CS2040S
Data Structures and Algorithms

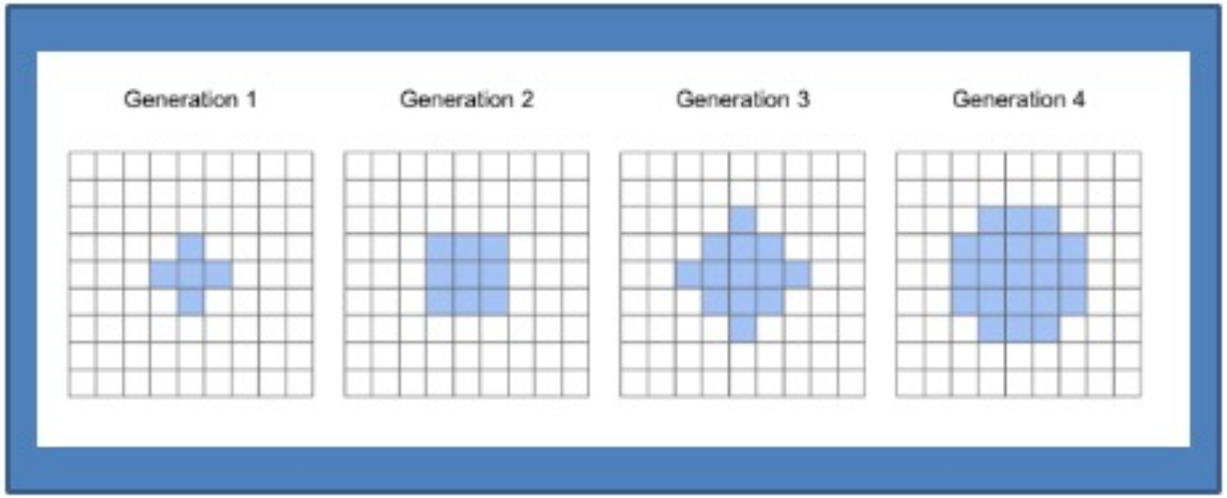
Puzzle of the Week: Squares

(Courtesy: Riddler)

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)?



Plan of the Week

QuickSort & QuickSelect

- (Paranoid) QuickSort Analysis
- QuickSelect

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

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

Announcements

Midterm : Monday March 6, 4pm (class time)

Location: MPSH 2A & 2B

Note: In person, pen and paper

Nota Bene: Please mark your calendar now.

Plan of the Week

QuickSort & QuickSelect

- (Paranoid) QuickSort Analysis
- QuickSelect

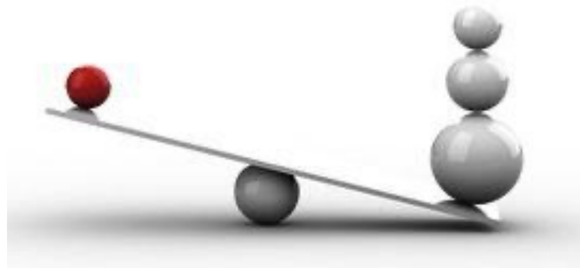
Trees

- Terminology
- Traversals
- Operations

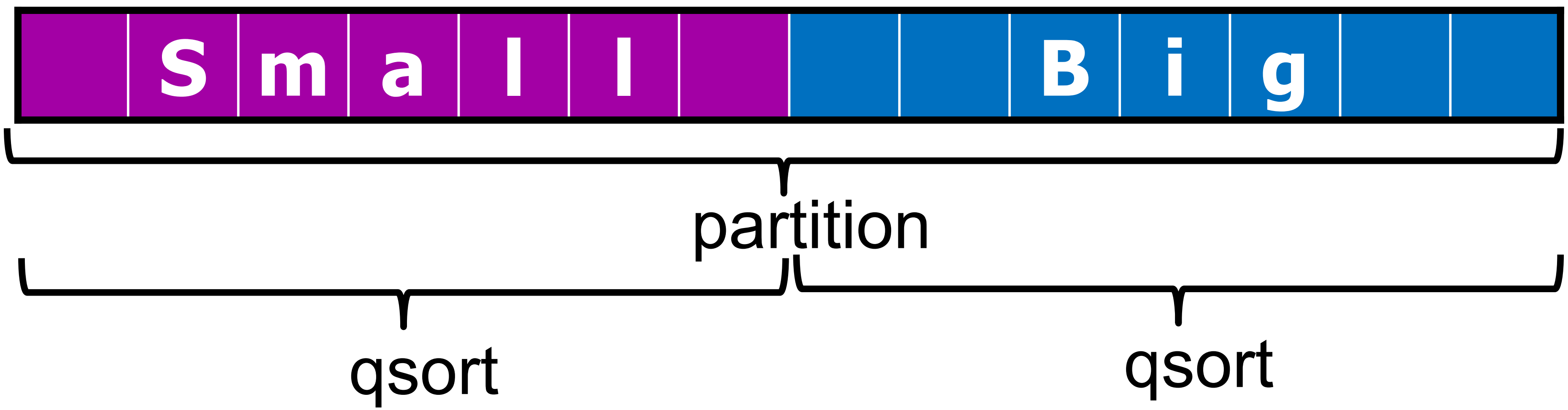
Balanced Trees

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

QuickSort



```
QuickSort(A[1..n], n)
    if (n==1) then return;
    else
        p = partition(A[1..n], n)
        x = QuickSort(A[1..p-1], p-1)
        y = QuickSort(A[p+1..n], n-p)
```



QuickSort

How to partition efficiently?

- Duplicates

  If you ignore duplicates, partitioning can be very slow!

- In-place partitioning

  One pass, use invariants to ensure correctness.

- Stability

  In-place partitioning isn't stable.

- Choosing a pivot (randomization)

  Deterministic pivots are generally bad.

  Randomization is an easy way to find a good pivot!

Choice of Pivot

Options:

- first element: A[1]
- last element: A[n]
- middle element: A[n/2]
- median of (A[1], A[n/2], A[n])

In the worst case, it does not matter!

All options are equally bad.

QuickSort Summary

- If we choose the pivot as A[1]:
  - Bad performance: $\Omega(n^2)$
- If we could choose the median element:
  - Good performance: $O(n \log n)$
- If we could split the array (1/10) : (9/10)
  - Good performance: $O(n \log n)$

QuickSort

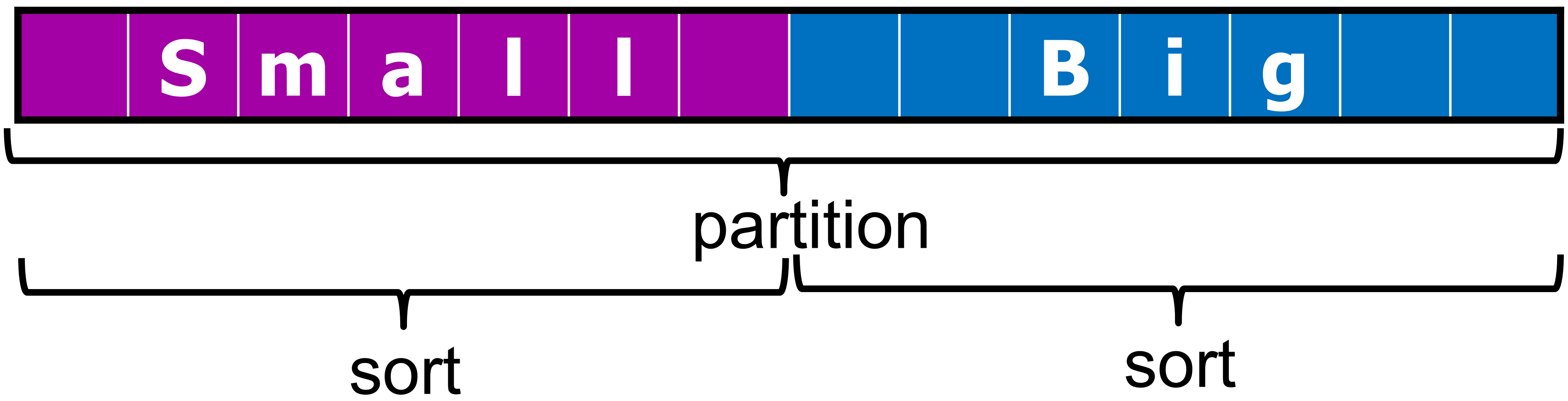
Key Idea:

- Choose the pivot at random.
- Most of the time: split will be at least $\frac{9}{10} : \frac{1}{10}$

Randomized Algorithms:

- Algorithm makes decision based on random coin flips.
- Can "fool" the adversary (who provides bad input)
- Running time is a *random variable*.

```
QuickSort(A[1..n], n)
    if (n == 1) then return;
    else
      pIndex = random(1, n)
      p = 3WayPartition(A[1..n], n, pindex)
      x = QuickSort(A[1..p-1], p-1)
      y = QuickSort(A[p+1..n], n-p)
```



Paranoid QuickSort

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
            pIndex = random(1, n)
            p = partition(A[1..n], n, pIndex)
        until p > (1/10)n and p < (9/10)n

        x = QuickSort(A[1..p–1], p–1)
        y = QuickSort(A[p+1..n], n–p)
```

Paranoid QuickSort

T(n) →

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
            pIndex = random(1,
            p = partition(A[1..n],
        until p > (1/10)n and p < (

        x = QuickSort(A[1..p–1], p
        y = QuickSort(A[p+1..n], n
```

T(n)

1

O(n)

T(k)

T(n-k)

9n/10 > k > n/10

Paranoid QuickSort

Key claim:

– We only execute the **repeat** loop O(1) times (in expectation).

Then we know:

$$T(n) <= T(n/10) + T(9n/10) + n(\text{\# iterations of } \textbf{repeat})$$
$$= O(n \log n)$$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = $(1 - p)$

How many flips to get at least one head?

If $p = ½$, the expected number of flips to get one head equals:

$$\mathbf{E}[X] = 1/p = 1/½ = 2$$

Paranoid QuickSort

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
                pIndex = random(1, n)
                p = partition(A[1..n], n, pIndex)
        until p > (1/10)n and p < (9/10)

        x = QuickSort(A[1..p–1], p–1)
        y = QuickSort(A[p+1..n], n–p)
```

How many repetitions?

If we choose a pivot at random, what is the probability that it is good?

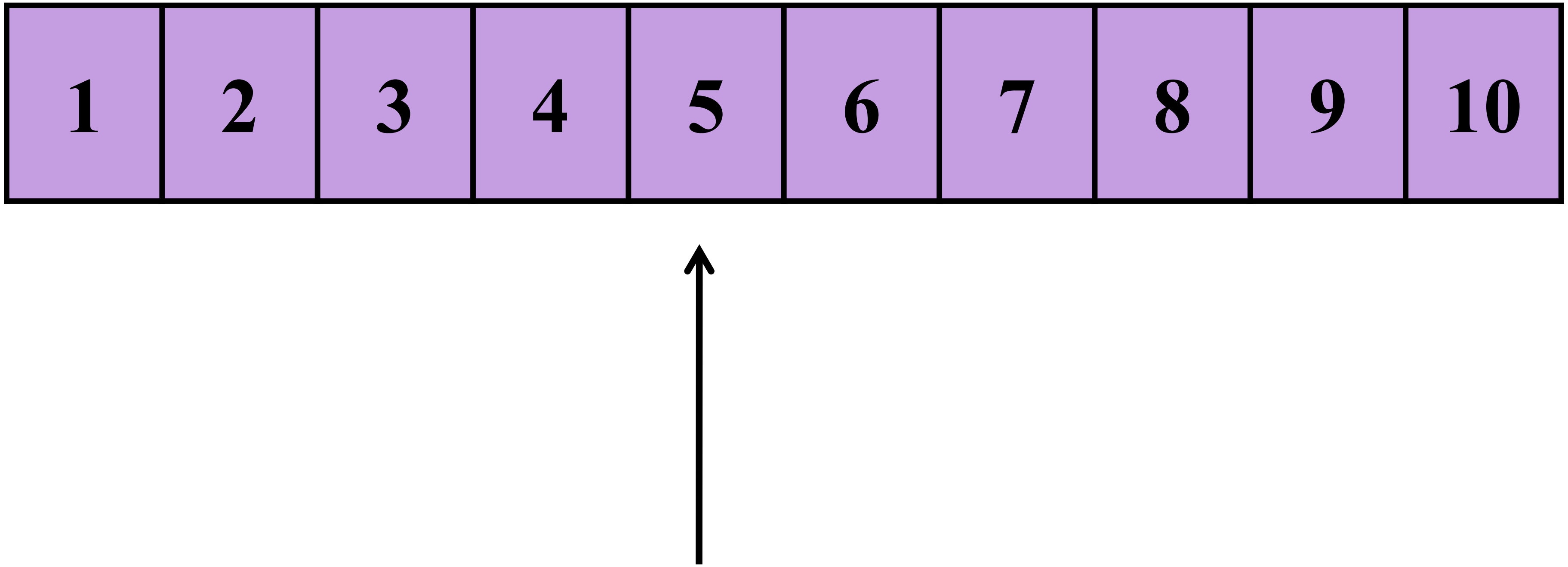
1. 1/10
2. 2/10
3. 8/10
4. 1/log(n)
5. 1/n
6. I have no idea.

ARCHIPELAGO

is open

Choosing a Good Pivot

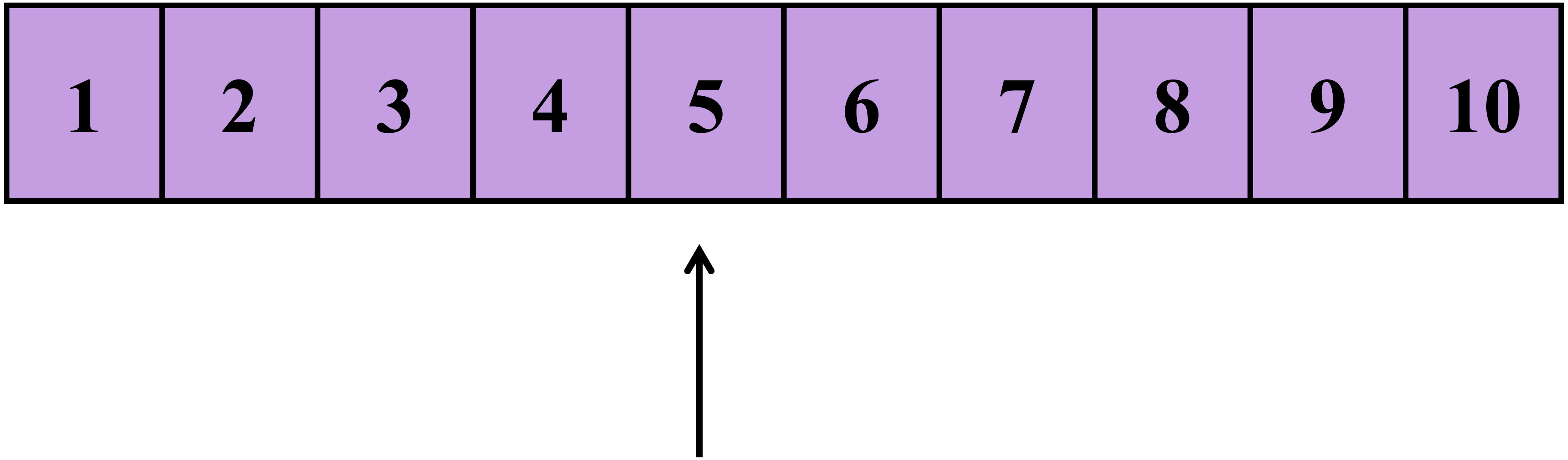
Imagine the array divided into 10 pieces:



Choose a random point at which to partition.

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

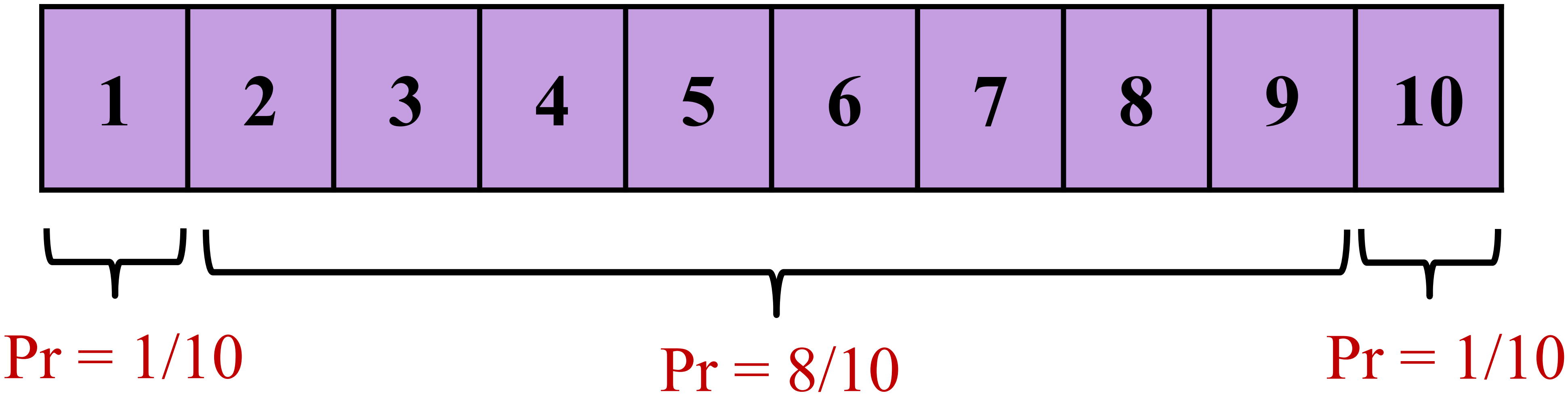


Choose a random point at which to partition.

- 10 possible events
- each occurs with probability 1/10

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

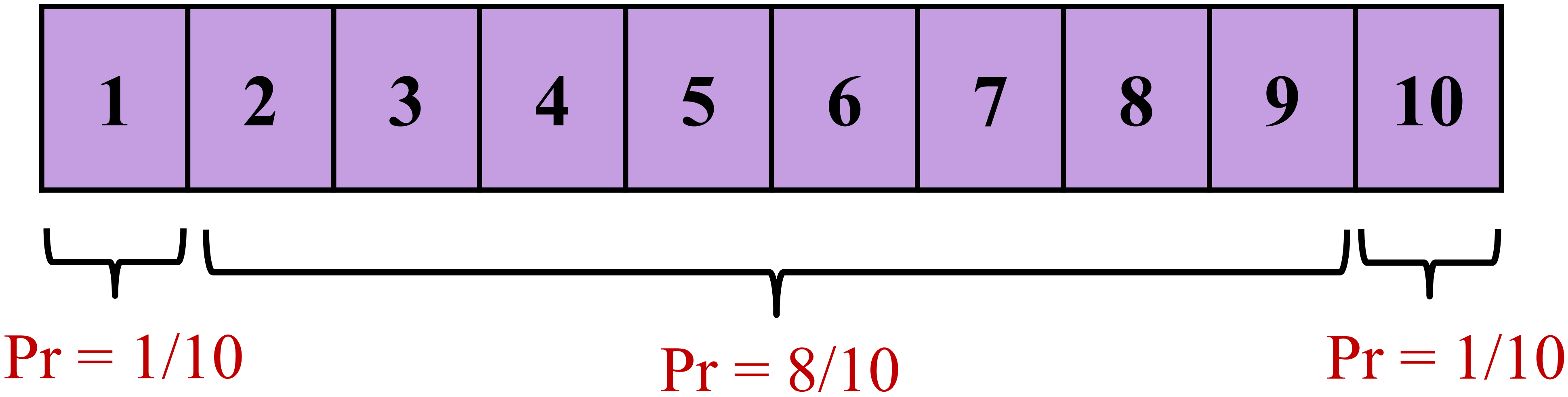


Choose a random point at which to partition.

- 10 possible events
- each occurs with probability 1/10

Choosing a Good Pivot

Imagine the array divided into 10 pieces:



Probability of a good pivot:

$p = 8/10$

$(1 - p) = 2/10$

Choosing a Good Pivot

Probability of a good pivot:

$$p = 8/10$$

$$(1 - p) = 2/10$$

Expected number of times to repeatedly choose a pivot to achieve a good pivot:

$$\mathbf{E}[\#\text{ choices}] = 1/p = 10/8 < 2$$

Paranoid QuickSort

QuickSort(A[1..n], n)

if (n==1) **then** return;

else

repeat

$pIndex$ = **random**(1, n)

p = partition(A[1..n], n, $pIndex$)

until $p > n/10$ **and** $p < n(9/10)$

How many repetitions?

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

Paranoid QuickSort

Key claim:

We only execute the **repeat** loop < 2 times

(in expectation).

Then we know:

$$\mathbf{E}[\mathrm{T}(n)] = \mathbf{E}[\mathrm{T}(k)] + \mathbf{E}[\mathrm{T}(n-k)] + \mathbf{E}[\#\text{ pivot choices}](n)$$

$$<= \mathbf{E}[\mathrm{T}(k)] + \mathrm{E}[\mathrm{T}(n-k)] + 2n$$

$$= \mathrm{O}(n \log n)$$

Regular QuickSort

Also true:

Expected running time is O(n log n).

With high probability, running time is O(n log n).

QuickSort

How to analyze?

- Divide-and-conquer recurrence

  It is sufficient to do a 90:10 split to get O(n log n) performance!

- What is the probability that a random pivot yields a 90:10 split? Quite good!
- Simplification: Paranoid QuickSort
- How many repetitions to find a good pivot, in expectation? O(1)
- Solve using recurrence: Linearity of expectation FTW.

QuickSort Optimization:

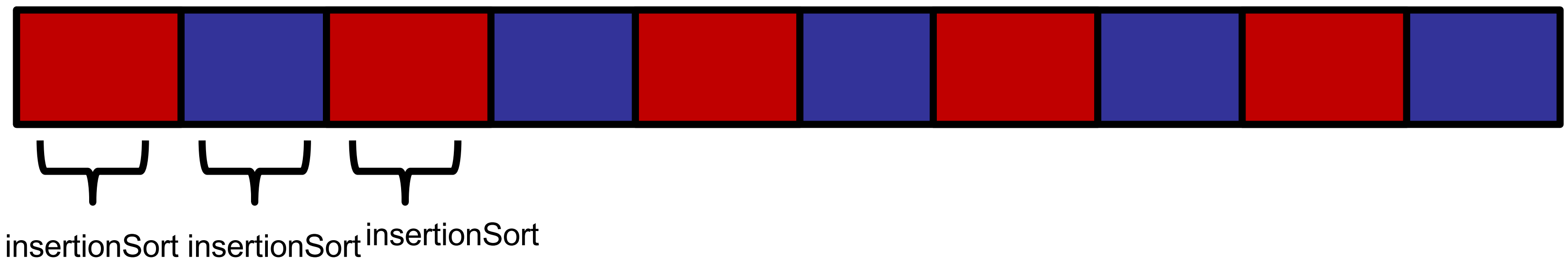
Base case?

1. Recurse all the way to single-element arrays.

QuickSort Optimization:

Base case?

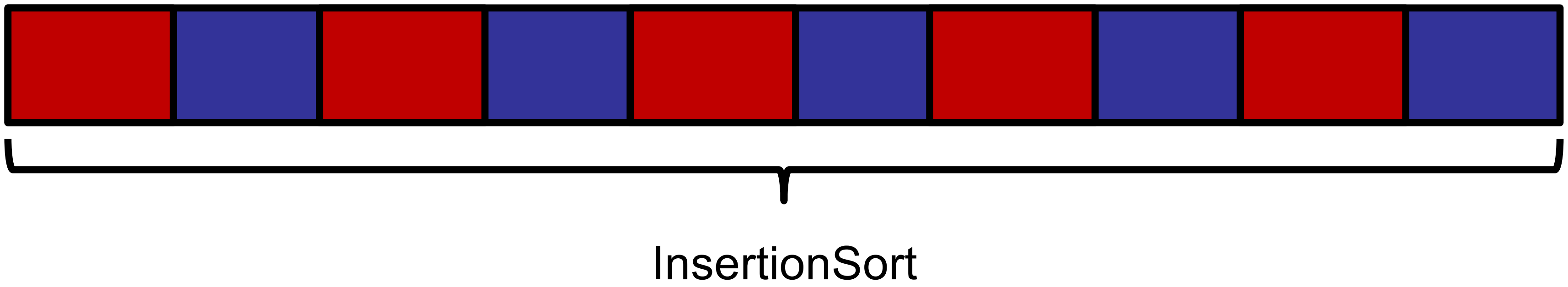
1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.



QuickSort Optimization:

Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.
3. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.



If the base case is size O(1), then what is the running time of the (single) InsertionSort?

1. $O(1)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^{3/2})$
5. $O(n^2)$
6. I have no idea.

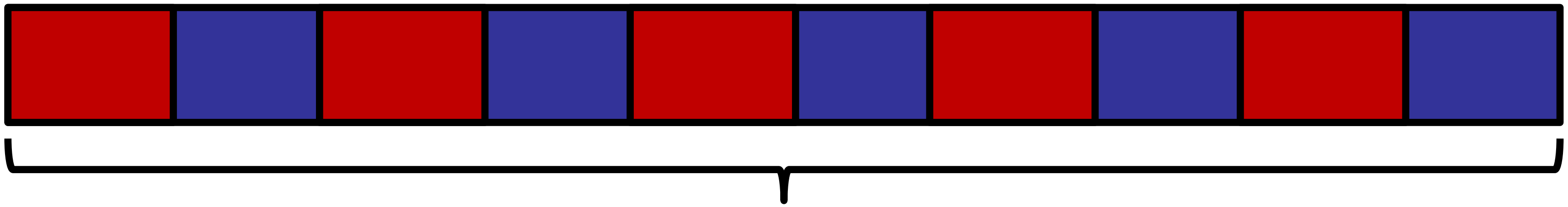
ARCHIPELAGO

is open

QuickSort Optimization:

Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.
3. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.



InsertionSort

Relies on fact that InsertionSort is very fast on *almost sorted* arrays!

Each inserted item moves at most distance O(1).

QuickSort

```
QuickSort(A[1..n])
    almostQuickSort(A[1..n], n)
    InsertionSort(A[1..n], n)

almostQuickSort(A[1..n], n)
    if (n < 10) then return
    else
        p = partition(A[1..n], n)
        x = almostQuickSort(A[1..p-1], p-1)
        y = almostQuickSort(A[p+1..n], n-p)
```

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

E.g.: Find the median ($k = n/2$)

Find the 7th element ($k = 7$)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

Option 1:

– Sort the array.

– Return element number k.

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

Option 1:

- Sort the array.
- Return element number k.

Running time?

ARCHIPELAGO
is open

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

Option 1:

- Sort the array.
- Return element number k.

Running time: O(n log n)

Order Statistics

Find k^{th} smallest element in an *unsorted* array:

x_{10}	x_2	x_4	x_1	x_5	x_3	x_7	x_8	x_9	x_6

Option 2:

- Only do the minimum amount of sorting necessary

Order Statistics

Key Idea: partition the array

x_2	x_4	x_1	x_3	x_5	x_7	x_8	x_9	x_6	x_{10}

Now continue searching in the correct half.

E.g.: Partition around x_5 and recursively search for x_3 in left half.

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Order Statistics

Example: search for 5th element

9	**22**	**13**	**17**	**5**	**3**	**100**	**6**	**19**	**8**

Partition around random pivot: 17

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**	**17**	**100**	**19**	**22**
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 5th element in left half.

9	**8**	**13**	**5**	**3**	**6**				
1	2	3	4	5	6				

Order Statistics

Example: search for 5th element

9	**8**	**13**	**5**	**3**	**6**				

Partition around random pivot: 8

6	**3**	**5**	**8**	**13**	**9**				
1	2	3	**4**	5	6				

Order Statistics

Example: search for 5th element

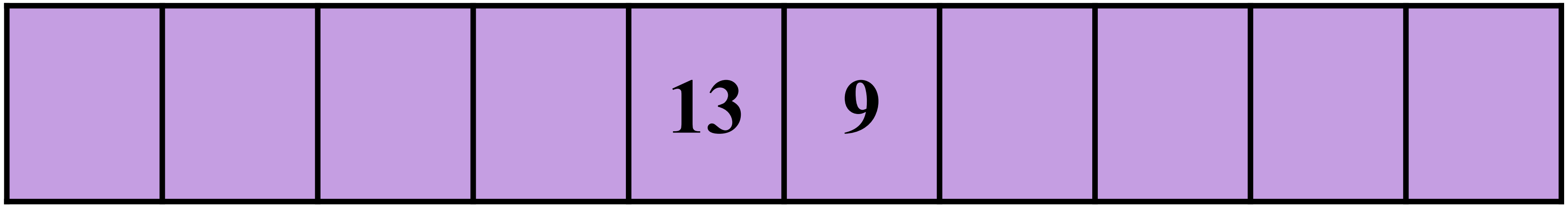
9	**8**	**13**	**5**	**3**	**6**				

Search for: 5 – 4 = 1 in right half

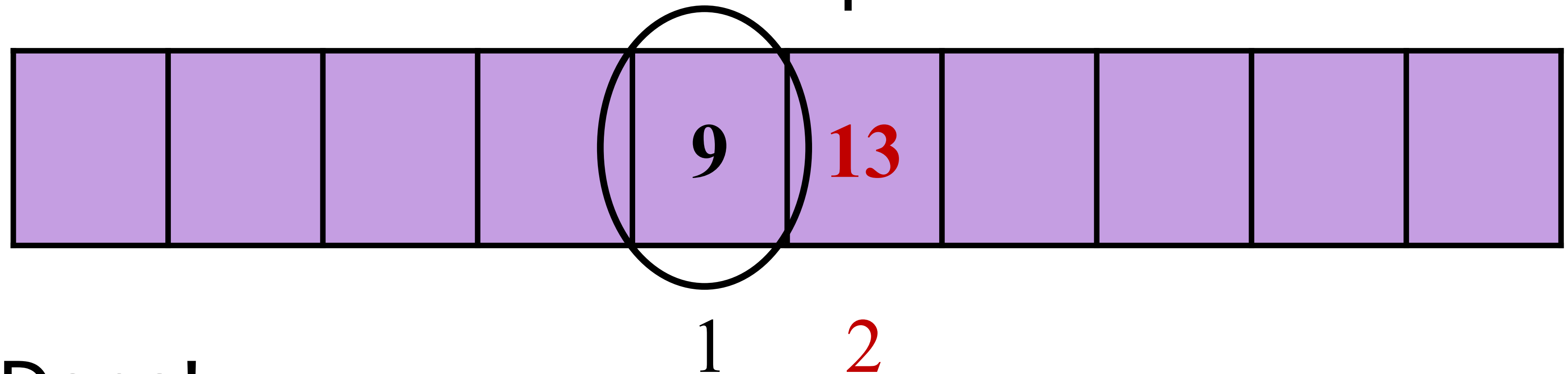
6	**3**	**5**	**8**	**13**	**9**				
1	2	3	**4**	5	6				

Order Statistics

Search for: 5 – 4 = 1 in right half



Partition around random pivot: 13



Done!

Finding the k^th smallest element

```
Select(A[1..n], n, k)
    if (n == 1) then return A[1];
    else  Choose random pivot index pIndex.
          p = partition(A[1..n], n, pIndex)
          if (k == p) then return A[p];
          else if (k < p) then
                return Select(A[1..p–1], k)
          else if (k > p) then
                return Select(A[p+1], k – p)
```

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 5th element

9	22	13	17	5	3	100	6	19	8

Partition around random pivot: 17

9	8	13	5	3	6	17	100	19	22
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 5th element on the left.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 8^{th} element

9	22	13	17	5	3	100	6	19	8

Partition around random pivot: 8

5	6	3	8	17	13	100	22	19	9
1	2	3	4	5	6	**7**	**8**	**9**	**10**

Search for 4^{th} element on the right.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 4th element

9	22	13	17	5	3	100	6	19	8

Partition around random pivot: 8

5	6	3	8	17	13	100	22	19	9
1	2	3	4	5	6	7	8	9	10

Return 8.

Finding the kth smallest element

```
Select(A[1..n], n, k)

    if (n == 1) then return A[1];
    else  Choose random pivot index pIndex.
          p = partition(A[1..n], n, pIndex)
          if (k == p) then return A[p];
          else if (k < p) then
                return Select(A[1..p–1], k)
          else if (k > p) then
                return Select(A[p+1], k – p)
```

Finding the k^{th} smallest element

Key point:

- Only recurse *once*!

- Why not recurse twice?
  - Does not help---the correct element is only on one side.
  - You do not need to sort both sides!
  - Makes it run a lot faster.
  - If you recurse on both sides, you are sorting!

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

```
repeat
        p = partition(A[1..n], n, pIndex)
until (p > n/10) and (p < 9n/10)
```

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[T(n)] \leq \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\ \text{partitions}](n)$$

cost of partitioning

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[\mathrm{T(n)}] \le \mathbf{E}[\mathrm{T(9n/10)}] + \mathbf{E}[\#\ \text{partitions}](n)$$
$$\le \mathbf{E}[\mathrm{T(9n/10)}] + 2n$$

Recurrence

What is the solution to the following recurrence?

$$T(n) \quad \leq \quad T(9n/10) + 2n$$

ARCHIPELAGO

is open

Recurrence

What is the solution to the following recurrence?

$$
\begin{aligned}
T(n) &\leq T(9n/10) + 2n \\
&= O(n)
\end{aligned}
$$

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$\mathbf{E}[\mathrm{T(n)}] \le \mathbf{E}[\#\ \text{partitions}](n) + \mathbf{E}[\mathrm{T(9n/10)}]$$

$$\le 2n + \mathbf{E}[\mathrm{T(9n/10)}]$$

$$\le 2n + 2n\,(9/10) + (9/10)\,\mathbf{E}[\mathrm{T(9n/10)}]$$

$$\le 2n + 2n\,(9/10) + 2n\,(9/10)^2 + \ldots\ldots$$

Analysis

Paranoid-Select:

Repeatedly partition until at least $n/10$ in each half of the partition.

Recurrence:

$$
\begin{aligned}
\mathbf{E}[\mathrm{T(n)}] &\leq \mathbf{E}[\mathrm{T(9n/10)}] + \mathbf{E}[\#\ \text{partitions}](n) \\
&\leq \mathbf{E}[\mathrm{T(9n/10)}] + 2n \\
&\leq \mathrm{O}(n)
\end{aligned}
$$

Recurrence: $T(n) = T(n/c) + O(n)$

Summary

QuickSort: O(n log n)

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics: O(n)

- Finding the k^{th} smallest element in an array.
- Key idea: partition
- Paranoid Select

Plan of the Week

Trees

- Terminology
- Traversals
- Operations

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

Implementation

Option 1: Sorted array

- insert : ?
- search : ?

Option 2: Unsorted array

- insert : ?
- search : ?

Option 3: Linked list

- insert : ?
- search : ?

ARCHIPELAGO

is open

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)

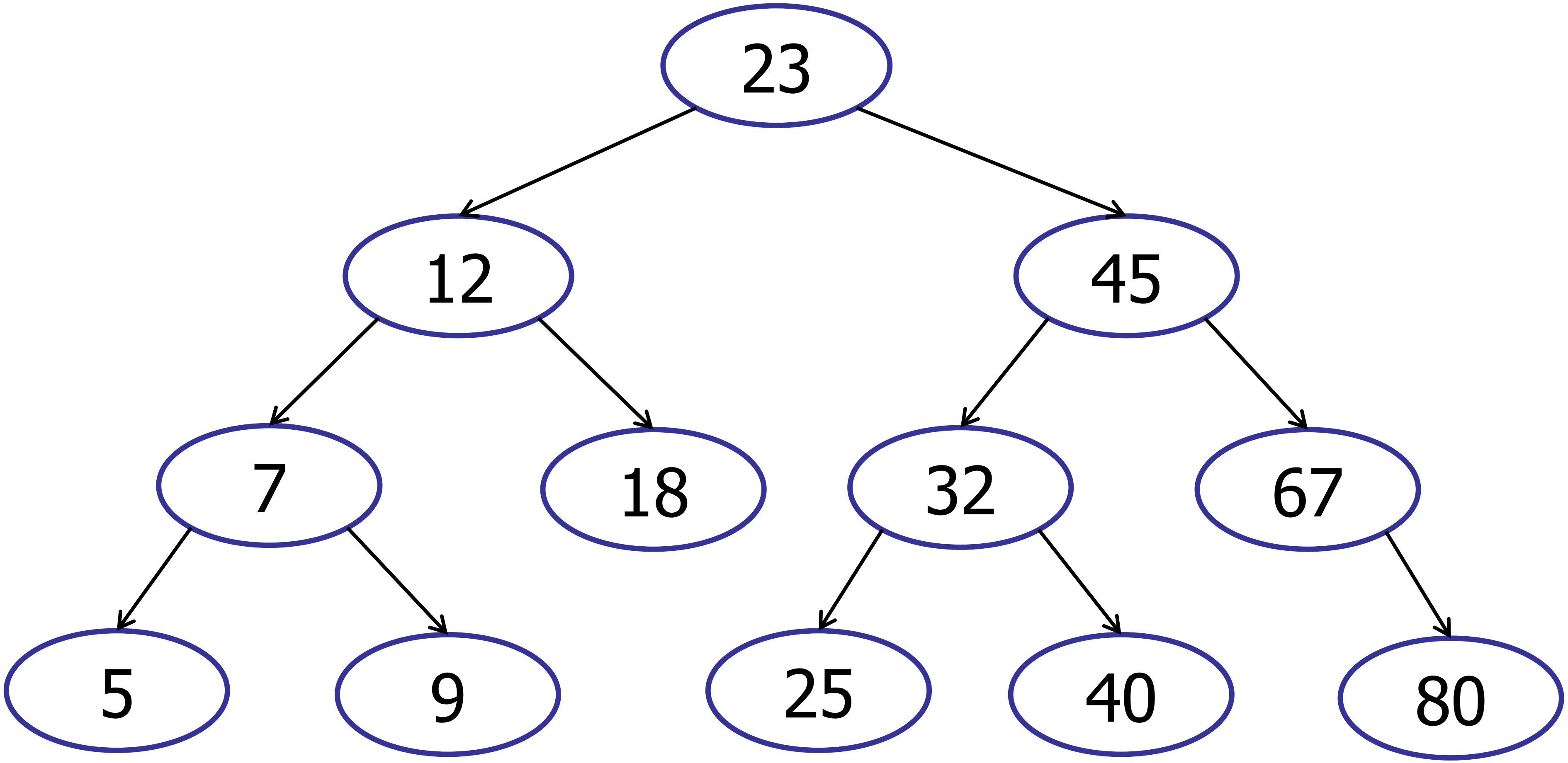
Dictionary Implementation

Possible Choices:

- Implement using an array
- Implement using a Java library (see: java.util.Vector or java.util.ArrayList).
- Implement using a queue.
- Implement using a linked list
- …

Dictionary

Implementation idea: Tree

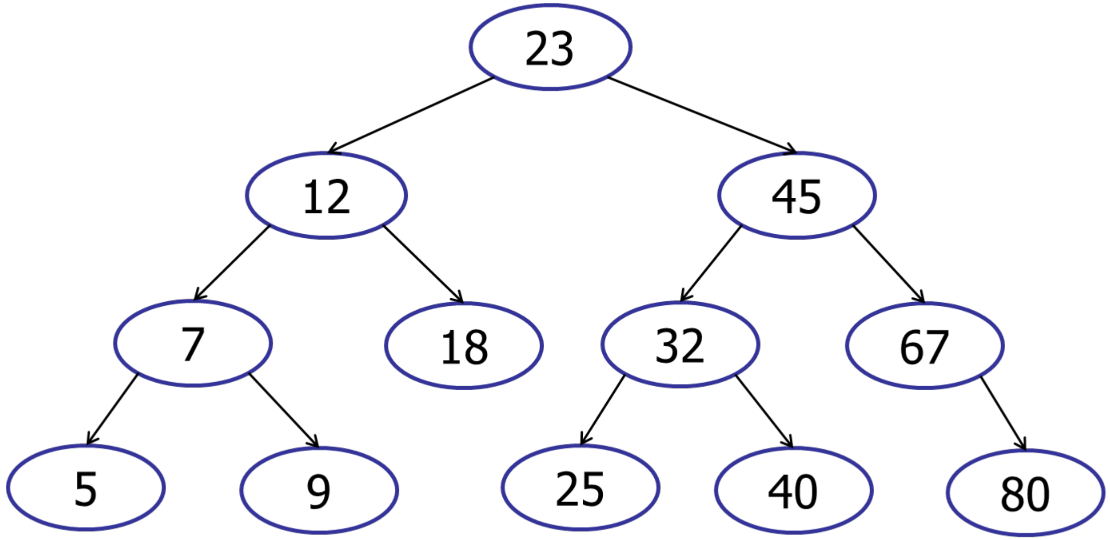


Dictionary

Implementation idea: Tree

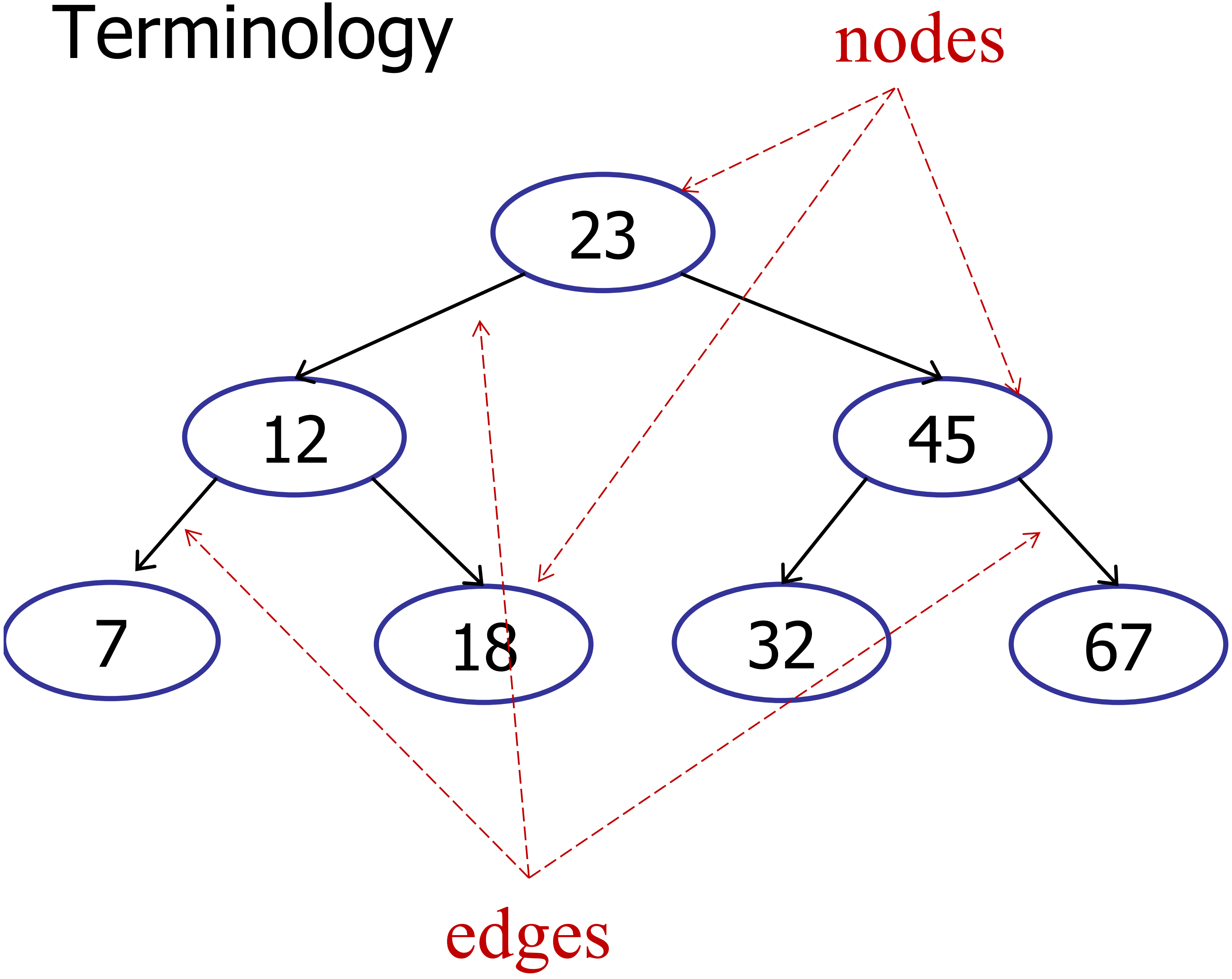
Critical Components:

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



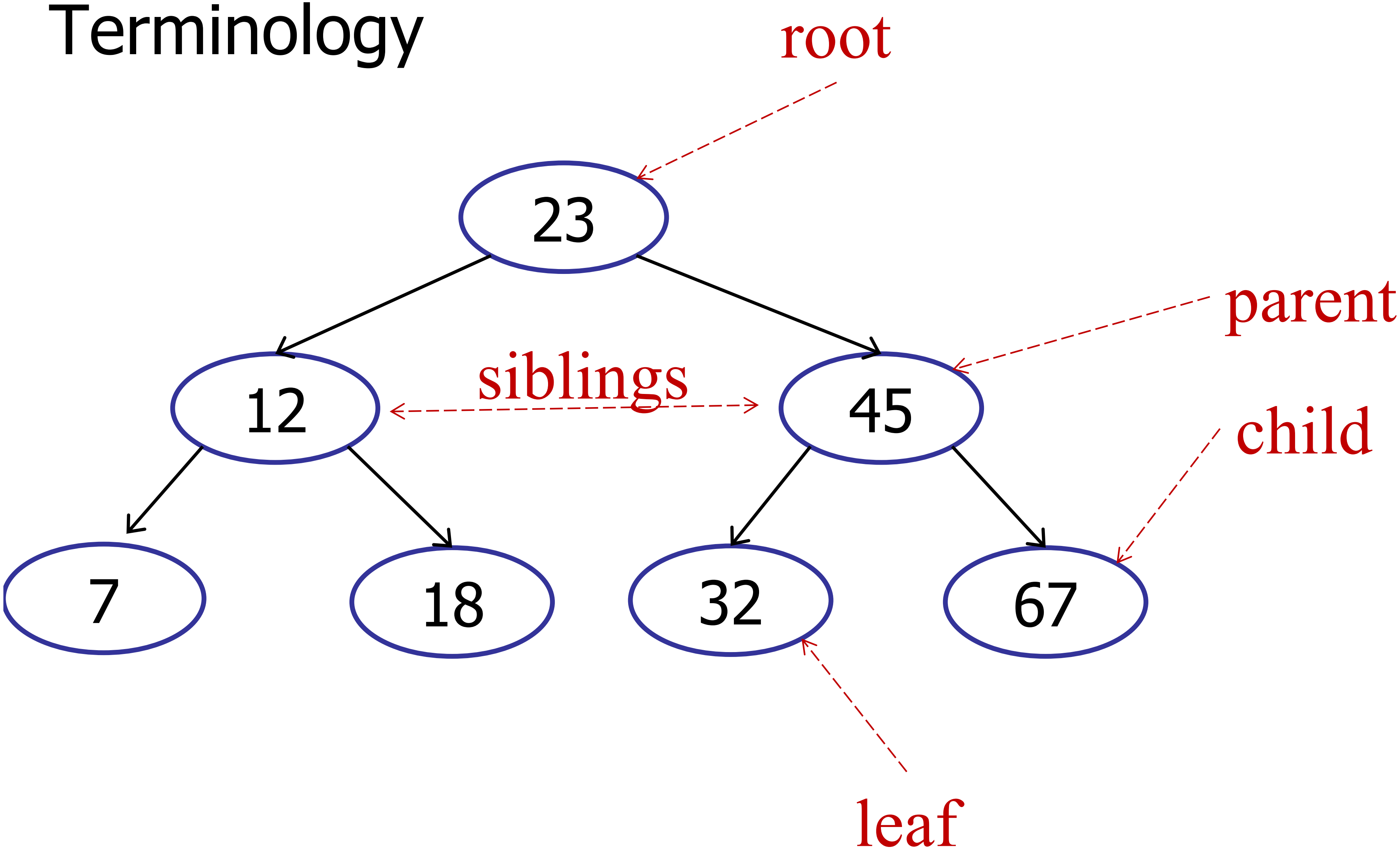
Binary Tree

Terminology

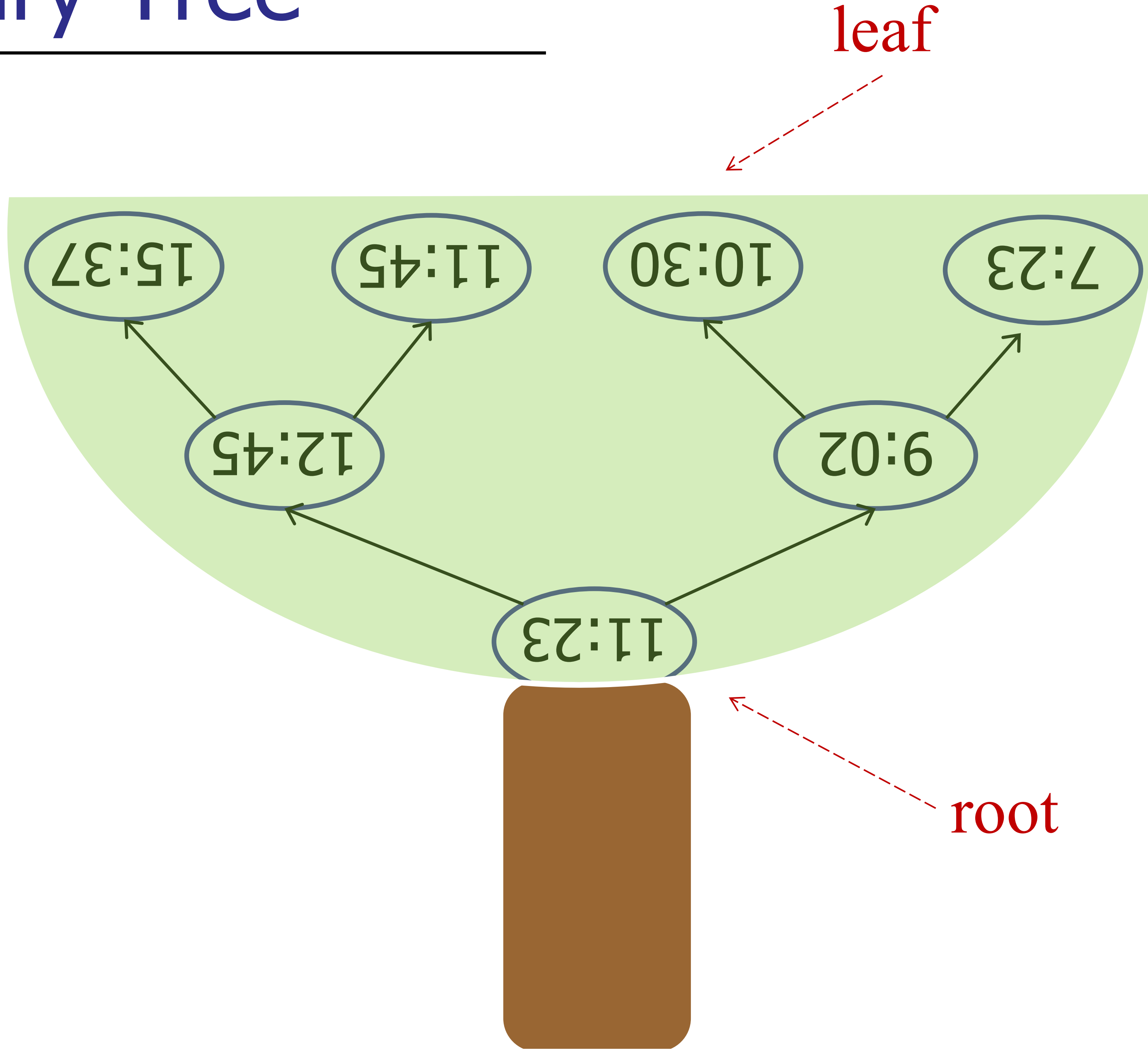


Binary Tree

Terminology

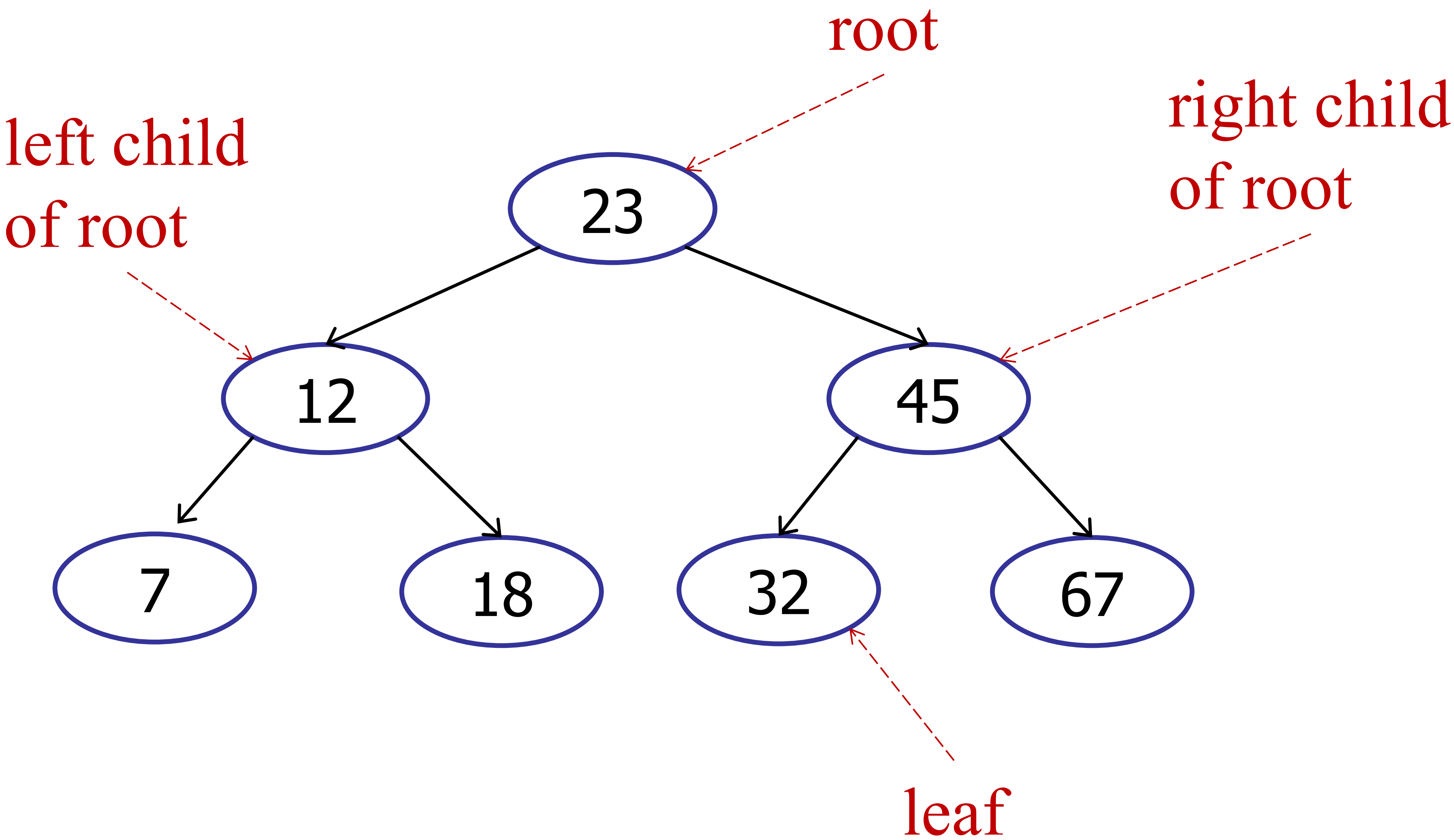


Binary Tree



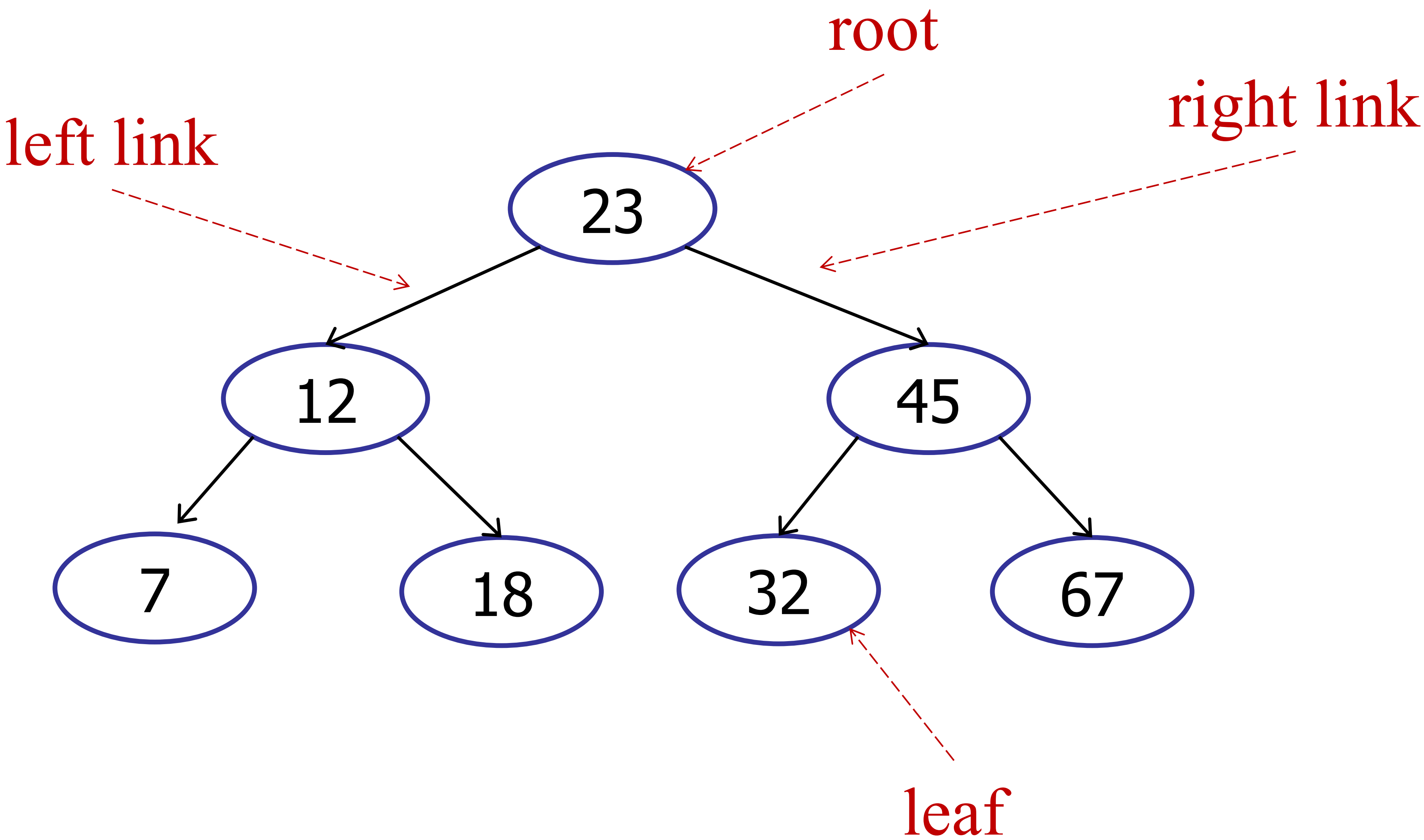
Binary Tree

Terminology



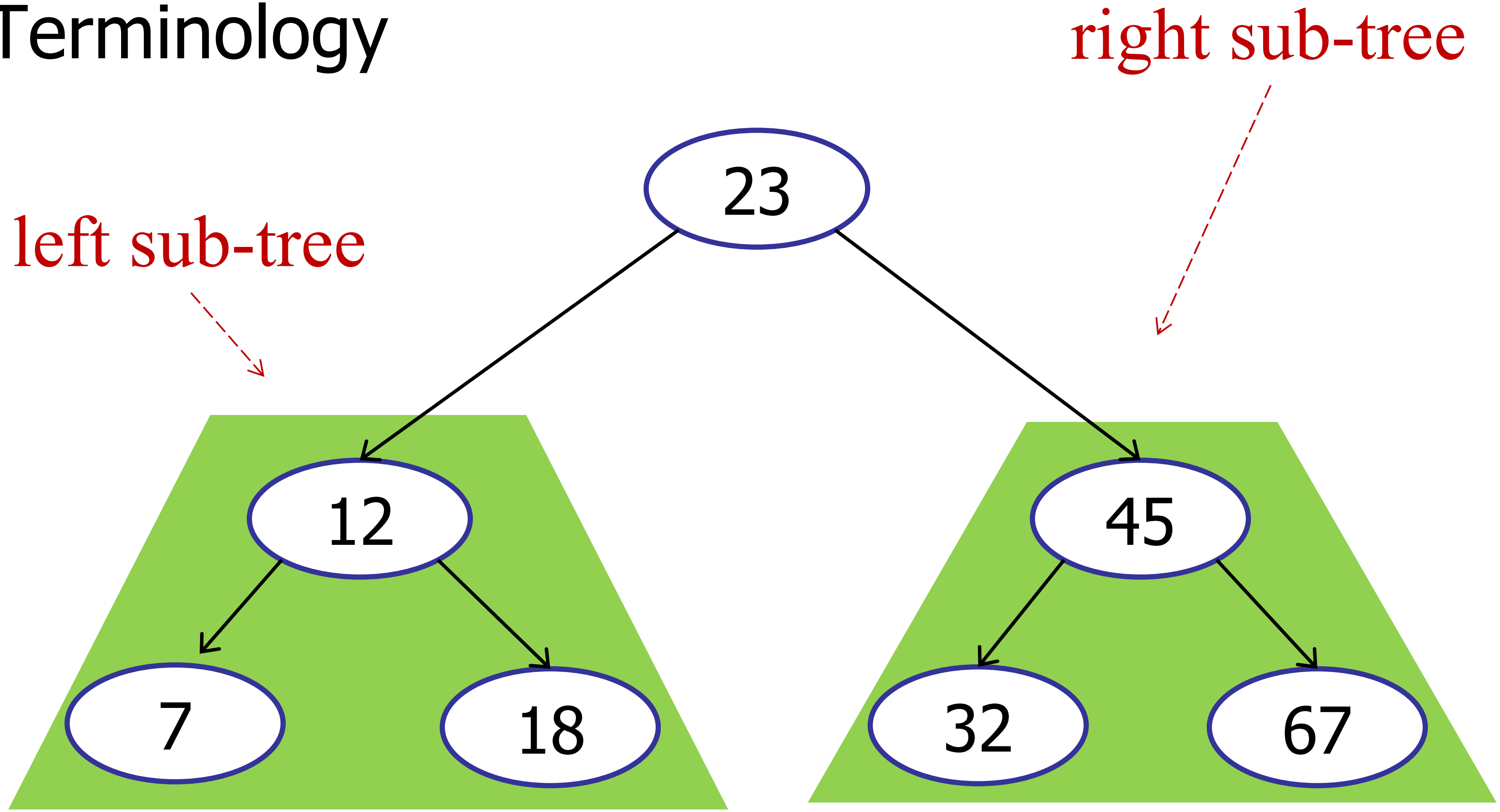
Binary Tree

Terminology



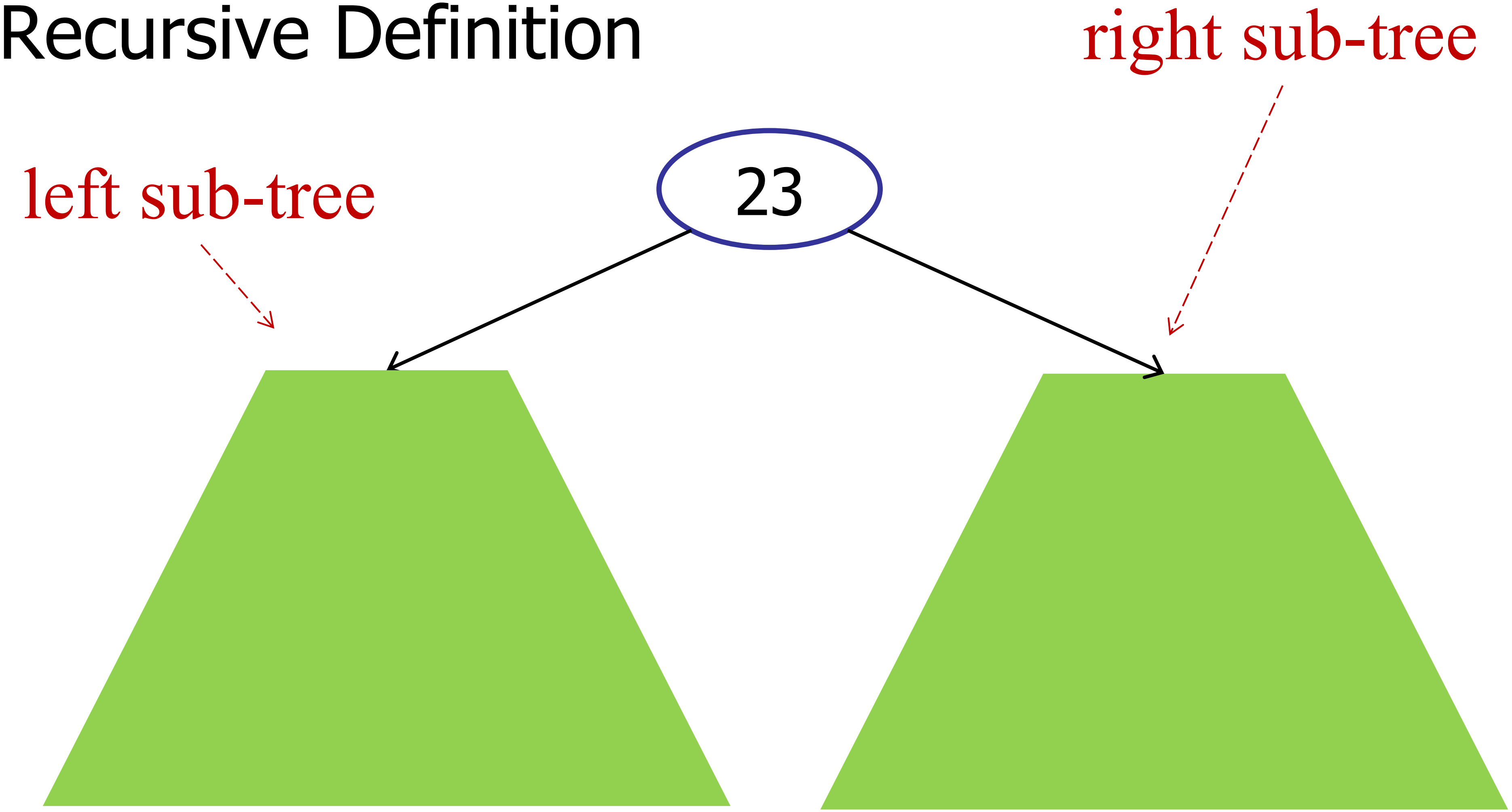
Binary Tree

Terminology



Binary Tree

Recursive Definition



A binary tree is either: **(a) empty**

(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??

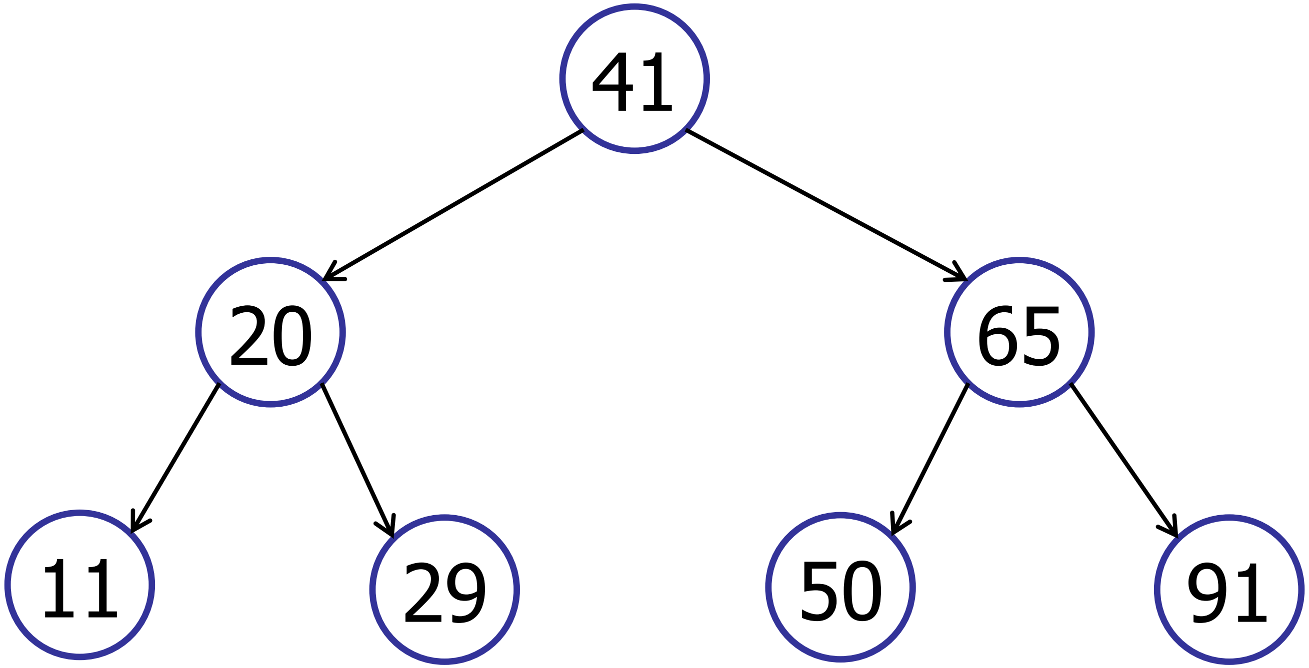
```
public class TreeNode {

