CS2040S
Data Structures and Algorithms

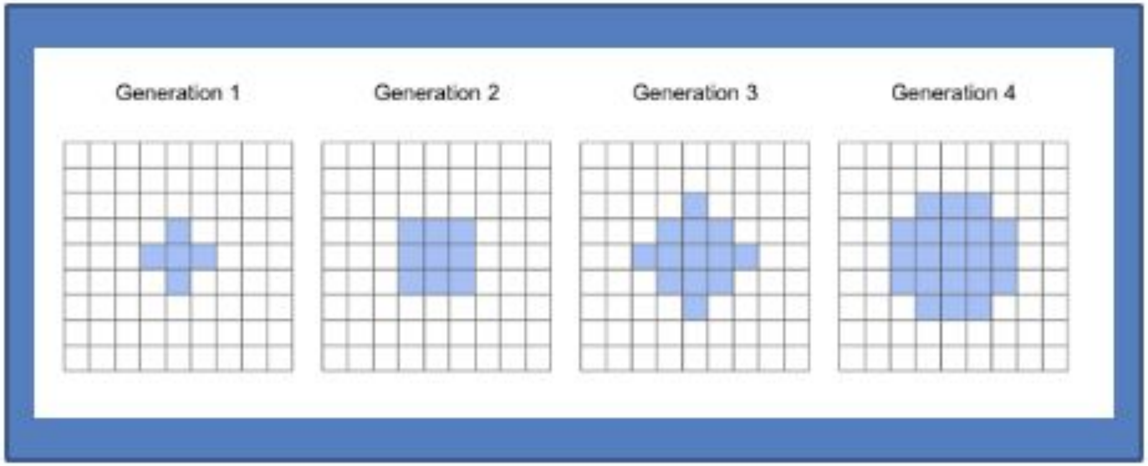
(Courtesy: Riddler)

Puzzle of the Week: Squares

Start with five shaded squares, infinite grid.

At every iteration, color a square if *at least* three neighboring were colored in the previous iteration.

As N gets large, how many squares will be shaded in generation N (as a function of N)?



Housekeeping:

Problem Set 4 Release:

- Wednesday Release 12-Feb
  - Duration: 1 week
  - Will be on trees

Plan:

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations

Plan:

Trees

- Terminology
- Traversals
- Operations

New concept! A data structure!

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations

Dictionary Interface

A collection of (key, value) pairs:

interface	IDictionary	
void	insert(Key k, Value v)	*insert (k,v) into table*
Value	search(Key k)	*get value paired with k*
Key	successor(Key k)	*find next key > k*
Key	predecessor(Key k)	*find next key < k*
void	delete(Key k)	*remove key k (and value)*
boolean	contains(Key k)	*is there a value for k?*
int	size()	*number of (k,v) pairs*

Dictionary Interface

A collection of (key, value) pairs:

Assumes that the keys can be totally ordered!

interface	**IDic**	
void	inse	le
Value	sear	k
Key	successor(Key k)	find next key > k
Key	predecessor(Key k)	*find next key < k*
void	delete(Key k)	*remove key k (and value)*
boolean	contains(Key k)	*is there a value for k?*
int	size()	*number of (k,v) pairs*

Dictionary

Implementation

Option 1: Sorted array

- insert : ?
- search : ?

Option 2: Unsorted array

- insert : ?
- search : ?

Option 3: Linked list

- insert : ?
- search : ?

Dictionary

Implementation

Option 1: Sorted array

- insert : add to middle of array = ??
- search : binary search = ??

Option 2: Unsorted array

- insert : add to end of array = ??
- search : unsorted = ??

Option 3: Linked list

- insert : add to head of list = ??
- search : list traversal = ??

Dictionary

Implementation

Option 1: Sorted array

- insert : add to middle of array = O(n)
- search : binary search = O(log n)

Option 2: Unsorted array

- insert : add to end of array = ??
- search : unsorted = ??

Option 3: Linked list

- insert : add to head of list = ??
- search : list traversal = ??

Dictionary

Implementation

Option 1: Sorted array

- insert : add to middle of array = O(n)
- search : binary search = O(log n)

Option 2: Unsorted array

- insert : add to end of array = O(1)
- search : unsorted = O(n)

Option 3: Linked list

- insert : add to head of list = ??
- search : list traversal = ??

Dictionary

Implementation

Option 1: Sorted array

- insert : add to middle of array = O(n)
- search : binary search = O(log n)

Option 2: Unsorted array

- insert : add to end of array = O(1)
- search : unsorted = O(n)

Option 3: Linked list

- insert : add to head of list = O(1)
- search : list traversal = O(n)

Dictionary

Implementation

Option 1: Sorted array

- insert : add to middle of array = O(n)
- search : binary search = O(log n)

Option 2: Unsorted array

- insert : add to end of array = O(1)
- search : unsorted = O(n)

Option 3: Linked list

- insert : add to head of list = O(1)
- search : list traversal = O(n)

Notice here that all the operations seem to have something be in linear time.

Can we do better?

Dictionary Implementation

Possible Choices:

- Implement using an array
- Implement using a queue.
- Implement using a linked list
- …

Binary Search Trees

1. Terminology and Definitions

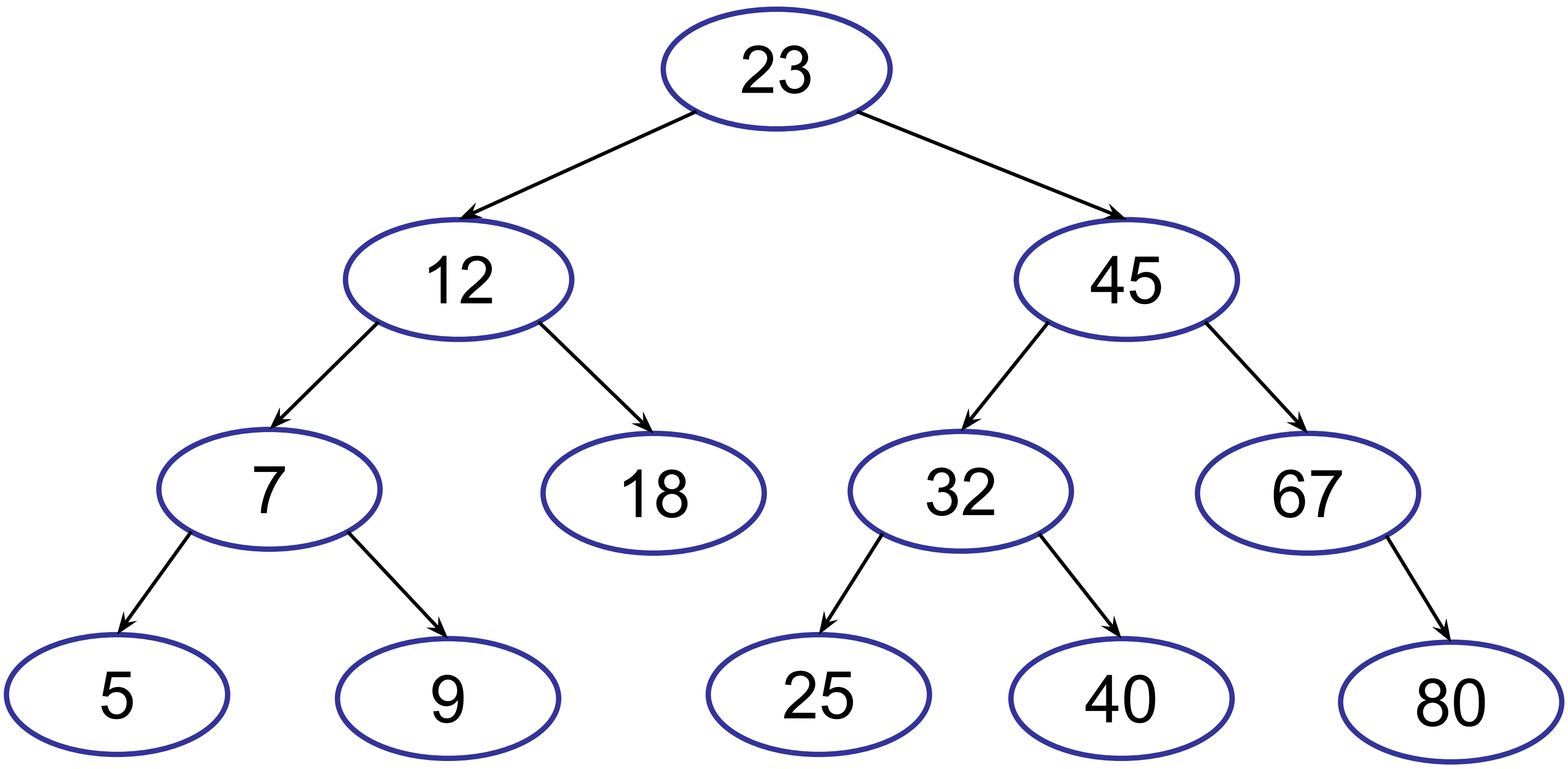
2. Basic operations:
   - height
   - search, insert
   - searchMin, searchMax

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

Dictionary

Implementation idea: Tree

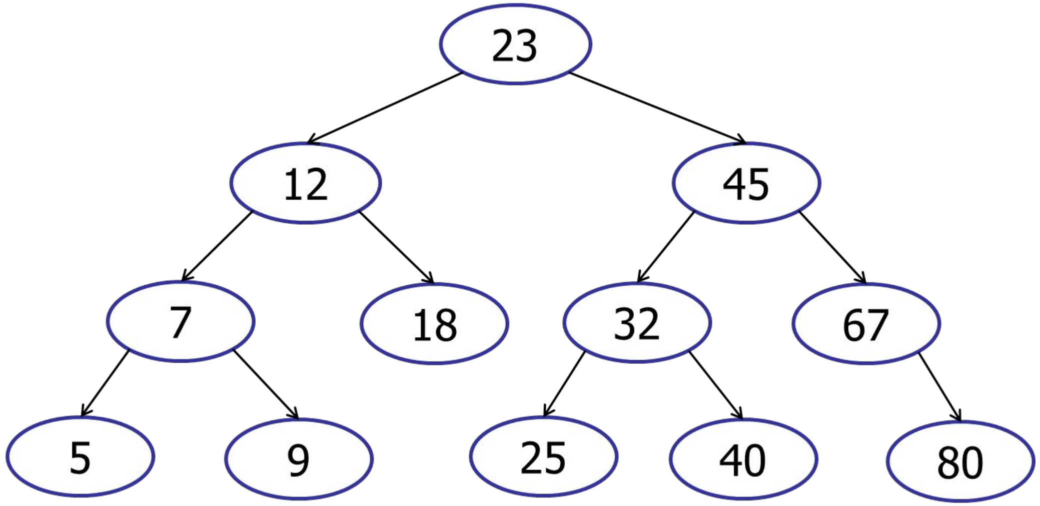


Dictionary

Implementation idea: Tree

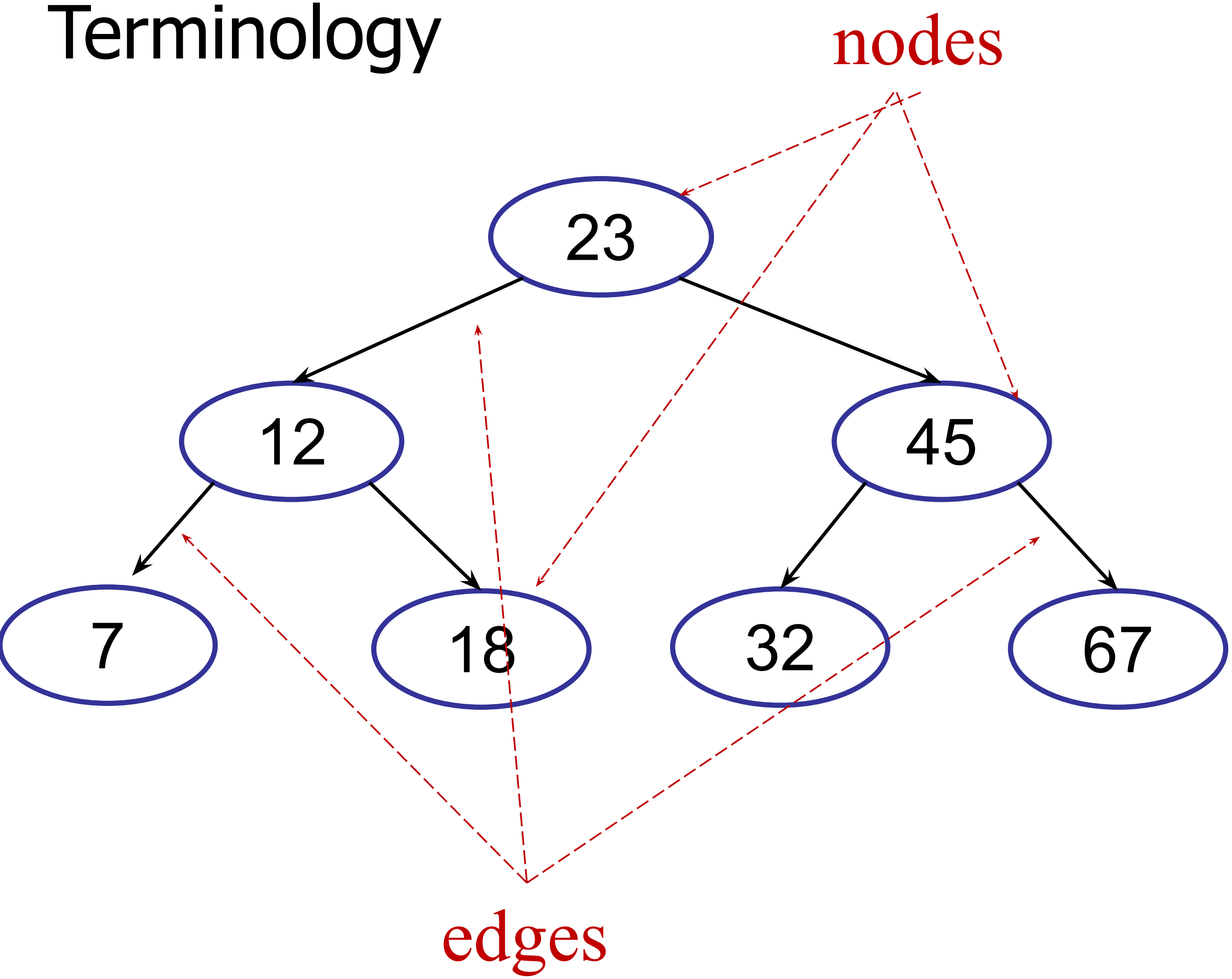
Critical Components:

- Nodes
- Edges directed from one node to another.
- Root (?)
- No cycles

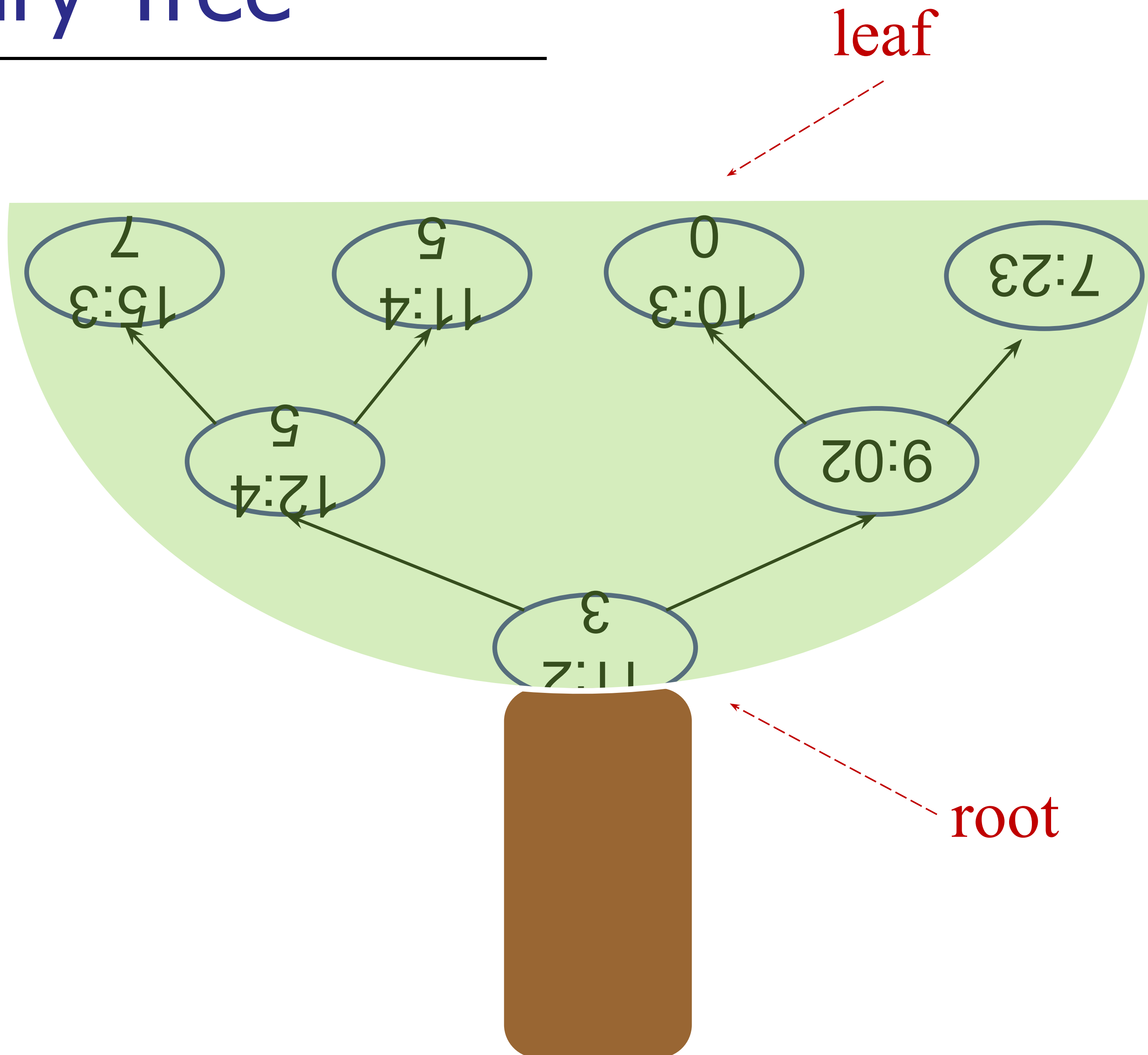


Binary Tree

Terminology



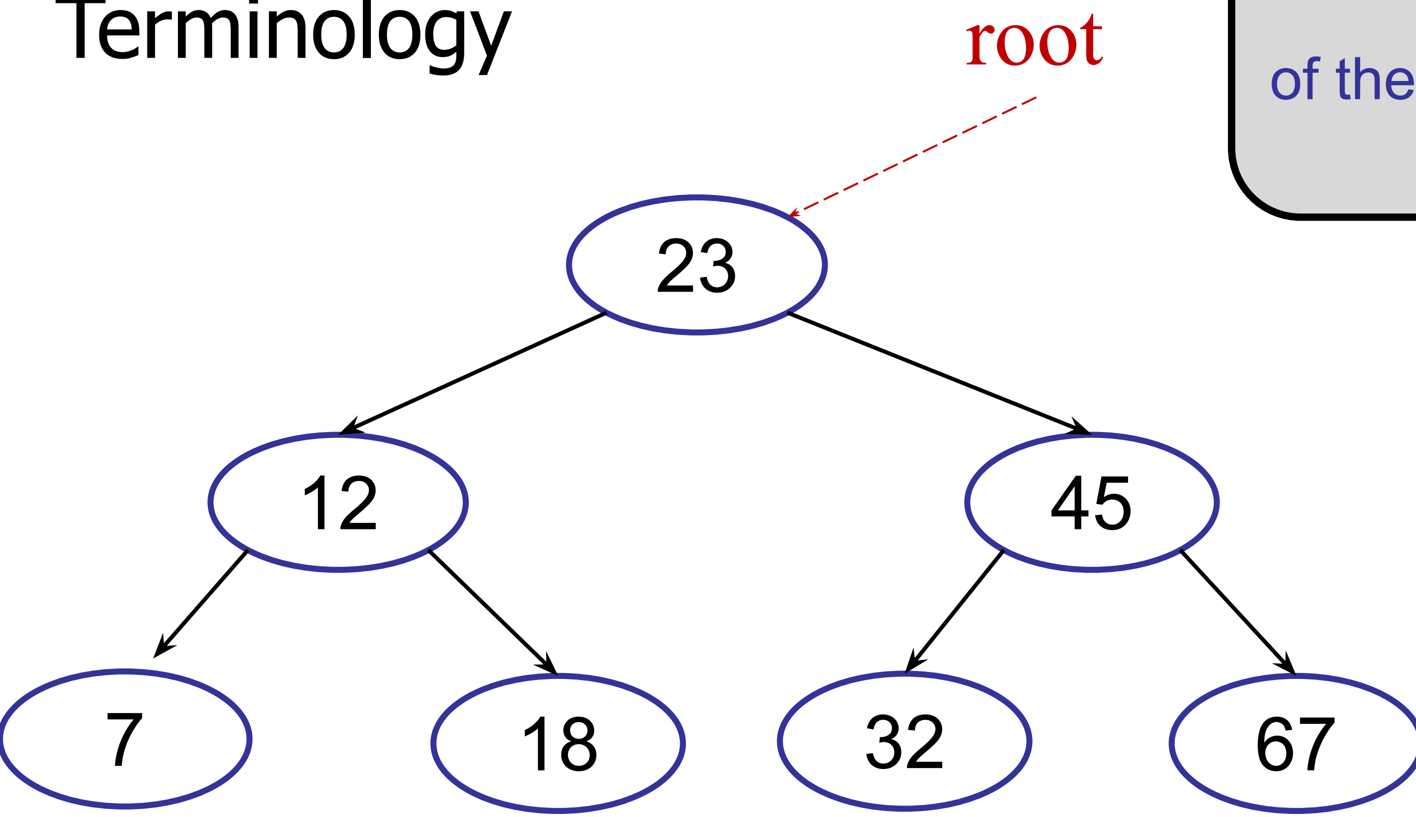
Binary Tree



Binary Tree

Terminology

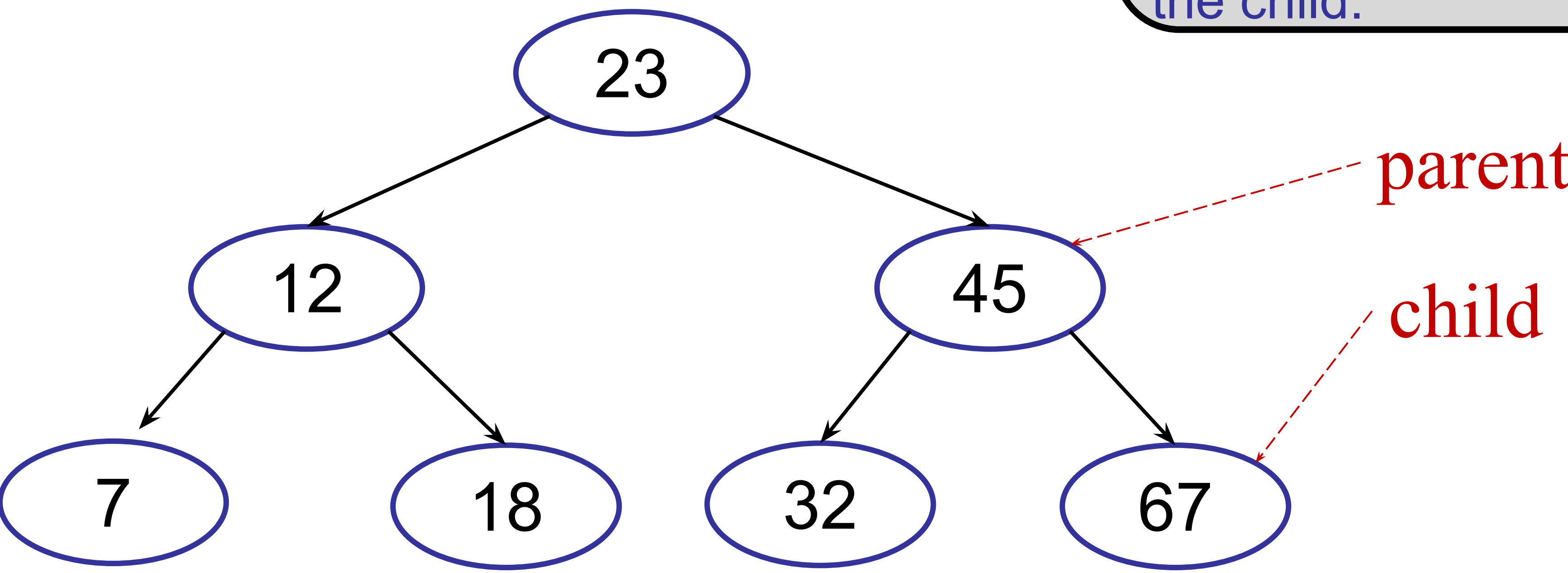
Root node is the node at the “start” of the tree



Binary Tree

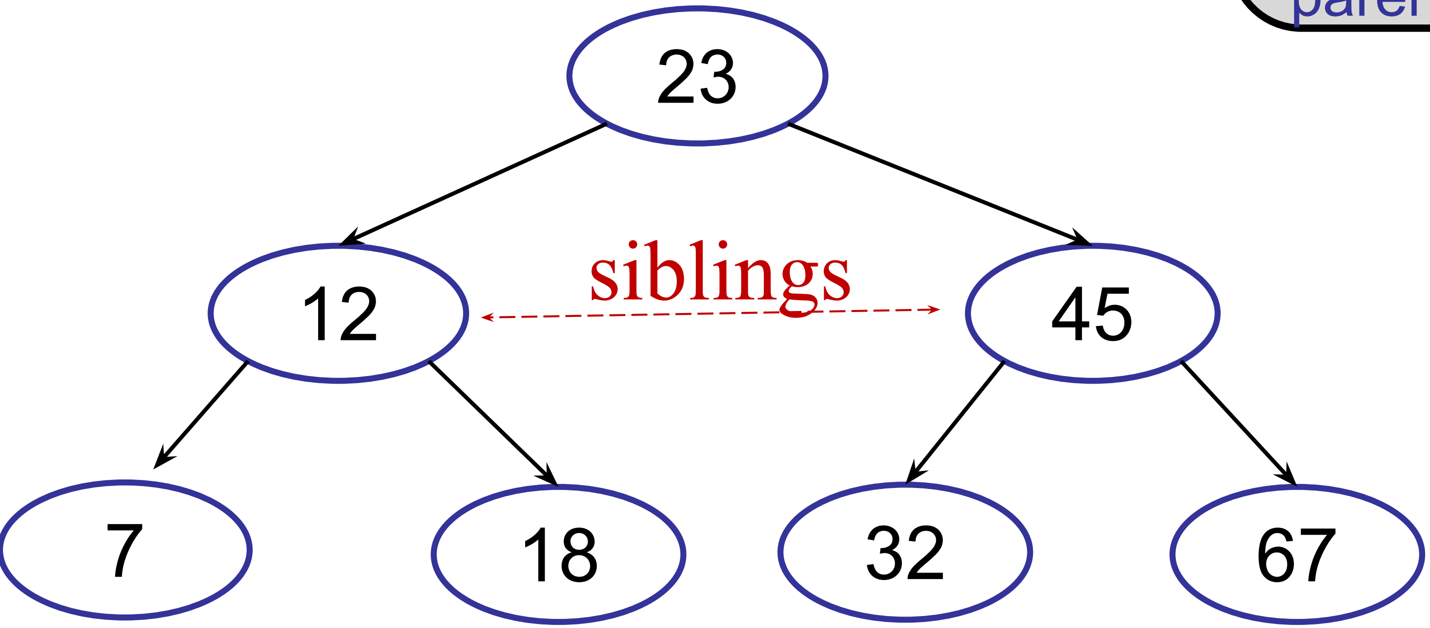
Terminology

A node is a "child" of a "parent" node if the parent points to the child.



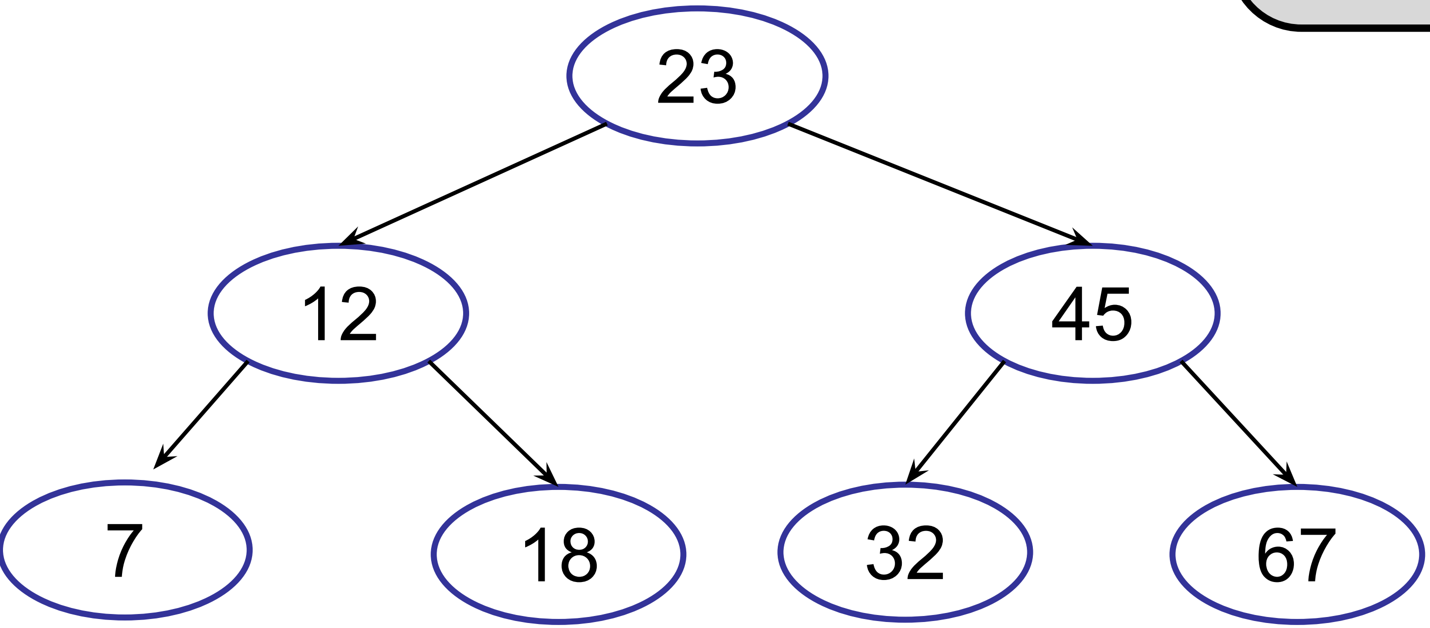
Binary Tree

Two nodes are called “siblings” if they have the same “parent”



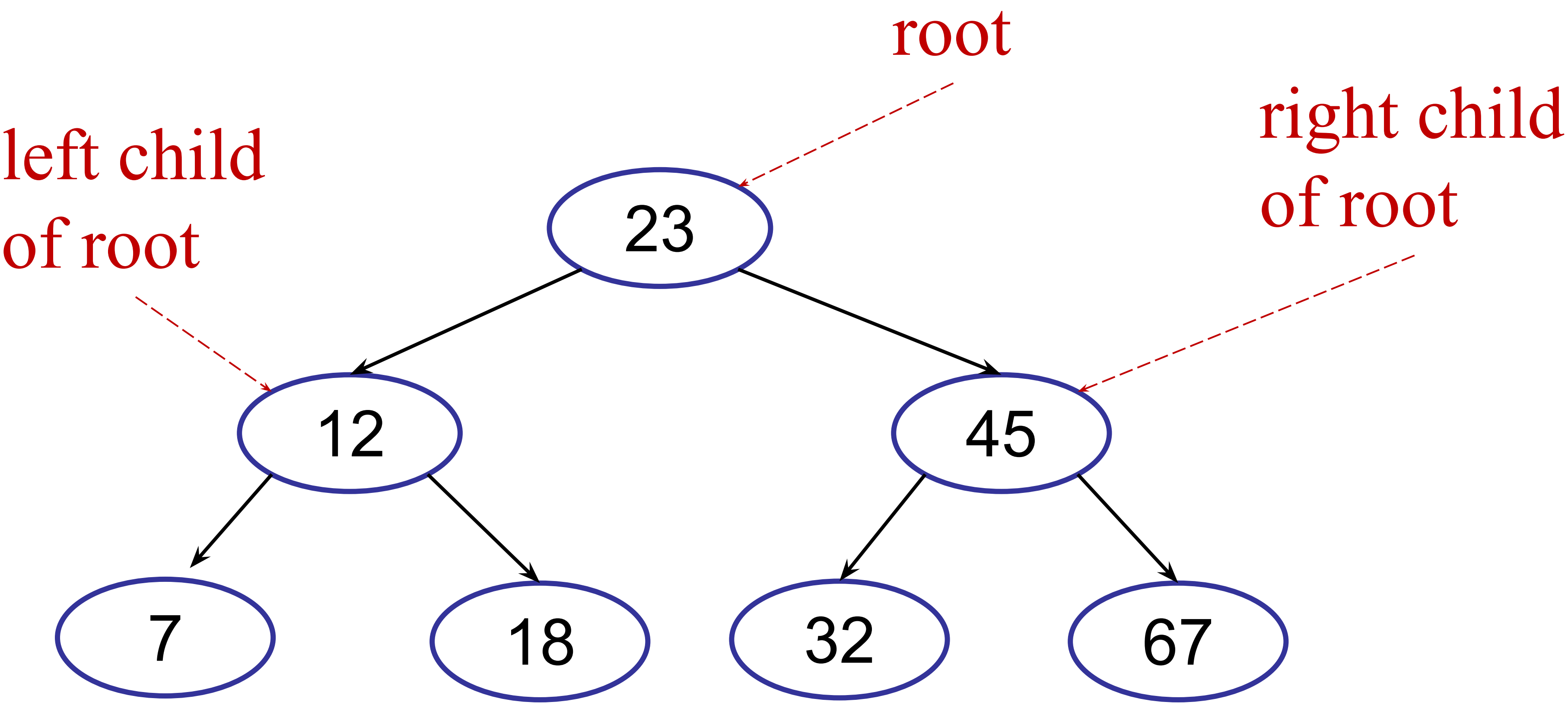
Binary Tree

A leaf has 0 children.



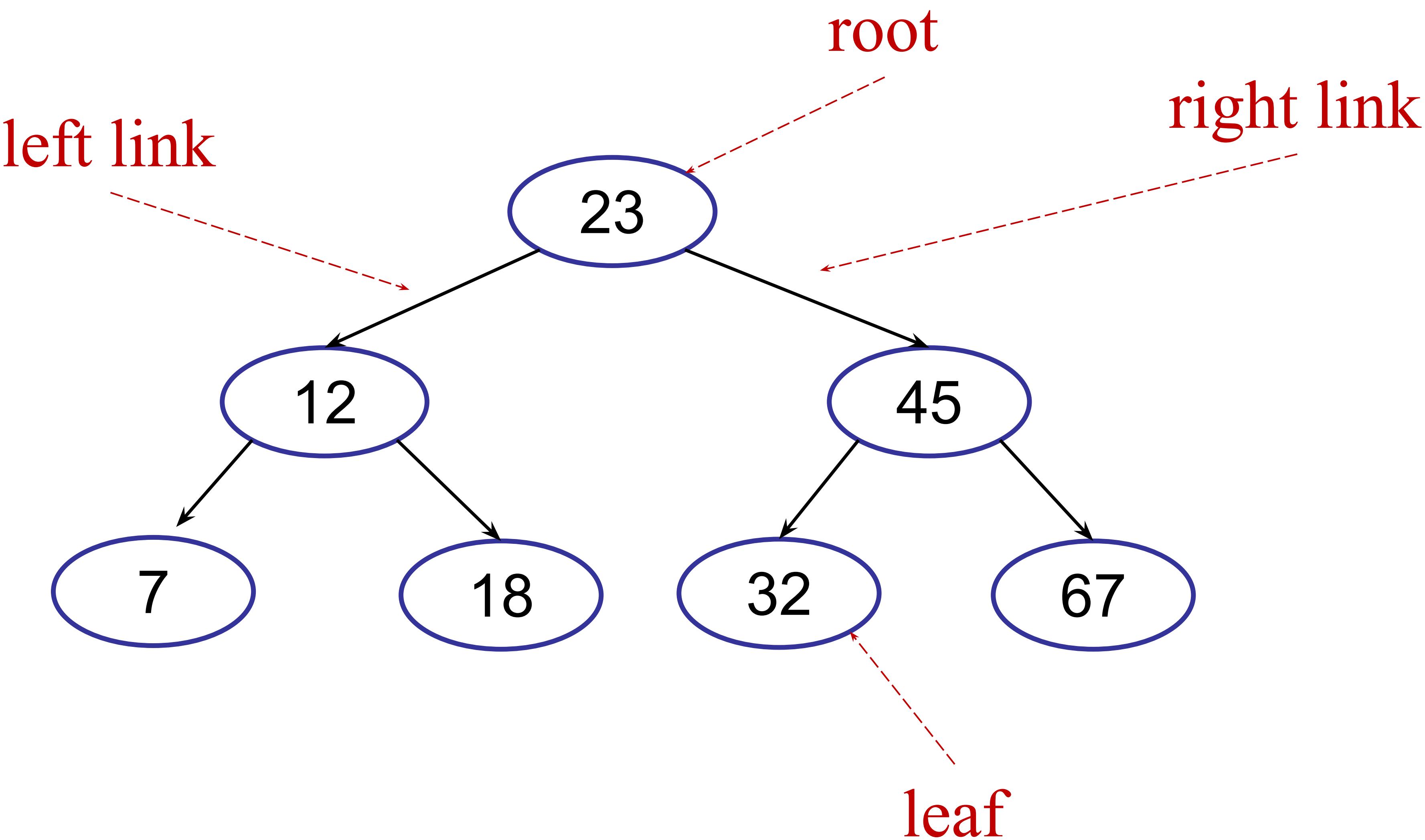
Binary Tree

Terminology



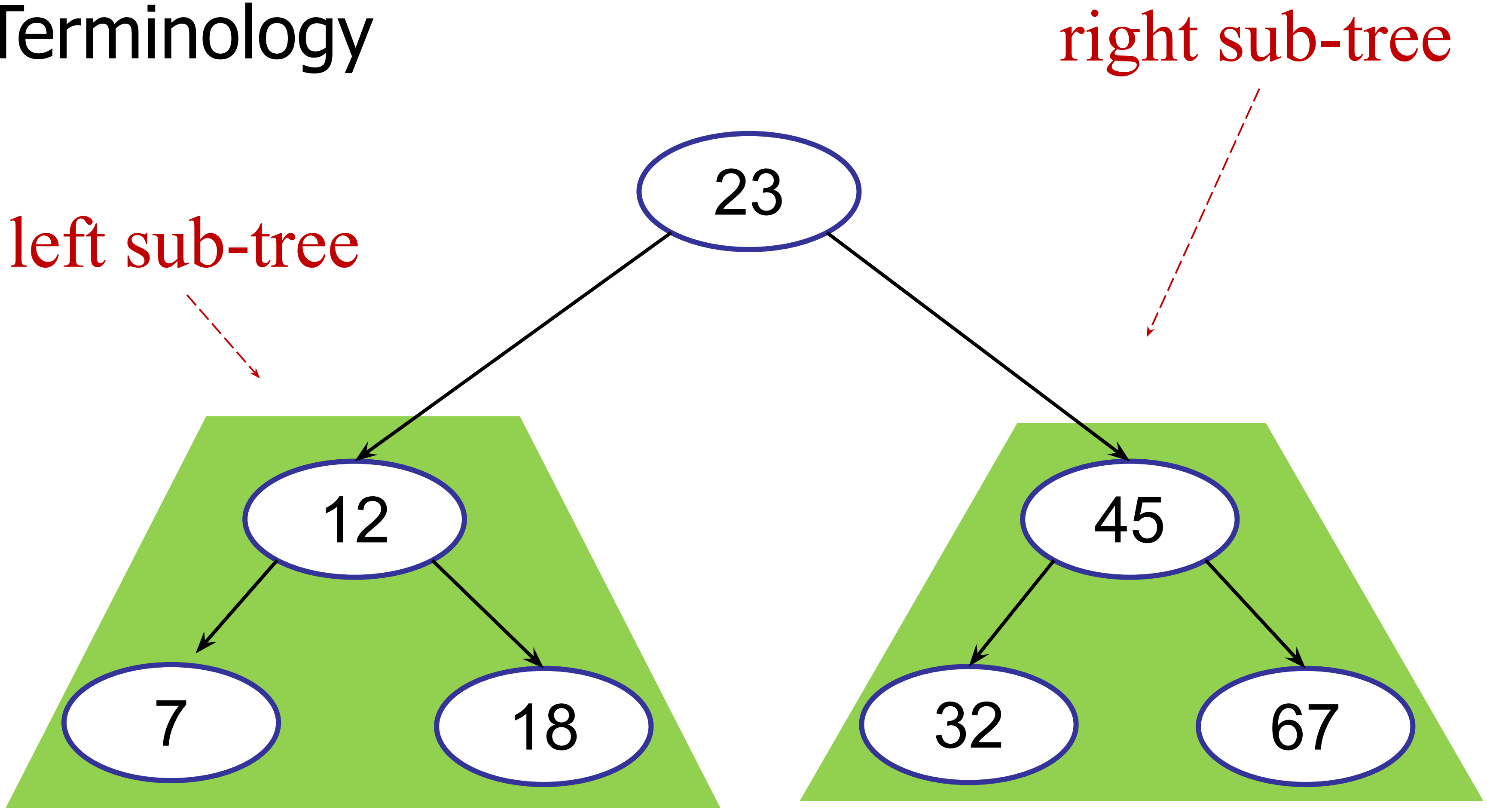
Binary Tree

Terminology



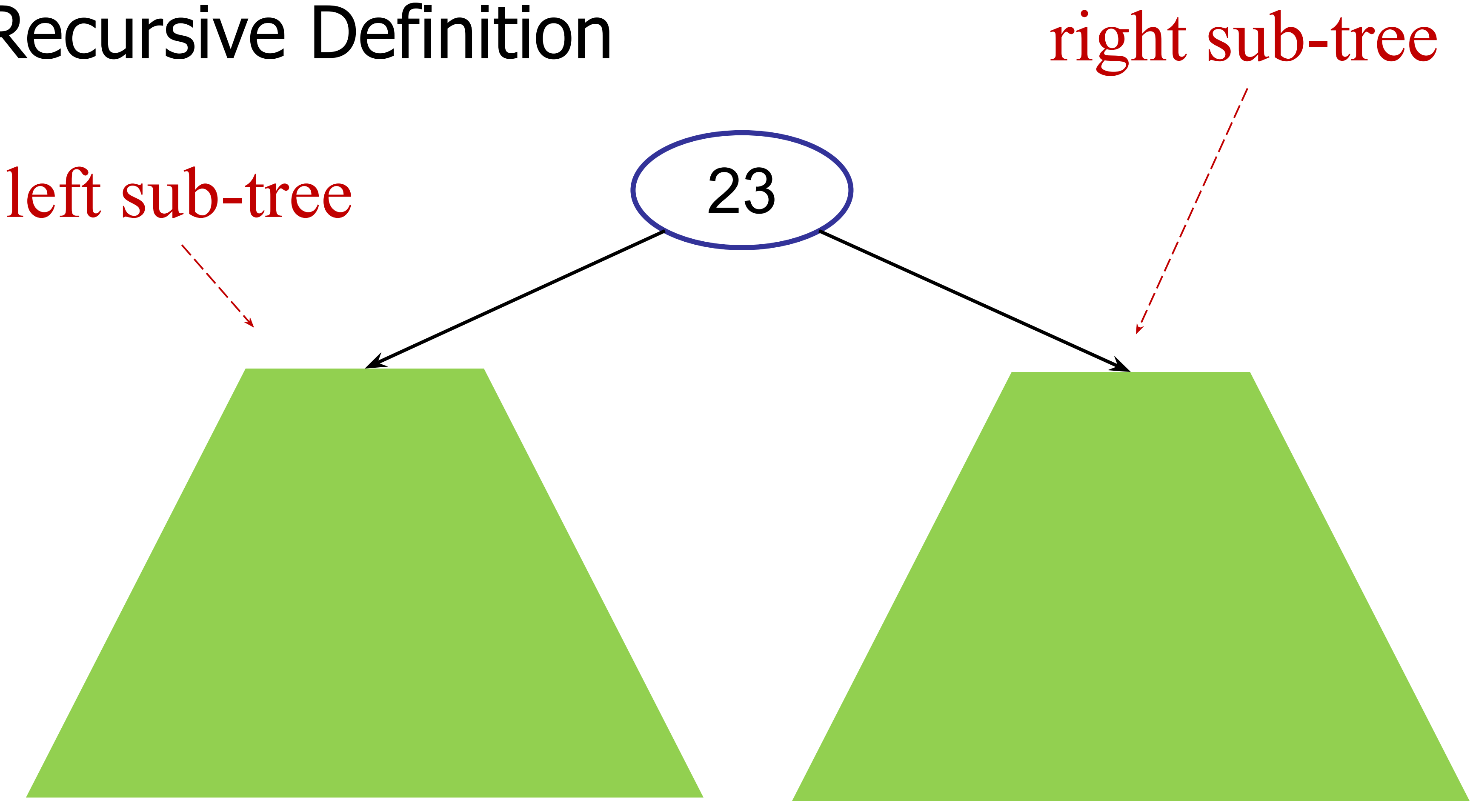
Binary Tree

Terminology



Binary Tree

Recursive Definition



A binary tree is either: (a) empty ; or
(b) a node pointing to two binary trees

Binary Tree

Java??

```
public class TreeNode {

    private TreeNode leftTree;
    private TreeNode rightTree;

    private KeyType key;
    private ValueType value;

    // Remainder of binary tree implementation
}
```

Binary Tree

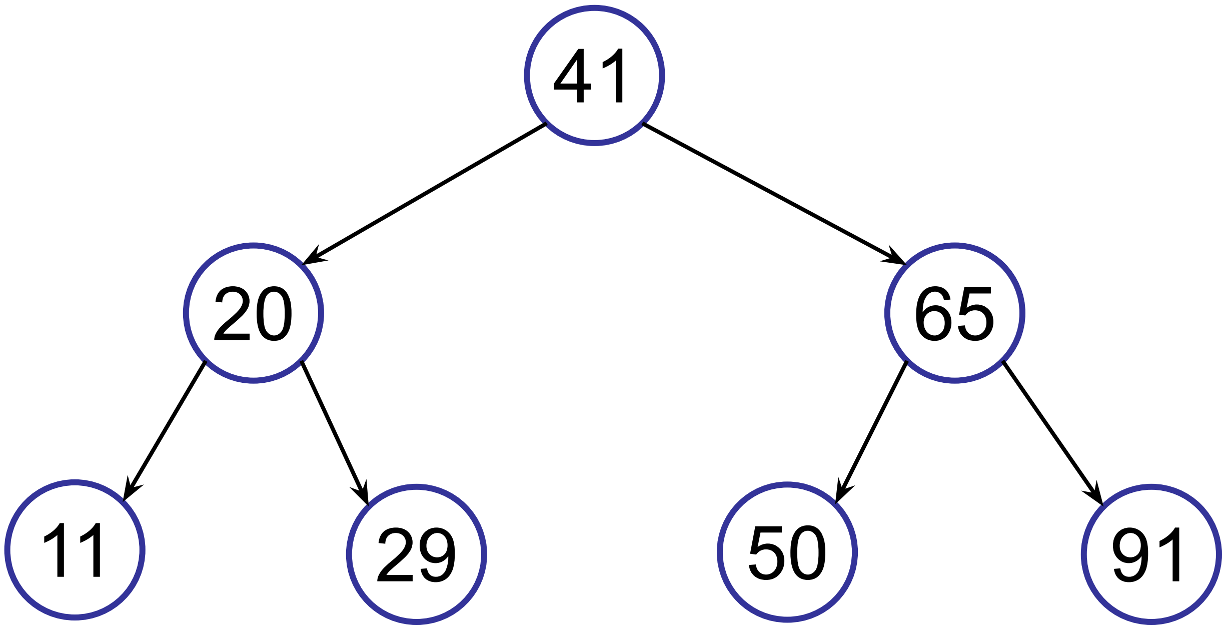
Java??

```
public class TreeNode {

    private TreeNode leftTree;
    private TreeNode rightTree;

    private int key;
    private int value;

    // Remainder of binary tree implementation
}
```

Example:

We want to store integer keys and values.

Binary **Search** Trees (BST)

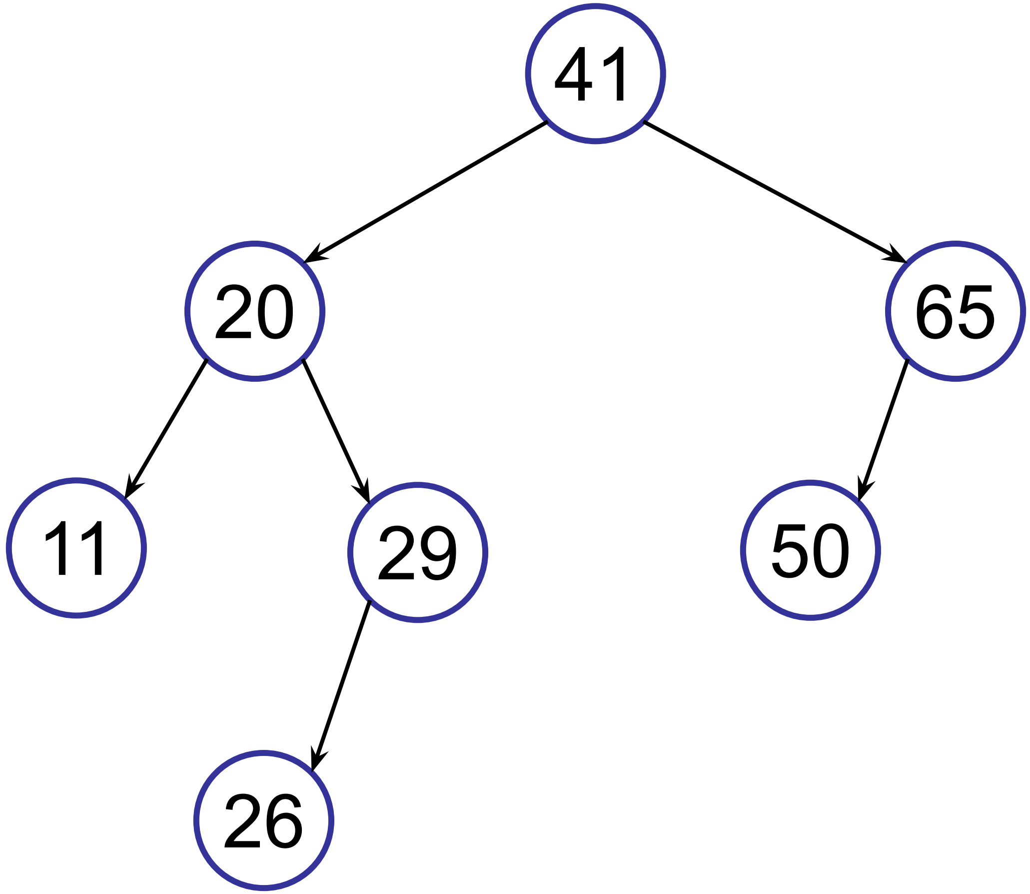


BST Property:

all keys in left sub-tree < key < all keys in right sub-tree

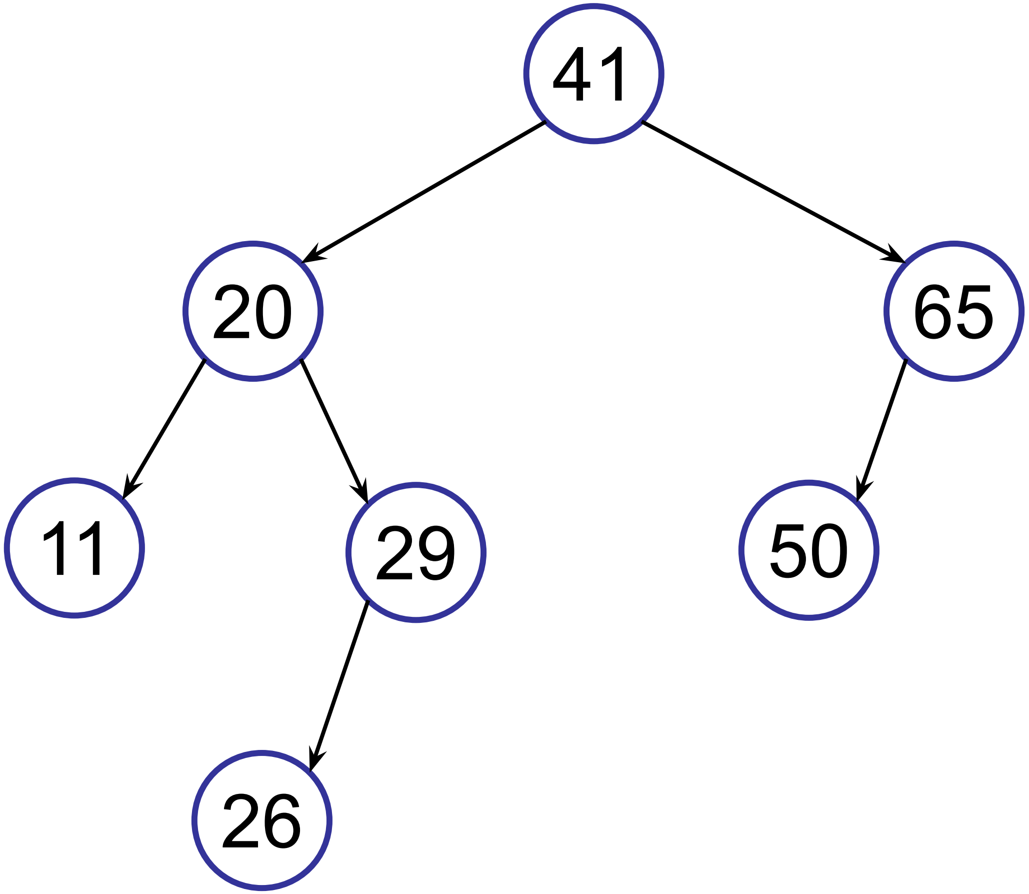
Is this a binary search tree?

1. Yes
2. No
3. I don't know.



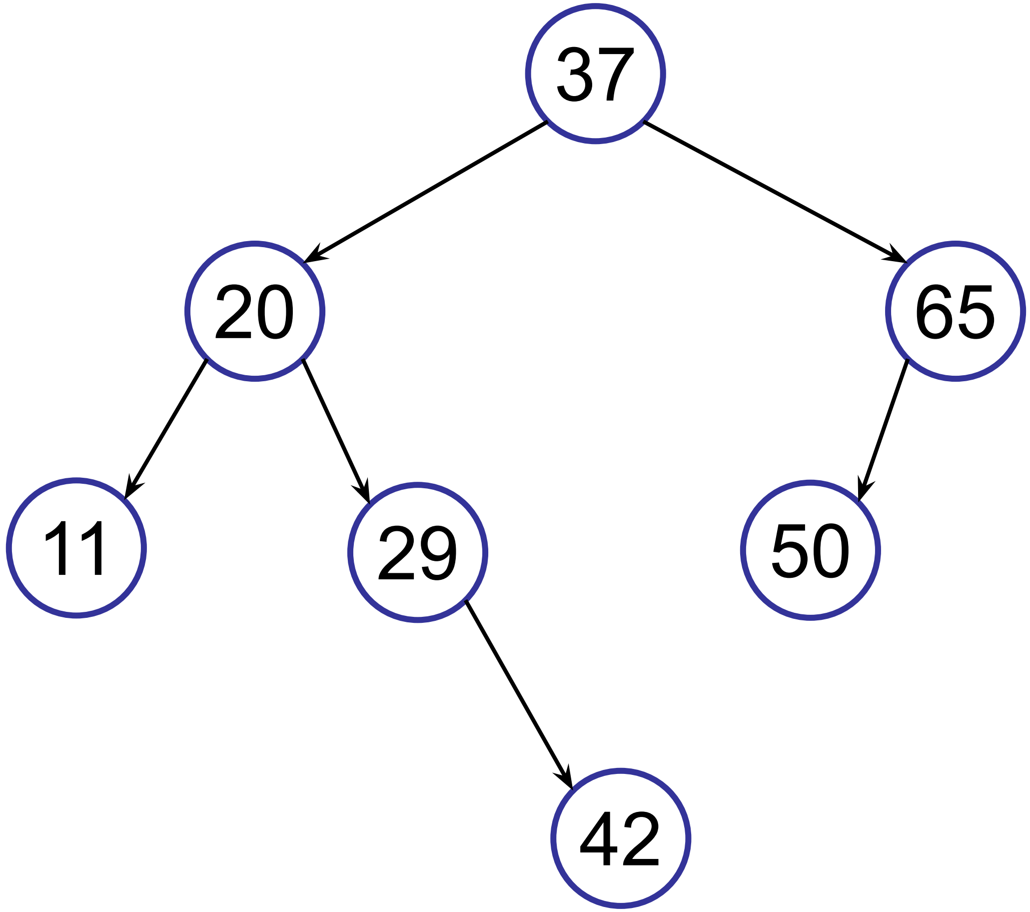
Is this a binary search tree?

1. Yes
2. No
3. I don't know.



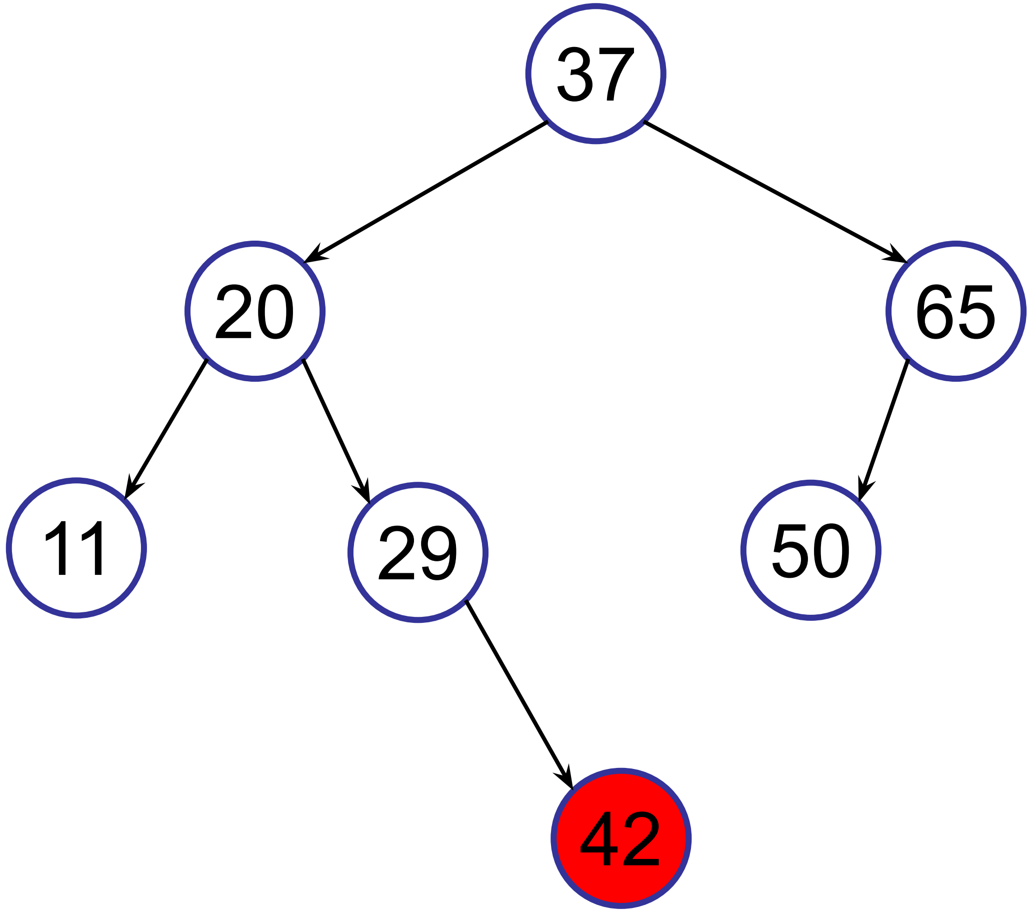
Is this a binary search tree?

1. Yes
2. No
3. I don't know.



Is this a binary search tree?

1. Yes
2. No
3. I don't know.



Binary Search Trees

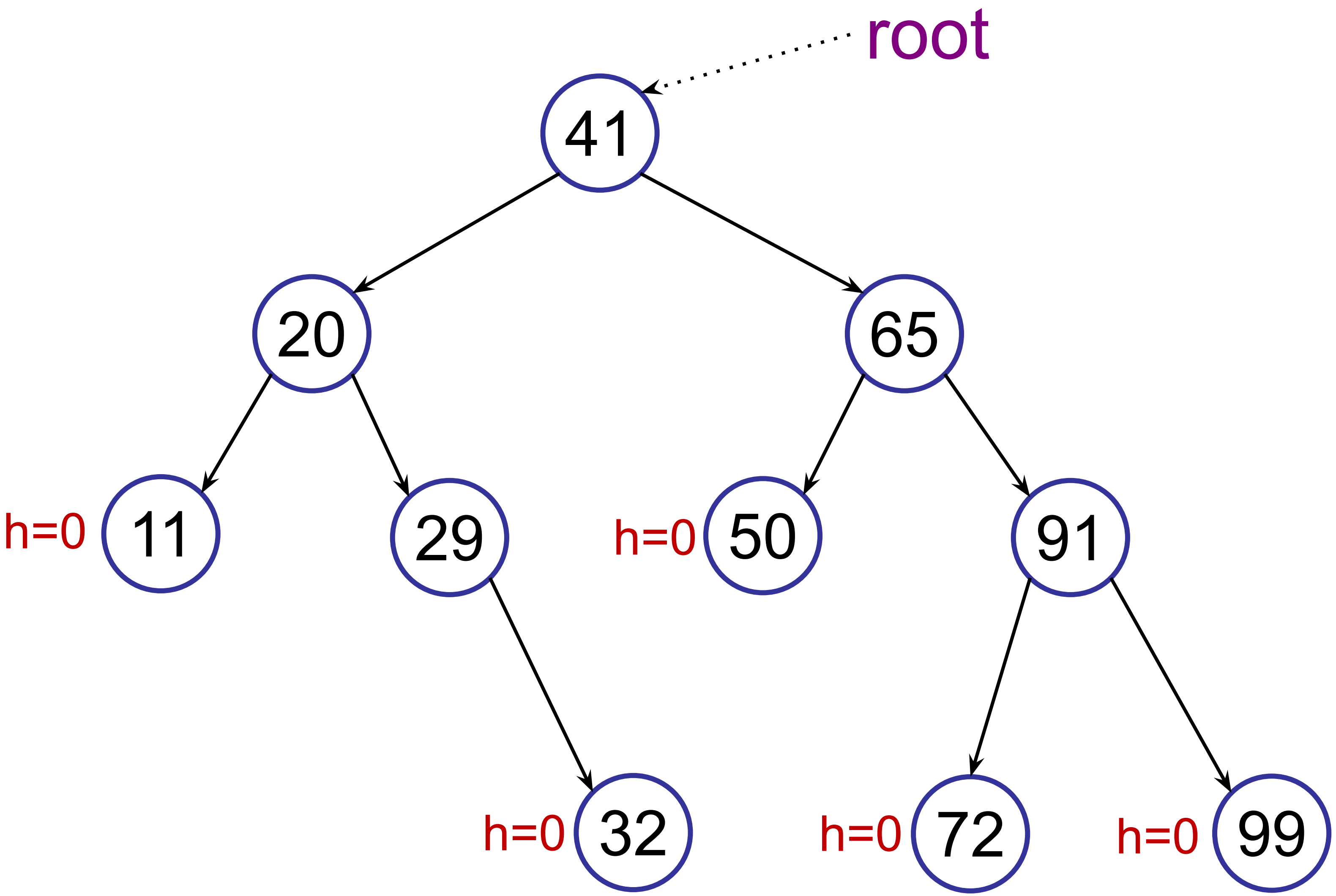
1. Terminology and Definitions

2. Basic operations:
   - height
   - search, insert
   - searchMin, searchMax

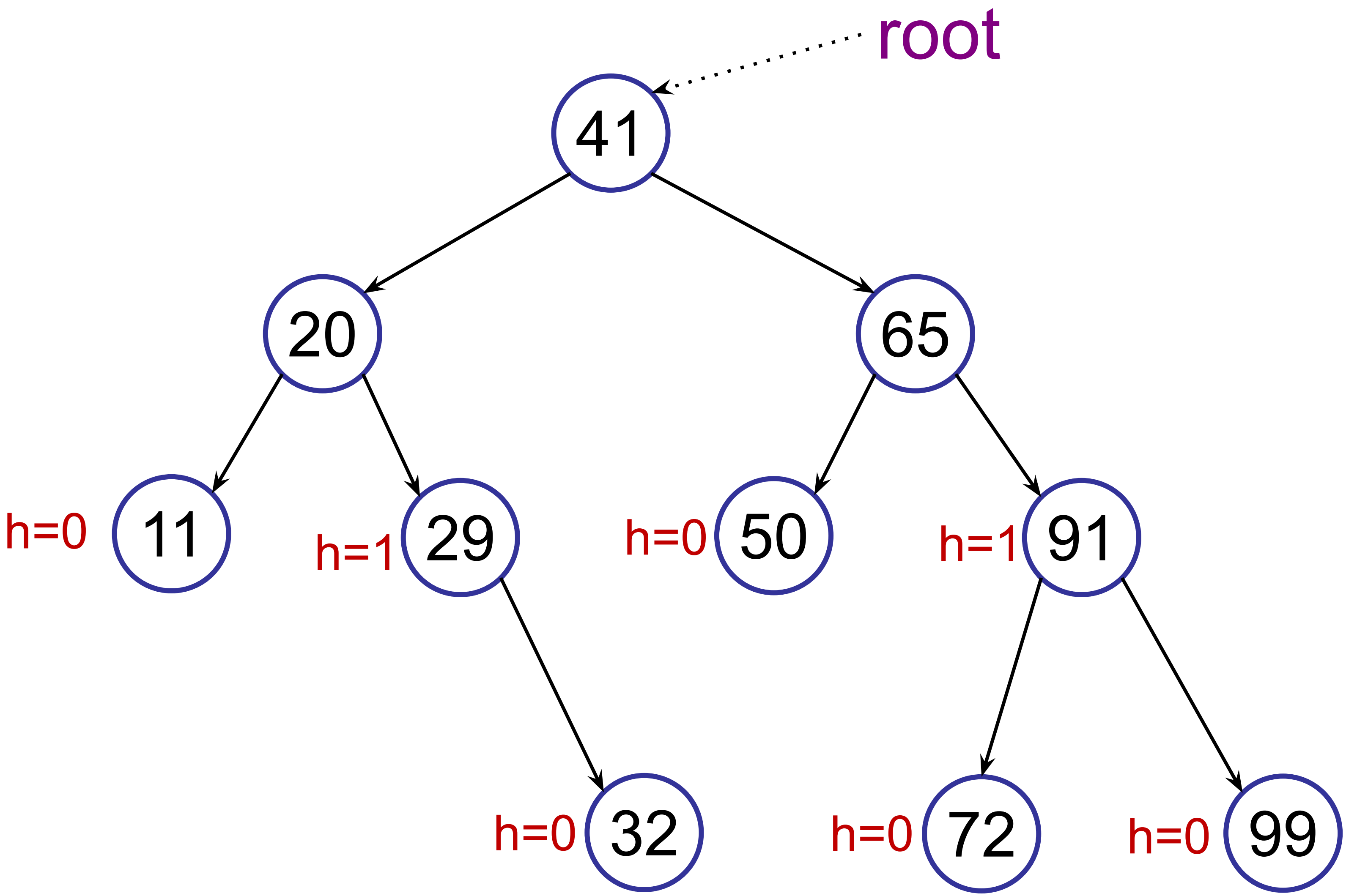
3. Traversals
   - in-order, pre-order, post-order

4. Other operations

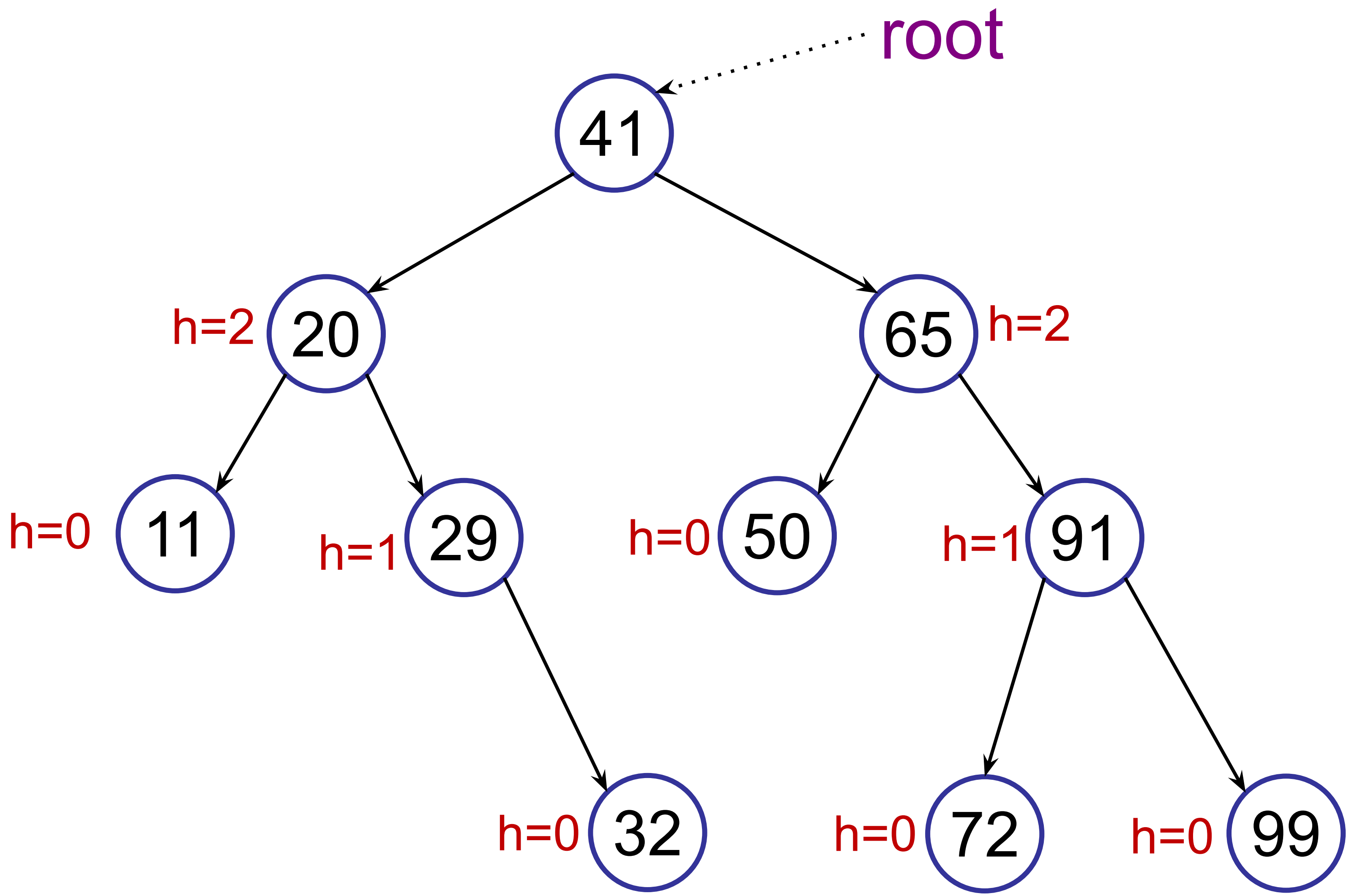
Height of a Binary Tree



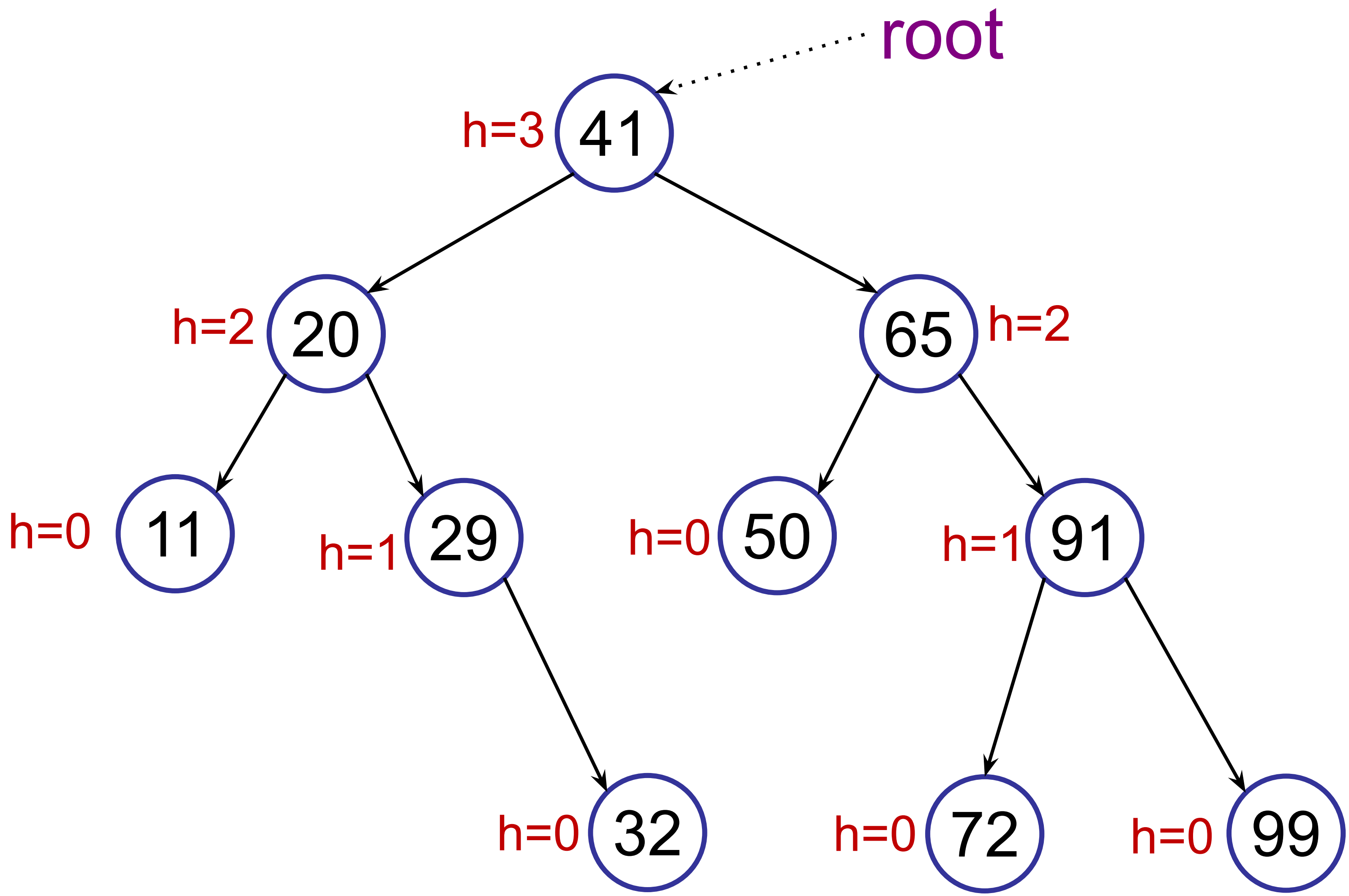
Height of a Binary Tree



Height of a Binary Tree



Height of a Binary Tree



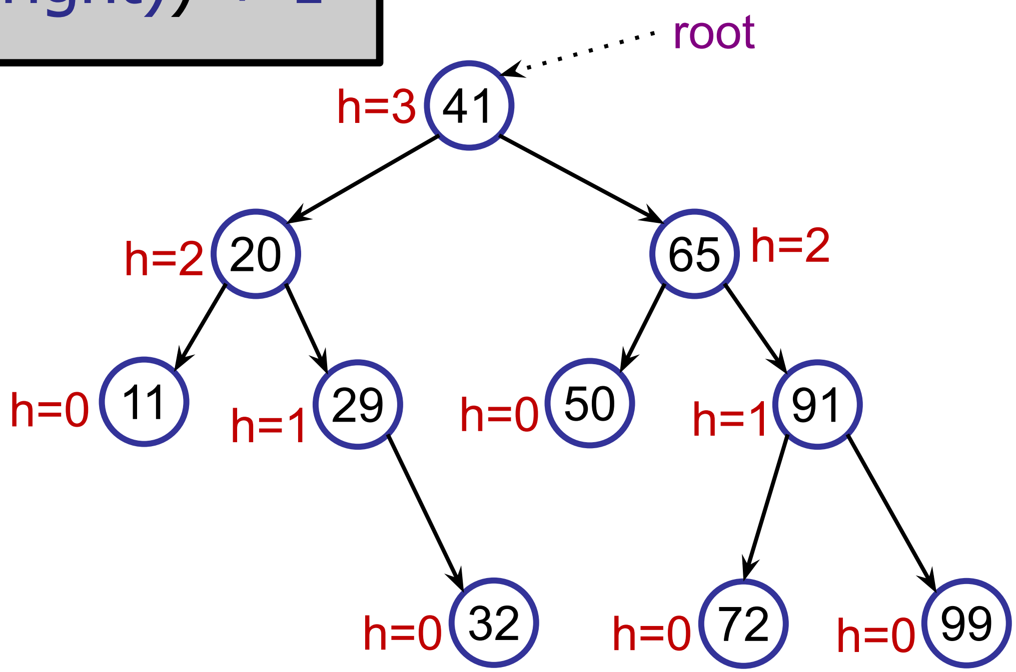
Height of a Binary Tree

Height:

Number of edges on longest path from root to leaf.

$h(v) = 0$ (if v is a leaf)

$h(v) = \max(h(v.left), h(v.right)) + 1$



(For simplicity: $h(null) = -1$)

Binary Tree

Calculating the heights

check for null

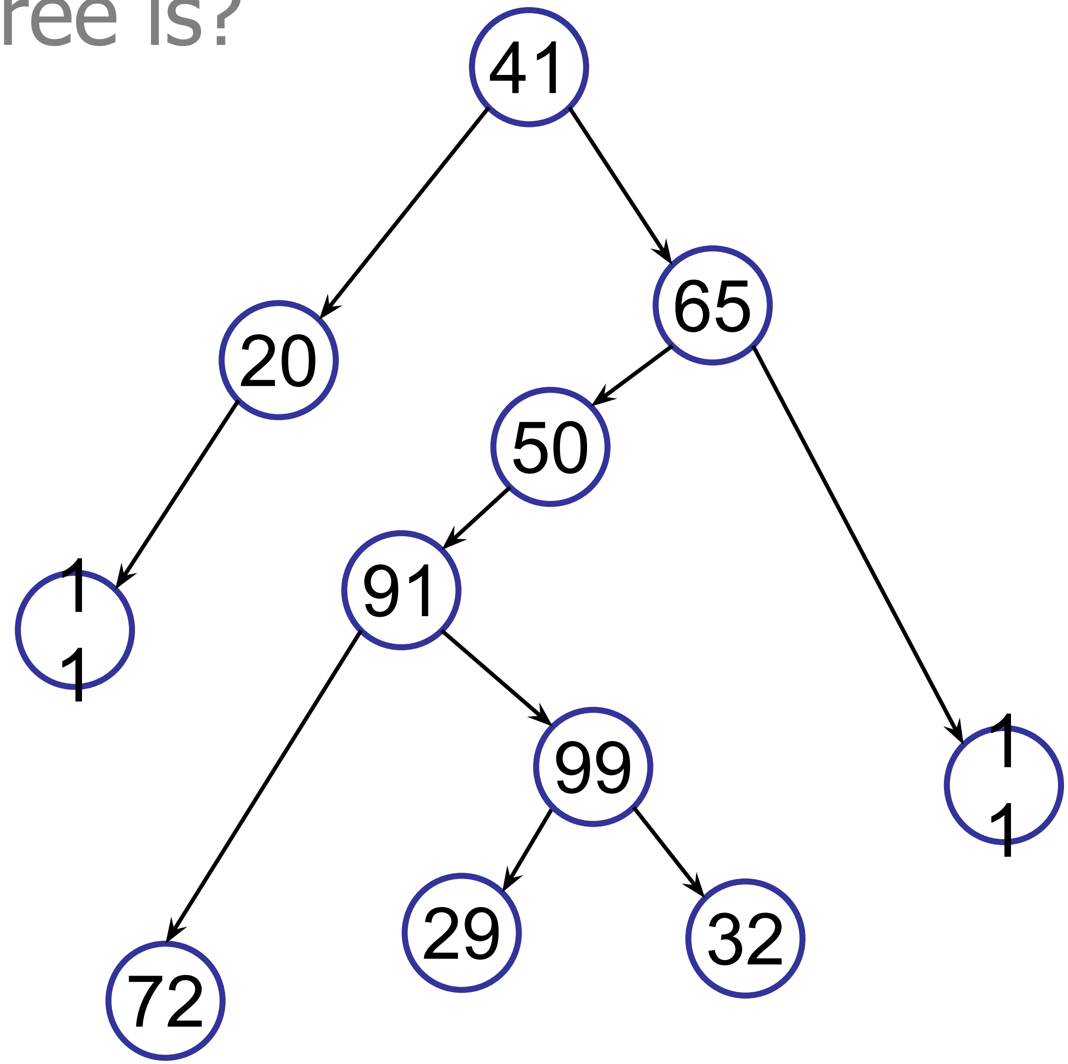
```
public int height(){
    int leftHeight = -1;
    int rightHeight = -1;
    if (leftTree != null)
        leftHeight = leftTree.height();
    if (rightTree != null)
        rightHeight = rightTree.height();
    return max(leftHeight, rightHeight) +  1;
}
```

max of subtrees

add 1

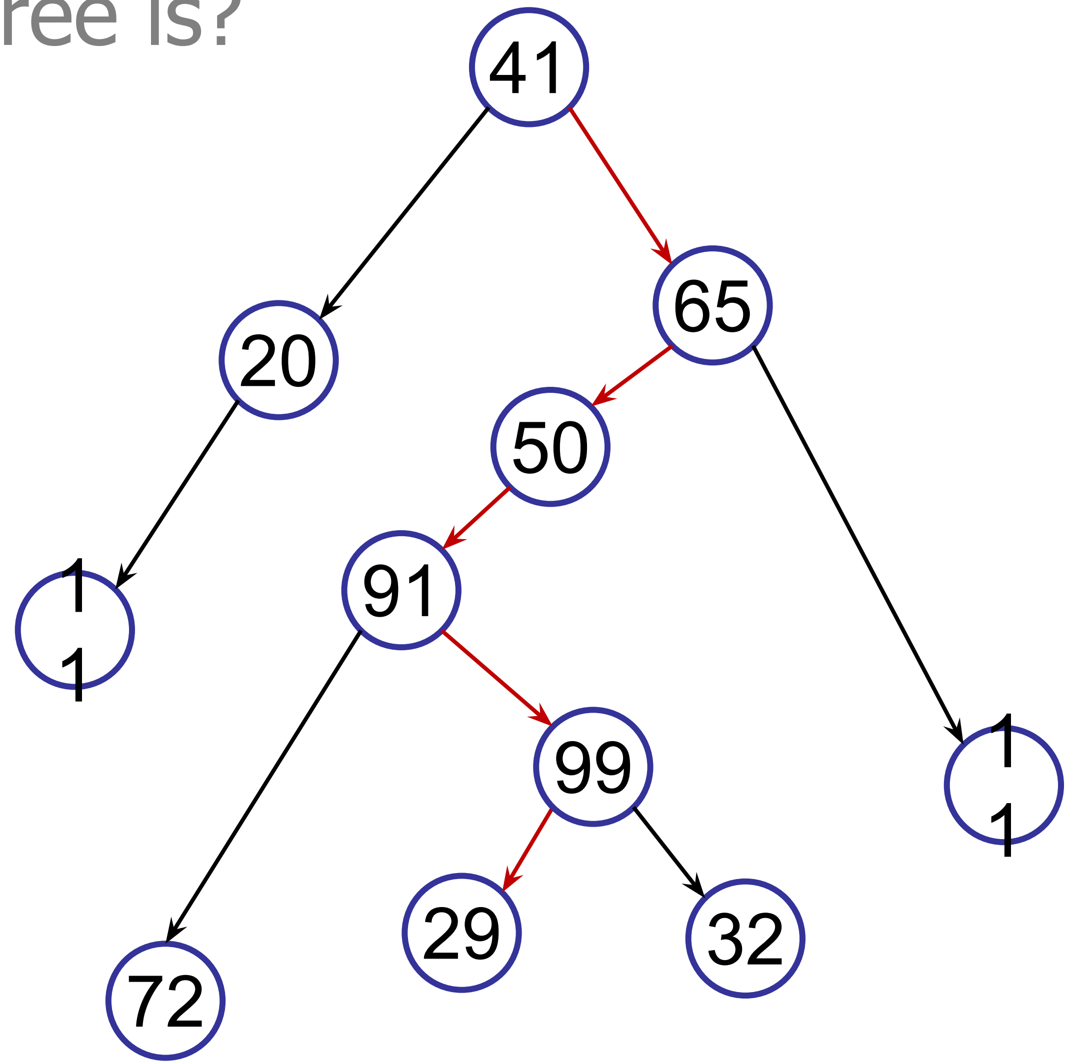
The height of this tree is?

1. 2
2. 4
3. 5
4. 6
5. 7
6. 42



The height of this tree is?

