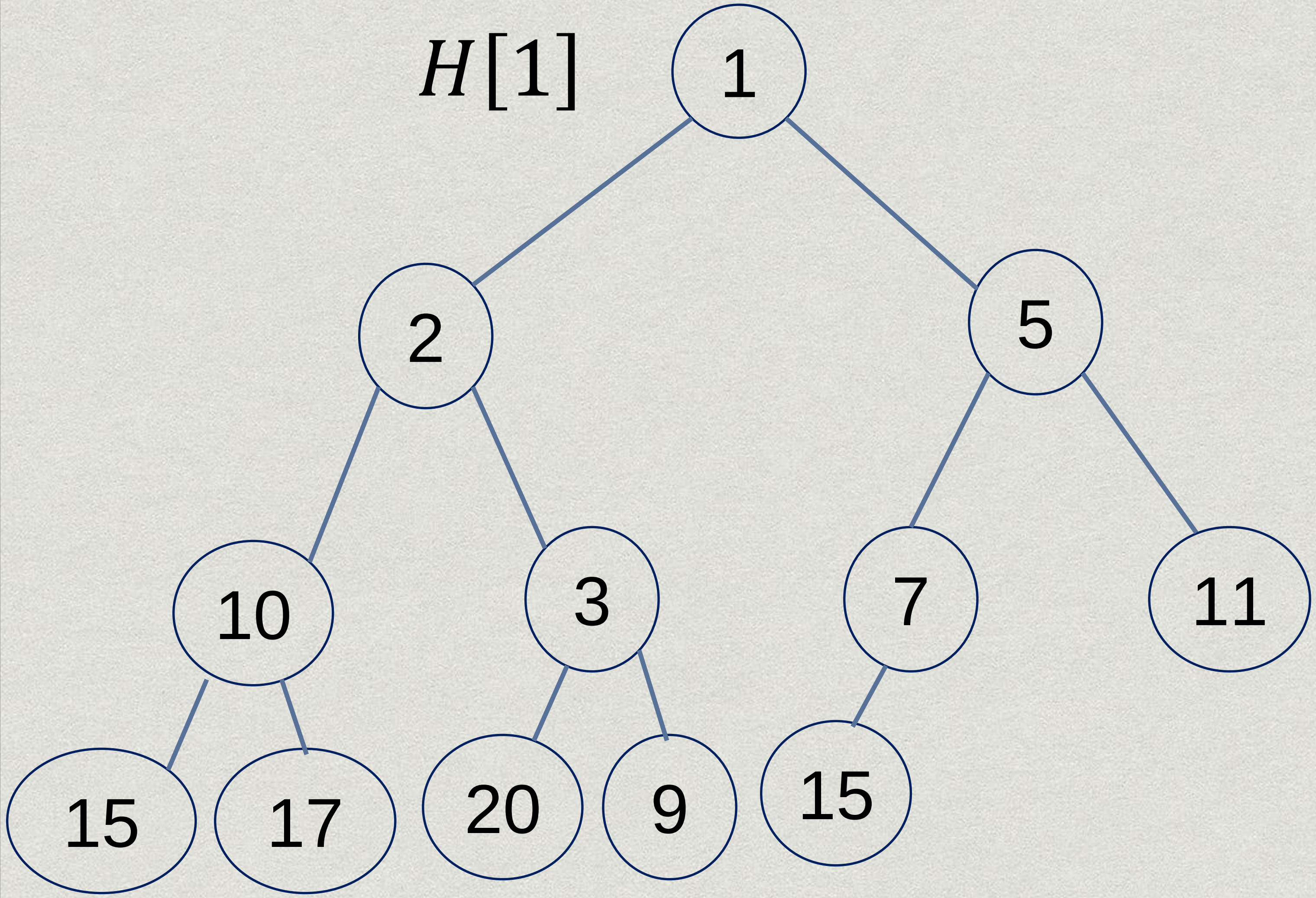
Elements are contiguous, so we can use an array to store the elements

Put all the remaining elements

1	2	5	10	3	7	11	15	17	20	9	15		

$n = 12$

Heap data structure



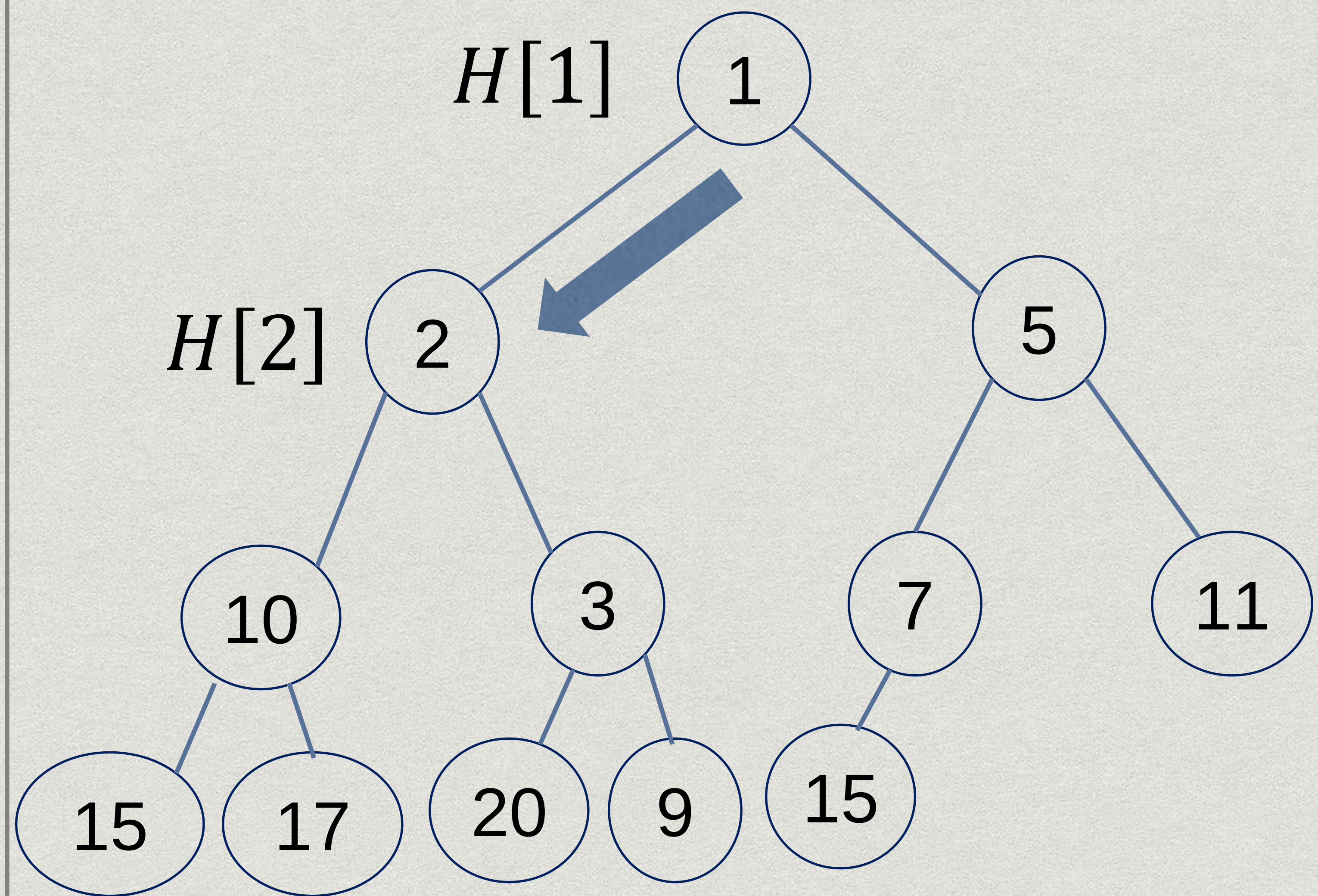
Use array $H[1..12]$ to store the elements

Start with $H[1]$ for convenience

1	2	5	10	3	7	11	15	17	20	9	15		

$n = 12$

Heap data structure



Use array $H[1..12]$
to store the elements

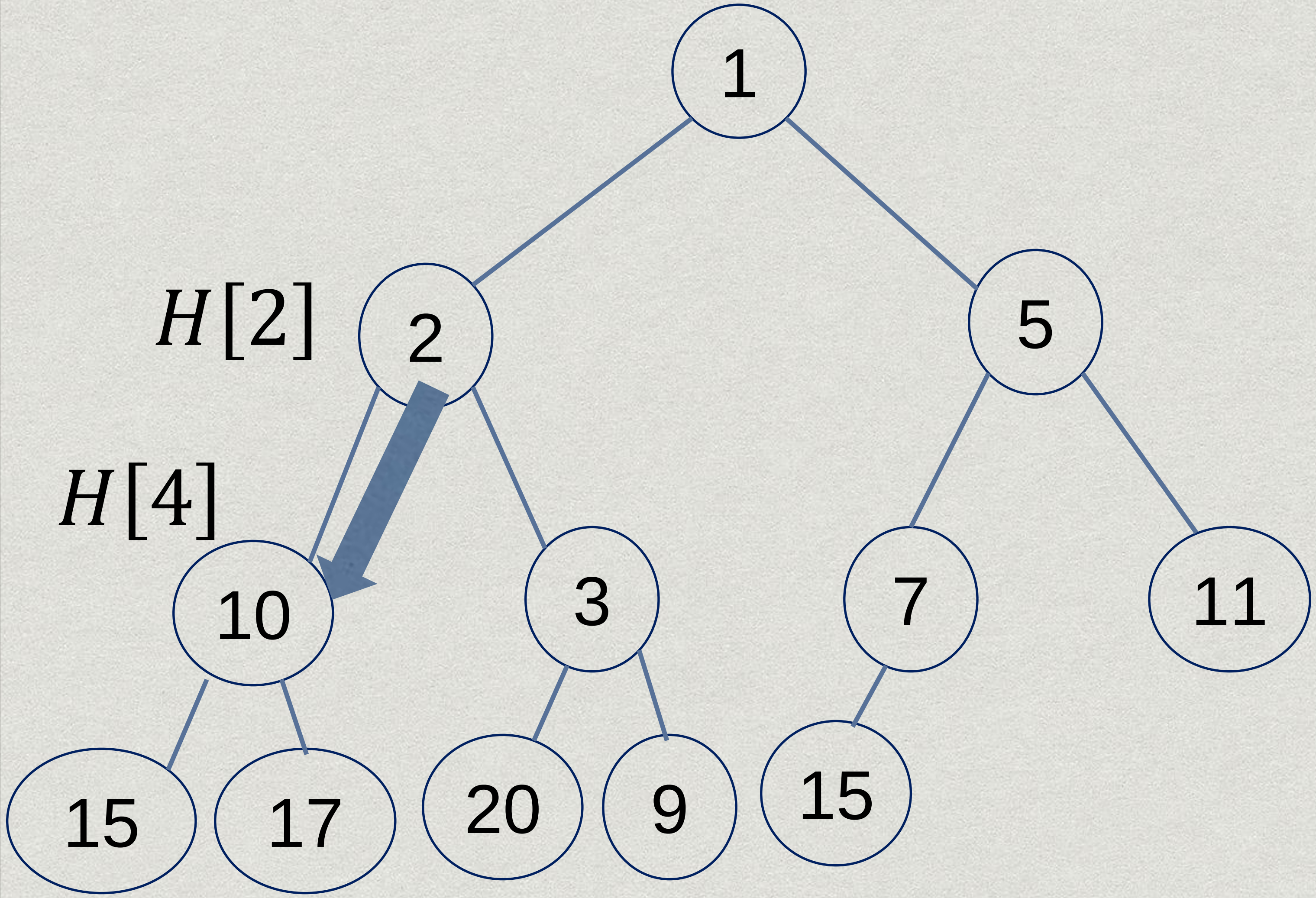
Start with $H[1]$
for convenience

$$leftChild(i) = 2i$$

1	2	5	10	3	7	11	15	17	20	9	15		

$$leftChild(1) = 2$$

Heap data structure



Use array $H[1..12]$
to store the elements

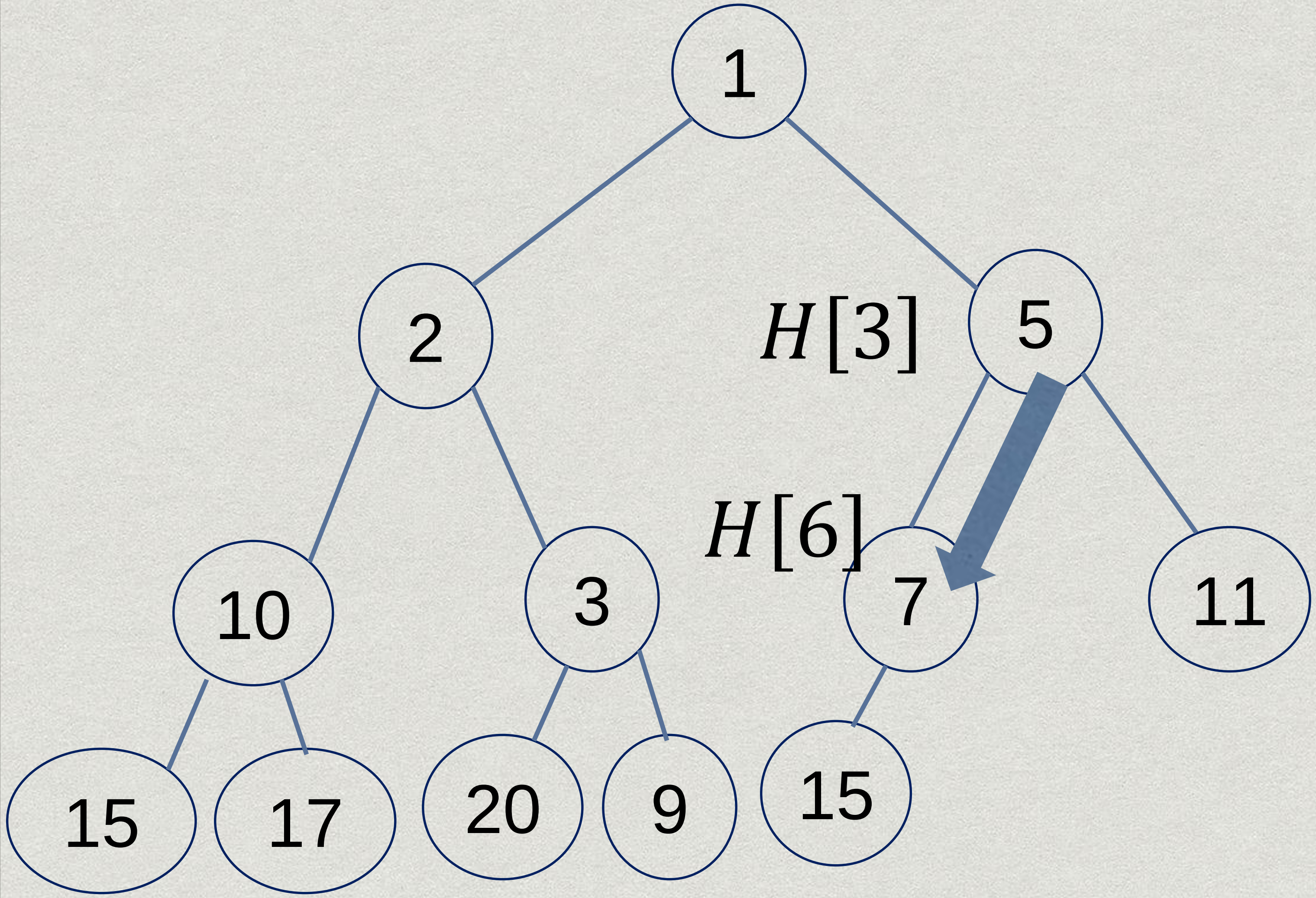
Start with $H[1]$
for convenience

$$leftChild(i) = 2i$$

1	2	5	10	3	7	11	15	17	20	9	15		

$$leftChild(2) = 4$$

Heap data structure



Use array $H[1..12]$ to store the elements

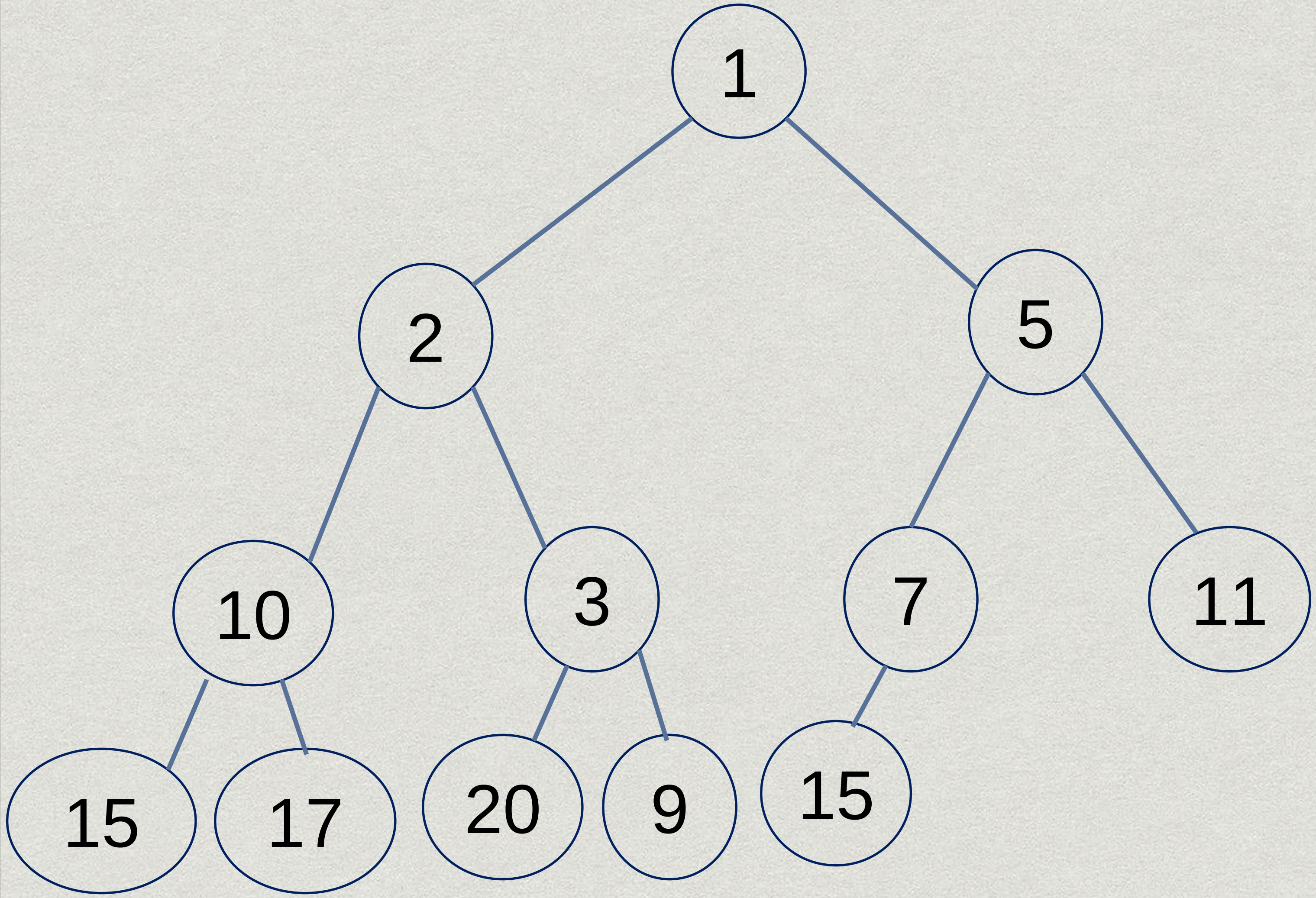
Start with $H[1]$ for convenience

$$leftChild(i) = 2i$$

1	2	5	10	3	7	11	15	17	20	9	15		

$$leftChild(3) = 6$$

Heap data structure



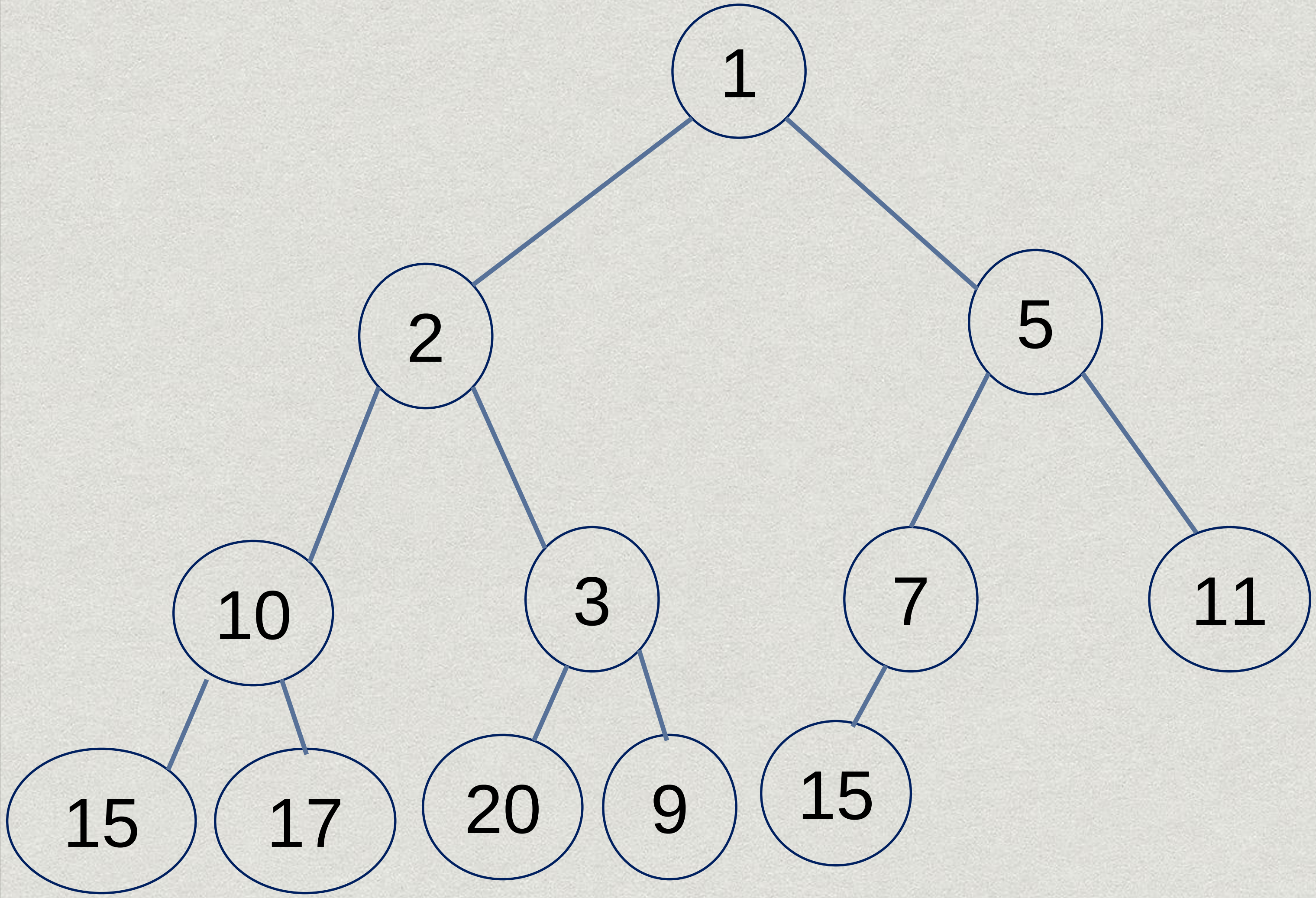
Use array $H[1..12]$
to store the elements

Start with $H[1]$
for convenience

$$leftChild(i) = 2i$$

1	2	5	10	3	7	11	15	17	20	9	15		

Heap data structure



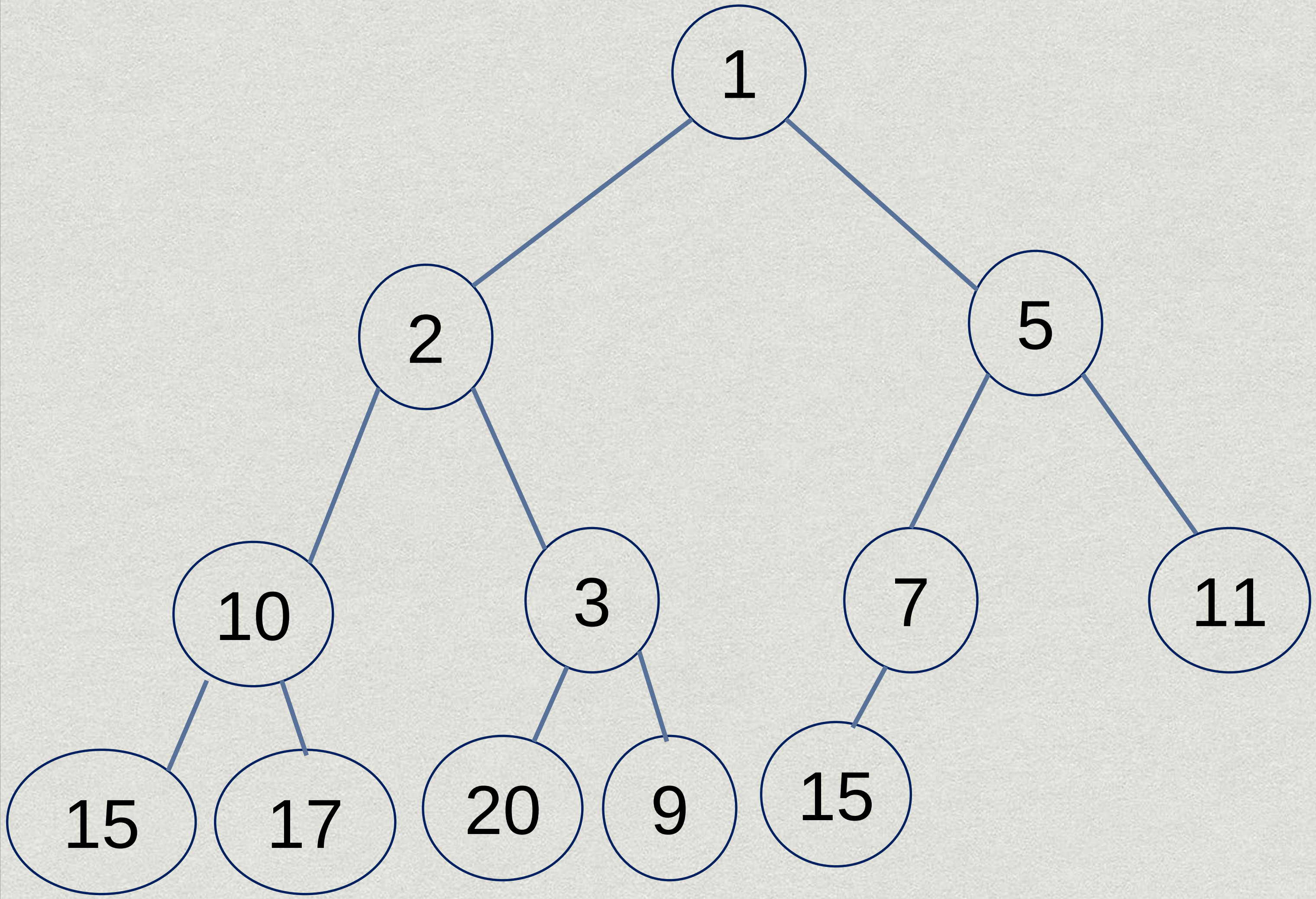
Use array $H[1..12]$
to store the elements

Start with $H[1]$
for convenience

$$leftChild(i) = 2i$$
$$rightChild(i) = 2i + 1$$

1	2	5	10	3	7	11	15	17	20	9	15		

Heap data structure



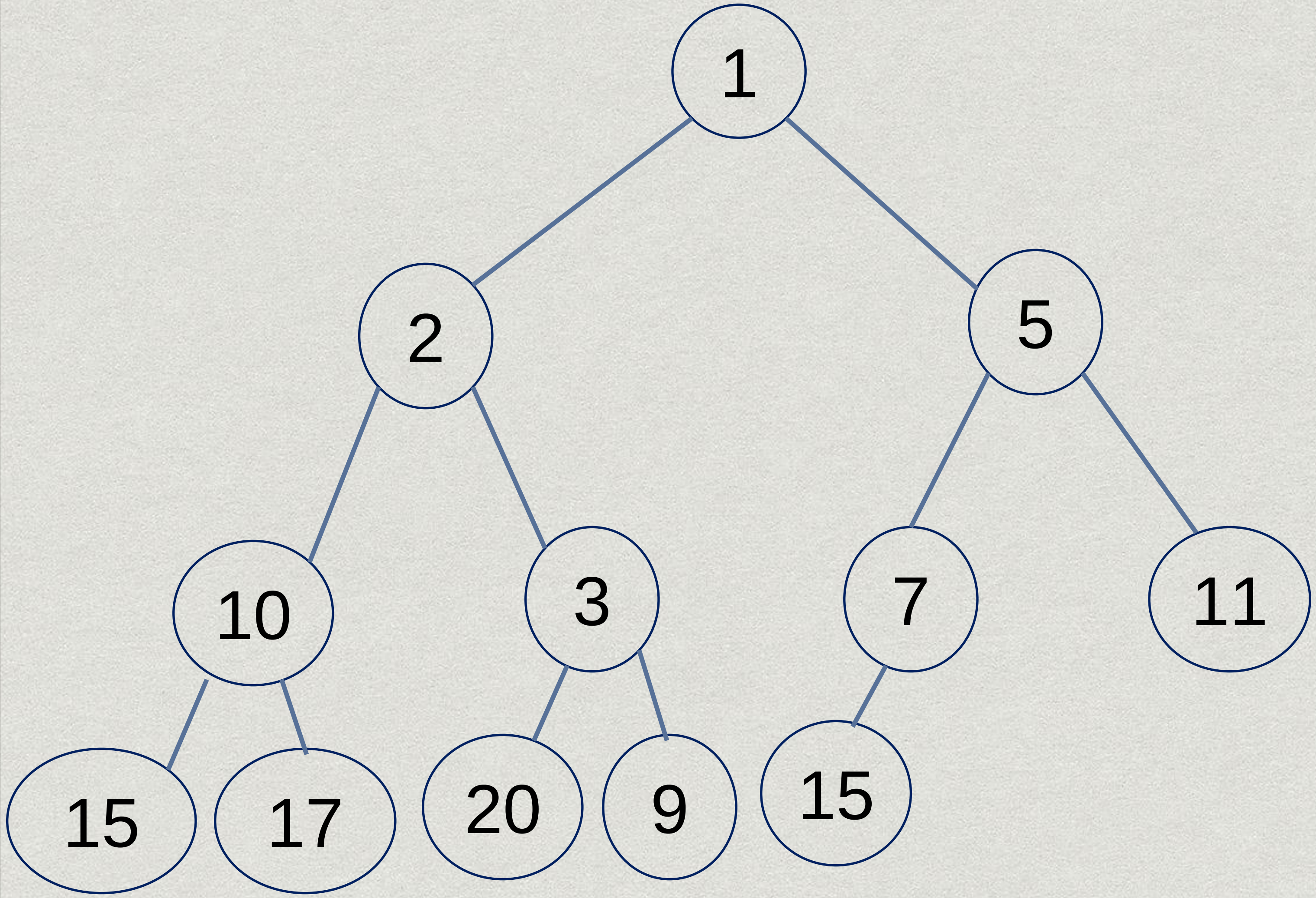
Use array $H[1\,..\,12]$ to store the elements

