QEEE DSA05
DATA STRUCTURES AND ALGORITHMS

G VENKATESH AND MADHAVAN MUKUND

LECTURE 10, 5 SEPTEMBER 2014

Comparing data structures

What is the difference between heap and array, especially if heap is implemented using array?

Comparing data structures

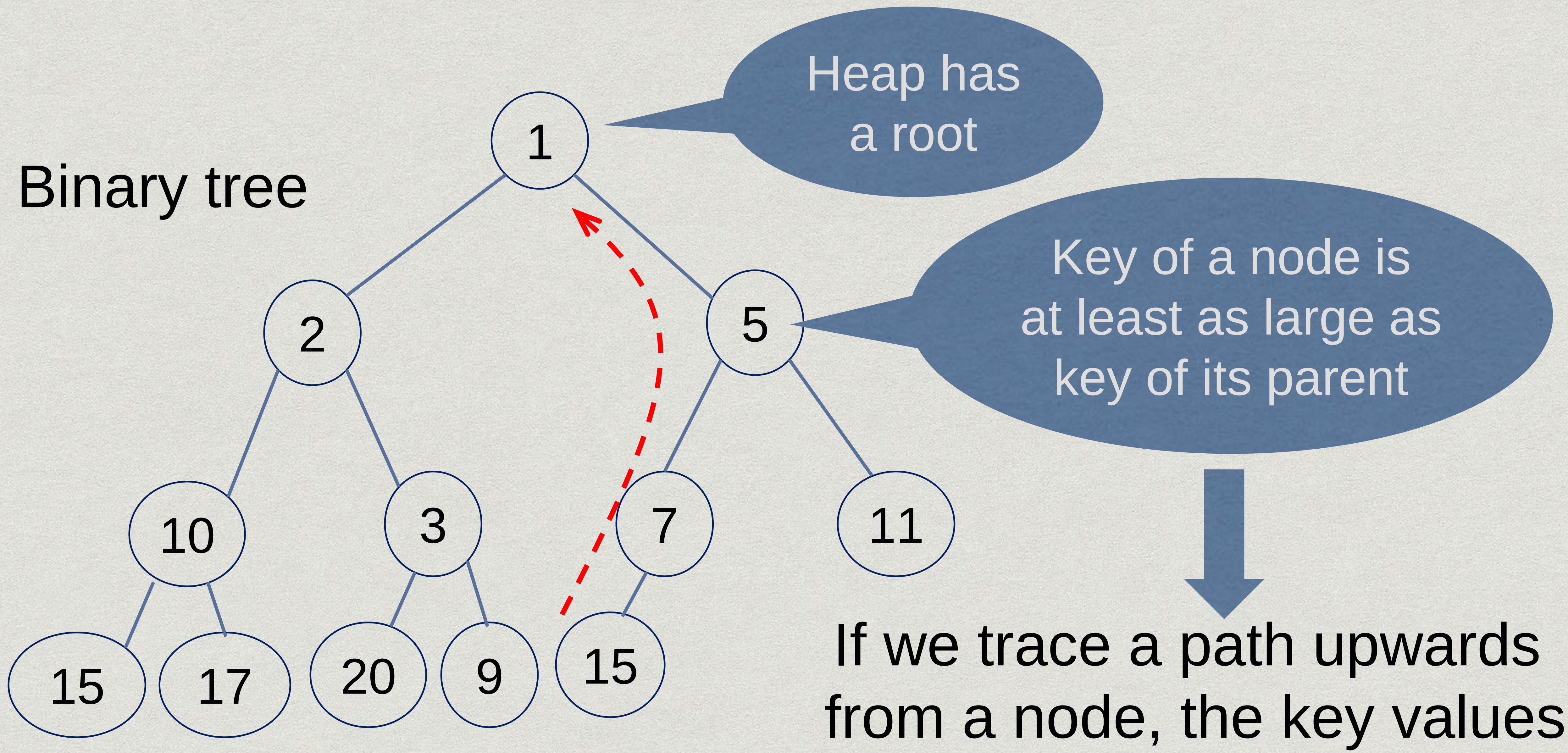
What is the difference between heap and array, especially if heap is implemented using array?

Array: Not arranged in any order

Sorted array: Elements always stored in order (either ascending or descending)

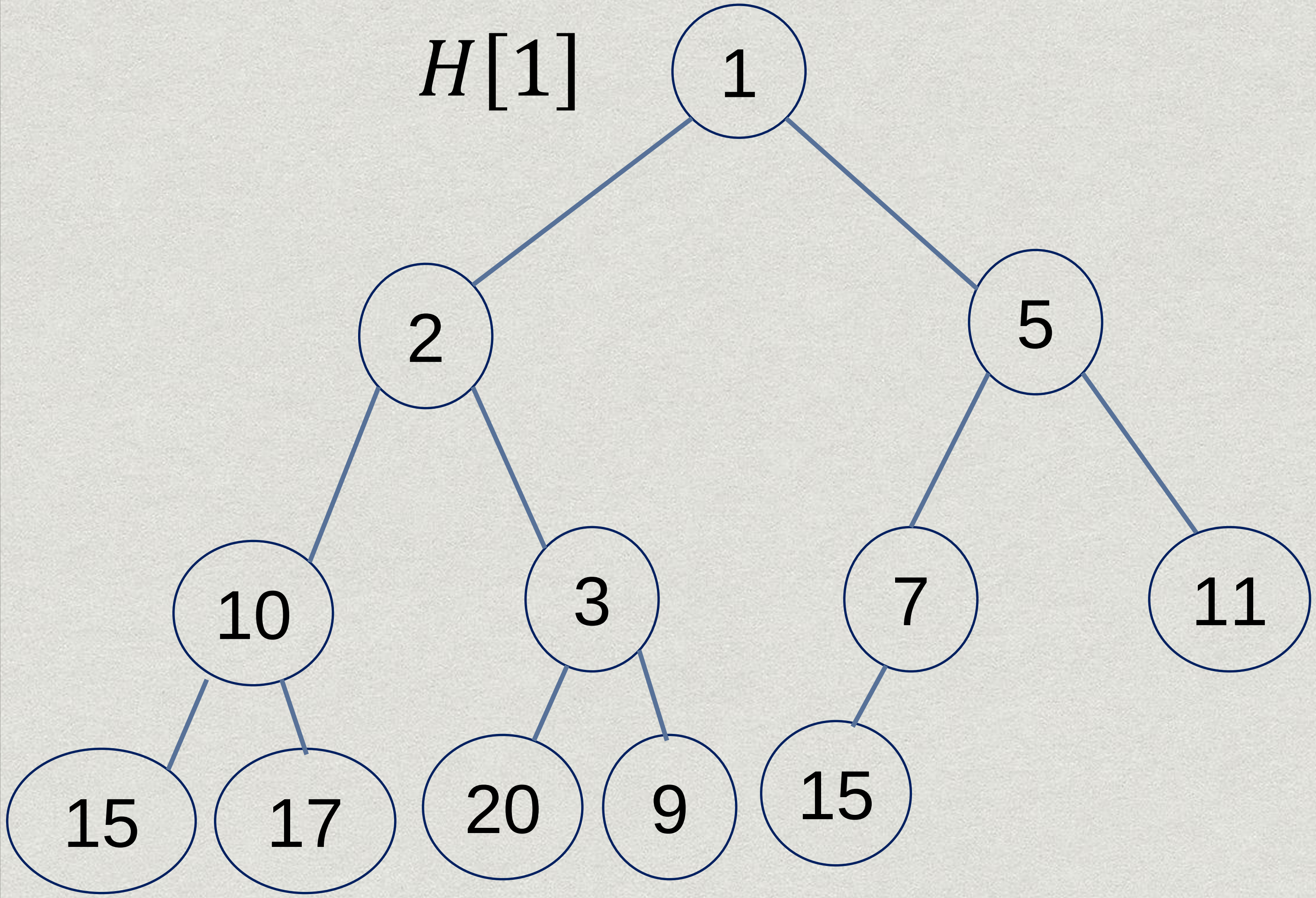
Heap: Elements stored in heap order (i.e. for each element, parent's key is equal or lower)

Heap data structure



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

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		

$n = 12$

Comparing data structures

What is the difference between heap and array, especially if heap is implemented using array?

<u>Array:</u> Not arranged in any order
<u>Sorted array:</u> Elements always stored in order (either ascending or descending)
<u>Heap:</u> Elements stored in heap order (i.e. for each element, parent's key is equal or lower)

	Array	Sorted Array	Heap
Select min	$O(n)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(n)$	$O(\log n)$

Flights landing at a runway

Flights are arriving at and departing from 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.

Flights landing at a runway

Flights are arriving at and departing from 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.

Flight landing/takeoff requests come in at different times

Trivandrum-Chennai Air India   4.23 pm

Chennai-Madurai SpiceJet   4.12 pm

Delhi-Chennai Air India   4.55 pm

Port Blair-Chennai Jet Airways   4.18 pm

...

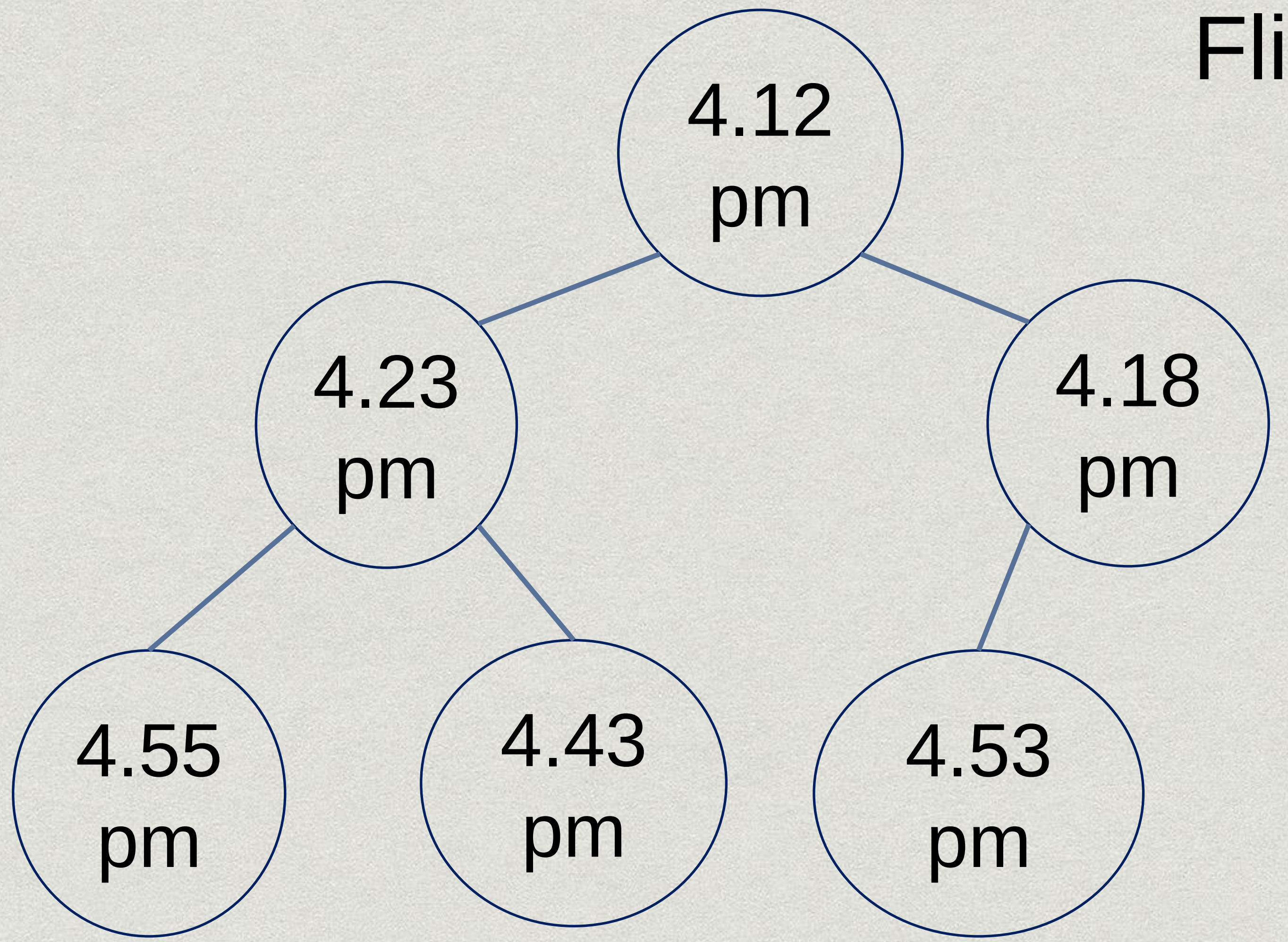
Can we use a heap?

Flight landing/takeoff requests come in at different times

4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

Entered into the heap

Can we use a heap?

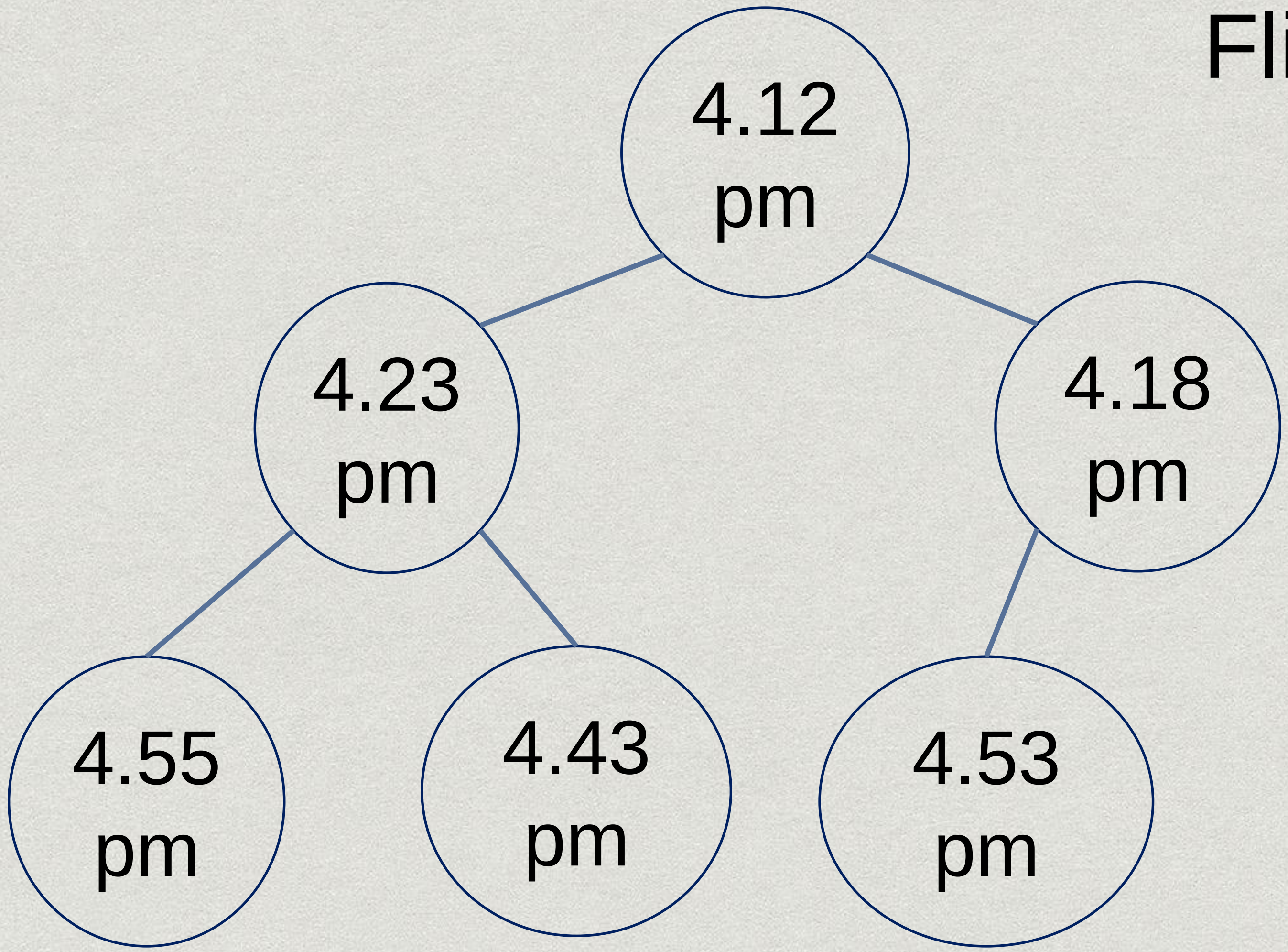


Flight landing/takeoff requests come in at different times

4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

Entered into the heap

Can we use a heap?



Flight landing/takeoff requests come in at different times

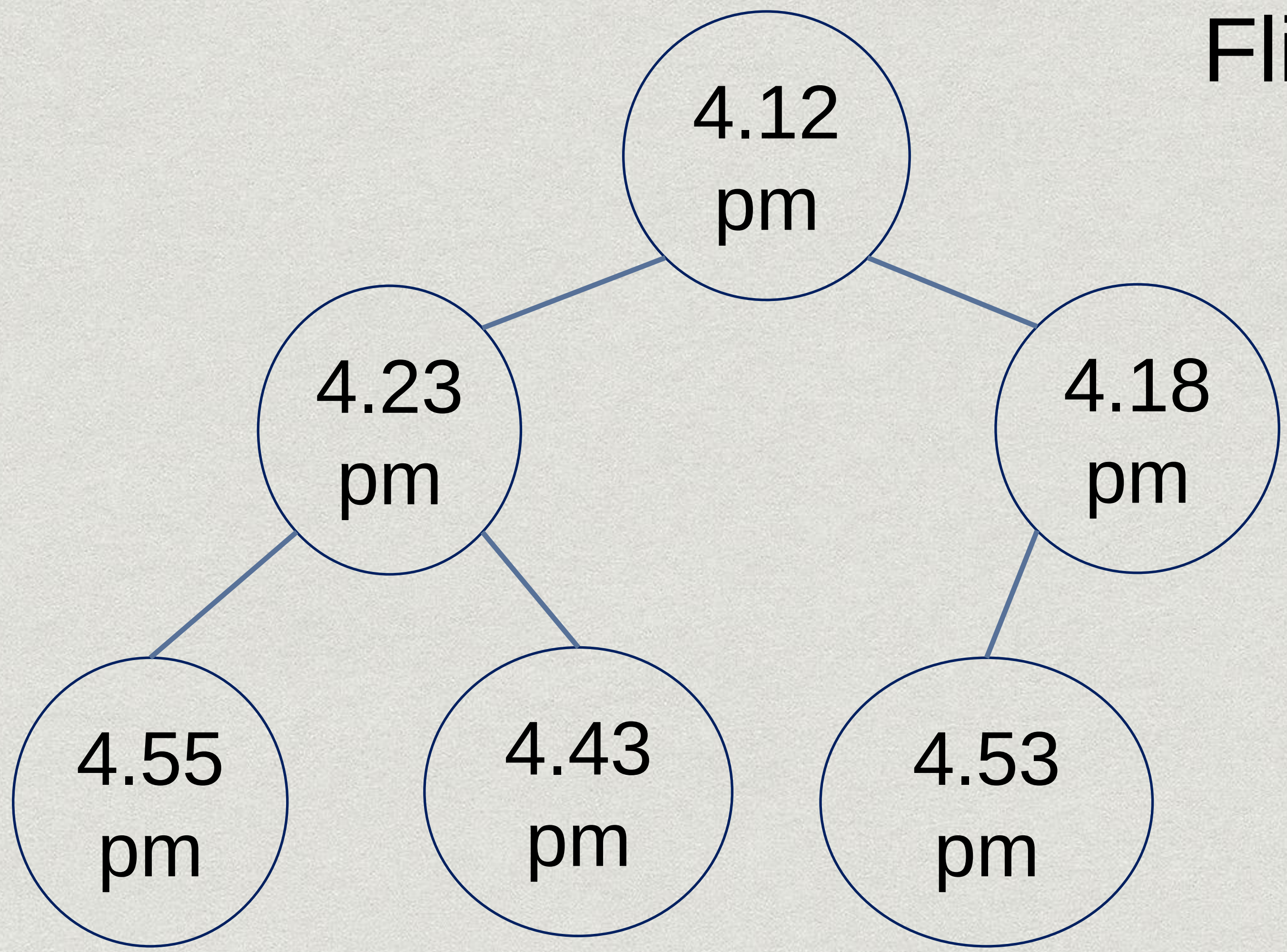
4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

Entered into the heap

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



Flight landing/takeoff requests come in at different times

Entered into the heap

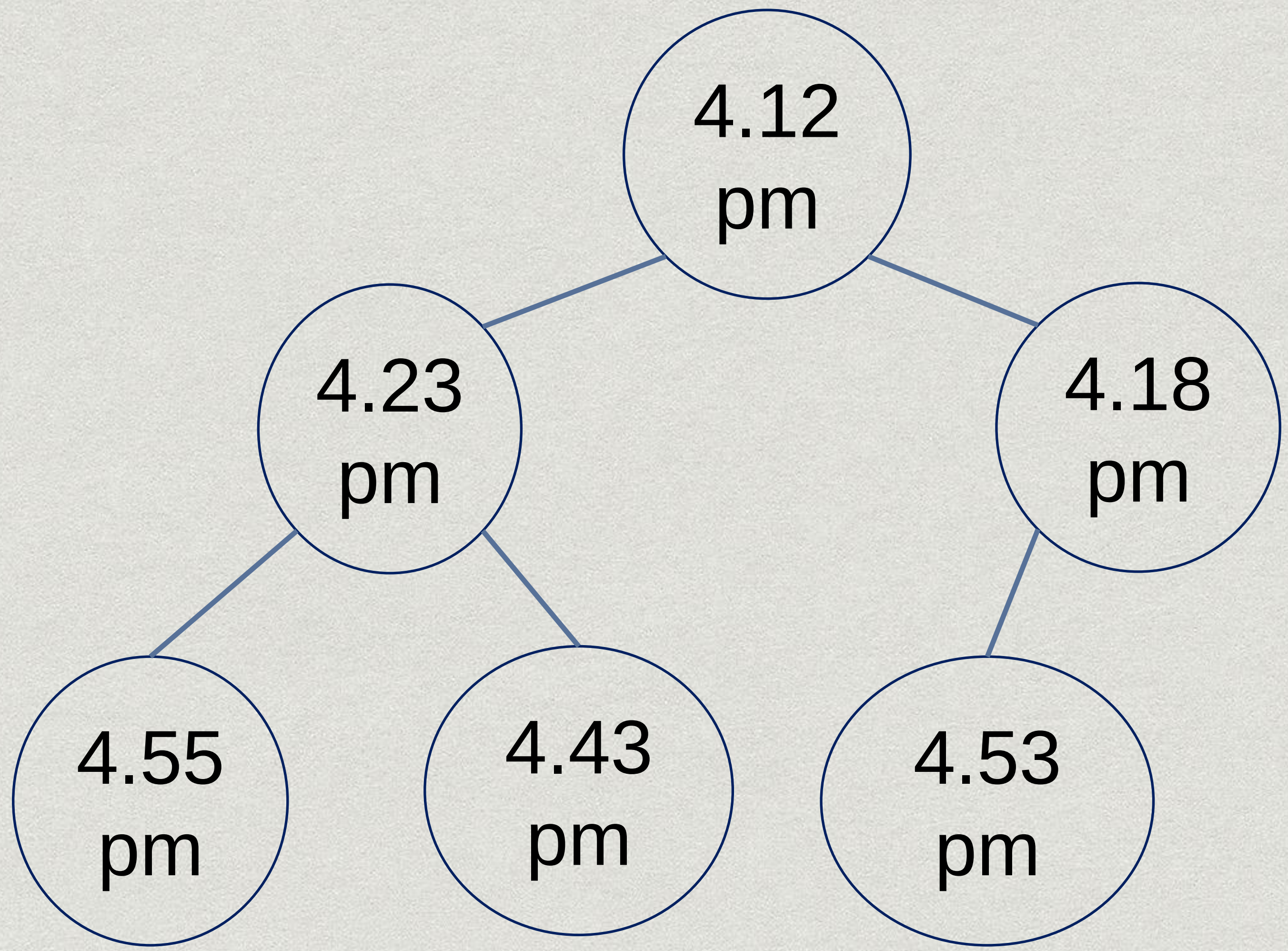
A few minutes before the minimum (say 4.10 pm), flight (SG MAA-IXM) is given takeoff permission

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?

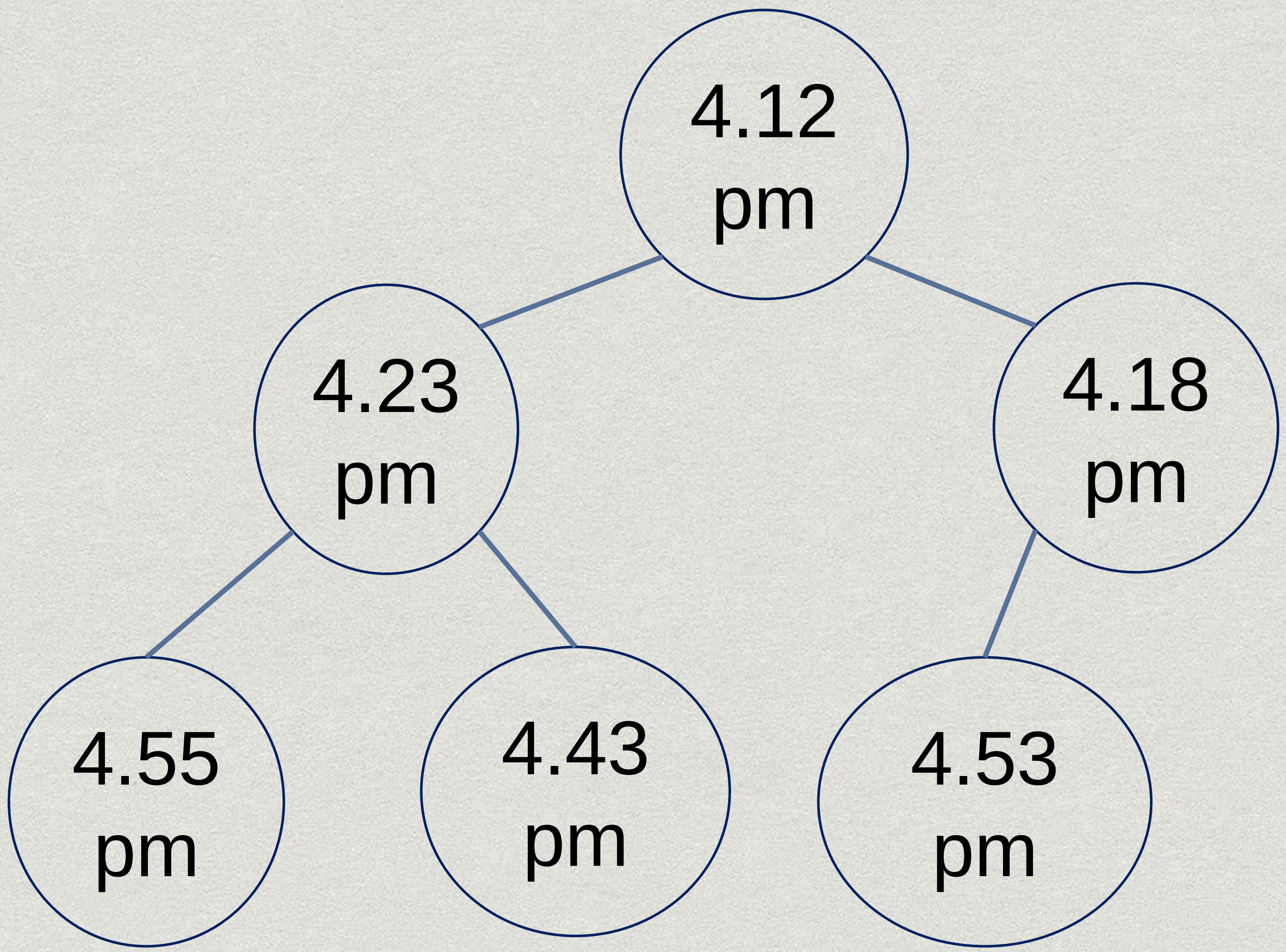
Suppose we want to ensure that the flights are spaced sufficiently apart



4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



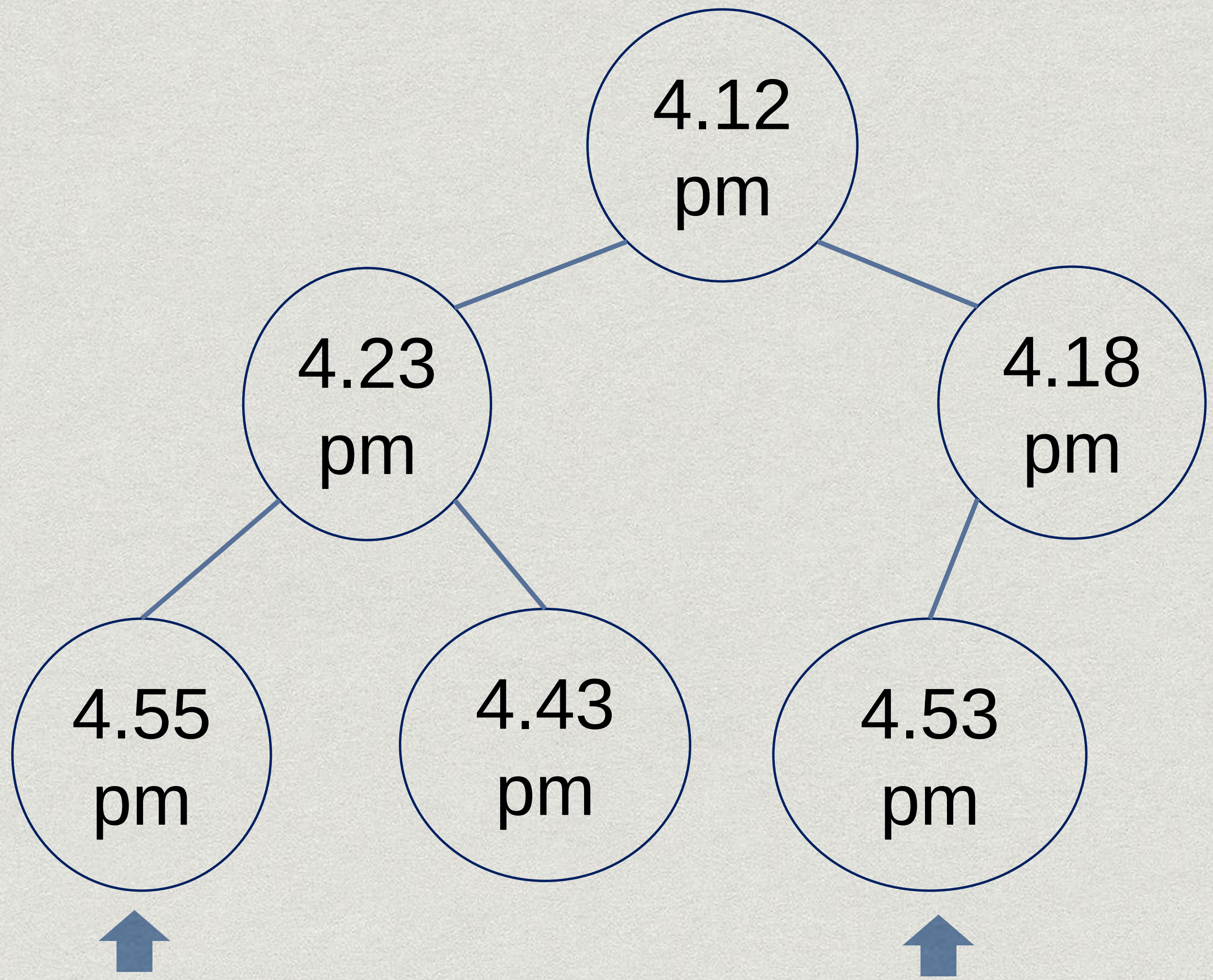
Suppose we want to ensure that the flights are spaced sufficiently apart

