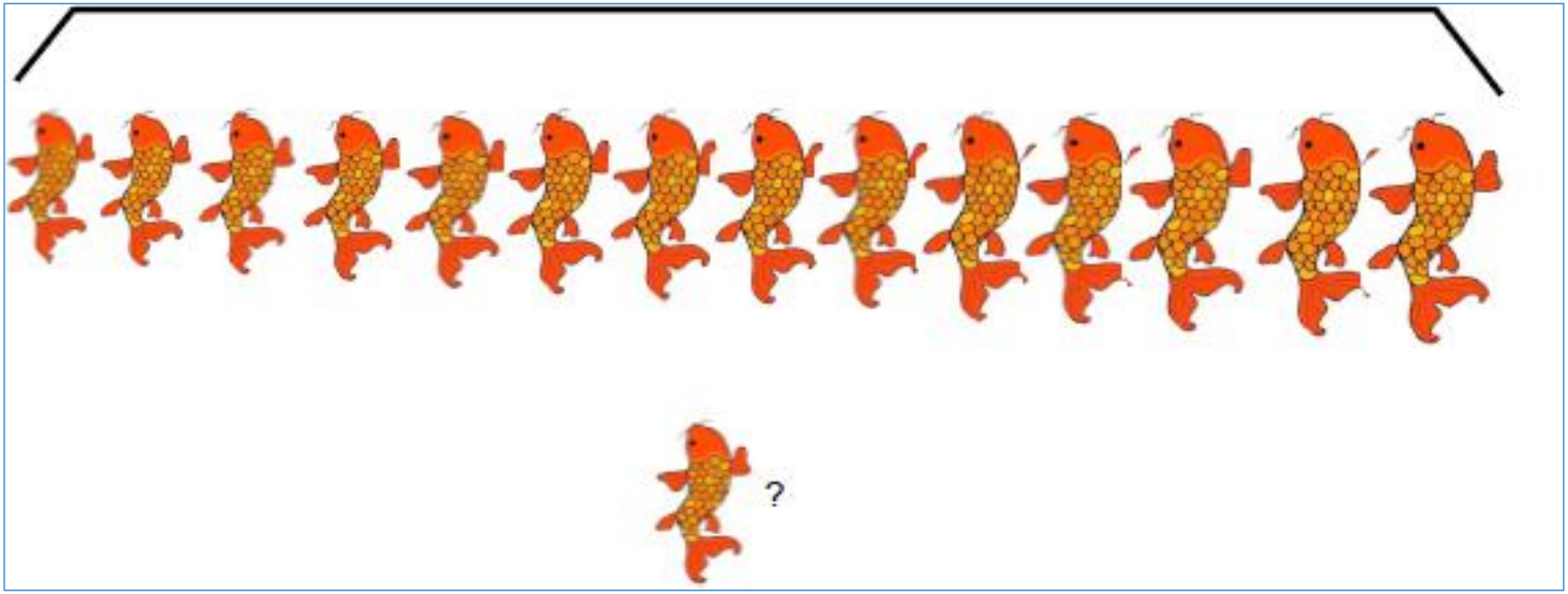
Search

- Binary Search
- Binary Tree
- Binary Search Tree

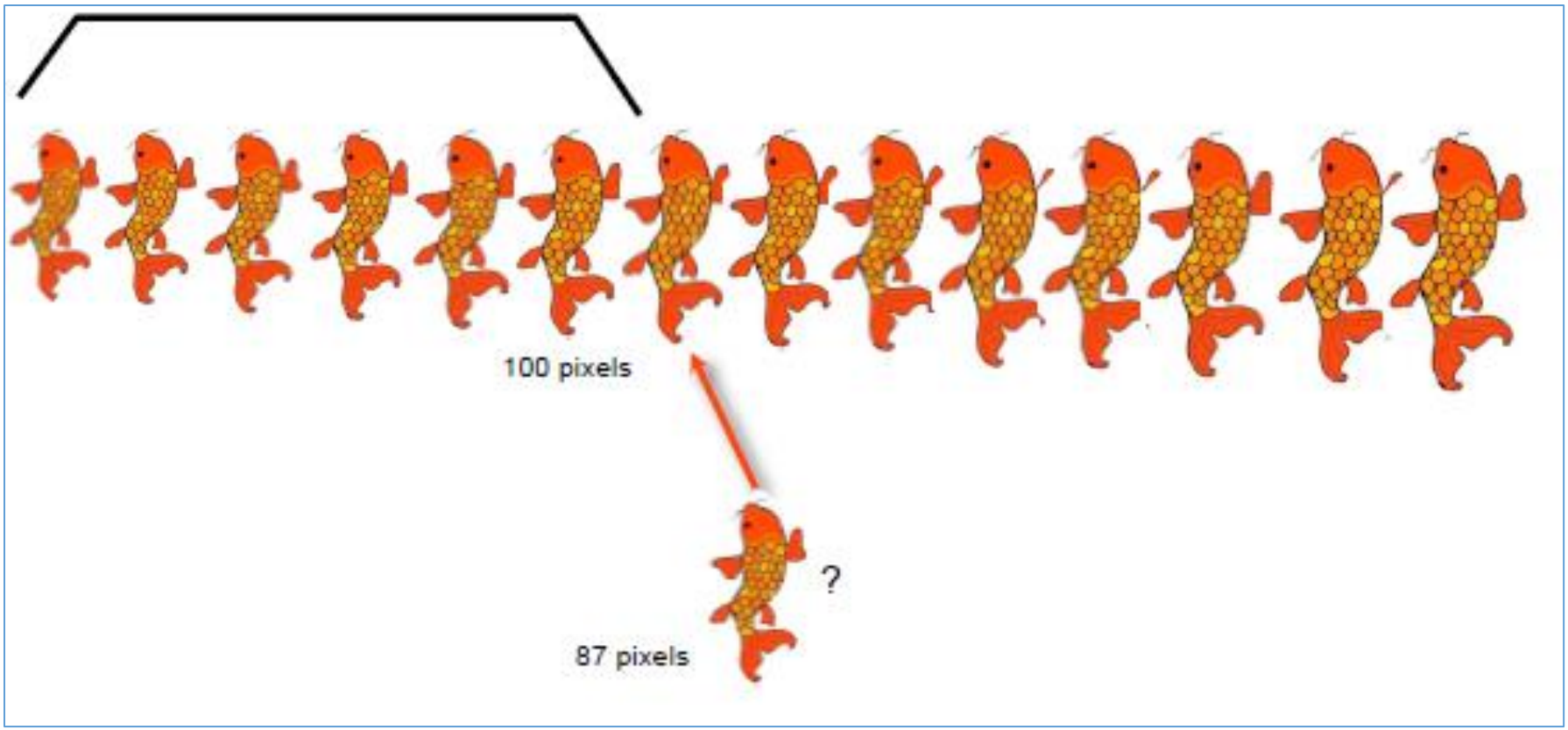
Number guessing

- I'm thinking of a number between 1 and 16
- You get to ask me, yes or no, is it greater than some number you choose
- How many questions do you need to ask?
- Which questions will you ask to get the answer quickest?

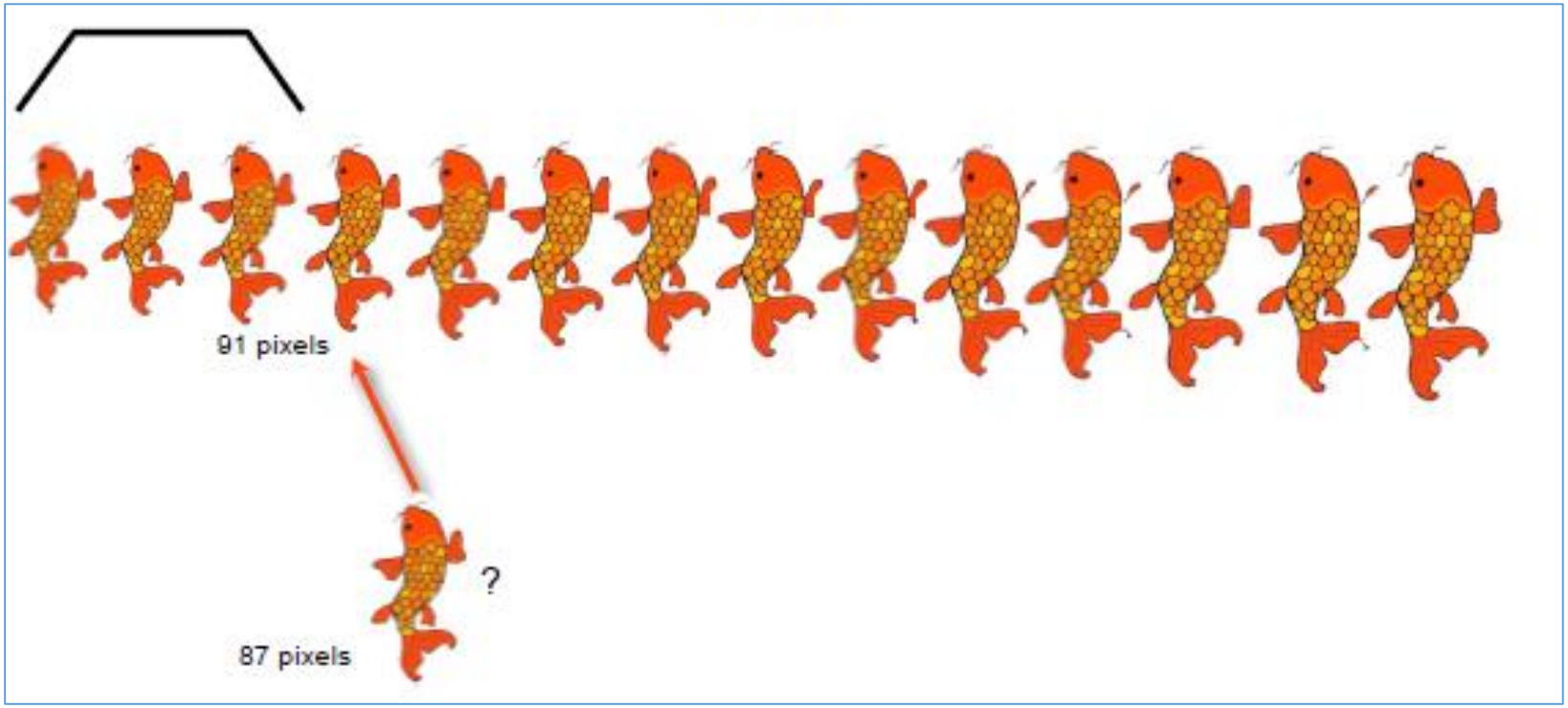
Binary Search in an Ordered List [1/5]



