CS2020
Data Structures and Algorithms

Welcome!

Announcements

Quiz 1 : February 12

- In class: be there!
- Be on time.
- Covers material through today's lecture

Bring to quiz:

- One sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else.

Announcements

Problem Set 3.1415 is due Monday night...

Problem Set 4 released on Wednesday

– Due after recess week.

Plan of the Day

Trees

Dictionaries

Dictionary Interface

interface	**IDictionary<Key extends Comparable<Key>, Value>**	
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: add to middle of array --- O(n)
- search : binary search through array --- O(log n)

Option 2: Linked list

- insert: add to middle of array --- O(n)
- search : no binary search in array --- O(n)

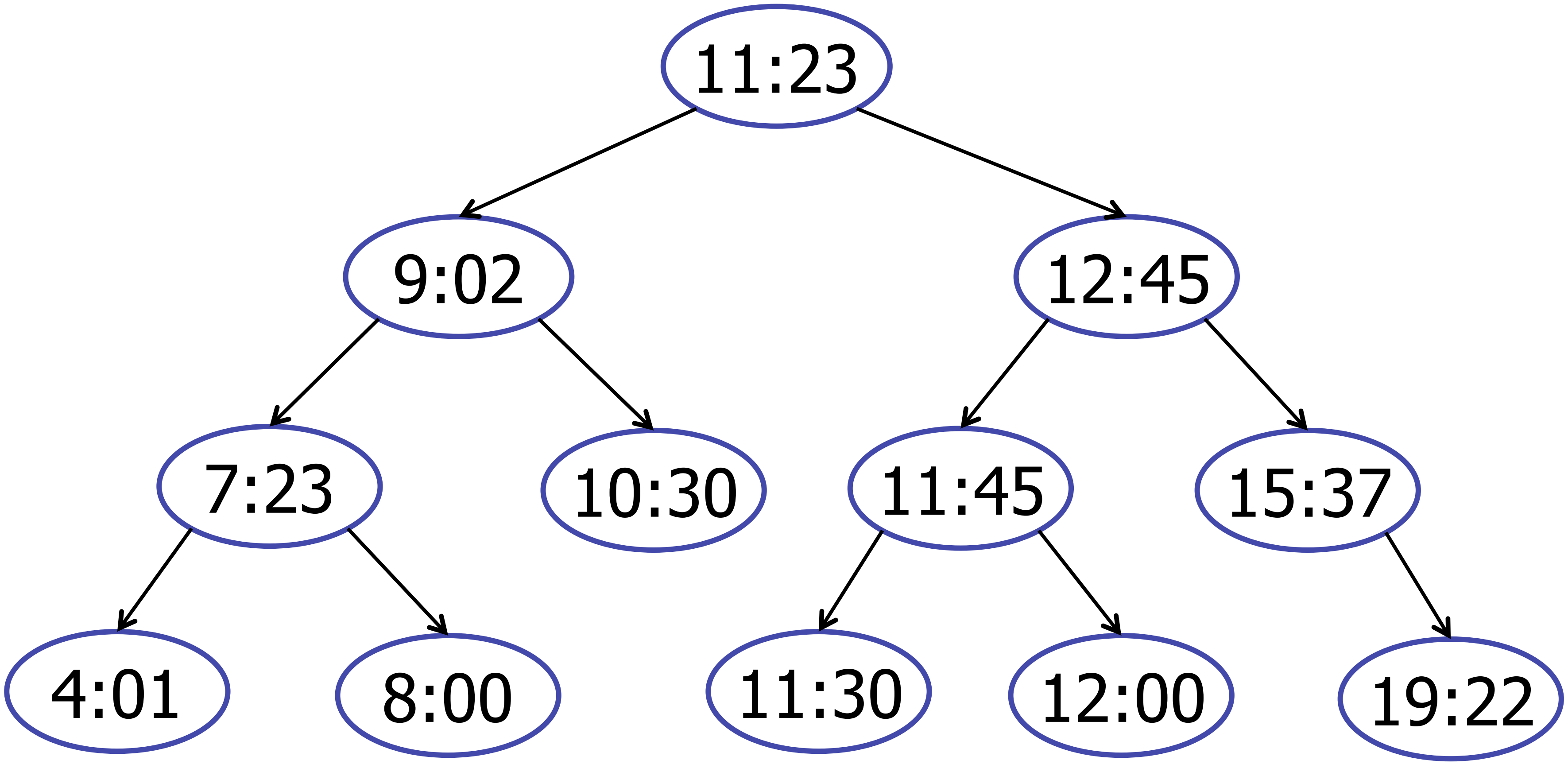
Dictionary Implementation

Possible Choices:

- Implement using an array (see: java.util.ArrayList).
- Implement using an array (see: java.util.Vector).
- Implement using a queue.
- Implement using a LinkedList
- …
- Implement using a tree.

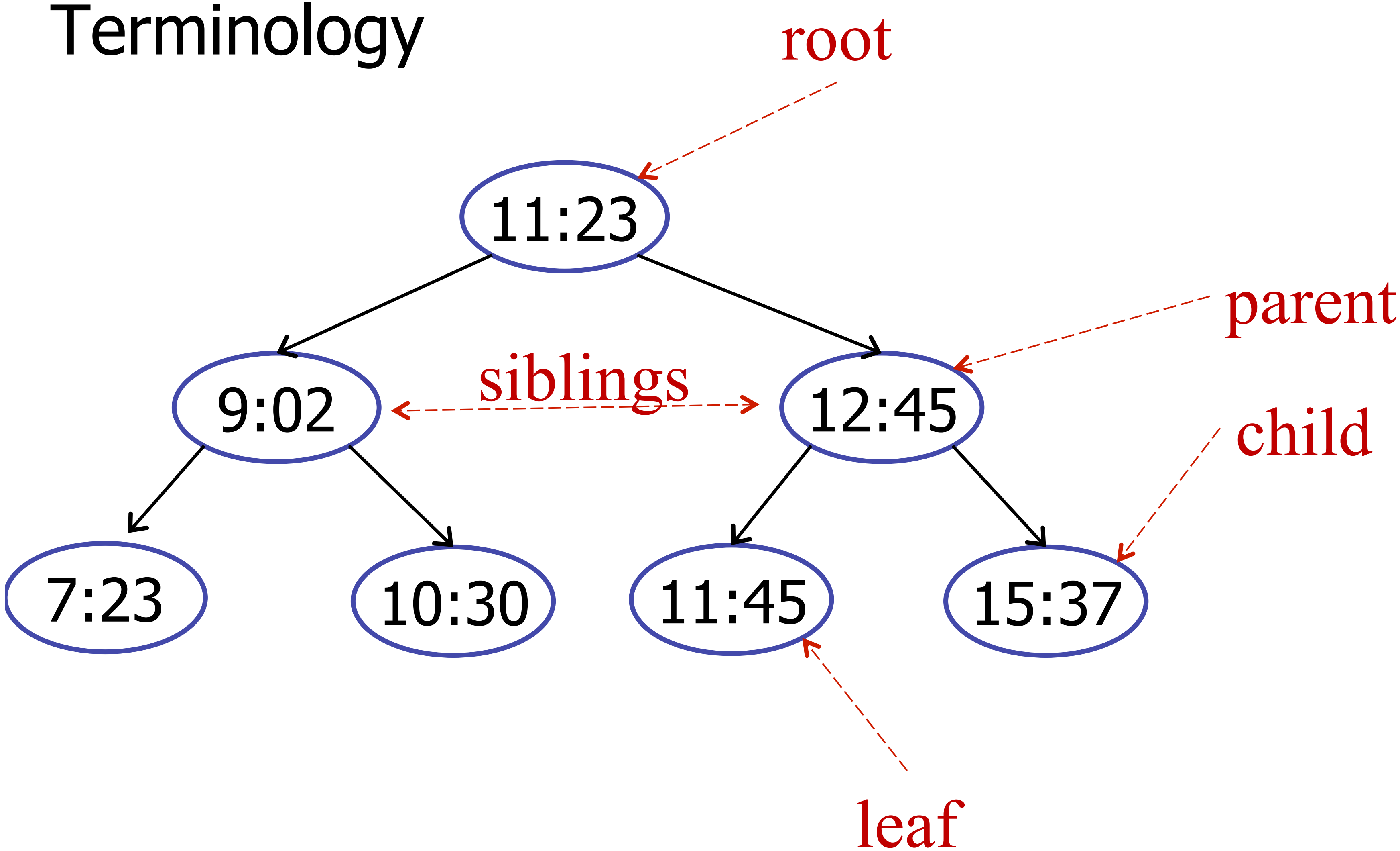
Dictionary

Implementation idea: Tree

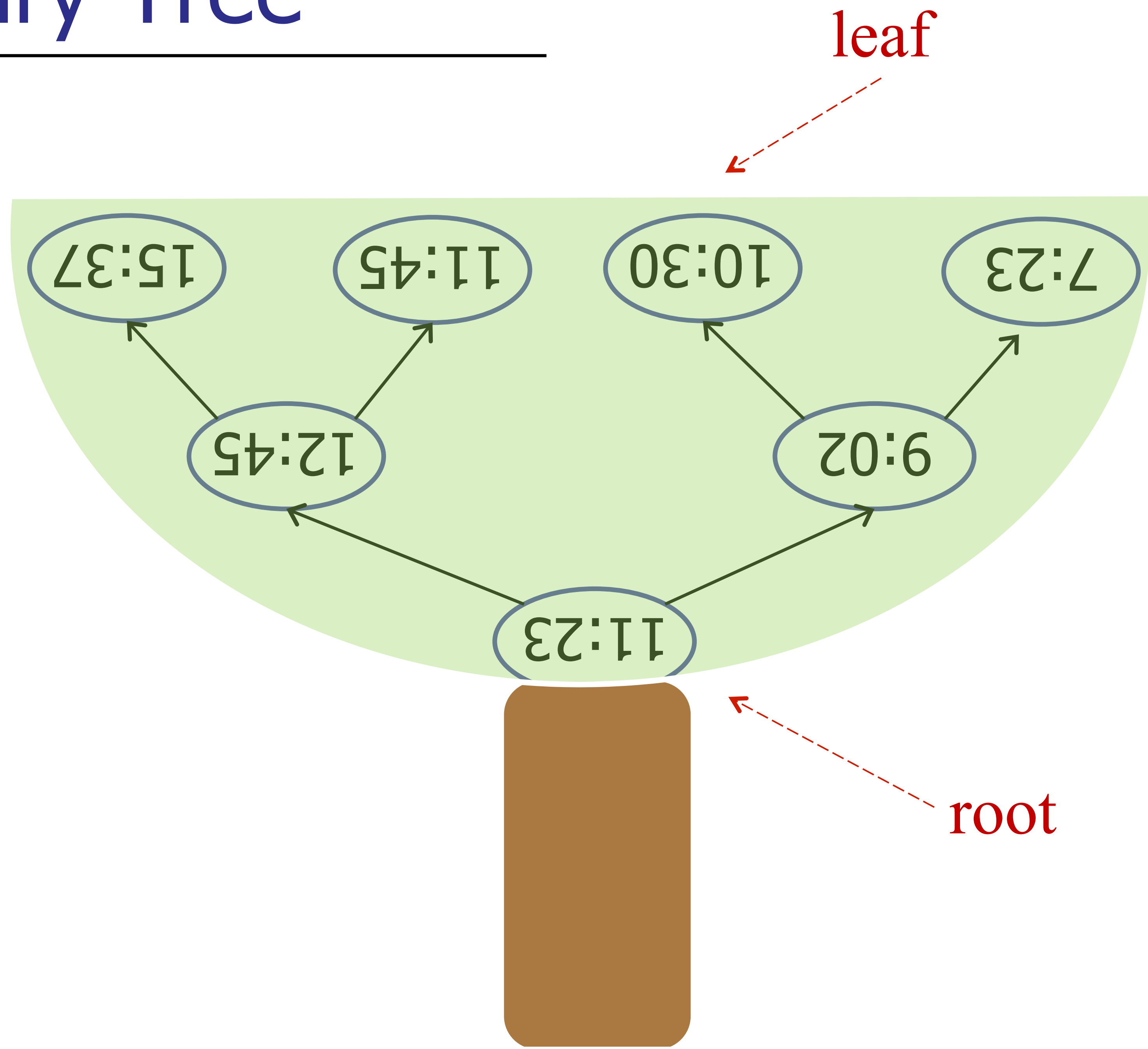


Binary Tree

Terminology

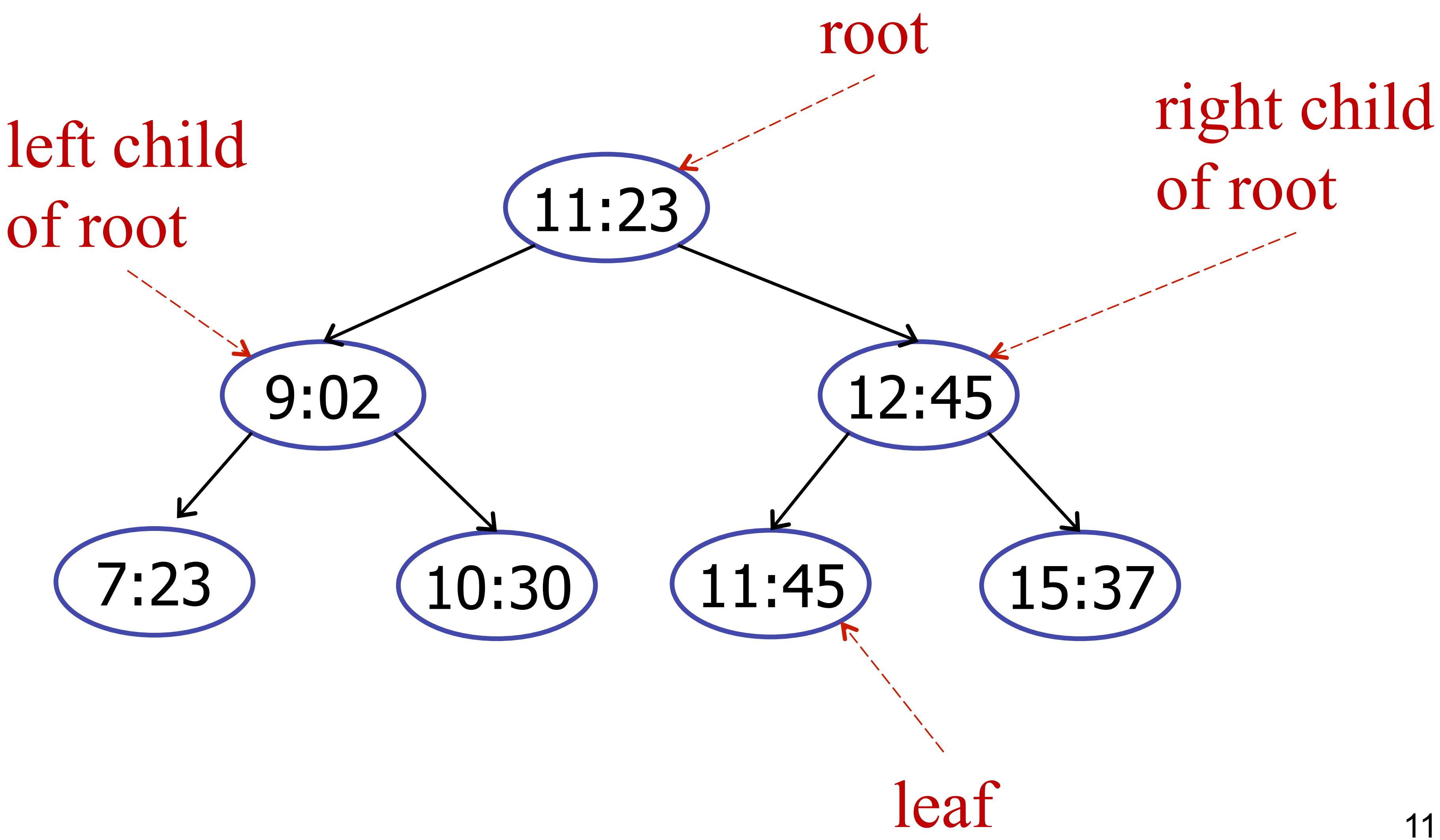


Binary Tree



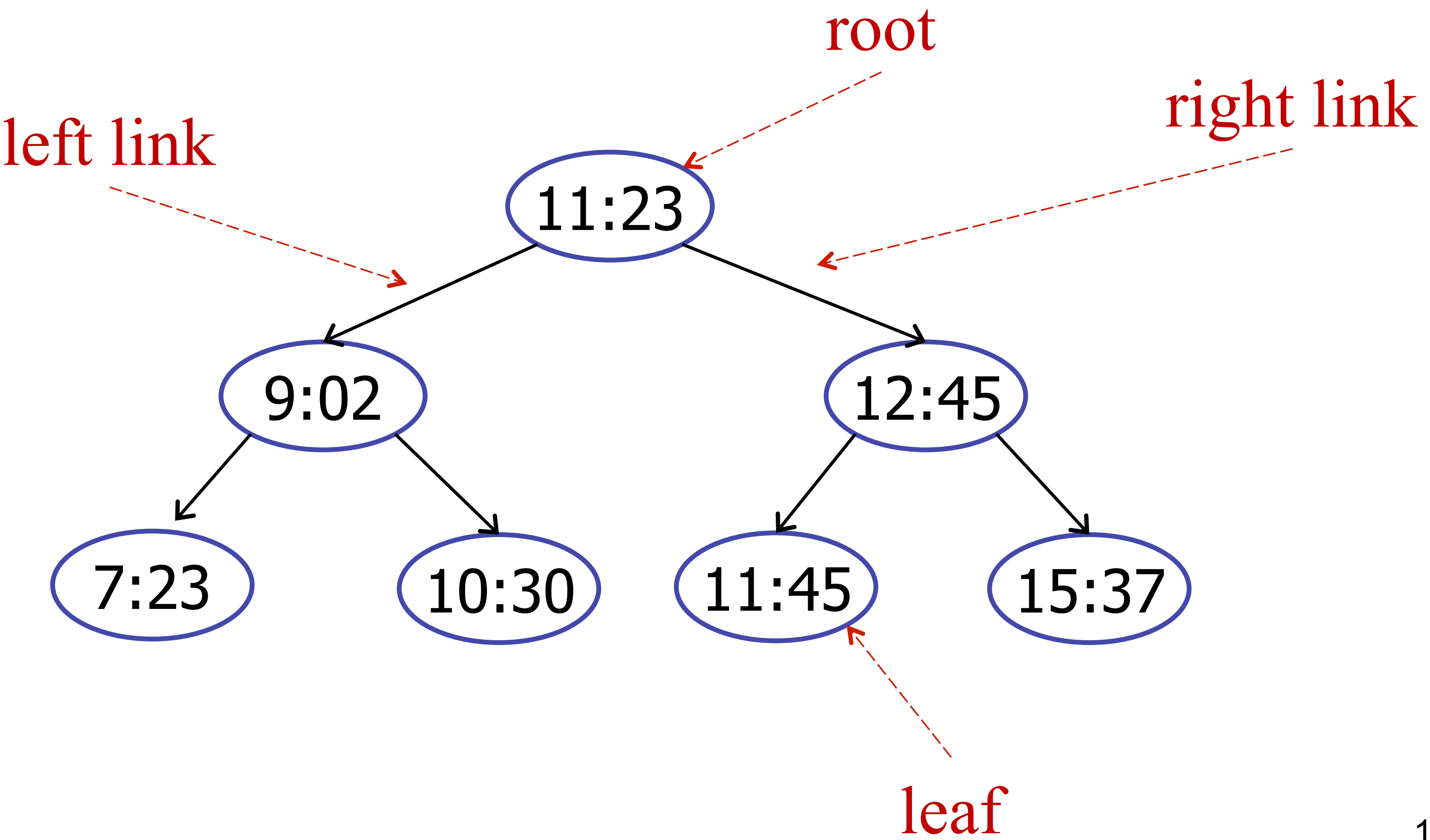
Binary Tree

Terminology



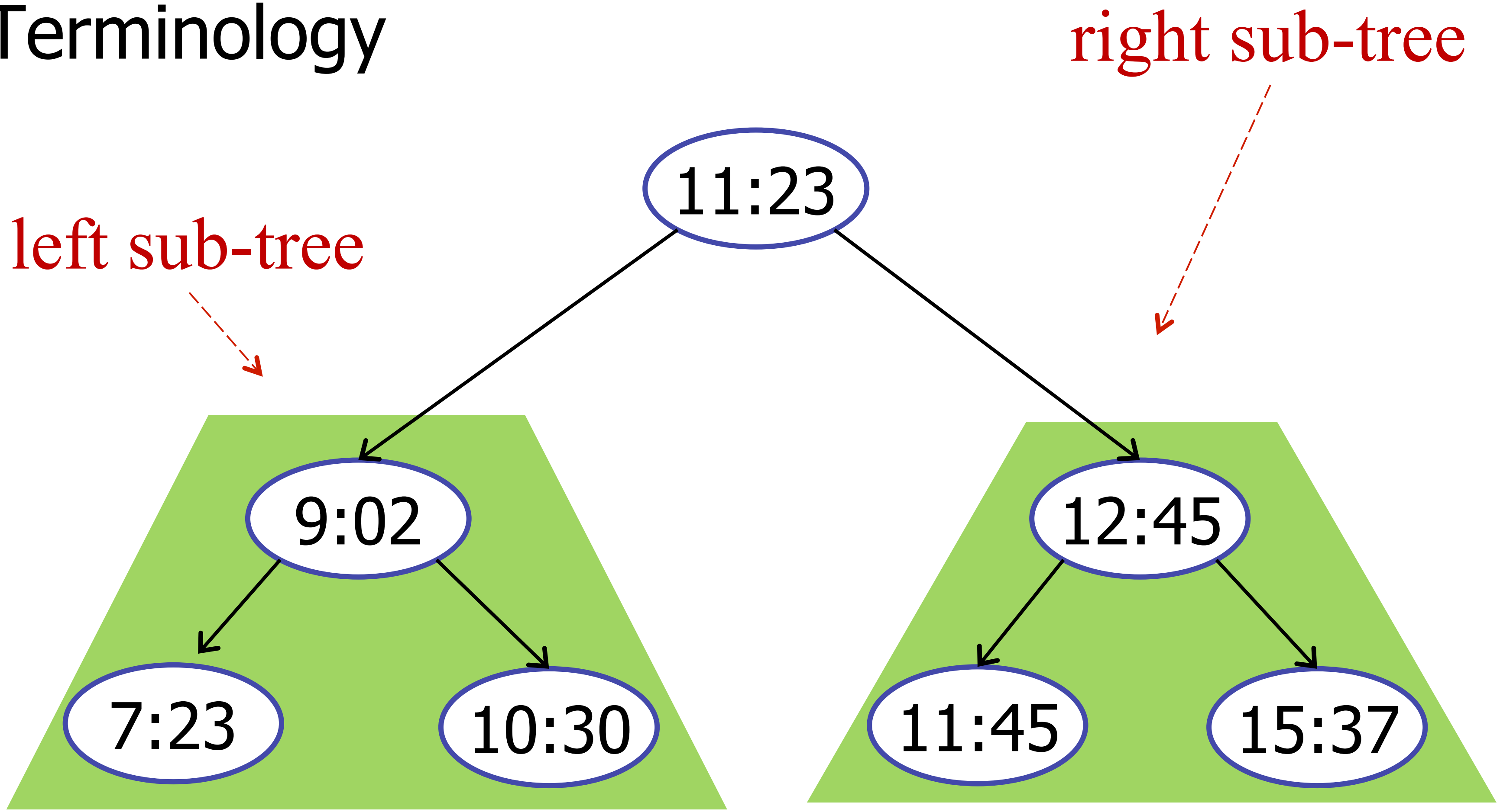
Binary Tree

Terminology



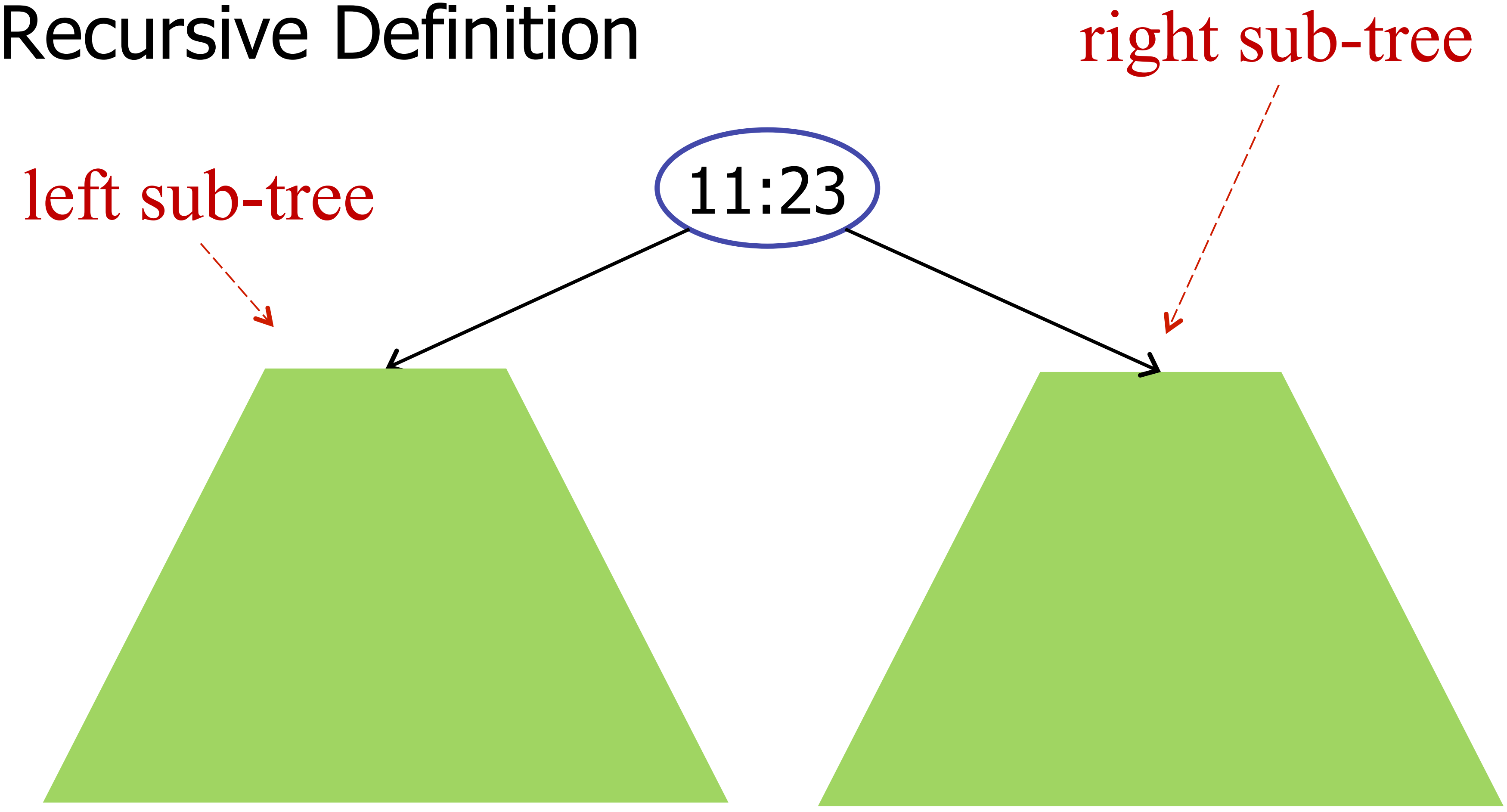
Binary Tree

Terminology



Binary Tree

Recursive Definition



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

Binary Tree

Java Definition

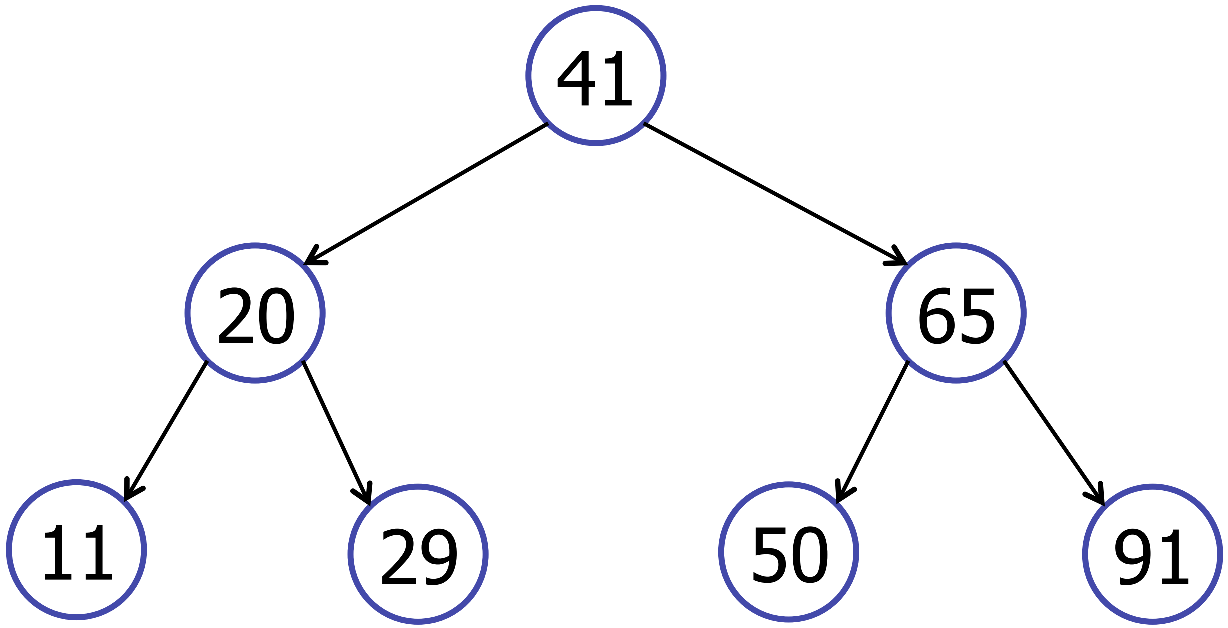
```
public class BinaryTree<Key extends Comparable<Key>, Value> {

      private BinaryTree<Key, Value> m_leftTree;
      private BinaryTree<Key, Value> m_rightTree;

      private Key m_key;
      private Value m_value;

      // Remainder of binary tree implementation
}
```

Binary **Search** Trees (BST)

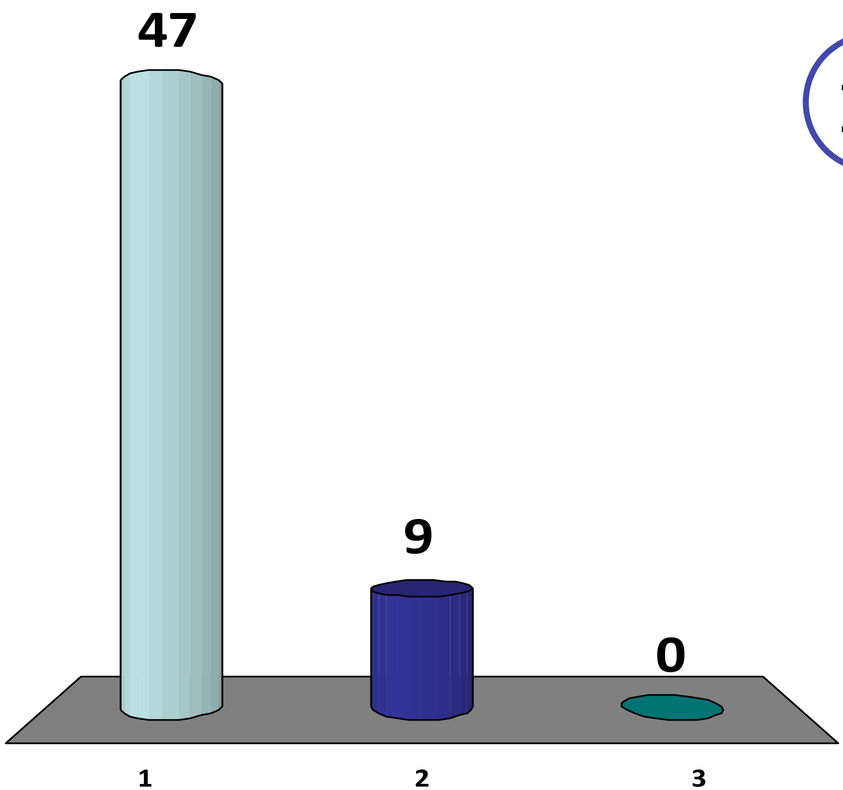


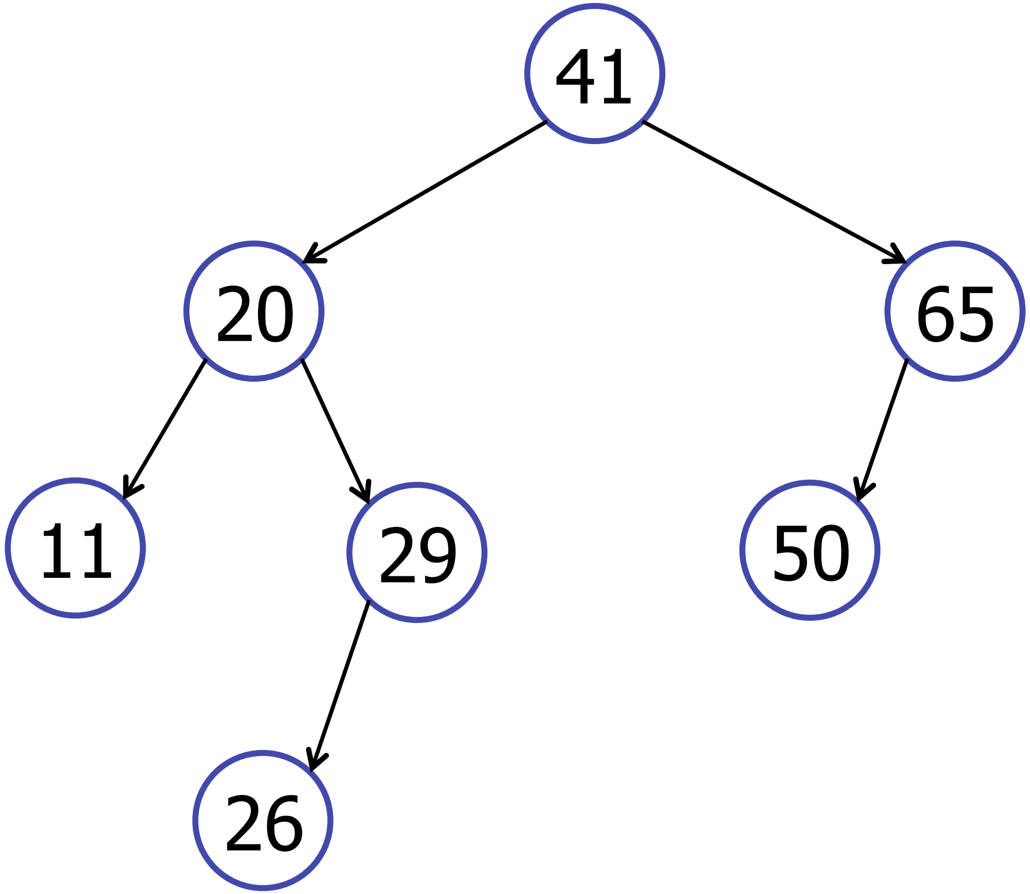
BST Property:

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

Is this a binary search tree?

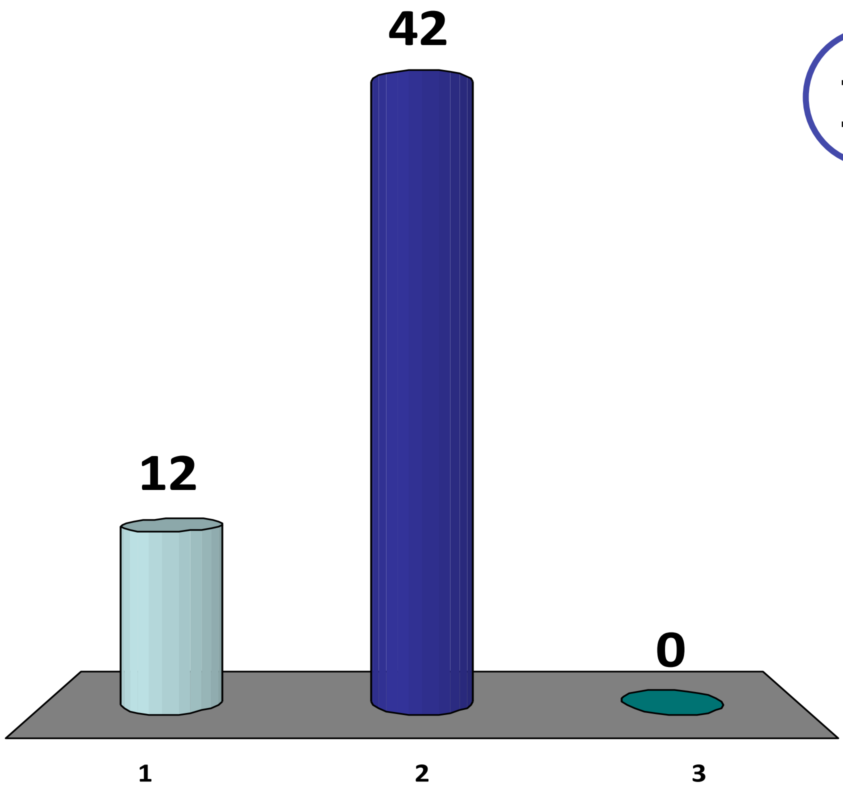
1. Yes
2. No
3. I don't know.

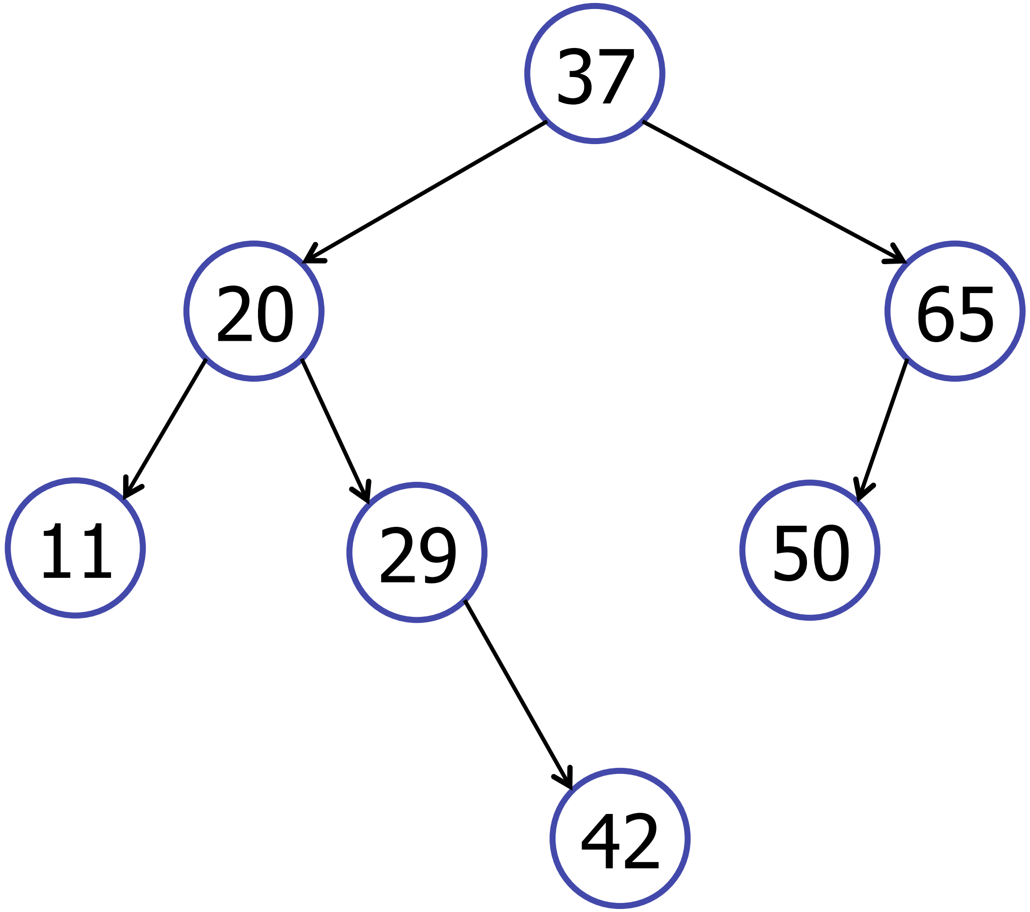




Is this a binary search tree?

1. Yes
2. No ✓
3. I don't know.





Binary Search Trees

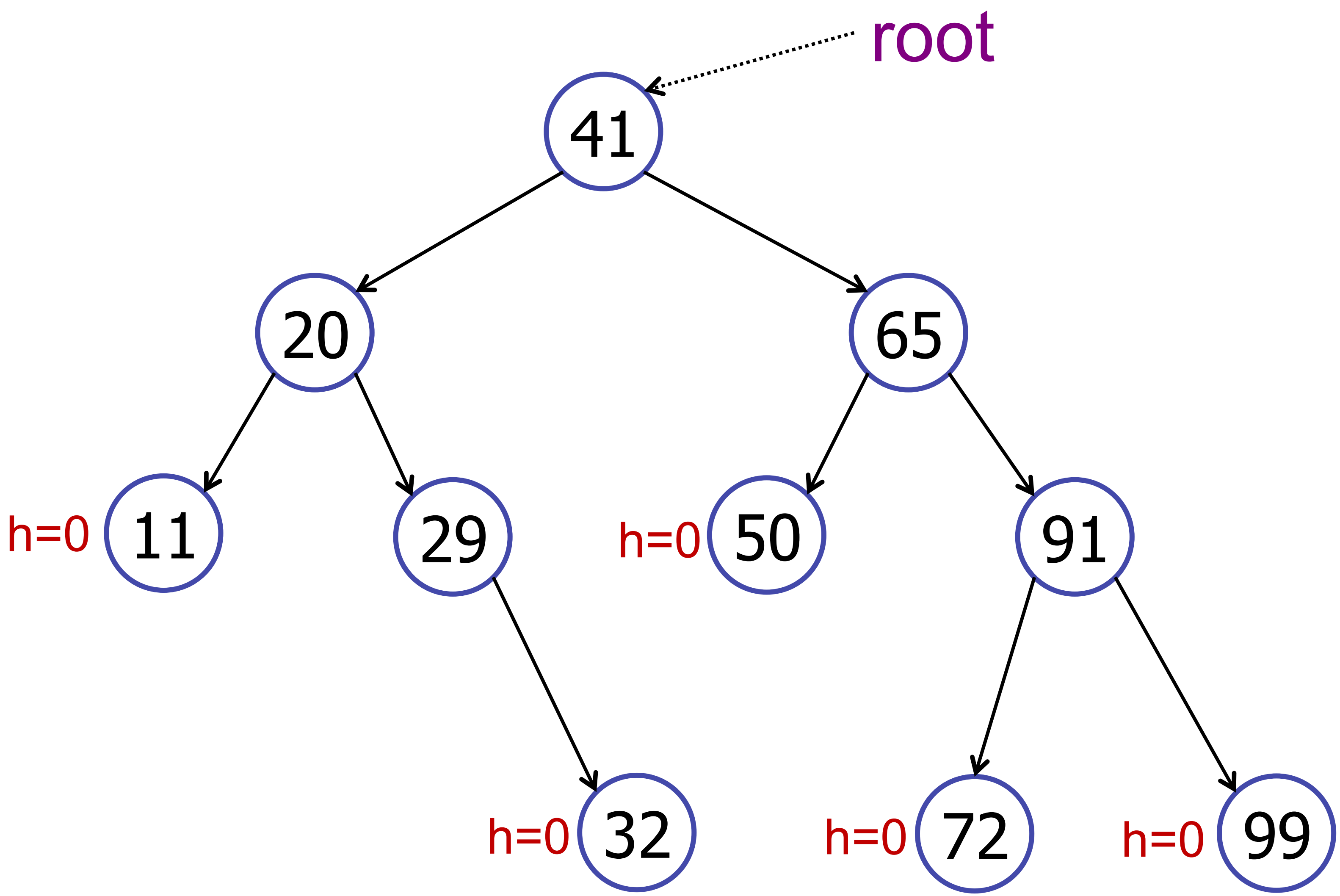
1. Terminology and Definitions

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

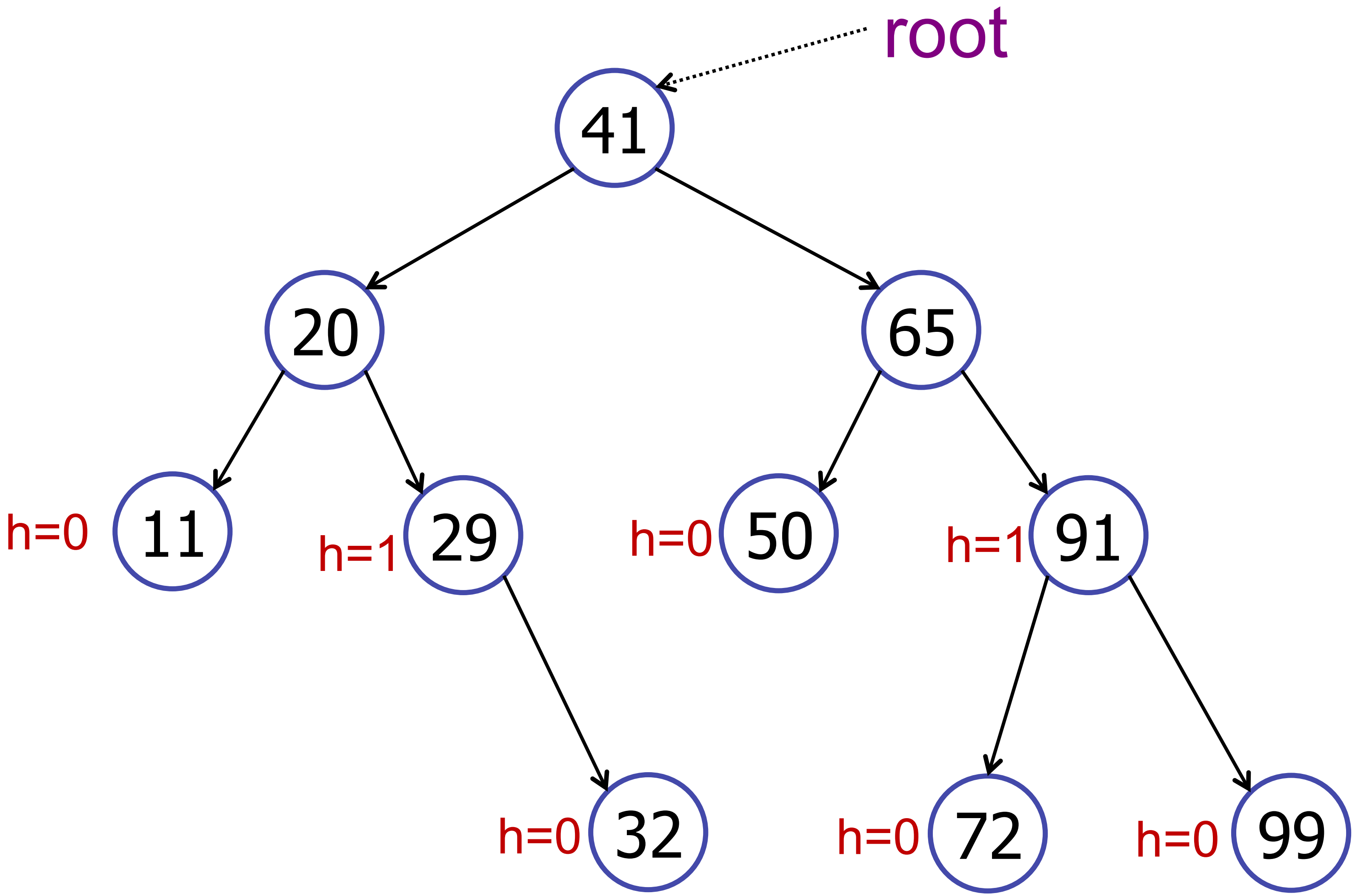
3. Traversals
   - in-order, pre-order, post-order

4. Other operations

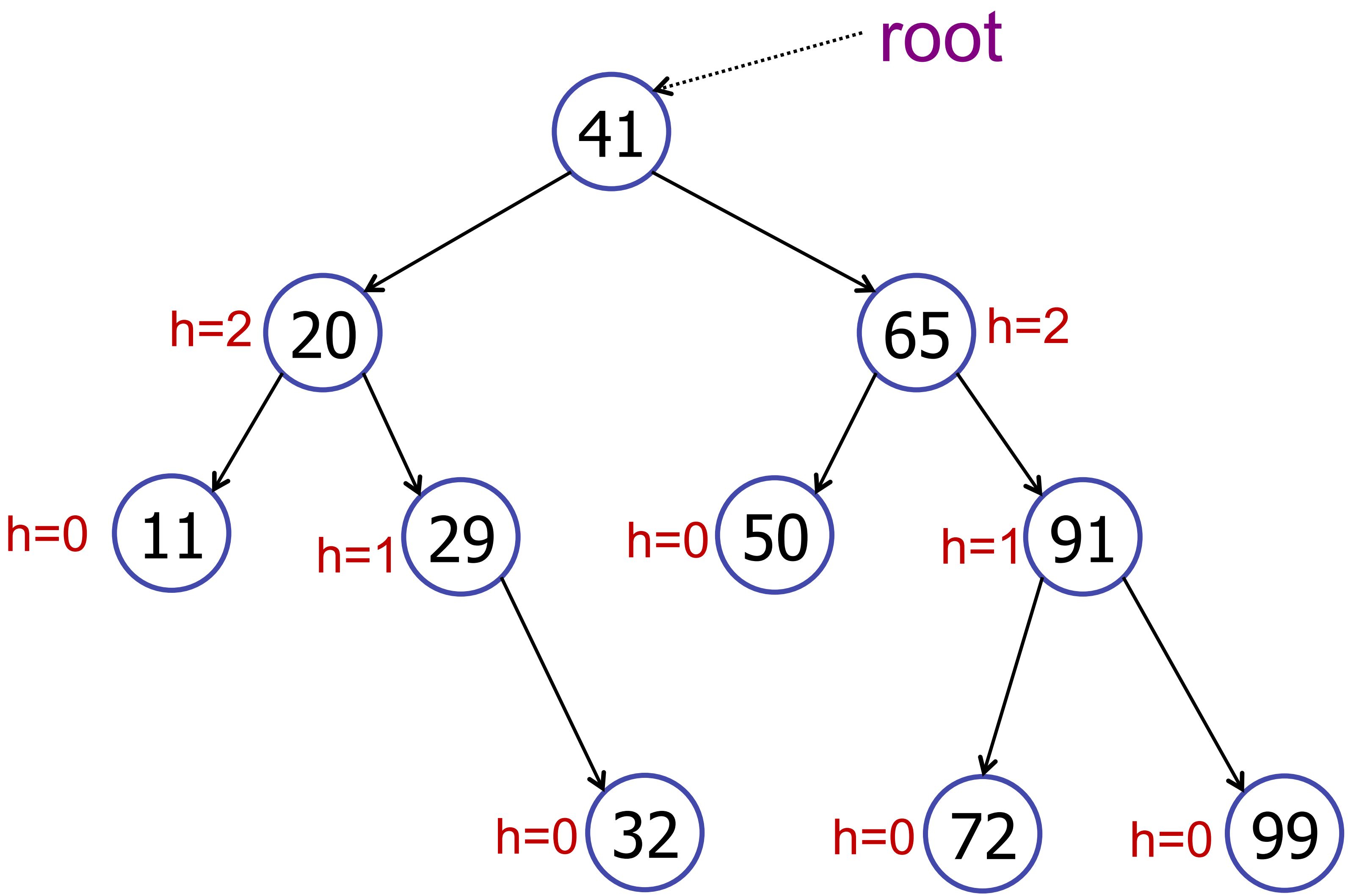
Height of a Binary Tree



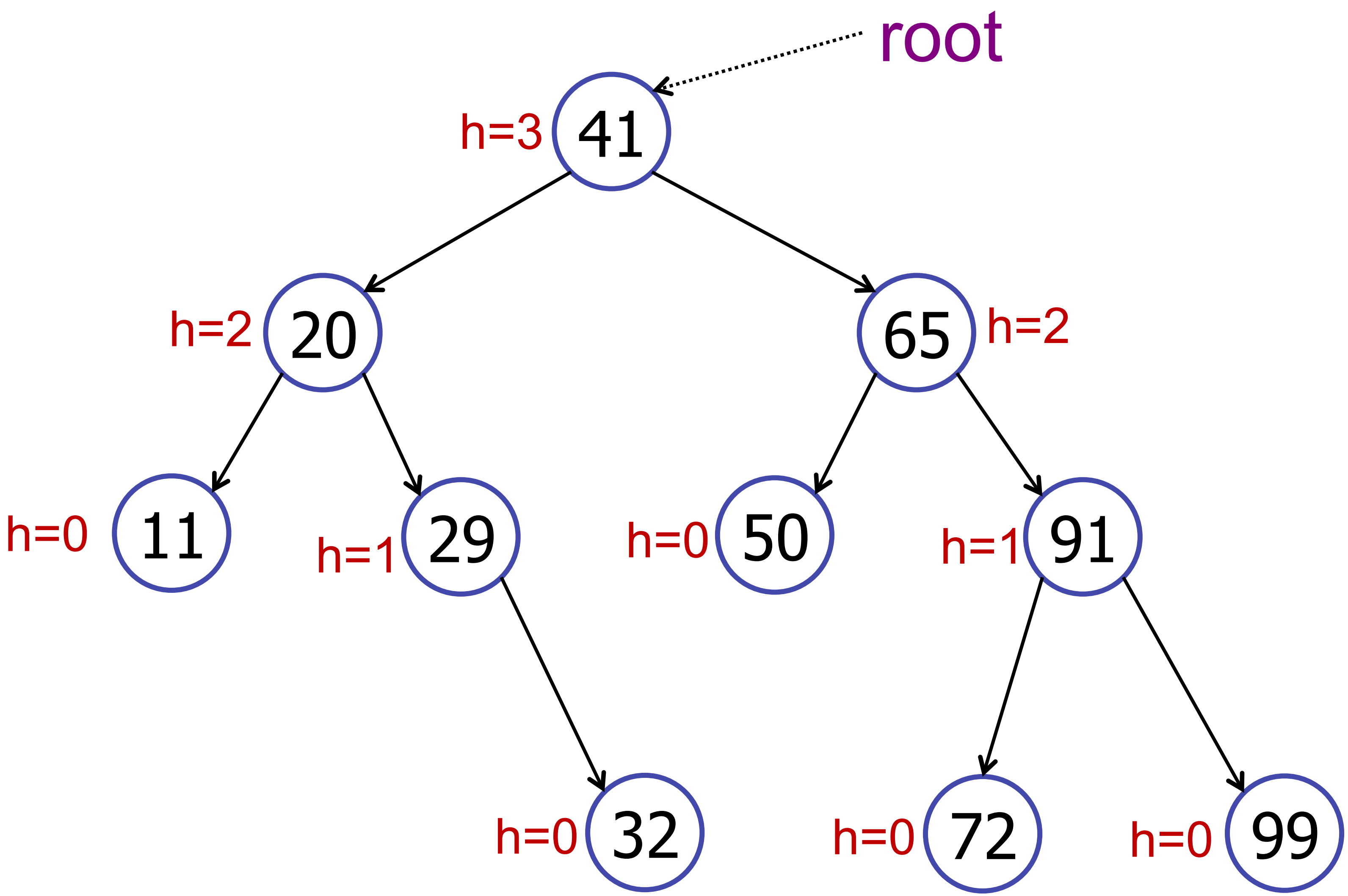
Height of a Binary Tree



Height of a Binary Tree



Height of a Binary Tree



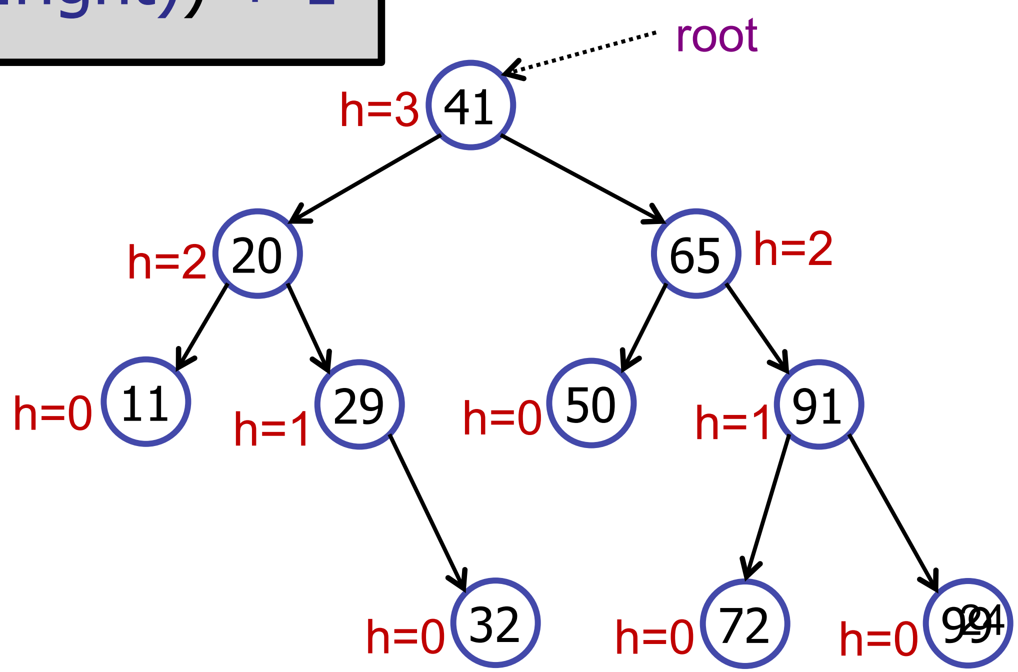
Height of a Binary Tree

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

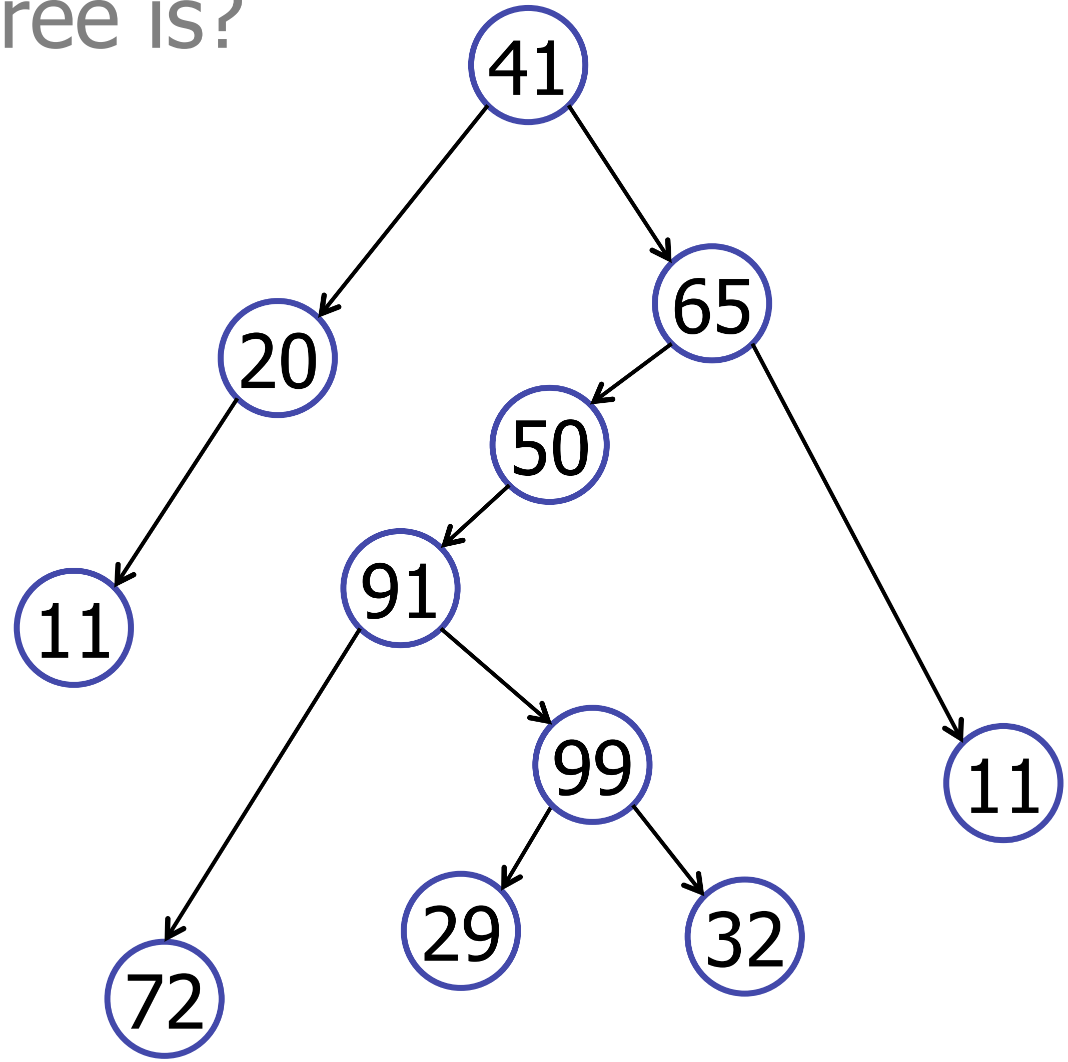
```
public int height(){
        int leftHeight = -1;
        int rightHeight = -1;
        if (m_leftTree != null)
              leftHeight = m_leftTree.height();
        if (m_rightTree != null)
              rightHeight = m_rightTree.height();
        return max(leftHeight, rightHeight) +  1;
}
```

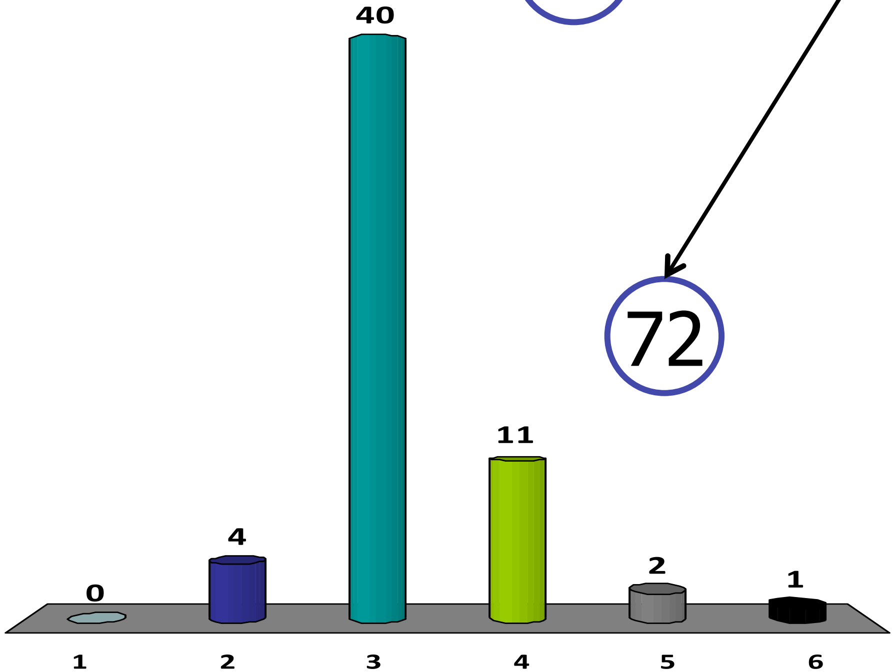
max of subtrees

add 1

The height of this tree is?

