CMPUT 175
Introduction to Foundations of Computing

Trees

Objectives

- Study a non-linear container called a Tree.
- Understand what a tree data structure is and what it is used for.
- Learn about a special kind of Tree called a Binary Tree.
- Introduce an implementation of Ordered Structure called the Binary Search Tree.

Outline of Lecture

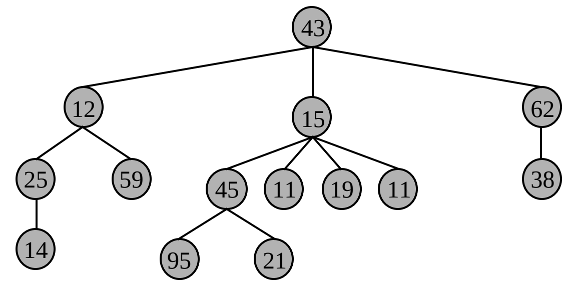
- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

Trees as we know them: an upside down world

Leaves

Branches

Roots



Root

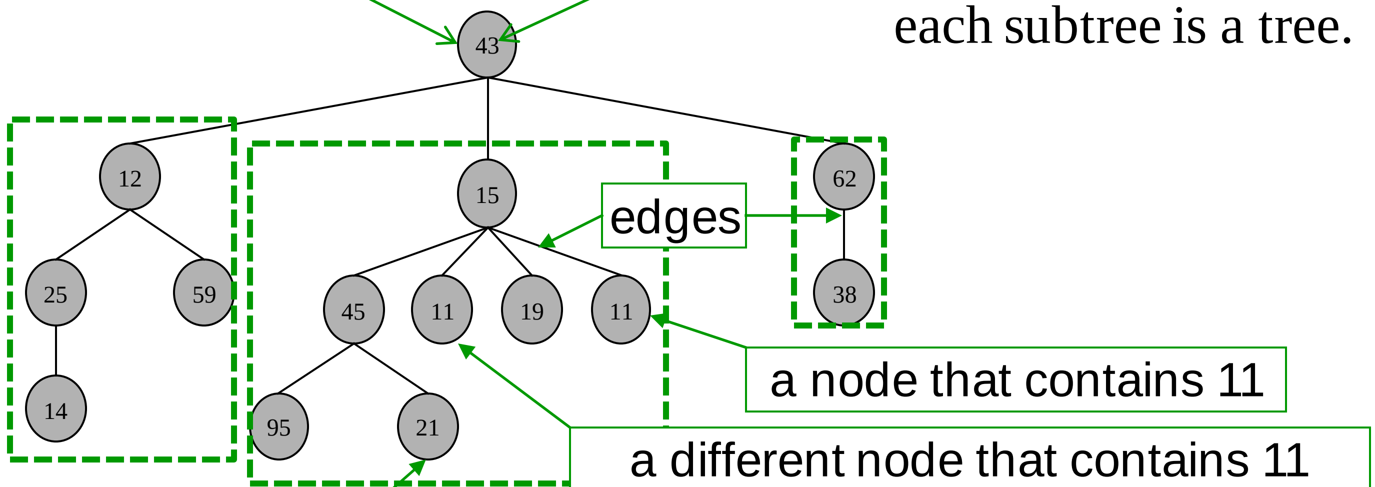
Leaves

Terminology

A tree has a root

A root contains a root element and a (possibly empty) list of subtrees, where each subtree is a tree.

Since each subtree is a tree it has its own root.



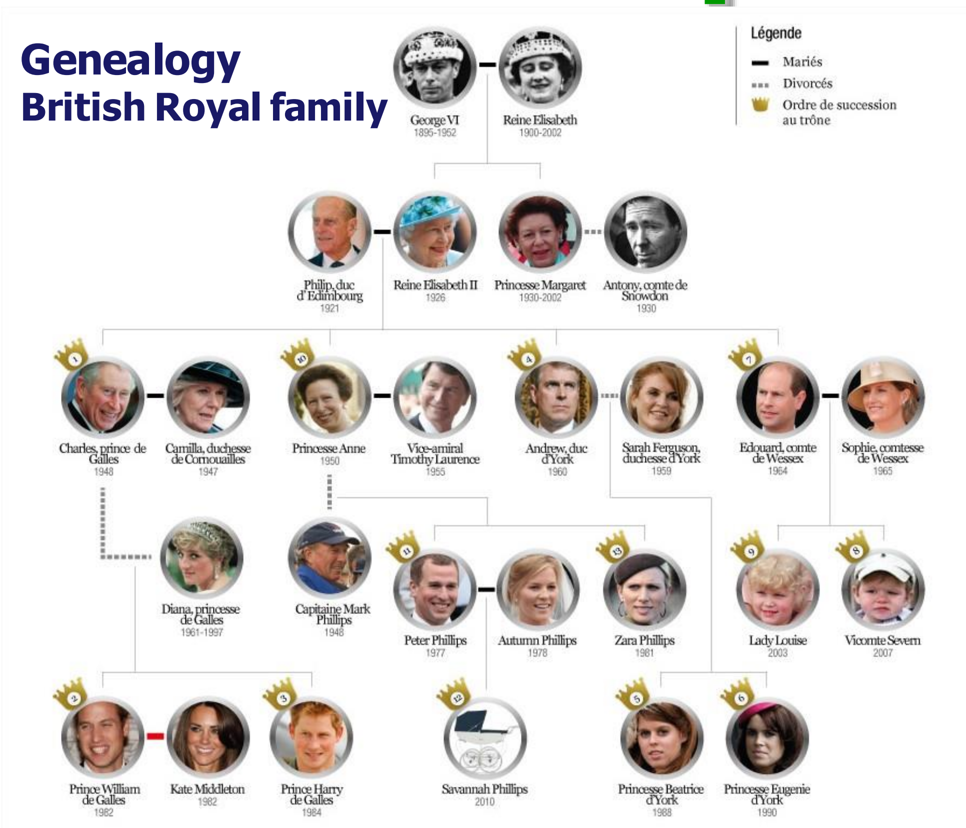
The relation between the root of a tree and the roots of its subtrees is called an edge and it is drawn as a line segment.

A leaf node is a node that does not have subtrees

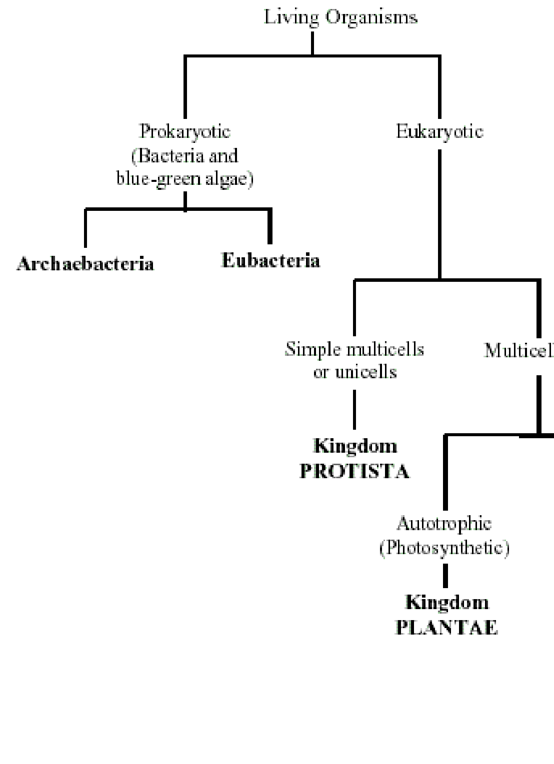
The term node is used to refer to an element at a particular location in a tree so that each node "contains" an element.

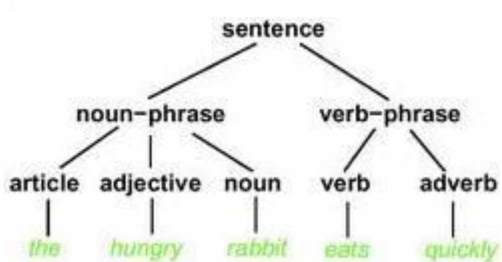
Examples of trees

Genealogy
British Royal family

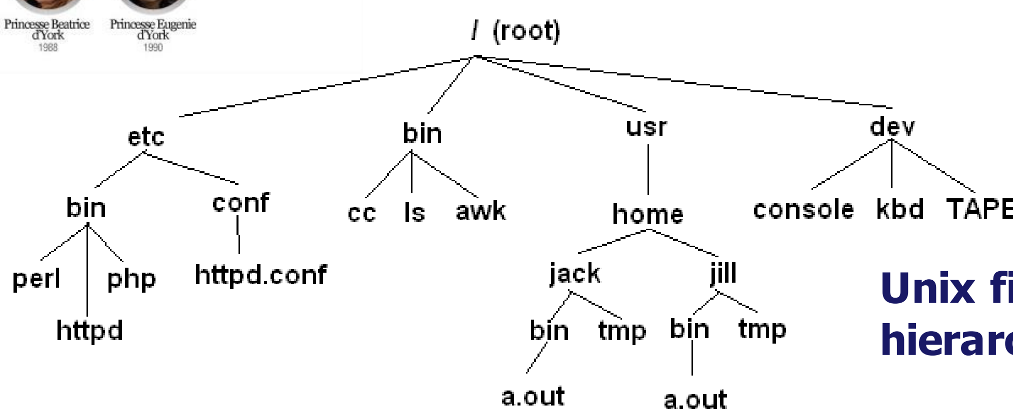


Taxonomic Kingdoms of Life



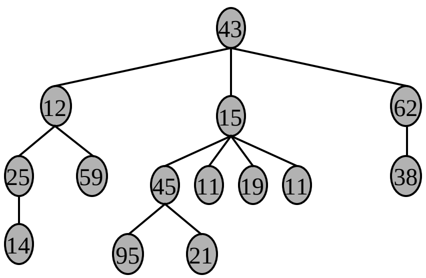


Derivation tree in NLP

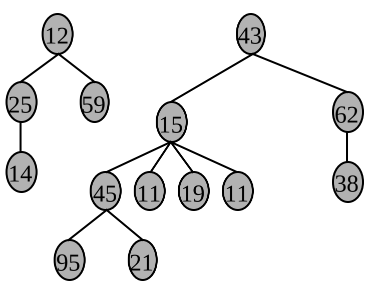


Unix file system hierarchy

Terminology Summary



- **Node**: fundamental part of a tree. It contains a data element
- **Edge**: fundamental part of a tree. It connects two nodes
- A node has a **parent** and can have **children**.
- In a tree, a node has only <u>one parent</u>. If there are many parents, we talk about a **graph**.
- **Root**: A unique node without incoming edge – i.e. a node without a parent.
- **Leaf-node**: a node without children
- Two nodes are **siblings** if they have the same parent.
- **Level** of node N: the number of edges to reach N from the root.
- **Height of a tree**: maximum level of any node in the tree
- **Subtree**: set of nodes and edges comprised of a parent and all its descendants.
- **Path**: a path from node n_1 to n_2 is the sequence of edges and nodes connecting node n_1 to n_2.
- **Forest**: a **forest** is a set of disjoint trees. Two trees are disjoint if they share no nodes and no edges.

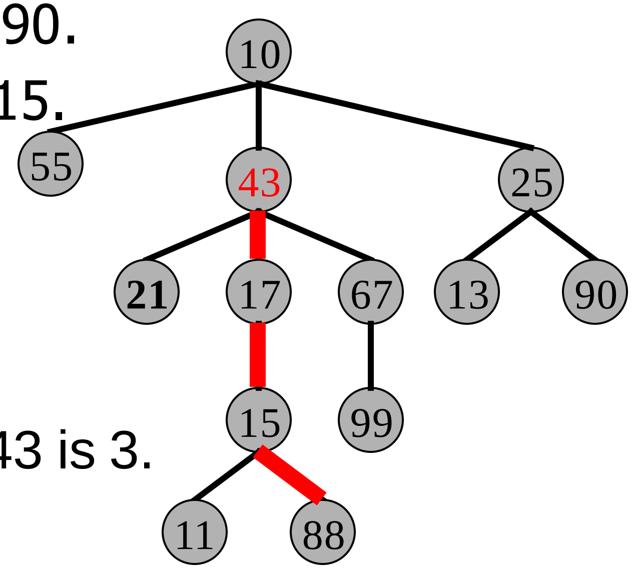


Trees - node relationships example

- The ancestors of node 21 are nodes: 43 and 10.
- The parent of node 21 is node 43.
- The children of node 43 are nodes: 21, 17 and 67.
- The descendants of node 43 are nodes: 21, 17, 67, 15, 11, 88 and 99.

An **interior** node is a node that has at least one child (i.e. not leaf node)

- The leaf nodes are: 55, 21, 11, 88, 99, 13 and 90.
- The interior nodes are: 10, 43, 25, 17, 67 and 15.
- The siblings of node 21 are nodes: 17 and 67.



The **length of a path** is the number of edges in it

- The path from node 88 to node 43 is in red.
- The length of the path from node 88 to node 43 is 3.
- The height of node 43 is 3.
- The height of the tree is the height of node 10: 4.

