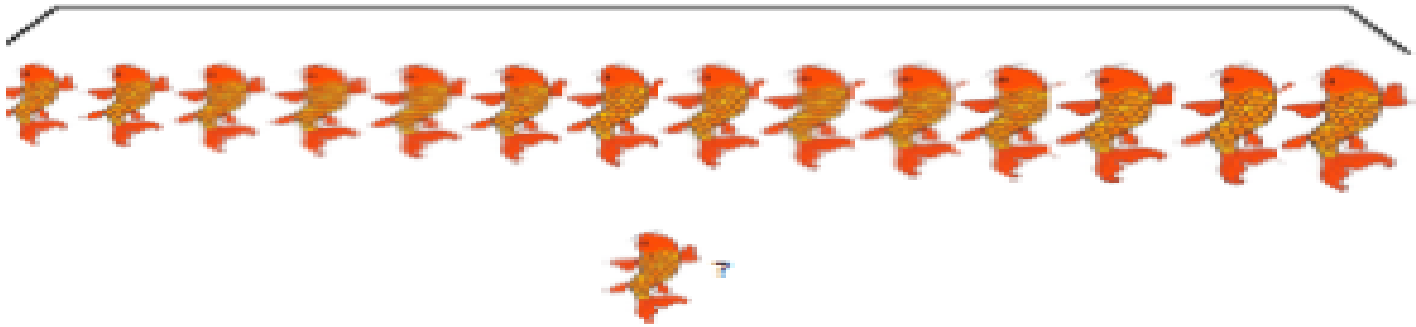
Binary Search and Binary Tree

BINARY SEARCH

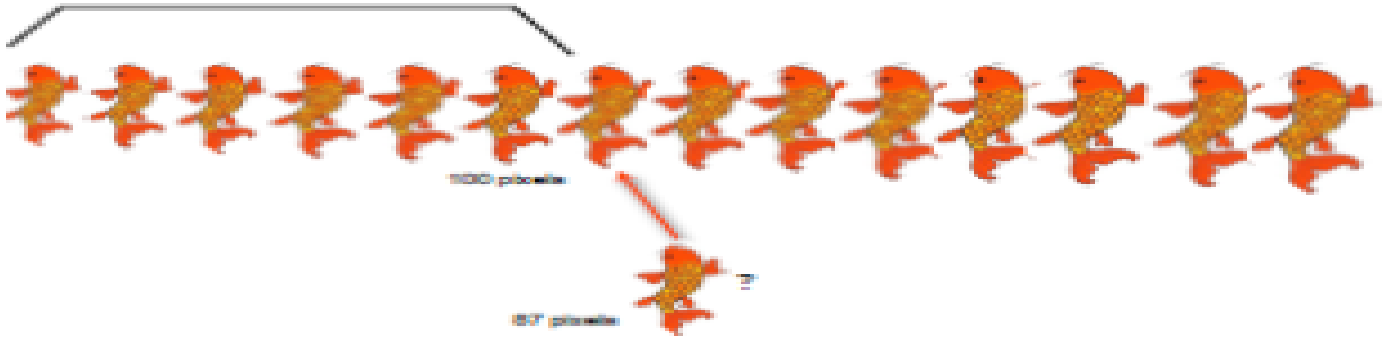
Number guessing

- 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



Binary Search in an Ordered List



Binary Search in an Ordered List

Binary Search in an Ordered List

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

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

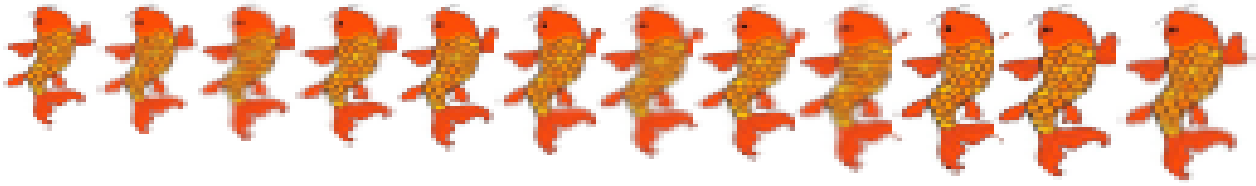
Not found: return None

Controlling the range of the search

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

lower = -1 *upper* = 12

0 11

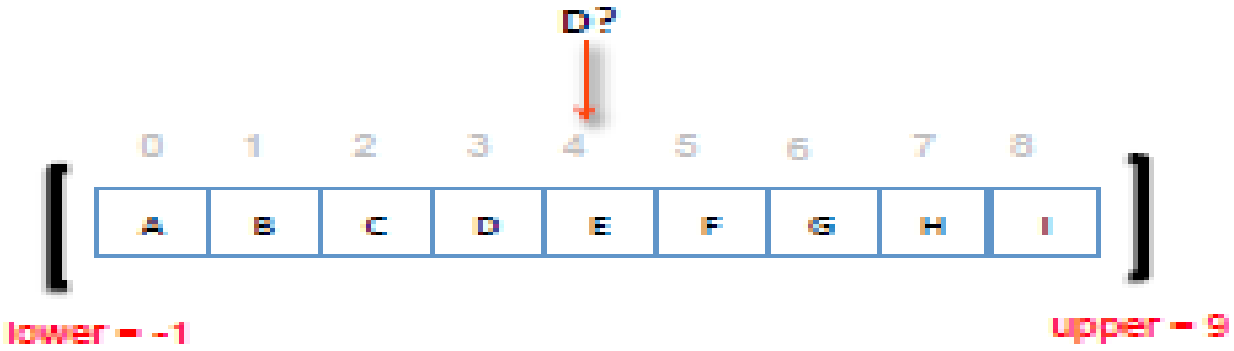


Controlling the range of the search

- *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

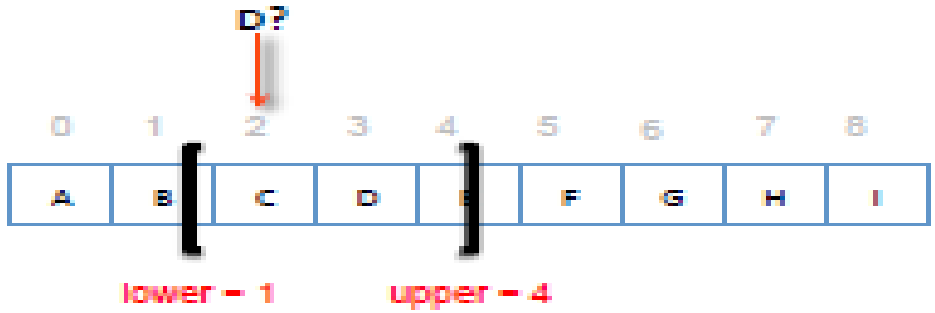


List sorted in ascending order.
Suppose we are searching for D.