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





Binary Search Trees

1. Terminology and Definitions

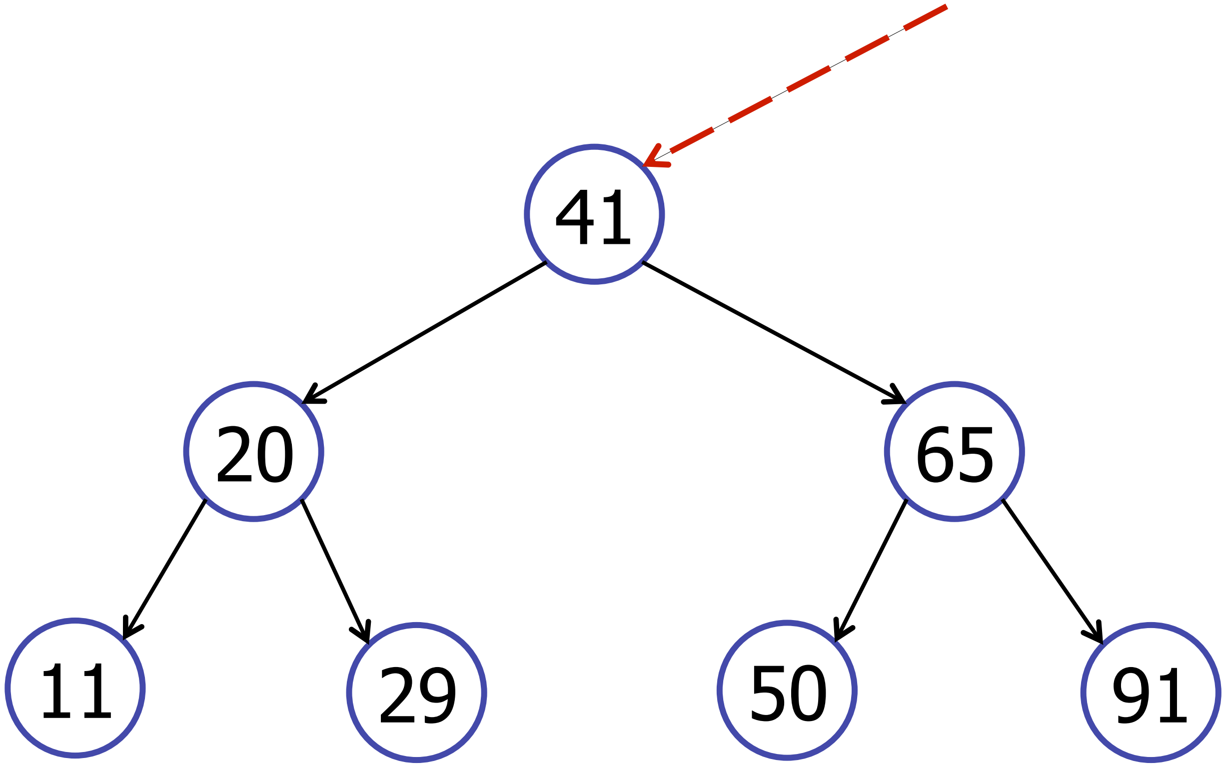
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

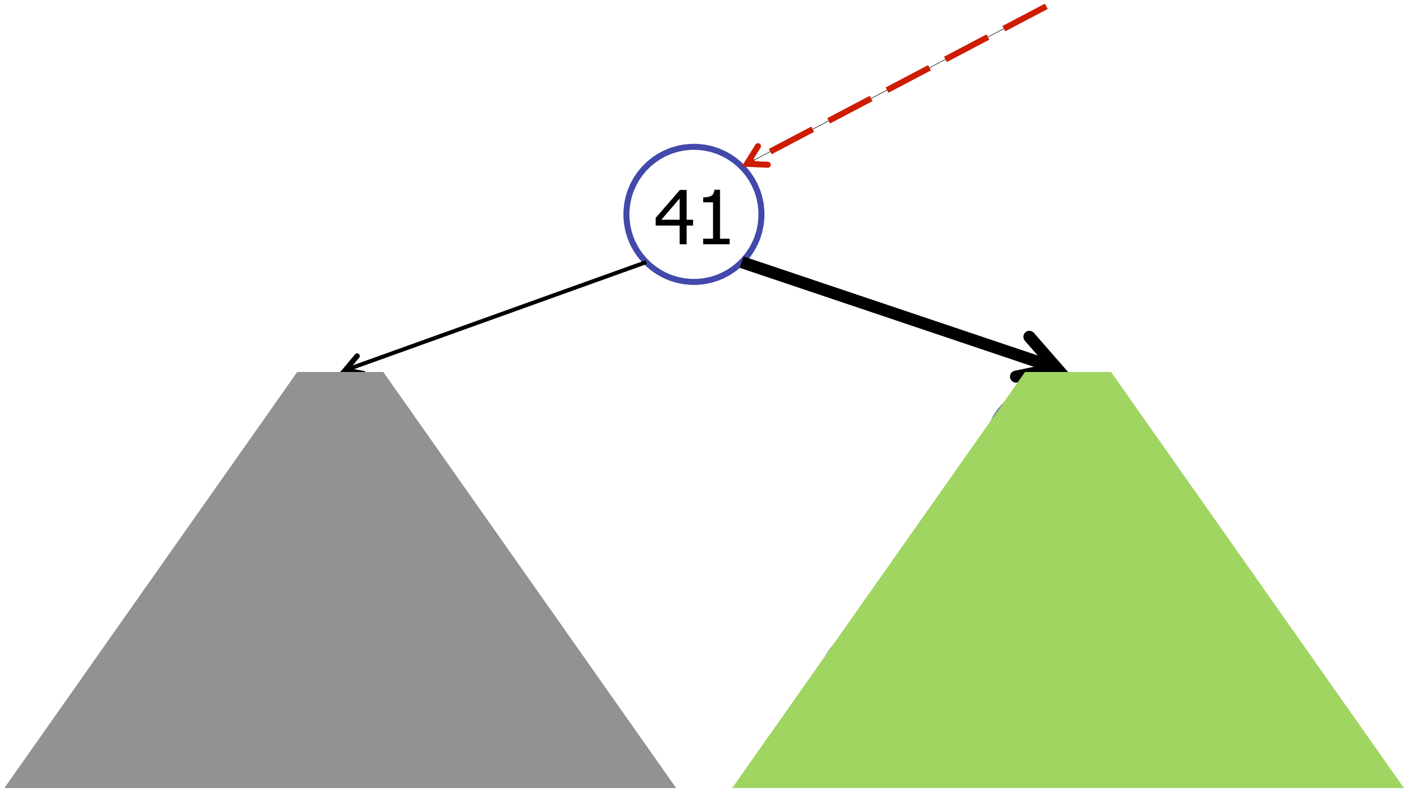
Binary Search Trees

Search for the maximum key:



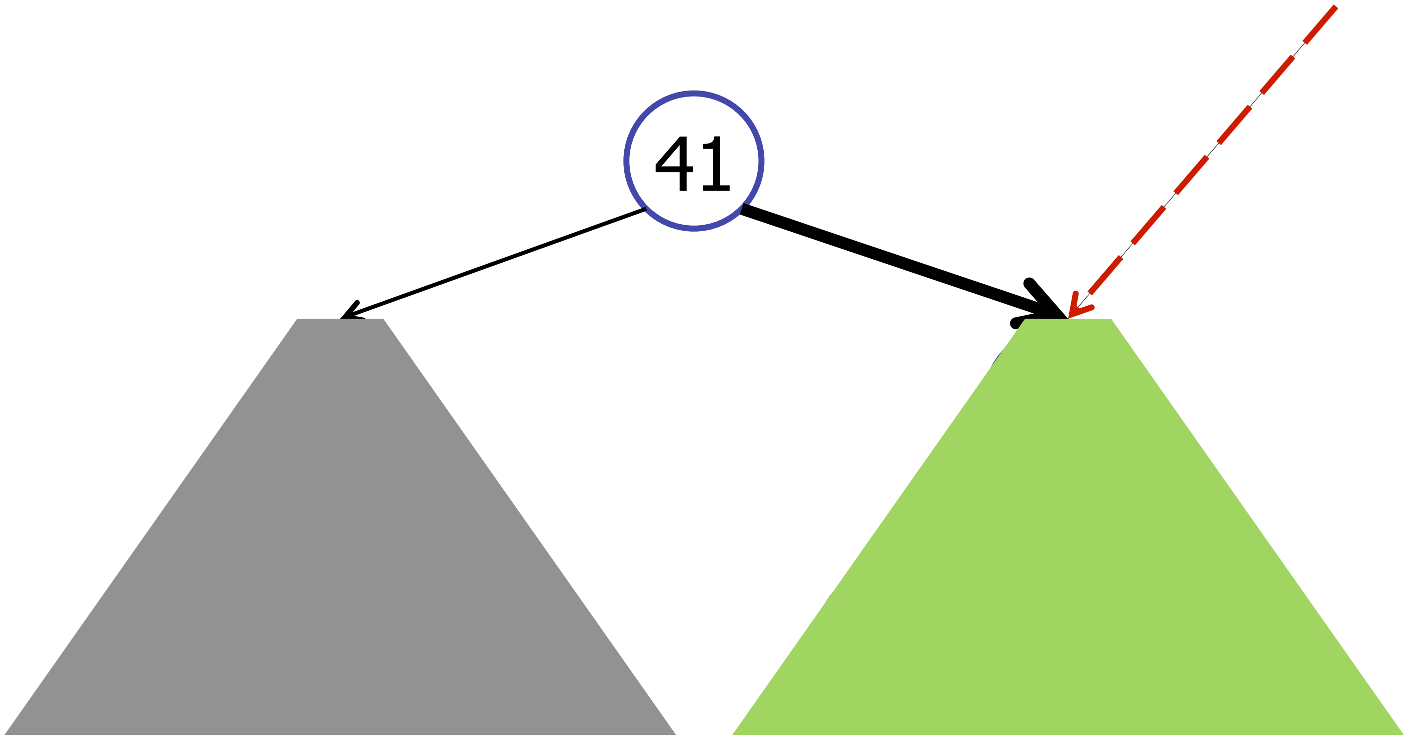
Binary Search Trees

Search for the maximum key:



Binary Search Trees

Search for maximum key:



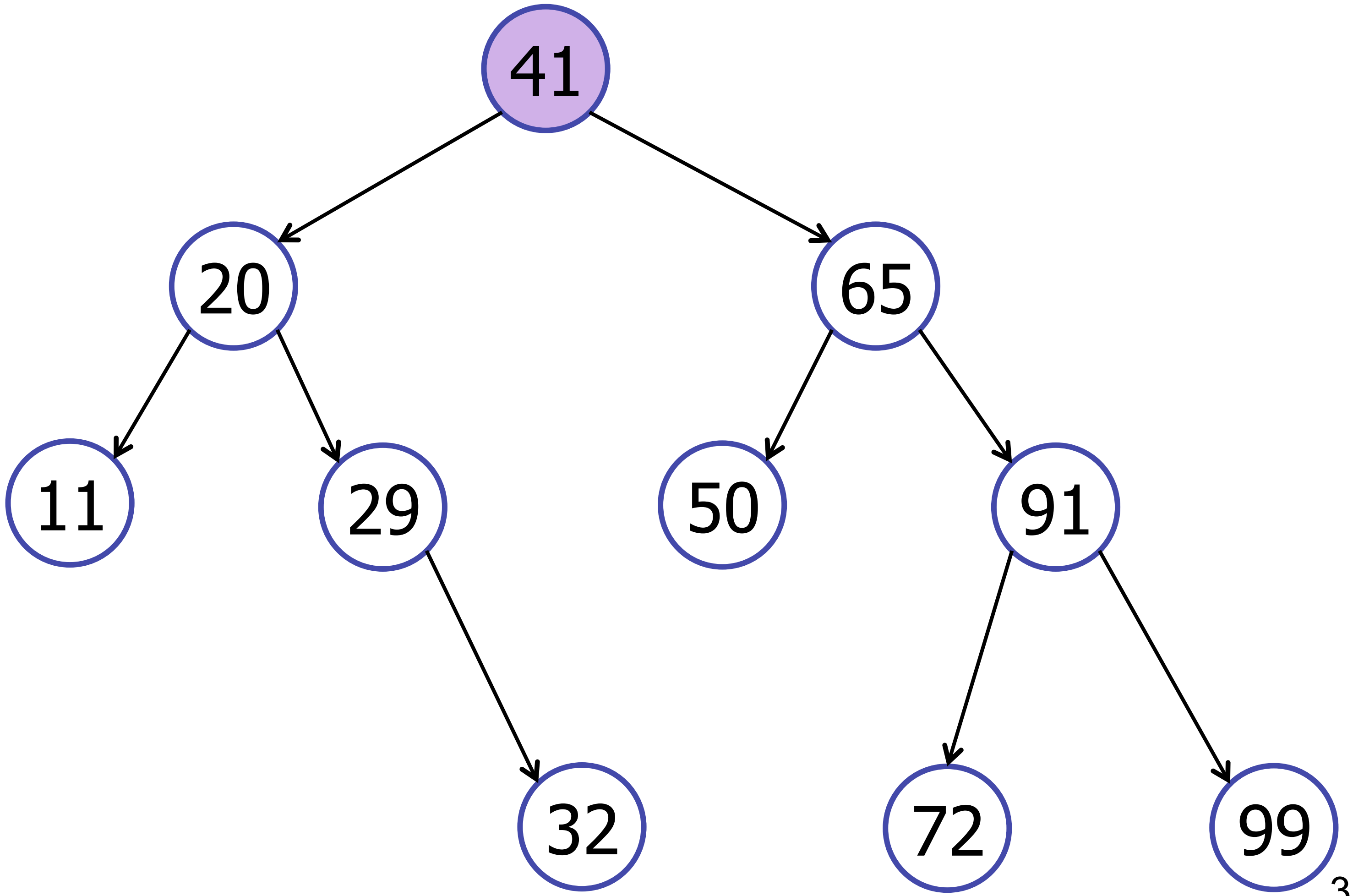
Binary Tree

Searching for the maximum key

```
public BinaryTree<Key> searchMax(){
        if (m_rightTree != null) {
                return m_rightTree.searchMax(key);
        }
        else return this; // Key is here!
}
```

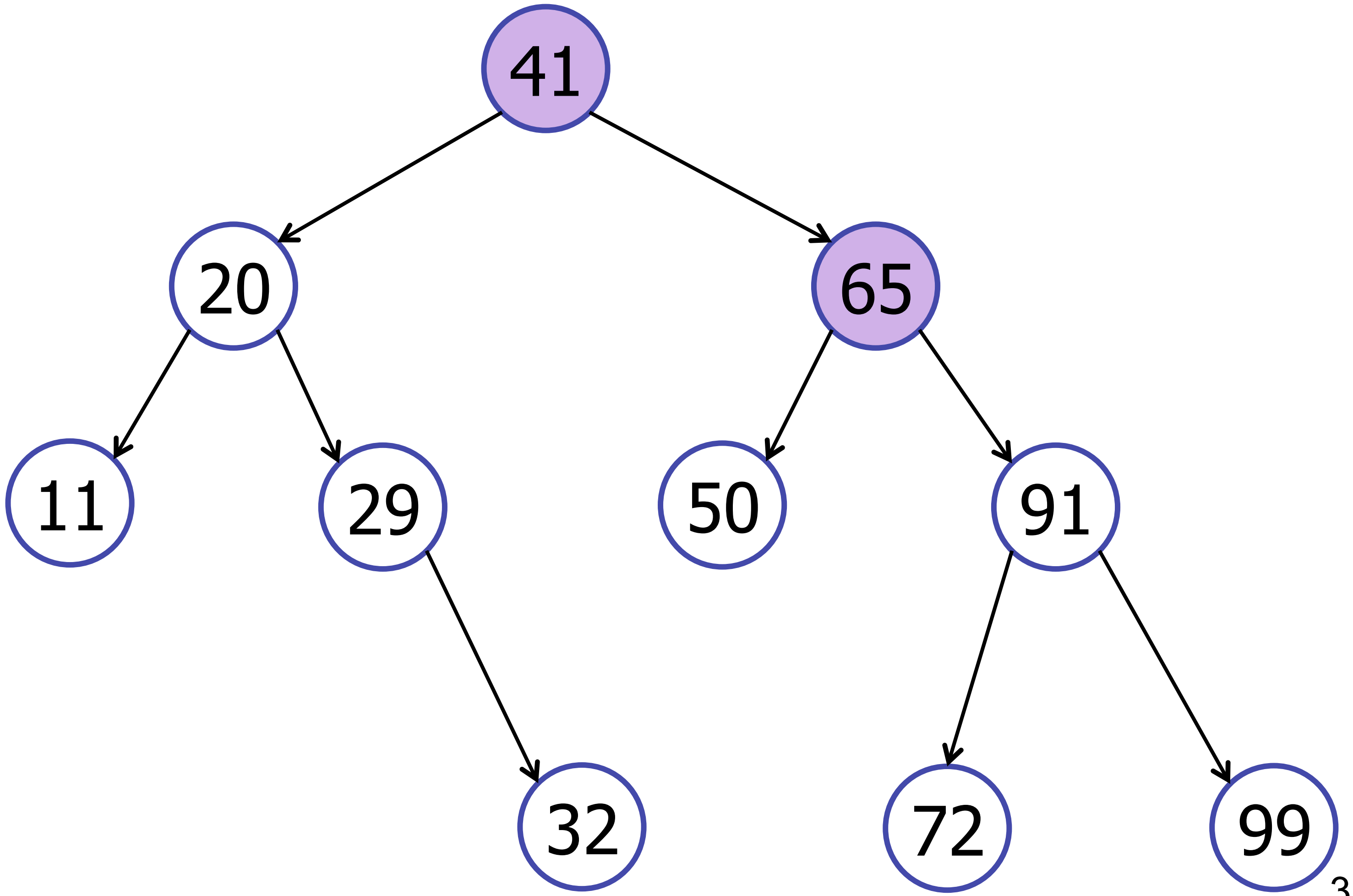
Binary Search Trees

searchMax()



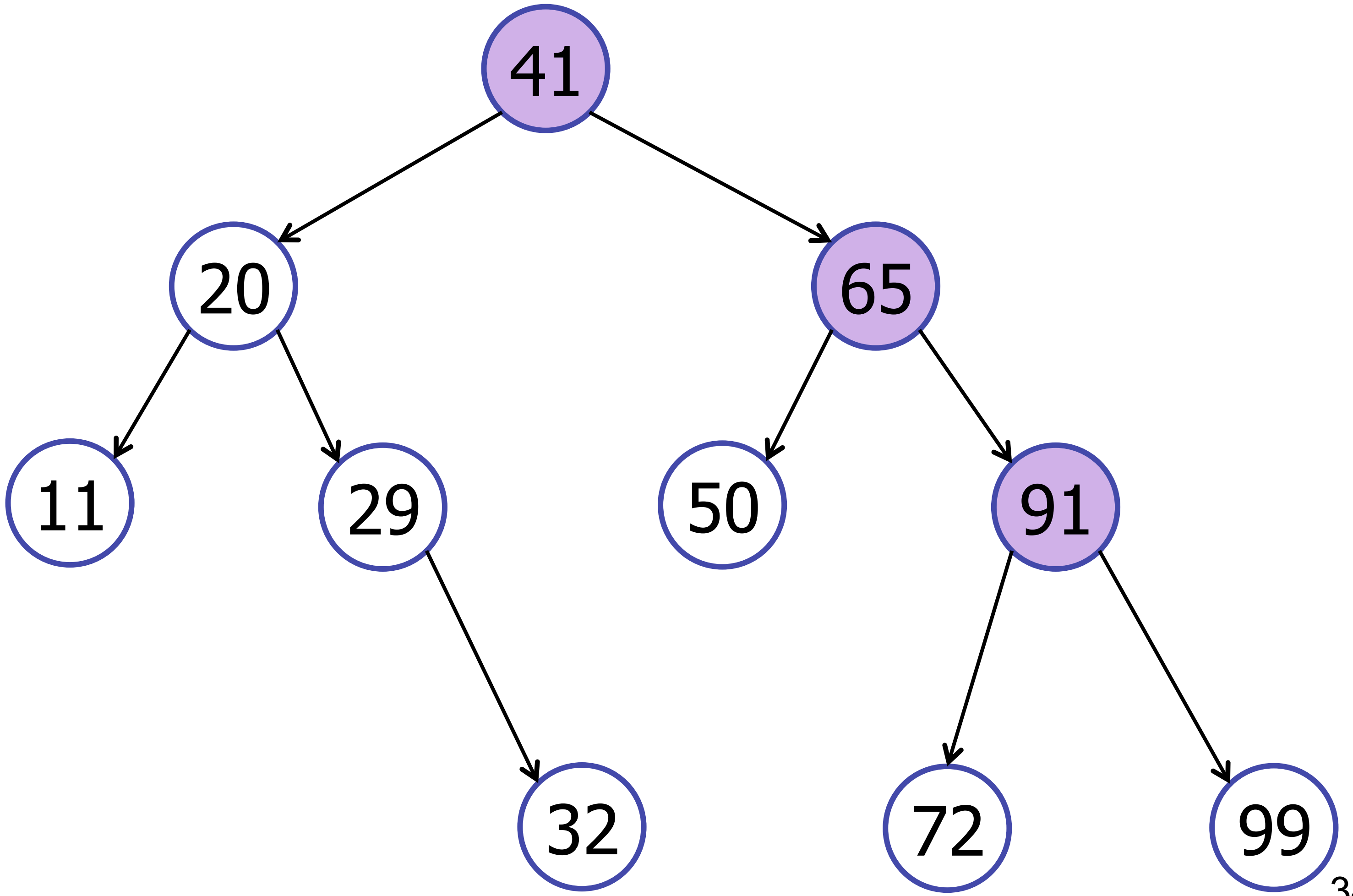
Binary Search Trees

searchMax()



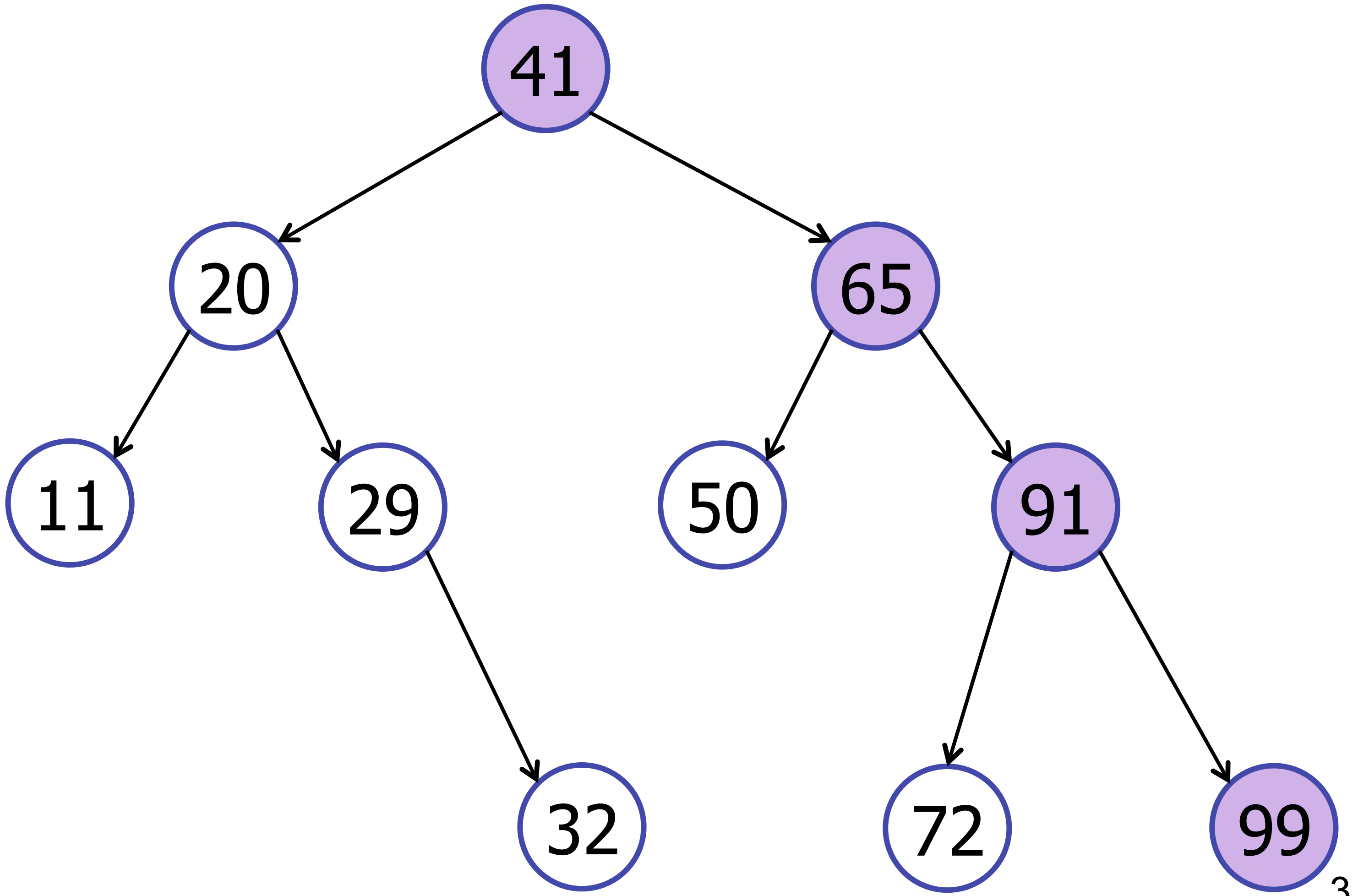
Binary Search Trees

searchMax()



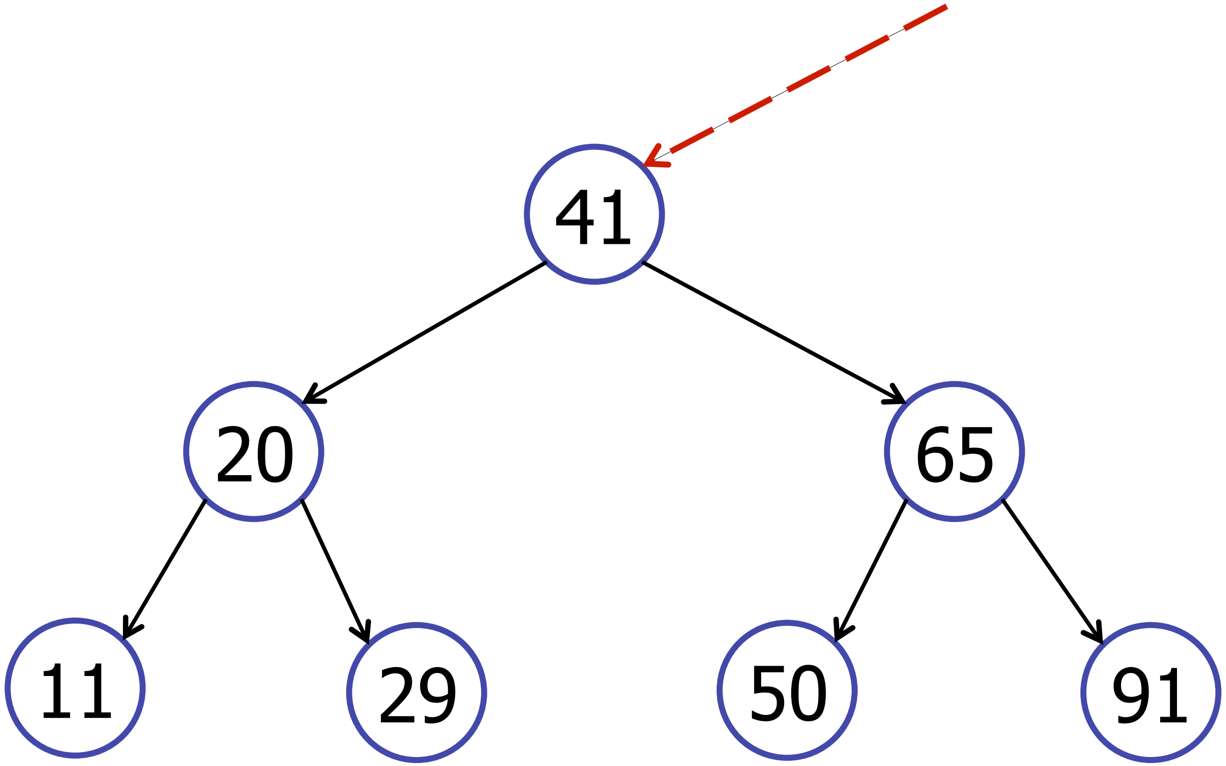
Binary Search Trees

searchMax()



Binary Search Trees

Search for the minimum key:



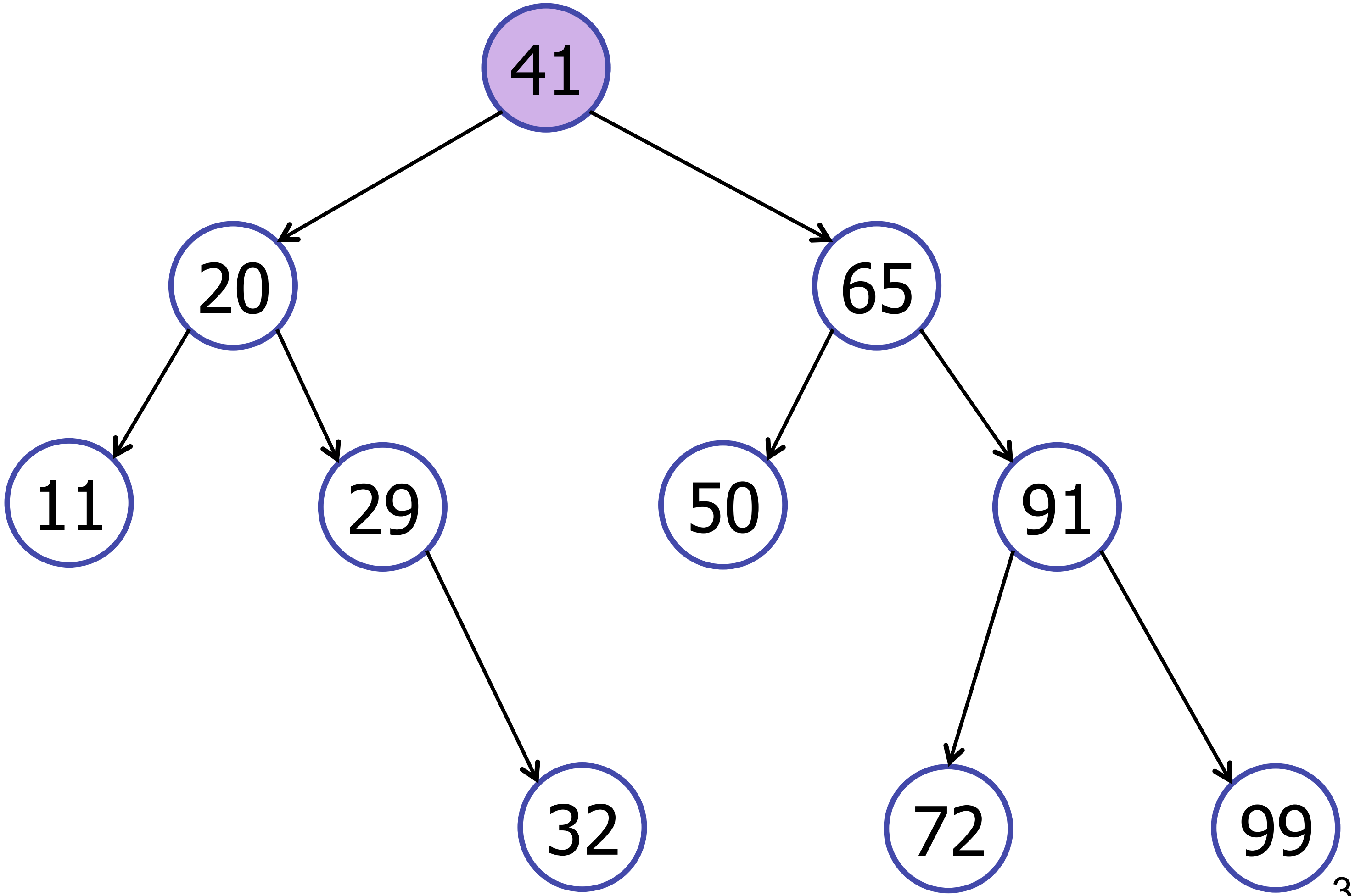
Binary Tree

Searching for the minimum key

```
public BinaryTree<Key> searchMin(){
        if (m_leftTree != null) {
                return m_leftTree.searchMin(key);
        }
        else return this; // Key is here!
}
```

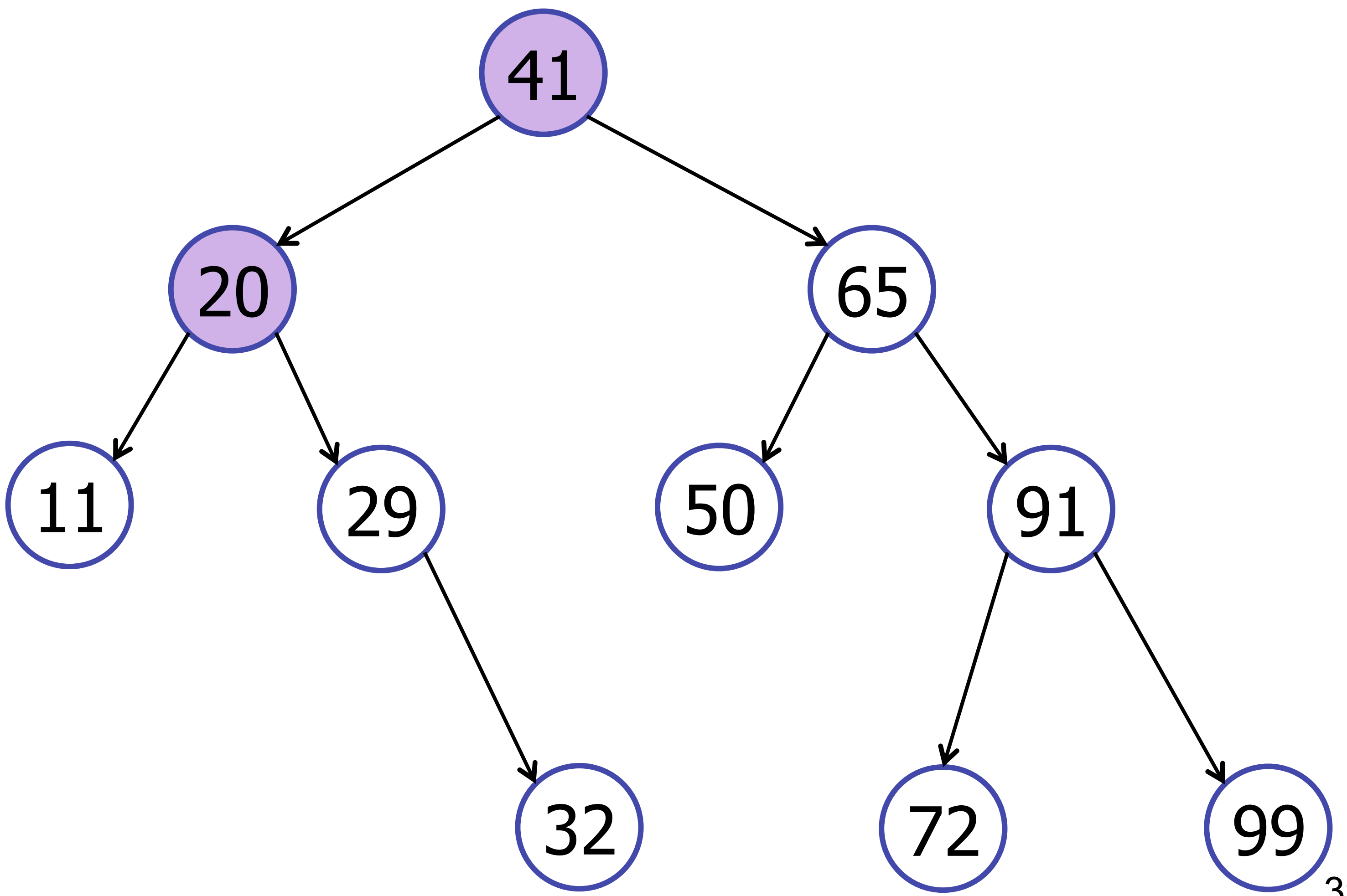
Binary Search Trees

searchMin()



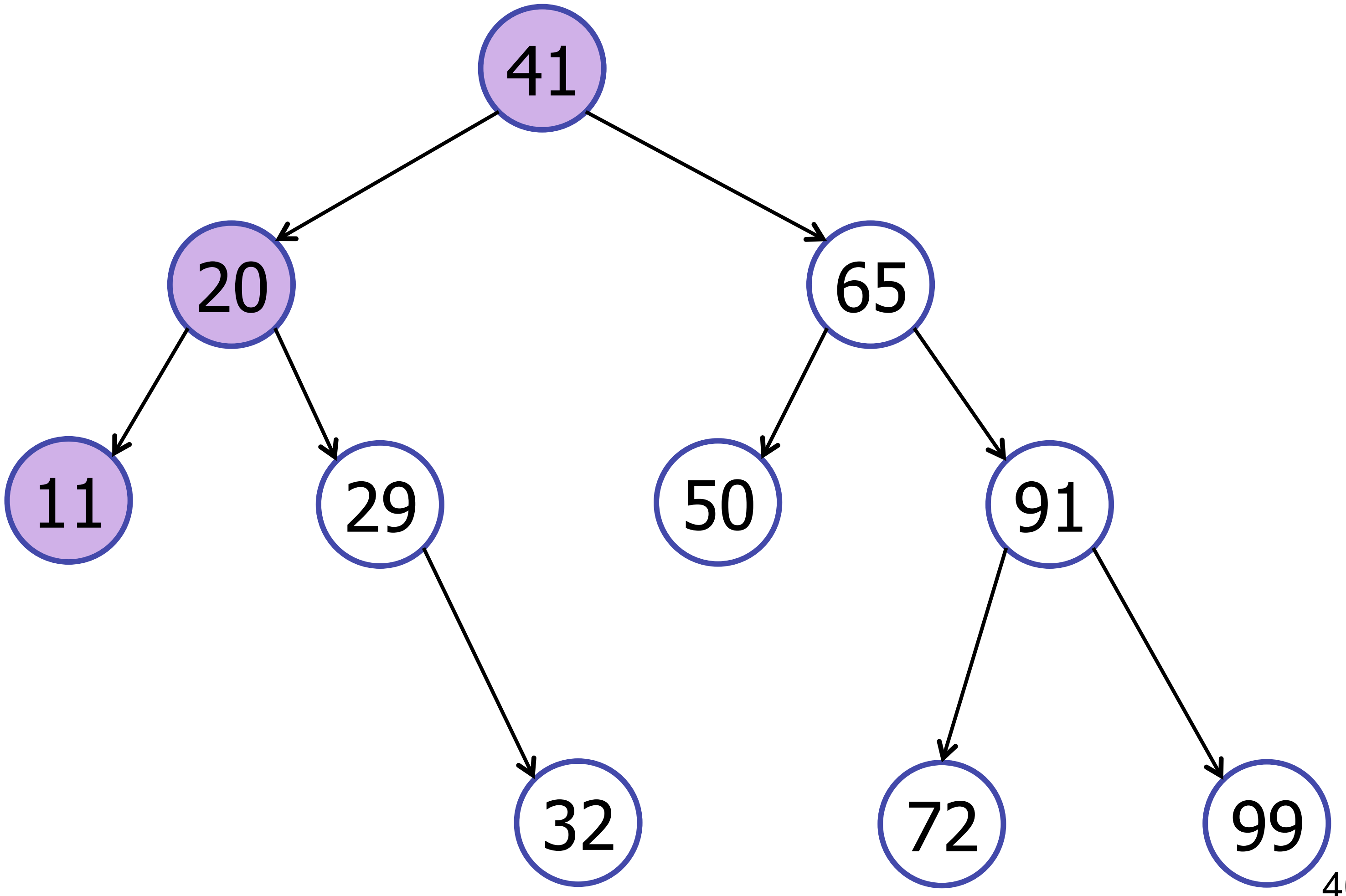
Binary Search Trees

searchMin()



Binary Search Trees

searchMin()



Binary Search Trees

1. Terminology and Definitions

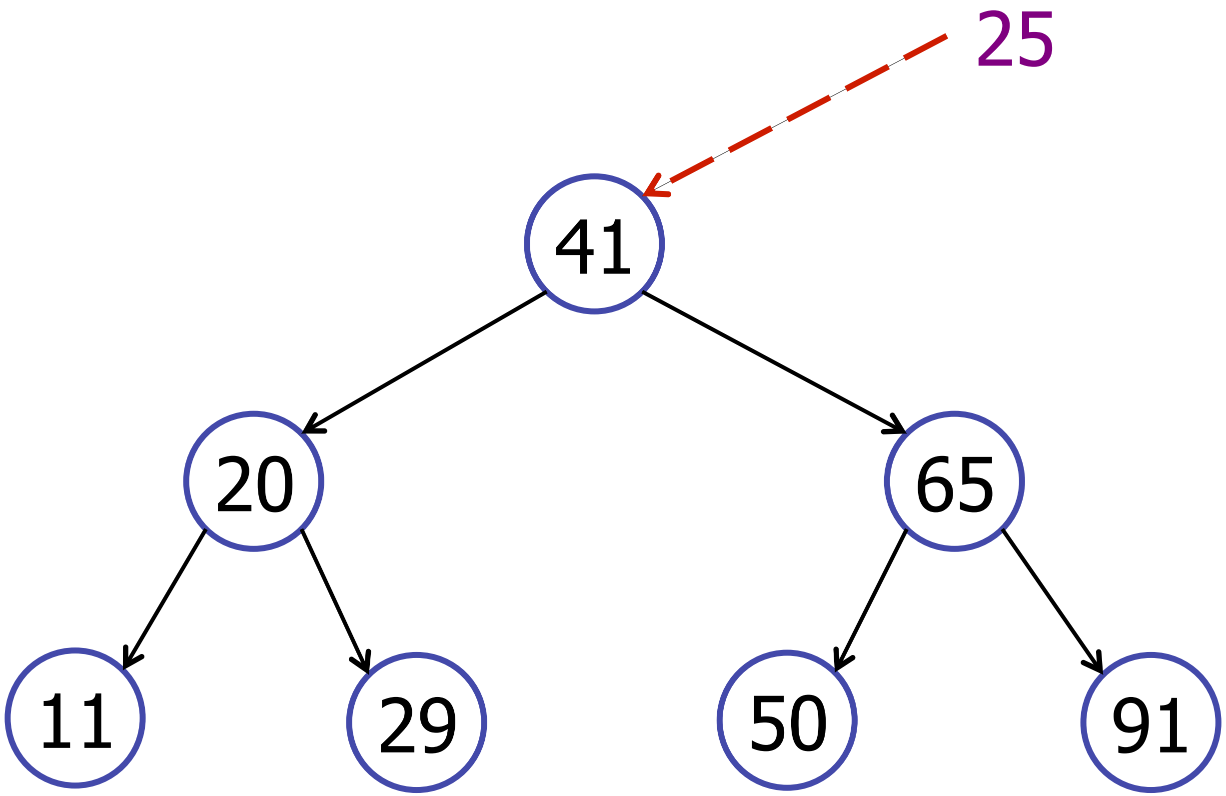
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

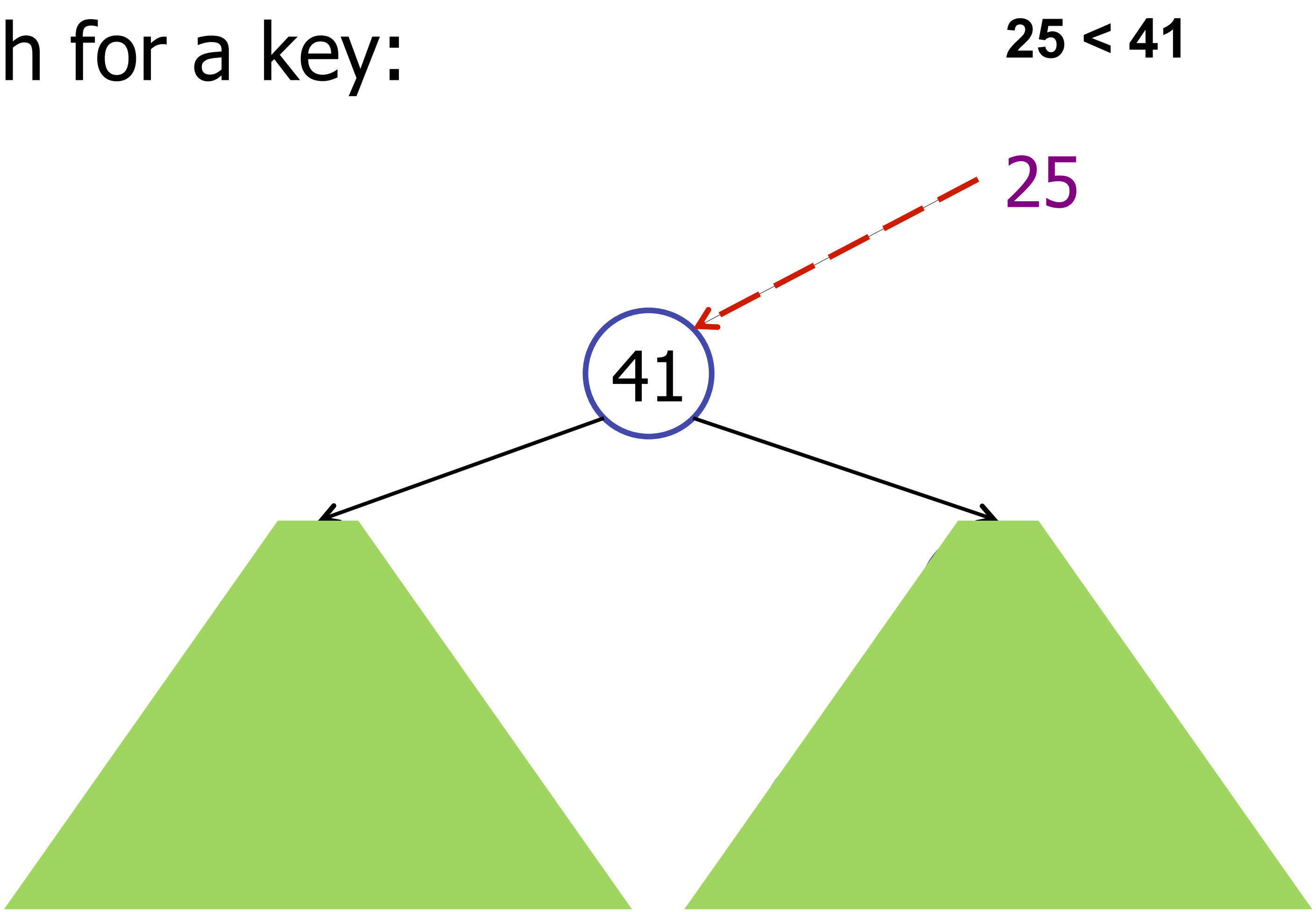
Binary Search Trees

Search for a key:



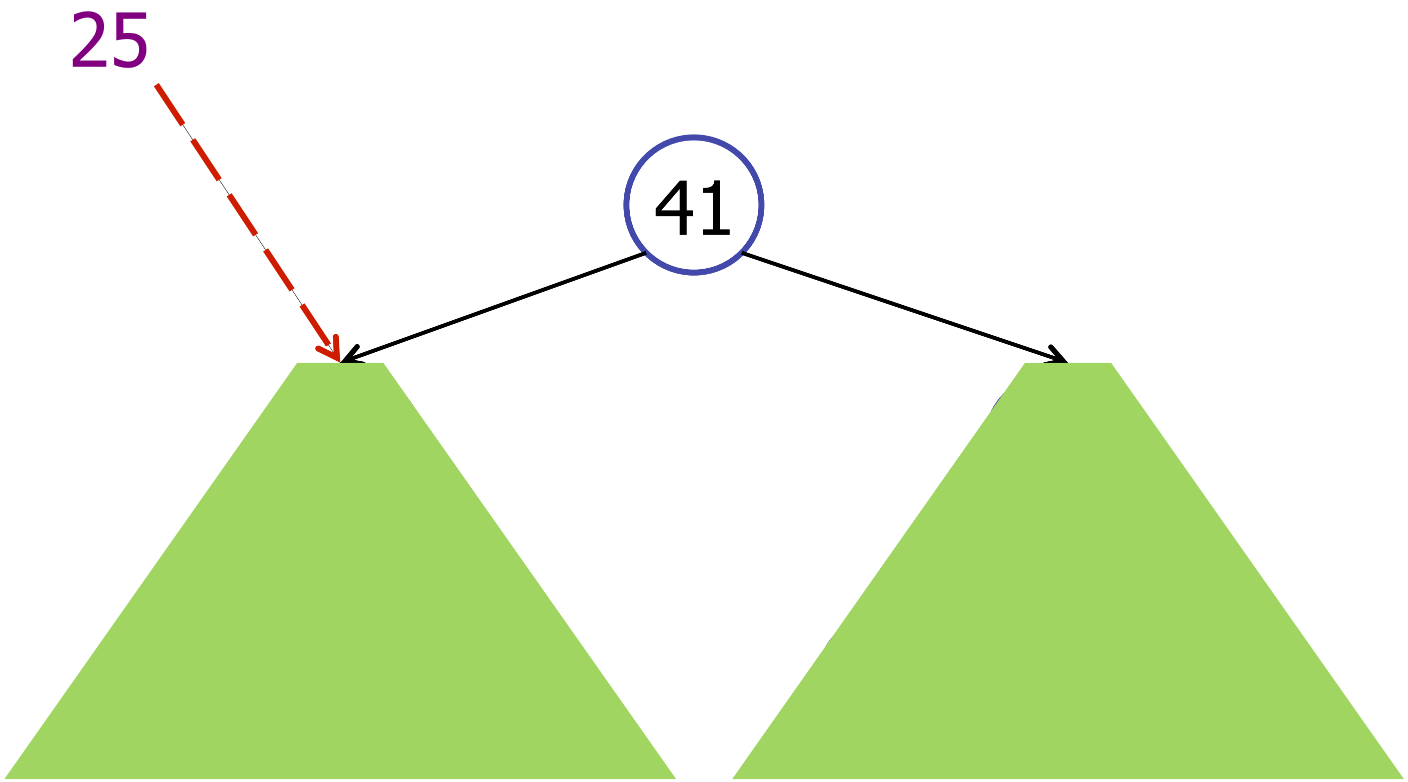
Binary Search Trees

Search for a key:



Binary Search Trees

Search for a key:



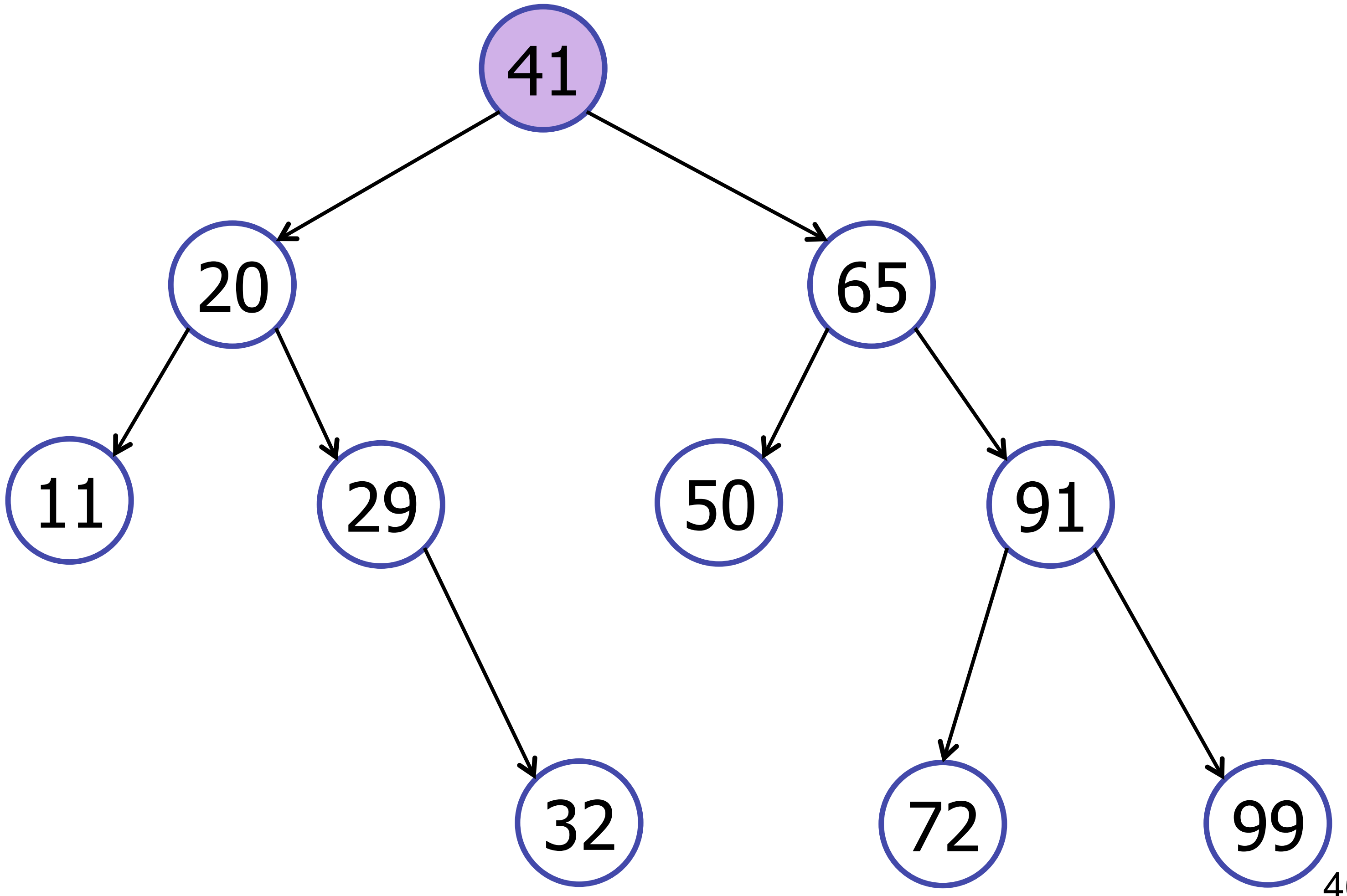
Binary Tree

Inserting a new key

```
public BinaryTree<Key> search(Key key){
        if (key.compareTo(m_key) < 0) {
                if (m_leftTree != null)
                        return m_leftTree.search(key);
                else return null;
        }
        else if (key.compareTo(m_key) > 0) {
                if (m_rightTree != null)
                        return m_rightTree.search(key);
                else return null;
        }
        else return this; // Key is here!
}
```

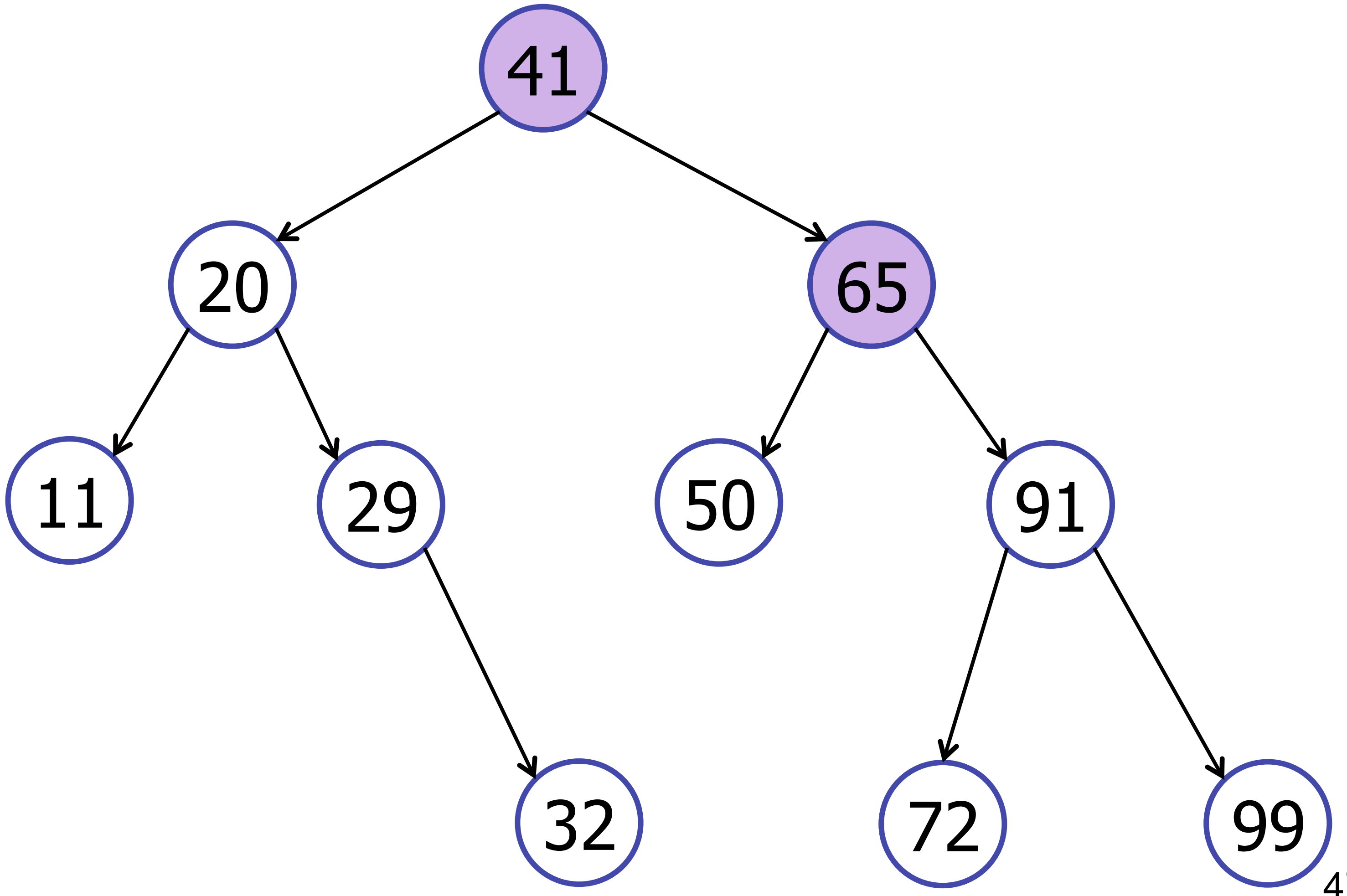
Binary Search Trees

search(72)



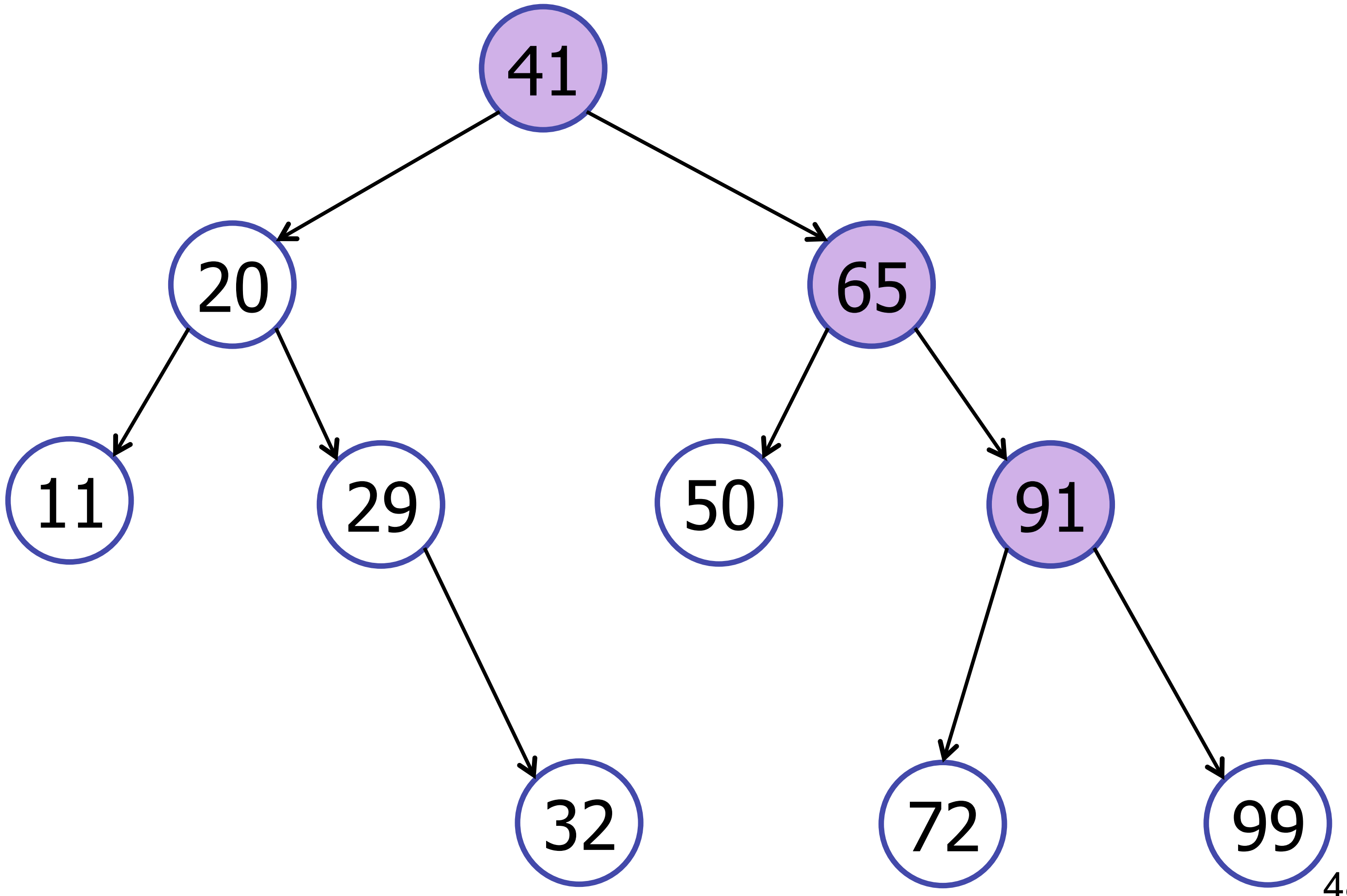
Binary Search Trees

search(72)