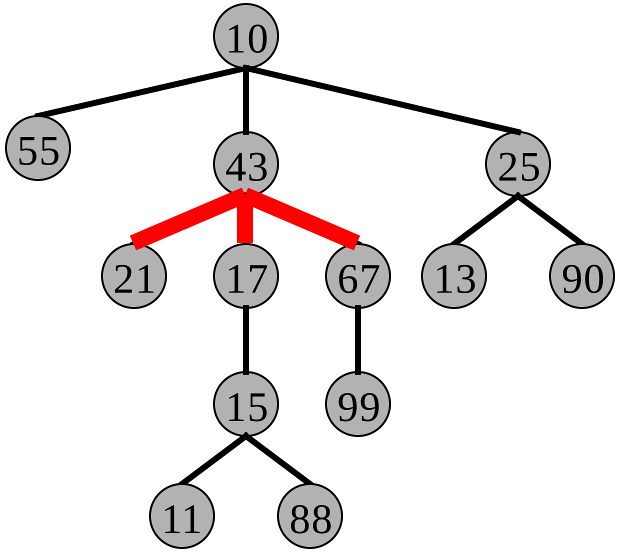
Trees - node relationships example

- The depth (level) of node 67 is the length of the path from node 10 to node 67 which is 2. Depth or level of root is 0.

Depth of node is from root to leaf
Height of node is from leaf to rot

The **degree** of a node is the number of children it has

- The degree of node 43 is 3 and the degree of node 25 is 2
- The degree of node 11 is 0.



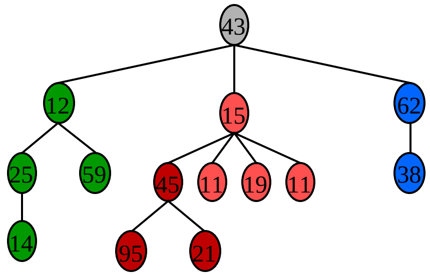
The **degree of a tree** is the maximum degree of any node in the tree

- The degree of the tree is 3 since 43, the node with the most children has a degree of 3.

The **degree of a binary tree** is always 2

Defining a tree

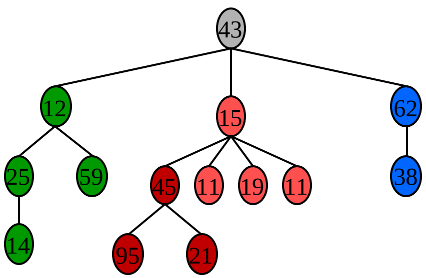
- A tree consists of a set of nodes and a set of edges such that:
  - One node of the tree is designated as the root node.
  - Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.
  - A unique path traverses from the root to each node.



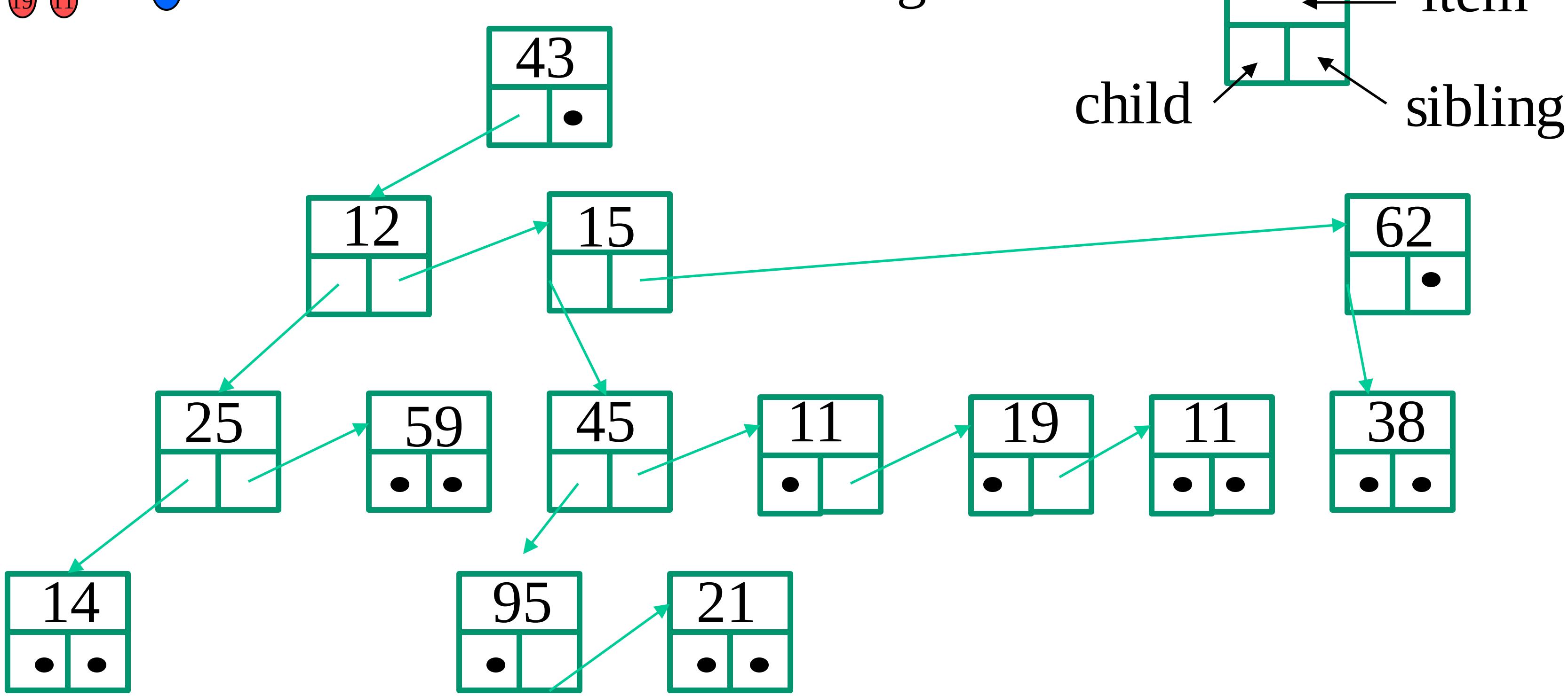
- A tree can be represented as a list containing a root and the list of sub-trees (descendants)
- [root, **[subtrees]**]
- A subtree is recursively defined the same as above

[43,[[12,[[25,[14]],[59]]],[15,[45,[[95],[21]]],[11],[19],[11]],[62,[38]]]]

Defining a tree

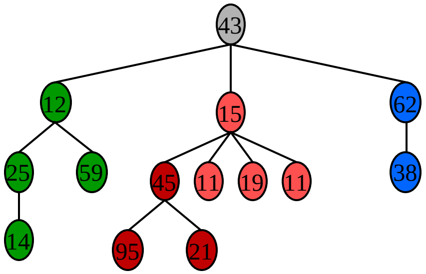


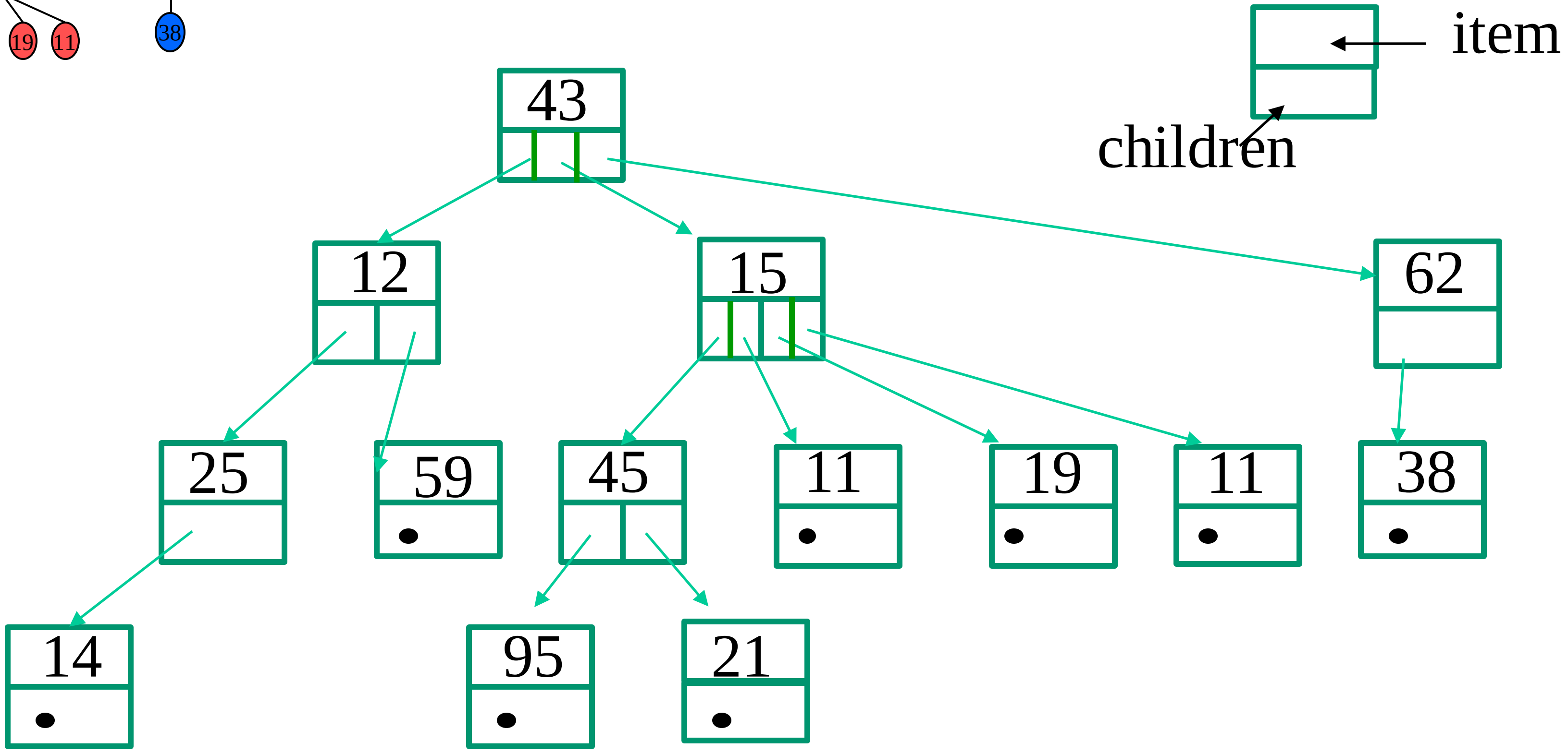
- A tree can also be represented as a linked list of nodes
- A node has an item, a link to the children list and a link to the sibling list



Defining a tree

- A tree can also be represented as a linked list of nodes
- A node has an item, and a list of children

