Binary Search Tree Data Structure

261217 Data Structures for Computer Engineers

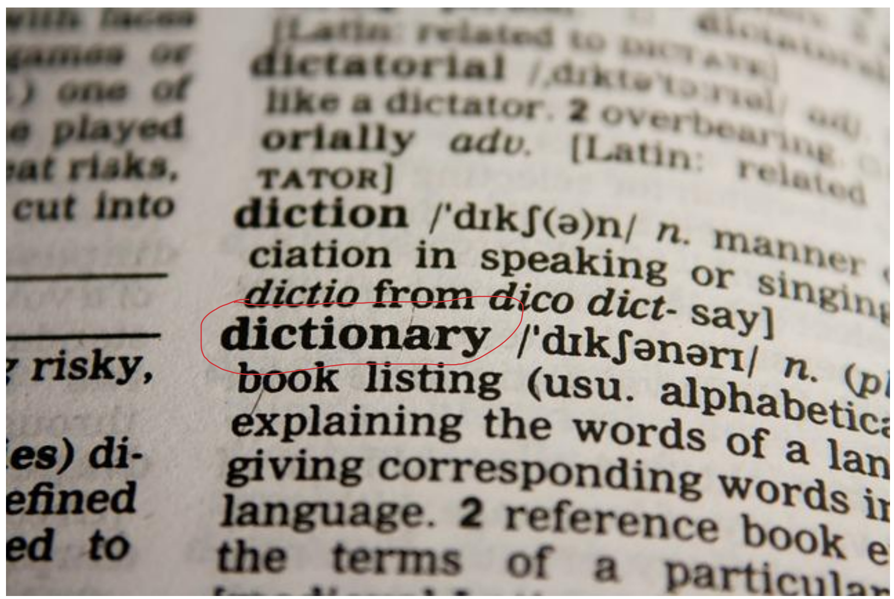
Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Dictionary Search

- Find information of a student given student ID
- Find all words that start with some given string
- For example: “dict”



Data Ranges

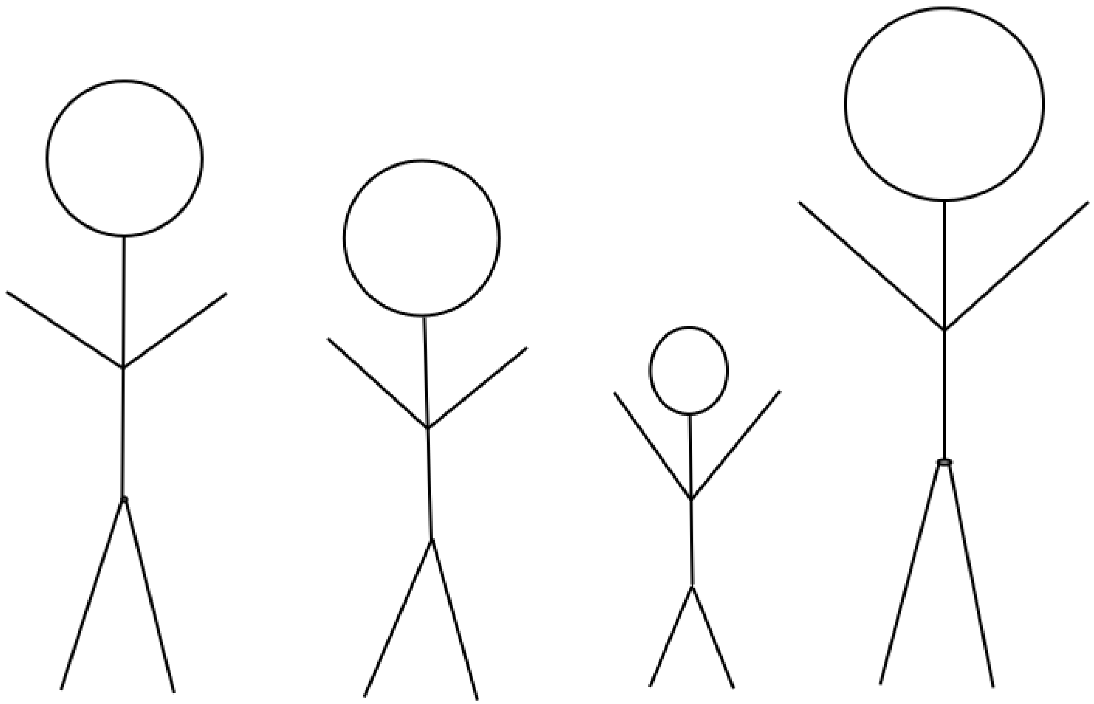
- Find all students that are candidate for “2nd class honor degree”
  - (3.25 <= gpa <= 3.49)
- Find all emails received in a given period.

Inbox

FROM	KNOW	TO	SUBJECT	SENT TIME ▾
"lawiki.i2p admin" <J5uF>		**Bote User <uhOd>**	**hi**	**Unknown**
anonymous		Bote User <uhOd>	Sanders 2016	Aug 30, 2015 3:27 PM
anonymous		Bote User <uhOd>	I2PCon 2016	Aug 30, 2015 3:25 PM
Anon Developer <gvbM>		**Bote User <uhOd>**	**Re: Bote changess**	**Aug 30, 2015 2:54 PM**
I2P User <uUUx>		**Bote User <uhOd>**	**Hello World!**	**Aug 30, 2015 2:51 PM**

Closest Height

- Find the person in your class whose height is closest to yours



Local Search

- A Local Search Data Structure stores a number of elements each with a key coming from an ordered set.
- It supports operations:
  - RangeSearch (x, y): Return all elements with keys between x and y
  - NearestNeighbors (z): Return the element with keys on either side of z

Example

1	4	6	7	10	13	15

RangeSearch (5, 12)

1	4	6	7	10	13	15

NearestNeighbors (3)

1	4	6	7	10	13	15

Dynamic Data Structure

- We would also like to be able to modify the data structures as we go
  - Insert(x): Adds an element with key x
  - Delete(x): Removes the element with key x

Example

1	4	6	7	10	13	15

Insert (3)

1	3	4	6	7	10	13	15

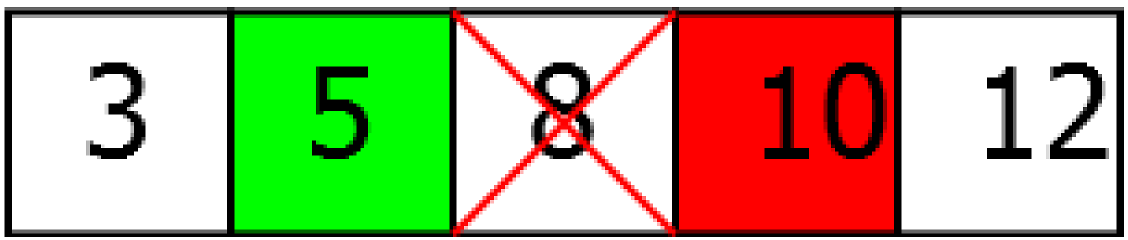
Delete (10)

1	3	4	6	7	13	15

Problem

- If an empty data structure is given these commands what does it output at the end?
  - Insert(3)
  - Insert(8)
  - Insert(5)
  - Insert(10)
  - Delete(8)
  - Insert(12)
  - NearestNeighbors(7)

Answer

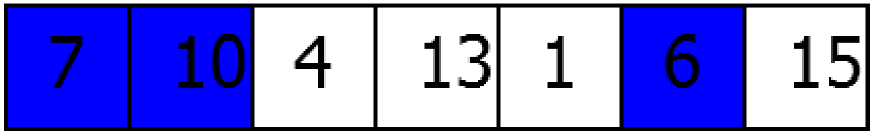


What should you use to implement Local Search Data Structure?

- Choices
  - Array
  - Sorted Array
  - Linked-list

Array

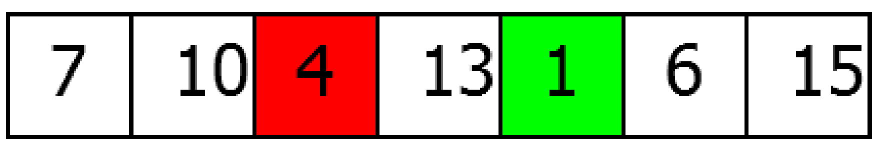
RangeSearch (for example 5<=x<=12)(incl. Find) O(n)



Array

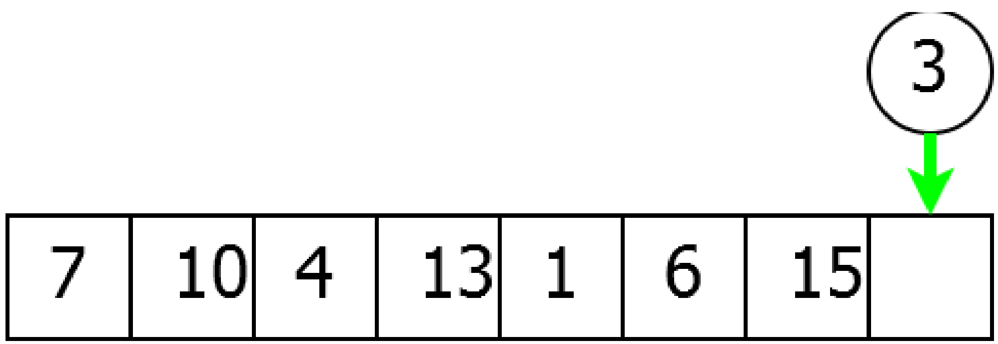
RangeSearch O(n) ✗

NearestNeighbors (e.g.3) O(n) ✗



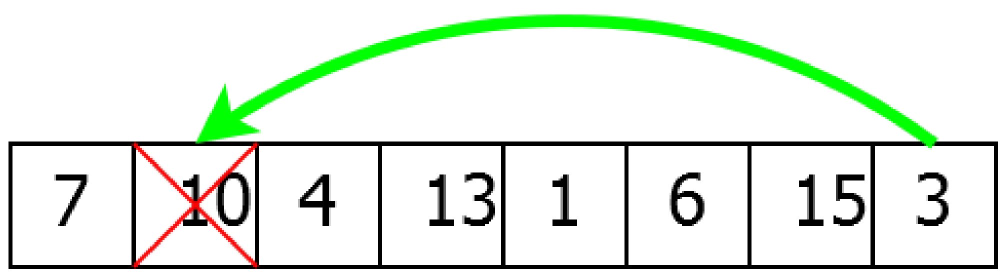
Array

RangeSearch	O(n) ✗
NearestNeighbors	O(n) ✗
Insert	O(1) ✓



Array

RangeSearch	O(n) ✗
NearestNeighbors	O(n) ✗
Insert	O(1) ✓
Delete (Replacement technique)	O(1) ✓

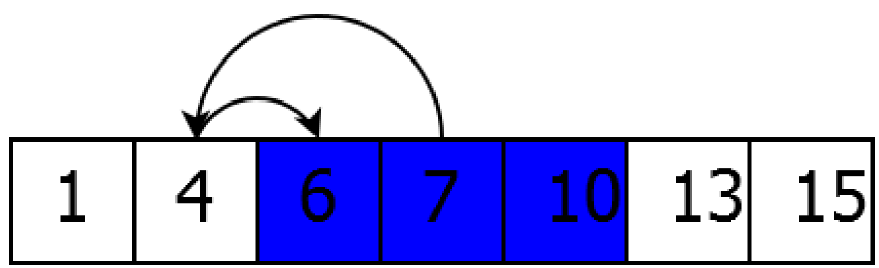


* Note that Replacement technique will destroy order property of the ordered array

Sorted Array

RangeSearch (for example 5<=x<=12) O(log n) ✓

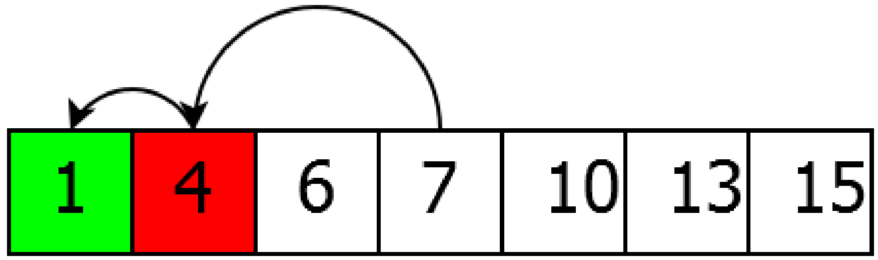
- Range Search in Sorted Array enables Binary Search
- Algo. Binary search to find the left end of the range in our array and that takes logarithmic time.
- And then scan through until we hit the right end of the range we want and return everything in the middle.
- Assume Range << N otherwise it will be O(n) for the range search



Sorted Array

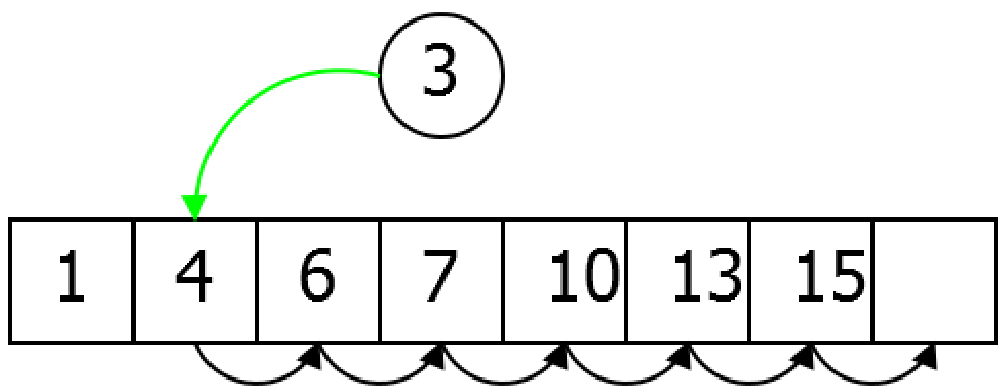
RangeSearch O(log n) ✓
NearestNeighbors (for example 3) O(log n) ✓

