BINARY SEARCH TREES

BST

Sorted arrays	Linked lists
Inserting a new item is quite slow // O(N)	Inserting a new item is very fast // O(1)
Searching is quite fast with binary search // O(logN)	Searching is sequental // O(N)
Removing an item is slow // O(N)	Removing an item is fast because of the references // O(1)

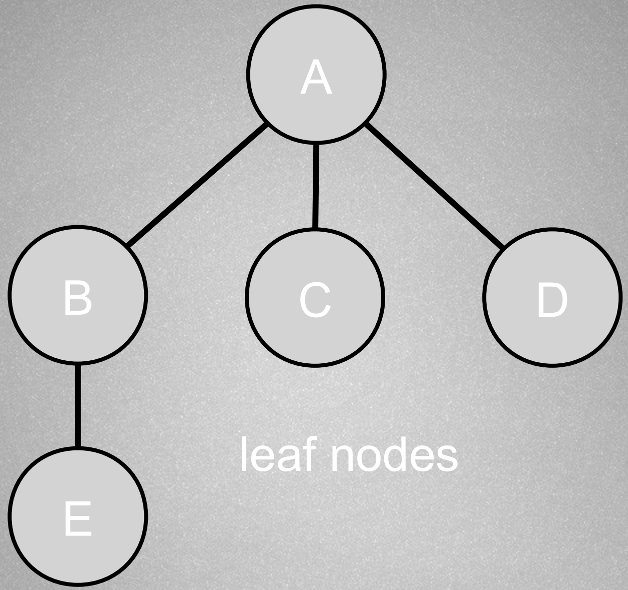
Sorted arrays	Linked lists
Inserting a new item is quite slow // O(N)	Inserting a new item is very fast // O(1)
Searching is quite fast with binary search // O(logN)	Searching is sequental // O(N)
Removing an item is slow // O(N)	Removing an item is fast because of the references // O(1)

Binary search trees are going to make all of these operations quite fast, with **O(log N)** time complexity !!! ~ predictable

Trees

root node: we have a reference to this, all
other nodes can be accessed via the root node

We have nodes with
the data and connection
between the nodes
// edges



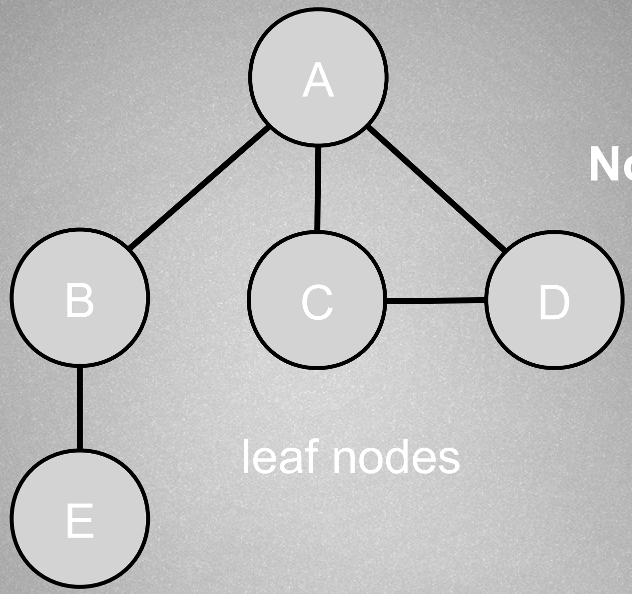
In a tree: there must be only a single path from the root node to any other nodes in the tree

~ if there are several ways to get to a given node: it is not a tree !!!

Trees

root node: we have a reference to this, all
other nodes can be accessed via the root node

We have nodes with
the data and connection
between the nodes
// edges



Not a tree !!!

leaf nodes

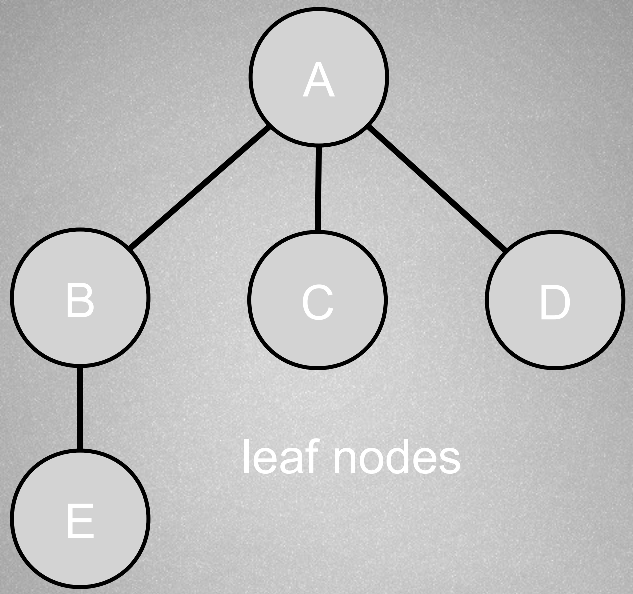
In a tree: there must be only a single path from the root node to any other nodes in the tree

~ if there are several ways to get to a given node: it is not a tree !!!

Trees

root node: we have a reference to this, all other nodes can be accessed via the root node

We have nodes with the data and connection between the nodes // edges

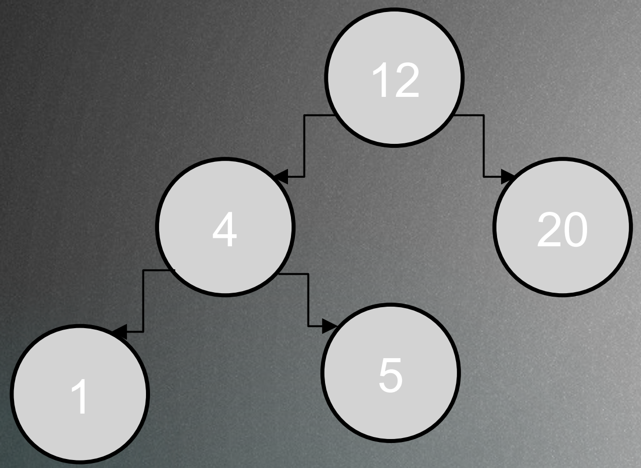


A node directly connected to another node → child

The opposite → parent node

Leaf nodes: with no children

Binary search trees

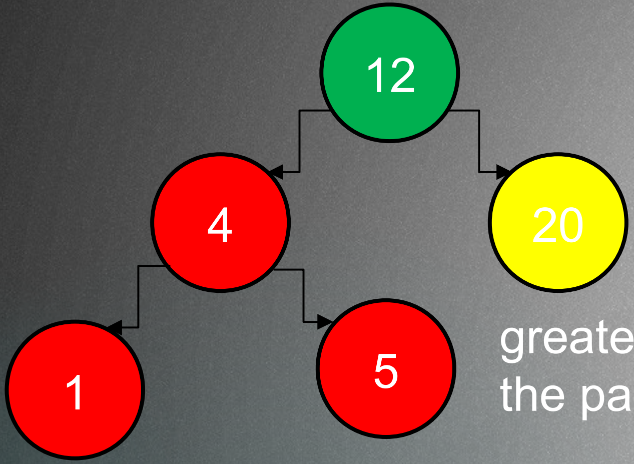


- every node can have at most two children: left and right child

- left child: smaller than the parent

- right child: greater than the parent

Binary search trees



greater than
the parent 12

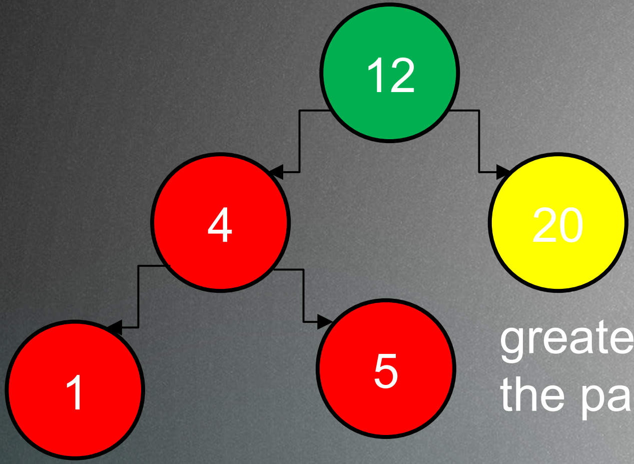
smaller than the
parent 12

- every node can have at most
  two children: left and right child

- left child: smaller than the parent

- right child: greater than the parent

Binary search trees



greater than
the parent 12

smaller than the
parent 12

- every node can have at most
  two children: left and right child

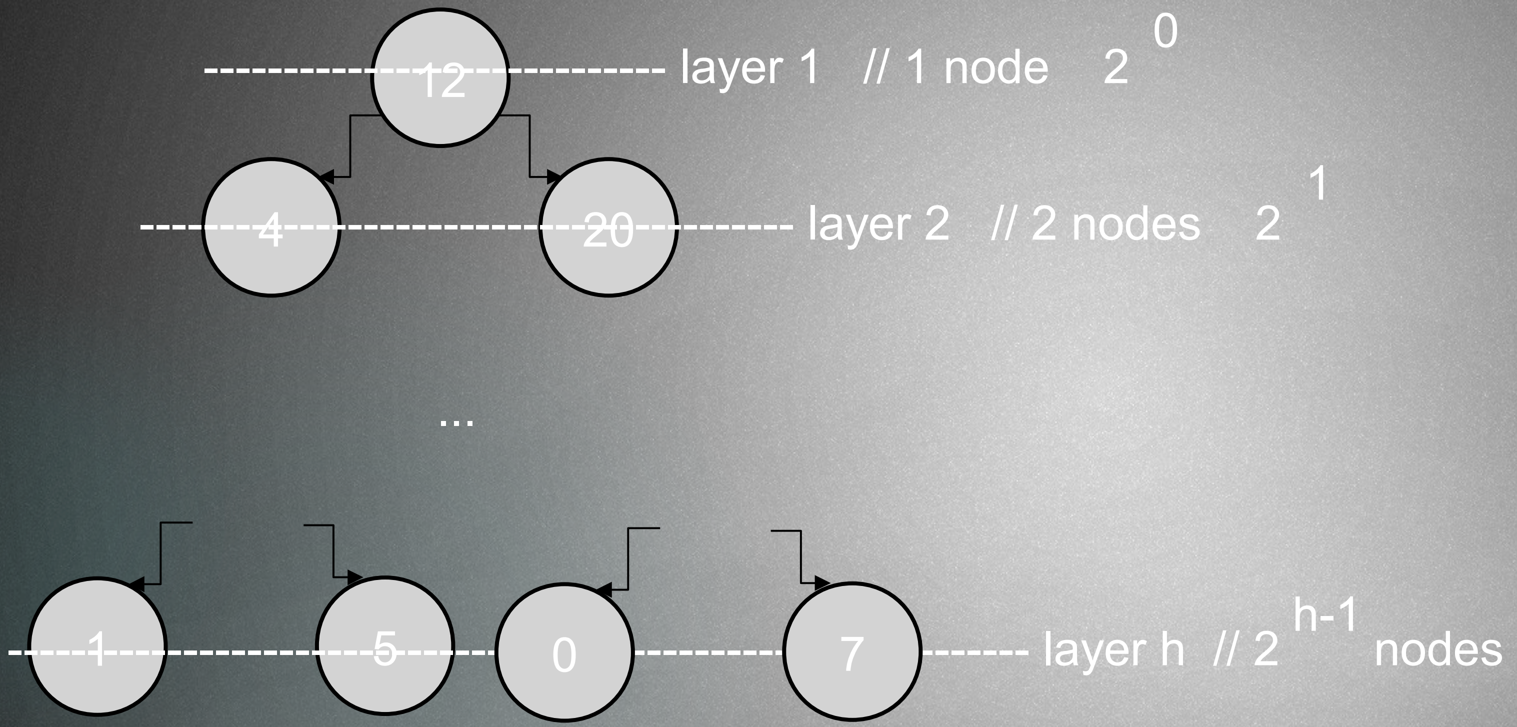
- left child: smaller than the parent

- right child: greater than the parent

Why is it good? On every decision we get rid of half of the data in which
we are searching !!! // like binary search
~ **O(logN)** time complexity

Binary search trees

Height of a tree: the number of layers it contains



In general: h ~ **O(logN)** if this is true the tree is said to be balanced
If it is not true the tree is unbalanced, which means it is asymmetric which is a PROBLEM !!!

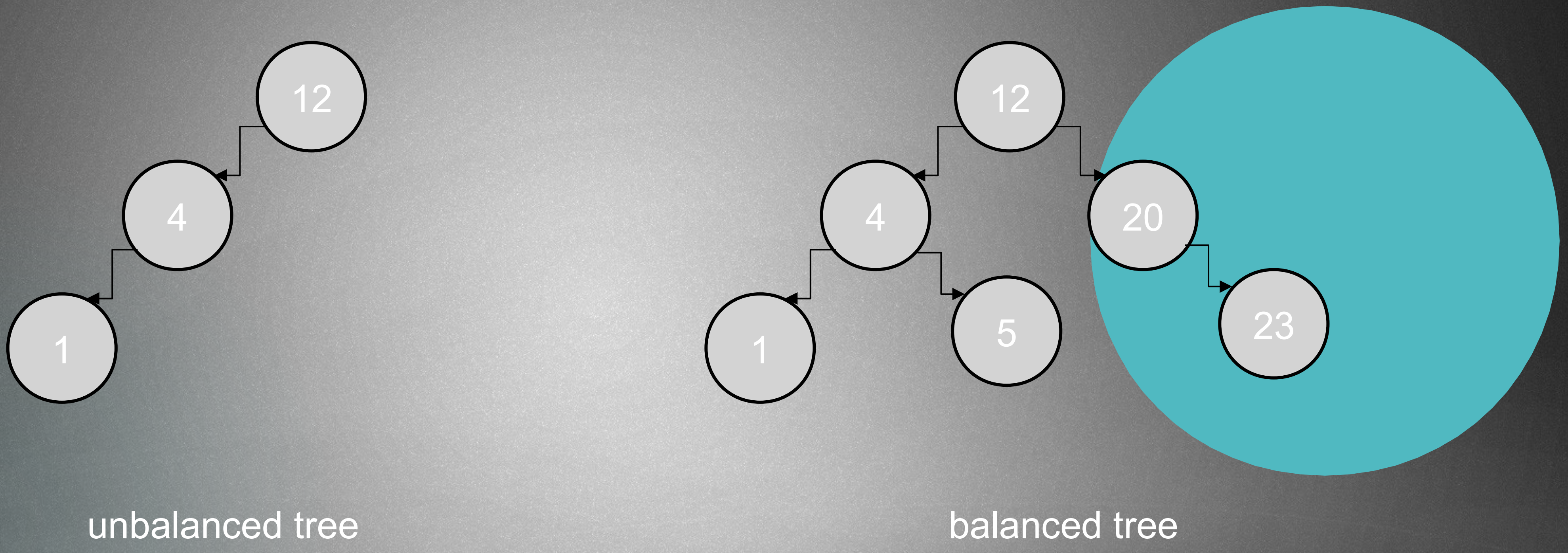
Trees

height of a tree: the number of layers it has

Height of the tree ‚h': the length of the path from the root to the deepest node in the tree

- we should keep the height of the tree at a minimum which is h=log n

- if the tree is unbalanced: the h=log n relation is no more valid and the operation's running time is no more logarithmic


12
4
1
unbalanced tree
12
4
20
1
5
23
balanced tree

Binary search trees

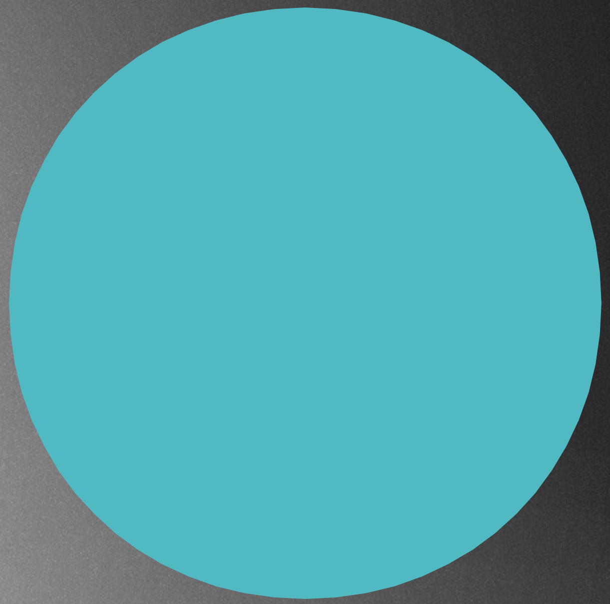
- Binary search trees are data structures
- Keeps the keys in sorted order: so that lookup and other operations can use the principle of binary search !!!
- Each comparison allows the operations to skip over half of the tree, so that each lookup/insertion/deletion takes time proportional to the logarithm of the number of items stored in the tree
- This is much better than the linear time O(N) required to find items by key in an unsorted array, but slower than the corresponding operations on hash tables

Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left

And so on ...

Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(12);



Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(12);

Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

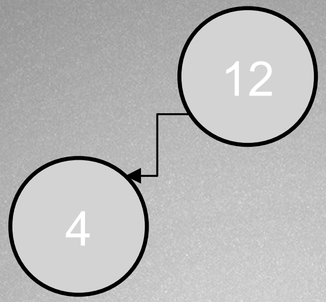
binarySearhTree.insert(4);

Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left

And so on ...

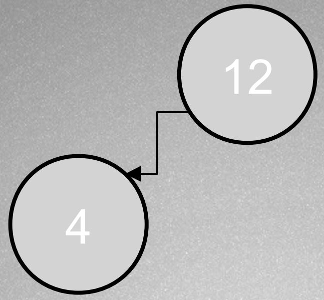
binarySearhTree.insert(4);

Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...



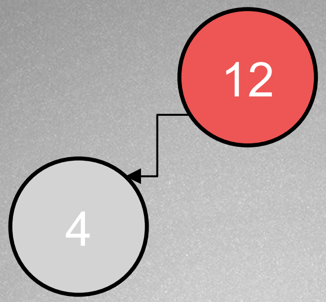
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(5);



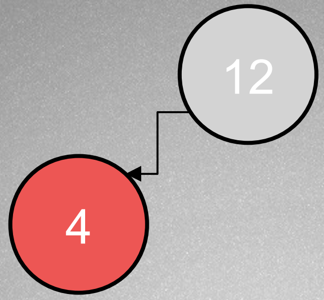
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(5);



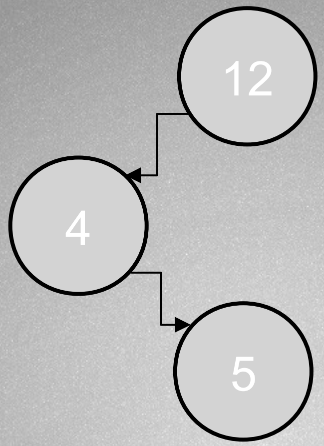
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(5);



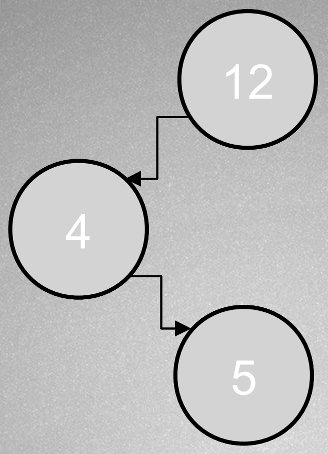
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left

And so on ...



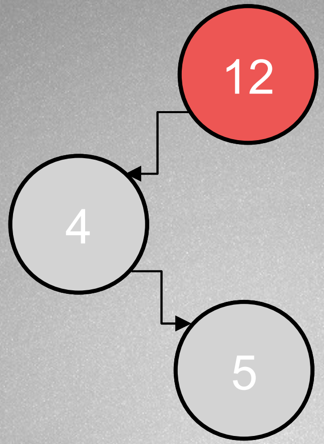
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(20);

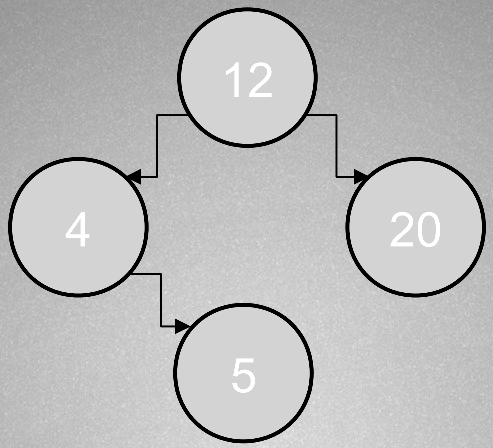


Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(20);

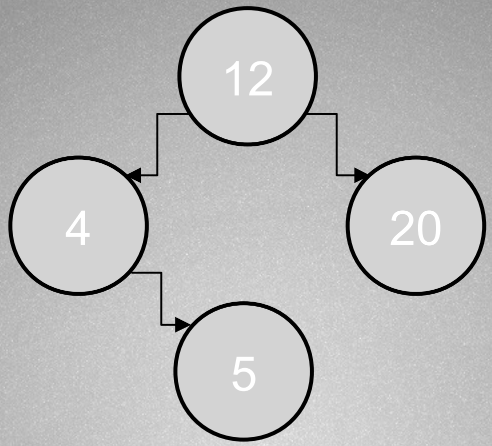


Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...



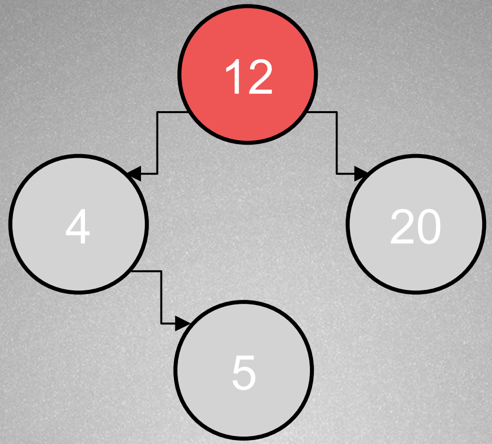
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(1);



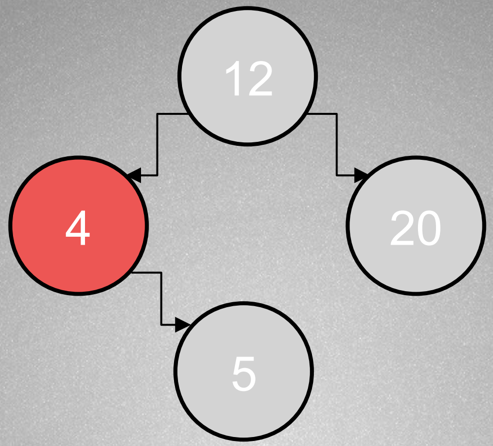
Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(1);

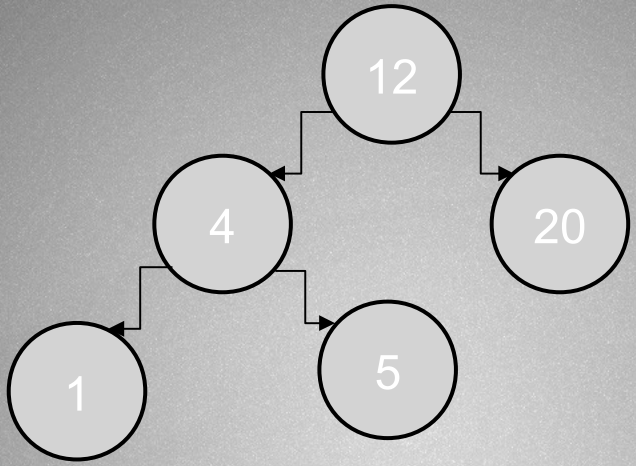


Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...

binarySearhTree.insert(1);

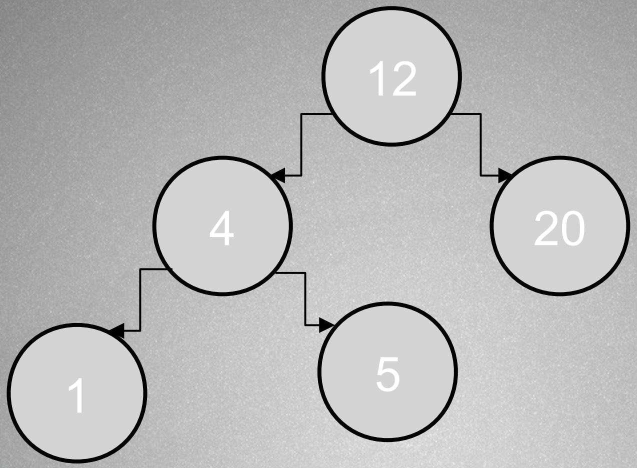


Insertion: we start at the root node. If the data we want to insert is greater than the root node we go to the right, if it is smaller, we go to the left
And so on ...



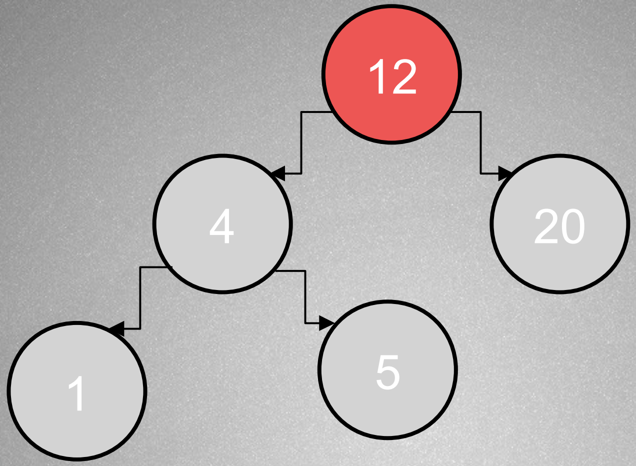
Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);



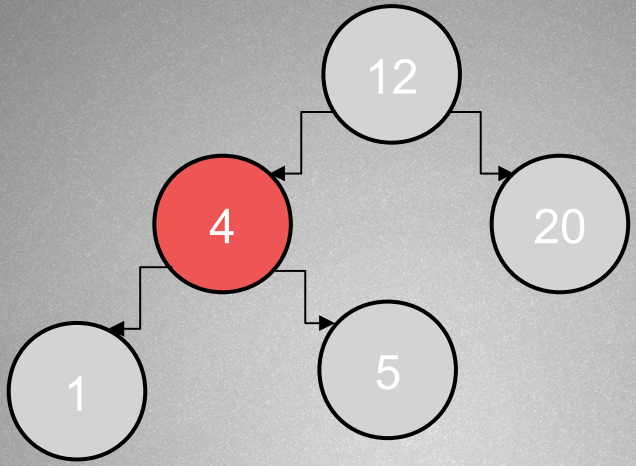
Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);



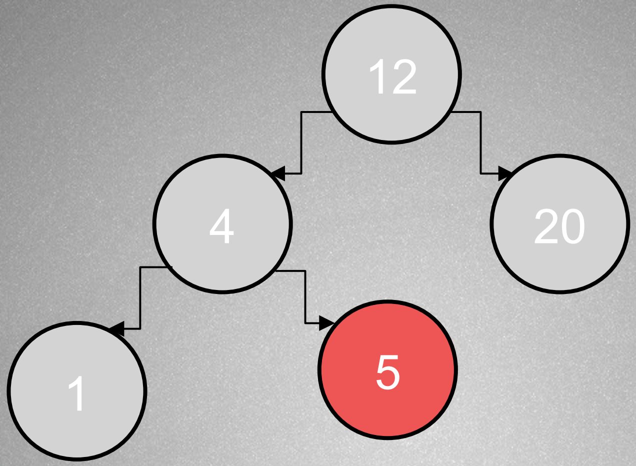
Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);



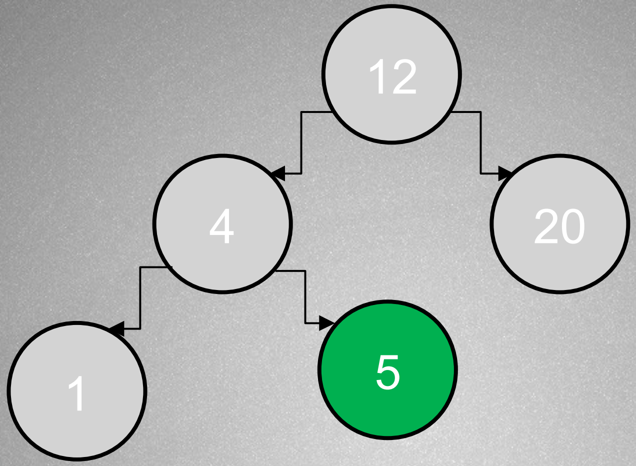
Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);



Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);

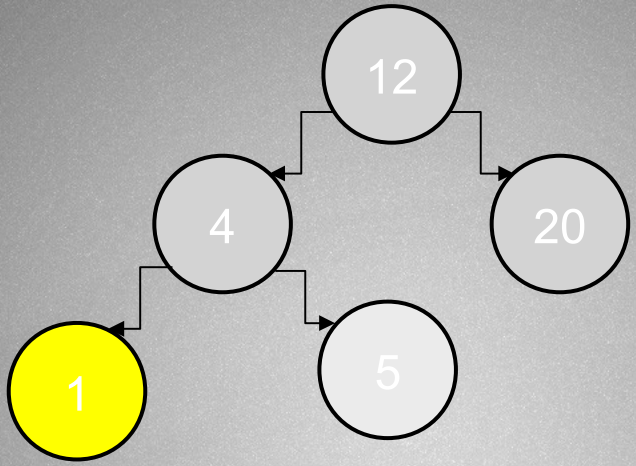


We have mangaed to find the item

On every decision: we discard half of the tree, so it is like binary search in a sorted array // O(logN)

Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

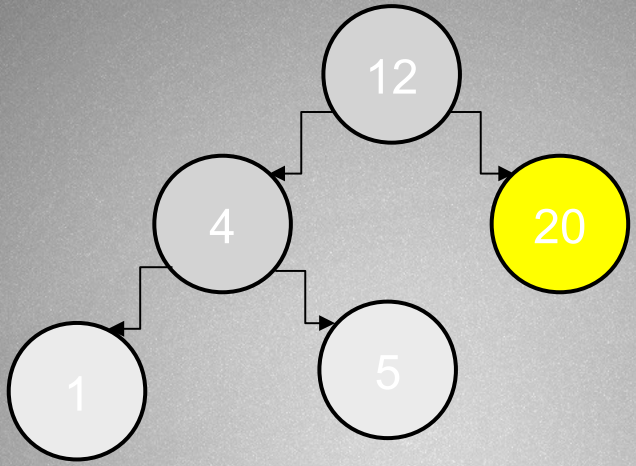
binarySearhTree.find(5);



We want to find the smallest node: we just have to go to the left as far as possible ... it will be the smallest !!!

