

Ch 12: Sorting and Searching

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort



Naïve Sorting $O(n*n)$

Efficient Sorting $O(n*\log(n))$

Naive Sorting: Selection Sort [1/3]

Sorting을 진행하면서 산출물을 Result List에 저장



Naive Sorting: Selection Sort

[2/3]

Sorted item을 담은 result_list 를 따로 두고 할 수도 있지만
resource 를 생각해서 input_list 에서 그냥 수행

29, 64, 73, 34, **20**

20, **64**, 73, 34, **29**

20, 29, **73**, **34**, 64

20, 29, 34, **73**, **64**

20, 29, 34, 64, 73

가장 작은값을 찾아서 첫번째
자리에 있는 값과 교체

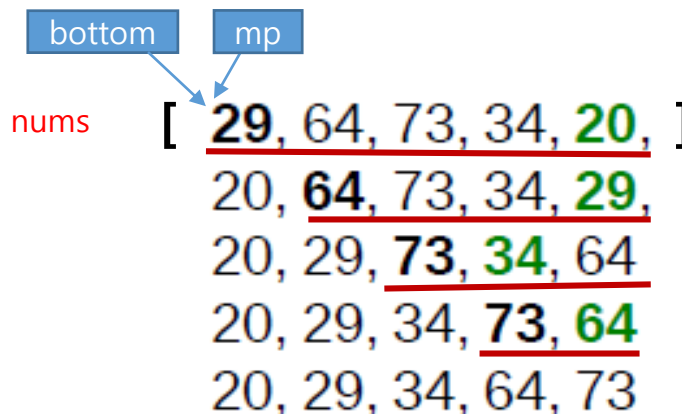
나머지에서 가장 작은값을 찾아서
두번째 자리에 있는 값과 교체



Selection Sort Code

[3/3]

```
def selSort(nums):  
    # sort nums into ascending order  
  
    n = len(nums)  
  
    # For each position in the list (except the very last)  
  
    for bottom in range(n-1):  
        # find the smallest item in nums[bottom]...nums[n-1]  
  
        mp = bottom                # bottom is smallest initially  
        for i in range(bottom+1, n): # look at each position  
            if nums[i] < nums[mp]:    # for loop 이 끝날때 가장 작은값의 Index는 i  
                mp = i              # i를 mp에 저장  
  
        # swap the smallest item to the bottom  
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```



가장 작은값을 찾아서 첫번째
자리에 있는 값과 교체

Insertion Sort Shot

[1/3]

Sorting을 진행하면서 산출물을 Result List에 저장

7을 어디에 insert 할까?
5을 어디에 insert 할까?
42을 어디에 insert 할까?
6을 어디에 insert 할까?



		7	5	42	6	3	15
1		7	5	42	6	3	15
2		7	5	42	6	3	15
3		7	5	42	6	3	15
4		7	5	42	6	3	15
5		7	5	42	6	3	15
6		7	5	42	6	3	15

Result List

7
5, 7
5, 7, 42
3, 5, 6, 7, 15, 42

Insertion Sort Shot

[2/3]

7을 어디에 insert 할까?

5을 어디에 insert 할까?

42을 어디에 insert 할까?

6을 어디에 insert 할까?



Sorted item을 담는 result_list 를 따로
두고 할 수도 있지만 resource 를
생각해서 input_list 에서 그냥 수행

1		7	5	42	6	3	15
2							
3		7	5	42	6	3	15
4							
5		5	7	42	6	3	15
6							
7		5	7	42	6	3	15
8							
9		5	7	42	6	3	15
10							
11		5	7	42	6	3	15
12							
13		5	7	6	42	3	15
14							
15		5	6	7	42	3	15
16							
17		5	6	7	42	3	15
18							
19		3	5	6	7	42	15
20							
21		3	5	6	7	42	15
22							
23		3	5	6	7	15	42
24							

Insertion Sort Code

[3/3]

http://en.wikipedia.org/wiki/Insertion_sort

Insertion sort works by taking elements from the unsorted list and inserting them at the right place in a new sorted list. The sorted list is empty in the beginning. Since the total number of elements in the new and old list stays the same, we can use the same list to represent the sorted and the unsorted sections.

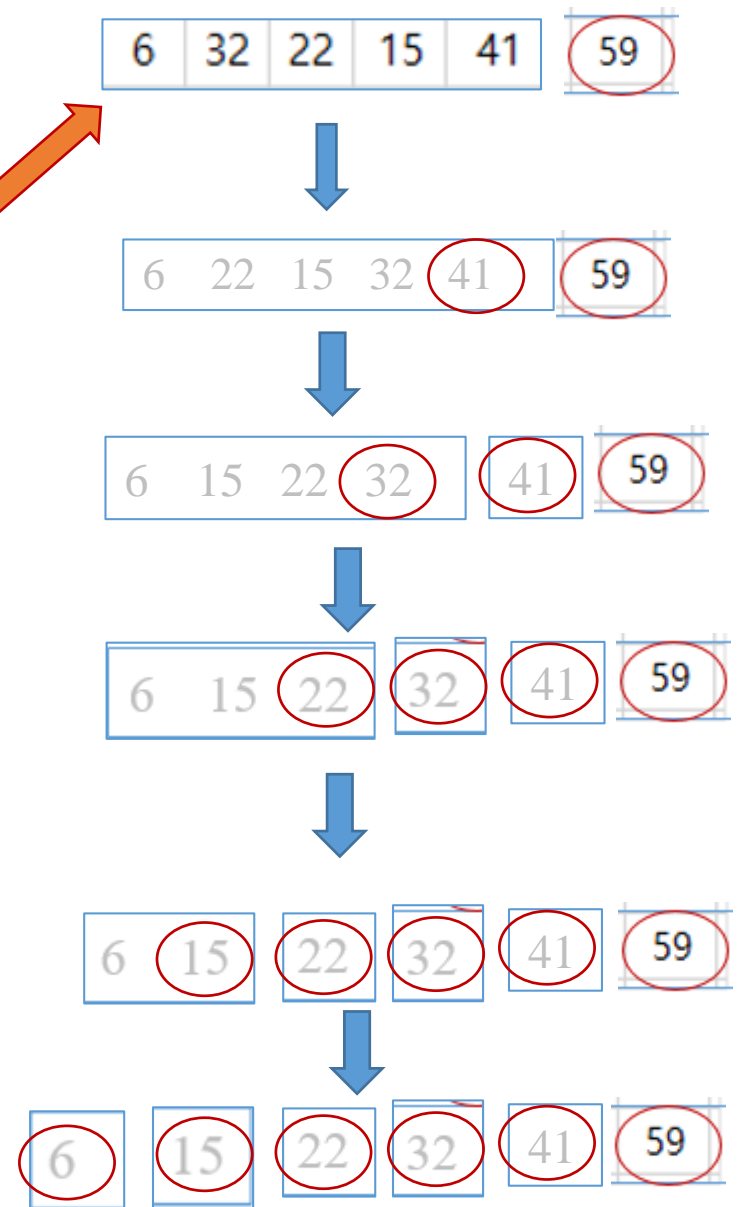
Insertion Sort in Python

```
1 def insertion_sort(items):
2     """ Implementation of insertion sort """
3     for i in range(1, len(items)):
4         j = i
5         while j > 0 and items[j] < items[j-1]:
6             items[j], items[j-1] = items[j-1], items[j]
7             j -= 1
```

처음에는 empty인 sorted list를 두고, Unsorted list에서 1개씩 element를 sorted list에 이동하면서 sorted list를 유지하도록 insertion 수행

Bubble Sort Shot

	Data					
1	32	6	41	22	15	59
2						
3	32	6	41	22	15	59
4						
5	6	32	41	22	15	59
6						
7	6	32	41	22	15	59
8						
9	6	32	41	22	15	59
10						
11	6	32	22	41	15	59
12						
13	6	32	22	15	41	59
14						
15	6	32	22	15	41	59



Bubble Sort

http://en.wikipedia.org/wiki/Bubble_sort

Bubble sort is one of the most basic sorting algorithm that is the simplest to understand. It's basic idea is to bubble up the largest(or smallest), then the 2nd largest and the 3rd and so on to the end of the list. Each bubble up takes a full sweep through the list.

Bubble Sort in Python

```
1  def bubble_sort(items):
2      """ Implementation of bubble sort """
3      for i in range(len(items)):
4          for j in range(len(items)-1-i):
5              if items[j] > items[j+1]:
6                  items[j], items[j+1] = items[j+1], items[j]      # Swap!
```

1번째 2번째 비교 → 필요한 swap 수행

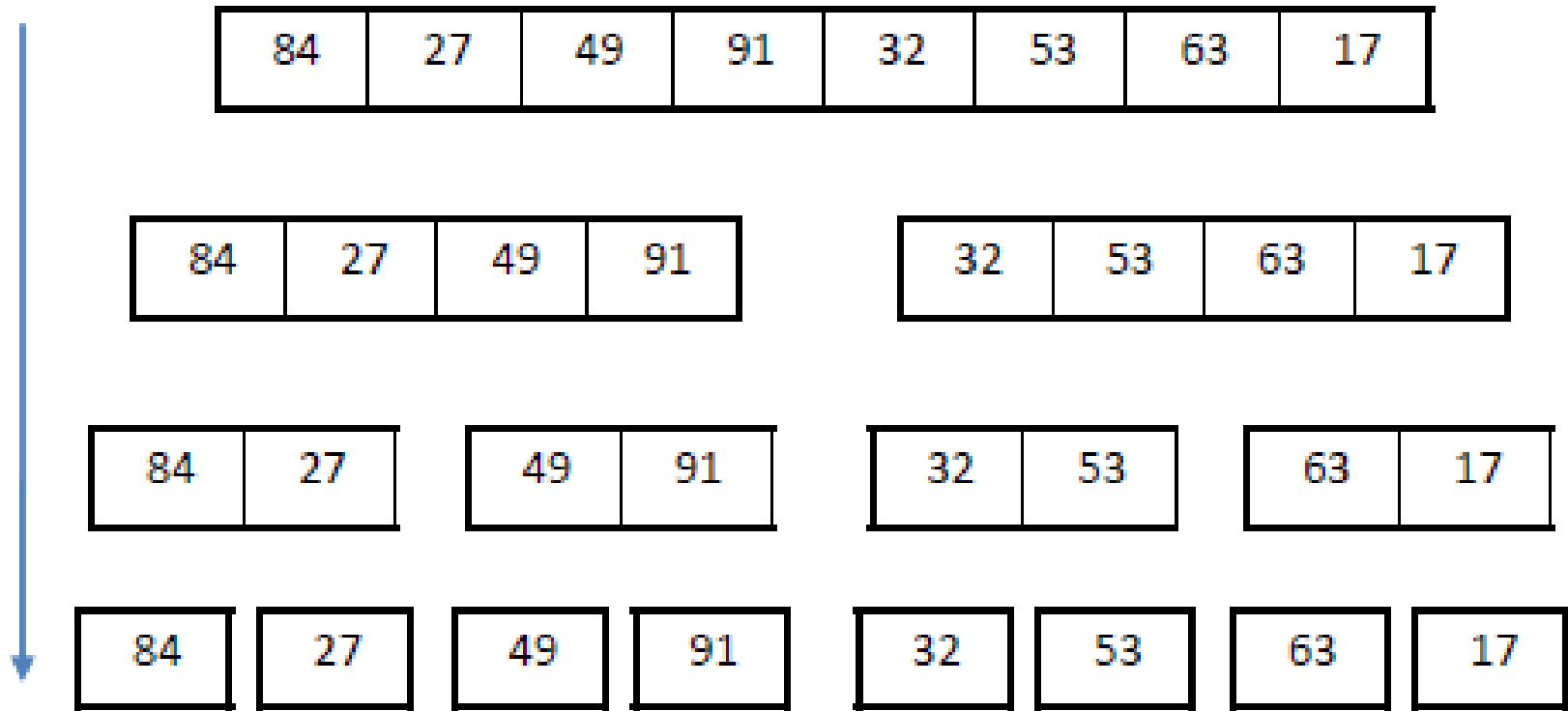
2번째 3번째 비교 → 필요한 swap 수행

...

(N-1)번째 N번째 비교 → 필요한 swap 수행

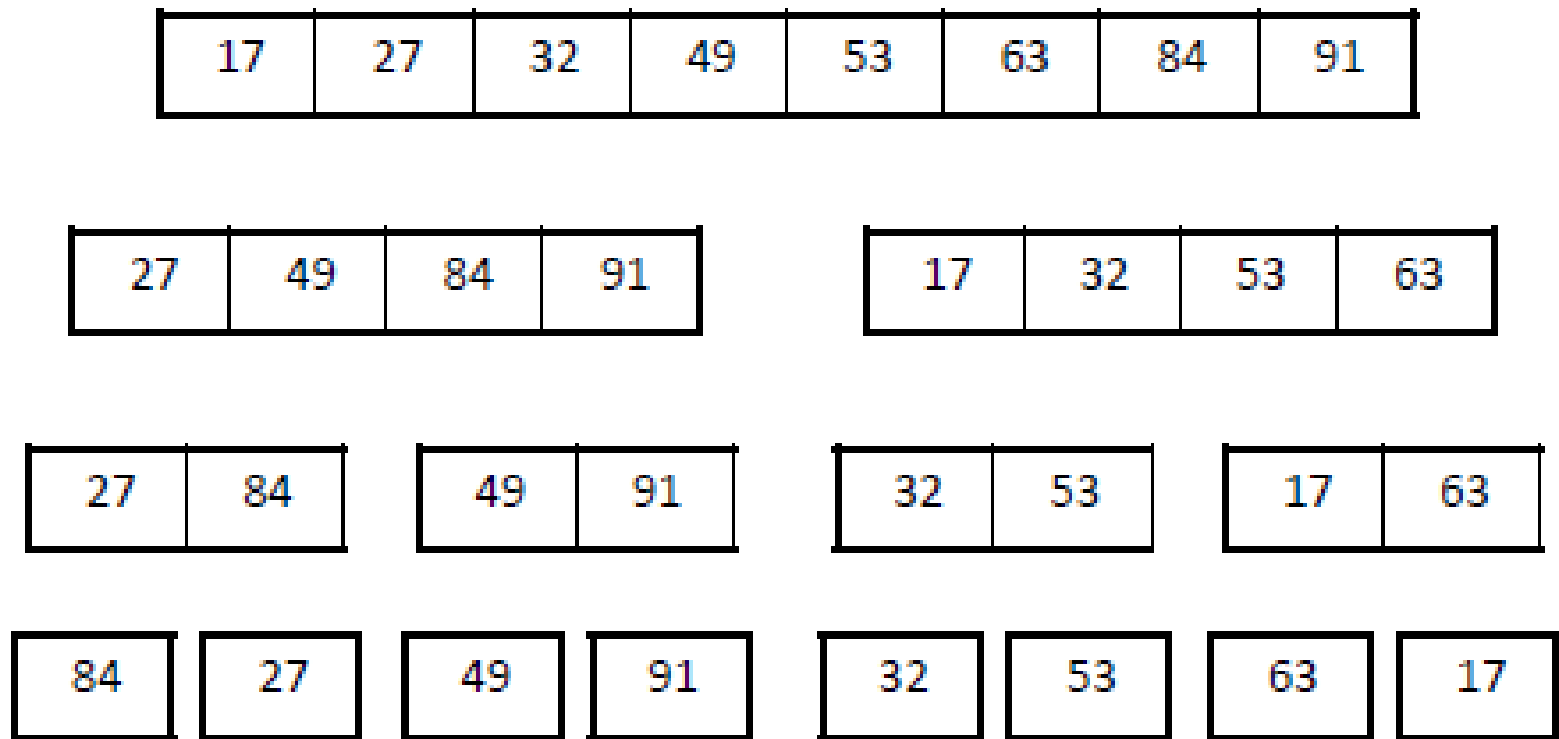
Merge-Sort: Divide Step (Split)

Unsorted Data



Merge-Sort: Conquer Step (Merge)

Final Sorted Data



앞에 있는 Naïve 방법들 보다 computation time은 efficient하지만
Memory resource의 사용은 less efficient

Python code for Merge-Sort

[1/2]