Find the key in logarithm time then return the elements on either sides



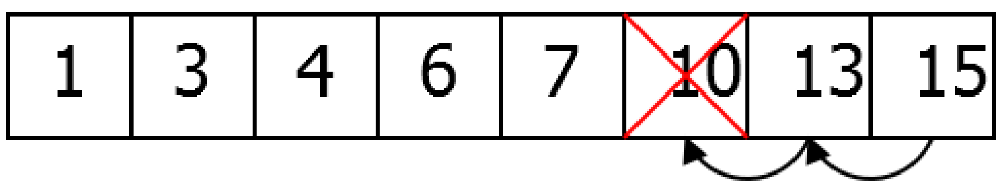
Sorted Array

RangeSearch	O(log n) ✓
NearestNeighbors	O(log n) ✓
Insert	O(n) ✗



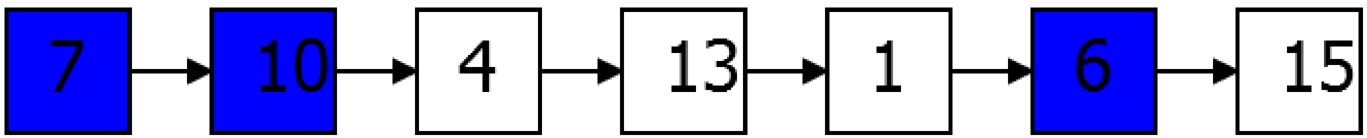
Sorted Array

RangeSearch	O(log n) ✓
NearestNeighbors	O(log n) ✓
Insert	O(n) ✗
Delete	O(n) ✗



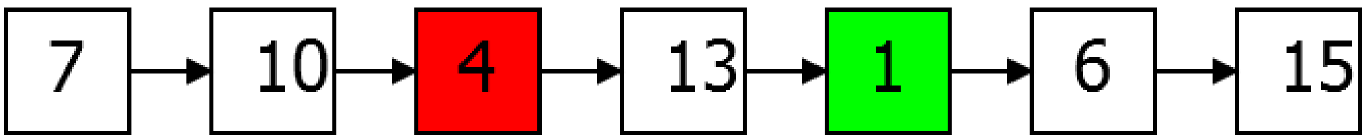
Linked List

RangeSearch (for example 5<=x<=12) O(n) ✗



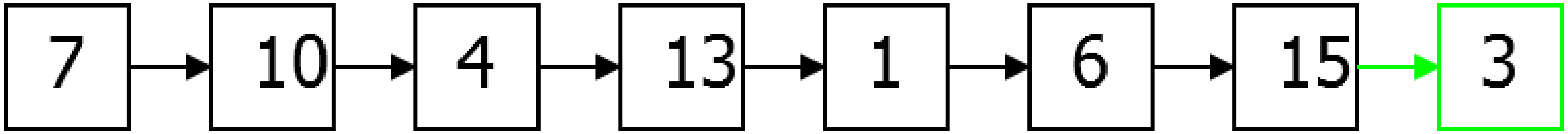
Linked List

RangeSearch	O(n) ✗
NearestNeighbors (e.g. 3)	O(n) ✗



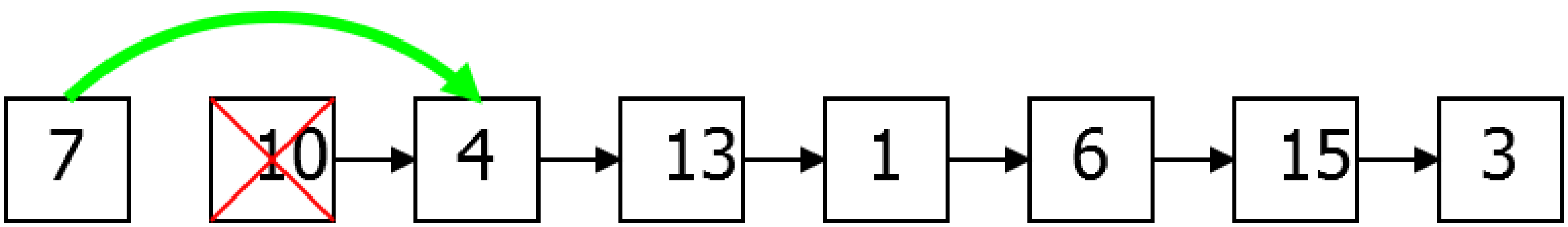
Linked List

RangeSearch	O(n) ✗
NearestNeighbors	O(n) ✗
Insert	O(1) ✓



Linked List

RangeSearch	O(n) ✗
NearestNeighbors	O(n) ✗
Insert	O(1) ✓
Delete	O(1) ✓



Need Something New

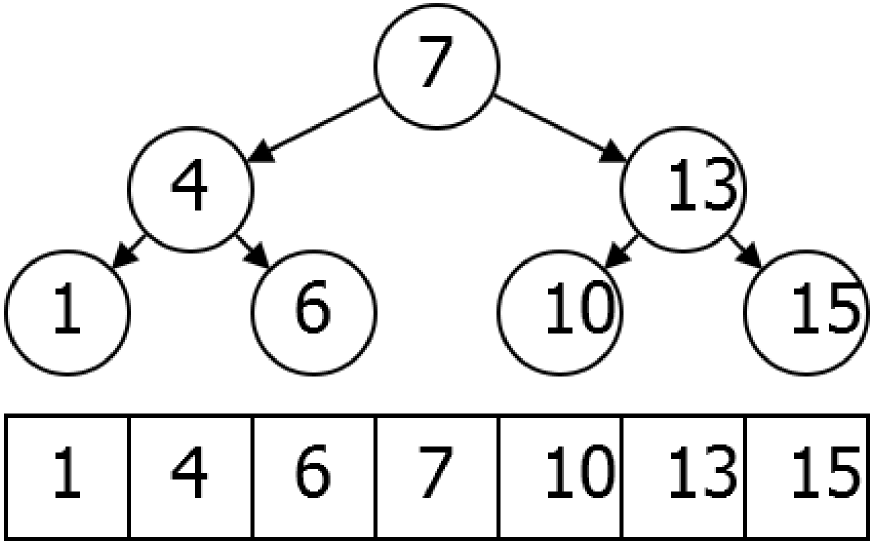
Problem

- Previous linear data structures won't work. We need something new
- Want data structure for local search
- Sorted arrays can do fast search but update is too slow

Let's combine Binary Search with Binary Tree

Binary Search Tree

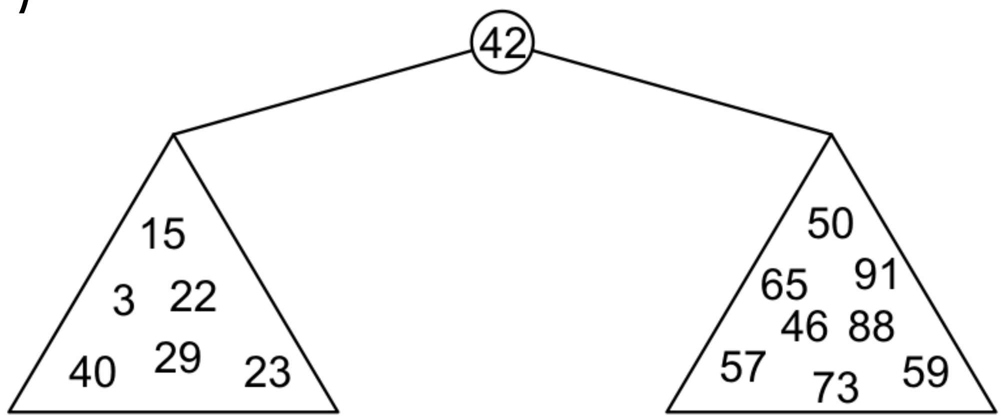
- Tree search is fast
- Tree is much easier to insert



What kind of traversal is this?

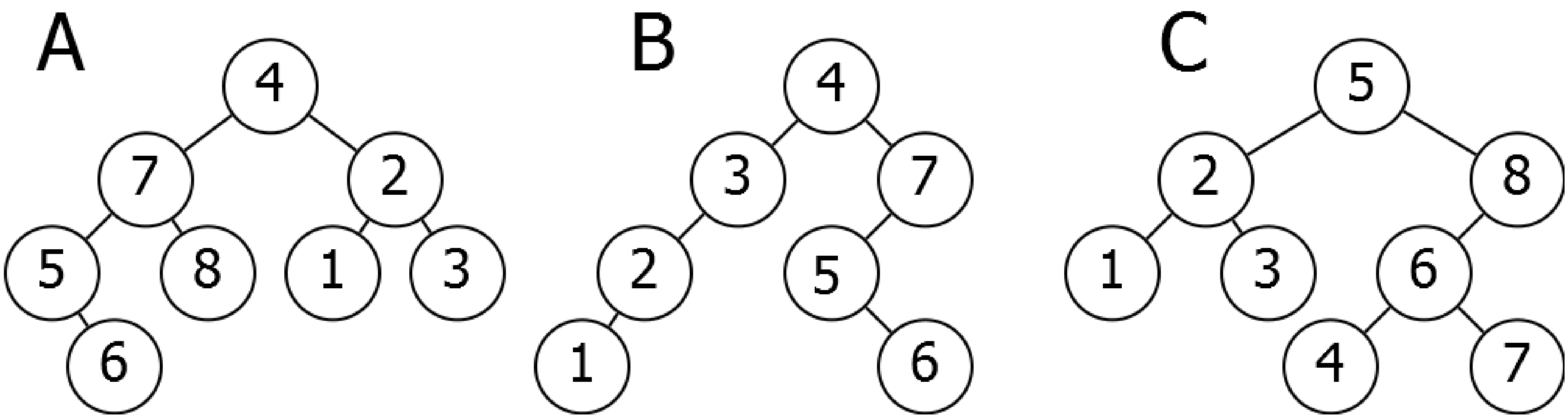
Search Key Property

- Key of the root nodes is always
  - Greater than any node in the left sub-tree
  - Smaller than any node in the right sub-tree
- Each of the two sub-trees is a binary search tree
- Assume that keys are unique (not duplicate on any other node)



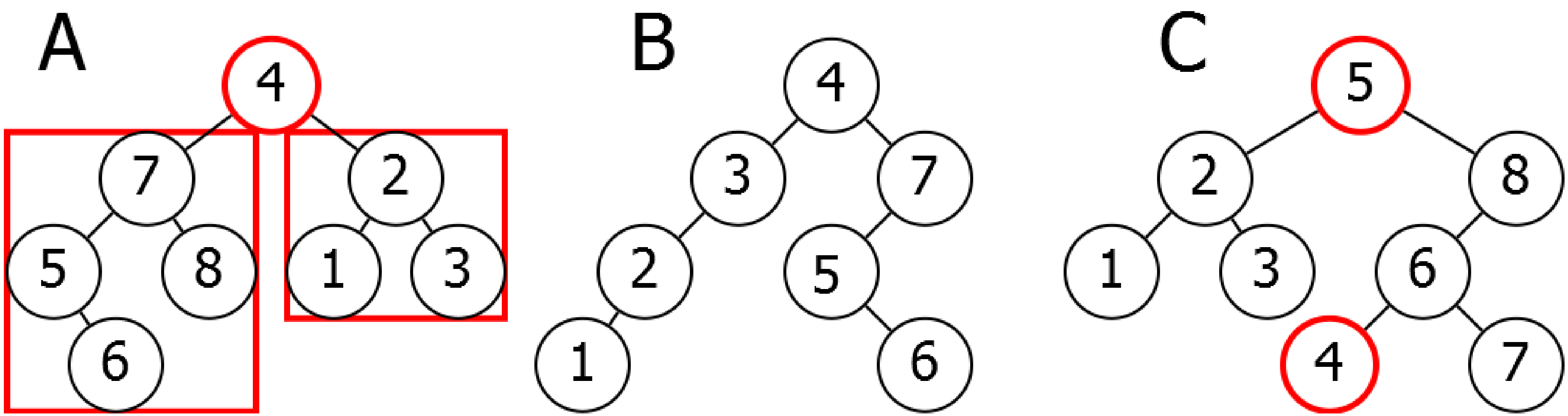
Problem

- Which of the following trees satisfies the Binary Search Tree Property?



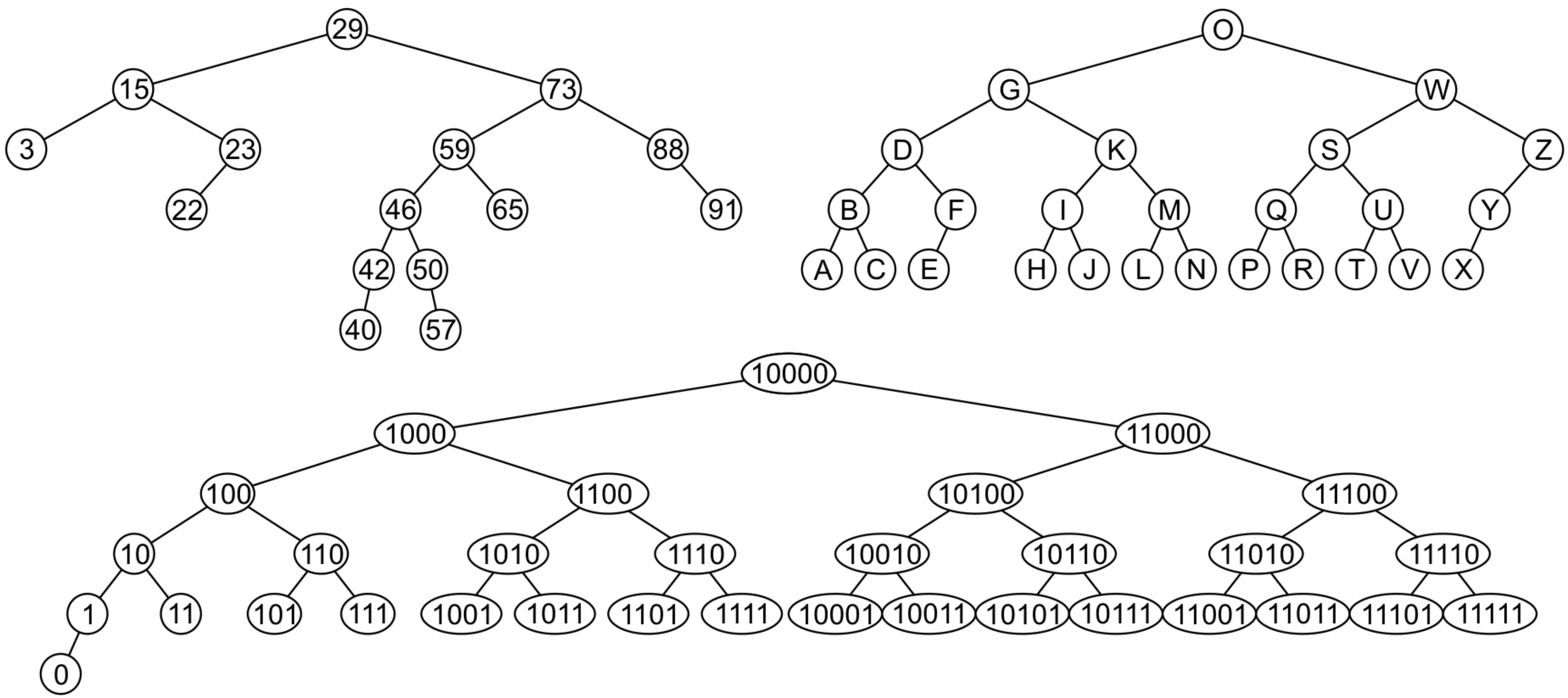
Problem

- Which of the following trees satisfies the Binary Search Tree Property?



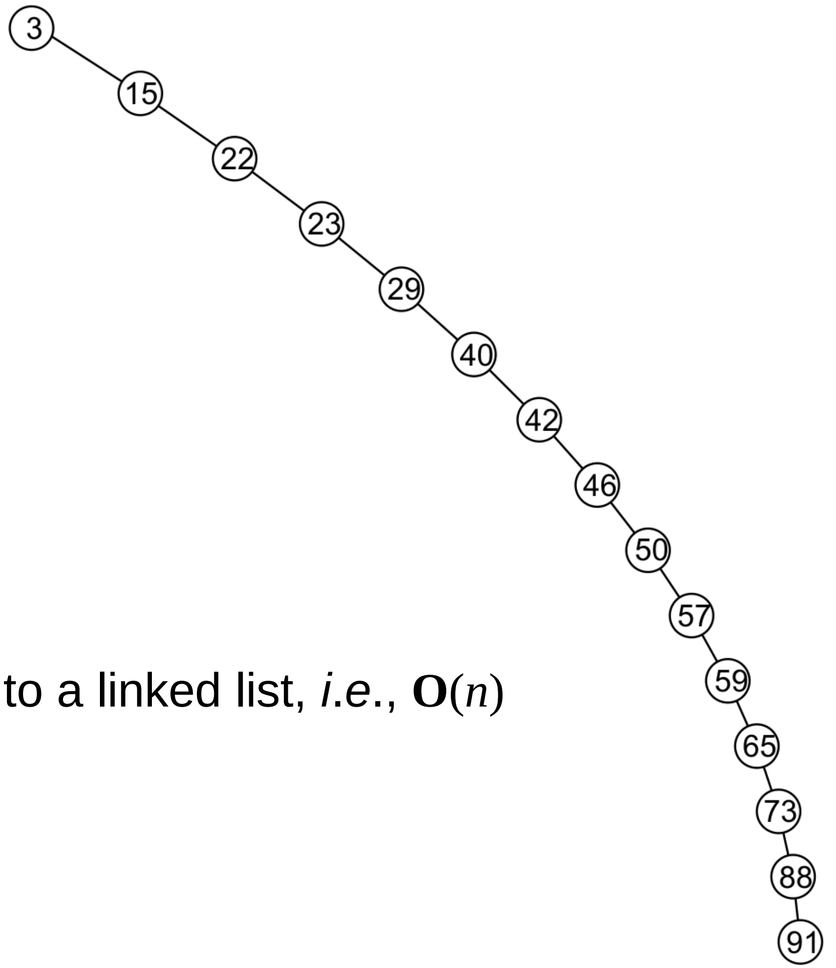
Examples

Here are other examples of binary search trees:



Examples

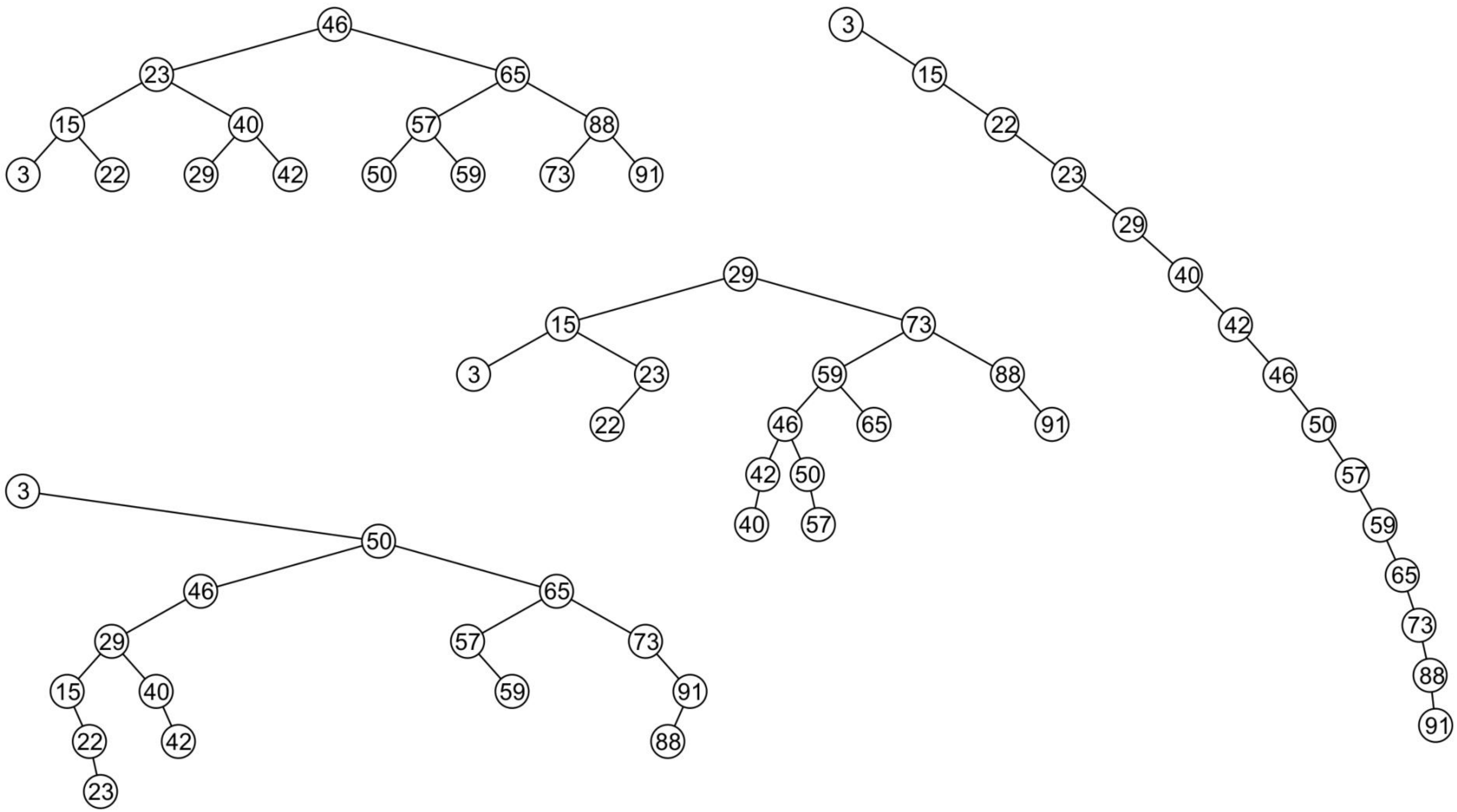
Unfortunately, it is possible to construct *degenerate* binary search trees



– This is equivalent to a linked list, *i.e.*, $\mathbf{O}(n)$

Examples

All these binary search trees store the same data

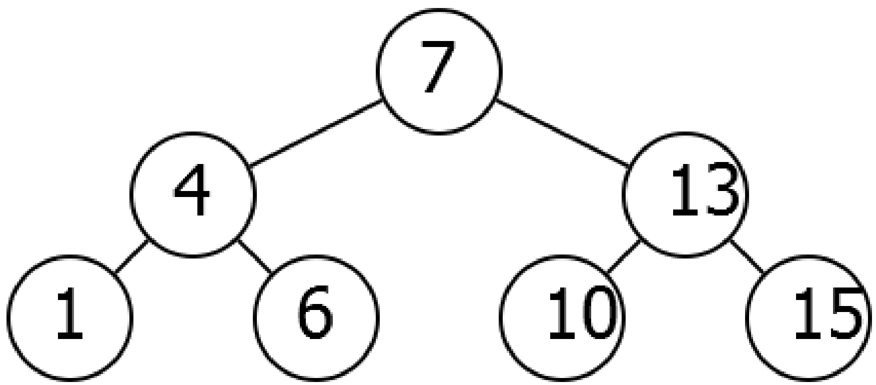


Binary Search Tree Operations

1. Find (FindClosest, FindMin, FindMax)
2. FindNext (FindPrevious)
3. RangeSearch
4. Insert
5. Delete
6. FindKthSmallest

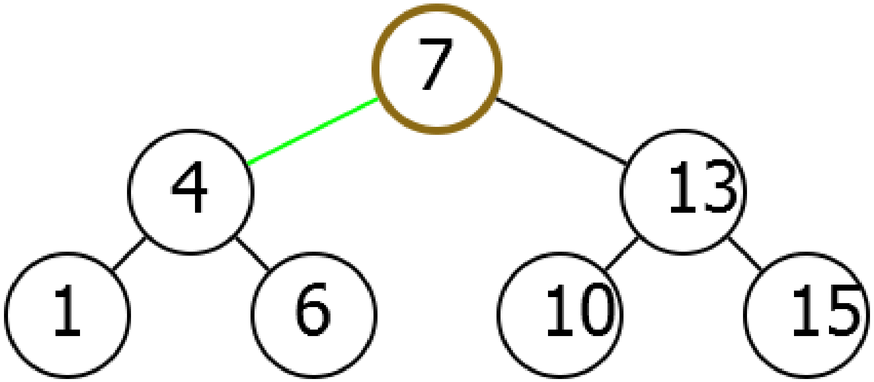
Find Operation

- Find(Node *tree*, Key *k*)
- Find a key *k* in a BST with root node *tree*
- Find(6)
- **Algorithm (Pseudocode)**
  - **If the search key is found, return the root**
  - **If the search key is less than the root, go left**
  - **If the search key is more than the root key, go right**



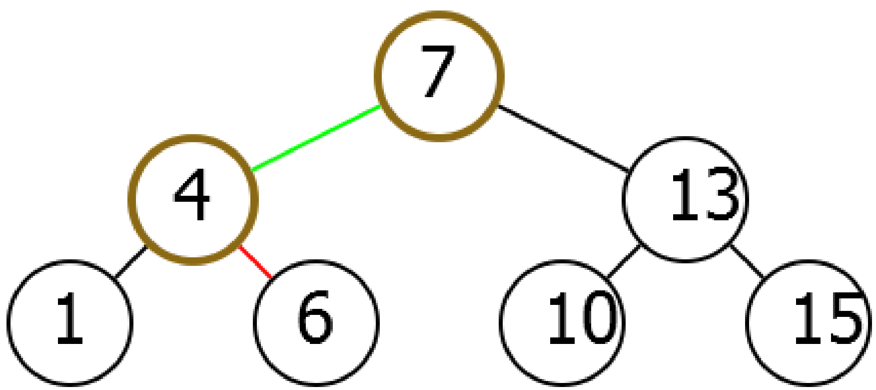
Find Operation

- Find(Node *tree*, Key *k*)
- Find a key *k* in a BST with root node *tree*
- Find(6)
- **Algorithm (Pseudocode)**
  - **If the search key is found, return the root**
  - **If the search key is less than the root, go left**
  - **If the search key is more than the root key, go right**



Find Operation

- Find(Node *tree*, Key *k*)
- Find a key *k* in a BST with root node *tree*
- Find(6)
- **Algorithm (Pseudocode)**
  - **If the search key is found, return the root**
  - **If the search key is less than the root, go left**
  - **If the search key is more than the root key, go right**



Find Operation

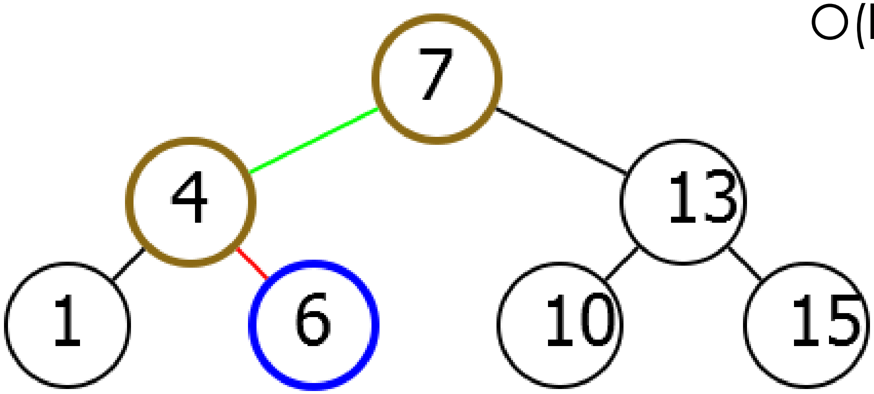
- Find(Node *tree*, Key *k*)
- Find a key *k* in a BST with root node *tree*
- Find(6)
- **Algorithm (Pseudocode)**
  - **If the search key is found, return the root**
  - **If the search key is less than the root, go left**
  - **If the search key is more than the root key, go right**

Tree with height h
What is the worst case run time of Find operation?

O(h)

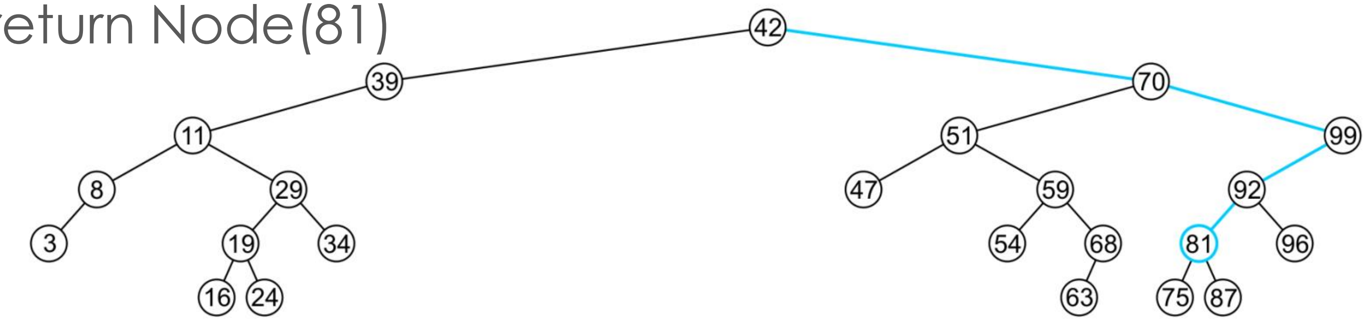
Complete BST with n nodes?

O(log n)

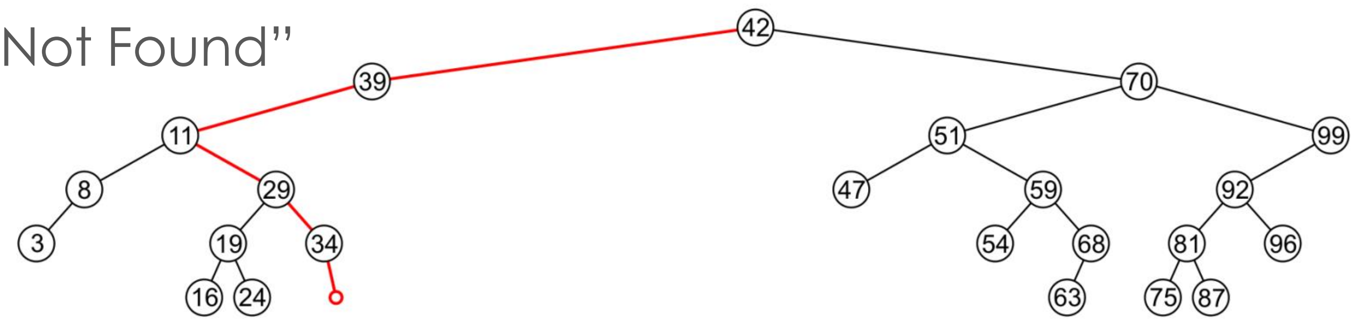


Find

- Find(81)
  - return Node(81)

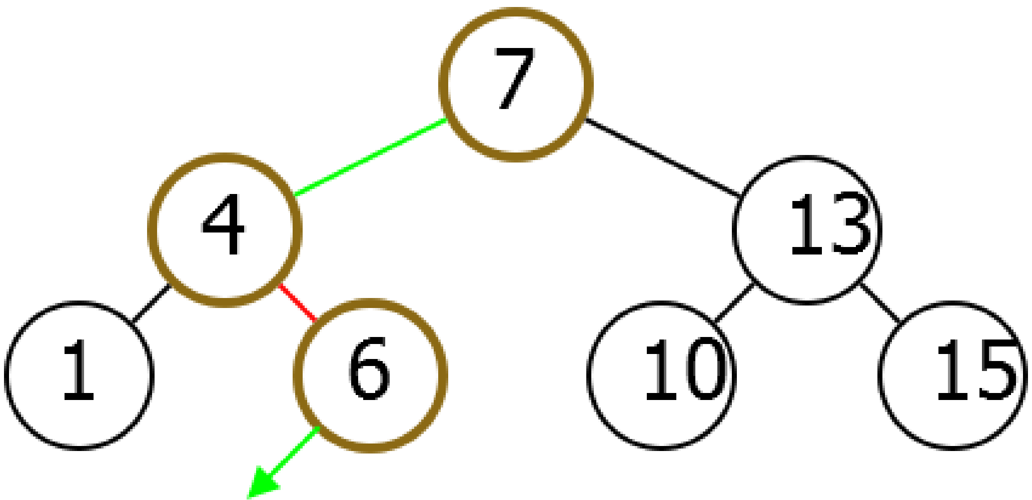


- Find(36)
  - “Not Found”



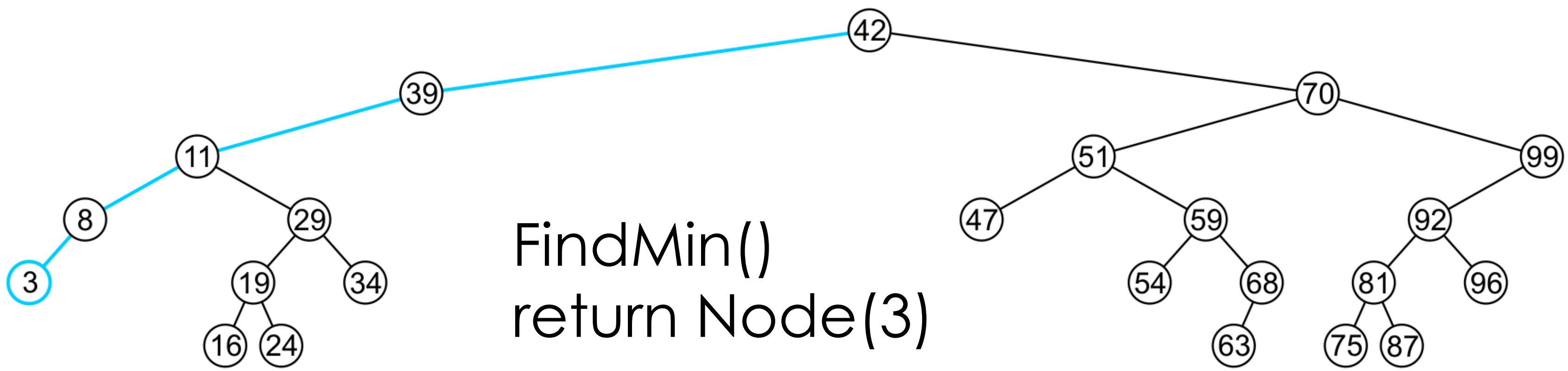
FindClosest

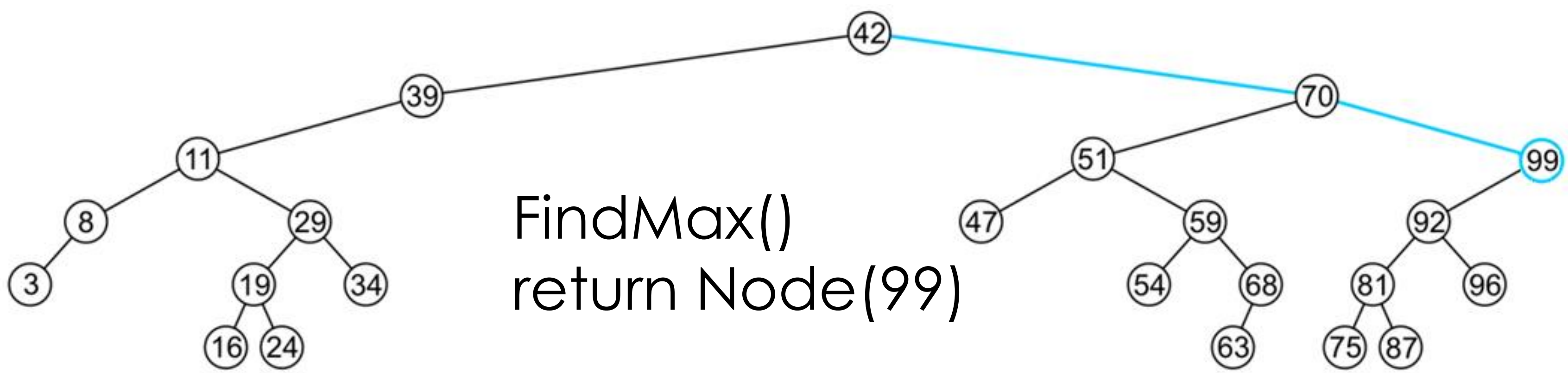
- Homework or Exams?