1. 2
2. 4
3. 5
4. 6
5. 7
6. 42



Binary Search Trees

1. Terminology and Definitions

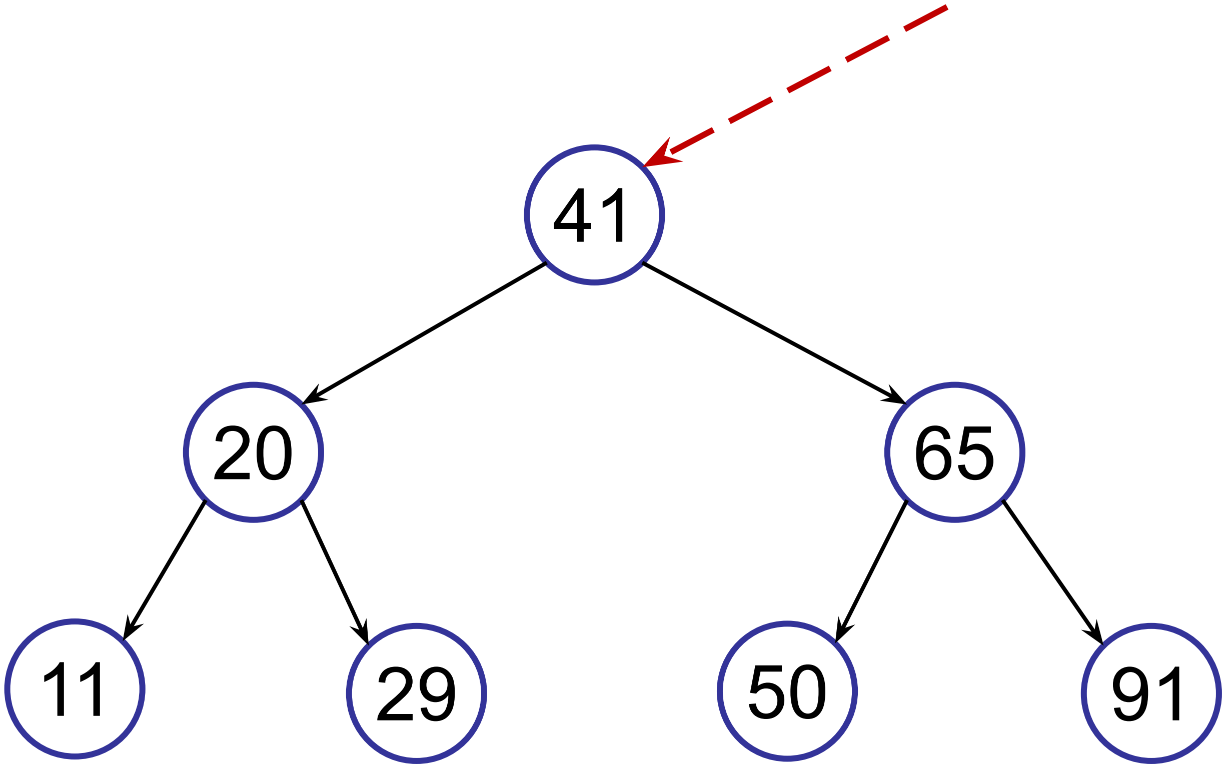
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

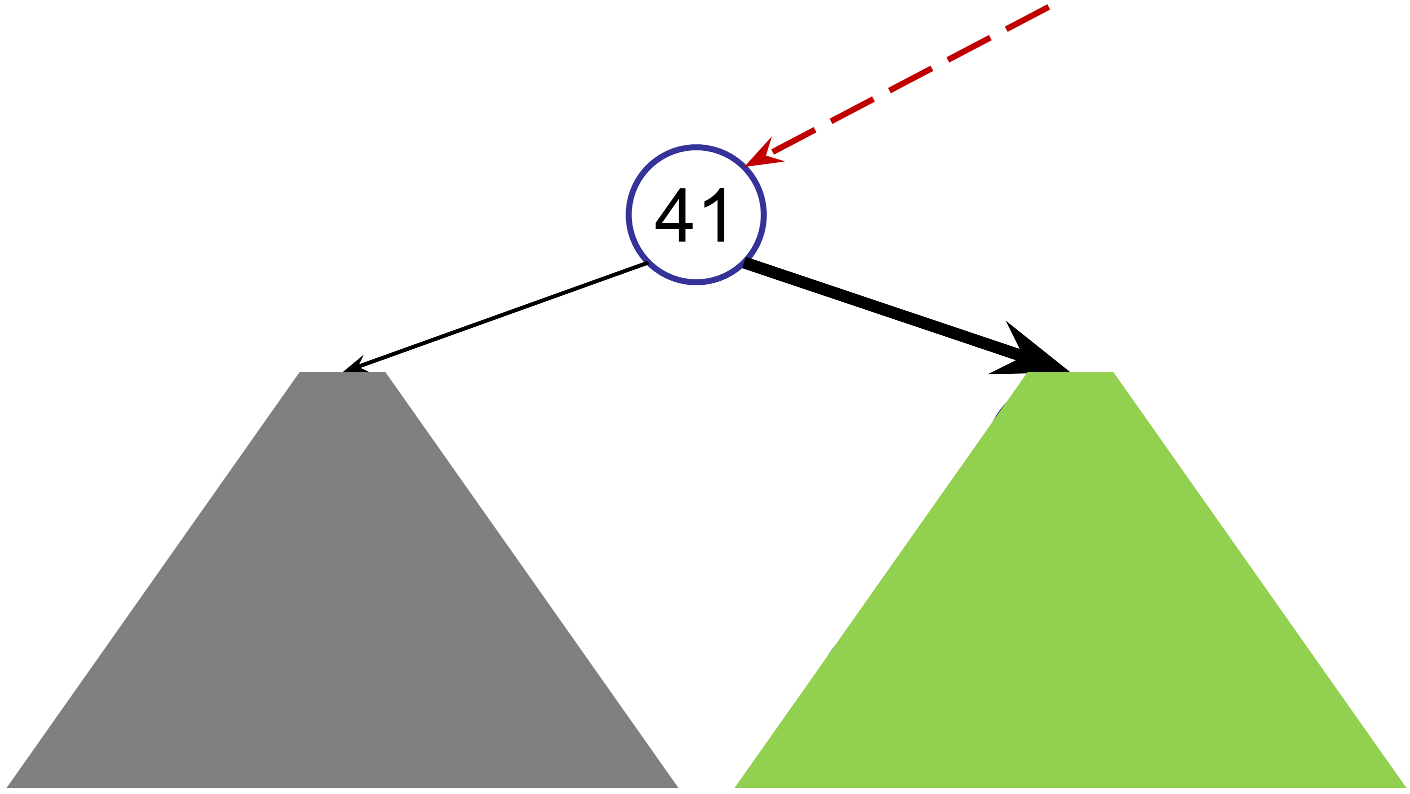
Binary Search Trees

Search for the maximum key:



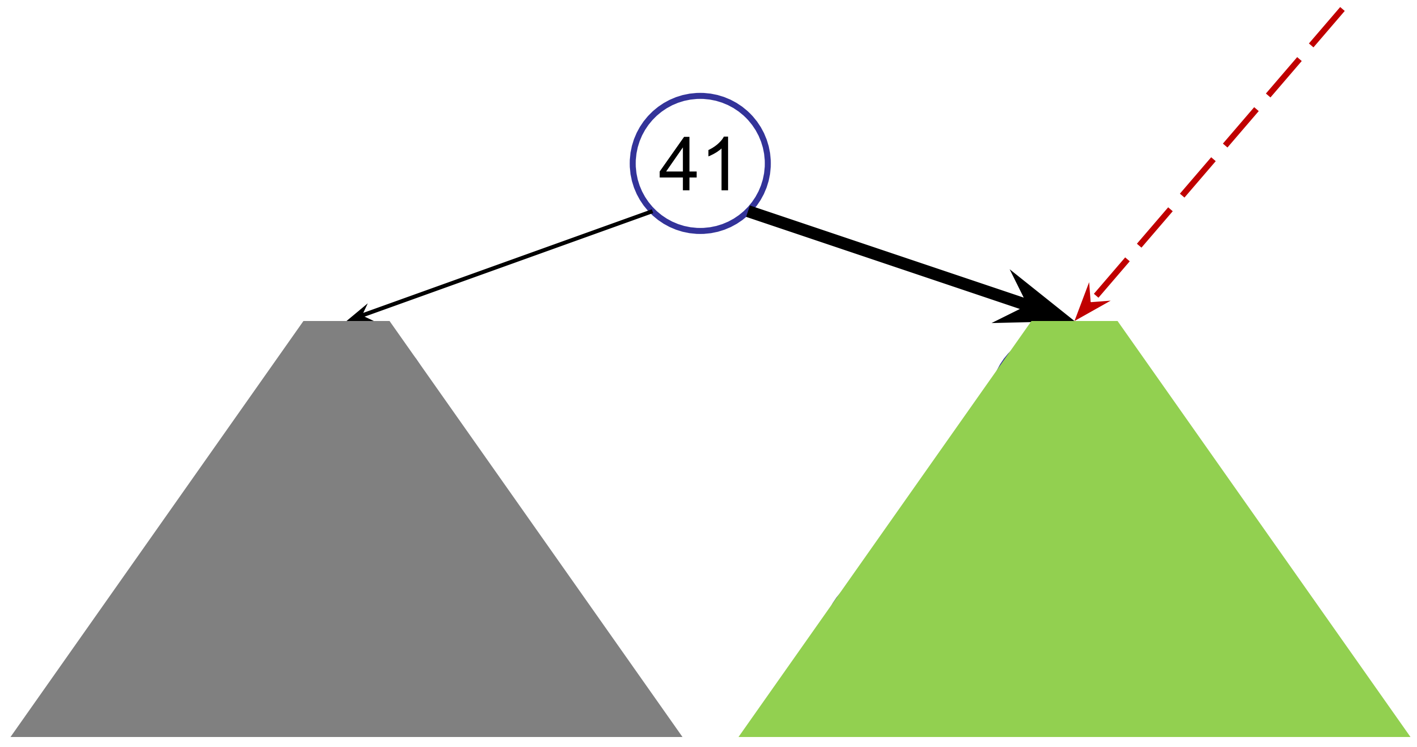
Binary Search Trees

Search for the maximum key:



Binary Search Trees

Search for maximum key



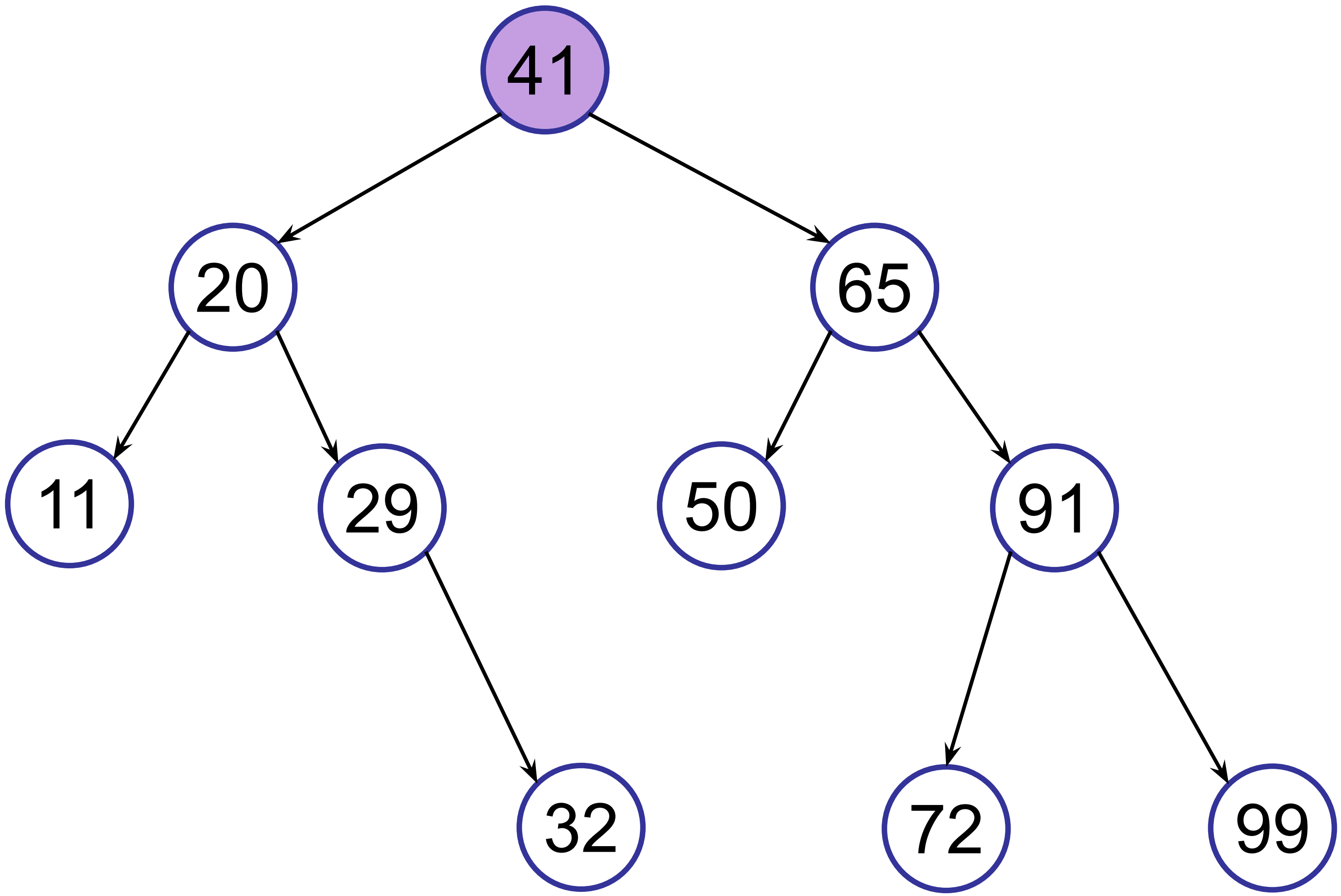
Binary Tree

Searching for the node with the maximum key

```
public TreeNode searchMax(){
    if (rightTree != null) {
        return rightTree.searchMax();
    }
    else return this; // Key is here!
}
```

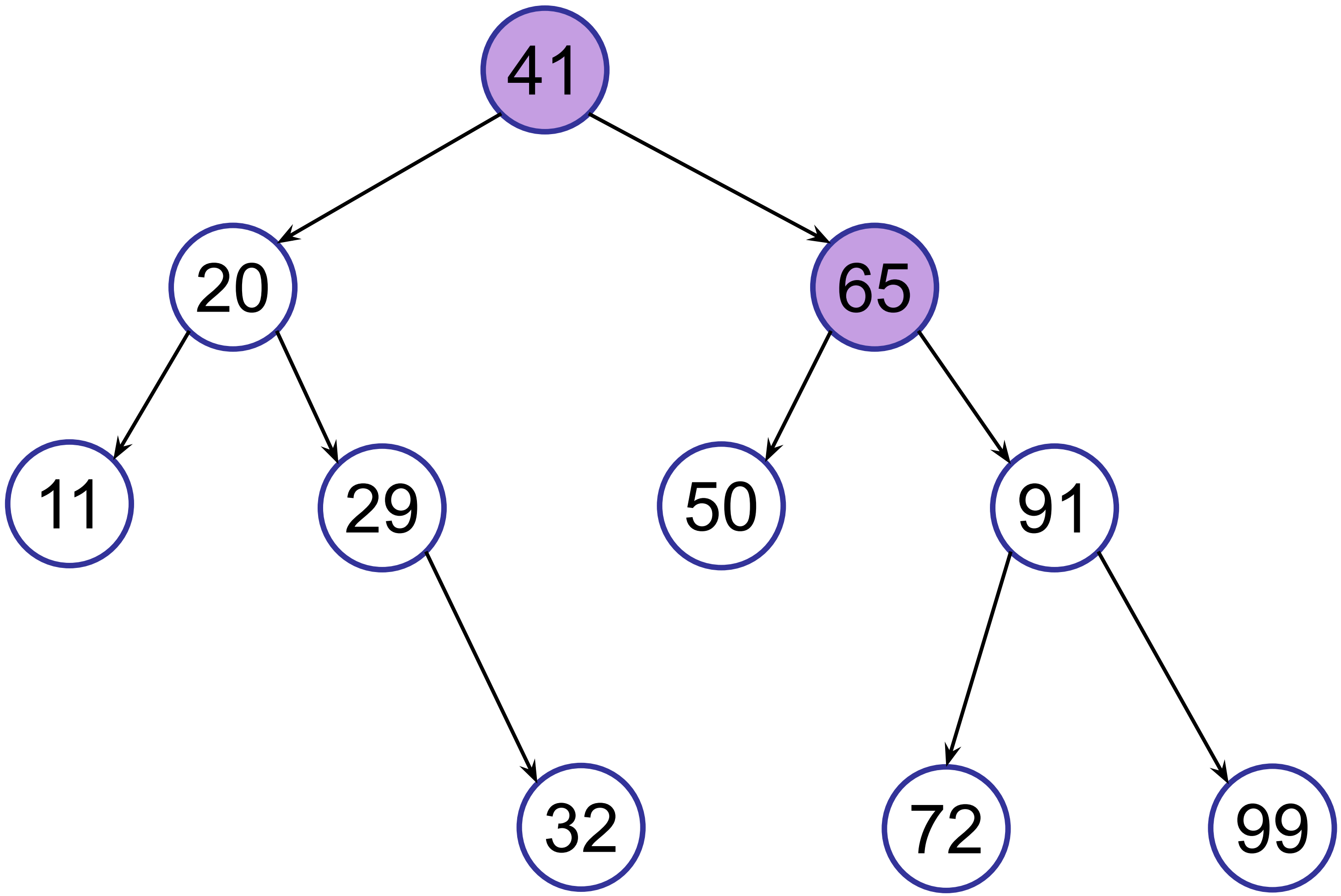
Binary Search Trees

searchMax()



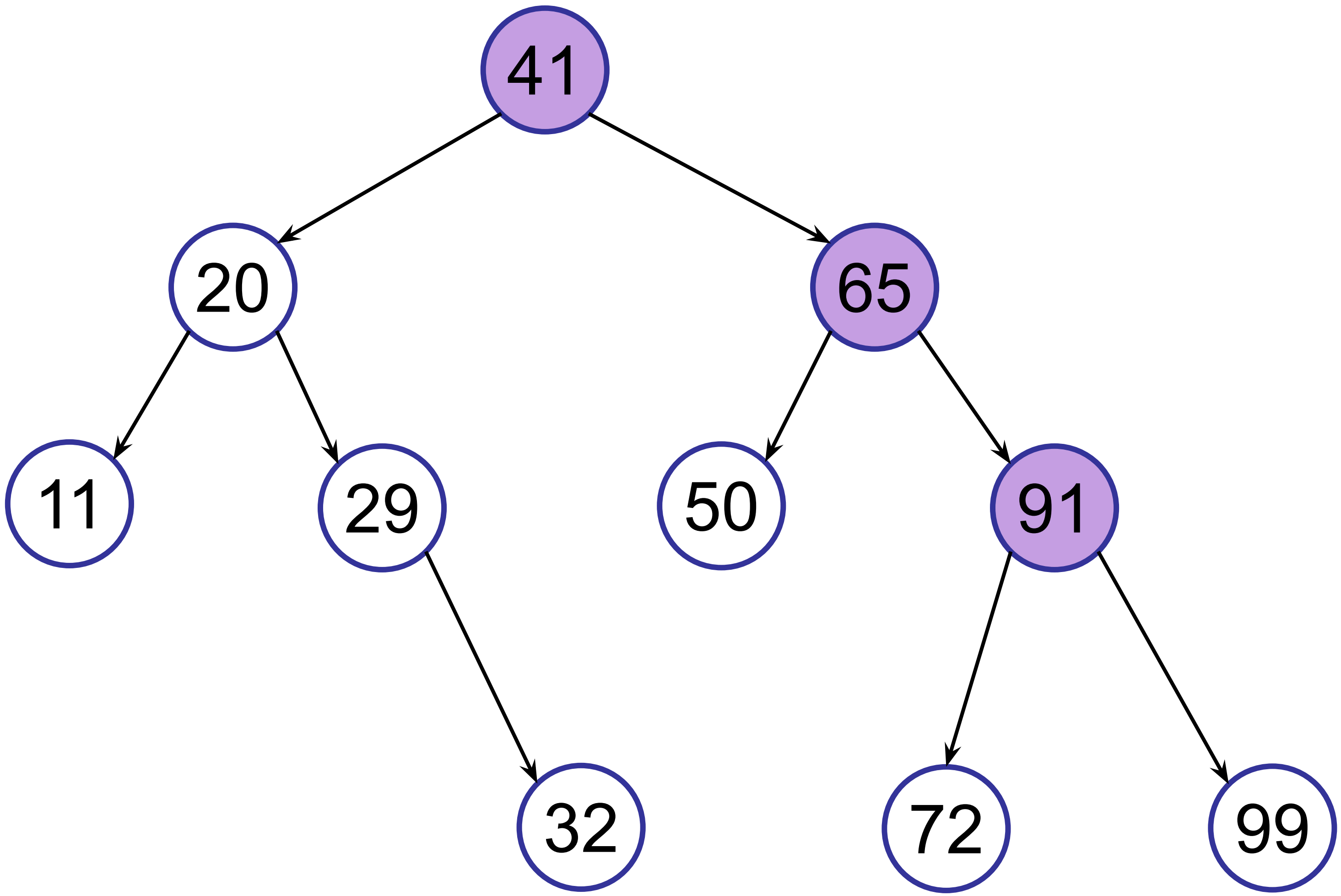
Binary Search Trees

searchMax()



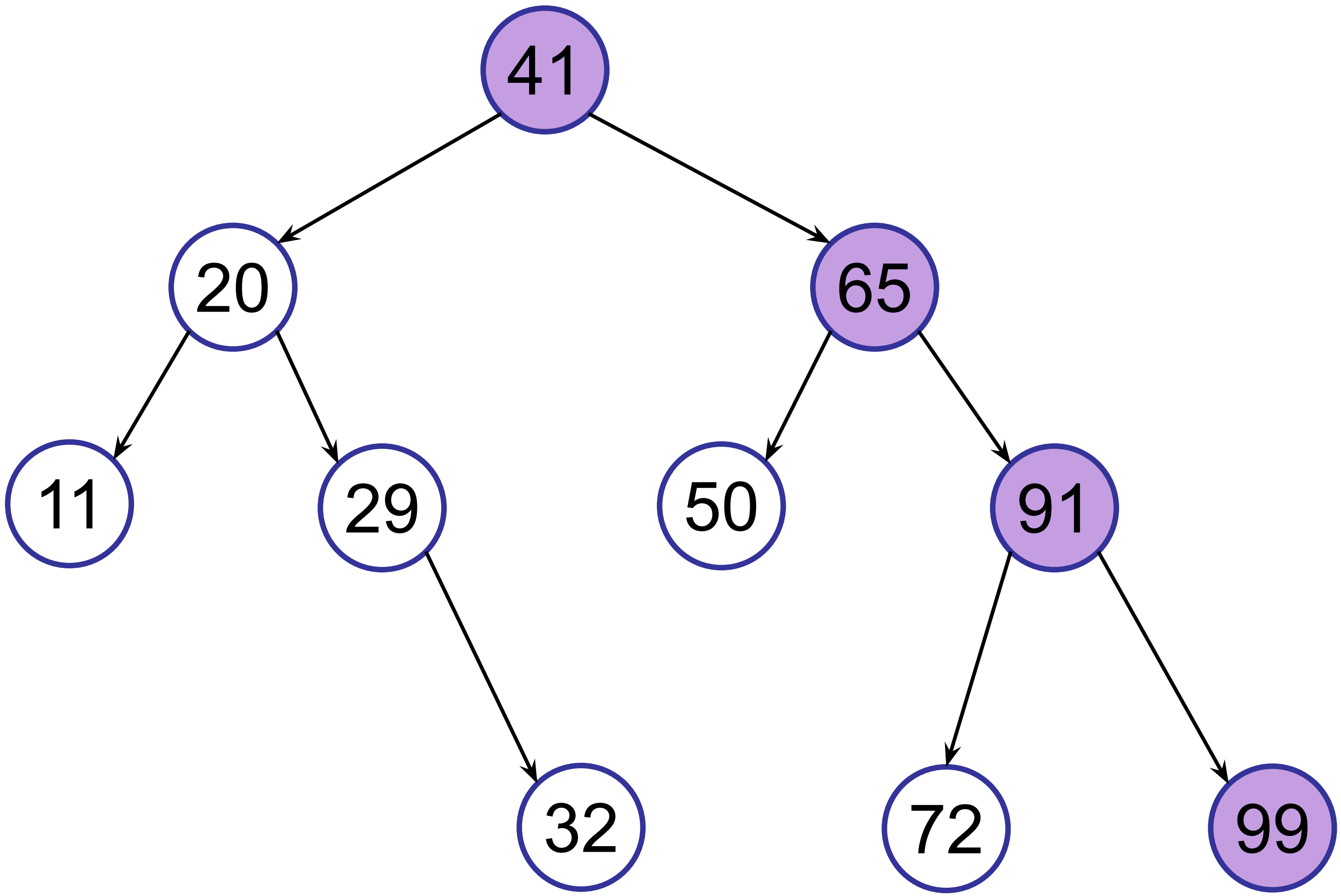
Binary Search Trees

searchMax()



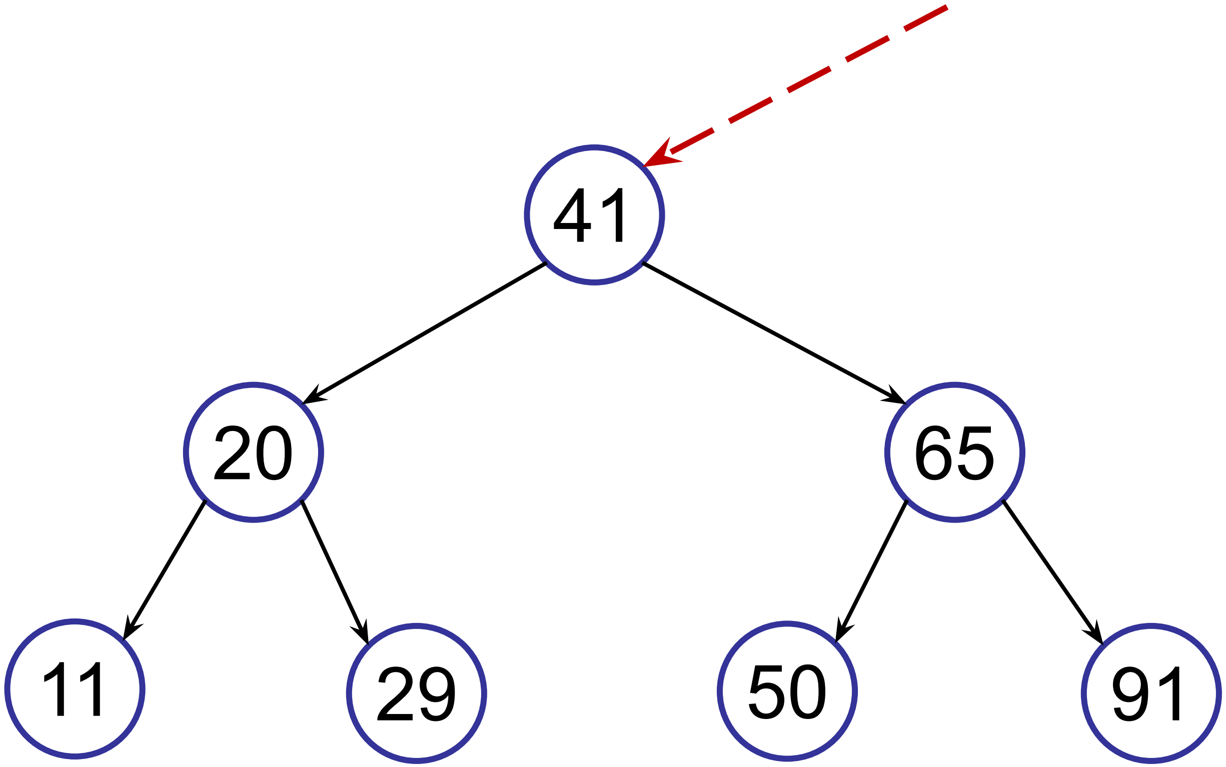
Binary Search Trees

searchMax()



Binary Search Trees

Search for the minimum key:



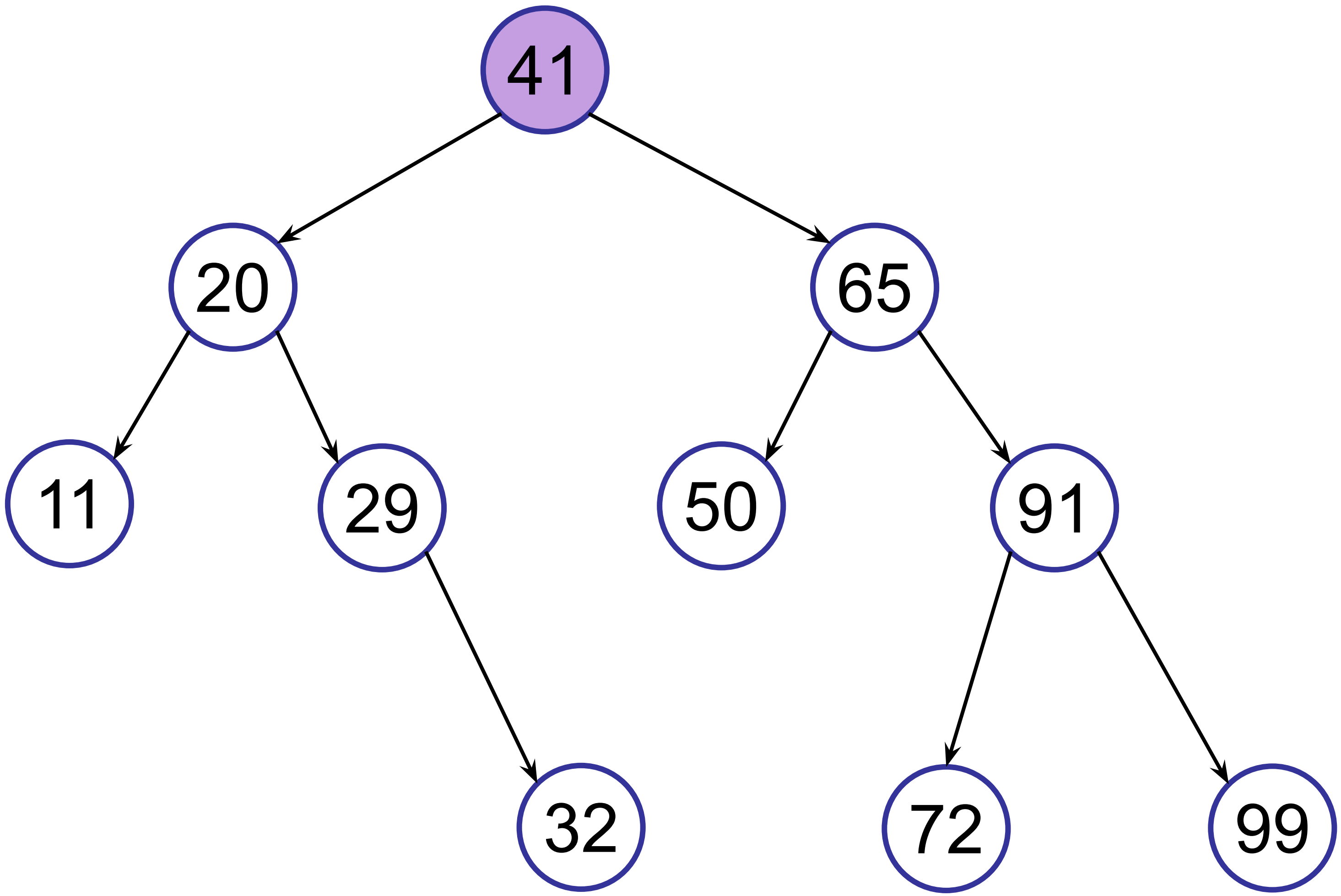
Binary Tree

Searching for the node with the minimum key

```
public TreeNode searchMin(){
    if (leftTree != null) {
        return leftTree.searchMin();
    }
    else return this; // Key is here!
}
```

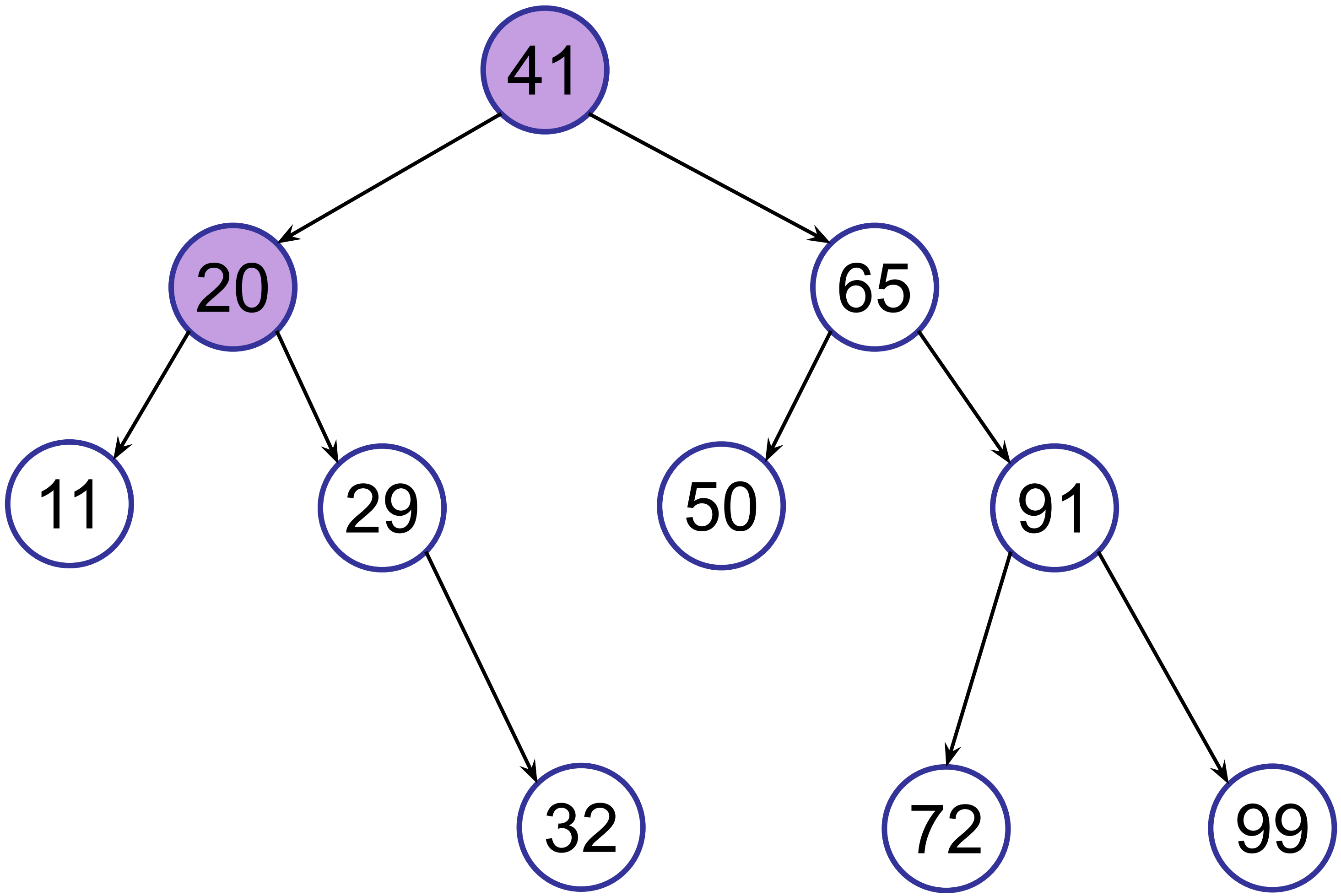
Binary Search Trees

searchMin()



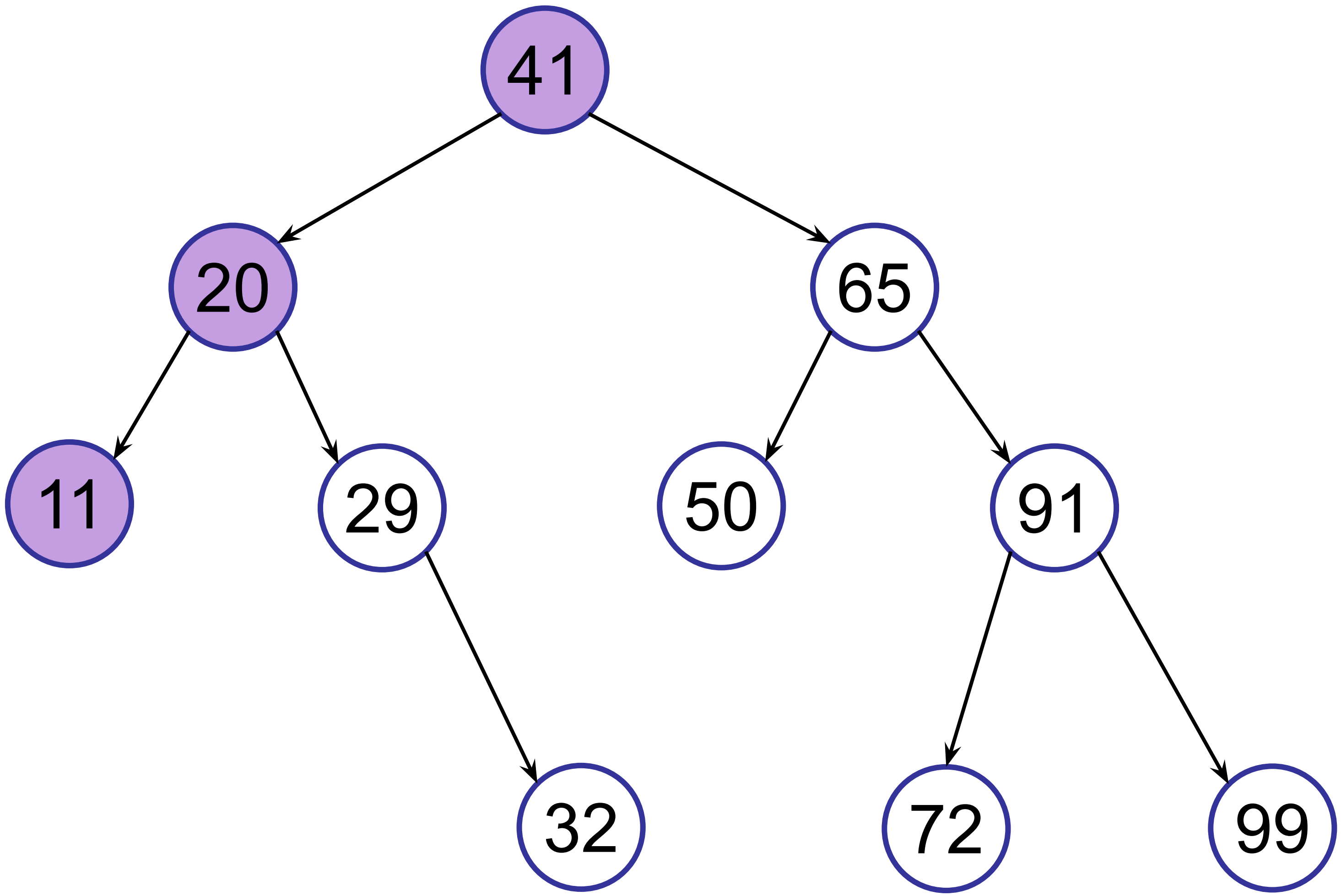
Binary Search Trees

searchMin()



Binary Search Trees

searchMin()



Binary Search Trees

1. Terminology and Definitions

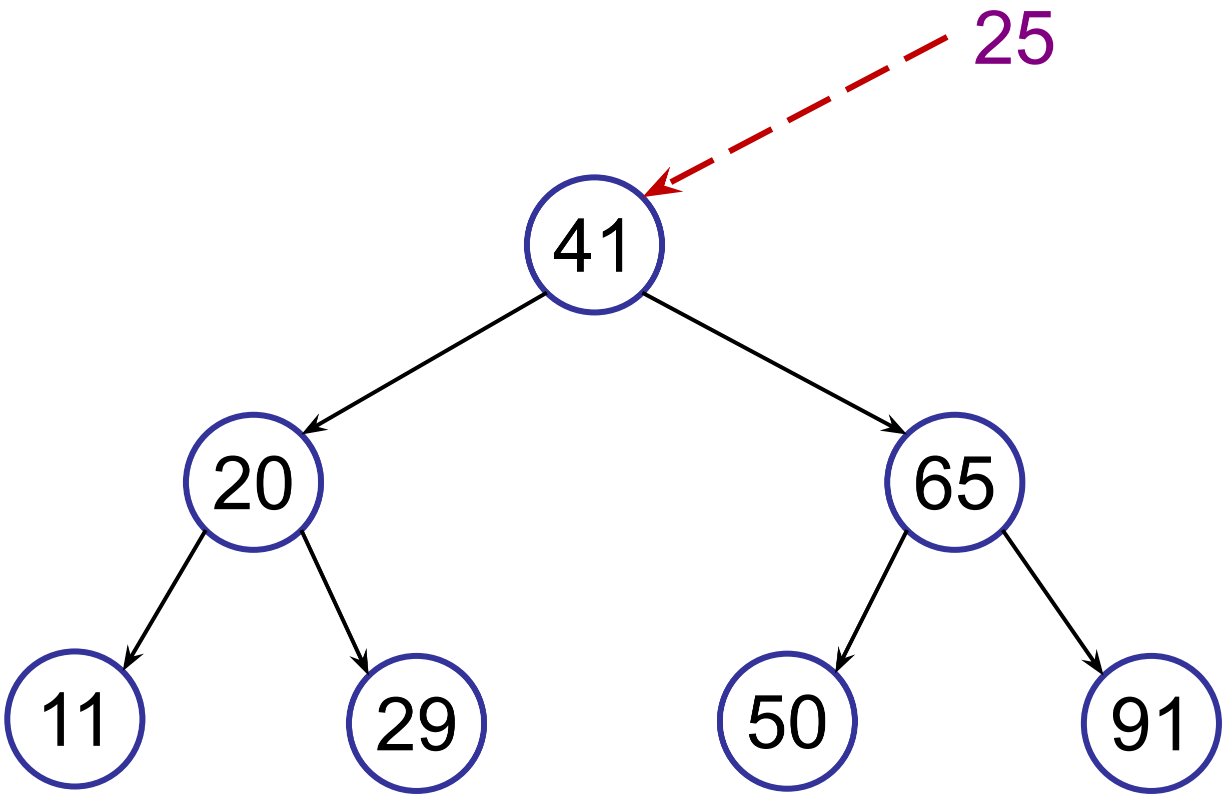
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

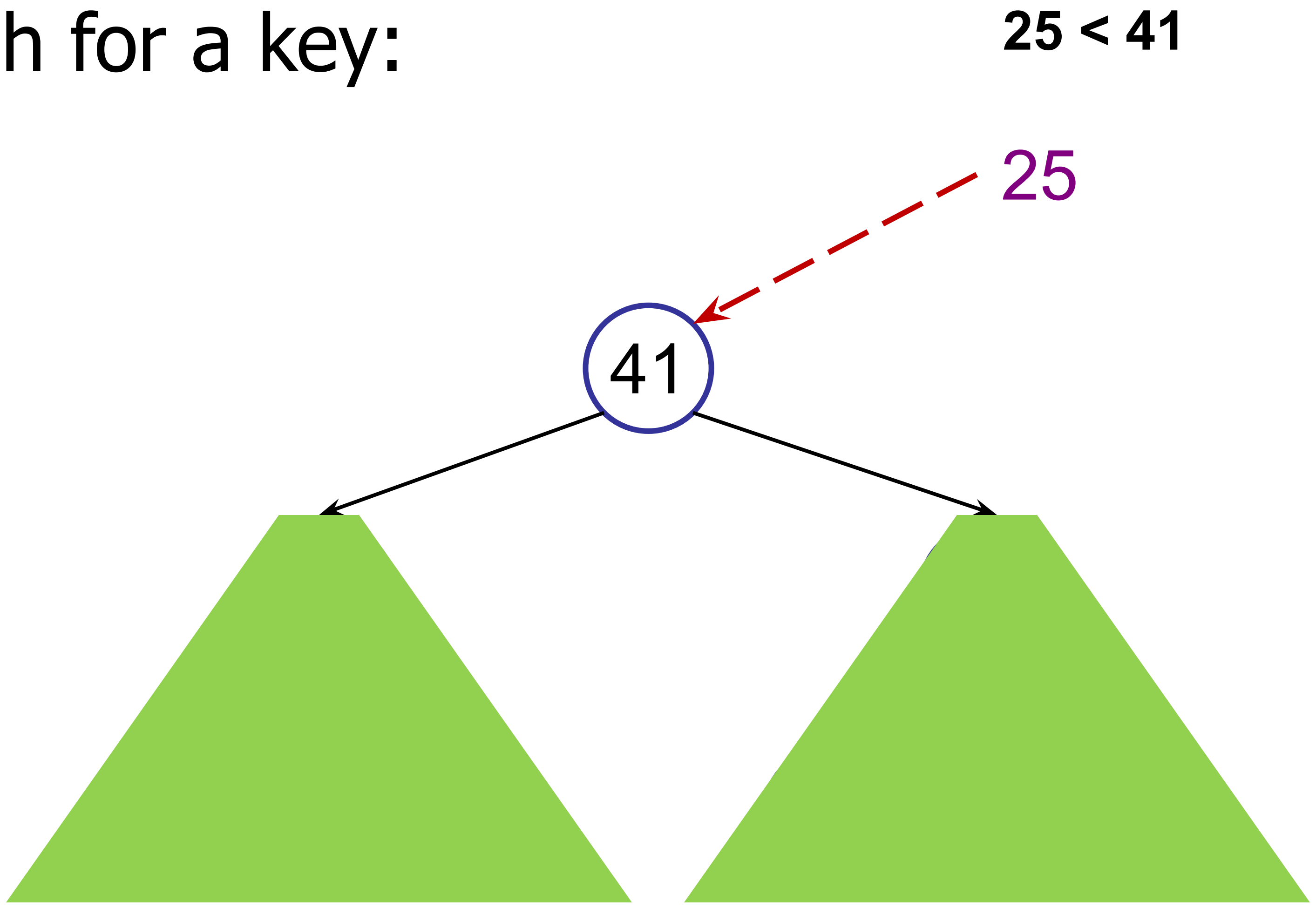
Binary Search Trees

Search for a key:



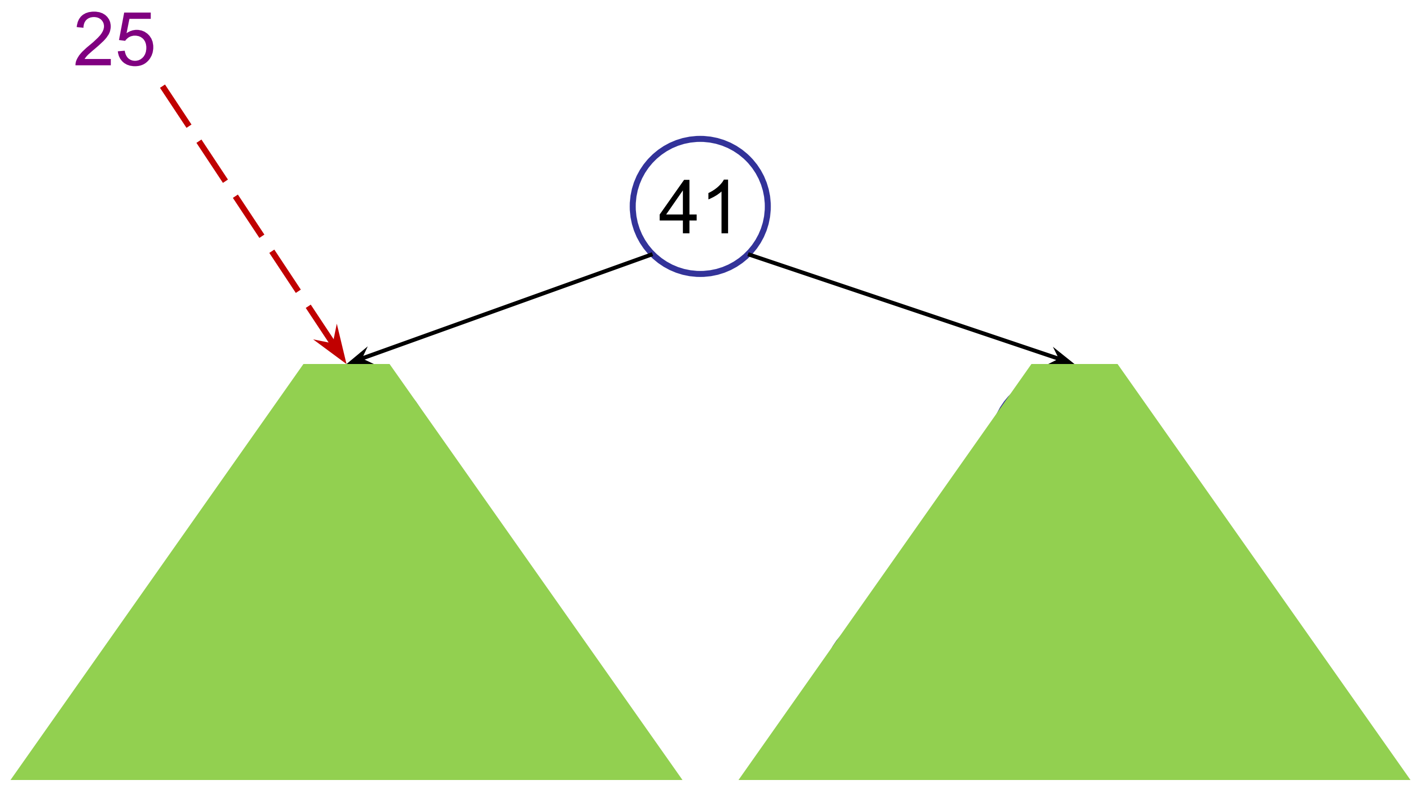
Binary Search Trees

Search for a key:



Binary Search Trees

Search for a key:



Binary Tree

Inserting a new key

```
public TreeNode search(int queryKey){
    if (queryKey < key) {
        if (leftTree != null)
            return leftTree.search(key);
        else return null;
    }
    else if (queryKey > key) {
        if (rightTree != null)
            return rightTree.search(key);
        else return null;
    }
    else return this; // Key is here!
}
```

Binary Tree

Inserting a new key

```
public TreeNode search(int queryKey){
    if (queryKey < key) {
        if (leftTree != null)
            return leftTree.search(key);
        else return null;
    }
    else if (queryKey > key) {
        if (rightTree != null)
            return rightTree.search(key);
        else return null;
    }
    else return this; // Key is here!
}
```

Binary Tree

Inserting a new key

```
public TreeNode search(int queryKey){
    if (queryKey < key) {
        if (leftTree != null)
            return leftTree.search(key);
        else return null;
    }
    else if (queryKey > key) {
        if (rightTree != null)
            return rightTree.search(key);
        else return null;
    }
    else return this; // Key is here!
}
```

Binary Tree

Inserting a new key

```
public TreeNode search(int queryKey){
    if (queryKey < key) {
        if (leftTree != null)
            return leftTree.search(key);
        else return null;
    }
    else if (queryKey > key) {
        if (rightTree != null)
            return rightTree.search(key);
        else return null;
    }
    else return this; // Key is here!
}
```

Binary Tree

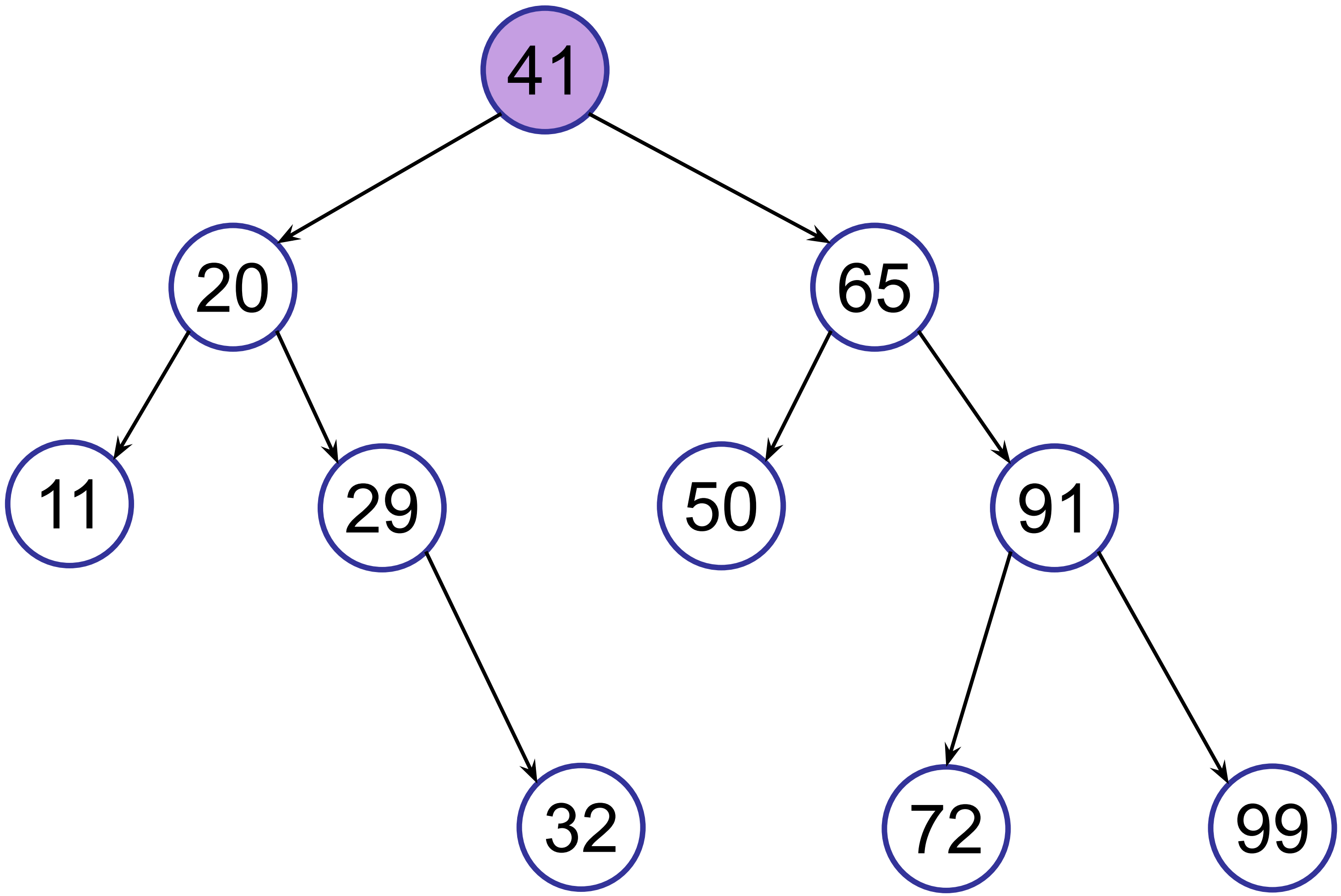
Inserting a new key

```
public TreeNode search(int queryKey){
    if (queryKey < key) {
        if (leftTree != null)
            return leftTree.search(key);
        else return null;
    }
    else if (queryKey > key) {
        if (rightTree != null)
            return rightTree.search(key);
        else return null;
    }
    else return this; // Key is here!
}
```

If we have no more sub-tree to recurse on, the key doesn't exist.

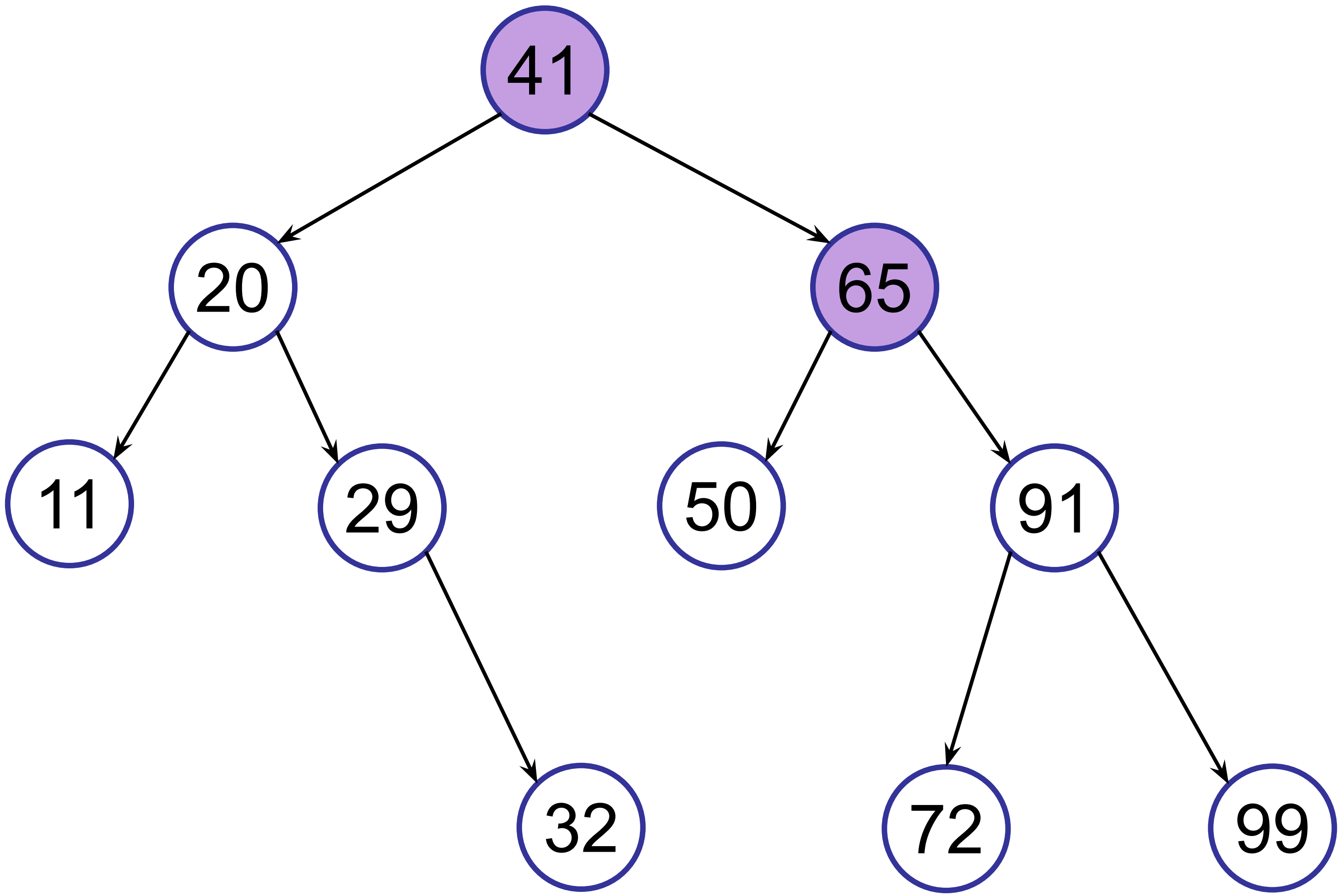
Binary Search Trees

search(72)



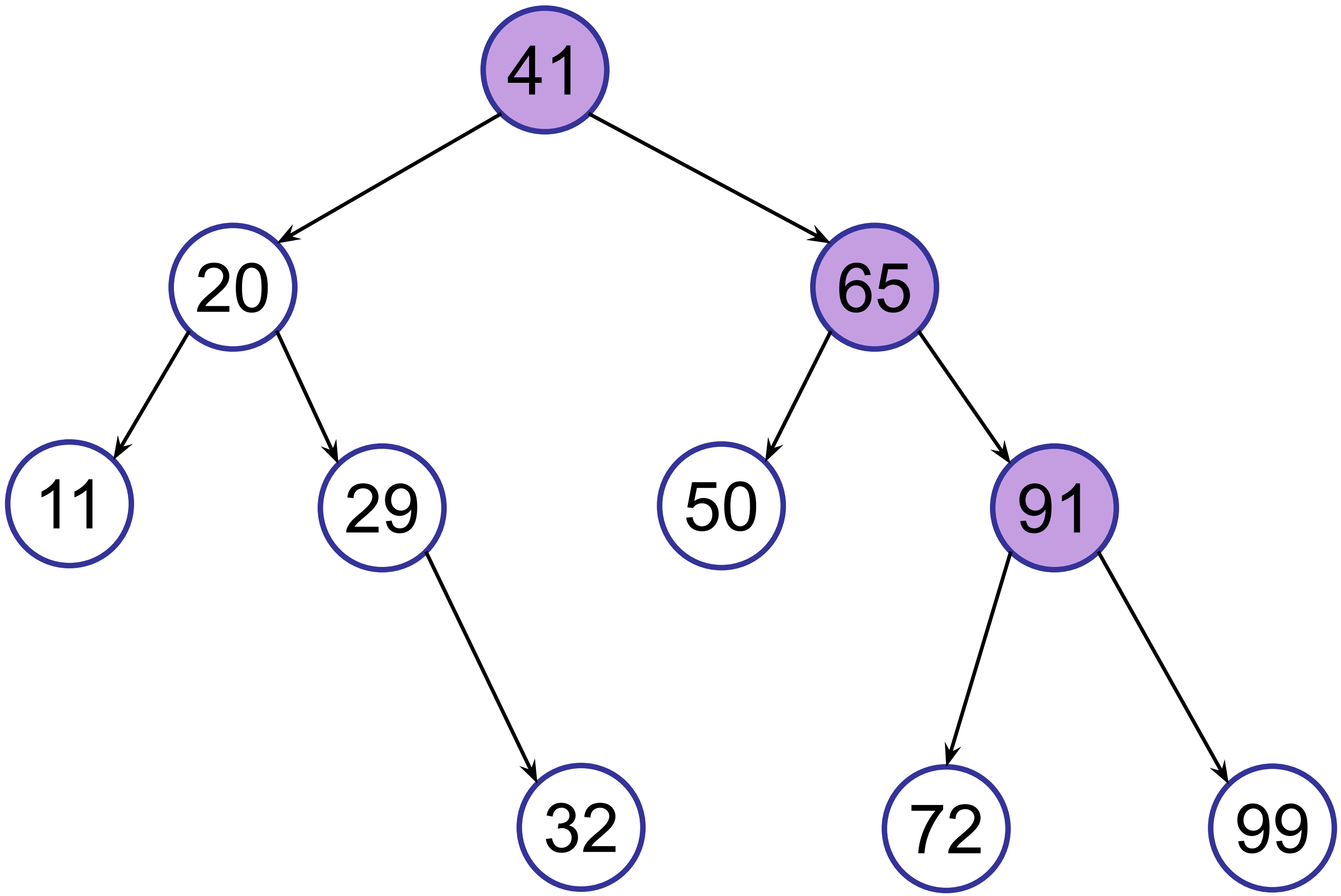
Binary Search Trees

search(72)



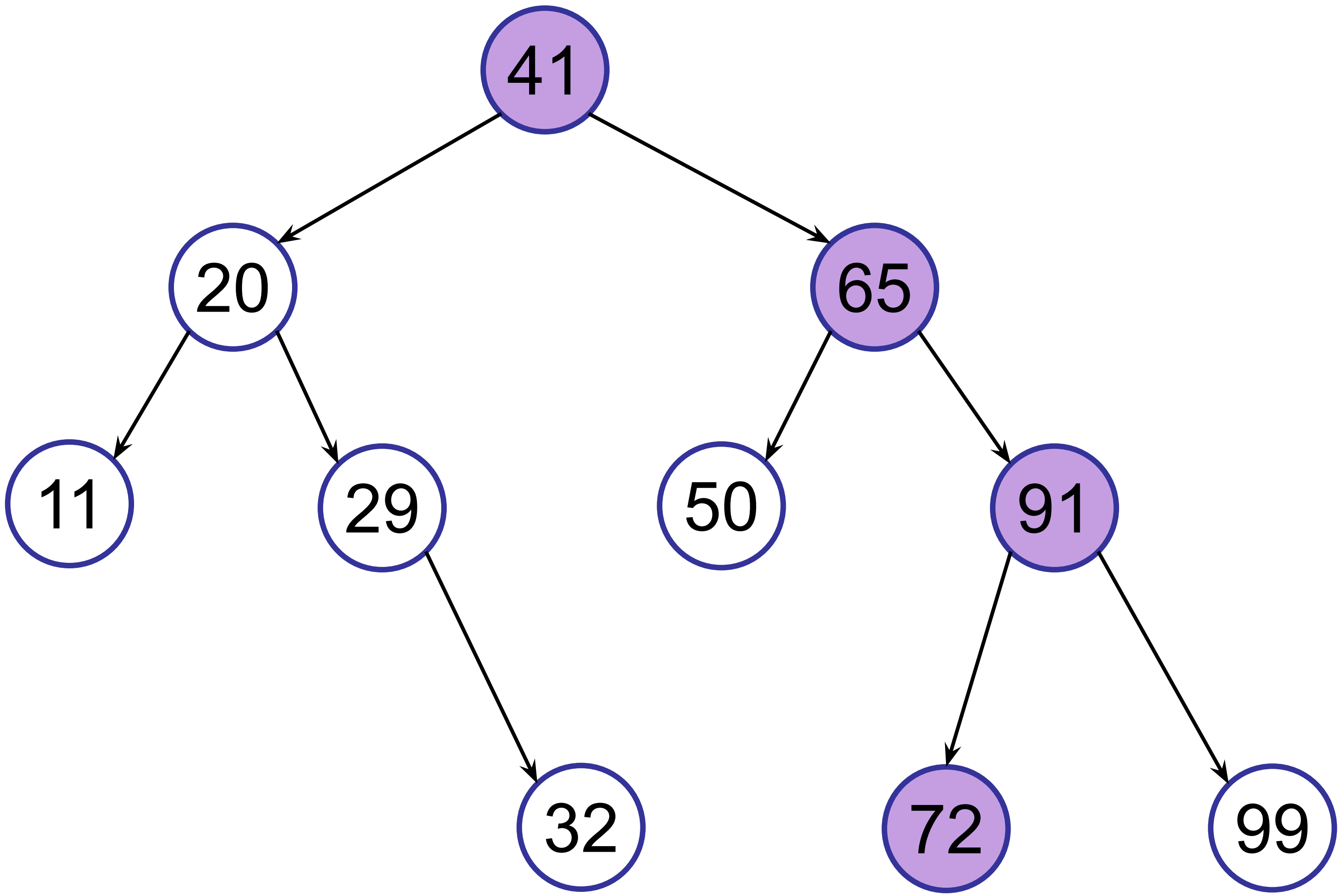
Binary Search Trees

search(72)



Binary Search Trees

search(72)



Binary Search Trees

1. Terminology and Definitions

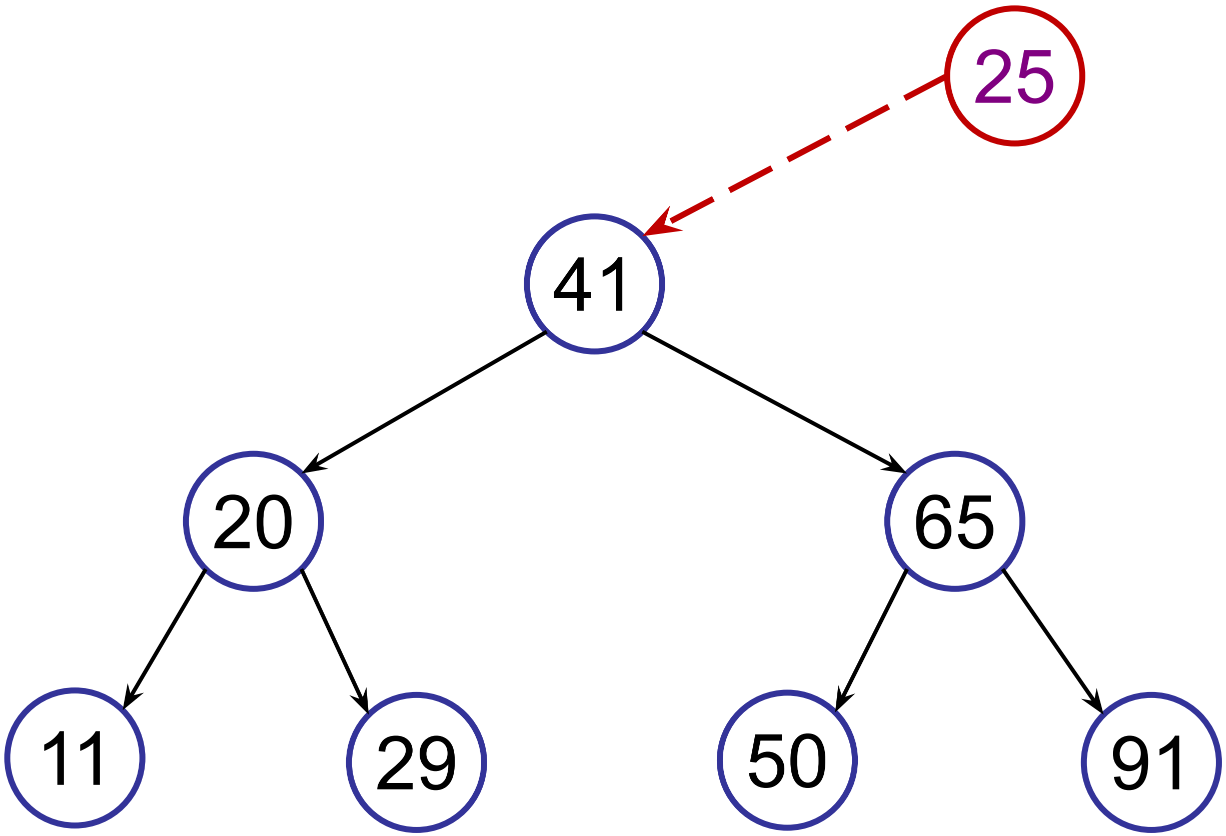
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

Binary Search Trees

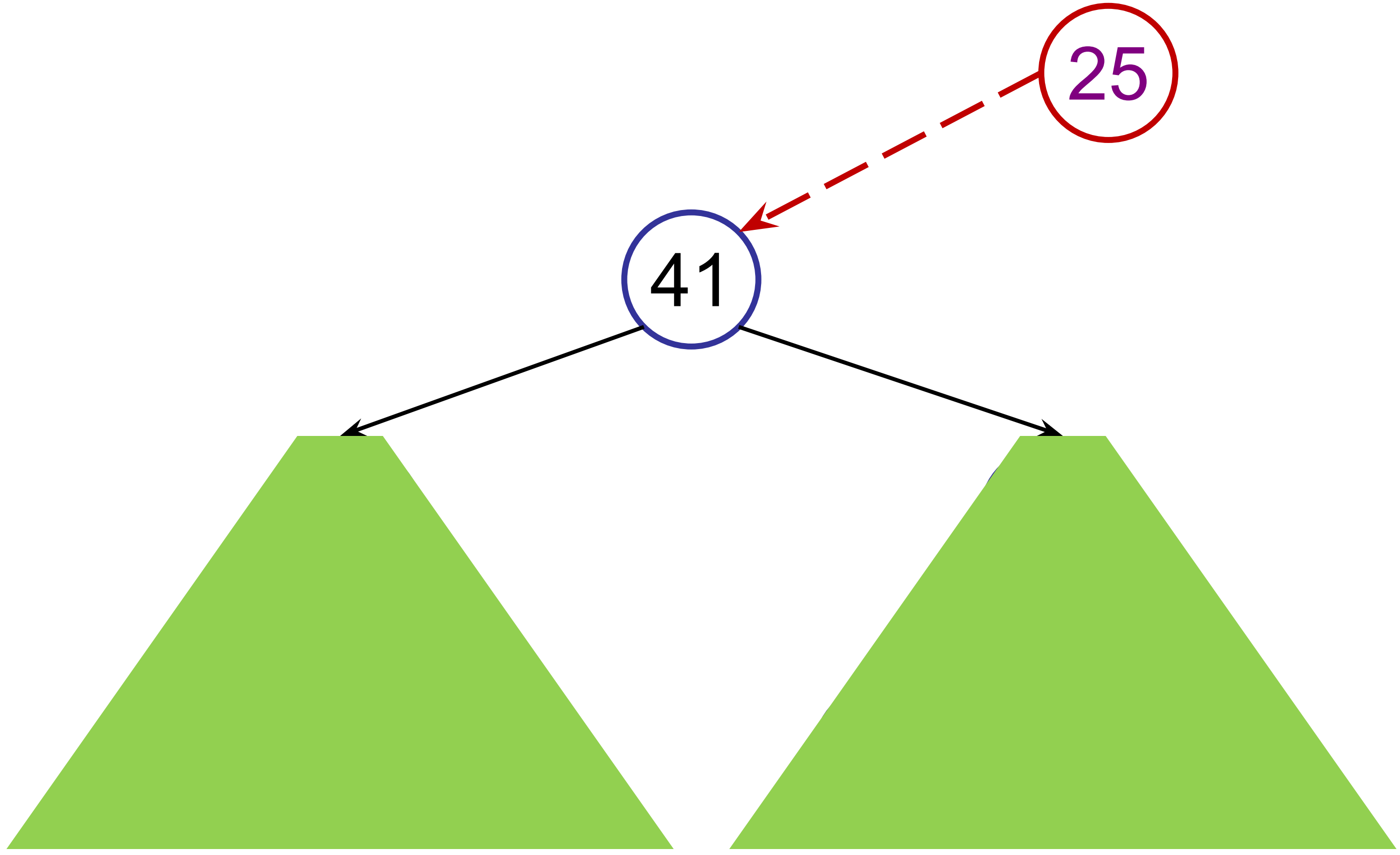
Inserting a new key:



Binary Search Trees

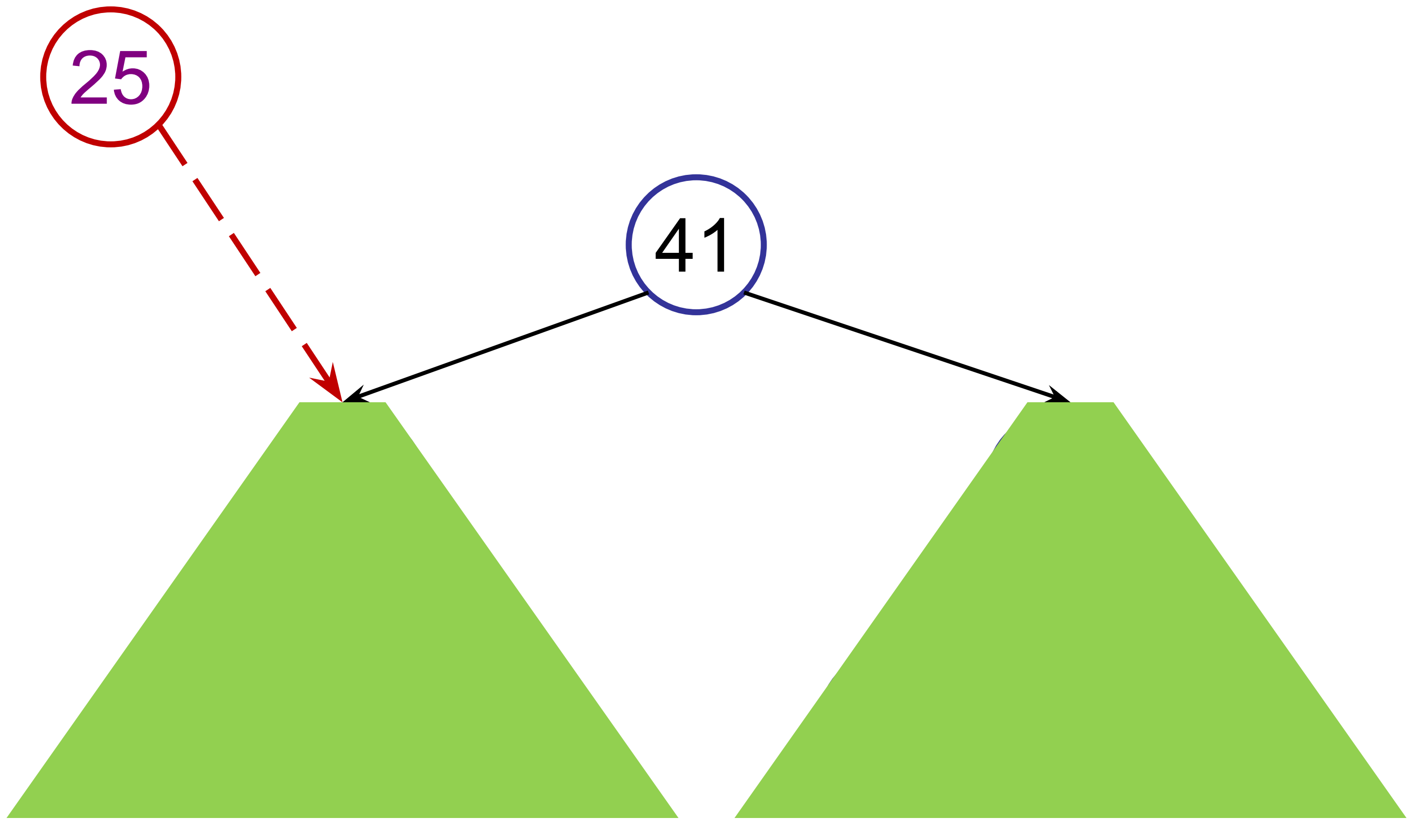
25 < 41

Inserting a new key:



Binary Search Trees

Inserting a new key:



Binary Tree

Inserting a new key

```
public void insert(int insKey, int intValue){
    if (insKey < key) {
        if (leftTree != null)
            leftTree.insert(insKey);
        else leftTree = new TreeNode(insKey,insValue);
    }
    else if (insKey > key) {
        if (rightTree != null)
            rightTree.insert(insKey);
        else rightTree = new TreeNode(insKey,insValue);
    }
    else return; // Key is already in the tree!
}
```

Binary Tree

Inserting a new key

```
public void insert(int insKey, int intValue){
    if (insKey < key) {
        if (leftTree != null)
            leftTree.insert(insKey);
        else leftTree = new TreeNode(insKey,insValue);
    }
    else if (insKey > key) {
        if (rightTree != null)
            rightTree.insert(insKey);
        else rightTree = new TreeNode(insKey,insValue);
    }
    else return; // Key is already in the tree!
}
```

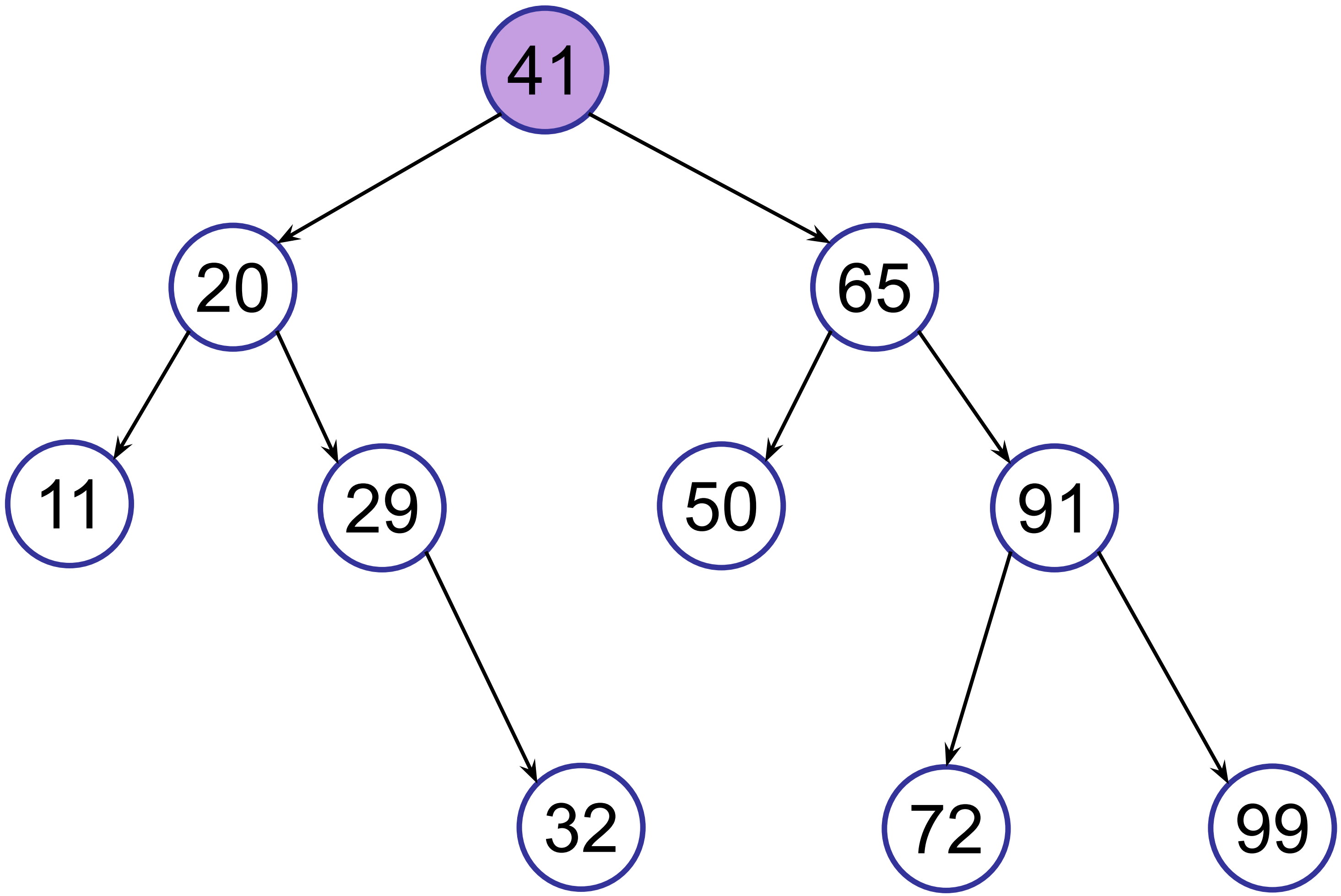
Binary Tree

Inserting a new key

```
public void insert(int insKey, int intValue){
    if (insKey < key) {
        if (leftTree != null)
            leftTree.insert(insKey);
        else leftTree = new TreeNode(insKey,insValue);
    }
    else if (insKey > key) {
        if (rightTree != null)
            rightTree.insert(insKey);
        else rightTree = new TreeNode(insKey,insValue);
    }
    else return; // Key is already in the tree!
}
```

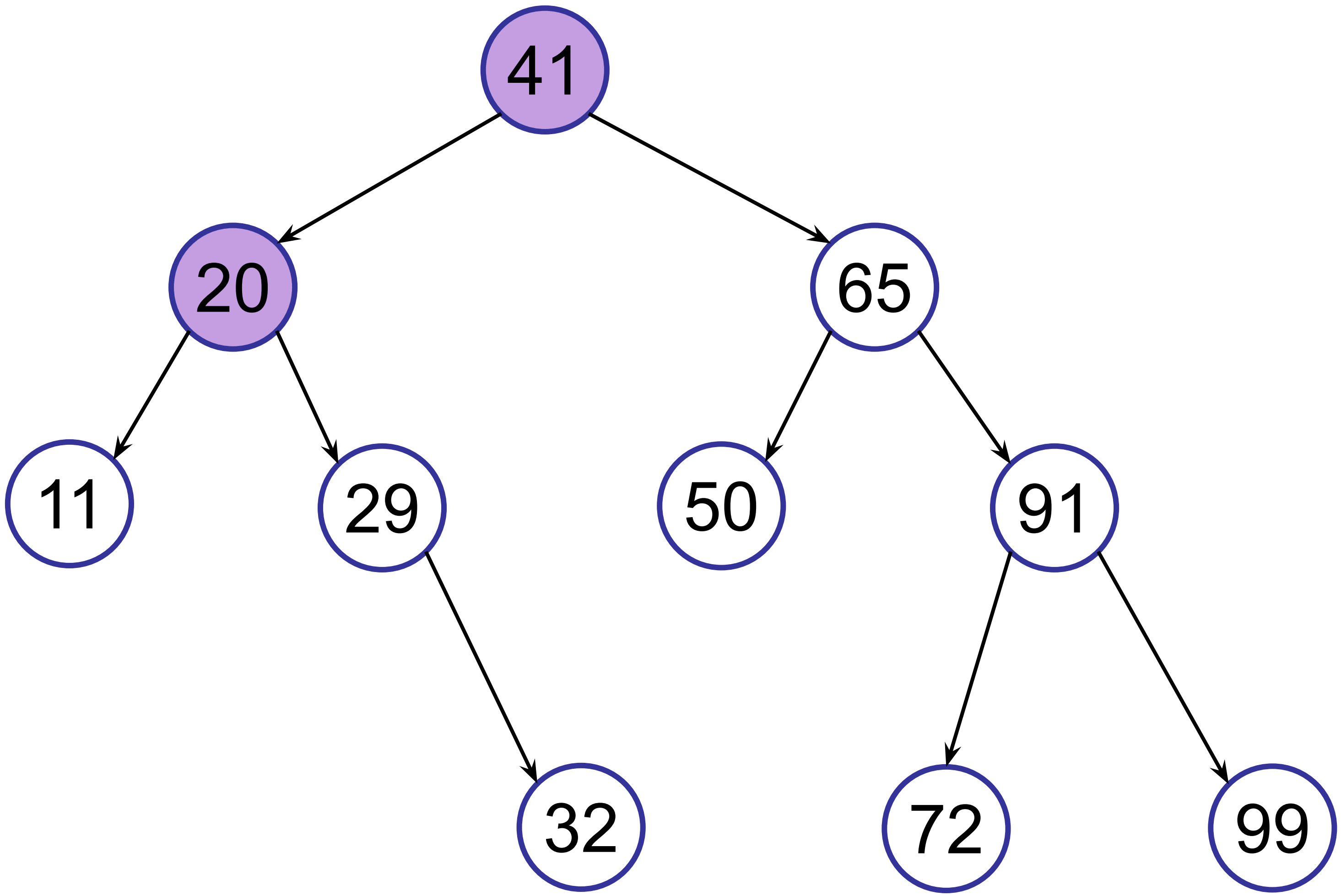
Binary Search Trees

insert(27)



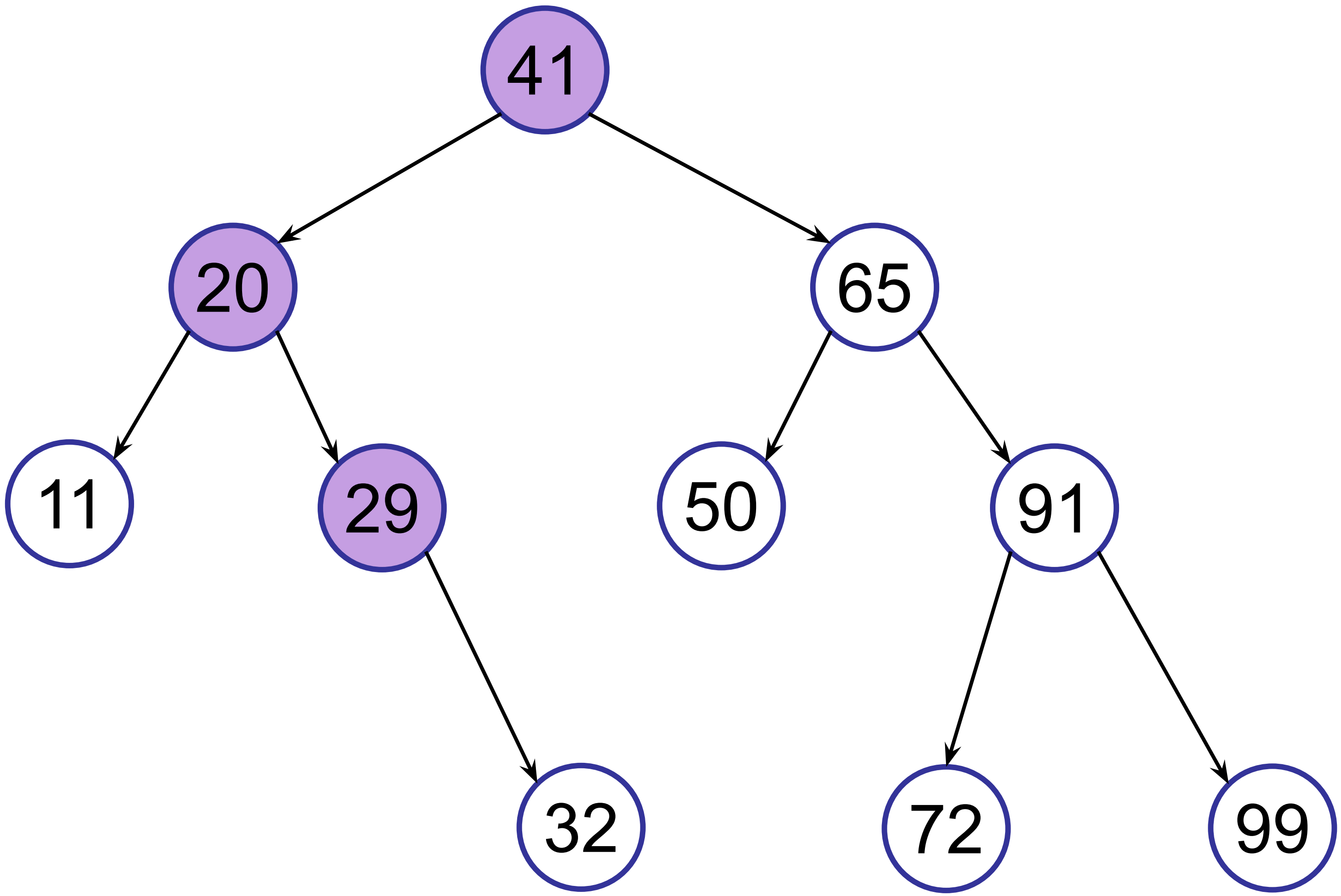
Binary Search Trees

insert(27)



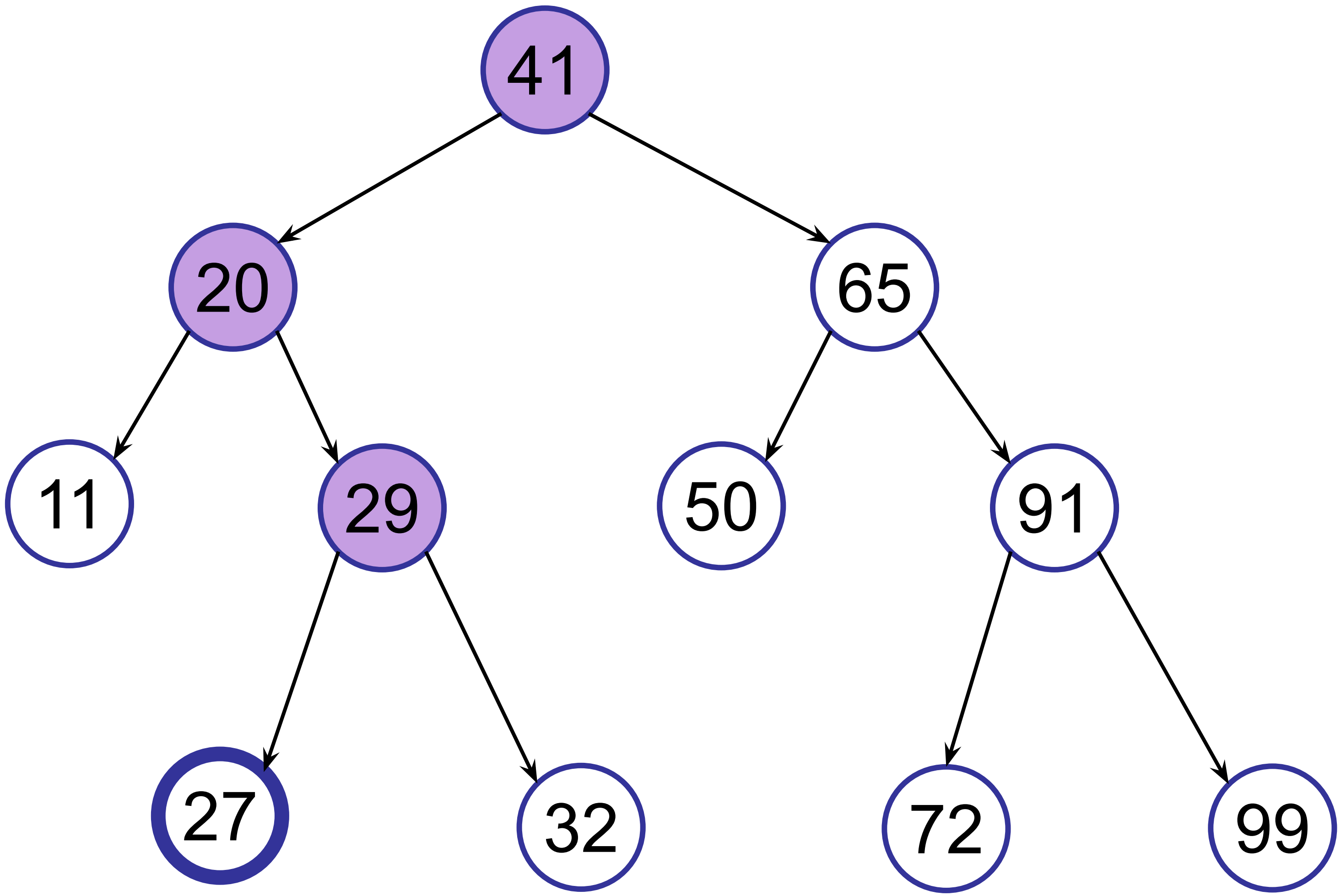
Binary Search Trees

insert(27)



Binary Search Trees

insert(27)



Binary Search Tree

What is the worst-case running time of search in a BST?

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

Binary Search Tree

What is the worst-case running time of search in a BST?

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

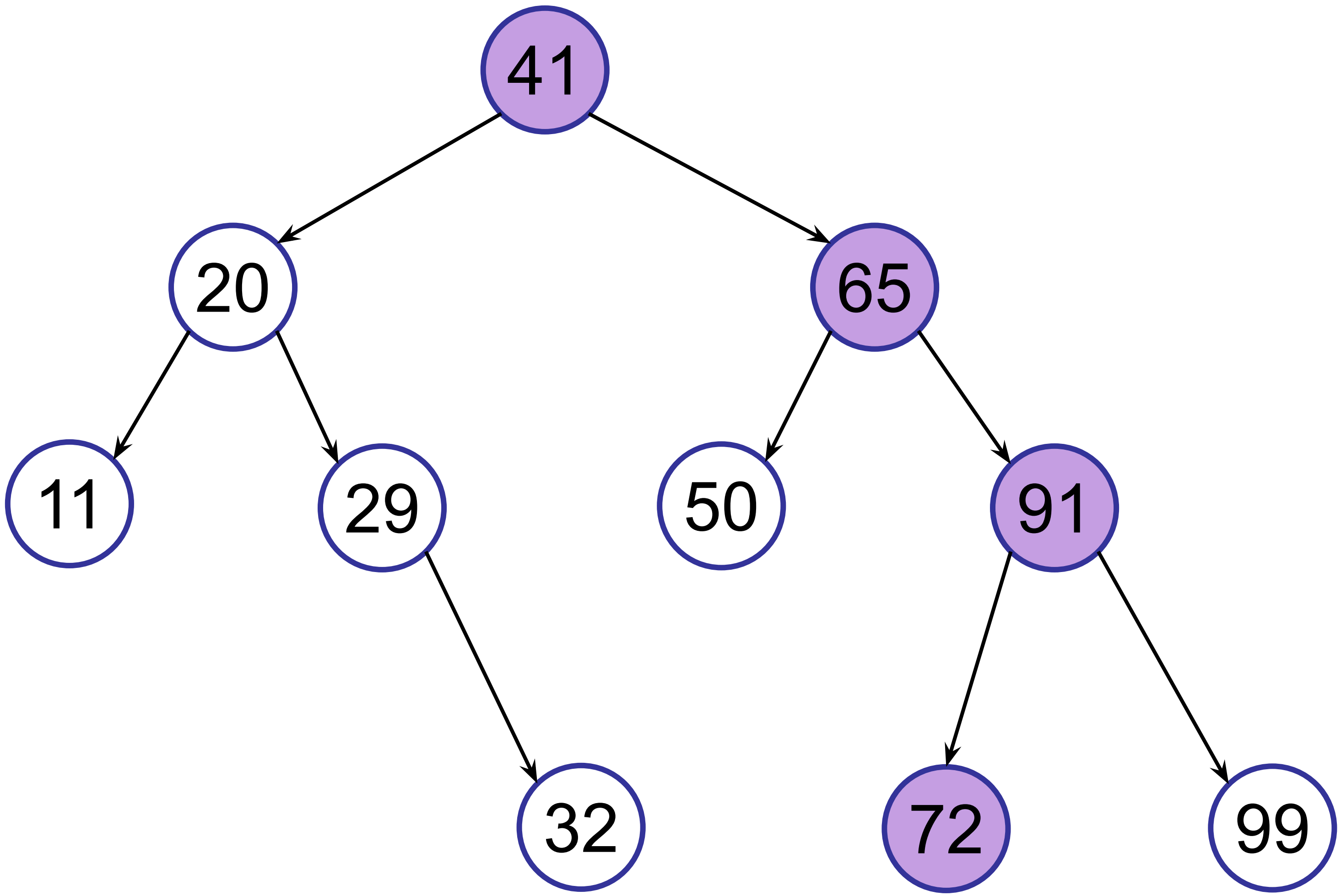
Binary Search Tree

What is the worst-case running time of search in a BST?