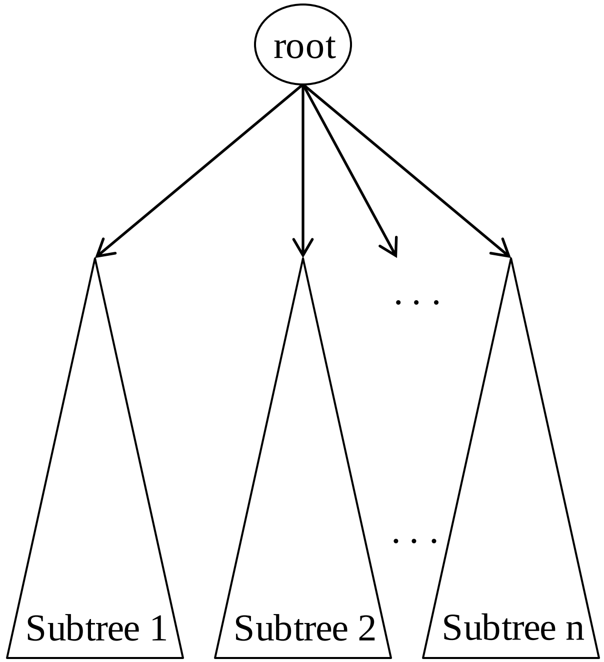


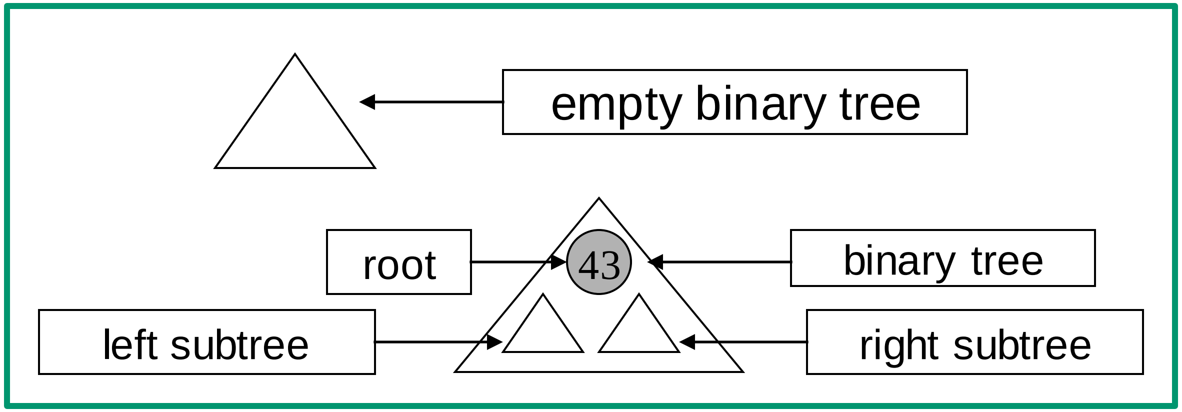


More on Defining a tree

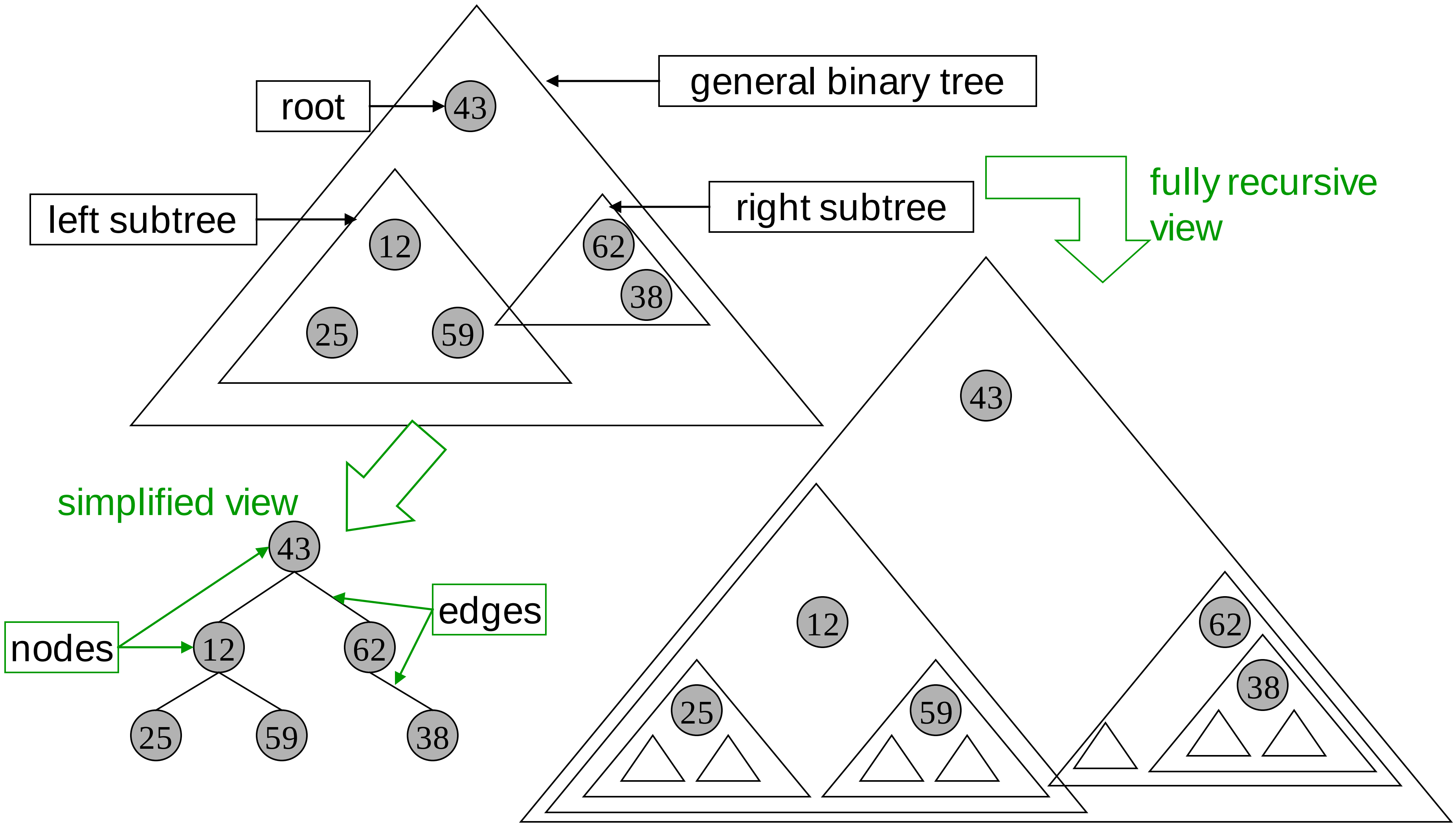
Recursive definition

- A tree can either be:
  - Empty
  - A root and a list of subtrees, each of which is a tree.
  - The root of each subtree is connect by an edge from the root of the parent tree.





Recursive Binary Tree

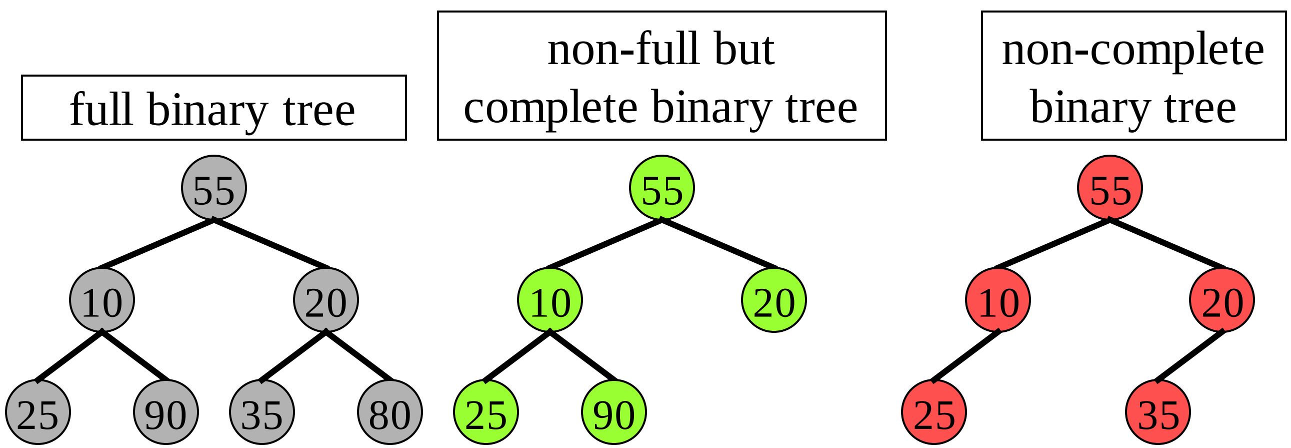


Binary Trees - terminology

- A binary tree is a tree in which nodes can have a maximum of 2 children.
- A binary tree is oriented if every node with 2 children differentiates between the children by calling them the left child and right child and every node with one child designates it either as a left child or right child.
- A node in a binary tree is full if it has arity 2.
- A full binary tree of height h has leaves only on level h and each of its interior nodes is full.
- A complete binary tree of height h is a full binary tree of height h with 0 or more of the rightmost leaves of level h removed.

Binary Tree Examples

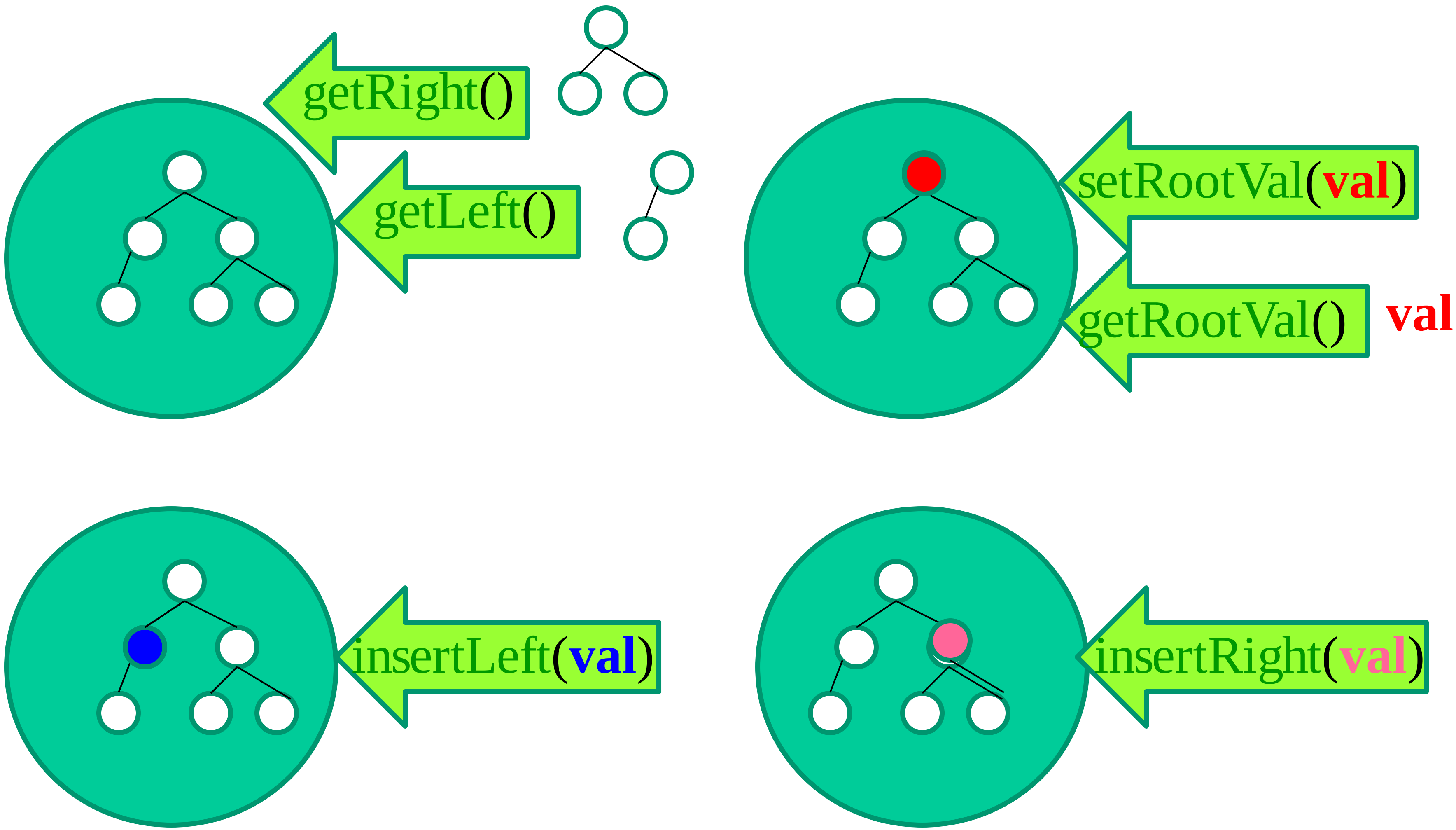
- Node 55 is full in all trees since it has arity 2 in all of them.
- Node 20 is full in the left tree, but not full in the middle tree where it has arity 0 and not full in the right tree where it has arity 1.



Outline of Lecture

- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

Binary Tree ADT interface



Binary Tree ADT interface

BinaryTree(root) creates a new instance of a binary tree with the given root

getLeft() returns the binary tree corresponding to the left child of the current node.

getRight() returns the binary tree corresponding to the right child of the current node.

setRootVal(val) stores the object in parameter val in the current node.

getRootVal() returns the object stored in the current node.

insertLeft(val) creates a new binary tree and installs it as the left child of the current node

Make the original left subtree, if exists, the left child of the new left child

insertRight(val) creates a new binary tree and installs it as the right child of the current node.

Make the original right subtree, if exists, the right child of the new right child

Extended Interface

We will NOT implement these

There could be many more methods such as

size() return the number of elements in the tree

isEmpty() return true iff the receiver is empty

clear() makes the receiver tree and all its subtrees empty

hasLeft() returns true iff the receiver's left subtree is non-empty

hasRight() returns true iff the receiver's right subtree is non-empty

hasParent() returns true iff the receiver has a parent

contains(anObject) returns true iff the receiver contains anObject

remove(anObject) if receiver contains an element equal to anObject one such element is removed and returned

add(anObject) anObject is added to the receiver. It will be added as the root or a child of the root if possible, otherwise recursively in the left subtree (left chosen arbitrarily).

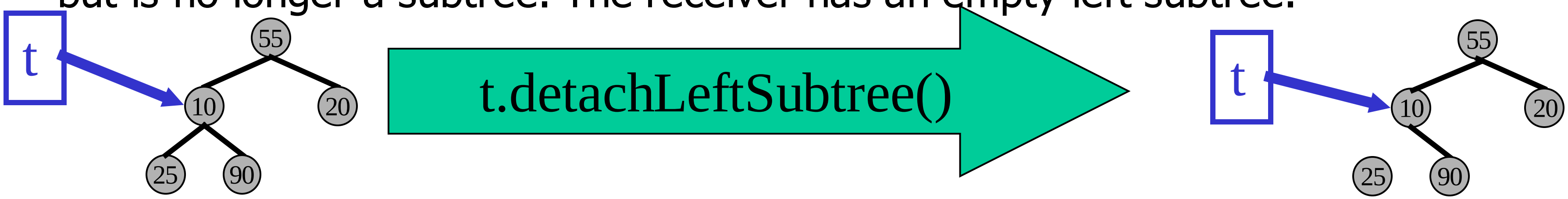
getParent() returns the parent node of the current node

Extended Interface

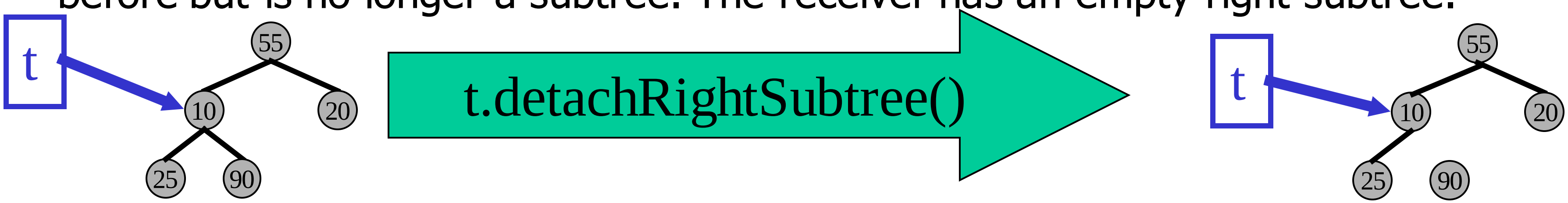
We will NOT implement these

There could be even more methods such as

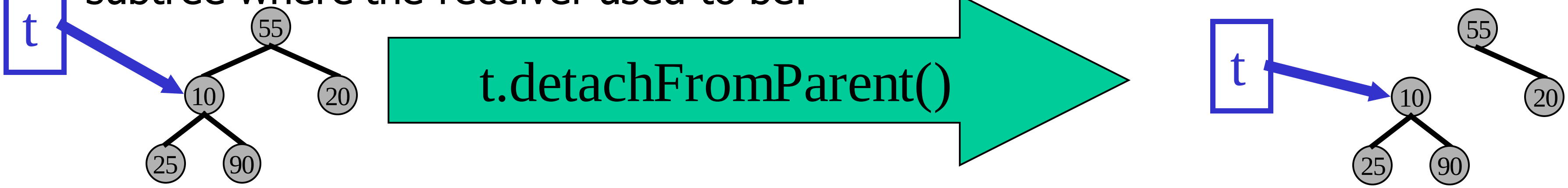
detachLeftSubtree() detaches the receiver's left subtree - it is the same as before but is no longer a subtree. The receiver has an empty left subtree.



detachRightSubtree() detaches the receiver's right subtree - it is the same as before but is no longer a subtree. The receiver has an empty right subtree.



detachFromParent() detaches the receiver from its parent. The receiver is the same tree as before but is no longer a subtree. The parent has an empty subtree where the receiver used to be.