3 min space between two flights

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



Suppose we want to ensure that the flights are spaced sufficiently apart

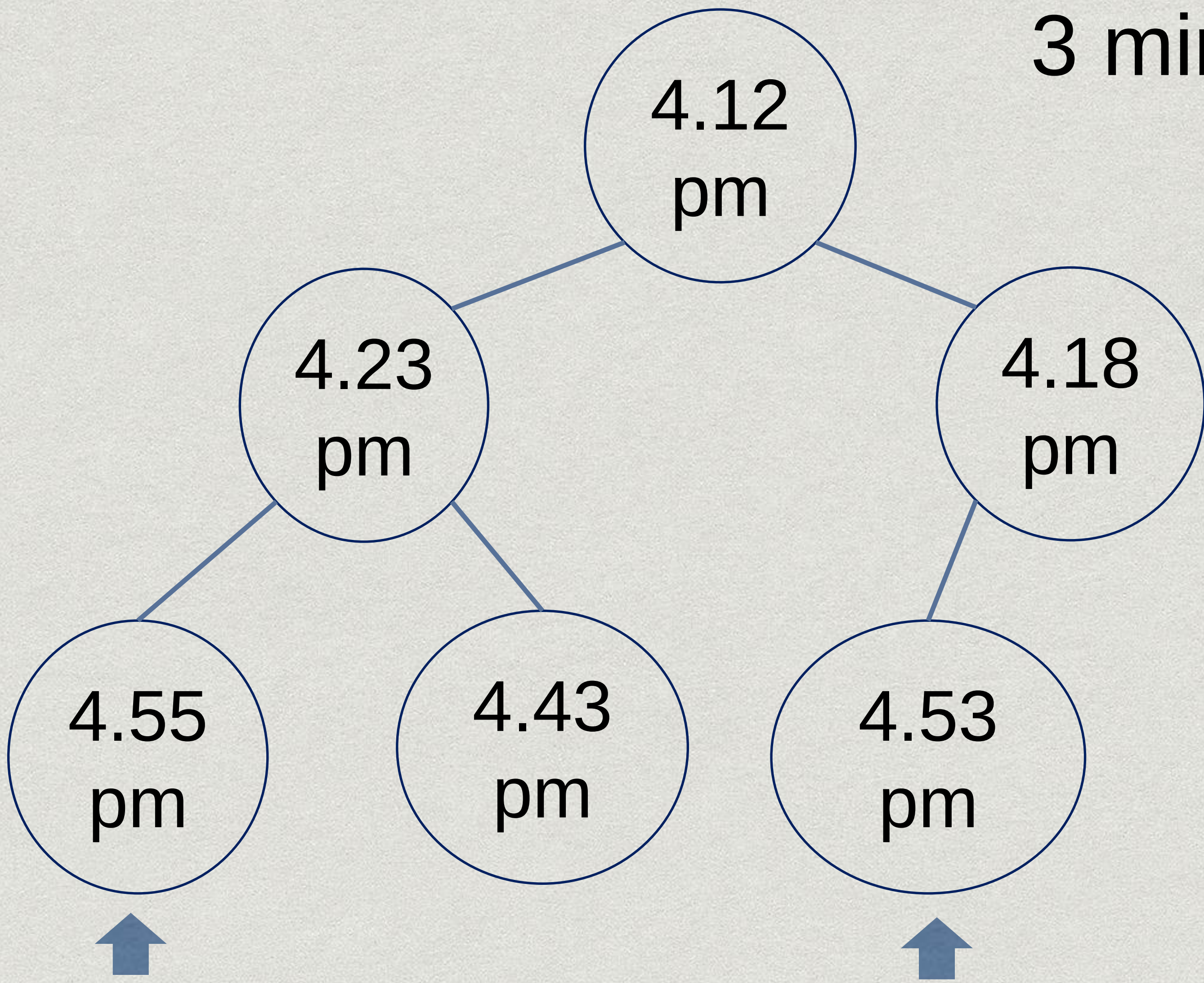
3 min space between two flights

Rule is violated

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



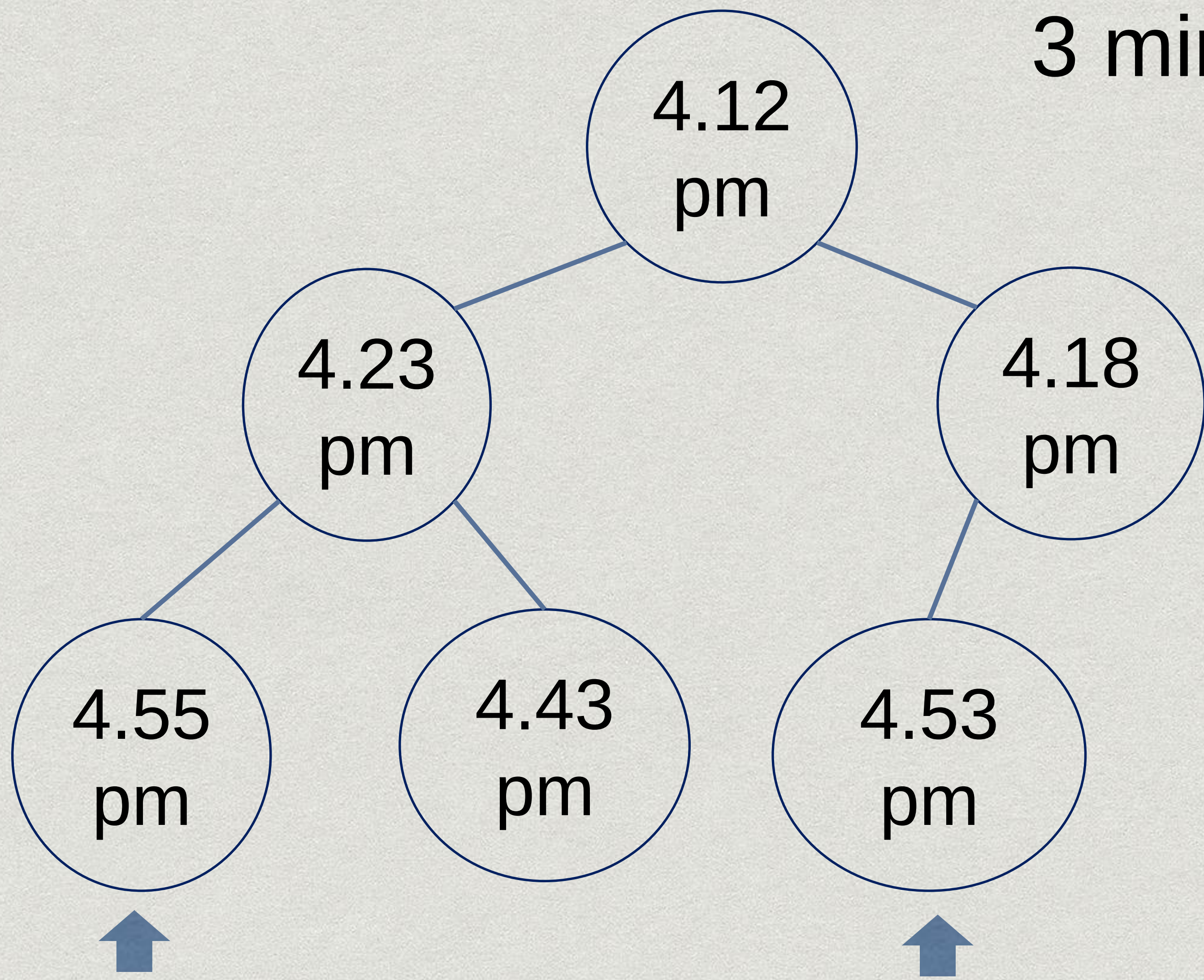
3 min space between two flights

Not easy to do this check in a heap data structure
$O(n)$ time to do this

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



3 min space between two flights

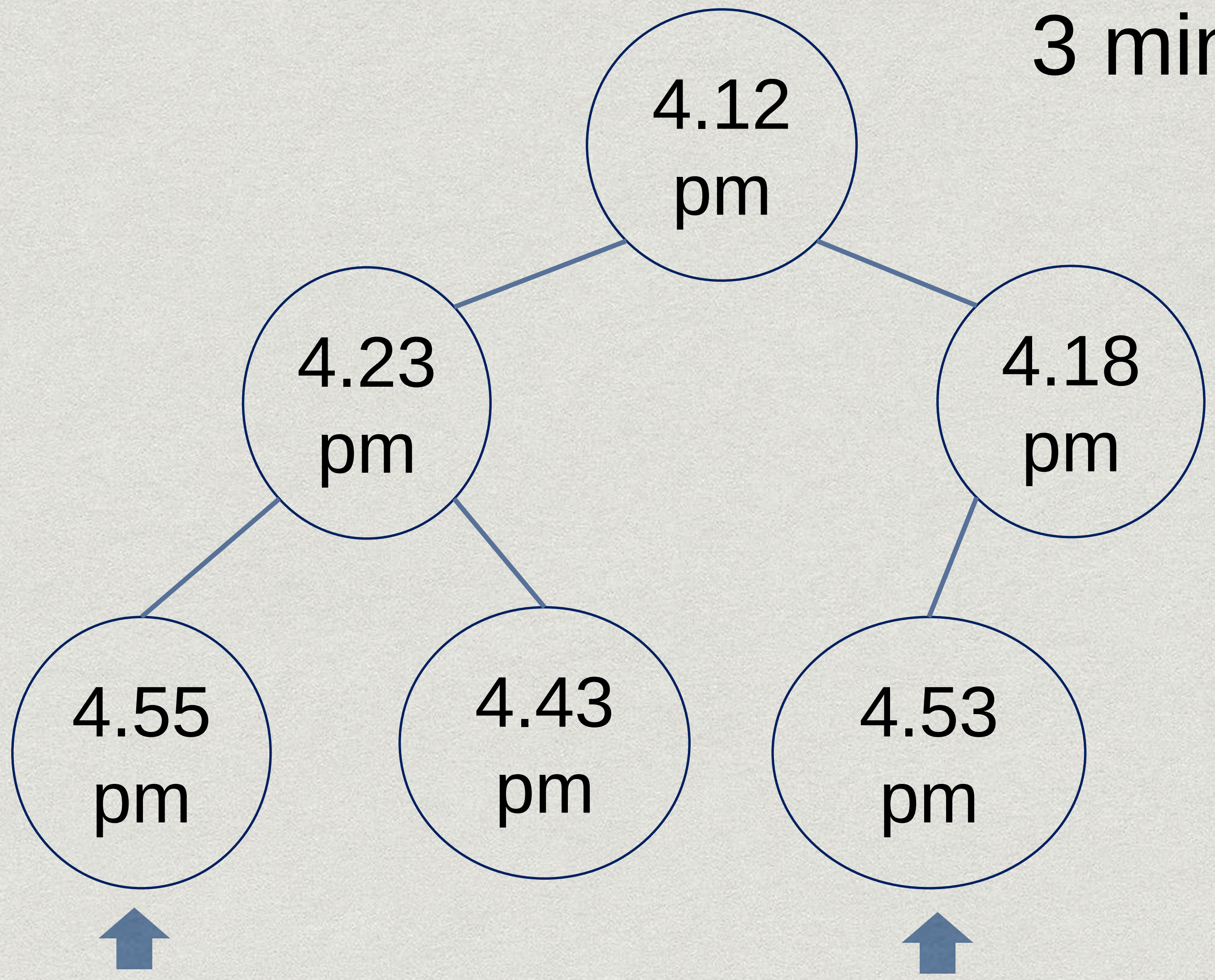
Not easy to do this check in a heap data structure
$O(n)$ time to do this

If we could find predecessor and successor easily, that would make the check easier to do

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



3 min space between two flights

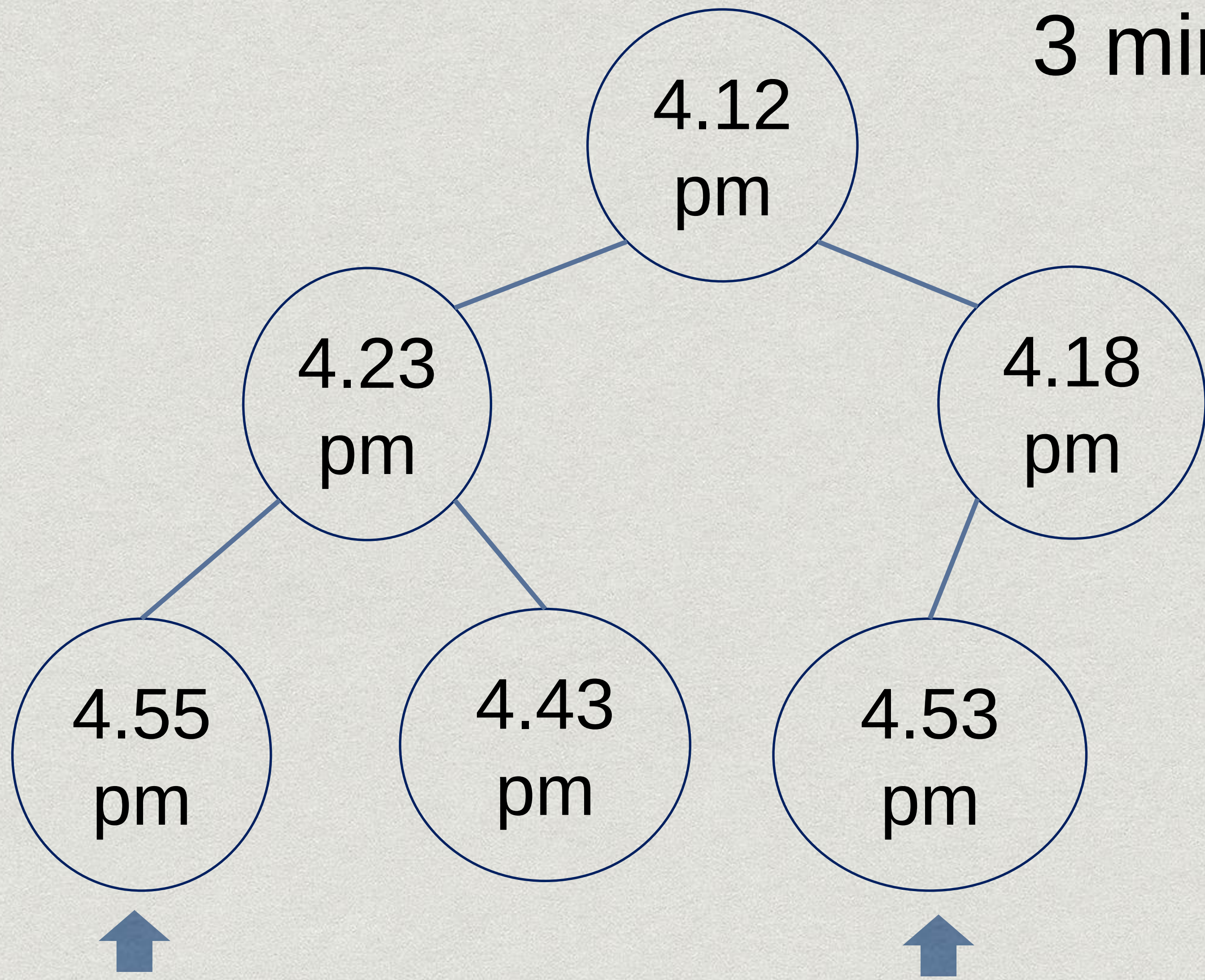
4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

Pred(4.18 pm) = 4.12 pm
Succ(4.18 pm) = 4.23 pm

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



3 min space between two flights

4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

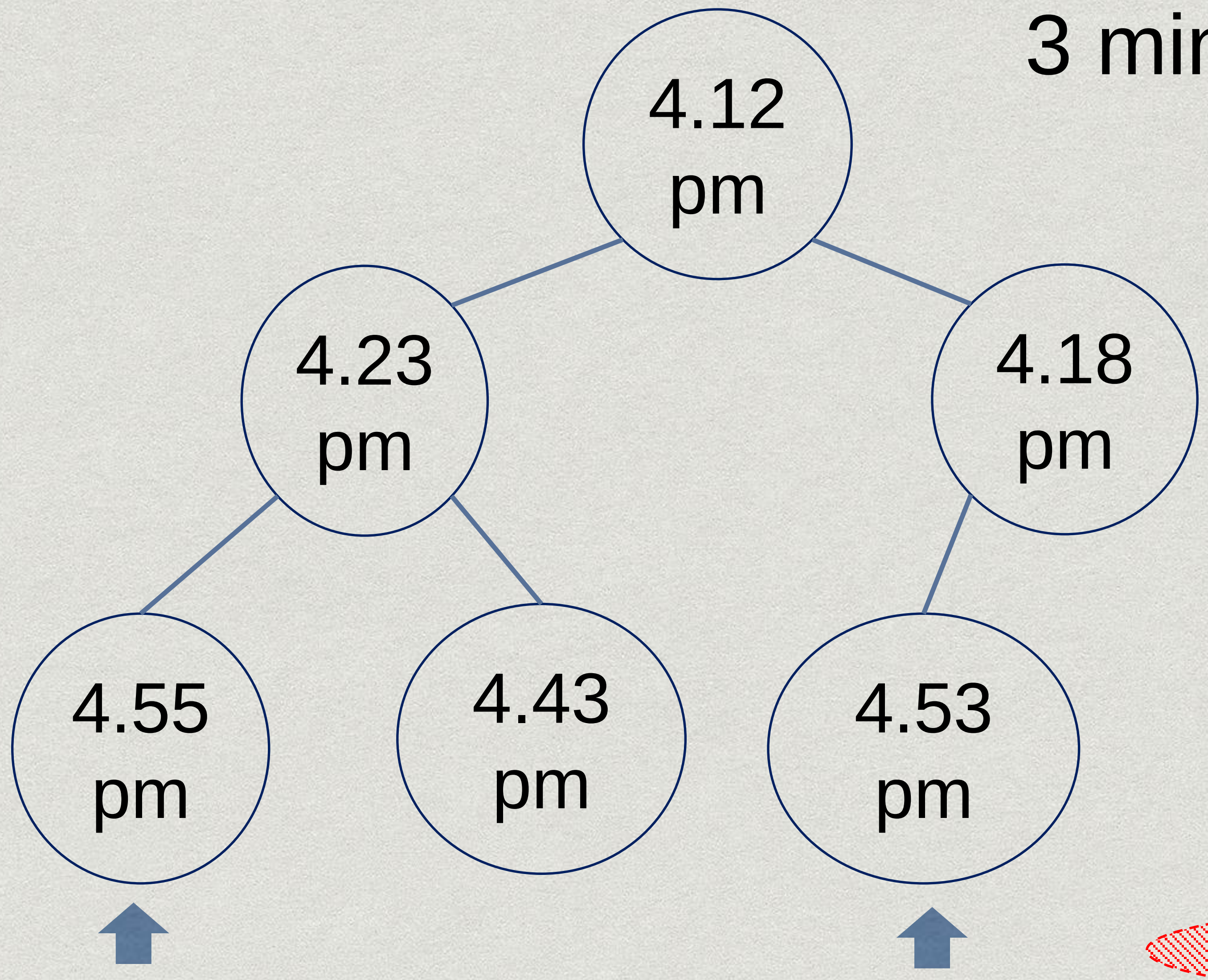
Pred(4.18 pm) = 4.12 pm
Succ(4.18 pm) = 4.23 pm

Pred(4.53 pm) = 4.43 pm
Succ(4.53 pm) = 4.55 pm

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Can we use a heap?



3 min space between two flights

4.23 pm, 4.12 pm,
4.55 pm, 4.18 pm,
4.43 pm, 4.53 pm

Pred(4.18 pm) = 4.12 pm
Succ(4.18 pm) = 4.23 pm

Pred(4.53 pm) = 4.43 pm
Succ(4.53 pm) = 4.55 pm

4.12 pm	4.23 pm	4.18 pm	4.55 pm	4.43 pm	4.53 pm		

$n = 6$

Comparing data structures

	Array	Sorted Array	Heap
Select min	$O(n)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(n)$	$O(\log n)$
Pred	$O(n)$	$O(1)$	$O(n)$
Succ	$O(n)$	$O(1)$	$O(n)$

Sorted array does well with respect to Pred and Succ, but not so well for Insert and Delete. Heap does well for Insert, Delete but not so well for Pred and Succ

Today's class

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

Need a new data structure

	Heap	Sorted Array	Binary Search Tree
Search	$O(n)$	$O(\log n)$	$O(\log n)$
Minimum	$O(1)$	$O(1)$	$O(\log n)$
Maximum	$O(n)$	$O(1)$	$O(\log n)$
Insert	$O(\log n)$	$O(n)$	$O(\log n)$
Delete	$O(\log n)$	$O(n)$	$O(\log n)$
Pred	$O(n)$	$O(1)$	$O(\log n)$
Succ	$O(n)$	$O(1)$	$O(\log n)$

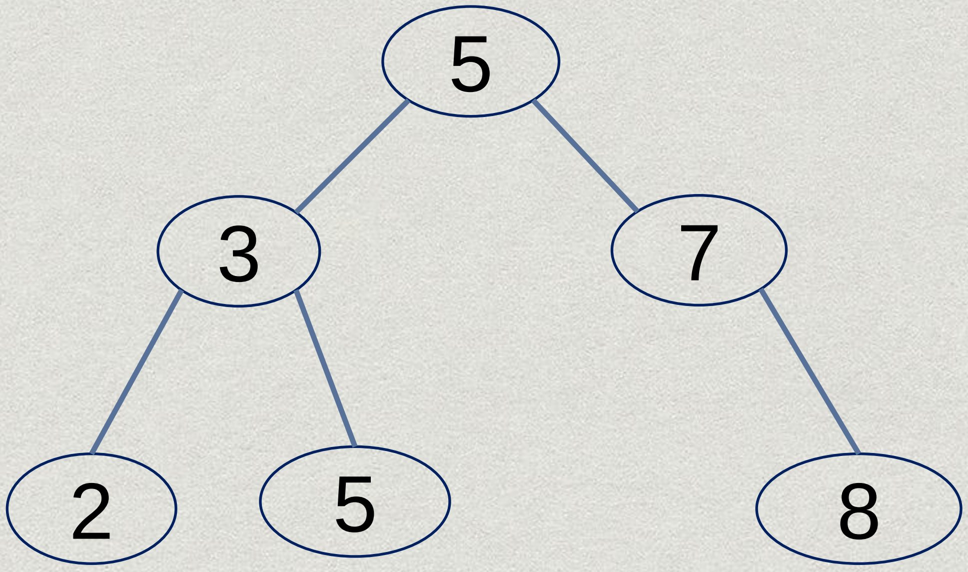
If we need to do search and insert/delete repeatedly, then a new data structure is required – binary search tree

What is a binary tree?

- For each node, we can find the leftchild, the rightchild and the parent.
- Has a ROOT node – which does not have a parent
  (All other nodes in the tree should have a parent)
- Some of the nodes may have left or right child missing
- If both left and right childs are absent, then the node is a LEAF node

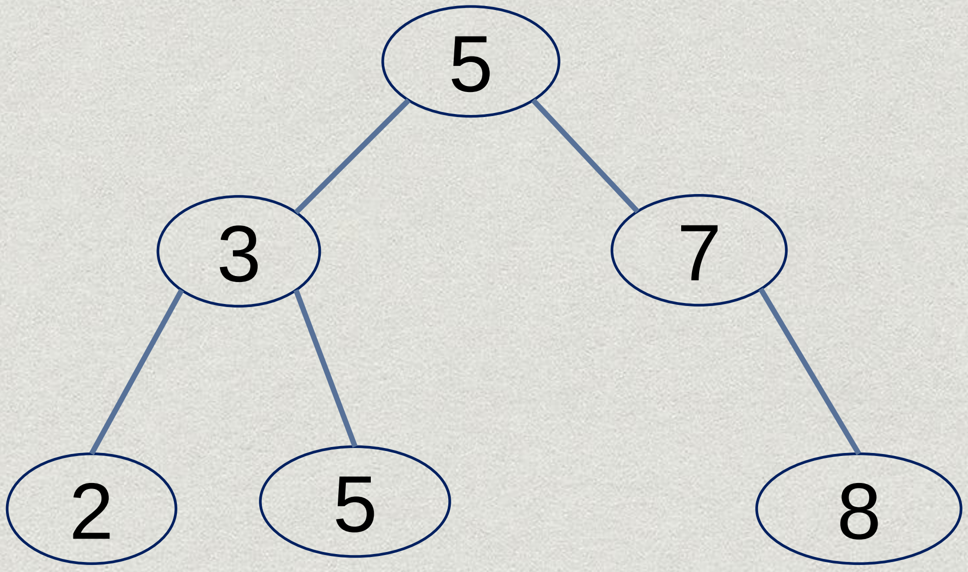
What is a binary tree?

- For each node, we can find the leftchild, the rightchild and the parent.
- Has a ROOT node – which does not have a parent (All other nodes in the tree should have a parent)
- Some of the nodes may have left or right child missing
- If both left and right childs are absent, then the node is a LEAF node



What is a binary tree?

- For each node, we can find the leftchild, the rightchild and the parent.
- Has a ROOT node – which does not have a parent (All other nodes in the tree should have a parent)
- Some of the nodes may have left or right child missing
- If both left and right childs are absent, then the node is a LEAF node



5 is the ROOT

For node 3: parent is 5,
children are 2 and 5

7 has no leftchild
2, 5 and 8 are LEAFs

What is a binary search tree?

- Is a binary tree

What is a binary search tree?

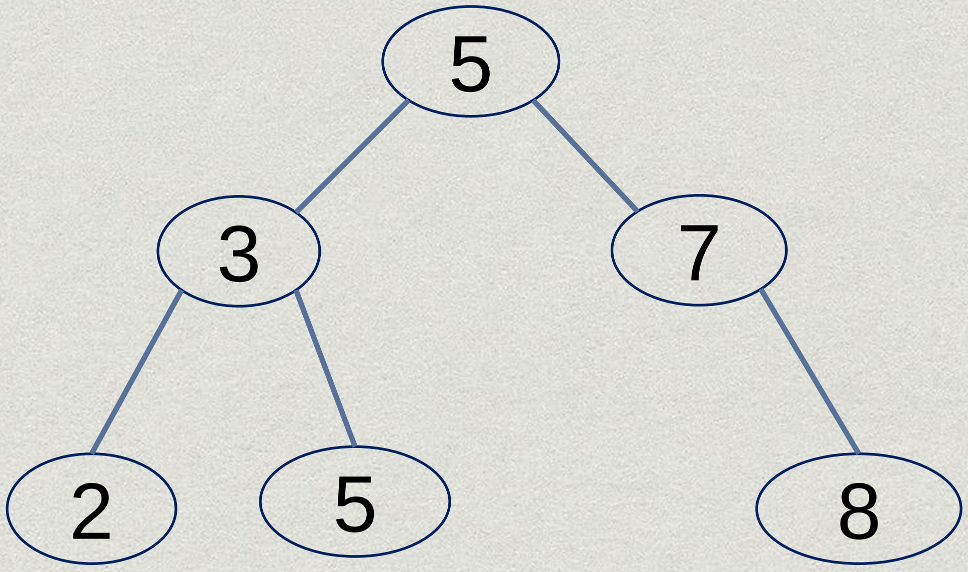
- Is a binary tree
- Each node x in the tree has a key value $key(x)$

What is a binary search tree?

- Is a binary tree
- Each node x in the tree has a key value $key(x)$
- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$
- Left subtree is the tree rooted at the leftchild of a node
- Right subtree is the tree rooted at the rightchild of a node

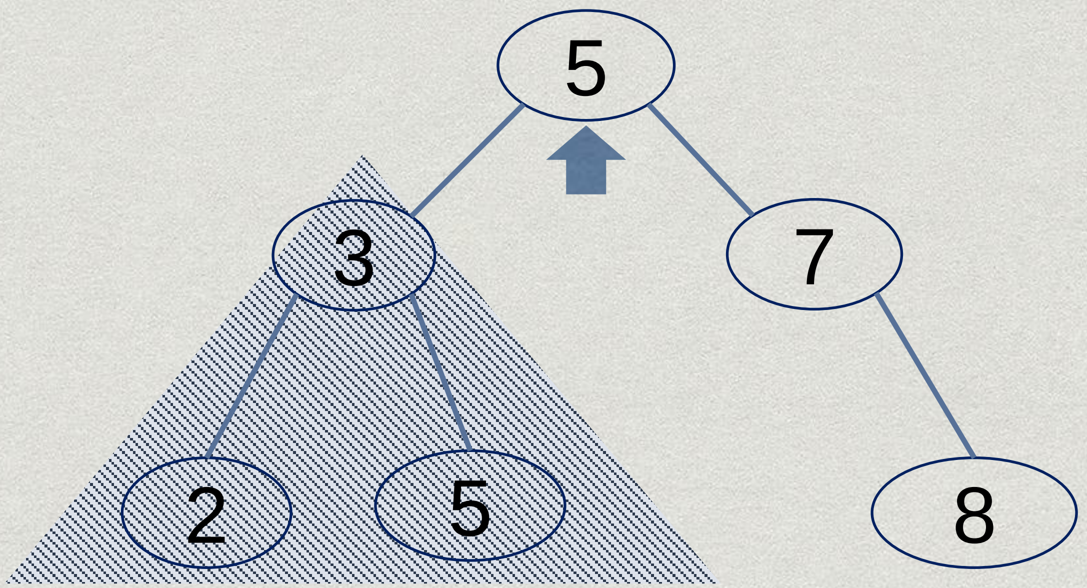
What is a binary search tree?

- Is a binary tree
- Each node x in the tree has a key value $key(x)$
- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$



What is a binary search tree?

- Is a binary tree
- Each node x in the tree has a key value $key(x)$
- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$

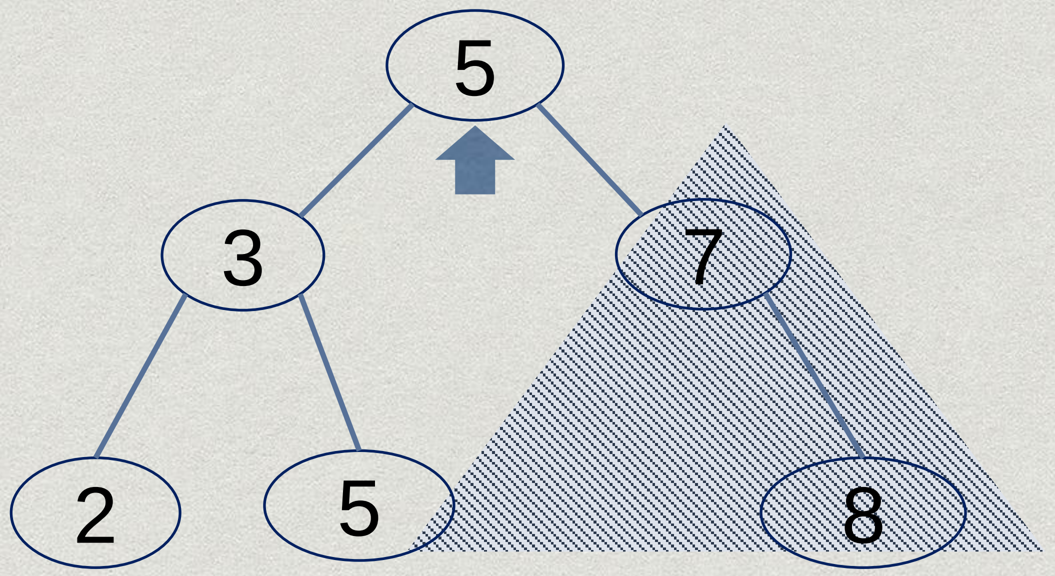


Left subtree of marked node is shown shaded

All the nodes in the left subtree have key values no greater than 5

What is a binary search tree?