Example

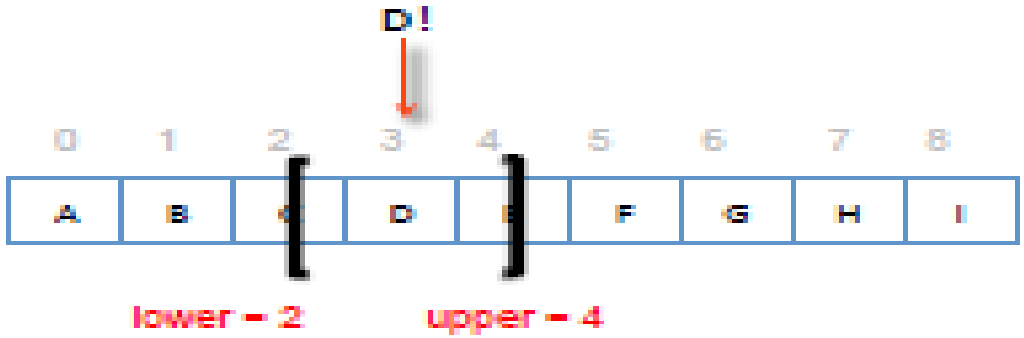
Each time we look at a smaller portion of the list within the window and ignore all the elements outside of the window

Example



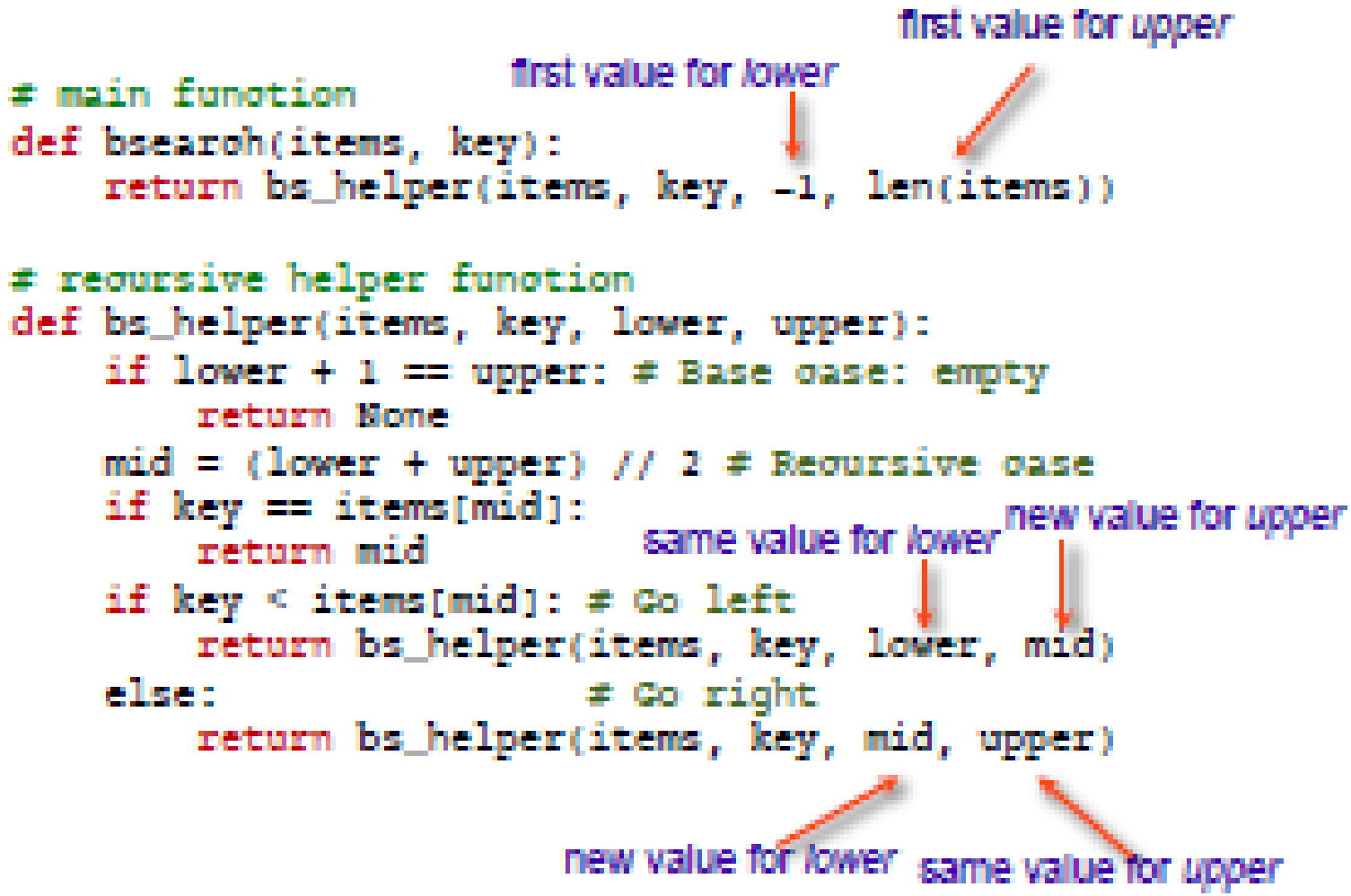
Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

Example



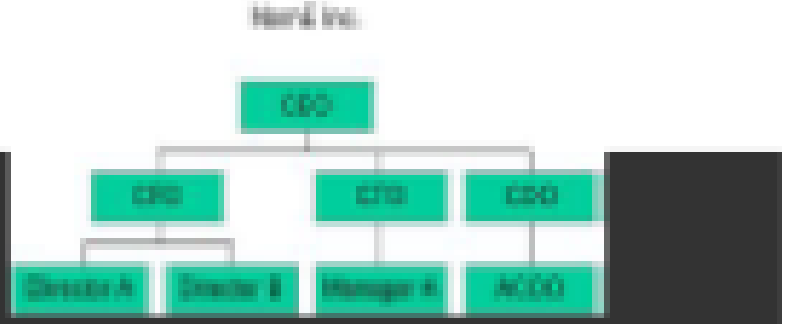
Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

