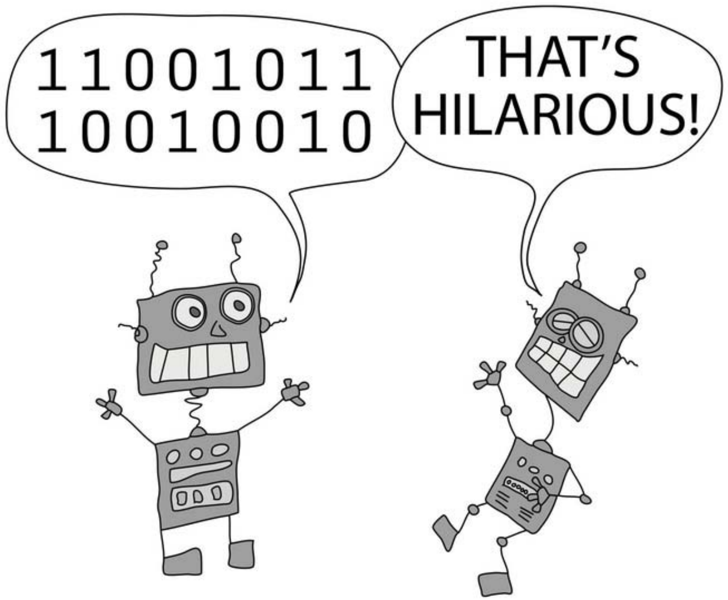
CompSci 105
Lecture 34 - 35 Contents

Binary Search Trees

Textbook: Chapter 6



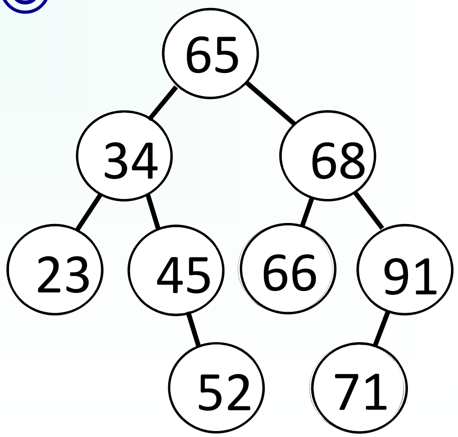
Trees can be very efficient

Trees are efficient. There are many algorithms which work on trees in O(log n) time.
Usually efficiency depends on the height of the tree.

We want to make use of this efficiency and use binary trees for searching / sorting etc. – how can we do this?

OBSERVATION: For a sorted (ordered) list we could very efficiently find a key using a divide and conquer technique.
IDEA: Design trees which define an order ☺

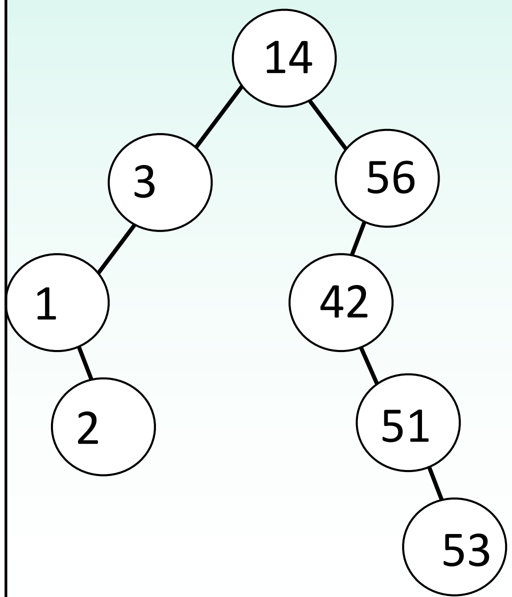
23	34	45	52	65	66	68	71	91

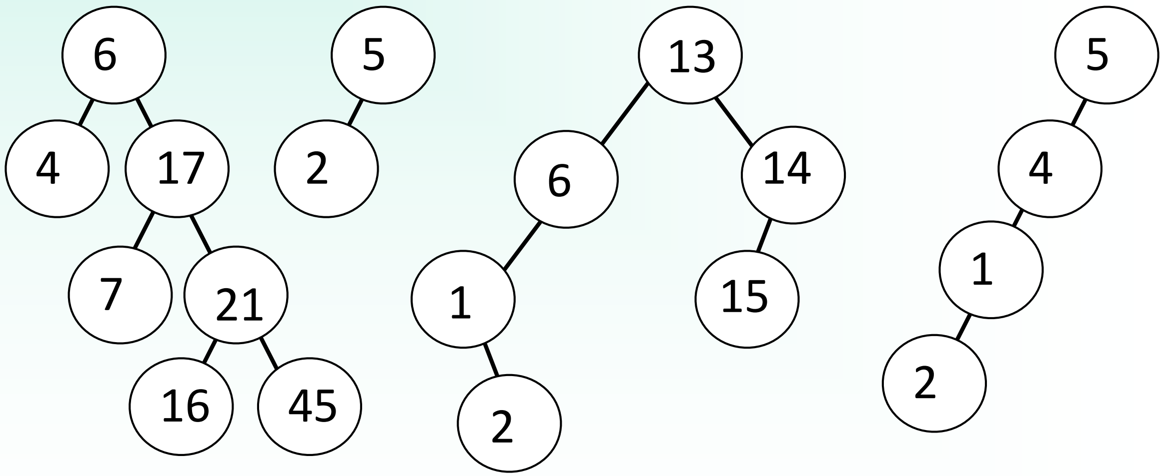


Binary search trees

Binary search trees are trees which have the following properties:

- For all nodes the values in the left subtree of that node are smaller than the value of the node
- For all nodes the values in the right subtree of that node are greater than the value of the node





Which of the above trees are binary search trees?

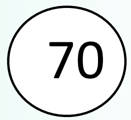
Binary search trees - insert

To demonstrate, we add a list of elements in the order they occur and ALWAYS MAINTAIN THE BINARY SEARCH TREE PROPERTY. For example, the following list:

70, 31, 93, 94, 14, 23, 73

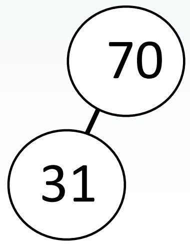
70, 31, 93, 94, 14, 23, 73

↑



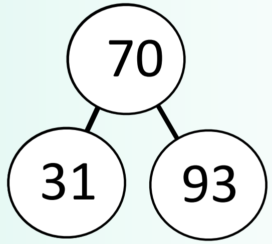
70, **31**, 93, 94, 14, 23, 73

↑

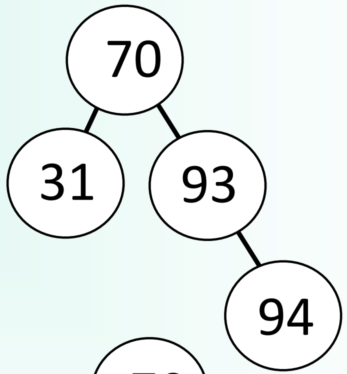


Binary search trees - insert

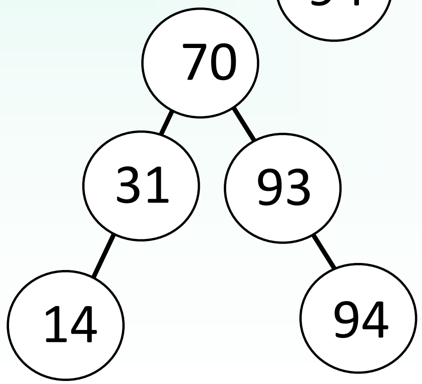
70, 31, **93**, 94, 14, 23, 73



70, 31, 93, **94**, 14, 23, 73



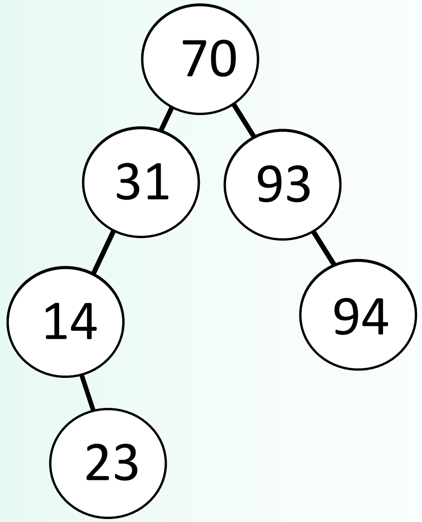
70, 31, 93, 94, **14**, 23, 73



Binary search trees - insert

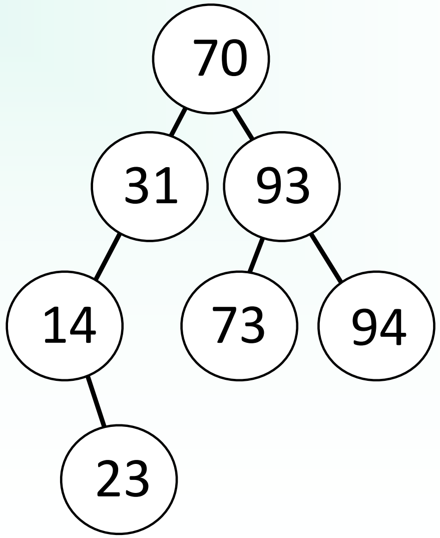
70, 31, 93, 94, 14, **23**, 73

↑



70, 31, 93, 94, 14, 23, **73**

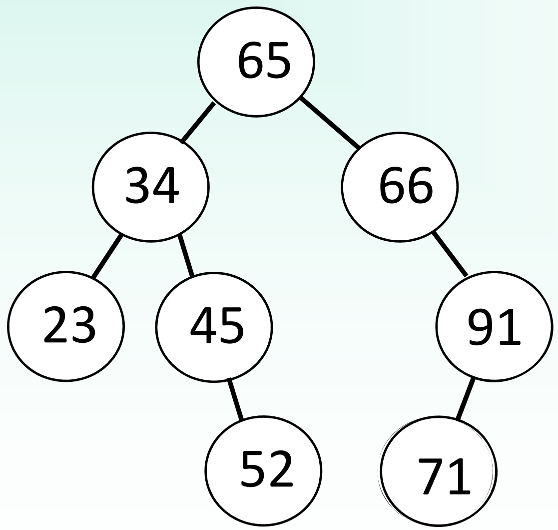
↑



Adding elements to a binary search trees

Create a binary search tree by adding the following values in the order given:

65 34 66 91 23 45 71 52



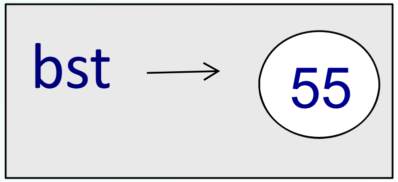
Binary search trees - code

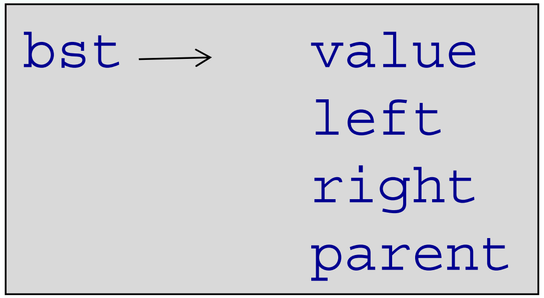

```
class BST:
   def __init__(self, value, parent=None):
      self.value = value
      self.left = None
      self.right = None
      self.parent = parent
```

We just use a single value in each node. Just the key.

Some jobs are easier if we have a reference to the parent node.

```
from BinarySearchTree import BST

def main():
   bst = BST(55)
main()
```





Binary search trees – book code


```
class BST:
  def __init__(self, key, value, … parent=None):
    self.key = key
    self.payload = value
    self.left = None
    self.right = None
    self.parent = parent

  def put(self, key, val):
    …
  def get(self, key):
    …
```

The book code uses a key and a payload.

We don't – trivial extension, more readable without

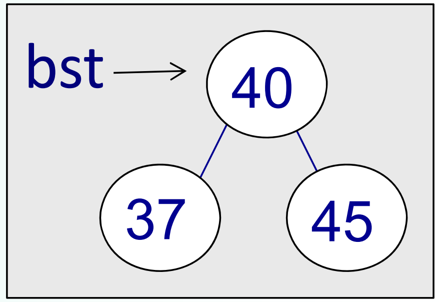
Binary search trees – insert code


```
class BST:
    …
  def insert(self, value):
    if value == self.value:
      return
    elif value < self.value:
      if self.left:
        self.left.insert(value)
      else:
        self.left = BST(value, parent=self)
    else:
      if self.right:
        self.right.insert(value)
      else:
        self.right = BST(value, parent=self)
```

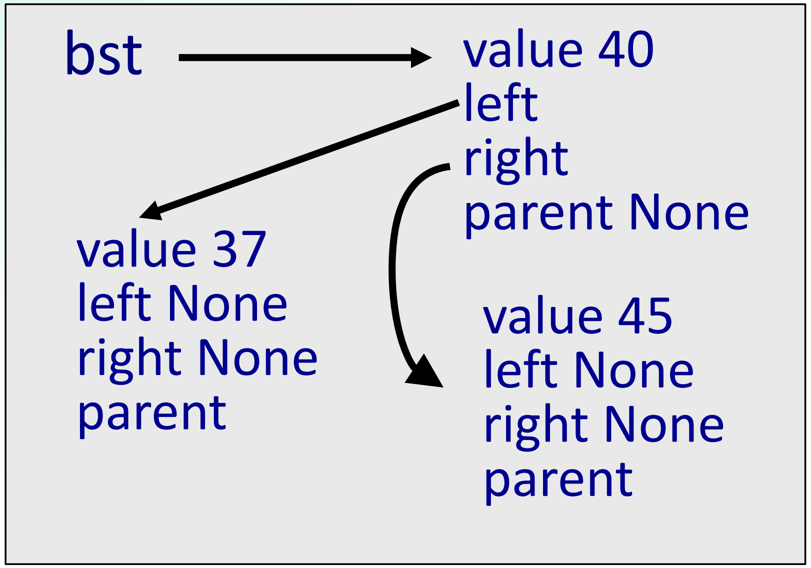
We are not allowing duplicates – if key exist already, insert does nothing

Binary search trees – insert code


```
class BST:
   def __init__(self, value, parent=None):
      …
   def insert(self, value):
      …
```



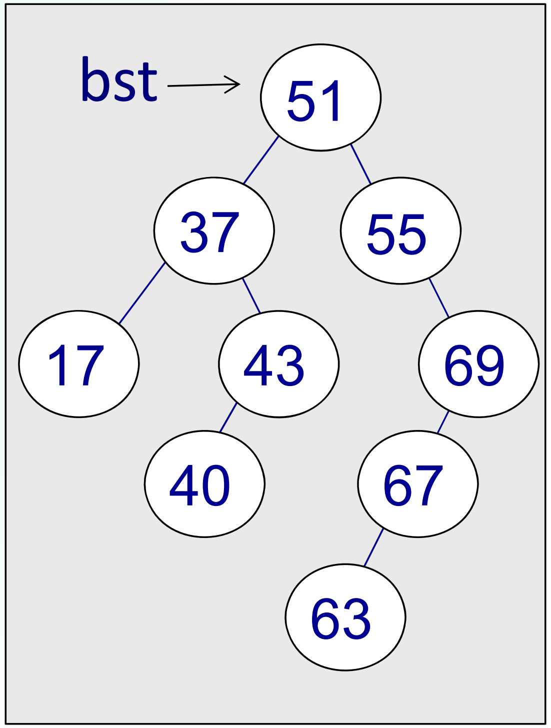

```
…

def main():
    bst = BST(40)
    bst.insert(37)
    bst.insert(45)

main()
```



Binary search trees – locate code


```
class BST:
    …
  def locate(self, value):
    if value == self.value:
        return self
    elif value < self.value and self.left:
        return self.left.locate(value)
    elif value > self.value and self.right:
*       return self.right.locate(value)
    else:
        return None
```

Finding a node in the binary search tree.



