UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Background

- Variety of data structures
  - Stacks (LIFO)
    → All operations: O(1)
  - Queues (FIFO)
    → Dequeue: O(n)
  - Linked Lists
    → Removing tail element: O(n)
  - Doubly-linked Lists
    → All operations: O(1)
  - Hash Tables
    → Locating bucket (O(1)), locating elements in bucket (O(n))

Background

All the data structures we have studied so far are LINEAR

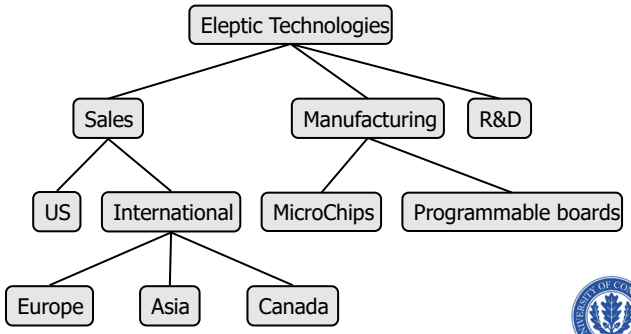
Breakthroughs come by thinking **nonlinearly!**

Module 9

Trees

Trees

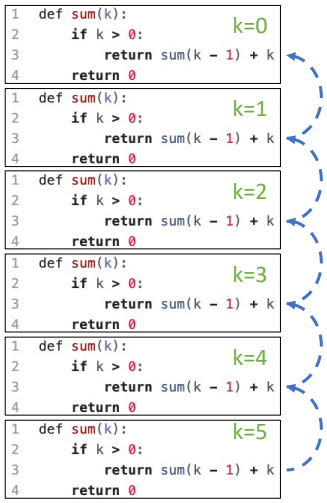
- In CS, Tree is an abstract data type that stores elements hierarchically.
- Non-linear data structure, relationship between objects/data are:
  → NOT defined as "before and after" relationship
  → Defined in hierarchical fashion - "above and below" relationship
- Examples:
  - Organizational Charts
  - File systems, etc
- Each element in a tree has a **parent** element and zero/more **children** elements, except the first/top element which is called **root**



Quick Recap of Recursion

Example: Adding K integers.

```
1  def sum(k):
2      if k > 0:
3          return sum(k - 1) + k
4      return 0
5  print(sum(5))
```



Quick Recap of Recursion

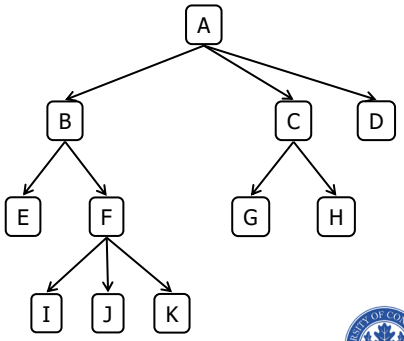

```
1  def sum(k):
2      if k > 0:
3          return sum(k - 1) + k
4      return 0
5  print(   15   )
```

```
def sum(k):
    if k > 0:
        return sum(k - 1) + k
    return 0
```

k=0 → 0 → k=1 → 1 → k=2 → 3 → k=3 → 6 → k=4 → 10 → k=5

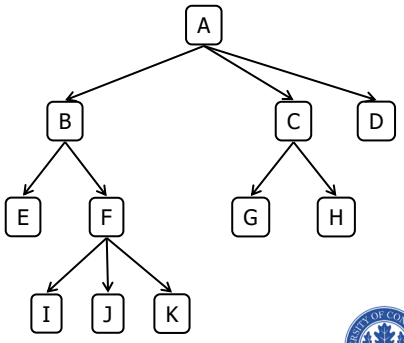
Trees Terminologies

- **Root**: Highest-most node without parent (A)
- **Edge:** Connection between two nodes to show a relationship between them
- **Path:** A path is an ordered list of nodes that are connected by edges (from top to bottom)
- **Parent:** A node is a parent of all nodes it connects to with outgoing edges
- **Children:** The set of nodes which have incoming edges from a parent node
- **Sibling:** Nodes that are children of the same parent



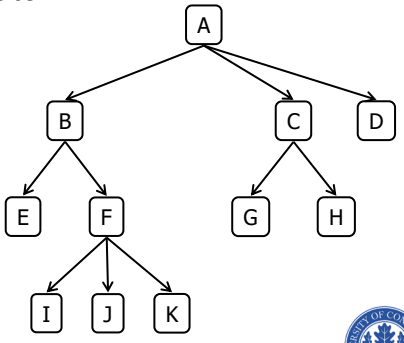
Trees Terminologies

- **Descendant** of node (x)**:** all nodes for which there is path from x. (child, grandchild, grand-grandchild)
- **Ancestors** of node (x)**:** all nodes which x is a descendant of (parent, grandparent, grand-grandparent)
- **Leaf Node:** Nodes which have no children (J, K, etc)
- **Subtree:** Set of nodes and edges comprised of a parent and all descendants of that parent (C-G-H)



Trees Terminologies

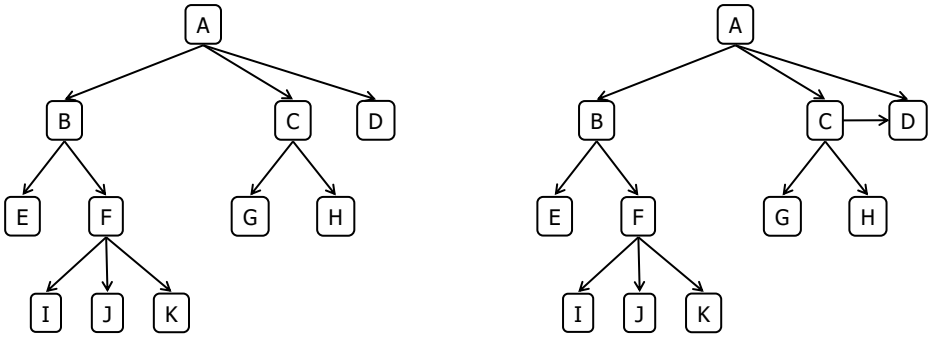
- **Height** of a node x**:** number of edges in longest path from the node x to a leaf
  → Height of a tree in this example is 3 (from root to leaf)
  → Height of B is 2 (from B to leaf)
- **Depth/Level** of node x**:** Number of edges in path from root to x
  → Depth of B is 1
  → Depth of J is 3
- **Degree** of a node**:** The number of its children
- **Degree** of a tree**:** The number of nodes in it



Trees Terminologies

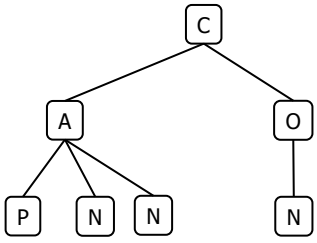
- In a tree, there must only be a single path from the root node to any other node

This is not a tree



Trees as Lists

- Tree can be defined recursively as a root with zero or more children
  - Each children can be a subtree
- As same as we conceptualized other data structures with Lists, we can do that for Trees as well
- T = [‘C’,
  [‘A’,
  [‘P’, ‘N’, ‘T’]
  ] ,
  [‘O’,
  [‘N’]
  ]
  ]



Activity

Fill the table with corresponding values

```
T = ['C',
        ['A',
            ['P', 'N', 'T']
        ],
        ['O',
            ['Z']
        ]
    ]
```

```
def printtree(T):
    print(T[0])
    for child in range(1, len(T)):
        printtree(T[child])
```

printtree(t) calls	T[0]	T[child]

Activity Solution

Fill the table with corresponding values

```
T = ['C',
        ['A',
            ['P', 'N', 'T']
        ],
        ['O',
            ['Z']
        ]
    ]
```

```
def printtree(T):
    print(T[0])
    for child in range(1, len(T)):
        printtree(T[child])
```

printtree(t) calls	T[0]	T[child]
1	C	['A', ['P', 'N', 'T']]
2	A	['P', 'N', 'T']
3	P	N
4	N	T
5	T	['O', ['Z']]
6	O	['Z']
7	Z	-

Trees ADT

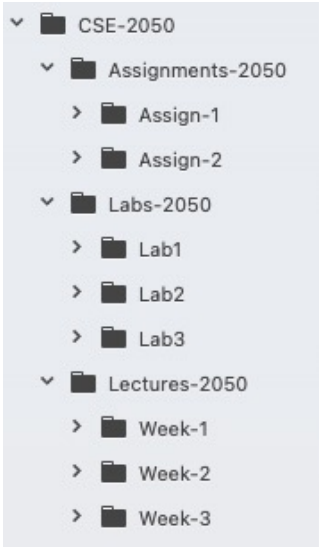
Very basic methods are:

__init__ (L): Initializes a new tree with data, children (list) and parent.

add_child: add a child to existing node

get_level: get a level of the current node

__str__ (): Return a string representing the entire tree.



Trees Implementation

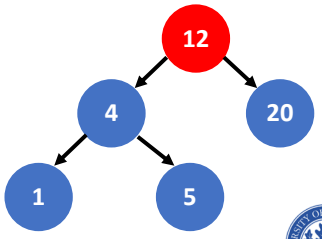
```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None
```

```
def add_child(self, child):
    child.parent = self
    self.children.append(child)
```

```
def get_level(self):
    level = 0
    node_parent = self.parent
    while node_parent != None:
        level += 1
        node_parent = node_parent.parent
    return level
```

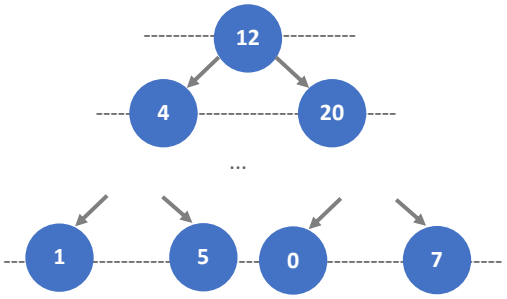
Binary Search Trees

- Binary search trees are data structures to organize data efficiently
- Every node in the tree can have at most 2 children (left child and right child)
  - left child is smaller than the parent node
  - right child is greater than the parent node
- The new data is placed in sorted order so that the search and other operations can use the principle of binary search with O(logn) running time
- We can access the root node exclusively as same as we can access head node of linked list
- All other nodes can be accessed via the root node



Binary Search Trees

- Number of nodes in a complete binary tree with height h



level 0 with 2^0 nodes

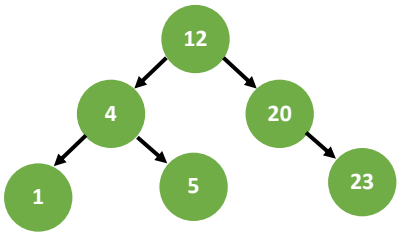
level 1 with 2^1 nodes

level h with 2^h nodes

$2^h = N$
$\log_2 2^h = \log_2 N$
$h = \log_2 N$
$h = O(\log N)$

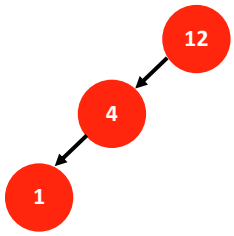
Binary Search Trees

- Balanced and Imbalanced BST



BALANCED TREE

- *Left and right nodes contain approximately same number of children*
- *operations are **O(logN)** always*



IMBALANCED TREE

- *Number of children on both sides differ significantly*
- ***O(N)** linear running time complexity*

Binary Search Trees Operations

Insert

Search

min

max

Delete

Traverse

Binary Search Trees | Operations | Insert

Insert(12)

Binary Search Trees | Operations | Insert

Insert(4)

Binary Search Trees — Operations — Insert

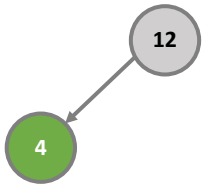
Insert(4)

Is new element smaller/greater than 12?

Binary Search Trees — Operations — Insert

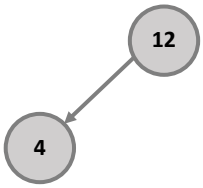
Insert(4)

Smaller → place it on the left.



Binary Search Trees — Operations — Insert

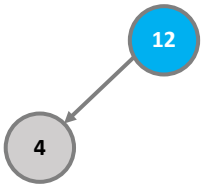
Insert(8)



Binary Search Trees — Operations — Insert

Insert(8)

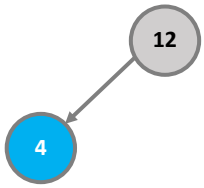
Is new element smaller/greater than 12?



Binary Search Trees — Operations — Insert

Insert(8)

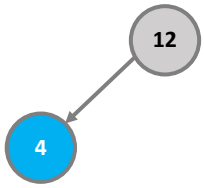
Smaller → check the next node on left.



Binary Search Trees — Operations — Insert

Insert(8)

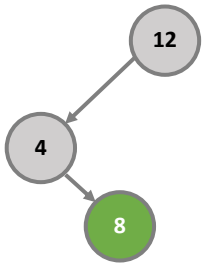
Is new element smaller/greater than 4?



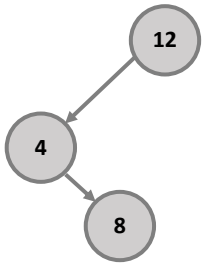
Binary Search Trees — Operations — Insert

Insert(8)

Greater → place it on the right.

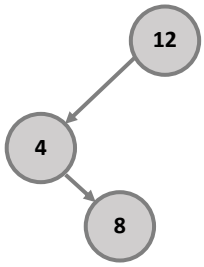


Binary Search Trees — Operations — Insert



Binary Search Trees — Operations — Insert

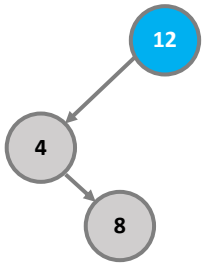
Insert(20)



Binary Search Trees — Operations — Insert

Insert(20)

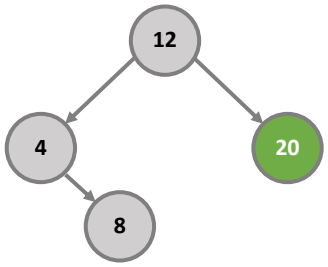
Is new element smaller/greater than 12?



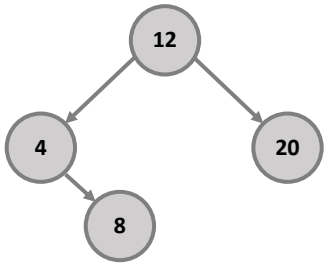
Binary Search Trees — Operations — Insert

Insert(20)

Greater → place it on the right.