Start with $H[1]$ for convenience

$$leftChild(i) = 2i$$
$$rightChild(i) = 2i + 1$$
$$parent(i) = \lfloor i/2 \rfloor$$

1	2	5	10	3	7	11	15	17	20	9	15		

Heap data structure



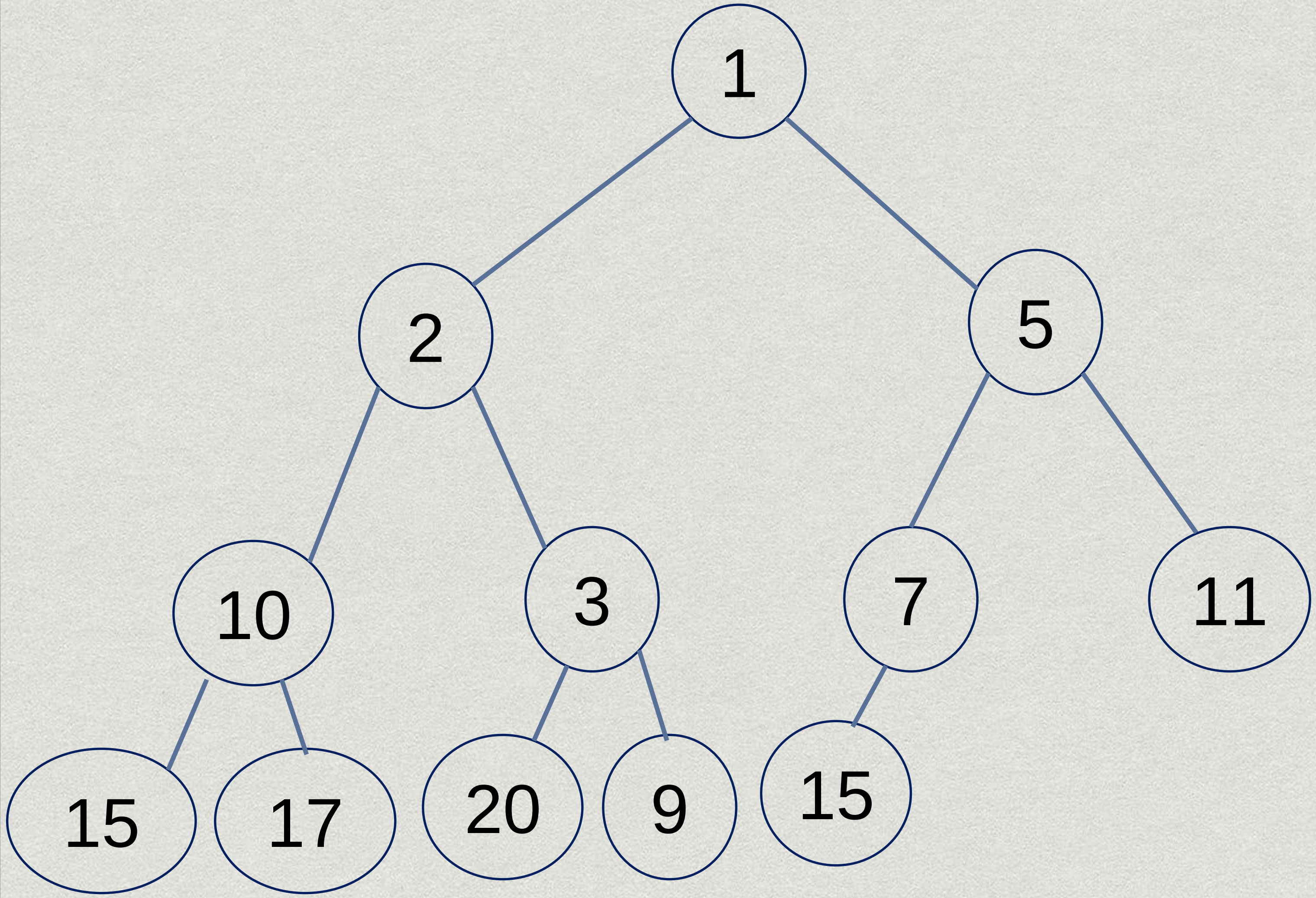
Use array $H[1..12]$
to store the elements

Start with $H[1]$
for convenience

$$leftChild(i) = 2i$$
$$rightChild(i) = 2i + 1$$
$$parent(i) = \lfloor i/2 \rfloor$$

1	2	5	10	3	7	11	15	17	20	9	15		

Heap data structure



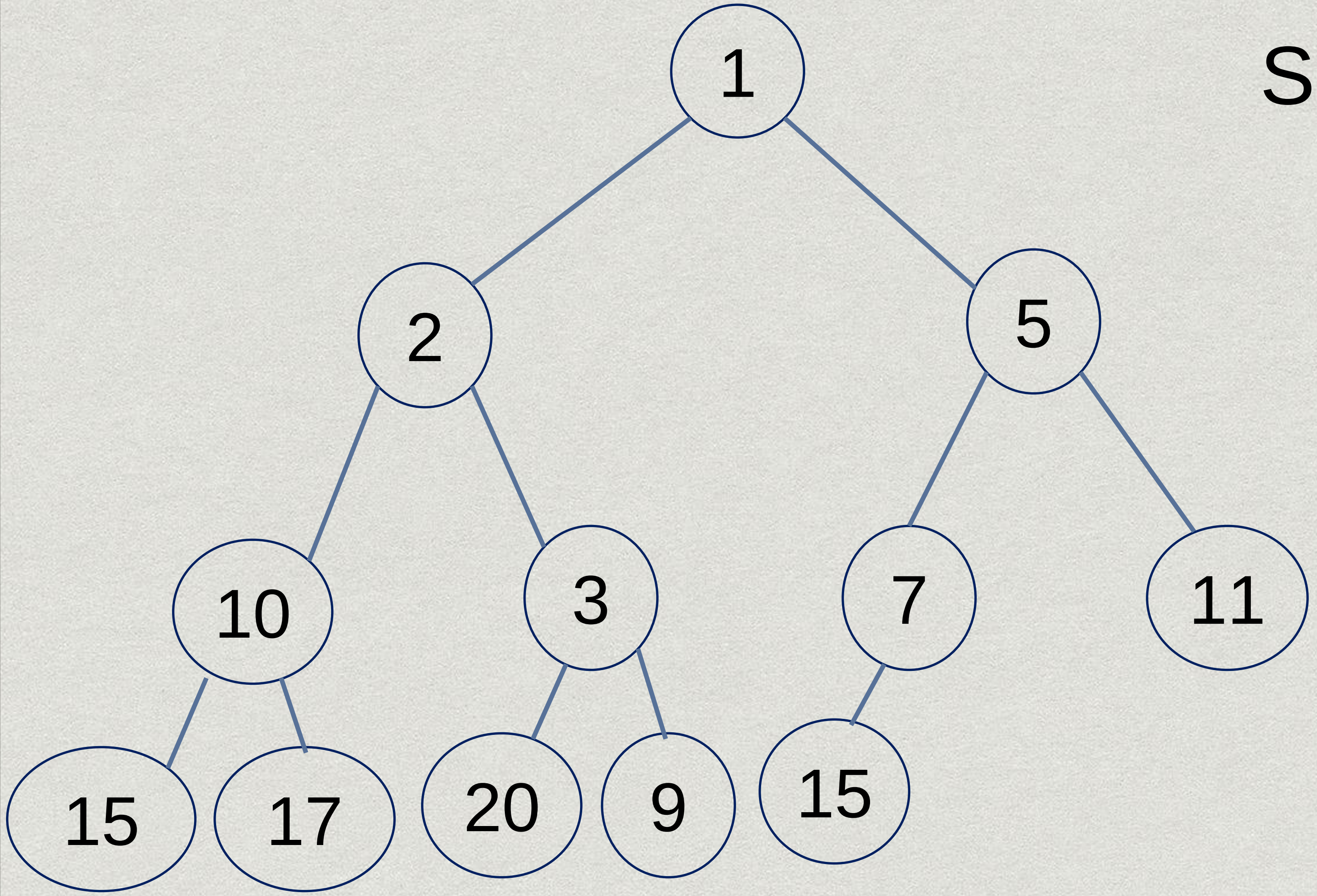
Use array $H[1..12]$
to store the elements

Start with $H[1]$
for convenience

$leftChild(i) = 2i$
$rightChild(i) = 2i + 1$
$parent(i) = \lfloor i/2 \rfloor$

1	2	5	10	3	7	11	15	17	20	9	15		

Heap operations

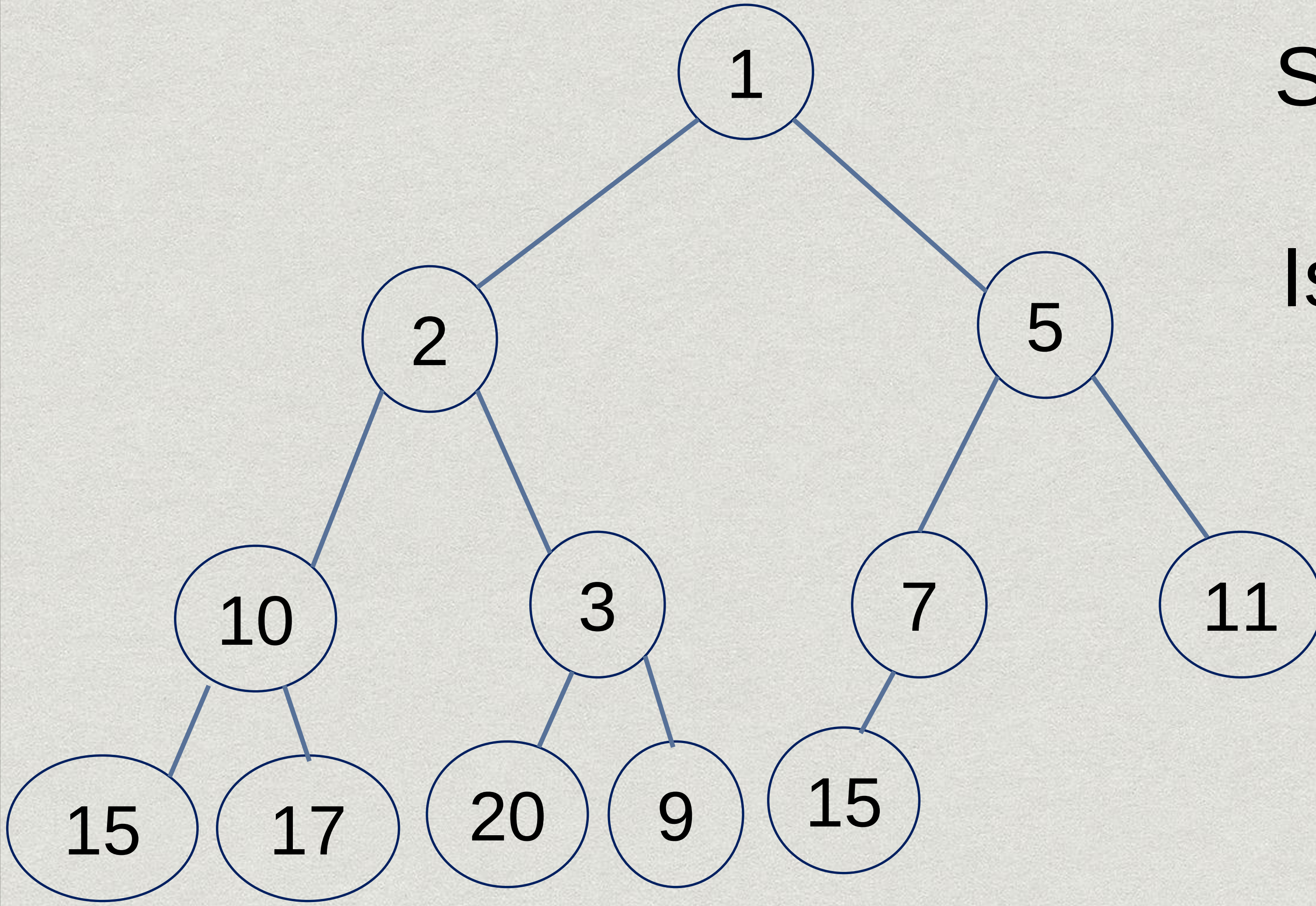


Selecting the min element

1	2	5	10	3	7	11	15	17	20	9	15		

$n = 12$

Heap operations



Selecting the min element

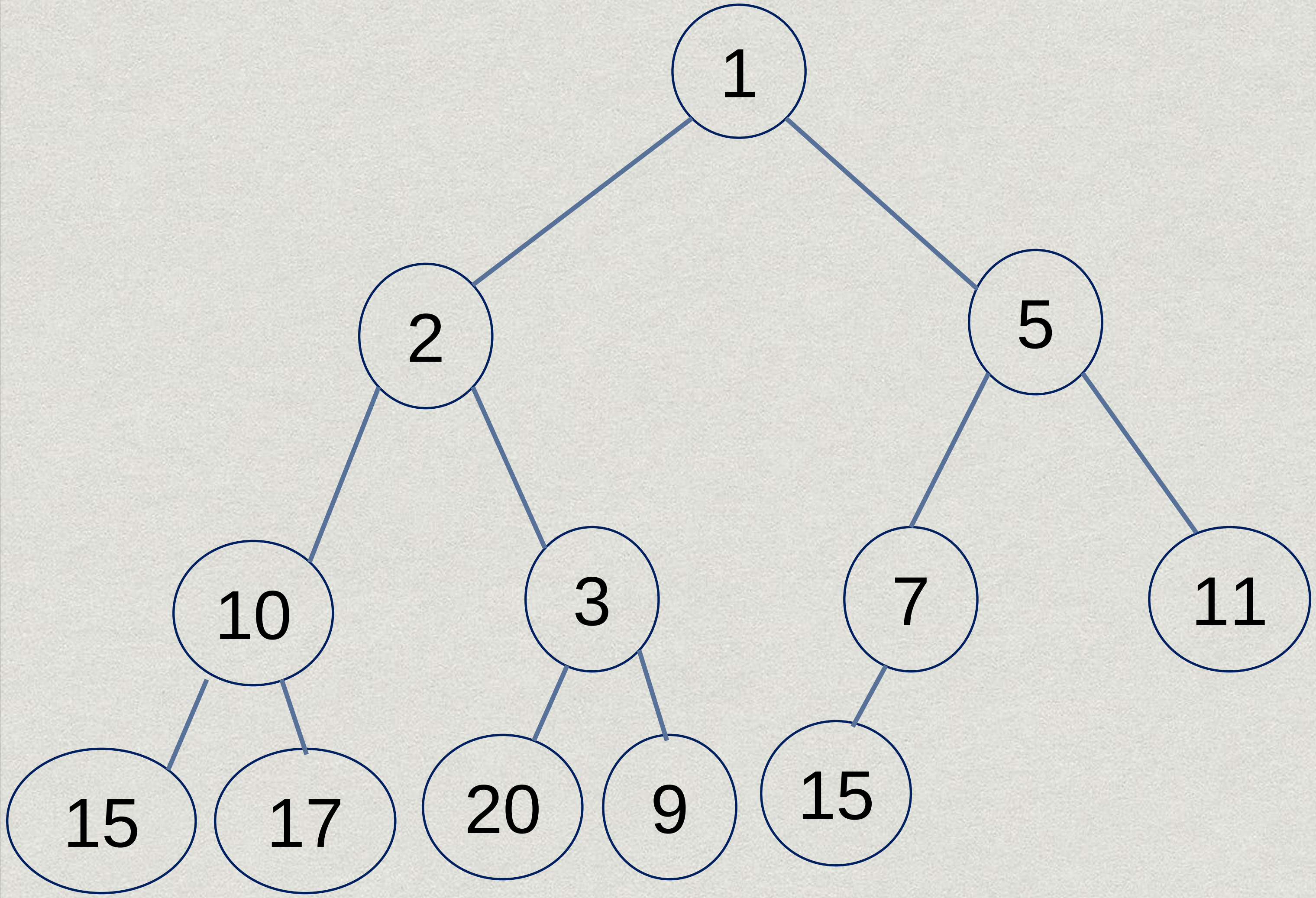
Is trivial – select the root !

1	2	5	10	3	7	11	15	17	20	9	15		

$n = 12$

Heap operations

Add an element



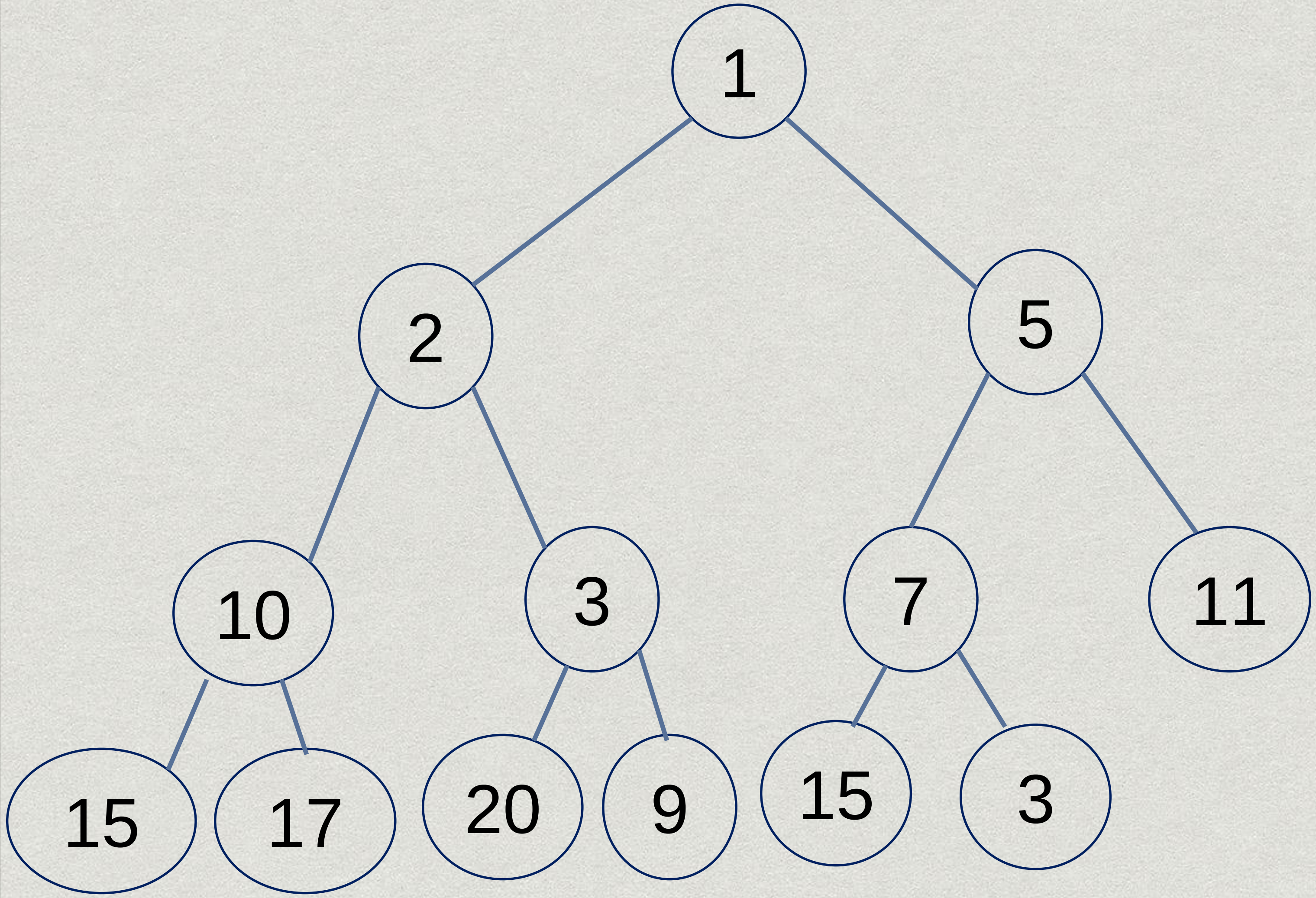
1	2	5	10	3	7	11	15	17	20	9	15		

$n = 12$

Heap operations

Add an element

Insert the element at the end of the heap



1	2	5	10	3	7	11	15	17	20	9	15	3	

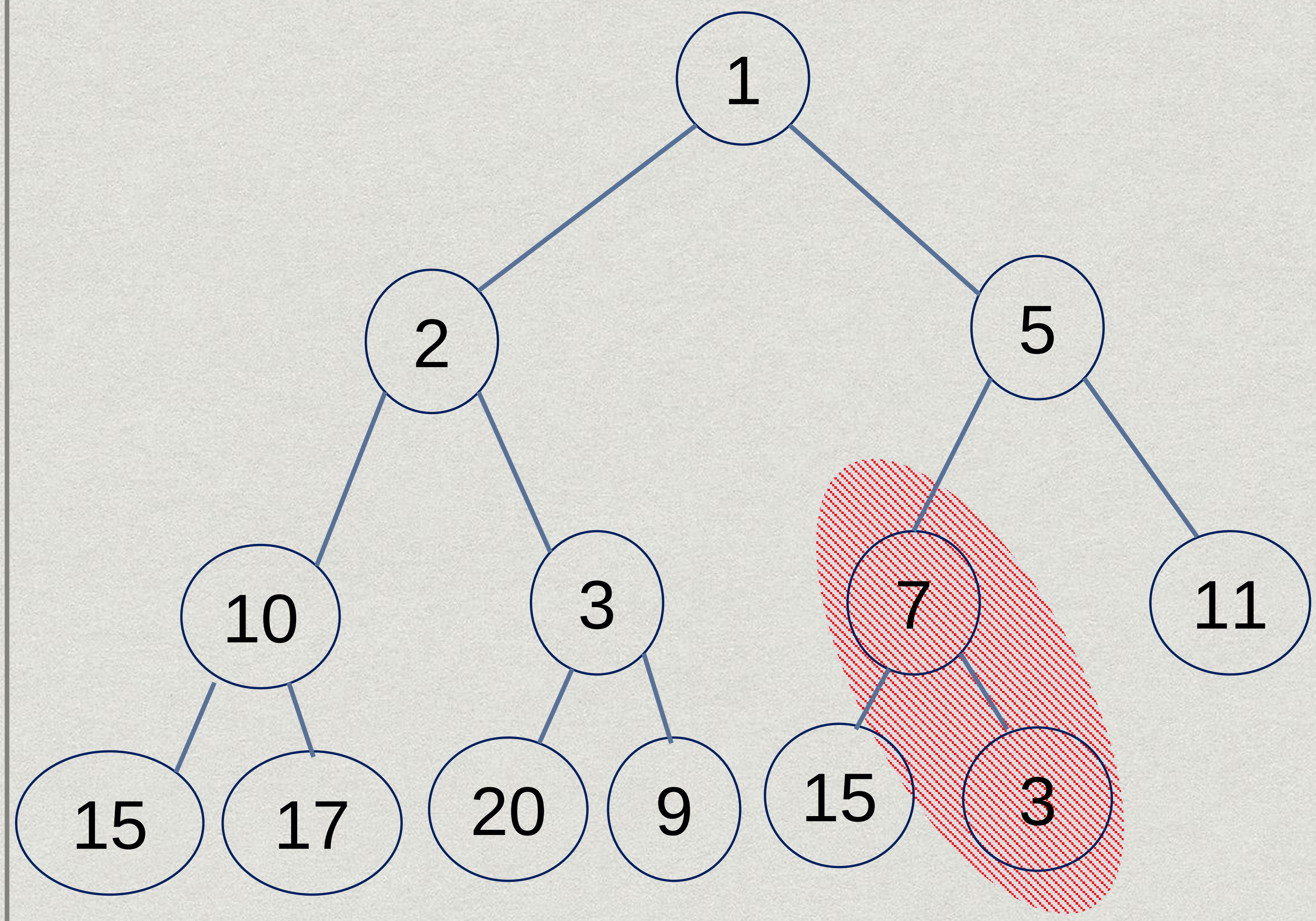
$n = 13$

Heap operations

Add an element

Insert the element at the end of the heap

Heap is damaged !