1. O(1)
2. O(log n) ???
3. O(n)
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

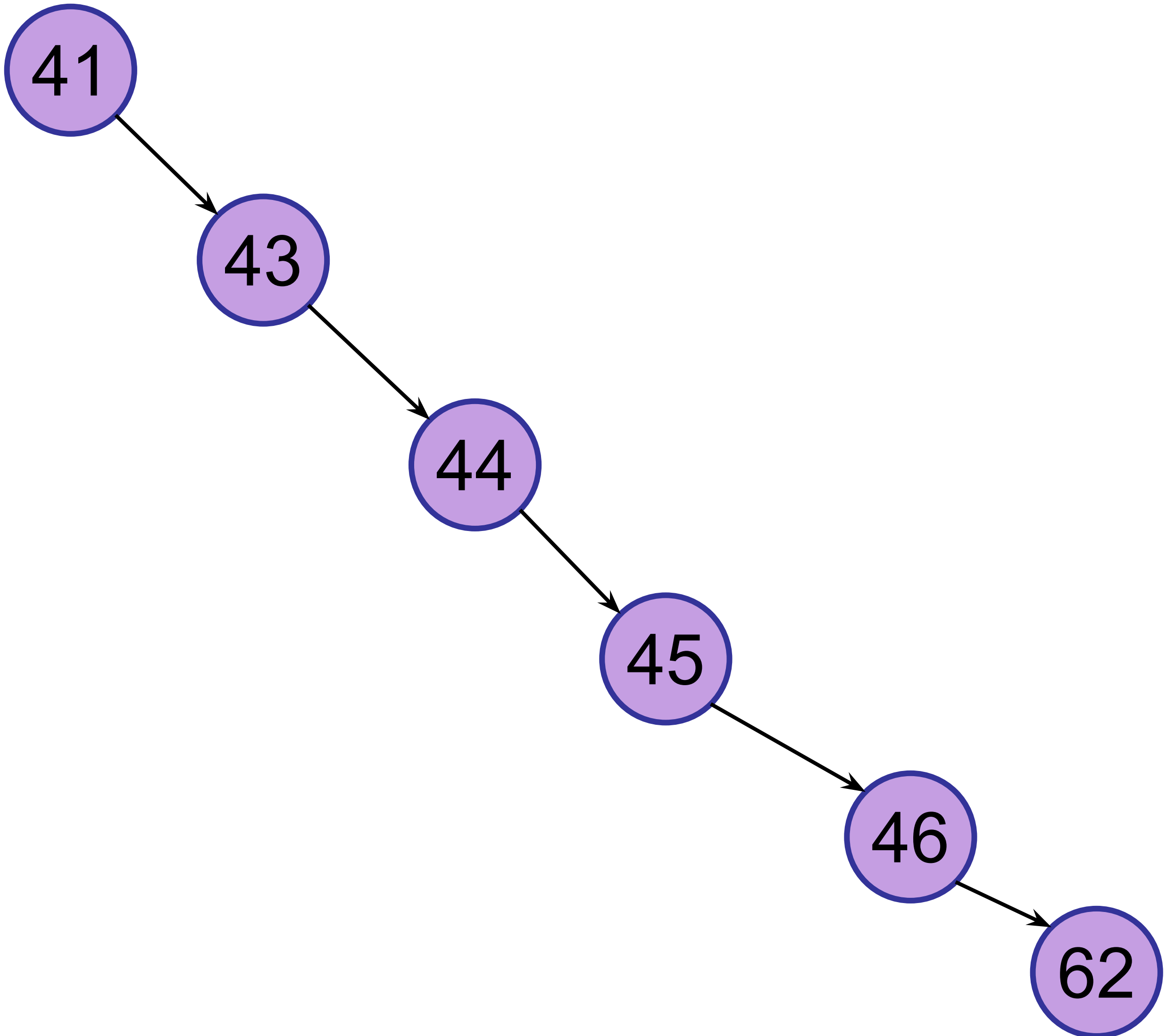
Binary Search Trees

search(72) : O(h)   h is the height of the tree



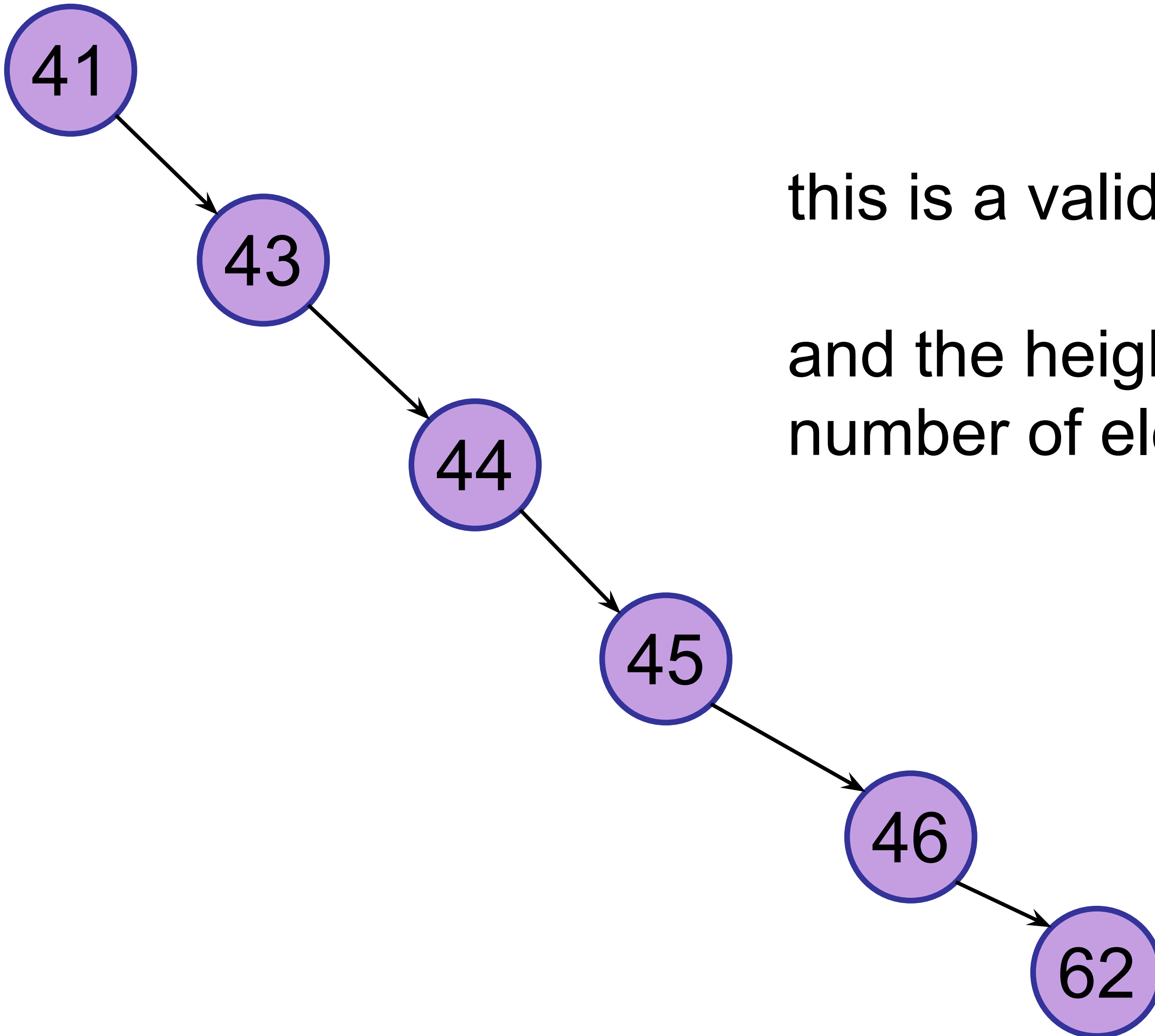
Binary Search Trees

search(72) : O(height)



Binary Search Trees

search(72) : O(height)



this is a valid BST

and the height + 1 =
number of elements

Binary Search Tree

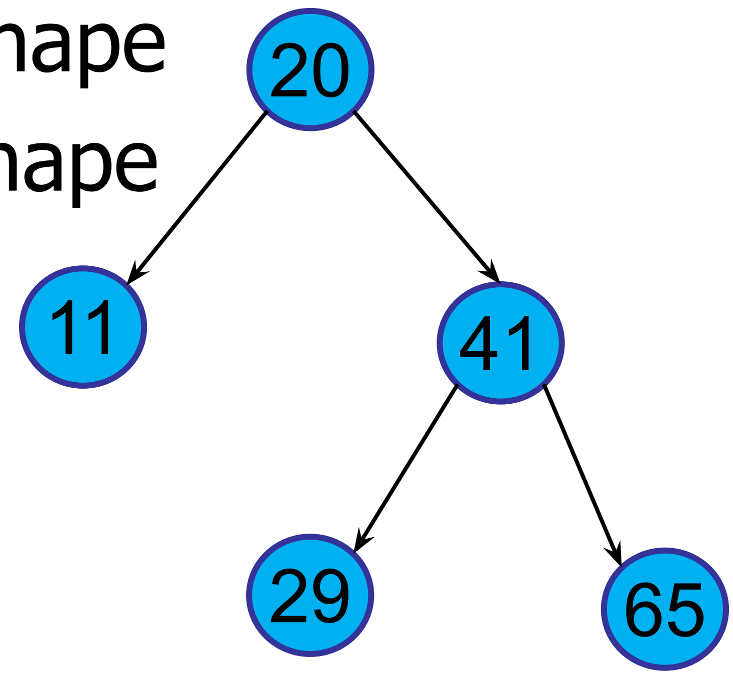
What is the worst-case running time of search in a BST?

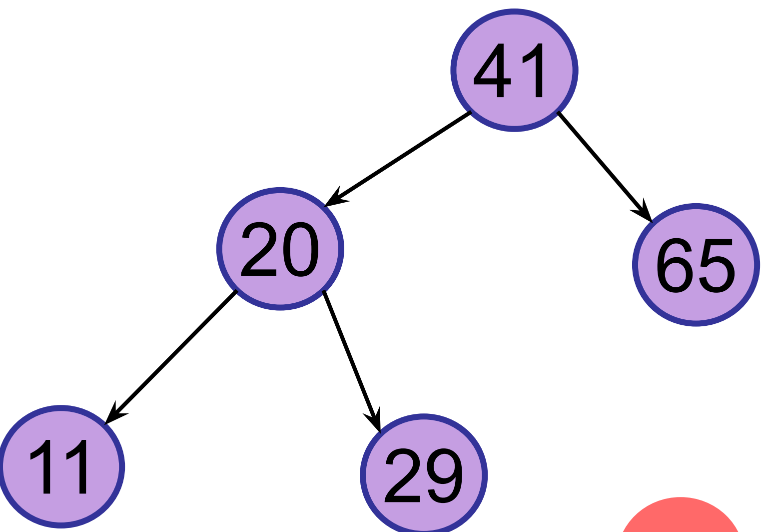
1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

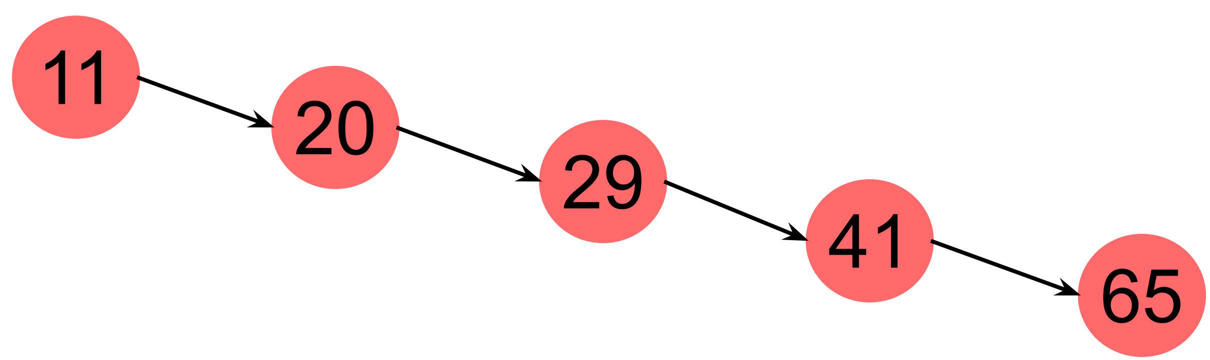
Tree Shape

Trees come in many shapes

- same set of keys ≠ same shape
- performance depends on shape



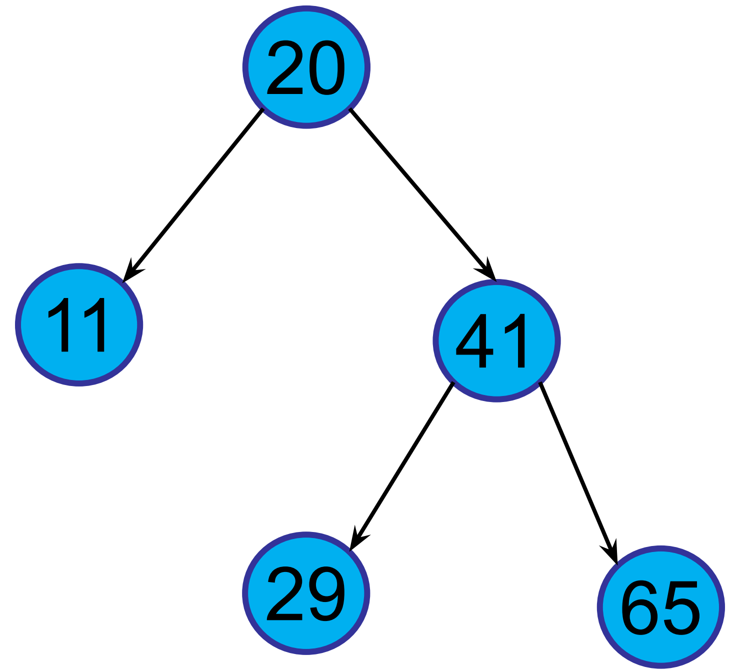


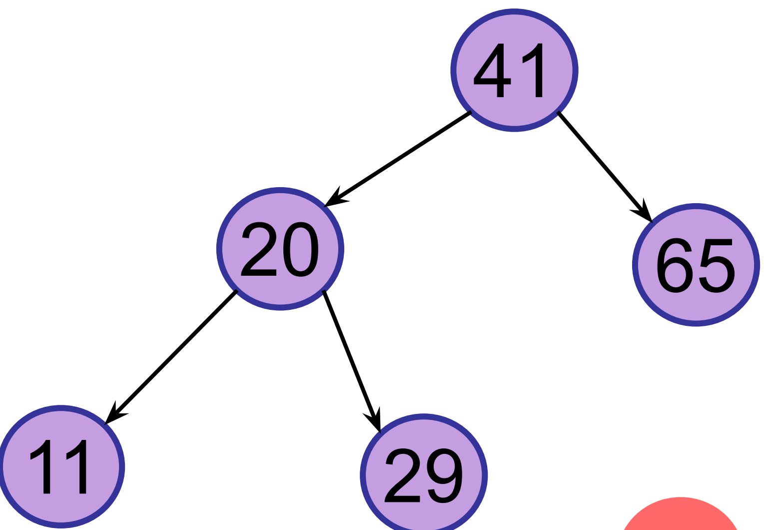


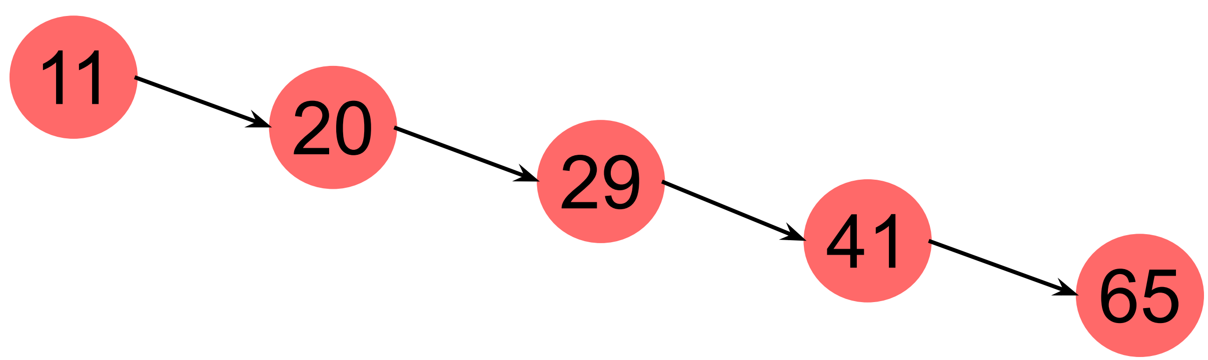
Tree Shape

What determines shape?

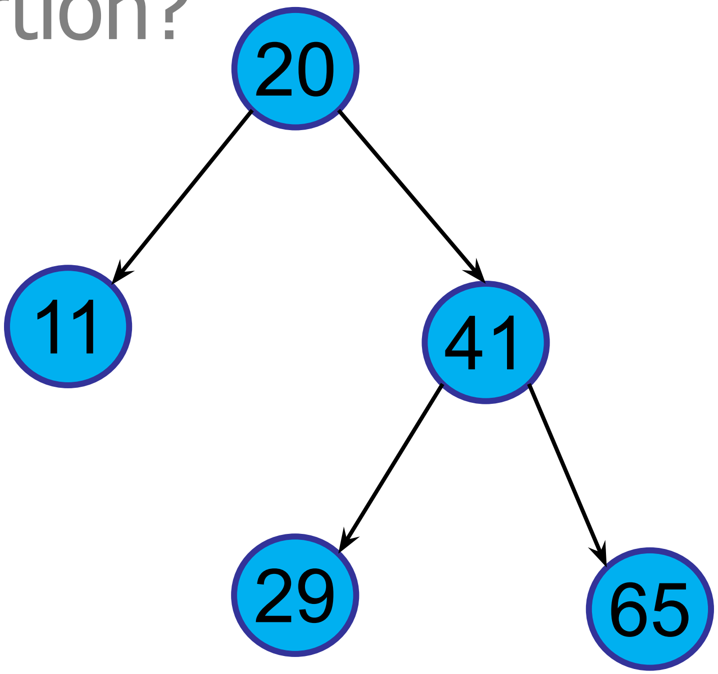
- Order of insertion of items





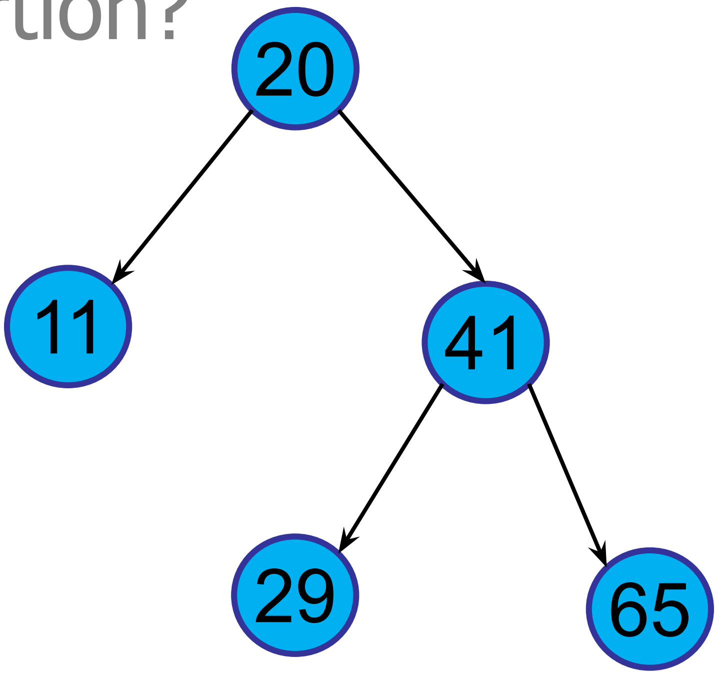


What was the order of insertion?



1. 11, 20, 29, 41, 65
2. 20, 11, 41, 29, 65
3. 11, 20, 41, 29, 65
4. 65, 41, 29, 20, 11
5. Impossible to tell.

What was the order of insertion?

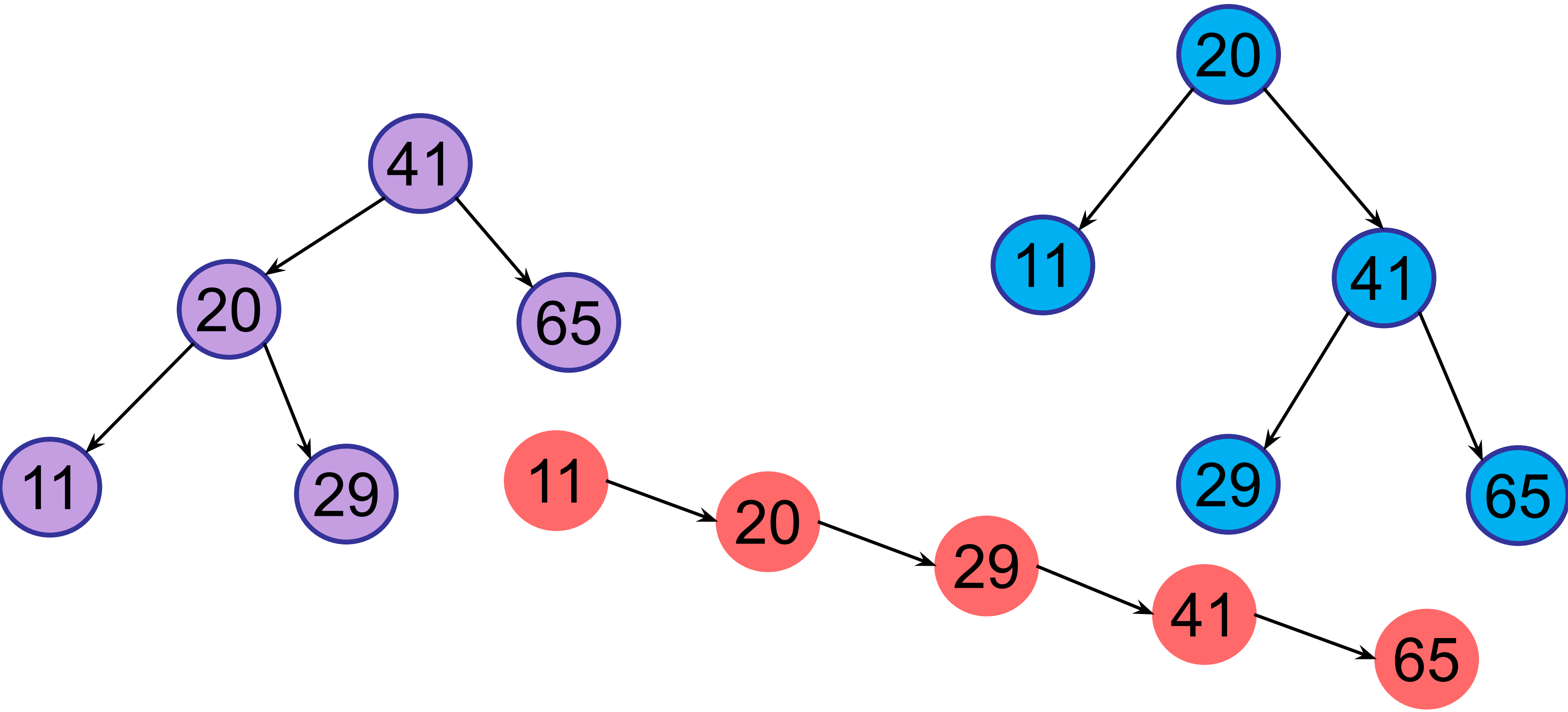


1. 11, 20, 29, 41, 65
2. 20, 11, 41, 29, 65
3. 11, 20, 41, 29, 65
4. 65, 41, 29, 20, 11
5. Impossible to tell.

Tree Shape

What determines shape?

- Order of insertion
- Does each order yield a unique shape?



Tree Shape

What determines shape?

- Order of insertion
- Does each order yield a unique shape? NO
  - # ways to order insertions: n!
  - # shapes of a binary tree? $\sim 4^n$

Catalan Numbers

Tree Shape

What determines shape?

- Order of insertion
- Does each order yield a unique shape? NO
  - # ways to order insertions: n!
  - # shapes of a binary tree? $\sim 4^n$

By Pigeonhole principle, this means that there exists at least 2 orderings that share the same shape.

Tree Shape

Catalan Numbers

C_n = # of trees with (n+1) nodes

C_n = # expressions with n pairs of matched parentheses

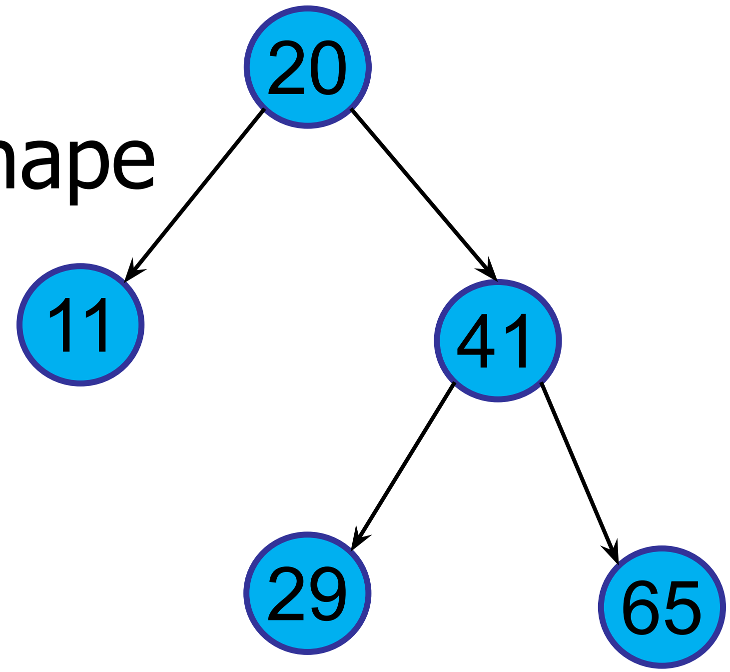
((()))   ()(())   (()())   (())()   ()()()

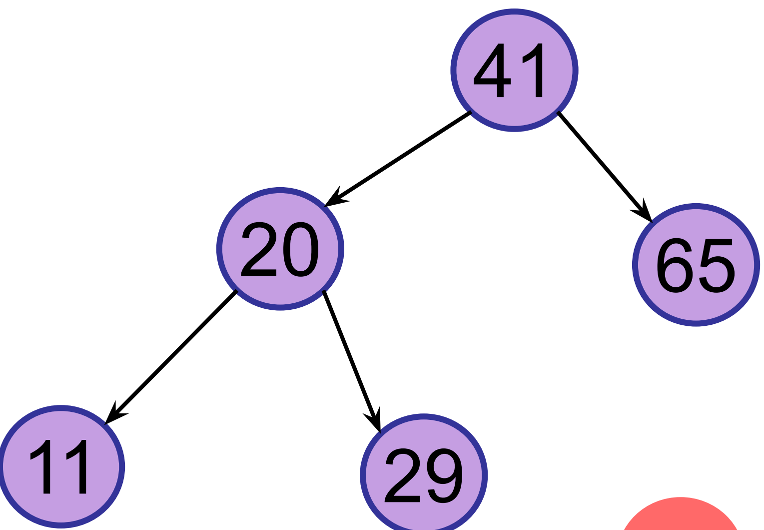
Puzzle: why are these the same?

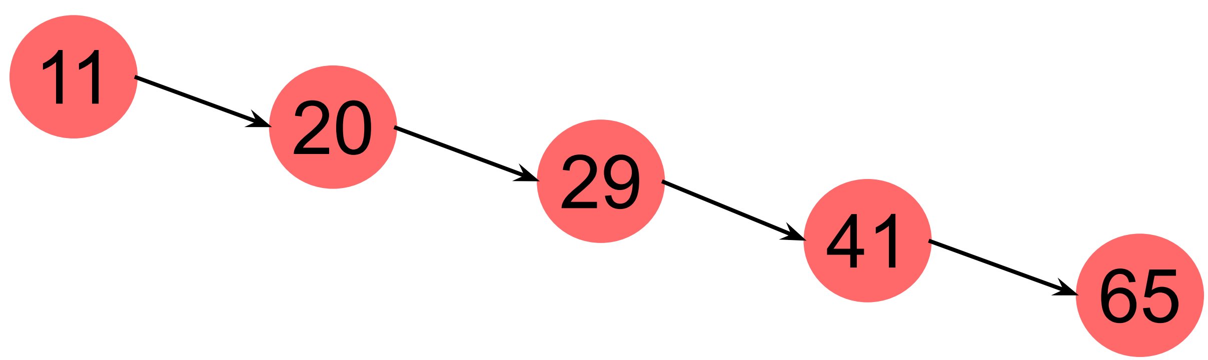
Tree Shape

Trees come in many shapes

- same keys ≠ same shape
- performance depends on shape



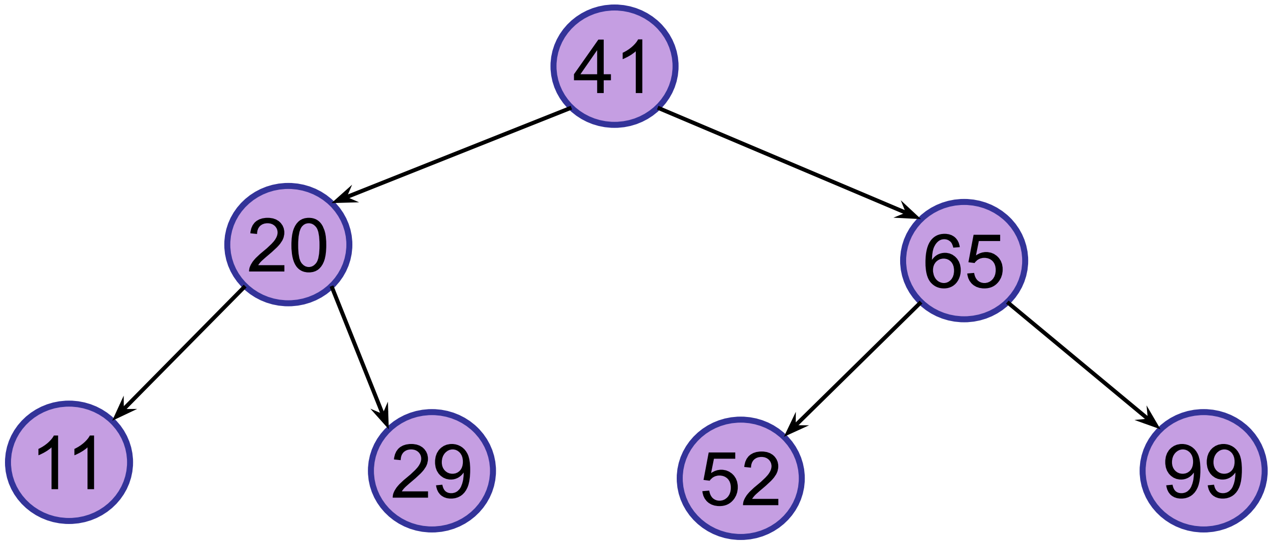




Tree Shape

Trees come in many shapes

- same keys $\neq$ same shape
- performance depends on shape
- insert keys in a *random* order $\Rightarrow$ balanced