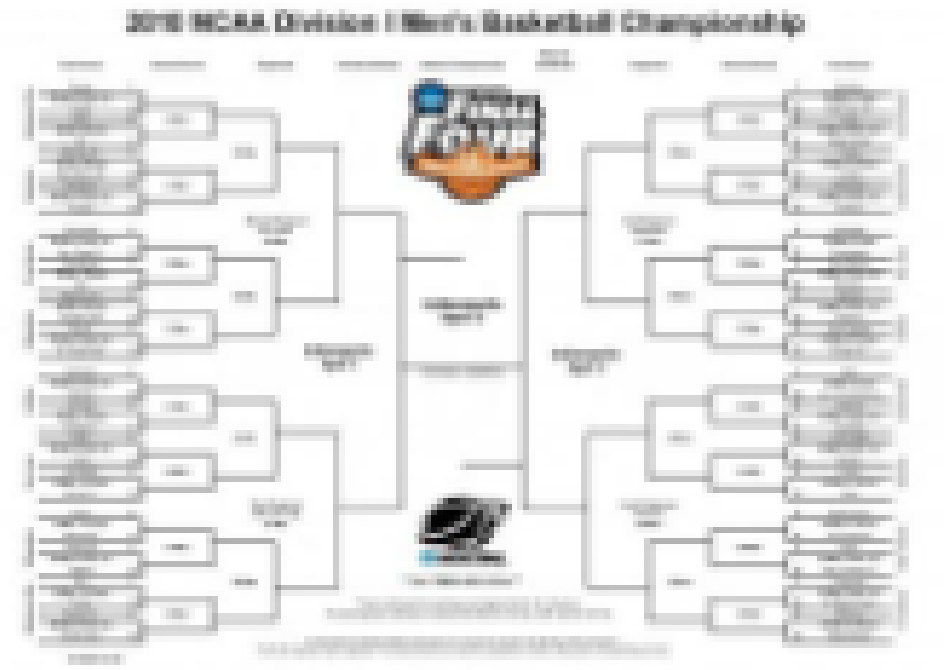
Recursive Binary Search in Python

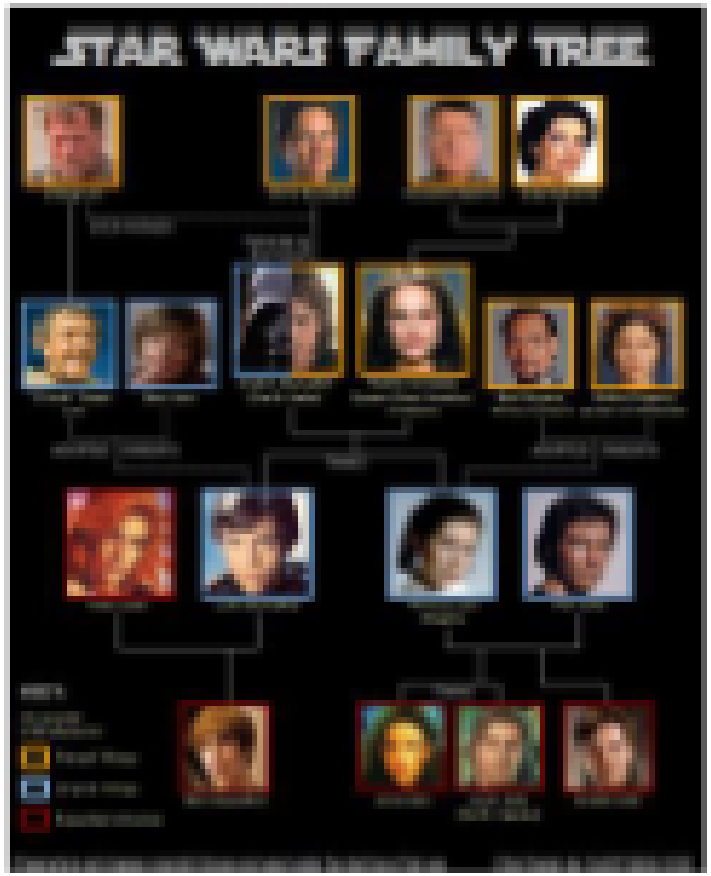


Tree Data Structure

Hierarchical Data



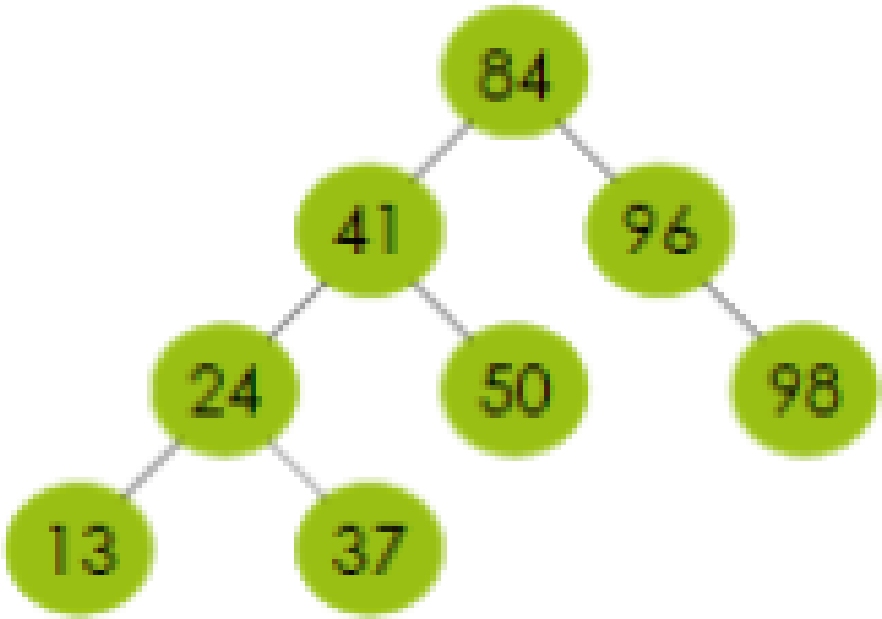




Trees

- 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

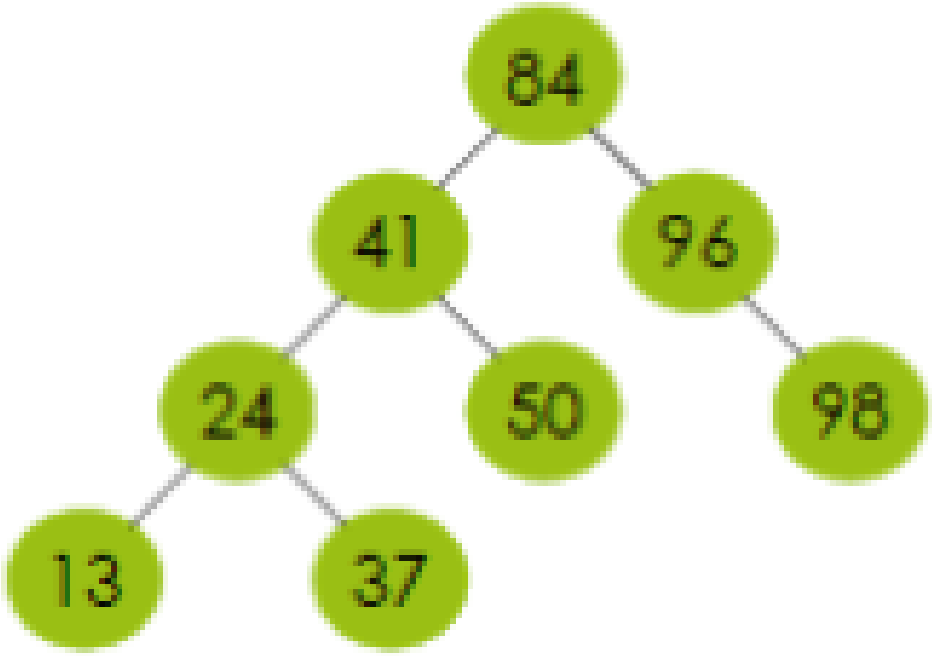
Binary Tree



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?