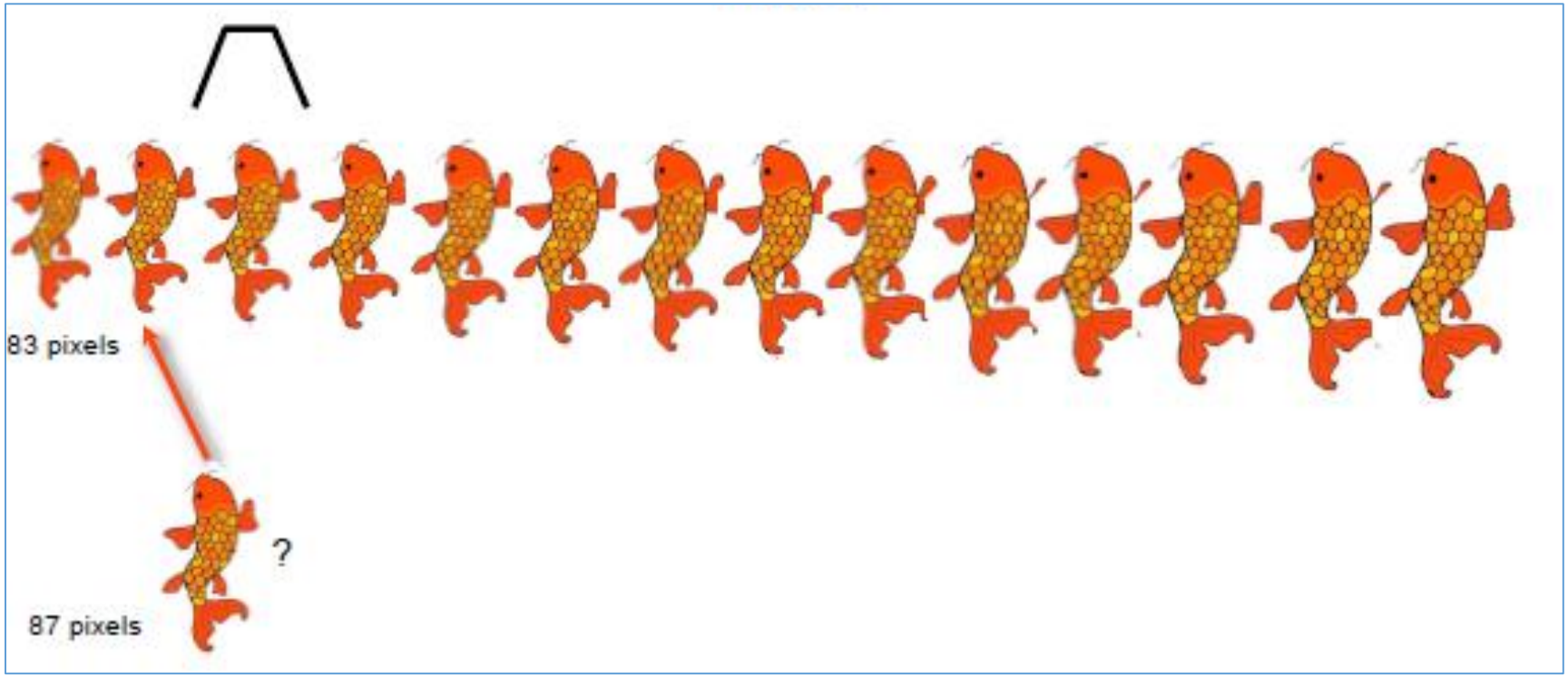
Binary Search in an Ordered List [2/5]



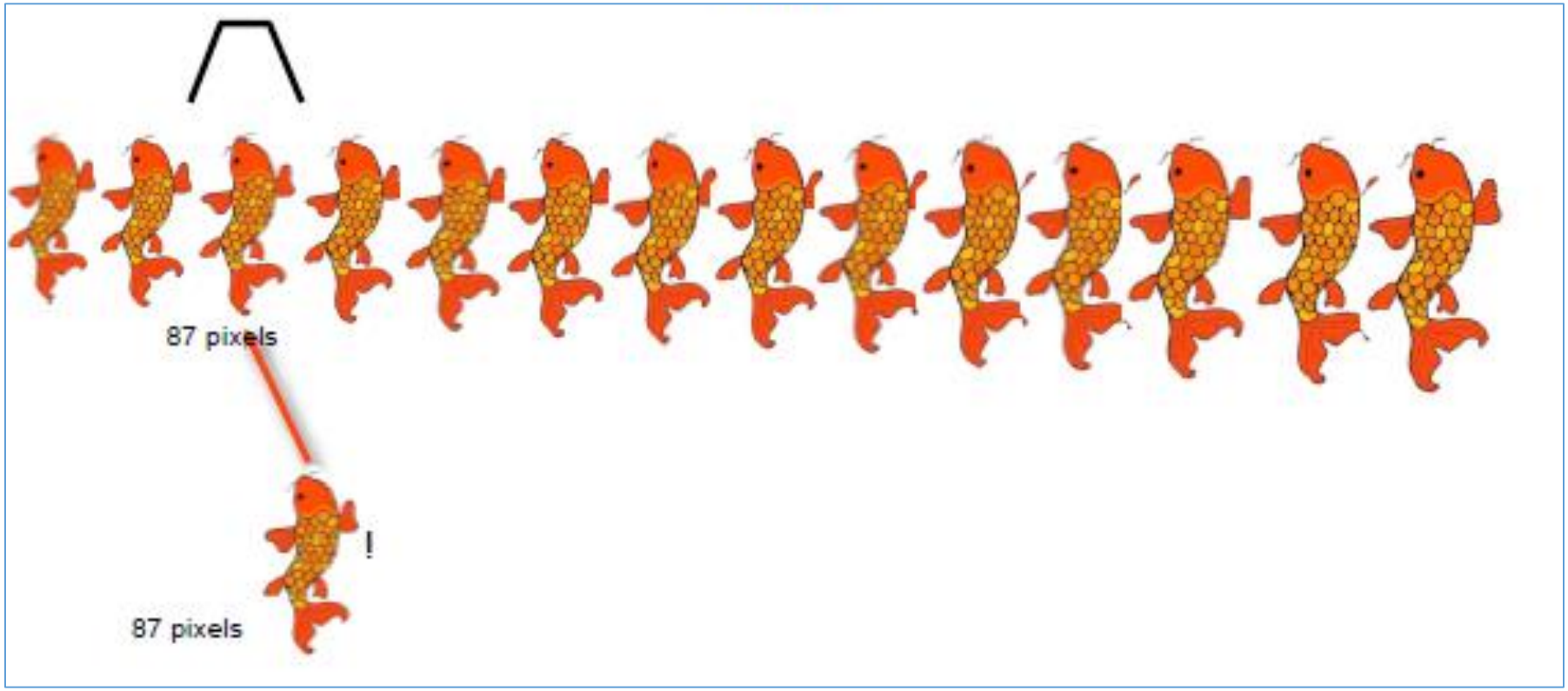
Binary Search in an Ordered List [3/5]



Binary Search in an Ordered List [4/5]



Binary Search in an Ordered List [5/5]



Specification: the Search Problem

- Input: A list of n unique elements and a key to search for
  - **The elements are sorted in increasing order.**
- Result: The index of an element matching the key, or `None` if the key is not found.

Recursive Algorithm

BinarySearch(list, key):

1. Return None if the list is empty.
2. Compare the key to the middle element of the list
3. Return the index of the middle element if they match
4. If the key is less than the middle element then
   return BinarySearch(first half of list,key)
   Otherwise, return BinarySearch(second half of list,key).

Example 1: Search for 73

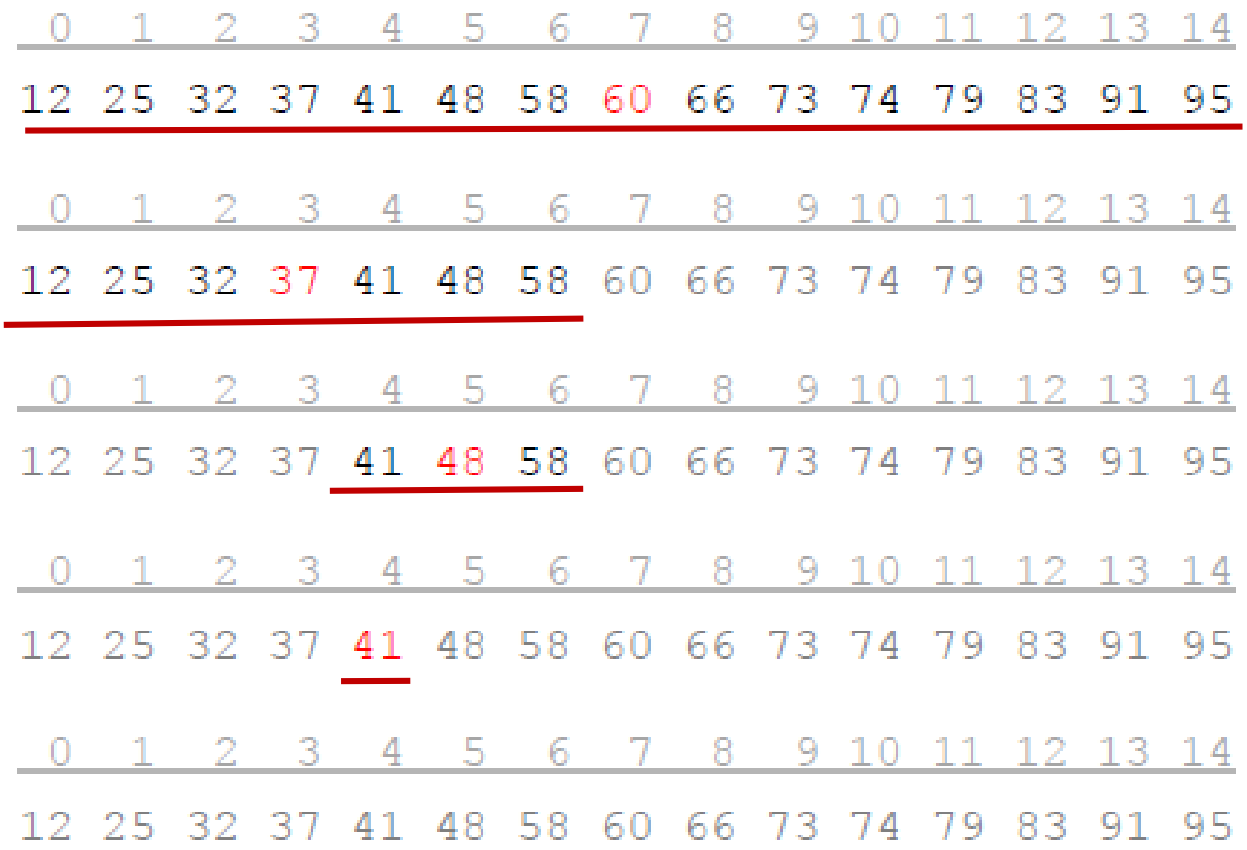
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Found: return 9