```
def msort(list):  
    if len(list) == 0 or len(list) == 1: # base case  
        return list[:len(list)] # copy the input  
    # recursive case  
    halfway = len(list) // 2  
    list1 = list[0:halfway]  
    list2 = list[halfway:len(list)]  
    newlist1 = msort(list1) # recursively sort left half  
    newlist2 = msort(list2) # recursively sort right half  
    newlist = merge(newlist1, newlist2)  
    return newlist
```

Divide
Step

Python code for Merge-Sort

[2/2]

```
def merge(a, b):  
    index_a = 0 # the current index in list a  
    index_b = 0 # the current index in list b  
    c = []  
    while index_a < len(a) and index_b < len(b):  
        if a[index_a] <= b[index_b]:  
            c.append(a[index_a])  
            index_a = index_a + 1  
        else:  
            c.append(b[index_b])  
            index_b = index_b + 1  
    # when we exit the loop  
    # we are at the end of at least one of the lists  
    c.extend(a[index_a:])  
    c.extend(b[index_b:])  
    return c
```



Quick Sort Shot

1		7	42	6	3	15	12	
2								
3		6	3	7	42	15	12	
4								
5		6	3		7	42	15	12
6								
7		3	6		7	42	15	12
8								
9			3	6	7	42	15	12
10								
11			3	6	7	42	15	12
12								
13			3	6	7		15	12
14								
15			3	6	7		15	12
16								
17			3	6	7		12	15
18								
19			3	6	7	12	15	42

Quick Sort

<http://en.wikipedia.org/wiki/Quicksort>

Quick sort works by first selecting a pivot element from the list. It then creates two lists, one containing elements less than the pivot and the other containing elements higher than the pivot. It then sorts the two lists and join them with the pivot in between. Just like the Merge sort, when the lists are subdivided to lists of size 1, they are considered as already sorted.

Quick Sort in Python

```
1  def quick_sort(items):
2      """ Implementation of quick sort """
3      if len(items) > 1:
4          pivot_index = len(items) // 2
5          smaller_items = []
6          larger_items = []
7
8          for i, val in enumerate(items):
9              if i != pivot_index:
10                 if val < items[pivot_index]:
11                     smaller_items.append(val)
12                 else:
13                     larger_items.append(val)
14
15             quick_sort(smaller_items)
16             quick_sort(larger_items)
17             items[:] = smaller_items + [items[pivot_index]] + larger_items
```

- Step1: Unsorted list의 첫번째 element를 중심으로 전체리스트를 left part와 right part로 재배열
- Step2: 나누어진 left part와 right part에서 step1을 수행

Test data and Time Measurement

```
import random
random_items = [ random.randint(0, 1000) for c in range(1000) ]