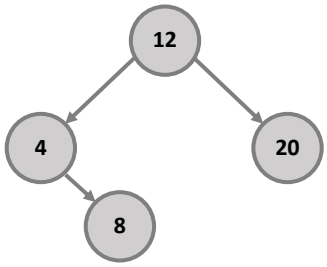


Binary Search Trees — Operations — Insert



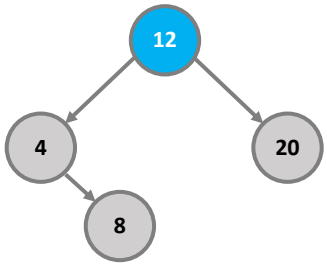
Binary Search Trees | Operations | Insert

Insert(27)



Binary Search Trees | Operations | Insert

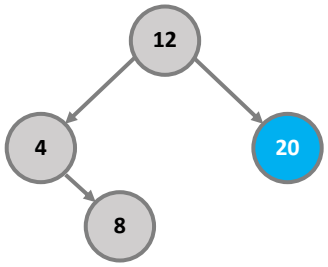
Insert(27)



Binary Search Trees — Operations

Insert

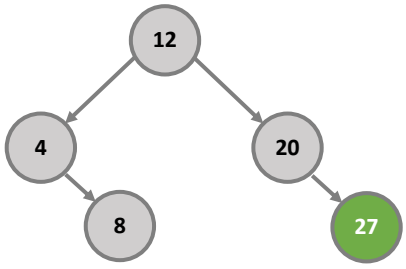
Insert(27)



Binary Search Trees — Operations

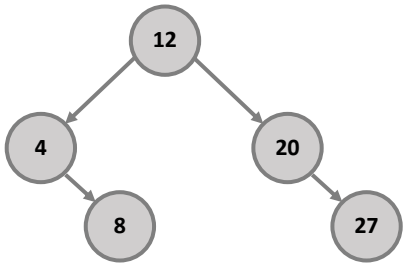
Insert

Insert(27)



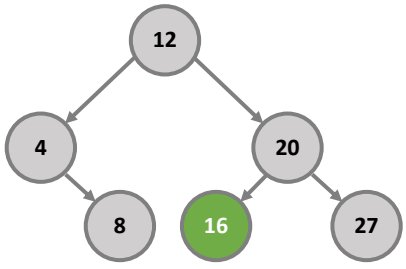
Binary Search Trees | Operations | Insert

Insert(16)



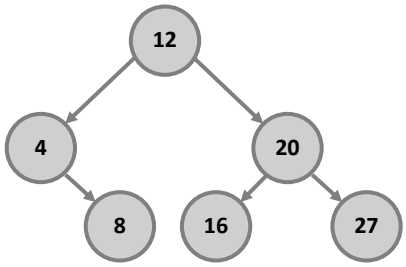
Binary Search Trees | Operations | Insert

Insert(16)



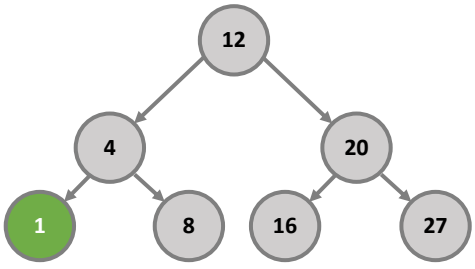
Binary Search Trees | Operations | Insert

Insert(1)

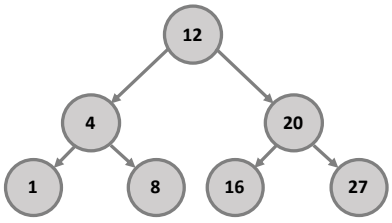


Binary Search Trees | Operations | Insert

Insert(1)



Binary Search Trees | Operations | Insert



Implementation

Create a Node class encapsulating the following information:

- Actual data
- Parent node
- Left child
- Right child

```
class BSTNode:
    def __init__(self, data, parent = None):
        self.data = data
        self.parent = parent
        self.left = None
        self.right = None
```


UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

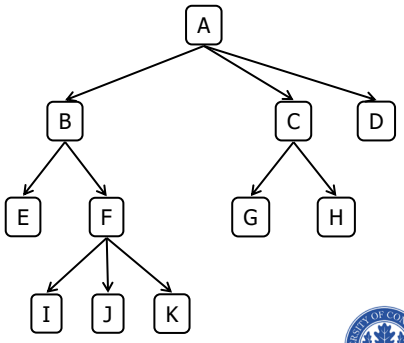
(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Quick Recap

- Overview of data structures studied so far
- Trees
  - Non linear data structure
- Terminologies
  - Root
  - Parent
  - Child
  - Leaf node
  - Depth/level
    - Number of edges from root to x
  - Trees representation as a list



Quick Recap

- Trees representation
  - As a list

```
T = ['C',
        ['A',
            ['P', 'N', 'T']
        ],
        ['O',
            ['Z']
        ]
    ]
```

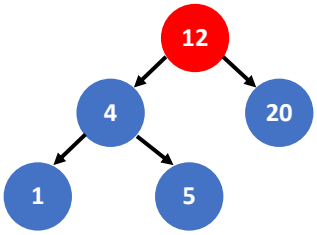
  - As a class

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None
```

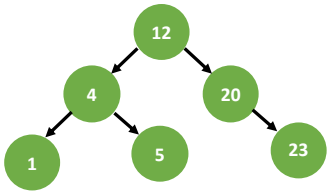
```
def add_child(self, child):
    child.parent = self
    self.children.append(child)
```

Quick Recap

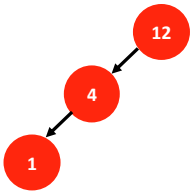
- Binary Search Trees →
  - Each node can have at most two child
  - Left child is smaller than the parent
  - Right child is greater than the parent



- Balanced and Imbalanced Trees



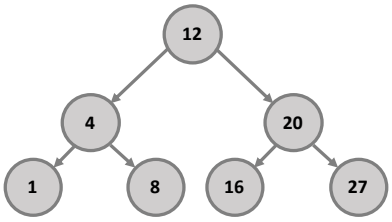
BALANCED TREE



IMBALANCED TREE

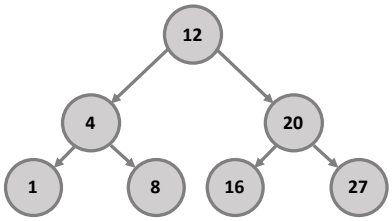
Quick Recap

- BST Operations
  - Insert, Search, Delete, Traverse




```
class BSTNode:
    def __init__(self, data, parent = None):
        self.data = data
        self.parent = parent
        self.left = None
        self.right = None
```

Binary Search Trees | Operations | Insert



Implementation

Create a BinarySearchTree class

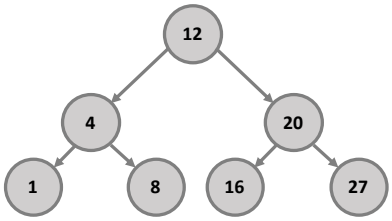
1. Initialize the root node with None

insert() method

1. Add root node, if it does not exist
2. Check if the new element is greater/smaller than the current parent.
3. If left/right child does not exist, add the new node on appropriate side
4. If the child node exist, recursively execute step 2 and 3.

Note: Assume that the duplicate elements are not allowed

Binary Search Trees | Operations | Insert



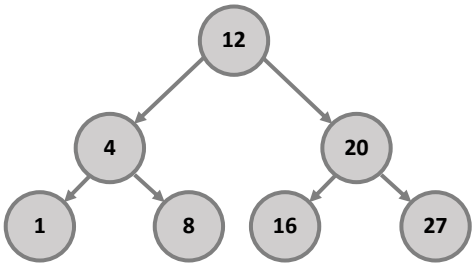
Implementation

```
class BinarySearchTree:
    def __init__(self):
        self.root = None
```

```
    def insert(self, data):
        if self.root is None:
            #if there is no root node
            self.root = BSTNode(data)
        else:
            #if root node exist, then add child at right location
            self._add_child(data, self.root)
```

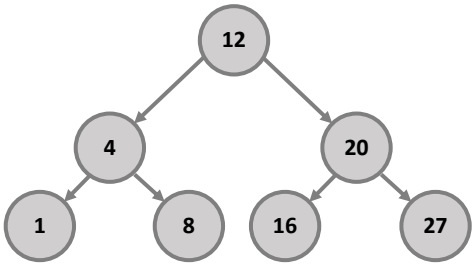
```
def _add_child(self, data, p_node):
    if data == p_node.data: #duplicate items cannot be added
        return

    if data < p_node.data: #the element is smaller than the root
        if p_node.left: #left child exist
            self._add_child(data, p_node.left)

        else: # left child does not exist
            p_node.left = BSTNode(data, p_node)

    else: #the element is greater than the root
        if p_node.right: #right child exists
            self._add_child(data, p_node.right)

        else: #right child does not exist
            p_node.right = BSTNode(data, p_node)
```


Binary Search Trees
Operations
Search



Binary Search Trees
Operations
Search

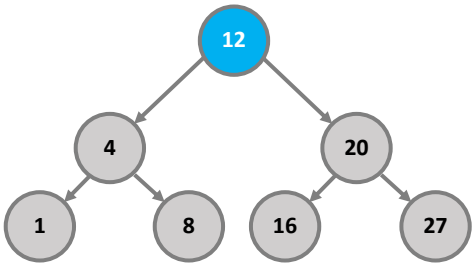
Search(8)



Binary Search Trees | Operations | Search

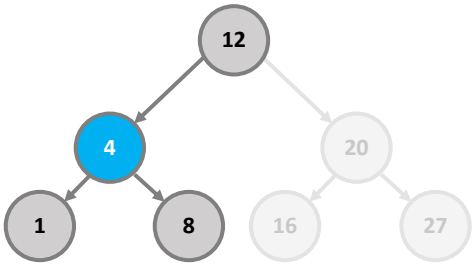
Search(8)

Is the element smaller/greater than the root?



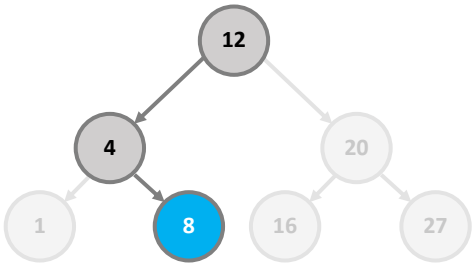
Binary Search Trees | Operations | Search

Search(8)



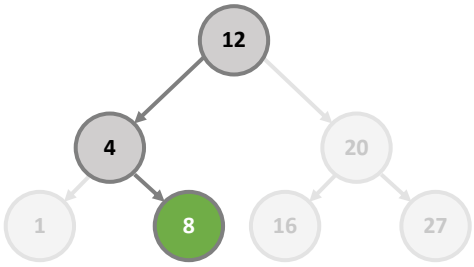
Binary Search Trees | Operations | Search

Search(8)

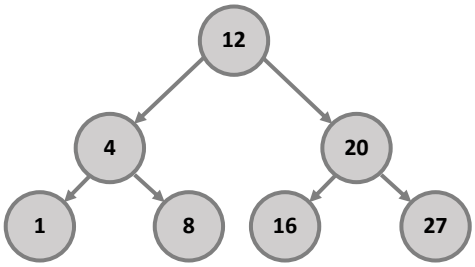


Binary Search Trees | Operations | Search

Search(8)

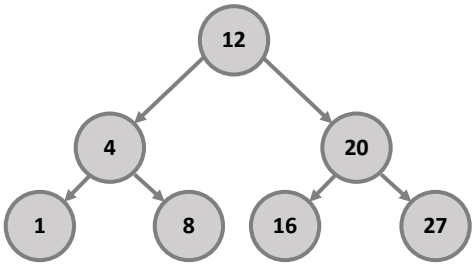


Binary Search Trees
Operations
Search



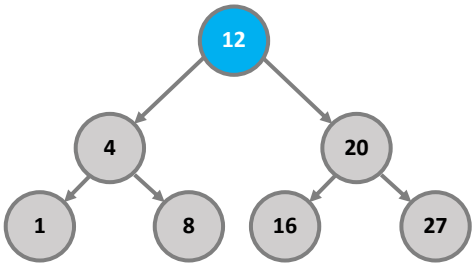
Binary Search Trees
Operations
Search

Search max()



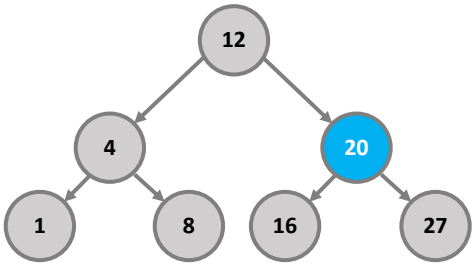
Binary Search Trees | Operations | Search

Search max()



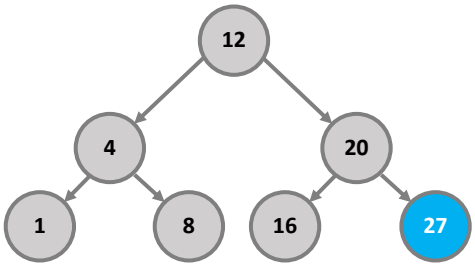
Binary Search Trees | Operations | Search

Search max()



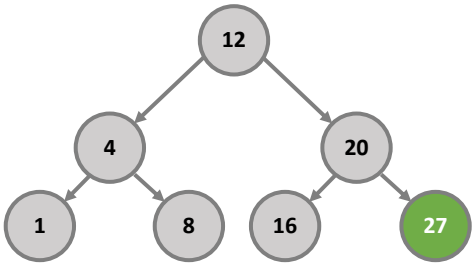
Binary Search Trees — Operations — Search

Search max()



Binary Search Trees — Operations — Search

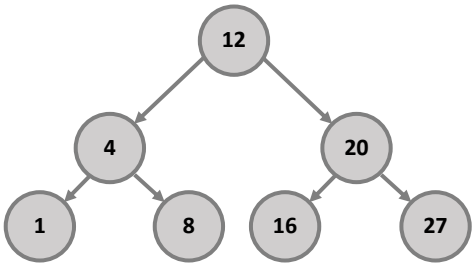
Search max()



the **maximum** item in the binary search tree is the **rightmost** item in the tree

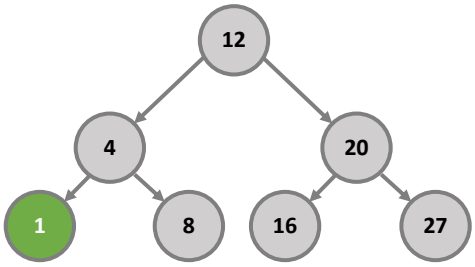
Binary Search Trees | Operations | Search

Search min()



Binary Search Trees | Operations | Search

Search min()



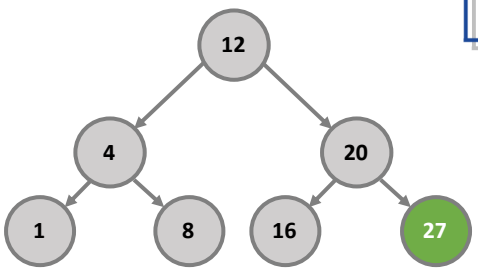
the **minimum** item in the binary search tree
is the **leftmost** item in the tree

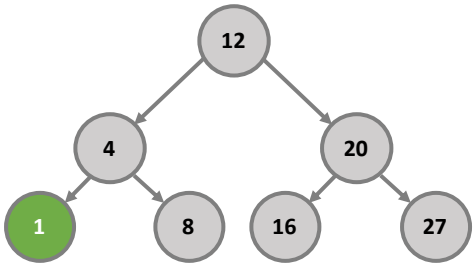
Binary Search Trees | Operations | Search

Implementation - get_max()

1. If root exist, proceed to step 2, else → None
2. For the current node, check if the right child exist
3. If right child does not exist, Return the current node data, else recurse step 2 again

Try it Yourself



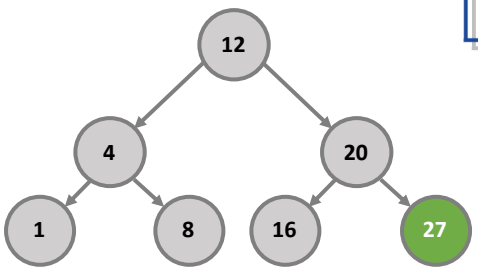


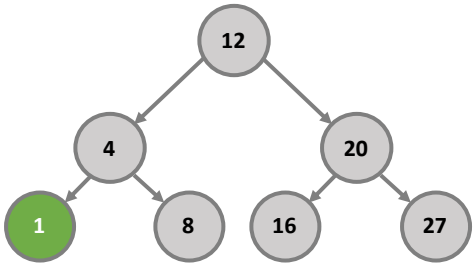
Binary Search Trees | Operations | Search

Implementation - get_max()