If I do a bst.locate(67) how many times is the line of code marked '*' executed.

Make a call to bst.locate(??) which causes the greatest number of comparisons. How many comparisons?

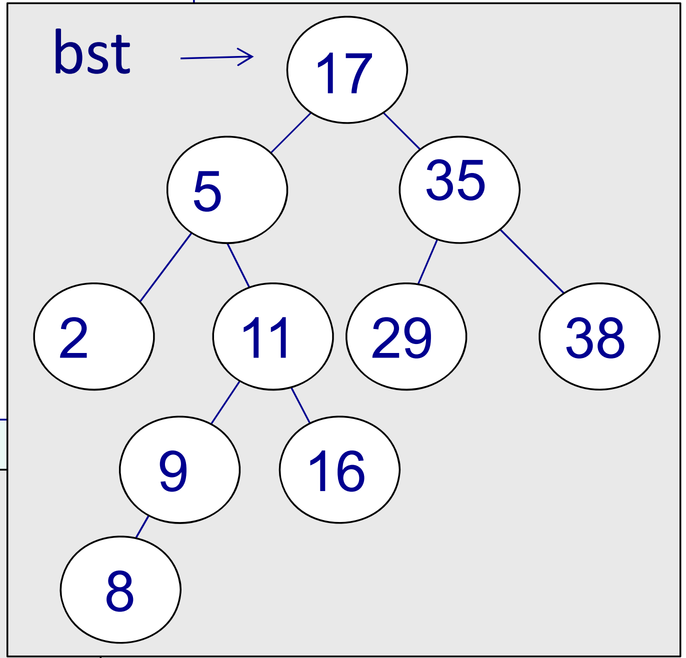
Binary search trees - code

Get a string representation of the tree.

```
class BST:
   def __str__(self):
      """Return a BST string representation"""
      return self.get_string(0)

   def get_string(self, spaces):
      info = ' ' * spaces + str(self.value)
      if self.left:
         info += '\n(l)' + self.left.get_string(spaces + 4)
      if self.right:
         info += '\n(r)' + self.right.get_string(spaces + 4)

      return info
```

Binary search trees - code


```
class BST:
   def __str__(self):
      return self.get_string(0)
   def get_string(self, spaces):
      info = ' ' * spaces + str(self.value)
      ...
```




```
...
def main():
    bst = BST(17)
    bst.insert(35)
    ...
    print(bst)
main()
```

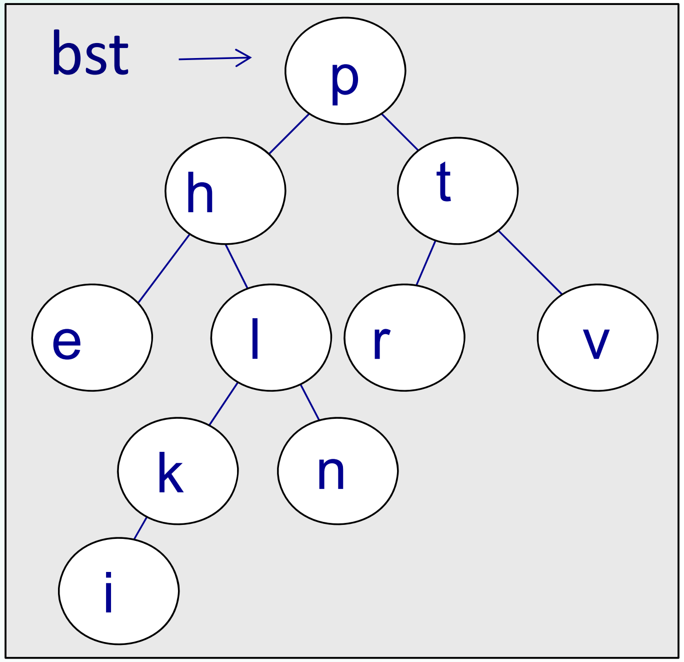
```
17
(l)    5
(l)         2
(r)         11
(l)              9
(l)                   8
(r)              16
(r)    35
(l)         29
(r)         38
```

Traversing trees - level order

The nodes of the tree can be traversed in different orders.

Level order visits the tree:
left to right, level by level.

p h t e l r v k n i



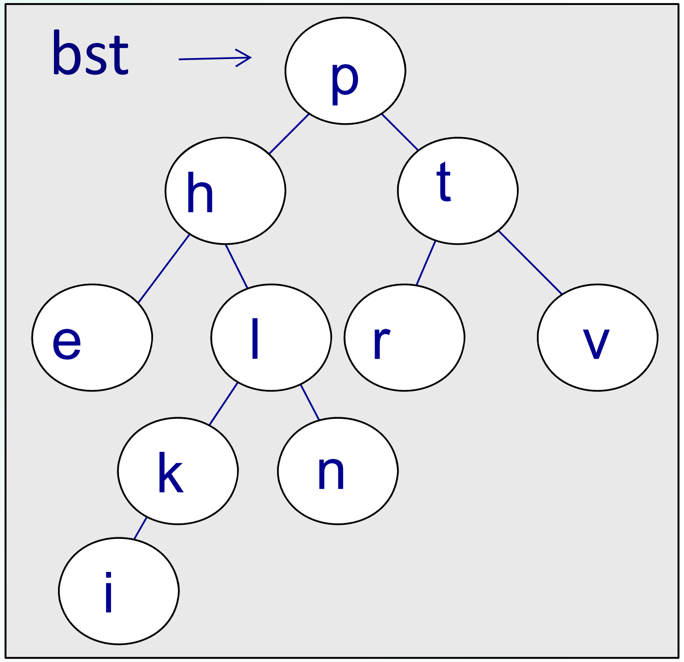
Traversing trees – inorder

The nodes of the tree can be traversed in different orders.

inorder visits the tree:

- left
- node
- right

e h i k l n p r t v



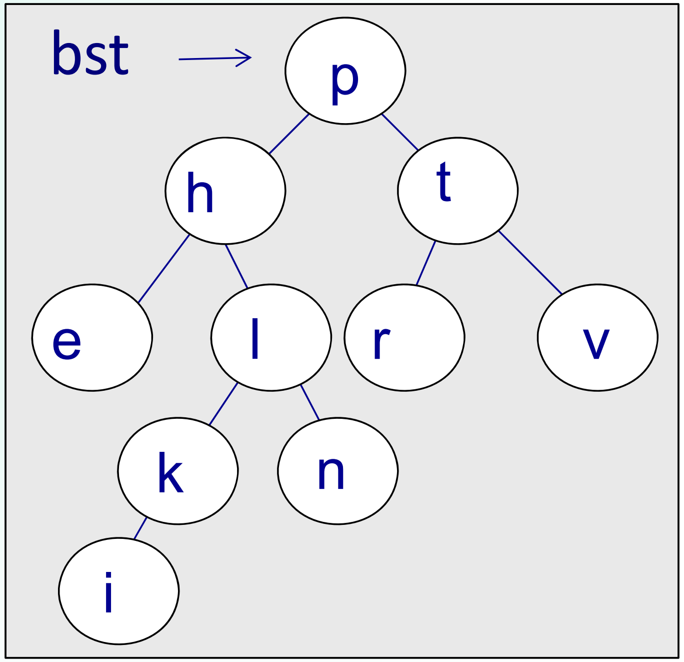
Traversing trees – postorder

The nodes of the tree can be traversed in different orders.