       private TreeNode leftTree;
       private TreeNode rightTree;

       private int key;
       private int value;

       // Remainder of binary tree implementation
}
```

Binary **Search** Trees (BST)

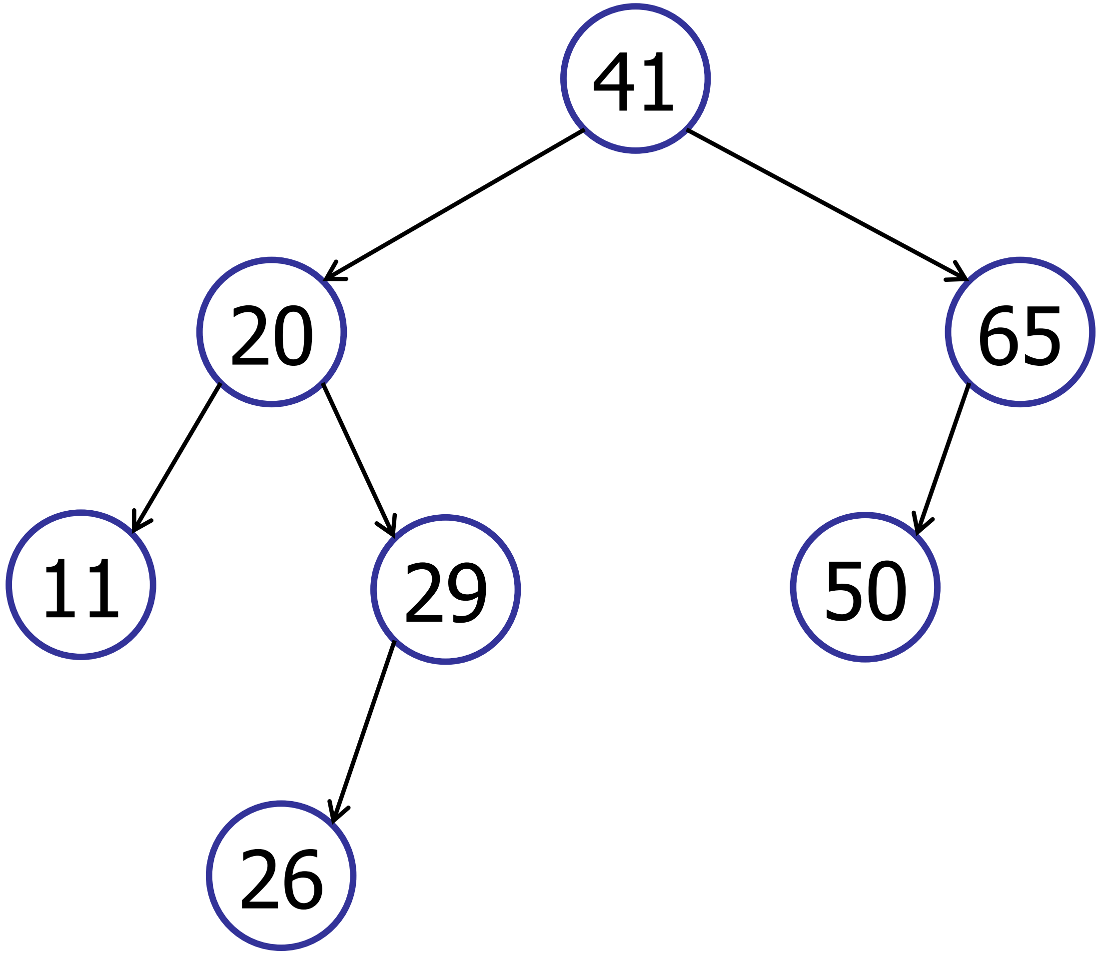


BST Property:

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

Is this a binary search tree?

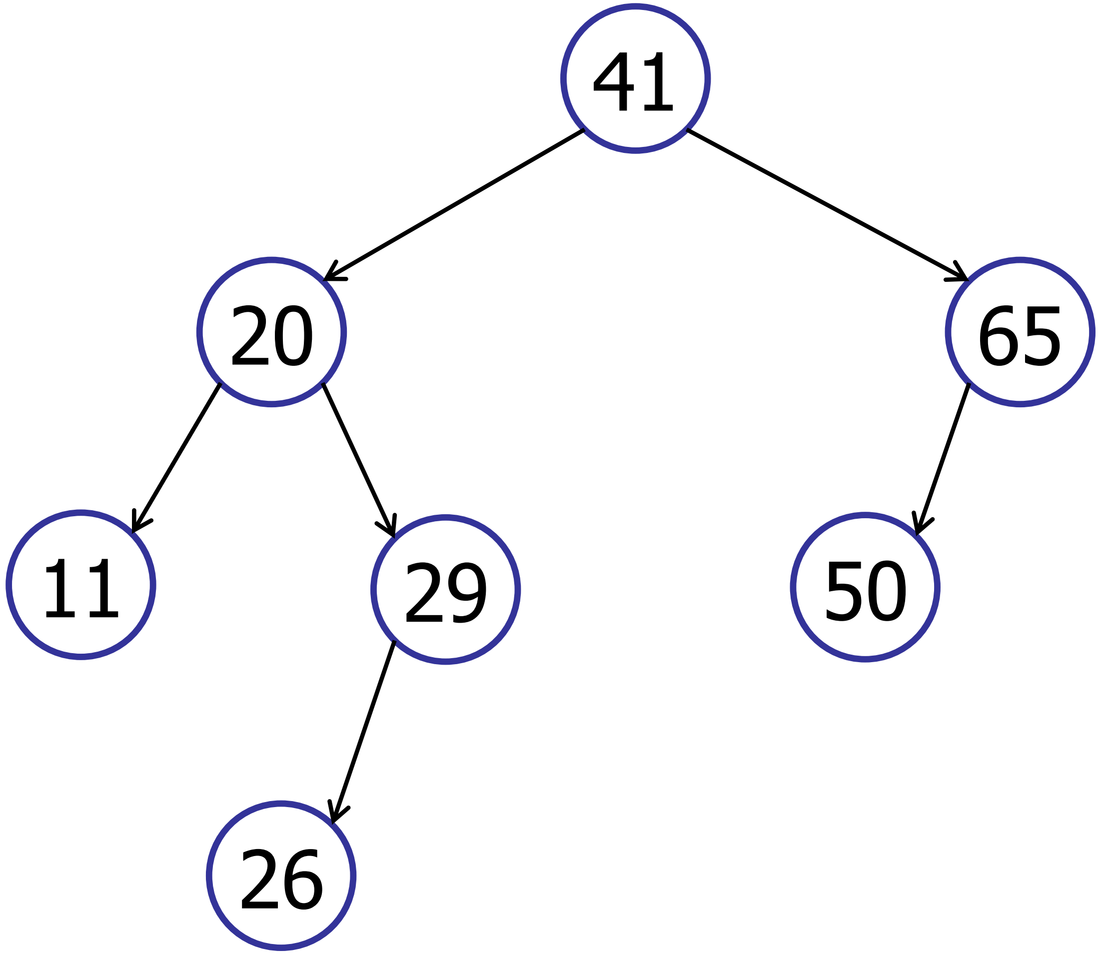
1. Yes
2. No
3. I don't know.



Yes or No on Zoom.

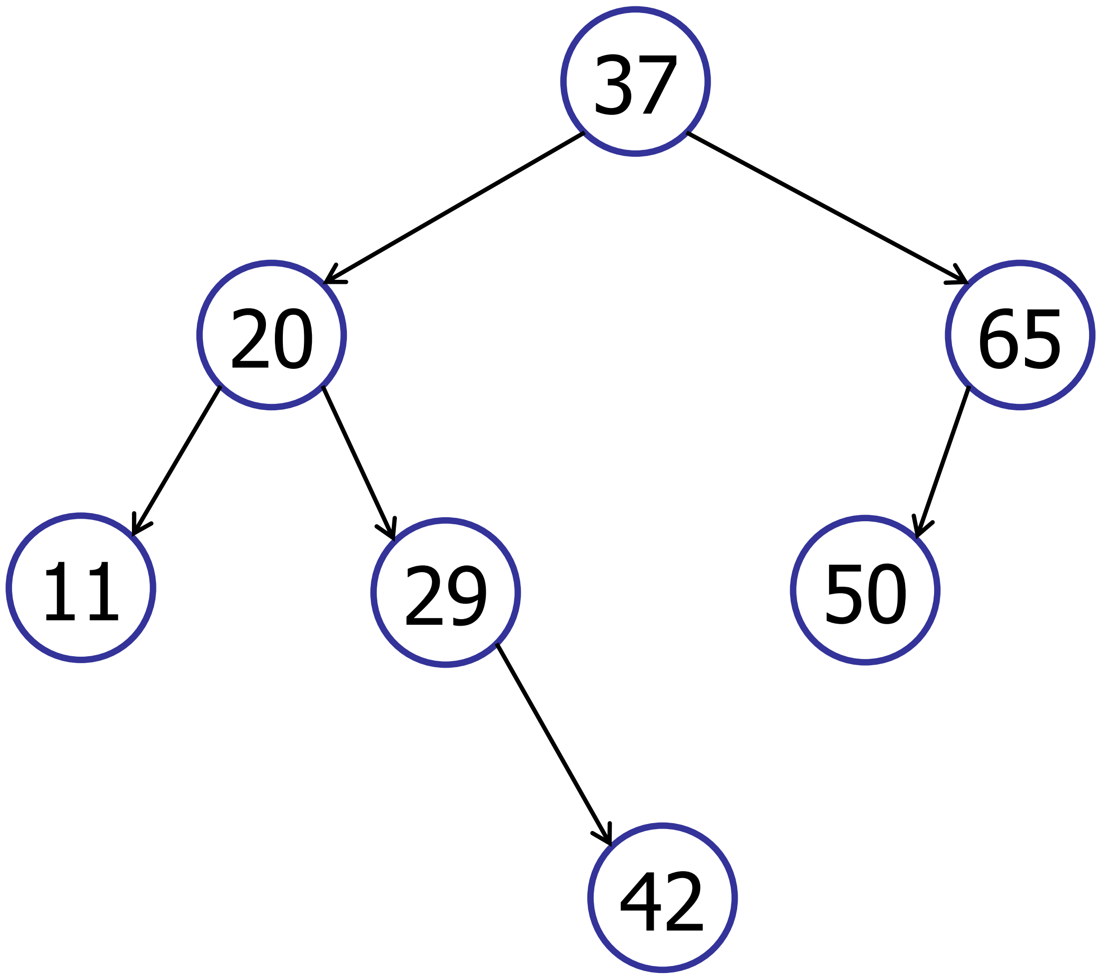
Is this a binary search tree?

✓ 1. Yes
2. No
3. I don't know.



Is this a binary search tree?

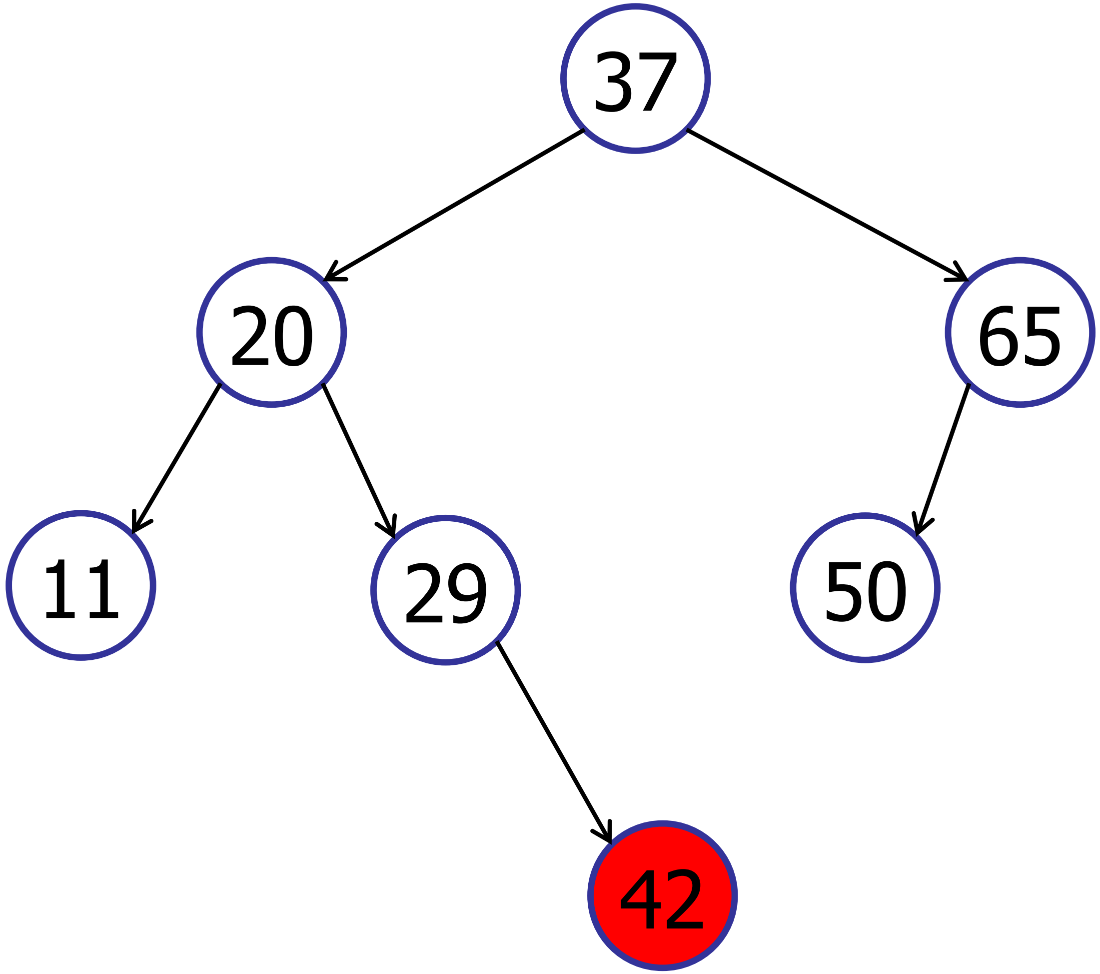
1. Yes
2. No
3. I don’t know.



Yes or No on Zoom.

Is this a binary search tree?

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



Binary Search Trees

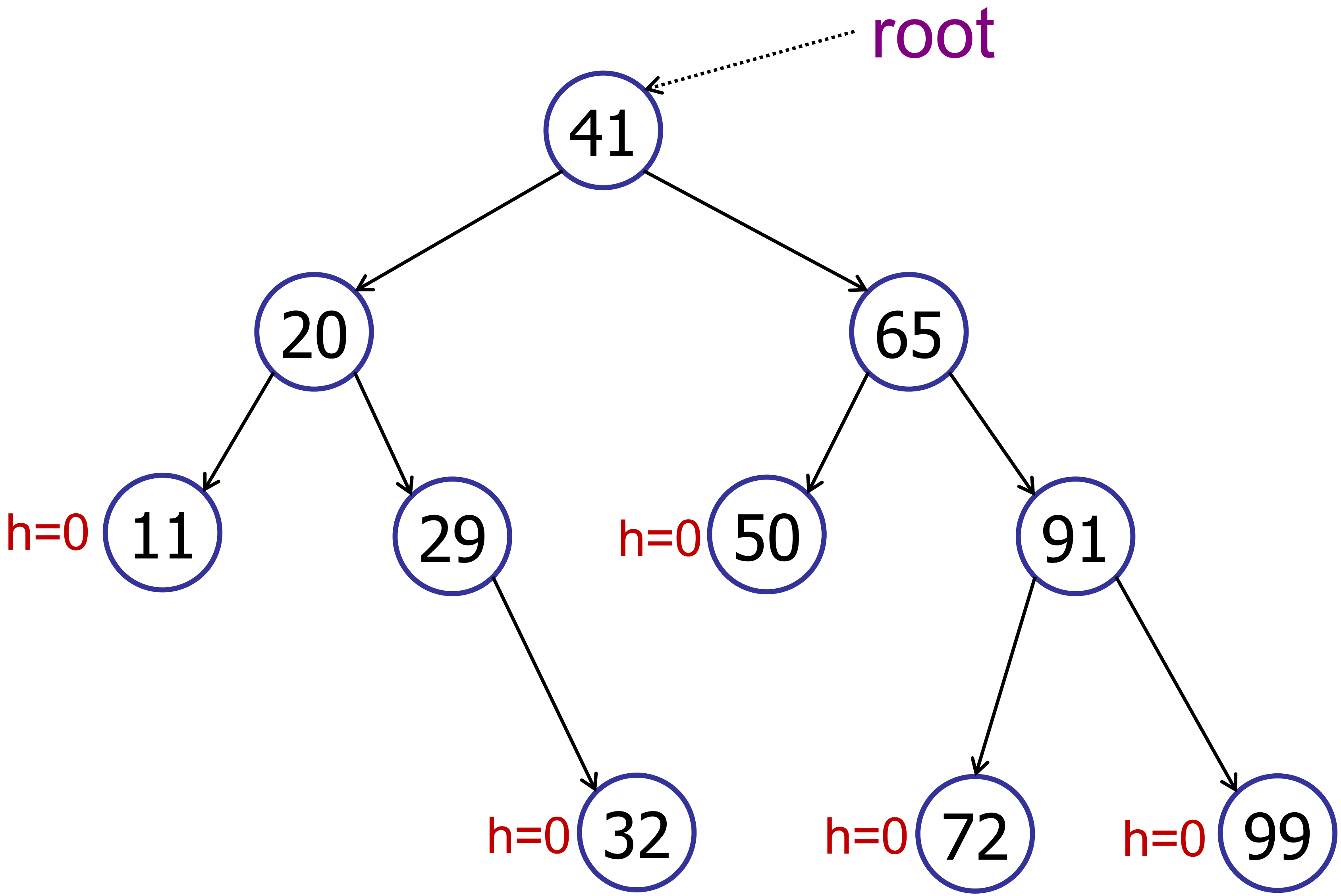
1. Terminology and Definitions

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

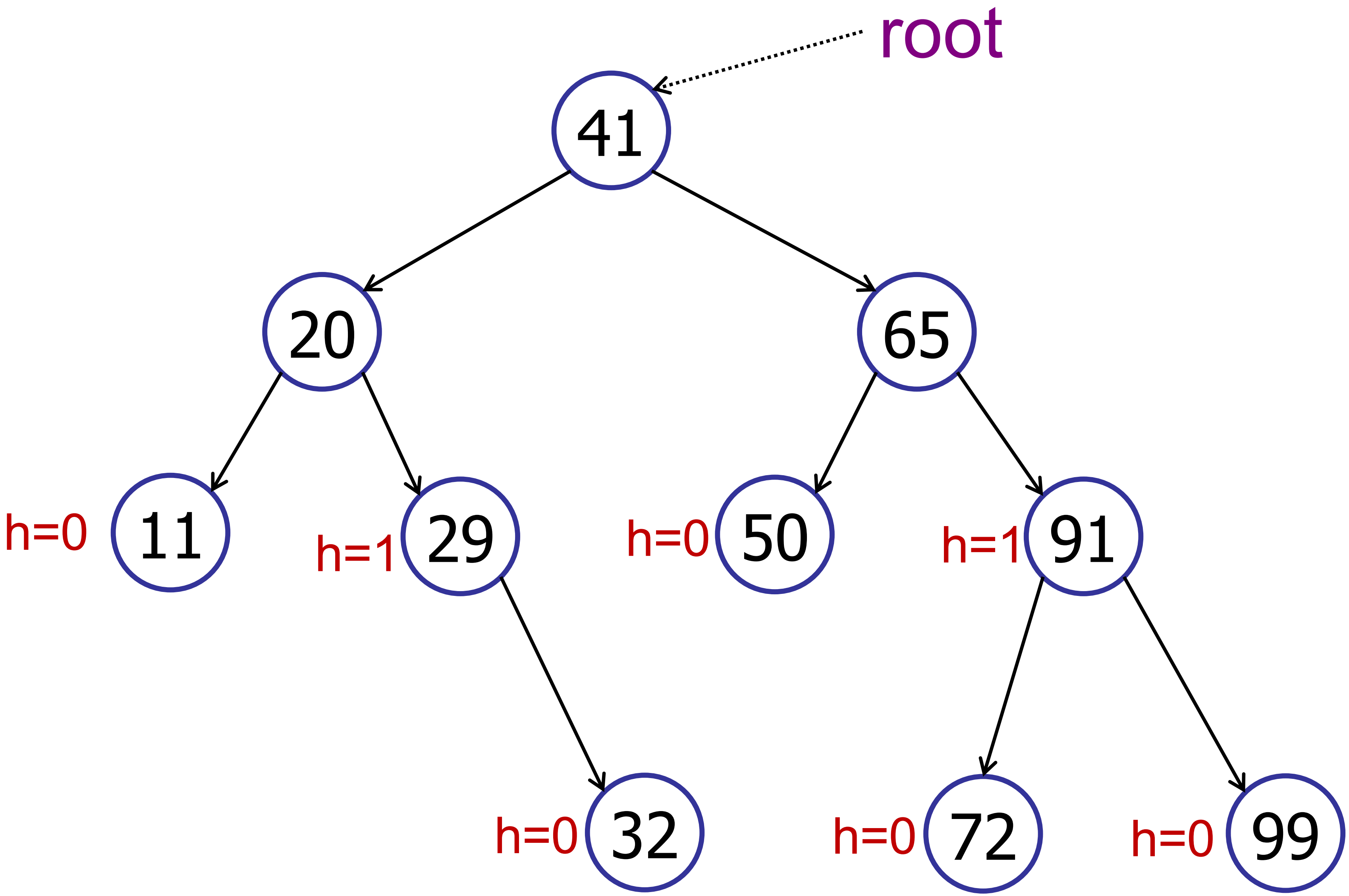
3. Traversals
   - in-order, pre-order, post-order

4. Other operations

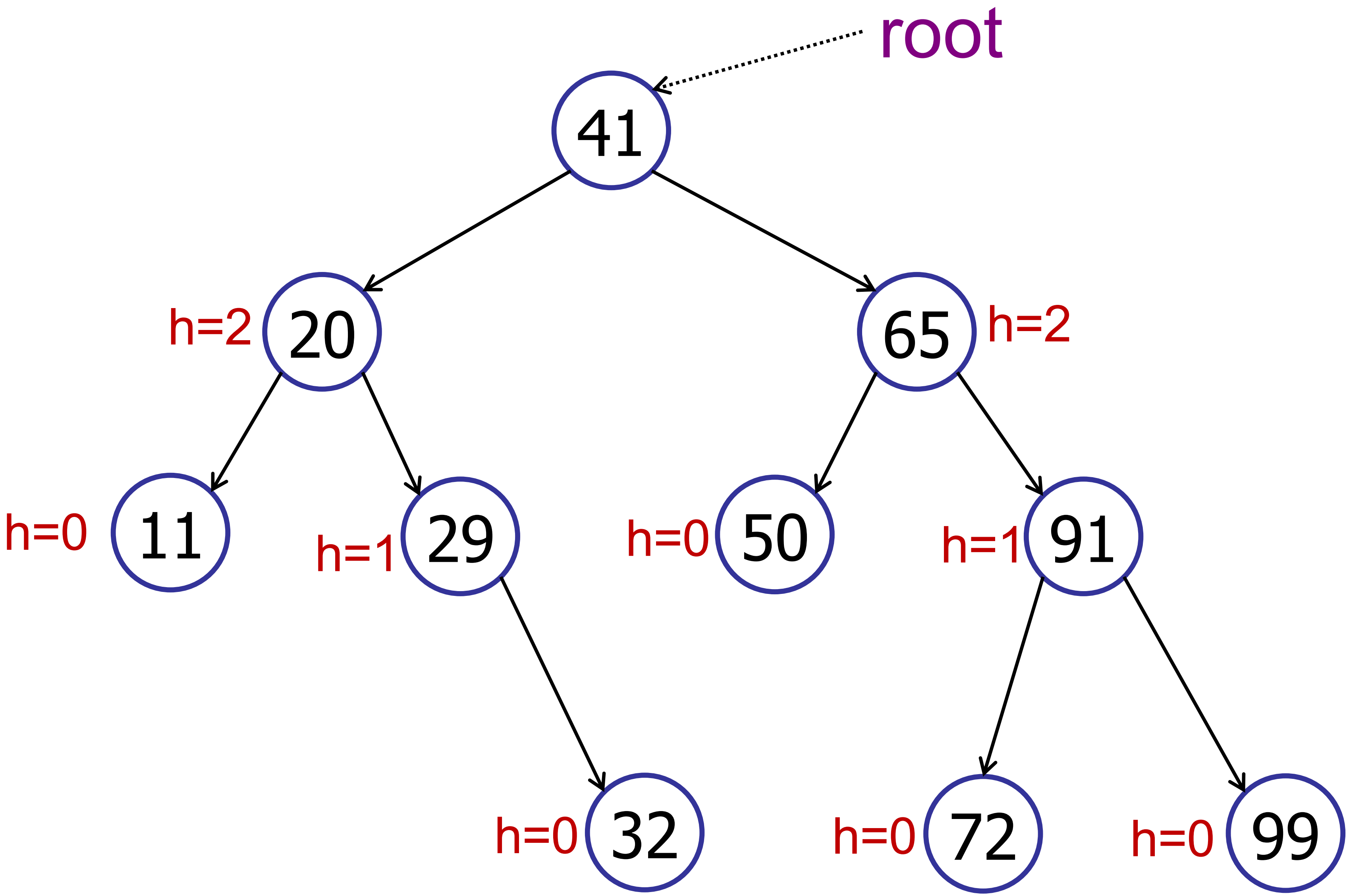
Height of a Binary Tree



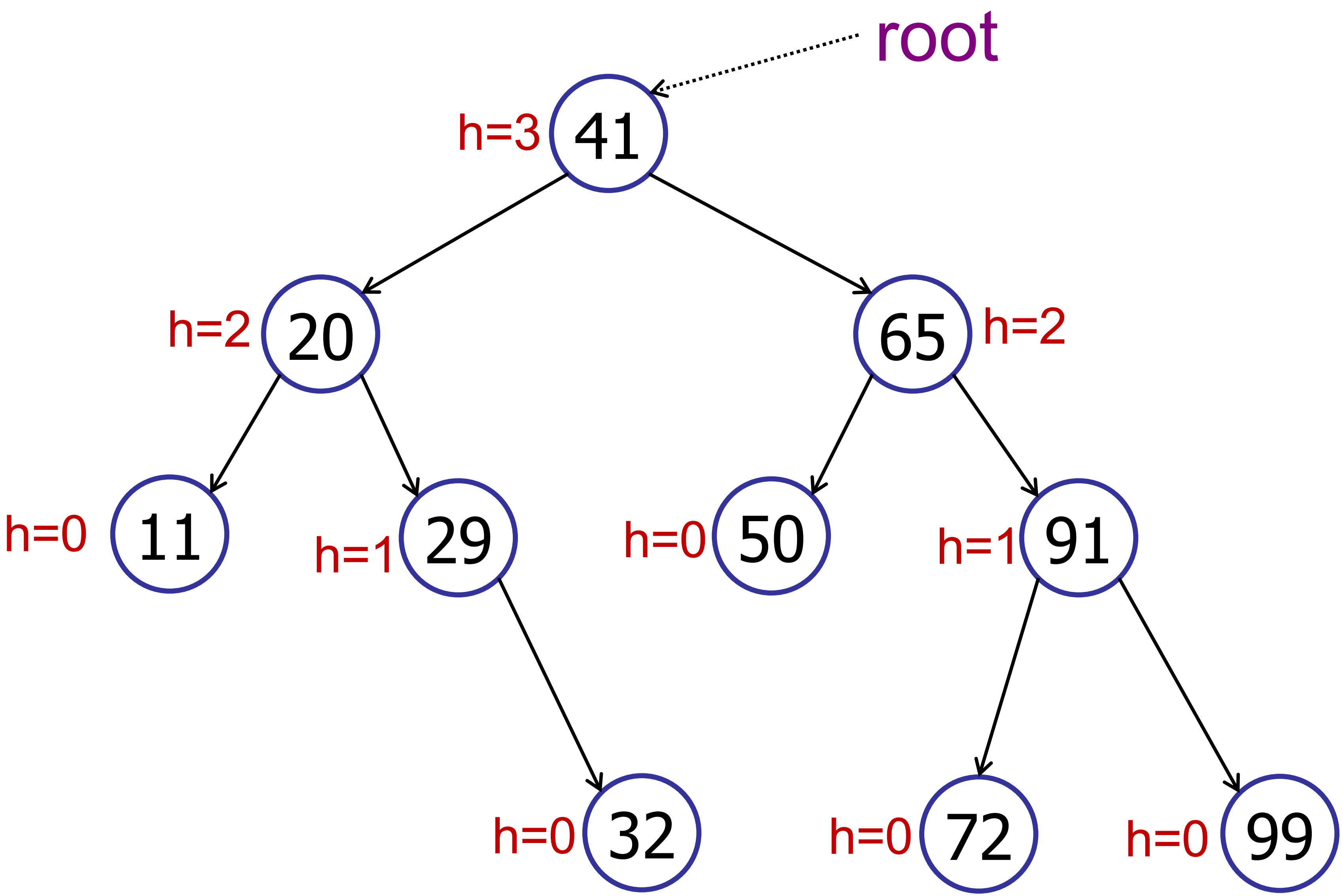
Height of a Binary Tree



Height of a Binary Tree



Height of a Binary Tree



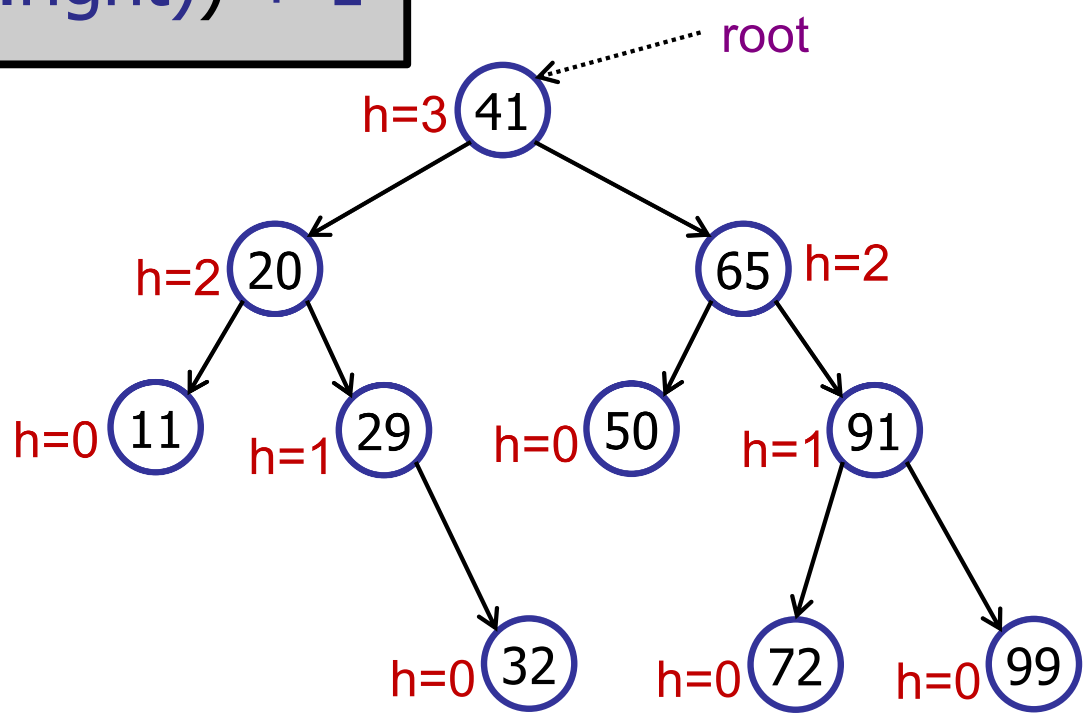
Height of a Binary Tree

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

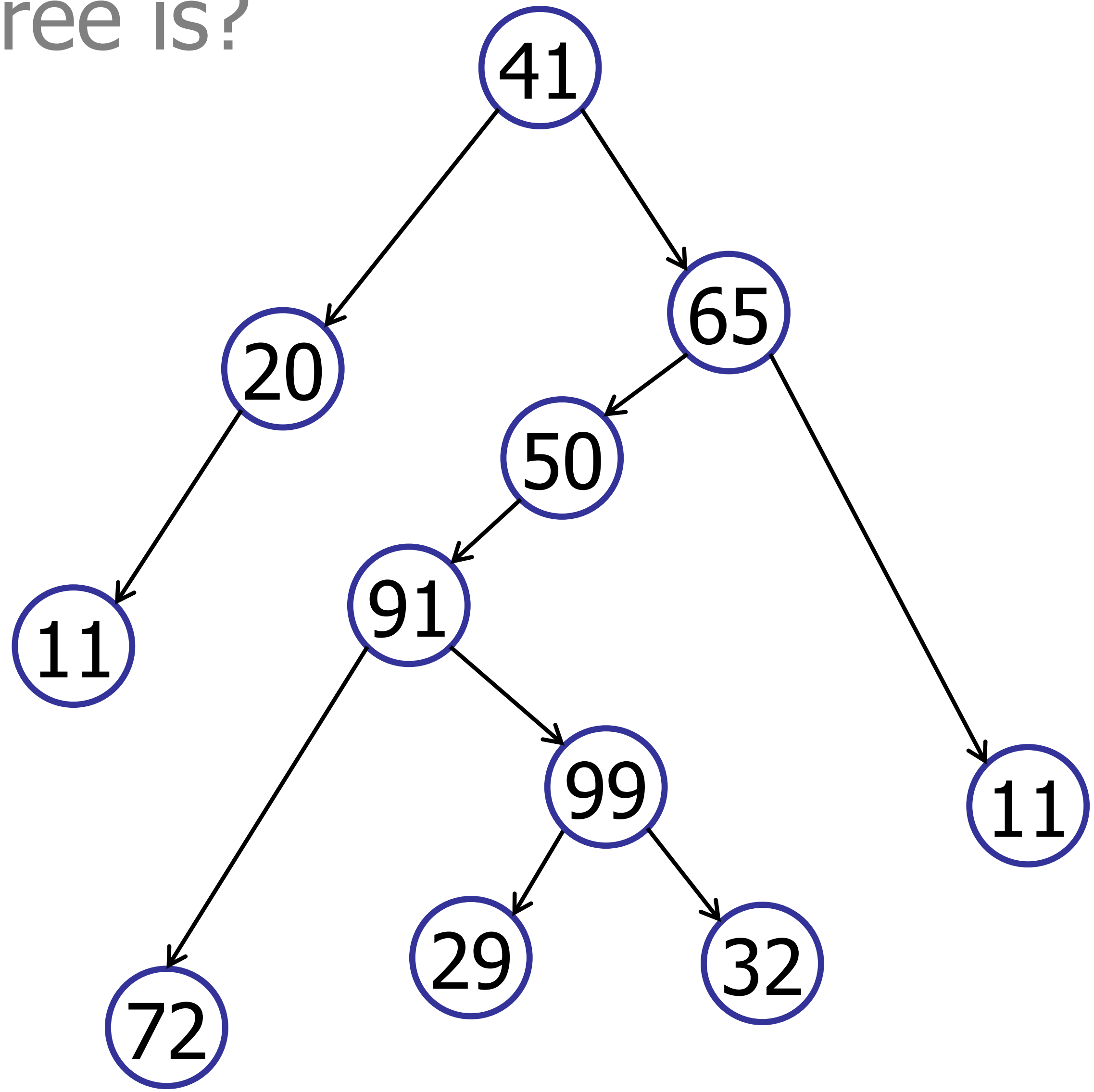
```
public int height(){
        int leftHeight = -1;
        int rightHeight = -1;
        if (leftTree != null)
                leftHeight = leftTree.height();
        if (rightTree != null)
                rightHeight = rightTree.height();
        return max(leftHeight, rightHeight) +  1;
}
```

max of subtrees

add 1

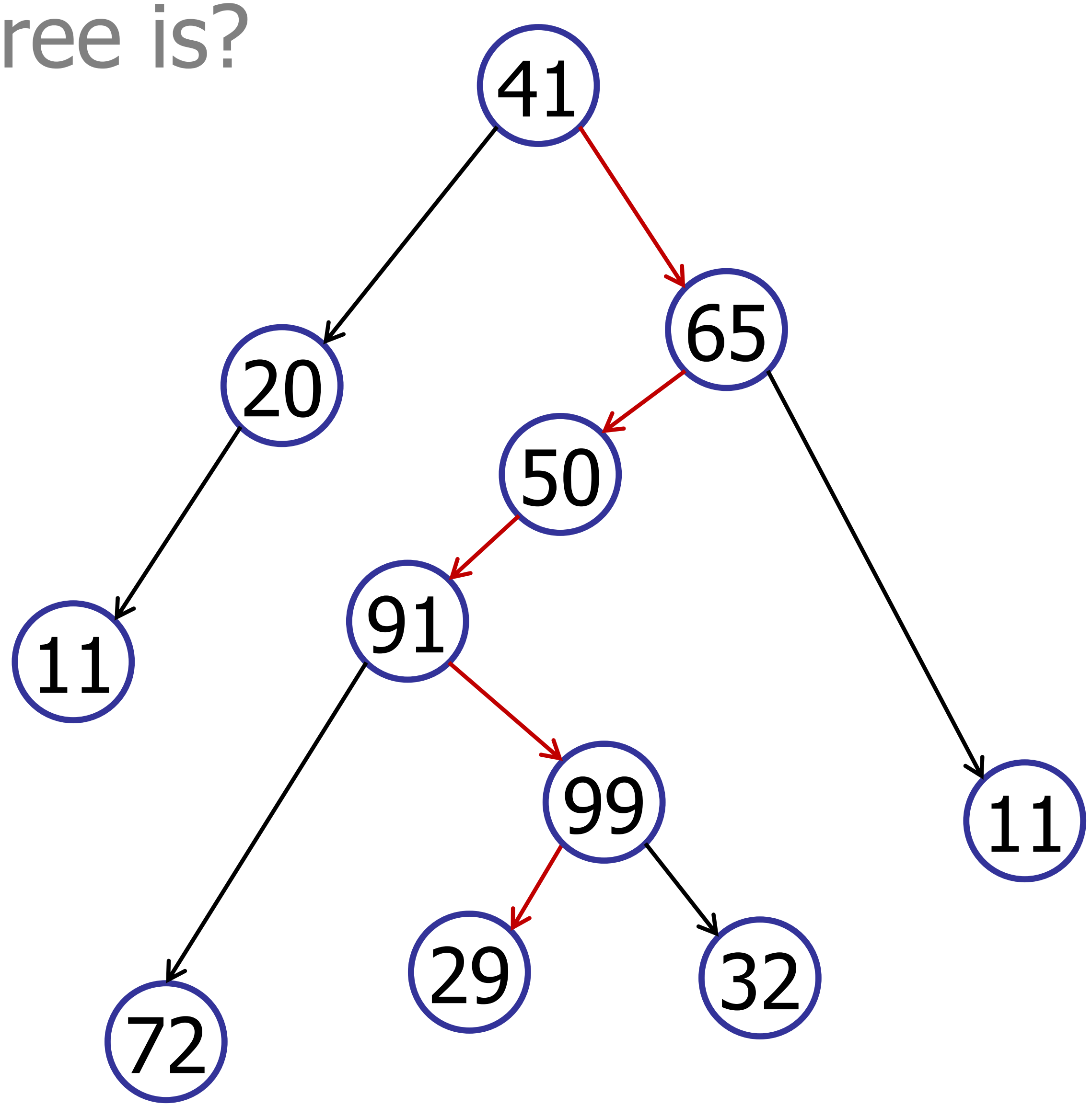
The height of this tree is?

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

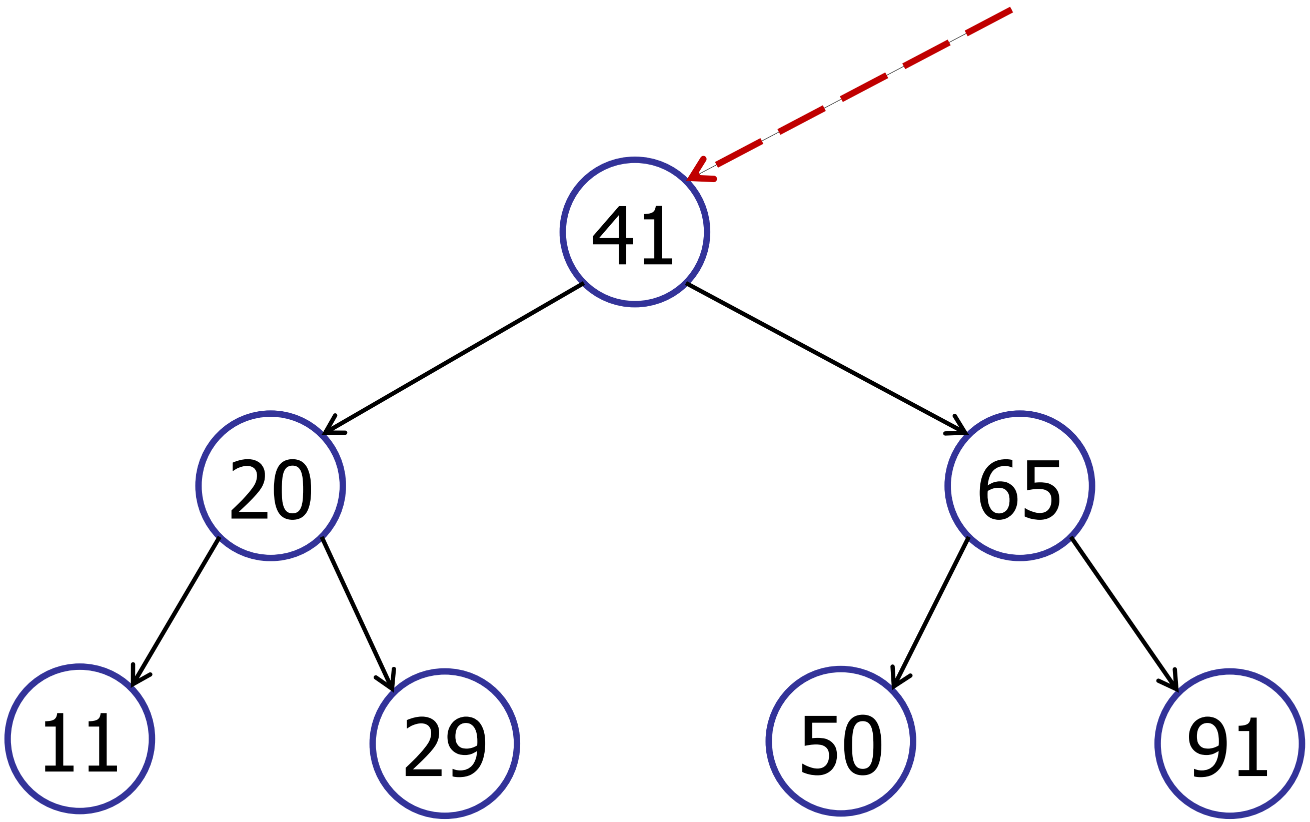


Binary Search Trees

1. Terminology and Definitions
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert
3. Traversals
   - in-order, pre-order, post-order
4. Other operations

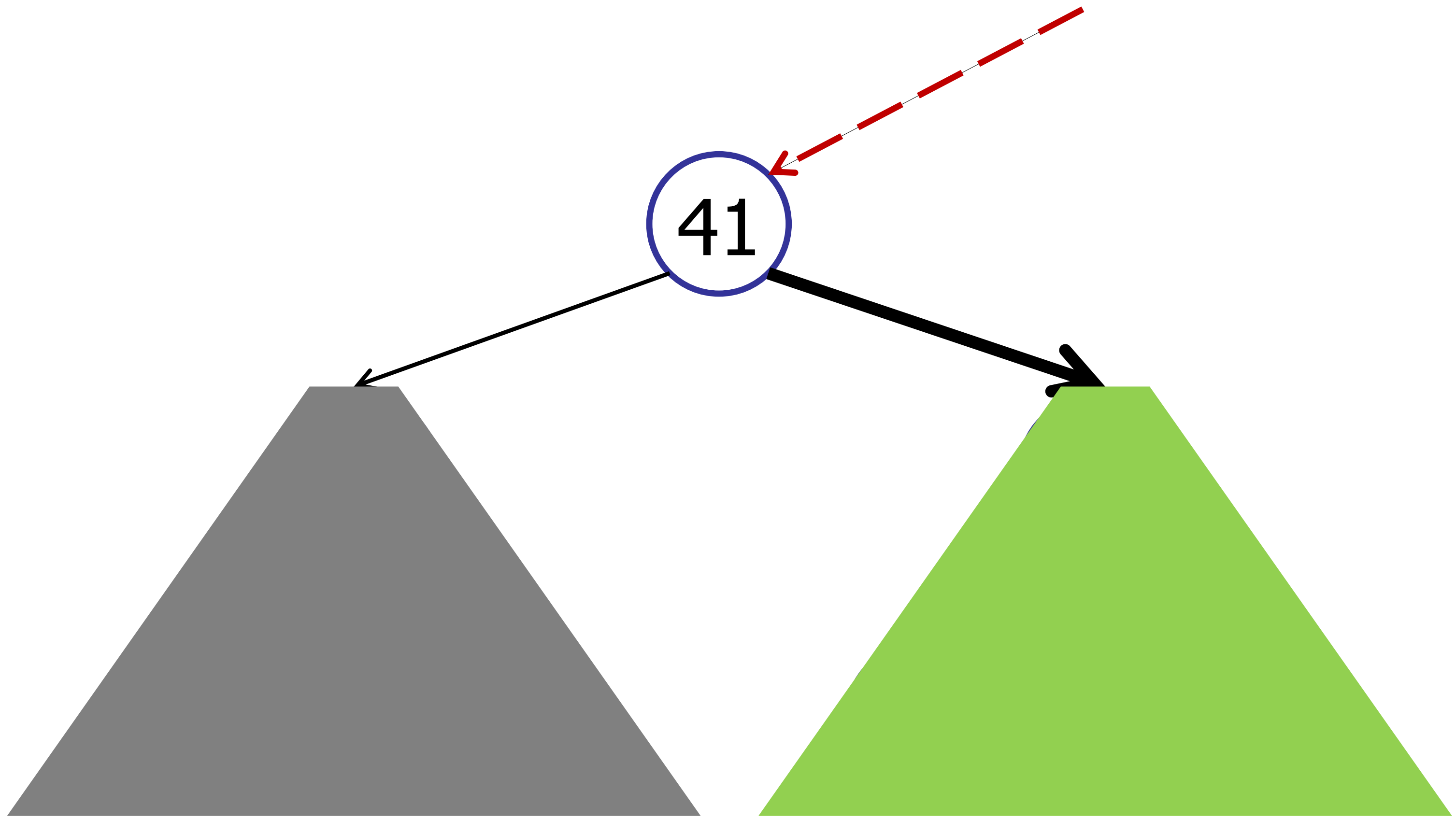
Binary Search Trees

Search for the maximum key:



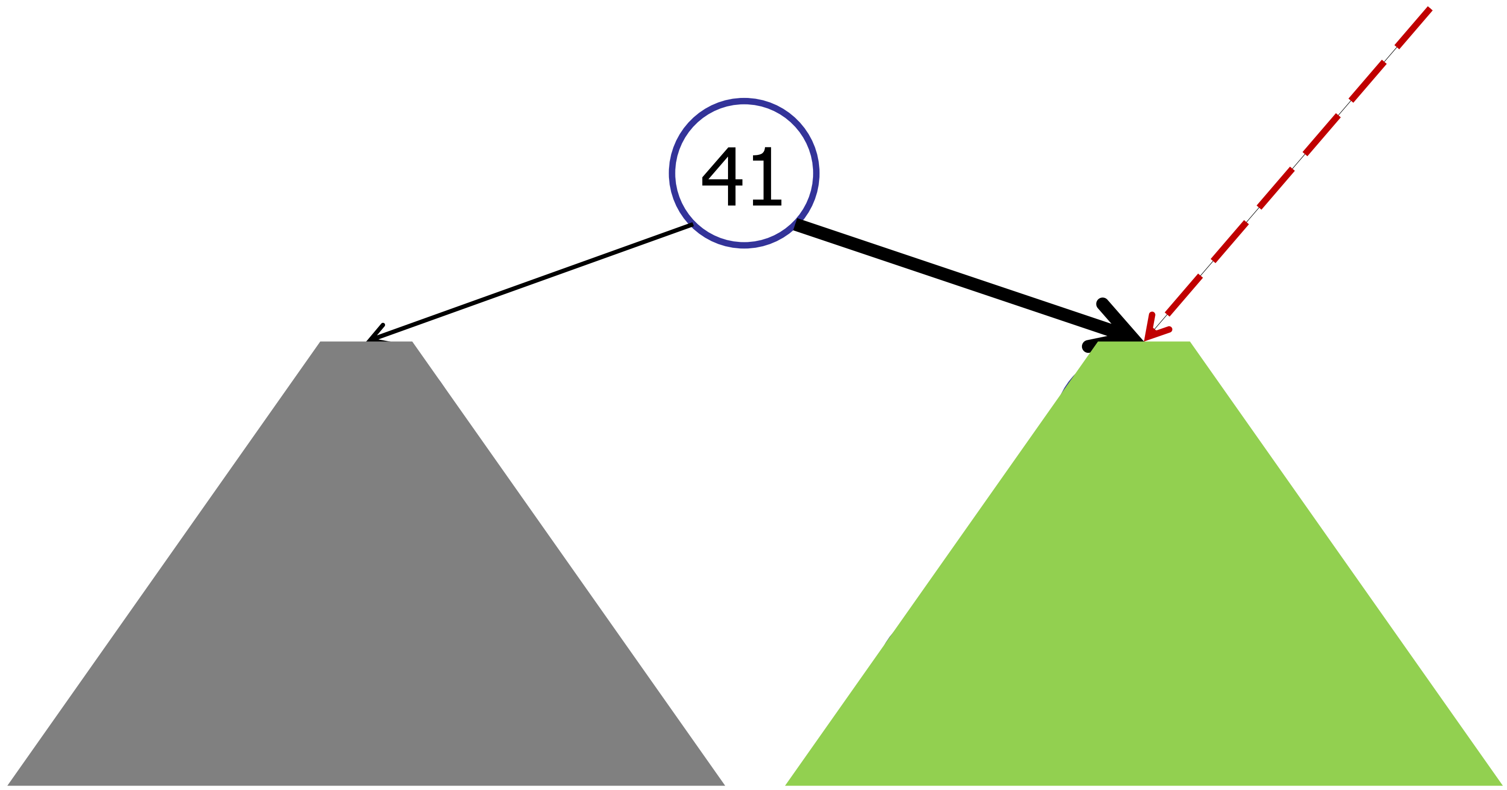
Binary Search Trees

Search for the maximum key:



Binary Search Trees

Search for maximum key



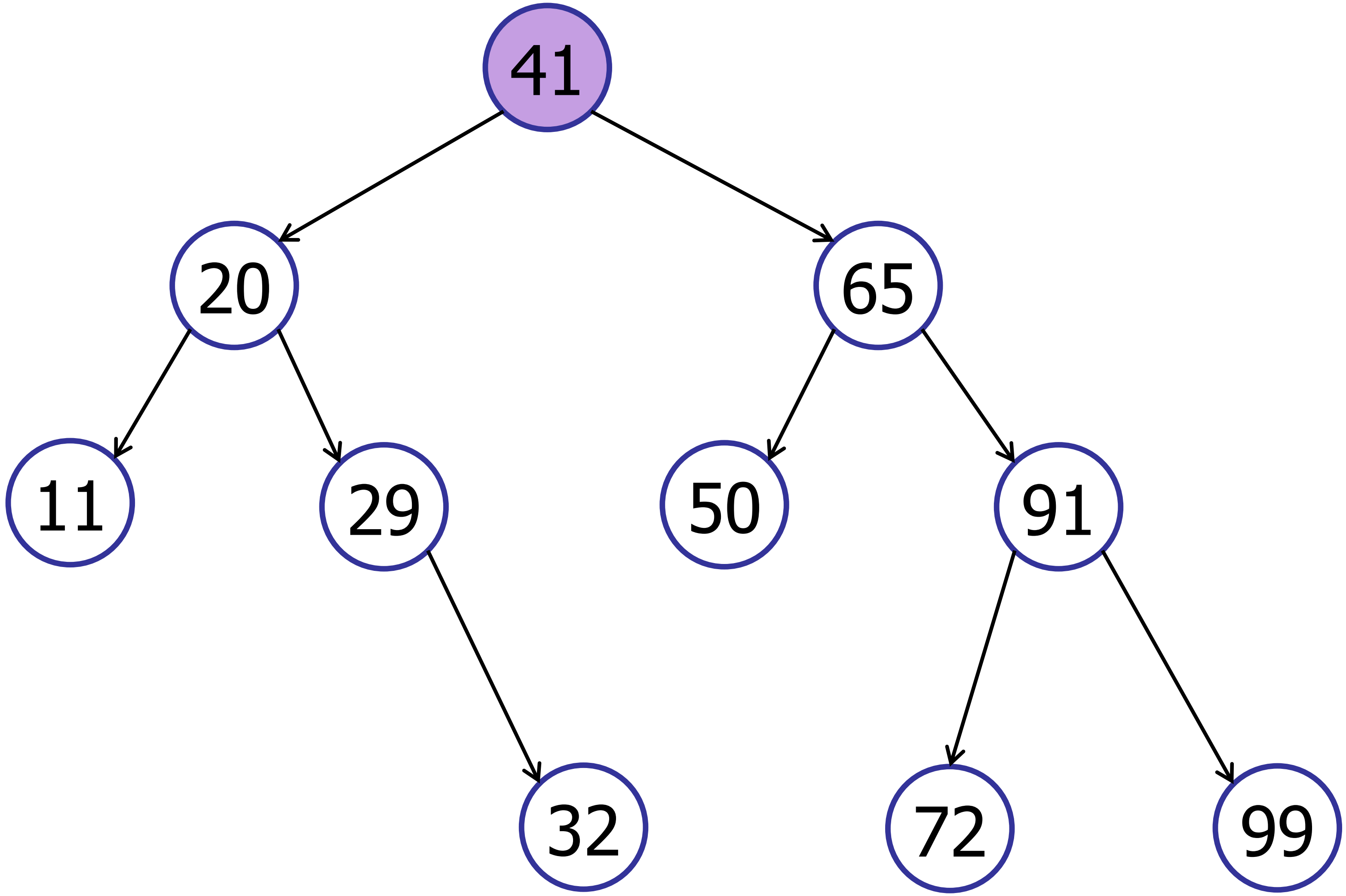
Binary Tree

Searching for the maximum key

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

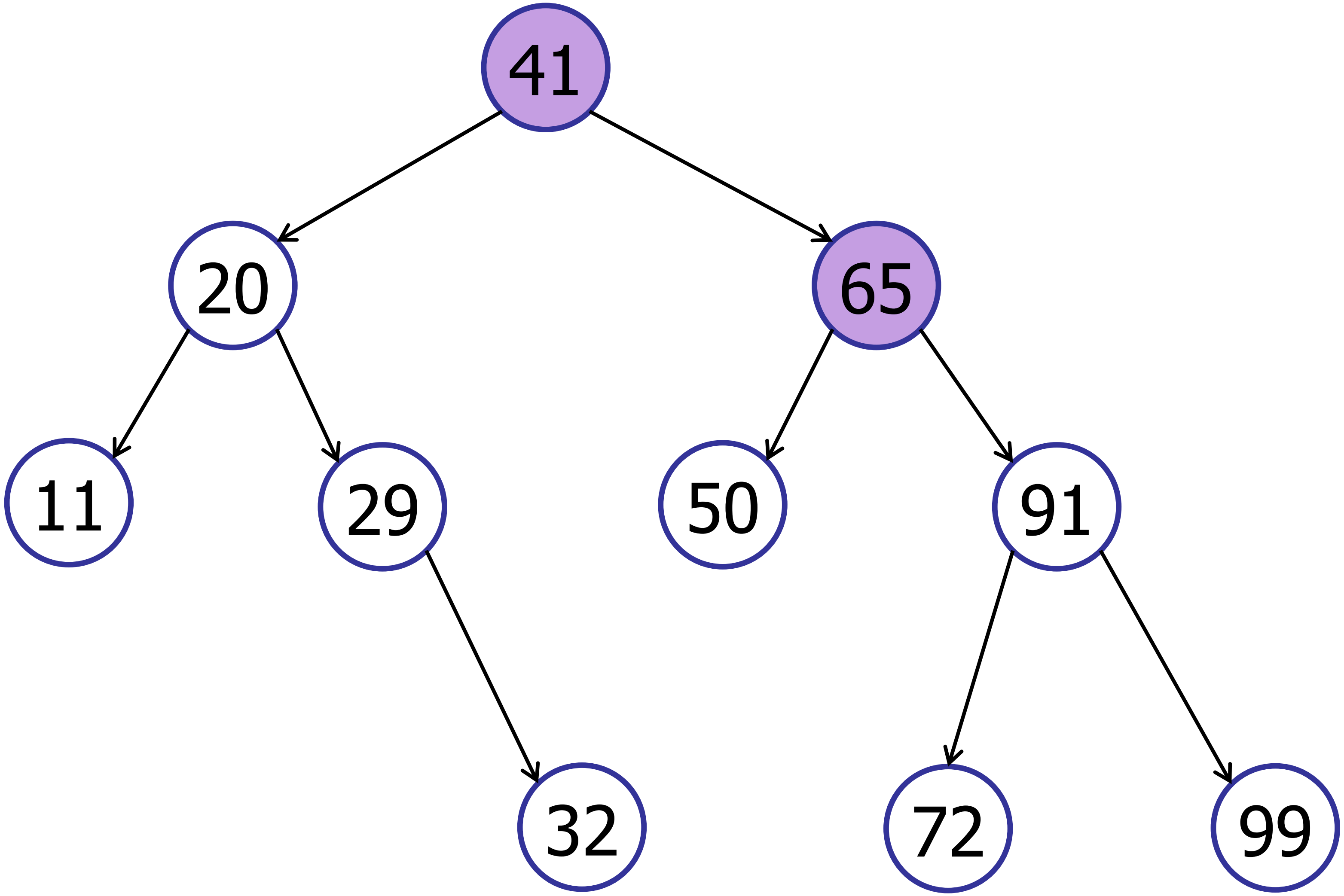
Binary Search Trees

searchMax()



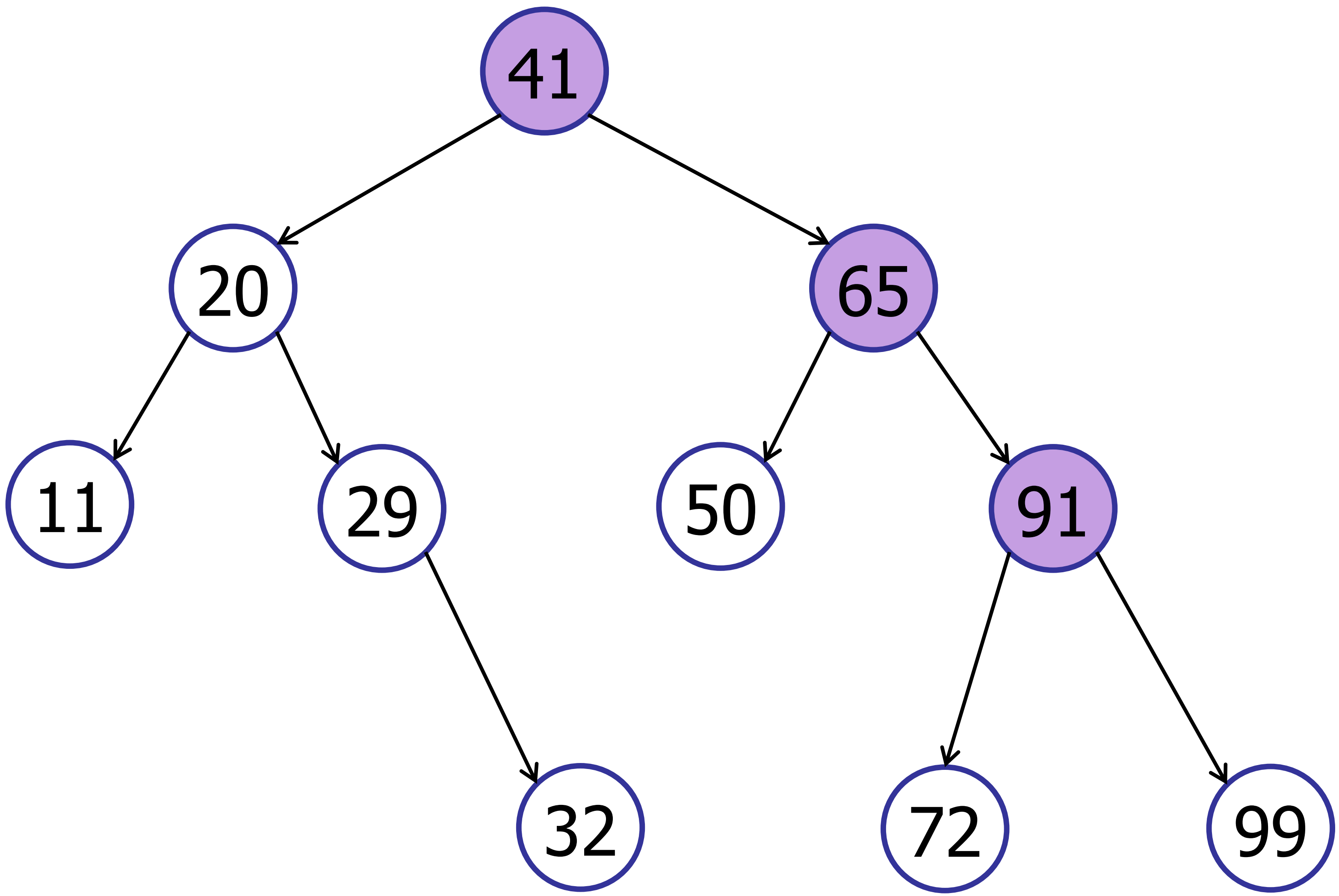
Binary Search Trees

searchMax()



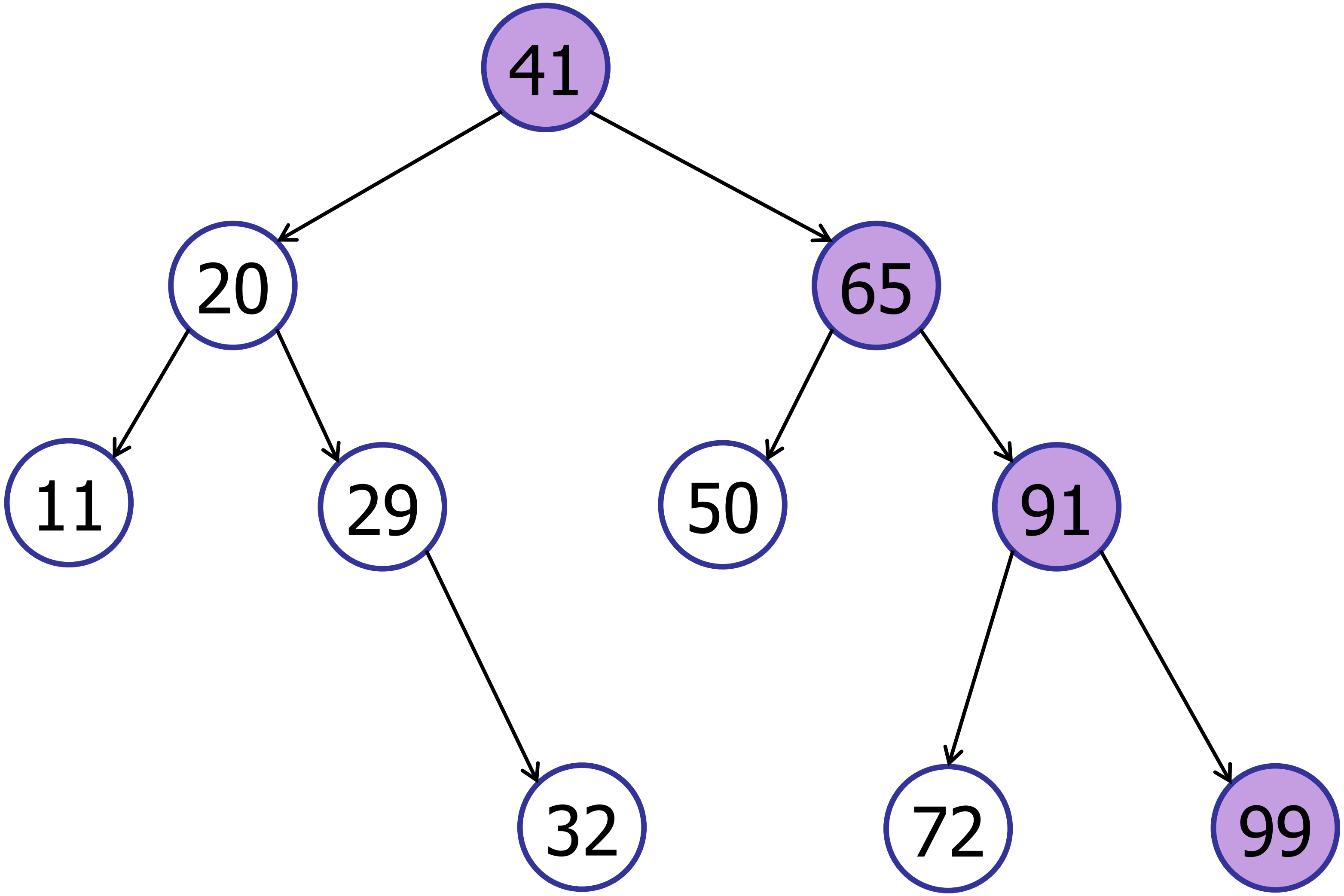
Binary Search Trees

searchMax()



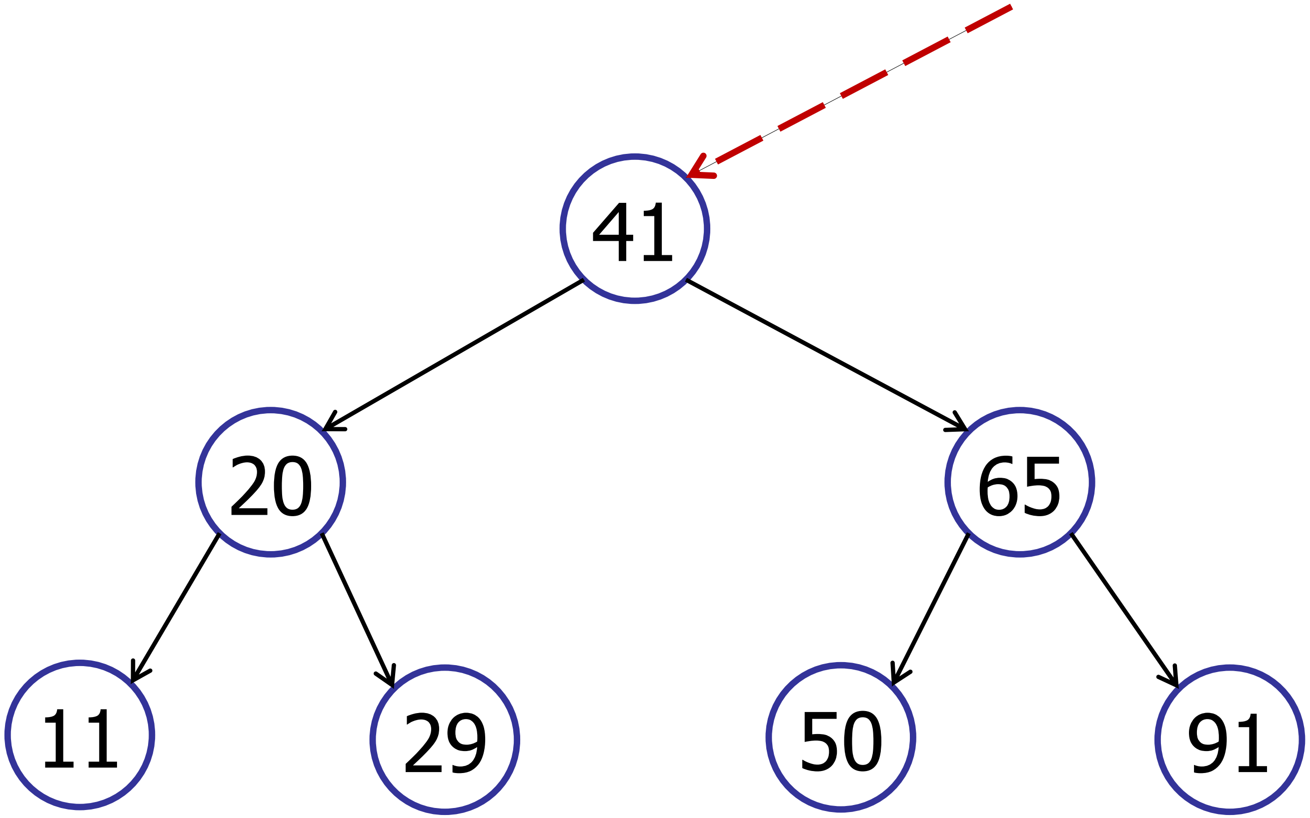
Binary Search Trees

searchMax()



Binary Search Trees

Search for the minimum key:



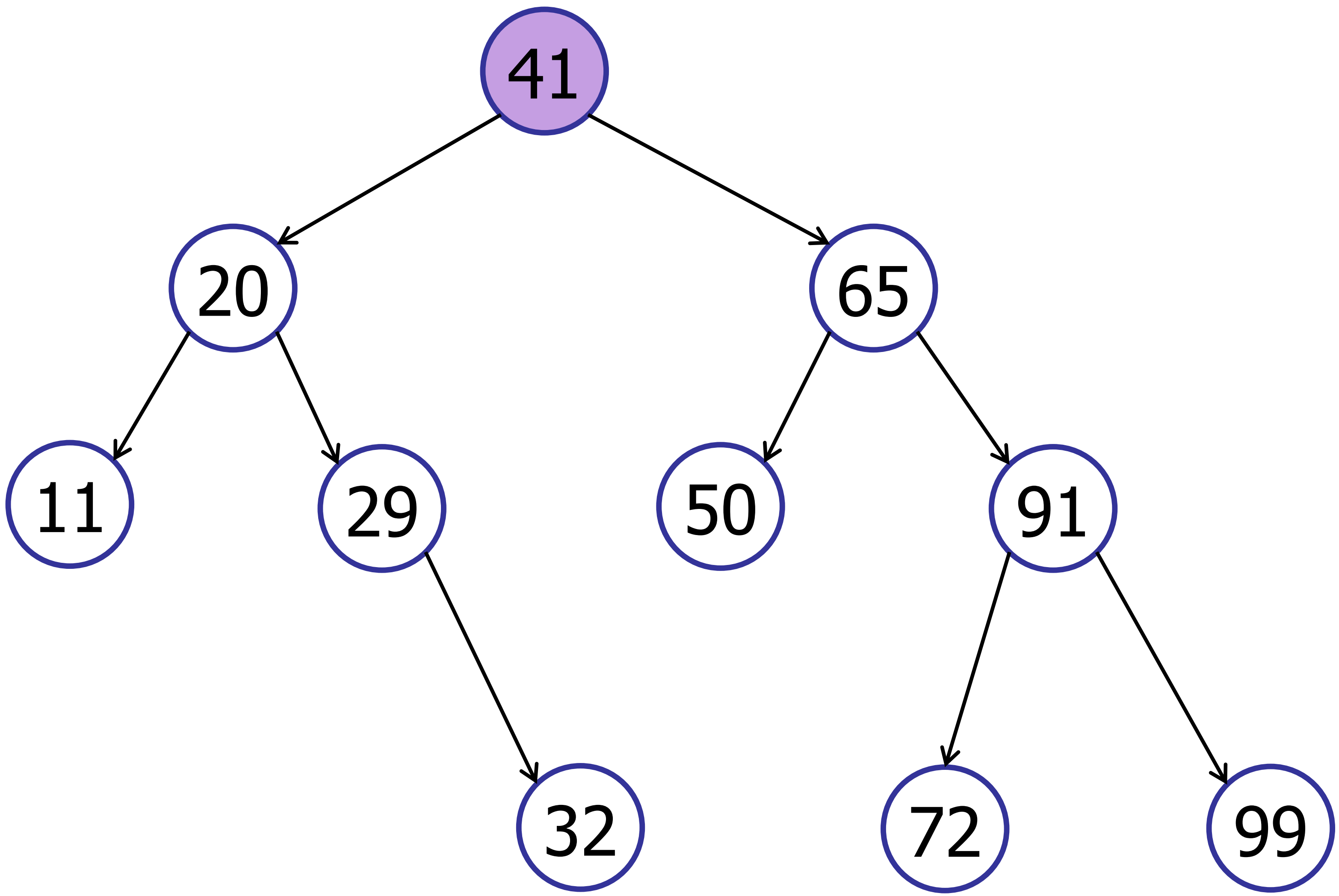
Binary Tree

Searching for the minimum key

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

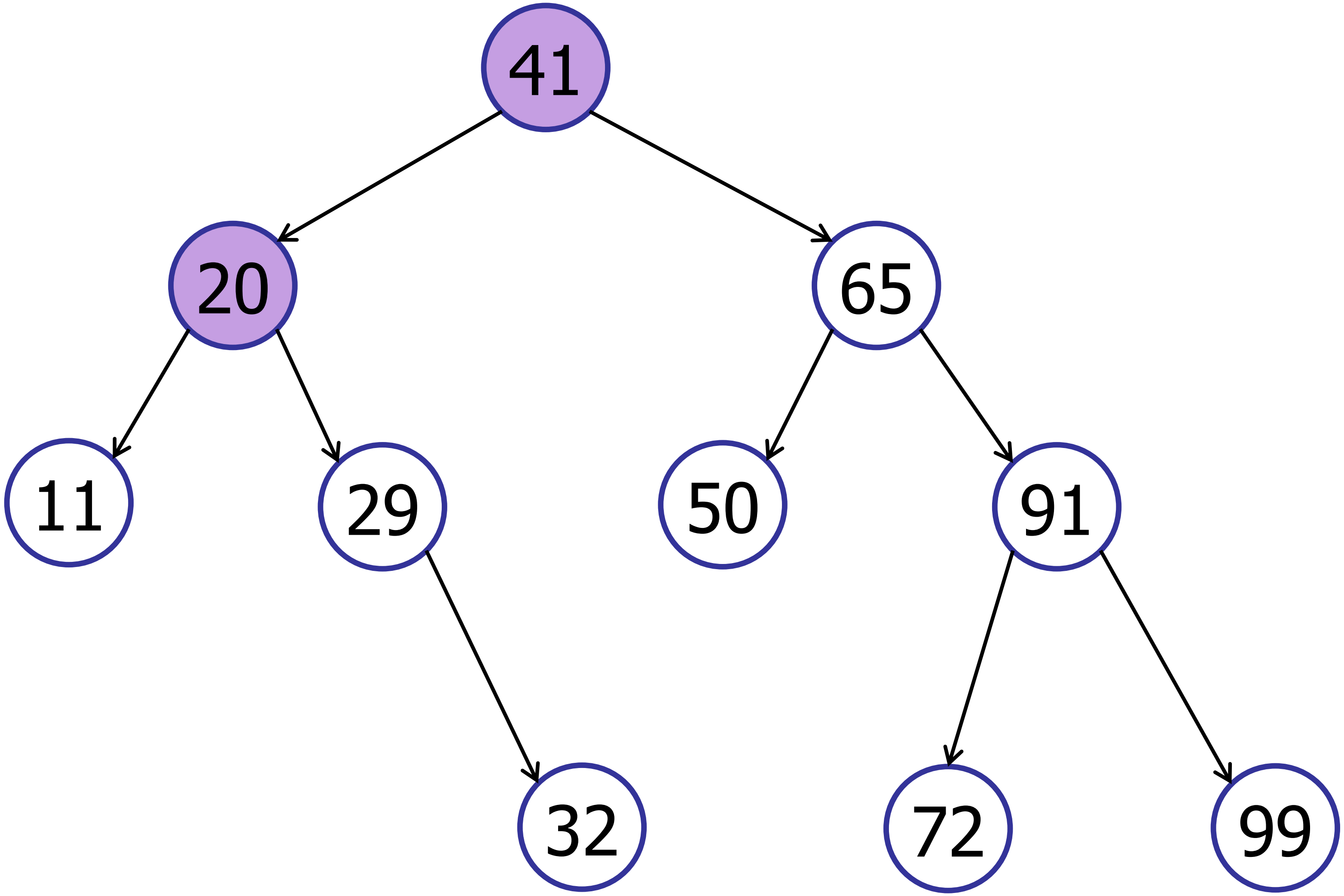
Binary Search Trees

searchMin()



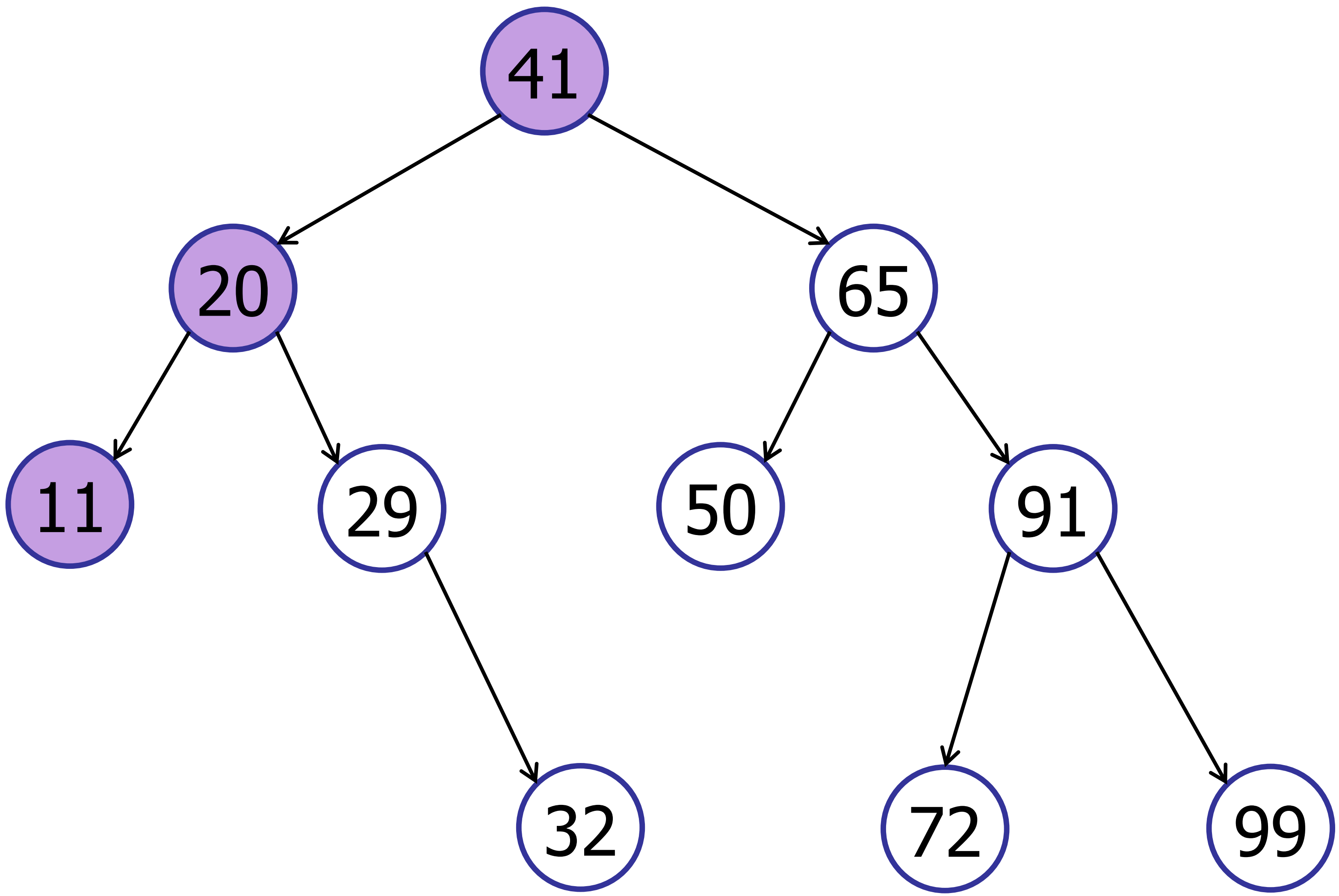
Binary Search Trees

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

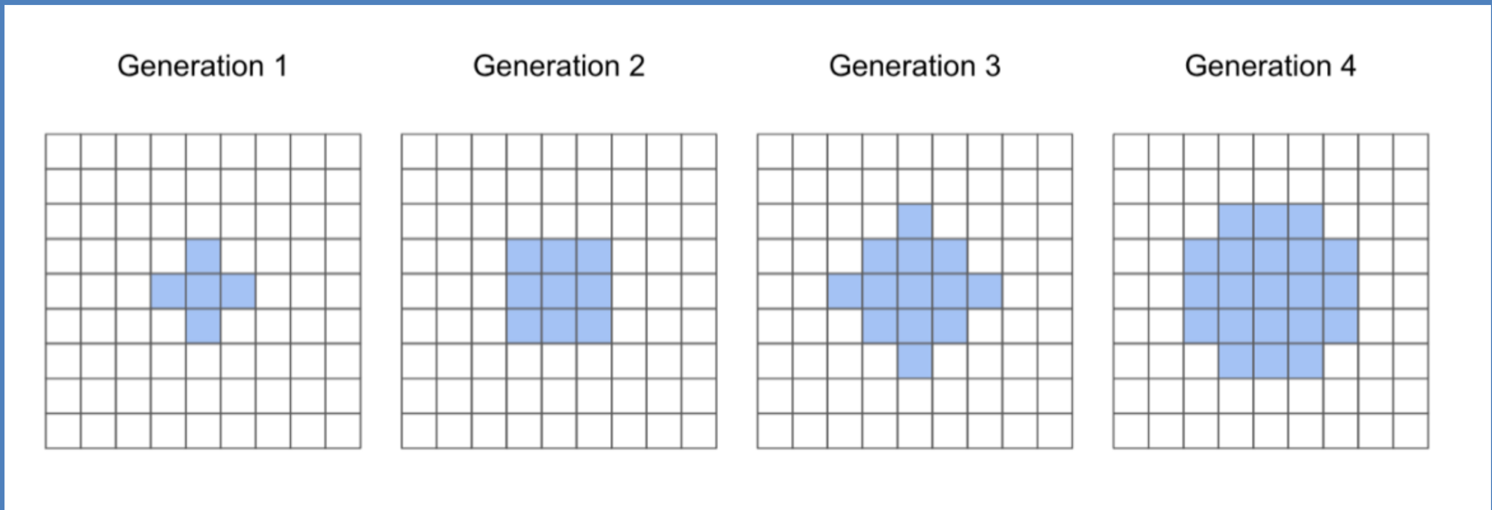
Puzzle of the Week: Squares

(Courtesy: Riddler)

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)?



Binary Search Trees

1. Terminology and Definitions

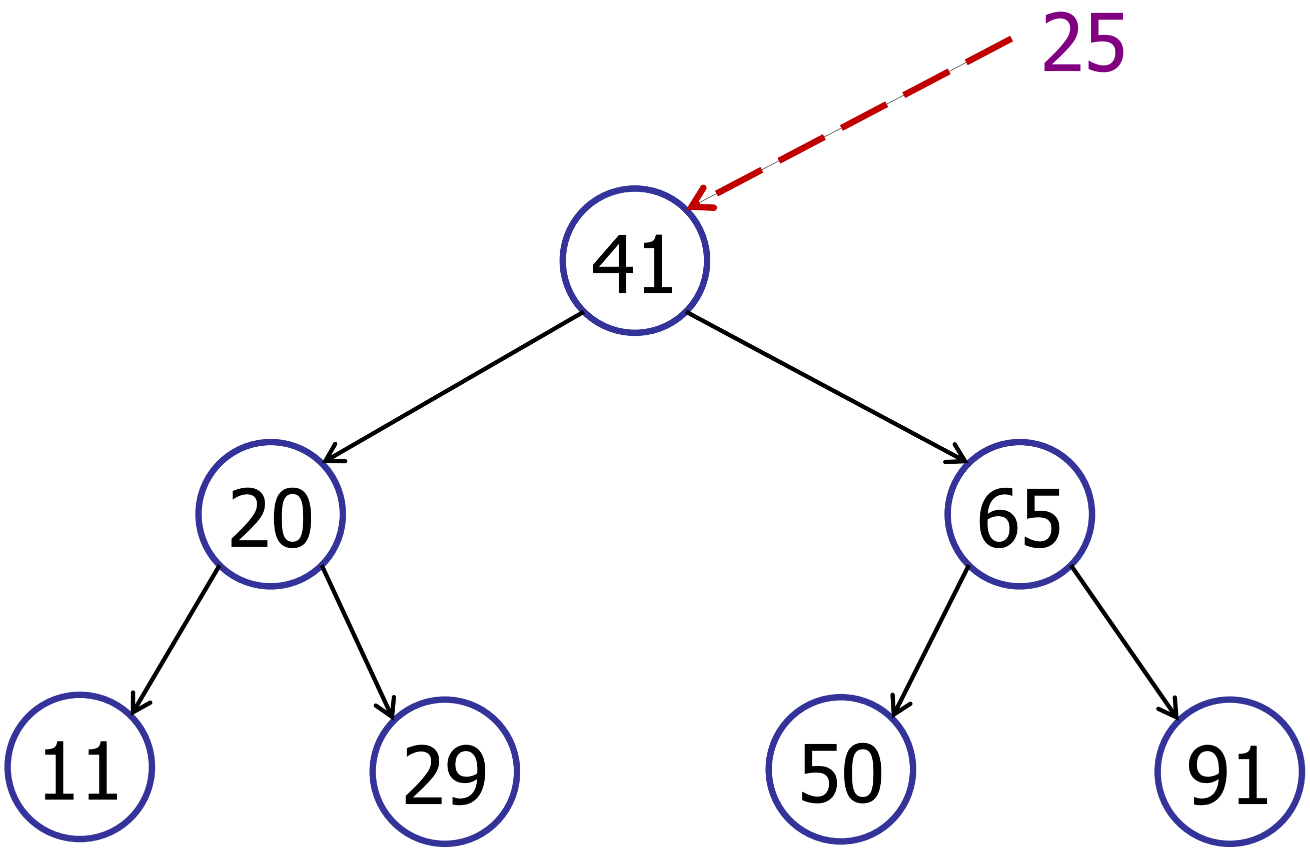
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

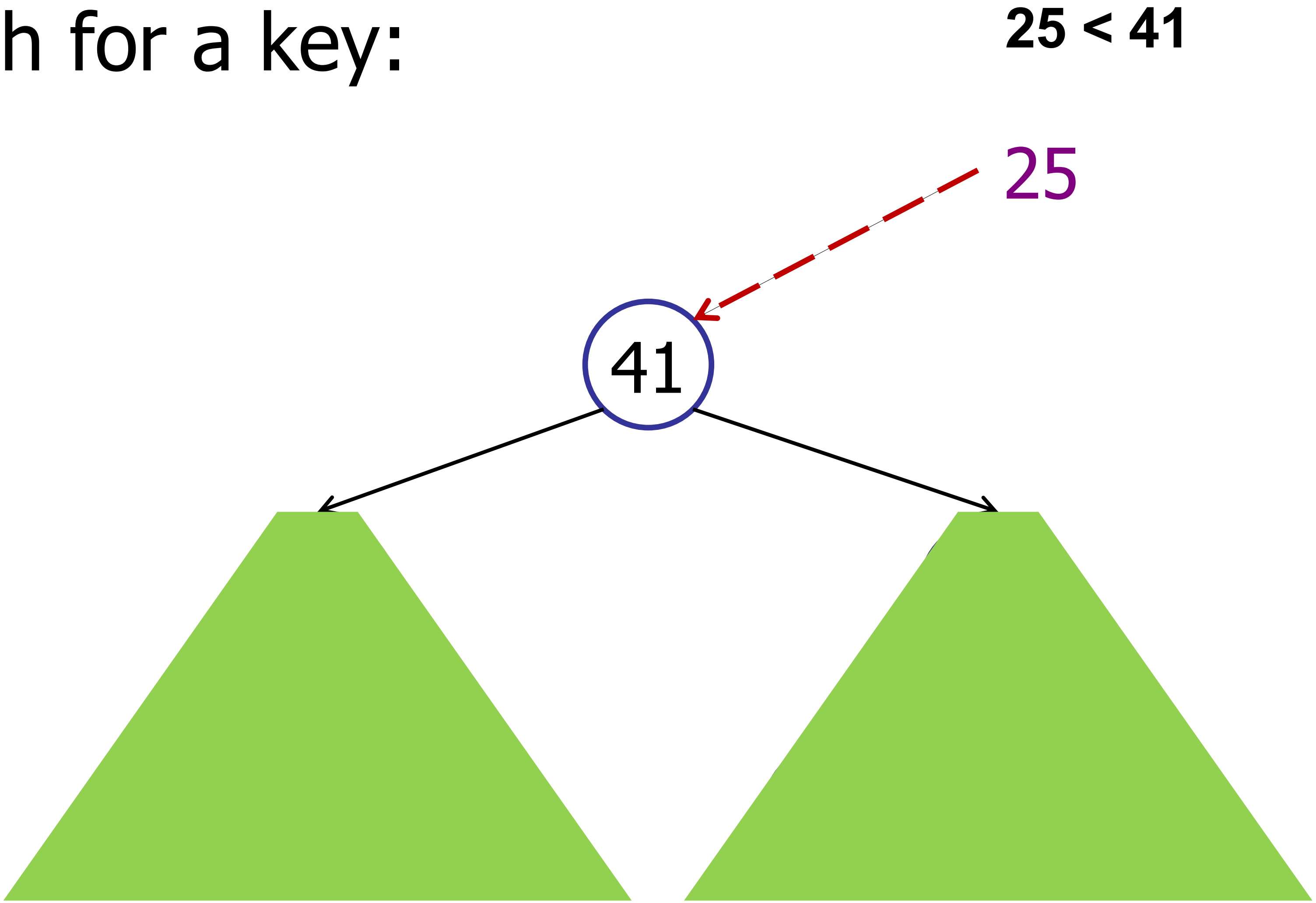
Binary Search Trees

Search for a key:



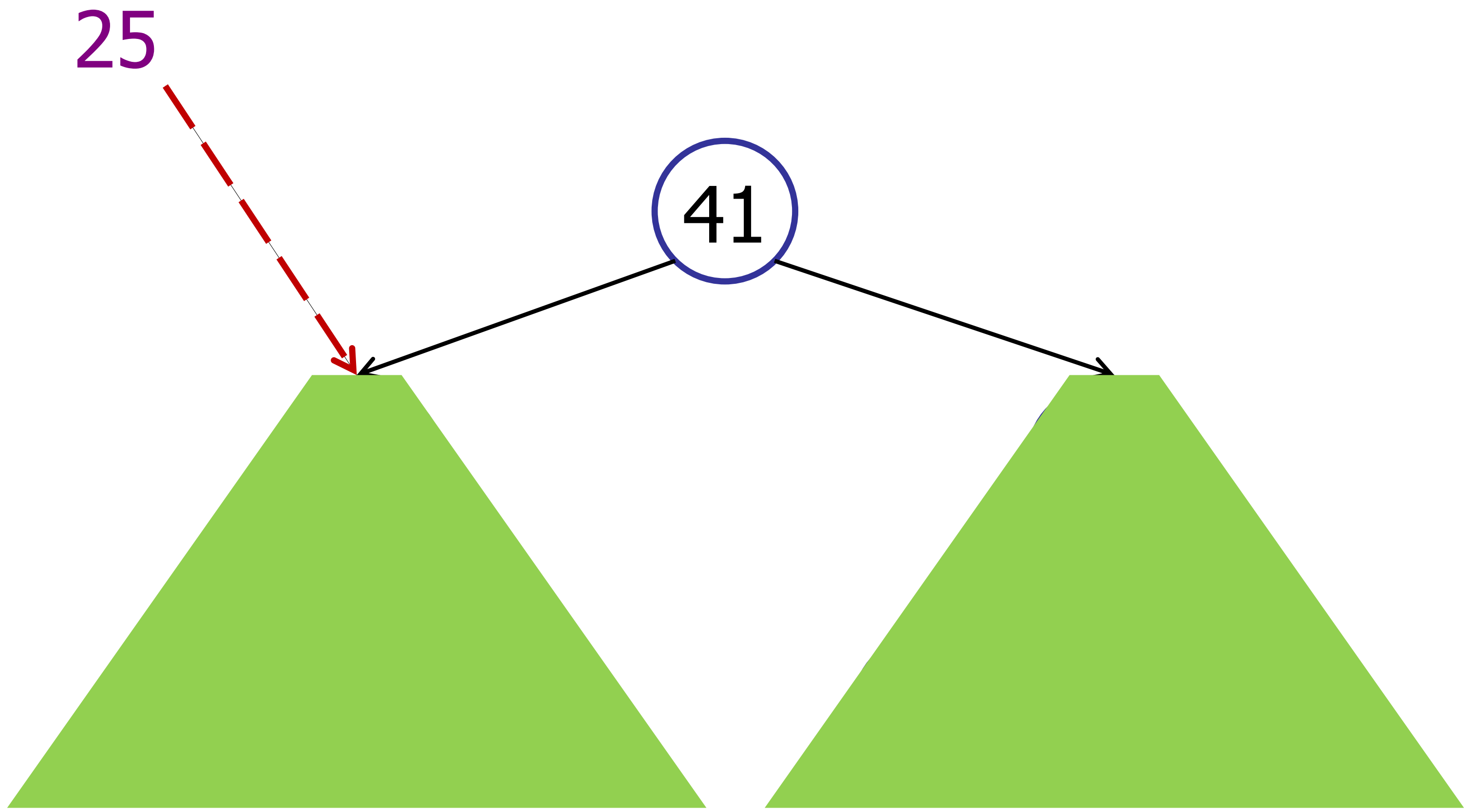
Binary Search Trees

Search for a key:



Binary Search Trees

Search for a key:



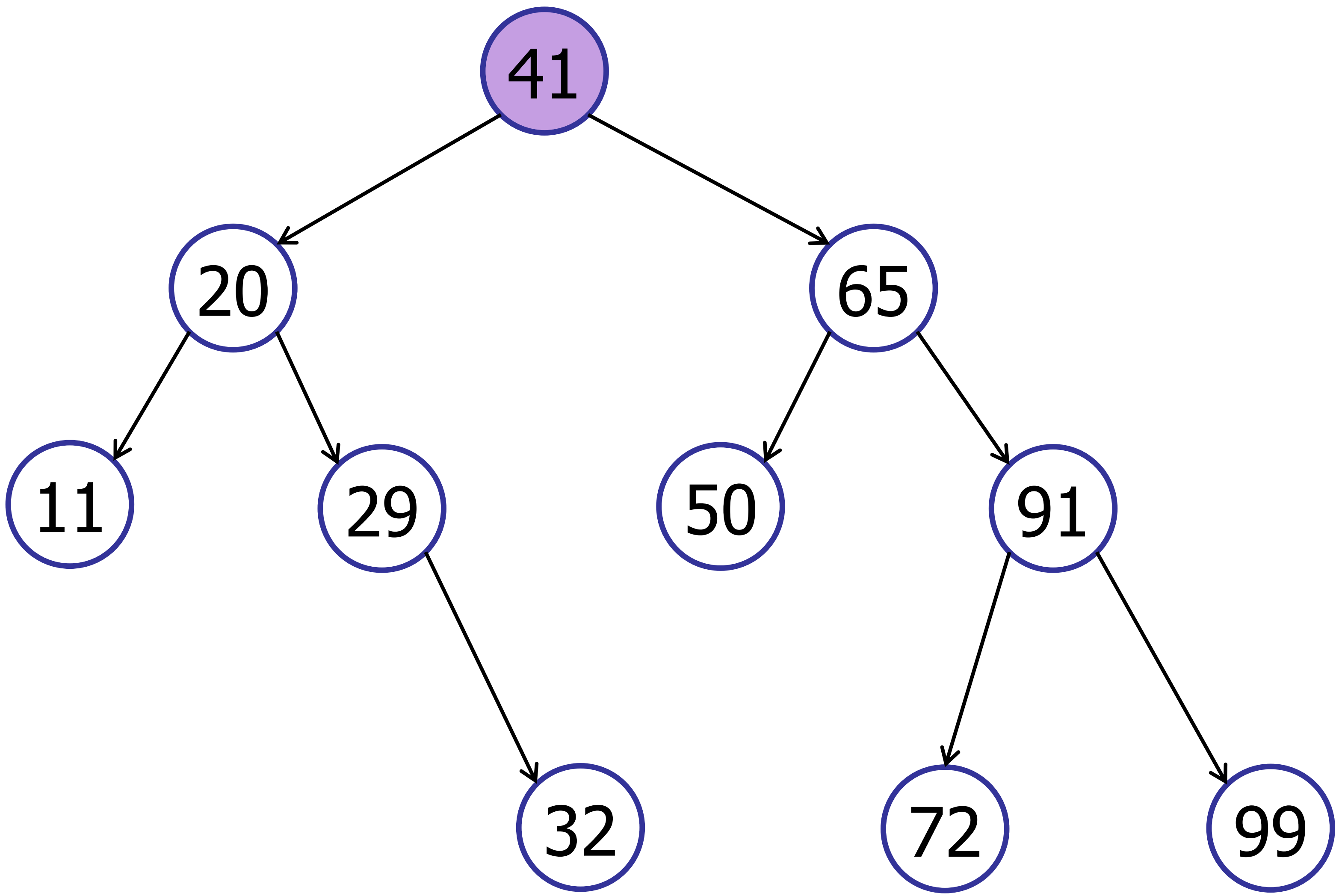
Binary Tree

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!
}
```

