Dmitri Stanchevici
Unit 5
Module 0

Ex.0.2

This algorithm would clearly have to be quadratic, with each word's reversal having to be compared with all other elements in the structure. Thus, the time taken to complete this algorithm is proportional to n^2. Thus we are dealing with a $f(n)=n^2$ function. In Big-Oh notation (or order notation), this algorithm can be described as

$O(n^2)$

We could add two more coefficients: the time taken for reversing the characters in the searched word and for the comparison required by contains() function. However, these operations are approximate constants (being only as large as the number of characters in the longest word), so they can be dropped from the above Oh-notation.

Ex. 0.3

The output comparing the performance of various data structures for determining whether words reversals are real words:

```
# words: 15127
Using a linked-list: count=207  timeTaken=1243
Using an array-list: count=207  timeTaken=799
Using a tree: count=207  timeTaken=19
Using a hashtable: count=207  timeTaken=3
```

Ex. 0.4

Lookfor for integers:

N=10,000

```
Using an array-list: count=5015  timeTaken=68
Using a tree: count=5015  timeTaken=4
Using a hashtable: count=5015  timeTaken=2
```

N=100,000

```
Using an array-list: count=4999  timeTaken=847
```

```
Using a tree: count=4999  timeTaken=5
Using a hashtable: count=4999  timeTaken=1
```

N=1,000,000

```
Using an array-list: count=5028  timeTaken=6802
Using a tree: count=5028  timeTaken=7
Using a hashtable: count=5028  timeTaken=2
```

Ex. 0.5

For an array with n elements, the speed of the search is proportional to **n** (taking on average **n/2** time units). In order-notation this speed will expressed as

O(n)

Ex. 0.6

Pseudocode for the contains() method in a linked list

1. create a **roving pointer** and assign the value **front** to it
2. while pointer != null
3. if pointer's data == k
4. return true
5. else
6. pointer = pointer.next
7. end of while loop
8. return false

Like with an array, this is an **O(n)** algorithm.

Ex. 0.7

The execution time of a search in binary try is **logarithmic**. In order notation (Big-Oh notation) it is

O(logN)

In a **logarithmic** algorithm (with the base being 2), when **N** is doubled the execution time grows **k** by one only. So in our exercise

N=15 results in k=4
N=31 (appr. twice of 15) k=5
N=63 k=6

Ex. 0.10

Remainder	Array Items
0	11
1	23
2	2, 13
3	3, 47
4	37
5	5
6	17
7	7, 29
8	19, 41
9	31
10	43

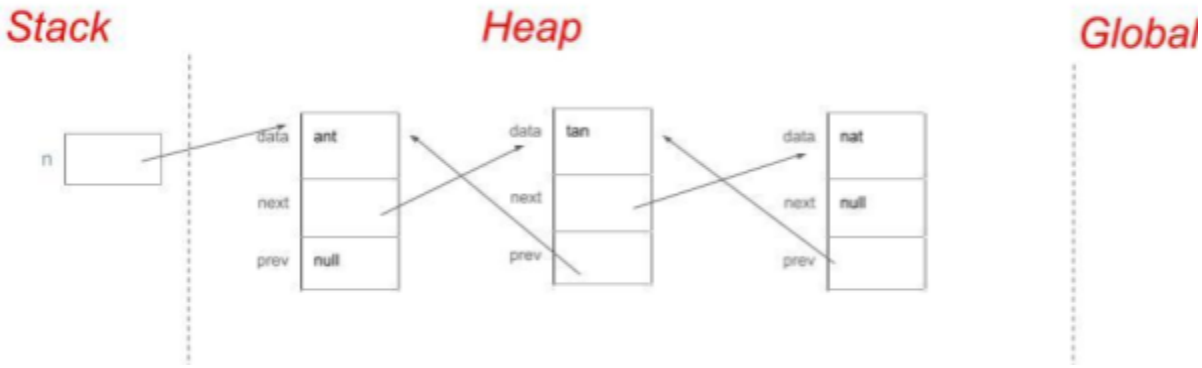
Ex. 0.13

```
class Node {
        String data
        Node next;
        Node prev;
}
```

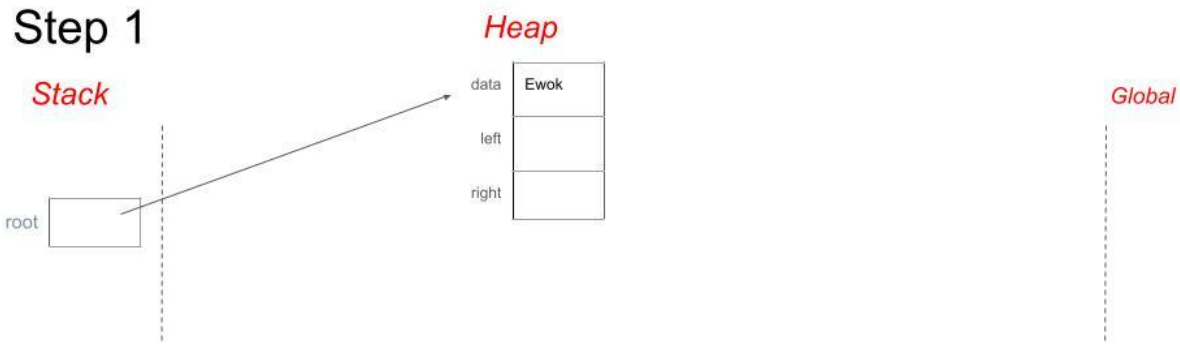
assume that the class using Node, has Node front and Node rear
Algorithm: add() in a doubly-linked list

Input: String s

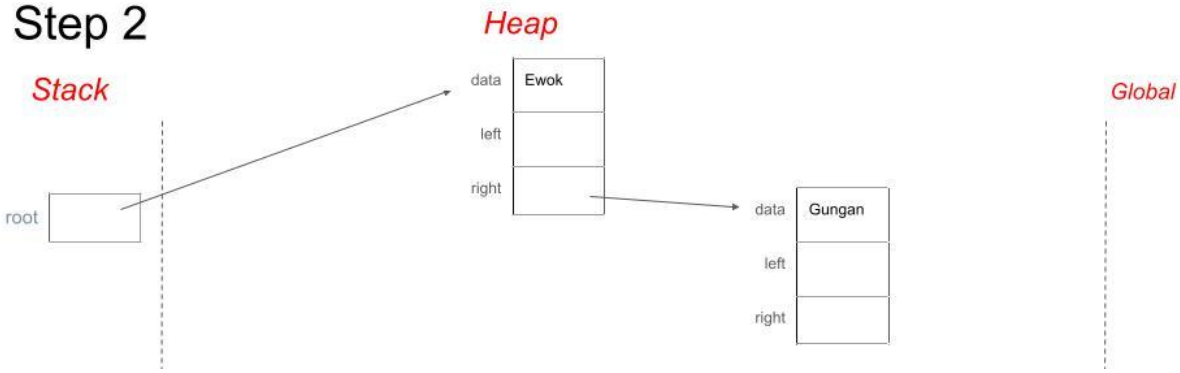
1. if front == null
2. create a new Node
3. assign s to this Node.data
4. else
5. create a new Node
6. assign s to this Node.data
7. assign this new Node to rear.next
8. assign rear to new Node.prev
9. assign new Node to rear
10. end-if/else
11. end-add()