Search: we start at the root node. If the data we want to find is greater than the root node we go to the right, if it is smaller, we go to the left until we find it !!!

binarySearhTree.find(5);



We want to find the smallest node: we just have to go to the left as far as possible ... it will be the smallest !!!

We want to find the largest node: we just have to go to the right as far as possible ... it will be the largest !!!

Delete: soft delete → we do not remove the node from the BST we just mark that it has been removed

~ not so efficient solution

Delete: soft delete → we do not remove the node from the BST we just mark that it has been removed

~ not so efficient solution

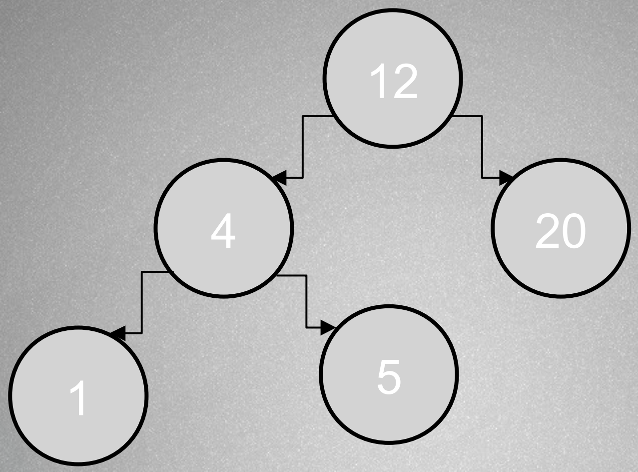
In the main **three** possible cases:

1.) The node we want to get rid of is a leaf node

2.) The node we want to get rid of has a single child

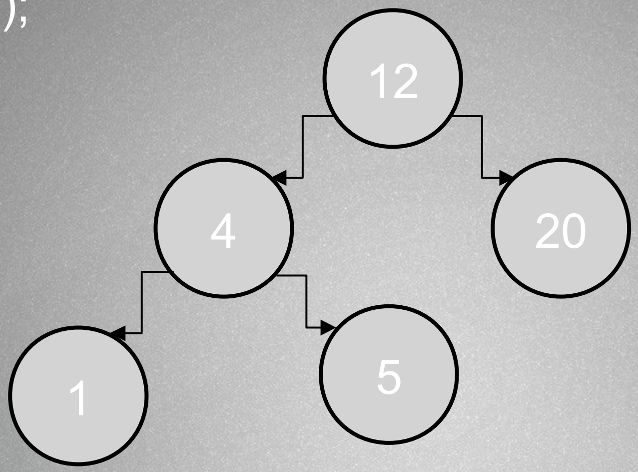
3.) The node we want to get rid of has 2 children

Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)



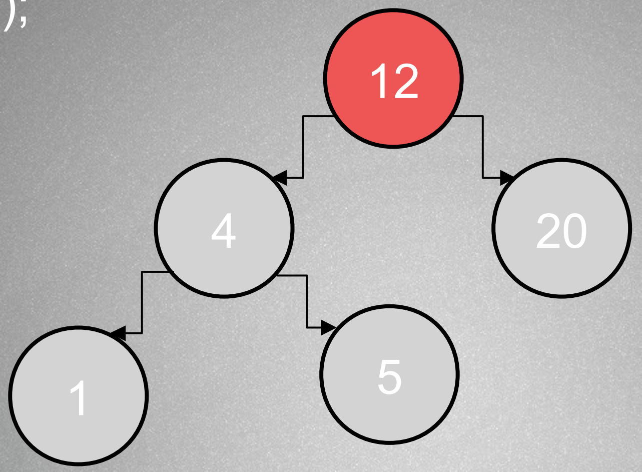
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



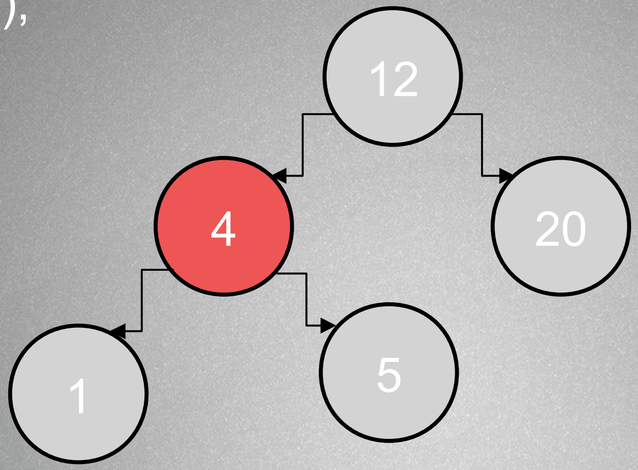
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



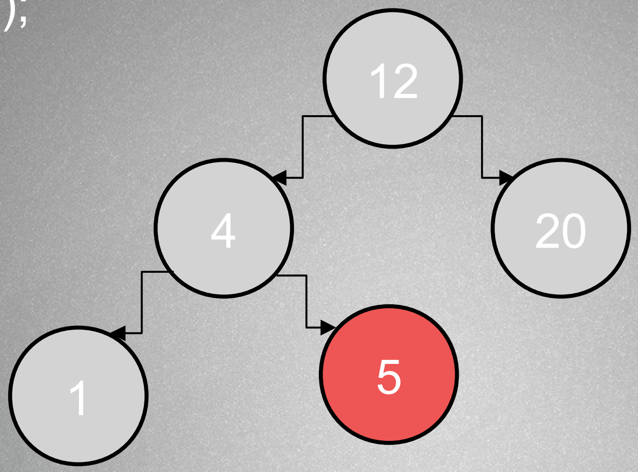
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



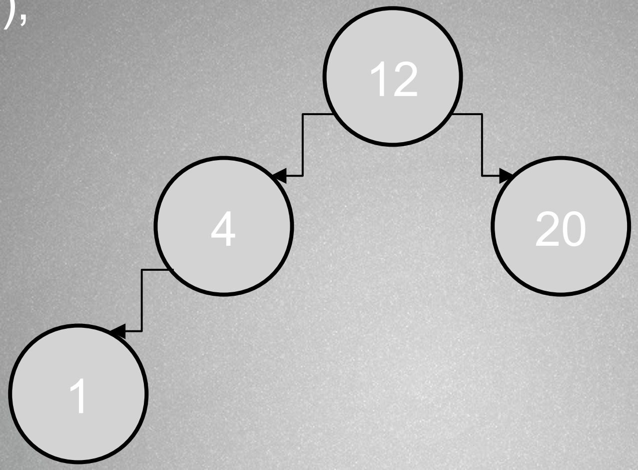
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);

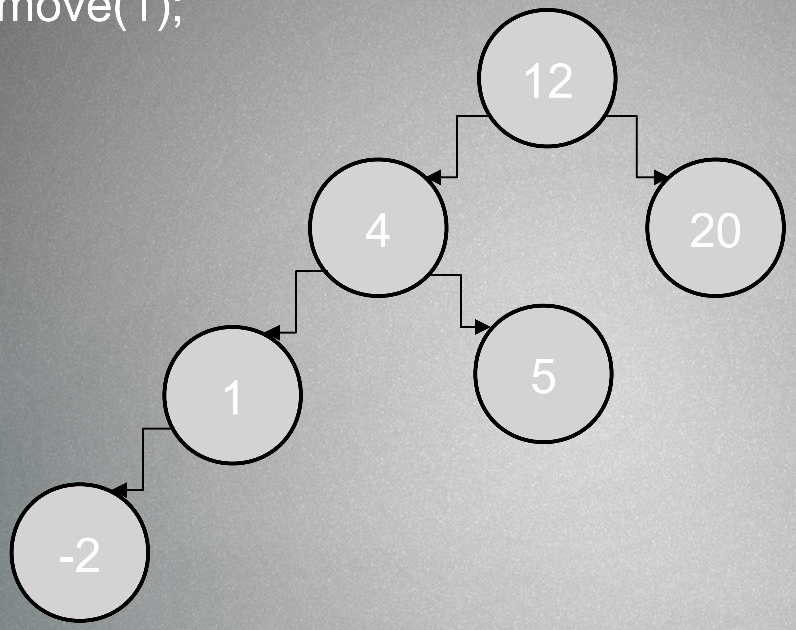


Complexity: we have to find the item itself + we have to delete it or set it to NULL

~ **O(logN)** find operation + **O(1)** deletion = **O(logN)** !!!

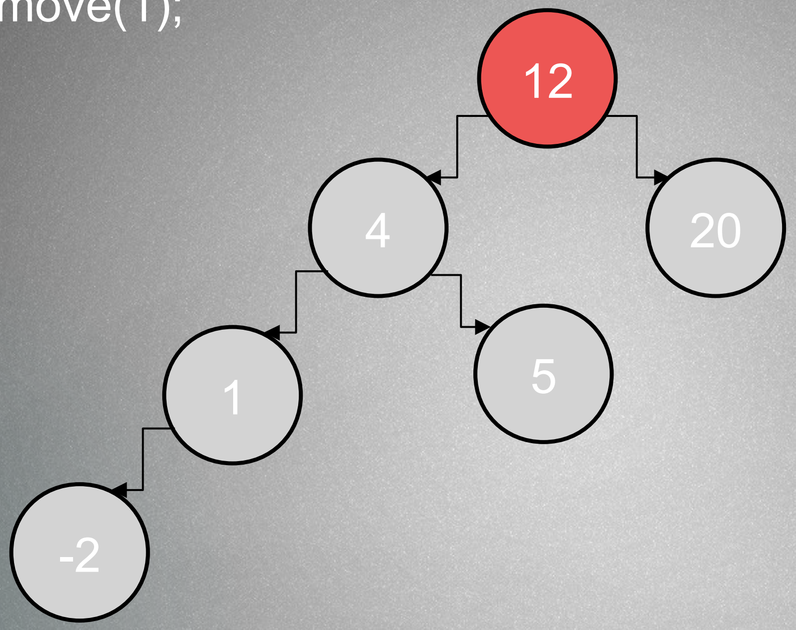
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



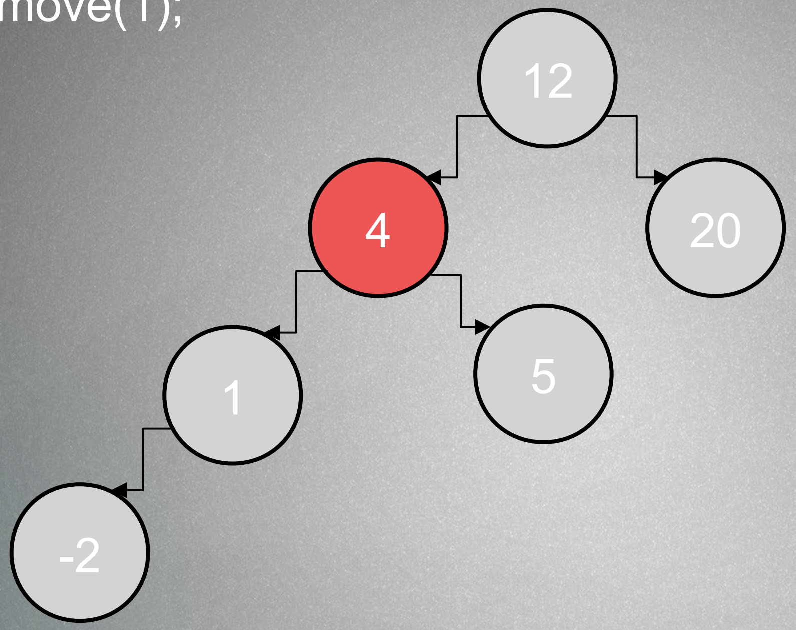
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



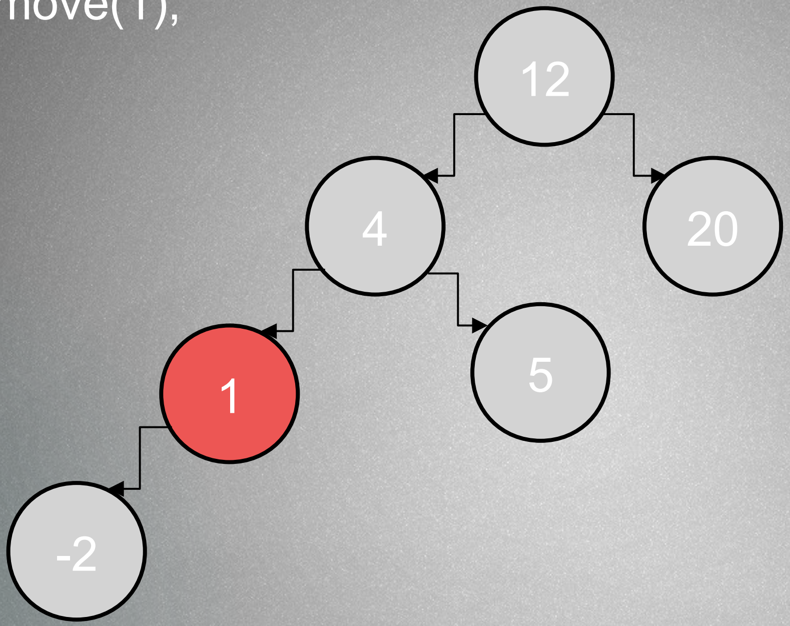
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



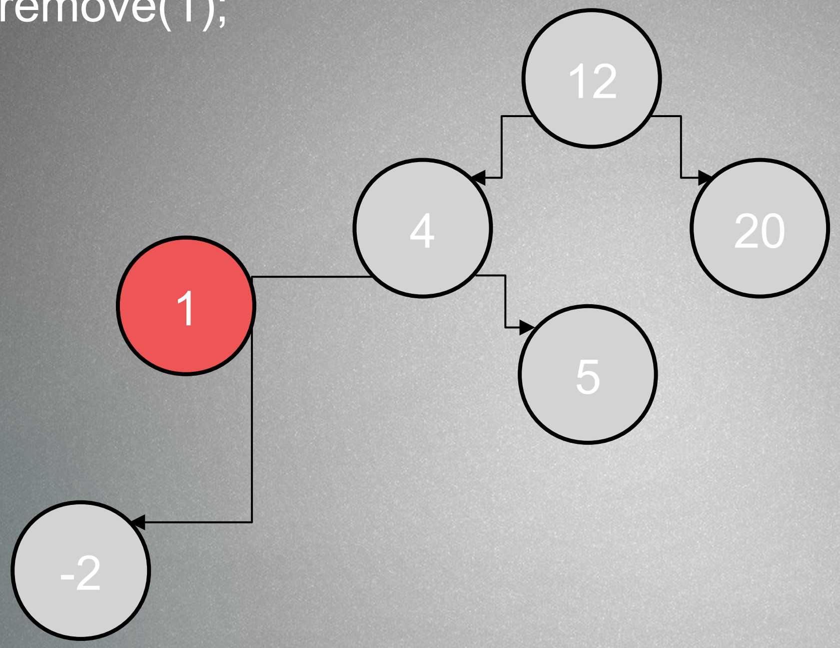
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



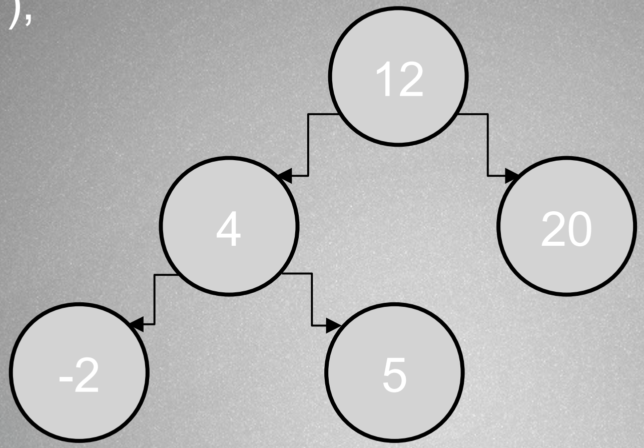
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



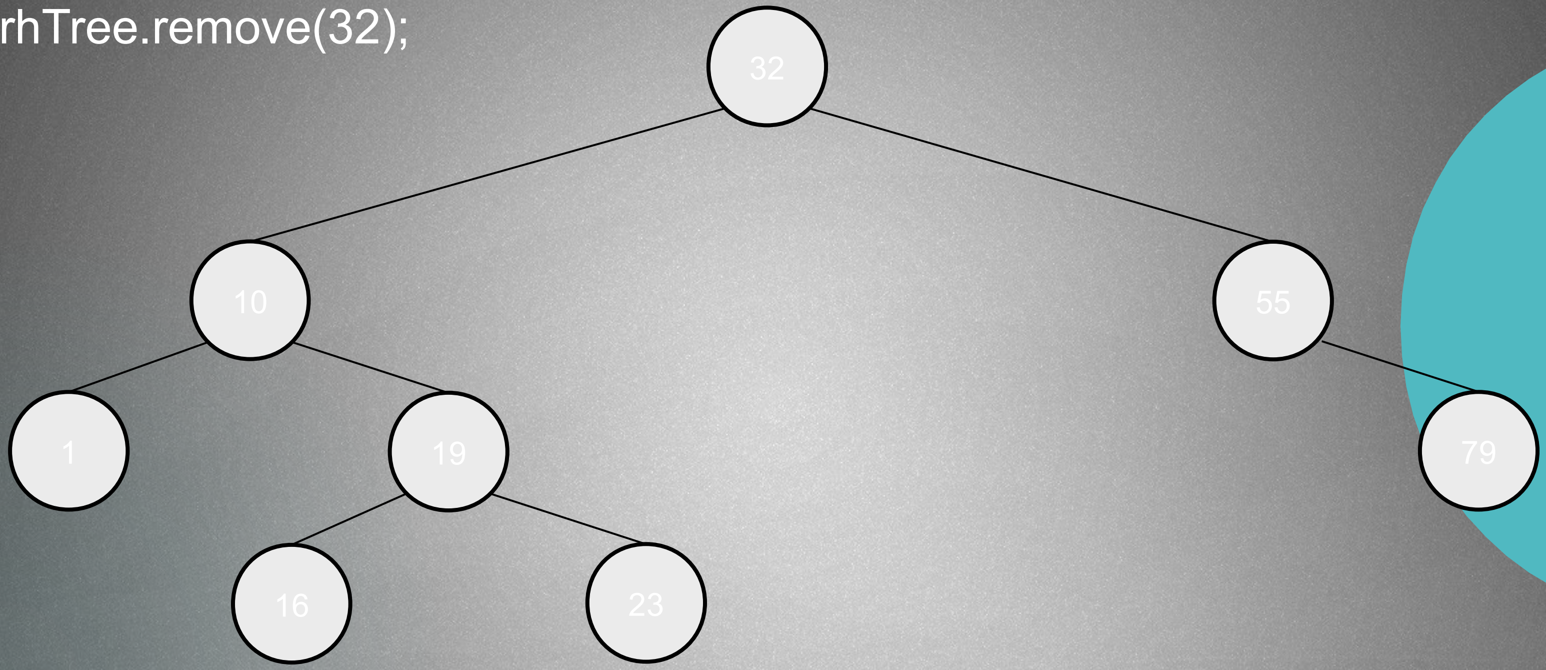
Complexity: first we have to find the item we want to get rid of and we have to update the references
~ set parent's pointer point to it's grandchild directly

O(logN) find operation + **O(1)** update references = **O(logN)** !!!

<u>Delete:</u> 3.) We want to get rid of a node that has two children

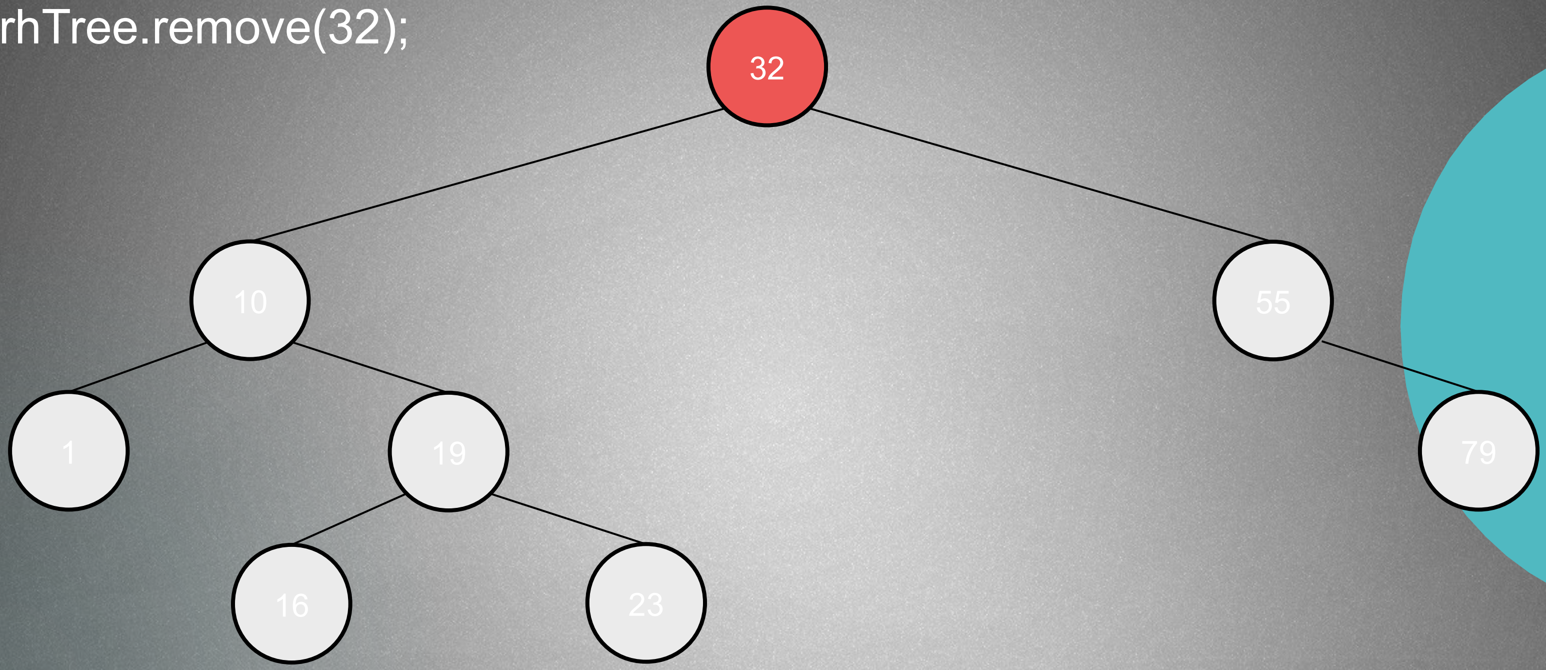
Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);



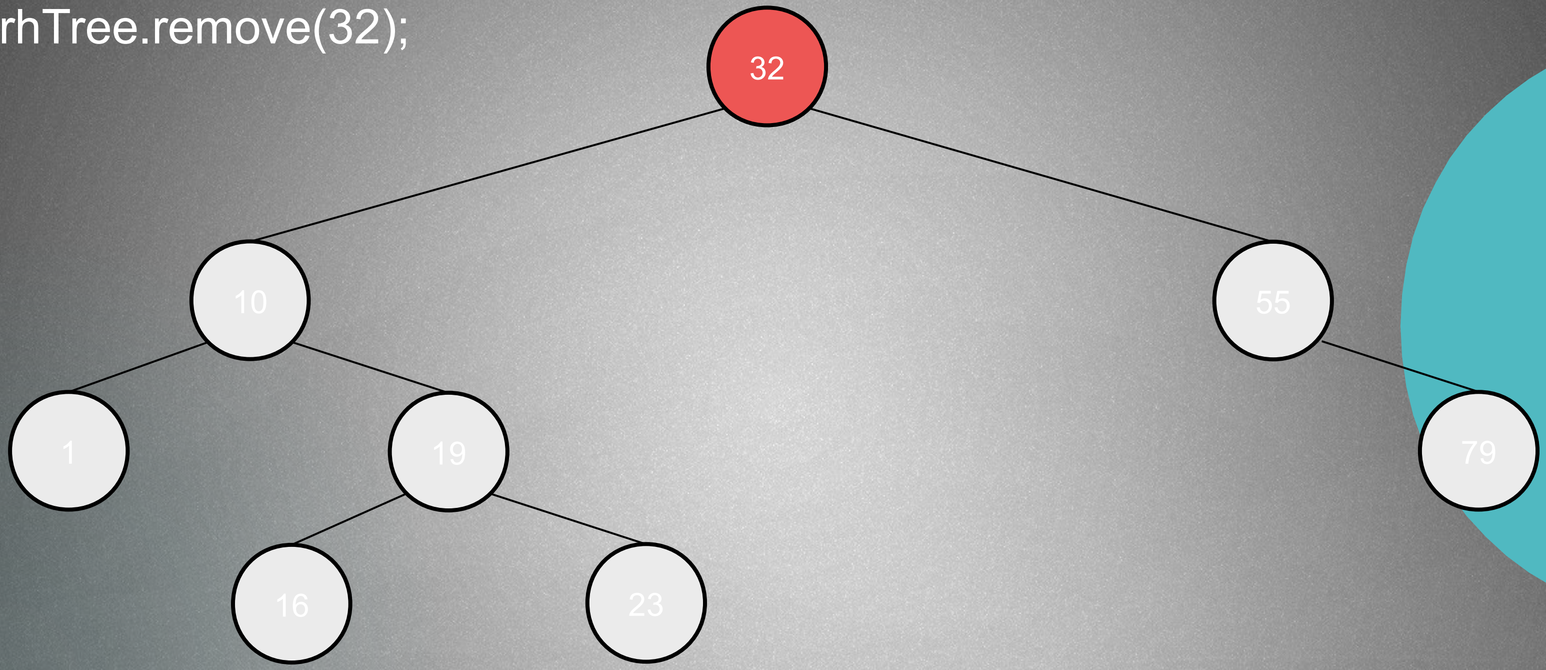
Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);



Delete: 3.) We want to get rid of a node that has two children

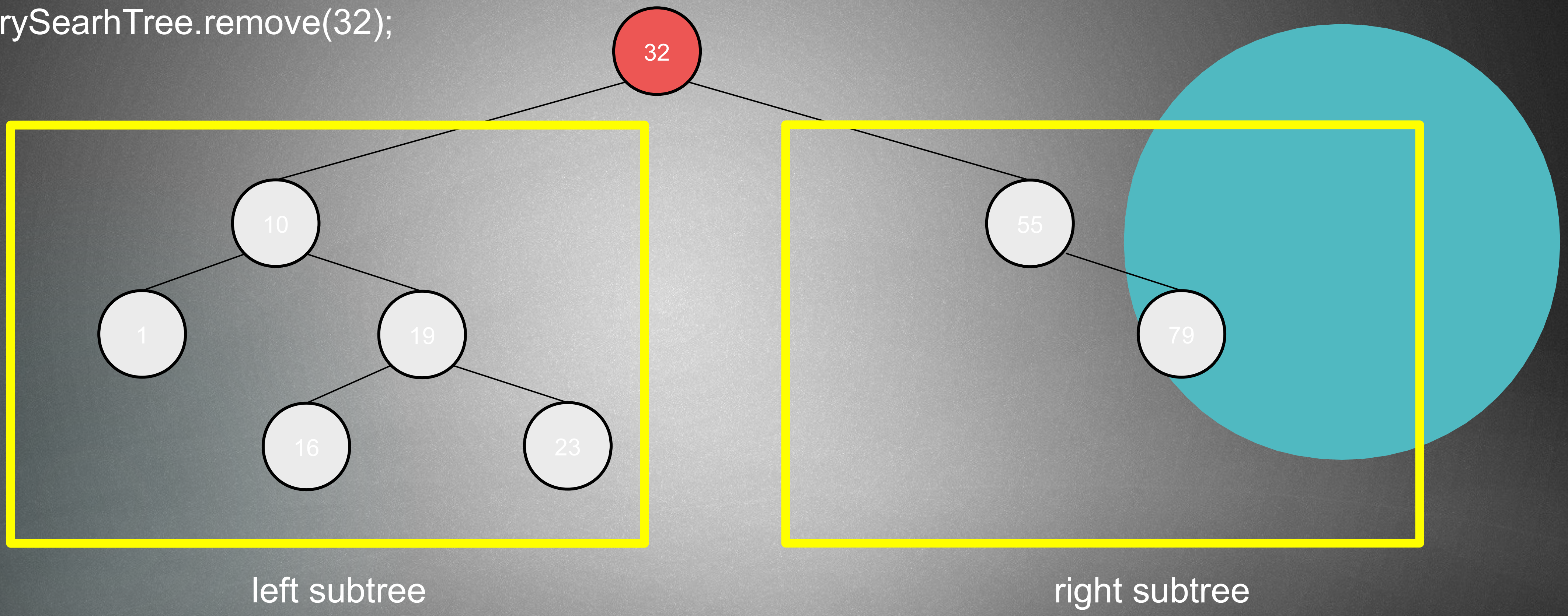
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