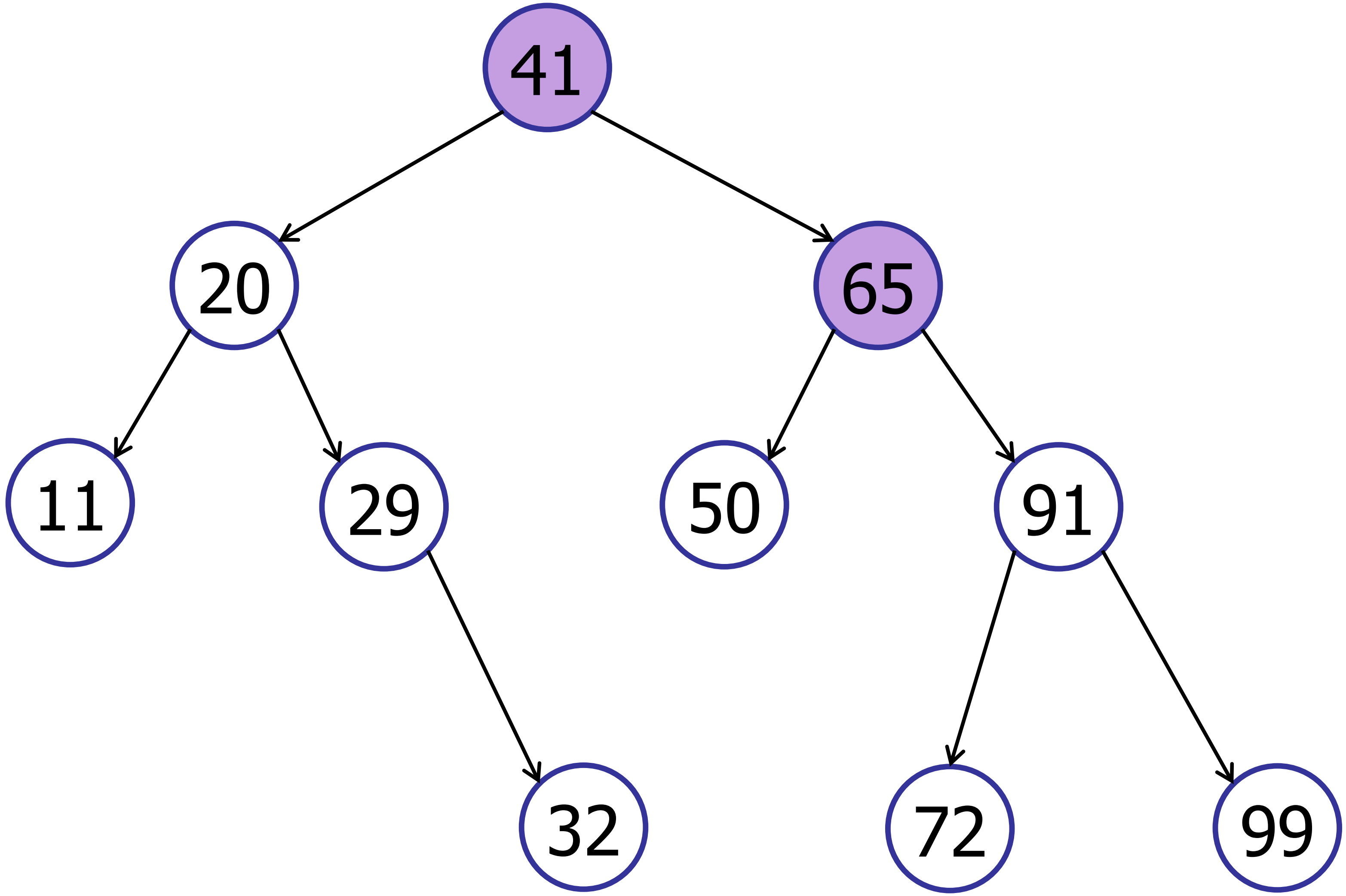
Binary Search Trees

search(72)



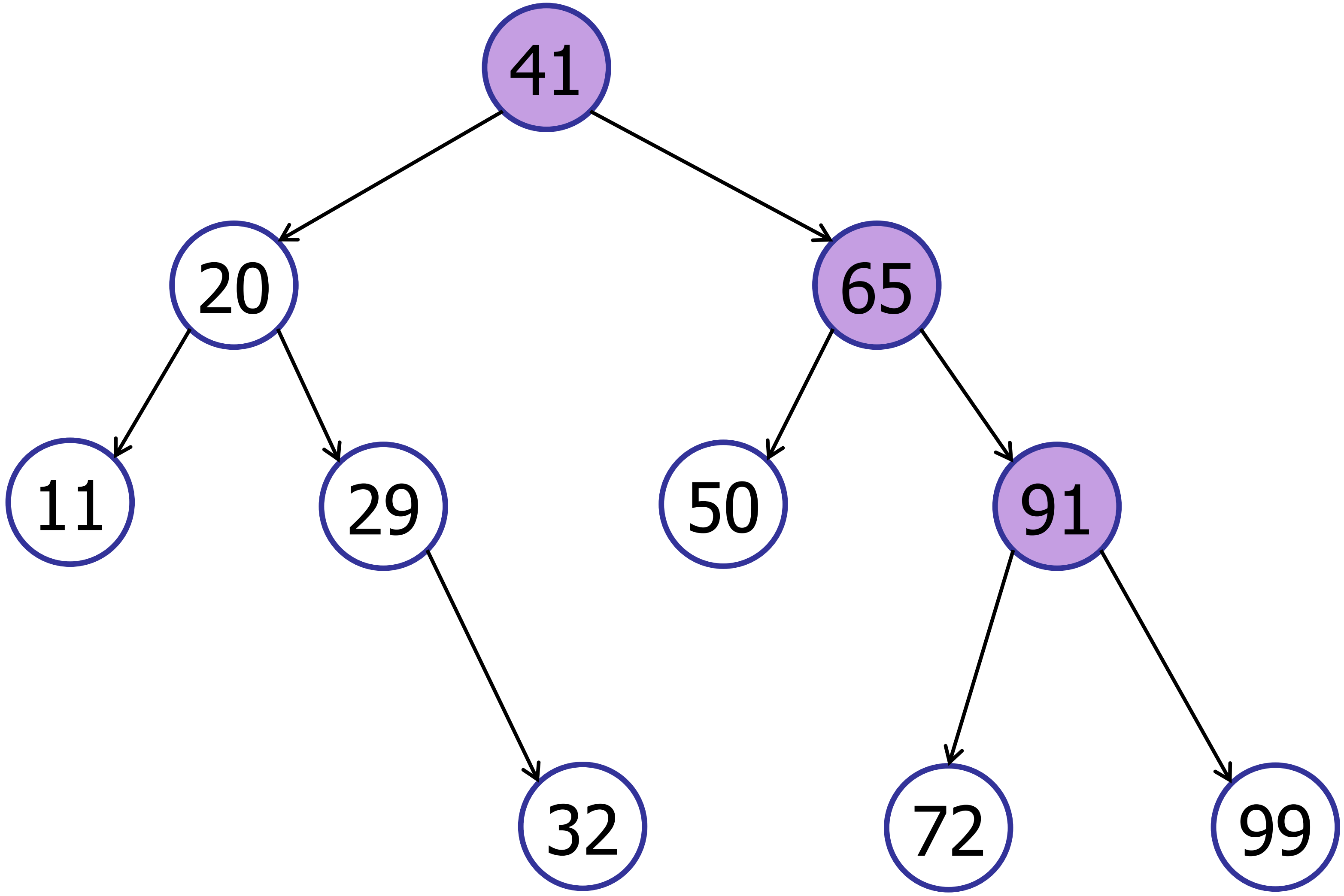
Binary Search Trees

search(72)



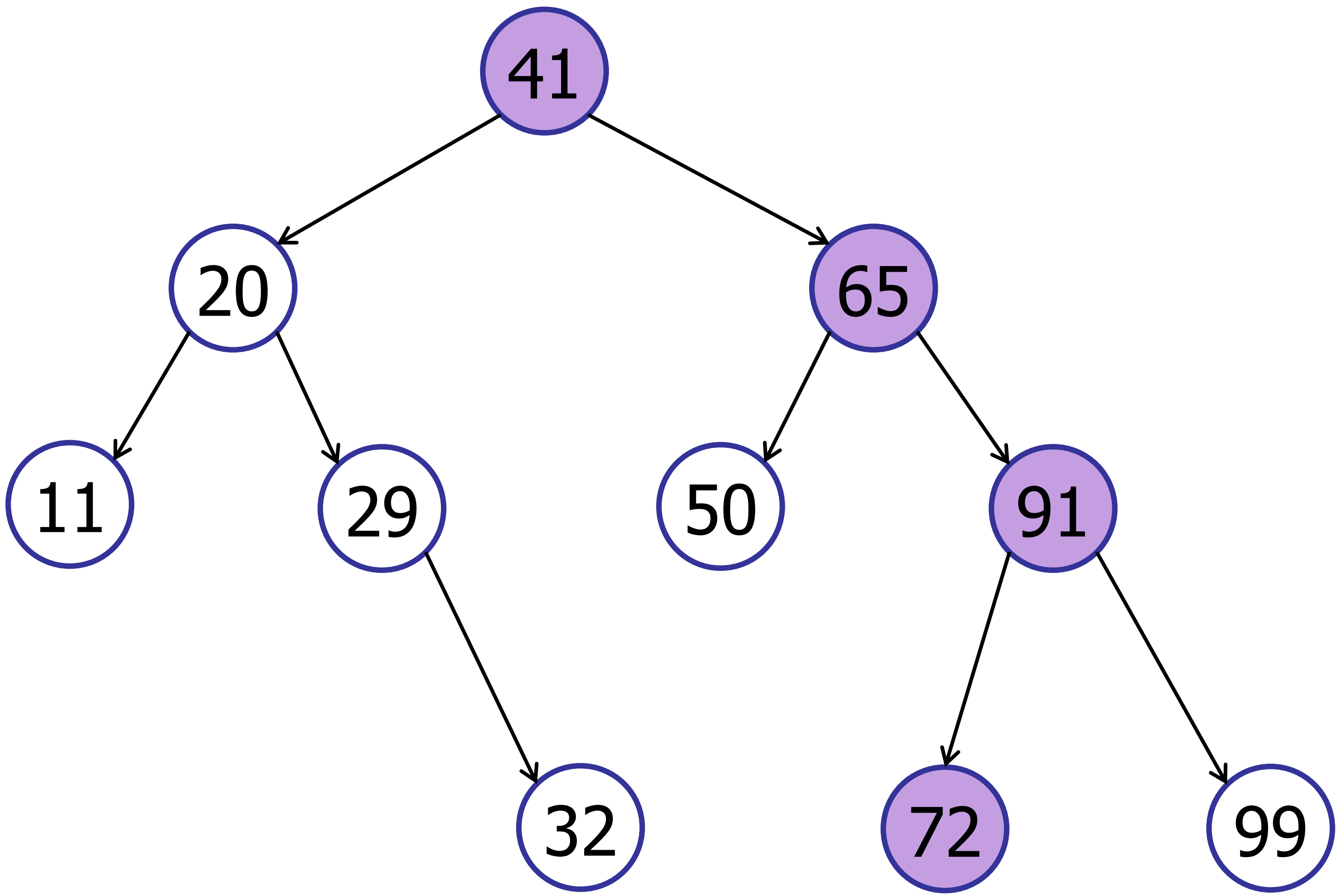
Binary Search Trees

search(72)



Binary Search Trees

search(72)