```
def get_max(self):
    if self.root:
        return self._get_right_child(self.root)
    else:
        return None
```

```
def _get_right_child(self, node):
    if node.right:
        return self._get_right_child(node.right)

    return node.data
```



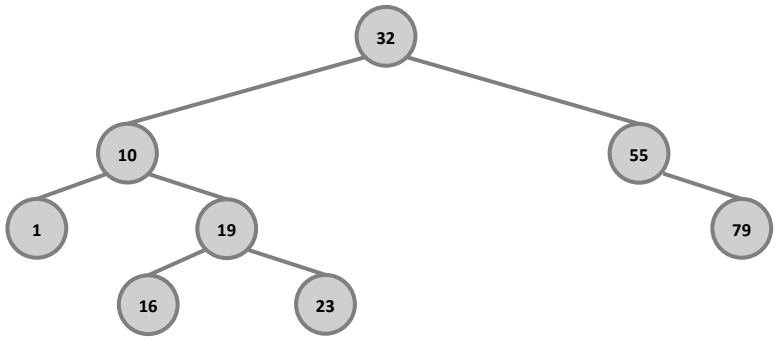


Binary Search Trees | Operations | Delete

1. Deleting a leaf node → node having no children
2. Deleting a node with 1 child
3. Deleting a node with 2 children

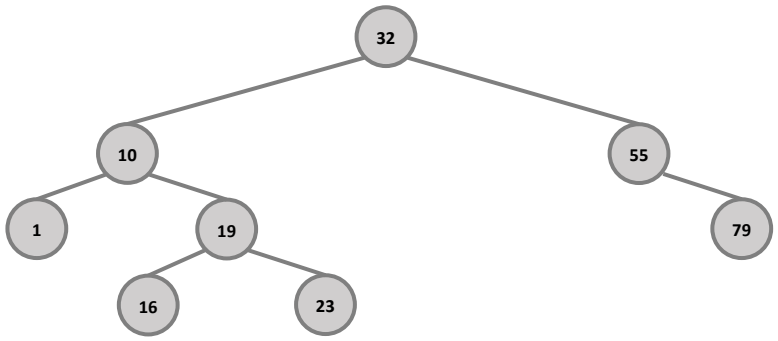
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → node having no children



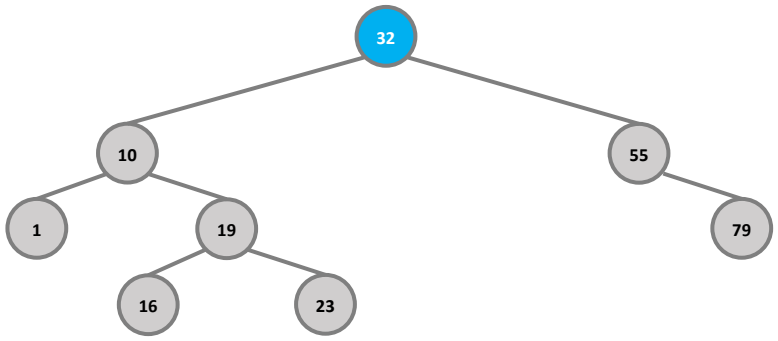
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → 23



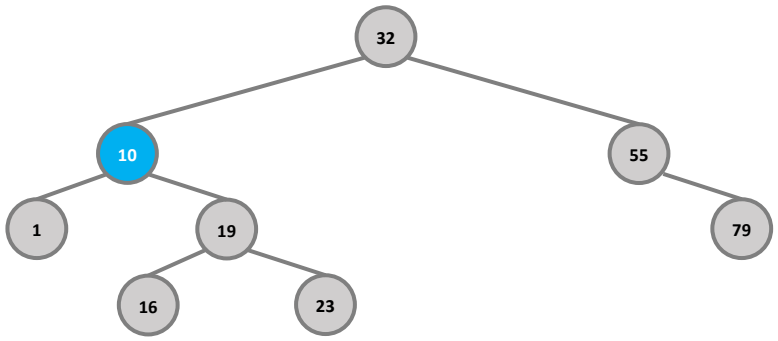
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → 23



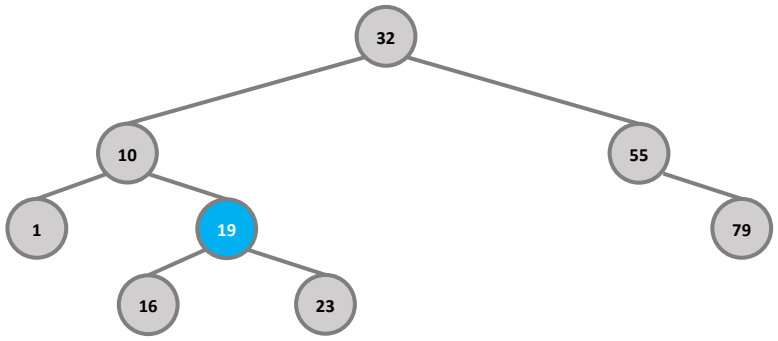
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → 23



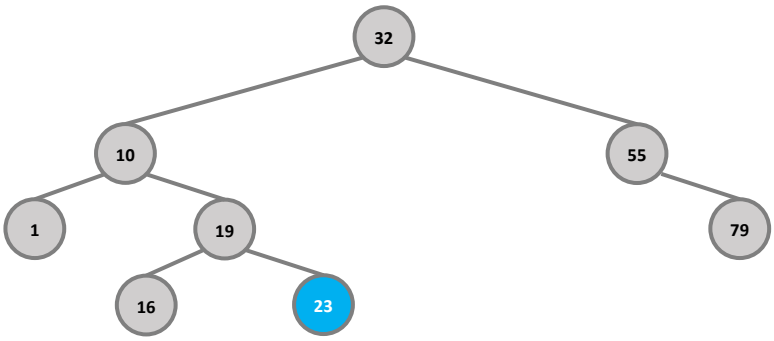
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → 23



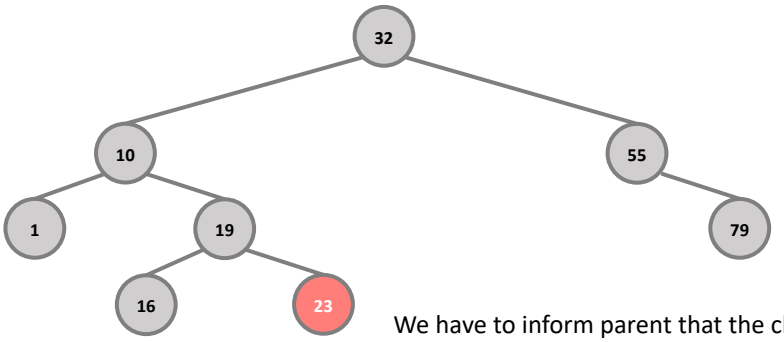
Binary Search Trees — Operations — Delete

1. Deleting a leaf node → 23



Binary Search Trees — Operations — Delete

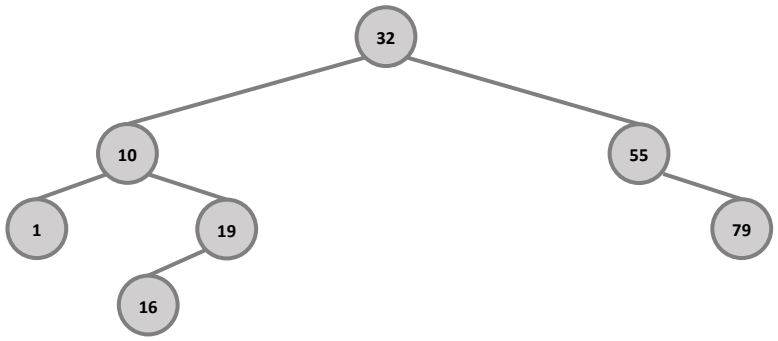
1. Deleting a leaf node → 23



We have to inform parent that the child has been removed.

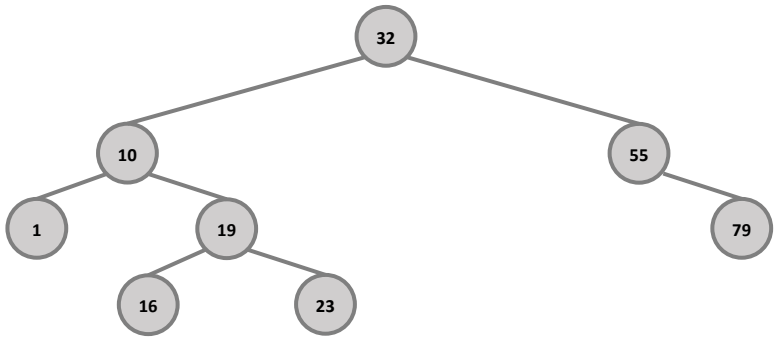
Binary Search Trees | Operations | Delete

1. Deleting a leaf node → 23



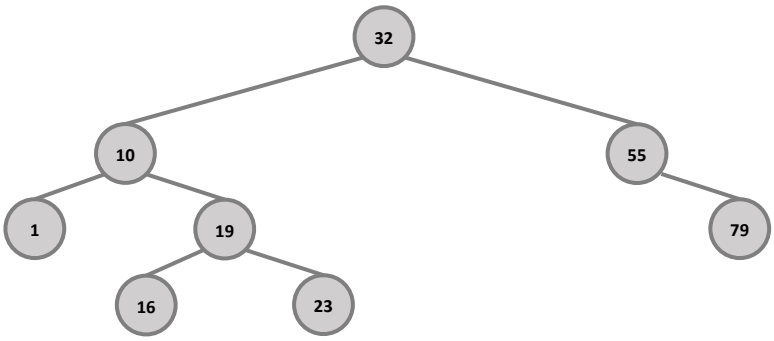
Binary Search Trees | Operations | Delete

2. Deleting a node with one child



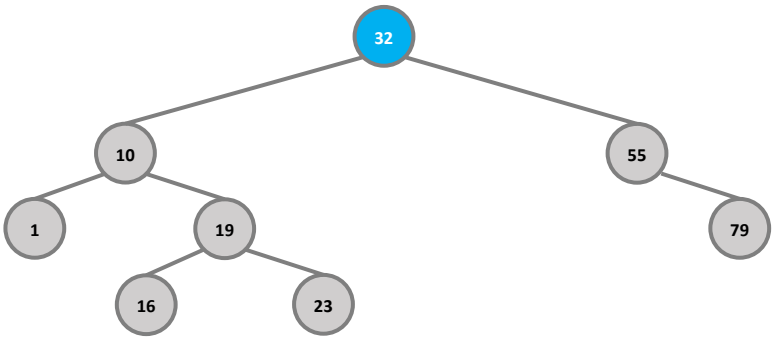
Binary Search Trees | Operations | Delete

2. Deleting a node with one child → 55



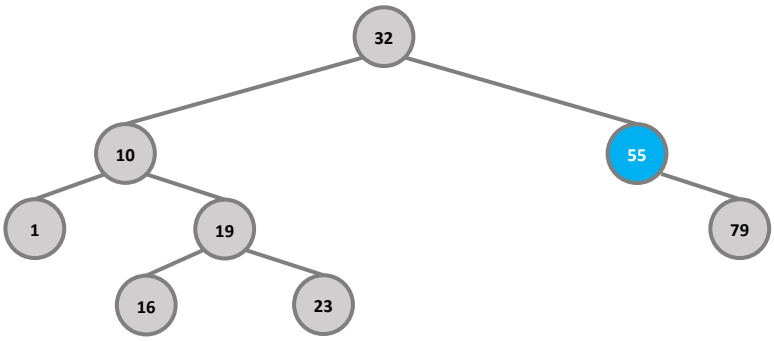
Binary Search Trees | Operations | Delete

2. Deleting a node with one child → 55



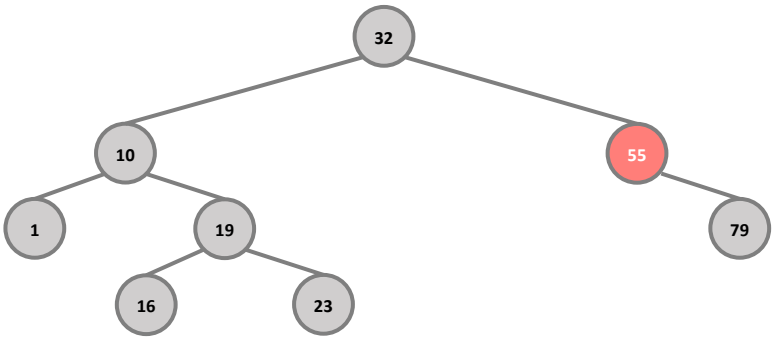
Binary Search Trees | Operations | Delete

2. Deleting a node with one child → 55



Binary Search Trees | Operations | Delete

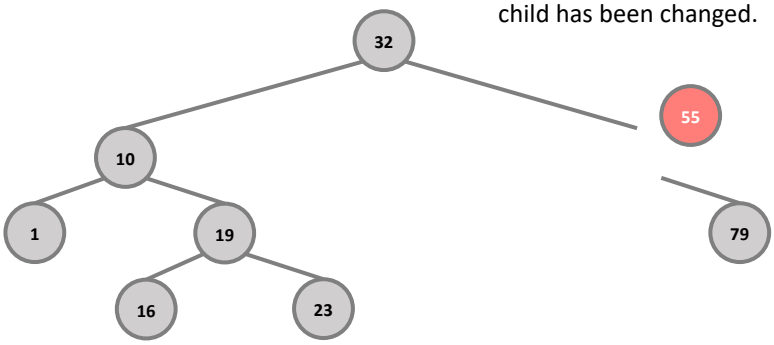
2. Deleting a node with one child → 55



Binary Search Trees | Operations | Delete

2. Deleting a node with one child → 55

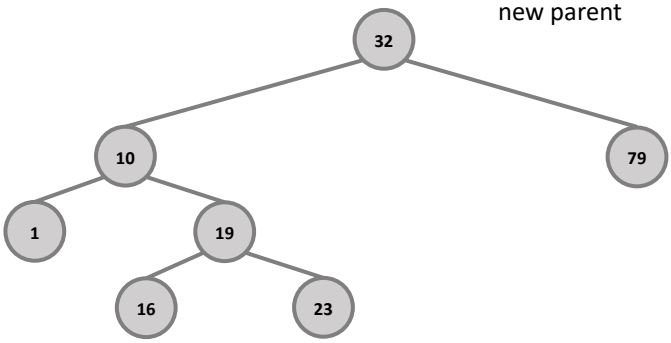
We have to inform parent that the right (or left) child has been changed.



Binary Search Trees | Operations | Delete

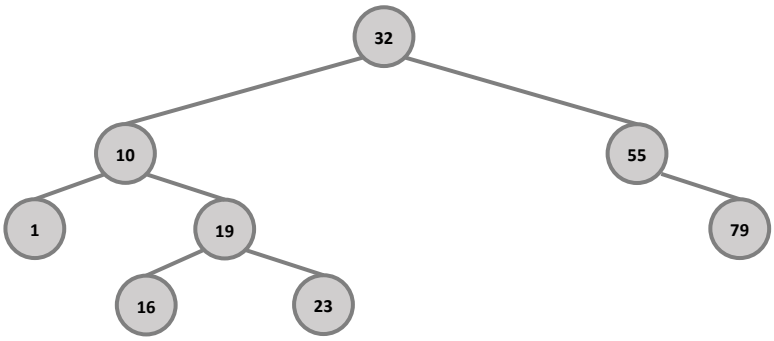
2. Deleting a node with one child → 55

We also have to update “the child” about its new parent