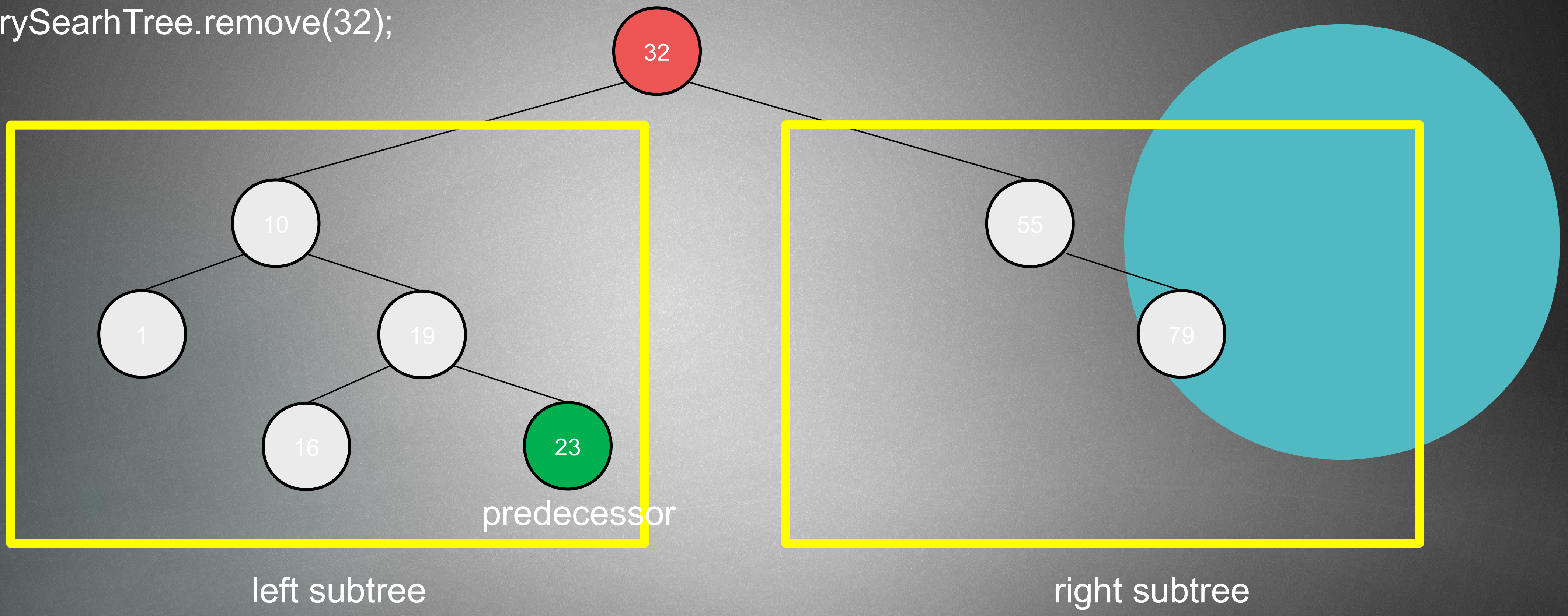
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

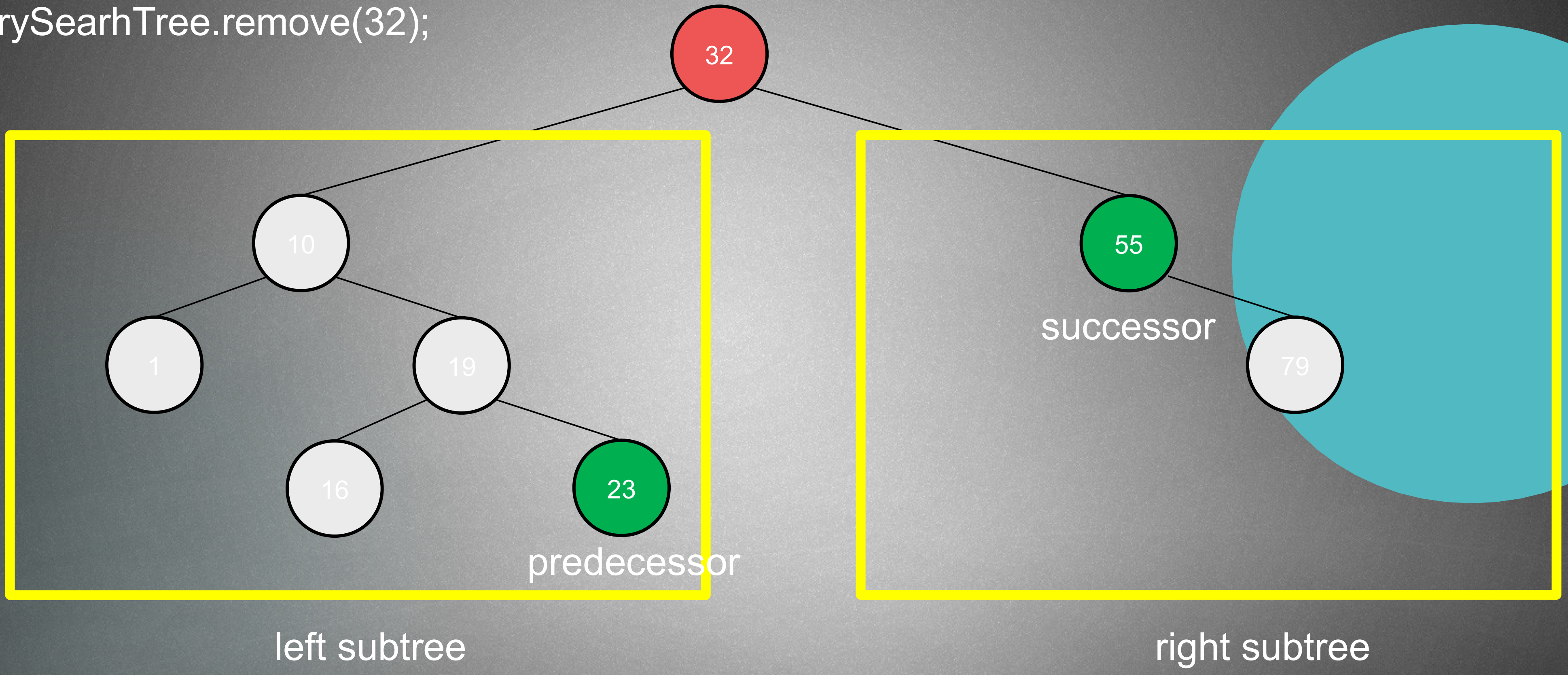
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

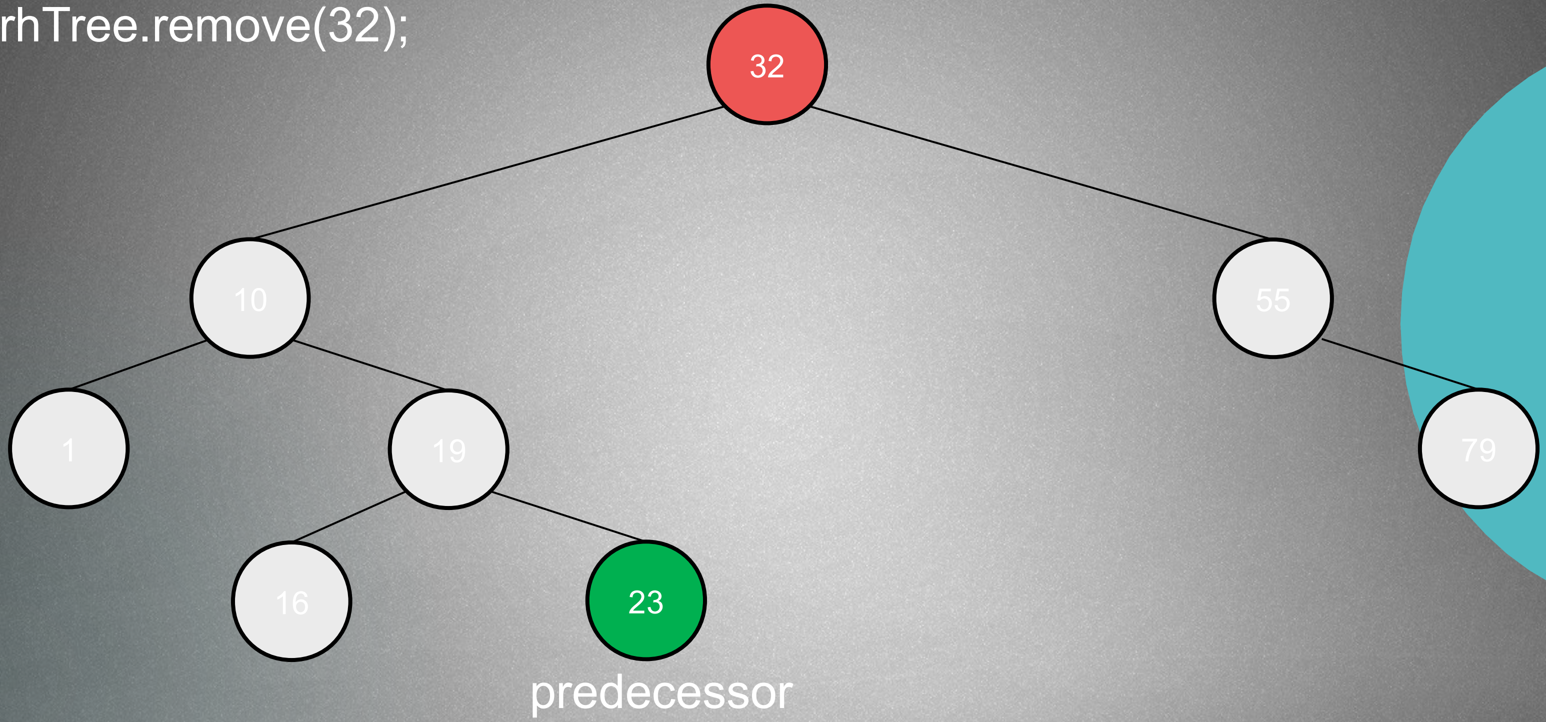
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

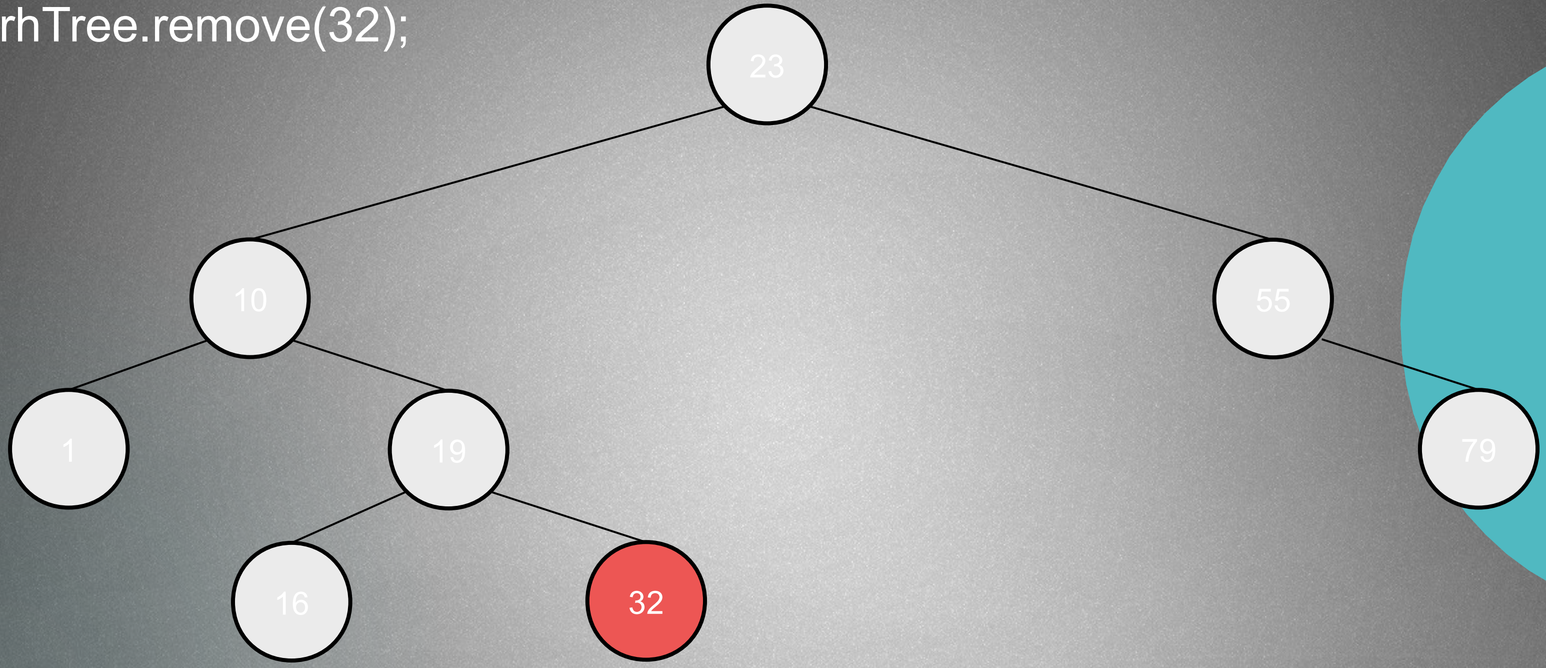
binarySearhTree.remove(32);



We look for the predecessor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

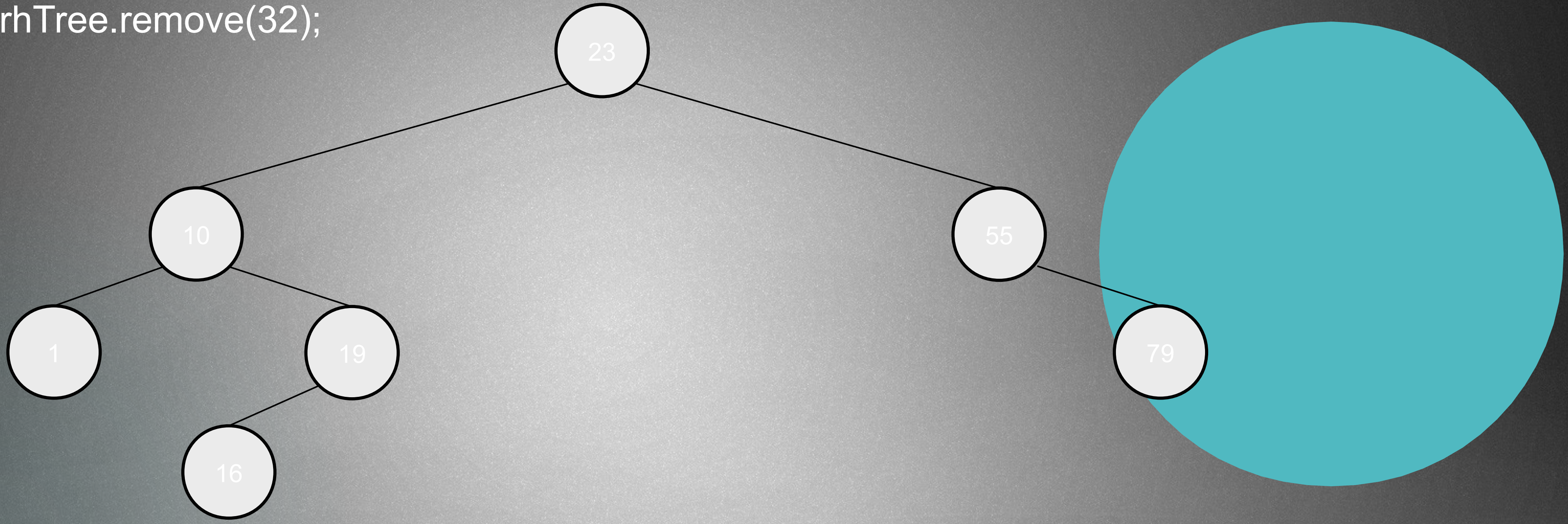
binarySearhTree.remove(32);



We look for the predecessor and swap the two nodes !!!
We end up at a case 1.) situation: we just have to set it to NULL

Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);

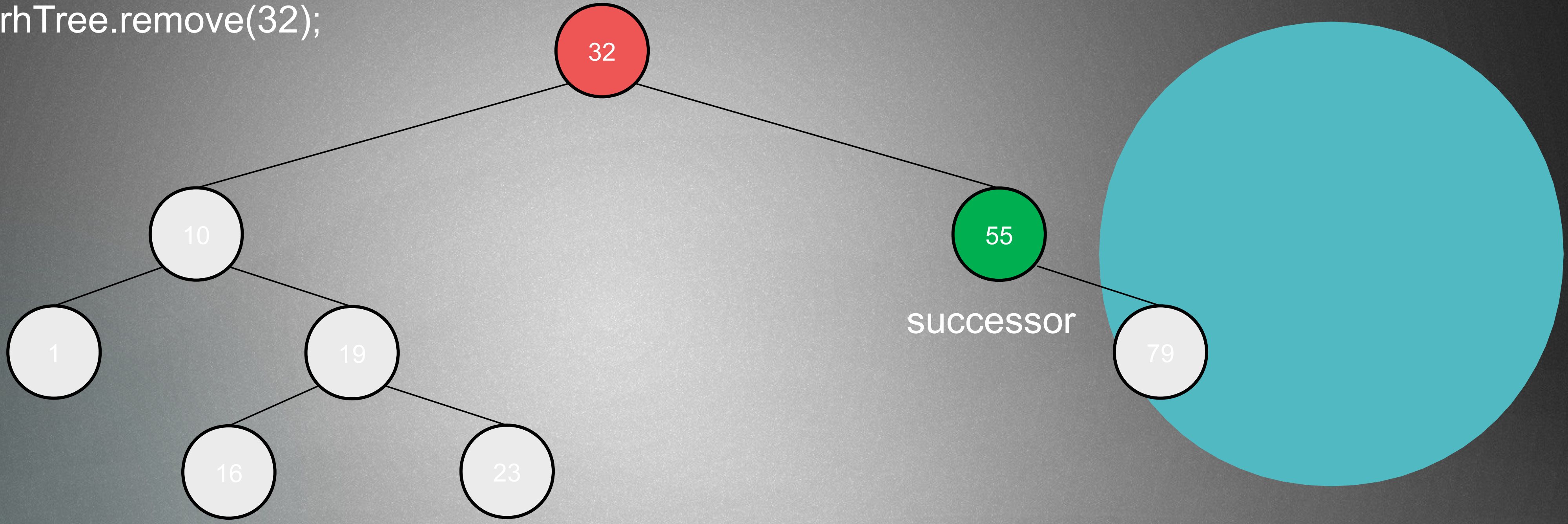


We look for the predecessor and swap the two nodes !!!

We end up at a case 1.) situation: we just have to set it to NULL

Delete: 3.) We want to get rid of a node that has two children

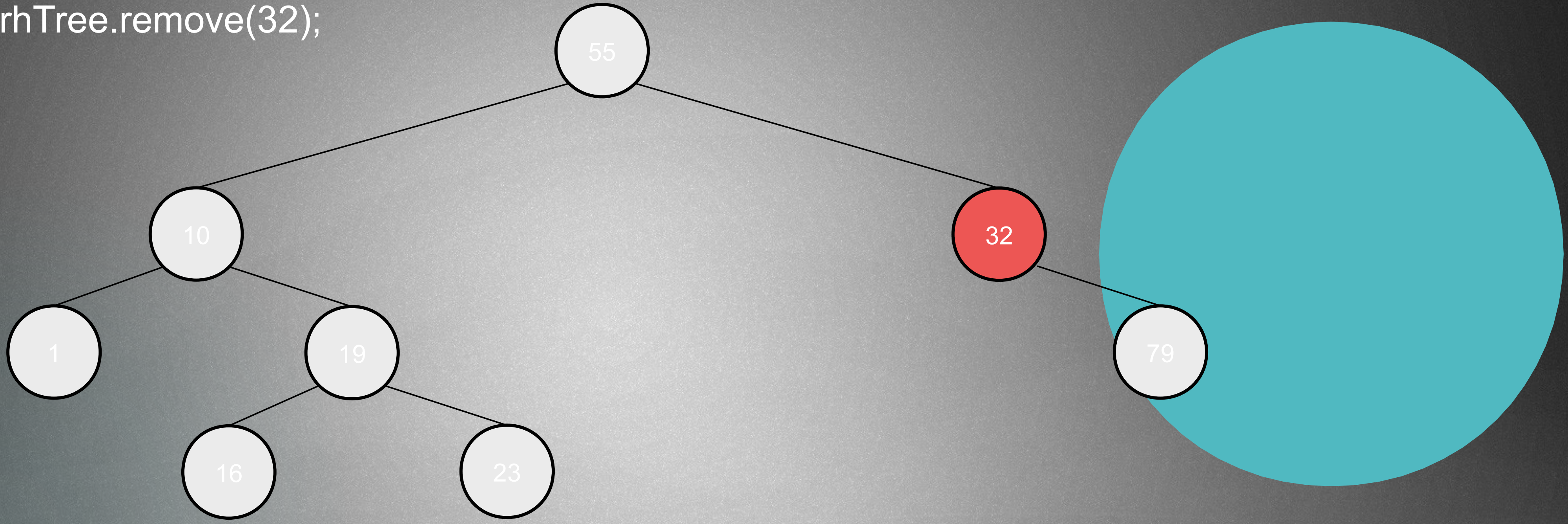
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

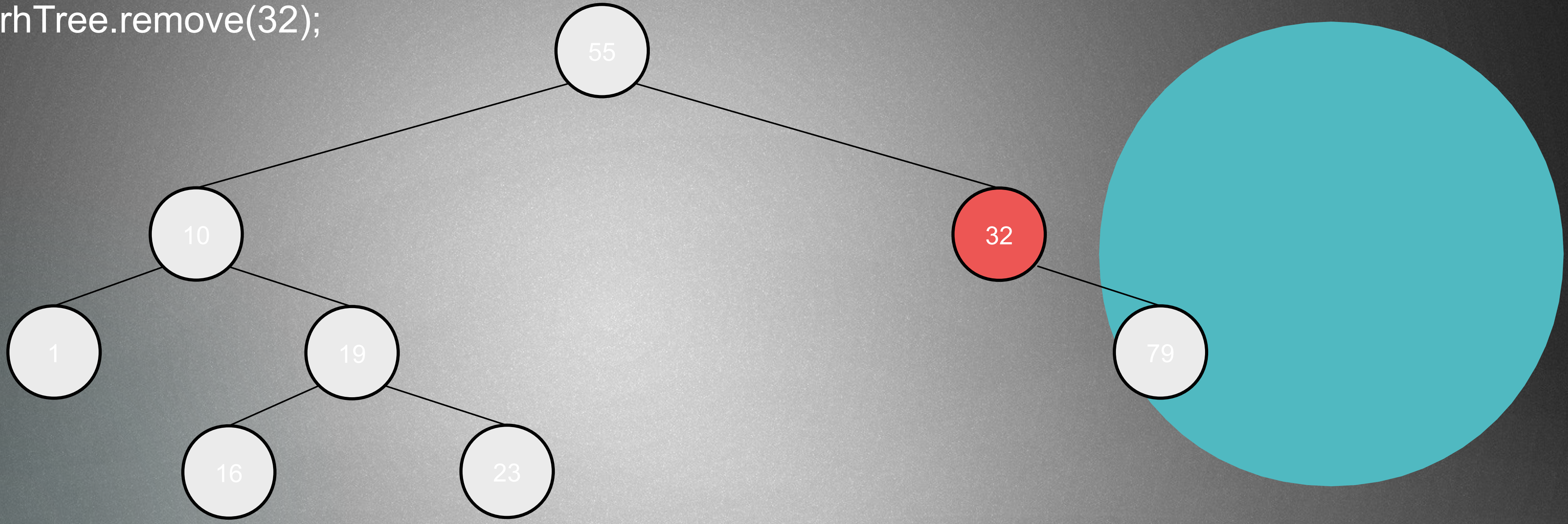
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

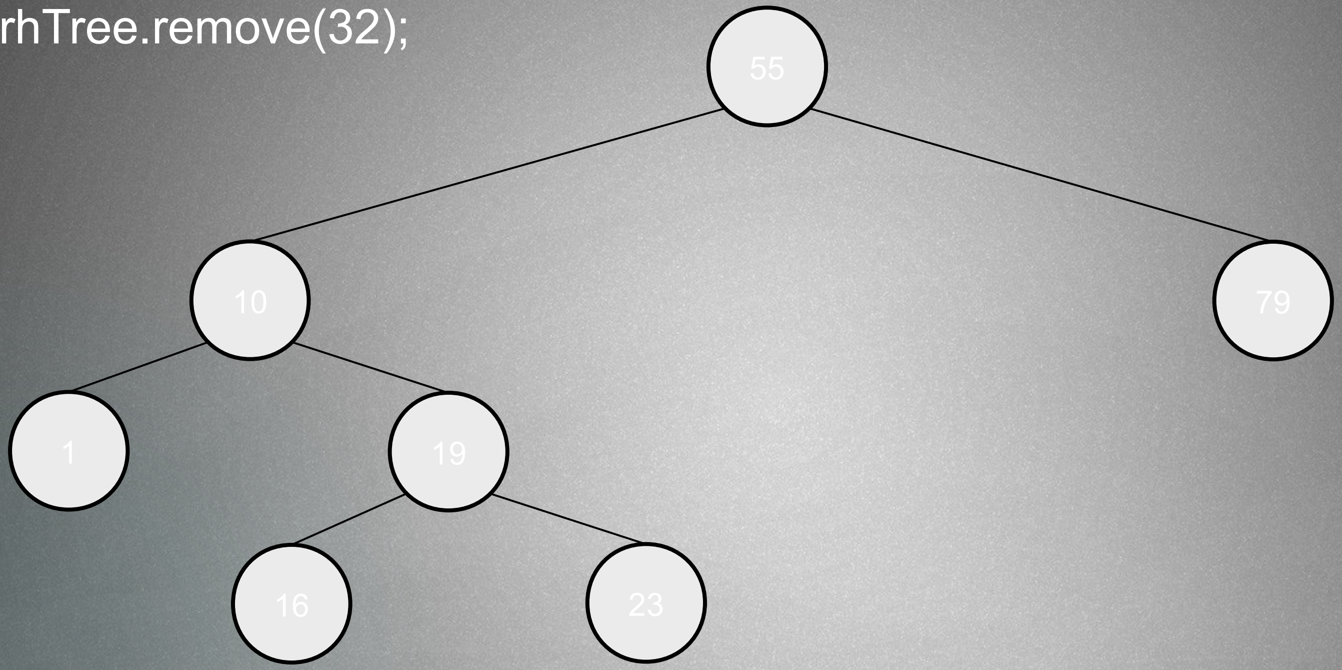
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!
This becomes the Case 2.) situation, we just have to update the references

Delete: 3.) We want to get rid of a node that has two children

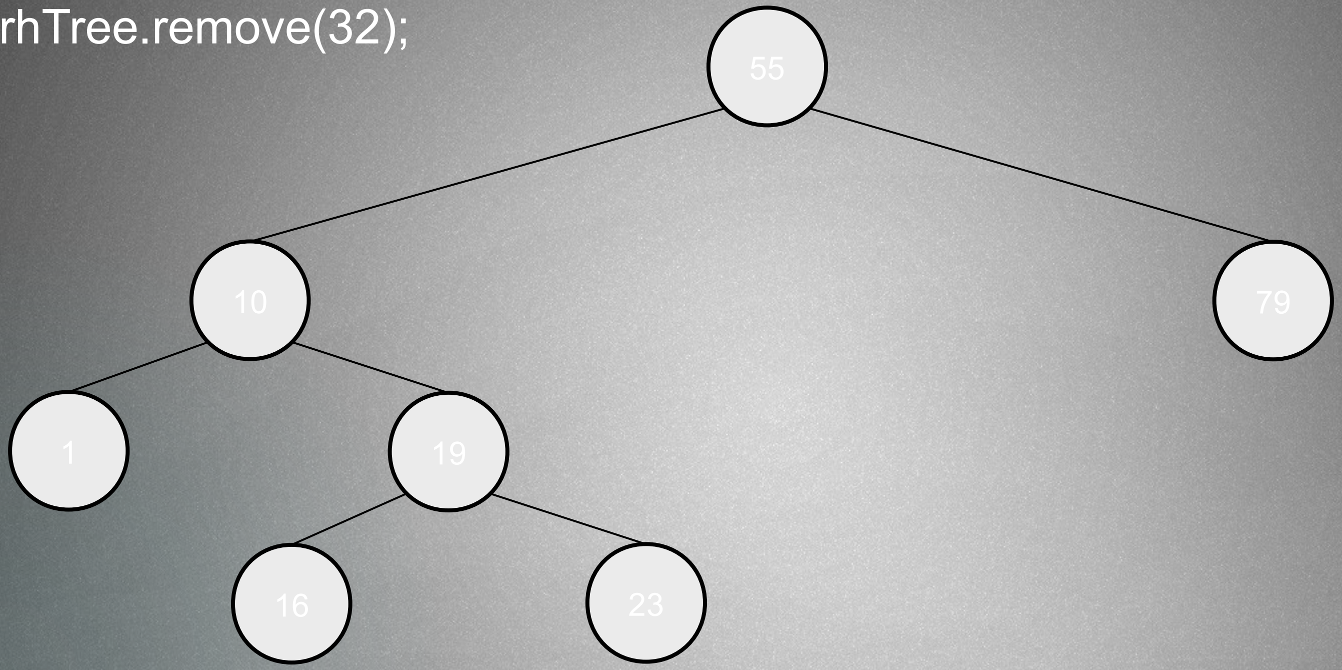
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!
This becomes the Case 2.) situation, we just have to update the references

Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);



Complexity: **O(logN)**

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

1.) In-order traversal

2.) Pre-order traversal

3.) Post-order traversal

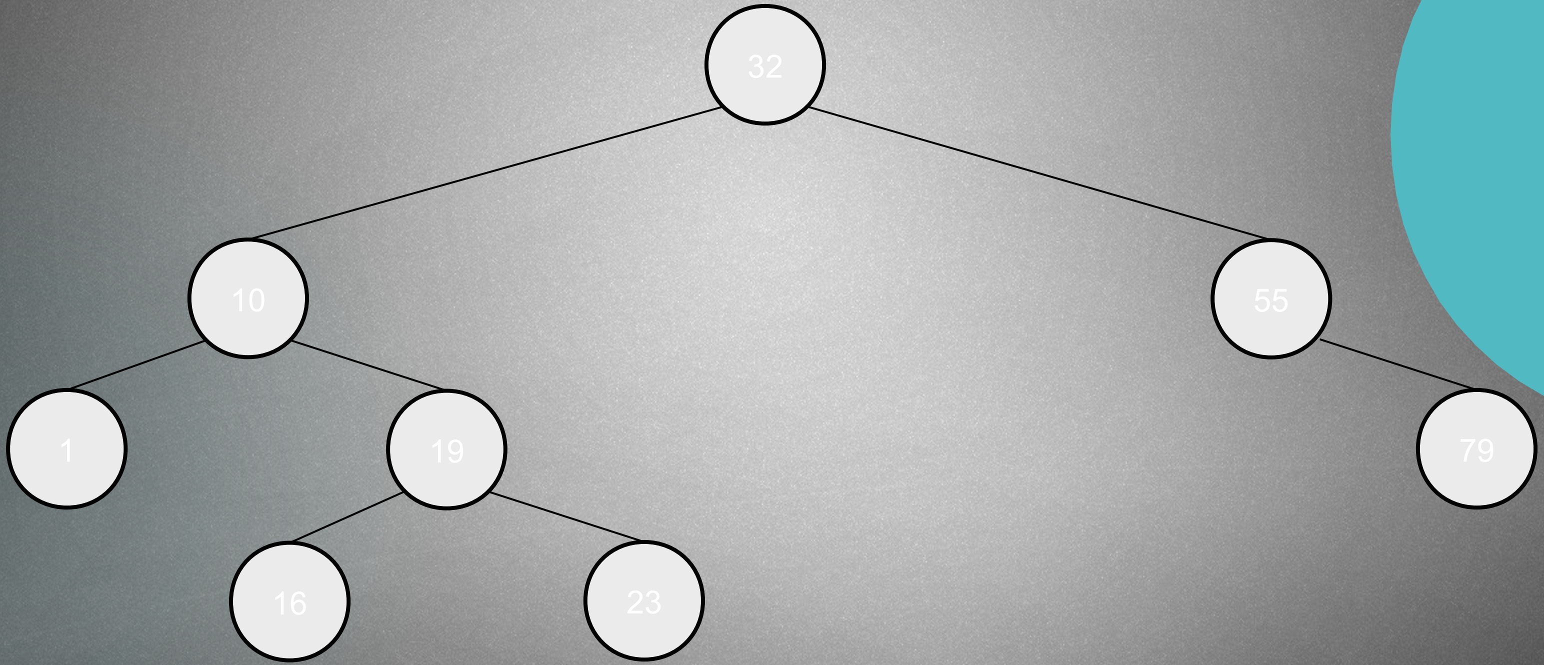
Traversal: sometimes it is neccessary to visit every node in the tree

We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!

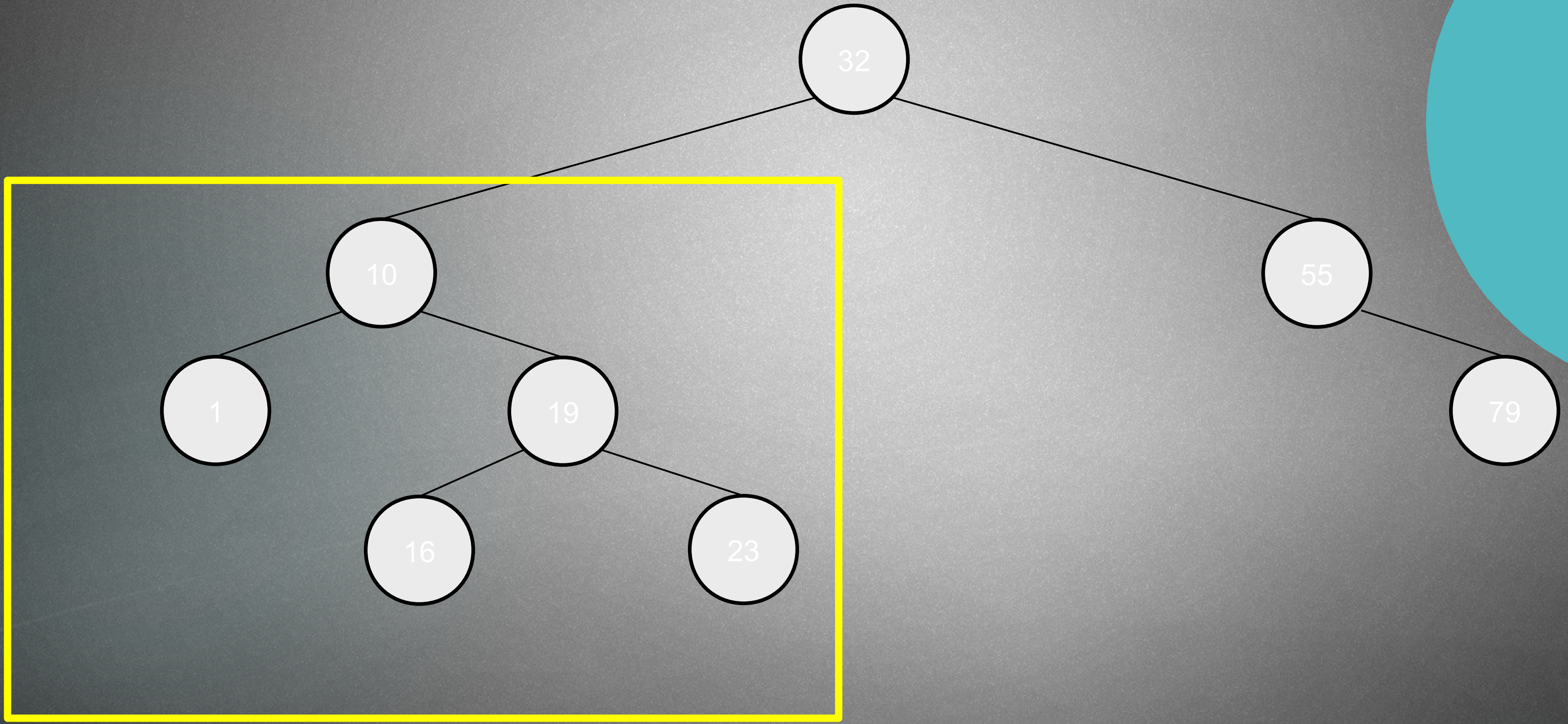
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

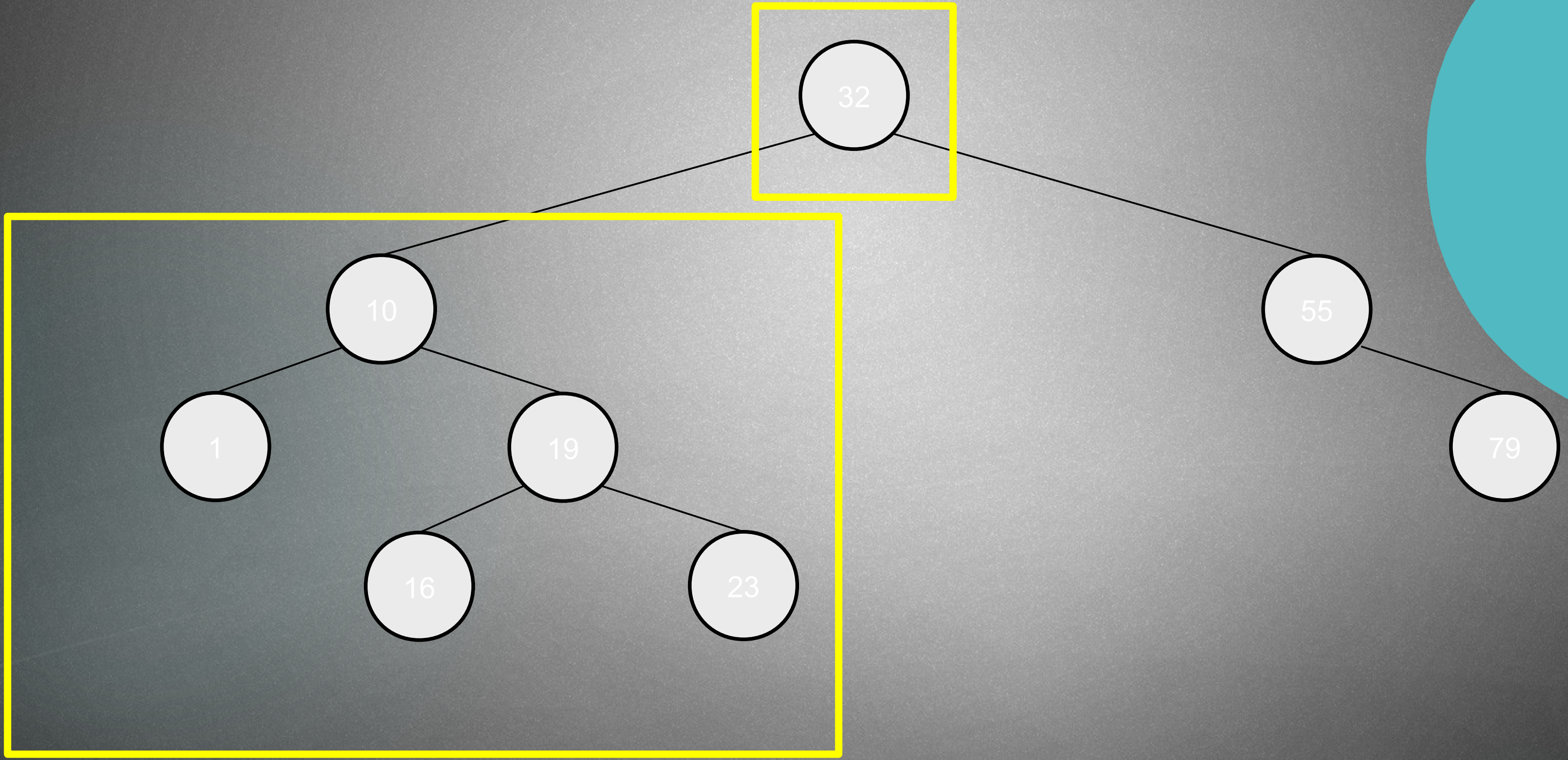
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree

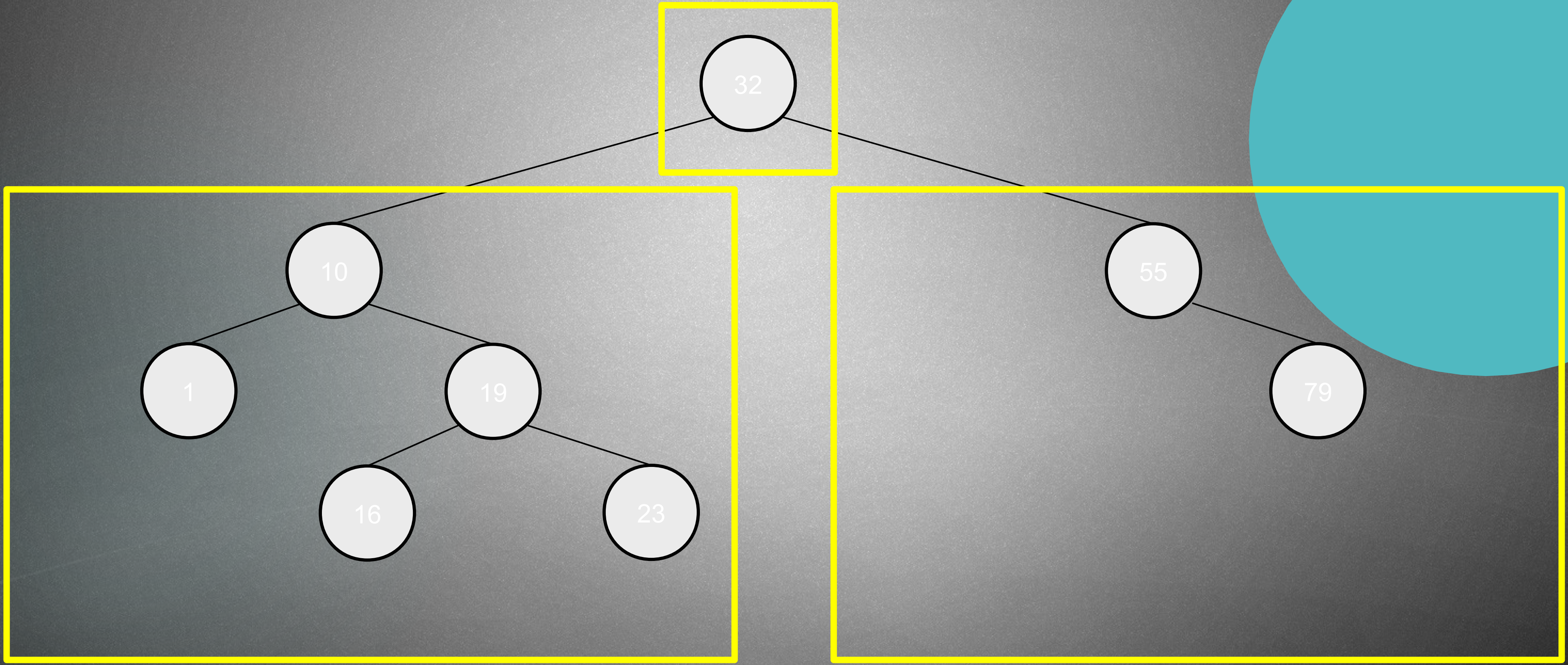
We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



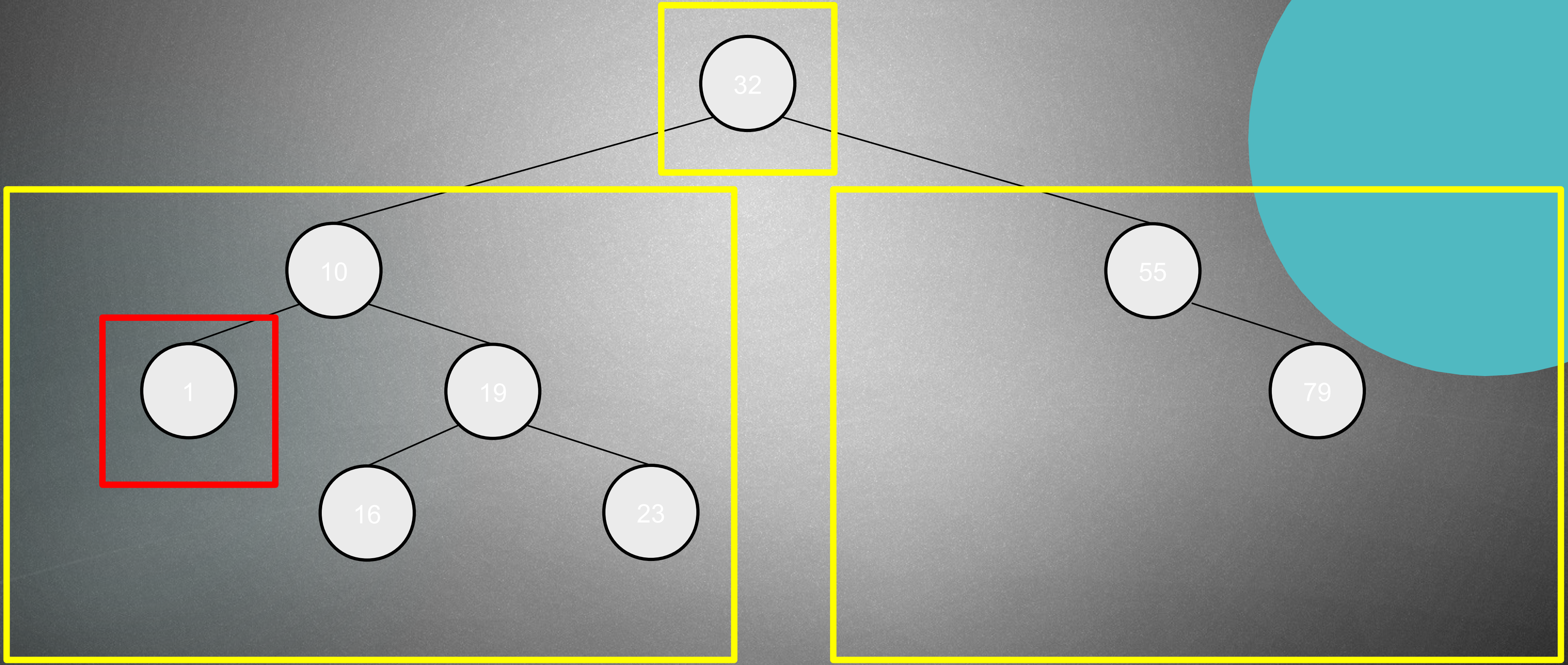
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

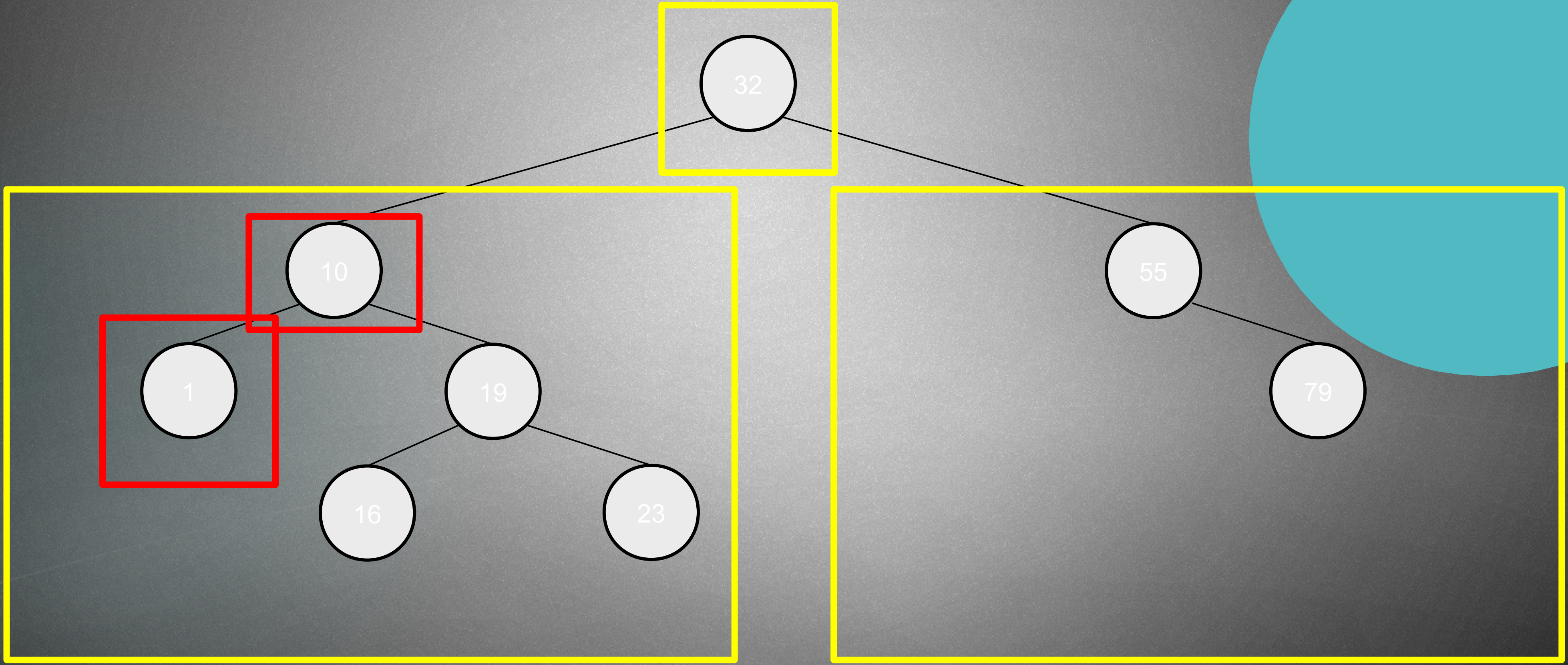
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree

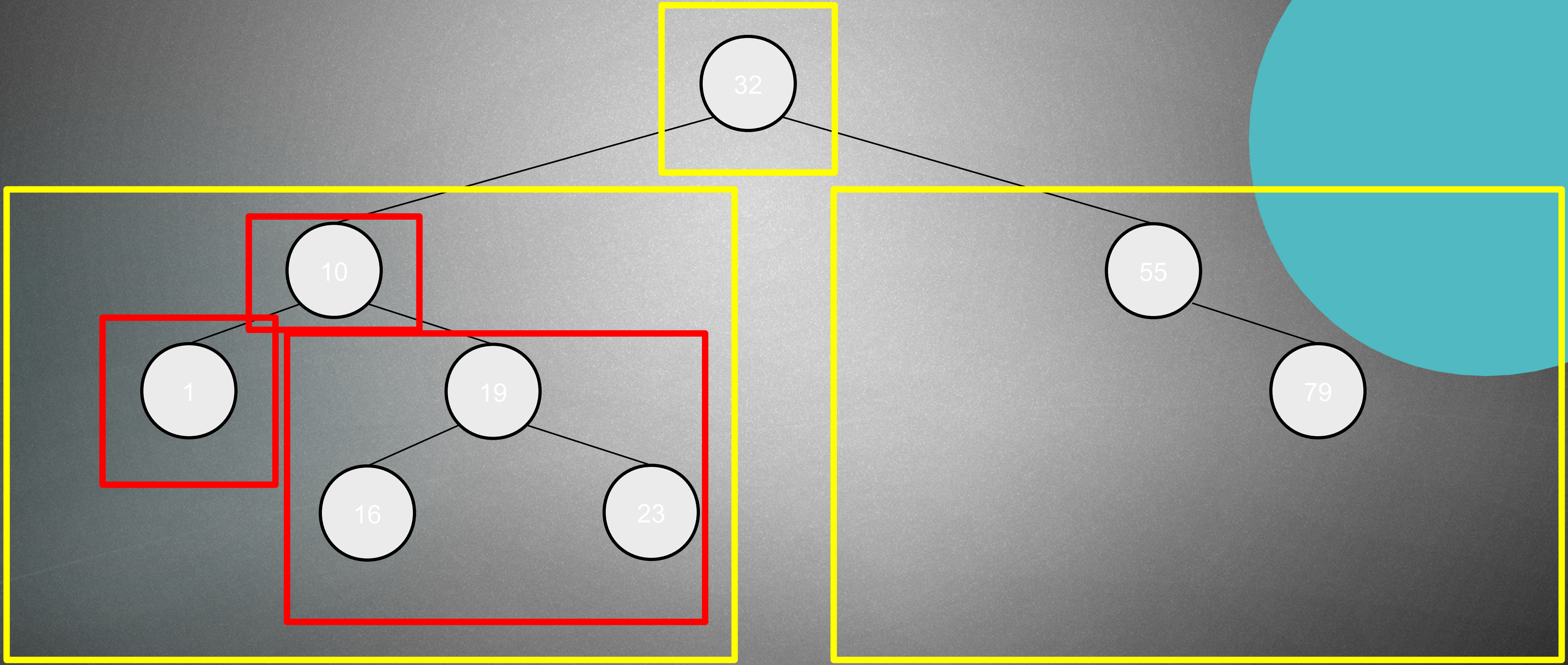
We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



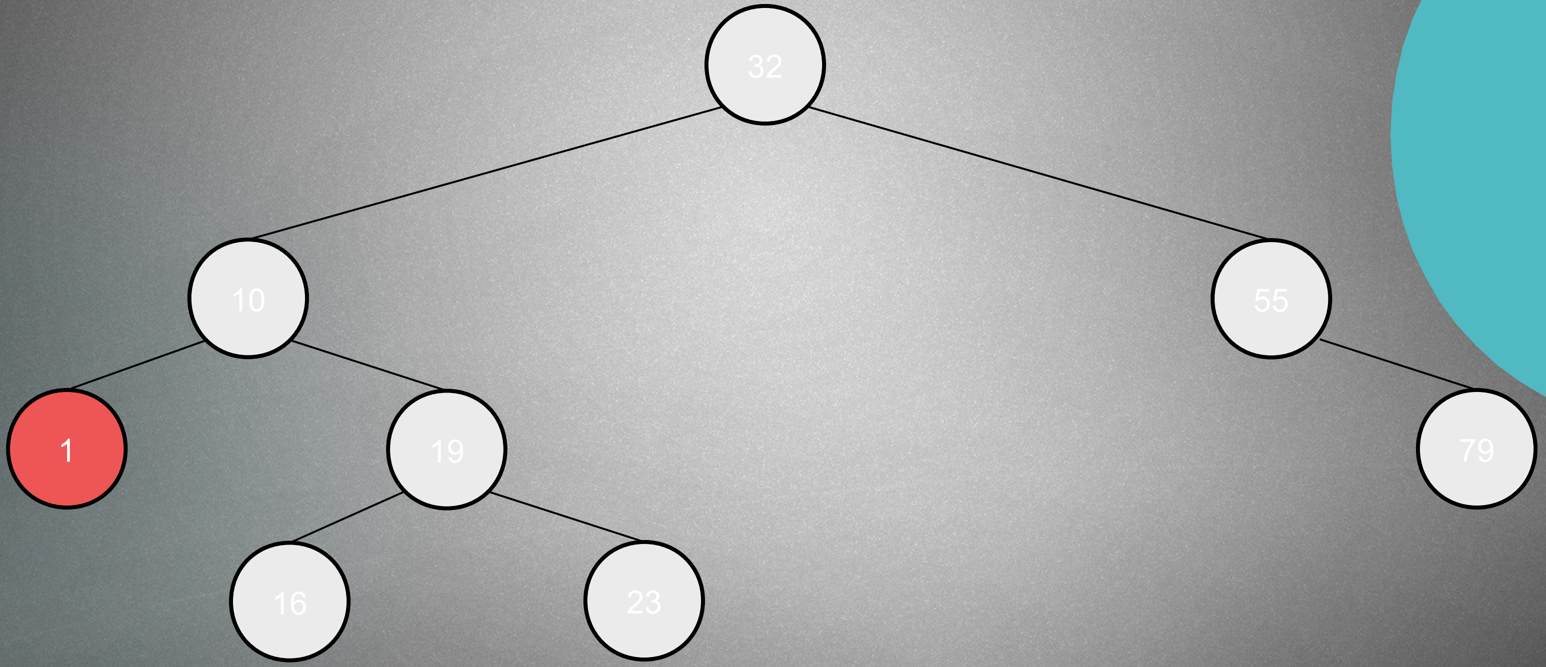
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