Extended Interface

We will NOT implement these

There could be even more methods such as:

attachLeft(newLeft) newLeft becomes the receiver's left subtree.
If the receiver had a left subtree it is detached.
If newLeft had a parent, it is detached before being attached to the receiver.

attachRight(newRight) newRight becomes the receiver's right subtree.
If the receiver had a right subtree it is detached.
If newRight had a parent, it is detached before being attached to the receiver.

attachParent(newParent, onLeft) newParent becomes the receiver's Parent.
If the receiver had a parent it is detached.
If onLeft is true the receiver becomes newParent's left subtree; otherwise it becomes newParent's right subtree.
If newParent had a subtree on that side it is detached.

What is the "receiver"?

Outline of Lecture

- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

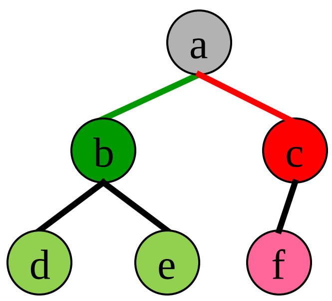
Implementation of ADT Binary Tree

- Determine the internal storage for both nodes and edges?

  - 1st attempt: List of lists.
    - Each subtree is a list with the root value the left subtree and the right subtree

  - 2nd attempt: Nodes and references
    - Each node is a node with two references pointing to two children if they exist. (similar to linked lists)

List of lists implementation



A tree is a triplet
[root,[left],[right]]

```
myTree =[ 'a',
              [ 'b',
                    ['d', [], []],
                    ['e' ,[], []]
              ],
              [ 'c',
                    ['f', [], []],
                    []
              ]
         ]
```

['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]

Implementing the Interface

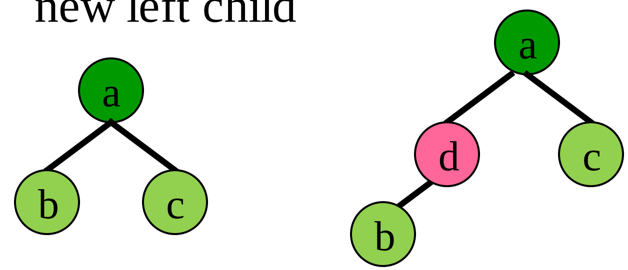
```
def BinaryTree(valueRoot):
    return [valueRoot, [],[] ]

def getLeft(root):
    # returns the left child of the current node
    return root[1]

def getRight(root):
    # returns the right child of the current node
    return root[2]

def getRootVal(root):
    #returns the object stored in the current node.
    return root[0]
```

```
def setRootVal(root,valueRoot):
    root[0]=valueRoot
```

Implementing the Interface

```
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

Make the original left subtree, if exists, the left child of the new left child

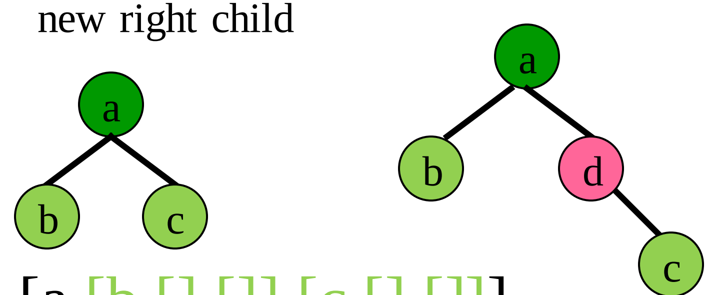


[a,[b,[],[]],[c,[],[]]]

[a,[d,[b,[],[]],[]],[c,[],[]]]

```
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch, [], []])
    return root
```

Make the original right subtree, if exists, the right child of the new right child



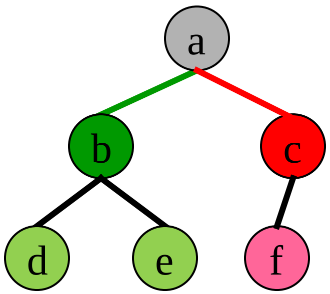
[a,[b,[],[]],[c,[],[]]]

[a,[b,[],[]], [d,[],[c,[],[]]]]

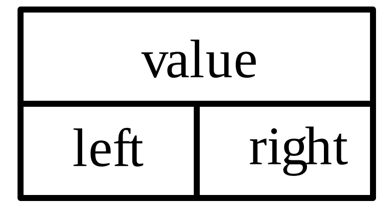
Limitations of 1st Implementation

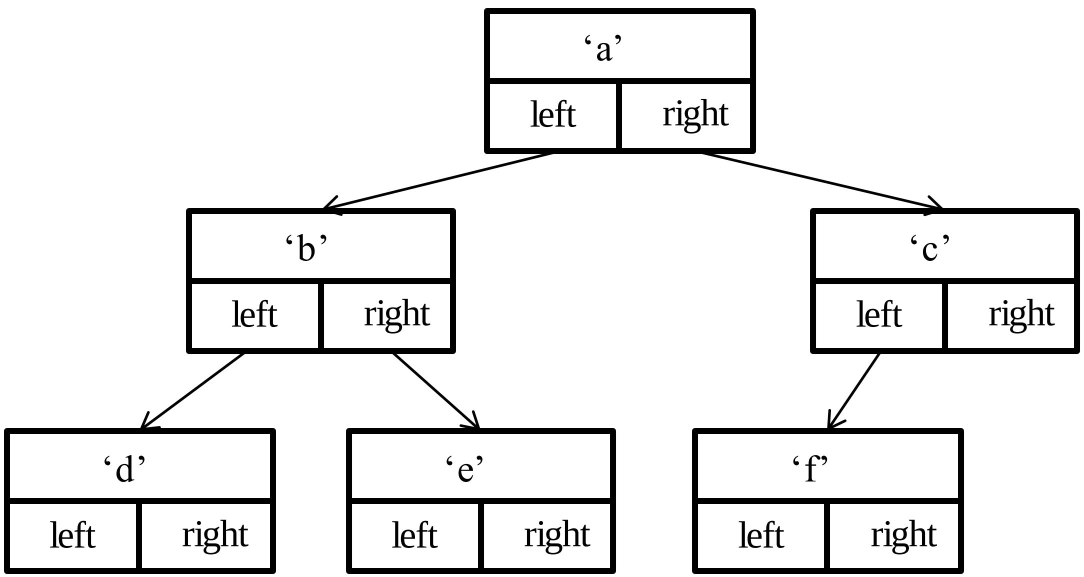
- Not very intuitive structure
- References are implicit
- There is an overhead due to many lists
- It is difficult to reorganize

Nodes and References implementation



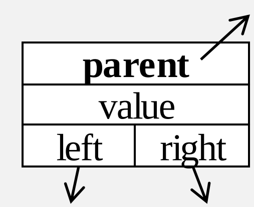
- Create a class BinaryTree where a node has a value and two references: a reference to the left child and a reference to the right child.





Some of the methods in the interface such as hasParent(), getParent(), detachFromParent(), attachParent(), may require the storage of the reference to the parent of a node.

We will NOT implement these

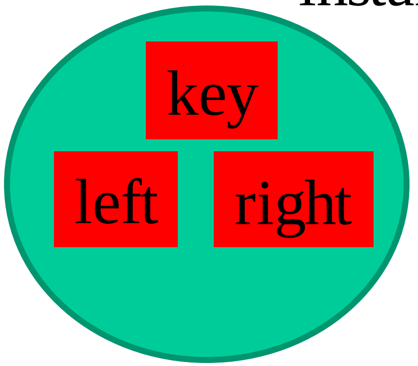


Implementing the Interface

```
class BinaryTree:
   def __init__(self,rootElement)
      self.key = rootElement
      self.left = None
      self.right = None

   def getLeft(self):
      return self.left

   def getRight(self):
      return self.right

   def getRootVal(self):
      return self.key
```

```
   def setRootVal(self,val):
      self.key=val
```

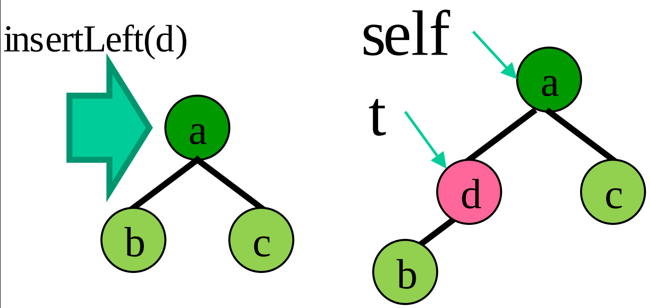
BinaryTree instance



Implementing the Interface

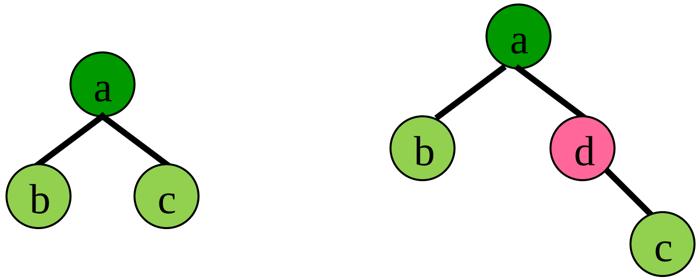
```
def insertLeft(self,newData):
    if self.left == None:
        self.left = BinaryTree(newData)
    else:
        t = BinaryTree(newData)
        t.left = self.left
        self.left = t
```

Make the original left subtree, if exists, the left child of the new left child