Example 2: Search for 42



Not found: return None

Controlling the Range of the Search [1/2]

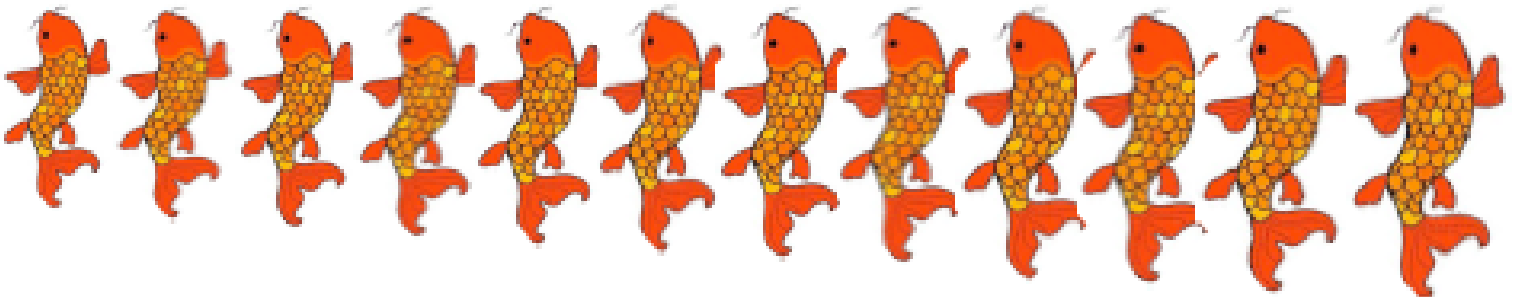
- Maintain three numbers: *lower*, *upper*, *mid*
- Initially *lower* is -1, *upper* is length of the list

lower = -1

upper = 12

0

11



Controlling the Range of the Search [2/2]

- *mid* is the midpoint of the range:

  mid = (*lower* + *upper*) // 2 (**integer** division)

  Example: *lower* = -1, *upper* = 9

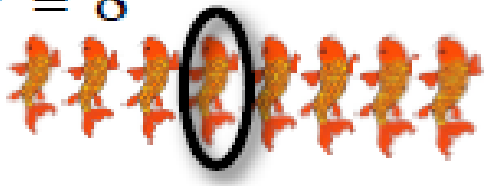
  (range has 9 elements)

  mid = 4

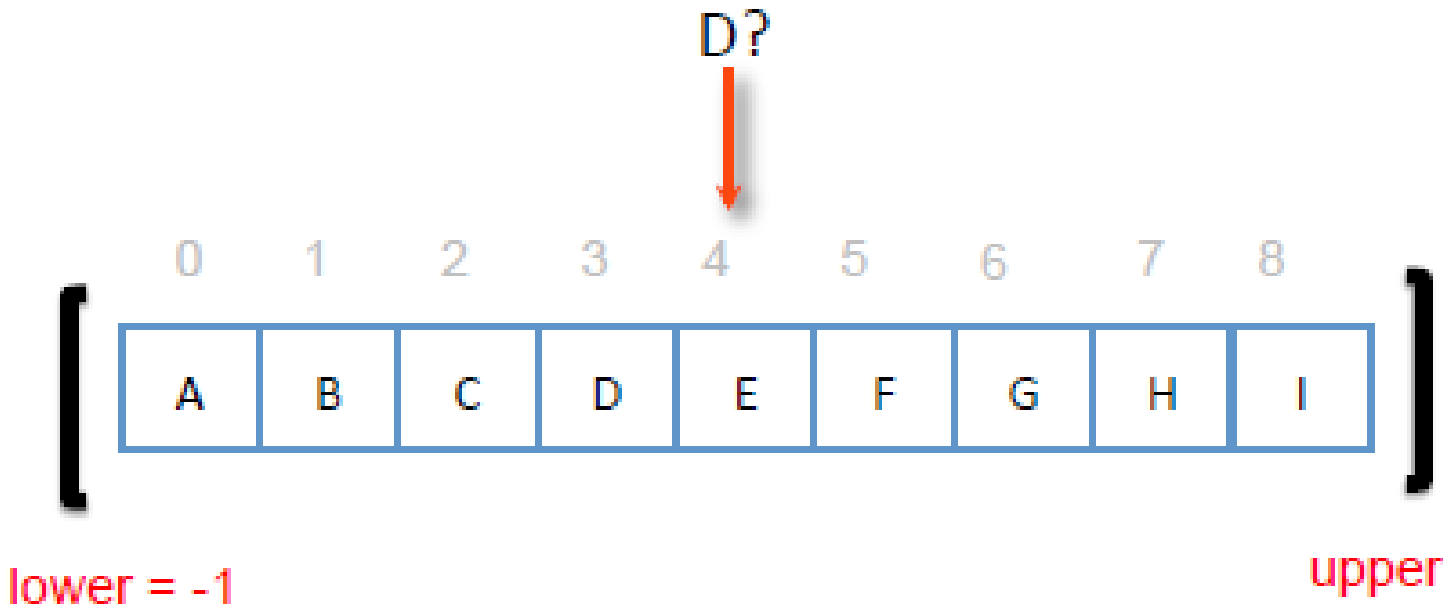
- What happens if the range has an even number of elements?

  Example: *lower* = -1, *upper* = 8

  mid = 3

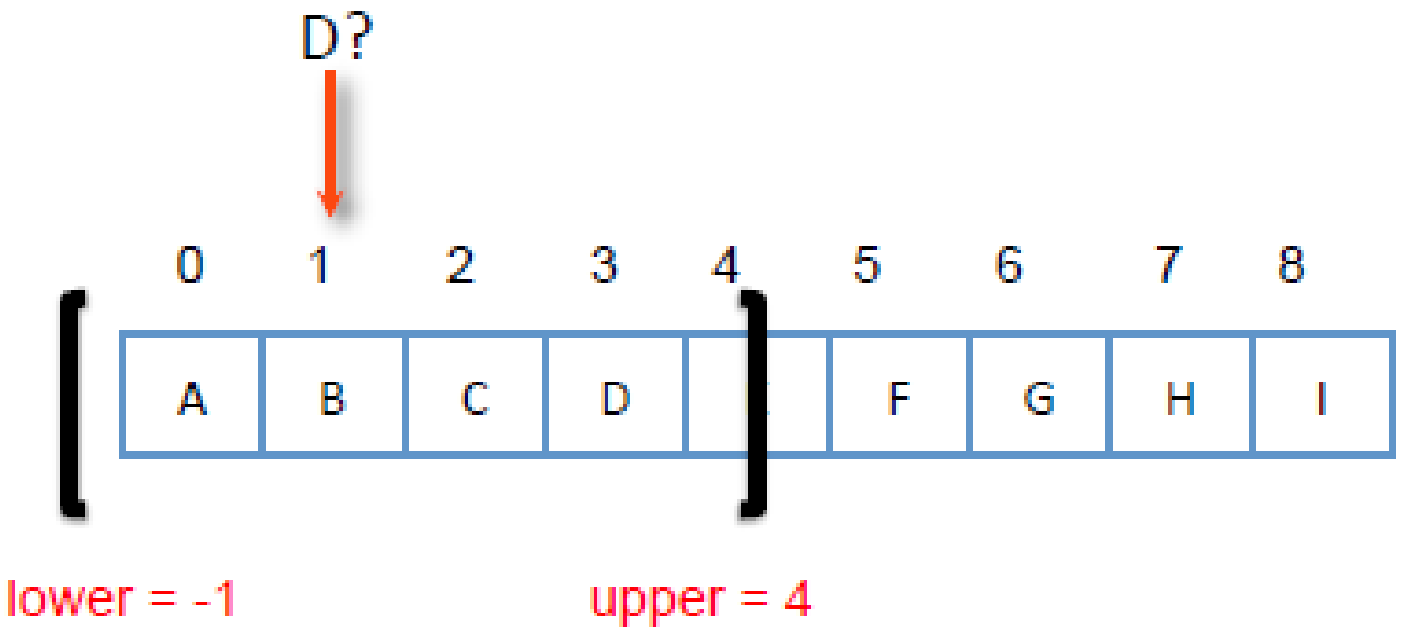


Example



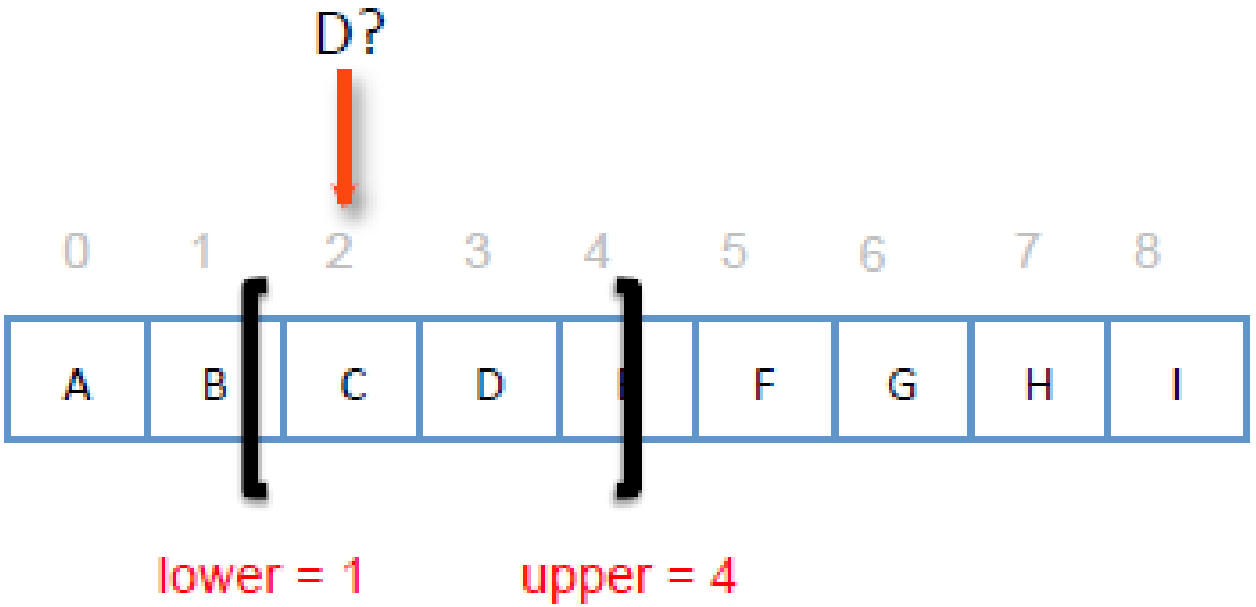
List sorted in ascending order.
Suppose we are searching for D.