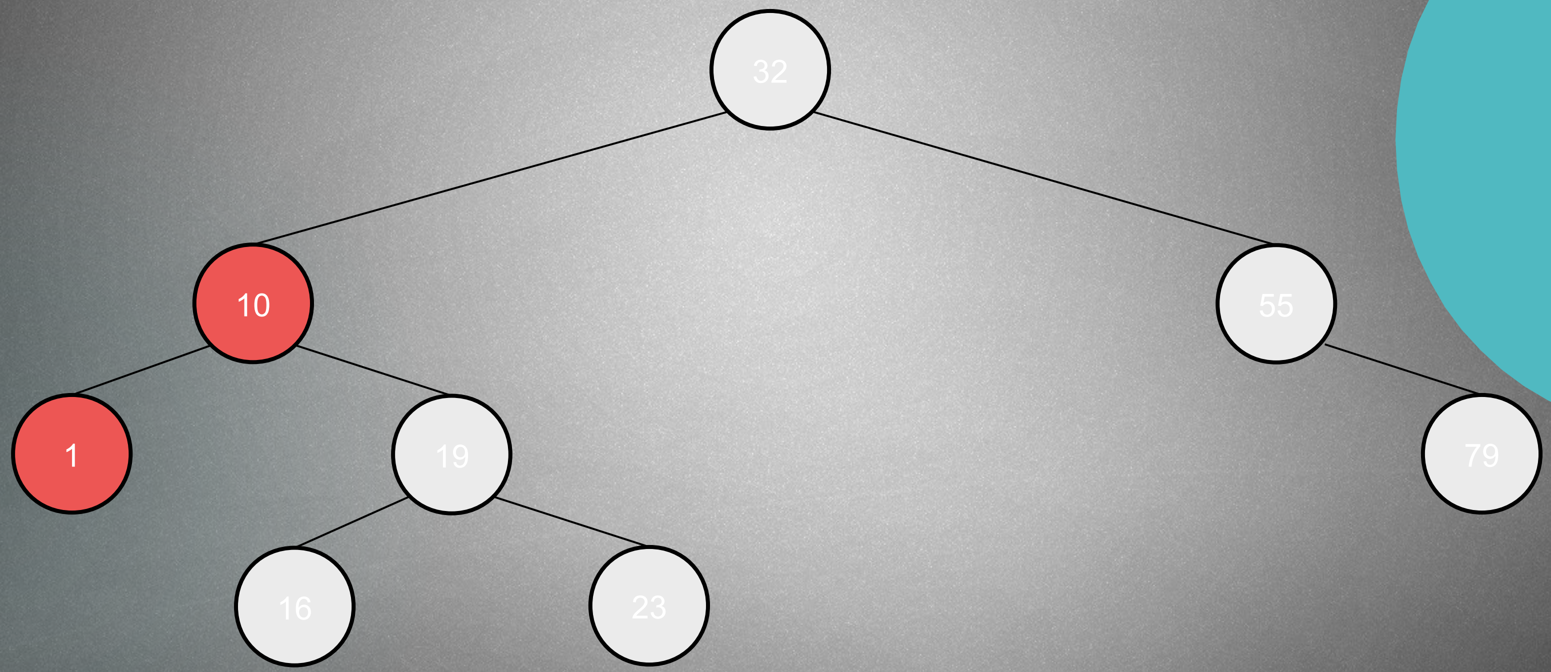
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

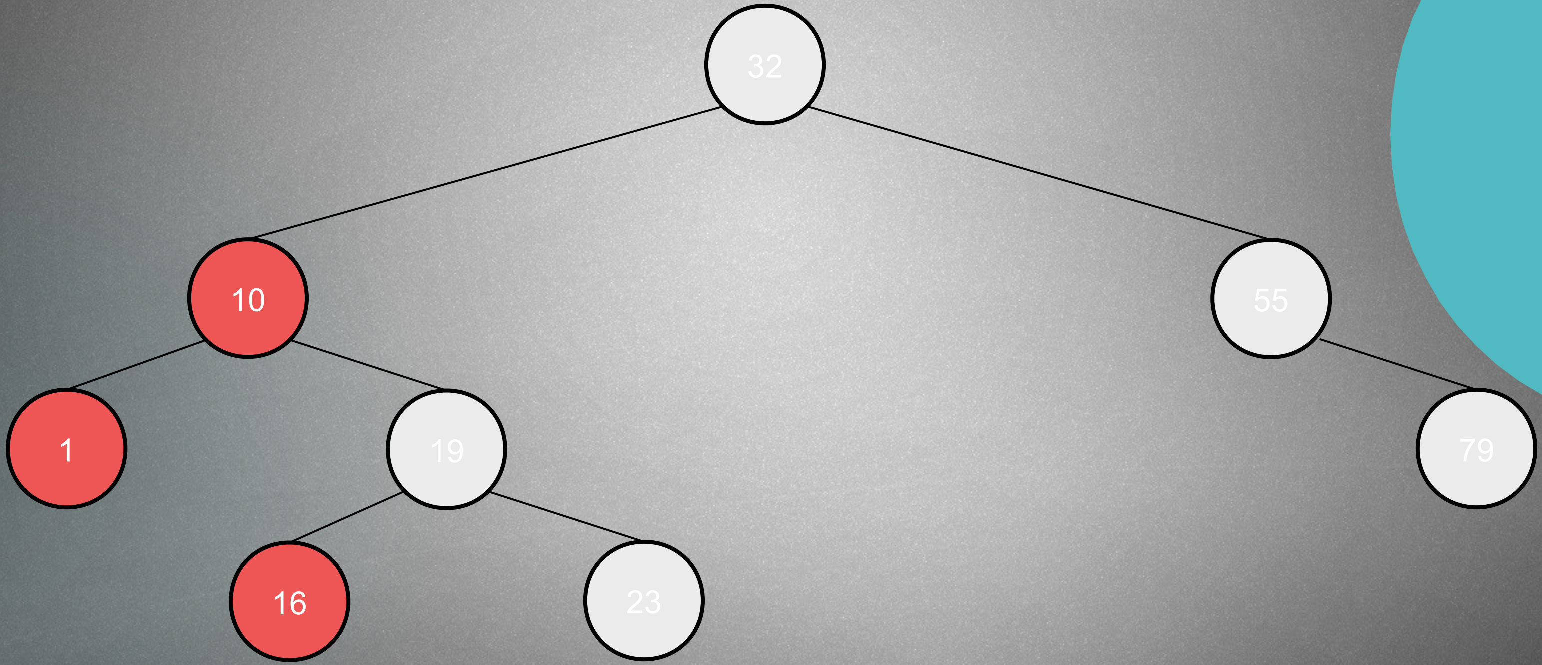
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

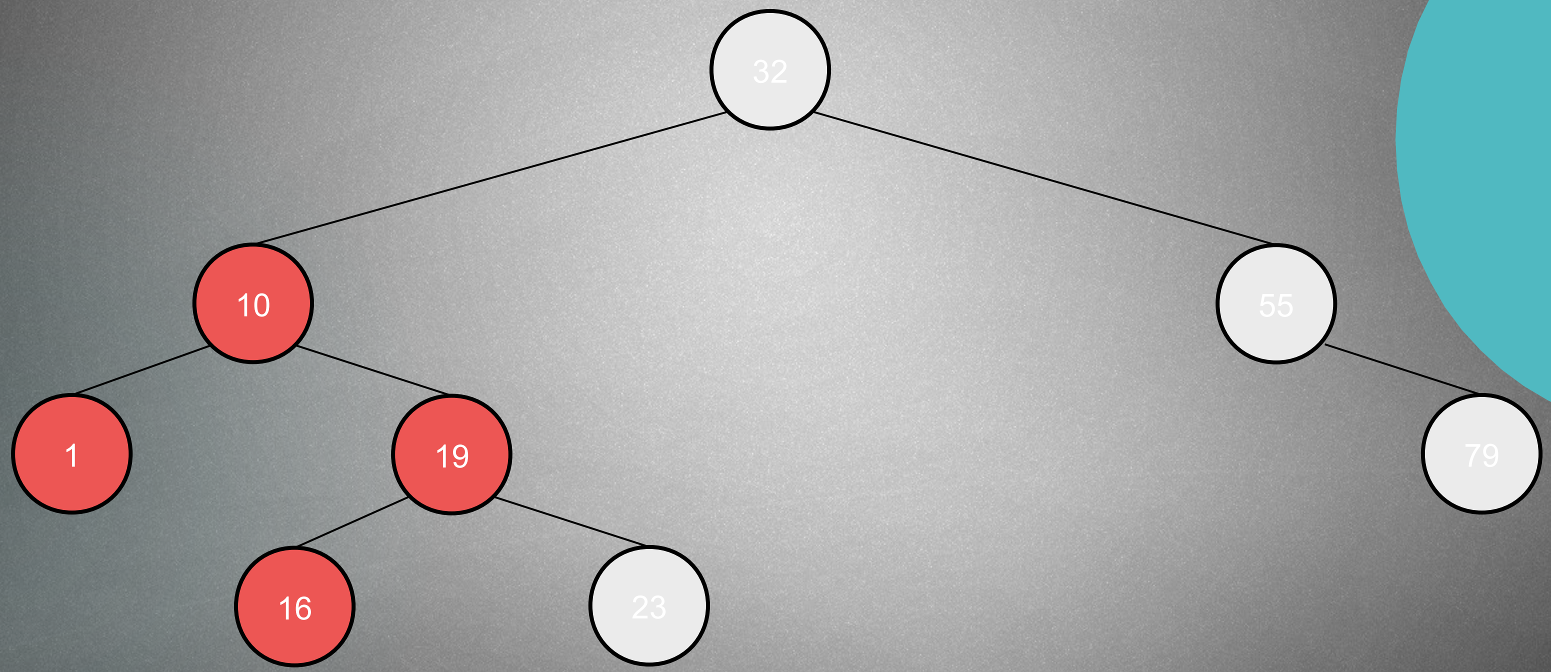
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

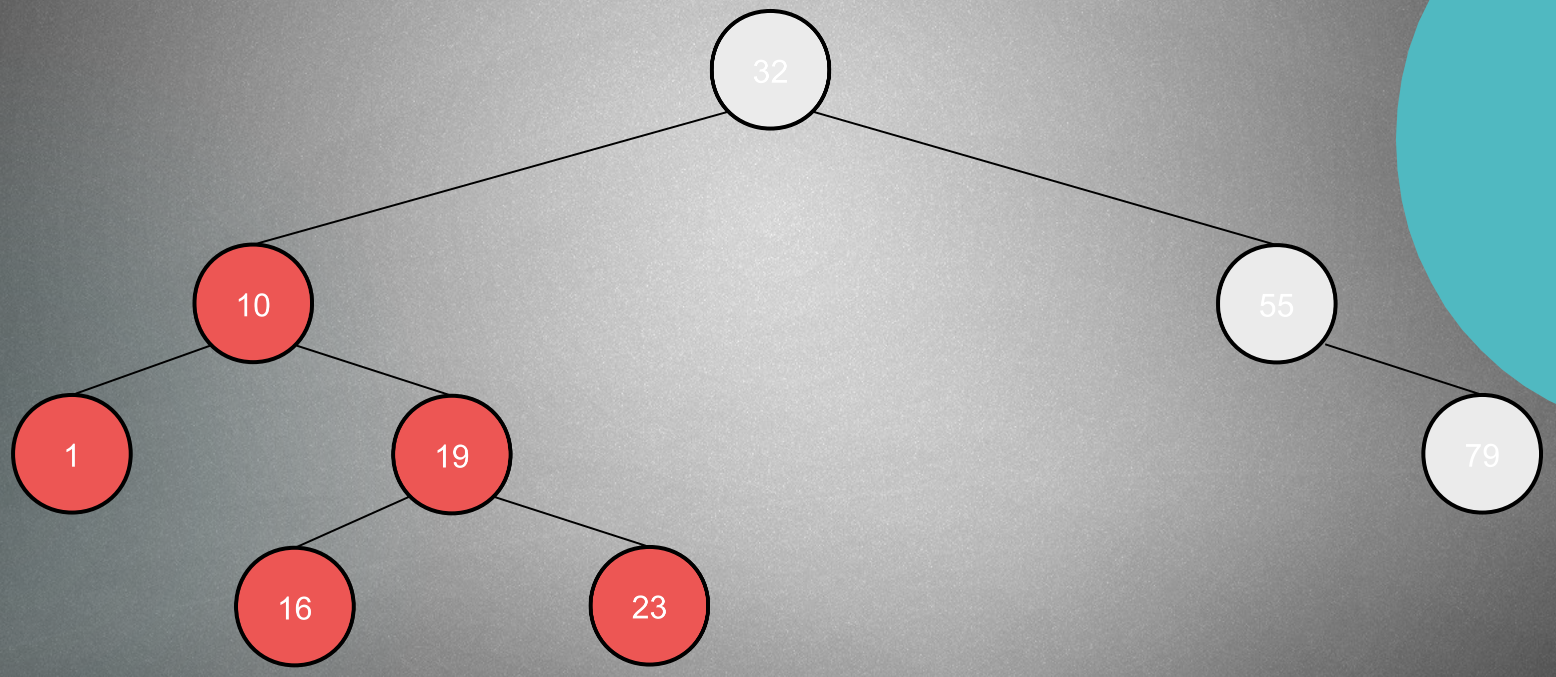
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

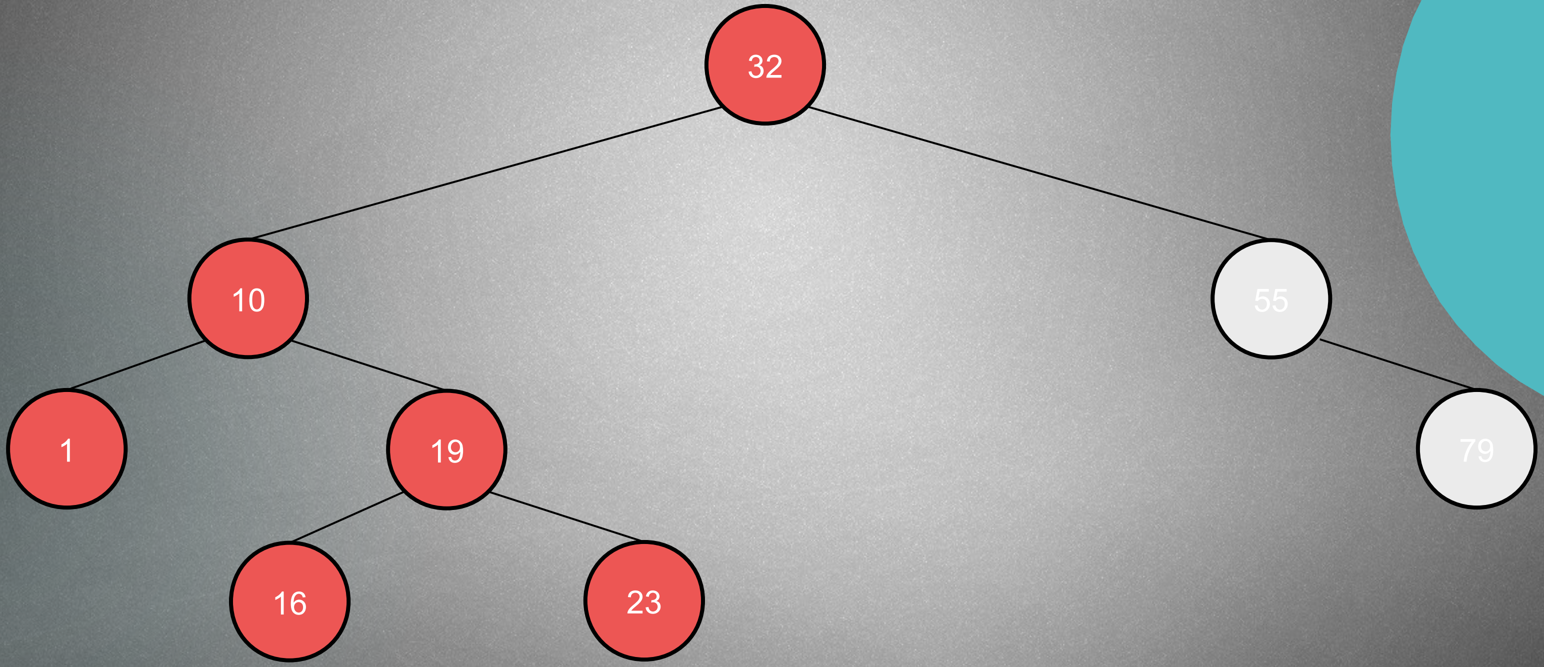
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19 – 23

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

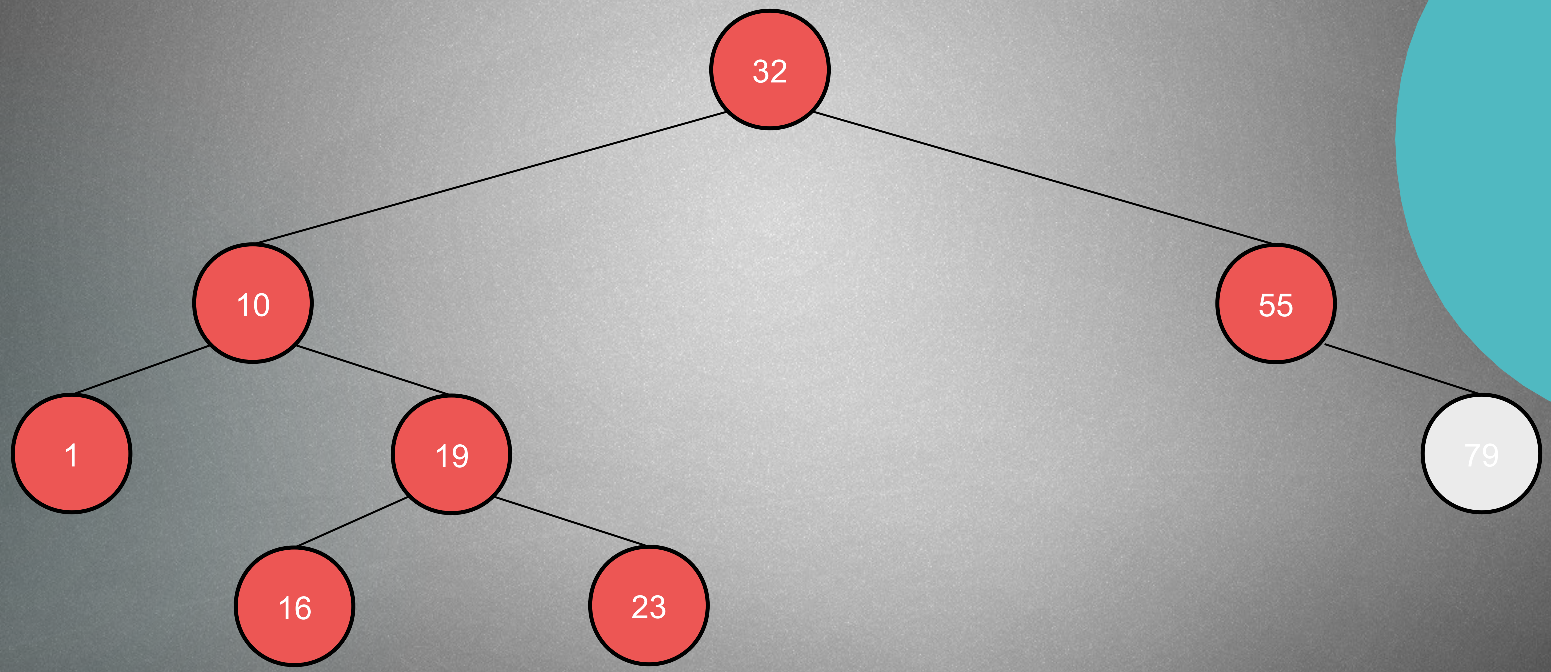
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19 – 23 – 32

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

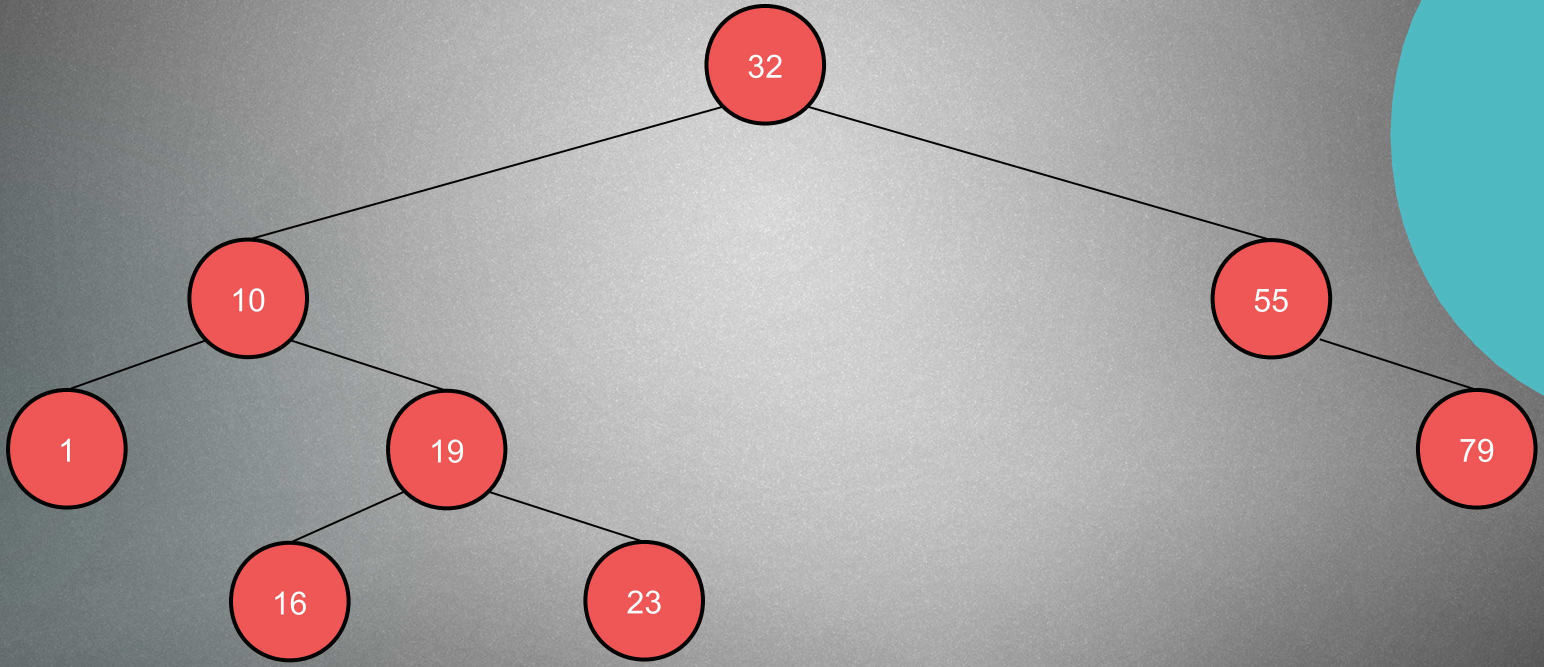
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19 – 23 – 32 – 55

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

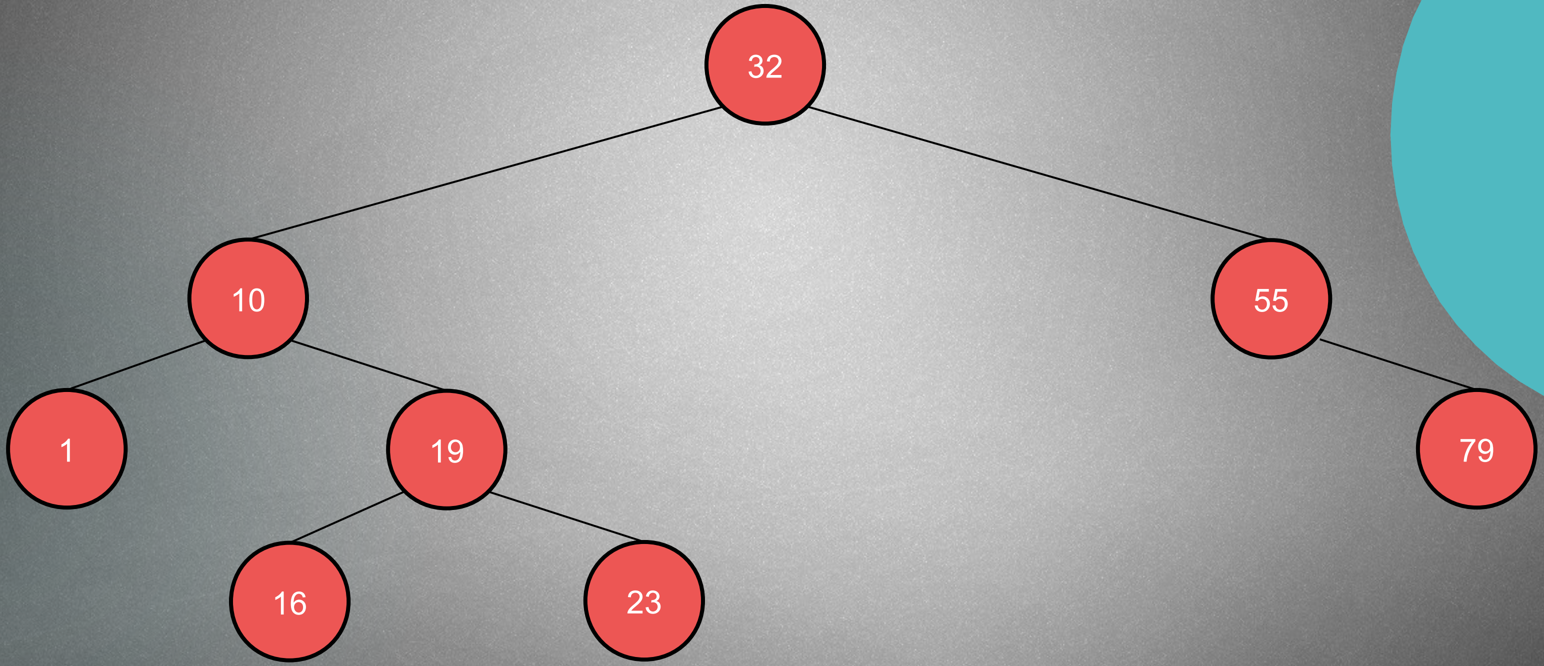
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19 – 23 – 32 – 55 – 79

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

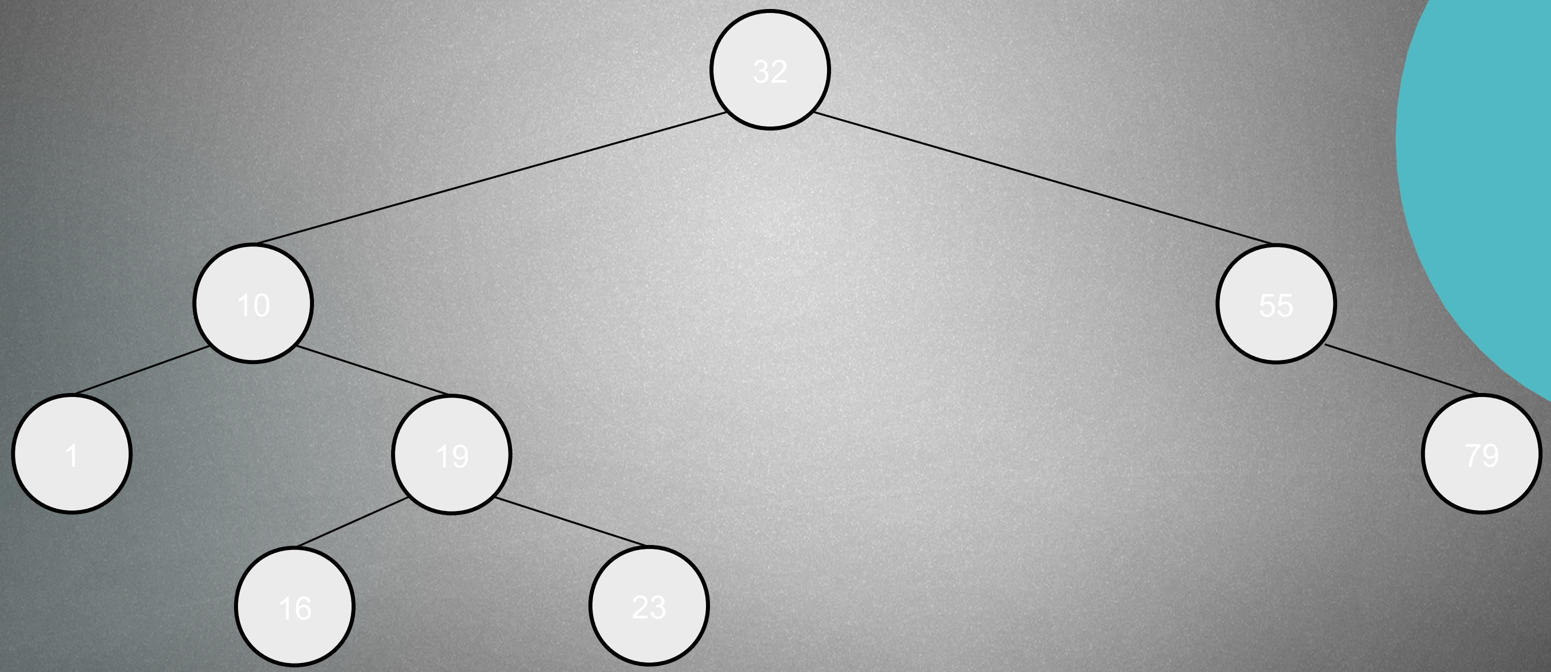
1.) In-order traversal: we visit the left subtree + the root node + the right subtree recursively !!!



In-order traversal solution: 1 – 10 – 16 – 19 – 23 – 32 – 55 – 79 SO IT IS THE NUMERICAL ORDERING !!!