1	2	5	10	3	7	11	15	17	20	9	15	3	

$n = 13$

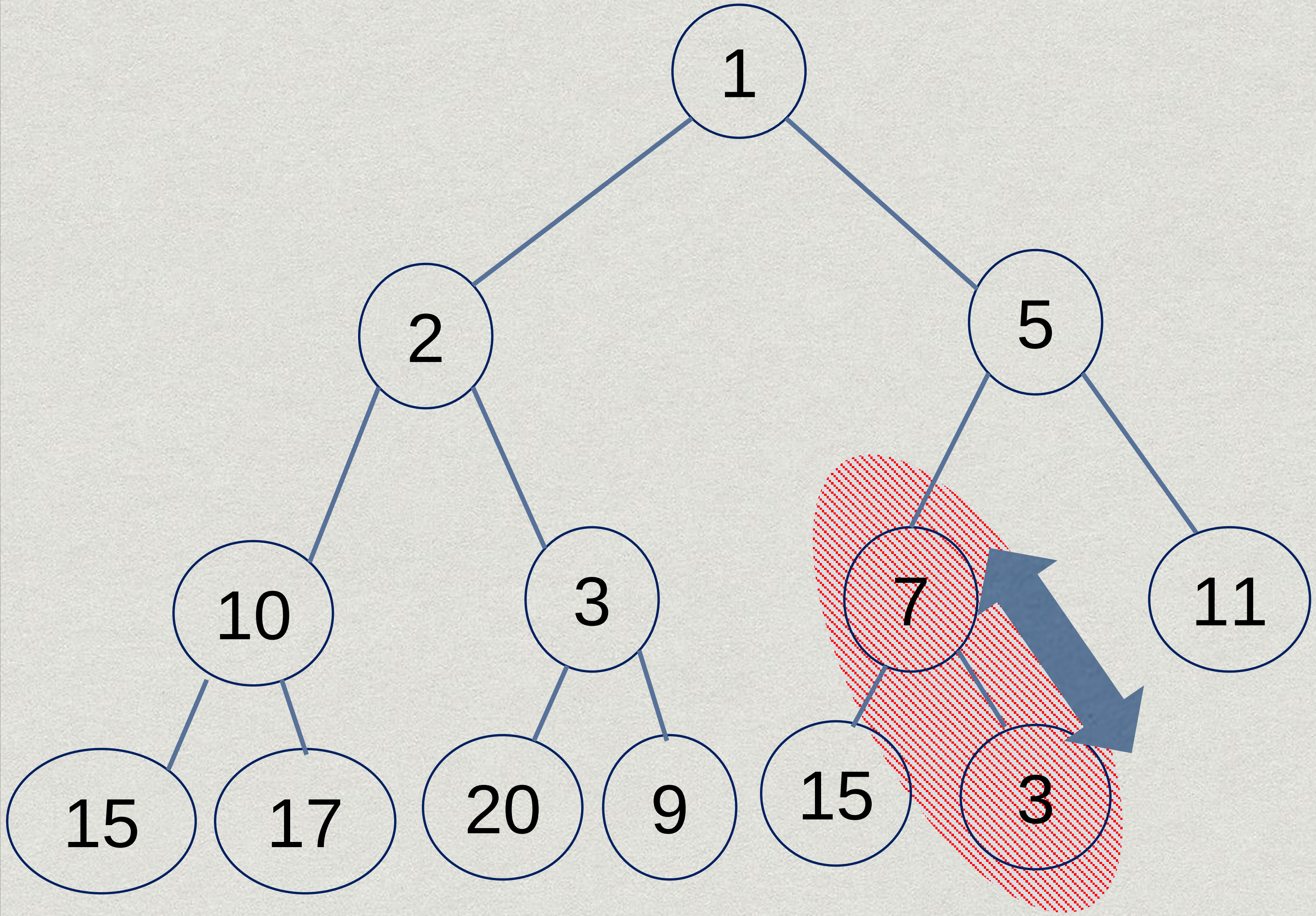
Heap operations

Add an element

Insert the element at the end of the heap

Heap is damaged

Can be fixed by swapping elements



1	2	5	10	3	7	11	15	17	20	9	15	3	

$n = 13$

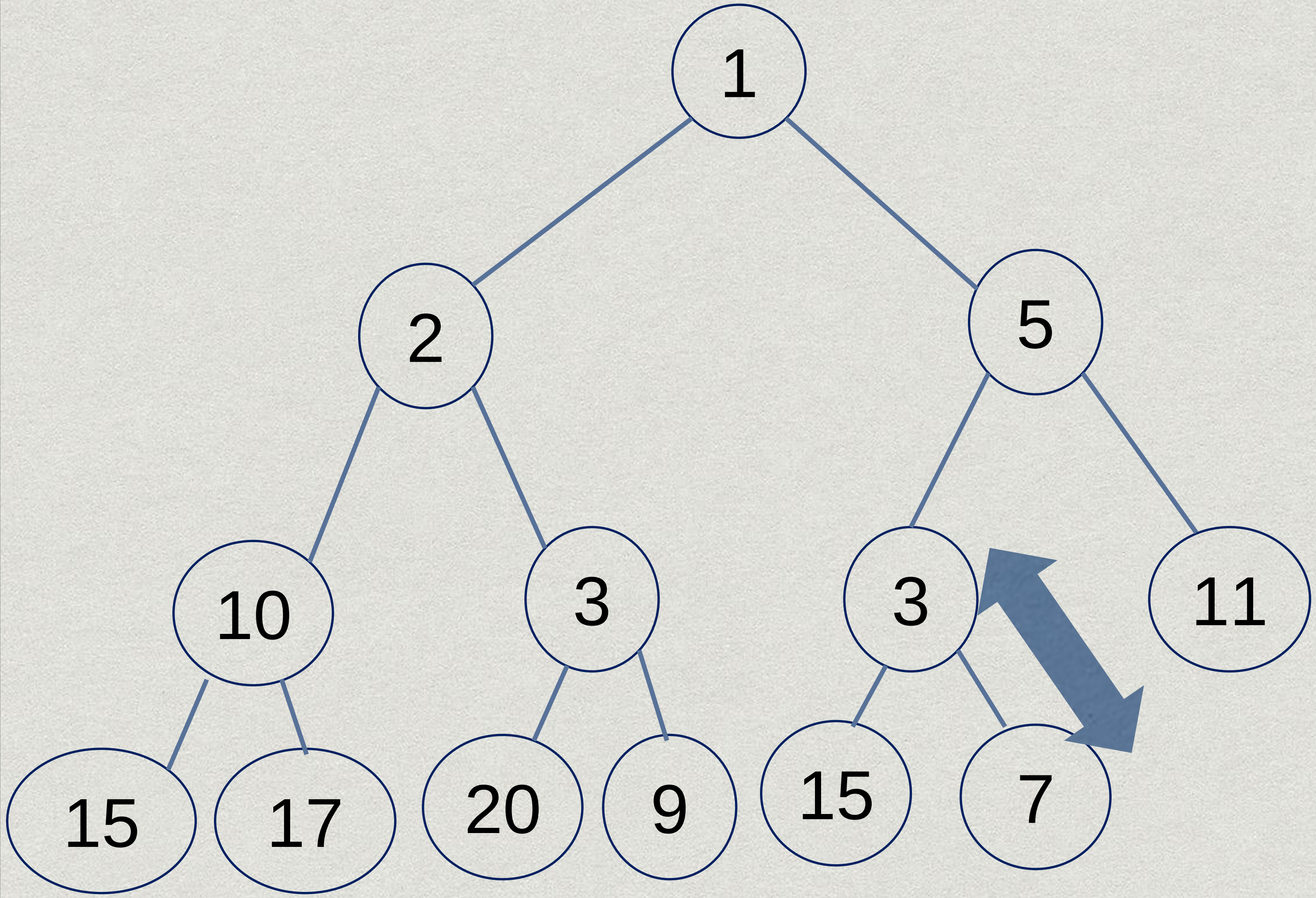
Heap operations

Add an element

Insert the element at the end of the heap

Heap is damaged

Can be fixed by swapping elements

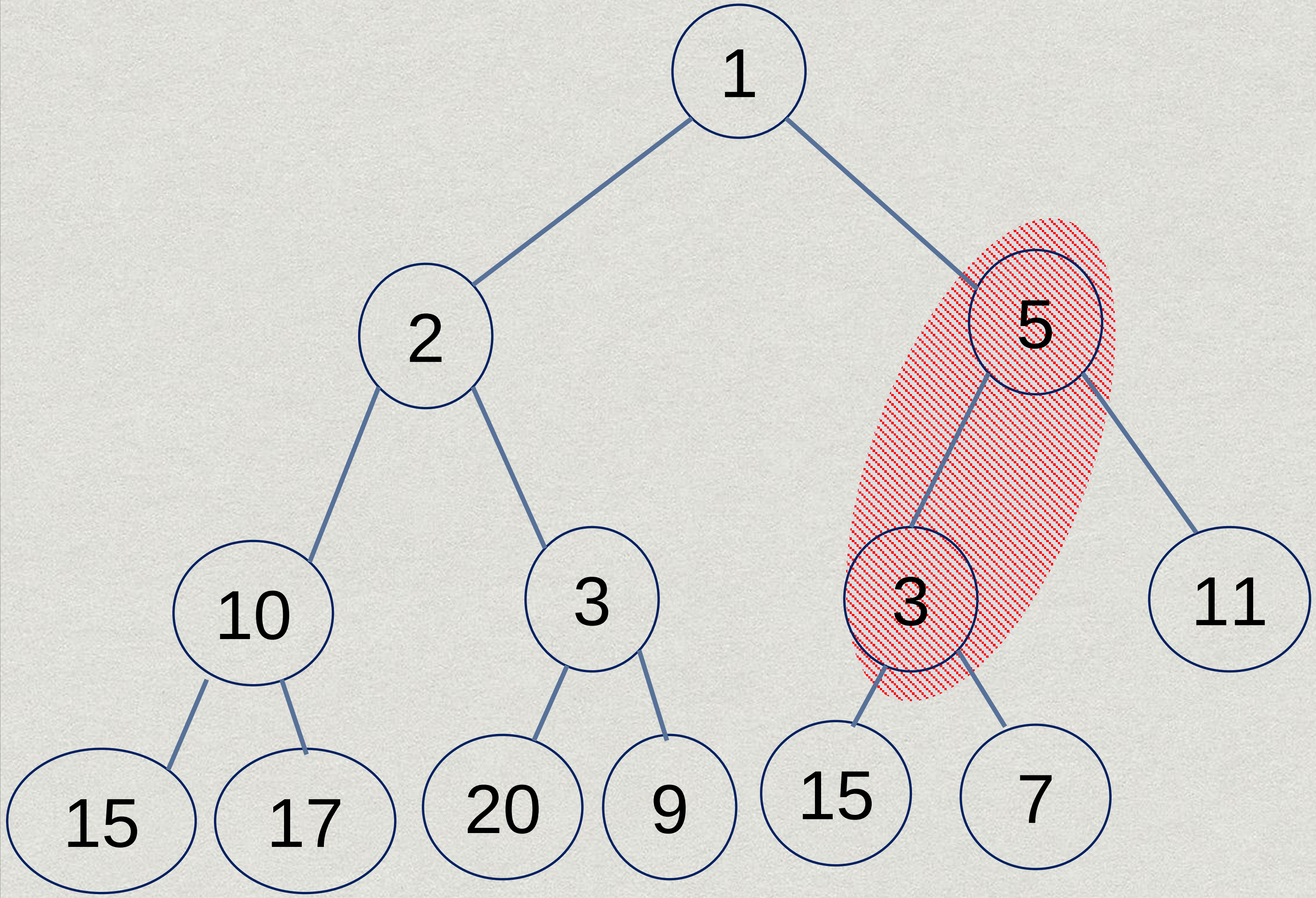


1	2	5	10	3	3	11	15	17	20	9	15	7	

$n = 13$

Heap operations

Add an element



Insert the element at the end of the heap

Heap is damaged

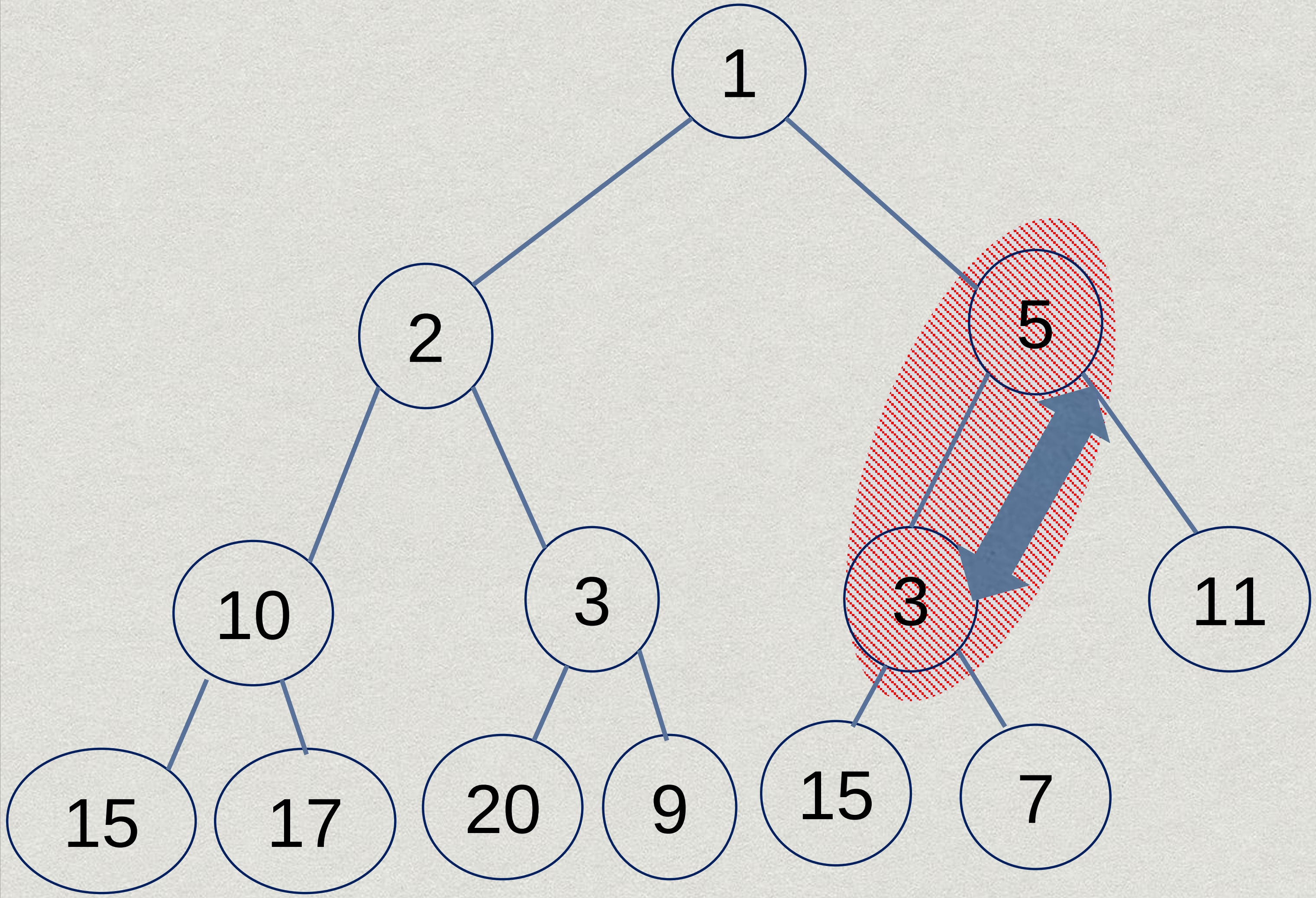
Can be fixed by swapping elements

Heap is still damaged !

1	2	5	10	3	3	11	15	17	20	9	15	7	

$n = 13$

Heap operations



Add an element

Insert the element at the end of the heap

Heap is damaged

Can be fixed by swapping elements

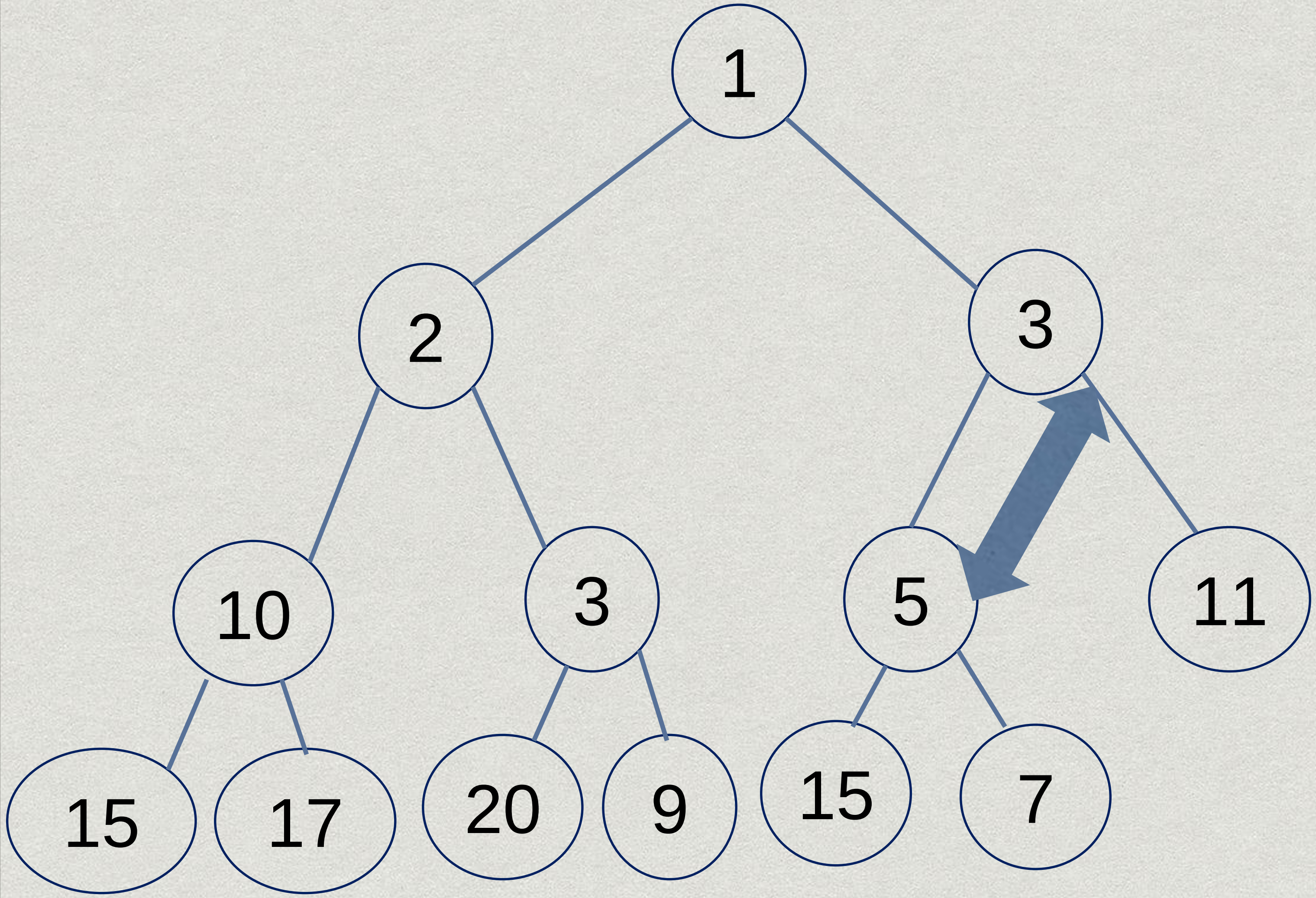
Heap is still damaged

Can be fixed by swapping elements

1	2	5	10	3	3	11	15	17	20	9	15	7	

$n = 13$

Heap operations



Add an element

Insert the element at the end of the heap

Heap is damaged

Can be fixed by swapping elements

Heap is still damaged

Can be fixed by swapping elements

Heap is OK now !

1	2	3	10	3	5	11	15	17	20	9	15	7	

$n = 13$

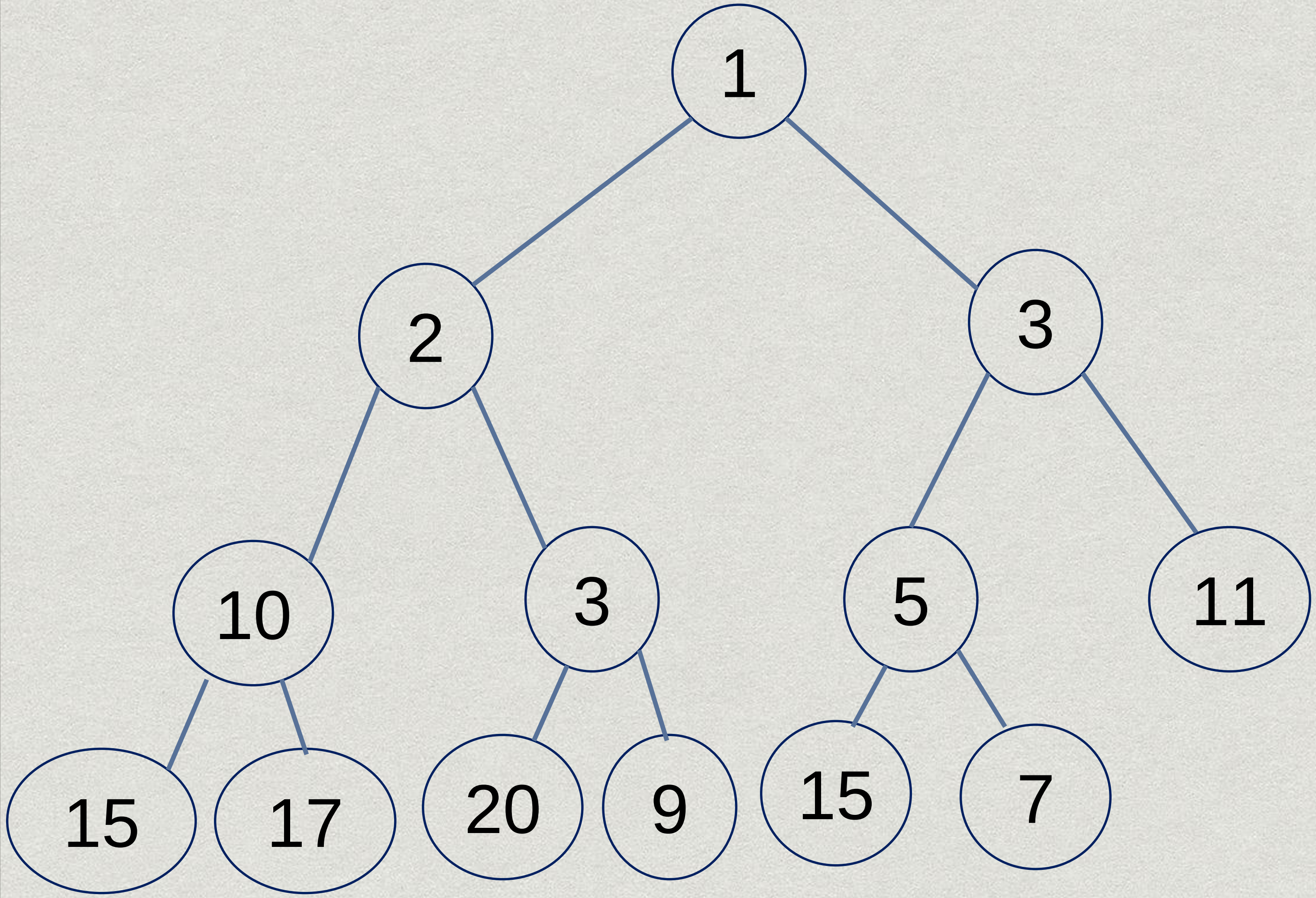
heapify_up

```
function heapify_up (H,i)  { //  fix a heap that is damaged at node i

  If i > 1 then
     j = parent(i)

      If key(H[i]) <  key(H[j]) then

        swap the array entries H[i] and H[j]
        heapify_up(H,j)

     Endif
  Endif

}
```