Example



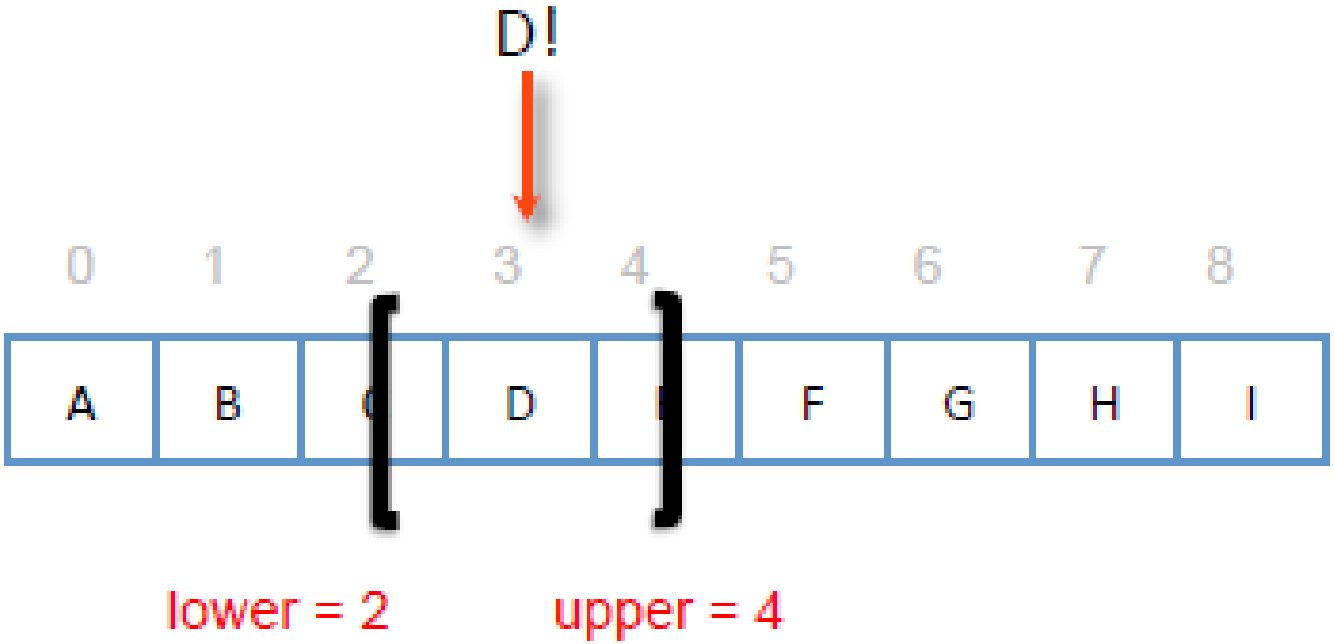
Each time we look at a smaller portion of the list within the window and ignore all the elements outside of the window

Example



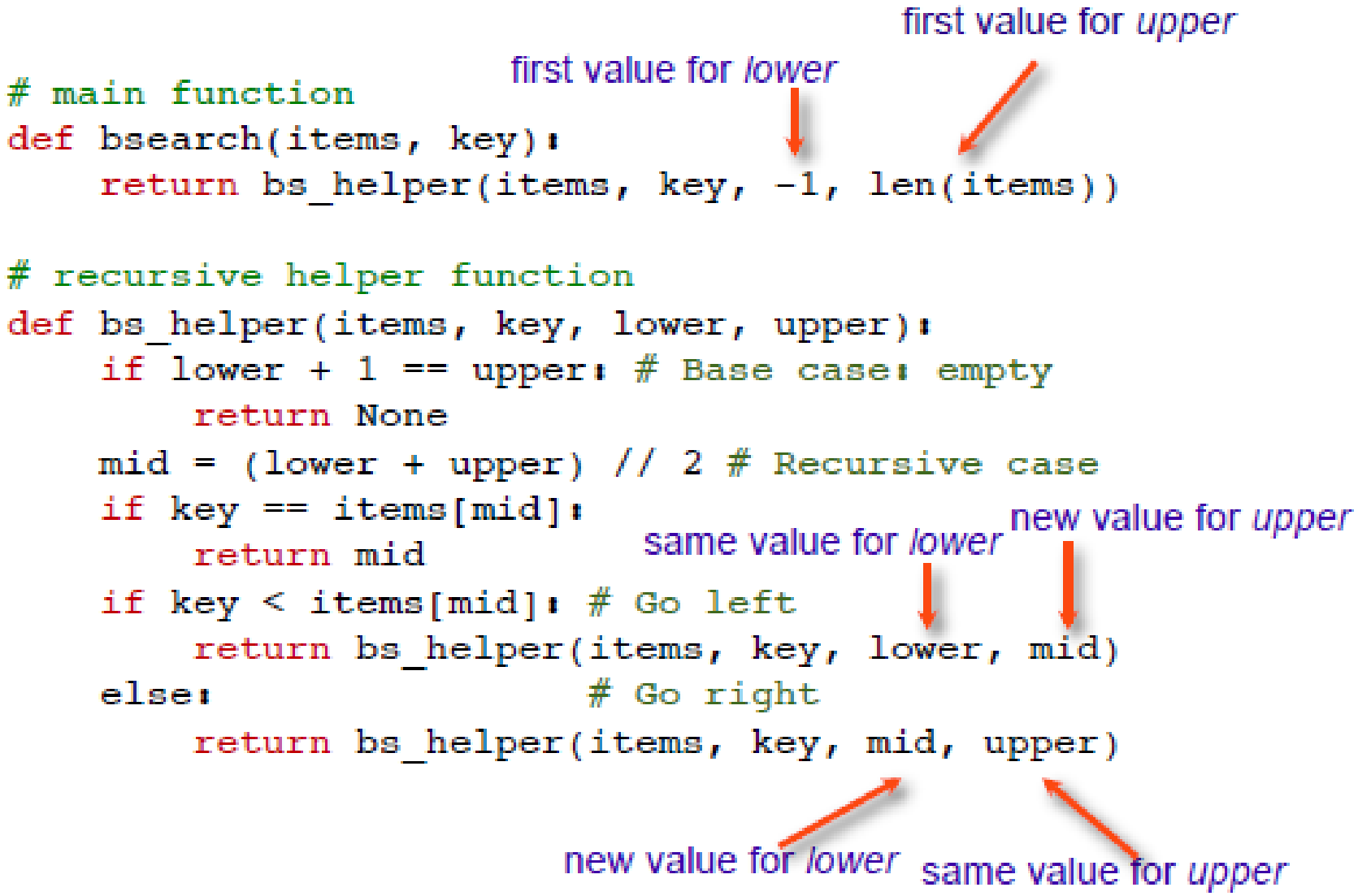
Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

Example



Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

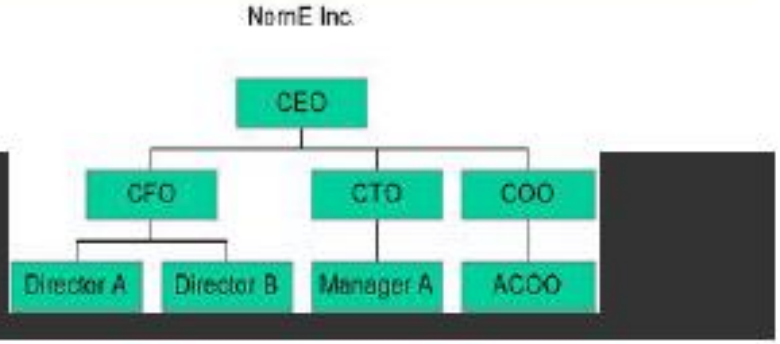
Recursive Binary Search in Python

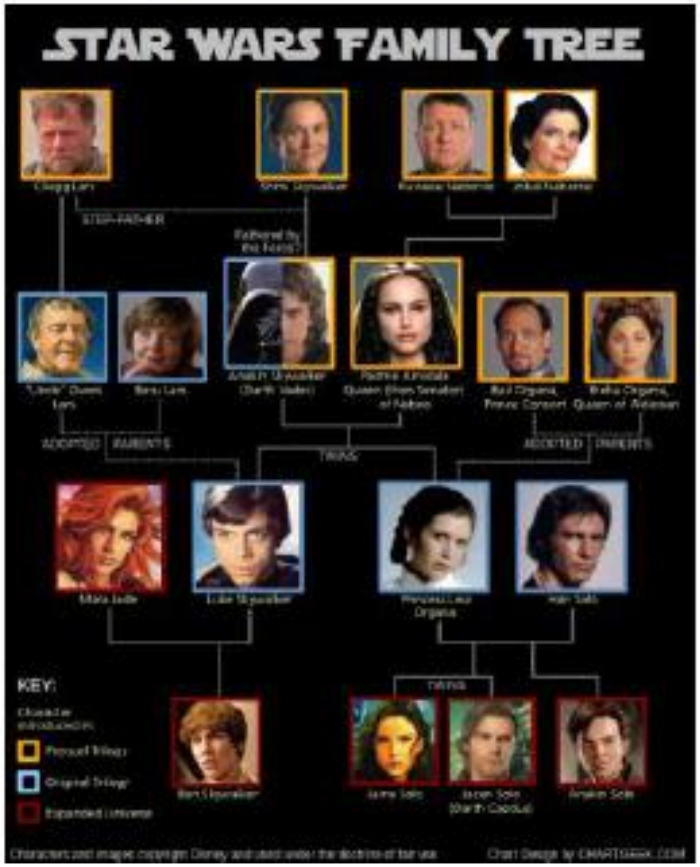


Search

- Binary Search
- Binary Tree
- Binary Search Tree

Hierarchical Data

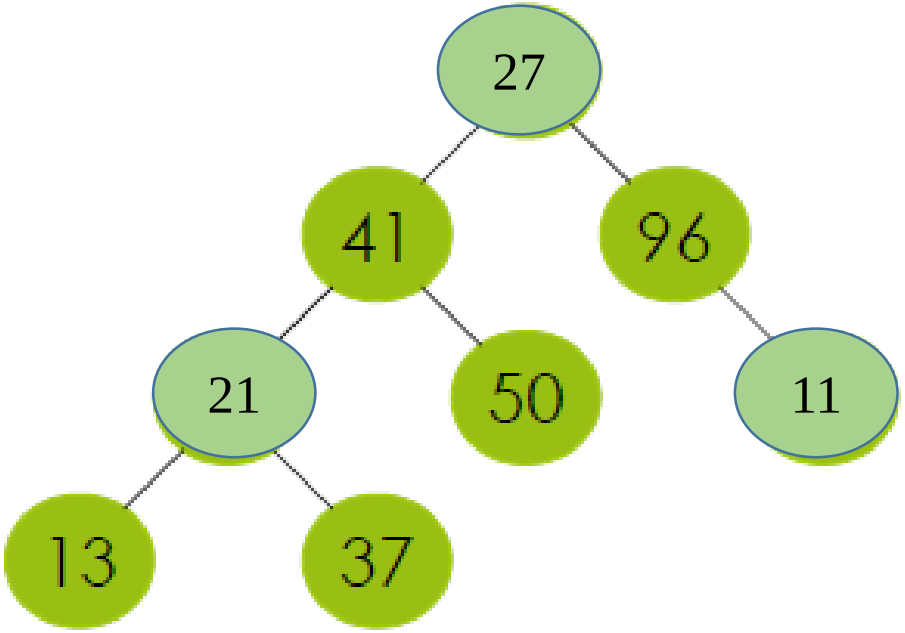




Trees

- A **tree** is a hierarchical data structure.
  - Every tree has a **node** called the **root**.
  - Each node can have 1 or more nodes as **children**.
  - A node that has no children is called a **leaf**.
- A common tree in computing is a **binary tree**.
  - A binary tree consists of nodes that have at most 2 children.
- Applications: data compression, file storage, game trees