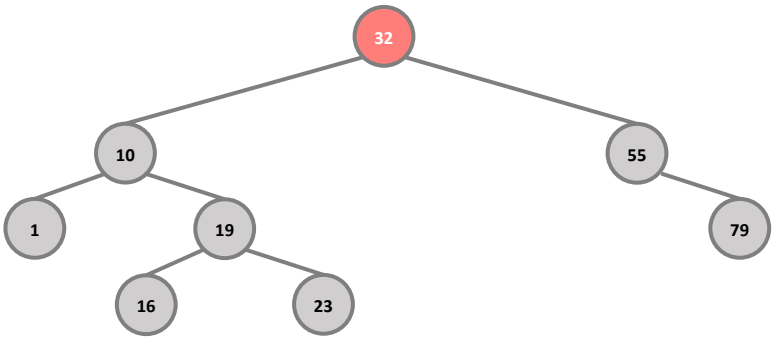
Binary Search Trees | Operations | Delete

3. Deleting a node with two child



Binary Search Trees | Operations | Delete

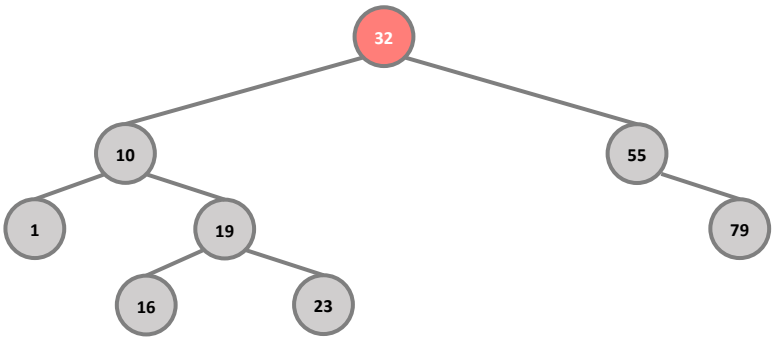
3. Deleting a node with two child → root node



Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

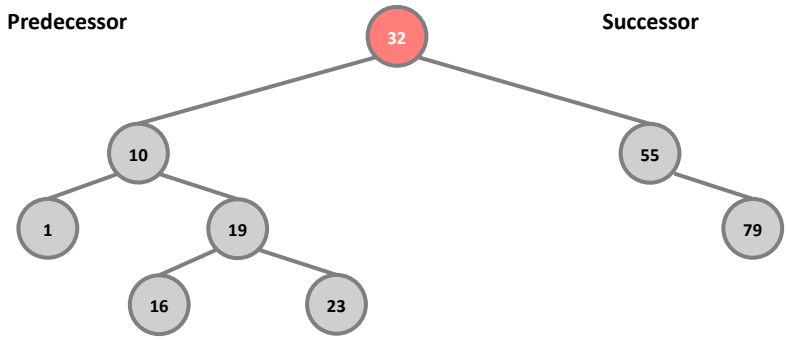
After deleting, what will be the new root?



Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

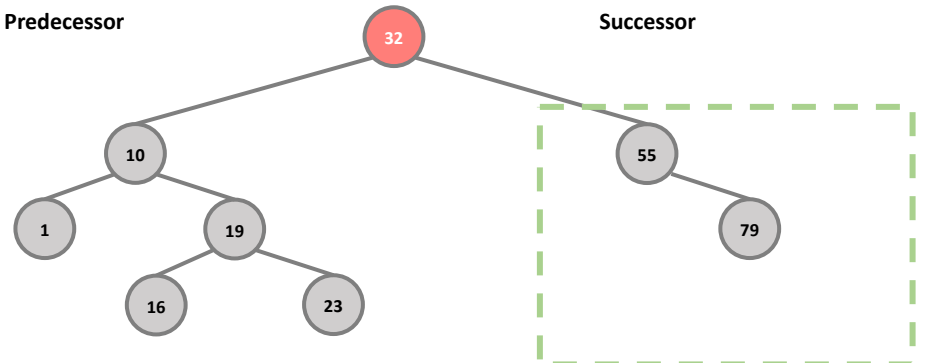
After deleting, what will be the new root?



Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

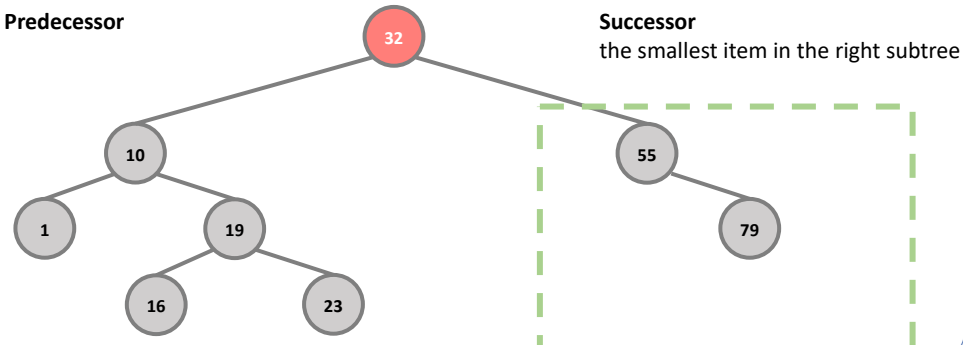
After deleting, what will be the new root?



Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

After deleting, what will be the new root?



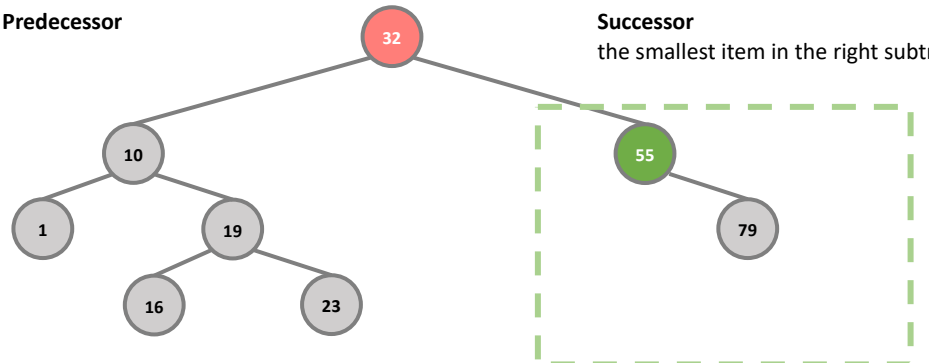
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

After deleting, what will be the new root?

Predecessor

Successor
the smallest item in the right subtree



CSE-2050 – Data Structures and Object-Oriented Design
Trees

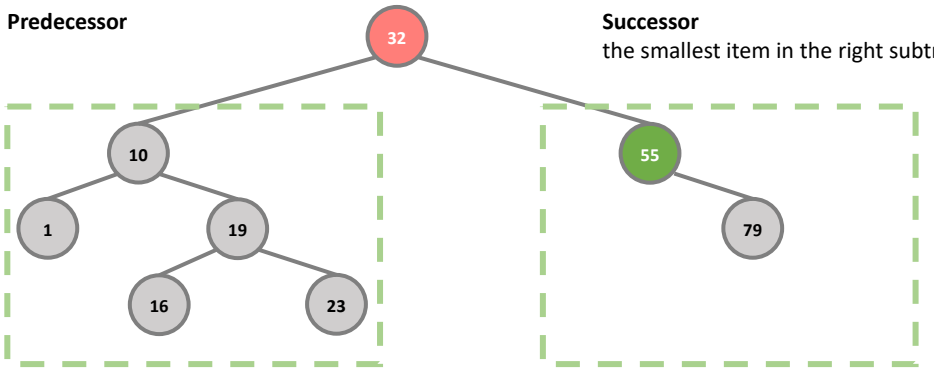
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

After deleting, what will be the new root?

Predecessor

Successor
the smallest item in the right subtree



Hasan Baig

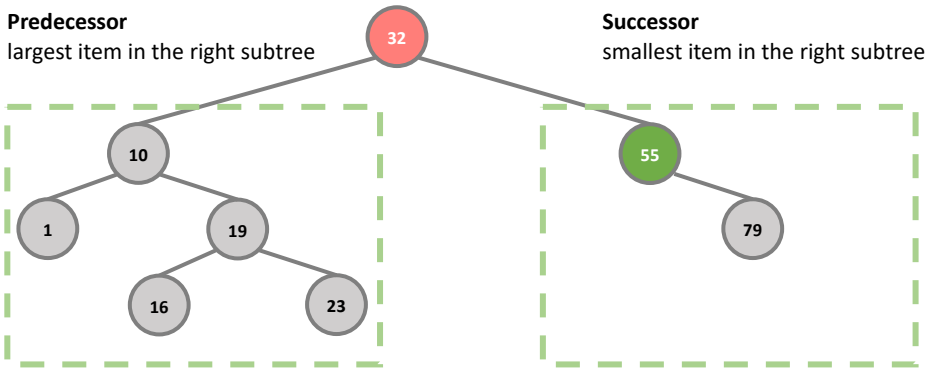
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

After deleting, what will be the new root?

Predecessor
largest item in the right subtree

Successor
smallest item in the right subtree



CSE-2050 – Data Structures and Object-Oriented Design
Trees

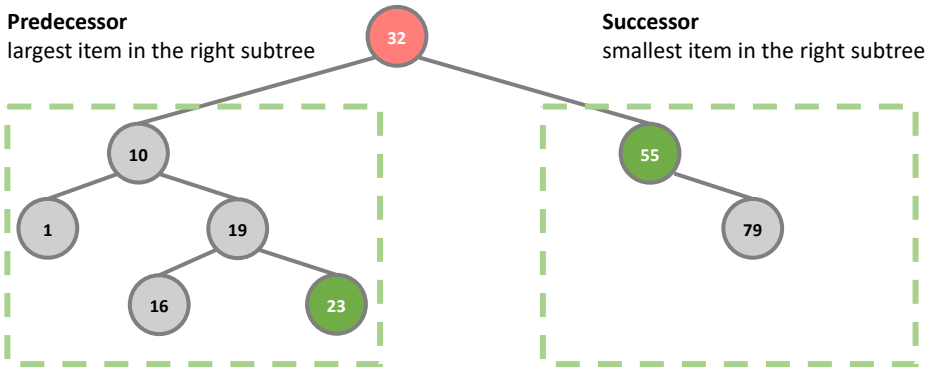
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node

After deleting, what will be the new root?

Predecessor
largest item in the right subtree

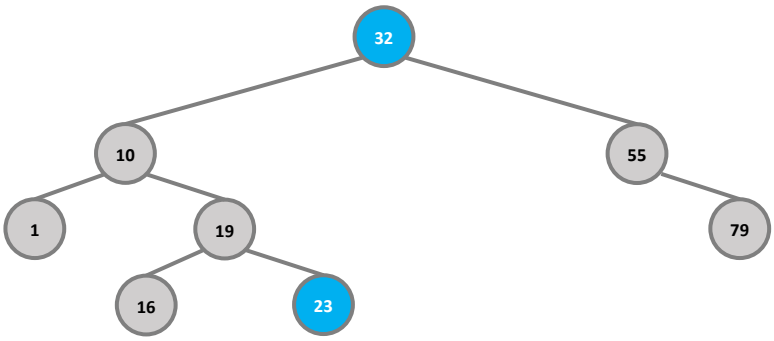
Successor
smallest item in the right subtree



Hasan Baig

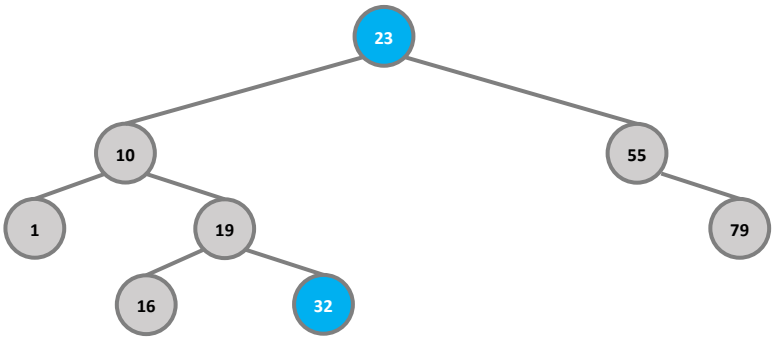
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node



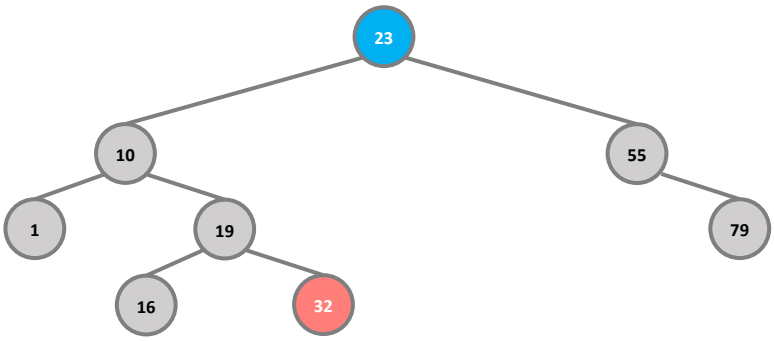
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node



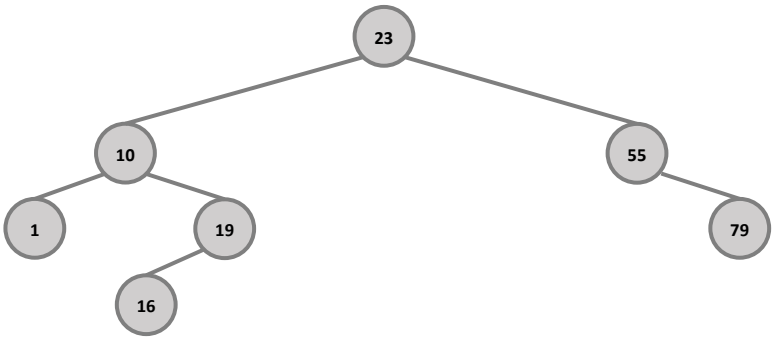
Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node



Binary Search Trees | Operations | Delete

3. Deleting a node with two child → root node



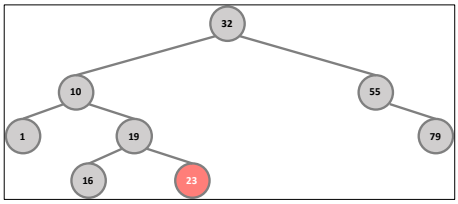
Binary Search Trees | Operations | Delete

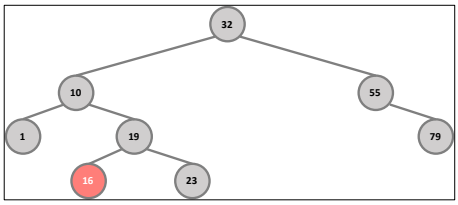
Implementation - delete()

1. Deleting a leaf node