$O(\log n)$

Heap operations

Delete an element:
say 1



1	2	3	10	3	5	11	15	17	20	9	15	7	

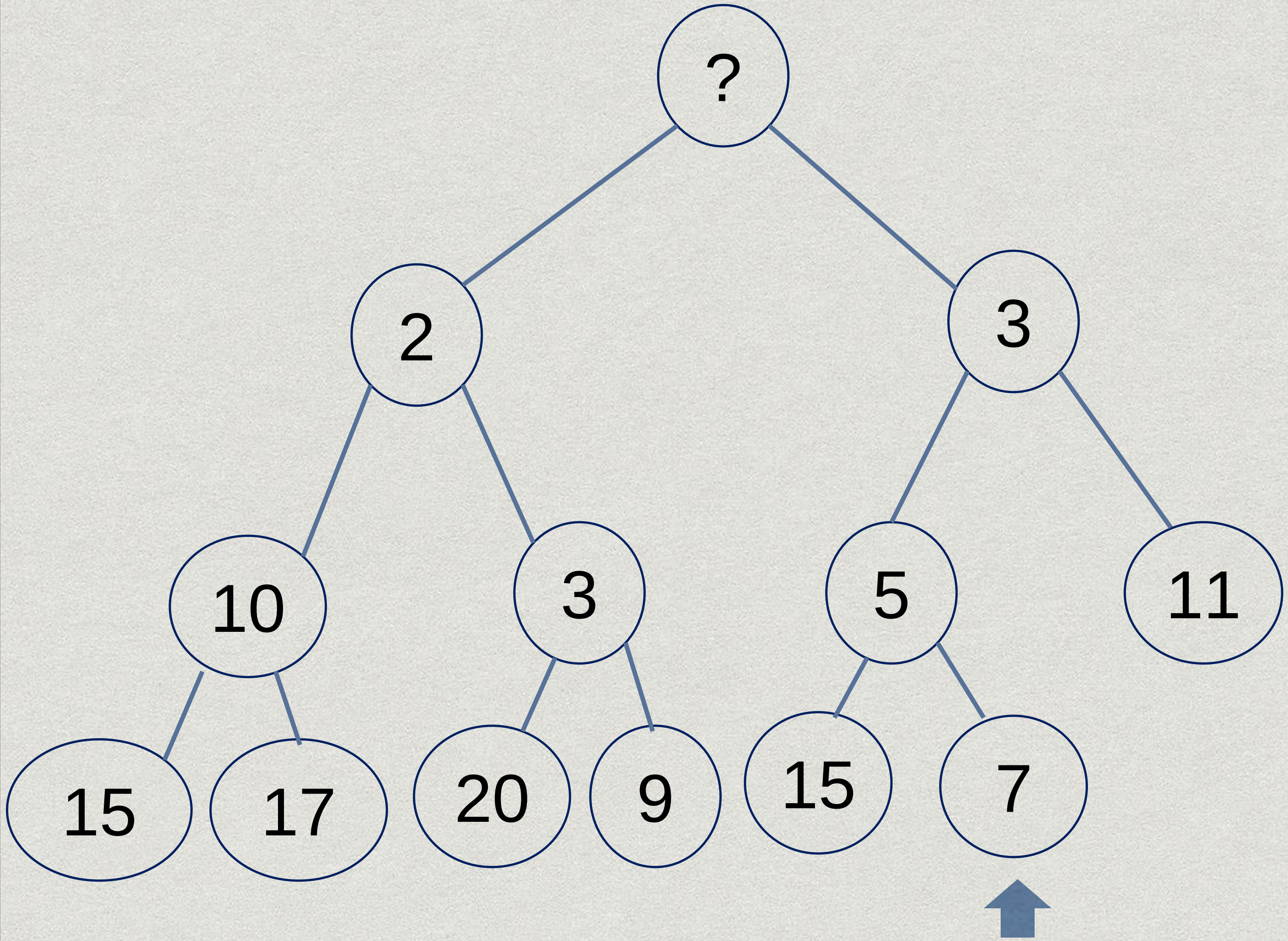
$n = 13$

Heap operations

Delete an element:
say 1

Leaves a hole at the top of the heap

Move the last element to that place



?	2	3	10	3	5	11	15	17	20	9	15	7	

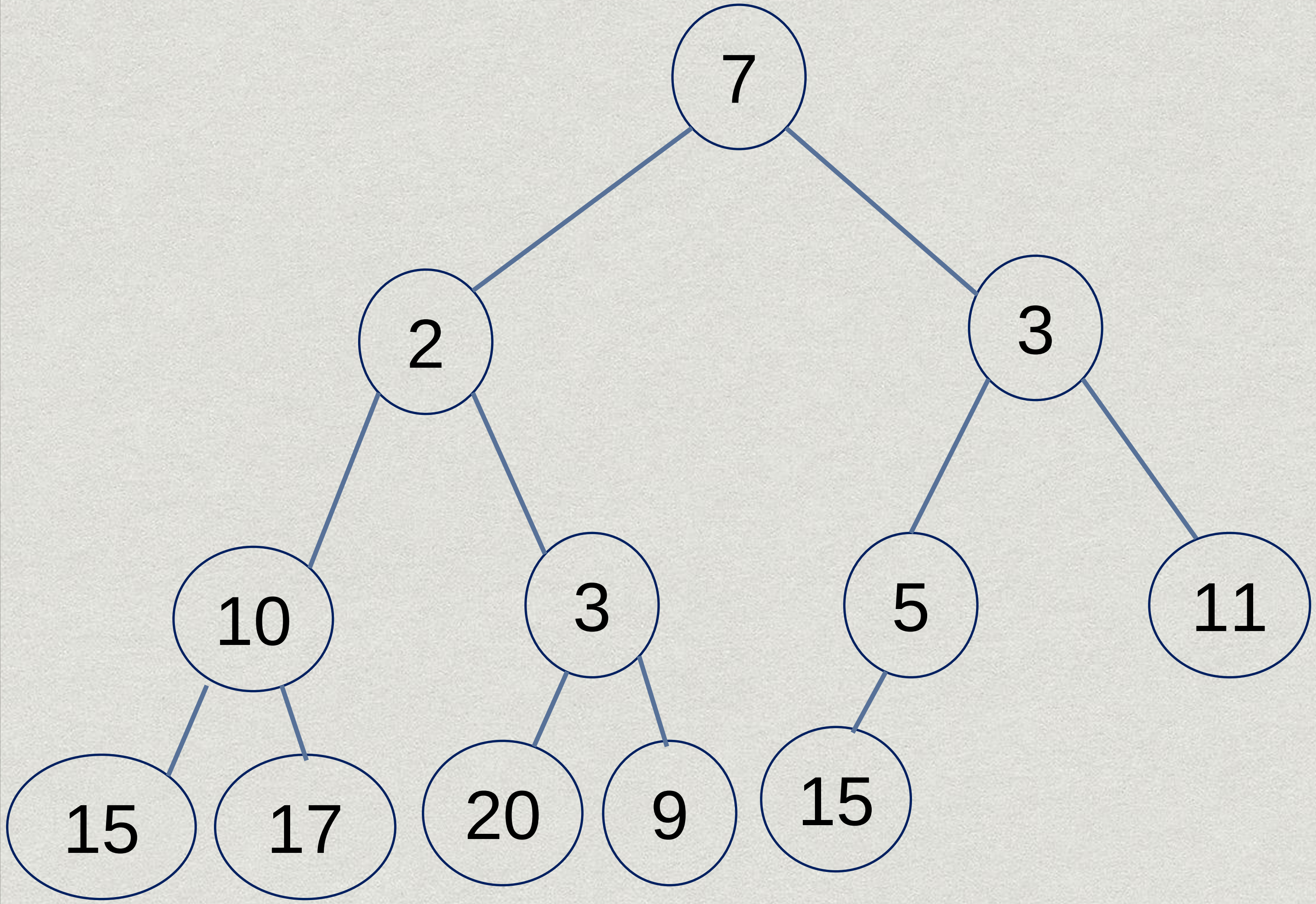
$n = 13$

Heap operations

Delete an element:
say 1

Leaves a hole at the top of the heap

Move the last element to that place



7	2	3	10	3	5	11	15	17	20	9	15		

$n = 12$

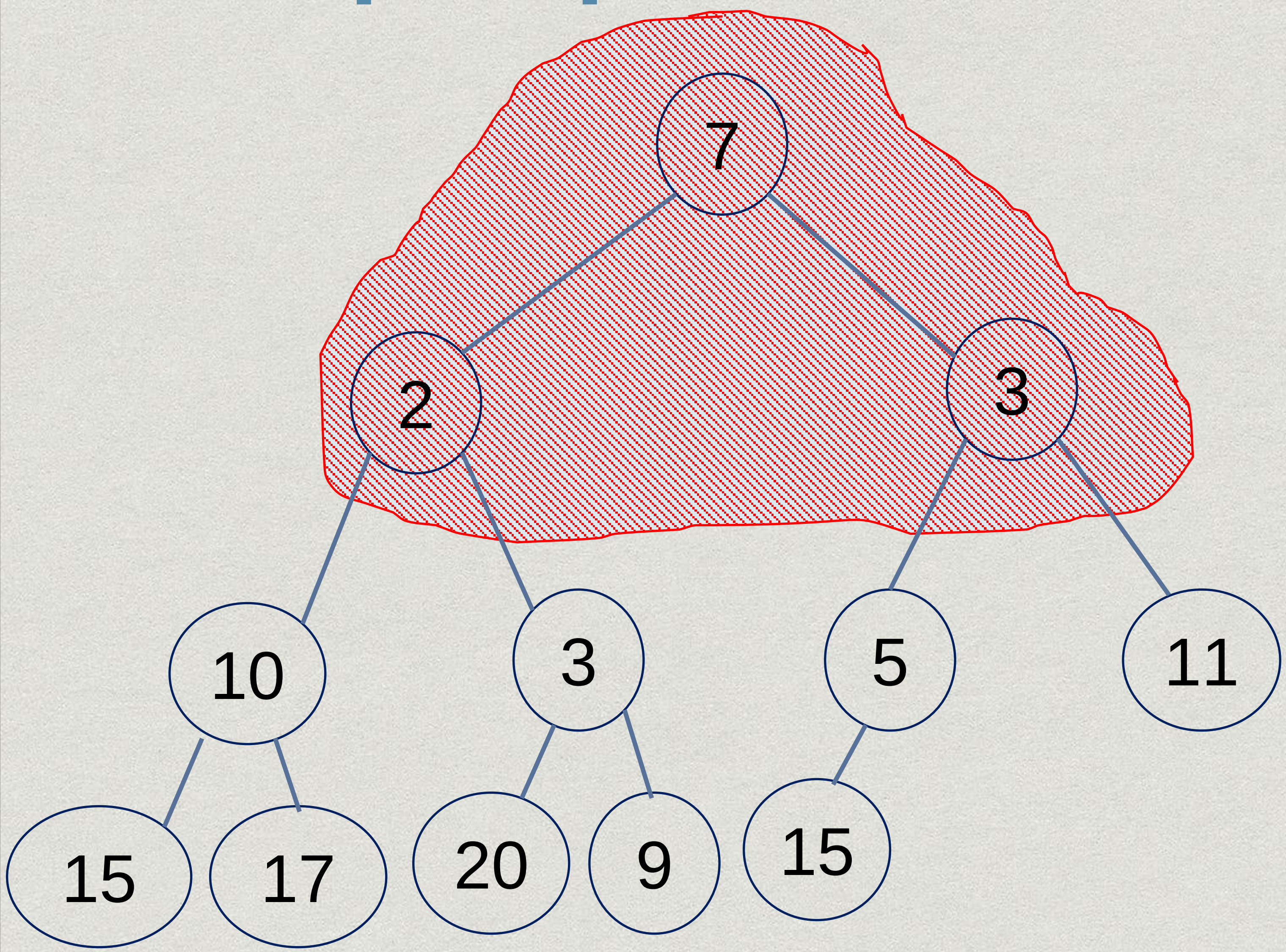
Heap operations

Delete an element: say 1

Leaves a hole at the top of the heap

Move the last element to that place

Heap is damaged



7	2	3	10	3	5	11	15	17	20	9	15		

$n = 12$

Heap operations

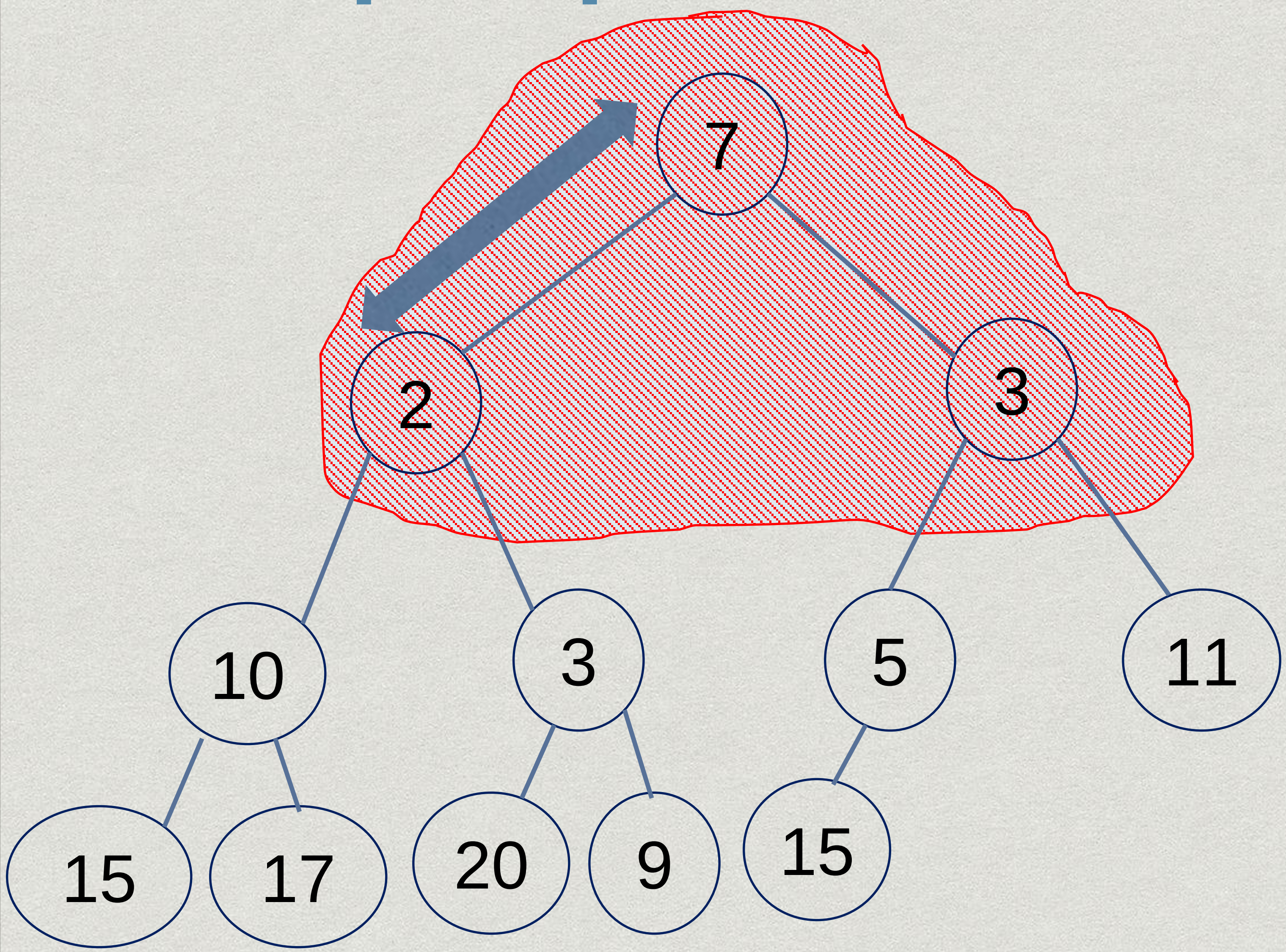
Delete an element: say 1

Leaves a hole at the top of the heap

Move the last element to that place

Heap is damaged

Swap with the smaller

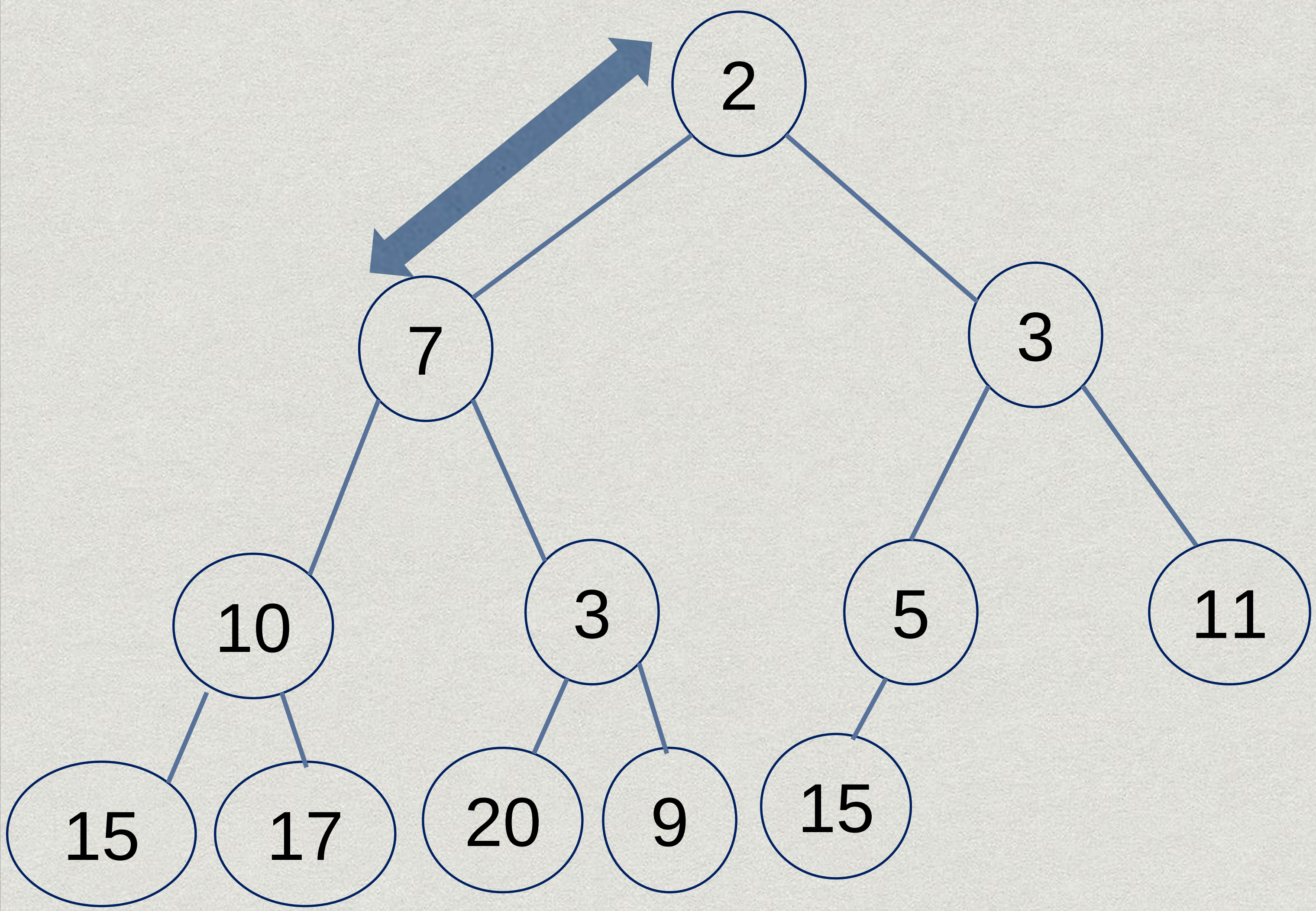


7	2	3	10	3	5	11	15	17	20	9	15		

$n = 12$

Heap operations

Delete an element:
say 1



Leaves a hole at the top of the heap

Move the last element to that place

Heap is damaged

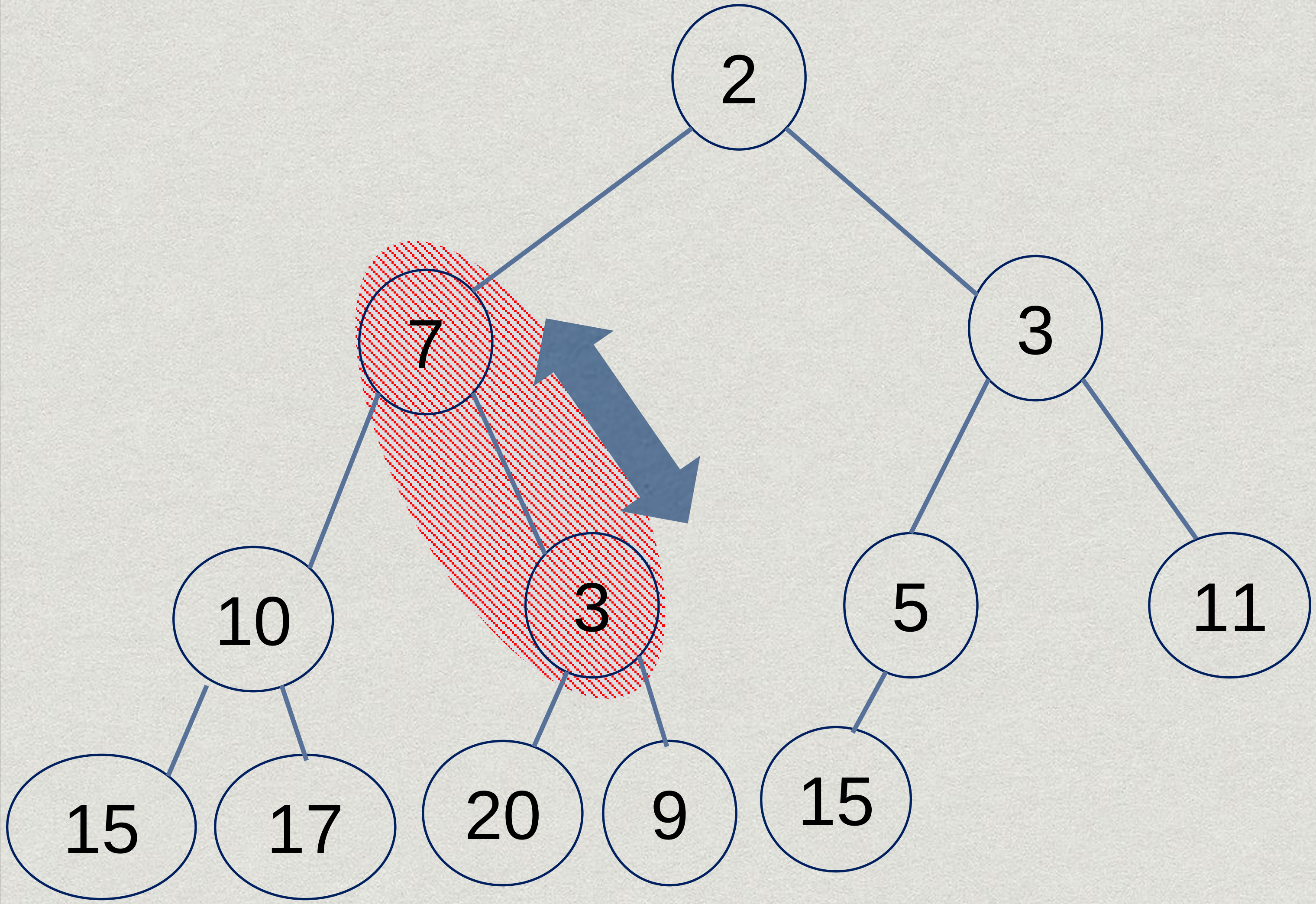
Swap with the smaller

2	7	3	10	3	5	11	15	17	20	9	15		

$n = 12$

Heap operations

Delete an element:
say 1



Leaves a hole at the top of the heap

Move the last element to that place

Heap is damaged

Swap with the smaller

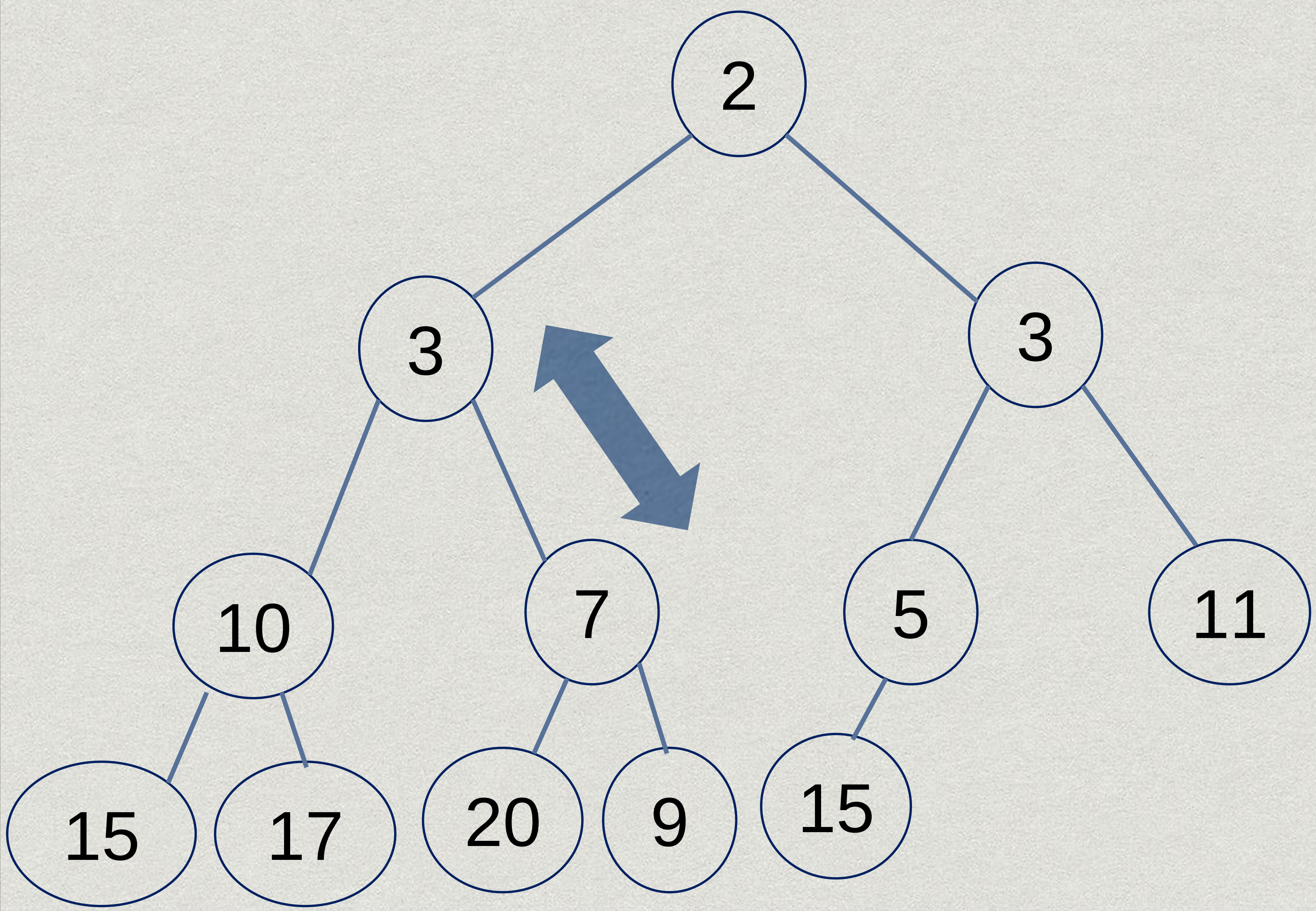
Heap is still damaged
Swap to fix

2	7	3	10	3	5	11	15	17	20	9	15		

$n = 12$

Heap operations

Delete an element:
say 1



Leaves a hole at the top of the heap

Move the last element to that place

Heap is damaged

Swap with the smaller

Heap is still damaged
Swap to fix
Heap is OK now !

2	3	3	10	7	5	11	15	17	20	9	15		

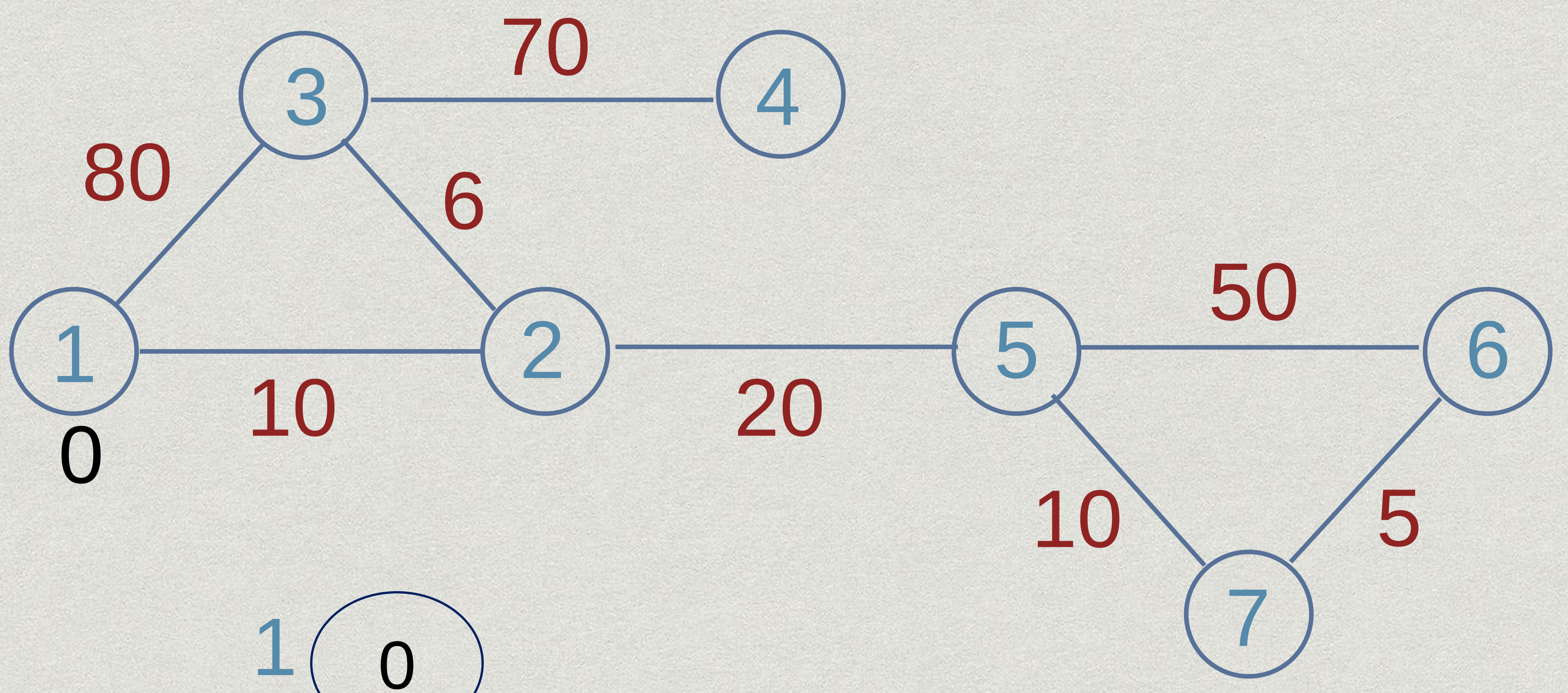
$n = 12$

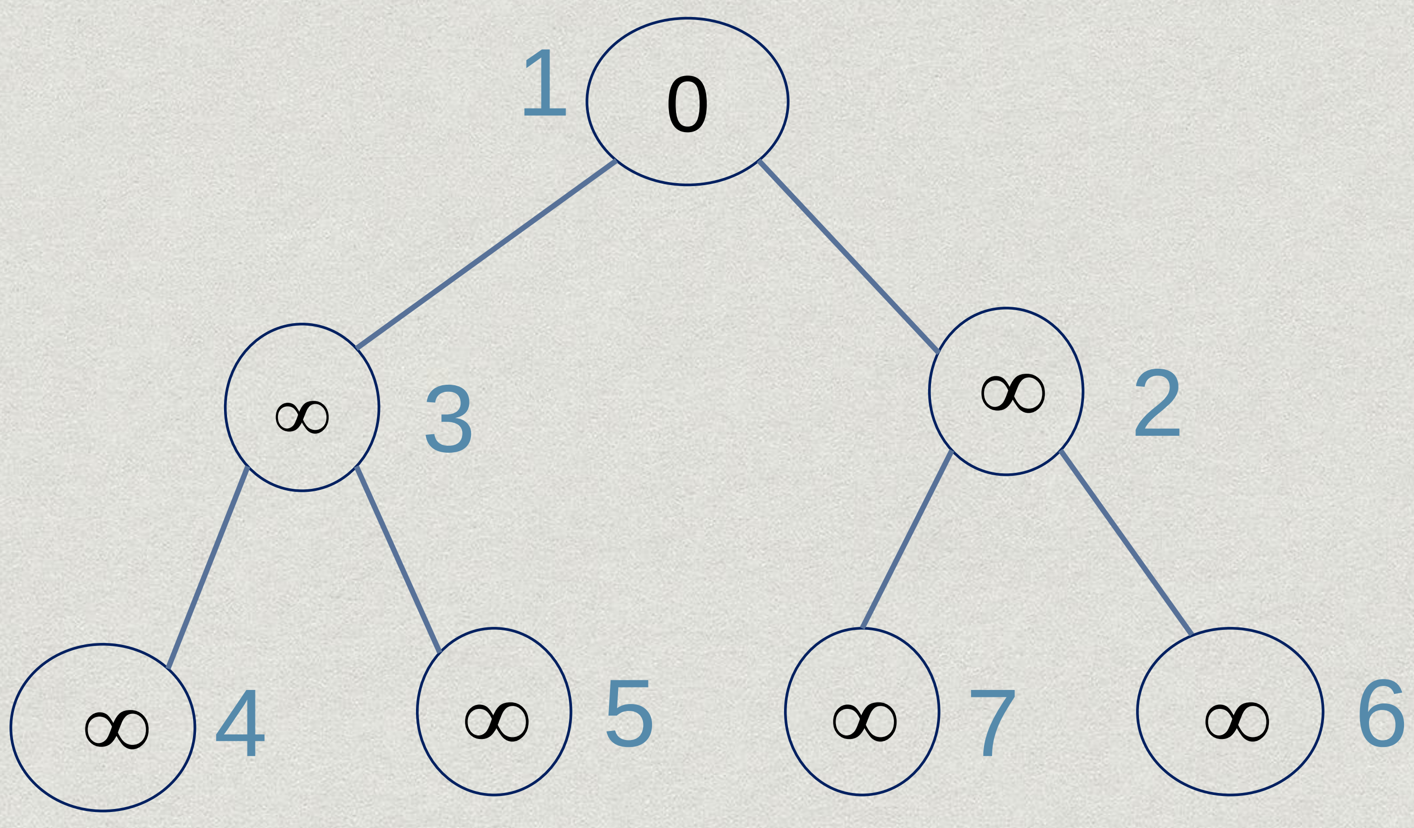
heapify_down

```
function heapify_down (H,i)  { //  fix a heap H that is damaged at node i
    n = length(H)
    If 2*i > n then
       Terminate with H unchanged                    // no children below i

   Else if 2*i < n then
       left = 2*i,   right = 2*i + 1
       If key(H[left]) > key(H[right]) then j = right  // choose the min between
       Else j = left                                 // left and right children
       Endif
   Else if 2*i == n then                             // single child below i
       j = 2*i

   Endif

    If key(H[j]) <  key(H[i]) then

         swap the array entries H[i] and H[j]
         heapify_down(H,j)

    Endif

}
```

$$O(\log n)$$

Revisit Dijkstra's algorithm

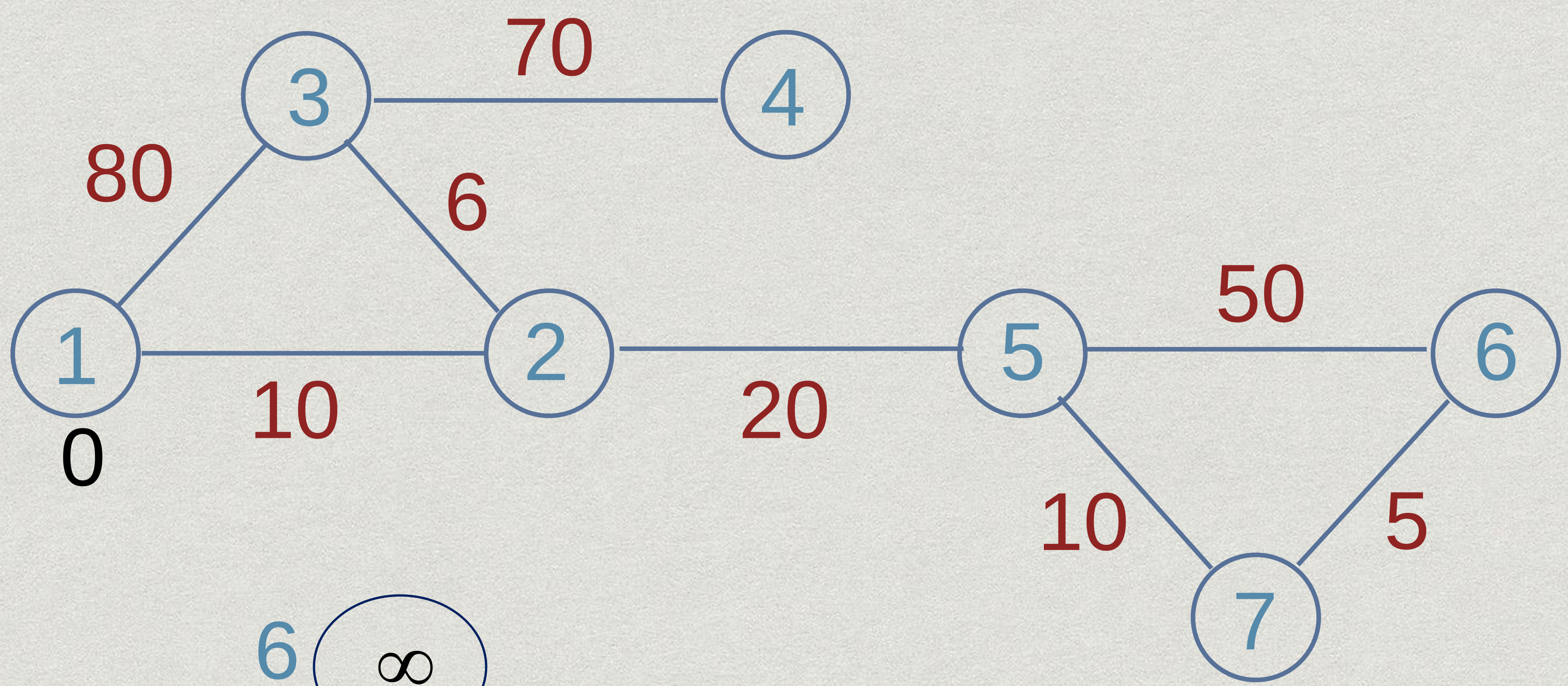


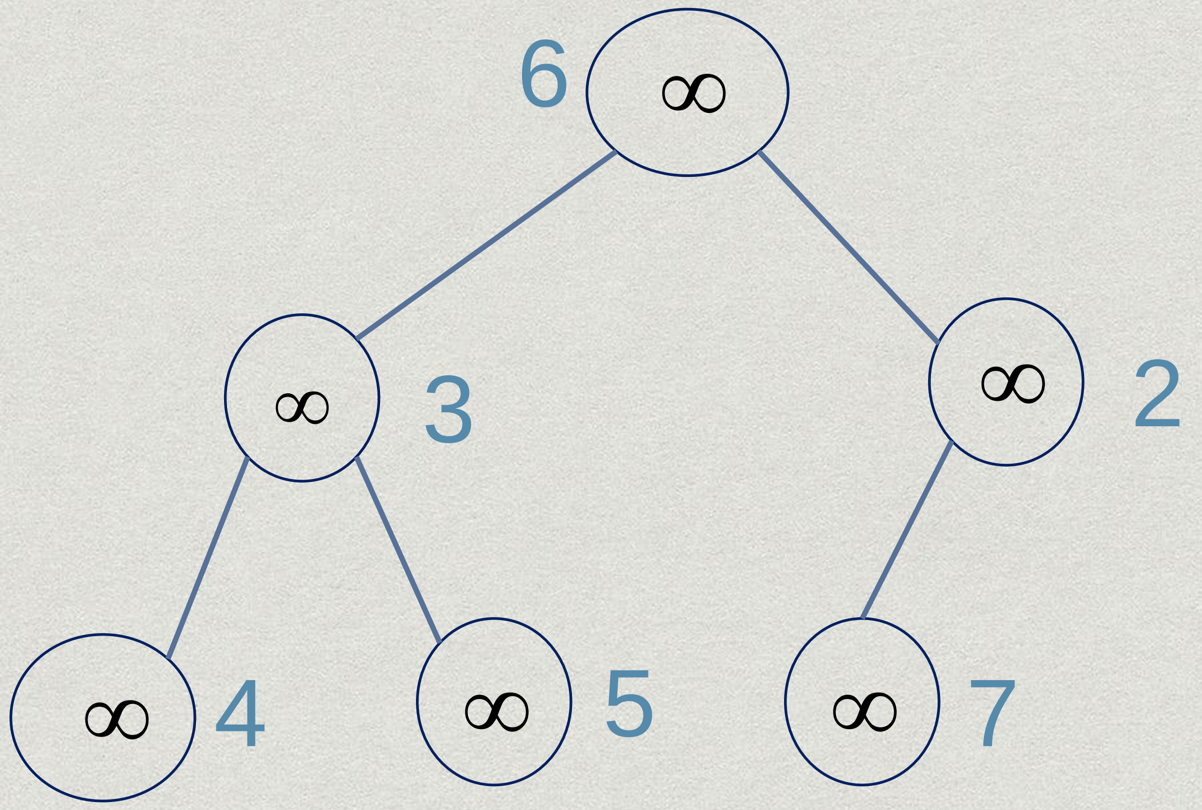


Initial heap
with key of 1 set to 0
and all others set to ∞

Node 1 is extracted,
6 is moved in its place

Revisit Dijkstra's algorithm

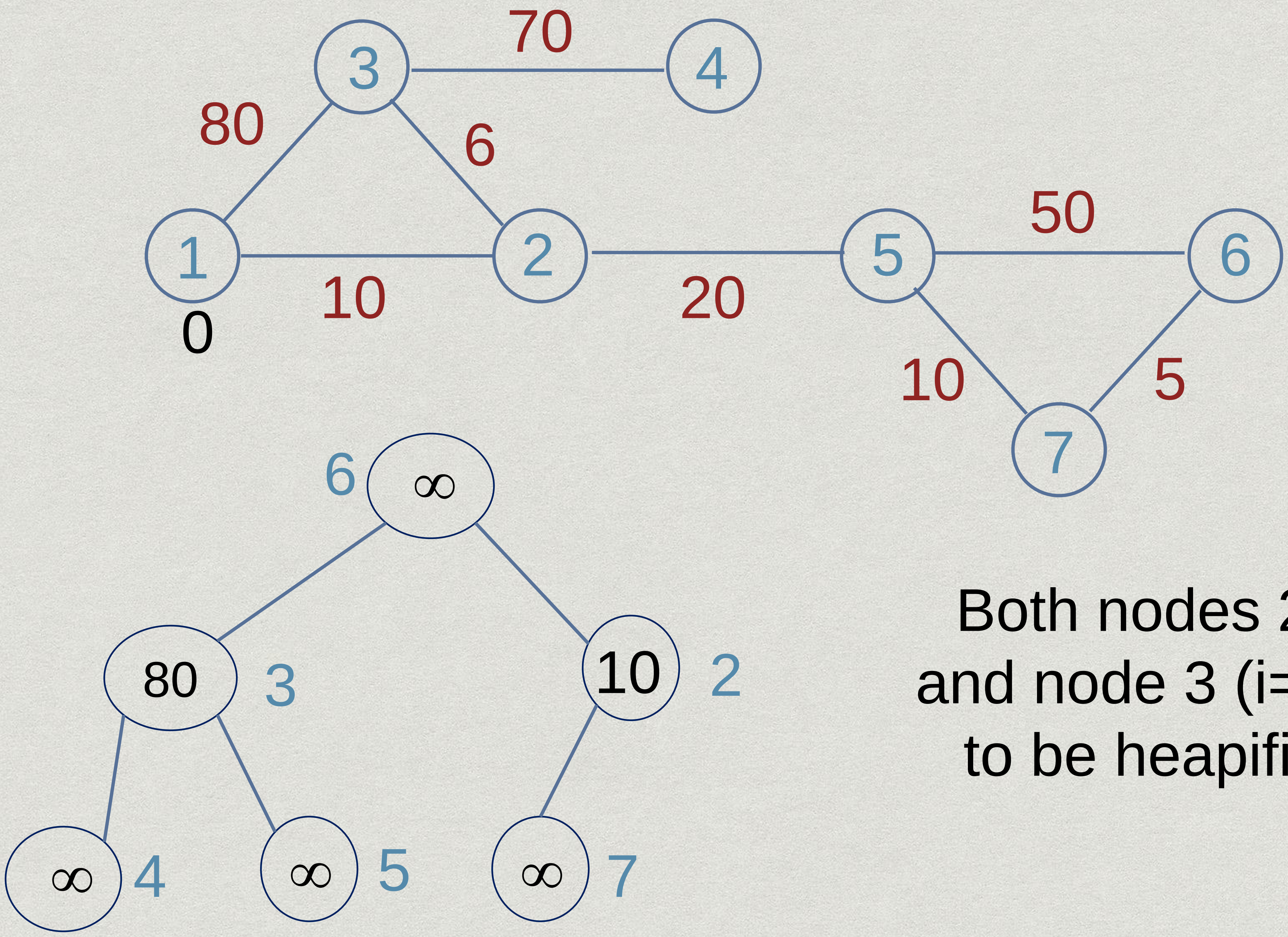




No need to heapify

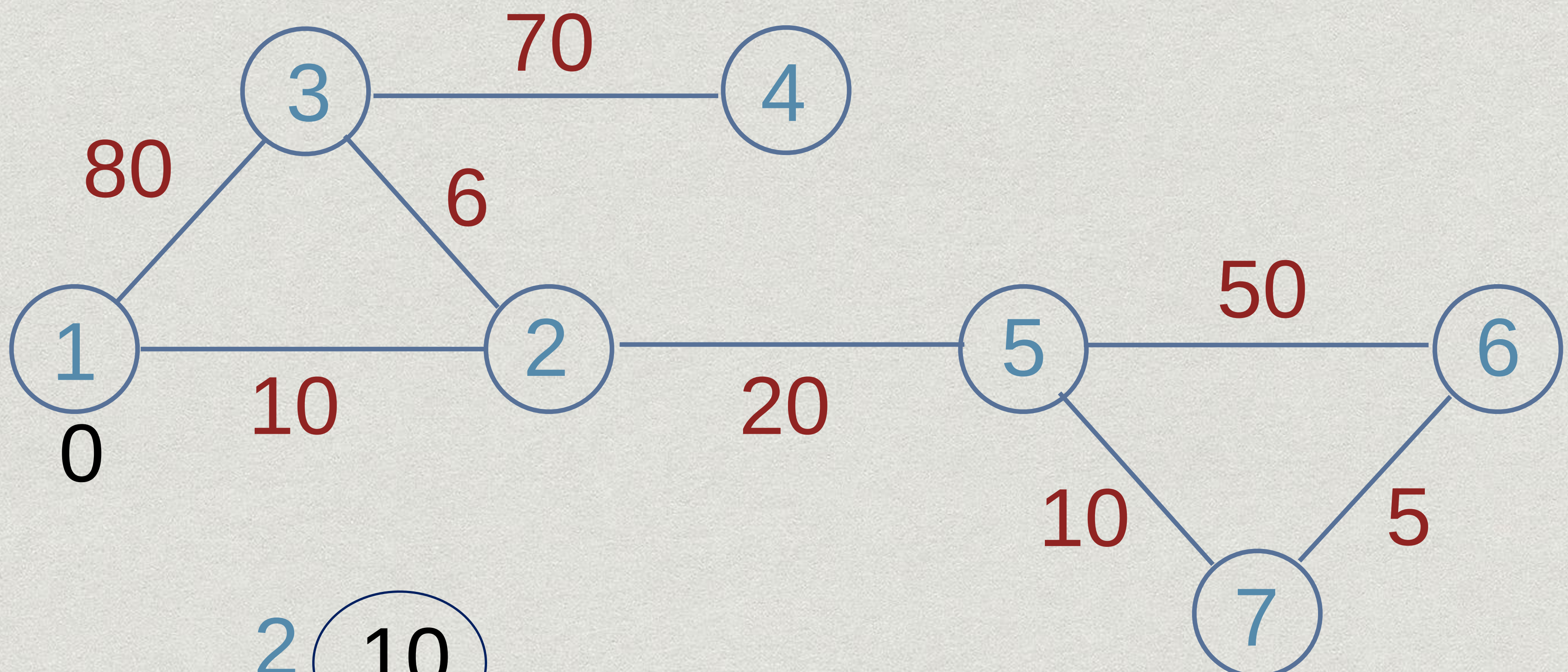
Now we need to do a distance update for node 1