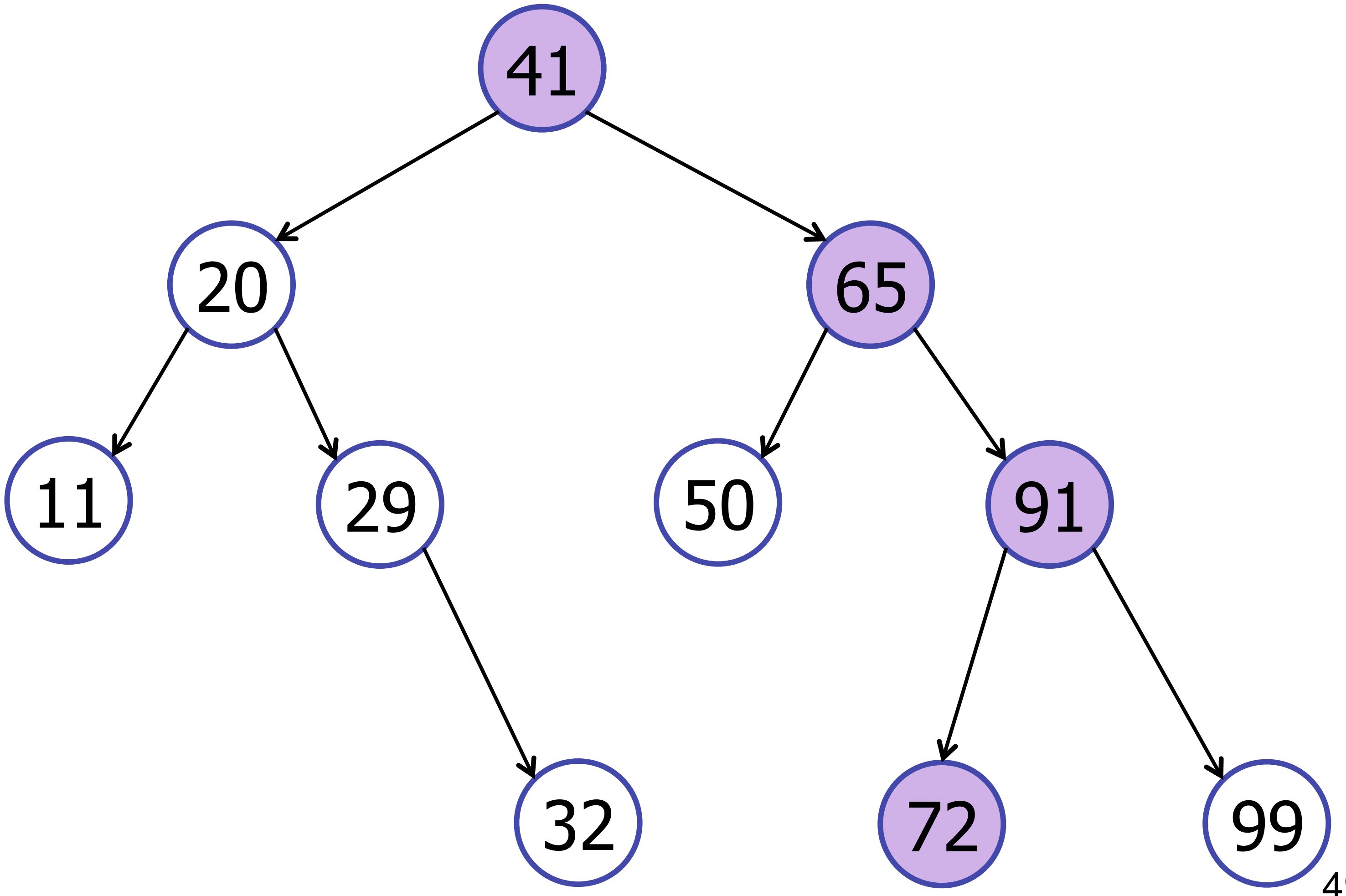
Binary Search Trees

search(72)



Binary Search Trees

search(72)



Binary Search Trees

1. Terminology and Definitions

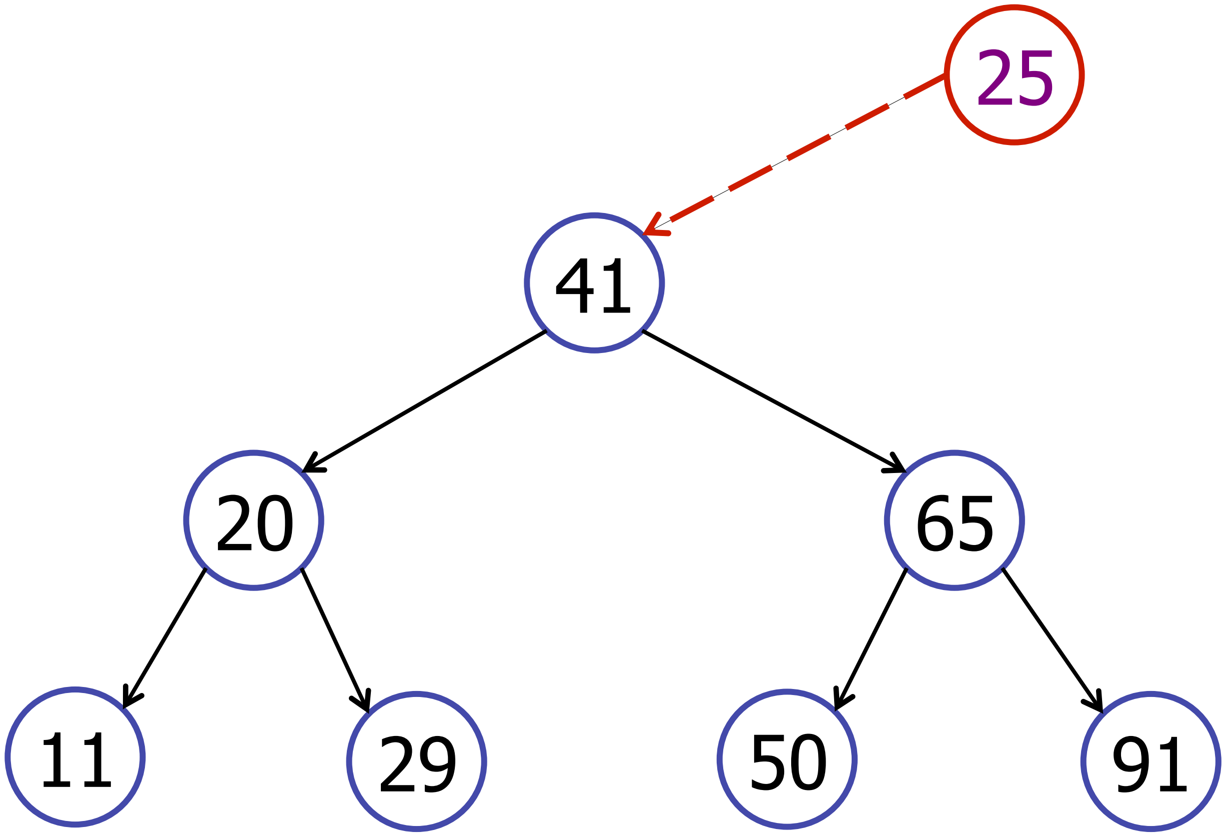
2. Basic operations:
   - height
   - searchMin, searchMax
   - search, insert

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

4. Other operations

Binary Search Trees

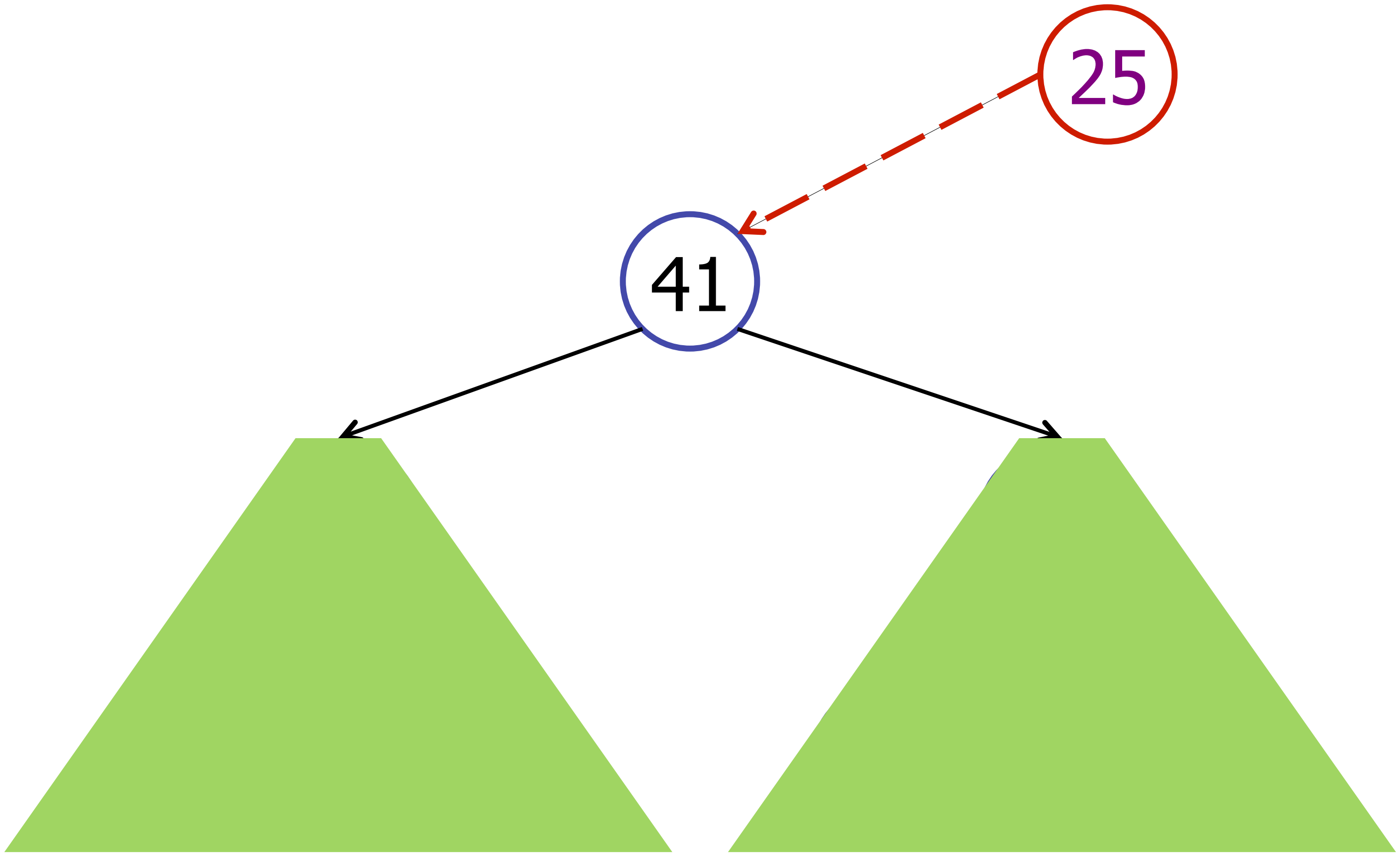
Inserting a new key:



Binary Search Trees

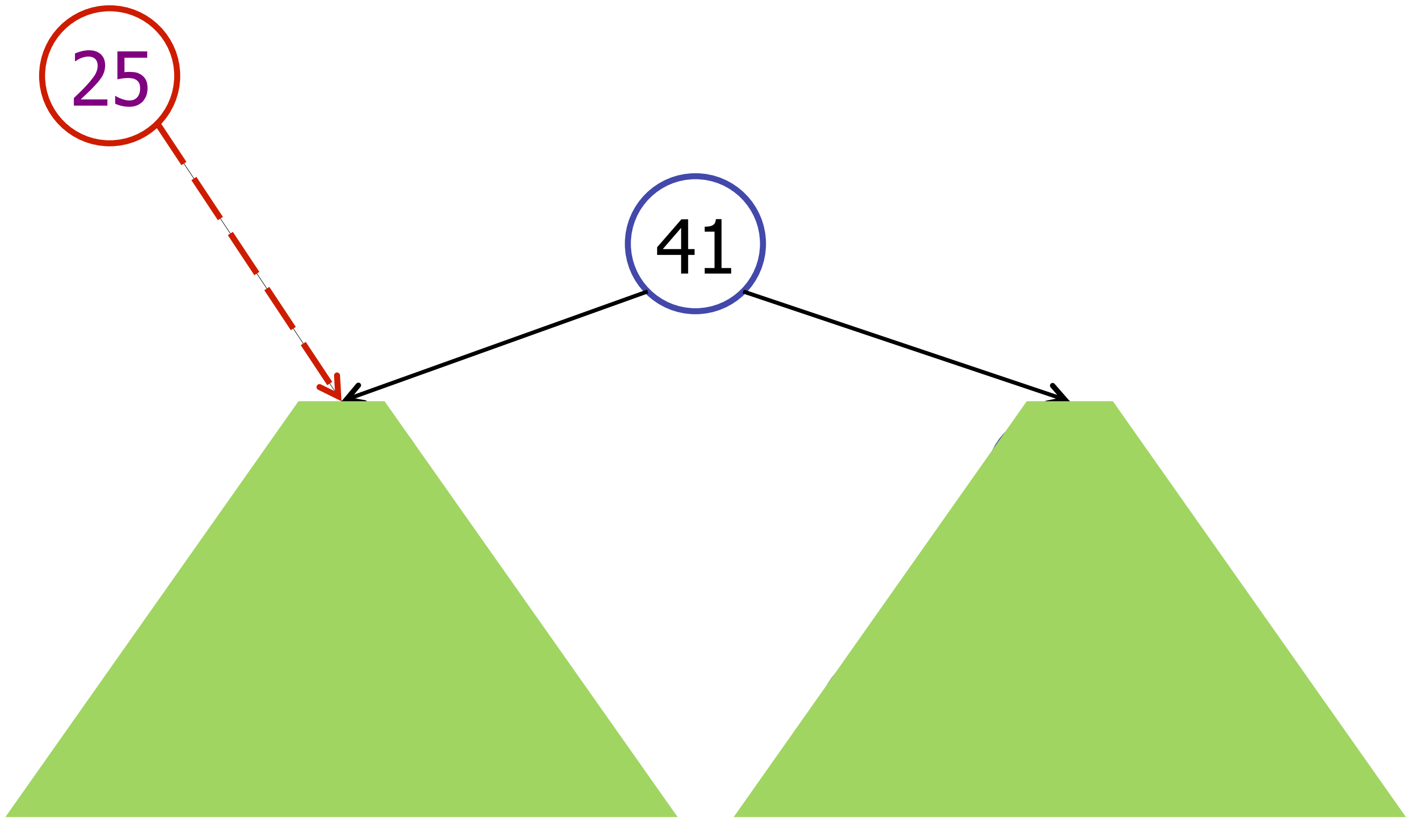
25 < 41

Inserting a new key:



Binary Search Trees

Inserting a new key:



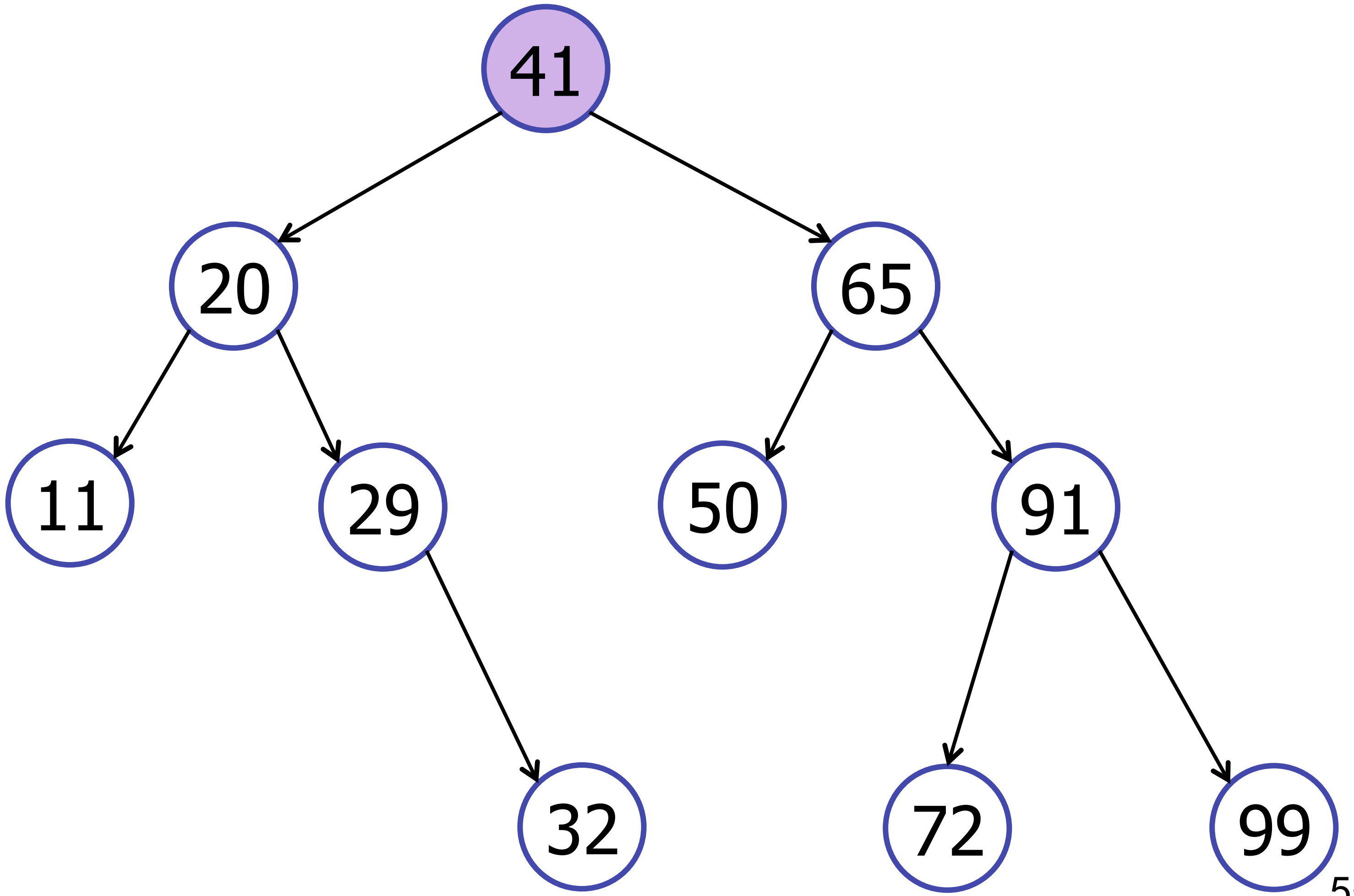
Binary Tree

Inserting a new key

```
public void insert(Key key){
        if (key.compareTo(m_key) < 0) {
                if (m_leftTree != null)
                        m_leftTree.insert(key);
                else m_leftTree = new BinaryTree<Key>(key);
        }
        else if (key.compareTo(m_key) > 0) {
                if (m_rightTree != null)
                        m_rightTree.insert(key);
                else m_rightTree = new BinaryTree<Key>(key);
        }
        else return; // Key is already in the tree!
}
```

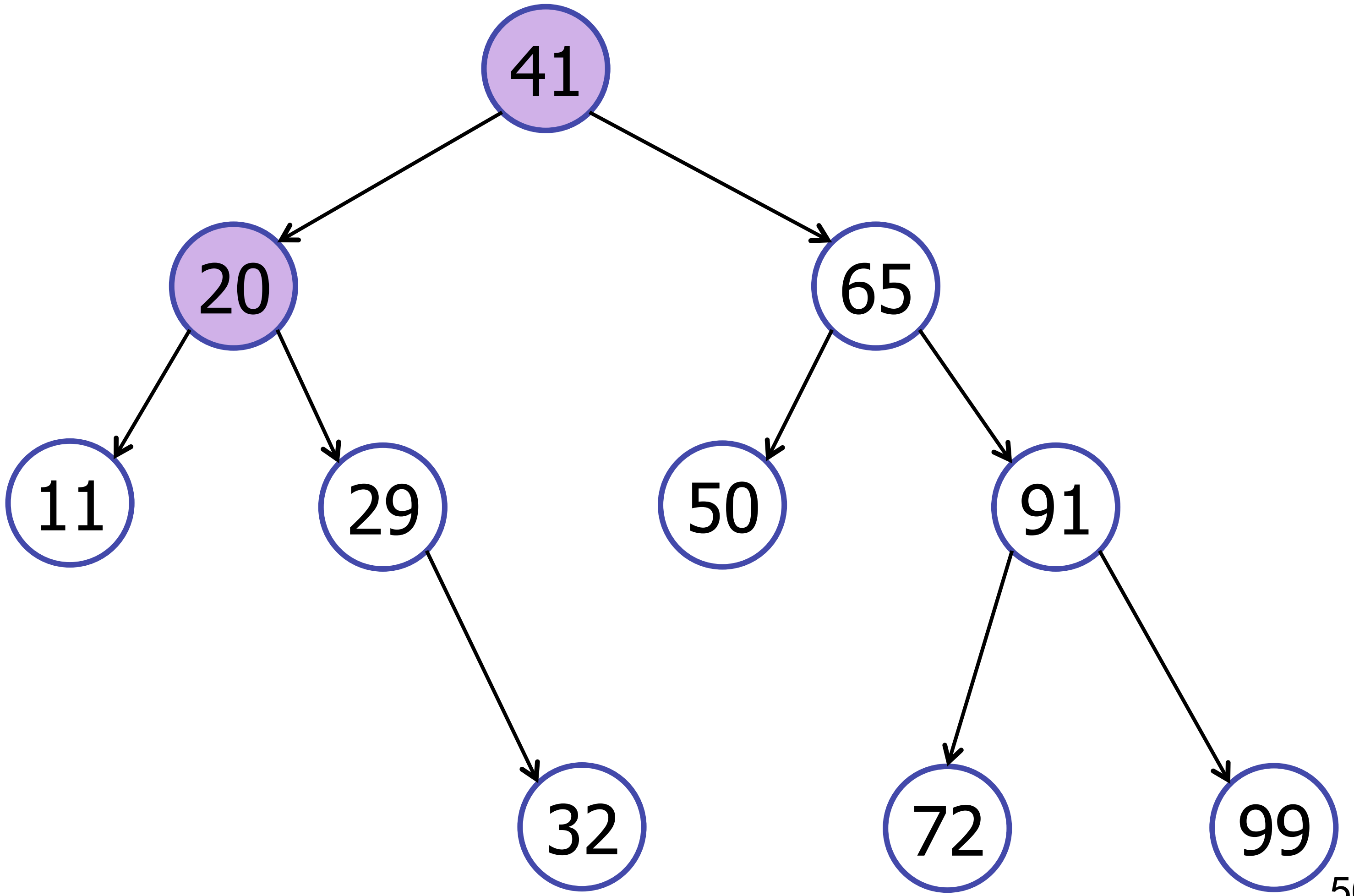
Binary Search Trees

insert(27)



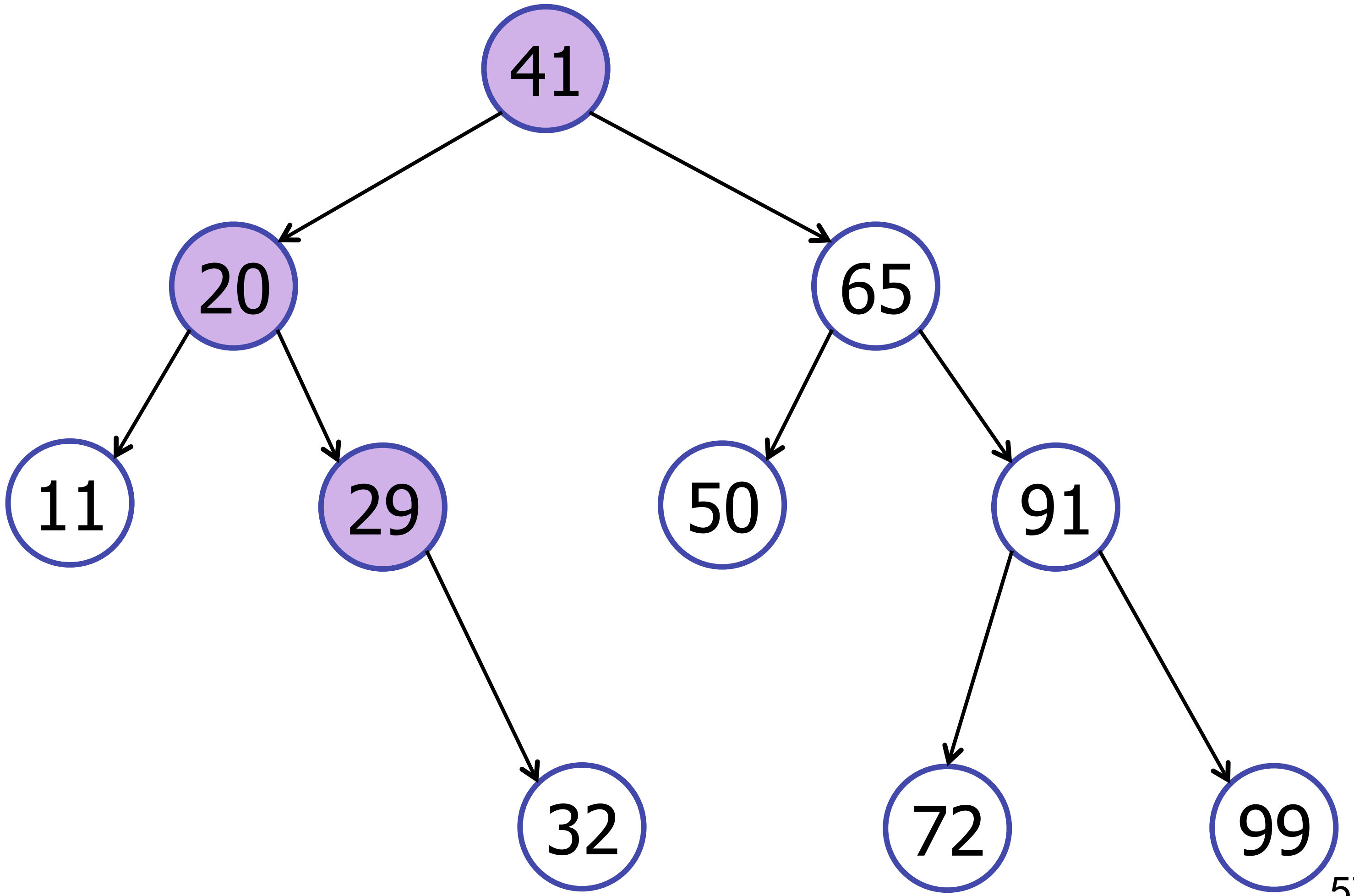
Binary Search Trees

insert(27)



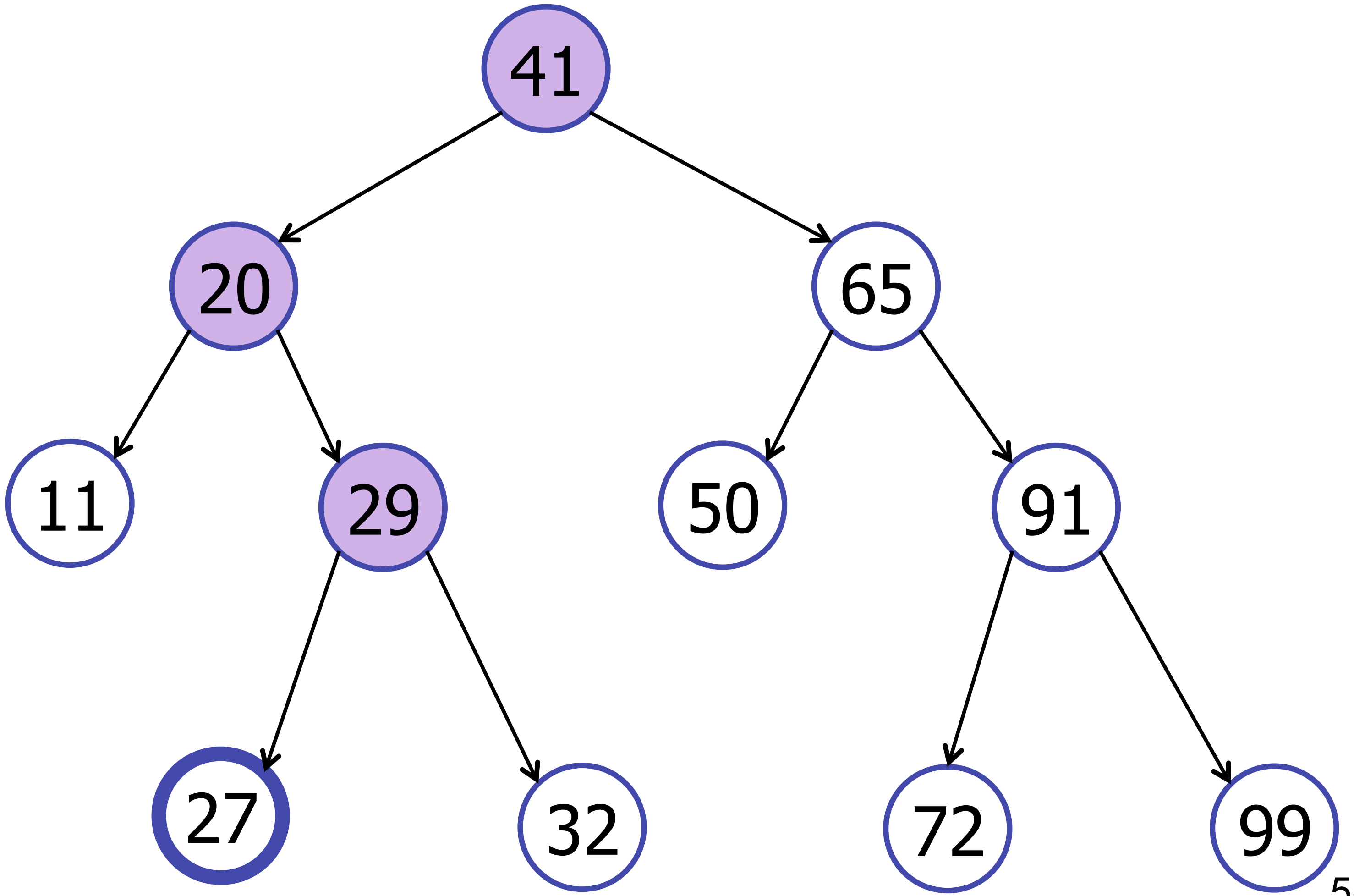
Binary Search Trees

insert(27)



Binary Search Trees

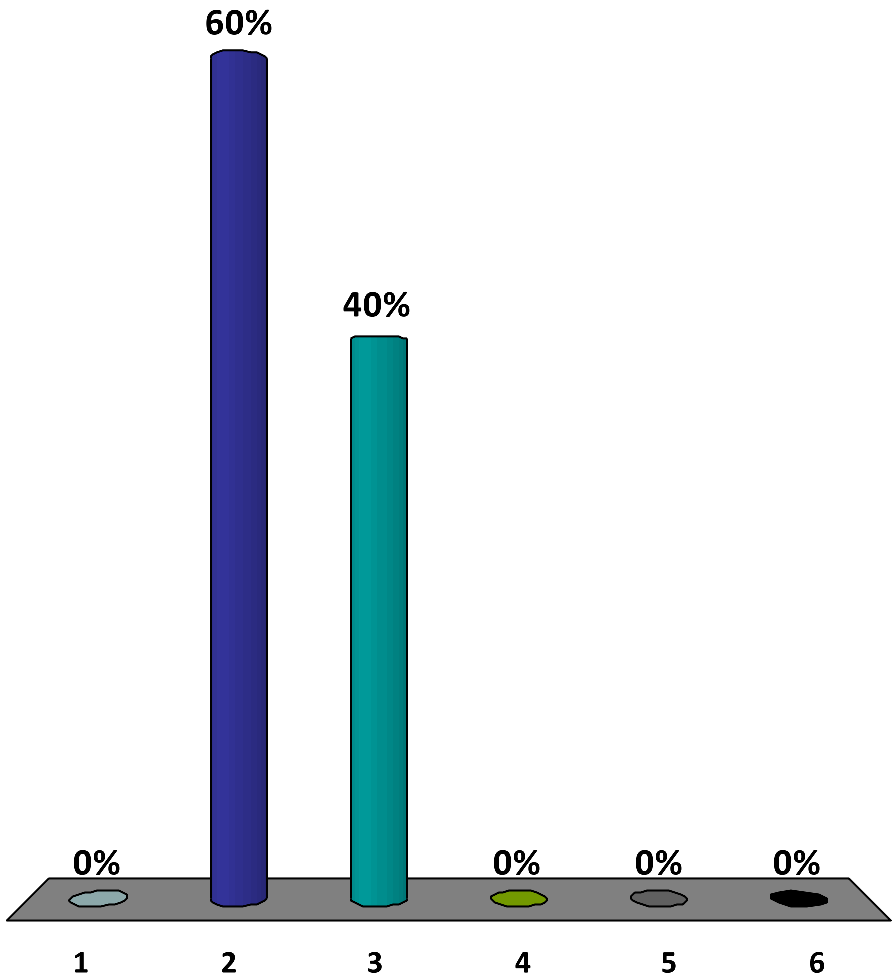
insert(27)



Binary Search Tree

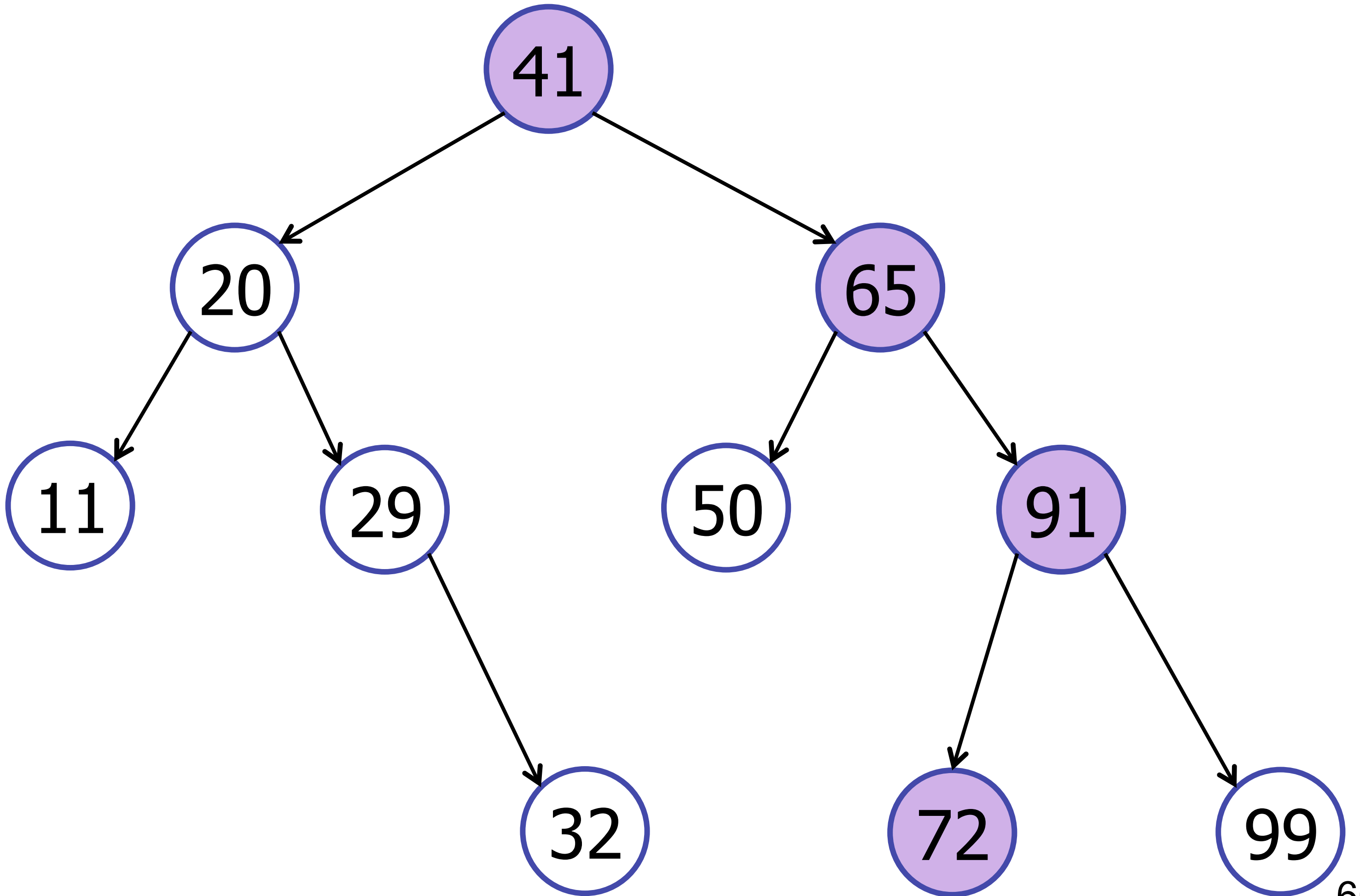
What is the worst-case running time of search in a BST?

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



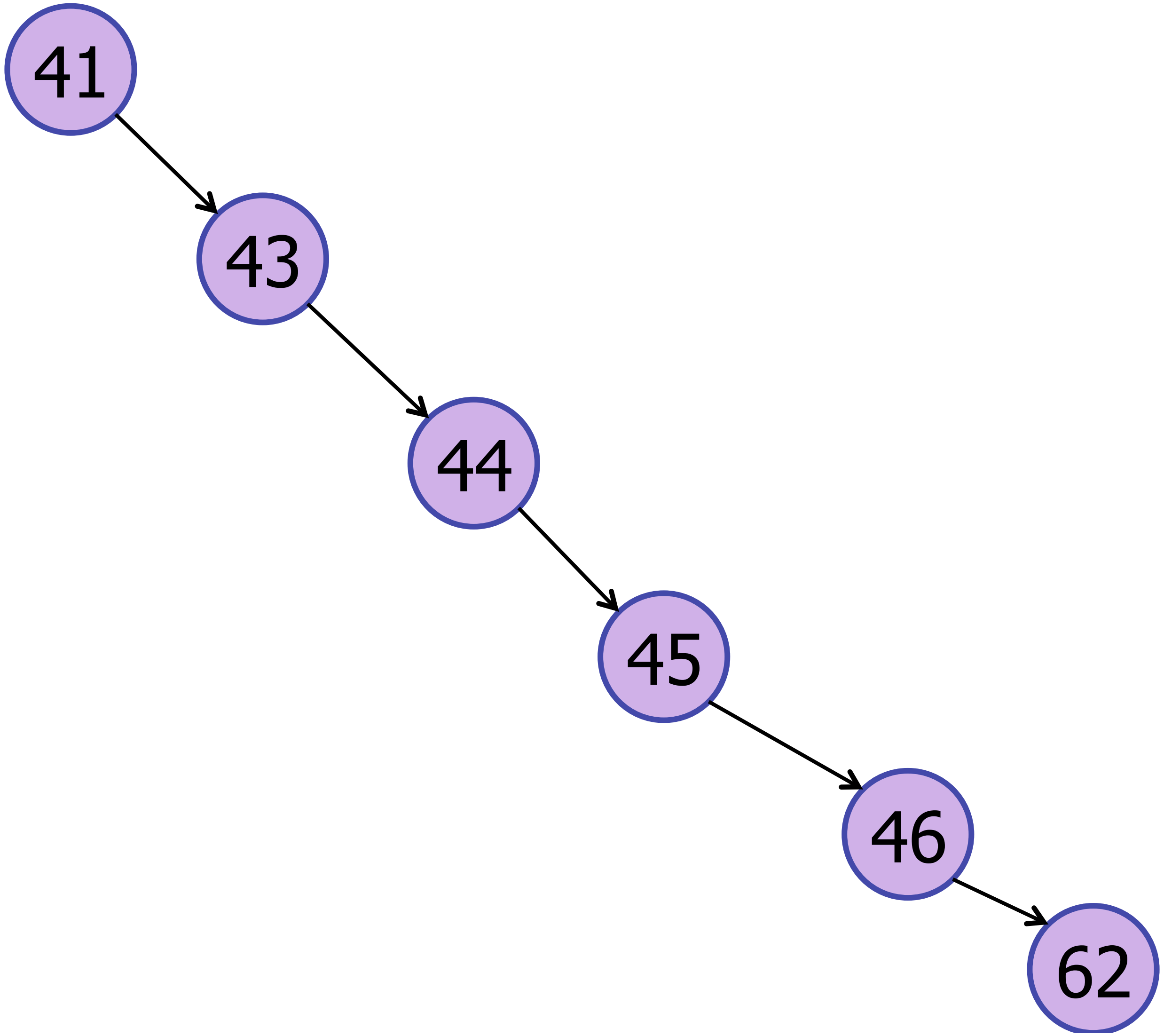
Binary Search Trees

search(72) : O(h)



Binary Search Trees

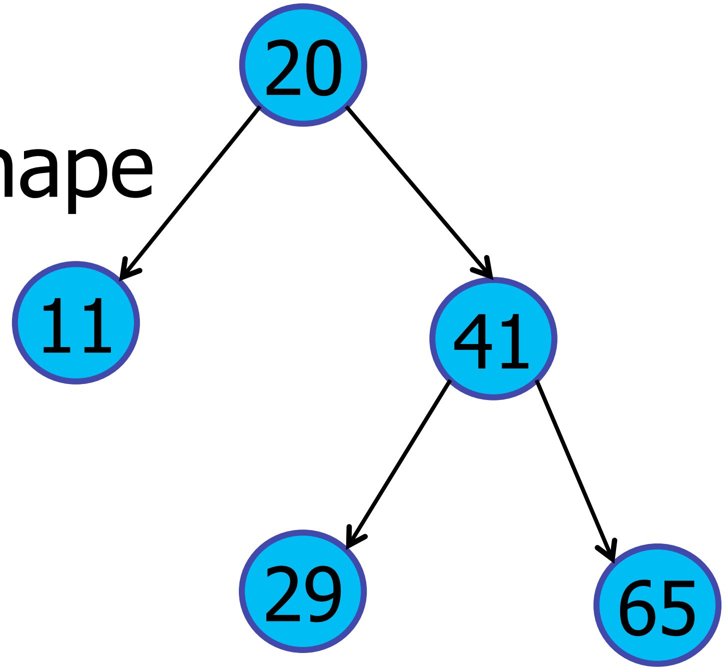
search(72) : O(h)

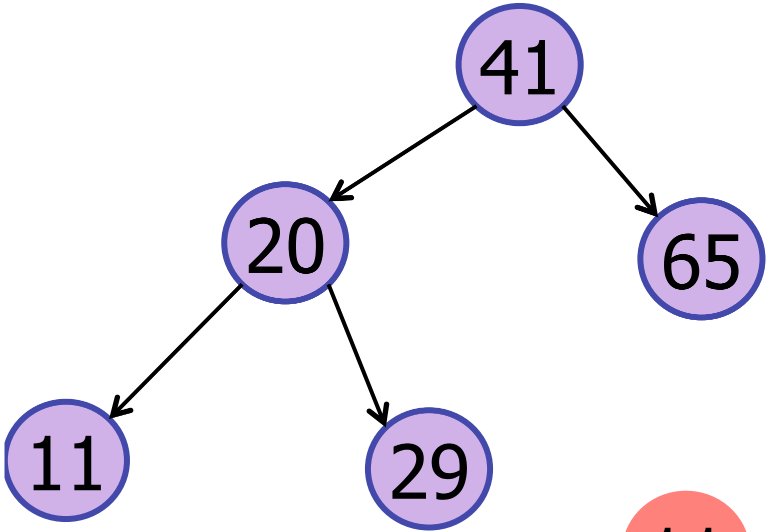


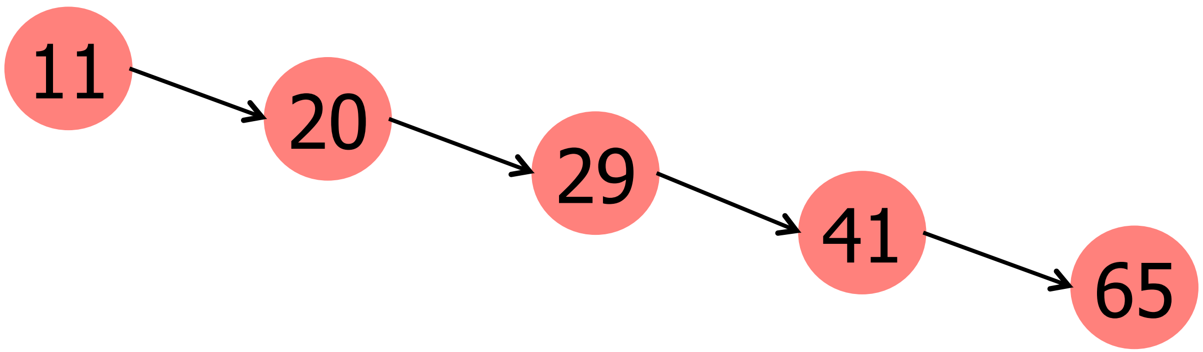
Tree Shape

Trees come in many shapes

- same keys ≠ same shape
- performance depends on shape



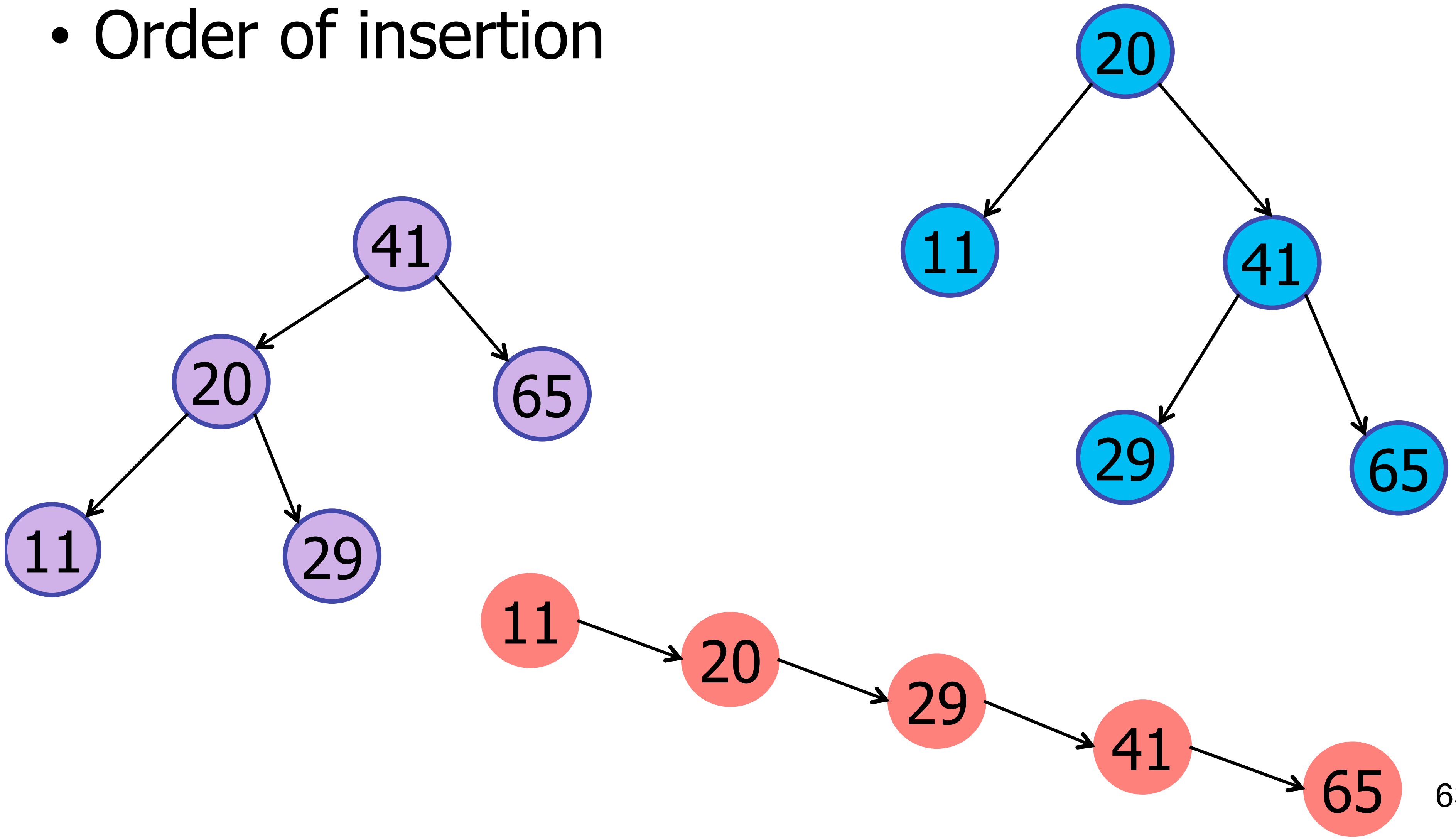




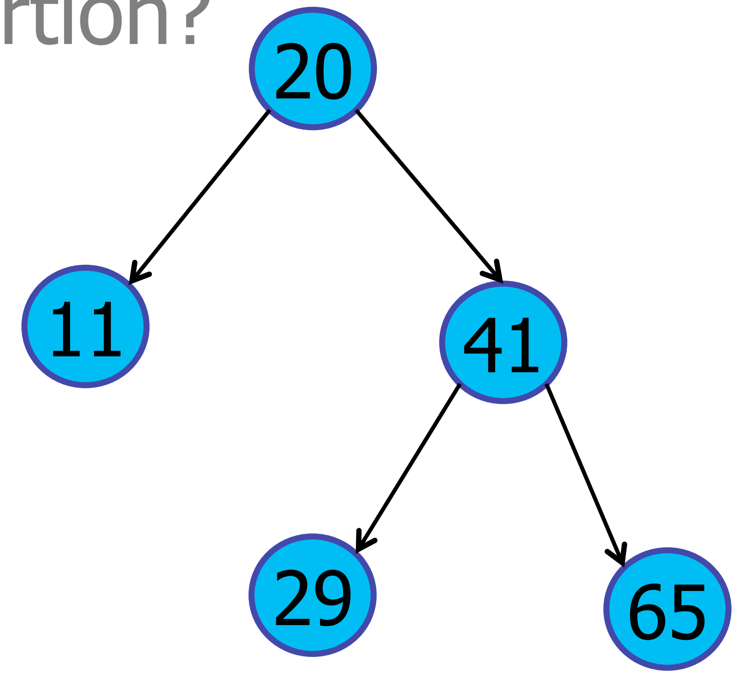
Tree Shape

What determines shape?

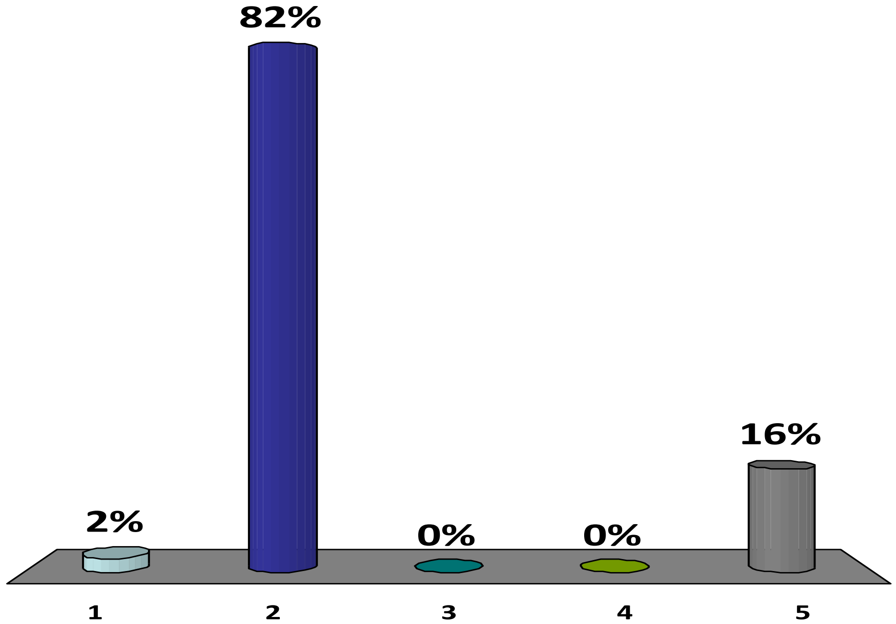
- Order of insertion



What was the order of insertion?



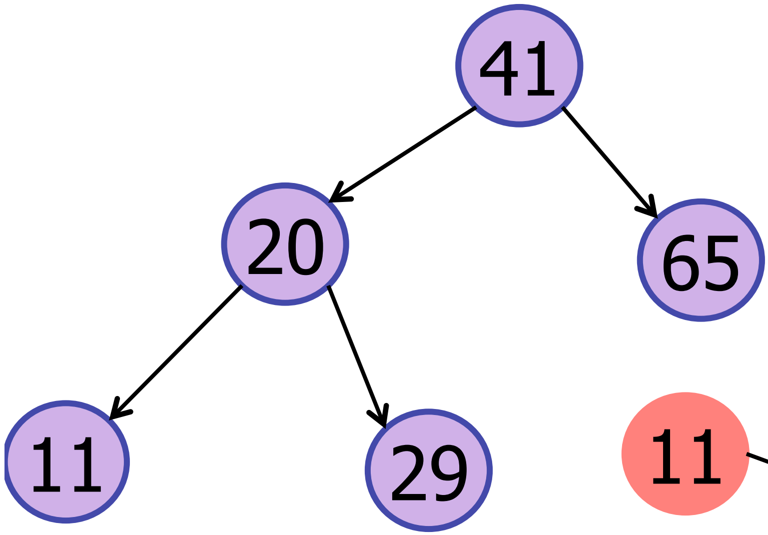
1. 11, 20, 29, 41, 65
2. 20, 11, 41, 29, 65
3. 11, 20, 41, 29, 65
4. 65, 41, 29, 20, 11
5. Impossible to tell.

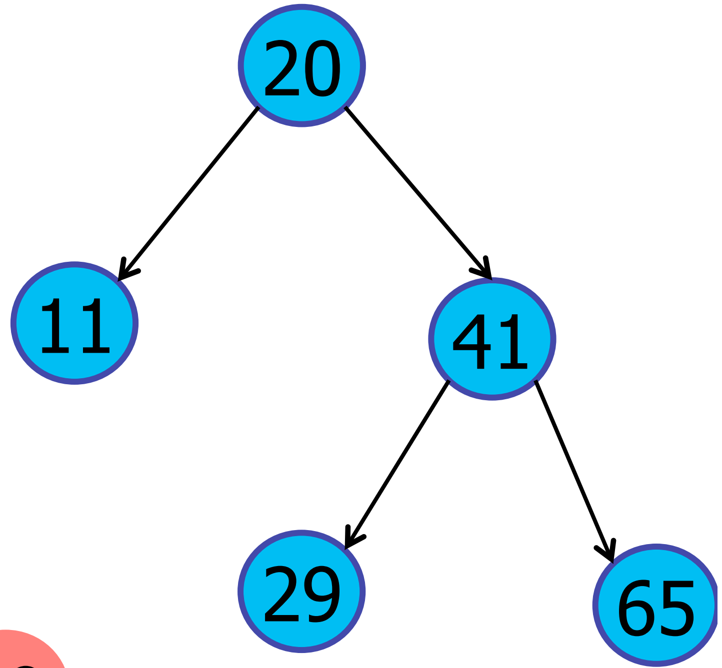


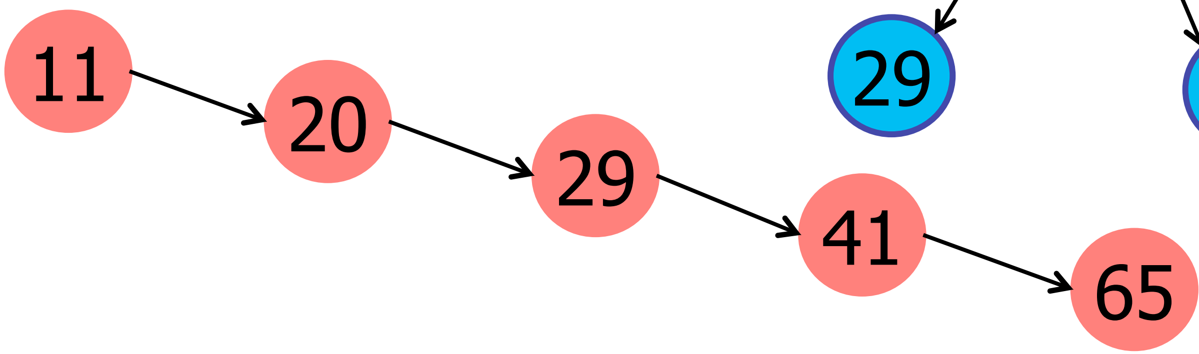
Tree Shape

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

Catalan Numbers

- C_n = # of trees with (n+1) leaves

- C_n = # expressions with n pairs of matched parentheses

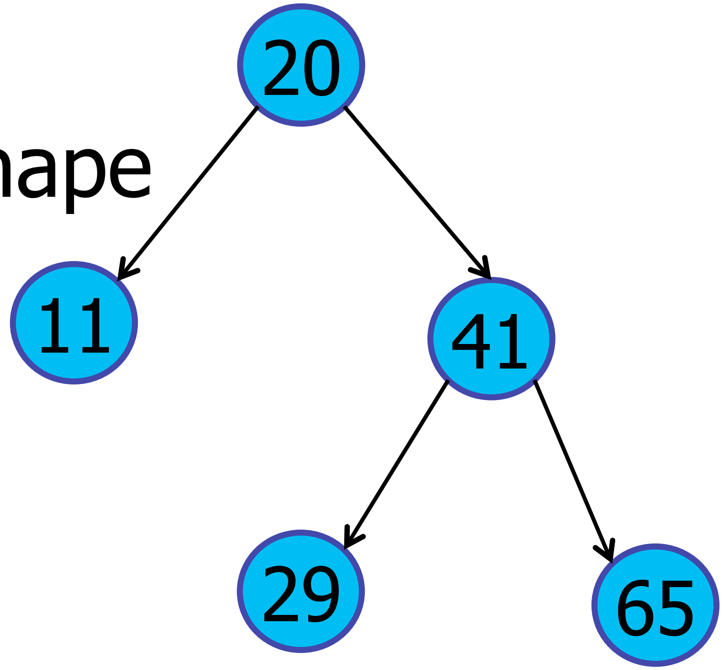
((())) ()(()) (()()) (())() ()()()

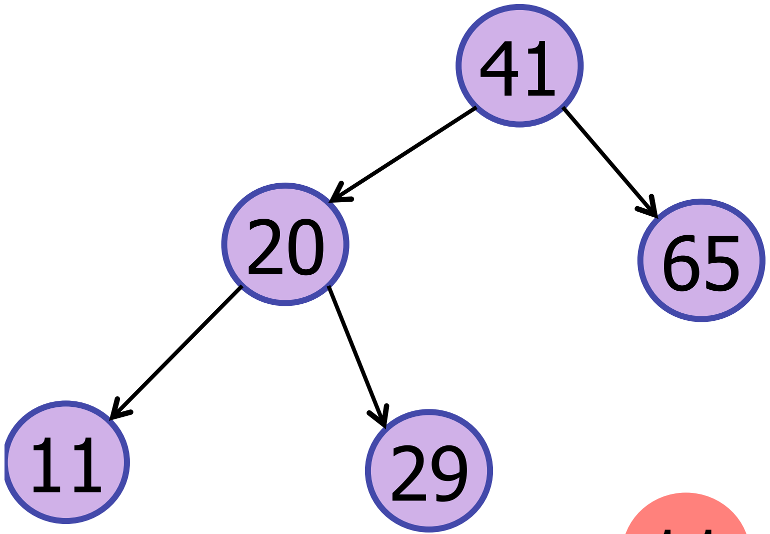
Why are these the same?