Binary Search Trees

1. Terminology and Definitions

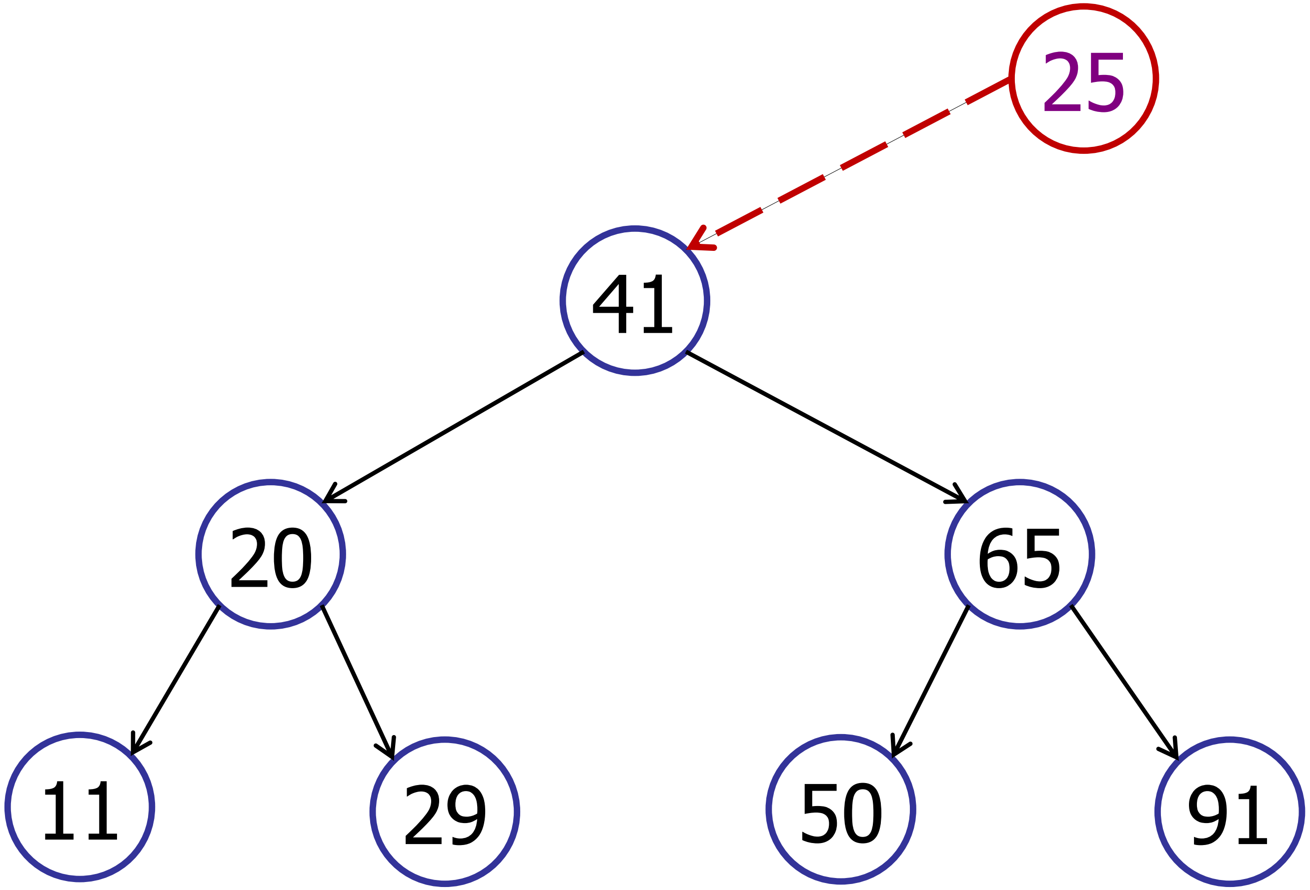
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

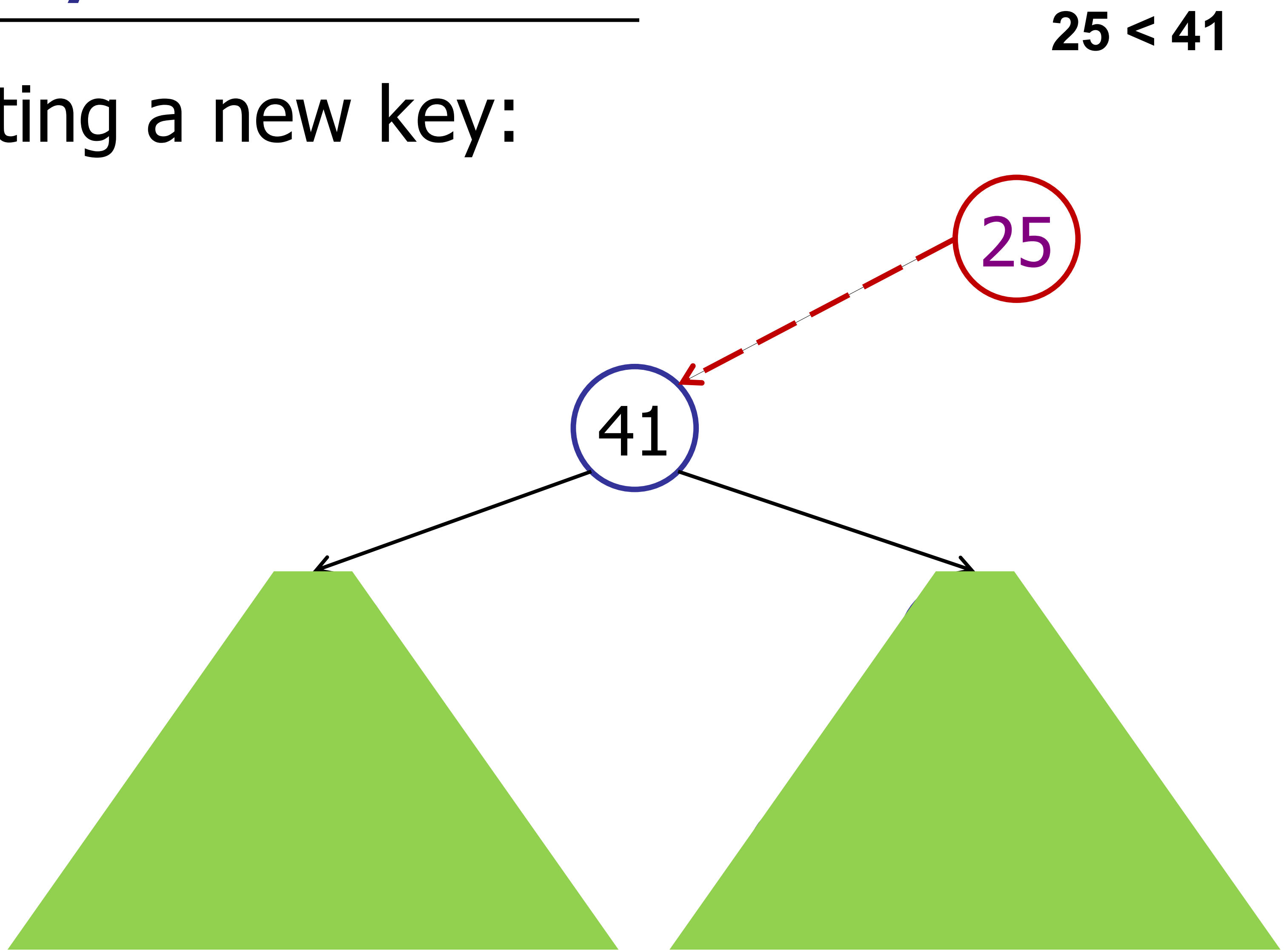
Binary Search Trees

Inserting a new key:



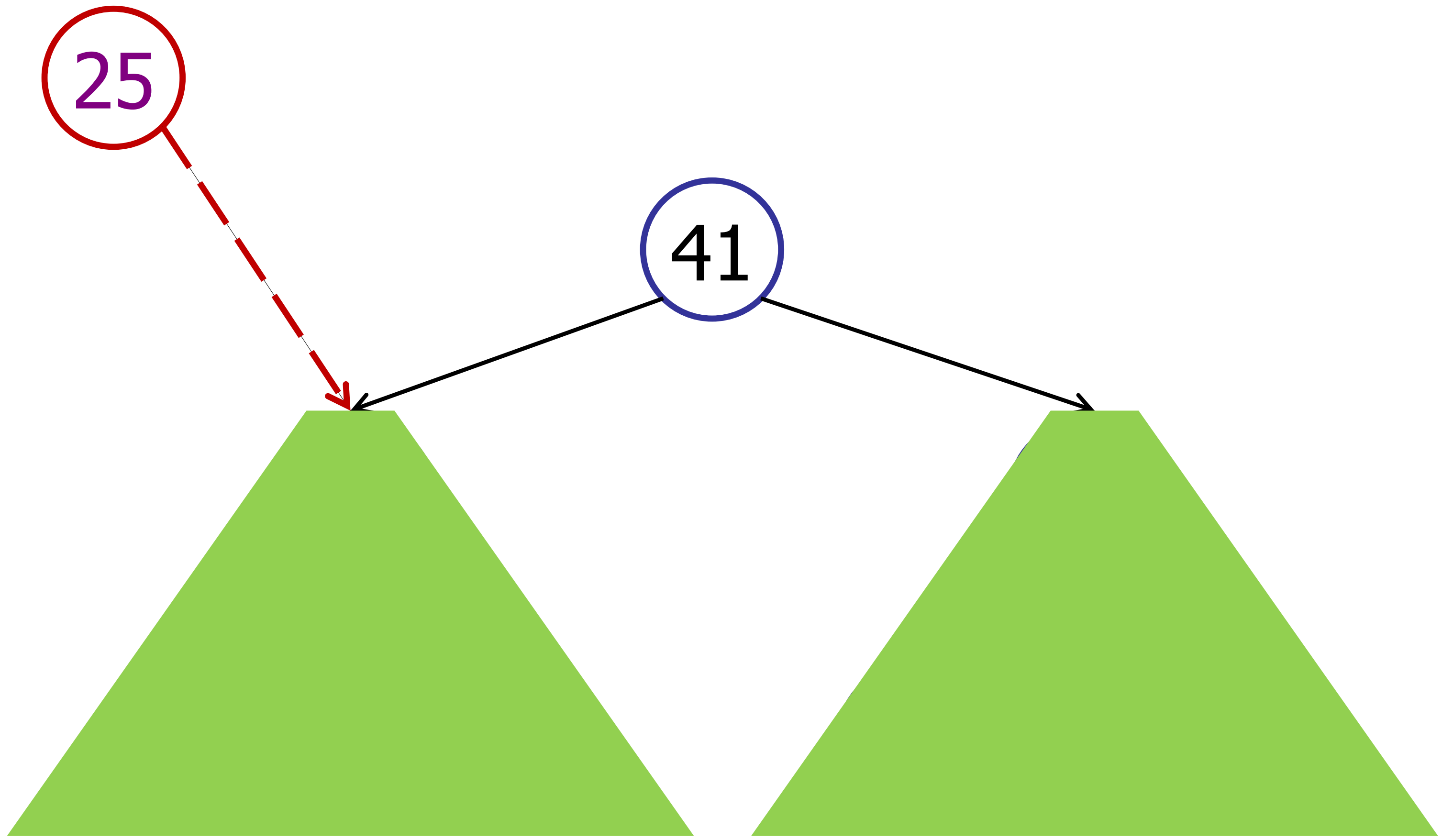
Binary Search Trees

Inserting a new key:



Binary Search Trees

Inserting a new key:



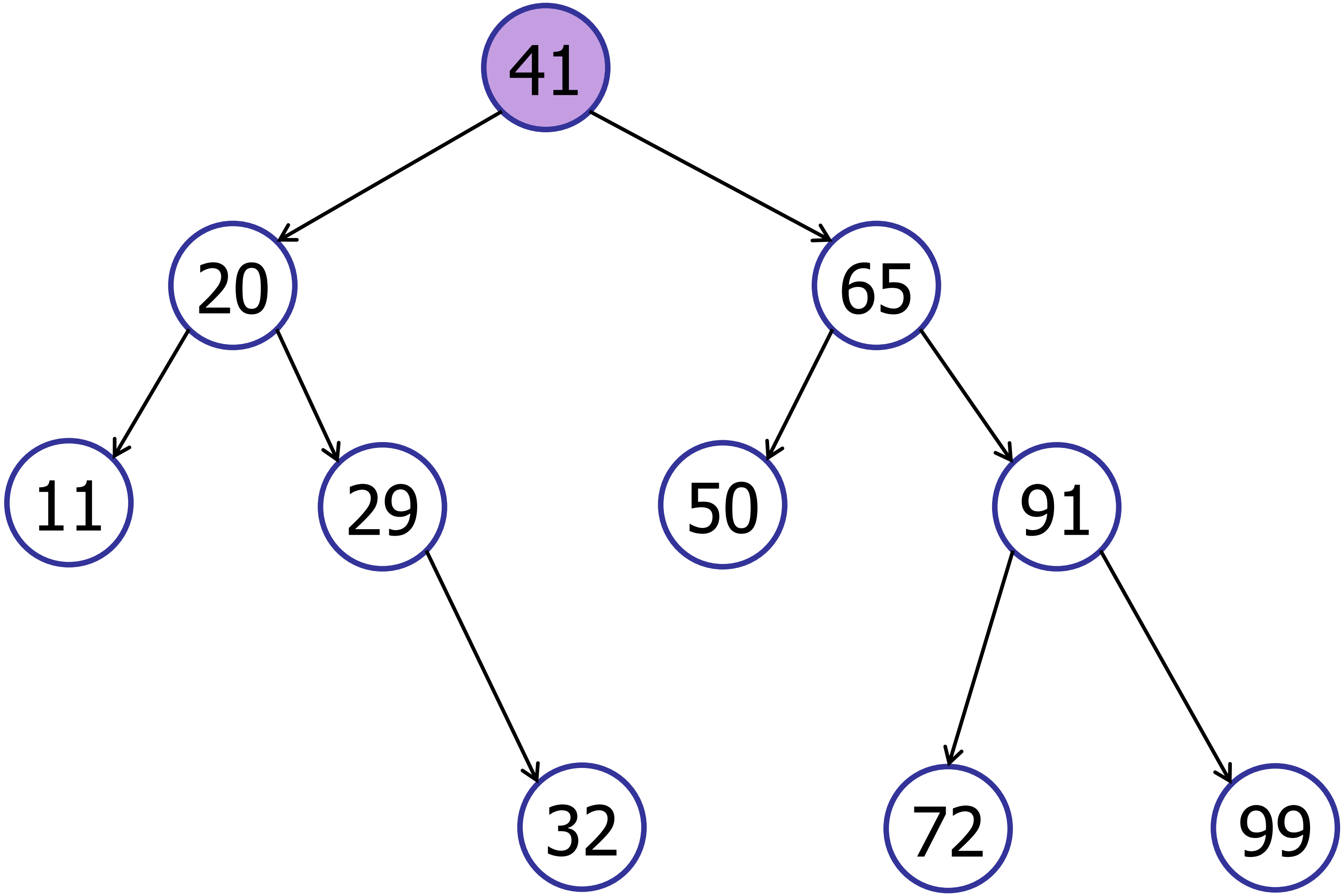
Binary Tree

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!
}
```

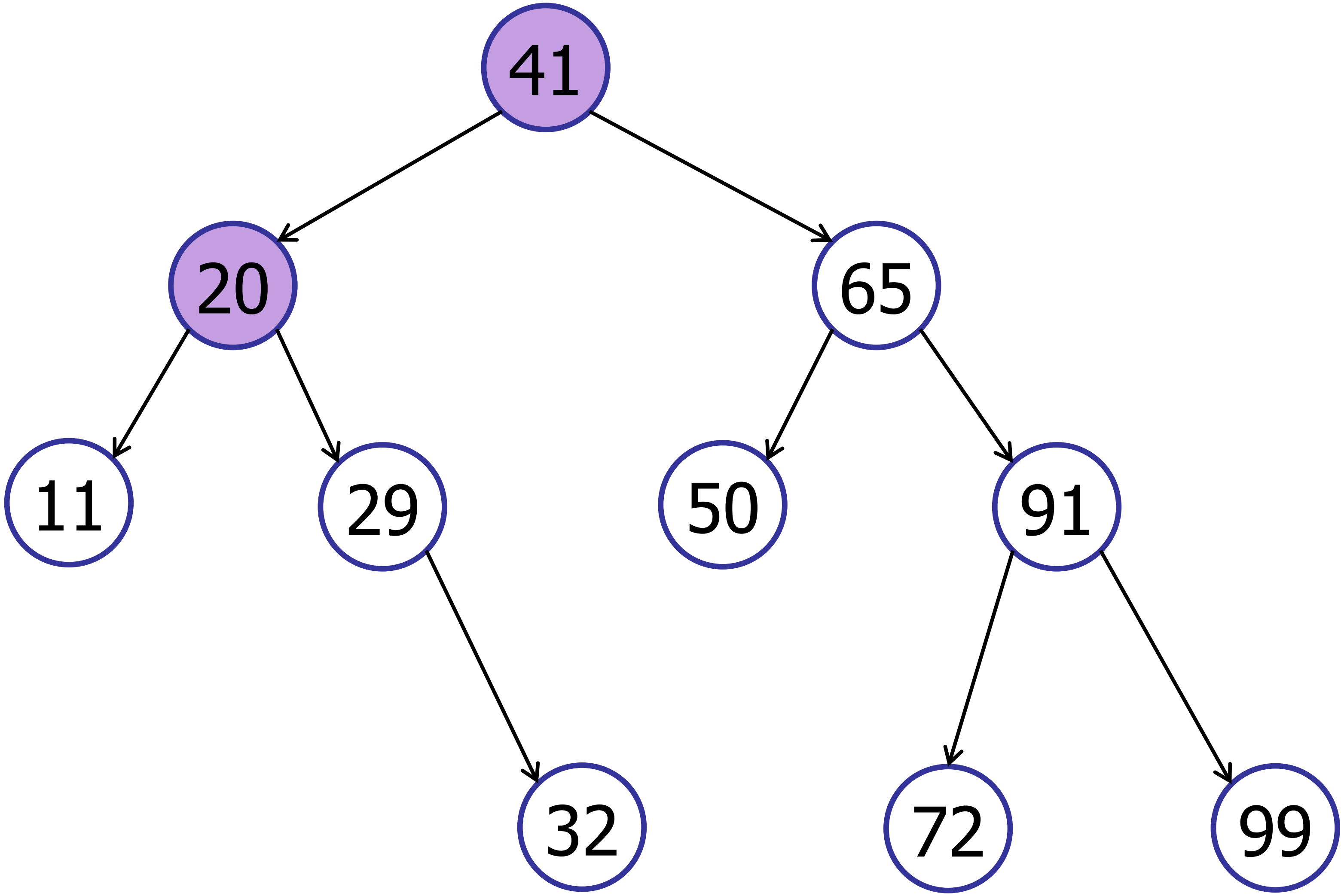
Binary Search Trees

insert(27)



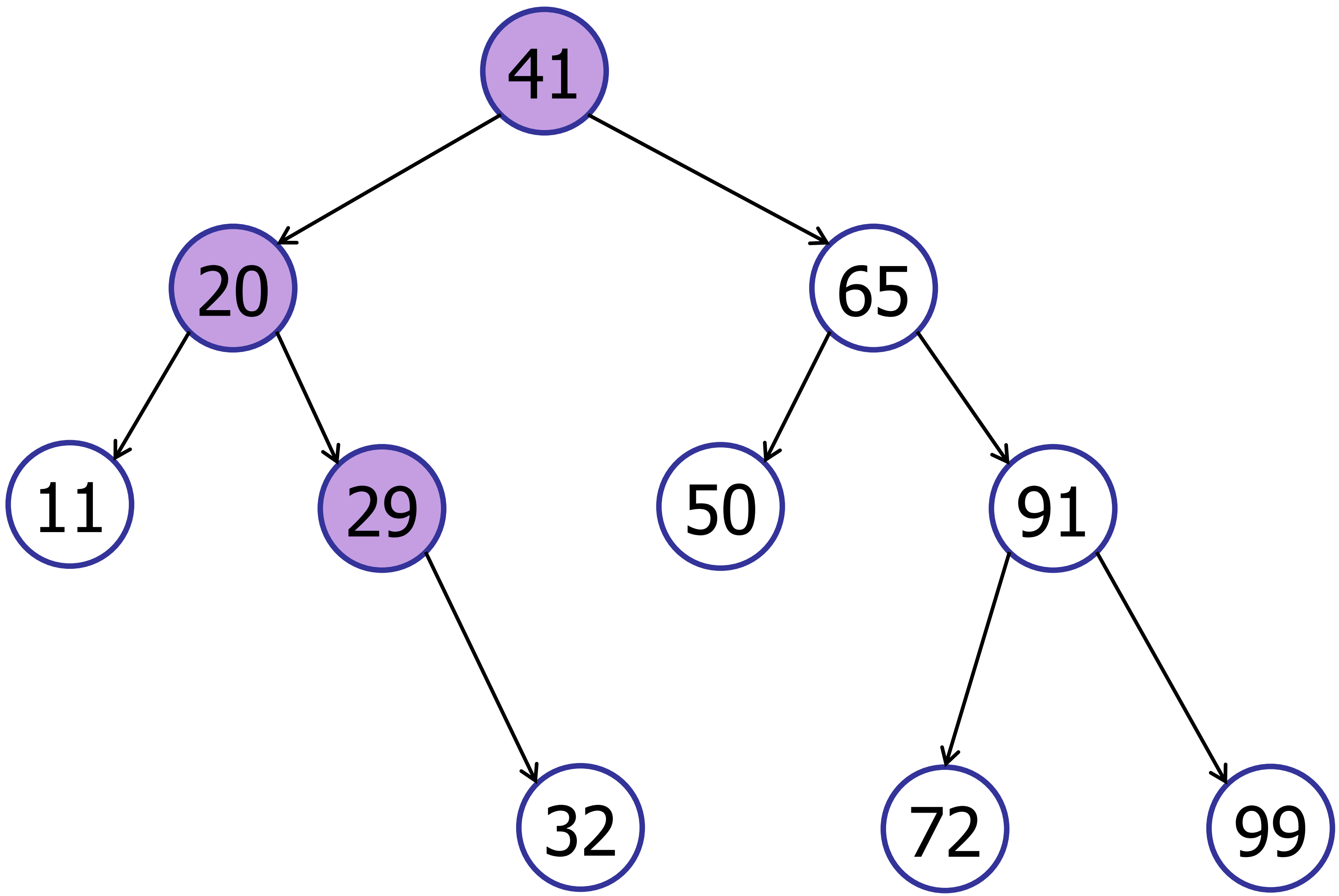
Binary Search Trees

insert(27)



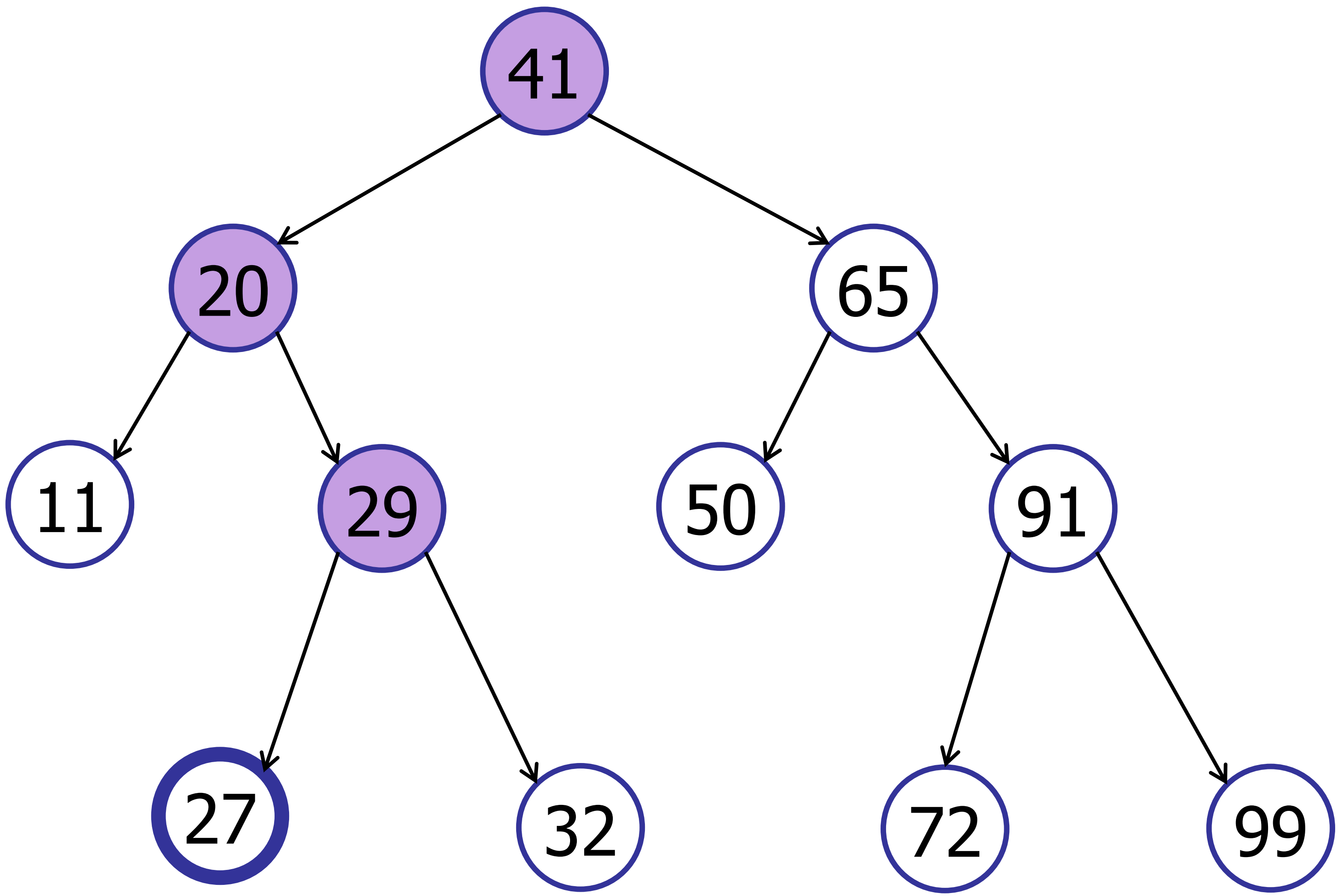
Binary Search Trees

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. $O(2^n)$

ARCHIPELAGO

is open

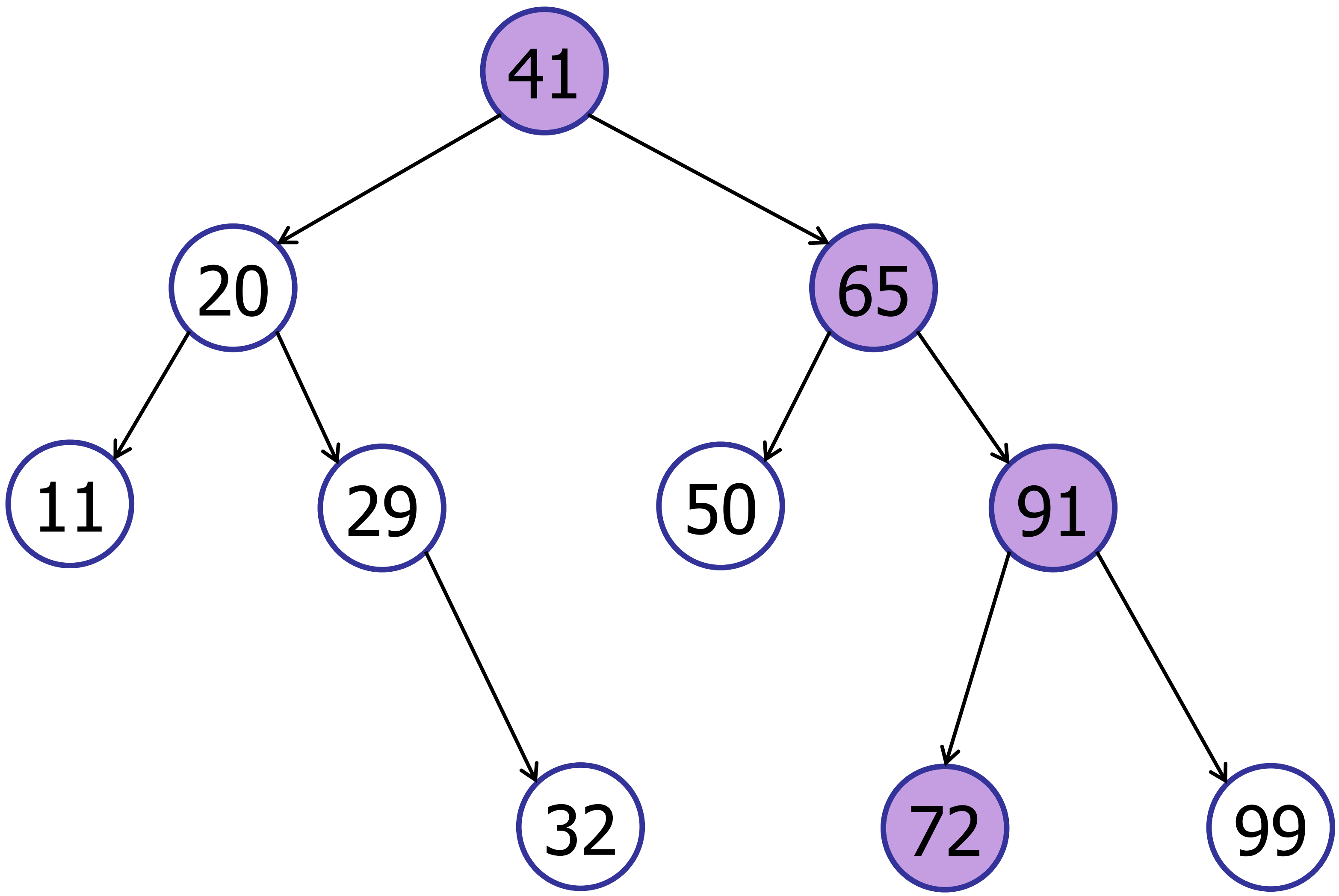
Binary Search Tree

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. $O(2^n)$

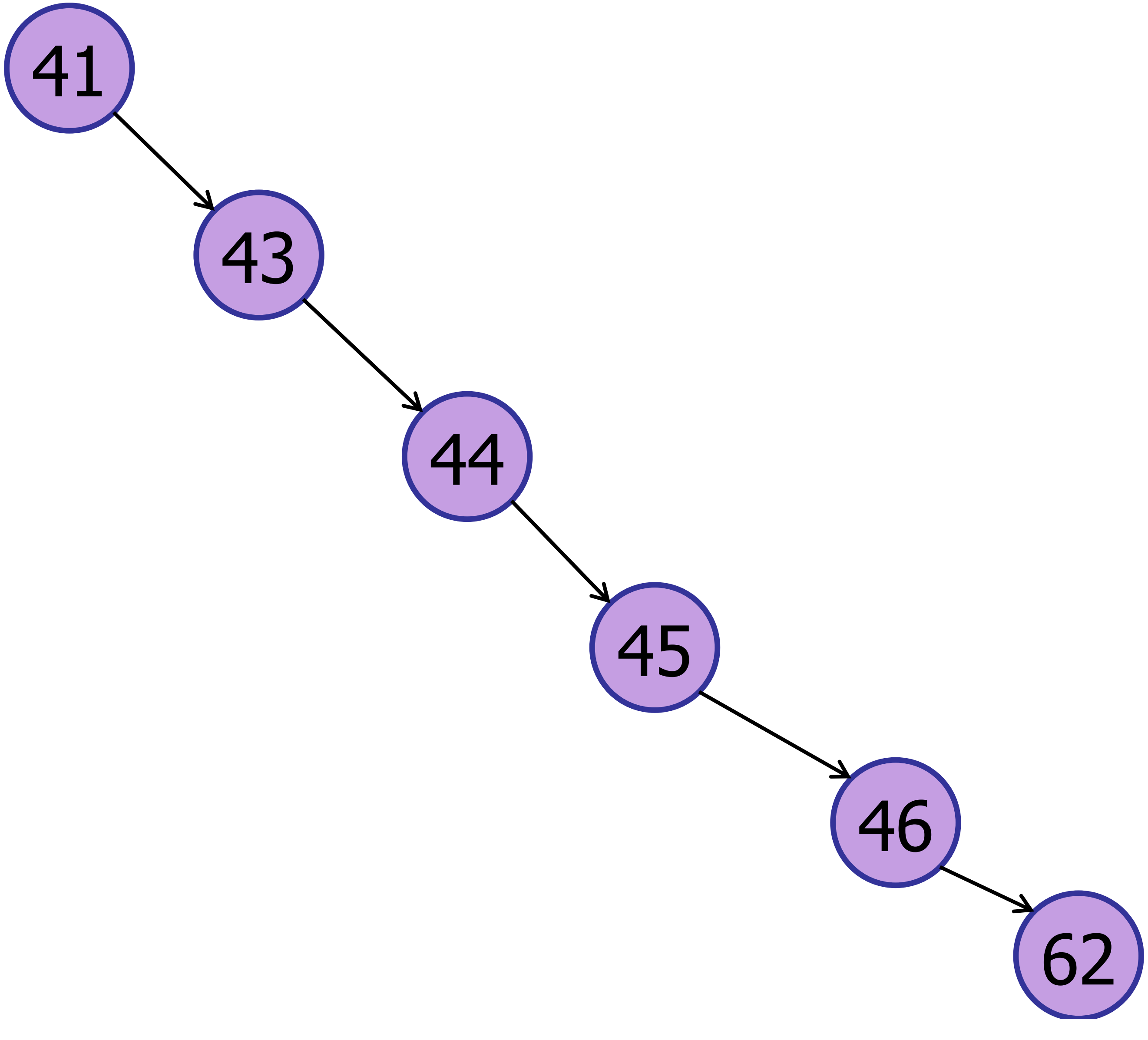
Binary Search Trees

search(72) : O(h)



Binary Search Trees

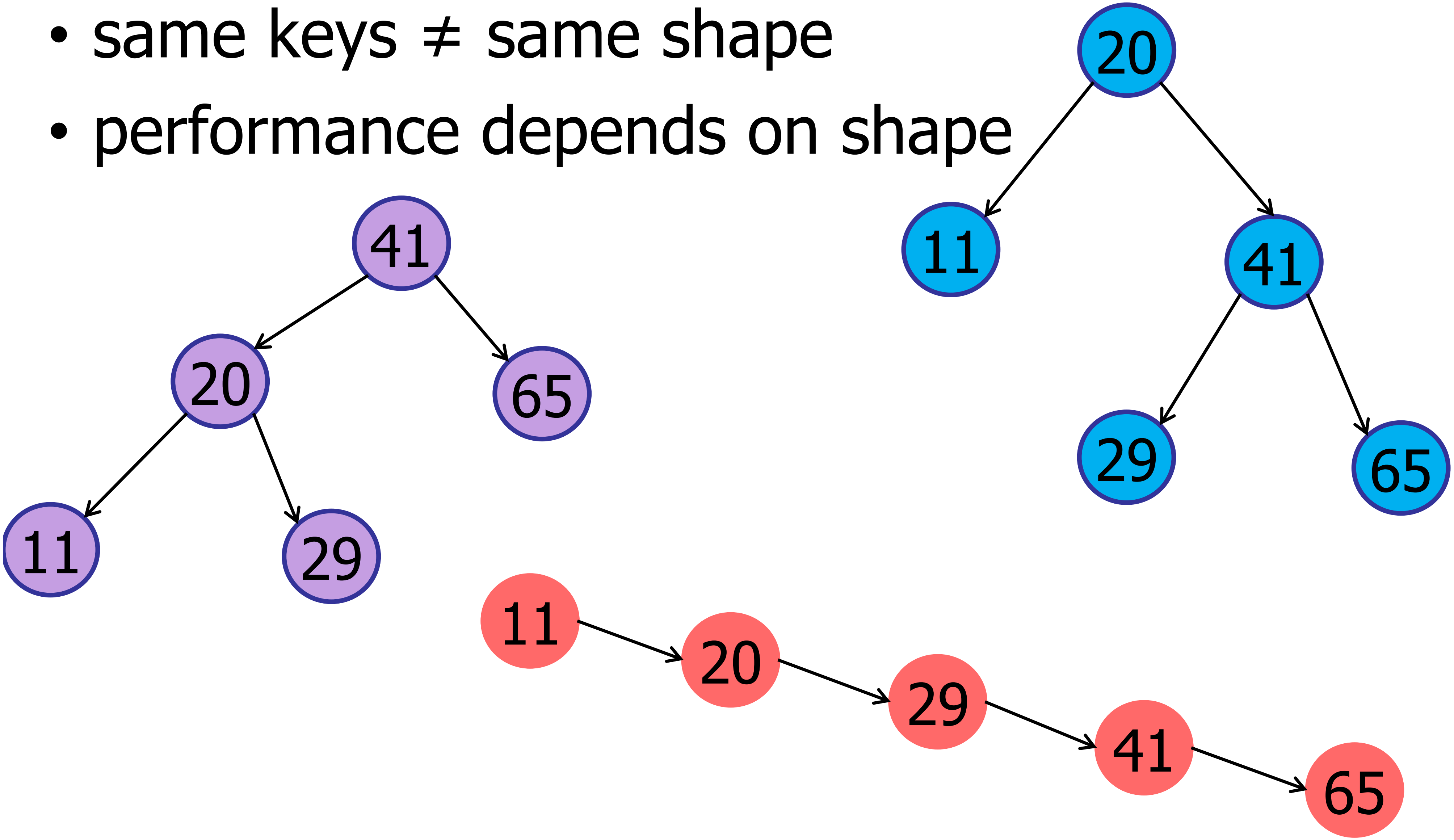
search(72) : O(height)



Tree Shape

Trees come in many shapes

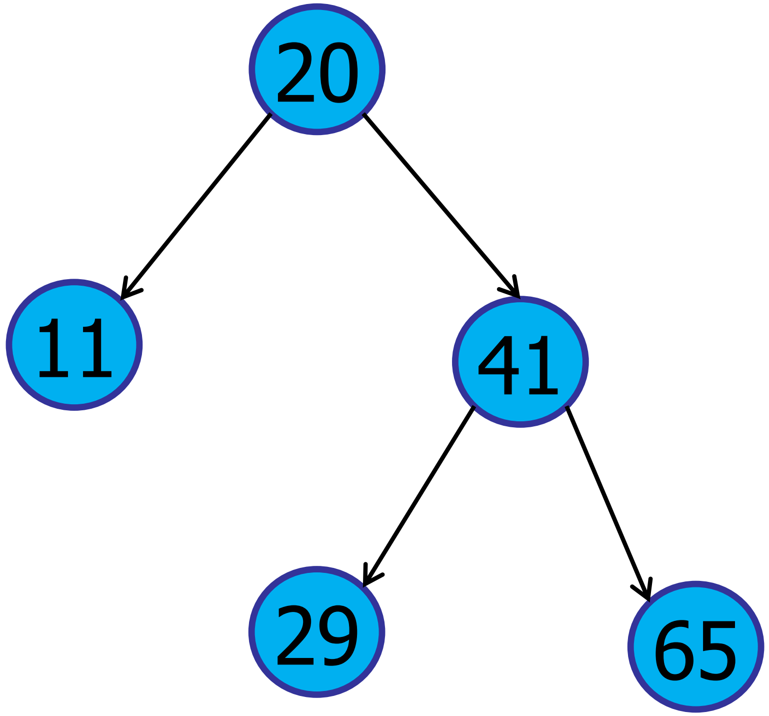
- same keys ≠ same shape
- performance depends on shape

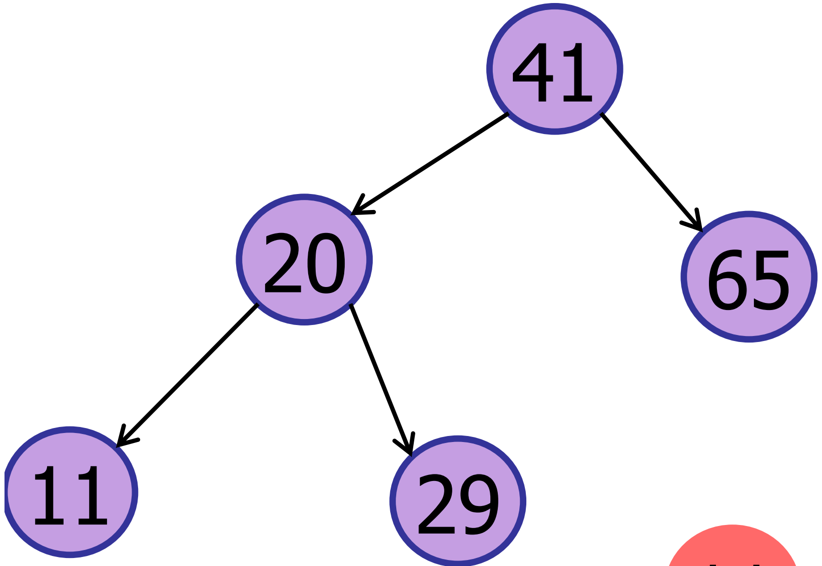


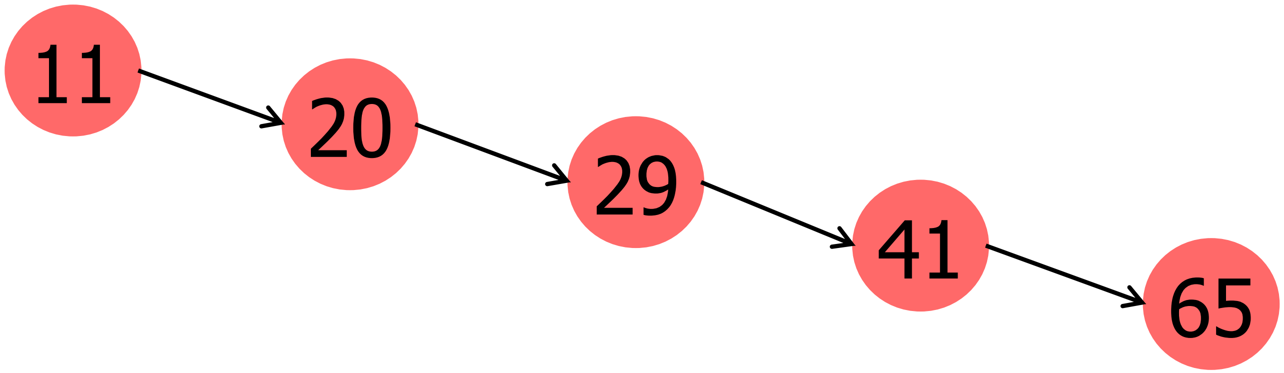
Tree Shape

What determines shape?

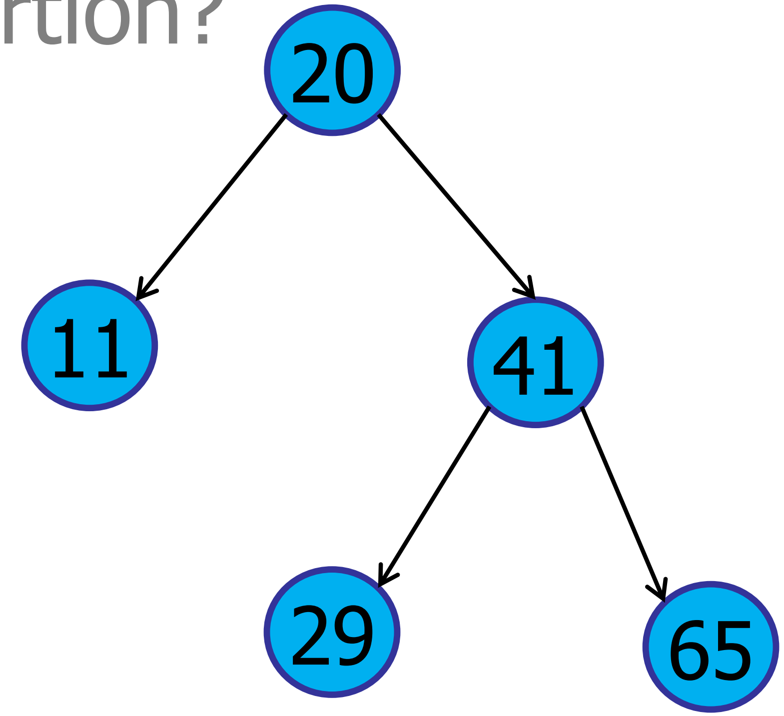
- Order of insertion







What was the order of insertion?

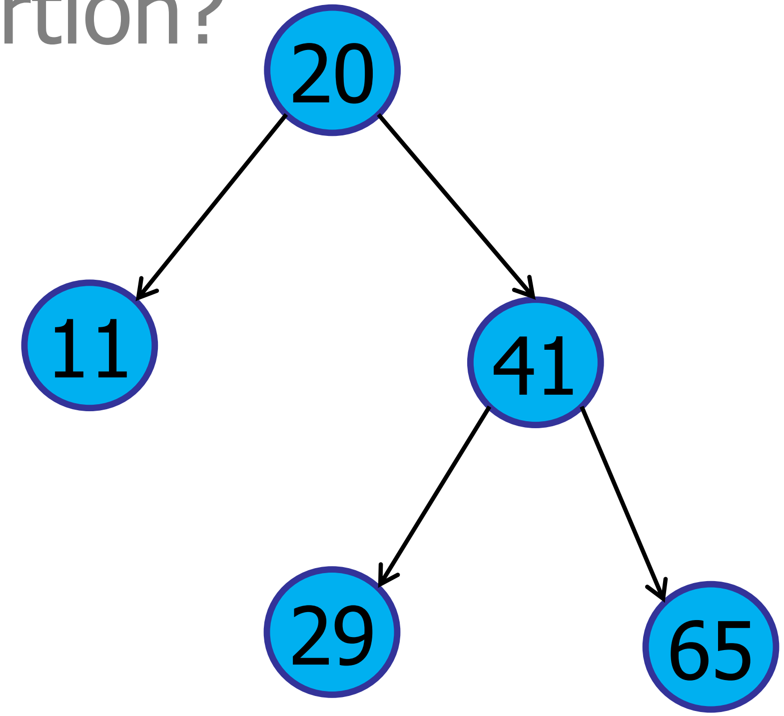


1. 11, 20, 29, 41, 65
2. 20, 11, 41, 29, 65
3. 11, 20, 41, 29, 65
4. 65, 41, 29, 20, 11
5. Impossible to tell.

ARCHIPELAGO

is open

What was the order of insertion?



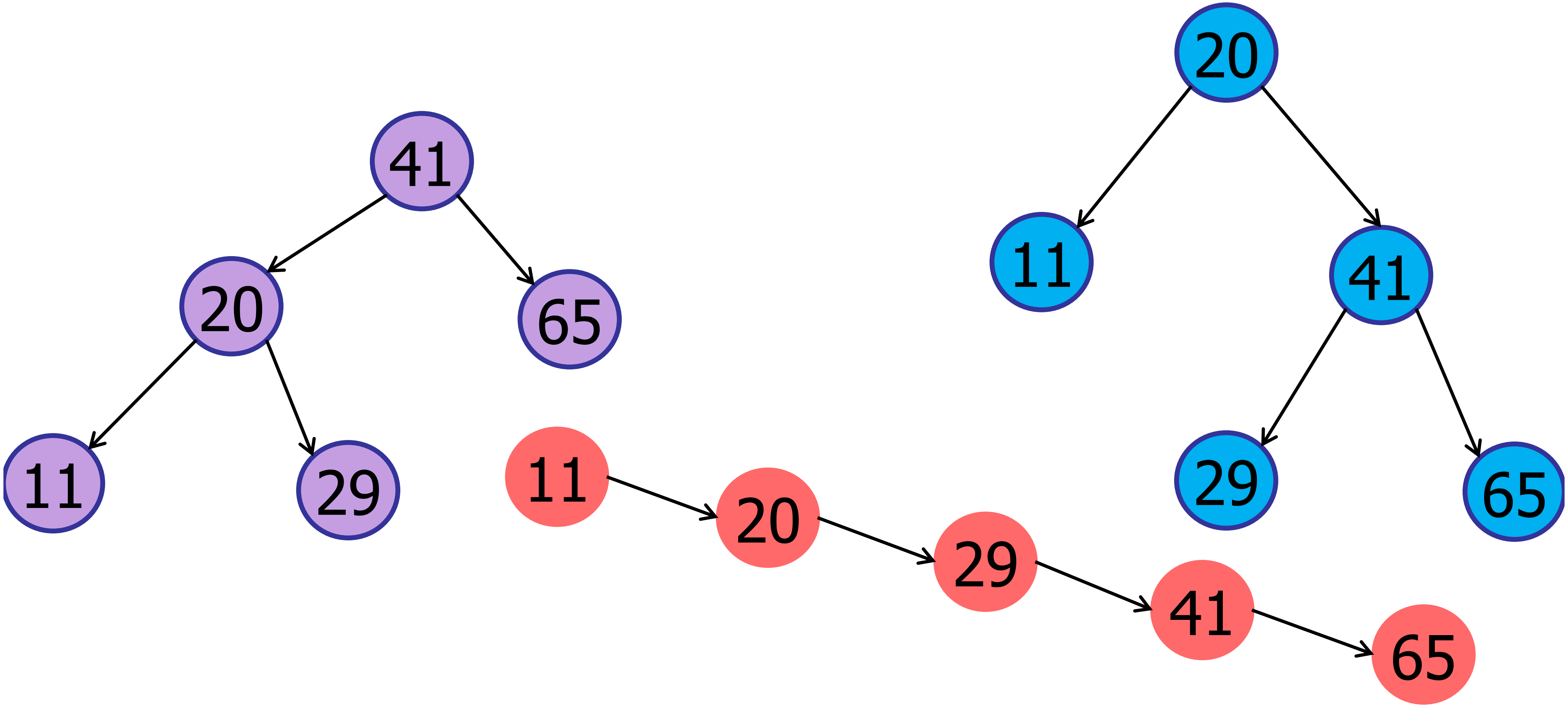
1. 11, 20, 29, 41, 65
2. 20, 11, 41, 29, 65
3. 11, 20, 41, 29, 65
4. 65, 41, 29, 20, 11
5. Impossible to tell.

Tree Shape

ARCHIPELAGO
is open

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

Catalan Numbers

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

C_n = # expressions with n pairs of matched parentheses

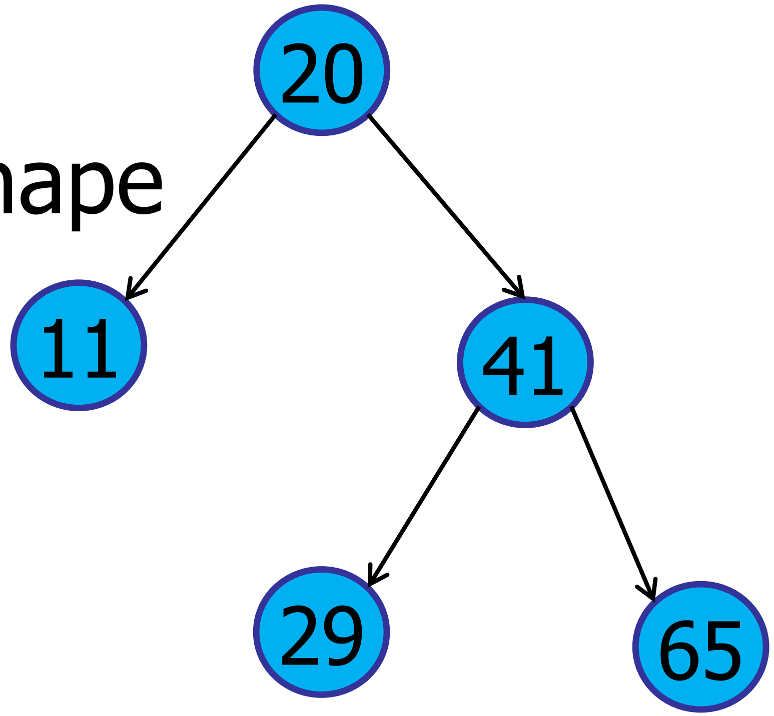
((()))   ()(())   (()())   (())()   ()()()

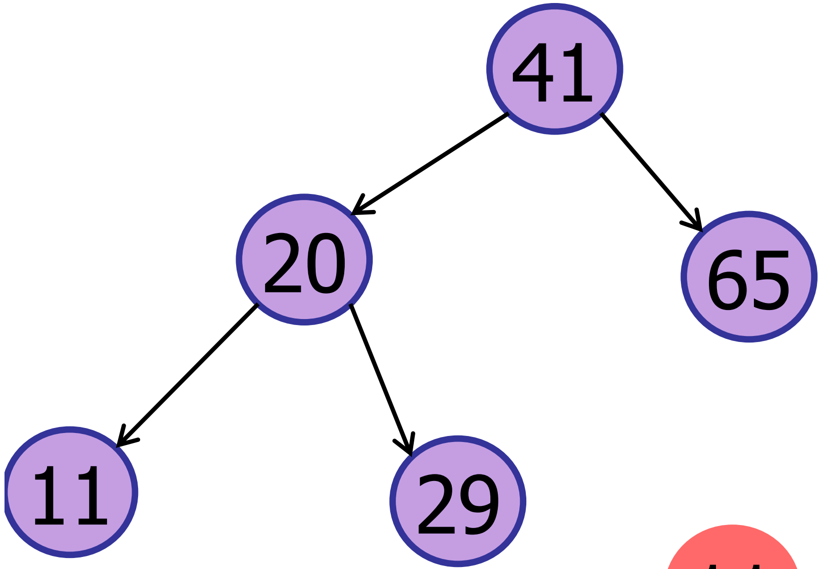
Puzzle: why are these the same?

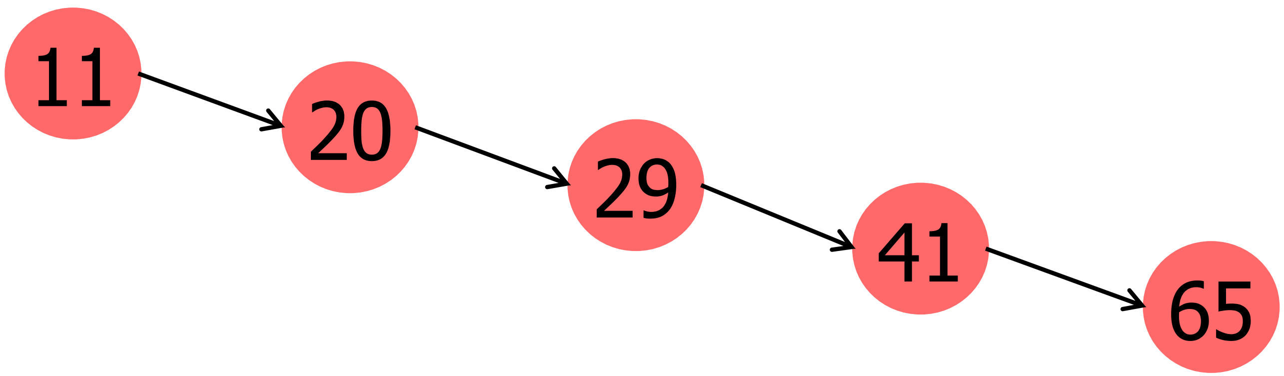
Tree Shape

Trees come in many shapes

- same keys ≠ same shape
- performance depends on shape



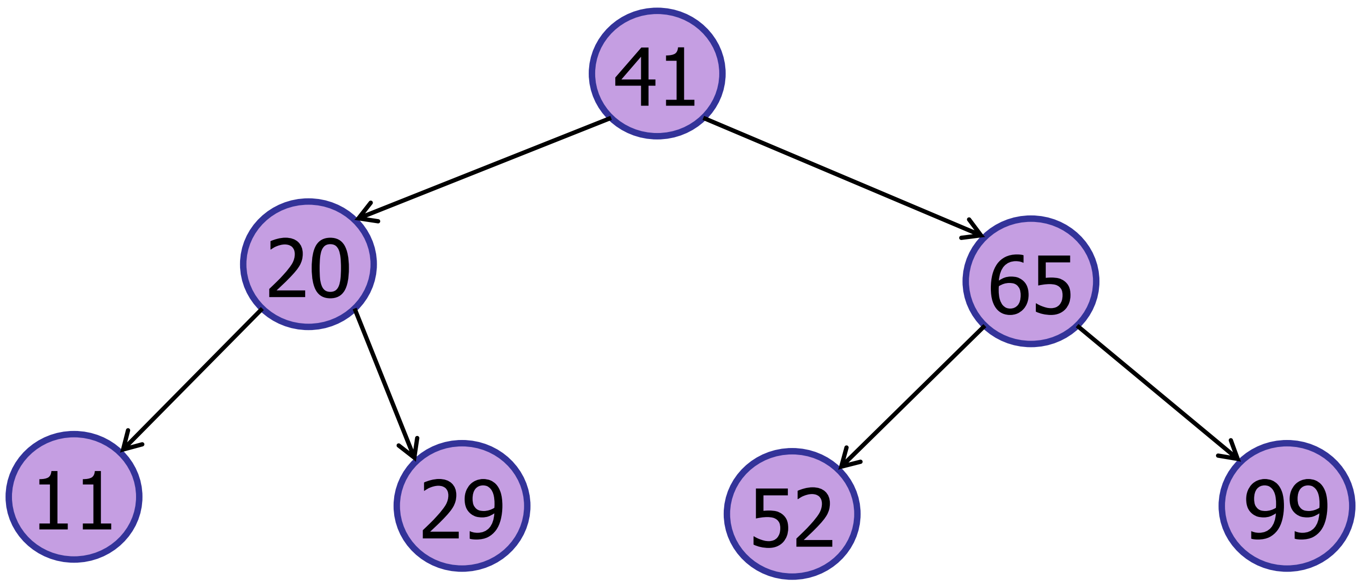




Tree Shape

Trees come in many shapes

- same keys ≠ same shape
- performance depends on shape
- insert keys in a *random* order ⇒ balanced



Binary Search Trees

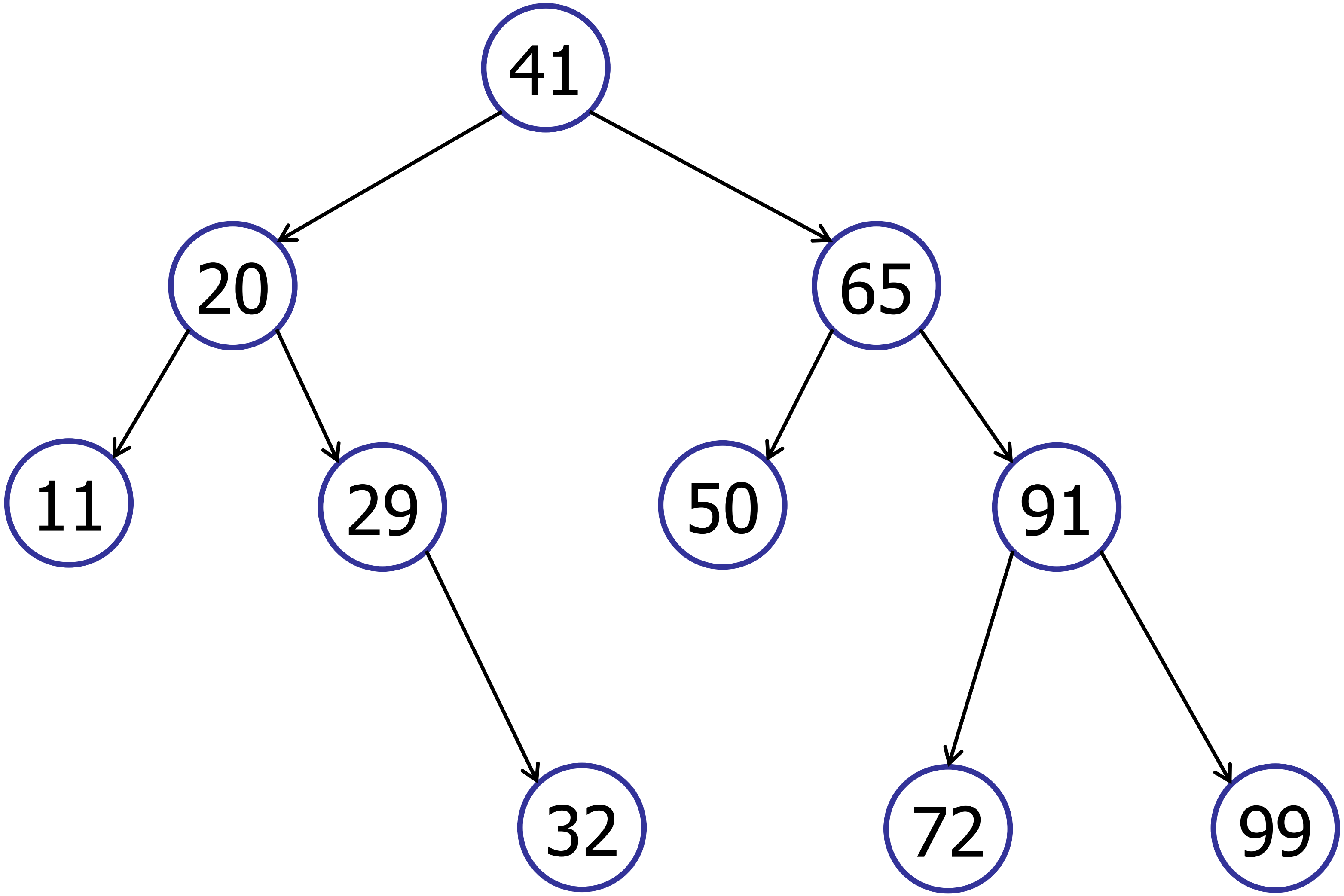
1. Terminology and Definitions

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

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

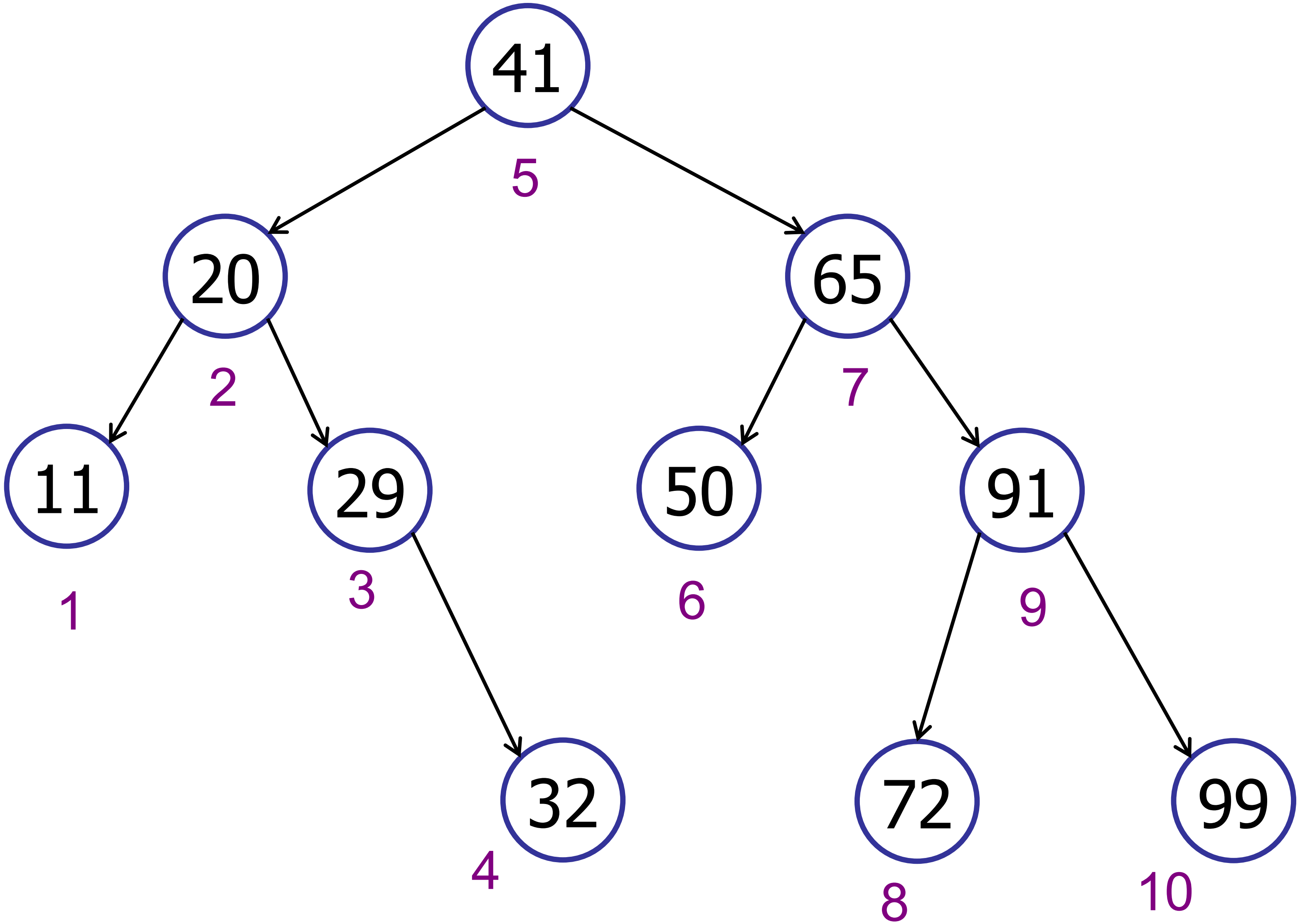
4. Other operations

Tree Traversal



11 20 29 32 41 50 65 72 91 99

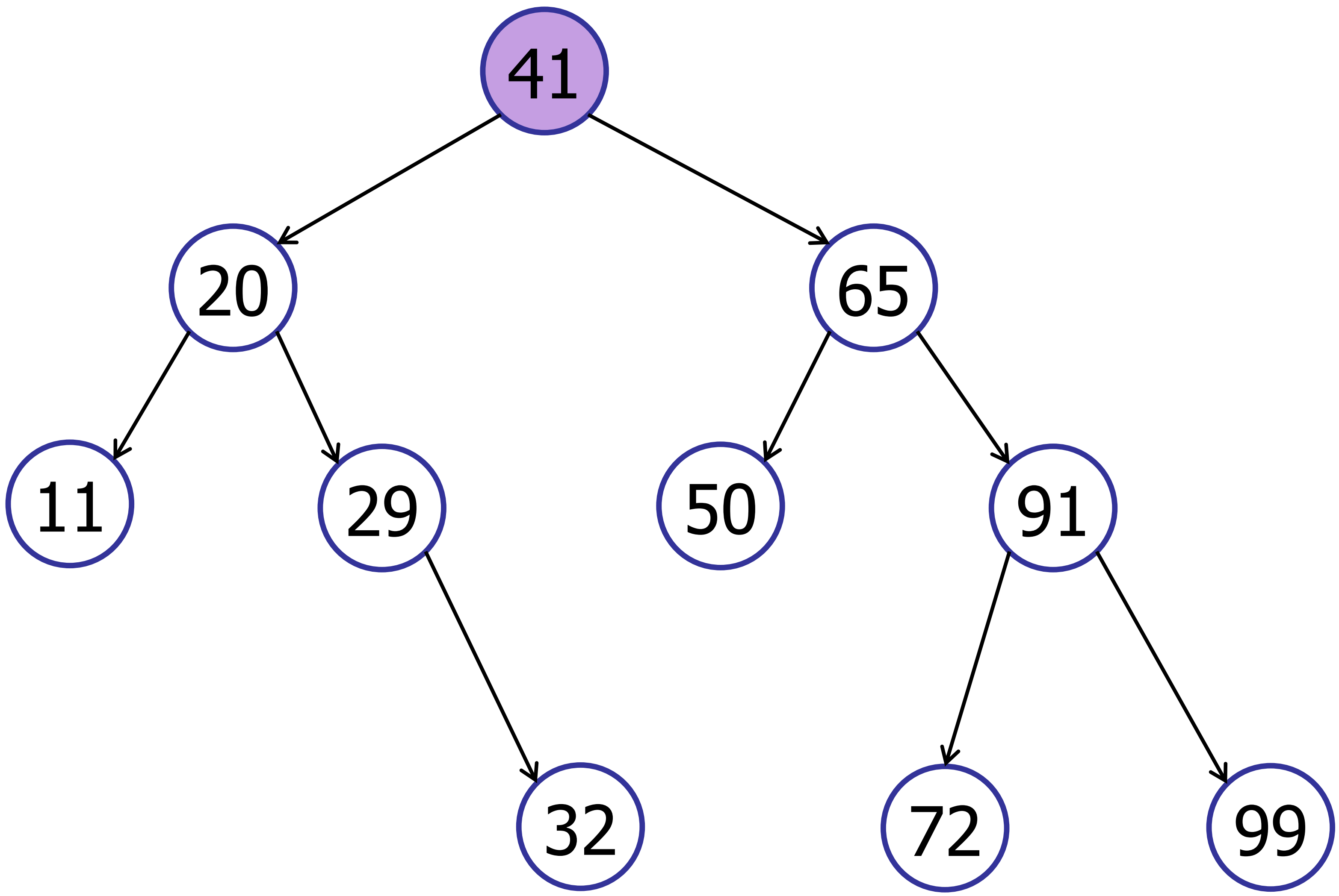
Tree Traversal



11 20 29 32 41 50 65 72 91 99

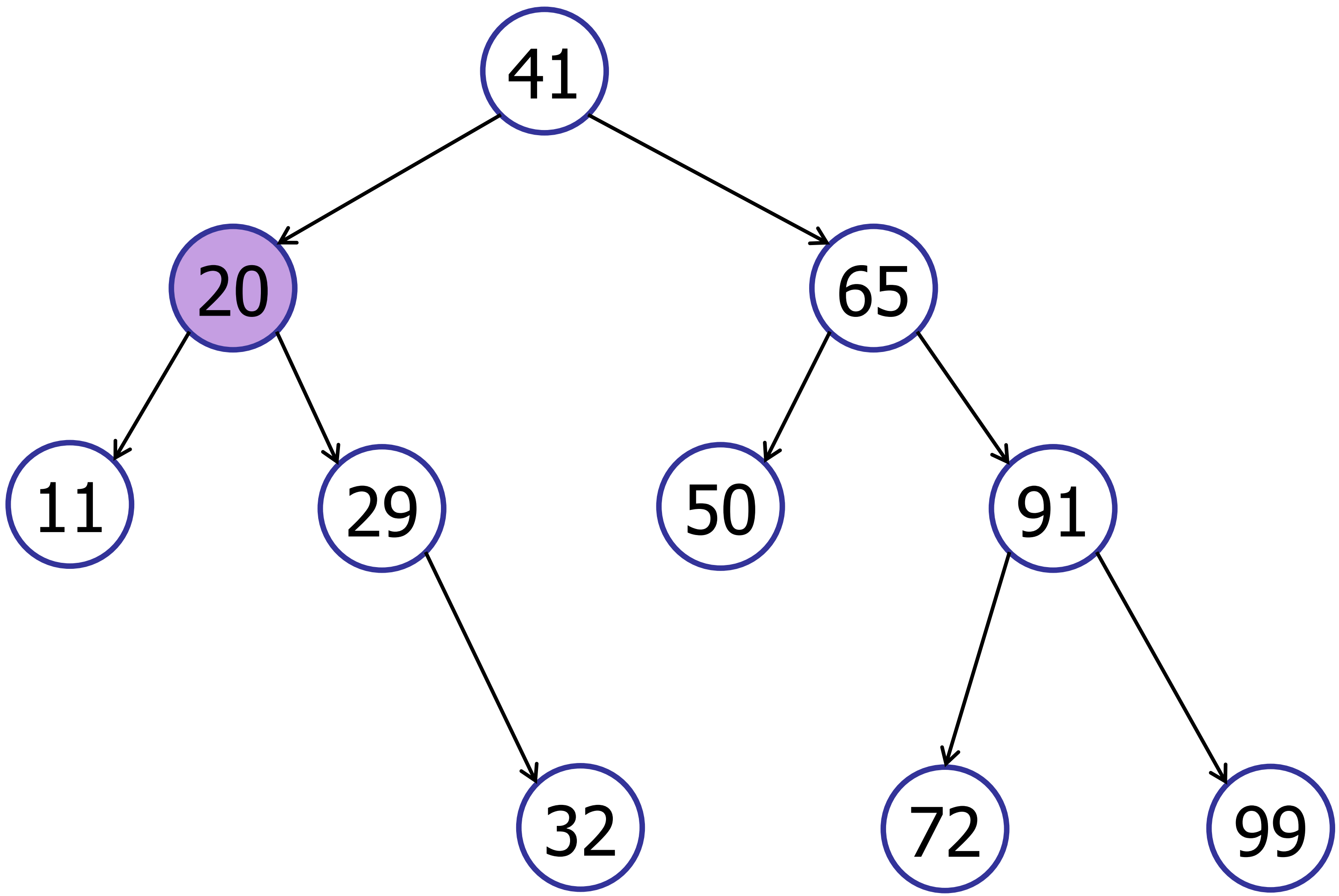
Tree Traversal

in-order-traversal



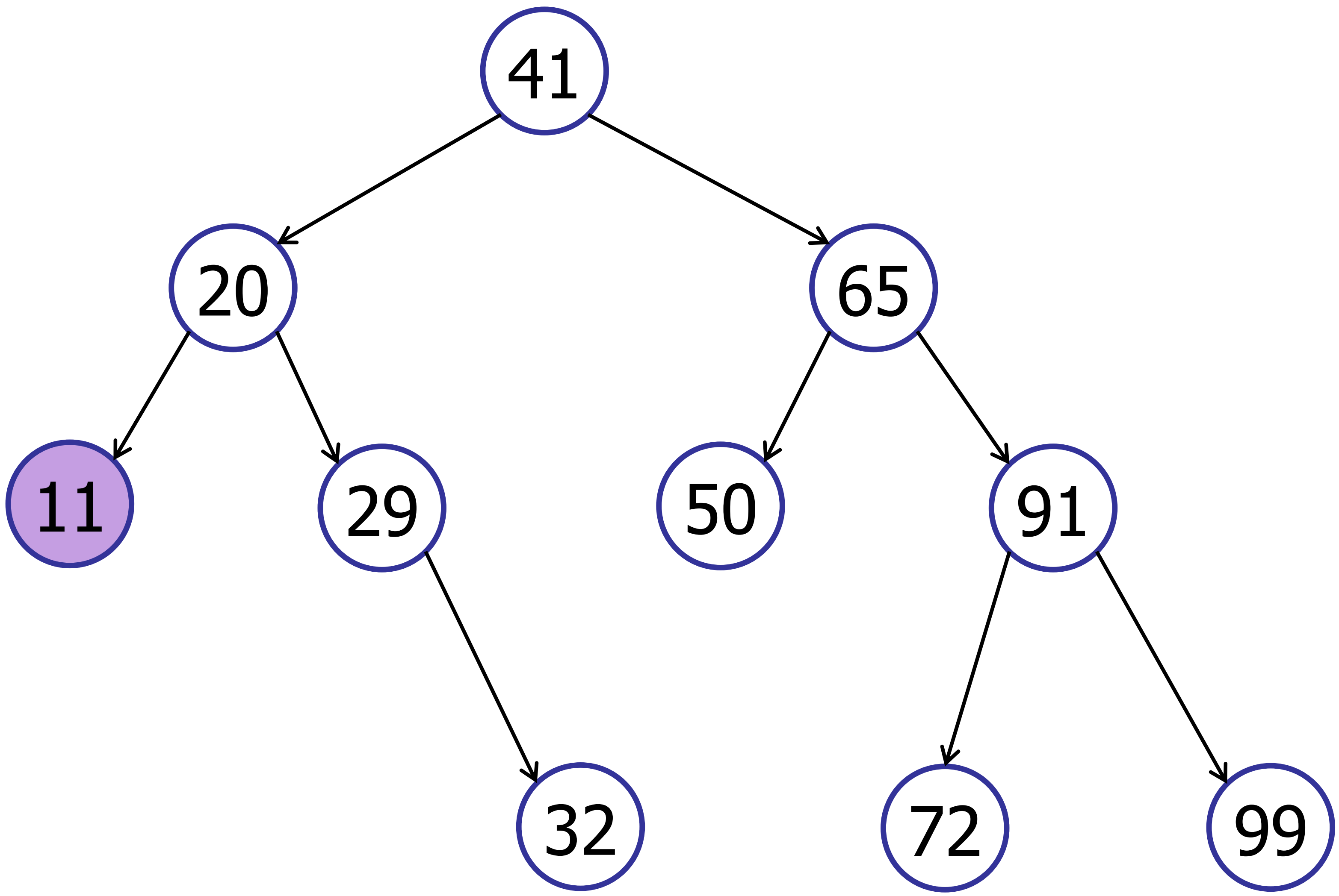
Tree Traversal

in-order-traversal



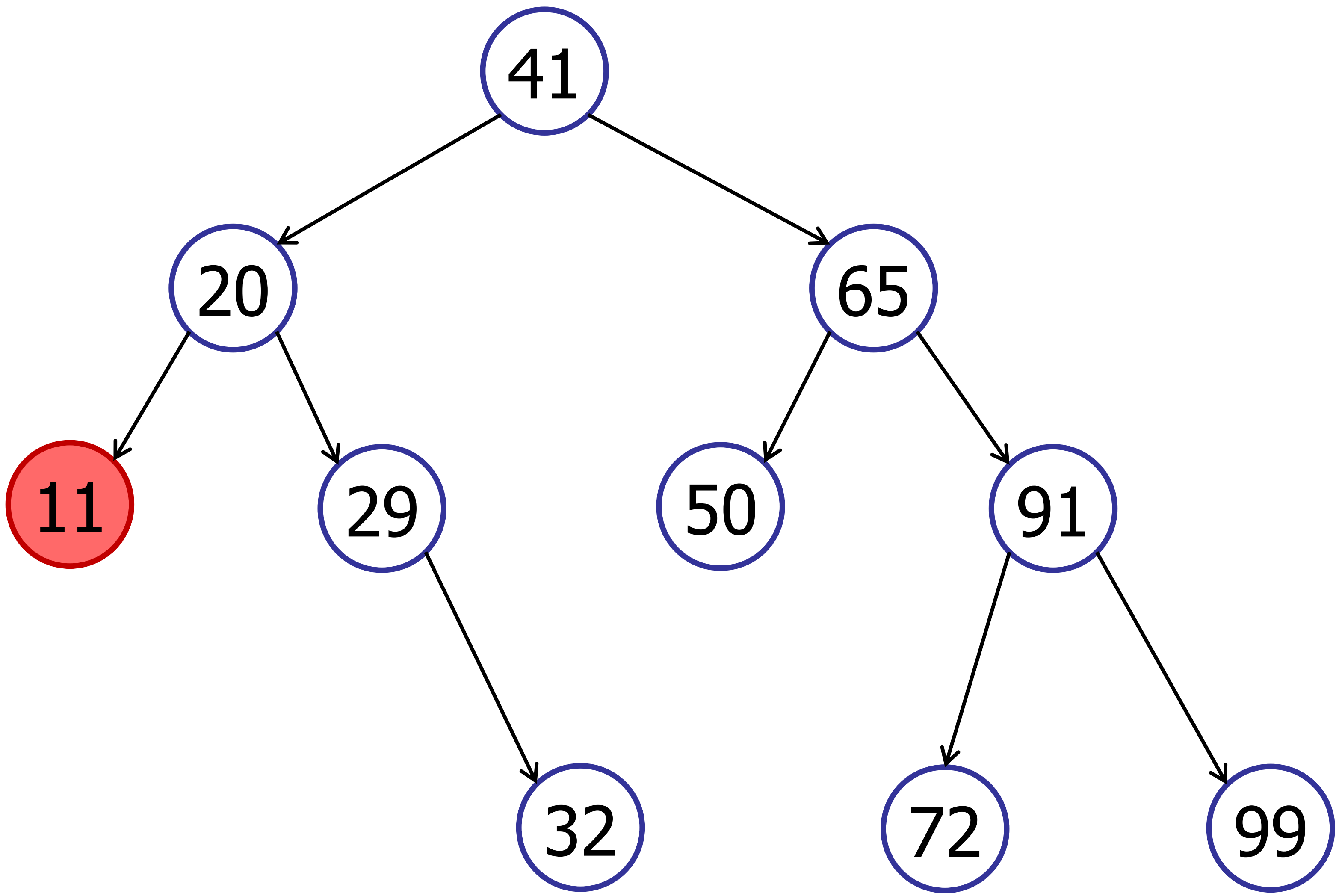
Tree Traversal

in-order-traversal



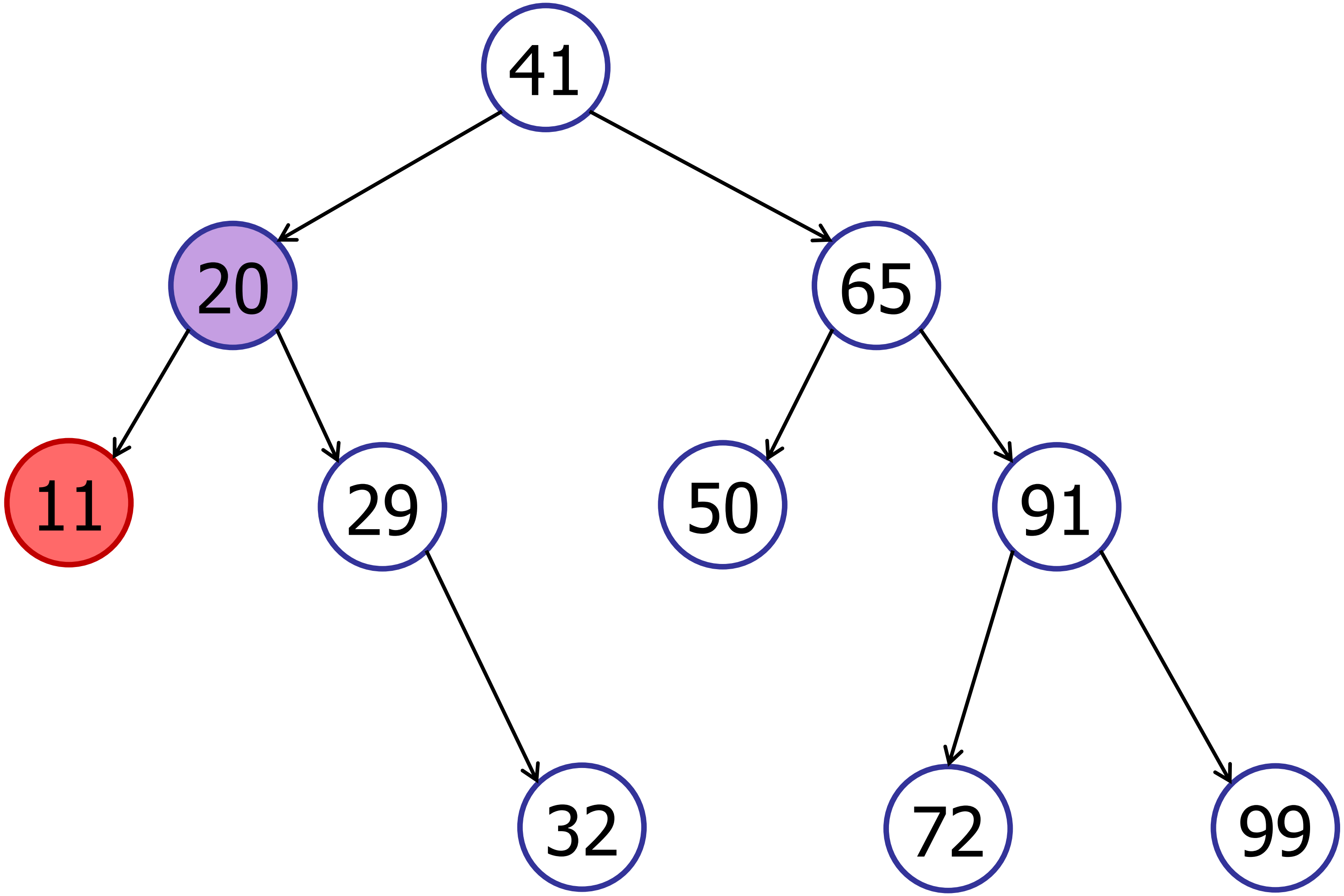
Tree Traversal

in-order-traversal



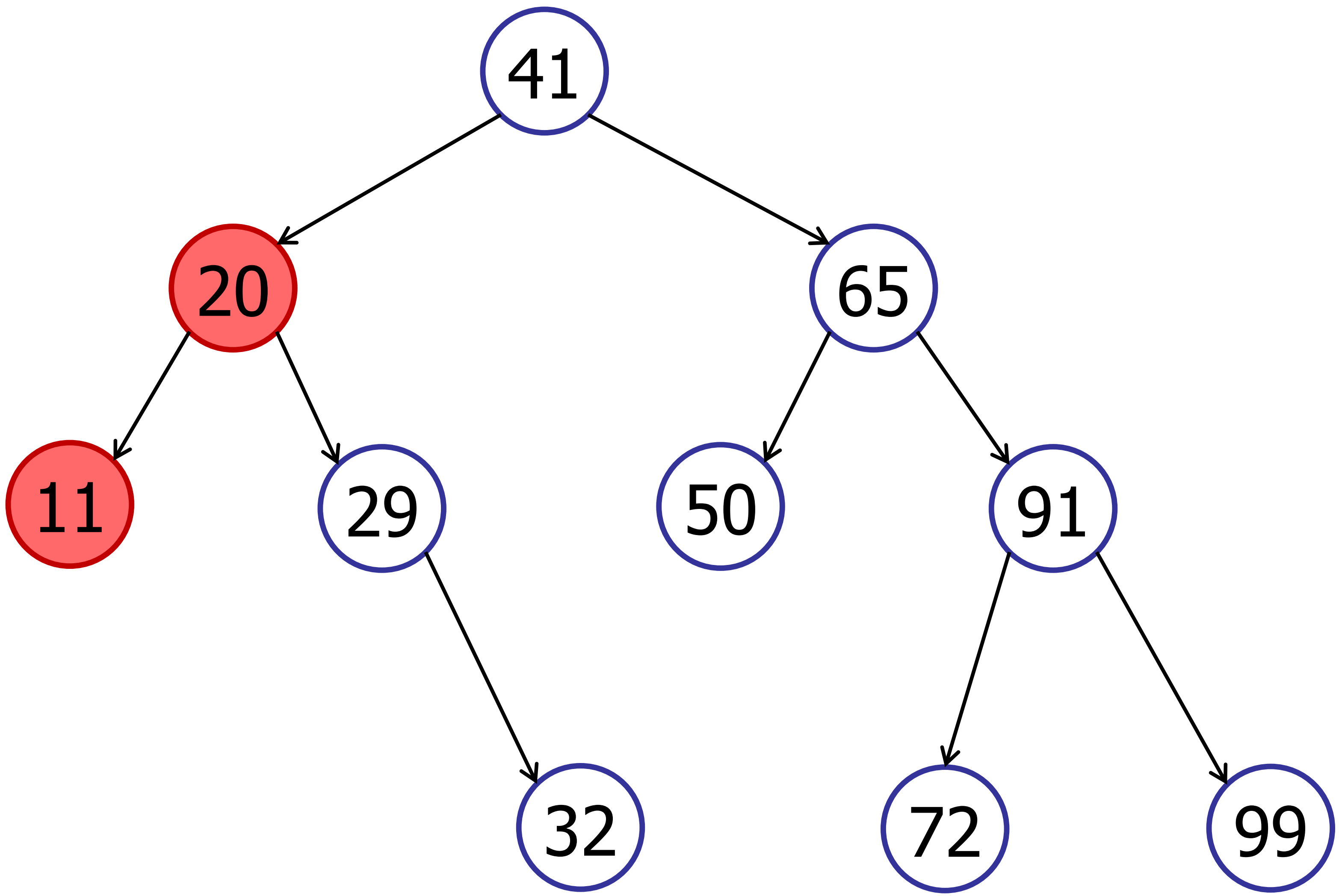
Tree Traversal

in-order-traversal



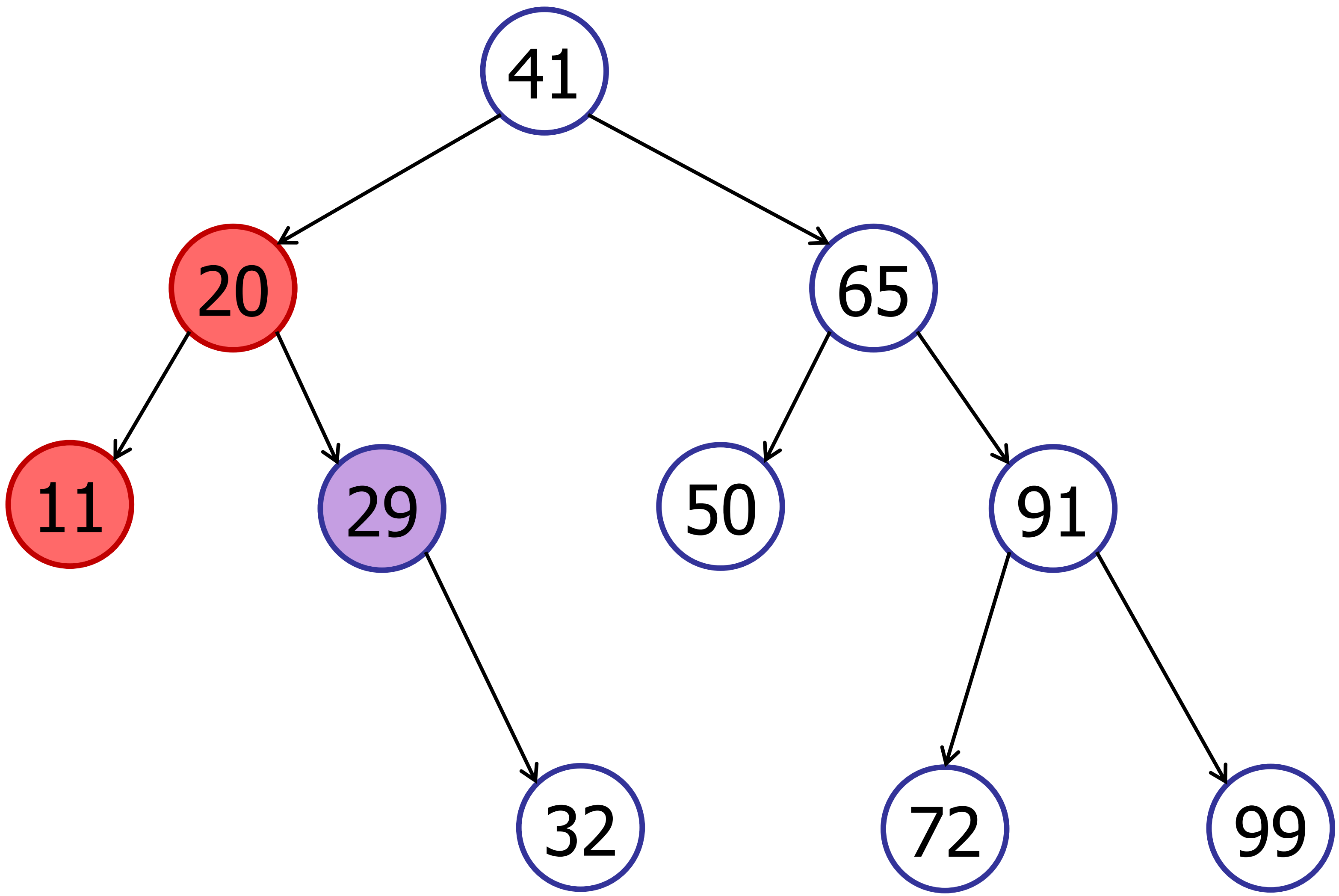
Tree Traversal

in-order-traversal



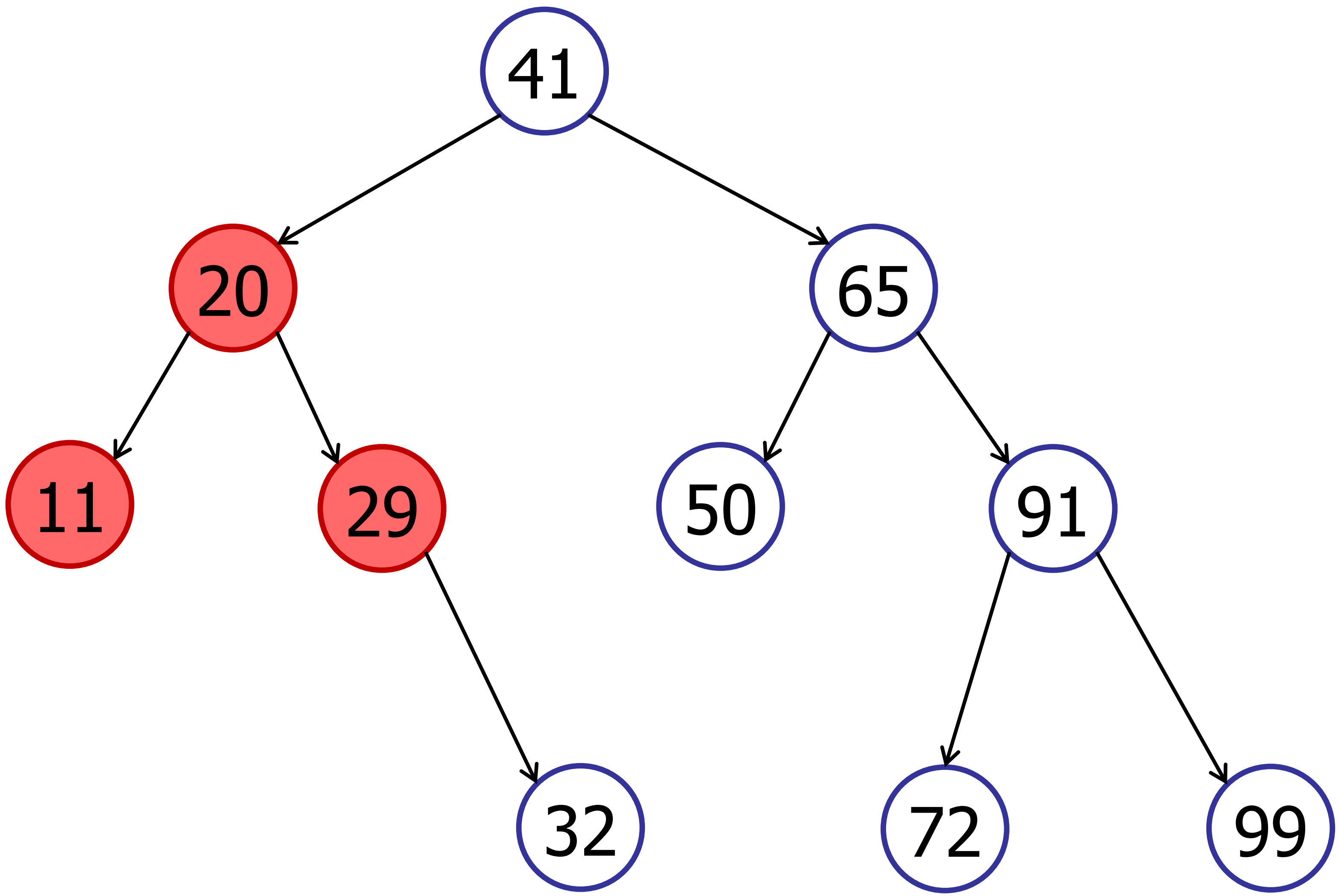
Tree Traversal

in-order-traversal



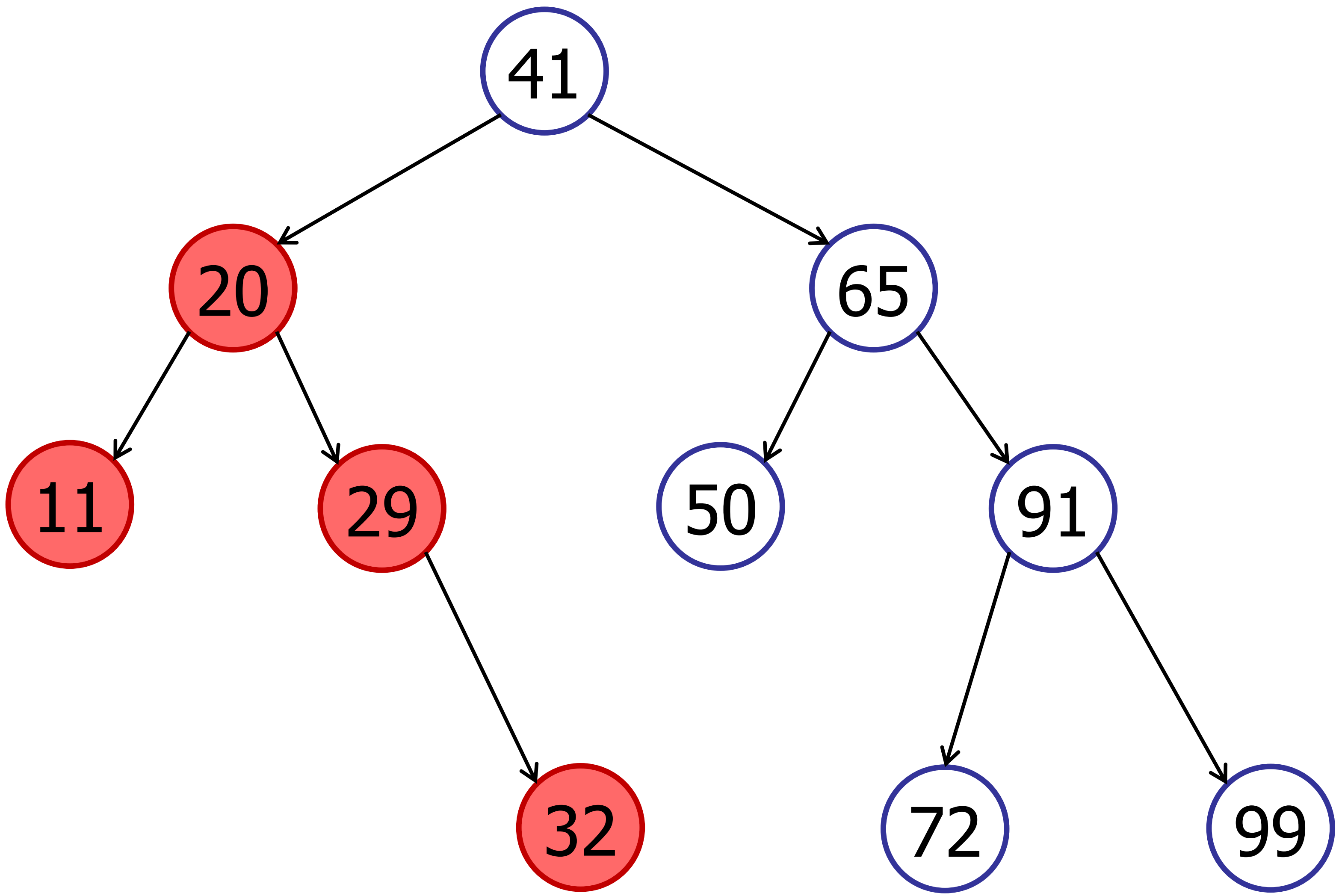
Tree Traversal

in-order-traversal



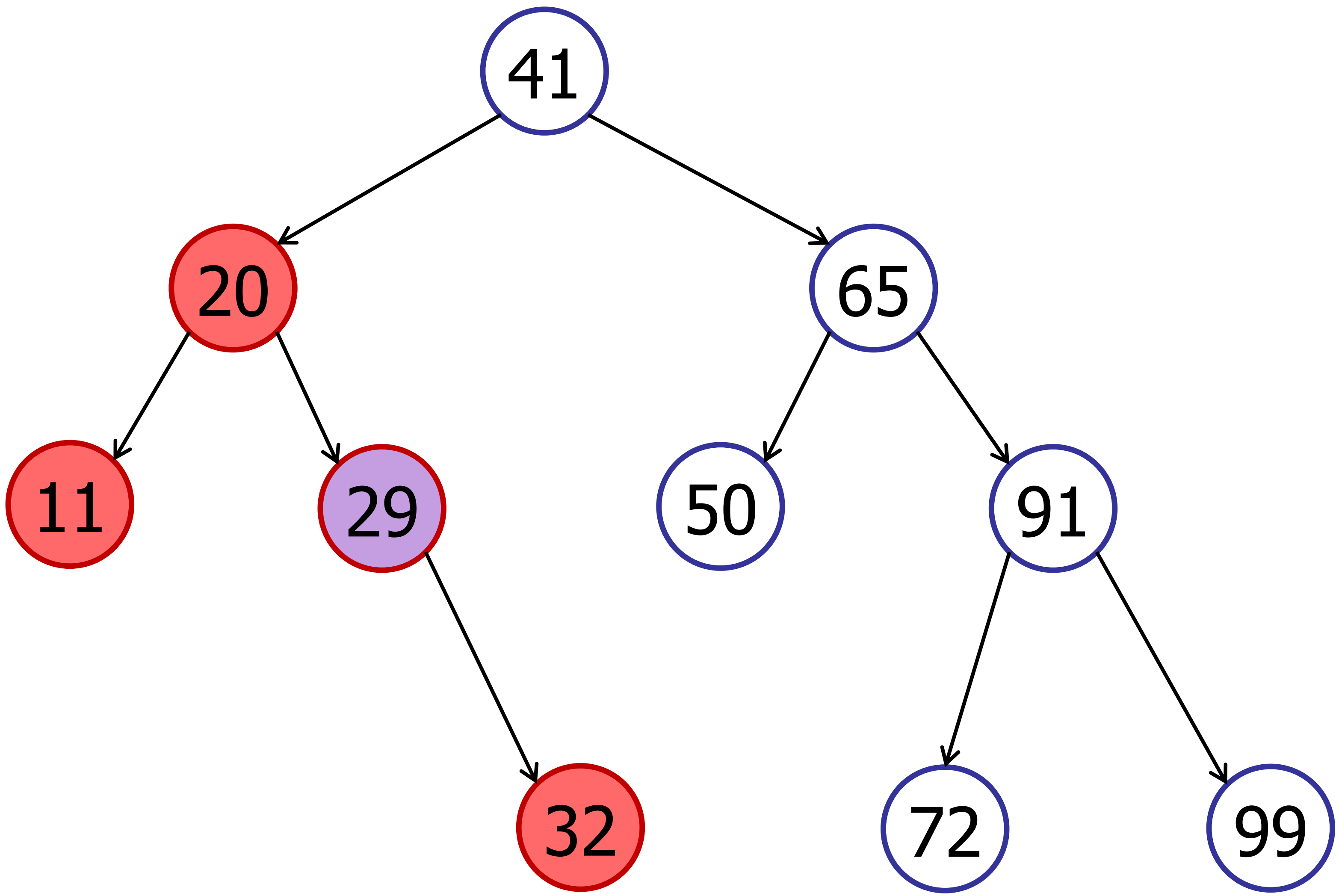
Tree Traversal

in-order-traversal



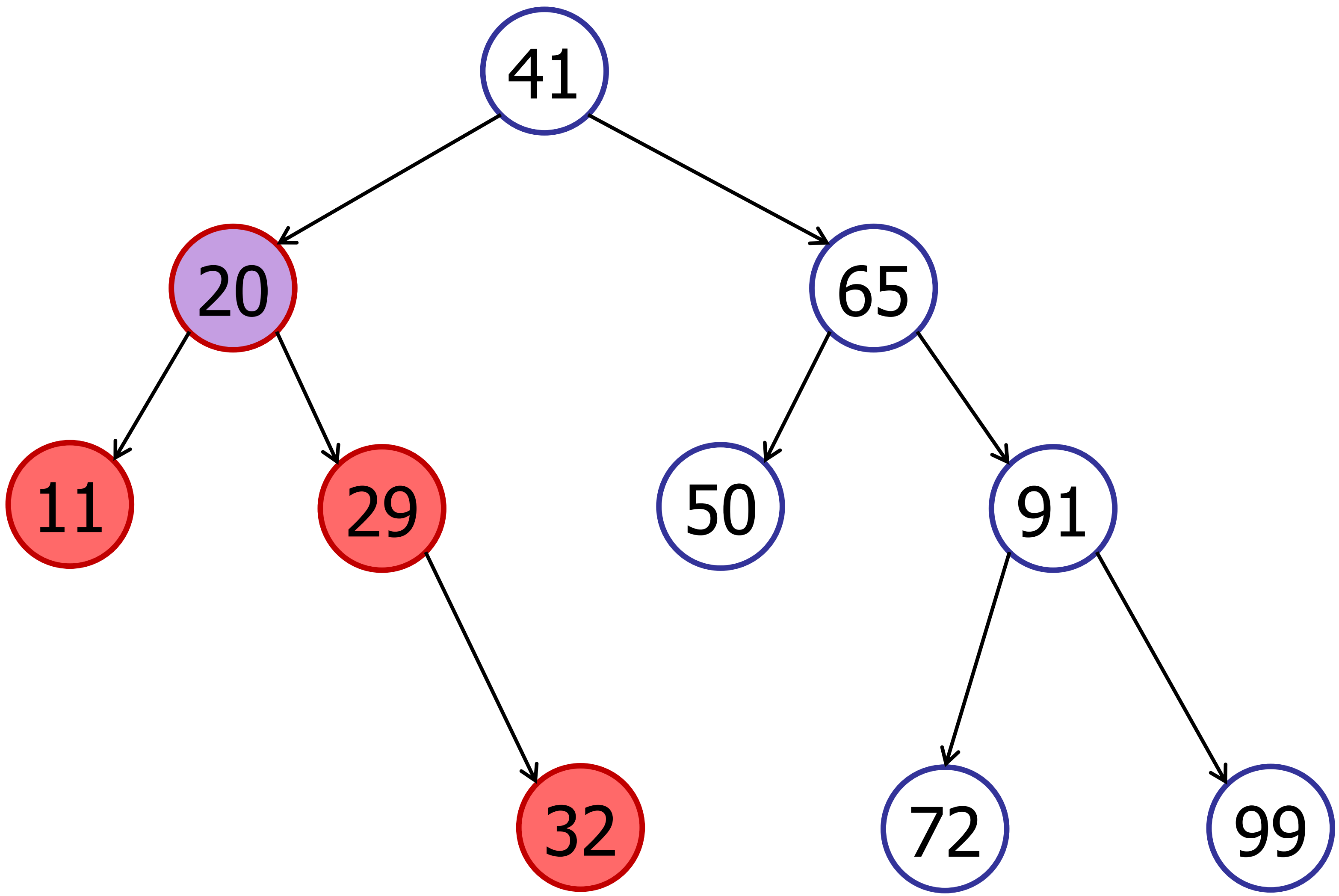
Tree Traversal

in-order-traversal



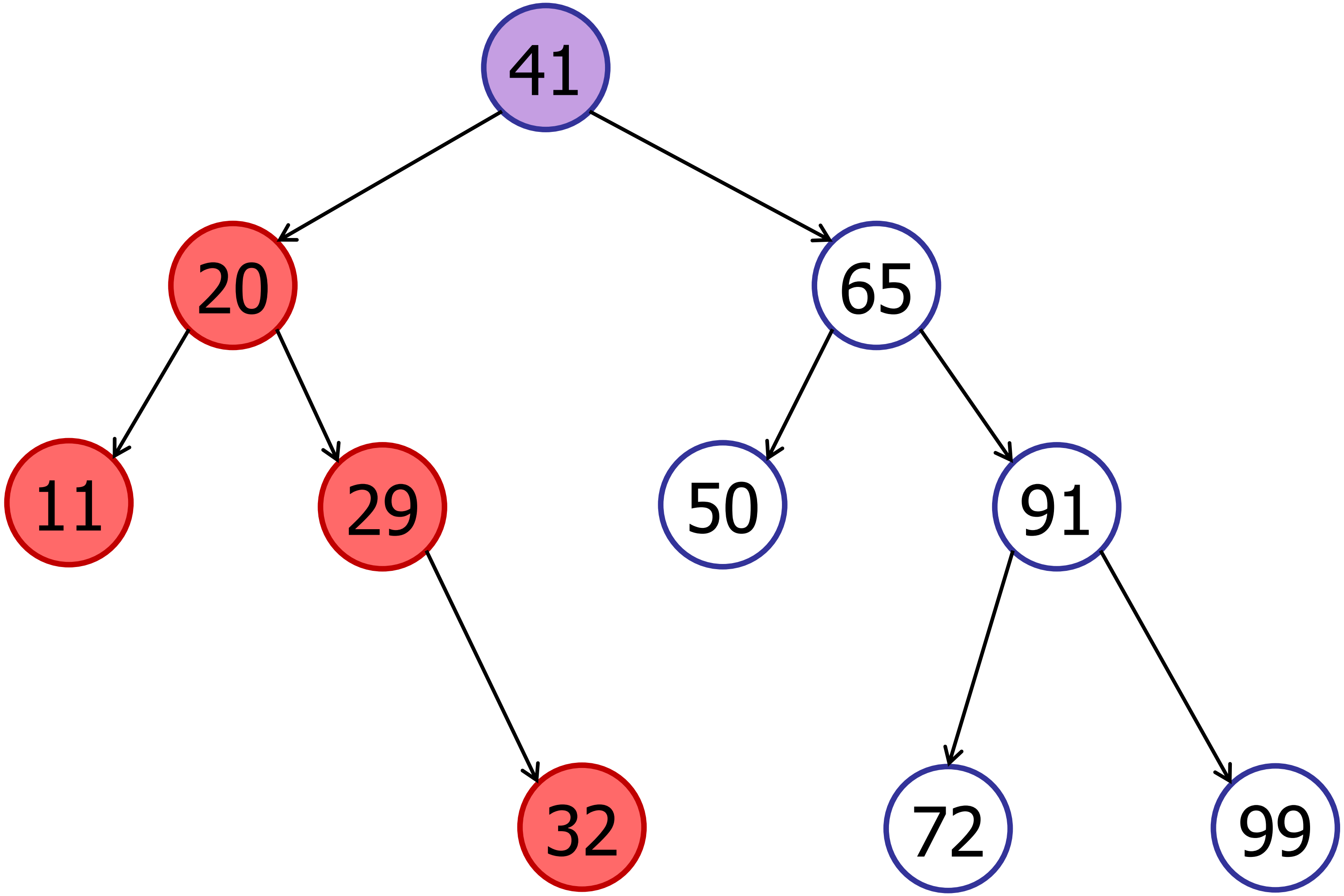
Tree Traversal

in-order-traversal



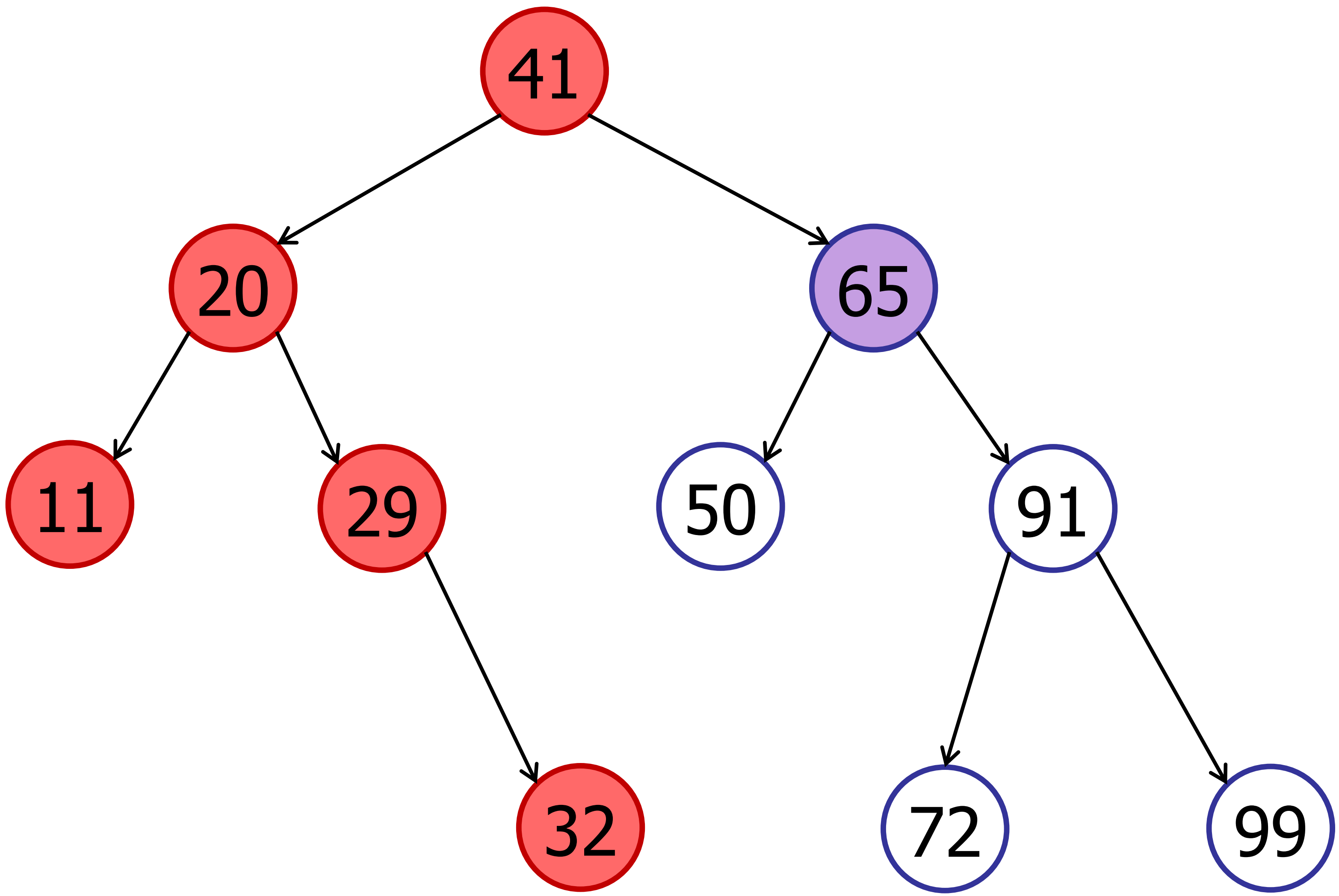
Tree Traversal

in-order-traversal



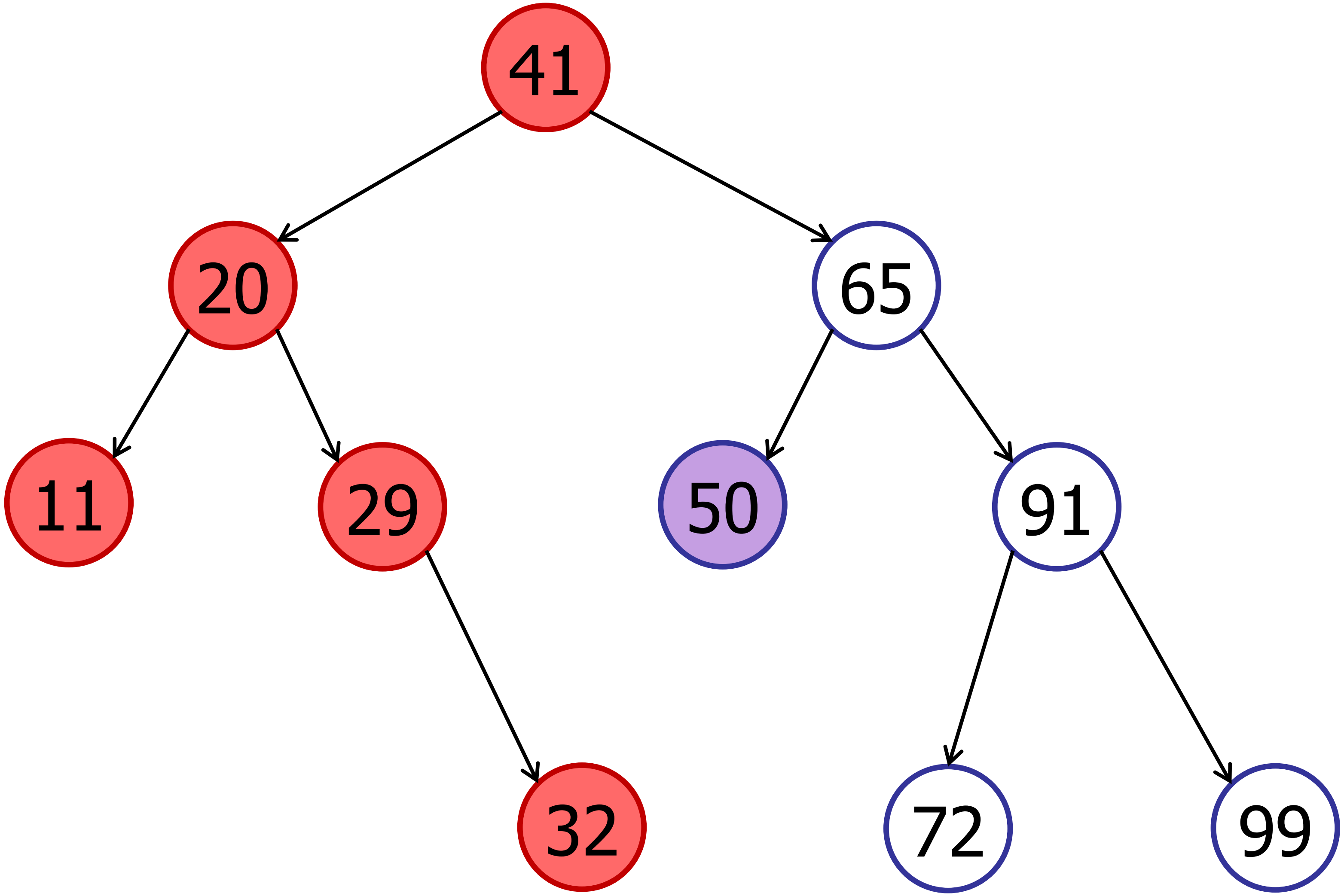
Tree Traversal

in-order-traversal



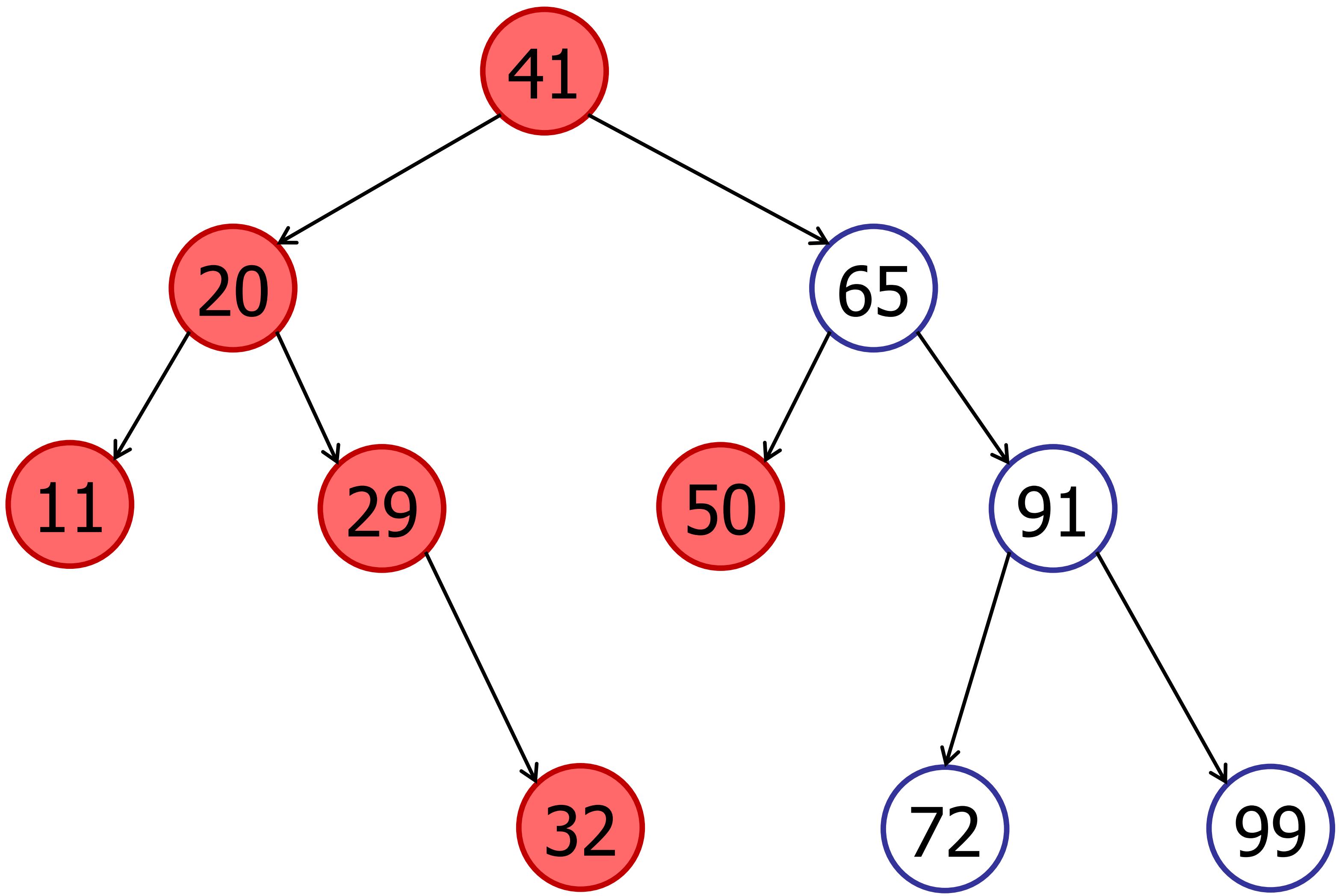
Tree Traversal

in-order-traversal



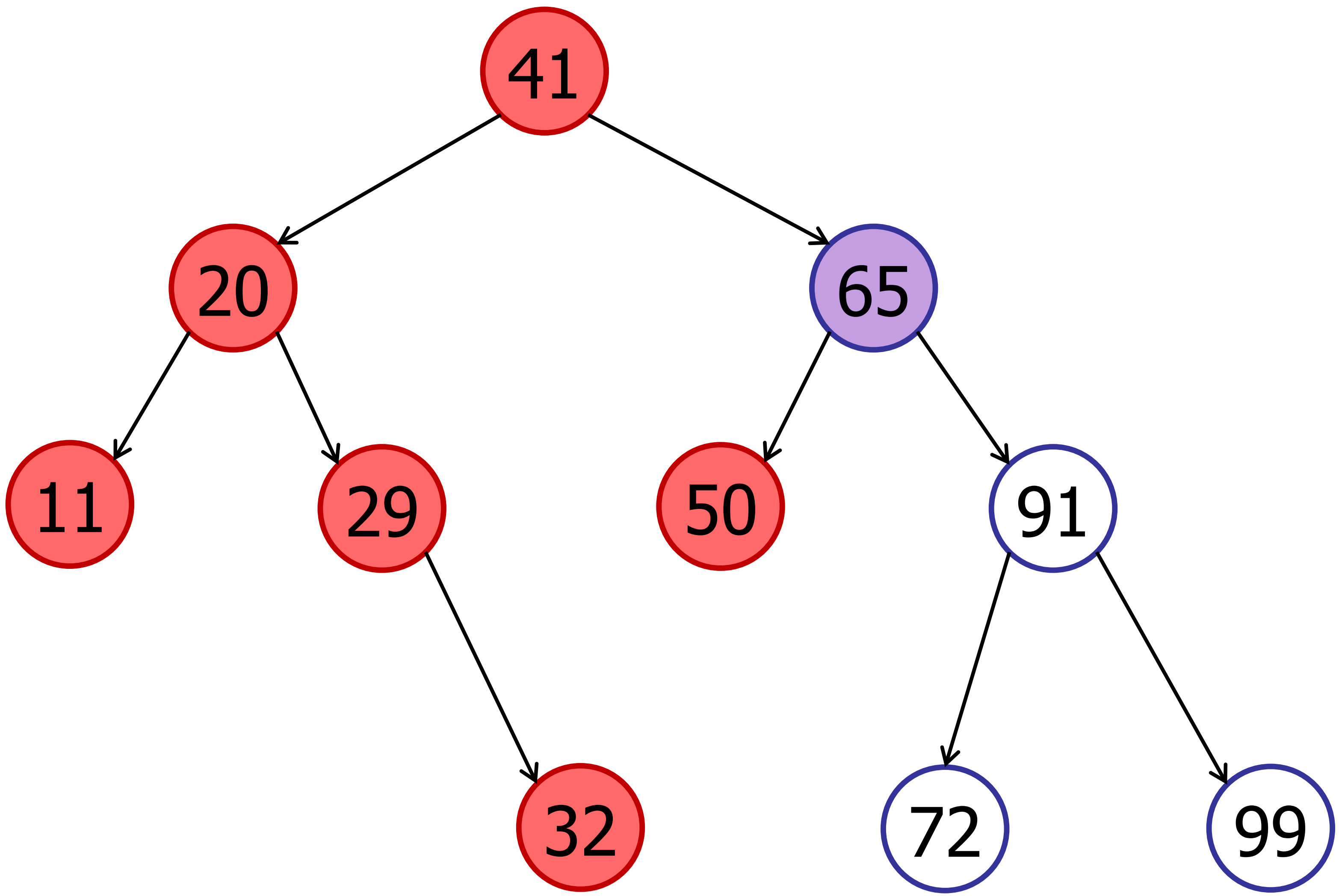
Tree Traversal

in-order-traversal



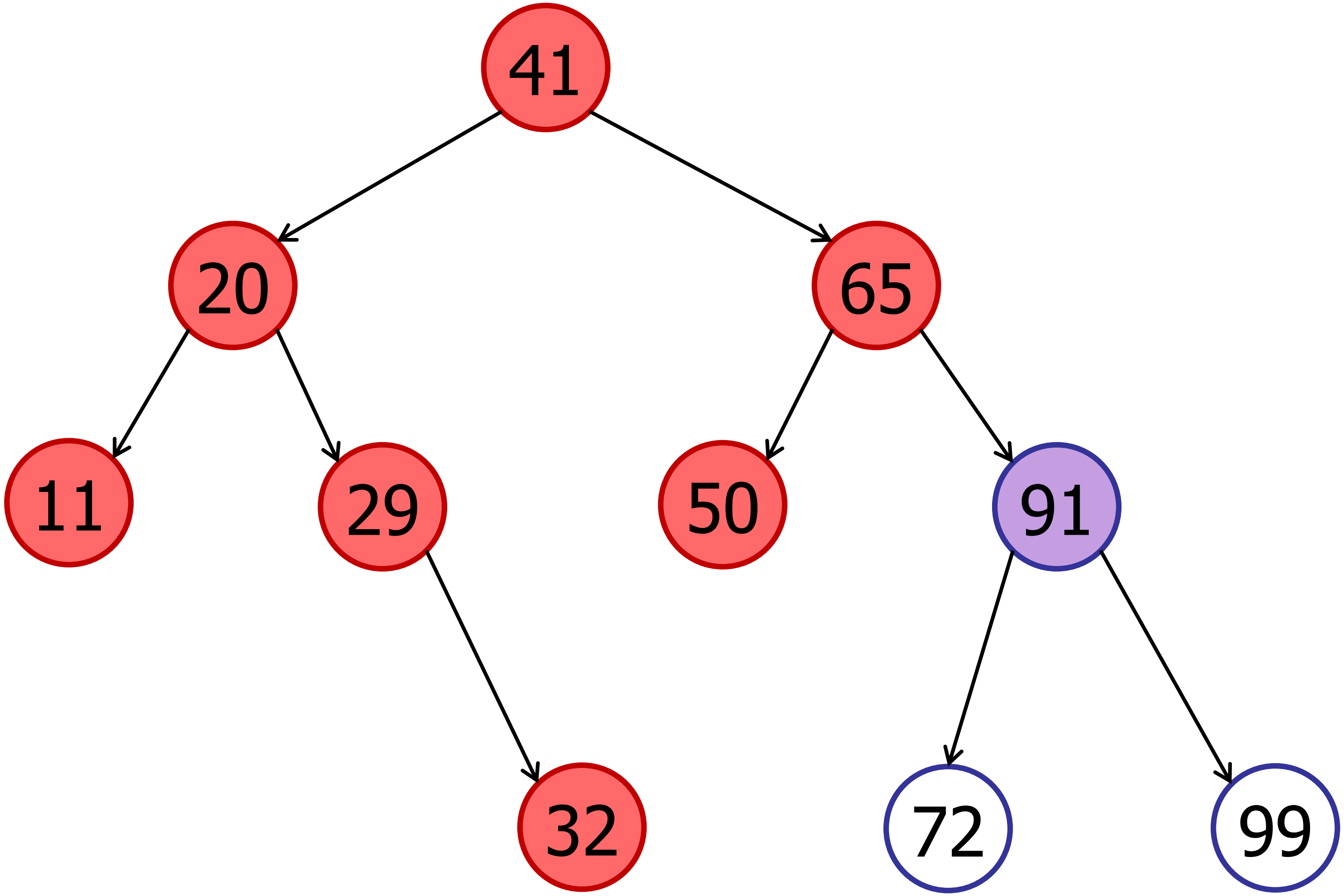
Tree Traversal

in-order-traversal



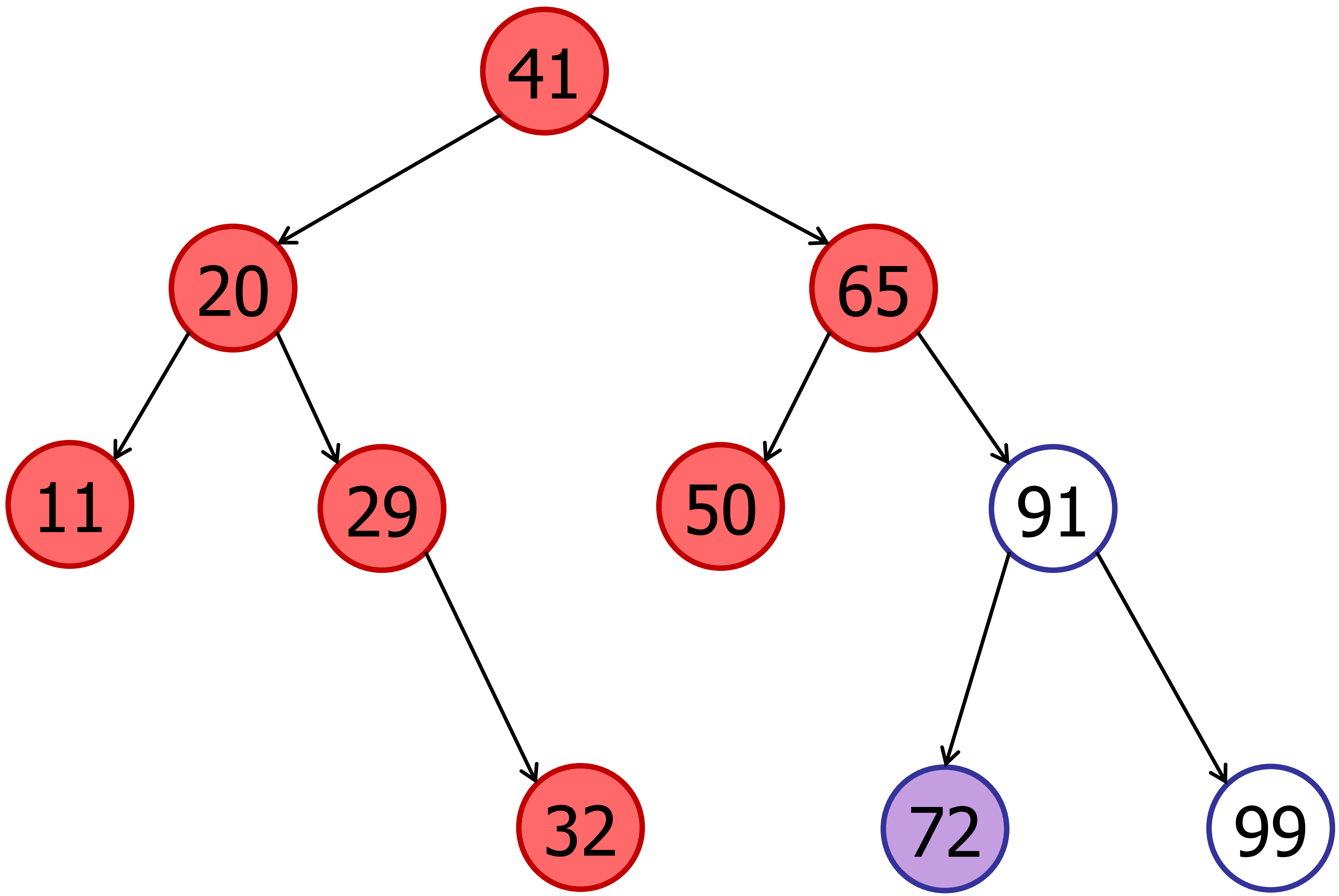
Tree Traversal

in-order-traversal



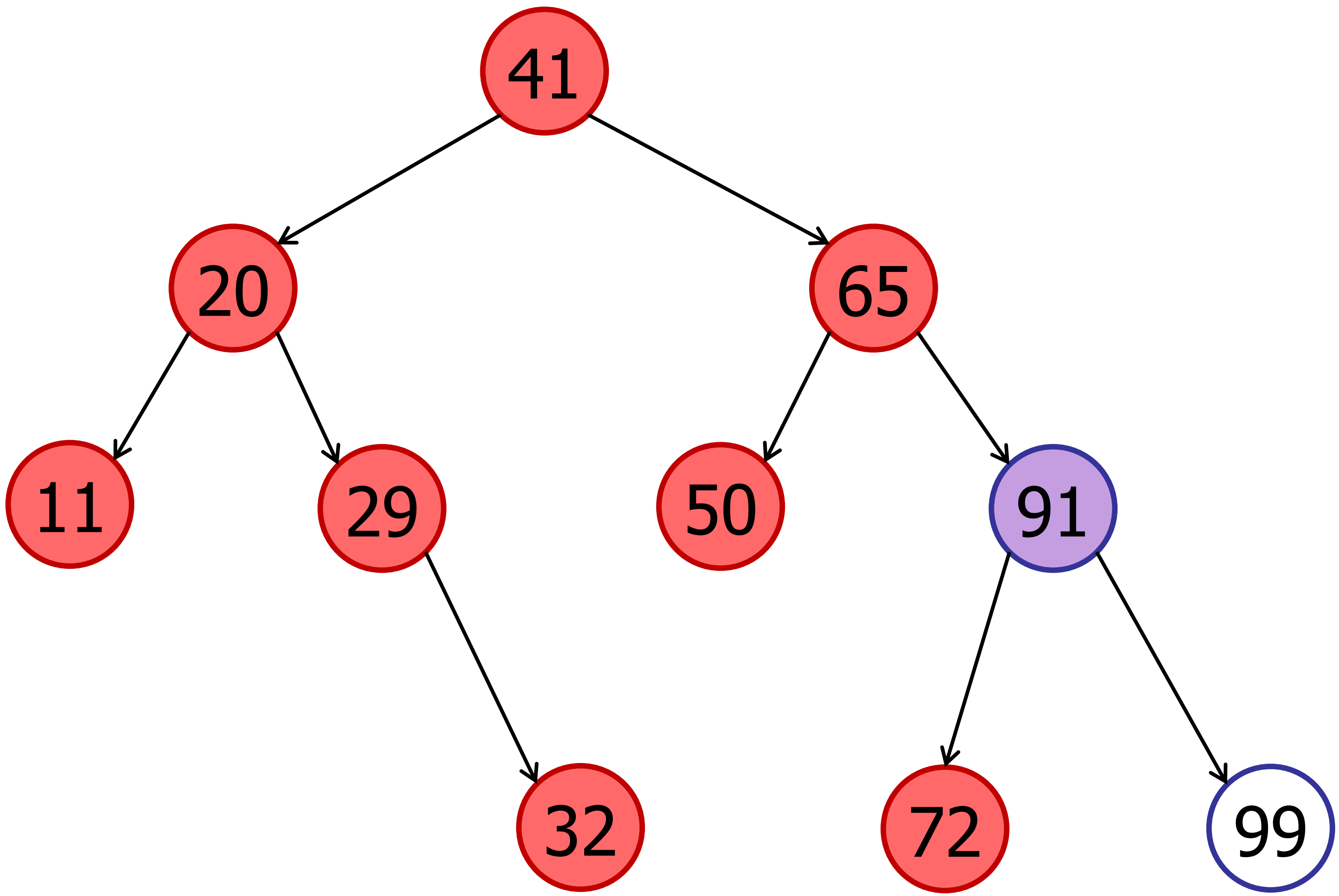
Tree Traversal

in-order-traversal



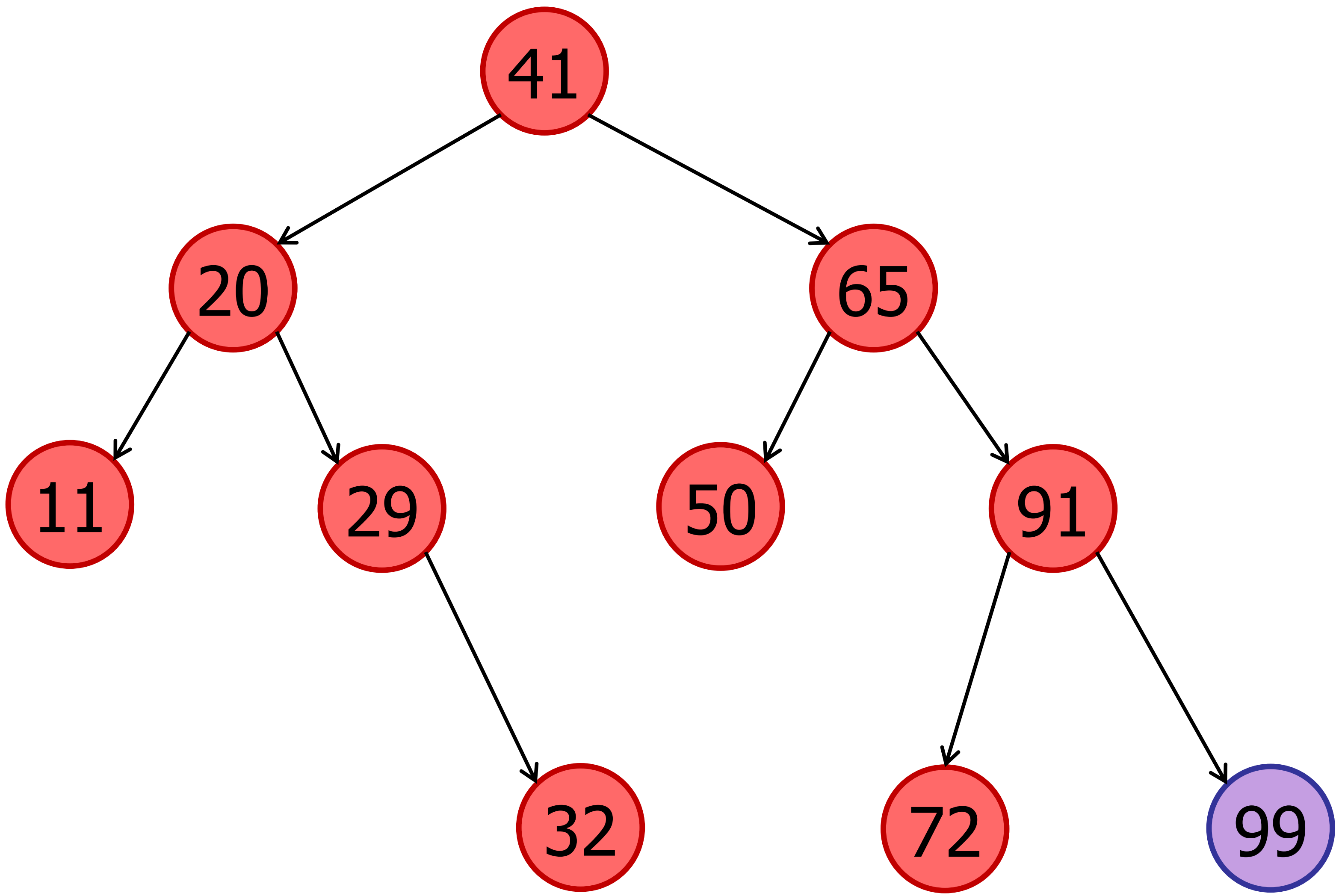
Tree Traversal

in-order-traversal



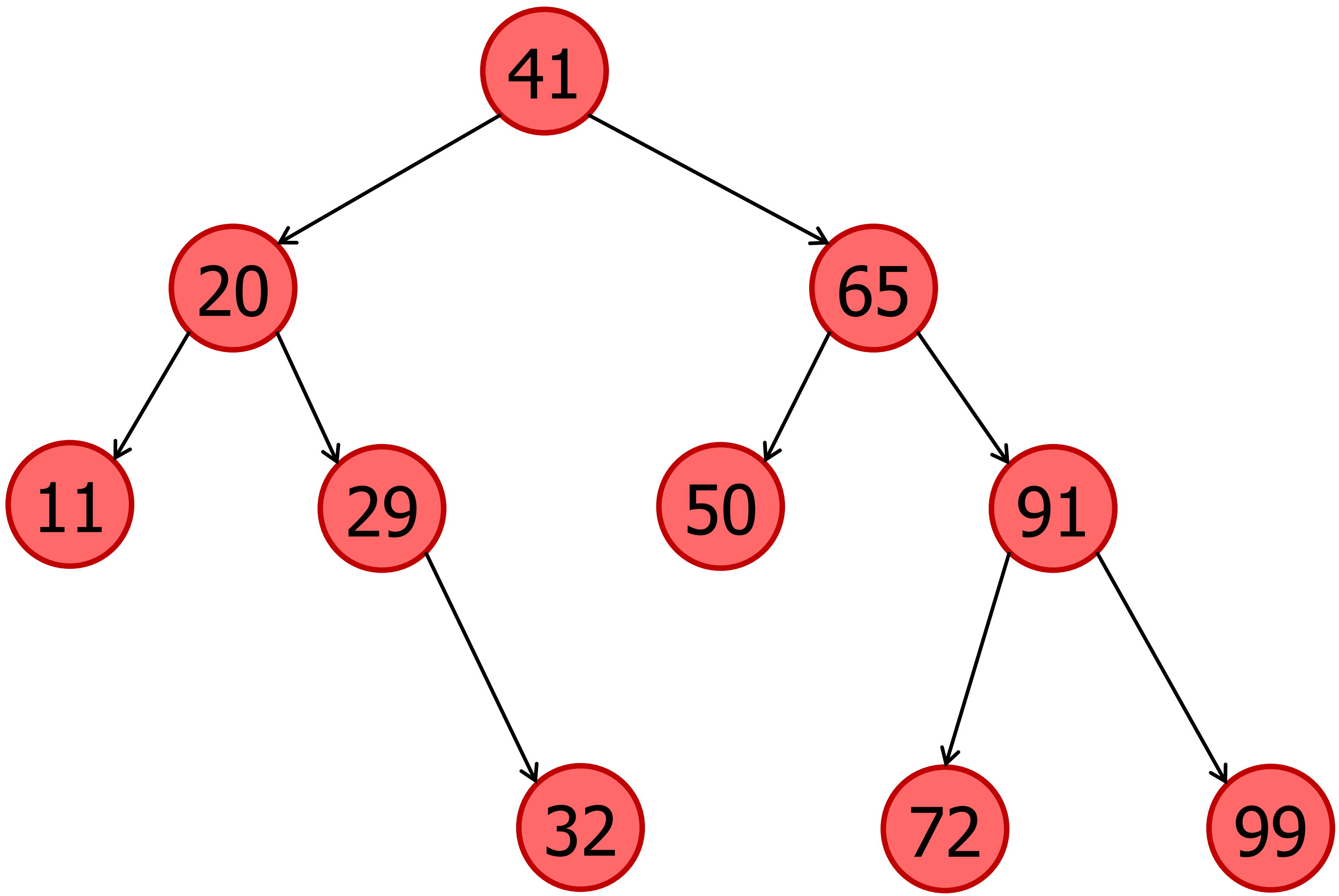
Tree Traversal

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();
}
```