FindClosest(5)
return Node(6)

FindMin and FindMax

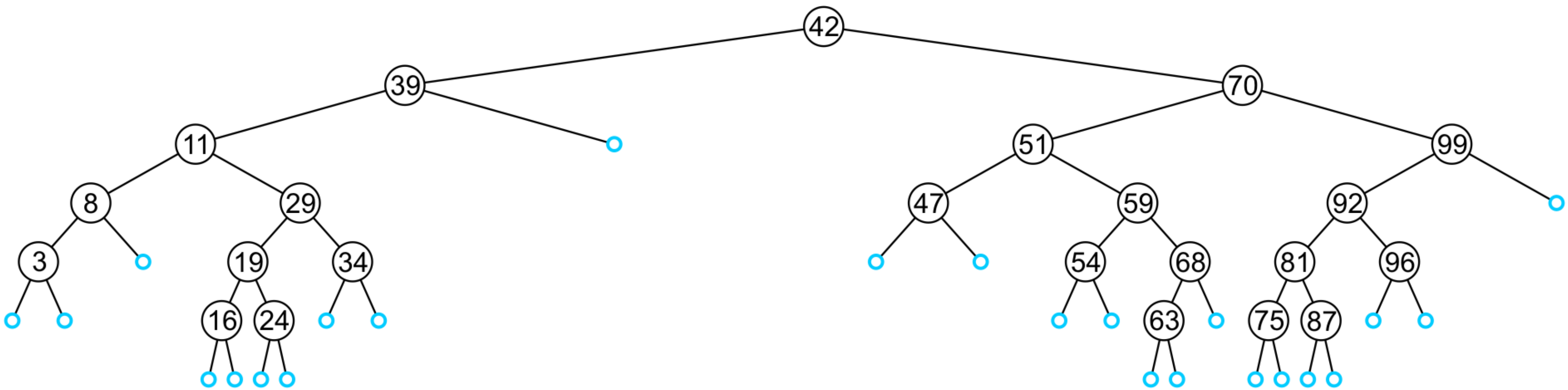




Insertion Operation

An insertion will be performed at a leaf node:

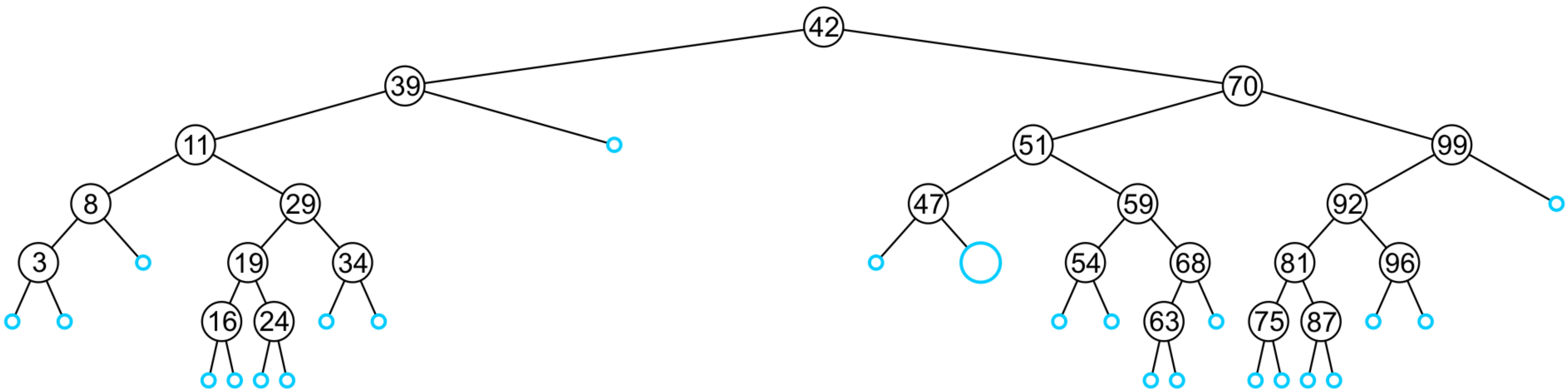
– Any empty node is a possible location for an insertion



The values which may be inserted at any empty node depend on the surrounding nodes

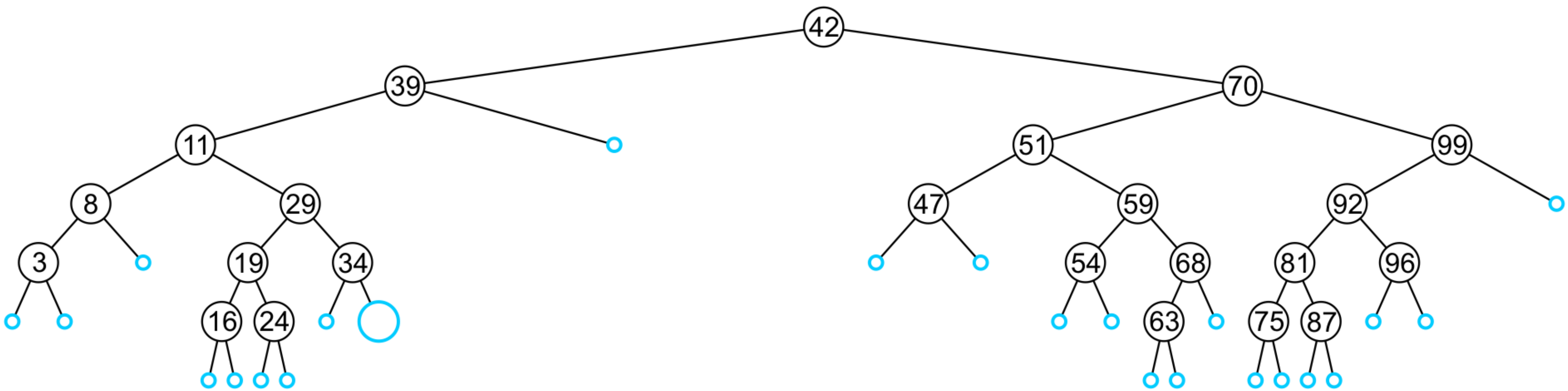
Insertion

For example, this node may hold 48, 49, or 50



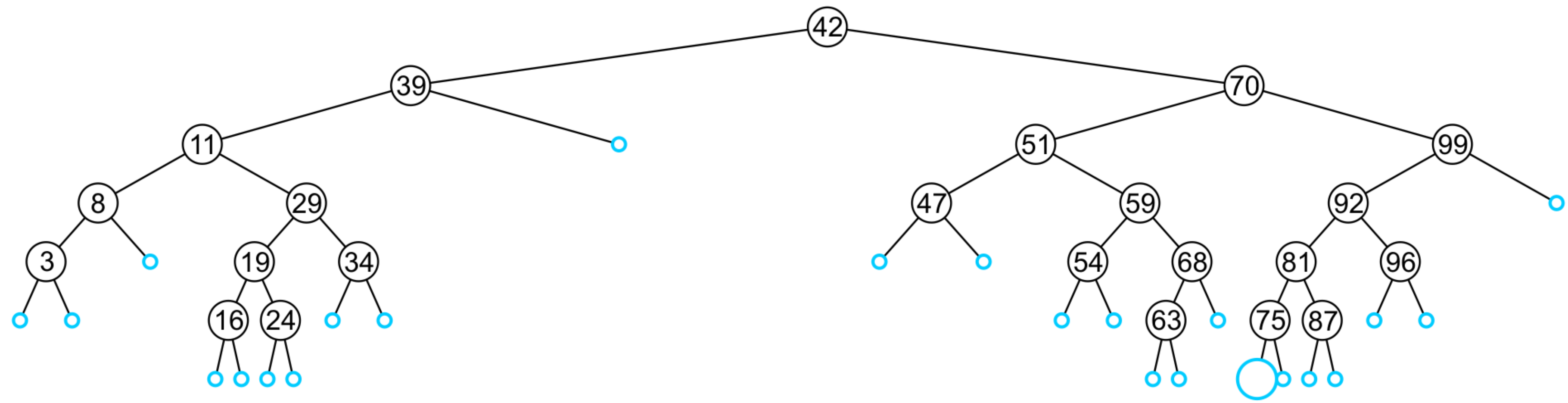
Insertion

For example, this node may hold 35, 36, 37, or 38



Insertion

For example, this node may hold 71 to 74



Insert Algorithm Ver.1 (Use findClosest)

Step. 1:

r = FindClosest(inserting key)

Step. 2

If r.key is the same as the inserting key, then return because the key is duplicate

Step. 3

- If the inserting key < r.key then hang the new node to the left child
- If the inserting key > r.key then hang the new node to the right child

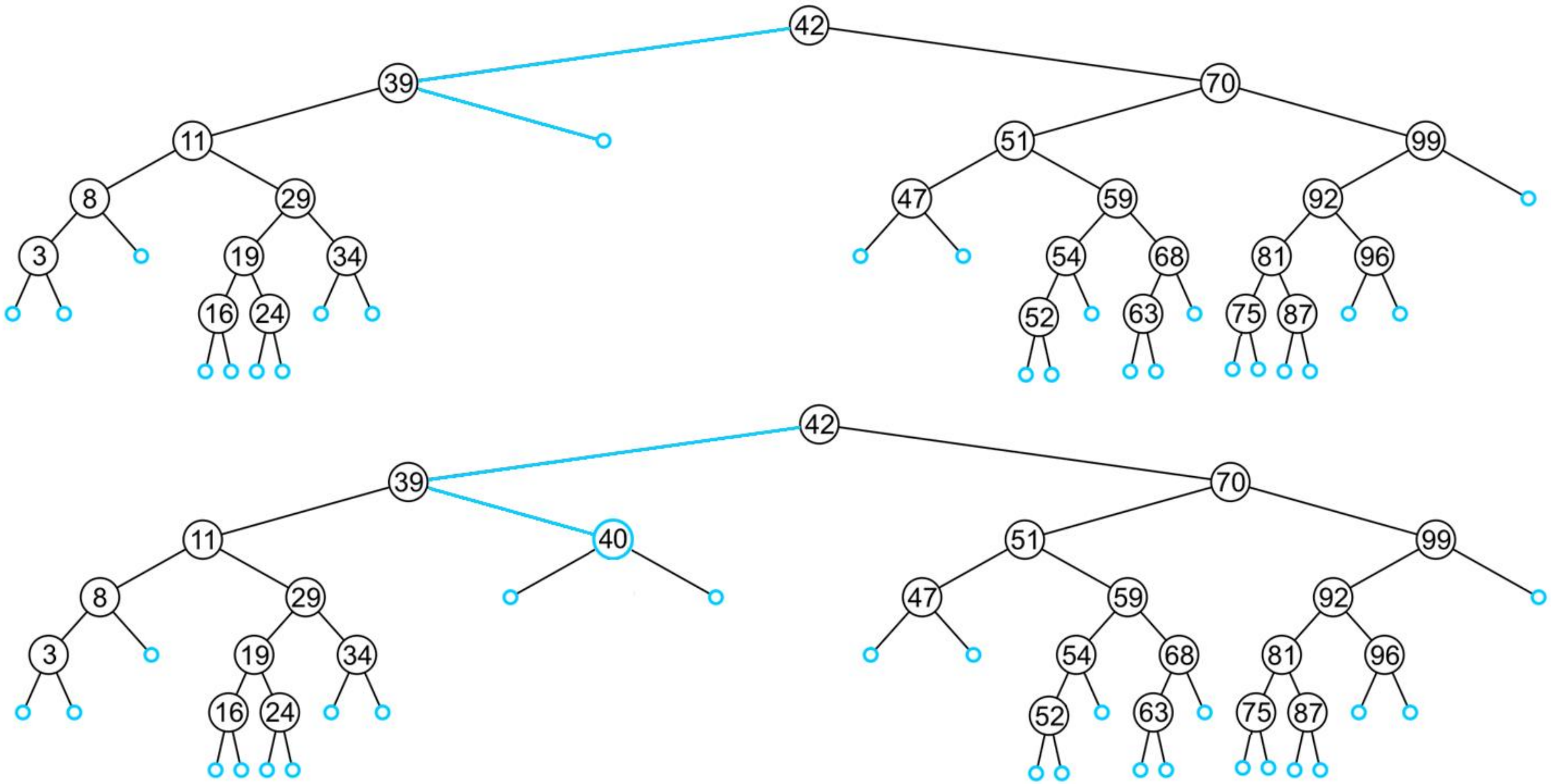
Insert Algorithm Ver.2 (Without findClosest)

Like Find, we will step through the tree

- If the inserting key < node.key then go left
- If the inserting key > node.key then go right
- If we find the object already in the tree, we will return
  - The object is already in the binary search tree (no duplicates)
- Until we arrive at an empty node
- The object will be inserted into that location
- The run time is $\mathbf{O}(h)$
- Complete tree with n nodes?

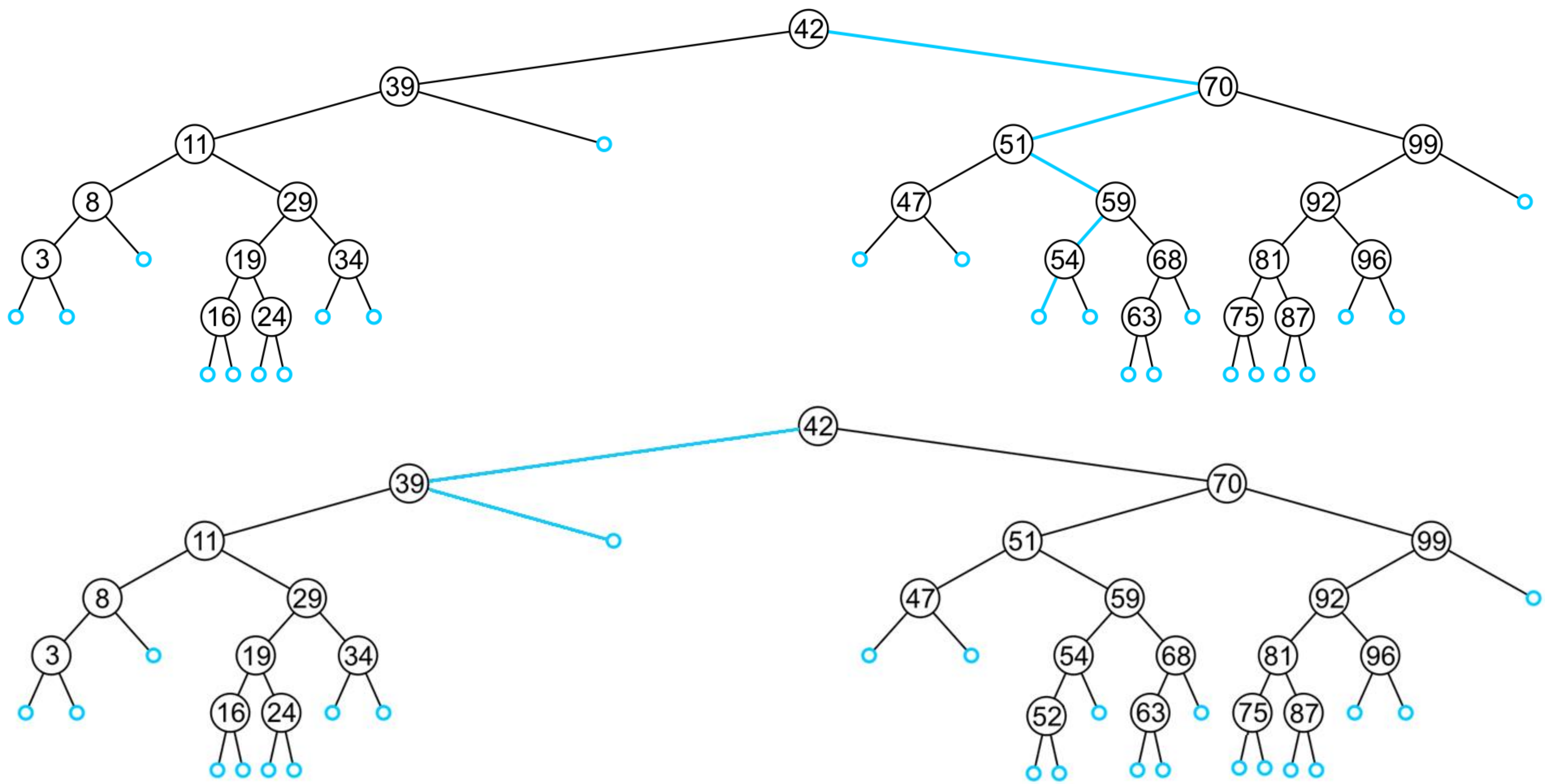
Insertion

- Insert(40)