- Is a binary tree
- Each node x in the tree has a key value $key(x)$
- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$



Right subtree of marked node is shown shaded

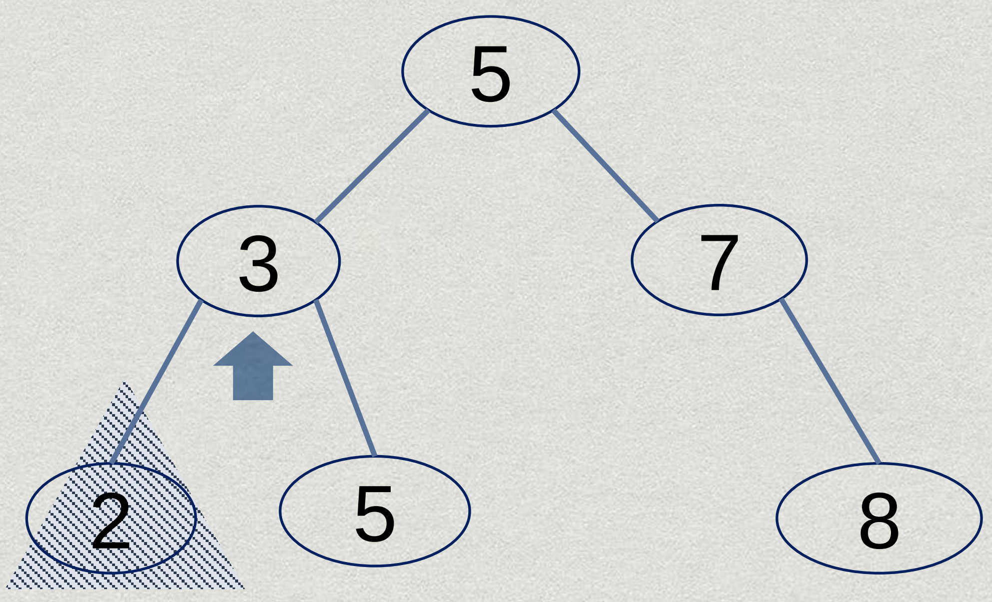
All the nodes in the right subtree have key values no less than 5

What is a binary search tree?

- Is a binary tree

- Each node x in the tree has a key value $key(x)$

- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$

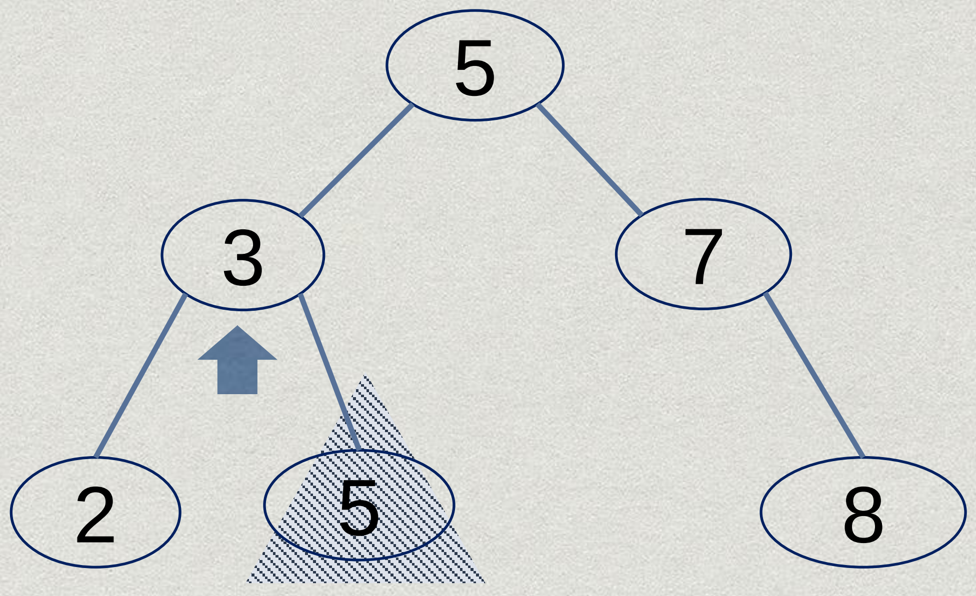


Left subtree of marked node is shown shaded

All the nodes in the left subtree have key values no greater than 3

What is a binary search tree?

- Is a binary tree
- Each node x in the tree has a key value $key(x)$
- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$



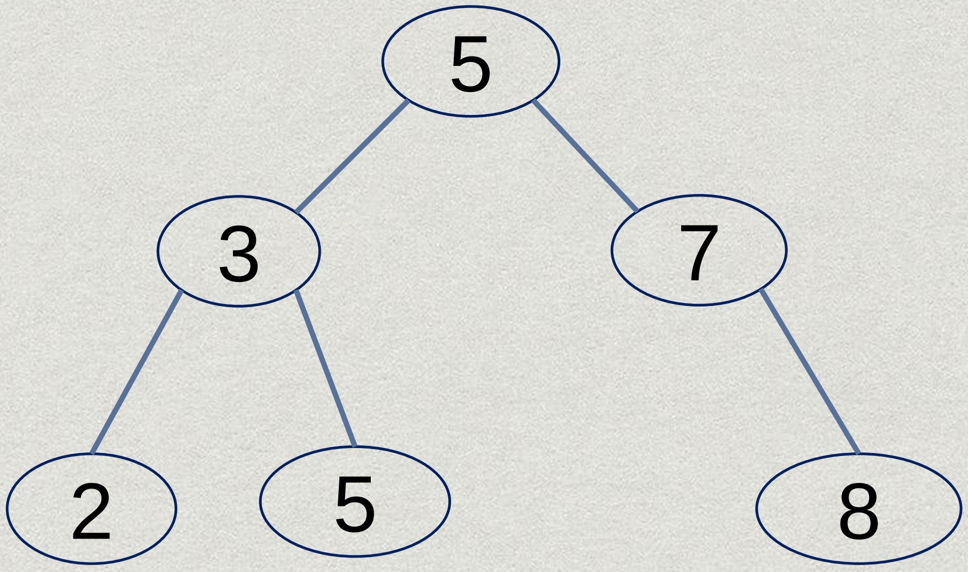
Right subtree of marked node is shown shaded

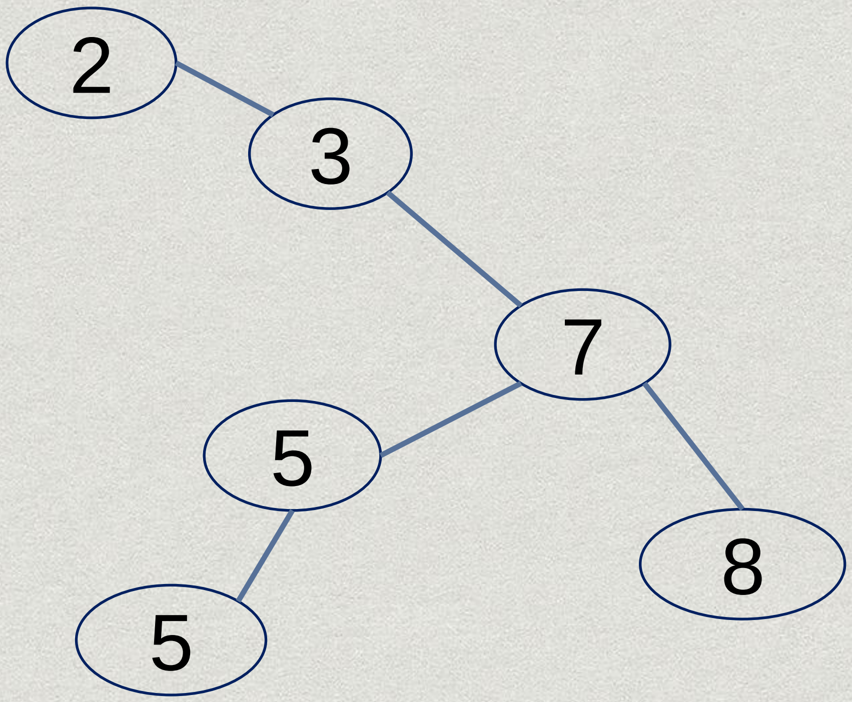
All the nodes in the right subtree have key values no less than 3

What is a binary search tree?

- Keys are always stored so as to satisfy the property:
  if node y is in the left subtree of x, then $key(y) \leq key(x)$
  if node y is in the right subtree of x, then $key(x) \leq key(y)$

There could be more than one way to store a set of elements in a binary search tree





Implementing a binary search tree

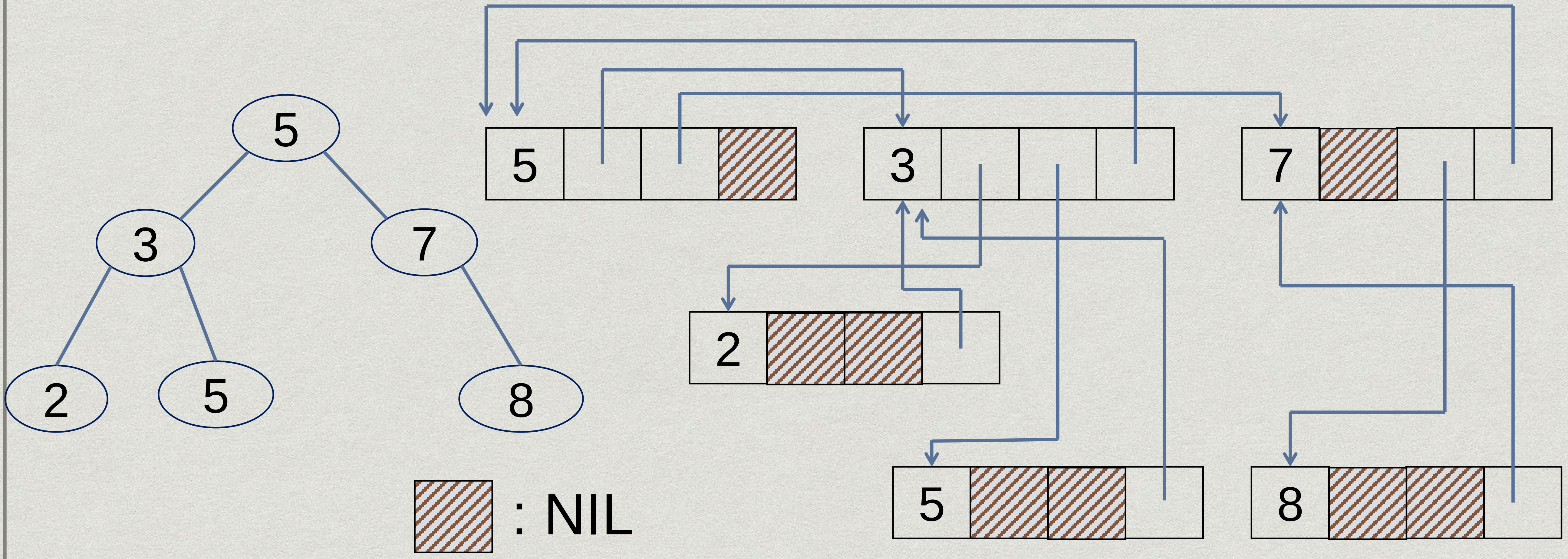
- Use pointers to implement a binary search tree
- Each node has a key field and three pointers – one each for parent, leftChild and rightChild

- If parent or any child is missing, the corresponding pointer value is NIL

Implementing a binary search tree

- Use pointers to implement a binary search tree
- Each node has a key field and three pointers – one each for parent, leftChild and rightChild

- If parent or any child is missing, the corresponding pointer value is NIL

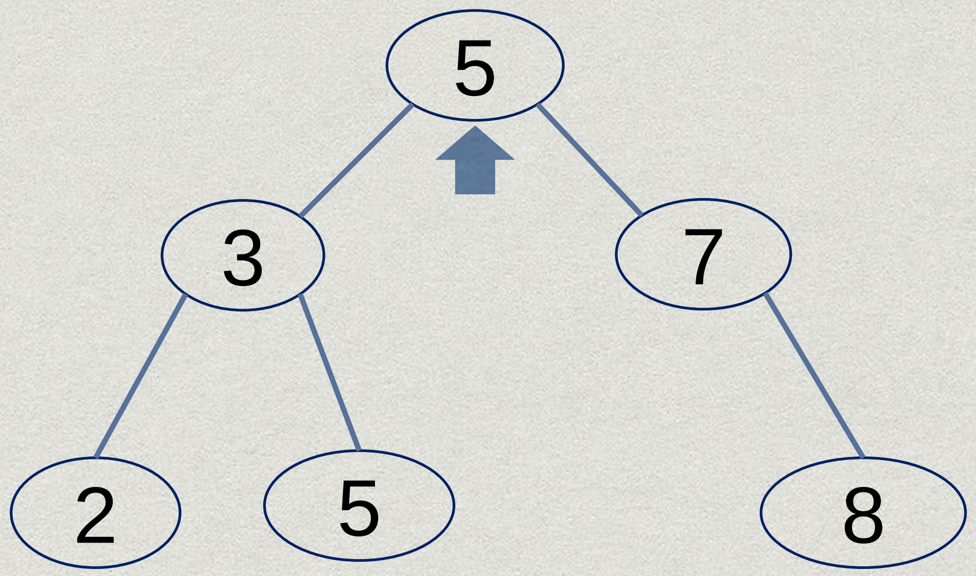


Inorder tree walk:
Printing out elements in sorted order

```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```

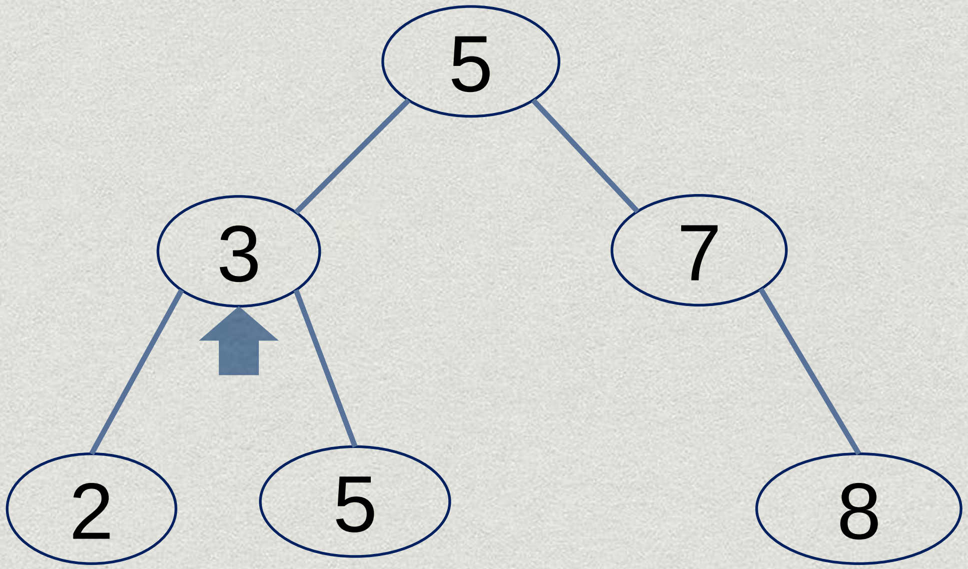
Inorder tree walk:
Printing out elements in sorted order

```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



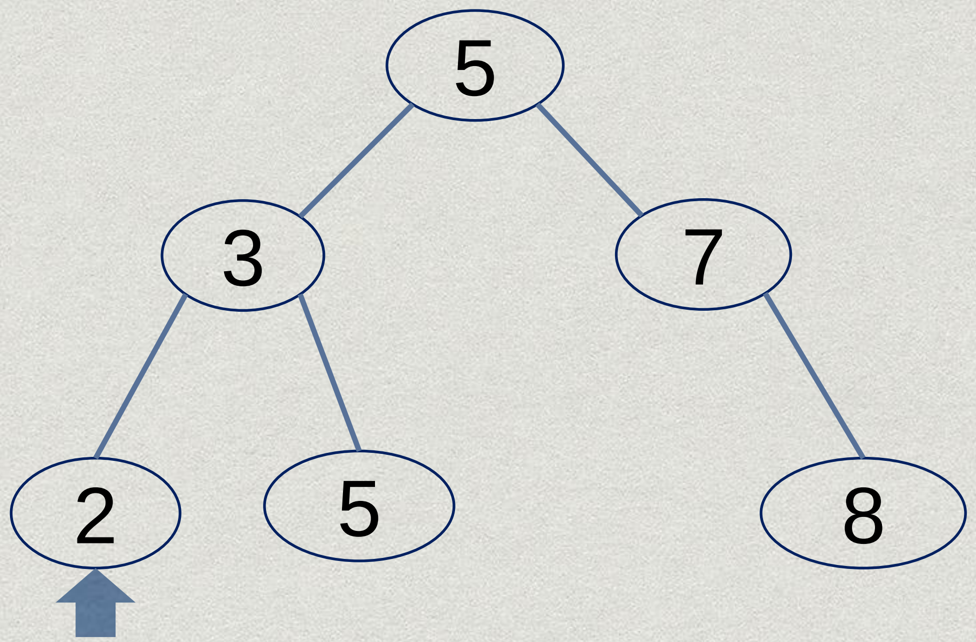
Inorder tree walk: Printing out elements in sorted order

```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



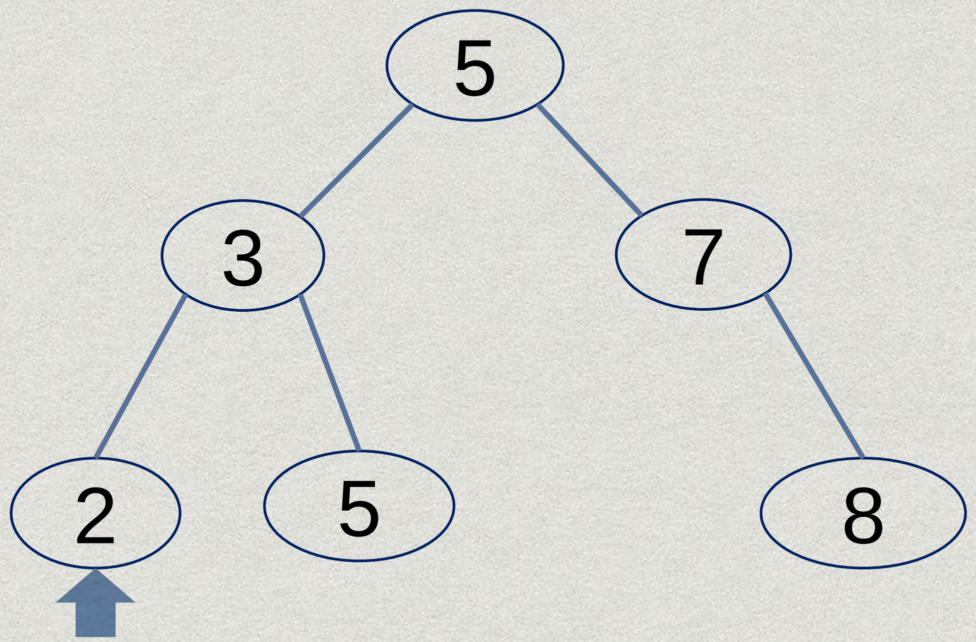
Inorder tree walk:
Printing out elements in sorted order

```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



Inorder tree walk: Printing out elements in sorted order

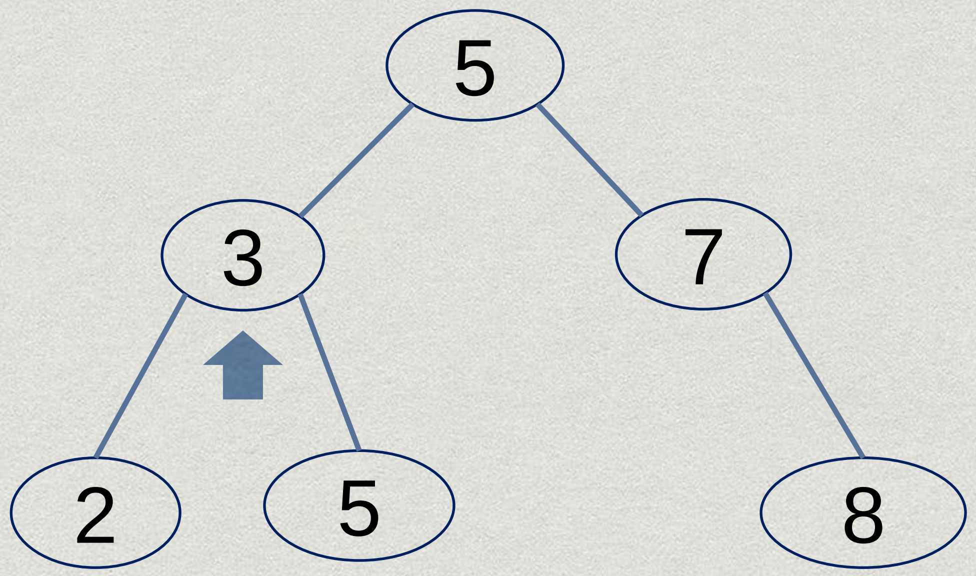
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2

Inorder tree walk:
Printing out elements in sorted order

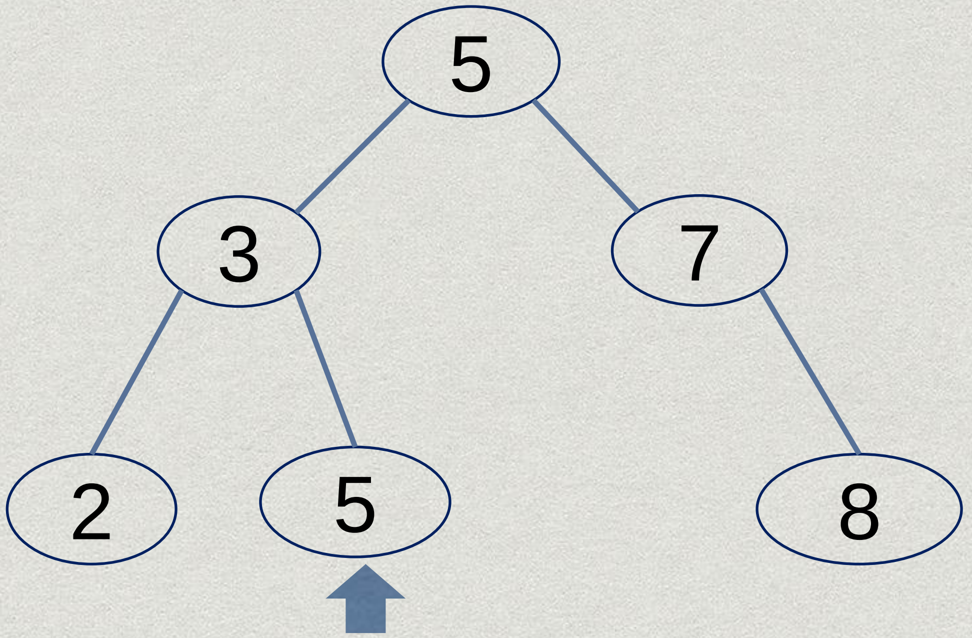
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3

Inorder tree walk:
Printing out elements in sorted order

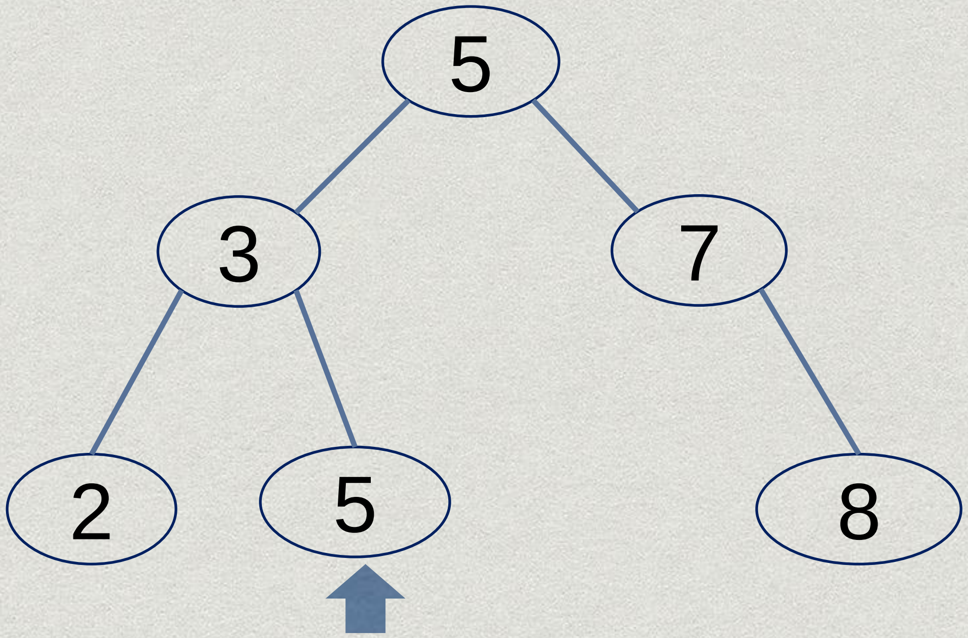
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3

Inorder tree walk:
Printing out elements in sorted order

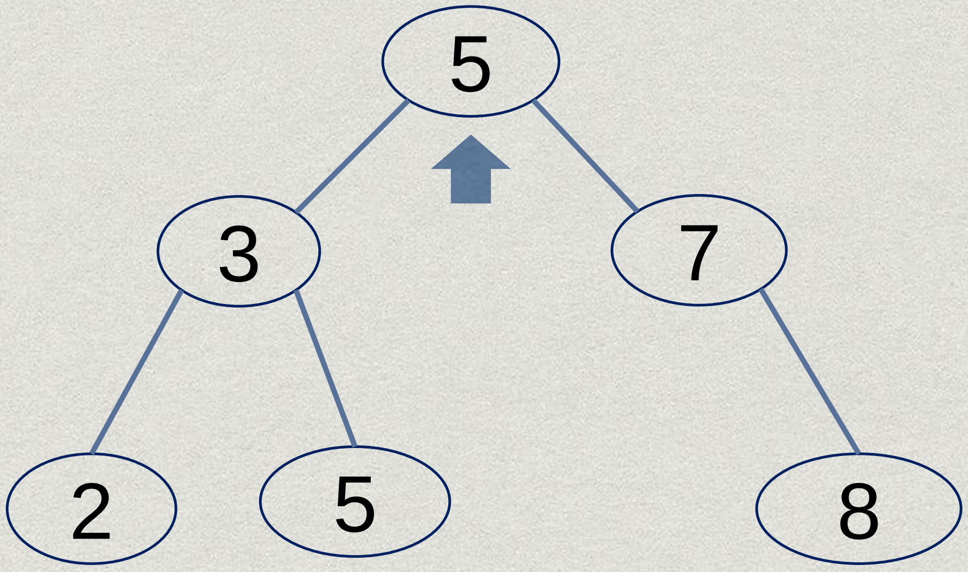
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5

Inorder tree walk:
Printing out elements in sorted order

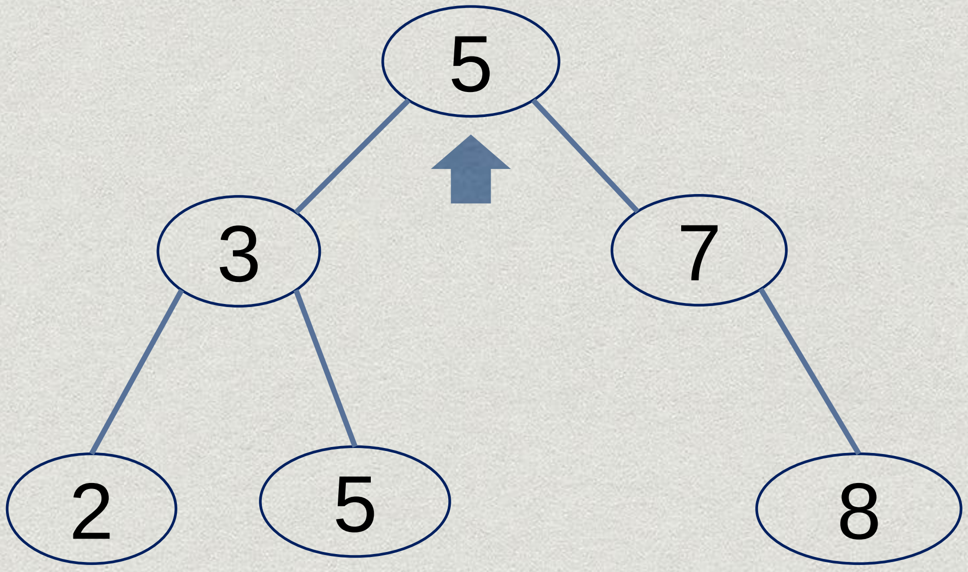
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5 5

Inorder tree walk:
Printing out elements in sorted order

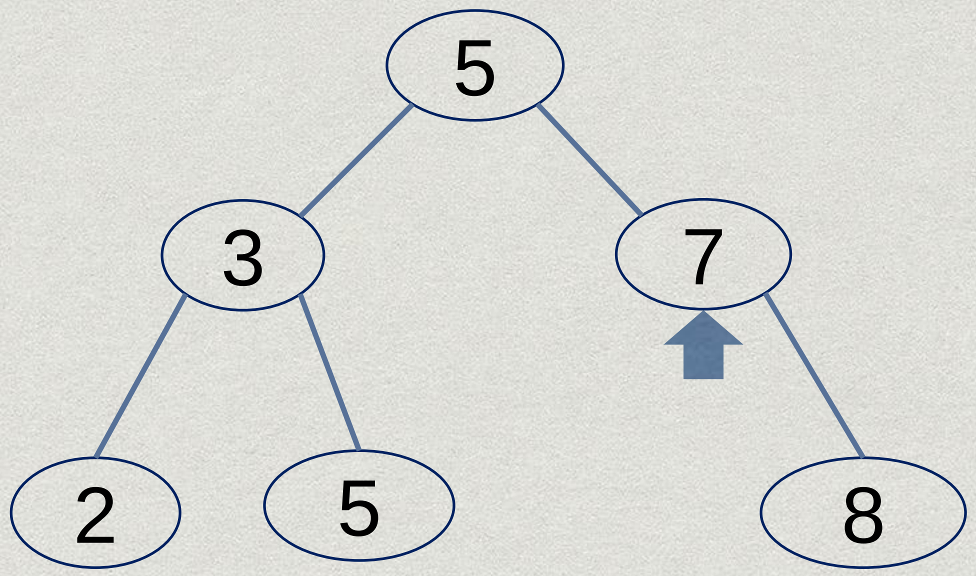
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5 5