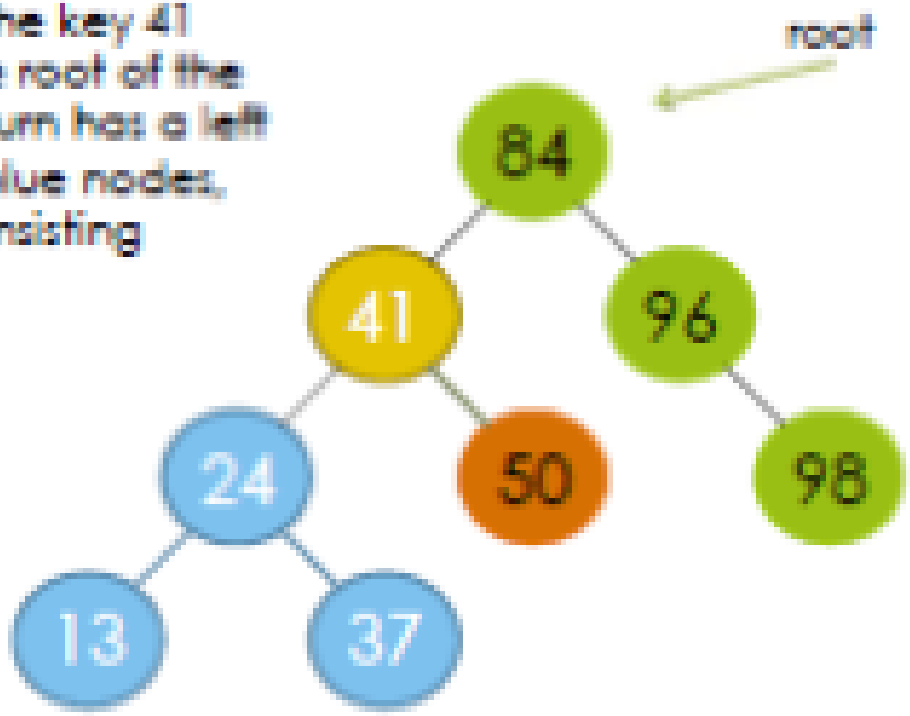
Binary Tree



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

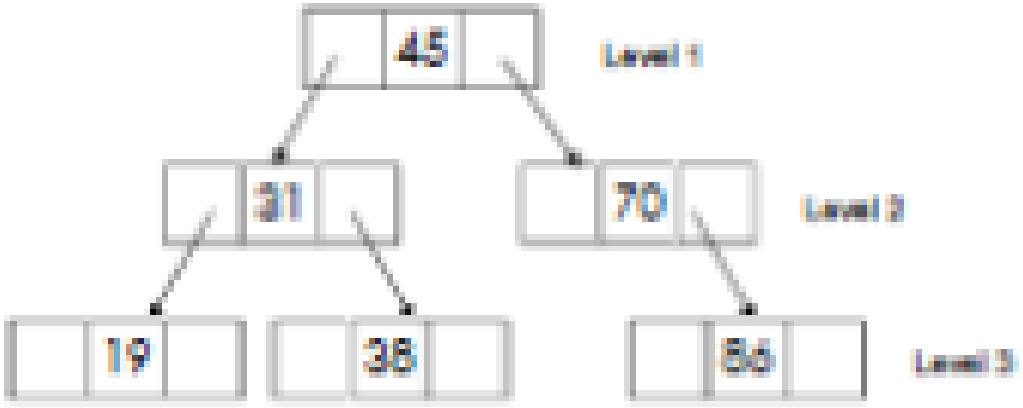
Binary Tree

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 Trees: 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.



