LECTURE 7: BINARY SEARCH TREES

Harold Soh
harold@comp.nus.edu.sg

ADMINISTRATIVE ISSUES: PROBLEM SET 2

Start 2019-09-08 15:59 UTC **Problem Set 2** **End** 2019-09-23 16:00 UTC

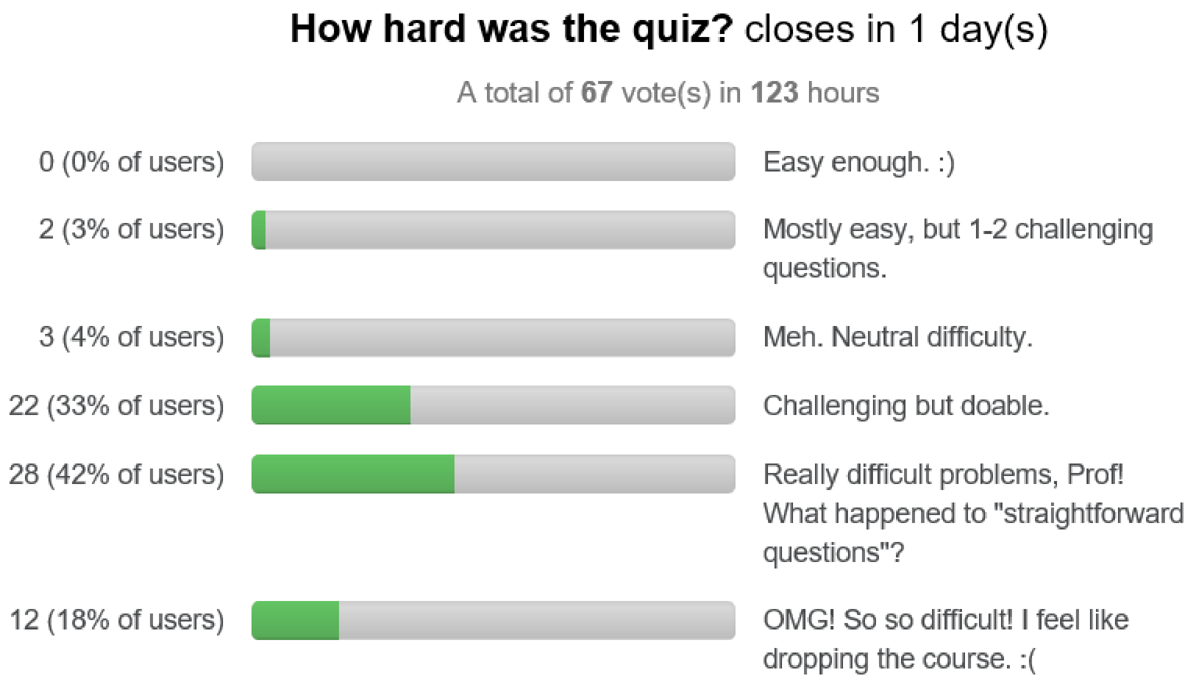
Time elapsed 14:19:35 **Time remaining** 345:40:26

- Deadline is Monday 23rd 23:59:19
- 3 graded problems
  - Kattis Quest
  - Cookie Selection
  - GCPC
- 2 challenge (ungraded) problems
  - BST
  - Shortsell

ADMINISTRATIVE ISSUES

Quiz 1

- Still grading
- "Absorbed material" means knowing how to apply and change alg/data structure to match problem.



QUIZ 1: SOME COMMON ISSUES

Problem 1 (Asymptotic Analysis):

- Problem 1c is a bonus. (requires induction to prove)
- Practice needed with Big-O
- We will organize a special class

Problem 2 (RPN) and **Problem 3** (Quicksort):

- Overly complex algorithms
- We will focus on associations and problem-solving techniques

General:

- Do not need to write too much.
  - Trying to smoke through will probably fail. :P
- Don’t expect us to “extrapolate” your answer.

LEARNING OUTCOMES

By the end of this session, students should be able to:

- Describe the **Binary Search Tree (BST)** and its operations
- Analyze **performance of BST operations**
- Relate the **importance of balance in a BST** to enable efficient operations.
- Explain the **pre-order, post-order, and in-order tree traversal** algorithms.

PROBLEM: POVERTY IDENTIFICATION

The Stop-Poverty charity calls:

To provide financial aid, Help identify families:

- earning exactly $a amount
- earning less than $a amount
- earning more than $a amount

What operations would we need to perform?

THE ORDERED DICTIONARY ADT

What Data Structures can we use to implement the Dictionary ADT?

Operations with k = key, v = value:

`insert(k, v)`: inserts an element with value v and key k

`search(k)`: returns the value with key k

`delete(k)`: deletes the element with key k

`contains(k)`: true if the dictionary contains an element with key k

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

`size()`: returns the size of the dictionary

*also called "Ordered Symbol Tables"

EXPLOITING STRUCTURE

The more "ordered" your structure:

- the more prior information you can exploit to speed up certain operations.

But: you will likely have to pay to maintain this order.

- some operations (e.g., that change the data) will become more expensive

EXPLOITING STRUCTURE

Unsorted array

30	70	12	50	6

Insert: O(1)
Max: O(n)
Search: O(n)

Sorted array

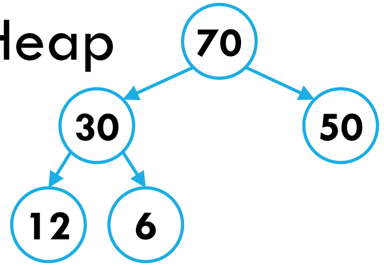
6	12	30	50	70

Insert: O(n)
Max: O(1)
Search: O(logn)

Unordered **Ordered**

EXPLOITING STRUCTURE

Binary Heap



Unsorted array

30	70	12	50	6

Insert: O(1)
Max: O(n)
Search: O(n)

70	30	50	12	6

Insert: O(logn)
Max: O(1)
Search: O(n)

Sorted array

6	12	30	50	70

Insert: O(n)
Max: O(1)
Search: O(logn)

Unordered **Ordered**

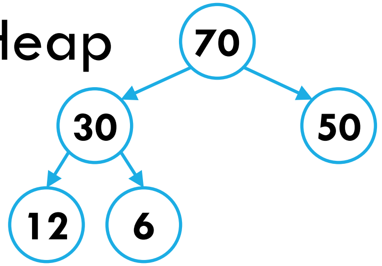
EXPLOITING STRUCTURE

Unsorted array

30	70	12	50	6

Insert: O(1)
Max: O(n)
Search: O(n)

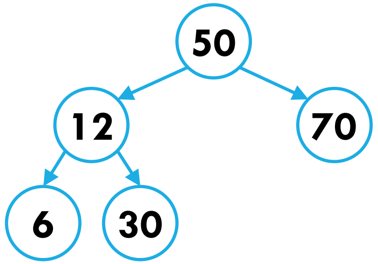
Binary Heap



70	30	50	12	6

Insert: O(logn)
Max: O(1)
Search: O(n)

Balanced Binary **Search** Tree



50	12	70	6	12

Insert: O(logn)
Max: O(logn)
Search: O(logn)

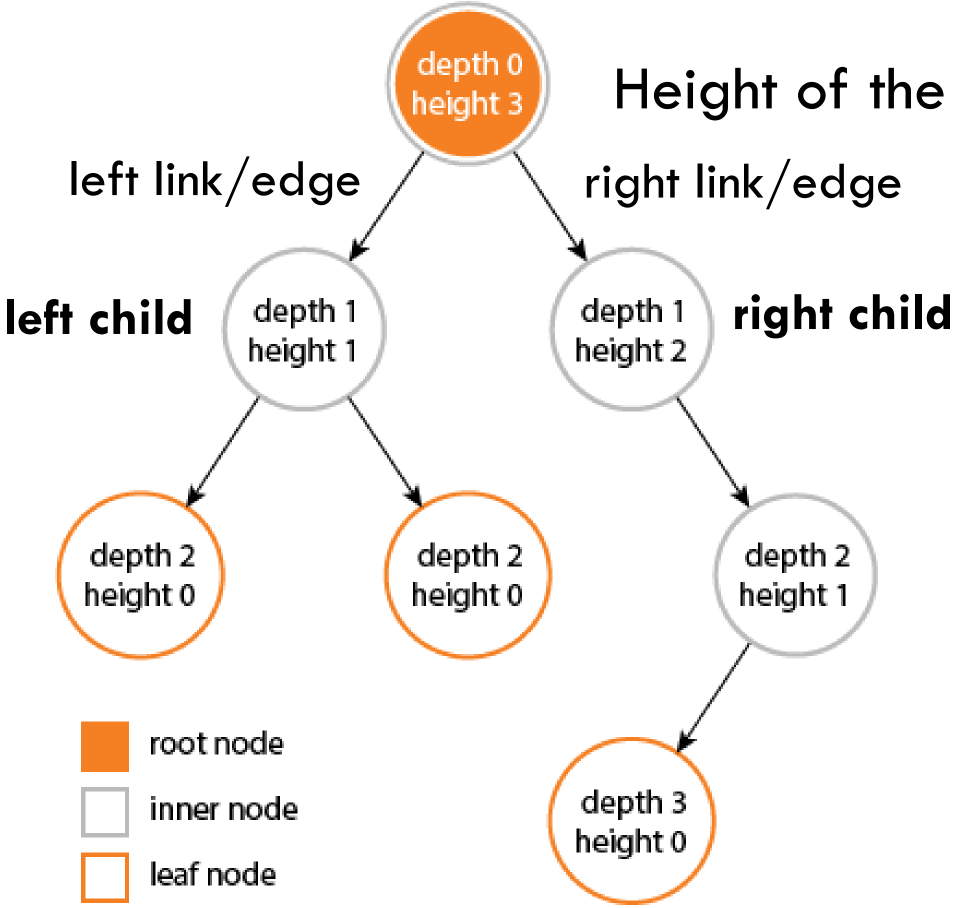
Sorted array

6	12	30	50	70

Insert: O(n)
Max: O(1)
Search: O(logn)

Unordered **Ordered**

A BINARY TREE

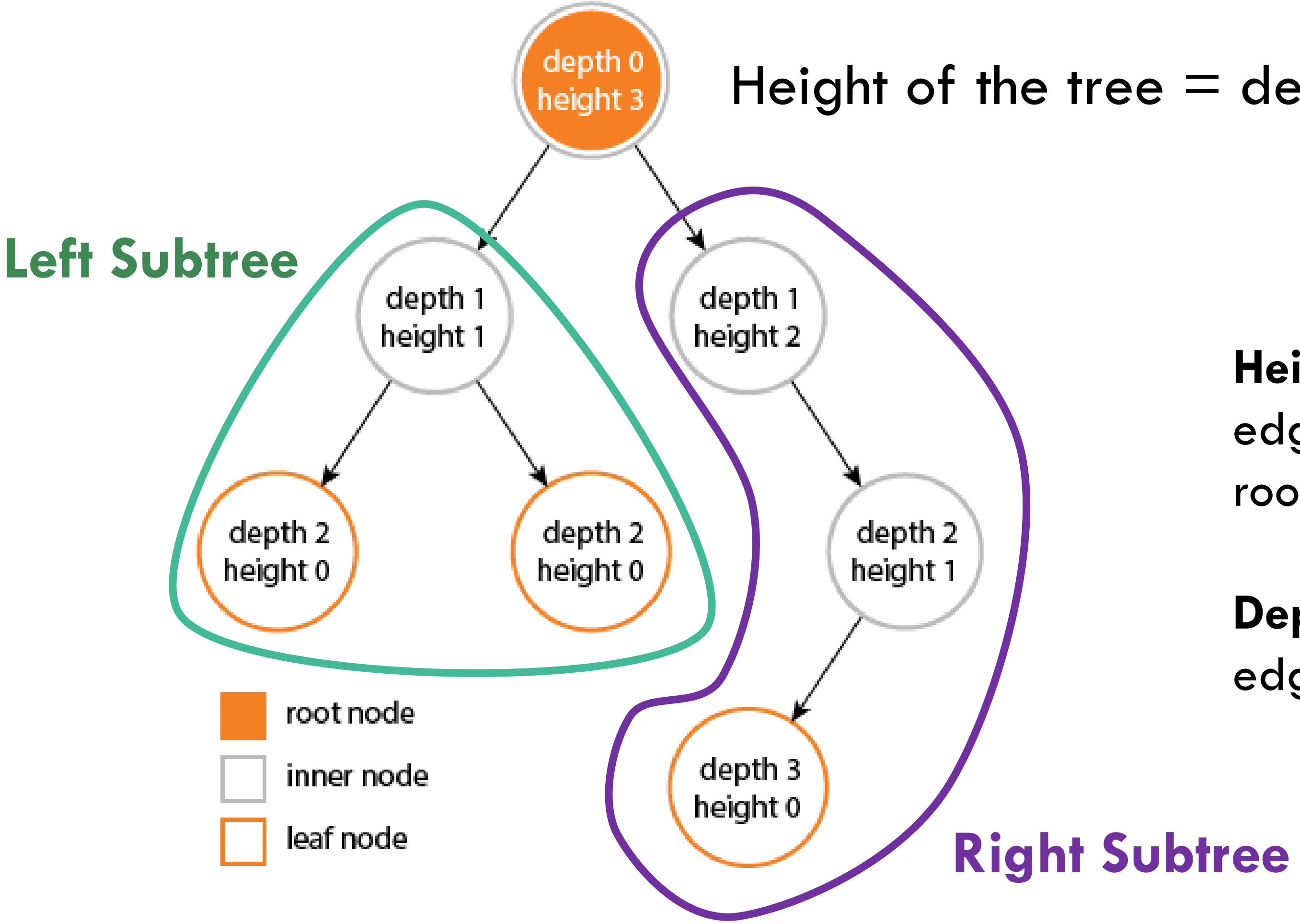


Height of the tree = depth of its deepest node

Height of a node: number of edges on the longest path from node to a leaf.

Depth of a node: number of edges to the root

A BINARY TREE



Height of the tree = depth of its deepest node

Height of a tree: number of edges on the longest path from root to a leaf.

Depth of a node: number of edges to the root

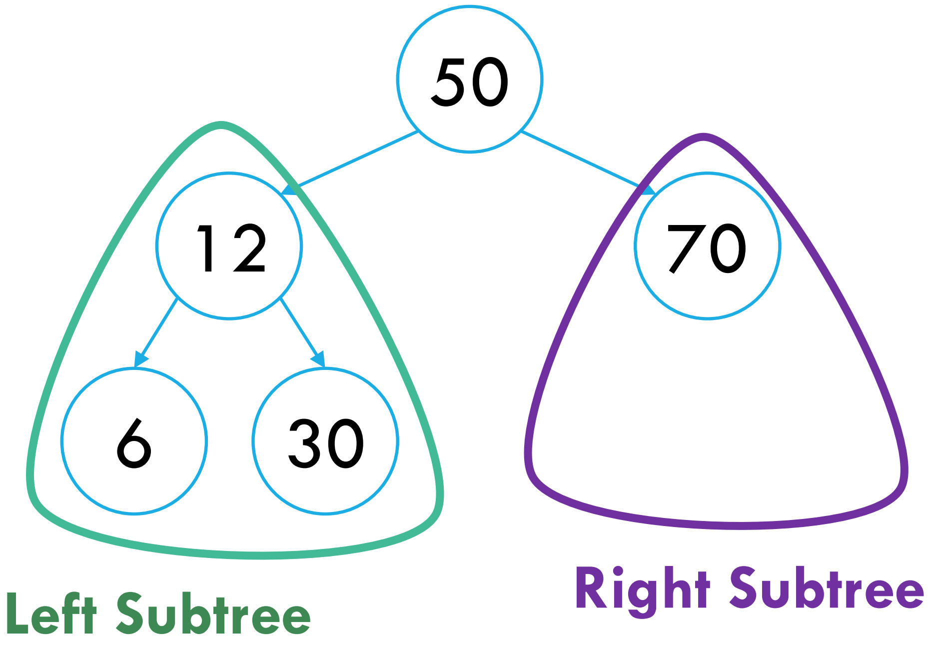
THE BINARY TREE JAVA CLASS

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

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


      private Key m_key;
      private Value m_value;

      // Remainder of binary tree implementation
}
```

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

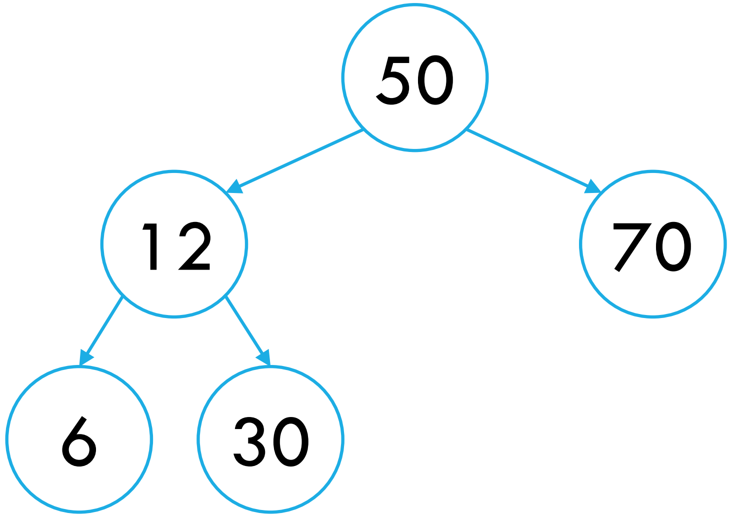


* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY **SEARCH** TREES

Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

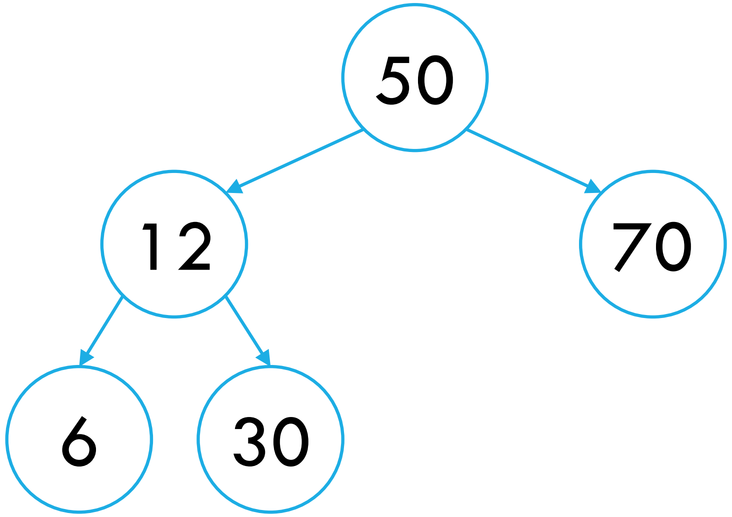


Is the image above a BST?

A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES



Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

Is the image above a BST?

A. Yes

B. No

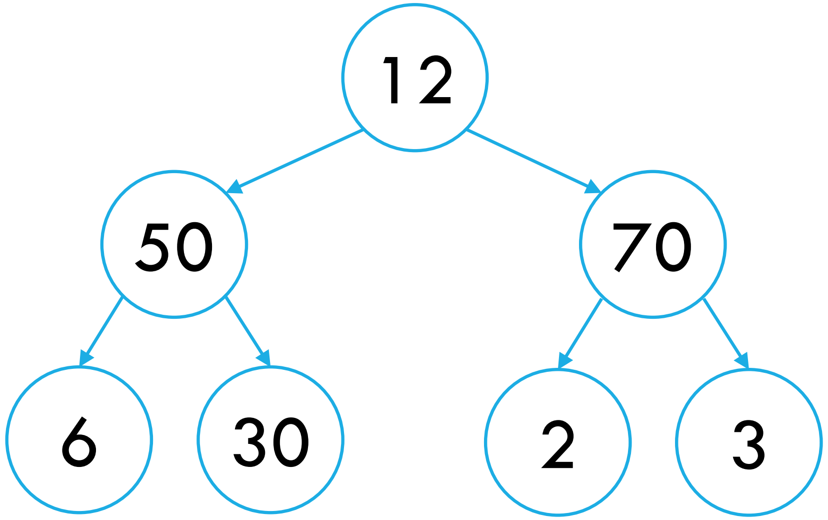
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



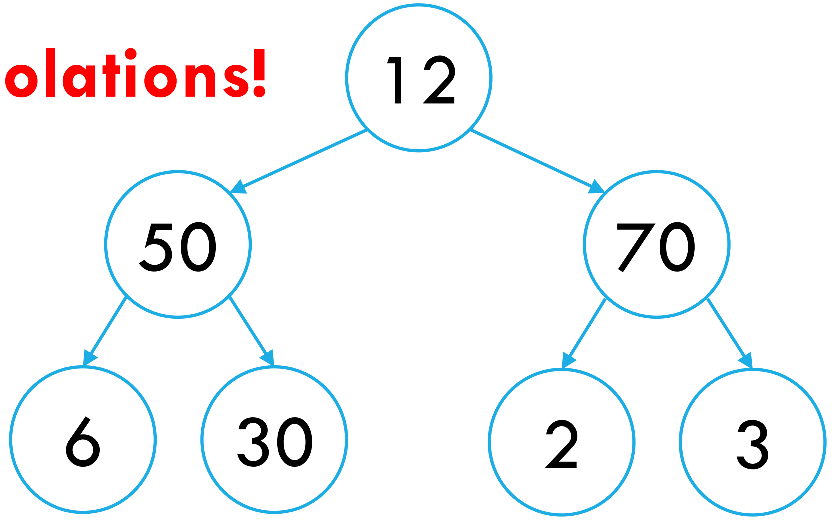
Is the image above a BST?

A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Lots of violations!



Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

Is the image above a BST?

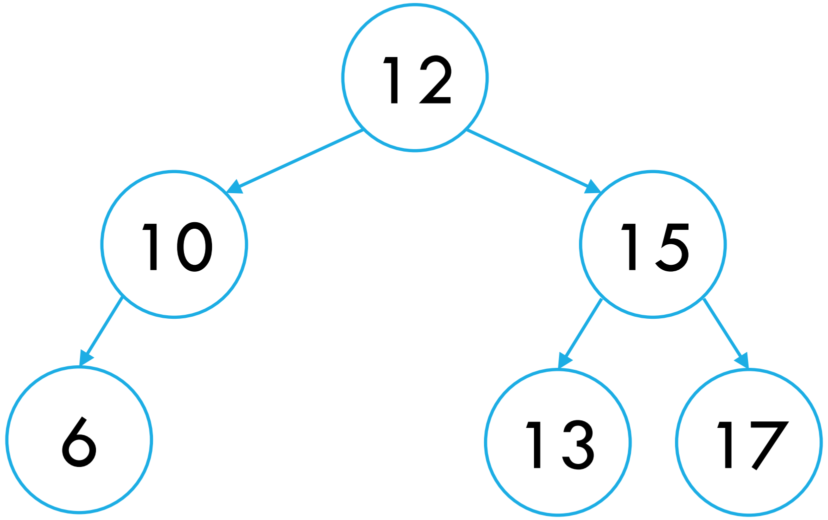
A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

A. Yes

B. No

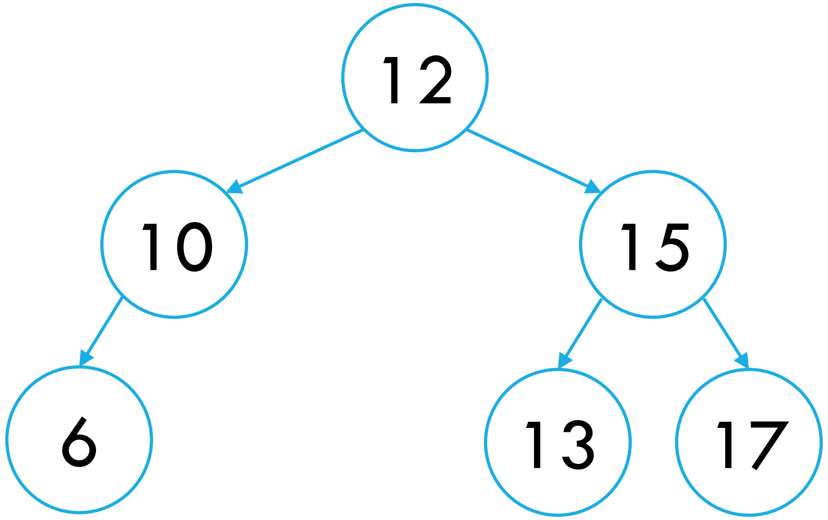
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

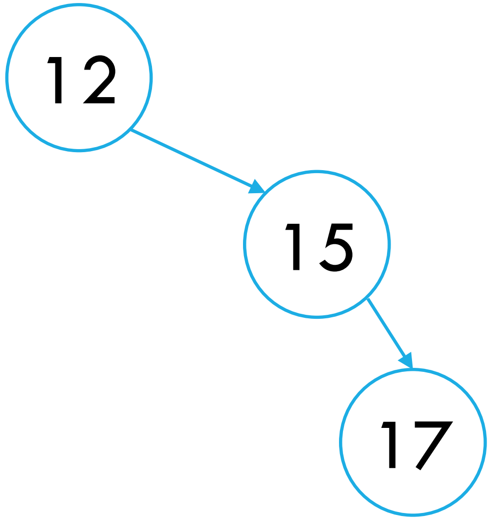
A. **Yes**

B. No

C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES



Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

Is the image above a BST?

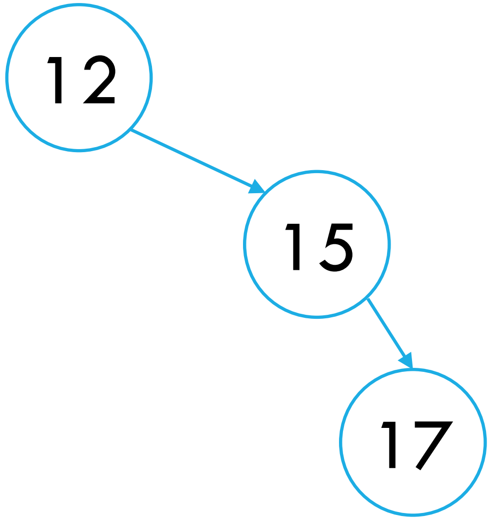
A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

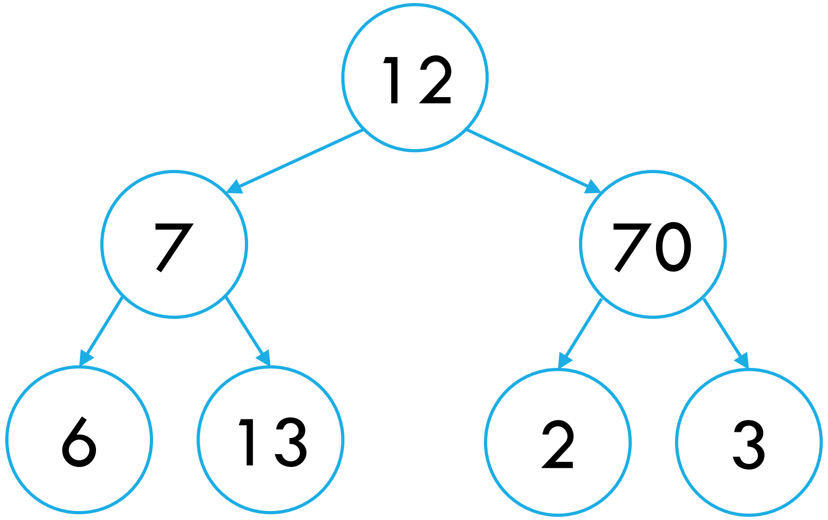
A. **Yes**
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

A. Yes

B. No

C. Maybe…

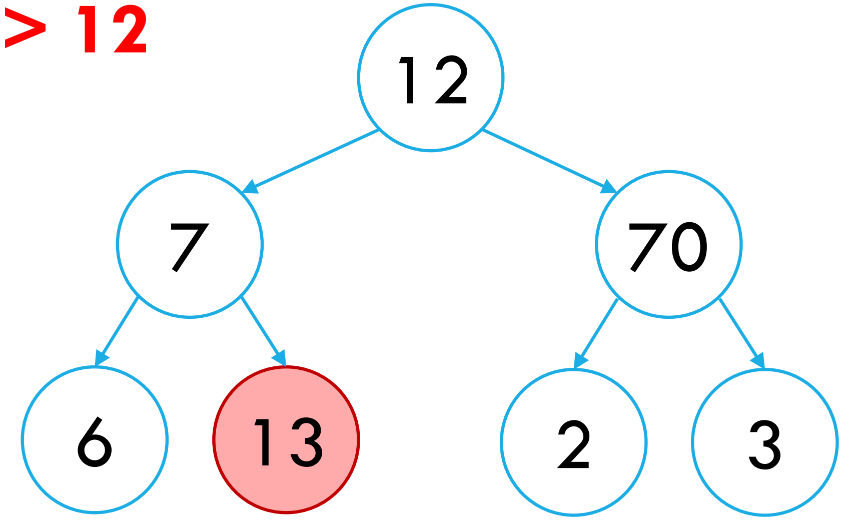
* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

13 > 12



Is the image above a BST?

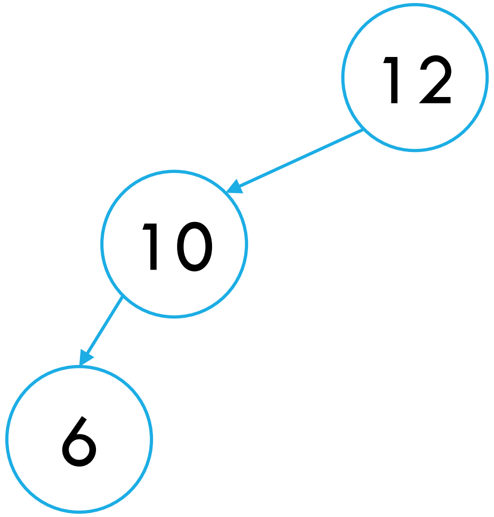
A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

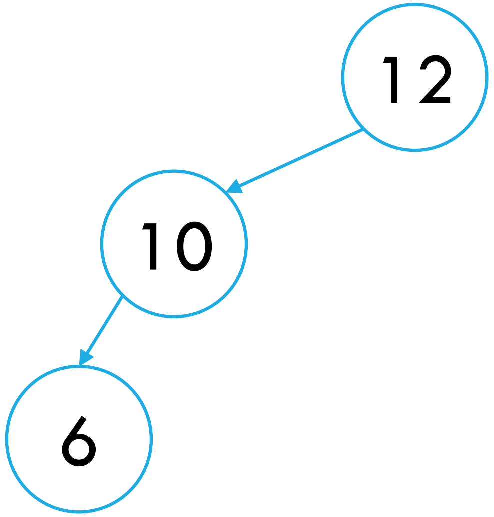
A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node’s left subtree only contains nodes with keys strictly less than the node’s key
- A node’s right subtree only contains nodes with keys strictly larger than the node’s key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



Is the image above a BST?

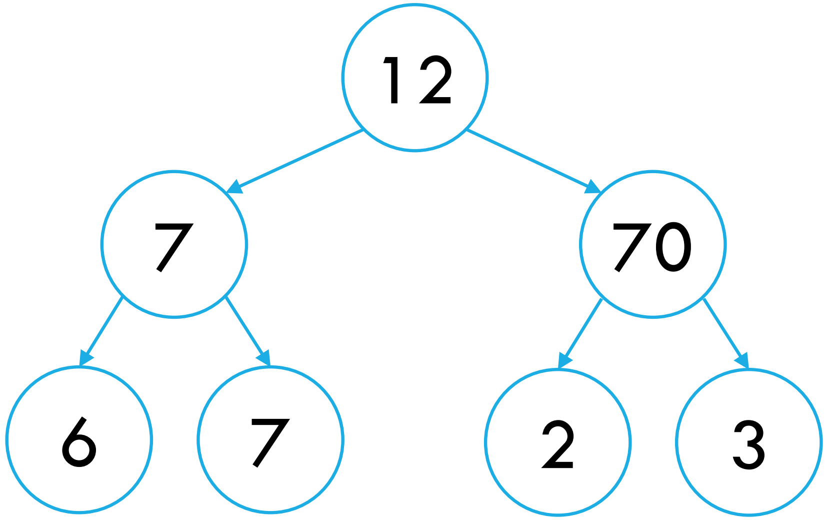
A. **Yes**
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.



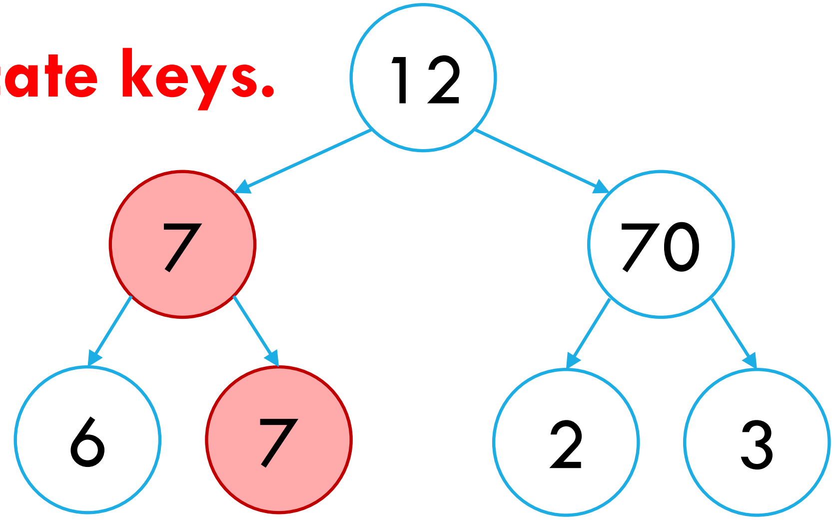
Is the image above a BST?

A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES

no duplicate keys.



Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys strictly less than the node's key
- A node's right subtree only contains nodes with keys strictly larger than the node's key
- The left and right subtrees are binary trees.
- All keys belong to a total order*.

Is the image above a BST?

A. Yes
B. No
C. Maybe…

* $\forall u, v \in K$ when $u \neq v$ either $u < v$ or $v > u$ (no two different keys can be considered equal)

BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

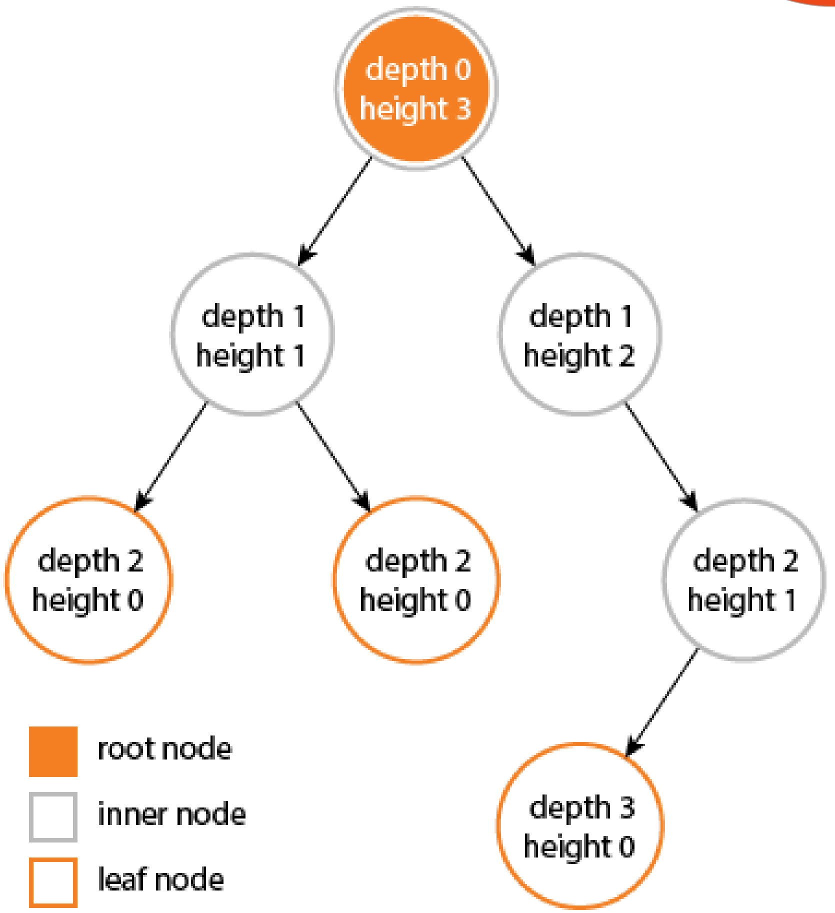
HEIGHT OF A BST

Number of edges on longest path from root to leaf

or Depth of the tree's deepest node

How do we compute it?

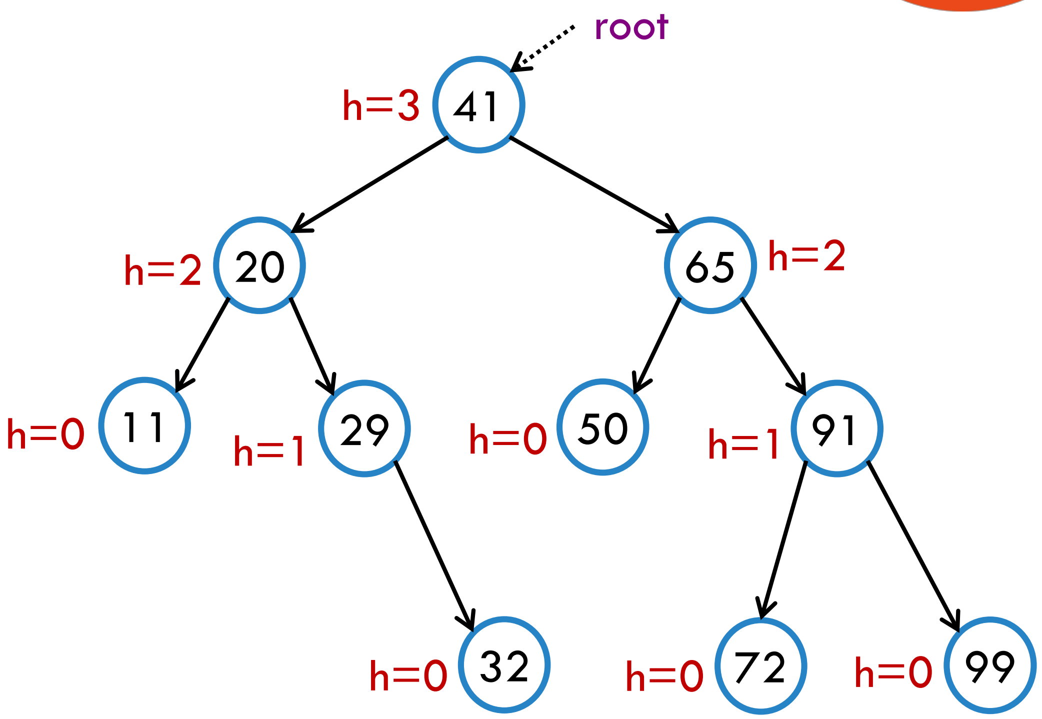
$$h(o) = 0 \text{ if } o \text{ is a leaf}$$
$$h(o) = \max\big(h(o.\text{left}), h(o.\text{right})\big) + 1$$



HEIGHT OF A BST: PSEUDOCODE

Call root.height()

```
function height()
    leftHeight = -1
    rightHeight = -1
    if m_leftTree is not null
        leftHeight = m_leftTree.height()
    if m_rightTree is not null
        rightHeight = m_rightTree.height()
    return max(leftHeight, rightHeight) + 1
```



BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

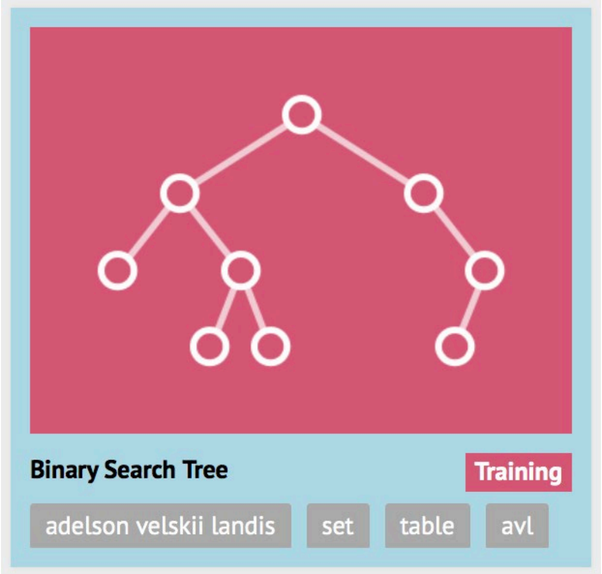
`successor()`: returns next key > k

`predecessor()`: returns next key < k

`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

covered in Visualgo!



BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

`insert(k,v)`: inserts an object v with key k

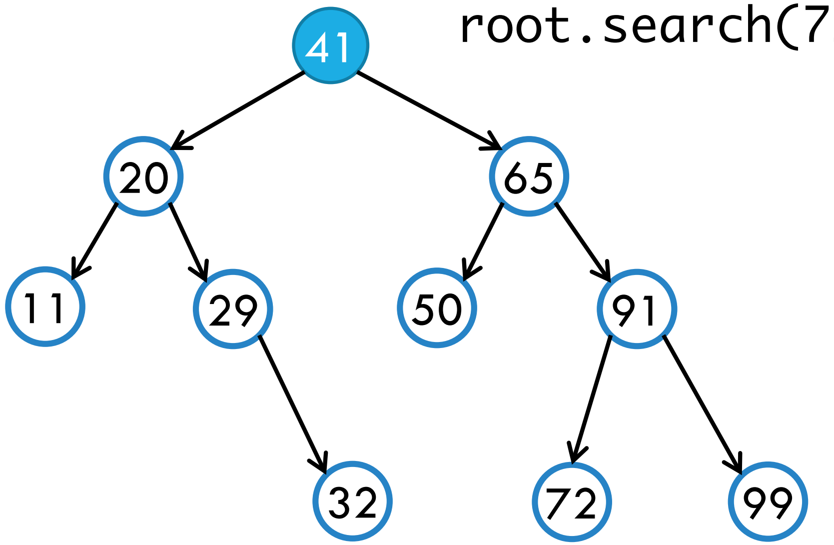
`delete(k)` : deletes the node with key k

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



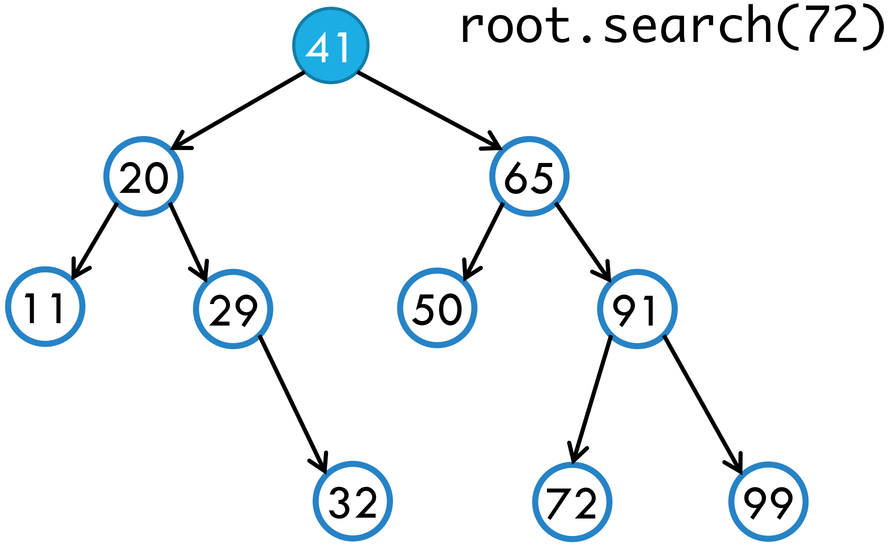