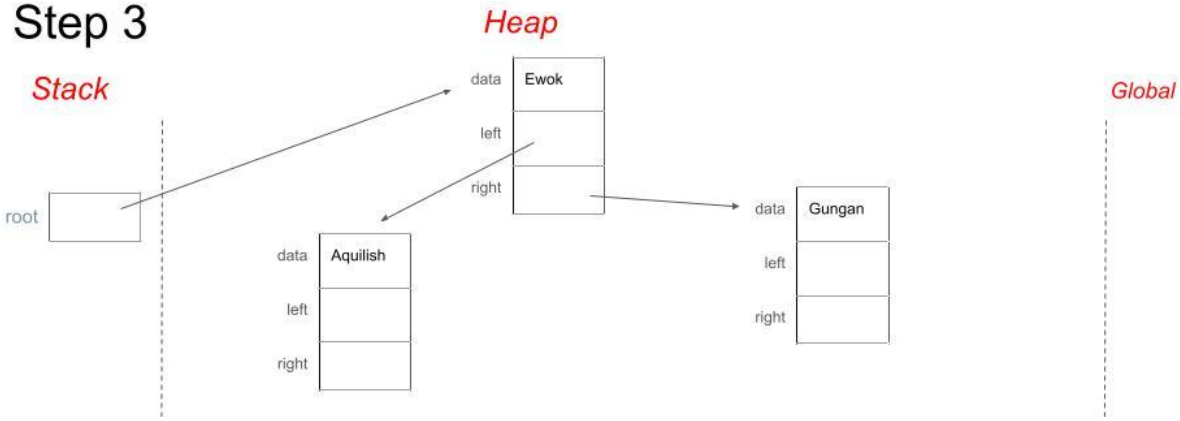
Ex. 0.14



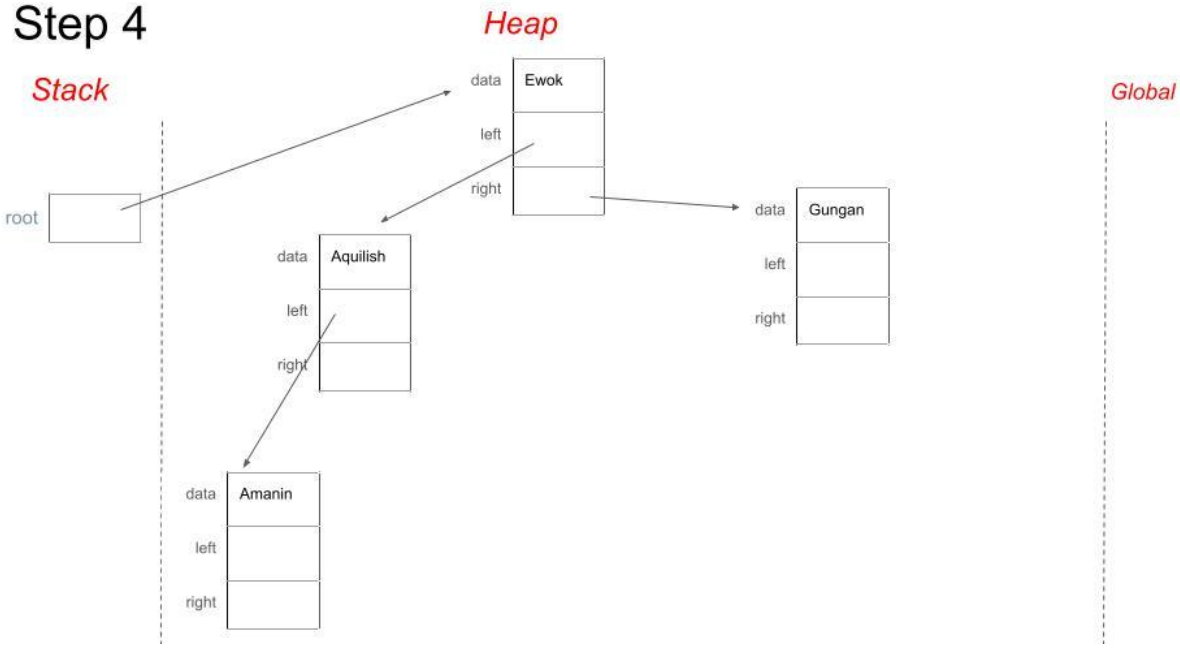
Step 2



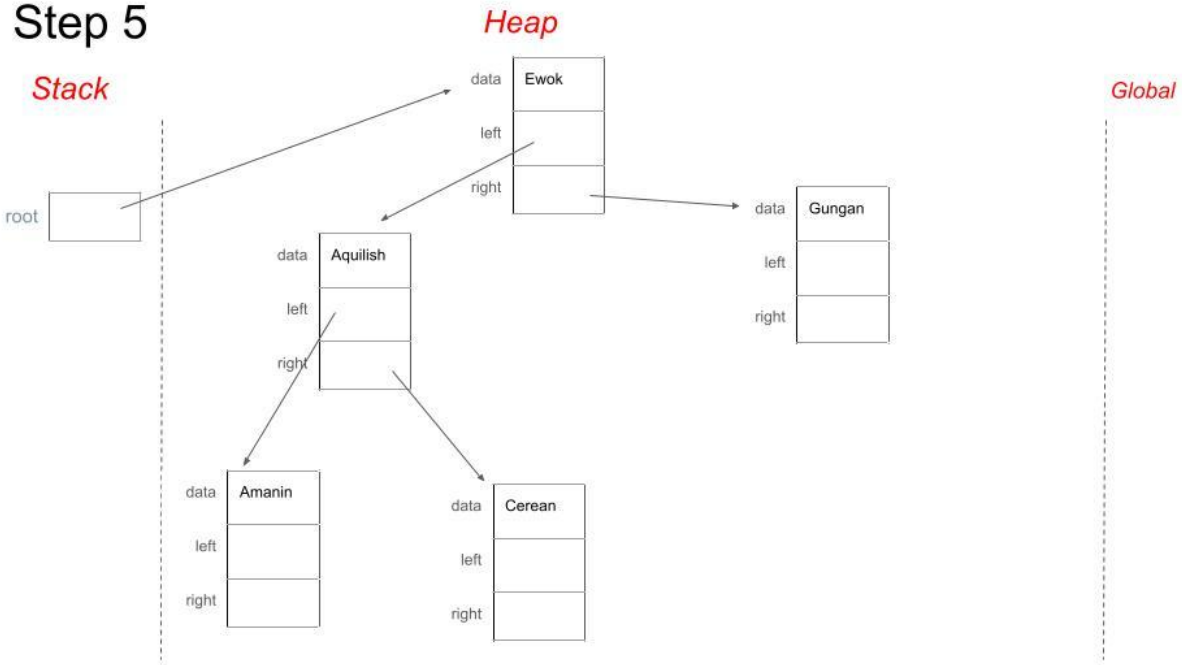
Step 3



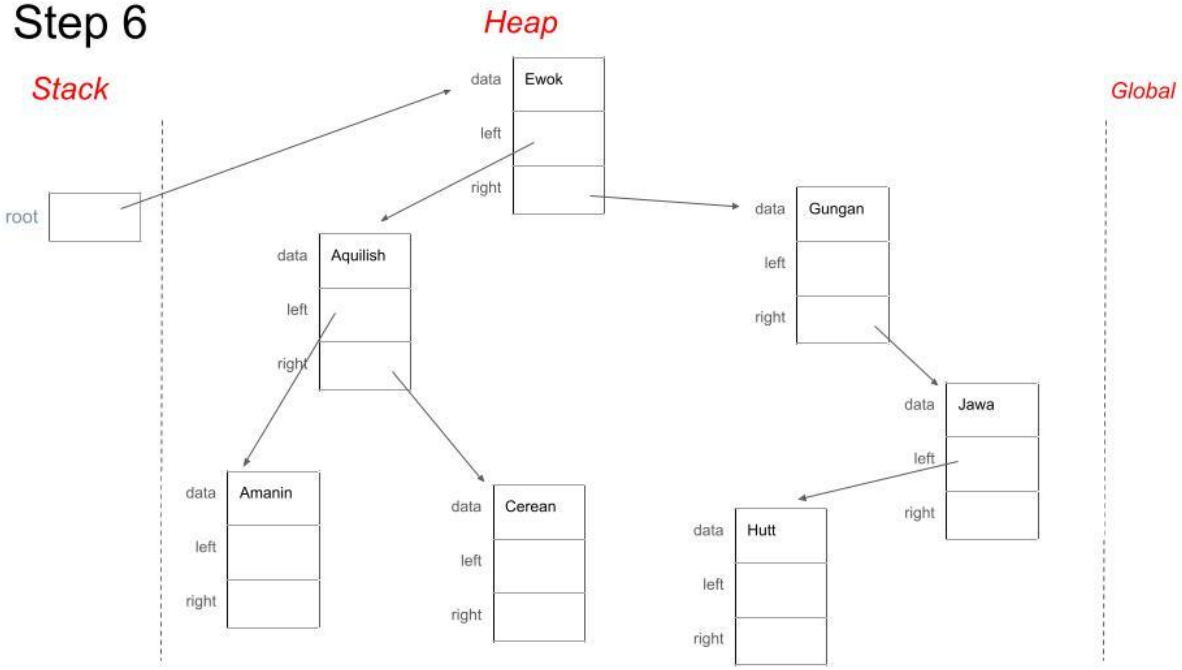
Step 4

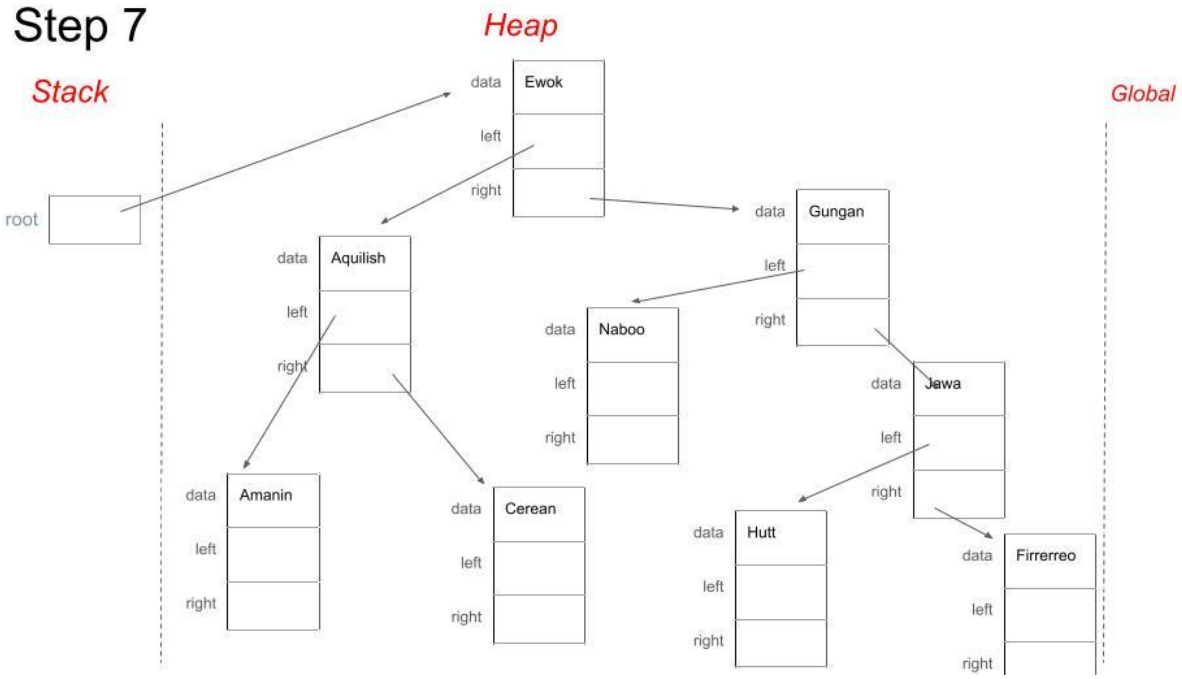


Step 5

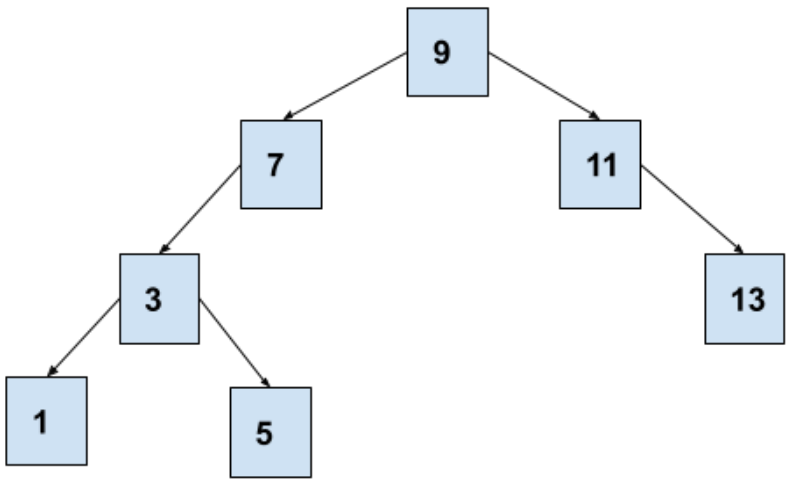

Heap