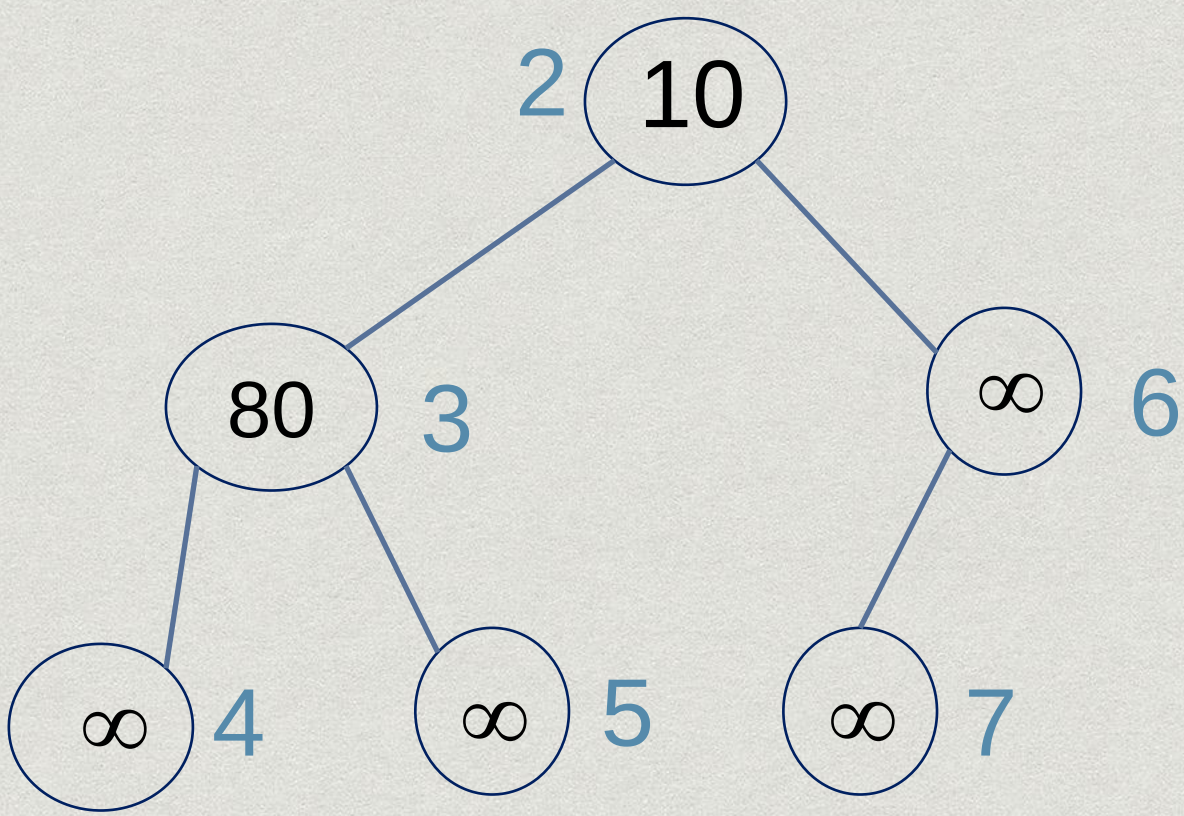
Revisit Dijkstra’s algorithm



Both nodes 2 (i=3) and node 3 (i=2) need to be heapified up

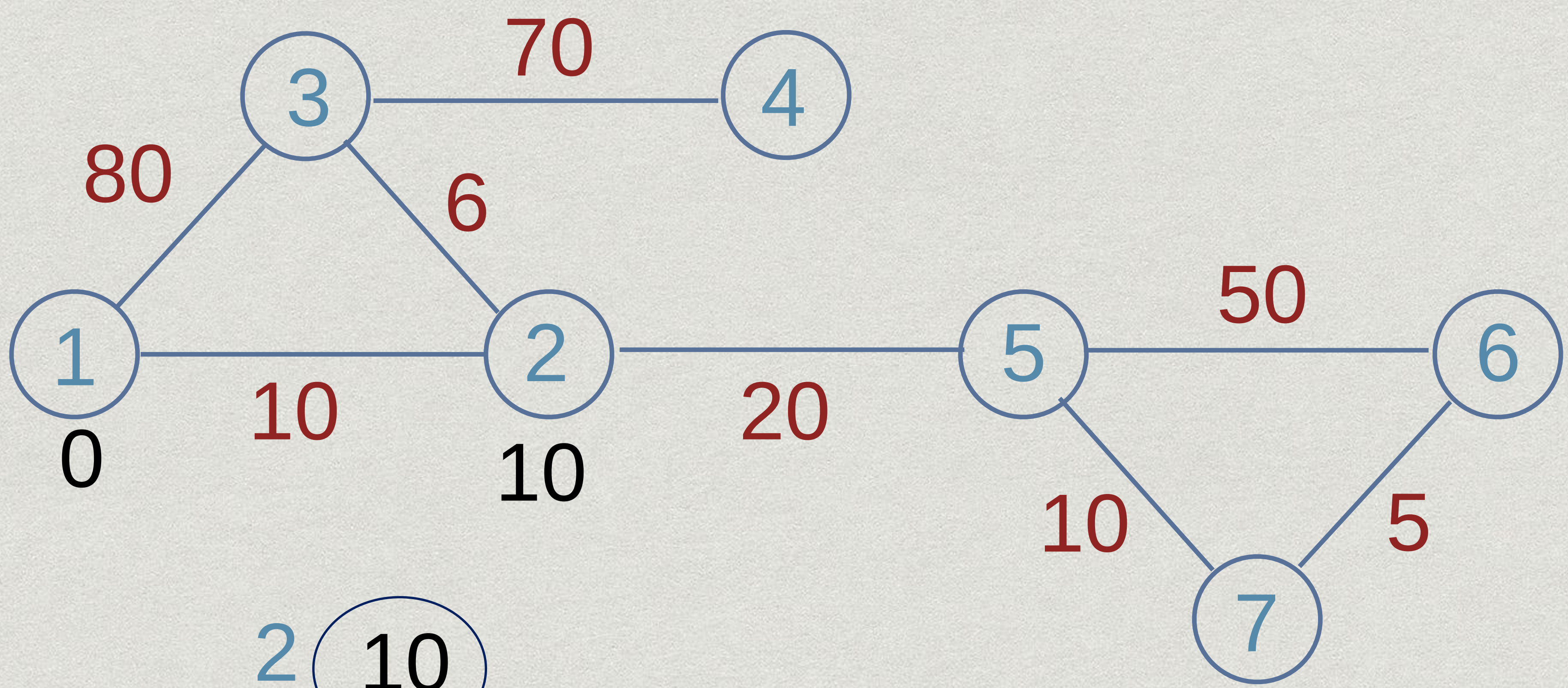
Revisit Dijkstra’s algorithm

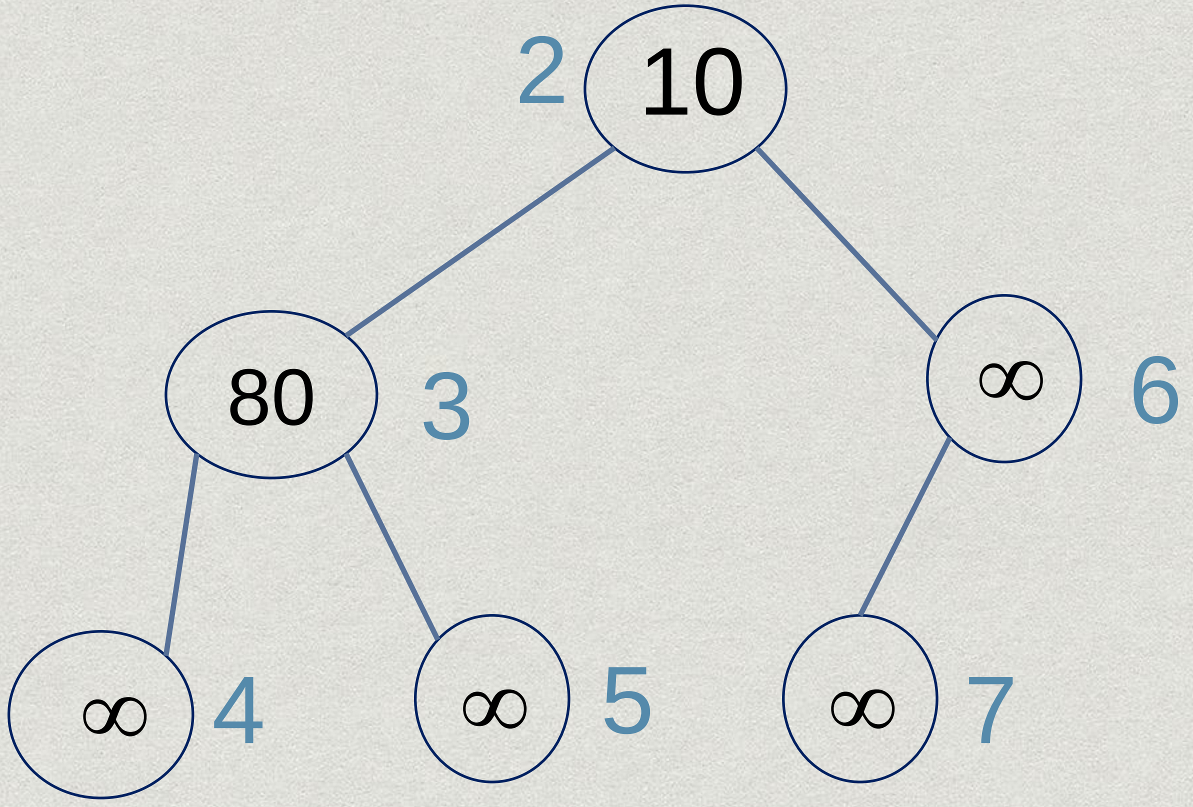




After heapify_up of i=3
we already have a heap

Revisit Dijkstra’s algorithm

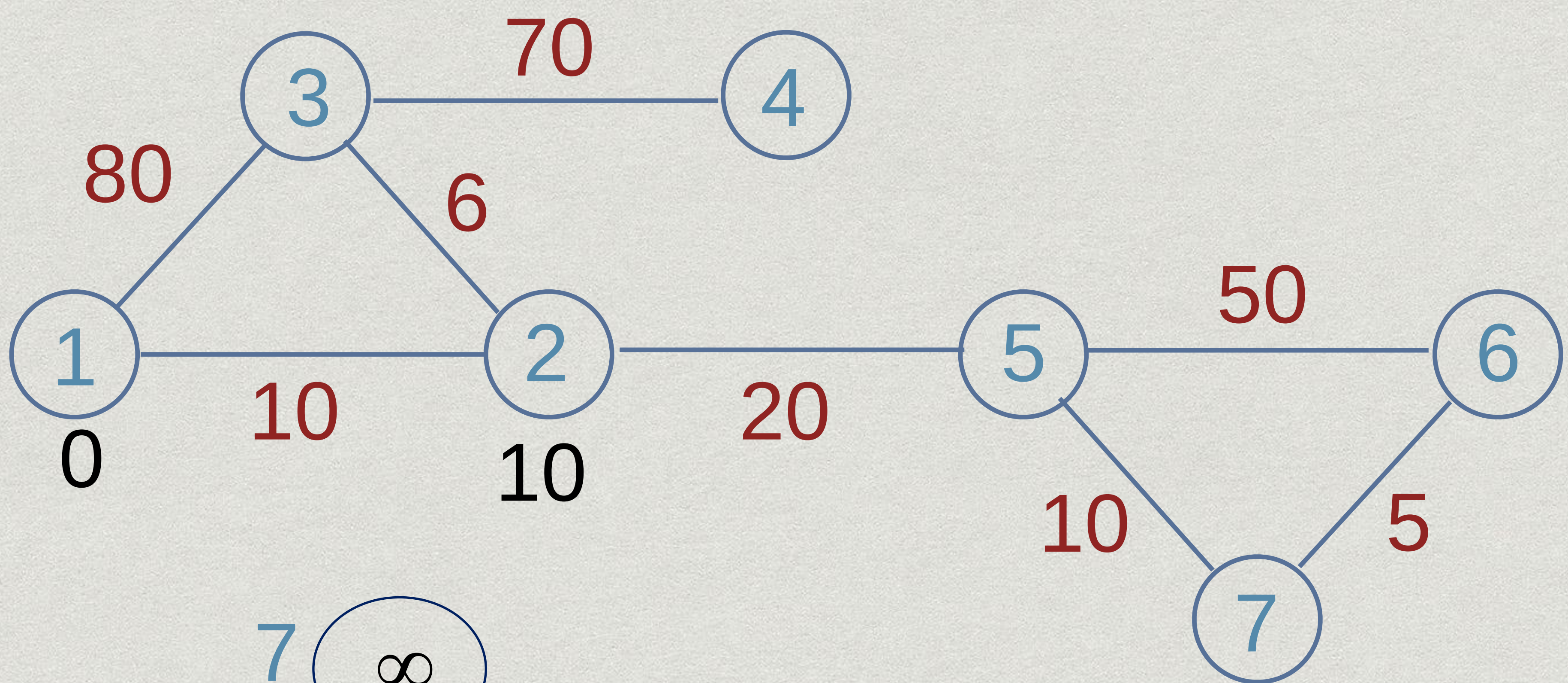


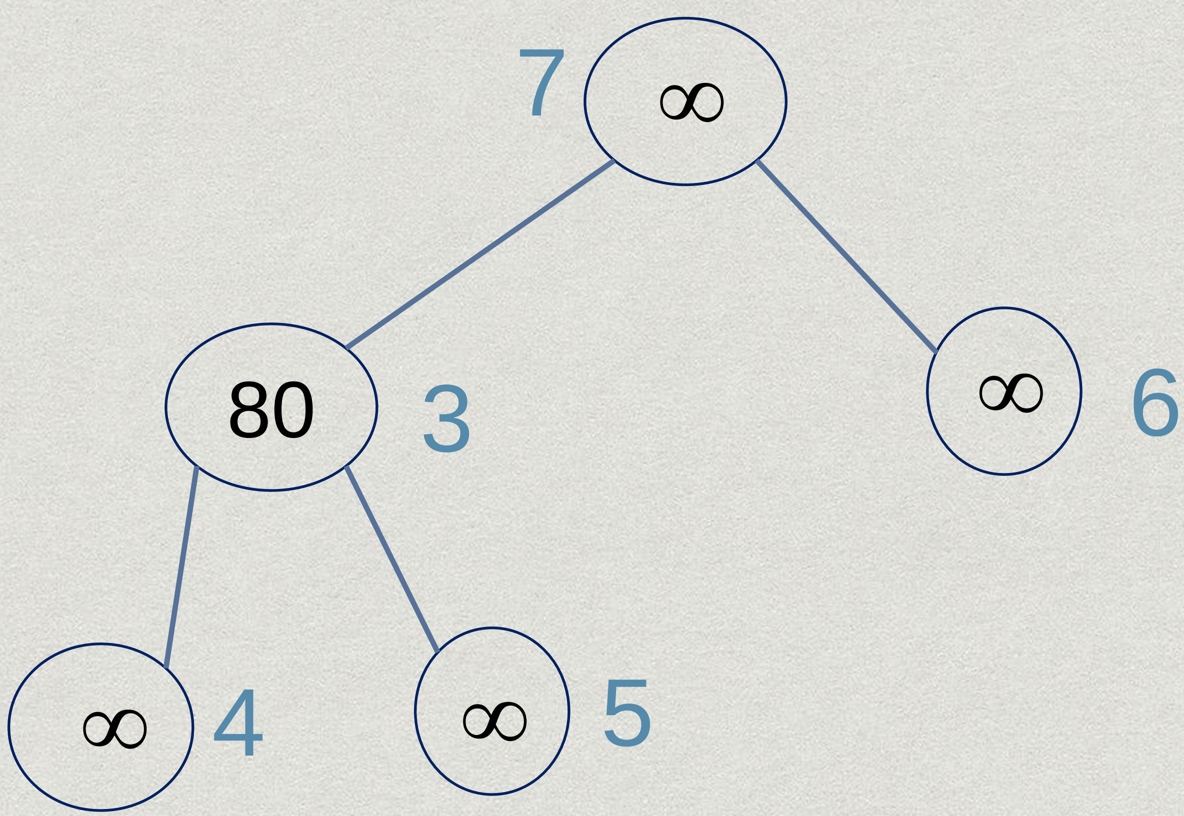


Select node 2

Delete node 2, it gets replaced by node 7

Revisit Dijkstra's algorithm



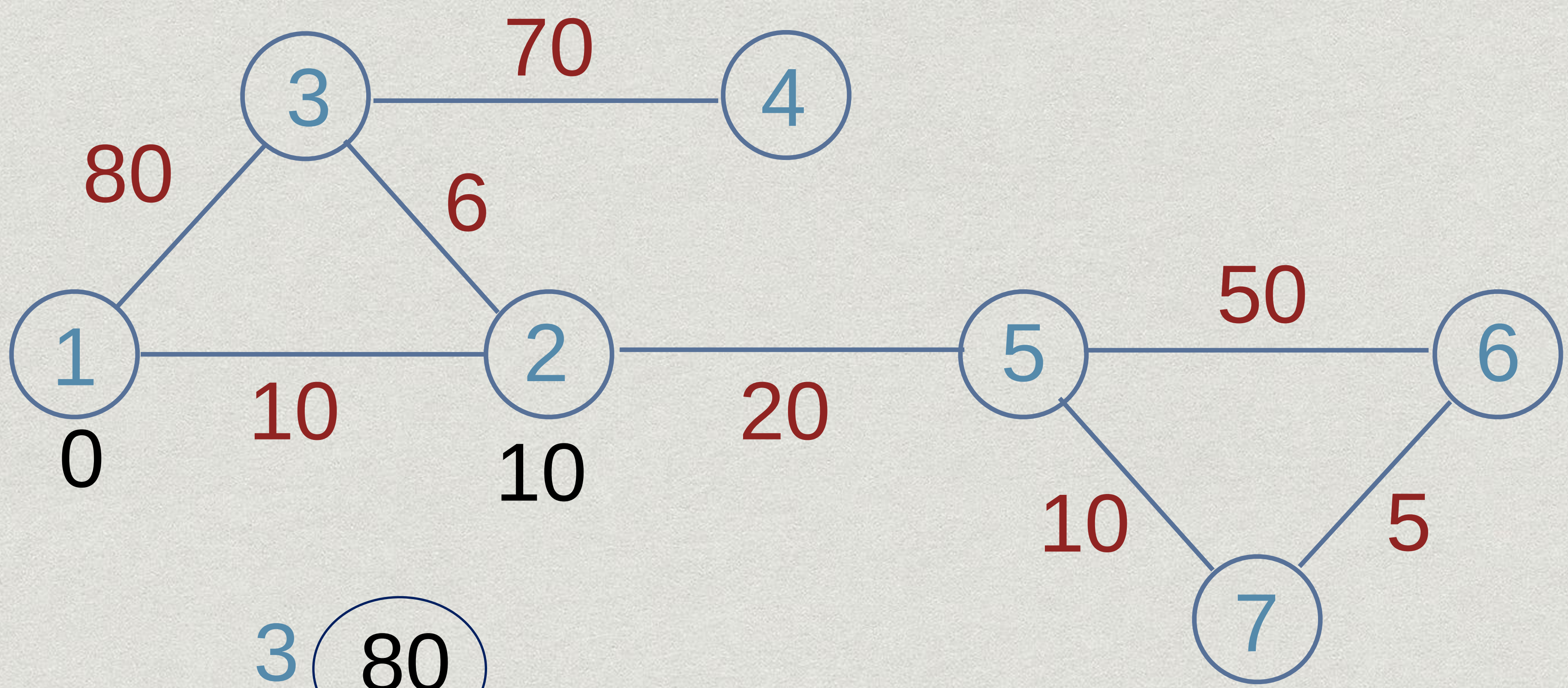


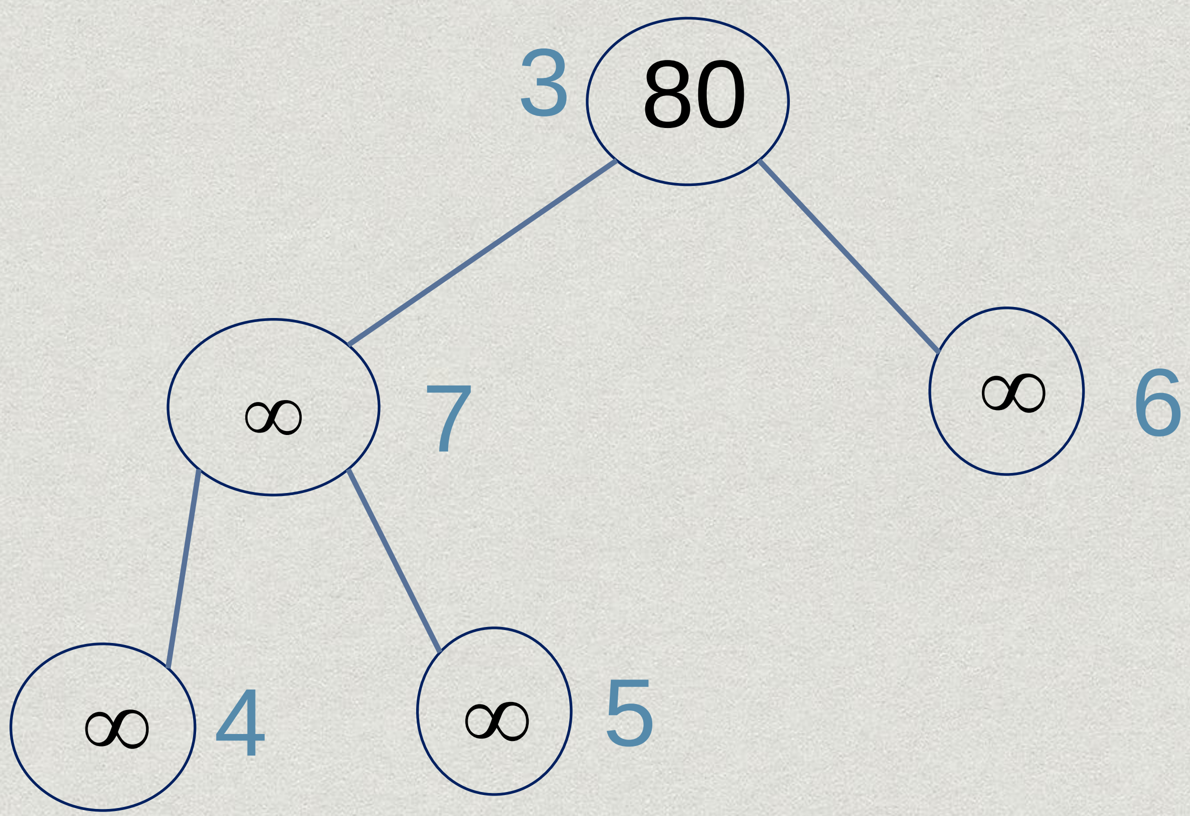
Select node 2

Delete node 2, it gets replaced by node 7

Heapify_down node 7 (i=1)