```
function search(Key k)
  if k < m_key


  else if k > m_key


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



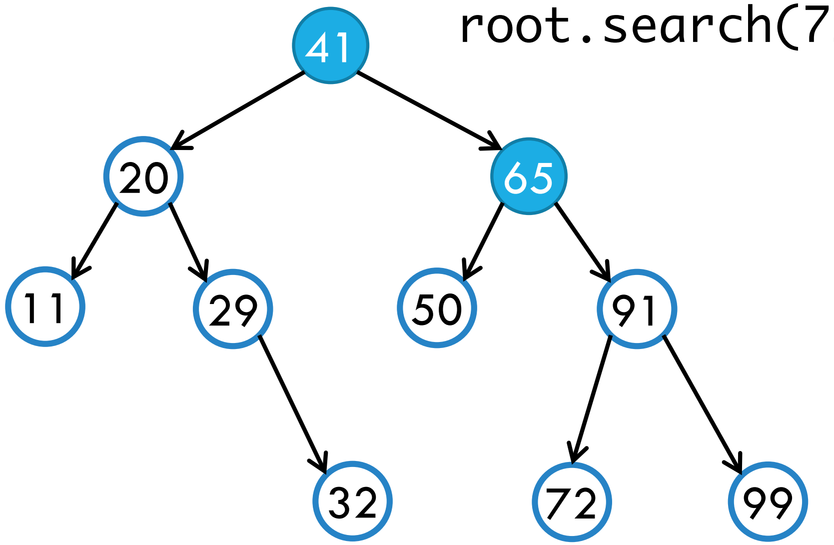

```
function search(Key k)
  if k < m_key


  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



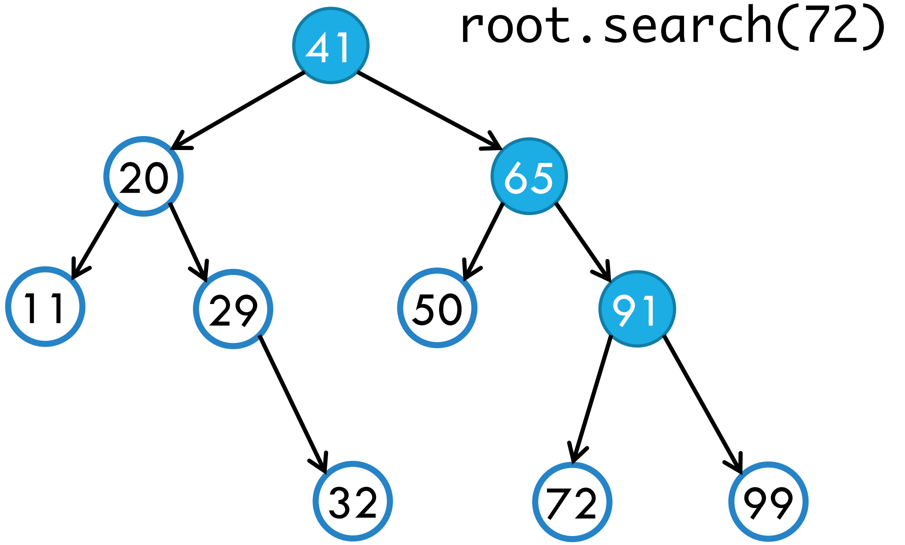

```
function search(Key k)
  if k < m_key



  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



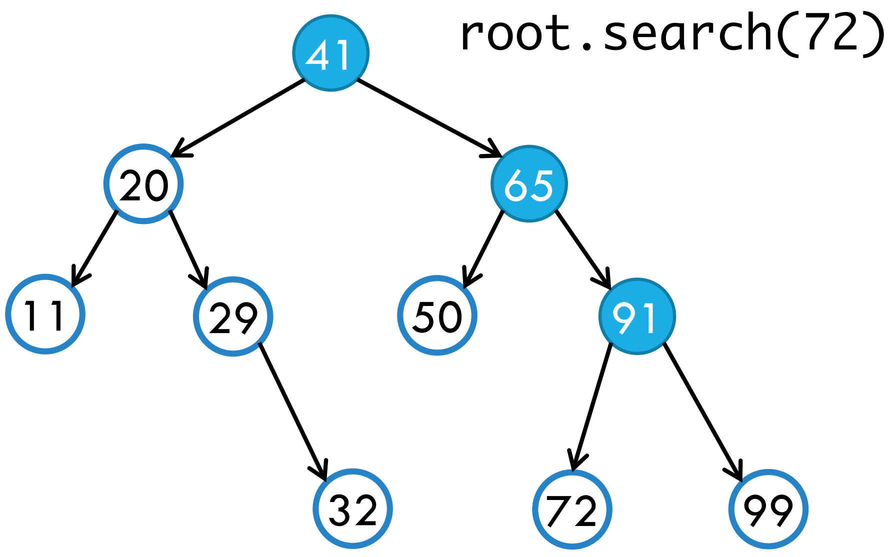

```
function search(Key k)
  if k < m_key


  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`




```
function search(Key k)
  if k < m_key

      return m_leftTree.search(k)

  else if k > m_key

      return m_rightTree.search(k)


  return this
```

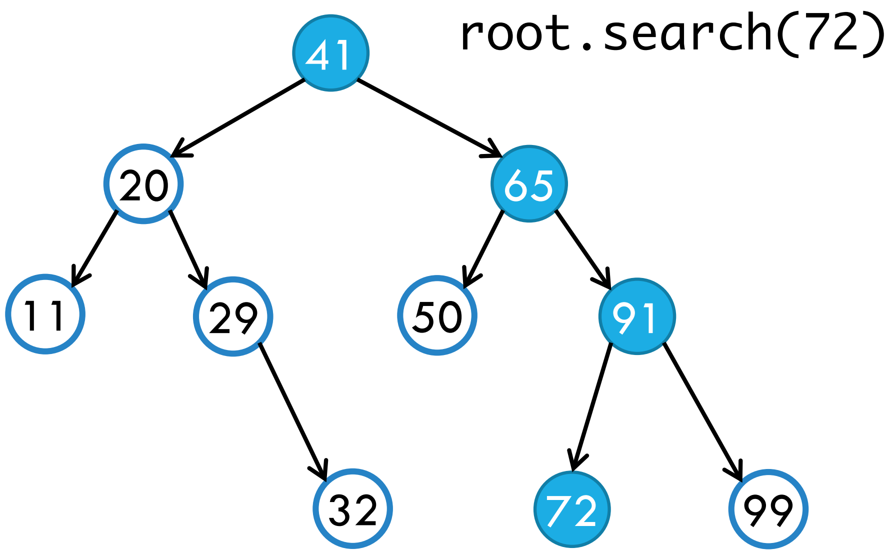
BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`

Is the code correct?



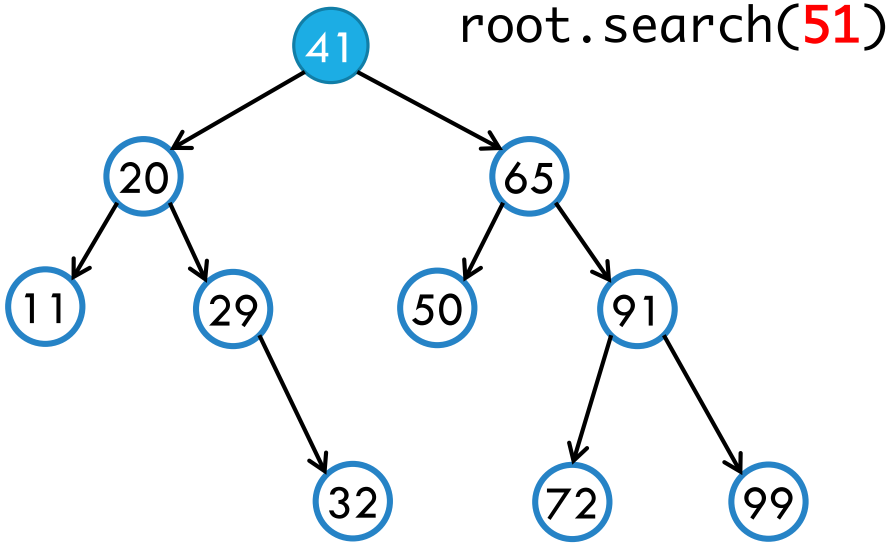

```
function search(Key k)
  if k < m_key

      return m_leftTree.search(k)

  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



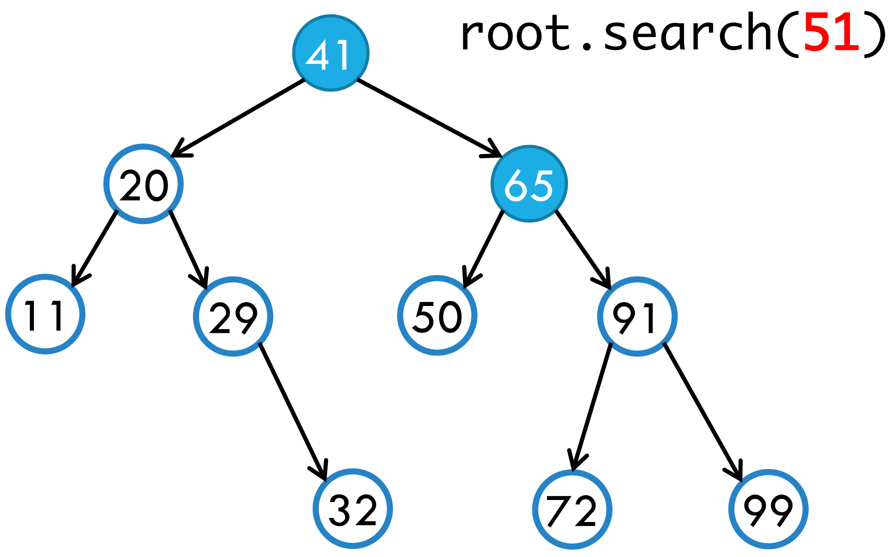

```
function search(Key k)
  if k < m_key

      return m_leftTree.search(k)

  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`



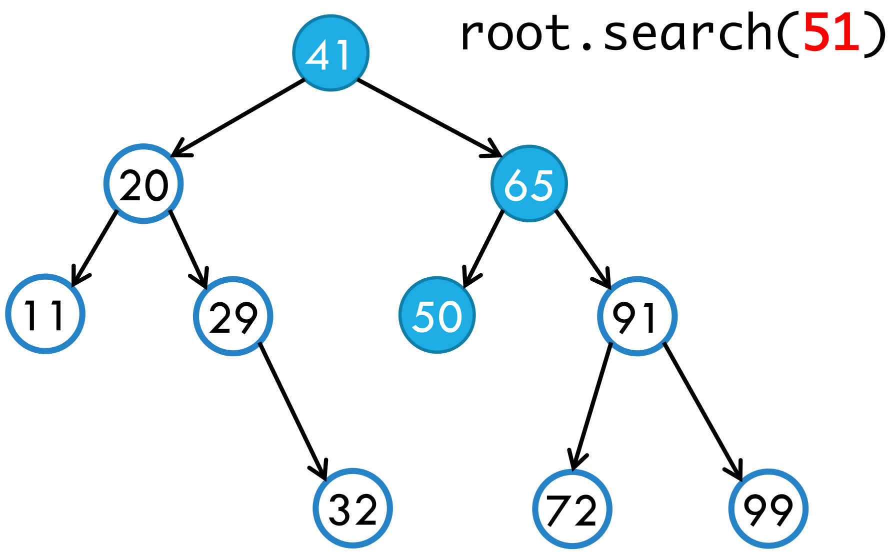

```
function search(Key k)
  if k < m_key

      return m_leftTree.search(k)

  else if k > m_key

      return m_rightTree.search(k)


  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`




```
function search(Key k)
  if k < m_key

      return m_leftTree.search(k)

  else if k > m_key

      return m_rightTree.search(k)


  return this
```

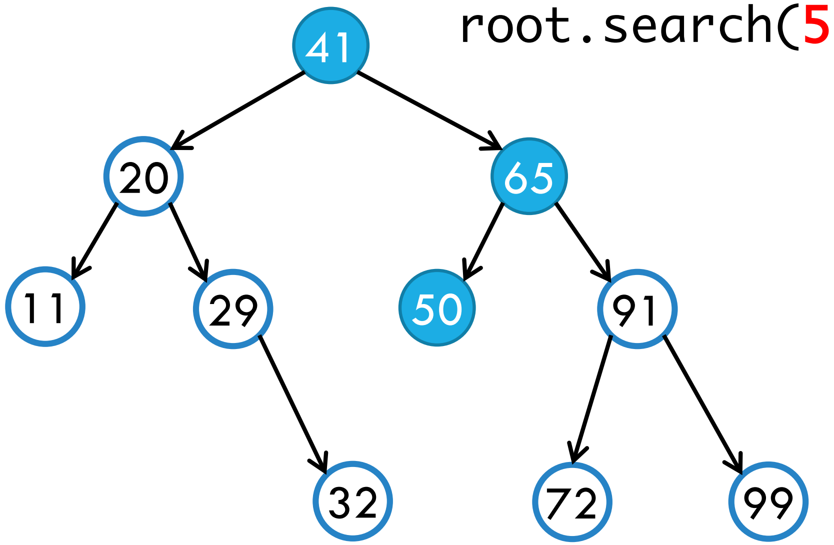
BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`

root.search(51)



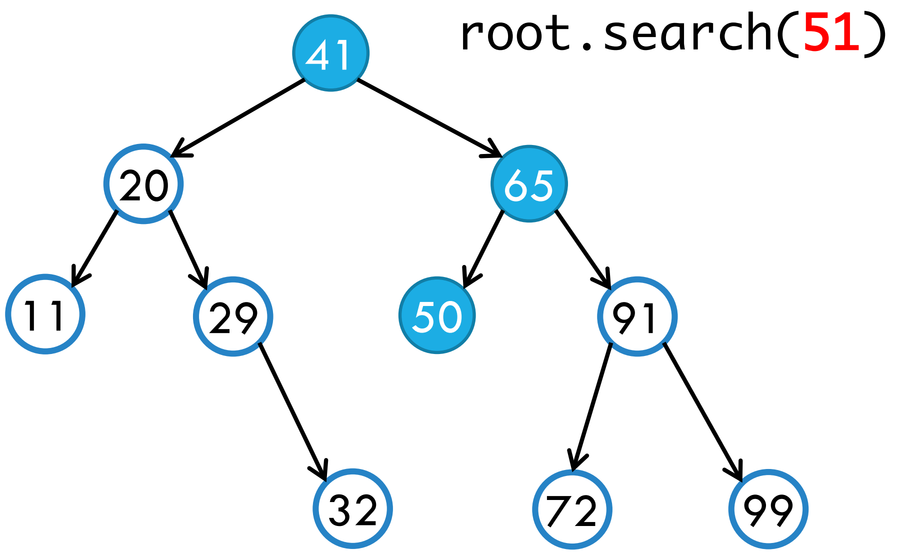

```
function search(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.search(k)
    else return null
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.search(k)
    else return null

  return this
```

BST SEARCHING

Idea: We **binary search** inside a BST.

Given key k that we want to find and current node curr

Call `curr.search(k)`




```
function search(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.search(k)
    else return null
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.search(k)
    else return null

  return this
```

BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

`insert(k,v)`: inserts an object v with key k

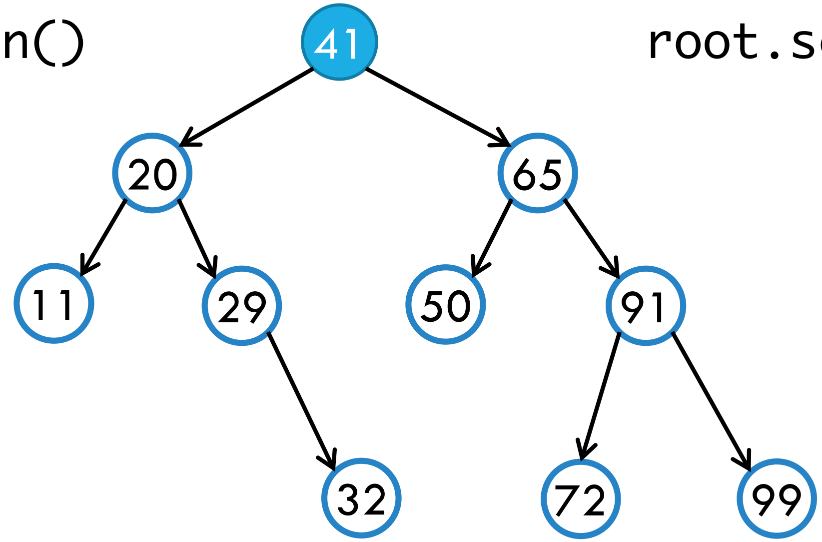
`delete(k)` : deletes the node with key k

SEARCHING FOR THE MINIMUM & MAXIMUM KEYS

Question: Where is the minimum key located?

Question: Where is the maximum key located?

`root.searchMin()`

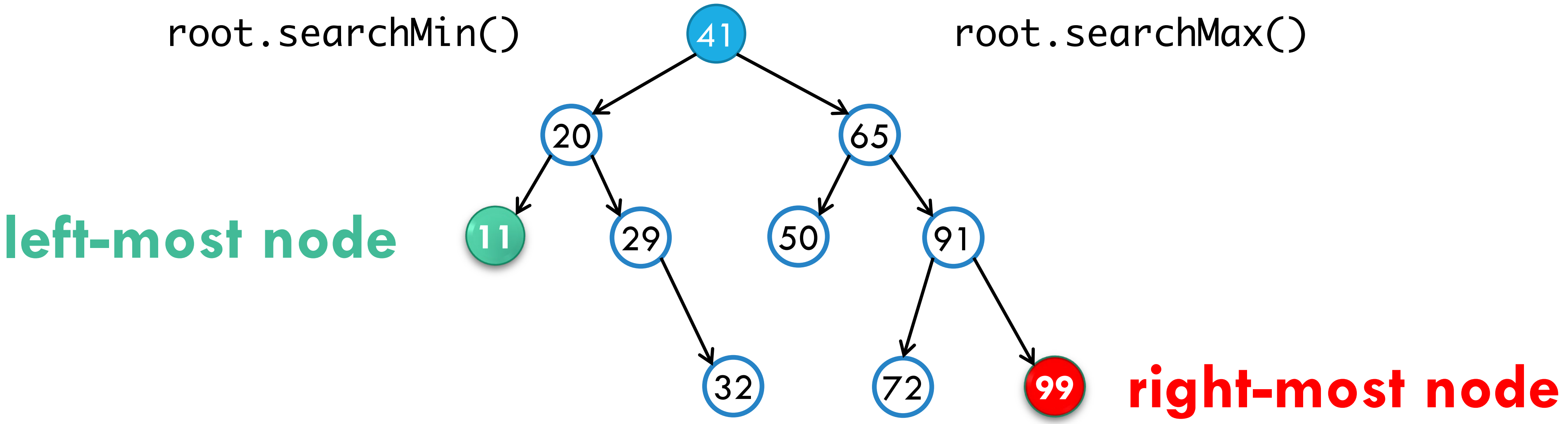


`root.searchMax()`

SEARCHING FOR THE MINIMUM & MAXIMUM KEYS

Question: Where is the minimum key located?

Question: Where is the maximum key located?



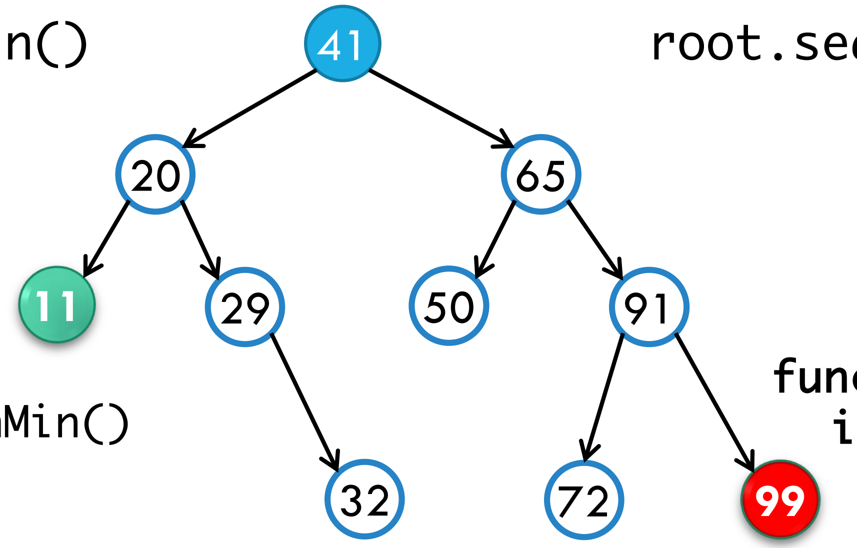
SEARCHING FOR THE MINIMUM & MAXIMUM KEYS

Question: Where is the minimum key located?

Question: Where is the maximum key located?

`root.searchMin()`

`root.searchMax()`




```
function searchMin()
  if m_leftTree is not null
    return m_leftTree.searchMin()

  return this
```

```
function searchMax()
  if m_rightTree is not null
    return m_rightTree.searchMax()

  return this
```

Poll Everywhere

https://bit.ly/2LvG9bq

SEARCHING: WHICH COSTS THE MOST?

In a standard BST, which of the following has the highest worst-case time complexity?

A. searchMin()
B. searchMax()
C. search(k)
D. Both A & B
E. They are all the same.

Poll Everywhere
https://bit.ly/2LvG9bq

SEARCHING: WHICH COSTS THE MOST?

In a standard BST, which of the following has the highest worst-case time complexity?

A. searchMin()
B. searchMax()
C. search(k)
D. Both A & B
E. They are all the same.

$O(h)$ where h is the height of the BST

Poll Everywhere

https://bit.ly/2LvG9bq

SEARCHING: WHICH COSTS THE MOST?

In a standard BST with *n* nodes, what is the worst case height?

A. $O(n)$

B. $O(\log n)$

C. $O(\lfloor \log n \rfloor)$

D. $O(n^2)$

E. 

Poll Everywhere

https://bit.ly/2LvG9bq

SEARCHING: WHICH COSTS THE MOST?

In a standard BST with *n* nodes, what is the worst case height?

A. $\boldsymbol{O(n)}$

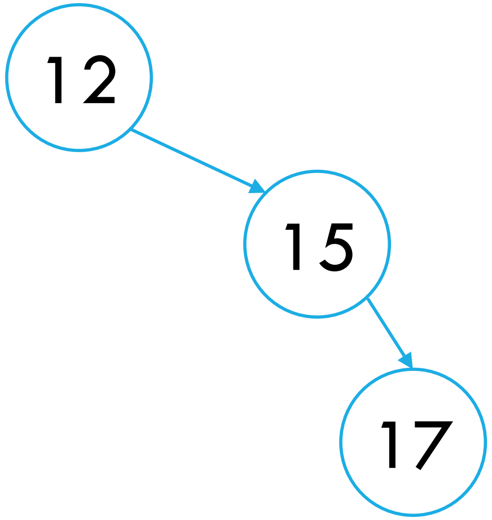
B. $O(\log n)$

C. $O(\lfloor \log n \rfloor)$

D. $O(n^2)$

E. 

BSTs can be lop-sided and form a chain.



BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

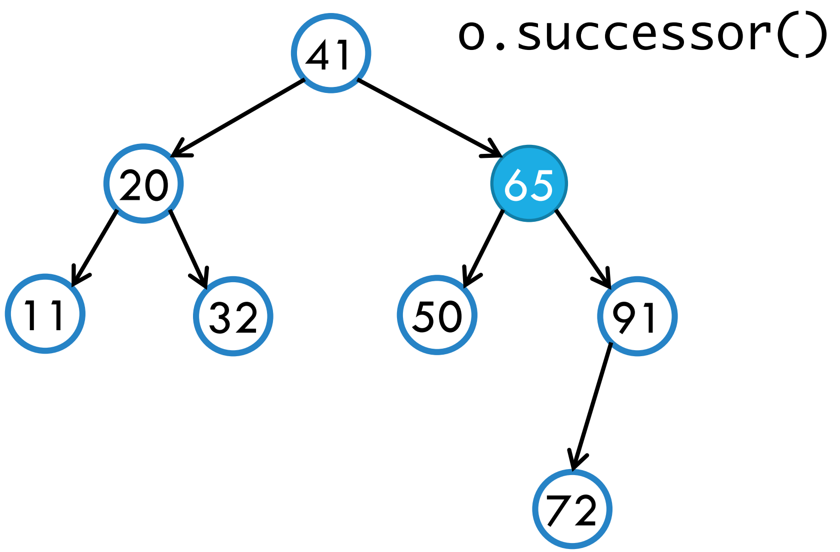
`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

FINDING THE SUCCESSOR NODE

Given a node o with key k:

o's successor = the node with the smallest key that is larger than k



o is the node with key 65. Which of the following is the key of o's successor node?

A. 41
B. 91
C. 72
D. 50
E. Impossible to determine

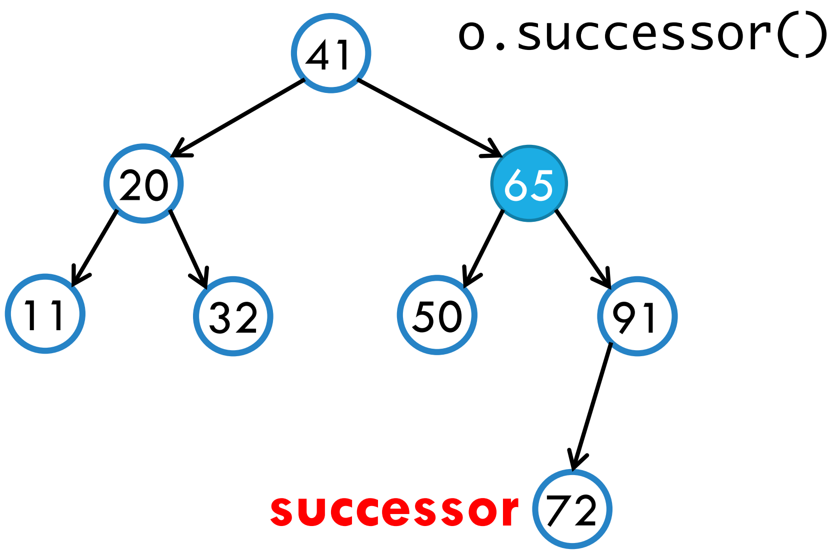
FINDING THE SUCCESSOR NODE

Given a node o with key k:

o's successor = the node with the smallest key that is larger than k

o is the node with key 65. Which of the following is the key of o's successor node?

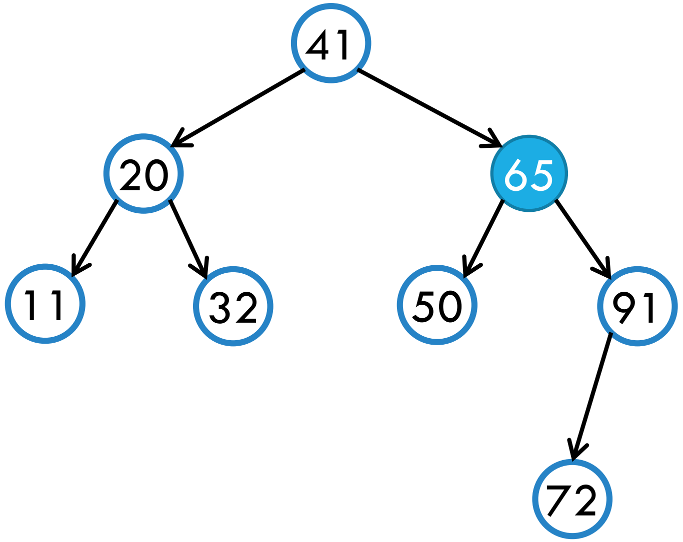
A. 41
B. 91
C. 72
D. 50
E. Impossible to determine



FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

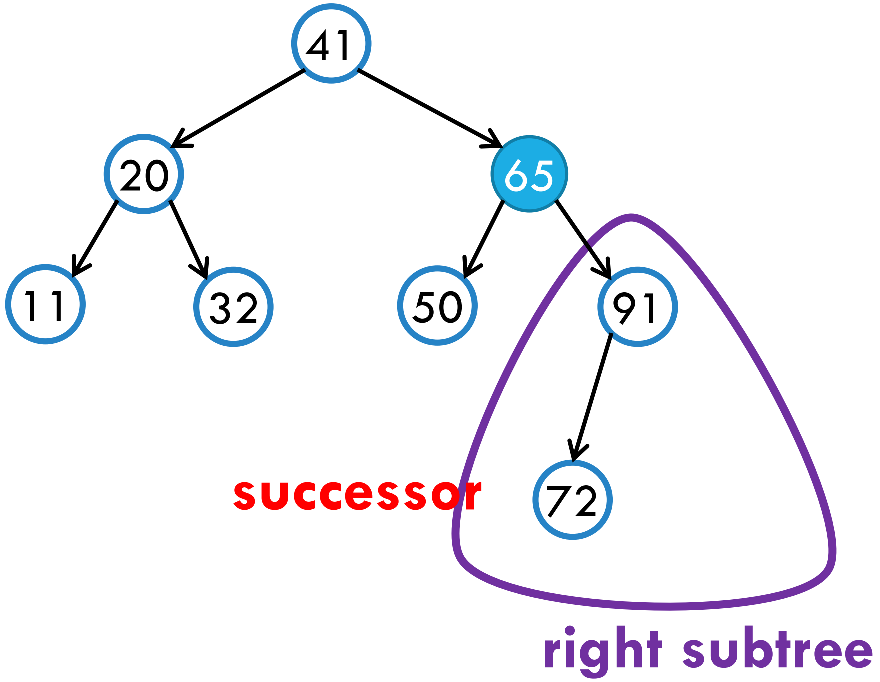
- it has a right subtree
    -
- it doesn't have a right subtree



FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

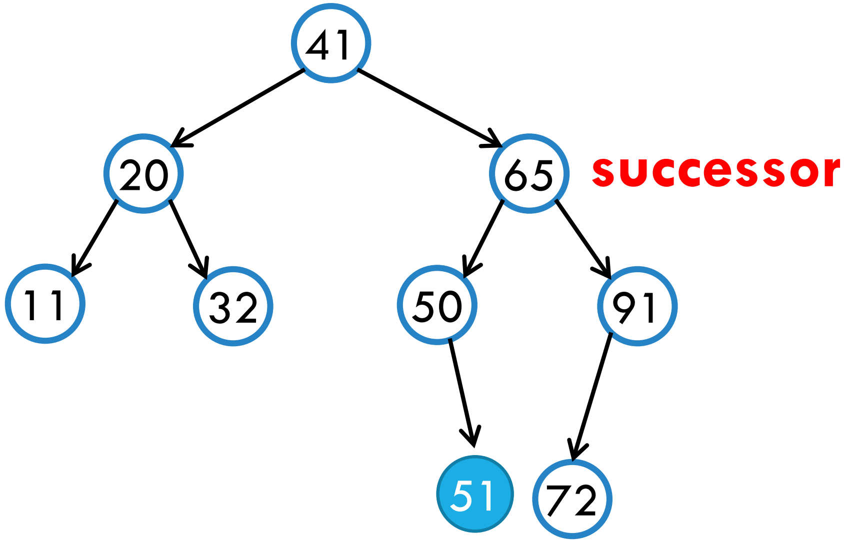
- it has a right subtree
  - **return searchMin(o.m_rightTree)**
- it doesn't have a right subtree



FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

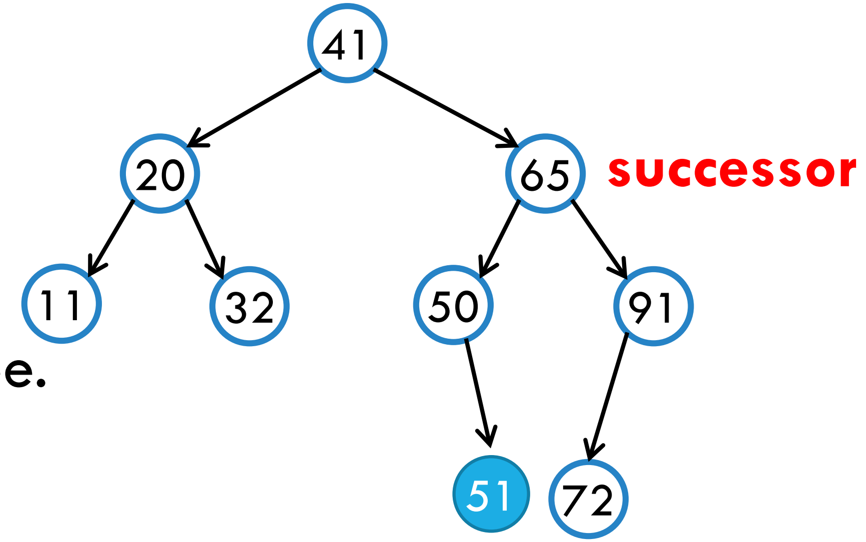
- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Where is the successor?**



FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

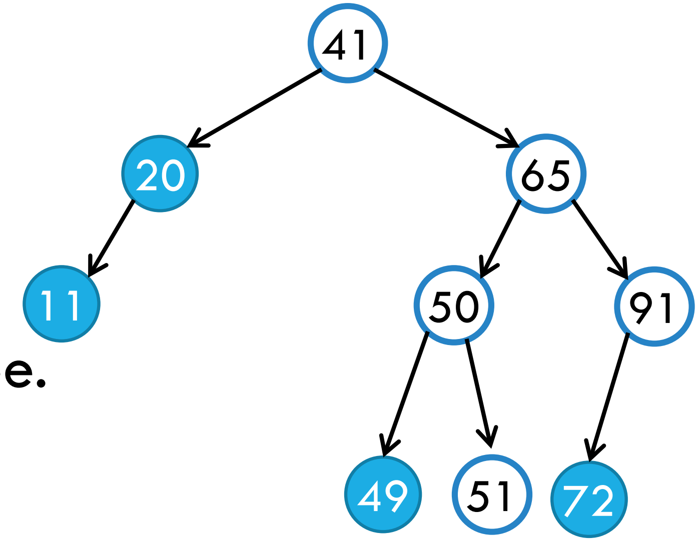
- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: ______________
    - o is the right child: ______________



FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: _____________
    - o is the right child: _____________

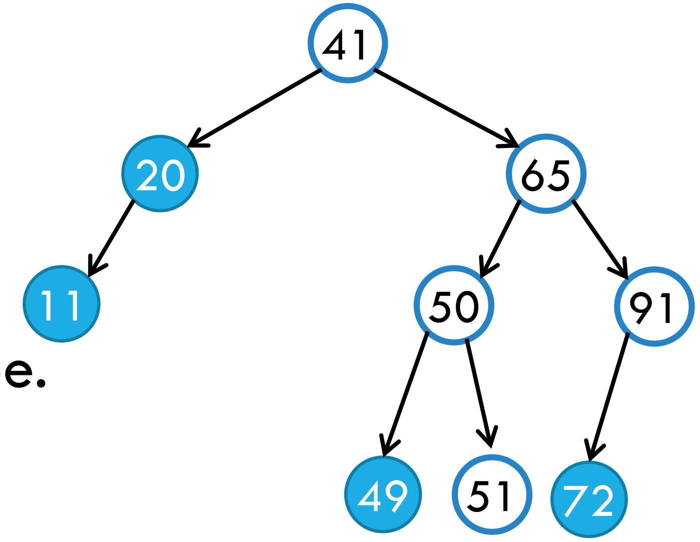


do you see a pattern?

FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: **successor is o's parent**
    - o is the right child: ______________

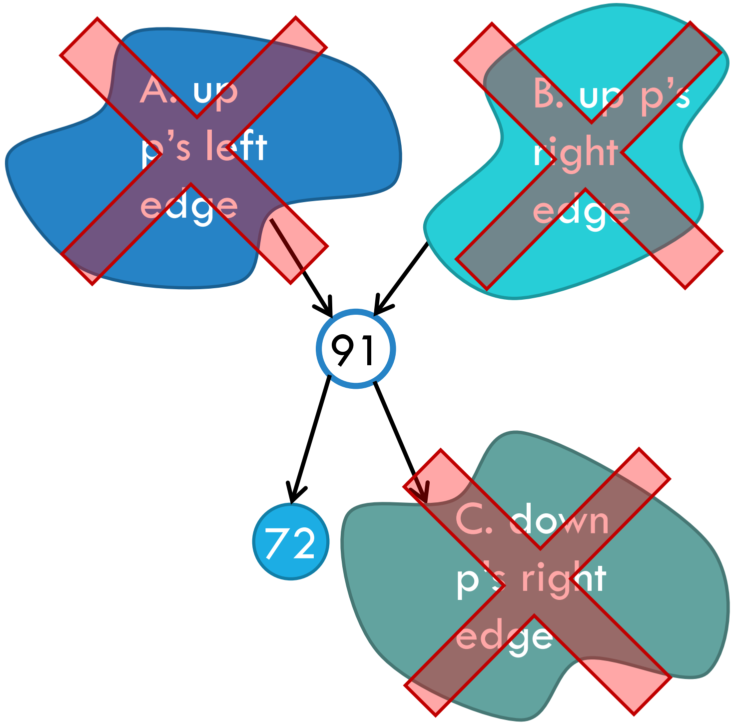


do you see a pattern?

FINDING THE SUCCESSOR NODE

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: **successor is o's parent**
    - o is the right child: ______________



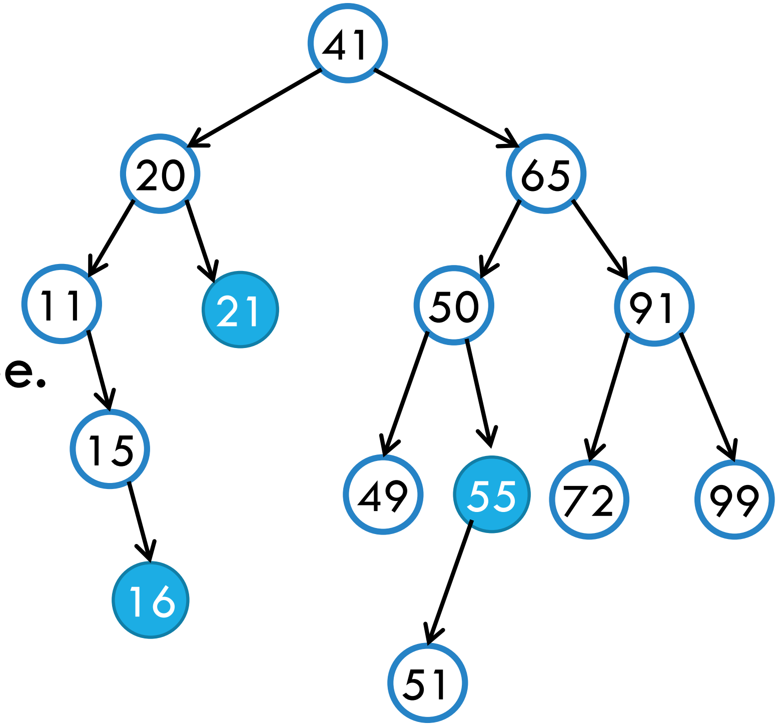
Examine each in turn

FINDING THE SUCCESSOR NODE

do you see a pattern?

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: successor is o's parent
    - o is the right child: _______________

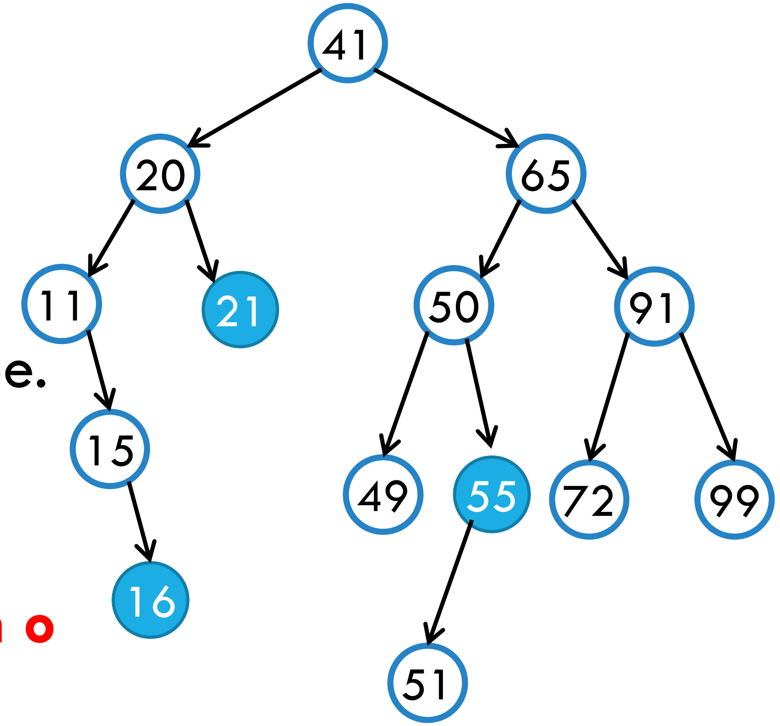


FINDING THE SUCCESSOR NODE

do you see a pattern?

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: successor is o's parent
    - o is the right child: **first ancestor up from o that has key > k. ("first right ancestor")**

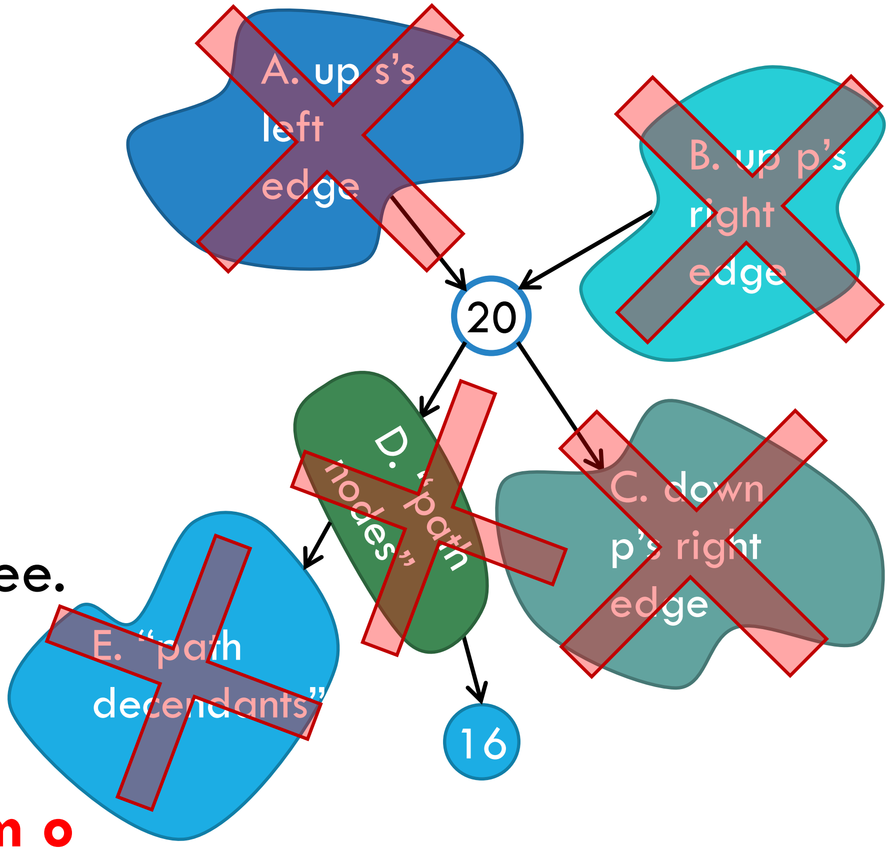


FINDING THE SUCCESSOR NODE

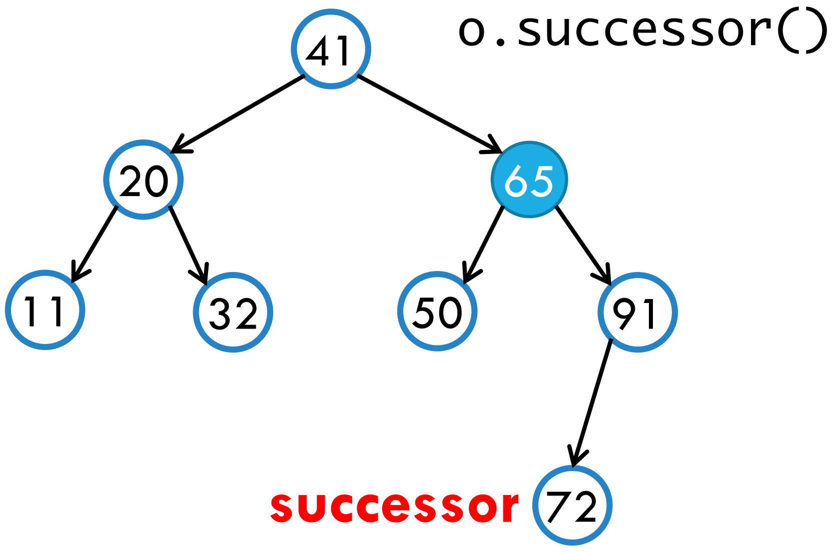
Examine each in turn

Given a node o, there are 2 cases:

- it has a right subtree
  - return searchMin(o.m_rightTree)
- it doesn't have a right subtree
  - **Claim:** successor will be higher up in the tree.
  - 2 cases:
    - o is the left child: successor is o's parent
    - o is the right child: **first ancestor up from o that has key > k. ("first right ancestor")**



SUCCESSOR PSEUDOCODE




```
function successor()
    if m_rightTree is not null then
      return m_rightTree.searchMin()
    else
      p = m_parent, temp = this
      while (p is not null) and (temp == p.rightTree)
        temp = p, p = temp.m_parent
      if p is null then return -1
      else return p
```

if m has no rightsubtree and m is right child

if m has no right subtree and m is left child

very similar for predecessor(); just reason about it slowly.

BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

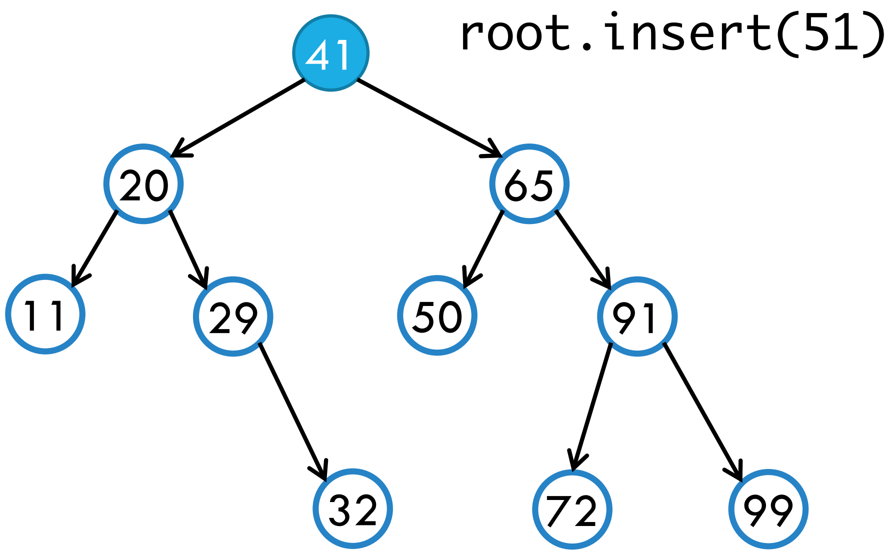
`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

BST INSERTION

Need to insert the key in the “right” place to preserve the BST properties.

Idea: We search until we can’t go any further and add the node there.

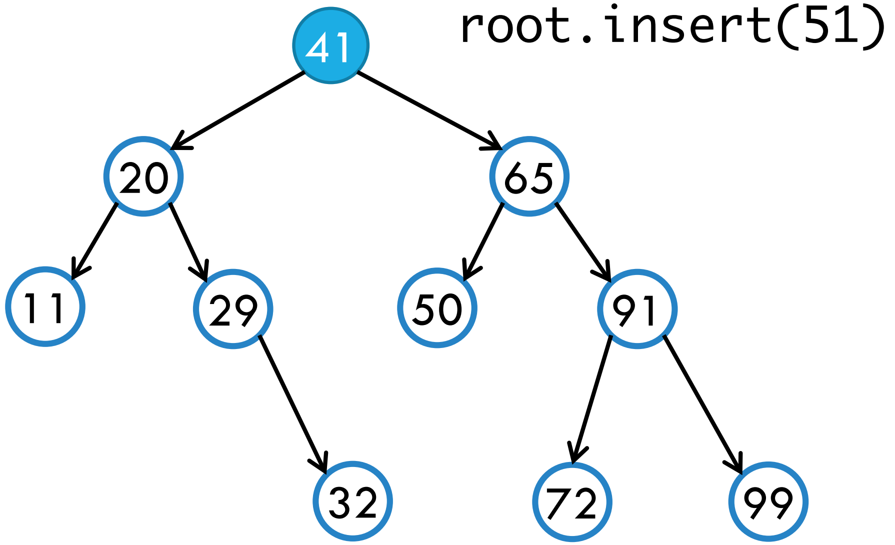



```
function insert(Key k)
  if k < m_key



  else if k > m_key



  else return
```

BST INSERTION

Need to insert the key in the "right" place to preserve the BST properties.

Idea: We search until we can't go any further and add the node there.

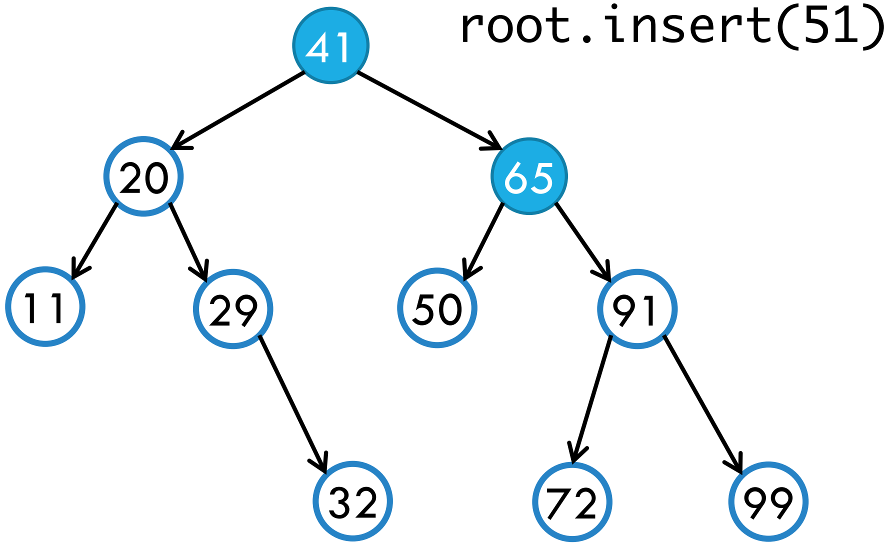



```
function insert(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.insert(k)
    else
      m_leftTree = new BinaryTree(k)
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.insert(k)
    else
      m_rightTree = new BinaryTree(k)
  else return
```

BST INSERTION

Need to insert the key in the “right” place to preserve the BST properties.

Idea: We search until we can’t go any further and add the node there.



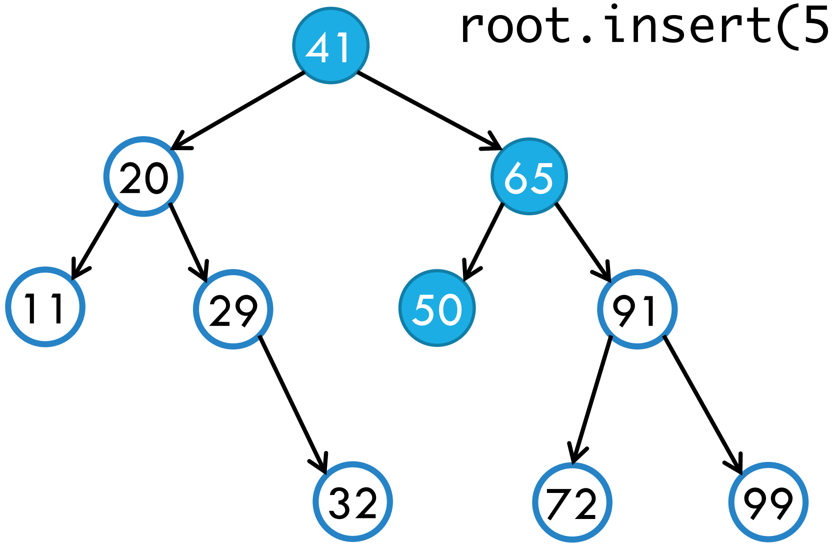

```
function insert(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.insert(k)
    else
      m_leftTree = new BinaryTree(k)
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.insert(k)
    else
      m_rightTree = new BinaryTree(k)
  else return
```

BST INSERTION

Need to insert the key in the “right” place to preserve the BST properties.

Idea: We search until we can’t go any further and add the node there.

root.insert(51)



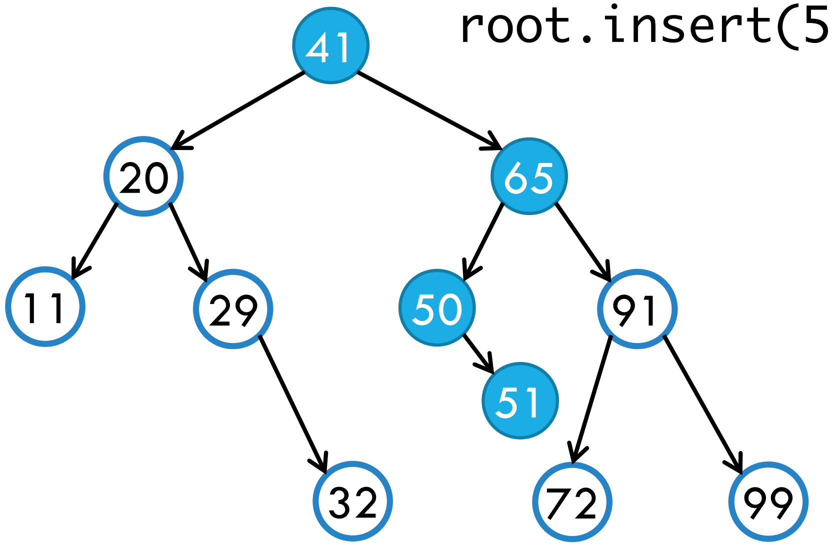

```
function insert(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.insert(k)
    else
      m_leftTree = new BinaryTree(k)
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.insert(k)
    else
      m_rightTree = new BinaryTree(k)
  else return
```

BST INSERTION

Need to insert the key in the "right" place to preserve the BST properties.

Idea: We search until we can't go any further and add the node there.

root.insert(51)



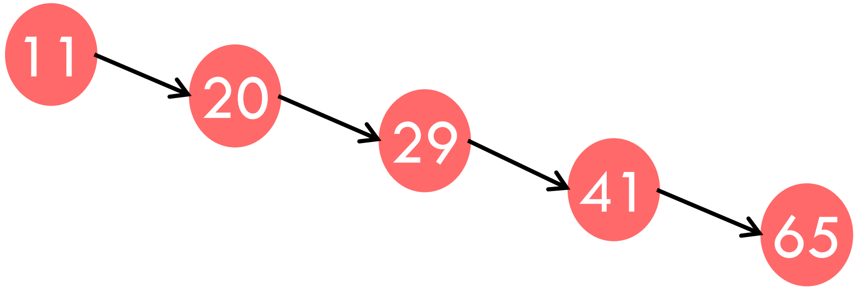

```
function insert(Key k)
  if k < m_key
    if m_leftTree is not null
      return m_leftTree.insert(k)
    else
      m_leftTree = new BinaryTree(k)
  else if k > m_key
    if m_rightTree is not null
      return m_rightTree.insert(k)
    else
      m_rightTree = new BinaryTree(k)
  else return
```

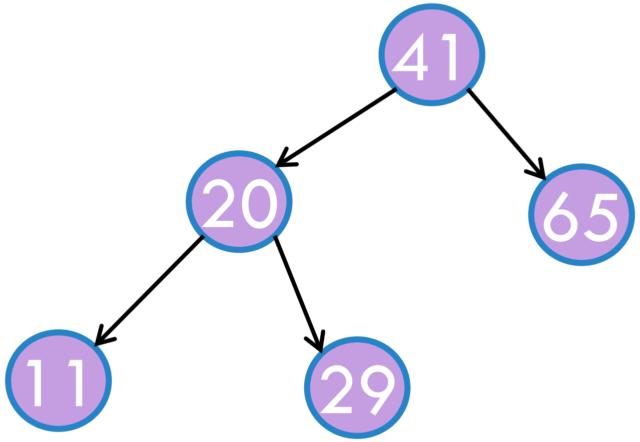
SHAPE OF THE TREE & ORDER OF INSERTION

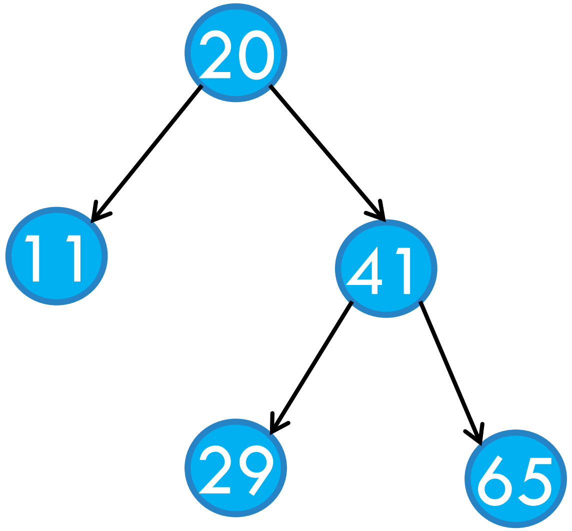
Performance of searching and inserts depends on height.

Height depends on shape

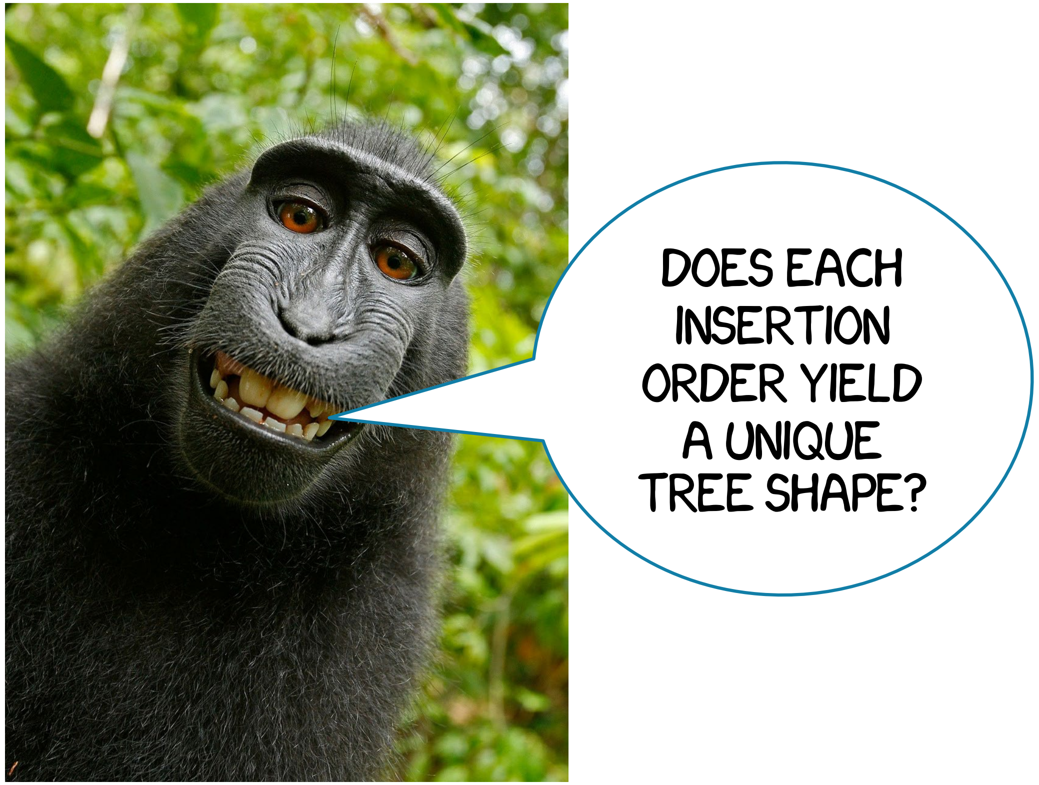
Shape depends on order of insertion (and deletions)



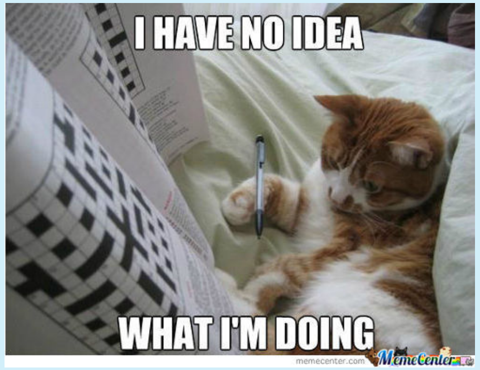




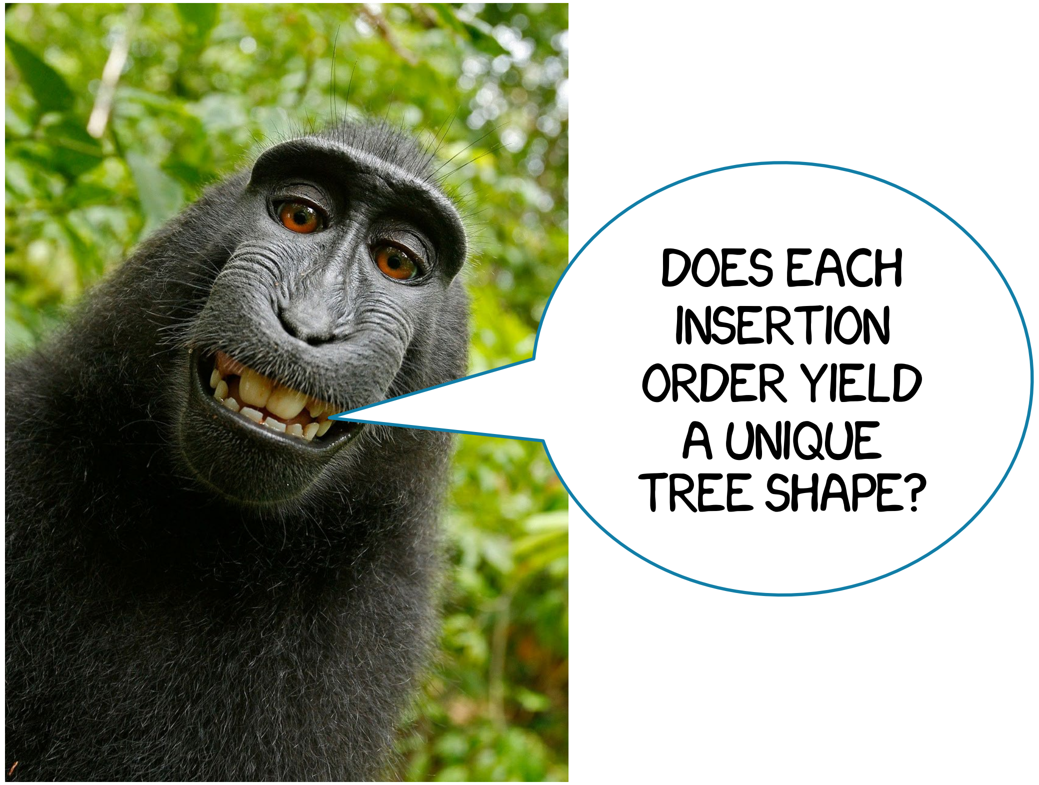
NARUTO PUZZLE: SHAPES AND INSERTIONS



Does each insertion order yield a *unique* shape?

A. Yes!
B. No!
C. Umm.. maybe.
D. 

NARUTO PUZZLE: SHAPES AND INSERTIONS

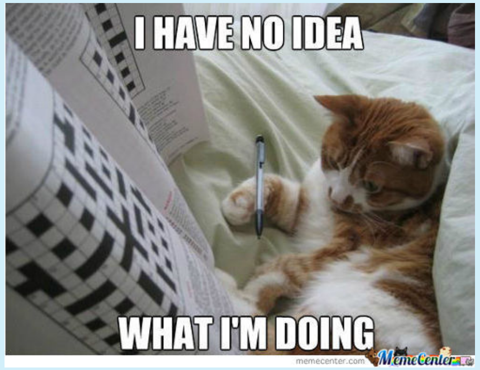


Does each insertion order yield a *unique* shape?

A. Yes!

B. No!

C. Umm.. maybe.

D. 

NARUTO PUZZLE: SHAPES AND INSERTIONS

How many ways to order insertions? $n!$

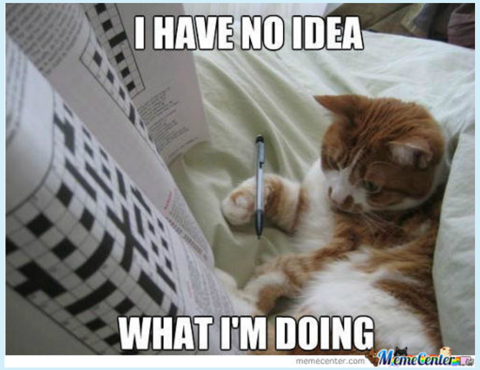
How many shapes of a binary tree?

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

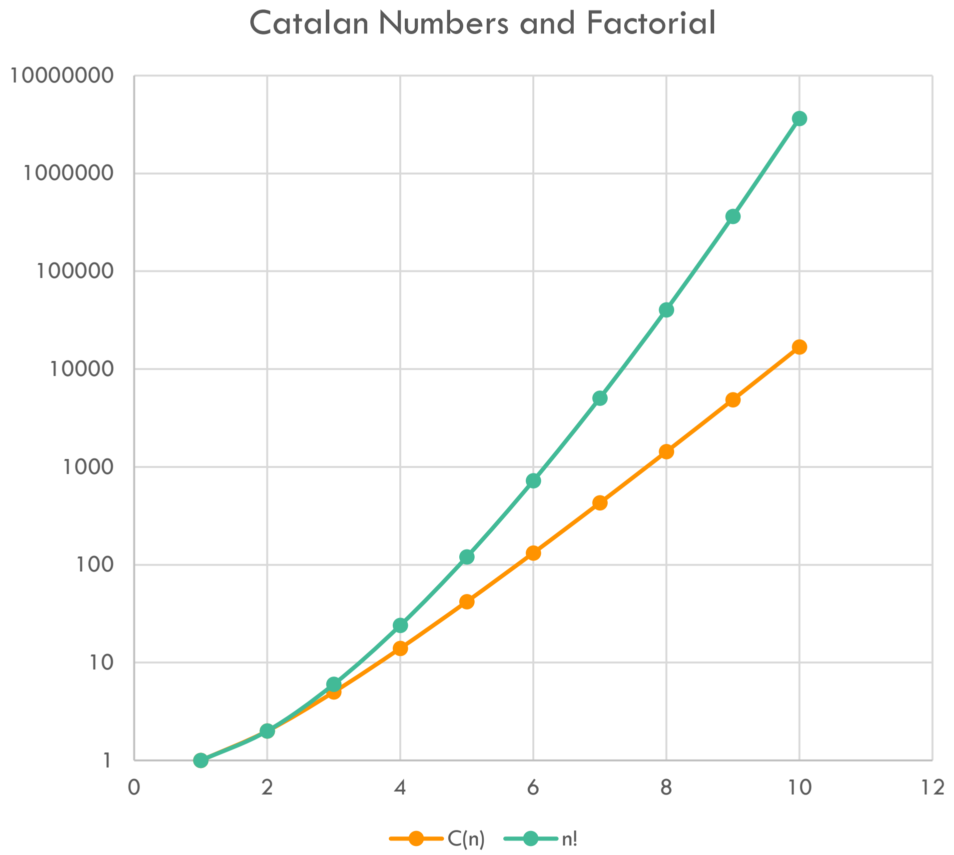
More at :
http://cs.lmu.edu/~ray/notes/binarytrees/

Does each insertion order yield a *unique* shape?

A. Yes!
B. No!
C. Umm.. maybe.
D.



NARUTO PUZZLE: SHAPES AND INSERTIONS

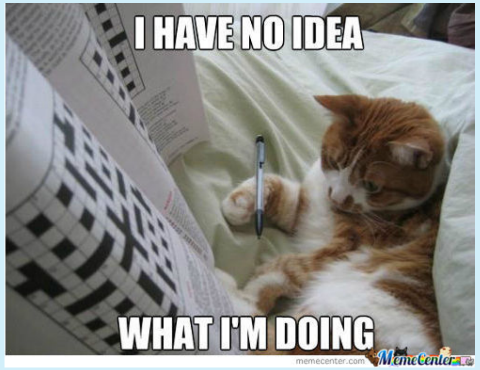


Does each insertion order yield a *unique* shape?

A. Yes!

B. No!

C. Umm.. maybe.

D. 

WHAT IF THE KEYS ARE INSERTED IN RANDOM ORDER?

What is the average-case time complexity of n distinct keys inserted into a BST in *random* order?

A. $O(1)$
B. $O(n)$
C. $O(\log n)$
D. 

WHAT IF THE KEYS ARE INSERTED IN **RANDOM** ORDER?

The proof for this one is long.

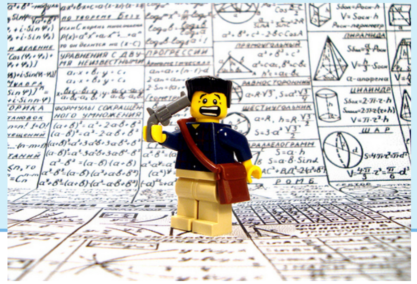
For interested readers: look at [CLRS Section 12.4] or [Sedgewick & Wayne, Section 3.2]

What is the average-case time complexity of n distinct keys inserted into a BST in *random* order?

A. $O(1)$

B. $O(n)$

C. $\boldsymbol{O(\log n)}$

D. 

BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key $> k$

`predecessor()`: returns next key $< k$

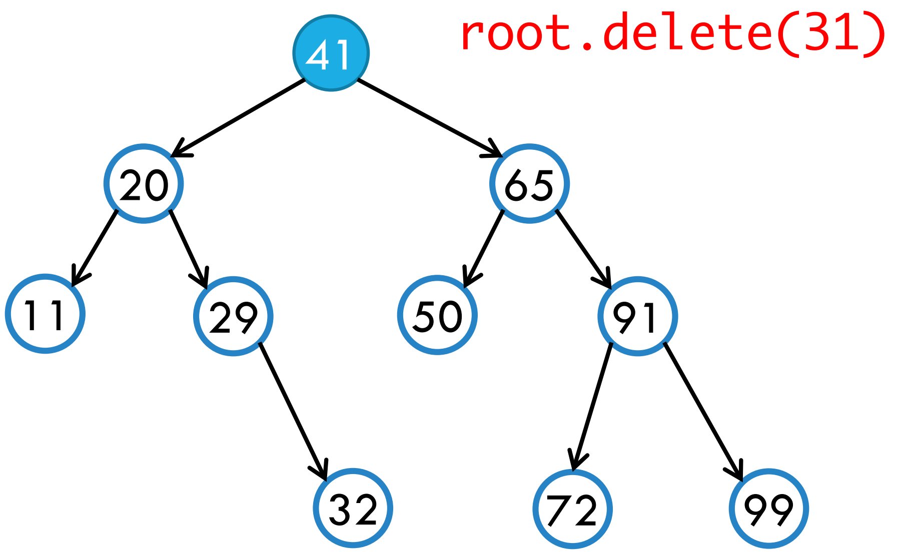
`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: __________
- Node is at a leaf: _____________
- Node is an internal node with 1 child: ______________
- Node is an internal node with 2 children: _______________

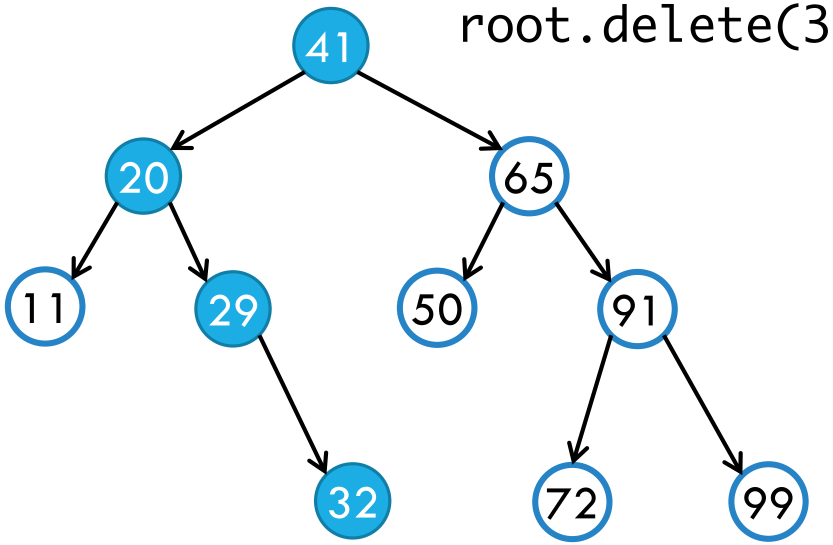


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: ______________
- Node is an internal node with 1 child: ______________
- Node is an internal node with 2 children: ______________

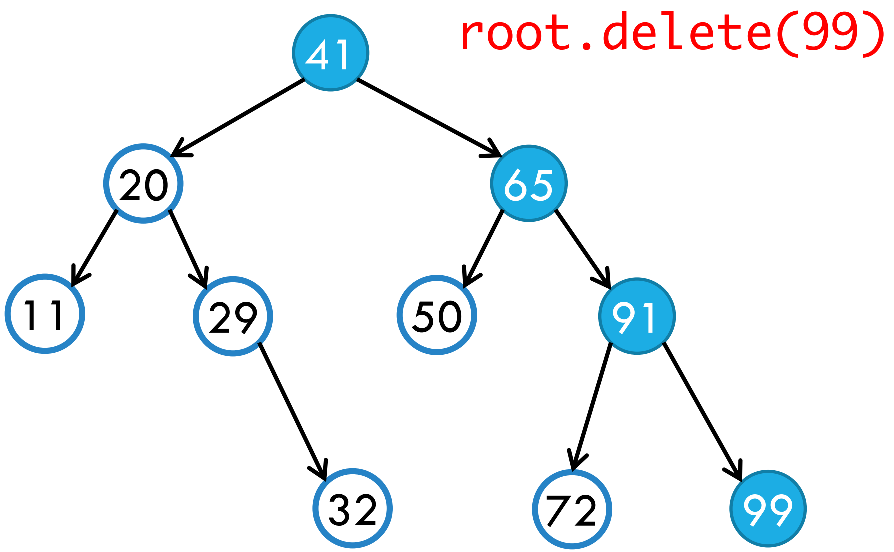
`root.delete(31)`



DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: ______________
- Node is an internal node with 1 child: ______________
- Node is an internal node with 2 children: ______________

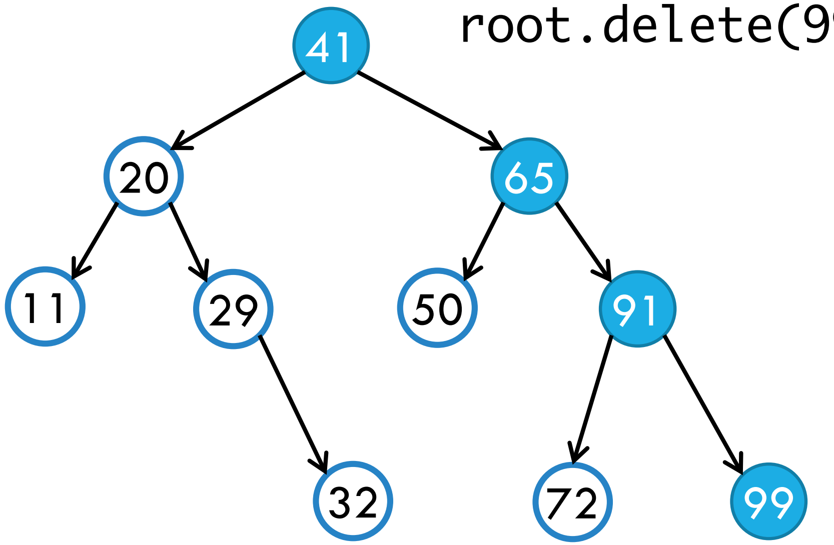


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: ________________
- Node is an internal node with 2 children: ________________

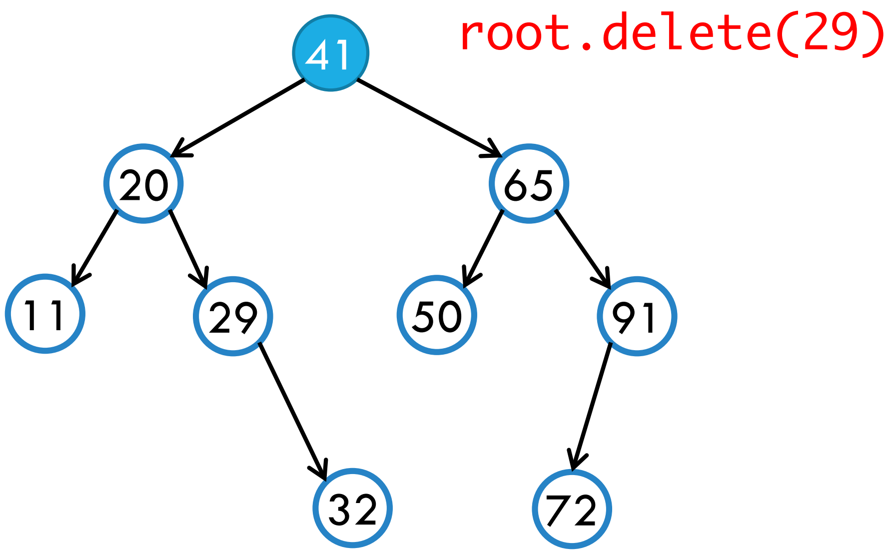
`root.delete(99)`



DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: ________________
- Node is an internal node with 2 children: _________________

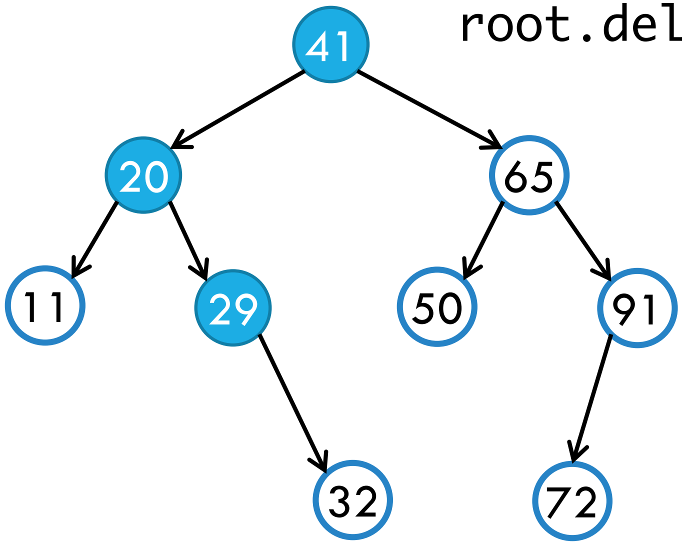


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: ________________
- Node is an internal node with 2 children: ________________

`root.delete(29)`

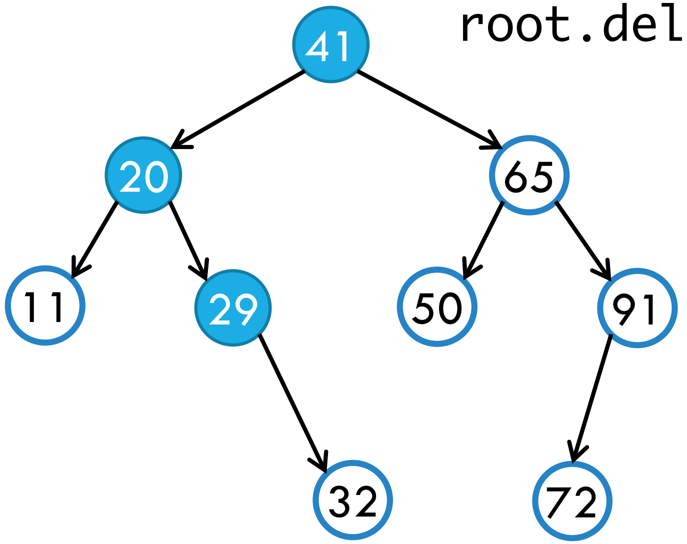


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children: ________________

root.delete(29)



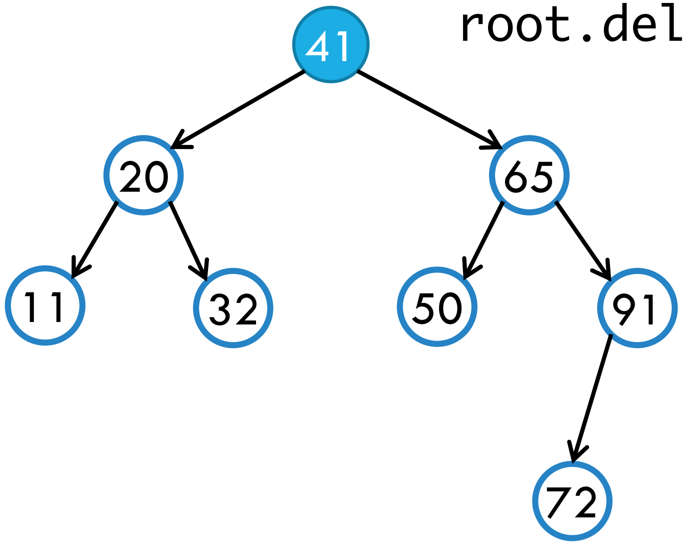
DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children:

____

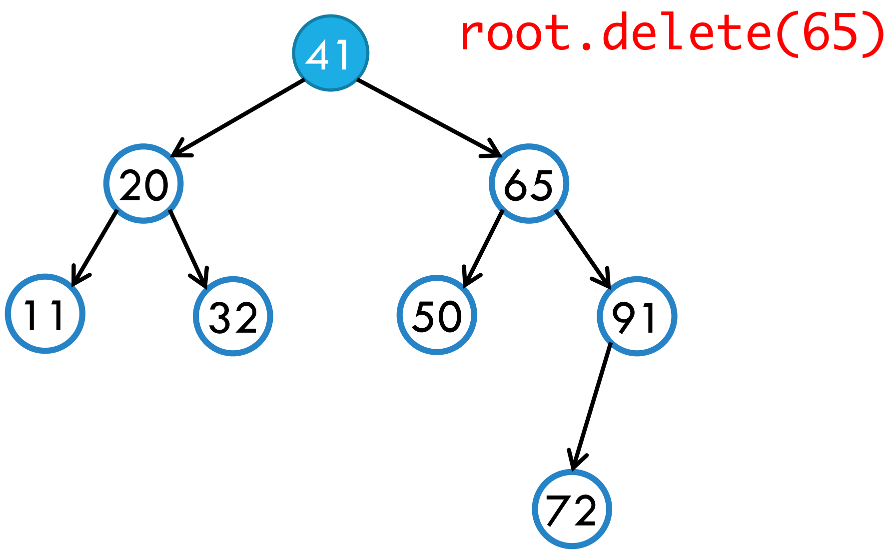
`root.delete(29)`



DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children: ________________

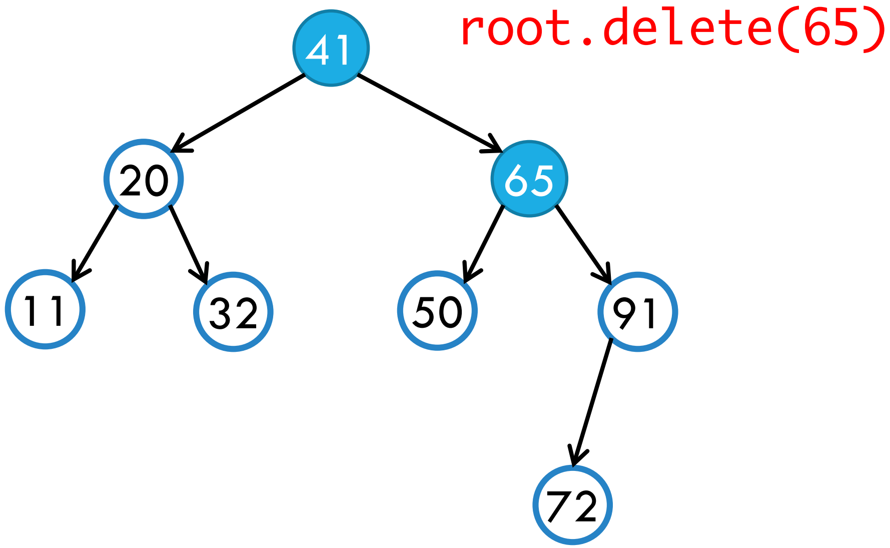


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children:

____

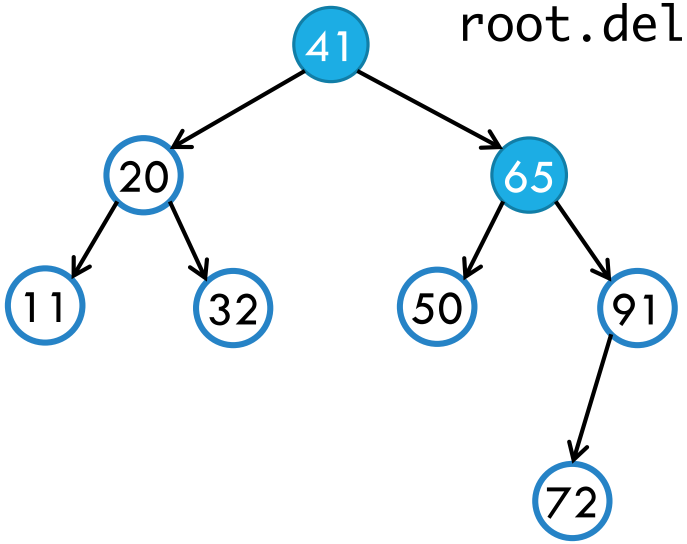


DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children: **replace node with its successor**

`root.delete(65)`

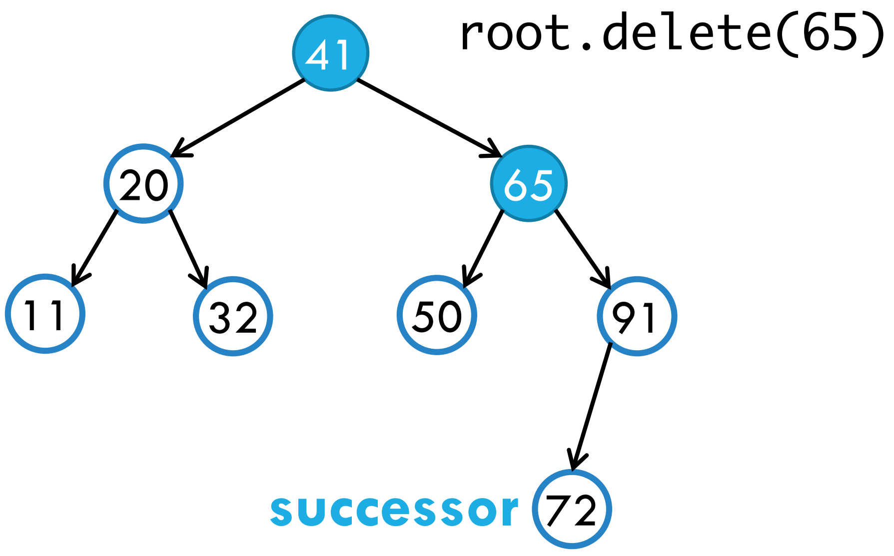


Successor = node with the smallest key larger than k

DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children: **replace node with its successor**

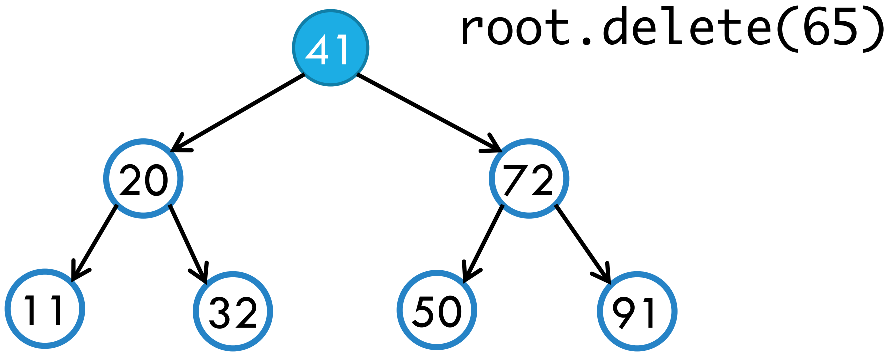


Successor = node with the smallest key larger than k

DELETING NODES

Idea: Search for the node, then handle one of 4 cases:

- The node is not found: **do nothing!**
- Node is at a leaf: **just remove!**
- Node is an internal node with 1 child: **remove node and connect child to parent**
- Node is an internal node with 2 children: **replace node with its successor**



Does replacing the node with the successor always work?