import time
startTime = time.clock()
sort_method(random_items)
endTime = time.clock()
elapsedTime = endTime - startTime
print("The elapsed time for sort_method( ) is: ", elapsedTime)
```

`time.clock() ~ time.time()`

`random.randint(n1, n2)` → $n1 \sim n2$ 사이의 random integer

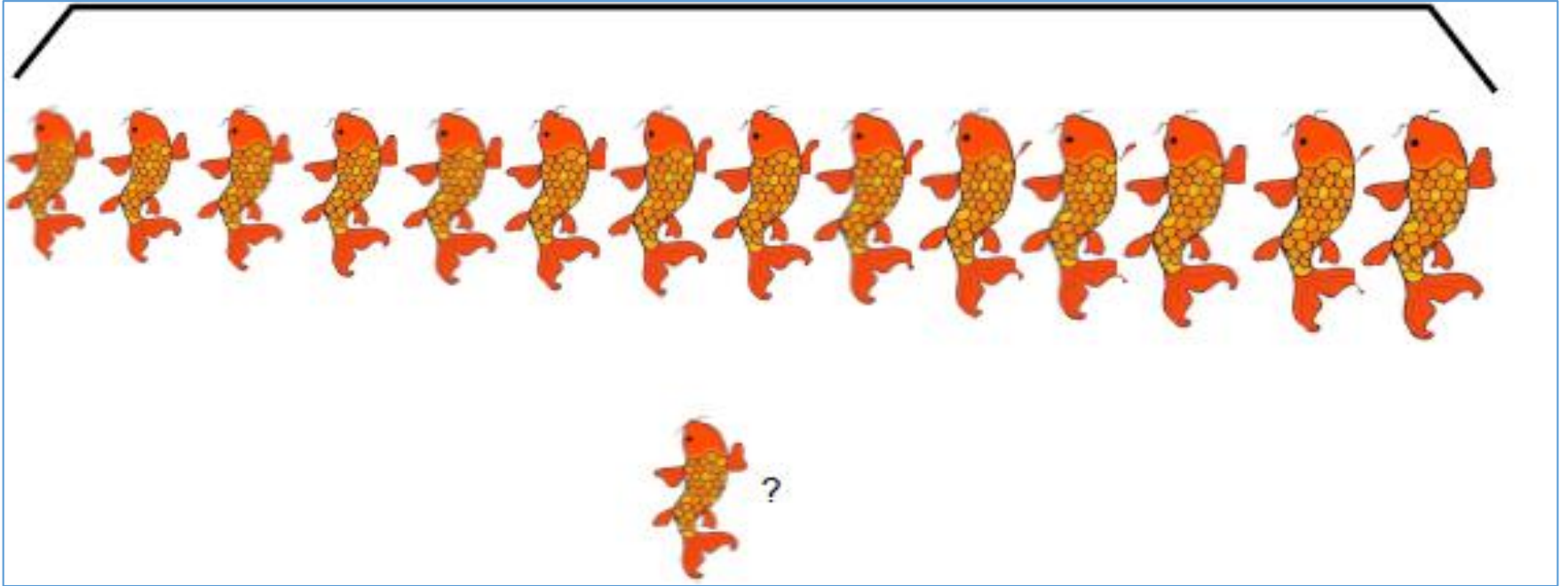
Searching

- Binary Search
- Binary Tree
- Binary Search Tree

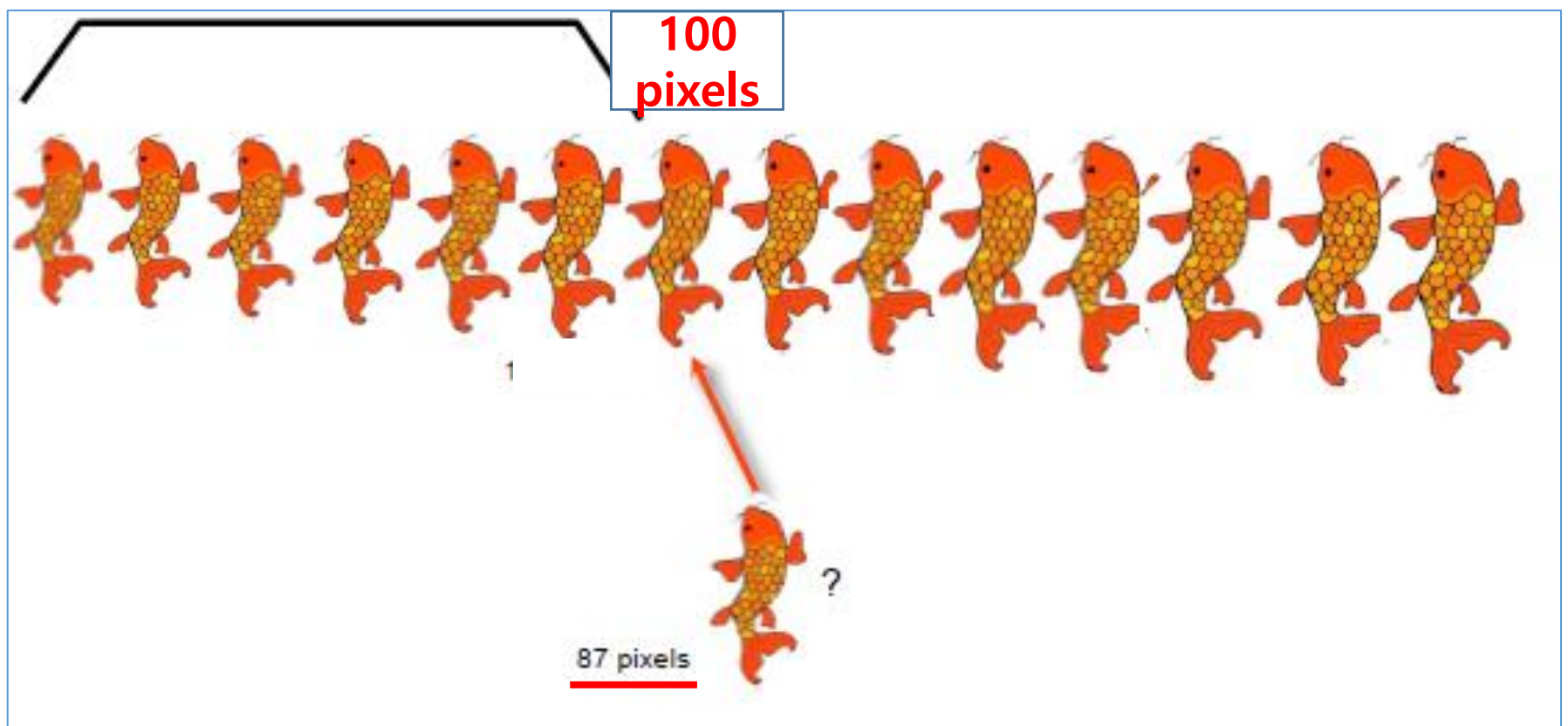
Number Guessing

- I am 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?

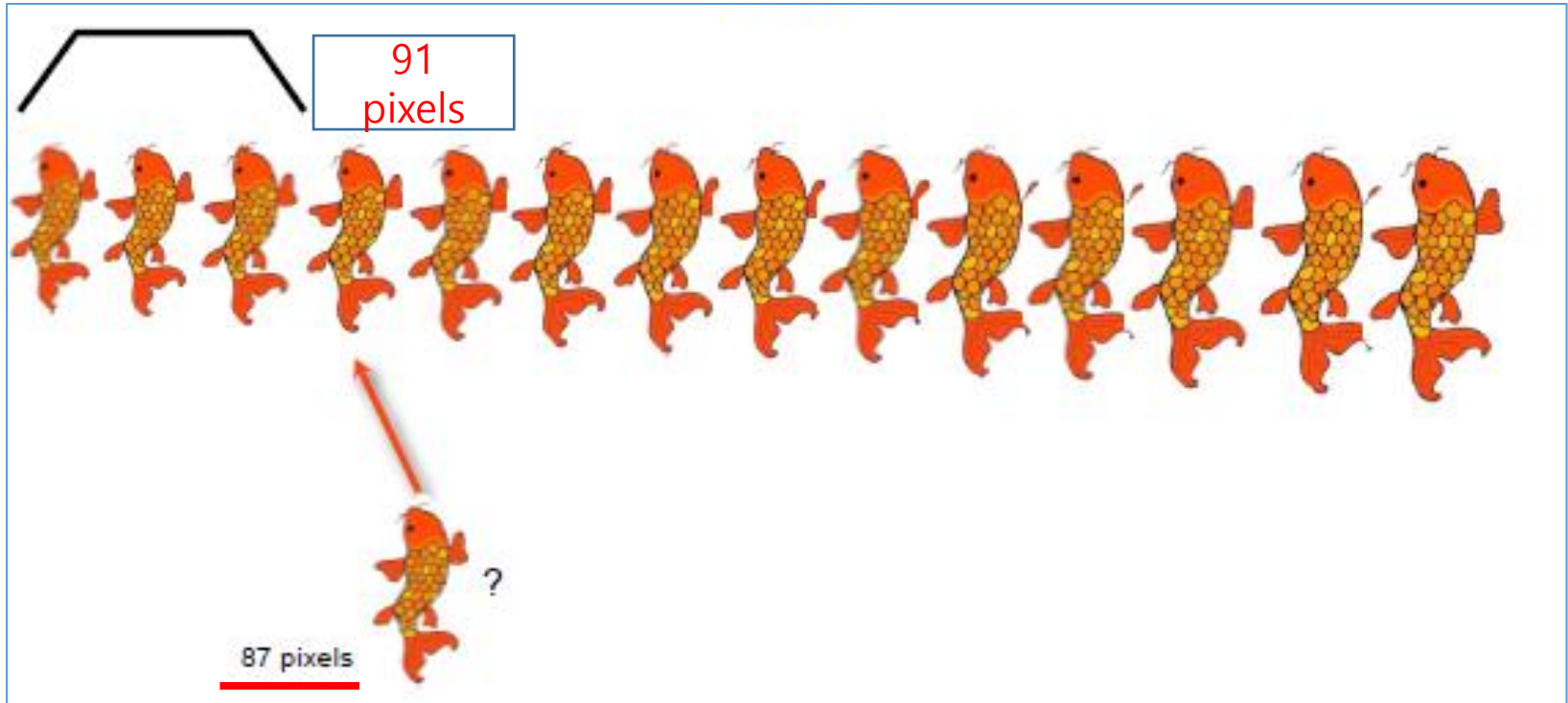
Binary Search in an Ordered List [1/5]



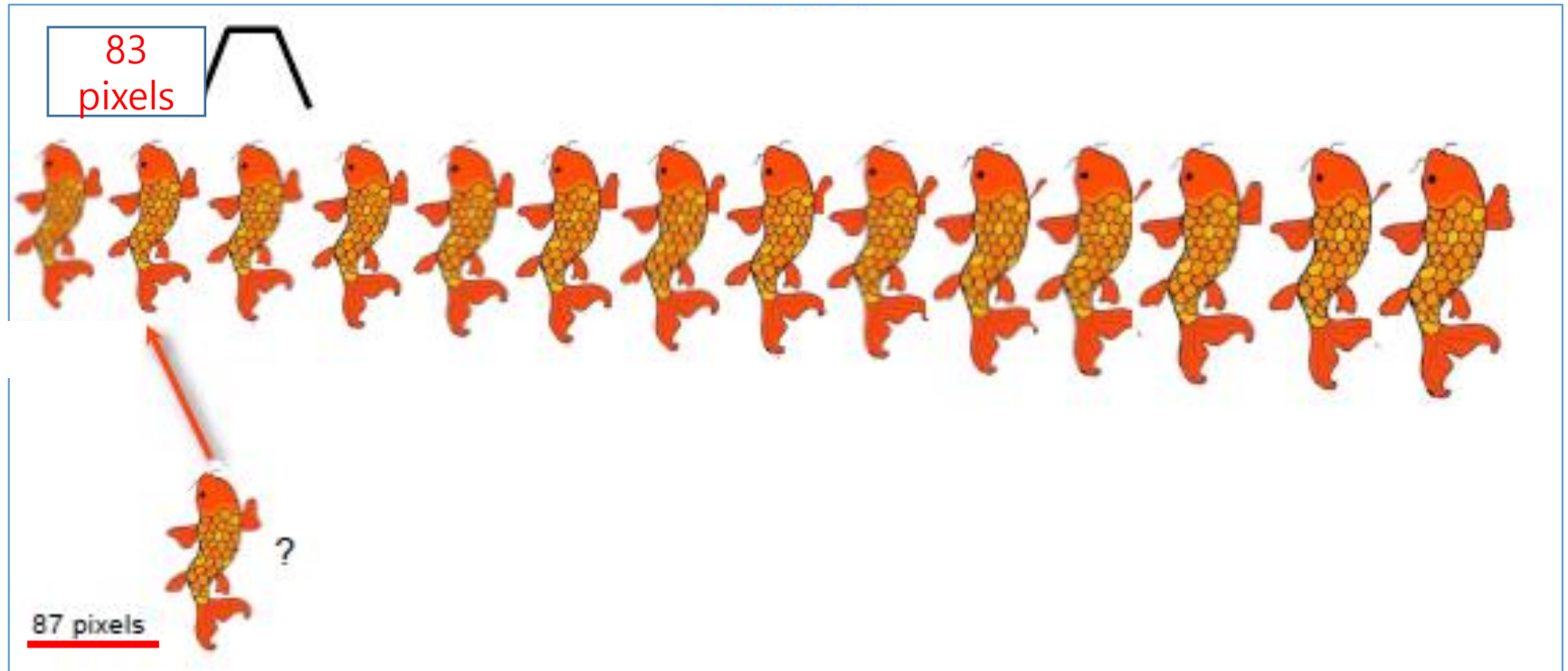
Binary Search in an Ordered List [2/5]



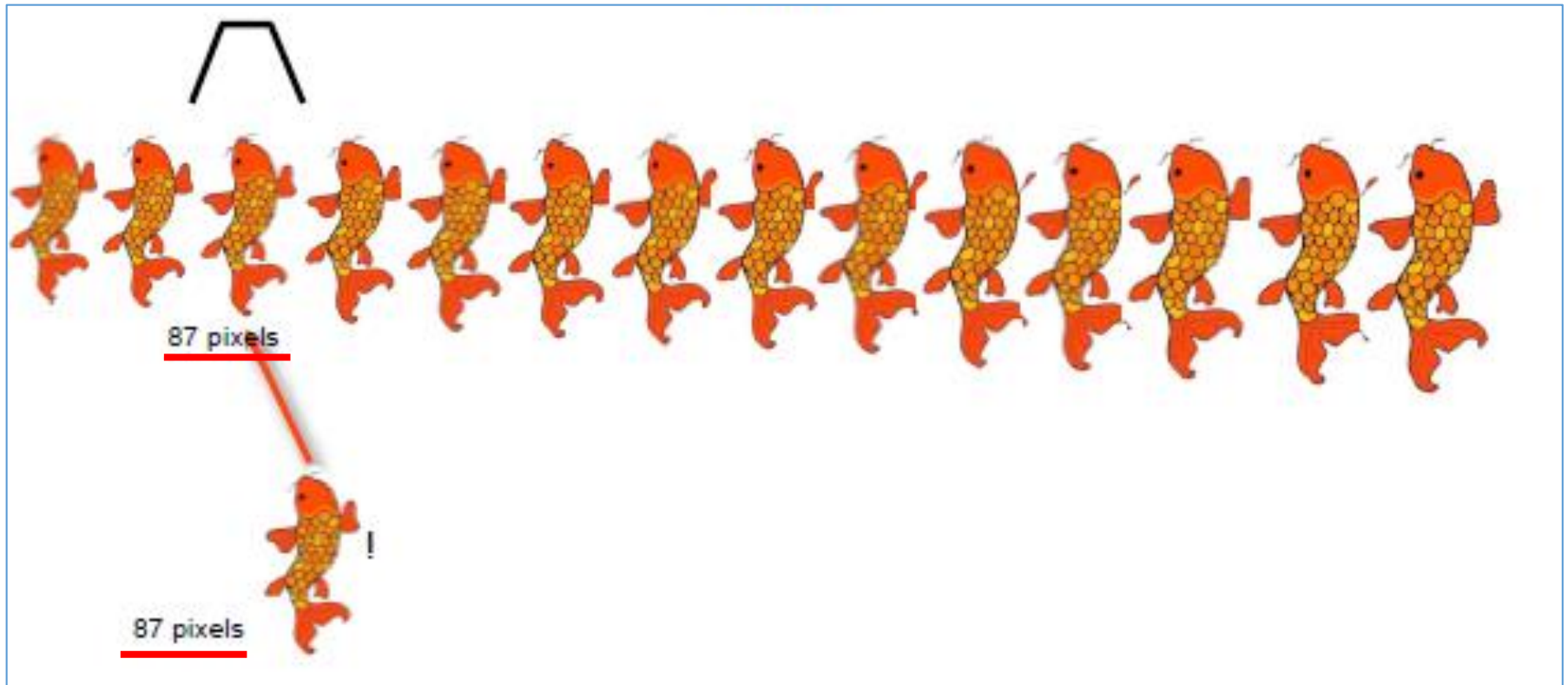
Binary Search in an Ordered List [3/5]



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).

Binary Search Example : 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

Binary Search Example : Search for 42

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

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

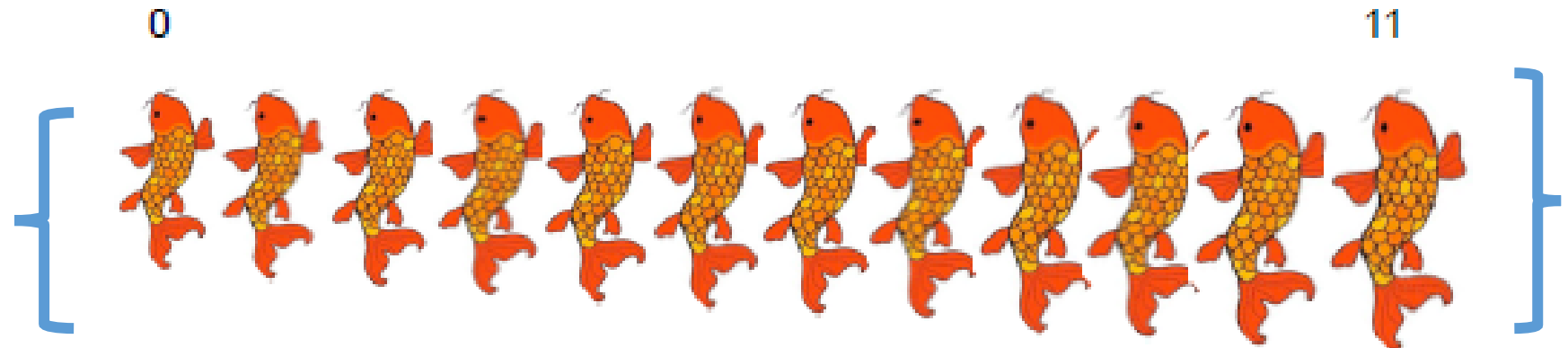
Not found: return None

Coding 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



Coding the Range of the Search [2/2]

- *mid* is the midpoint of the range:

$$mid = \underline{(lower + upper) // 2} \text{ (integer division)}$$

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