Binary Search Trees

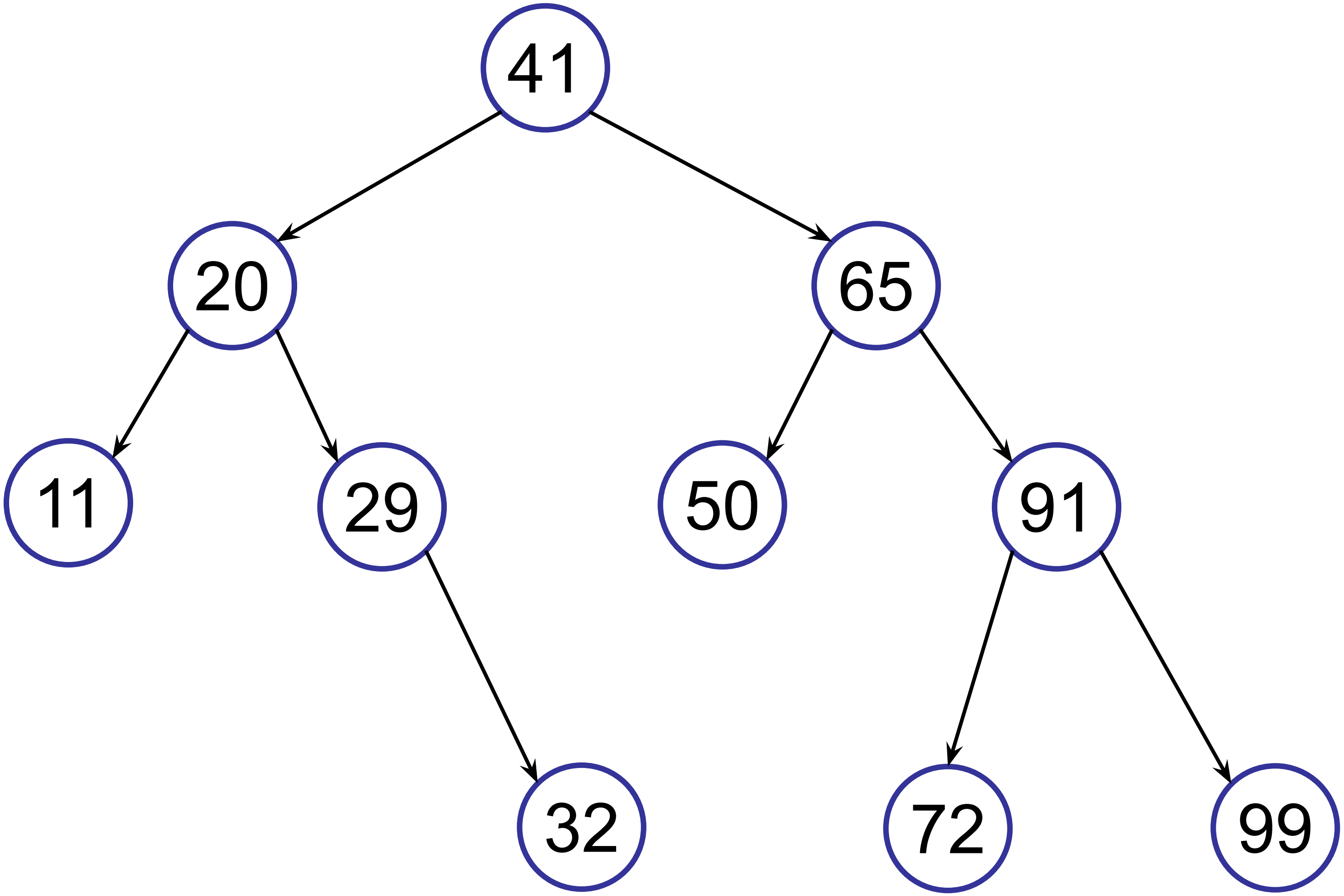
1. Terminology and Definitions

2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

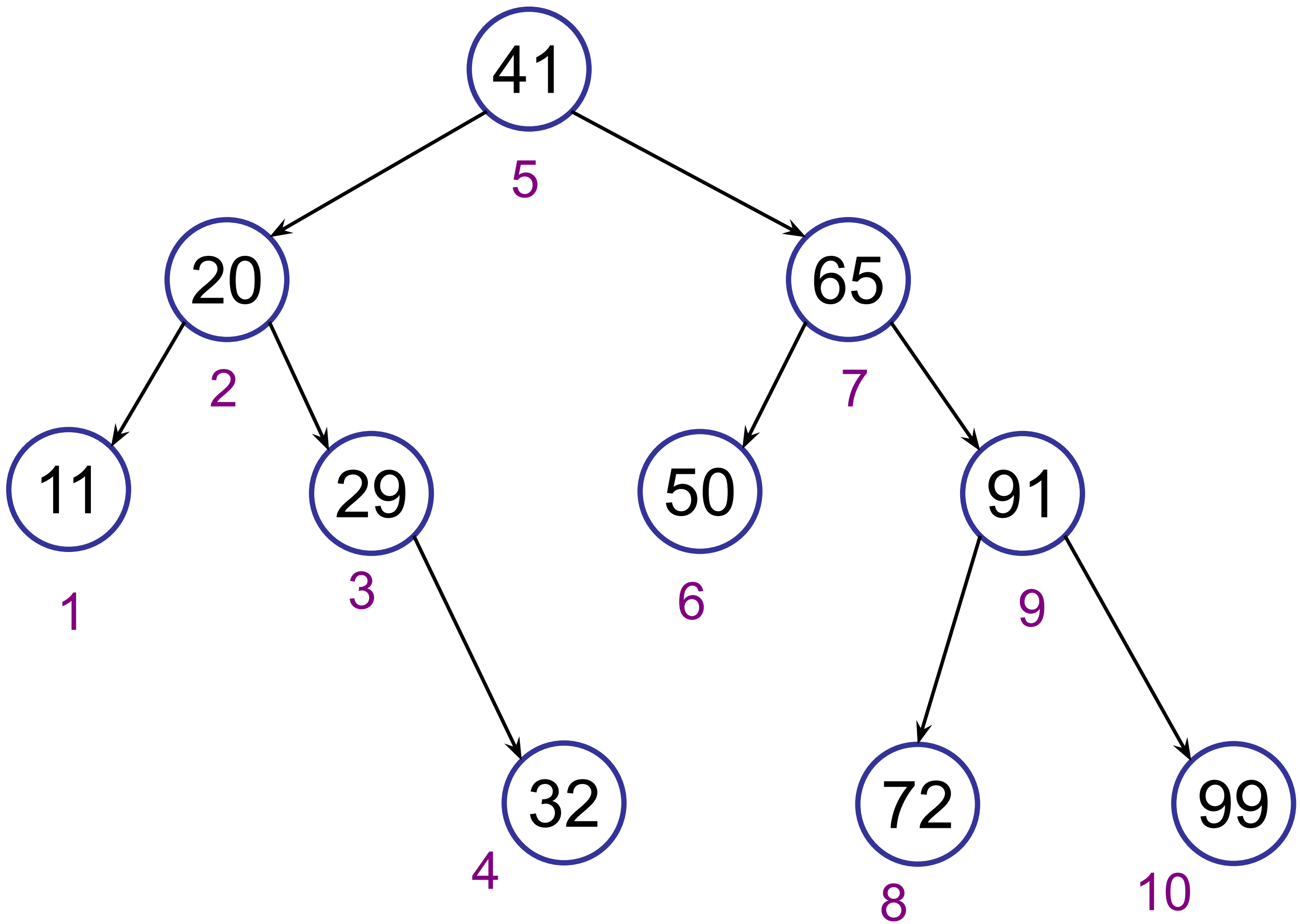
4. Other operations

Tree Traversal



11 20 29 32 41 50 65 72 91 99

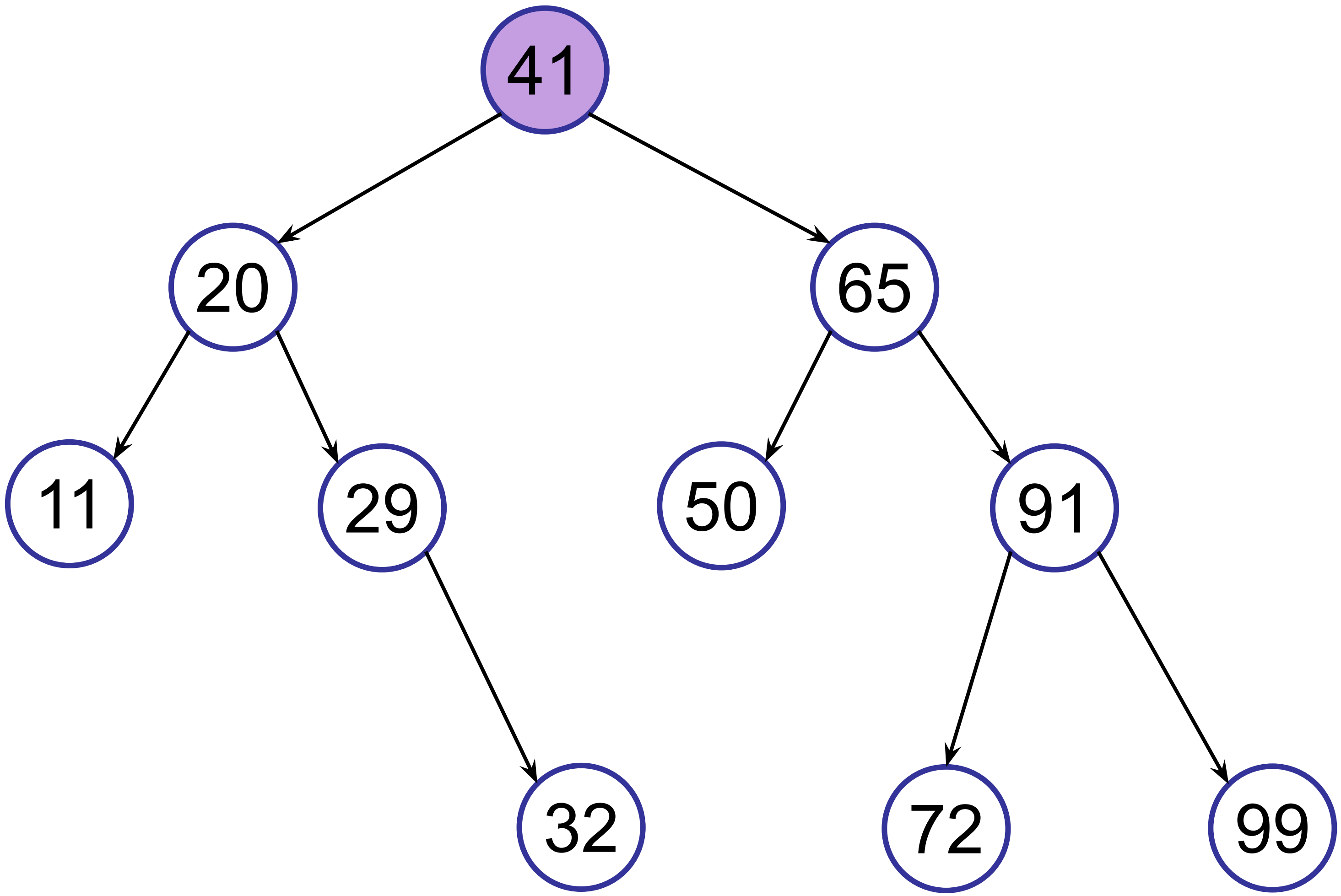
Tree Traversal



11 20 29 32 41 50 65 72 91 99

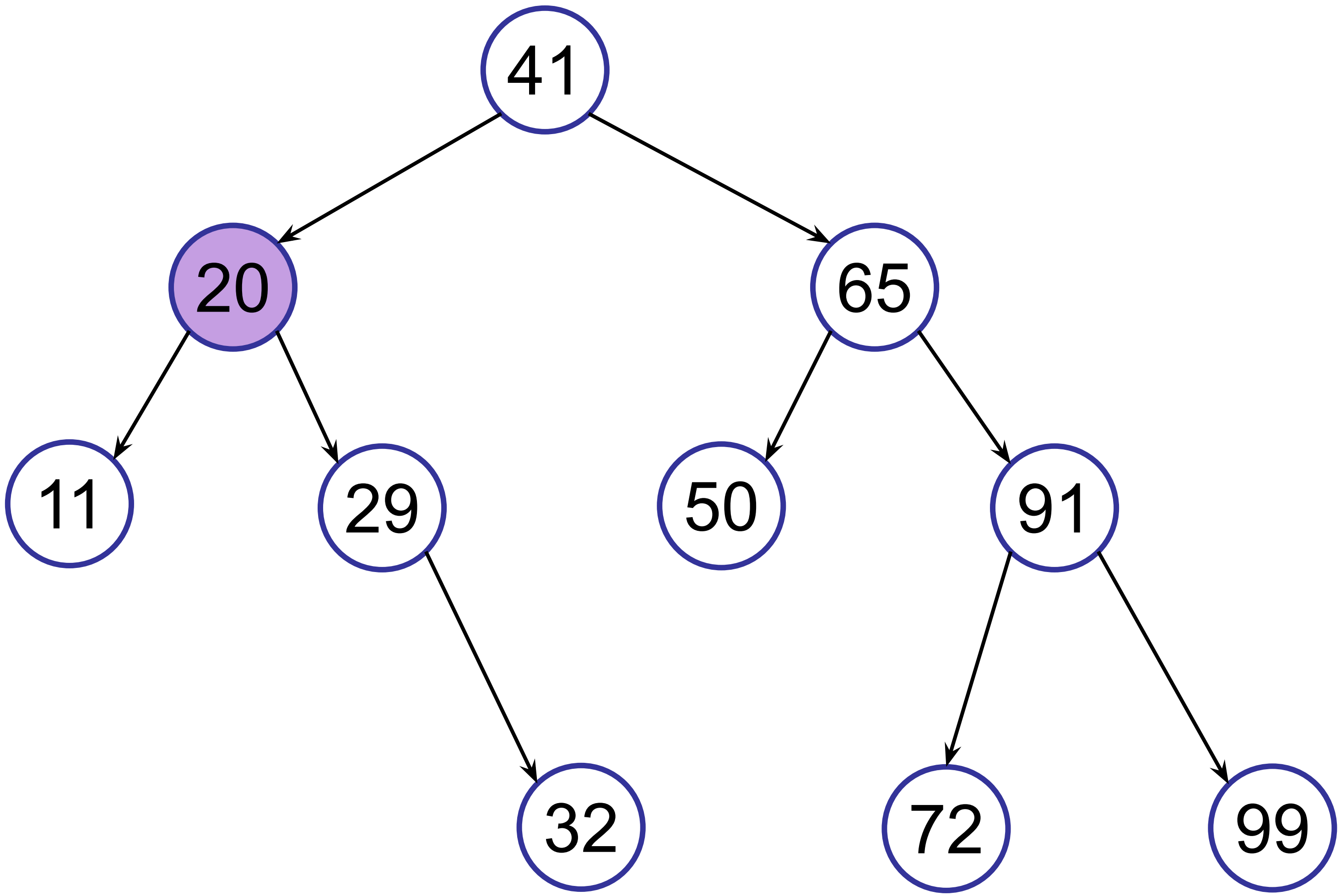
Tree Traversal

in-order-traversal



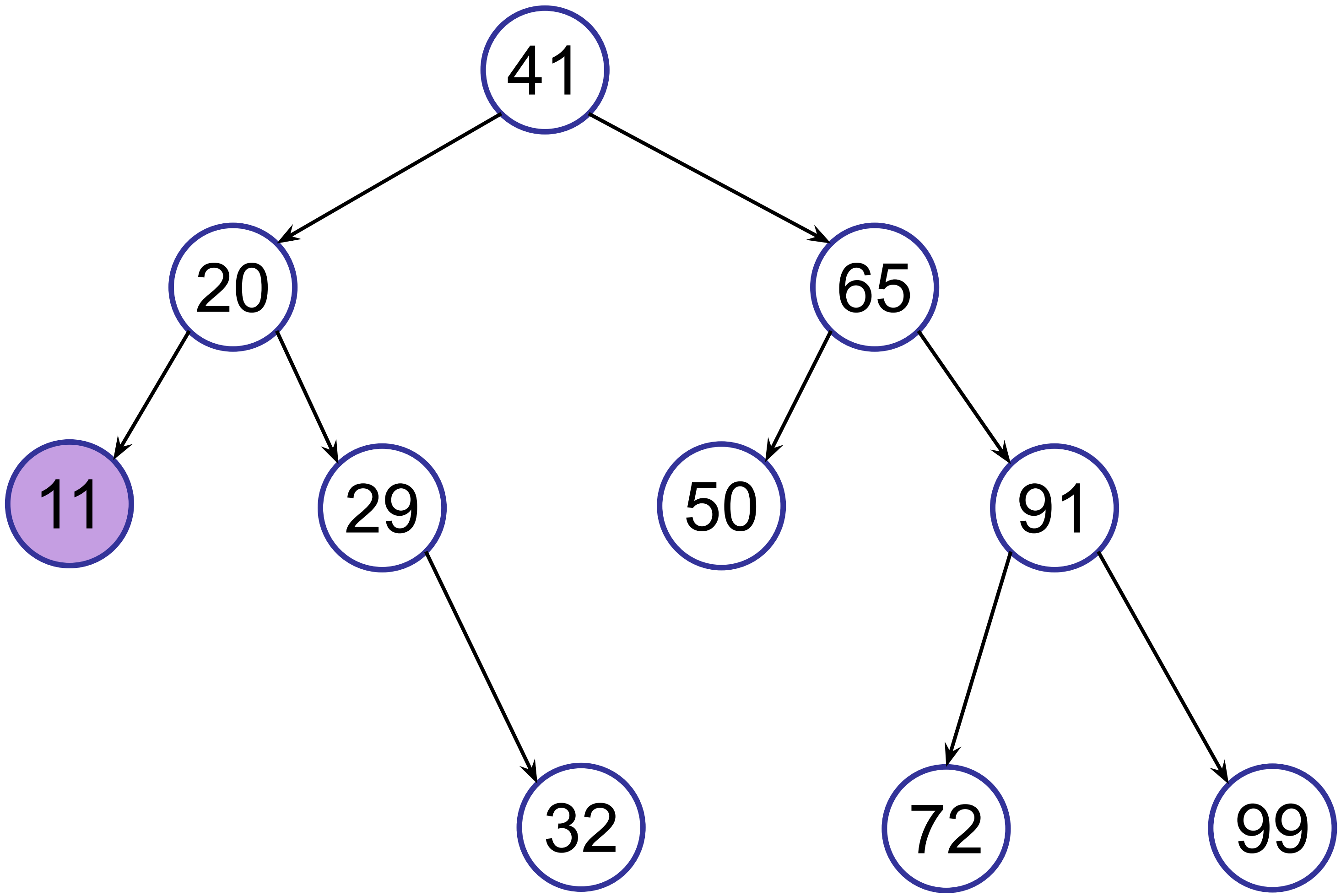
Tree Traversal

in-order-traversal



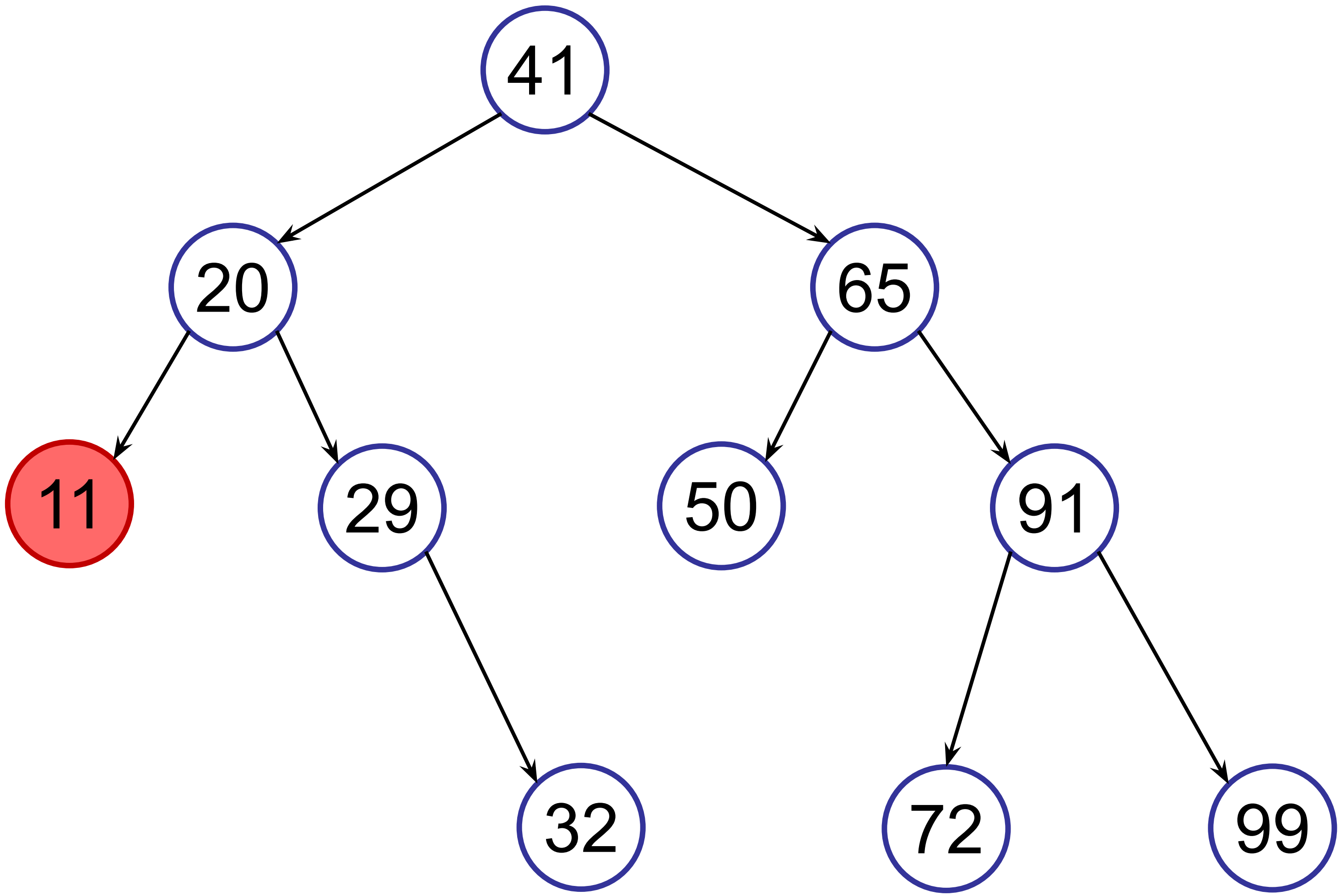
Tree Traversal

in-order-traversal



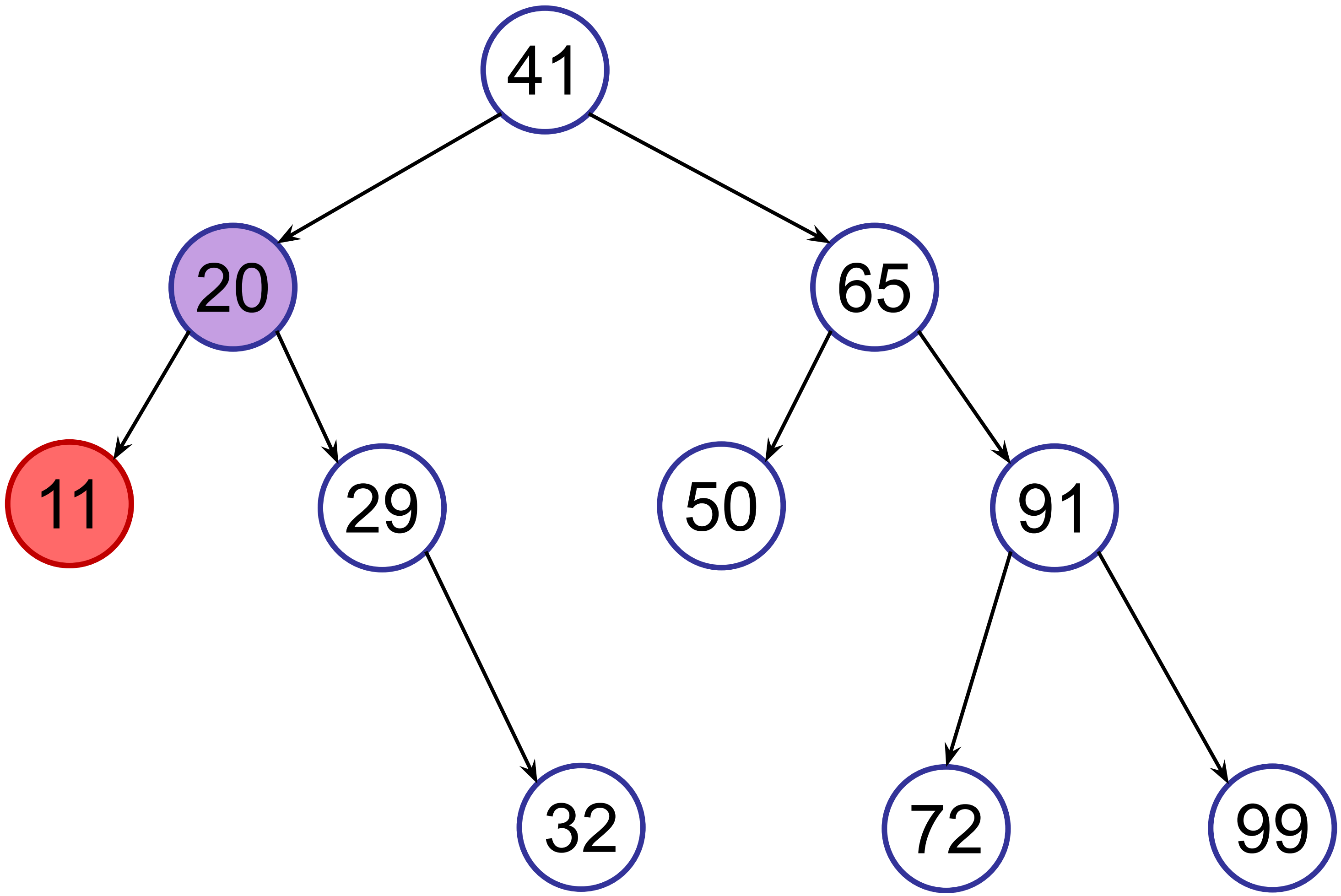
Tree Traversal

in-order-traversal



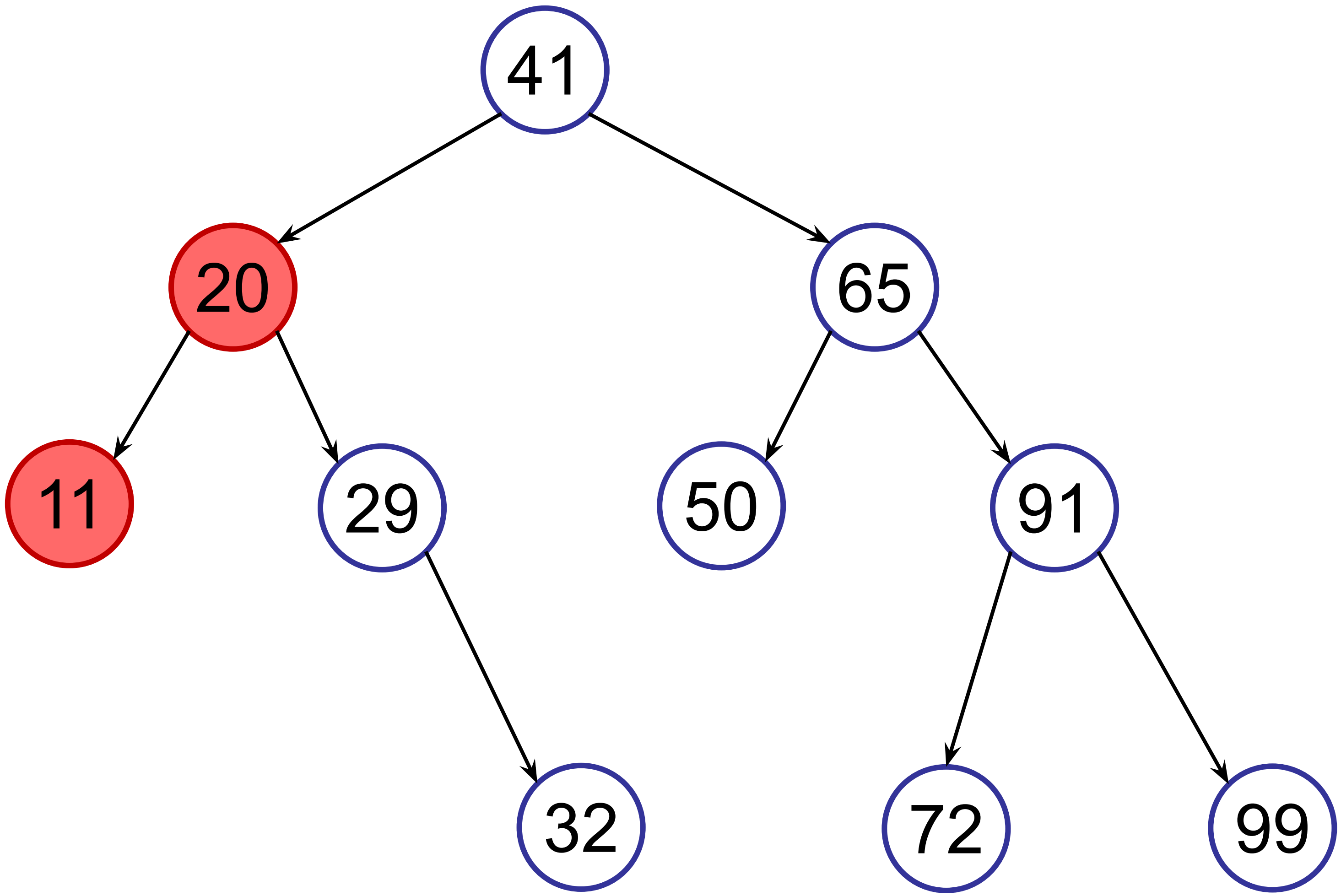
Tree Traversal

in-order-traversal



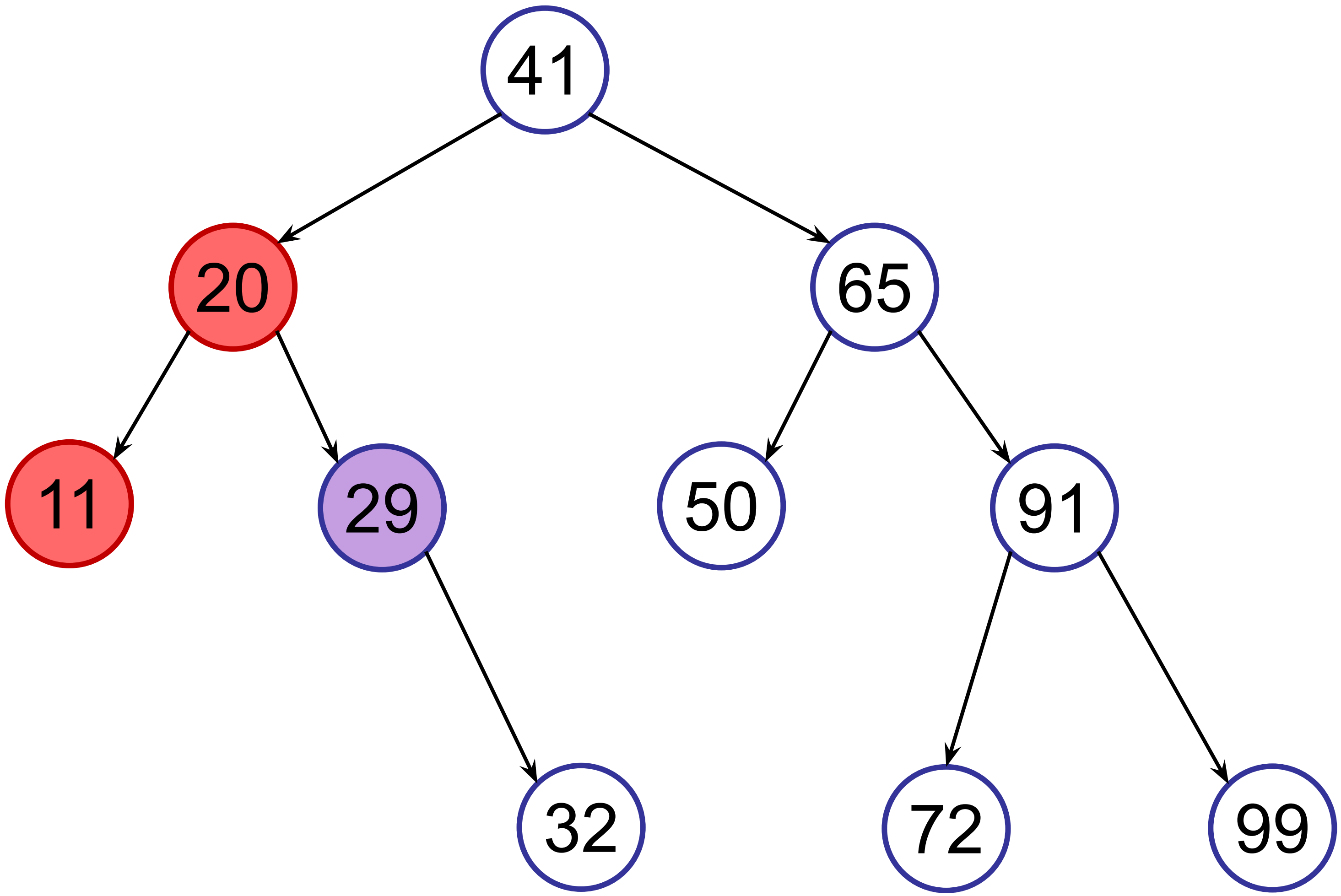
Tree Traversal

in-order-traversal



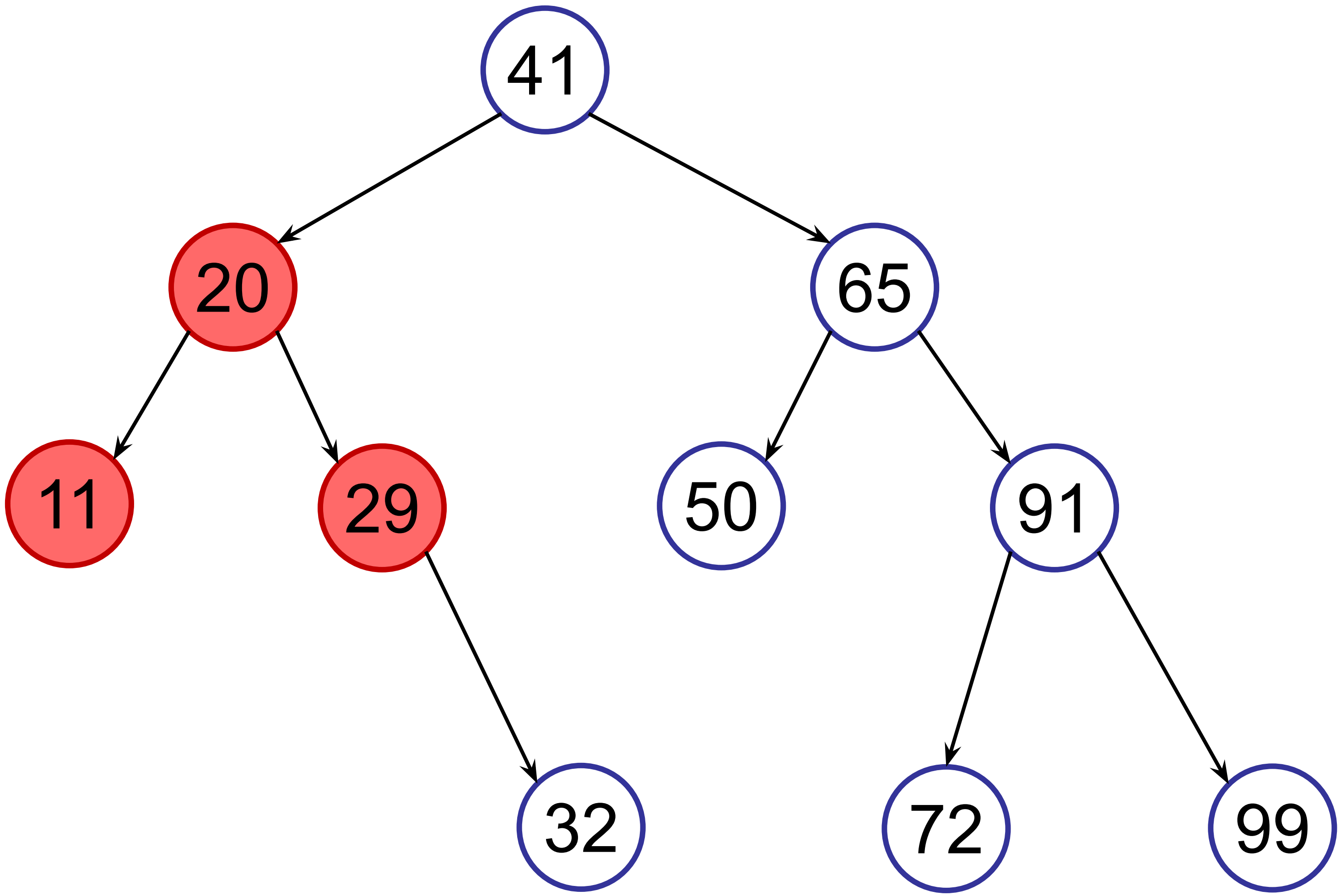
Tree Traversal

in-order-traversal



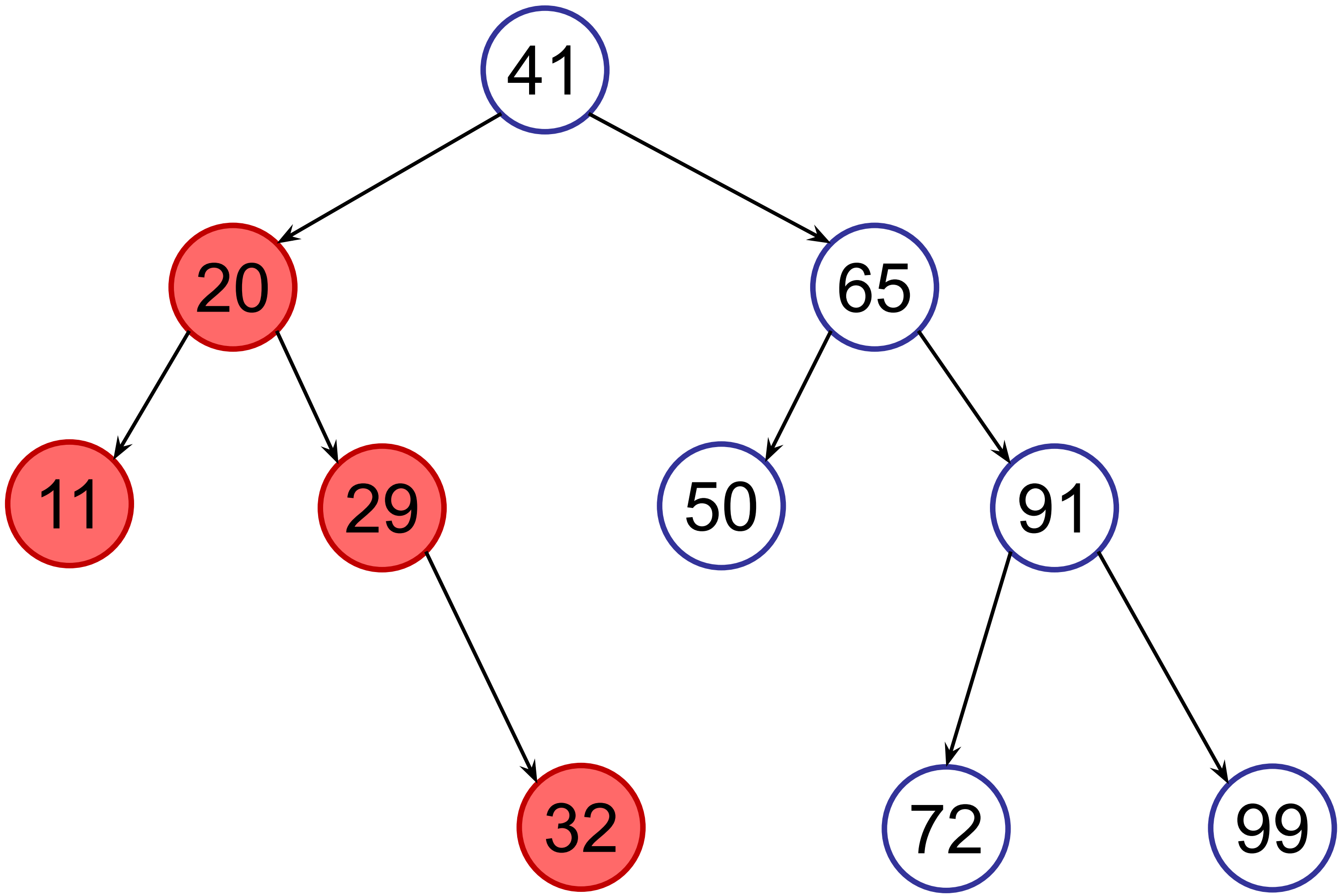
Tree Traversal

in-order-traversal



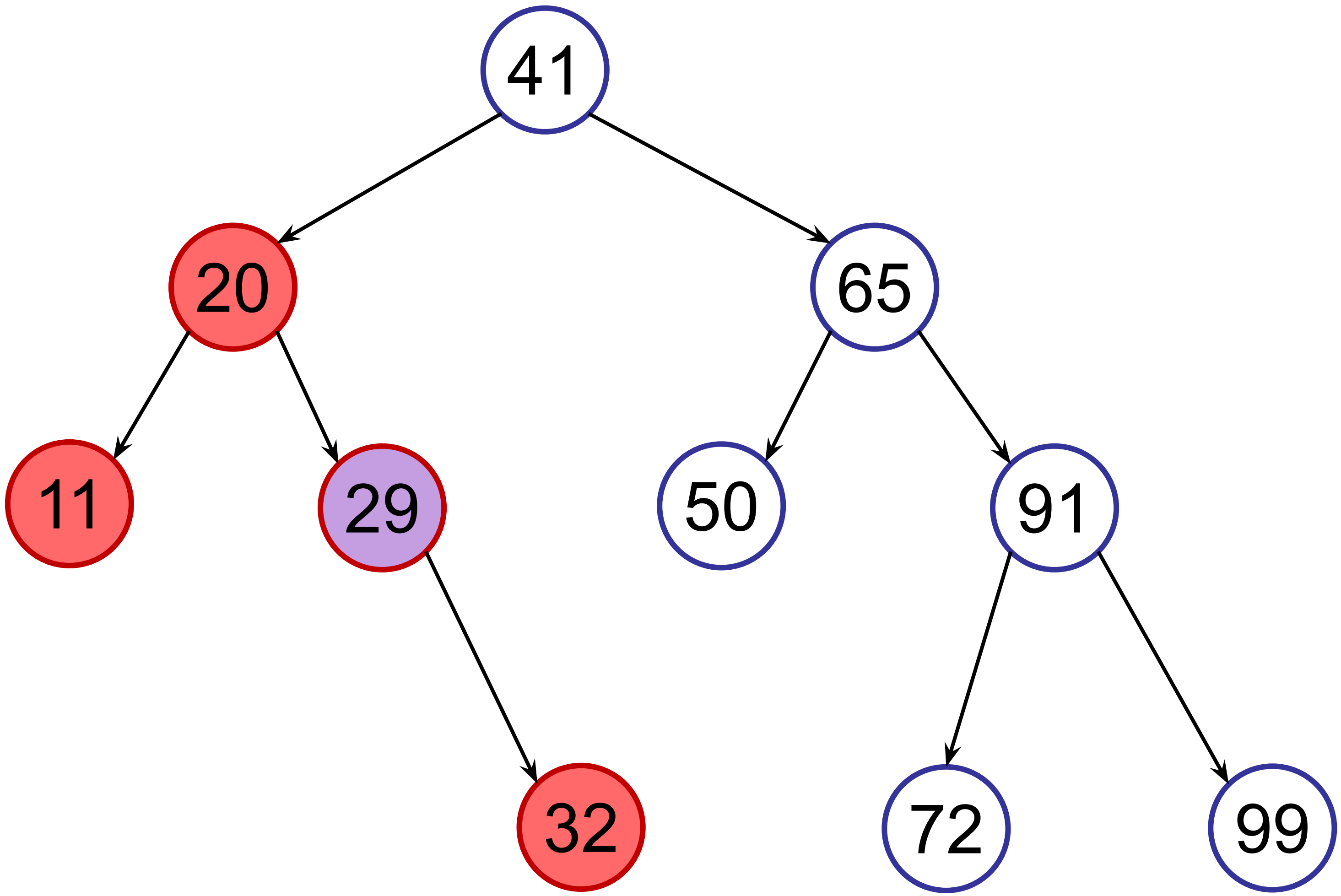
Tree Traversal

in-order-traversal



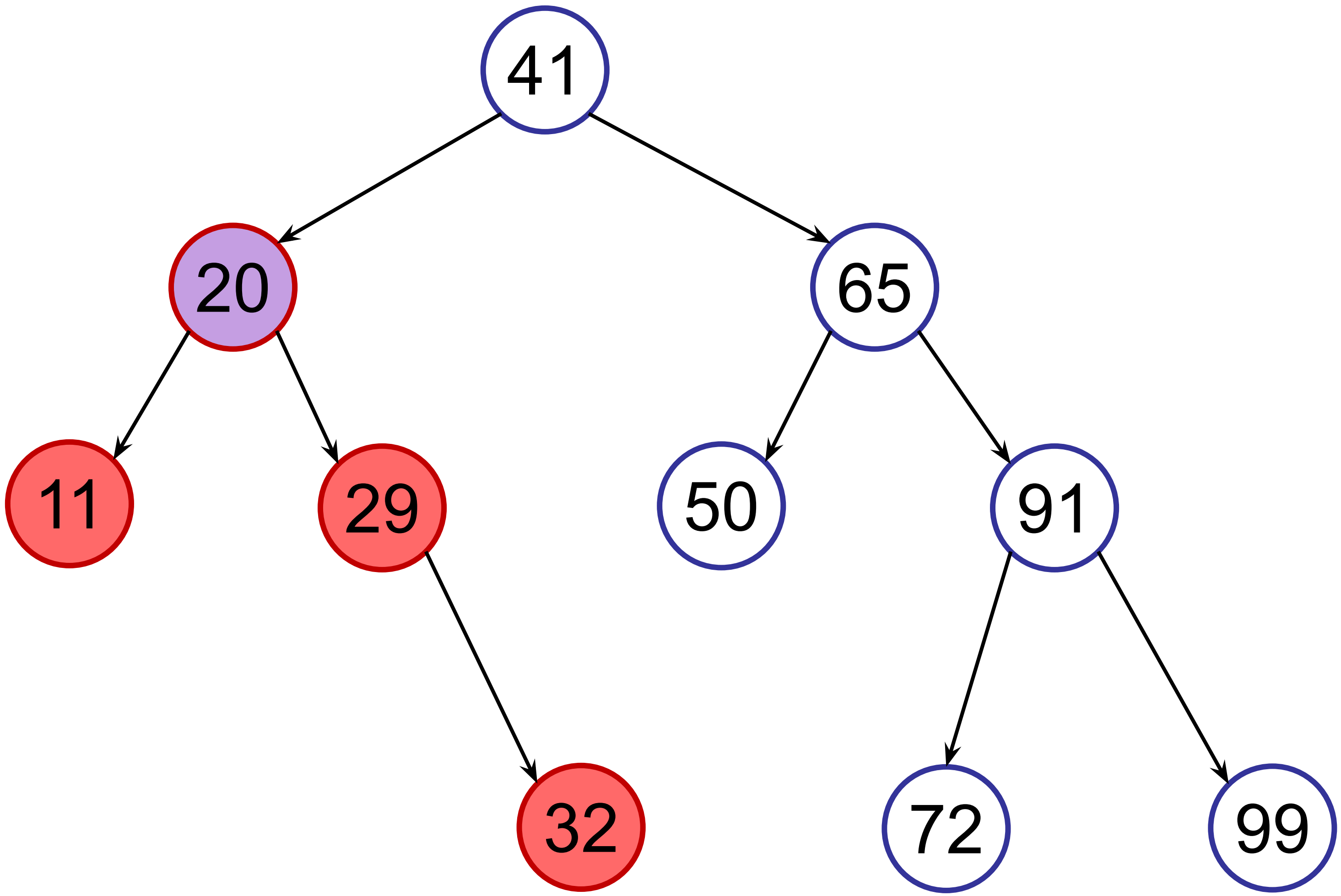
Tree Traversal

in-order-traversal



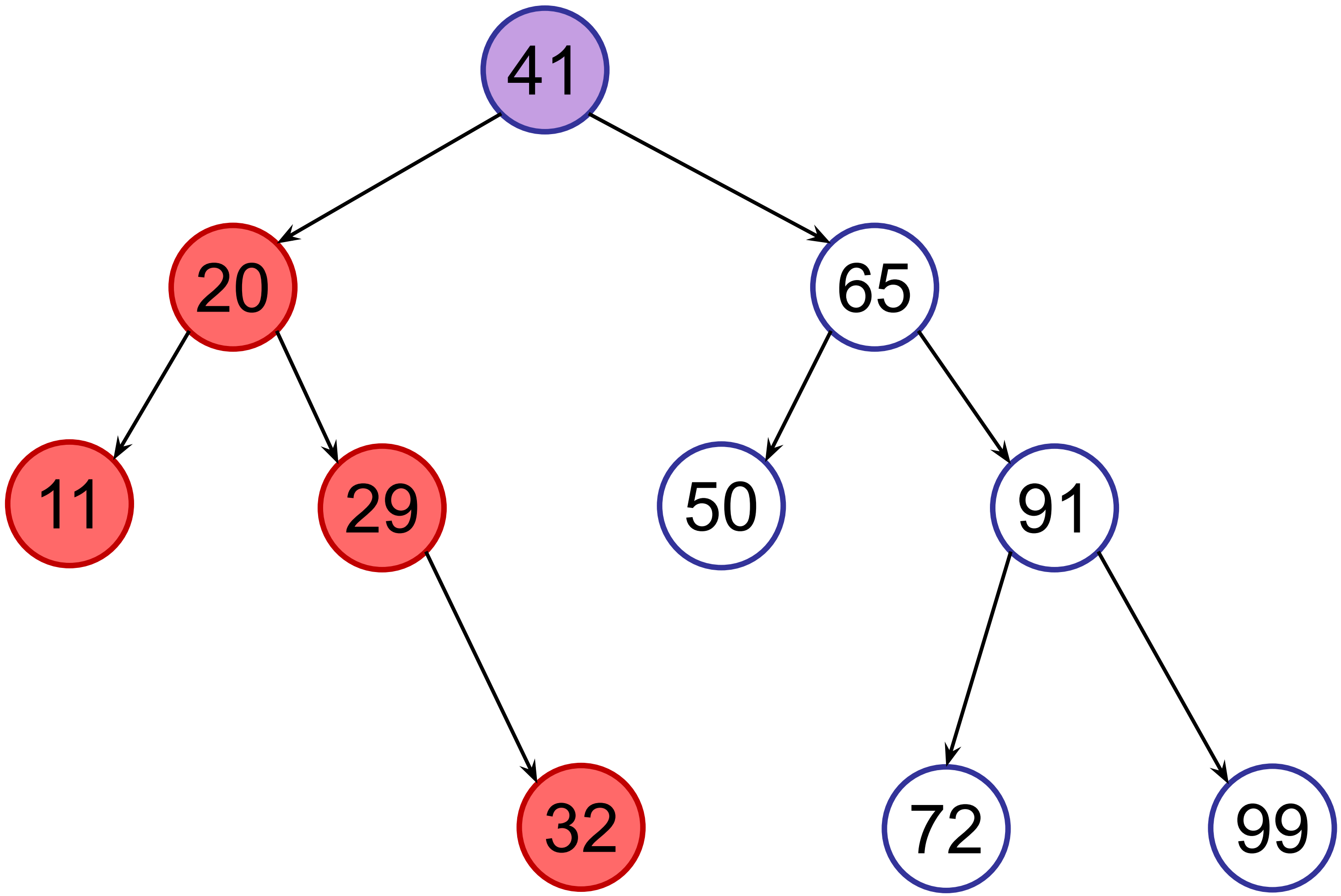
Tree Traversal

in-order-traversal



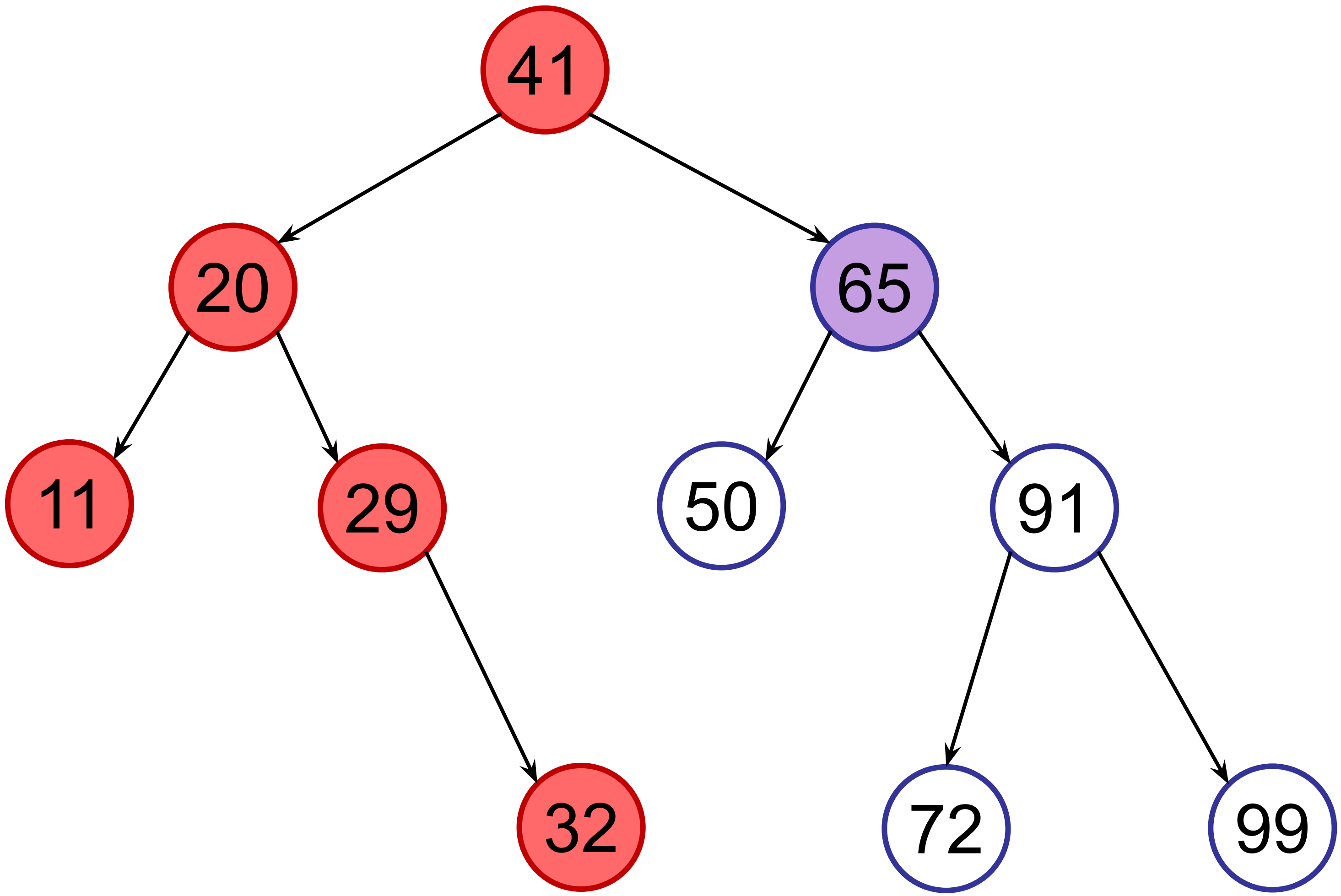
Tree Traversal

in-order-traversal



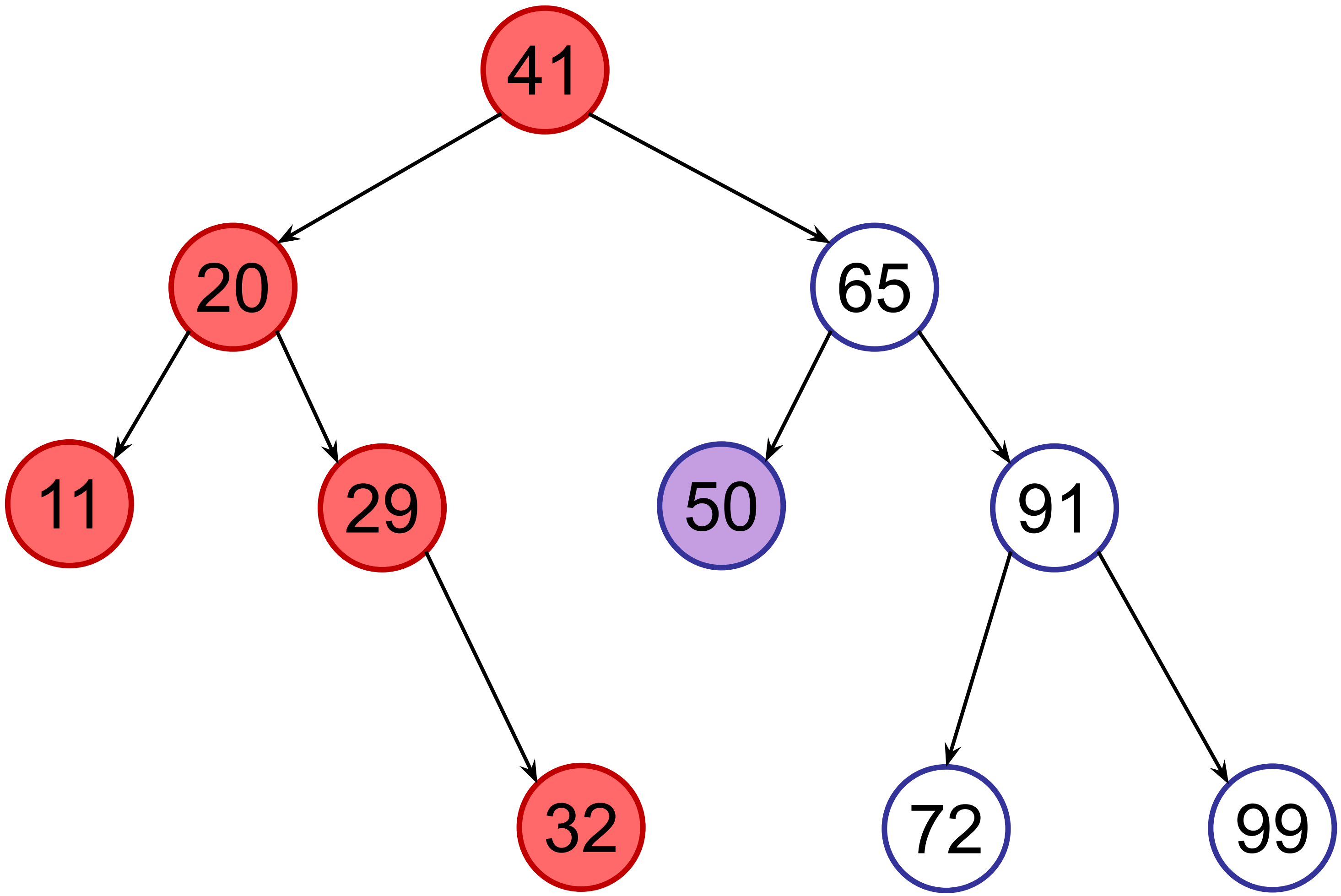
Tree Traversal

in-order-traversal



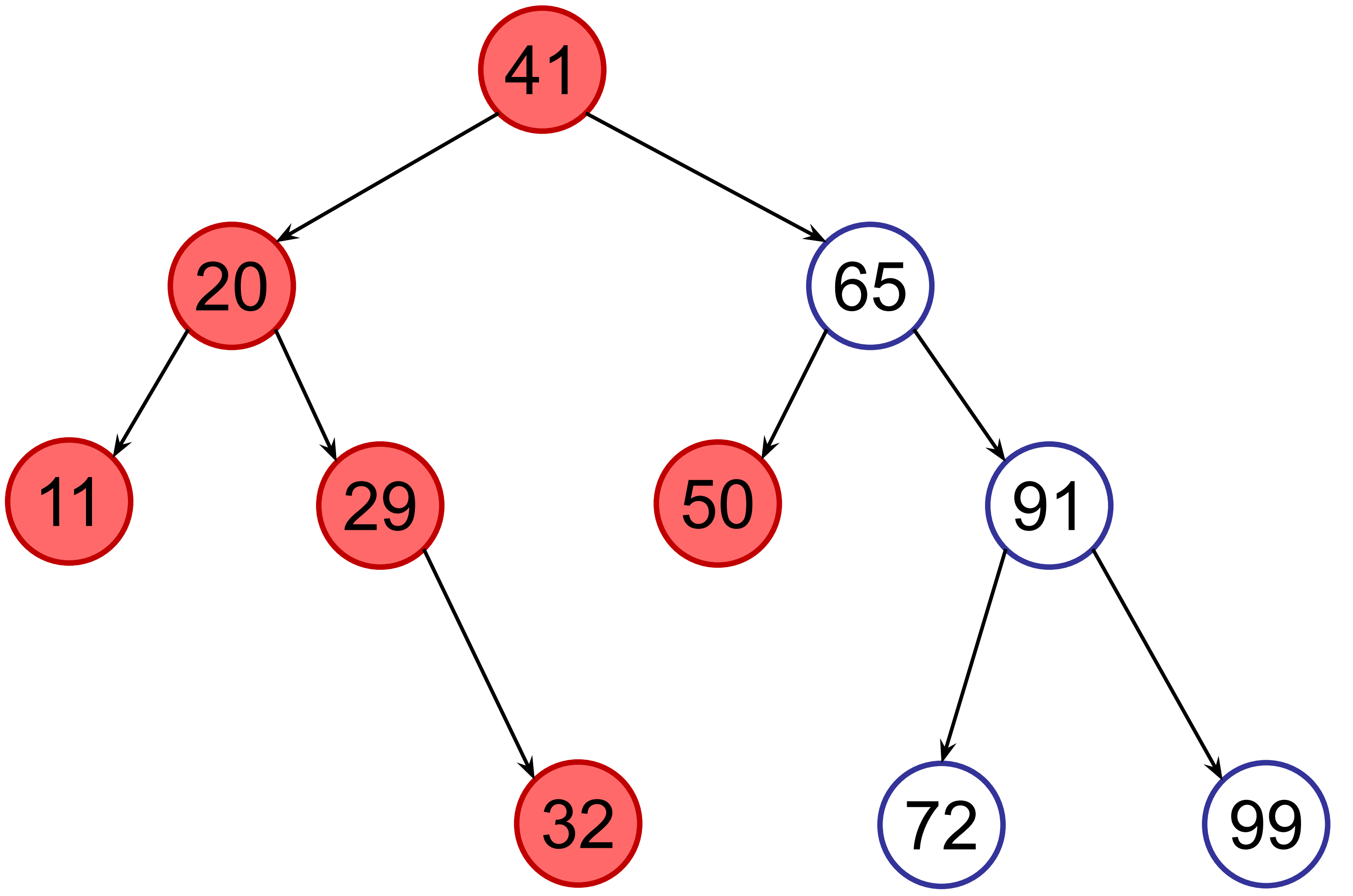
Tree Traversal

in-order-traversal



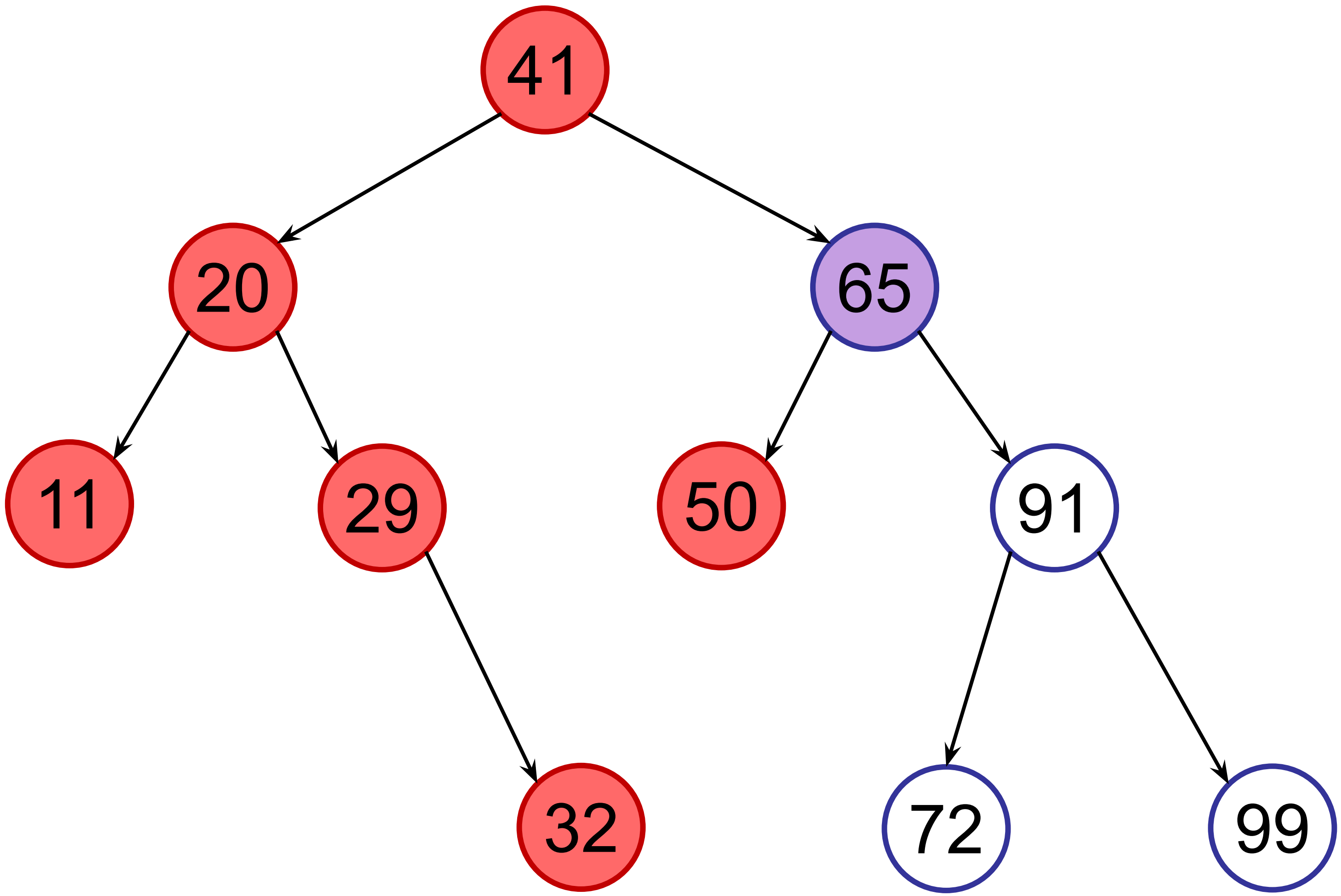
Tree Traversal

in-order-traversal



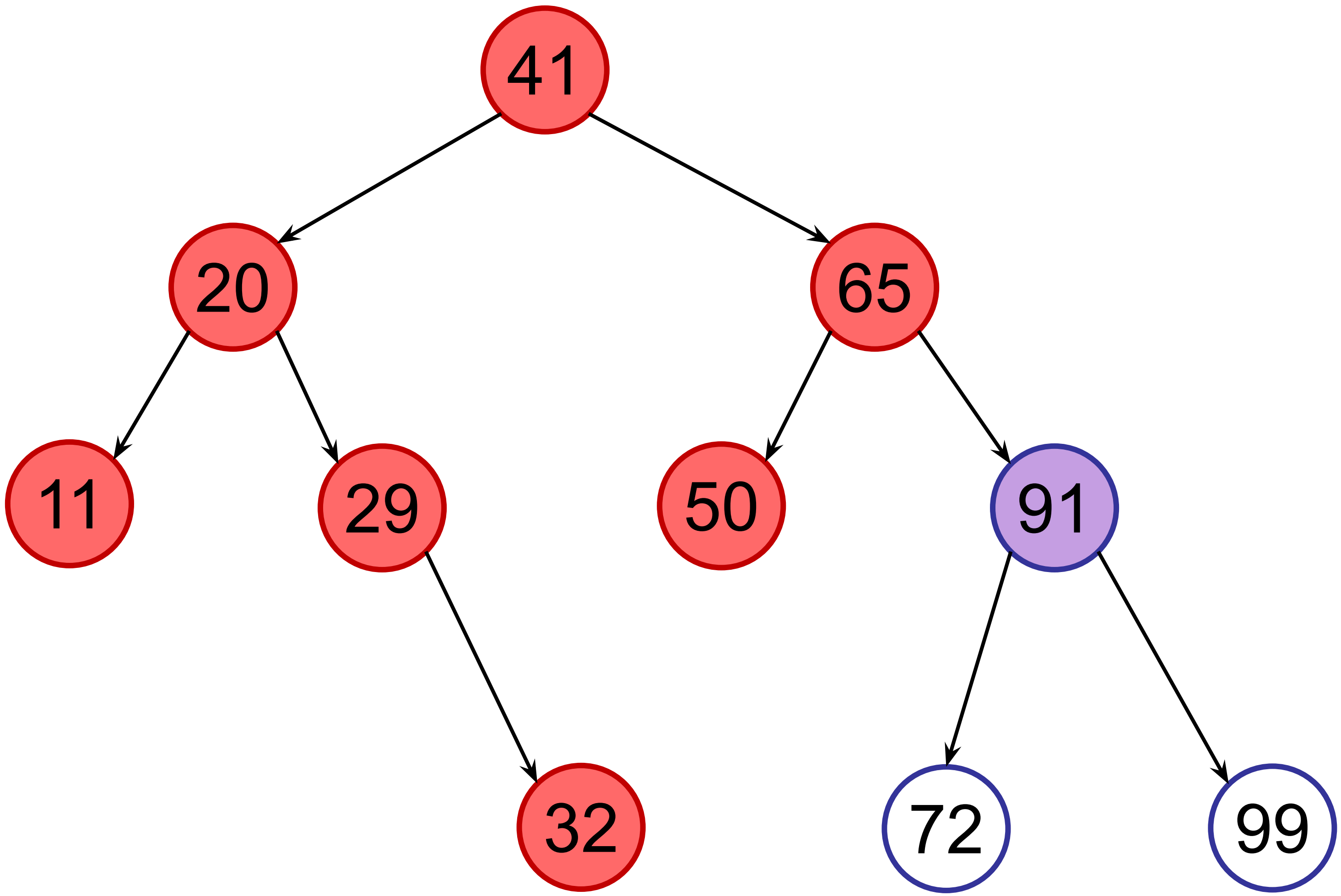
Tree Traversal

in-order-traversal



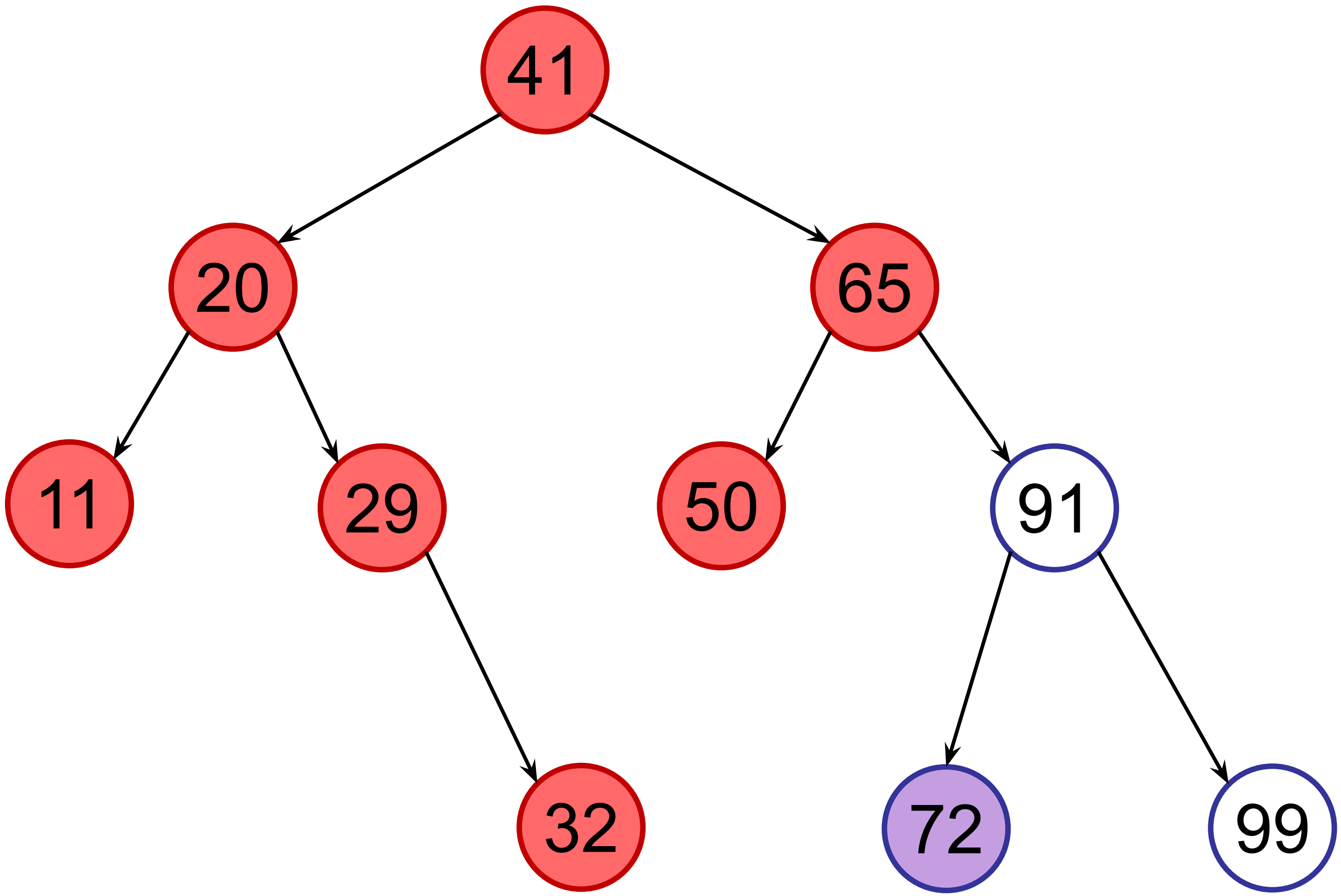
Tree Traversal

in-order-traversal



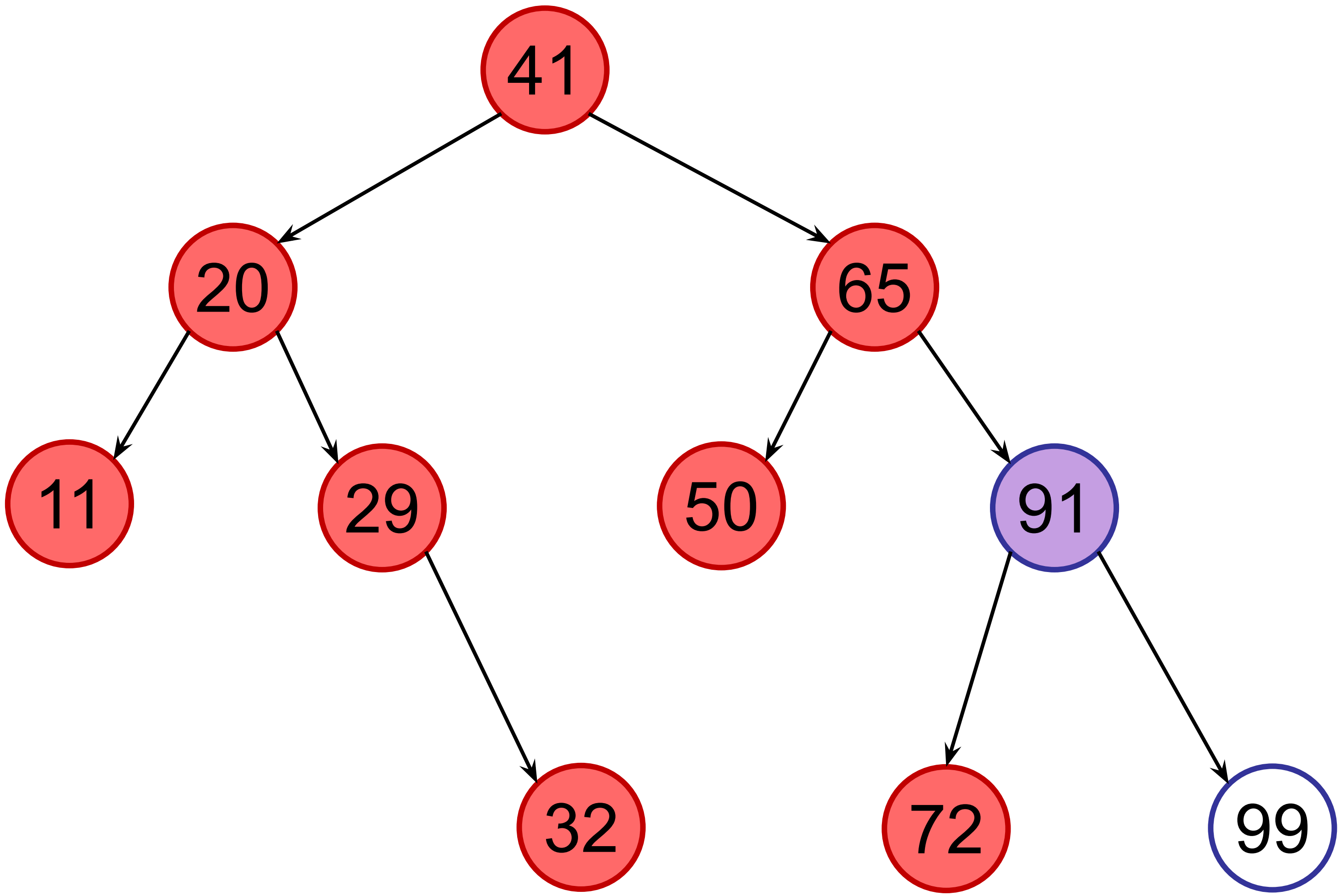
Tree Traversal

in-order-traversal



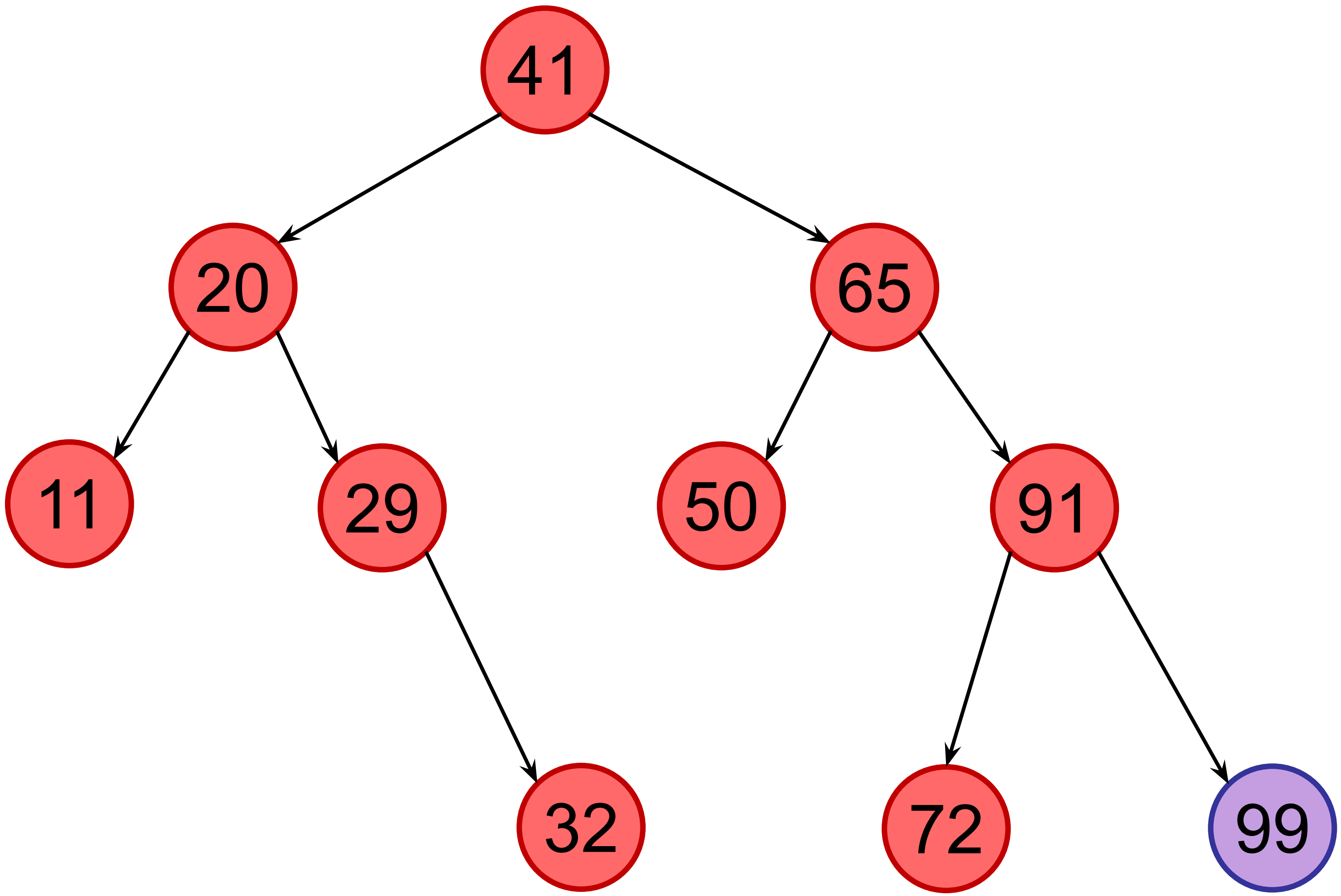
Tree Traversal

in-order-traversal



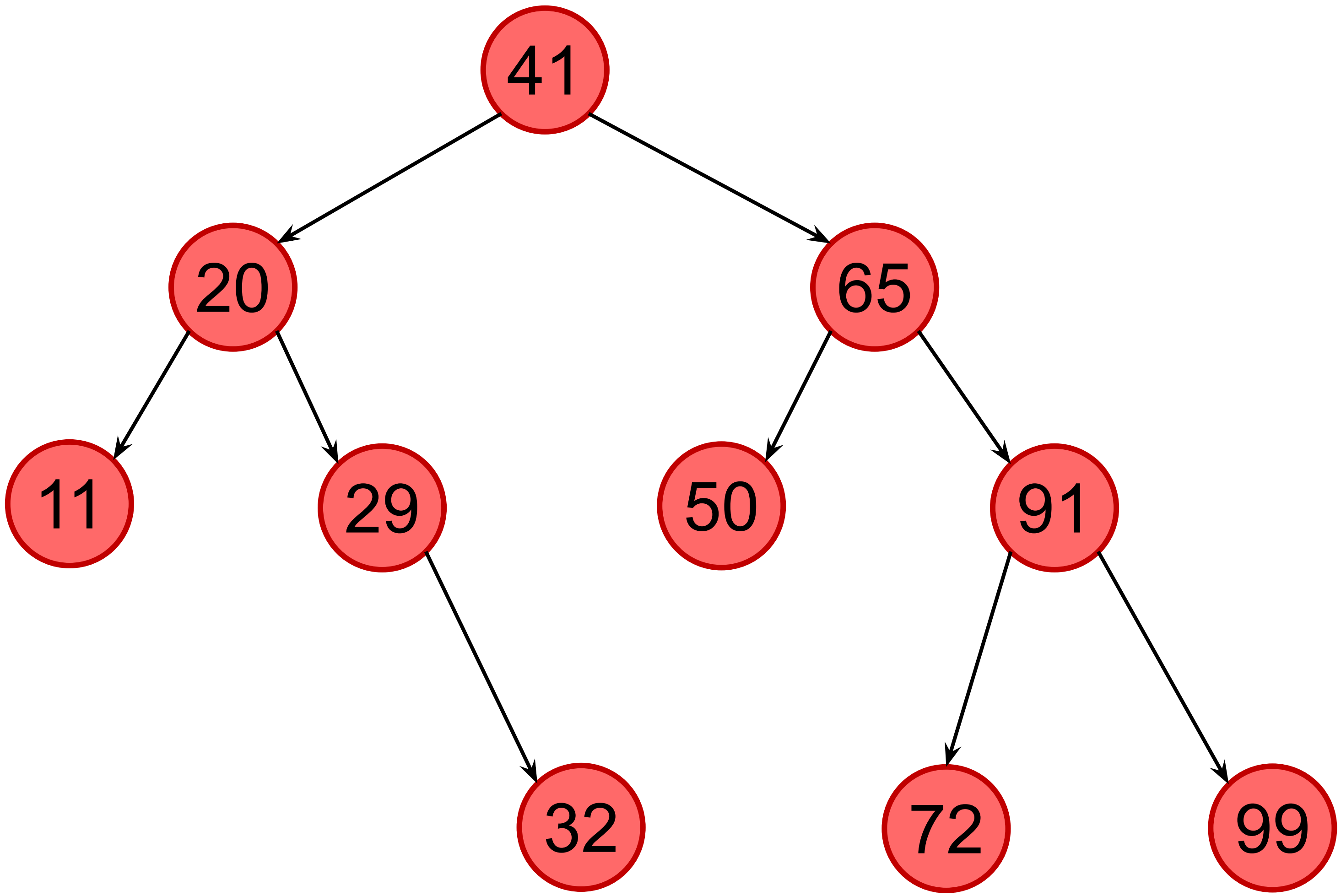
Tree Traversal

in-order-traversal



Tree Traversal

in-order-traversal



Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();

    visit(this);


    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
         leftTree.in-order-traversal();

    visit(this);

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();

    visit(this);

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

How long does an in-order-traversal take?

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. $O(2^n)$

Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();

    visit(this);

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Running time: O(n)

- visits each node at most once

Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
         leftTree.in-order-traversal();


    visit(this);


    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Running time: O(n)

- visits each node at most once, each visit costs O(1)

Tree Traversal

in-order-traversal(v)

```
public void in-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
         leftTree.in-order-traversal();

    visit(this);

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Running time: O(n)

- n nodes x O(1) work per node = O(n)

How long does an in-order-traversal take?

1. $O(1)$
2. $O(\log n)$
3. $\boxed{O(n)}$
4. $O(n \log n)$
5. $O(n^2)$
6. $O(2^n)$

Tree Traversal

in-order-traversal(v)

- left-subtree
- SELF
- right-subtree

pre-order-traversal(v)

- SELF
- left-subtree
- right-subtree

post-order-traversal(v)

- left-subtree
- right-subtree
- SELF

Tree Traversals

pre-order-traversal(v)

```
public void pre-order-traversal(){
    visit(this);

    // Traverse left sub-tree
    if (leftTree != null)
         leftTree.in-order-traversal();


    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

Tree Traversals

pre-order-traversal(v)

```
public void pre-order-traversal(){
    visit(this);

    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();
}
```

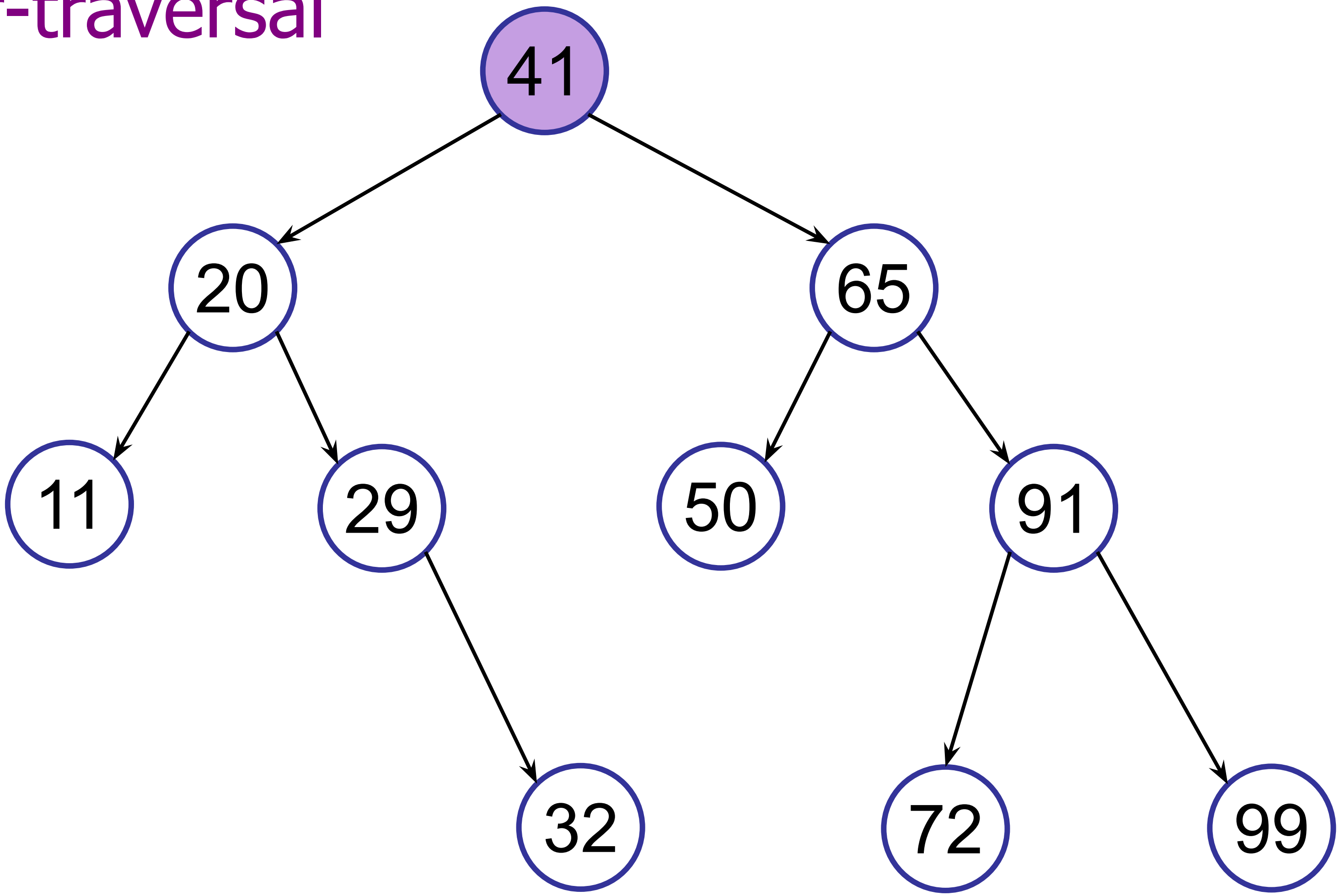
Tree Traversals

pre-order-traversal(v)

```
public void pre-order-traversal(){
    visit(this);

    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();

    // Traverse right sub-tree
    if (rightTree != null)
       rightTree.in-order-traversal();
}
```

Tree Traversals

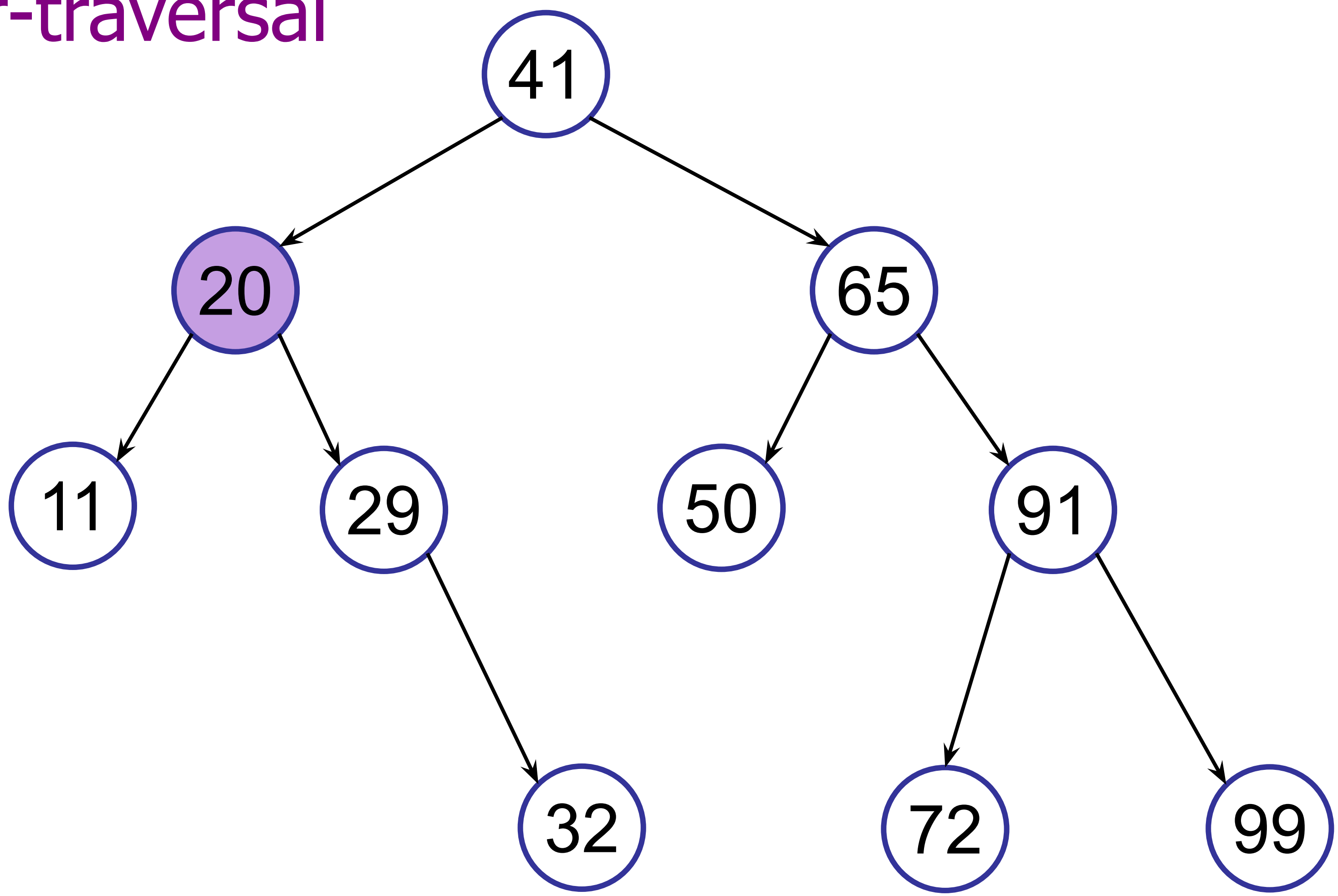
pre-order-traversal



41

Tree Traversals

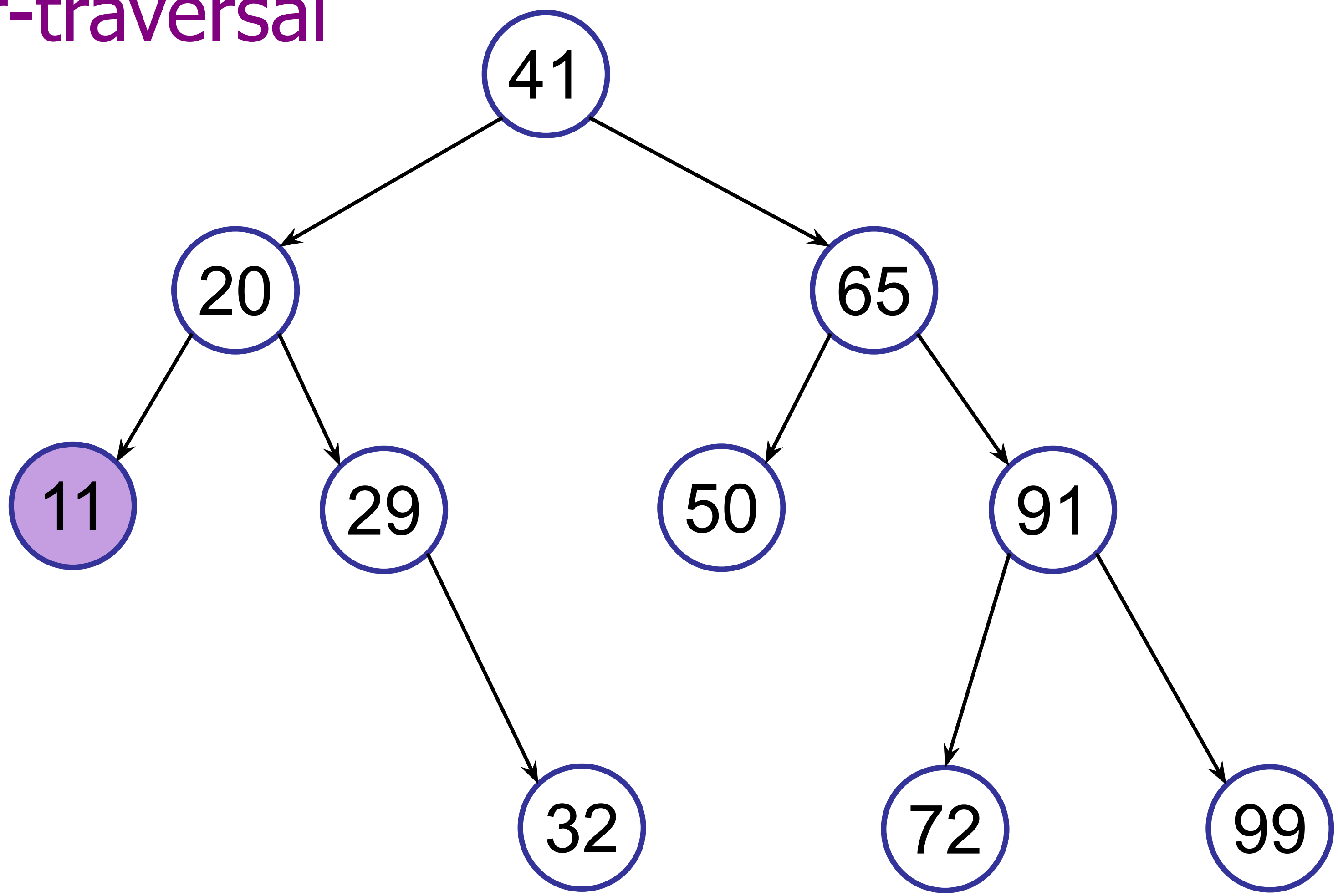
pre-order-traversal



41 20

Tree Traversals

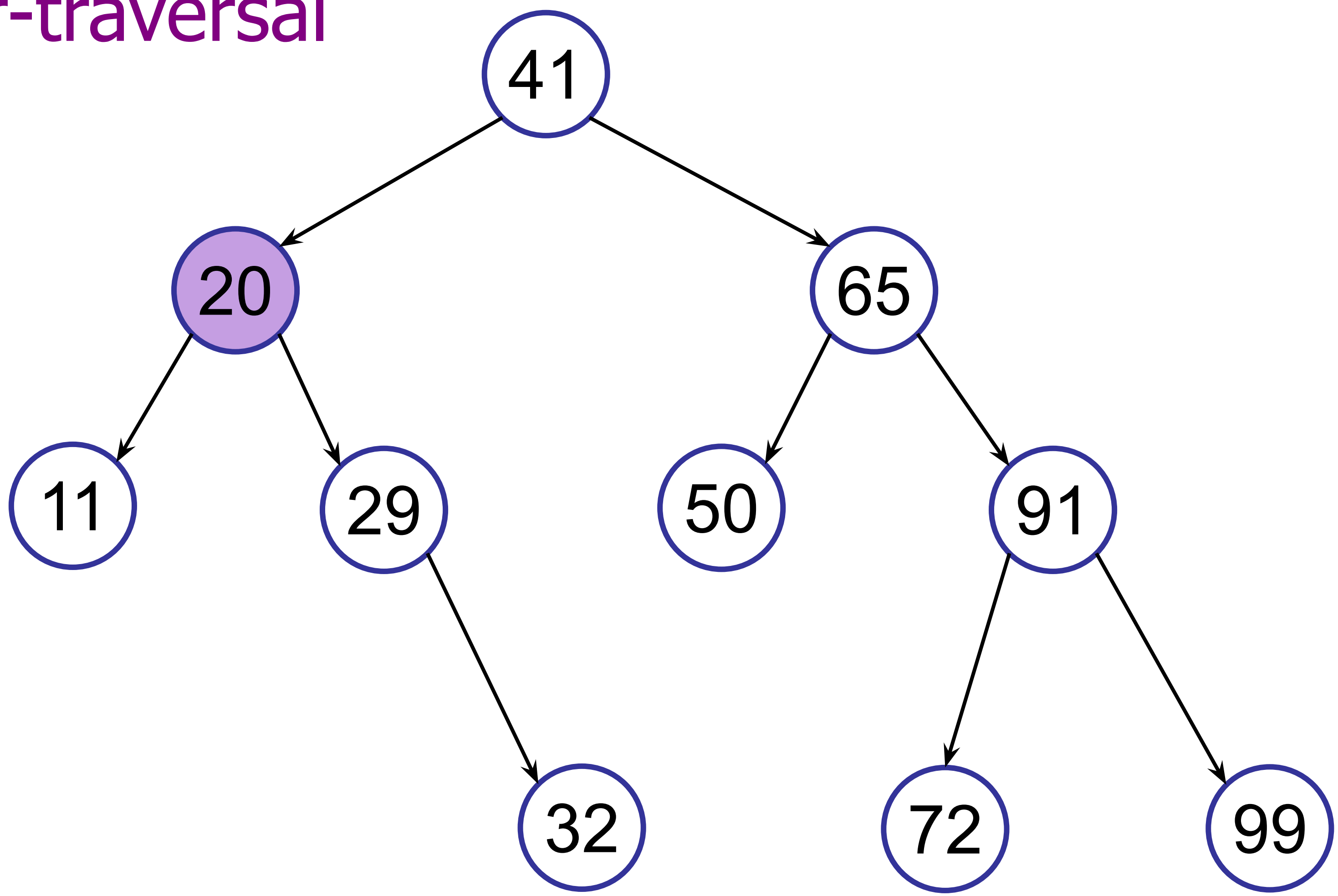
pre-order-traversal



41 20
11

Tree Traversals

pre-order-traversal

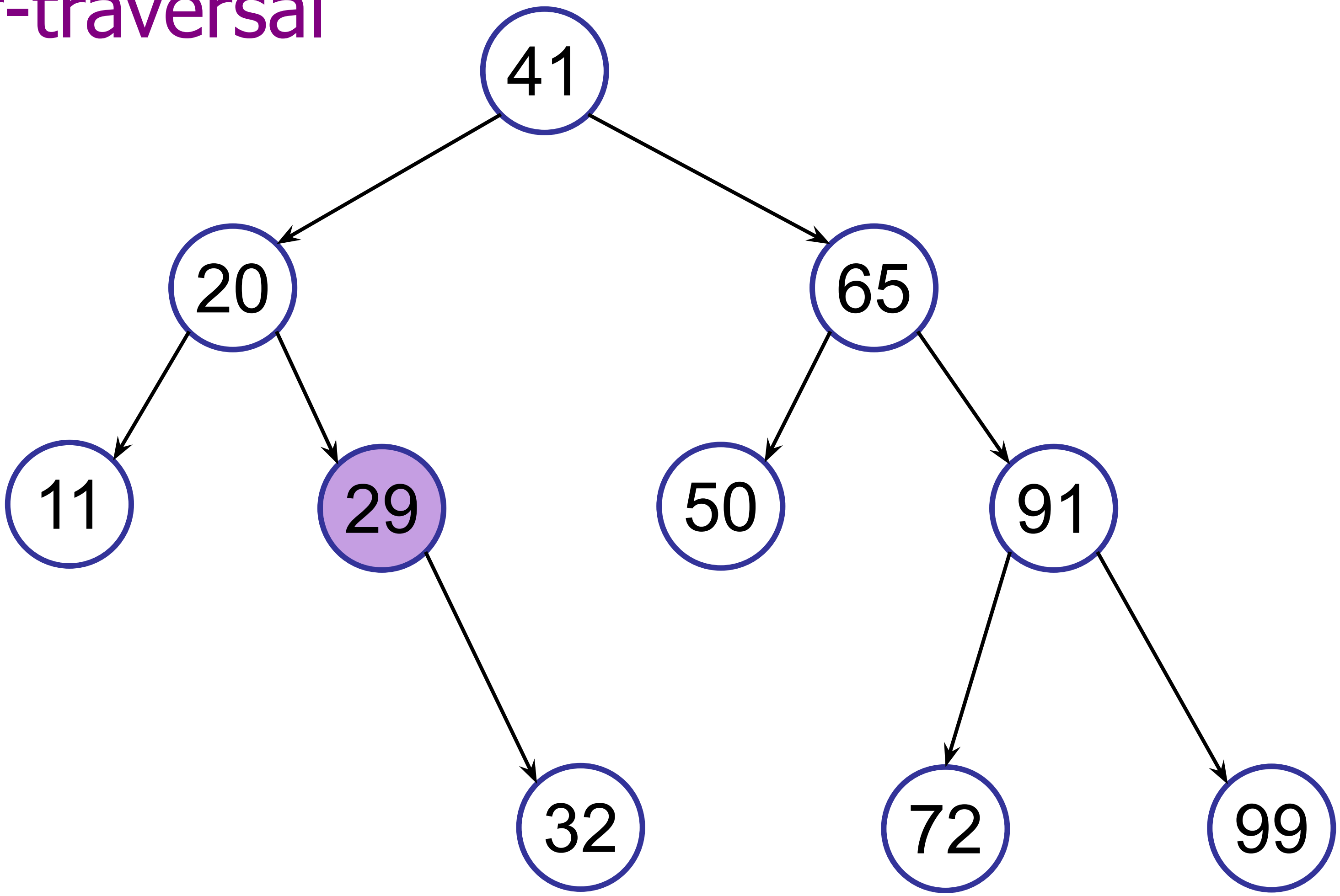


41 20

11

Tree Traversals

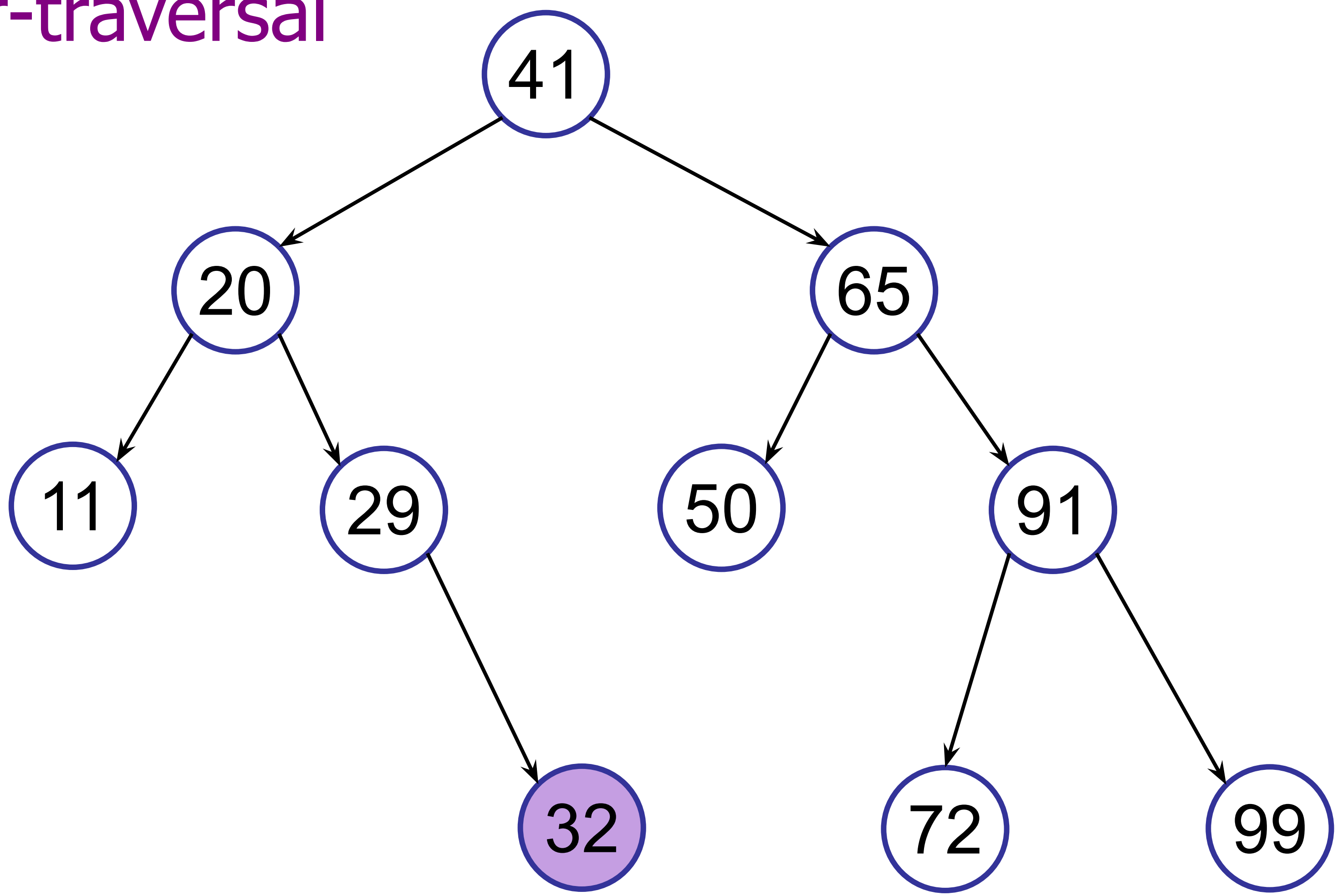
pre-order-traversal



41 20 11 29

Tree Traversals

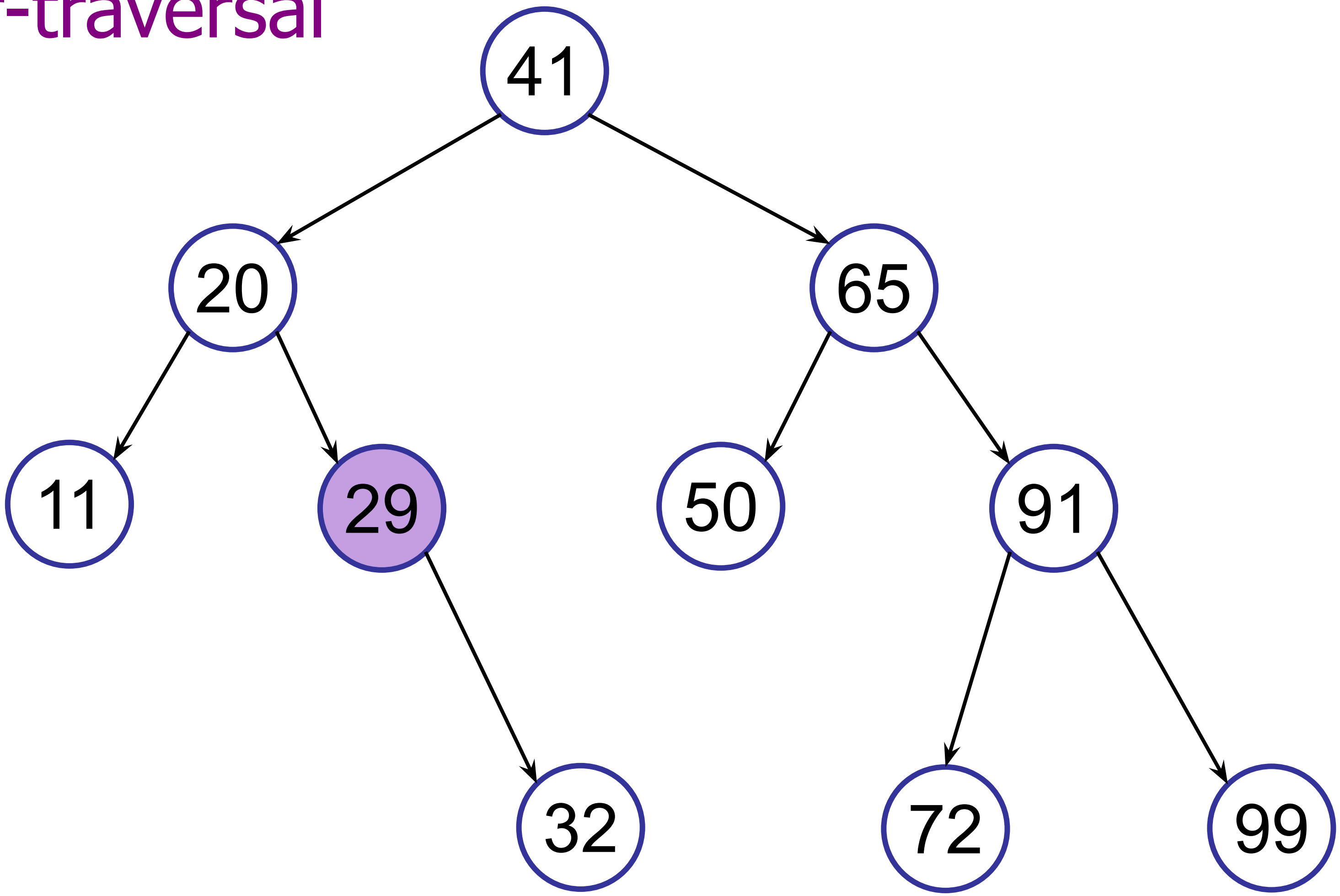
pre-order-traversal



41 20 11 29 32

Tree Traversals

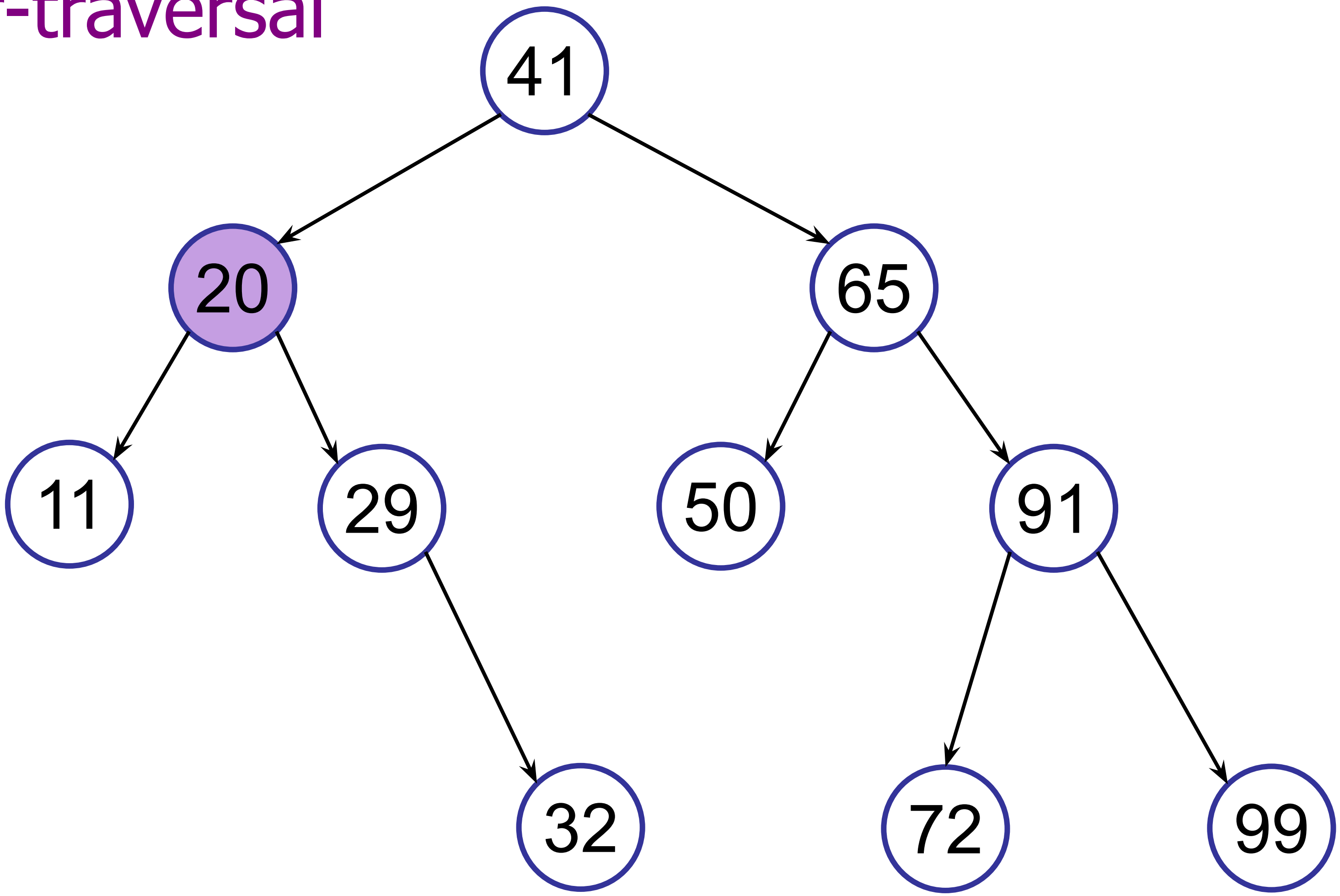
pre-order-traversal



41 20 11 29 32

Tree Traversals

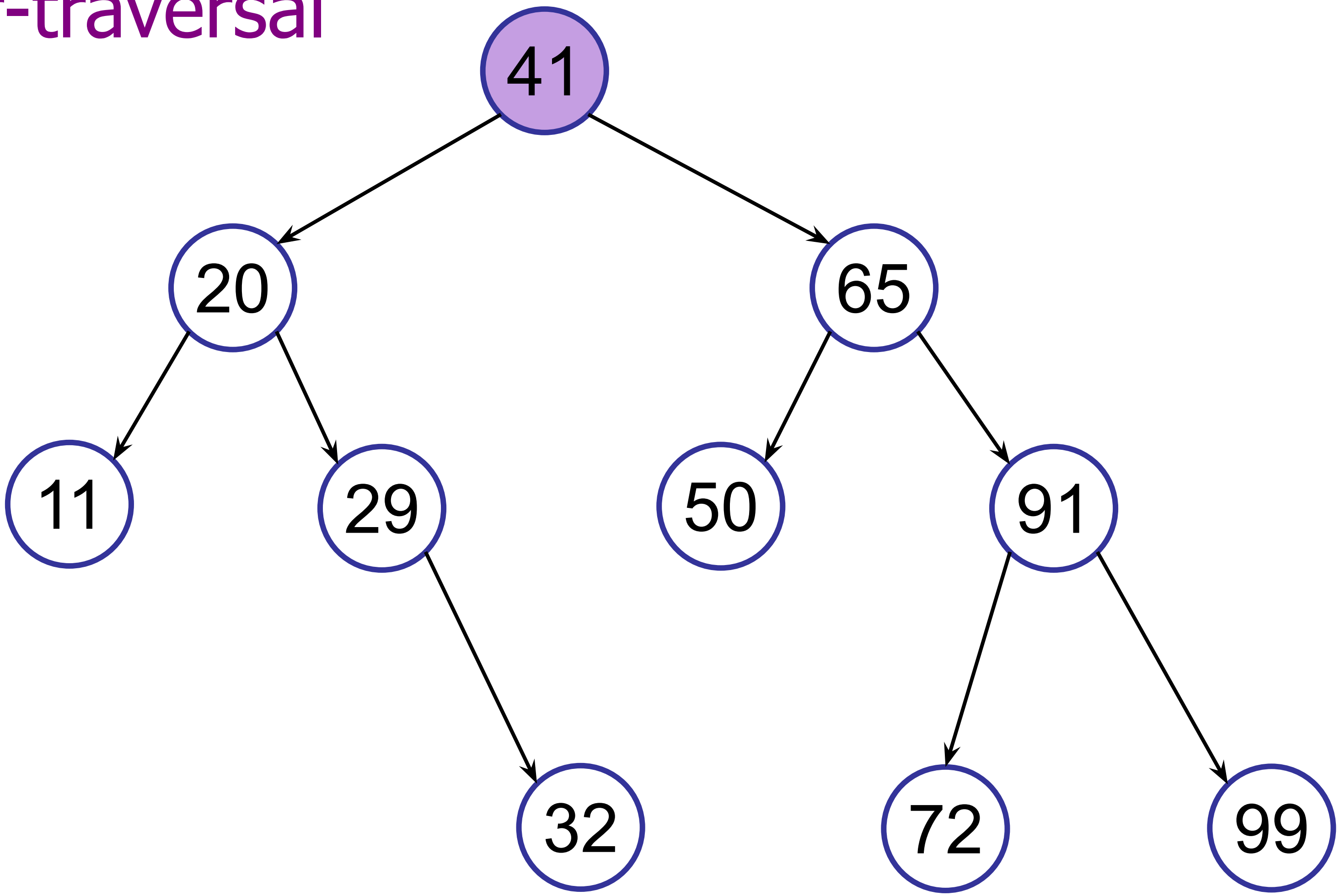
pre-order-traversal



41 20 11 29 32

Tree Traversals

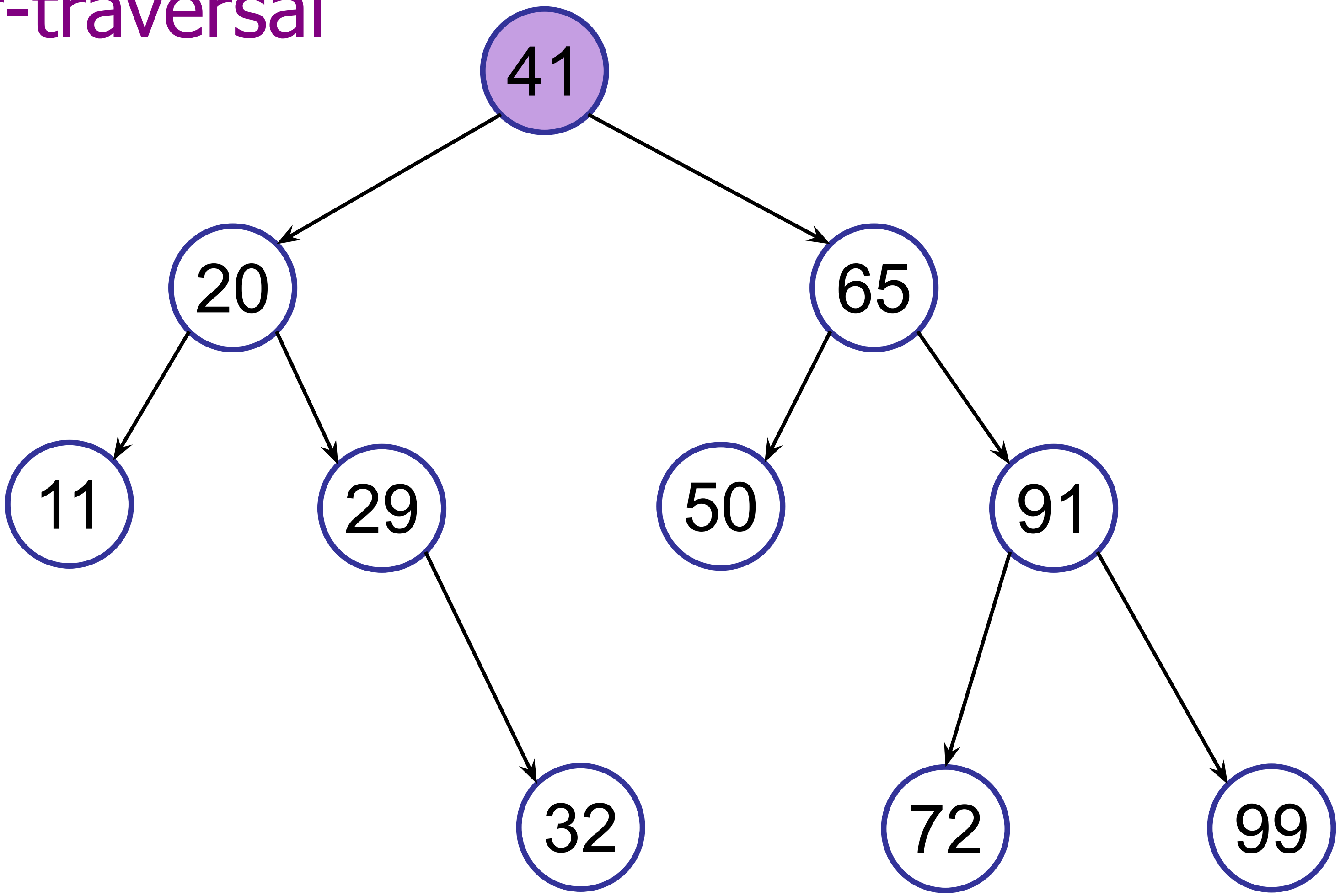
pre-order-traversal



41 20 11 29 32

Tree Traversals

pre-order-traversal



41 20 11 29 32 65 50 91 72 99

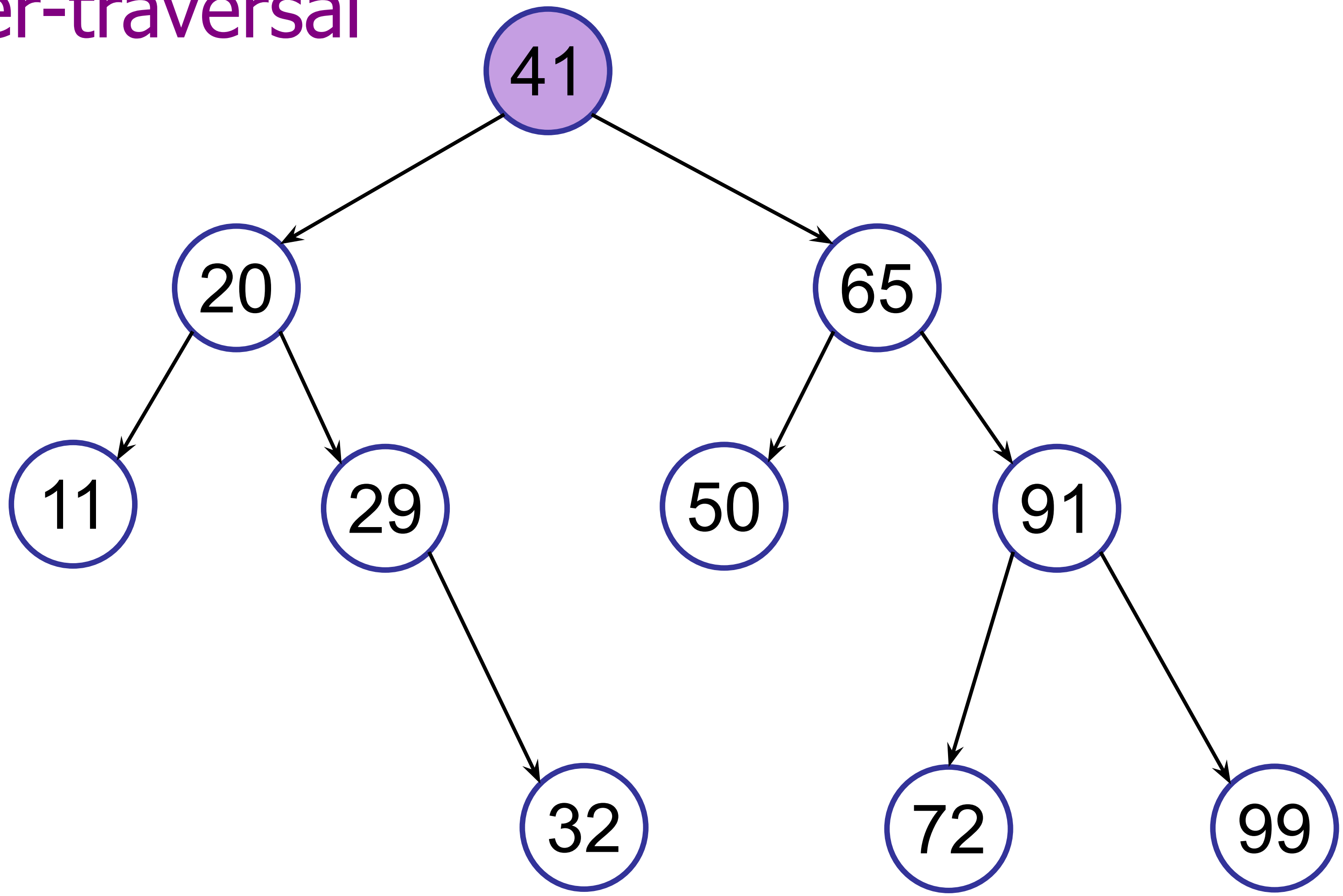
Tree Traversals

post-order-traversal(v)

```
public void post-order-traversal(){
    // Traverse left sub-tree
    if (leftTree != null)
         leftTree.in-order-traversal();

    // Traverse right sub-tree
    if (rightTree != null)
        rightTree.in-order-traversal();

    visit(this);
}
```

Tree Traversals

post-order-traversal



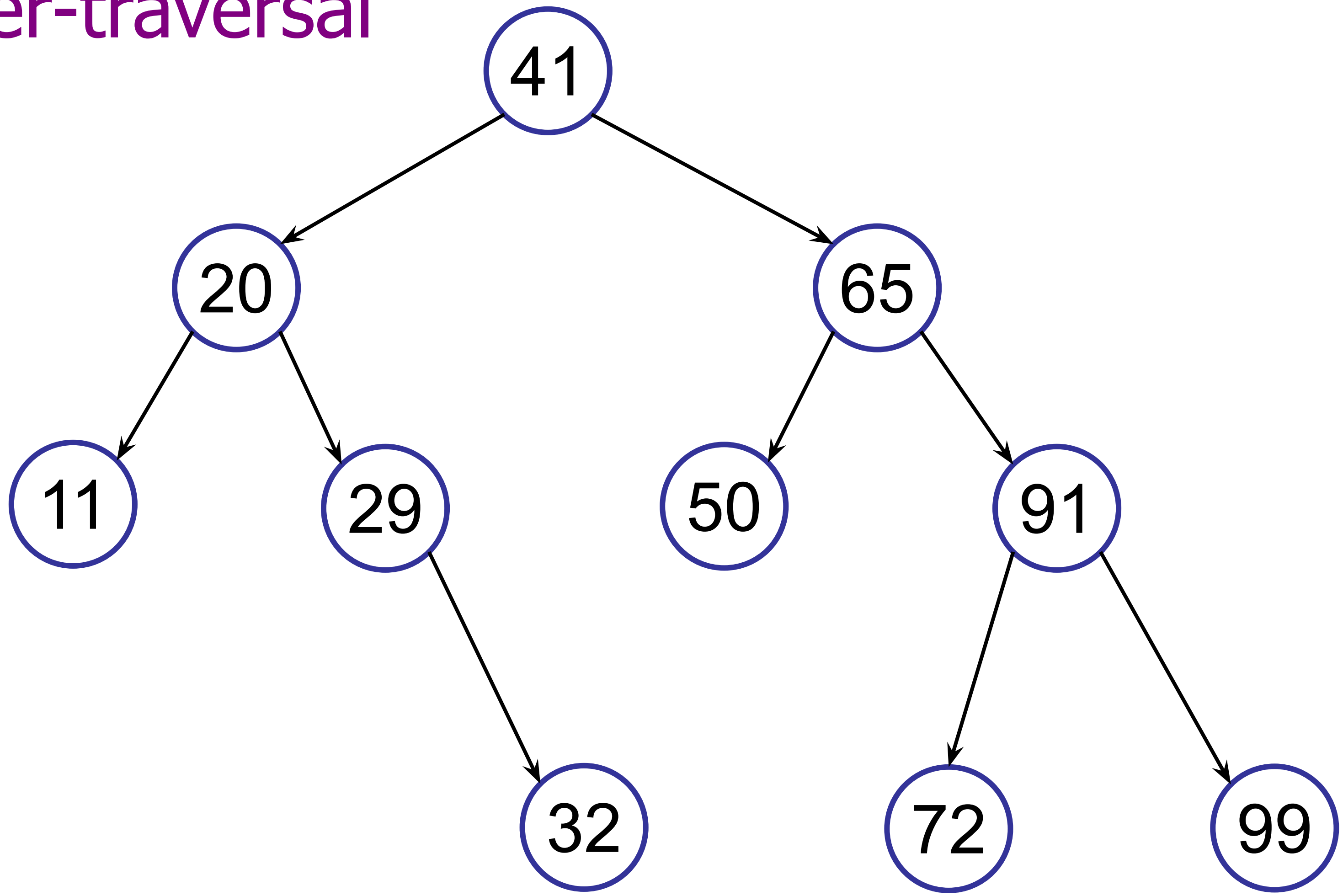
11 32 29 20 50 72 99 91 65 41

Tree Traversals

1. In-order
2. Pre-order
3. Post-order
4. Level-Order traversal ⟵ New!

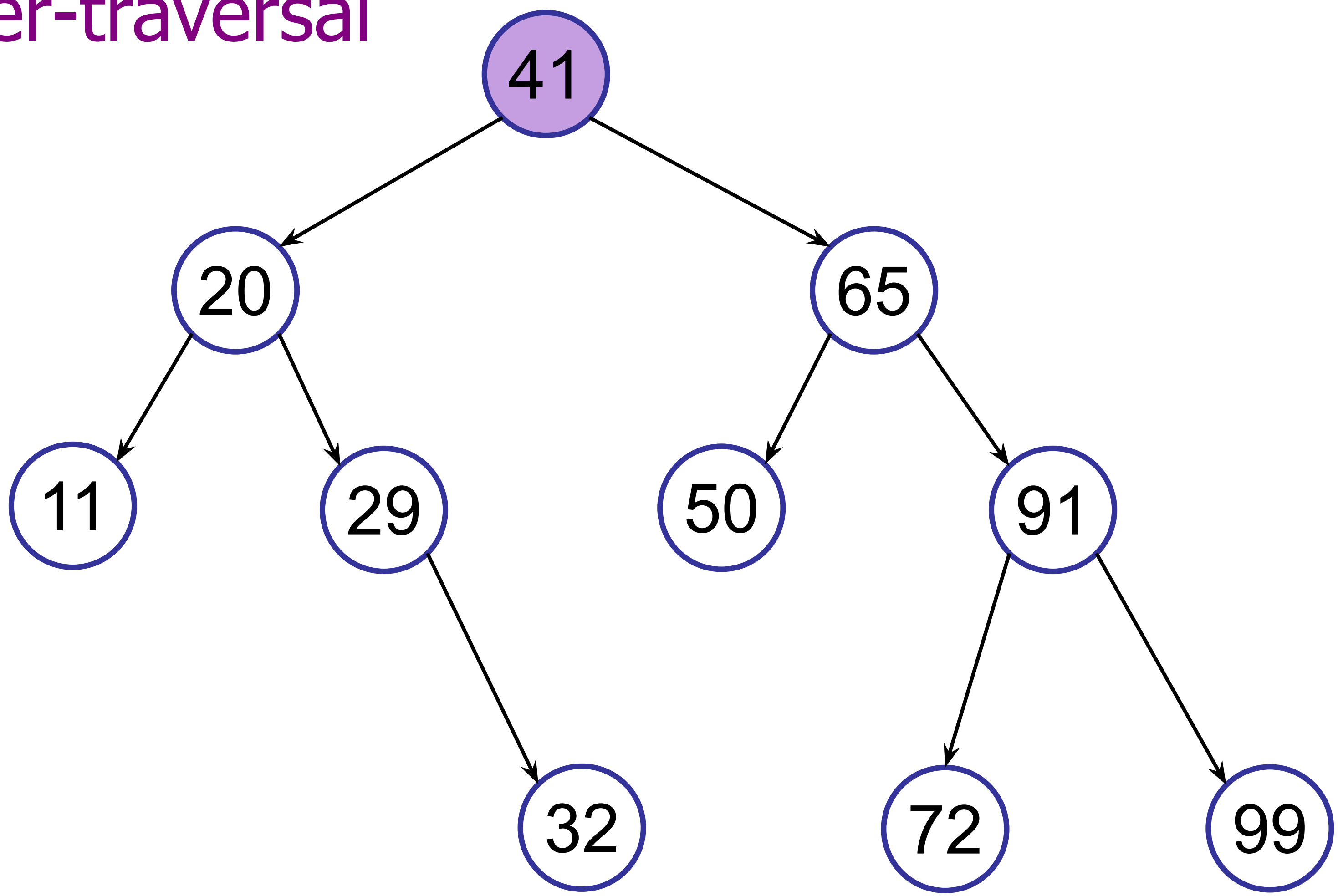
Tree Traversals

level-order-traversal



Tree Traversals

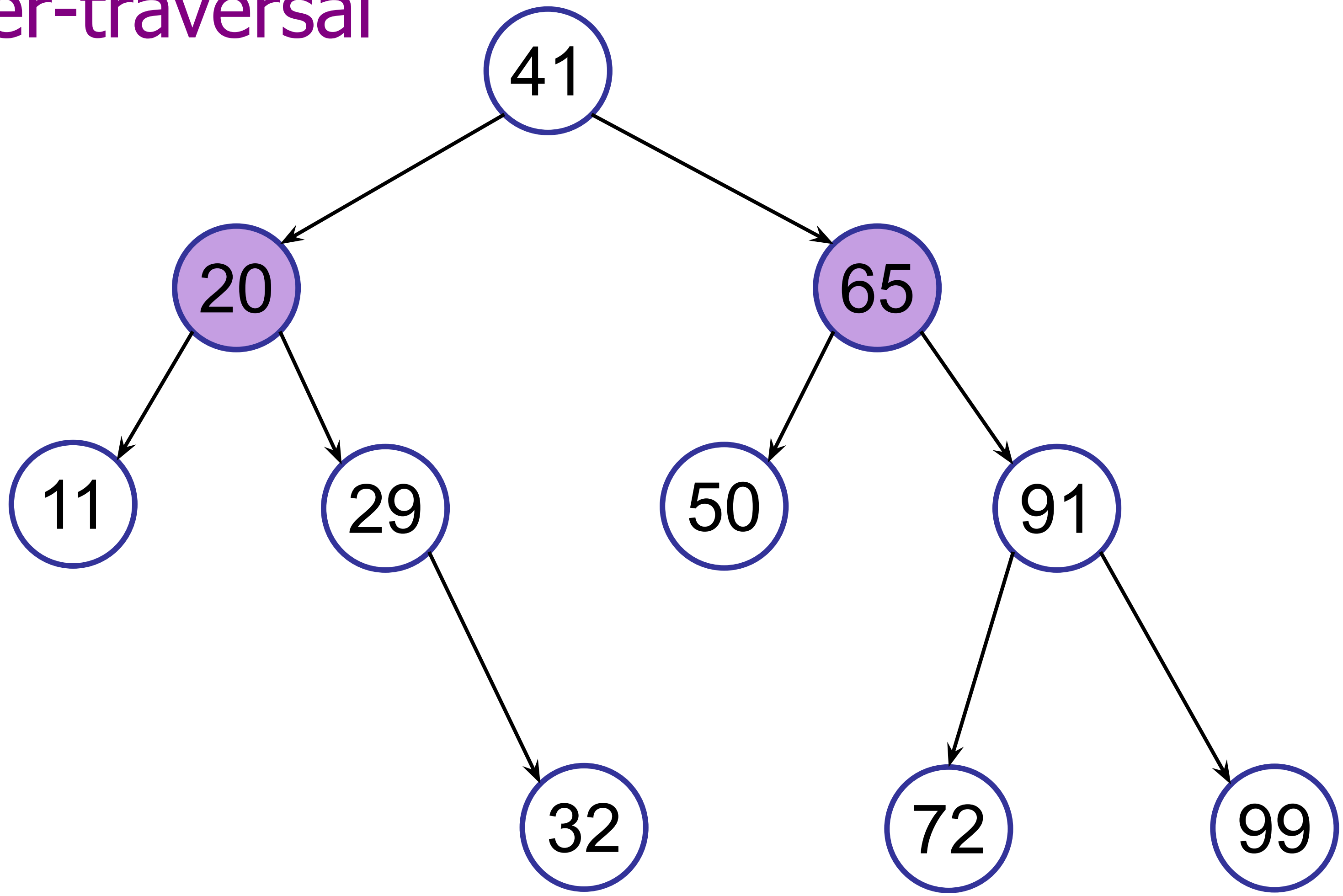
level-order-traversal



41

Tree Traversals

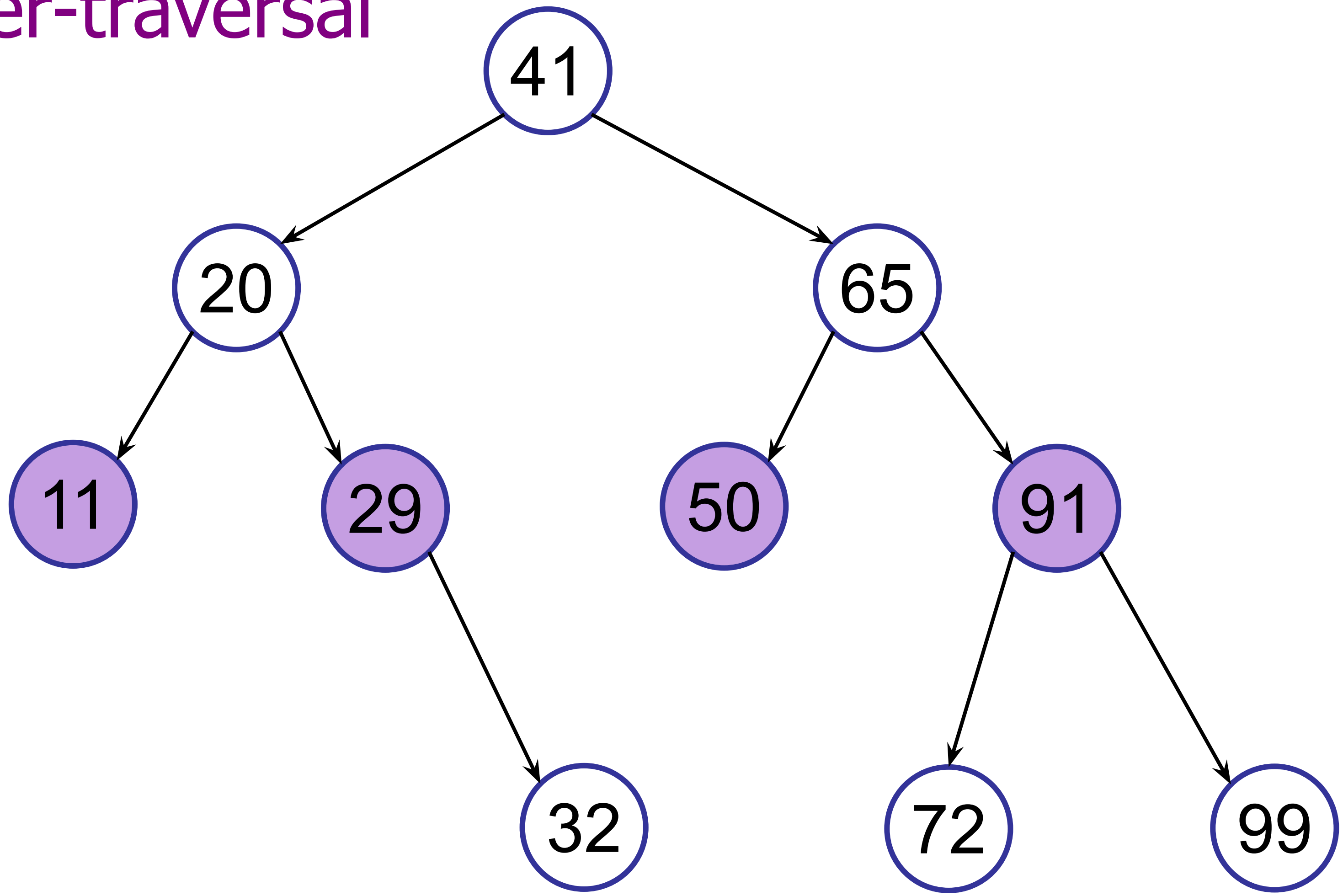
level-order-traversal



41 20 65

Tree Traversals

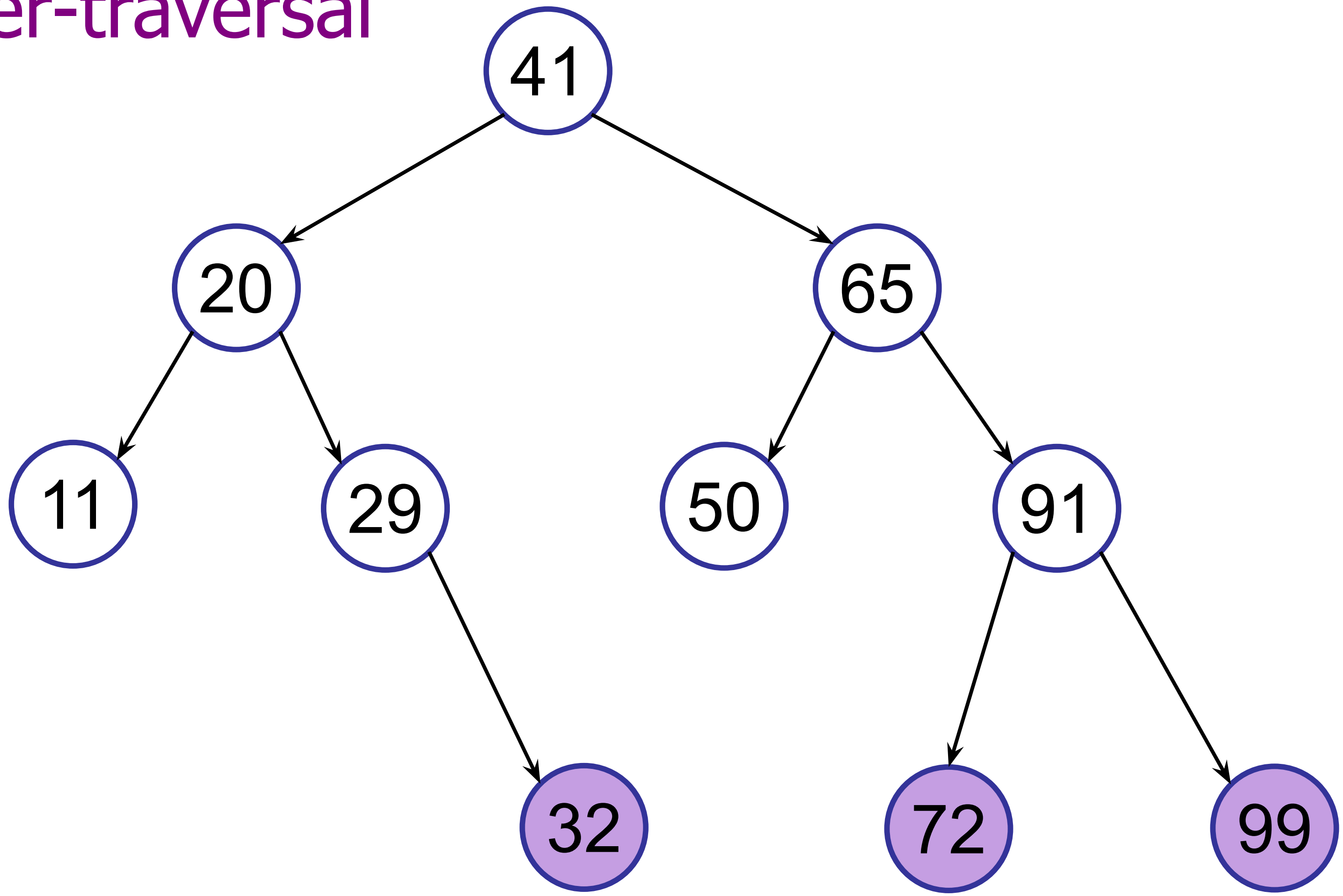
level-order-traversal



41 20 65 11 29 50 91

Tree Traversals

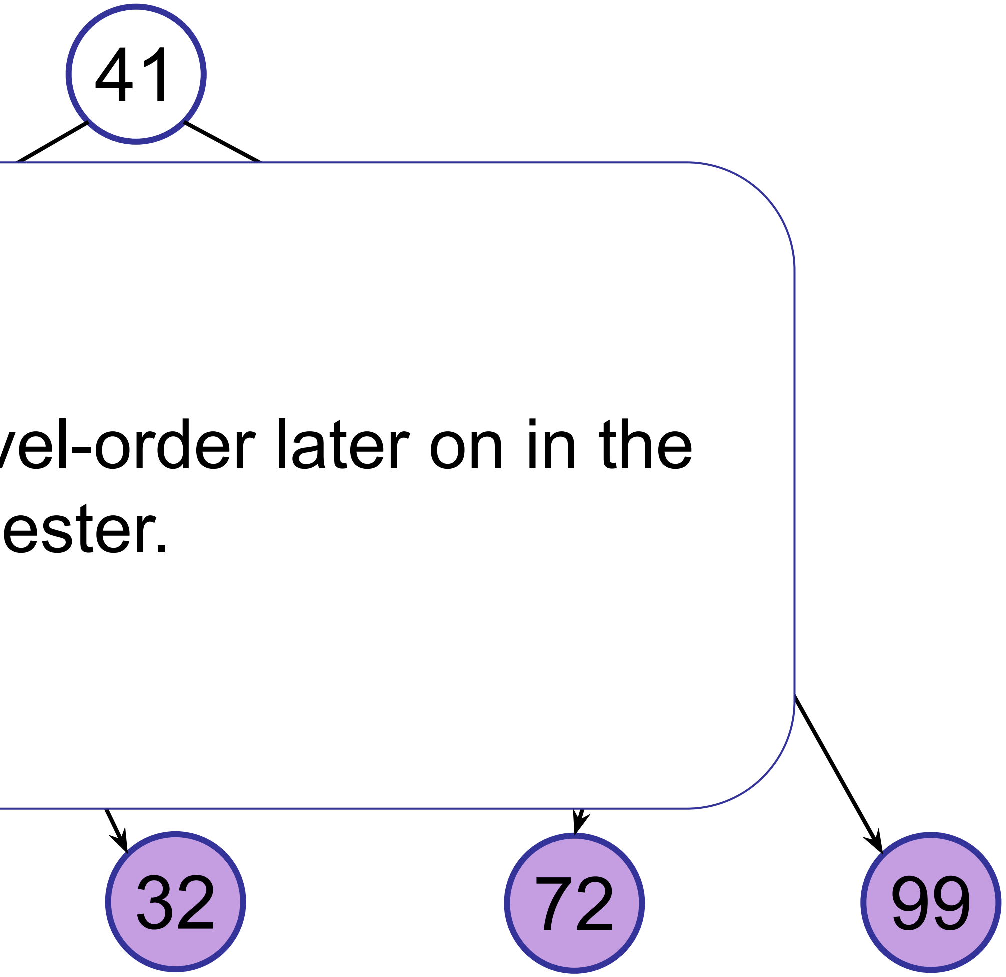
level-order-traversal



41 20 65 11 29 50 91 32 72 99

Tree Traversals

level-order-traversal



Will see how to do level-order later on in the semester.

41 20 65 11 29 50 91 32 72 99

Binary Search Trees

1. Terminology and Definitions

2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

Airport Scheduling

Dictionary

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

Example:

Storing plane departure times in 2400h format in our dictionary.

Airport Scheduling

Dictionary

6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

Use case:

Given some time t, we want to find the next plane that is going to take off.

Airport Scheduling

Dictionary

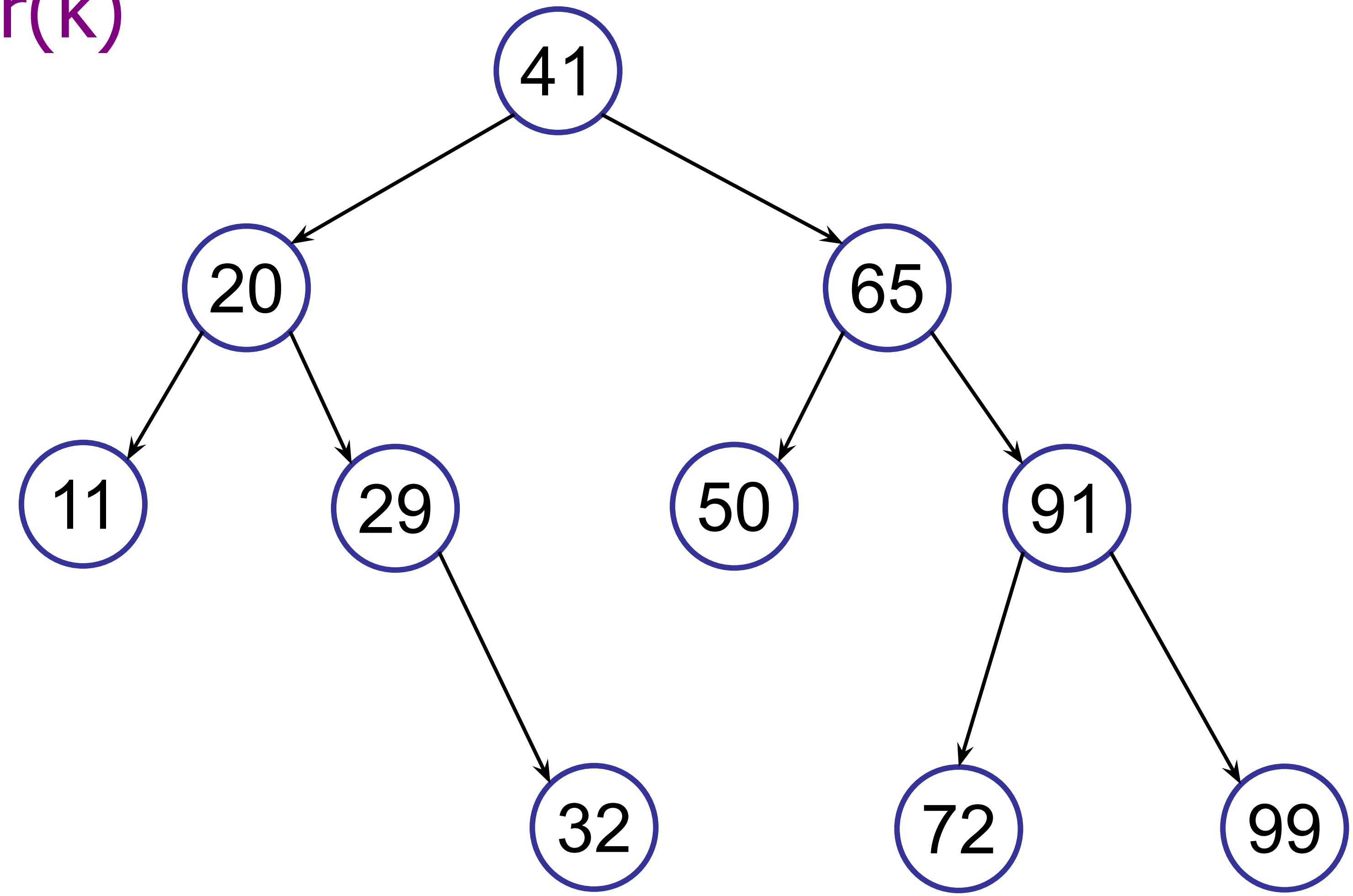
6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- successor(8:24) = 12:21

How do we implement this?