Insertion

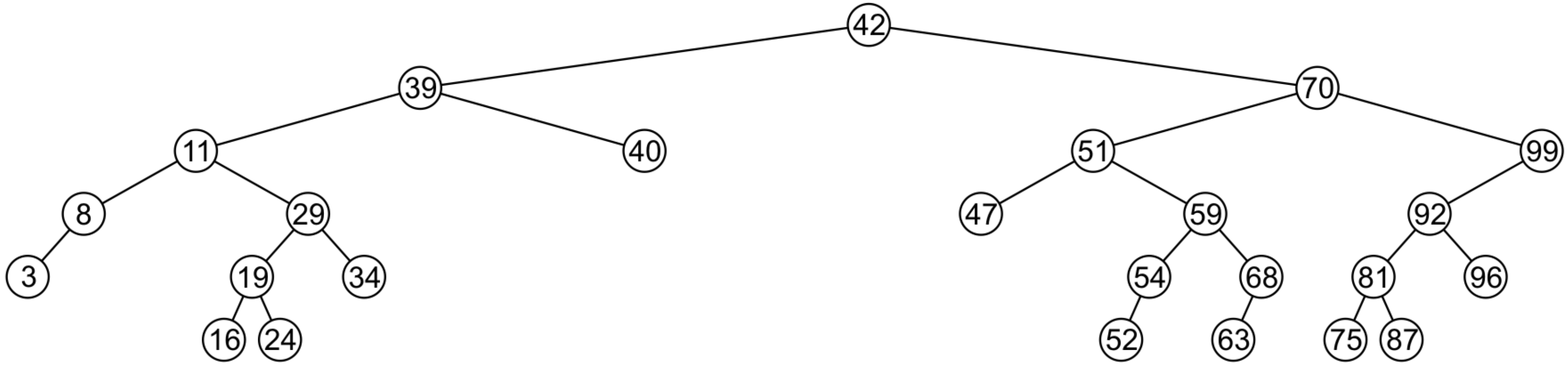
- Insert(52)



Deletion Operation

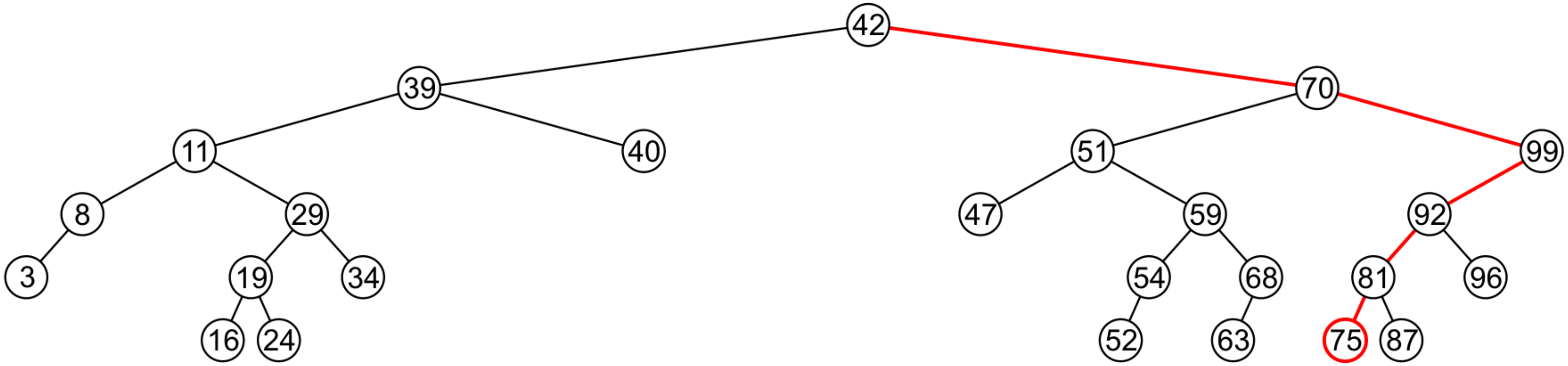
- There are 2 major cases with 3 sub-cases (6 cases in totol)
  - Delete the root
  - Delete Non-root nodes

- For each major case, there are 3 sub-cases for consideration:
  1. The node has no children (leaf node)
  2. The node has only one child
  3. The node has two children (full node)



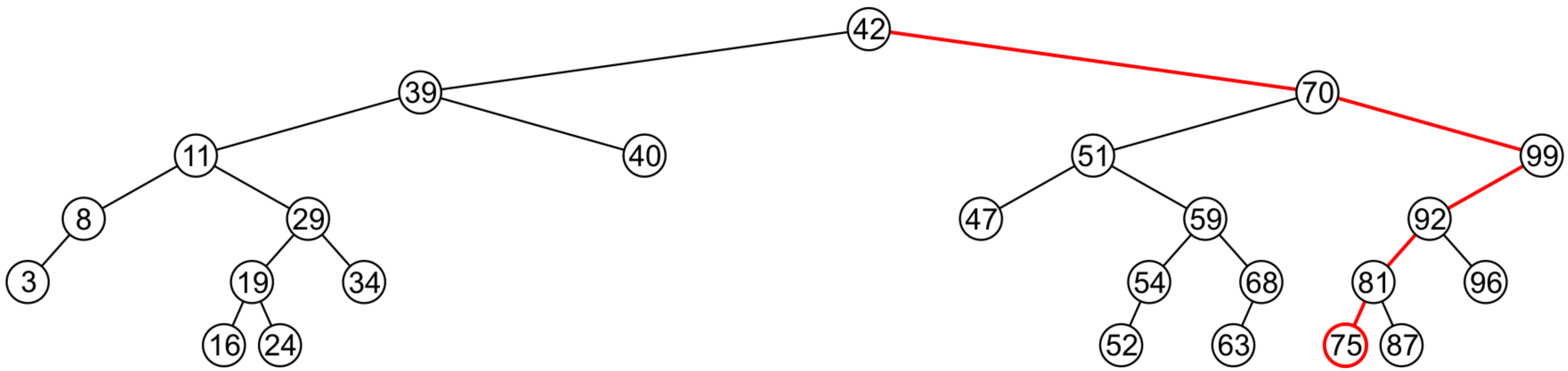
Deletion Operation: Case 1

- A leaf node can be simply deleted
- Delete(75)
- Find the Node(key==75) first
- Check if it is a leaf node?



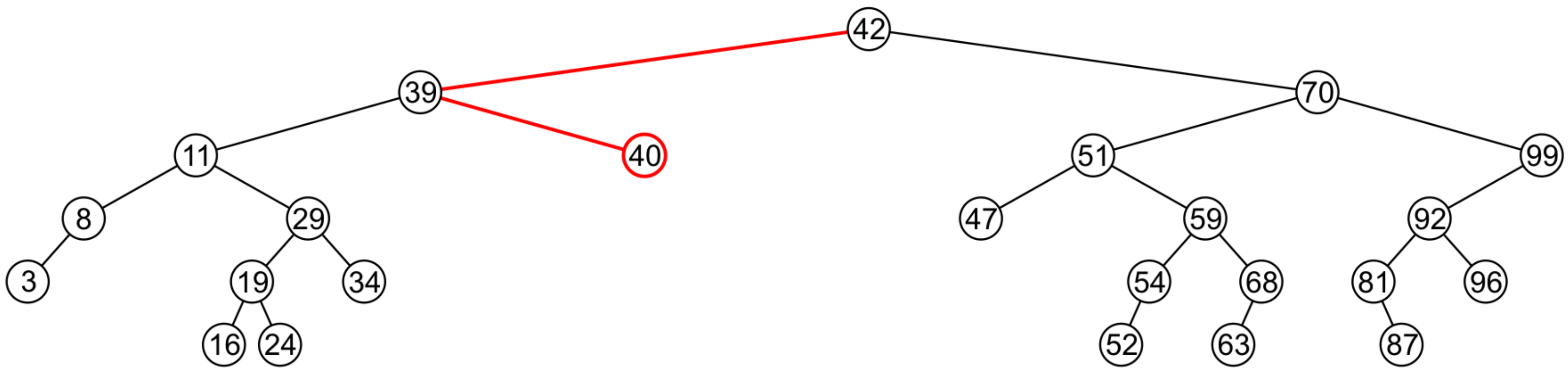
Deletion Operation: Case 1

- A leaf node can be simply deleted
- Delete(75)
- If yes, this will be Case 1 -> Simply delete the node
- Set the pointer of the parent to the node to null (node.parent.left = null)
  - You may need to explicitly delete Node(75) in C or C++



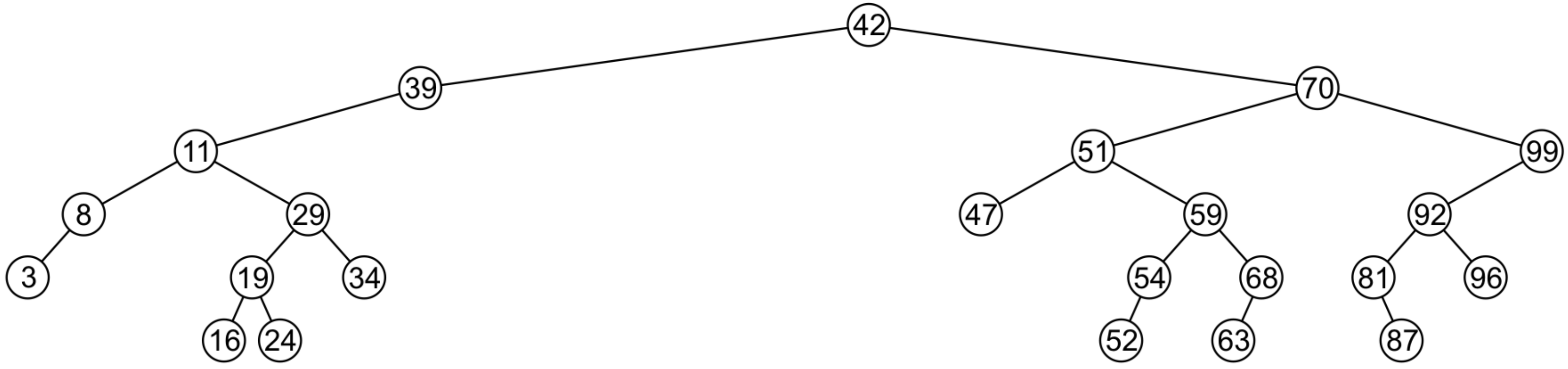
Deletion Operation: Case 1

- A leaf node can be simply deleted
- Delete(40)
- Find Node(key==40) and check if it is the leaf node



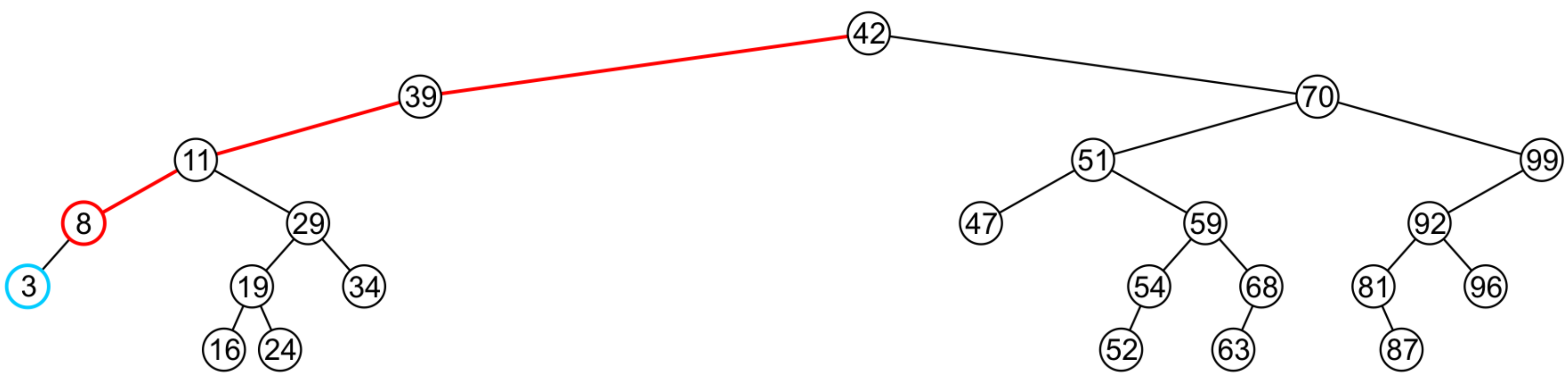
Deletion Operation: Case 1

- A leaf node can be simply deleted
- Delete(40)
- if yes (Case 1), then simply delete by seting the parent's pointer to the node to null (node.parent.right = null)
  - You may need to explicitly delete Node(39) in C or C++



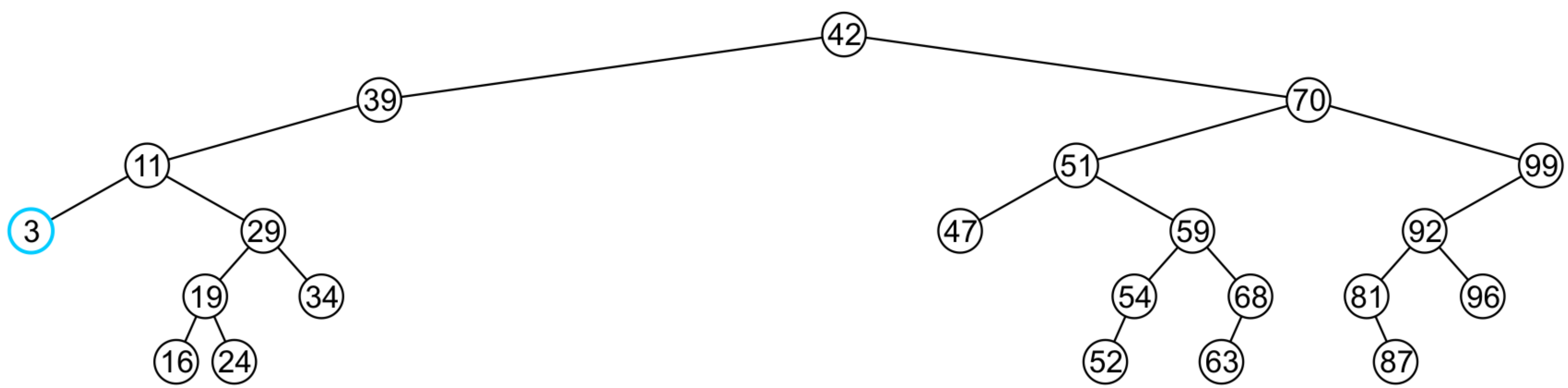
Deletion Operation: Case 2

- If a node has only one child, we can simply replace the node with its child (sub-tree)
- Delete(8)
- Find parent of the Node(key==8) first, and then check if it has only a child (if yes, this will be Case 2)
- If yes (Case 2), then delete the node and promote its child (node.parent.left = node.left)



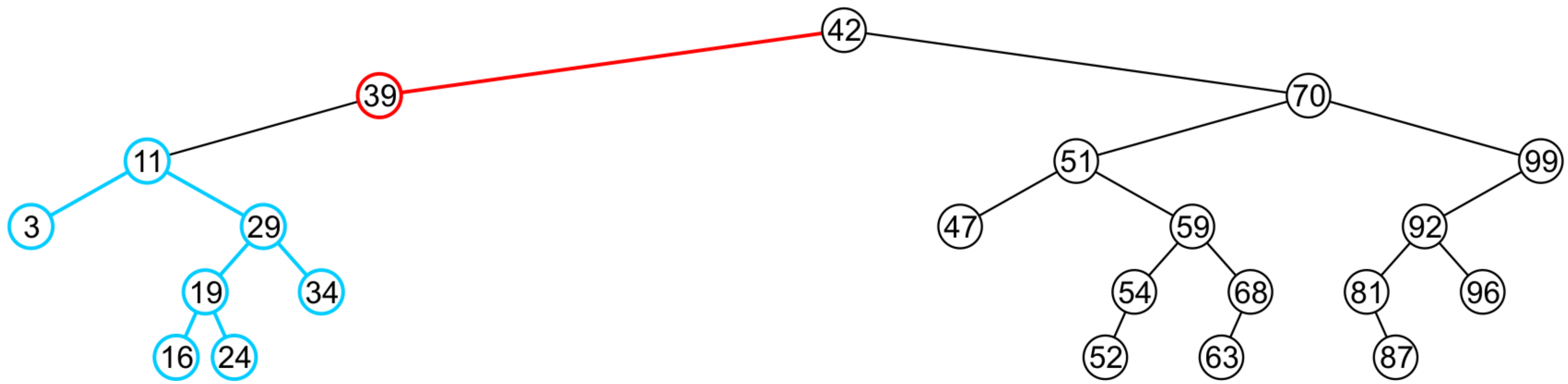
Deletion Operation: Case 2

- If a node has only one child, we can simply replace the node with its child (sub-tree)
- Delete(8)
- If yes (Case 2), delete the node and promote its child (node.parent.left = node.left)