(range has 9 elements)

mid = 4



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

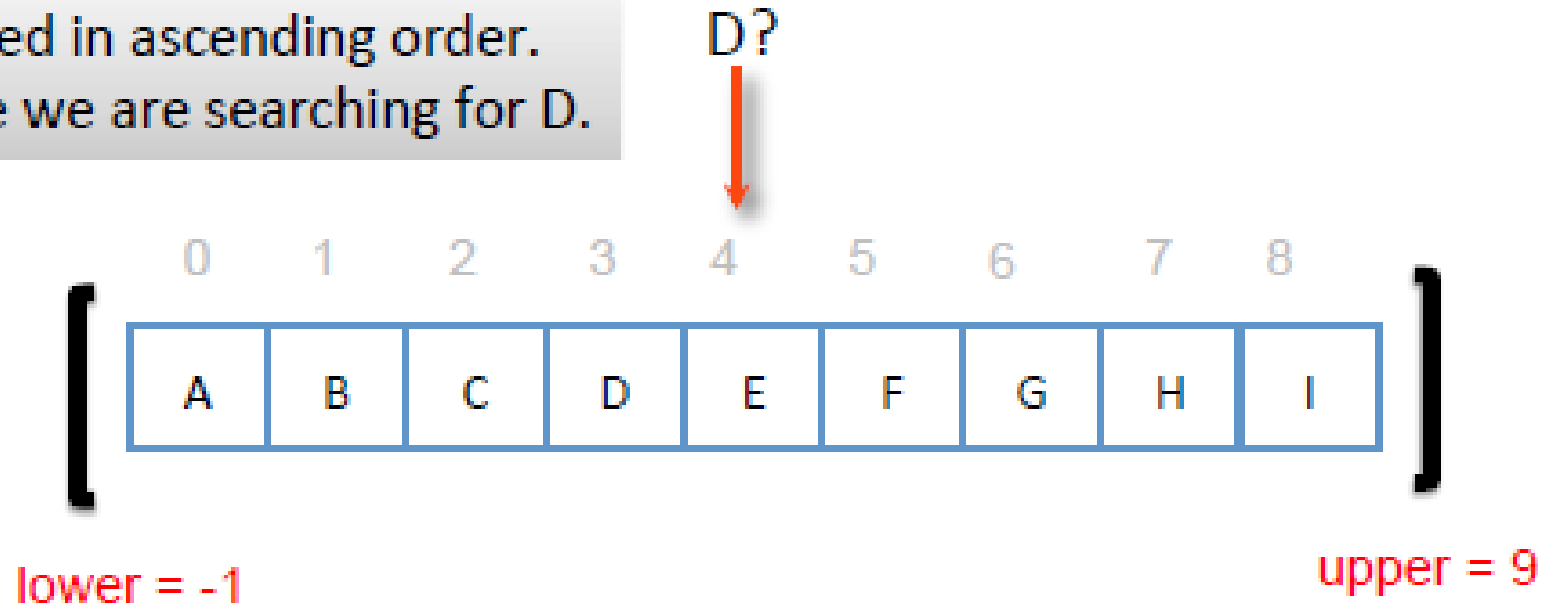
Example: *lower* = -1, *upper* = 8

mid = 3



Midpoint Calculation Example [1/4]

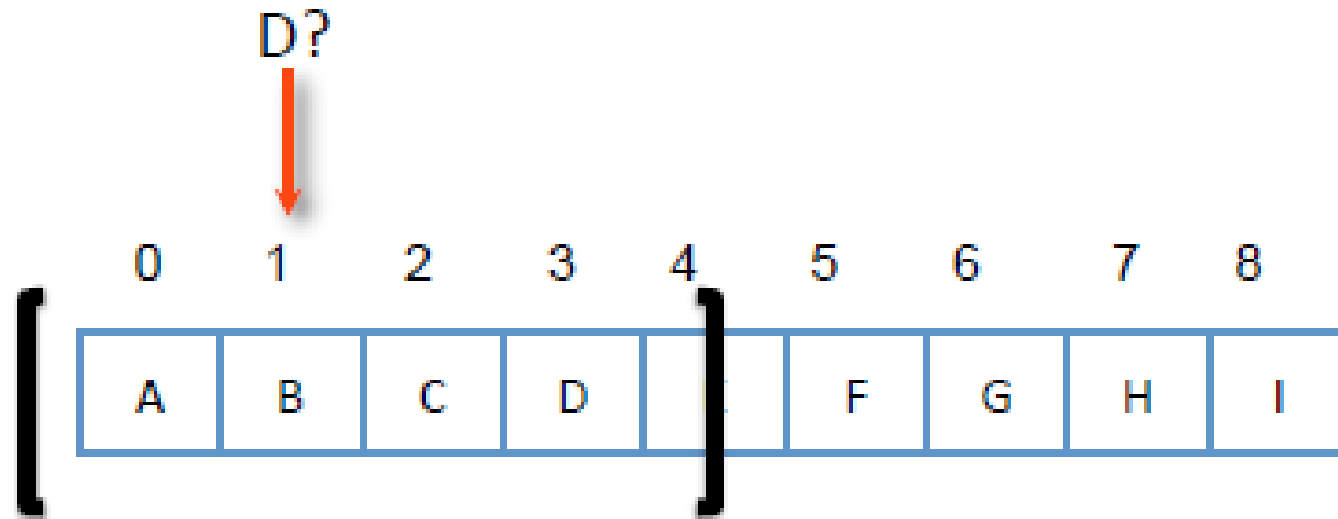
List sorted in ascending order.
Suppose we are searching for D.



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

Midpoint $\rightarrow 8 // 2 \rightarrow 4$

Midpoint Calculation Example [2]



lower = -1

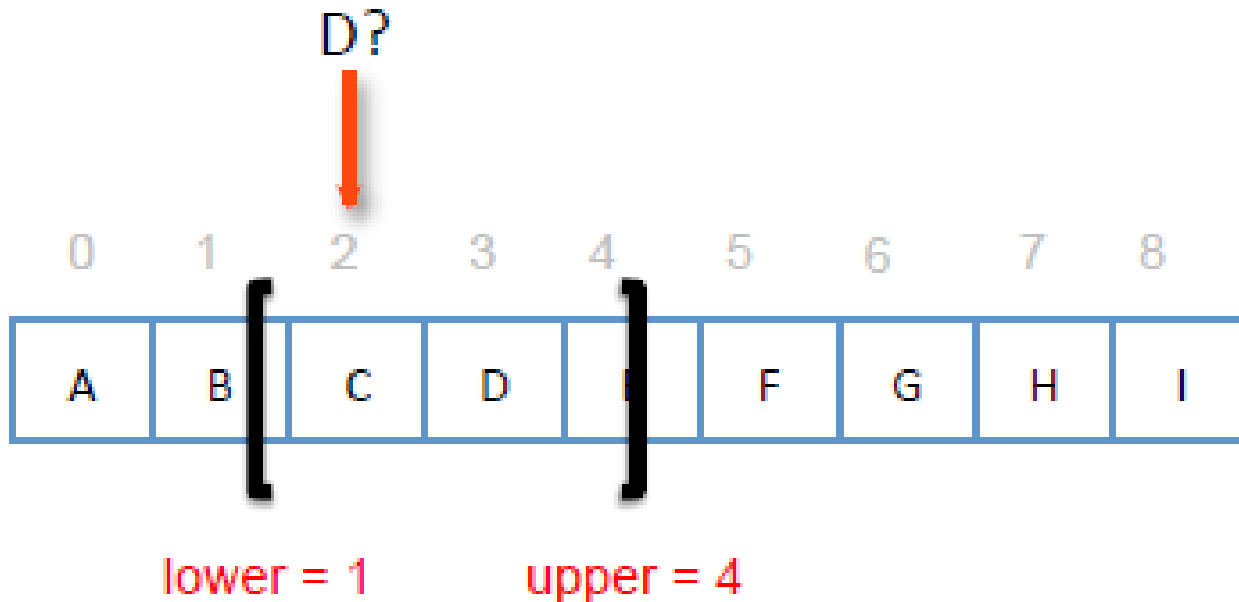
upper = 4

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

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

Midpoint $\rightarrow 3 // 2 \rightarrow 1$

Midpoint Calculation Example [3/4]

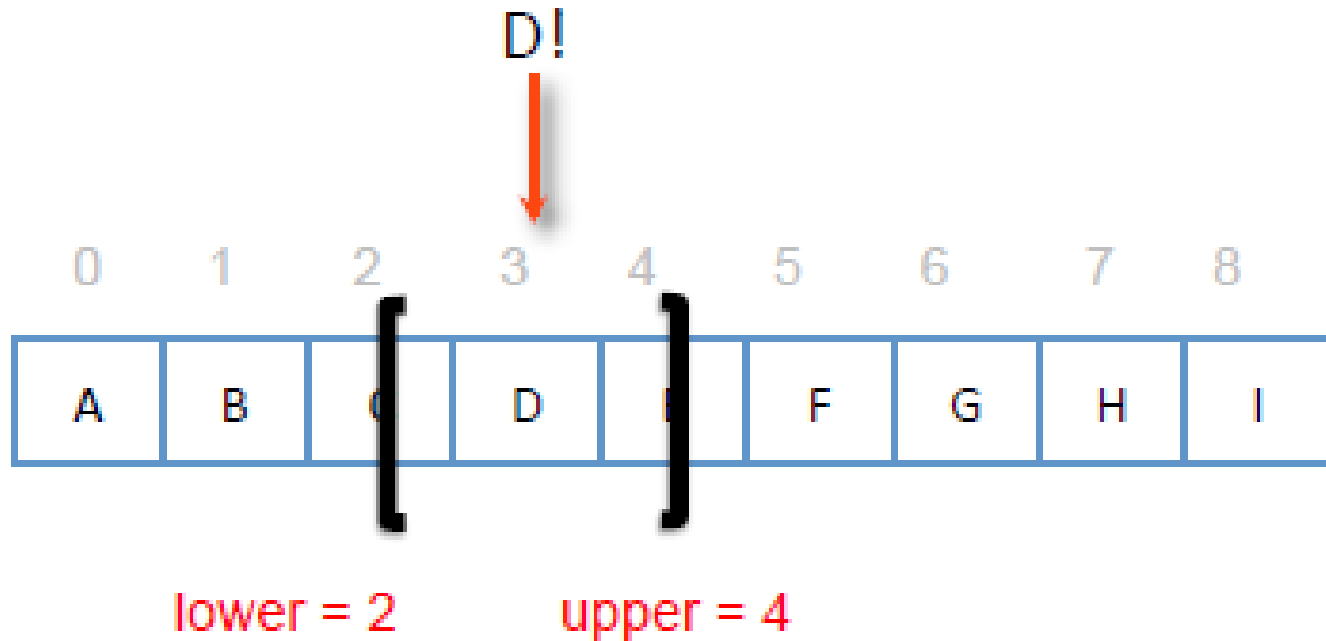


Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

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

Midpoint $\rightarrow 5 // 2 \rightarrow 2$

Midpoint Calculation Example [4/4]



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

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

Midpoint $\rightarrow 6 // 2 \rightarrow 3$

Coding Binary Search

- `items = [1, 3, 6, 9, 23, 44, 66, 92]`
- `key`
- `bsearch(items, key)`
- We need 2 pointers : `lower` and `upper`
- `bs_helper(items, key, lower, upper)`

Recursive Binary Search in Python