Using Nested Lists

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

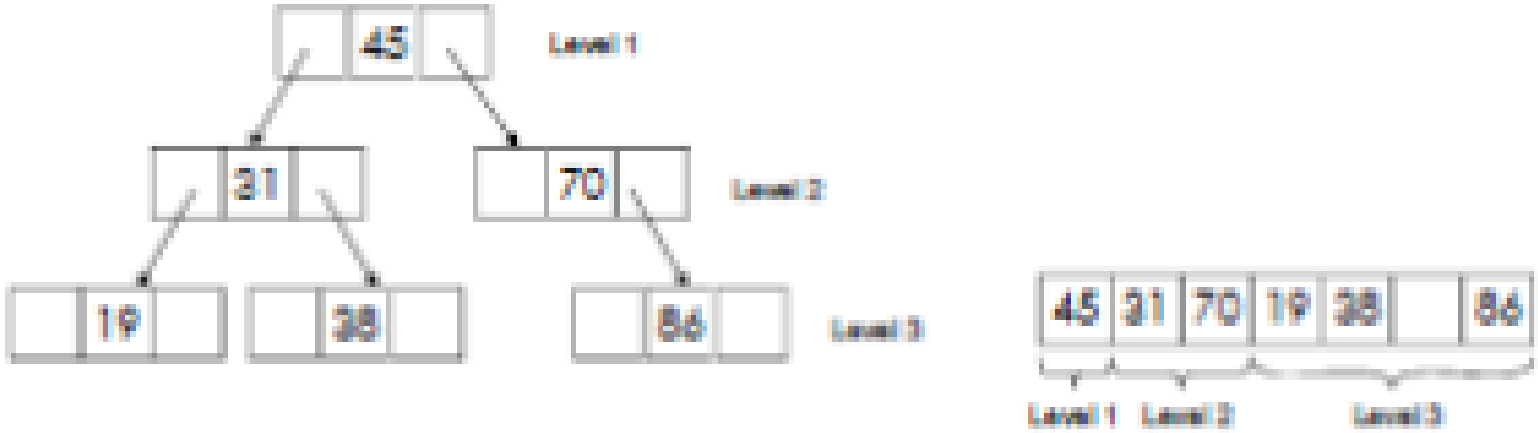

```
[45, left, right]

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

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

[ ] stands for an empty tree

Arrows point to subtrees

Using One Dimensional Lists

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



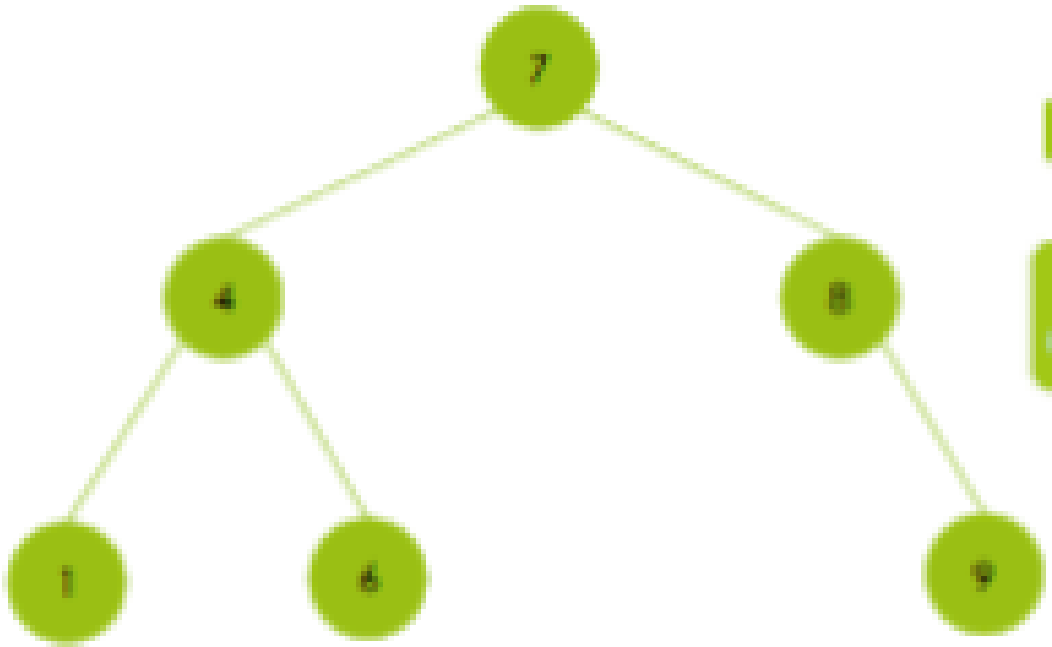
Dynamic Date Set Operations

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

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

Example: Binary Search Tree

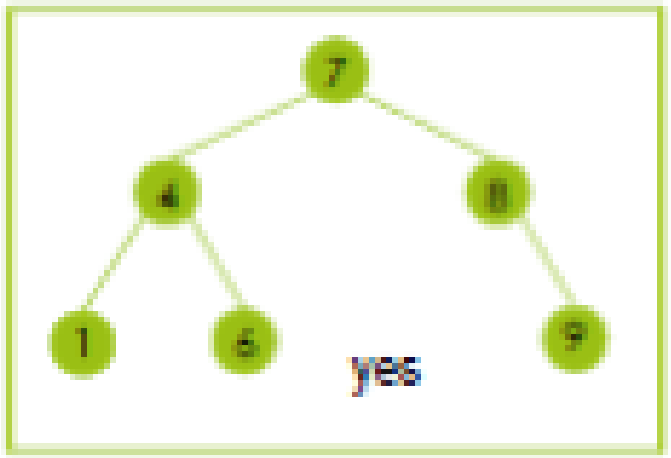
BST ordering invariant: At any node with key *k*, all keys of elements in the left subtree are strictly less than *k* and all keys of elements in the rightsubtree are strictly greater than *k* (assume that there are no duplicates in the tree)

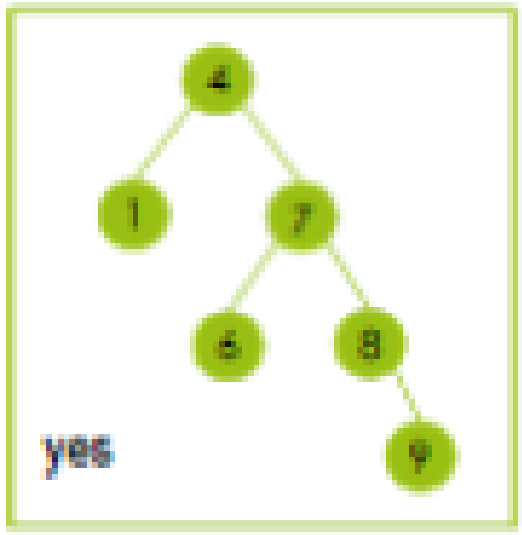


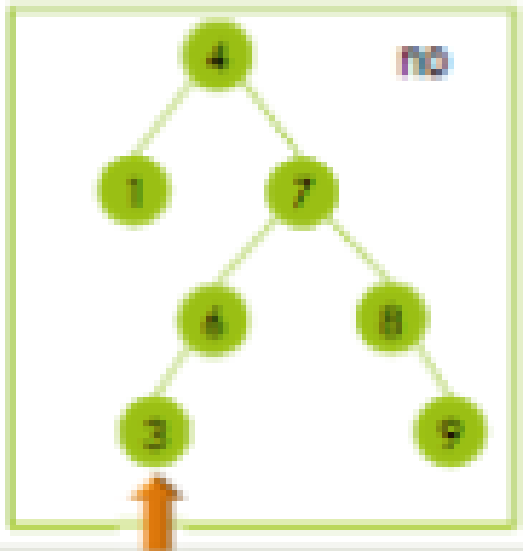