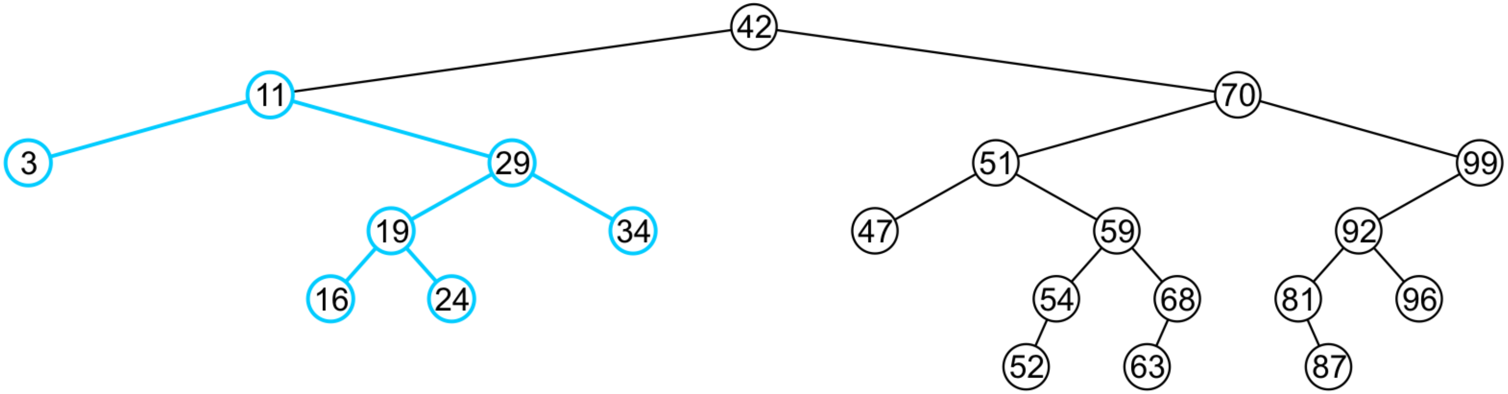
Deletion Operation: Case 2

- If a node has only one child, we can simply replace the node with its child (sub-tree)
- Delete(39)
- Find parent of the Node(key==39) first, and then check if it has only one child (if yes, this will be Case 2)



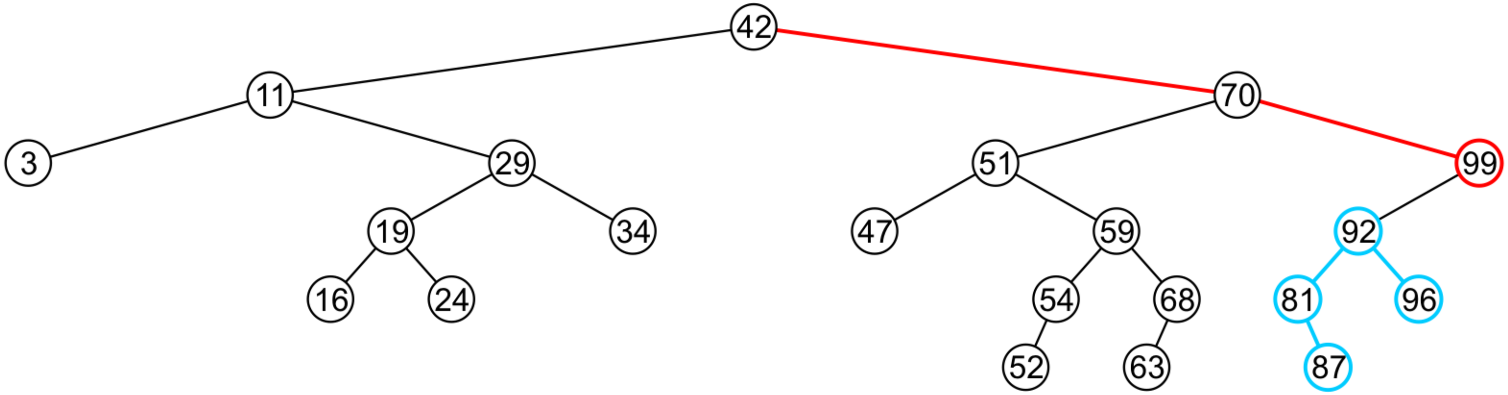
Deletion Operation: Case 2

- If a node has only a left child, we can simply replace the node with the left child (sub-tree)
- Delete(39)
- If Case 2, then you can delete the node and promote its child (sub-tree) (node.parent.left = node.left)



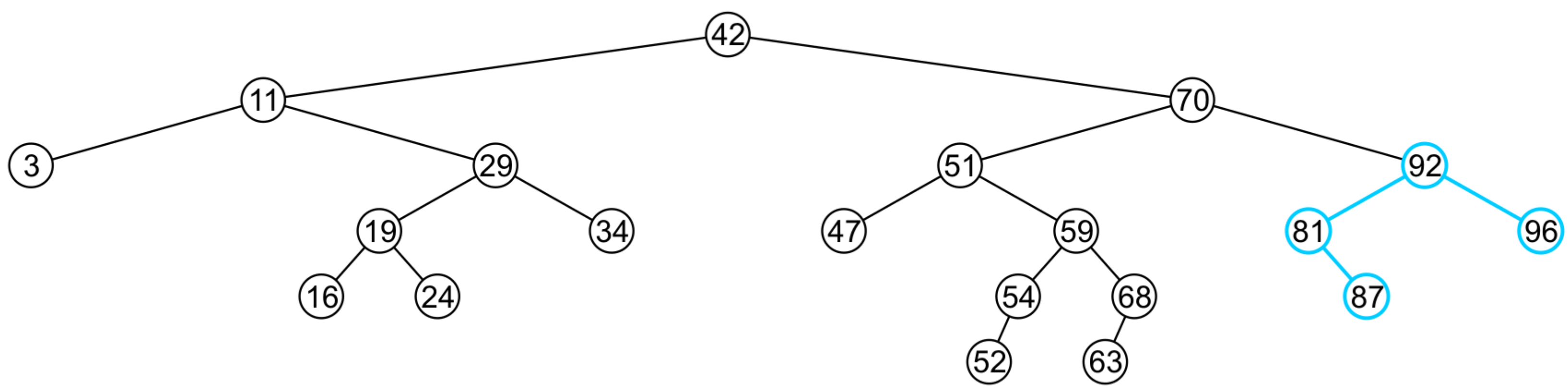
Deletion Operation: Case 2

- If a node has only one child, we can simply replace the node with its child (sub-tree)
- Delete(99)
- Find parent of the Node(key==99) first, and then check if it has only a left child (if yes, this will be Case 2)



Deletion Operation: Case 2

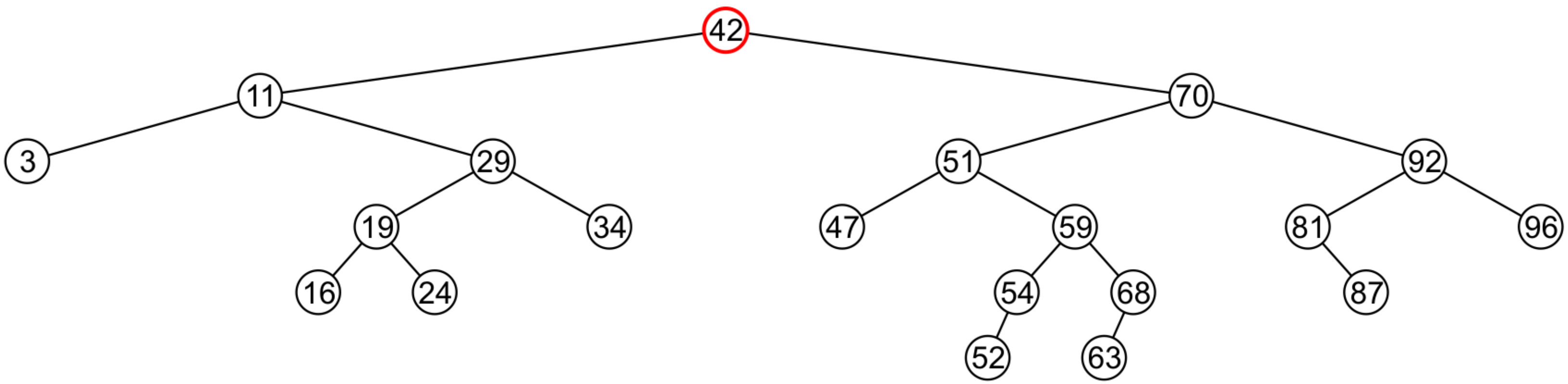
- If a node has only one child, we can simply replace the node with its child (sub-tree)
- Delete(99)
- If yes (Case 2), delete the node and promote its left child (sub-tree) (node.parent.left = node.left)



Deletion Operation: Case 3

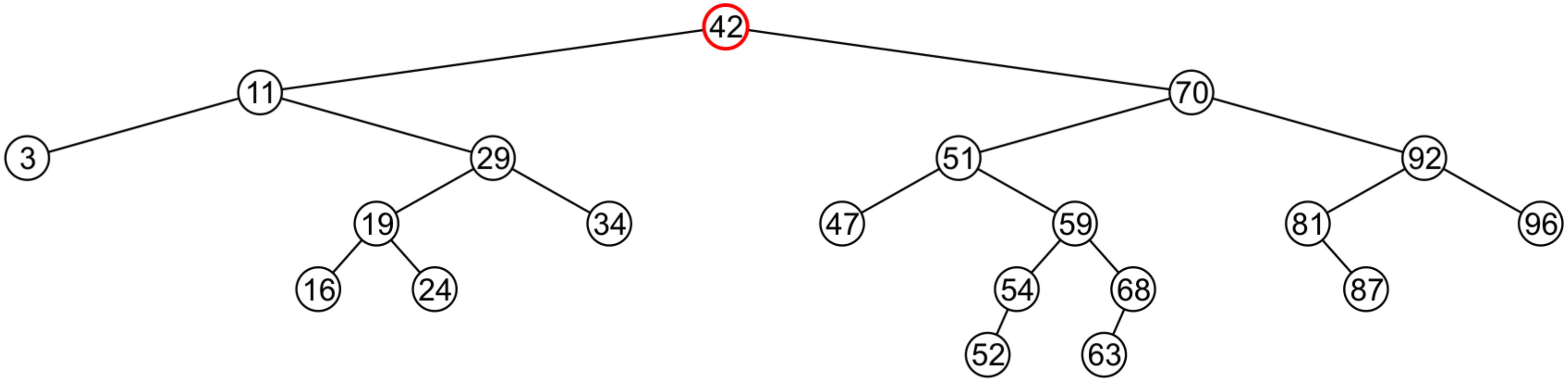
- Finally, If a node has two children (full node), then there are two operations we need to perform
  1. Replace the node with the minimum key from the right sub-tree
  2. Recursively delete the minimum node in the right sub-tree
     - Promote right sub-tree of the minimum node (if any)
  - Alternatively, you can replace the node with the maximum key from the left-sub tree

- Delete (42)



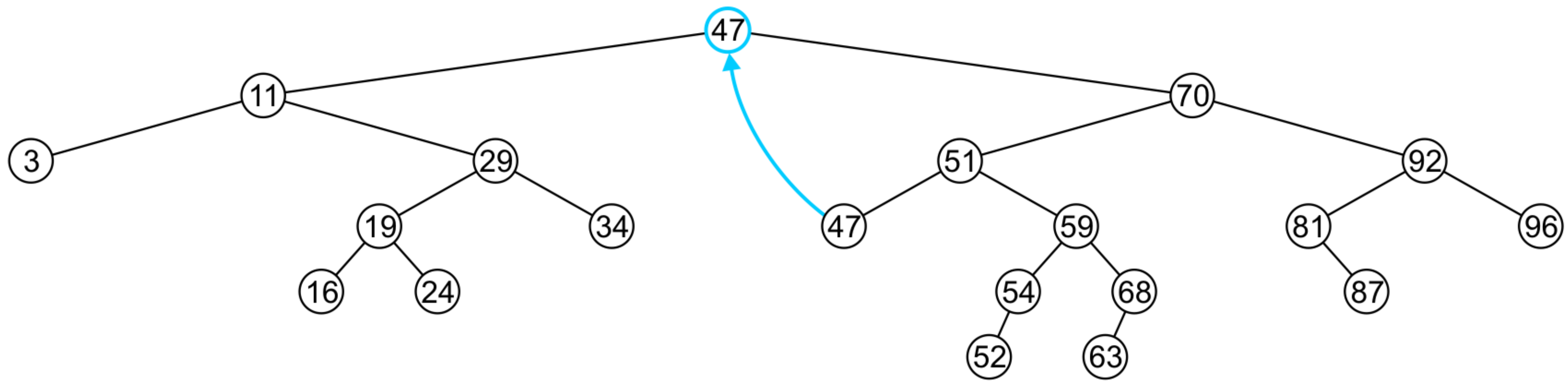
Deletion Operation: Case 3

- Delete (42)
- Firstly find the Node(42) in the tree and then check if the node has two children, if yes then this will be Case 3
- If Case 3, then find a node with the minimum key in the right sub-tree
  - Node(47)



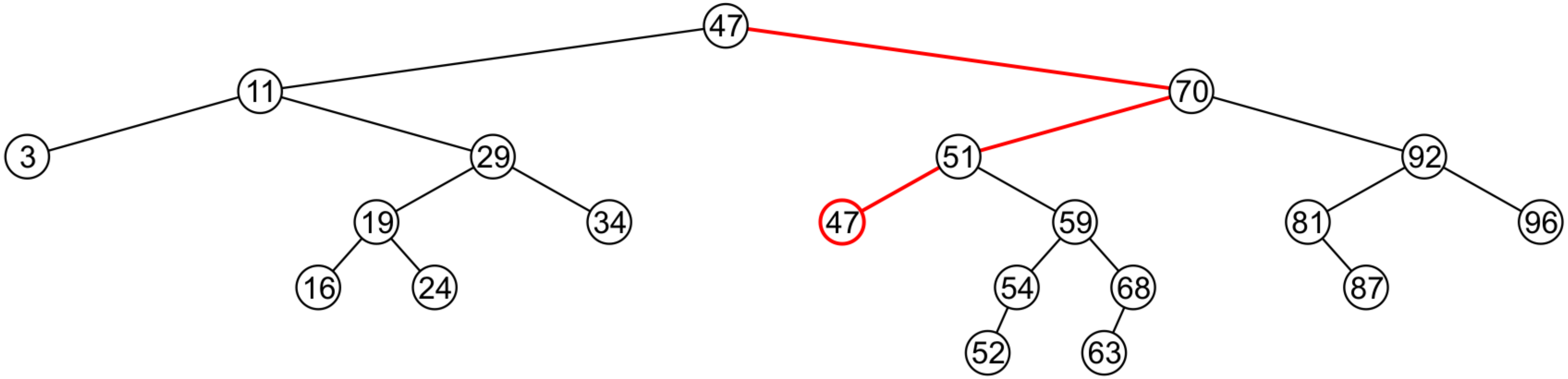
Deletion Operation: Case 3

- Delete (42)
- Then replace the Node(42) with the Node(47)
- We can temporarily have two copies of 47 in the tree



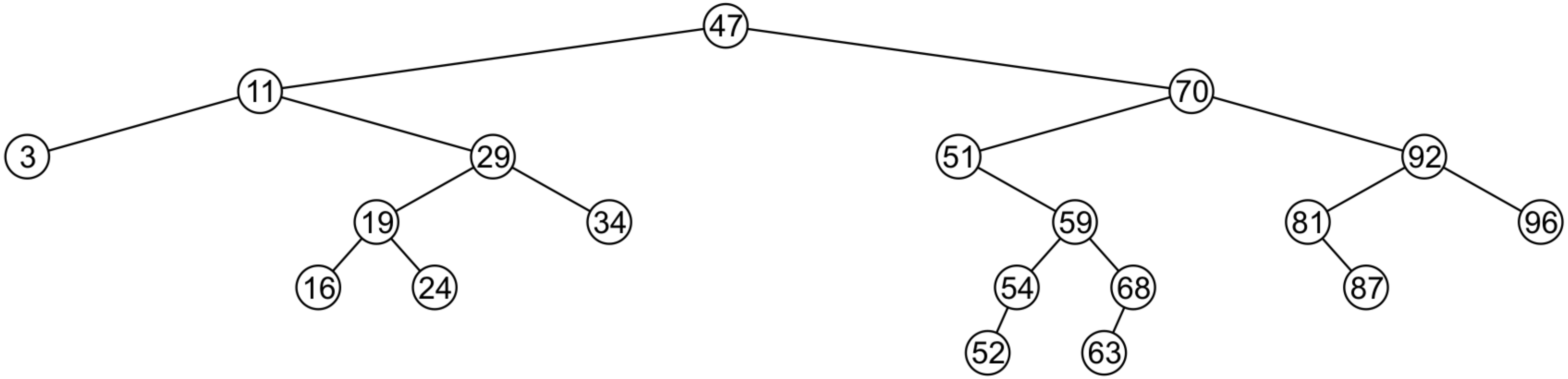
Deletion Operation: Case 3

- Delete (42)
- We now can recursively delete the old Node(47) from the right sub-tree
  - Node 47 is a leaf node in the right sub-tree (Simply delete)



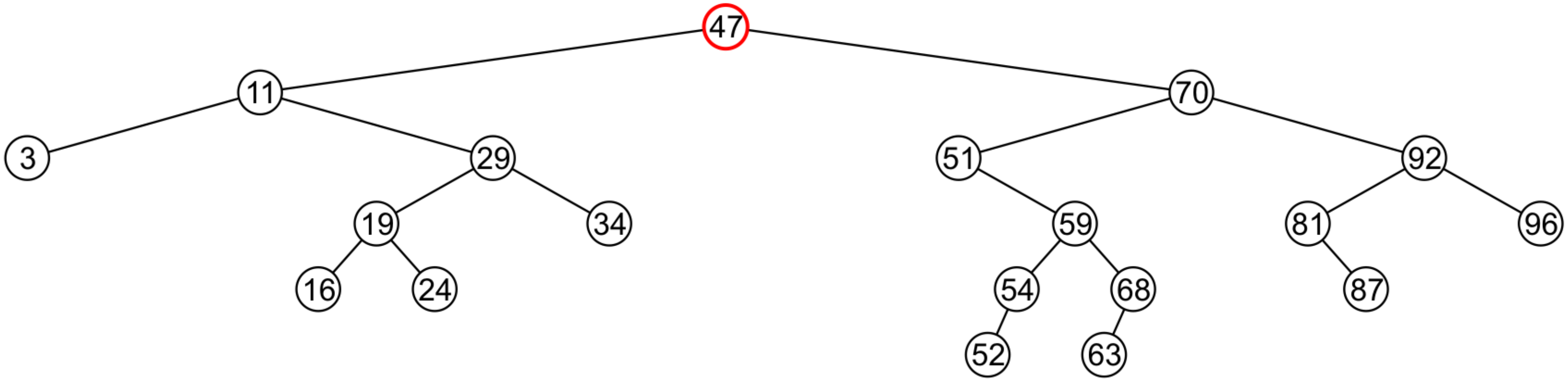
Deletion Operation: Case 3

- Delete (42)
- Note 47 was the least object in the right sub-tree
- Replace the least node in the right sub-tree with the deleting node guarantee that the tree will be still sorted.