```
# main function
def bsearch(items, key):
    return bs_helper(items, key, -1, len(items))

# recursive helper function
def bs_helper(items, key, lower, upper):
    if lower + 1 == upper: # Base case: empty
        return None
    mid = (lower + upper) // 2 # Recursive case
    if key == items[mid]:
        return mid
    if key < items[mid]: # Go left
        return bs_helper(items, key, lower, mid)
    else: # Go right
        return bs_helper(items, key, mid, upper)
```

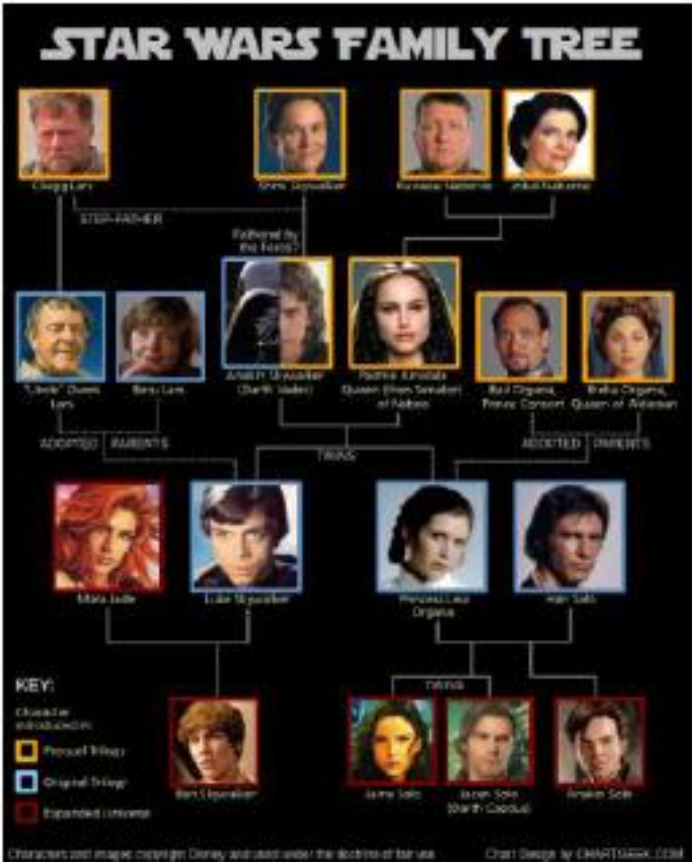
Diagram illustrating the recursive binary search function:

- first value for *lower***: Points to the `-1` argument in the `bsearch` function call.
- first value for *upper***: Points to the `len(items)` argument in the `bsearch` function call.
- same value for *lower***: Points to the `lower` argument in the recursive call `bs_helper(items, key, lower, mid)`.
- new value for *upper***: Points to the `mid` argument in the recursive call `bs_helper(items, key, lower, mid)`.
- new value for *lower***: Points to the `mid` argument in the recursive call `bs_helper(items, key, mid, upper)`.
- same value for *upper***: Points to the `upper` argument in the recursive call `bs_helper(items, key, mid, upper)`.

Search

- Binary Search
- Binary Tree
- Binary Search Tree

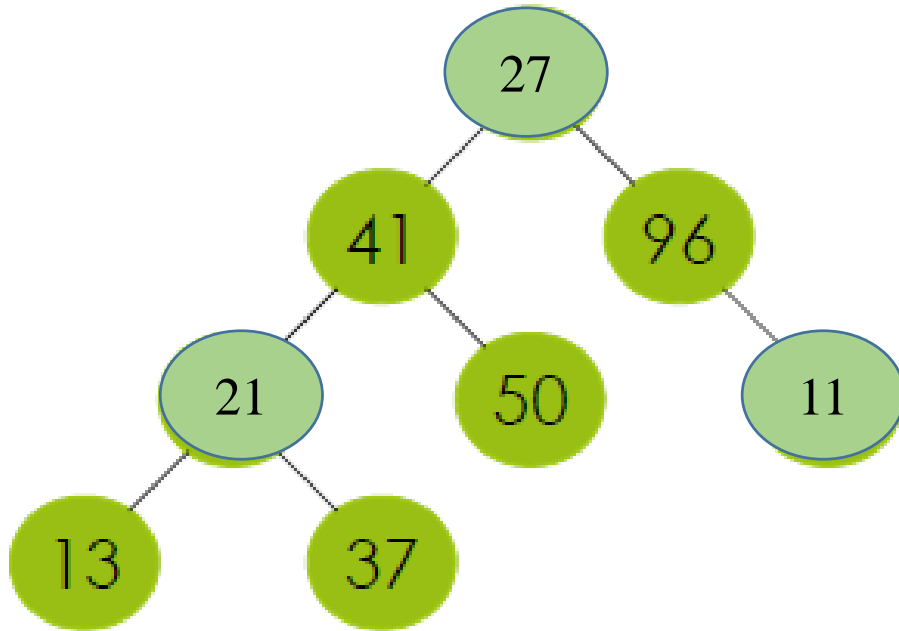
Hierarchical Data



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: File Storage, Game Trees, Family Trees, Data Compression, and many many more.....