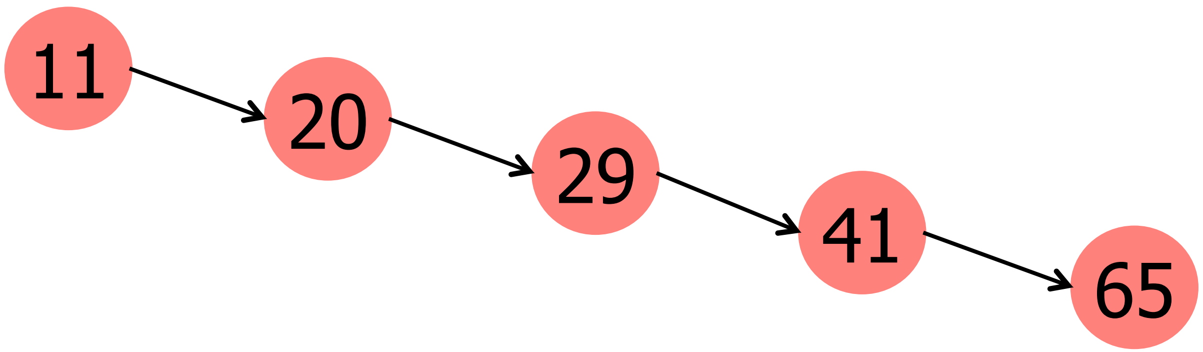
Tree Shape

Trees come in many shapes

- same keys ≠ same shape
- performance depends on shape



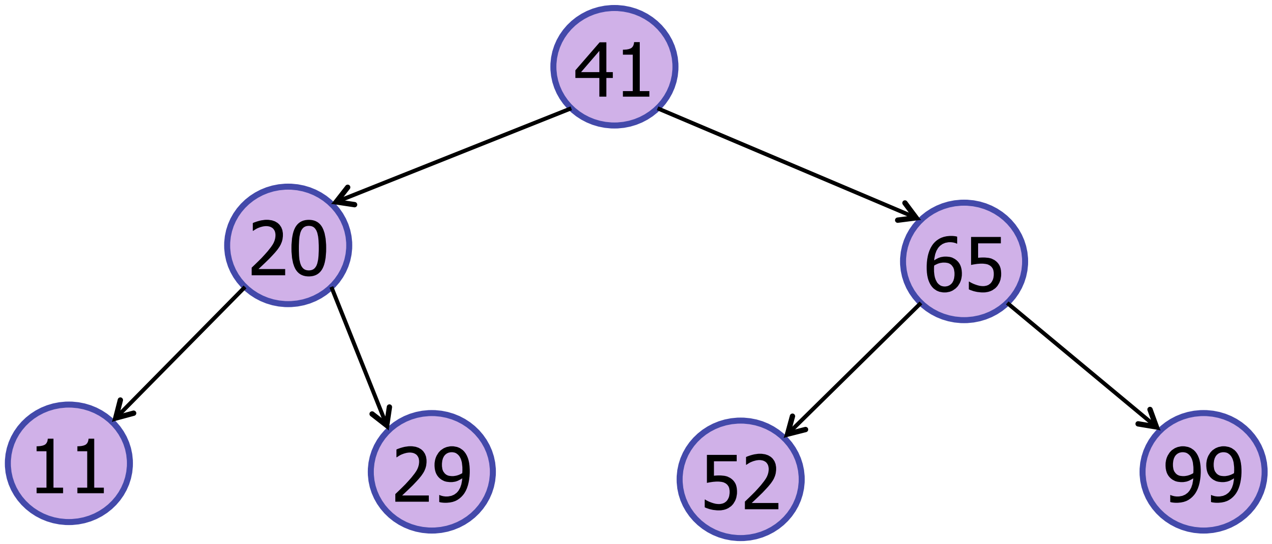




Tree Shape

Trees come in many shapes

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



Binary Search Trees

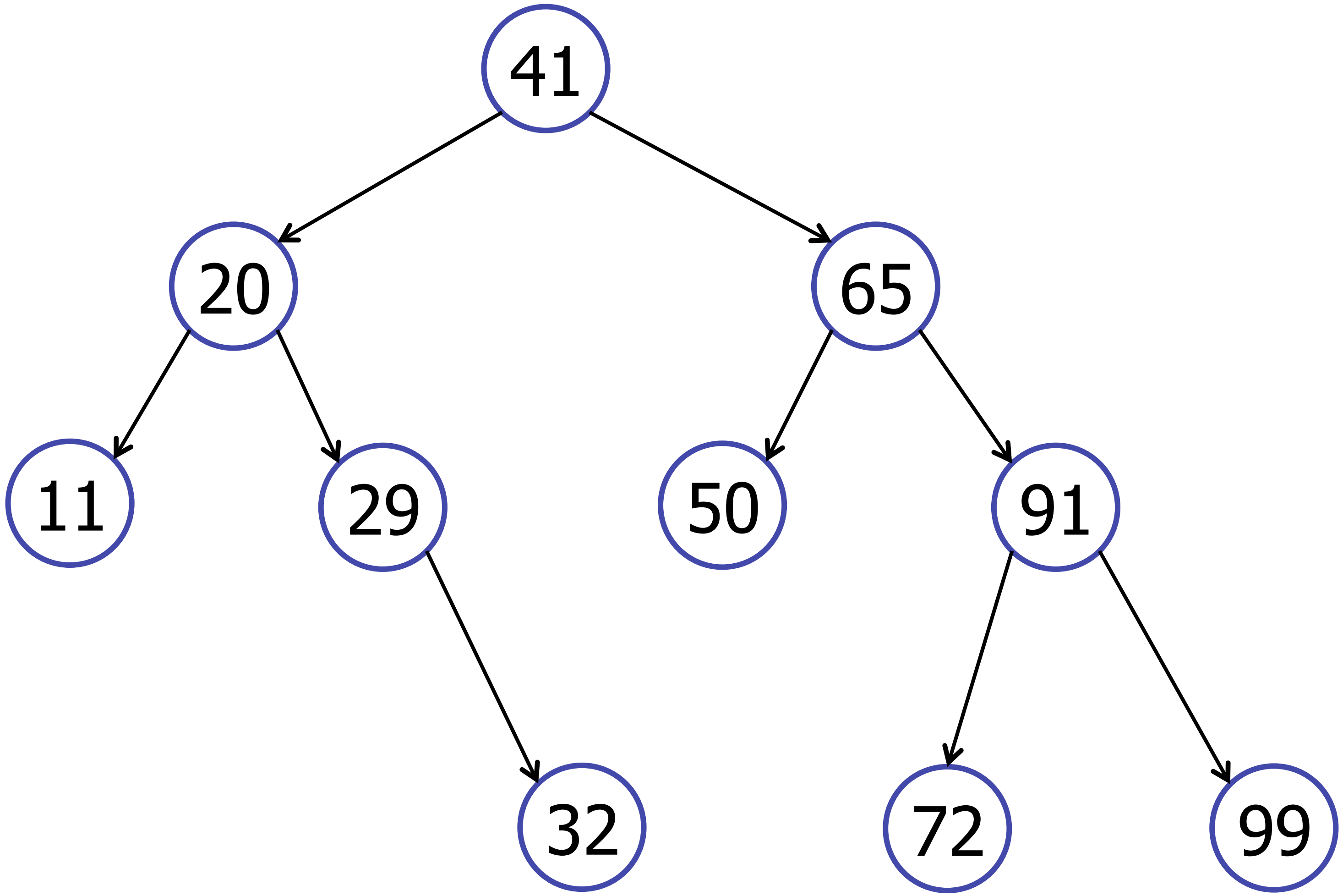
1. Terminology and Definitions

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

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

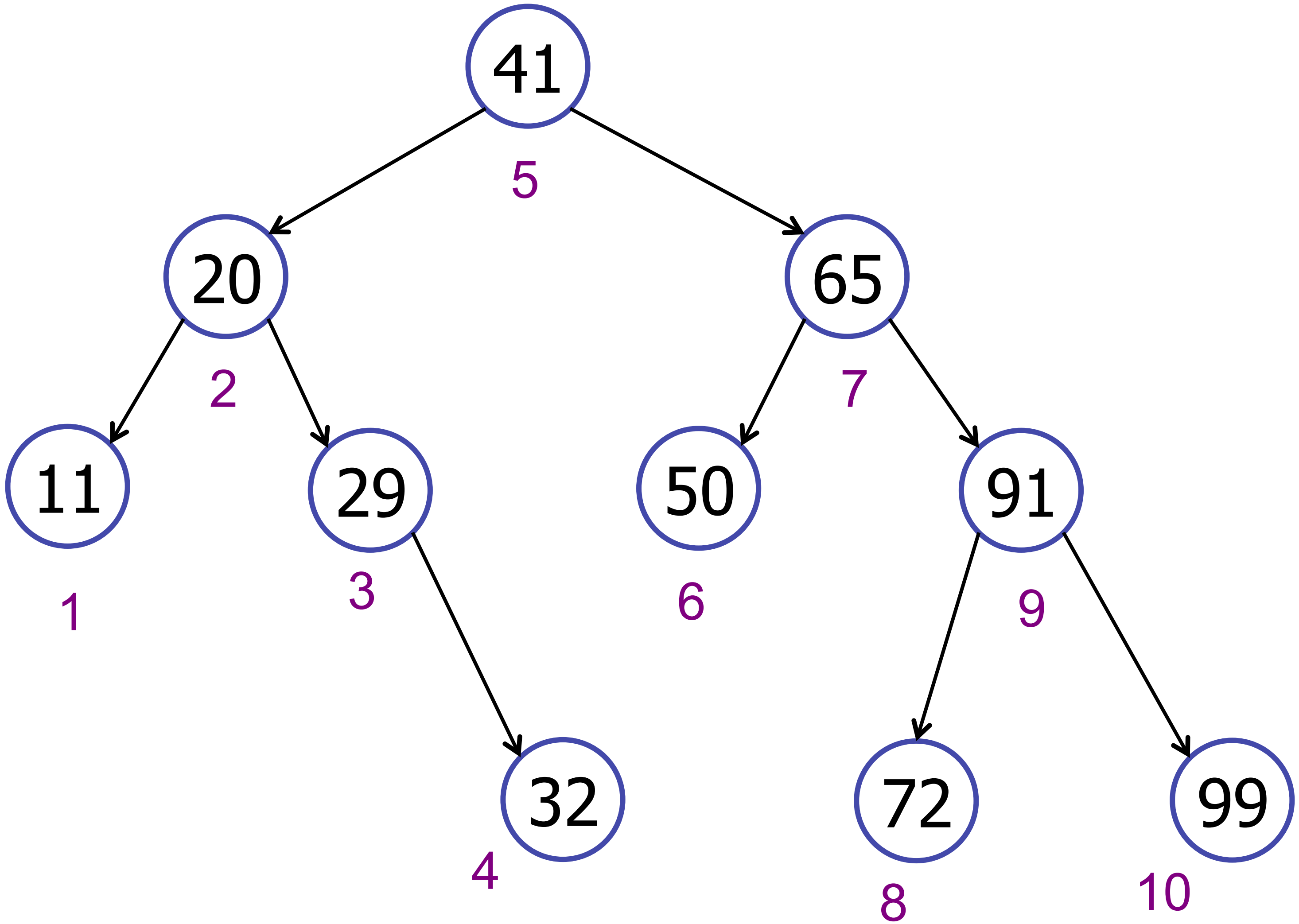
4. Other operations

Tree Traversal



11 20 29 32 41 50 65 72 91 99

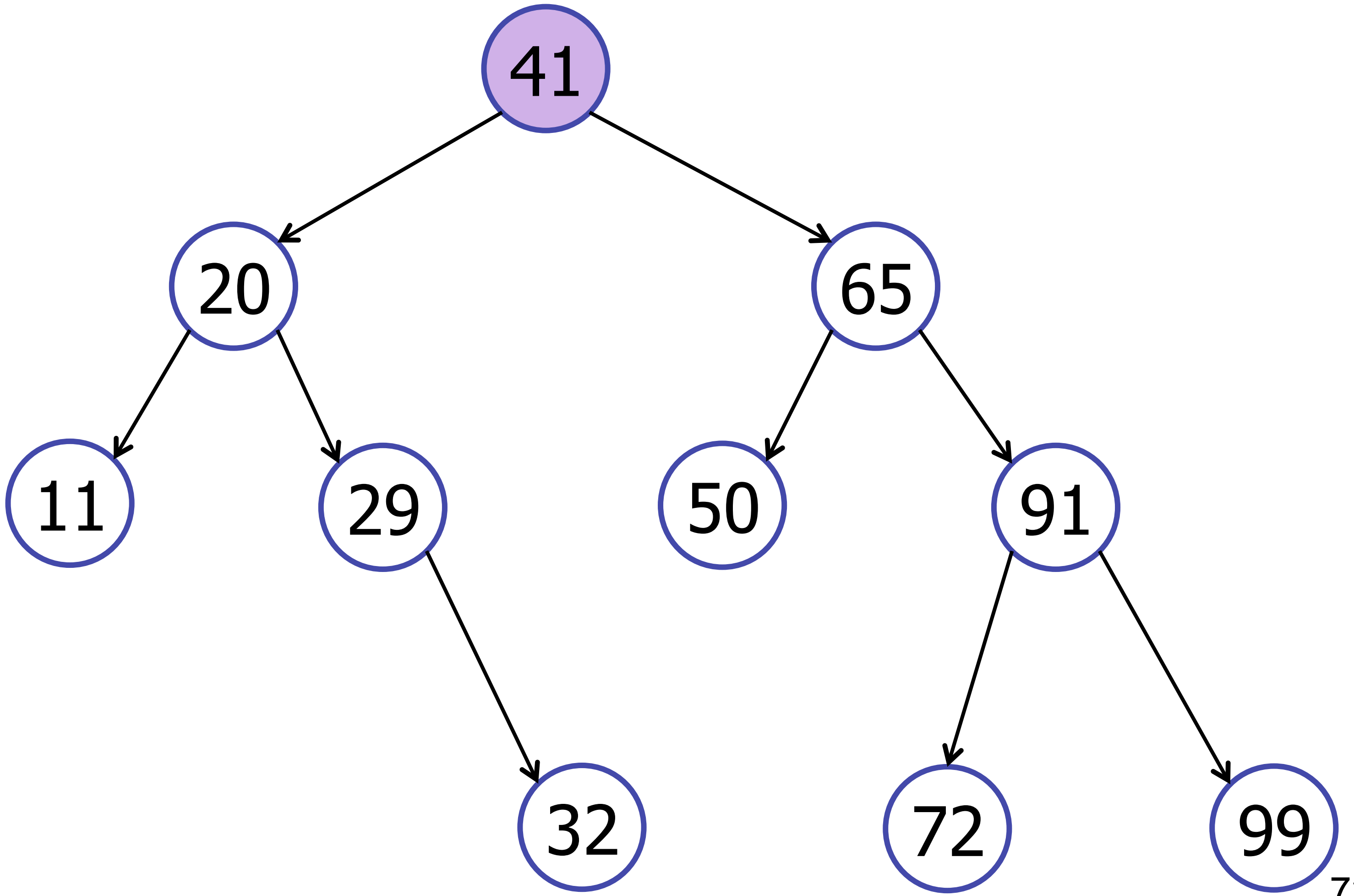
Tree Traversal



11 20 29 32 41 50 65 72 91 99

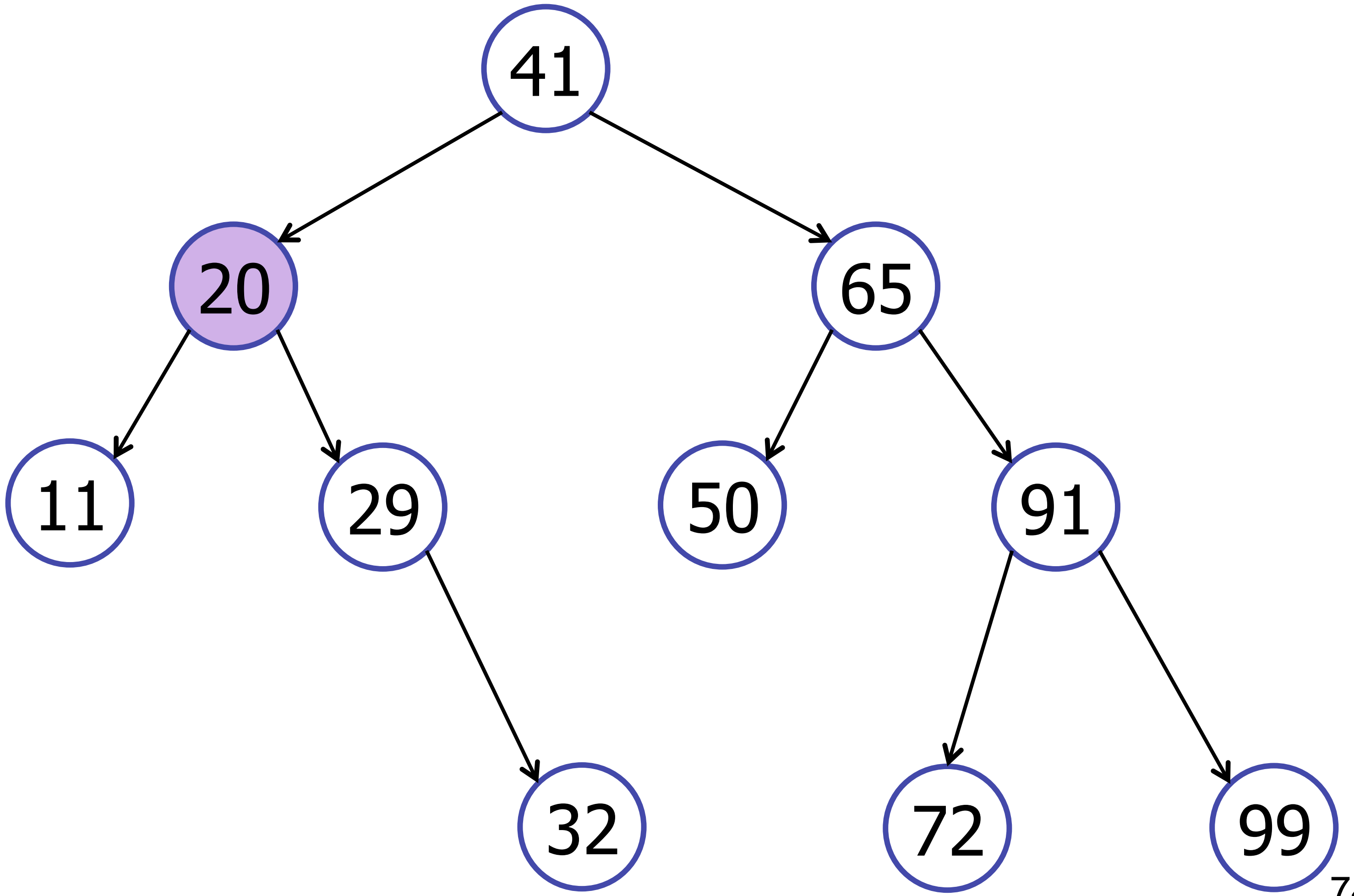
Tree Traversal

in-order-traversal



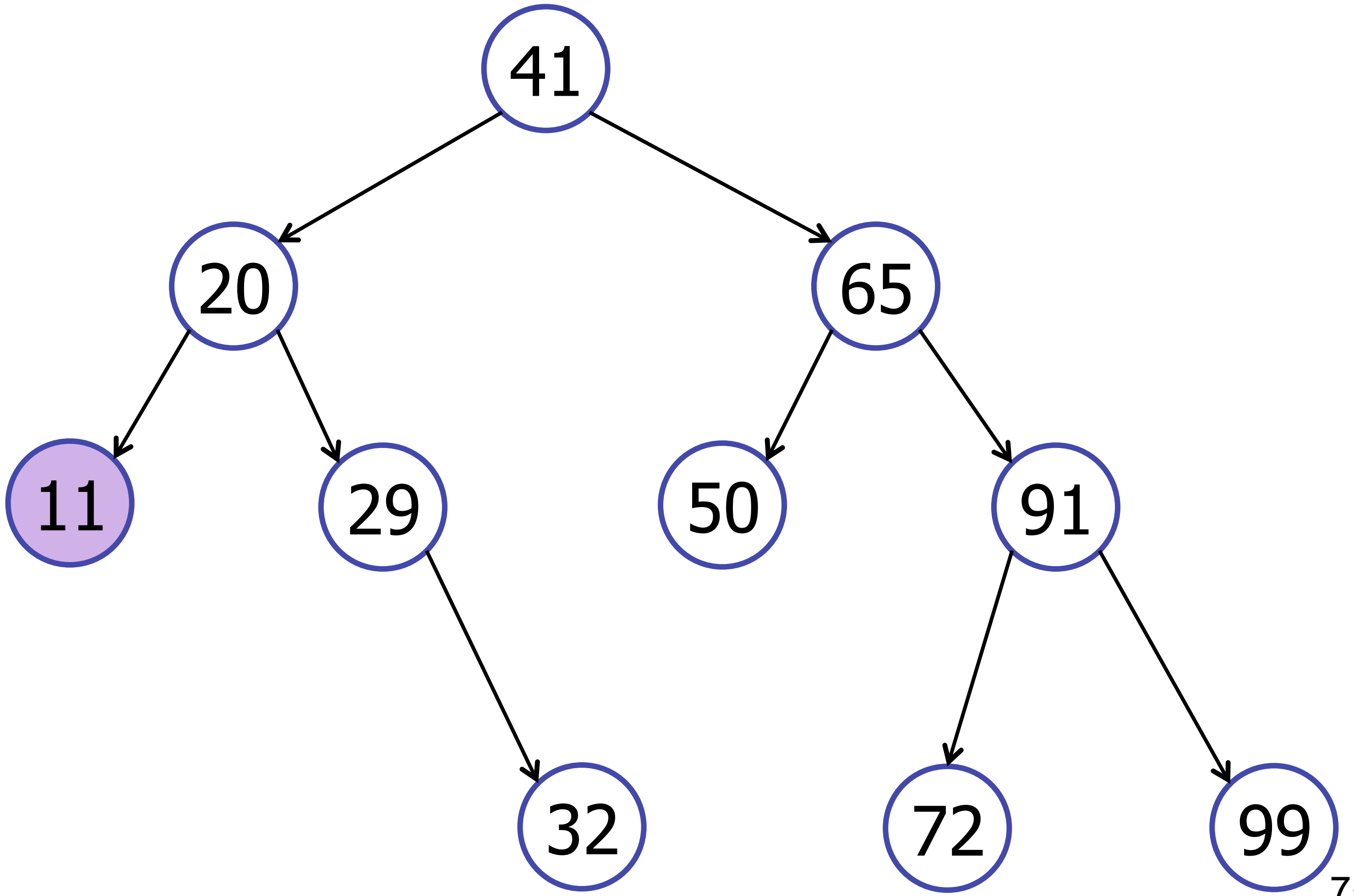
Tree Traversal

in-order-traversal



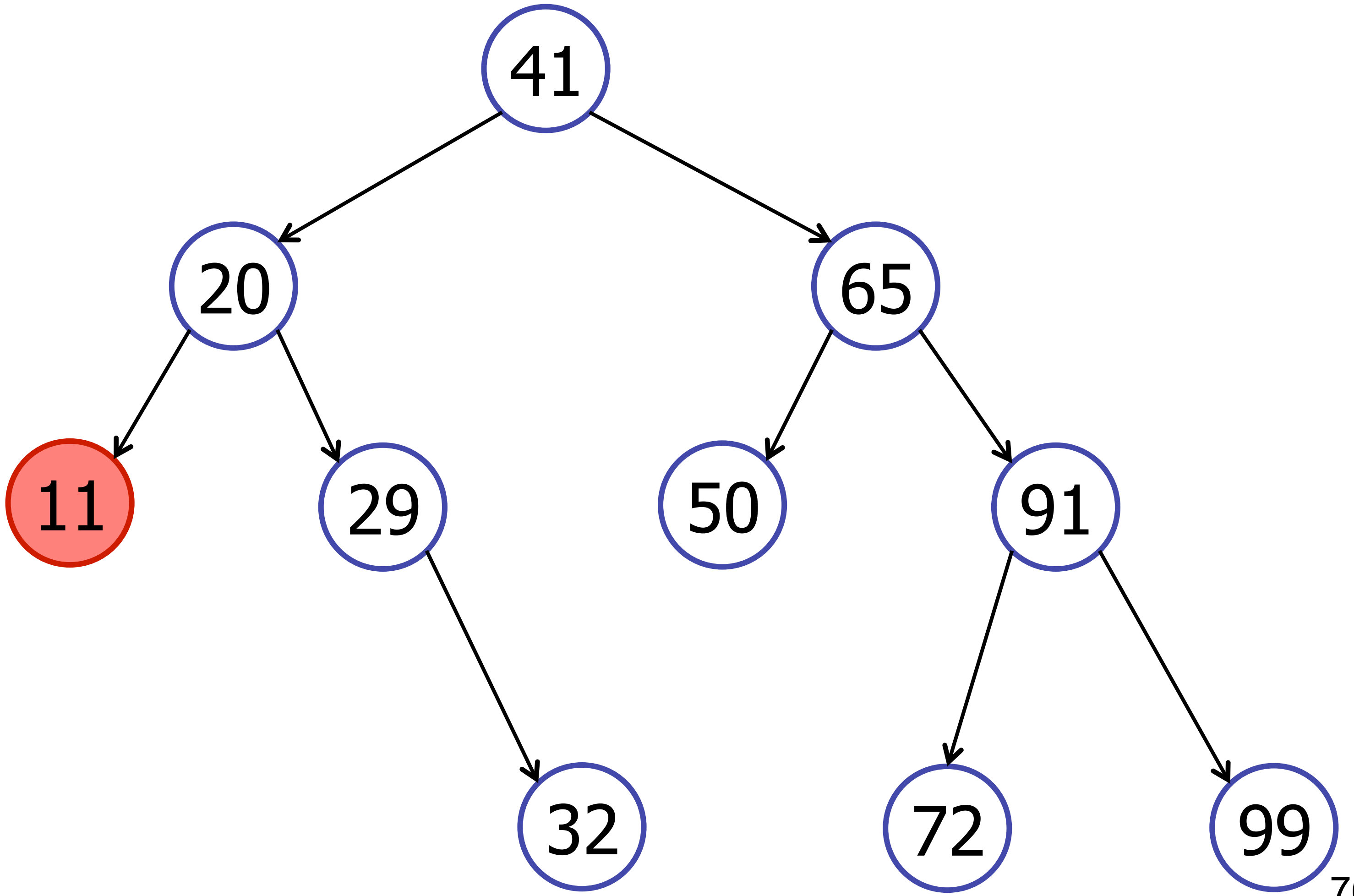
Tree Traversal

in-order-traversal



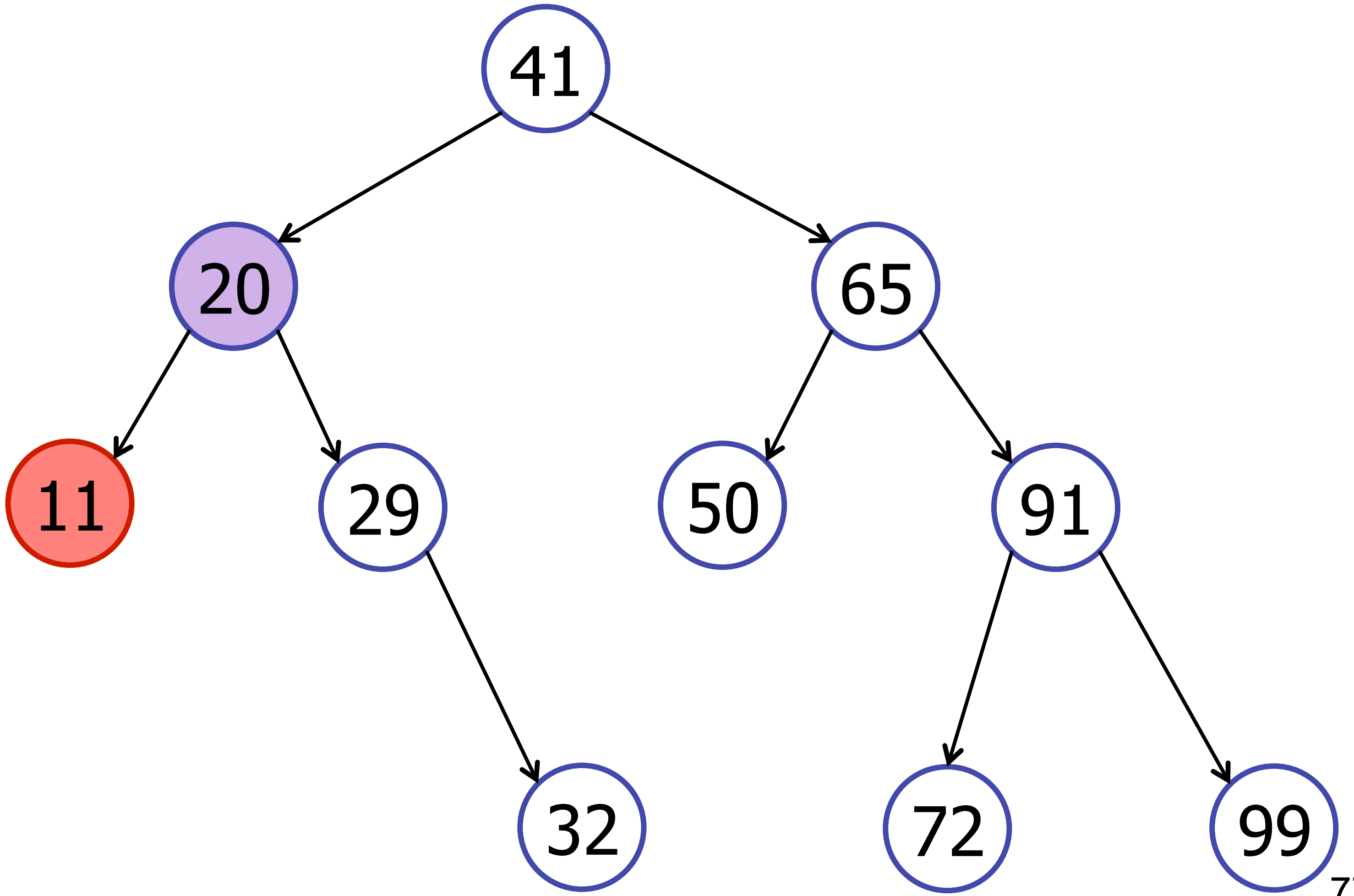
Tree Traversal

in-order-traversal



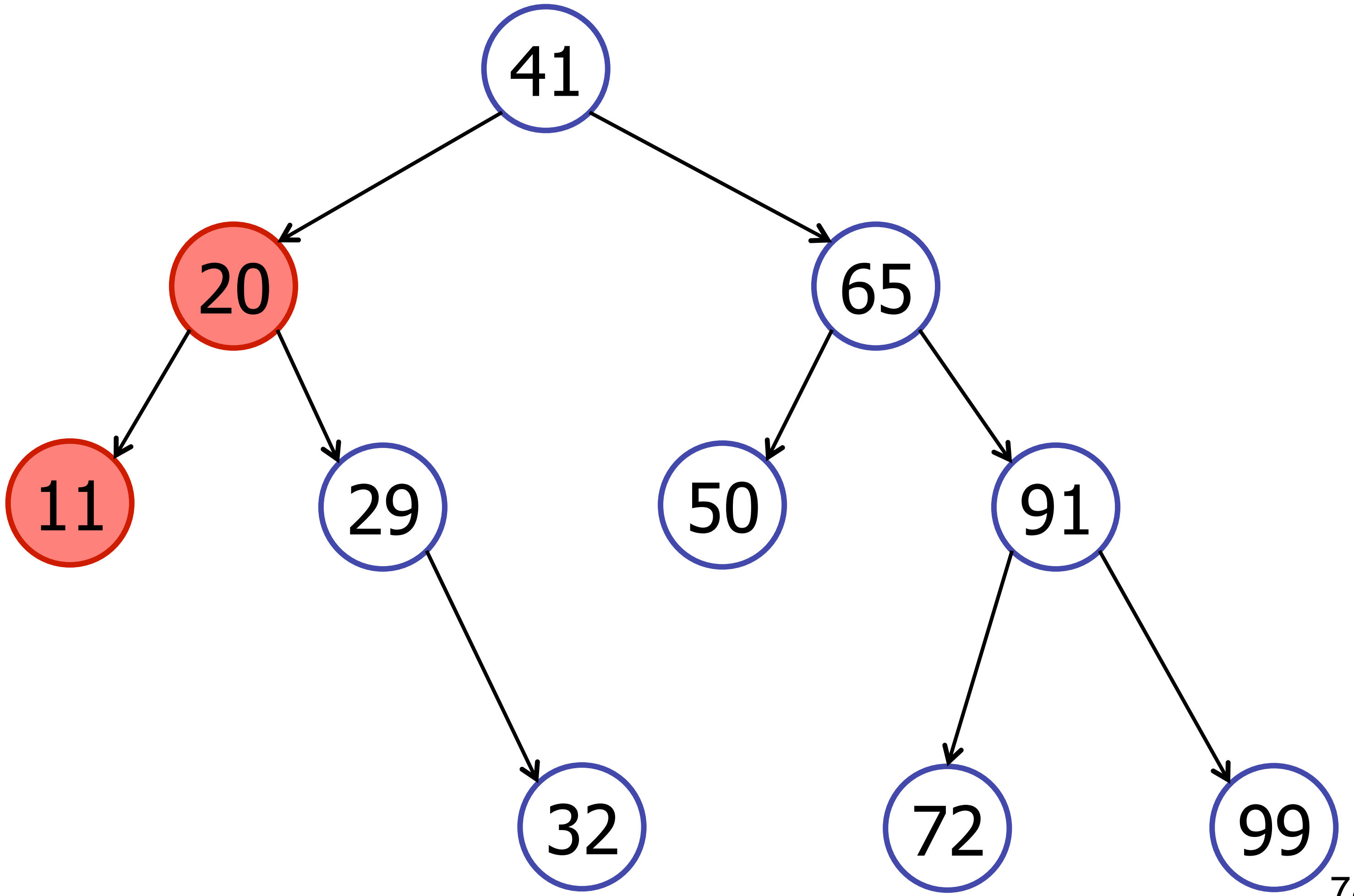
Tree Traversal

in-order-traversal



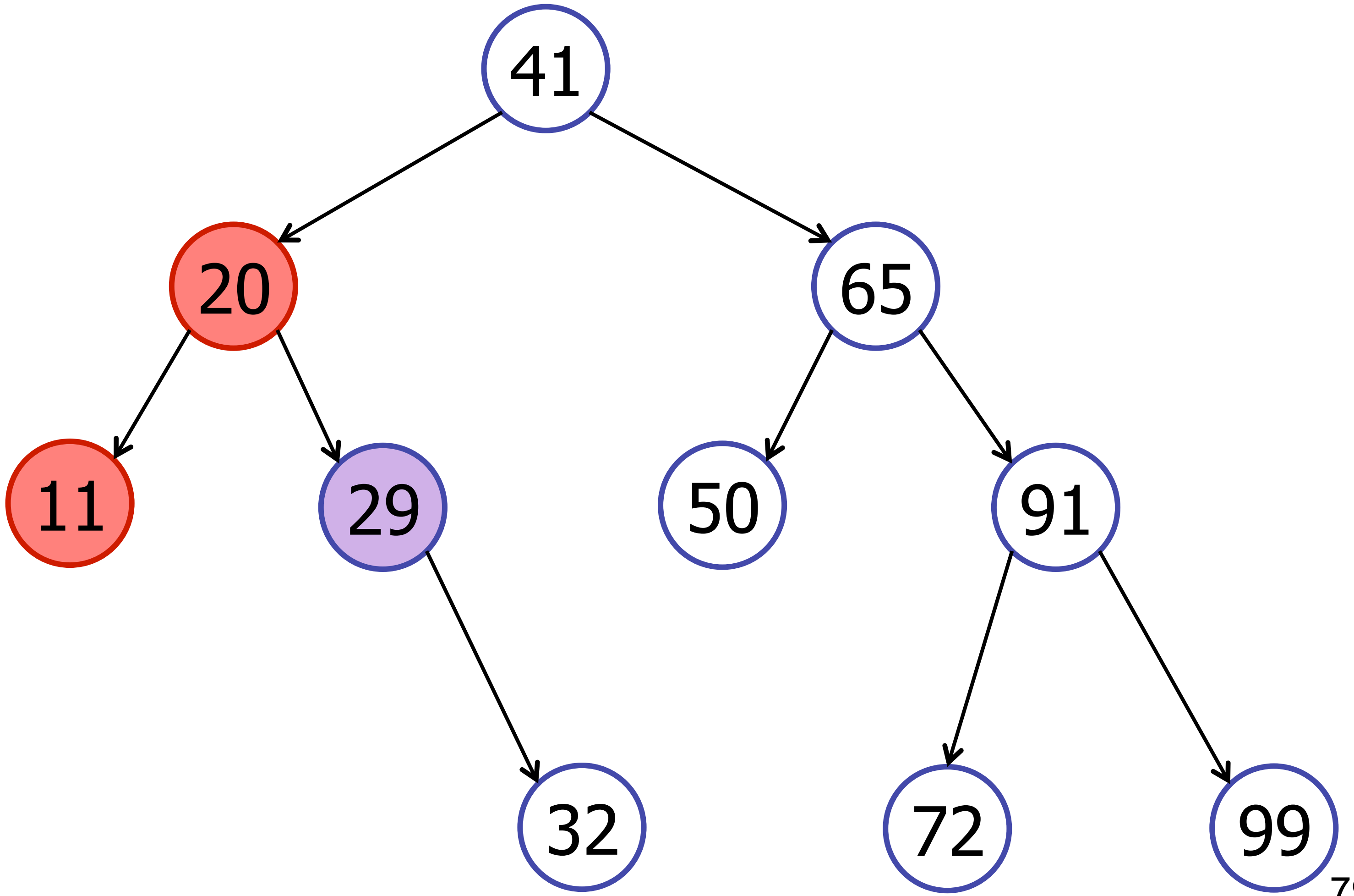
Tree Traversal

in-order-traversal



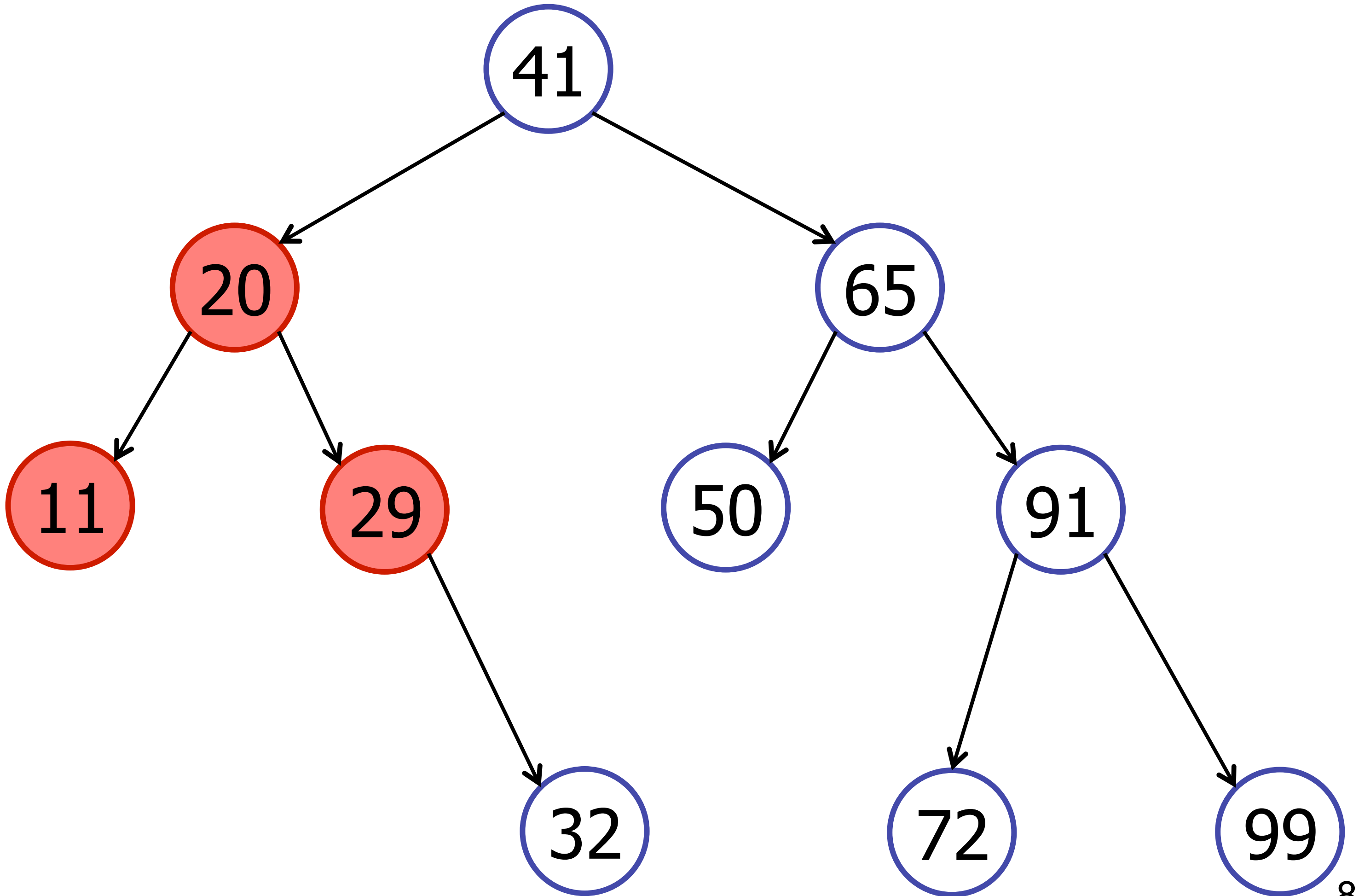
Tree Traversal

in-order-traversal



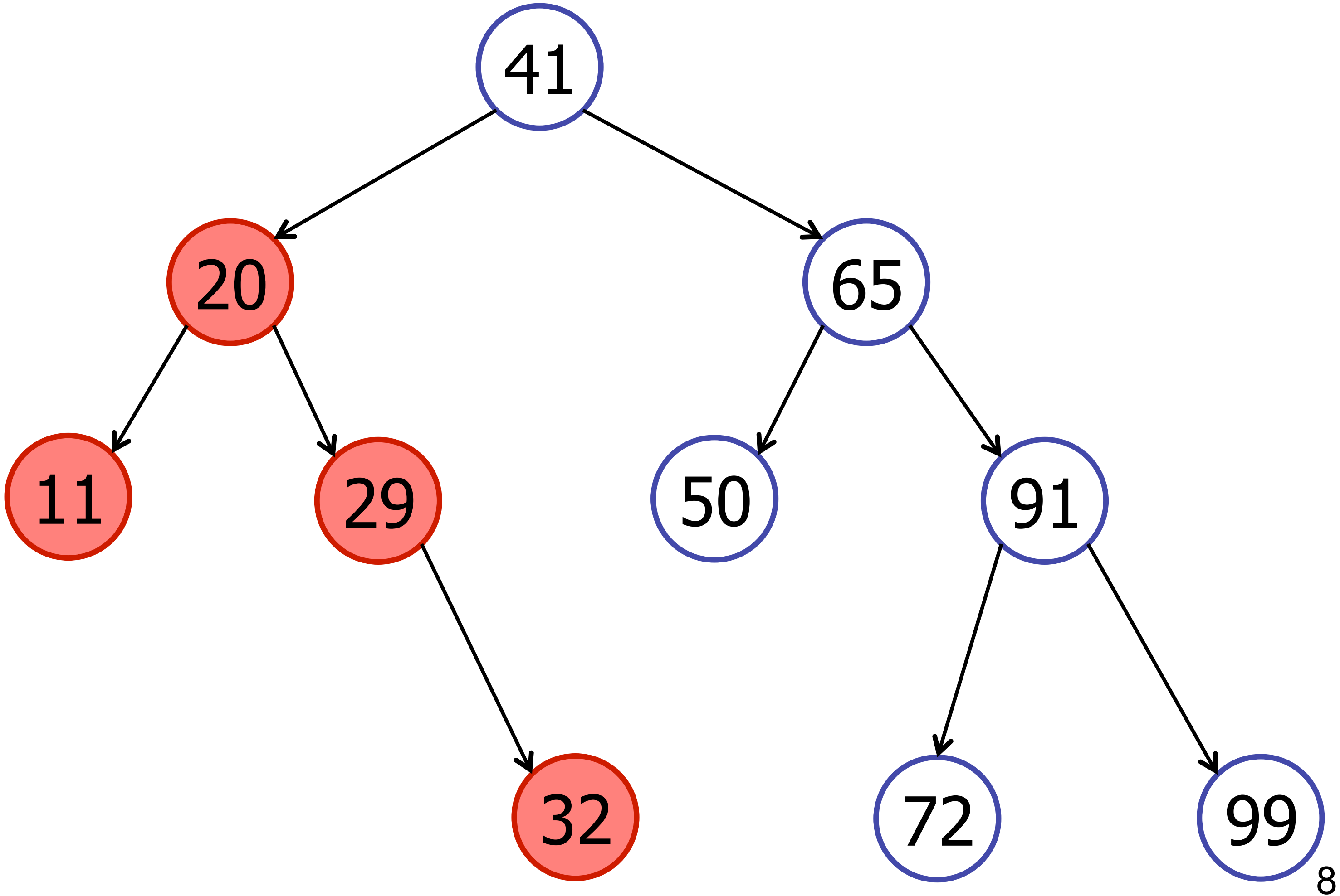
Tree Traversal

in-order-traversal



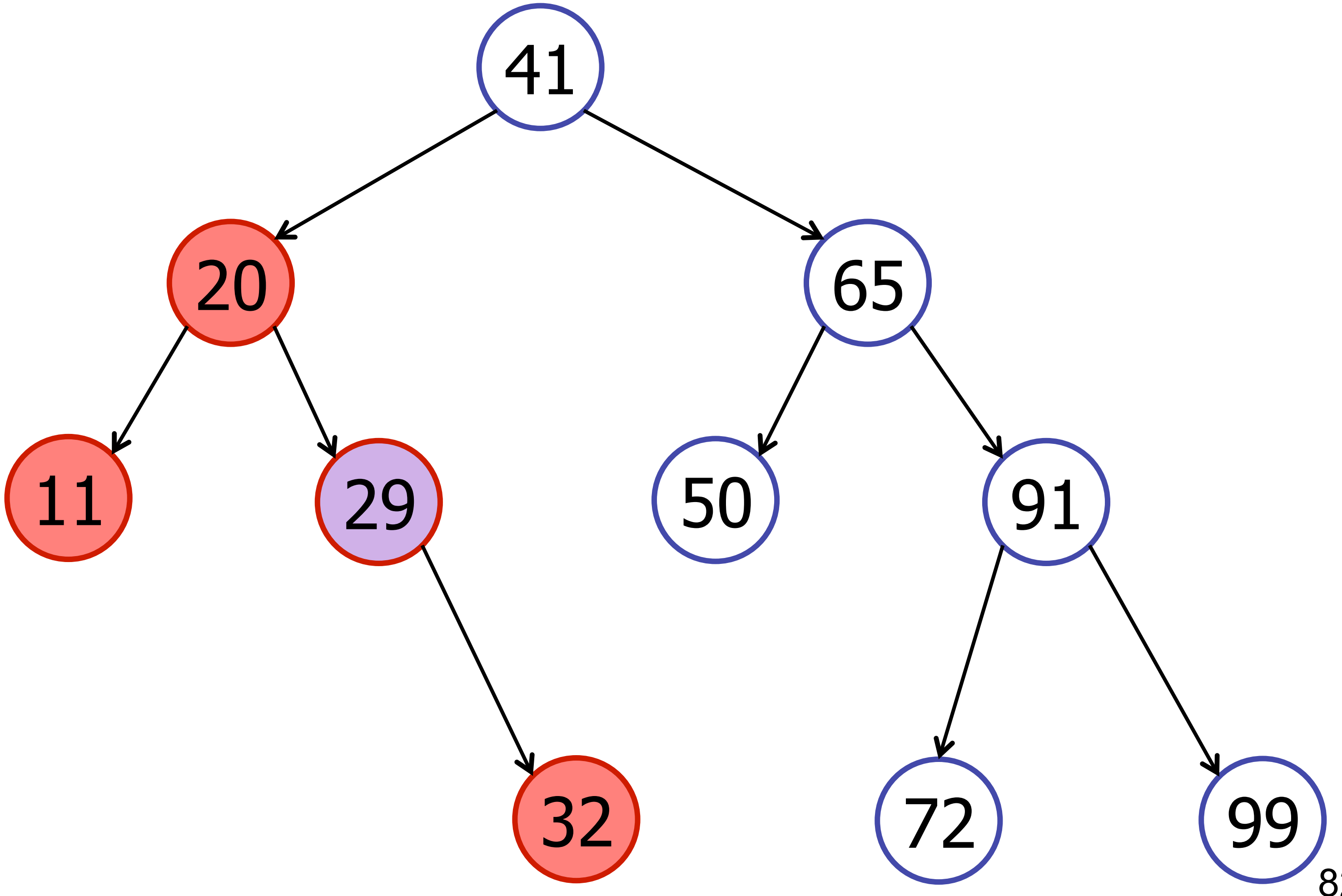
Tree Traversal

in-order-traversal



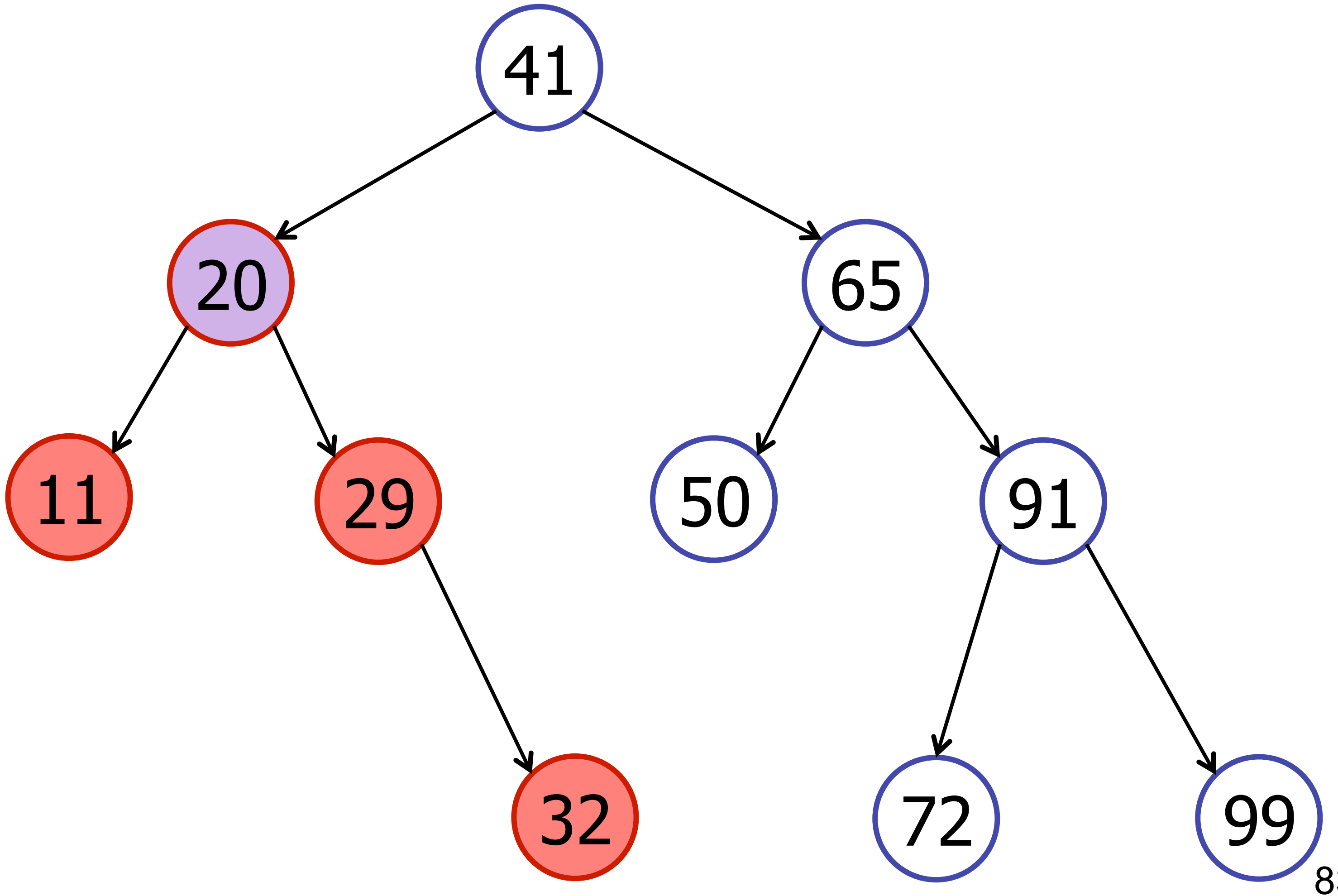
Tree Traversal

in-order-traversal



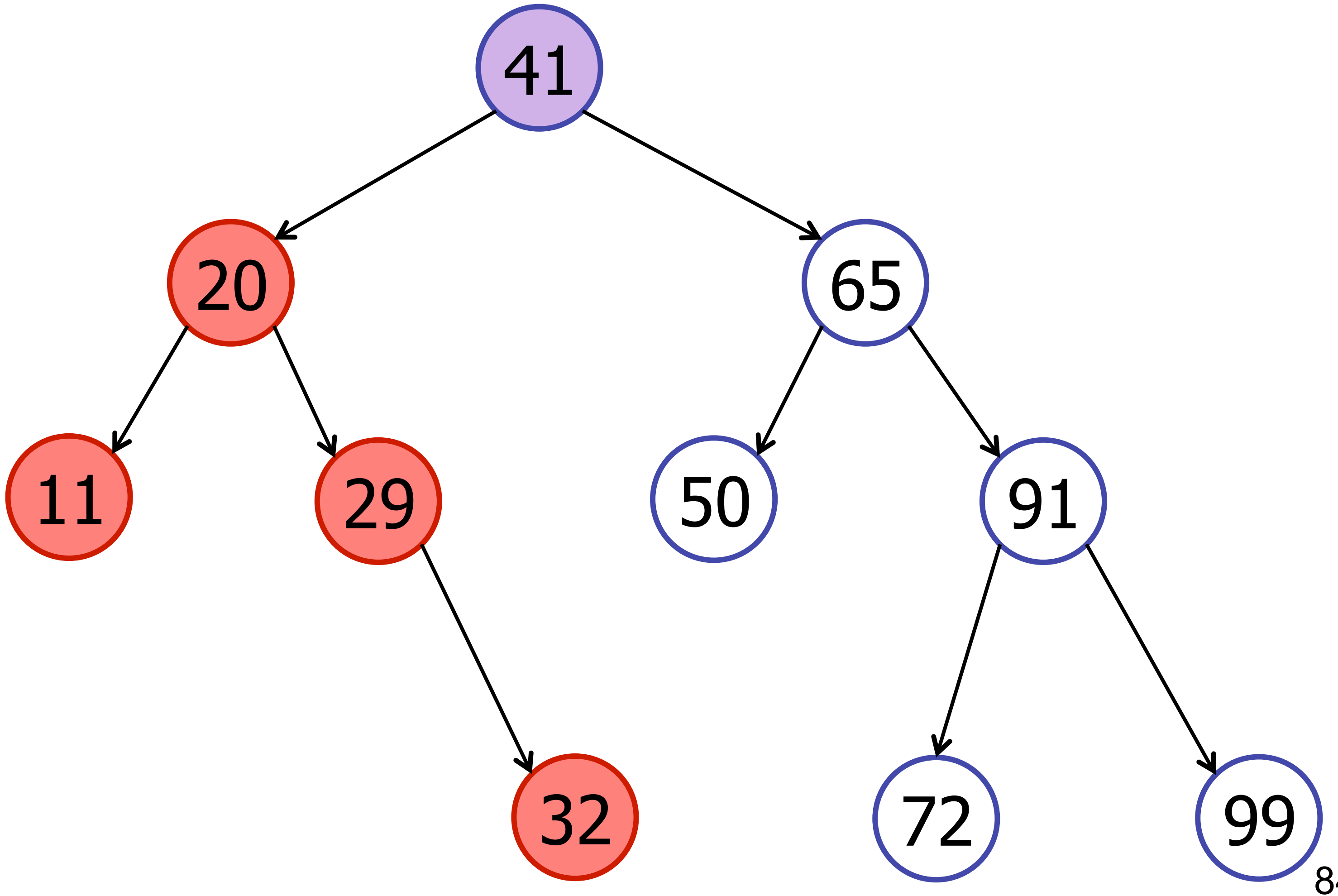
Tree Traversal

in-order-traversal



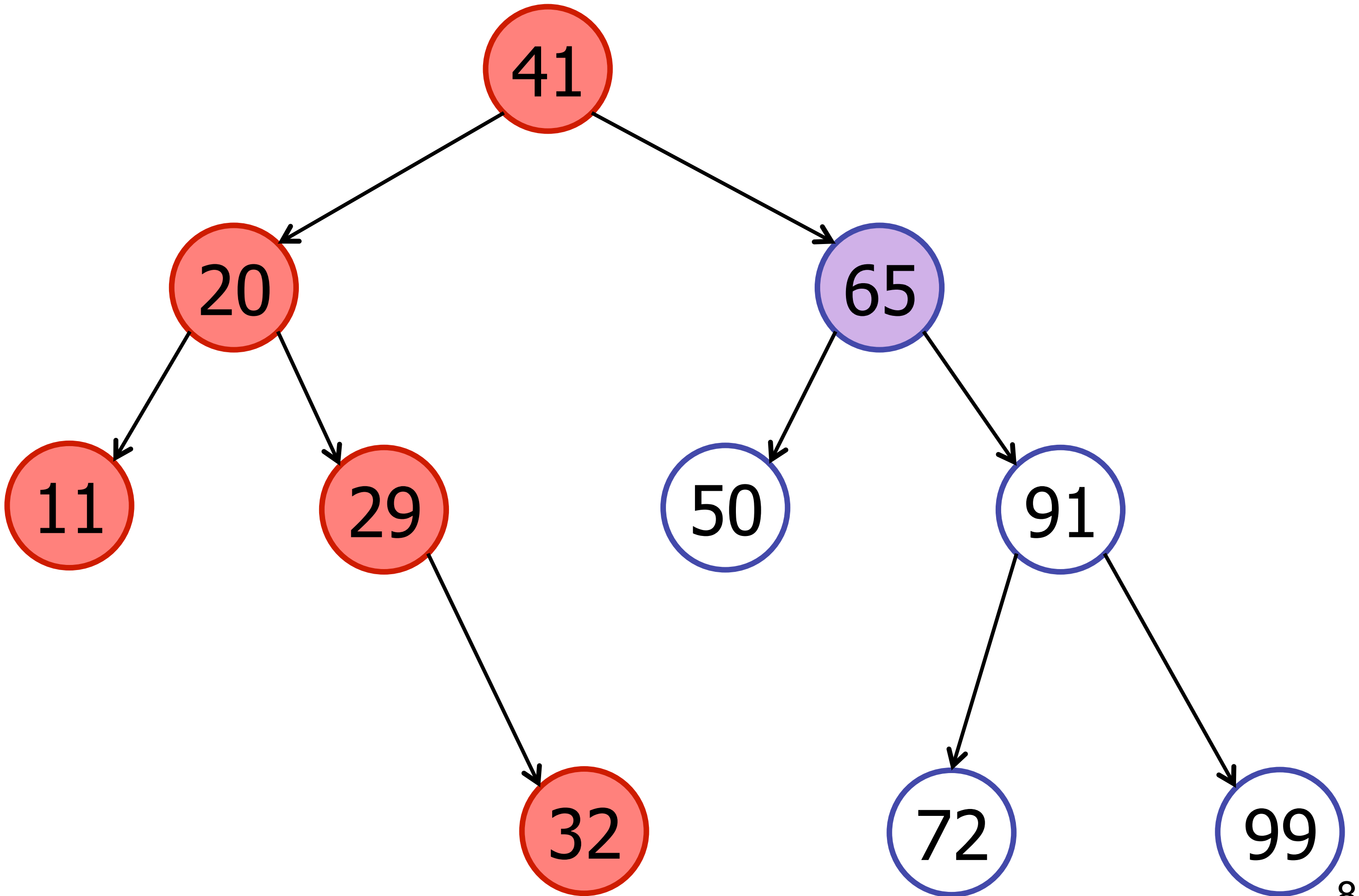
Tree Traversal

in-order-traversal



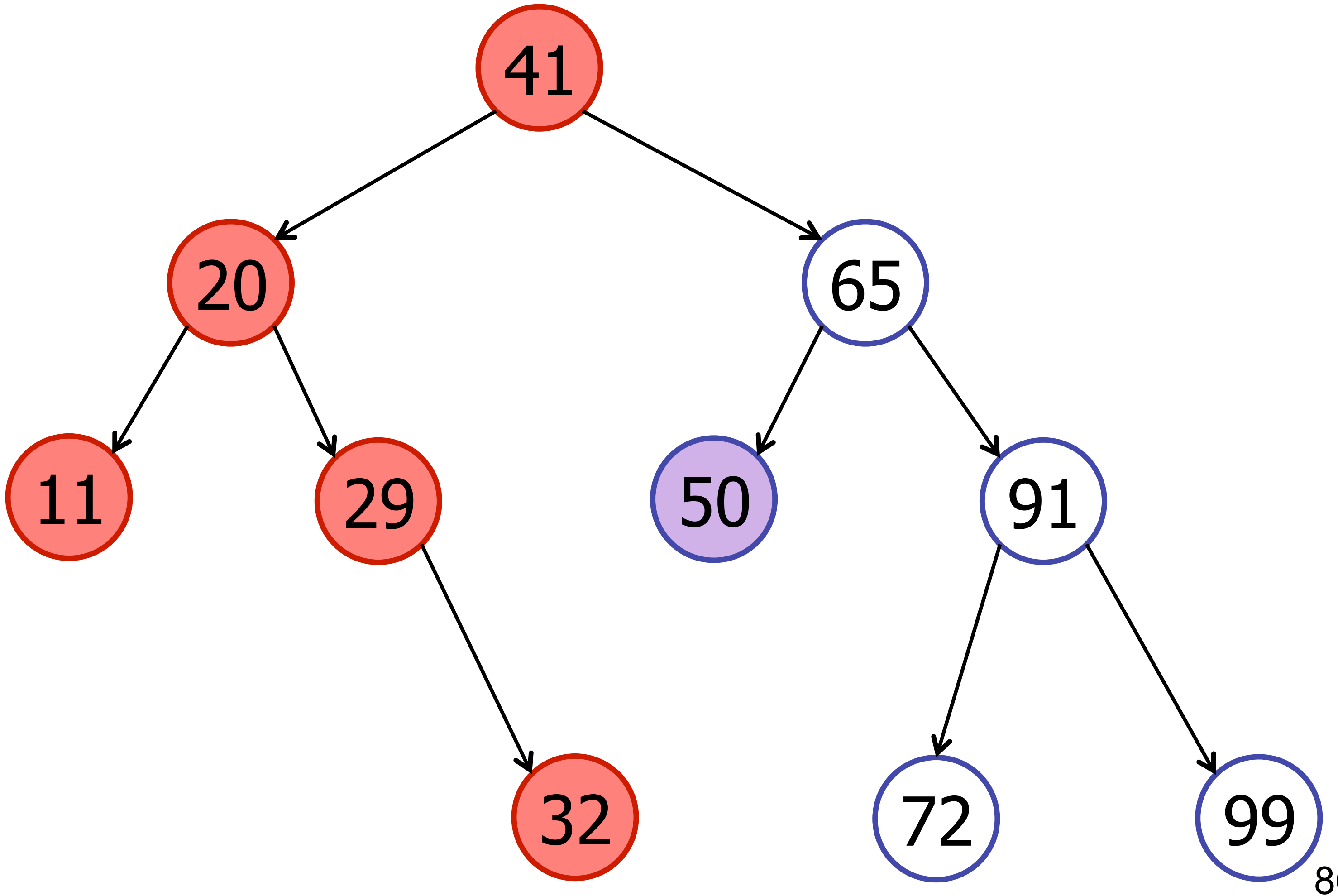
Tree Traversal

in-order-traversal



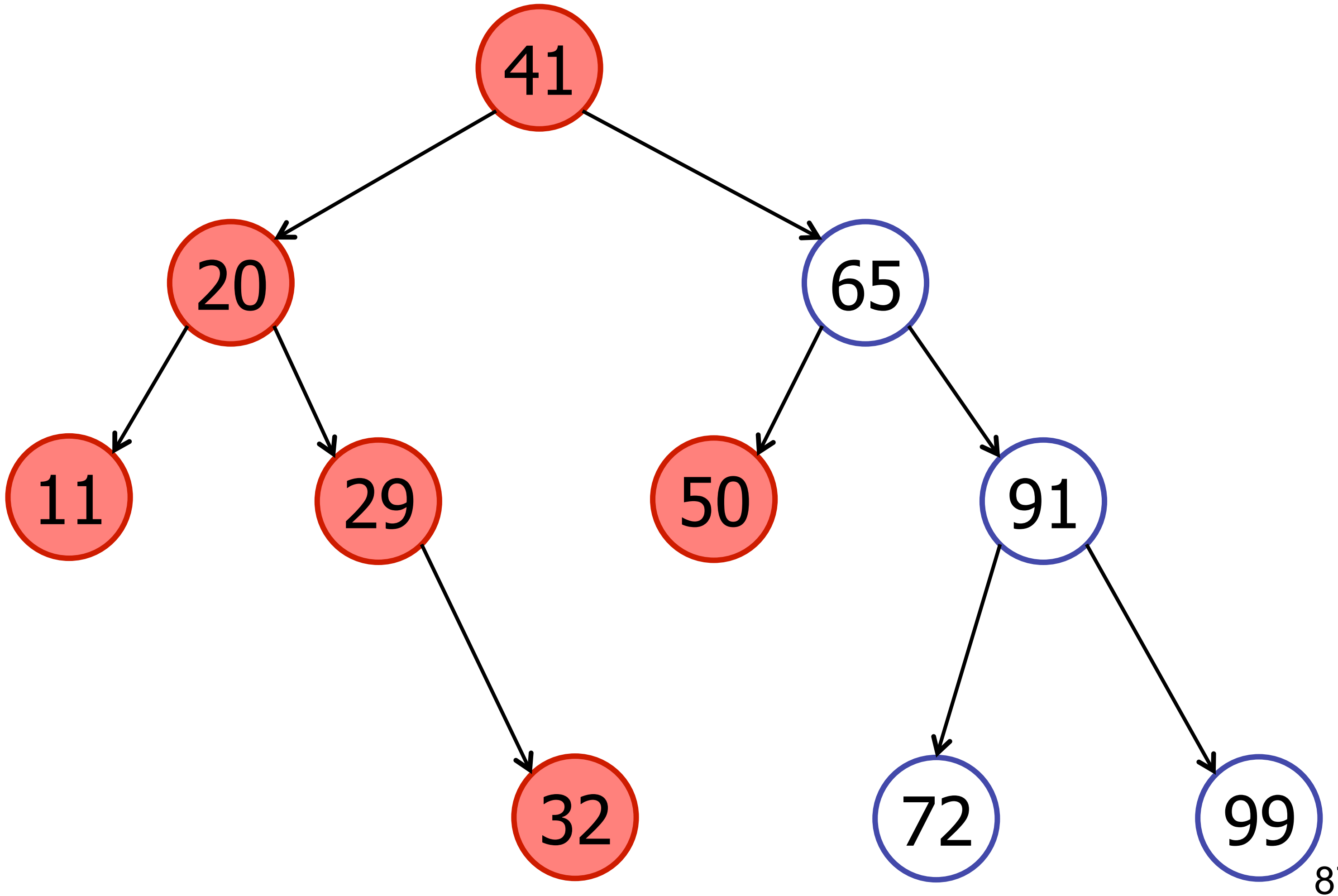
Tree Traversal

in-order-traversal



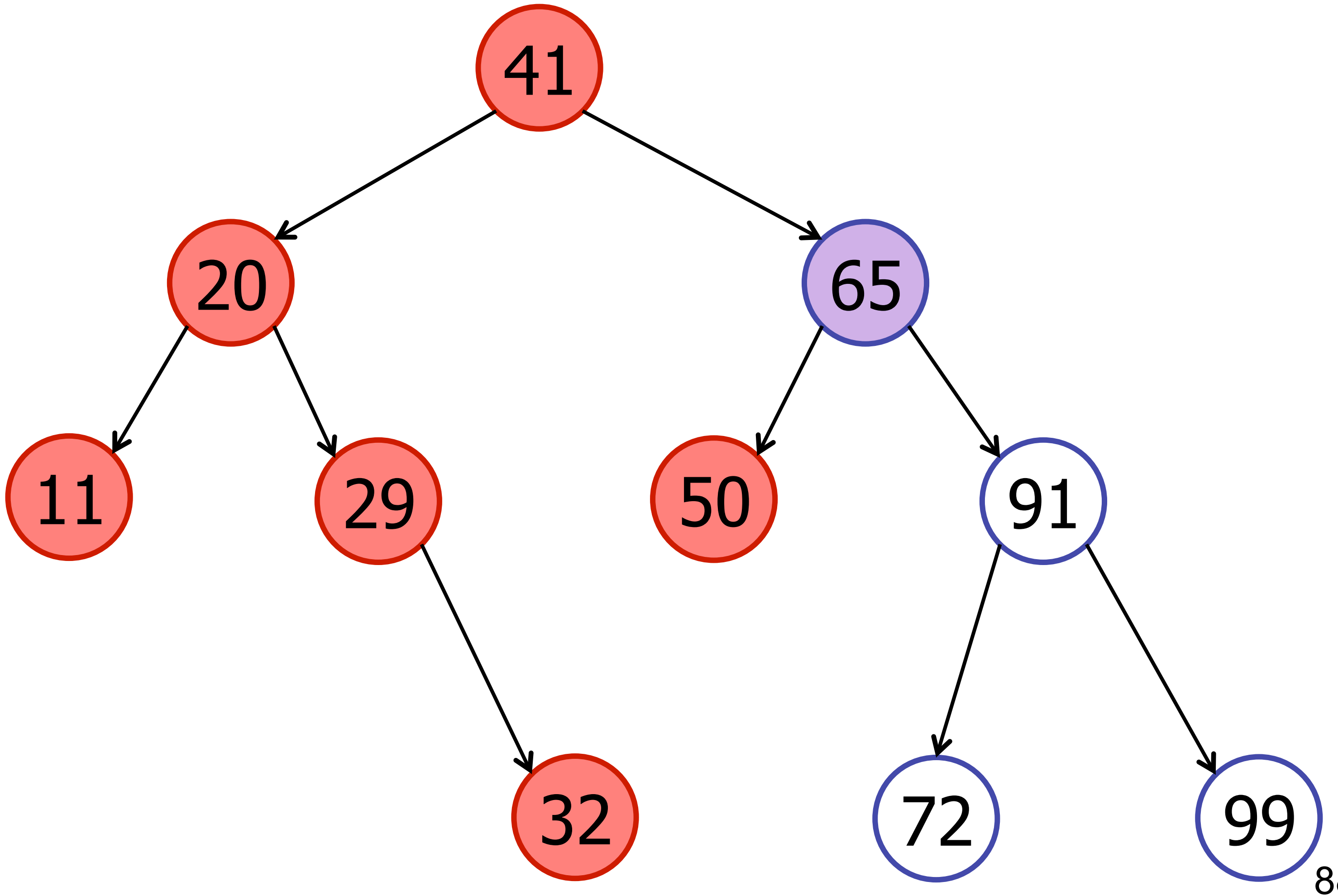
Tree Traversal

in-order-traversal



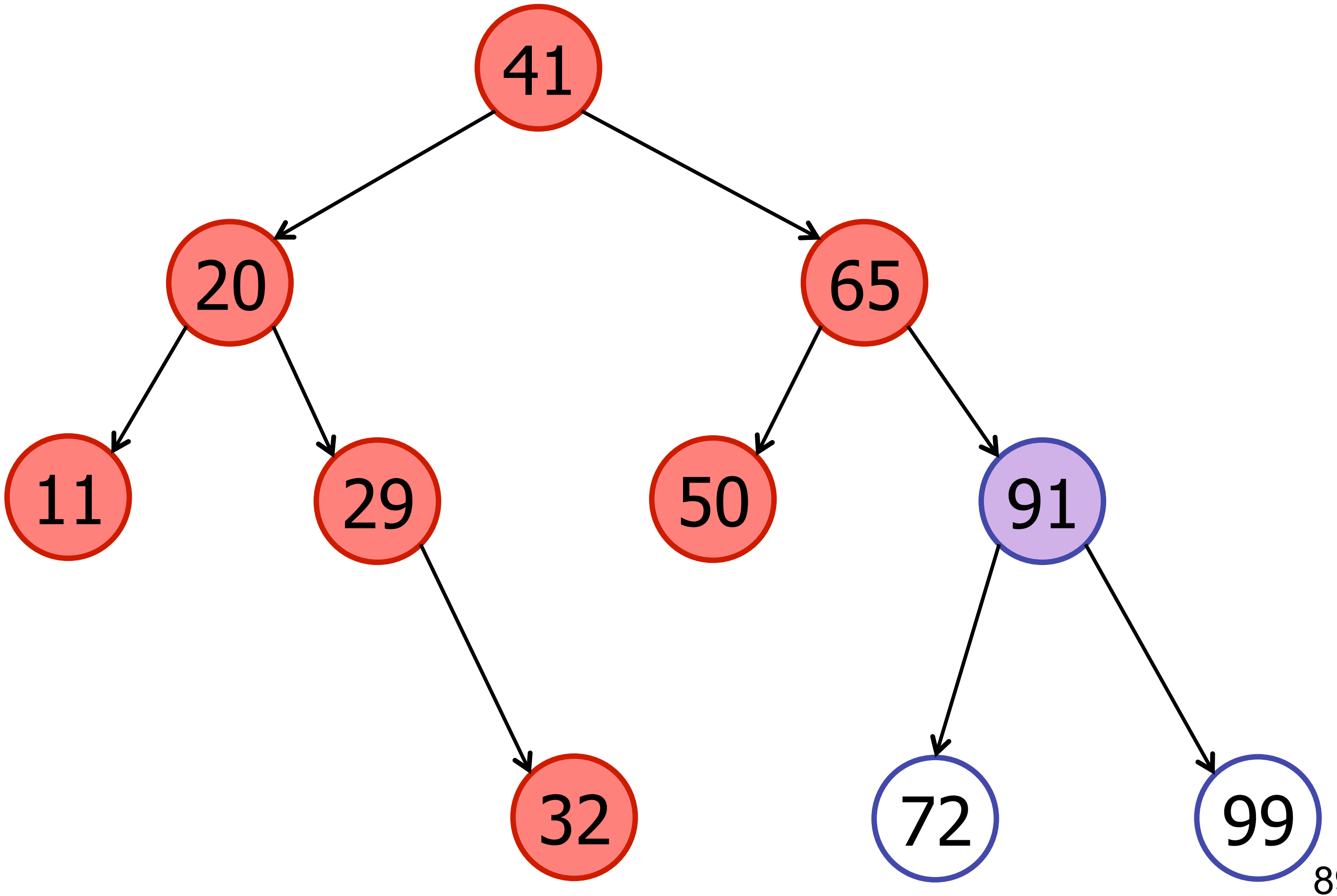
Tree Traversal