Inorder tree walk: Printing out elements in sorted order

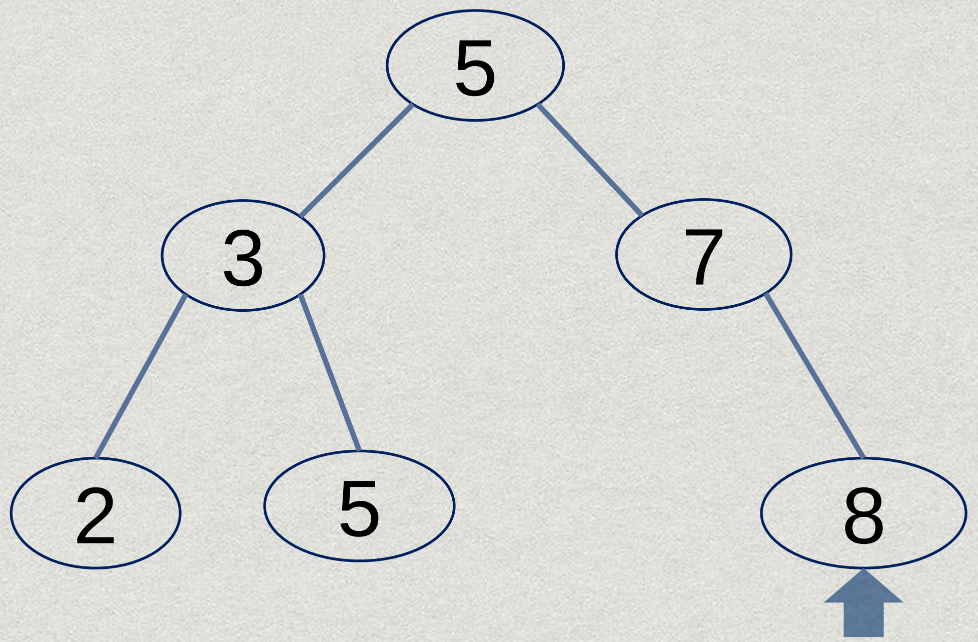
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5 5 7

Inorder tree walk:
Printing out elements in sorted order

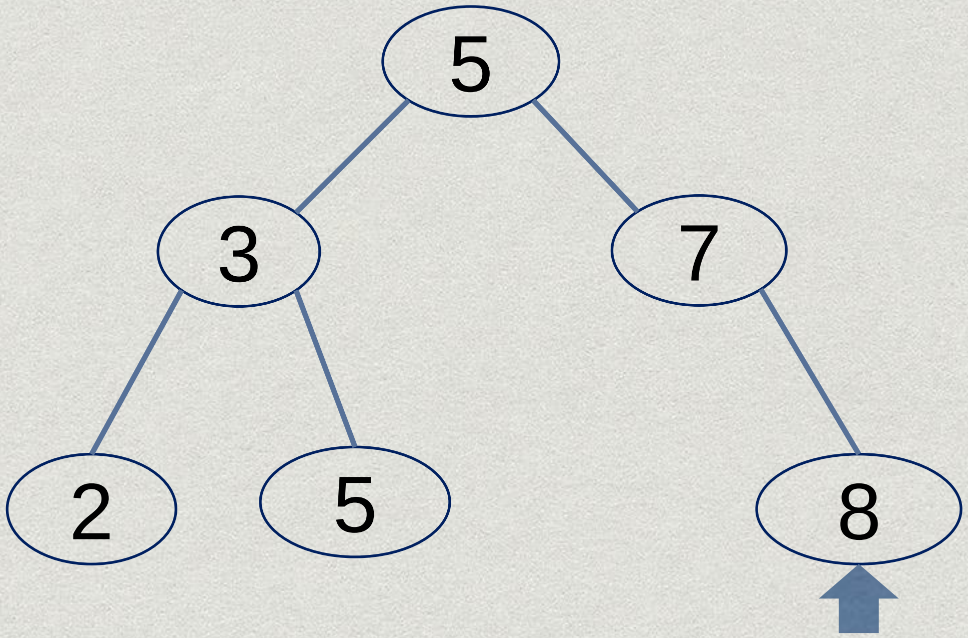
```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5 5 7

Inorder tree walk: Printing out elements in sorted order

```
Inorder_Tree_Walk(x) {
    if x ≠ NIL:
        Inorder_Tree_Walk(left_subtree(x))
        Print key(x)
        Inorder_Tree_Walk(right_subtree(x))
    endif
}
```



2 3 5 5 7 8

Searching a tree

```
Node Tree_Search(x,k) {
    if x == NIL or k = key(x):
        return (x)
    endif

    if k < key(x):
        return Tree_Search( leftChild(x), k )
    else
        return Tree_Search( rightChild(x), k )
    endif
}
```

Searching a tree (iterative form)

```
Node Tree_Search(x,k) {

    while x ≠ NIL or k ≠ key(x):

        if k < key(x):
            x = leftChild(x)
        else
            x = rightChild(x)
        endif

    endwhile

    return x
}
```

Finding minimum and maximum

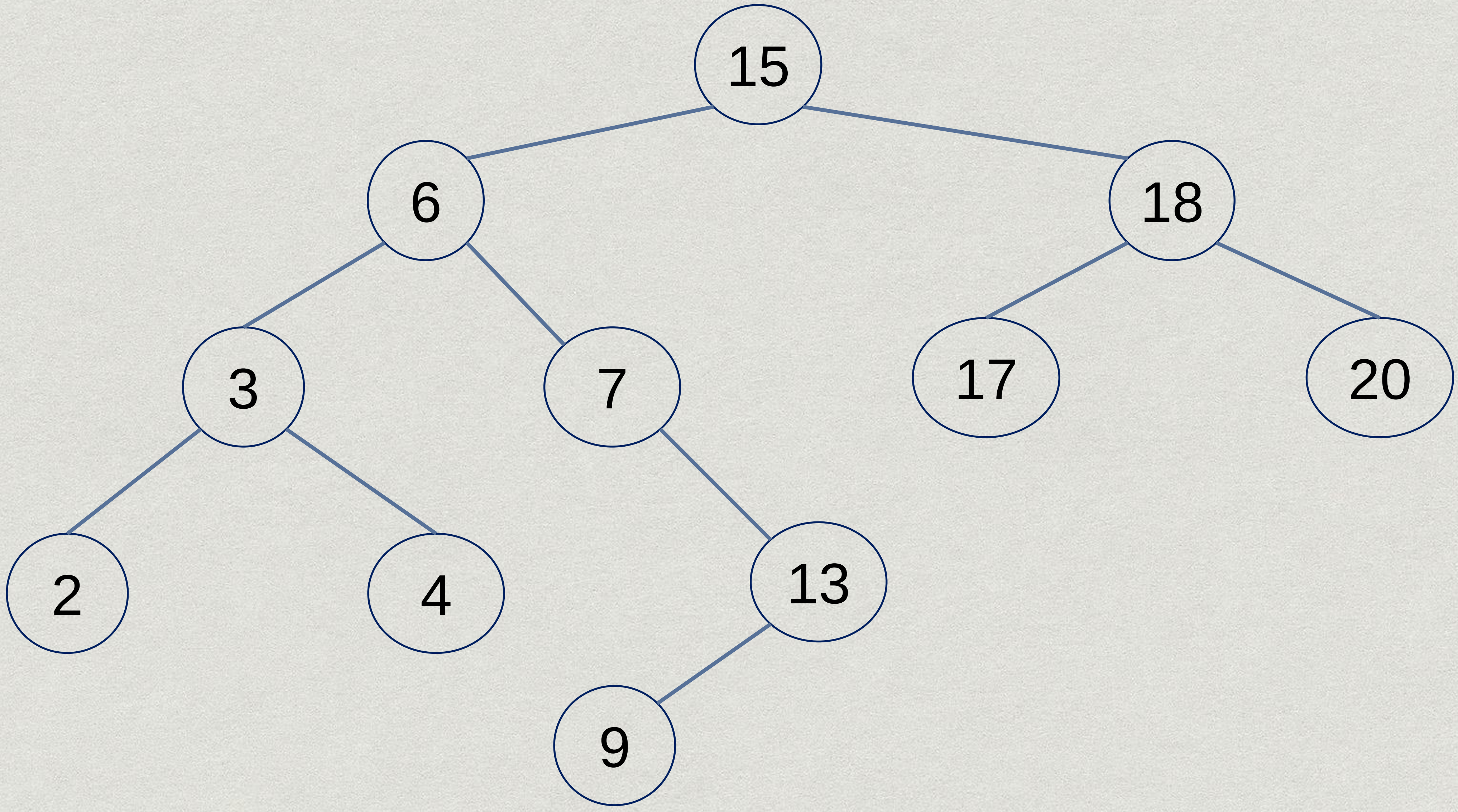
```
Node Tree_Min(x) {