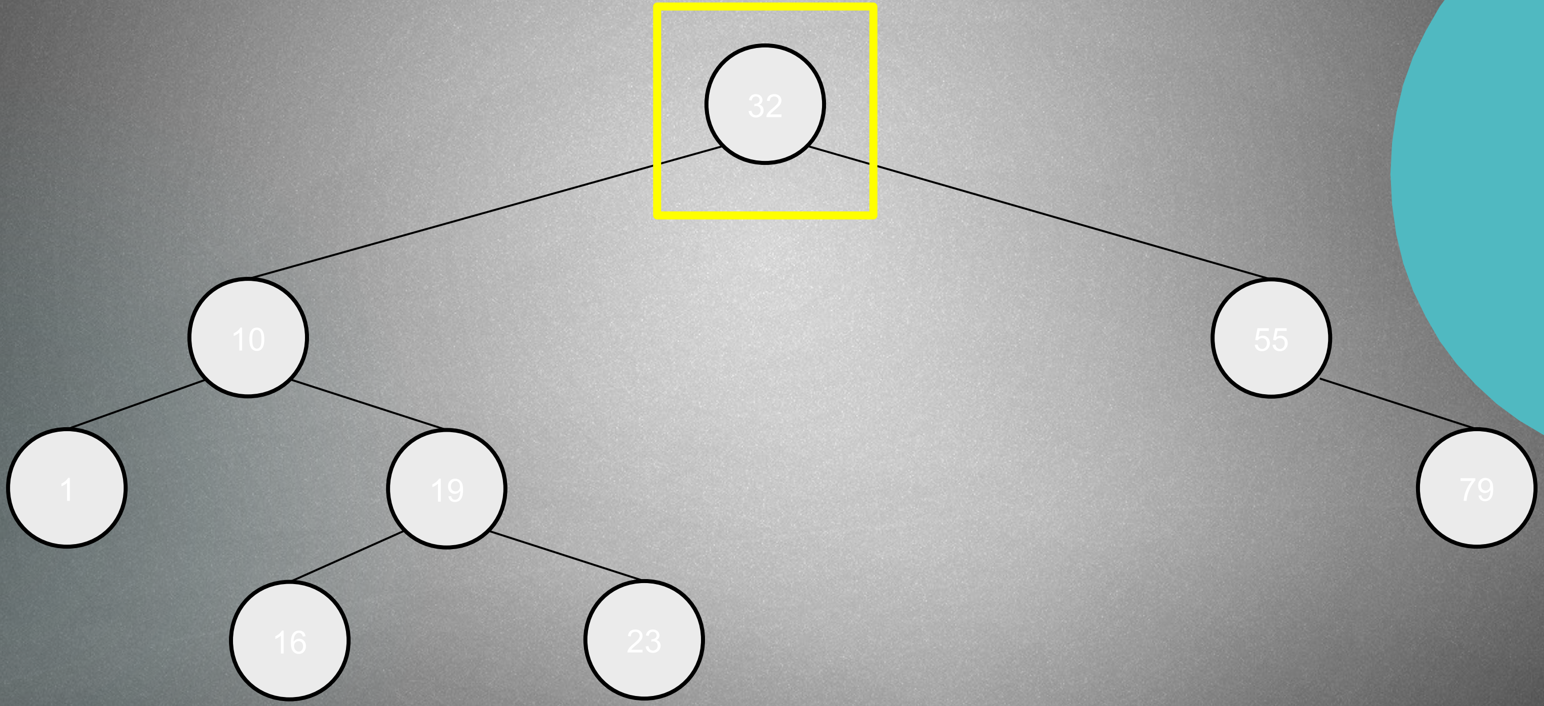
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



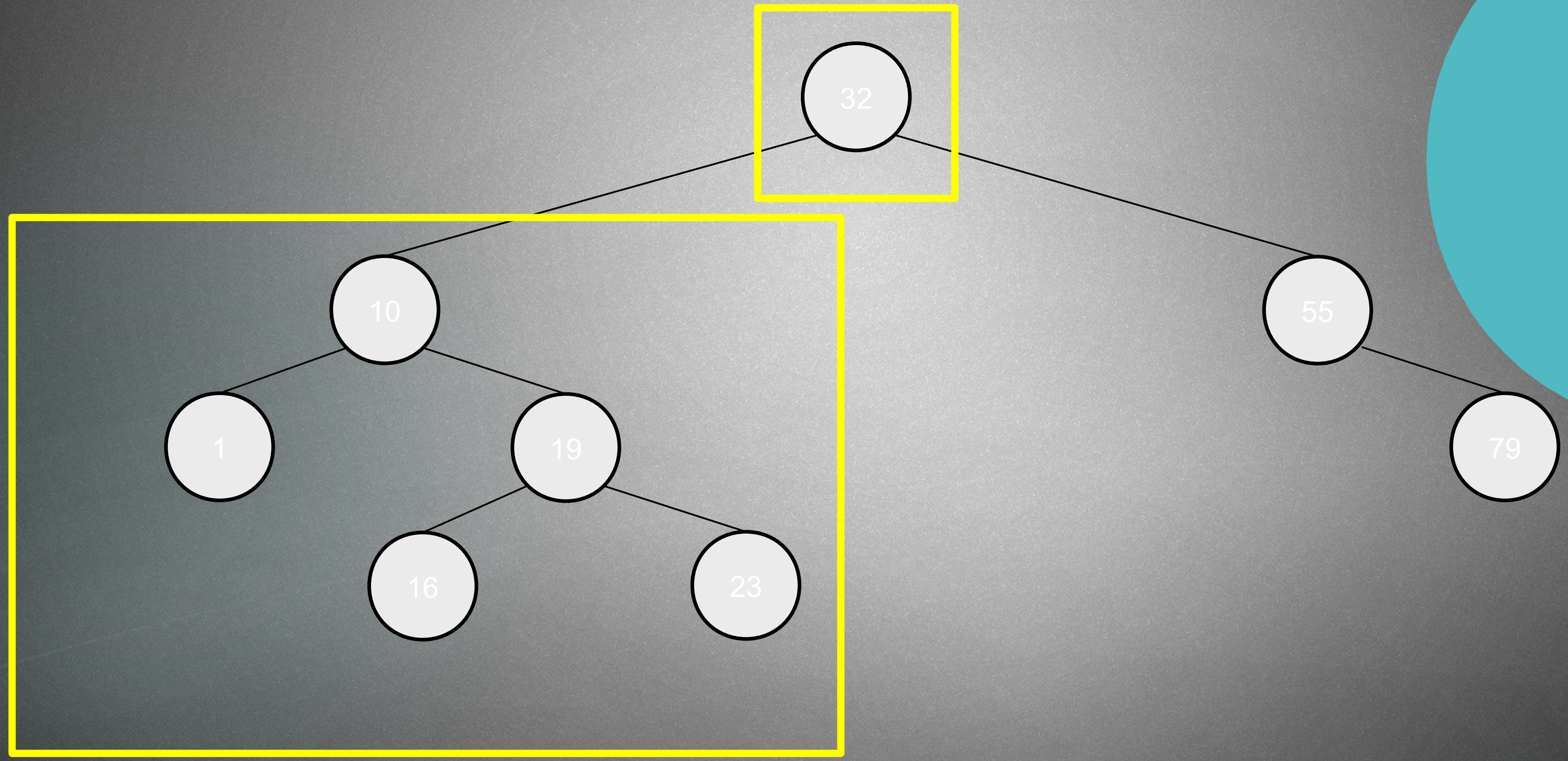
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



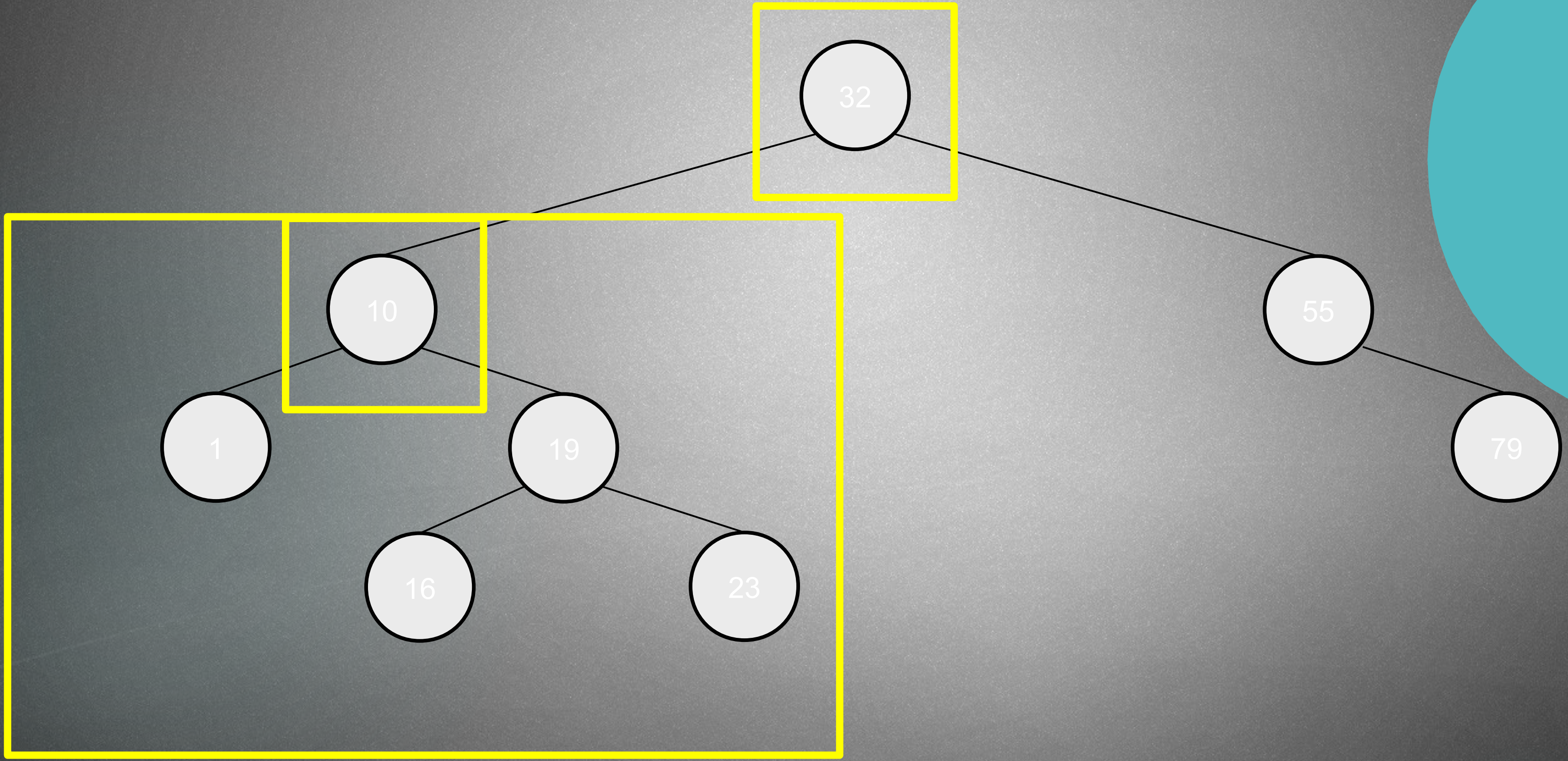
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

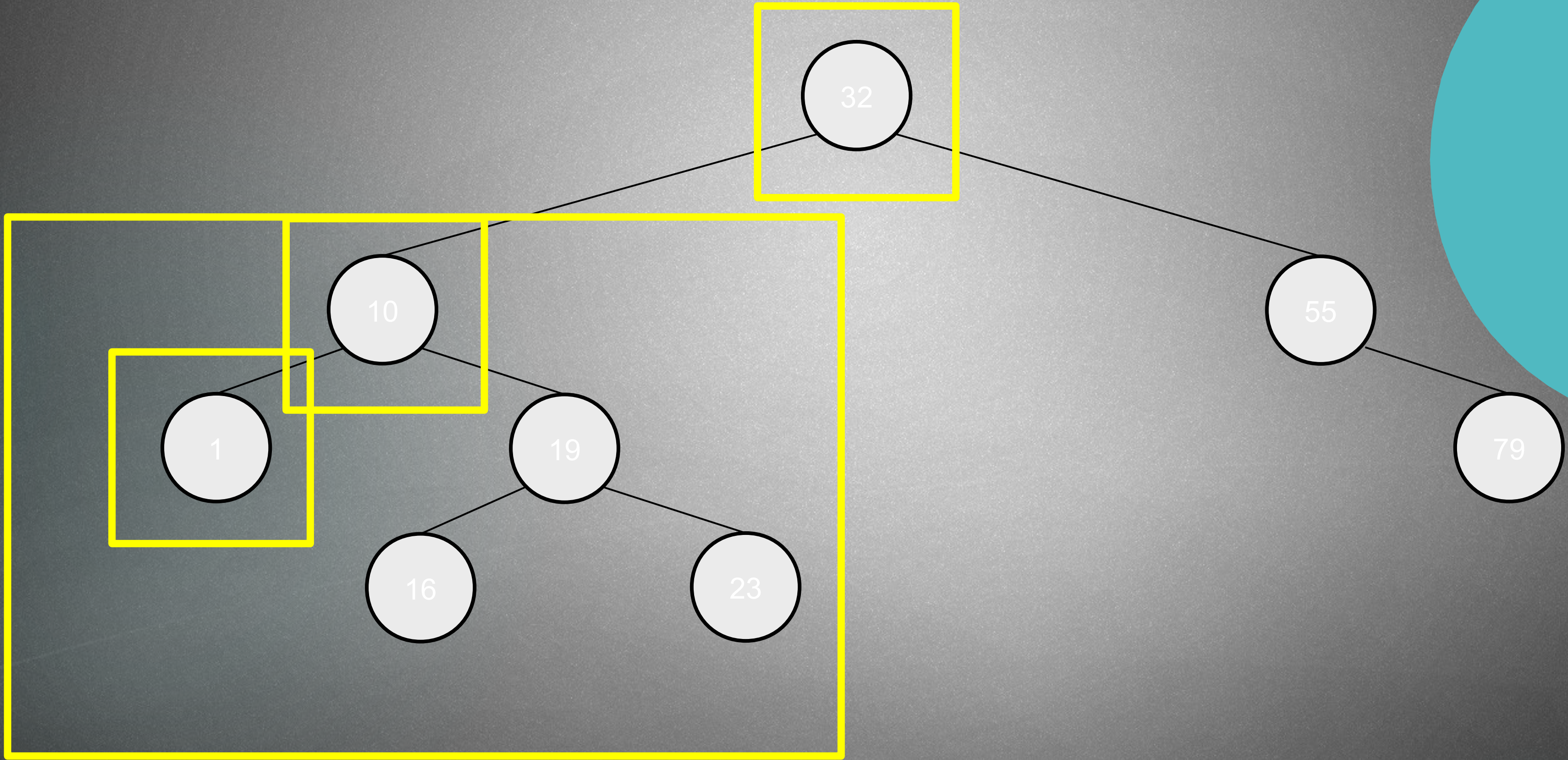
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree

We can do it several ways

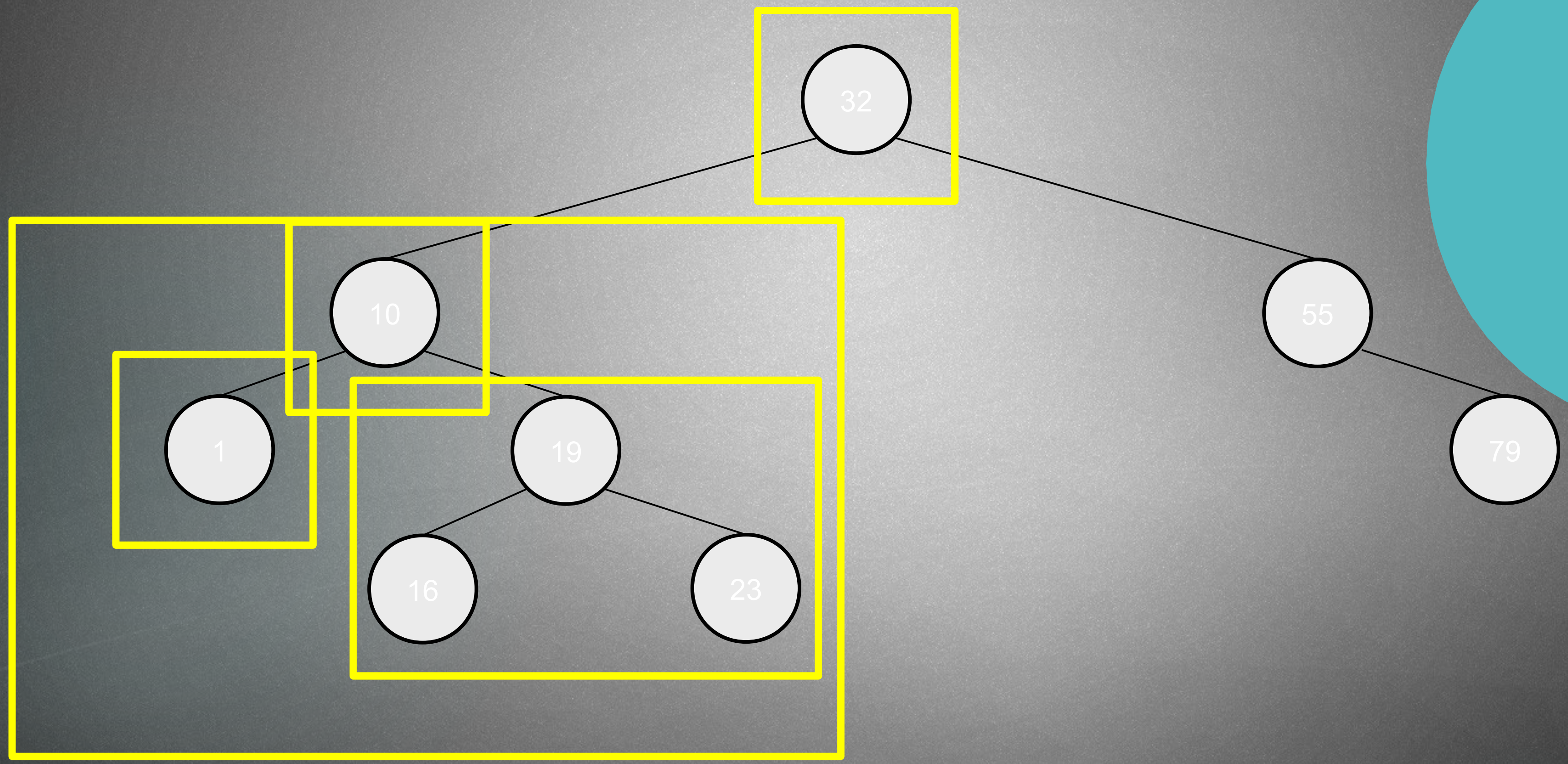
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree

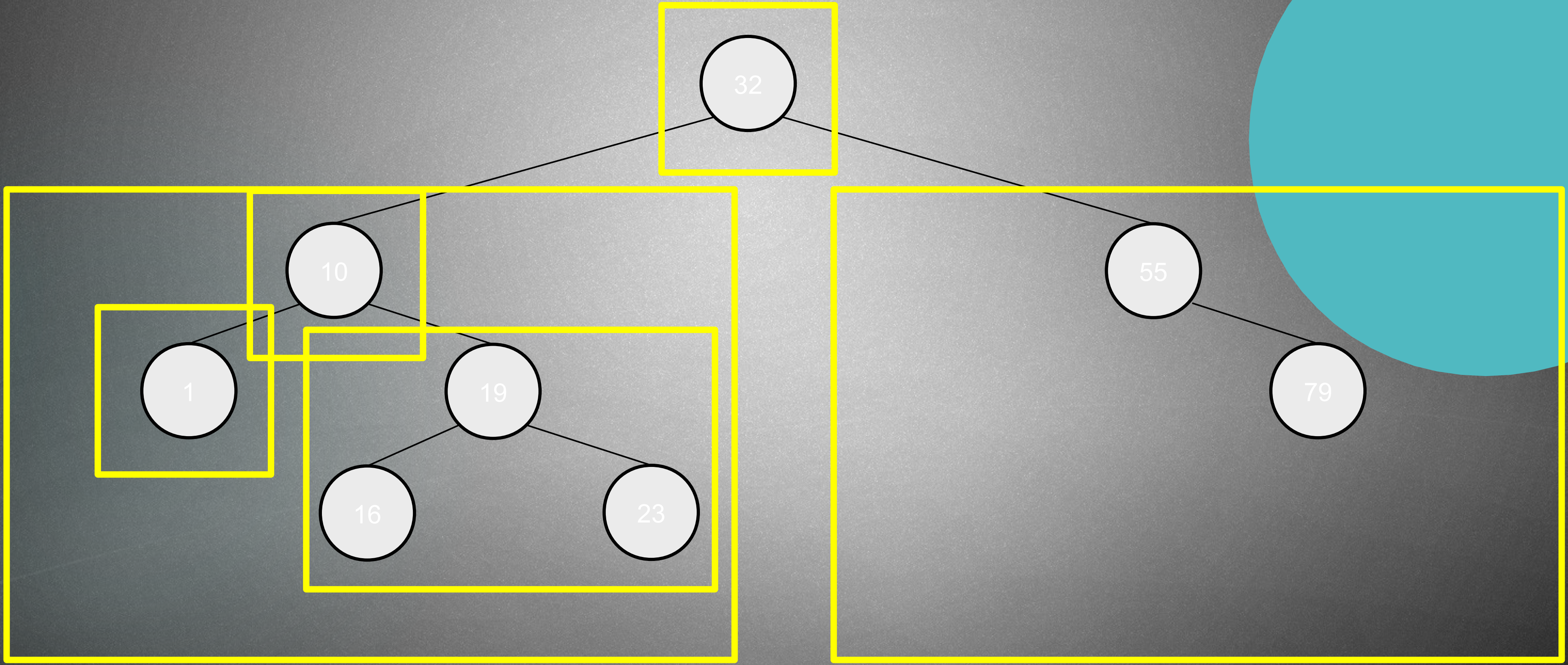
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



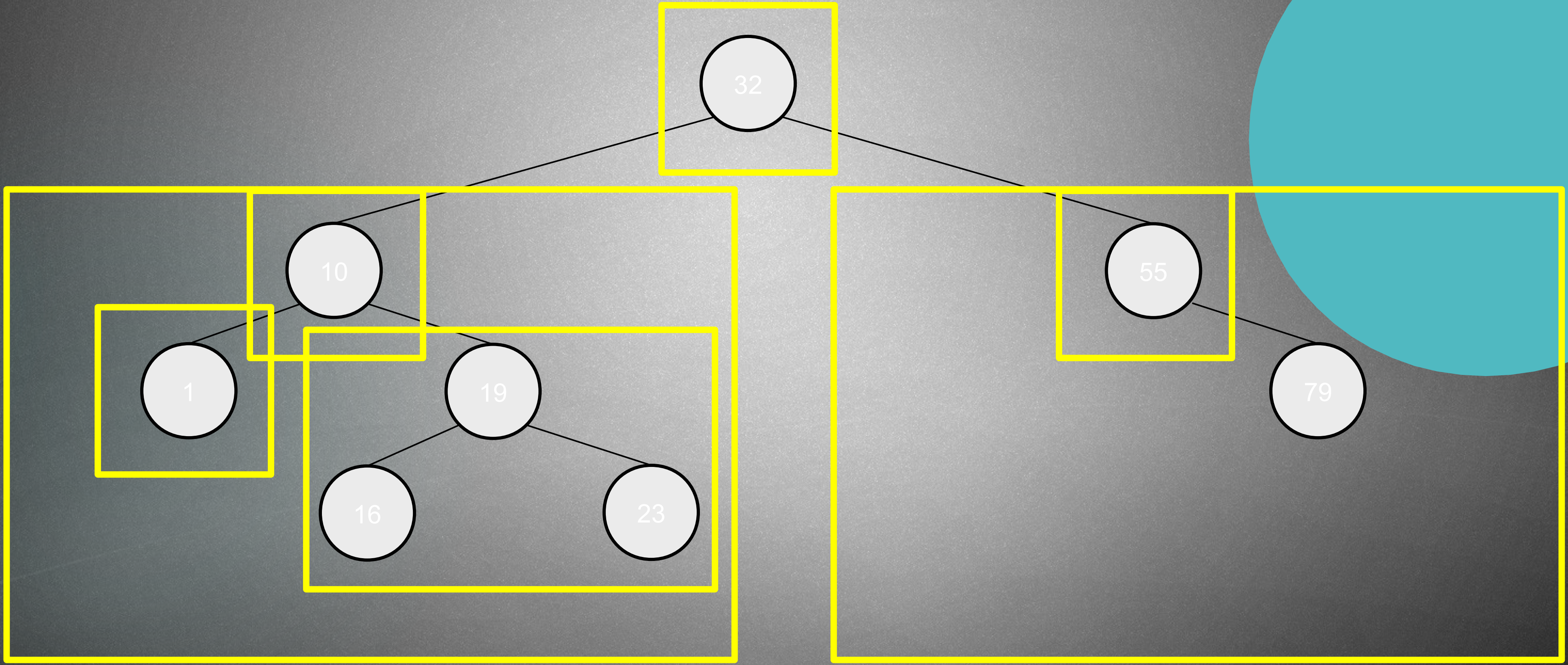
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



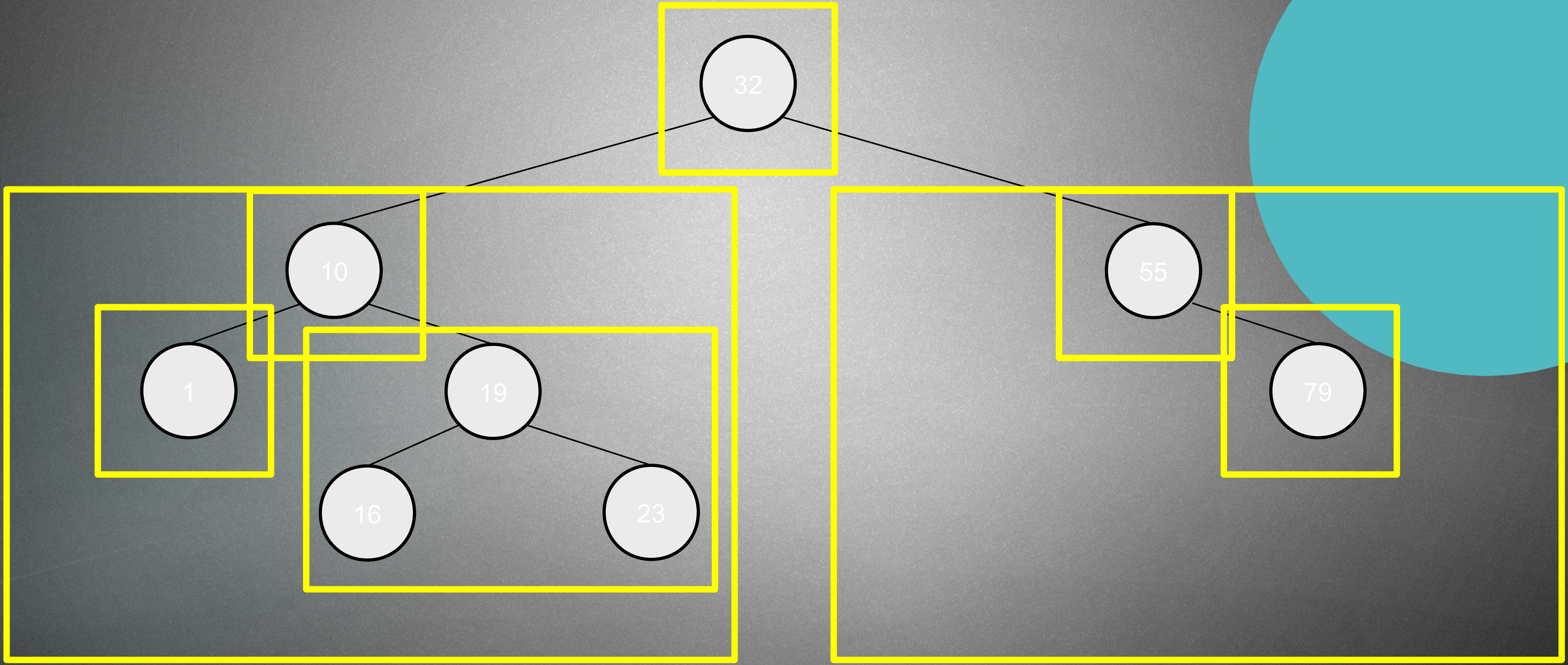
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree +
the right subtree recursively !!!