```
def insertRight(self,newData):
    if self.right == None:
        self.right = BinaryTree(newData)
    else:
        t = BinaryTree(newData)
        t.right = self.right
        self.right = t
```

Make the original right subtree, if exists, the right child of the new right child



Outline of Lecture

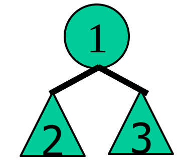
- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

Binary Tree Traversals

- There are four common binary tree traversals:

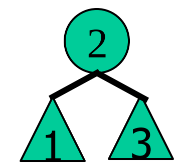
Preorder: process root then left subtree then right subtree

Root (Left) (Right)



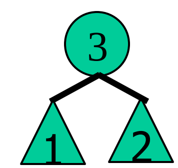
Inorder: process left subtree then root then right subtree

(Left) **Root** (Right)



Postorder: process left subtree then right subtree then root

(Left) (Right) **Root**



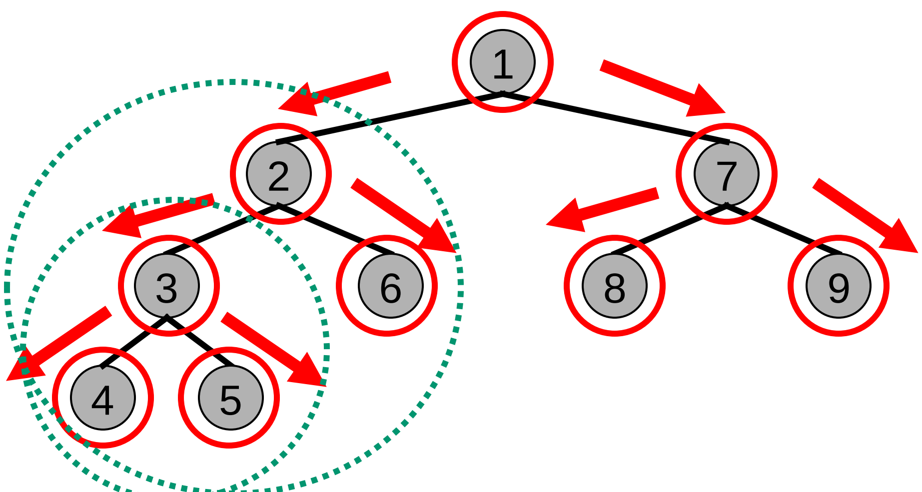
Levelorder: process nodes of level i, before processing nodes of level i + 1 (not supported in the solution shown here)

- Processing of left and right subtrees is done recursively

Binary Tree Traversals Example

Root (Left) (Right)

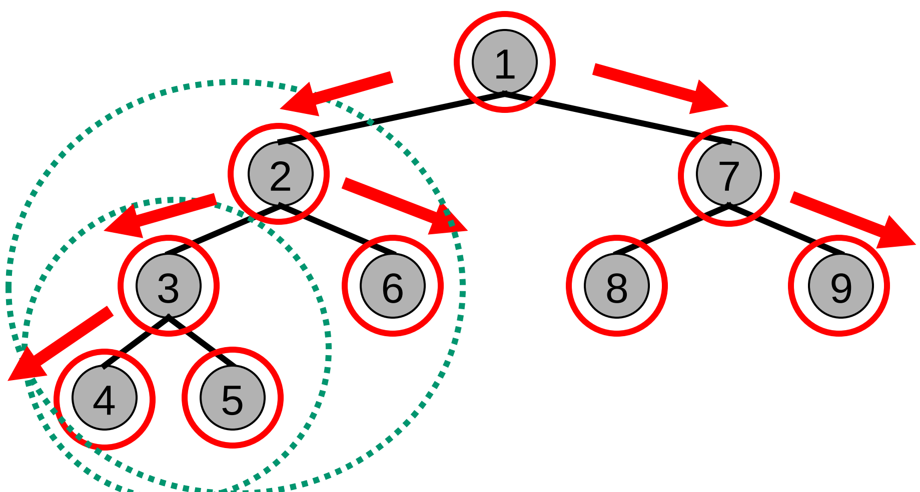
- Preorder: 1 2 3 4 5 6 7 8 9
- Inorder: 4 3 5 2 6 1 8 7 9
- Postorder: 4 5 3 6 2 8 9 7 1
- Levelorder: 1 2 7 3 6 8 9 4 5



Binary Tree Traversals Example

- Preorder: 1 2 3 4 5 6 7 8 9
- Inorder: 4 3 5 2 6 1 8 7 9
- Postorder: 4 5 3 6 2 8 9 7 1
- Levelorder: 1 2 7 3 6 8 9 4 5

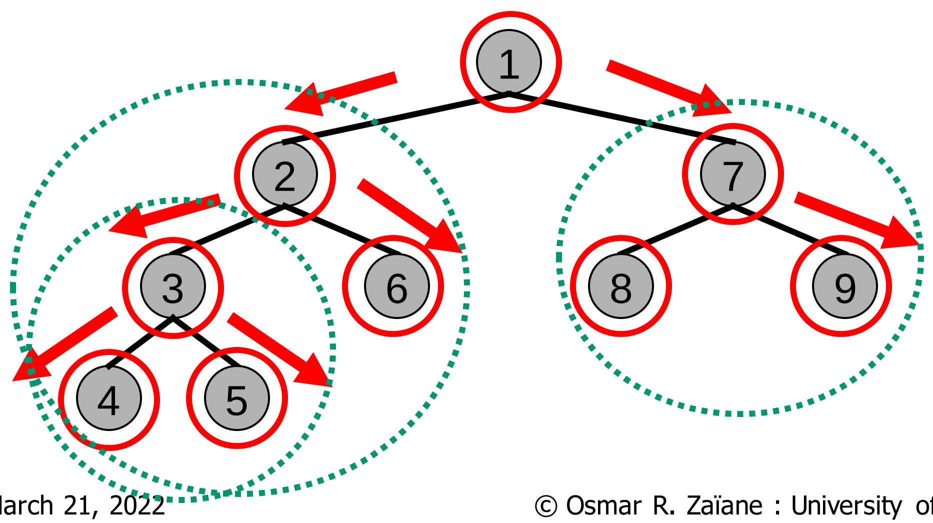
(Left) **Root** (Right)



Binary Tree Traversals Example

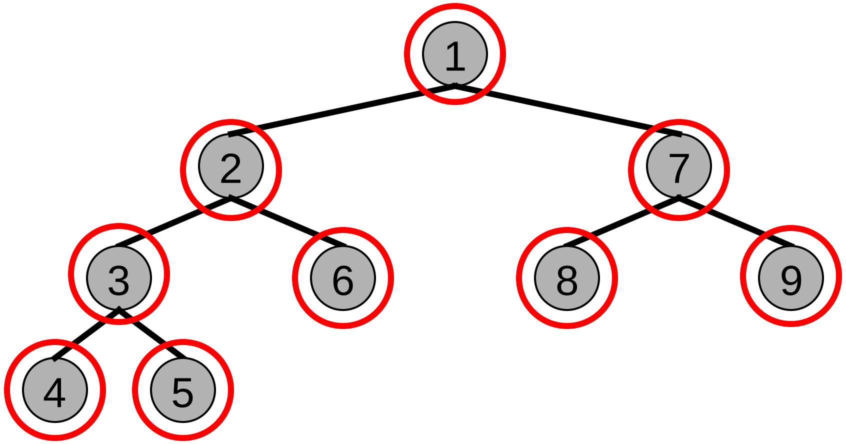
- Preorder: 1 2 3 4 5 6 7 8 9
- Inorder: 4 3 5 2 6 1 8 7 9
- Postorder: 4 5 3 6 2 8 9 7 1
- Levelorder: 1 2 7 3 6 8 9 4 5

(Left) (Right) **Root**



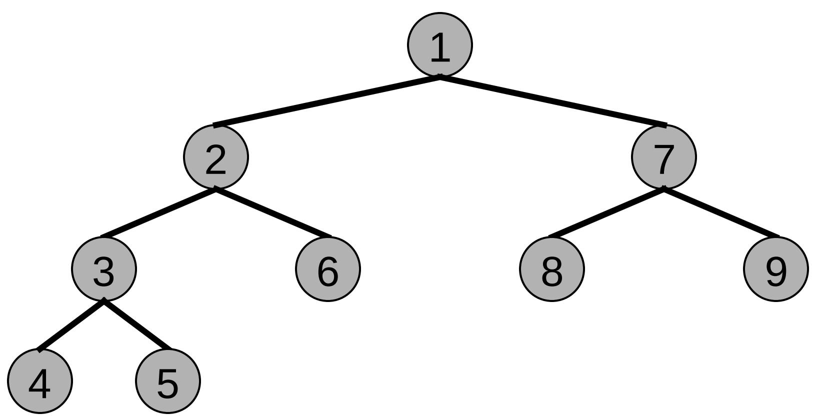
Binary Tree Traversals Example

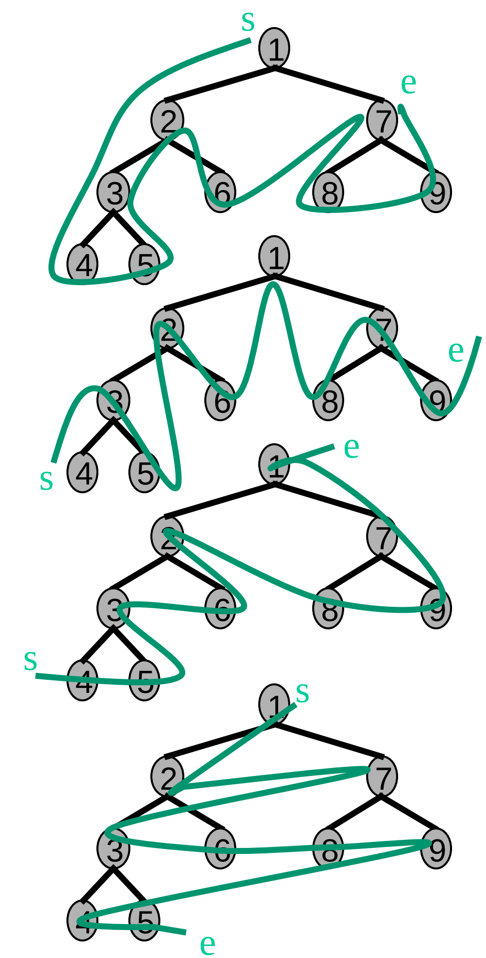
- Preorder: 1 2 3 4 5 6 7 8 9
- Inorder: 4 3 5 2 6 1 8 7 9
- Postorder: 4 5 3 6 2 8 9 7 1
- Levelorder: 1 2 7 3 6 8 9 4 5



Binary Tree Traversals Example

- Preorder: 1 2 3 4 5 6 7 8 9
- Inorder: 4 3 5 2 6 1 8 7 9
- Postorder: 4 5 3 6 2 8 9 7 1
- Levelorder: 1 2 7 3 6 8 9 4 5

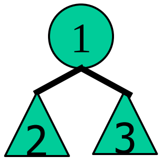




Implementing the Traversals

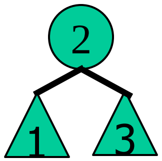
Processing here is "print"

```
def preorder(tree):
    if tree != None:
        print(tree.getRootVal())
        preorder(tree.getLeft())
        preorder(tree.getRight())
```



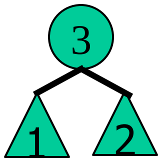
1. Process root
2. Process Left subtree
3. Process Right subtree

```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeft())
        print(tree.getRootVal())
        inorder(tree.getRight())
```



1. Process Left subtree
2. Process root
3. Process Right subtree

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeft())
        postorder(tree.getRight())
        print(tree.getRootVal())
```



1. Process Left subtree
2. Process Right subtree
3. Process root

Outline of Lecture

- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

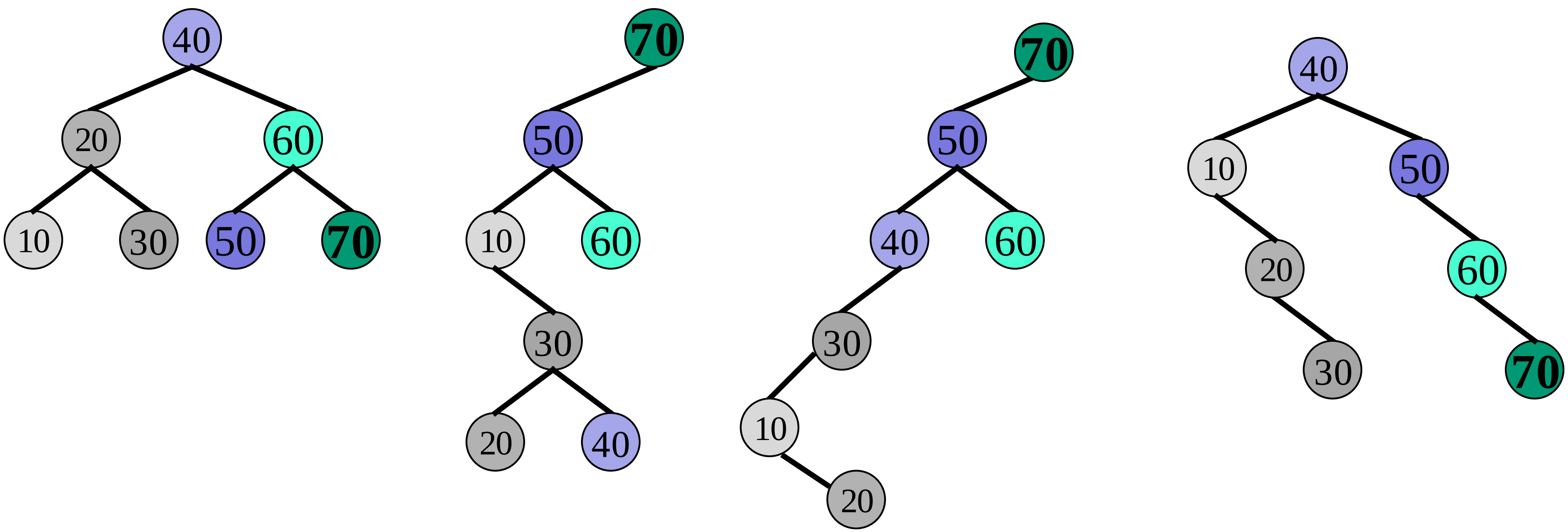
Binary Search Tree

- We want to combine the advantage of a binary search with the advantage of just fixing a few links during adding and removal to obtain an implementation called a BinarySearchTree where:
  - The time for finding an element is O(log n) comparisons due to a binary search.
  - The time for adding or removing an element is O(log n) comparisons to find it or its place and O(1) assignments to fix links.

- A binary search tree (BST) is a binary tree in which, for every node, the element in the node is greater than or equal to all the elements in the left subtree and less than or equal to all the elements in the right subtree.

Sort Order in BinarySearchTrees

- Many different BSTs can be formed from the same set of elements.

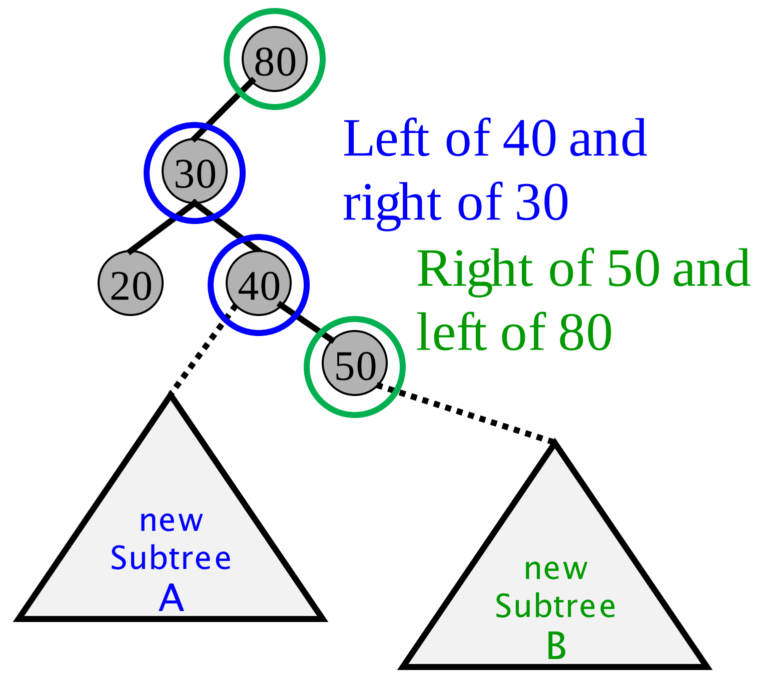


- **However, an inorder traversal always produces the elements in sorted order: 10, 20, 30, 40, 50, 60, 70.**

Attaching a subtree to a BST

- We must add preconditions to the “attach” methods to ensure the defining property of a BST is maintained. All the elements in the new subtree must lie in a range defined by all the ancestor elements and the position in the tree.