Successor = node with the smallest key larger than k

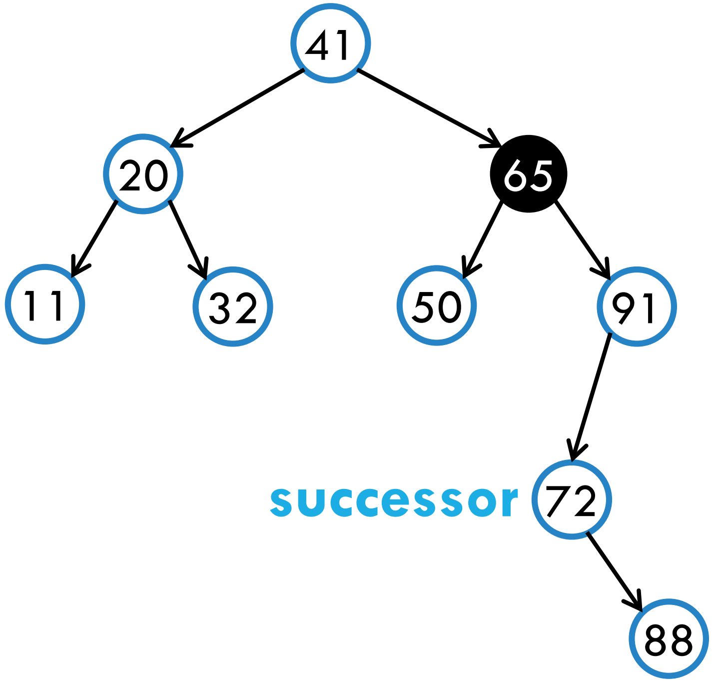
YES, IT DOES.

For this to work:

1. The successor *s* maintains the BST property when inserted to where k used to be.
2. The successor *s* has at most 1 child

Why do we want condition 2?

Because we can then just re-attach the successor's child to s's parent



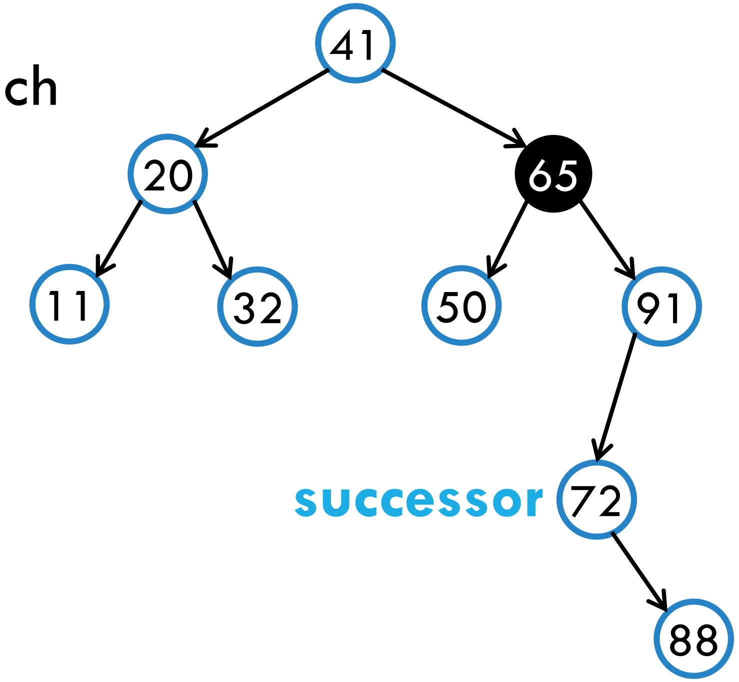
YES, IT DOES.

For this to work:

1. The successor *s* maintains the BST property when inserted to where k used to be.
2. The successor *s* has at most 1 child, so we can re-attach

Why does 1 hold?

- By definition $s > k$ so, s must be in k's right subtree.
- nodes in k's left subtree must be $< s$
- nodes k's right subtree (excluding s) must be $> s$
- Hence, inserting s to k's position is valid.



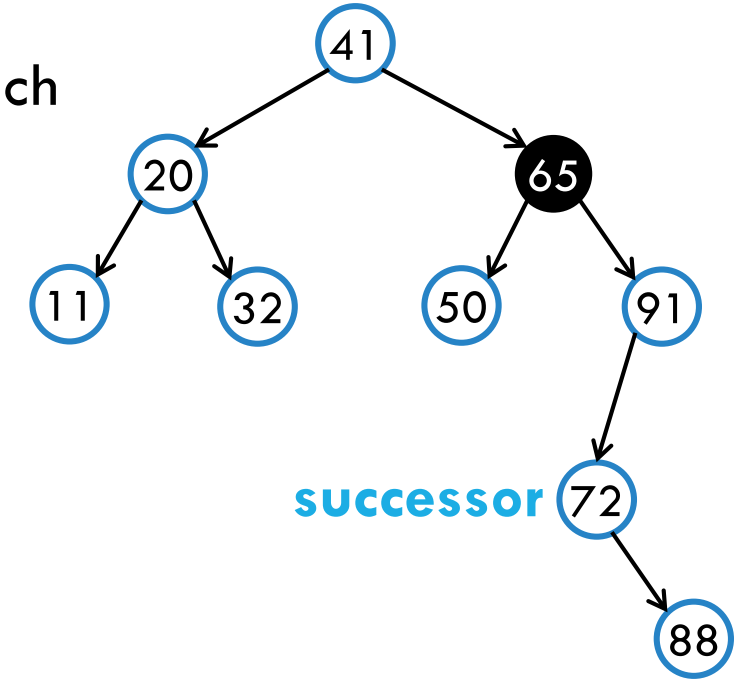
YES, IT DOES.

For this to work:

1. The successor *s* maintains the BST property when inserted to where k used to be.
2. The successor *s* has at most 1 child, so we can re-attach

Why does 2 hold?

- *s* is the minimum element of *k*'s right subtree.
- Can the minimum element of a BST have a left child? ______



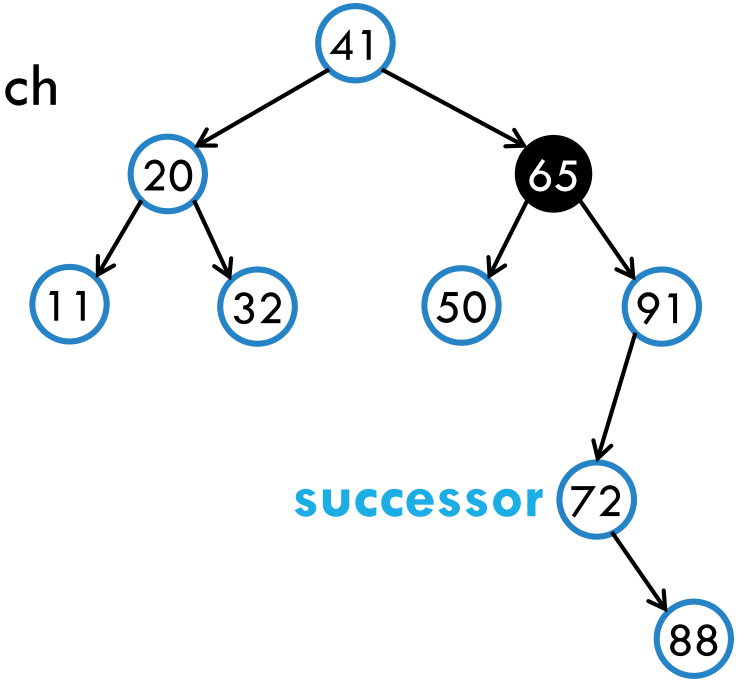
YES, IT DOES.

For this to work:

1. The successor *s* maintains the BST property when inserted to where k used to be.
2. The successor *s* has at most 1 child, so we can re-attach

Why does 2 hold?

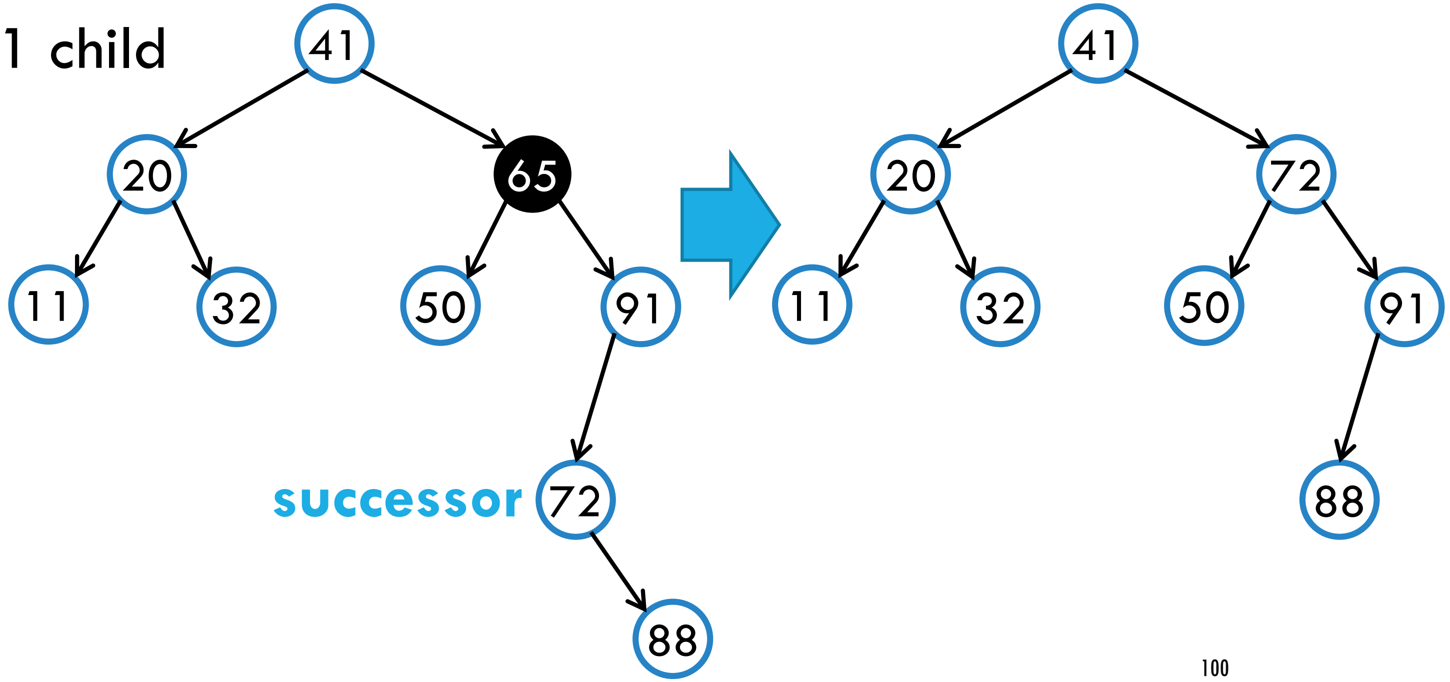
- *s* is the minimum element of *k*'s right subtree.
- Can the minimum element of a BST have a left child?
  No
- So, *s* has no children or a right child.



YES, IT DOES.

For this to work:

1. The successor *s* maintains the BST property when inserted to where k used to be.
2. The successor *s* has at most 1 child



BINARY SEARCH TREES (BST): BASIC OPERATIONS

`height()`: returns the height of the node

`search(k)` : returns an object v with key k

`searchMin()` : finds the minimum key k

`searchMax()` : finds the maximum key k

`successor()`: returns next key > k

`predecessor()`: returns next key < k

`insert(k,v)`: inserts an object v with key k

`delete(k)` : deletes the node with key k

OPERATIONS & COSTS

height(): $O(n)$

search(k) : $O(h)$

searchMin() : $O(h)$

searchMax() : $O(h)$

successor(): $O(h)$

predecessor(): $O(h)$

insert(k,v): $O(h)$

delete(k) : $O(h)$

if the tree is imbalanced, $O(n)$

THE ORDERED DICTIONARY ADT

Operations with k = key, v = value:

✓ `insert(k, v)`: inserts an element with value v and key k

✓ `search(k)`: returns the value with key k

✓ `delete(k)`: deletes the element with key k

✓ `contains(k)`: true if the dictionary contains an element with key k

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

✓ `size()`: returns the size of the dictionary

*also called "Ordered Symbol Tables"

PROBLEM: POVERTY IDENTIFICATION

The Stop-Poverty charity calls:

To provide financial aid, Help identify families:

- earning exactly $a amount
- earning less than or equal to $a
- earning more than or equal to $a

NOT SO BASIC BST OPERATIONS

`floor(k)`: returns next key $\leq$ k

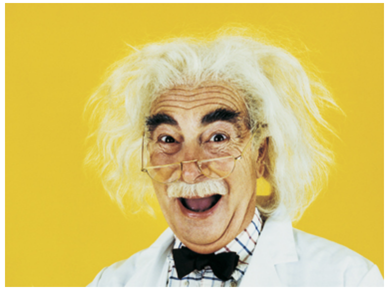
`ceiling(k)`: returns next key $\geq$ k

`inorder(Node o)`: returns nodes of the tree rooted at o in order

`copyTree(Node o)`: returns a copy of the tree rooted at o

`deleteTree(Node o)`: deletes the tree rooted at o

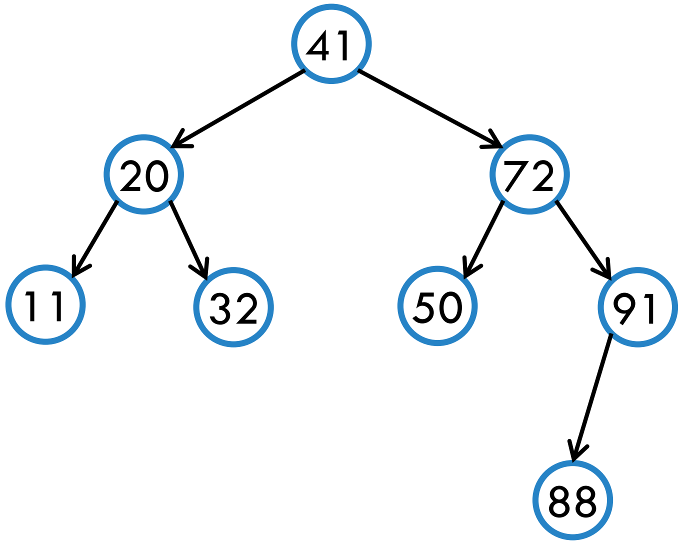
Just for fun!



CEILING FOR KEY K

Question: for a given key k, how do we find the ceiling (the smallest key in the BST larger or equal to k)?

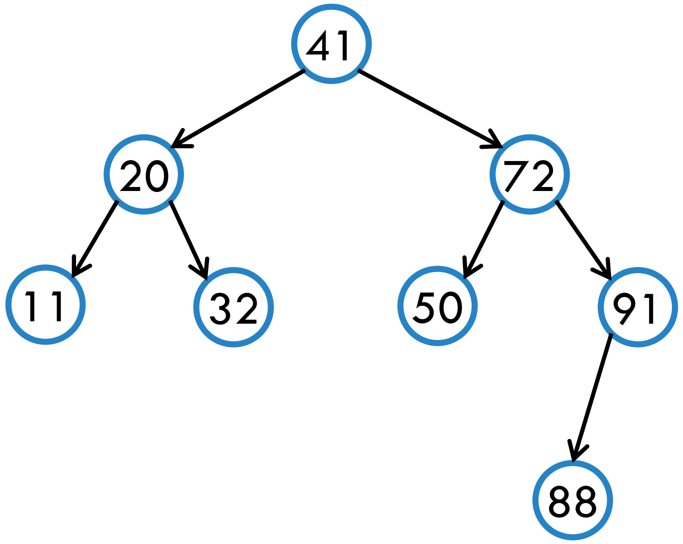
```
ceiling(root, k)
```



CEILING FOR KEY K

Question: for a given key k, how do we find the ceiling (the smallest key in the BST larger or equal to k)?

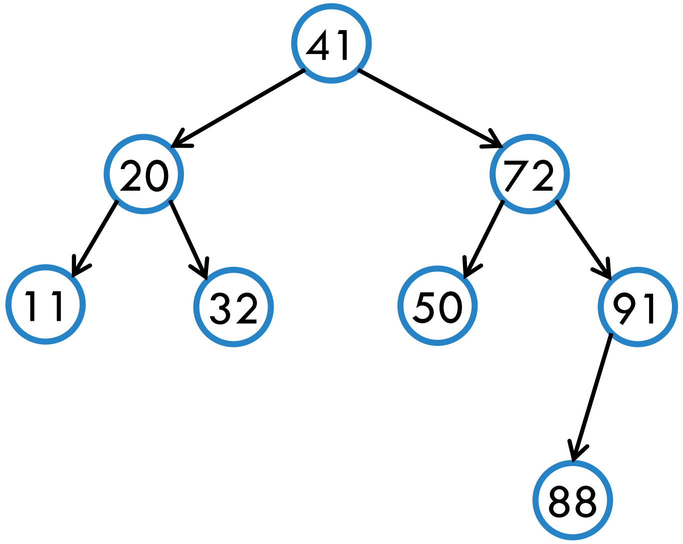
```
ceiling(root, k)

function ceiling(Node x, k)
    if x is null then return null
    if x.key == k then ?
    if k > x.key then
        ?
    if k < x.key then
        ?
```



CEILING FOR KEY K

Question: for a given key k, how do we find the ceiling (the smallest key in the BST larger or equal to k)?

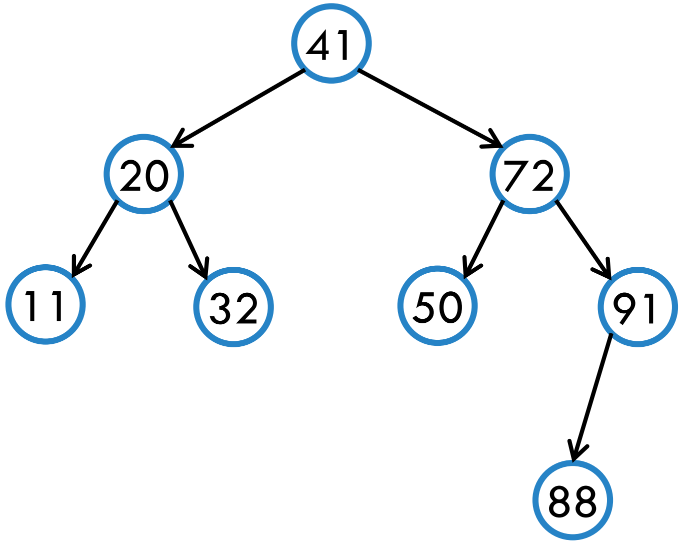
```
ceiling(root, k)

function ceiling(Node x, k)
    if x is null then return null
    if x.key == k then return x
    if k > x.key then
        ?
    if k < x.key then
        ?
```



CEILING FOR KEY K

Question: for a given key k, how do we find the ceiling (the smallest key in the BST larger or equal to k)?

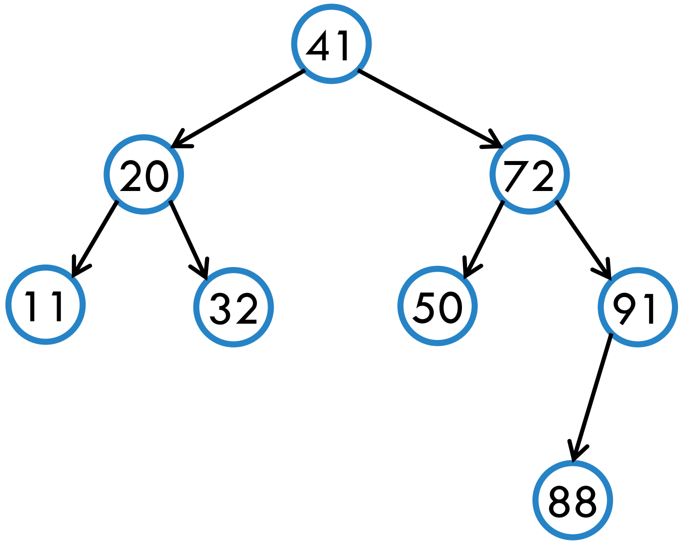
```
ceiling(root, k)

function ceiling(Node x, k)
    if x is null then return null
    if x.key == k then return x
    if k > x.key then
        return ceiling(x.rightTree, k)
    if k < x.key then
        ?
```



CEILING FOR KEY K

Question: for a given key k, how do we find the ceiling (the smallest key in the BST larger or equal to k)?

```
ceiling(root, k)

function ceiling(Node x, k)
    if x is null then return null
    if x.key == k then return x
    if k > x.key then
        return ceiling(x.rightTree, k)
    if k < x.key then
        t = ceiling(x.leftTree, k)
        if t is not null then return t
        else return x
```



Floor is similar. Try it out!

NOT SO BASIC BST OPERATIONS

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

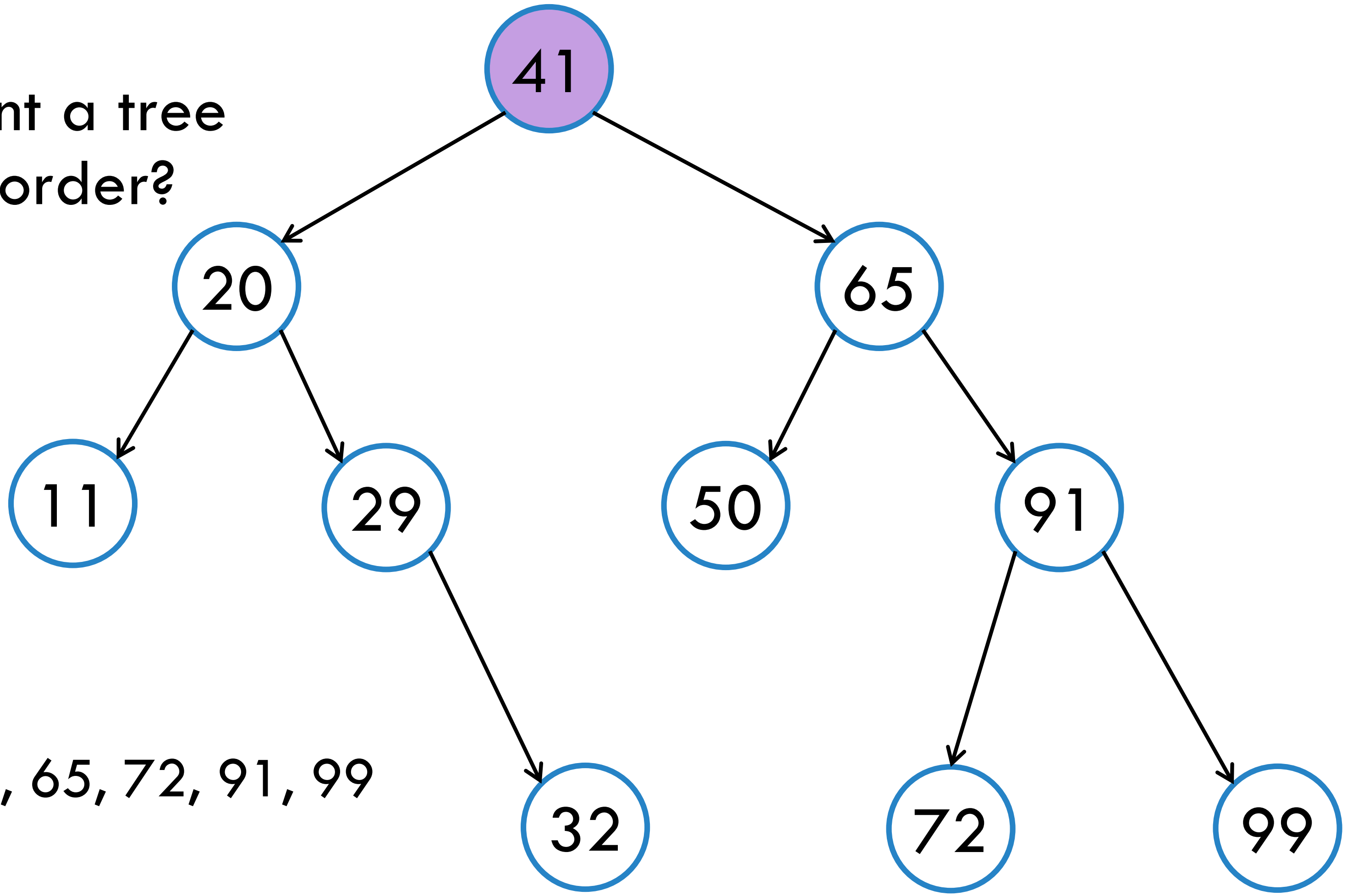
`inorder(Node o)`: returns nodes of the tree rooted at o in order

`copyTree(Node o)`: returns a copy of the tree rooted at o

`deleteTree(Node o)`: deletes the tree rooted at o

PRINTING THE NODES IN ORDER

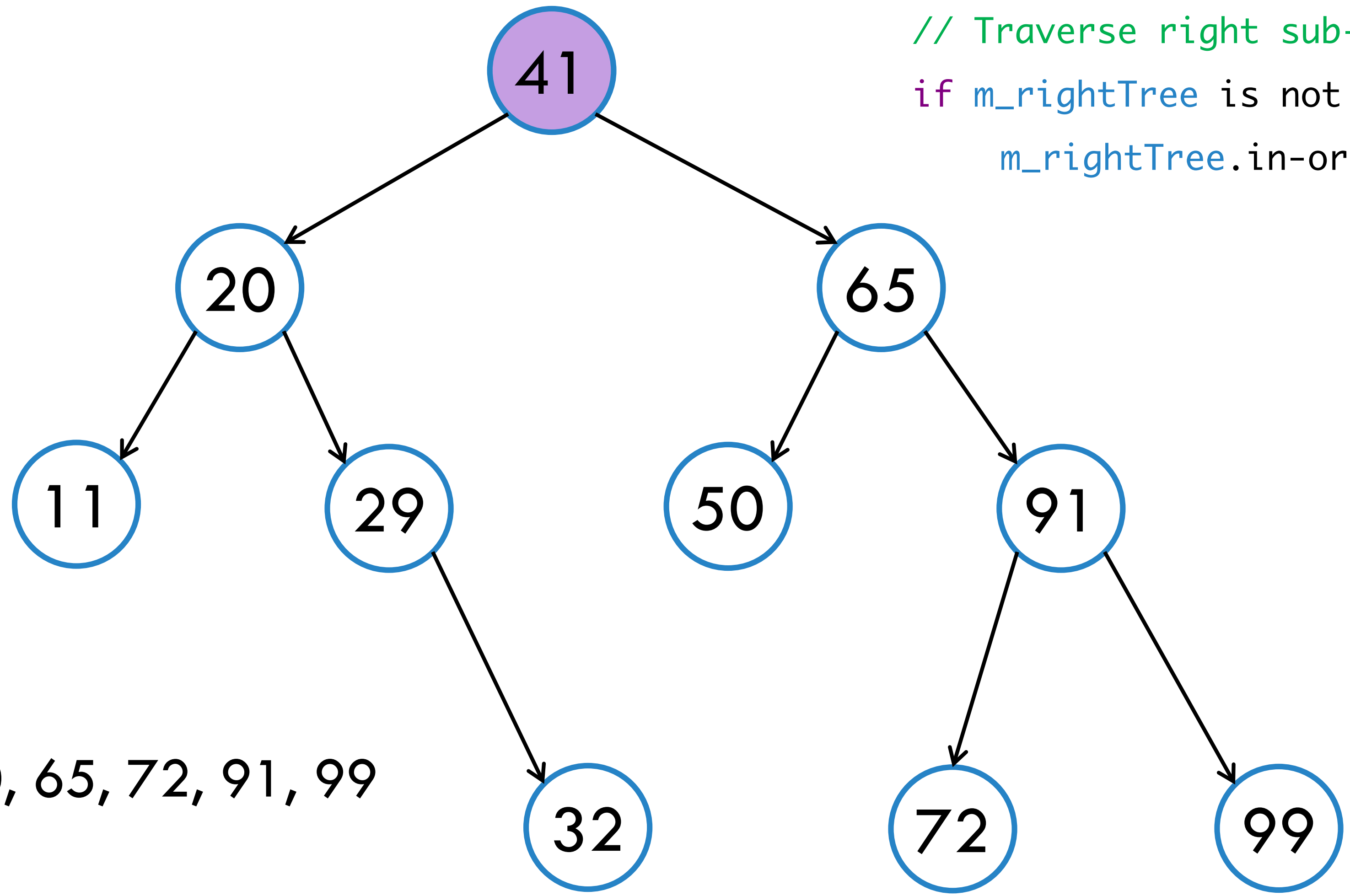
How can we print a tree rooted at 41 in order?