Stack
Global
root
data
Ewok
left
right
data
Gungan
left
right
data
Aquilish
left
right
data
Amanin
left
right
data
Cerean
left
right

Step 6

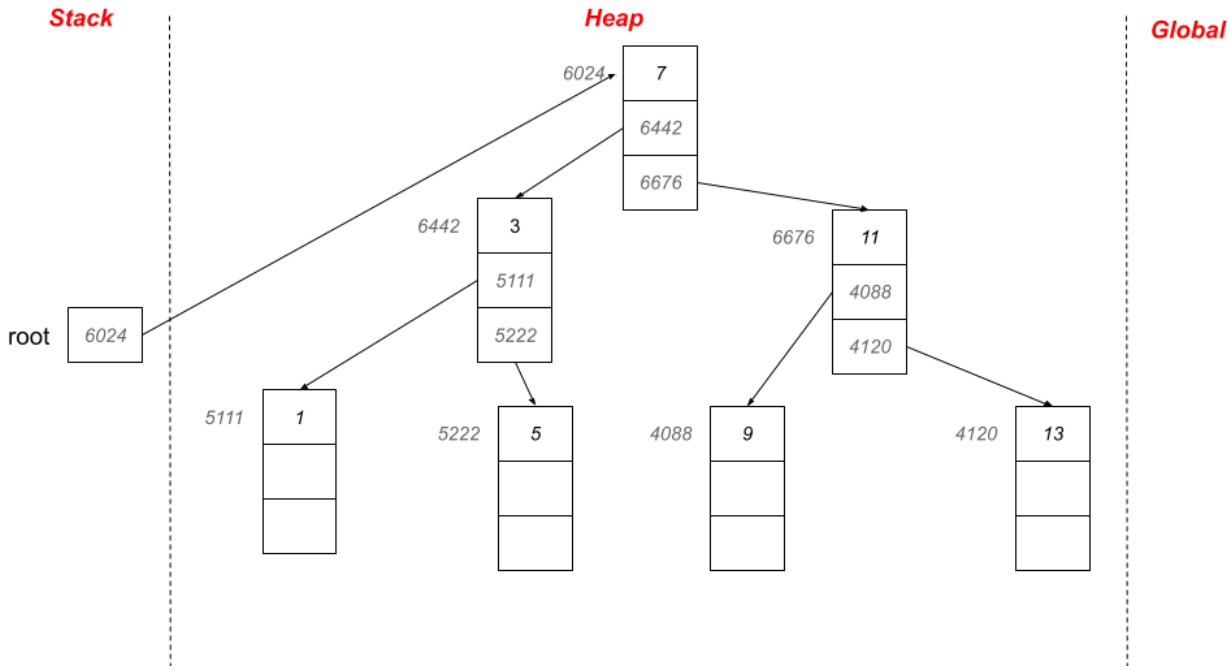

Heap
Stack
Global
root
data
Ewok
left
right
data
Gungan
left
right
data
Aquilish
left
right
data
Jawa
left
right
data
Amanin
left
right
data
Cerean
left
right
data
Hutt
left
right


Step 7
Heap
Stack
Global
root
data
Ewok
left
right
data
Gungan
left
right
data
Aquilish
left
right
data
Naboo
left
right
data
Jawa
left
right
data
Amanin
left
right
data
Cerean
left
right
data
Hutt
left
right
data
Firrerreo
left
right