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. $O(2^n)$

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. $O(2^n)$

Note: searching for all the items is going to be slower!

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)

- 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)

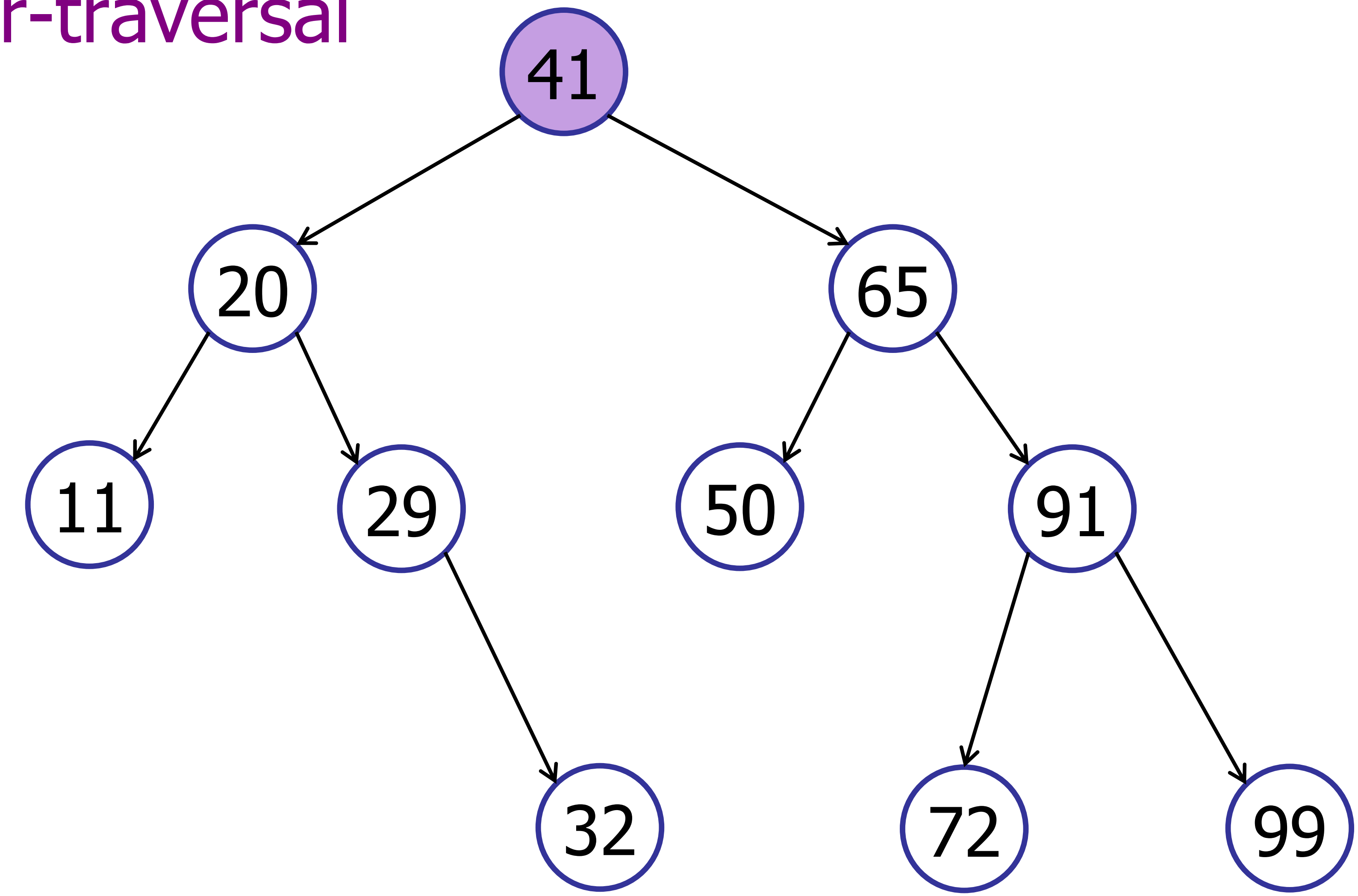
```
public void pre-order-traversal(){
        visit(this);

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

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

Tree Traversals

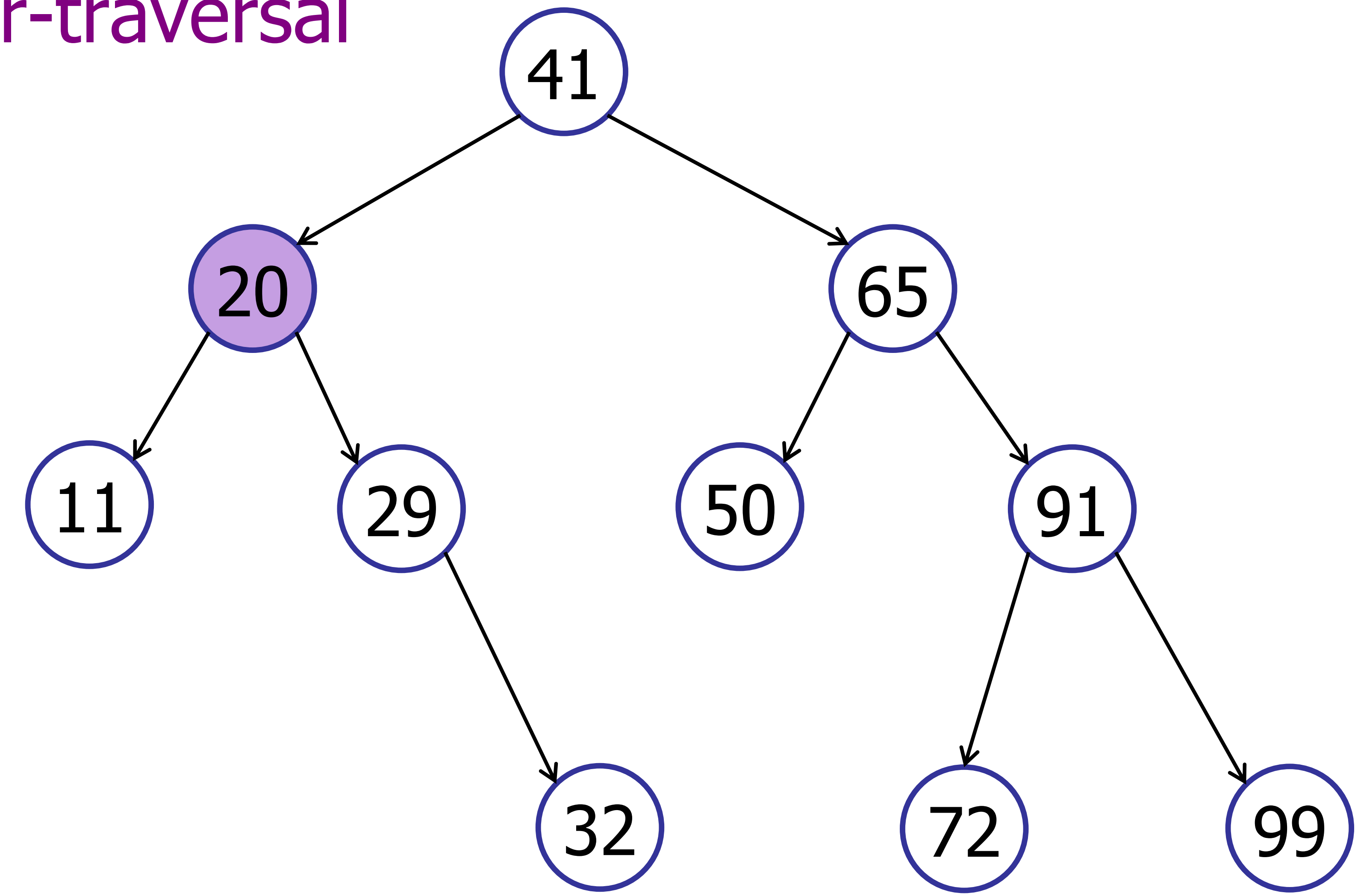
pre-order-traversal



41

Tree Traversals

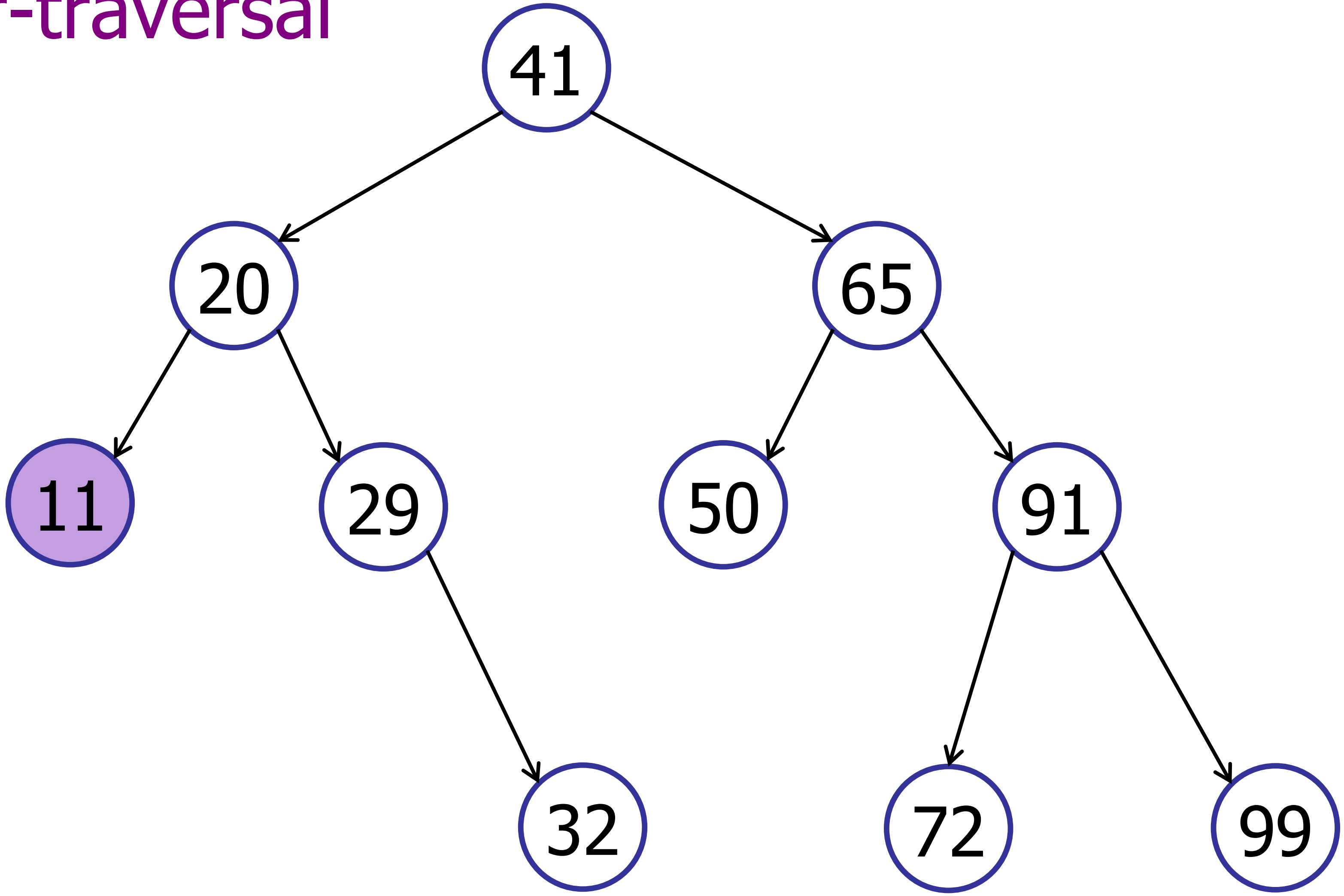
pre-order-traversal



41 20

Tree Traversals

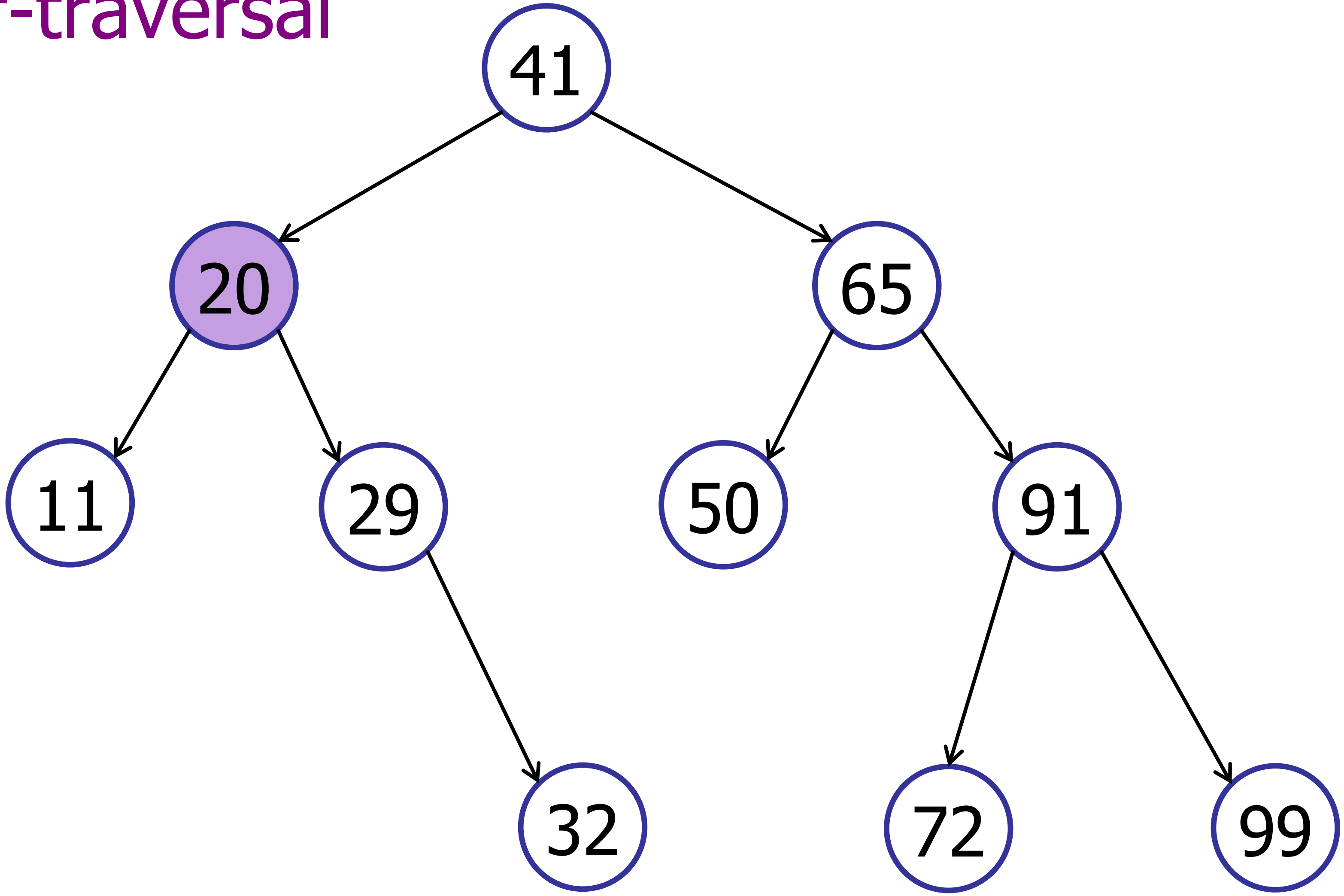
pre-order-traversal



41 20 11

Tree Traversals

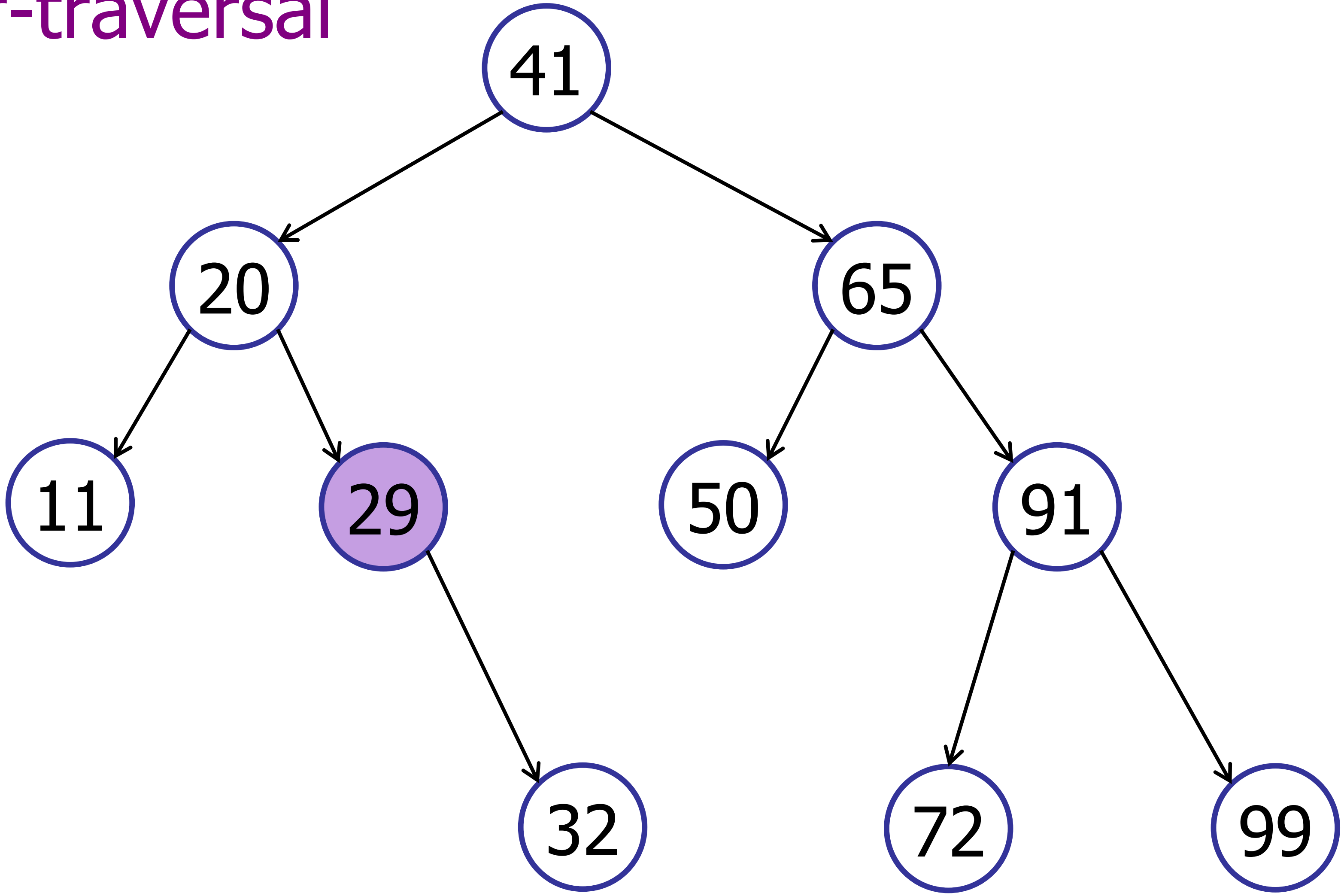
pre-order-traversal



41 20 11

Tree Traversals

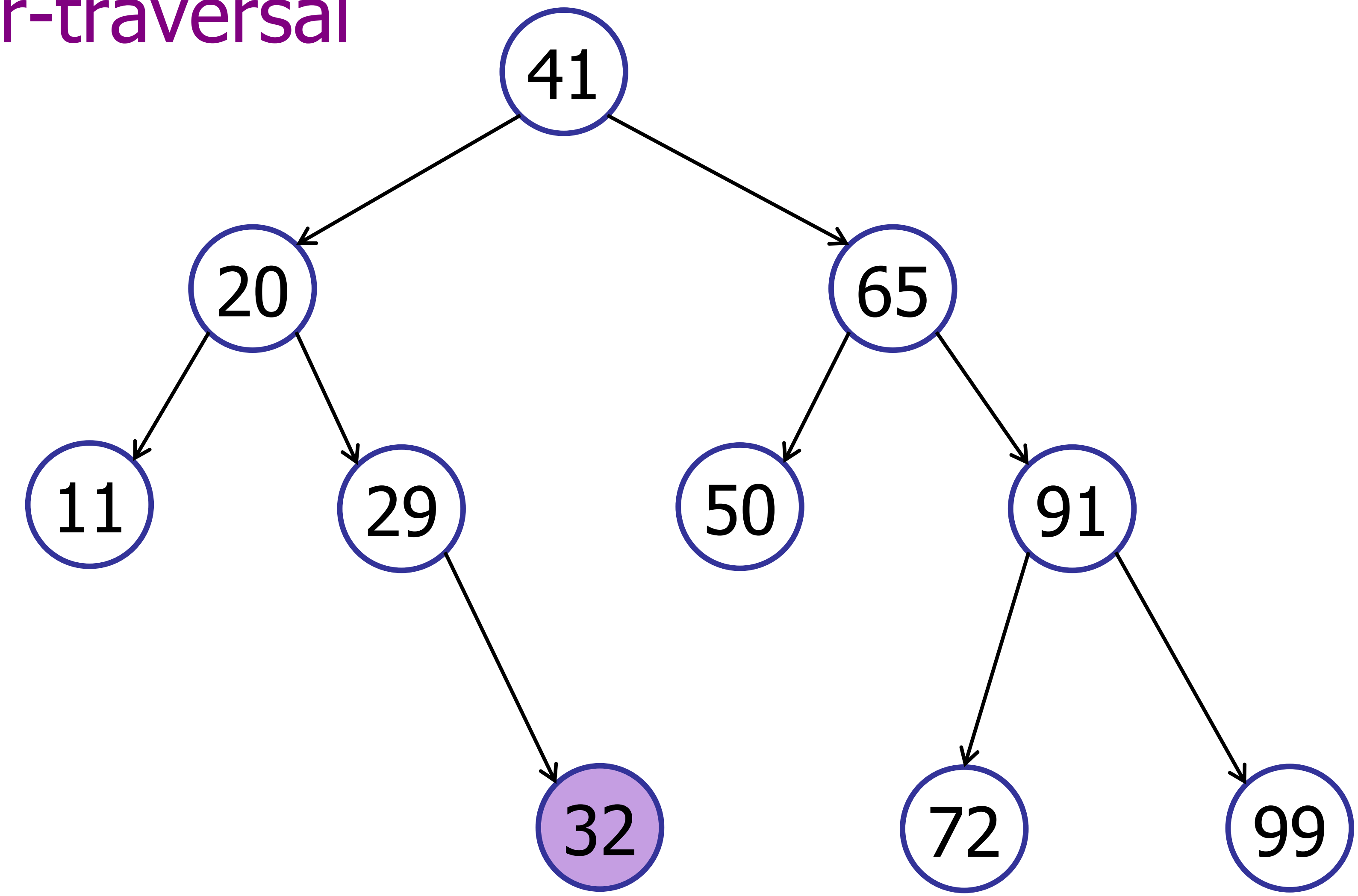
pre-order-traversal



41 20 11 29

Tree Traversals

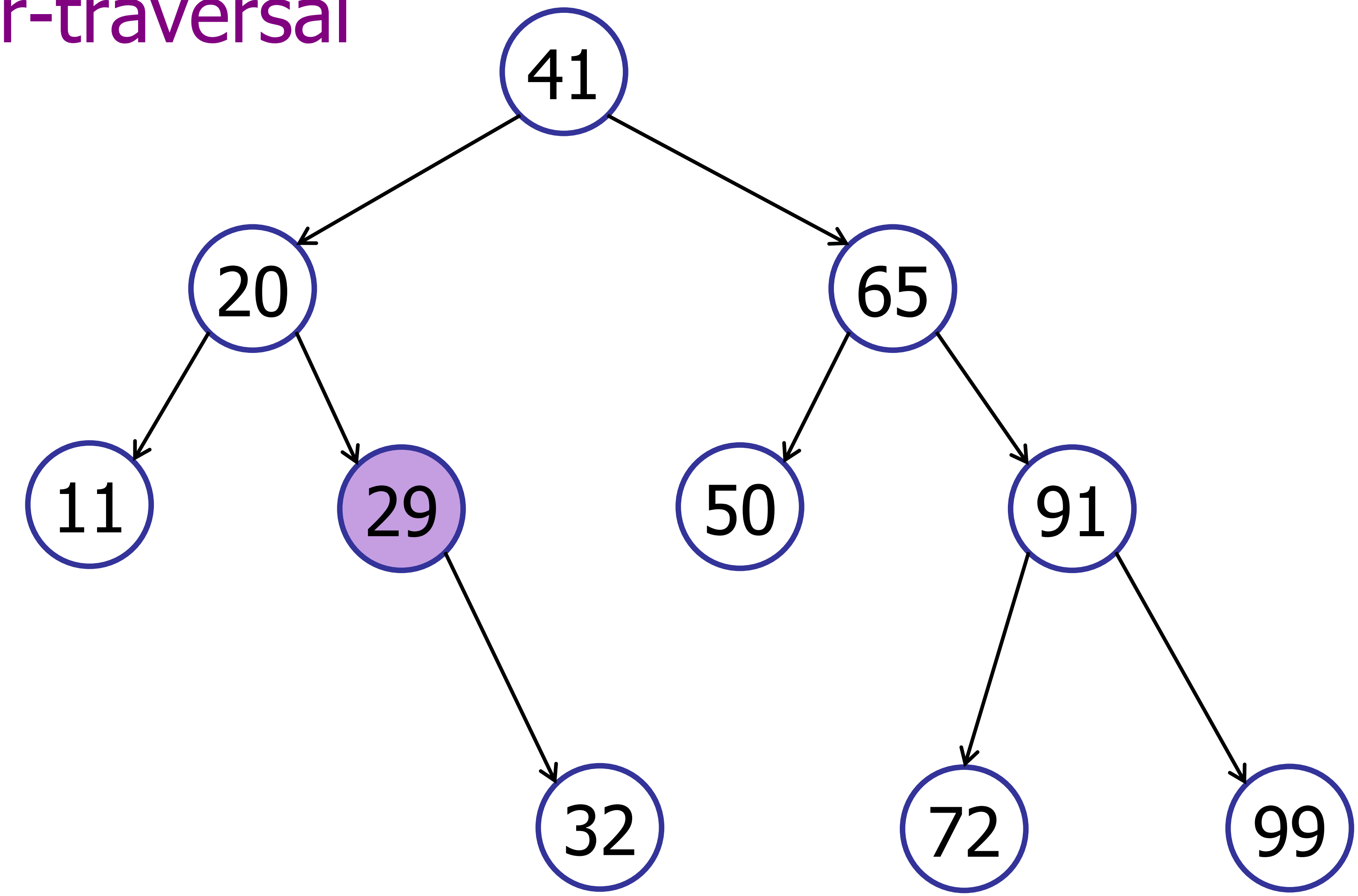
pre-order-traversal



41 20 11 29 32

Tree Traversals

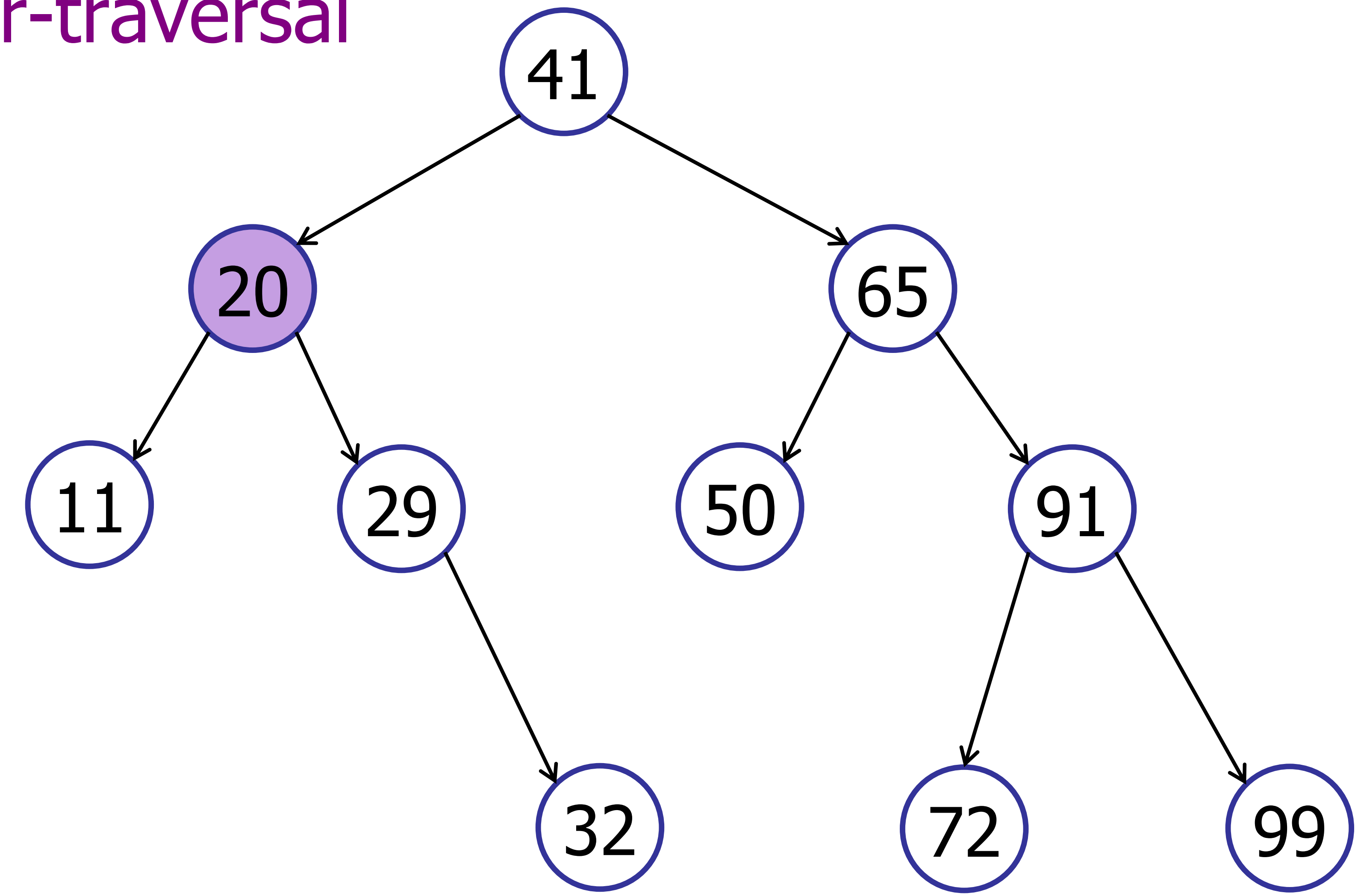
pre-order-traversal



41 20 11 29 32

Tree Traversals

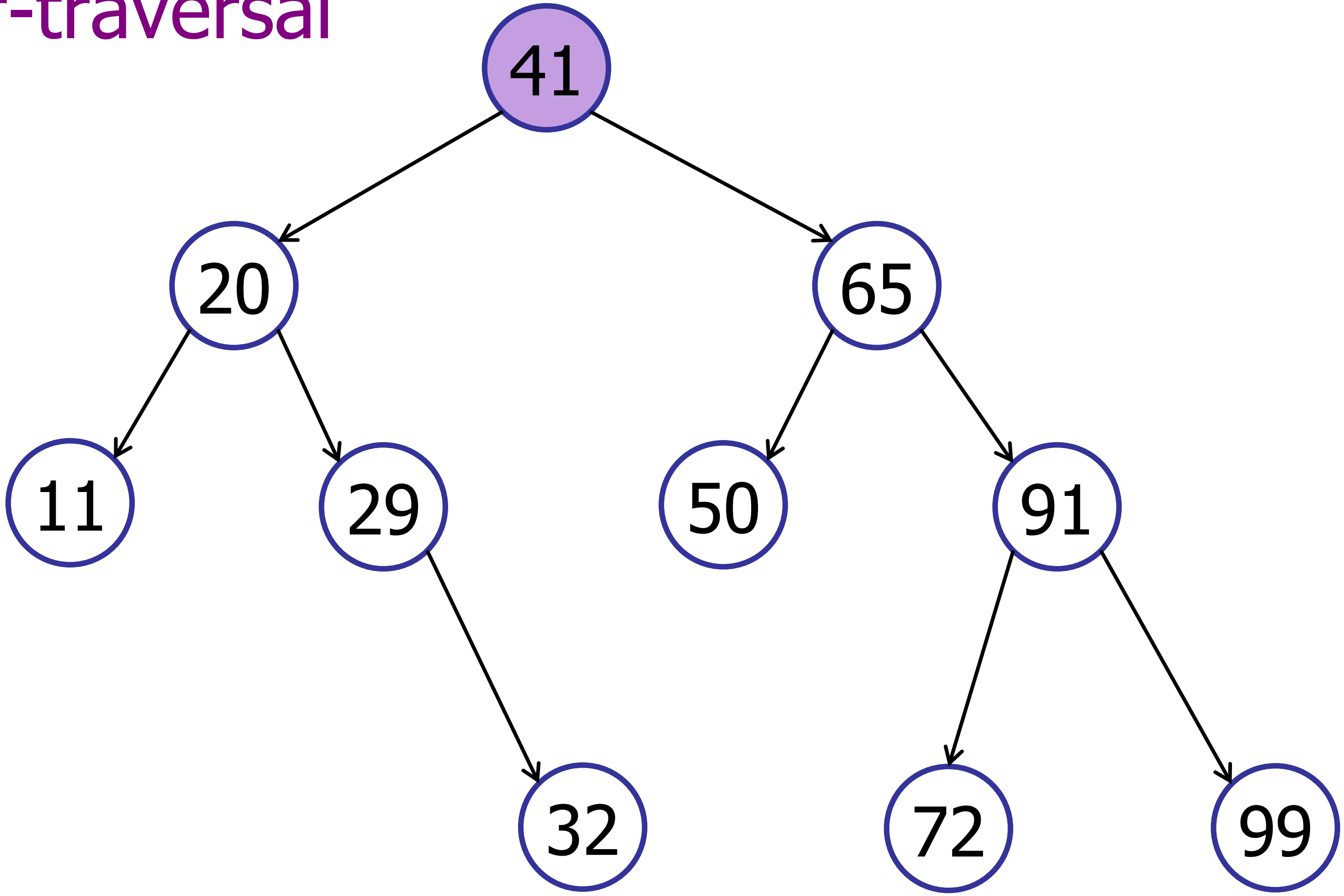
pre-order-traversal



41 20 11 29 32

Tree Traversals

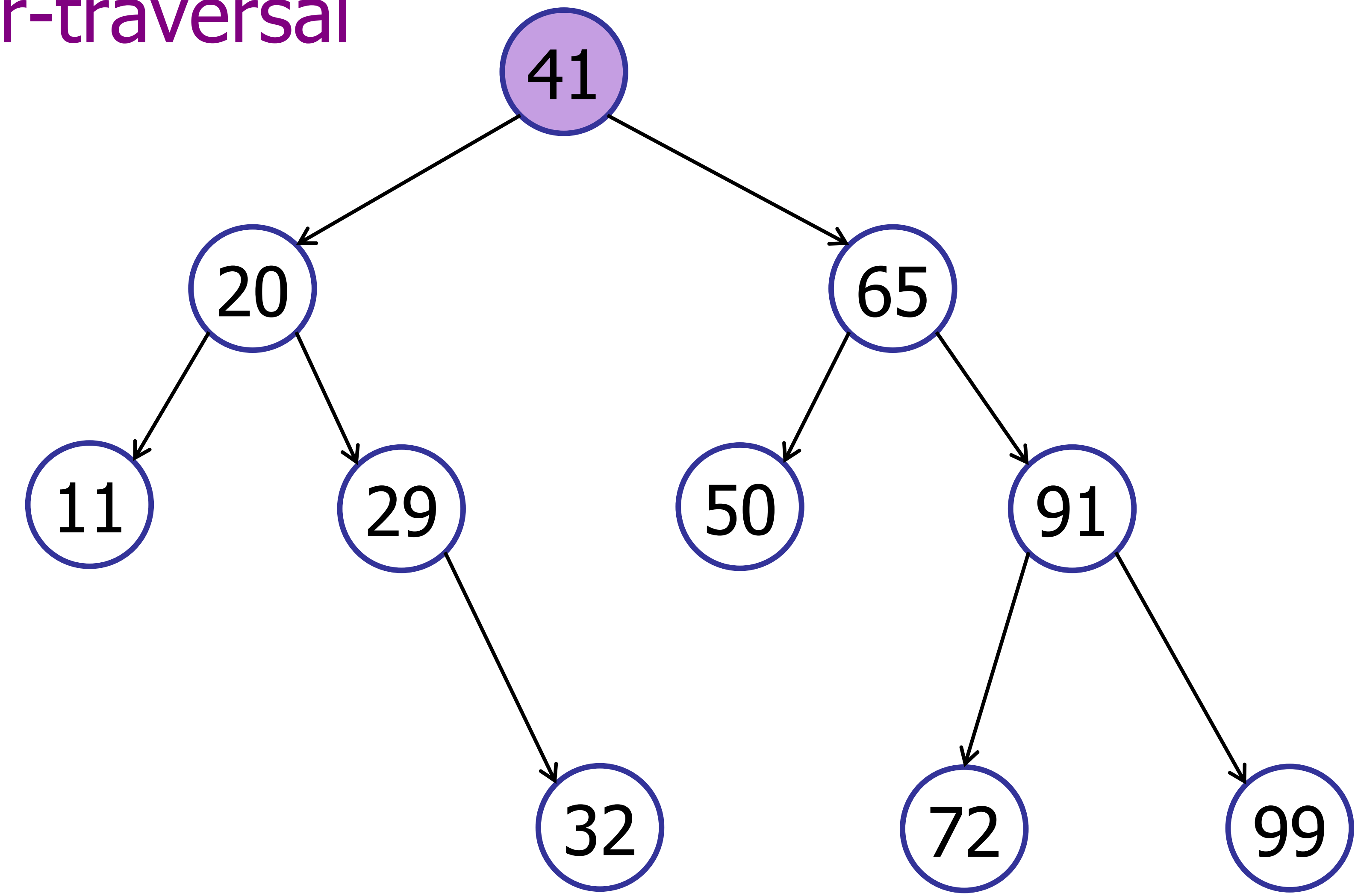
pre-order-traversal



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)

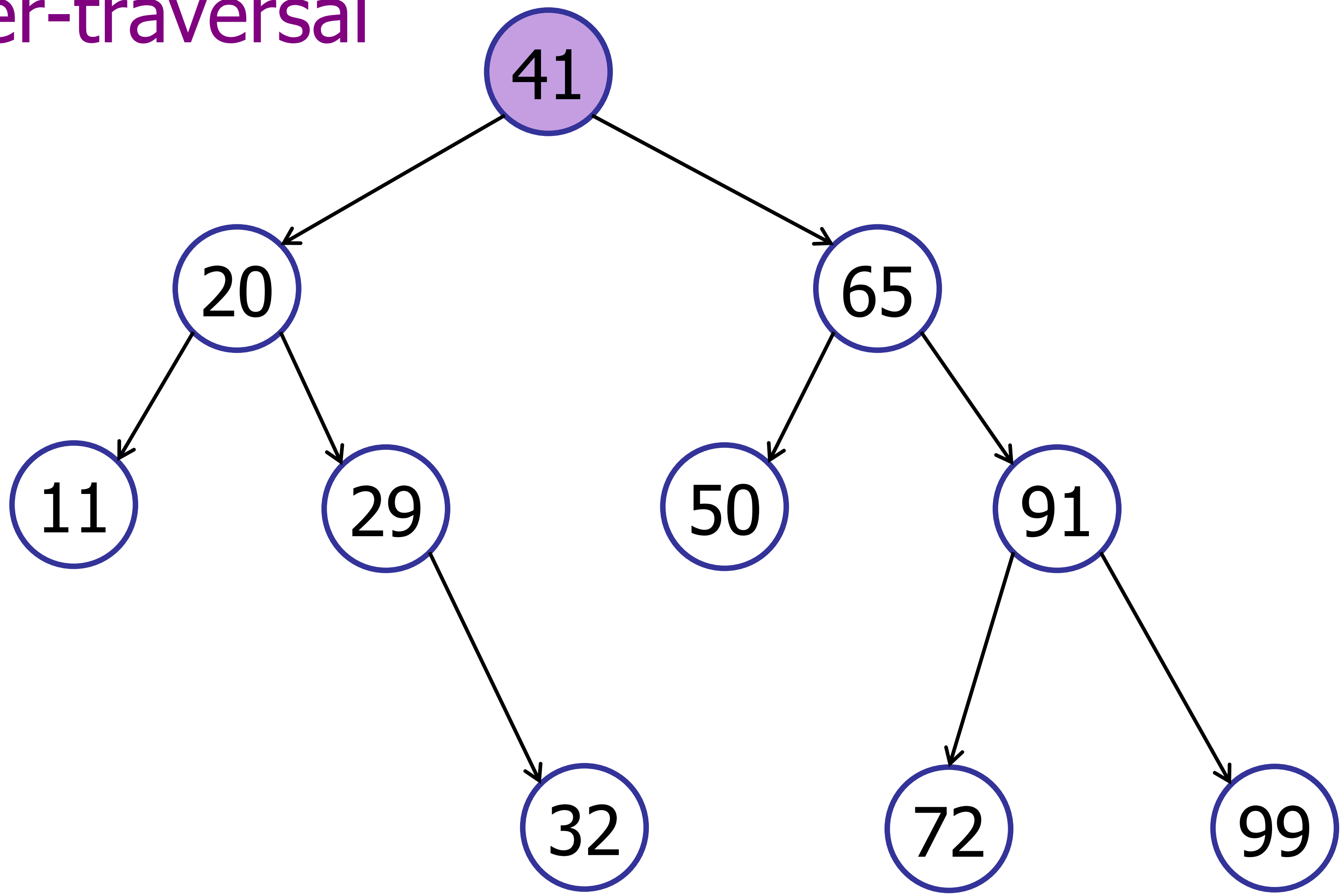
```
public void post-order-traversal(){
       // Traverse left sub-tree
       if (leftTree != null)
               leftTree.in-order-traversal();