11, 20, 29, 32, 41, 50, 65, 72, 91, 99

PRINTING THE NODES IN ORDER

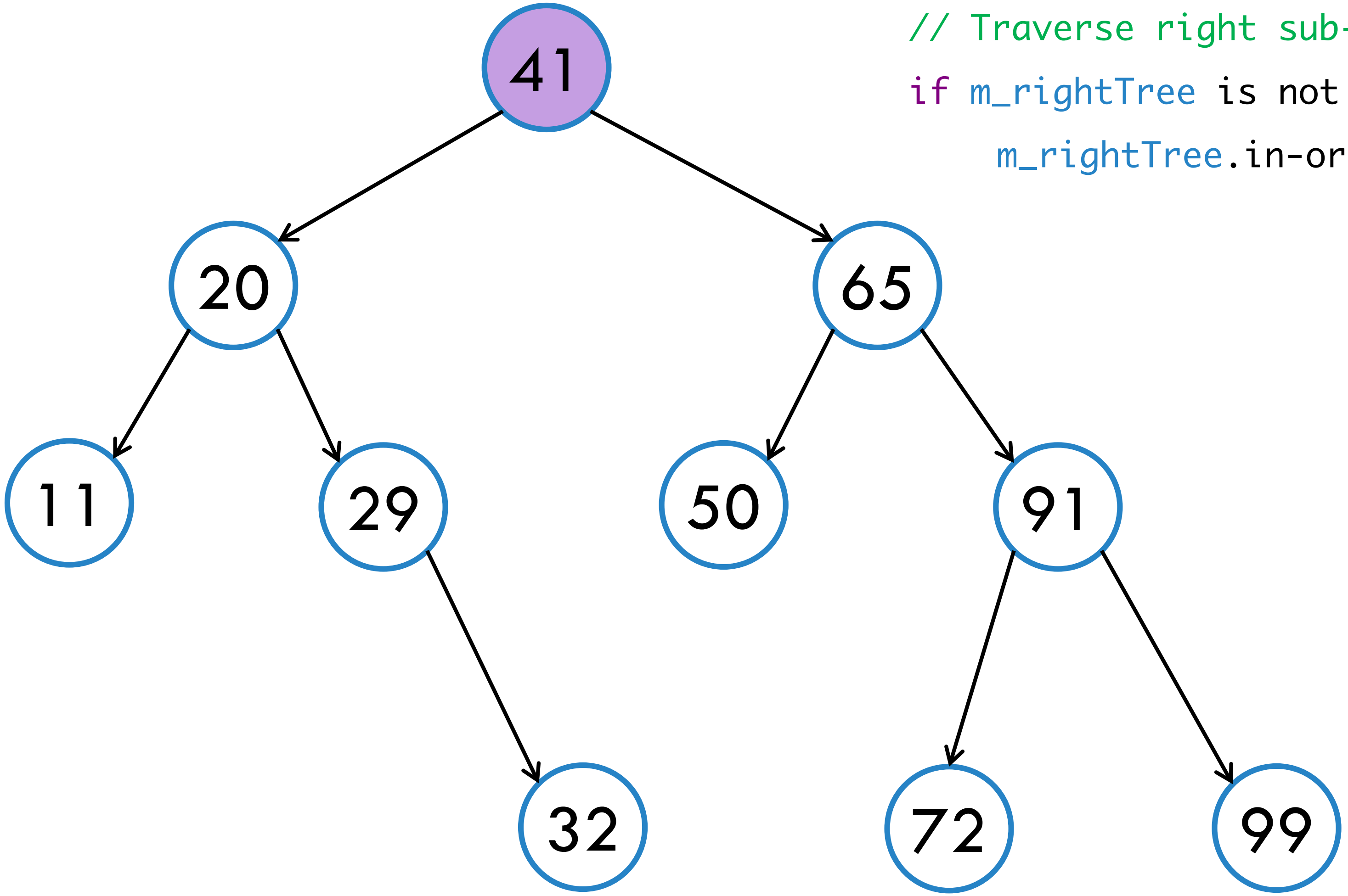
```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



11, 20, 29, 32, 41, 50, 65, 72, 91, 99

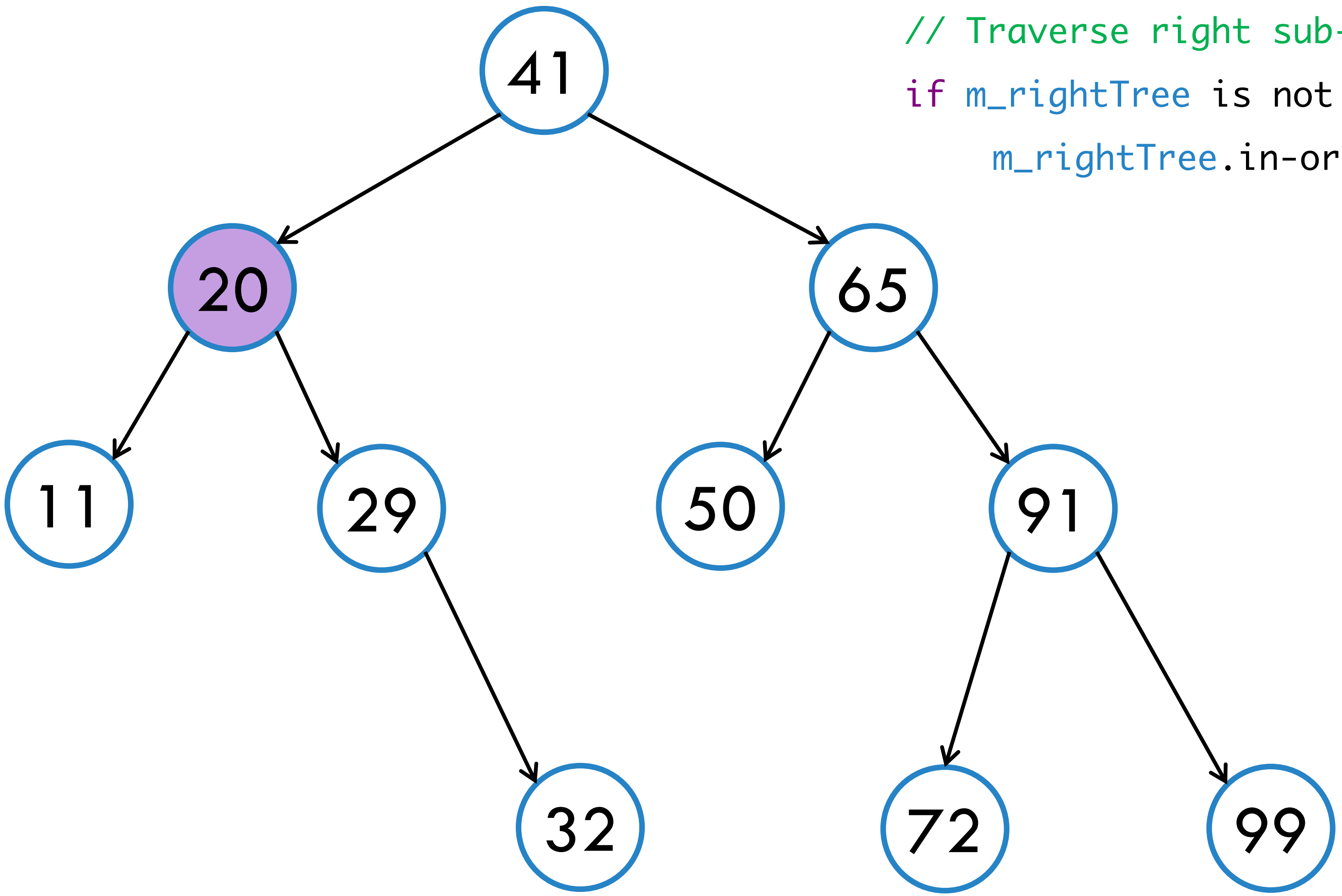
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



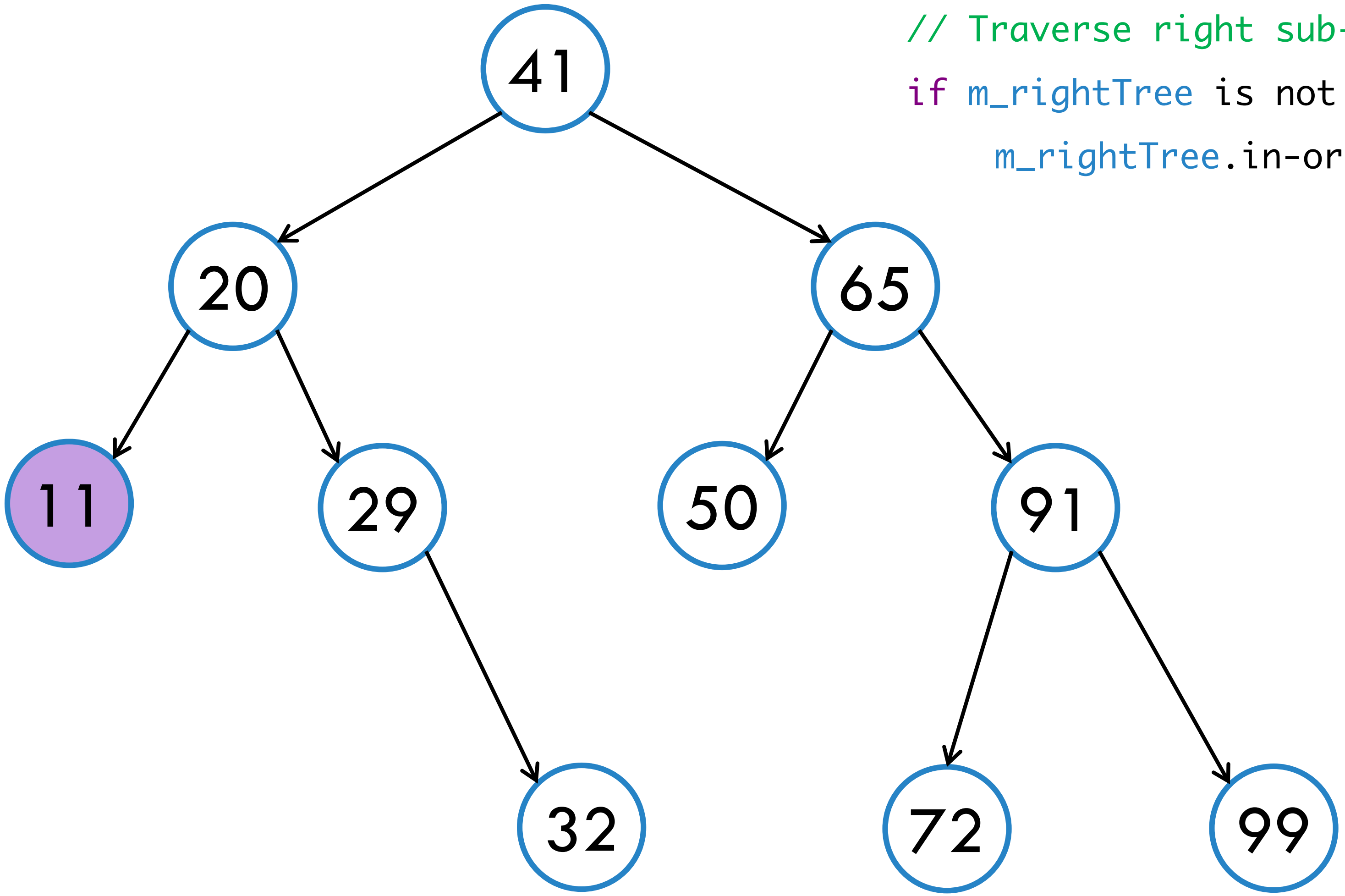
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



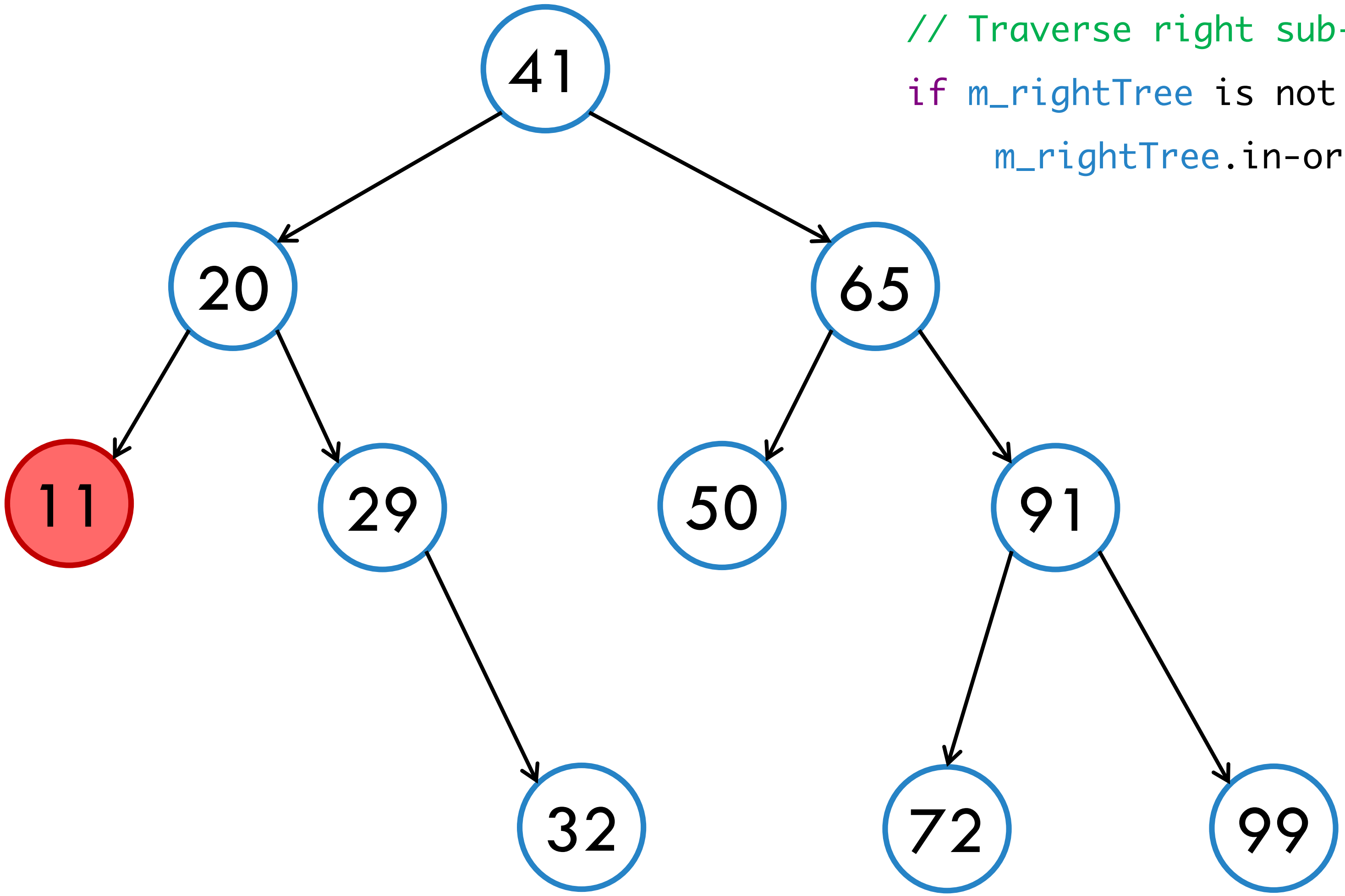
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



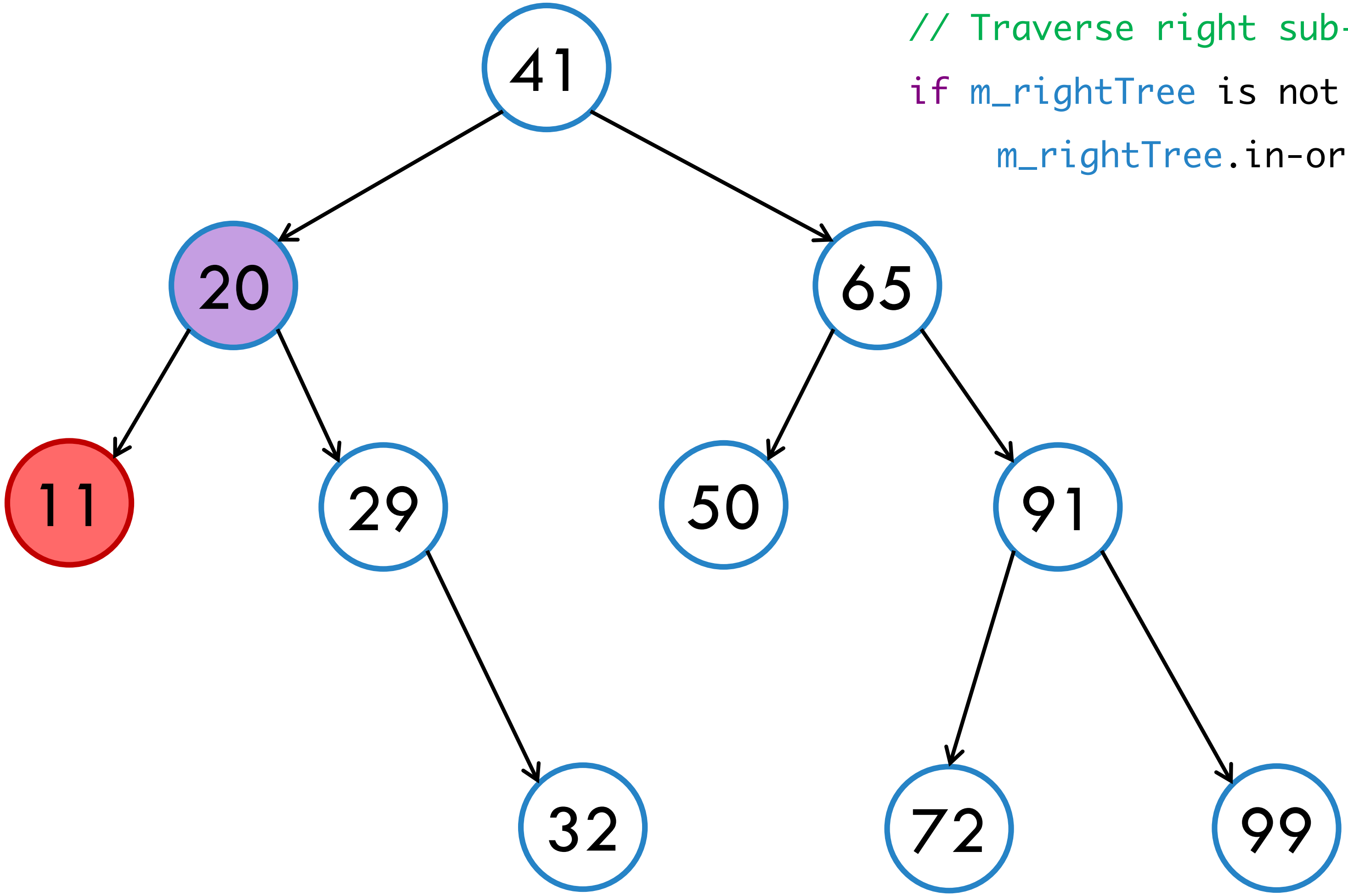
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



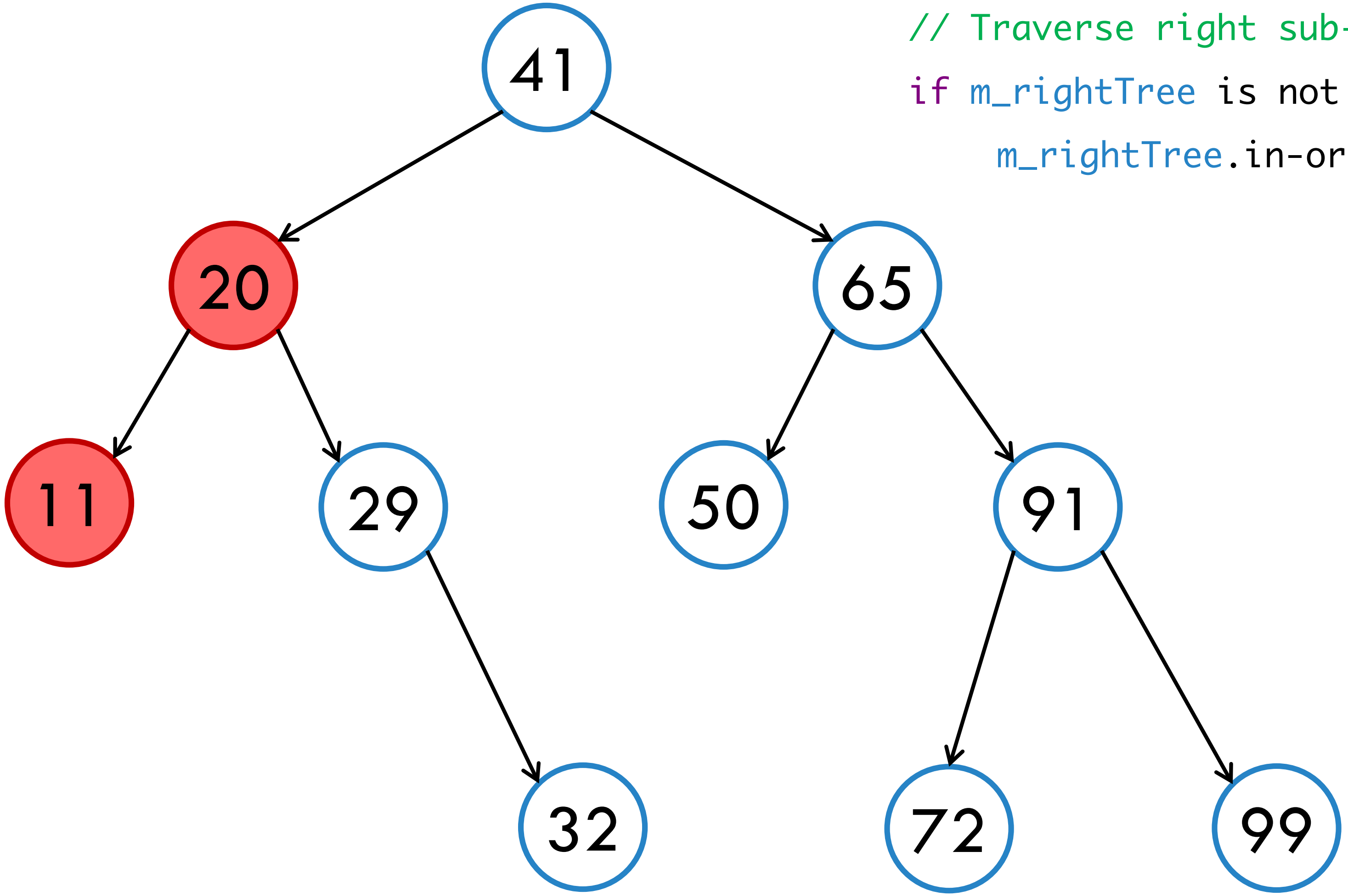
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



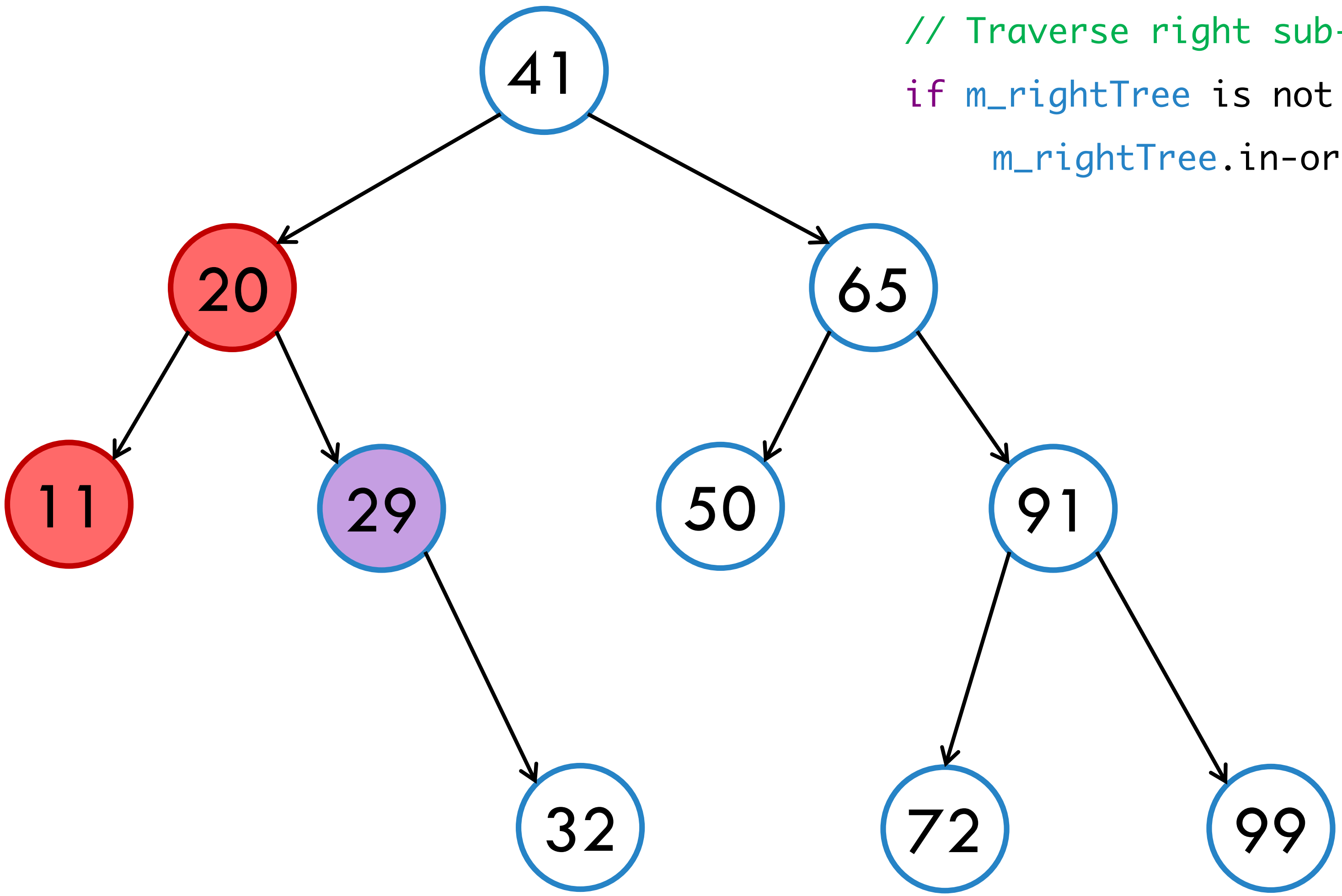
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



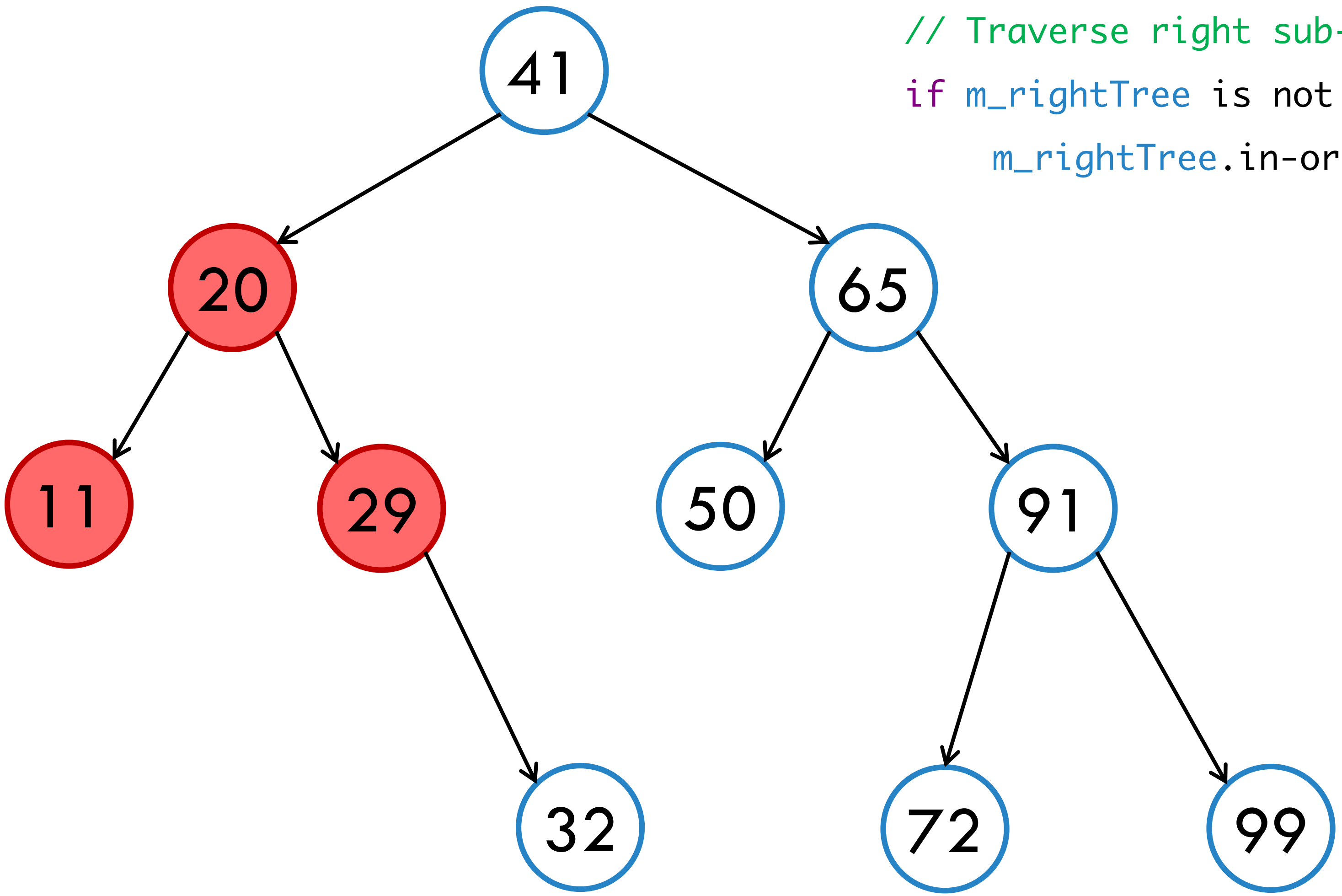
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



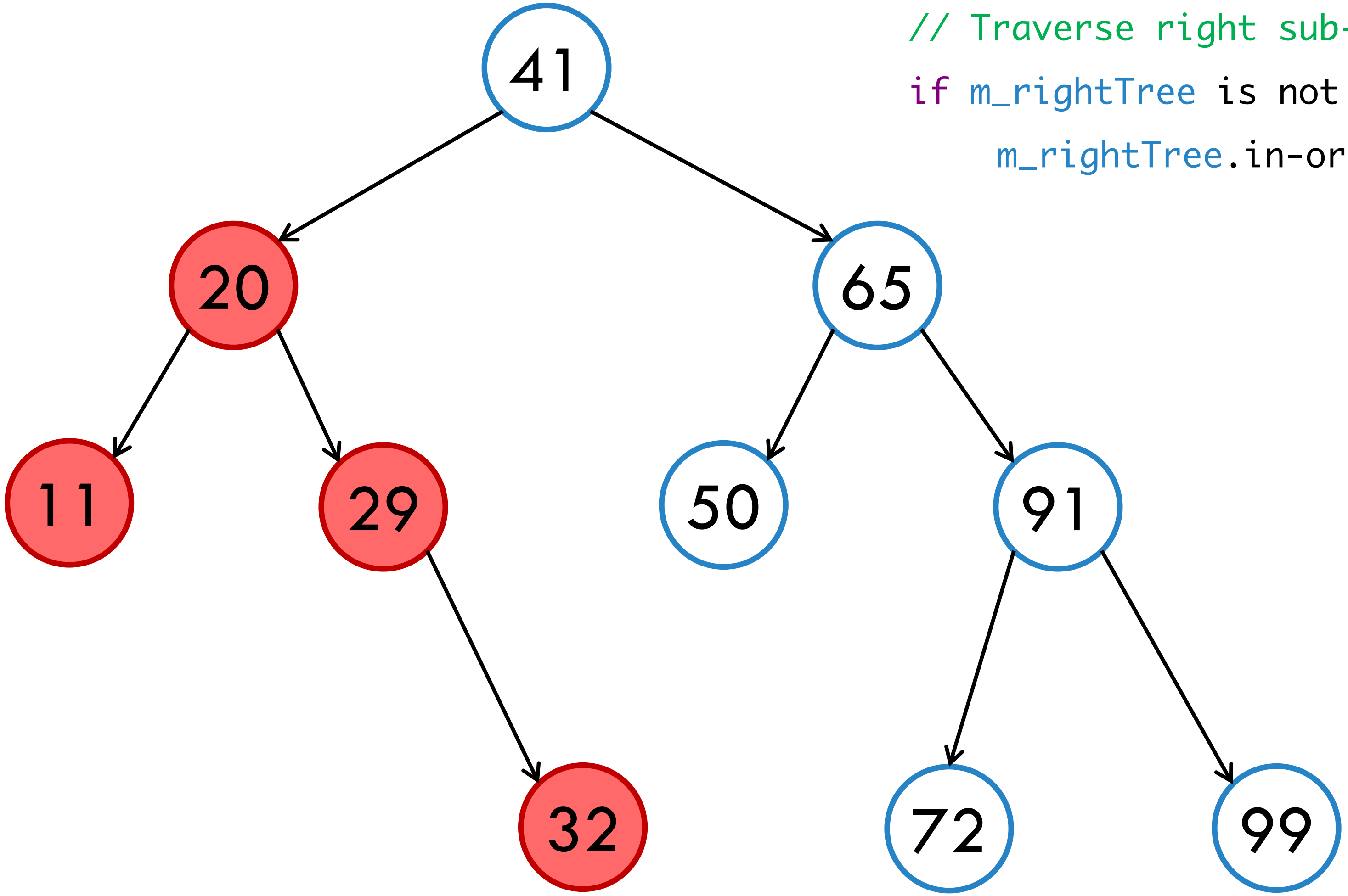
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



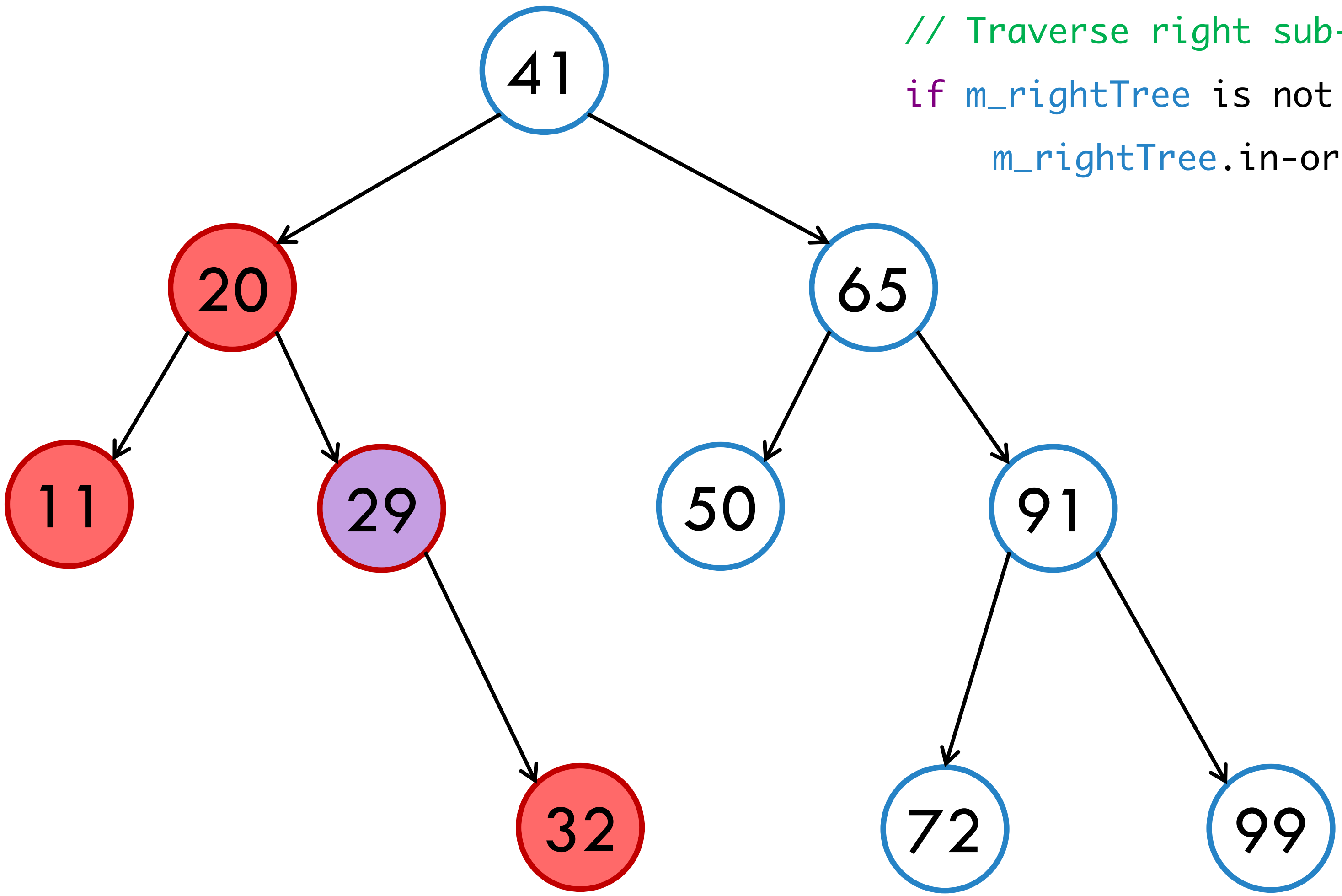
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



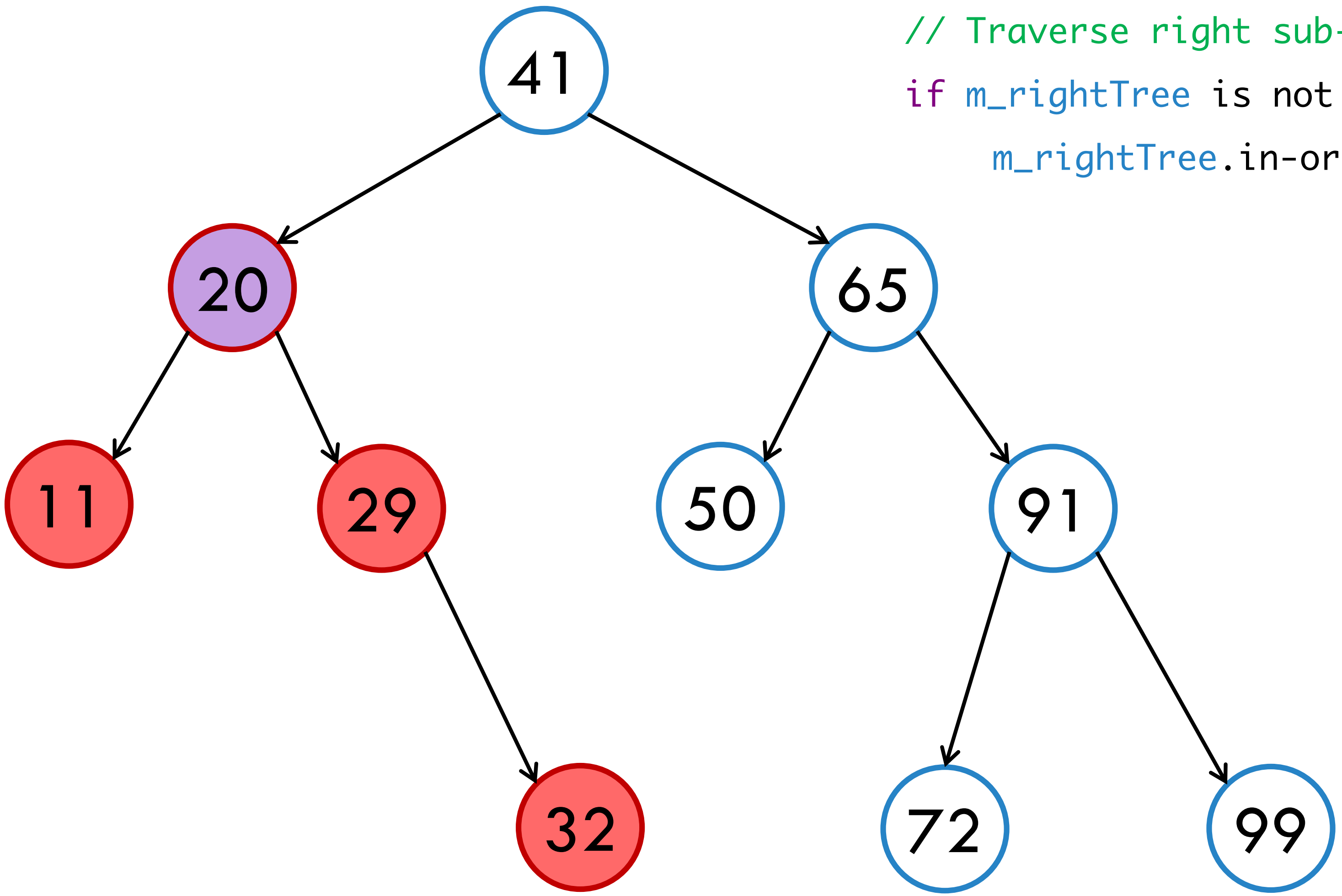
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



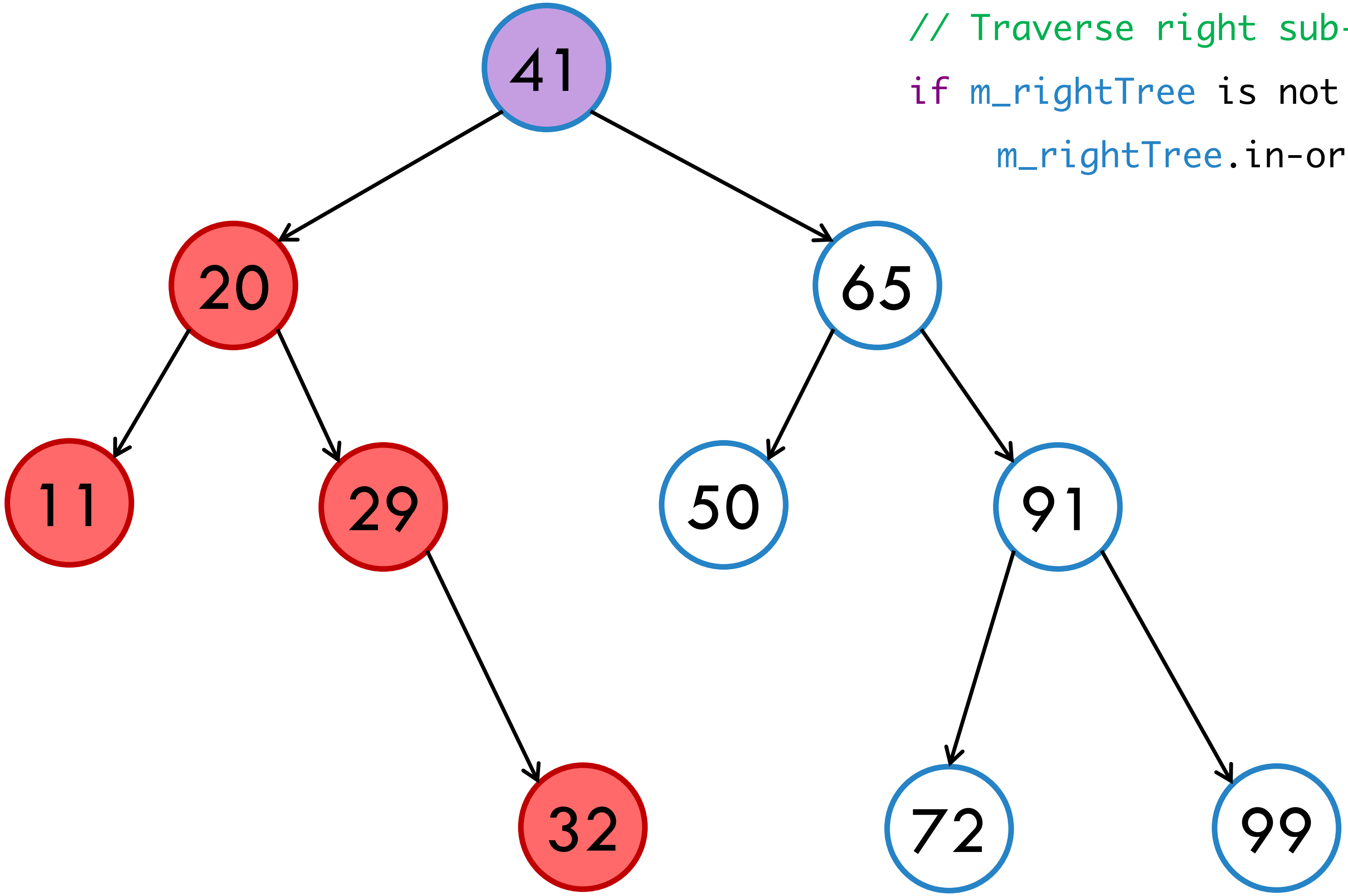
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



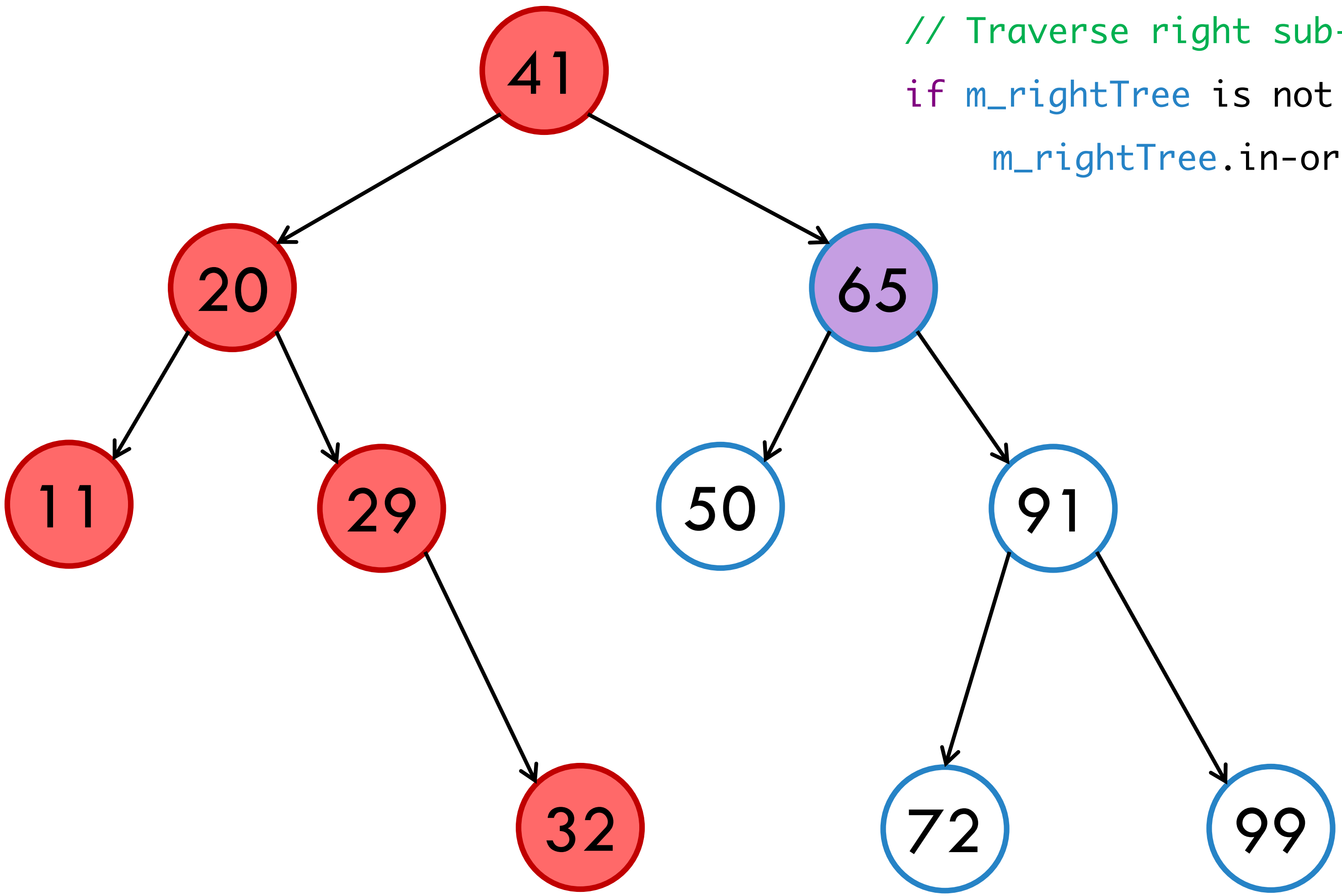
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



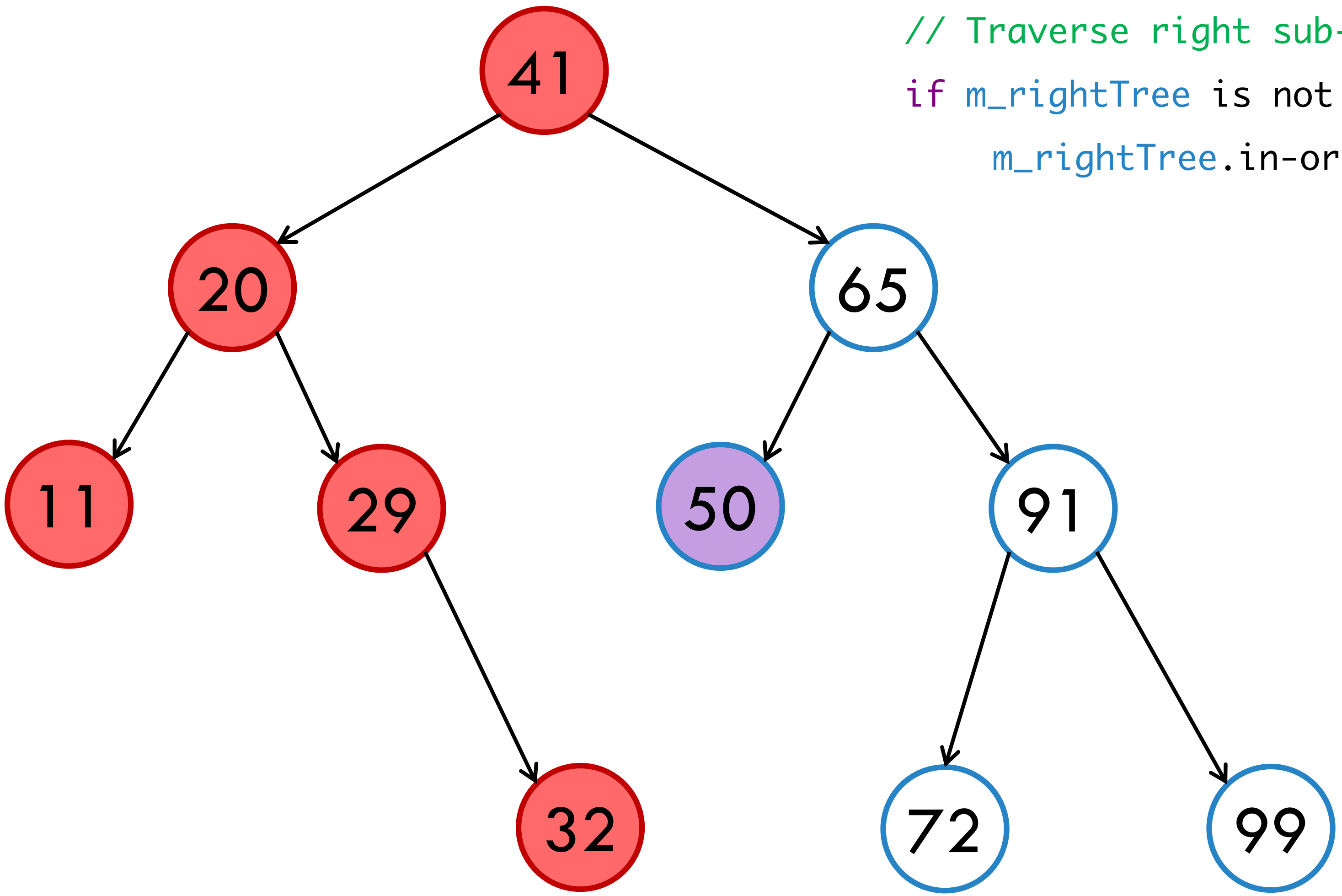
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



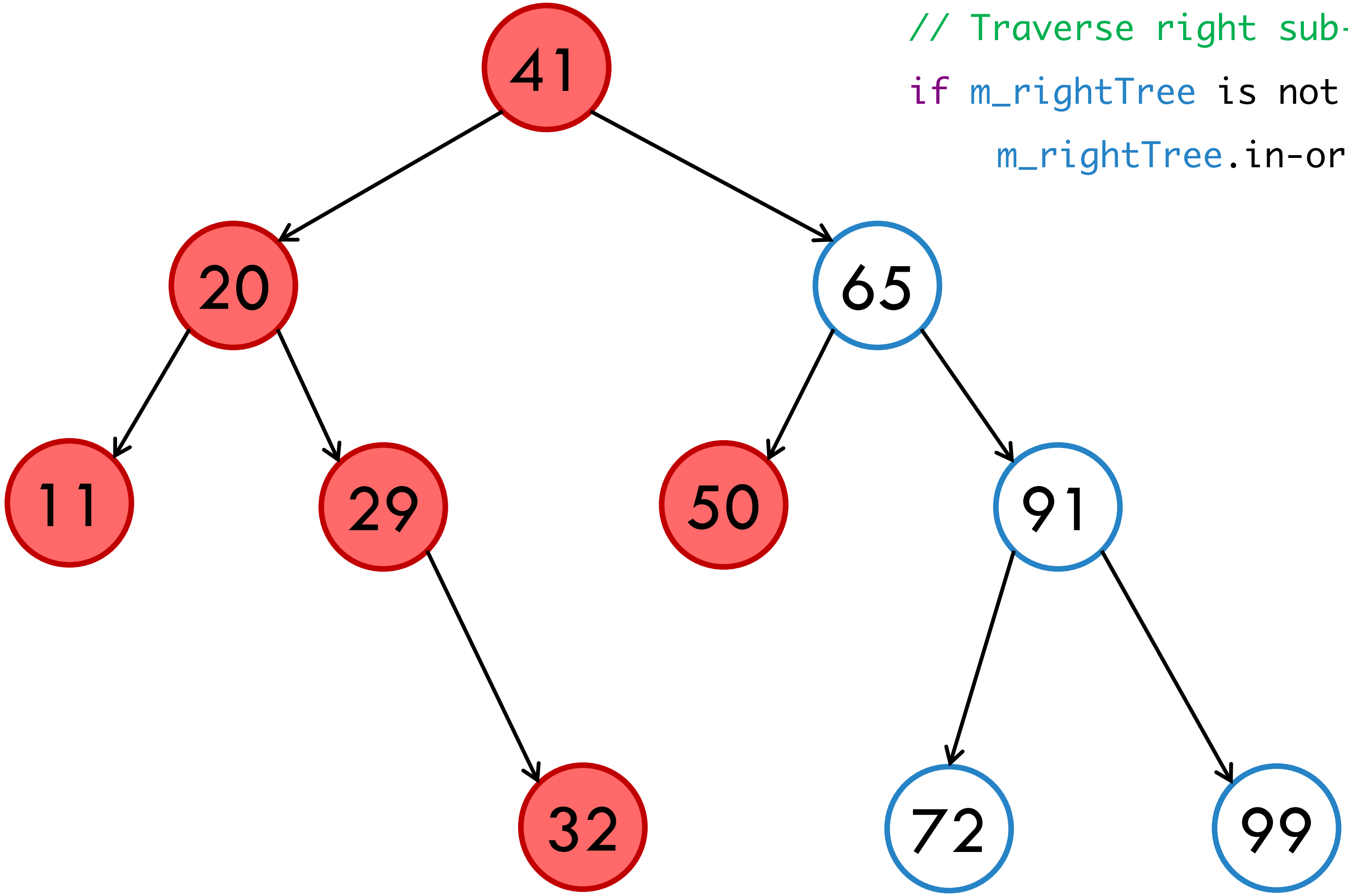
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



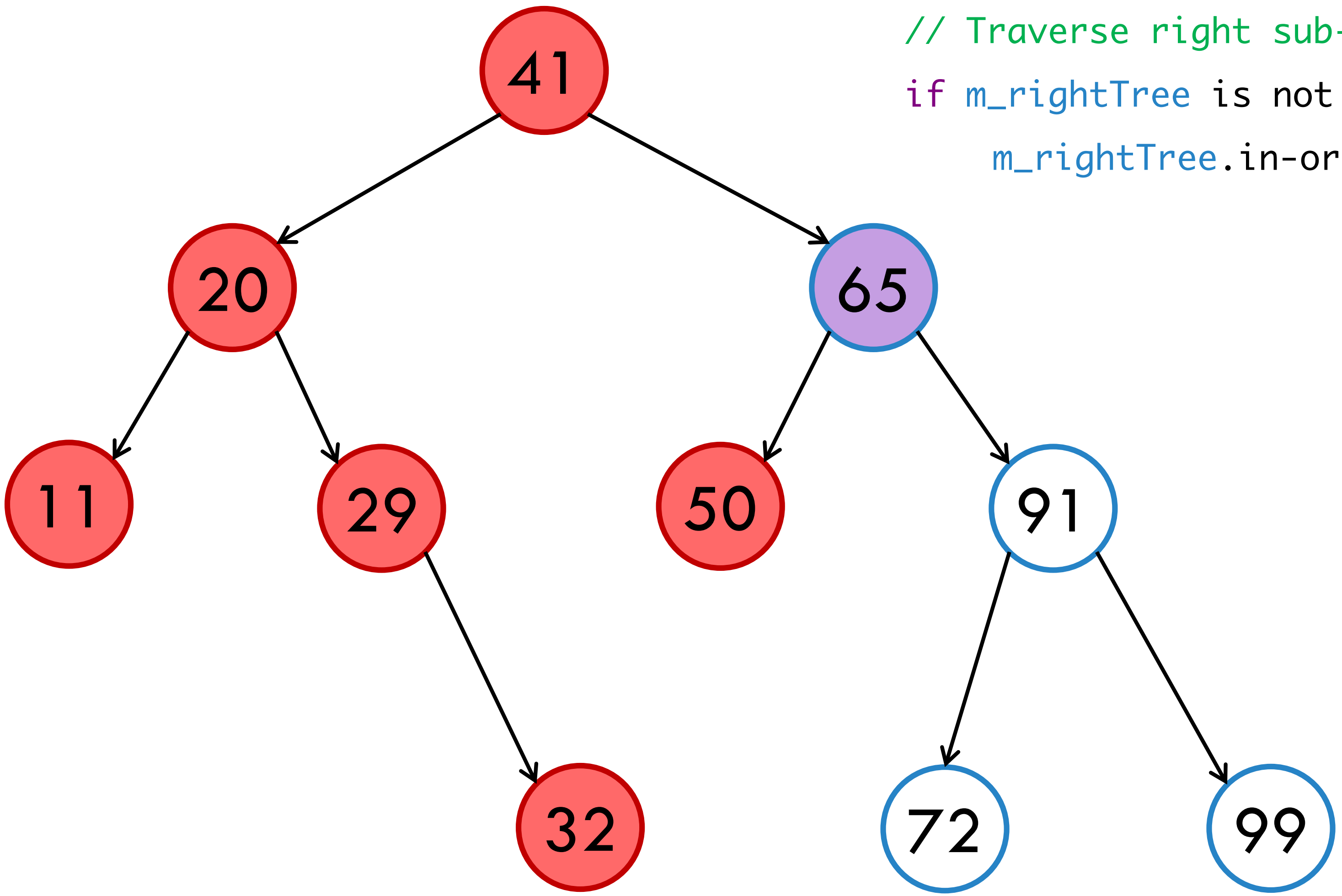
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



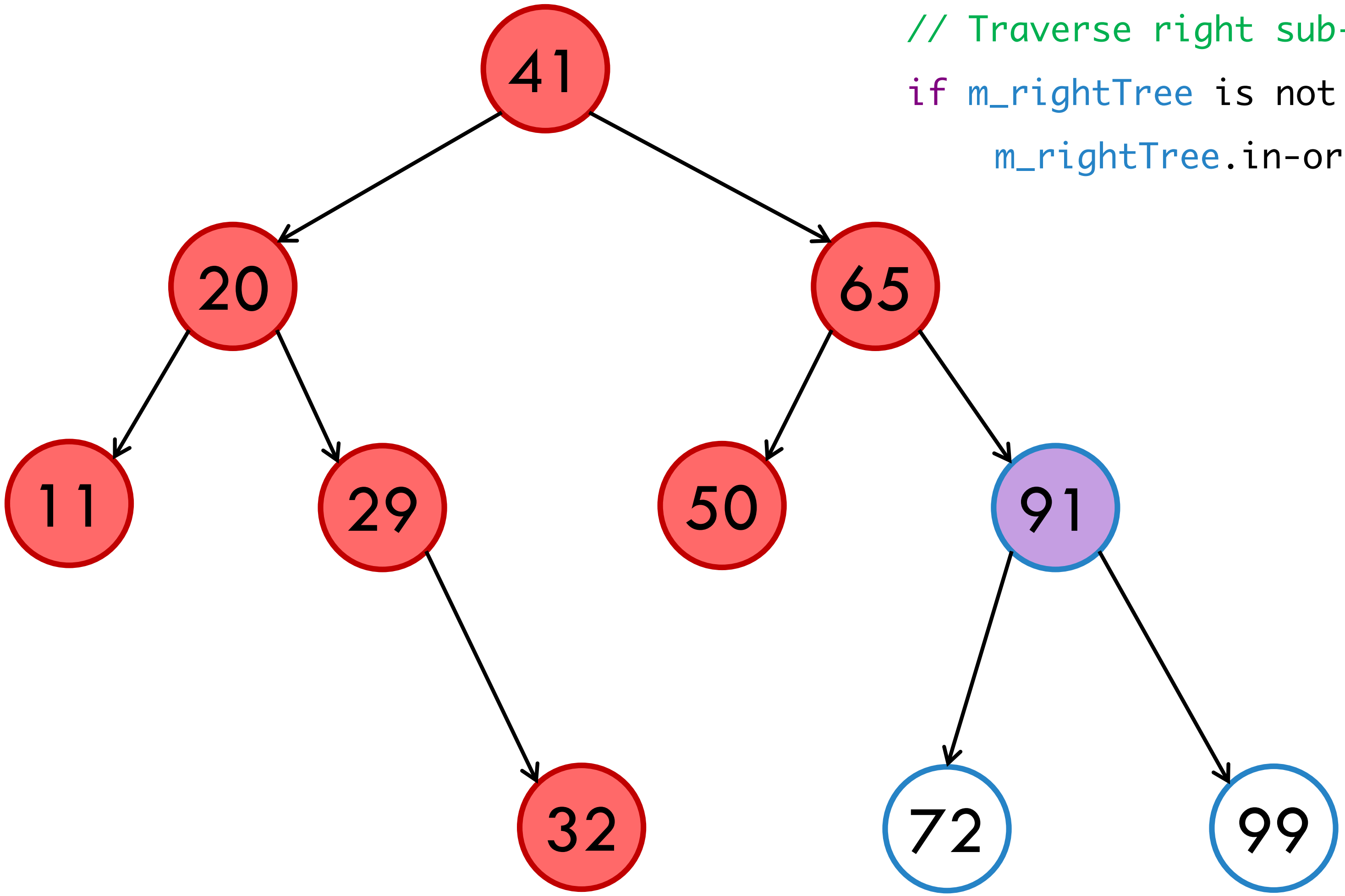
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



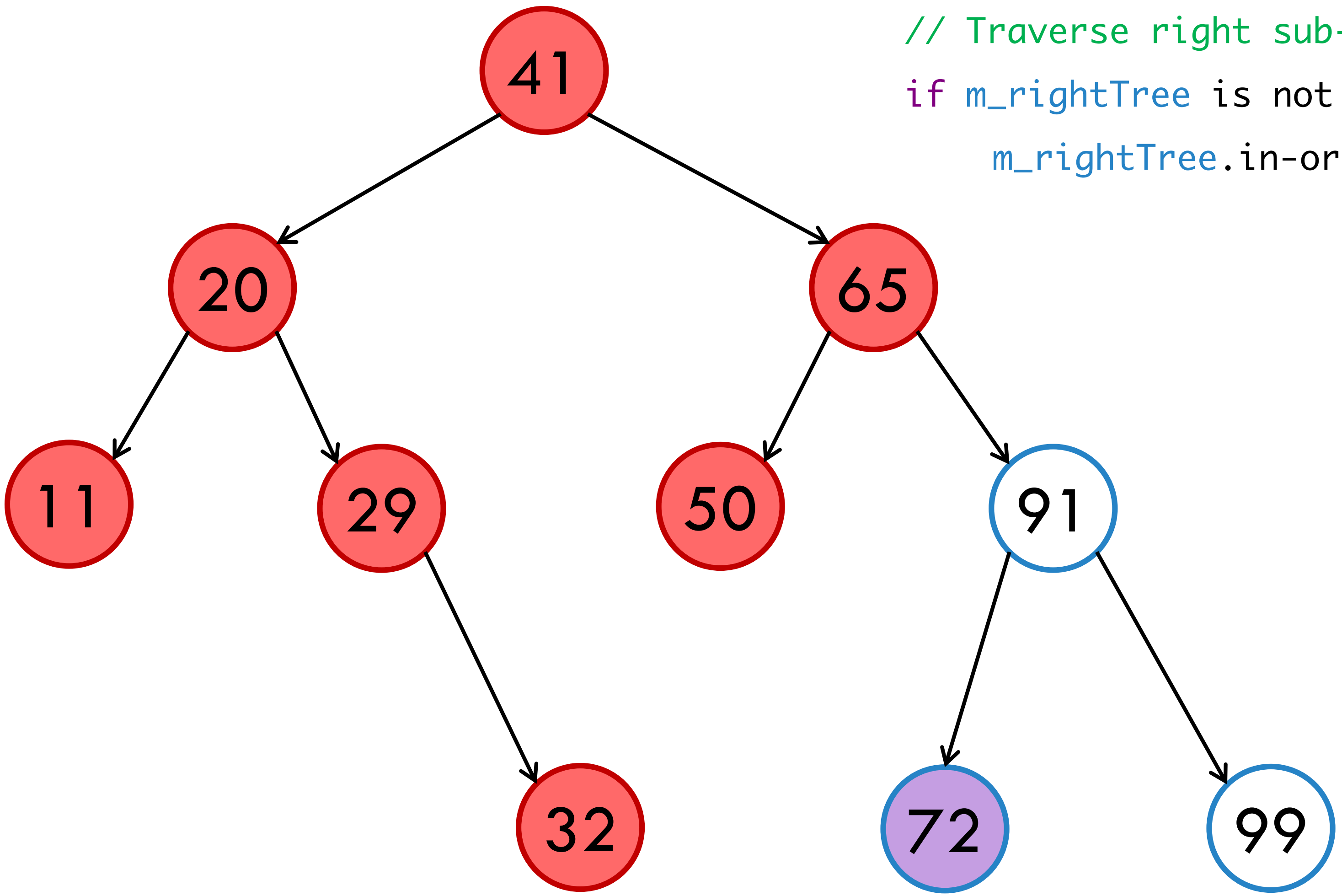
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



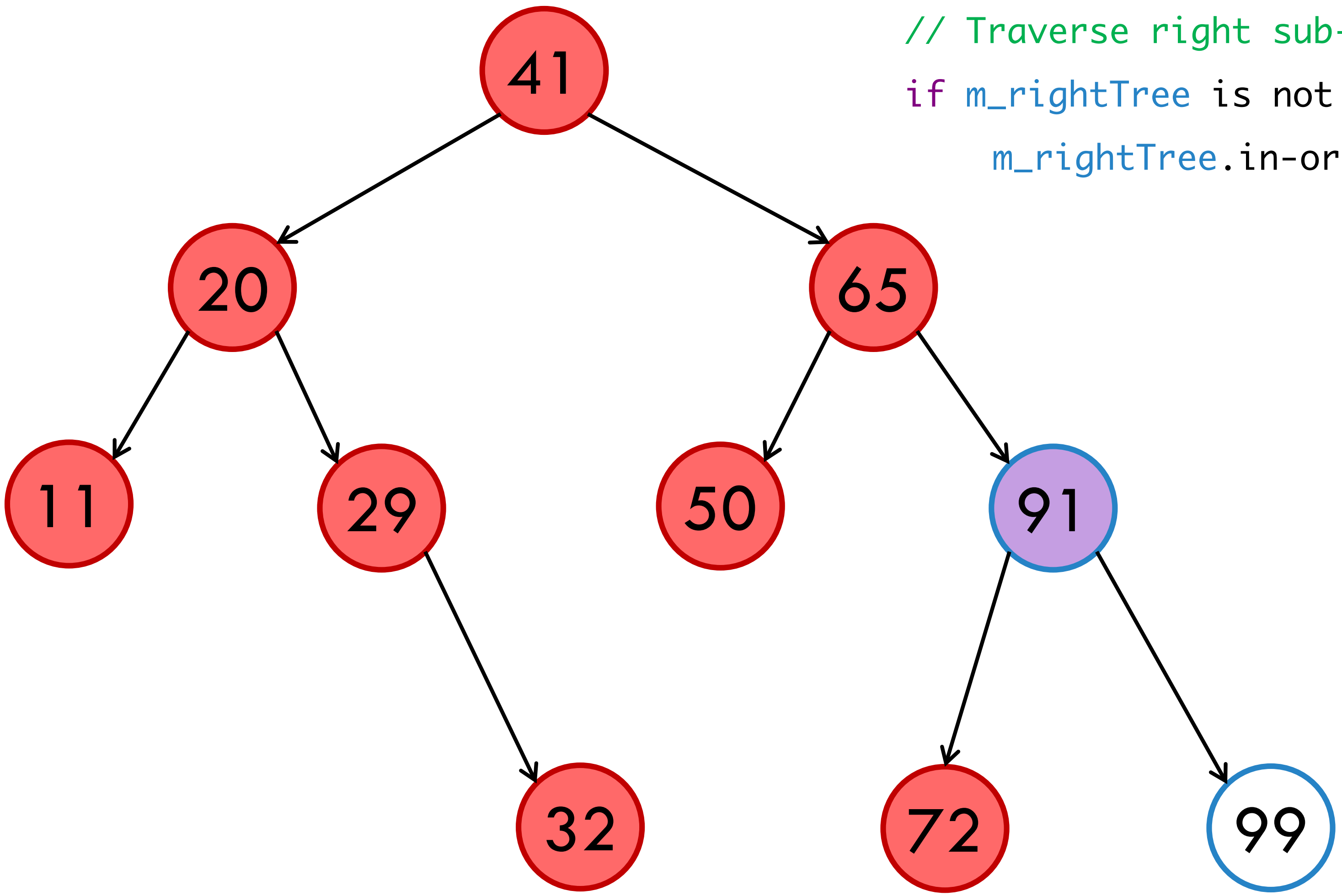
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



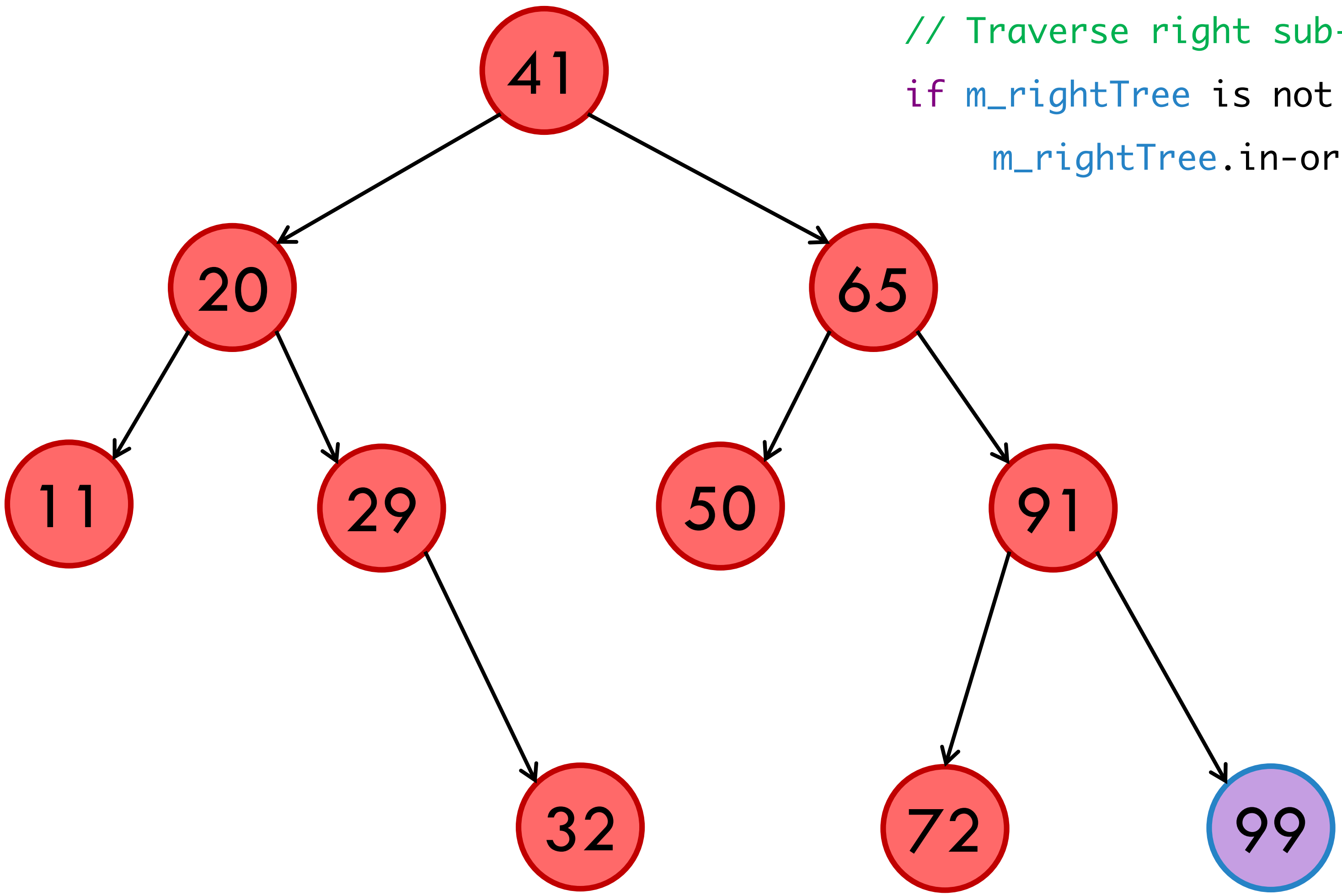
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



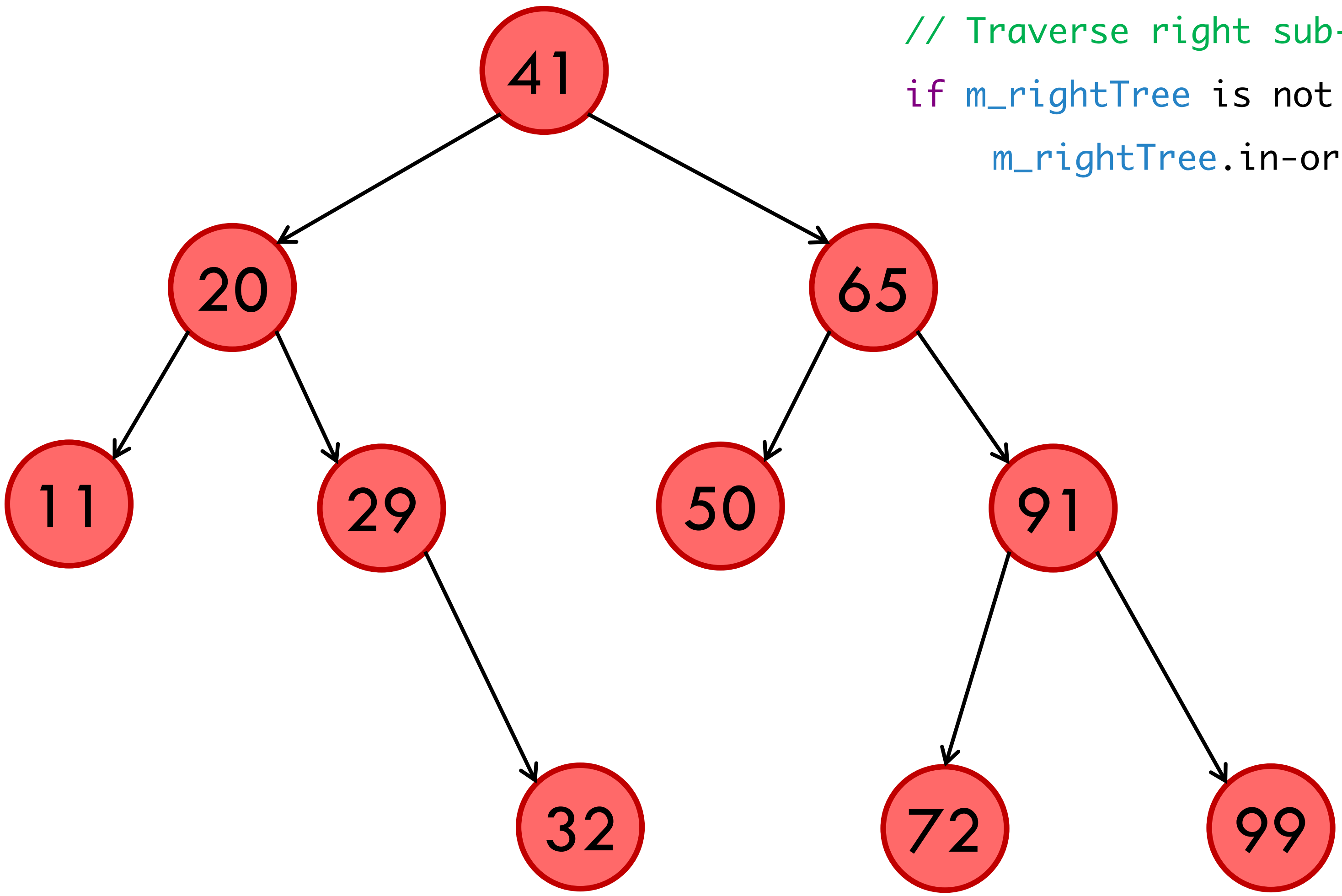
INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



INORDER EXAMPLE:

```
function in-order-traversal()
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.in-order-traversal()
    visit(this)
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.in-order-traversal()
```



INORDER: JAVA CODE

```
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();
}
```

TREE TRAVERSALS

Pre-Order	In-Order	Post-order
SELF LEFT-SUBTREE RIGHT-SUBTREE	LEFT-SUBTREE SELF RIGHT-SUBTREE	LEFT-SUBTREE RIGHT-SUBTREE SELF

NOT SO BASIC BST OPERATIONS

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

`inorder(Node o)`: returns nodes of the tree rooted at o in order

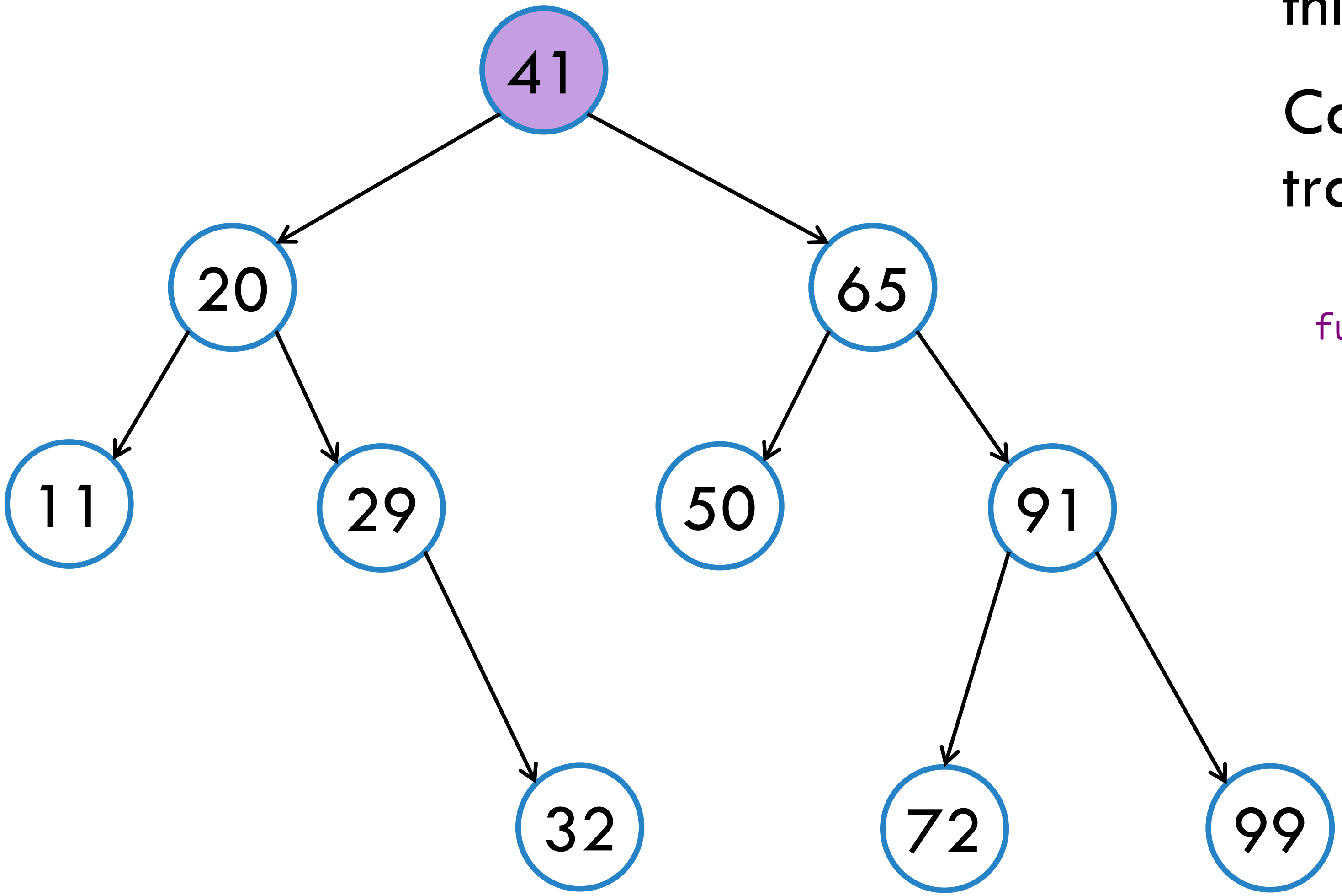
`copyTree(Node o)`: returns a copy of the tree rooted at o

`deleteTree(Node o)`: deletes the tree rooted at o

TREE TRAVERSALS

Pre-Order	In-Order	Post-order
SELF LEFT-SUBTREE RIGHT-SUBTREE	LEFT-SUBTREE SELF RIGHT-SUBTREE	LEFT-SUBTREE RIGHT-SUBTREE SELF

DEEP-COPYING A TREE

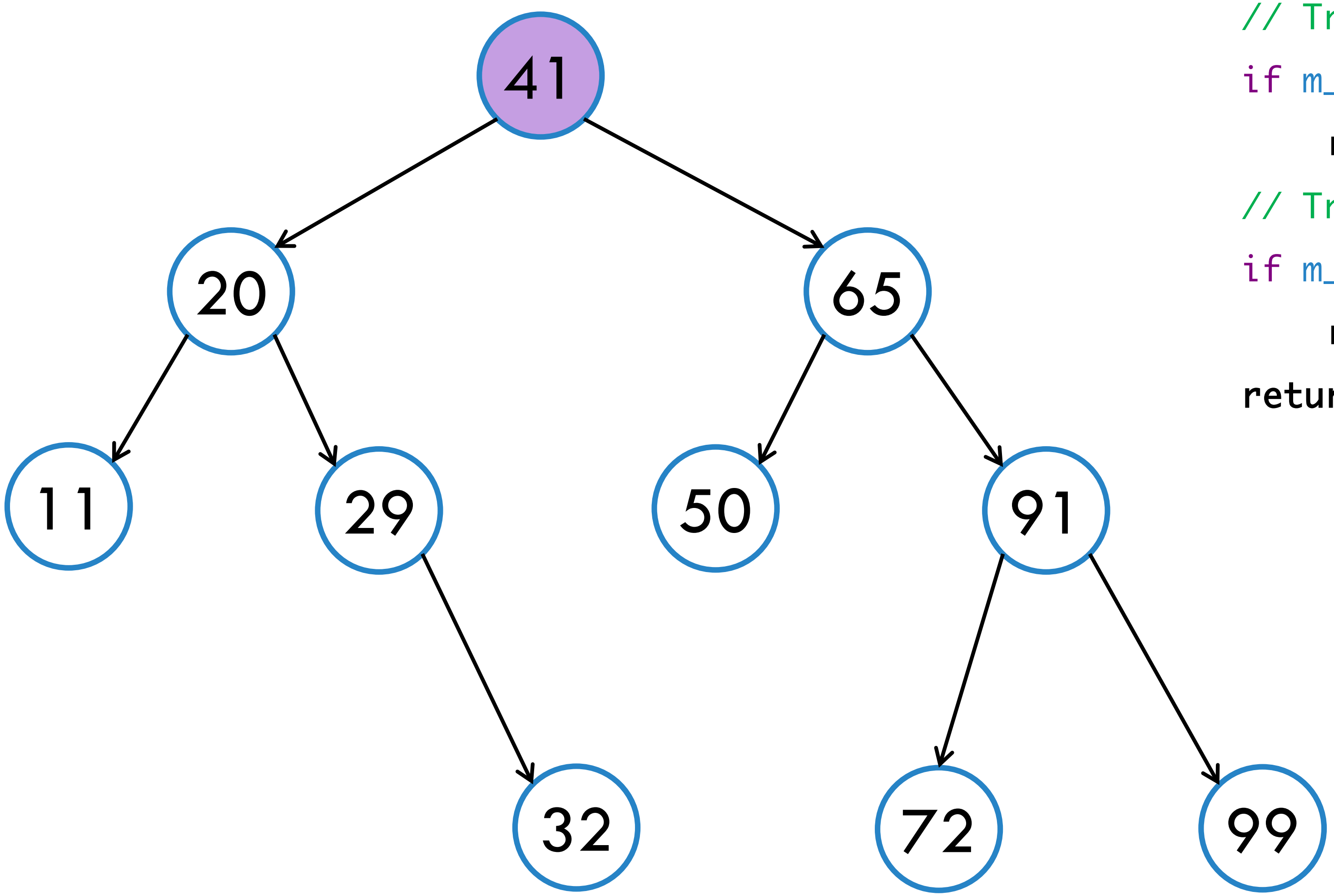


Want to make a copy of this tree.

Can do it via pre-order traversal.

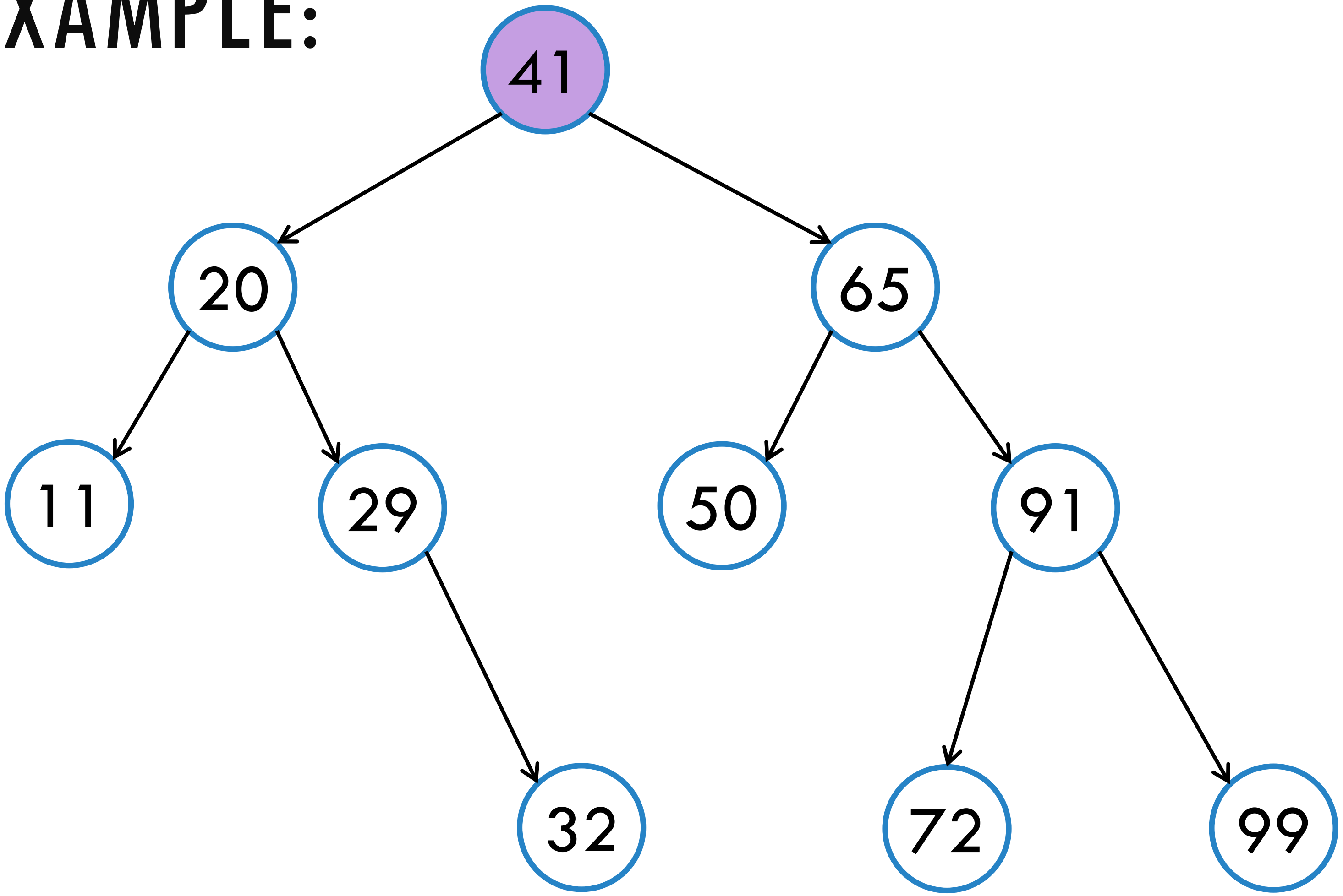
```
function pre-order-traversal()
    visit(this)
    // Traverse left sub-tree
    if m_leftTree is not null
        m_leftTree.pre-order-traversal()
    // Traverse right sub-tree
    if m_rightTree is not null then
        m_rightTree.pre-order-traversal()
```

DEEP-COPYING A TREE



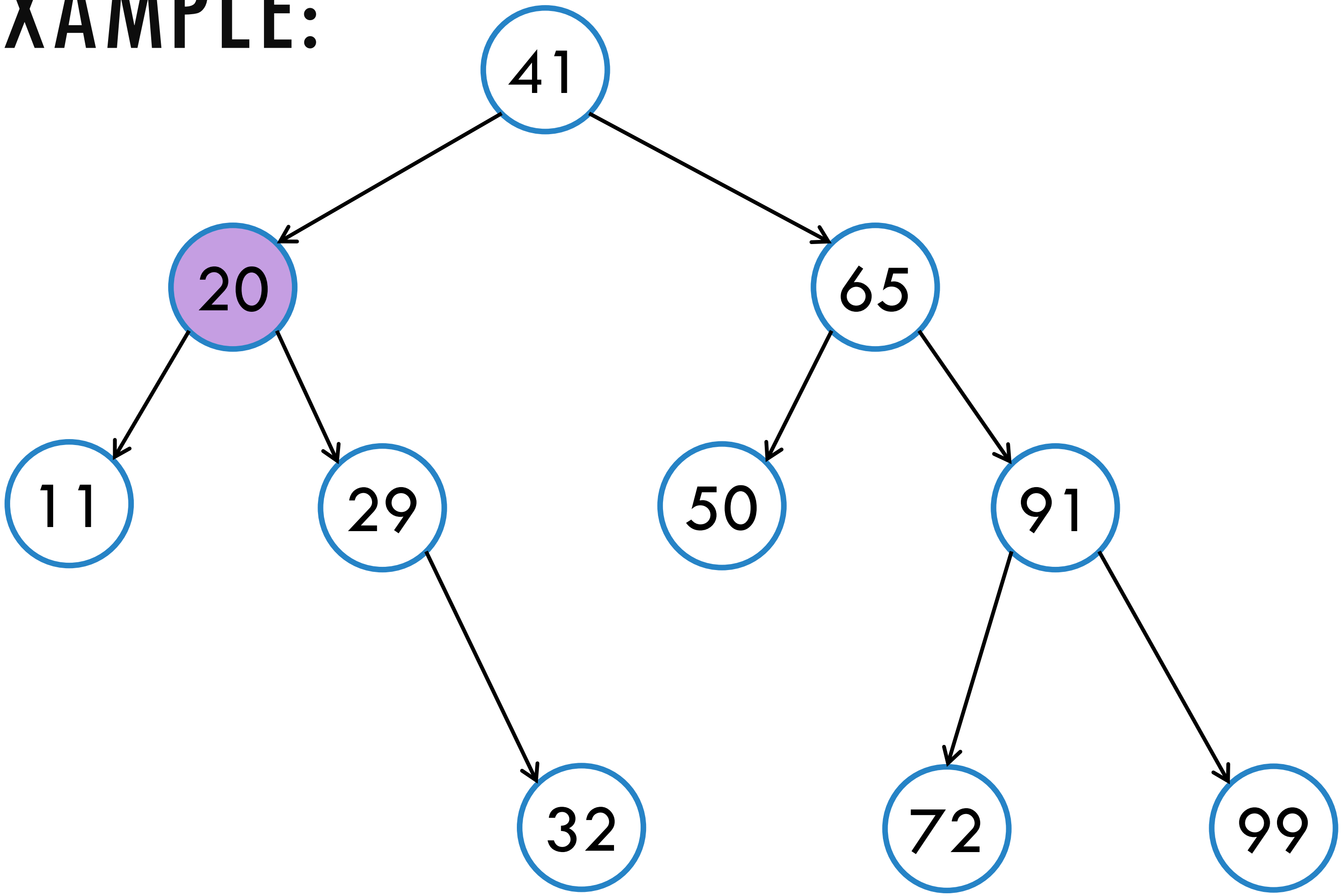