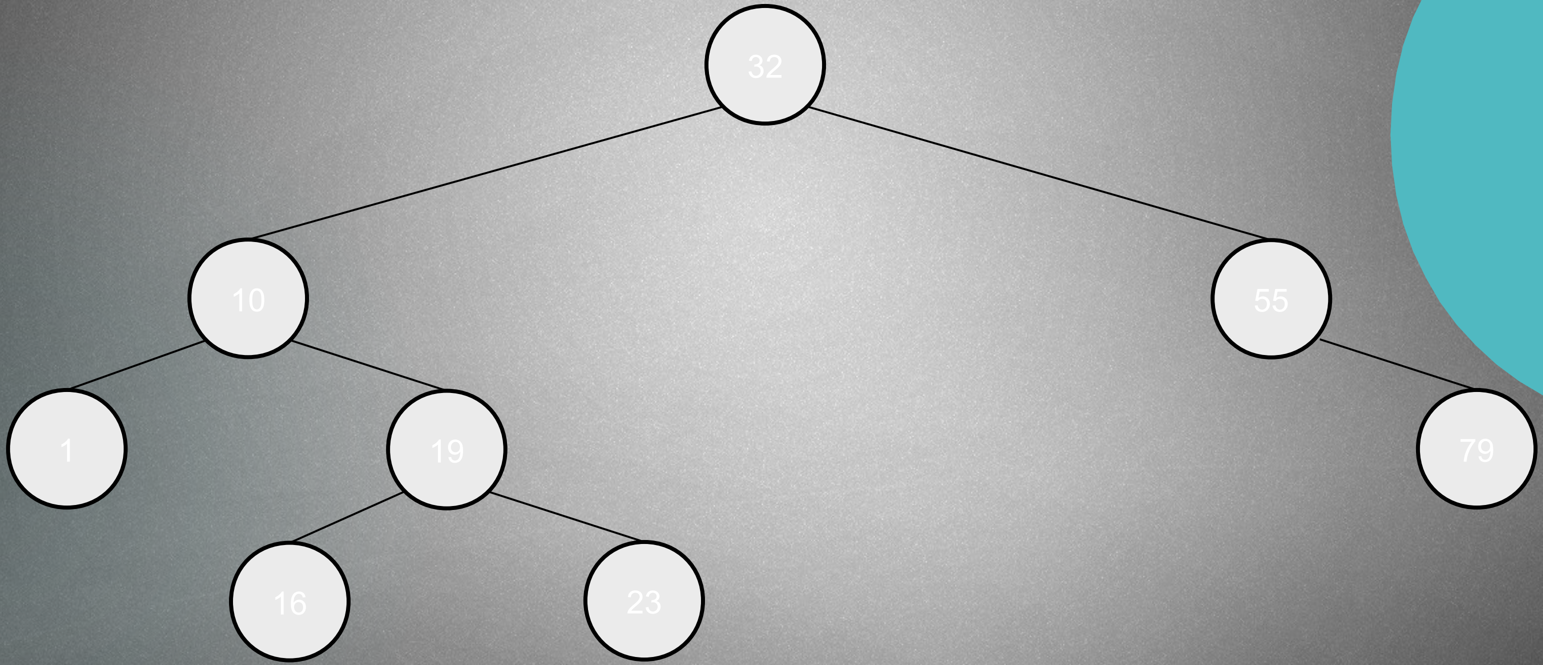
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

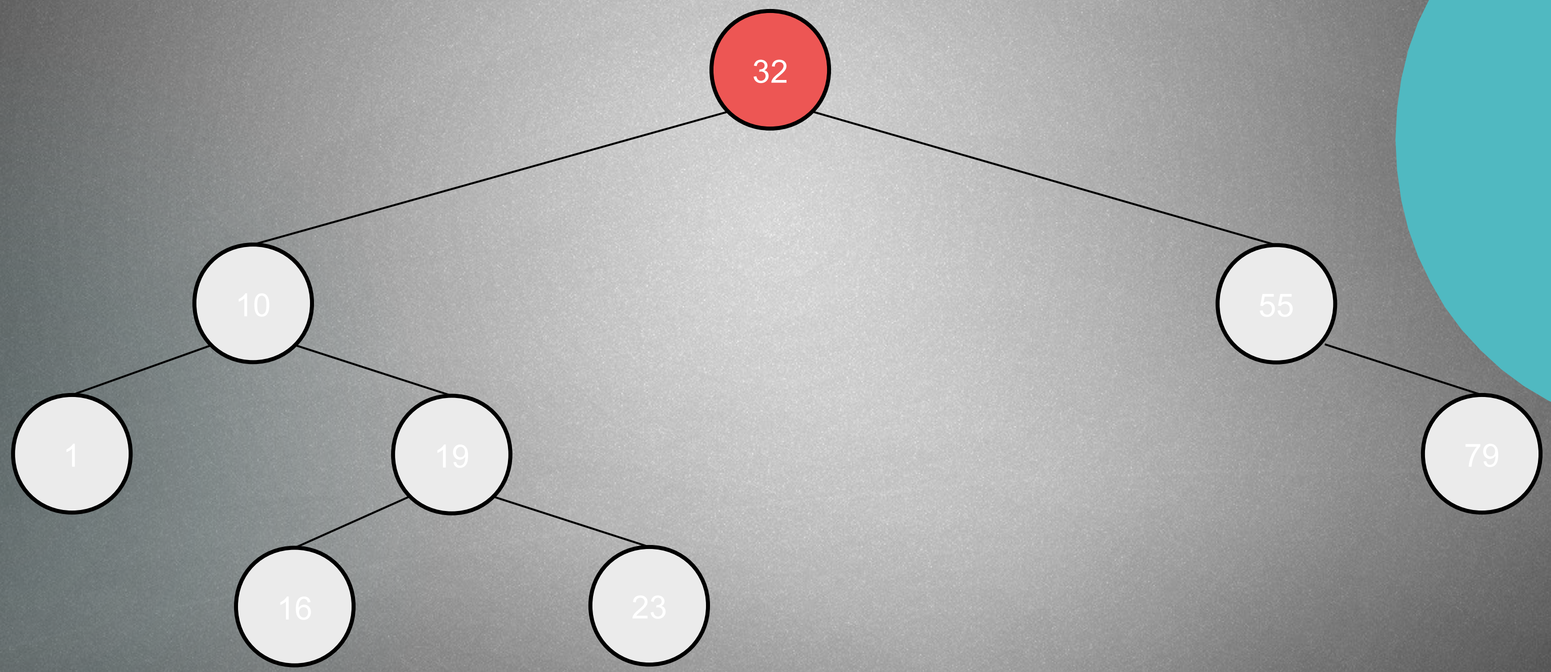
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal:

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!

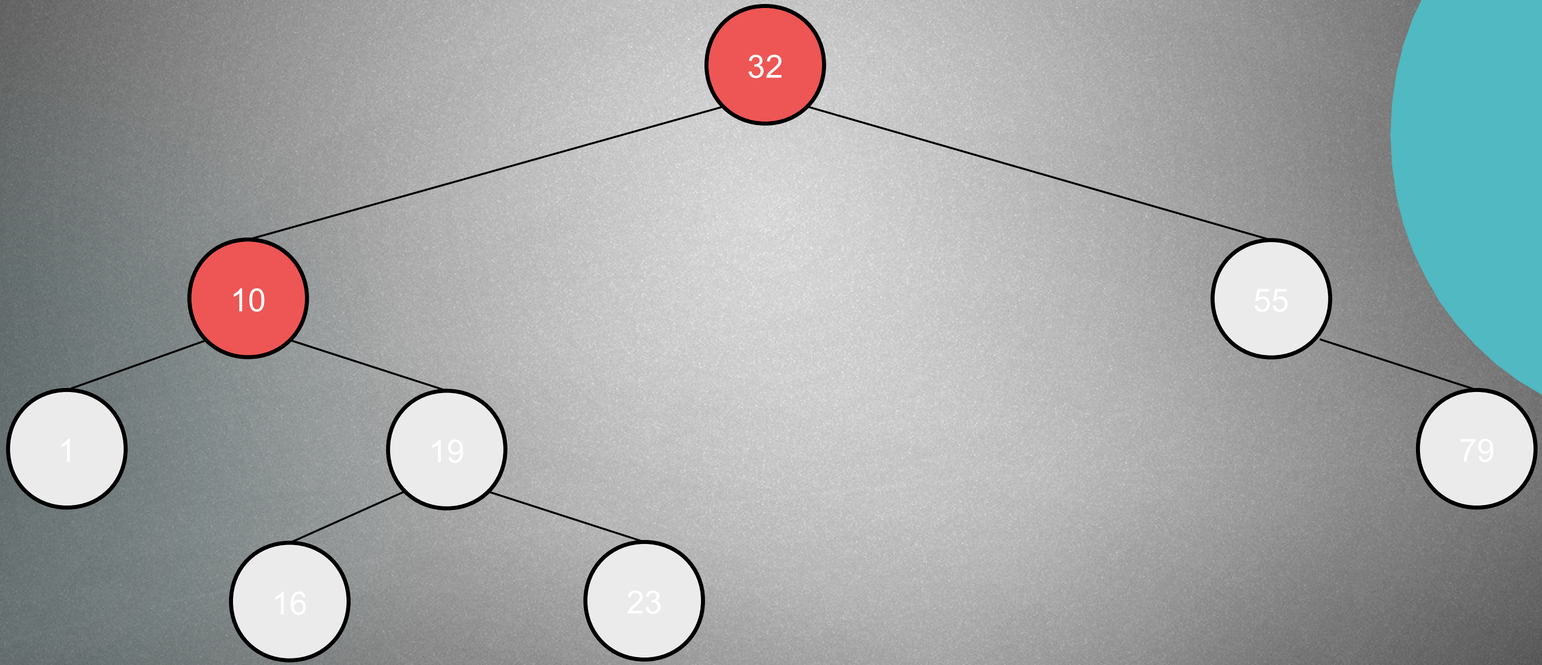


Pre-order traversal: 32

Traversal: sometimes it is neccessary to visit every node in the tree

We can do it several ways

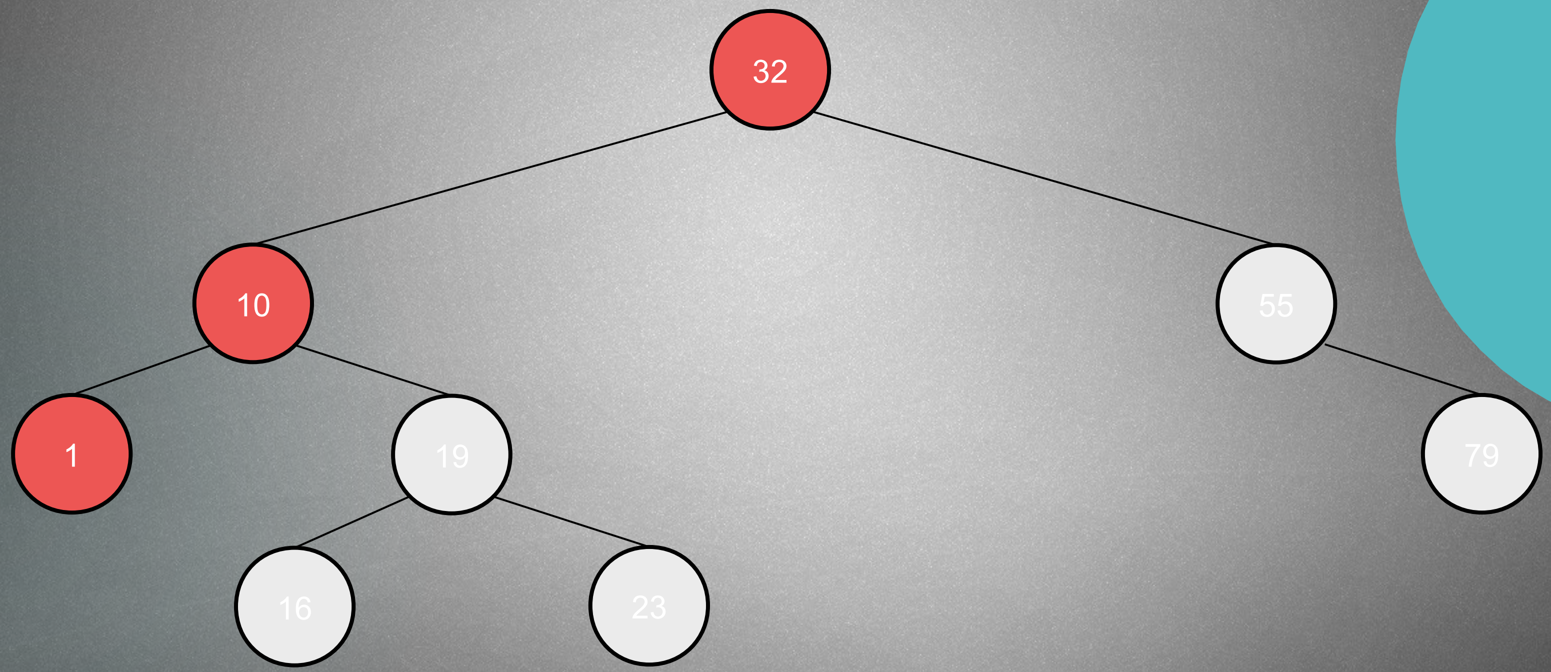
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

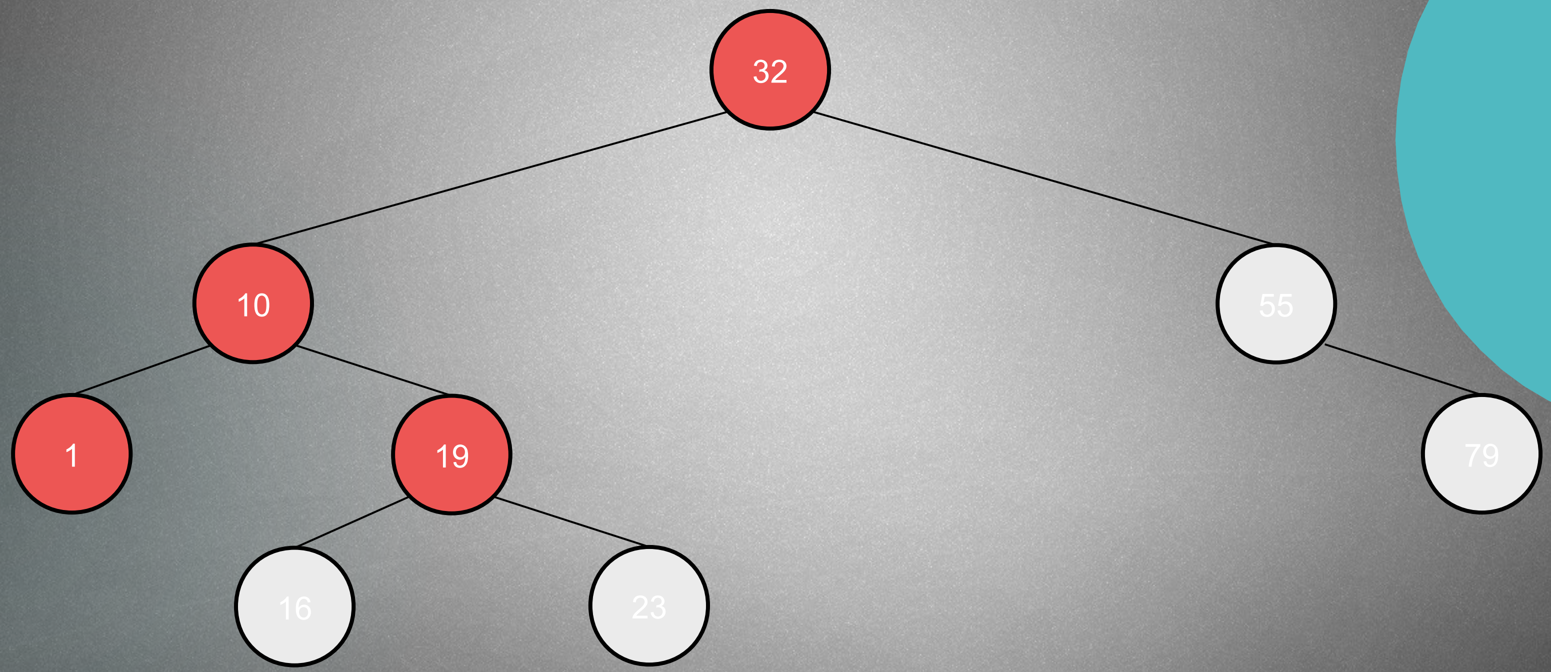
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

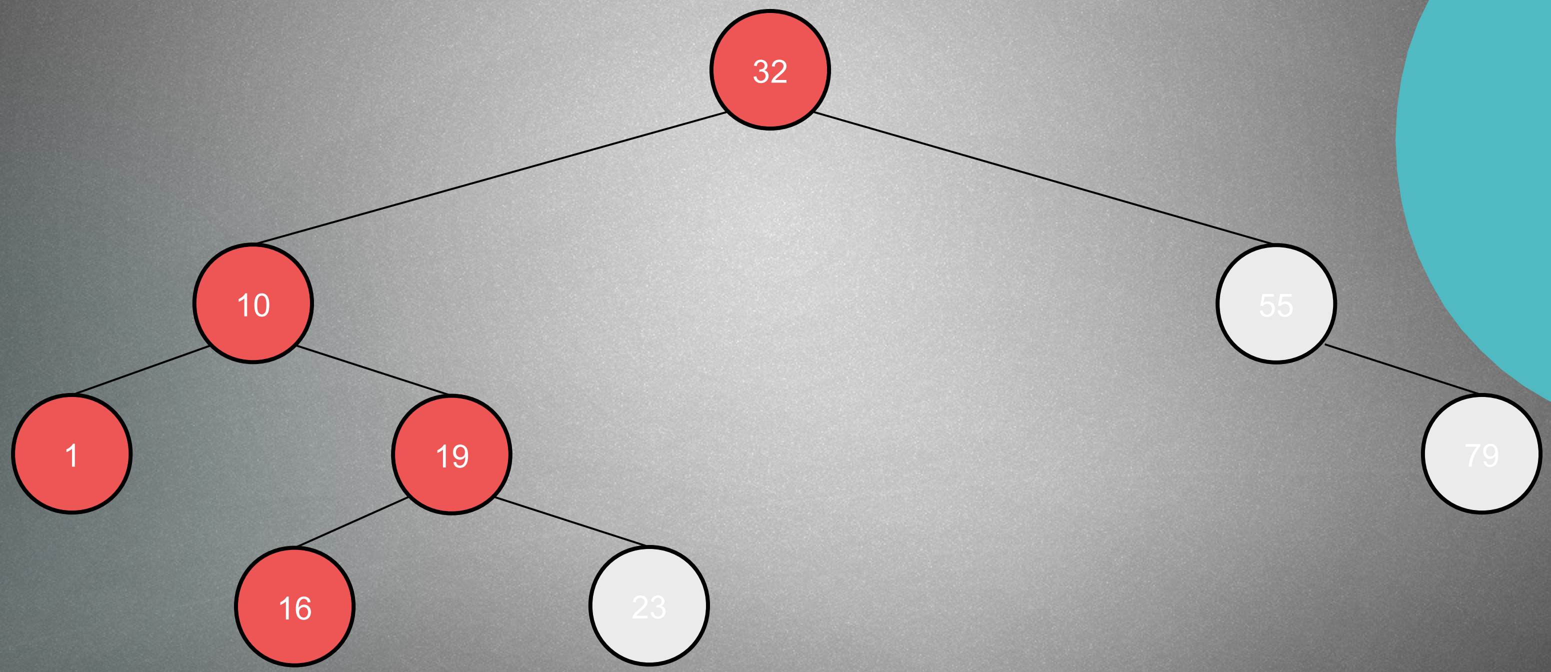
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1 – 19

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

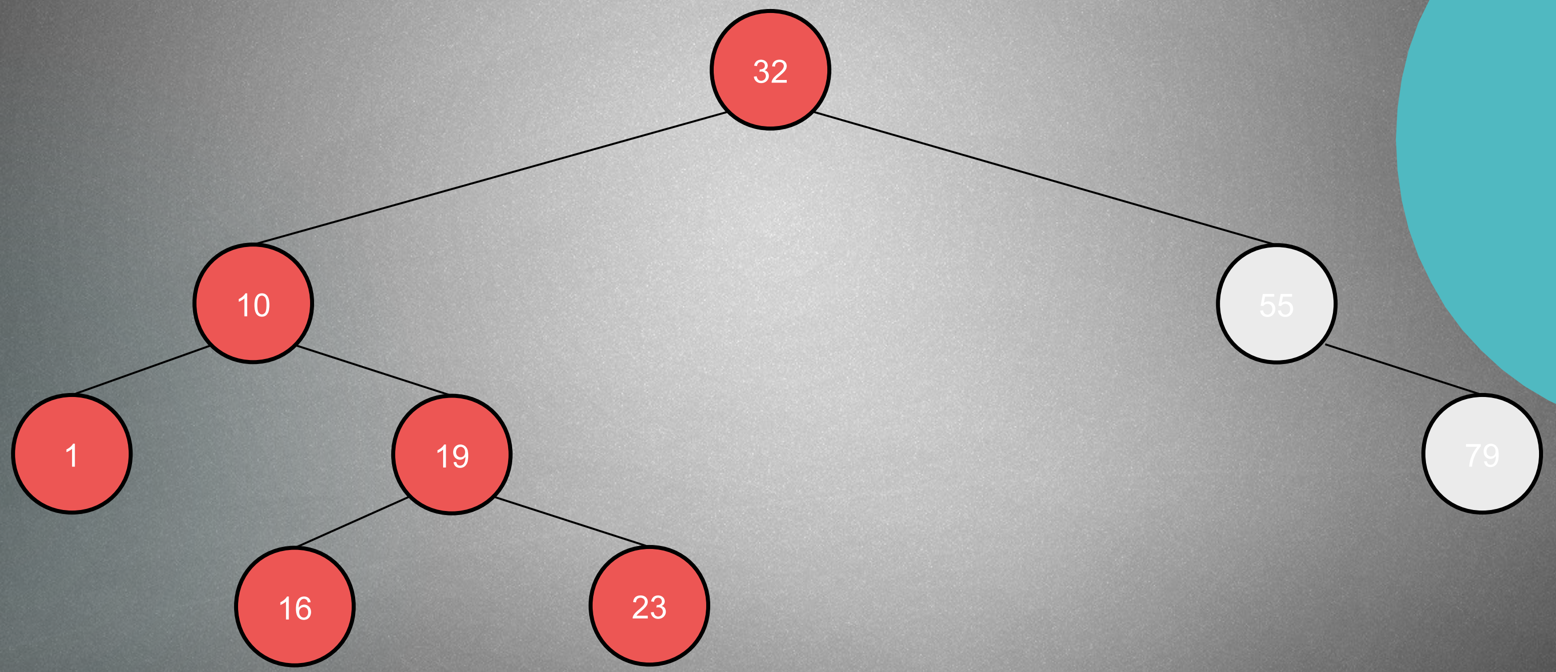
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1 – 19 – 16

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

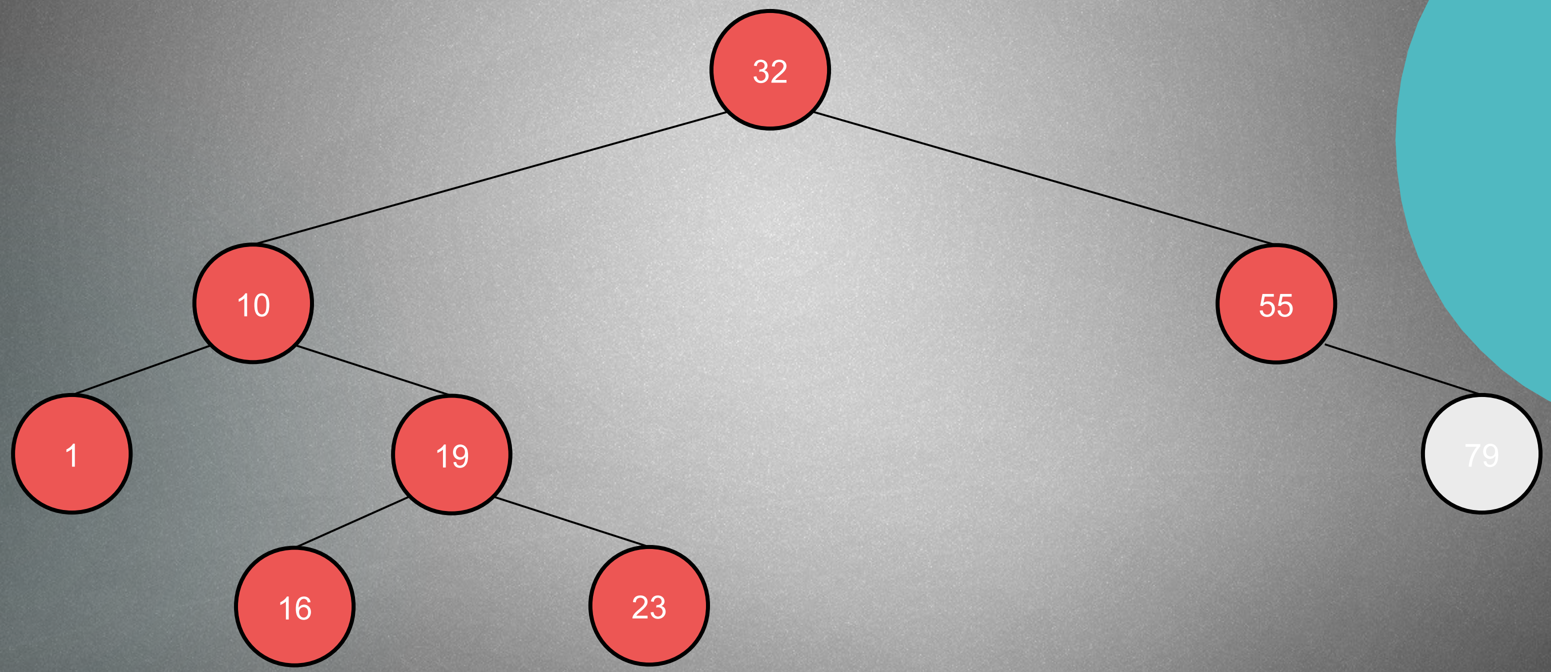
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1 – 19 – 16 – 23

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

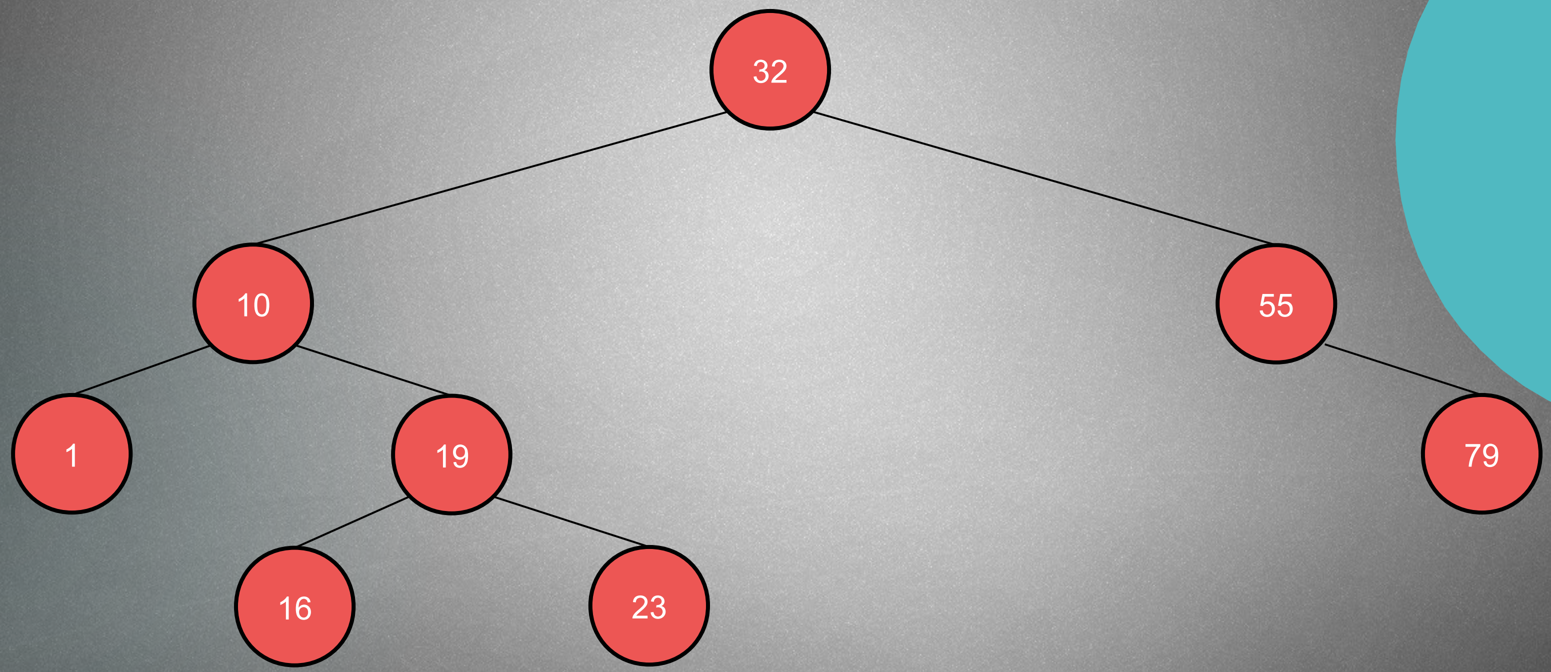
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1 – 19 – 16 – 23 – 55

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

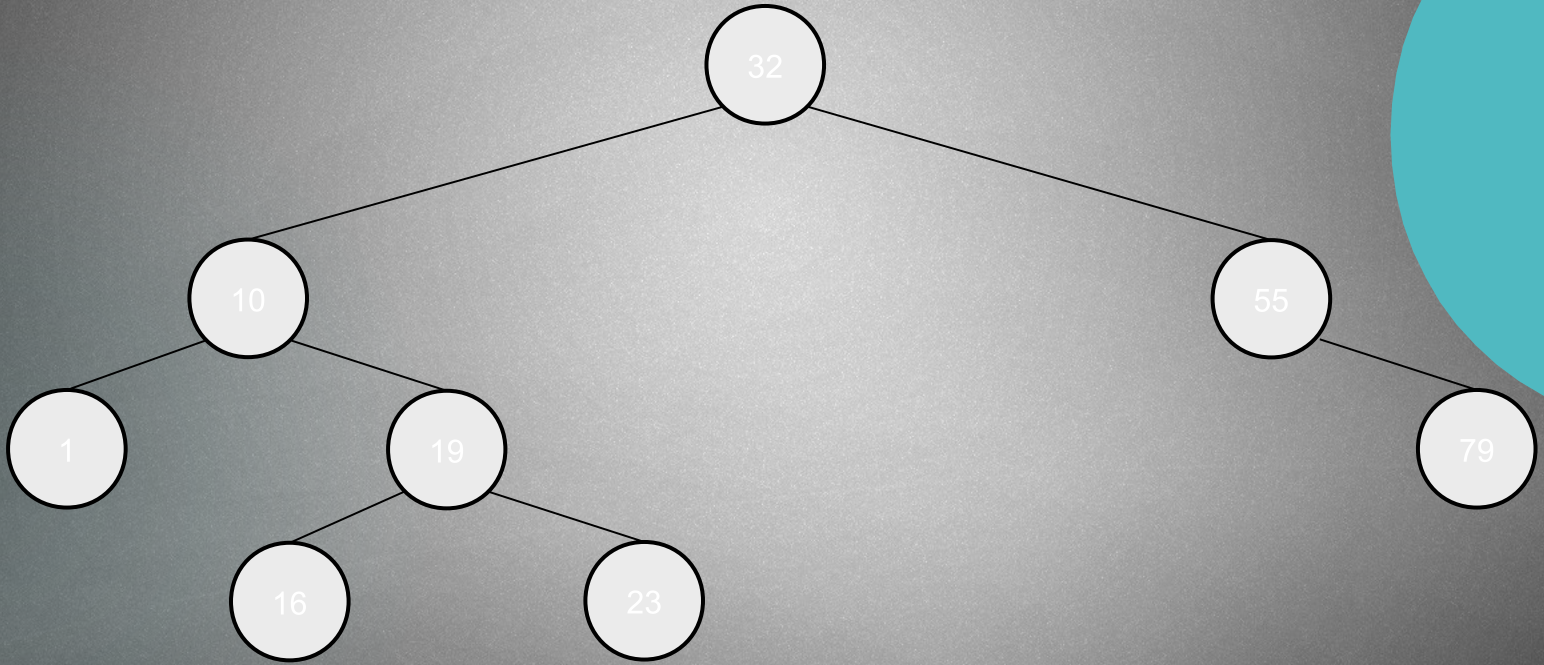
2.) Pre-order traversal: we visit the root+ left subtree + the right subtree recursively !!!