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**?

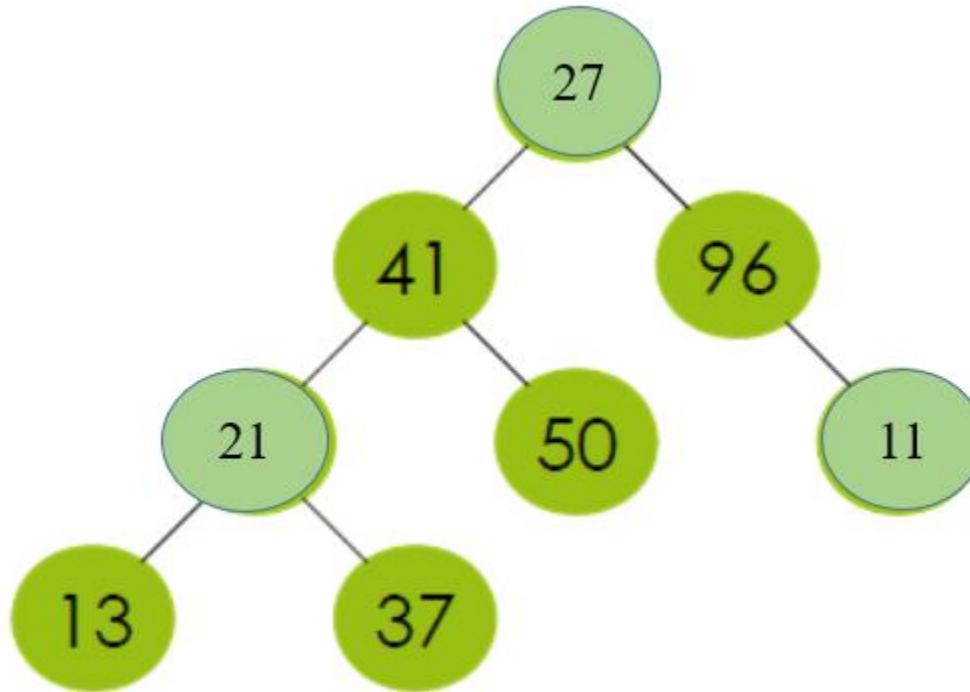
Which ones are the **leaves (external nodes)**?

Which ones are **internal nodes**?

What is the **height** of this tree?

학번	이름	학과
27	이순신	해양학과
41	임꺽정	산림과
96	사오정	동물학과
..
...

Binary Tree [2/4]

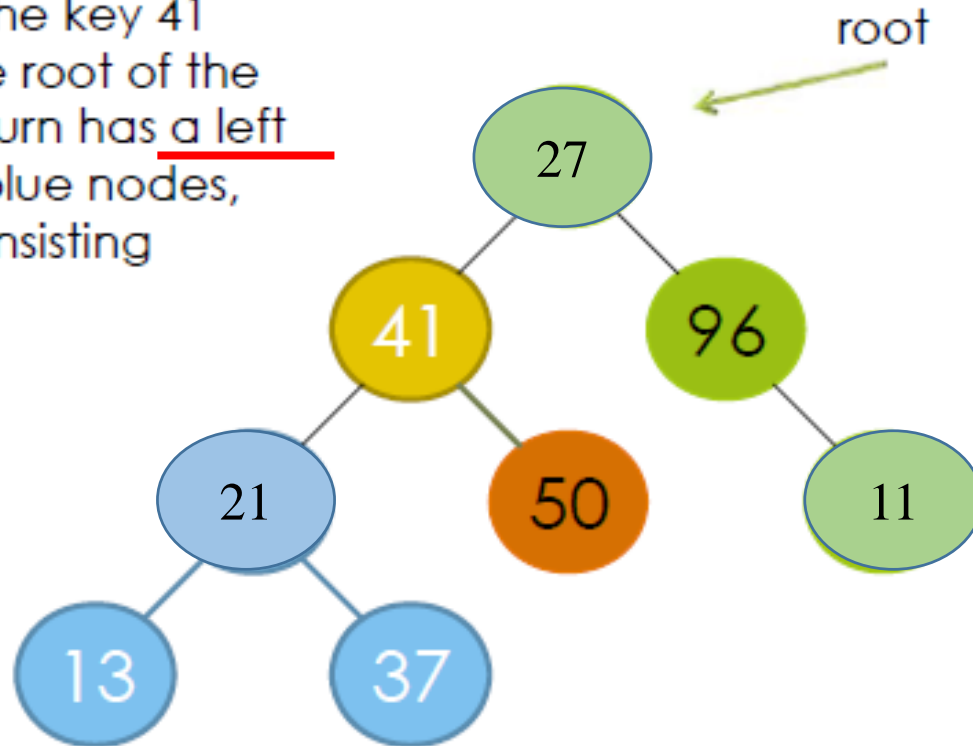


- The root contains the data value 84
- There are 4 leaves in this binary tree: nodes containing 13, 37, 50, 98
- There are 3 internal nodes in this binary tree: nodes containing 21, 41, 96
- This binary tree has height 3: considering root is a level 0, the maximum level among all nodes is 3

Binary Tree [3/4]

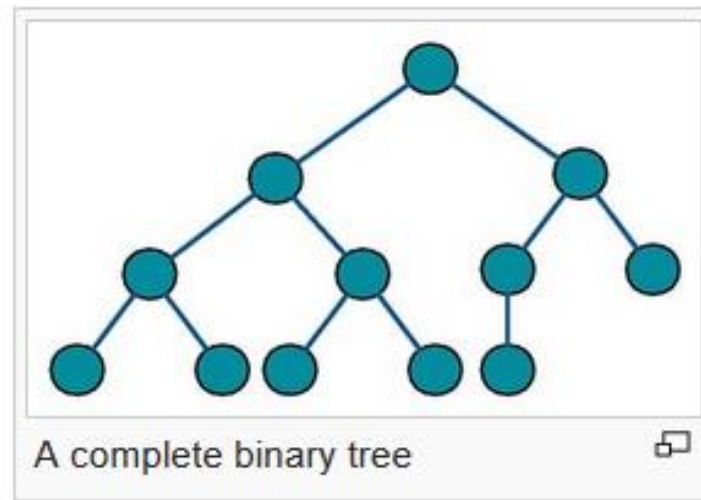
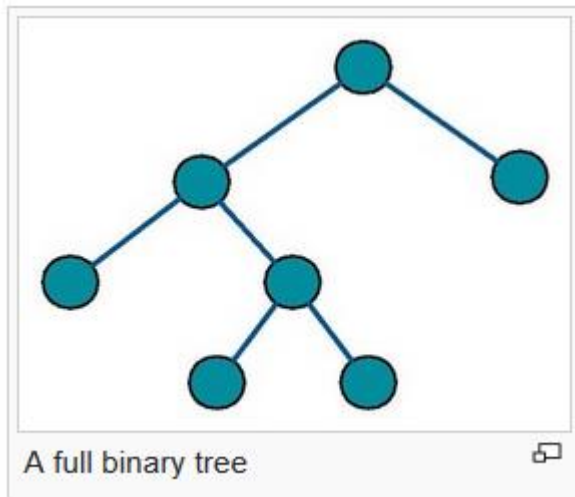
Note the **recursive** structure:

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.



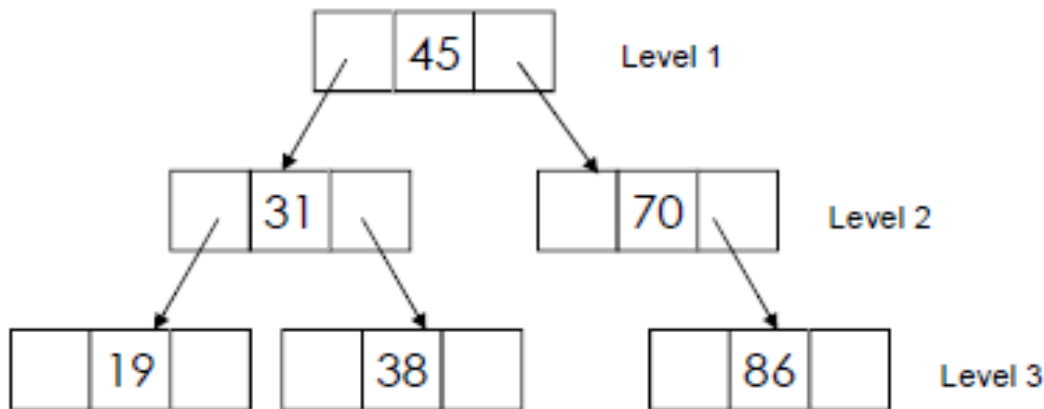
Binary Tree [4/4]

- 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 .



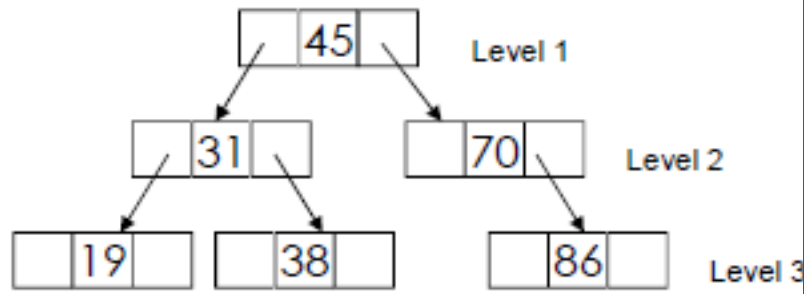
Binary Tree Implementation

- 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 Nested Lists

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



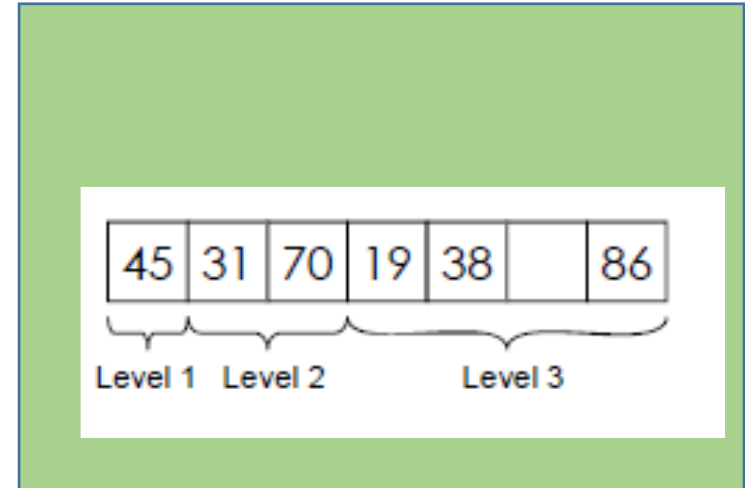
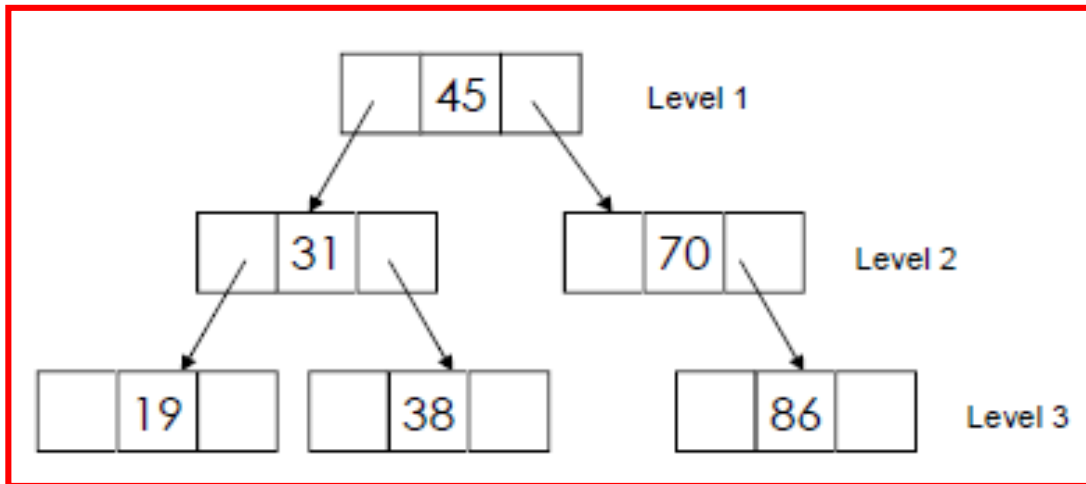
```
[45, left, right]
      |         |
      v         v
[45, [31, left, right], [70, left, right]]
      |         |
      v         v
[45, [31, [19, [], []], [38, [], []]], [70, [], [86, [], []]]]