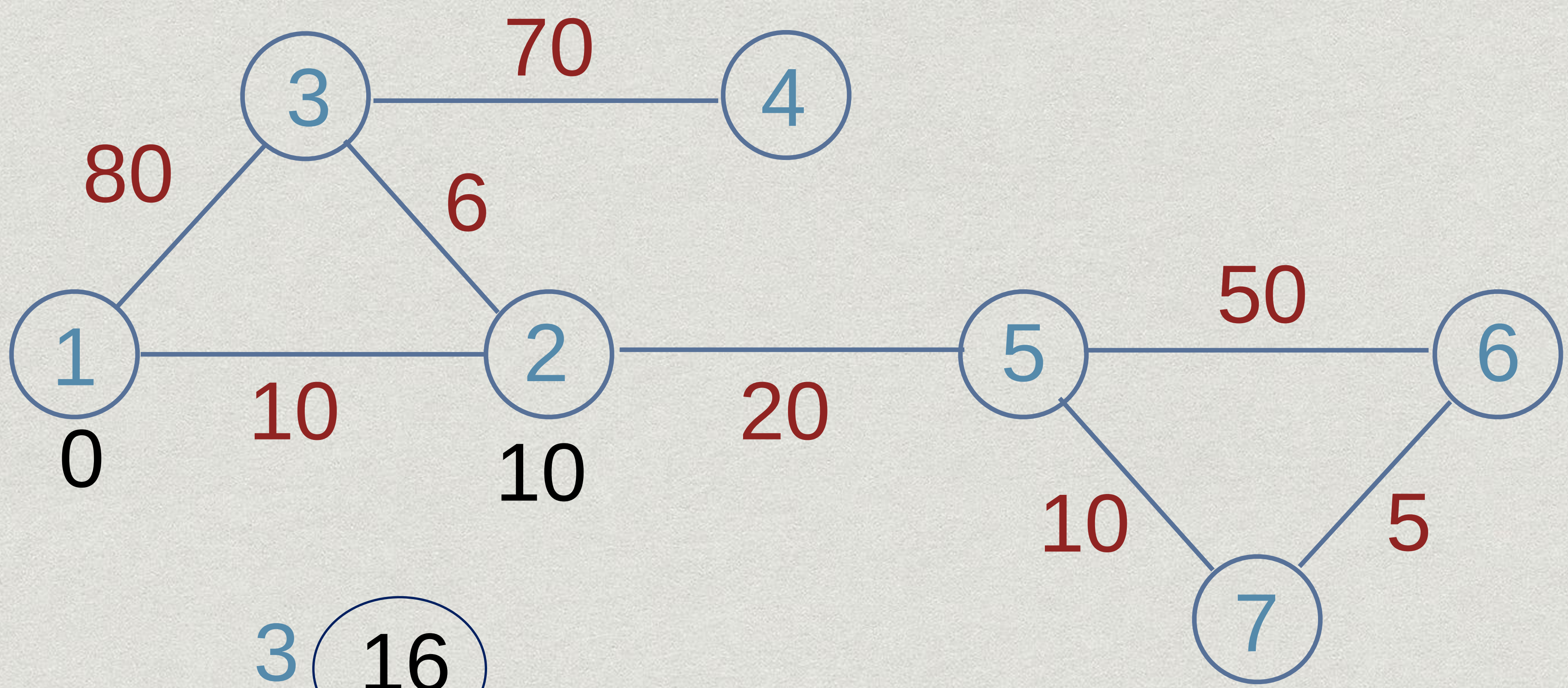
Revisit Dijkstra's algorithm

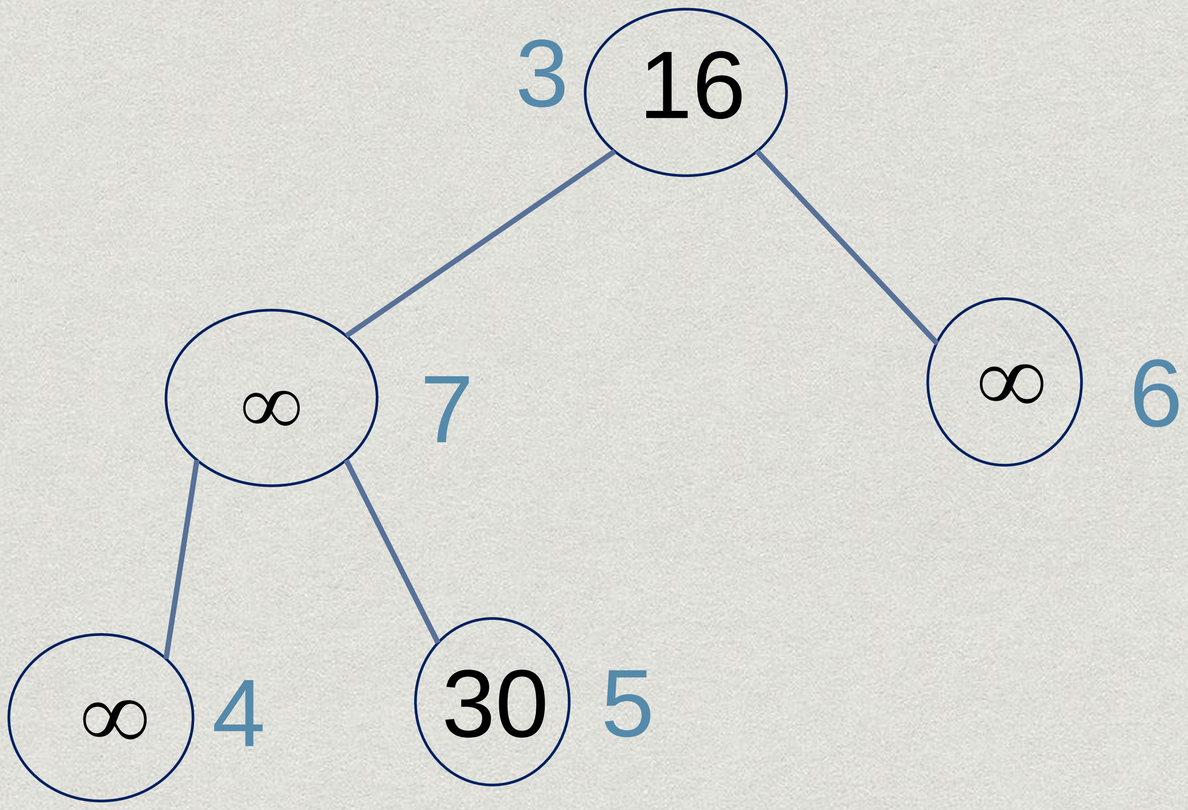




Do a distance update for node 2

Revisit Dijkstra's algorithm

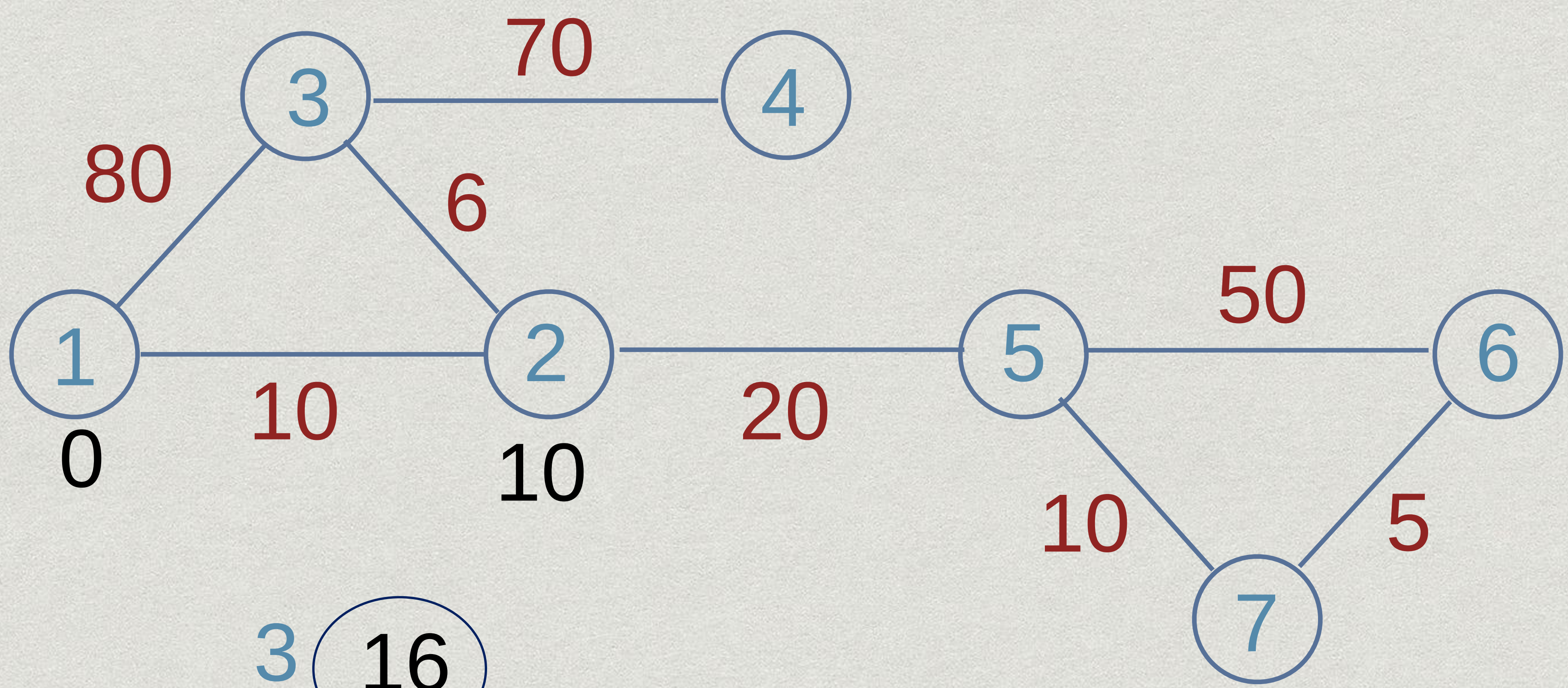


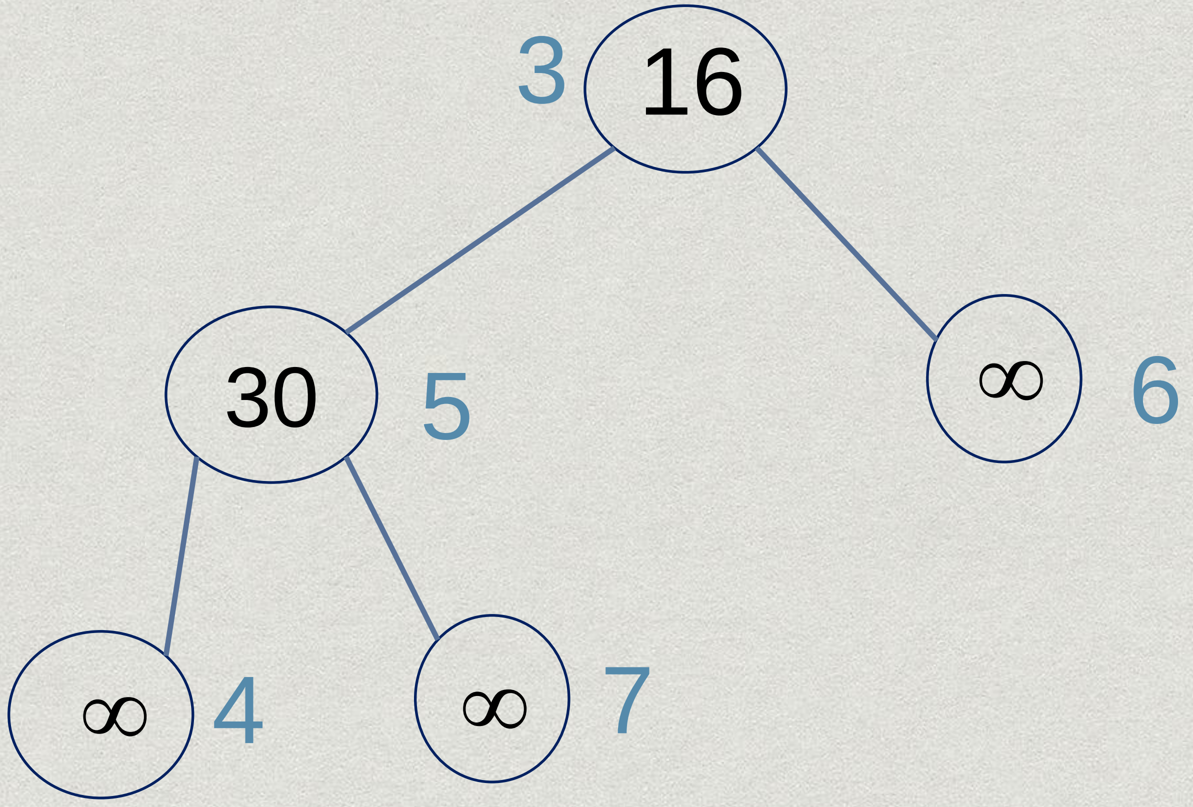


Do a distance update for node 2

Heapify_up for nodes 3 (i=1) and 5 (i=5)