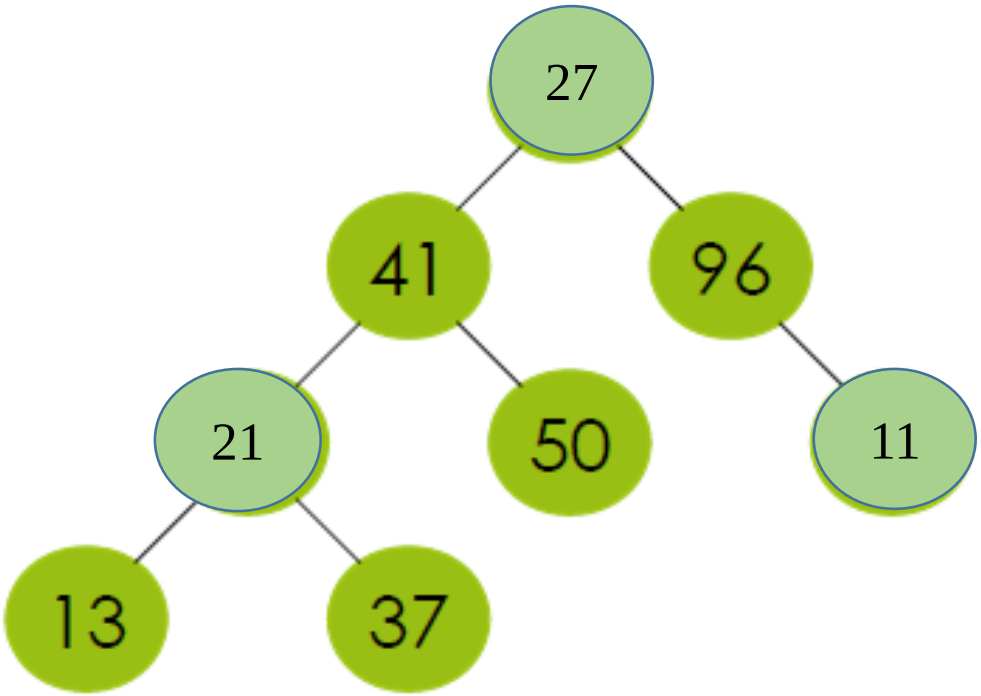
Binary Tree [1/4]



In order to illustrate main ideas we label the tree nodes with the keys only.
In fact, every node would also store the rest of the data associated with that key. Assume that our tree contains integers keys.

Which one is the root?
Which ones are the leaves (external nodes)?
Which ones are internal nodes?
What is the height of this tree?

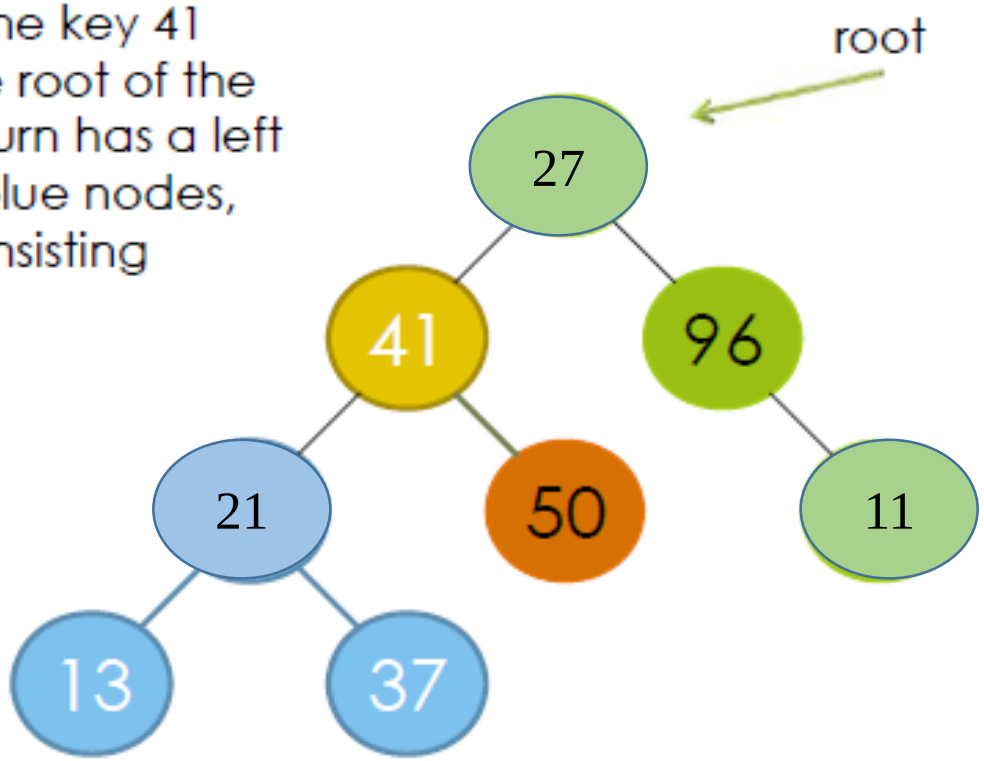
Binary Tree [2/4]



The root contains the data value 84.
There are 4 leaves in this binary tree: nodes containing 13, 37, 50, 98.
There are 3 internal nodes in this binary tree: nodes containing 41, 96, 24
This binary tree has height 3 – considering root is at level 0,
the maximum level among all nodes is 3

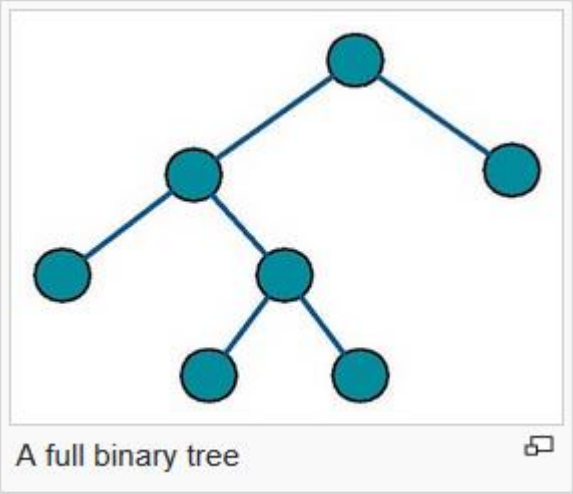
Binary Tree [3/4]

Note the **recursive** structure:
The yellow node with the key 41 can be viewed as the root of the left subtree , which in turn has a left subtree consisting of blue nodes, and a right subtree consisting of orange nodes.

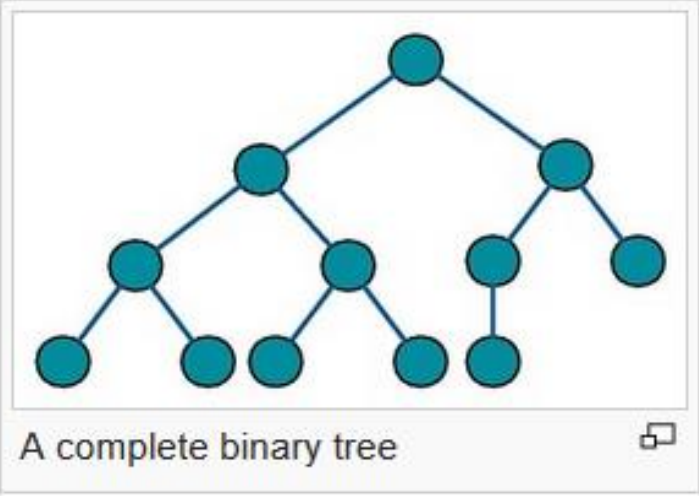


Binary Tree [4/4]

- A **rooted** binary tree has a root node and every node has at most two children.
- A **full** binary tree (sometimes referred to as a **proper** or **plane** binary tree) is a tree in which every node in the tree has either 0 or 2 children
- In a **complete** binary tree every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes at the last level h.



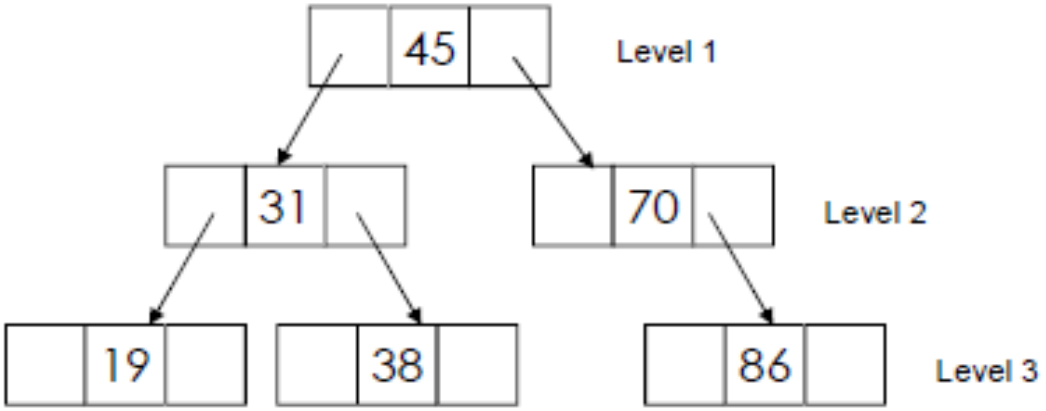
A full binary tree



A complete binary tree

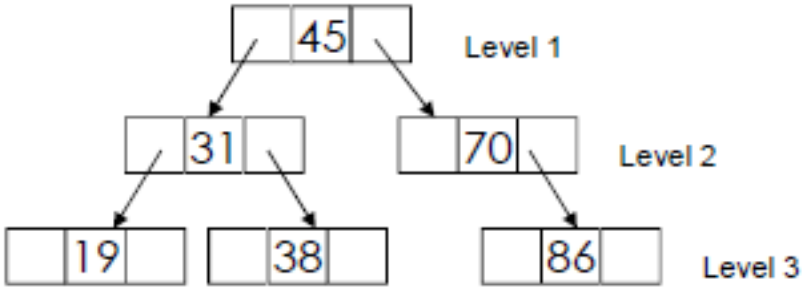
Binary Tree Implementation

- One common implementation of binary trees uses nodes like a linked list does.
  - Instead of having a "next" pointer, each node has a "left" pointer and a "right" pointer.