postorder visits the tree,

- left
- right
- node

e i k n l h r v t p

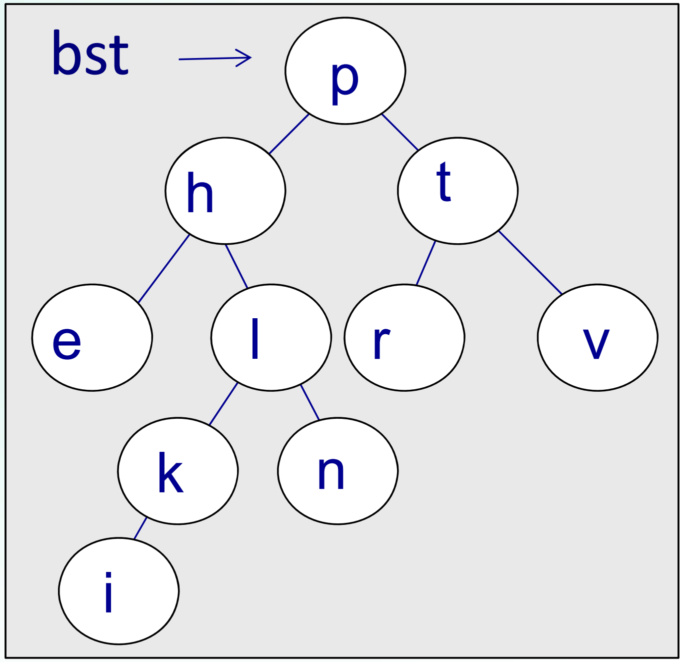


Traversing trees – preorder

The nodes of the tree can be traversed in different orders.

preorder visits the tree,
- node
- left
- right

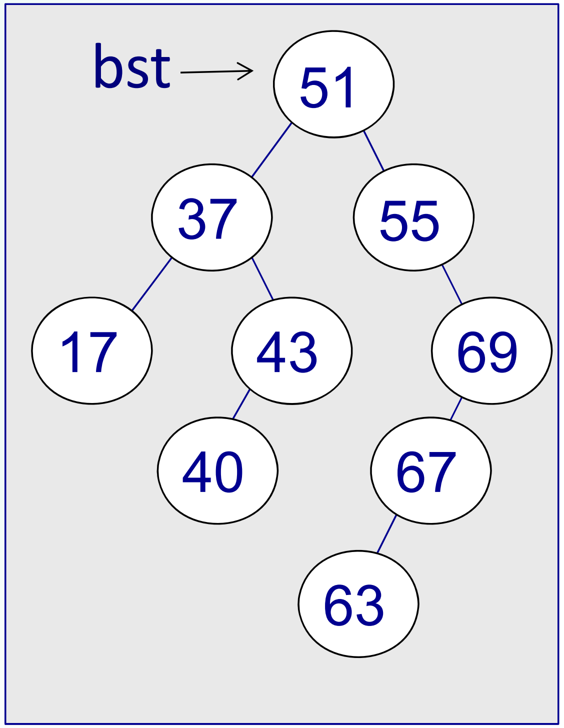
p h e l k i n t r v



Binary search trees – inorder string


```
class BST:
    ...
  def inorder(self):
     info = ""
     if self.left:
        info += self.left.inorder()
     info += str(self.value) + " "
     if self.right:
        info += self.right.inorder()
     return info
```

Get the inorder traversal string.




```
...
def main():
   bst = BST(51)
   bst.insert(35)
   ...
   print(bst.inorder())
```

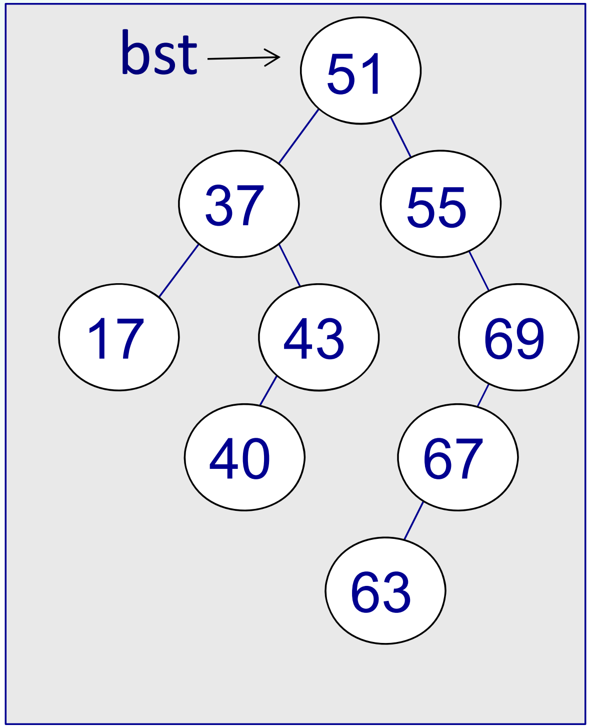
17 37 40 43 51 55 63 67 69

Binary search trees – from lists


```
class BST:
    def __init__ (self, ... ):
        ...
```

```
...
def create_from_list(a_list):
    bst = BST(a_list[0])
    for i in range(1, len(a_list)):
        bst.insert(a_list[i])
    return bst
def main():
    a_list = [          ???          ]
    bst = create_from_list(a_list)
    print(bst.inorder())
main()
```

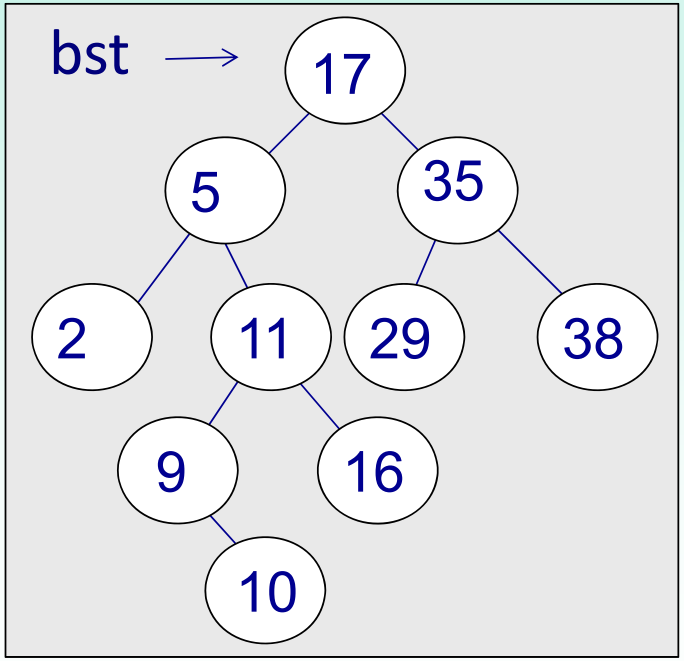
Complete the list which will create the tree below:



Binary search trees – deleting

Deleting nodes is a little bit trickier than inserting

We have to maintain the binary search tree property



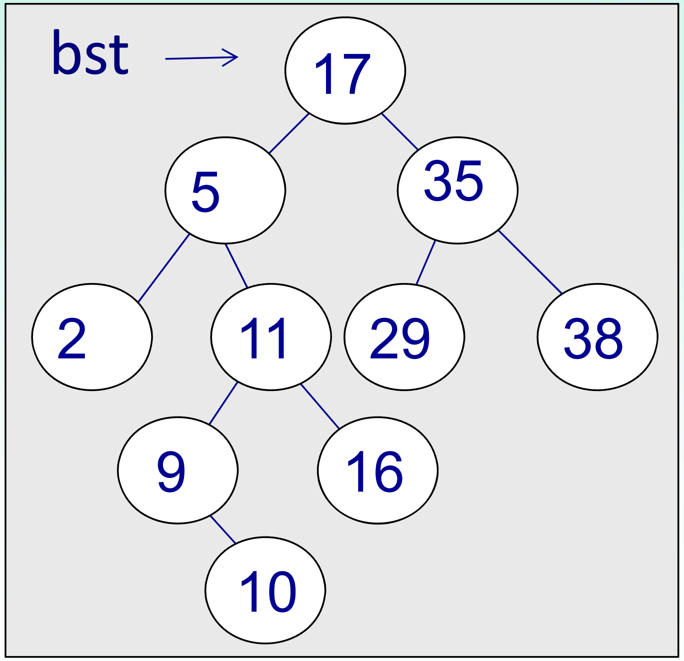
Three cases to consider:

```
...
def main():
    bst = create_from_list([ ...  ])
    bst = bst.delete(16)      case1
    bst = bst.delete(9)       case2
    bst = bst.delete(5)       case3
main()
```

Remember: we also have to think of the parent variable.