Revisit Dijkstra's algorithm

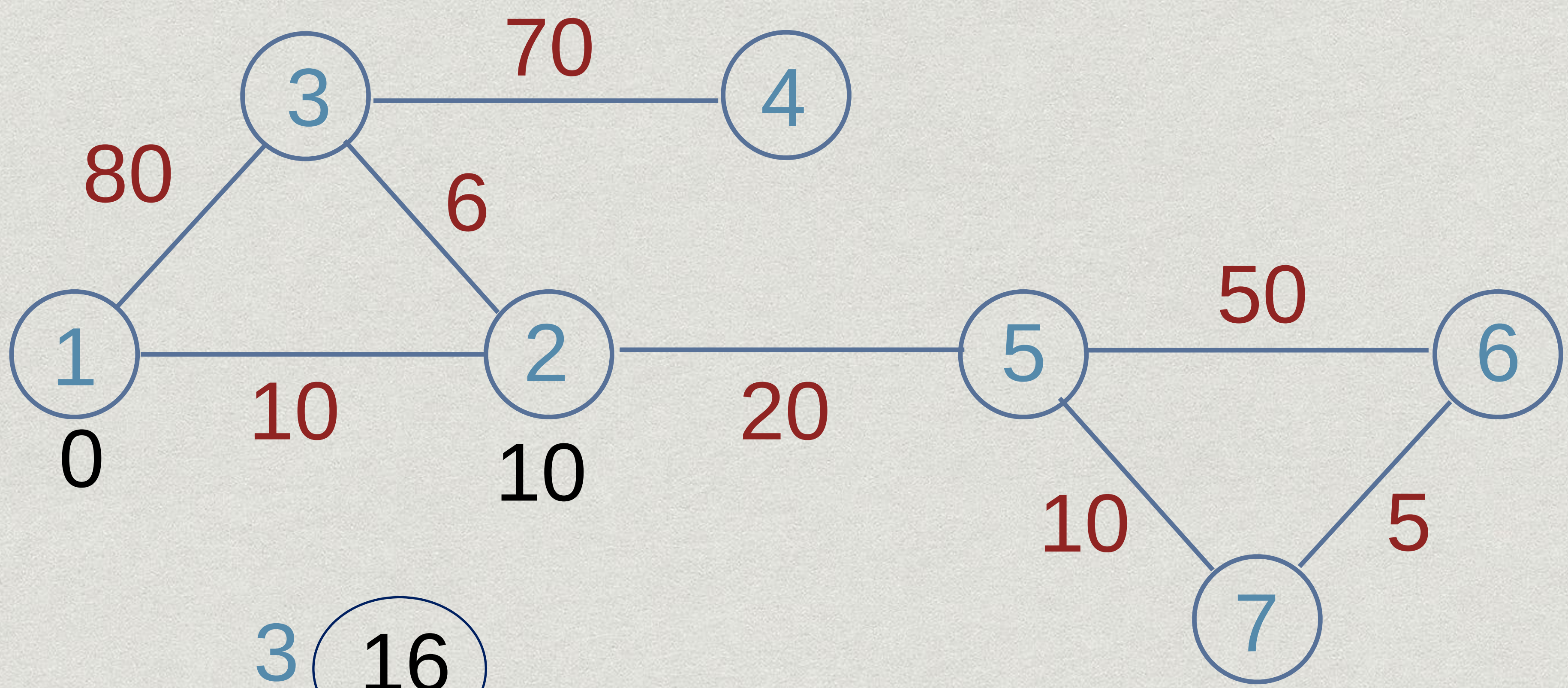


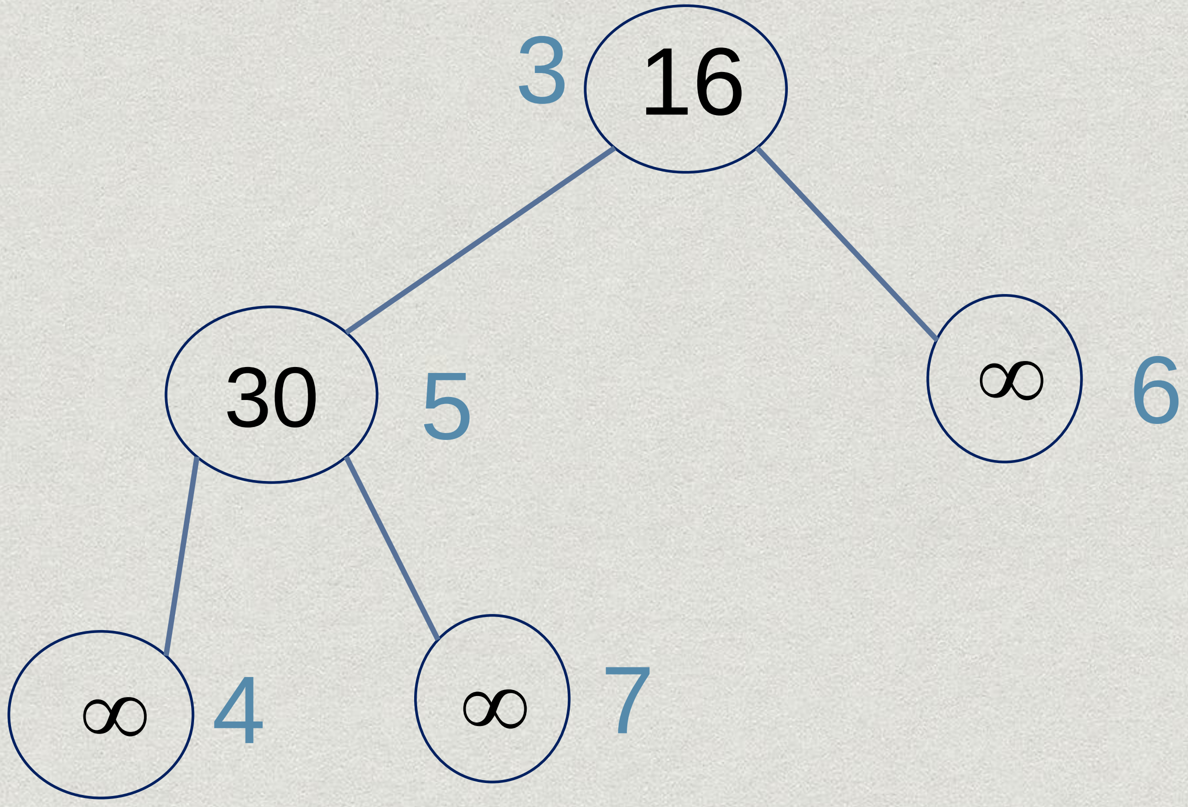


Do a distance update for node 2

Heapify_up for i=5 produces a heap

Revisit Dijkstra’s algorithm

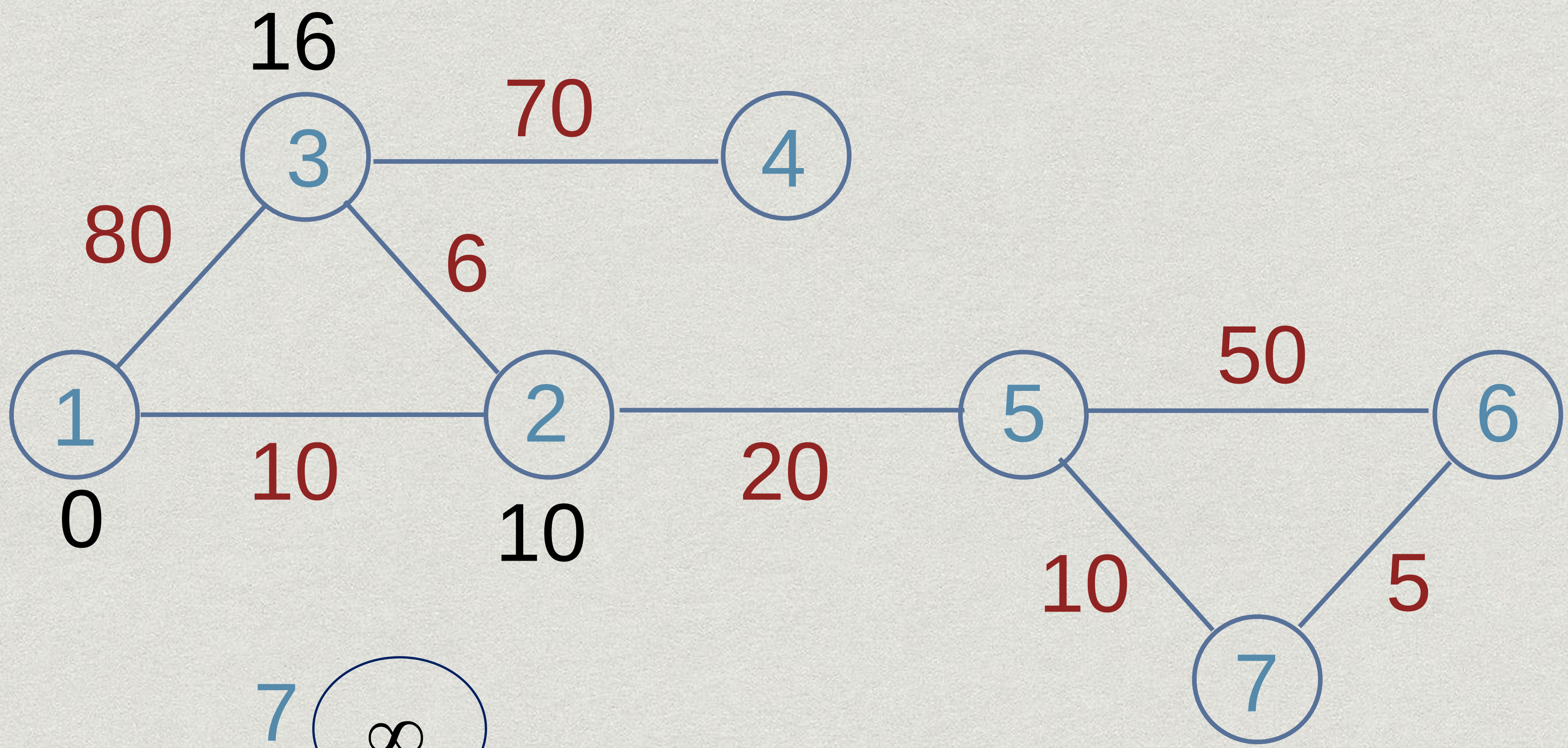


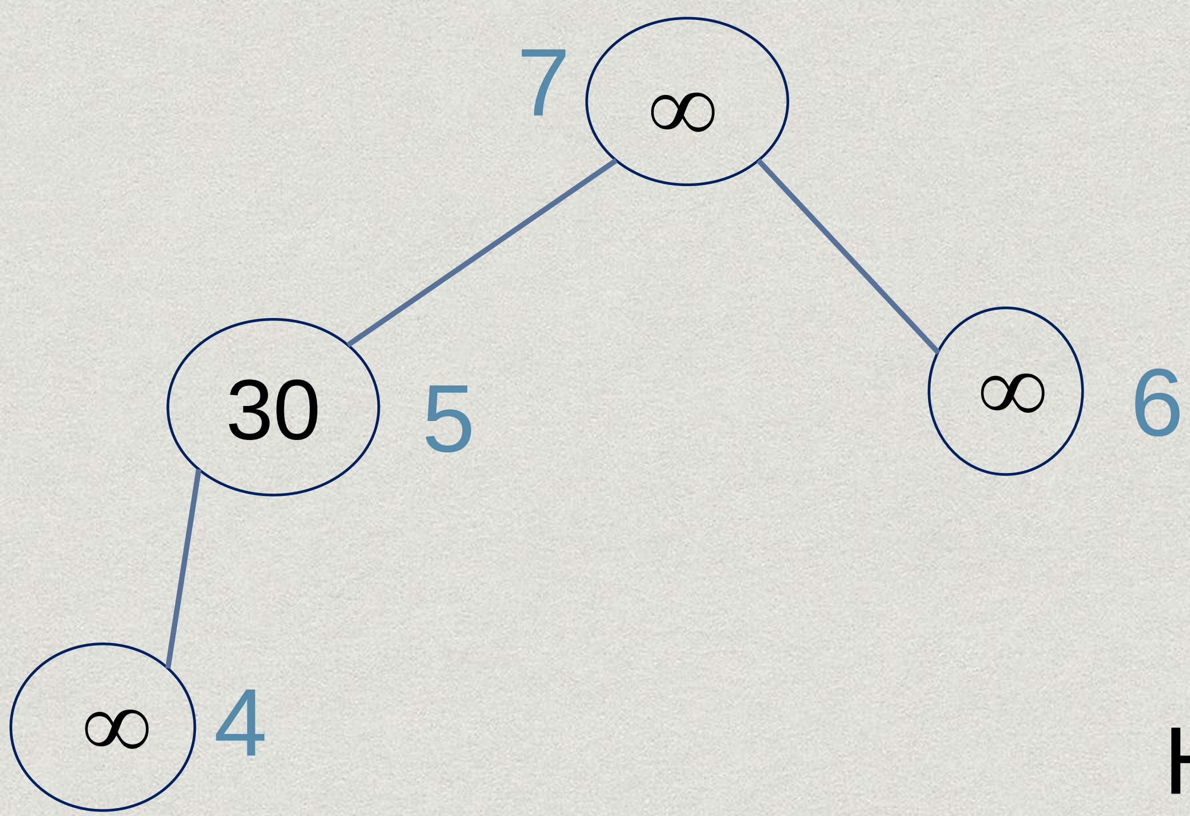


Select node 3

Delete node 3
It is replaced by node 7

Revisit Dijkstra’s algorithm



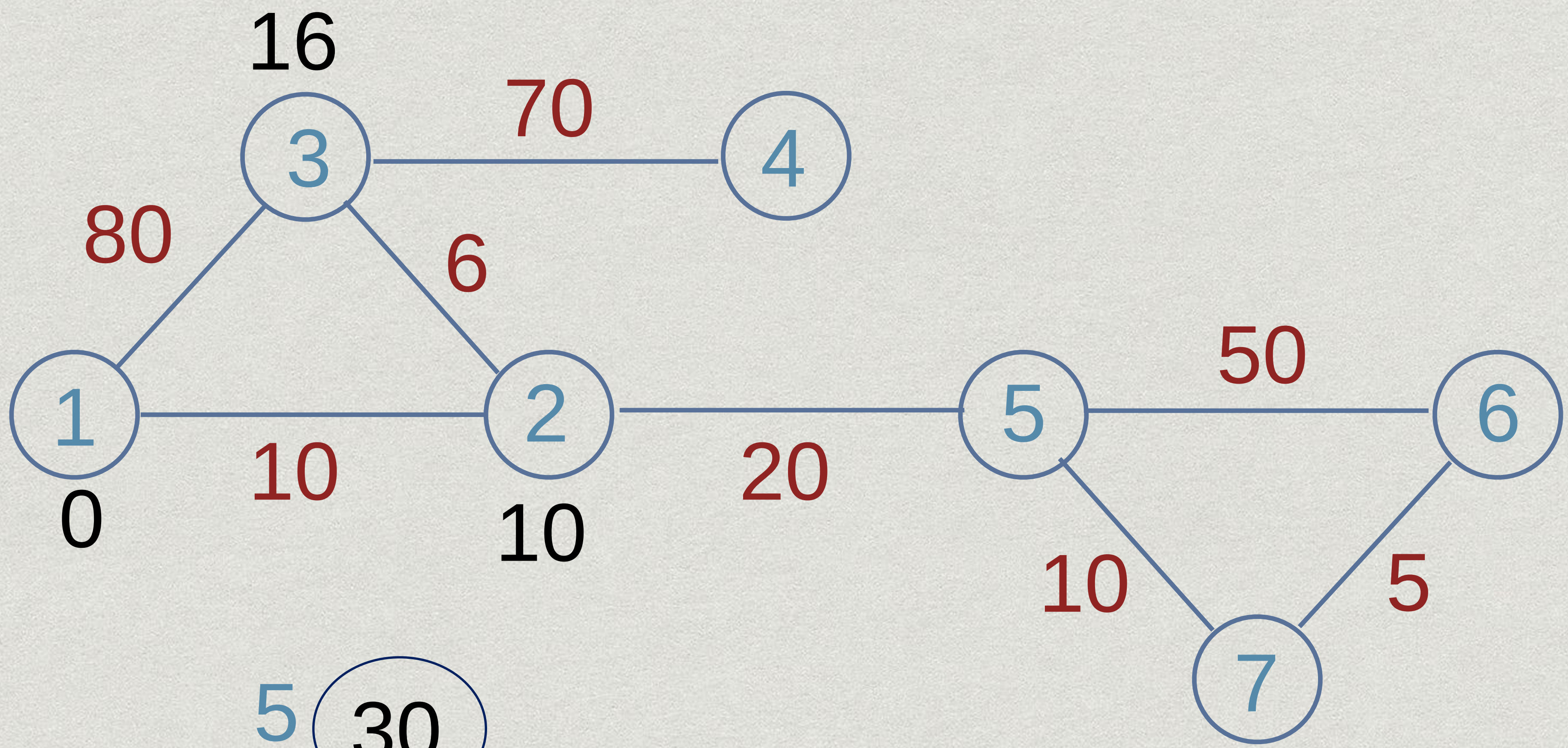


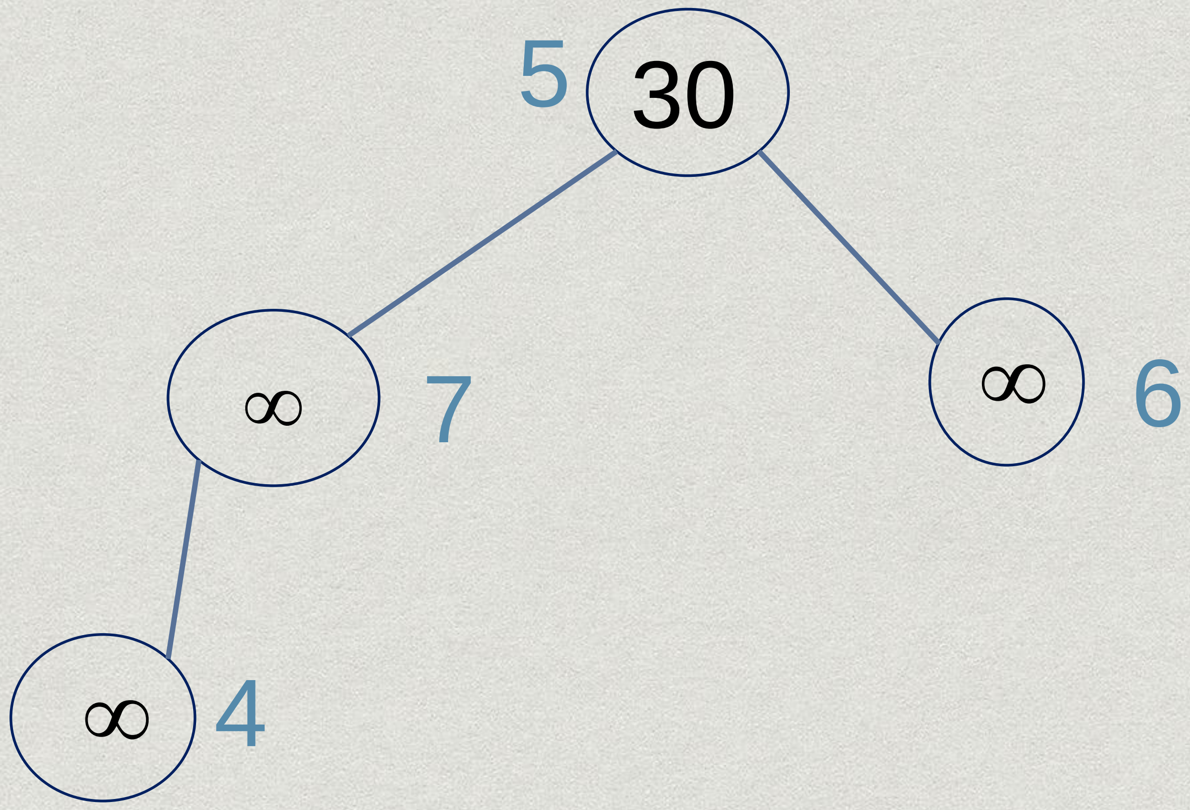
Select node 3

Delete node 3
It is replaced by node 7

Heapify_down required for i=1

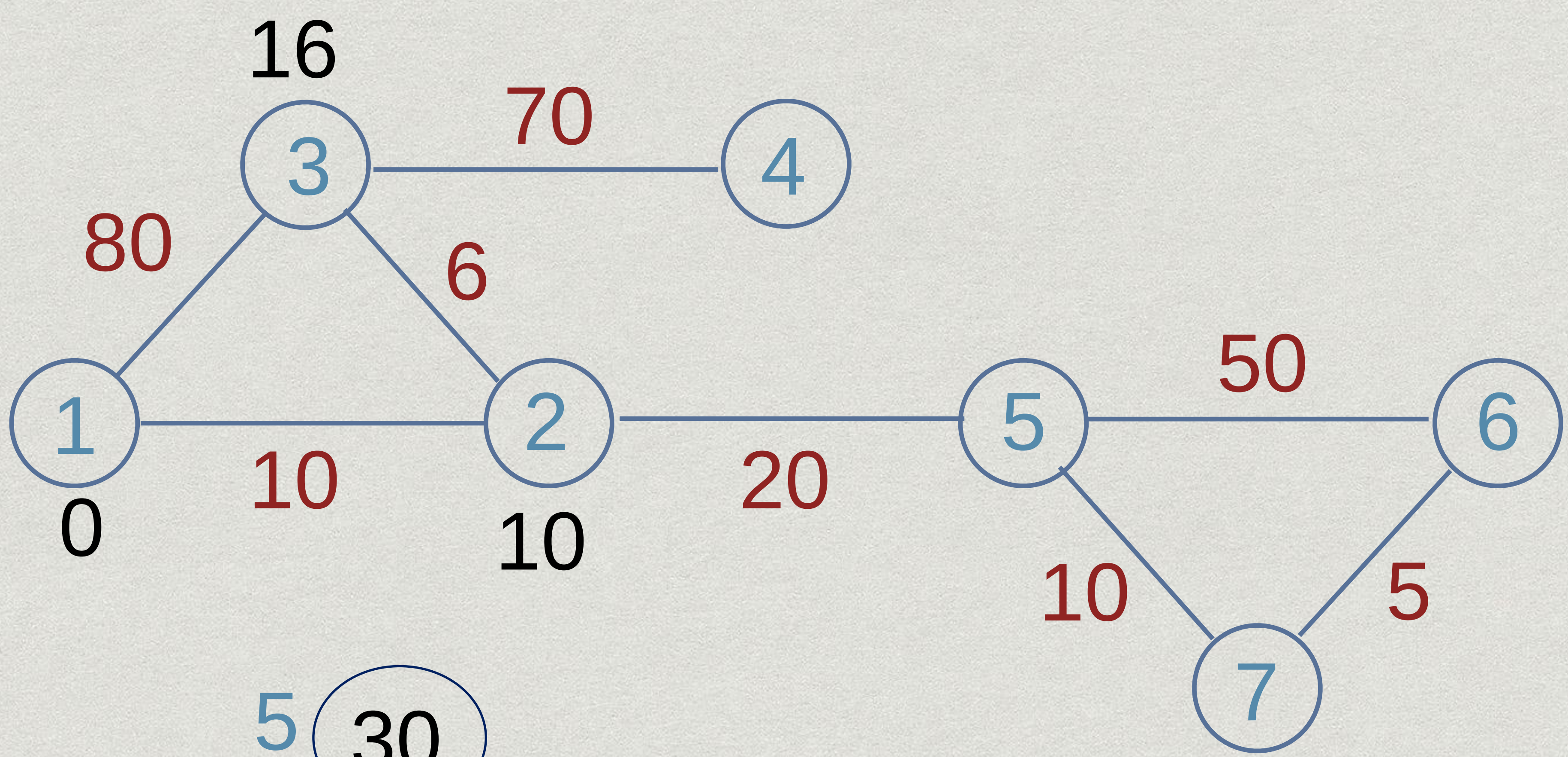
Revisit Dijkstra's algorithm

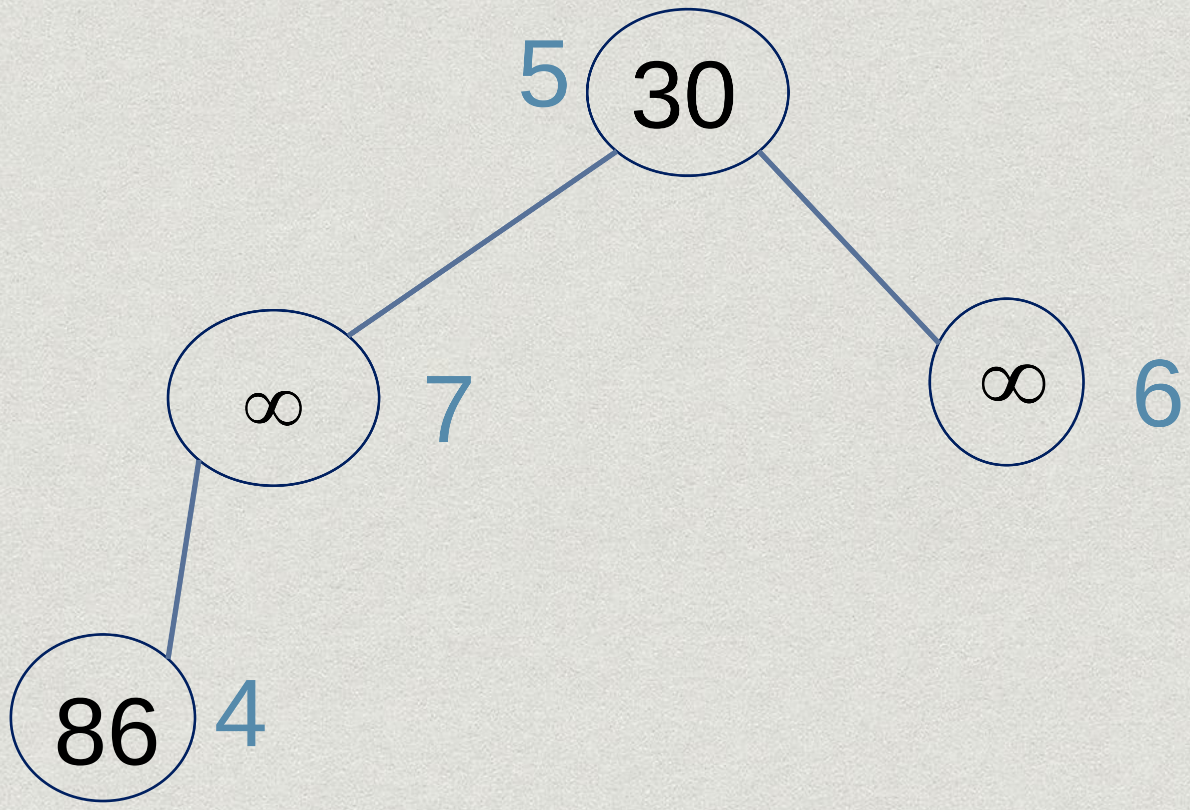




Distance update for node 3

Revisit Dijkstra's algorithm

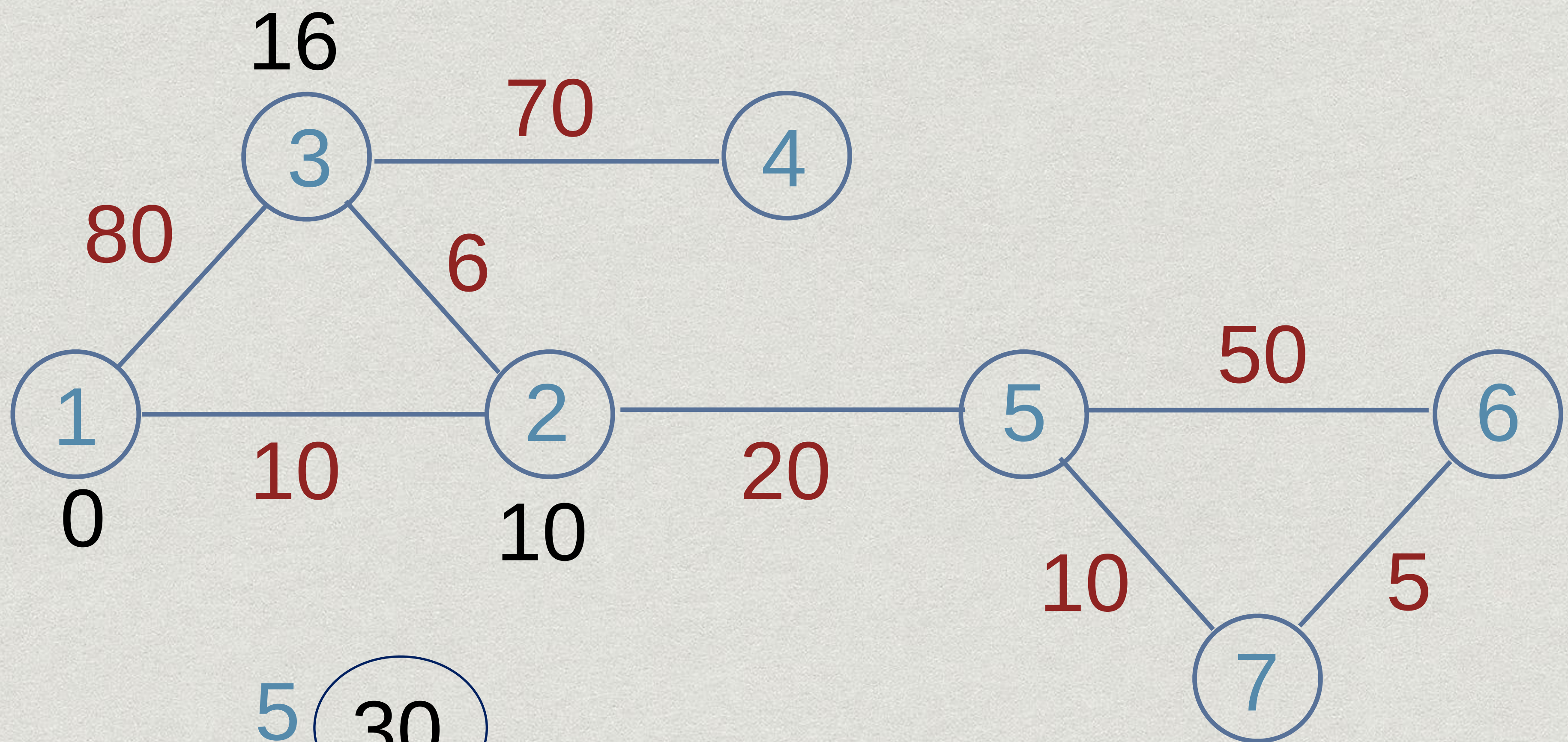


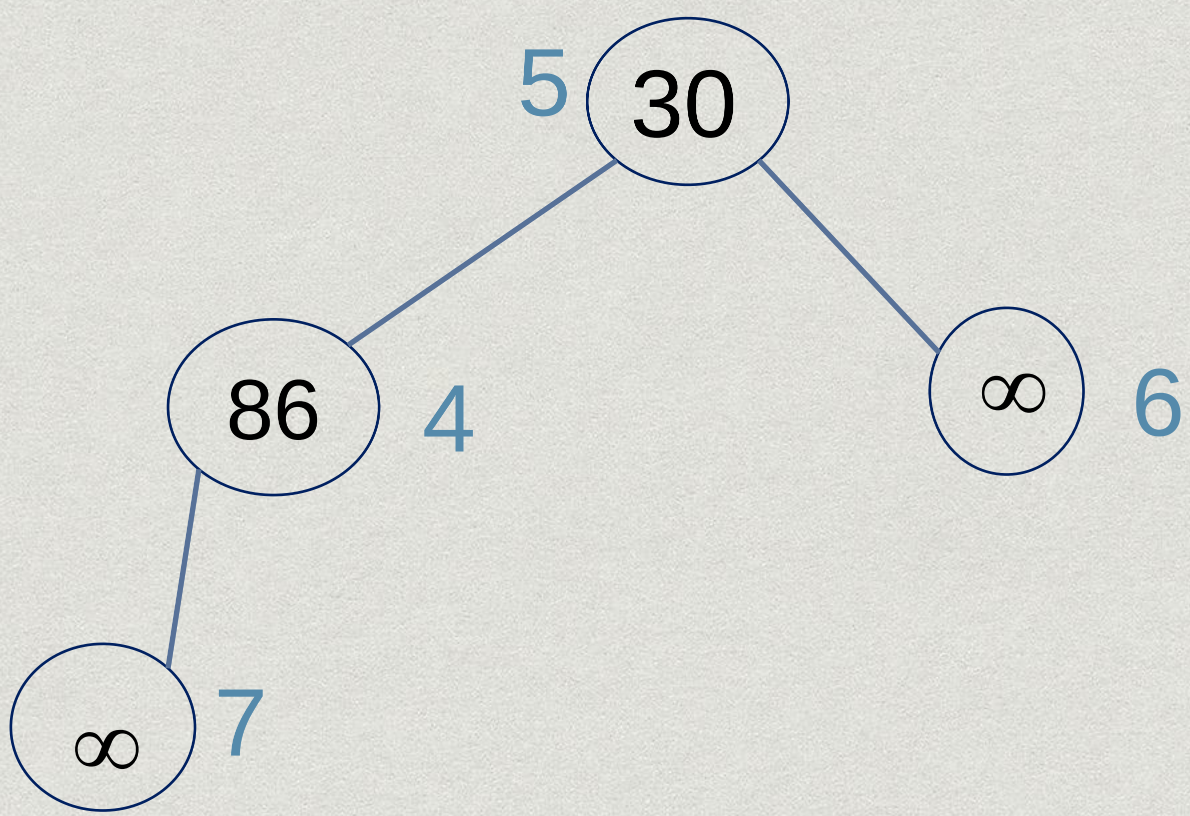


Distance update for node 3

Heapify_up required for i=4

Revisit Dijkstra's algorithm

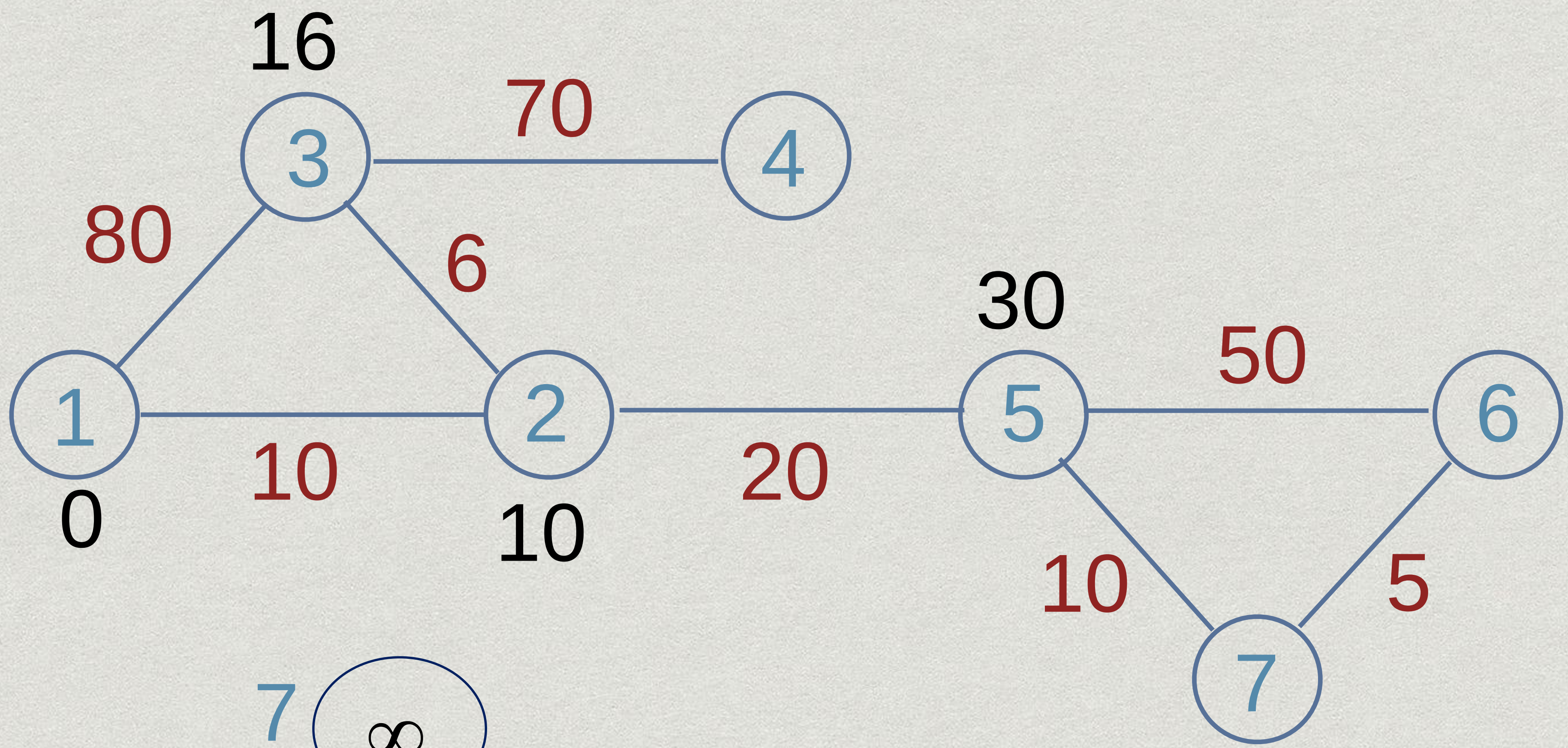


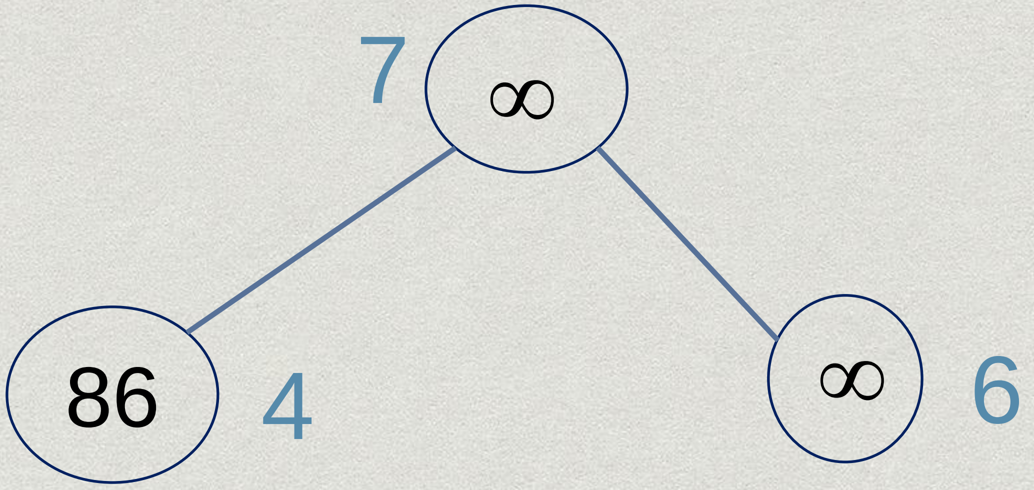


Distance update for node 3

Heapify_up required for i=4

Revisit Dijkstra's algorithm



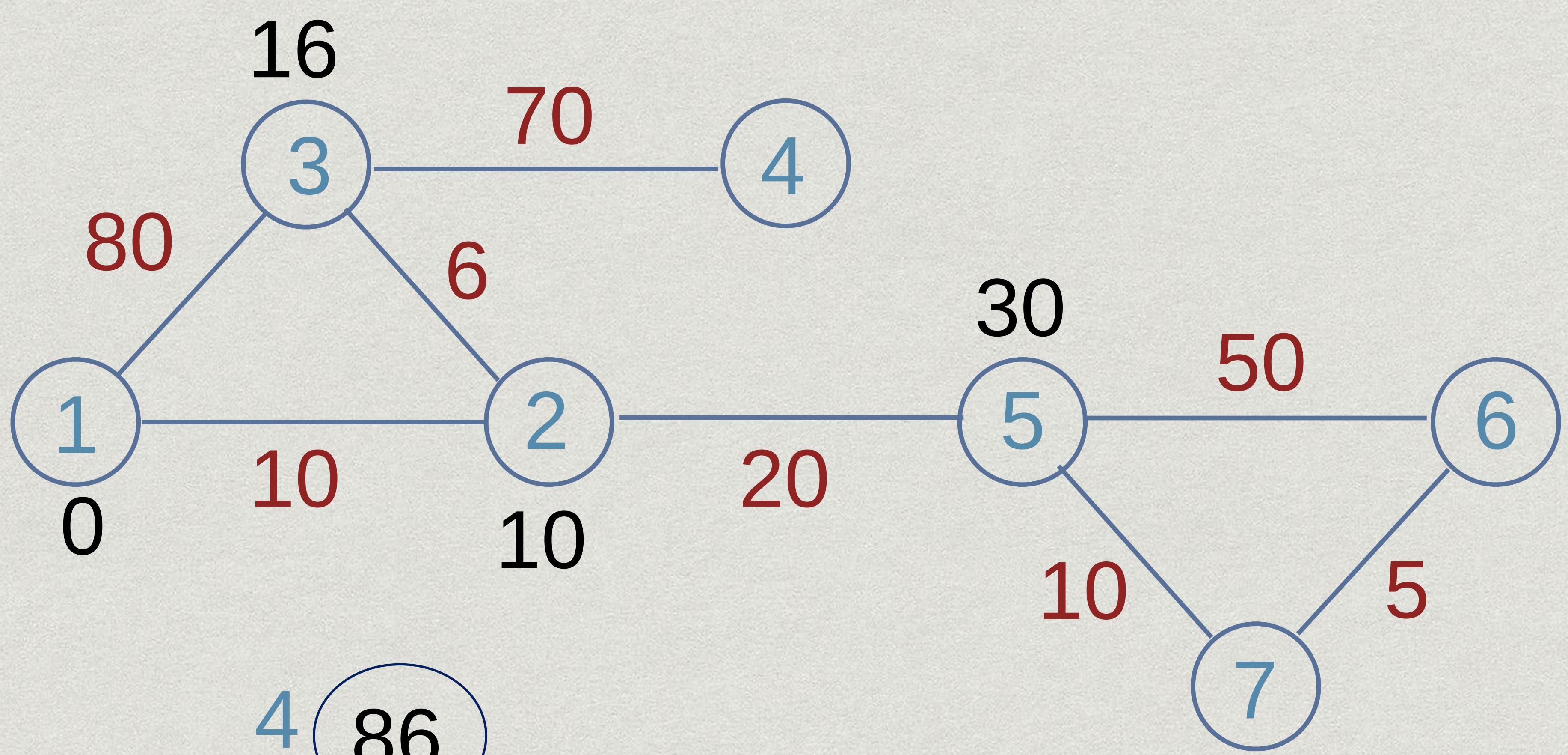


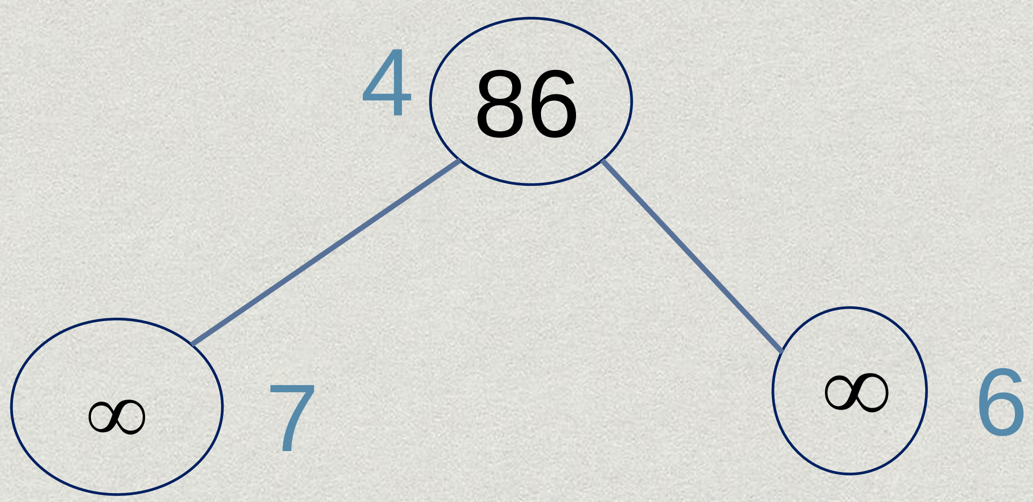
Select node 5

Delete node 5, it gets replaced by node 7

Heapify_down required for i=1

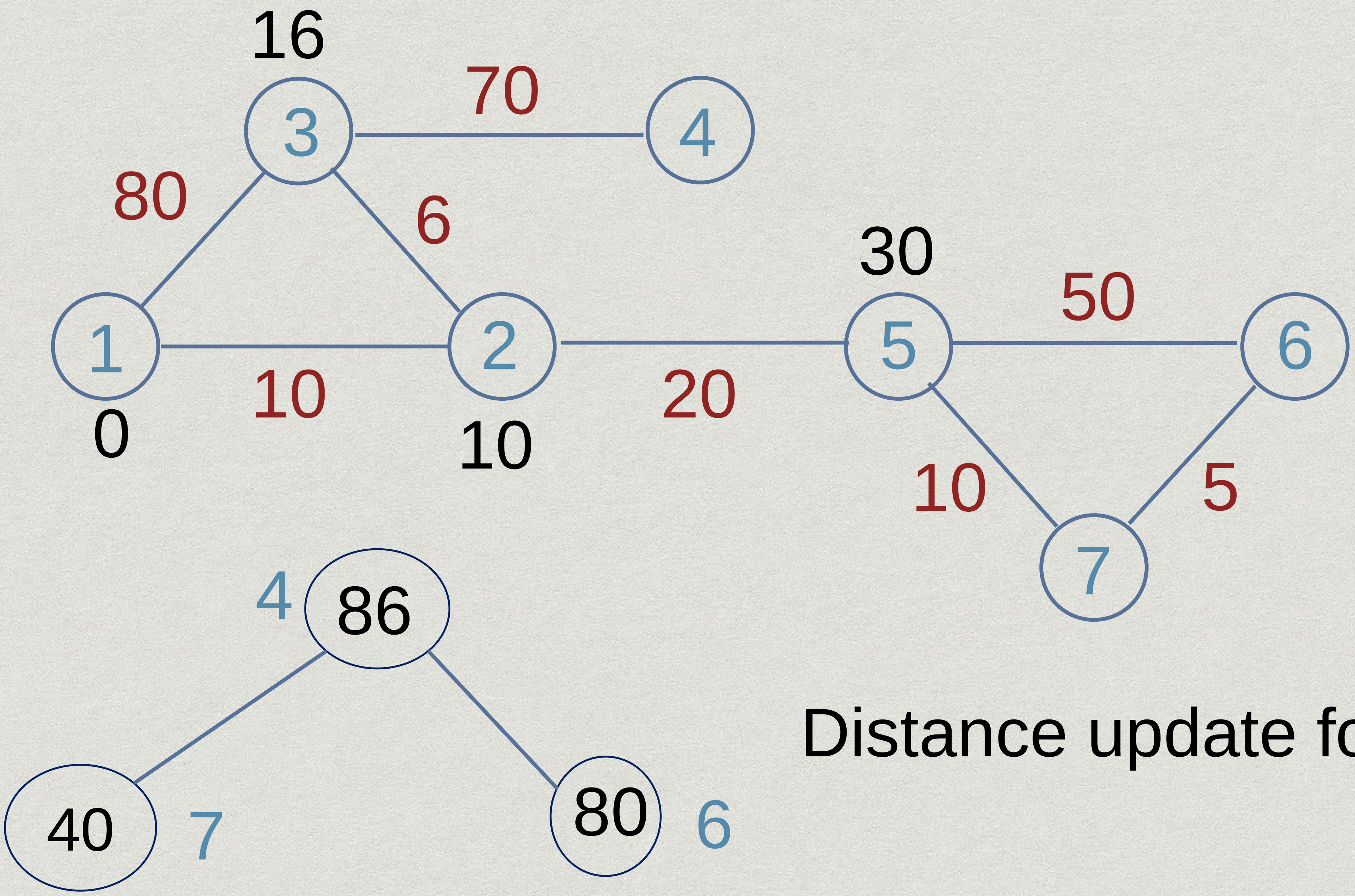
Revisit Dijkstra's algorithm





Distance update for node 5

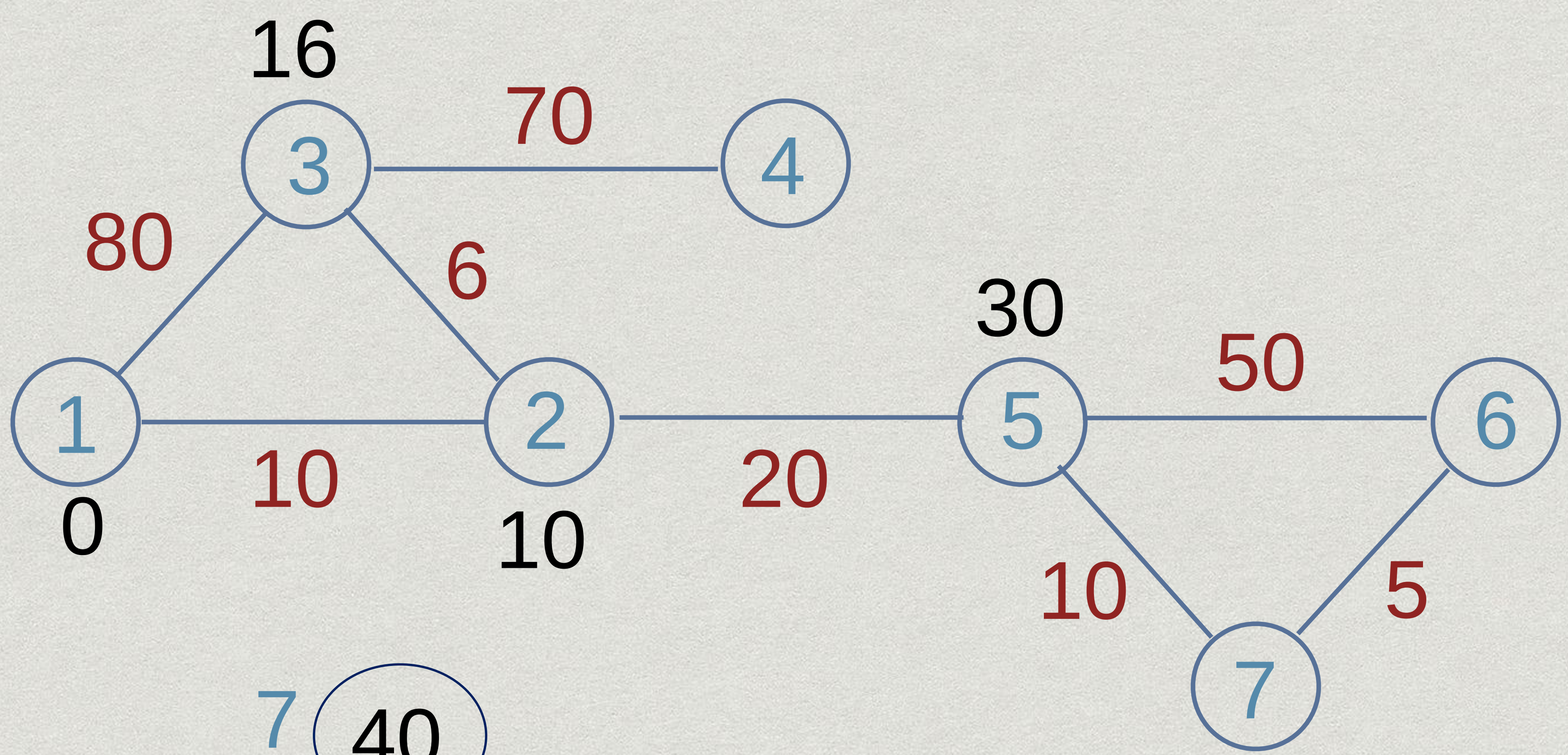
Revisit Dijkstra's algorithm

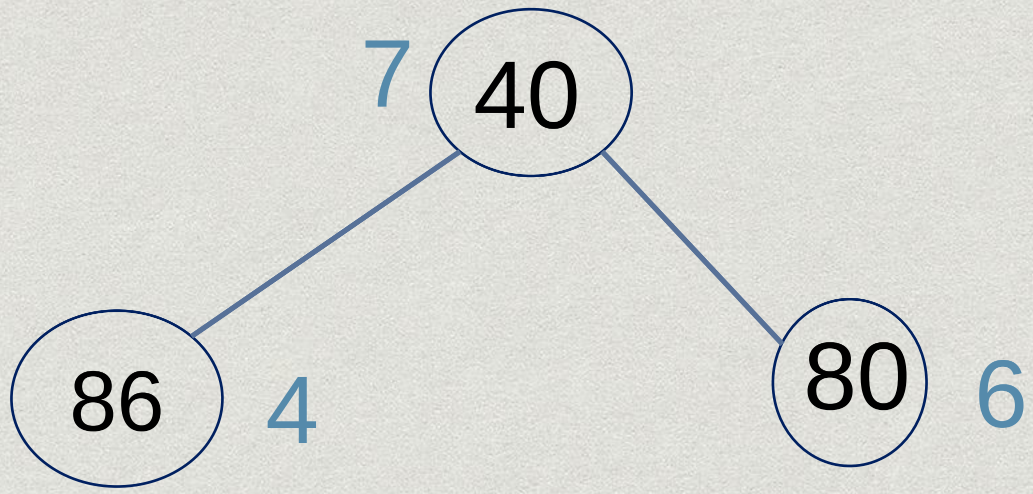


Distance update for node 5

Both i=2 and i=3 need heapify_up

Revisit Dijkstra's algorithm

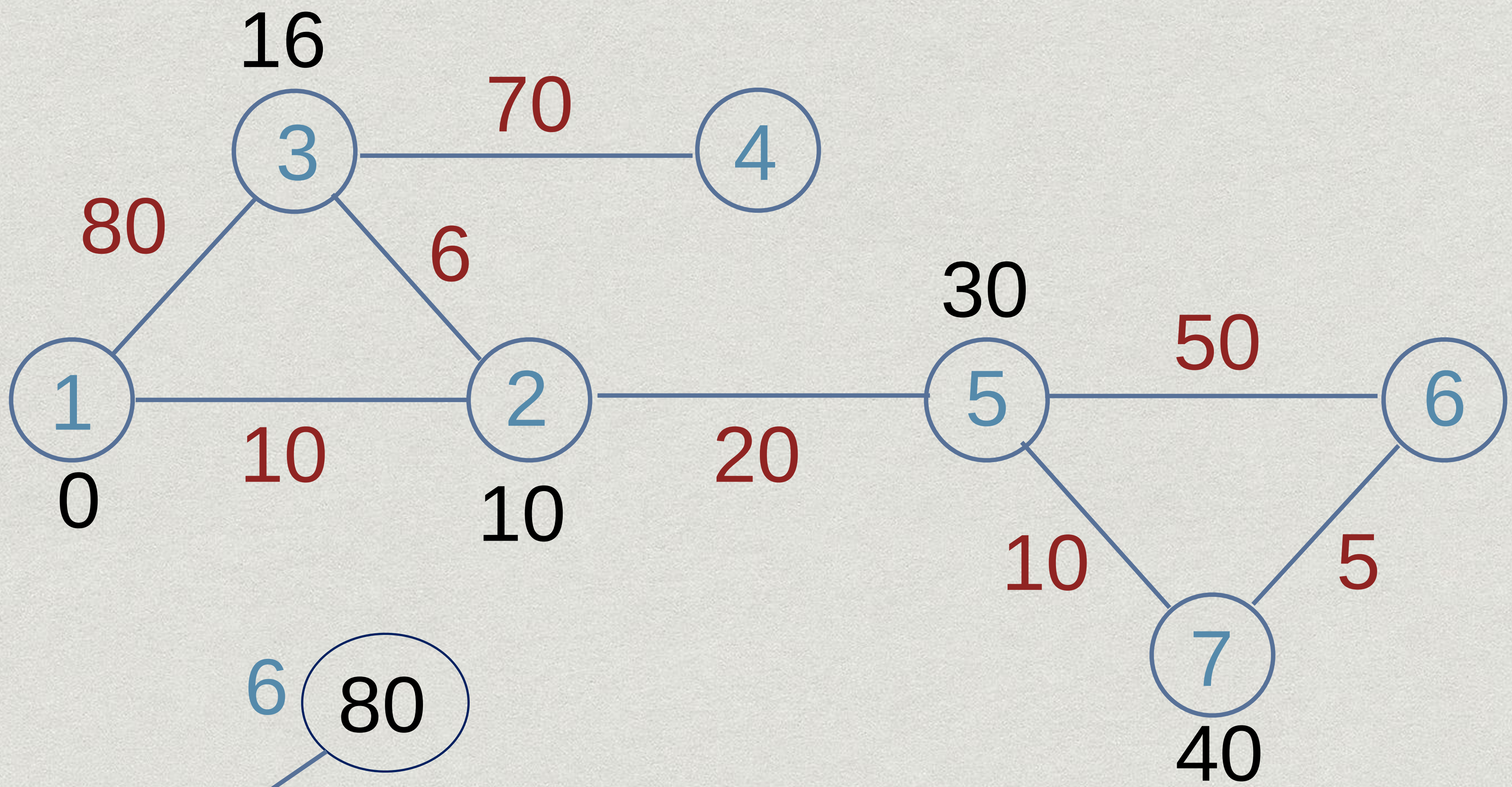


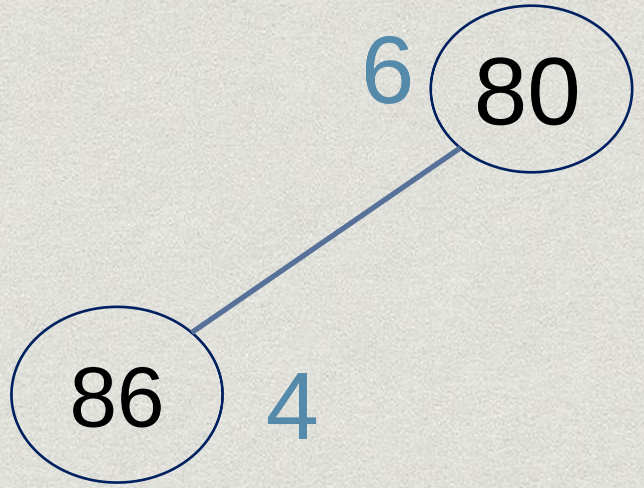


Select node 7

Delete node 7,
it gets replaced
by node 6