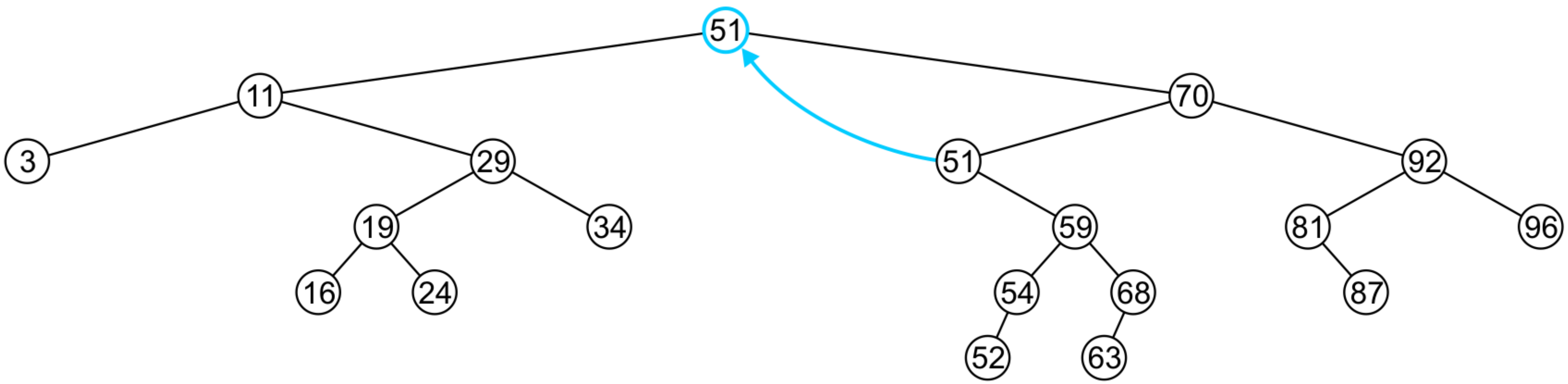
Deletion Operation: Case 3

- Suppose we want to delete the root again
- Delete (47)



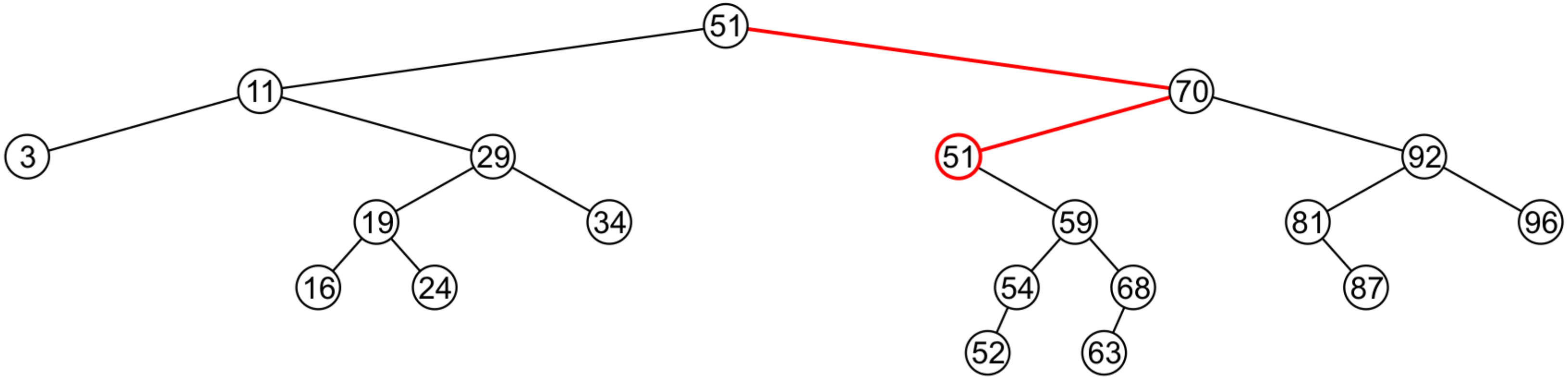
Deletion Operation: Case 3

- Suppose we want to delete the root again
- Delete (47)
  - We must replace the Node(47) with the minimum of the right sub-tree (Node(51))



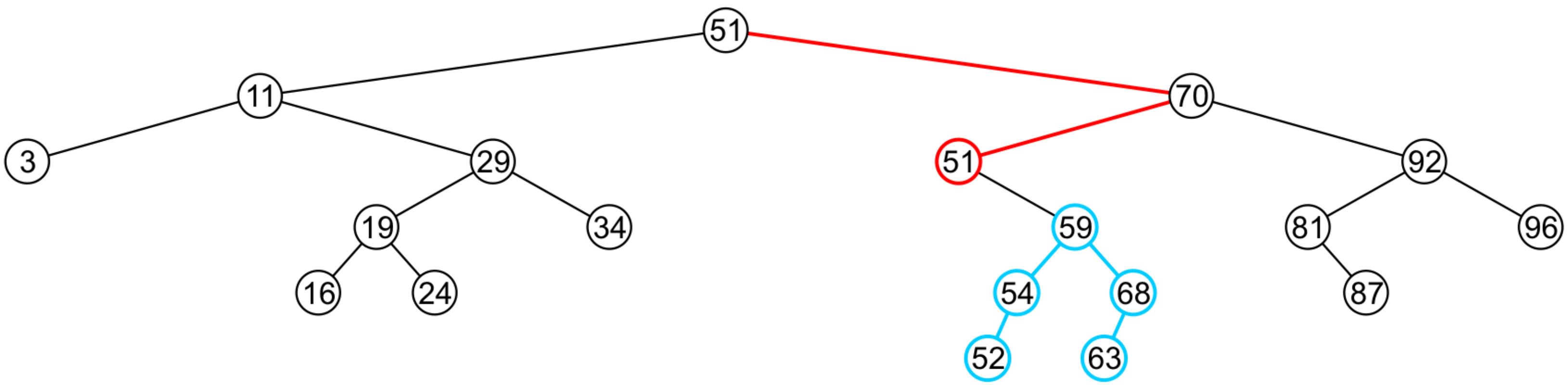
Deletion Operation: Case 3

- Suppose we want to delete the root again
- Delete (47)
  - We must proceed by deleting Node(51) from the right sub-tree



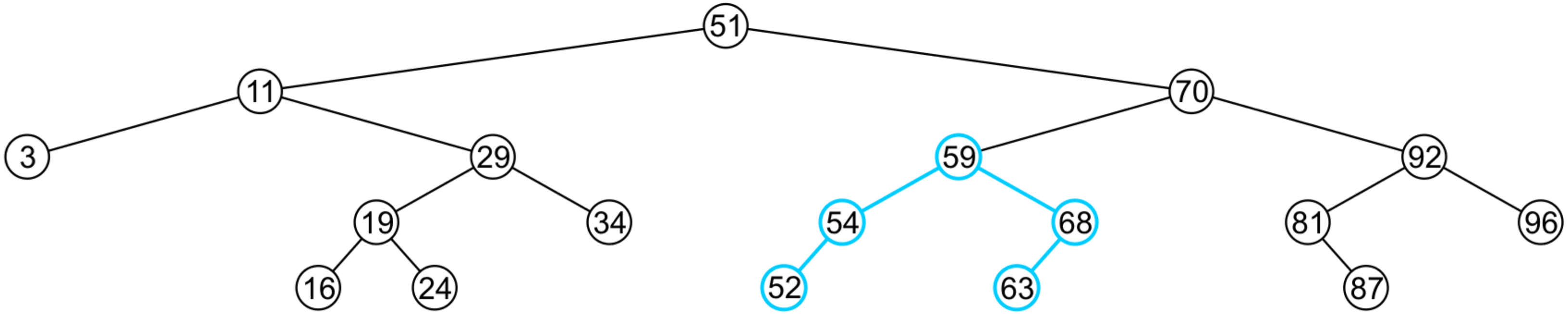
Deletion Operation: Case 3

- Suppose we want to delete the root again
- Delete (47)
  - Note that Node(51) has one child



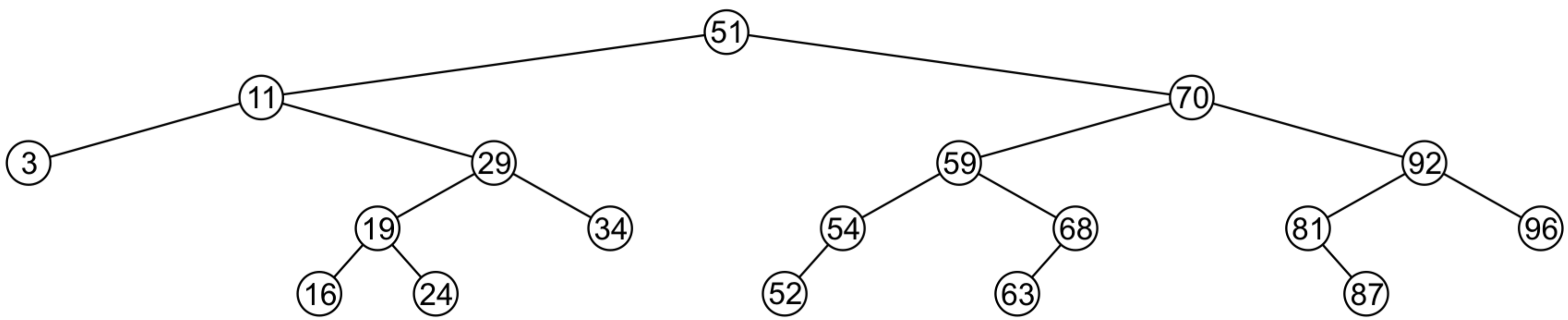
Deletion Operation: Case 3

- Suppose we want to delete the root again
- Delete (47)
  - Promote Node(51)'s child (Node(59))



Deletion Operation: Case 3

- Note that after seven removals, the remaining tree is still correctly sorted



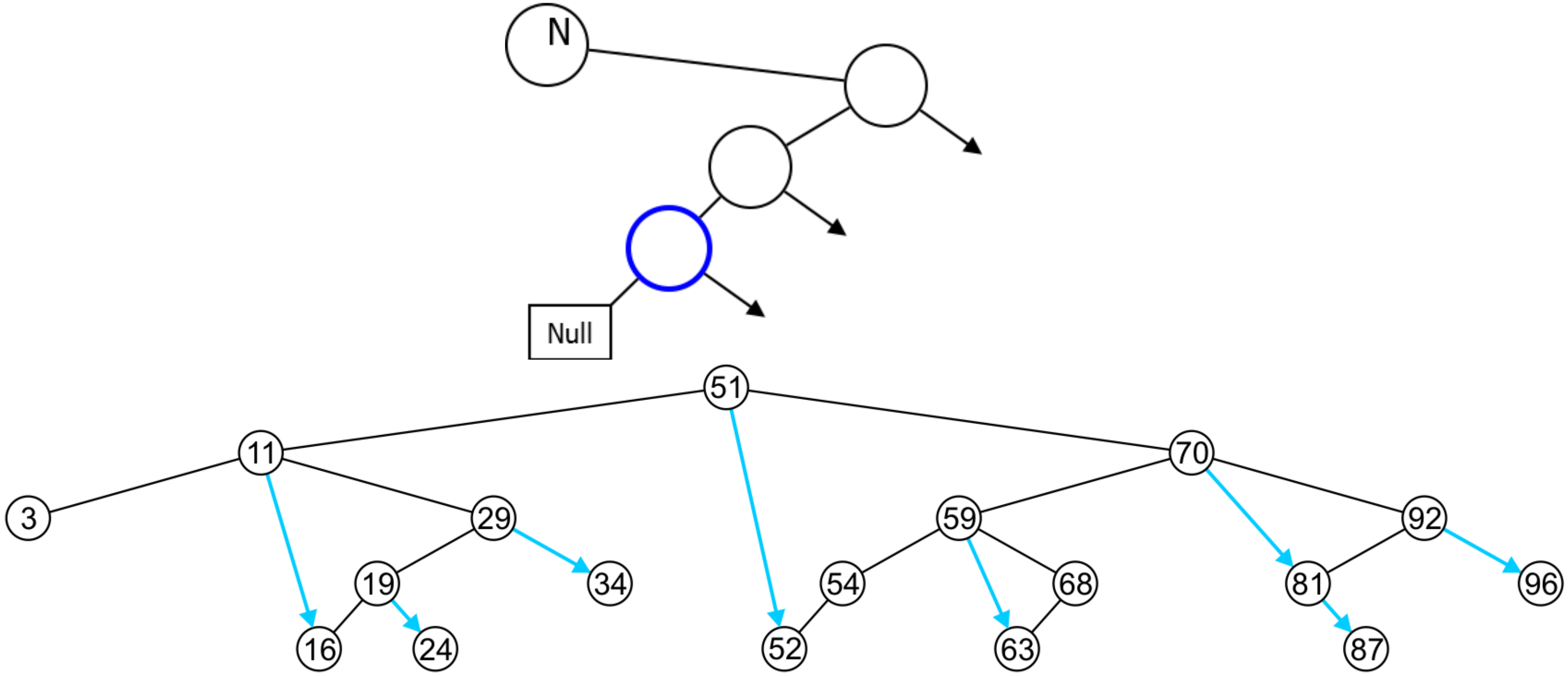
Demo: Delete Root Nodes

1. A tree with just one node
2. Root node with one left child
3. Root node with one right child
4. Root node with both children

FindNext (Case I)

To find the next largest object:

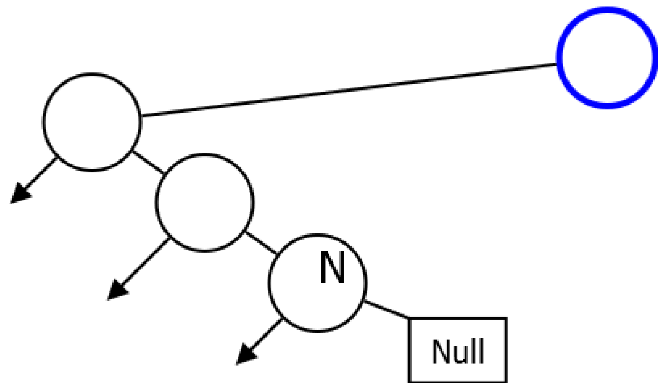
- There are two cases:
- Case I: If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object
- Just use FindMin(tree.right)