What range of elements are permitted in subtrees A and B ?



Subtree A: less than 40 and greater than 30

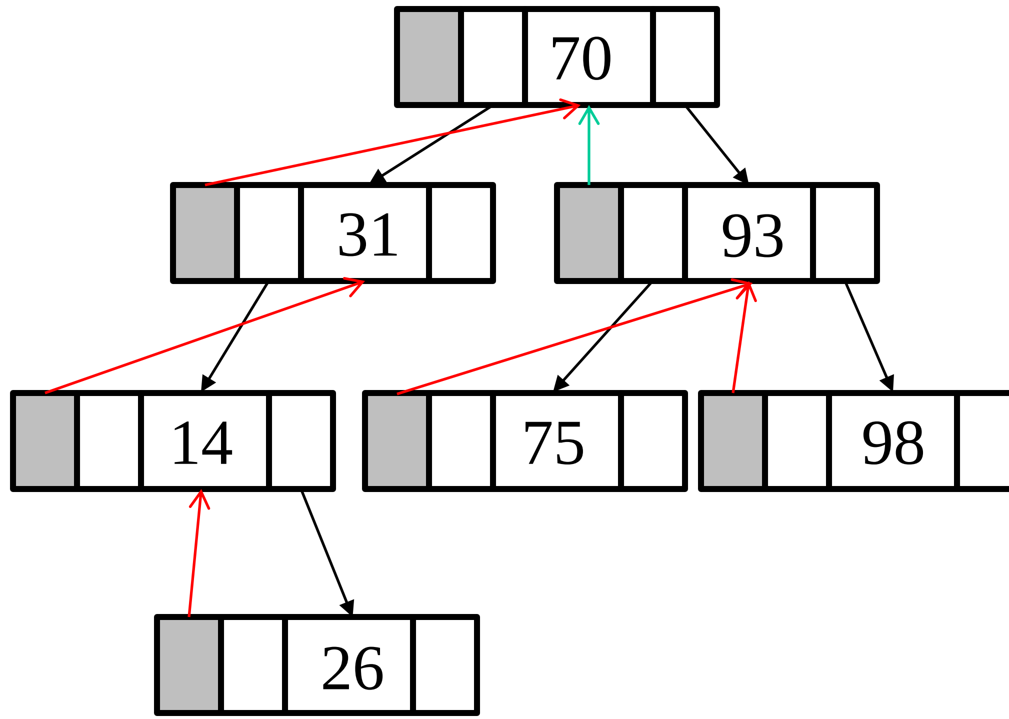
Subtree B: greater than 50 and less than 80

What do we store in BST

- We can store (key, value) pairs in the nodes.
- Each node in the tree represents a key
- The advantage is that:
  - We can traverse the tree to get an ordered list on the keys (inorder traversal)
  - We can search the value of a key in O(log(n))
  - We can insert a key, value pair in O(log(n))
  - We can delete a key in O(log(n))

Binary Search Tree

- All the keys in the left sub-tree are smaller than the root key
- All the keys in the right sub-tree are larger than the root key



In addition to left and right references, we can have a reference to the parent

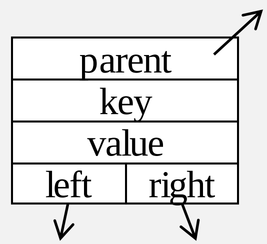
Interface to a binary Search Tree

- BinarySearchTree() Creates a new, empty data structure.
- put(key,value) Adds a new key-value pair to the tree; Replaces the old value of the key, if key exists.
- get(key) Returns the value stored in the tree for the given key if exists, or Returns None otherwise.
- delete(key) Deletes the key-value pair from the tree.
- getSize() Returns the number of key-value pairs stored in the tree
- getRoot() Returns the root node

We need a class for tree nodes

- Fields in a node? Key; Value; left and right Child; Parent

```
class TreeNode:
    def __init__(self,key,val,left=None,right=None,parent=None):
        self.__key = key
        self.__value=val
        self.__left = left
        self.__right = right
        self.__parent = parent

    def getKey(self):
        return self.__key

    def getValue(self):
        return self.__value
```

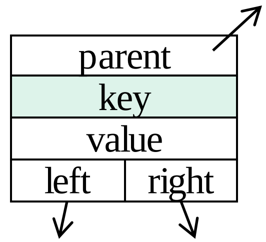


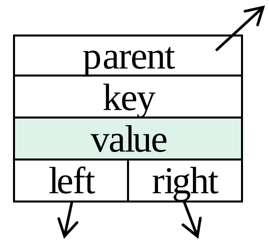

```
    def getLeft(self):
        return self.__left

    def getRight(self):
        return self.__right

    def getParent(self):
        return self.__parent
```

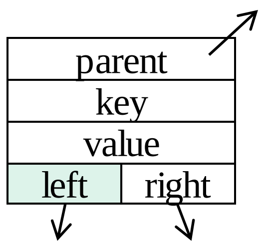
After the getters, the setters

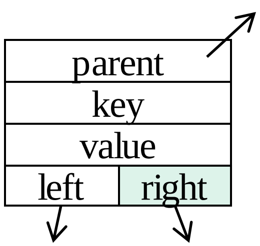
- Fields in a node? Key; Value; left and right Child; Parent

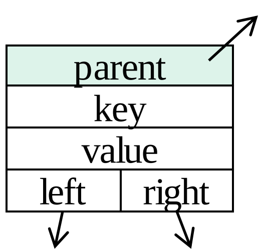
```
def setKey(self,key):
    self.__key=key

def setValue(self,val):
    self.__value=val
```









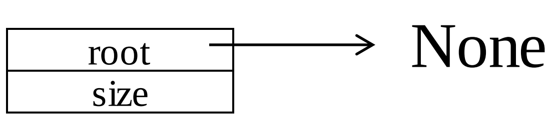


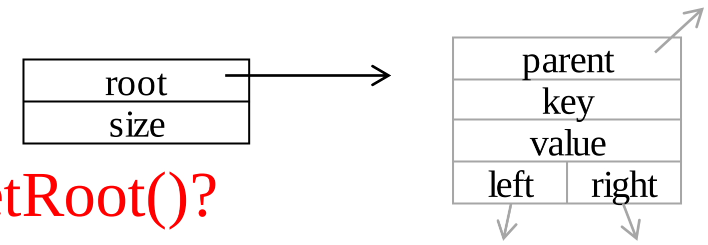

```
def setLeft(self,leftChild):
    self.__left = leftChild

def setRight(self,rightChild):
    self.__right = rightChild

def setParent(self,newParent):
    self.__parent=newParent
```

Implementing the BST

- We will cache the size and have a root
- We will implement: put(key,value); get(key); delete(key) and getSize()
- We will also implement an inorder traversal method

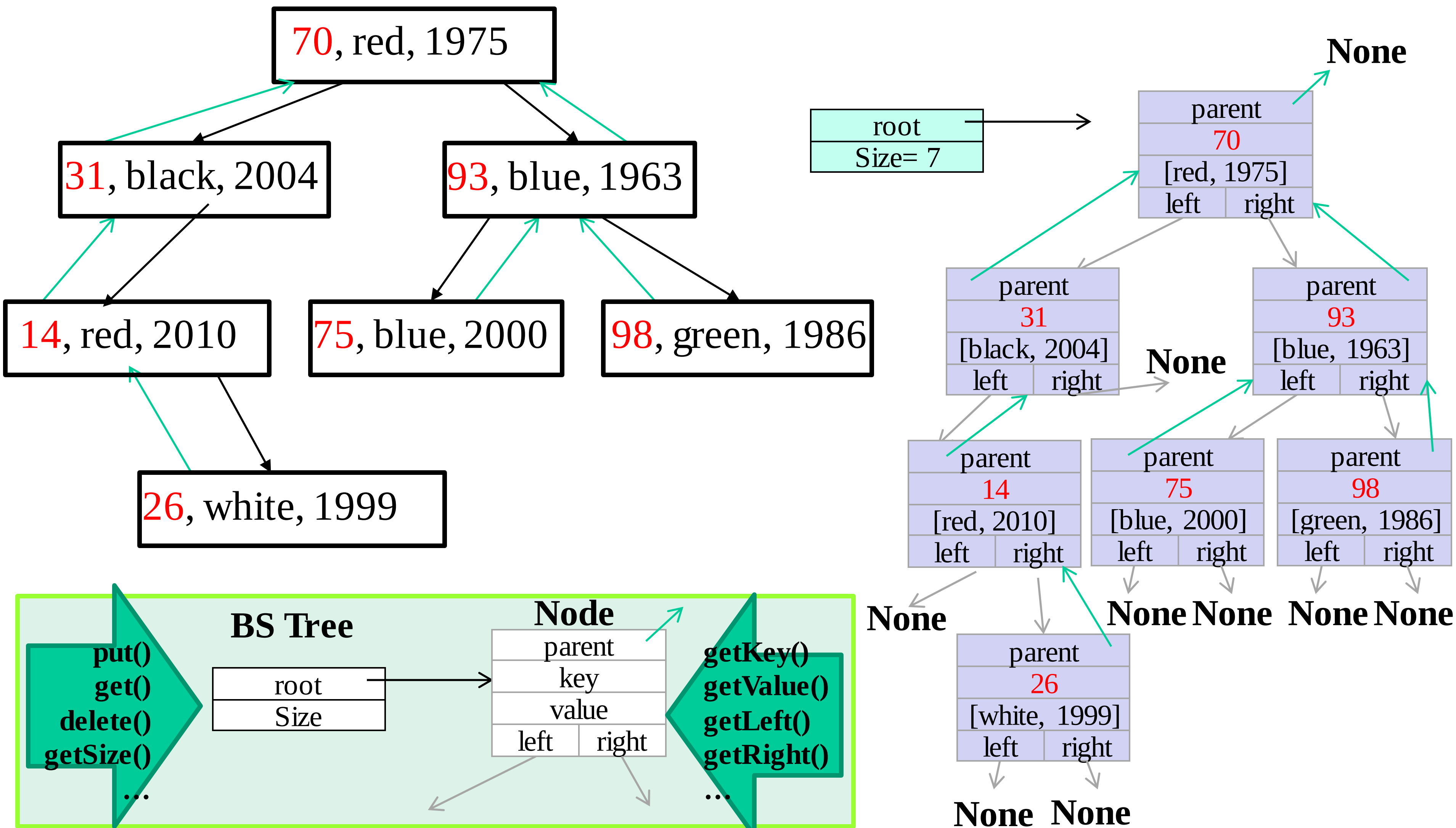
```
class BinarySearchTree:
    def __init__(self):
        self.__root = None
        self.__size =0

    def getSize(self):
        return self.__size

    def getRoot(self):
        return self.__root
```





Why don’t we have setSize() and setRoot()?

Trees and nodes

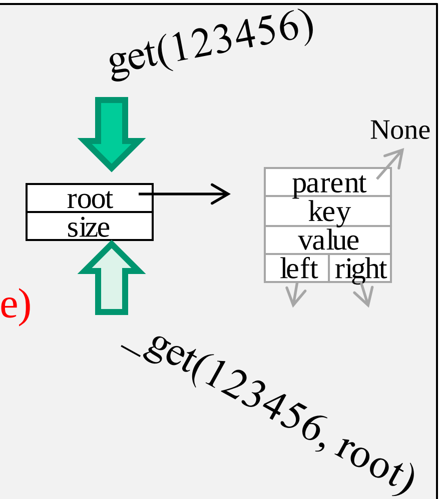


Get a value of a key

```
def get(self,key):
    # to retrieve the value associated with the given key
    return self._get(key,self.__root)
```

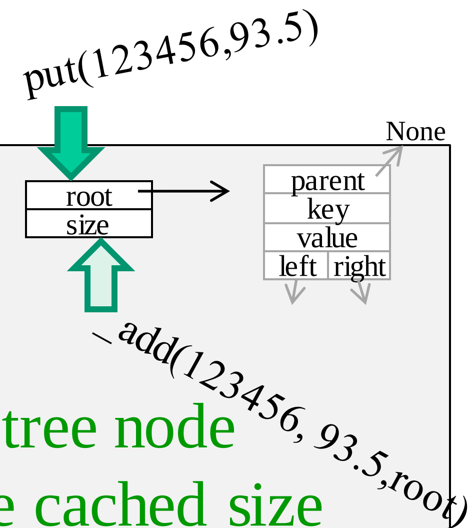
Reference to node(i.e. root of subtree)

```
def _get(self,key,currentNode):
    # helper function to get the key's value of a node
    if not currentNode:                    # the node is empty
        return None
    elif key == currentNode.getKey():   # the key is the one we look for
        return currentNode.getValue()
    elif key < currentNode.getKey():
        return self._get(key,currentNode.getLeft())      # search left
    else:
        return self._get(key,currentNode.getRight())    # search right
```



Add a key value pair

- If the root doesn't exist yet, we will create a node as the root and put this key value pair in it.
- If the root exists as a node, we will ask this node to add the new pair (with a helper method)




```python
def put(self,key,val):
    # to insert a key-value pair
    if not self.__root:                         # root is None
        self.__root = TreeNode(key,val)         # create a new tree node
        self.__size += 1                        # increment the cached size
    else:
        self._add(key,val,self.__root)          # call a helper method to do this
```