Binary Tree Implementation using Python List [1/2] using Nest Lists

- Languages like Python do not let programmers manipulate pointers explicitly.
- We could use Python lists to implement binary trees. For example:



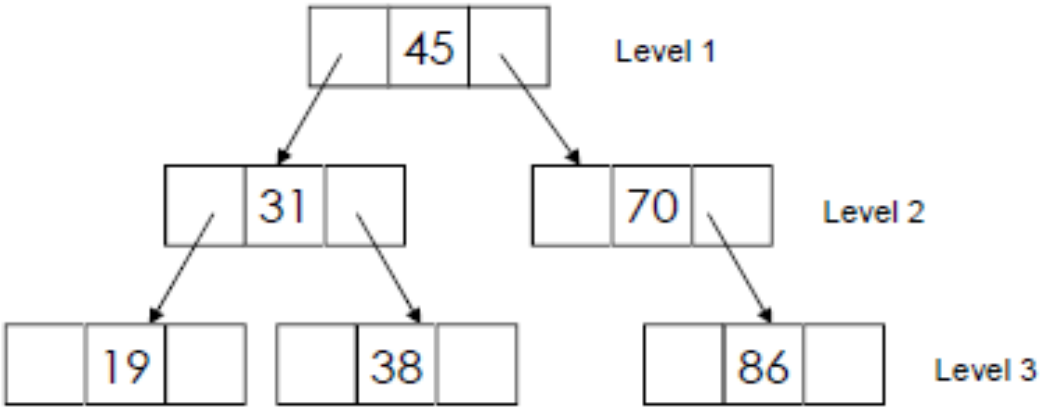

```
[45, left, right]

[45,[31,left,right],[70,left,right]]

[45, [31, [19,[],[]], [38,[],[]]],
      [70,[], [86, [], []]]
]
```

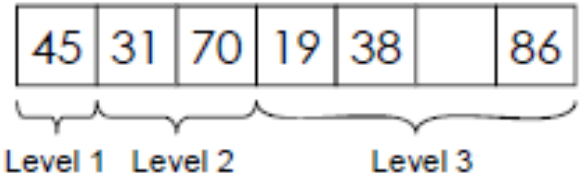
[ ] stands for an empty tree

Arrows point to subtrees

Binary Tree Implementation using Python List [2/2] using 1D Lists

- We could also use a flat (one-dimensional list).





Operations for Dynamic Data Set like Tree

- Insert
- Delete
- Search
- Find min/max
- ...

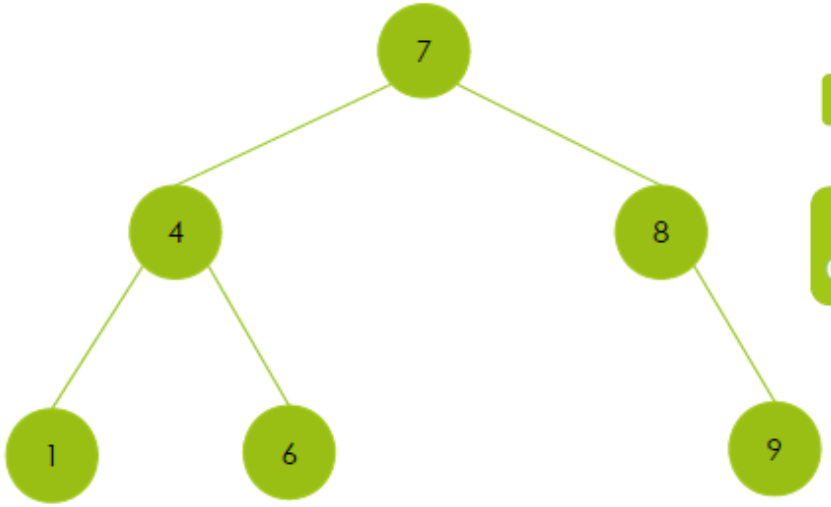
Choosing a specific data structure has consequences on which operations can be performed faster.

Search

- Binary Search
- Binary Tree
- Binary Search Tree

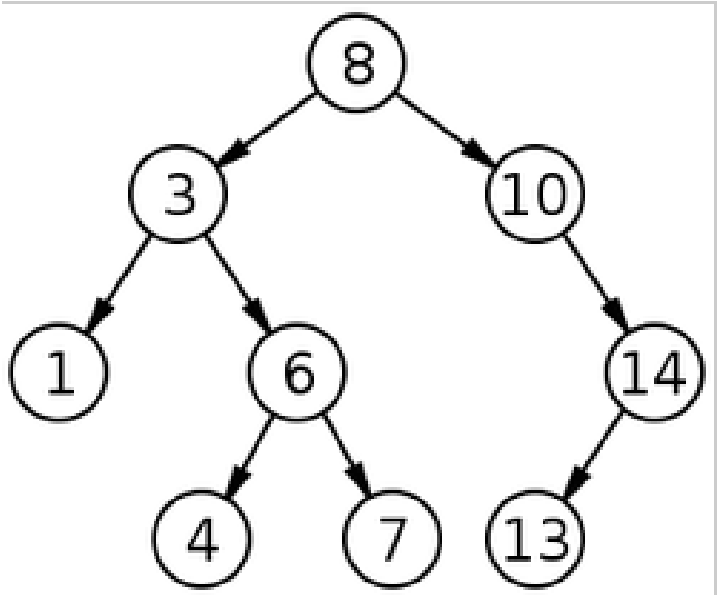
Binary Search Tree

BST ordering invariant: At any node with key *k*, all keys of elements in the left subtree are strictly less than *k* and all keys of elements in the rightsubtree are strictly greater than *k* (assume that there are no duplicates in the tree)



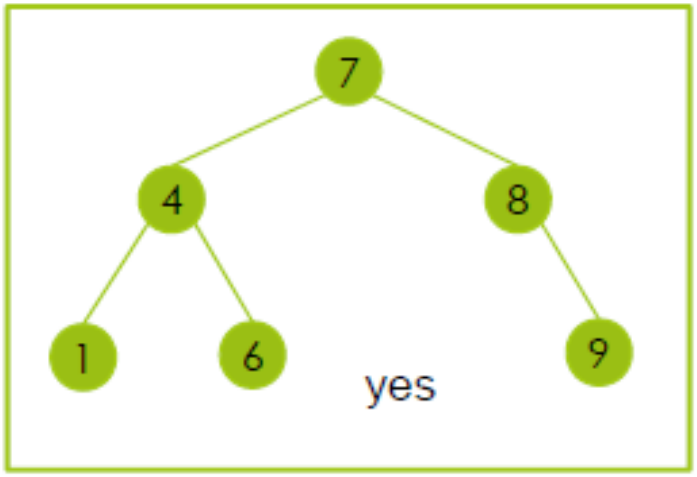
Binary tree

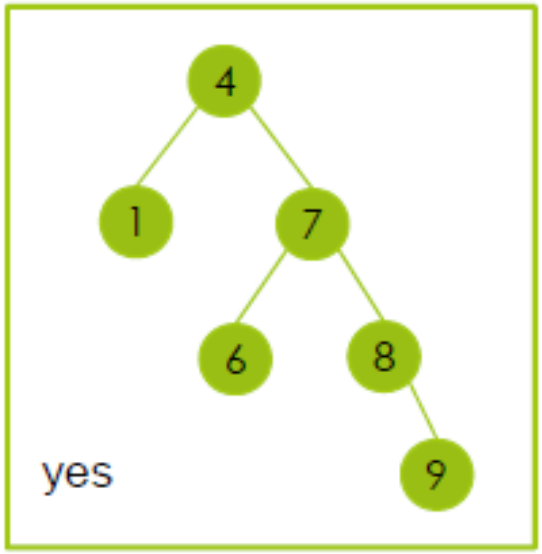
Satisfies the ordering invariant

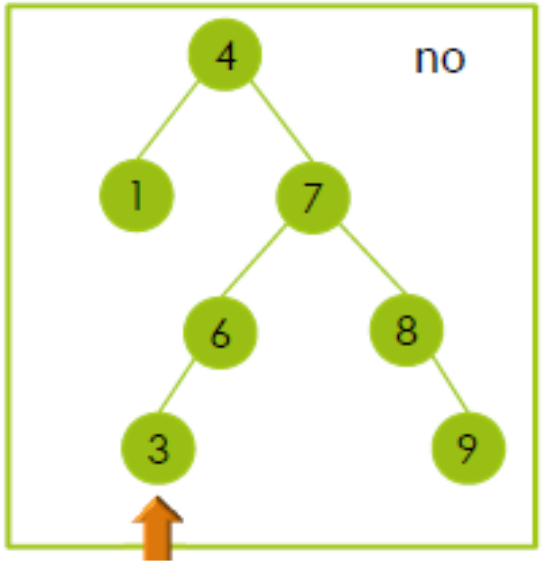


A binary search tree of size 9 and depth 3, with 8 at the root. The

Test: Is this a BST?