Binary tree

Satisfies the ordering invariant

Test: Is this a BST?





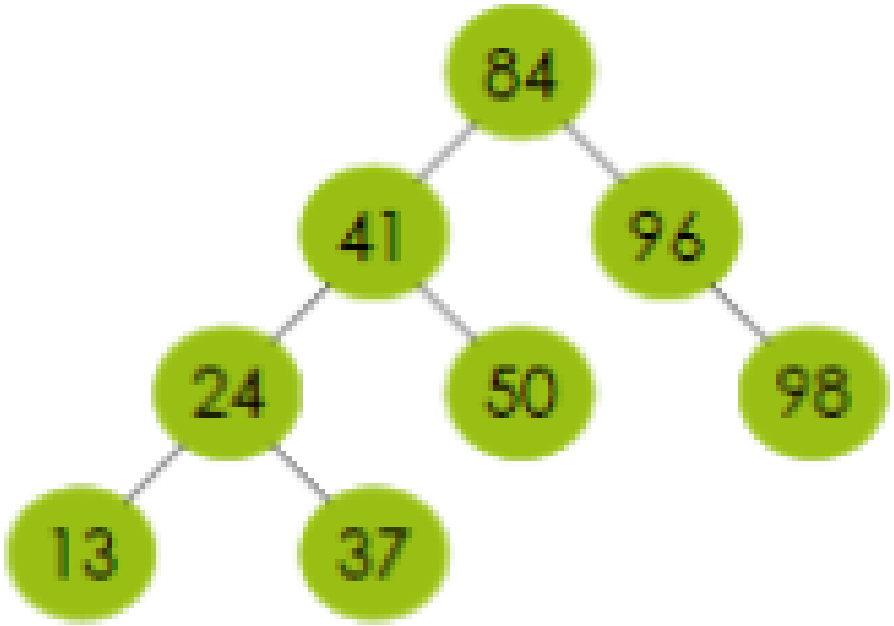


Inserting into a BST

- 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.

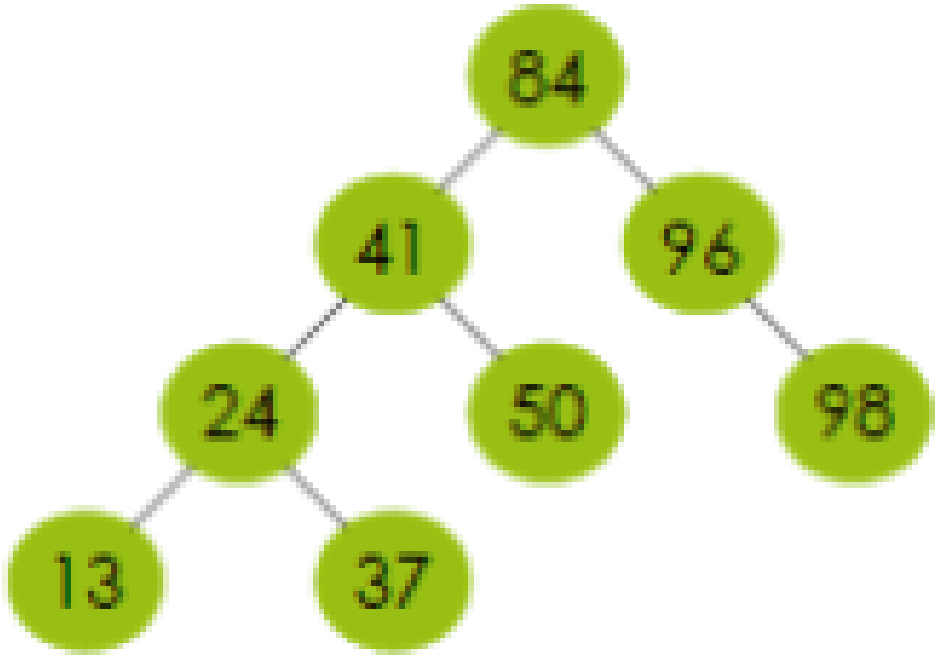
Example

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



Using a BST

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

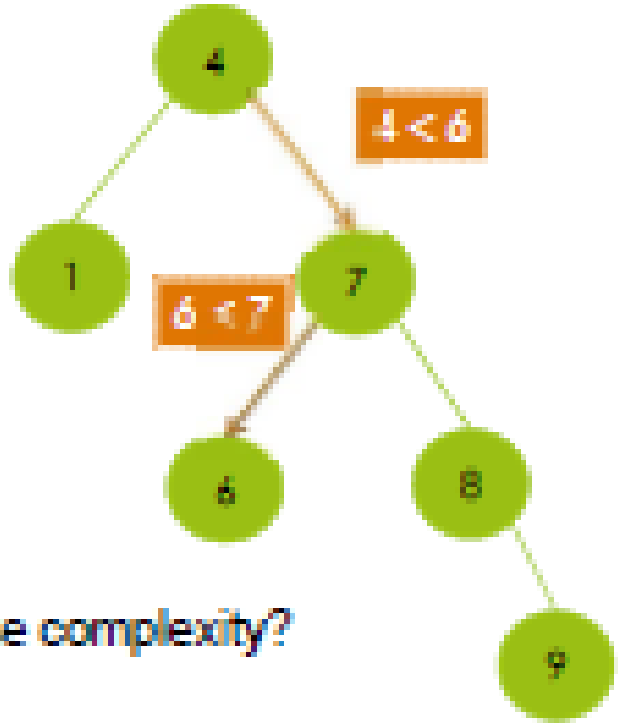


Searching a BST

- 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.

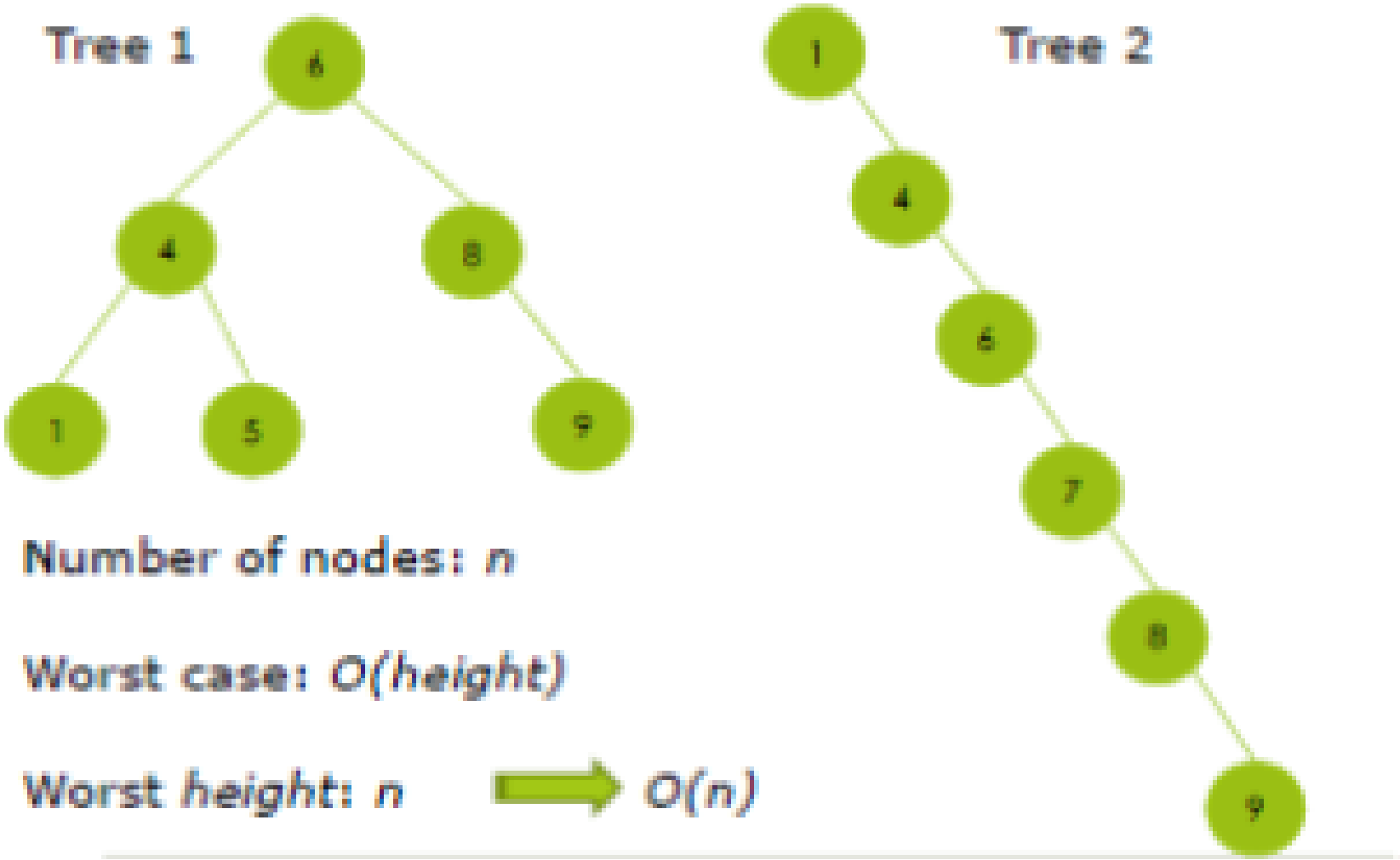
Searching the tree

Example: searching for 6



Can we form a conjecture about worst case complexity?

Time complexity of search

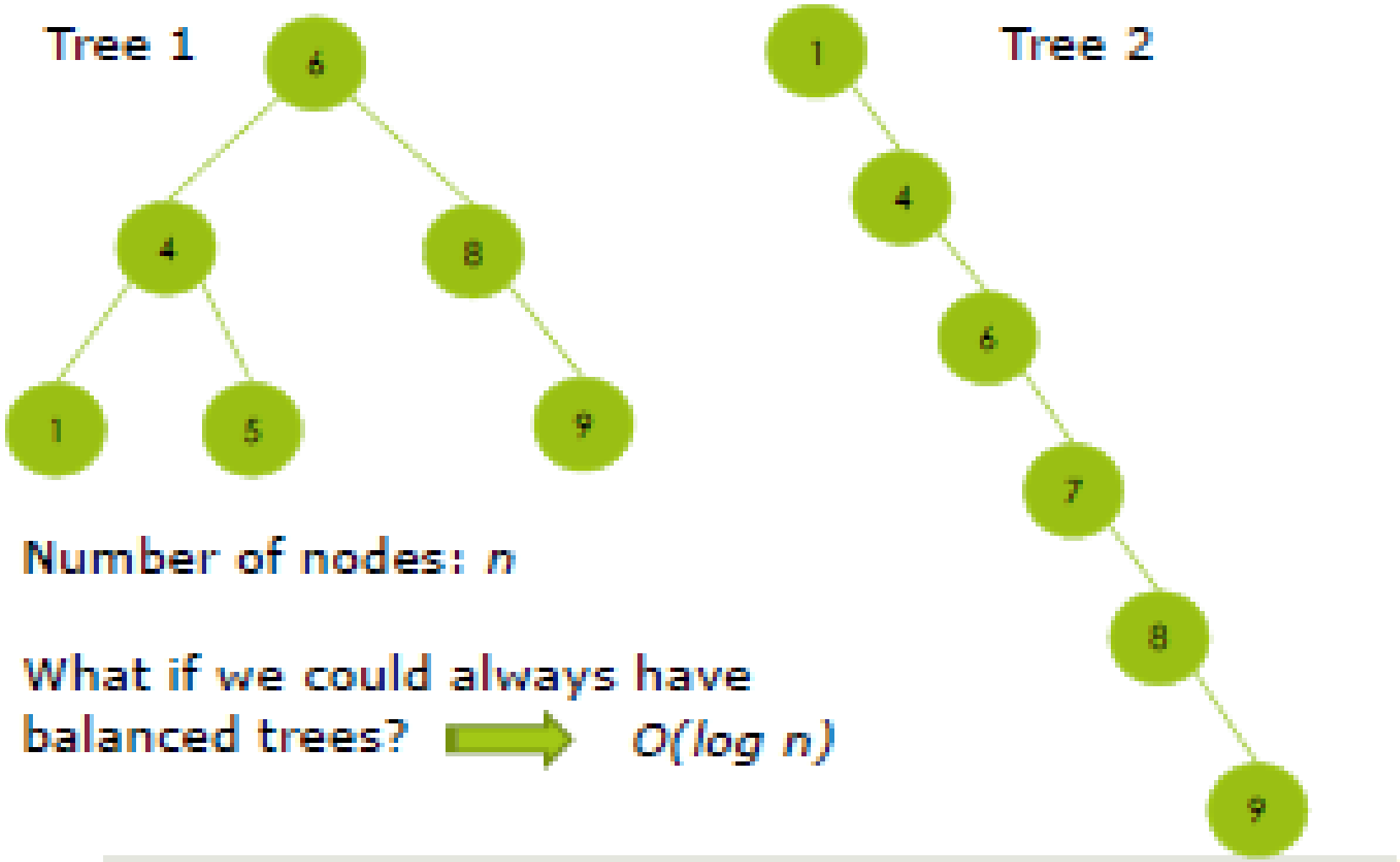


Number of nodes: n

Worst case: $O(height)$

Worst *height*: n ⟹ $O(n)$

Time complexity of search



Number of nodes: *n*

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

Team Project 4: 1번