Successor: Key not in the Tree

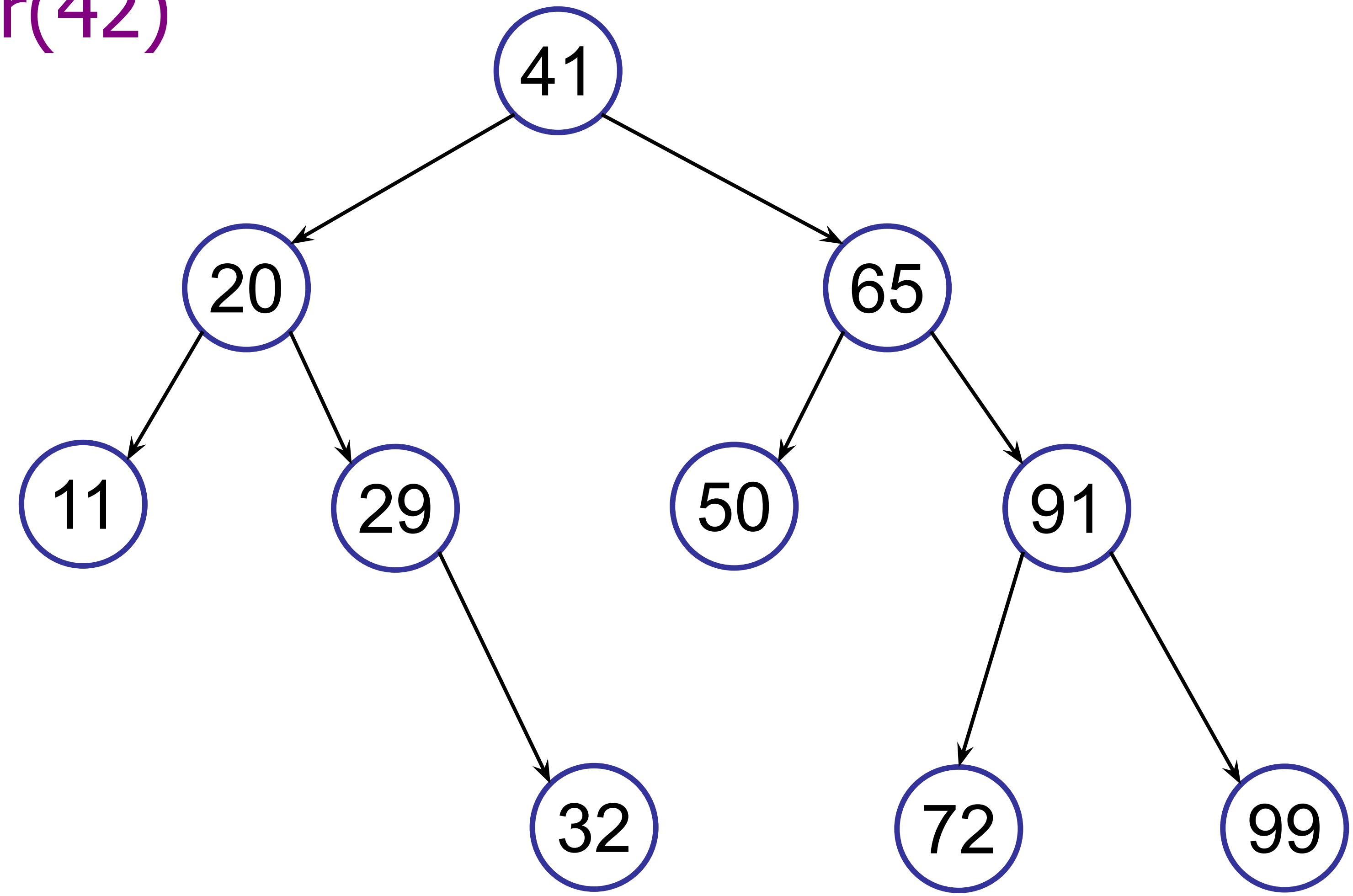
successor(k)



2 possible cases: Either k is in the tree or it's not

Successor: Key not in the Tree

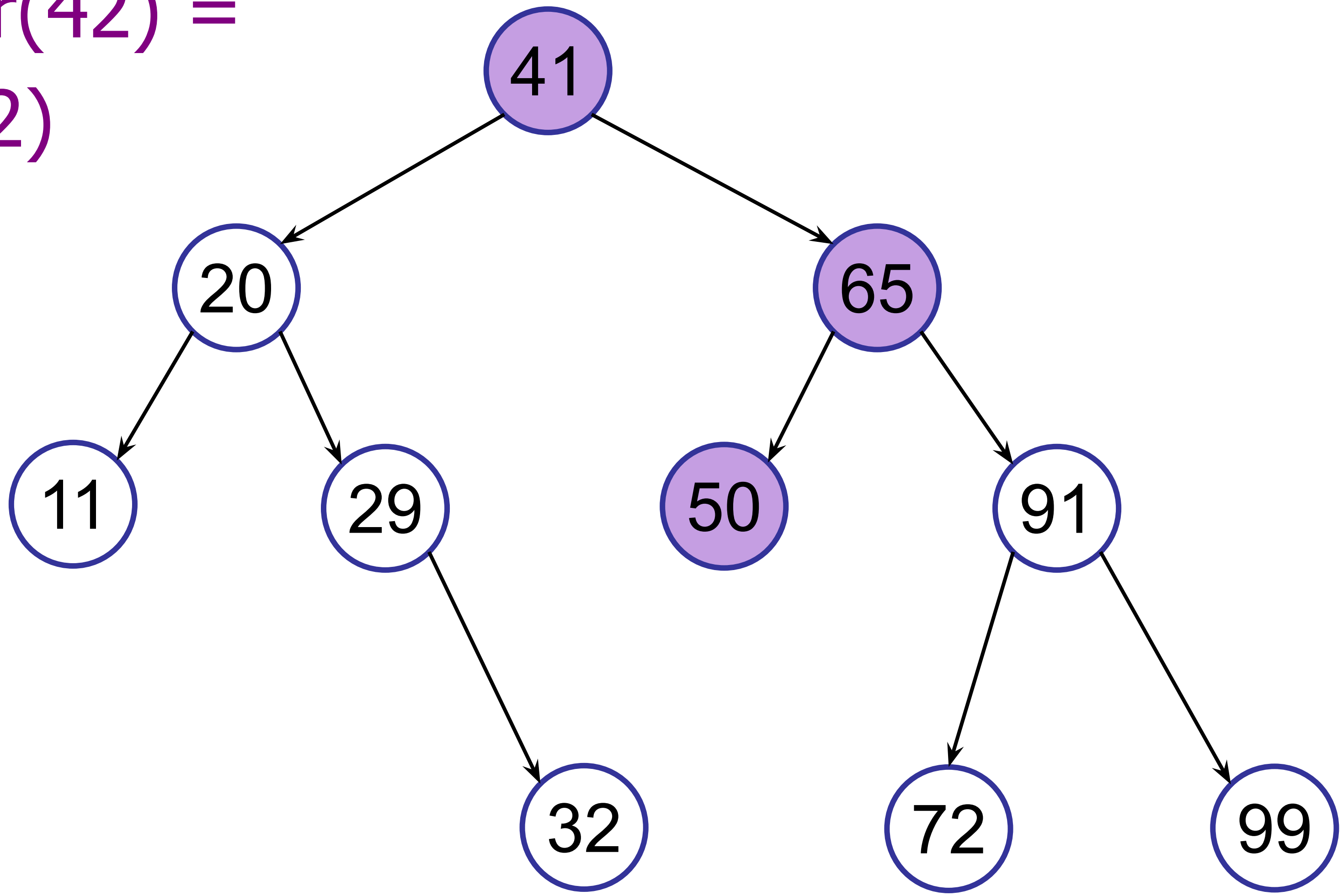
successor(42)



E.g. Key 42 is not in the tree

Successor: Key not in the Tree

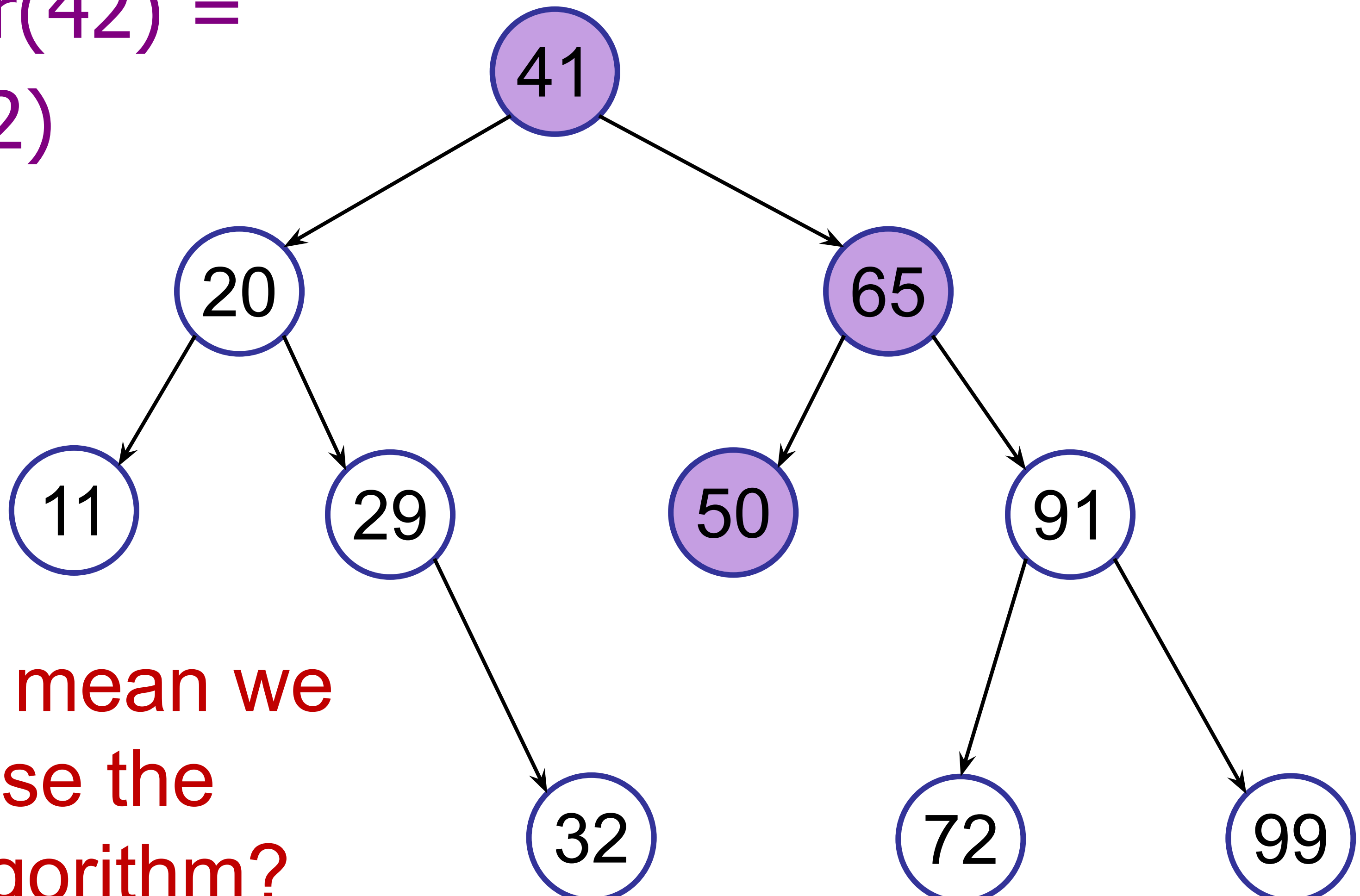
successor(42) =
search(42)



E.g. Key 42 is not in the tree

Successor: Key not in the Tree

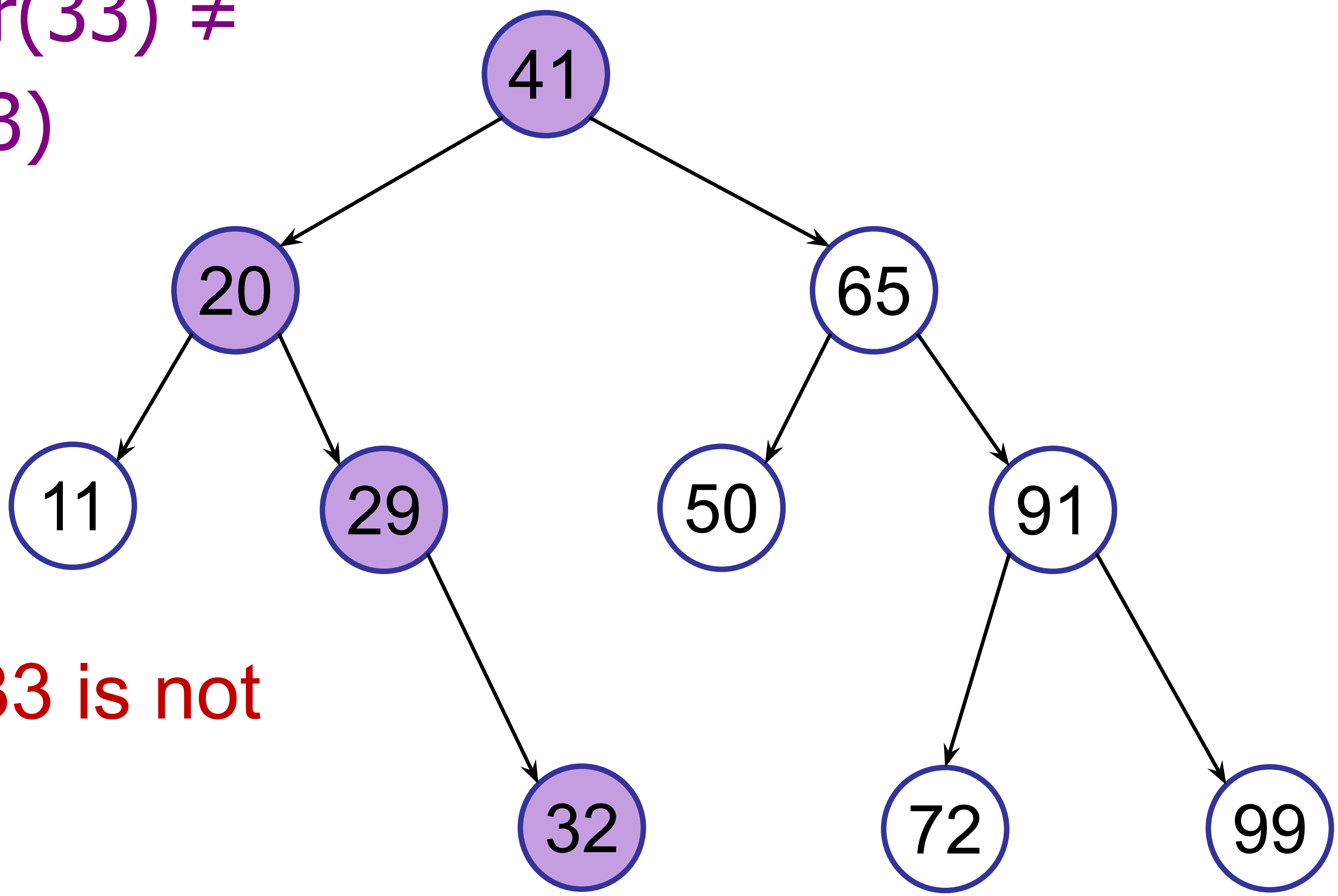
successor(42) =
search(42)



Does this mean we can just use the search algorithm?

Successor: Key not in the Tree

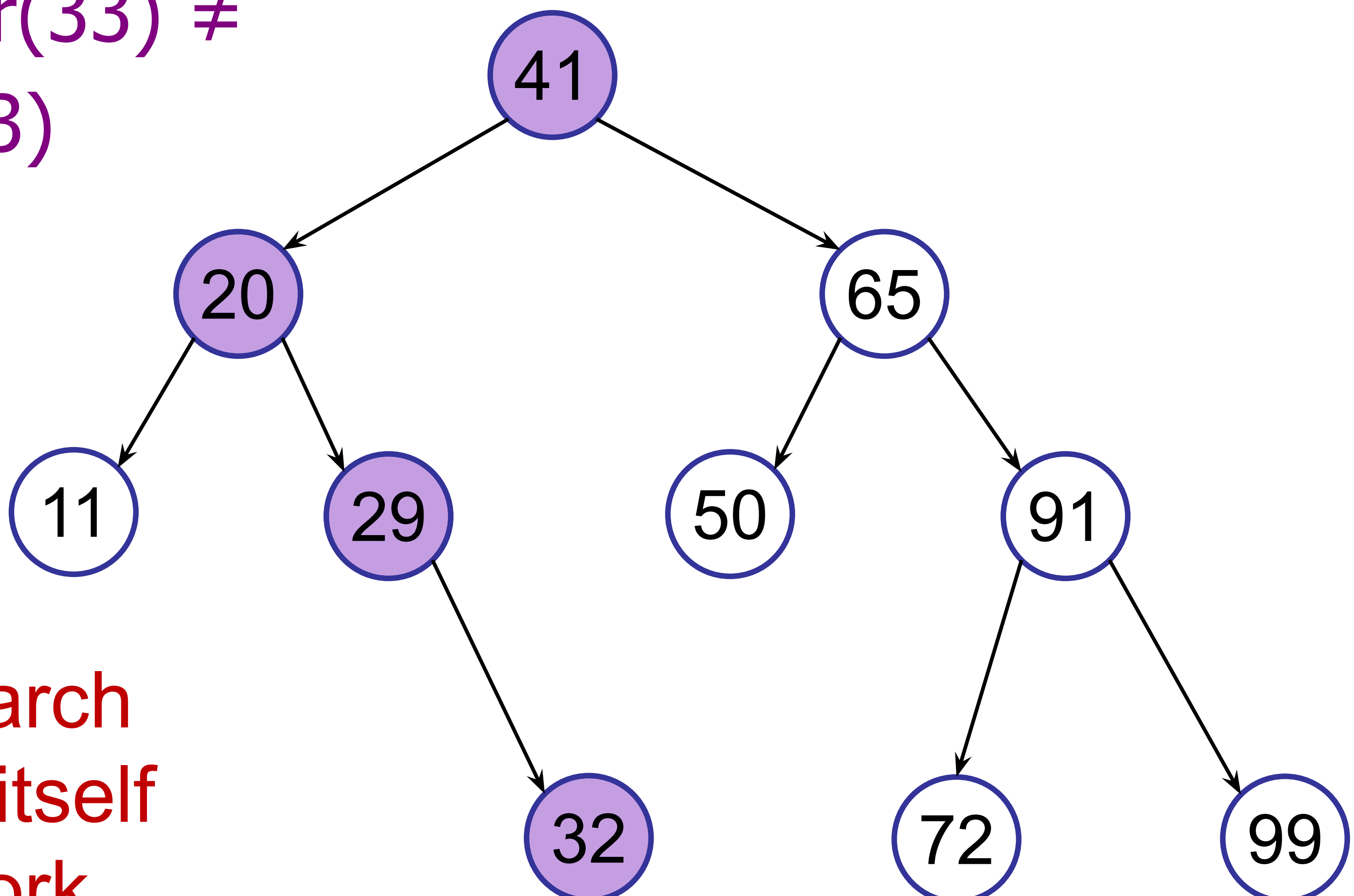
successor(33) ≠ search(33)



E.g. Key 33 is not in the tree

Successor: Key not in the Tree

successor(33) ≠ search(33)

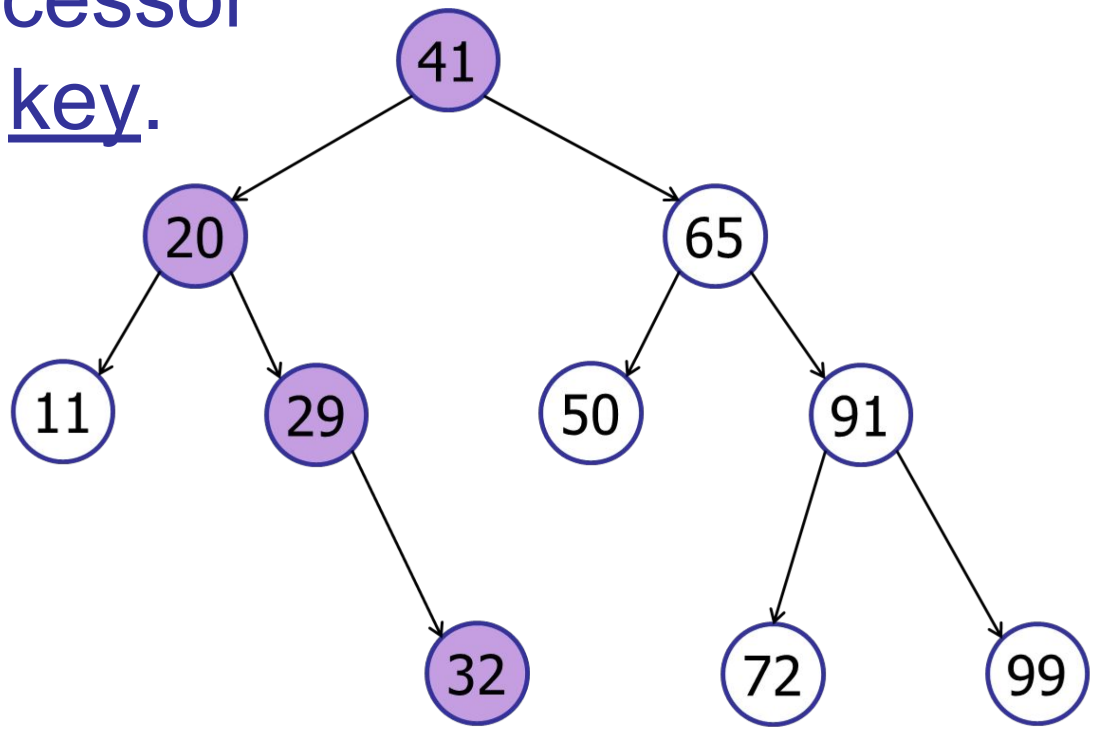


So the search algorithm itself doesn't work to find successors

Successor: Key not in the Tree

But notice: If you search for key not in the tree:

- Either find predecessor or successor of key.

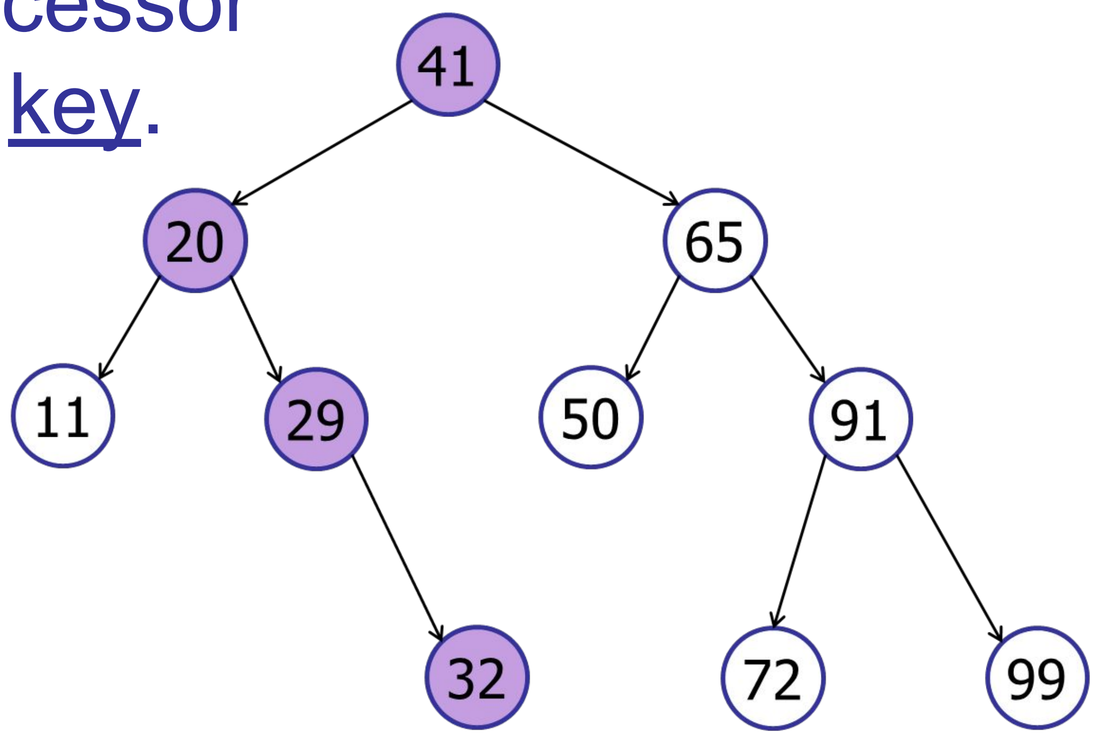


Successor: Key not in the Tree

But notice: If you search for <u>key</u> not in the tree:

- Either find predecessor or successor of <u>key</u>.

Why?

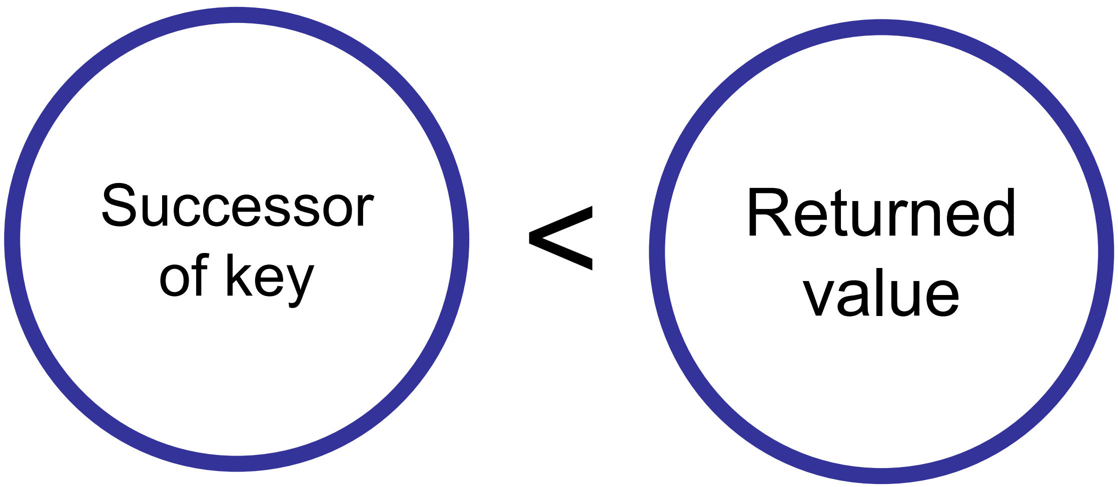


Successor: Key not in the Tree

Assume not:

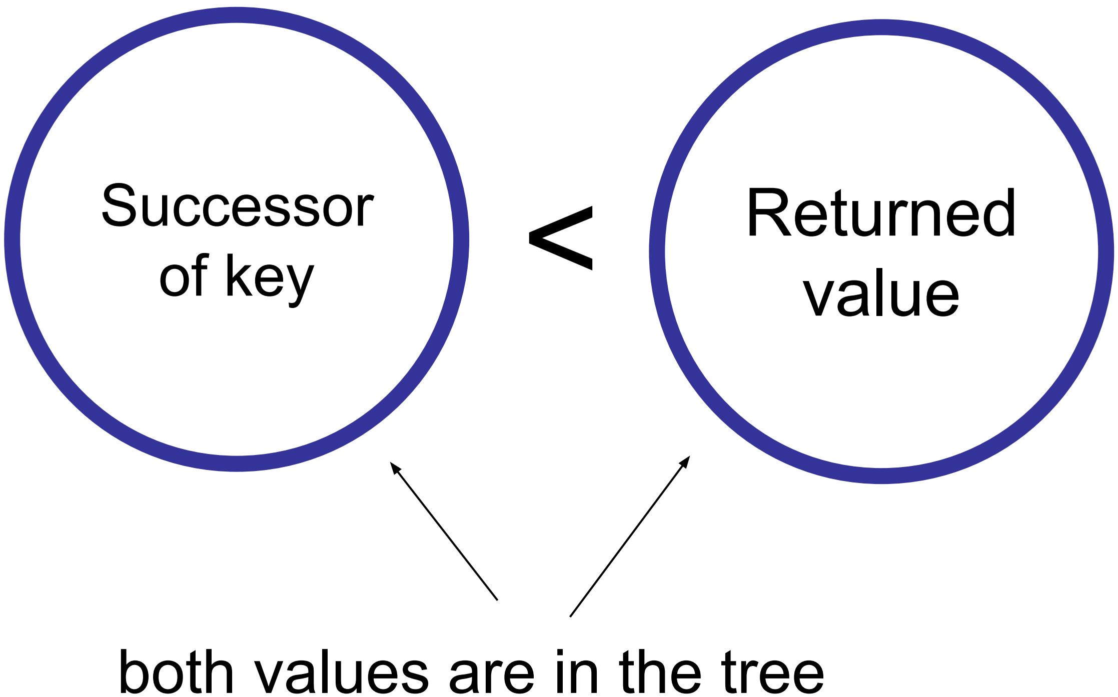
Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



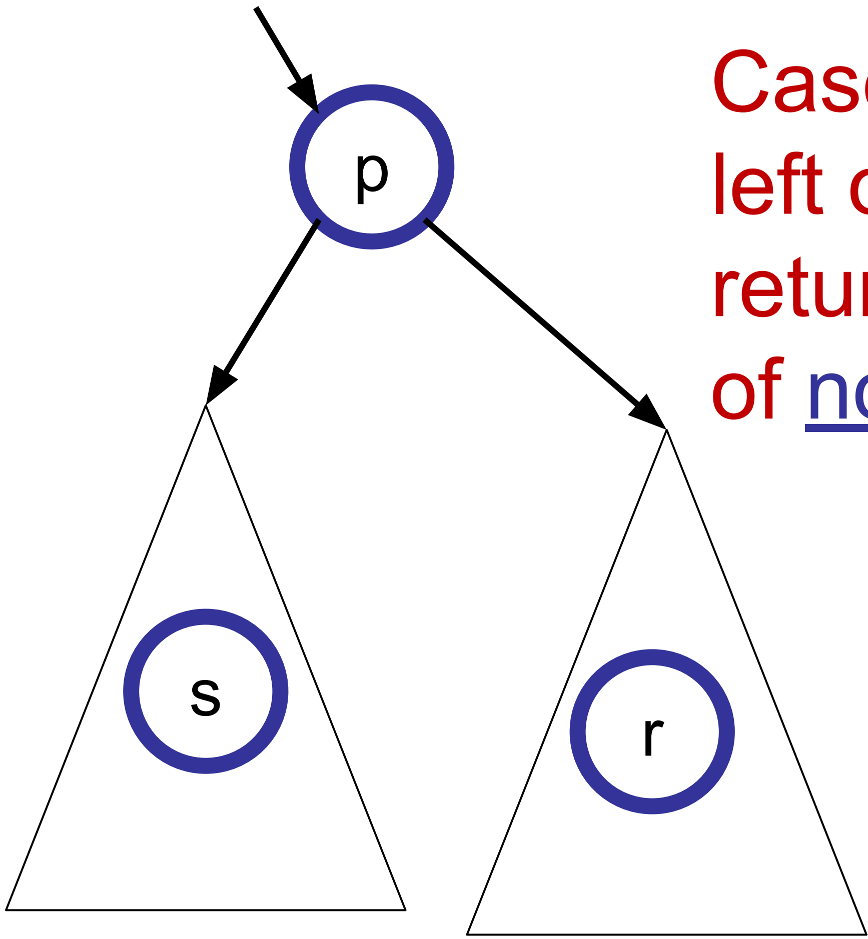
Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



Successor: Key not in the Tree

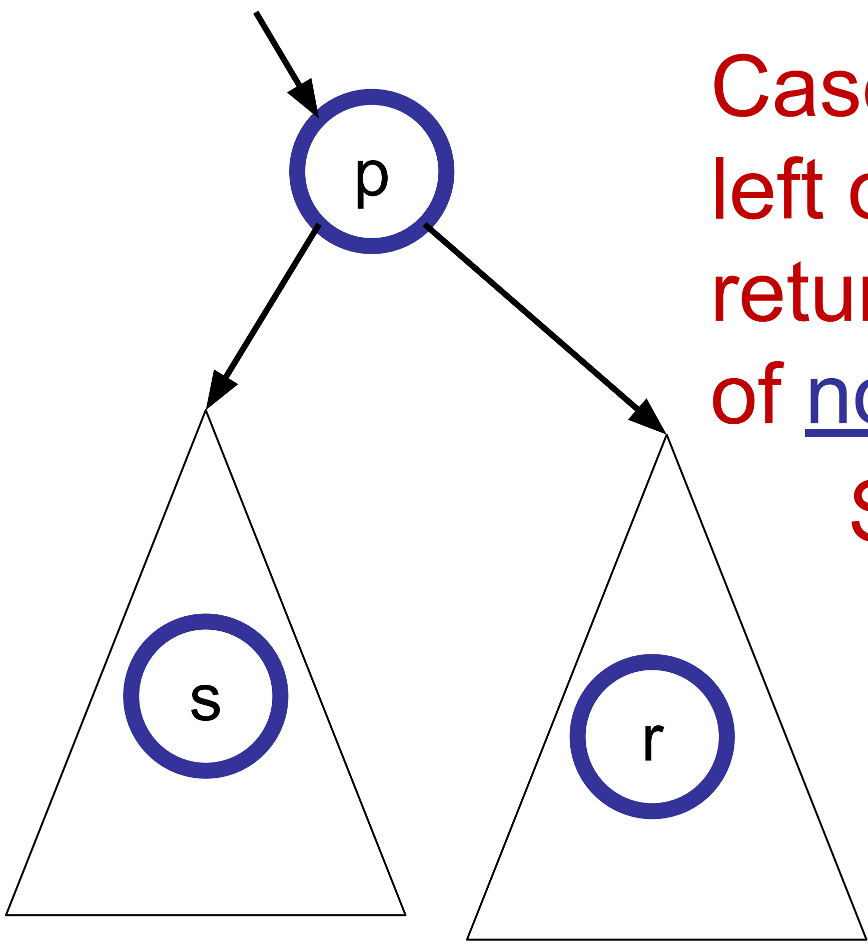
Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



Case 1a: successor node s is left of some node p and returned node r is to the right of node p

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.

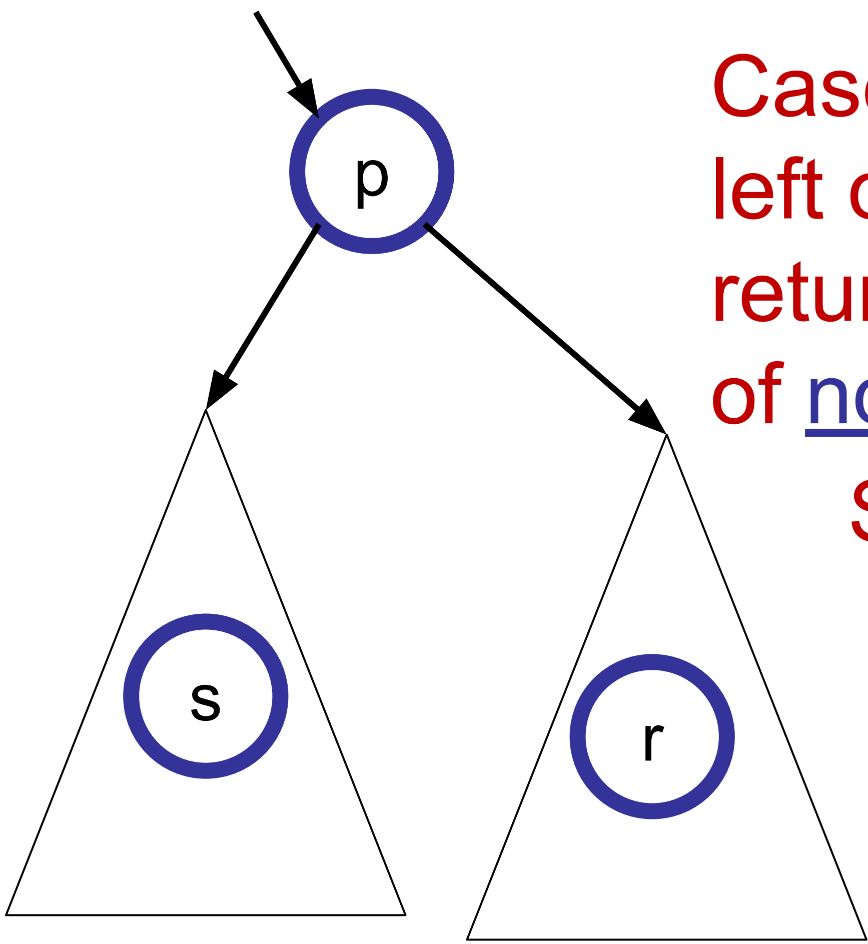


Case 1a: successor node s is left of some node p and returned node r is to the right of node p

Since: key < s < p

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



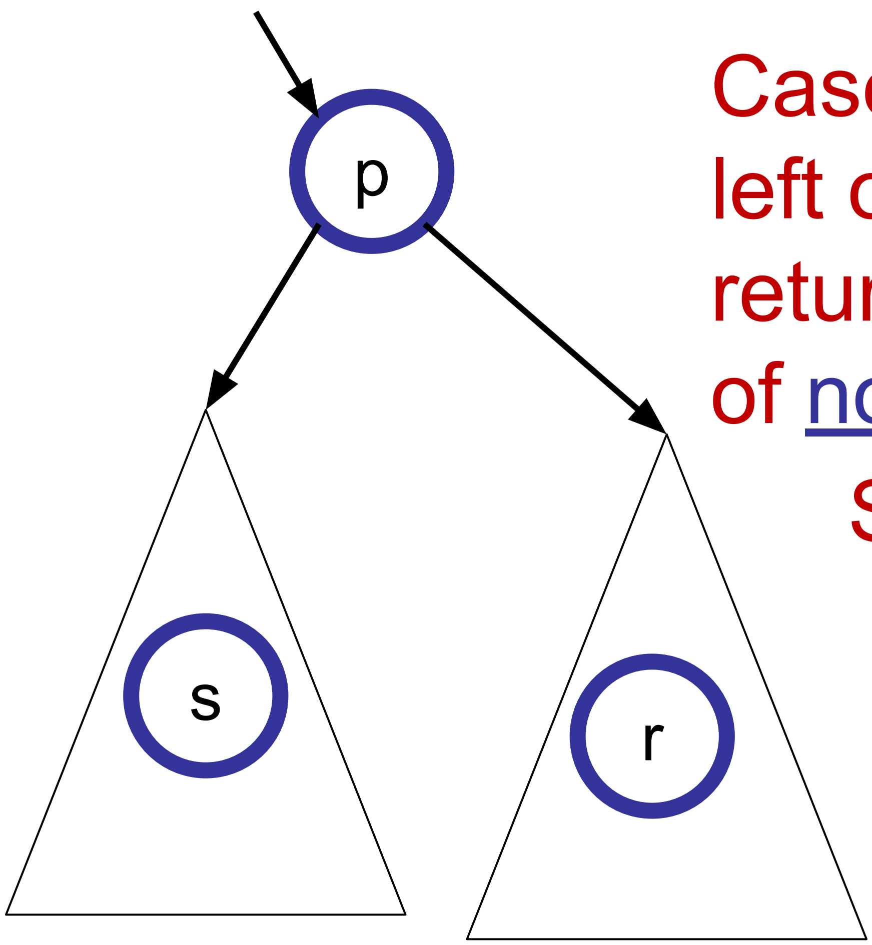
Case 1a: successor node s is left of some node p and returned node r is to the right of node p

Since: key < s < p

Our search algorithm should recurse left

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



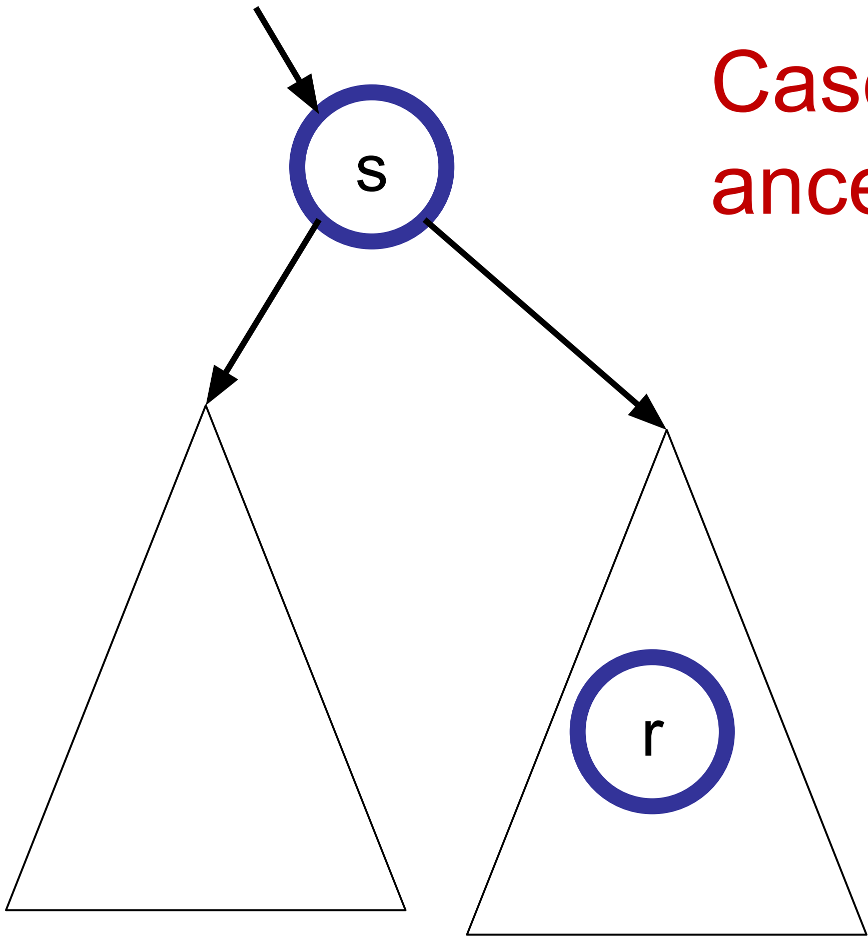
Case 1a: successor node s is left of some node p and returned node r is to the right of node p

Since: key < s < p

Our search algorithm cannot return node r

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



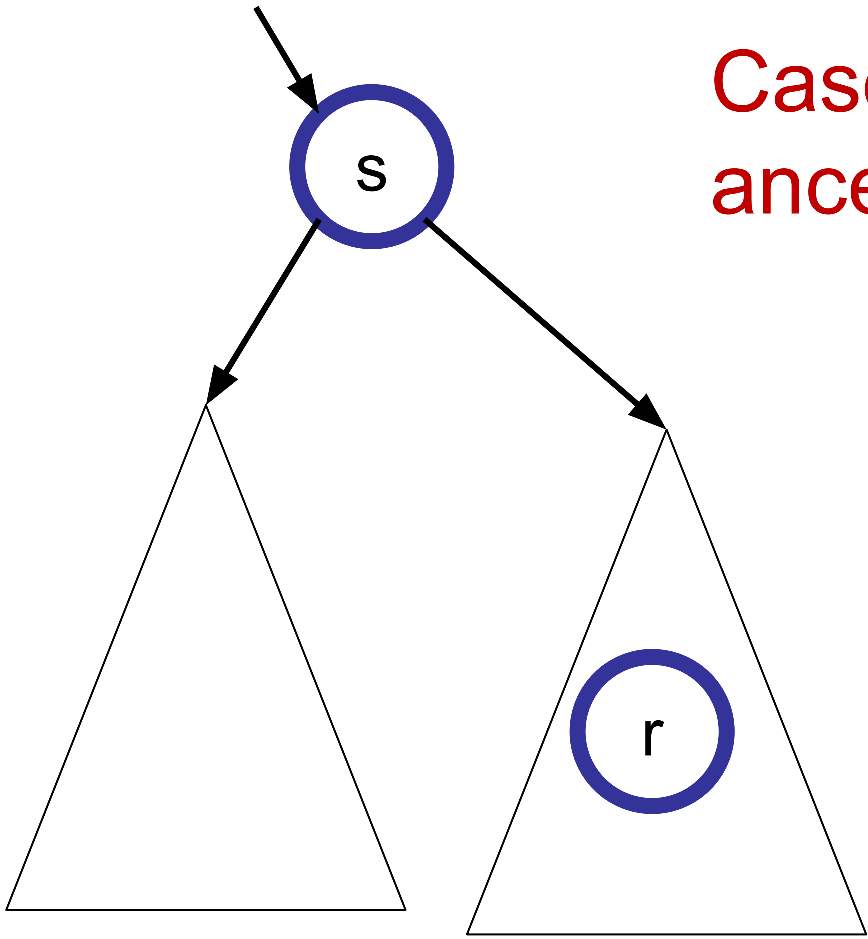
Case 1b: successor node s is ancestor of returned node r

Since: key < s < r

Our search algorithm should recurse left

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



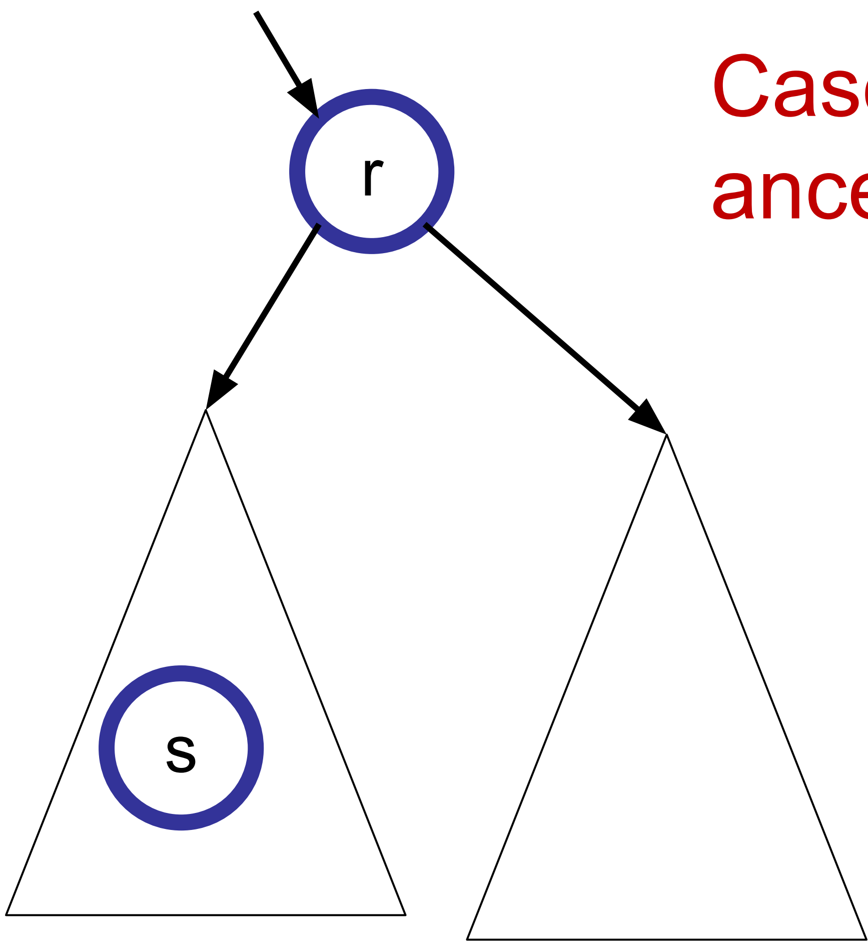
Case 1b: successor node s is ancestor of returned node r

Since: key < s < r

Our search algorithm cannot return node r

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



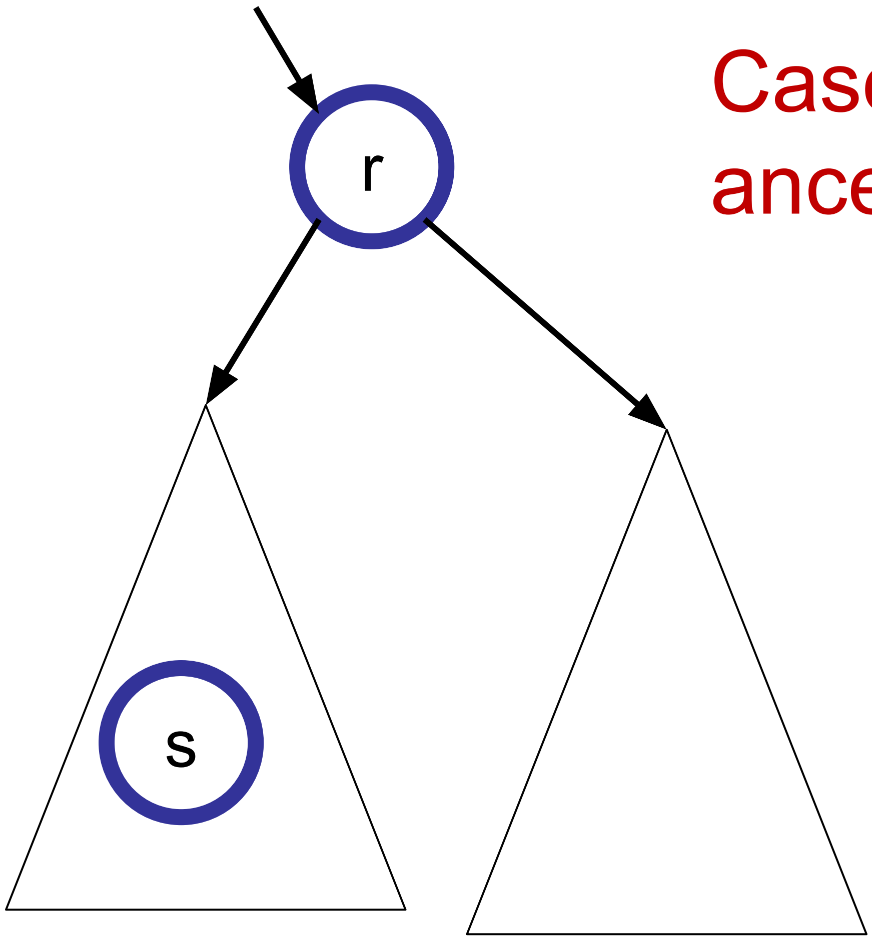
Case 1c: successor node s is ancestor of returned node r

Since: key < s < r

Our search algorithm should recurse left

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



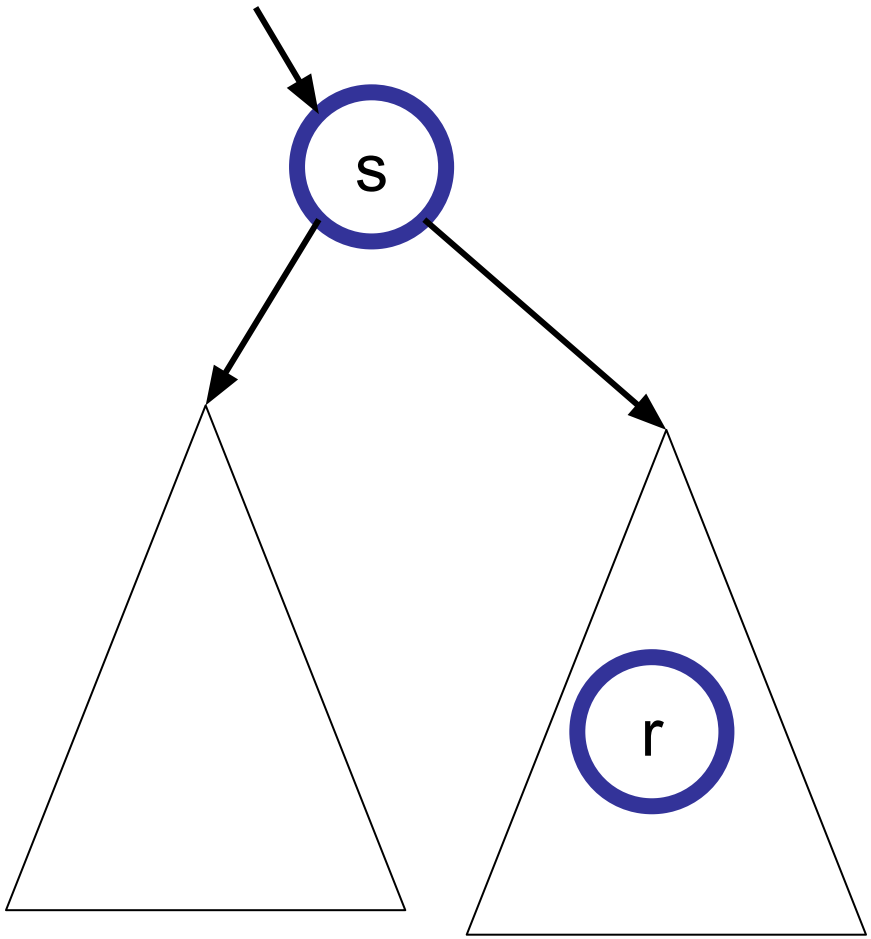
Case 1c: successor node s is ancestor of returned node r

Since: key < s < r

Our search algorithm cannot return node r

Successor: Key not in the Tree

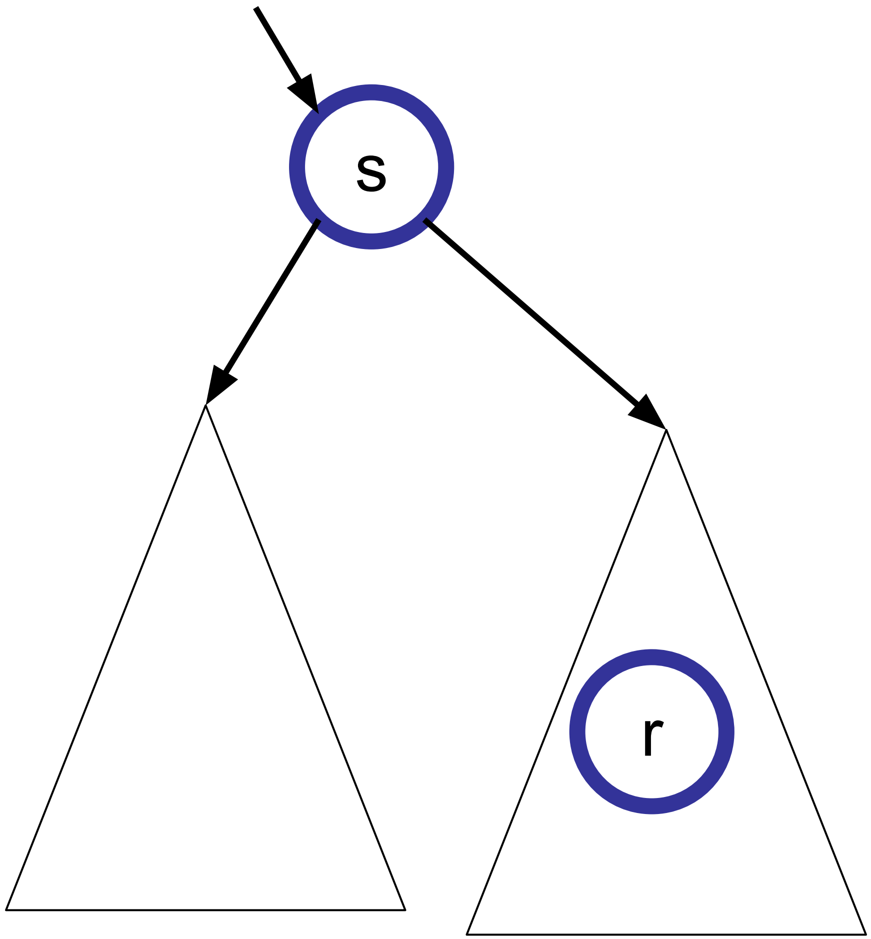
Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



Case 1 derives a contradiction!

Successor: Key not in the Tree

Assume not: Case 1, search(key) returns node that is larger than actual successor of key.



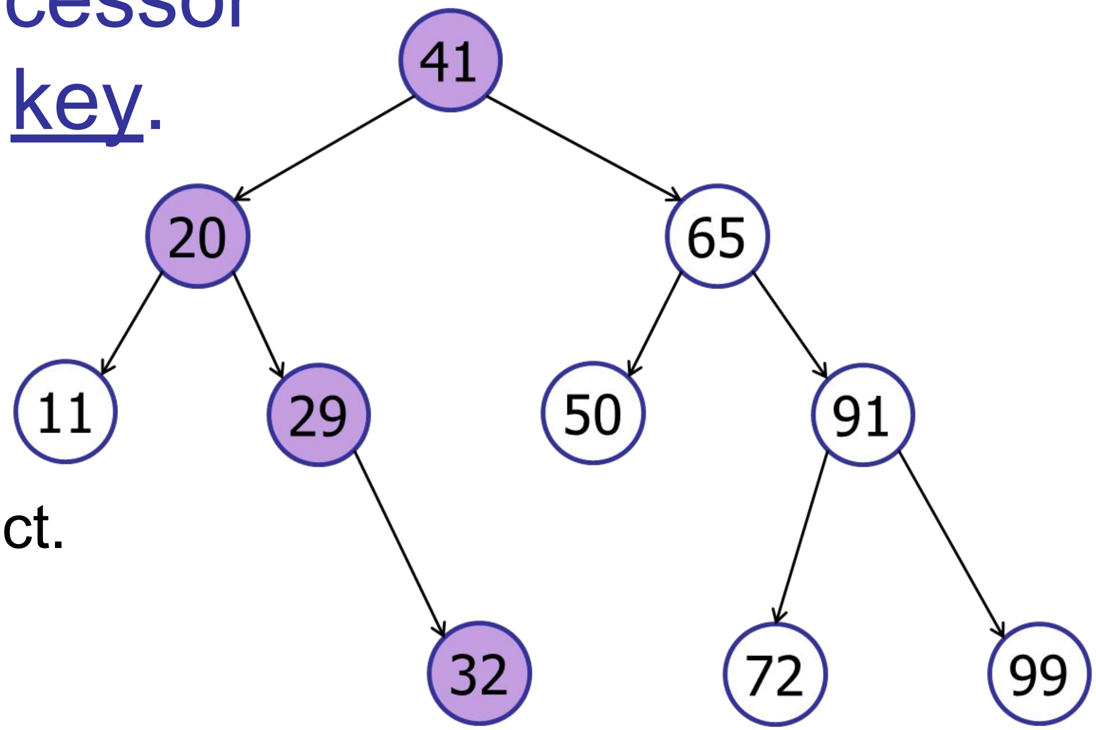
You can argue similarly in case 2 where search(key) returns node that is smaller than predecessor of key

Successor: Key not in the Tree

But notice: If you search for <u>key</u> not in the tree:

- Either find predecessor or successor of <u>key</u>.

we will make use of this fact.



Successor Queries

Basic strategy: successor(key)

1. Search for key in the tree.
2. If (result > key), then return result.
3. If (result <= key), then search for successor of result.

Successor Queries

Basic strategy: successor(key)

proven it is indeed the successor

1. Search for key in the tree.
2. If (result > key), then return result.
3. If (result <= key), then search for successor of result.

Successor Queries

Basic strategy: successor(key)

proven it is indeed the successor

1. Search for key in the tree.

2. If (result > key), then return result.

3. If (result <= key), then search for successor of result.

if result == key, successor(key) is the true successor

Successor Queries

Basic strategy: successor(key)

proven it is indeed the successor

1. Search for key in the tree.

2. If (result > key), then return result.

3. If (result <= key), then search for successor of result.

if result == key, successor(key) is the true successor

if result < key, then successor(result) is the first smallest result > key (because key was not in tree!)

Successor Queries

Basic strategy: successor(key)

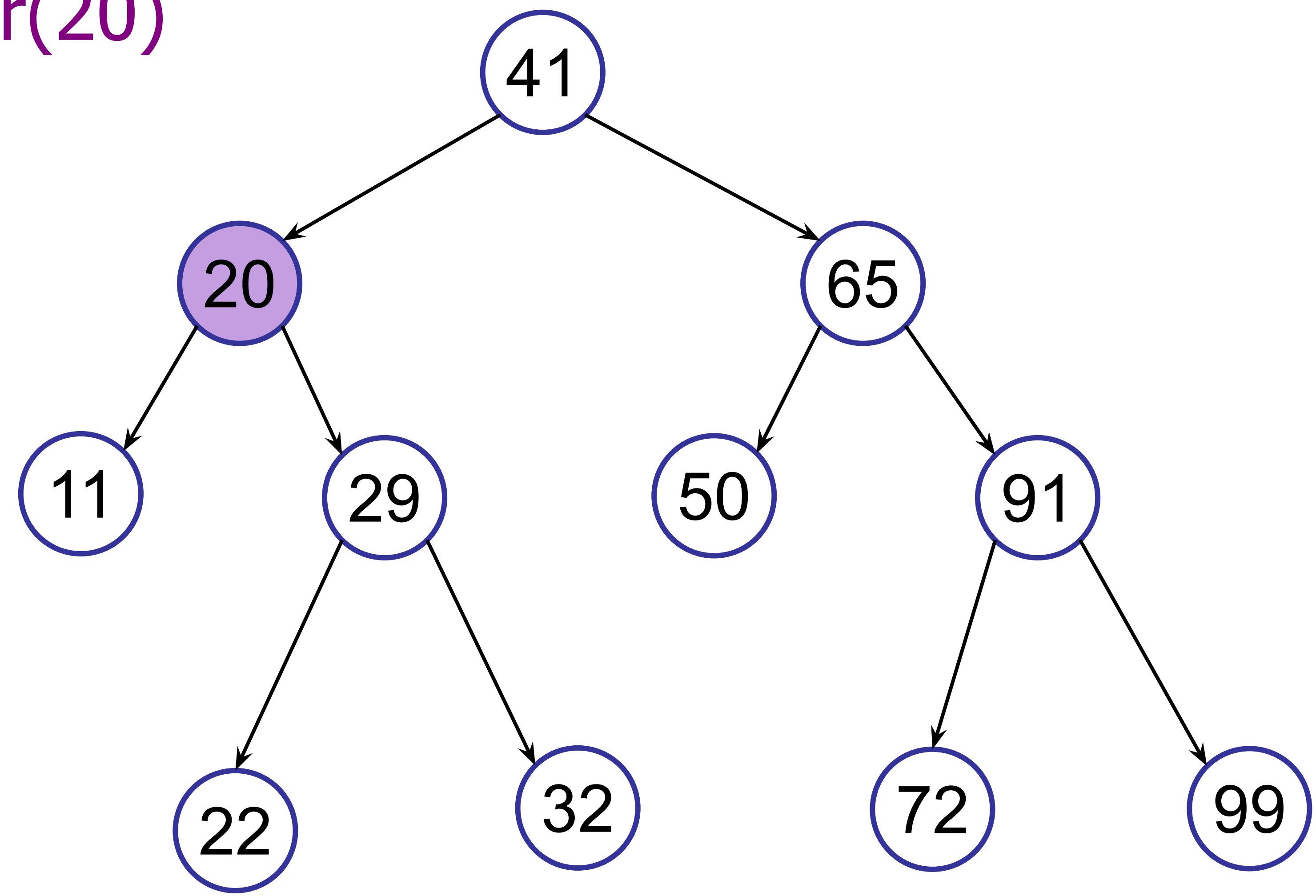
proven it is indeed the successor

1. Search for key in the tree.
2. If (result > key), then return result.
3. If (result <= key), then search for successor of result.

In the bottom case: we are searching for a **successor of an item that is guaranteed to be in a tree.**
Not the same problem as before where item was not in the tree!

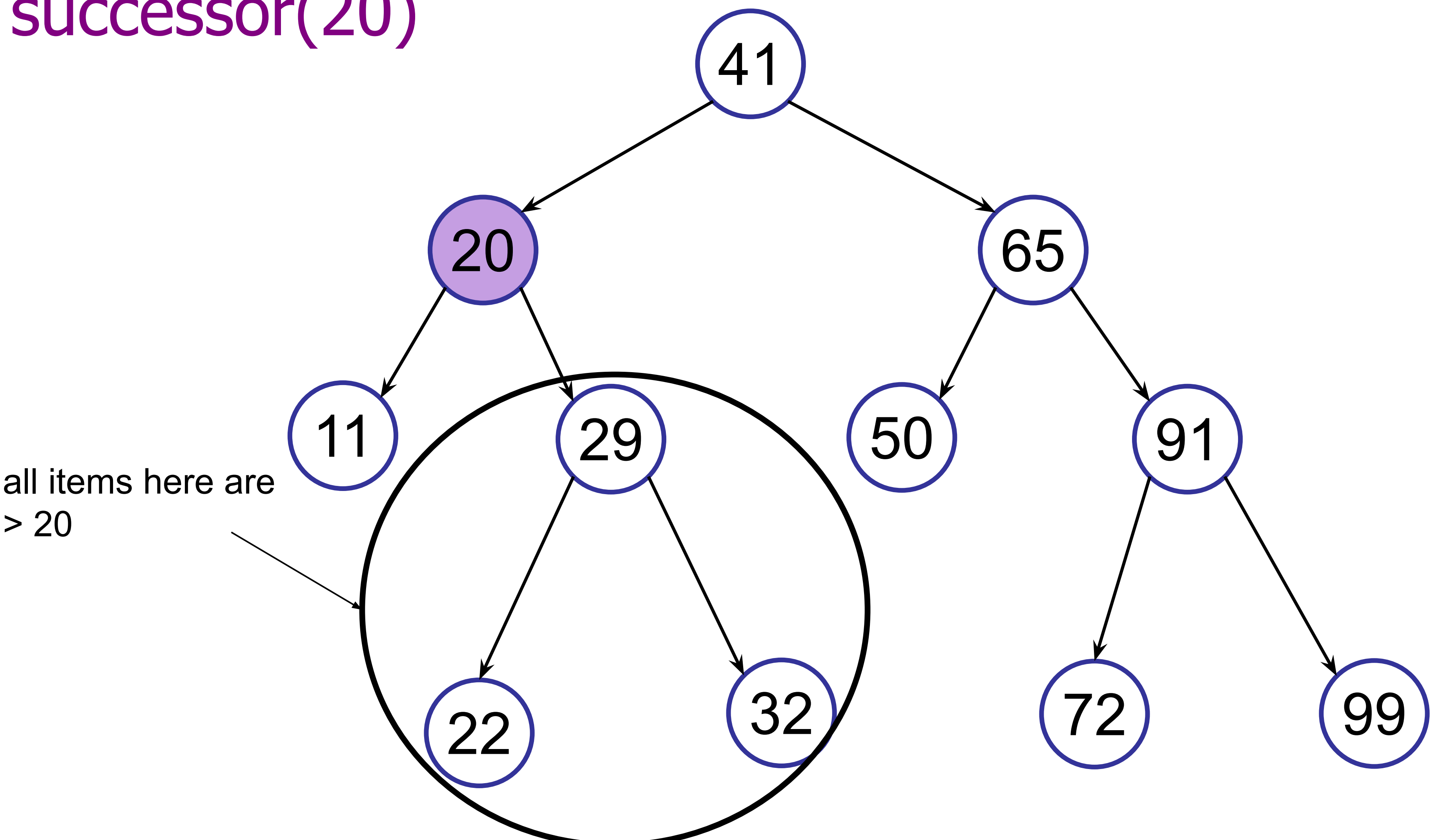
Successor: Key in the Tree

successor(20)



Successor Queries

successor(20)

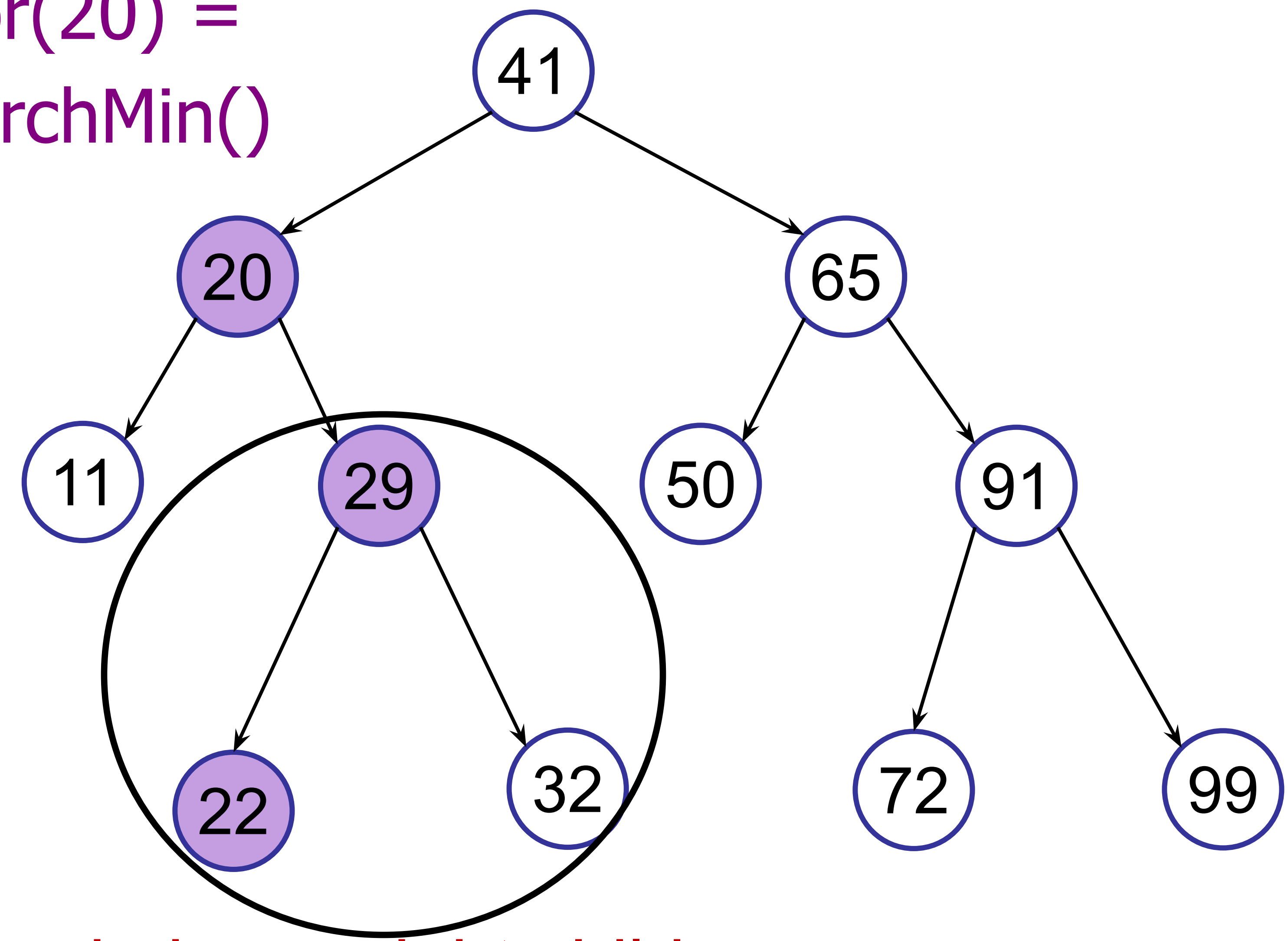


Case 1: node has a right child.

Successor Queries

successor(20) =
right.searchMin()

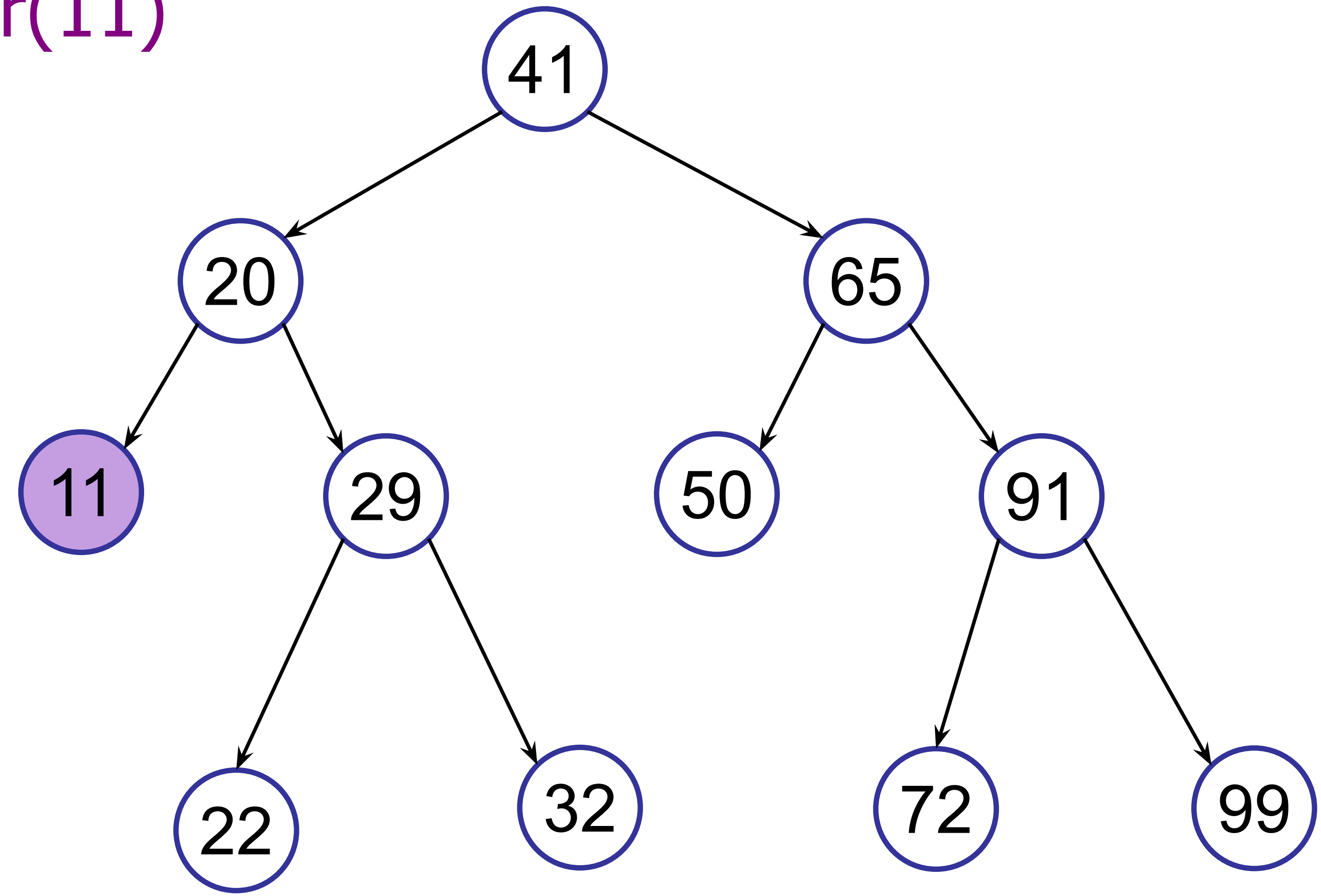
smallest item in
right subtree



Case 1: node has a right child.

Successor Queries

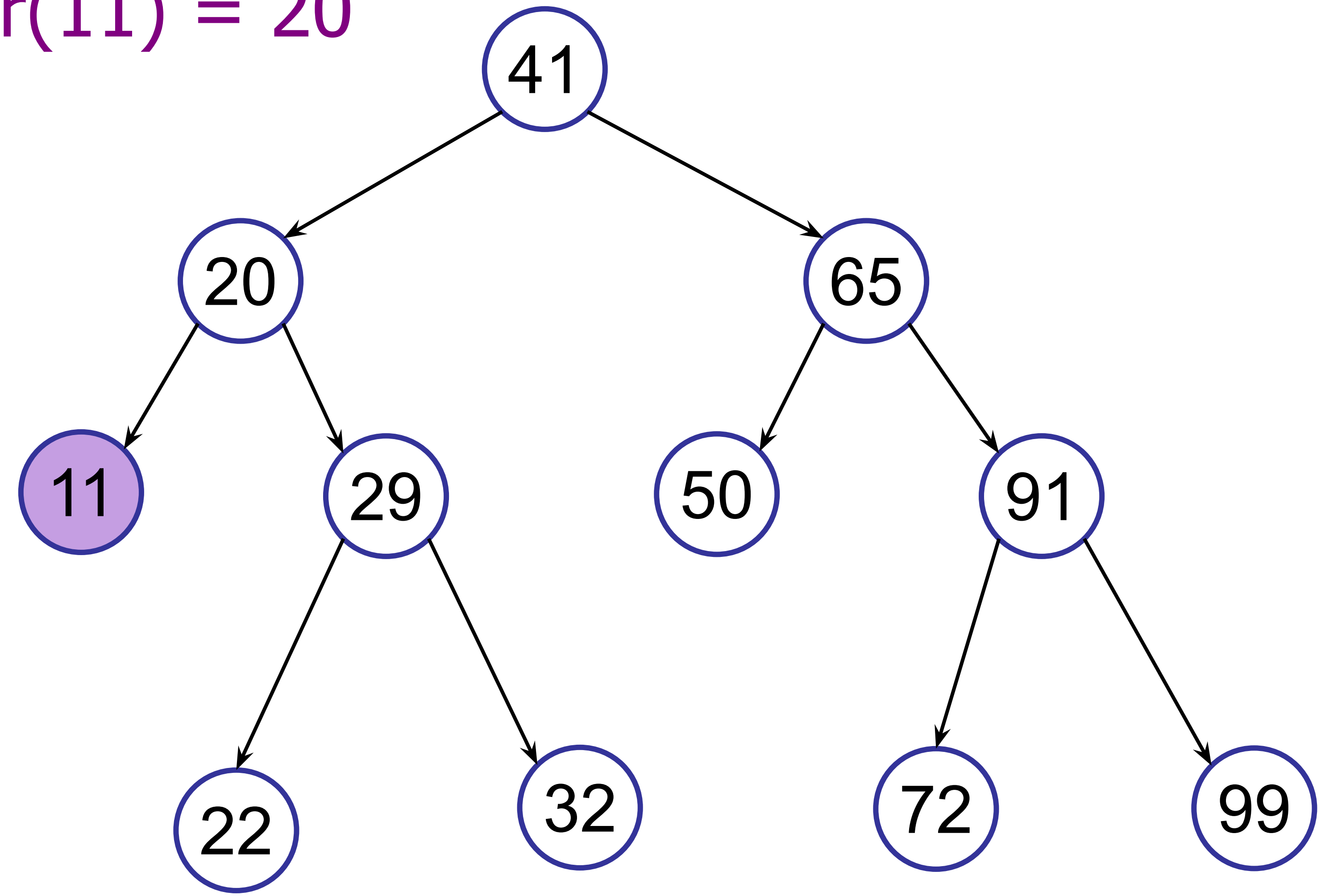
successor(11)



Case 2: node has no right child.

Successor Queries

successor(11) = 20



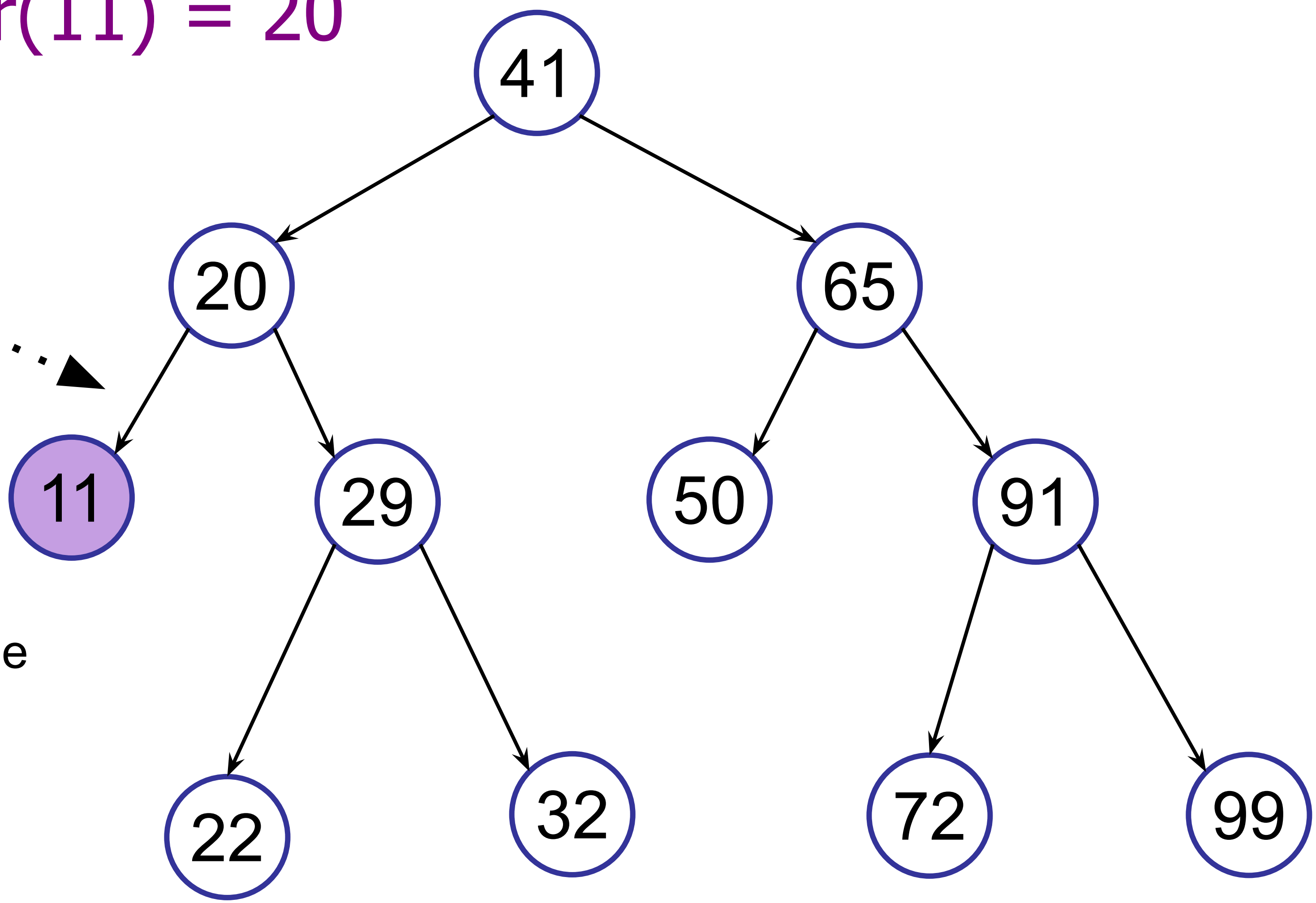
Case 2: node has no right child.

Successor Queries

successor(11) = 20

11 is the left child of its parent.

so parent is the first value bigger than it

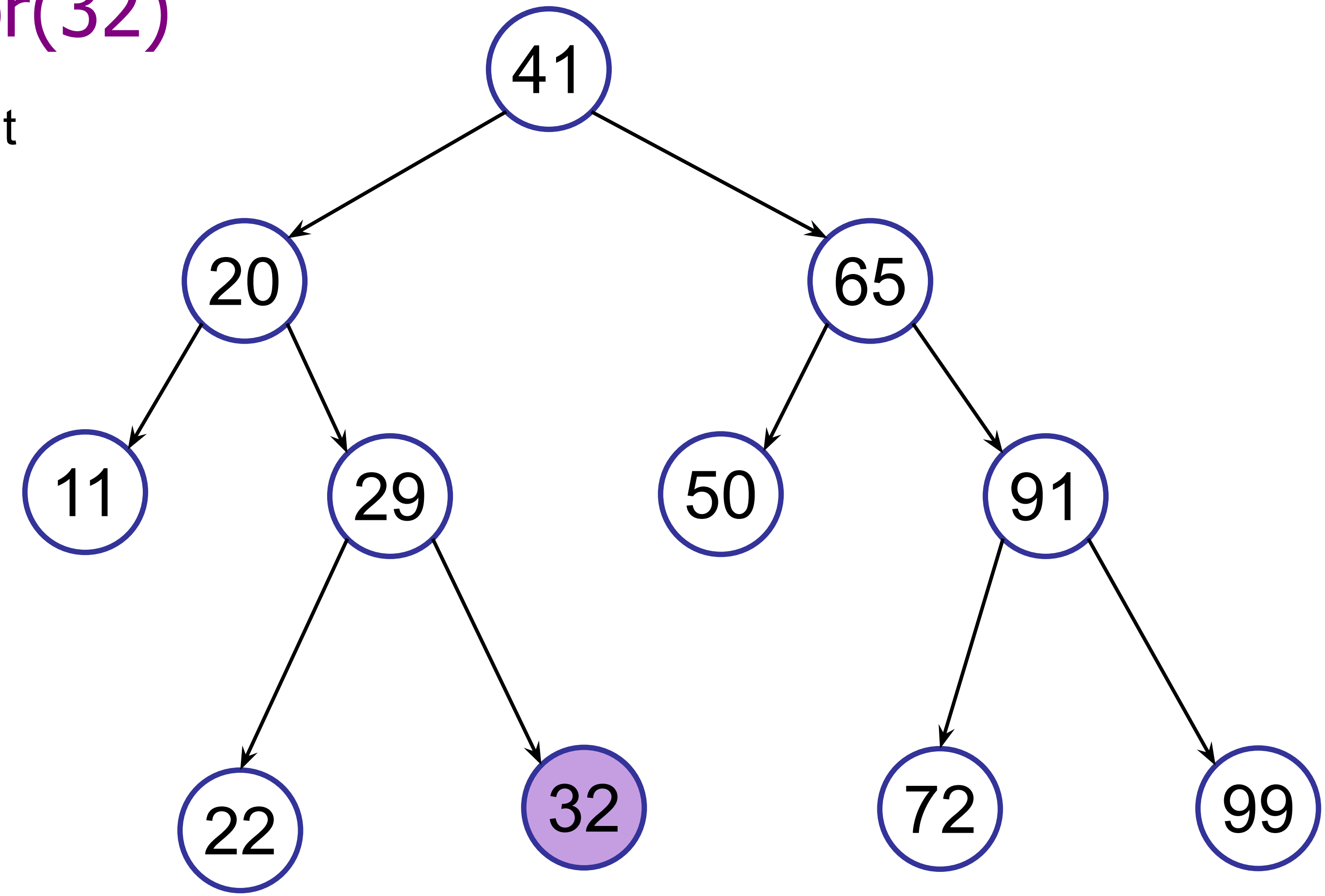


Case 2: node has no right child.

Successor Queries

successor(32)

32 is the right child of its parent



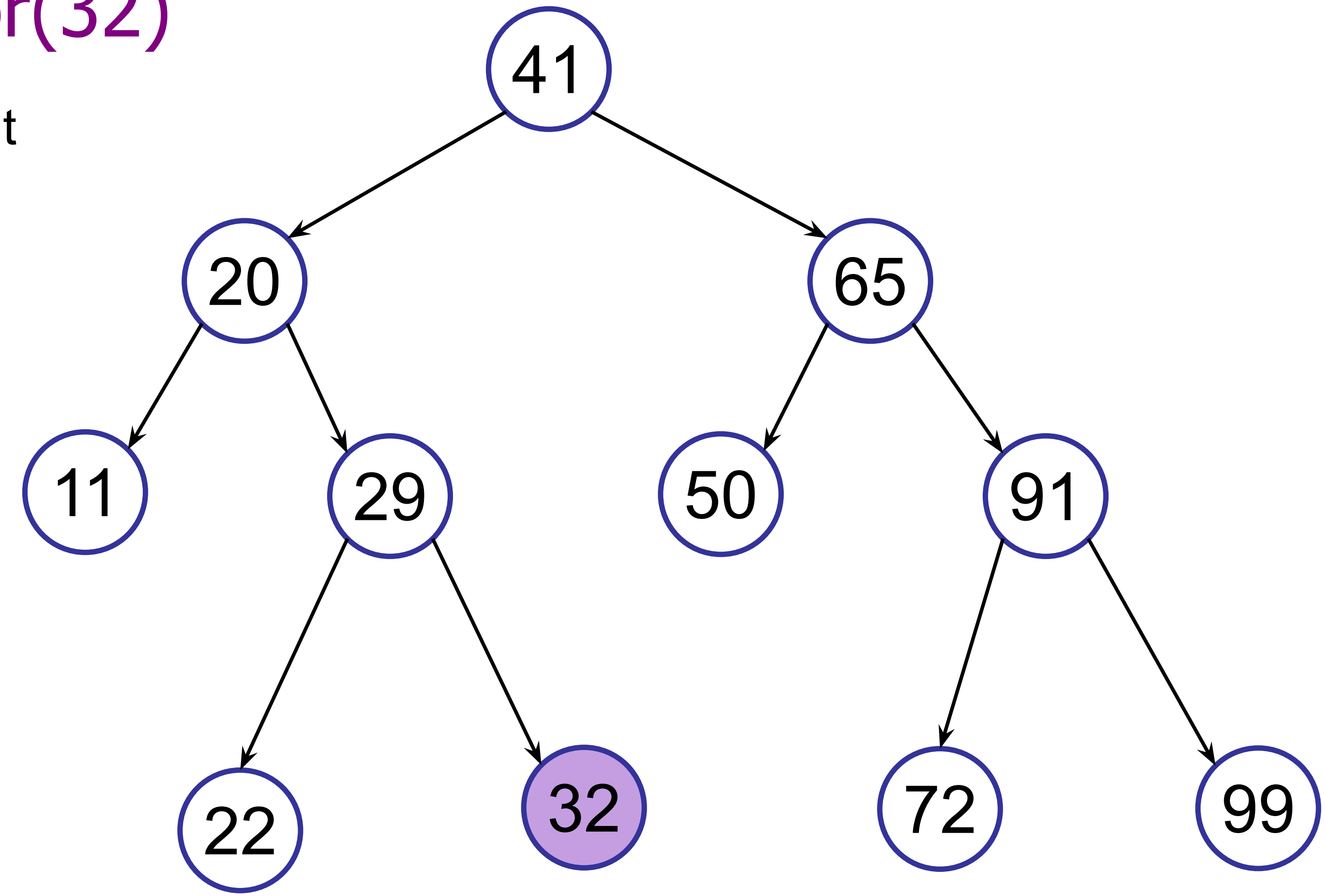
Case 2: node has no right child.

Successor Queries

successor(32)

32 is the right child of its parent

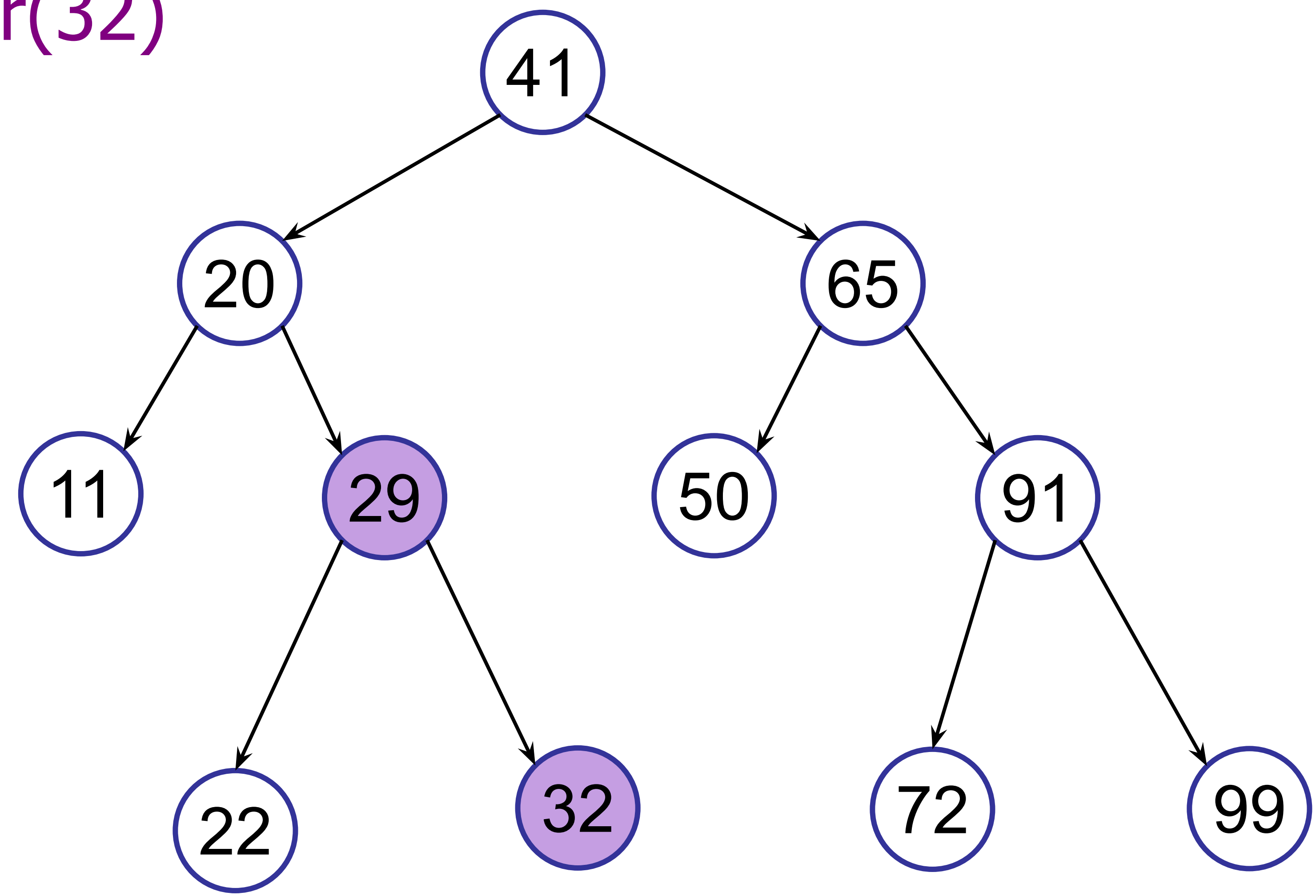
How now brown cow?



Case 2: node has no right child.

Successor Queries

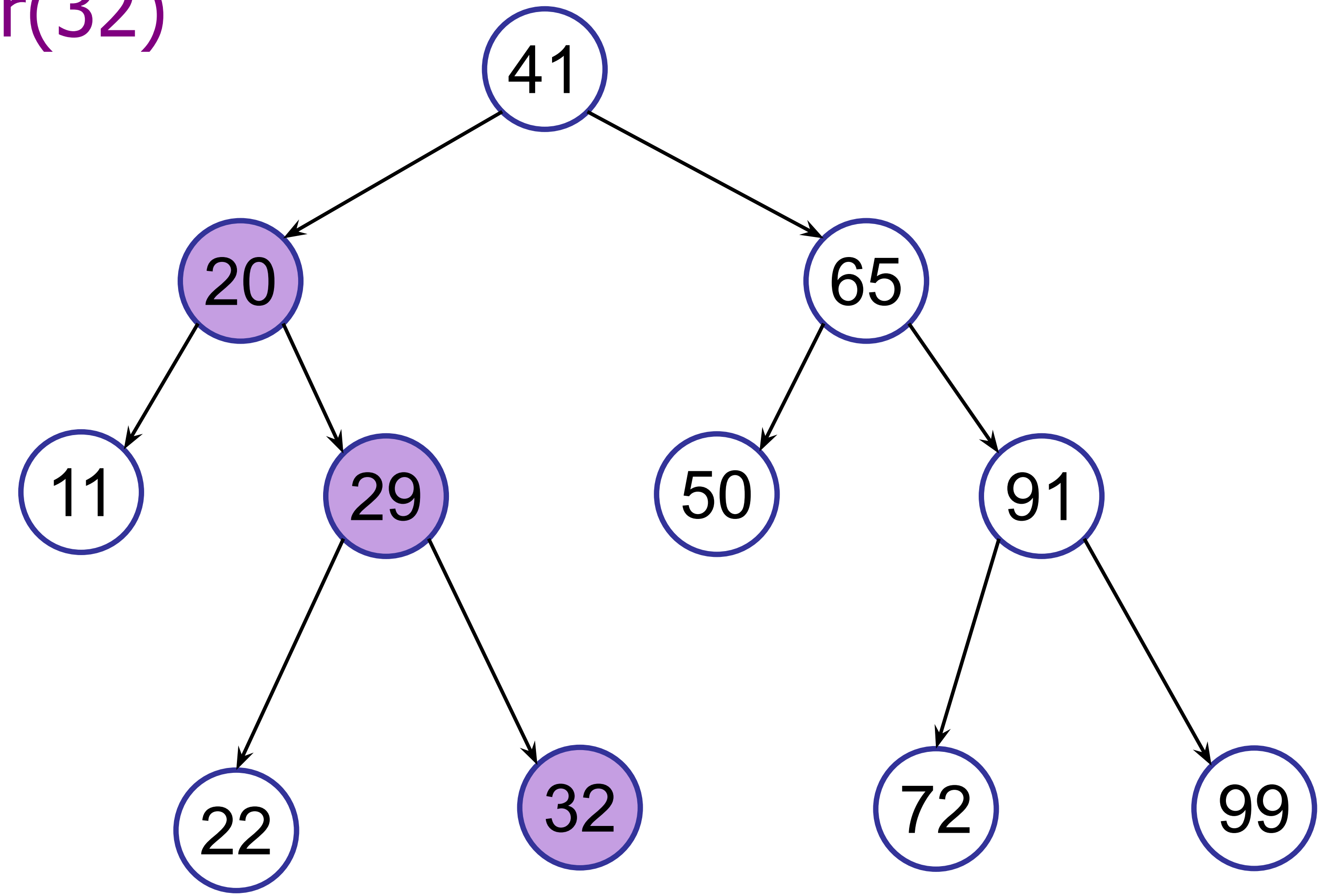
successor(32)



Case 2: node has no right child.