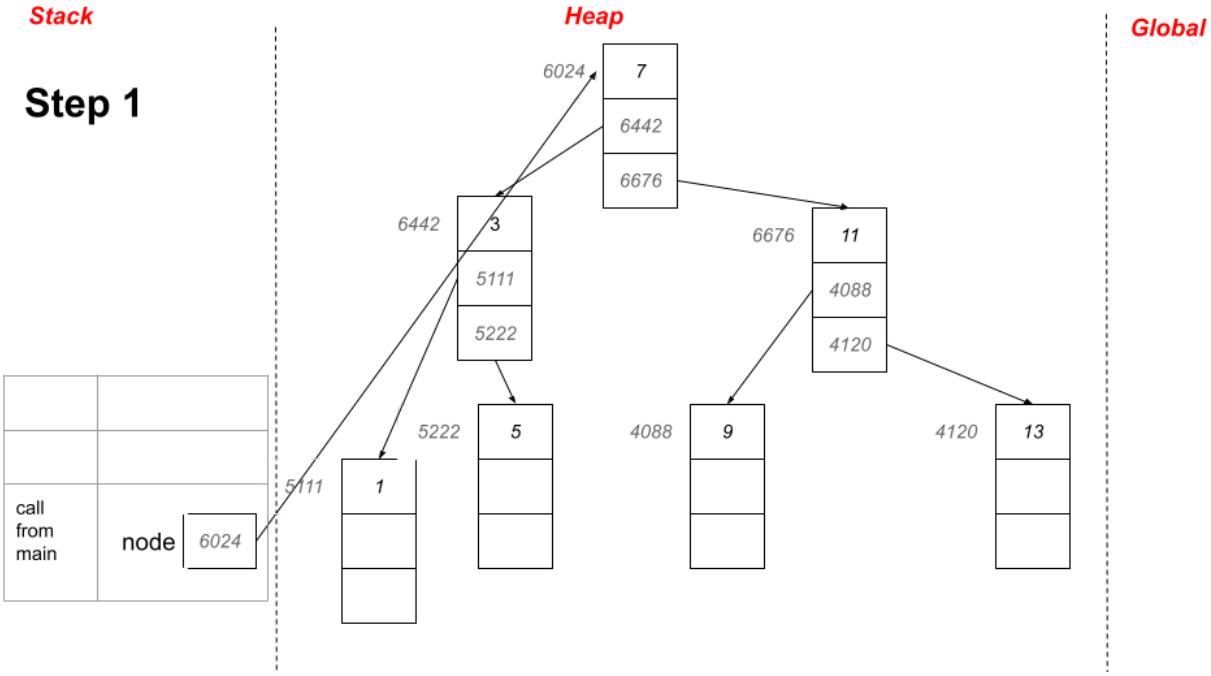
Ex. 0.15

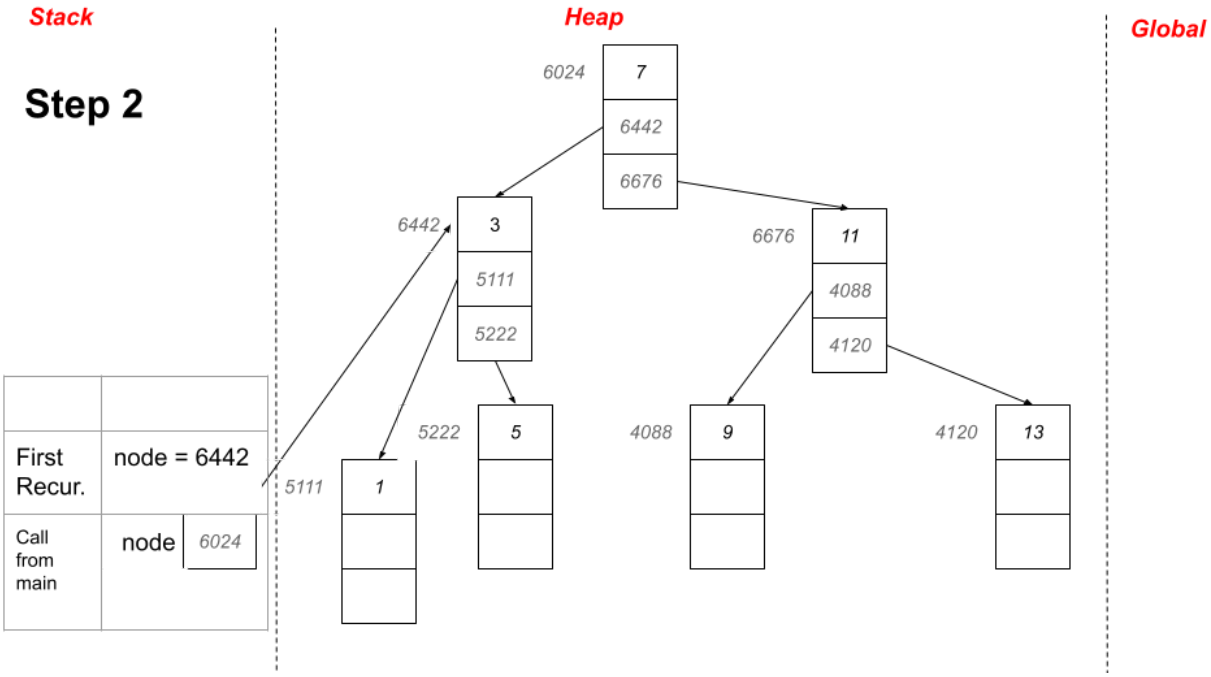

9
7
11
3
13
1
5

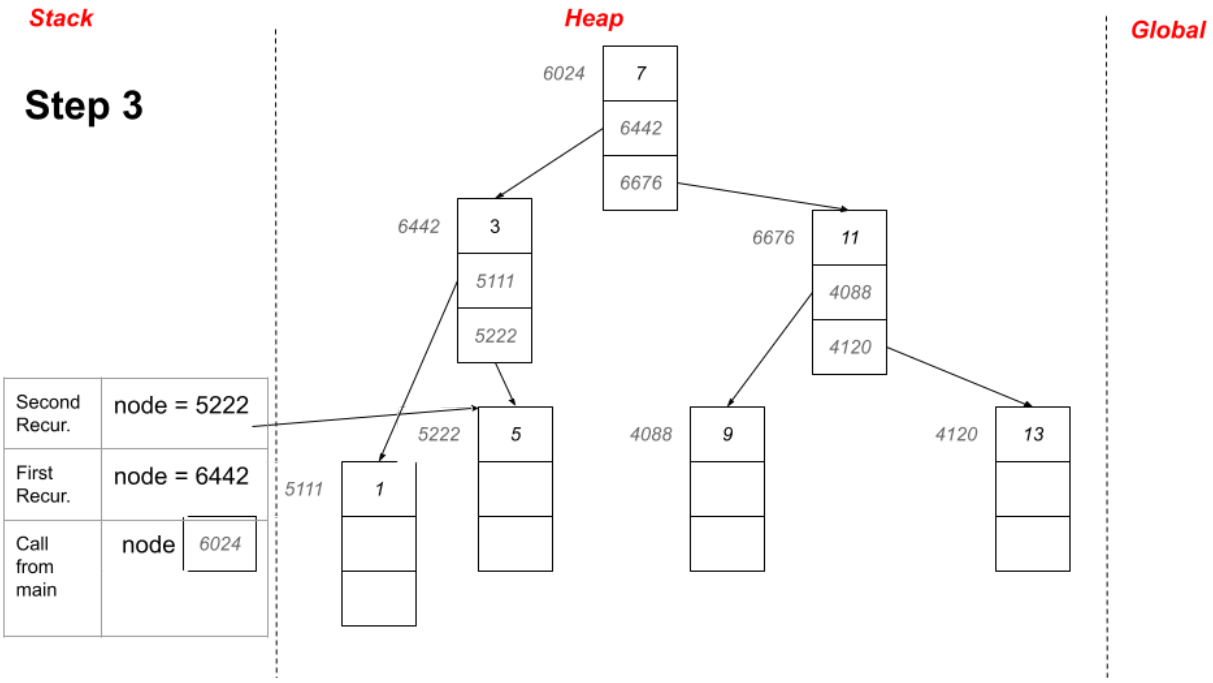
Ex. 0.16


Stack
Heap
Global
root
6024
6024
7
6442
6676
6442
3
5111
5222
6676
11
4088
4120
5111
1
5222
5
4088
9
4120
13