in-order-traversal



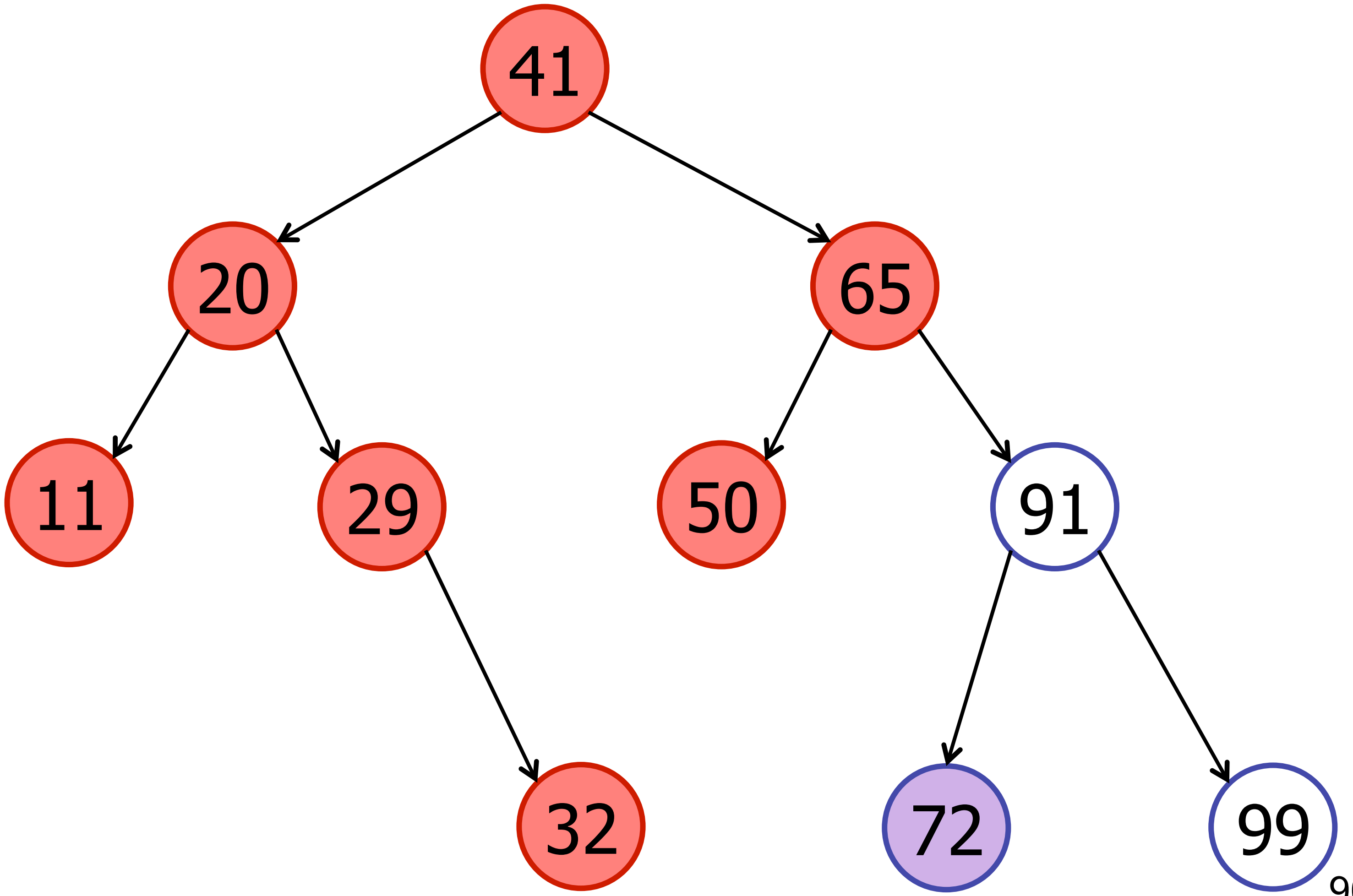
Tree Traversal

in-order-traversal



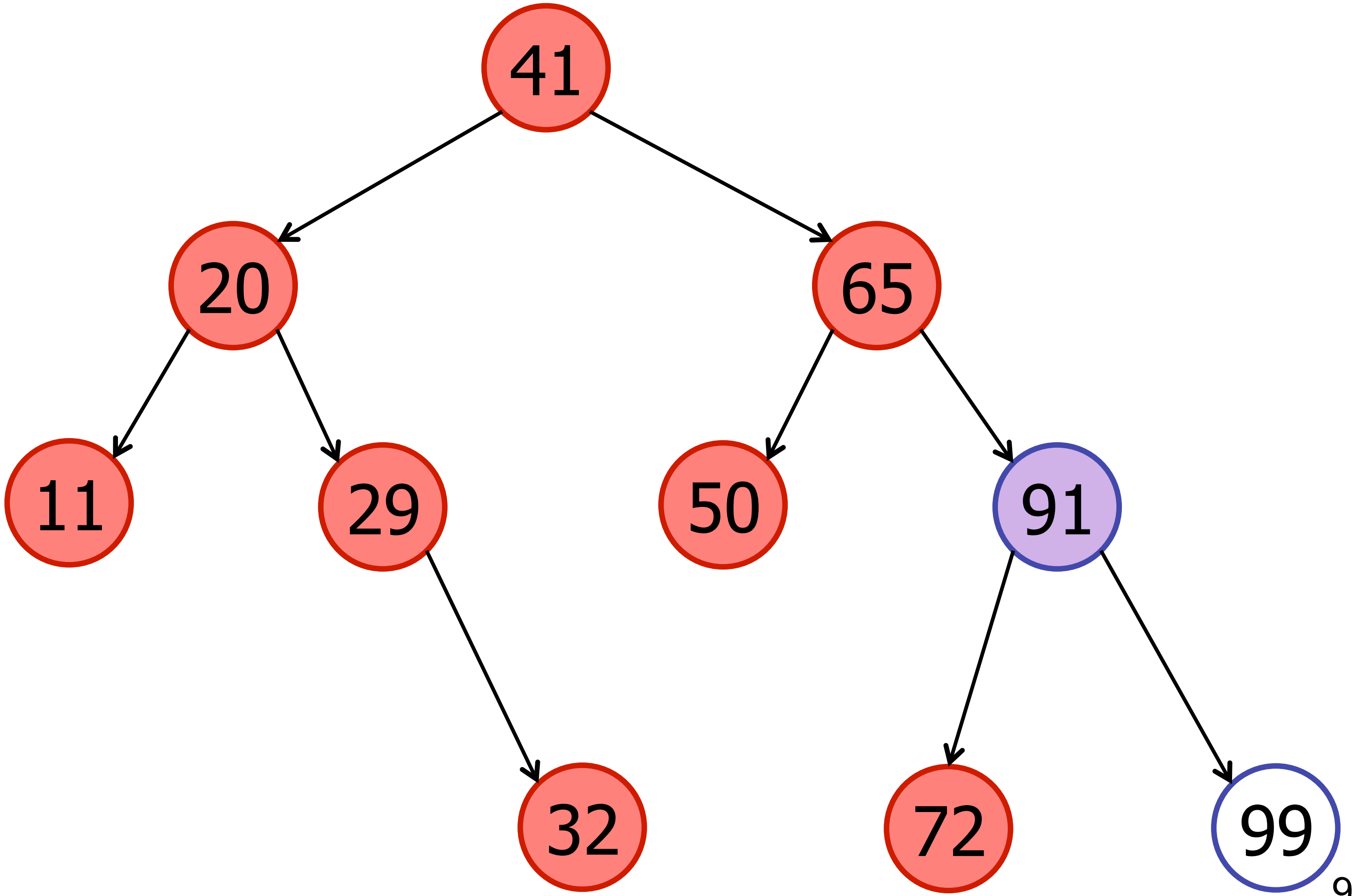
Tree Traversal

in-order-traversal



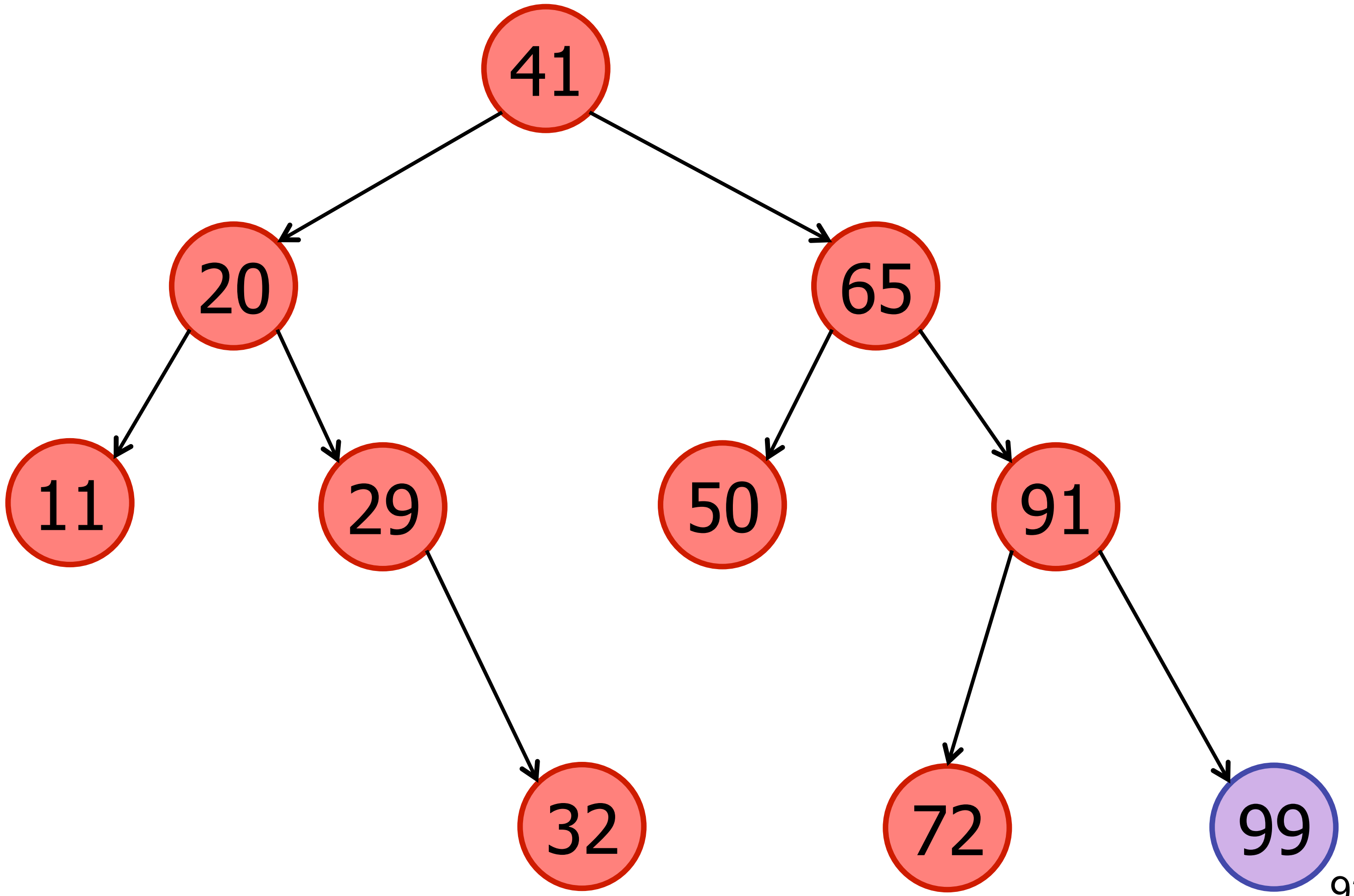
Tree Traversal

in-order-traversal



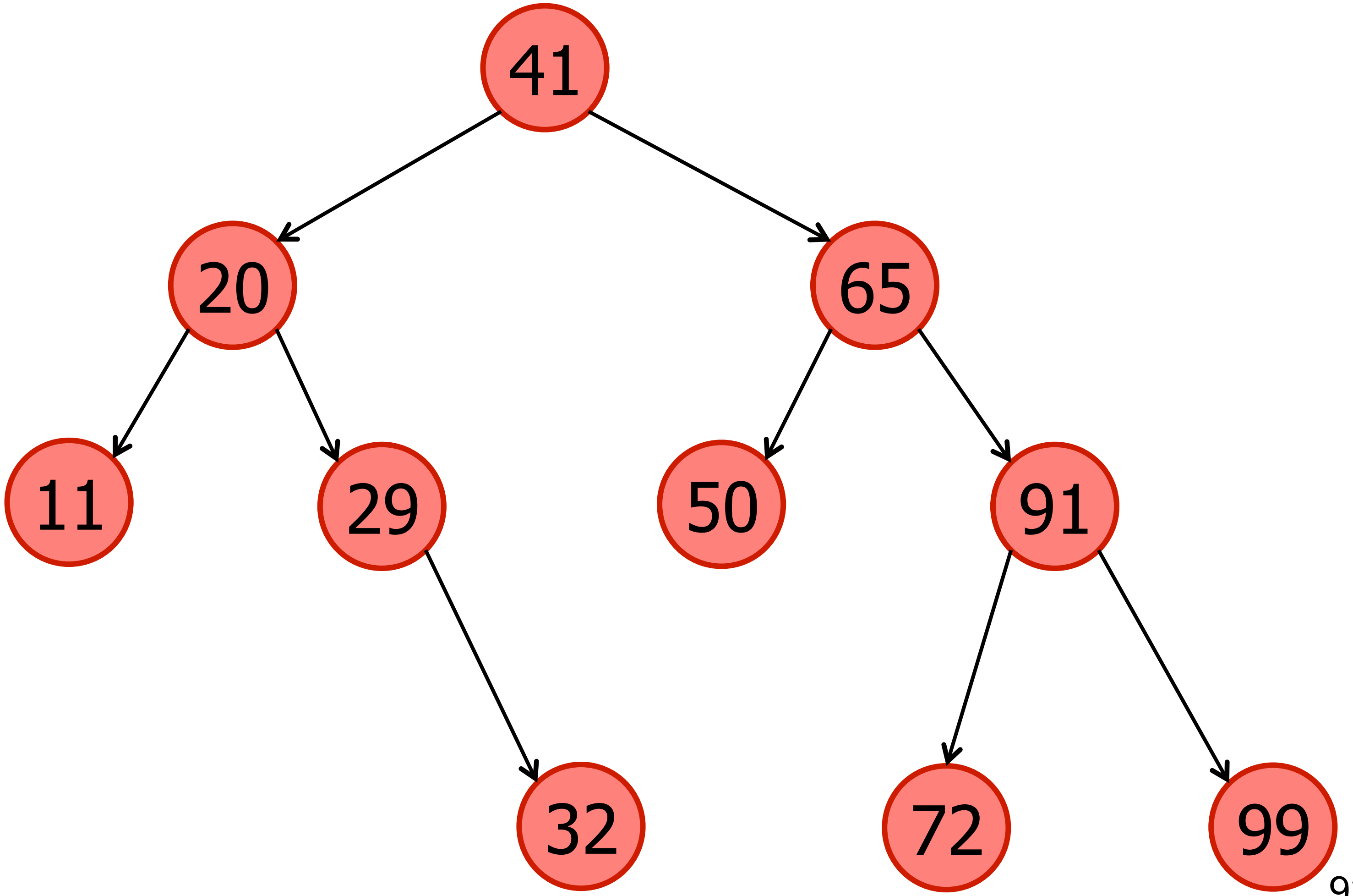
Tree Traversal

in-order-traversal



Tree Traversal

in-order-traversal



Tree Traversal

in-order-traversal(v)

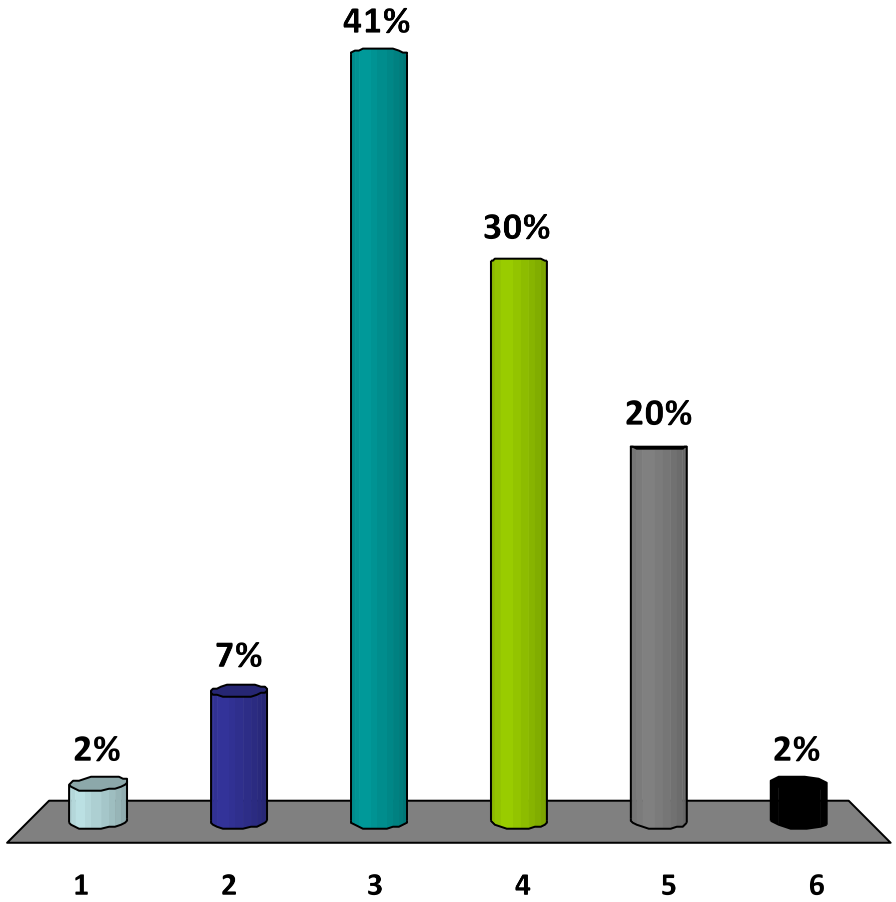
```
public void in-order-traversal(){
        // Traverse left sub-tree
        if (m_leftTree != null)
                m_leftTree.in-order-traversal();

        visit(this);

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

       visit(this);

       // Traverse right sub-tree
       if (m_rightTree != null)
             m_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)

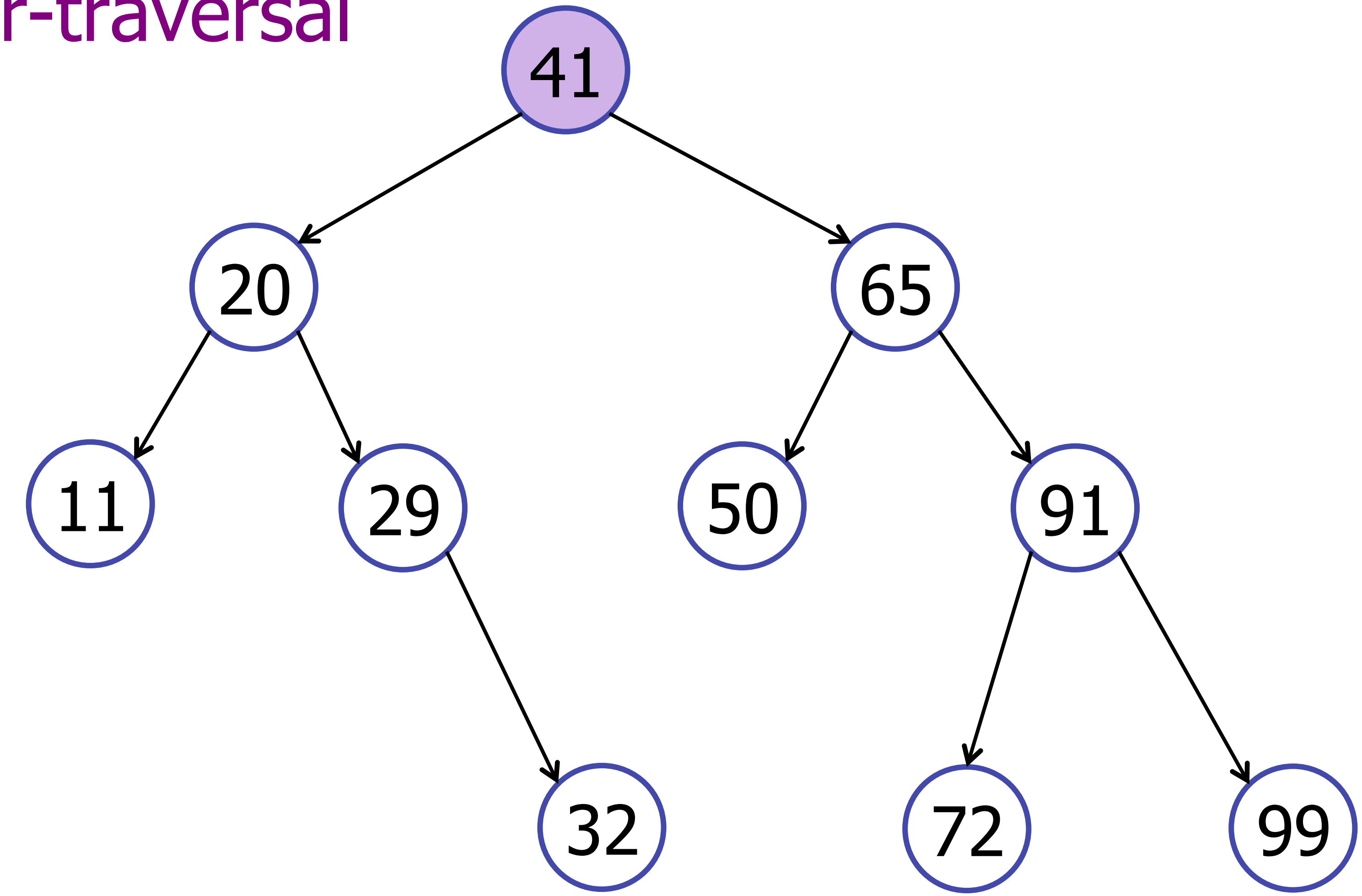
```
public void pre-order-traversal(){
        visit(this);

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

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

Tree Traversals

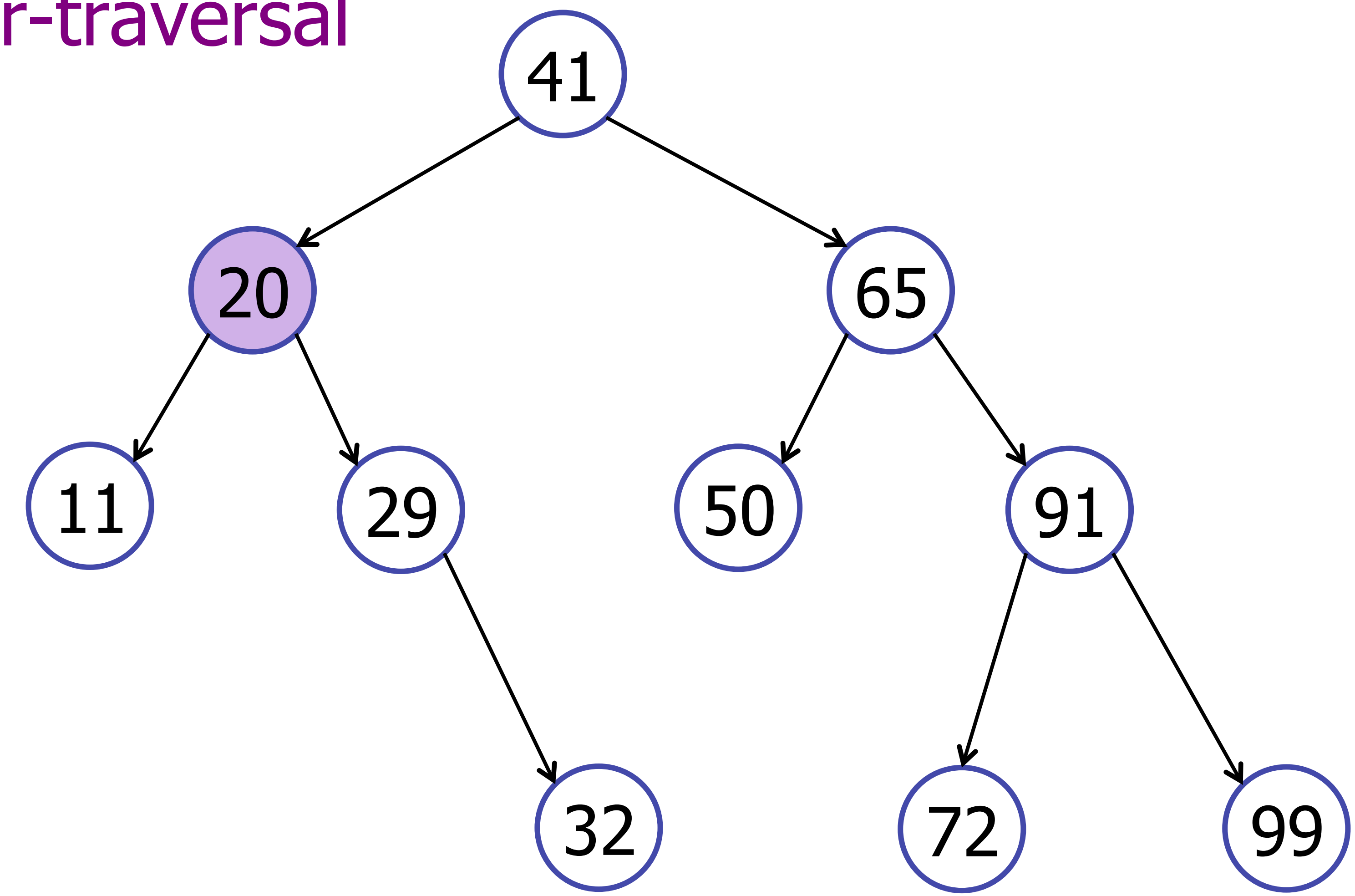
pre-order-traversal



41

Tree Traversals

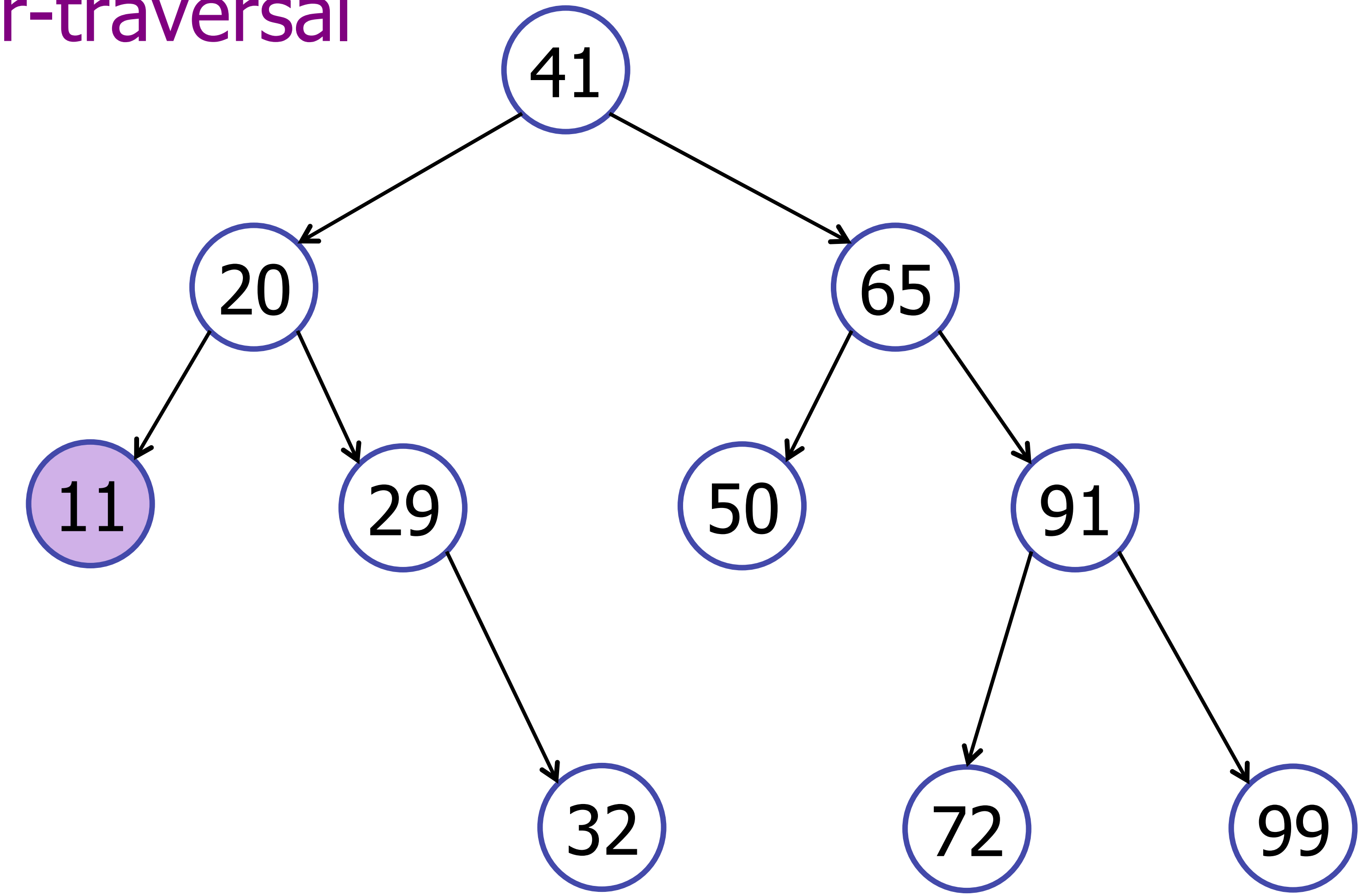
pre-order-traversal



41 20

Tree Traversals

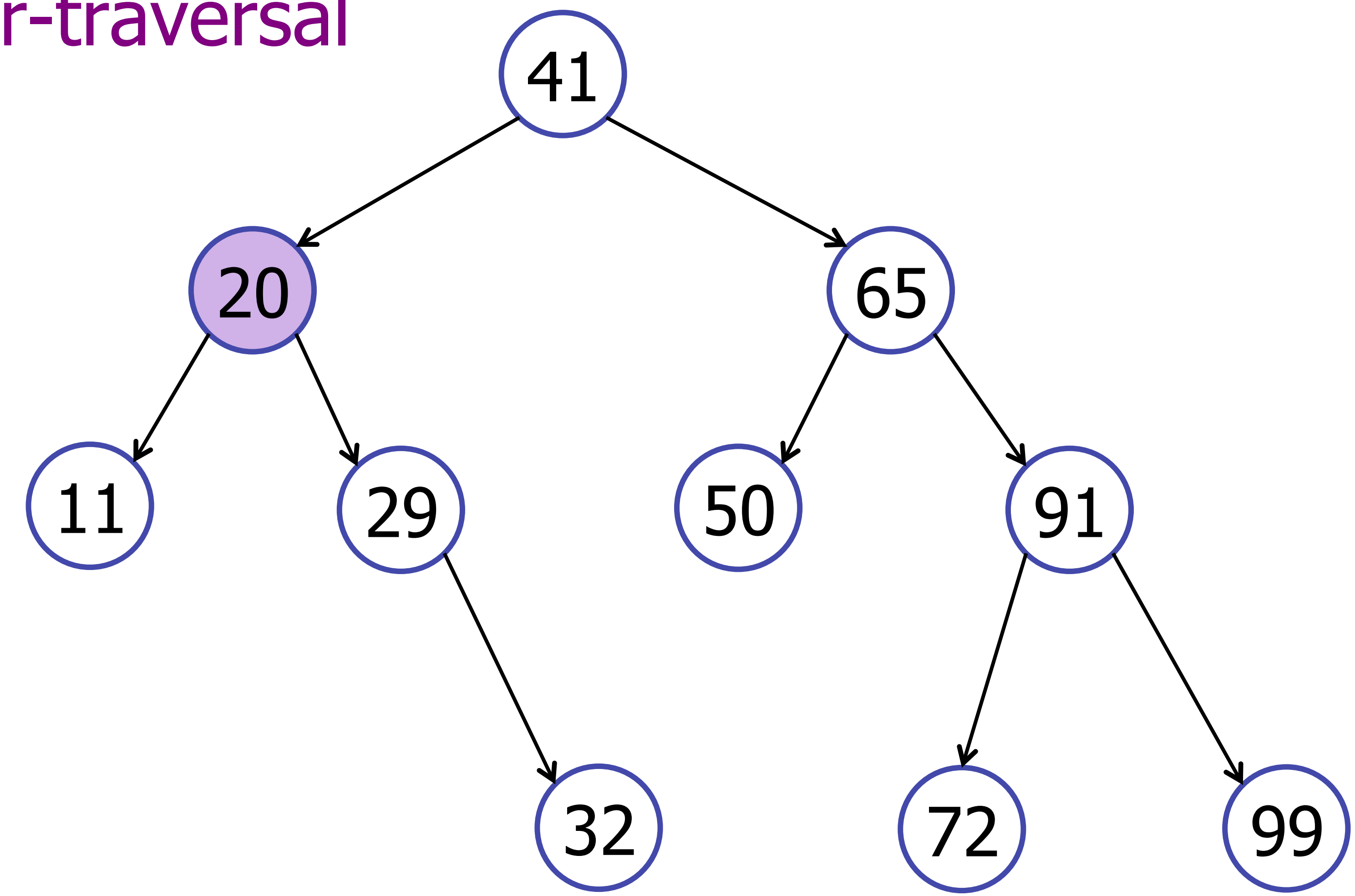
pre-order-traversal



41 20 11

Tree Traversals

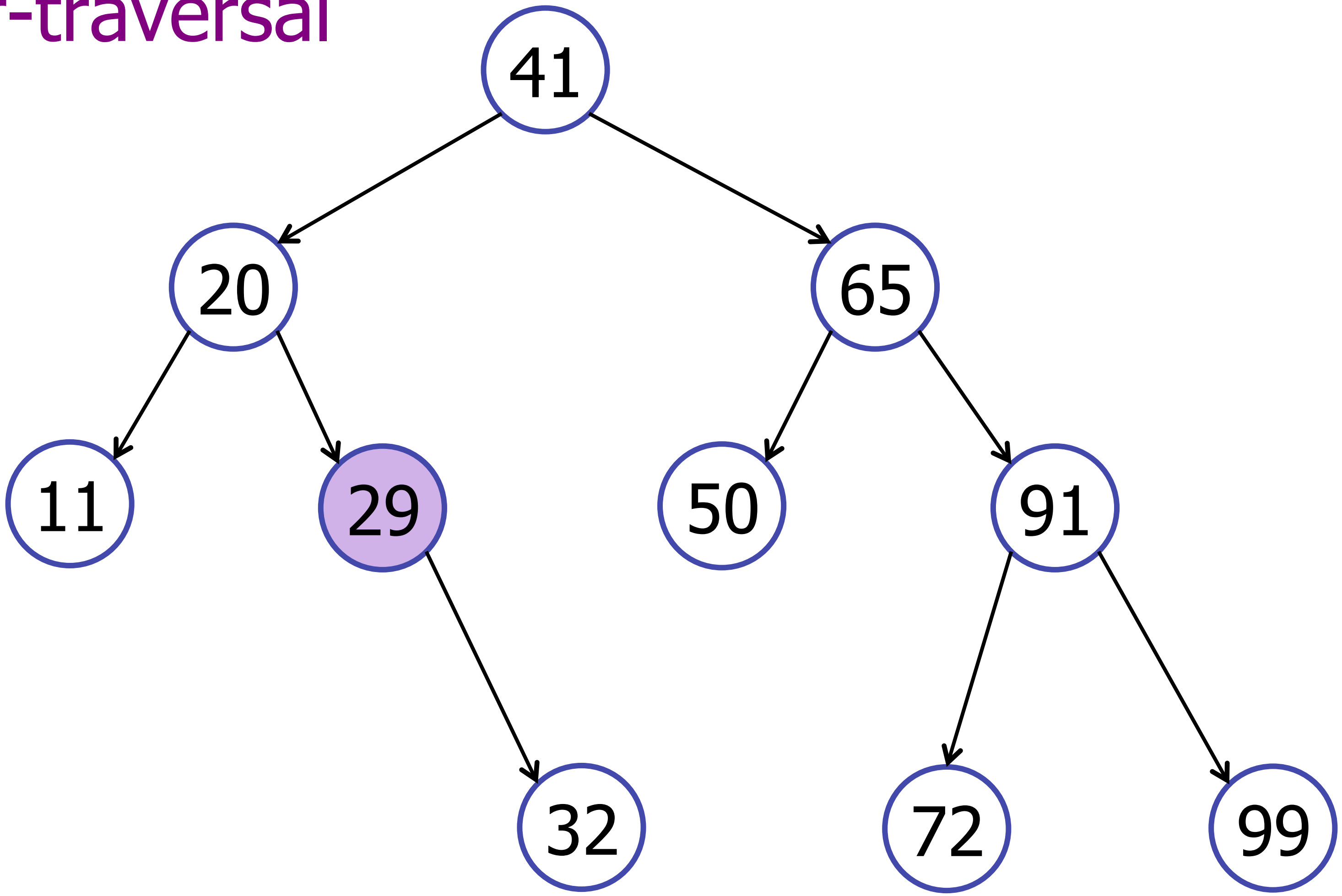
pre-order-traversal



41 20 11

Tree Traversals

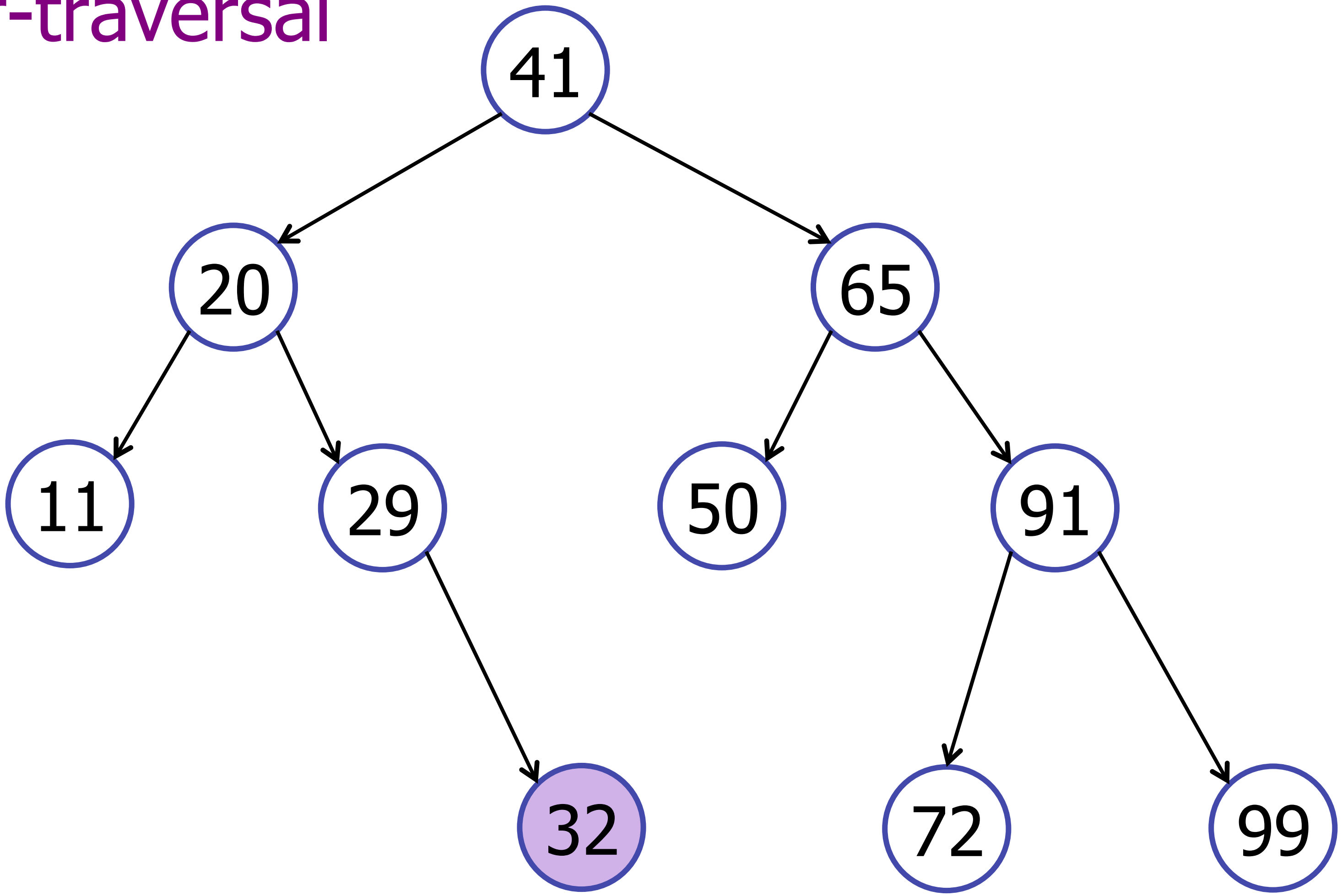
pre-order-traversal



41 20 11 29

Tree Traversals

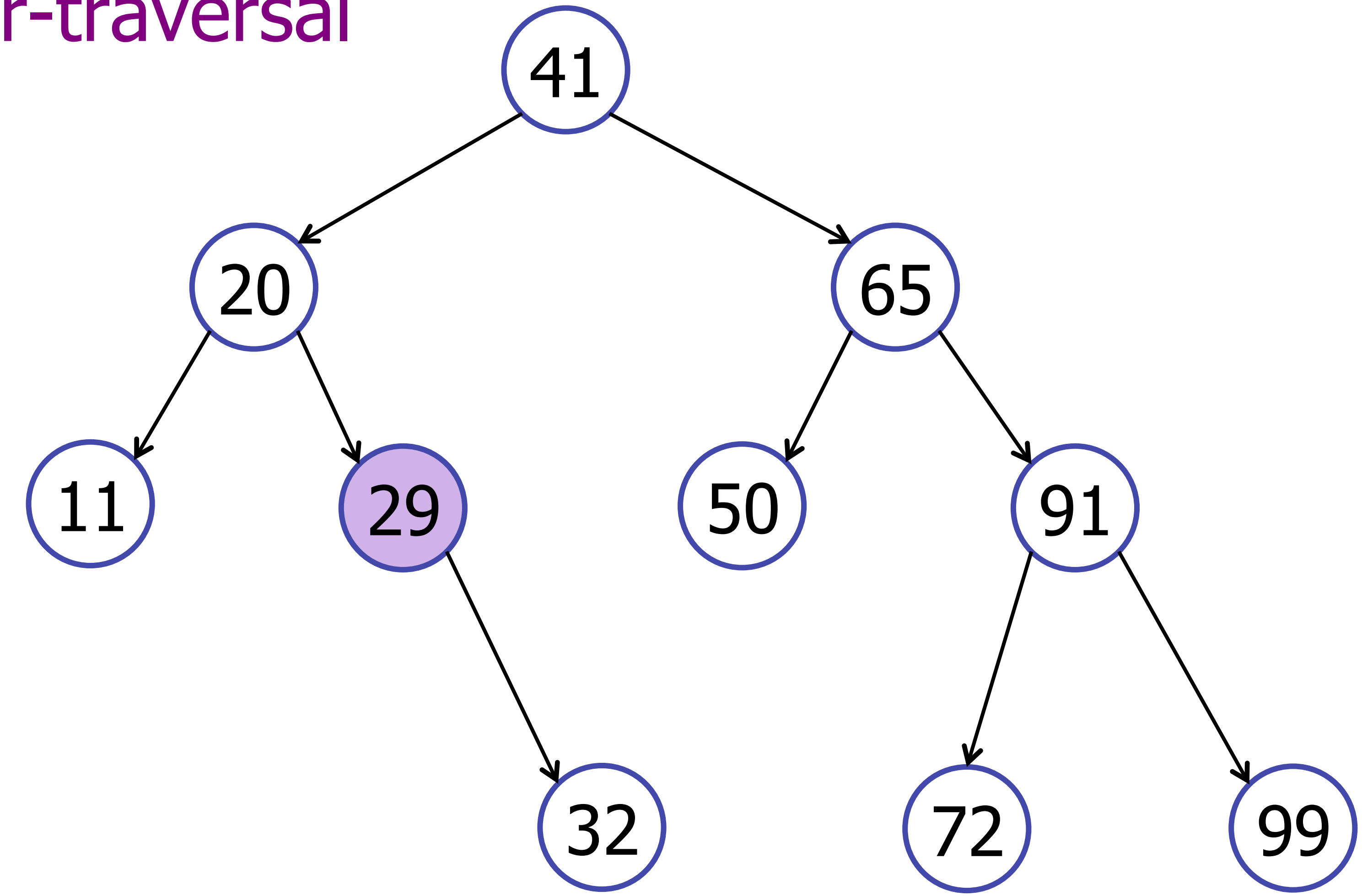
pre-order-traversal



41 20 11 29 32

Tree Traversals

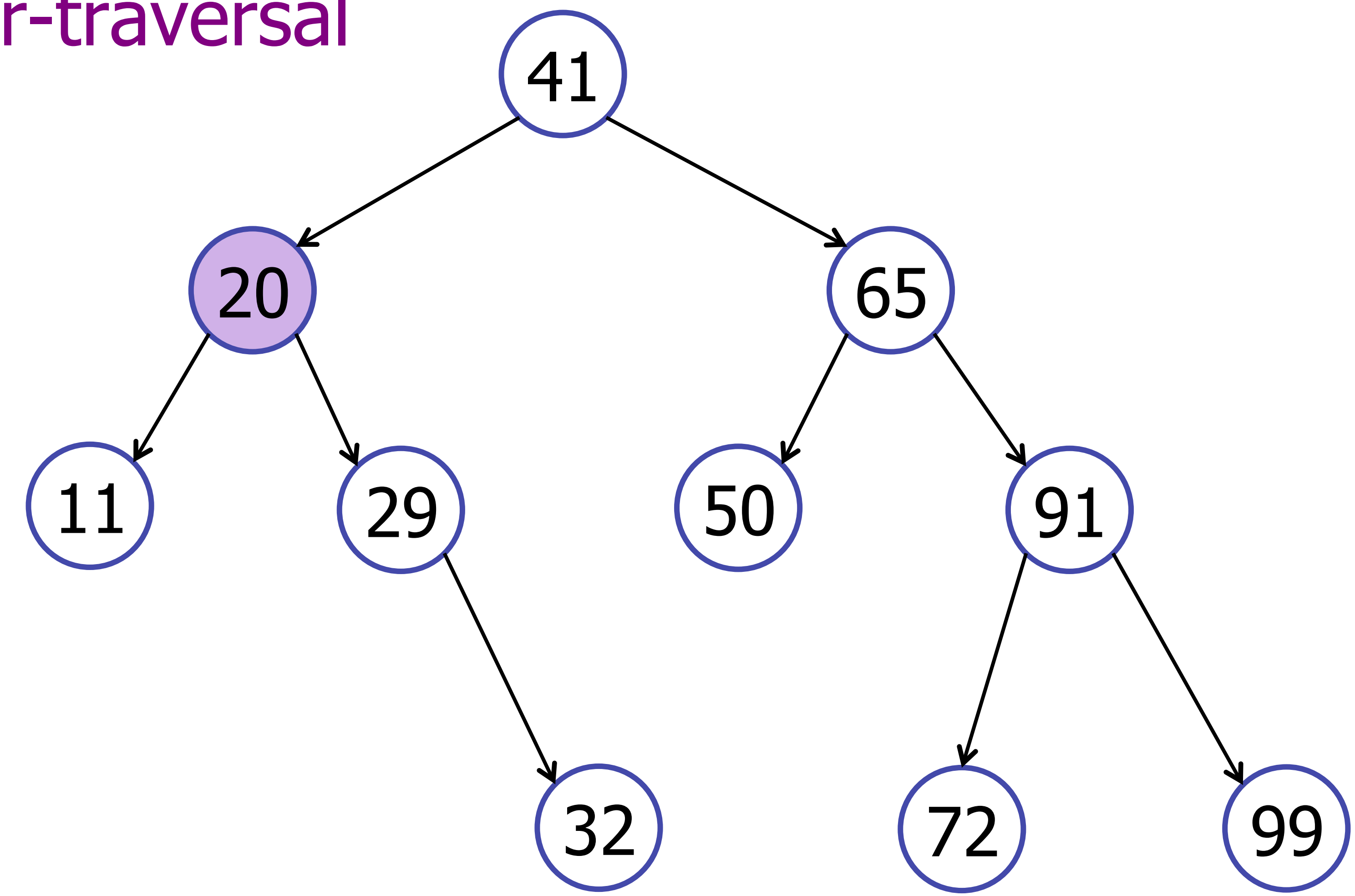
pre-order-traversal



41 20 11 29 32

Tree Traversals

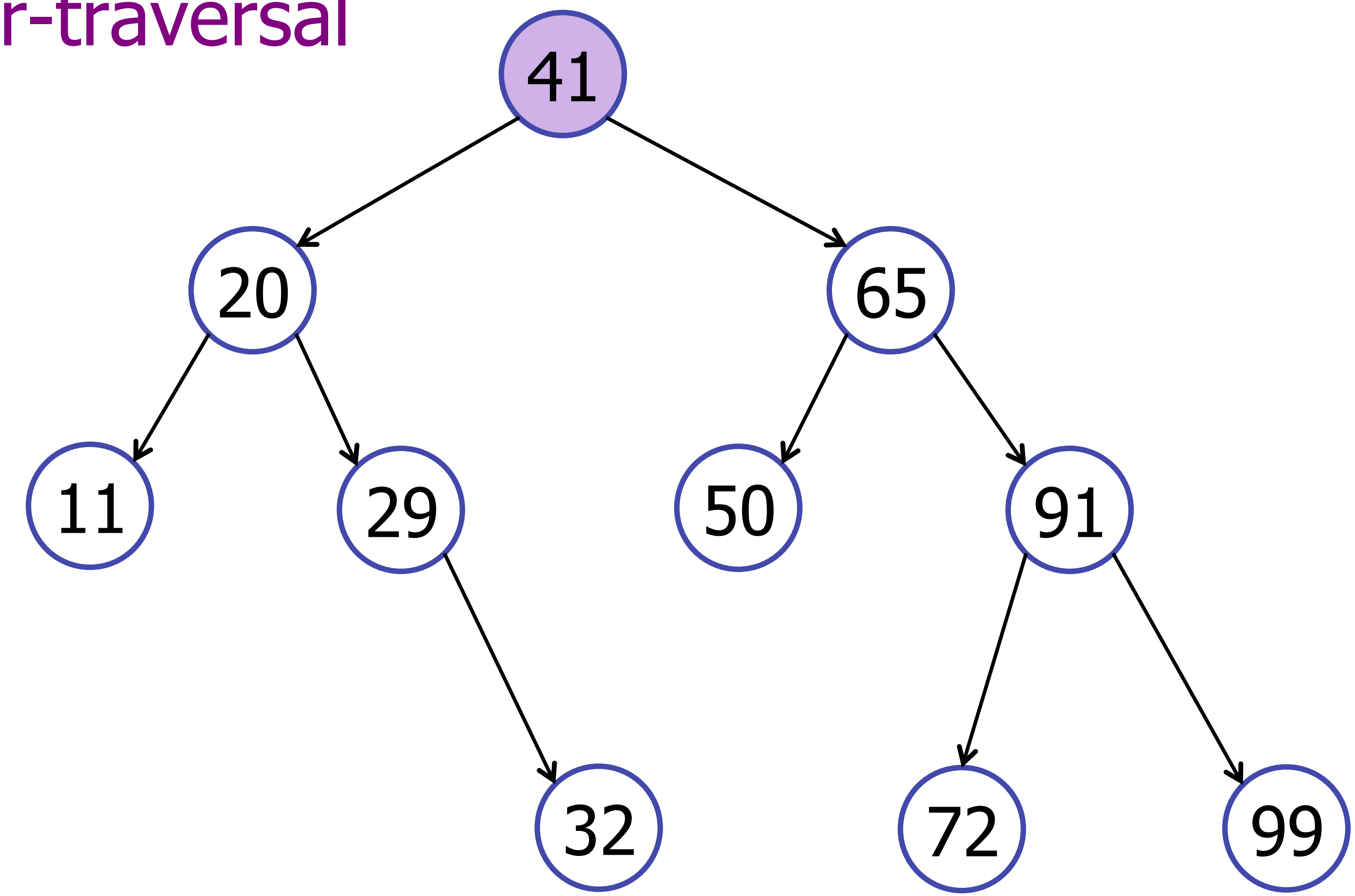
pre-order-traversal



41 20 11 29 32

Tree Traversals

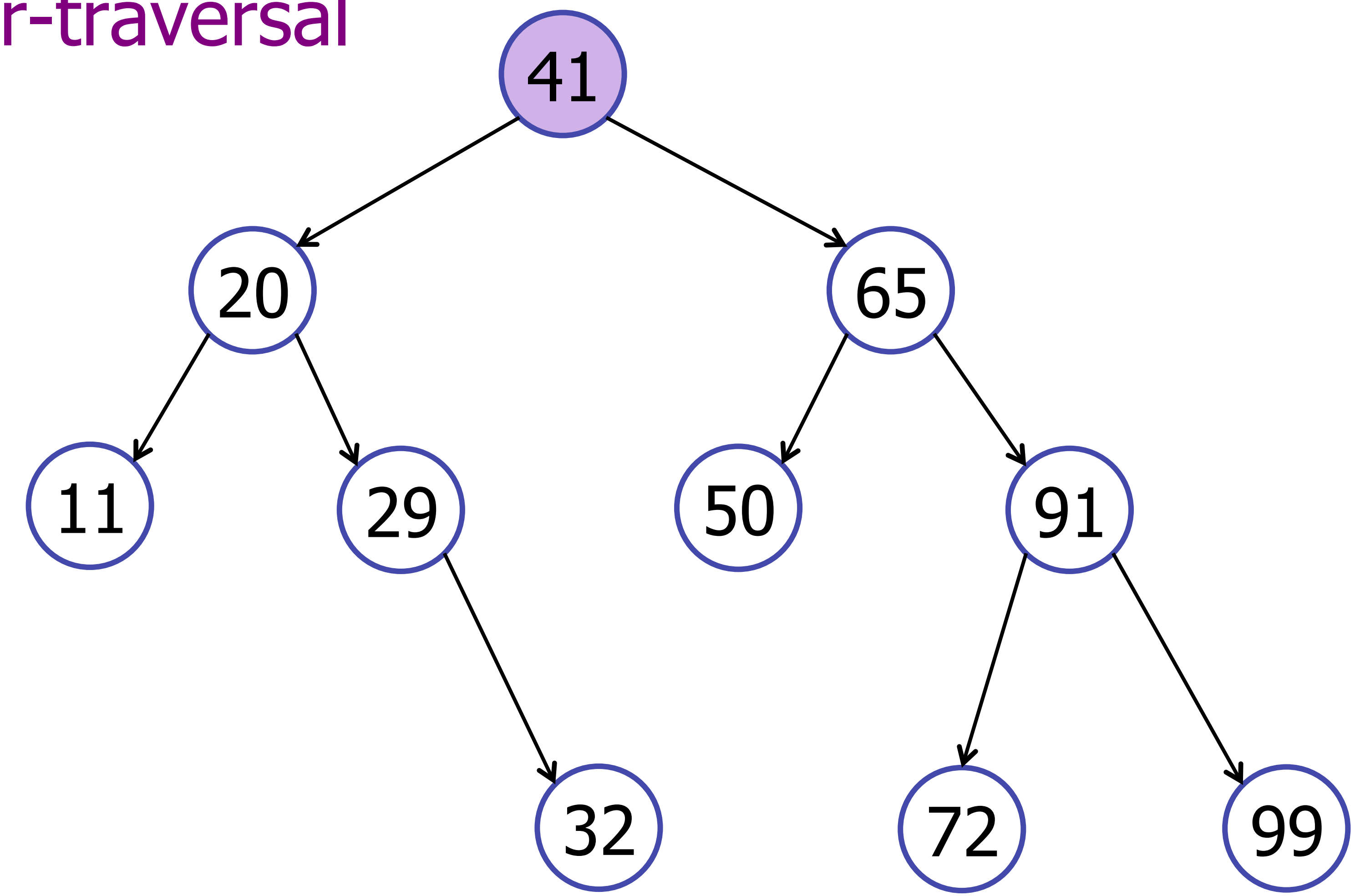
pre-order-traversal



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)

