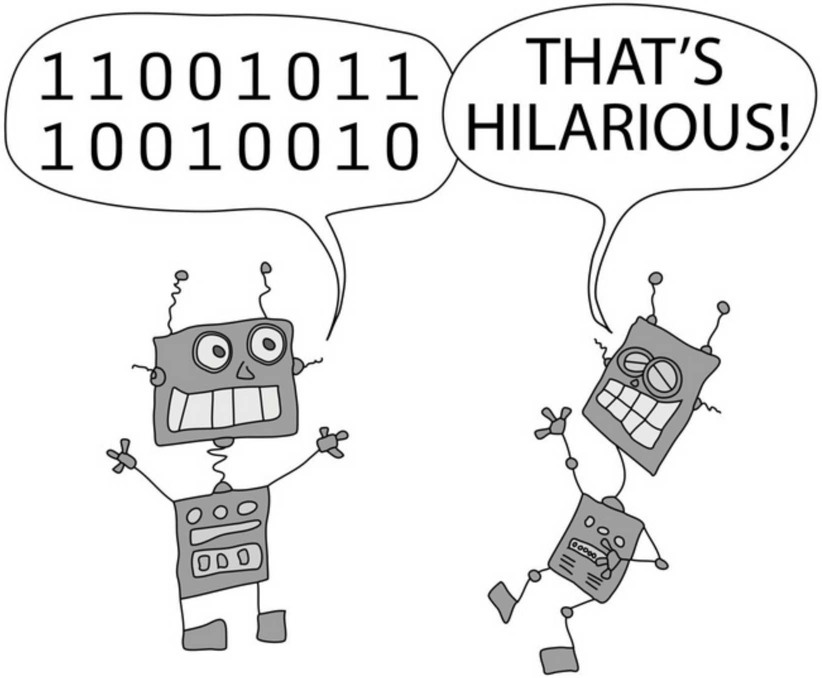
Binary Search Trees



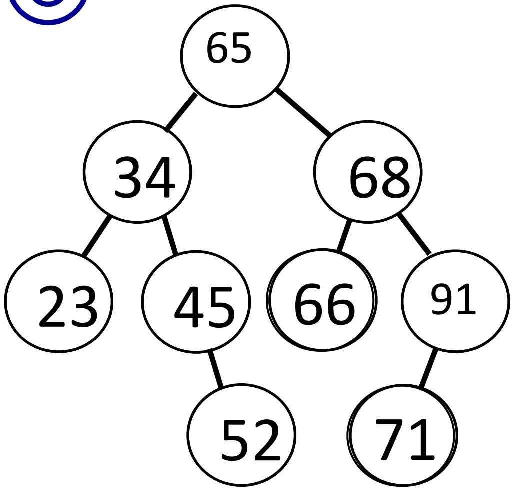
Trees can be very efficient

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 ☺

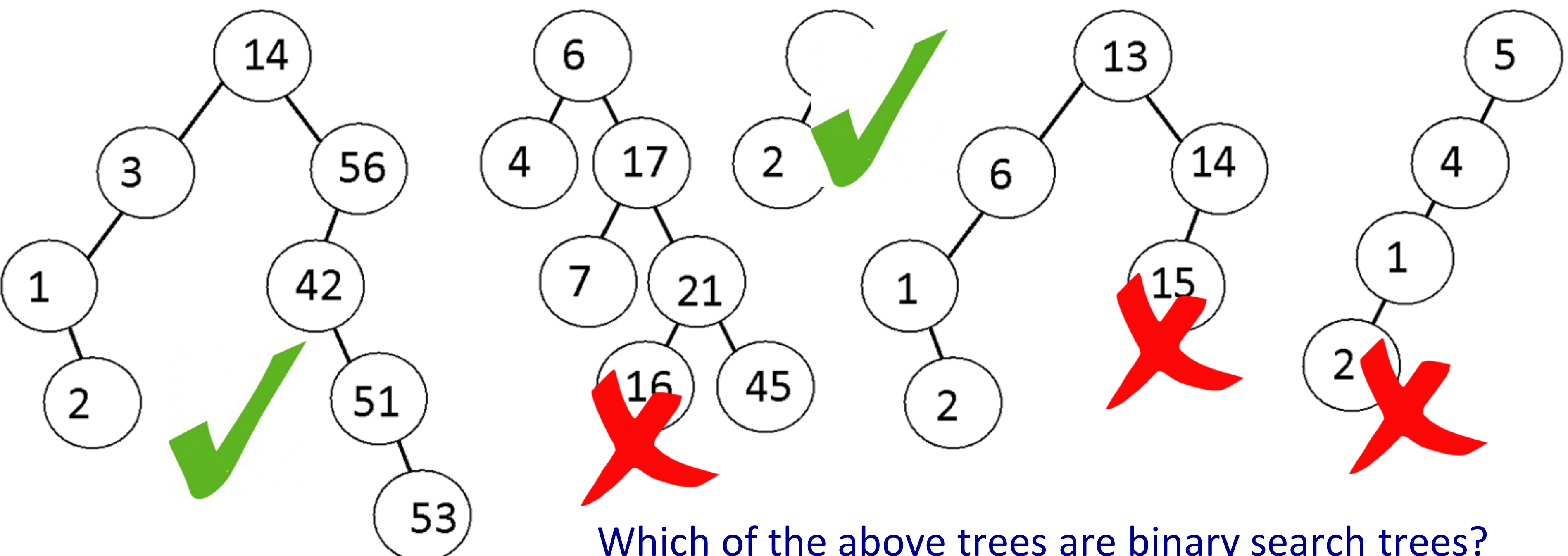
23	34	45	52	65	66	68	71	91



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 – Searching

- Implement the search function for a binary search tree

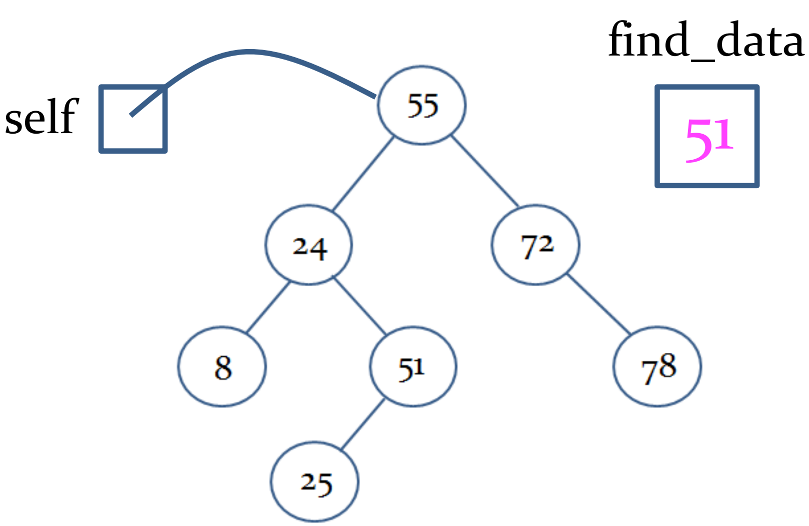
Compare the find_data with the current root node data

1. **If the two are equal, we have found the data.**
   Return True

2. **If the find_data is less than the current node data, we know the find_data either isn't in the tree or is in the left sub-tree.**
   Search the left sub-tree.

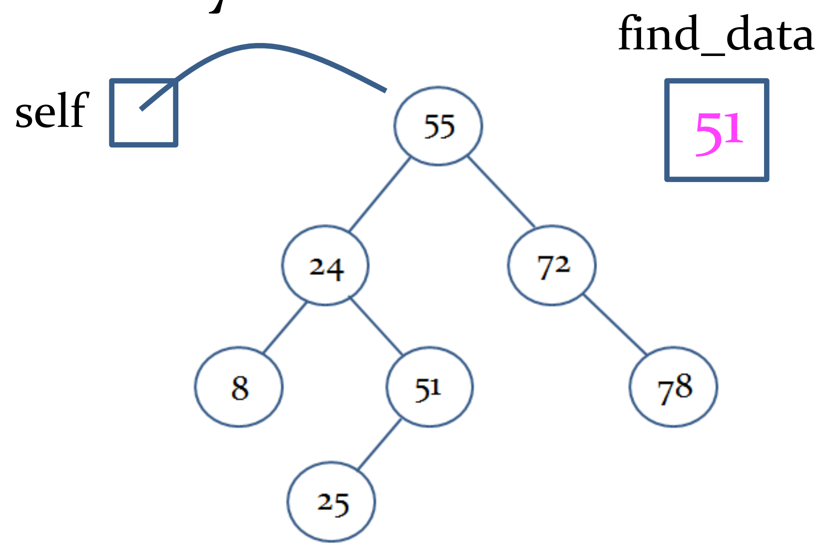
3. **If the find_data is greater than the current node data.**
   Search the right sub-tree

4. **If we reach a node with no children and we haven't found a match for the data, it isn't in the tree.**
   Return False



Binary search trees – Searching

- Implement the search function for a binary search tree



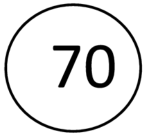

```
class BinarySearchTree:

  def __init__(self, data):
     self.__data = data
     self.__left = None
     self.__right = None

  def search(self, find_data):
     if self.get_data() == find_data:
         return True
     elif find_data < self.get_data() and self.get_left()!= None:
         return self.get_left().search(find_data)
     elif find_data > self.get_data() and self.get_right()!= None:
         return self.get_right().search(find_data)
     else:
         return False
```