    while leftChild(x) ≠ NIL:
       x = leftChild(x)
    endwhile

    return x
}
```

```
Node Tree_Max(x) {

    while rightChild(x) ≠ NIL:
       x = rightChild(x)
    endwhile

    return x
}
```

Minimum is found by going down the left side of the tree till a node without a left child

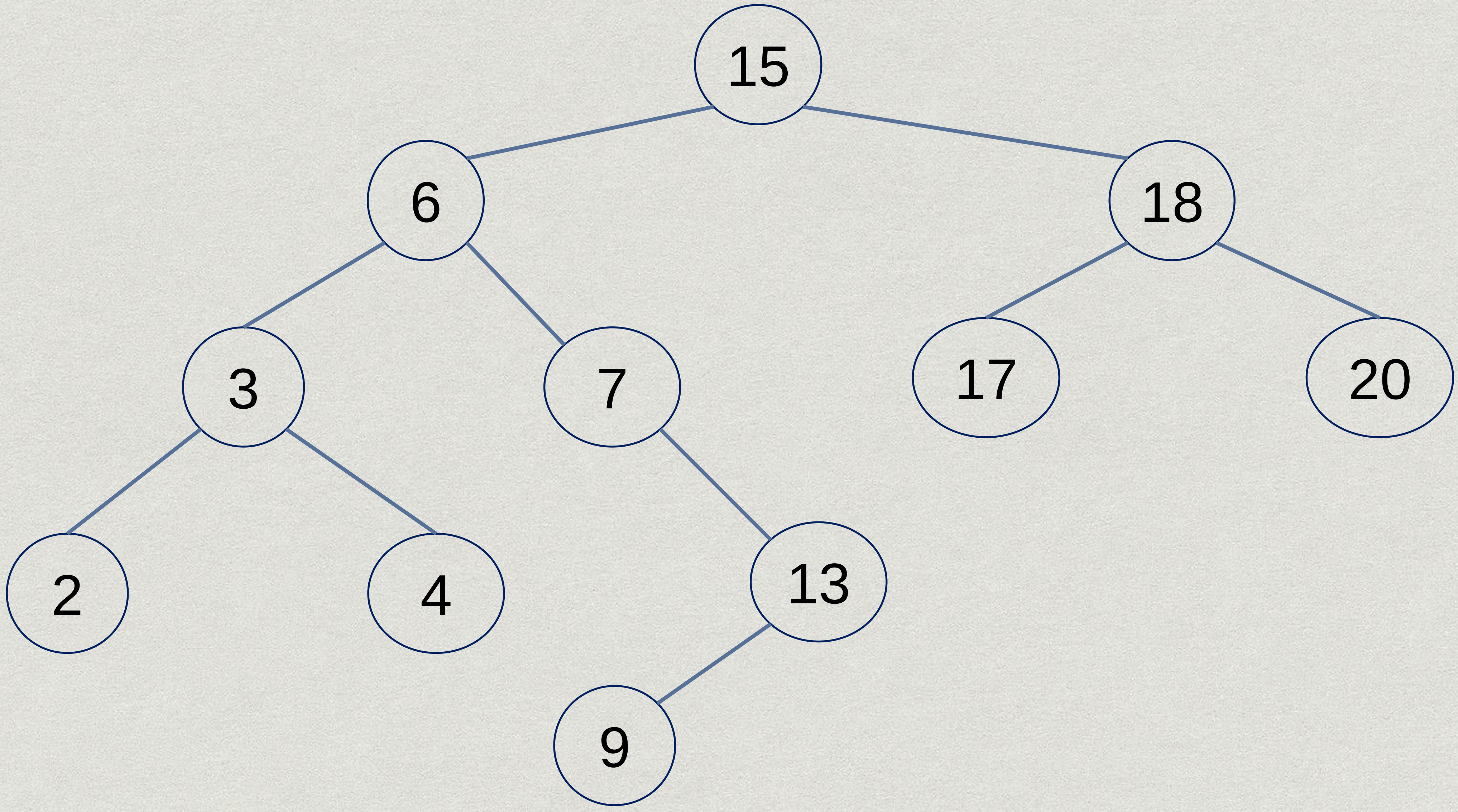
Maximum is found by going down the right side of the tree till a node without a right child

Finding minimum and maximum



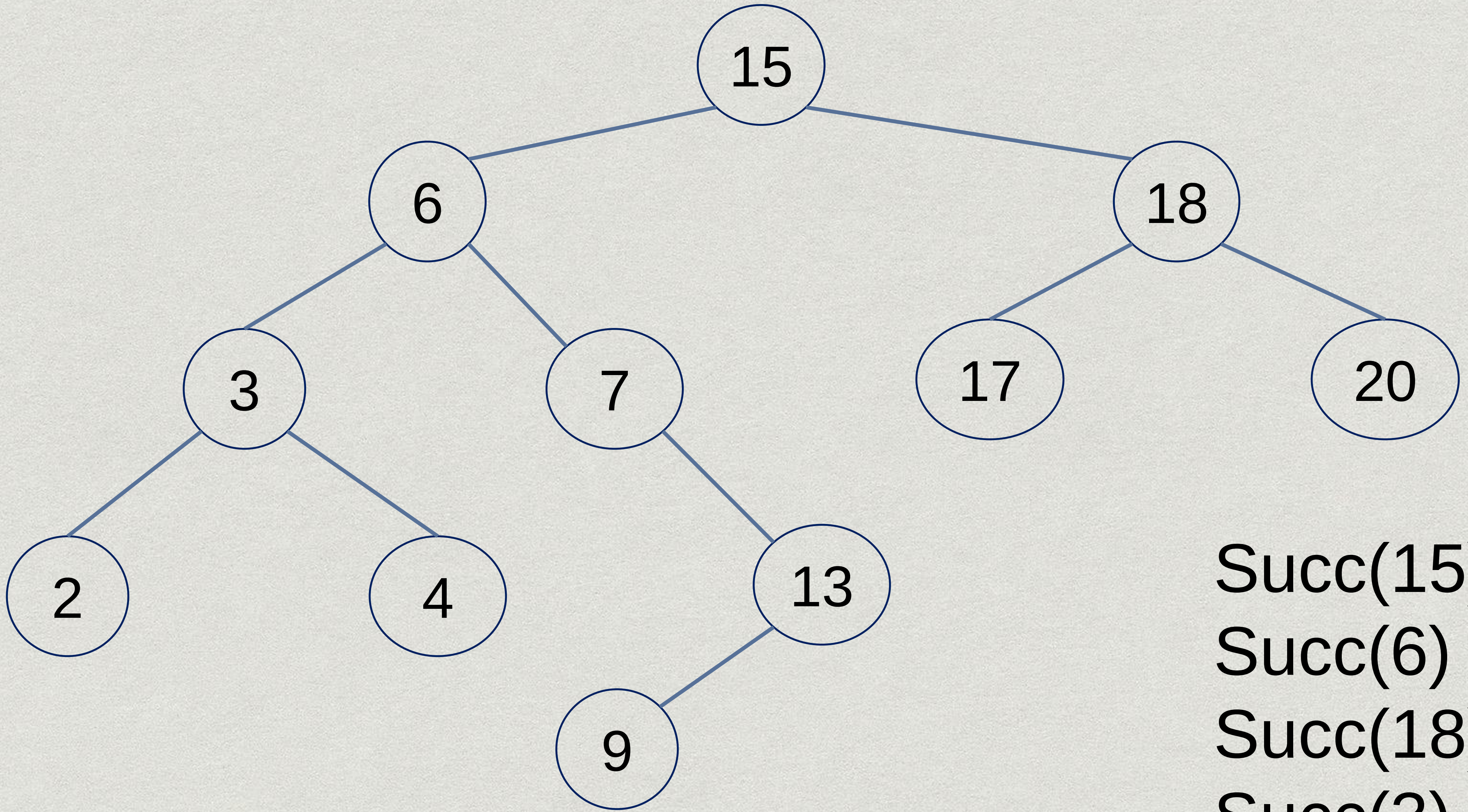
Min is 2 and Max is 20

Finding the successor to a node



If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node



Succ(15) = 17
Succ(6) = 7
Succ(18) = 20
Succ(3) = 4
Succ(7) = 9

If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

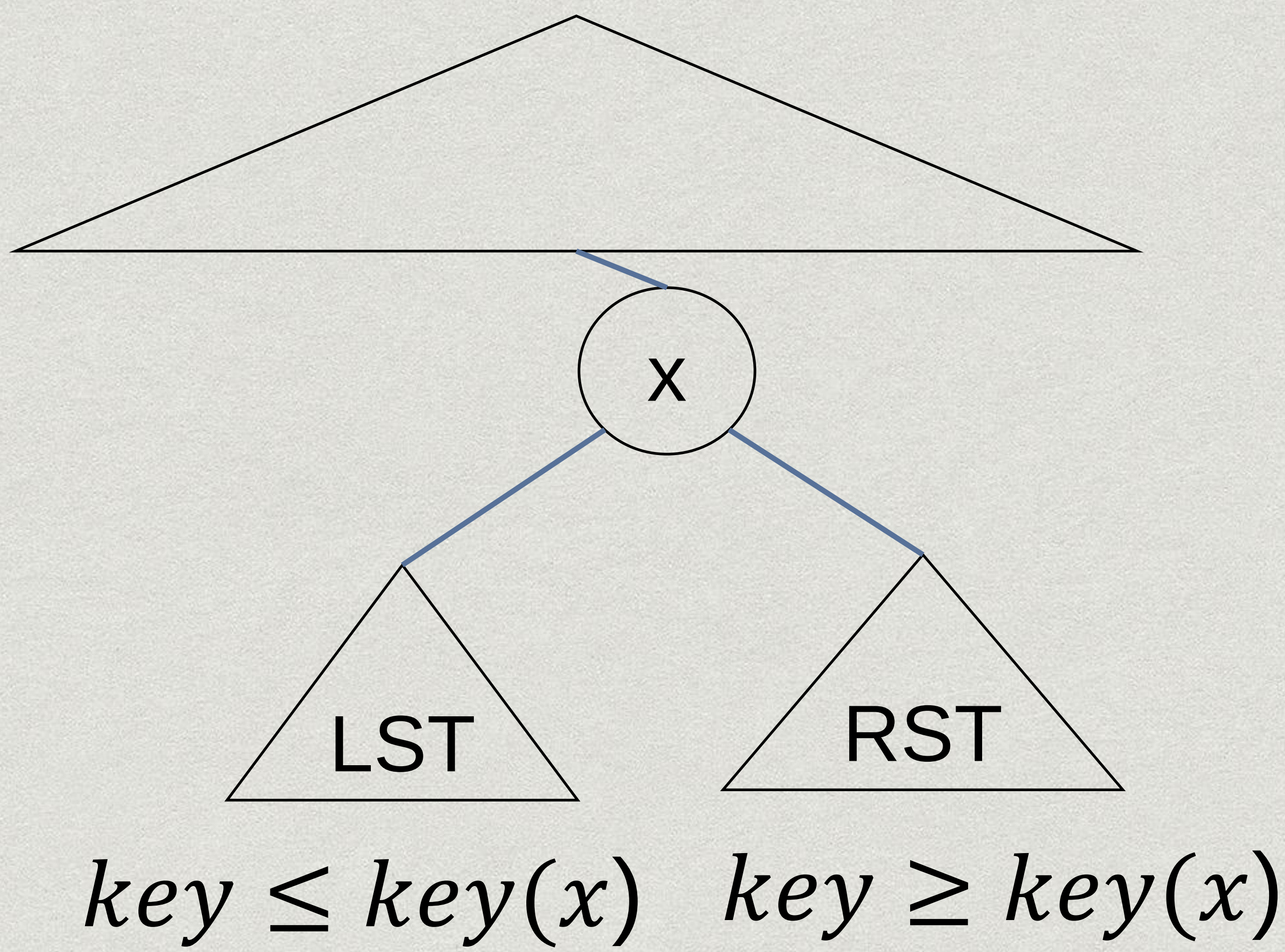
Why is this correct?

If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

Why is this correct?

Consider a node x located somewhere in the tree



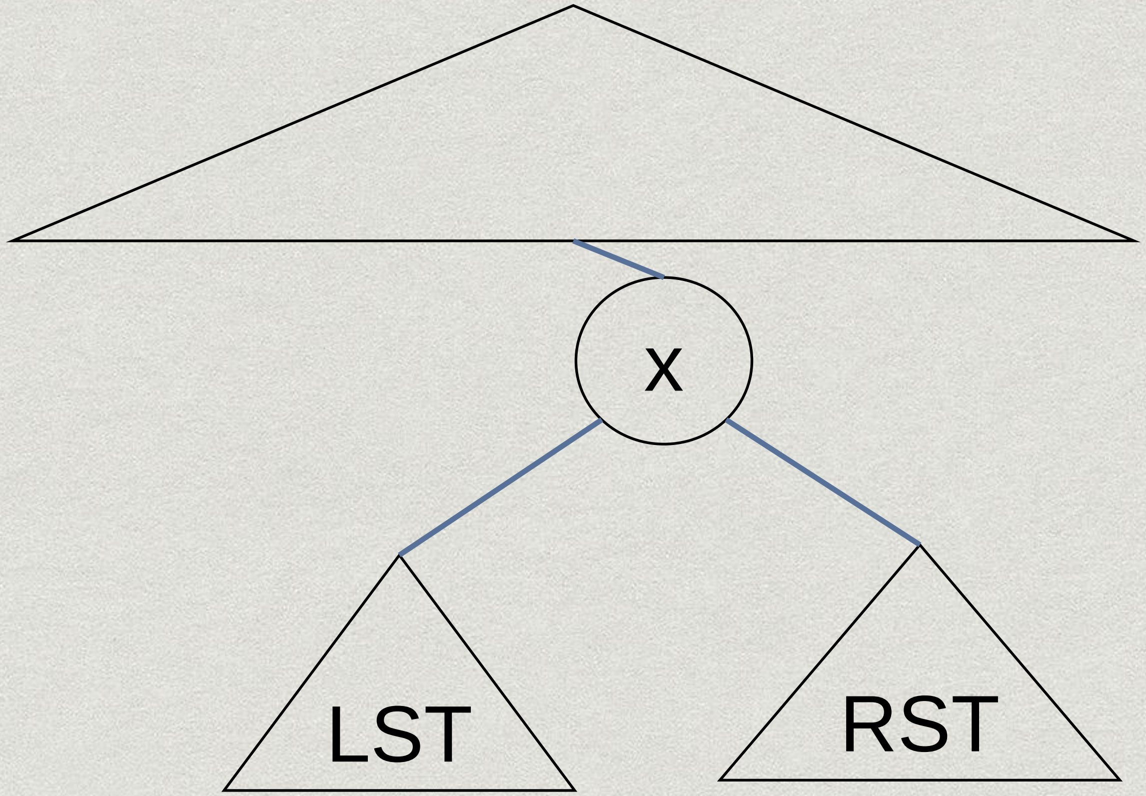
If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

Why is this correct?

Consider a node x located somewhere in the tree

Remember inorder tree walk?



$key \leq key(x)$ $key \geq key(x)$

If a node has a right child,

then the succ is the min of the right subtree of the node

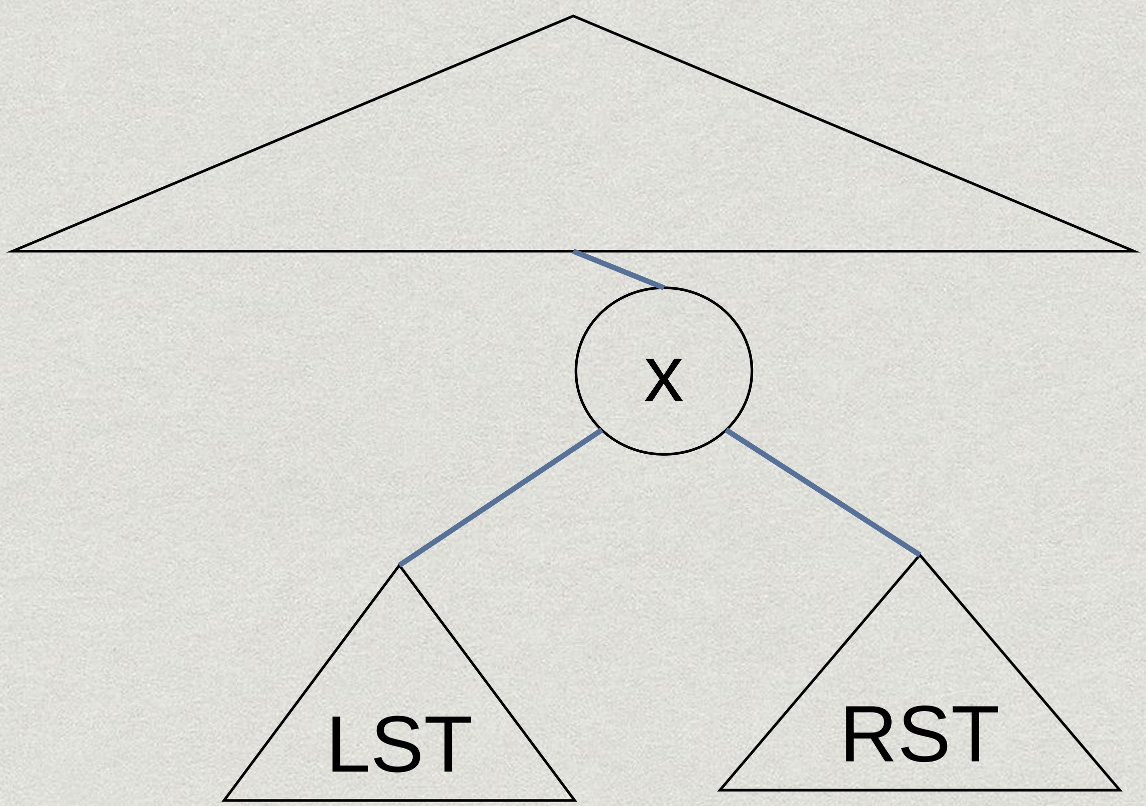
Finding the successor to a node

Why is this correct?

Consider a node x located somewhere in the tree

Remember inorder tree walk?

Produces keys in sorted order

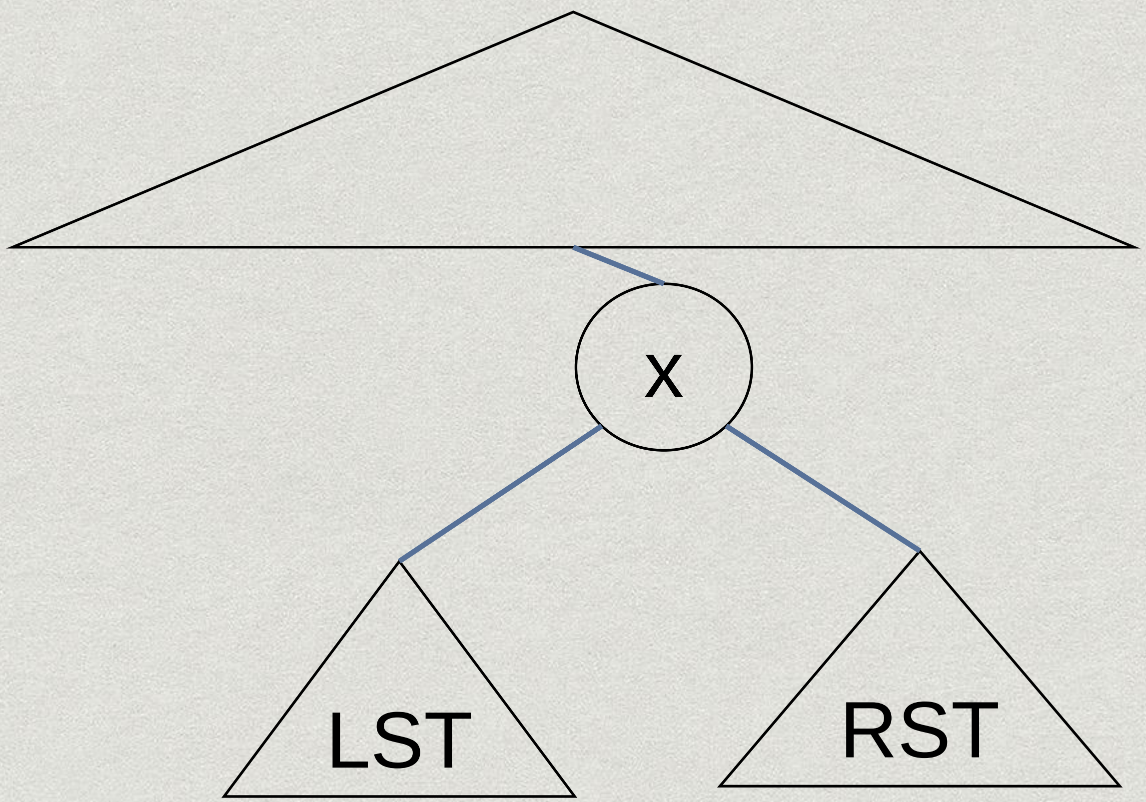


$key \leq key(x)$ $key \geq key(x)$

If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

Why is this correct?



$key \leq key(x)$ $key \geq key(x)$

Consider a node x located somewhere in the tree

Remember inorder tree walk?

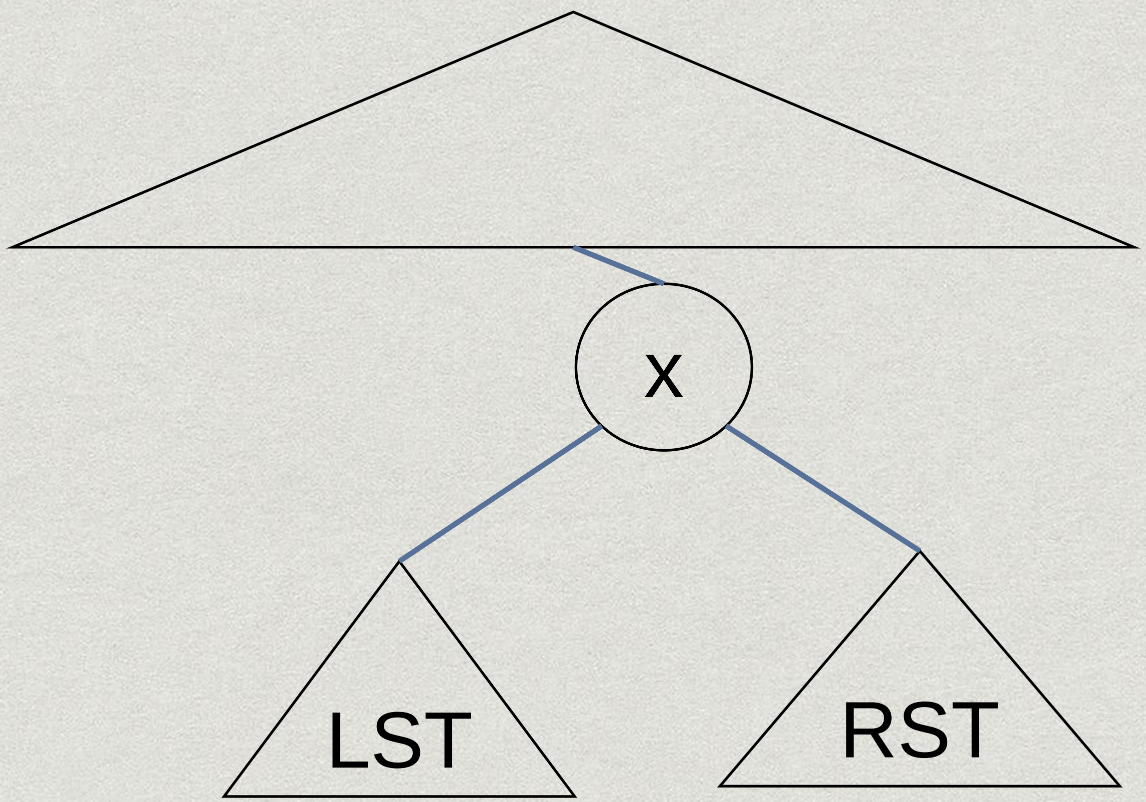
Produces keys in sorted order

…
Keys from LST
Key(x)
Keys from RST
…

If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

Why is this correct?



$key \leq key(x)$ $key \geq key(x)$

Consider a node x located somewhere in the tree

Remember inorder tree walk?

Produces keys in sorted order

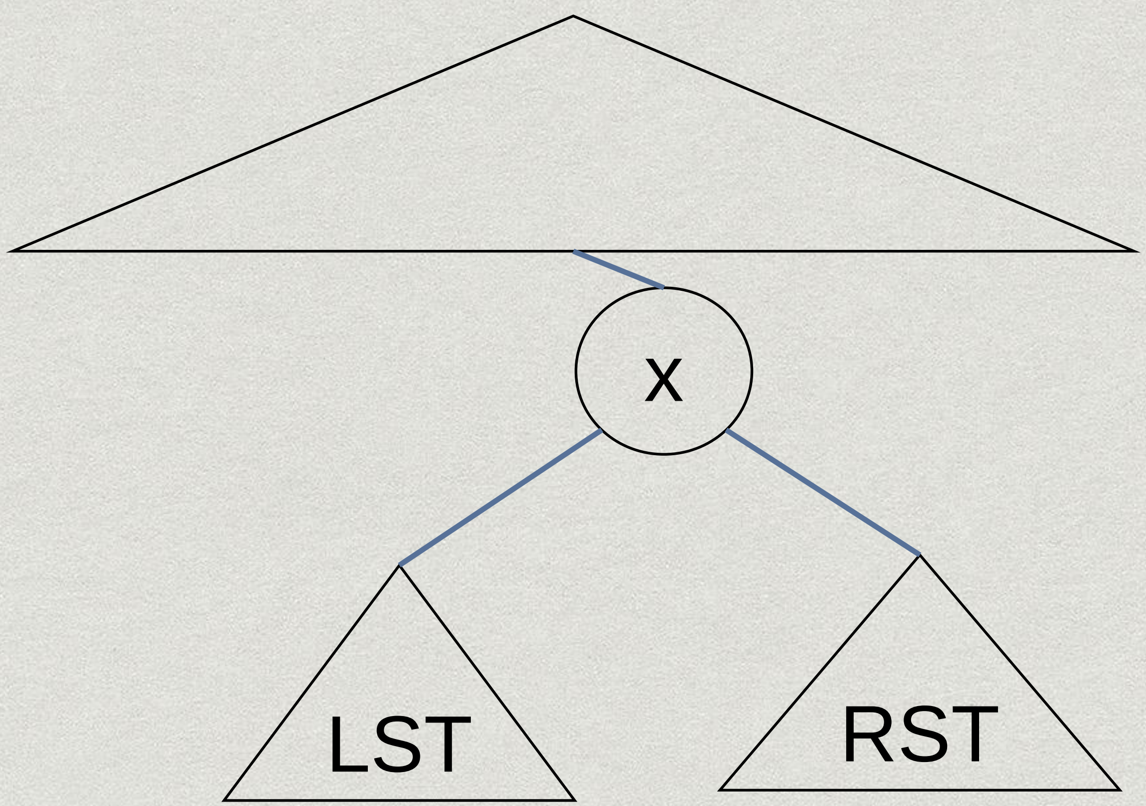
…
Keys from LST
Key(x)
Keys from RST
…

Min of right subtree has to be Succ

If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node

Why is this correct?



$key \leq key(x)$ $key \geq key(x)$

Consider a node x located somewhere in the tree

Remember inorder tree walk?

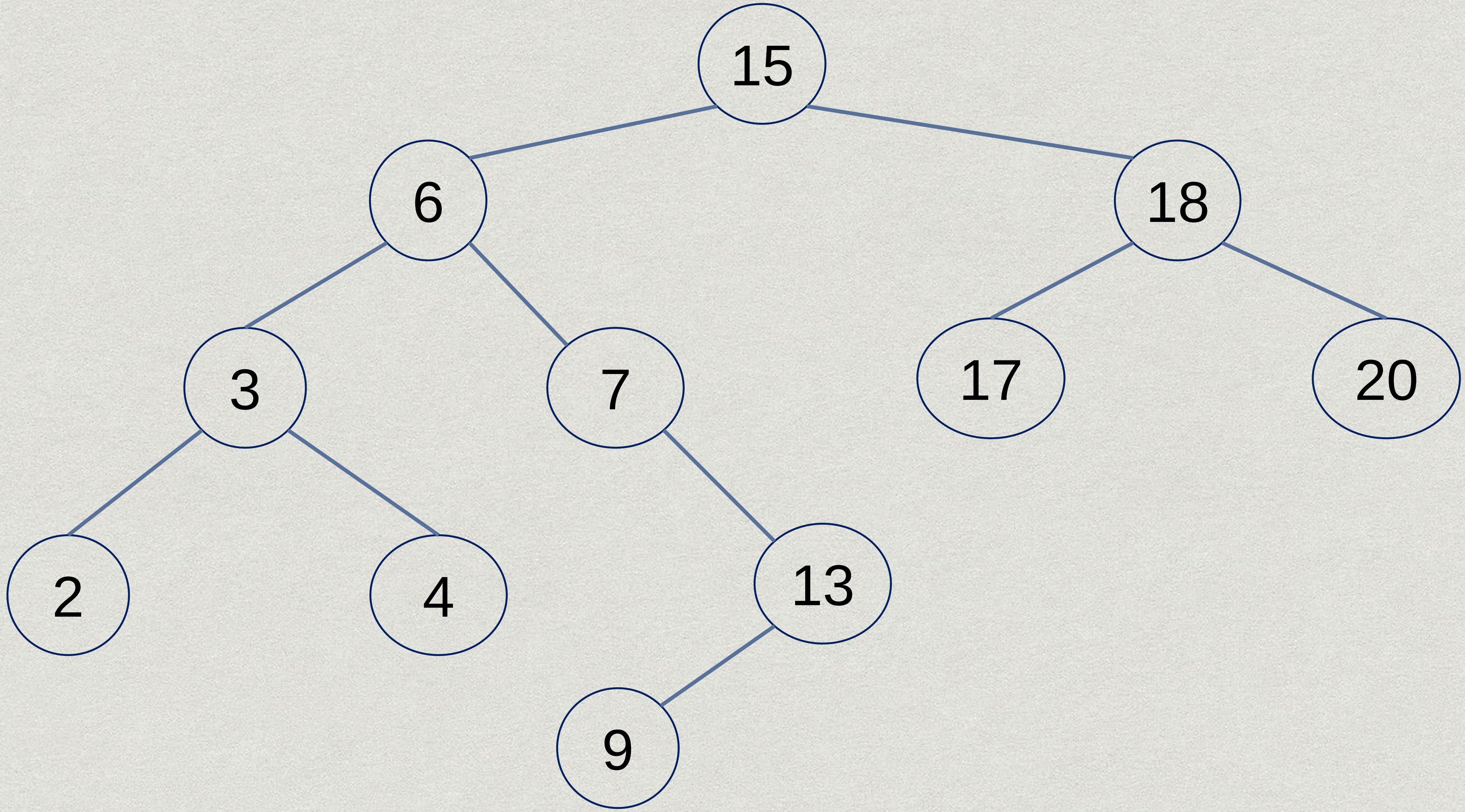
Produces keys in sorted order

…
Keys from LST
Key(x)
Keys from RST
…

Min of right subtree has to be Succ

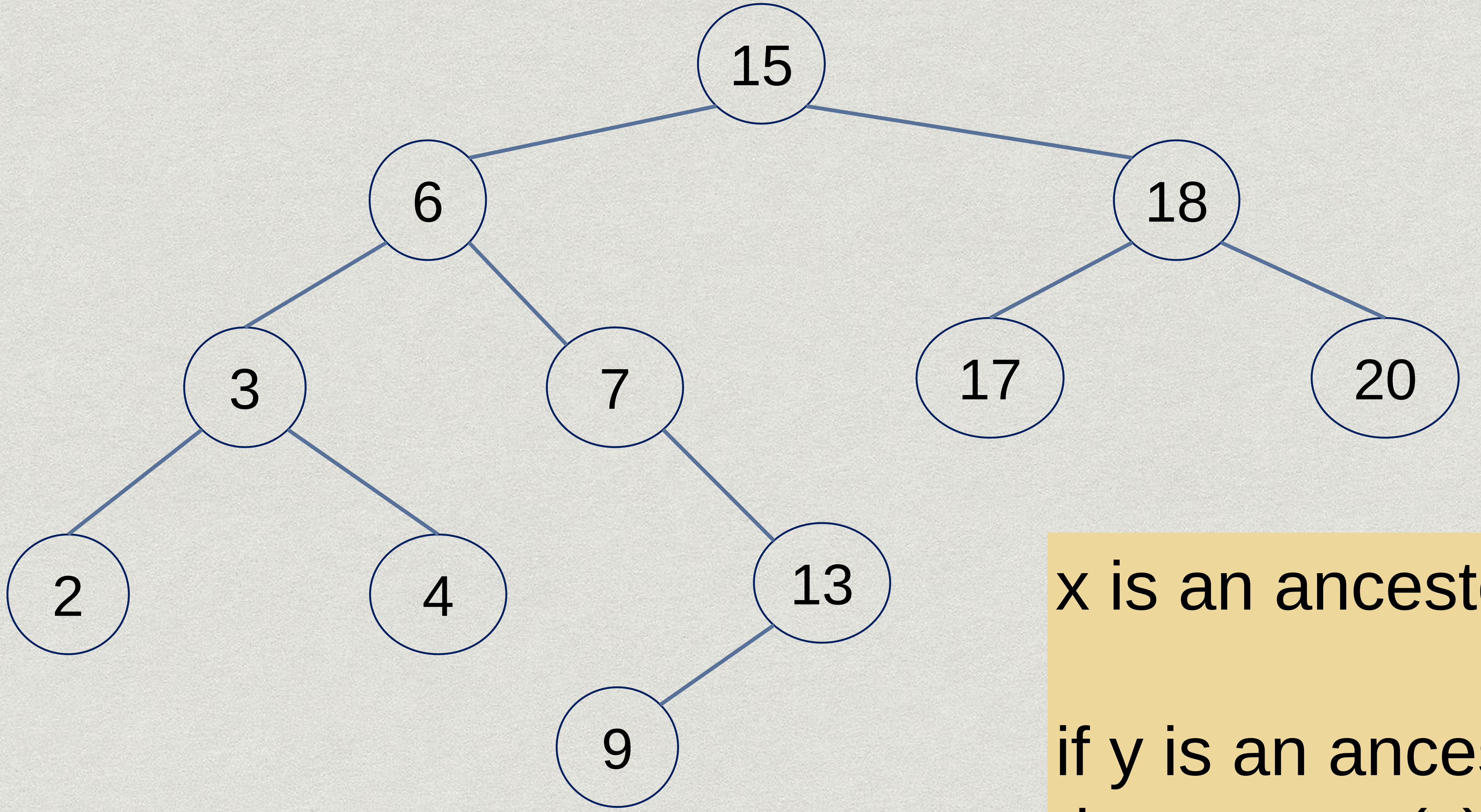
If a node has a right child,
then the succ is the min of the right subtree of the node

Finding the successor to a node



If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

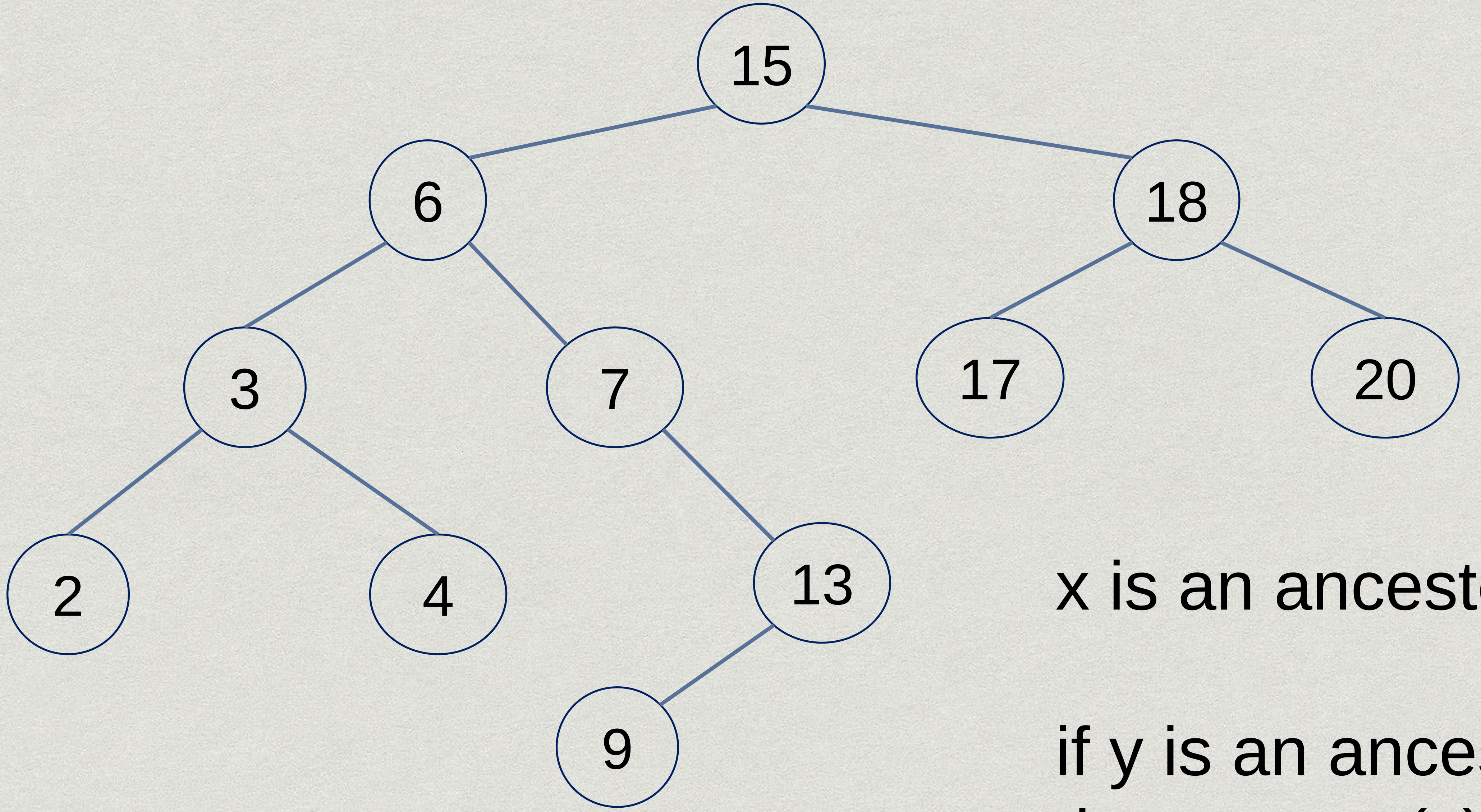


x is an ancestor of x

if y is an ancestor of x, then parent(y) is an ancestor of x