```
def delete(self, data):
    if self.root:
        self.remove_node(data, self.root)
```

```
def remove_node(self, data, node):
    if node is None:
        return

    if data < node.data:
        self.remove_node(data, node.left)
    elif data > node.data:
        self.remove_node(data, node.right)
```



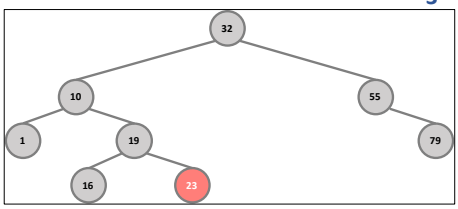


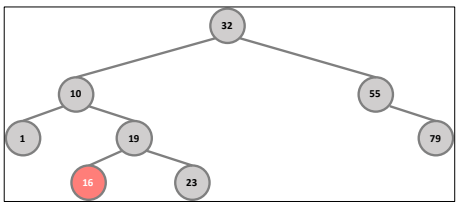
Binary Search Trees | Operations | Delete

Implementation - delete()

1. Deleting a leaf node

```
    else: #once the required node to be deleted is found
        if node.left is None and node.right is None:
            print(f"Removing a leaf node with data: {node.data}")
            parent = node.parent

            if parent is not None:
                if parent.right == node:
                    parent.right = None
                if parent.left == node:
                    parent.left = None
            else: #if the element we are removing is root
                self.root = None

            del node
```



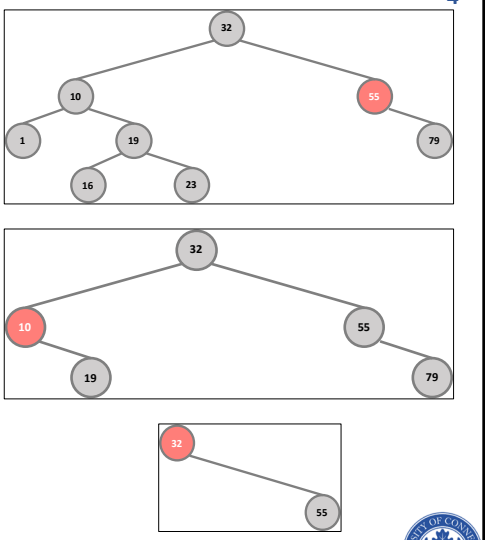


Binary Search Trees | Operations | Delete

Implementation - delete()

2. Deleting a node with one child (on right)

```
elif node.left is None and node.right is not None:
    print(f"Removing node having a right child with data: {node.data}")
    parent = node.parent

    if parent is not None:
        if parent.right == node:
            parent.right = node.right
        if parent.left == node:
            parent.left = node.right
    else:
        self.root = node.right

    node.right.parent = parent

    del node
```

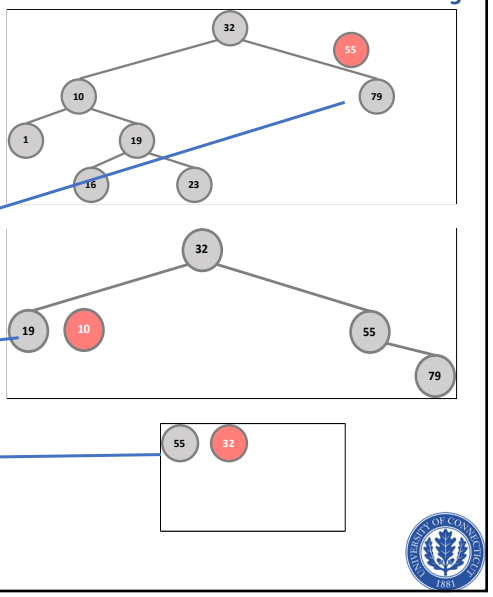


Binary Search Trees | Operations | Delete

Implementation - delete()

2. Deleting a node with one child (on right)

```
elif node.left is None and node.right is not None:
    print(f"Removing node having a right child with data: {node.data}")
    parent = node.parent

    if parent is not None:
        if parent.right == node:
            parent.right = node.right
        if parent.left == node:
            parent.left = node.right
    else:
        self.root = node.right

    node.right.parent = parent

    del node
```

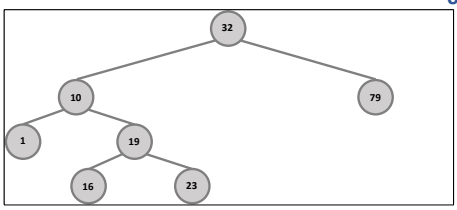


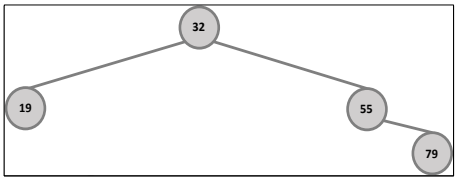
Binary Search Trees | Operations | Delete

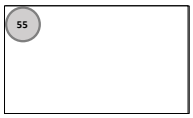
Implementation - delete()

2. Deleting a node with one child (on right)

```
elif node.left is None and node.right is not None:
    print(f"Removing node having a right child with data: {node.data}")
    parent = node.parent

    if parent is not None:
        if parent.right == node:
            parent.right = node.right
        if parent.left == node:
            parent.left = node.right
    else:
        self.root = node.right

    node.right.parent = parent

    del node
```







Binary Search Trees | Operations | Delete

Implementation - delete()

2. Deleting a node with one child (on left)

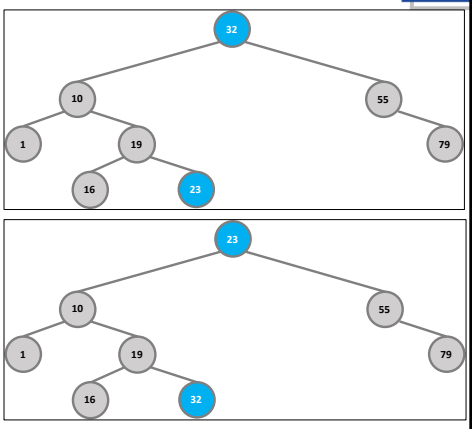
```
elif node.left is not None and node.right is None:
    print(f"Removing node having a left child with data: {node.data}")
    parent = node.parent

    if parent is not None:
        if parent.right == node:
            parent.right = node.left
        if parent.left == node:
            parent.left = node.left
    else:
        self.root = node.left

    node.left.parent = parent

    del node
```

Binary Search Trees | Operations | Delete

Implementation - delete()

3. Deleting a node with two children

```
        else: #when both children are present
            print(f"Removing node having the left child ({node.left.data}) \
            and right child ({node.right.data})")

            predecessor = self.get_predecessor(node.left)
            print("predecessor before: ", predecessor.data)
            predecessor.data, node.data = node.data, predecessor.data
            print("predecessor after: ", predecessor.data)
            self.remove_node(data, predecessor)

def get_predecessor(self, node):
    if node.right:
        return self.get_predecessor(node.right)