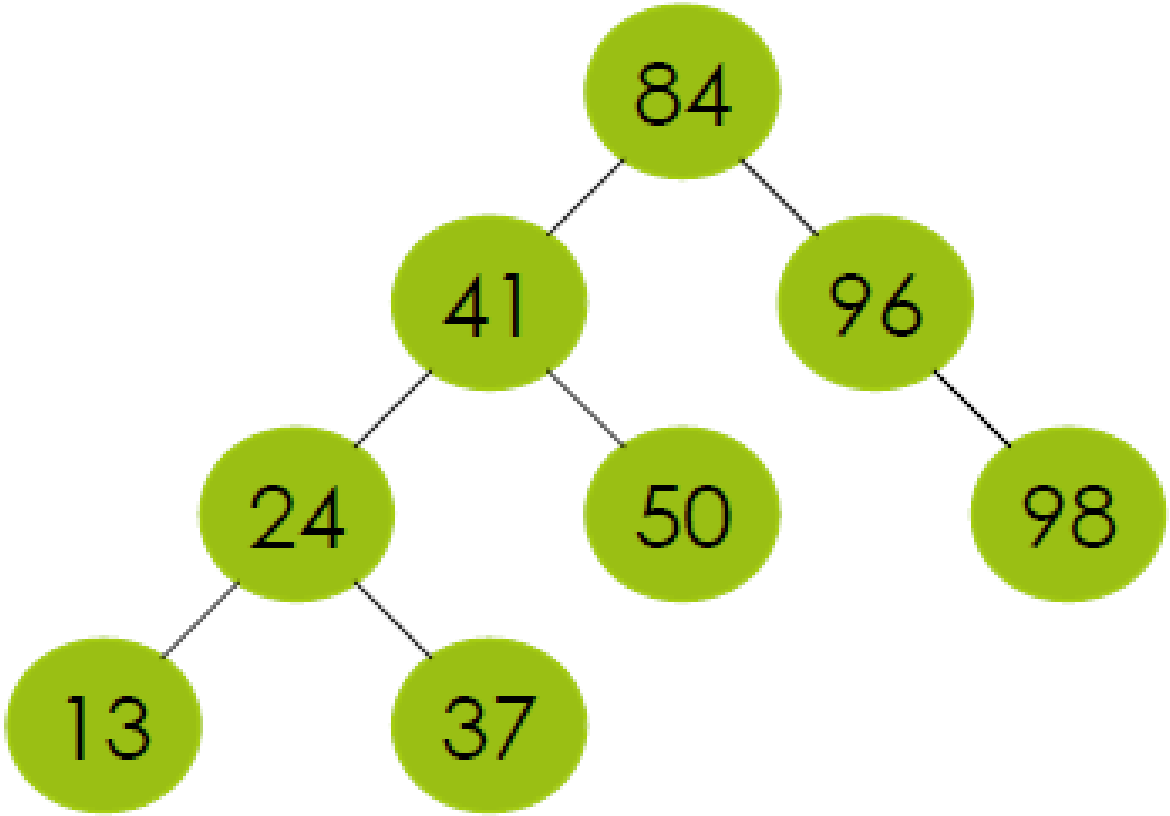


Inserting into a BST

- For each data value that you wish to insert into the binary search tree:
  - Start at the root and compare the new data value with the root.
  - If it is less, move down left. If it is greater, move down right.
  - Repeat on the child of the root until you end up in a position that has no node.
  - Insert a new node at this empty position.

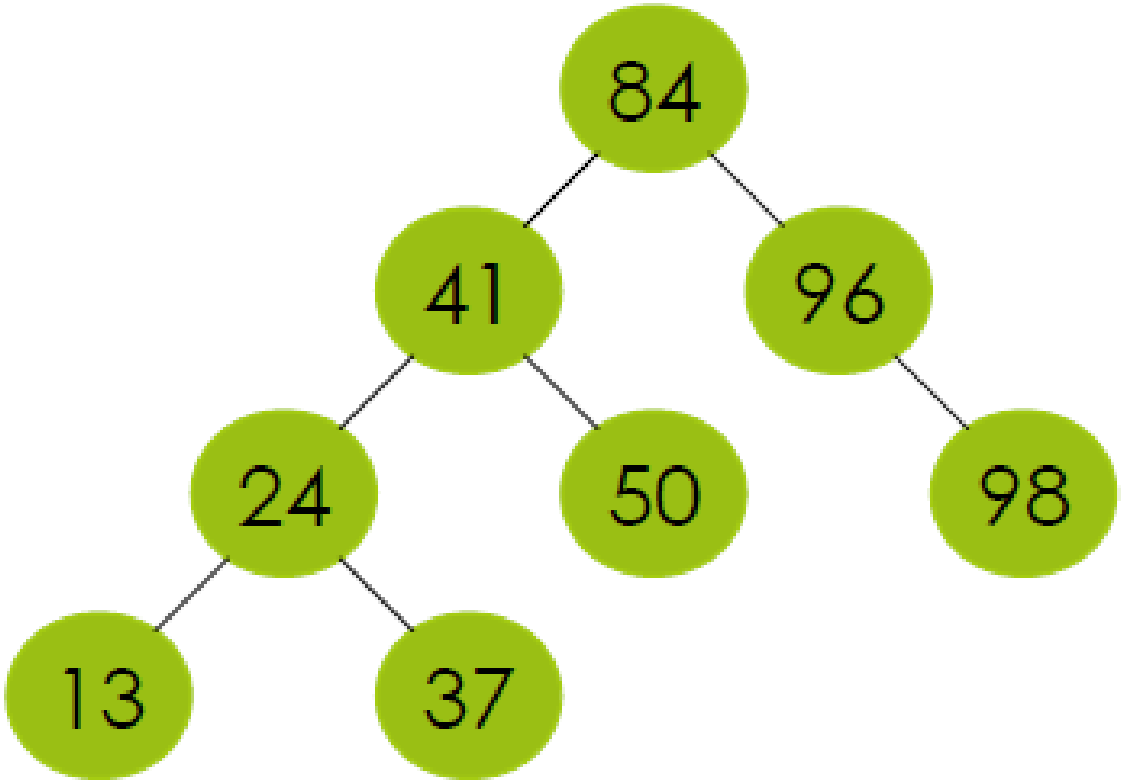
Building Binary Search Tree

- Insert: 84, 41, 96, 24, 37, 50, 13, 98



Search in Binary Search Tree [1/3]

- How would you search for an element in a BST?

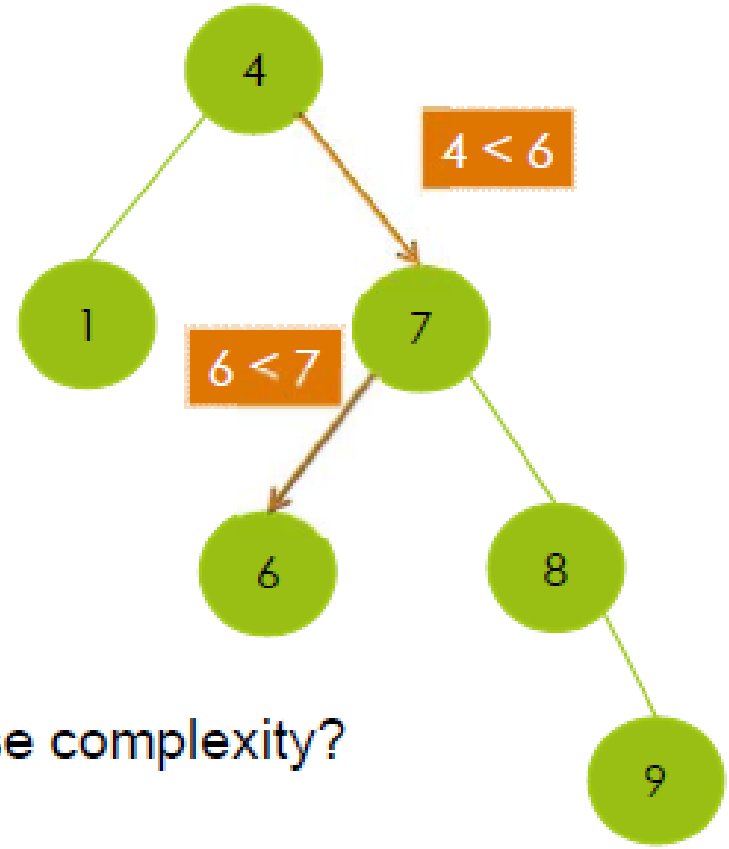


Search in Binary Search Tree [2/3]

- For the key that you wish to search
  - Start at the root and compare the key with the root. If equal, key found.
  - Otherwise
    - If it is less, move down left. If it is greater, move down right. Repeat search on the child of the root.
    - If there is no non-empty subtree to move to, then key not found.

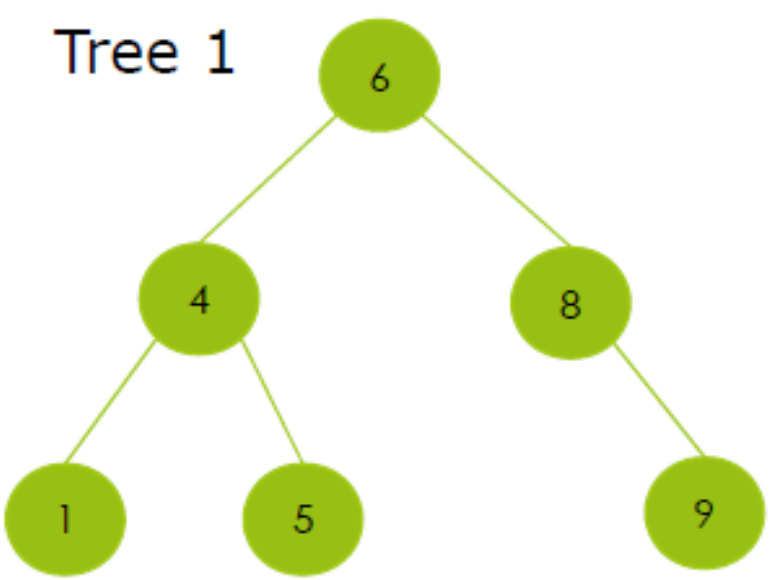
Search in Binary Search Tree [3/3]

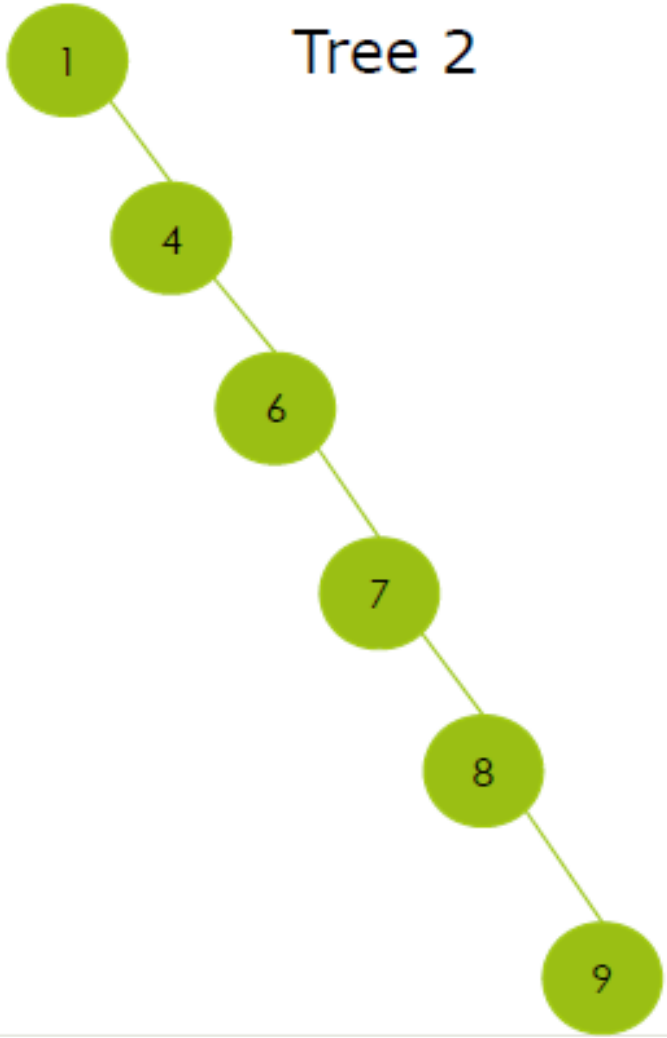
Example: searching for 6



Can we form a conjecture about worst case complexity?