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node



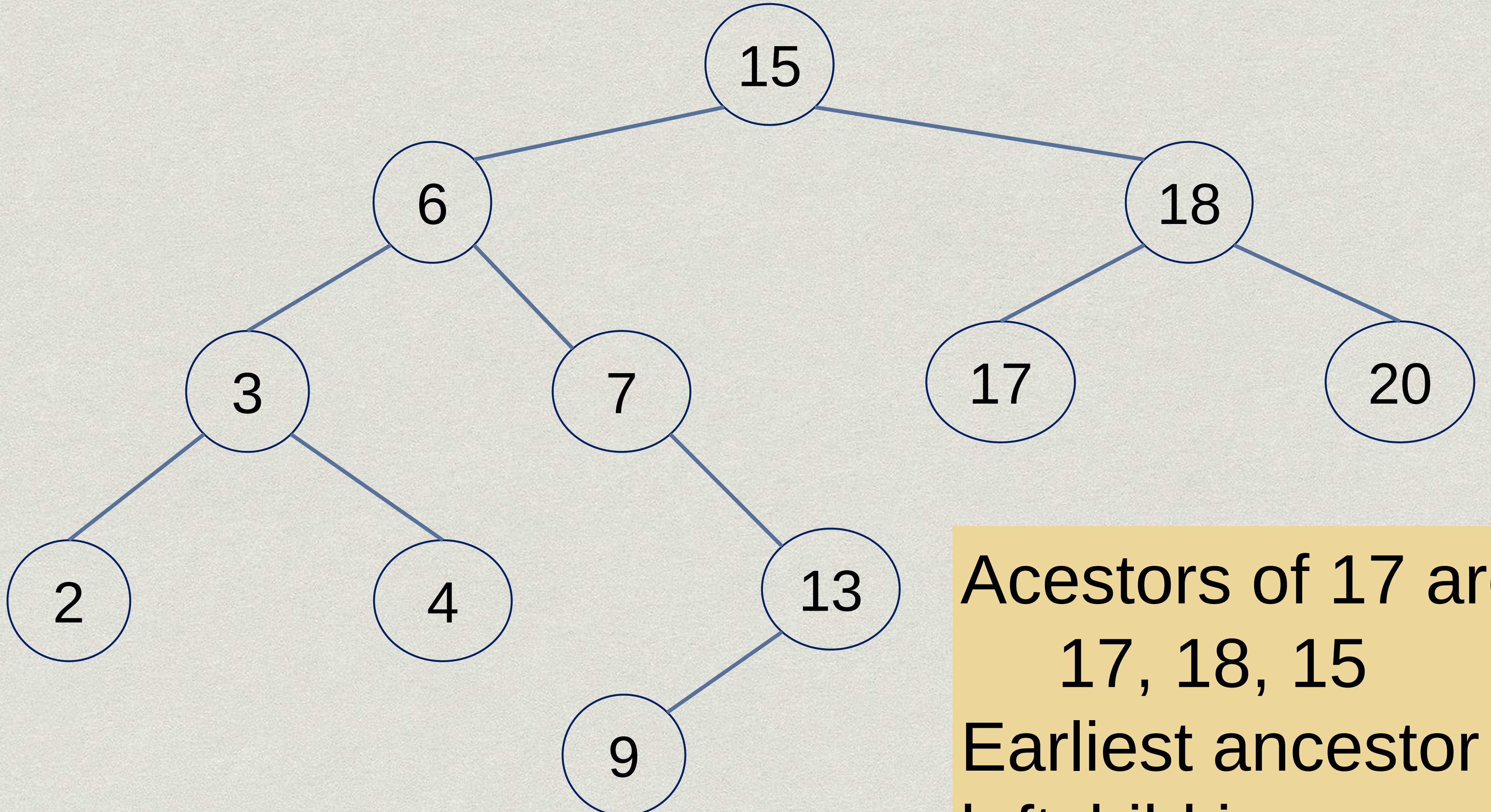
Acestors of 9 are 9, 13, 7, 6, 15

x is an ancestor of x

if y is an ancestor of x, then parent(y) is an ancestor of x

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

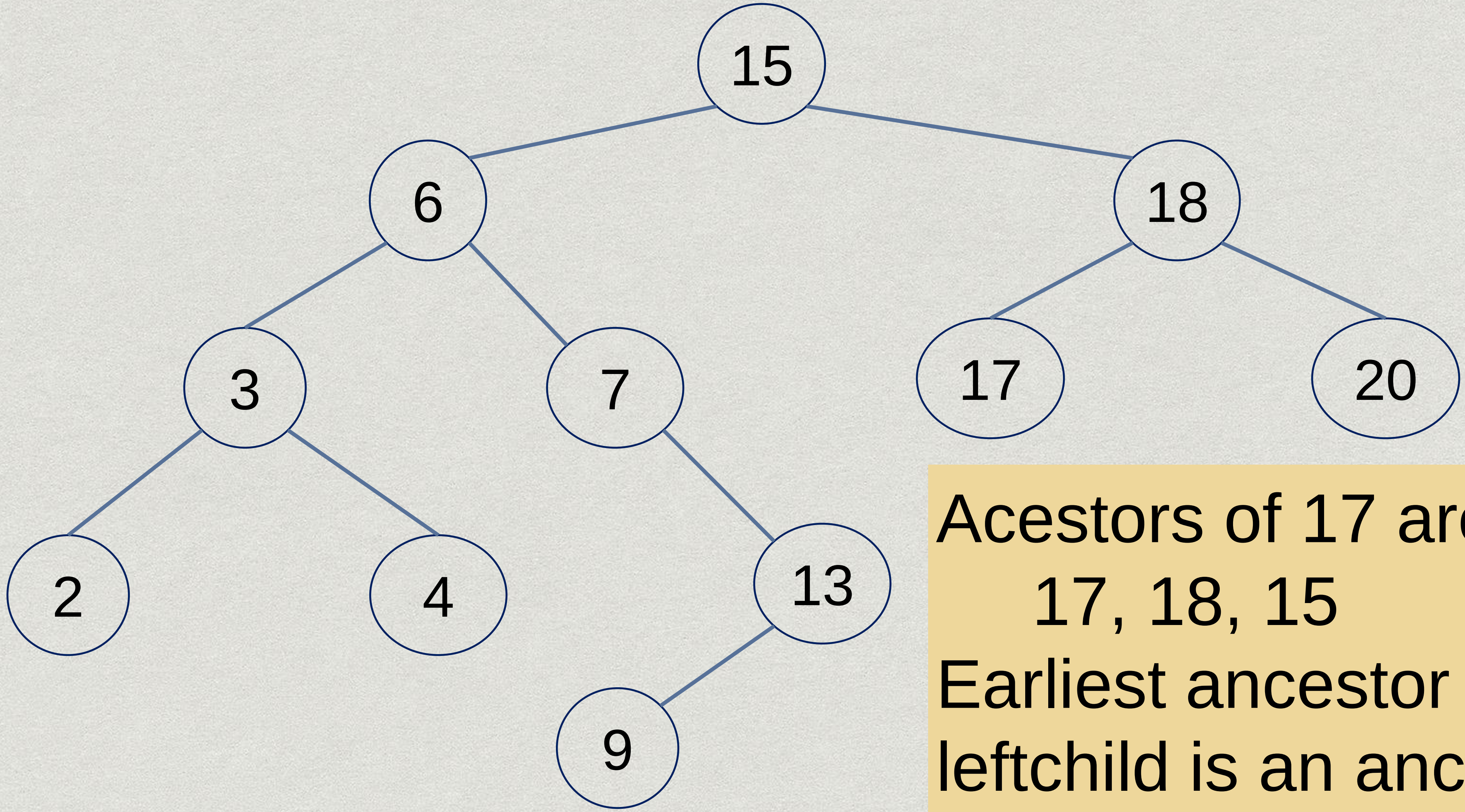
Finding the successor to a node



Acestors of 17 are
17, 18, 15
Earliest ancestor whose leftchild is an ancestor is 18

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

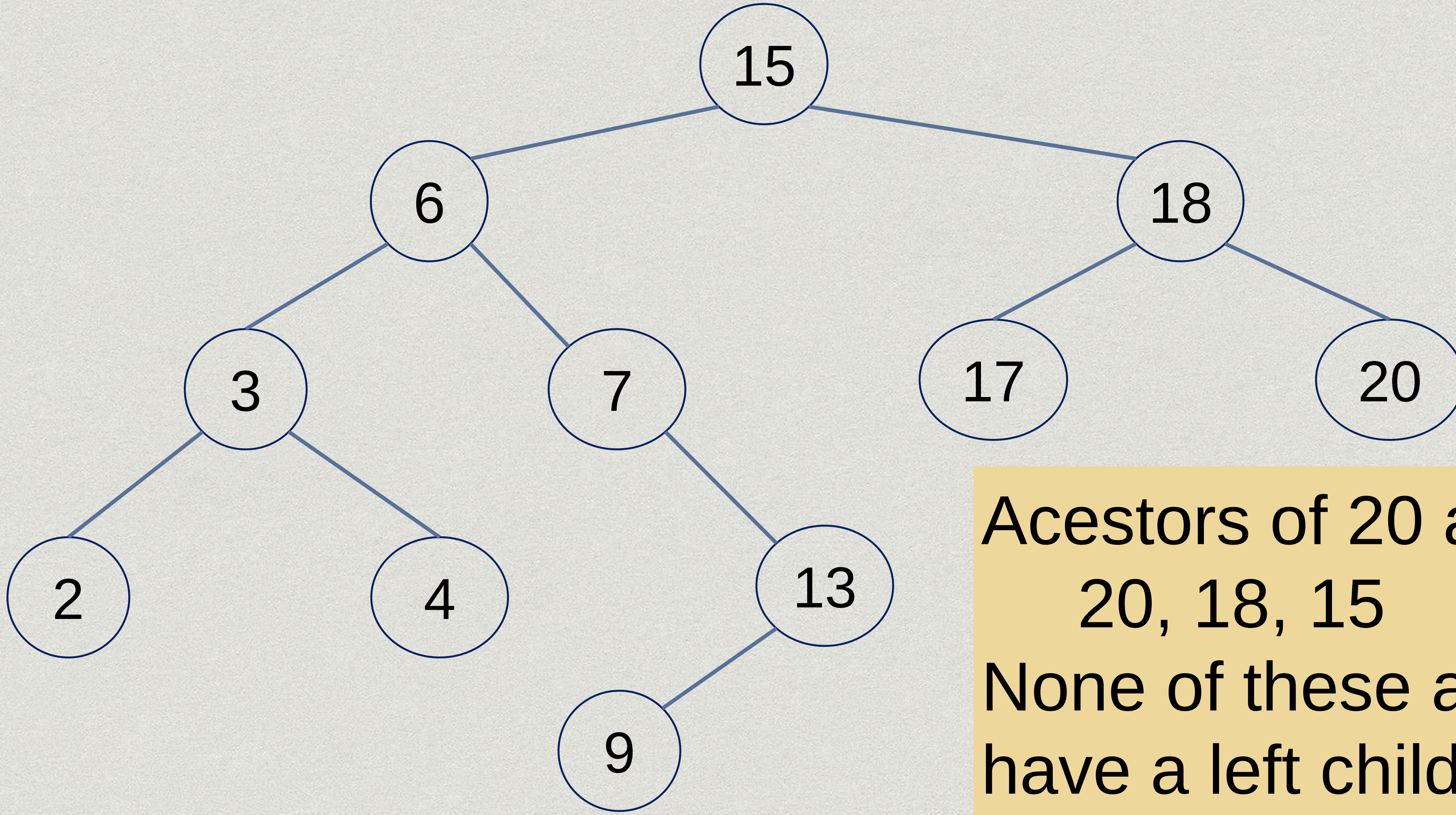


Acestors of 17 are
17, 18, 15
Earliest ancestor whose leftchild is an ancestor is 18

Succ(17) = 18

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

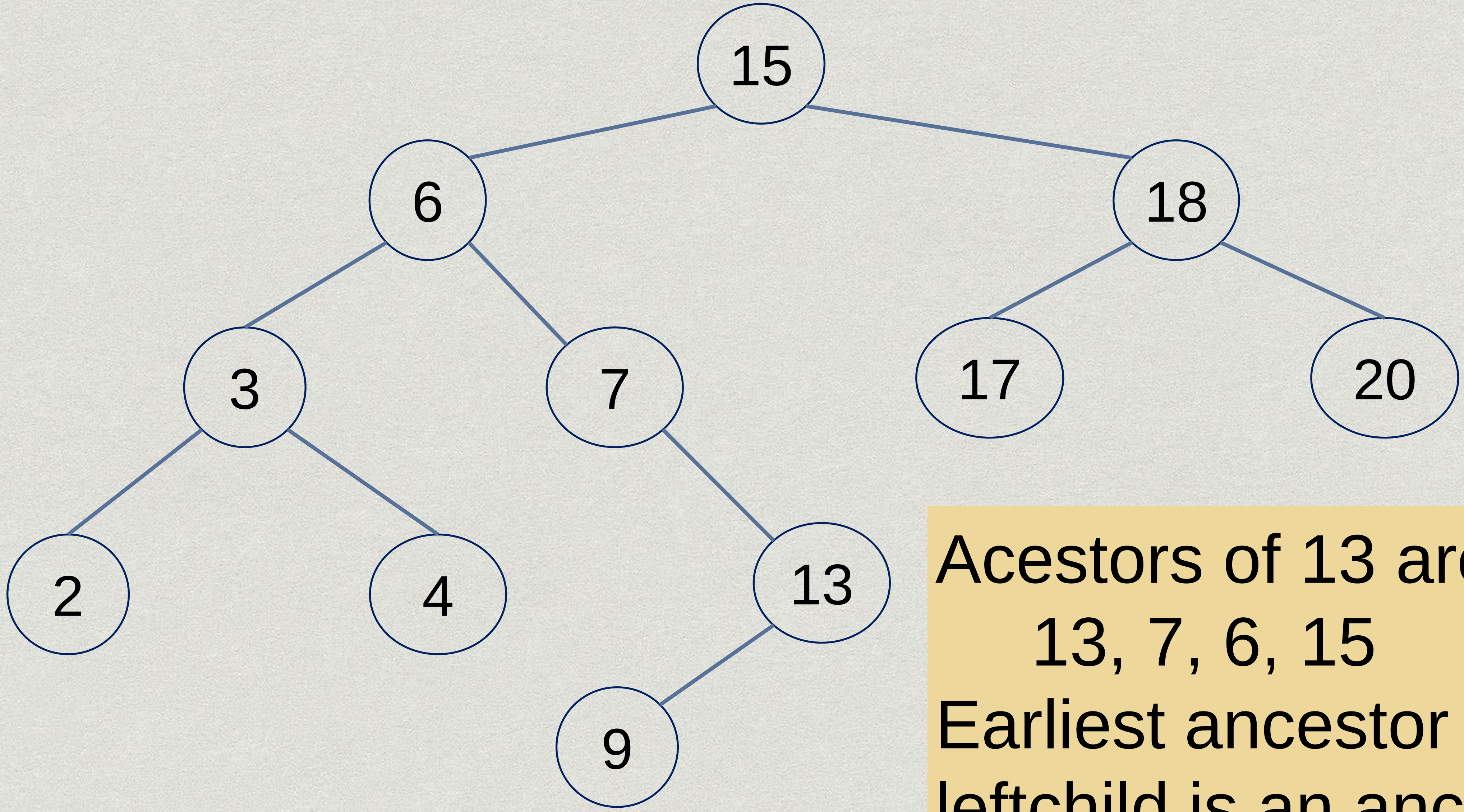
Finding the successor to a node



Acestors of 20 are
20, 18, 15
None of these ancestors have a left child which is an ancestor
Succ(20) = NIL

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

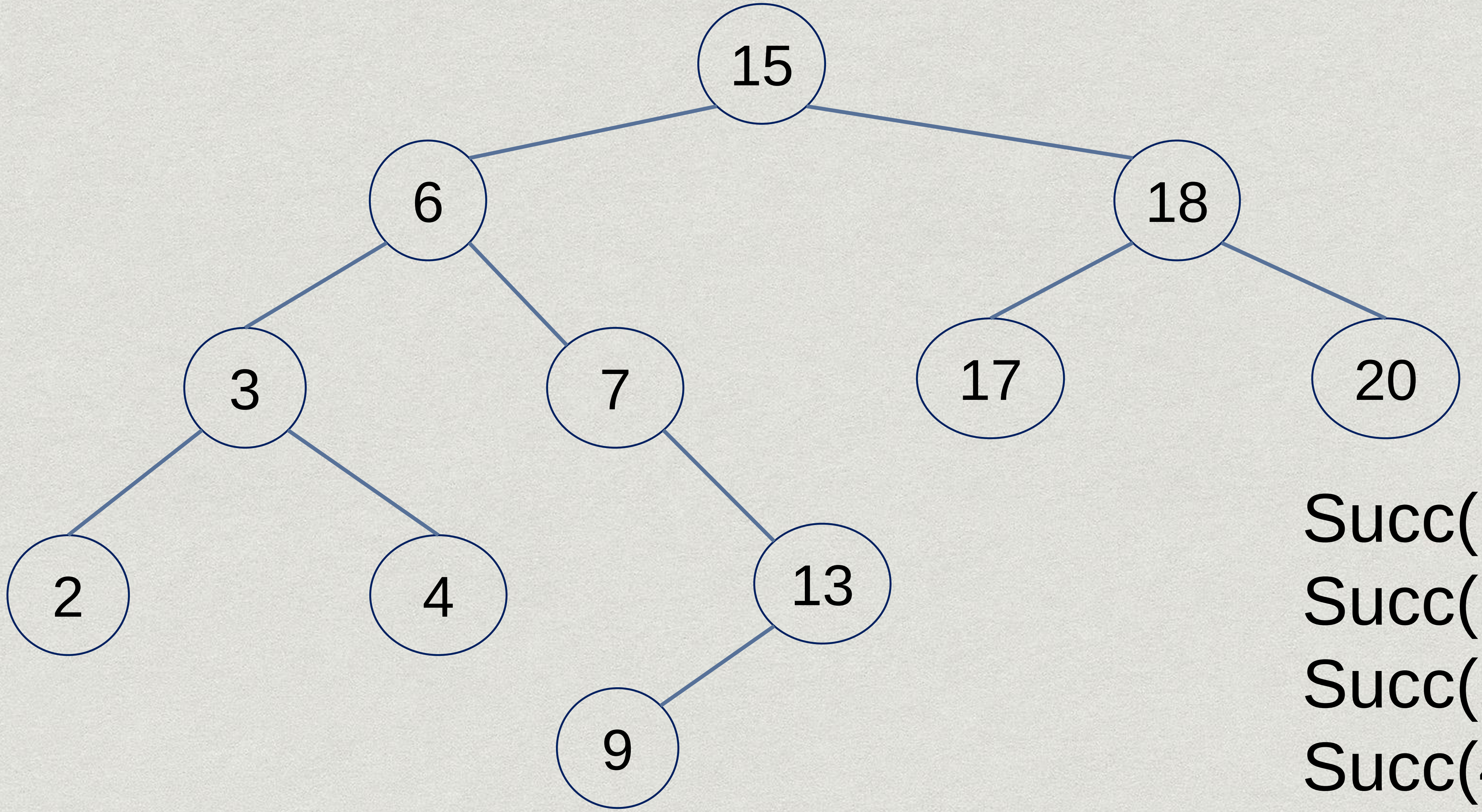
Finding the successor to a node



Acestors of 13 are
13, 7, 6, 15
Earliest ancestor whose leftchild is an ancestor is 15
Succ(13) = 15

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node



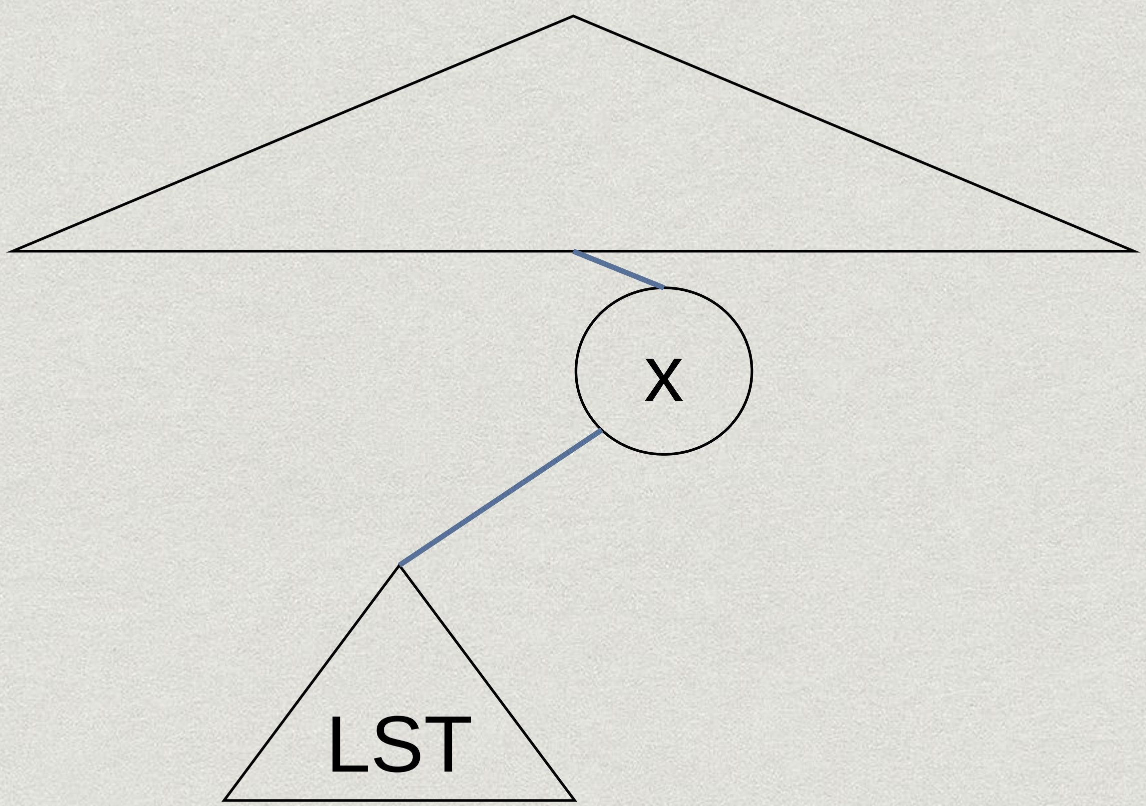
Succ(17) = 18
Succ(20) = NIL
Succ(2) = 3
Succ(4) = 6
Succ(13) = 15
Succ(9) = 13

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

Why is this correct?

Consider a node x located somewhere in the tree

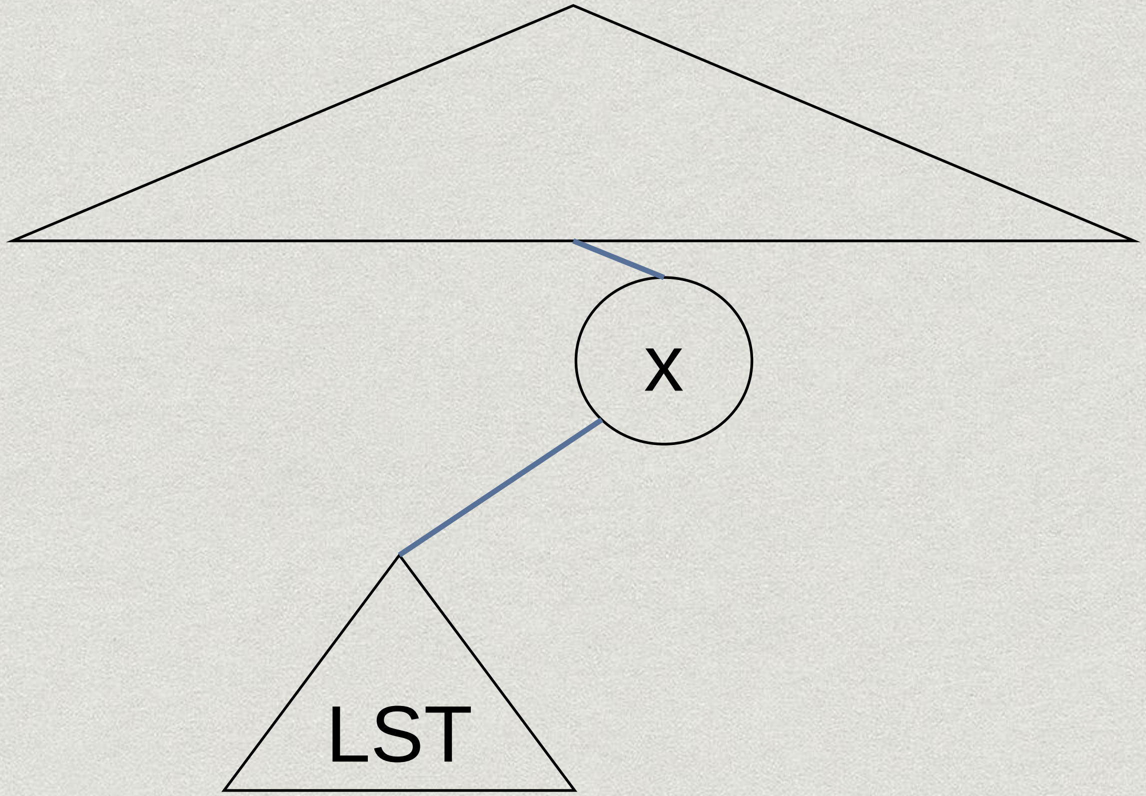


$key \leq key(x)$

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

Why is this correct?



$key \leq key(x)$

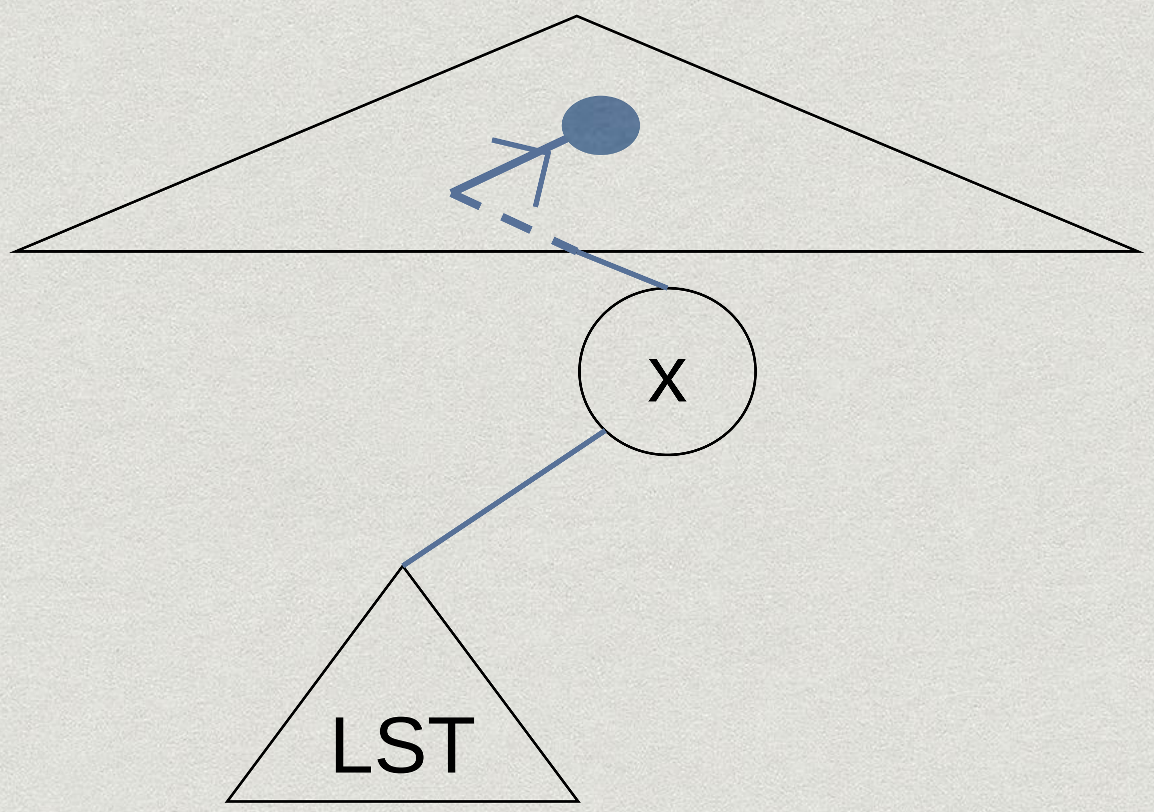
Consider a node x located somewhere in the tree

What is the next key printed by inorder tree walk after x?

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

Why is this correct?



$key \leq key(x)$

Consider a node x located somewhere in the tree

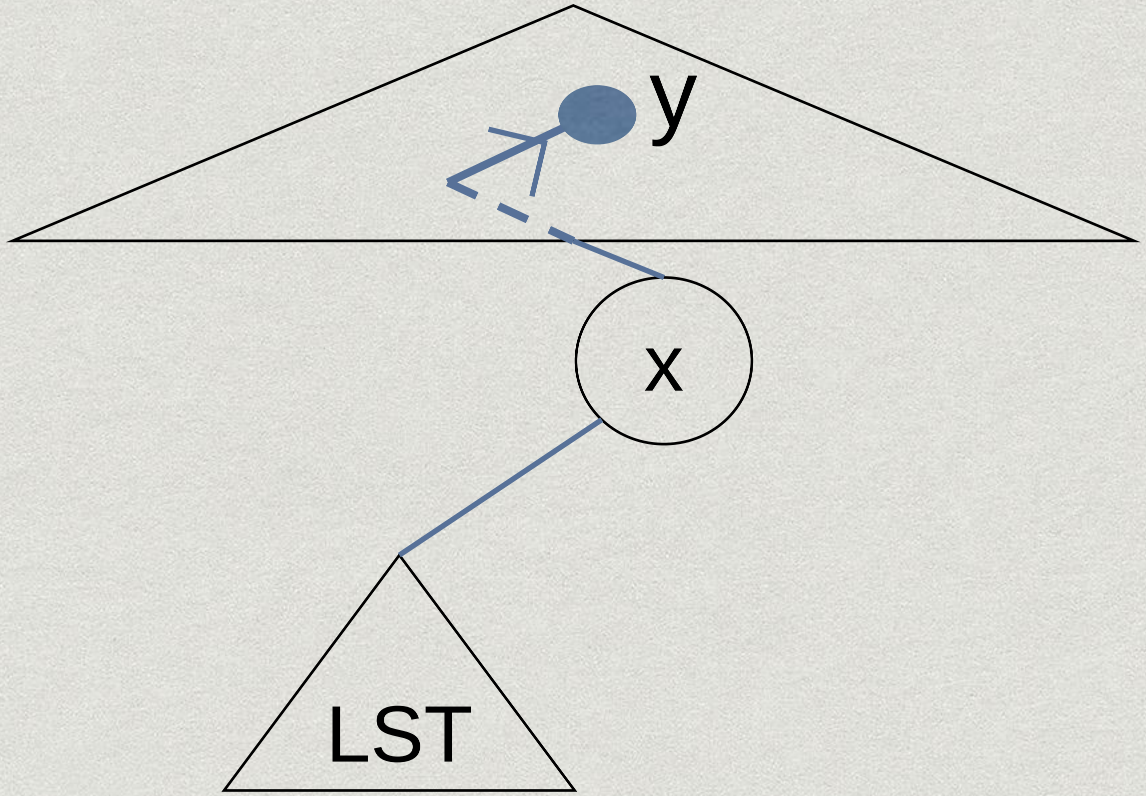
What is the next key printed by inorder tree walk after x?

Walk up from the tree till you first turn right – that is the node that gets printed next

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

Why is this correct?



$key \leq key(x)$

Consider a node x located somewhere in the tree

What is the next key printed by inorder tree walk after x?

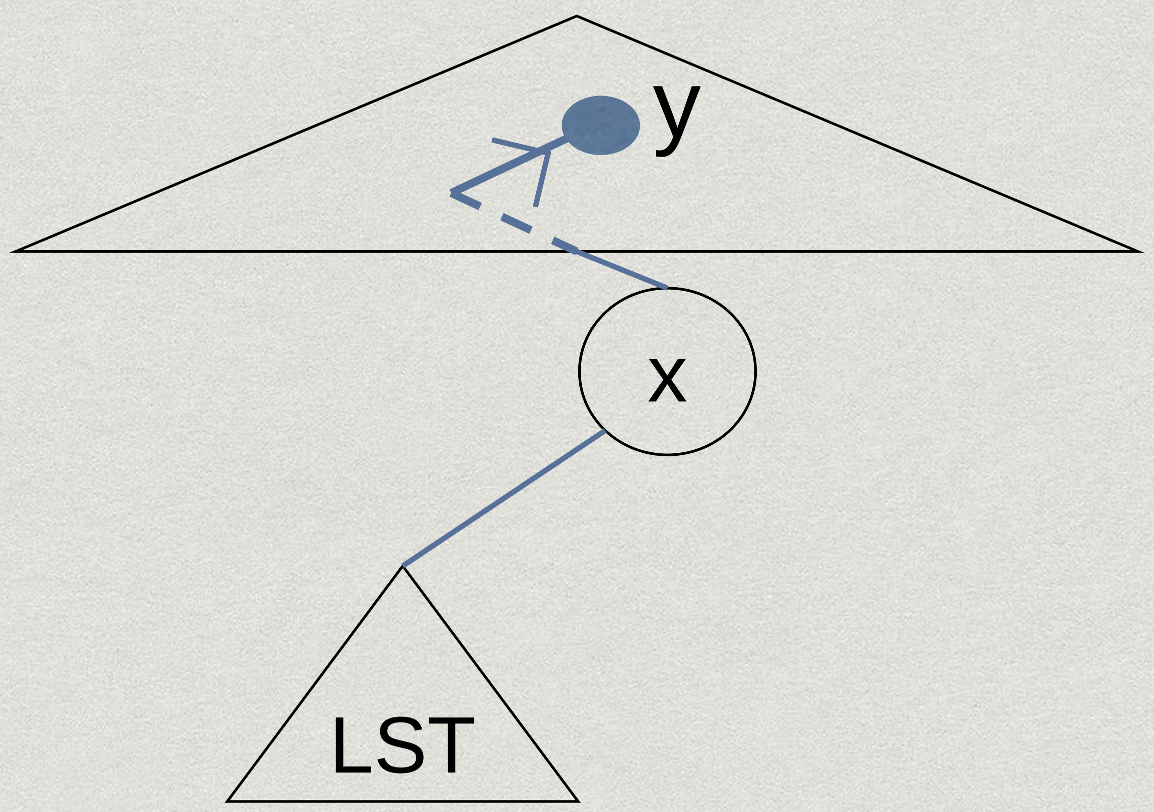
Walk up from the tree till you first turn right – that is the node that gets printed next

Succ(x) = y

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

Why is this correct?



$key \leq key(x)$

Consider a node x located somewhere in the tree

What is the next key printed by inorder tree walk after x?

Walk up from the tree till you first turn right – that is the node that gets printed next

Succ(x) = y

If a node does not have a right child, then the succ is earliest ancestor whose leftchild is also an ancestor

Finding the successor to a node

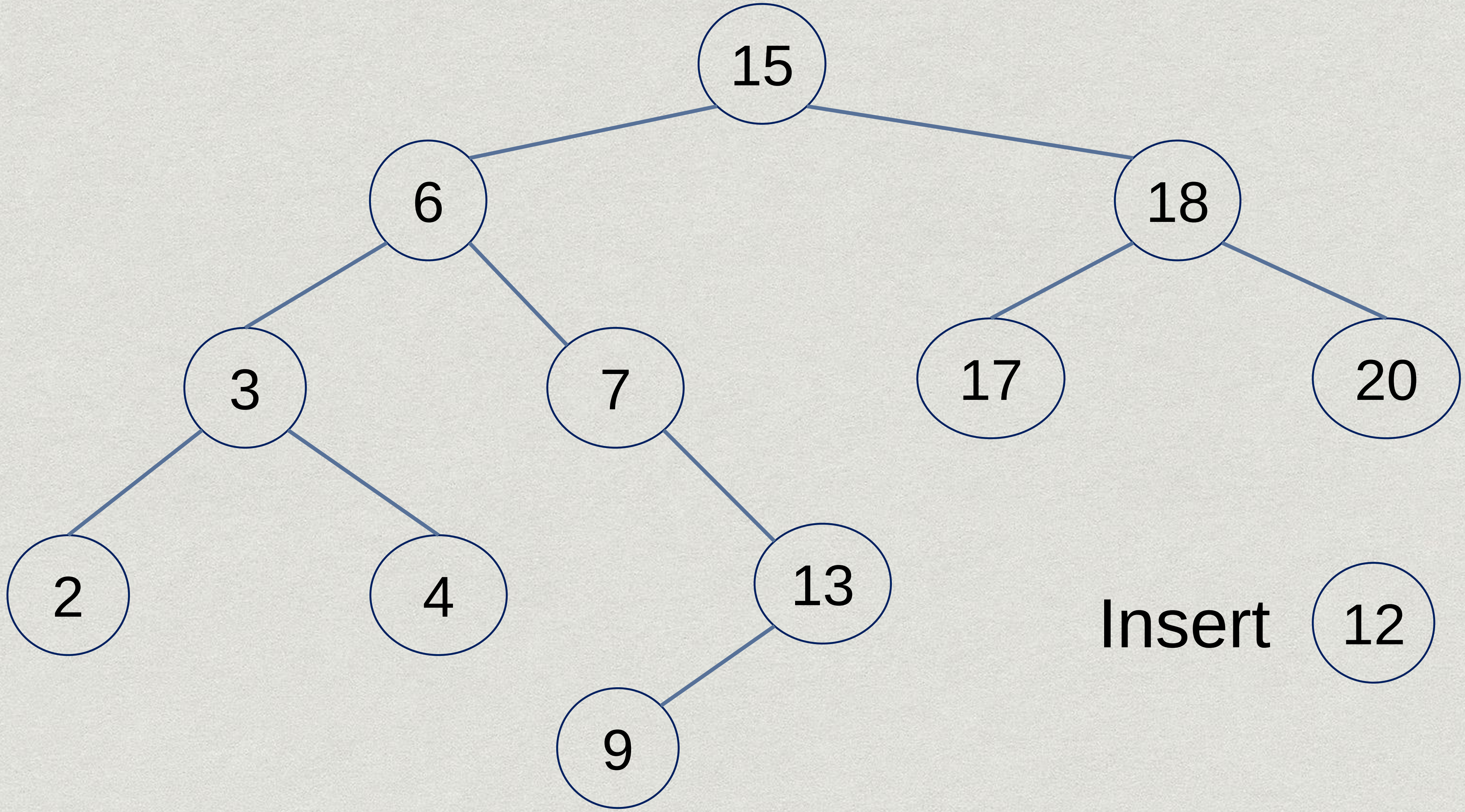
```
Node Tree_Succ(x) {