```
function deepCopy()
    n = new Node(key, value)
    // Traverse left sub-tree
    if m_leftTree is not null
        n.m_leftTree = m_leftTree.deepCopy()
    // Traverse right sub-tree
    if m_rightTree is not null
        n.m_rightTree = m_rightTree.deepCopy()
    return n
```

PREORDER EXAMPLE:



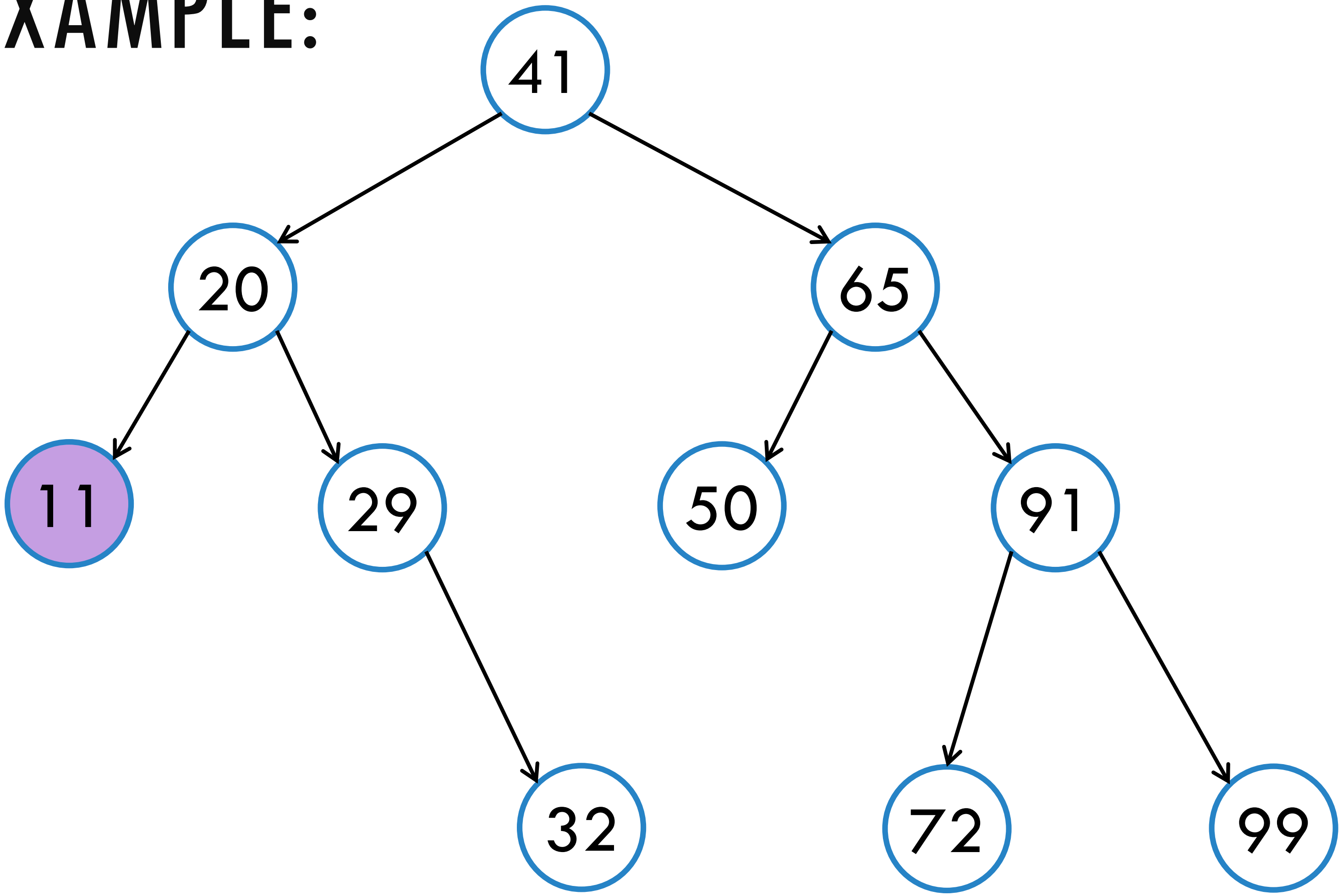
41

PREORDER EXAMPLE:



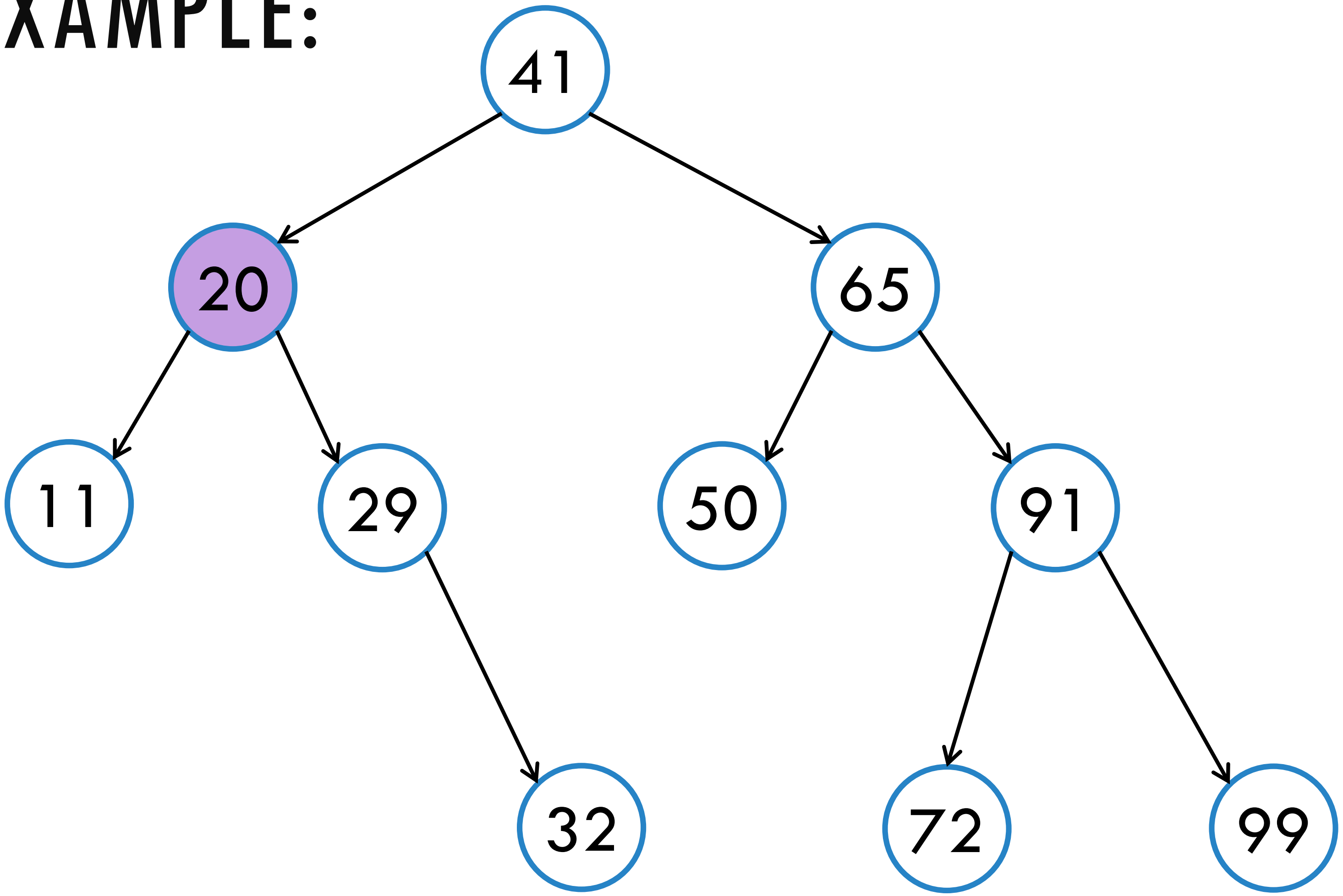
41 20

PREORDER EXAMPLE:



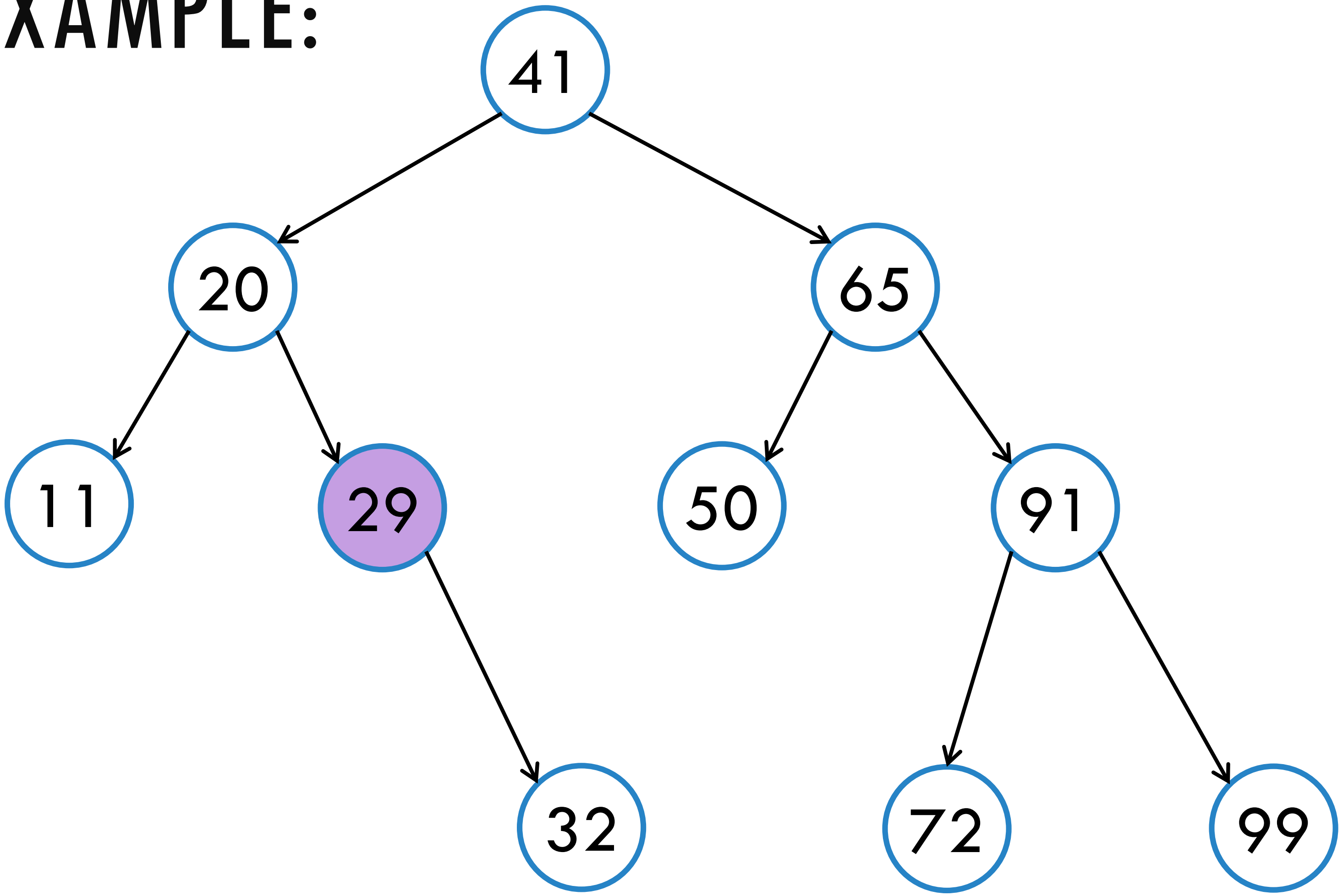
41 20 11

PREORDER EXAMPLE:



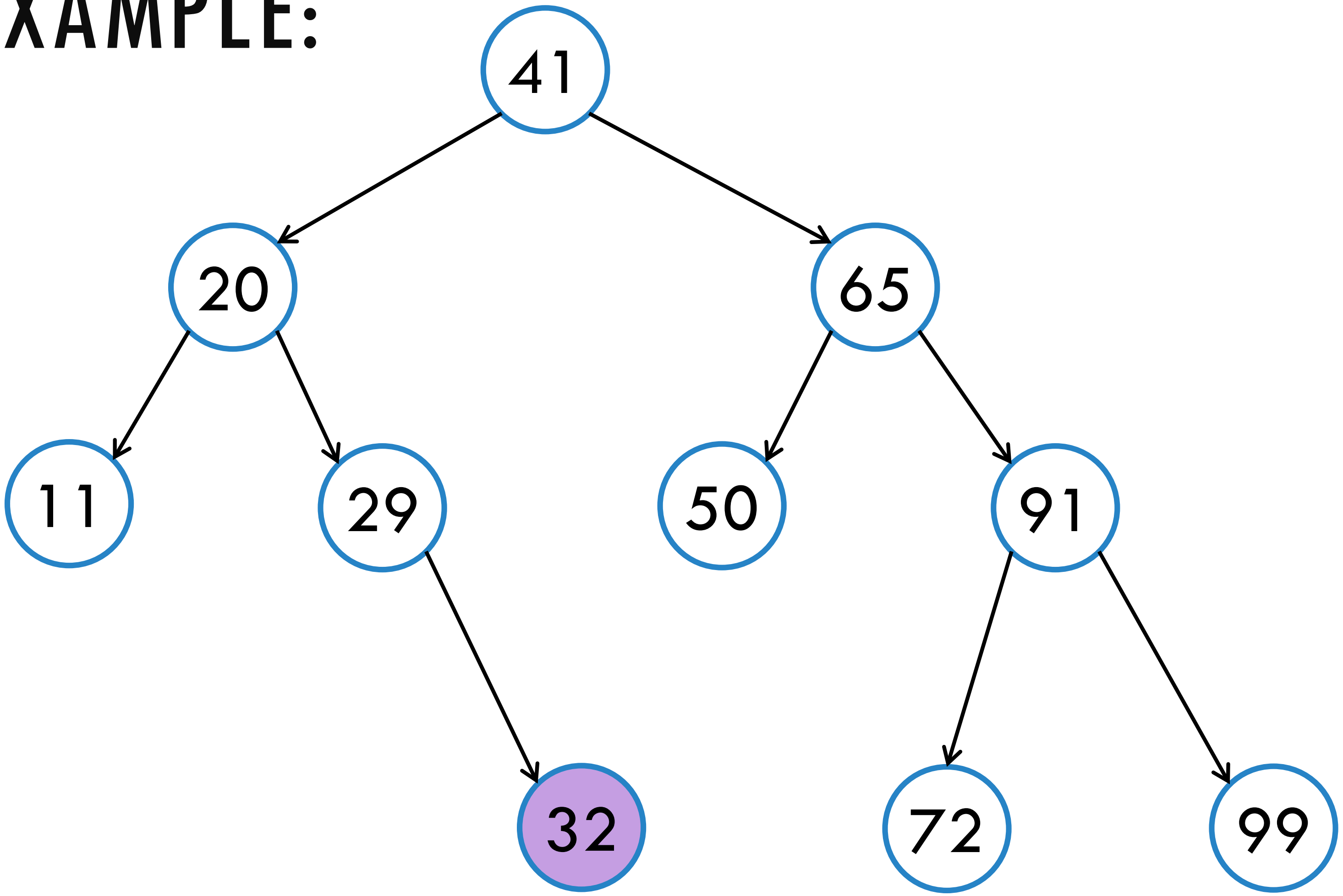
41 20 11

PREORDER EXAMPLE:



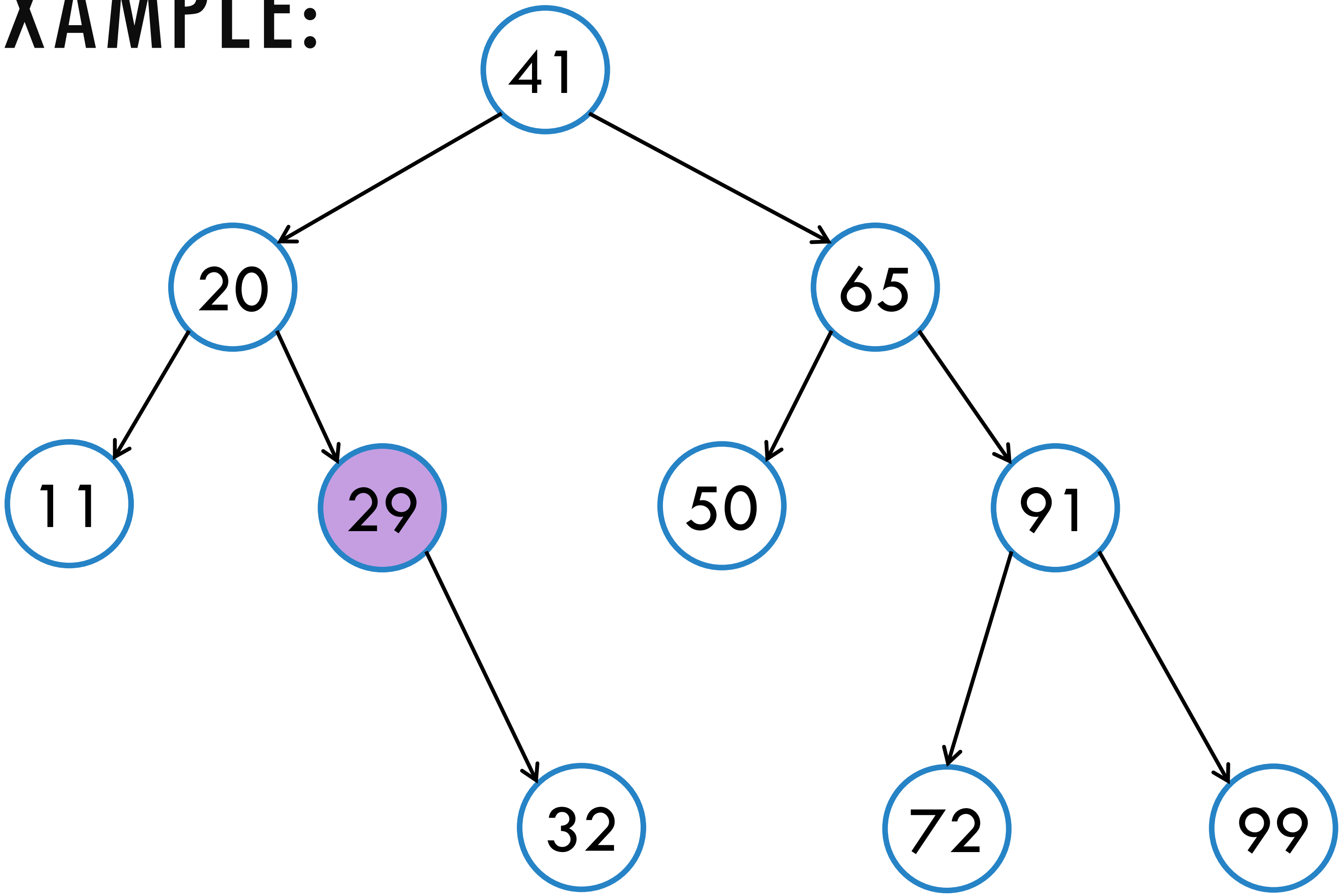
41 20 11 29

PREORDER EXAMPLE:



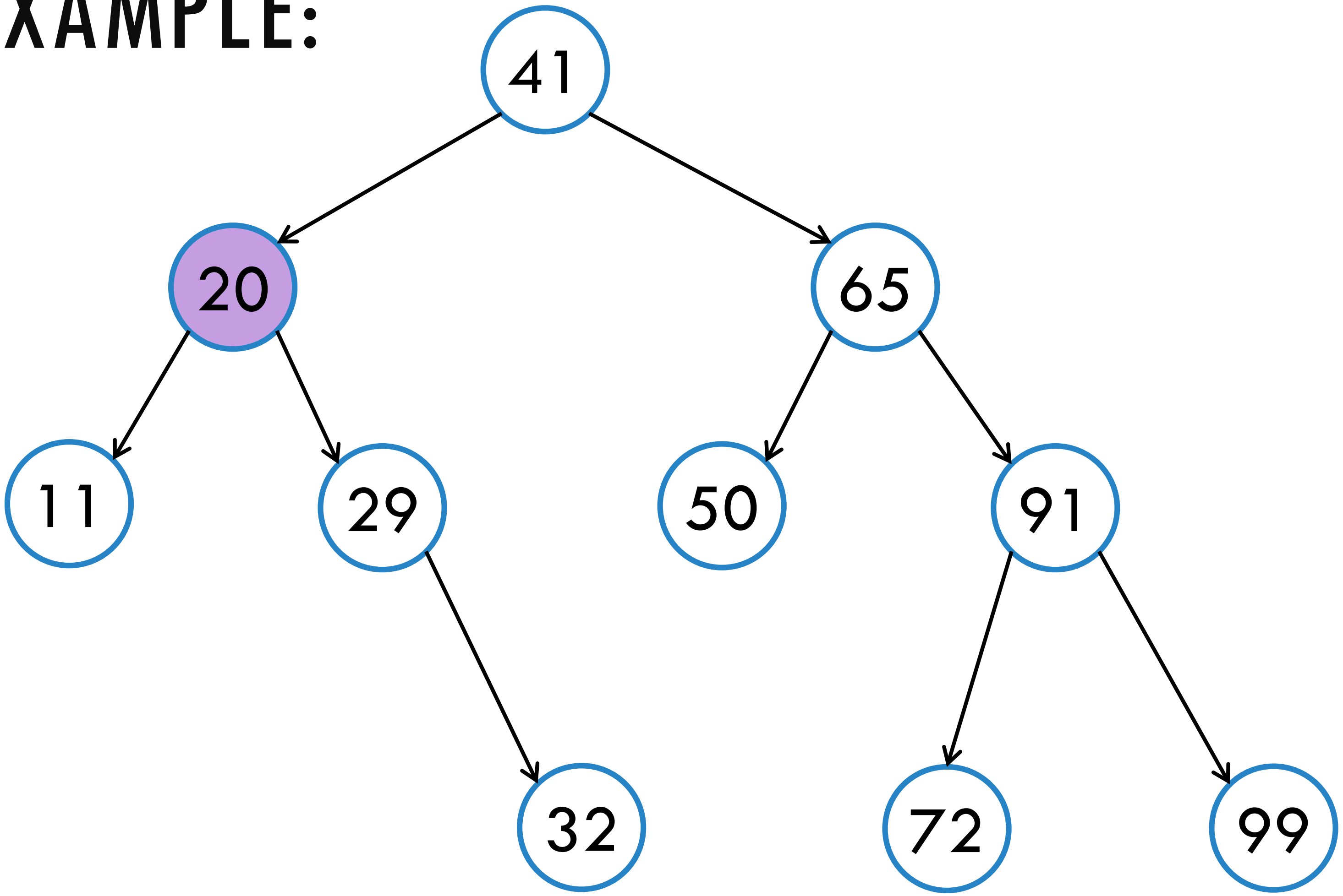
41 20 11 29 32

PREORDER EXAMPLE:



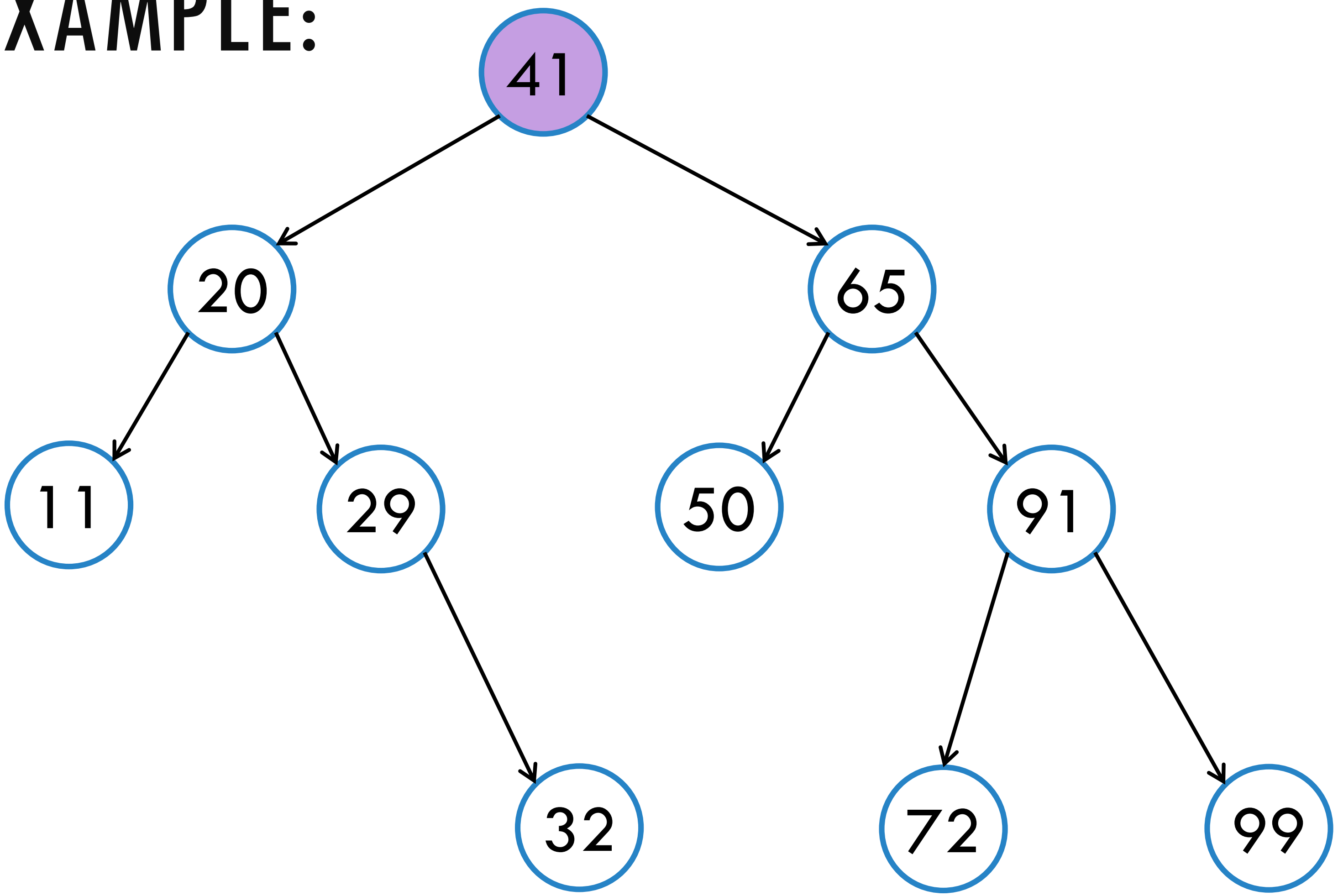
41 20 11 29 32

PREORDER EXAMPLE:



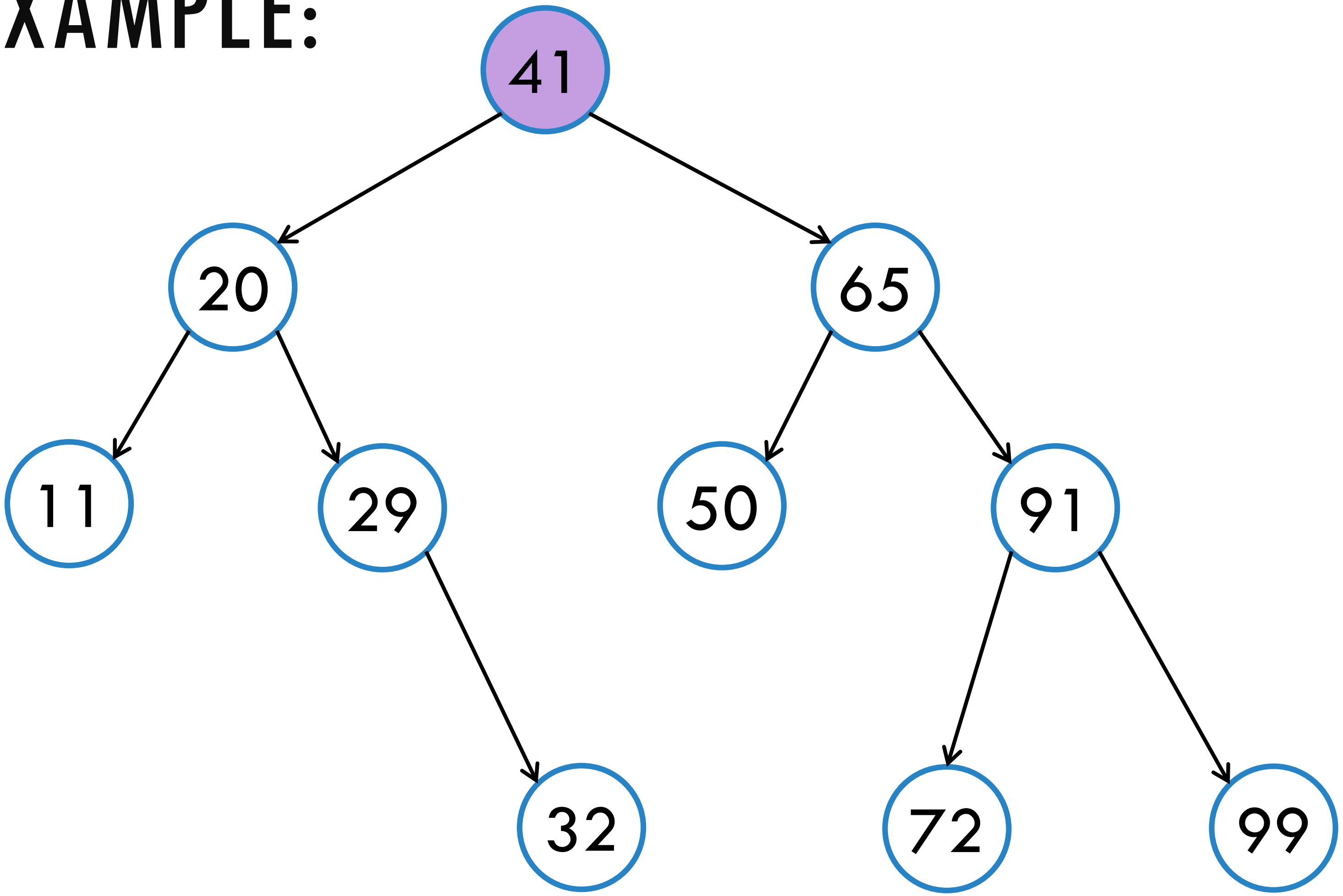
41 20 11 29 32

PREORDER EXAMPLE:



41 20 11 29 32

PREORDER EXAMPLE:



41 20 11 29 32 65 50 91 72 99

NOT SO BASIC BST OPERATIONS

`floor(k):` returns next key $\leq$ k

`ceiling(k):` returns next key $\geq$ k

`inorder(Node o):` returns nodes of the tree rooted at o in order

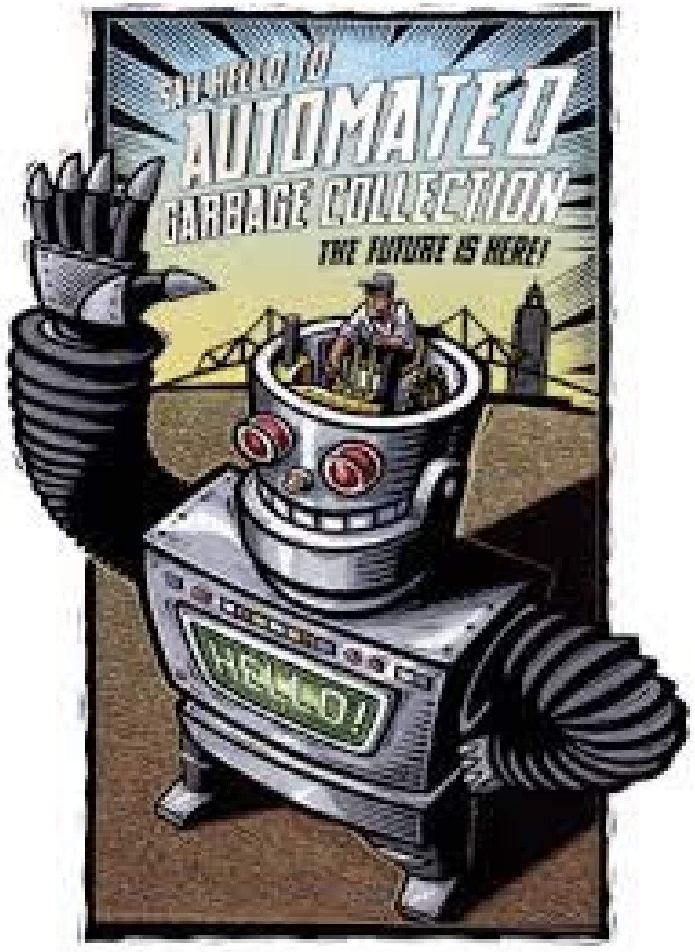
`copyTree(Node o):` returns a copy of the tree rooted at o

`deleteTree(Node o):` deletes the tree rooted at o

TODAY WITH JAVA

To Delete a Tree, you can just do this:

```java
root = null;
```



IN THE OLD DAYS... OR WITH C/C++

with no automatic garbage collection.

We have to delete node by node via some traversal

TREE TRAVERSALS

Pre-Order	In-Order	Post-order
SELF LEFT-SUBTREE RIGHT-SUBTREE	LEFT-SUBTREE SELF RIGHT-SUBTREE	LEFT-SUBTREE RIGHT-SUBTREE SELF

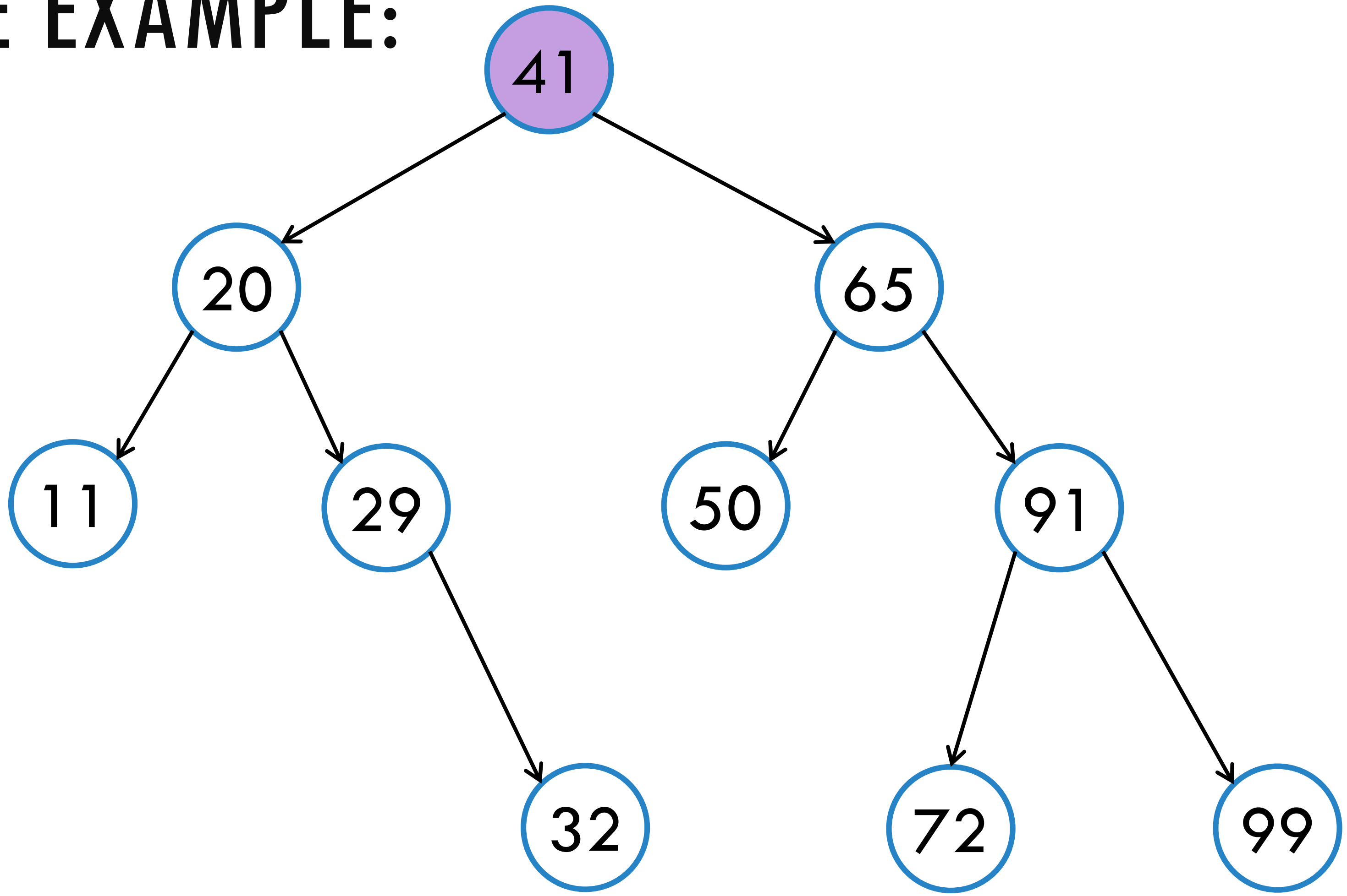
IN THE OLD DAYS… OR WITH C/C++

with no automatic garbage collection.

We have to delete node by node via **post-order** traversal

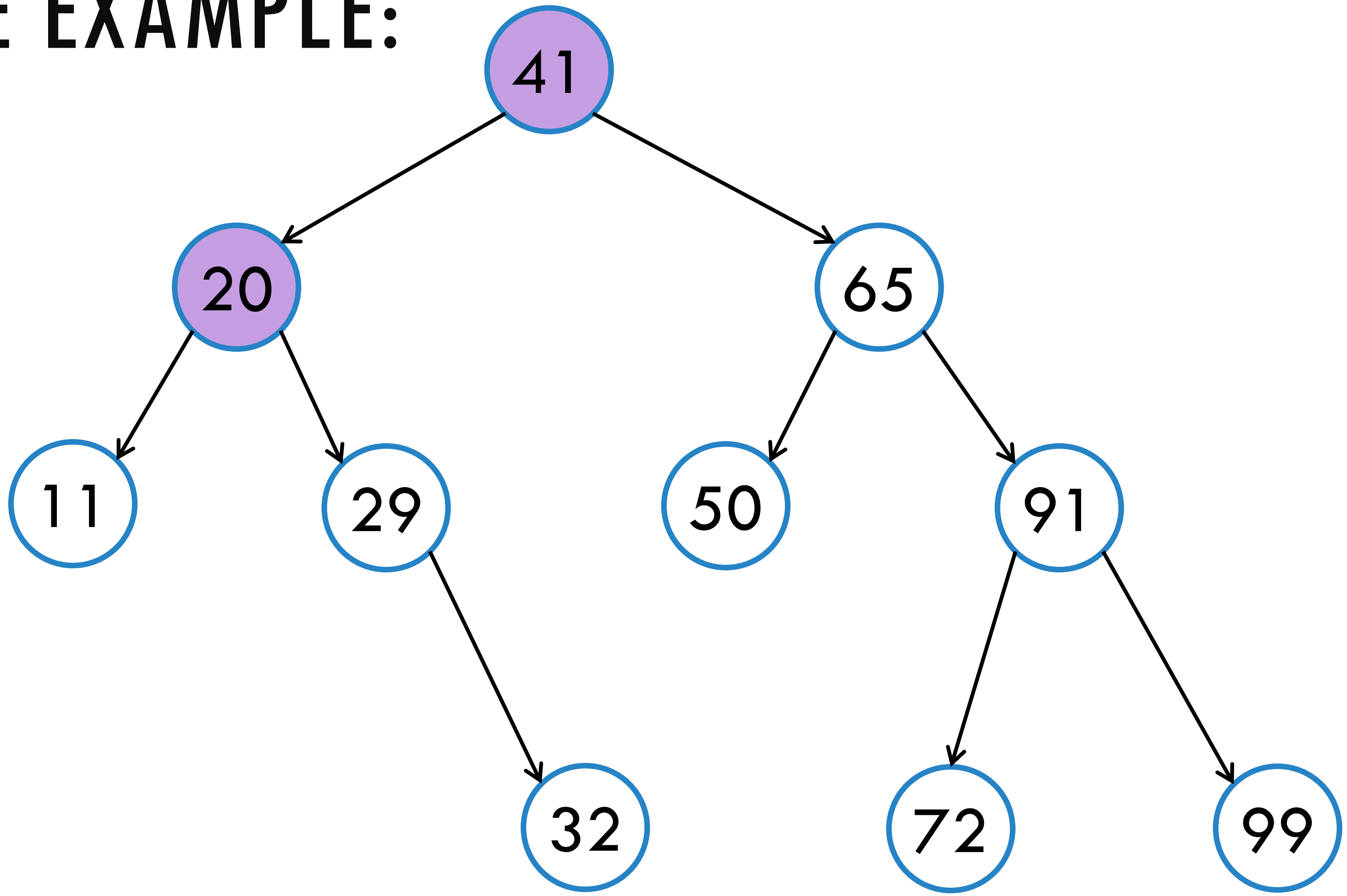
```
function deleteTree(Node n)
    // Traverse left sub-tree
    if n.m_leftTree is not null
        deleteTree(n.m_leftTree)
    // Traverse right sub-tree
    if n.m_rightTree is not null then
        deleteTree(n.m_rightTree)
    free(n)
```

DELETE TREE EXAMPLE:

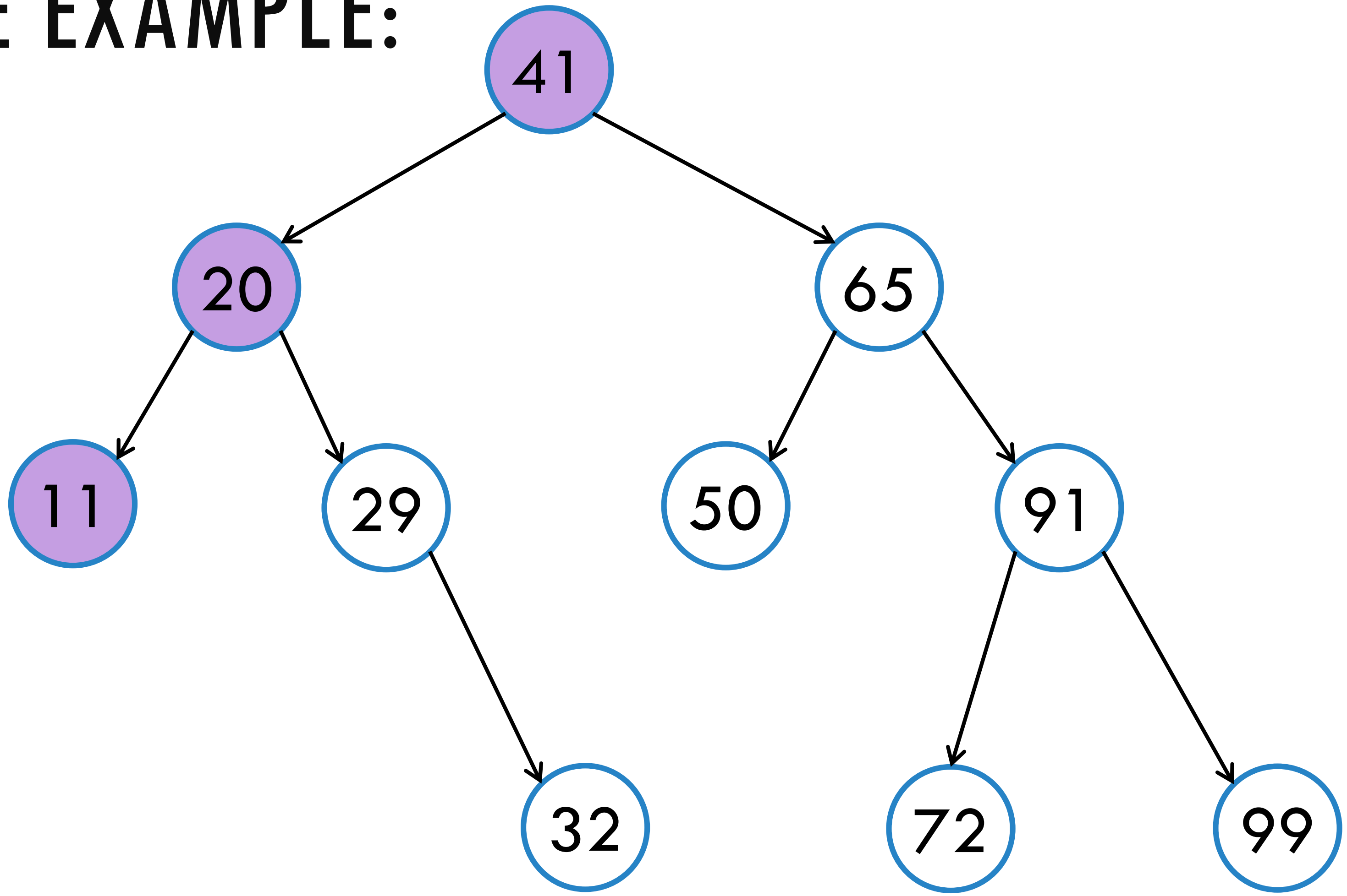


11 32 29 20 50 72 99 91 65 41

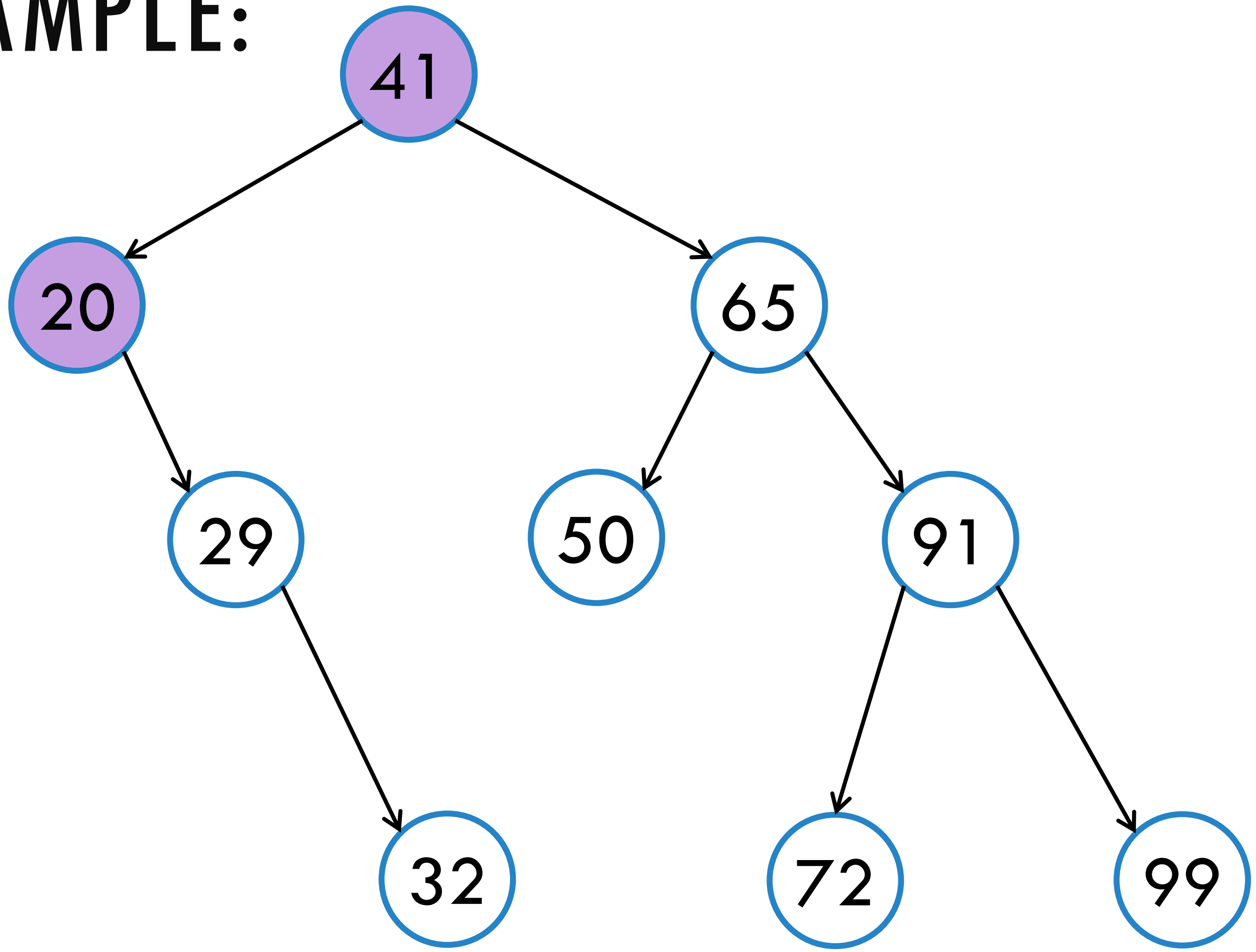
DELETE TREE EXAMPLE:



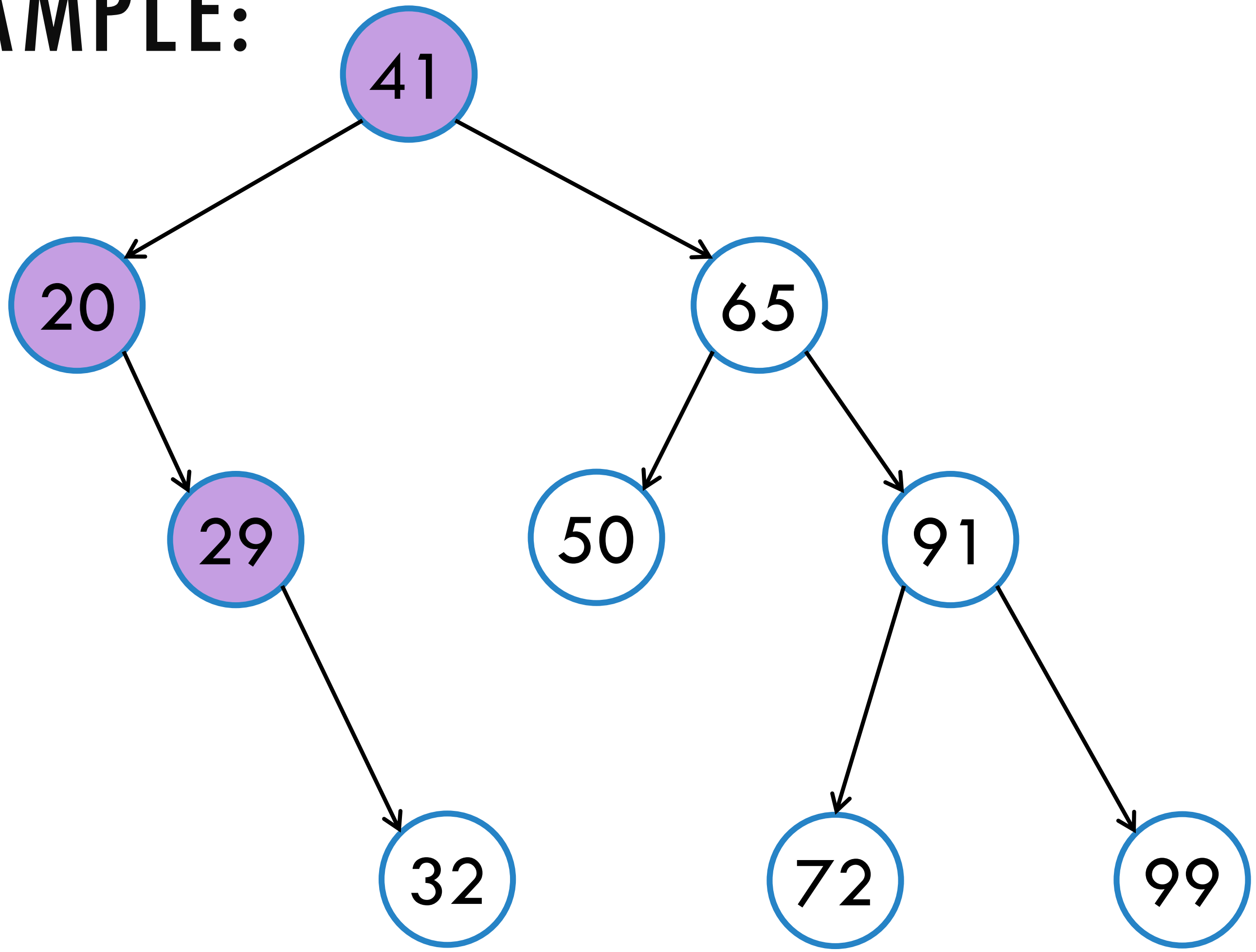
DELETE TREE EXAMPLE:



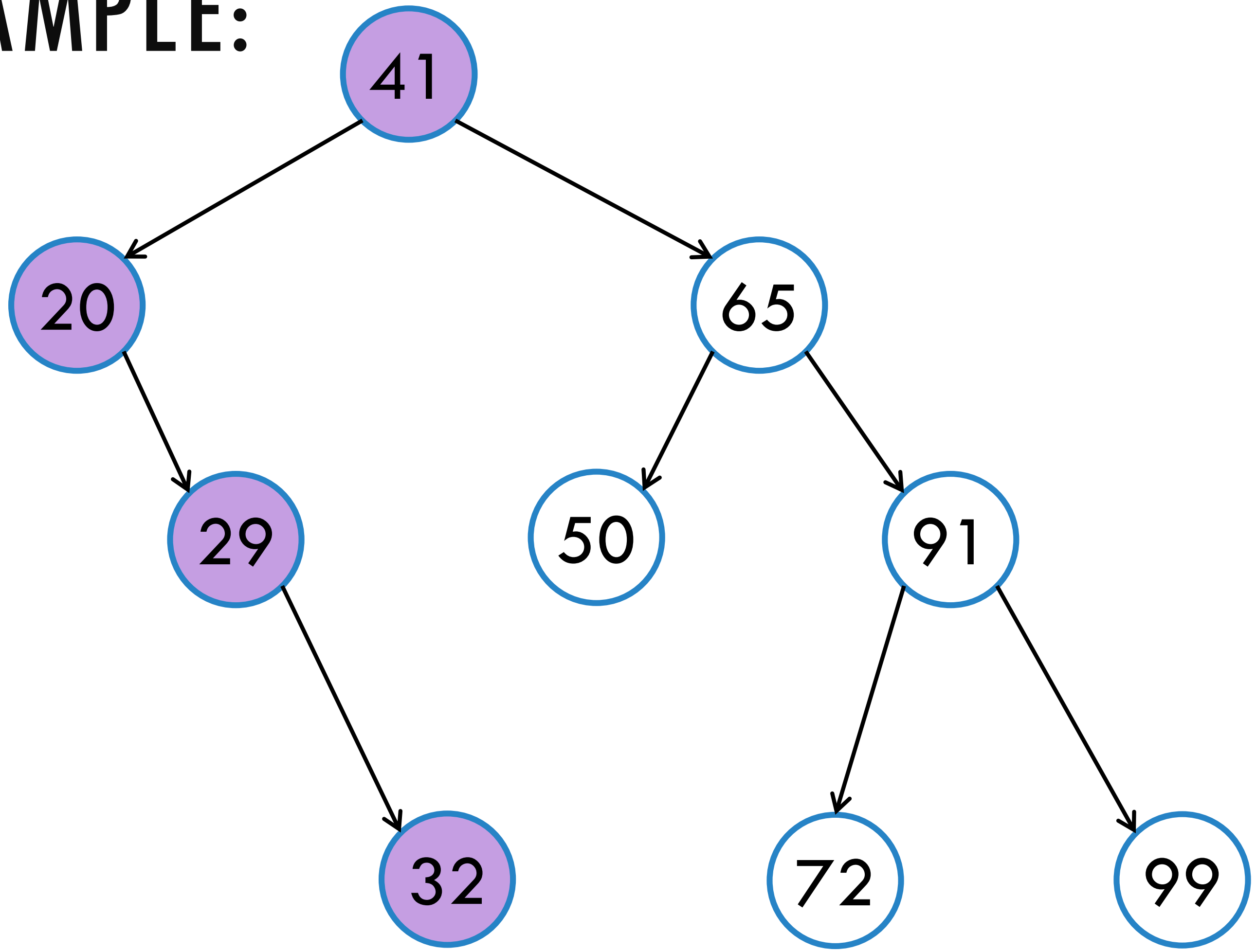
DELETE TREE EXAMPLE:



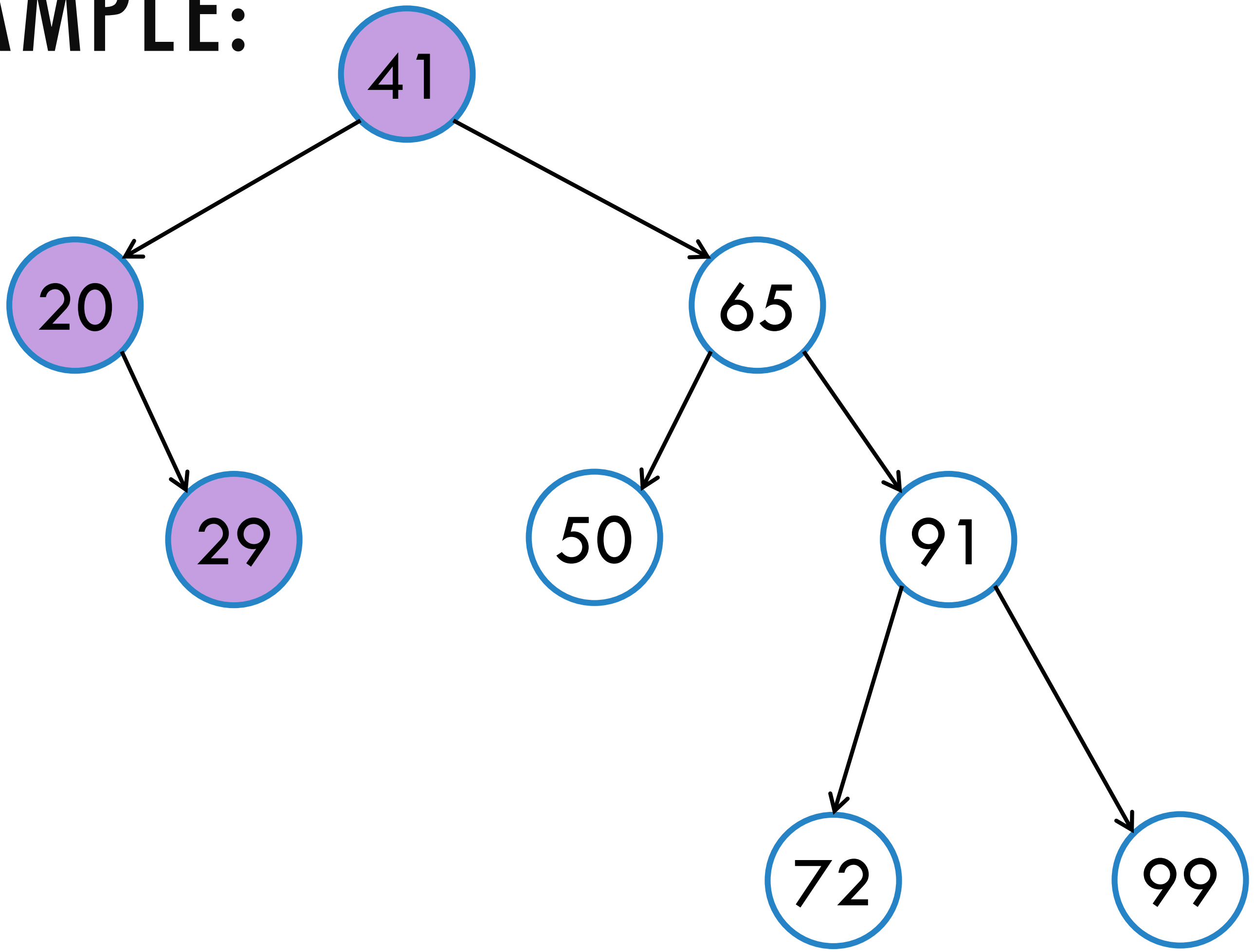
DELETE TREE EXAMPLE:



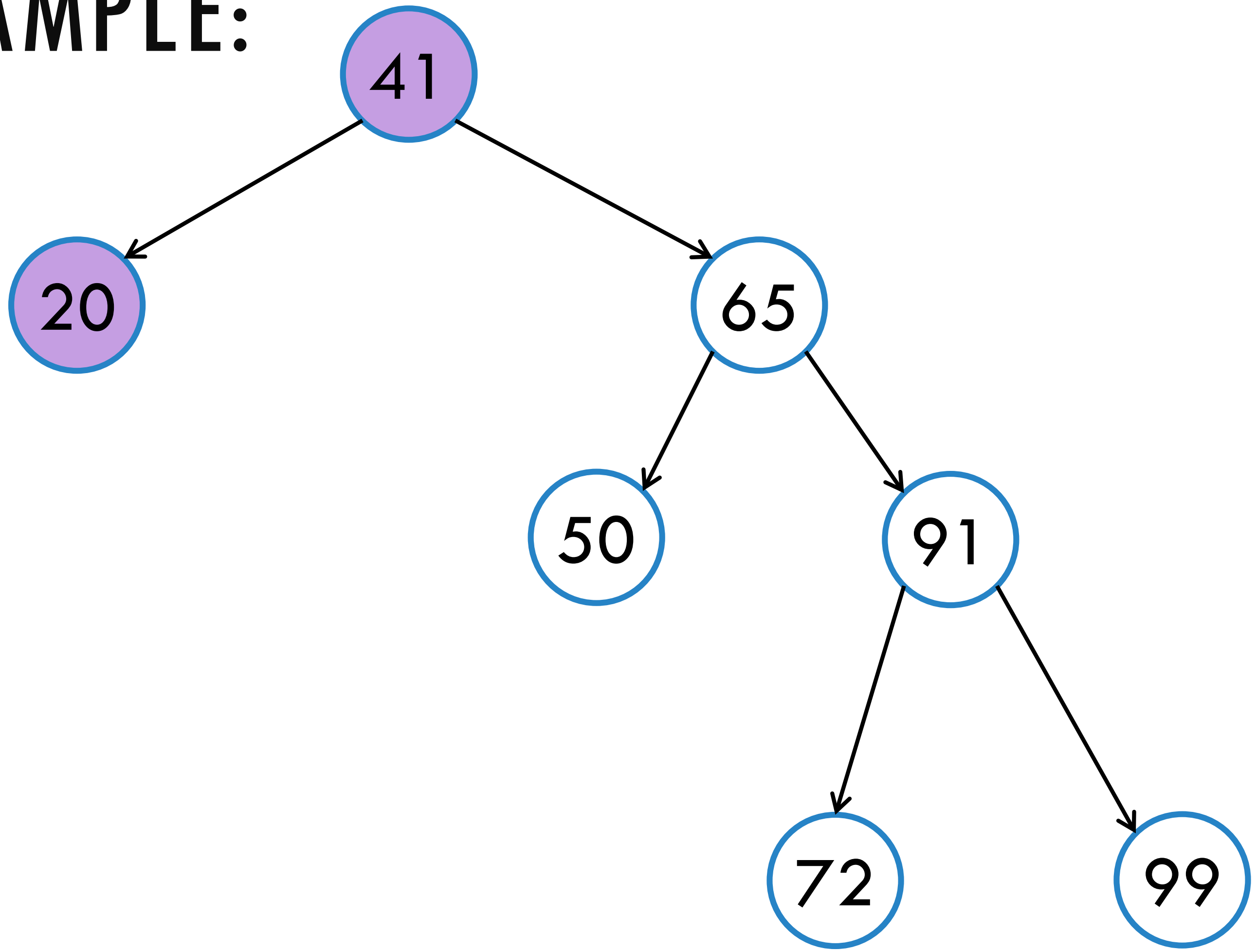
DELETE TREE EXAMPLE:



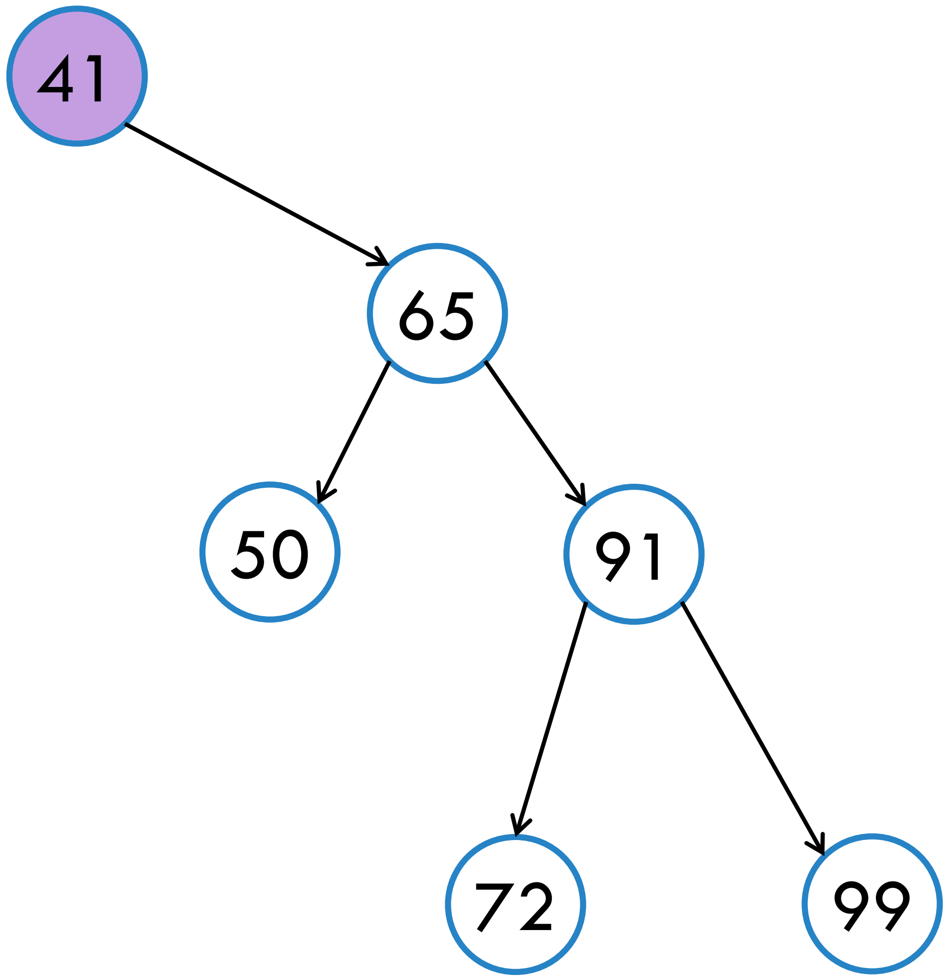
DELETE TREE EXAMPLE:



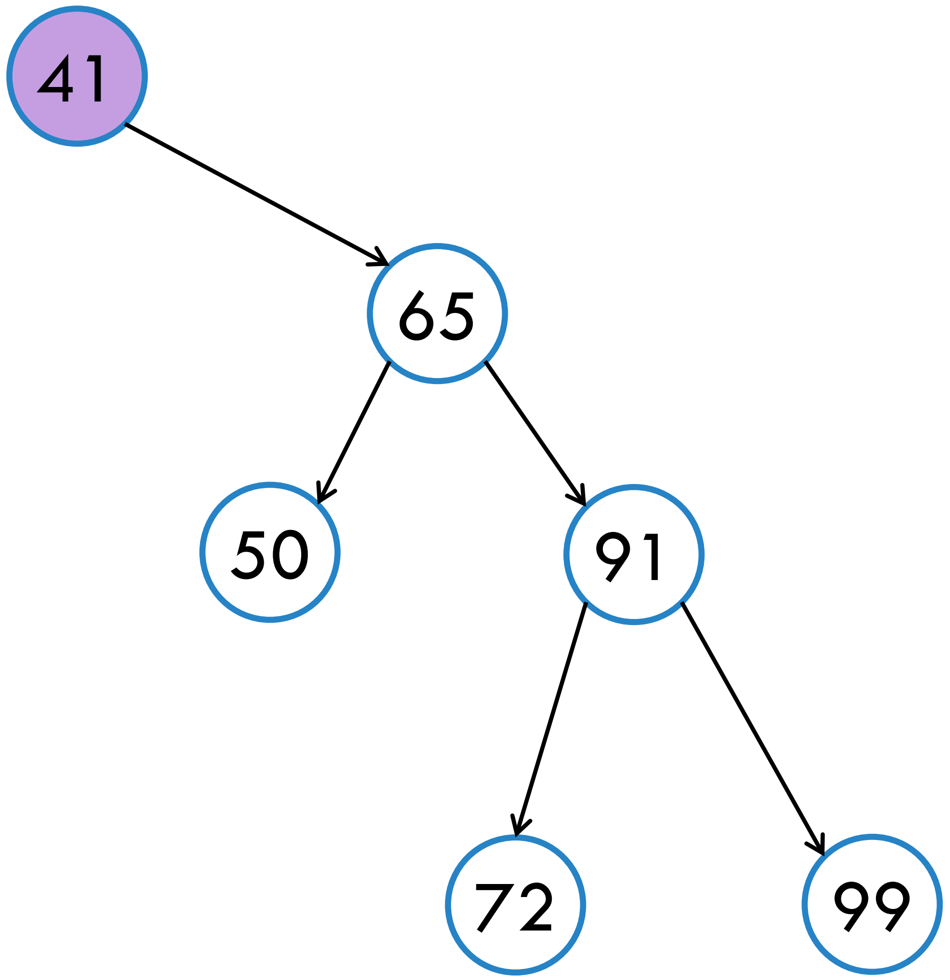
DELETE TREE EXAMPLE:



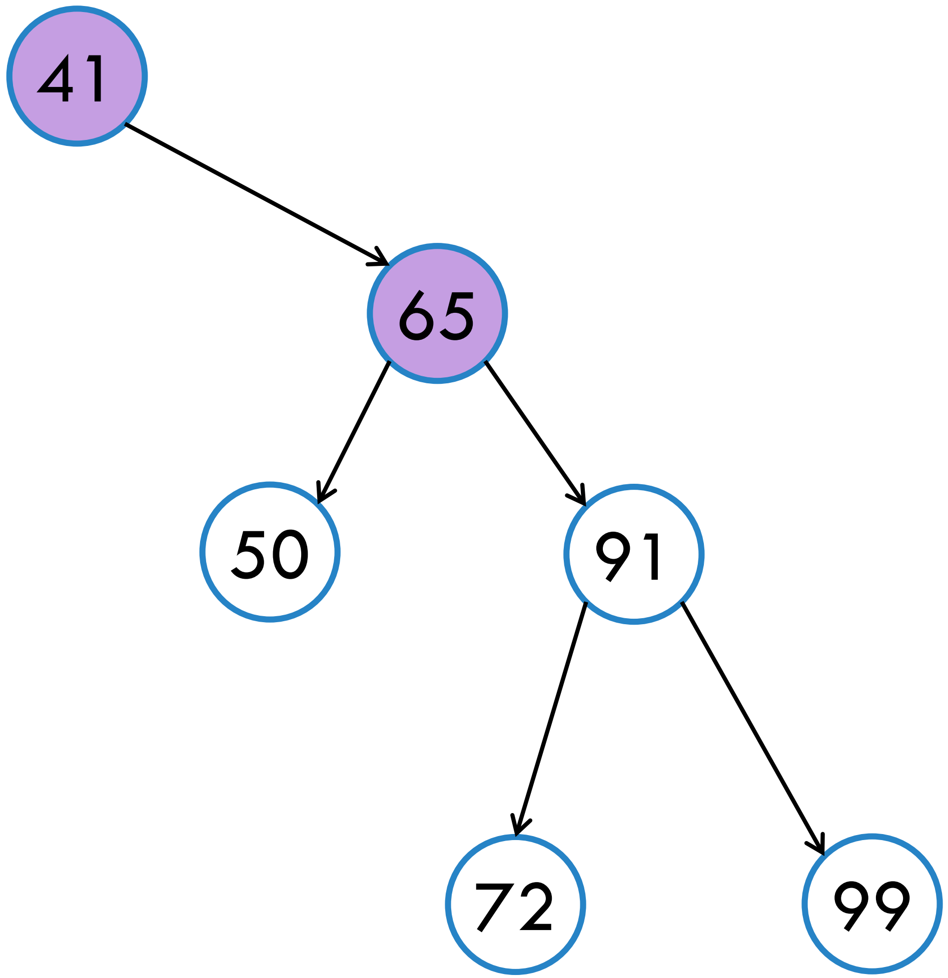
DELETE TREE EXAMPLE:



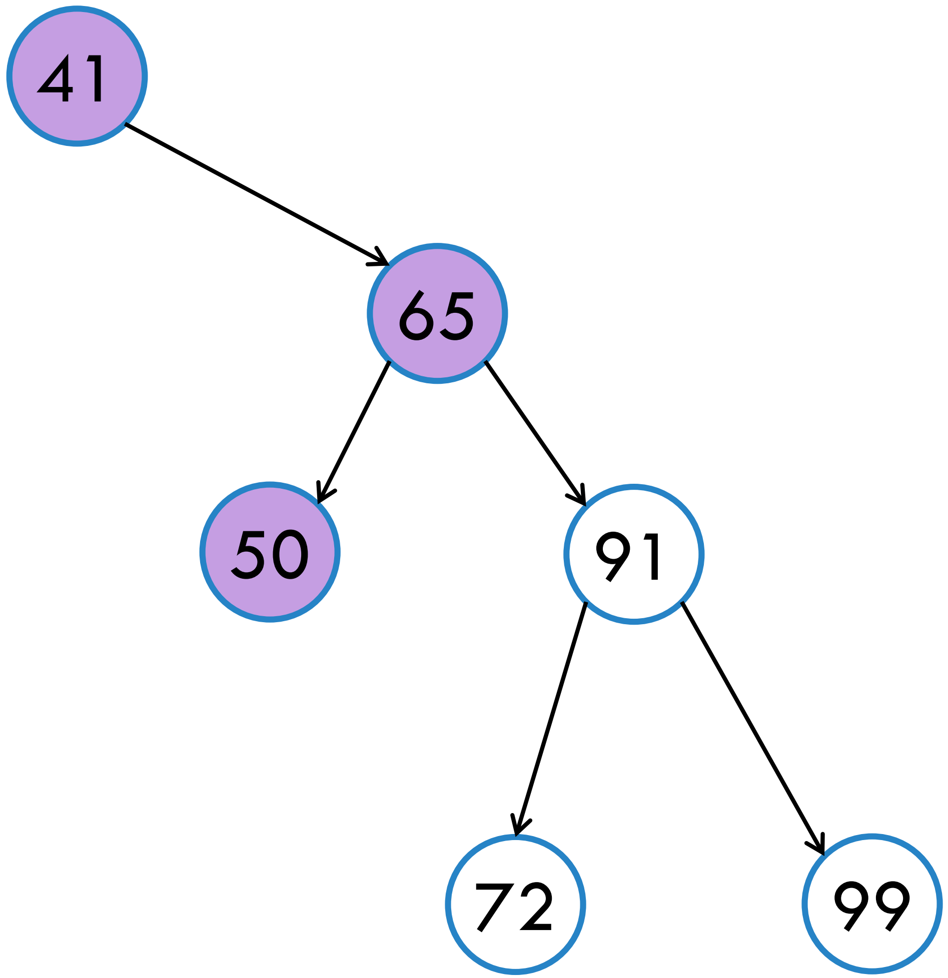
DELETE TREE EXAMPLE:



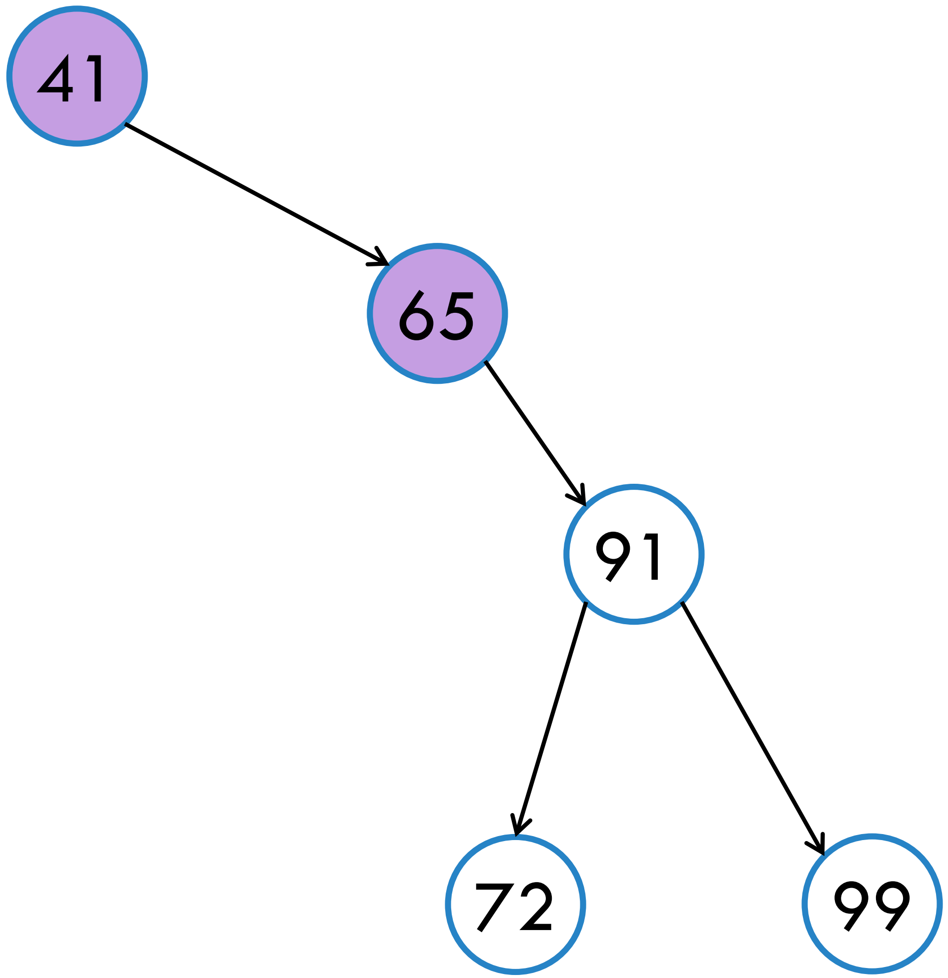
DELETE TREE EXAMPLE:



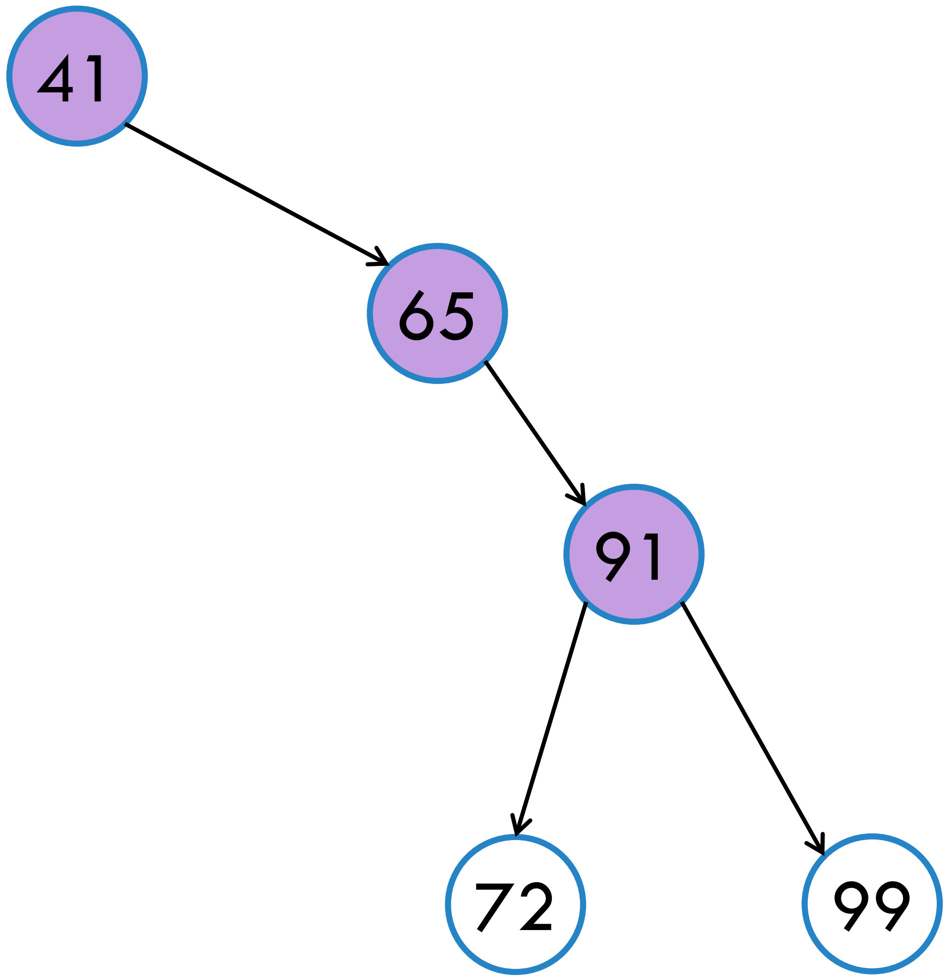
DELETE TREE EXAMPLE:



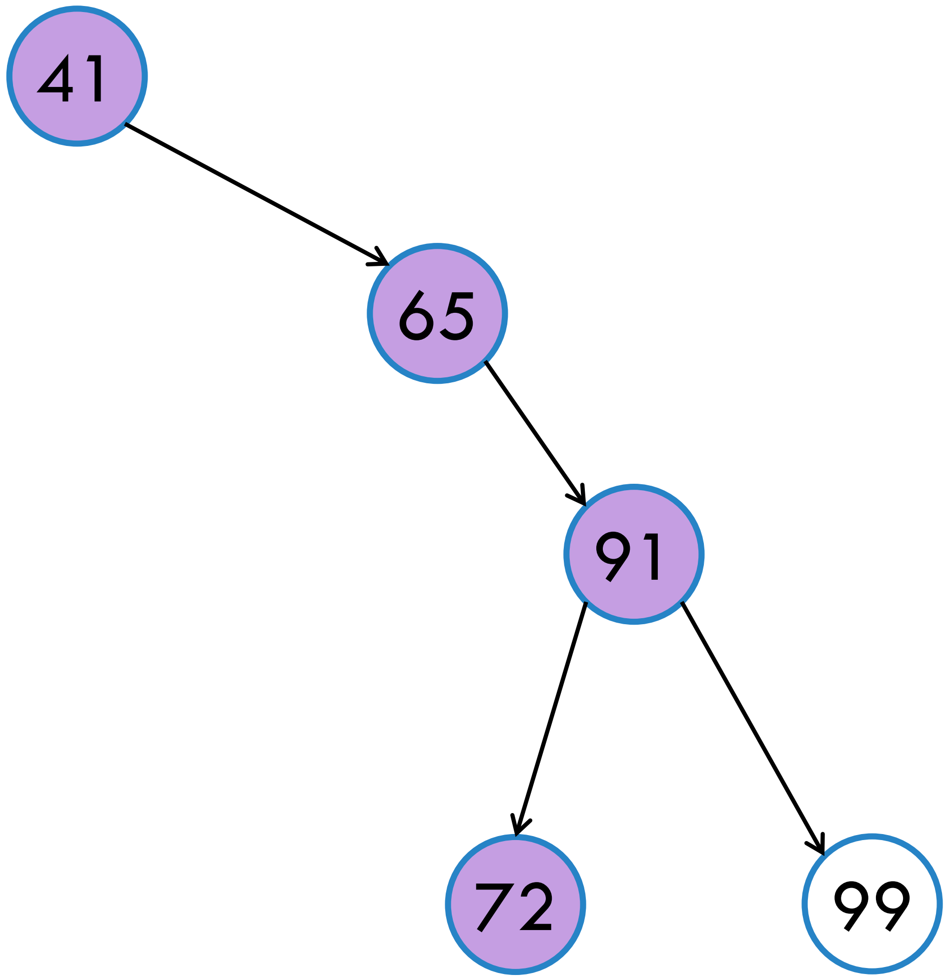
DELETE TREE EXAMPLE:



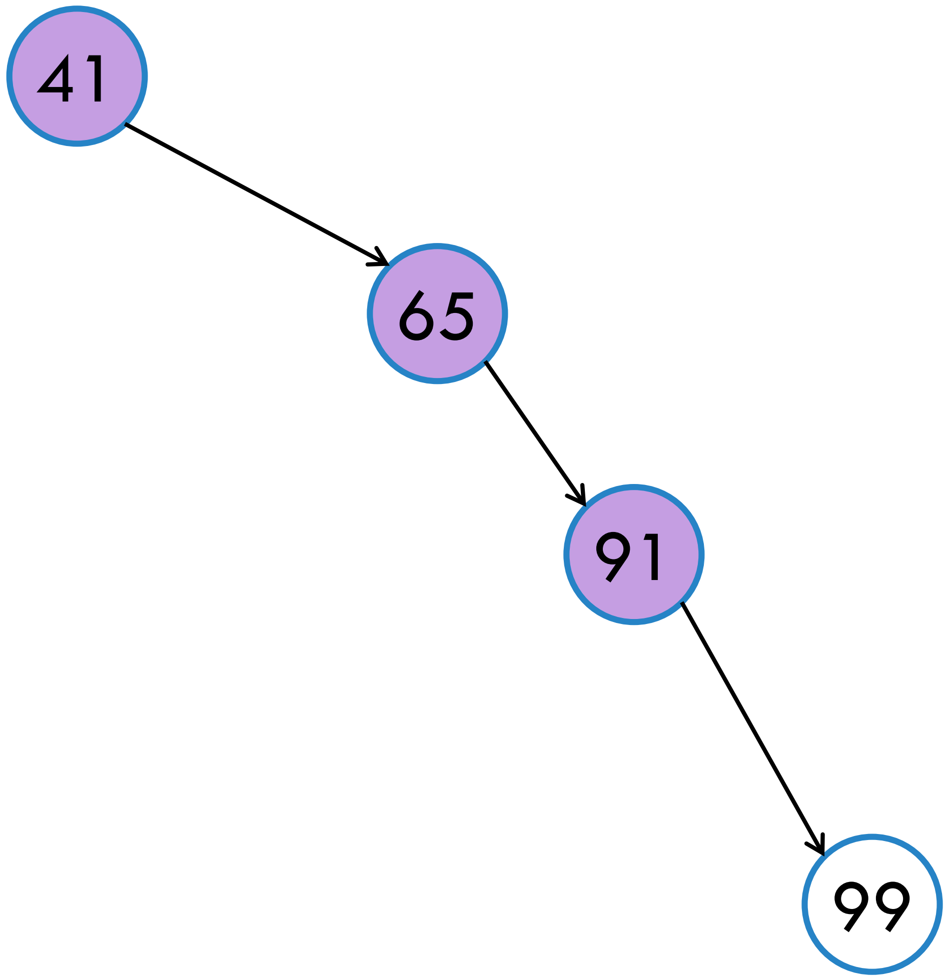
DELETE TREE EXAMPLE:



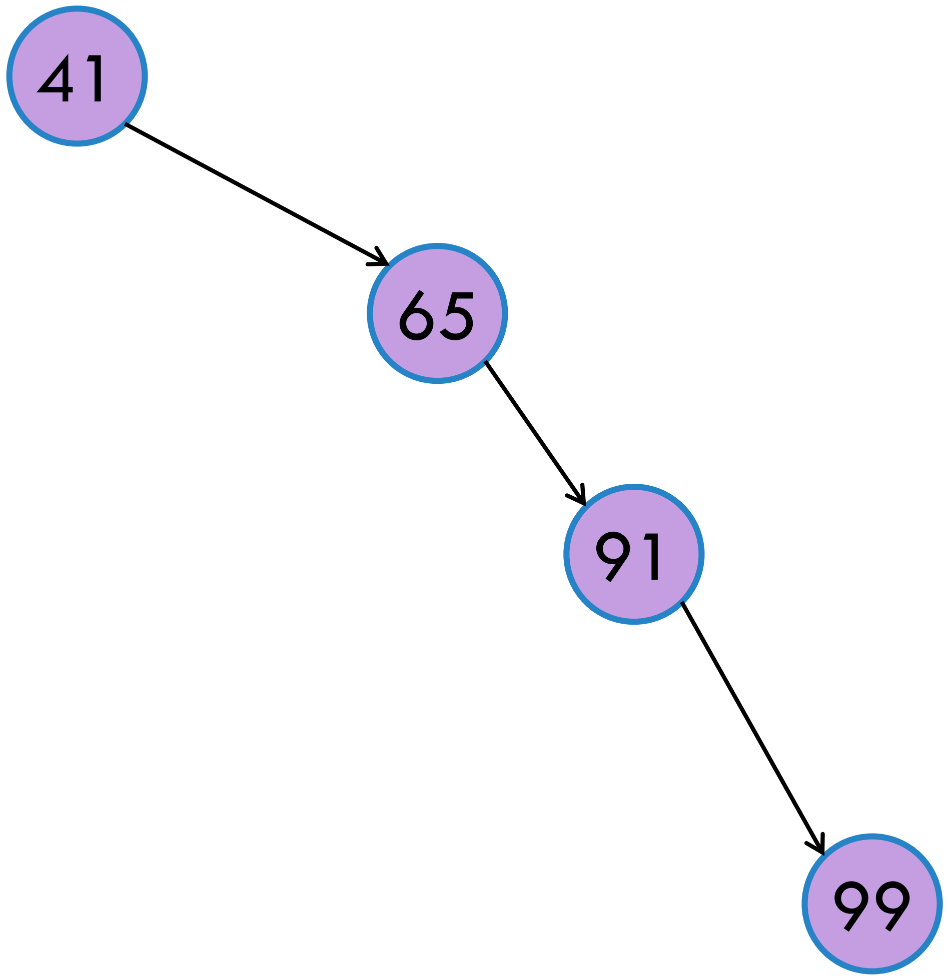
DELETE TREE EXAMPLE:



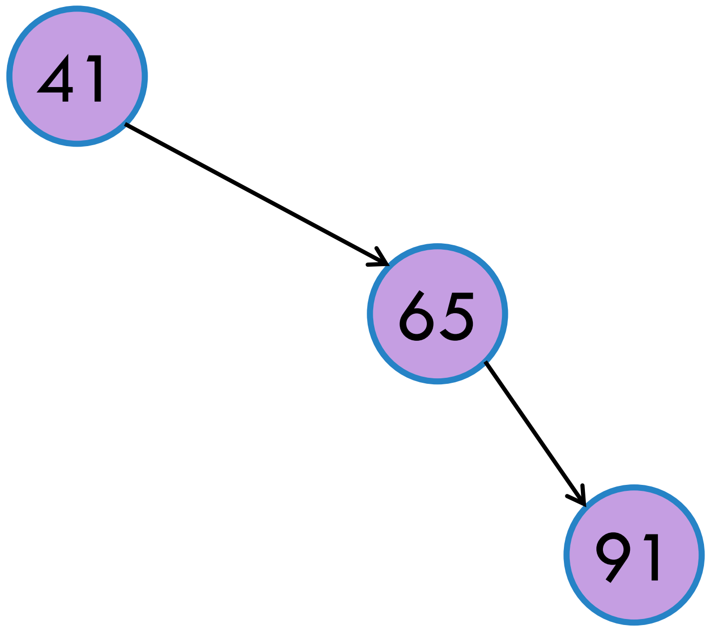
DELETE TREE EXAMPLE:



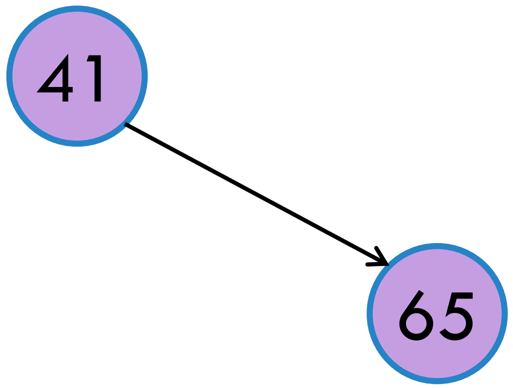
DELETE TREE EXAMPLE:



DELETE TREE EXAMPLE:



DELETE TREE EXAMPLE:



DELETE TREE EXAMPLE:

DELETE TREE EXAMPLE:

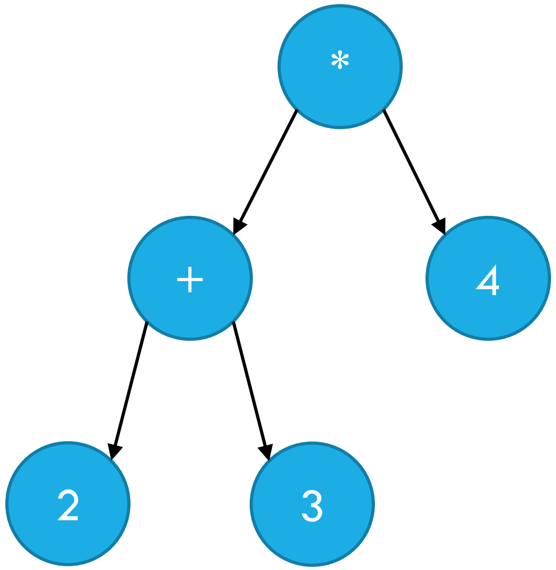
RPN: IN-ORDER, PRE-ORDER, POST-ORDER

Remember Quiz 1 on **Reverse Polish Notation?**

"Normal": $(2+3)*4 \ = \ 20$ **"Infix notation"**

RPN: $2,3,+,4,*$ **"Post-fix notation"**

Which **traversal** do we use to evaluate an expression encoded in a Binary Tree?



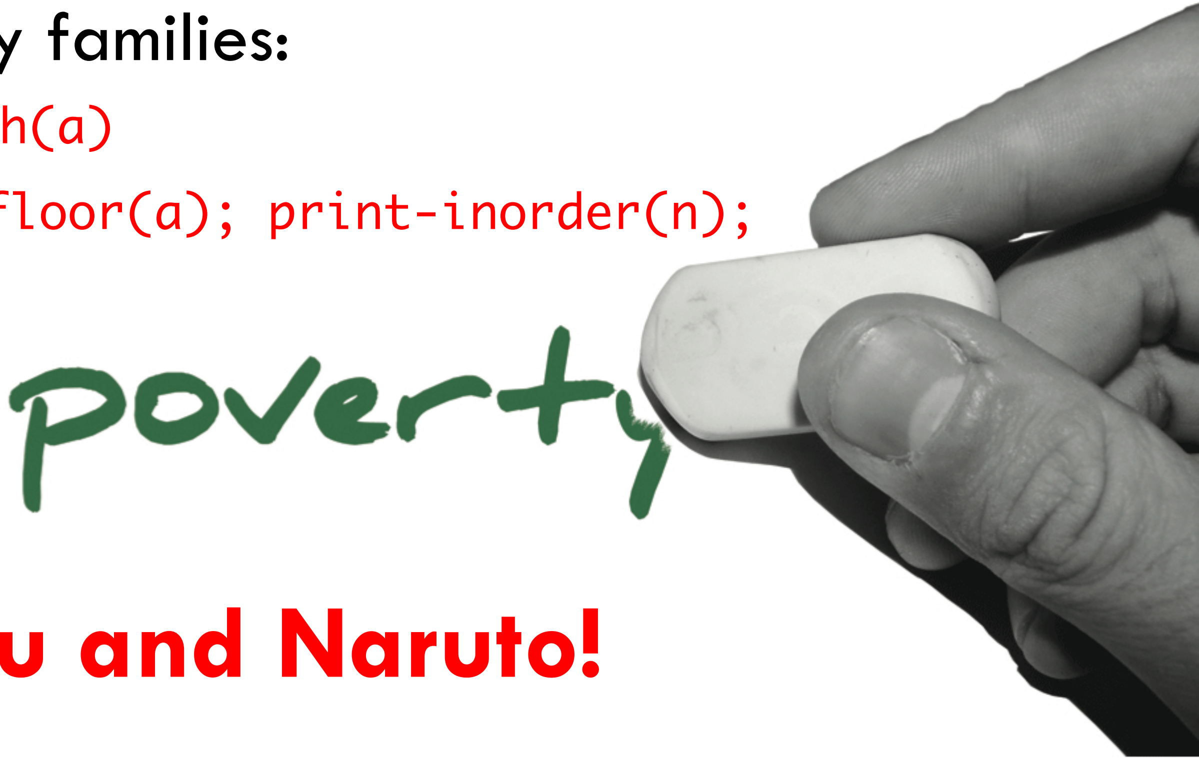
BACK TO OUR PROBLEM

The Stop-Poverty charity calls:

To provide financial aid, Help identify families:

- earning exactly \$a amount `n = search(a)`
- earning less than or equal to \$a `n = floor(a); print-inorder(n);`
- earning more than or equal to \$a:

  `n = ceiling(a); print-inorder(n);`



another win for you and Naruto!

LEARNING OUTCOMES

By the end of this session, students should be able to:

- Describe the **Binary Search Tree (BST)** and its operations
- Analyze **performance of BST operations**
- Relate the **importance of balance in a BST** to enable efficient operations.
- Explain the **pre-order, post-order, and in-order tree traversal** algorithms.

QUESTIONS?