BST deleting – no children

CASE 1: deleting a node with no children



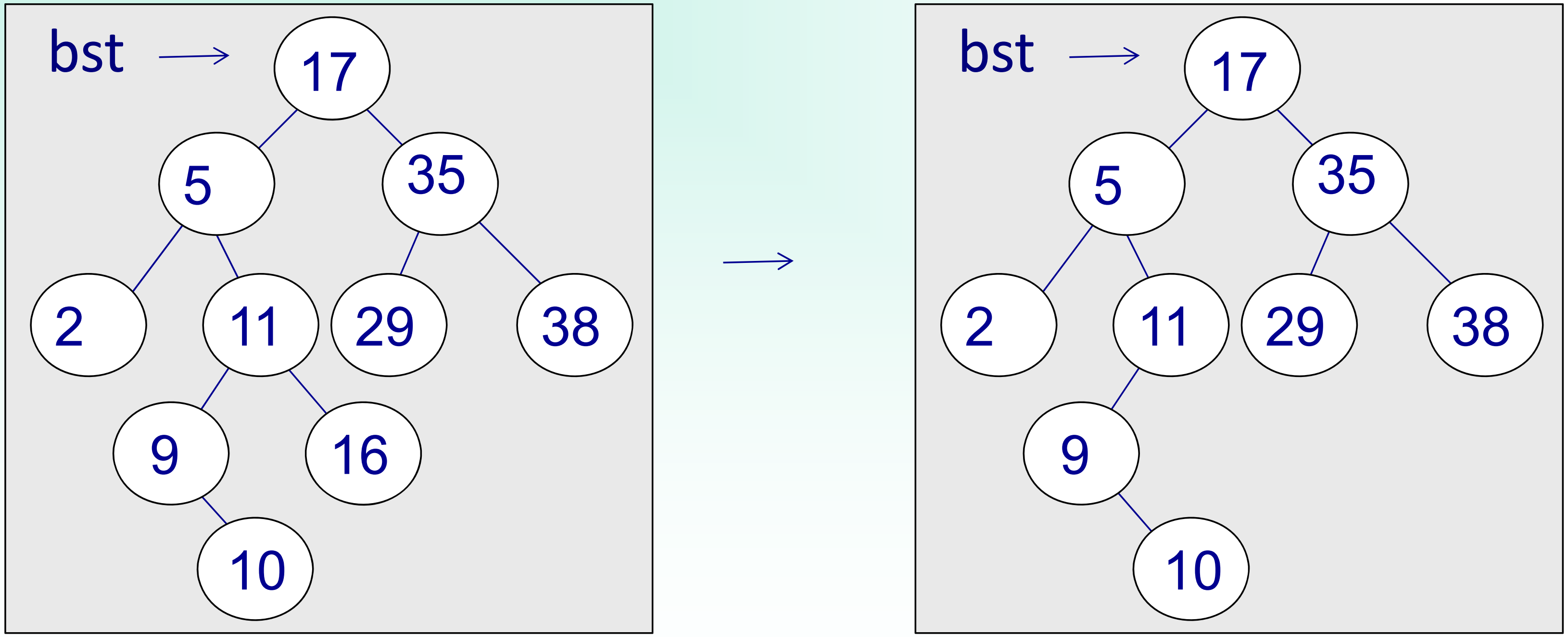
CASE 1: remove node from tree, remove parent pointer, return resulting tree

```
…
def main():                              case1
    bst = create_from_list([ …  ])
    bst = bst.delete(16)
main()
```

BST deleting – no children

CASE 1: deleting a node with no children

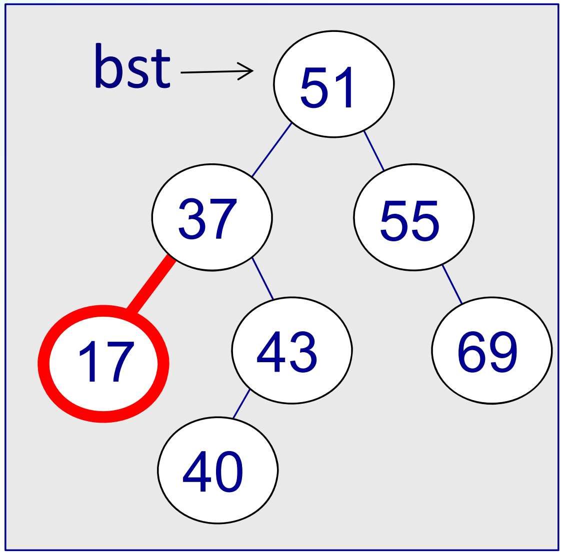
bst = bst.**delete(**16**)**



BST deleting – no children

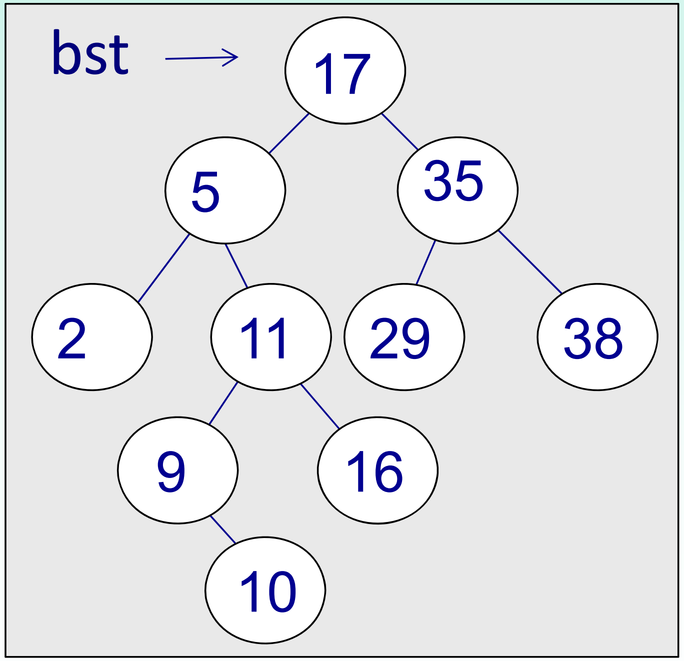
CASE 1: remove node from tree, remove parent pointer, return resulting tree

```
def delete(self, value):
    node = self.locate(value)
    if node==None:
        return self        # value not in tree, do nothing, return tree
    elif (node.left==None and node.right==None):
                    # CASE 1: node is leaf
        if (node.parent == None):
            return None # node is root
        elif (node.parent.left==node):
            node.parent.left=None
        else:
            node.parent.right=None
        node.parent = None
        return self
```



BST deleting – one child

CASE 2: deleting a node with one child only.