    return node
```



Binary Search Trees | Traversal

Tree traversal means visiting every node of the binary search tree exactly once in O(n) linear running time.

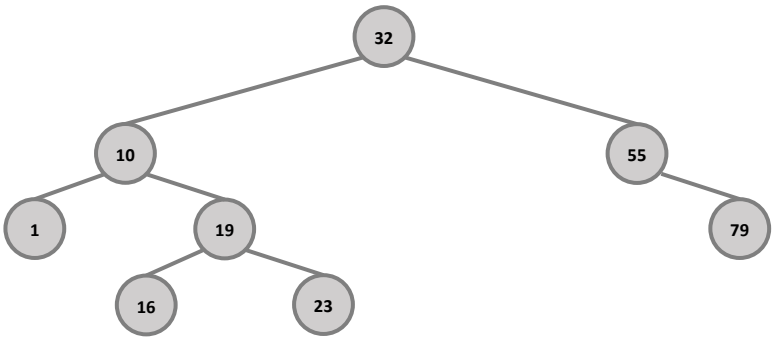
1. Pre-order traversal
   *Visit **root** node first, then **left**-subtree and finally **right**-subtree*

2. Post-order traversal
   *Visit **left**-subtree first, then **right**-subtree, and finally the **root** node*

3. In-order traversal
   *Visit **left**-subtree first, then **root** node, and finally **right**-subtree*

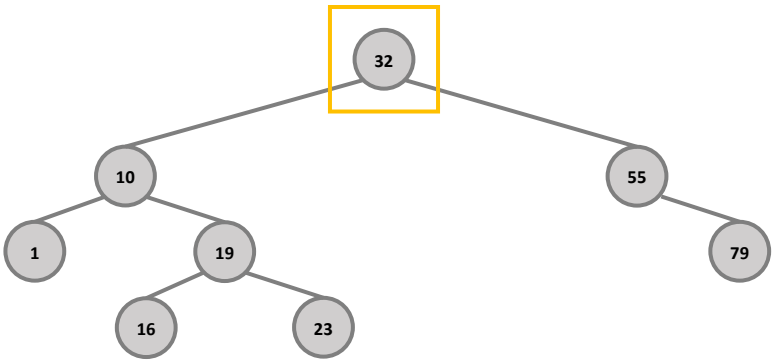
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



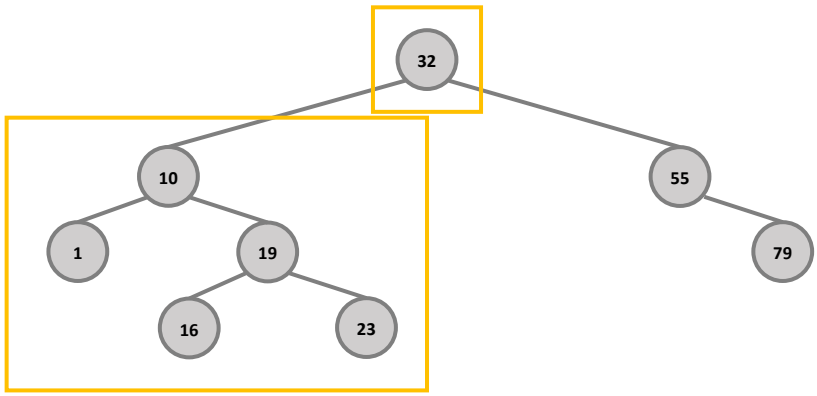
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



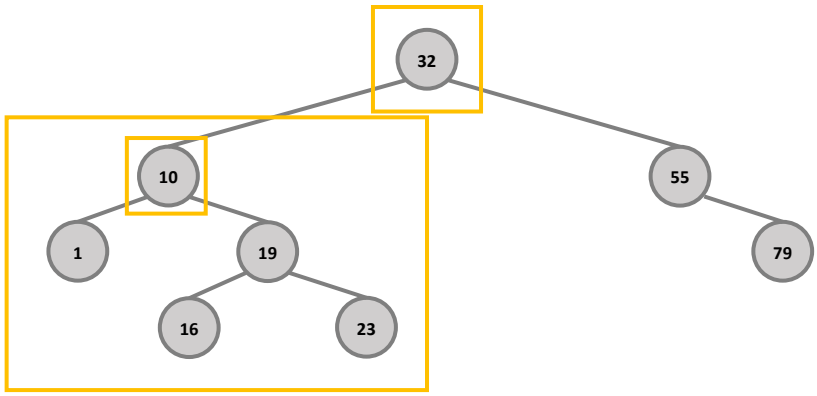
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



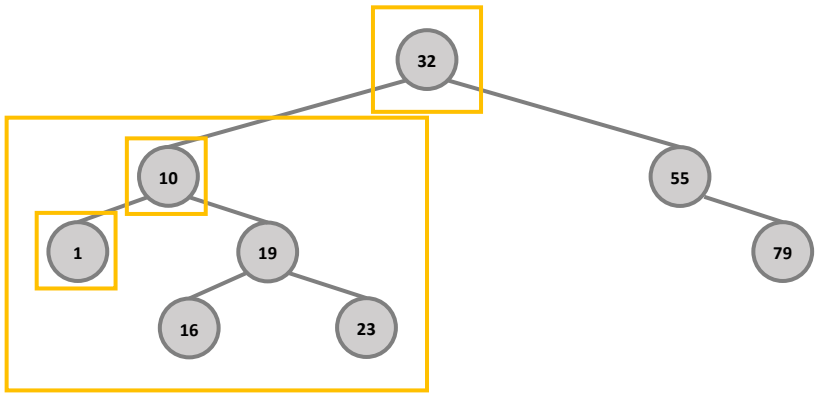
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



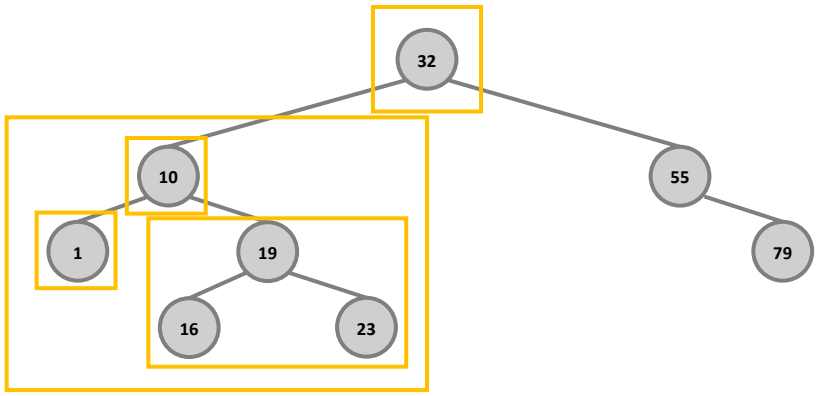
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



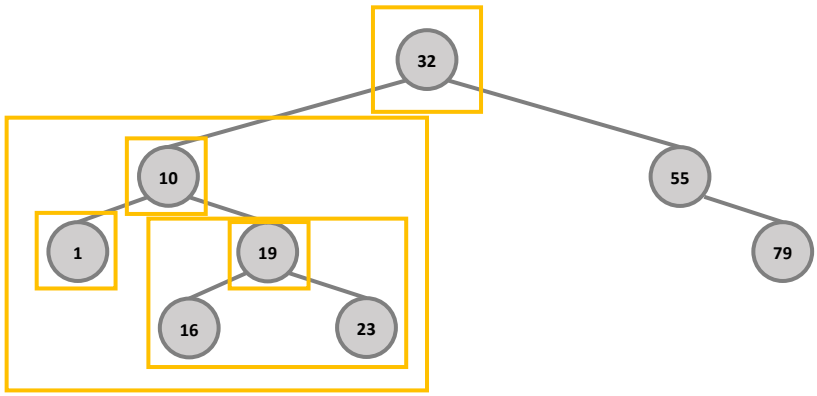
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



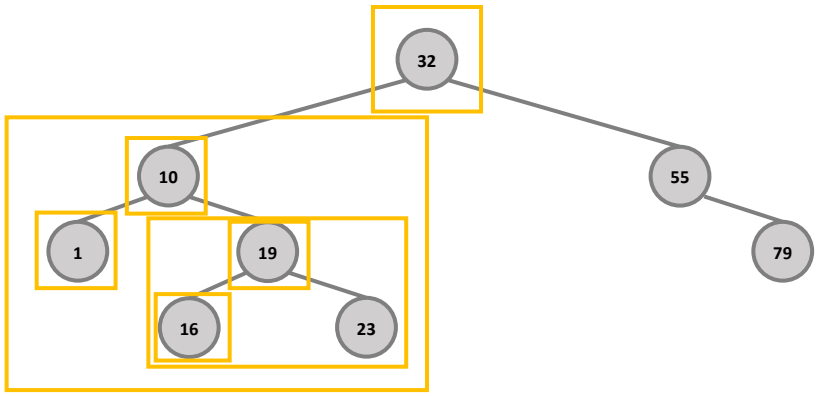
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



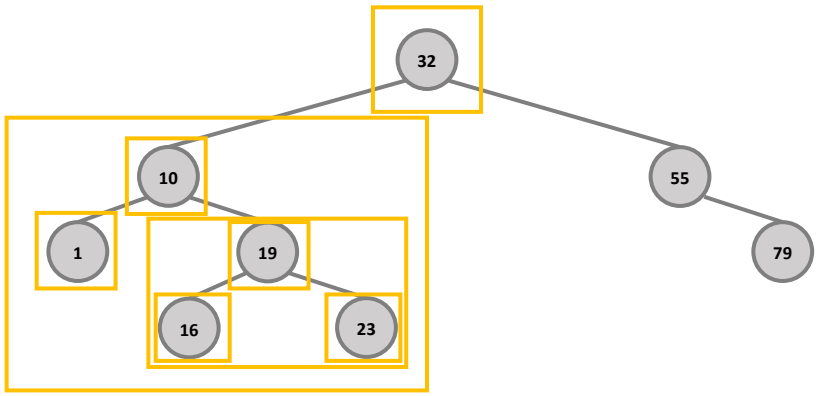
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



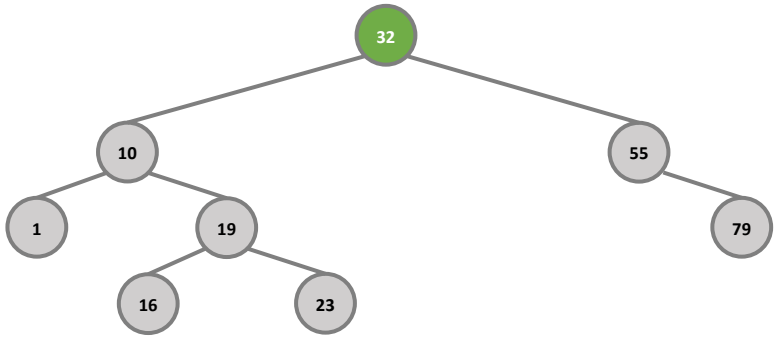
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



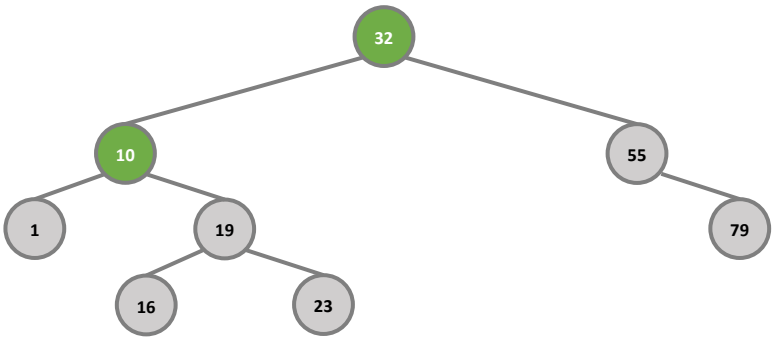
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



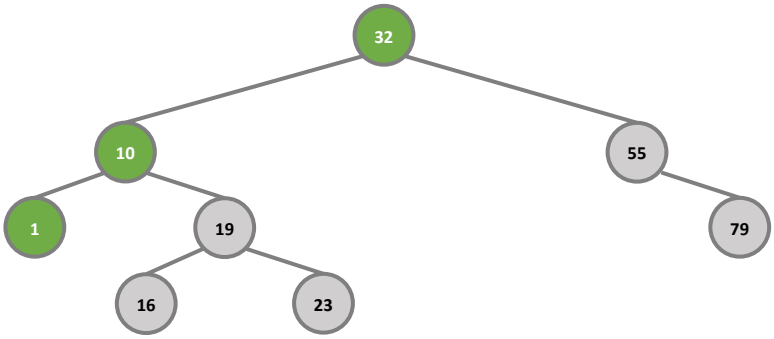
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



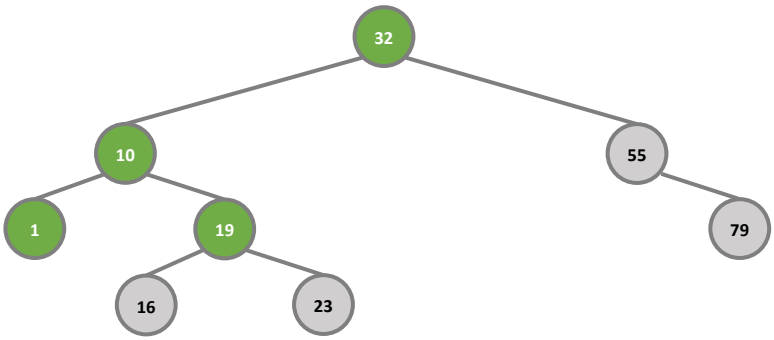
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



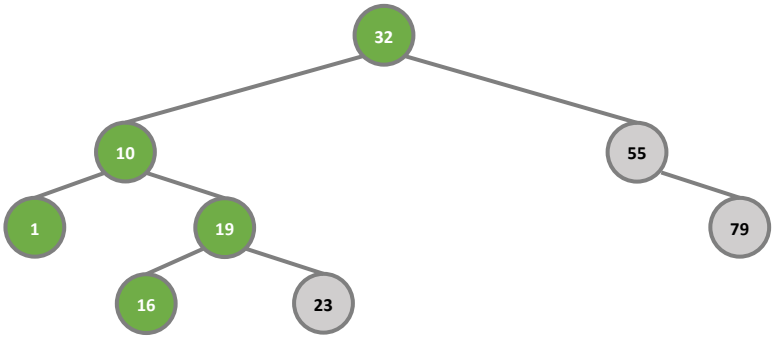
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



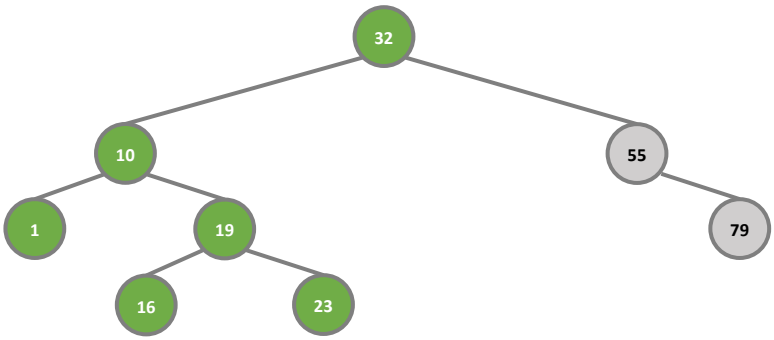
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



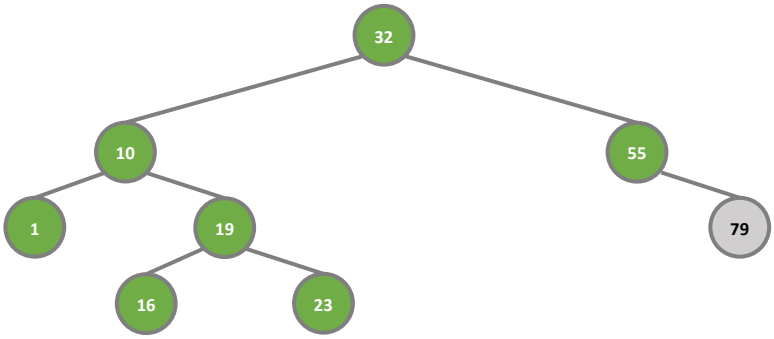
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



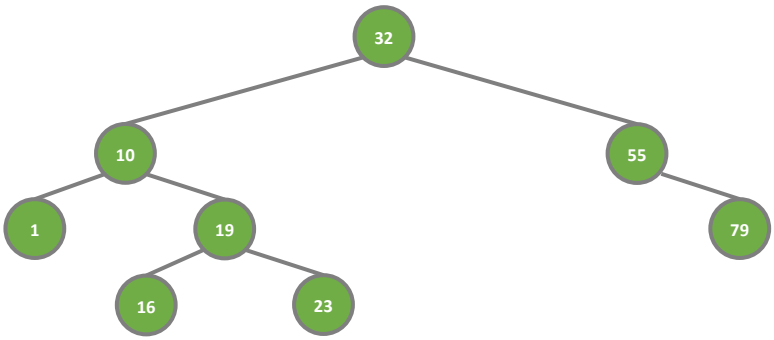
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



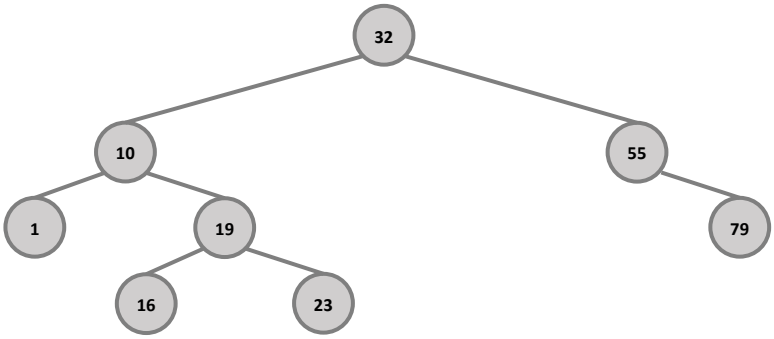
Binary Search Trees — Traversal — Pre-order

Root node → left subtree → right subtree



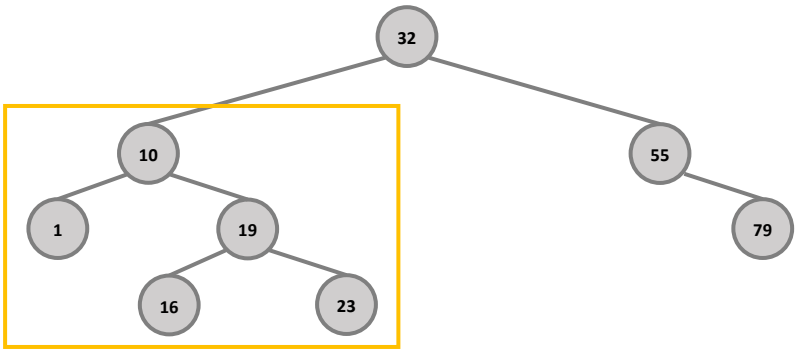
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



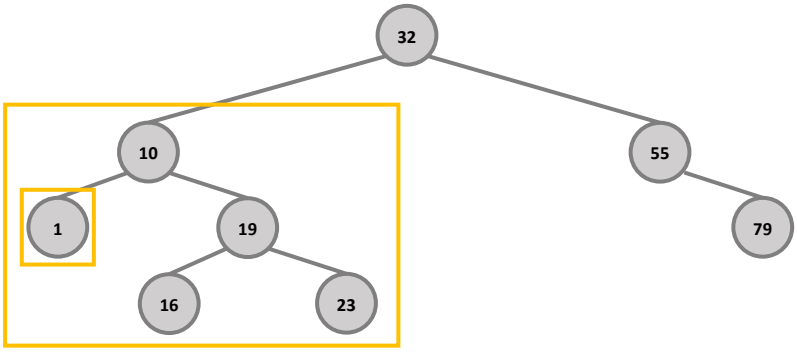
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



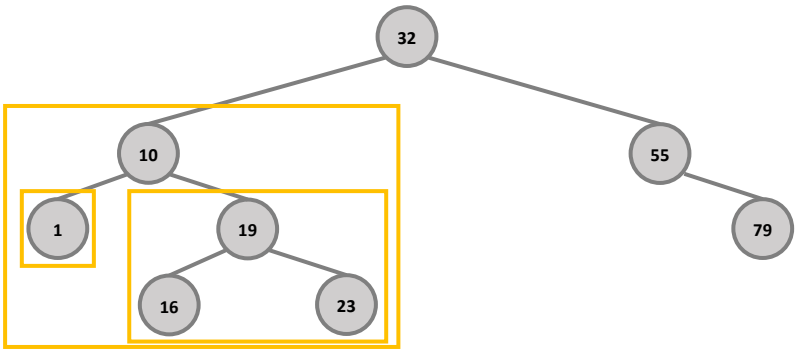