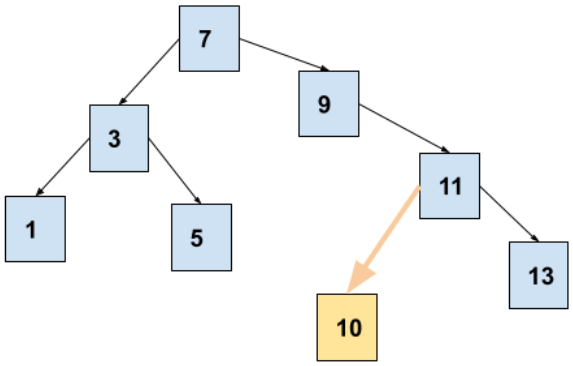
Tracing through recursive calls to find 5:






Stack
Heap
Global
Step 3
Second Recur.
node = 5222
First Recur.
node = 6442
Call from main
node
6024
6024
7
6442
6676
6442
3
5111
5222
6676
11
4088
4120
5222
5
5111
1
4088
9
4120
13

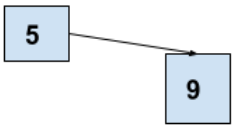
Ex. 0.17


7
3
9
1
5
11
13
10

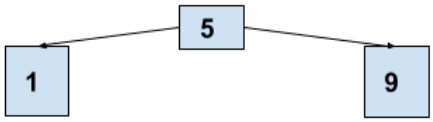
Ex. 0.18

Step 1

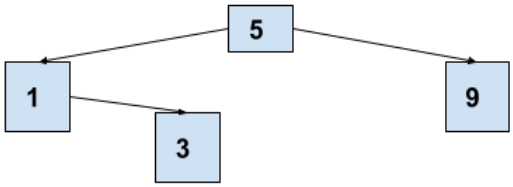
Step 2



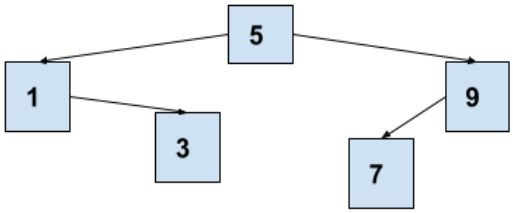
Step 3



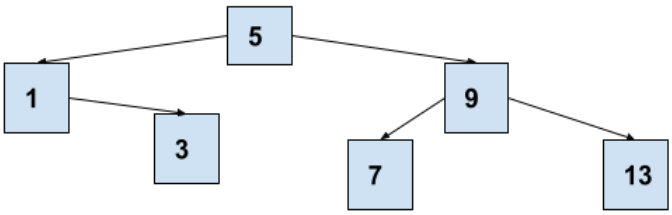
Step 4



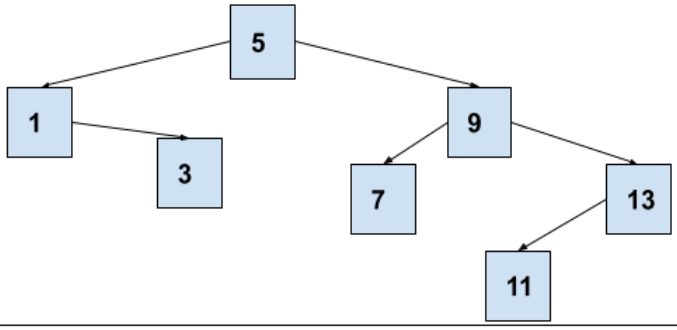
Step 5



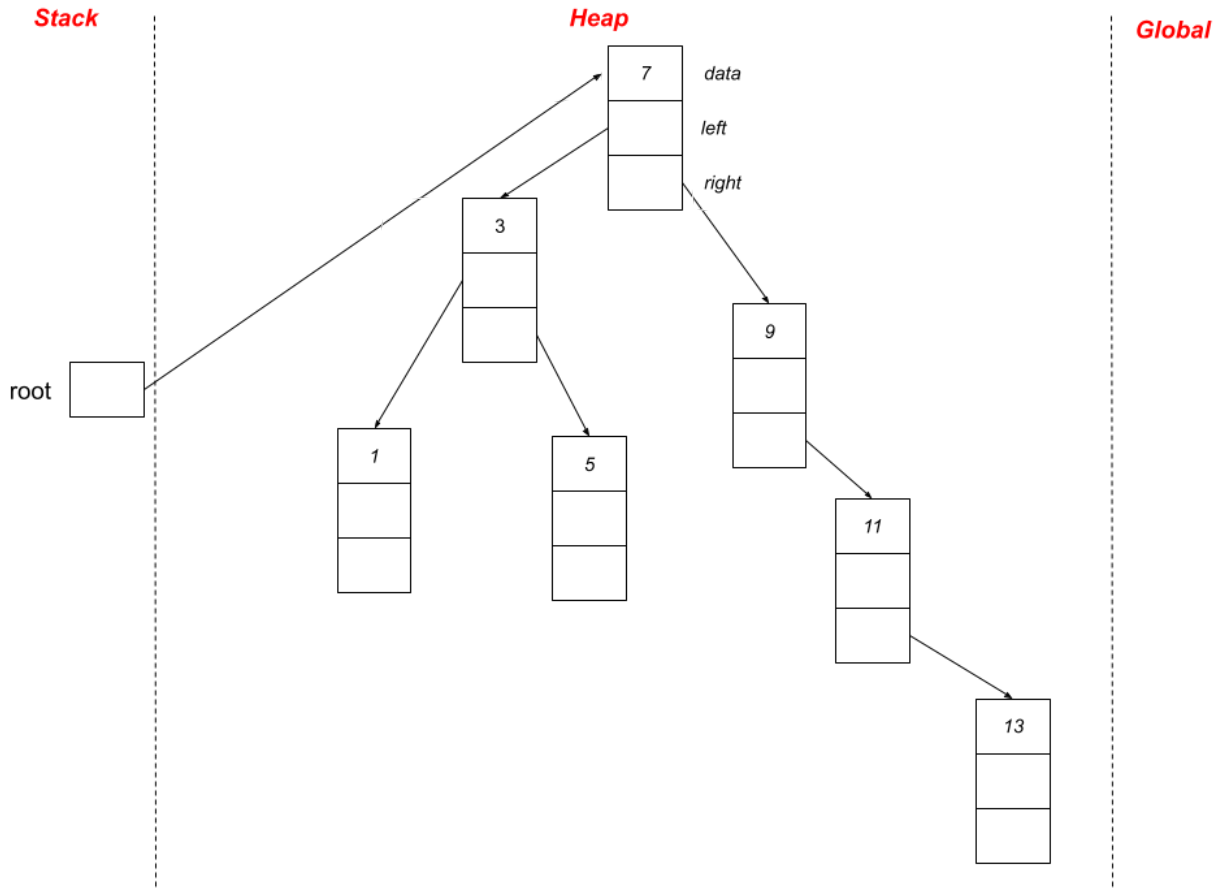
Step 6



Step 7



Ex. 0.19



Ex. 0.22

In-Order

Tracing execution of an in-order traversal, showing the stack contents:

Step	Stack	Output
1.	node.data = 7 [printInOrder (node.left)]	
2.	node.data = 3 [printInOrder (node.left)]	
3.	node.data = 1 [printInOrder (node.left)]	
4.	[node=null => return]	
5.	node.data = 1 [print note.data] [printInOrder (node.right)]	1
6.	[node=null => return]	
7.	node.data = 3 [print note.data] [printInOrder (node.right)]	3
8.	node.data = 5 [printInOrder (node.left)]	
9.	[node=null => return]	
10.	node.data = 5 [print note.data] [printInOrder (node.right)]	5
11.	[node=null => return]	
12.	node.data = 3 [return]	
13.	node.data = 7 [print note.data] [printInOrder (node.right)]	7
14.	node.data = 9 [printInOrder (node.left)]	

15.	[node=null => return]	
16.	node.data = 9 [print note.data] [printInOrder (node.right)]	9
17.	node.data = 11 [printInOrder (node.left)]	
18.	[node=null => return]	
19.	node.data = 11 [print note.data] [printInOrder (node.right)]	11
20.	node.data = 13 [printInOrder (node.left)]	
21.	[node=null => return]	
22.	node.data = 13 [print note.data] [printInOrder (node.right)]	13
23.	[node=null => return]	
24.	node.data = 11 return	
25.	node.data = 9 return	
26.	node.data = 7 return	

Pre-Order

Tracing execution of an pre-order traversal, showing the stack contents:

Step	Stack	Output
1.	node.data = 7 [print note.data] [printInOrder (node.left)]	7
2.	node.data = 3 [print note.data]	3

	[printInOrder (node.left)]	
3.	node.data = 1 [print note.data] [printInOrder (node.left)]	1
4.	[node=null => return]	
5.	node.data = 1 [printInOrder (node.right)]	
6.	[node=null => return]	
7.	node.data = 1 [return]	
8.	node.data = 3 [printInOrder (node.right)]	
9.	node.data = 5 [print note.data] [printInOrder (node.left)]	5
10.	[node=null => return]	
11.	node.data = 5 [printInOrder (node.right)]	
12.	[node=null => return]	
13.	node.data = 5 [return]	
14.	node.data = 3 [return]	
15.	node.data = 7 [printInOrder (node.right)]	
16.	node.data = 9 [print note.data] [printInOrder (node.left)]	9
17.	[node=null => return]	
18.	node.data = 9 [printInOrder (node.right)]	
19.	node.data = 11 [print note.data] [printInOrder (node.left)]	11

20.	[node=null => return]	
21.	node.data = 11 [printInOrder (node.right)]	
22.	node.data = 13 [print note.data] [printInOrder (node.left)]	13
23.	[node=null => return]	
24.	node.data = 13 [printInOrder (node.right)]	
25.	[node=null => return]	
26.	node.data = 13 [return]	
27.	node.data = 11 [return]	
28.	node.data = 9 [return]	
29.	node.data = 7 [return]	