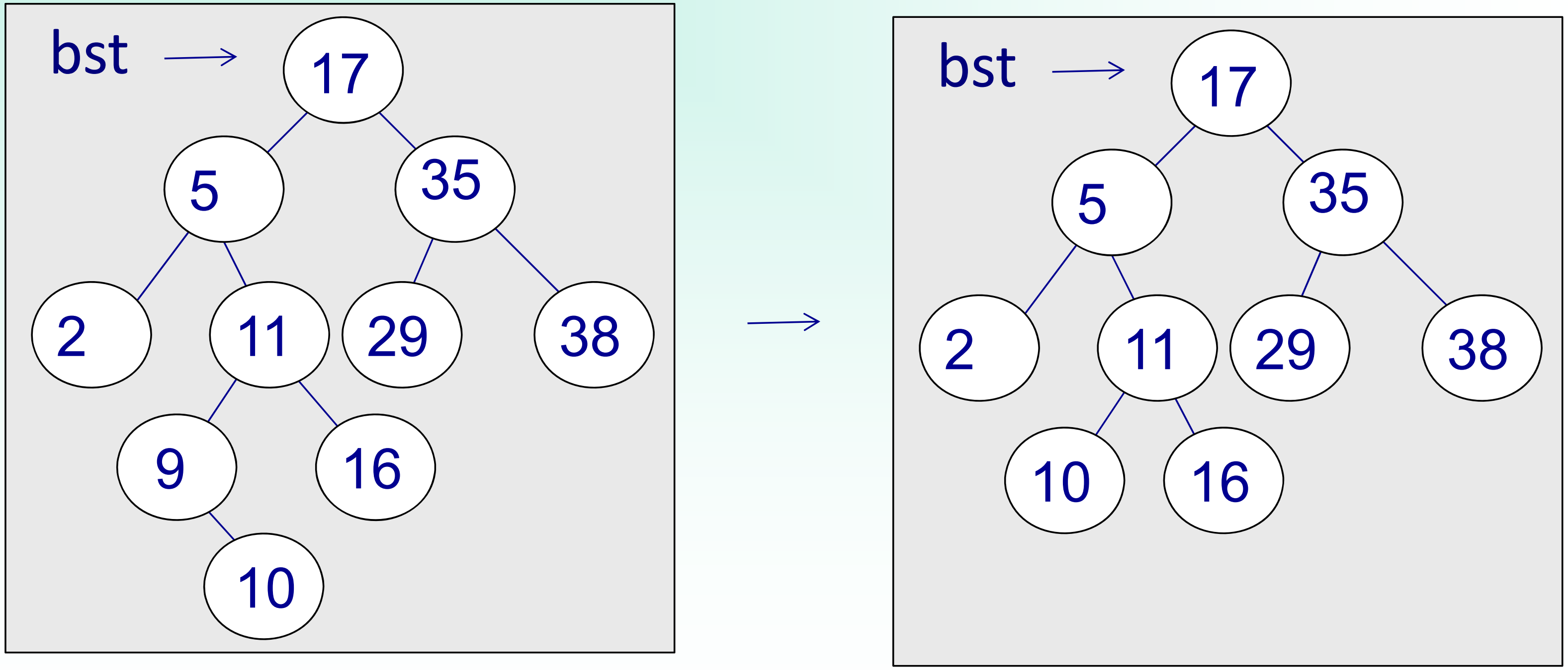
CASE 2: delete the node and shift its child up to take its place by changing the parent link.

```
...
def main():
    bst = create_from_list([ ...  ])
    bst = bst.delete(9)        case2
main()
```

BST deleting – one child

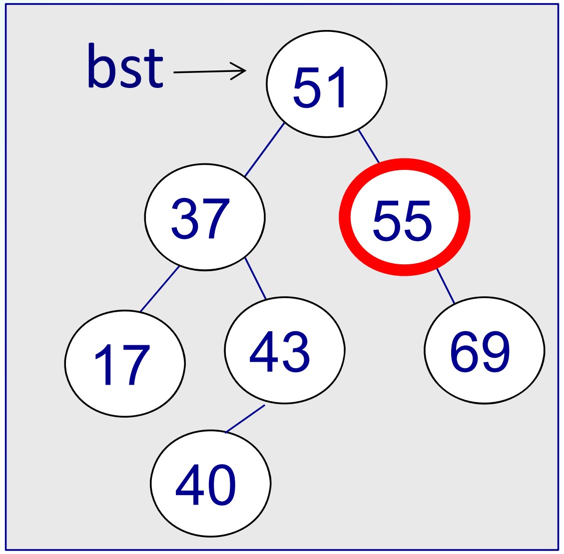
CASE 2: delete the node and shift its child up to take its place by changing the parent link.

bst = bst.**delete**(9)



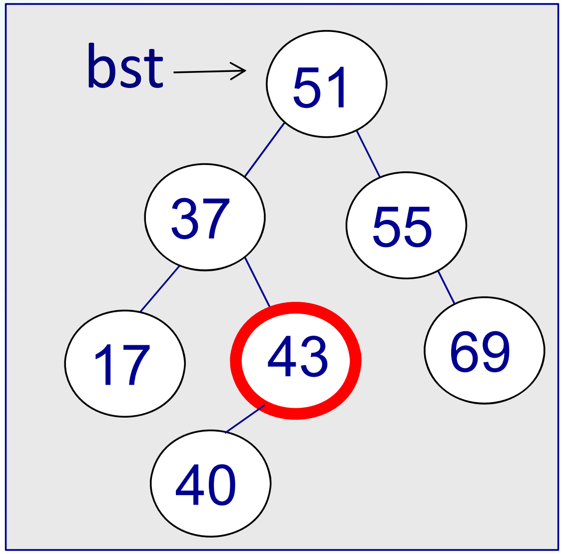
BST deleting – one child

```python
elif (node.left==None): # CASE 2a: node has only right child
    if (node.parent== None):
        node.right.parent = None
        return node.right
    elif (node.parent.left==node):
        node.parent.left=node.right
        node.right.parent=node.parent
    else:
        node.parent.right=node.right
        node.right.parent=node.parent
    node.parent = None
    node.right = None
    return self
```



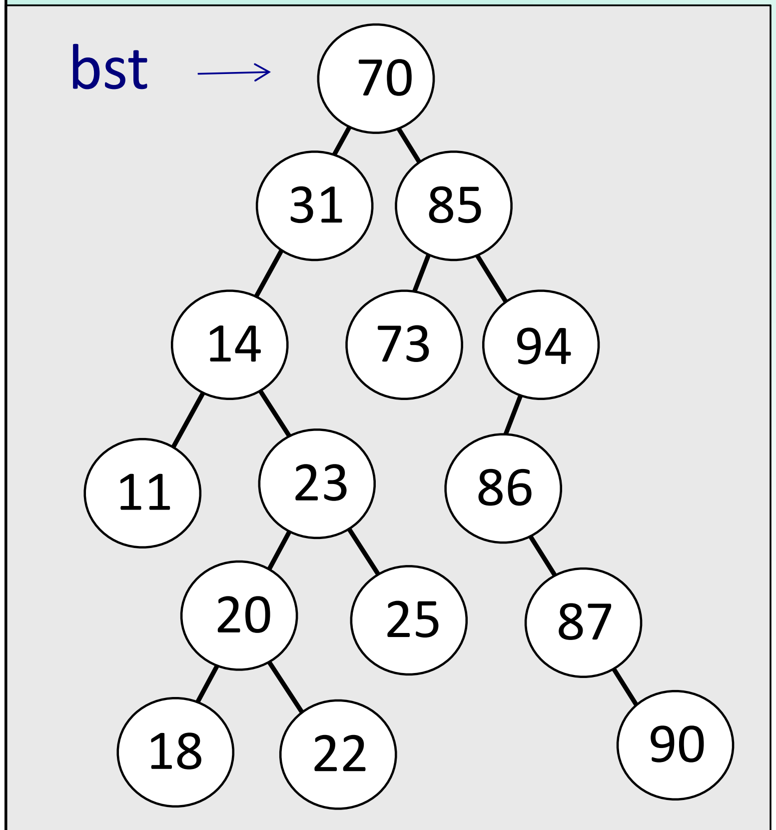
BST deleting – one child

```
elif (node.right==None): #  CASE 2b: node has only left child
    if (node.parent==None):
        node.left.parent = None
        return node.left
    elif (node.parent.left==node):
        node.parent.left=node.left
        node.left.parent=node.parent
    else:
        node.parent.right=node.left
        node.left.parent=node.parent
    node.parent = None
    node.left = None
    return self
```



What is the inorder successor?

This is the next biggest value when an inorder traversal is done on the tree.