    if rightChild(x) ≠ NIL:
        return  Tree_Min( rightChild(x) )
    else
        y = parent(x)
        while y ≠ NIL and x == rightChild(y)
            x = y
            y = parent(y)
        endwhile
        return y
    endif
}
```

Finding the predecessor to a node

```
Node Tree_Pred(x) {

    if leftChild(x) ≠ NIL:
        return  Tree_Max( leftChild(x) )
    else
        y = parent(x)
        while y ≠ NIL and x == leftChild(y)
            x = y
            y = parent(y)
        endwhile
        return y
    endif
}
```

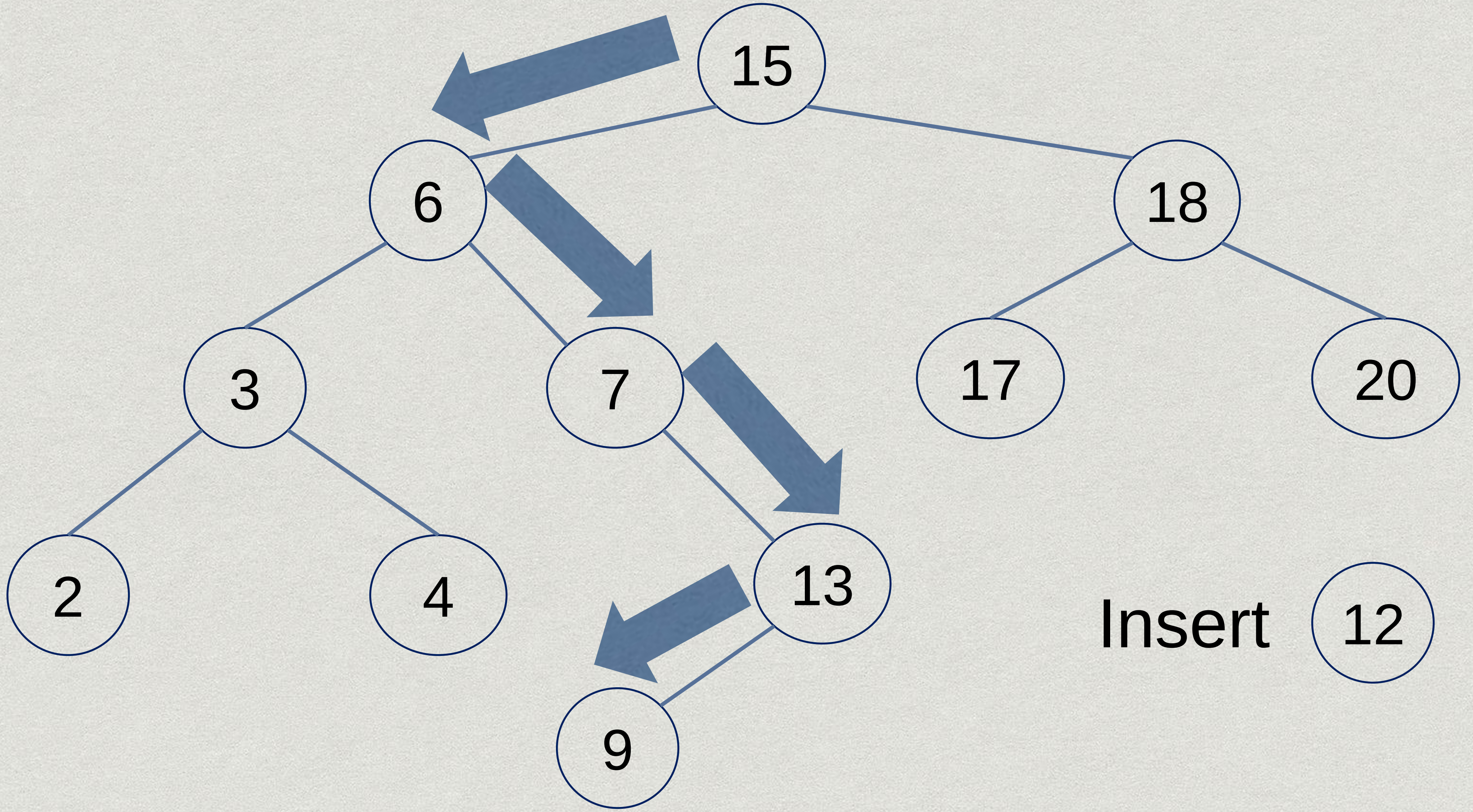
Predecessor method is very similar to successor – take leftChild instead of rightChild

Inserting a node



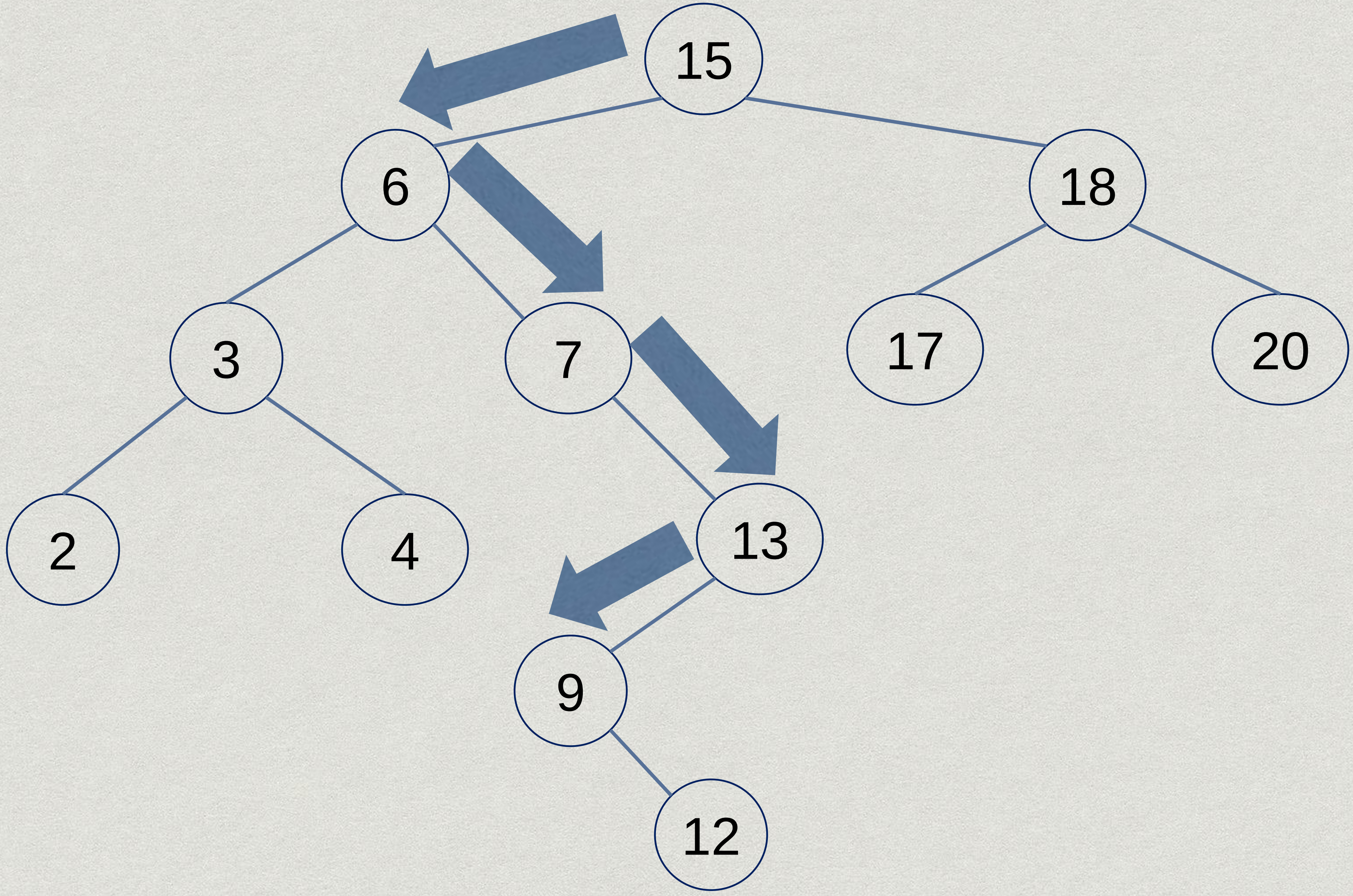
Follow the same method as searching for 12
Then insert it at the right place

Inserting a node



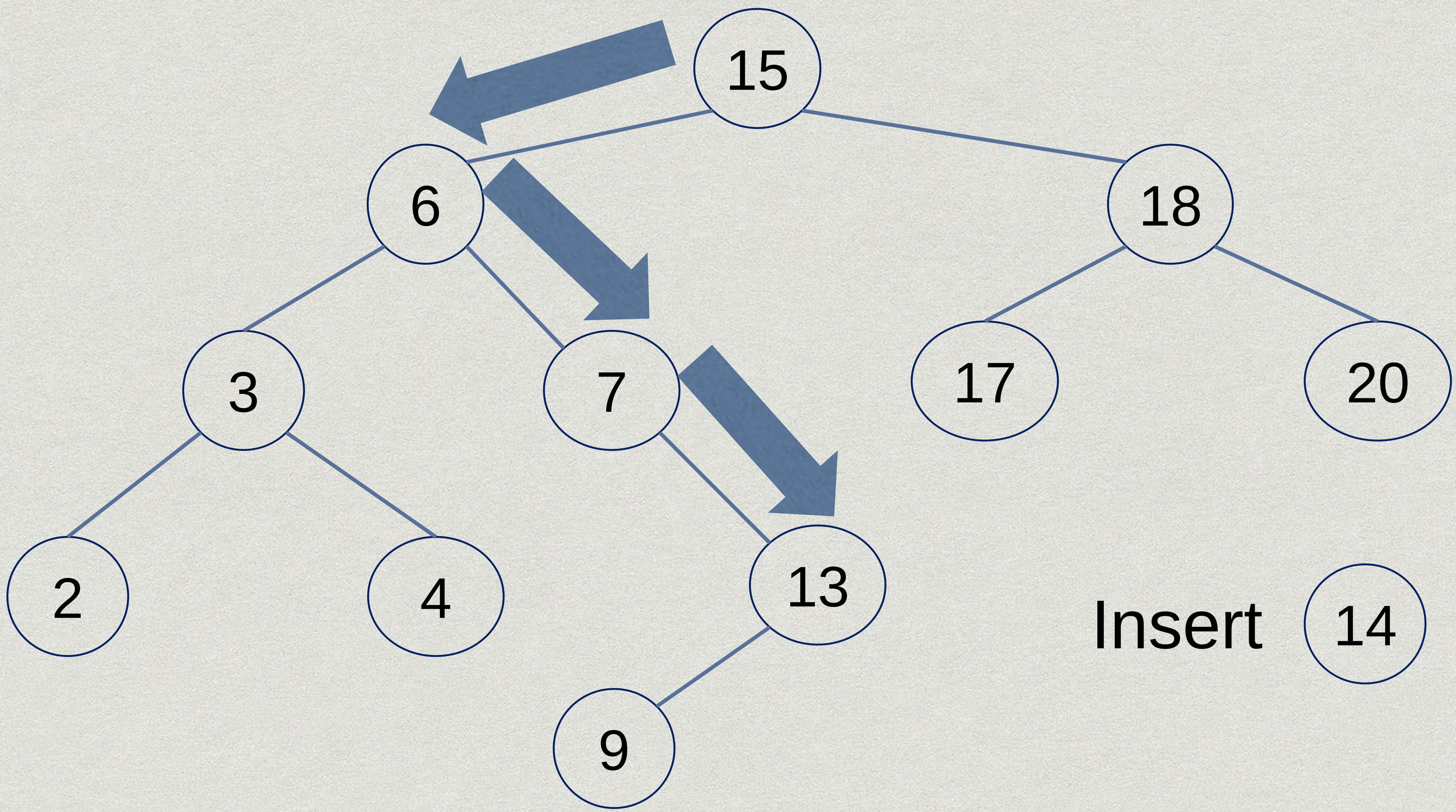
Follow the same method as searching for 12
Then insert it at the right place

Inserting a node



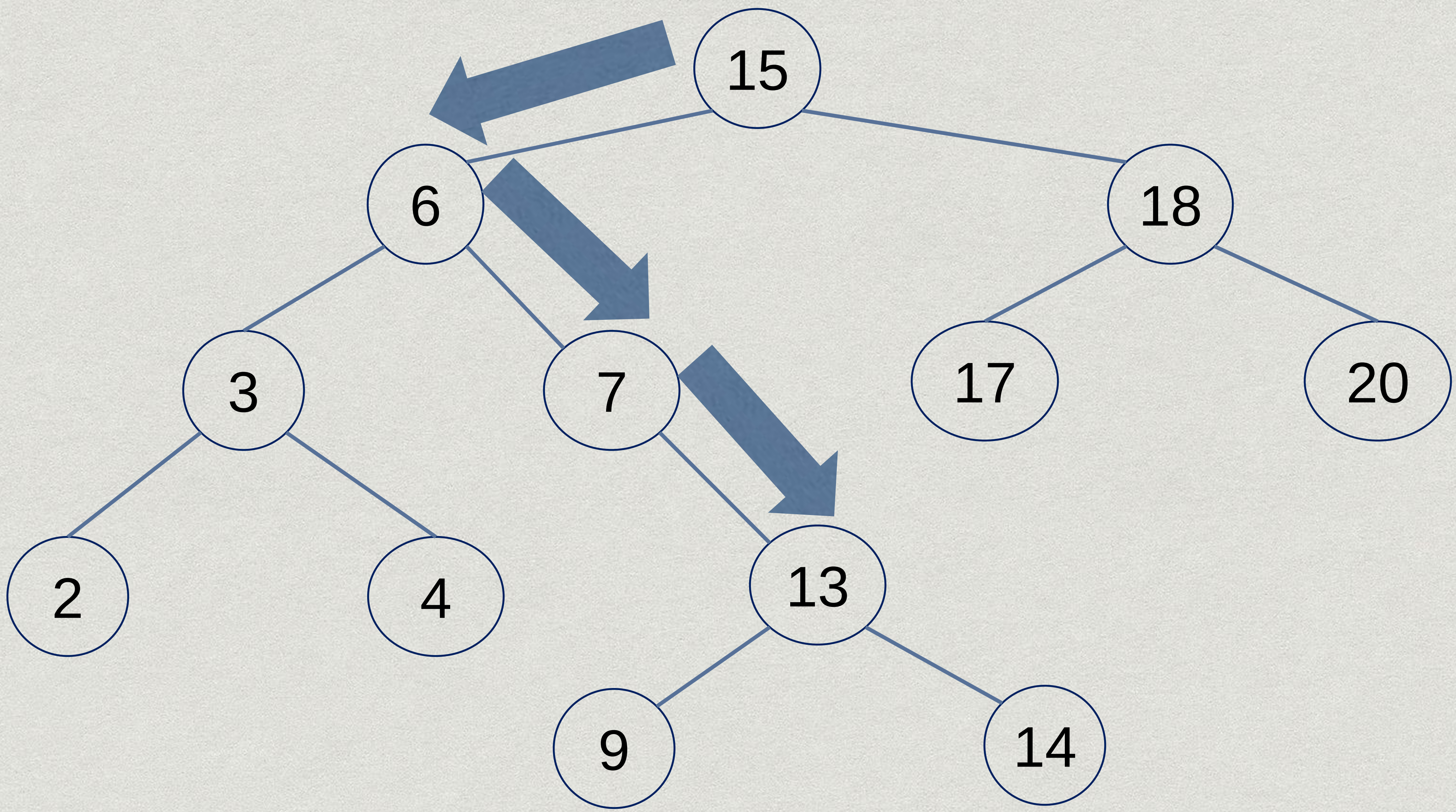
Follow the same method as searching for 12
Then insert it at the right place

Inserting a node



Follow the same method as searching for 14
Then insert it at the right place

Inserting a node

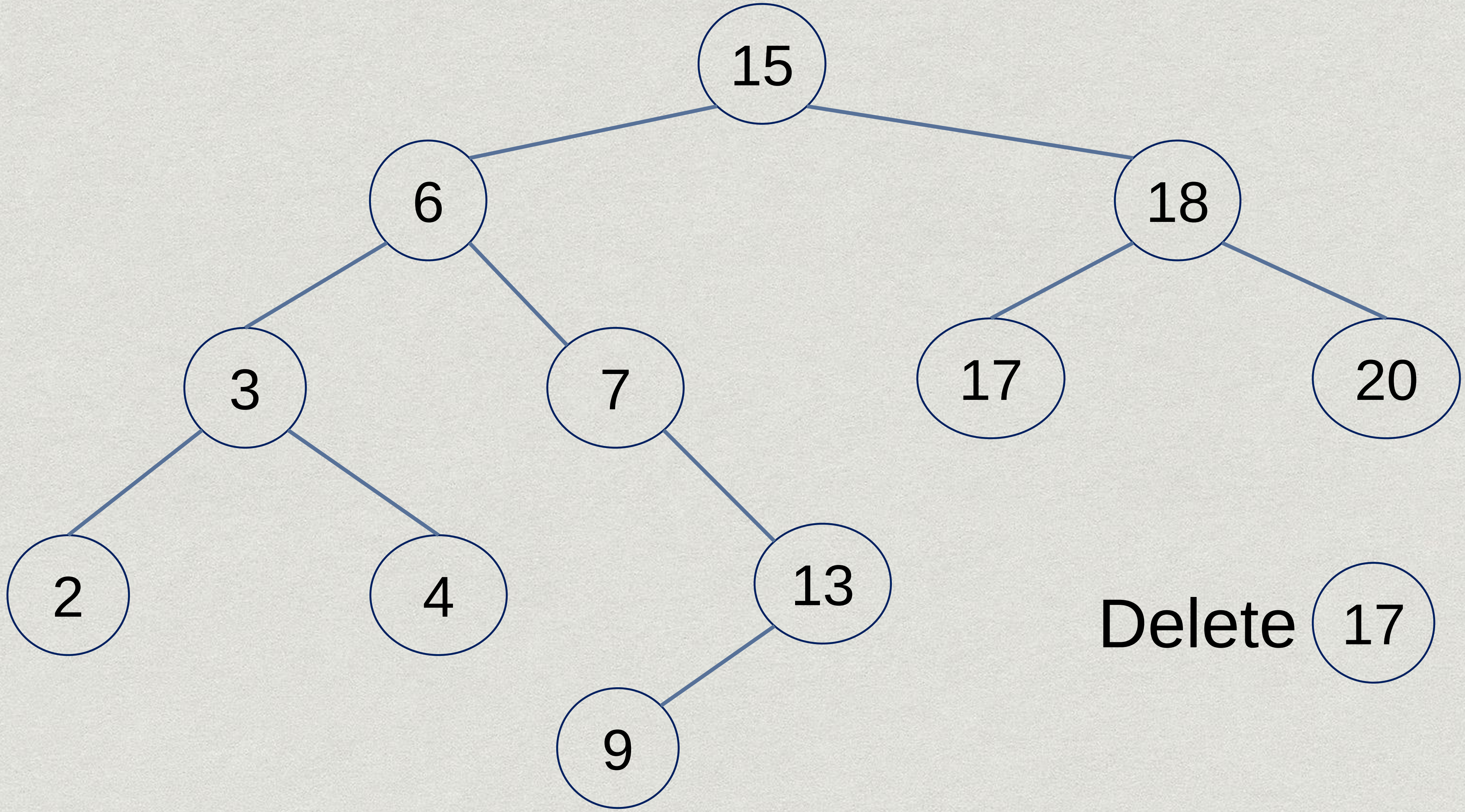


Follow the same method as searching for 14
Then insert it at the right place

Inserting a node into a tree

```
Tree_Insert(T,z) {
   y = NIL
   x = root(T)
   while x ≠ NIL:
     y = x
     if key(z) < key(x):
       x = leftChild(x)
     else
       x = rightChild(x)
     endif
   endwhile