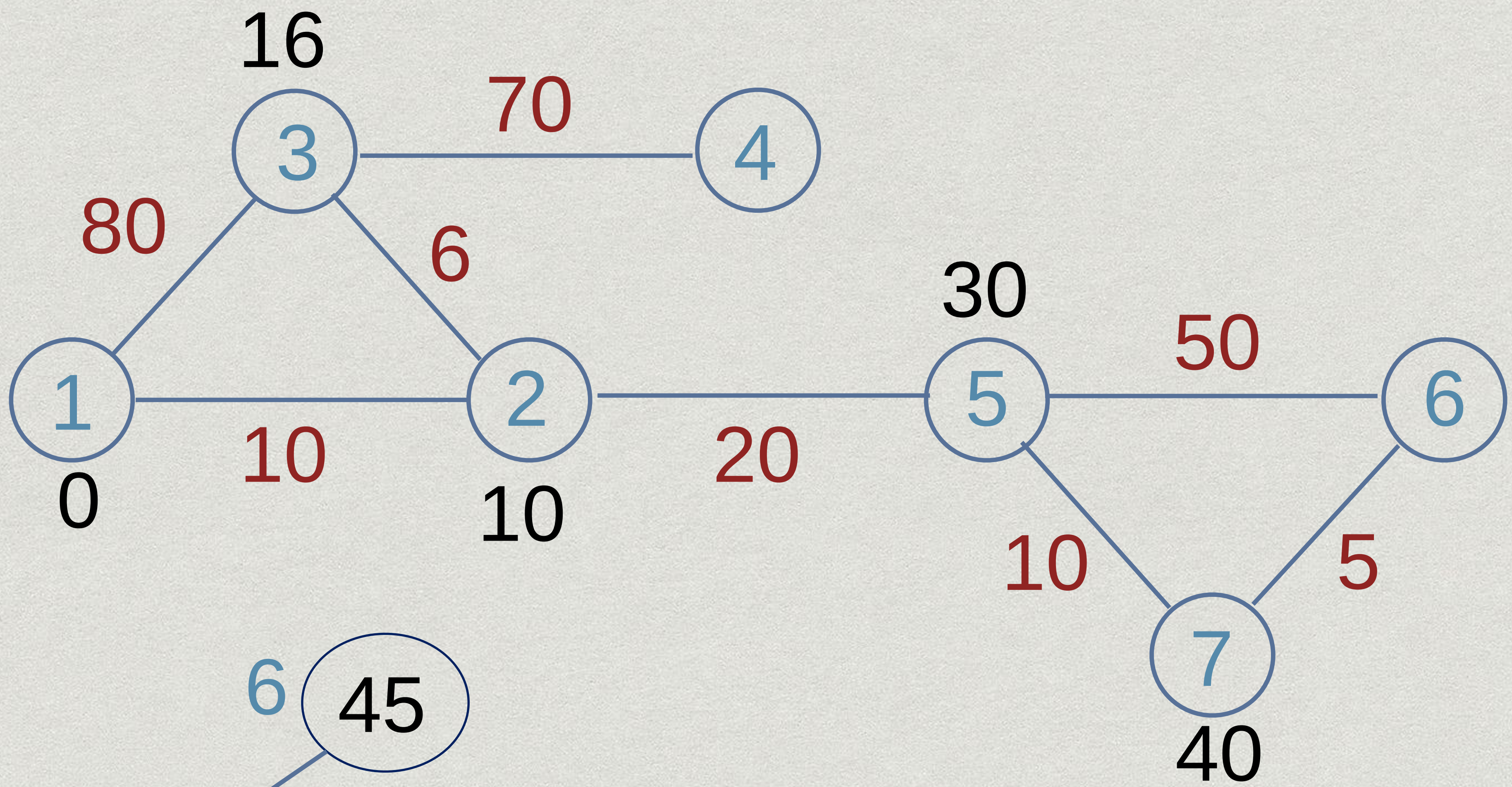
Revisit Dijkstra's algorithm

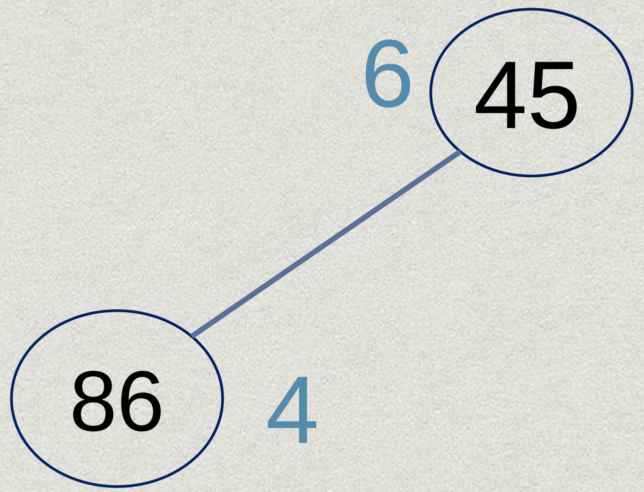




Distance update for node 7

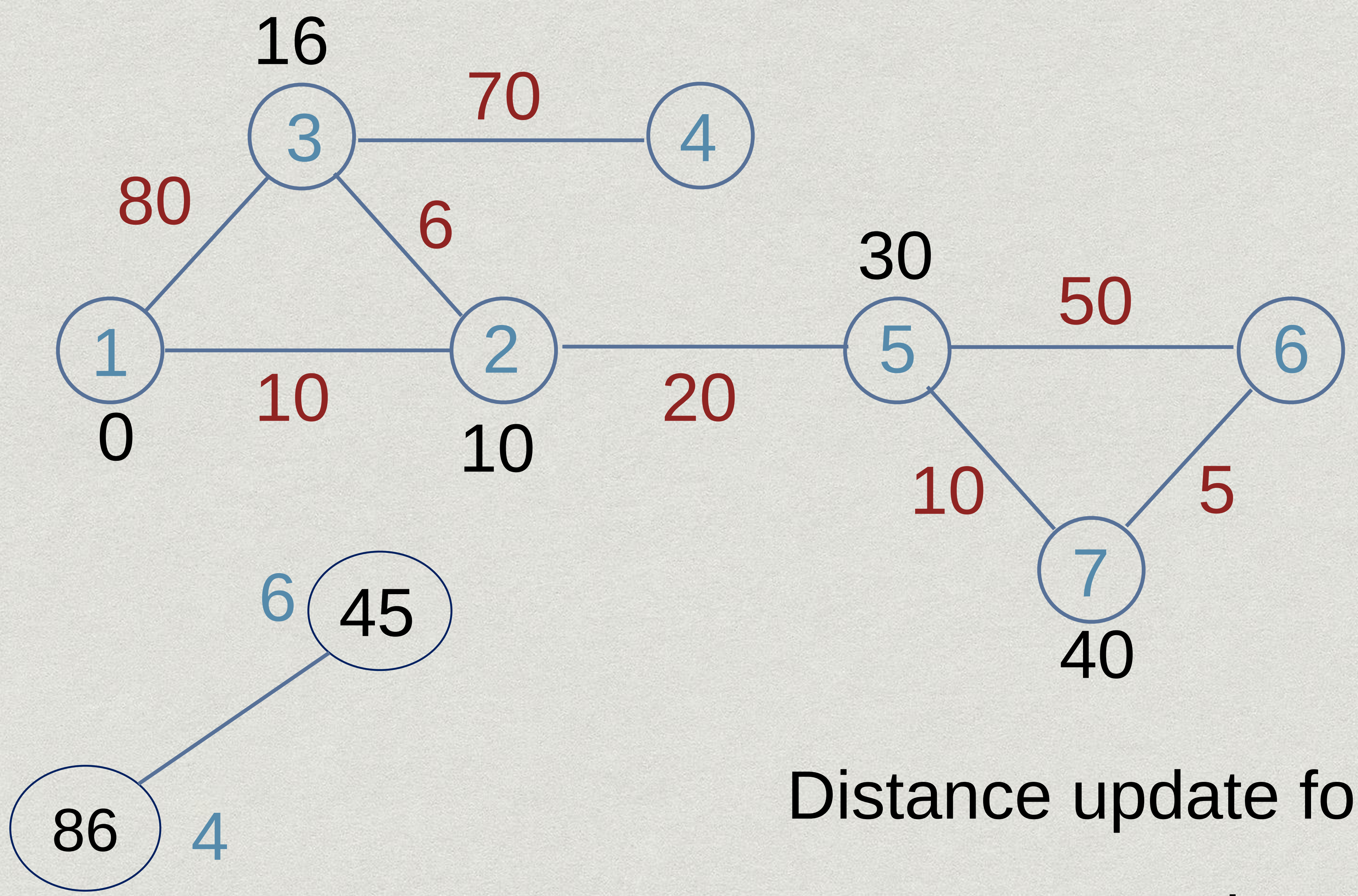
Revisit Dijkstra's algorithm





Distance update for node 7

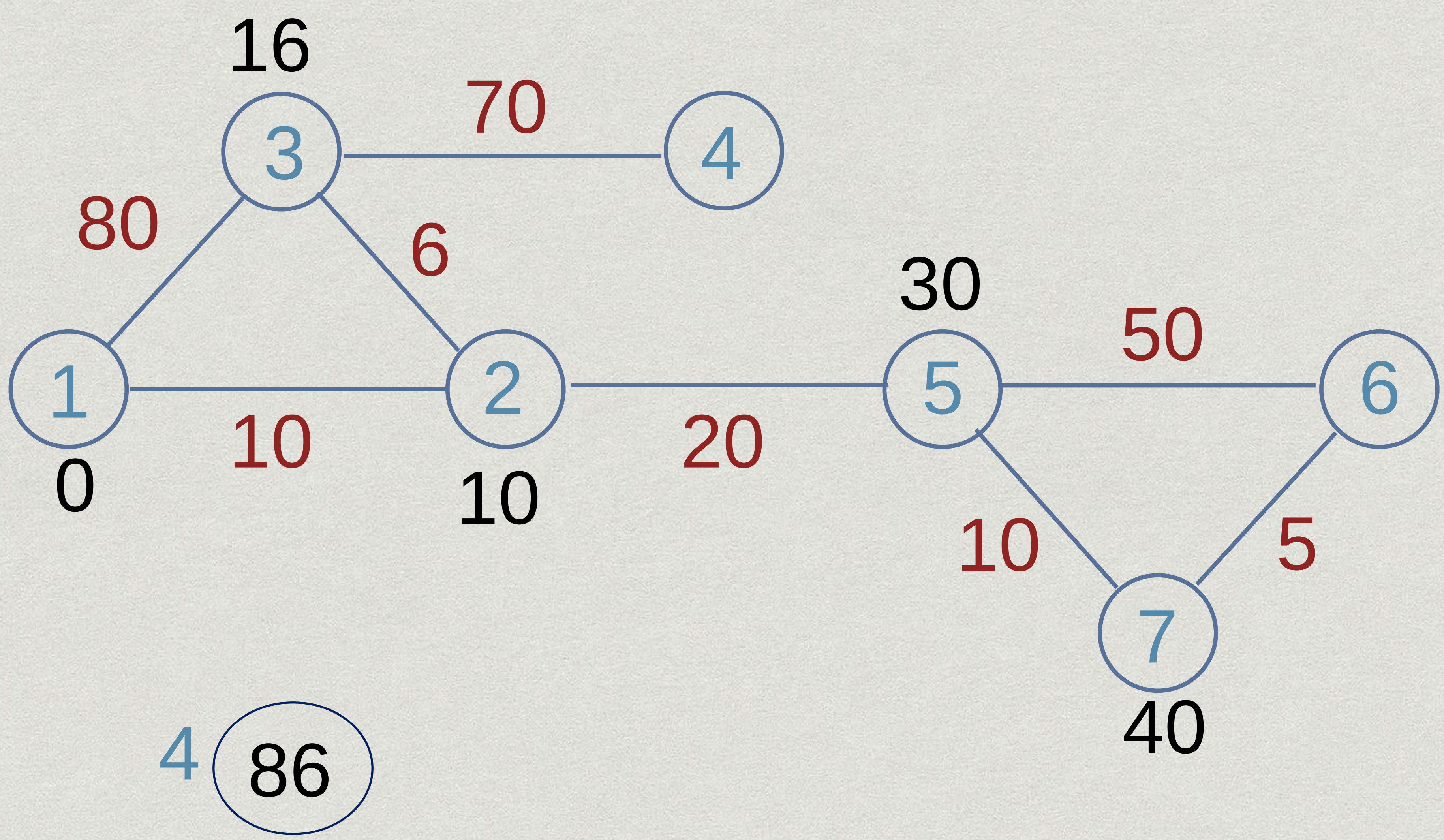
Revisit Dijkstra's algorithm



Distance update for node 7

Heap not damaged

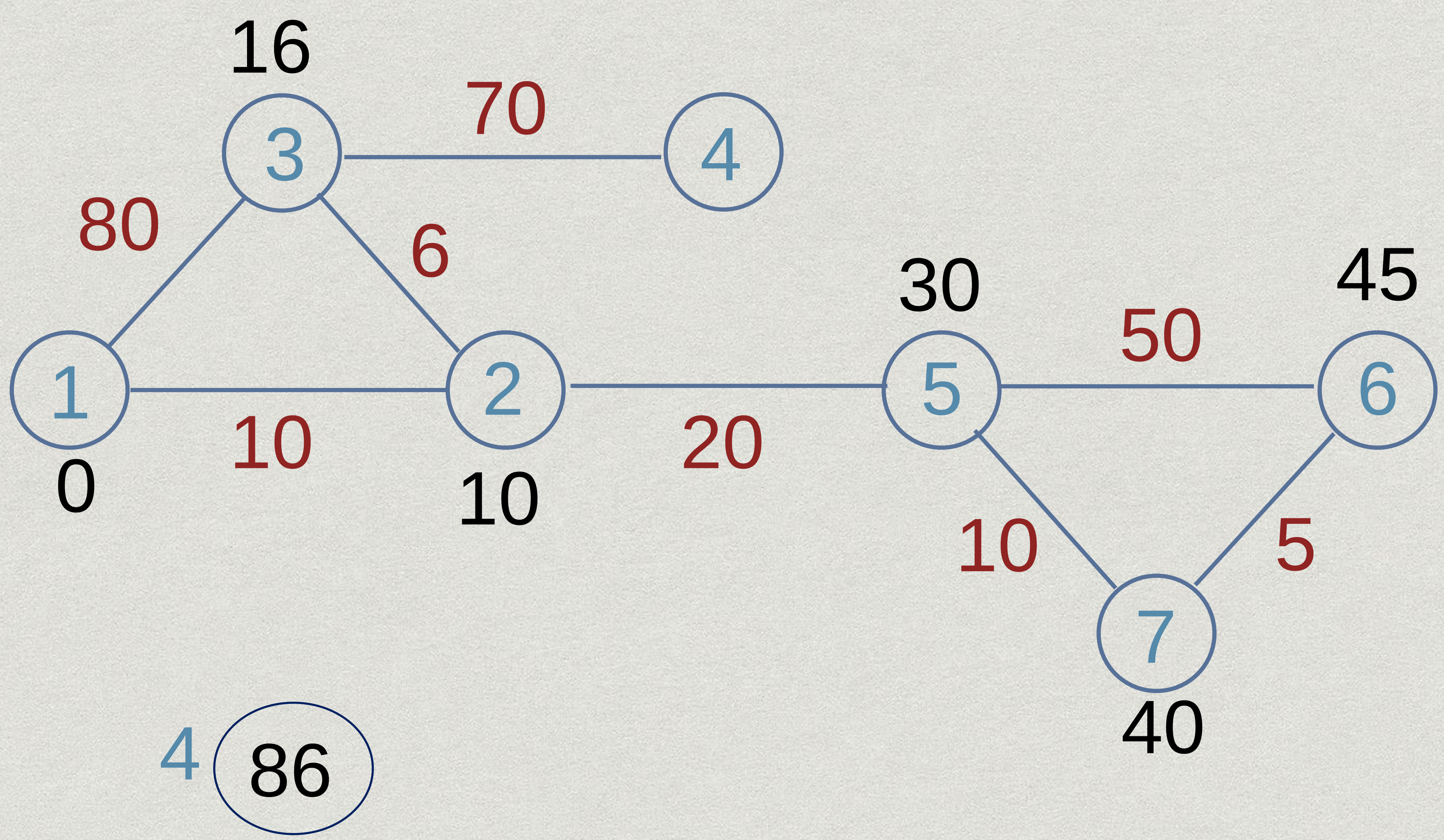
Revisit Dijkstra's algorithm



Select node 6

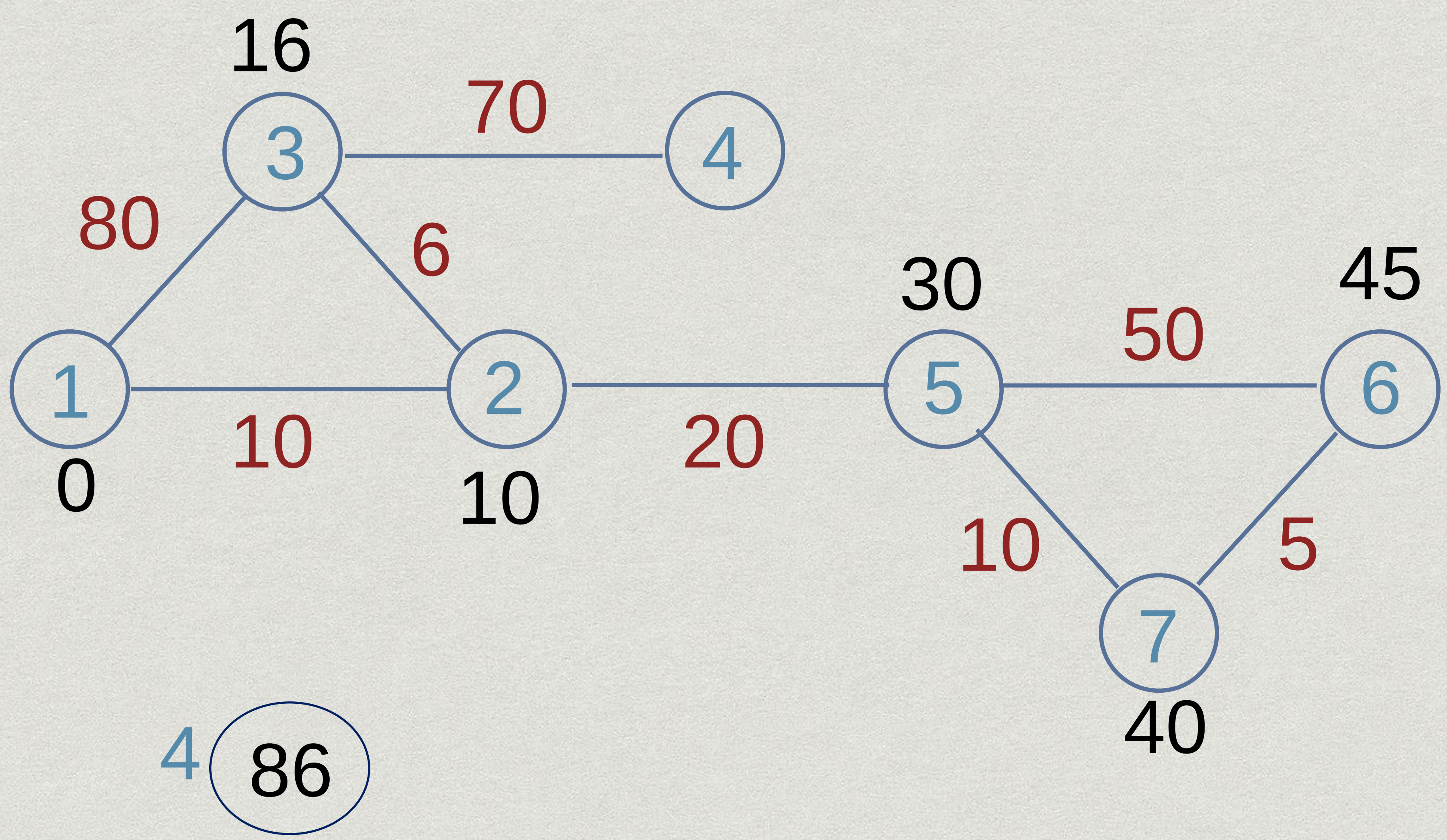
Delete node 6,
it gets replaced by node 4

Revisit Dijkstra's algorithm



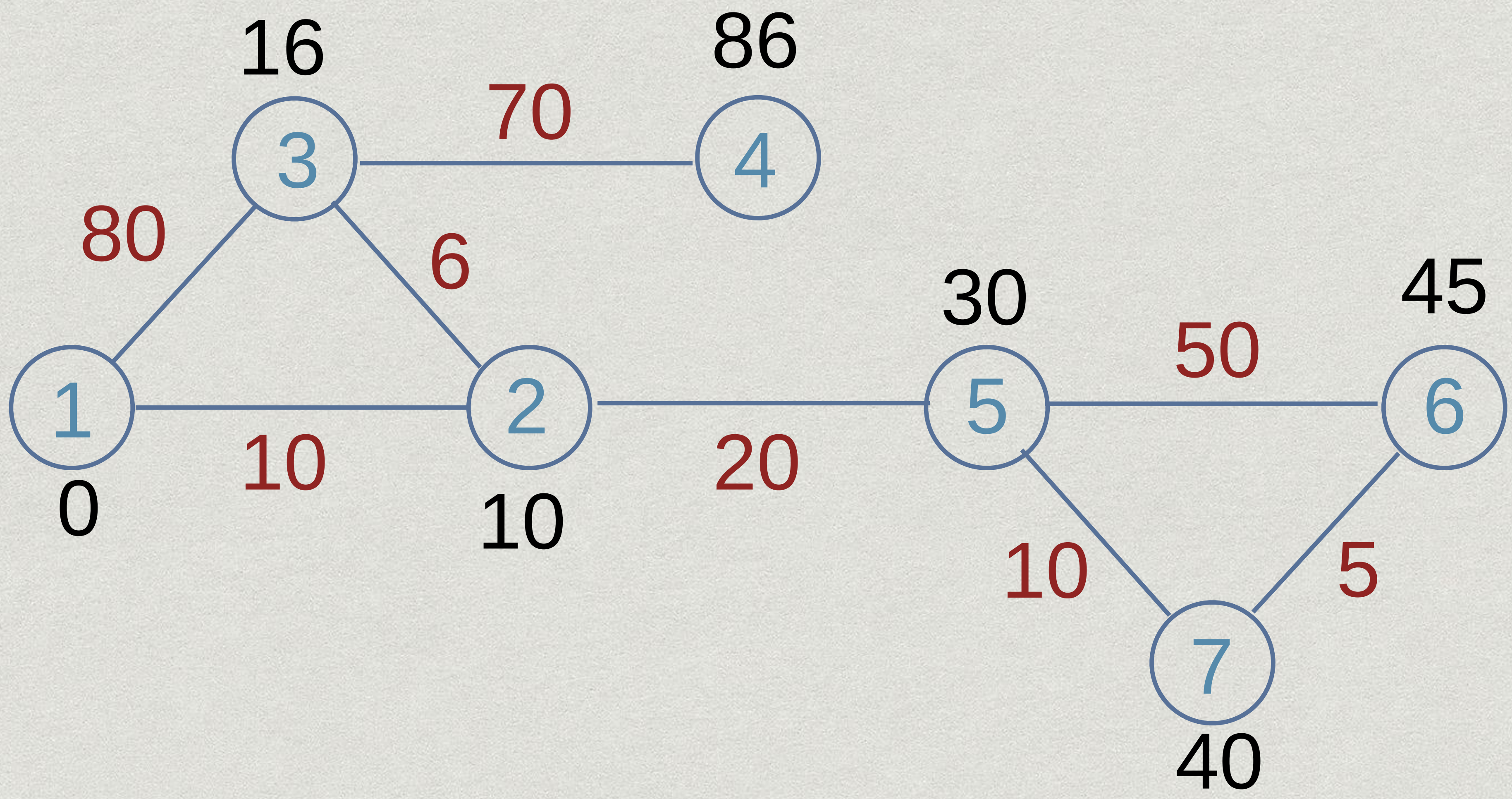
Do a distance update for node 6

Revisit Dijkstra's algorithm



Select node 4
and delete it

Revisit Dijkstra's algorithm



Select node 4
and delete it

Improving Dijkstra's algorithm

Key steps in the algorithm:

- Select an unvisited node u with the least Distance value
  - This can be done in $O(1)$ time using a heap

- Remove u from the list of unvisited nodes
  - This can be done in $O(\log n)$ time using a heap:
    delete the root and heapify_down

- Access all neighbours of u and update distance
  - can be done in $O(\log n)$ time per edge using a heap:
    change_key and then heapify_up
    (note distance only reduces – never increases)

- Overall complexity then becomes $O((n+m)\log n)$

Summary

- Priority queues are data structures used for maintaining a set of elements each of which has a key value
  - operations: add an element, delete an element, select element with lowest key value

- Priority queues can be implemented using heaps – in which case, each of the operations above can be done in $\log n$ time

- Revisit Dijkstra's algorithm. Use priority queues to reduce complexity to $O((n+m)\log n)$