A *dictionary* is an unordered mutable collection of key/value pairs. It associates a key (for example a word) with a value (for example a definition for that word). Any particular key can appear at most once in a dictionary and keys must be immutable. While the keys in a dictionary are guaranteed to be unique, the associated values are not. In the file `histogram.py`, define a function `histogram(d)` that takes a dictionary `d` as input and returns another dictionary representing a histogram of the original dictionary's values. Specifically, the keys of the new dictionary are the unique values of the input dictionary, and the values of the new dictionary are counts of the number of times each value appears in the original dictionary.

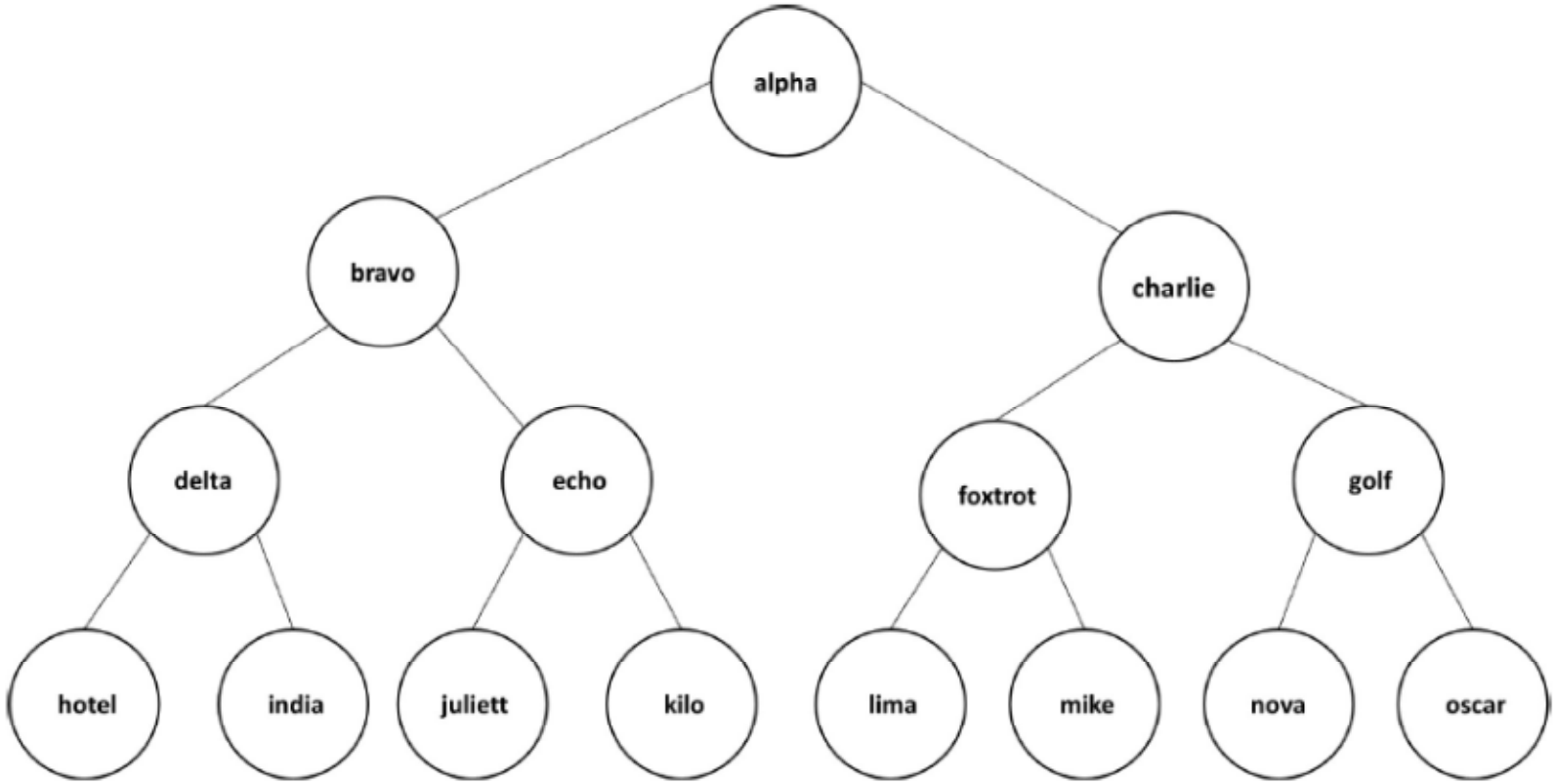
Hint: For any dictionary `d`, `list(d.values())` returns a list consisting of all the values in `d`. You may also find it convenient to use the `count` method for lists: `lst.count(x)` returns the number of occurrences of `x` in lst.

Example usage:

```
>>> letters = {1: "a", 2: "b", 3:"a"}
>>> histogram(letters)
{'b': 1, 'a': 2}
>>> letters = {1: "a", 2: "b", 3:"c"}
>>> histogram(letters)
{'b': 1, 'a': 1, 'c': 1}
>>> letters[4] = "a"
>>> letters[5] = "b"
>>> letters[6] = "a"
>>> histogram(letters)
{'b': 2, 'a': 3, 'c': 1}
>>>
```

Team Project 4: 2번

Recall that, one way to represent the nodes of a **full** binary tree is with a list, where the first element contains the root, the next two elements contain the next level of the tree (the children of the root), the next four elements contain the next level of the tree (two containing the children of the root's left child and two containing the children of the root's right child), the next eight elements contain the next level (the two children of each of the nodes at the previous level), and so on, depending on how many levels the tree has.



The binary tree above with nodes labeled with strings "alpha", "bravo", ... would be represented by the Python list:

```
["alpha", "bravo", "charlie", "delta", "echo", "foxtrot", "golf", "hotel", "india", "juliet", "kilo", "lima", "mike", "nova", "oscar"]
    0         1        2          3        4        5          6        7        8         9        10      11      12      13      14
```

Let us look at the indices of left children:

- The node at 0 has its left child at 1 (alpha's left child is bravo)
- The node at 1 has its left child at 3 (bravo's left child is delta)
- The node at 2 has its left child at 5 (charlie's left child is foxtrot)
- The node at 3 has its left child at 7 (delta's left child is hotel)
- The node at 4 has its left child at 9 (echo's left child is juliet)

Do you see a pattern? There are simple formulas that can be used to calculate the indices of a node's left child, right child, and parent from that node's index.

a. Define a function `left_index(i)` (in `left_index.py`) that, when passed the index `i` of a node, will return the index of that node's left child. (You do not need to worry about whether the node has a left child; when the node does not actually have a left child, the function should still return where its left child would be if it had one. You may assume that `i` is a non-negative integer.)
b. Define a function `right_index(i)` (in `right_index.py`) that, when passed the index of a node, will return the index of that node's right child. (You do not need to worry about whether the node has a right child; when the node does not actually have a right child, the function should still return where its right child would be if it had one. You may assume that `i` is a non-negative integer.)
c. Define a function `parent_index(i)` (in `parent_index.py`) that, when passed the index of a node, will return the index of that node's parent. (You may assume that `i` is a positive integer.) Hint: Think about how this part relates to the previous parts.
d. Define a function `is_leaf(tree, i)` (in `leaf.py`) that, when passed the list representation `tree` of a full binary tree and the index of a node `i` , will return True if the node has no children and False otherwise (a node with no children is called a leaf). You may assume that `i` is a non-negative index less than the length of `tree`. You may want to use some of the functions you defined in the previous parts.
   Example usage:

```
>>> a = ["alpha", "bravo", "charlie", "delta", "echo", "foxtrot", "golf", "hotel", "india", "juliett", "kilo", "lima", "mike", "nova", "osca
["alpha", "bravo", "charlie", "delta", "echo", "foxtrot", "golf", "hotel", "india", "juliett", "kilo", "lima", "mike", "nova", "oscar"]
>>> is_leaf(a,0)
False
>>> is_leaf(a,13)
True
```

Team Project 4: 3번

- In the file first_perfect_square.py, define a Python function first_perfect_square(numbers) that takes a list of integers (which might be empty) as a parameter. It should return the index of the first number in the list that is a perfect square. Caution: return the index, not the value in the list at that index! If no element of the list is a perfect square, return -1. We don't care what your function does if its input is not a list of integers (that is, it's allowed to crash if the input is not a list of integers.)

- Example of first_perfect_square:

```
bash-3.2$ python3 -i first_perfect_square.py
>>> first_perfect_square(list(range(5)))
0
>>> first_perfect_square([2, 4, 6, 8, 10, 12])
1
>>> first_perfect_square([6, 8, 10, 12, 9])
4
>>> first_perfect_square([1,1])
0
>>> first_perfect_square([-6, 6, -2, 2, -3, 3])
-1
>>> first_perfect_square([42])
-1
>>> first_perfect_square([])
-1
>>> first_perfect_square([12345678912345678**2])
0
```

Team Project 4: 4번

- In the file num_perfect_squares.py, define a Python function num_perfect_squares(numbers) that takes a list of integers (which might be empty) as a parameter. It should return the number of elements of the input list that are perfect squares. We don't care what your function does if its input is not a list of integers (that is, it's allowed to crash.)

- Example:

```
bash-3.2$ python3 -i num_perfect_squares.py
>>> num_perfect_squares([])
0
>>> num_perfect_squares([0])
1
>>> num_perfect_squares([0,1])
2
>>> num_perfect_squares(list(range(10)))
4
>>> num_perfect_squares([3]*10)
0
>>> num_perfect_squares([4]*10)
10
>>> num_perfect_squares([-4, -2, 0, 2, 4])
2
```