// we have located
// the right place to insert

// we perform the
// insert now
   parent(z) = y
   if y == NIL:
      root(T) = z
   else
     if key(z) < key(y):
       leftChild(y) = z
     else
       rightChild(y) = z
     endif
   endif
}
```

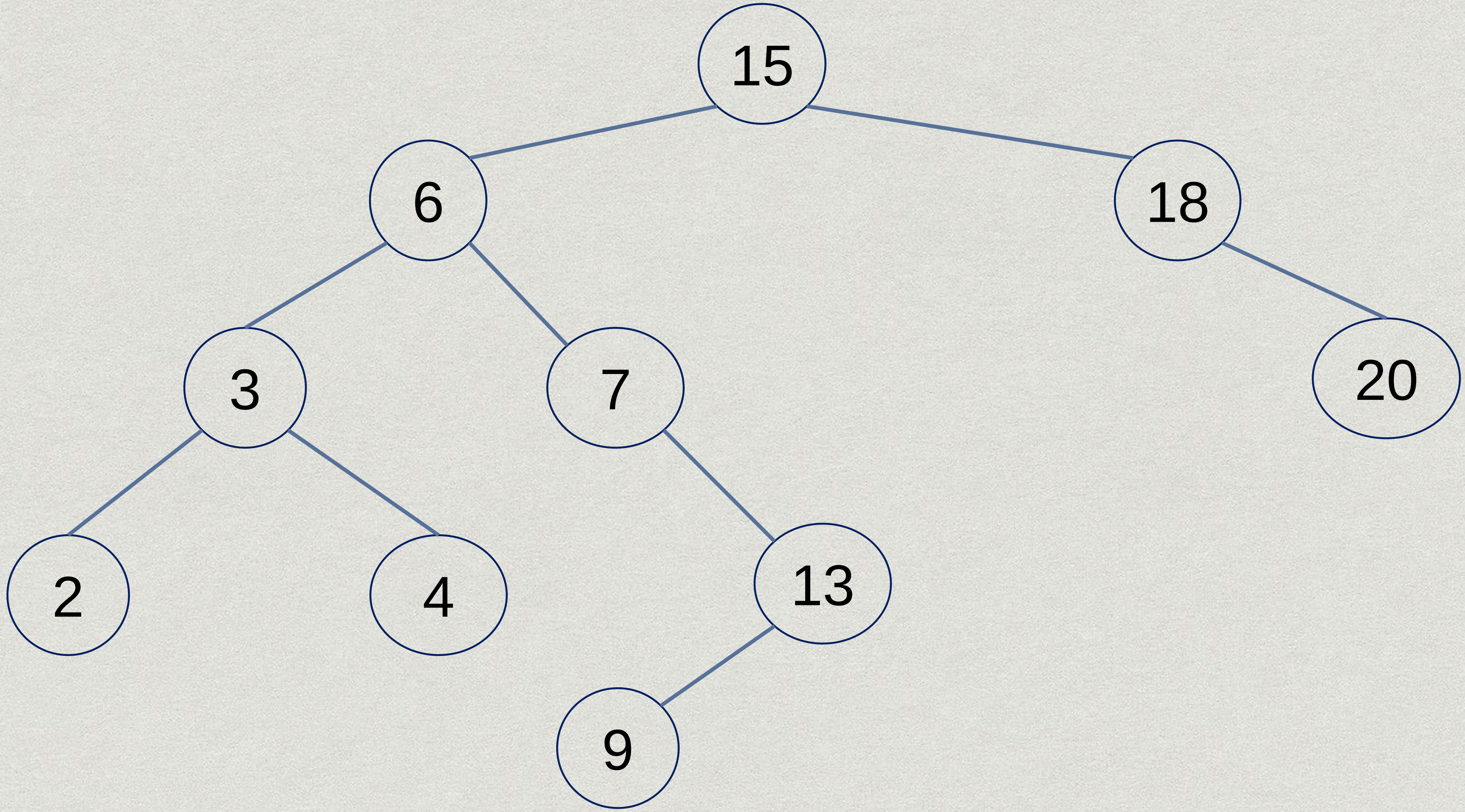
Deleting a node



Delete 17

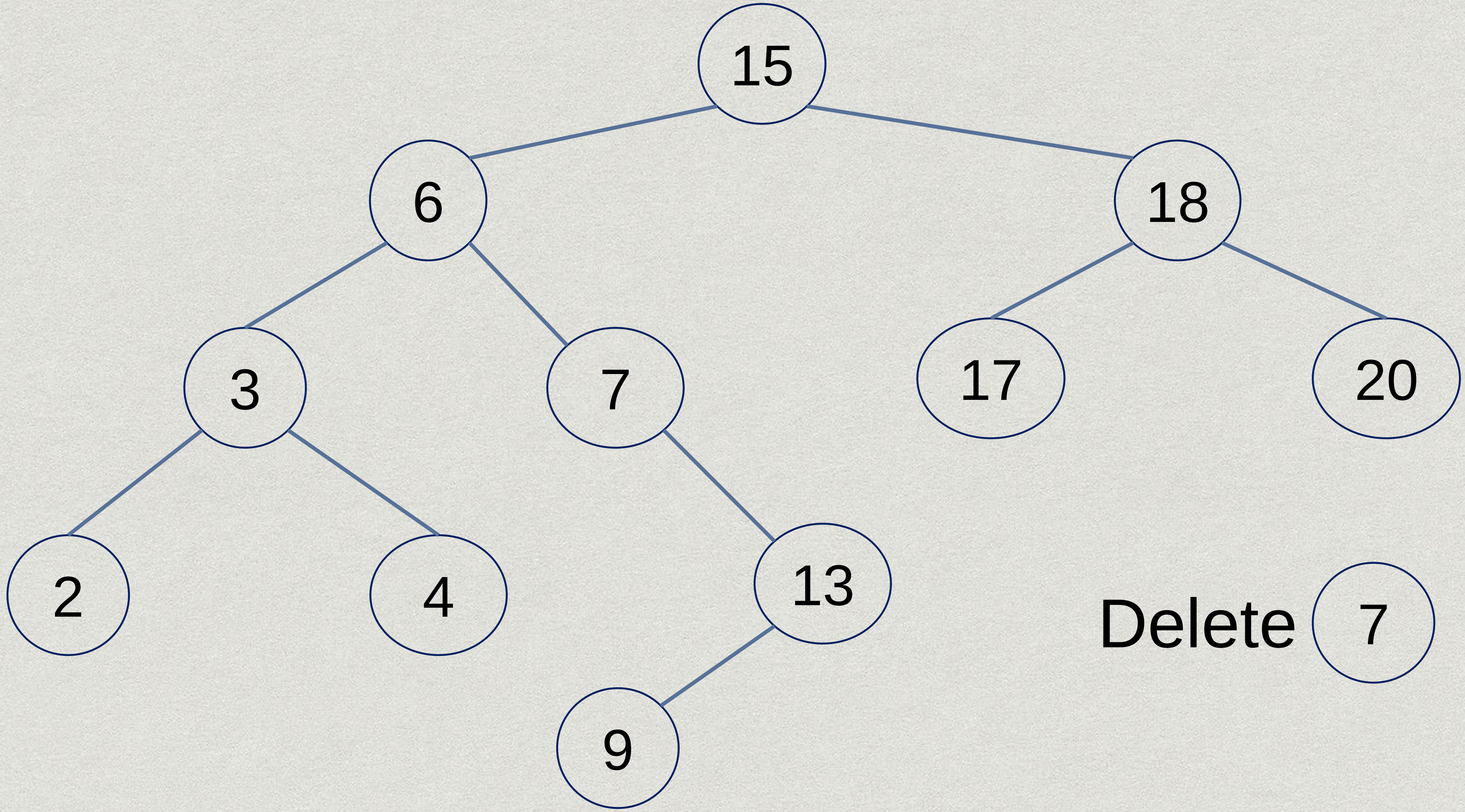
Case 1: Delete a node with no children

Deleting a node



Case 1: Delete a node with no children
Easy – Just delete the node !

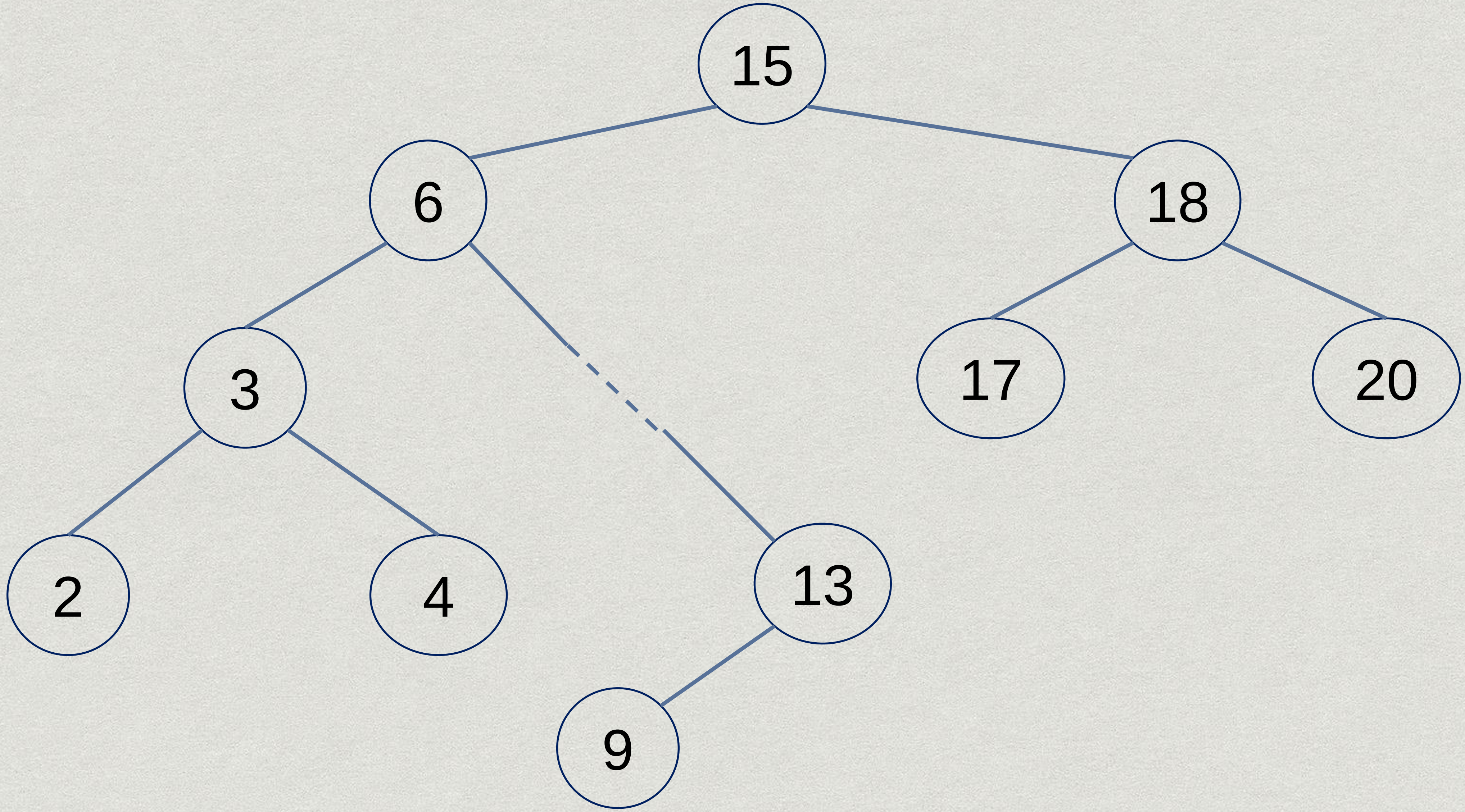
Deleting a node



Delete 7

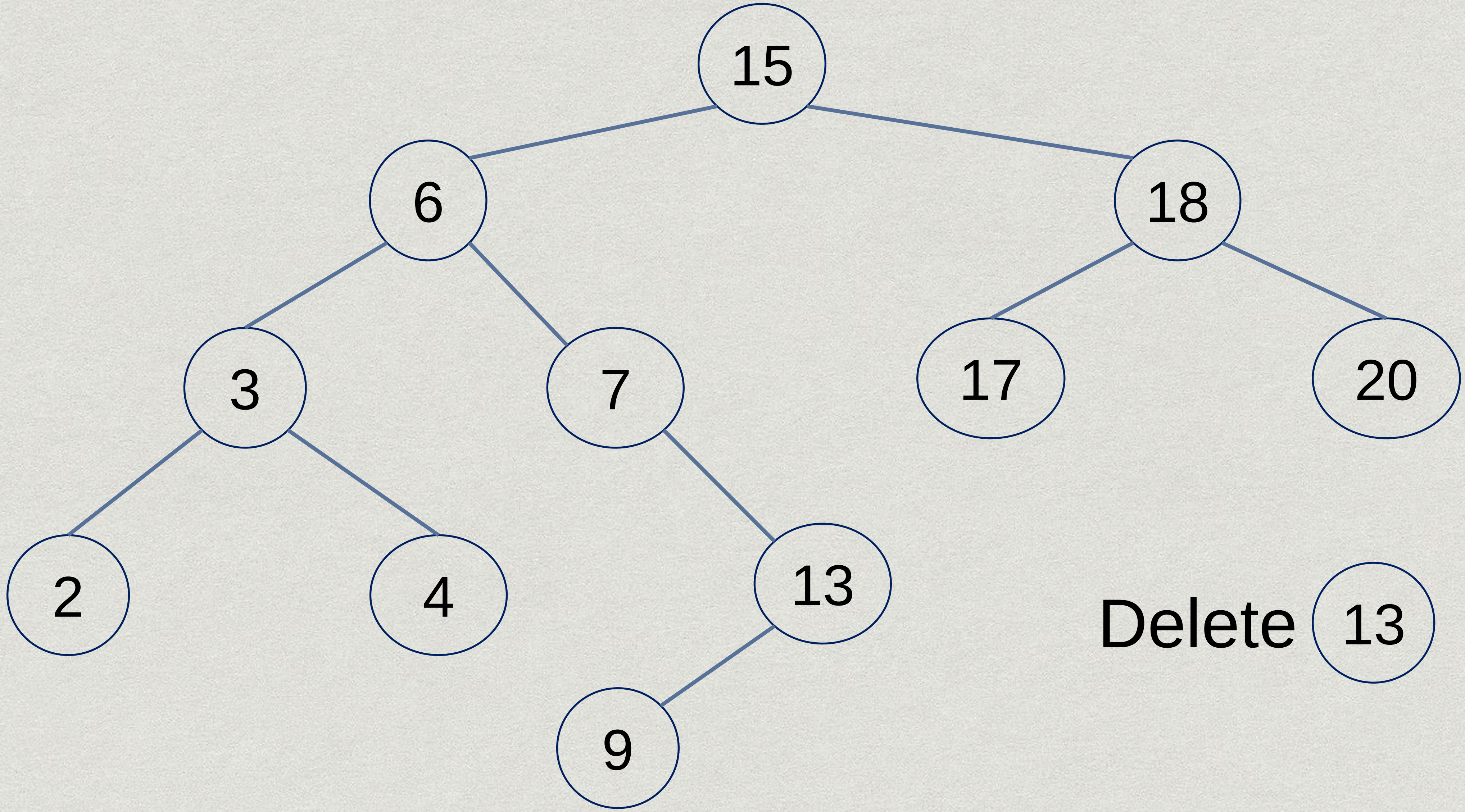
Case 2: Delete a node with one child

Deleting a node



Case 2: Delete a node with one child
Splice out the node

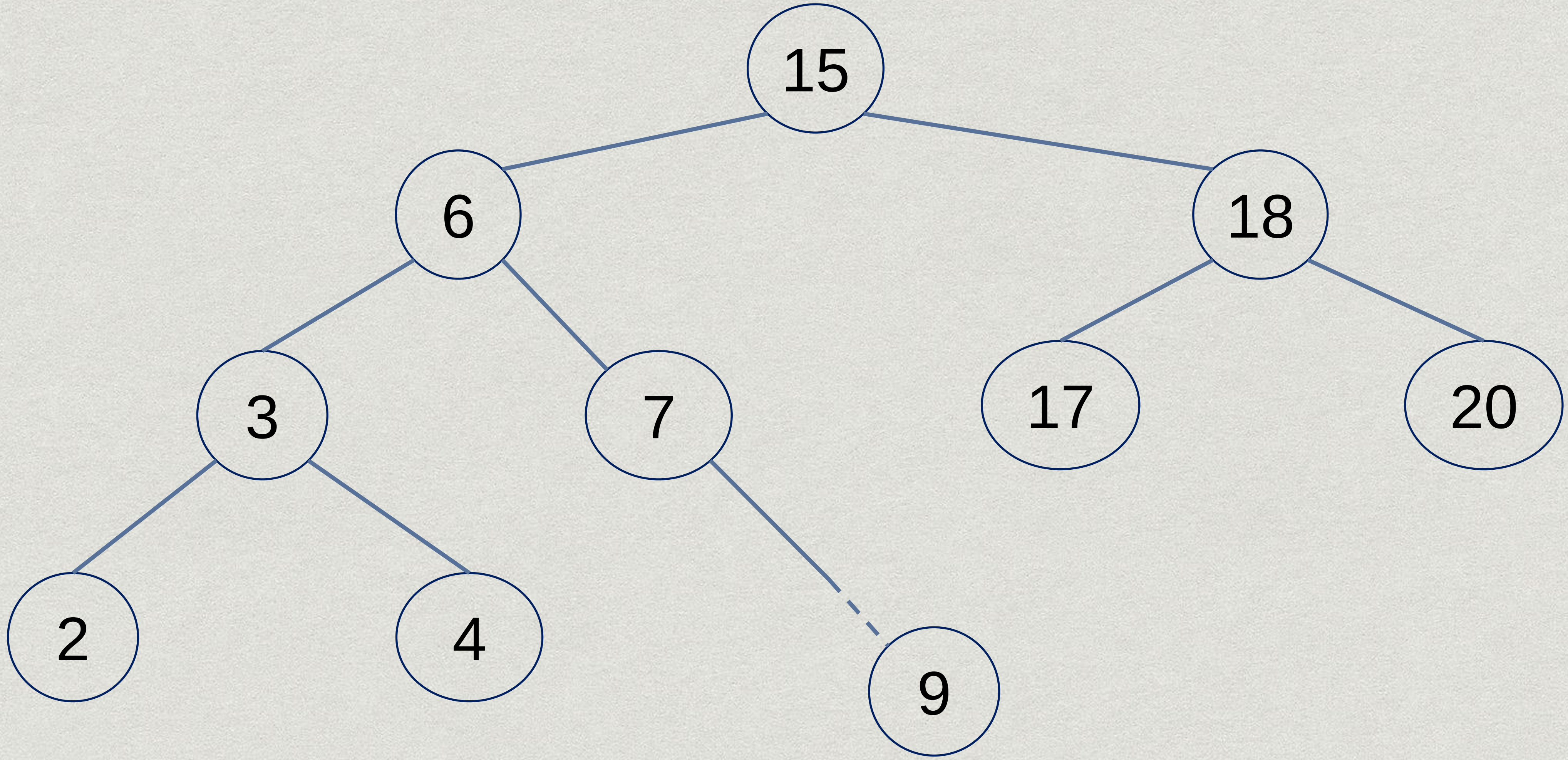
Deleting a node



Delete 13

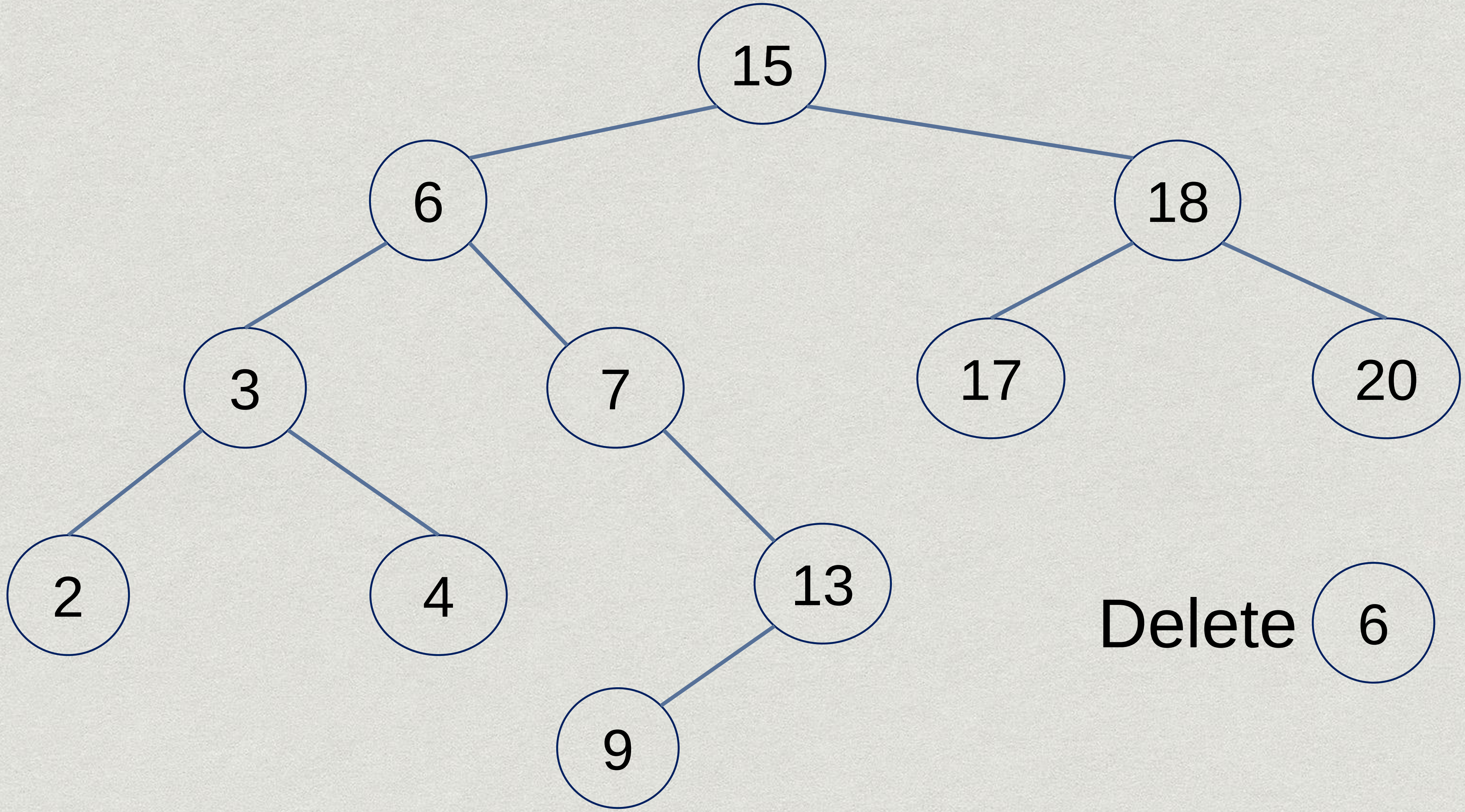
Case 2: Delete a node with one child

Deleting a node



Case 2: Delete a node with one child
Splice out the node

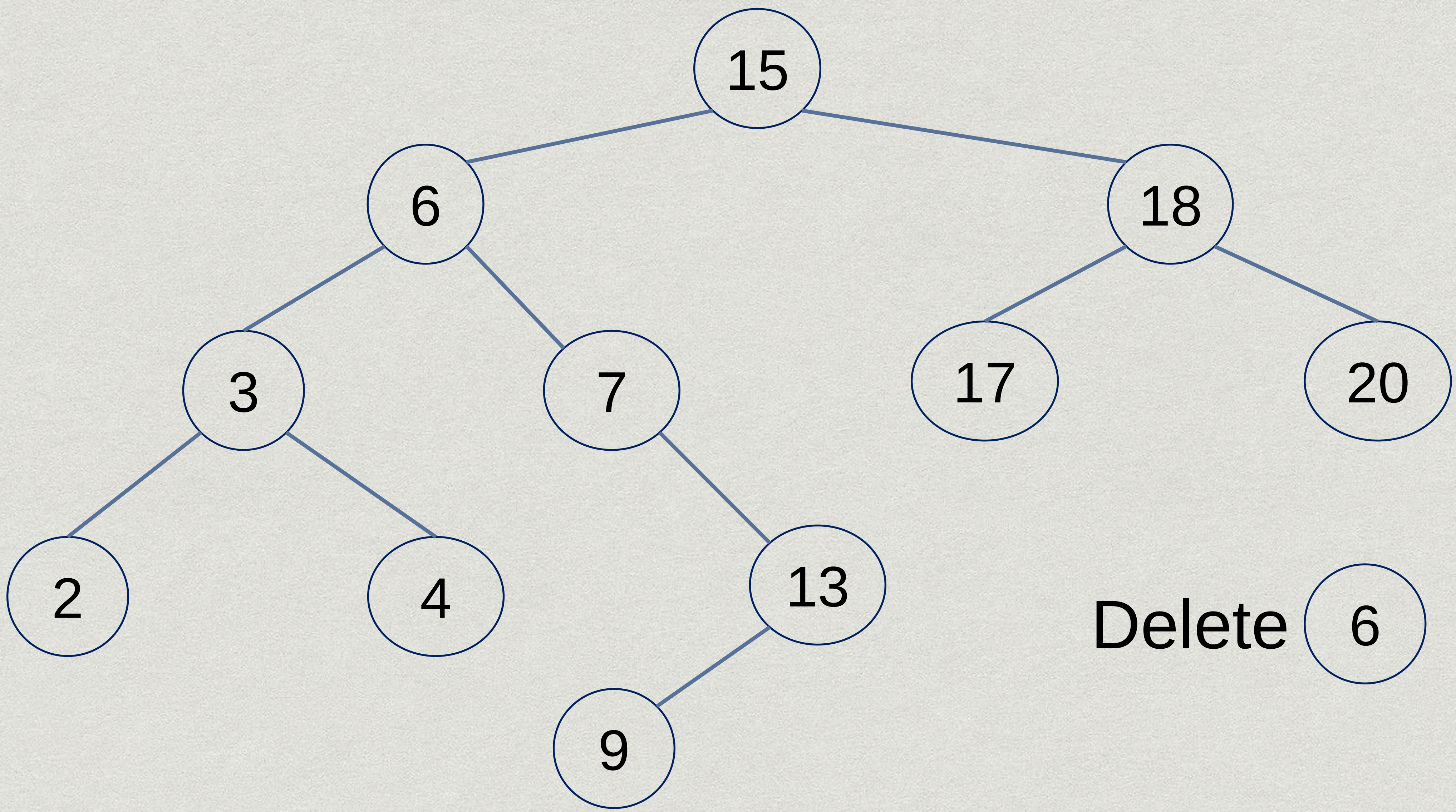
Deleting a node



Delete 6

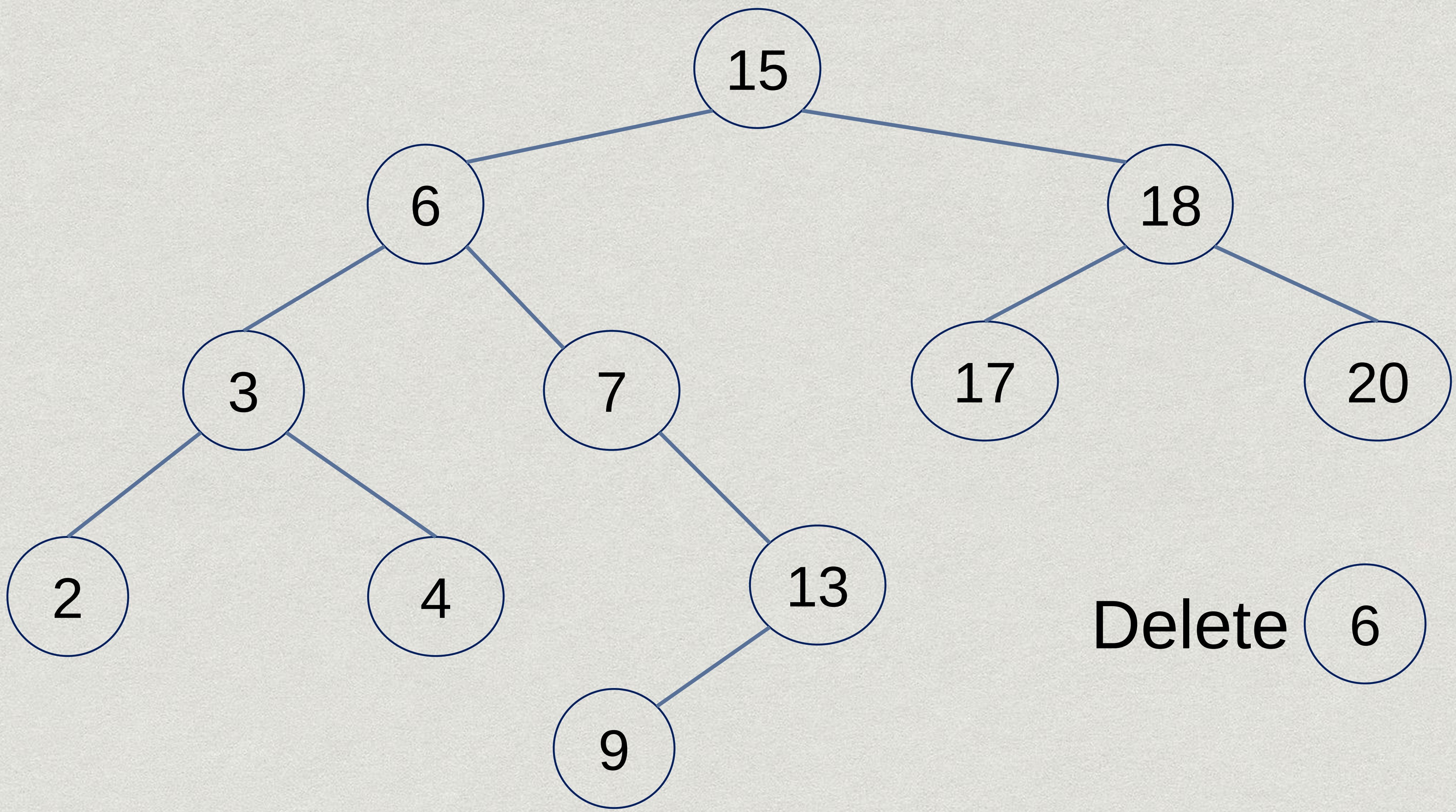
Case 3: Delete a node with two children

Deleting a node



Case 3: Delete a node with two children
First find the successor of 6 … min of the right sub tree of 6

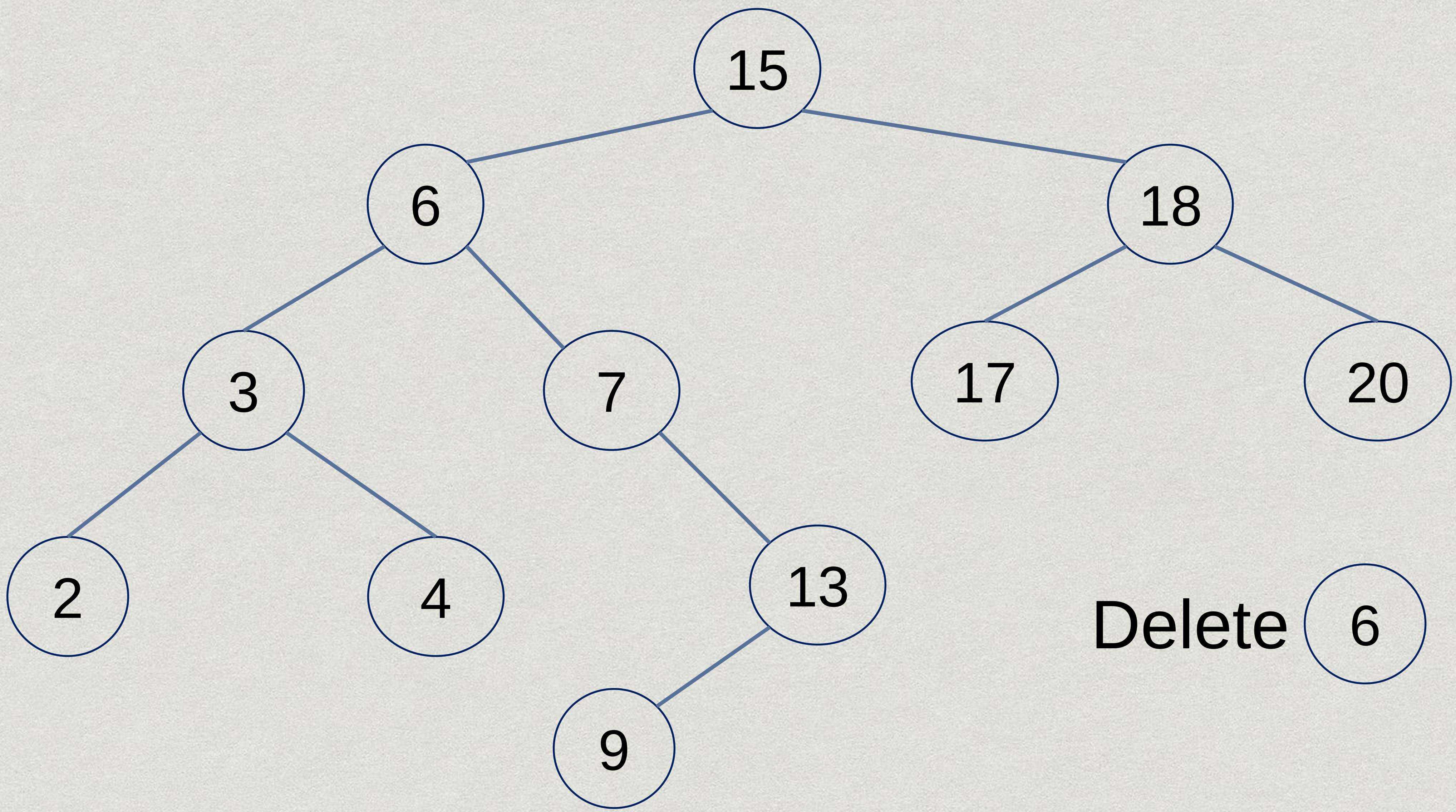
Deleting a node



Case 3: Delete a node with two children

First find the successor of 6 … succ is 7

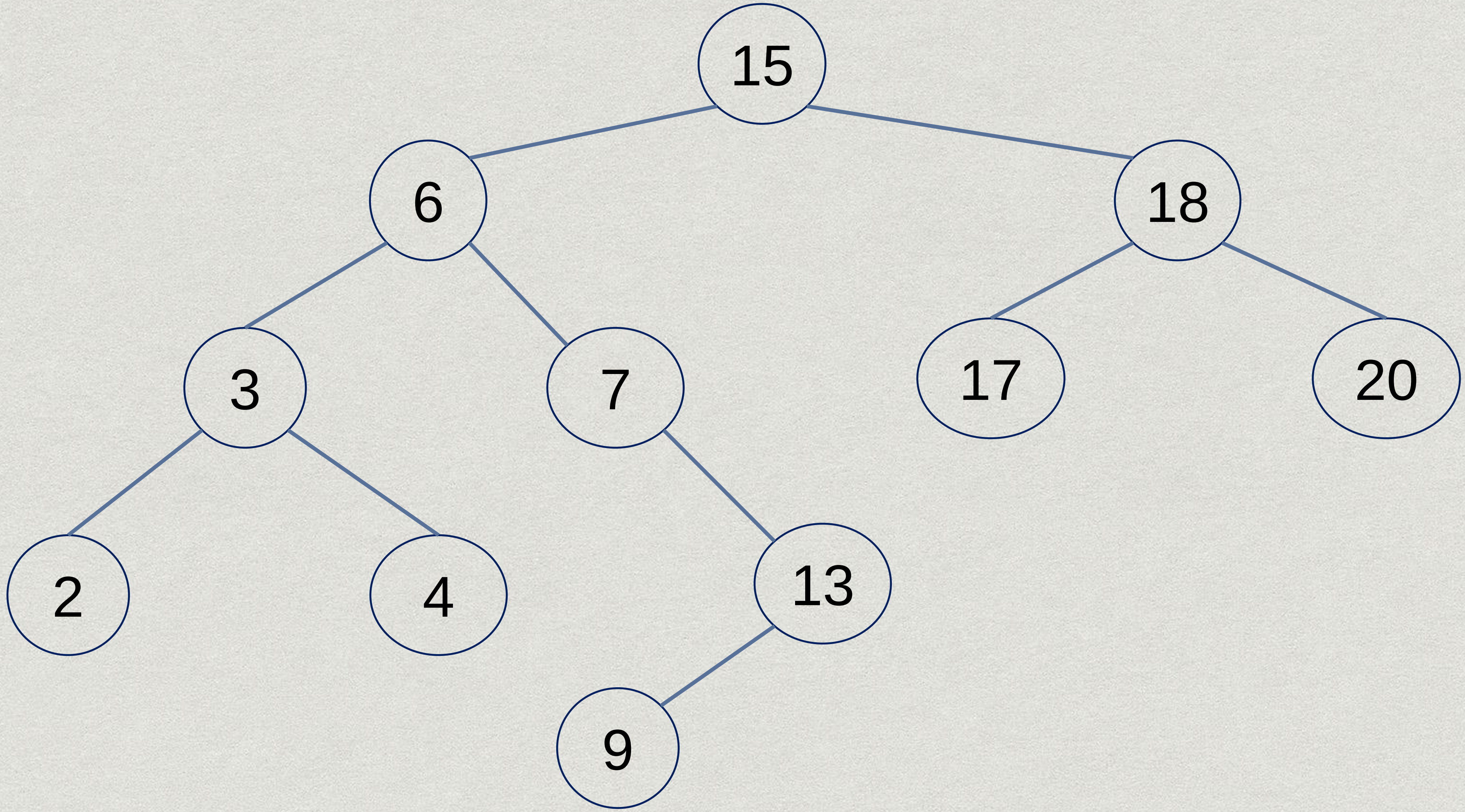
Deleting a node



Delete 6

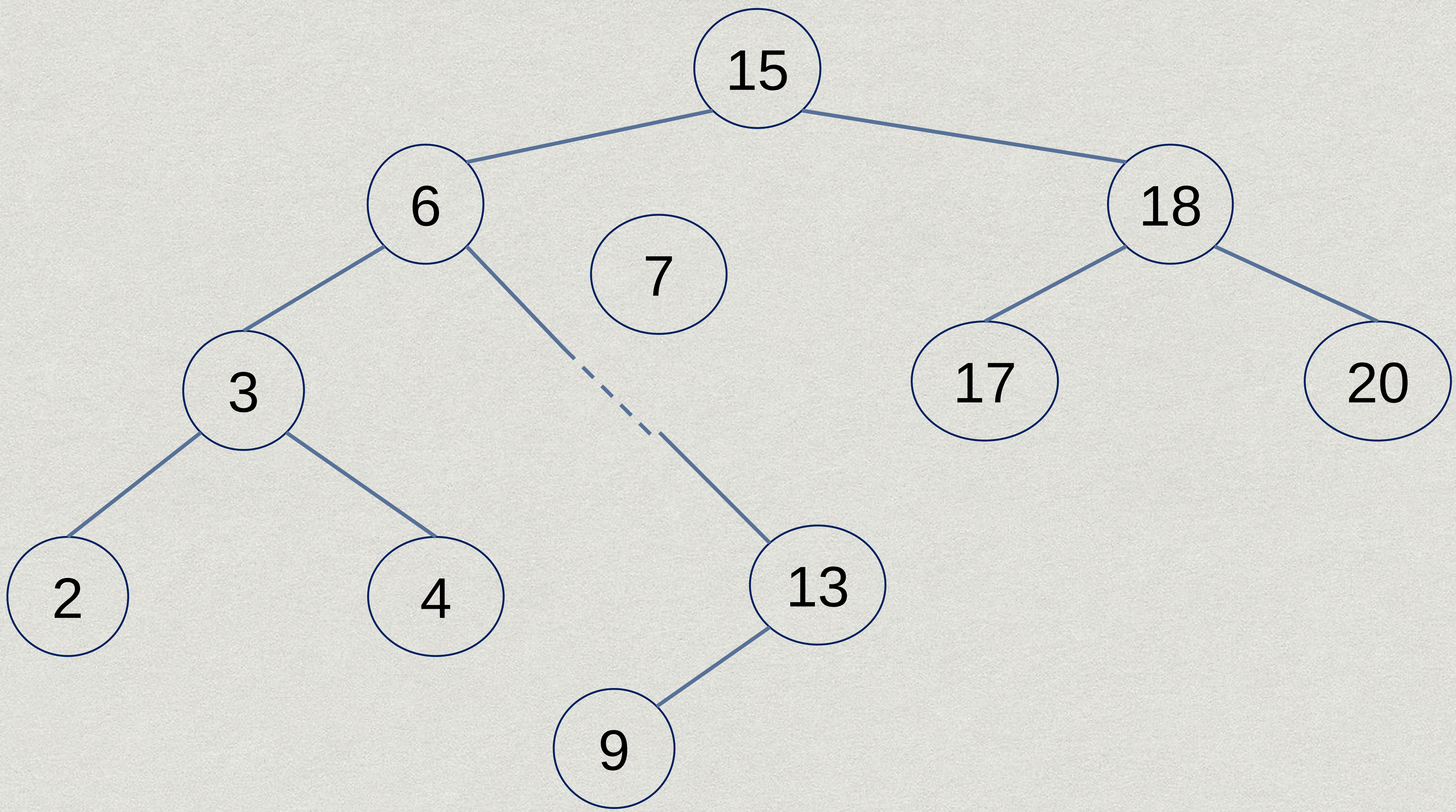
Case 3: Delete a node with two children
First find the successor of 6 … succ is 7
Note: succ will have only right child … since min is found by going down the left of the tree till there is no leftChild

Deleting a node



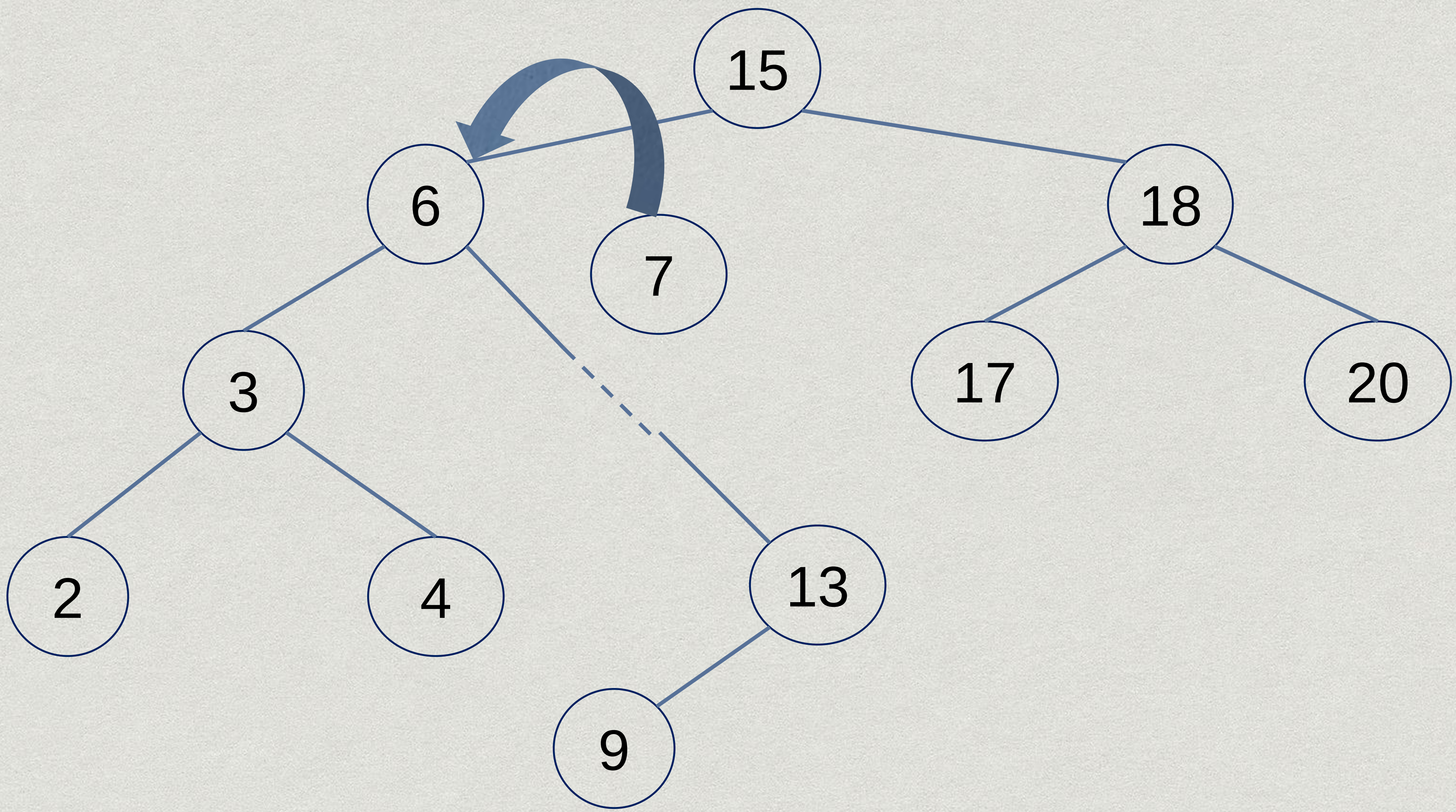
Case 3: Delete a node with two children
Find succ … node 7
Now splice out node 7

Deleting a node



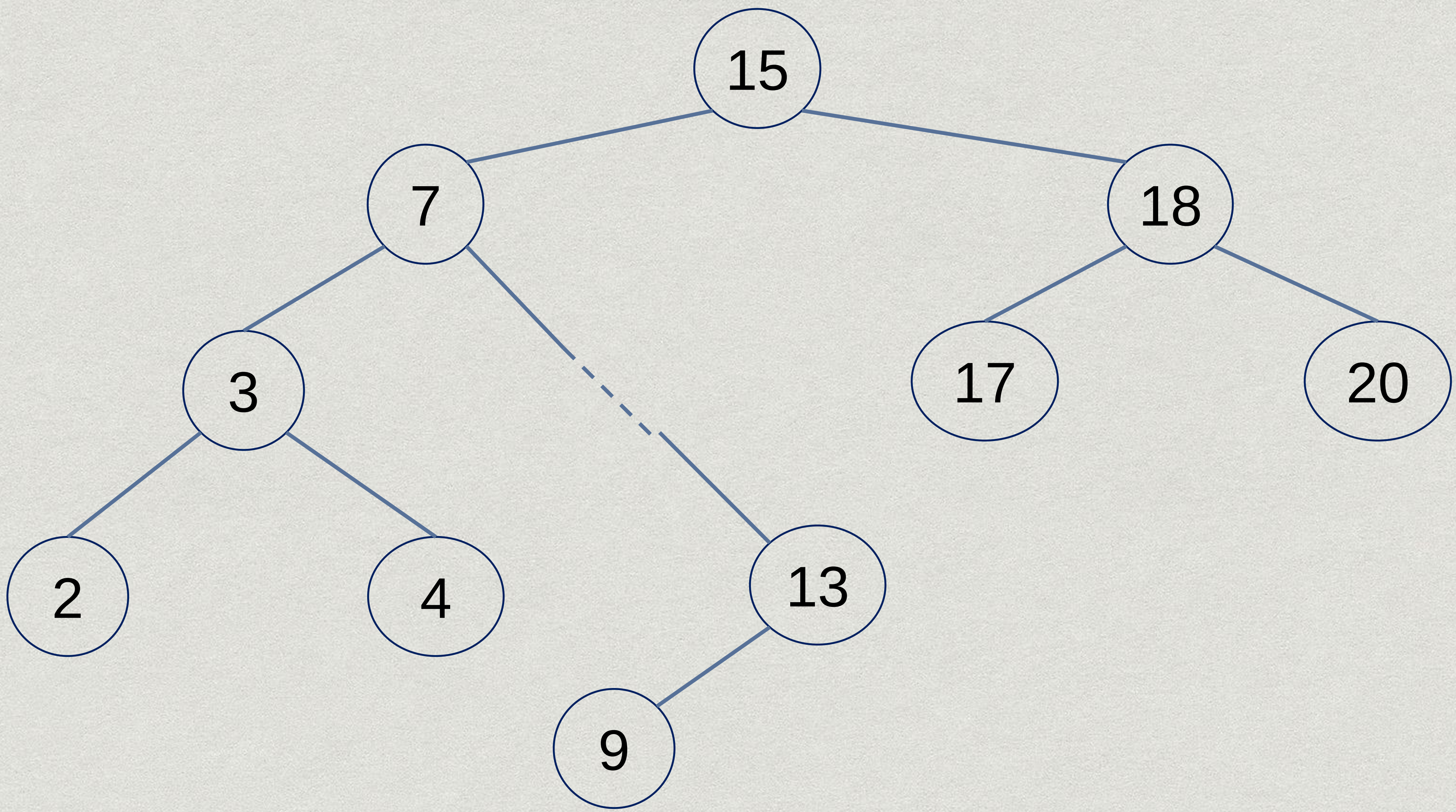
Case 3: Delete a node with two children
Find succ … node 7
Now splice out node 7

Deleting a node



Case 3: Delete a node with two children
Find succ … node 7
Now splice out node 7
… and replace contents of node 6 with that of node 7

Deleting a node



Case 3: Delete a node with two children
Find succ … node 7
Now splice out node 7
… and replace contents of node 6 with that of node 7

Deleting a node from a tree

```
Tree_Delete(T,z) {
    if leftChild(z) == NIL or rightChild(z) == NIL:
        y = z
    else
        y = Tree_Succ(z)
    endif
 // node to splice
//  is z or succ(z)

   if leftChild(y) ≠ NIL:
        x = leftChild(y)
    else
        x = rightChild(y)
    endif

    if x ≠ NIL:
        parent(x) = parent(y)
    endif
    if parent(y) == NIL:
        root(T) = x
    else
     if  y == leftChild(parent(y)):
          leftChild(parent(y)) = x
     else
          rightChild(parent(y)) = x
     endif
     if y ≠ z … copy y to z
}
```

Complexity of algorithms so far

- All operations we saw so far on binary search trees (Search, Min, Max, Succ, Pred, Insert, Delete) can be done in time proportional to the height of the tree (length of longest path from root to a leaf)

Complexity of algorithms so far

- All operations we saw so far on binary search trees (Search, Min, Max, Succ, Pred, Insert, Delete) can be done in time proportional to the height of the tree (length of longest path from root to a leaf)

- Height of a binary search tree can be properly managed and kept within $\log n$ - use rotations to keep the tree balanced - AVL trees or red-black trees

Complexity of algorithms so far

- All operations we saw so far on binary search trees (Search, Min, Max, Succ, Pred, Insert, Delete) can be done in time proportional to the height of the tree (length of longest path from root to a leaf)

- Height of a binary search tree can be properly managed and kept within $\log n$ - use rotations to keep the tree balanced - AVL trees or red-black trees

- A randomly generated binary search tree will have an expected height of $\log n$

Summary

- Heaps are not good for determining succ and pred
- A new data structure is required when we have to do search along with insert/delete repeatedly

- Binary search trees are binary trees with elements stored in binary search tree order – i.e. keys of a node are no less than keys in its left subtree and no greater than keys in its right subtree

- All operations on binary search trees can be done in time proportional to the height of the tree

- Height of a binary search tree can be properly managed and kept within $\log n$