      |
      v
[]
```

[] stands for an empty tree

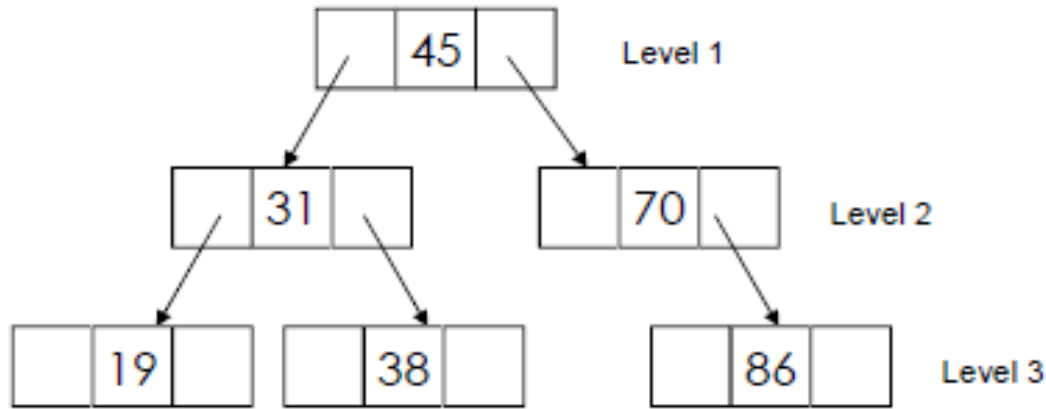
Arrows point to subtrees

Binary Tree Implementation using 1D List

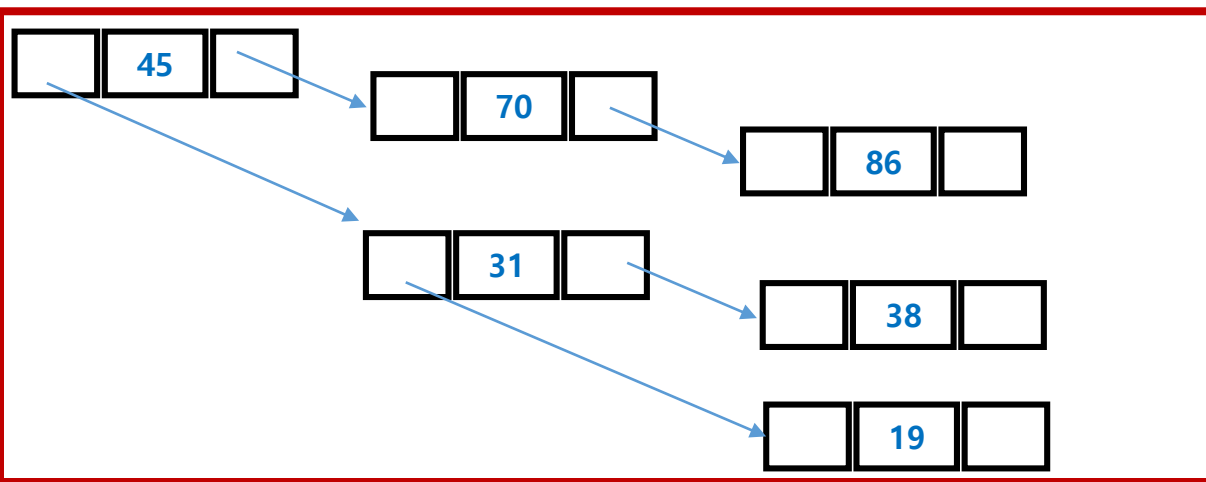
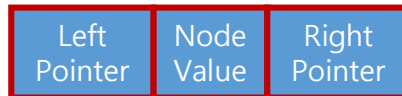
- We could also use a flat (one-dimensional list).



Binary Tree Implementation using Python OOP



User-defined Type (class) → BinaryTreeNode



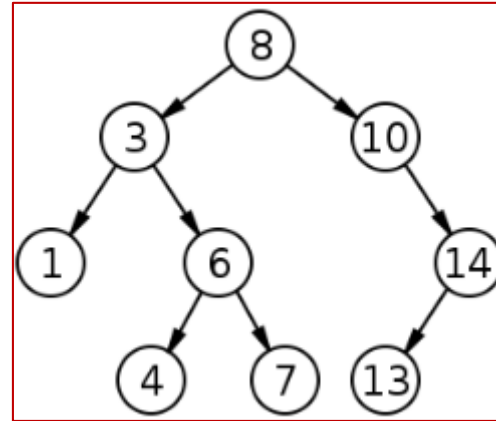
Operations for Dynamic Data Set like Tree

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

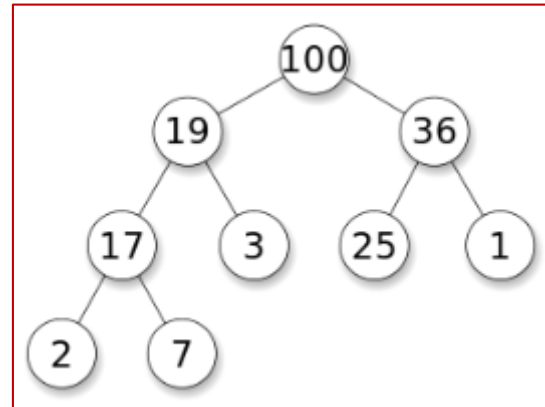
Choosing a specific data structure has consequences on which operations can be performed faster.

Applications of Binary Tree

- **Binary Search Tree**: 모든 Node에서 Left Subtree의 모든 값이 Node 보다 작고 Right Subtree의 모든 값이 Node보다 큰 상태의 Binary Tree



- **Heap** : 모든 Node가 Child Node보다 큰값을 가지는 Binary Tree



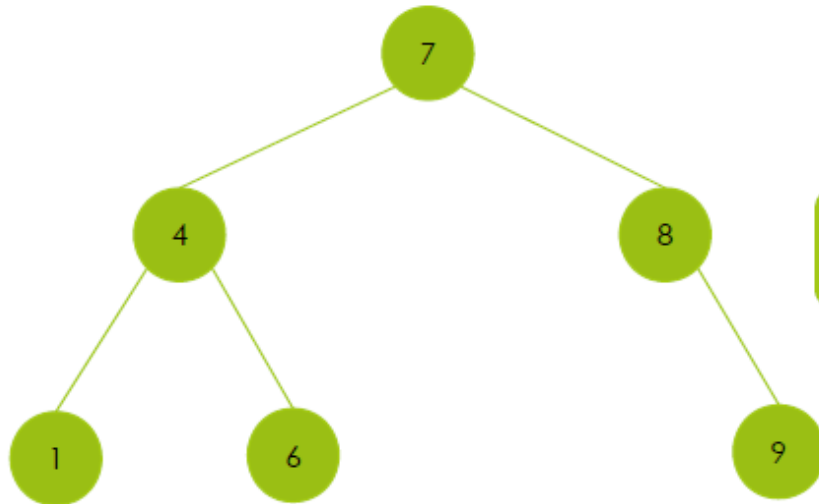
- Binary Tries
- Huffman Code Tree
- And so on.....

Search

- Binary Search
- Binary Tree
- Binary Search Tree

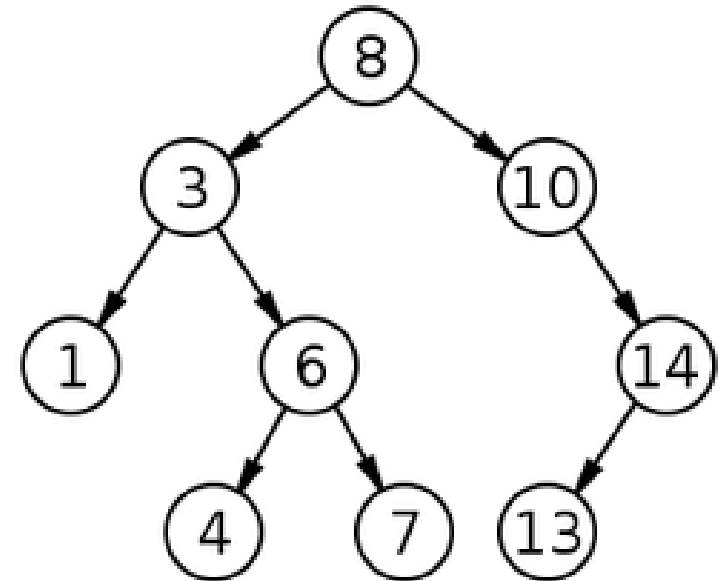
Binary Search Tree

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 right subtree are strictly greater than k (assume that there are no duplicates in the tree)



Binary tree

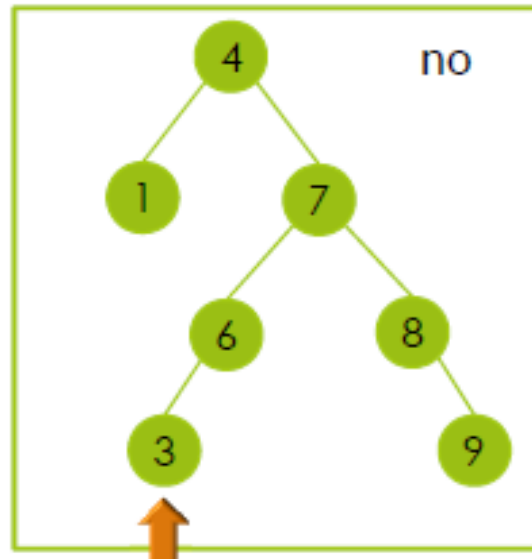
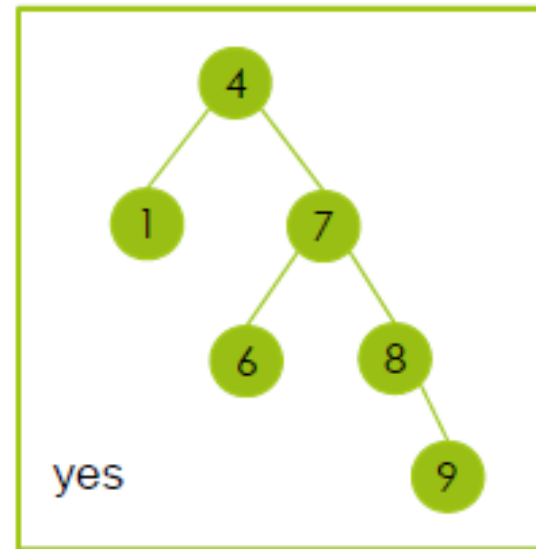
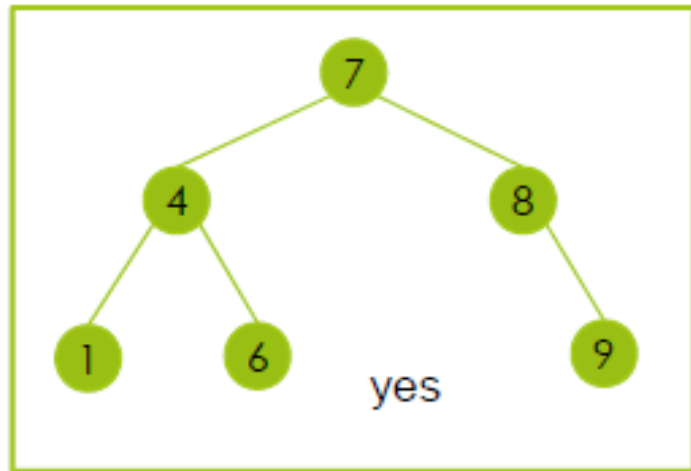
Satisfies the ordering invariant



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