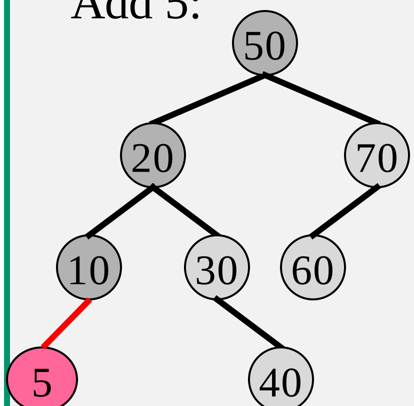
Helper method: Add a key value pair

```
def _add(self,key,val,currentNode):   # add at a given node
    if key < currentNode.getKey():
        if not currentNode.getLeft():    # left doesn't exist, create a node
            currentNode.setLeft(TreeNode(key,val,parent=currentNode))
            self.__size += 1
        else:
            self._add(key,val,currentNode.getLeft())   # add to the left
    elif key == currentNode.getKey():                  # key already exists
        currentNode.setValue(val)                      # update value
    else:
        if not currentNode.getRight():    # right doesn't exist, create a node
            currentNode.setRight(TreeNode(key,val,parent=currentNode))
            self.__size += 1
        else:
            self._add(key,val,currentNode.getRight()) # add to the right
```

Adding an Element to a BST

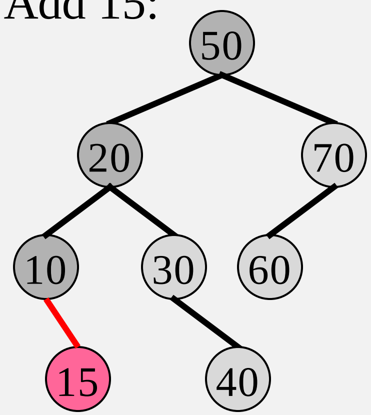
- An element is always added in a new leaf node.
- If the key already exists, update the value
- Let's focus on keys and do the following examples:

Add 5:



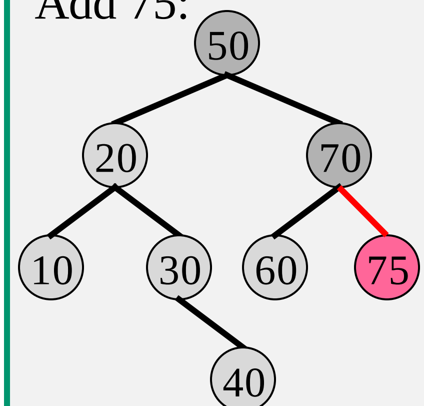
Root (50), left to 20
20, left to 10
Insert left to 10

Add 15:



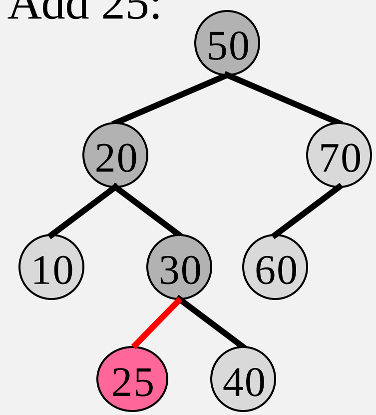
Root (50), left to 20
20, left to 10
Insert right to 10

Add 75:



Root (50), right to 70
Insert right to 70

Add 25:

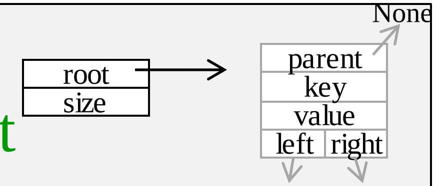


Root (50), left to 20
20, right to 30
Insert left to 30

- If keys are allowed to repeat in the BST, reaching the key, if the left subtree of that node is empty, insert it as the left child. Otherwise recursively insert it into the left subtree.

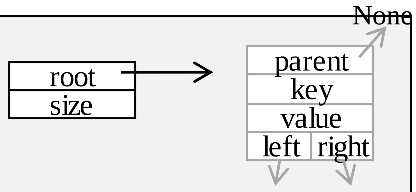
Removing a key value pair

- Locate the node with the key to be deleted – nodeToRemove




```
def delete(self,key):
    if self.__size > 1:    # There is more than just the root
        nodeToRemove = self._locate(key,self.__root)
        if nodeToRemove:    # we located the node to remove – not NONE
            self._remove(nodeToRemove)
            self.__size = self.__size-1
        else:
            raise KeyError('Error, key not in tree')
    elif self.__size == 1 and self.__root.getKey() == key:
        self.__root = None
        self.__size = self.__size - 1
    else:                           # we have one node and it is not the key
        raise KeyError('Error, key not in tree')
```

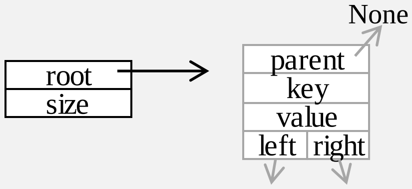
Locating a key value pair



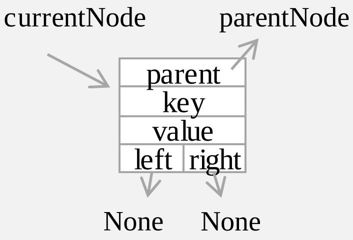

```
def _locate(self,key,currentNode):
    # returns the node that contains the key or None
    if not currentNode:                          # the node is empty
        return None
    elif key == currentNode.getKey():    # the key is the one we look for
        return currentNode
    elif key < currentNode.getKey():
        return self._locate(key,currentNode.getLeft())      # search left
    else:
        return self._locate(key,currentNode.getRight())     # search right
```

Helper method: remove a node

- The remove helper function when removing a node, it has 3 cases:
  - (1) Node has no children; (2) Node has 1 child; (3) Node has 2 children

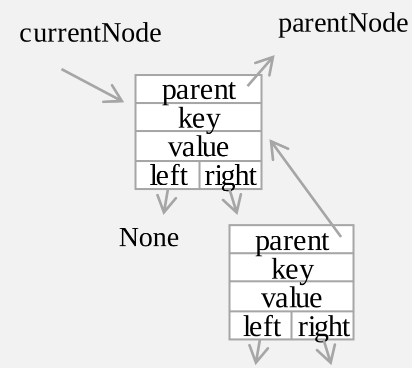


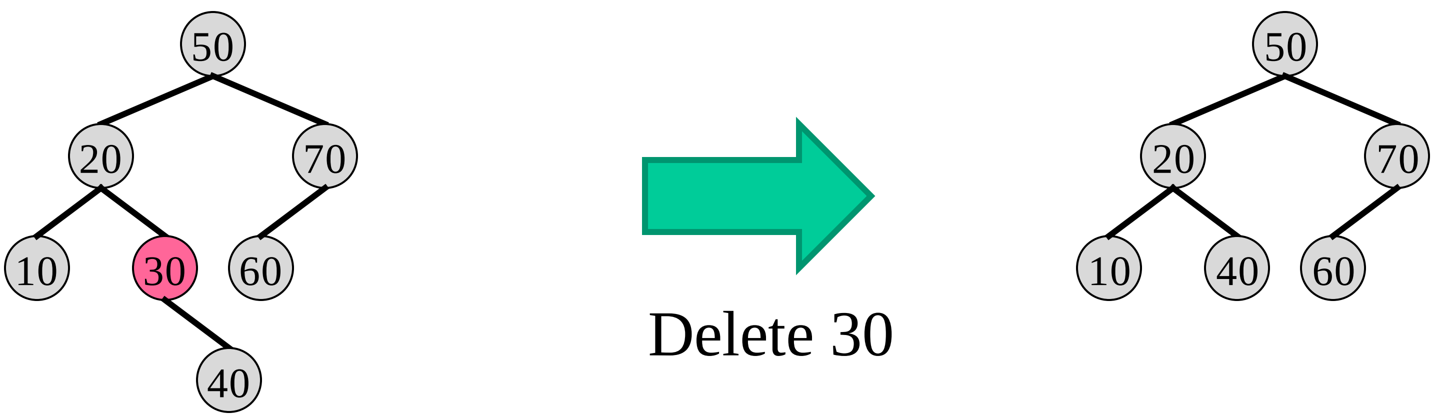

```
def _remove(self,currentNode):
    parentNode = currentNode.getParent()
    leftNode = currentNode.getLeft()
    rightNode = currentNode.getRight()

    # first case: the node to be deleted has no children
    if leftNode ==None and rightNode == None:
        if currentNode == parentNode.getLeft():
            parentNode.setLeft(None)
        else:
            parentNode.setRight(None)
```



Helper method: case 2 (one child)

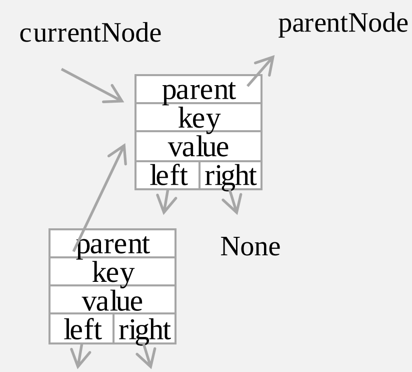
```
# second case: the node to be deleted has one child on the right
elif leftNode==None and rightNode:    # the one child is on the right
    if parentNode == None:            # the node to delete is the root
        self.__root = rightNode       # the unique child is now the root
    elif currentNode == parentNode.getLeft():
        parentNode.setLeft(rightNode)
    else:
        parentNode.setRight(rightNode)
    rightNode.setParent(parentNode)
```





Helper method: case 2 (one child)

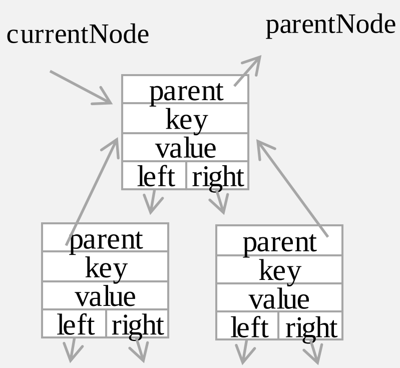
```
# second case: the node to be deleted has one child on the left
elif leftNode and rightNode==None:    # the one child is on the left
    if parentNode == None:             # the node to delete is the root
        self.__root = leftNode          # the unique child is now the root
    elif currentNode == parentNode.getLeft():
        parentNode.setLeft(leftNode)
    else:
        parentNode.setRight(leftNode)
    leftNode.setParent(parentNode)
```



Helper method: case 3 (two children)

- Find a successor: Go to the right subtree (all larger that current node) and find the smallest element
- Copy (swap) this smallest element to the current node then recursively remove it.

```
# third case: the node to be deleted has two children
else:
    # finding the smallest key (extreme left) of the right subtree
    swap = self._findSmallest(currentNote.getRight())
    # copying the key and value
    currentNode.setKey(swap.getKey())
    currentNode.setValue(swap.getValue())
    # removing the node we copied
    self._remove(swap)
```



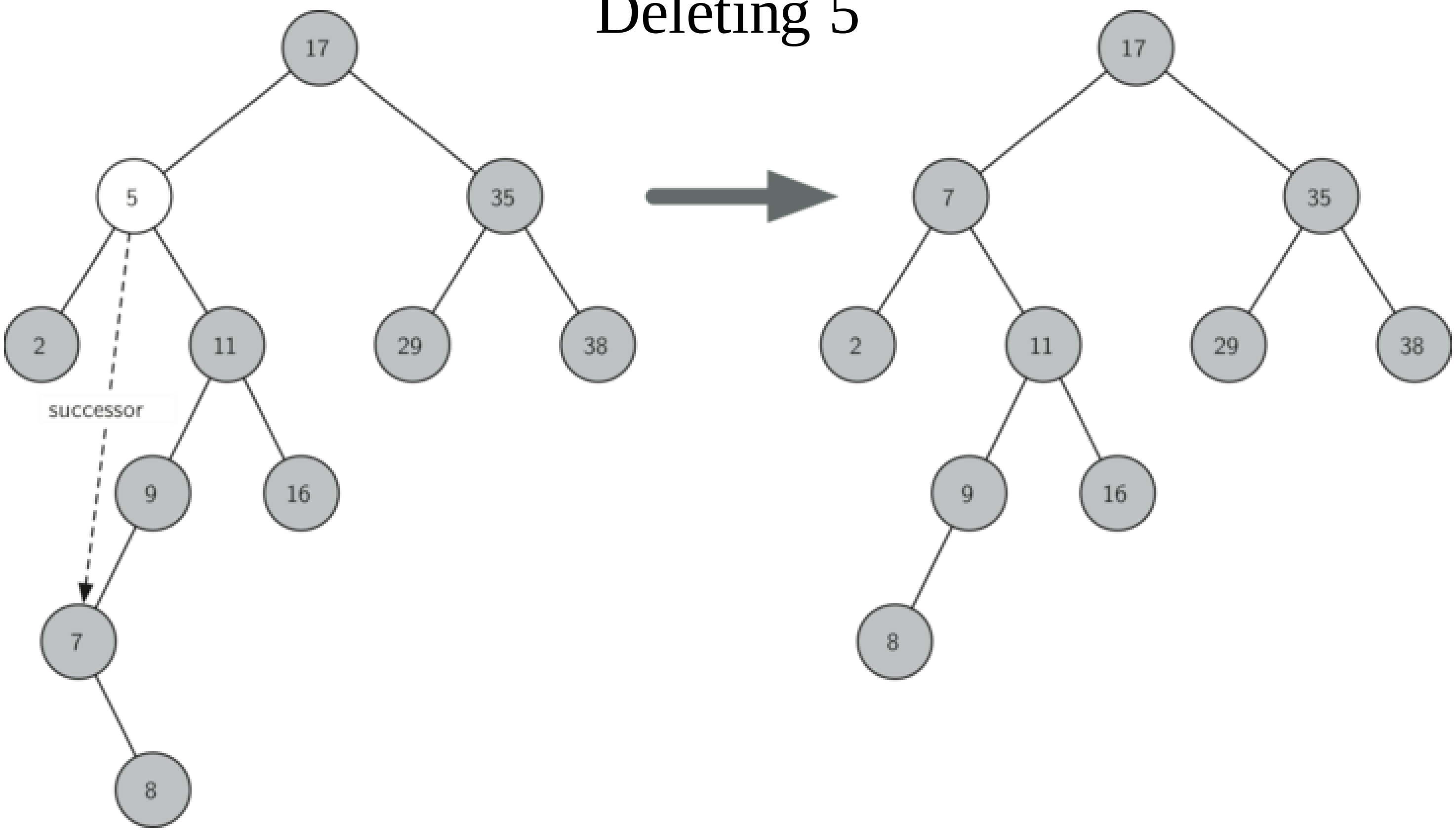
Successor: Finding the smallest

- Traverse the tree always to the left since the left child is smaller than root.
- If there is no left child then the current node is the smallest

```
def _findSmallest(self,currentNote):
    if currentNode.getLeft():          # there is someone smaller
        return self._findSmallest(currentNote.getLeft())
    else:
        return currentNode
```