How do we find the inorder successor of a node?

The inorder successor of 85?

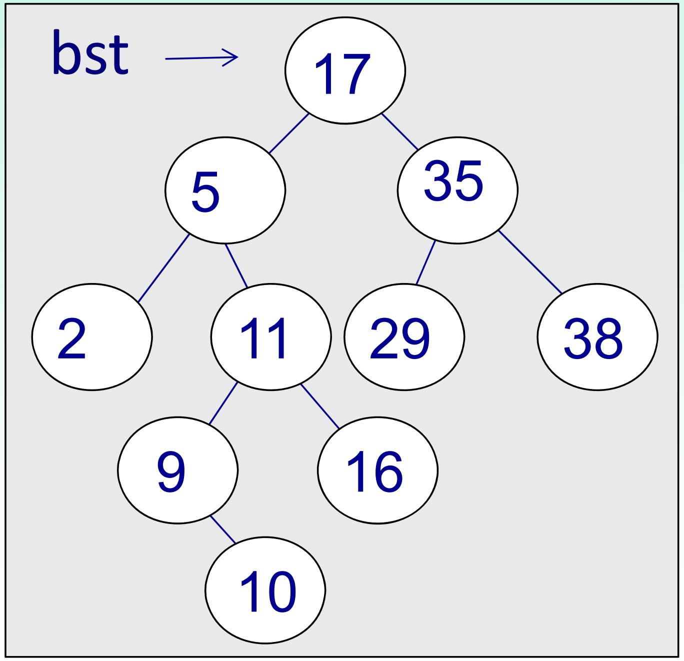
The inorder successor of 23?

The inorder successor of 14?

The inorder successor of 70?

BST deleting – two children

CASE 3: deleting a node with two children.



CASE 3: Replace the value in the node with its inorder successor.

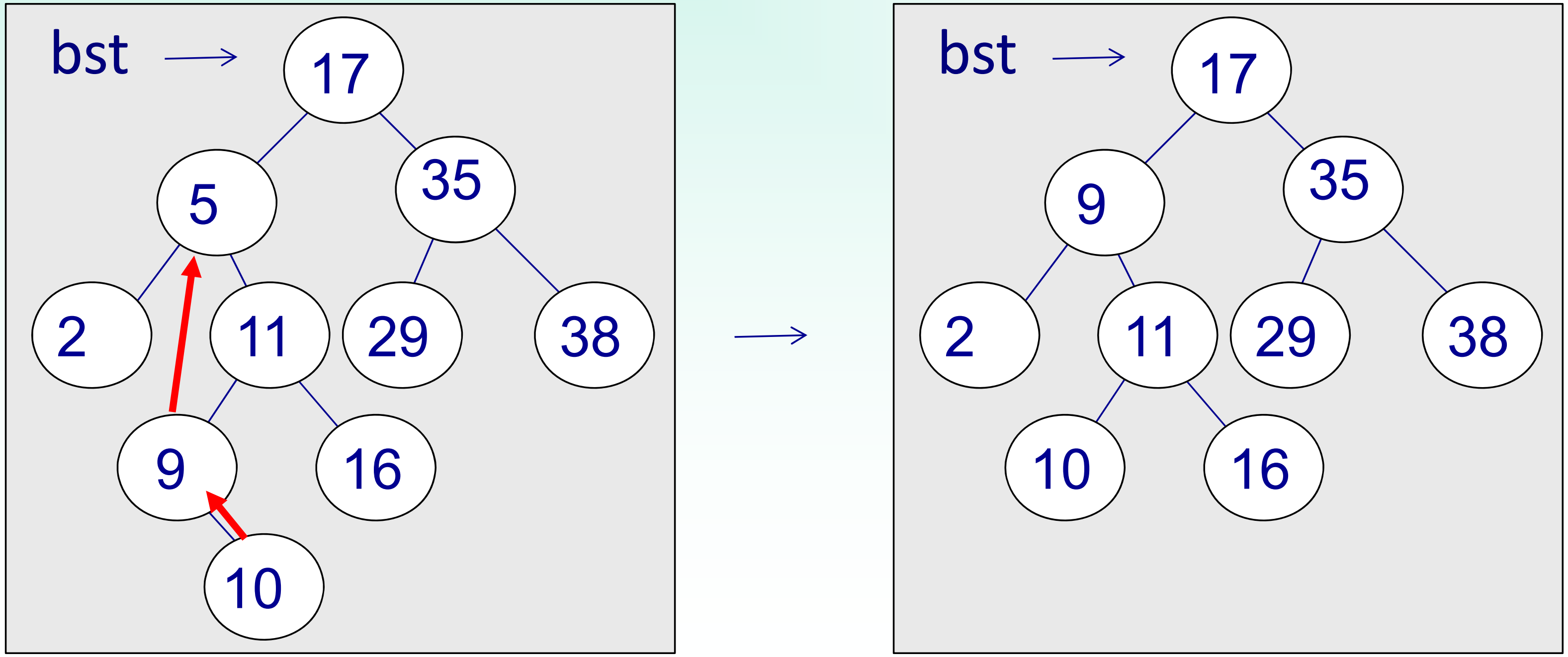
We will also have to delete the inorder successor node. But that node has at most one child! (think why)

```
...
def main():
    bst = create_from_list([ ...  ])
    bst = bst.delete(5)     case3
main()
```

BST deleting – two children

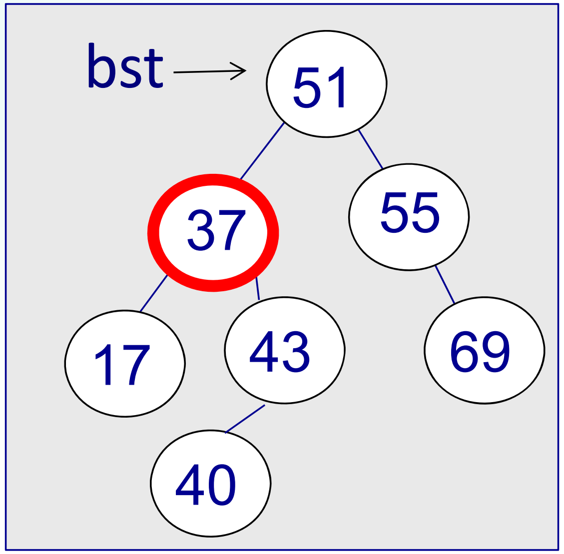
CASE 3: Replace the value in the node with its inorder successor. We will also have to delete the inorder successor node (max 1 child – think about why ☺).

bst.**delete(**5**)**



BST deleting – two children

```
else:                   # CASE 3: Node has left and right child
    succ = node.right             # Find inorder successor
    while succ.left:
        succ = succ.left
    node.value = succ.value
    succ = succ.delete(succ.value)
    return self
```



Performance of BST

NOTE: A tree is balanced if for every node its left and right subtree vary in height by at most one

If BST is balanced than height is O(log n) and hence insert, locate, delete are all O(log n)!

Can show that average running times for insert, locate, delete are all O(log n)!