Test: Is this a BST?

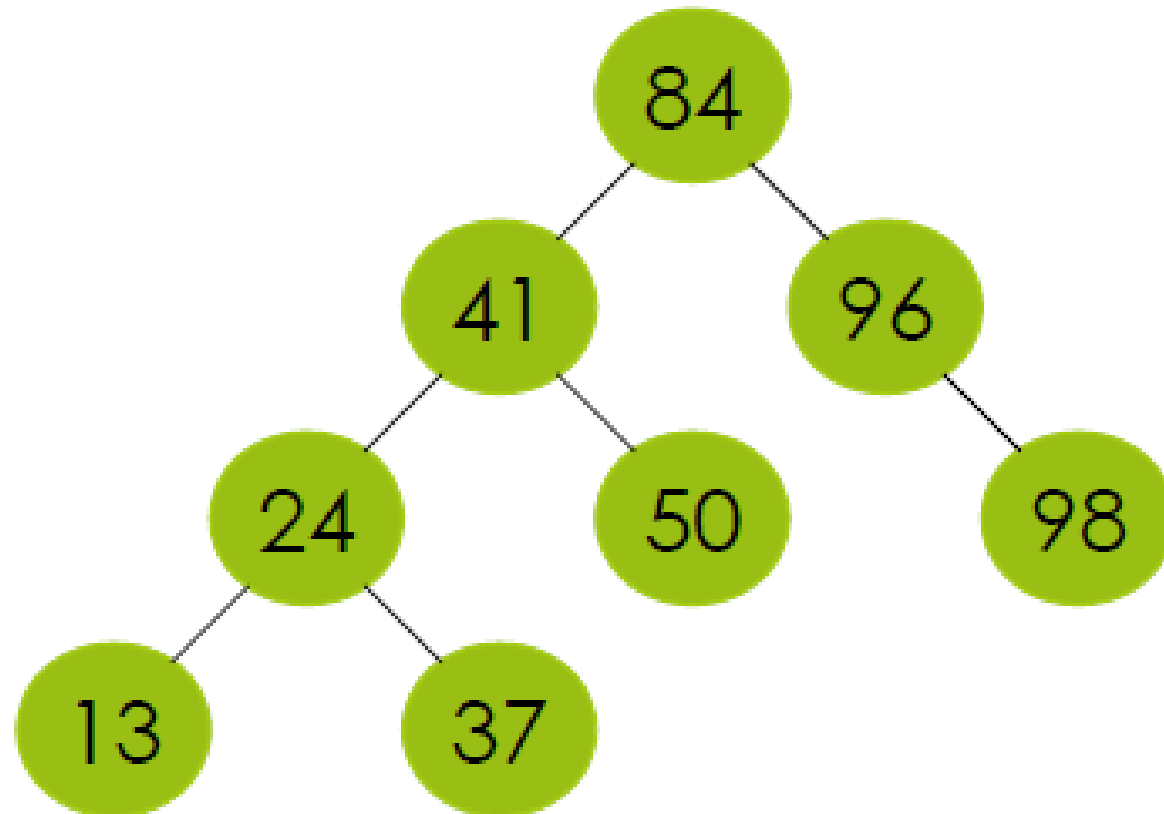


Inserting into a BST

- 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.

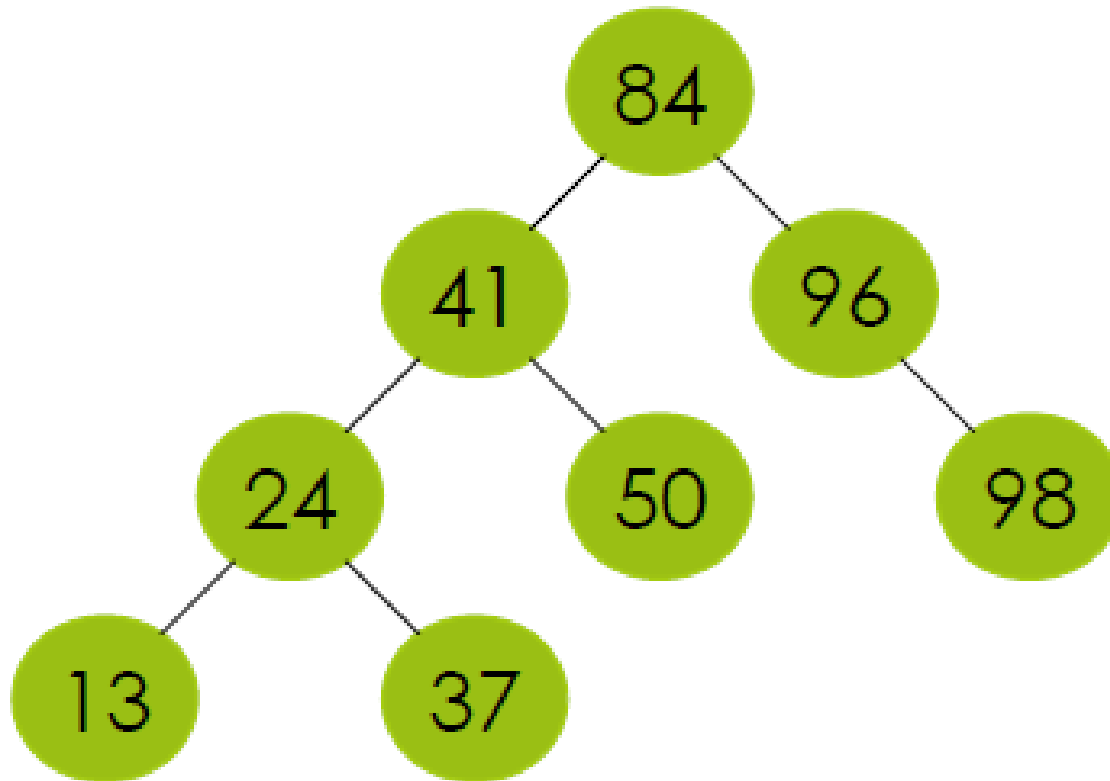
Building Binary Search Tree

■ 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?

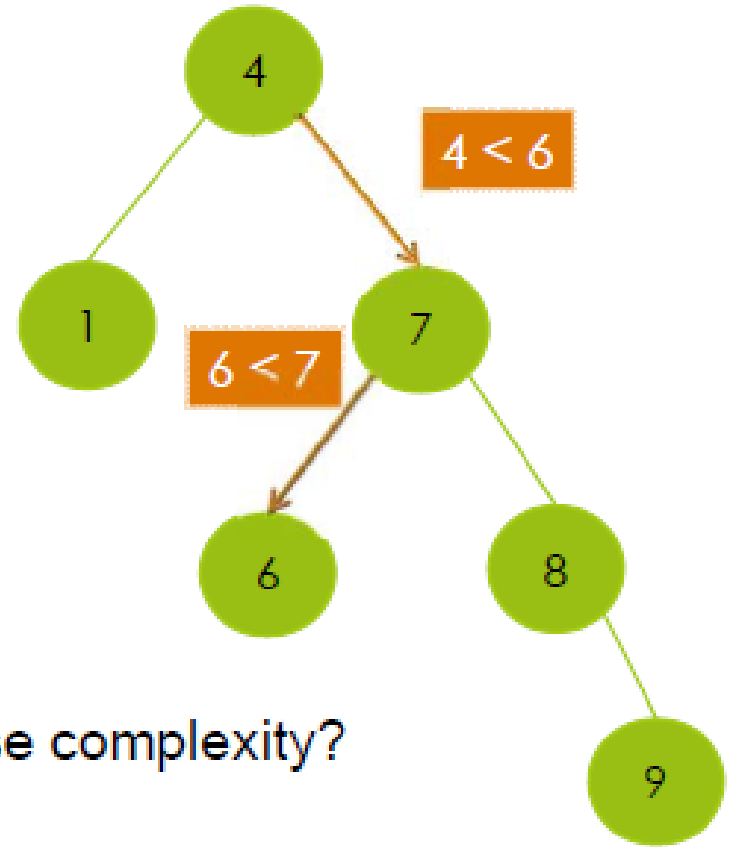


Search in Binary Search Tree [2/3]

- 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.

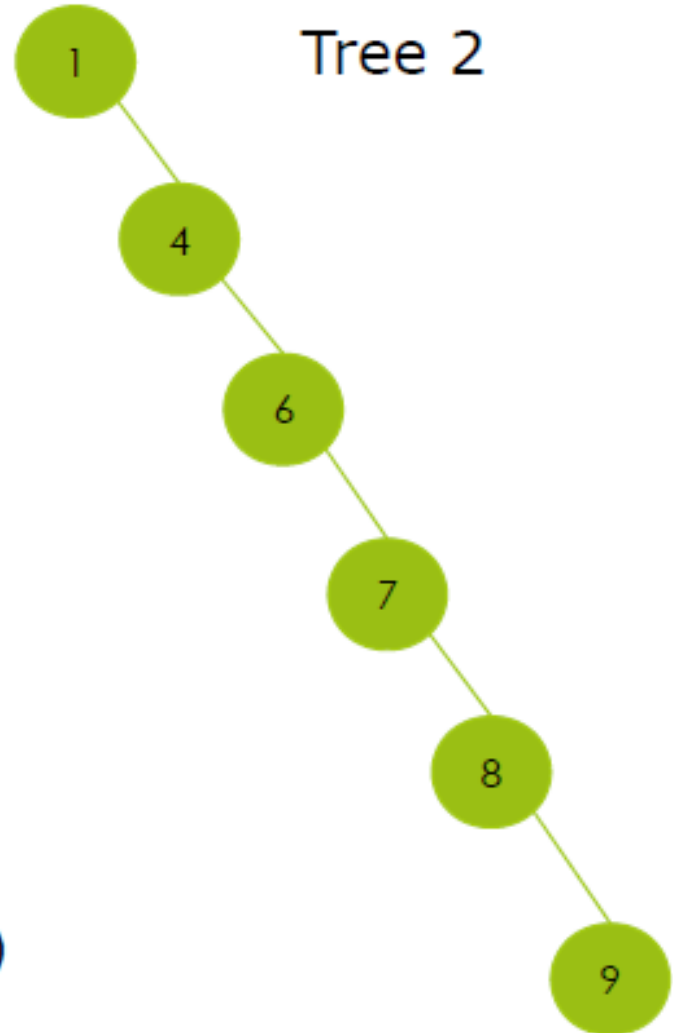
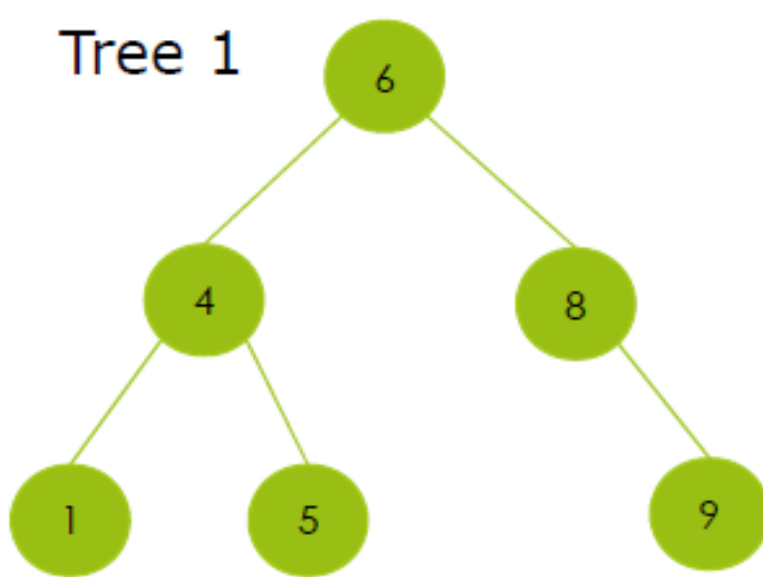
Search in Binary Search Tree [3/3]

Example: searching for 6



Can we form a conjecture about worst case complexity?

Time Complexity of Search in Tree [1/3]

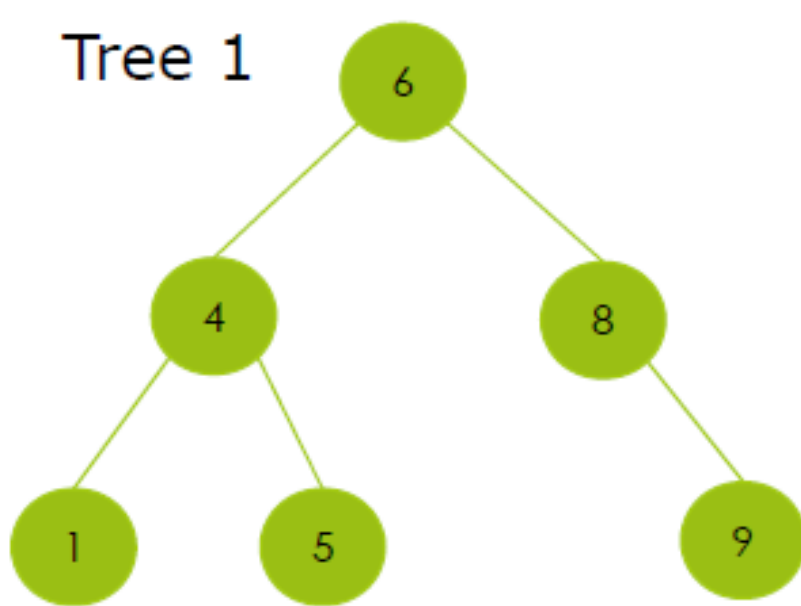


Number of nodes: n


Worst case: $O(\text{height})$

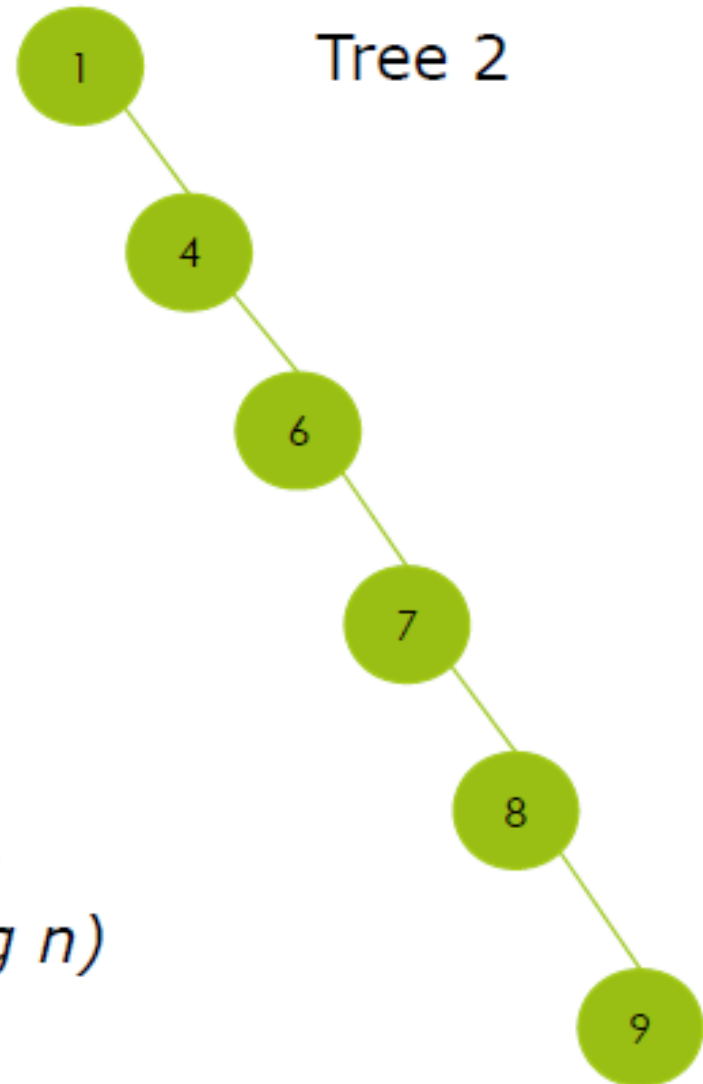
Worst height: $n \longrightarrow O(n)$

Time Complexity of Search in Tree [2/3]



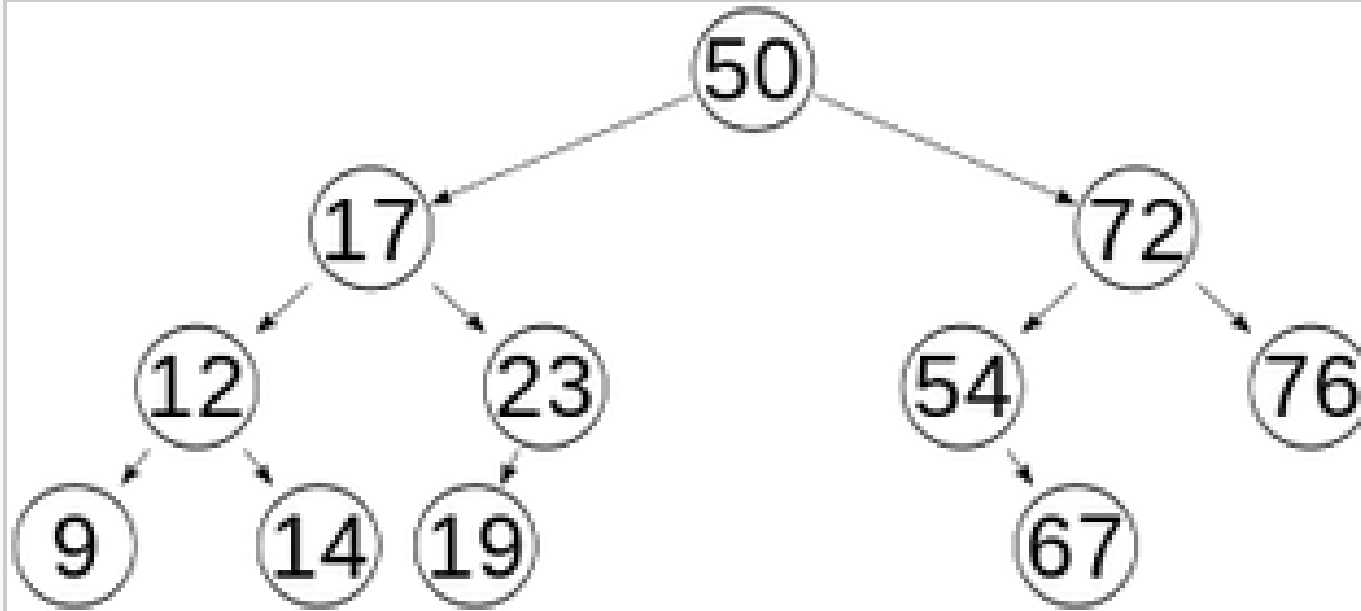
Number of nodes: n

What if we could always have
balanced trees?  $O(\log n)$



Time Complexity of Search in Tree [3/3]

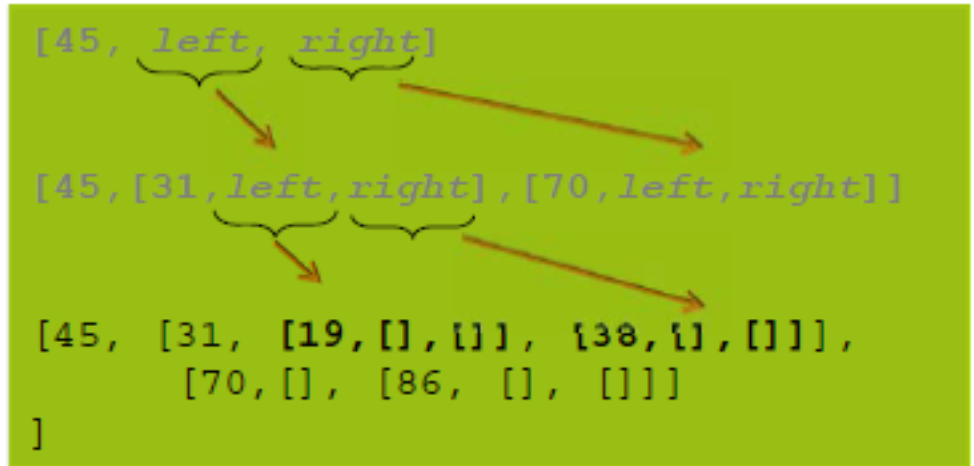
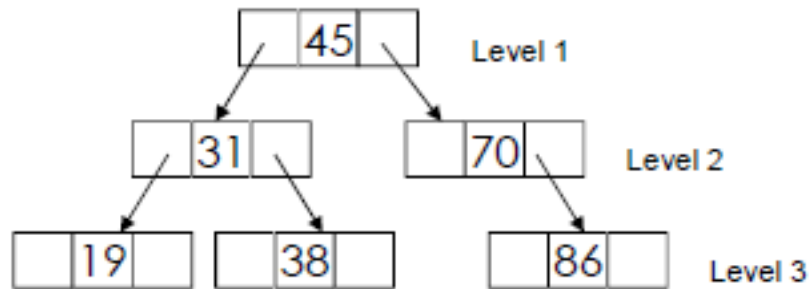
- **AVL tree** (Georgy **A**delson-**V**elsky and Evgenii **L**andis)
- A self-balancing binary search tree
- 모든 Node의 left subtree와 right subtree의 height difference가 less than one!



Example AVL tree



Binary Search Tree Implementation using Nested List



[] 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