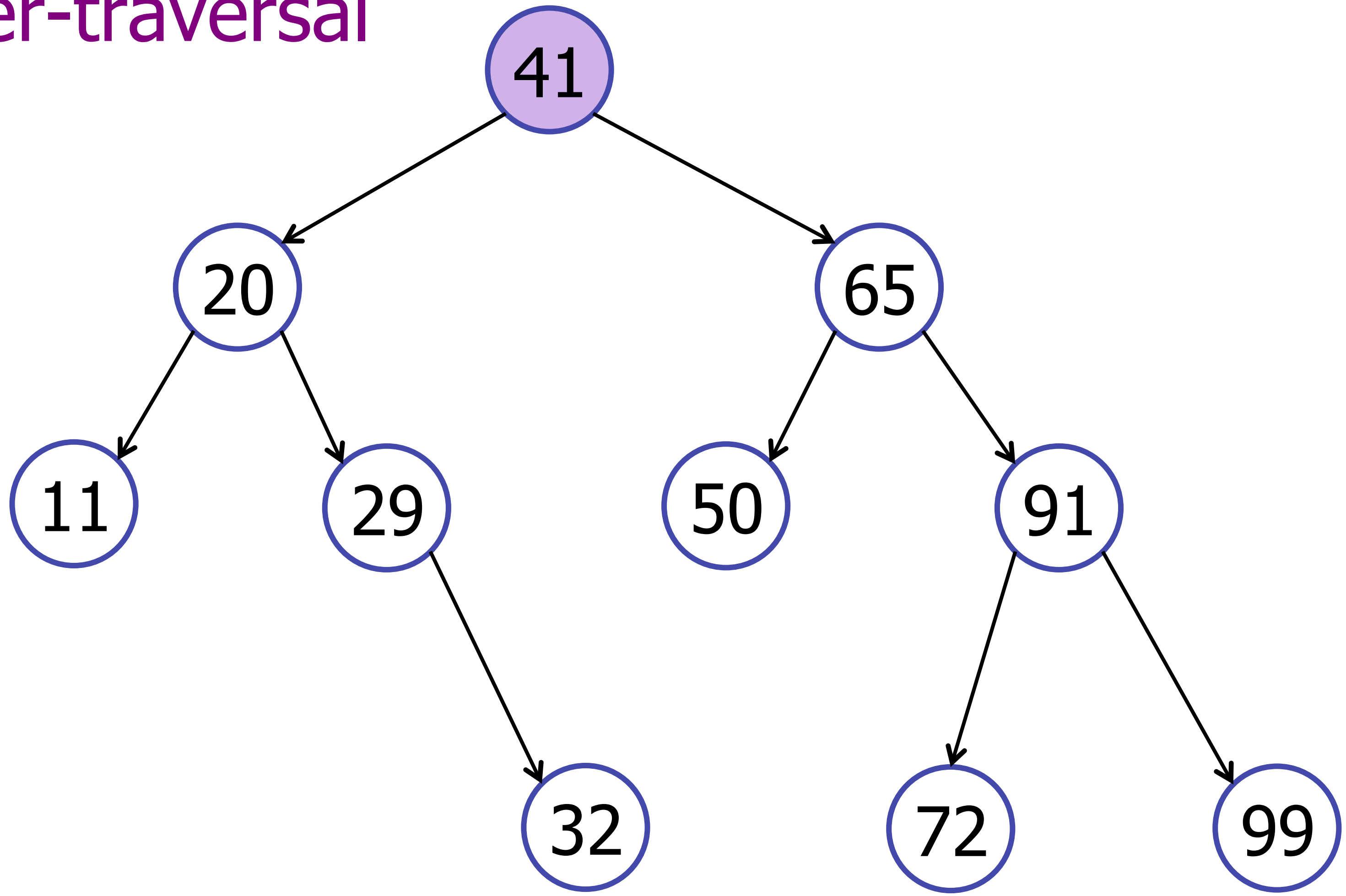
```
public void post-order-traversal(){
        // Traverse left sub-tree
        if (m_leftTree != null)
                m_leftTree.in-order-traversal();

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

        visit(this);
}
```

Tree Traversals

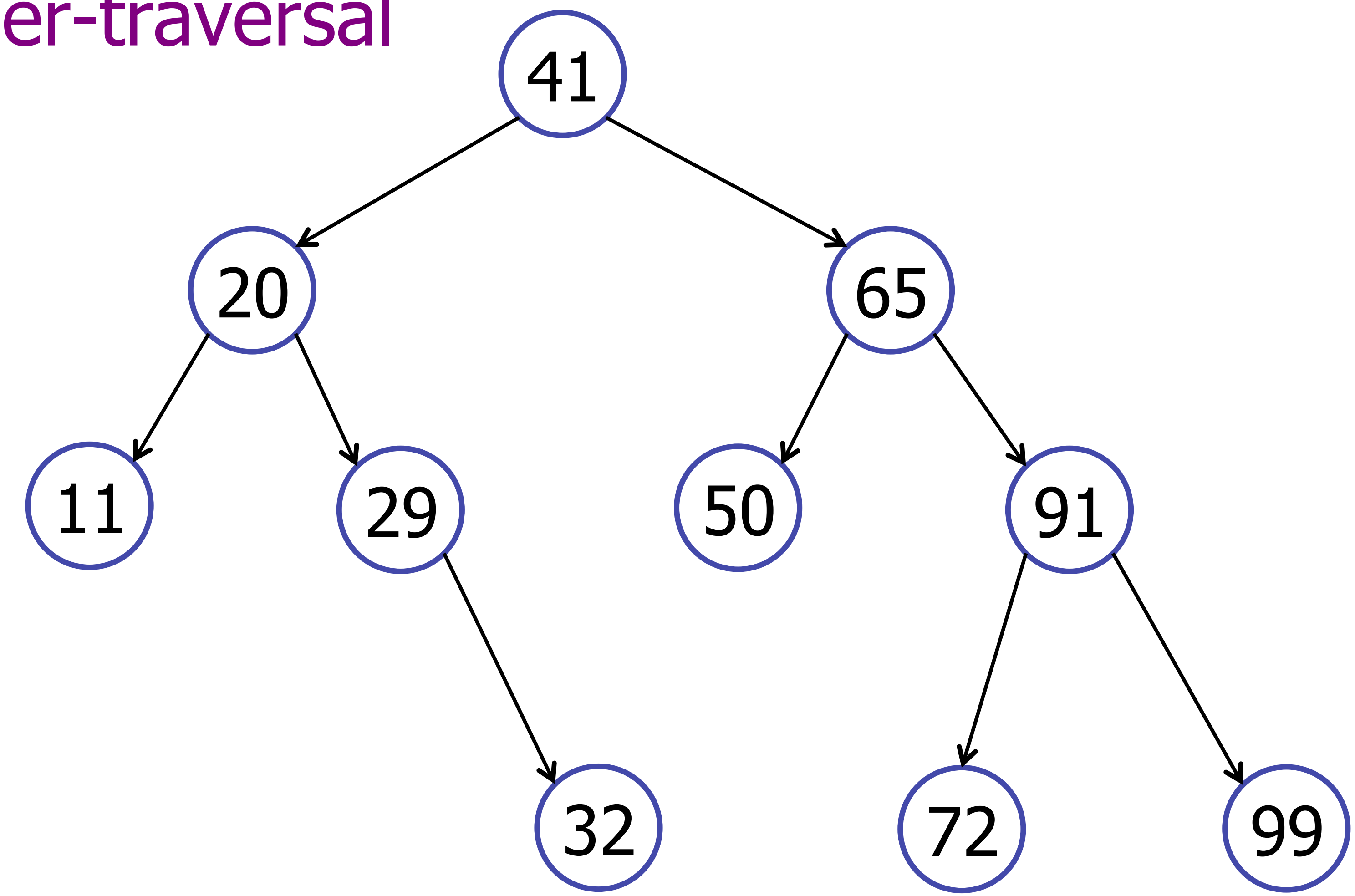
post-order-traversal



11 32 29 20 50 72 99 91 65 41

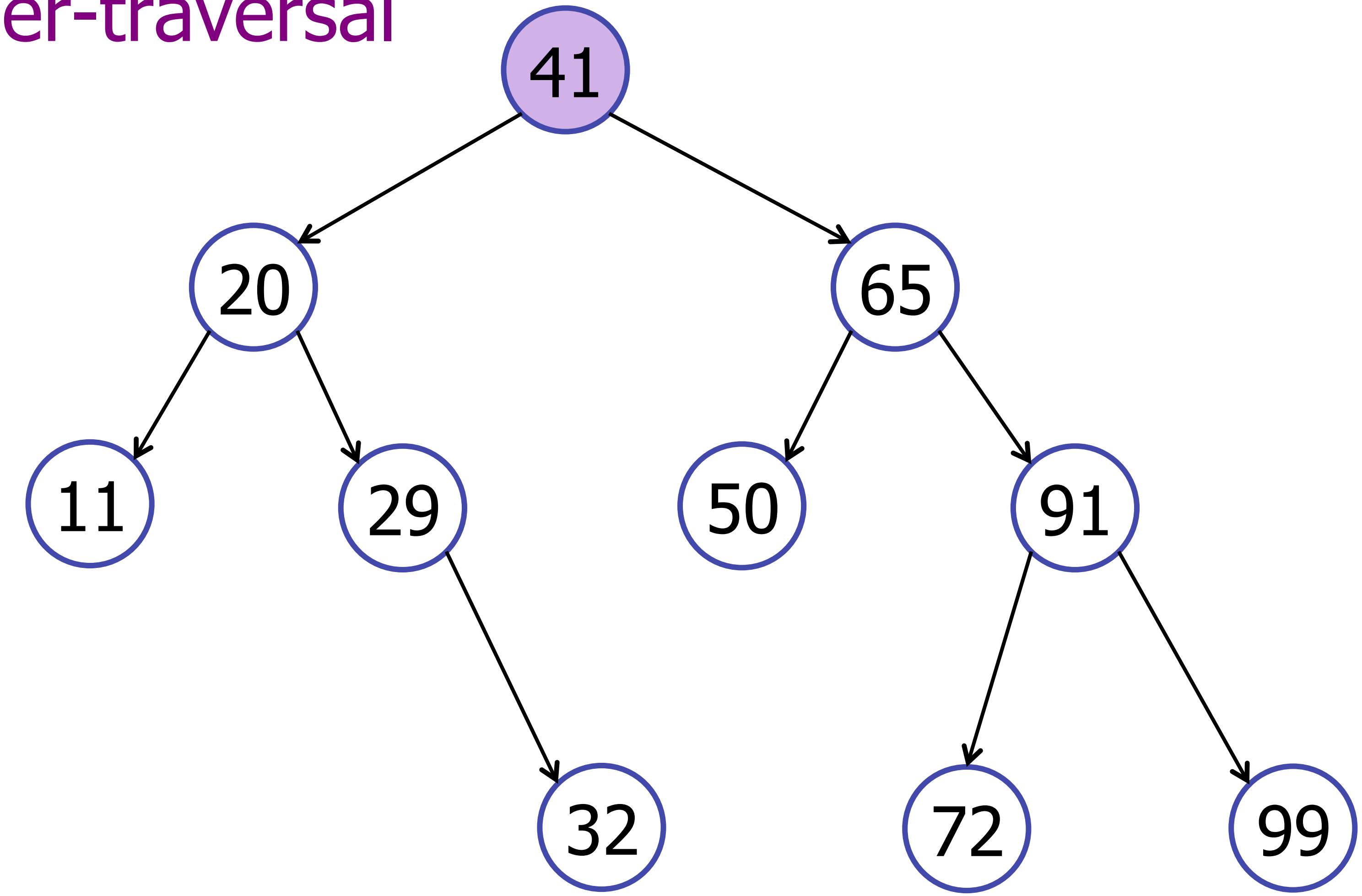
Tree Traversals

level-order-traversal



Tree Traversals

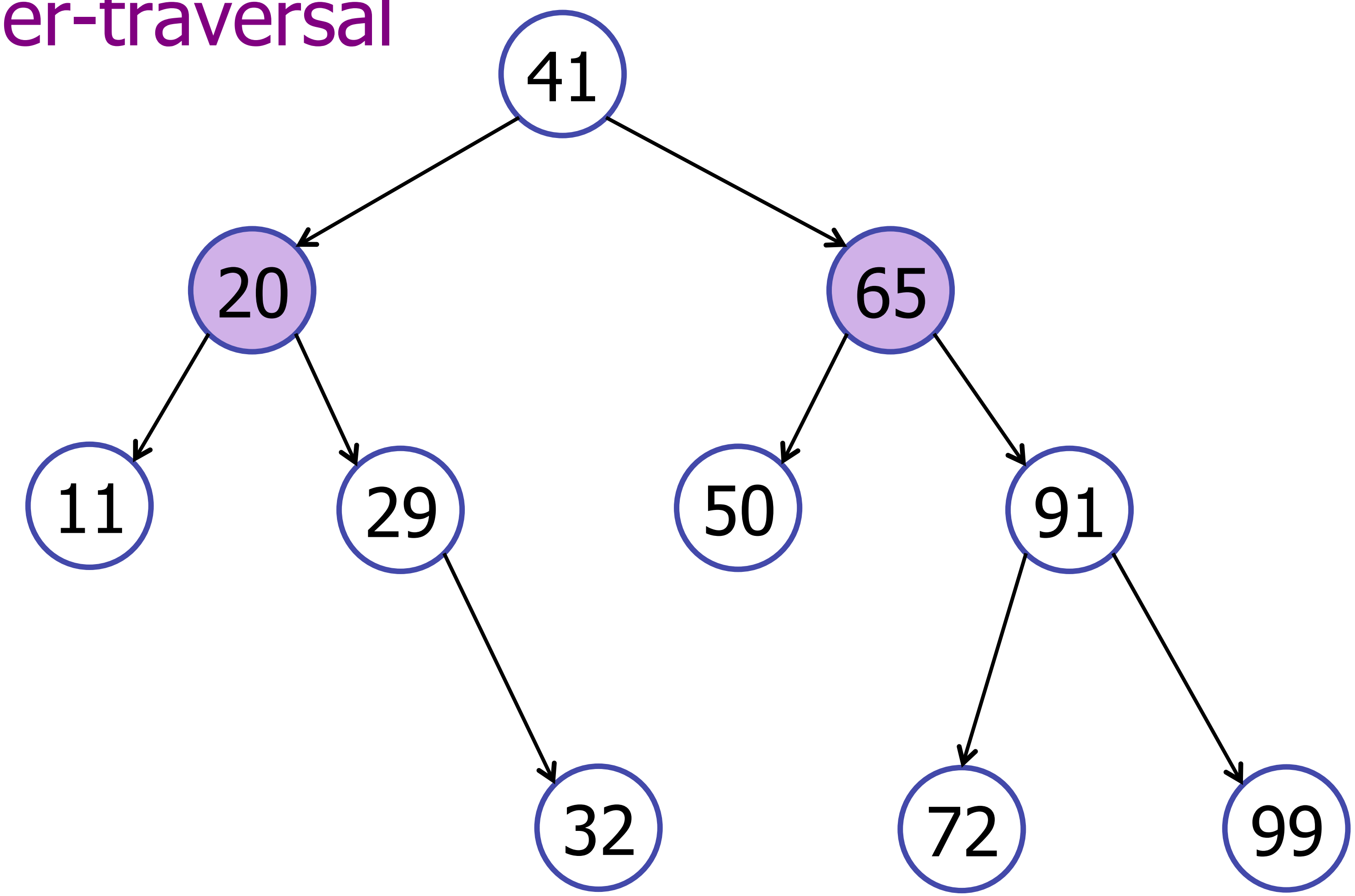
level-order-traversal



41

Tree Traversals

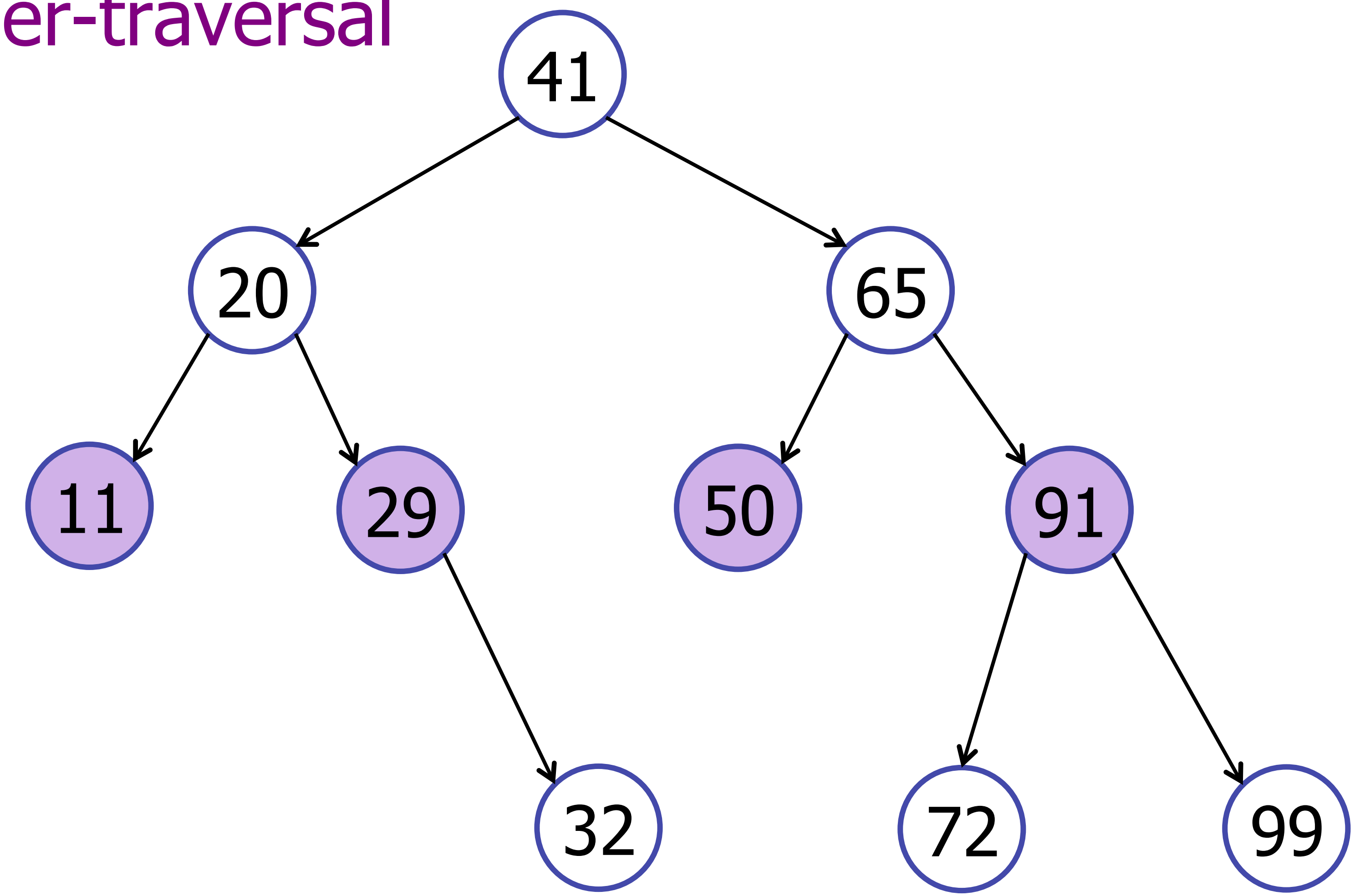
level-order-traversal



41 20 65

Tree Traversals

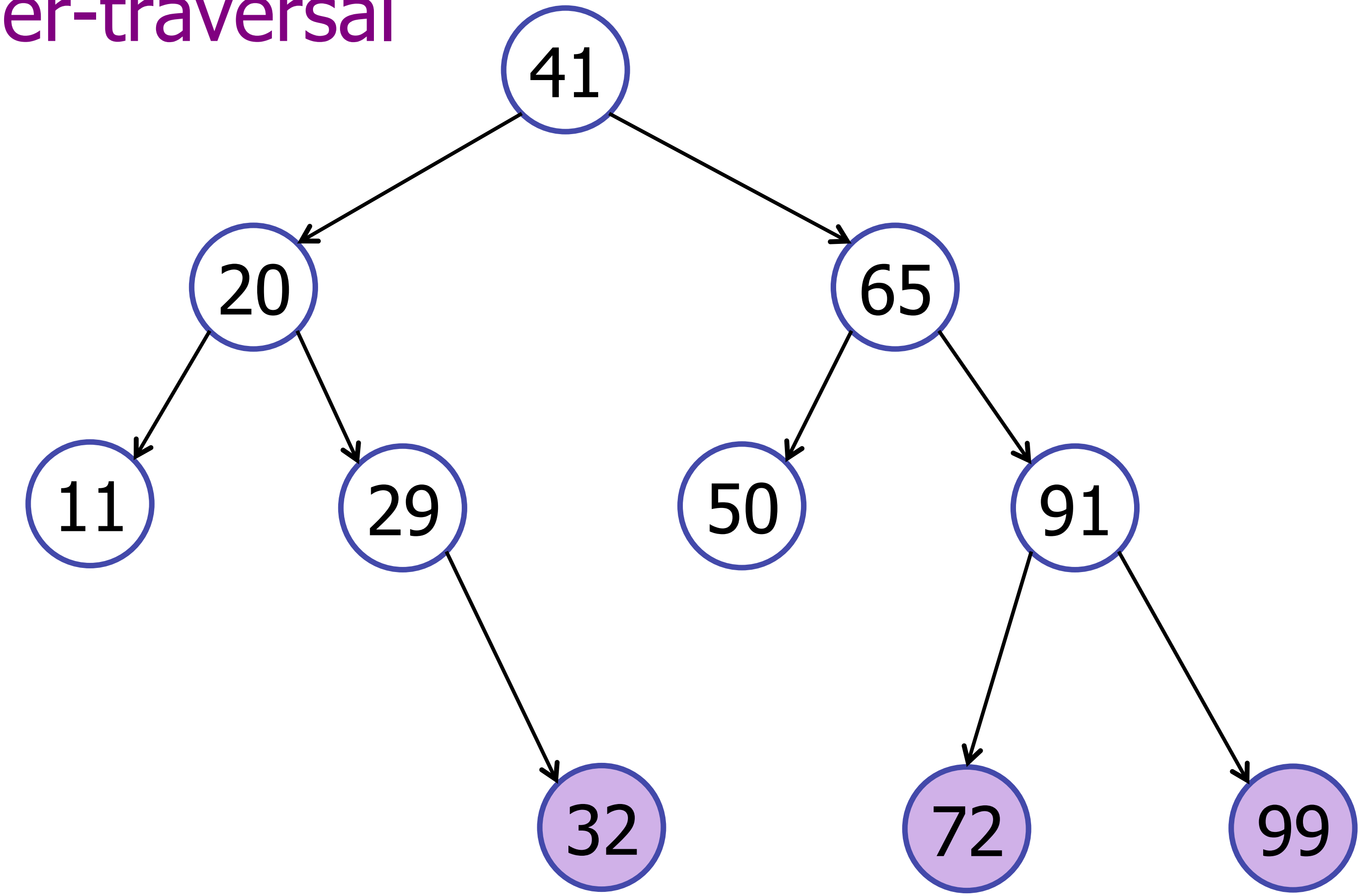
level-order-traversal



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 iterator
- in-order iterator
- post-order iterator
- level-order iterator

Tree Traversals

Tree implements Iterable<Key>

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

Tree Traversals

Tree implements Iterable<Key>

```
private class TreeIterator implements Iterator<Key>{

      BinaryTree currentNode;

      public boolean hasNext(){
            return (current != null);
      }

      public Key next(){
            // What goes here?
      }
}
```

Binary Search Trees

1. Terminology and Definitions

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

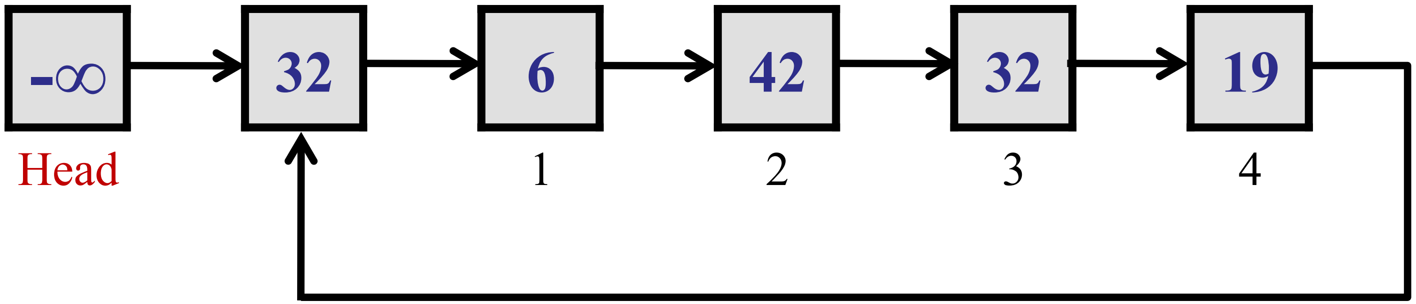
3. Traversals
   - in-order, pre-order, post-order

4. Other operations

Puzzle Break

Standard Interview Question 2:

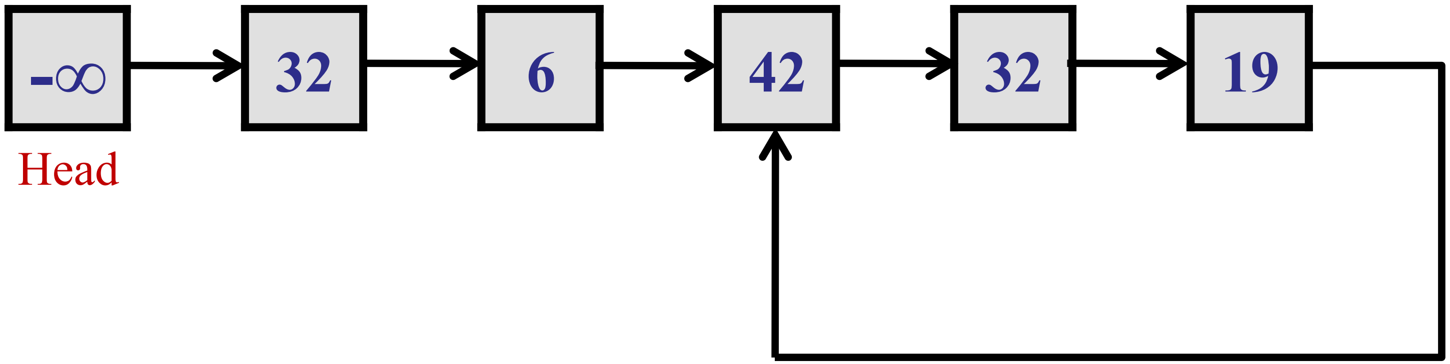
– A linked list may be circular…



Puzzle Break

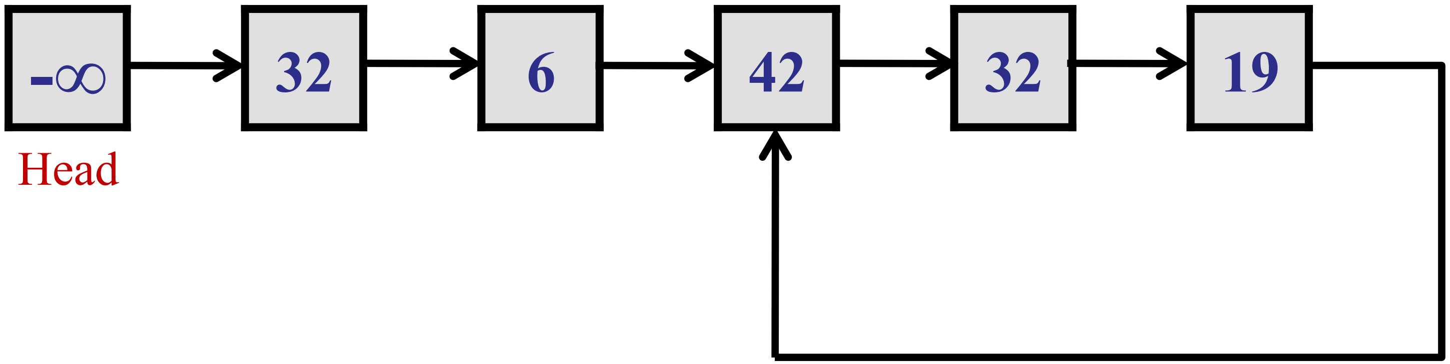
Standard Interview Question 2:

– Or a linked list may contain a loop of unknown size…



Puzzle Break

Does the linked list have a loop?



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

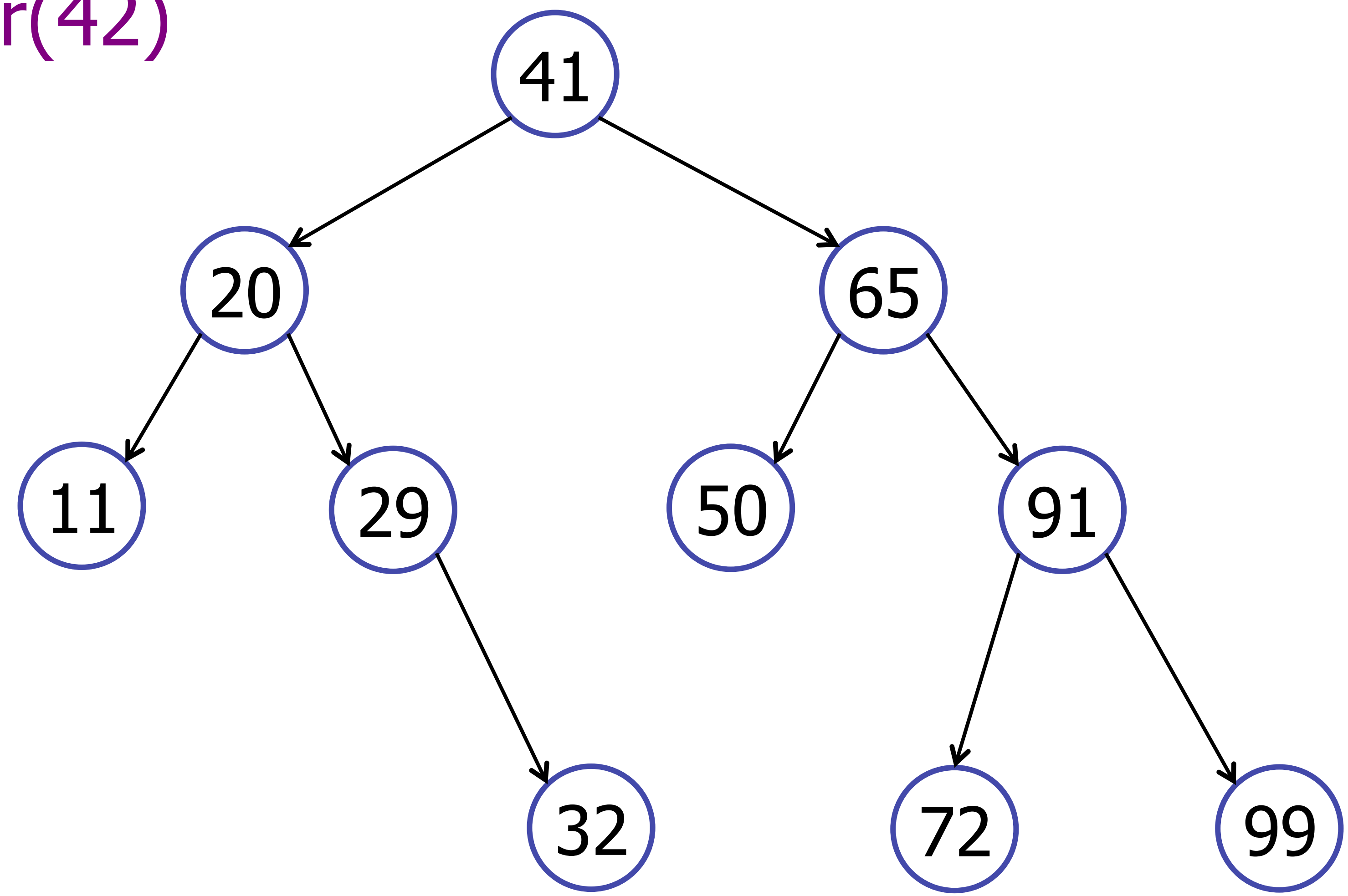
6:35	**7:00**	**7:19**	**8:21**	**12:21**	**14:23**	**14:42**			

- successor(8:24) = 12:21

How do we implement this?

Successor Queries

successor(42)

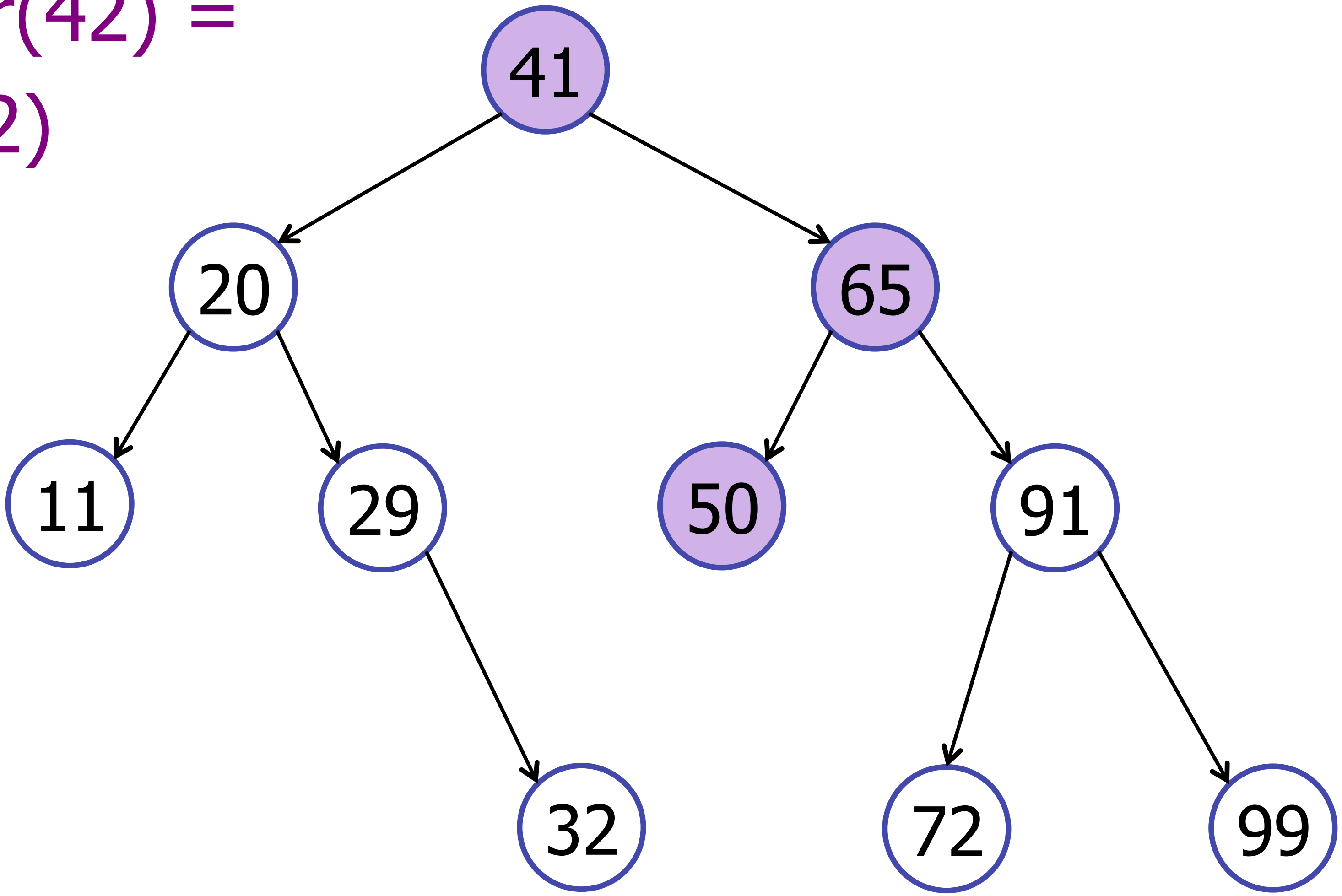


Key 42 is not in the tree

Successor Queries

successor(42) =

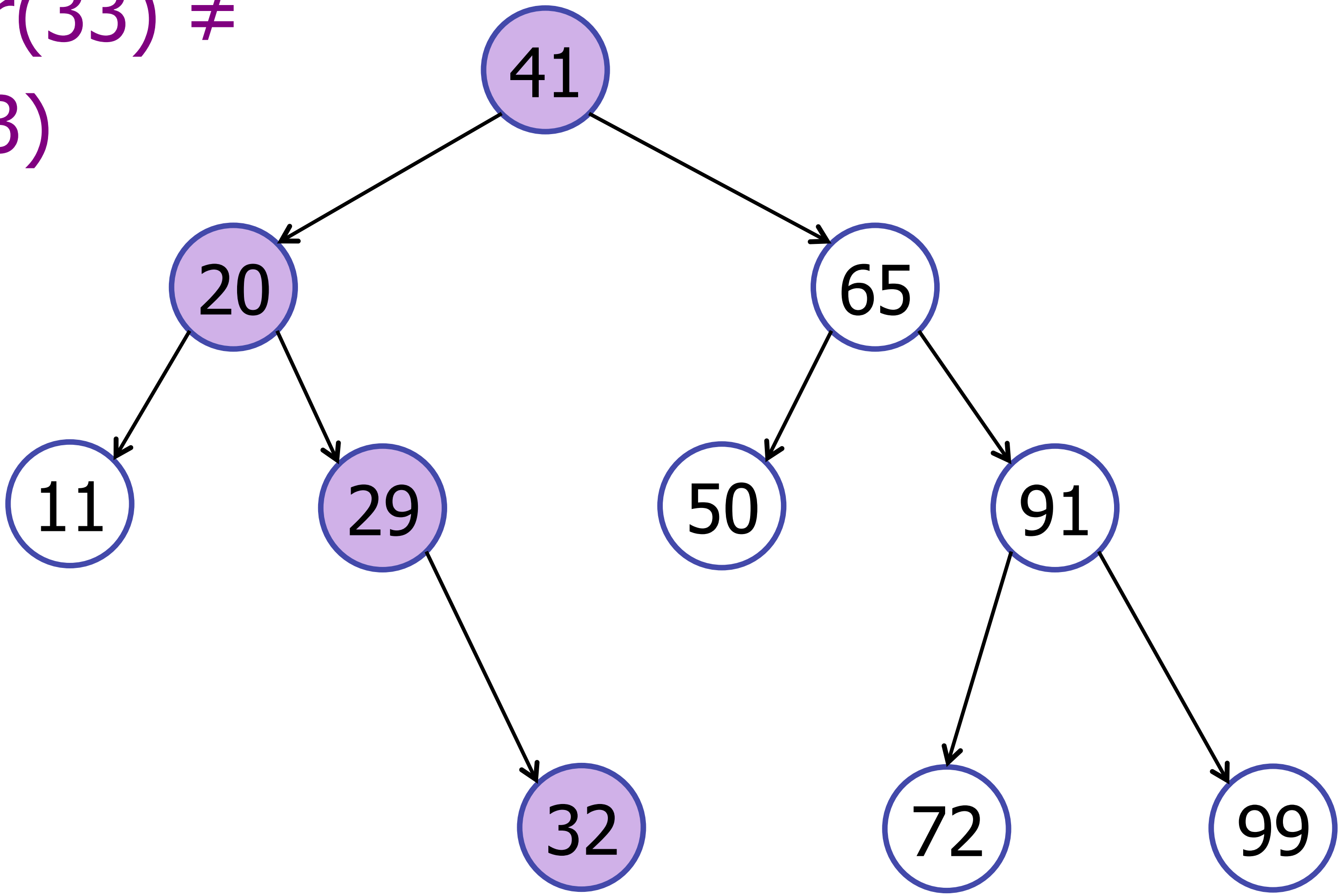
search(42)



Key 42 is not in the tree

Successor Queries

successor(33) ≠
search(33)



Key 33 is not in the tree

Successor Queries

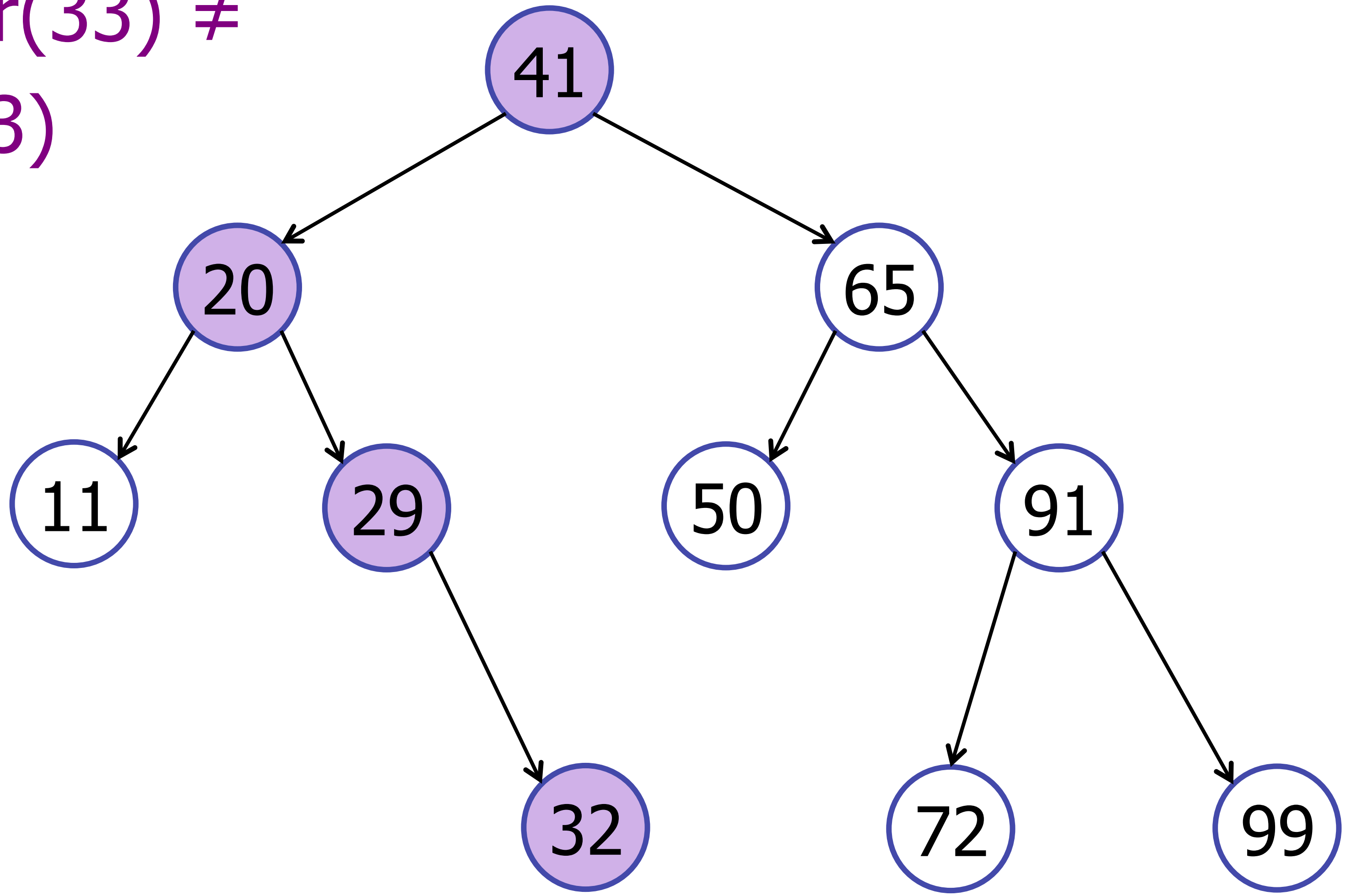
Basic strategy: successor(key)

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

successor(33) ≠

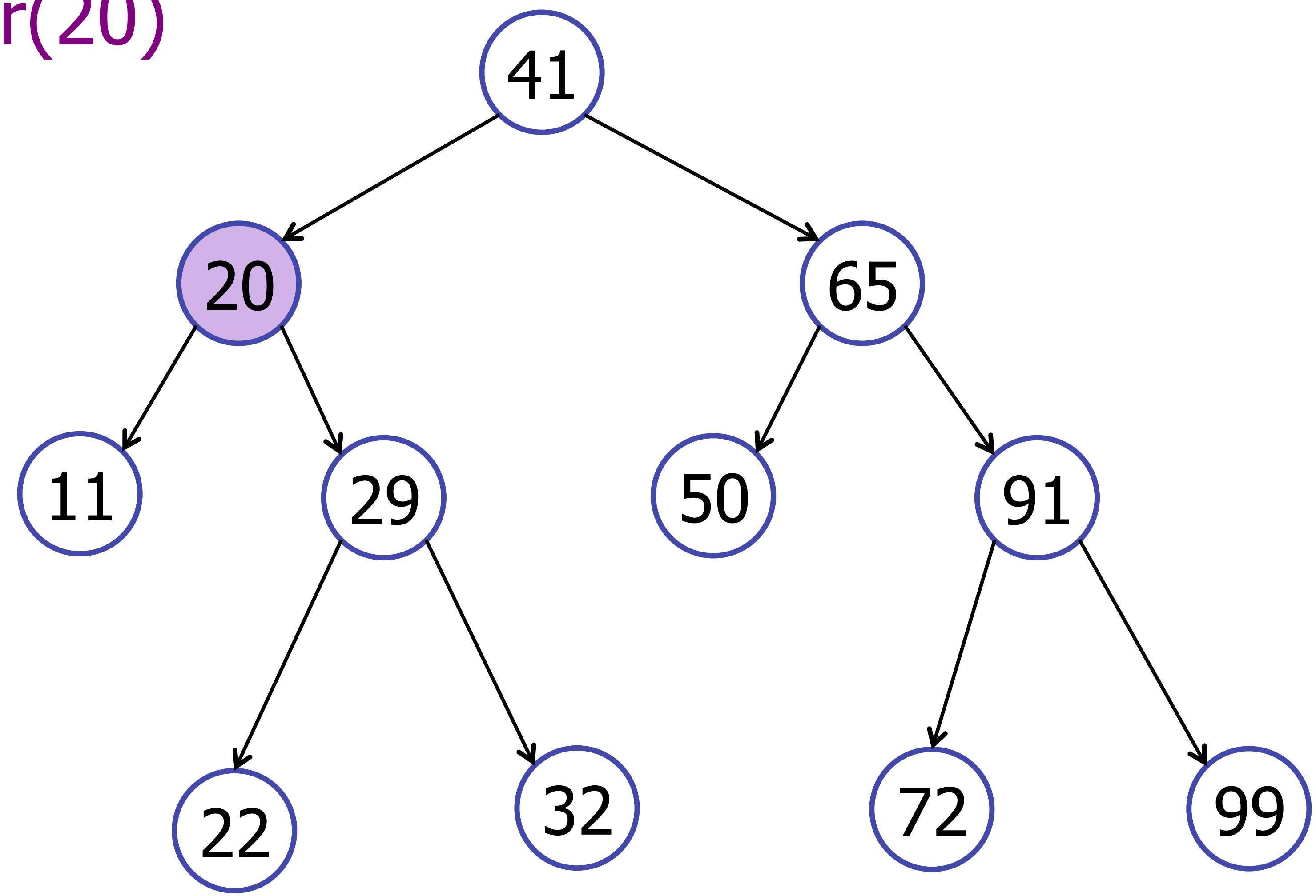
search(33)



Key 33 is not in the tree

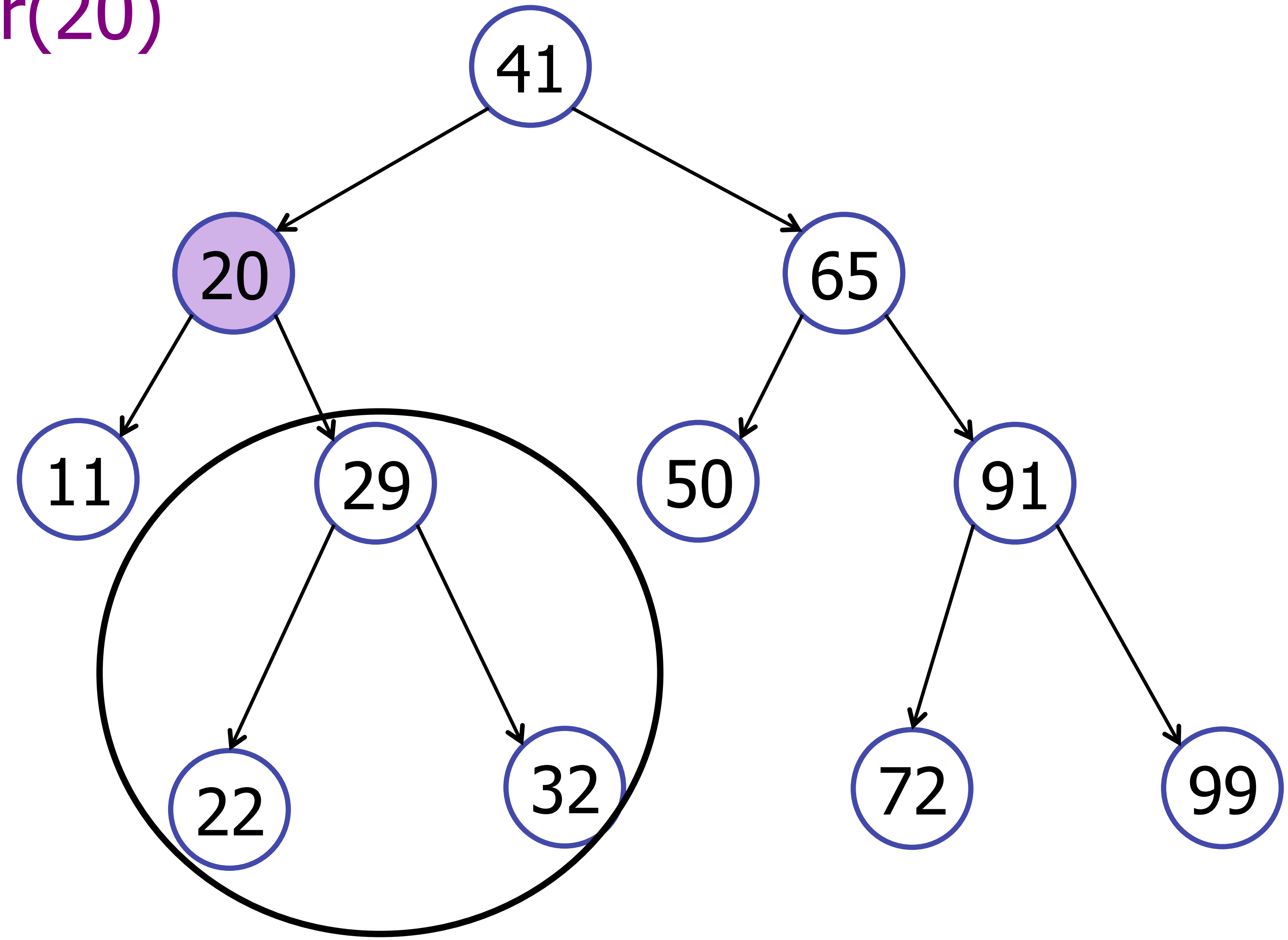
Successor Queries

successor(20)



Successor Queries

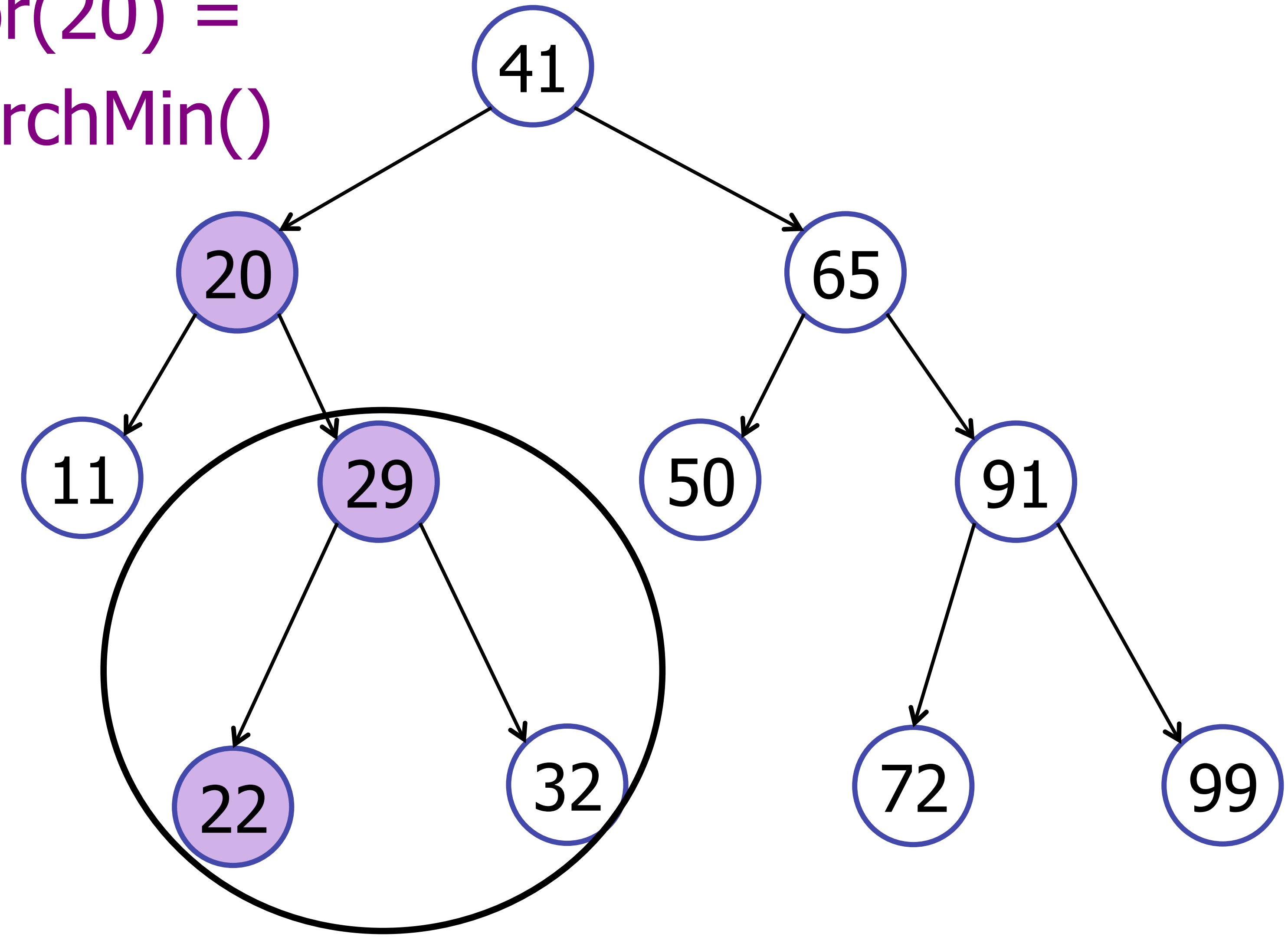
successor(20)



Case 1: node has a right child.

Successor Queries

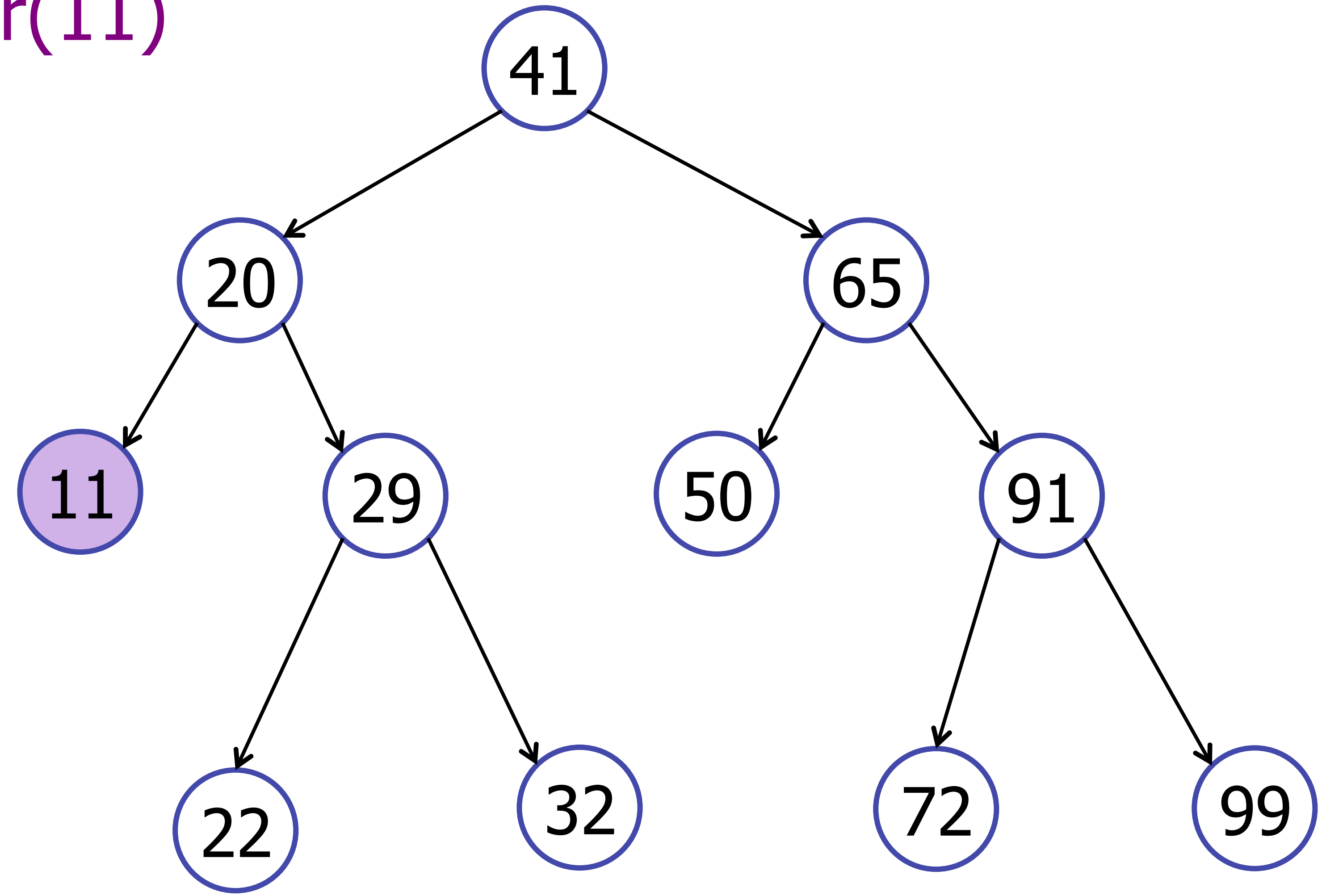
successor(20) =
right.searchMin()



Case 1: node has a right child.

Successor Queries

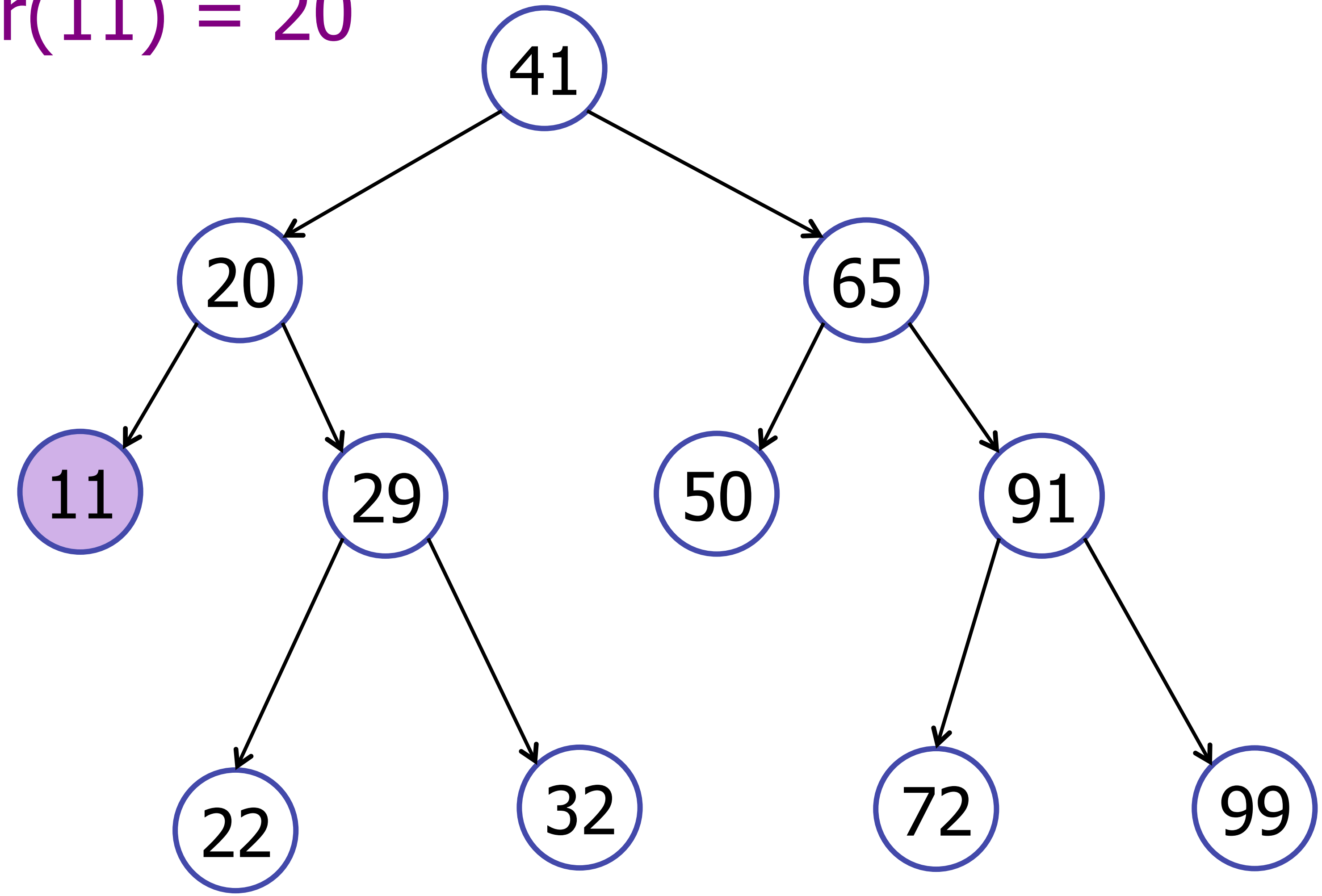
successor(11)



Case 2: node has no right child.