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

       visit(this);
 }
```

Tree Traversals

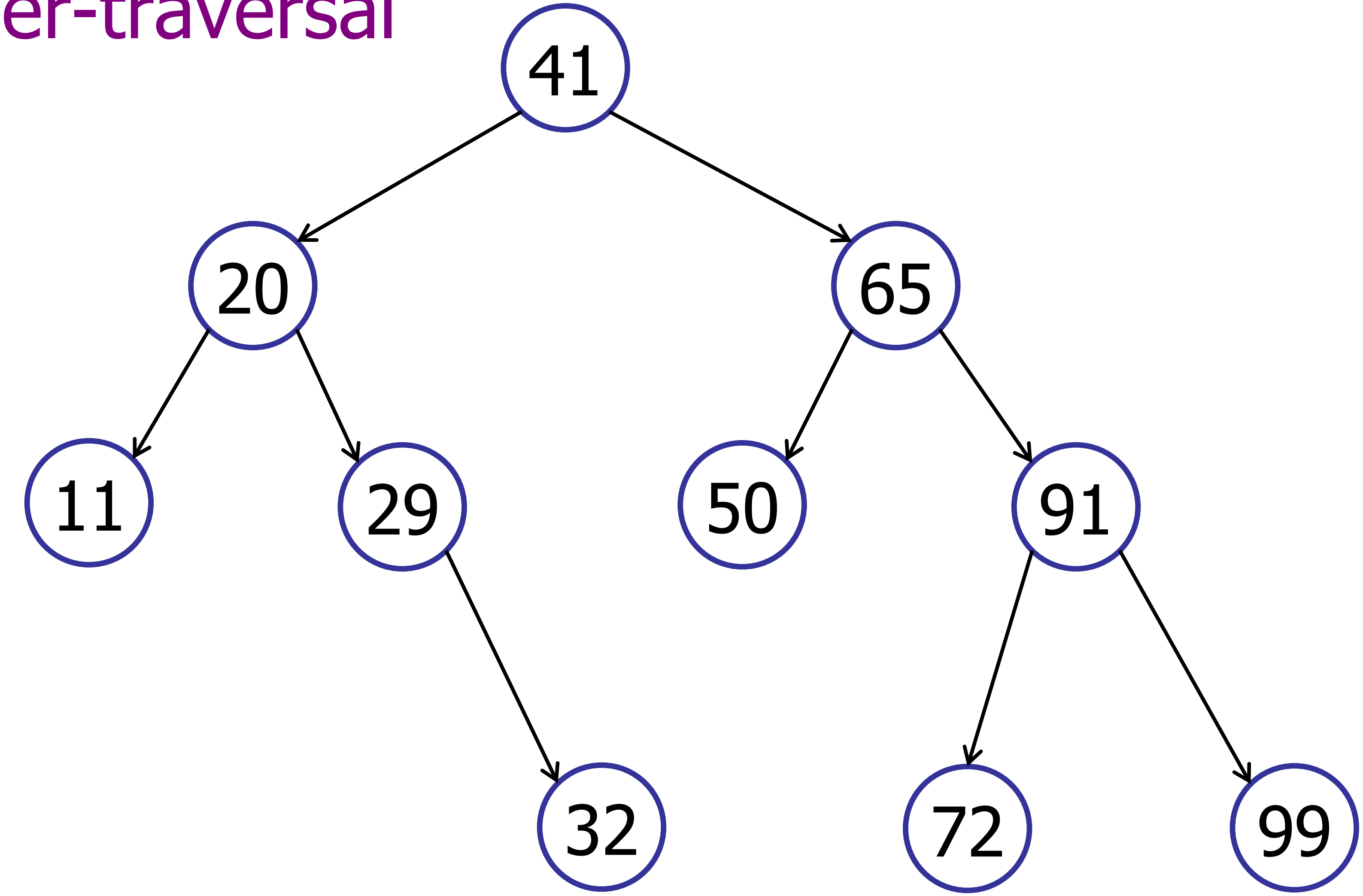
post-order-traversal



11 32 29 20 50 72 99 91 65 41

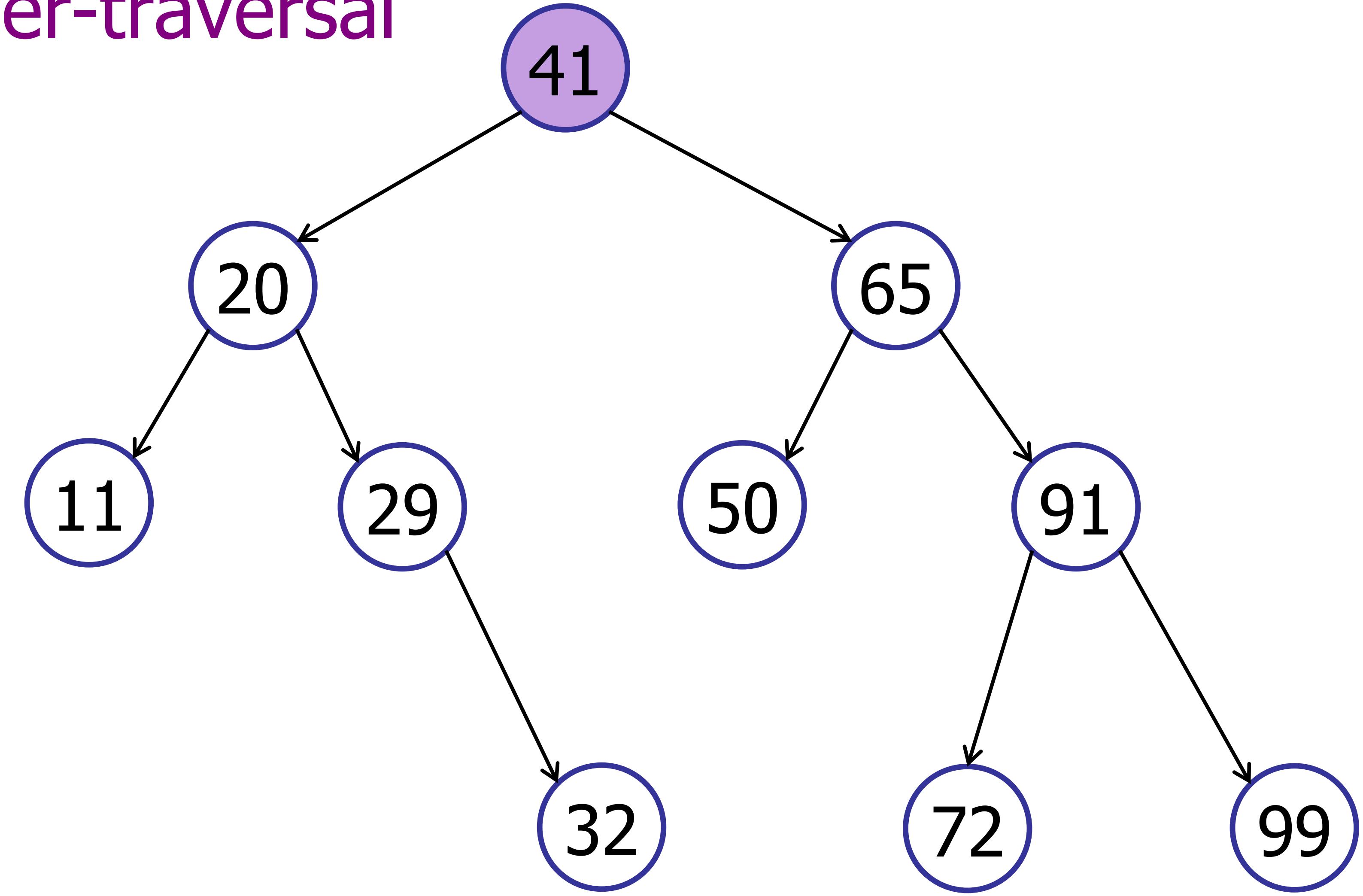
Tree Traversals

level-order-traversal



Tree Traversals

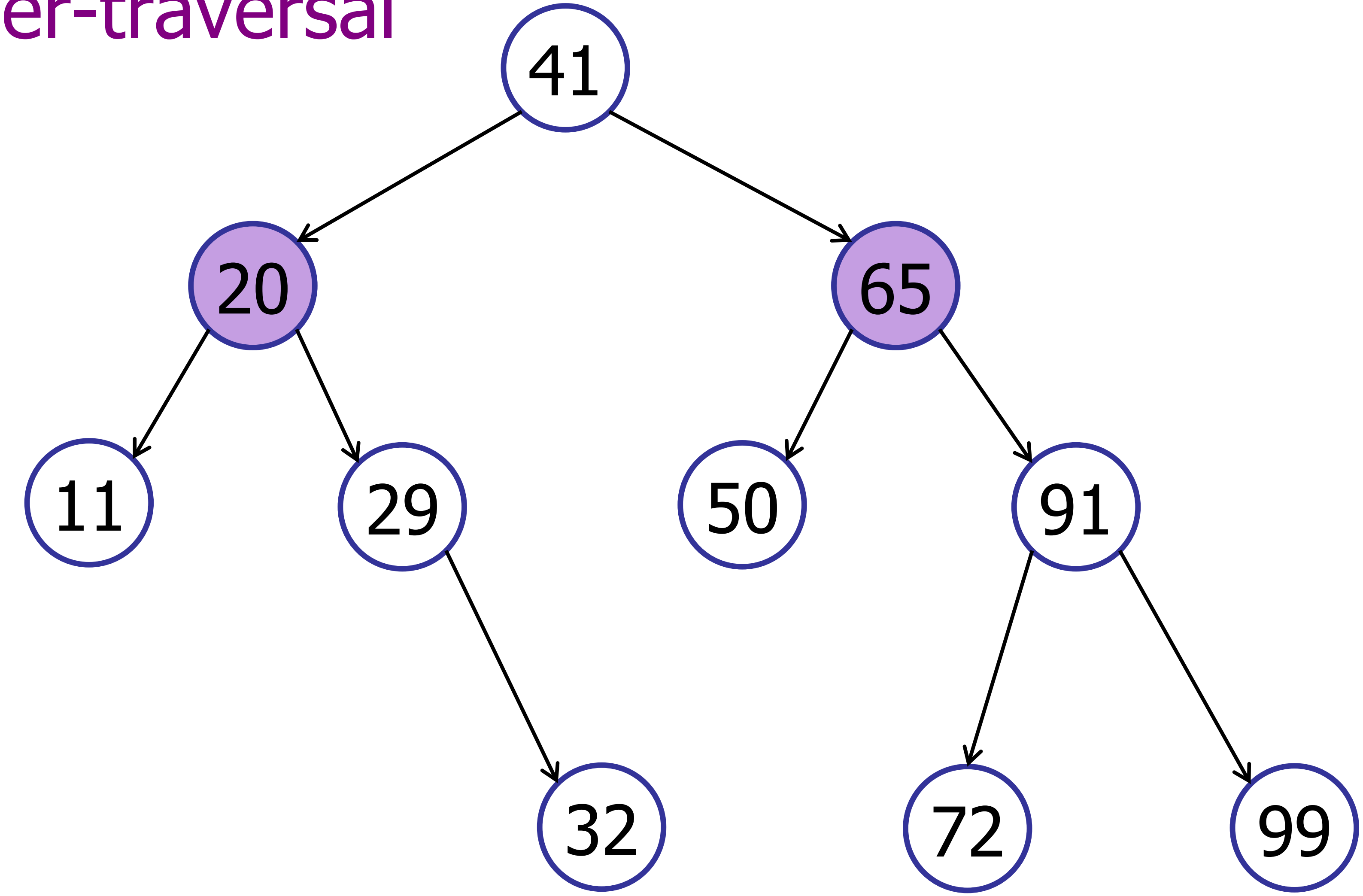
level-order-traversal



41

Tree Traversals

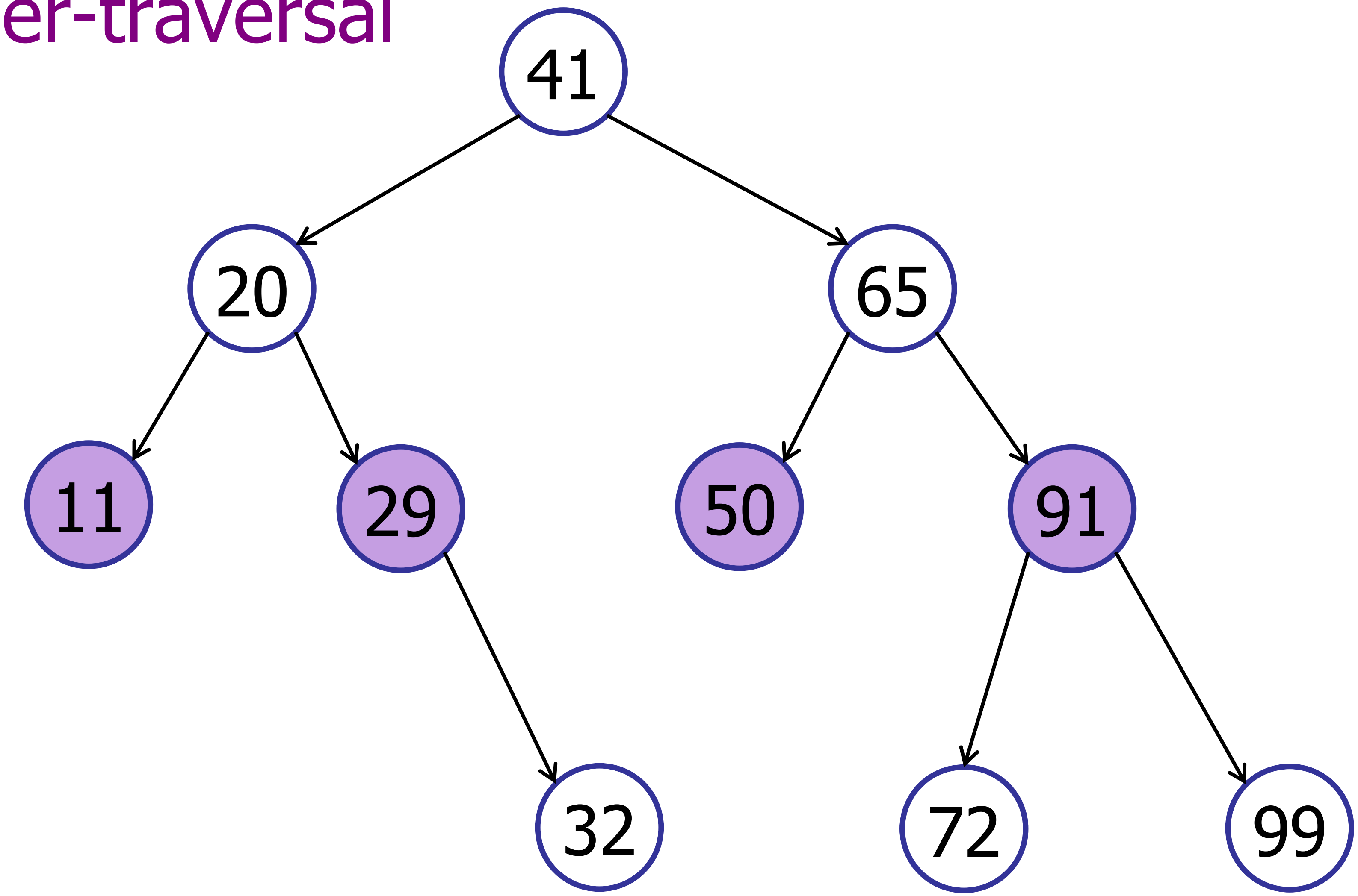
level-order-traversal



41 20 65

Tree Traversals

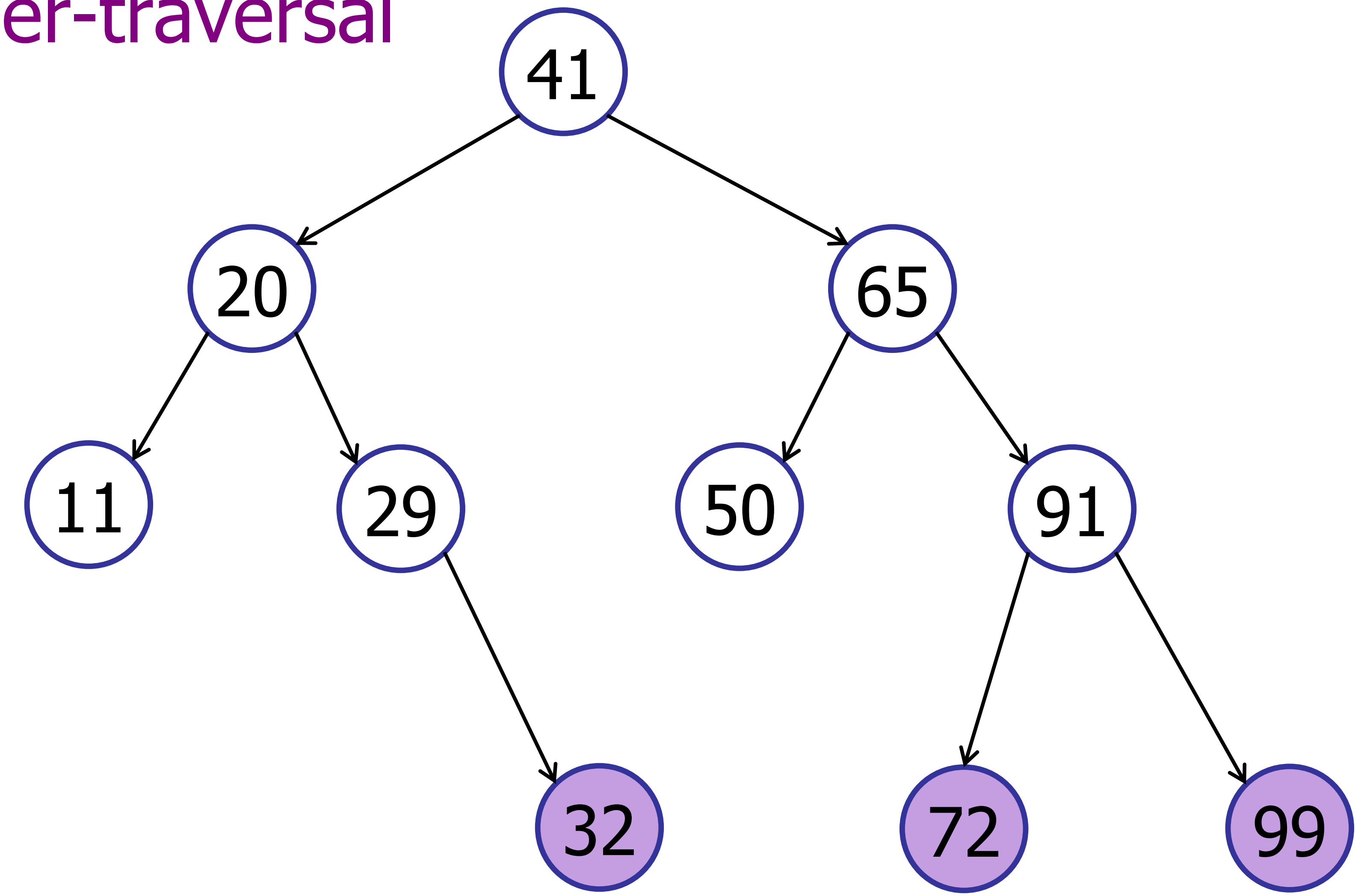
level-order-traversal



41 20 65 11 29 50 91

Tree Traversals

level-order-traversal



41 20 65 11 29 50 91 32 72 99

Tree Traversals

Several varieties:

- pre-order
- in-order
- post-order
- level-order

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

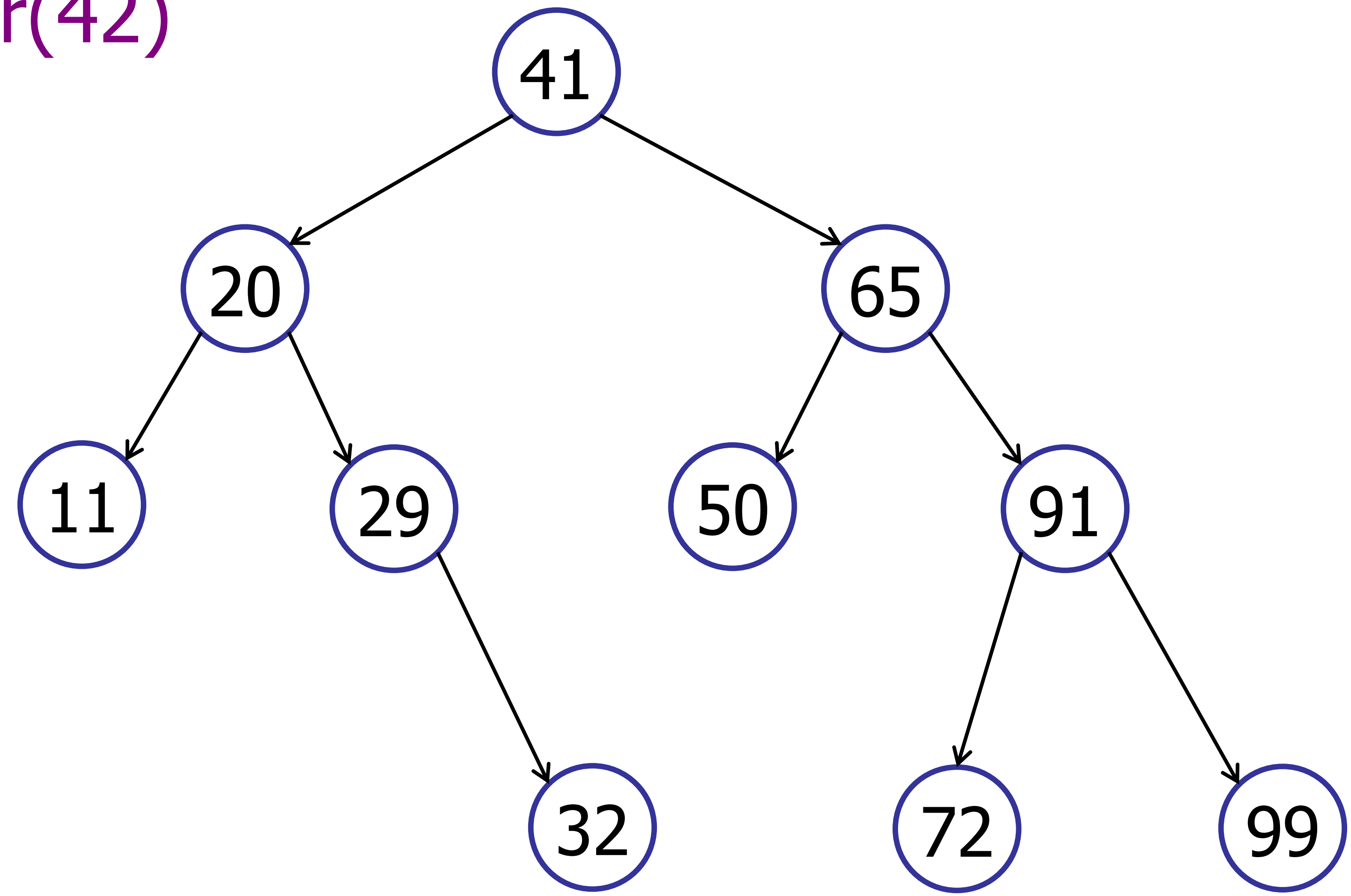
6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- successor(8:24) = 12:21

How do we implement this?

Successor: Key not in the Tree

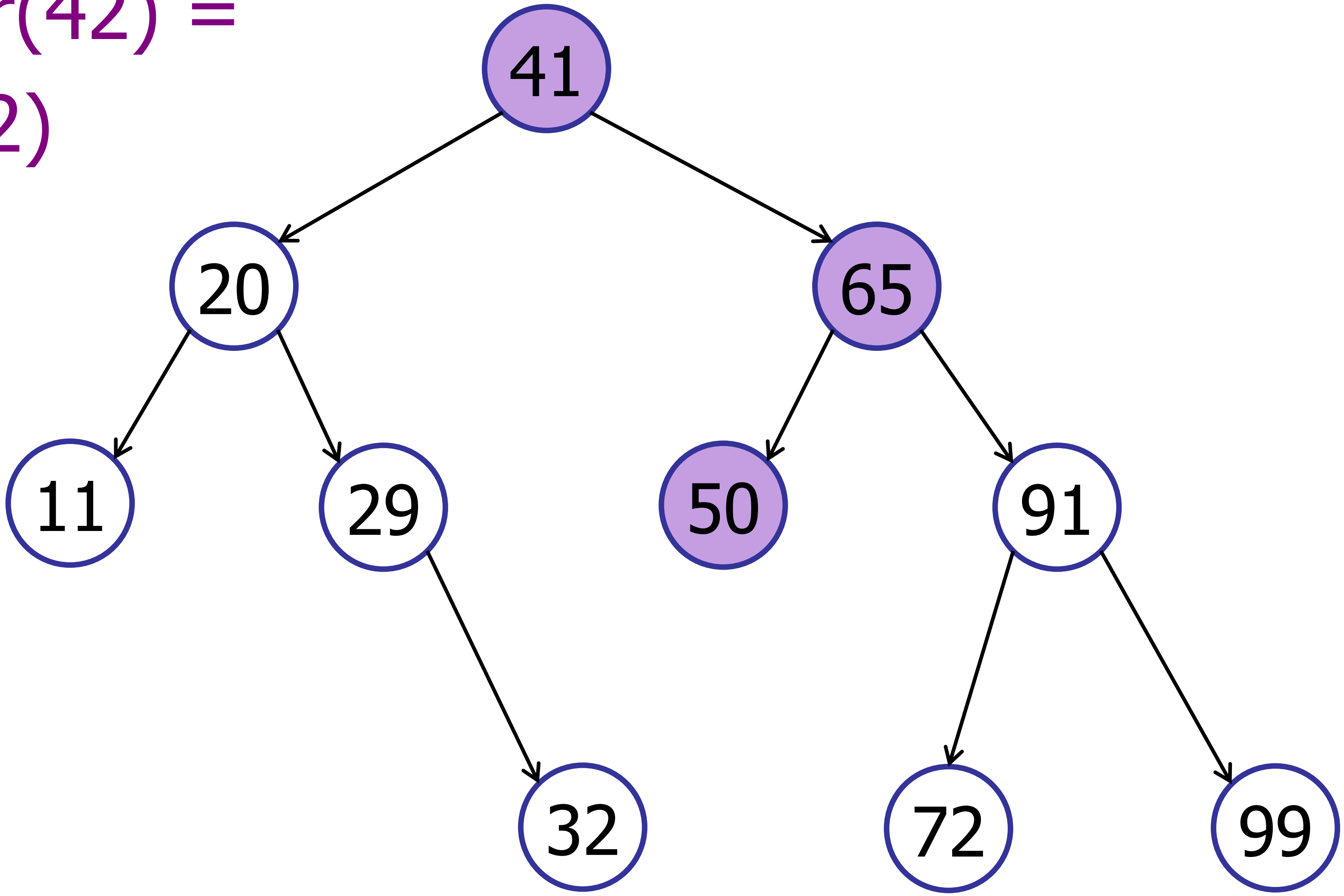
successor(42)



Key 42 is not in the tree

Successor: Key not in the Tree

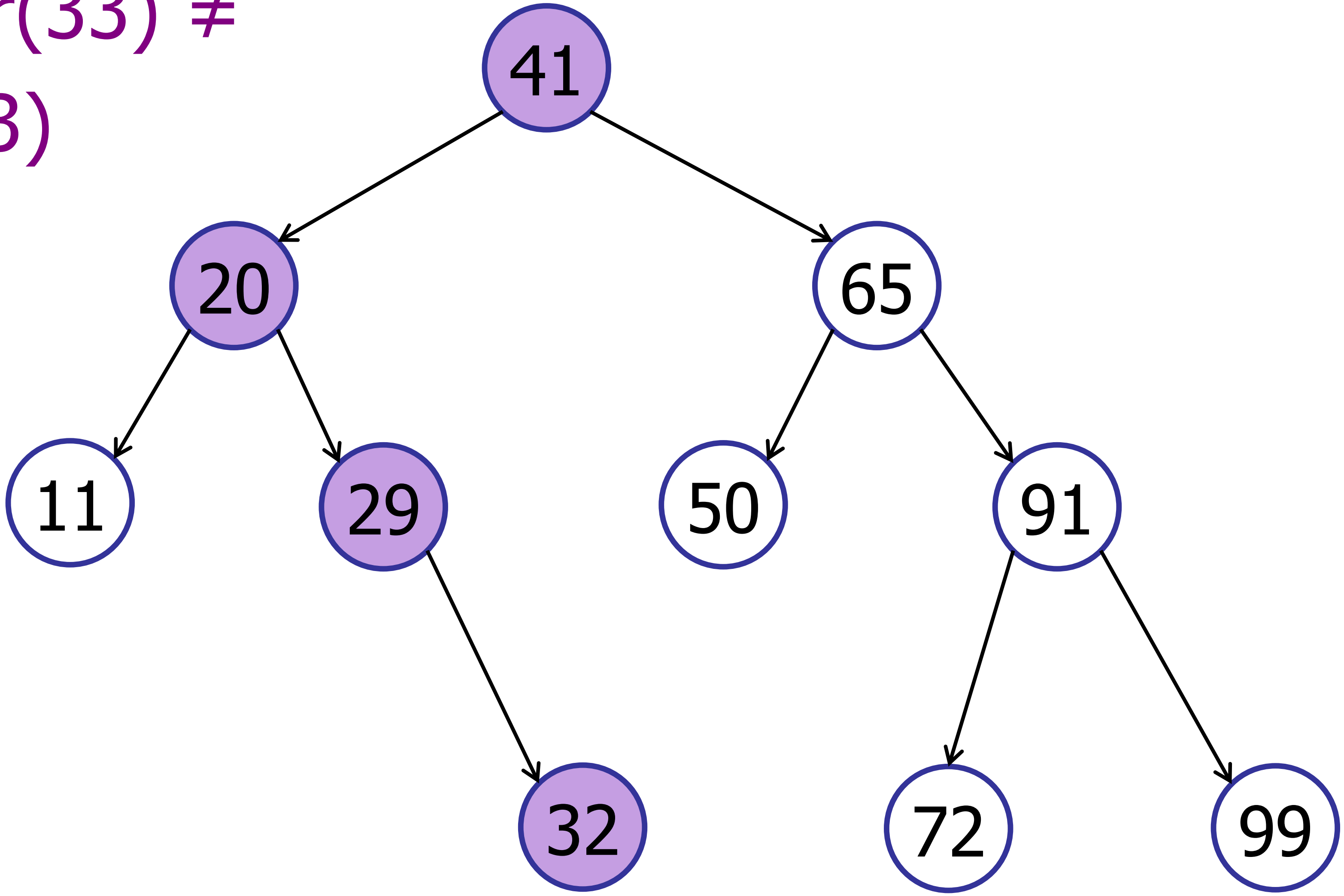
successor(42) =
search(42)



Key 42 is not in the tree

Successor: Key not in the Tree

successor(33) ≠
search(33)

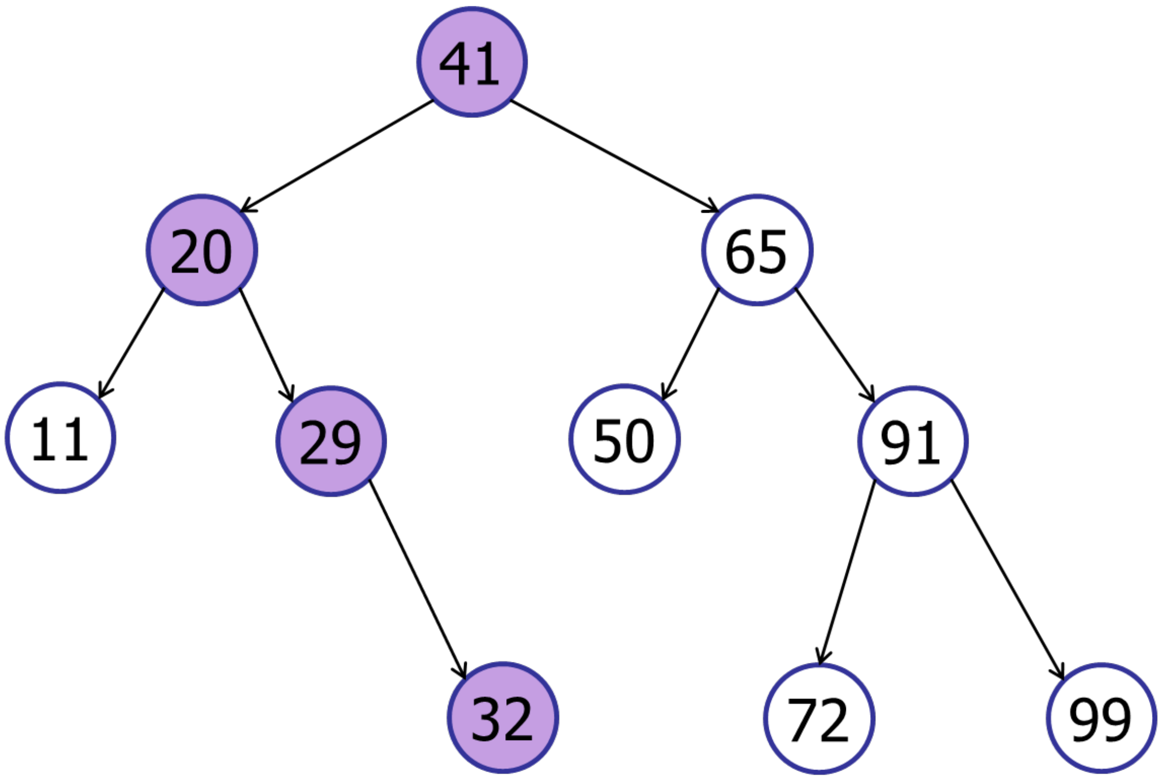


Key 33 is not in the tree

Successor: Key not in the Tree

If you search for a key not in the tree:

➔ either find predecessor
or successor.



Successor Queries

Basic strategy: successor(key)

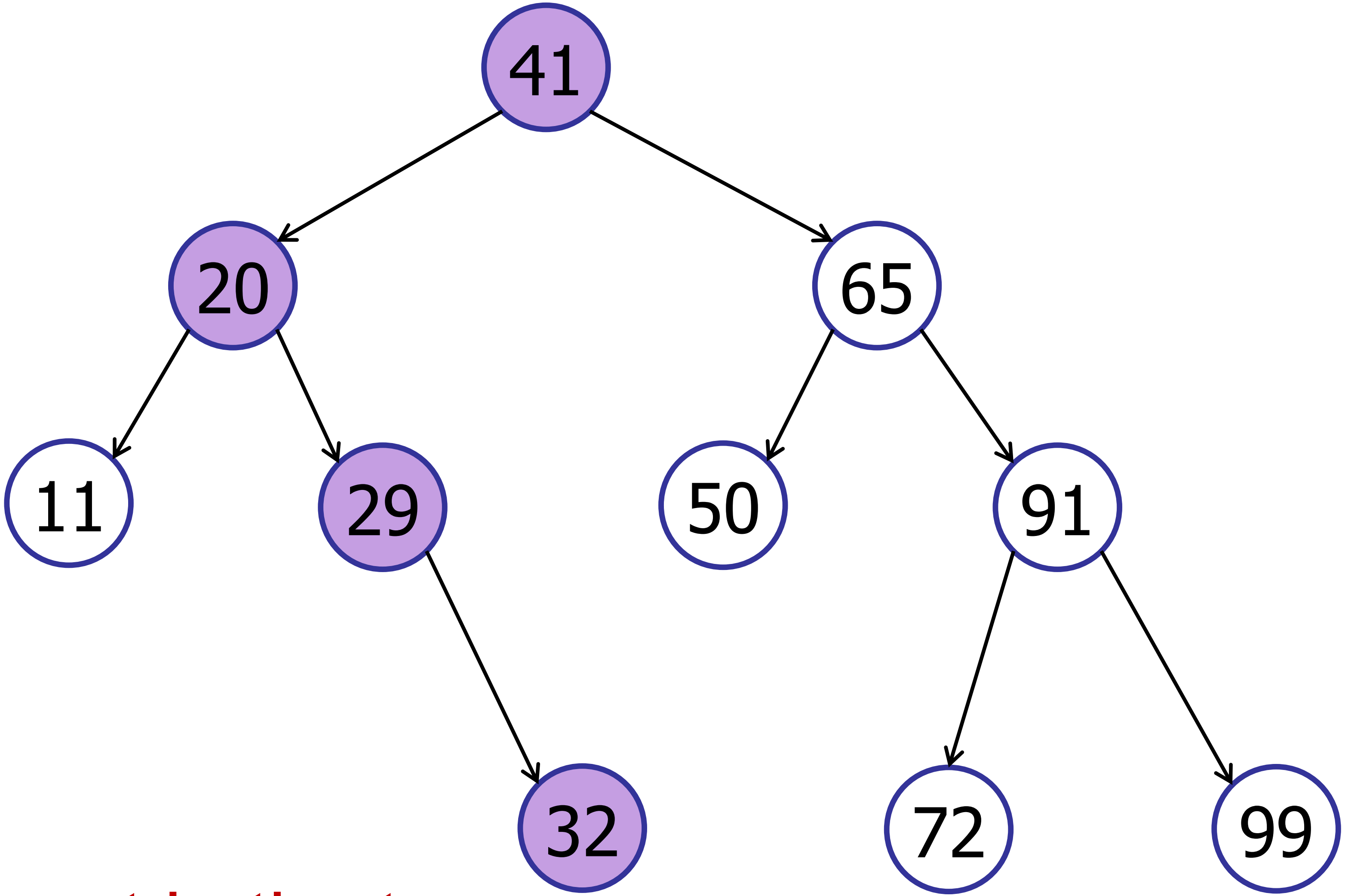
1. Search for key in the tree.

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

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

Successor: Key not in the Tree

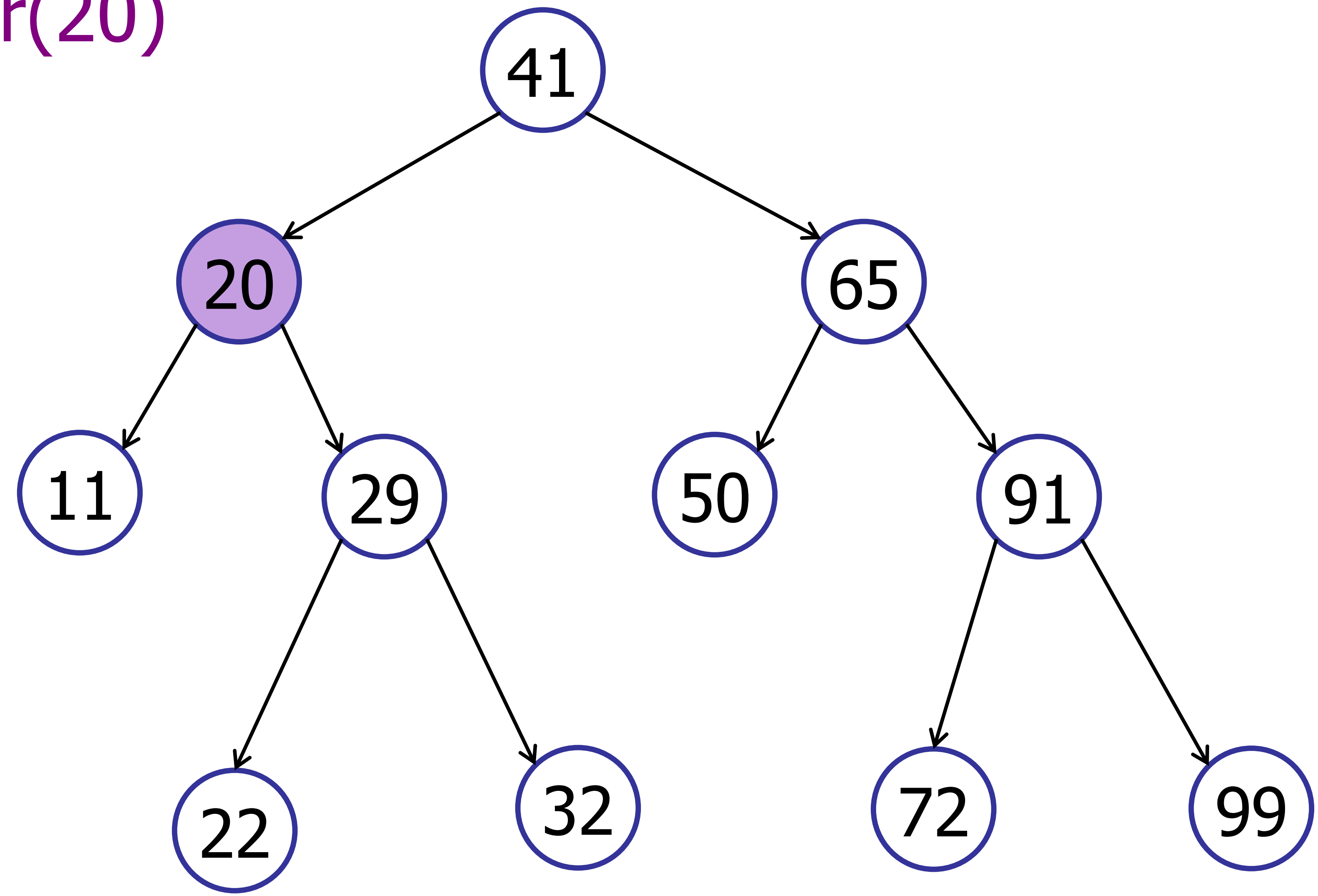
successor(33) = successor(32)



Key 33 is not in the tree

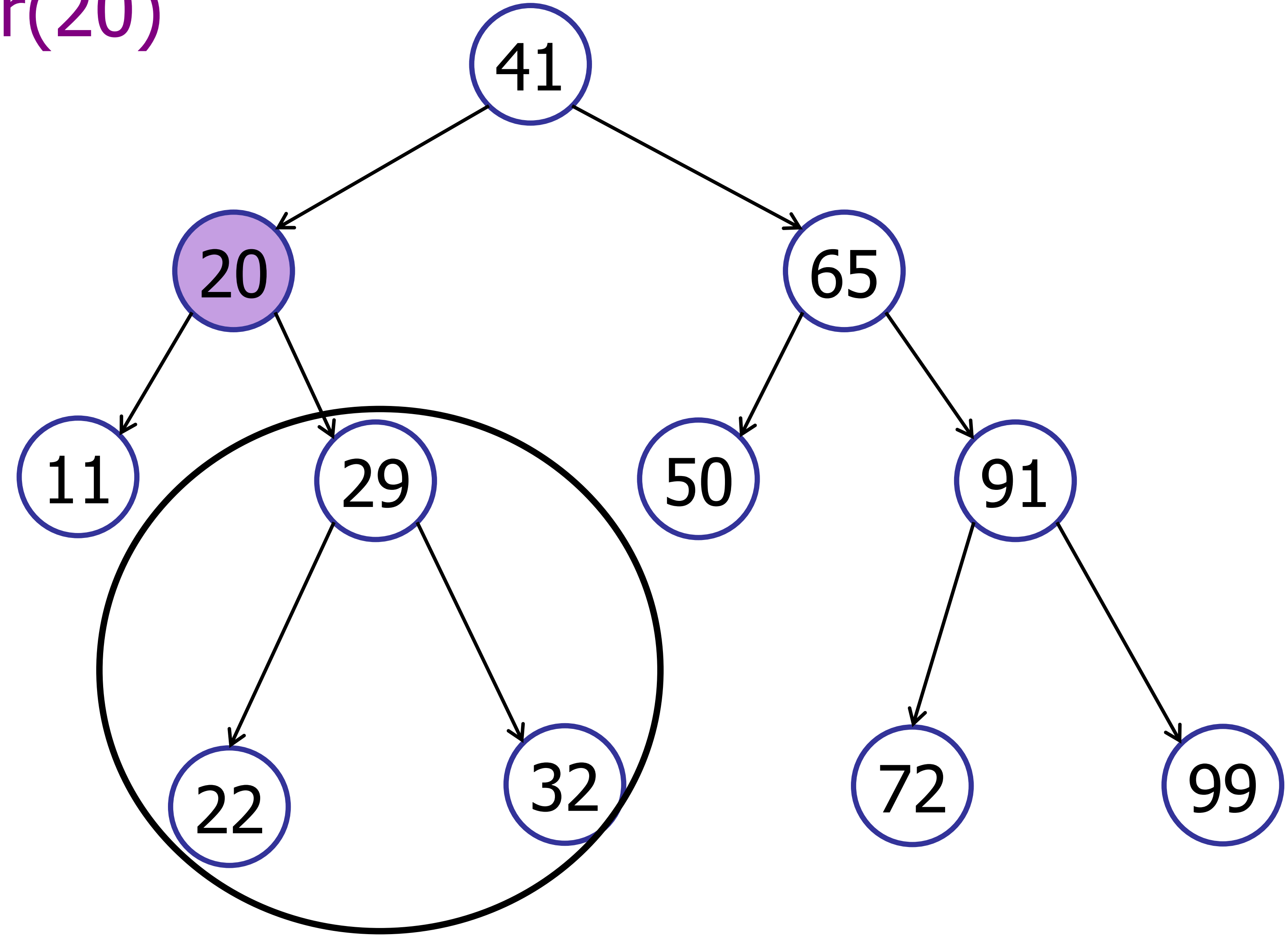
Successor: Key in the Tree

successor(20)



Successor Queries

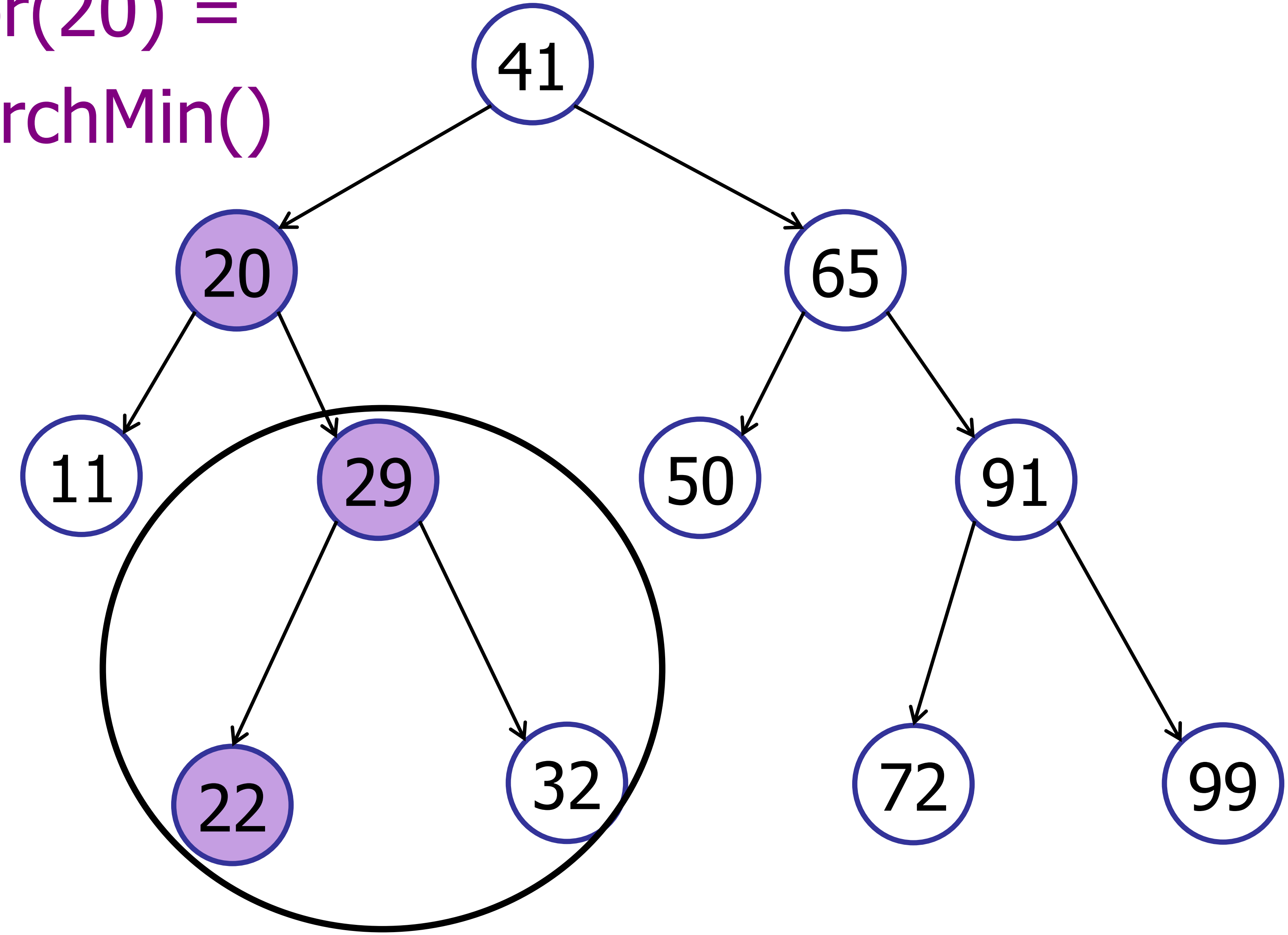
successor(20)



Case 1: node has a right child.

Successor Queries

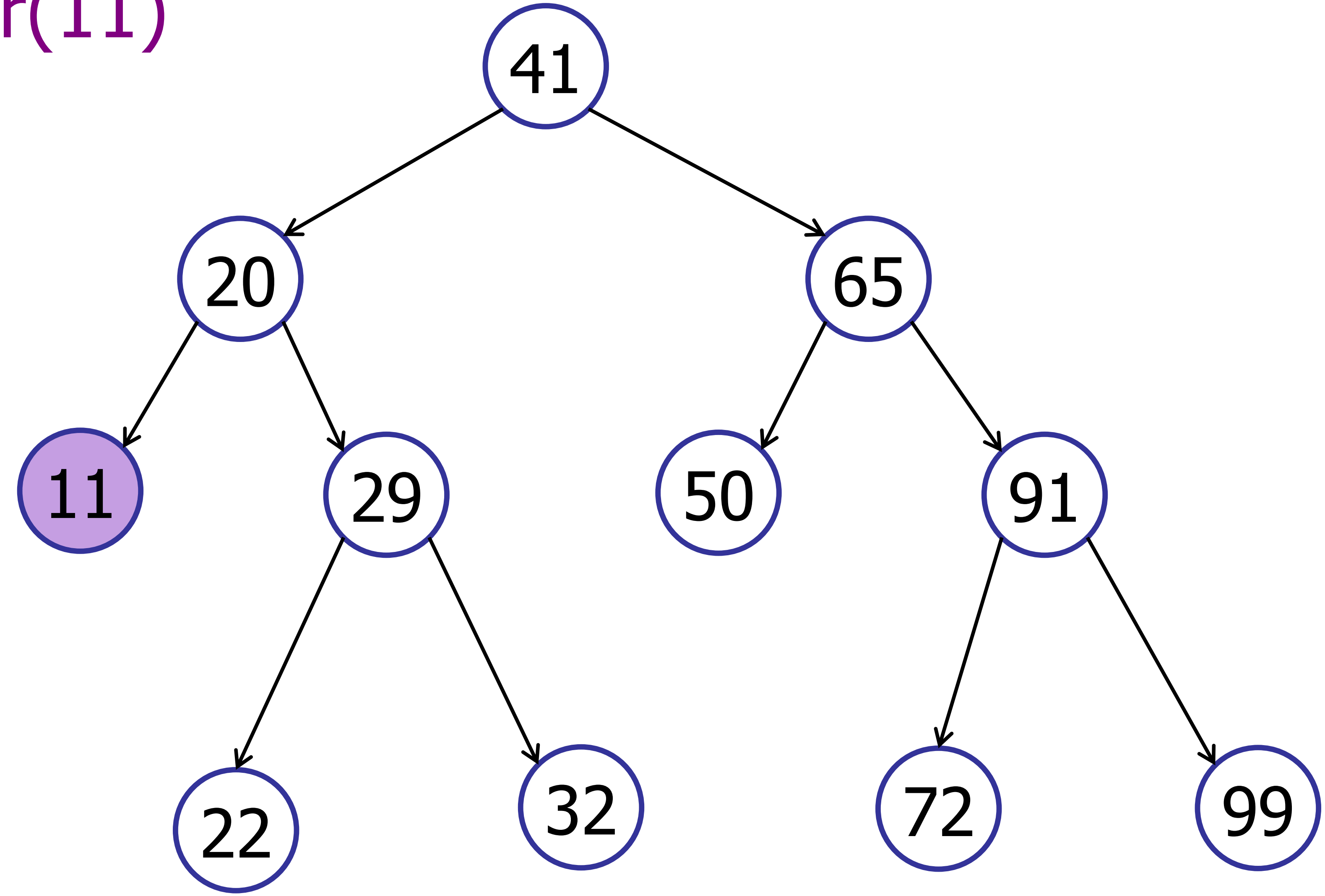
successor(20) =
right.searchMin()



Case 1: node has a right child.

Successor Queries

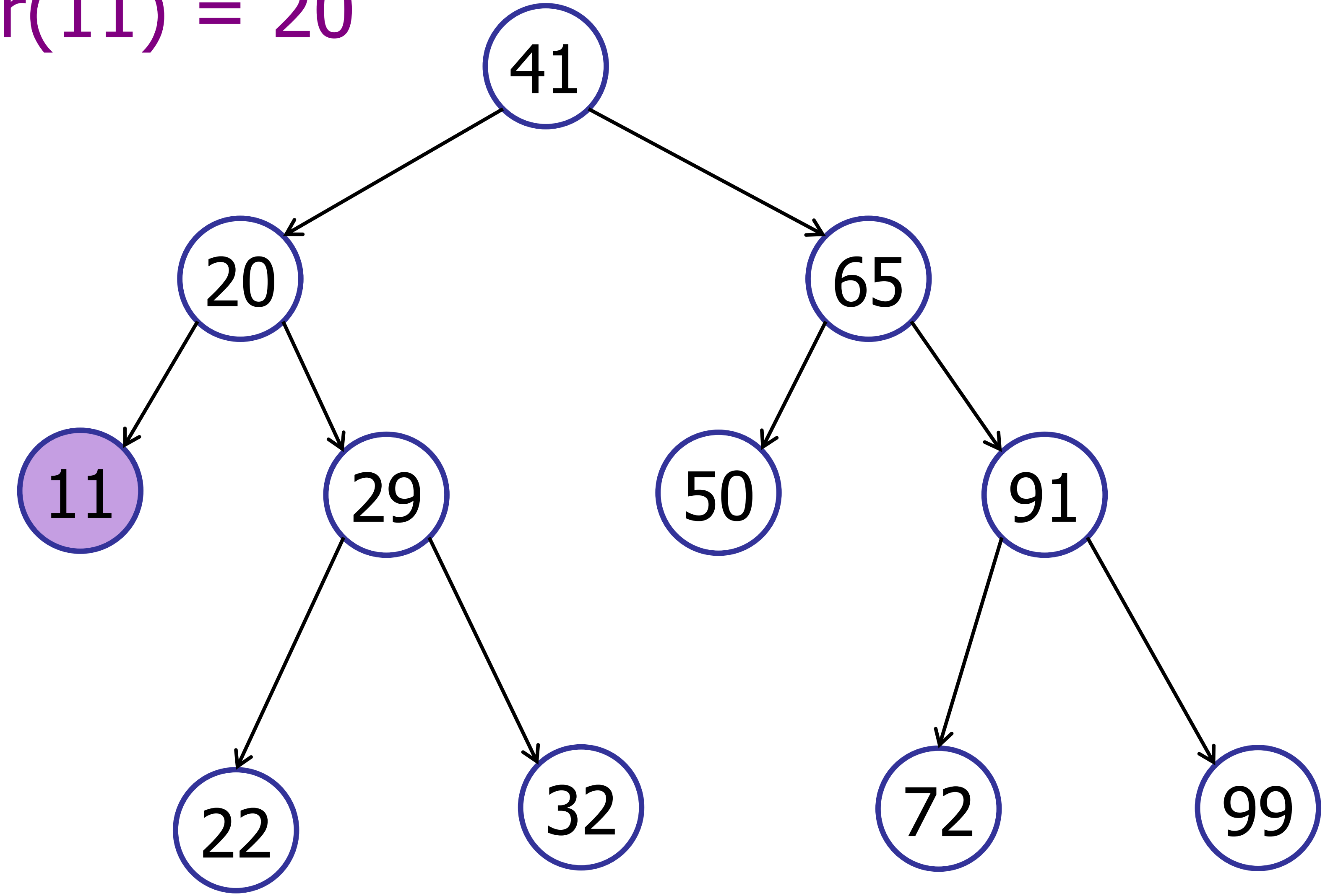
successor(11)



Case 2: node has no right child.

Successor Queries

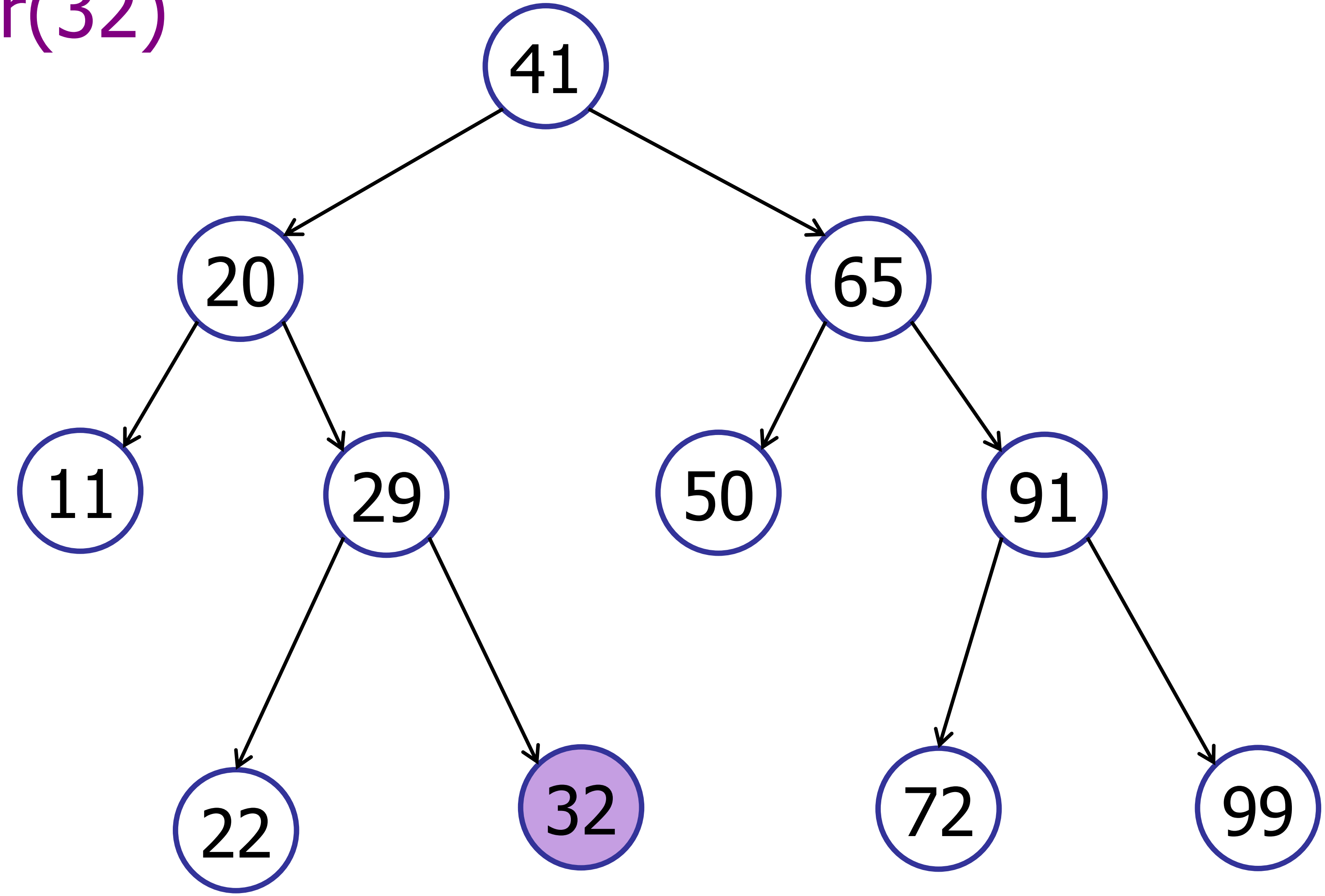
successor(11) = 20



Case 2: node has no right child.

Successor Queries

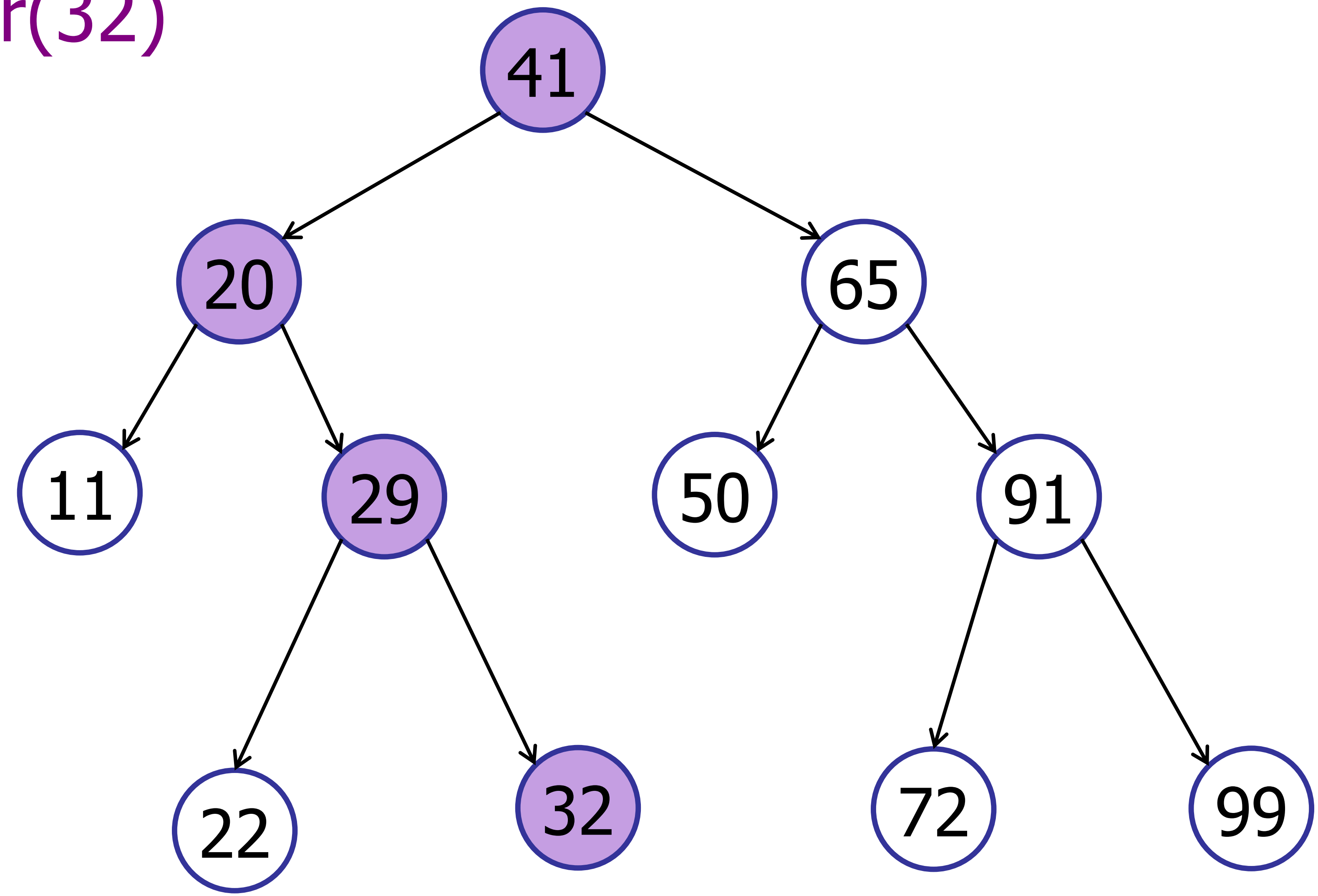
successor(32)



Case 2: node has no right child.

Successor Queries

successor(32)



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;
}
```