Ex. 0.23

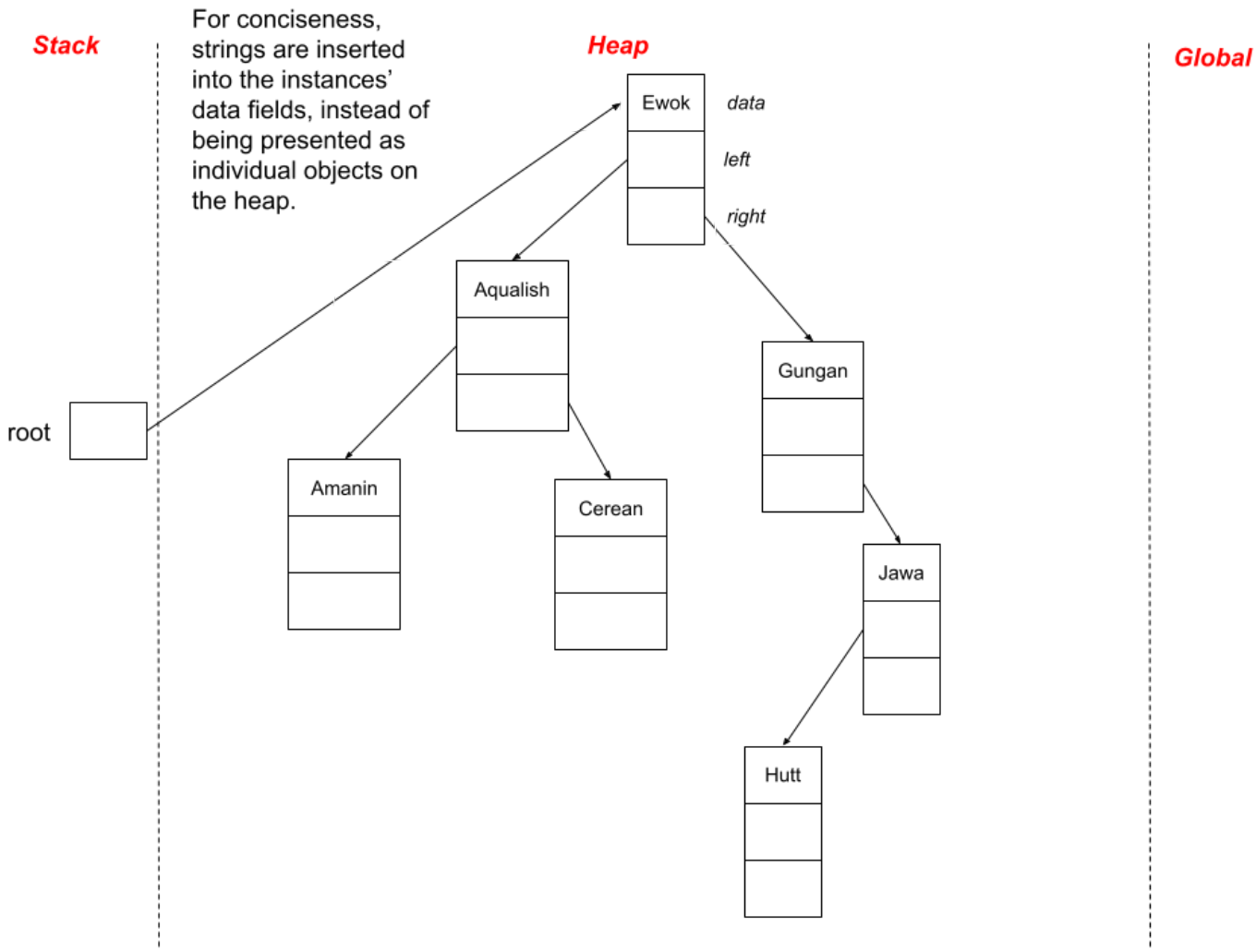
Pseudocode for **Post-Order** traversal

Algorithm: printPostorder (node)

1. **if** node=null
2. **return**
3. **endif**
4. printPostorder (node.left)
5. printPostorder (node.right)
6. print node.data

The output of the example given in this part of the instructional materials should be:
1, 5, 3, 13, 11, 9, 7

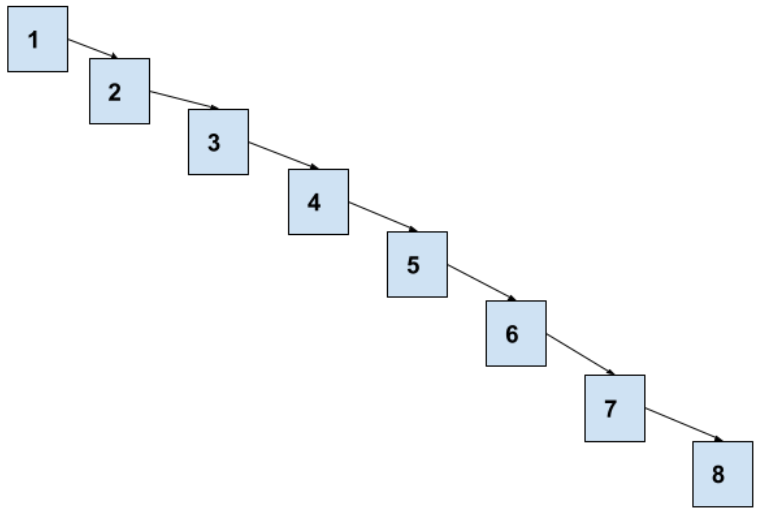
Ex. 0.26



Ex. 0.27

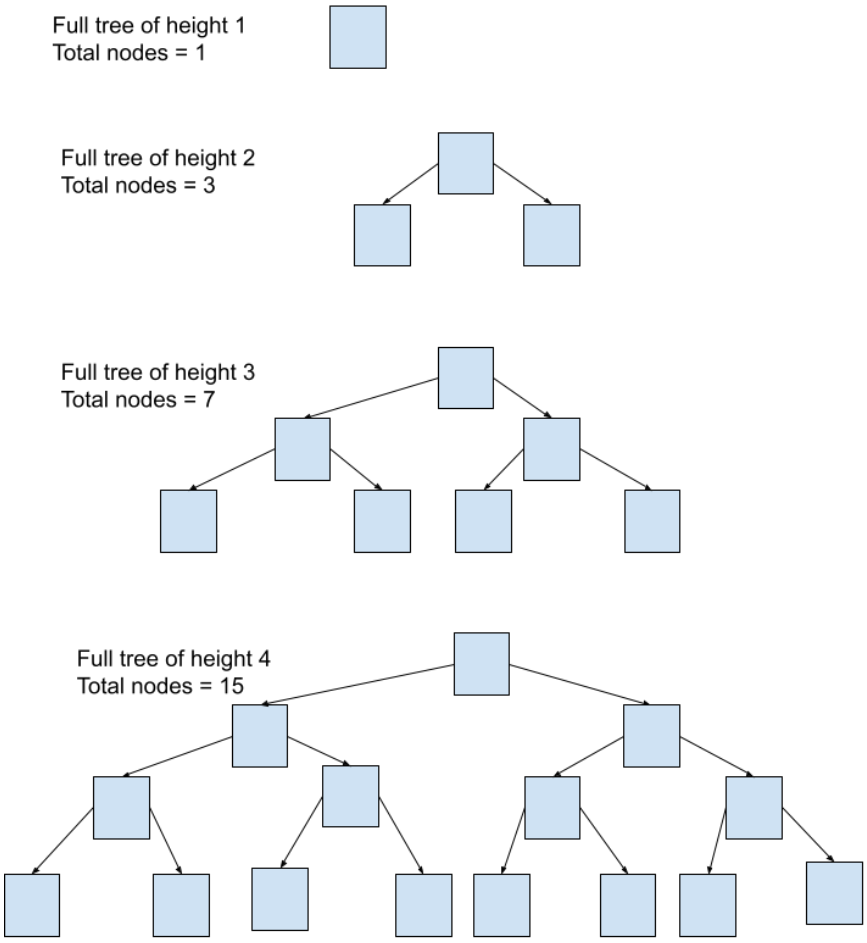
In a binary tree with the numbers 7, 3, 11, 1, 5, 9, 13 inserted in this order, the height (or depth) is **3**.

Ex. 0.28



The height of this tree is **8**.

Ex. 0.30



- In a full tree of height **h,** the tree has $\mathbf{2^h - 1}$ nodes.
- If a full tree has **n** nodes, its height is $\mathbf{\log_2(n + 1)}$.

Ex. 0.31

There is a question mark next to the insertion time for ArrayList because if the array underlying the ArrayList runs out of its present space, then a new, larger array is created and the elements of the old, smaller array is copied over into this new array. Such copying increases the insertion time for this one occasion by n (because copying the elements will take O(n) time).

Ex. 0.32

The pattern in the given array of lists is that the array's index number is the result of the calculation **K % 5,** where K is a number in one of the lists in the array. For example, under index 0, there is 10 because 10 % 5 = 0; under index 1, there are 11, 46, and 51 because applying %5 to all of them results in 1; etc.