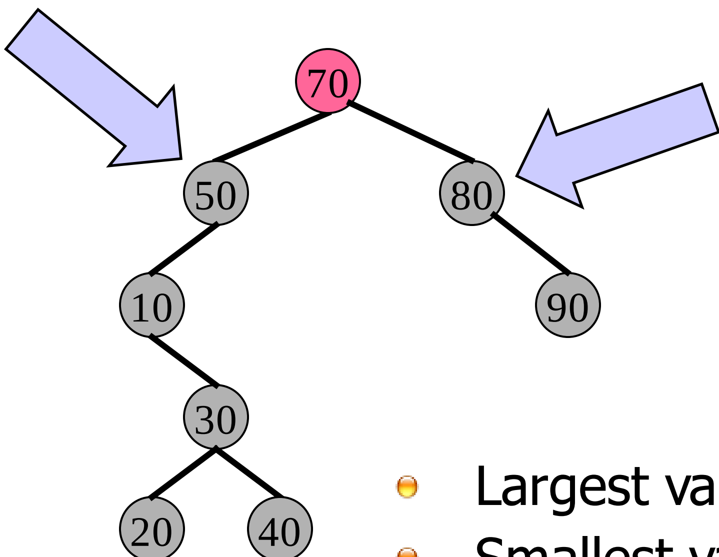
Example deleting node with 2 children

Deleting 5

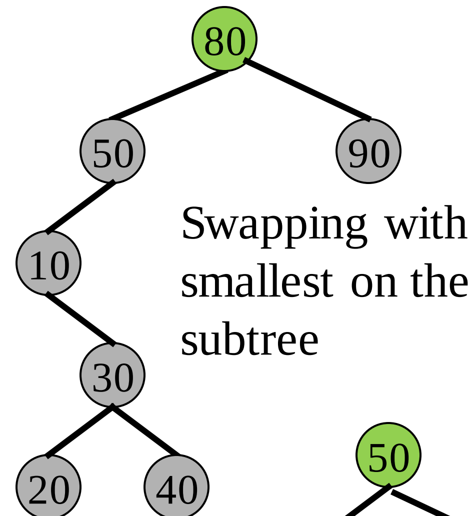


Another alternative for successor

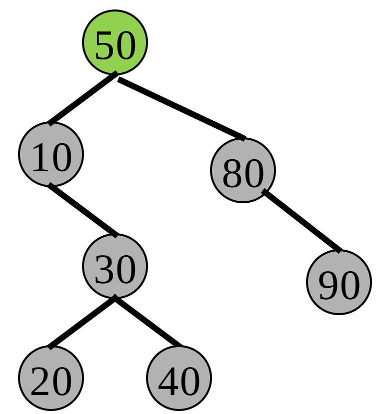
Which elements are suitable for replacing 70?



- Largest value in left subtree
- Smallest value in right subtree



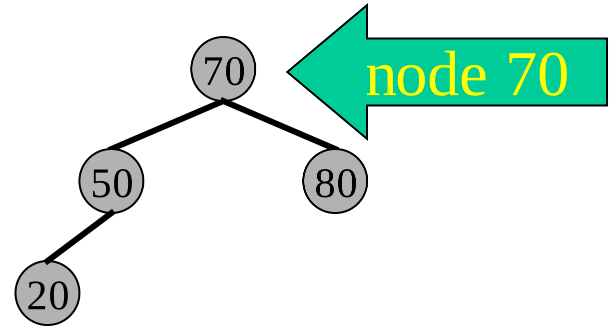
Swapping with 80: smallest on the right subtree



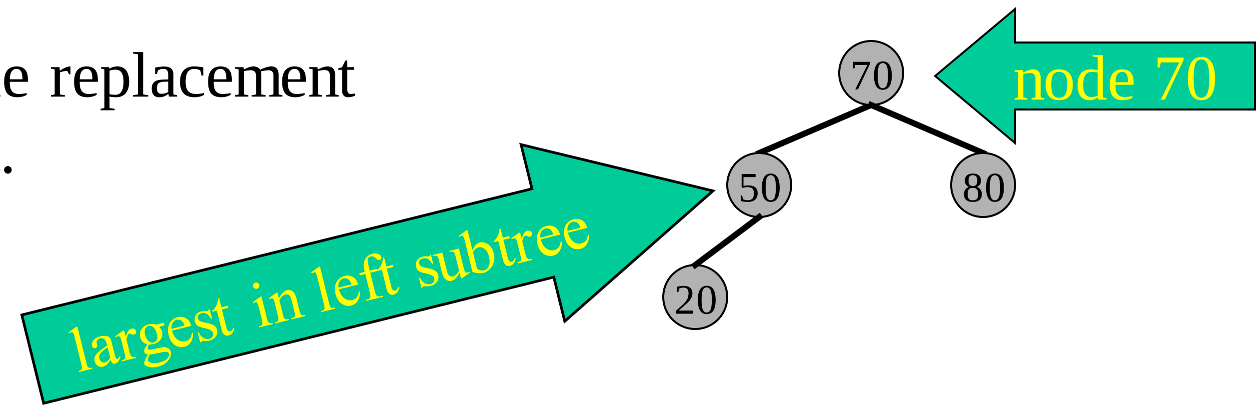
Swapping with 50: largest on the left subtree

Example: remove 70

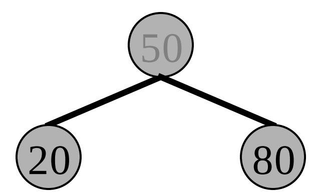
1. Find the node containing 70.



2. Find a suitable replacement below that node.



3. Replace 70 by the replacement.



4. Delete the node with the replacement

Traversing inorder

```
def inorder(self):
    self._inorder(self.__root)

def _inorder(self,node):
    # process the left (smaller than root);
    # then process root;
    # then process the right (larger than root)
    if node:                                    # check if node exists
        self._inorder(node.getLeft())
        print(node.getKey(),node.getValue()) # Processing is printing
        self._inorder(node.getRight())
```

Outline of Lecture

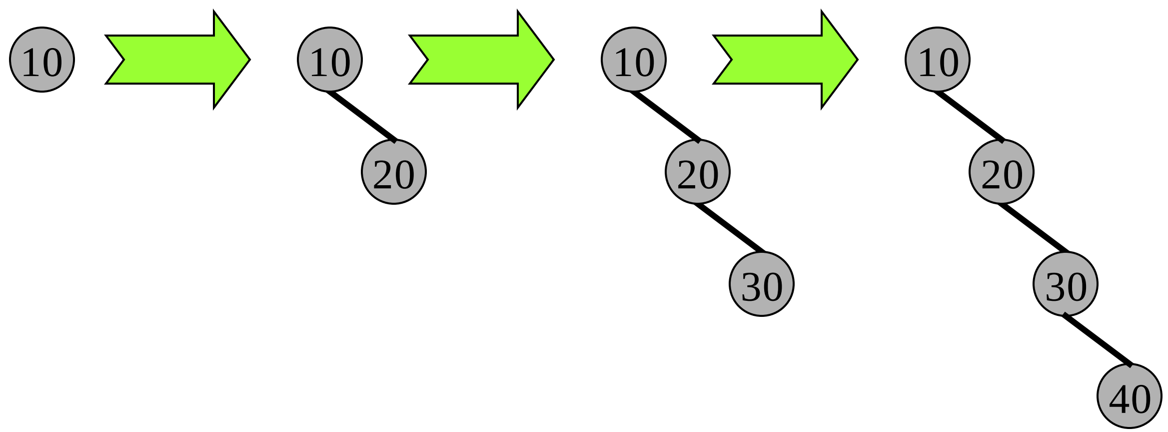
- Tree Terminology
- Binary Tree Interface
- Binary Tree Implementation
- Tree Traversals
- Binary Search Tree
- Balanced and unbalanced BST

Efficiency of Binary Search Trees

- Each of the time consuming operations is O(h), where h is the height of the tree.
- If the tree is fairly balanced, h = log(n) so the operations are O(log n)
- However, since many binary trees can be made from the same elements, we could be unlucky when we make our tree and have a tree with height h = n, which is essentially a linked list and the search time is O(n).
- What insertion orders cause problems?

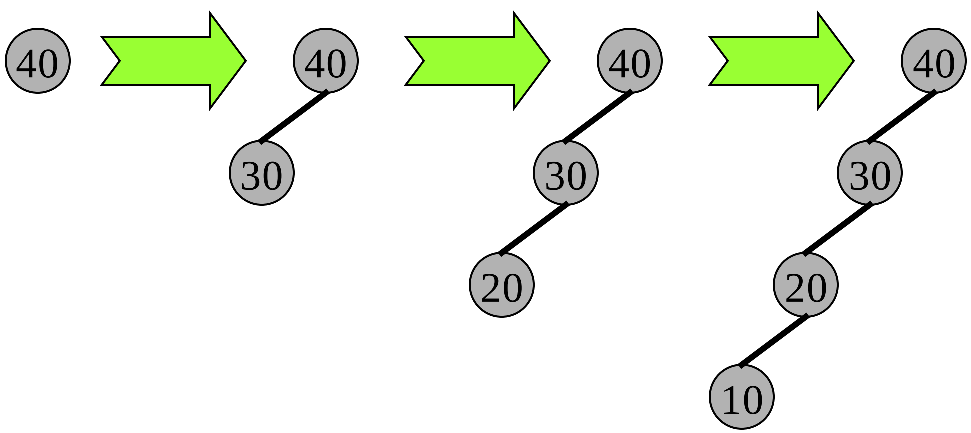
Unbalanced Binary Search Trees 1

- Look at a tree with insertion order: 10, 20, 30, 40:



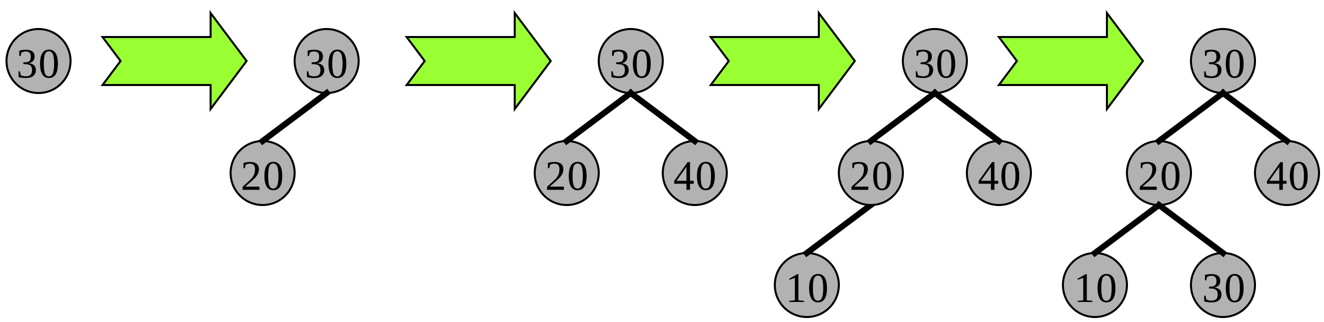
Unbalanced Binary Search Trees 2

- Look at a tree with insertion order: 40, 30, 20, 10:



Balanced Binary Search Trees

- Look at a tree with insertion order: 30, 20, 40, 10, 30:



Balancing Binary Search Trees

- There are sophisticated algorithms for balancing binary search trees, but we won't cover them in this course.
- However, if your BST becomes unbalanced, here is a simple approach:
  - Traverse your BST and put each element into an array.
  - Create a new BST and insert the elements from the array into the new BST, **in a random order**.
- This approach can also be used as an efficient sort:
  - Traverse your unsorted container (in any order), putting the elements into a BST.
  - Traverse the BST (inorder) and put the elements back into the container.
  - Since adding an element to a BST is O(log n) and there are n elements to add, the insertion part of the algorithm is O(n log n).
  - Since the final inorder traversal is O(n), the total sort algorithm complexity is O(n log n)
  - What would be the space complexity?