Successor Queries

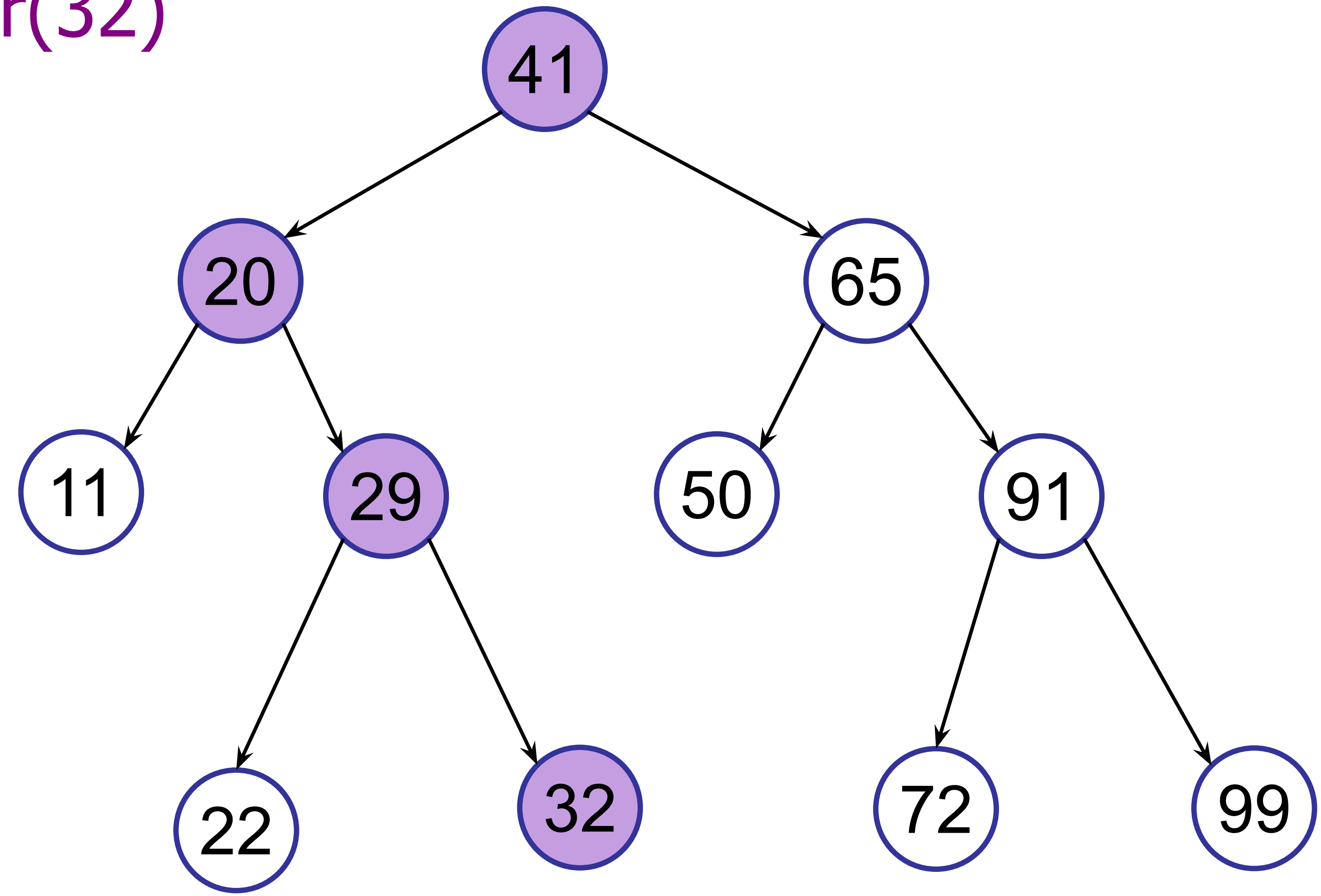
successor(32)



Case 2: node has no right child.

Successor Queries

successor(32)

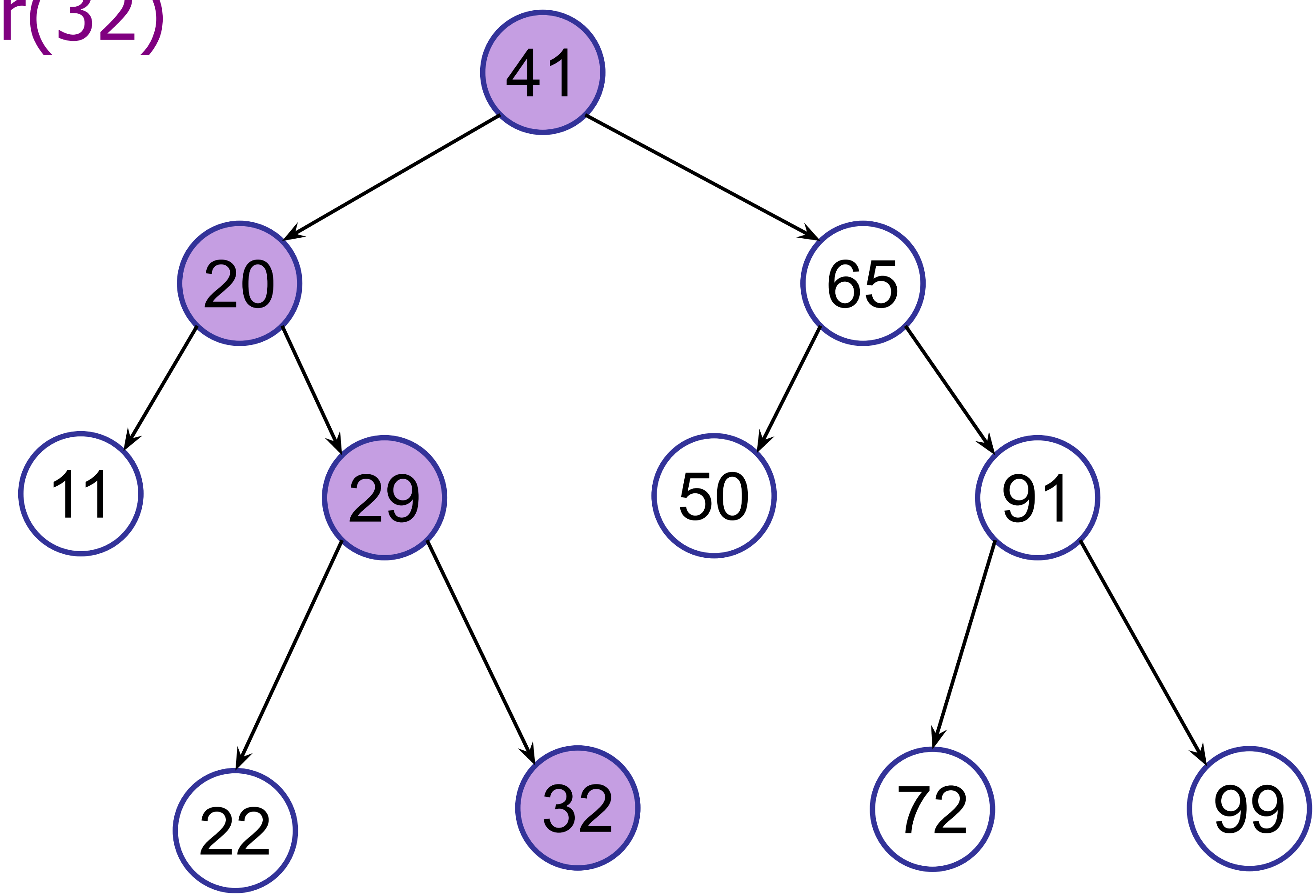


Case 2: node has no right child.

Successor Queries

successor(32)

successor of
41 is
successor of
32



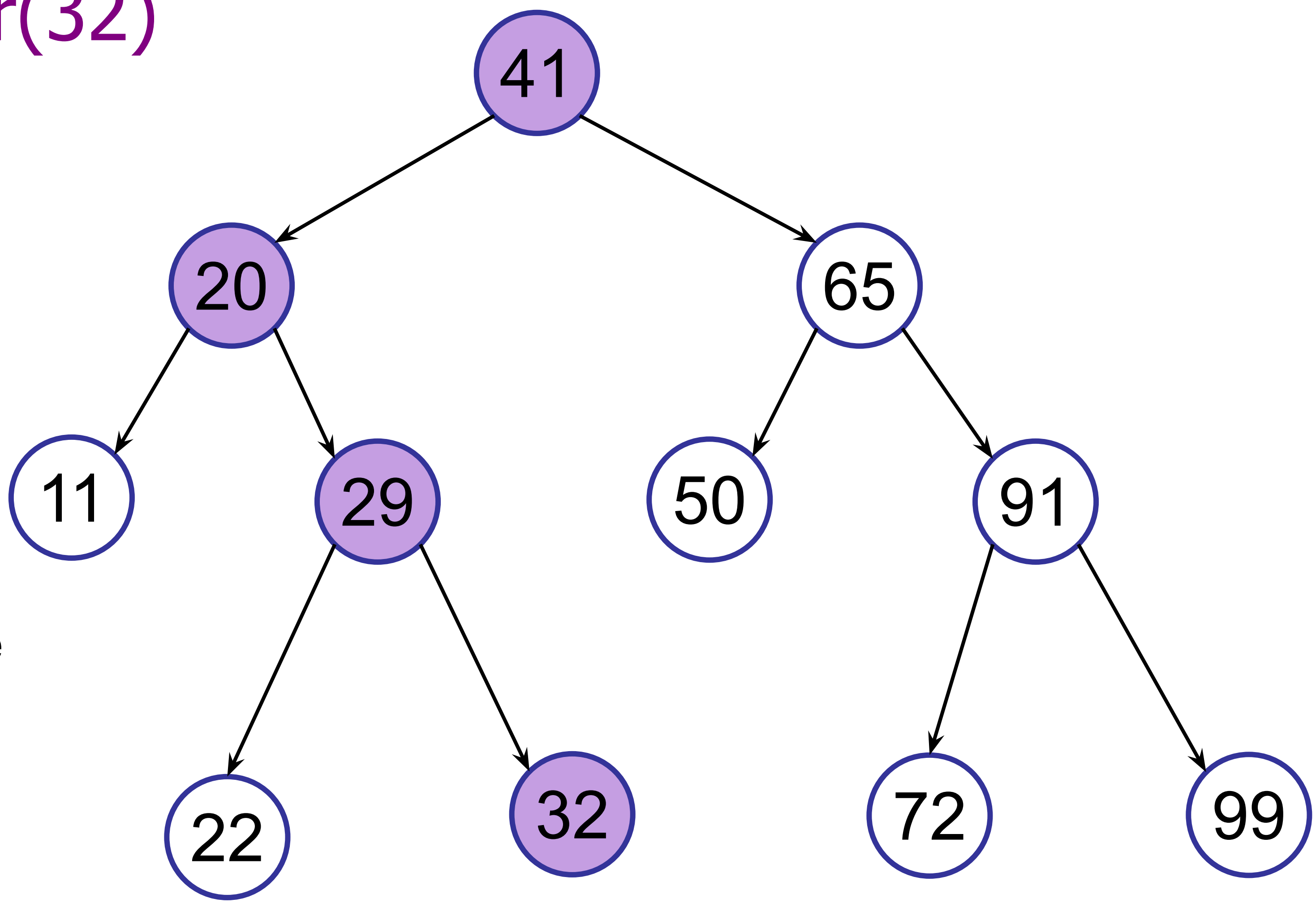
Case 2: node has no right child.

Successor Queries

successor(32)

successor of 41 is successor of 32

this was done recursively!



Case 2: node has no right child.

Successor Queries

Find the next TreeNode:

```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }
    return parent;
}
```

Successor Queries

Find the next TreeNode:

```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }
    return parent;
}
```

Successor Queries

Find the next TreeNode:

```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }
    return parent;
}
```

Successor Queries

Find the next TreeNode:

```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }
    return parent;
}
```

root.parent == null

Binary Search Trees

1. Terminology and Definitions

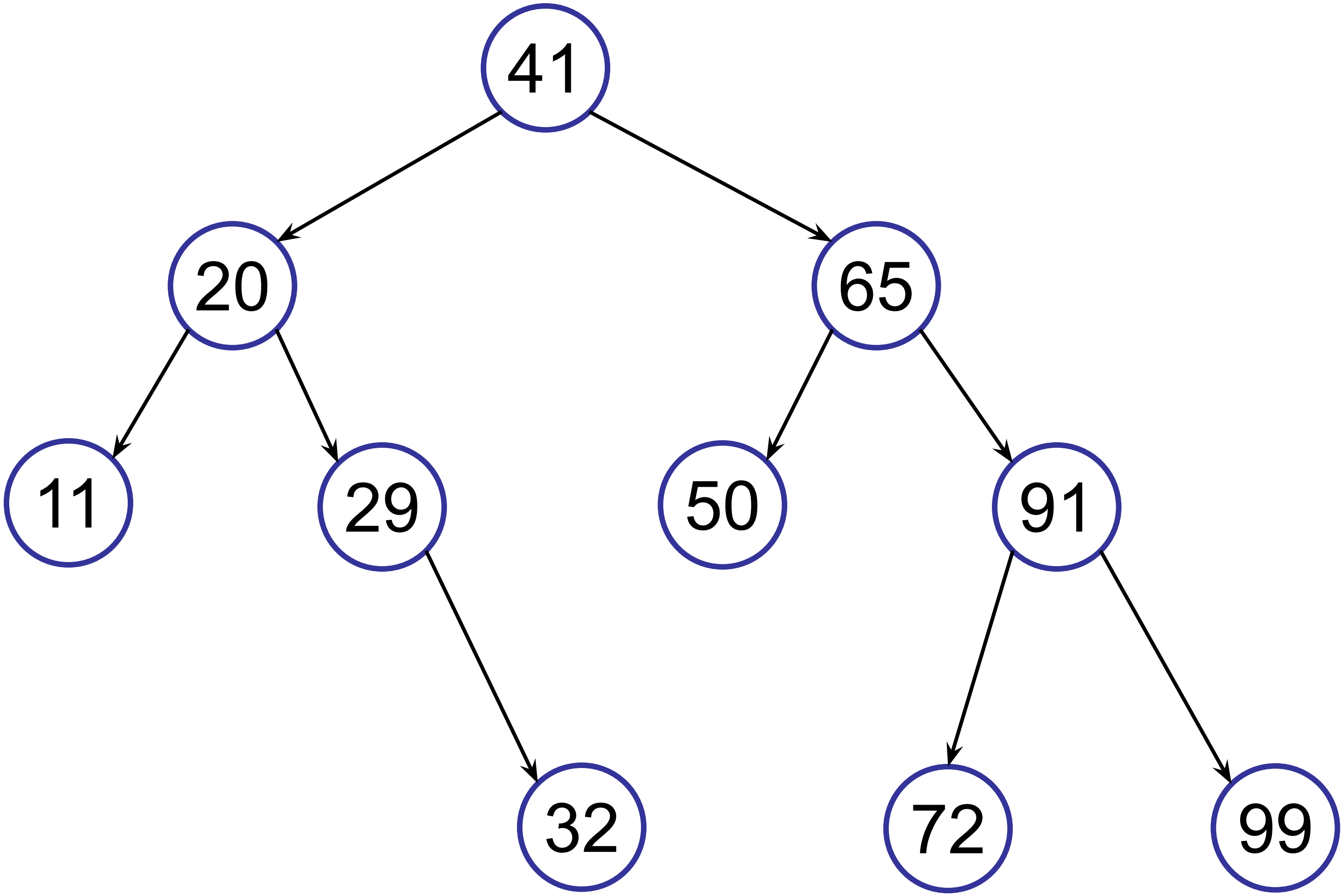
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

3. Traversals
   - in-order, pre-order, post-order

4. Other operations

Binary Search Tree

delete(v)

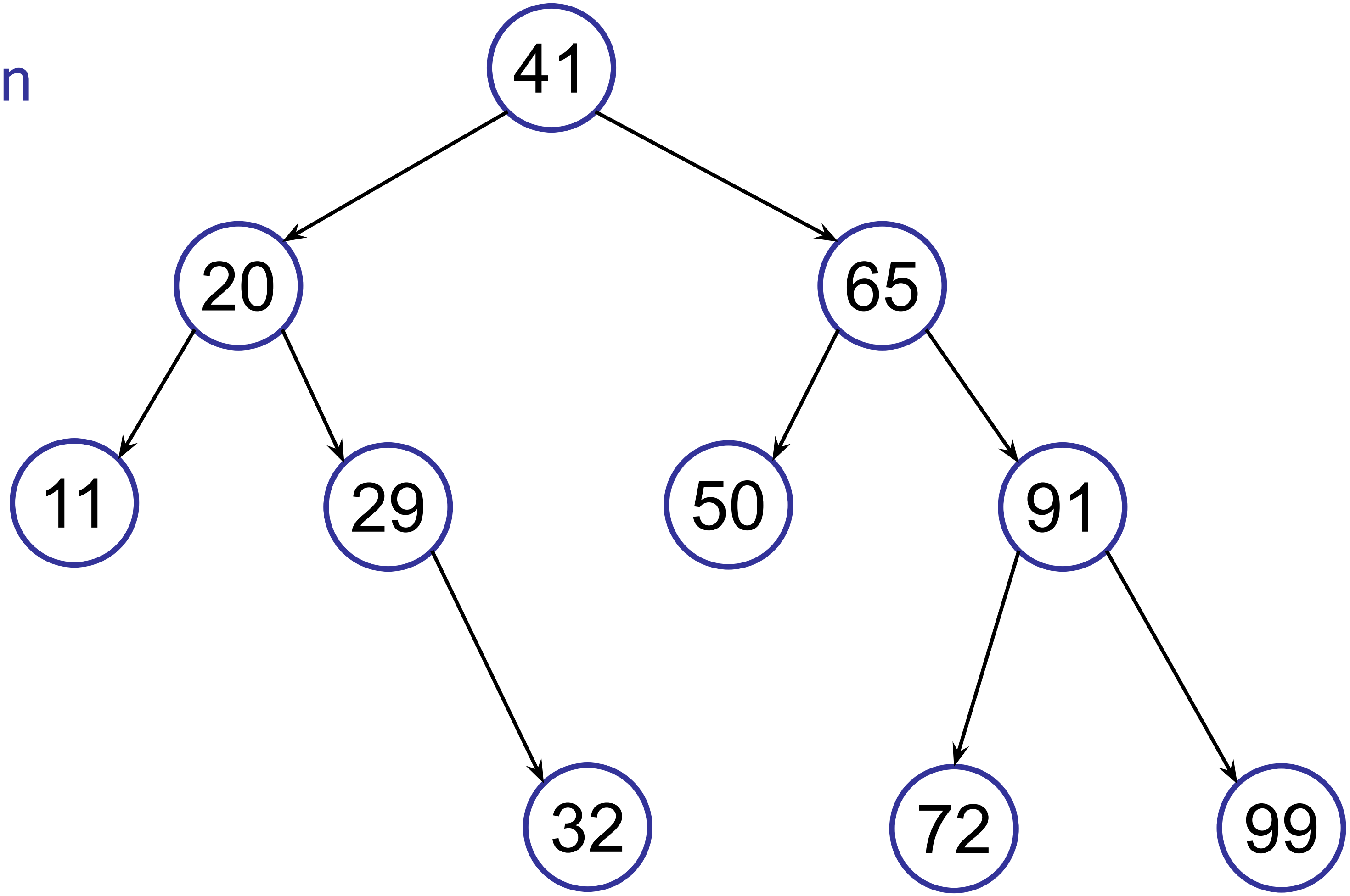


Binary Search Tree

delete(v)

Three cases:

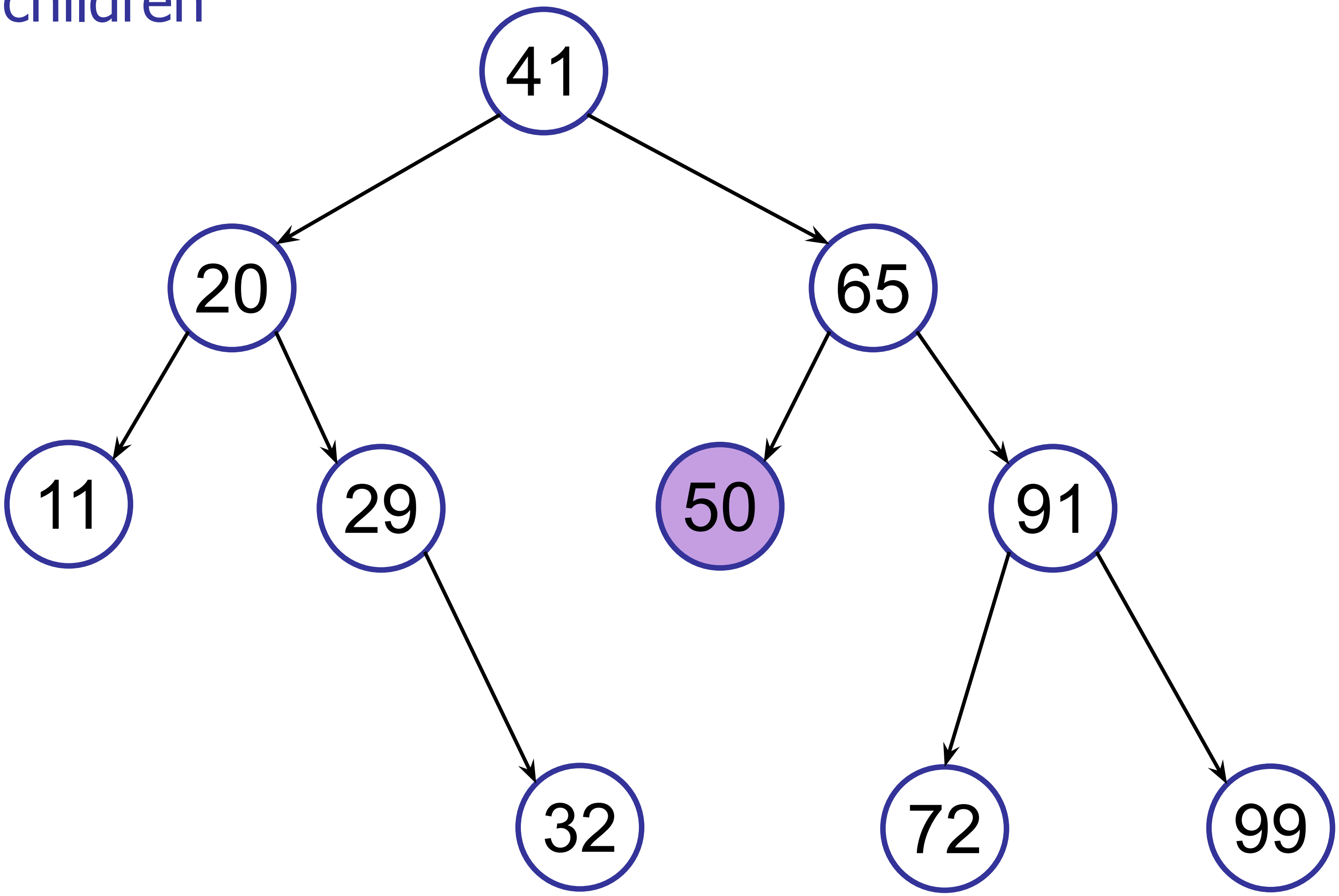
1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(50)

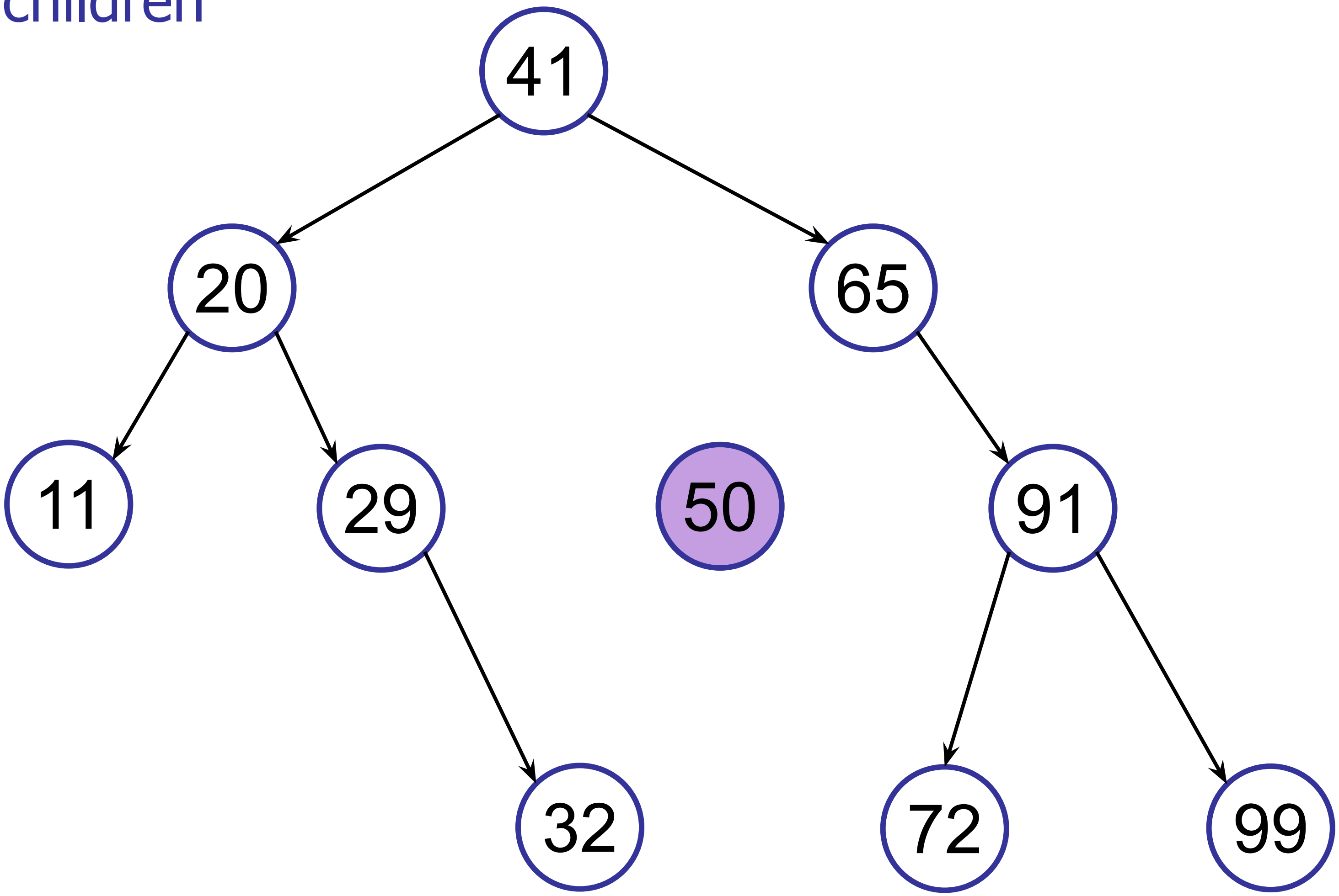
Case 1: No children



Binary Search Tree

delete(50)

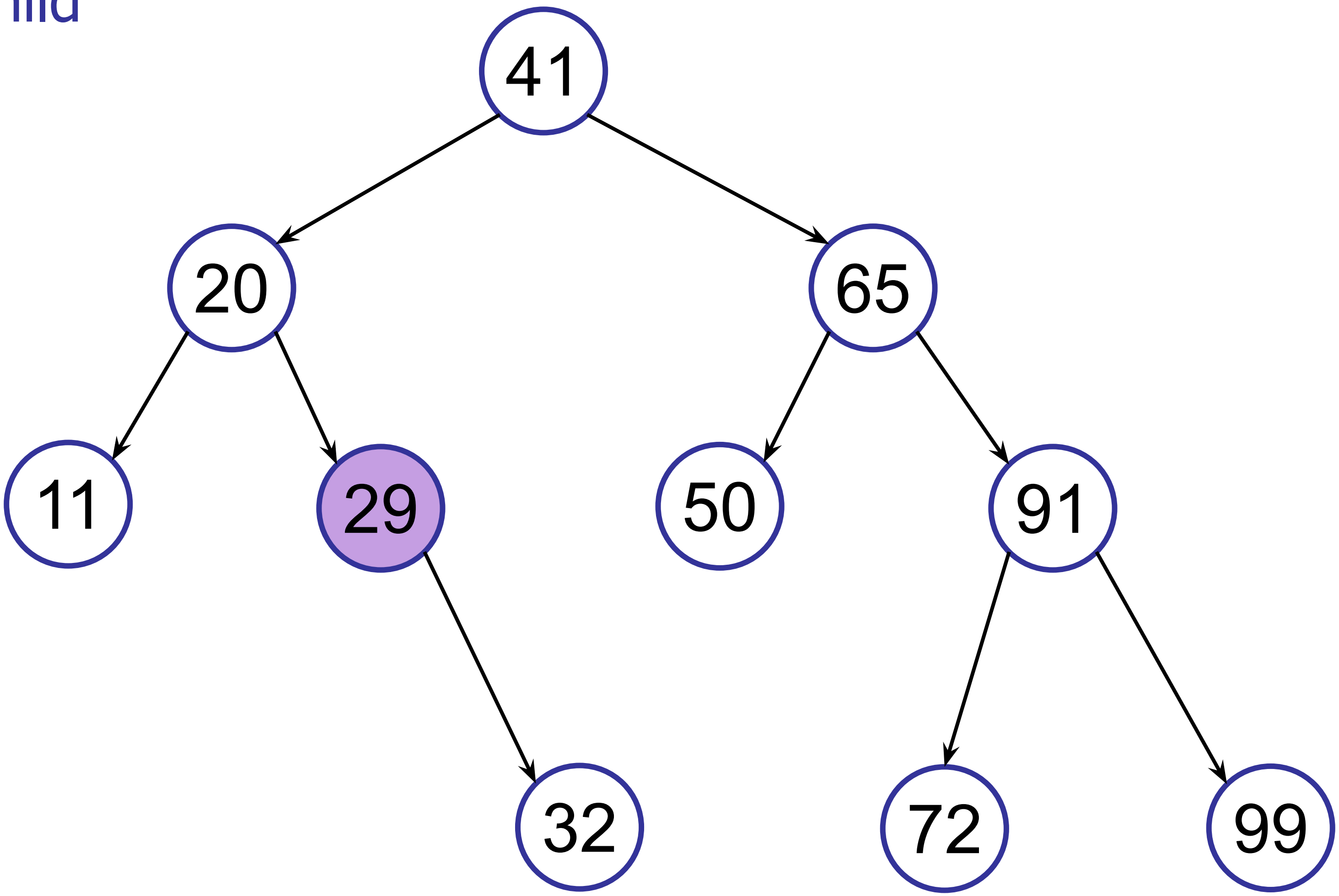
Case 1: No children



Binary Search Tree

delete(29)

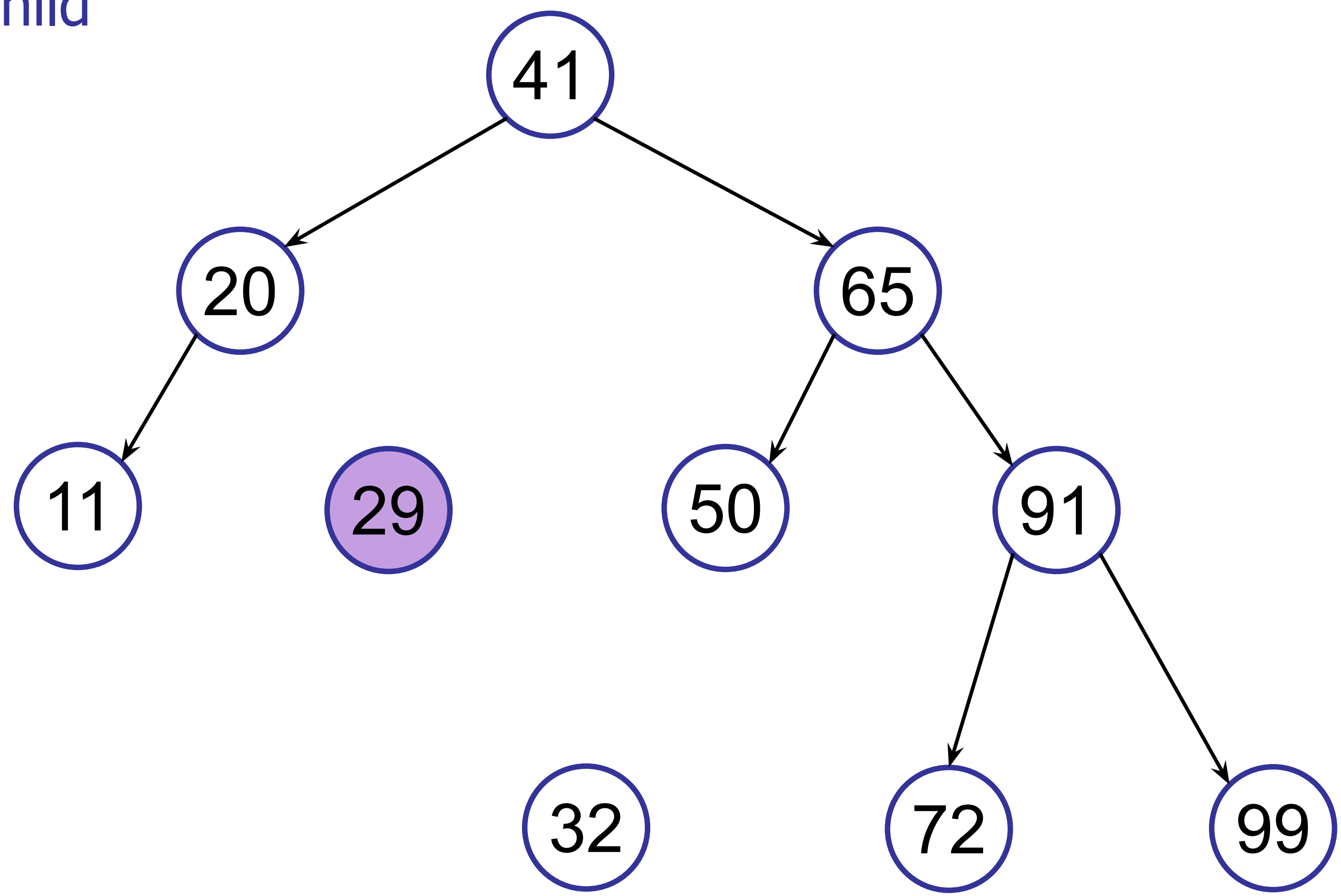
Case 2: 1 child



Binary Search Tree

delete(29)

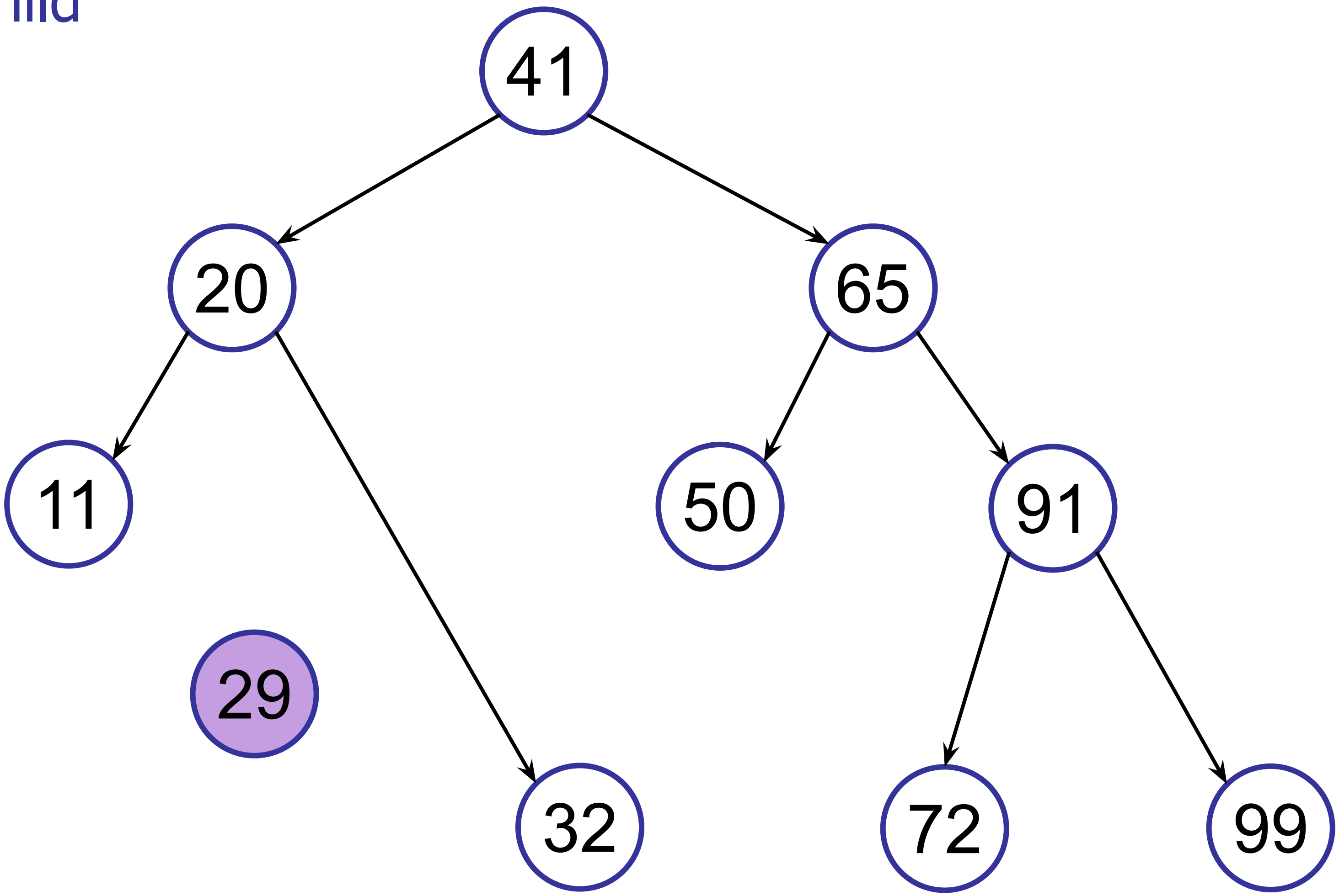
Case 2: 1 child



Binary Search Tree

delete(29)

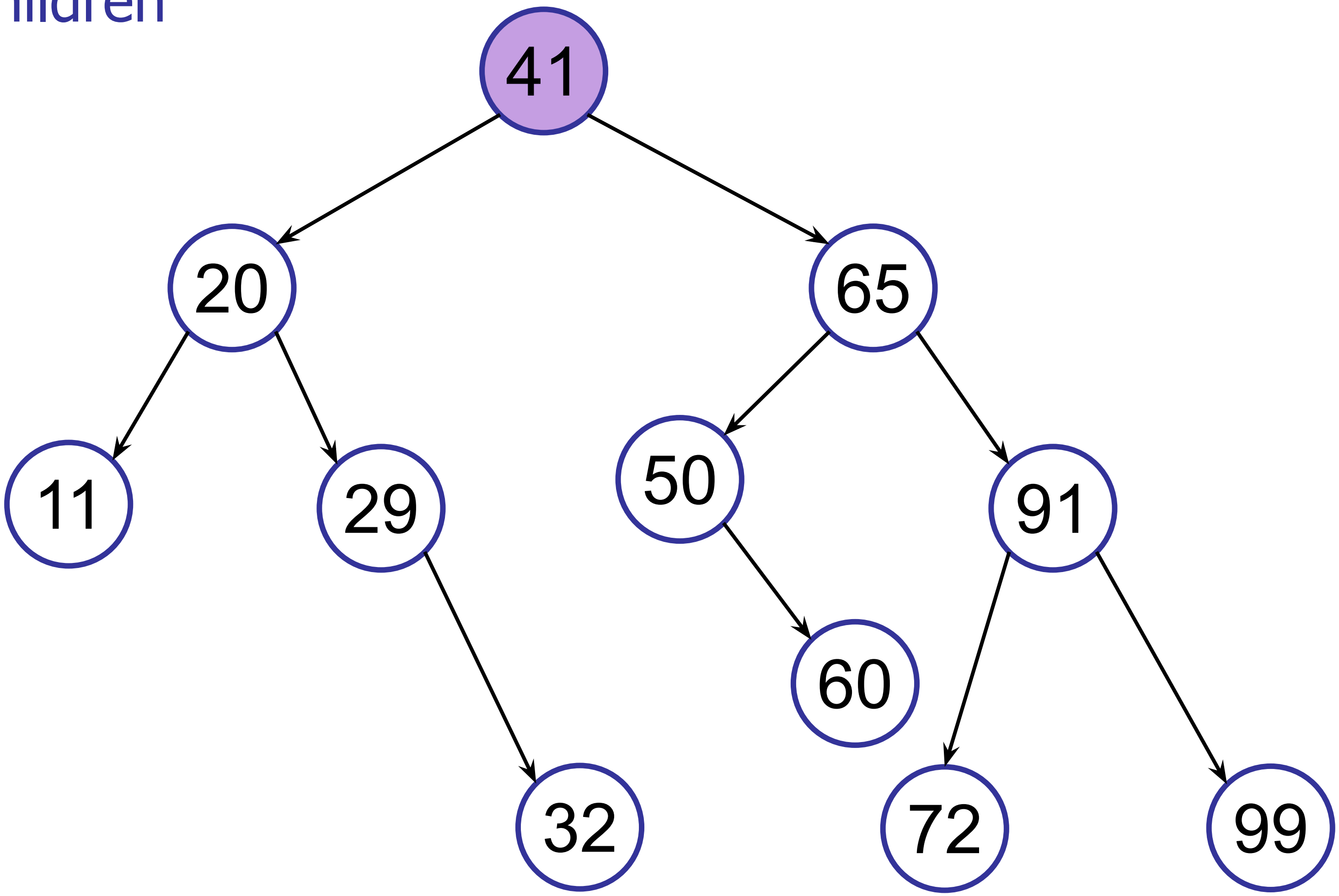
Case 2: 1 child



Binary Search Tree

delete(41)

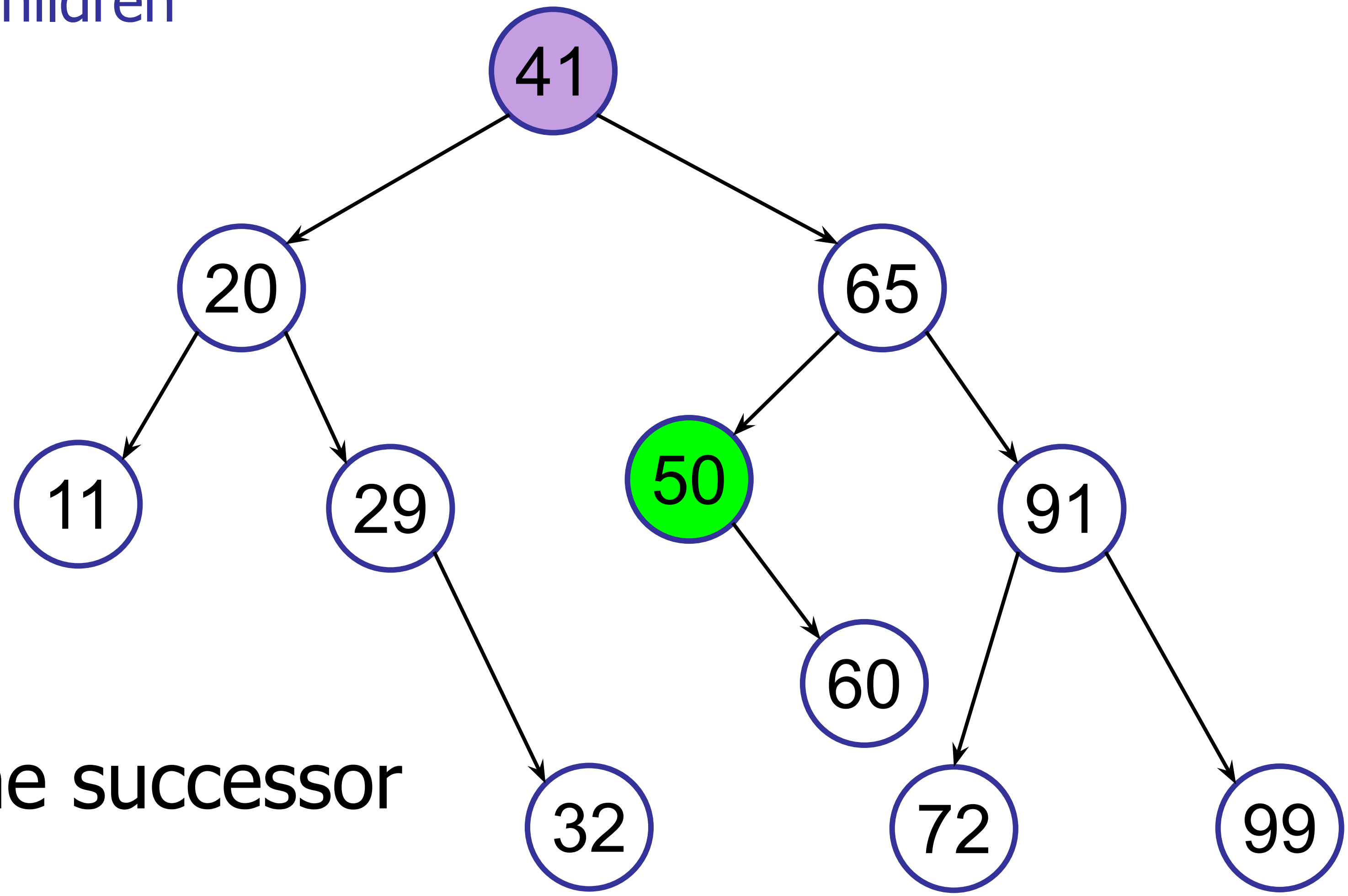
Case 3: 2 children



Binary Search Tree

delete(41)

Case 3: 2 children



50 is the successor

Binary Search Tree

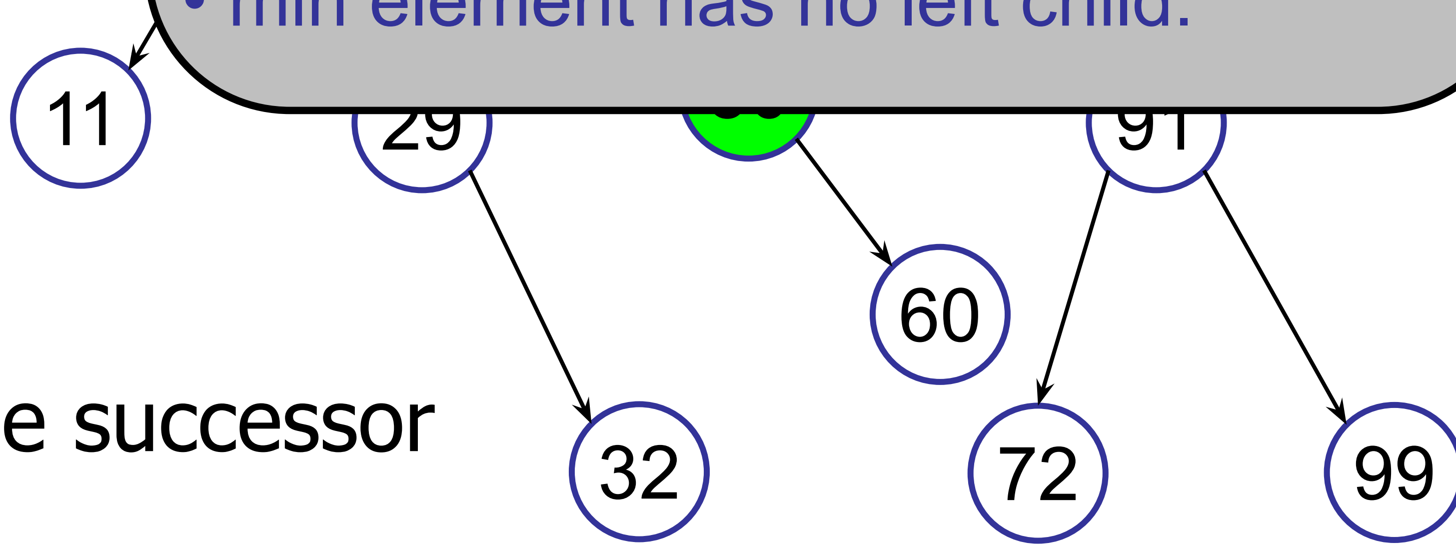
delete(41)

Case 3: 2 childre

Claim: successor of deleted node has at most 1 child!

Proof:

- Deleted node has two children.
- Deleted node has a **right** child.
- successor() = right.findMin()
- min element has no left child.

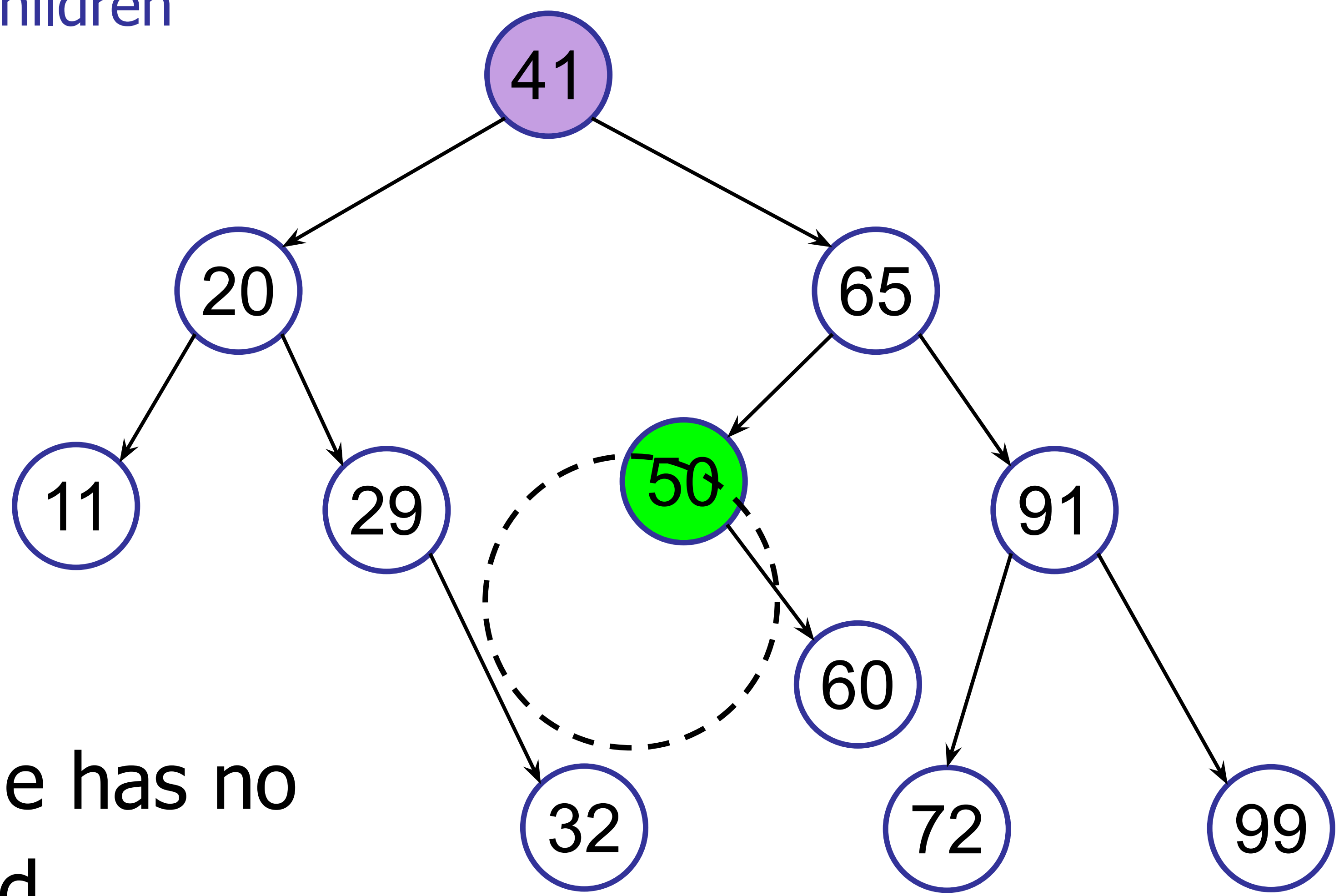


50 is the successor

Binary Search Tree

delete(41)

Case 3: 2 children

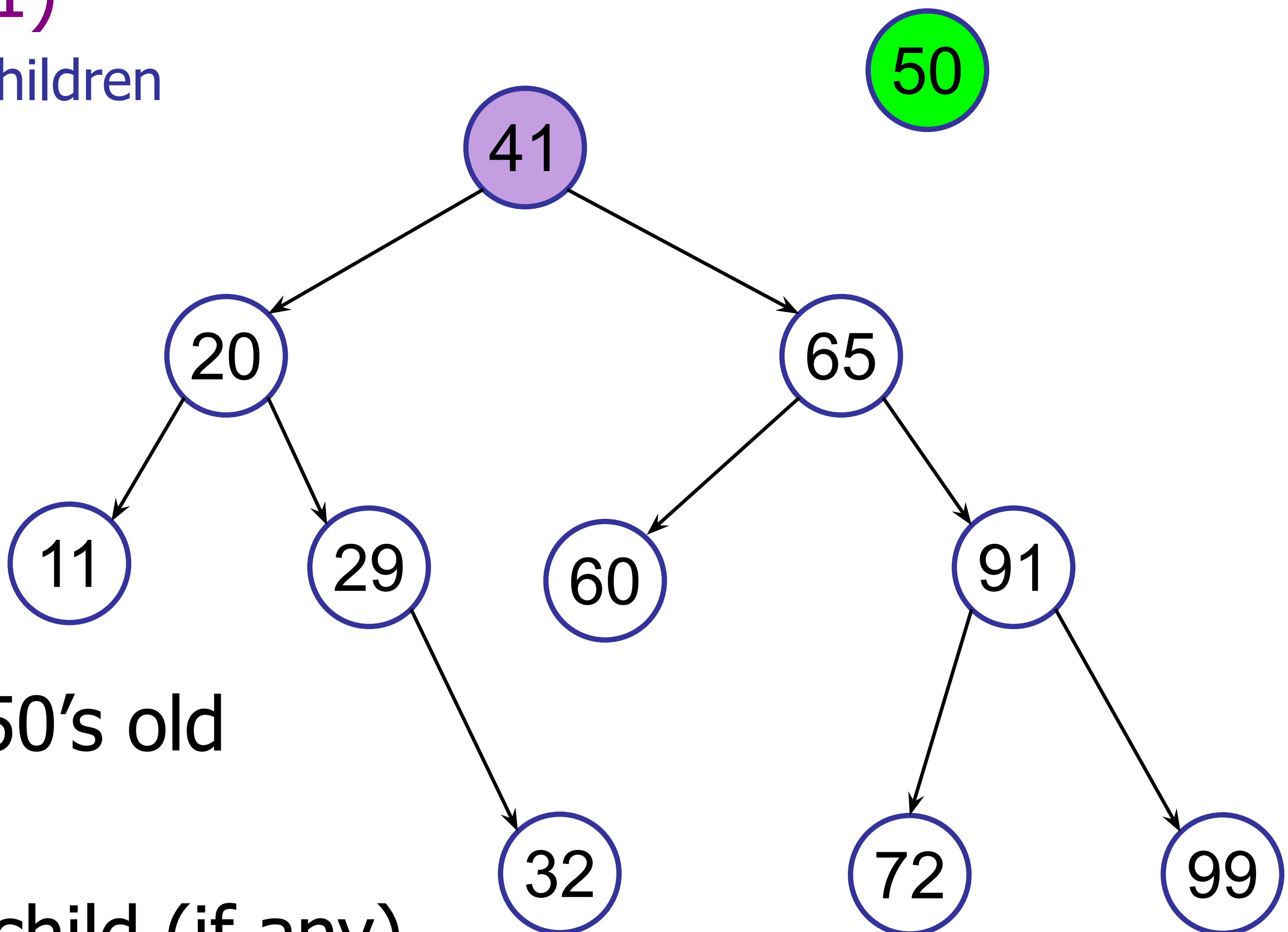


50 is the has no left child

Binary Search Tree

delete(41)

Case 3: 2 children

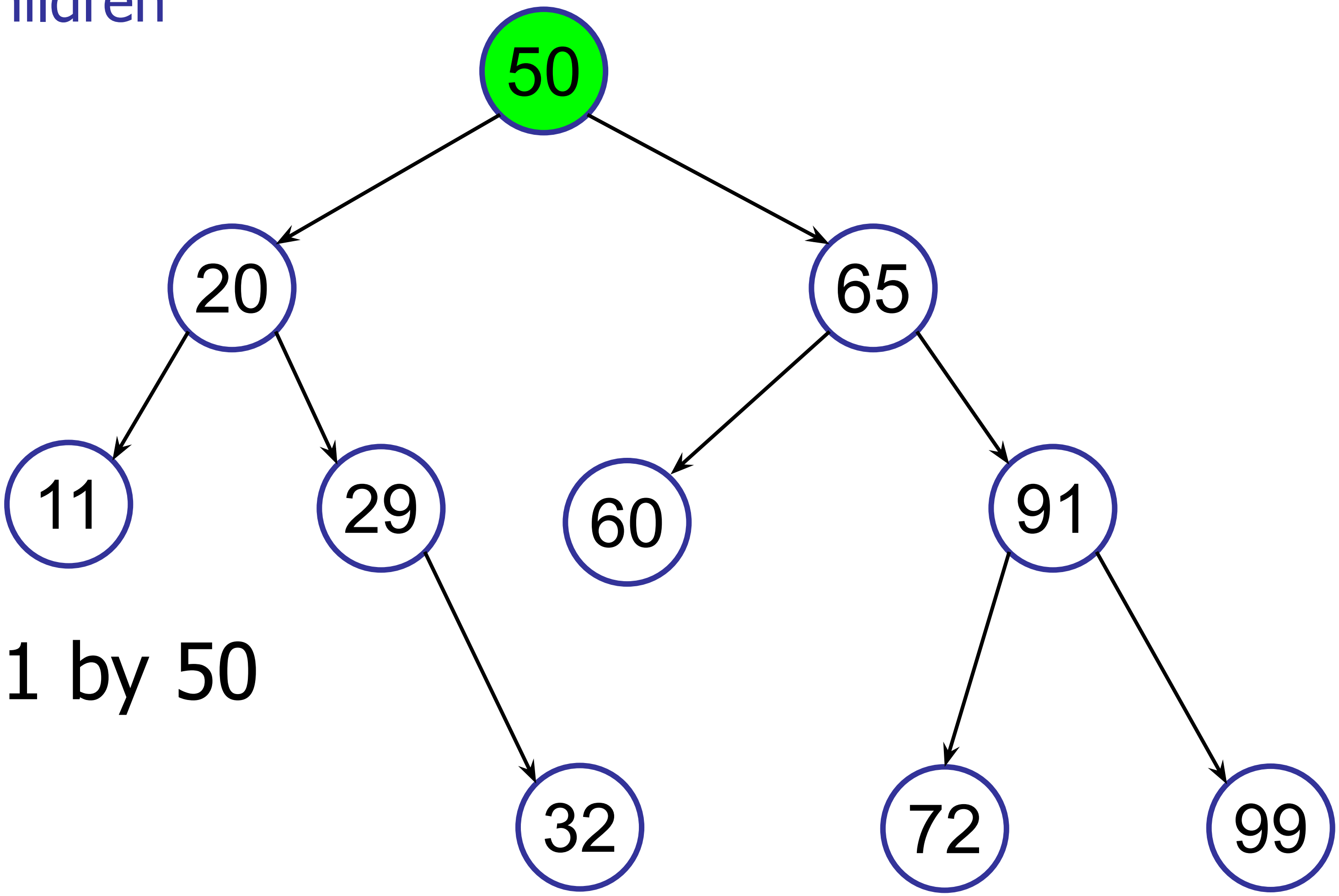


Connect 50's old parent to the right child (if any).

Binary Search Tree

delete(41)

Case 3: 2 children

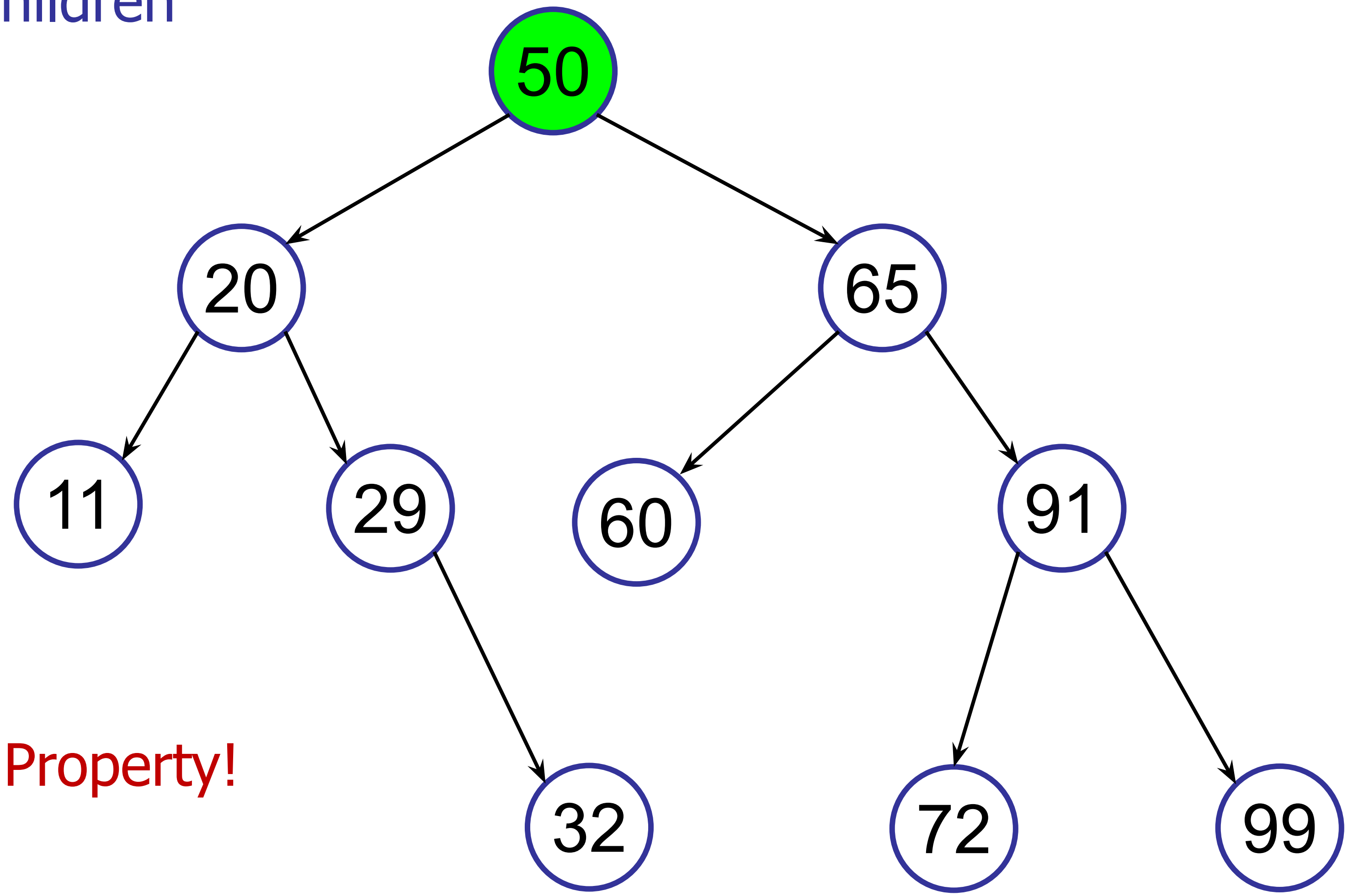


Replace 41 by 50

Binary Search Tree

delete(41)

Case 3: 2 children

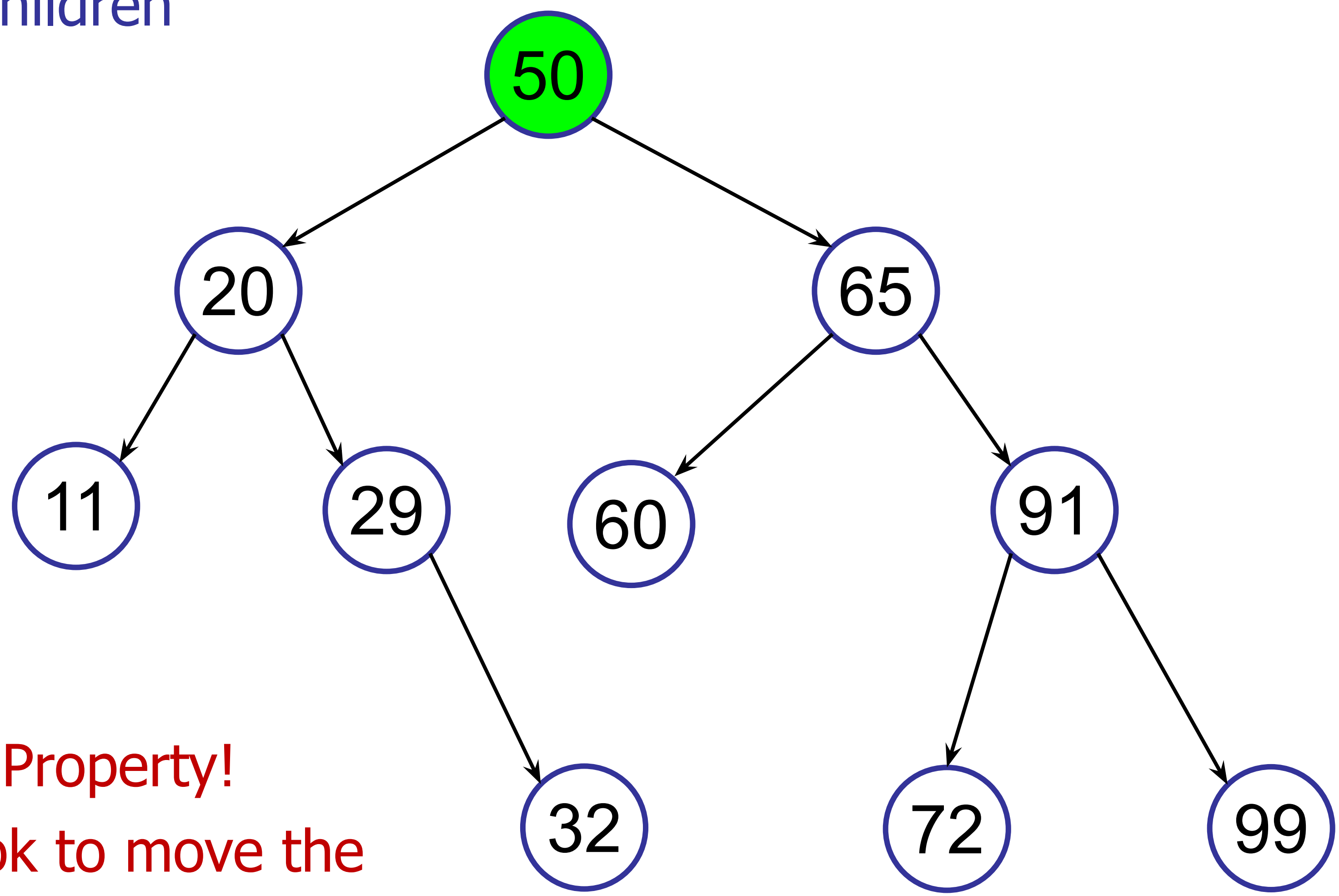


Check BST Property!

Binary Search Tree

delete(41)

Case 3: 2 children



Check BST Property!

Why was it ok to move the right subtree of 50 up?

Binary Search Tree

delete(v) Running time: O(height)

Three cases:

1. No children:
   - remove v
2. 1 child:
   - remove v
   - connect child(v) to parent(v)
3. 2 children
   - x = successor(v)
   - delete(x)
   - remove v
   - connect x to left(v), right(v), parent(v)

Binary Search Tree

Modifying Operations

- insert: O(h)
- delete: O(h)

Query Operations:

- search: O(h)
- predecessor, successor: O(h)
- findMax, findMin: O(h)
- in-order-traversal: O(n)

Plan of the Day

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations

Part 2

On the importance of being balanced

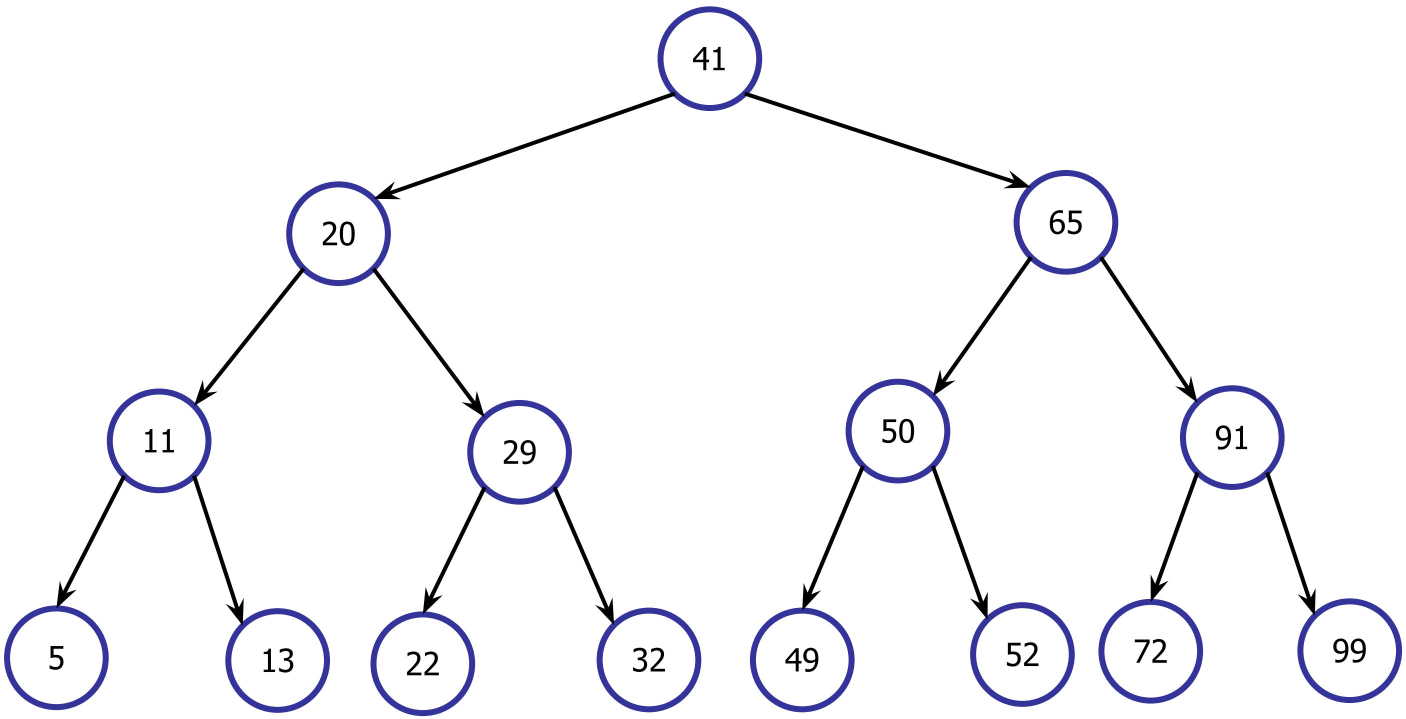
Part 2

On the importance of being balanced

- Height-balanced binary search trees
- AVL trees
- Rotations

The Importance of Being Balanced

Operations take O(height) time



What is the largest possible height h?

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(\mathrm{sqrt}(n))$
4. $\Theta(n)$
5. $\Theta(n^2)$

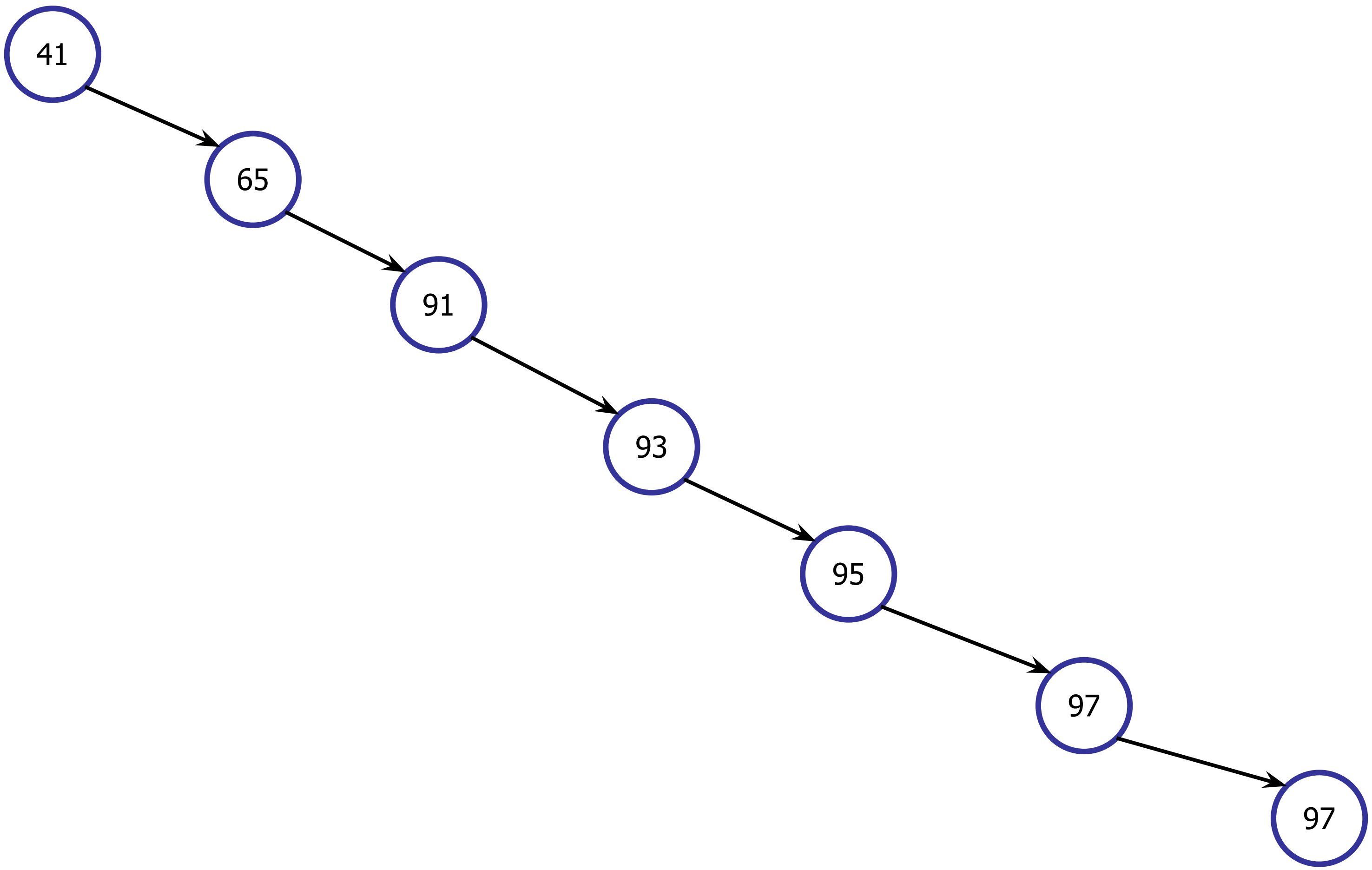
What is the largest possible height h?

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(\mathrm{sqrt}(n))$
4. $\Theta(n)$
5. $\Theta(n^2)$

The Importance of Being Balanced

Operations take O(h) time

$h \leq n$



What is the smallest possible height h?

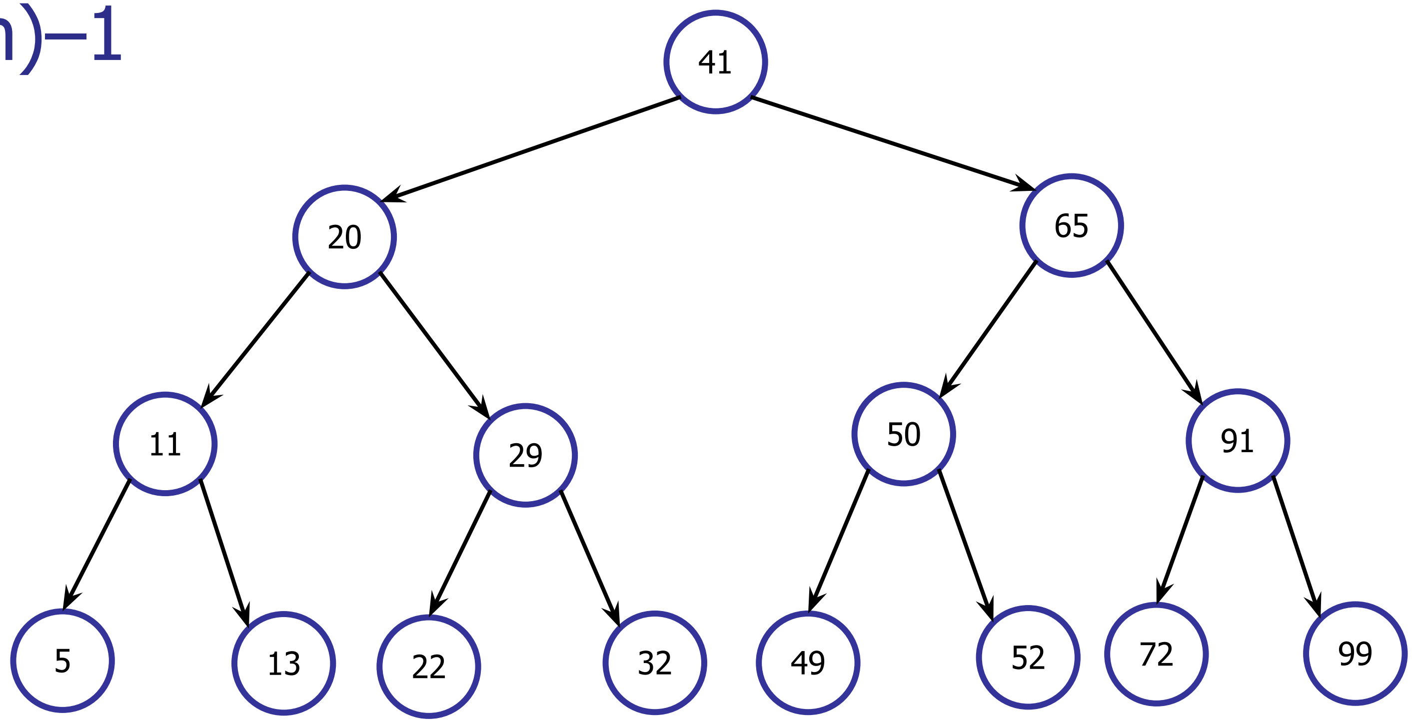
1. $\Theta(1)$
2. $\Theta(\log\log n)$
3. $\Theta(\log n)$
4. $\Theta(\mathrm{sqrt}(n))$
5. $\Theta(n)$
6. $\Theta(n^2)$

What is the smallest possible height h?

1. $\Theta(1)$
2. $\Theta(\log\log n)$
3. $\Theta(\log n)$
4. $\Theta(\text{sqrt}(n))$
5. $\Theta(n)$
6. $\Theta(n^2)$

The Importance of Being Balanced

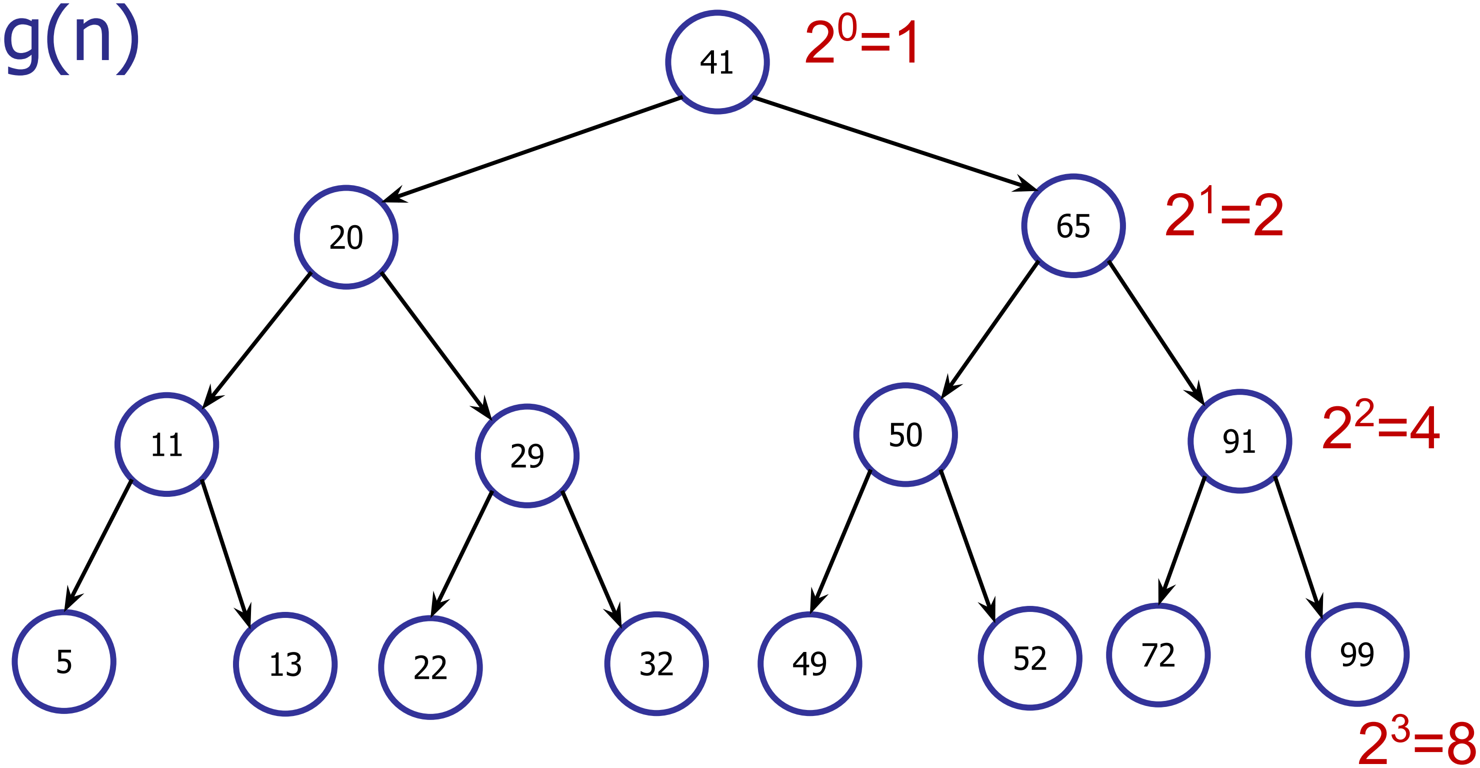
- Operations take O(h) time
  - $h \geq \log(n) - 1$



The Importance of Being Balanced

Operations take O(h) time

$h+1 \geq \log(n)$



$n \leq 1 + 2 + 4 + \ldots + 2^h$

$\leq 2^0 + 2^1 + 2^2 + \ldots + 2^h < 2^{h+1}$

The Importance of Being Balanced

Operations take O(h) time

$\log(n) - 1 \le h \le n$

Key definition

A BST is <u>balanced</u> if $h = O(\log n)$

On a balanced BST: all operations run in $O(\log n)$ time.

The Importance of Being Balanced

Operations take O(h) time

$\log(n) - 1 \le h \le n$

Key definition

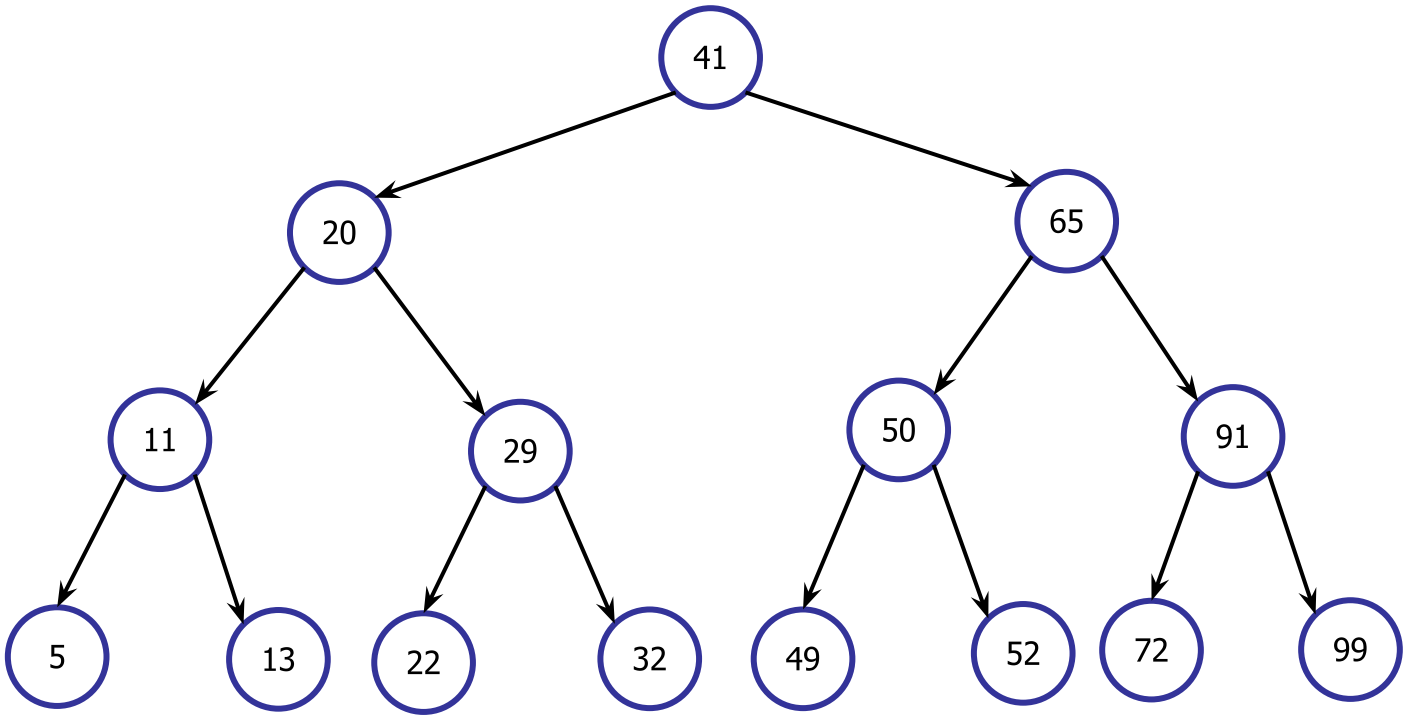
A BST is balanced if $h = O(\log n)$

On a balanced BST: all operations run in $O(\log n)$ time.

Side note: Items might be closer to the root, operations on those items might take less than $O(\log n)$ time.

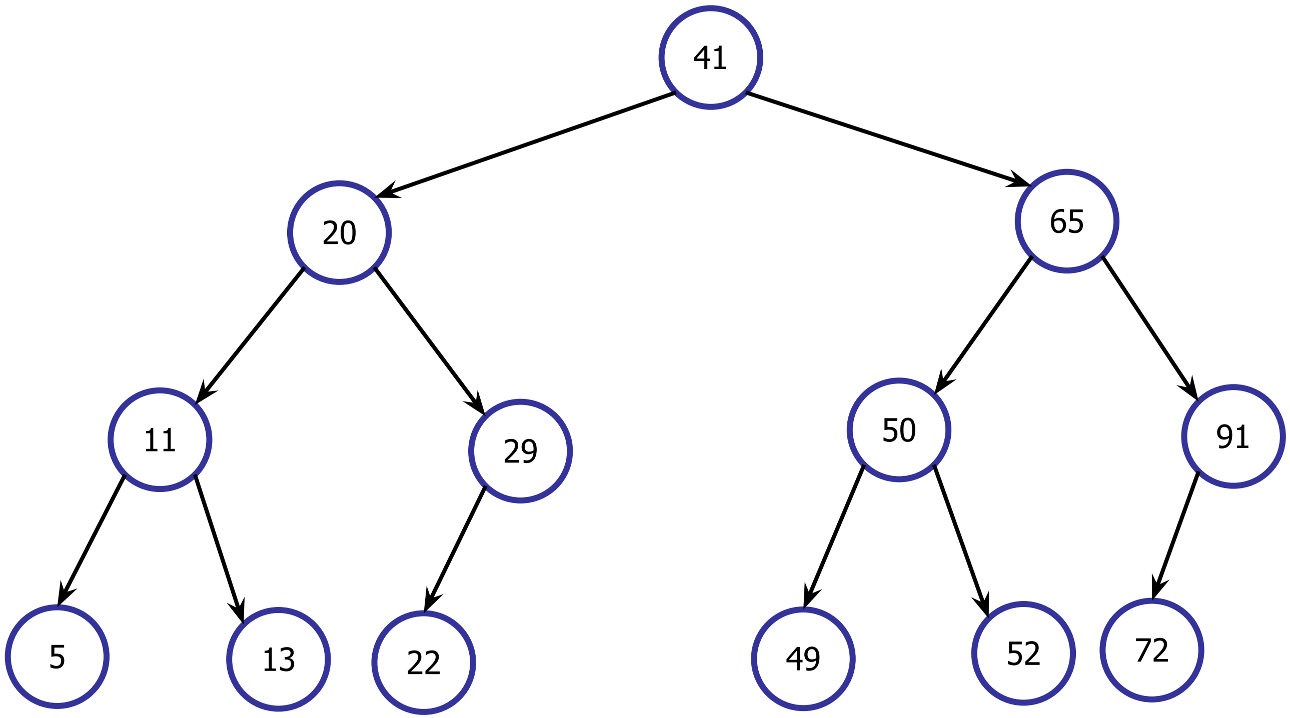
The Importance of Being Balanced

Perfectly balanced:



The Importance of Being Balanced

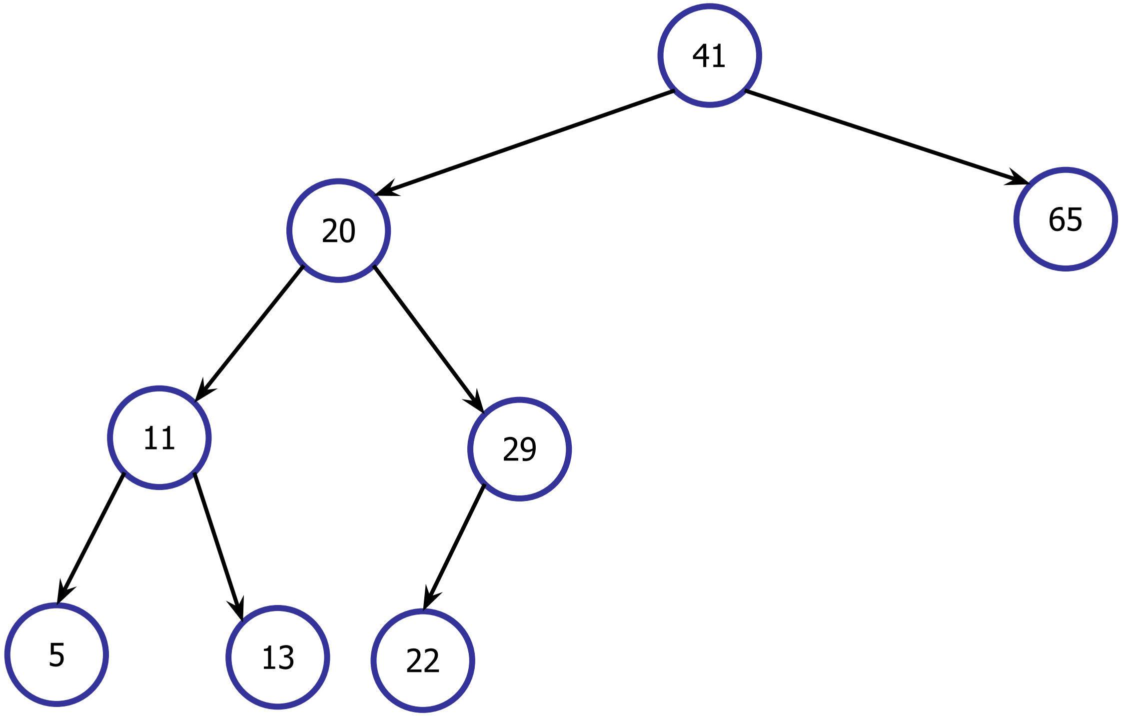
Almost perfectly balanced:



Every subtree has (almost) the same number of nodes.