Successor Queries

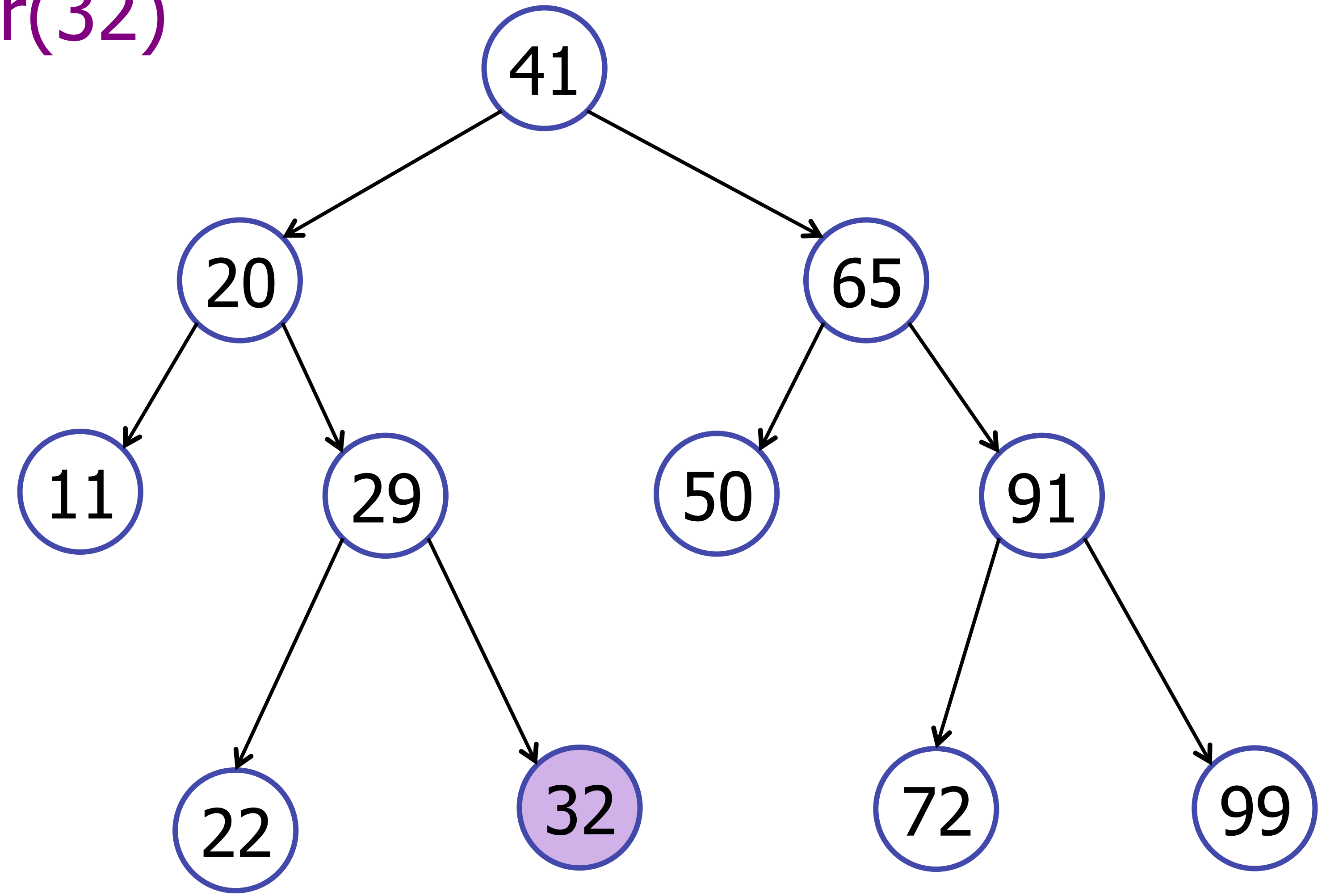
successor(11) = 20



Case 2: node has no right child.

Successor Queries

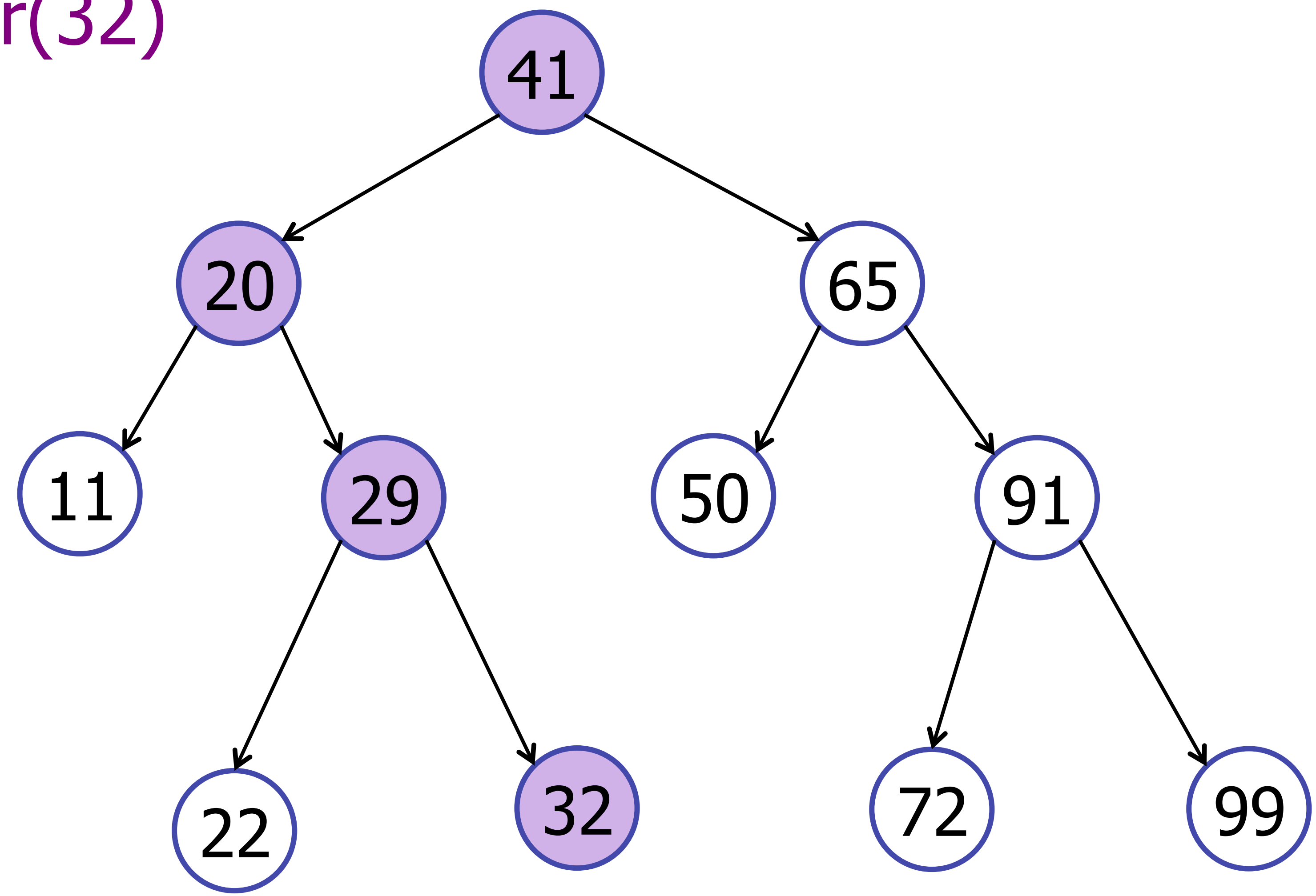
successor(32)



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<Key> successor(){
      if (m_rightTree != null)
            return m_rightTree.searchMin();

      TreeNode parent = m_parentTree;
      TreeNode child = this;
      while ((parent != null) && (child = parent.m_rightTree))
            child = parent;
            parent = child.m_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

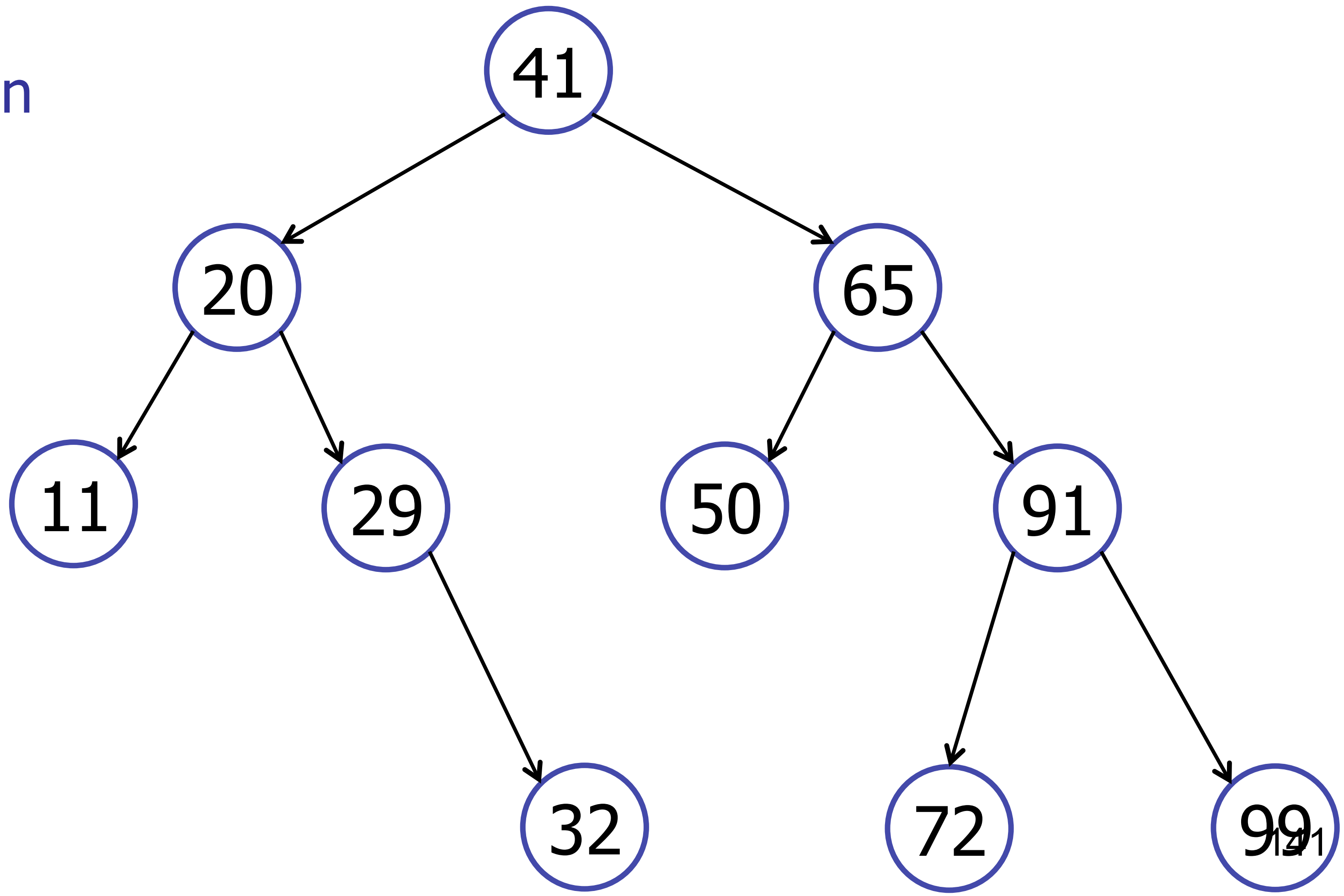
4. Other operations

Binary Search Tree

delete(v)

Three cases:

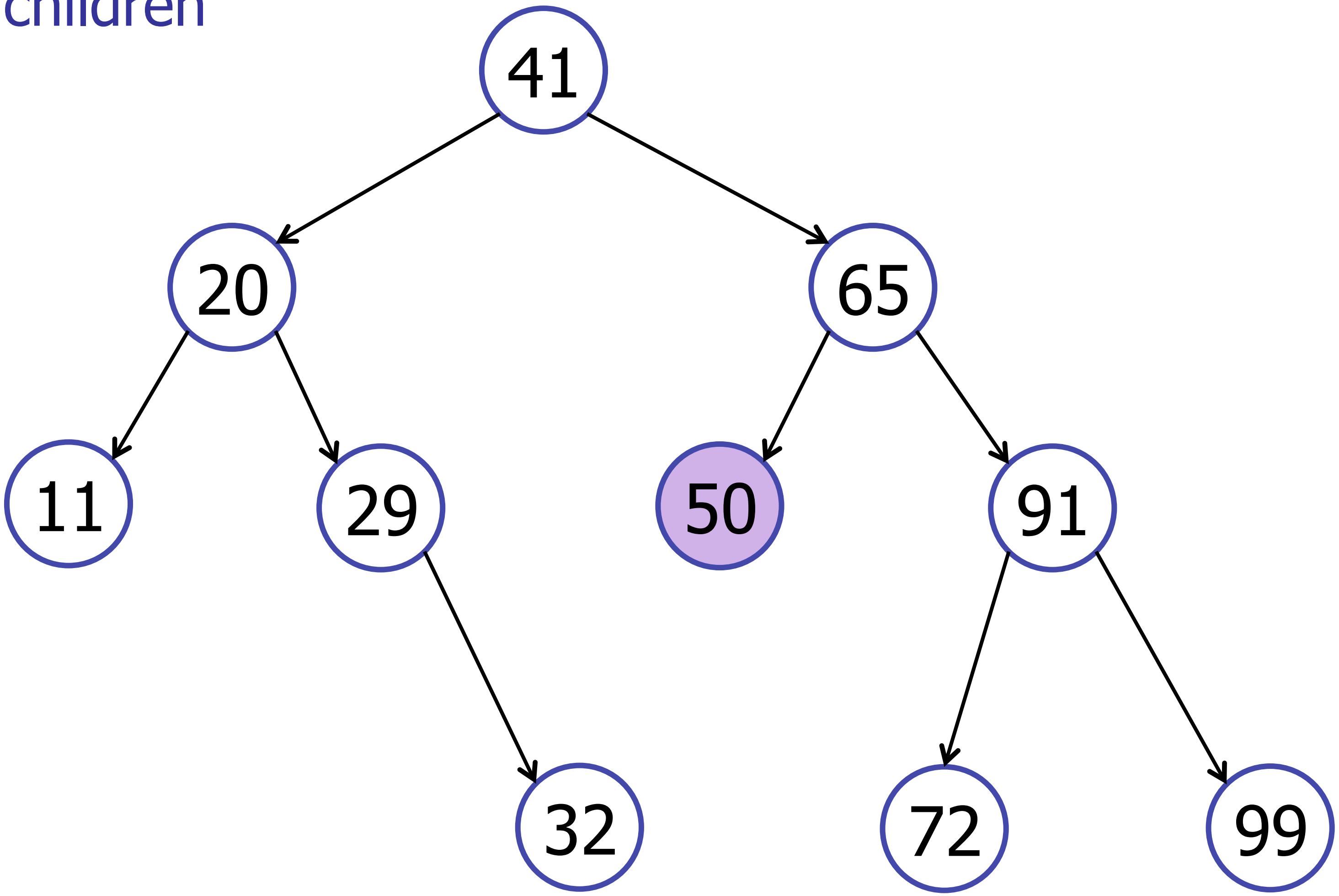
1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(50)

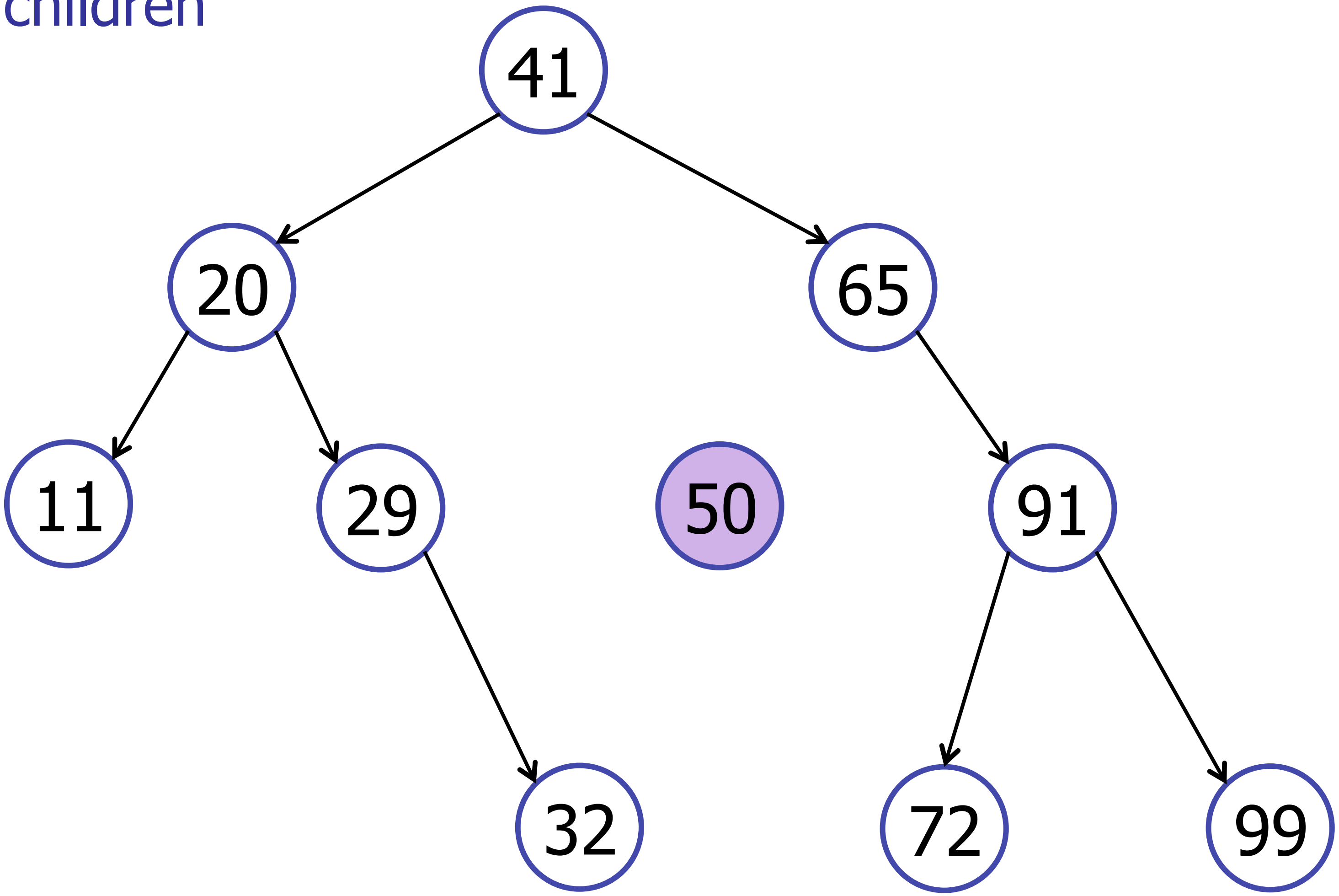
Case 1: No children



Binary Search Tree

delete(50)

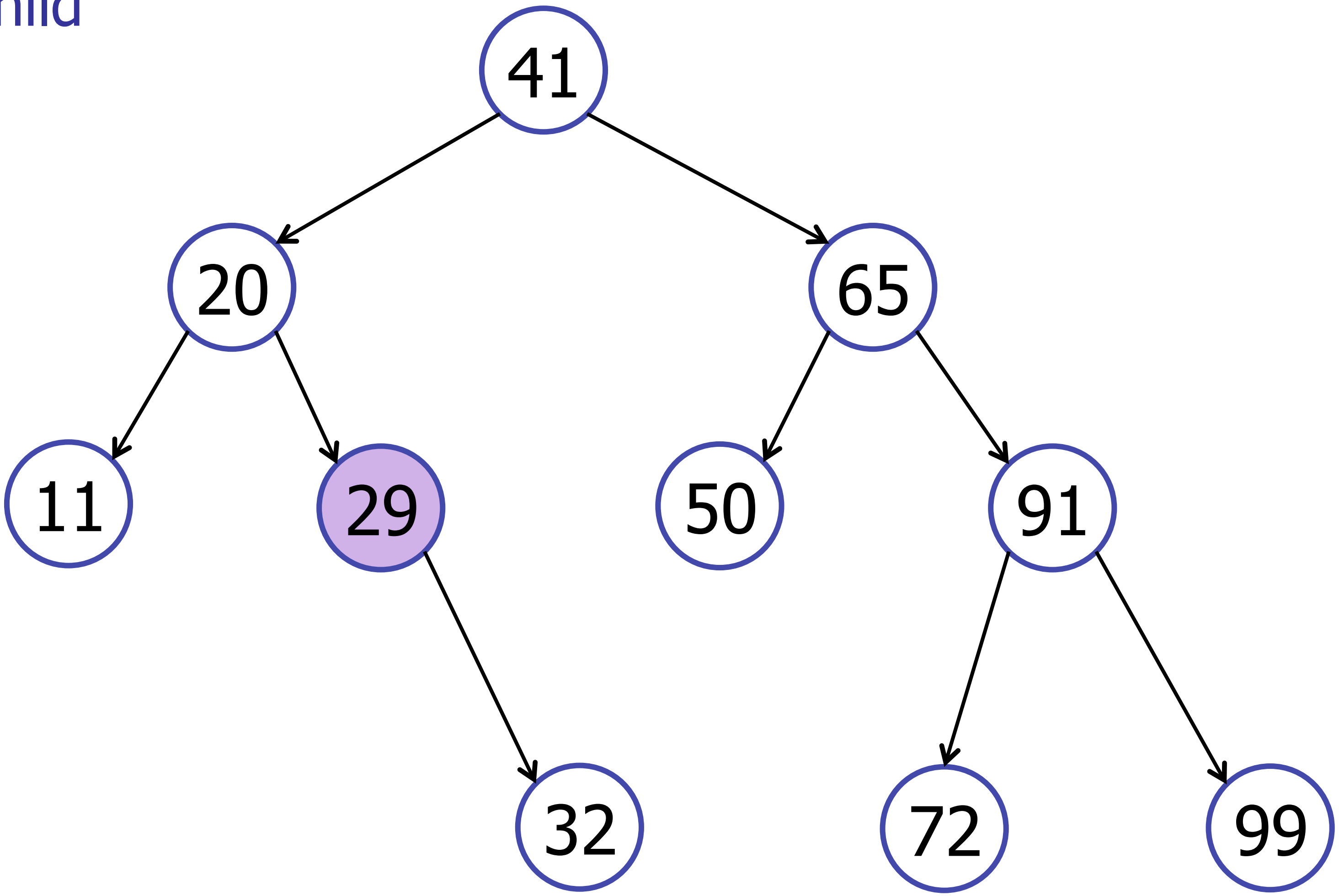
Case 1: No children



Binary Search Tree

delete(29)

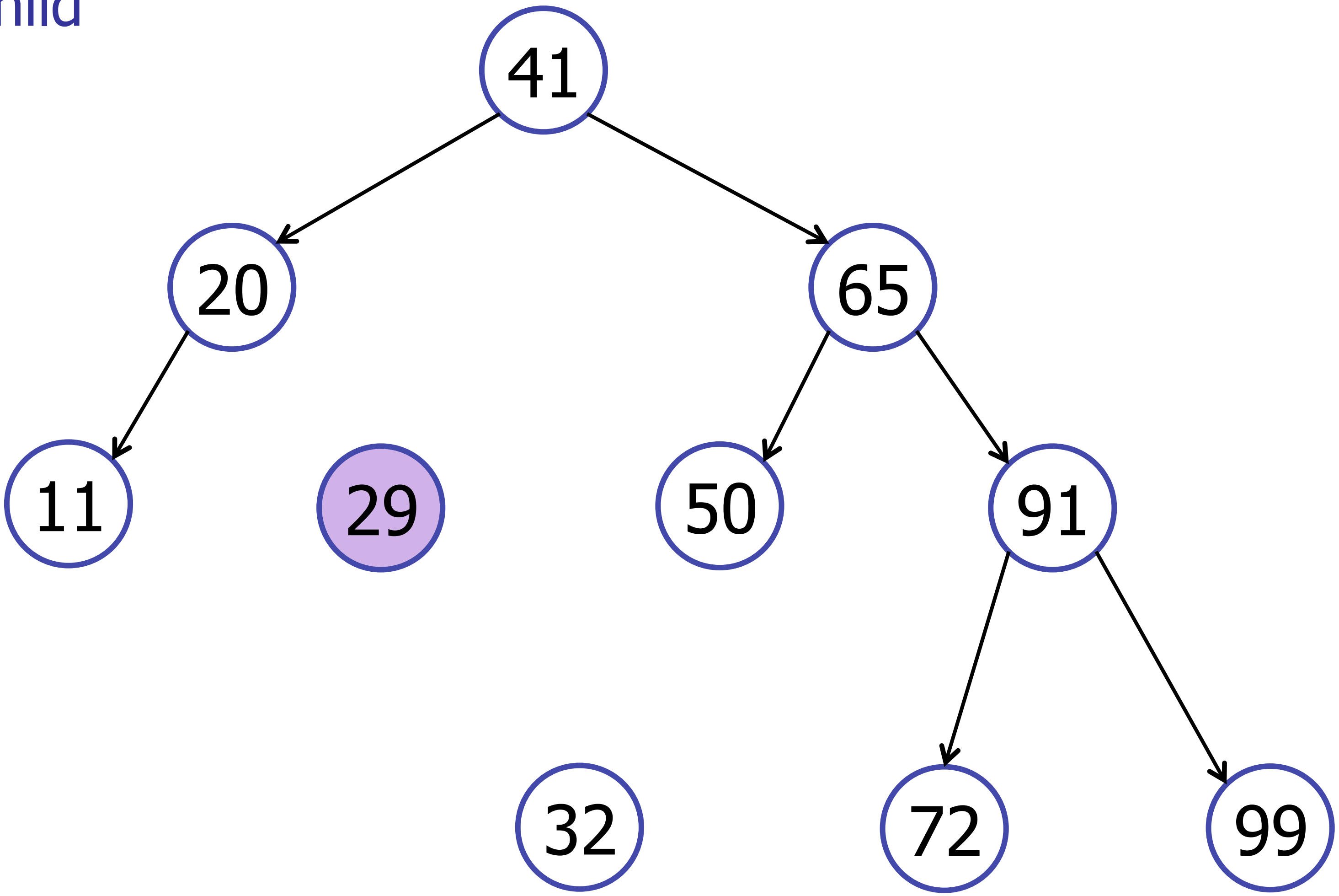
Case 2: 1 child



Binary Search Tree

delete(29)

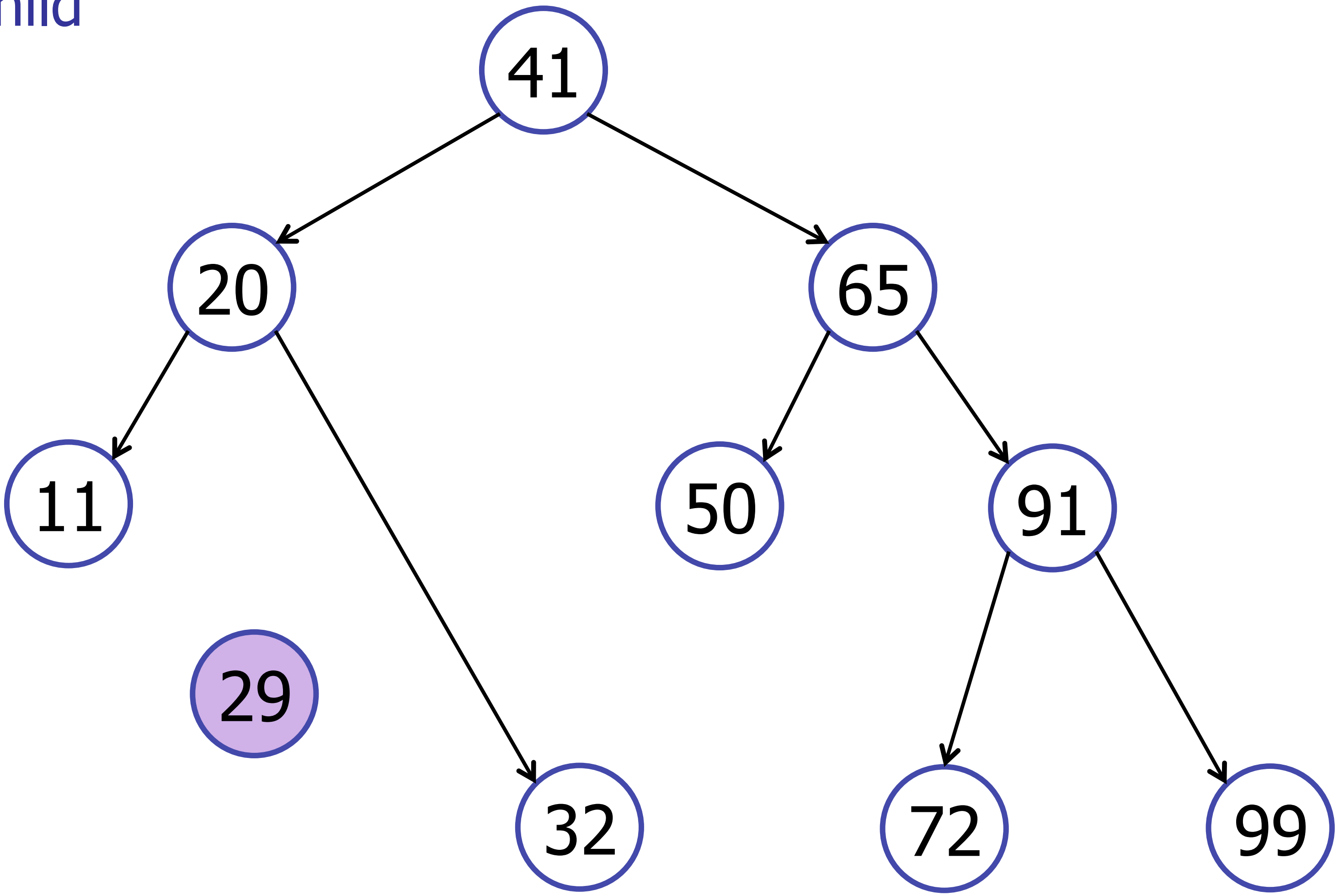
Case 2: 1 child



Binary Search Tree

delete(29)

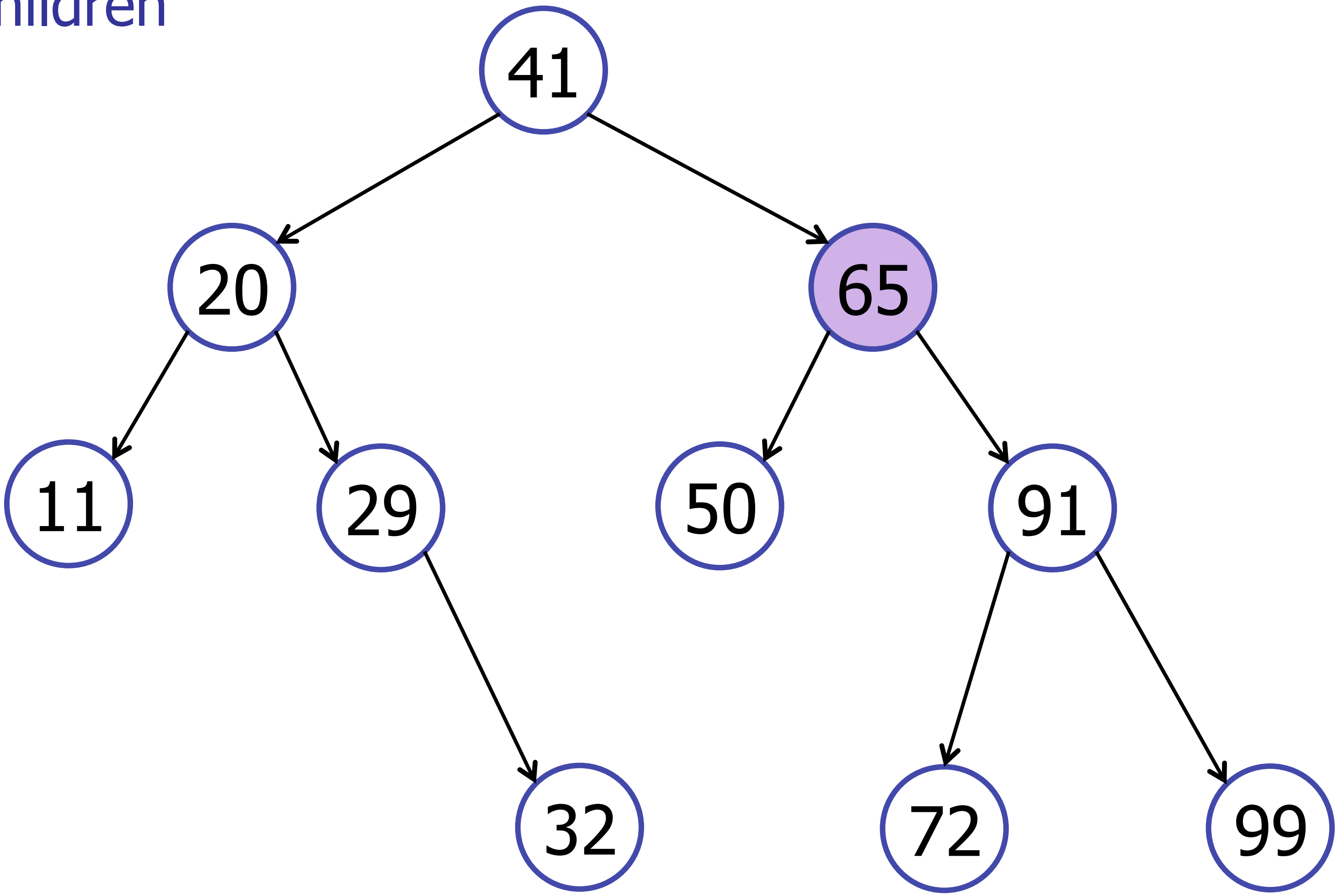
Case 2: 1 child



Binary Search Tree

delete(65)

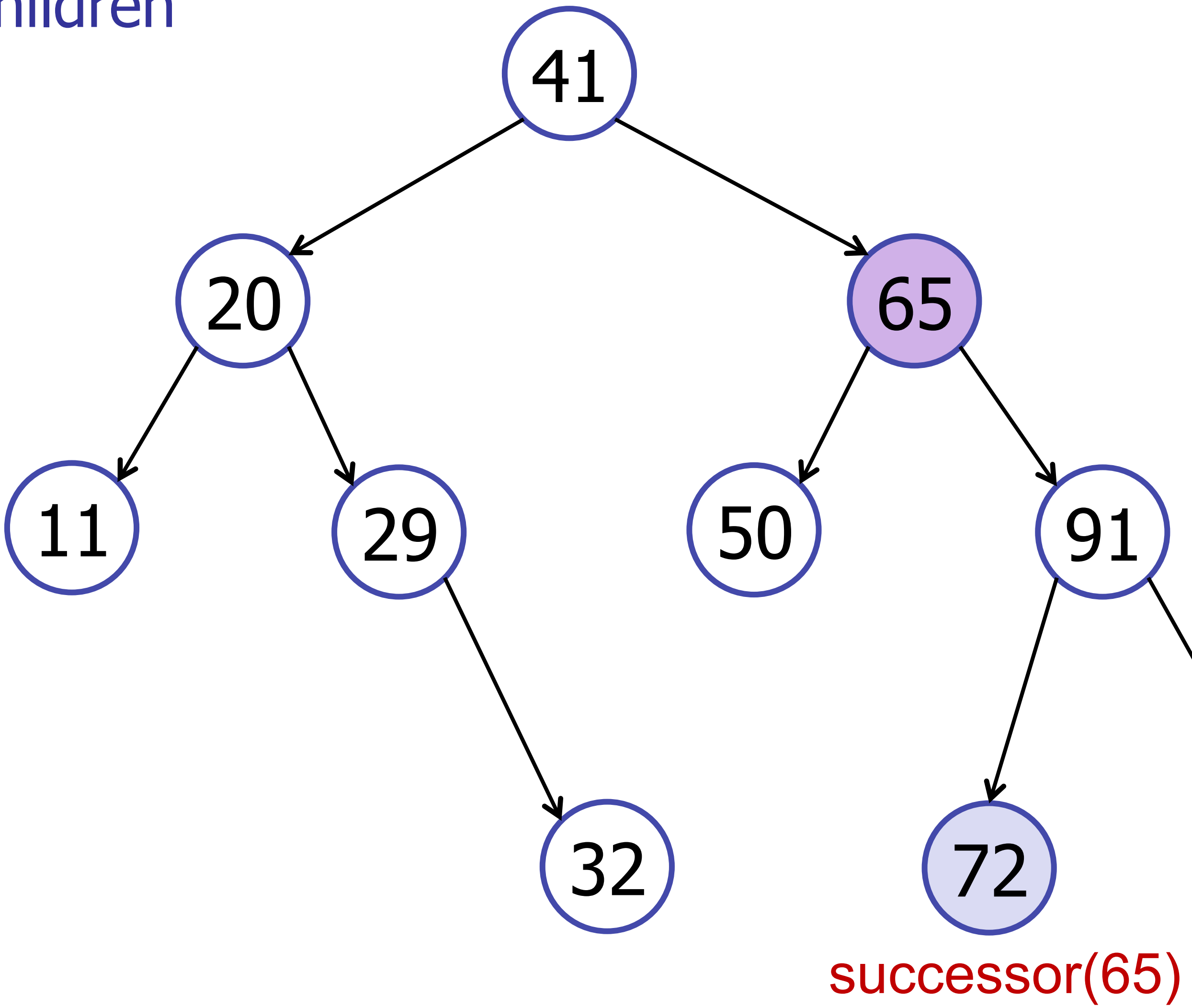
Case 3: 2 children



Binary Search Tree

delete(65)

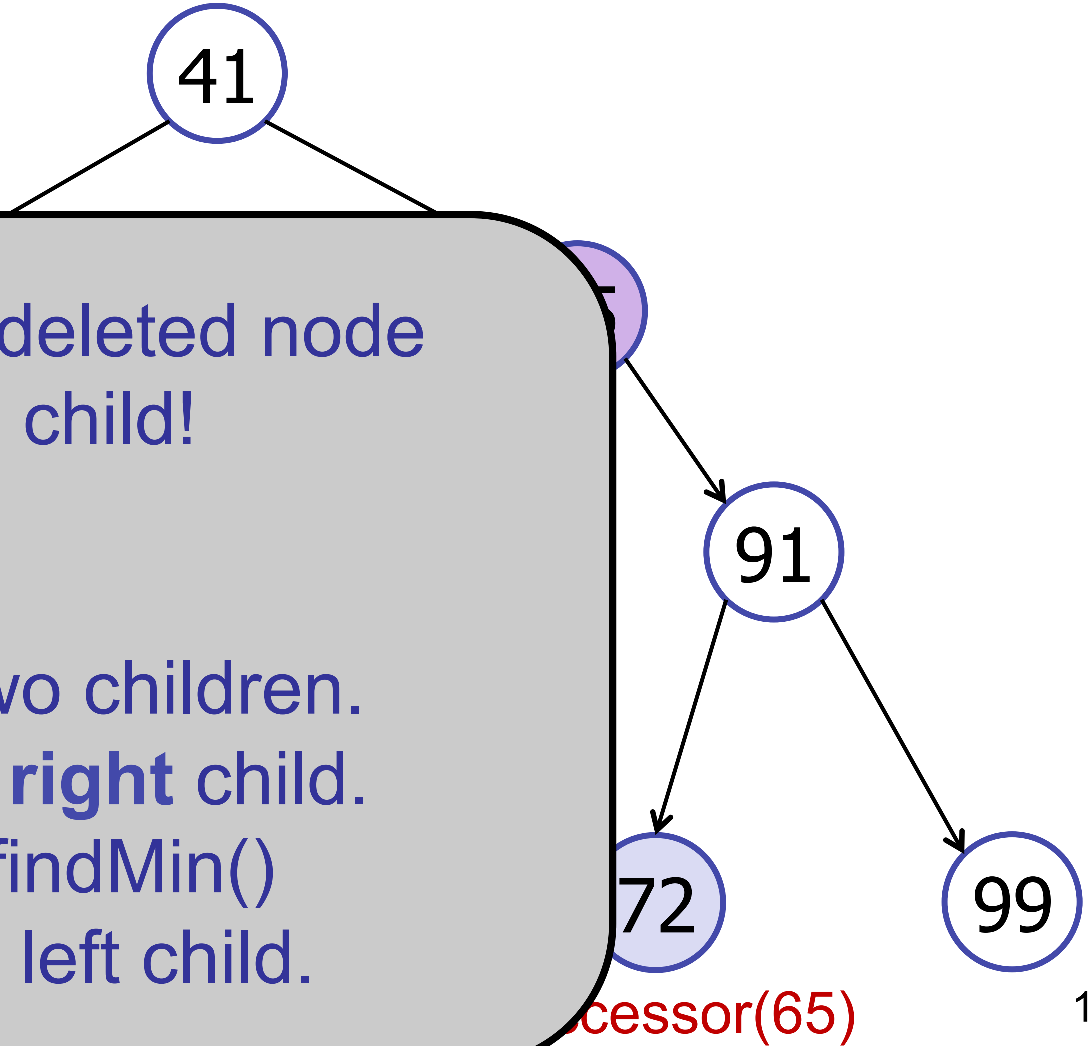
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



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

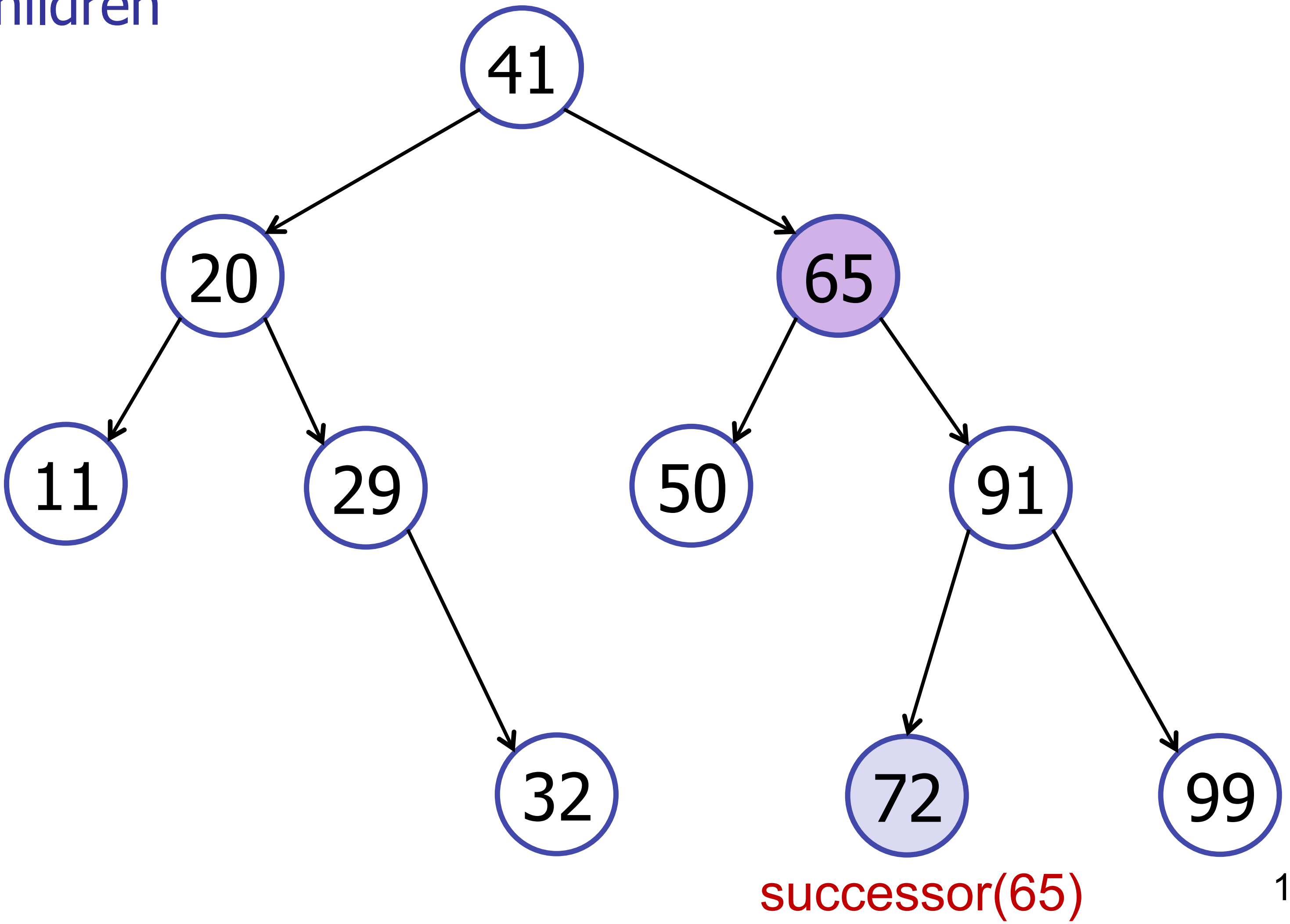
Proof:

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

Binary Search Tree

delete(65)

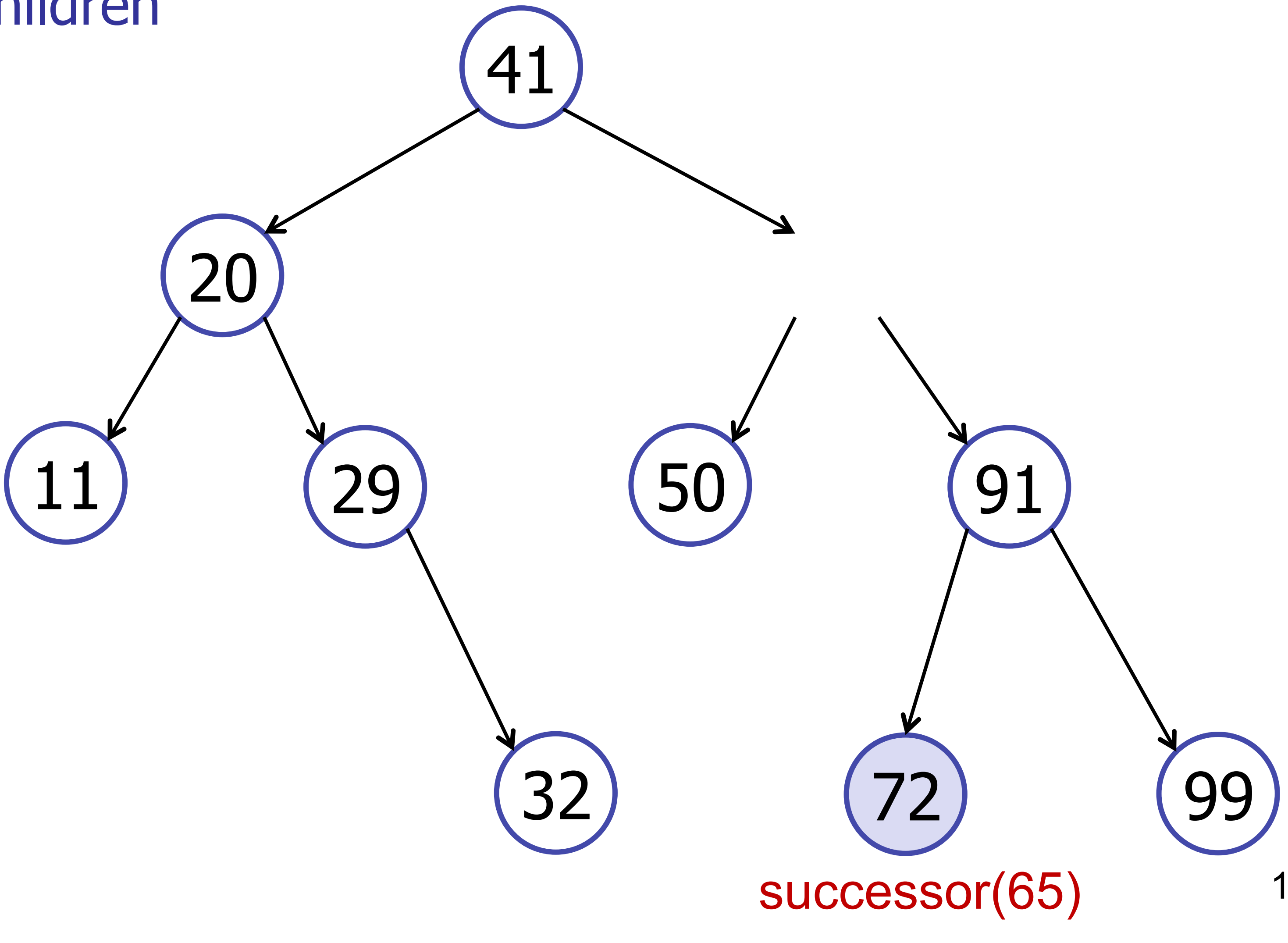
Case 3: 2 children



Binary Search Tree

delete(65)

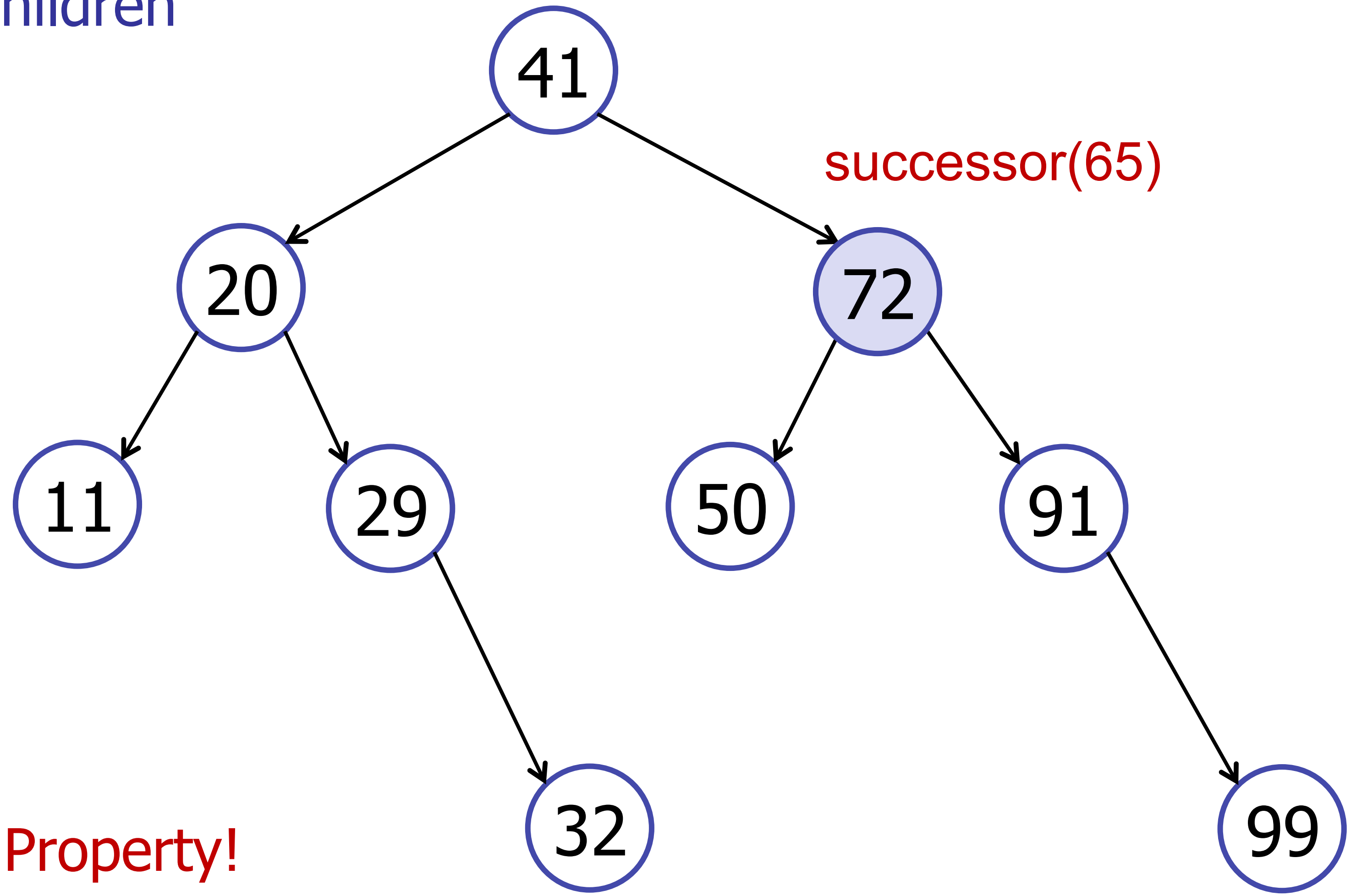
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



Check BST Property!

Binary Search Tree

delete(v) Running time: O(h)

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

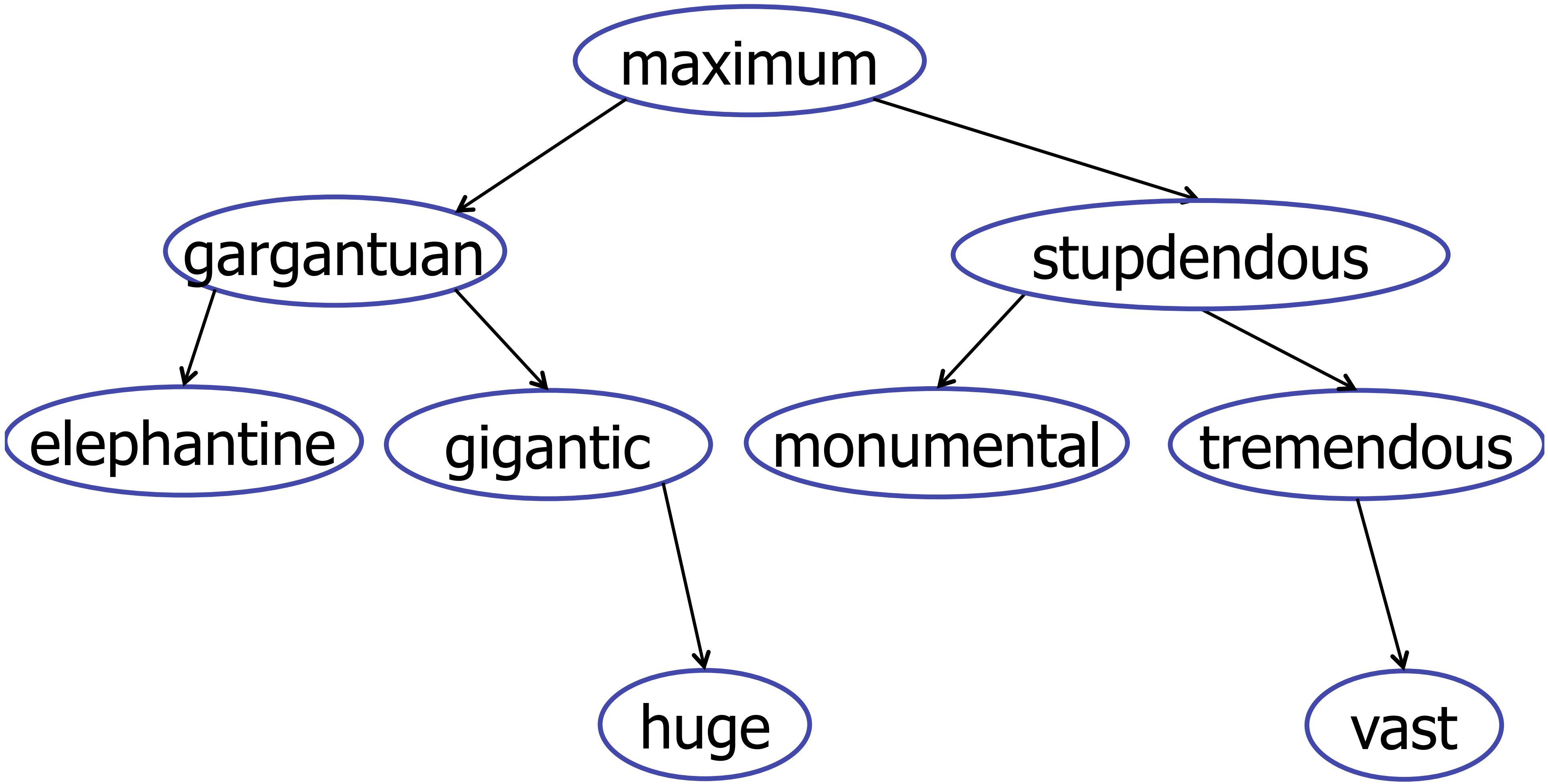
Modifying Operations

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

Query Operations:

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

What about text strings?



Implement a searchable dictionary!

What about text strings?

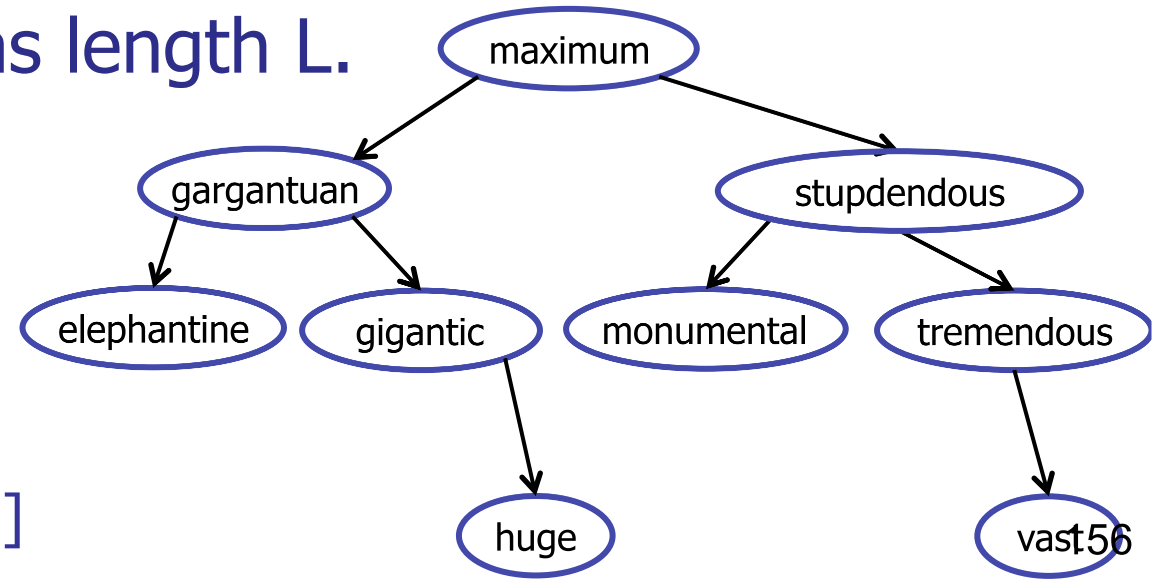
Cost of comparing two strings:

- Cost[A.compareTo(B)] = min(A.length, B.length)
- Compare strings letter by letter (?)

Cost of tree operation:

- Assume string has length L.
- Cost:

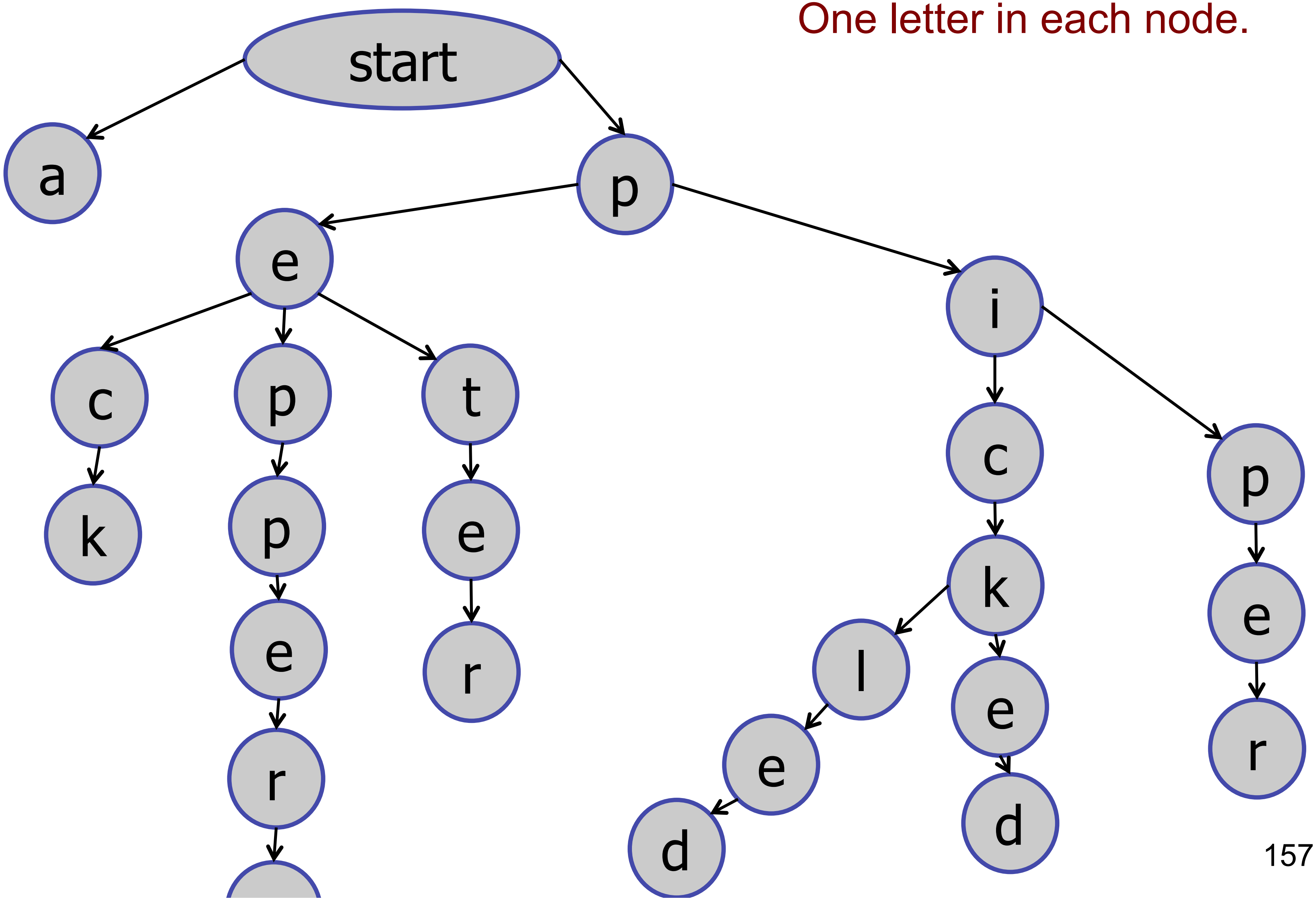
  O(hL)



[Optimizations are possible.]