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



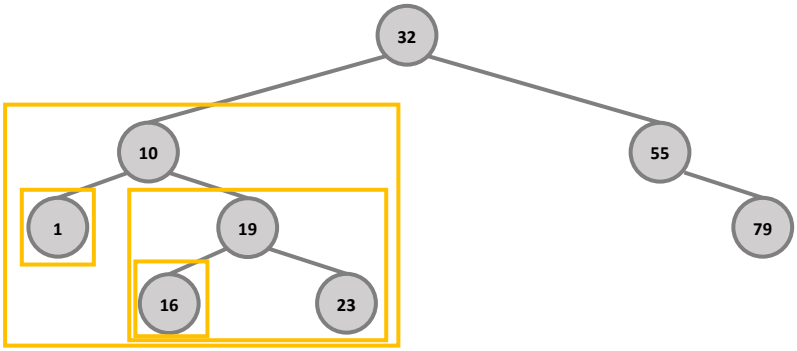
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



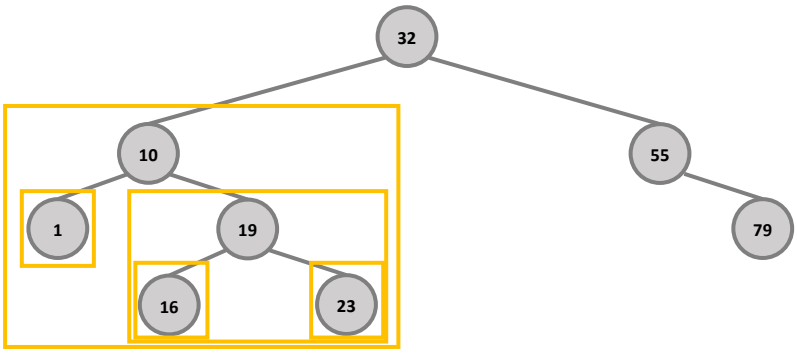
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



Binary Search Trees — Traversal — Post-order

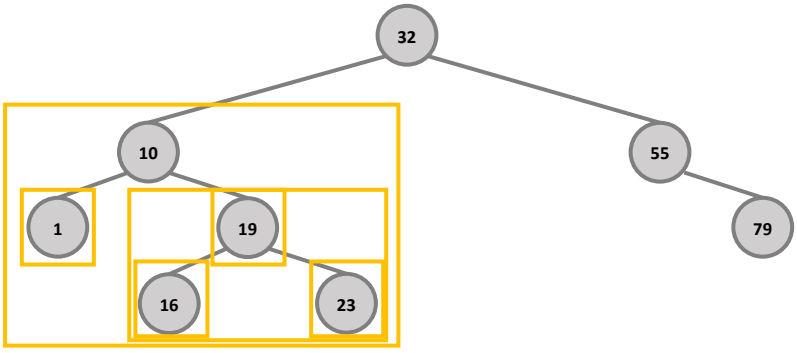
Left subtree → Right subtree → Root node



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees — Traversal — Post-order

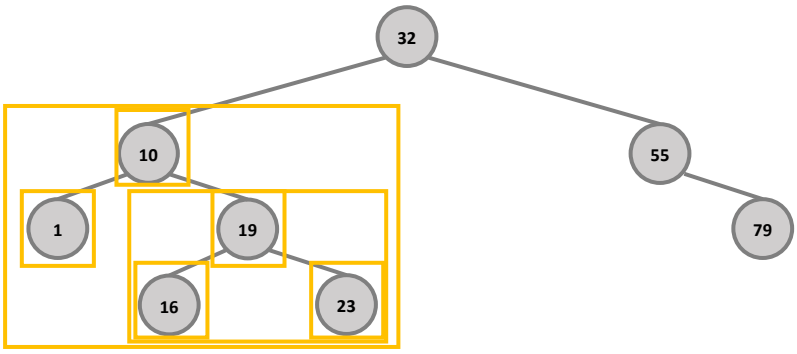
Left subtree → Right subtree → Root node



Hasan Baig

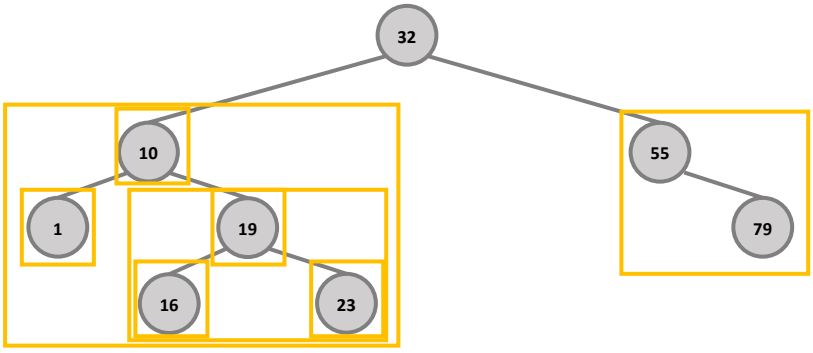
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



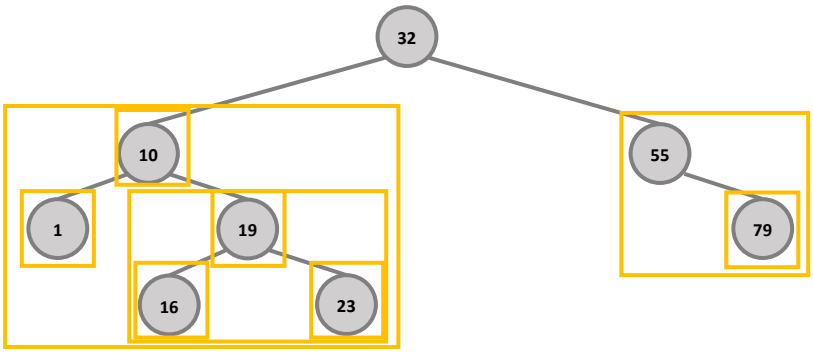
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



Binary Search Trees — Traversal — Post-order

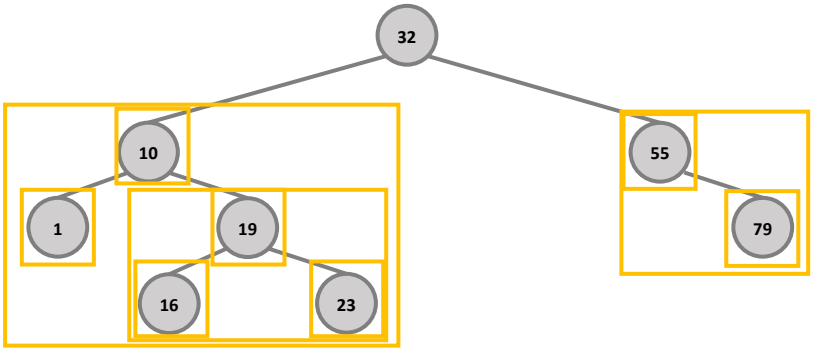
Left subtree → Right subtree → Root node



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees — Traversal — Post-order

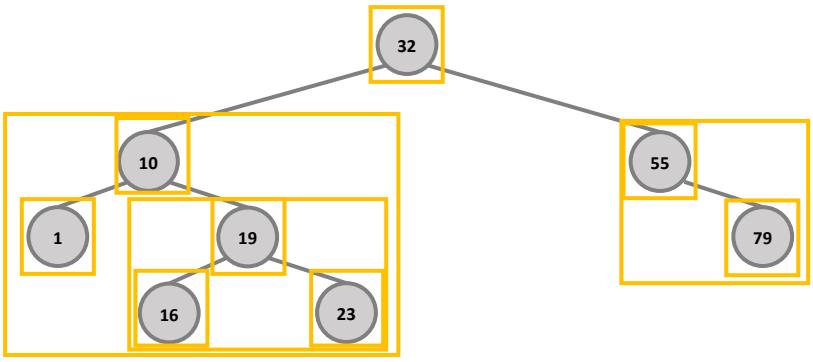
Left subtree → Right subtree → Root node



Hasan Baig

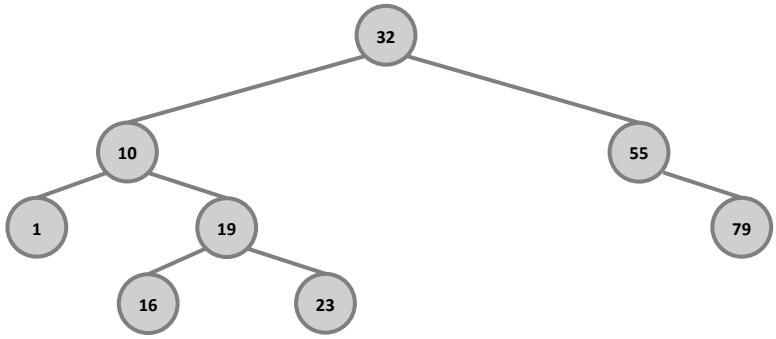
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



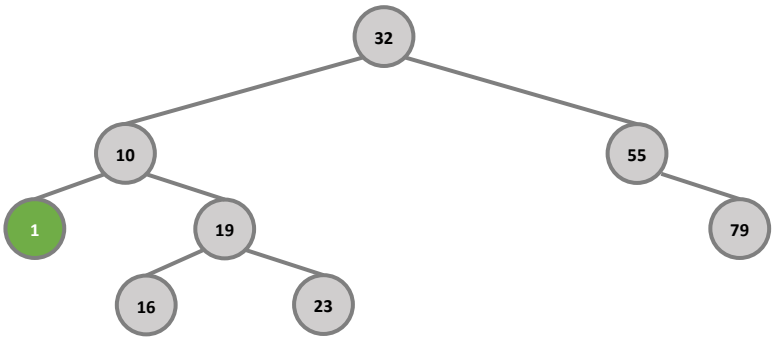
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



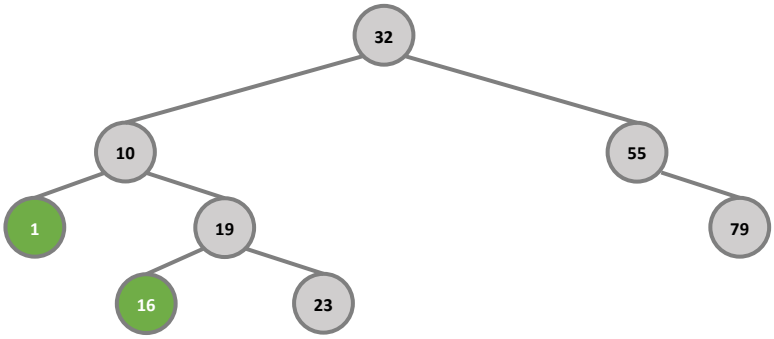
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



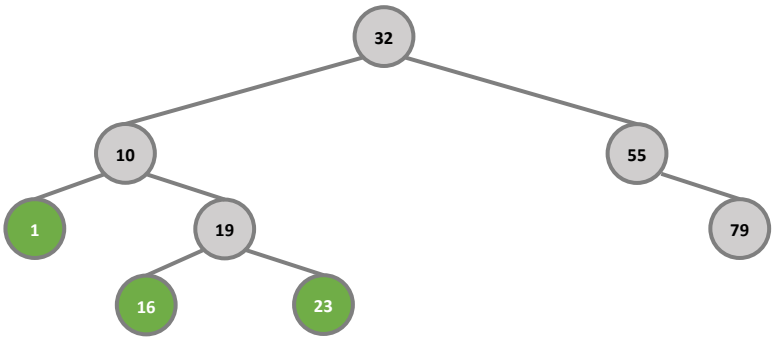
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



Binary Search Trees — Traversal — Post-order

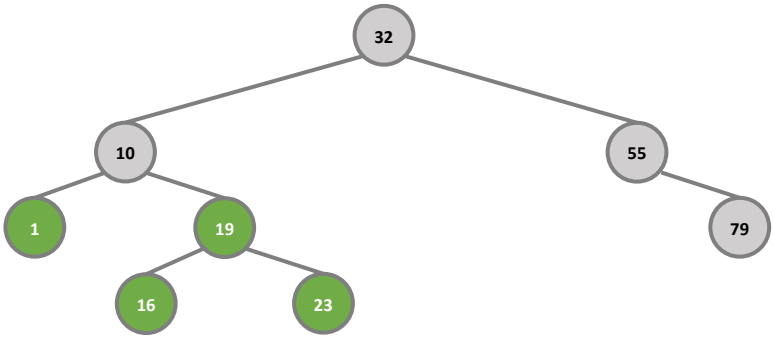
Left subtree → Right subtree → Root node



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees — Traversal — Post-order

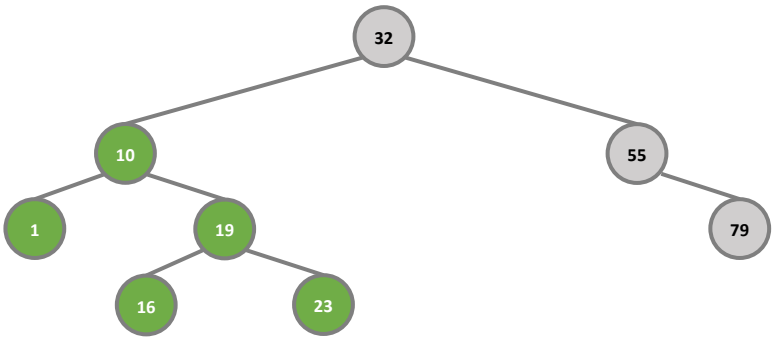
Left subtree → Right subtree → Root node



Hasan Baig

Binary Search Trees — Traversal — Post-order

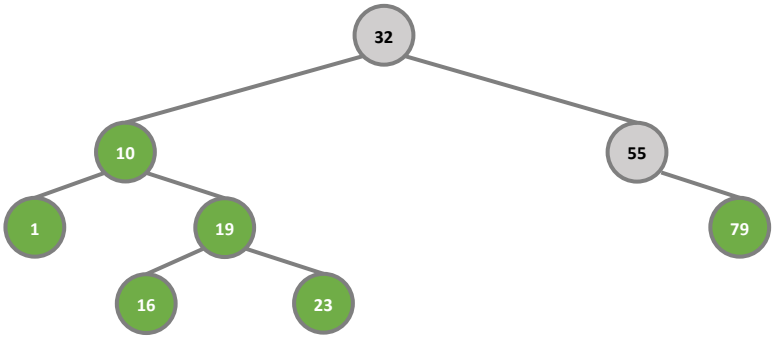
Left subtree → Right subtree → Root node



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees — Traversal — Post-order

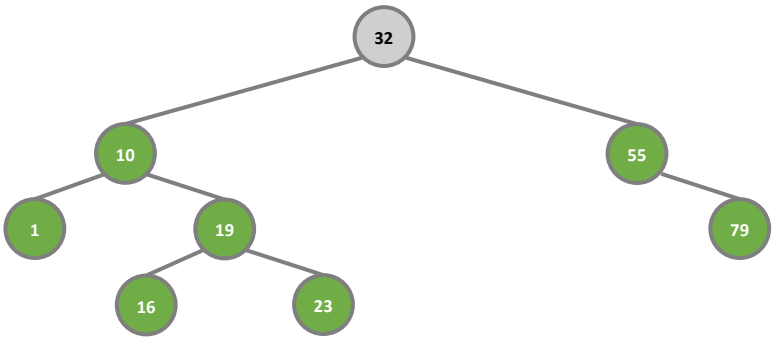
Left subtree → Right subtree → Root node



Hasan Baig

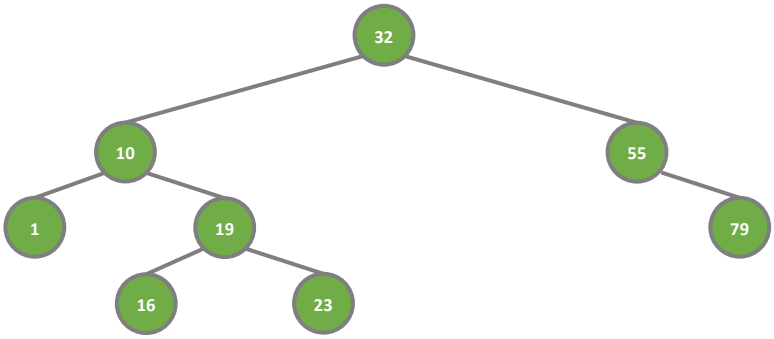
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



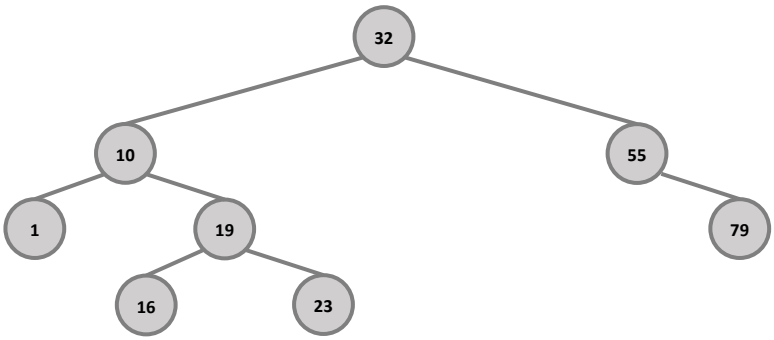
Binary Search Trees — Traversal — Post-order

Left subtree → Right subtree → Root node



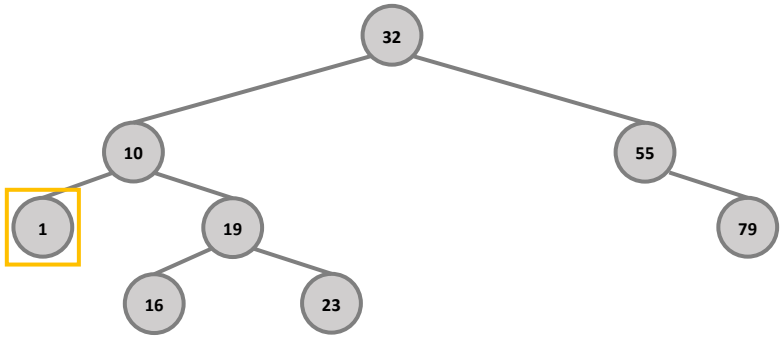
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