Time Complexity of Search in Tree [1/3]



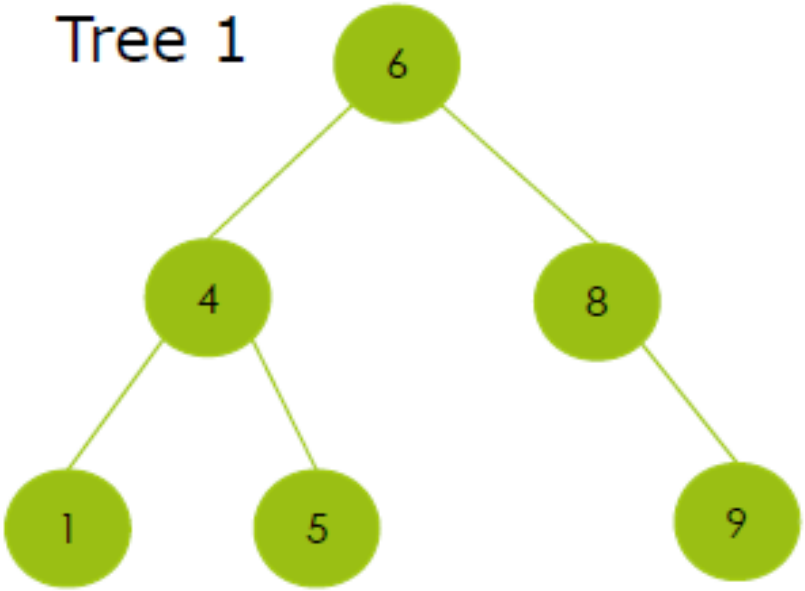


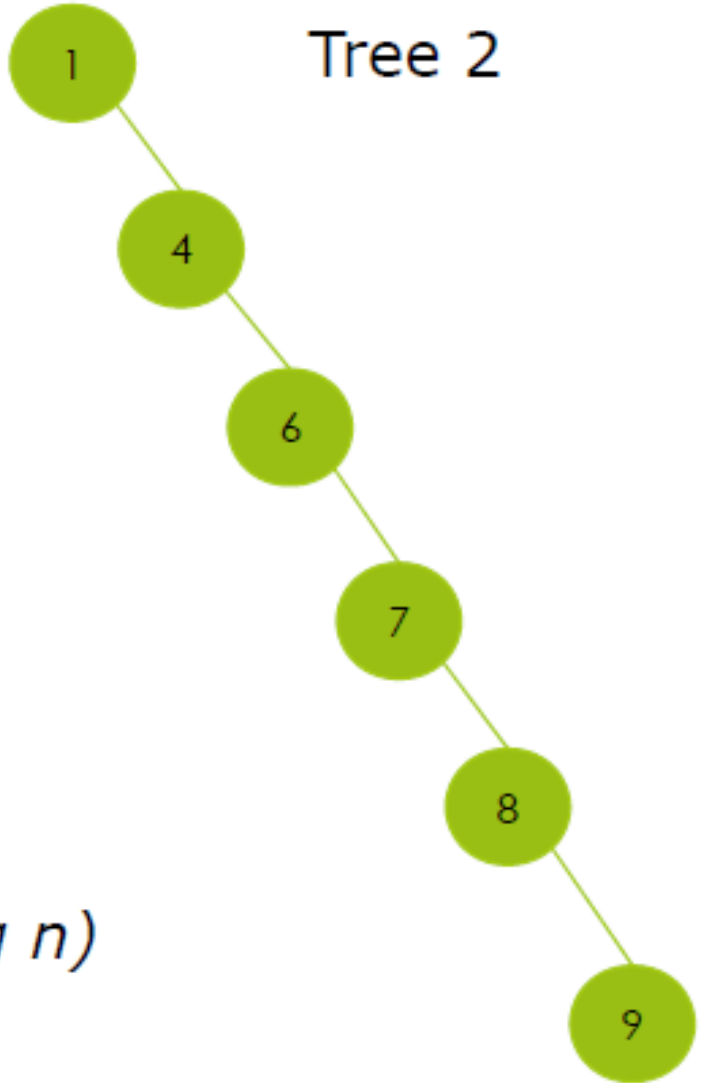
Number of nodes: *n*

Worst case: *O(height)*

Worst *height*: *n* ⇒ *O(n)*

Time Complexity of Search in Tree [2/3]



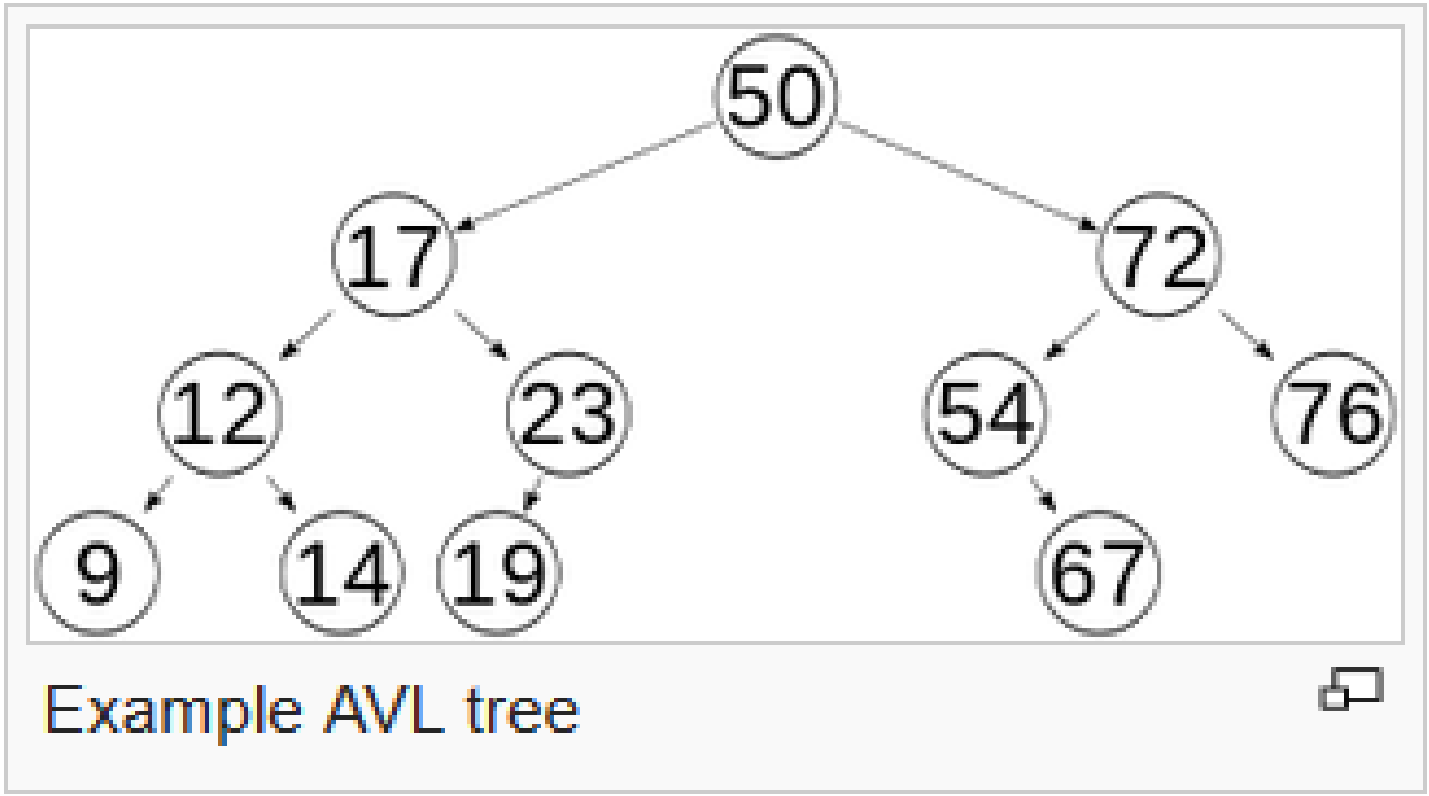


Number of nodes: *n*

What if we could always have balanced trees? ⇒ *O(log n)*

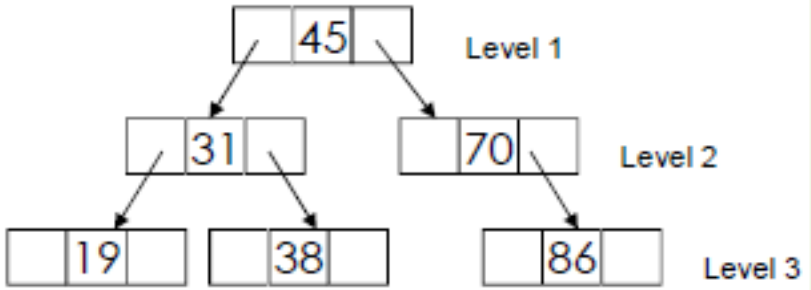
Time Complexity of Search in Tree [3/3]

- an **AVL tree** (Georgy Adelson-Velsky and Evgenii Landis) tree
- A self-balancing binary search tree
- 모든 Node의 left subtree와 right subtree의 height difference가 less than one!



Example AVL tree

Binary Search Tree Implementation using Nested List




```
[45, left, right]

[45,[31,left,right],[70,left,right]]

[45, [31, [19,[],[]], [38,[],[]]],
      [70,[], [86, [], []]]
]
```

[ ] stands for an empty tree

Arrows point to subtrees

Python Code for BST using Nested Lists

```
def bst_build(slist):

    if len(slist) == 0:
        return []
    if len(slist) == 1:
        return slist + [[]] + [[]]

    if len(slist) % 2 == 1:
            mid = len(slist) // 2
    else:
            mid = (len(slist) // 2) - 1

    print(" root: ", slist[mid]," left tree:  ", slist[0:mid], " right tree: ", slist[mid+1:])

    return [slist[mid]] + [bst_build(slist[0:mid])] + [bst_build(slist[mid+1:])]


def bst_in_list(lst):
    slist = sorted(lst)
    print("The sorted data: ", slist)
    return bst_build(slist)


def bst_search(bs_tree, key):
    if bs_tree == []:
        return print("Sorry, there is no such key in the BST")
    if bs_tree[0] == key:
        return print("Yes, The key is in the BST")
    if bs_tree[0] > key:
        bst_search(bs_tree[1], key)
    else:
        bst_search(bs_tree[2], key)
```

Insertions and Deletions are very very inconvenient in this representation