The Importance of Being Balanced

Not perfectly balanced:



Left tree has 6, right tree has 1.

Balanced Search Trees

Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
- Scapegoat Trees (Anderson 1989)

Balanced Search Trees

Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
- Scapegoat Trees (Anderson 1989)

The Importance of Being Balanced

How to get a balanced tree:

- Define a good property of a tree.
- Show that if the good property holds, then the tree is balanced.
- After every insert/delete, make sure the good property still holds. If not, fix it.

Invariant

AVL Trees [Adelson-Velskii & Landis 1962]

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

Step 1: Define Height Balance

Step 2: Maintain Balance

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

- In every node v, store <u>height</u>:

  v.height = h(v)

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

– In every node v, store height:

v.height = h(v)

Why? Because then we don't have to recompute it when we need it.

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

- In every node v, store <u>height</u>:

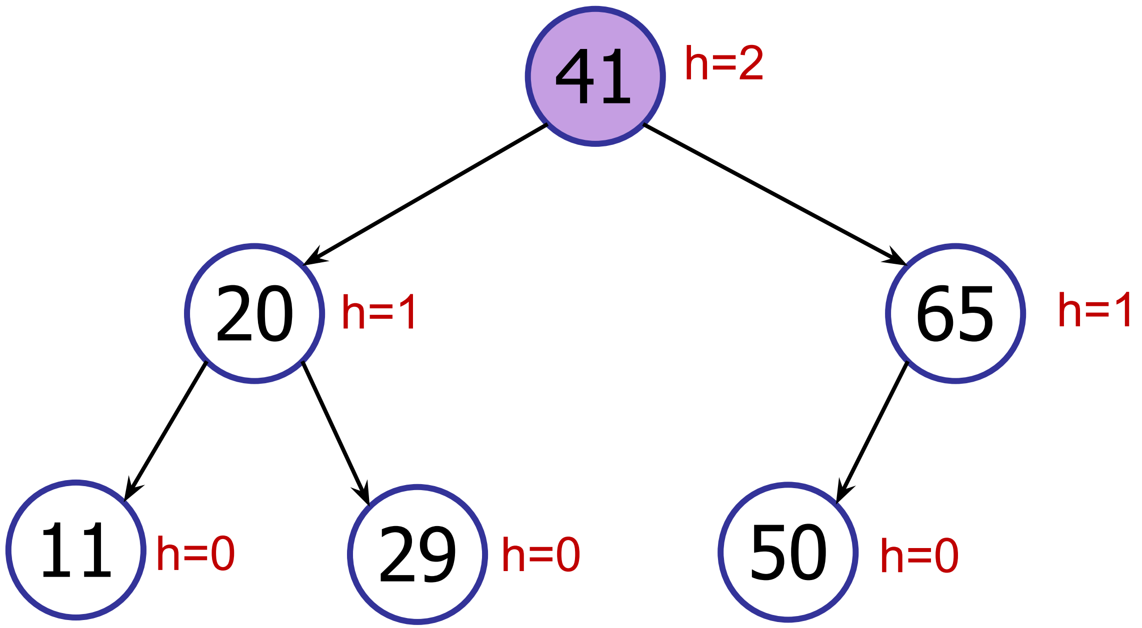
  v.height = h(v)

- On insert & delete operations, update <u>height</u>:

```
insert(x)
    if (x < key)
        left.insert(x)
    else right.insert(x)
    height = max(left.height, right.height) + 1
```

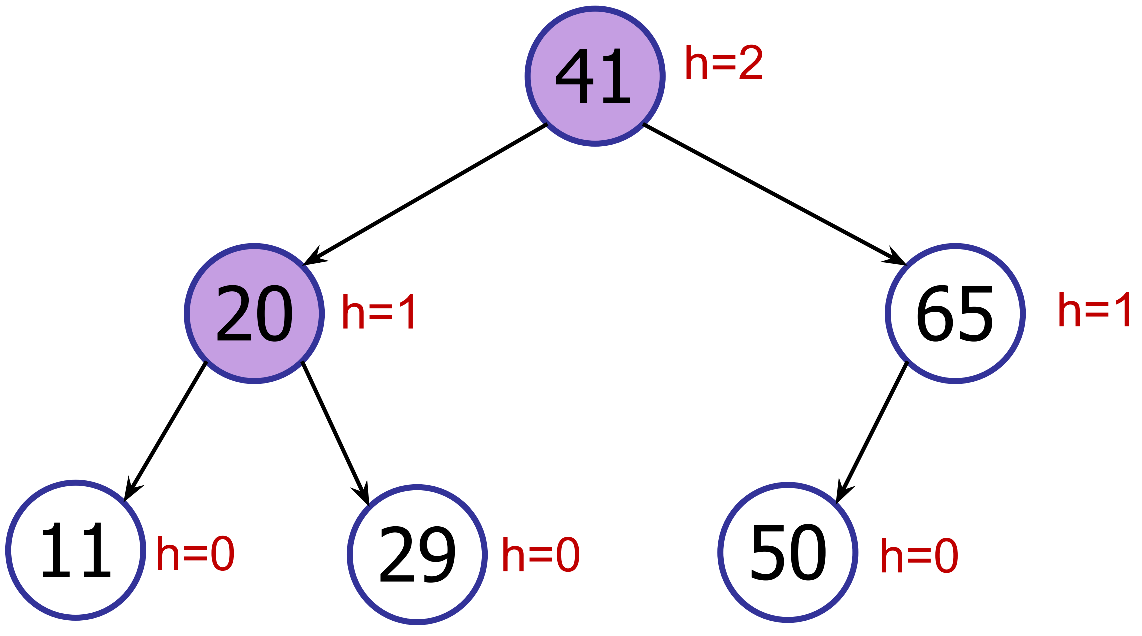
Binary Search Trees

insert(27)



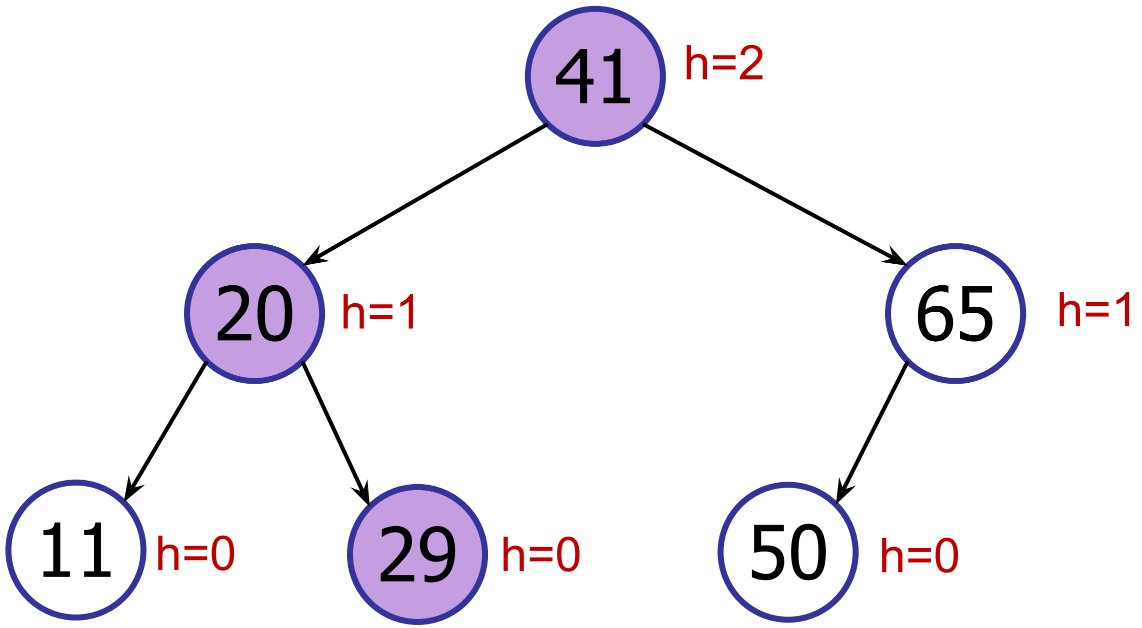
Binary Search Trees

insert(27)



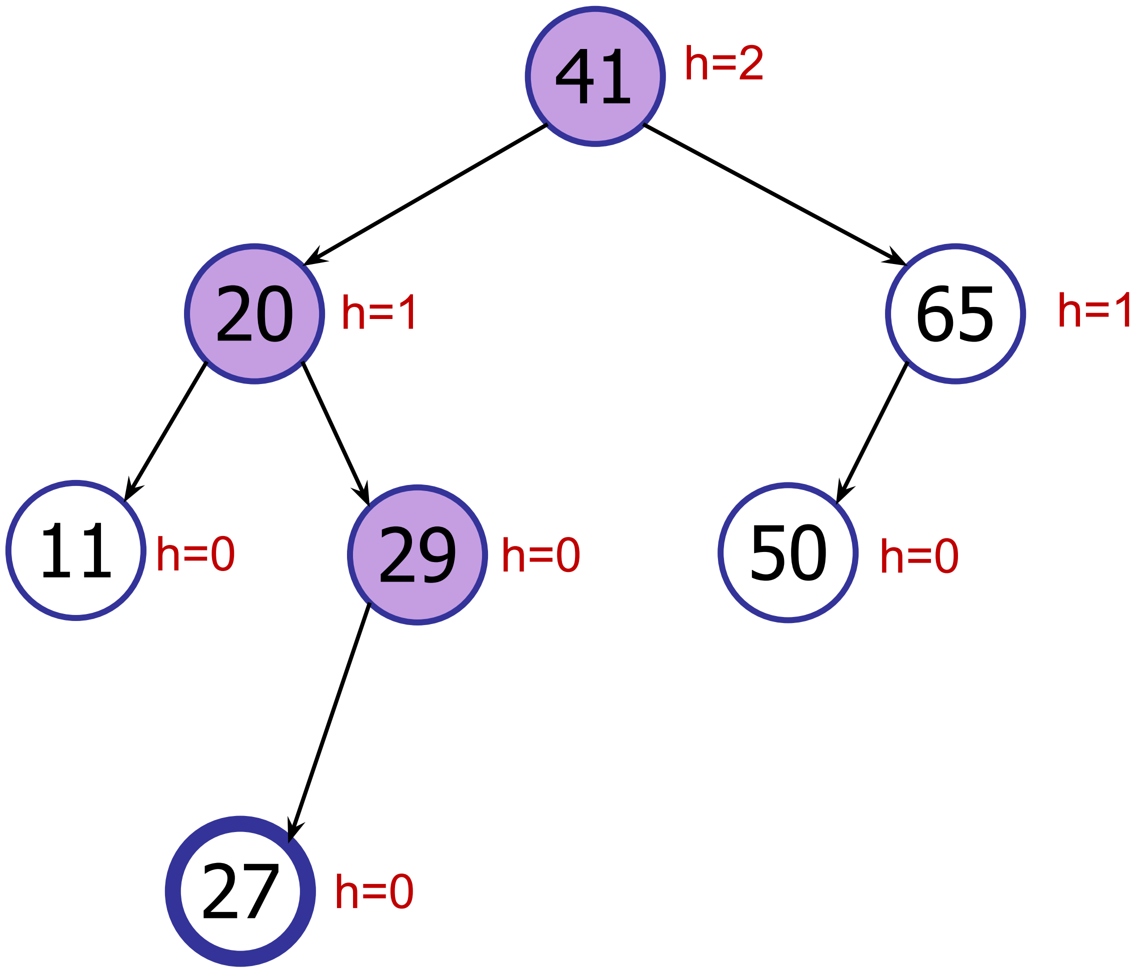
Binary Search Trees

insert(27)



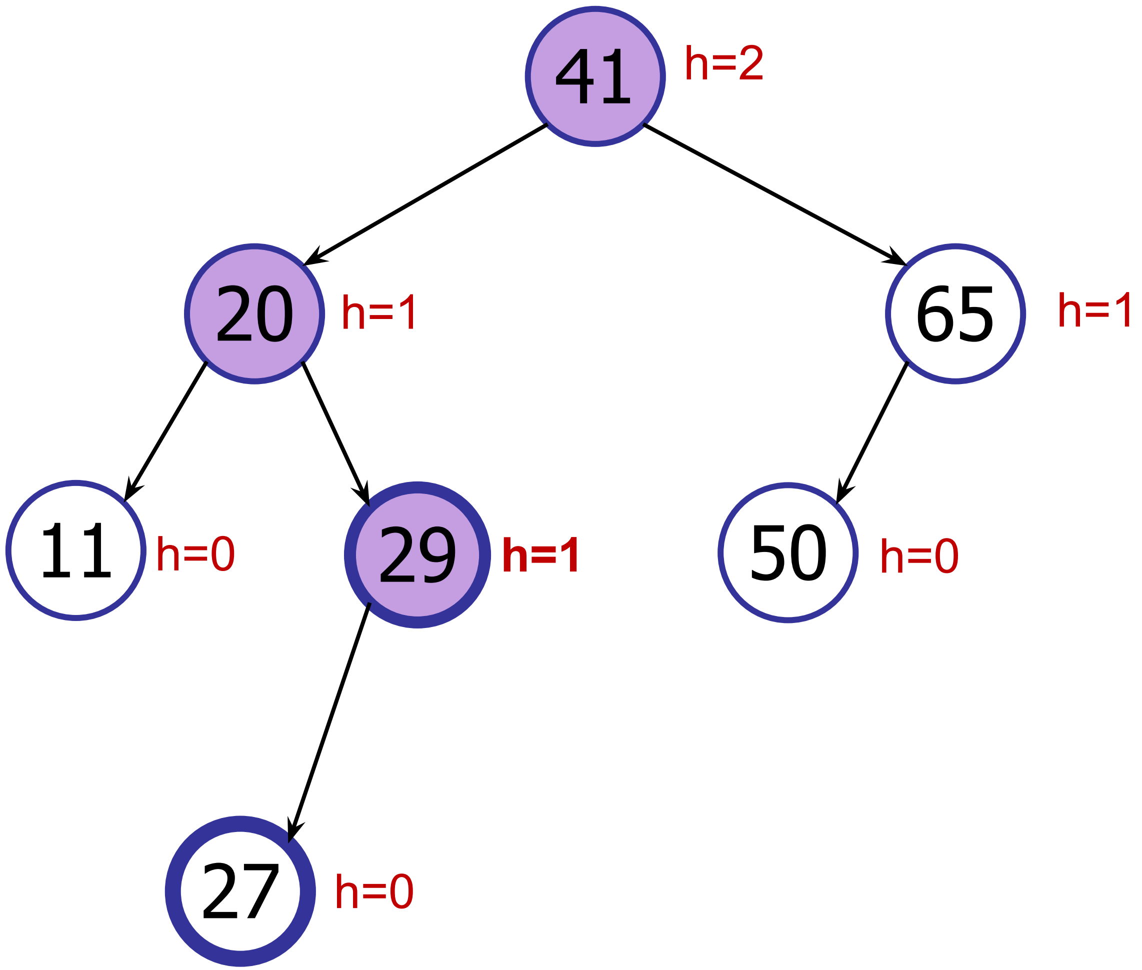
Binary Search Trees

insert(27)



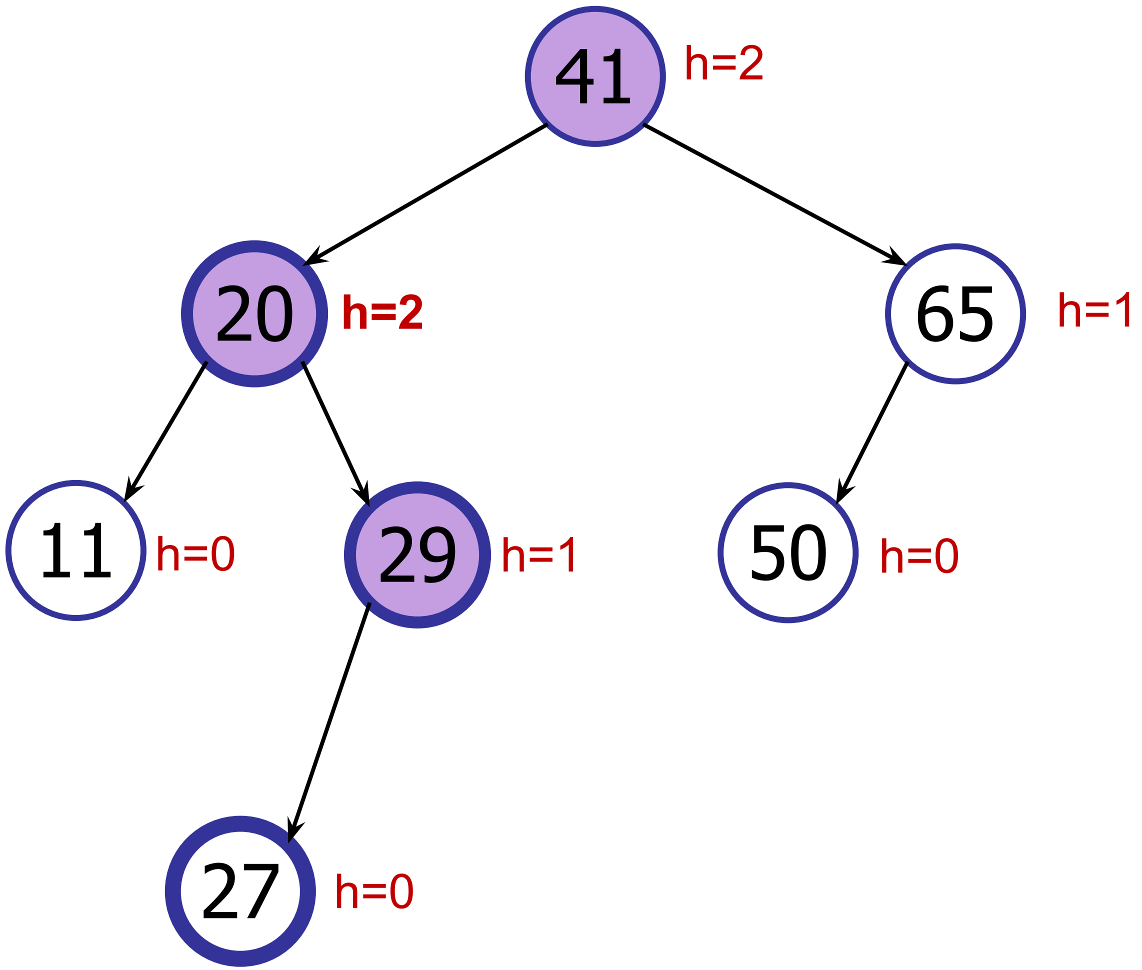
Binary Search Trees

insert(27)



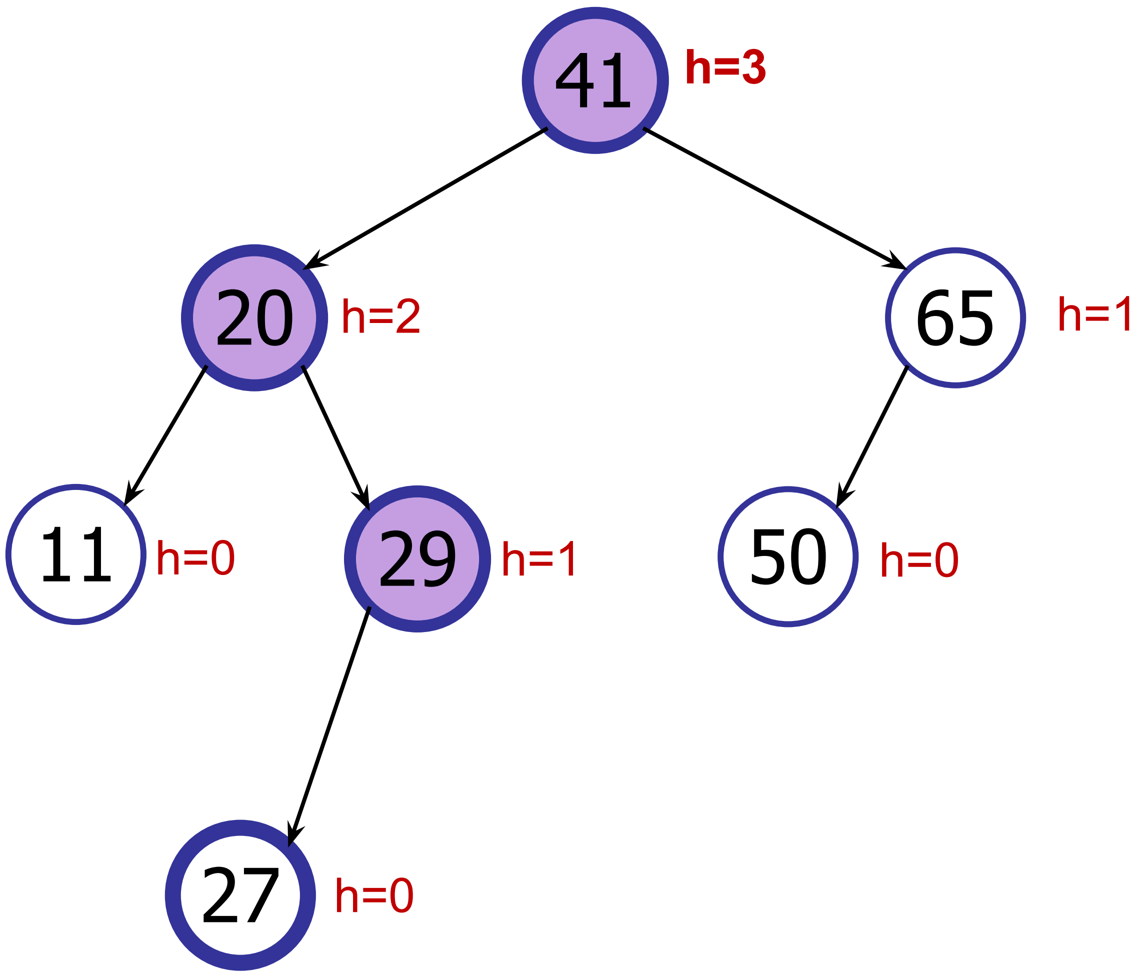
Binary Search Trees

insert(27)



Binary Search Trees

insert(27)



AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

- In every node v, store height:

  v.height = h(v)

- On insert & delete update height:

```
insert(x)
    if (x < key)
        left.insert(x)
    else right.insert(x)
    height = max(left.height, right.height) + 1
```

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

Step 1: Define Height Balance

Step 2: Maintain Balance

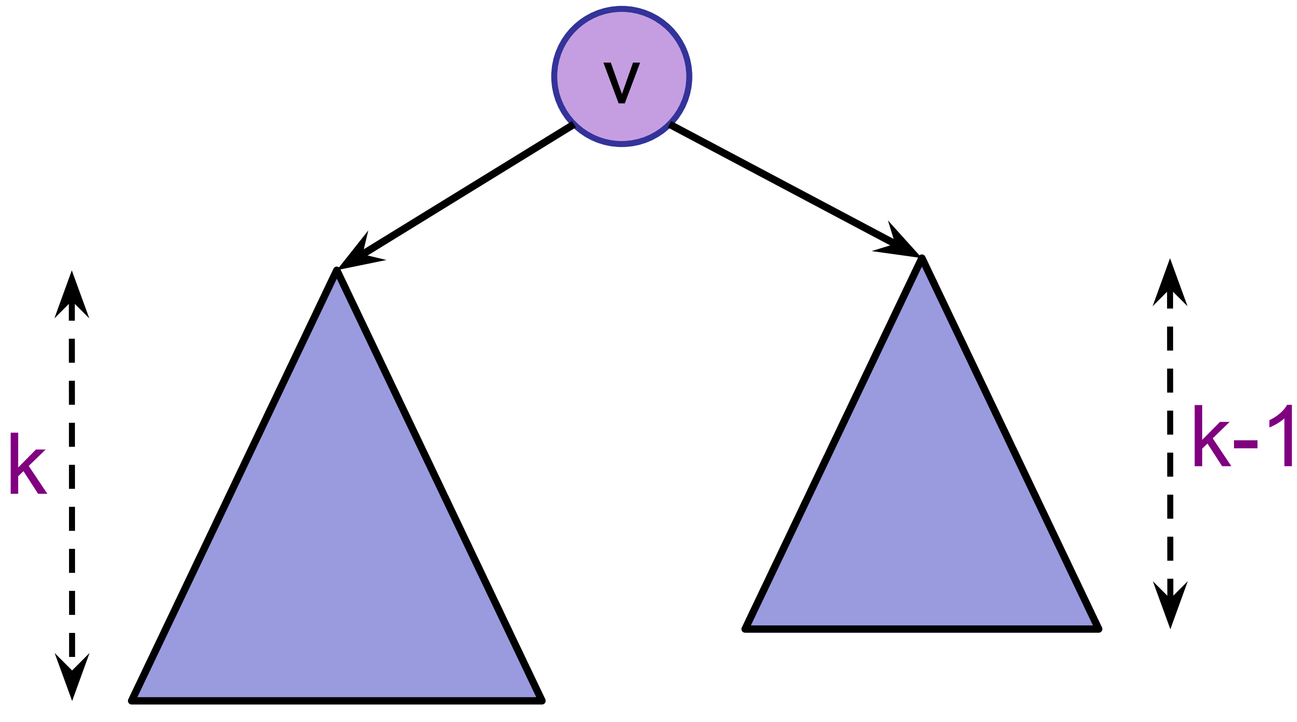
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

– A node v is <u>**height-balanced**</u> if:

$$|v.left.height - v.right.height| \le 1$$



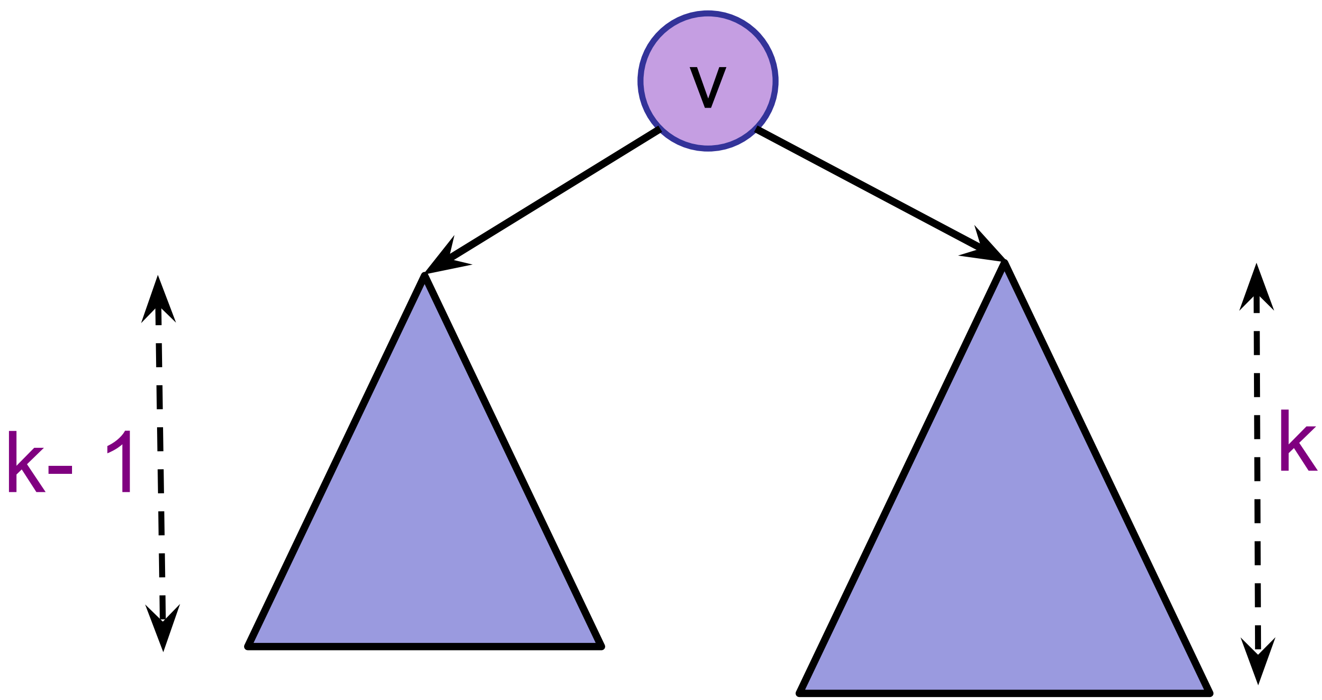
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

– A node v is **height-balanced** if:

$$|v.left.height - v.right.height| \leq 1$$



AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Define Invariant

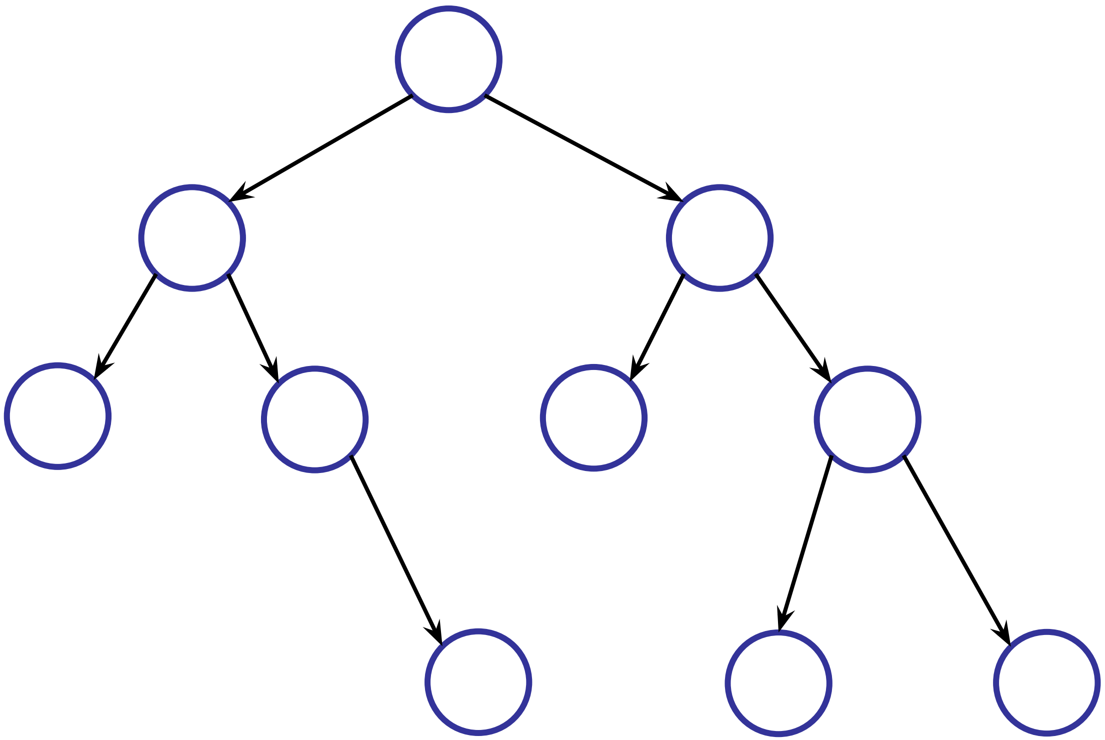
- A node v is <u>height-balanced</u> if:

  $$|v.left.height - v.right.height| \leq 1$$

- A binary search tree is <u>height balanced</u> if **every** node in the tree is height-balanced.

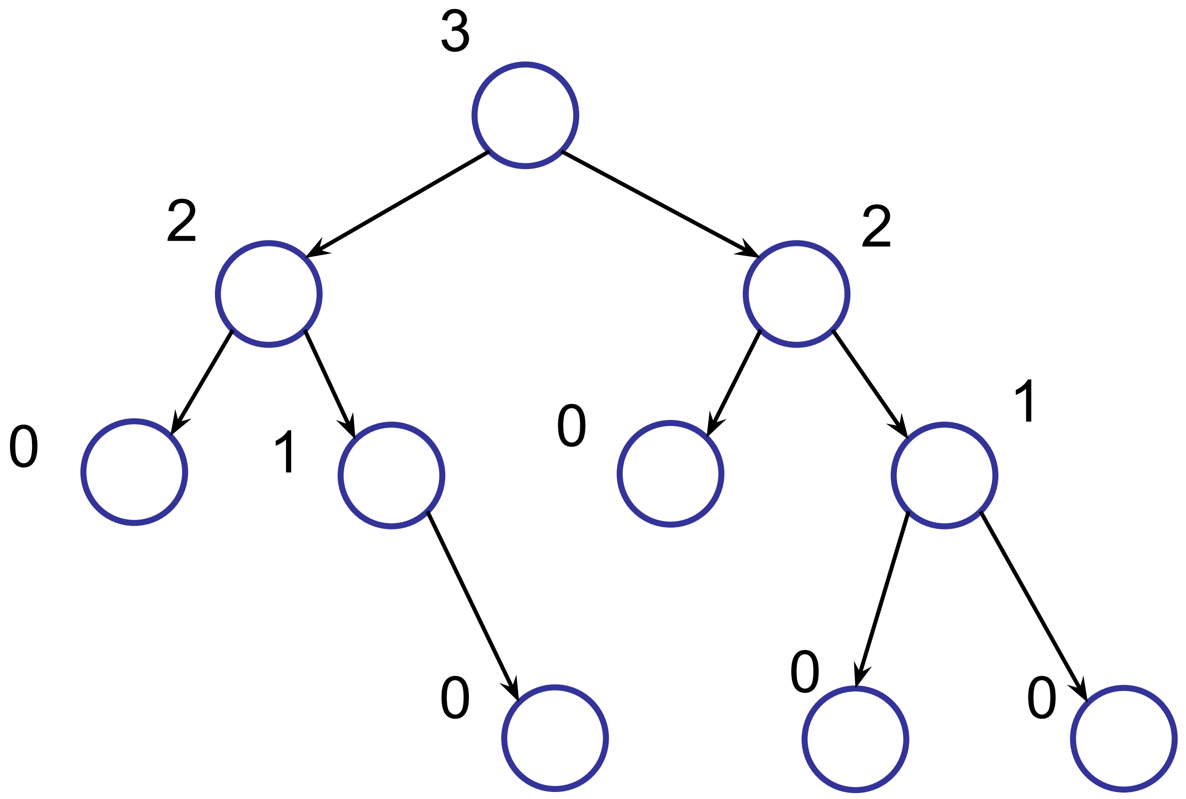
Is this tree height-balanced?

1. Yes
2. No
3. I’m confused.



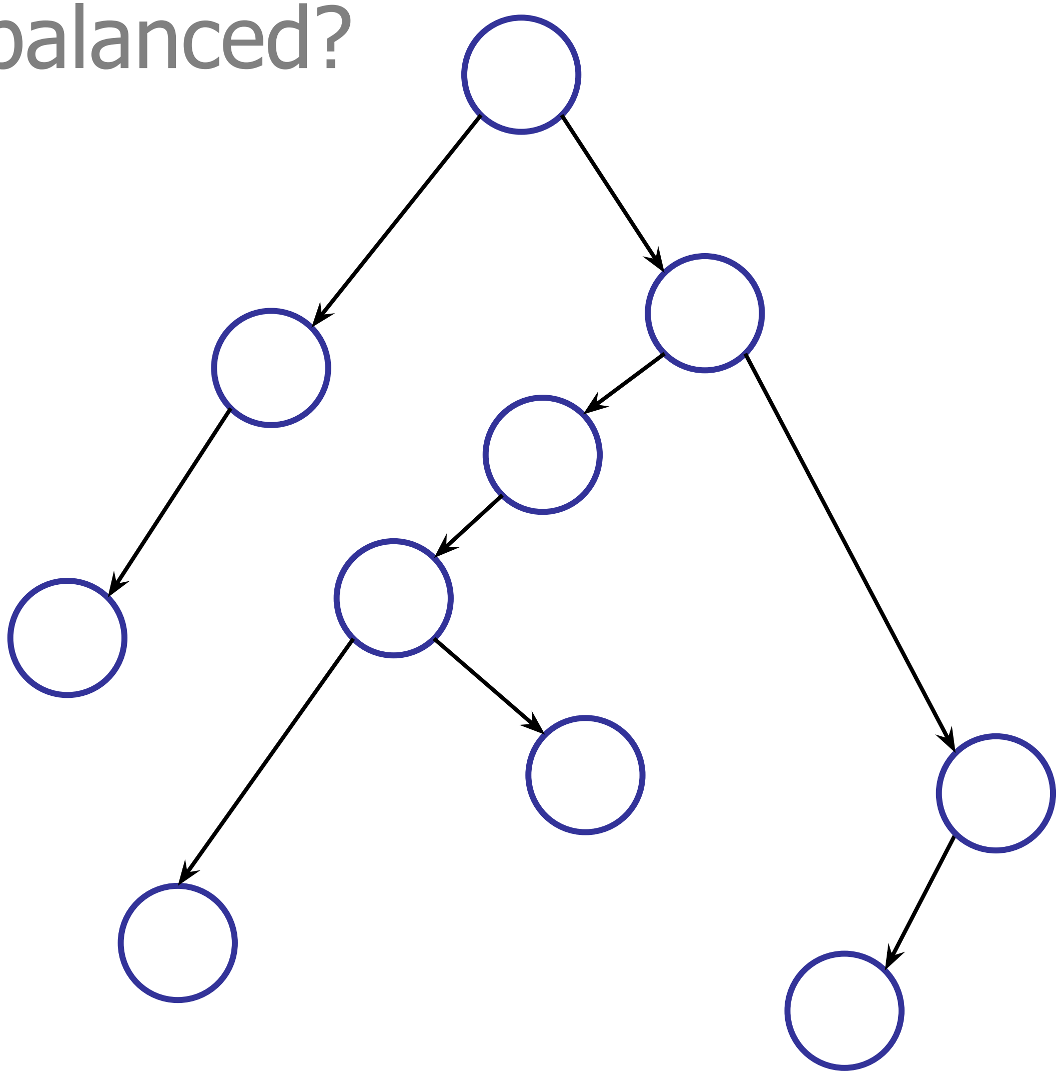
Is this tree height-balanced?

1. Yes
2. No
3. I’m confused.



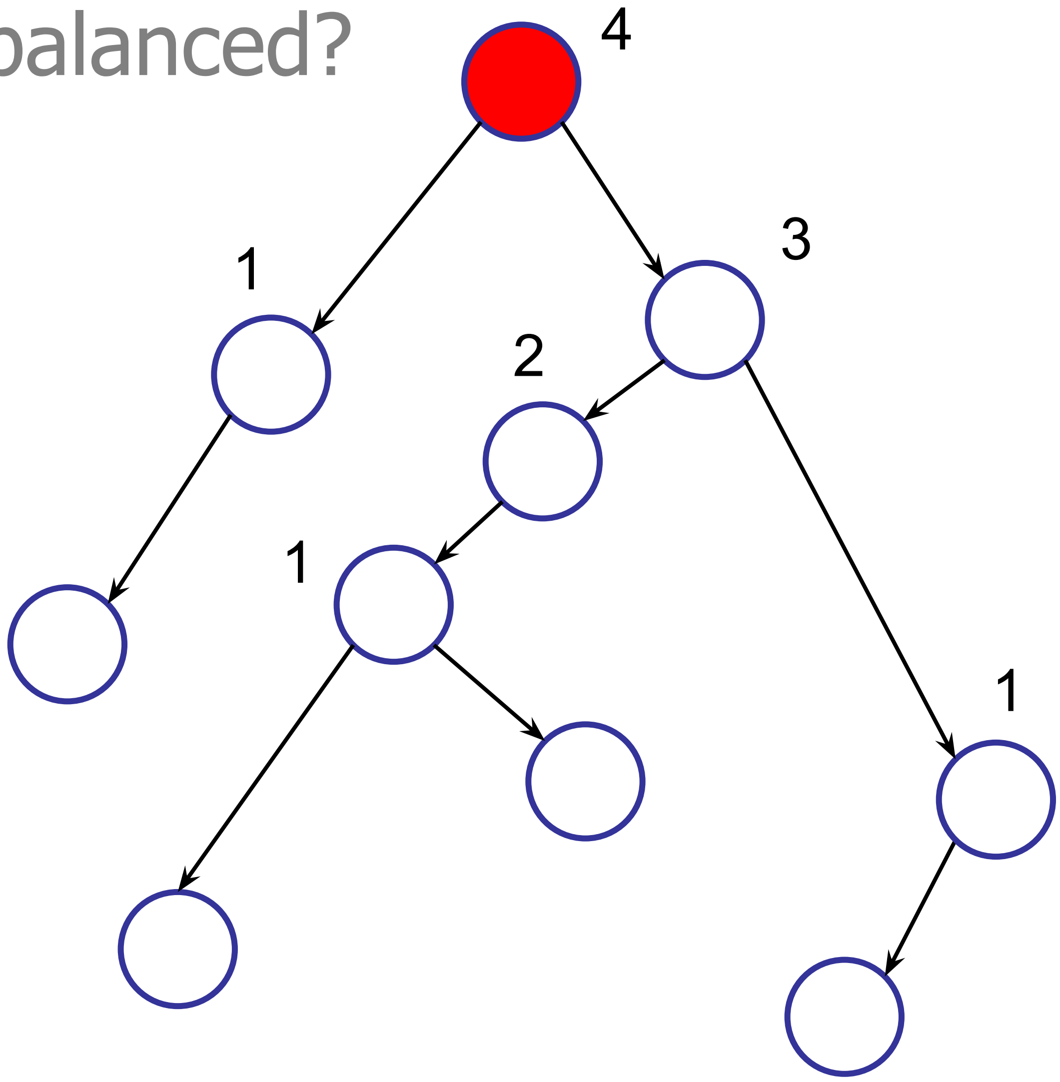
Is this tree height-balanced?

1. Yes
2. No
3. I'm confused.



Is this tree height-balanced?

1. Yes
2. No
3. I’m confused.



Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has **<u>at most</u>** height $h < 2\log(n)$.

Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has **at most** height $h < 2\log(n)$.

If we can prove this fact, we can say our operations cost $O(h) = O(\log n)$ time.

Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has **at most** height $h < 2\log(n)$.

$\Leftrightarrow h/2 < \log(n)$

$\Leftrightarrow 2^{h/2} < 2^{\log(n)}$

$\Leftrightarrow 2^{h/2} < n$

Equivalent claim:

A height-balanced tree with height h has

at least $n > 2^{h/2}$ nodes

Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has **at most** height $h < 2\log(n)$.

$\Leftrightarrow h/2 < \log(n)$

$\Leftrightarrow 2^{h/2} < 2^{\log(n)}$

$\Leftrightarrow 2^{h/2} < n$

We will prove this claim instead

Equivalent claim:

A height-balanced tree with height h has

at least $n > 2^{h/2}$ nodes

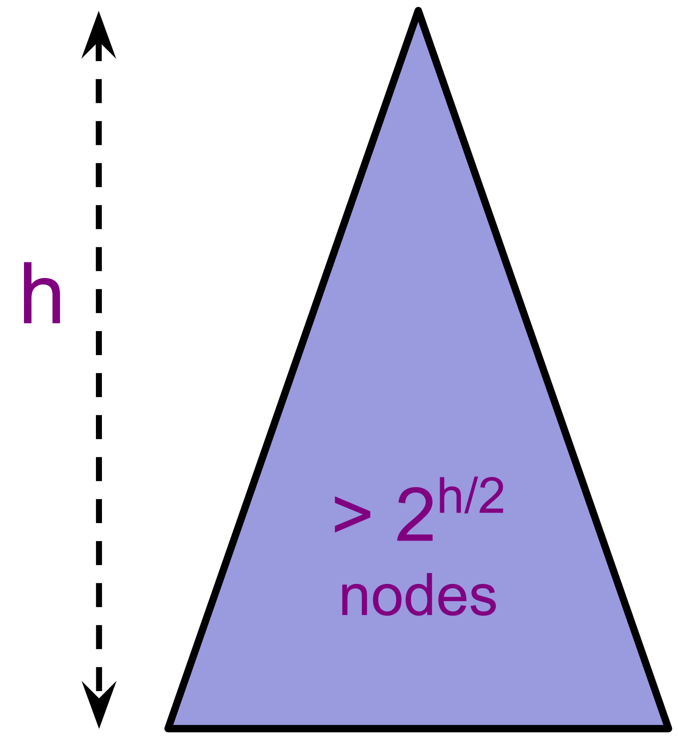
Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

Show:

$n_h > 2^{h/2}$

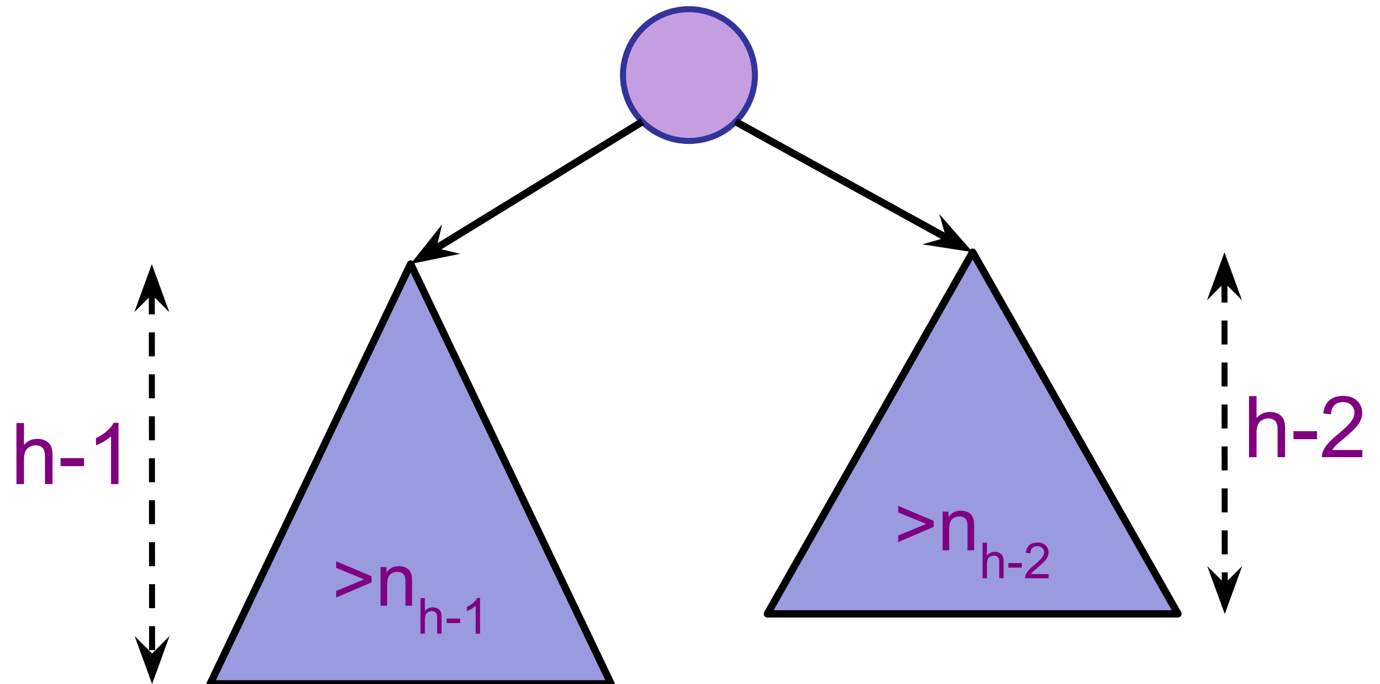


Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$n_h \geq 1 + n_{h-1} + n_{h-2}$



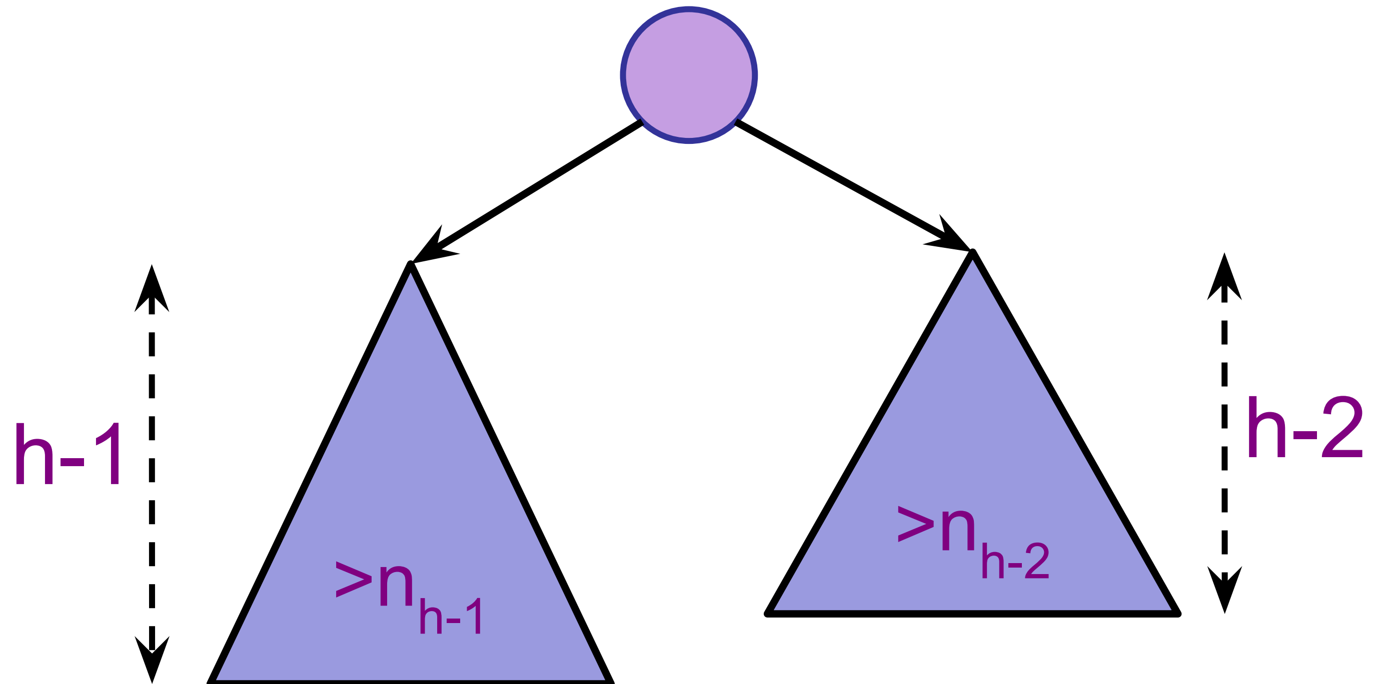
Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$



Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$

$$\geq 4n_{h-4}$$

$$\geq 8n_{h-6}$$

$$\geq \ldots$$

How many times?

Base case: $n_0 = 1$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2^1 n_{h-2}$$

$$\geq 2^2 n_{h-4}$$

$$\geq 2^3 n_{h-6}$$

$$\geq \ldots \geq 2^k n_0$$

What is k?

Base case:
$n_0 = 1$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$

$$\geq 2^{h/2}\, n_0$$

$$\geq 2^{h/2}$$

Base case:
$n_0 = 1$

Height-Balanced Trees

Claim:

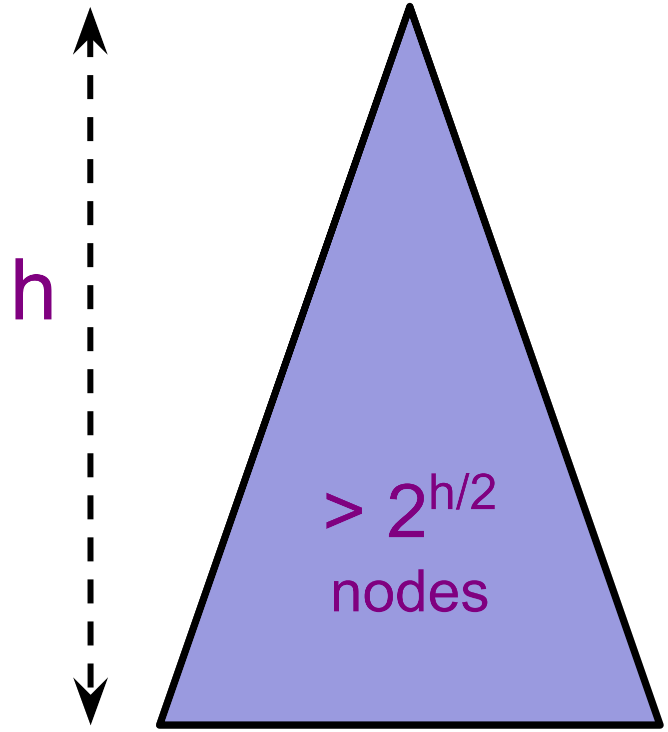
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

$n_h > 2^{h/2}$

$\Leftrightarrow$

$h < 2\log(n_h)$



Height-Balanced Trees

Claim:

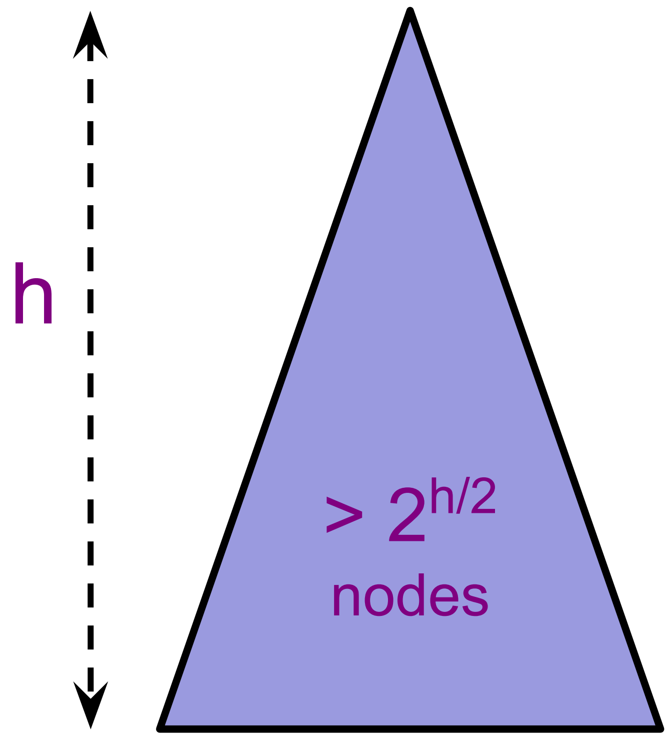
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

$n_h > 2^{h/2}$

$\Leftrightarrow$

$h < 2\log(n_h)$



Height-Balanced Trees

Claim:

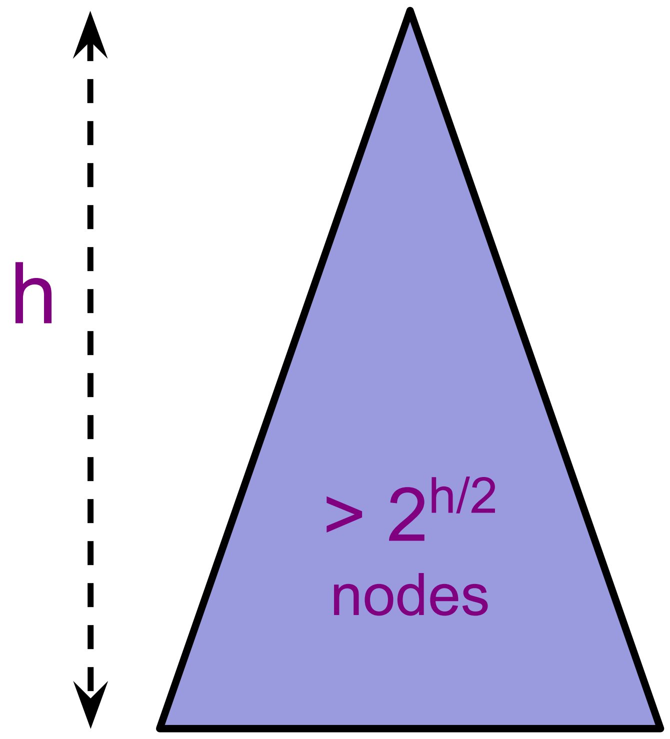
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

$$n_h > 2^{h/2}$$

$$\Leftrightarrow$$

$$h < 2\log(n_h)$$



Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has height $h < 2\log(n)$.

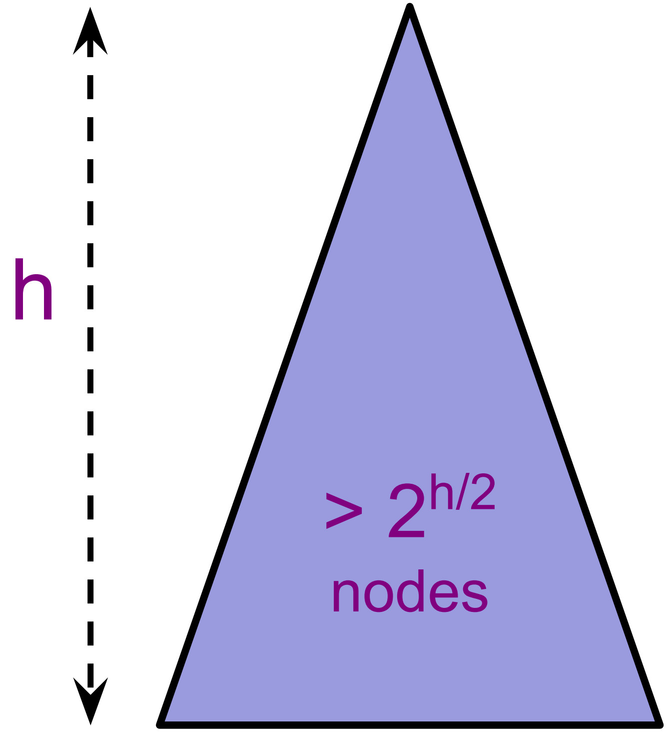
Show:

Is this constant tight?

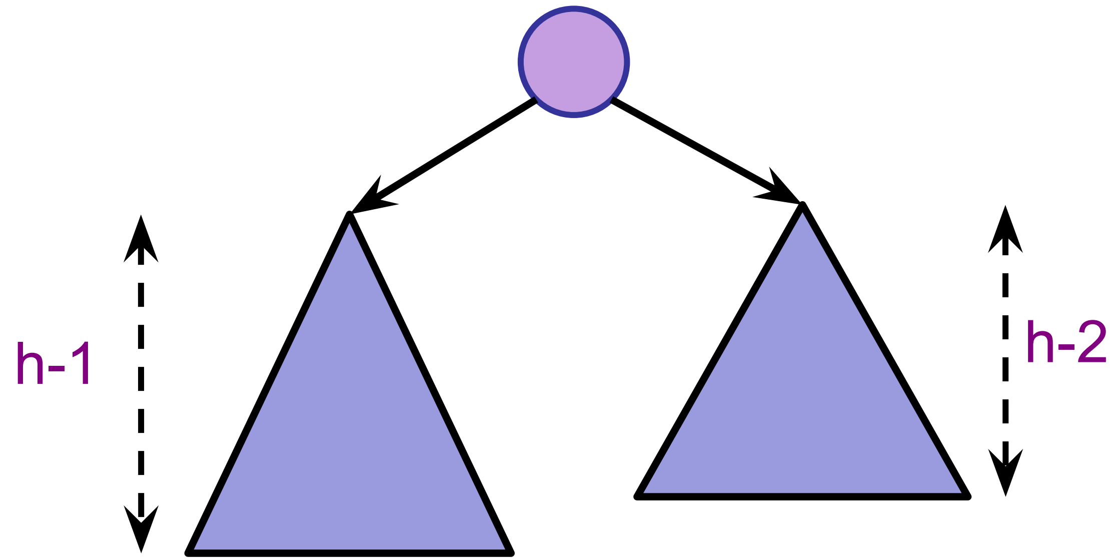
$$n_h > 2^{h/2}$$

$$\Leftrightarrow$$

$$h < 2\log(n_h)$$



Height-Balanced Trees



Show (induction):

F_n = n^{th} Fibonacci number

$n_h = F_{h+2} - 1 \cong \varphi^{h+1}/\sqrt{5} - 1$ (rounded to nearest int)

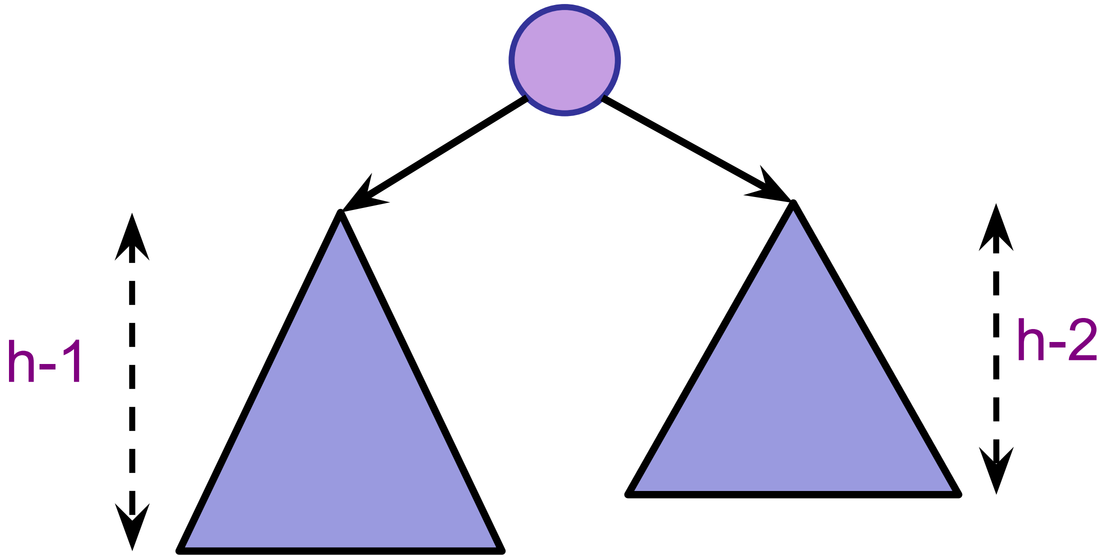
$h \cong \log(n) / \log(\varphi)$ $\qquad \varphi \cong 1.618$

$h \cong 1.44 \log(n)$

Height-Balanced Trees

Claim:

A height-balanced tree is balanced, i.e., has height $h = O(\log n)$.



AVL Trees [Adelson-Velskii & Landis 1962]

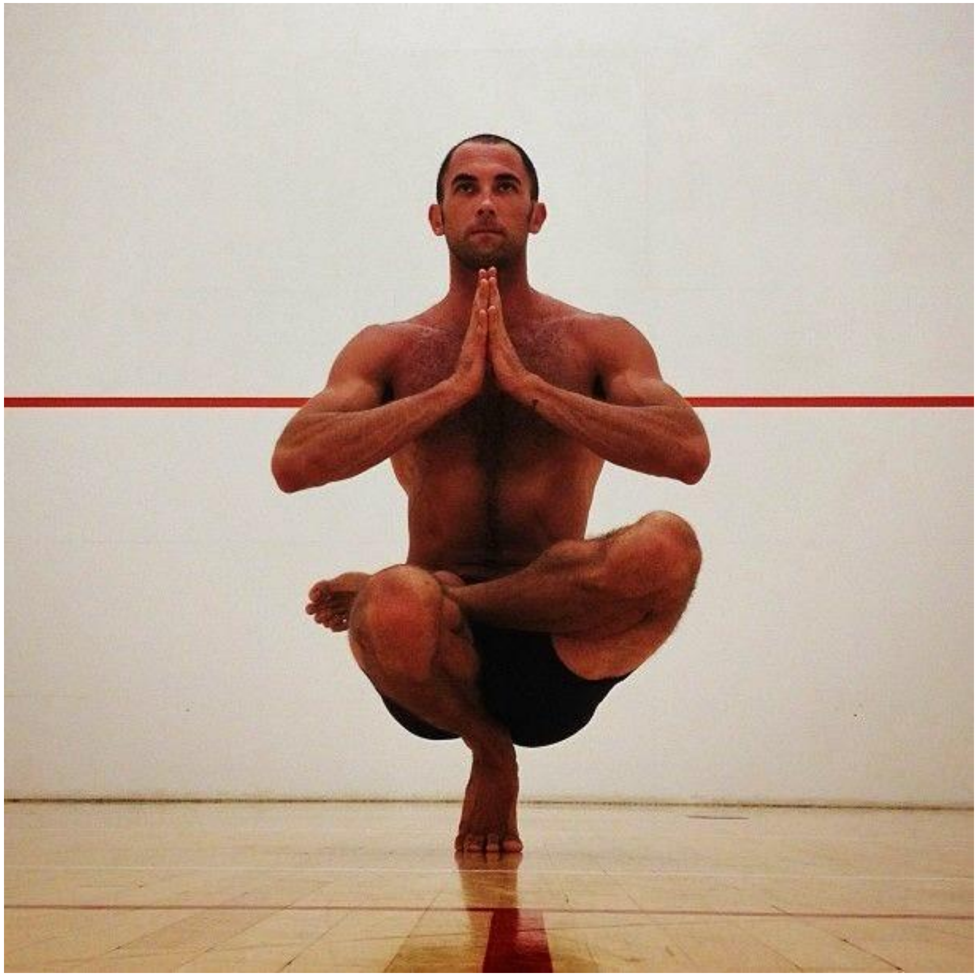
Step 0: Augment

Step 1: Define Height Balance

Step 2: Maintain Balance

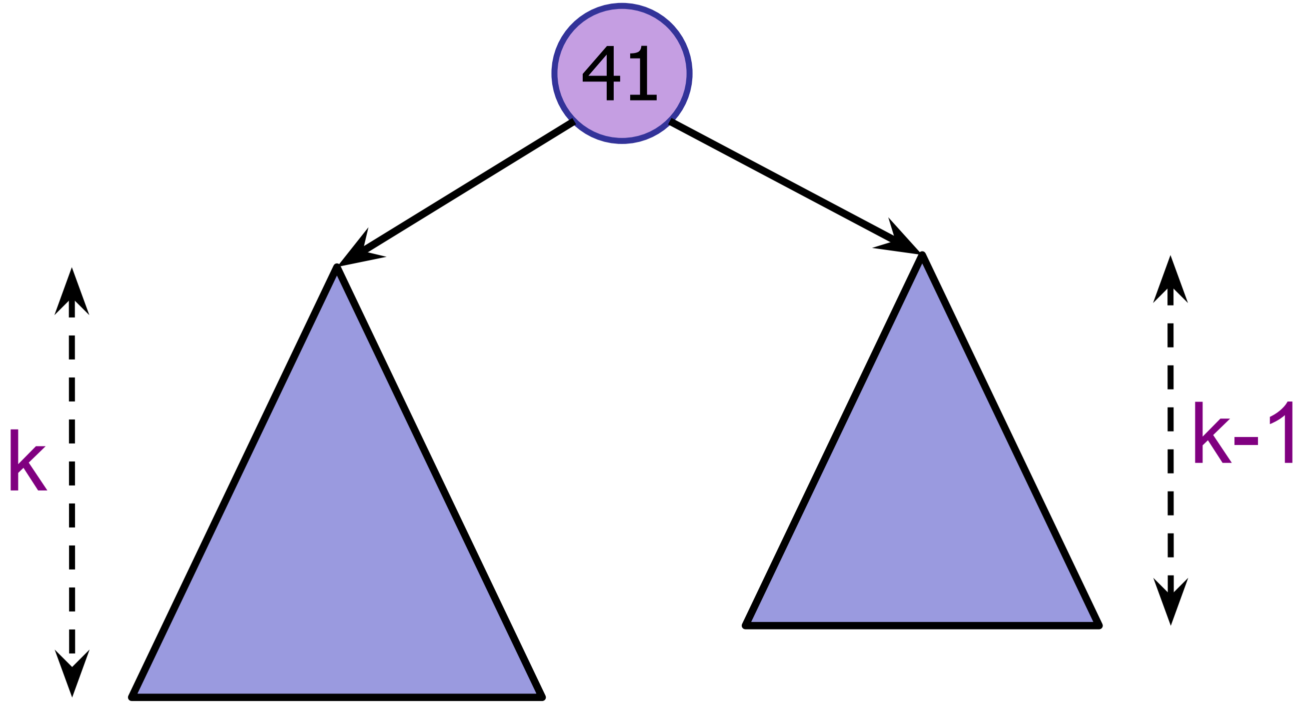
It's good that we don't have to

Balance perfectly



AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Show how to maintain height-balance



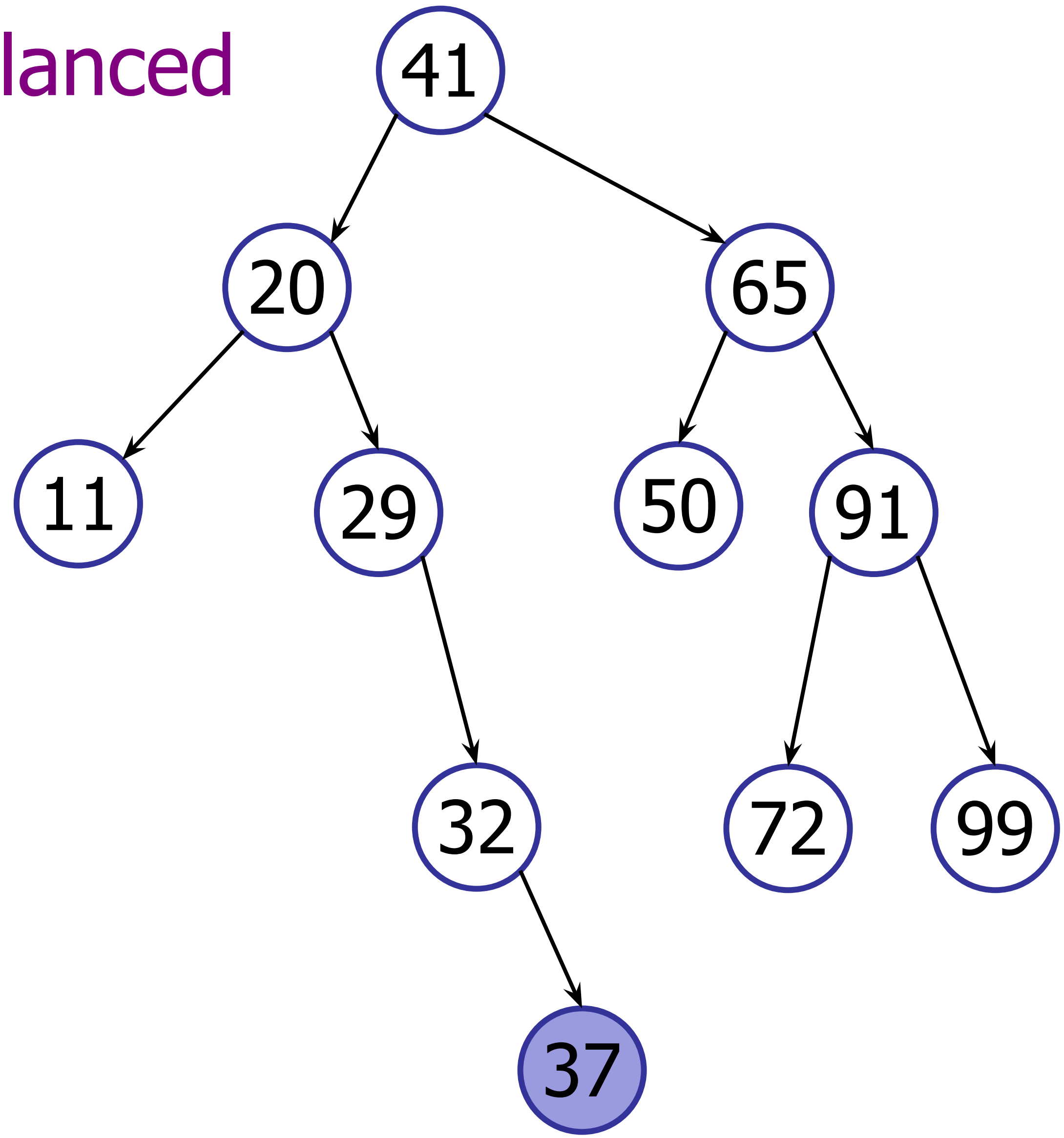
Inserting in an AVL Tree

Before insertion, balanced

insert(37)

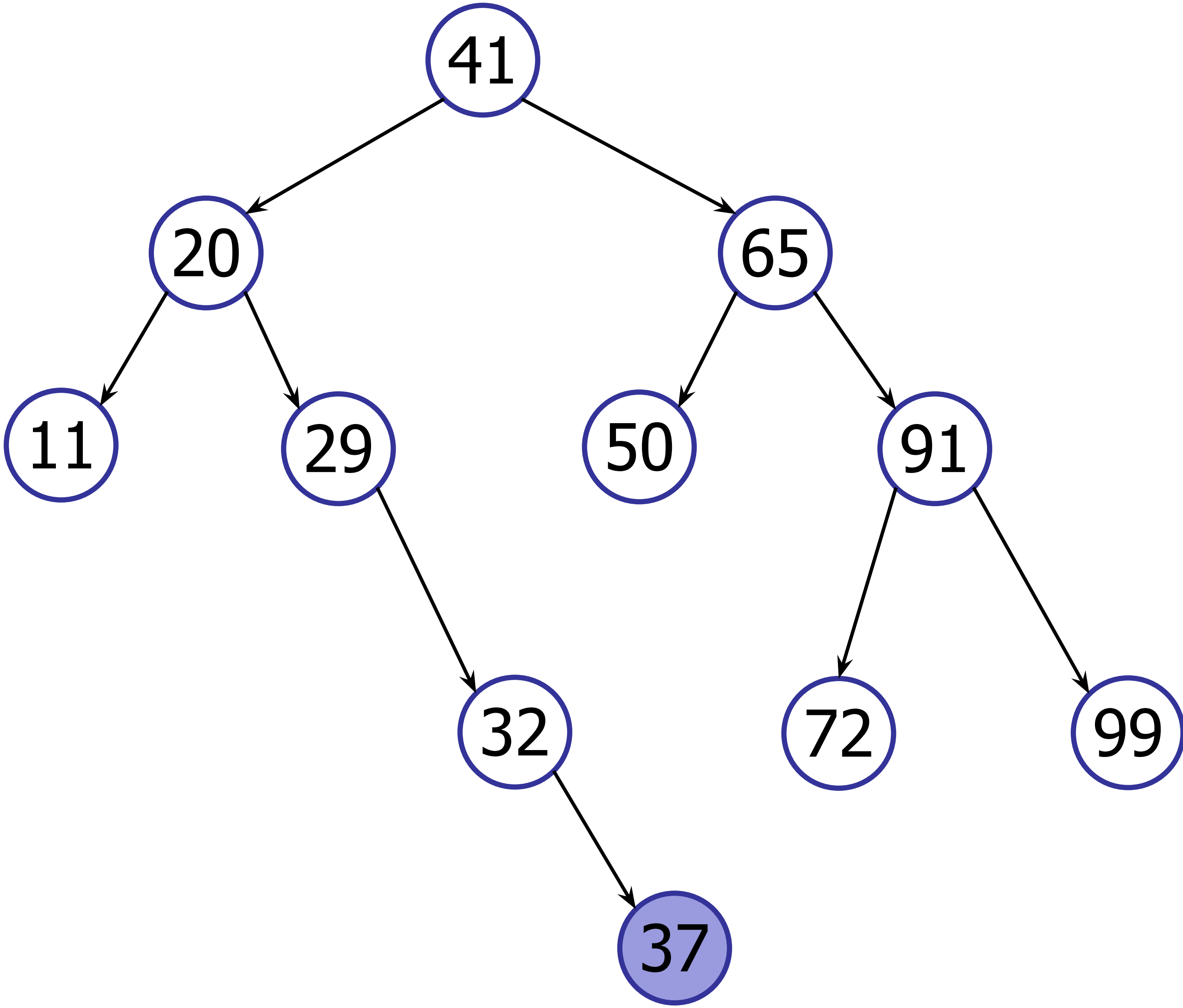
No longer balanced
after insertion!

Need to rebalance!



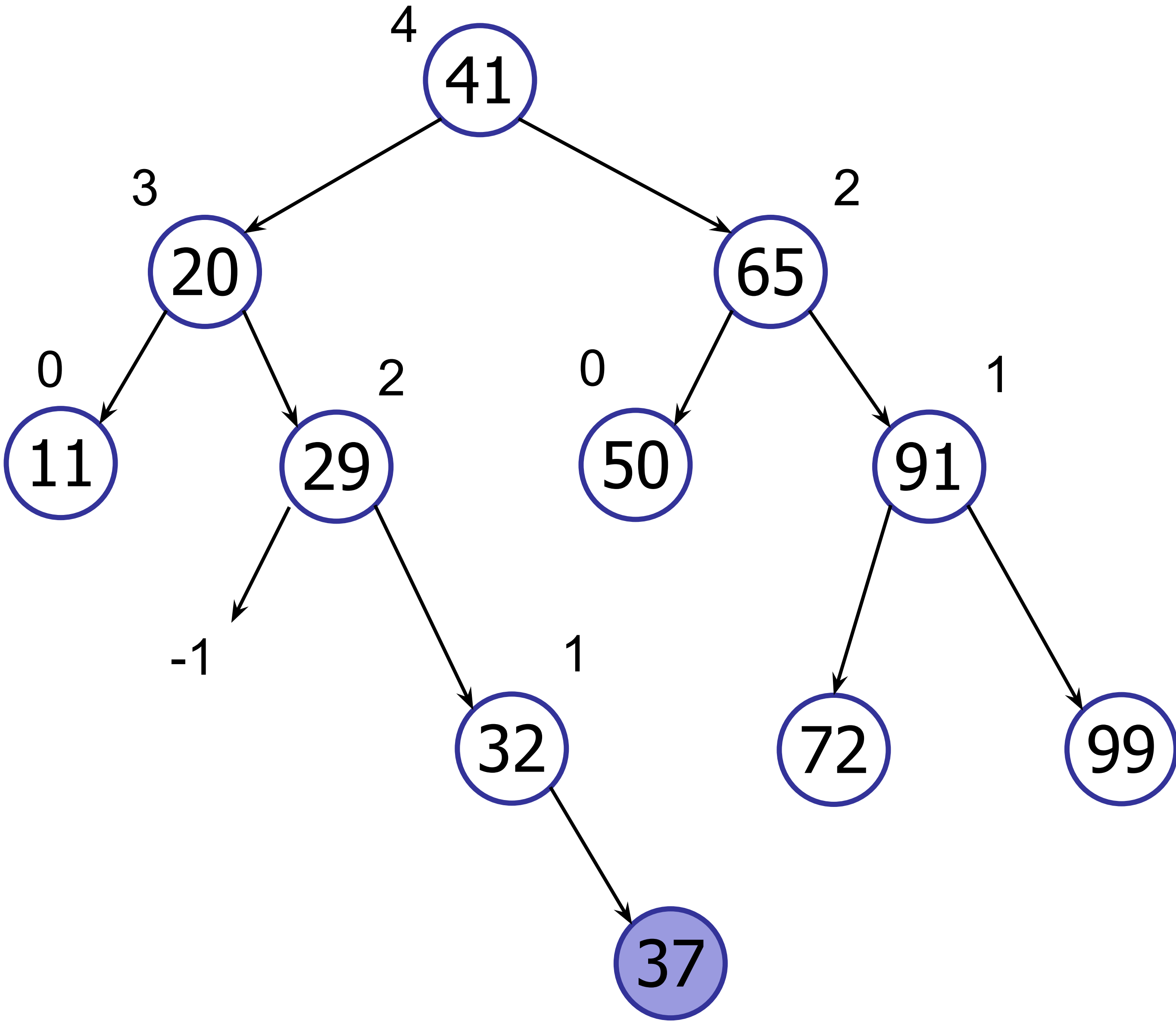
Which nodes need rebalancing?

1. 41
2. 20
3. 11
4. 29
5. 32
6. 37
7. 65



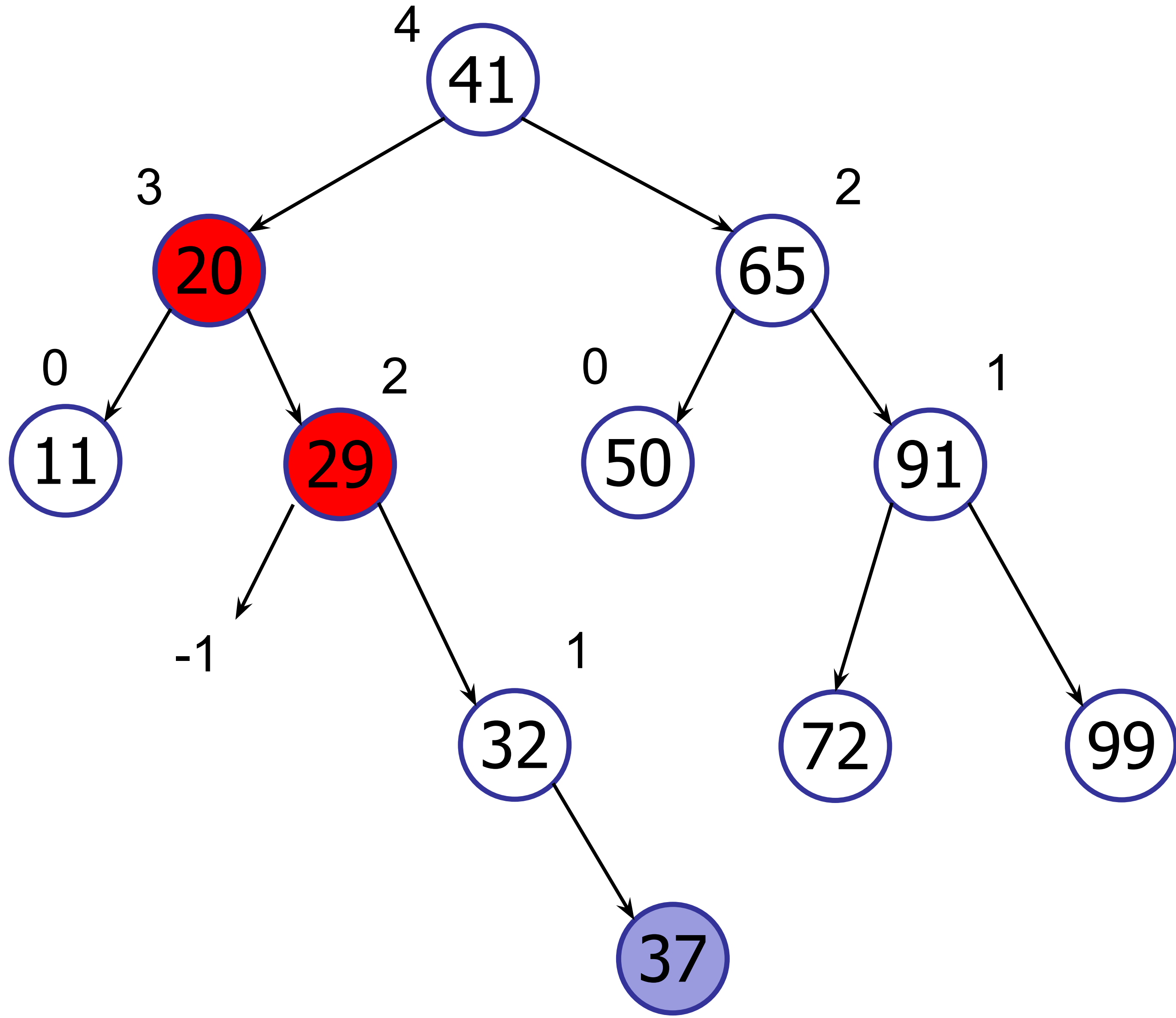
Which nodes need rebalancing?

1. 41
2. 20
3. 11
4. 29
5. 32
6. 37
7. 65



Which nodes need rebalancing?