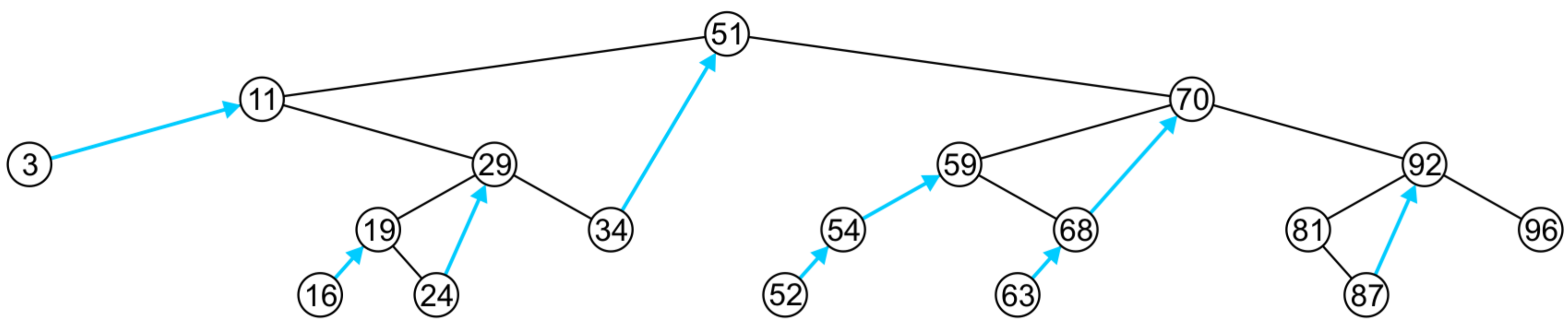


FindNext (Case II)

To find the next largest object:

- Case II: If there is no right sub-tree, the next largest object (if any) should be somewhere in the path from the node to the root
- Keep going up and check if node.key < node.parent





FindNext Algorithm

Node FindNext (Node node)

```
if node.right ≠ null
    return LeftDescendant ( node.right ) // Case I
else
    return RightAncestor(node) // Case II
```

Node LeftDescendant (Node node) // Case I

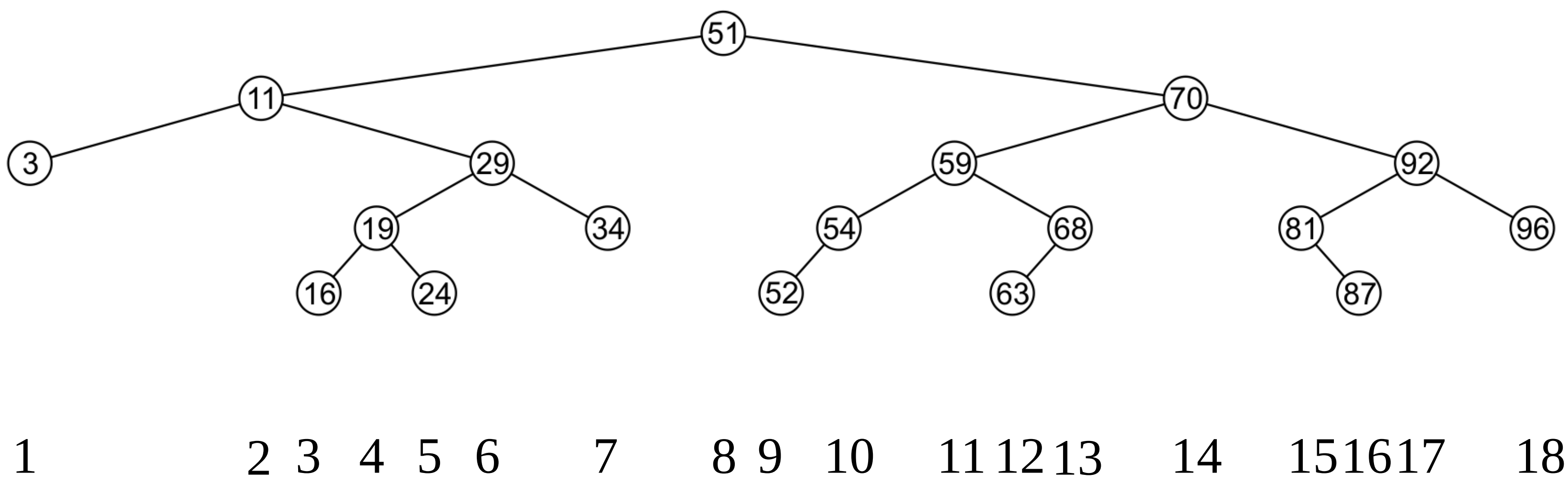
```
if node.left = null
    return node
else
    return LeftDescendant (node.left )
```

Node RightAncestor (Node node) // Case II

```
if node.key < node.parent.key
    return node.parent
else
    return RightAncestor (node.parent )
```

Find the k^{th} (smallest) node

- **Algorithm**
  - **In-Order Traversal**
  - **Return k-th element of the array (start with index 1)**
  - **Complexity:**
    - **O(n)**
  - **Need complexity of O(log n)**
    - **Require recursive implementation**

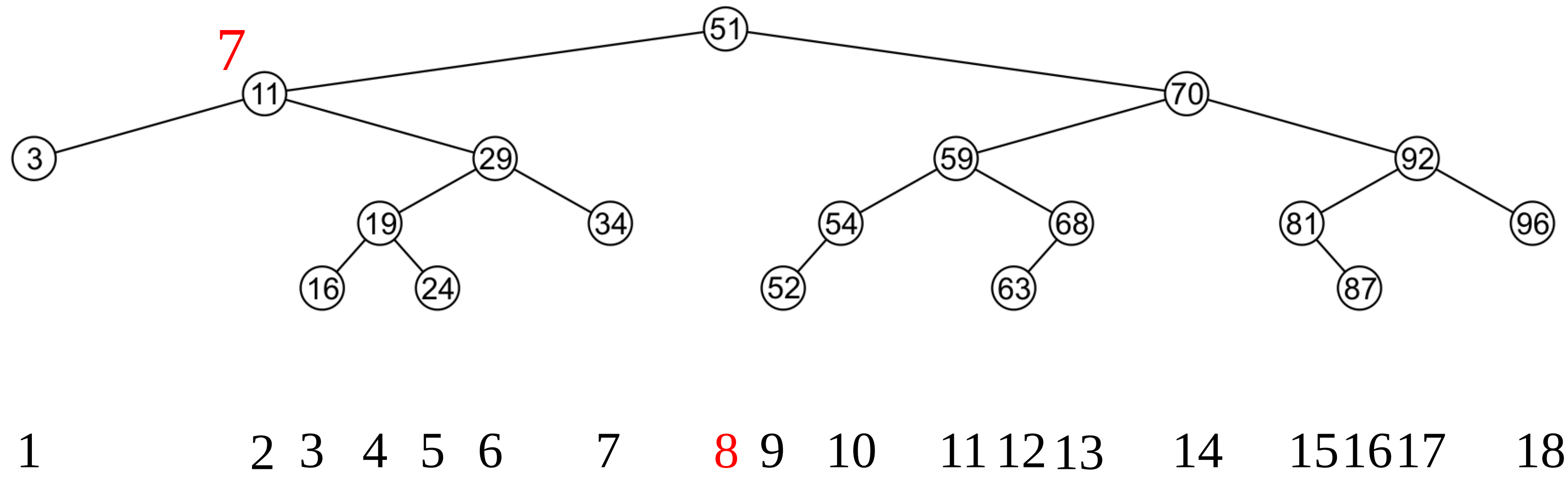


Find the k^th (smallest) node

- **Algorithm**
  - **Assume the left sub-tree of the node has a size l**
  - **If k = l + 1, return the root node**
  - **If k < l + 1, return the k-th node of the left sub-tree**
  - **If k > l + 1, return the (k – l – 1)-th node of the right sub-tree**

Find 8^th node (k=8)?

- Start from root
- Calculate size of the left sub-tree (l) -> 7
- k = l + 1?
- return Node(51)

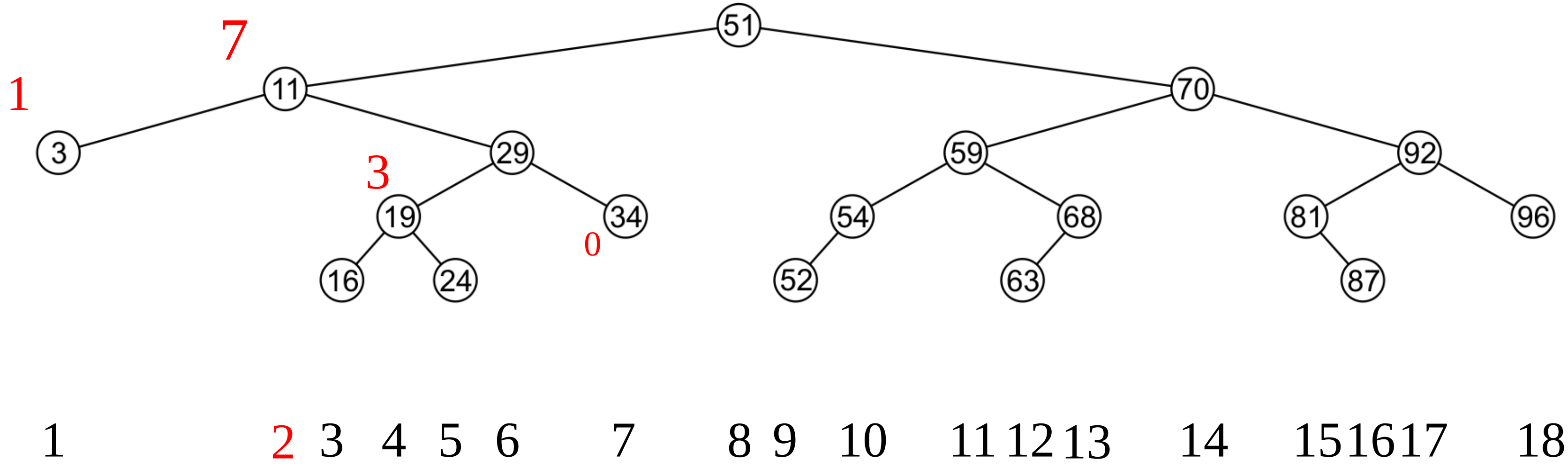


Find the k^{th} (smallest) node

- **Algorithm**
  - **Assume the left sub-tree of the node has a size l**
  - **If k = l + 1, return the root node**
  - **If k < l + 1, Find the k-th node of the left sub-tree**
  - **If k > l + 1, Find the (k – l – 1)-th node of the right sub-tree**

Find 7th node (k=7)?

- Start from root
- l = 7; k<l+1? Go left with k = 7
- l = 1; k>l+1? Go right with k = 5
- l = 3; k>l+1? Go right with k = 1
- l = 0; k=l+1? Return Node(34)

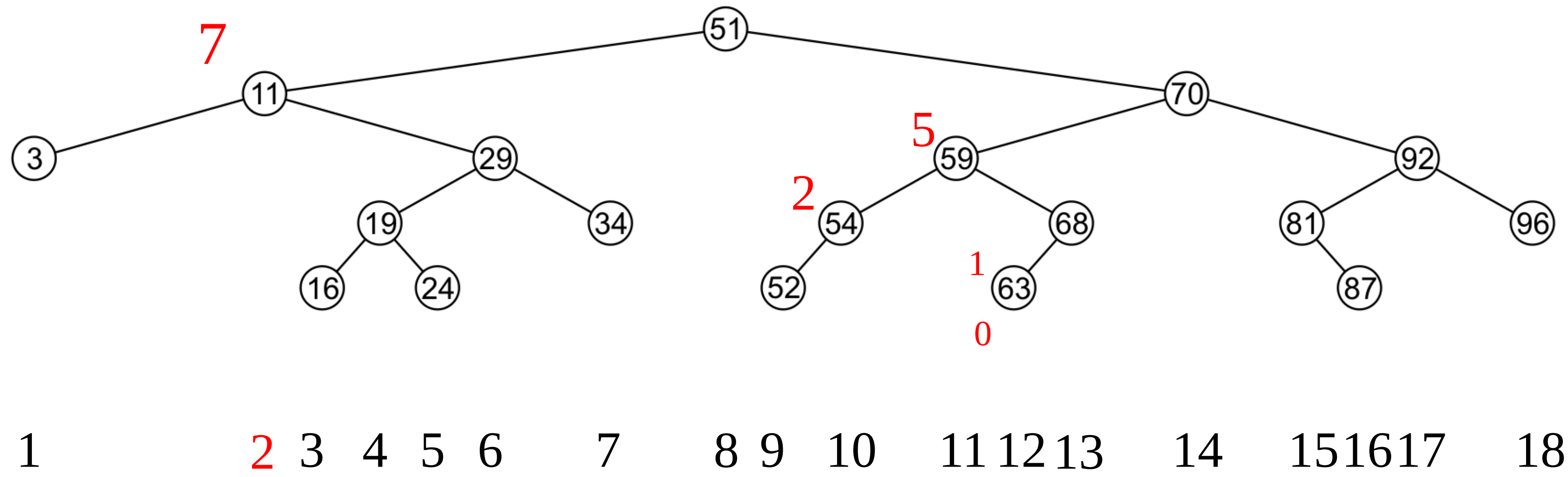


Find the k^{th} (smallest) node

- **Algorithm**
  - **Assume the left sub-tree of the node has a size l**
  - **If k = l + 1, return the root node**
  - **If k < l + 1, Find the k-th node of the left sub-tree**
  - **If k > l + 1, Find the (k – l – 1)-th node of the right sub-tree**

Find 12^{th} node (k=12)?

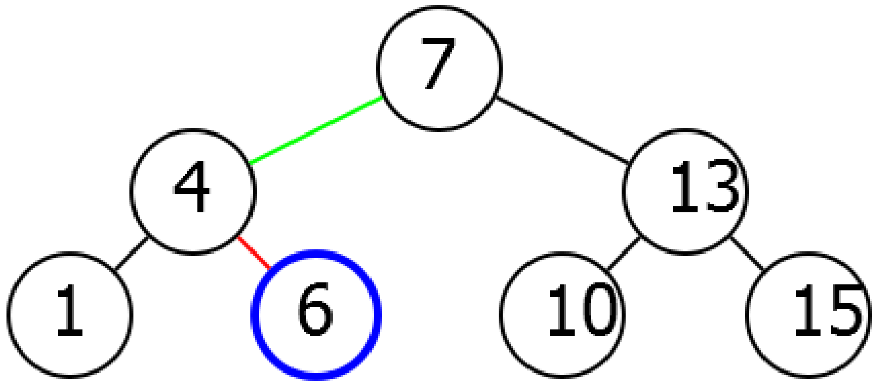
- Start from root
- l = 7; k>l+1? Go right with k = 4
- l = 5; k<l+1? Go left with k = 4
- l = 2; k>l+1? Go right with k = 1
- l = 1; k<l+1? Go left with k = 1
- l = 0; k=l+1? return Node(63)



RangeSearch

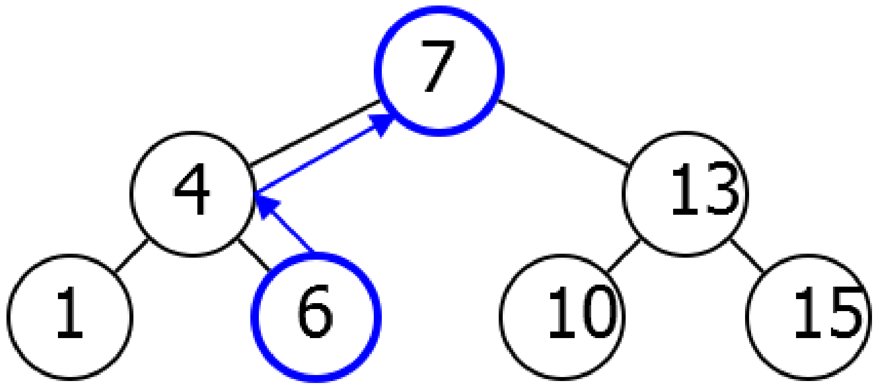
Homework or Exam?

RangeSearch(5, 12)



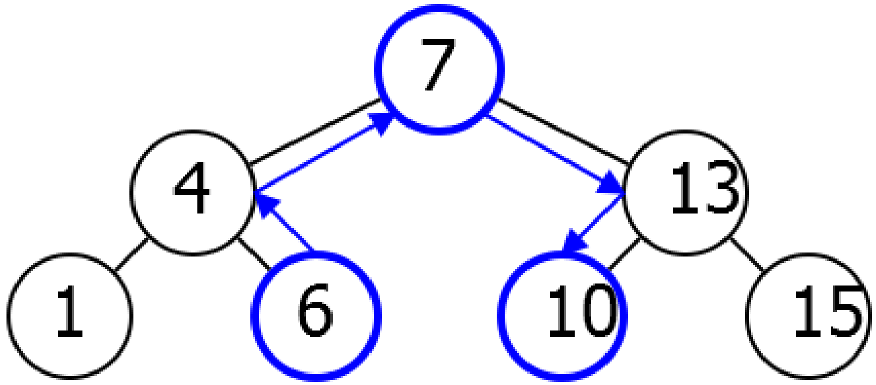
RangeSearch

RangeSearch(5, 12)



RangeSearch

RangeSearch(5, 12)



RangeSearch Implementation

RangeSearch(x, y, root) Pseudocode

```
List L = new List();
Node N = FindClosest(x, root)
while N.key <= y
    if N.key >= x
        L.Append( N)
    N = Next(N)
Return L
```