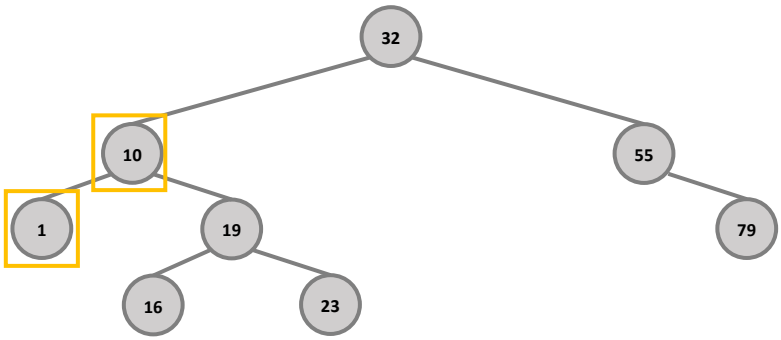
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



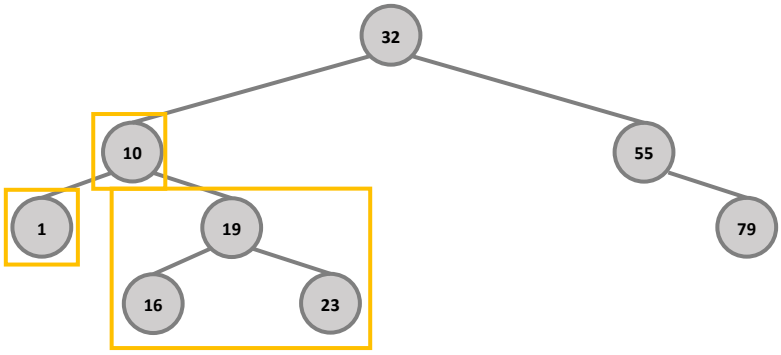
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



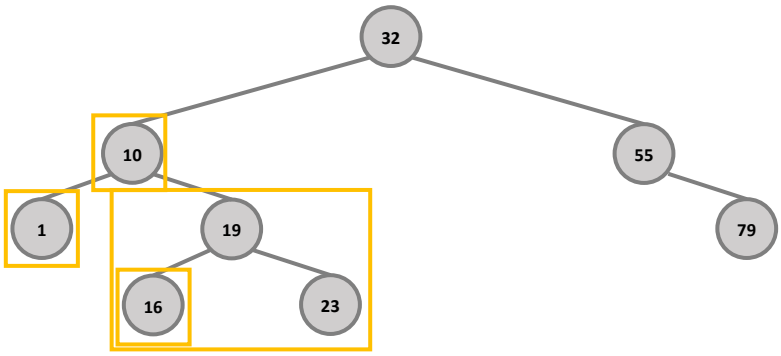
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



Binary Search Trees — Traversal — In-order

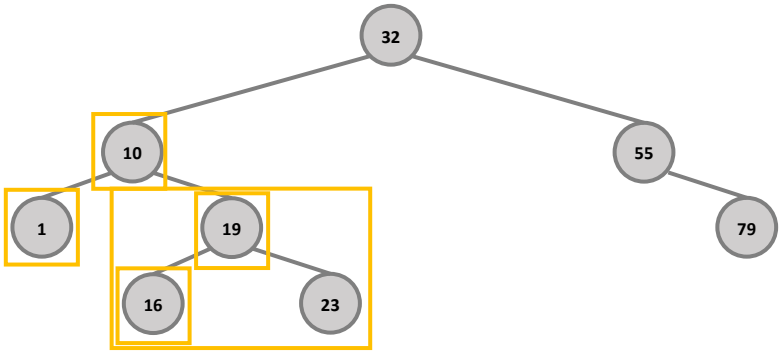
Left subtree → Root node → Right subtree



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees — Traversal — In-order

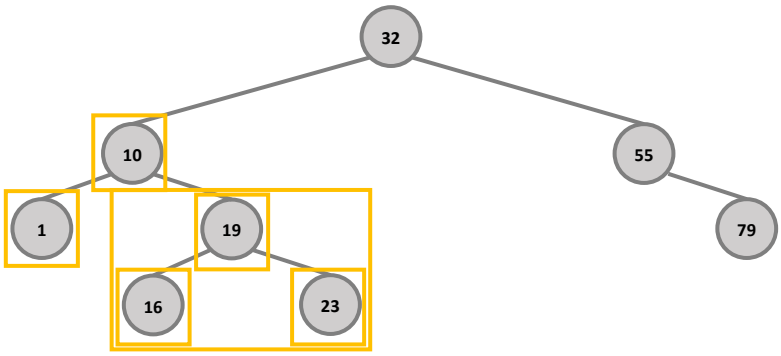
Left subtree → Root node → Right subtree



Hasan Baig

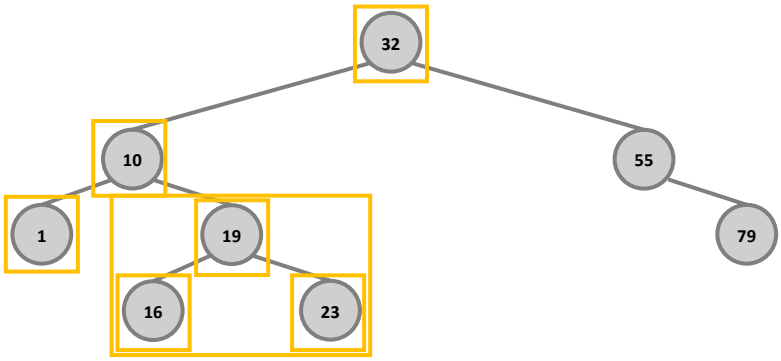
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



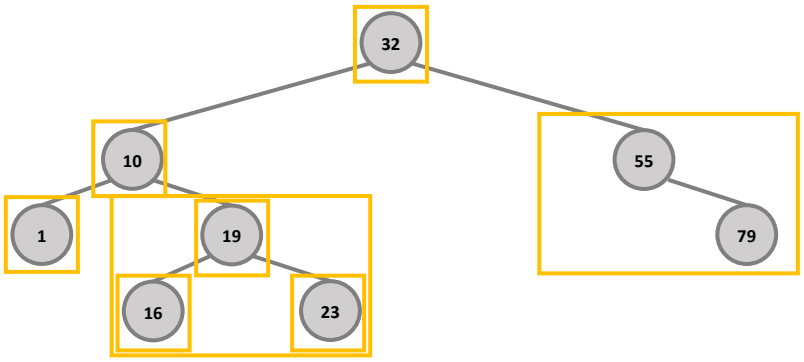
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



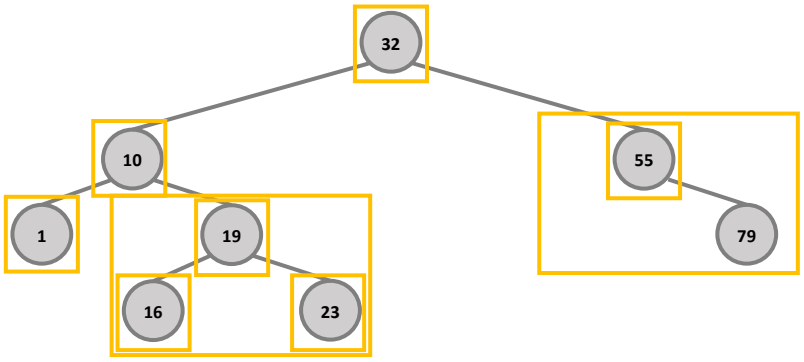
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



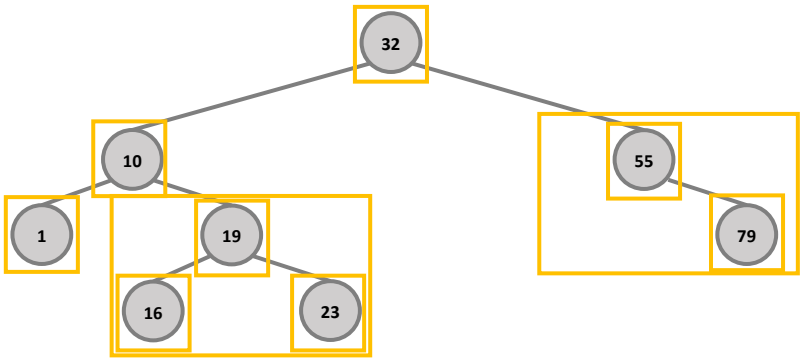
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



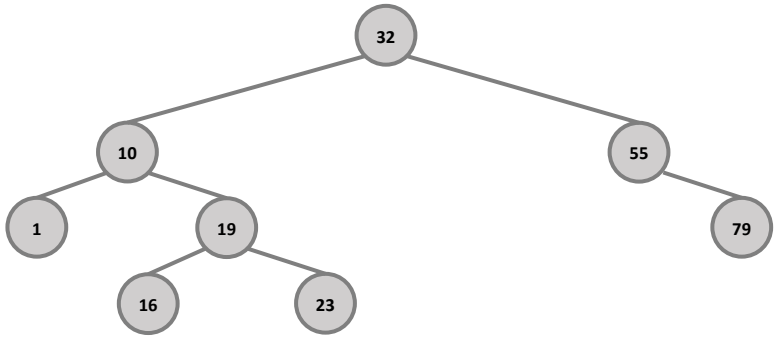
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



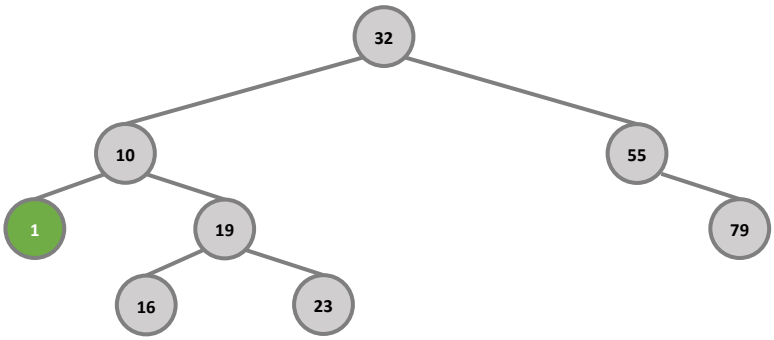
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



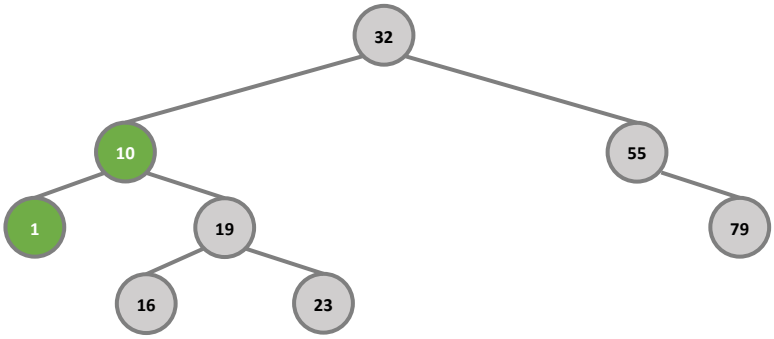
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree

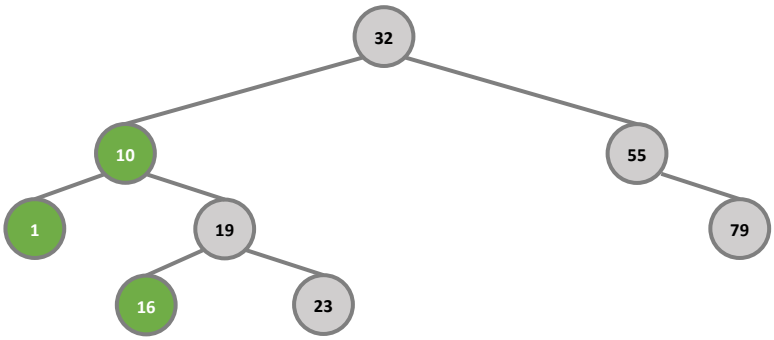


Binary Search Trees

Traversal

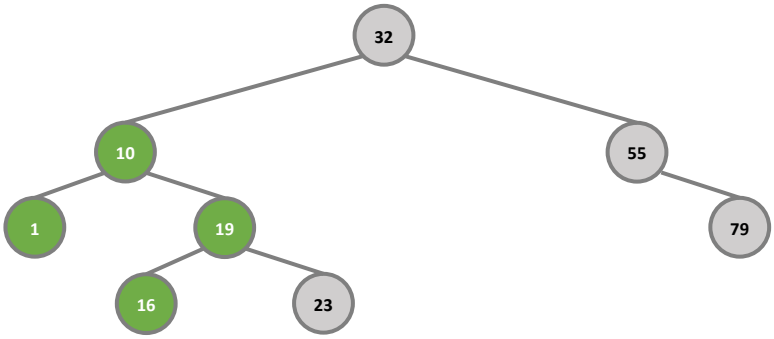
In-order

Left subtree → Root node → Right subtree



CSE-2050 – Data Structures and Object-Oriented Design
Trees

Binary Search Trees
Traversal
In-order
Left subtree → Root node → Right subtree



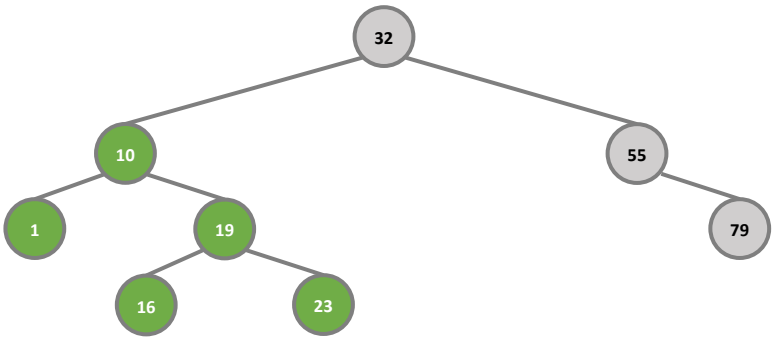
Hasan Baig

Binary Search Trees

Traversal

In-order

Left subtree → Root node → Right subtree

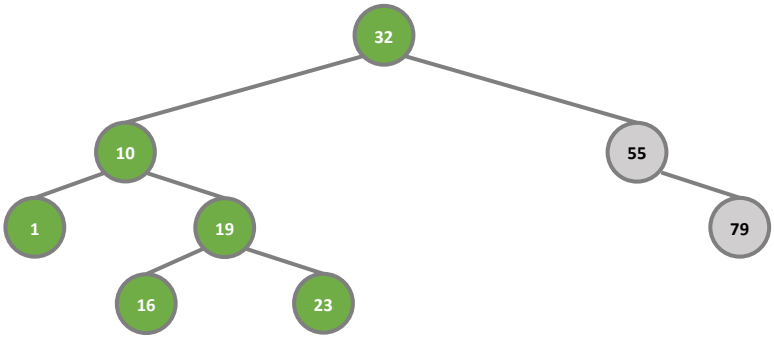


Binary Search Trees

Traversal

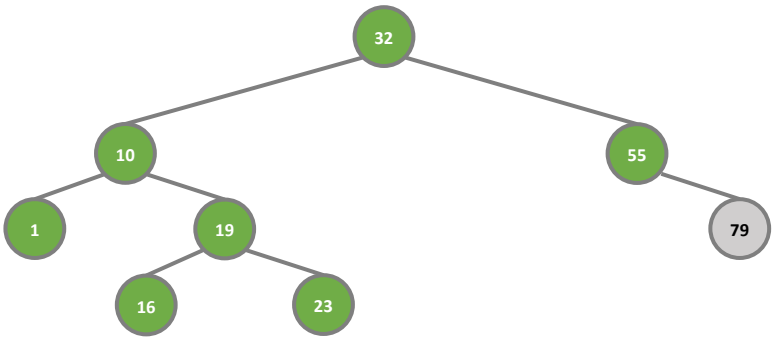
In-order

Left subtree → Root node → Right subtree



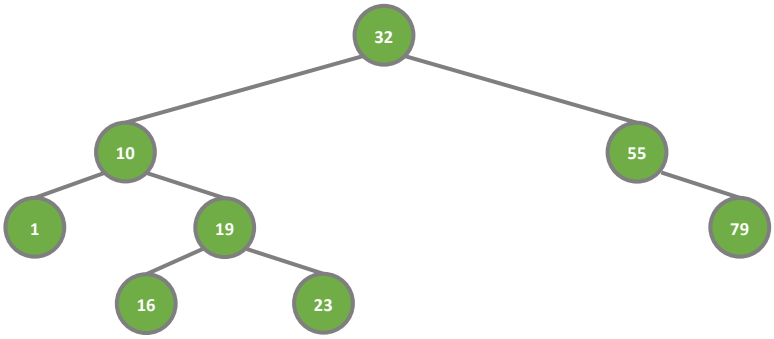
Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



Binary Search Trees — Traversal — In-order

Left subtree → Root node → Right subtree



Left subtree → Root node → Right subtree

Binary Search Trees

Traversal

In-order

Pre-order

- Root node → left subtree → right subtree

32, 10, 1, 19, 16, 23, 55, 79

Post-order

- Left subtree → Right subtree → Root node

1, 16, 23, 19, 10, 79, 55, 32

In-order

- Left subtree → Root node → Right subtree

1, 10, 16, 19, 23, 32, 55, 79

- Returns elements in sorted order