Worst case is O(n) ☹

BUT ☺ : Can create tree which is always balanced and hence always O(log n) [AVL tree - not part of this lecture]
Another famous tree is the Splay tree, which has an amortised cost of O(log n)

Advantages of BST

Compared to unsorted list:

- Insert is slightly slower (O(log n) vs. O(1)), but delete and find are much faster (O(log n) vs. O(n))

Compared to sorted list:

- Both have O(log n) find operation, but BST can also insert and delete in O(log n)

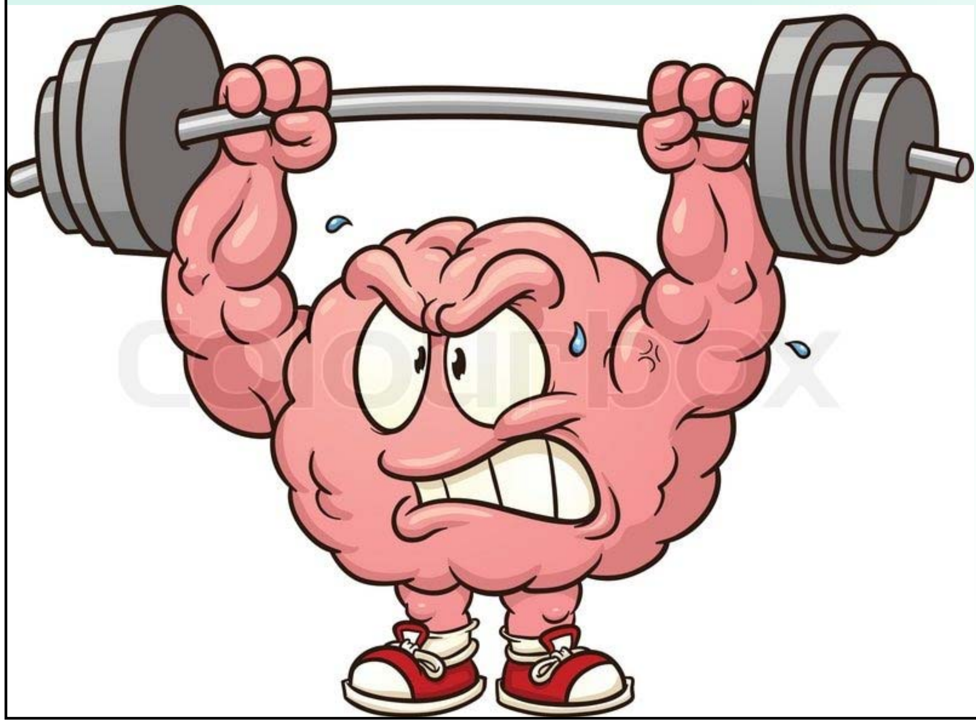
Compared to heap:

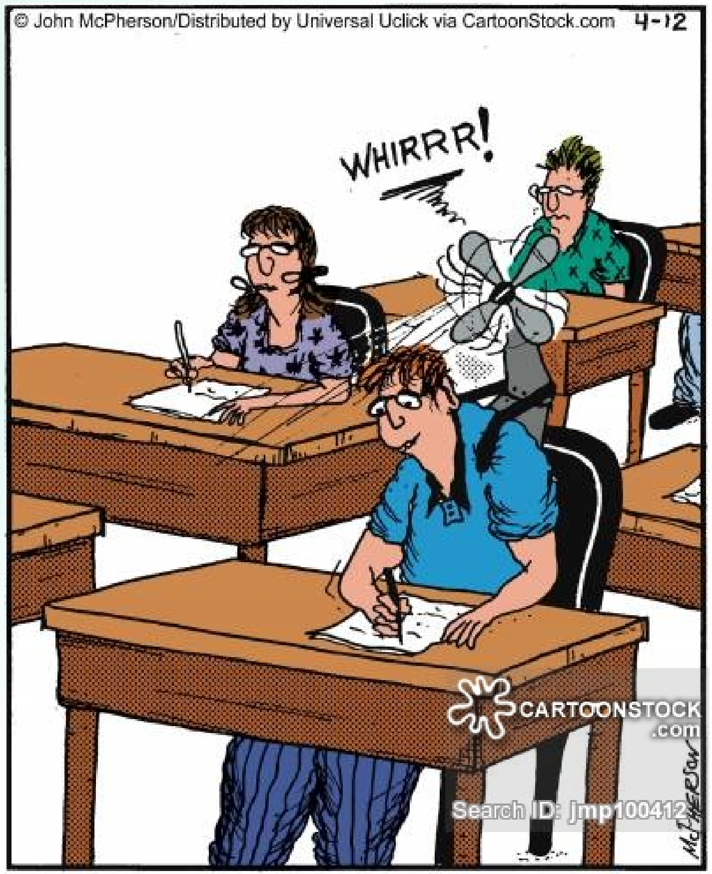
- Can access all elements without removing them
- Can list elements in sorted order in O(n)

  NOTE: Can use BST for sorting (Tree Sort):

  Insert n elements and output in inorder

Exercises





Garrett knew it was important to keep his brain from overheating during big tests.

Binary search trees – past exam Q1

Draw the binary search tree structure after inserting the following integer search key values into an empty binary search tree in the order given:

40, 20, 10, 60, 70, 45, 50, 15, 55

Draw the binary search tree structures (draw 3 trees) after deleting the following search key values in the order given:

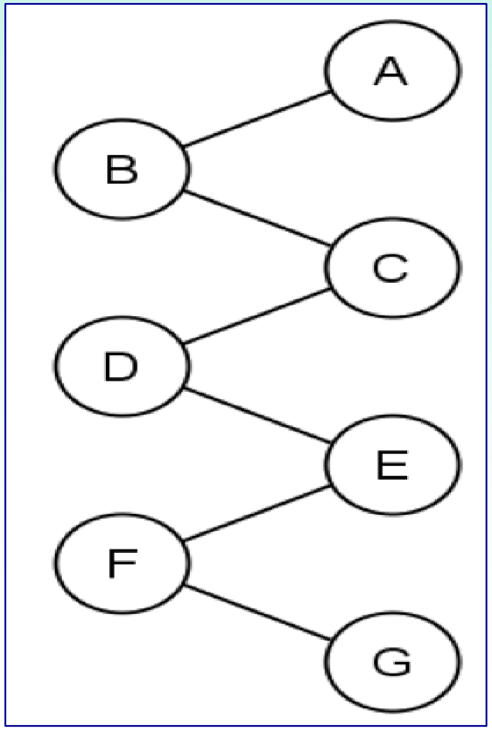
i) **20**

ii) **40**

iii) **45**

Binary search trees – past exam Q2

The following diagram shows a binary tree with the root node containing the value, A. Write the pre-order, in-order and post-order traversals of the following binary tree.



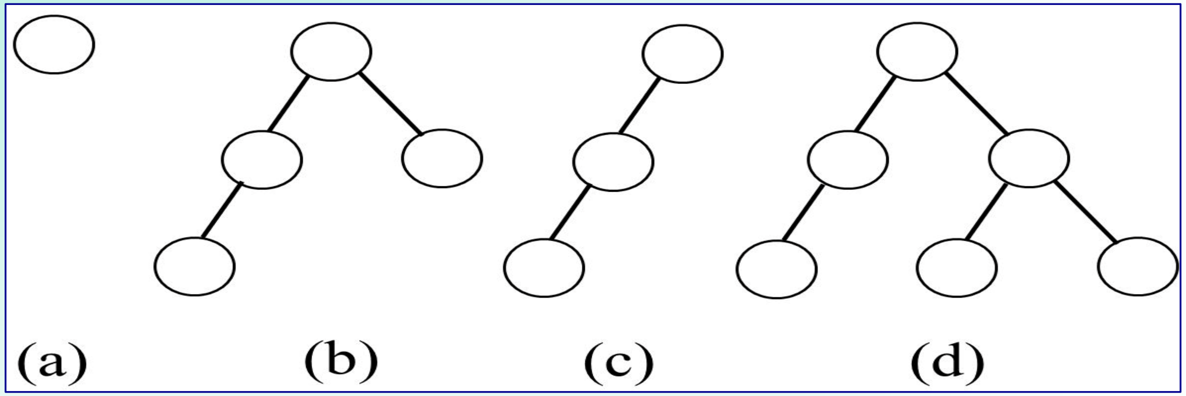
pre-order:

in-order:

post-order:

Binary search trees – past exam Q3

Consider the following binary trees. For each binary tree, indicate if it is complete, full and/or balanced.



(a)Complete: yes/no
Full: yes/no
Balanced: yes/no

(b)Complete: yes/no
Full: yes/no
Balanced: yes/no

(c)Complete: yes/no
Full: yes/no
Balanced: yes/no

(d)Complete: yes/no
Full: yes/no
Balanced: yes/no