Binary Search Trees

1. Terminology and Definitions

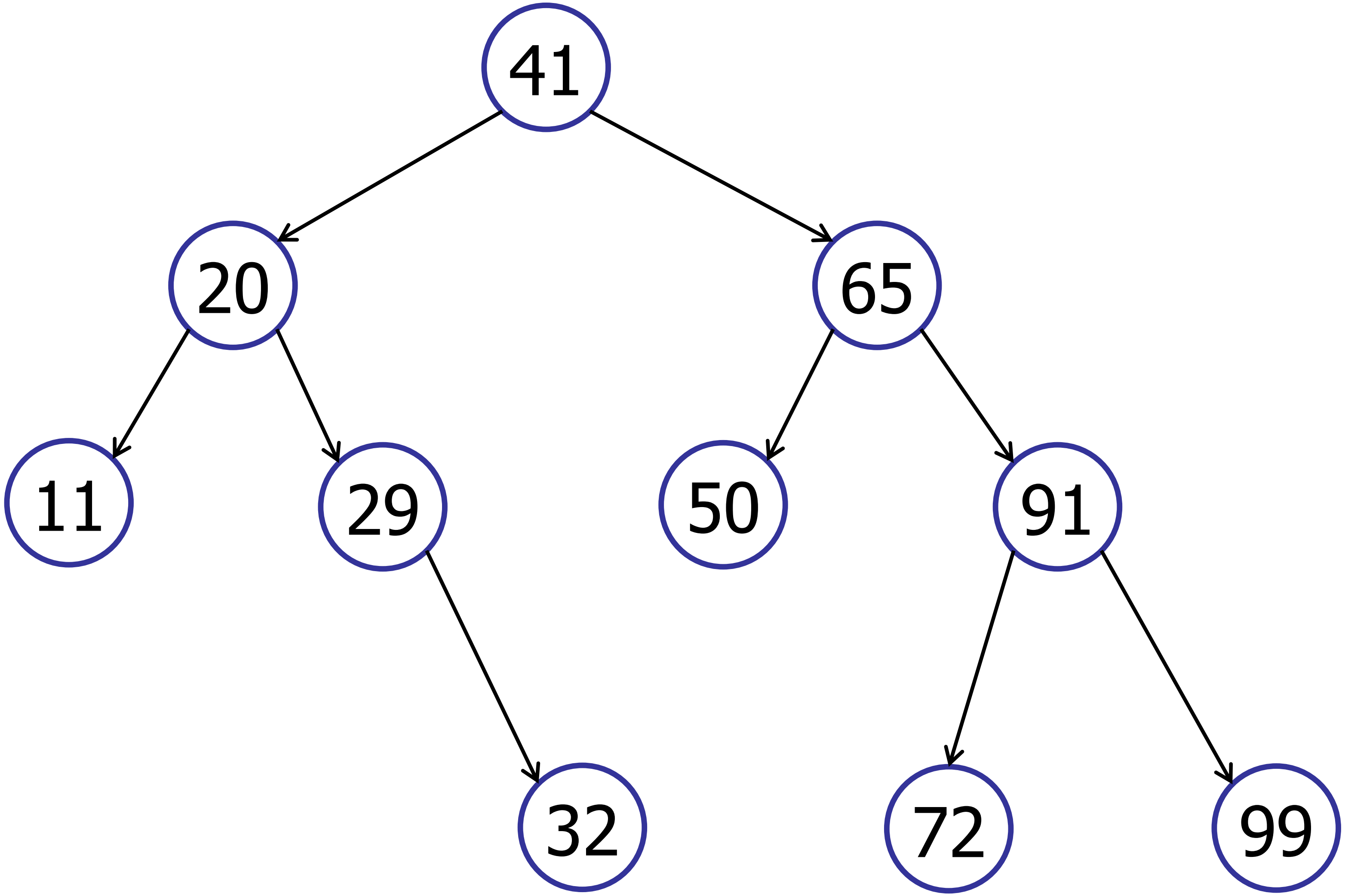
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

Binary Search Tree

delete(v)

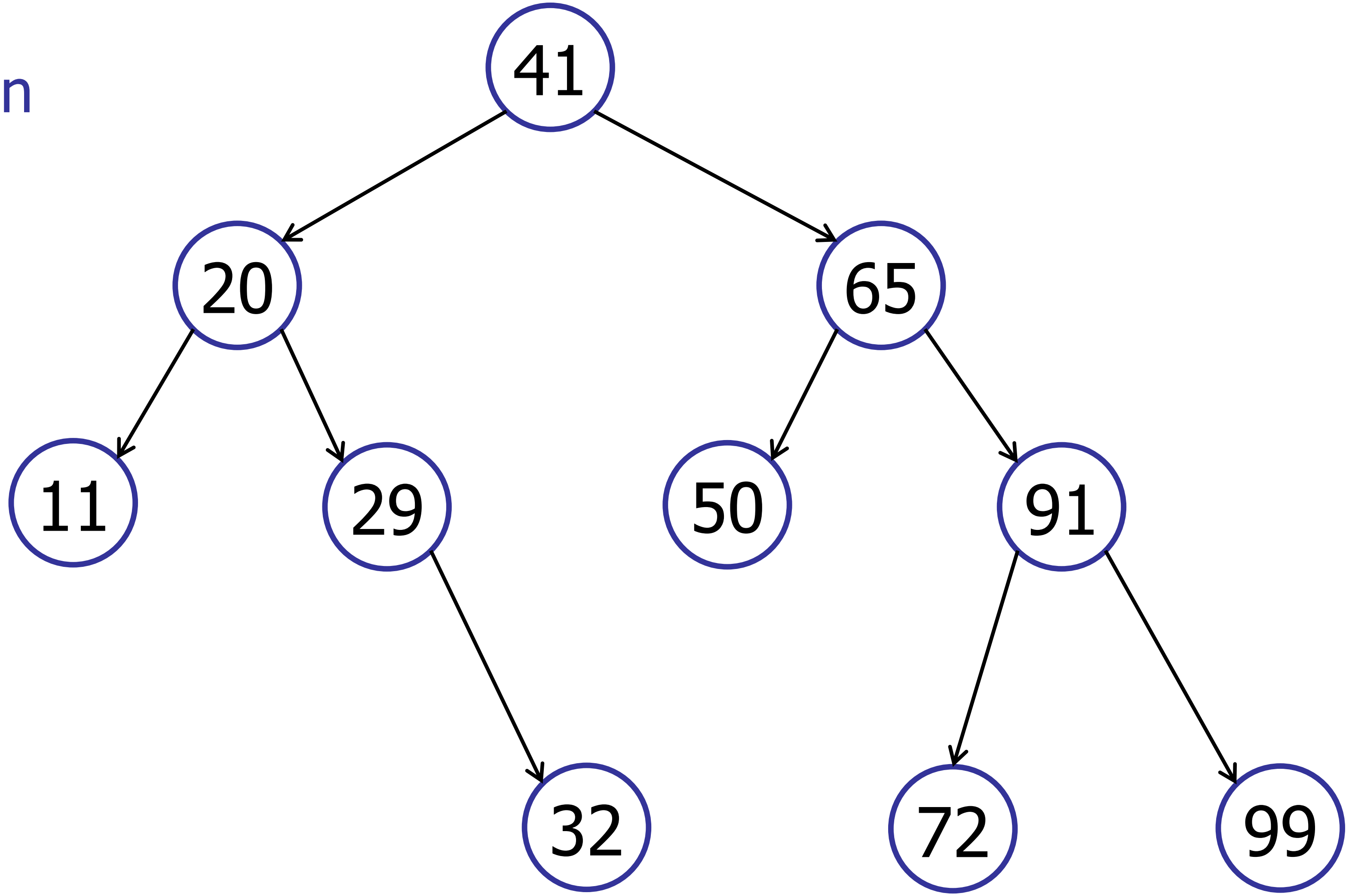


Binary Search Tree

delete(v)

Three cases:

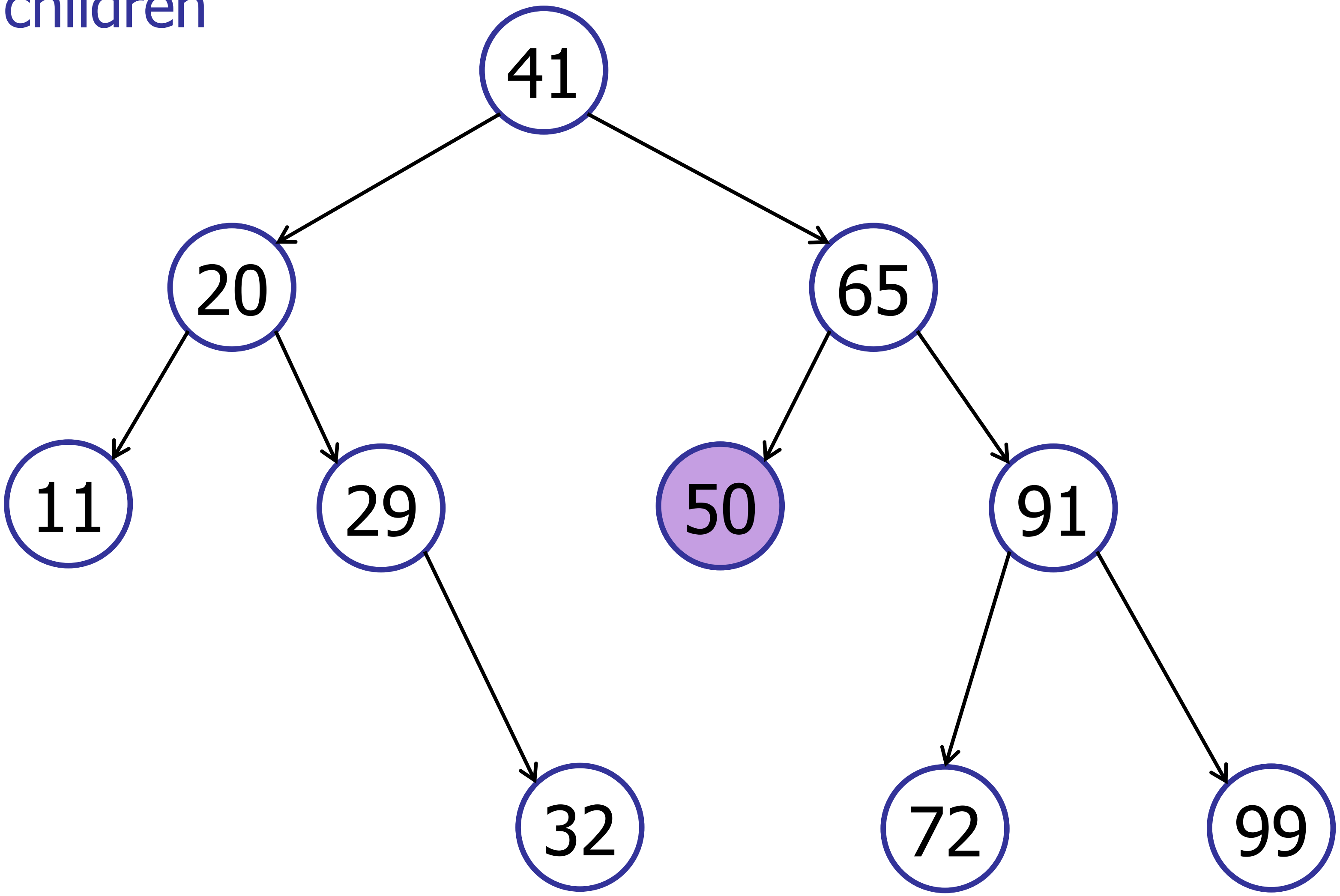
1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(50)

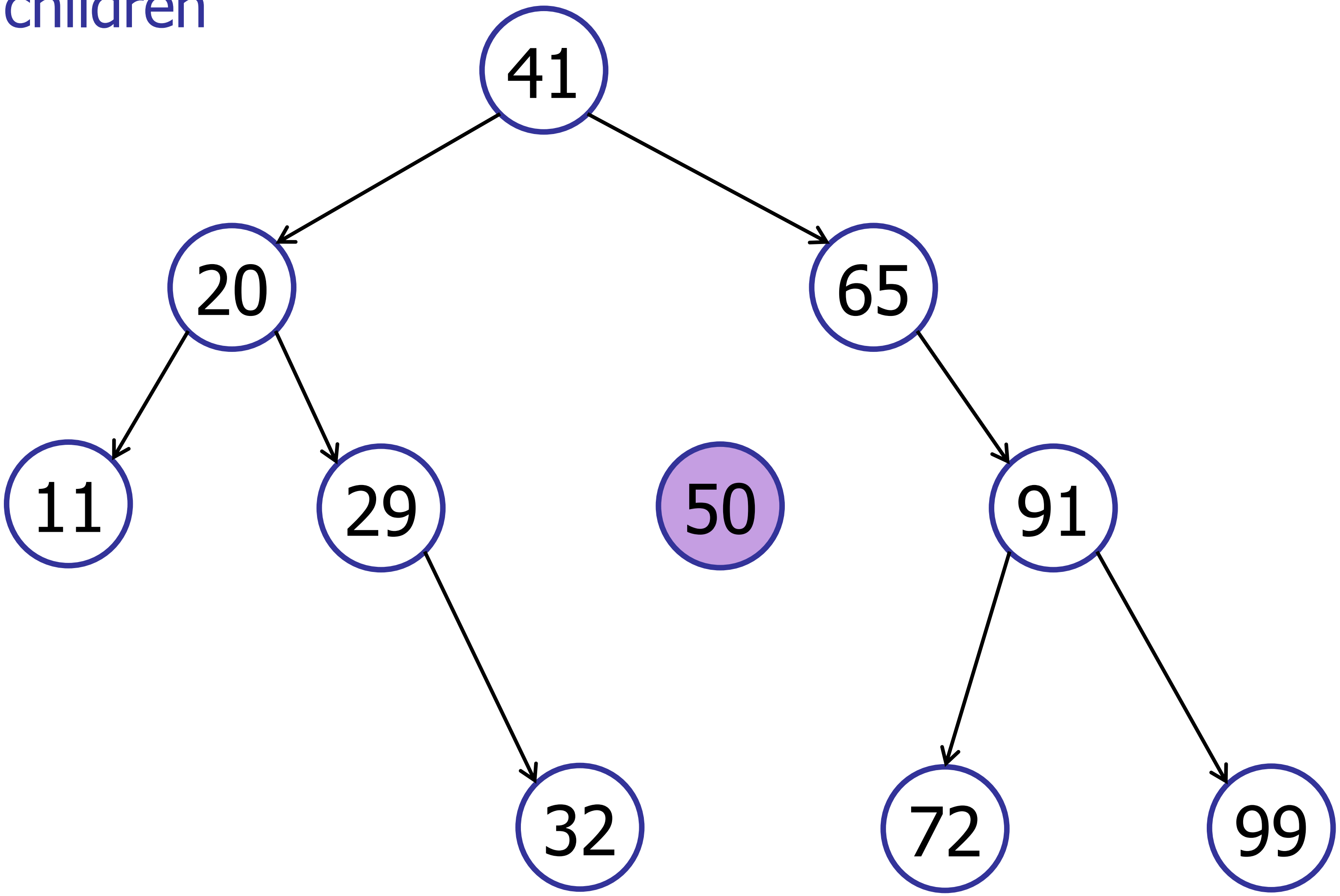
Case 1: No children



Binary Search Tree

delete(50)

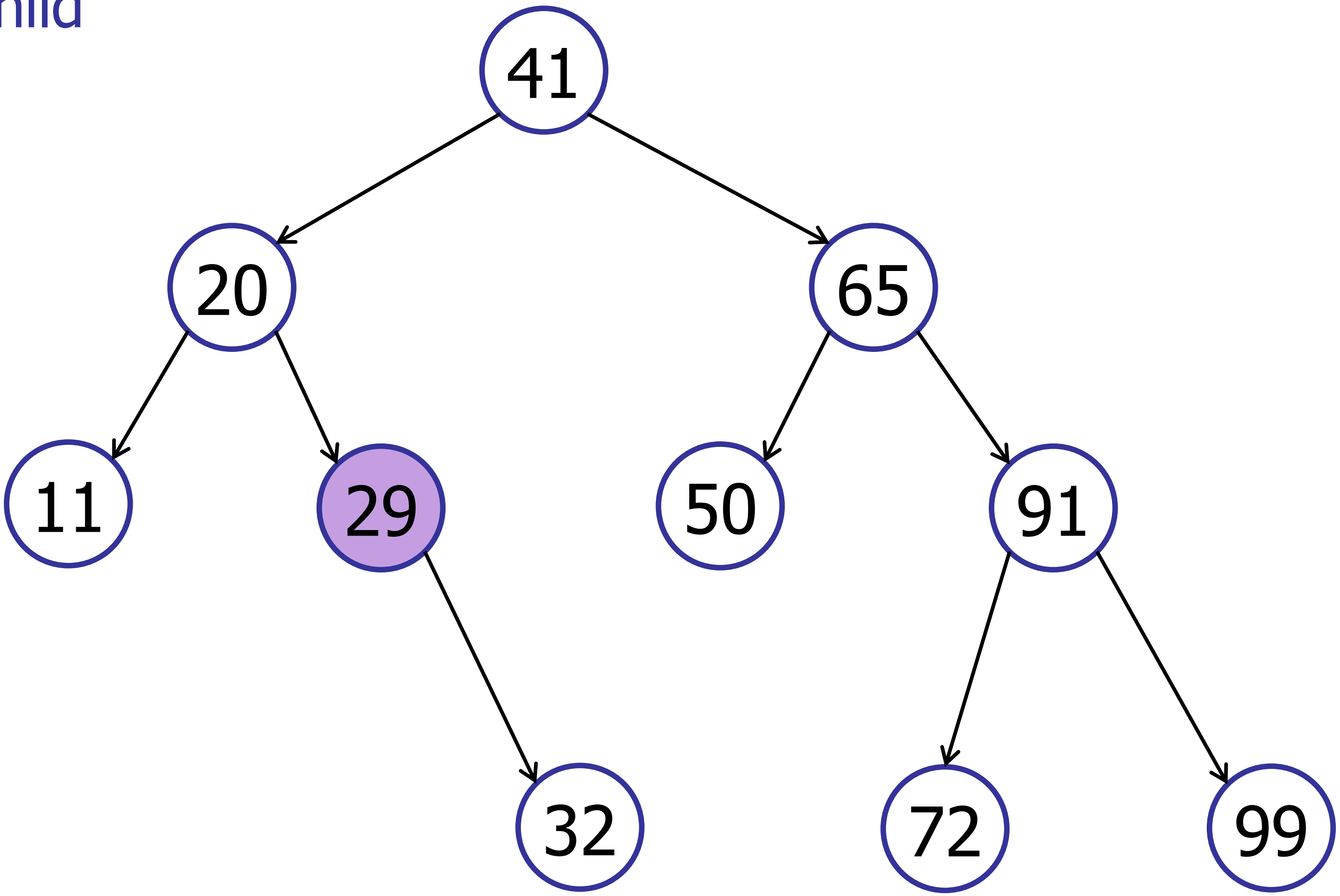
Case 1: No children



Binary Search Tree

delete(29)

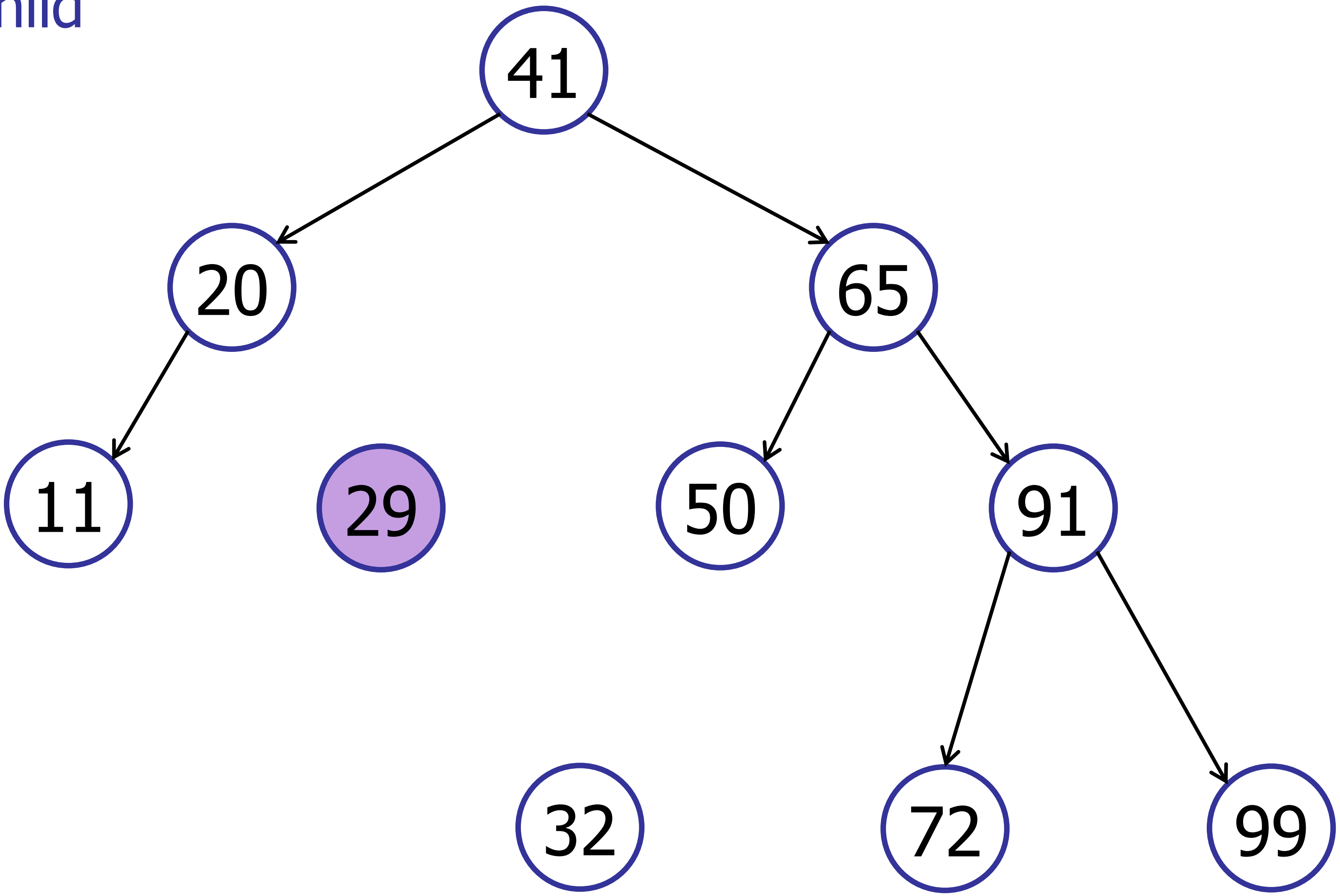
Case 2: 1 child



Binary Search Tree

delete(29)

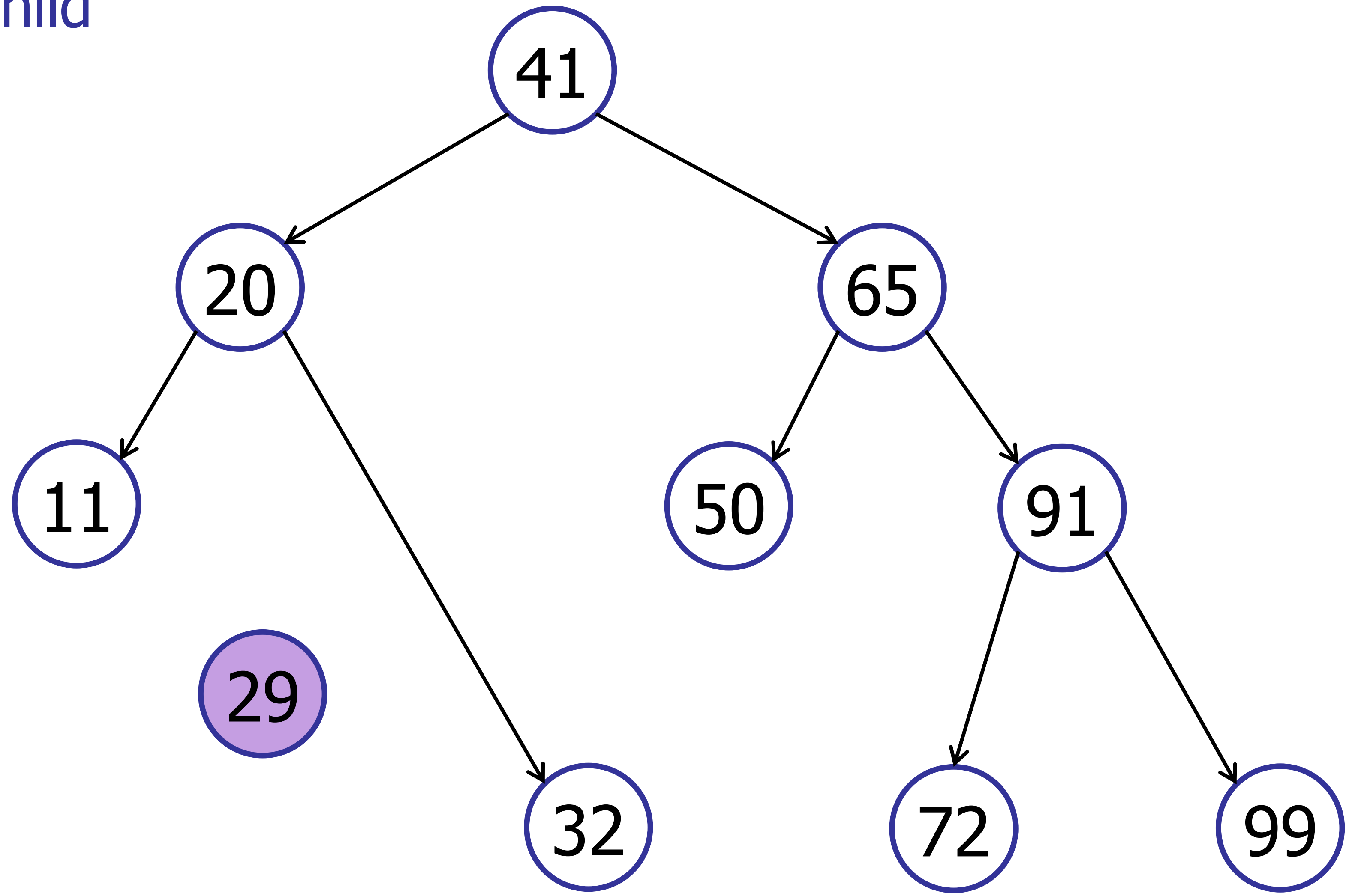
Case 2: 1 child



Binary Search Tree

delete(29)

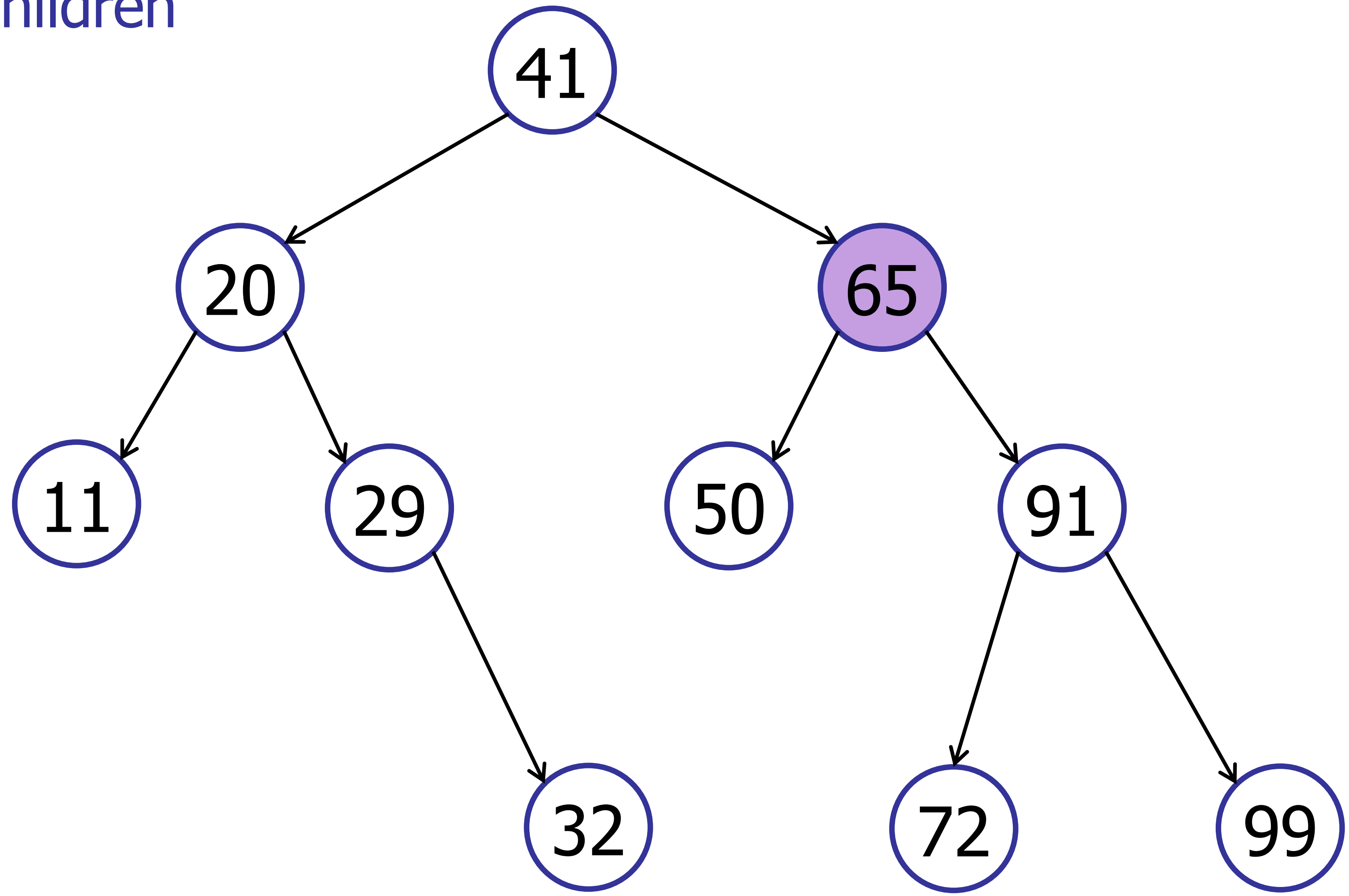
Case 2: 1 child



Binary Search Tree

delete(65)

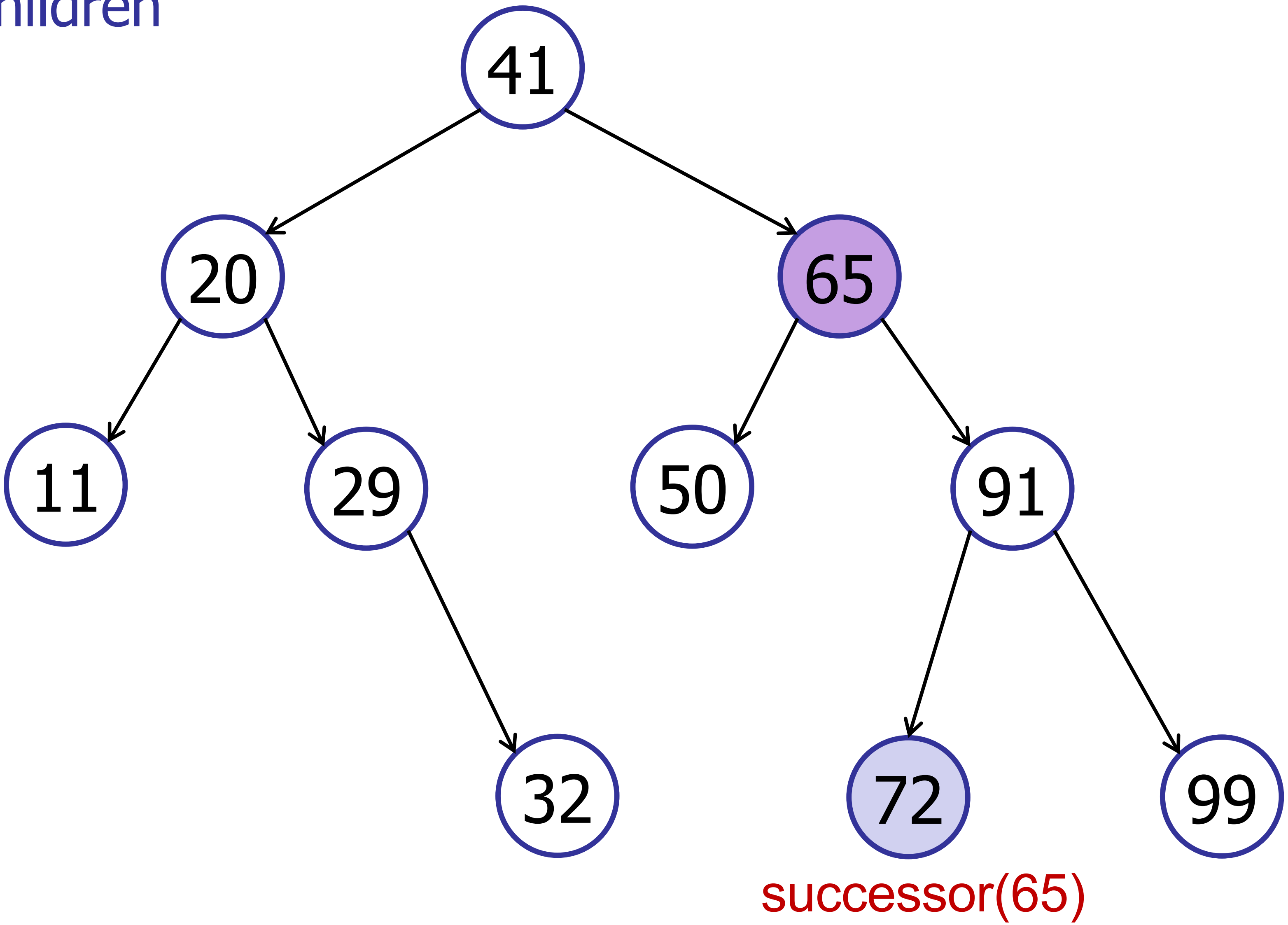
Case 3: 2 children



Binary Search Tree

delete(65)

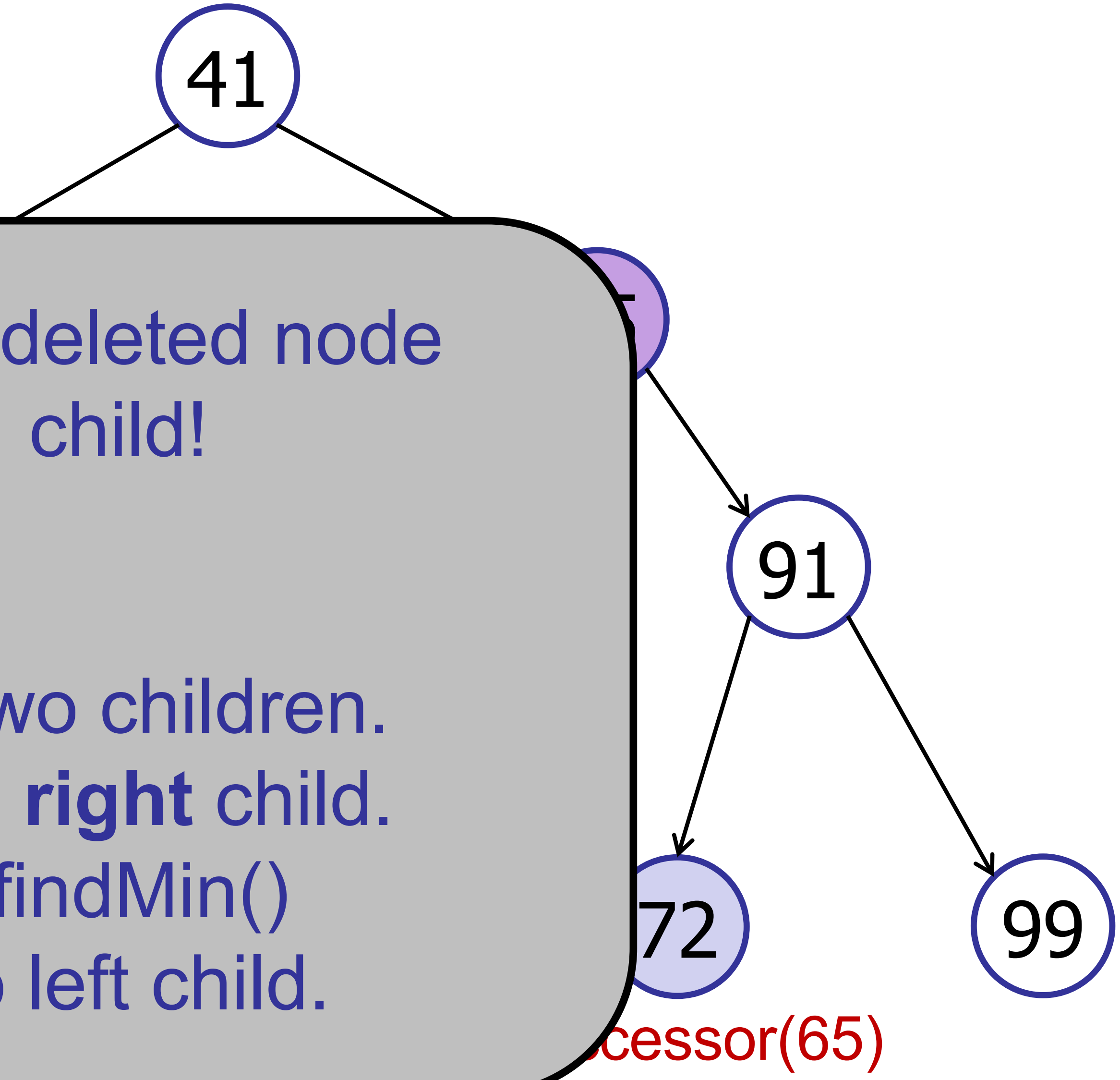
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



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

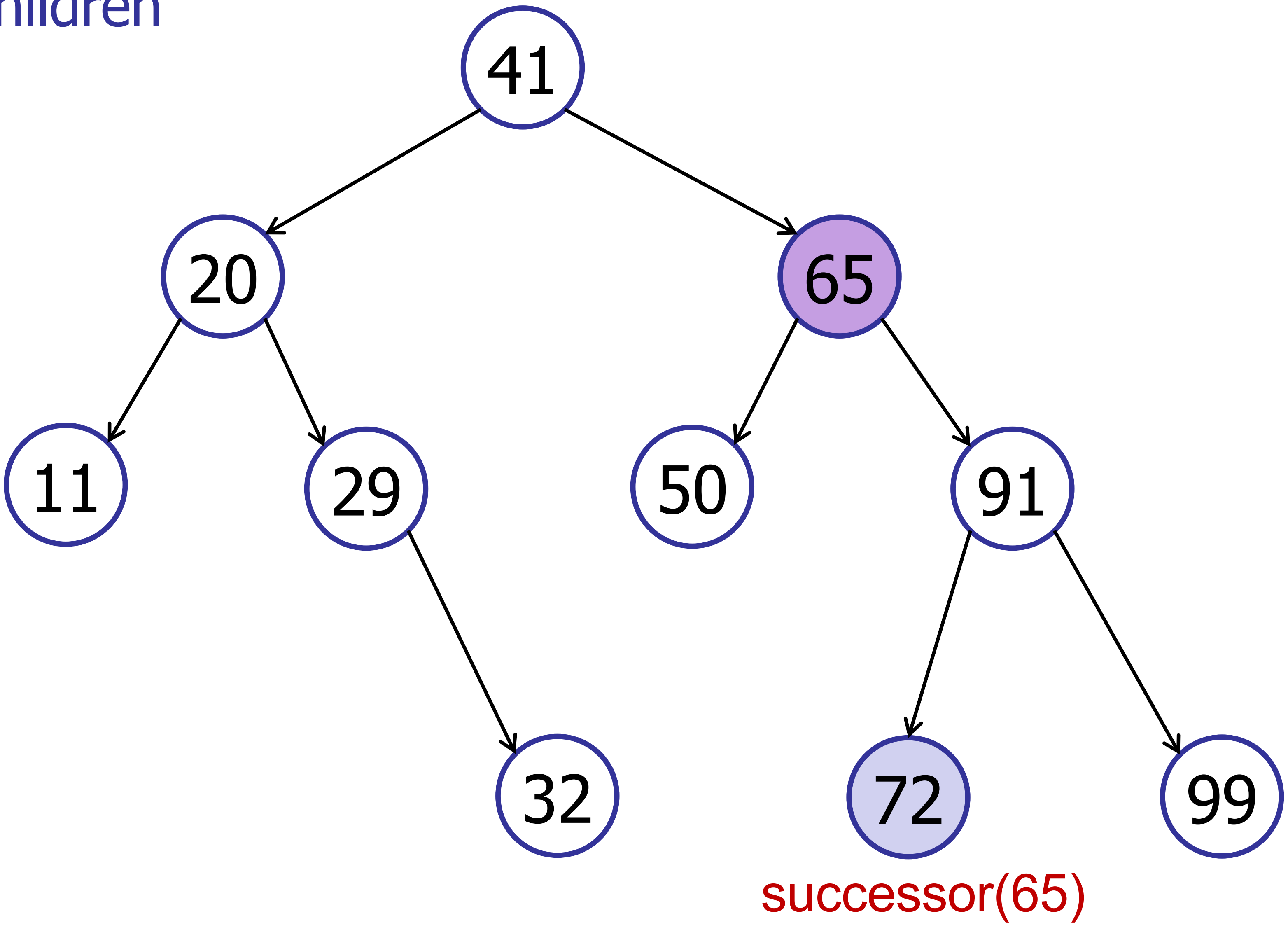
Proof:

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

Binary Search Tree

delete(65)

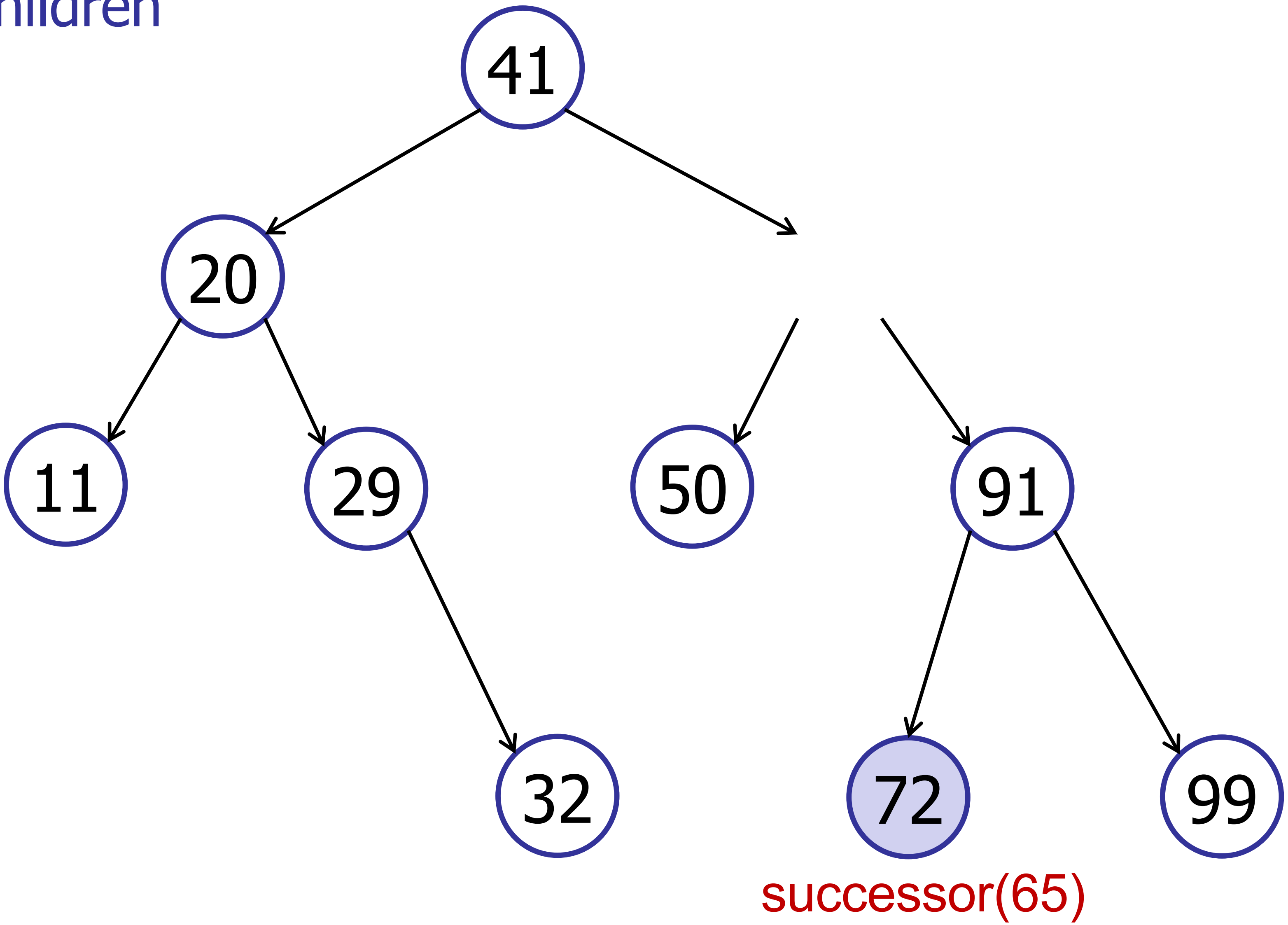
Case 3: 2 children



Binary Search Tree

delete(65)

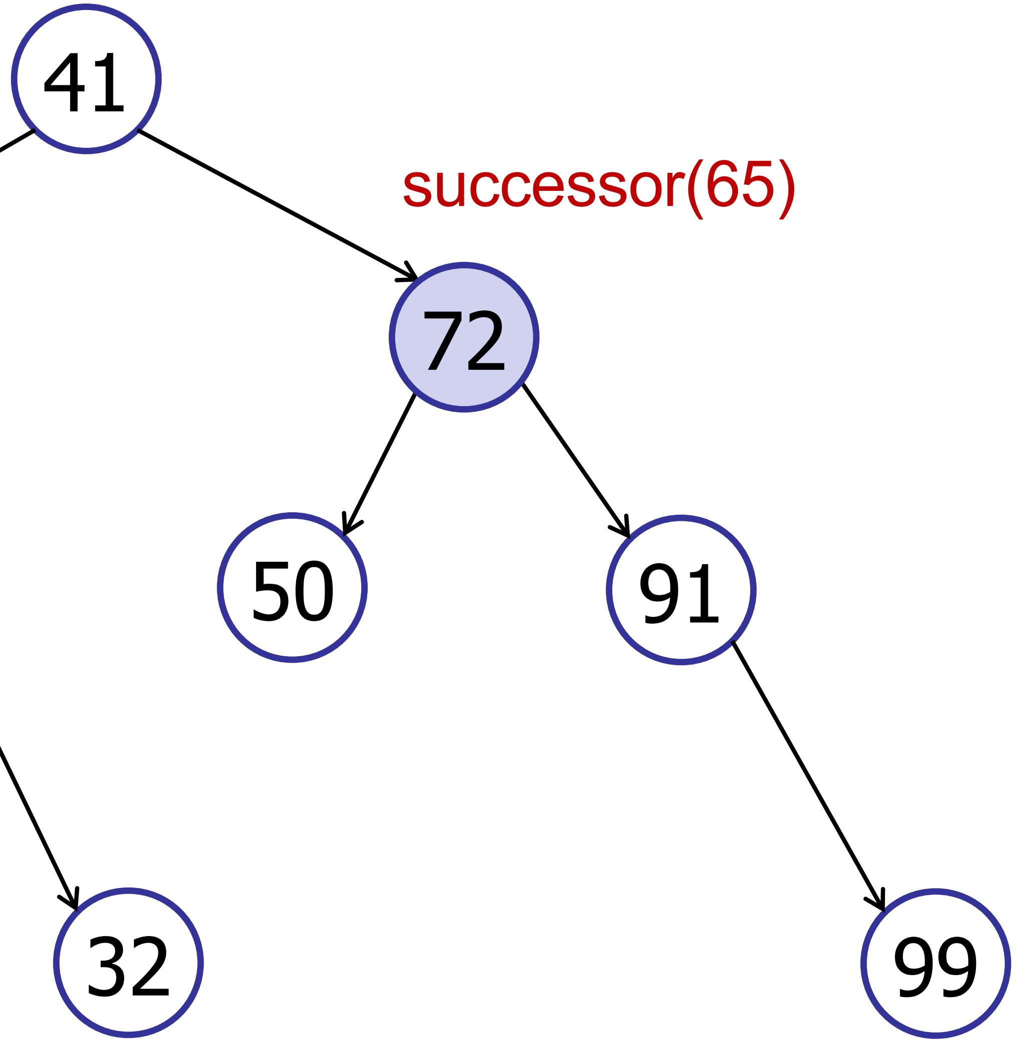
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



Check BST Property!

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