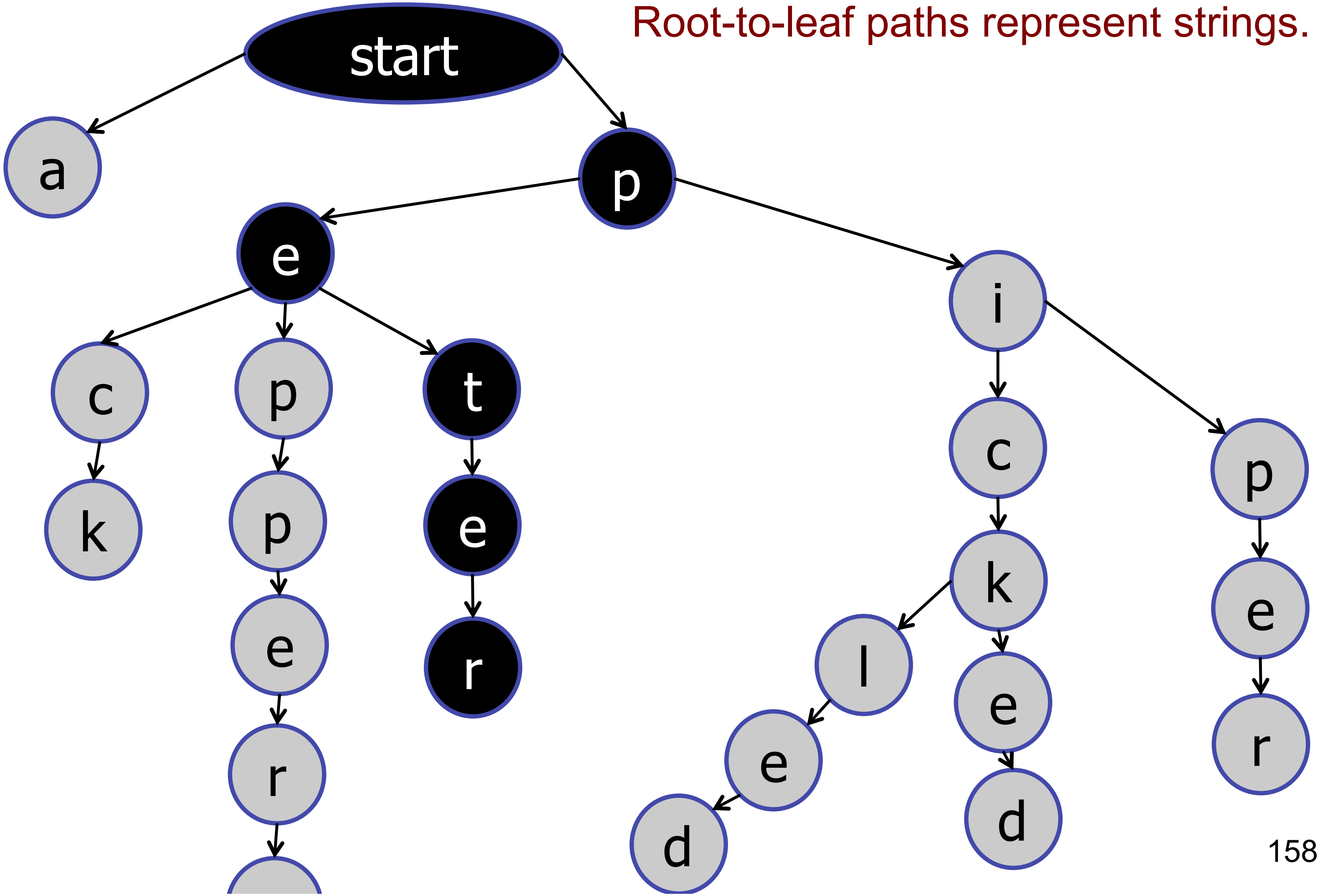
Trie [prounounced: try]

One letter in each node.



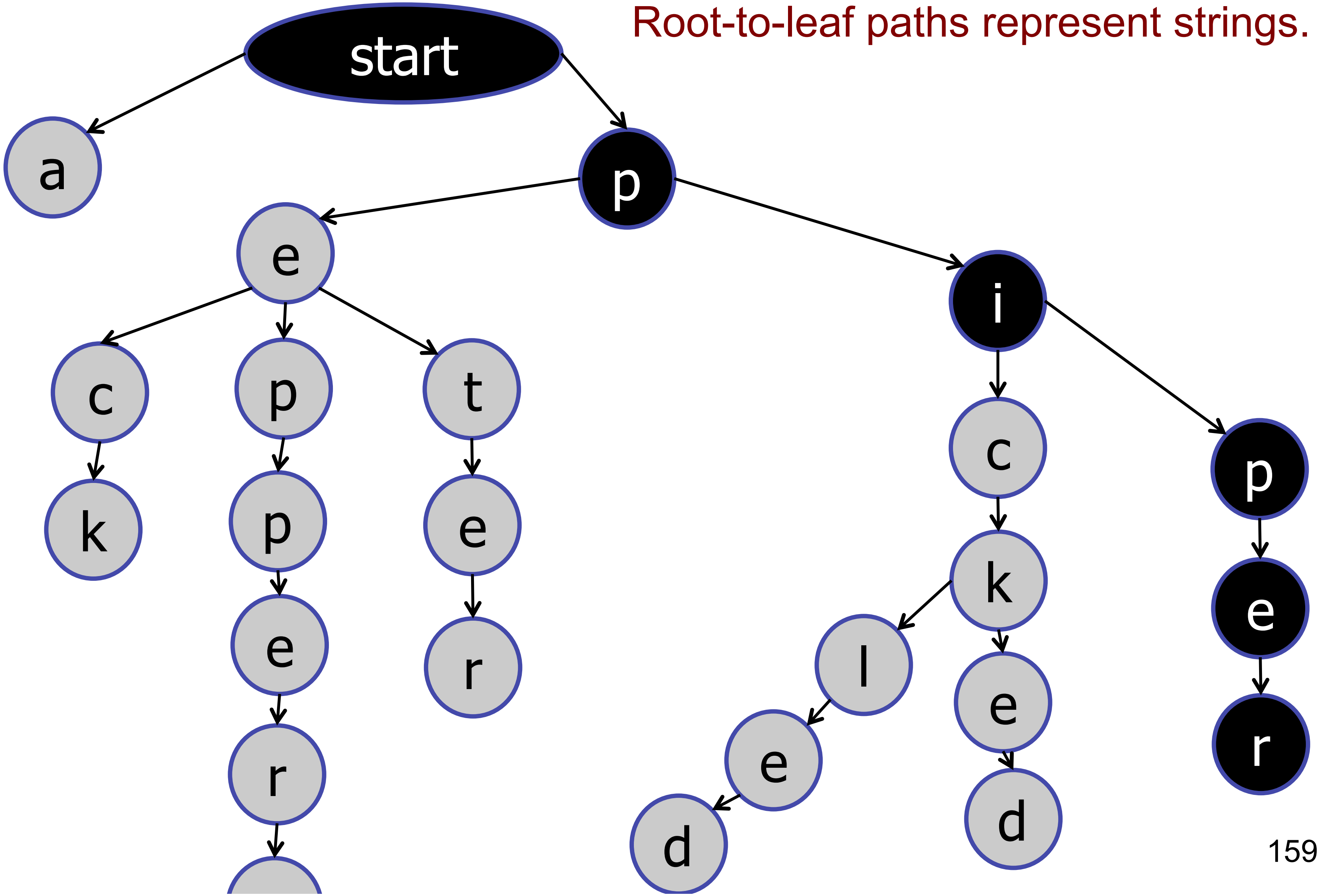
Trie [prounounced: try]

Root-to-leaf paths represent strings.



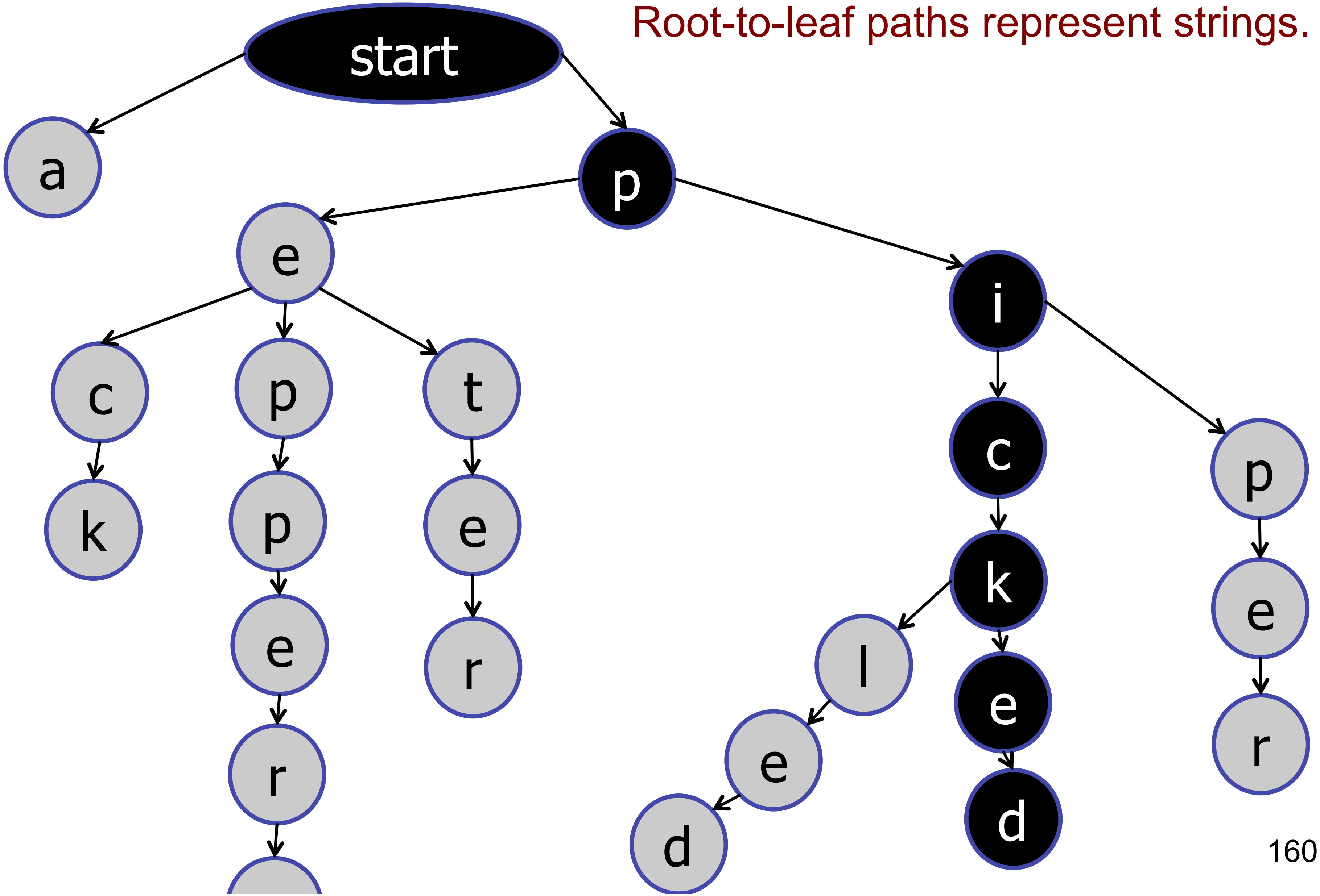
Trie [prounounced: try]

Root-to-leaf paths represent strings.



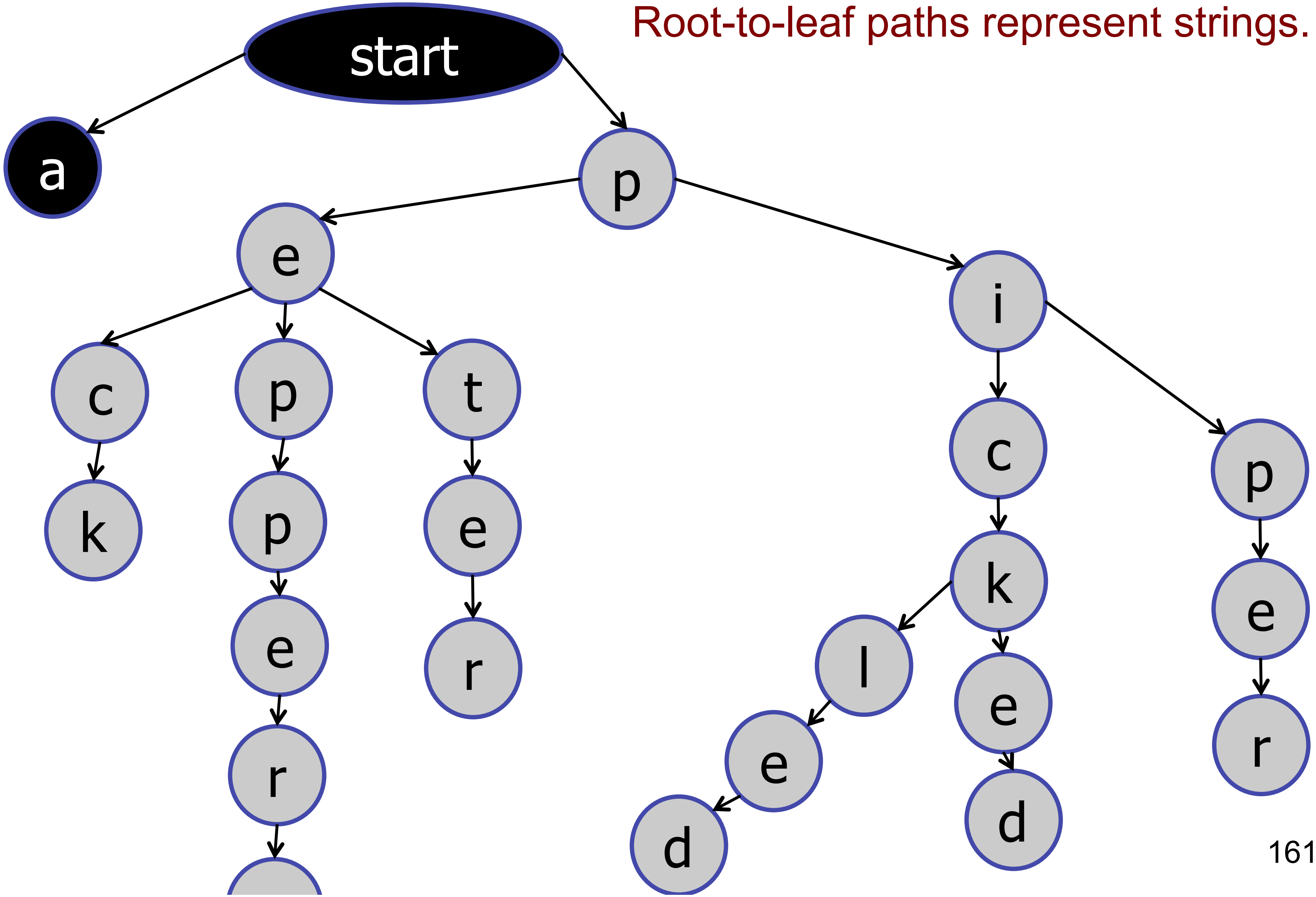
Trie [prounounced: try]

Root-to-leaf paths represent strings.



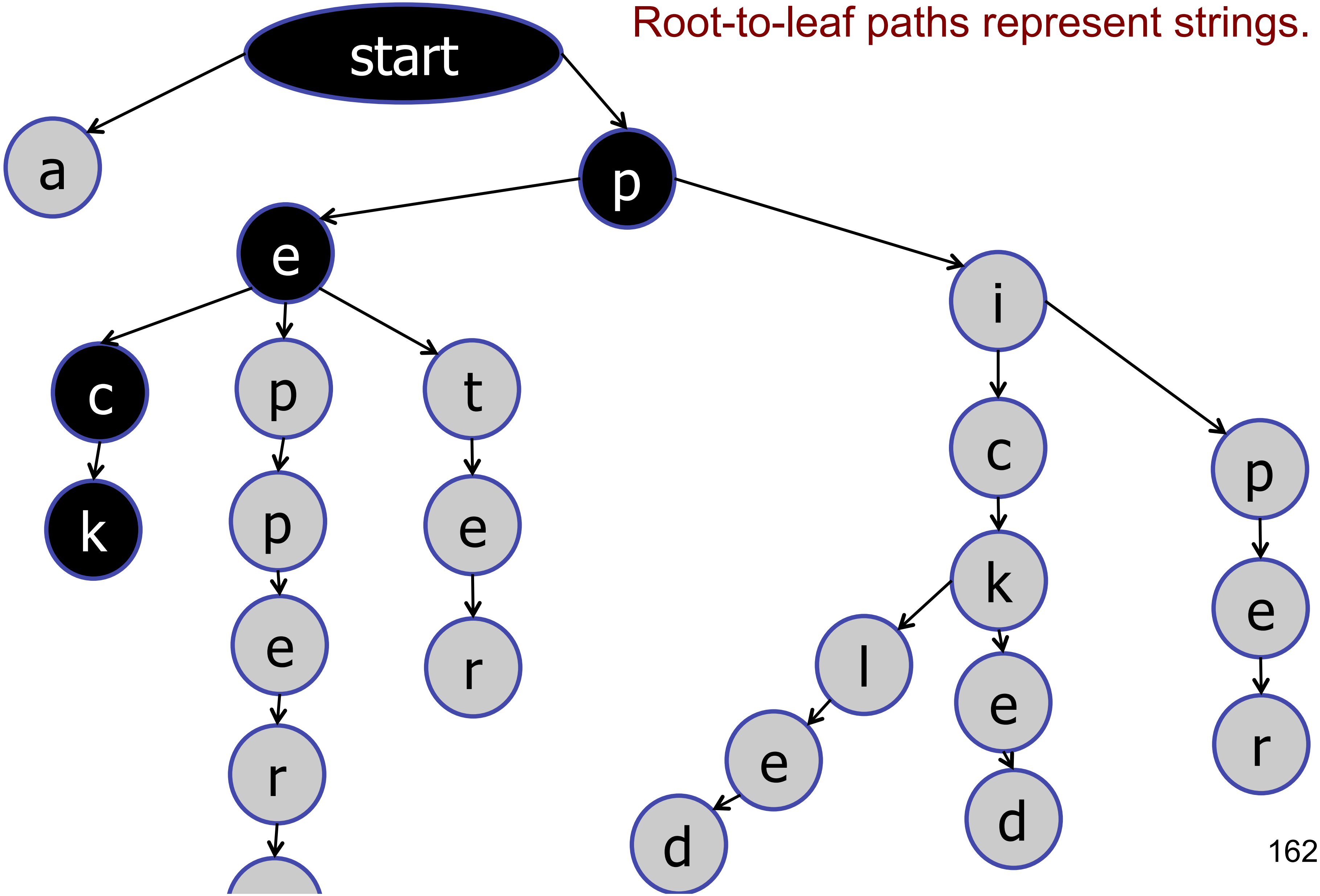
Trie [prounounced: try]

Root-to-leaf paths represent strings.



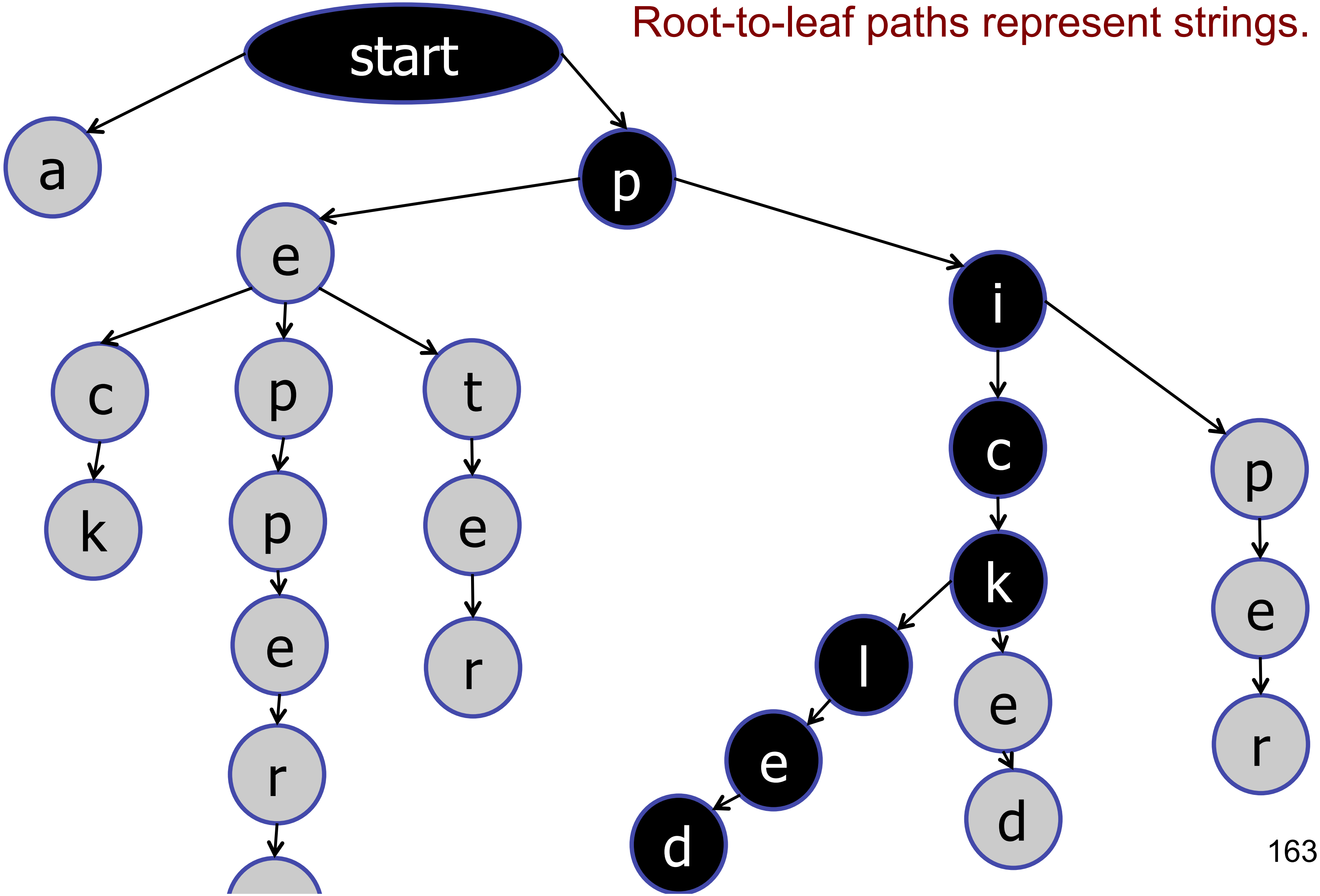
Trie [prounounced: try]

Root-to-leaf paths represent strings.



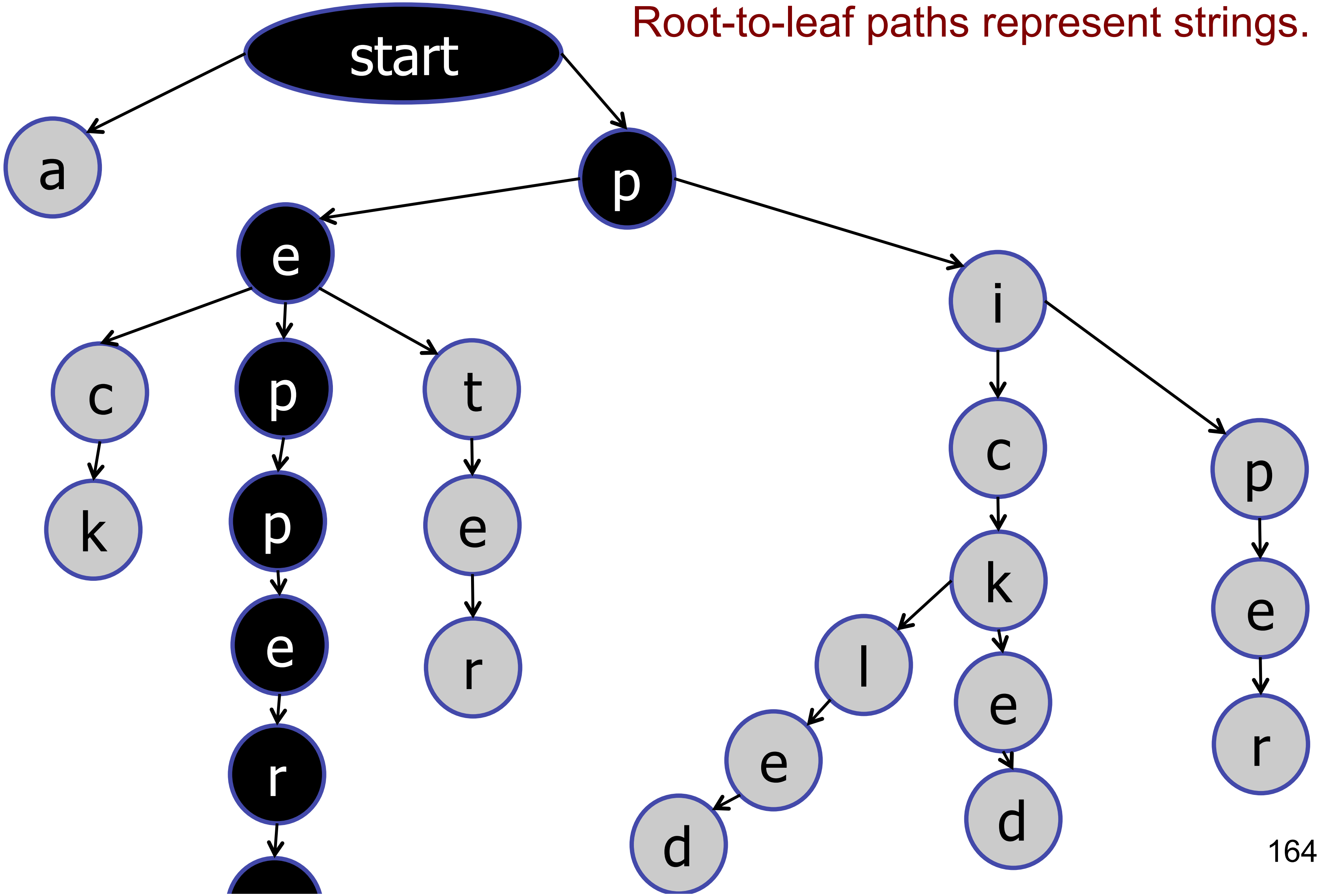
Trie [prounounced: try]

Root-to-leaf paths represent strings.



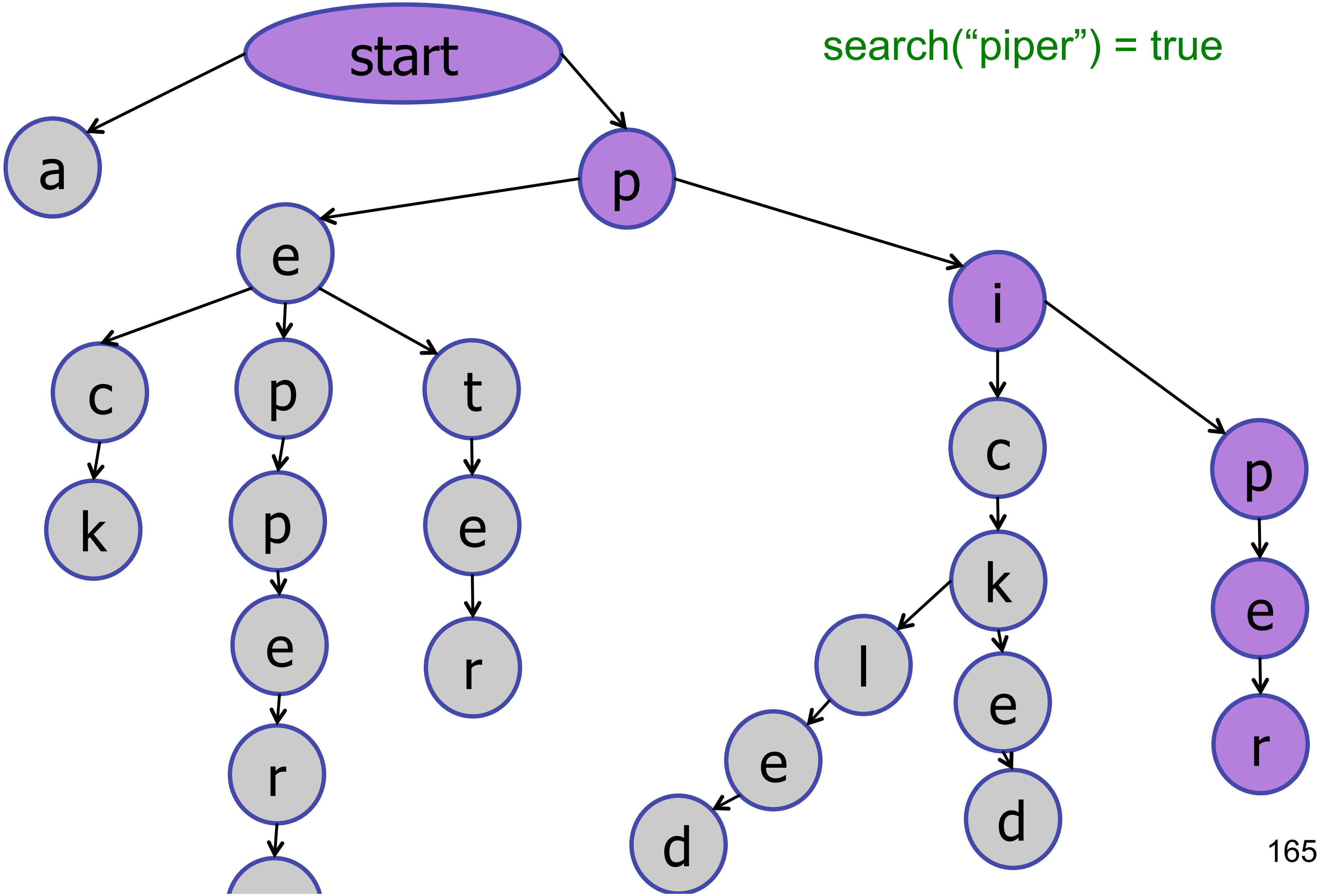
Trie [prounounced: try]

Root-to-leaf paths represent strings.

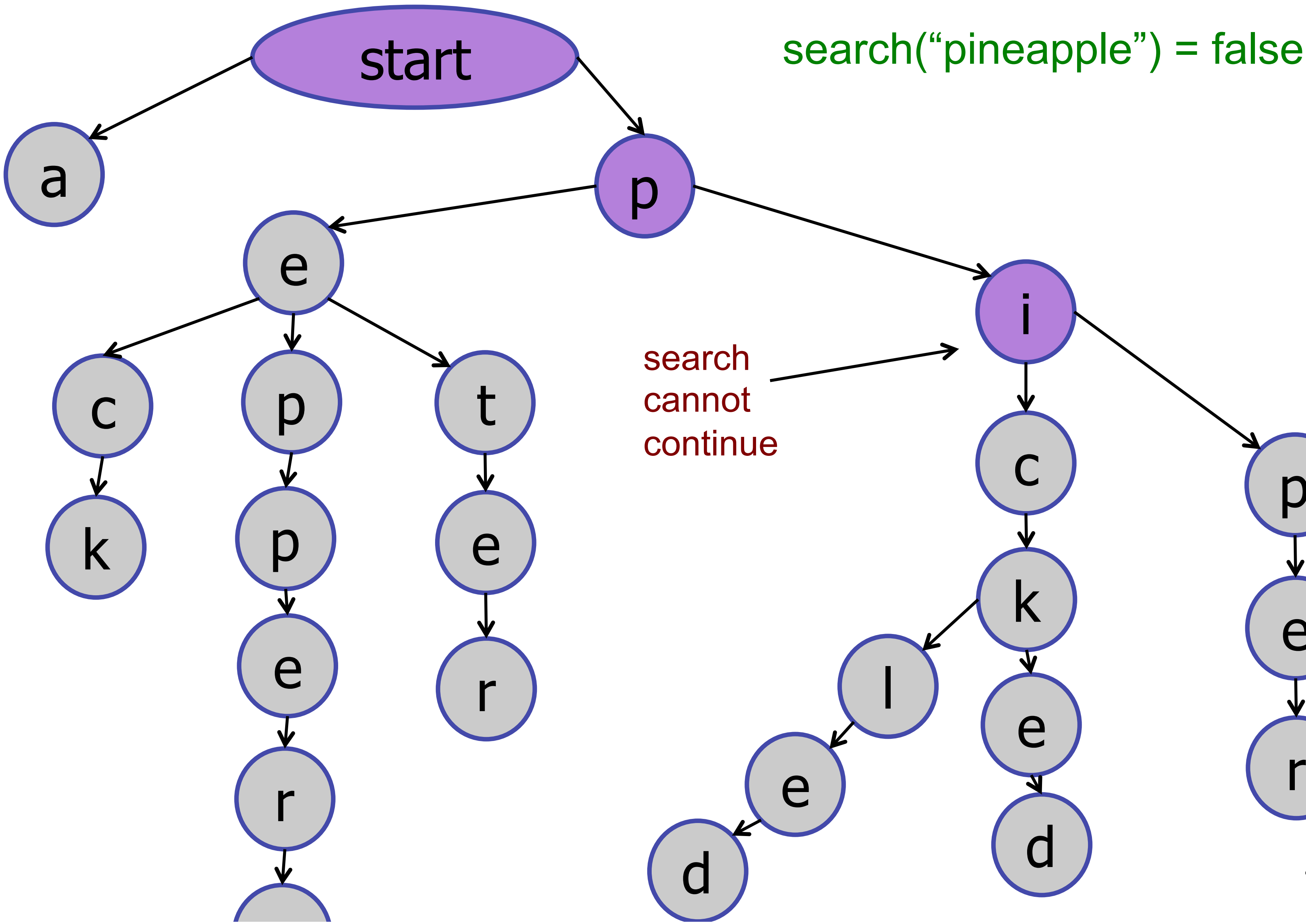


Searching a Trie

search(“piper”) = true

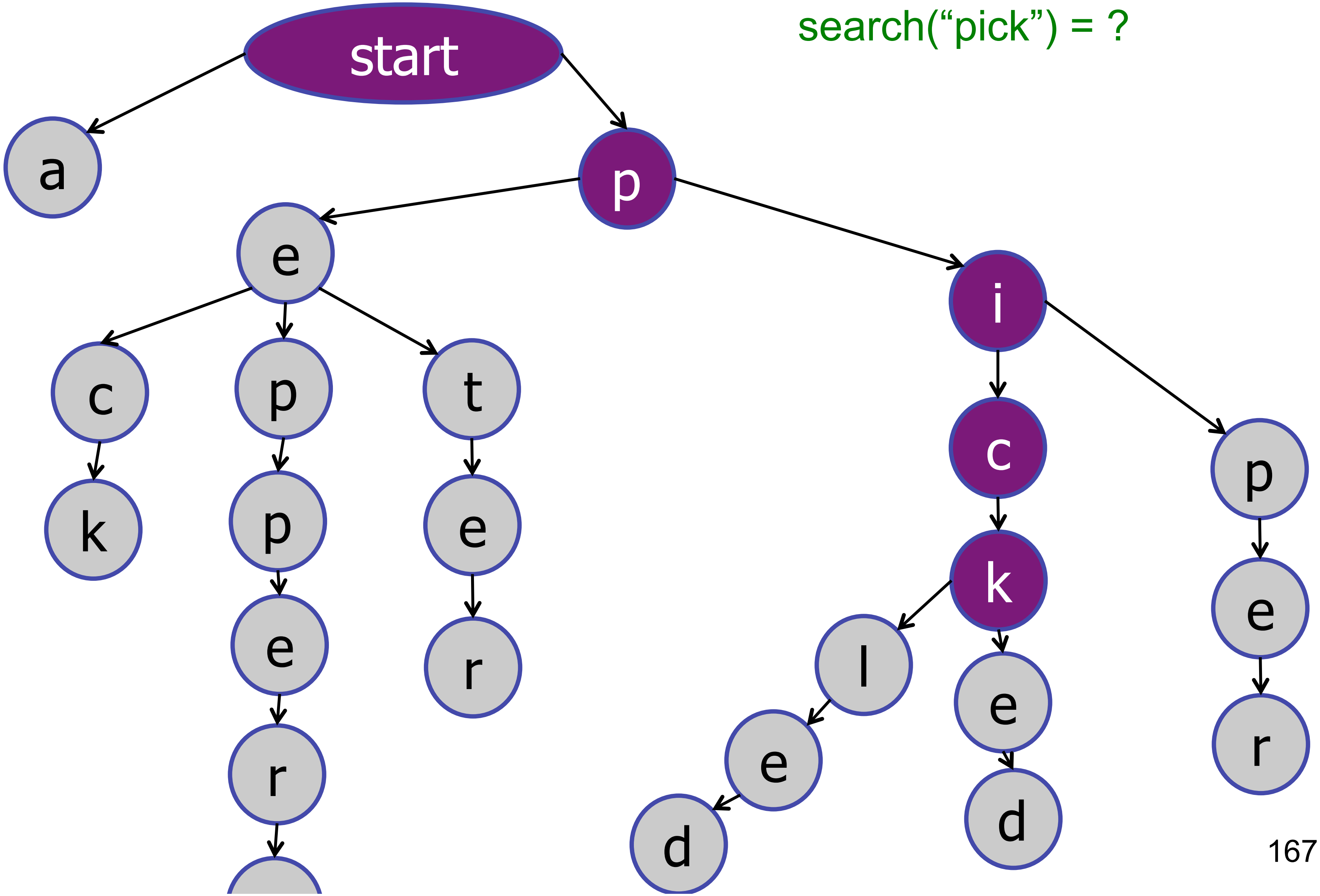


Searching a Trie



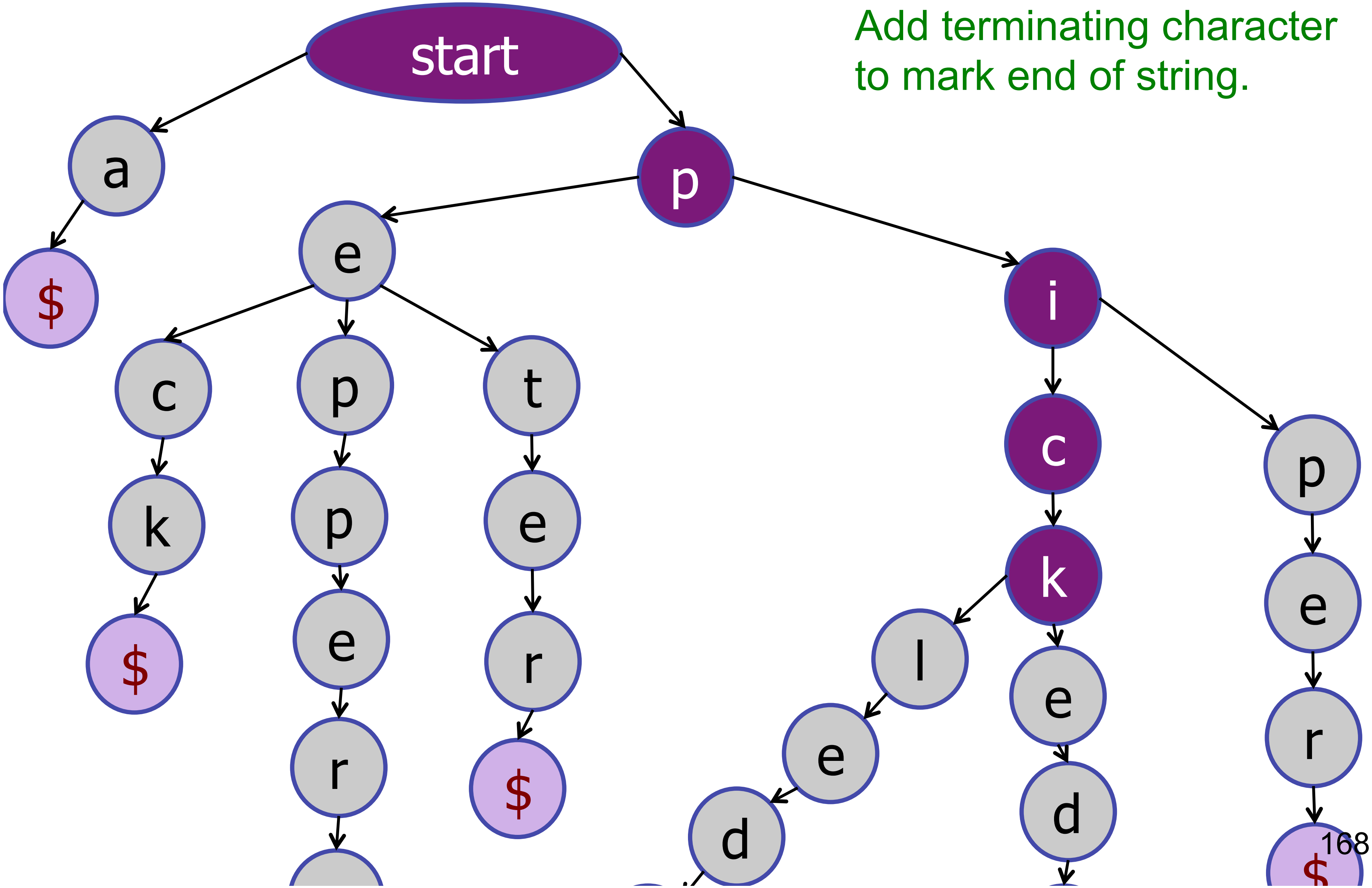
Trie Details

search(“pick”) = ?



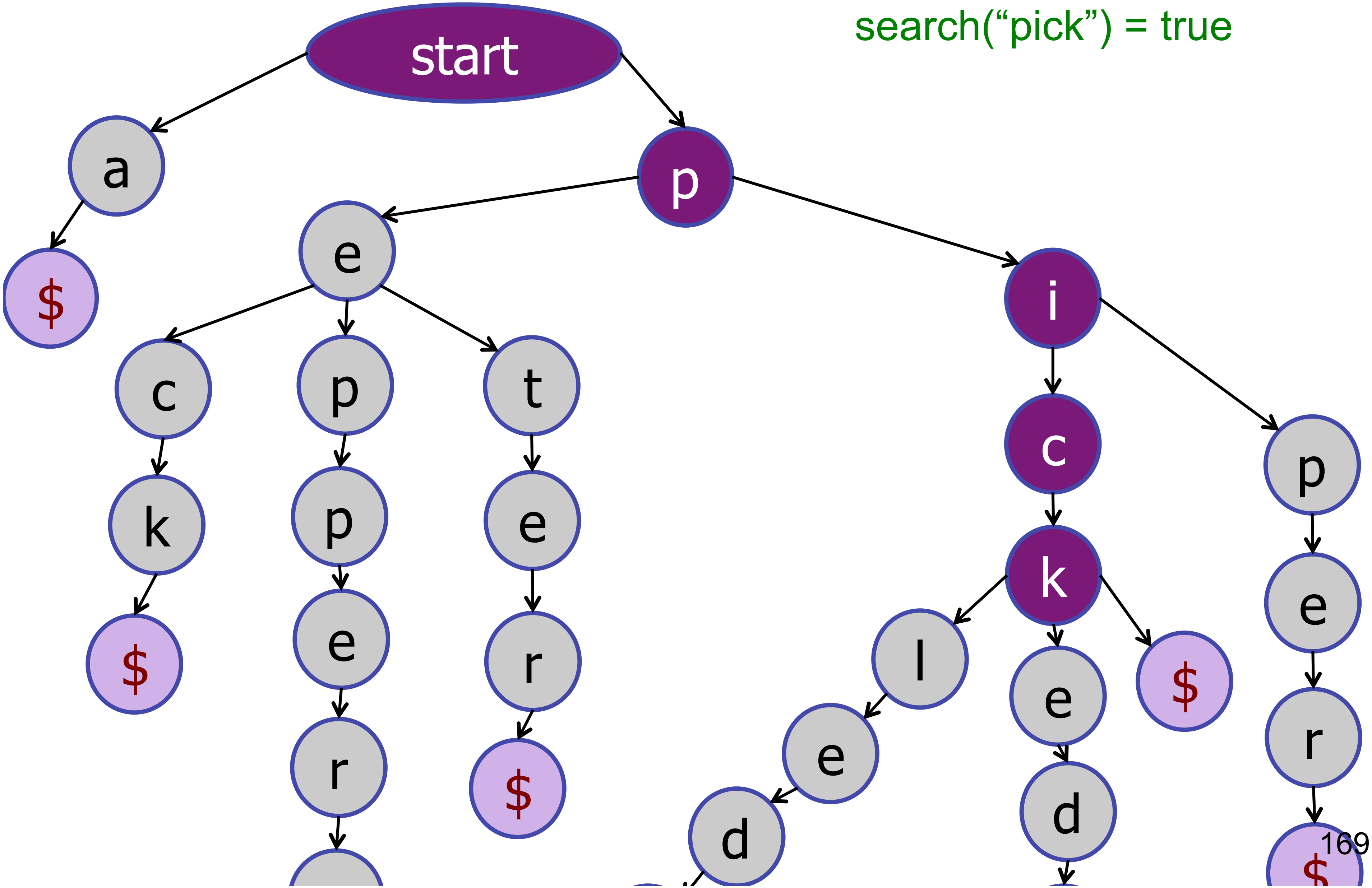
Trie Details

Add terminating character to mark end of string.



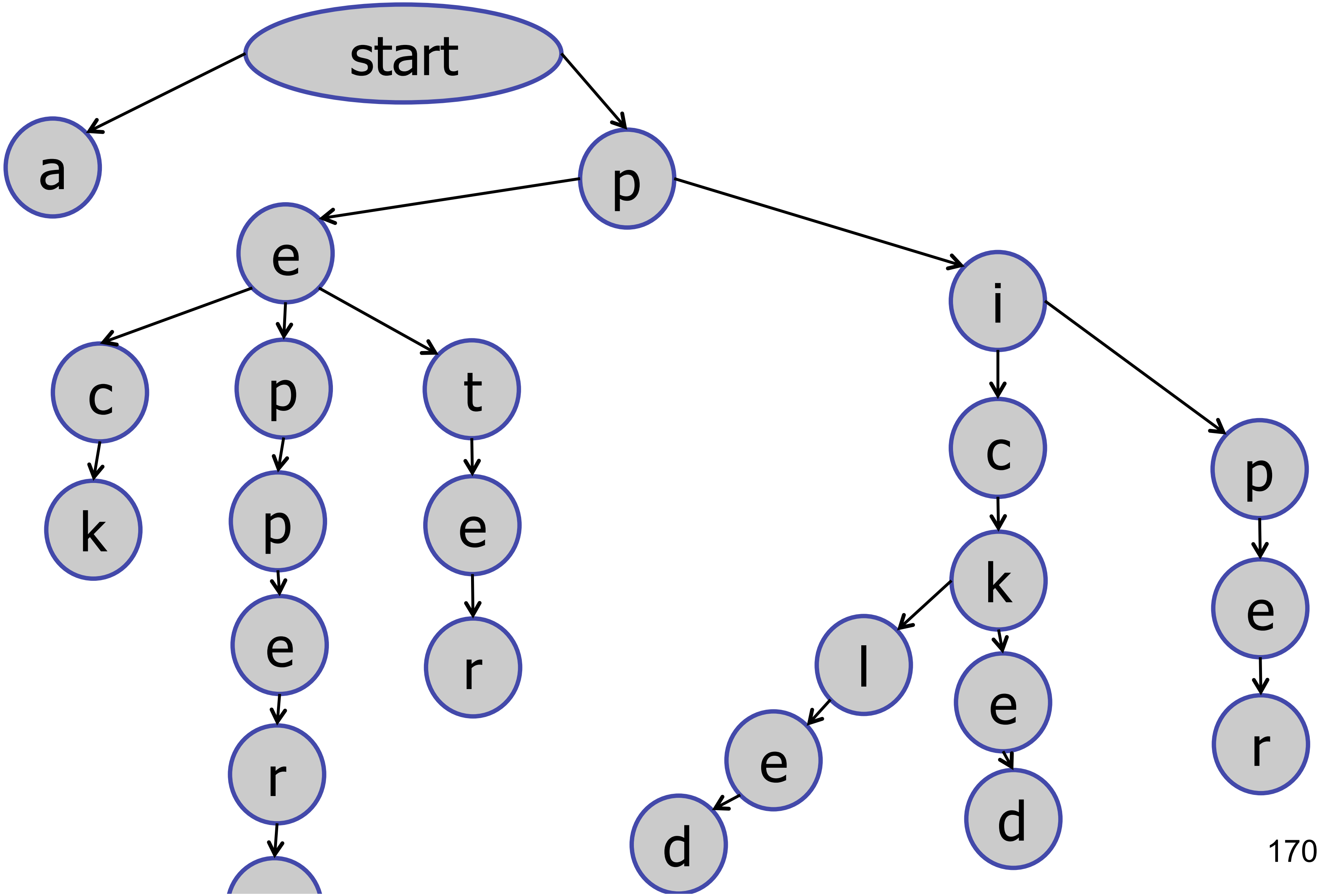
Trie Details

search(“pick”) = true



Trie

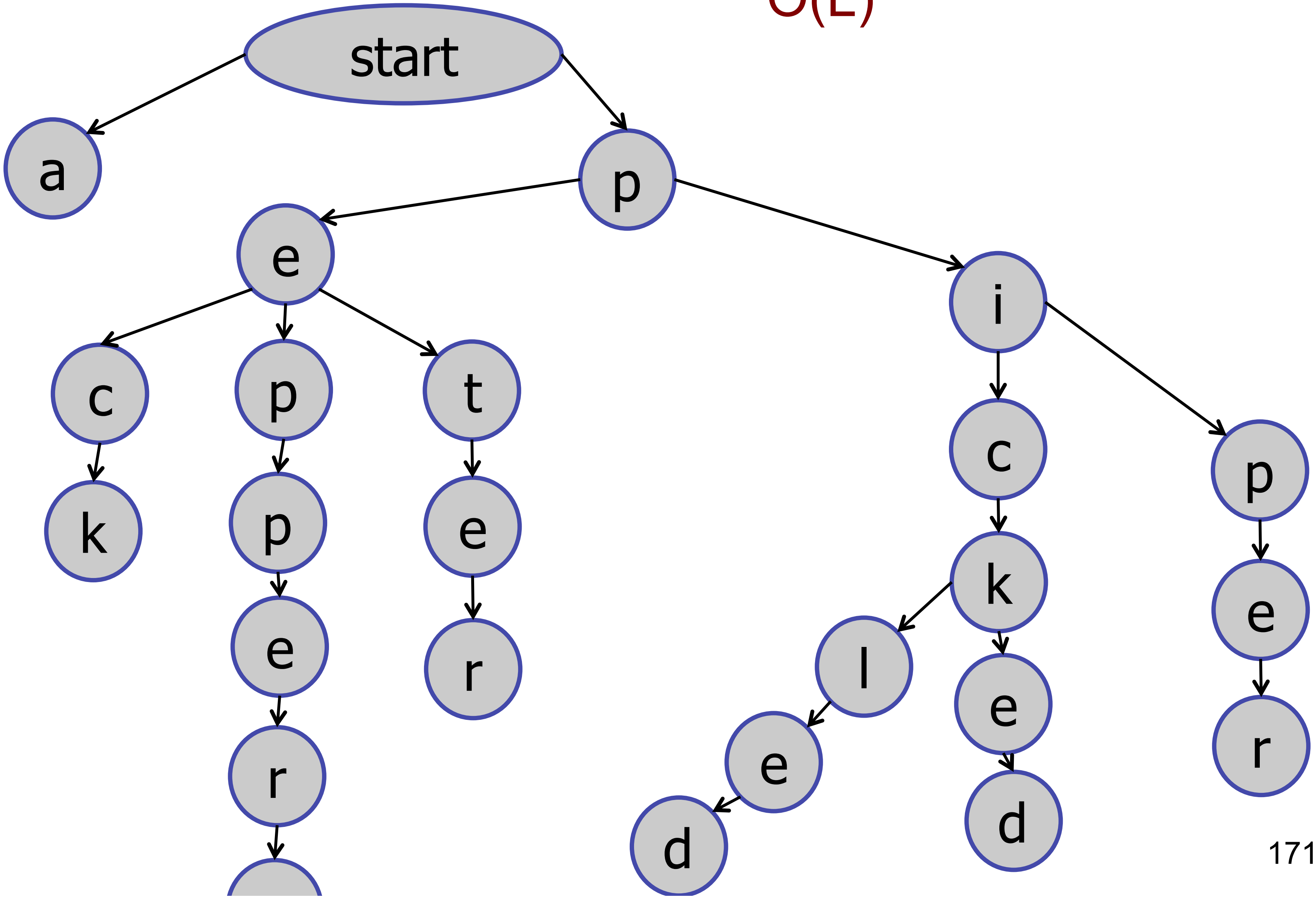
Cost to search for a string of length L?



Trie

Cost to search for a string of length L?

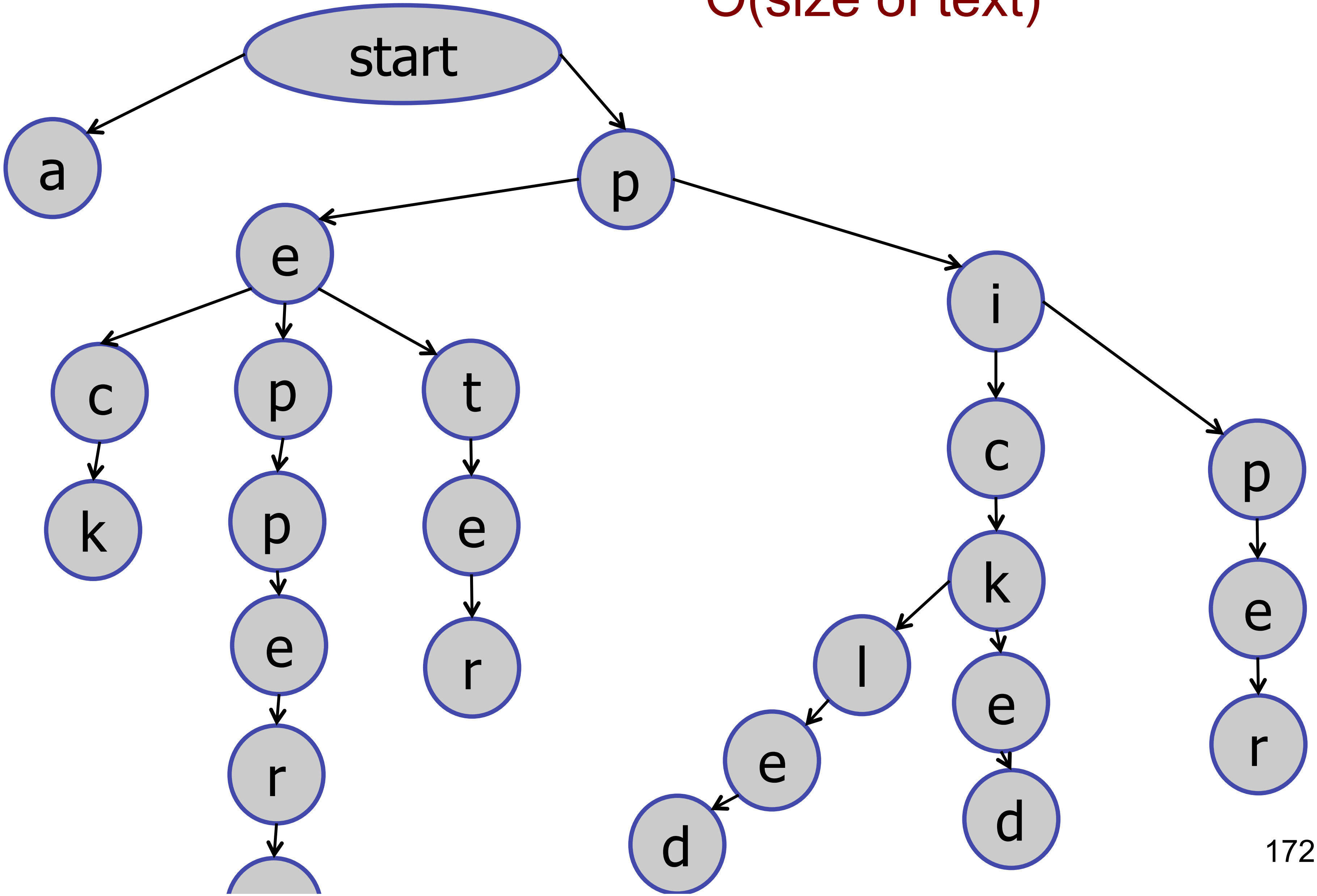
O(L)



Trie

Space for storing a try?

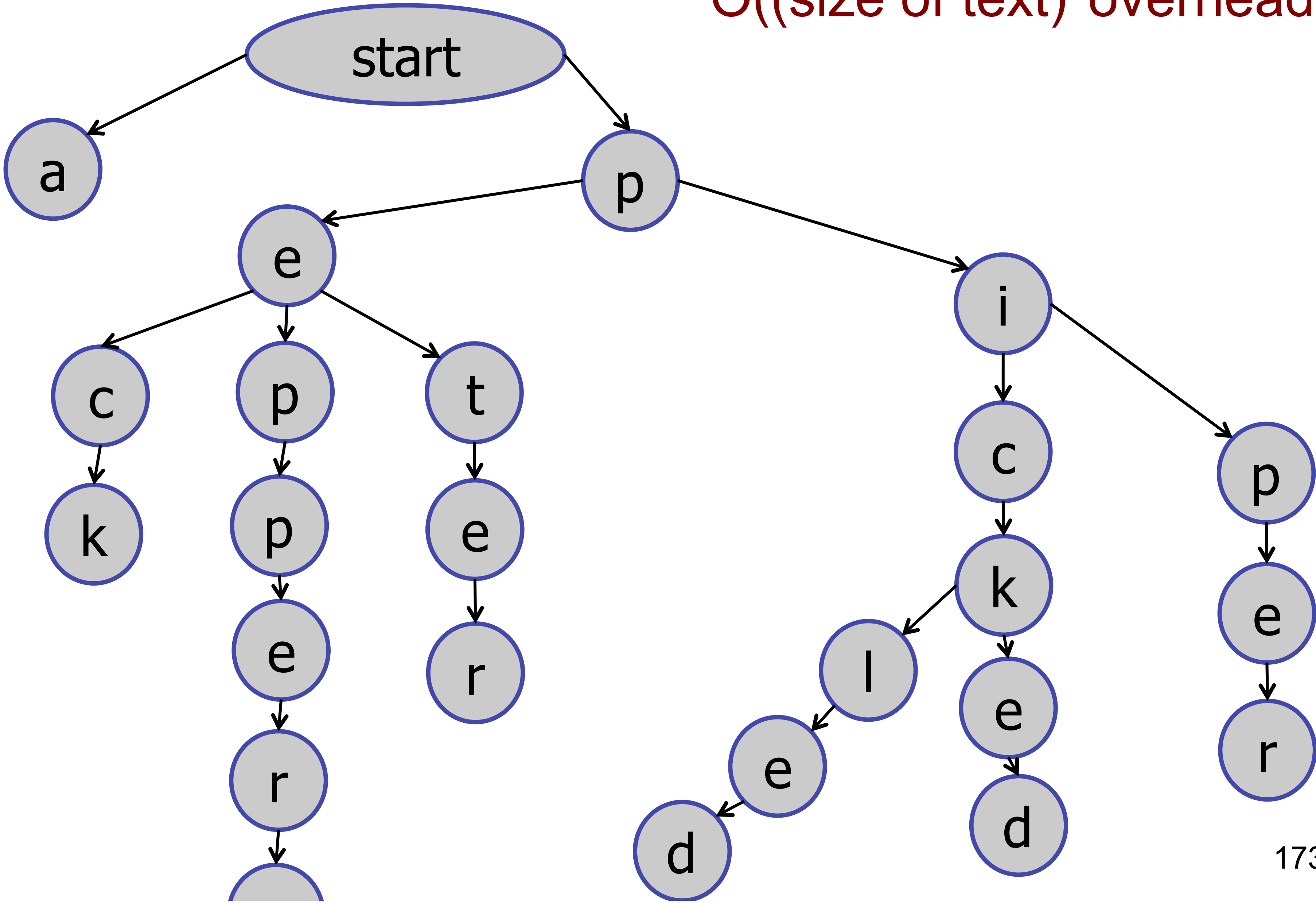
O(size of text)



Trie

Space for storing a try?

O((size of text)*overhead)



Trie Tradeoffs

Time:

- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Even faster if string is not in trie!

Trie Tradeoffs

Time:

- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Space:

- Trie tends to use more space.
- BST and Trie use O(text size) space.
- But Trie has more nodes and more overhead.

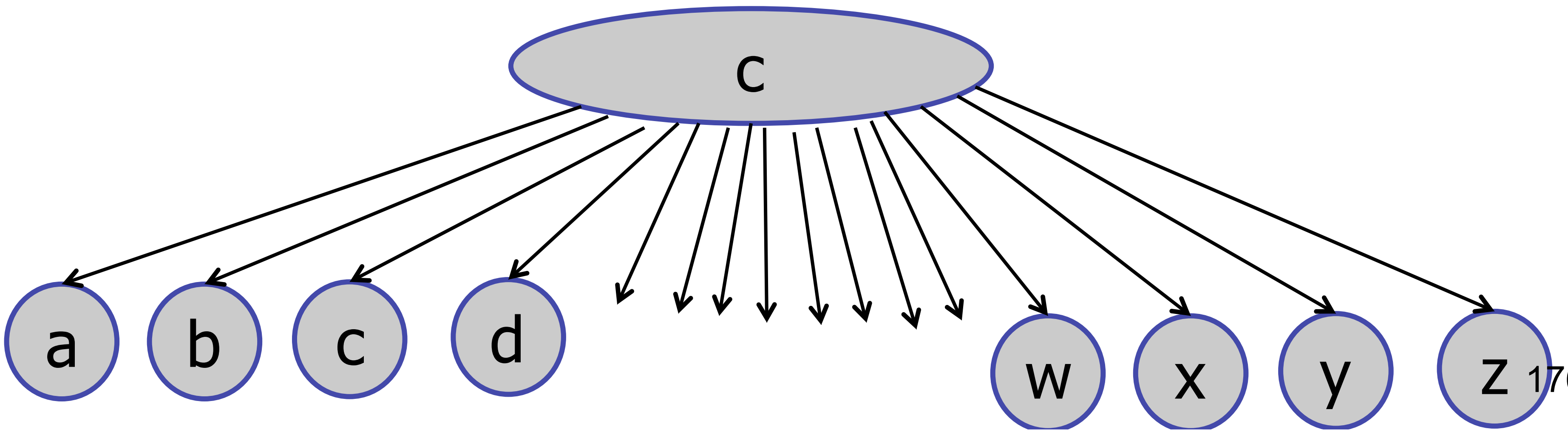
Trie Space

Trie node:

- Has many children.
- For strings: fixed degree.
- Ascii character set: 256

wasted space?

```
TrieNode children[] = new TrieNode[256];
```



Trie Applications

String dictionaries

- Searching
- Sorting / enumerating strings

Partial string operations:

- Prefix queries: find all the strings that start with pi.
- Long prefix: what is the longest prefix of "pickling" in the trie?
- Wildcards: find a string of the form "pi??le" in the trie.

Announcements

Quiz 1 : February 12

- In class: be there!
- Be on time.
- Covers material through today's lecture

Bring to quiz:

- One sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else.