Pre-order traversal: 32 – 10 – 1 – 19 – 16 – 23 – 55 – 79

Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

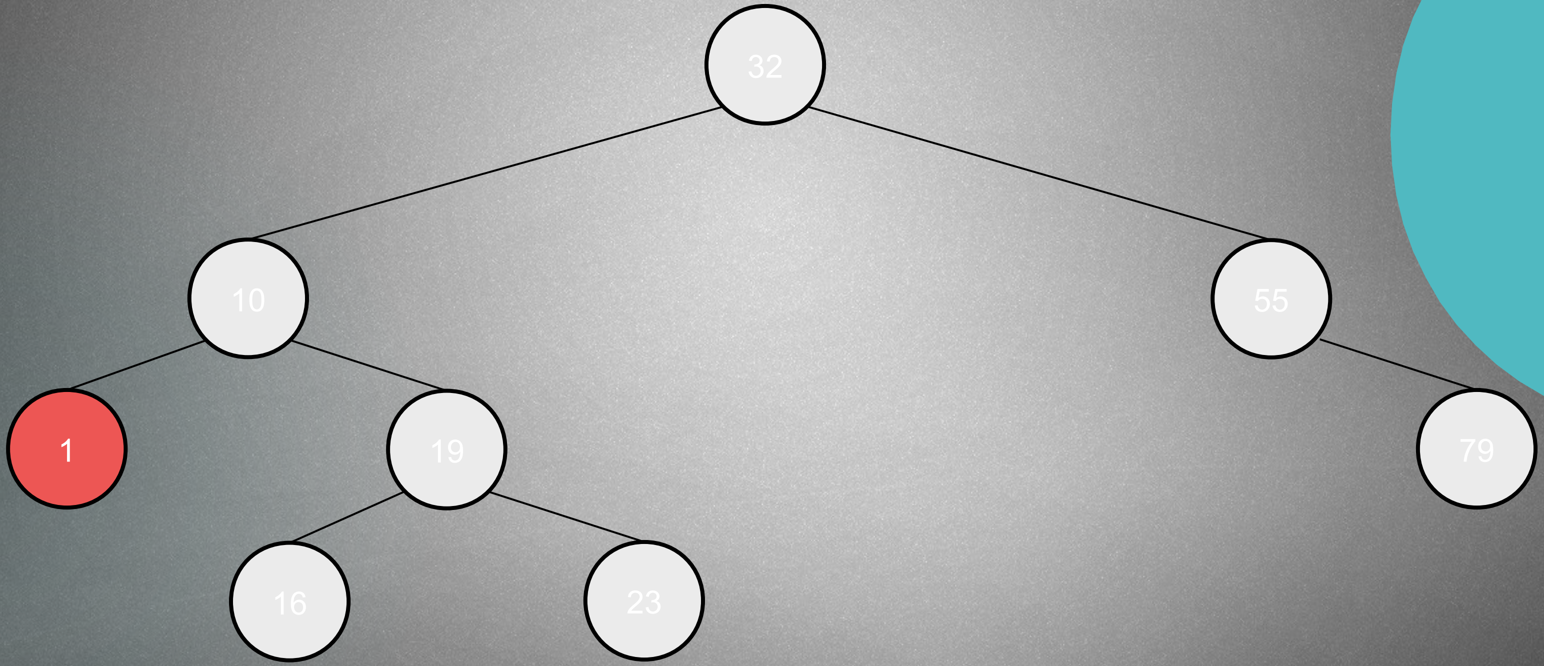
3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



Pre-order traversal:

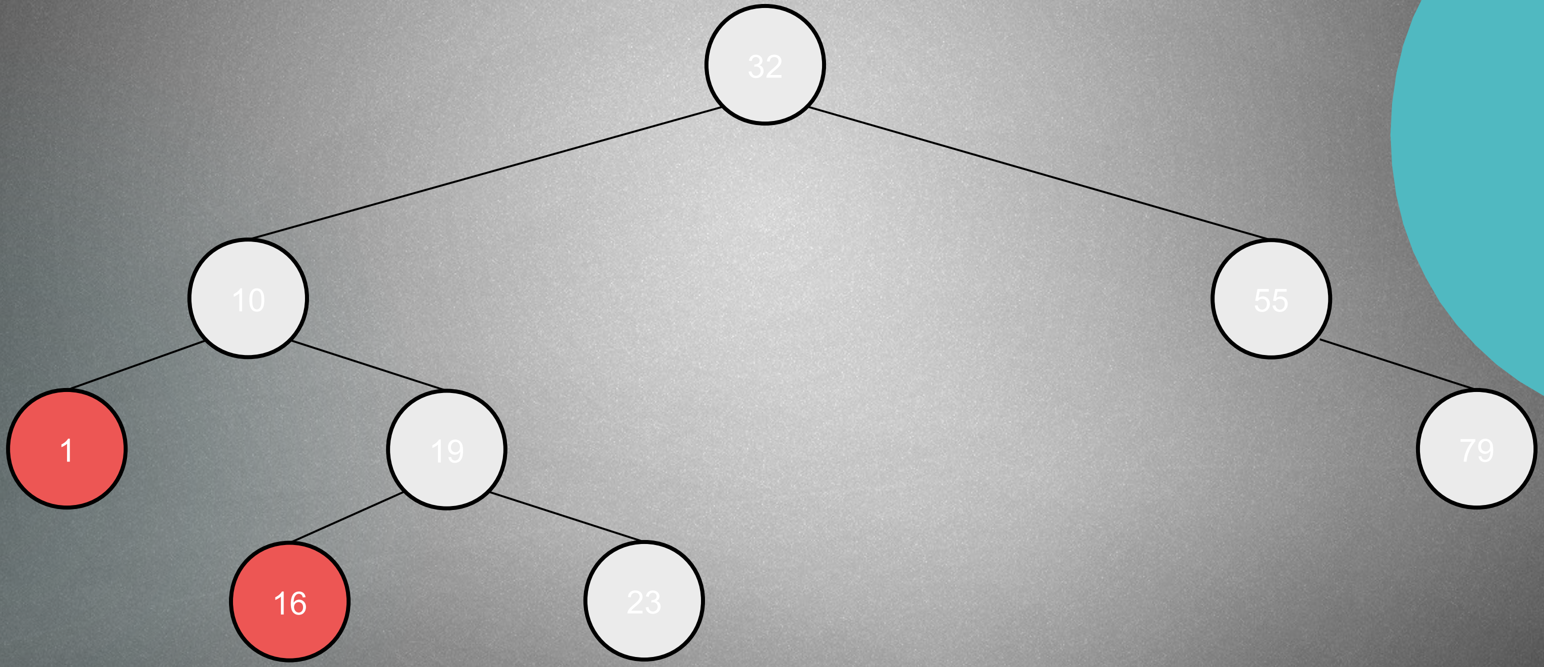
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



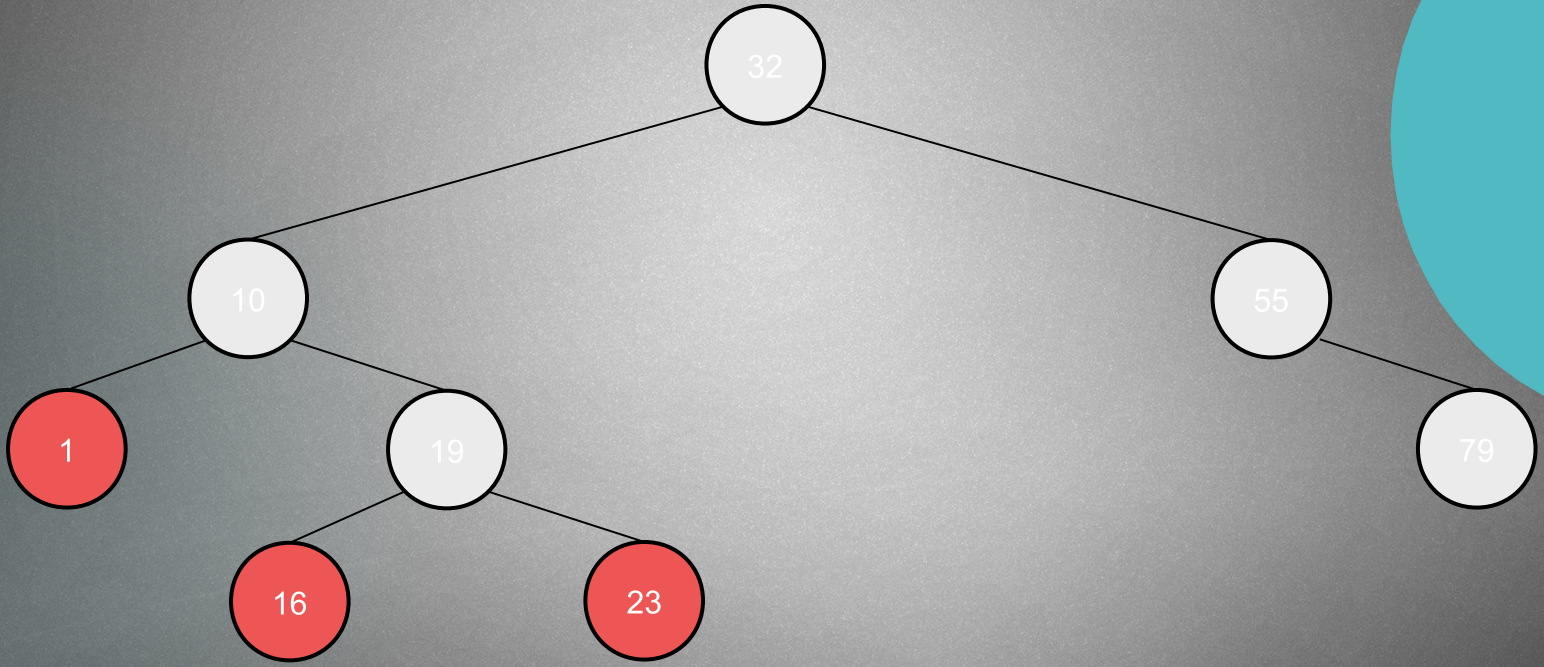
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



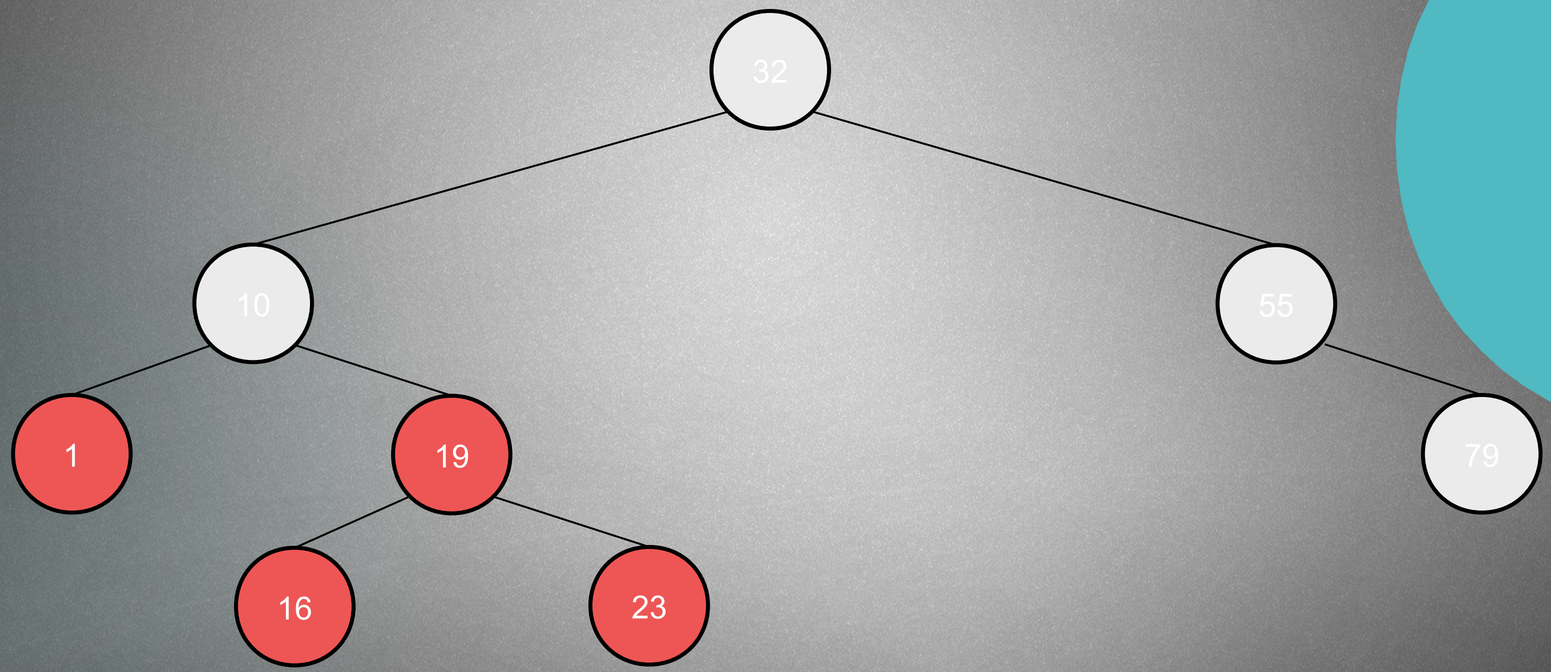
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



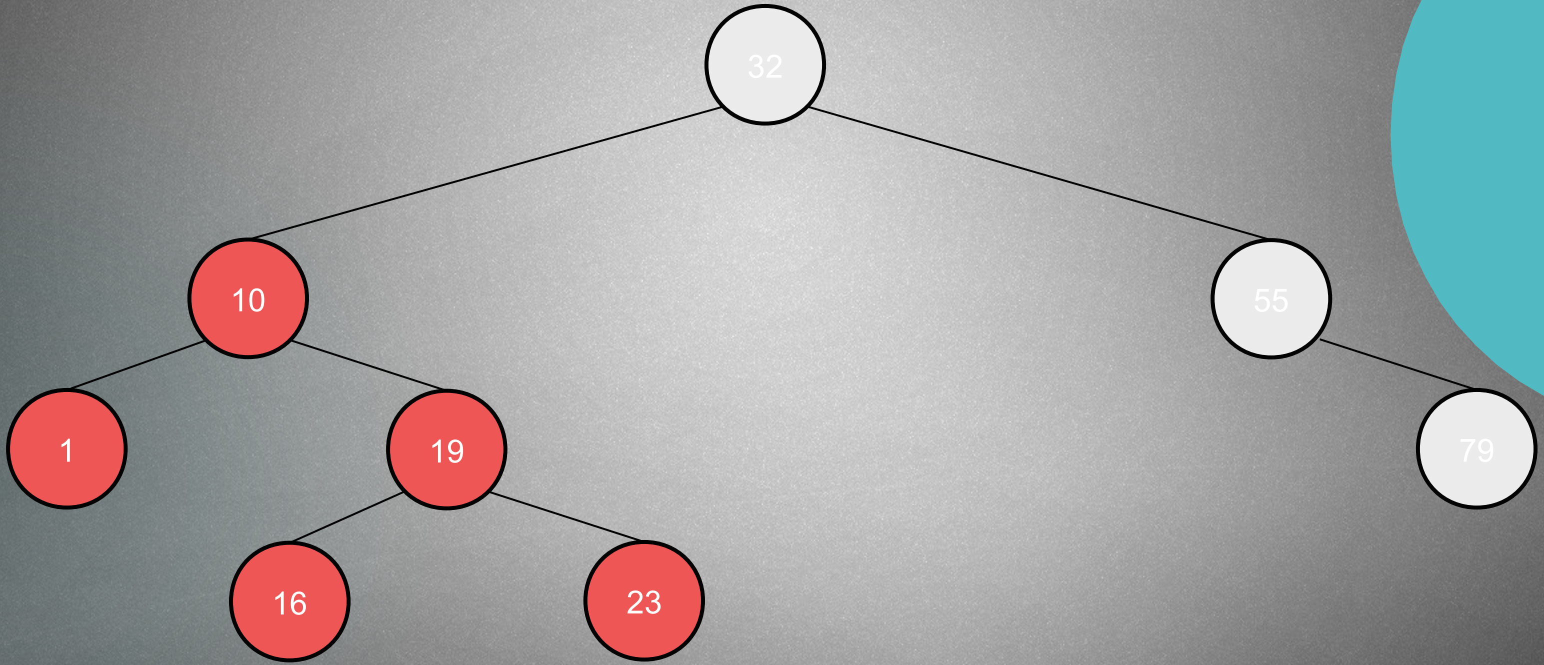
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



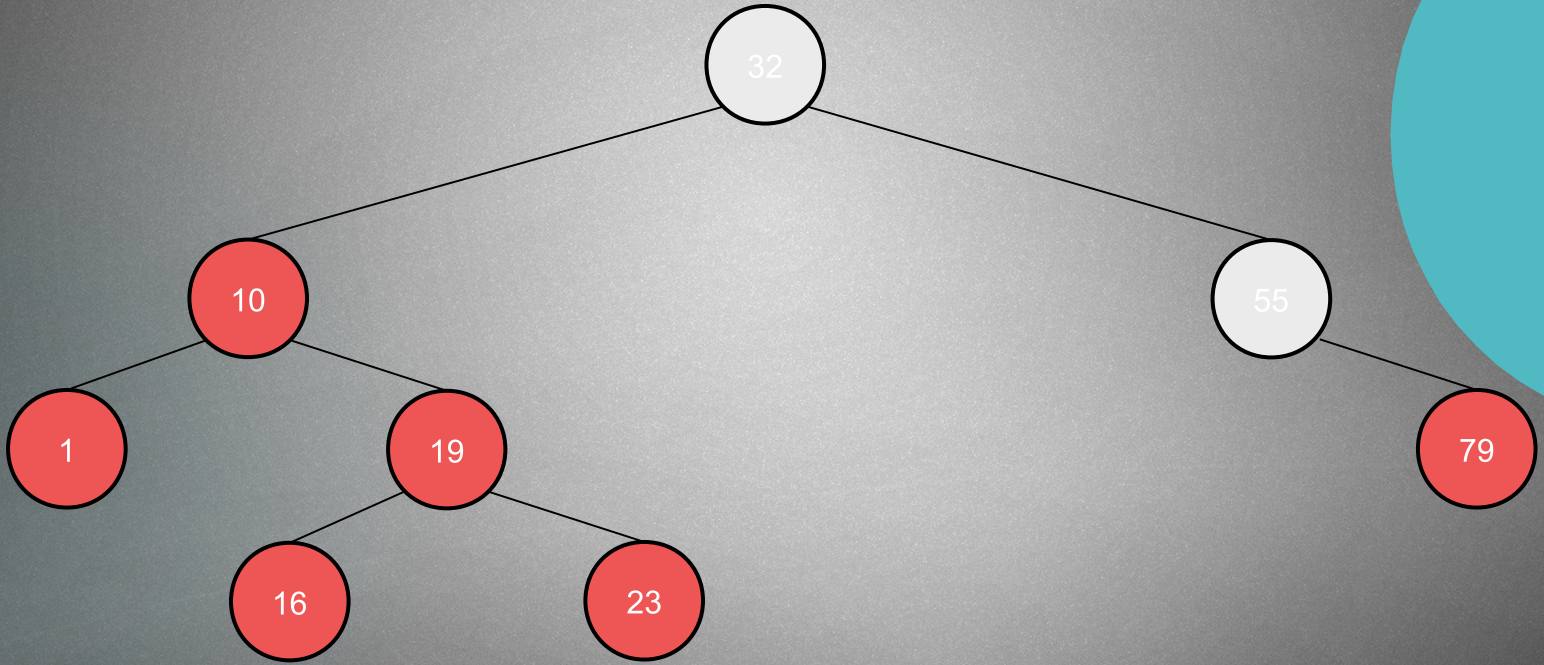
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



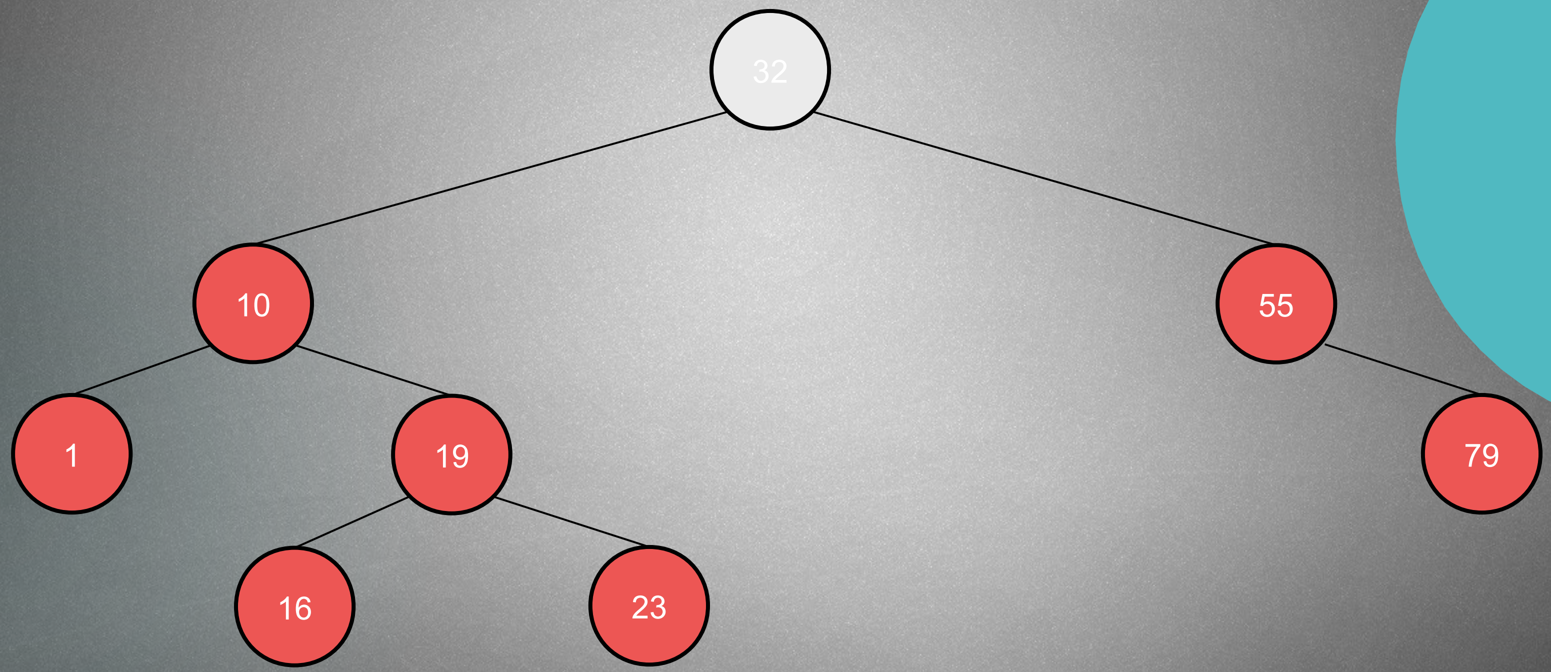
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



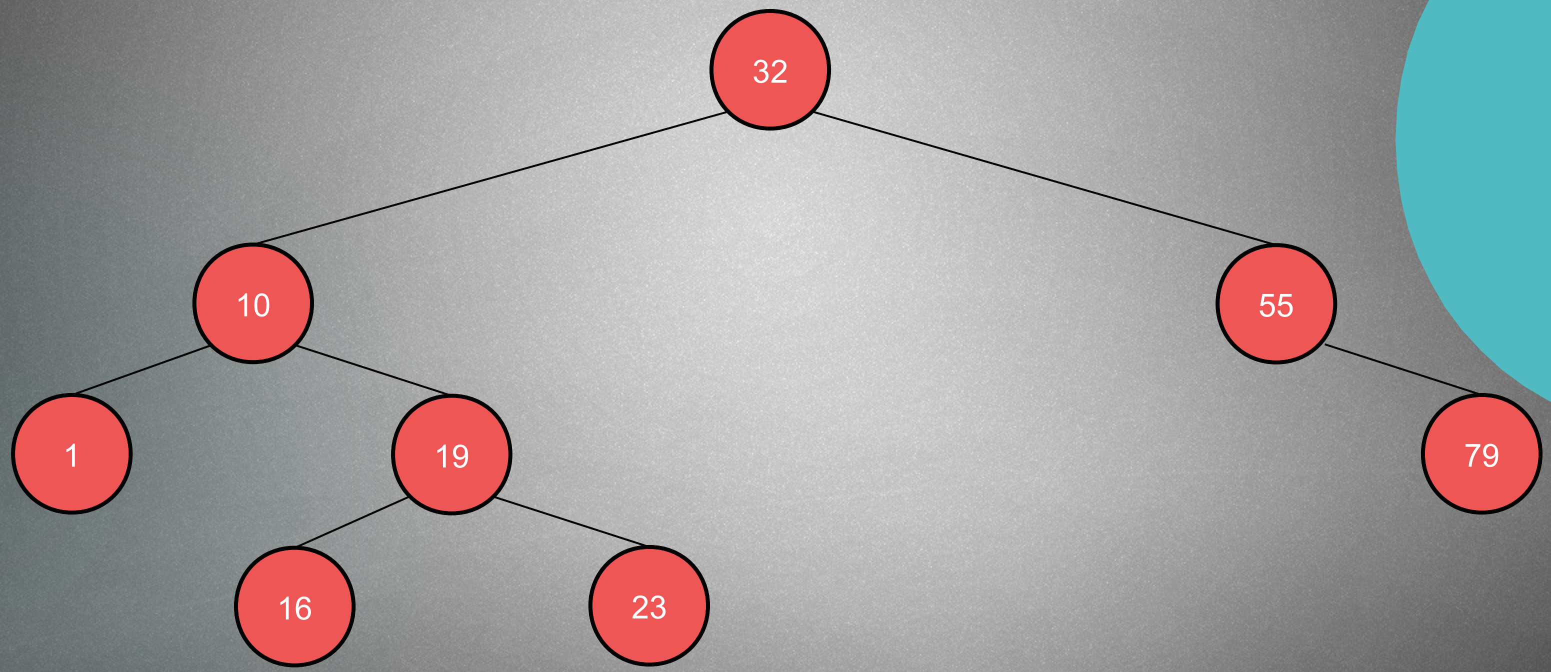
Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



Traversal: sometimes it is neccessary to visit every node in the tree
We can do it several ways

3.) Post-order traversal: we visit the left subtree+ right subtree + the root recursively !!!



	Average case	Worst case
Space	O(n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

- What about the worst case scenarios?
- - if the tree becomes unbalanced: the operations running times can be
- reduced to **O(N)** in the worst case
- - thats why it is important to keep a tree as balanced as possible