Binary search trees - insert

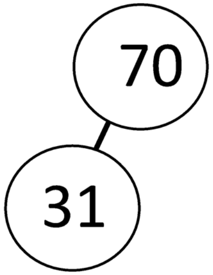
To demonstrate, we add a list of elements in the order they occur and ALWAYS MAINTAIN THE BINARY SEARCH TREE PROPERTY. For example, the following list:

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

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

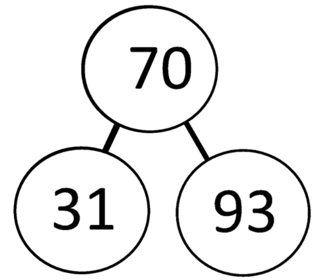


70, **31**, 93, 94, 14, 23, 73

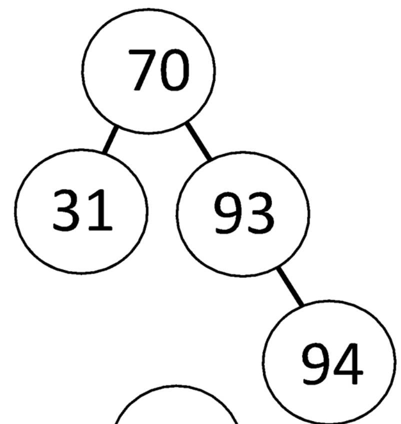


Binary search trees - insert

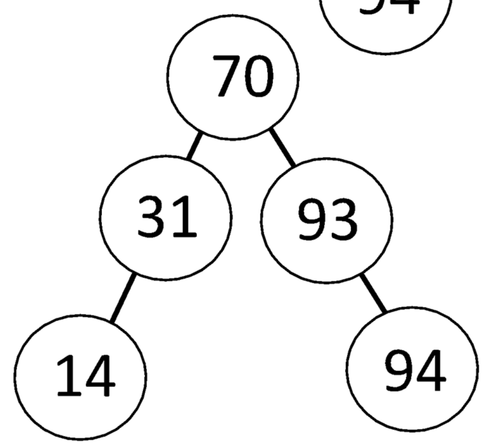
70, 31, **93**, 94, 14, 23, 73



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

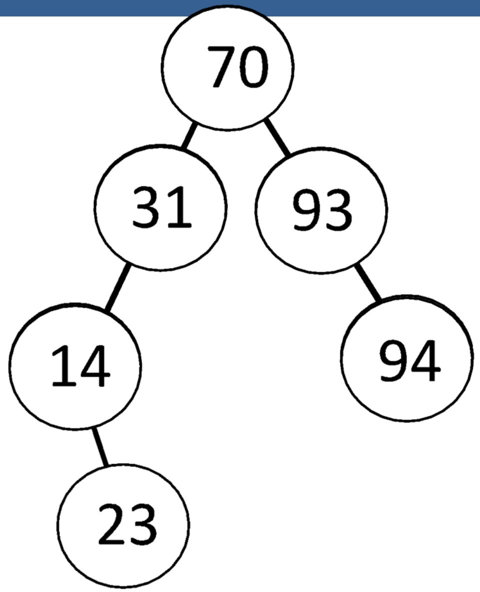


70, 31, 93, 94, **14**, 23, 73

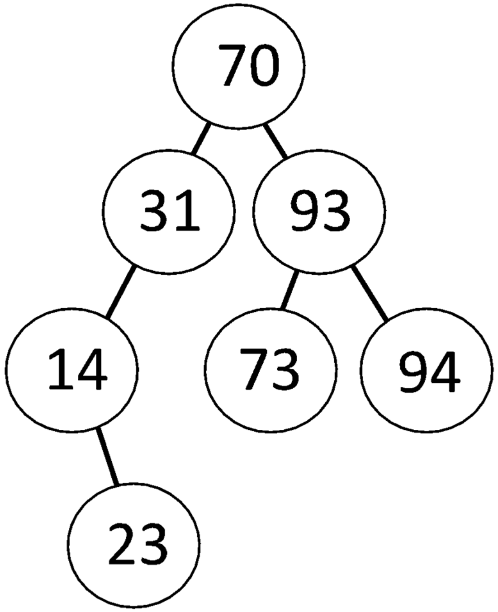


Binary search trees - insert

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



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



Binary search trees

- Implement the insert function for a binary search tree

```
class BinarySearchTree:

    def __init__(self, data):
        self.__data = data
        self.__left = None
        self.__right = None

    def insert(self, new_data):
        if new_data == self.get_data():
            return
        elif new_data < self.get_data():
            if self.get_left() == None:
                self.set_left(BinarySearchTree(new_data))
            else:
                self.get_left().insert(new_data)
        else:
            if self.get_right() == None:
                self.set_right(BinarySearchTree(new_data))
            else:
                self.get_right().insert(new_data)
```

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 then the height is O(log n) and hence insert, locate, delete are all O(log n)!

Can show that the 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 $O(\log n)$

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

Insert n elements and output in inorder