1. 41
2. 20
3. 11
4. 29
5. 32
6. 37
7. 65

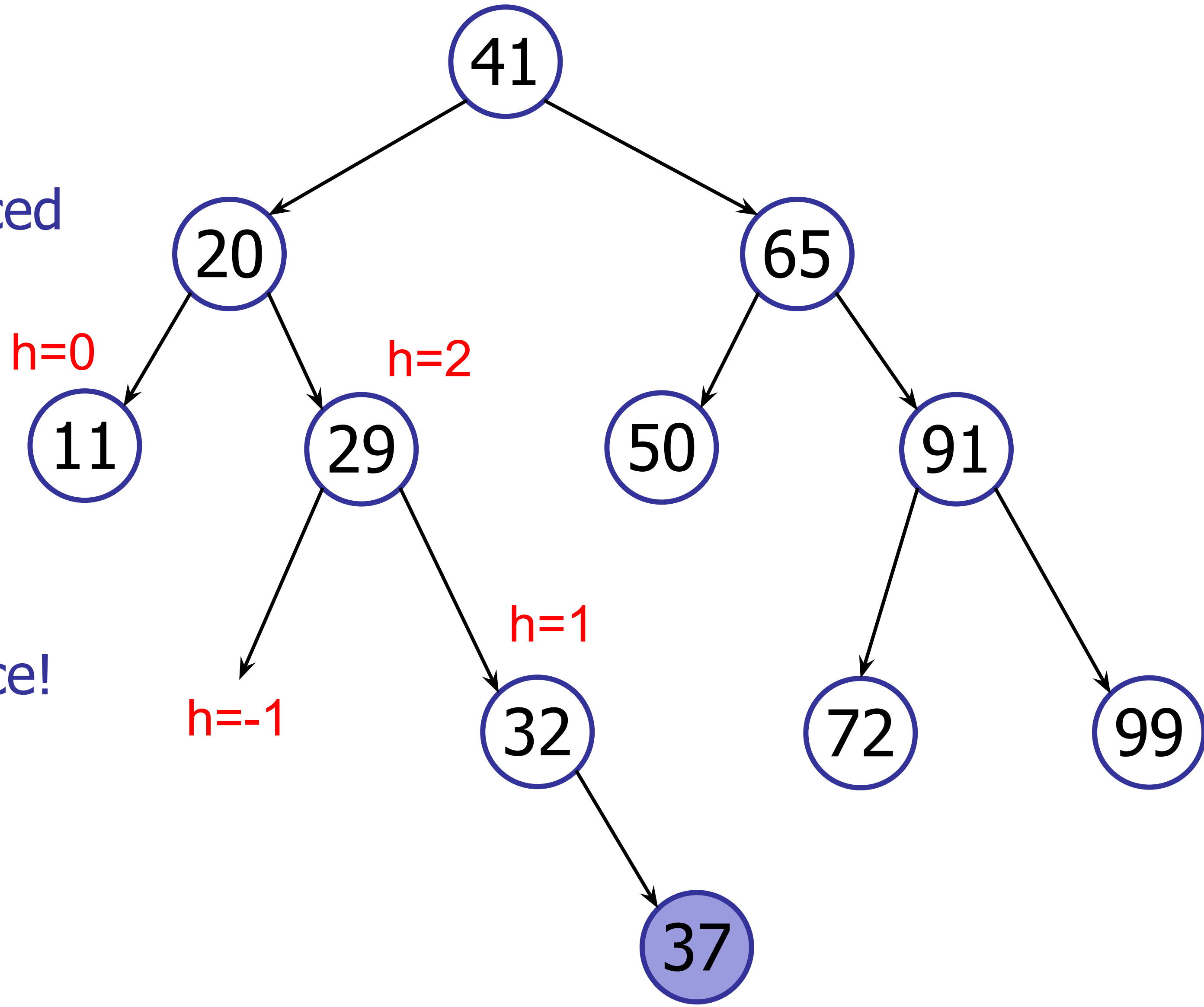


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

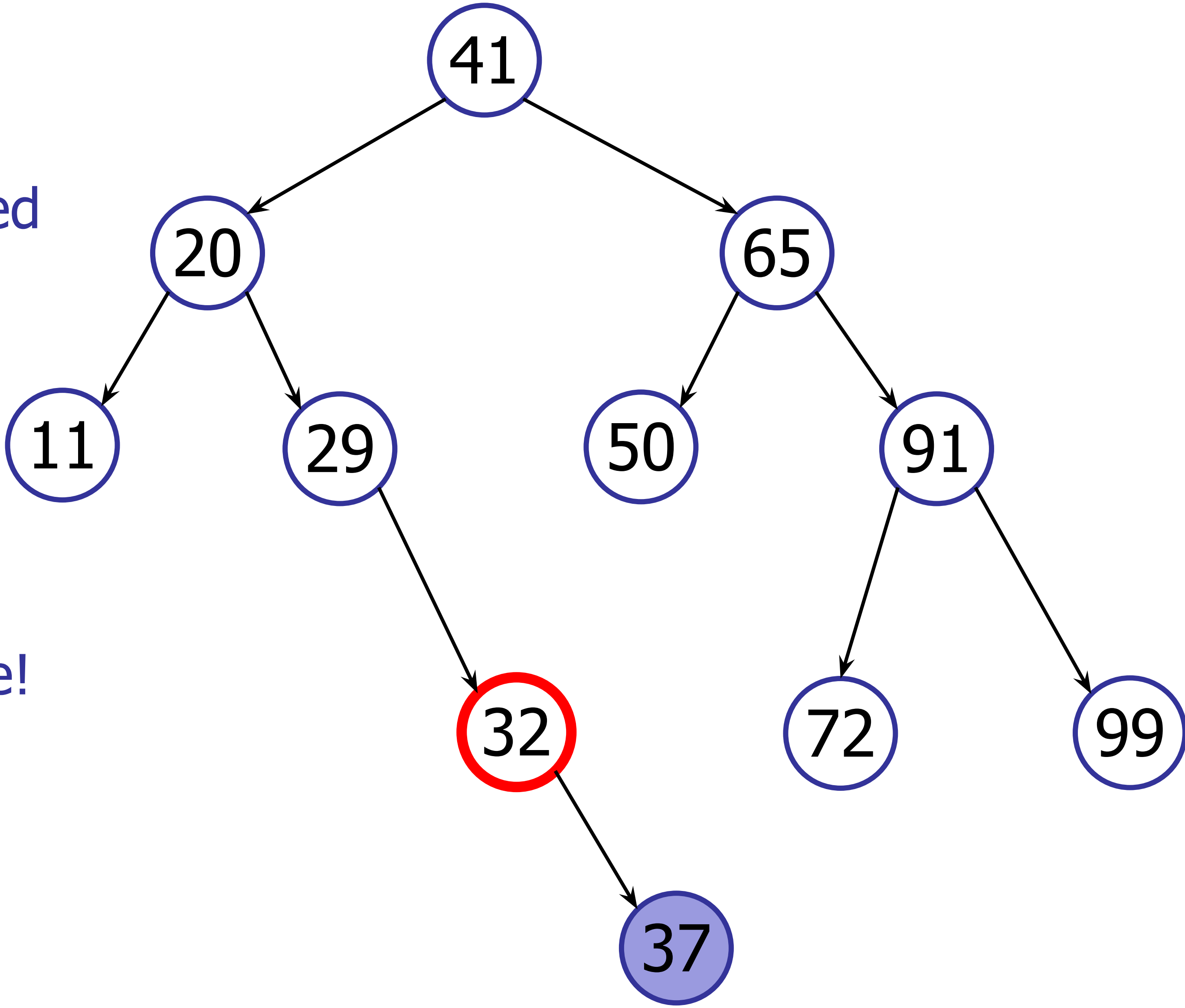


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

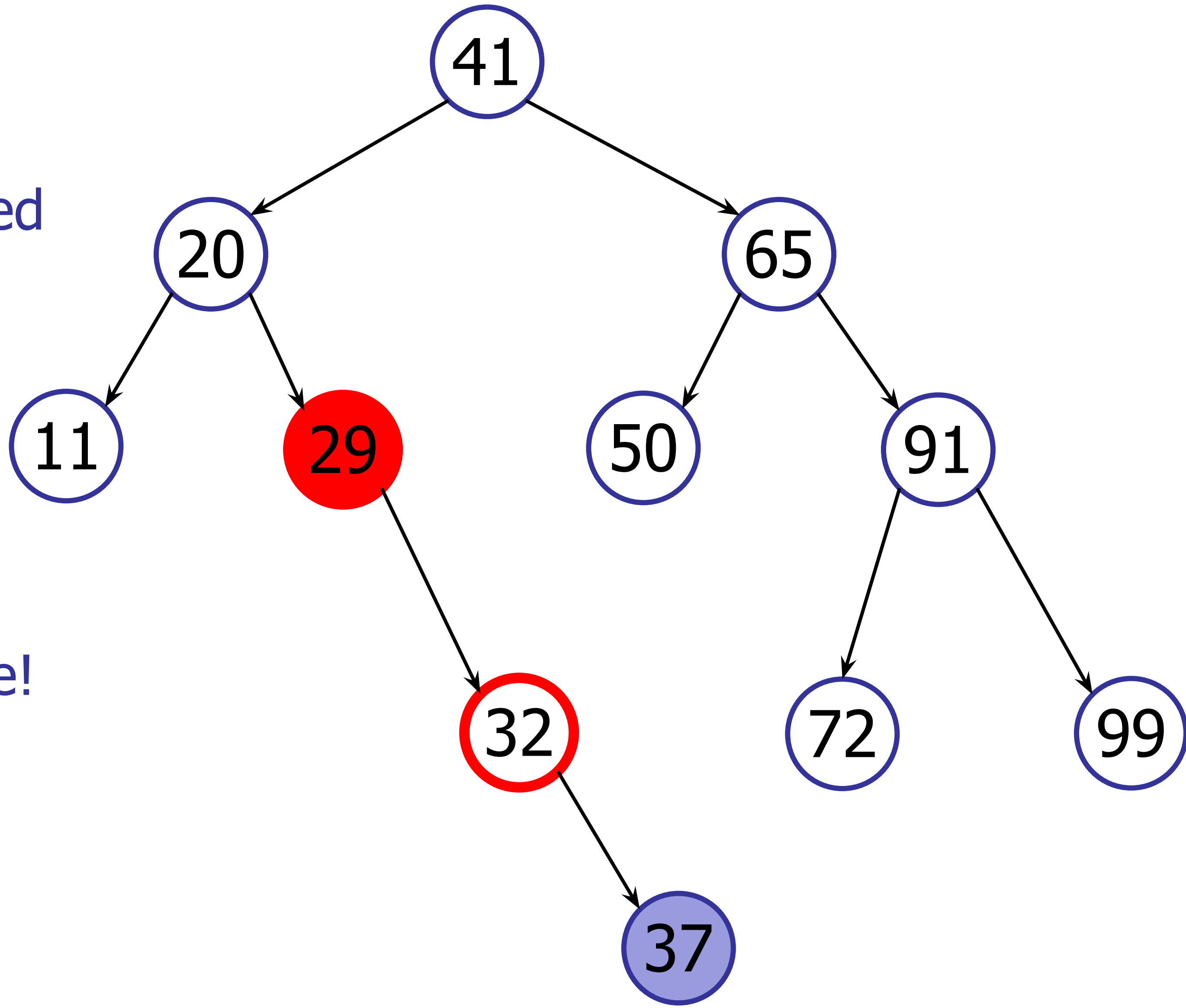


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

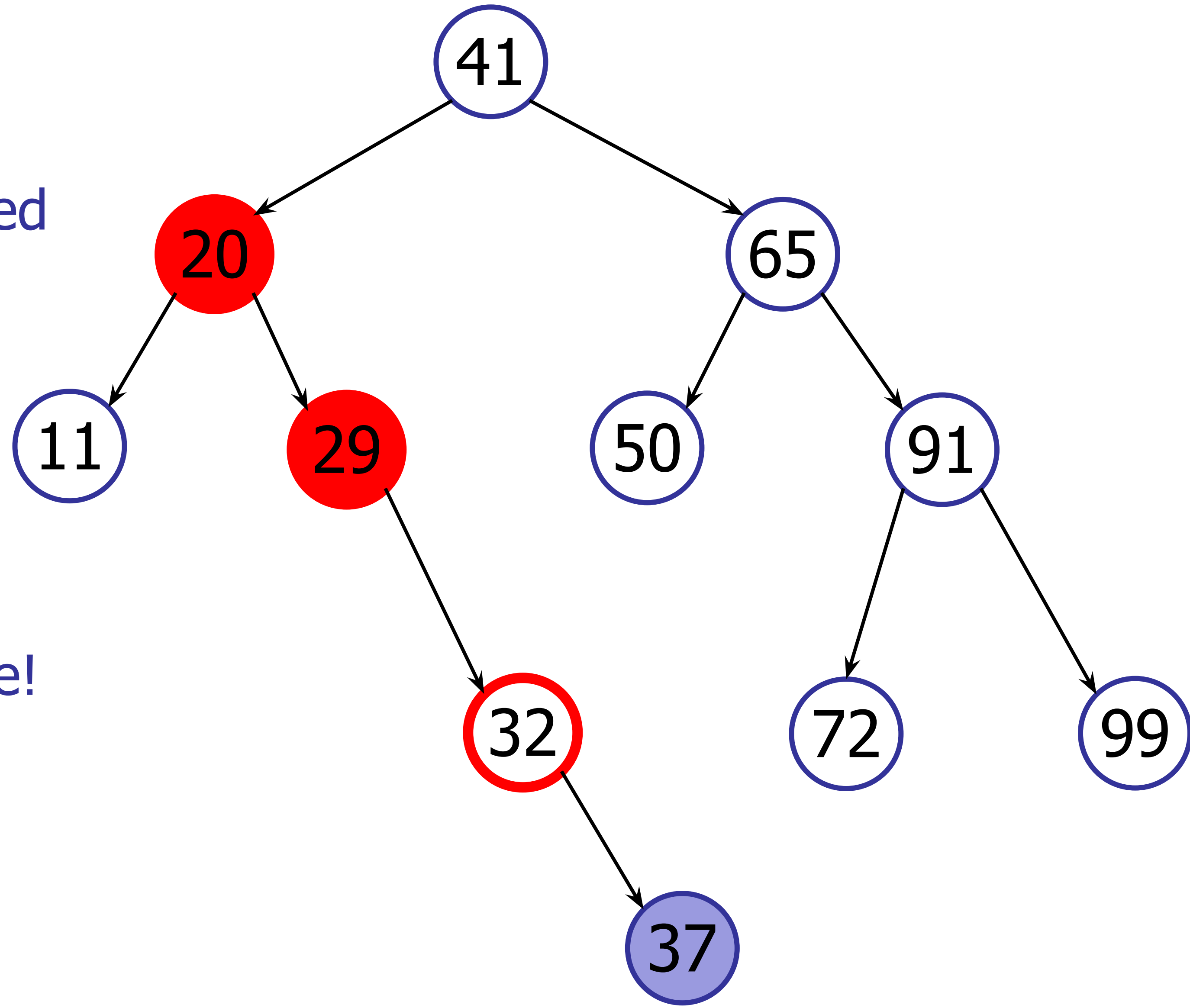


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

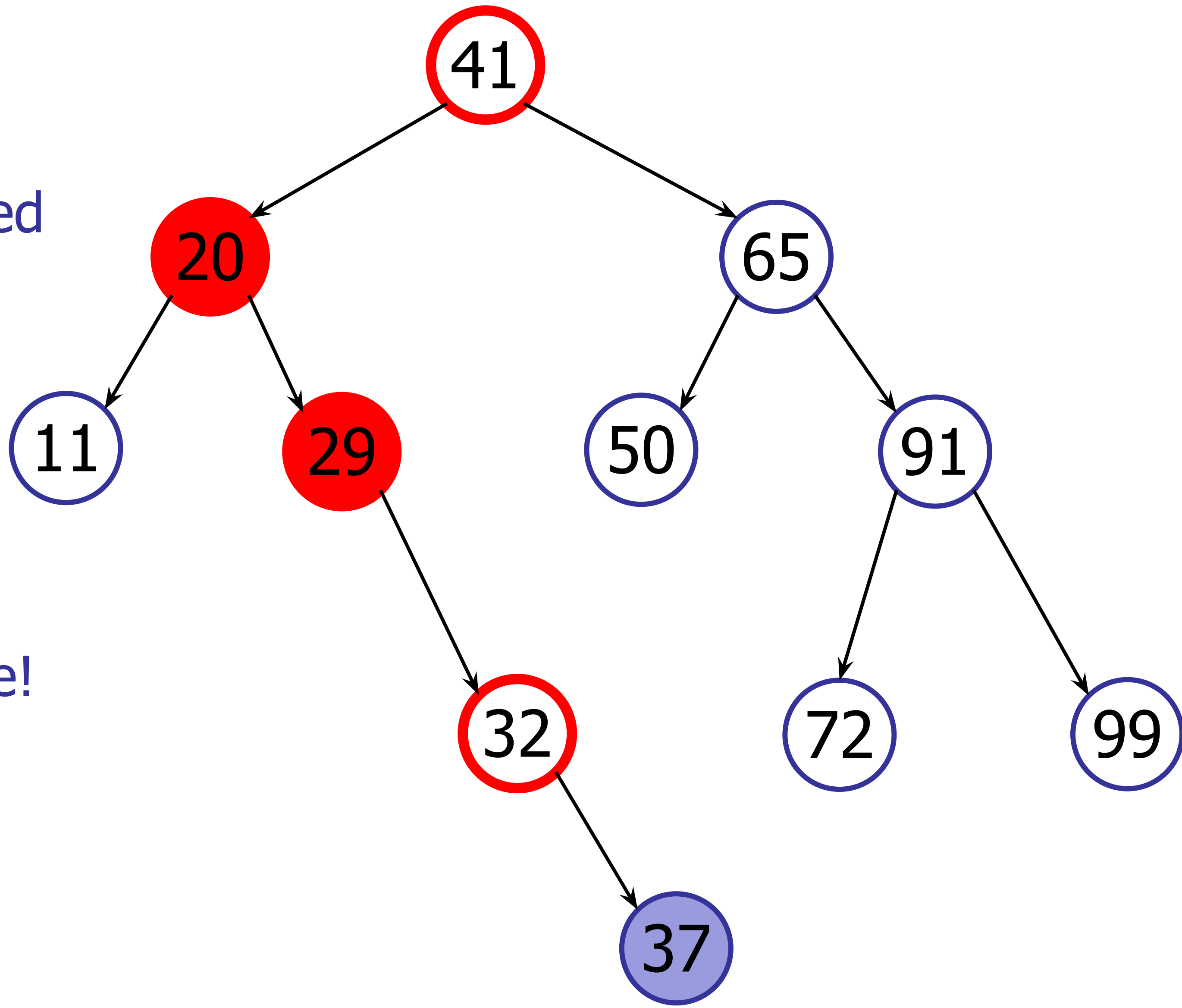


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

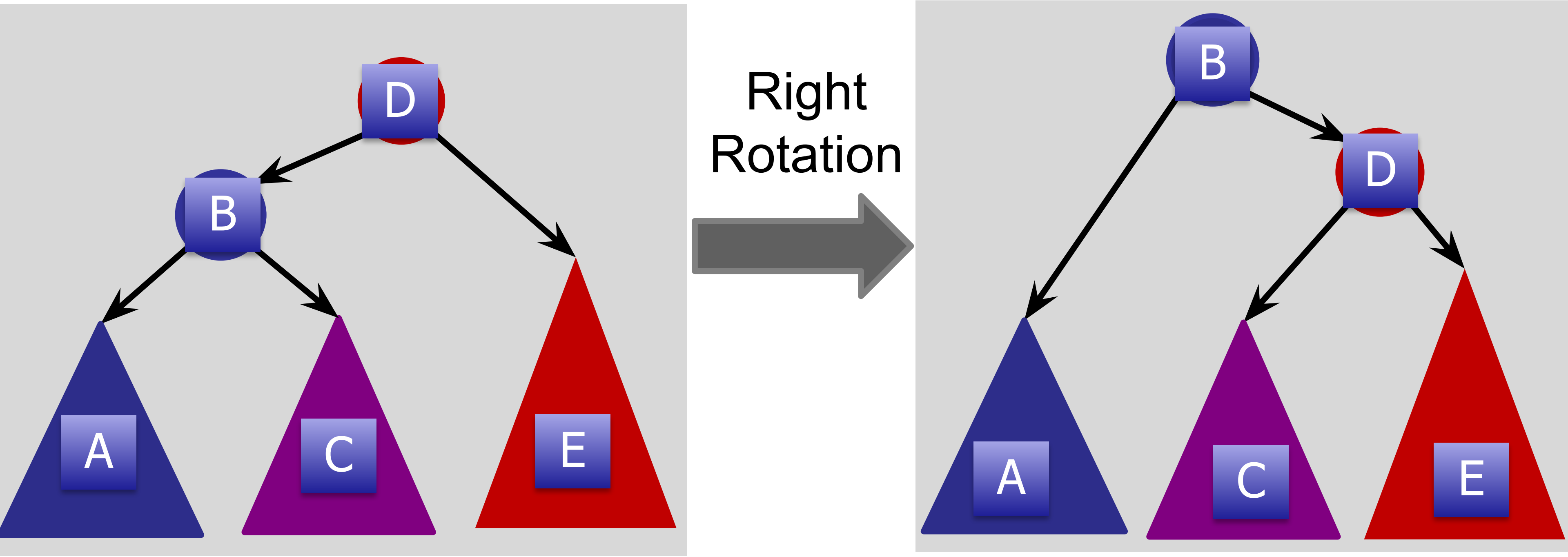
Need to rebalance!



Trick to rebalance the tree

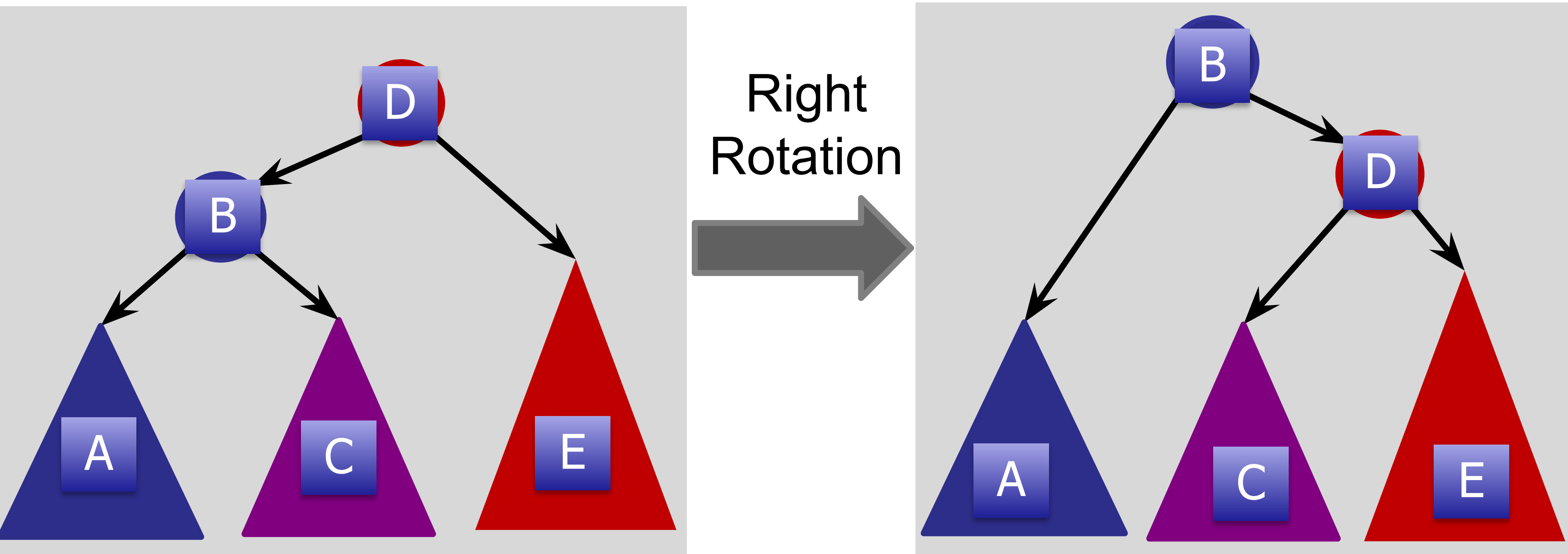
Tree rotation!

Tree Rotations



A < B < C < D < E

Tree Rotations

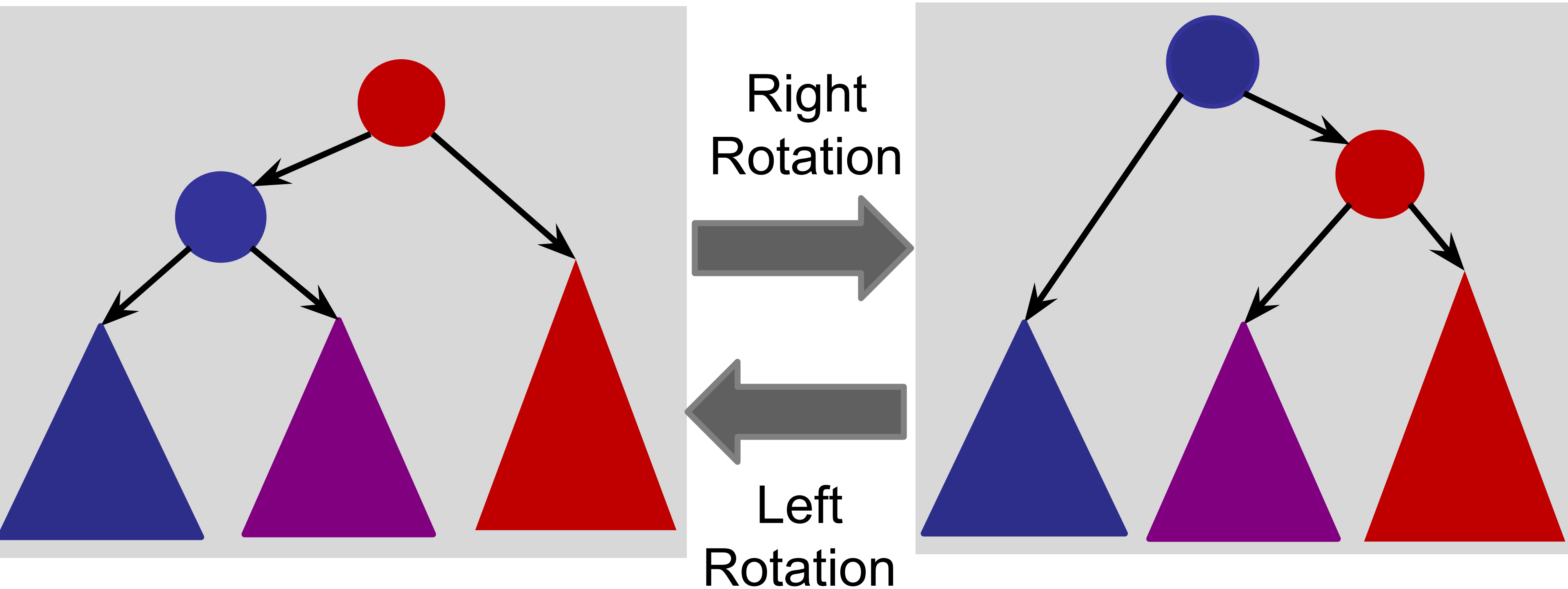


$A < B < C < D < E$

Rotations maintain ordering of keys.

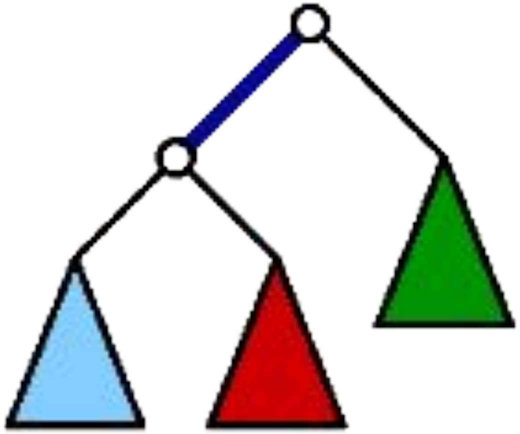
$\Rightarrow$ Maintains BST property.

Tree Rotations

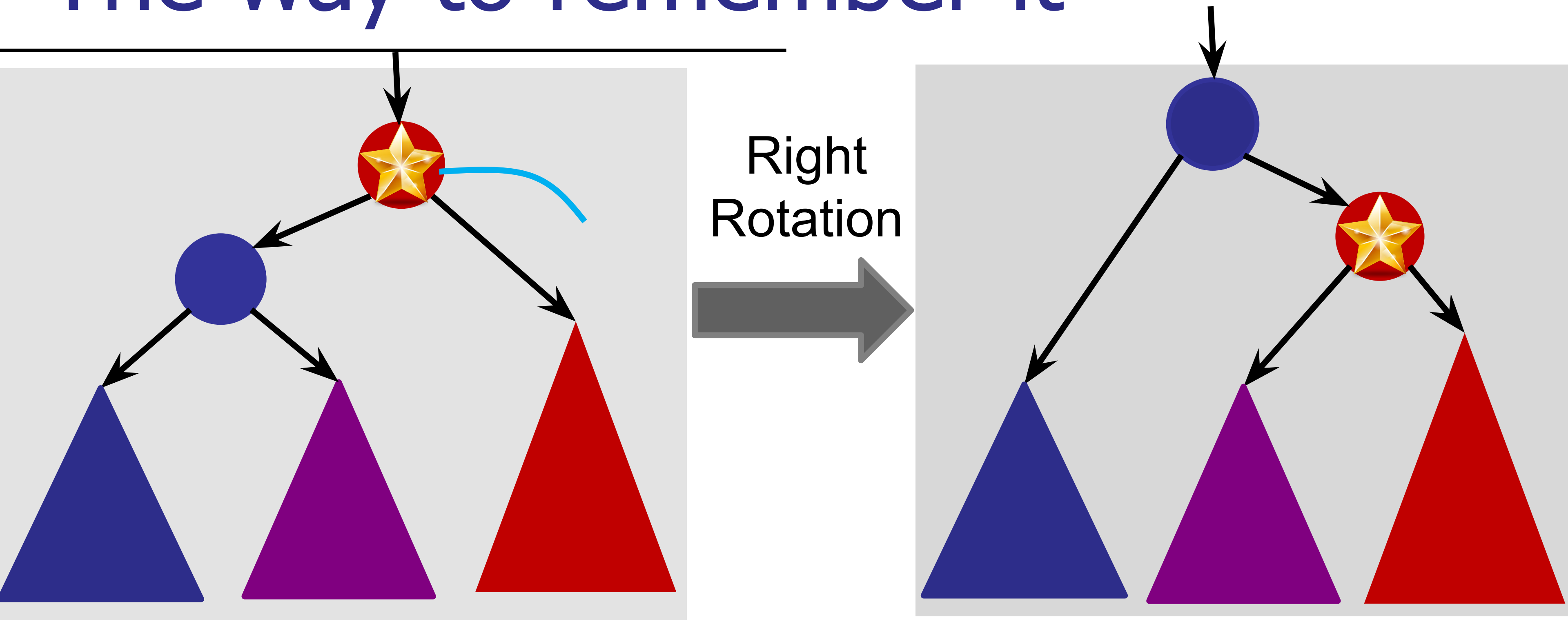


Wait….

What is a left rotation and what is a right rotation!?

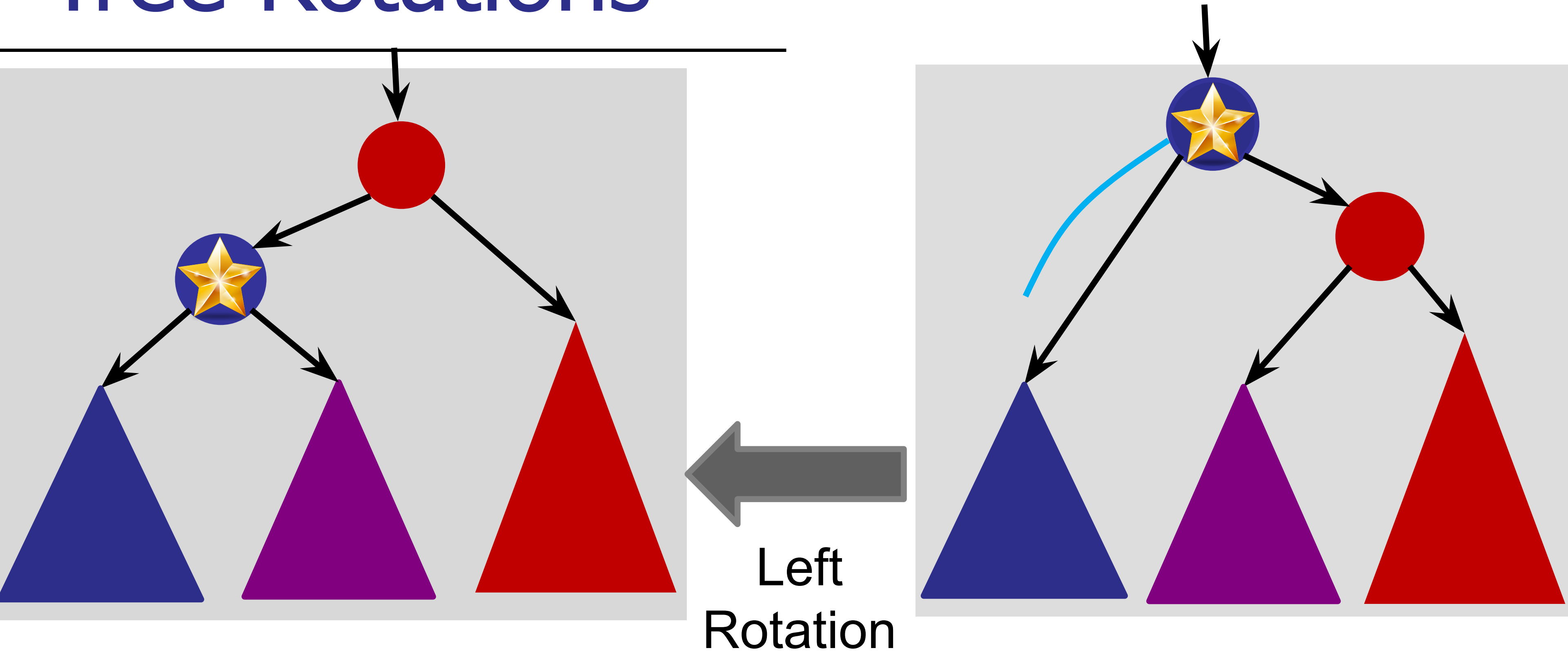


The way to remember it



The root of the subtree moves right

Tree Rotations

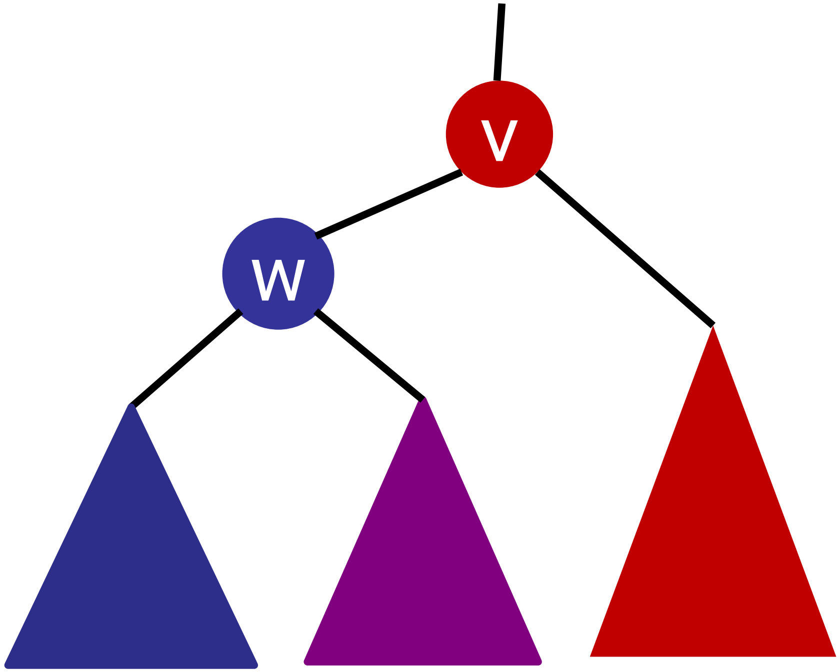


The root of the subtree moves left

Rotations

right-rotate(v) // assume v has left != null

w = v.left

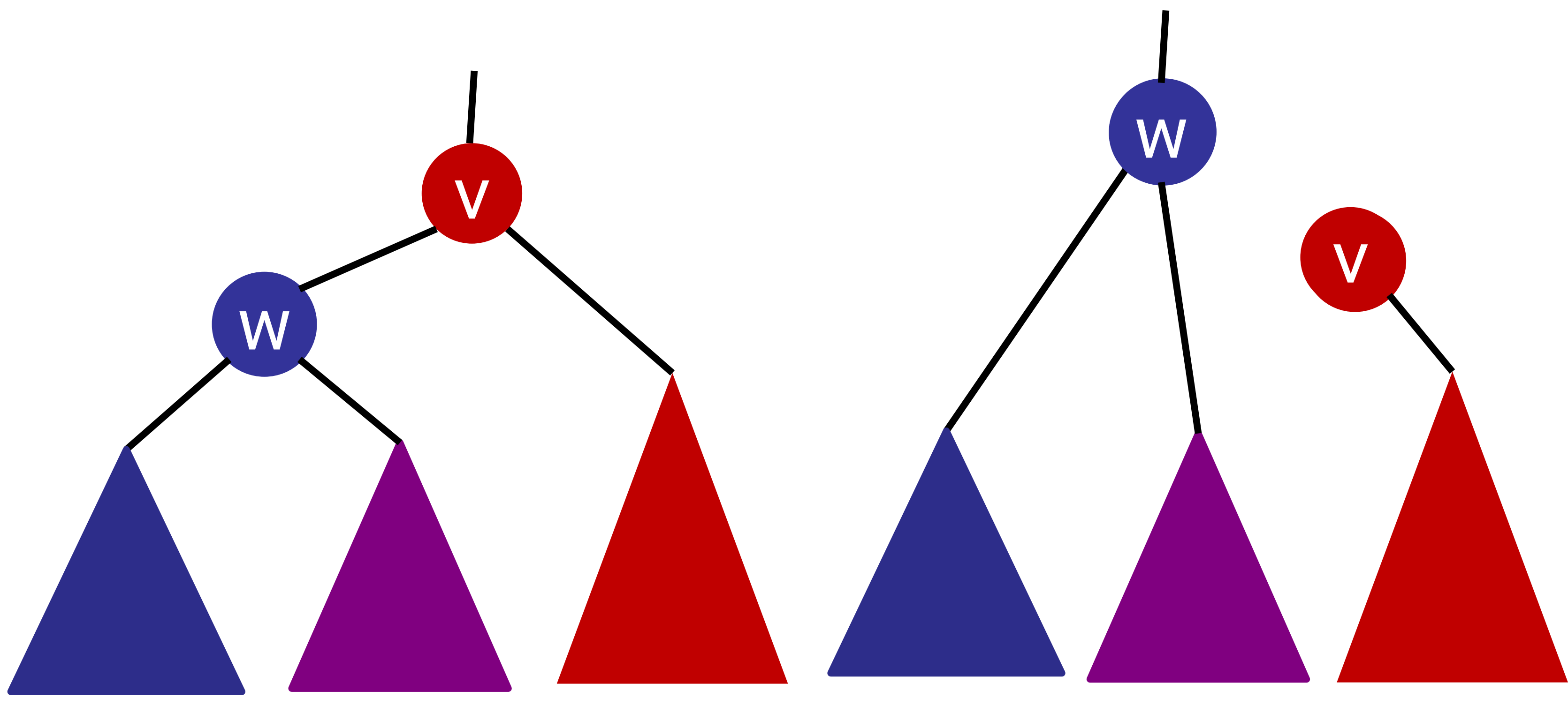


Rotations

right-rotate(v) // assume v has left != null

w = v.left

w.parent = v.parent



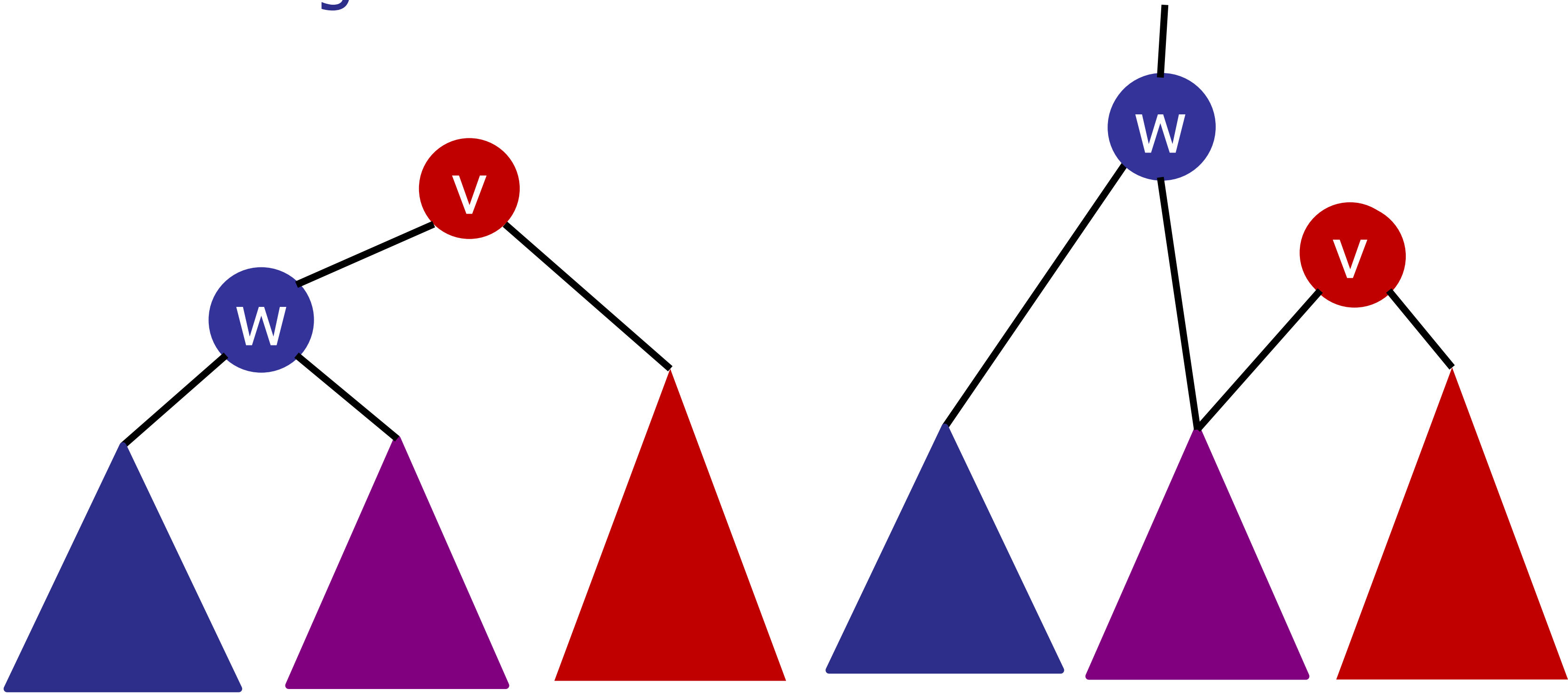
Rotations

right-rotate(v) // assume v has left != null

w = v.left

w.parent = v.parent

v.left = w.right



Rotations

right-rotate(v) // assume v has left != null

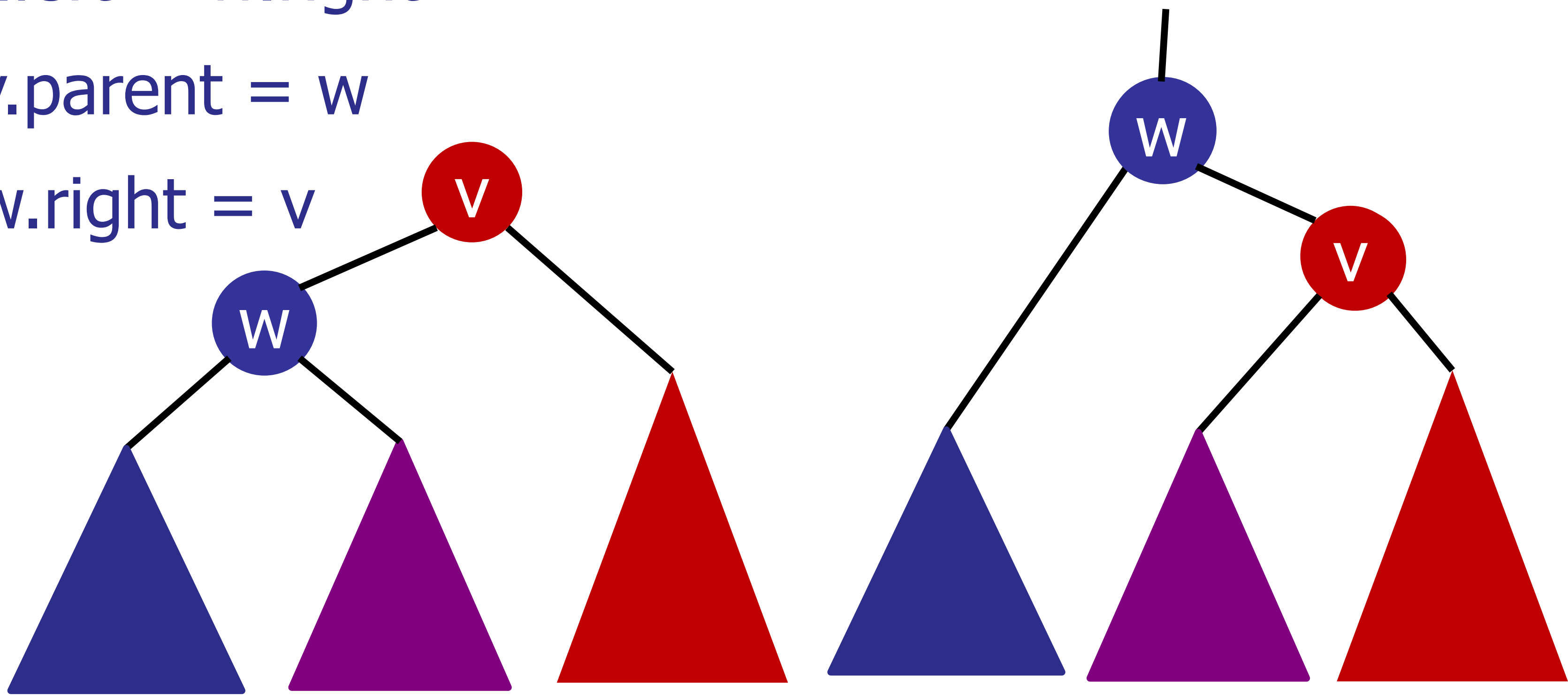
w = v.left

w.parent = v.parent

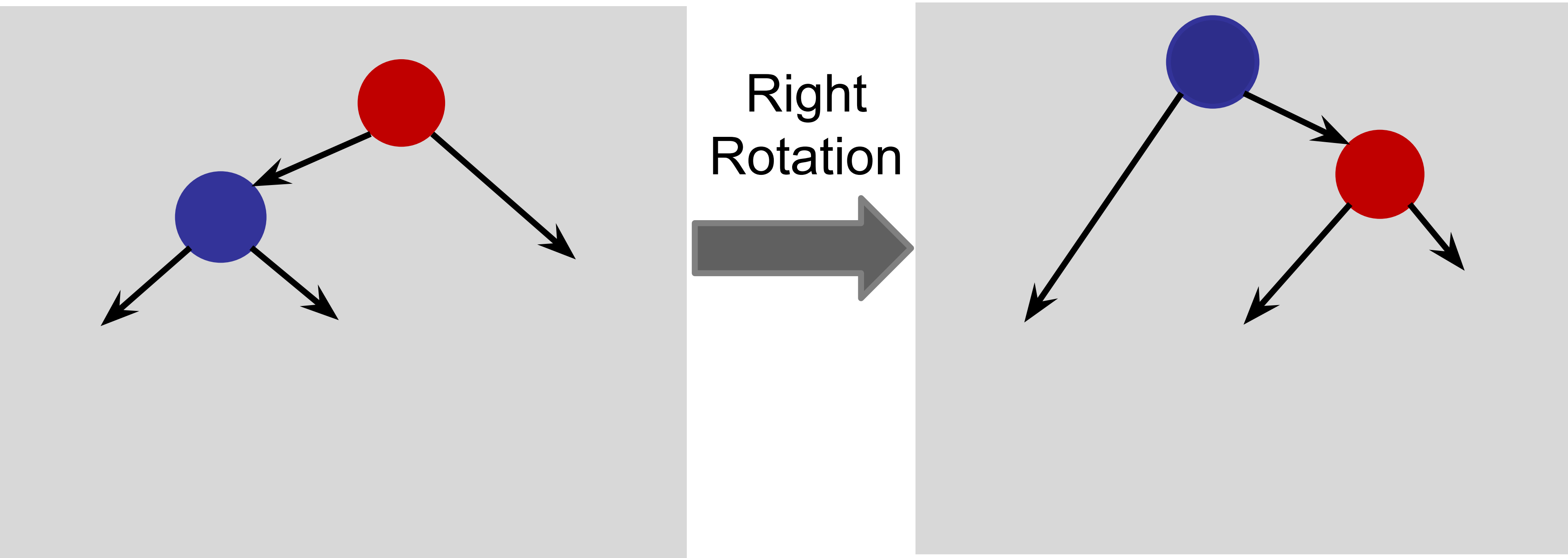
v.left = w.right

v.parent = w

w.right = v



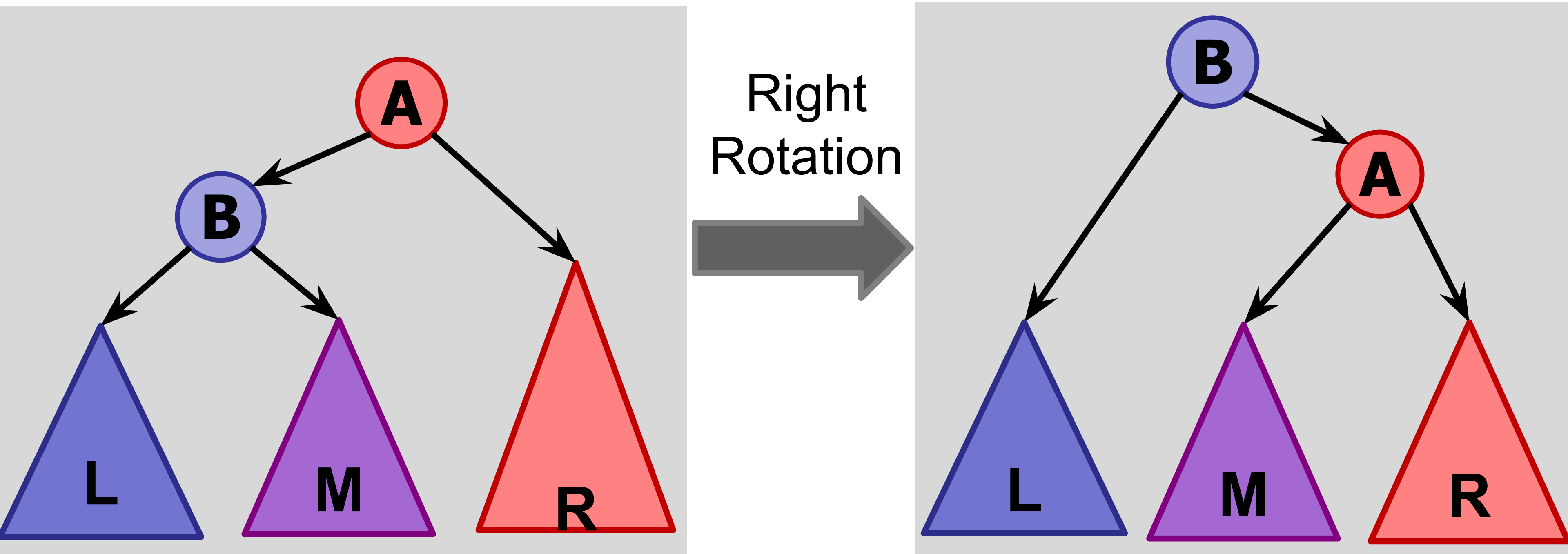
Tree Rotations



rotate-right requires a left child

rotate-left requires a right child

Tree Rotations



After insert:

Use tree rotations to restore balance.

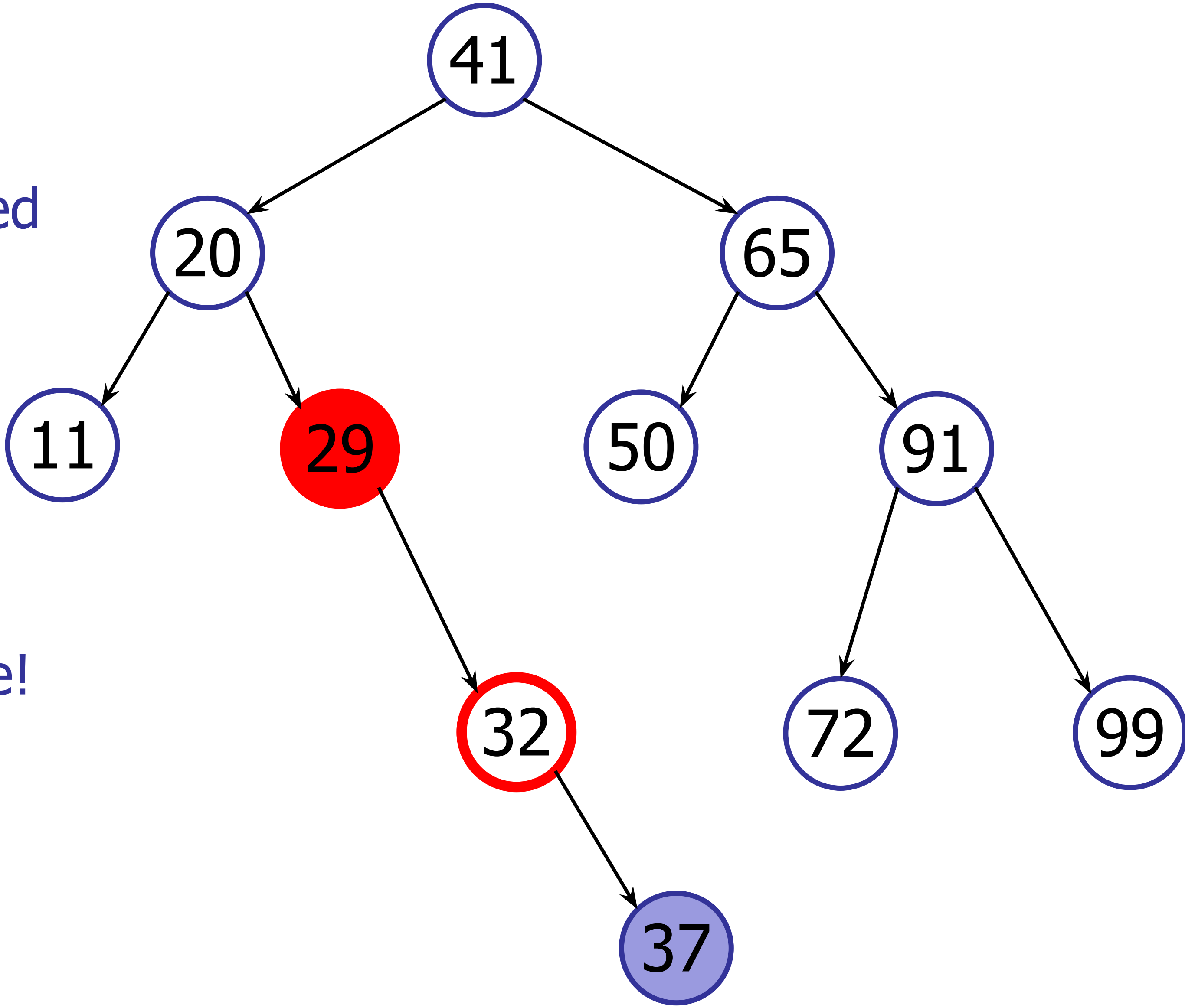
Height is out-of-balance by 1

Inserting in an AVL Tree

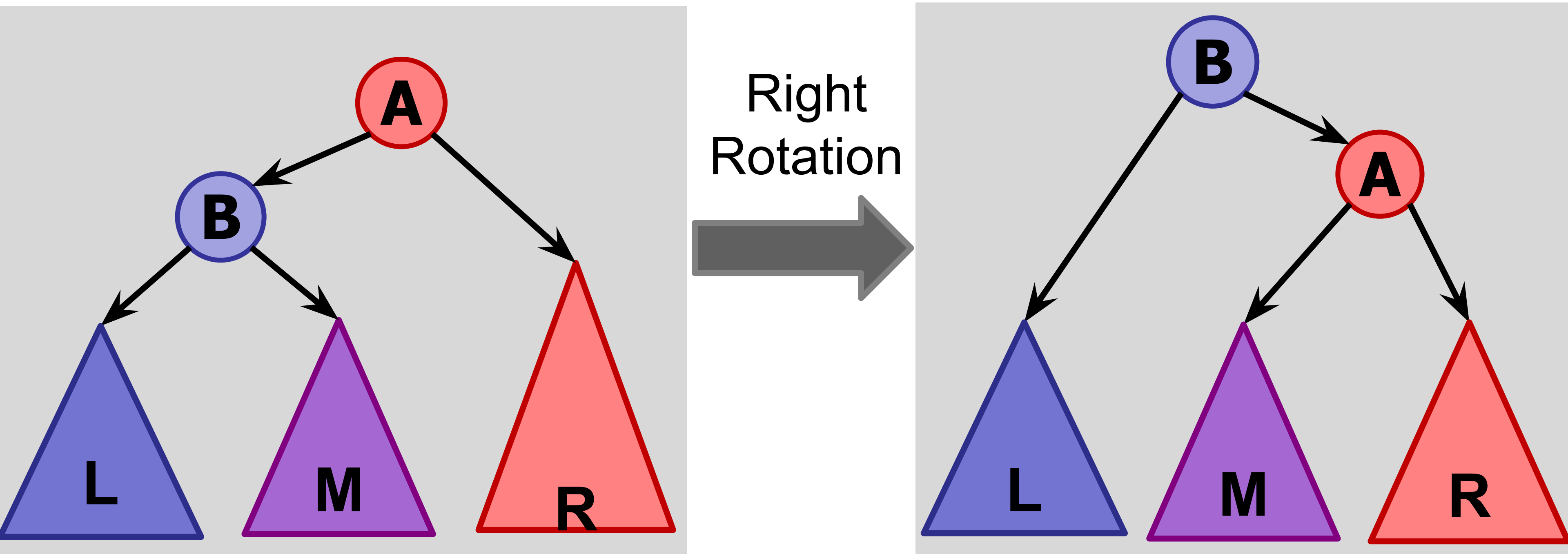
insert(37)

No longer balanced after insertion!

Need to rebalance!



Tree Rotations

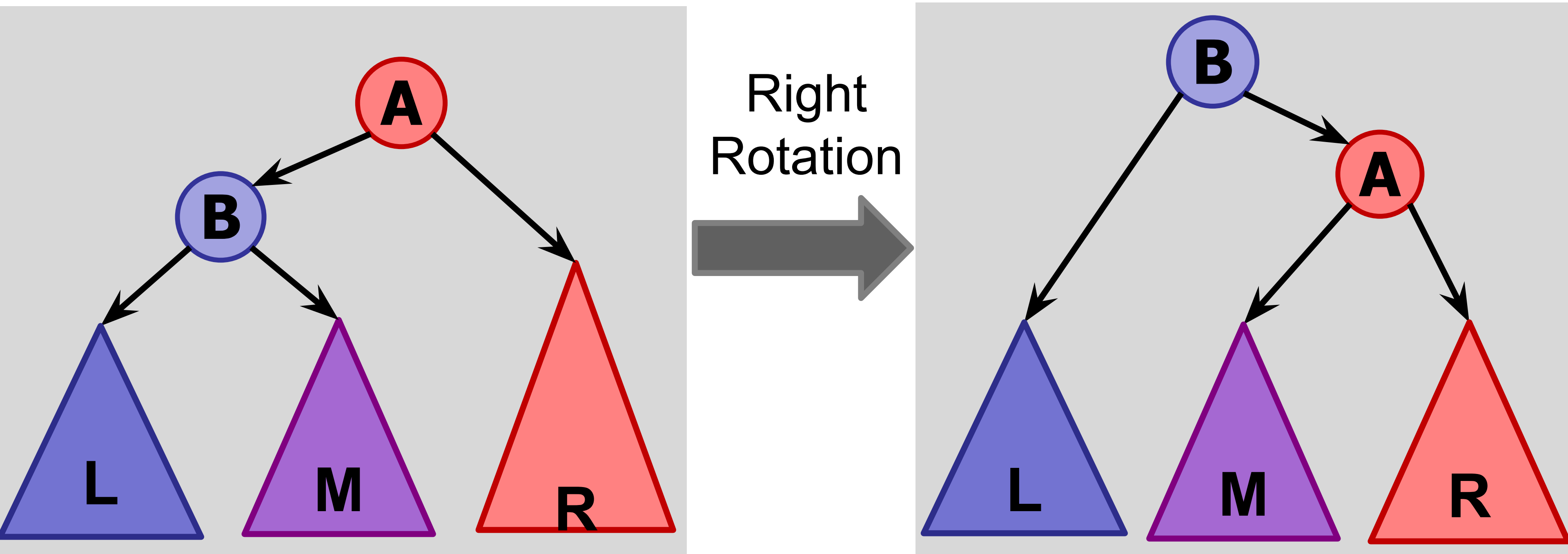


Use tree rotations to restore balance.

After insert, start at bottom, work your way up.

Assume subtree rooted at A is **LEFT-heavy**.

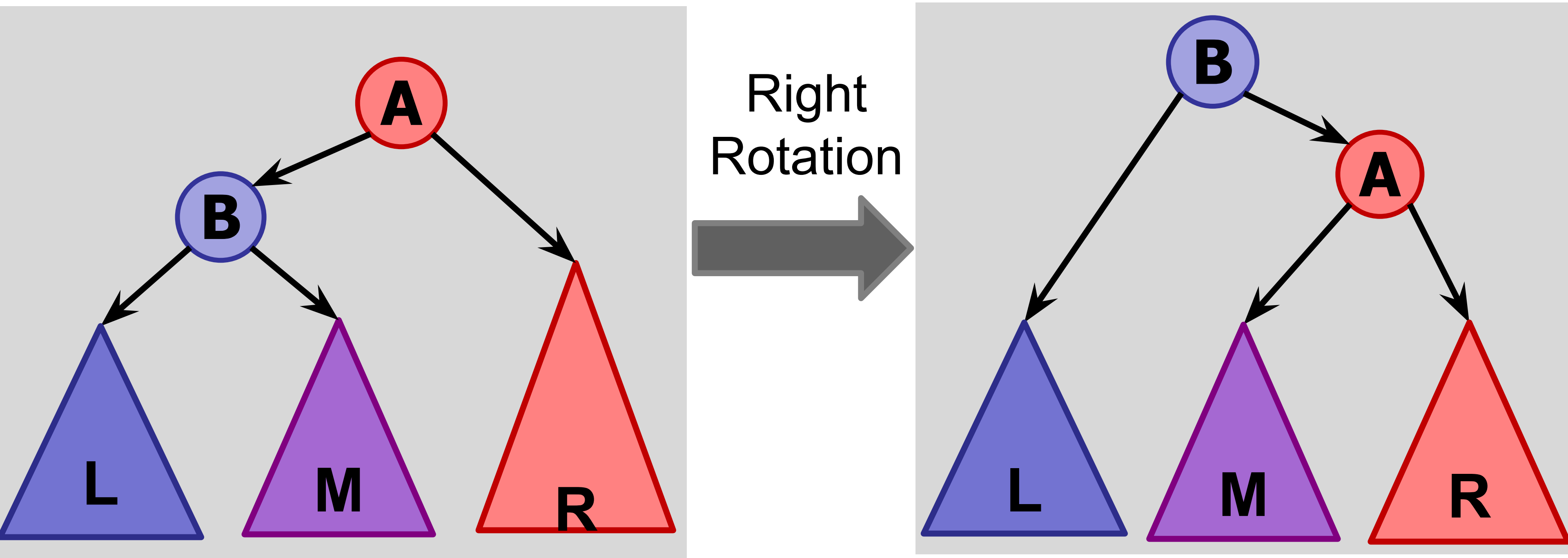
Tree Rotations



Assume subtree rooted at A is **LEFT-heavy**.

Left-heavy: Left subtree is taller than right subtree

Tree Rotations

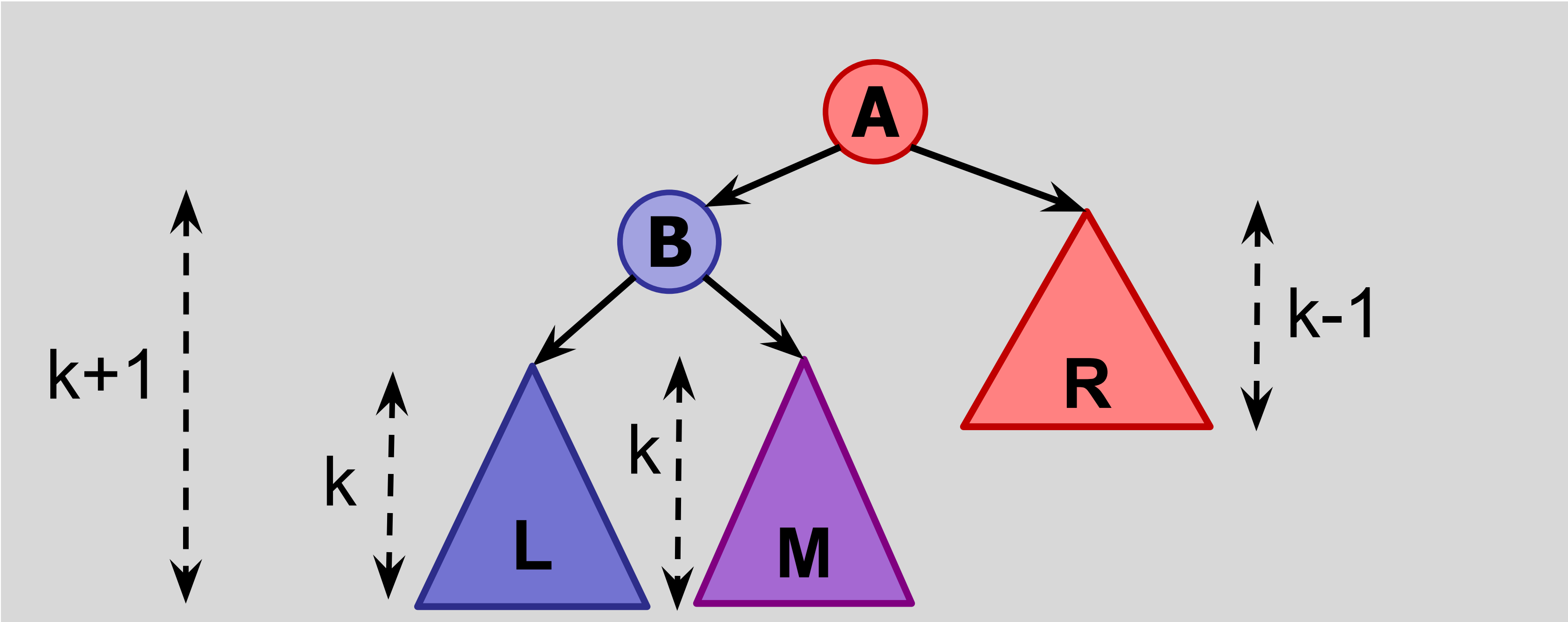


Assume subtree rooted at A is **LEFT-heavy**.

Left-heavy: Left subtree is taller than right subtree

3 cases: B is left-heavy, B is balanced, B is right-heavy

Tree Rotations (Left Heavy)

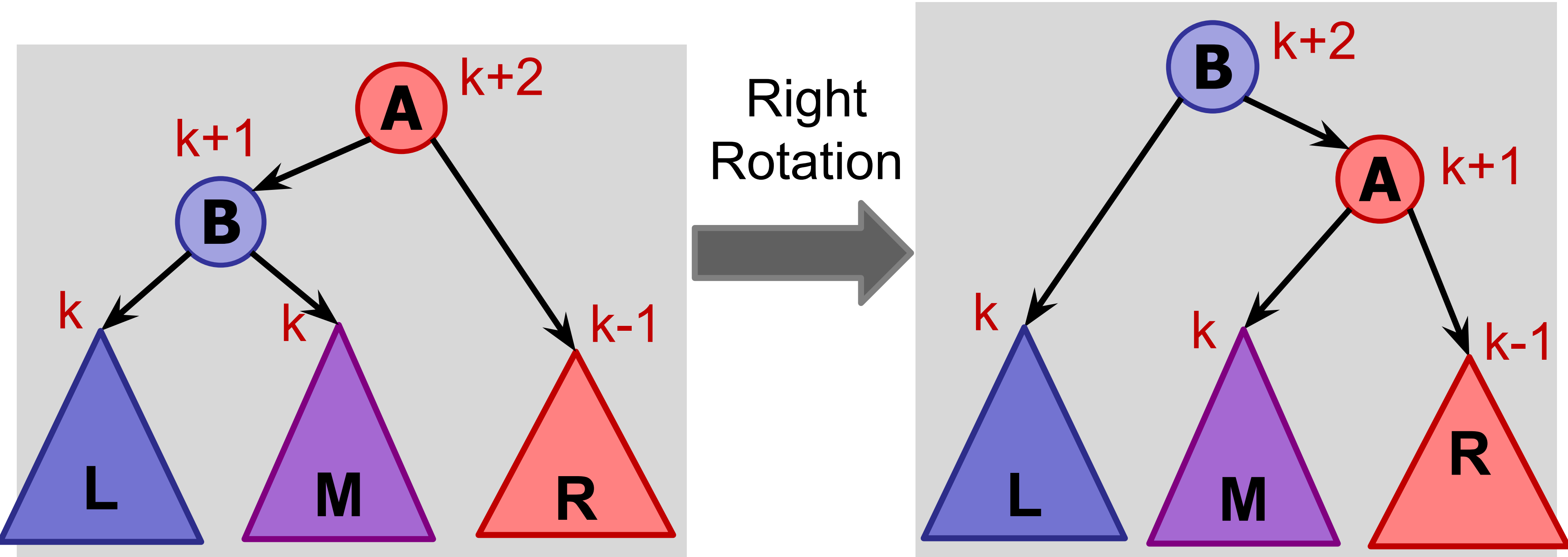


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

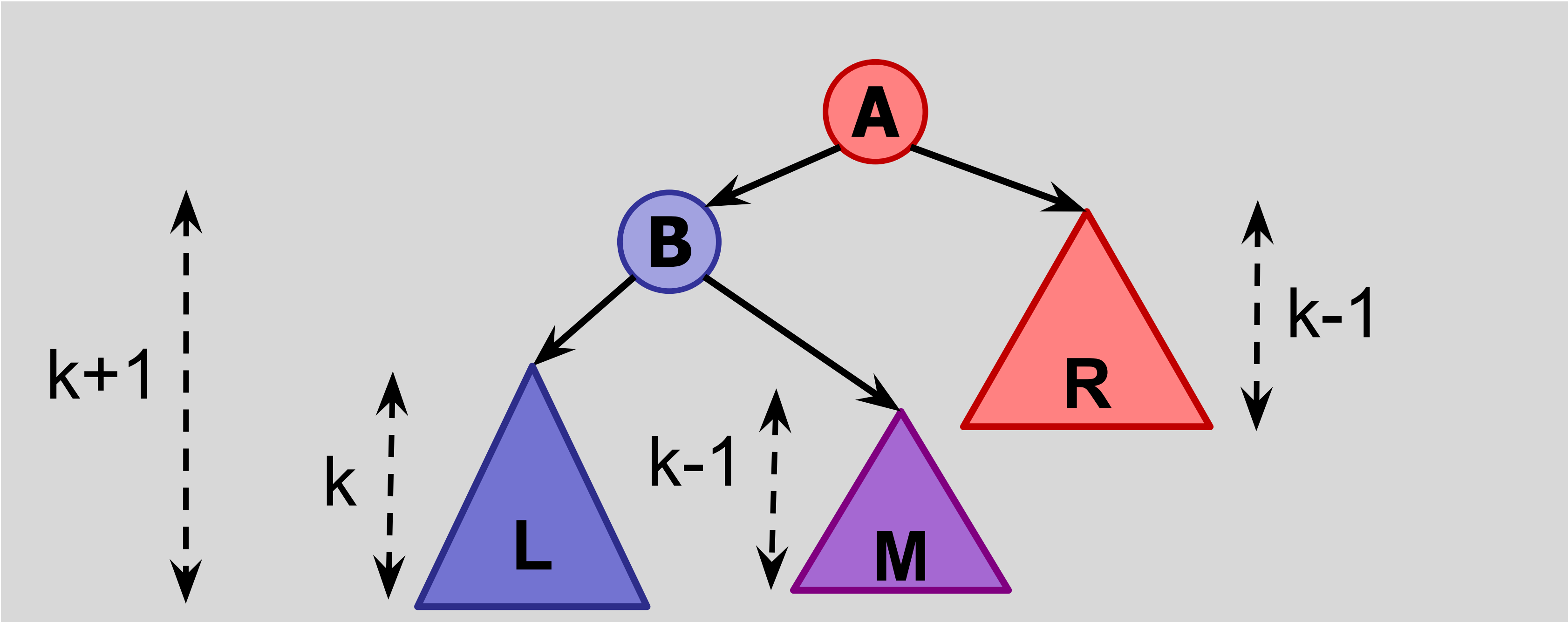


right-rotate:

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations (Left Heavy)

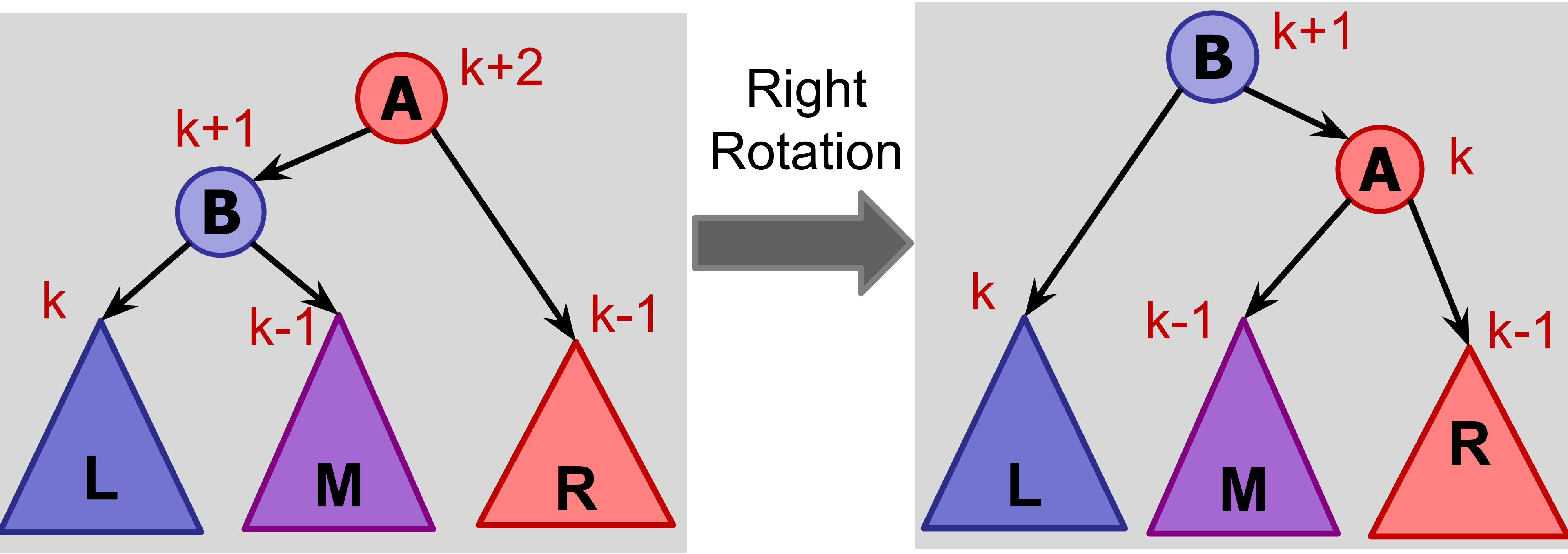


Assume **A** is the lowest node in the tree violating balance property.

Case 2: **B** is left heavy : h(**L**) = h(**M**) + 1

h(**R**) = h(**M**)

Tree Rotations

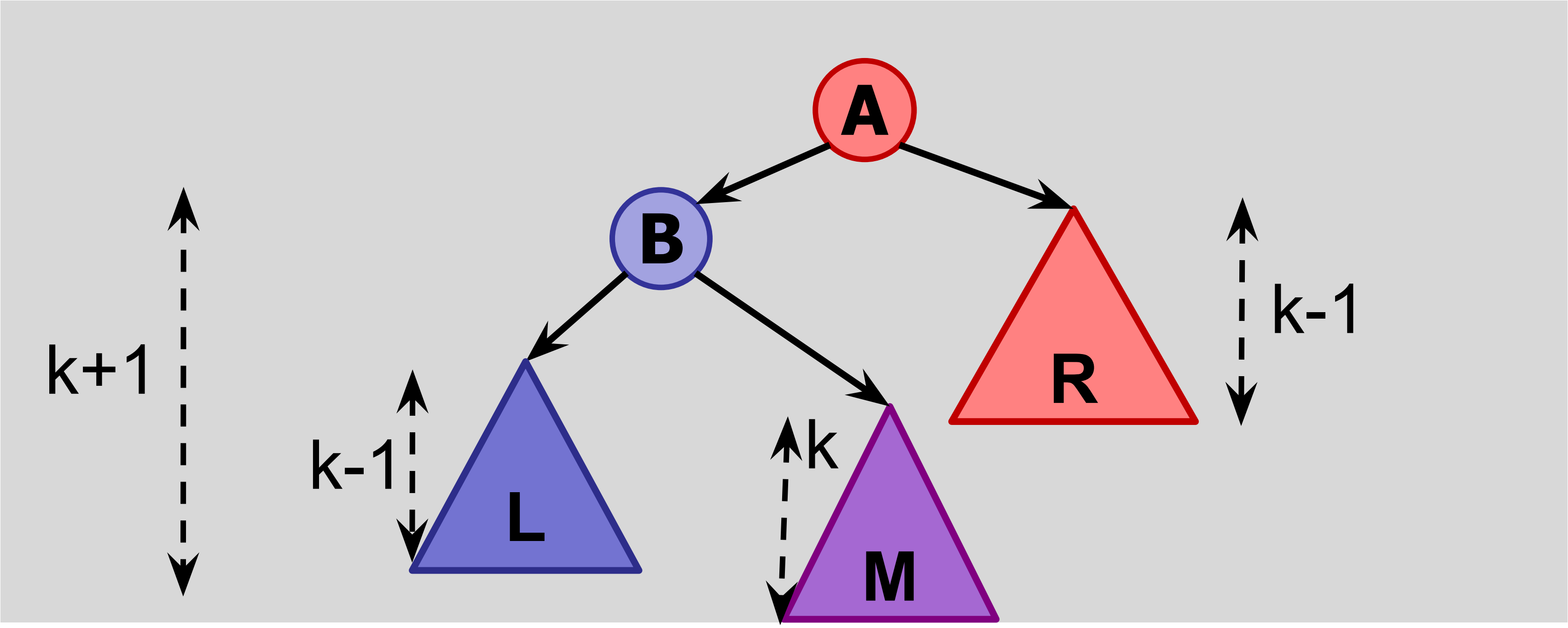


right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

h(**R**) = h(**M**)

Tree Rotations (Left Heavy)

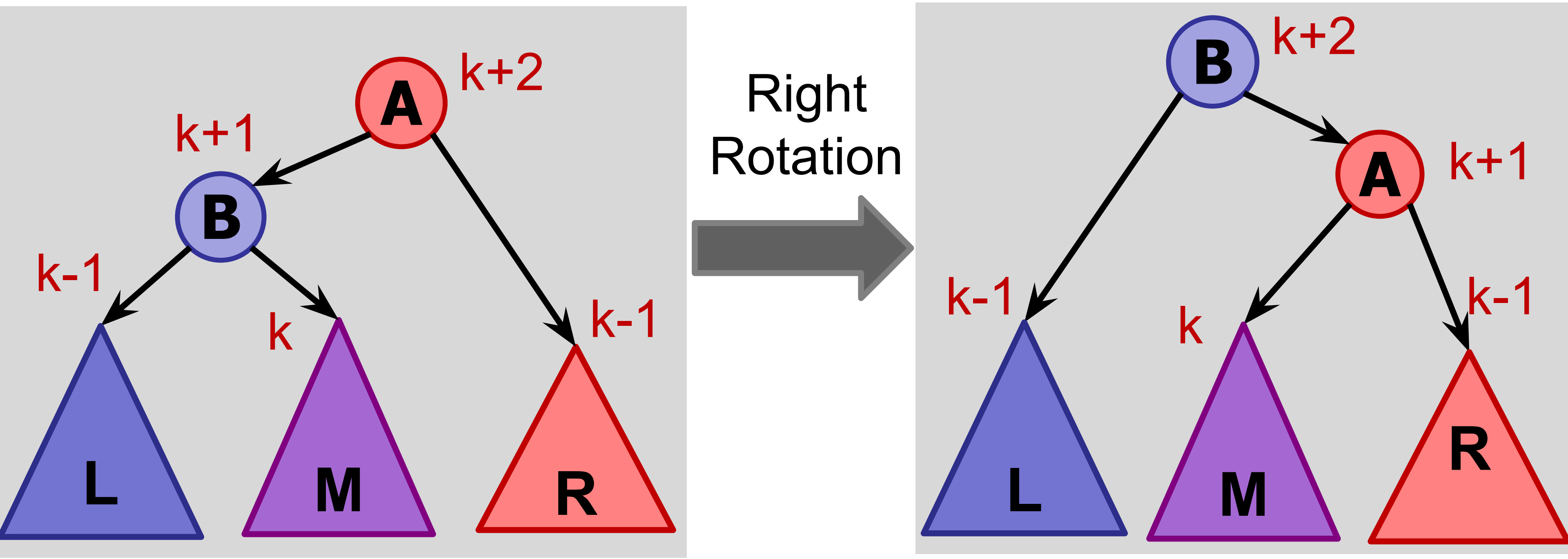


Assume **A** is the lowest node in the tree violating balance property.

Case 3: **B** is right heavy : h(**L**) = h(**M**) - 1

h(**R**) = h(**L**)

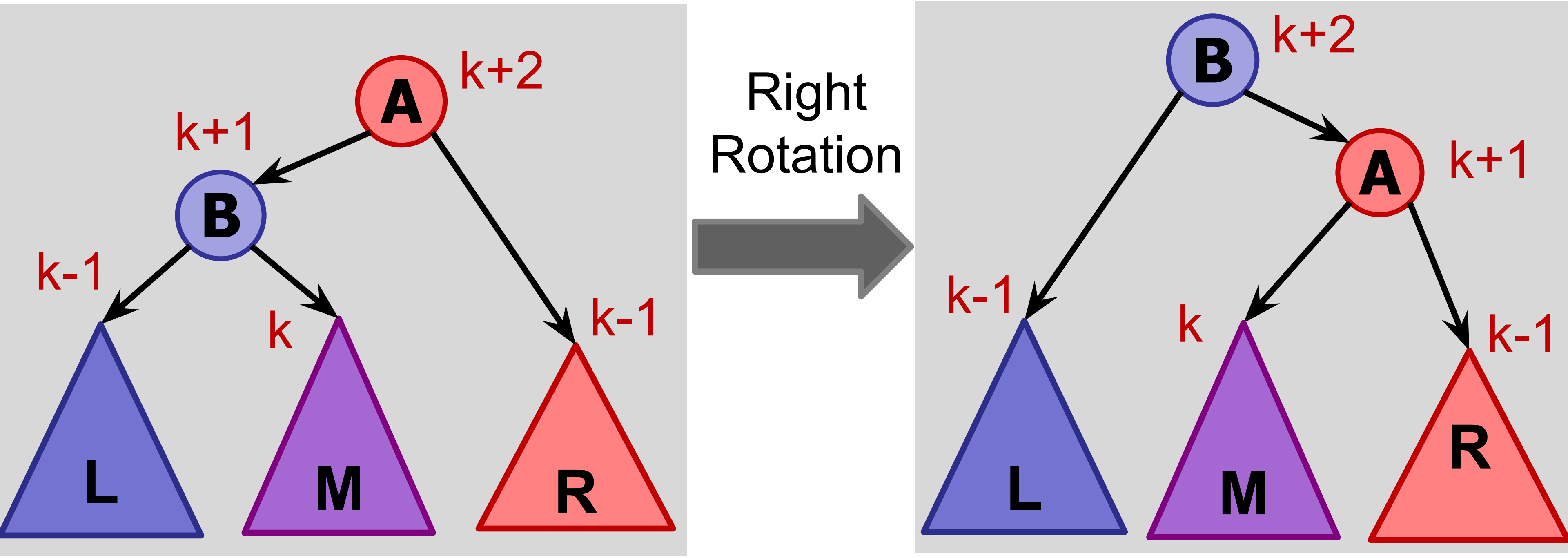
Tree Rotations



right-rotate:

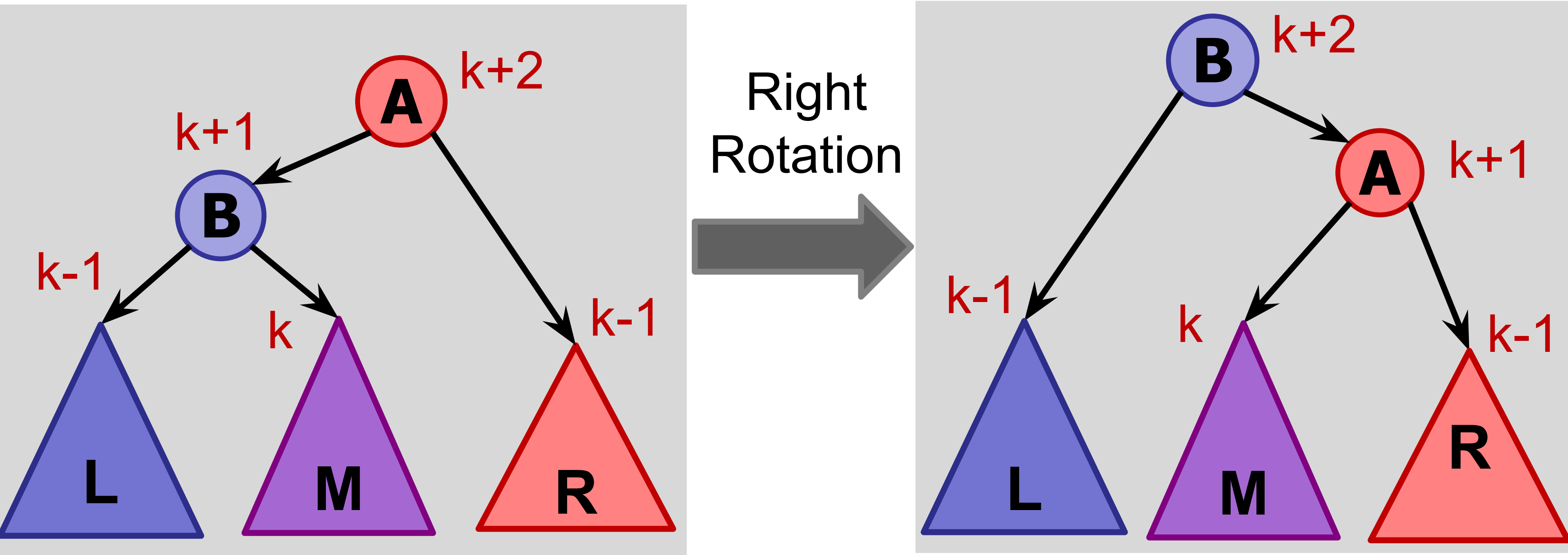
Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)



Are we done?

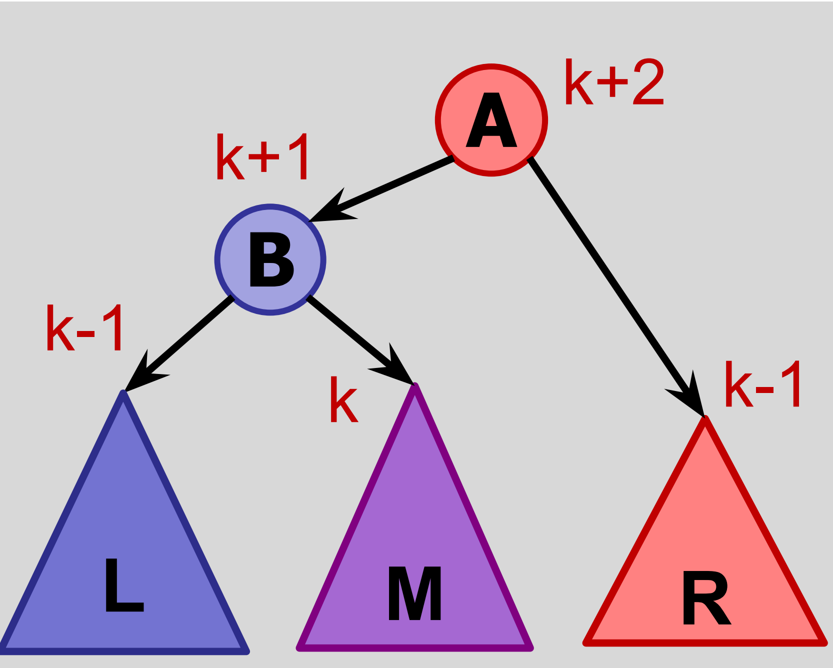
1. Yes.
2. No.
3. Maybe.



Are we done?

1. Yes.
2. No.
3. Maybe.

Tree Rotations



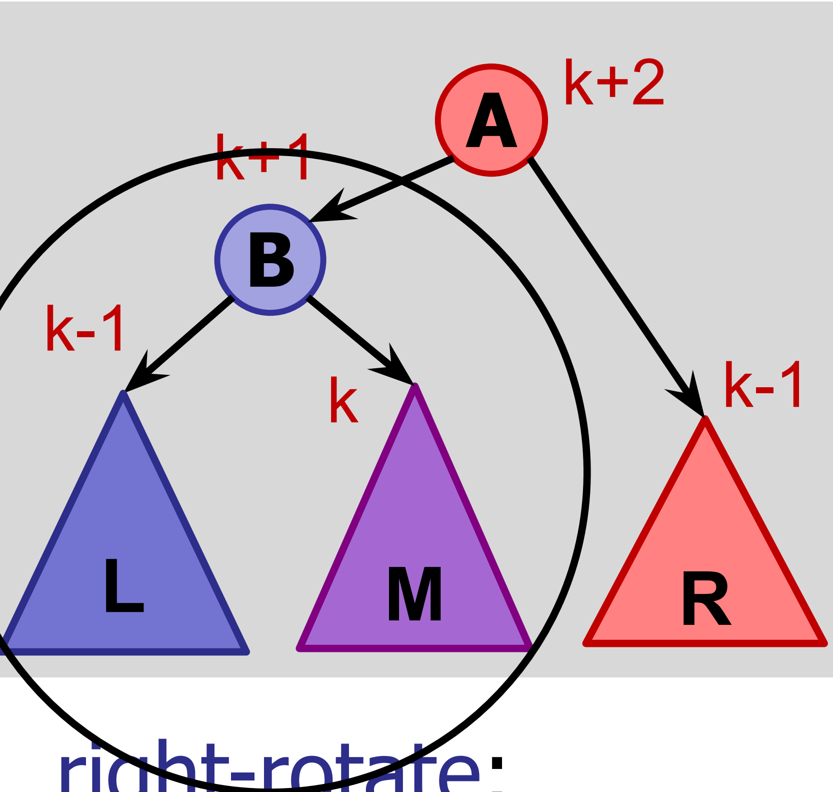
Let's do something first before we right-rotate(A)

right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)

Tree Rotations



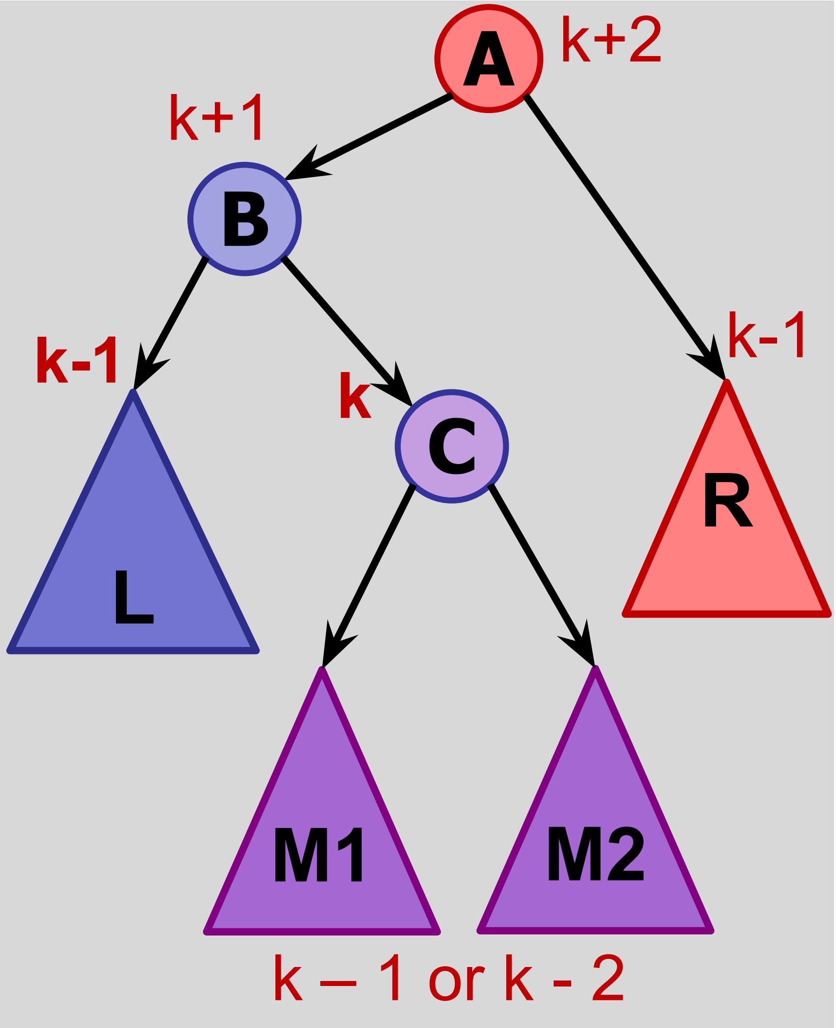
Let's do something first before we right-rotate(A)

right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

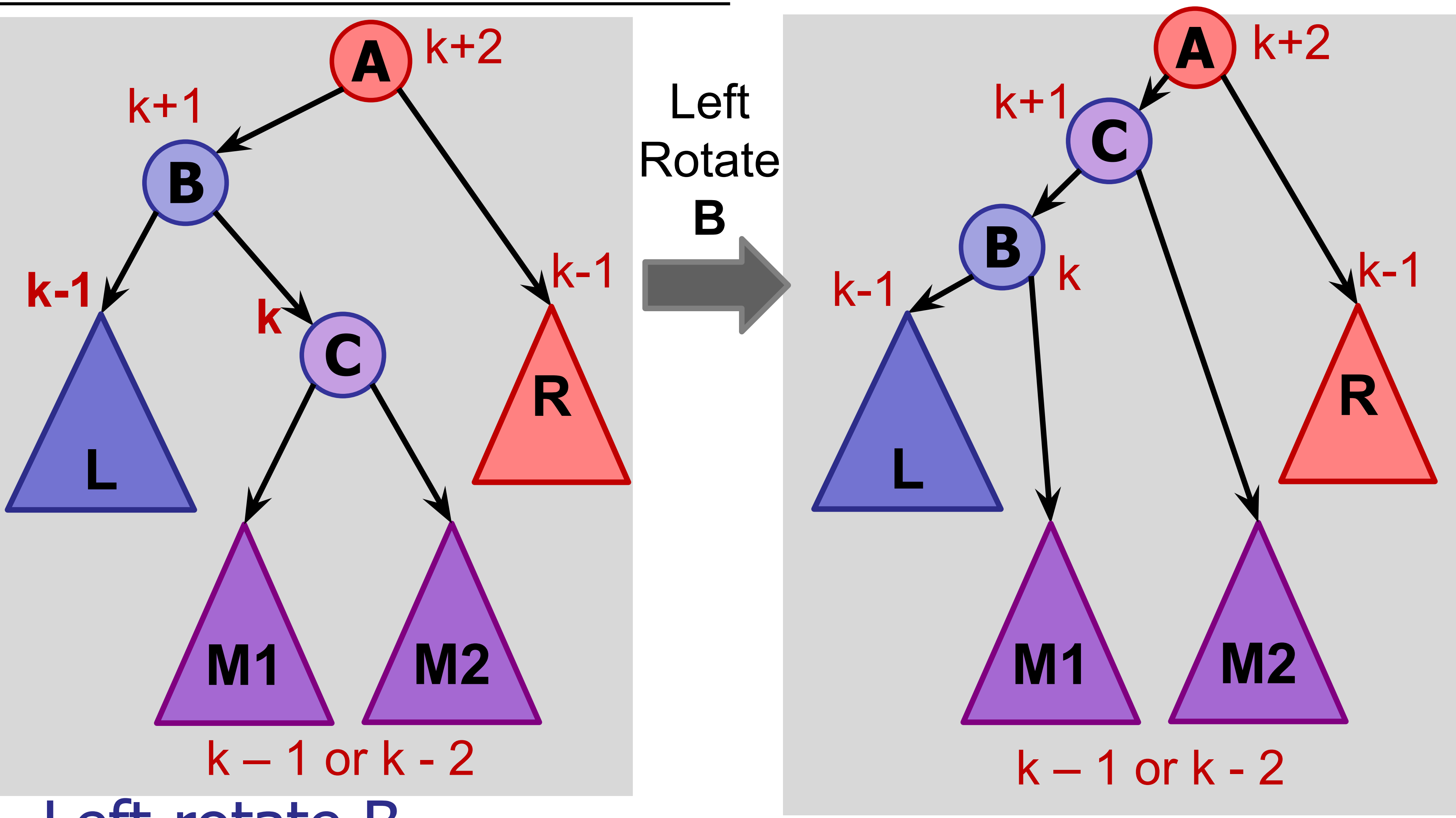
h(**R**) = h(**L**)

Tree Rotations



Left-rotate B

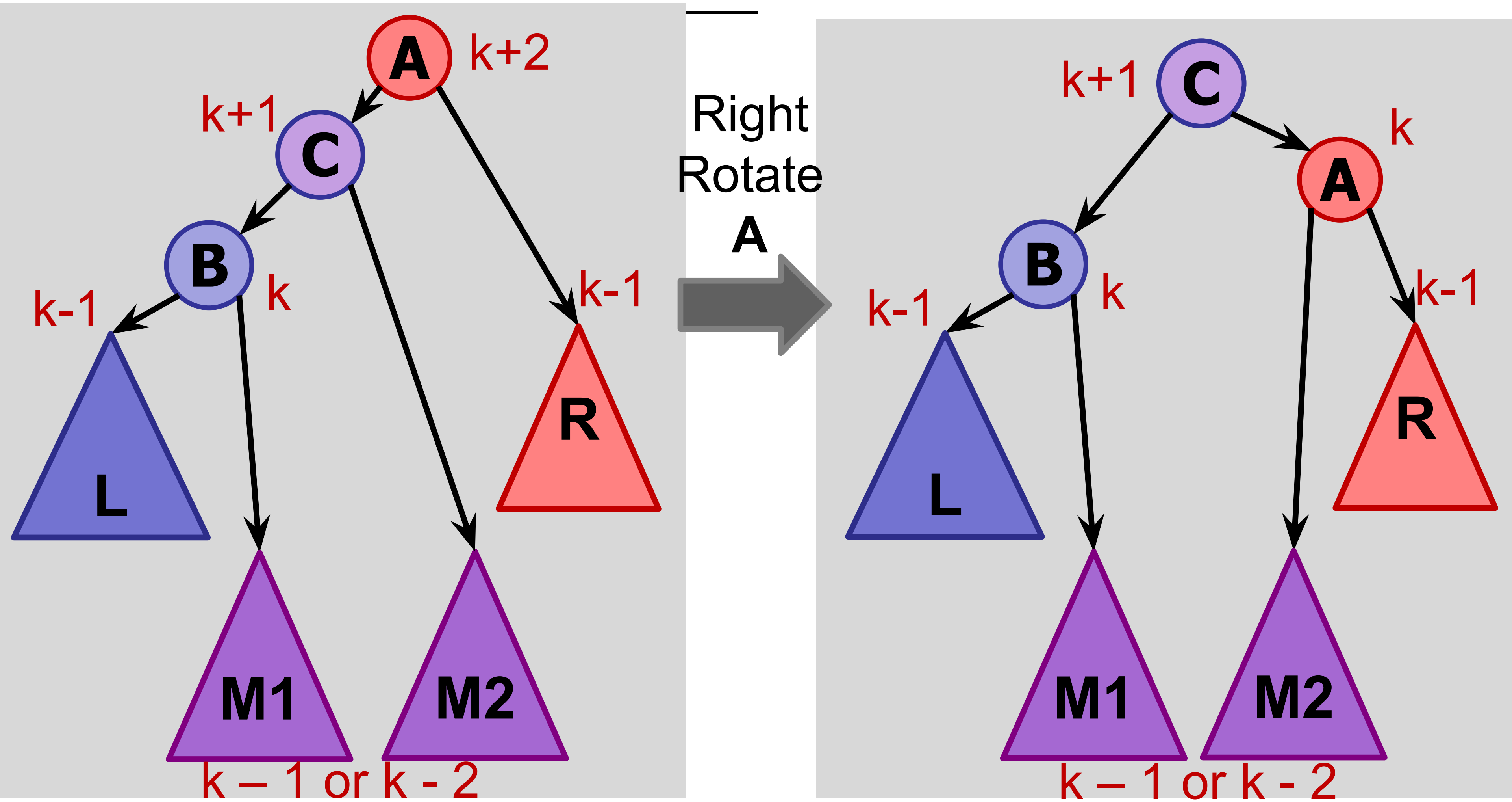
Tree Rotations



Left-rotate B

After left-rotate B: **A** and **C** still out of balance.

Tree Rotations



After right-rotate A: all in balance.

Rotations

Summary:

If v is out of balance and left heavy:

1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

Symmetric three cases….

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

<u>Question:</u>

Why isn’t it 2log(n)?

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

We can actually bound it by 2

Insert in AVL Tree

Summary:

- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.
  - Then we are done

Insert in AVL Tree

Summary:

- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.
  - Then we are done

Note: only need to perform two rotations

- Why?
- In cases 2, 3: reduce height of sub-tree by 1
- Case 3: Next week

Today and Next Week

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations