Heaps
(priority queues)

the printer example

- You have 100 documents to print
- Routing 100 jobs to the printer could be done with a queue

remember Lecture 4: stacks (LIFO) and queues (FIFO)

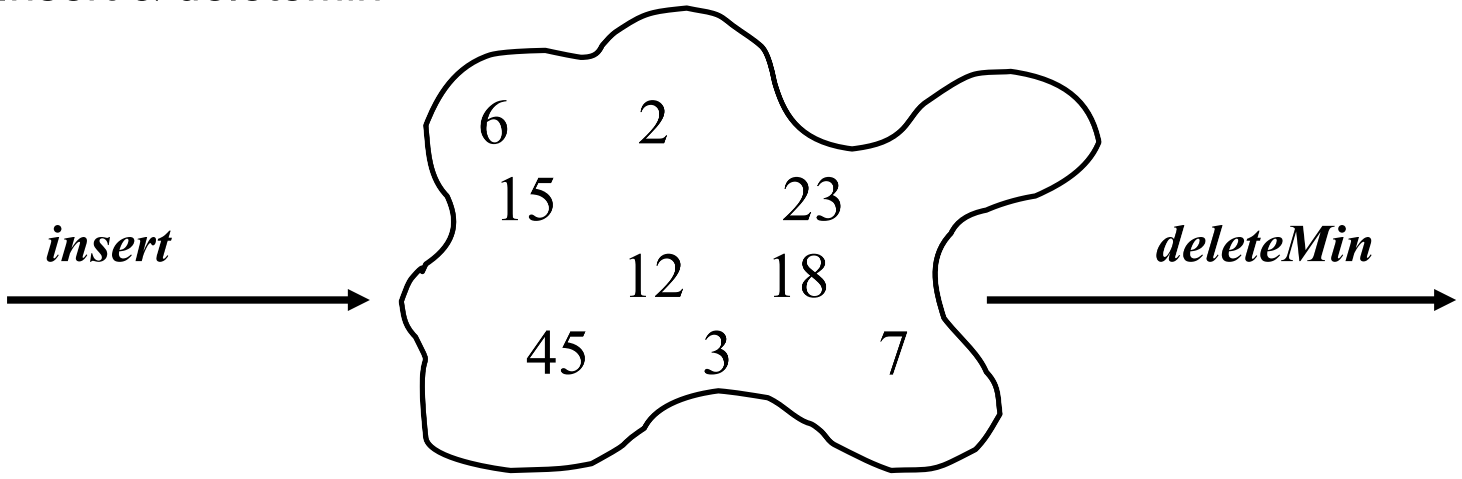
- What if 99 of the documents are single-page and 1 is 100 pages?
- It makes sense to print first the single-page documents
- A stack does not support such selection, i.e., assigning priorities to items

In general:

- Short jobs should go first
- Most urgent jobs should go first

Major operations

- Insert & deleteMin



- Fundamental property we are implementing:

  for two elements in the queue, *x* and *y*, if *x* has a **lower** “priority” value than *y*, *x* will be deleted before *y*

…using arrays and linked lists

	insert	deleteMin
Unsorted list (Array)	O(1)	O(n)
Unsorted list (Linked-List)	O(1)	O(n)
Sorted list (Array)	O(n)	O(1)*
Sorted list (Linked-List)	O(n)	O(1)

Binary heap properties

- **Structure**

  a “good” binary tree – *completeness*

  Note: we could be talking about BST, but it would be an overkill (we do not need the BST property – Lecture 5)

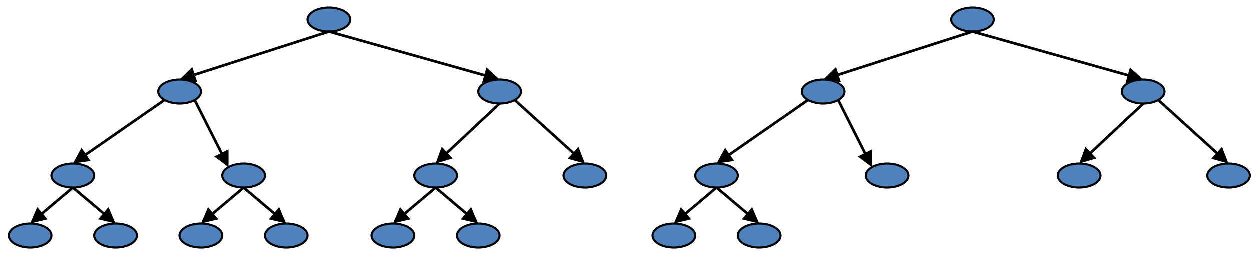
- **Order**

  item priorities determine the locations in the tree

structure

complete binary tree

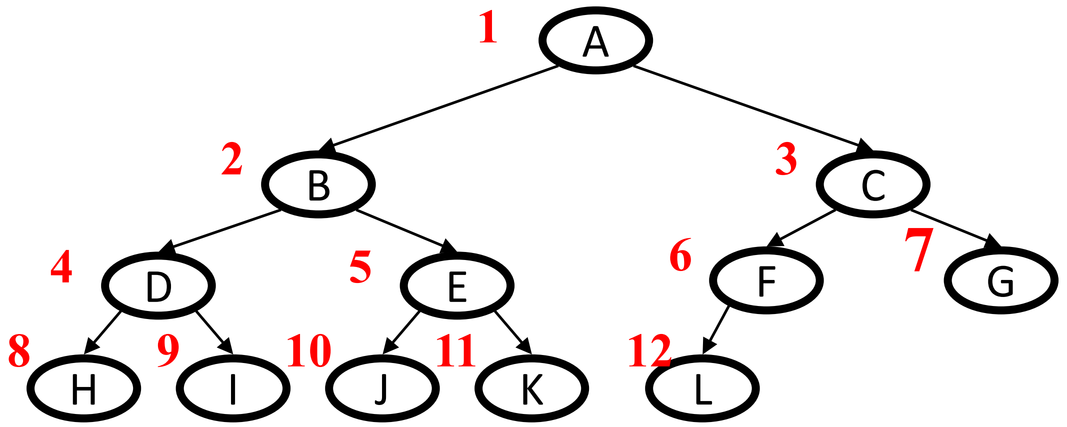
- A binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right



- Depth is always O(log n); next open location always known

traversal

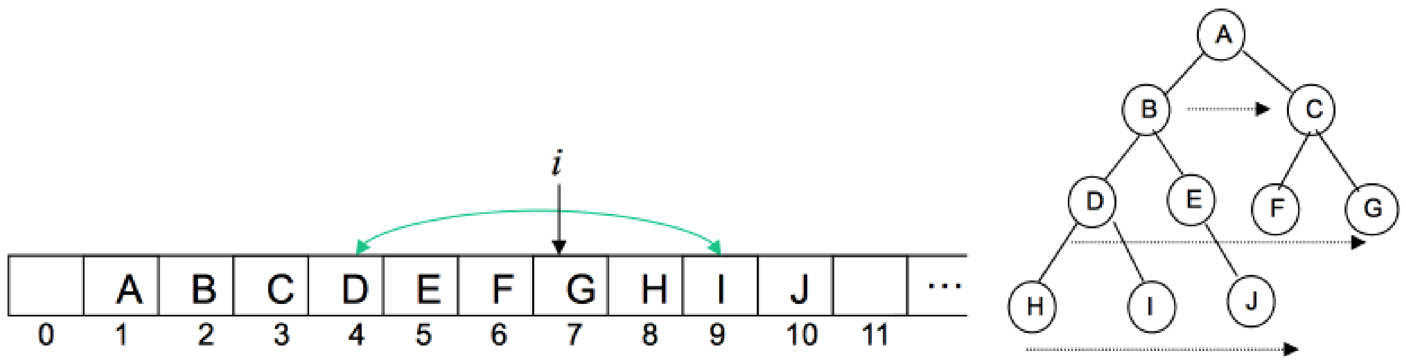
from node **i**:
left child - right child -parent



implicit (array) implementation:

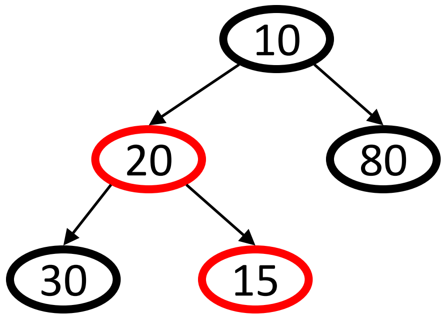
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**
1	2	3	4	5	6	7	8	9	10	11	12

The left child of *a*[*i*] is *a*[2*i*], the parent of *a*[*i*] is *a*[□*i*/2□]

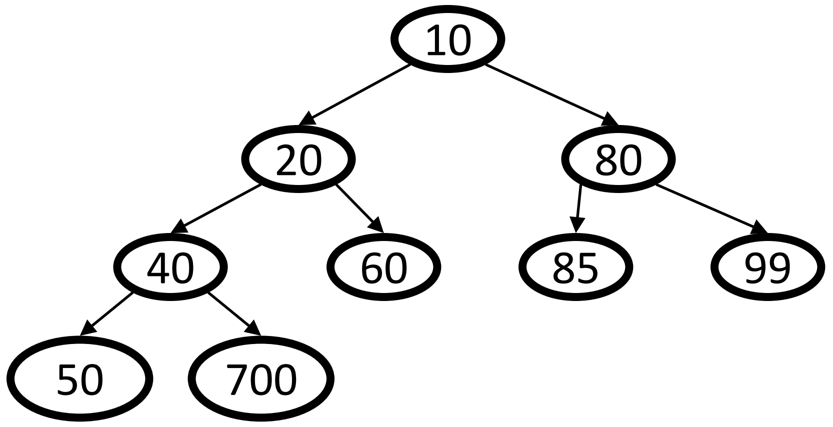


order

- for every non-root node *X*, the key in the parent of *X* is smaller than (or equal to) the key in *X*
- the root is the smallest element
- *findMin* is a constant time operation.



not a heap

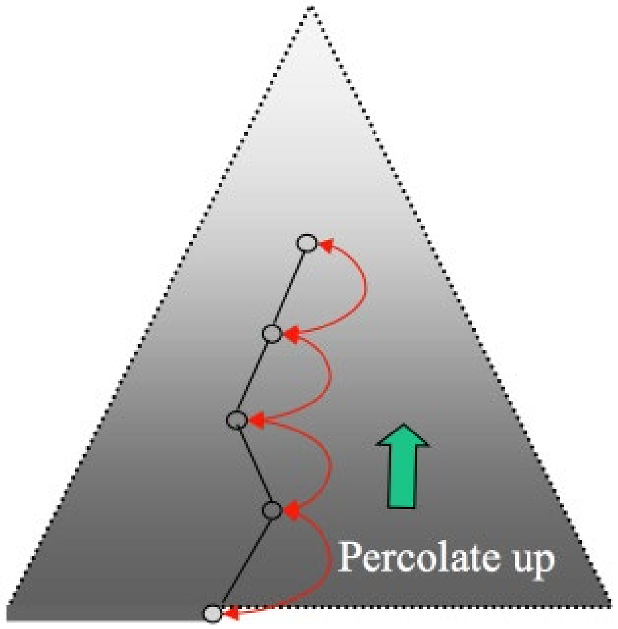


insert

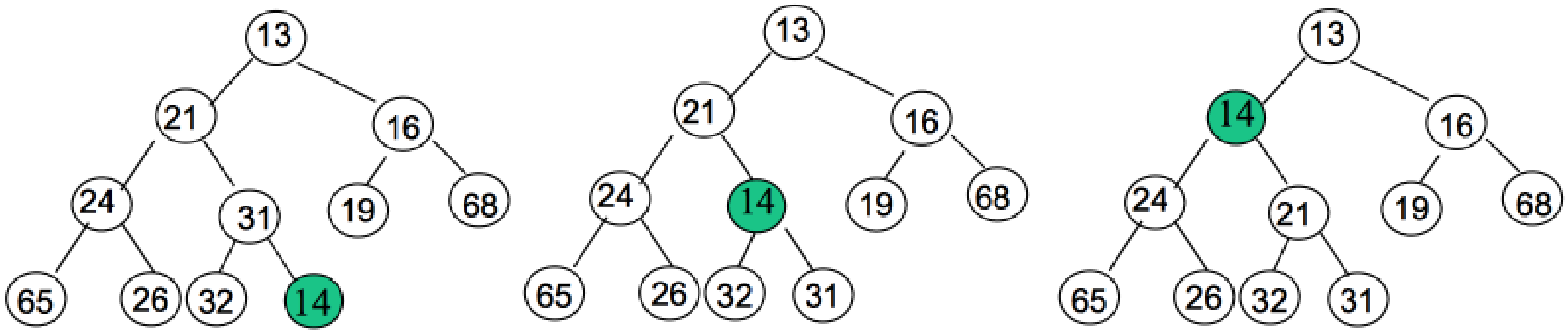
Procedure: percolateUP

(propagate a “bubble” upwards--towards the root)

1. Generate an empty node at the end of the array
2. If the new element *X* can be put in the empty node without violating the heap order, do it; otherwise move the parent of the empty node into the empty node to generate a new empty node (hole) at the parent node.
3. Repeat (2) with the new empty node until *X* can be inserted.



example

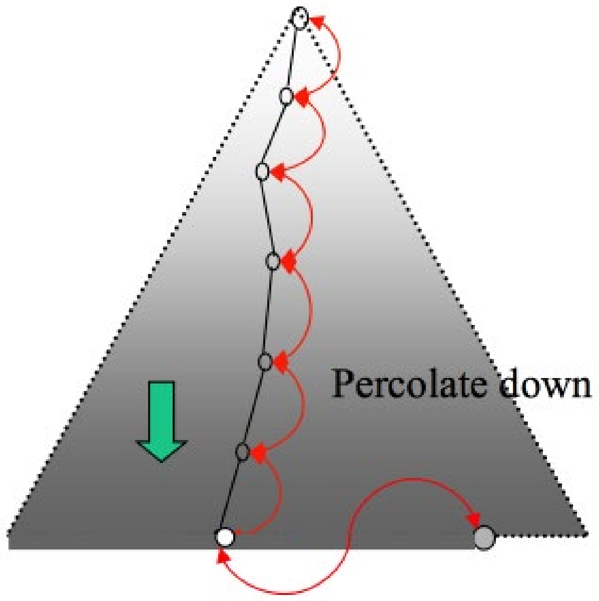

13
21
16
24
31
19
68
65
26
32
14
13
21
16
24
14
19
68
65
26
32
31
13
14
16
24
21
19
68
65
26
32
31

deleteMin

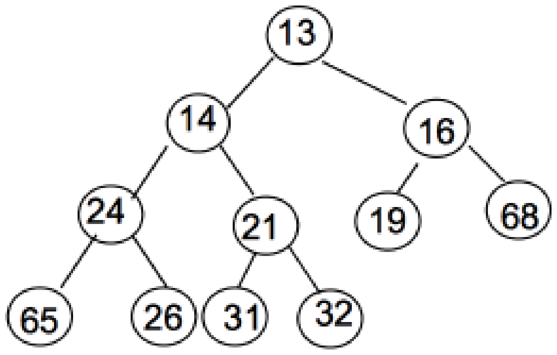
Procedure: percolateDOWN

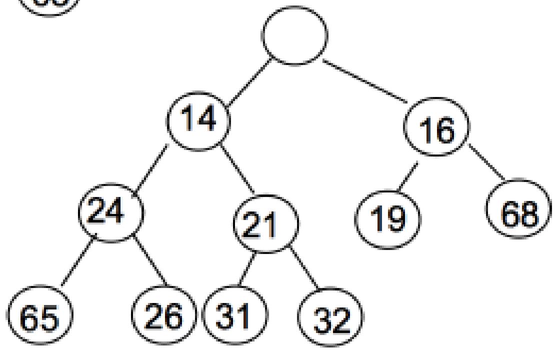
(propagate a “bubble” downwards--towards the leaves)

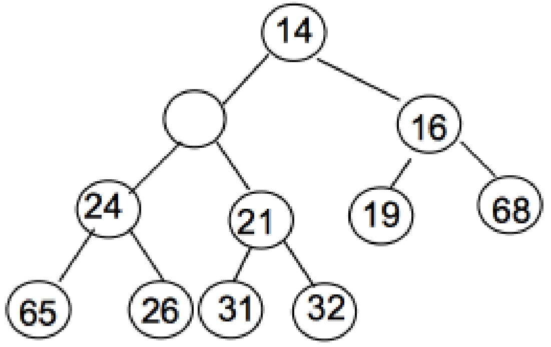
- Remove the root, which generate a hole at the root.
- If the last element of the heap can be moved into the hole, do it, otherwise, move the smaller child of the hole into the hole, which generates a new hole.
- Repeat (2) with the new hole until the last element of the heap can be placed.

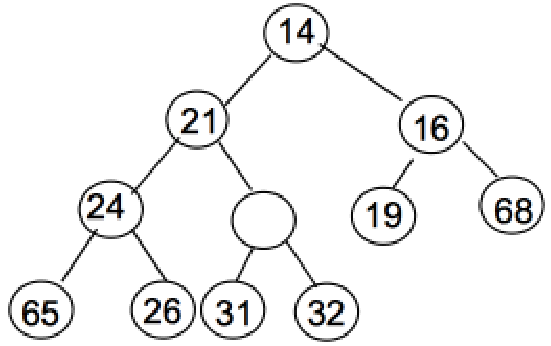


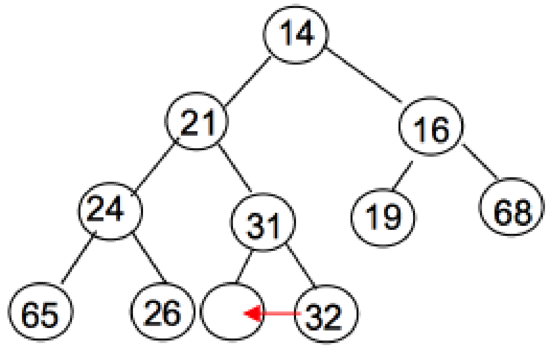
example


13
14
16
24
21
19
68
65
26
31
32


14
16
24
21
19
68
65
26
31
32


14
16
24
21
19
68
65
26
31
32


14
21
16
24
19
68
65
26
31
32

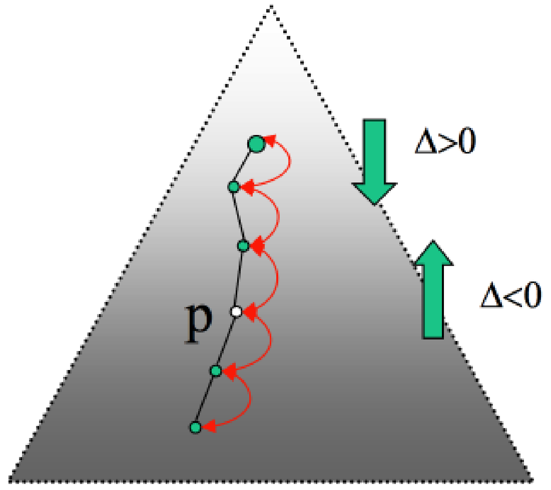

14
21
16
24
31
19
68
65
26
32

Other operations

- In many applications the priority of an object in a priority queue may change over time
  - e.g., if a job has been sitting in the printer queue for a long time increase its priority
- Must have some (separate) way of finding the position in the queue of the object to change (*e.g.* a hash table)

decresingKey (*P*, □) : *O(logN)*

incresingKey (*P*, □) : *O(logN)*



BuildHeap

- buildHeap: build a heap from *N* input items
- *N* insert operations: *O(N)* average, *O(N logN)* worst case
  1. put all element into a binary tree in arbitrary order
  2. check all non-leaf nodes in a bottom-up order
  3. for each node being checked, compare with its children to ensure heap-order, percolate down if necessary

example (from the textbook)

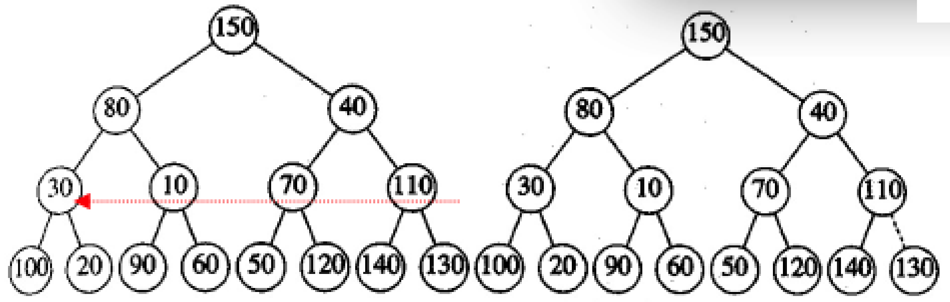


Figure 6.15 Left: initial heap; right: after percolateDown(7)

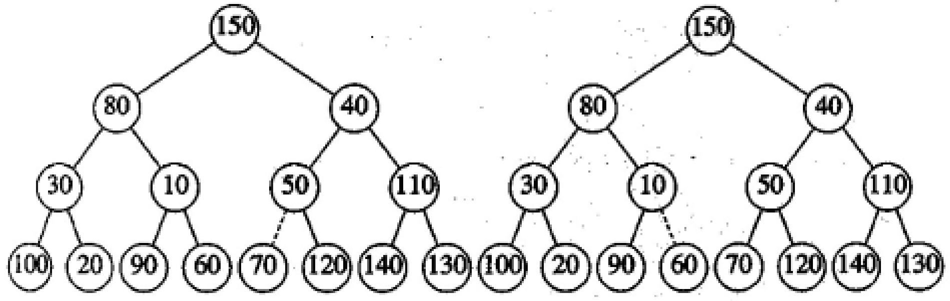


Figure 6.16 Left: after percolateDown(6); right: after percolateDown(5)

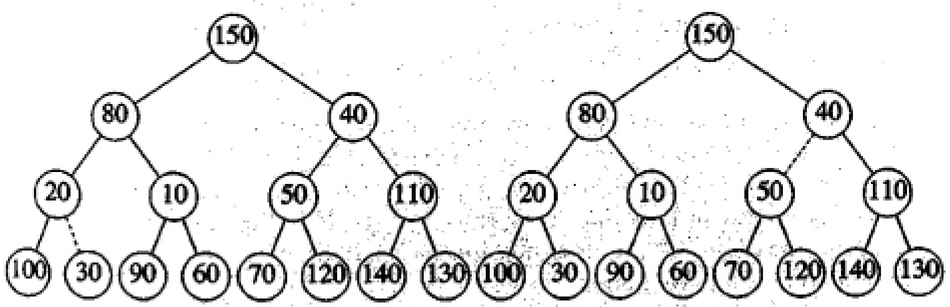
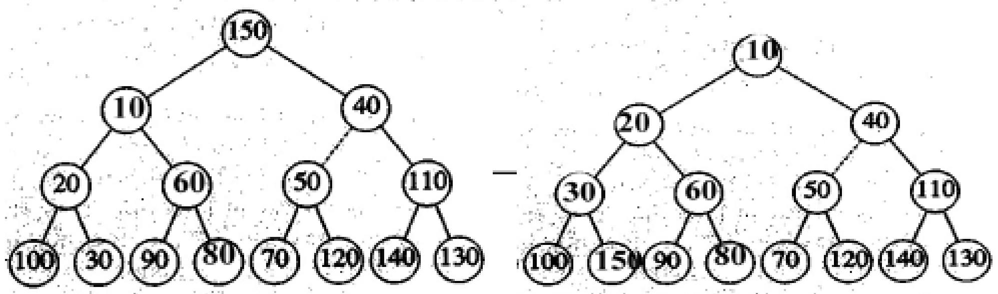


Figure 6.17 Left: after percolateDown(4); right: after percolateDown(3)



application: selection

- The selection problem:

  given *N* elements, find the *kth* smallest (or largest) element

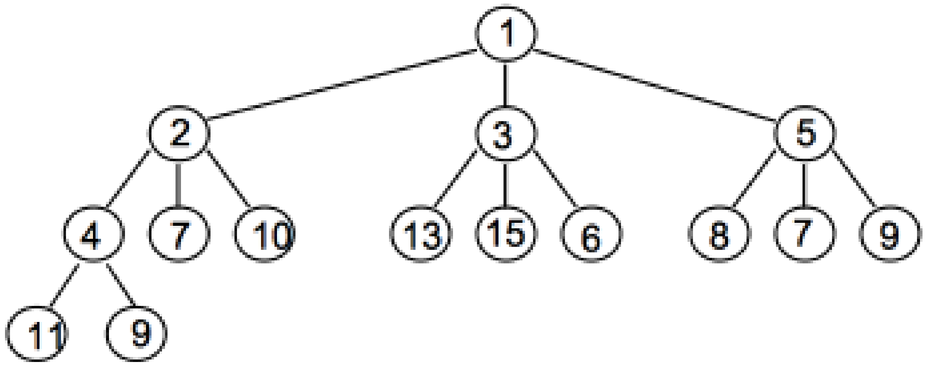
- A simple algorithm: sort the *N* elements in increasing order, and take the *kth* element. A simple sorting algorithm costs $O(N^2)$.

- The heap selection
  1. Read the *N* elements into an array
  2. Apply the buildHeap operation
  3. Perform *k* deleteMin operations

  Worst case:O(N+klogN)=O(NlogN)
  When k = N, the algorithm becomes a O(N logN) sorting algorithm.

Generalization: d-heaps

- An extension of binary heap — a *d-ary* tree structure



- *insert: O(log$_d$N)*
- *deleteMin: O(d log$_d$N) →compare with d children*
- Array implementation is not as efficient
- Maybe used for disk storage (similar to *B*-trees)
- “Merge” operation is hard

One more operation: **merge**

- When we have 2 sets of job queues, we may want to merge into one queue
- Merge two heaps H1 and H2 of size O(N)
- The simple idea:

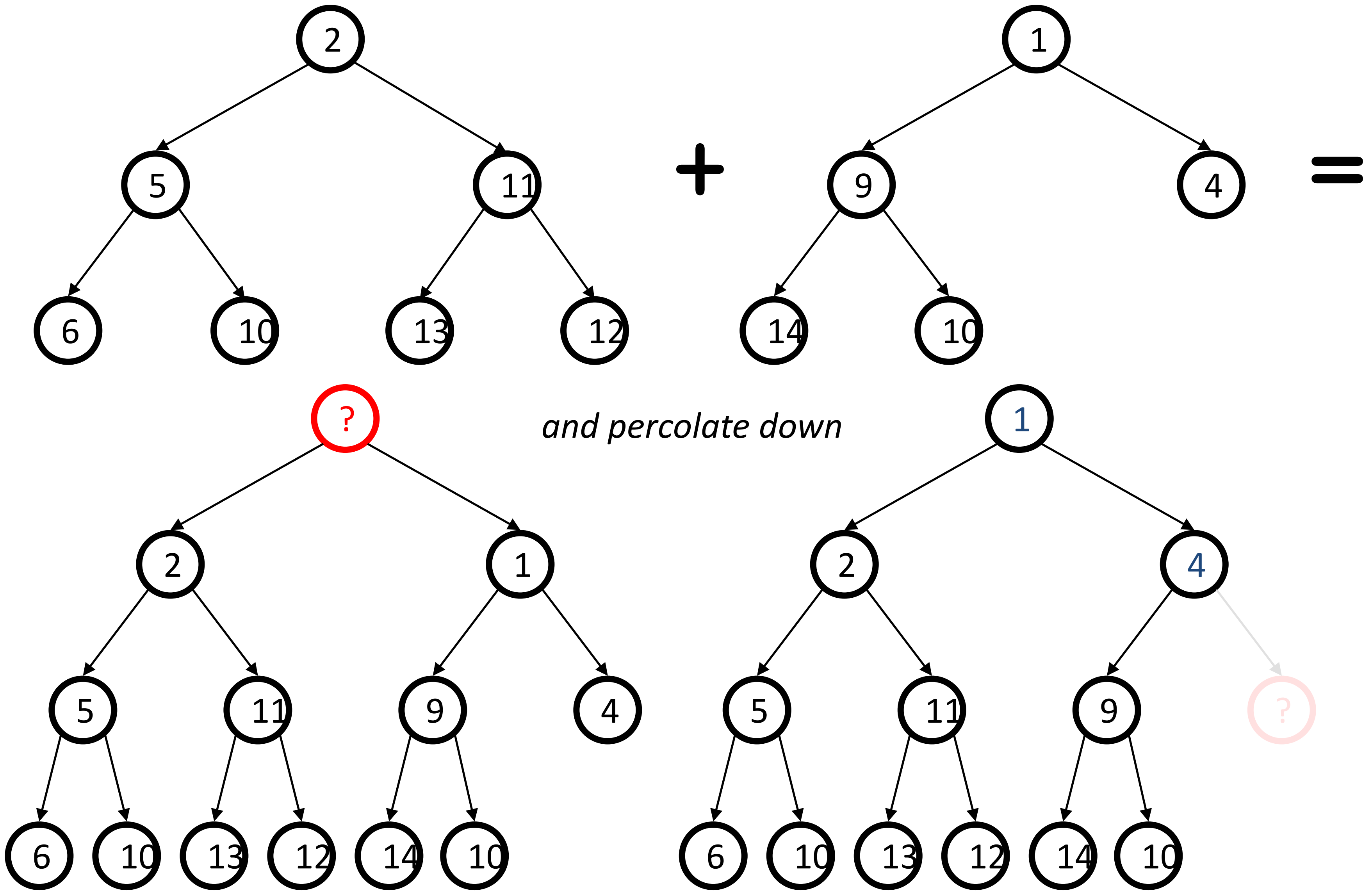
  Copy H2 at the end of H1 (assuming array implementation) and BuildHeap
- Can we do O(logN)?

IDEA Focus all heap maintenance work in one small part of the heap

-- Make one heap unbalanced (left-heavy)

-- Do all the merging work on the right

…what we could do instead



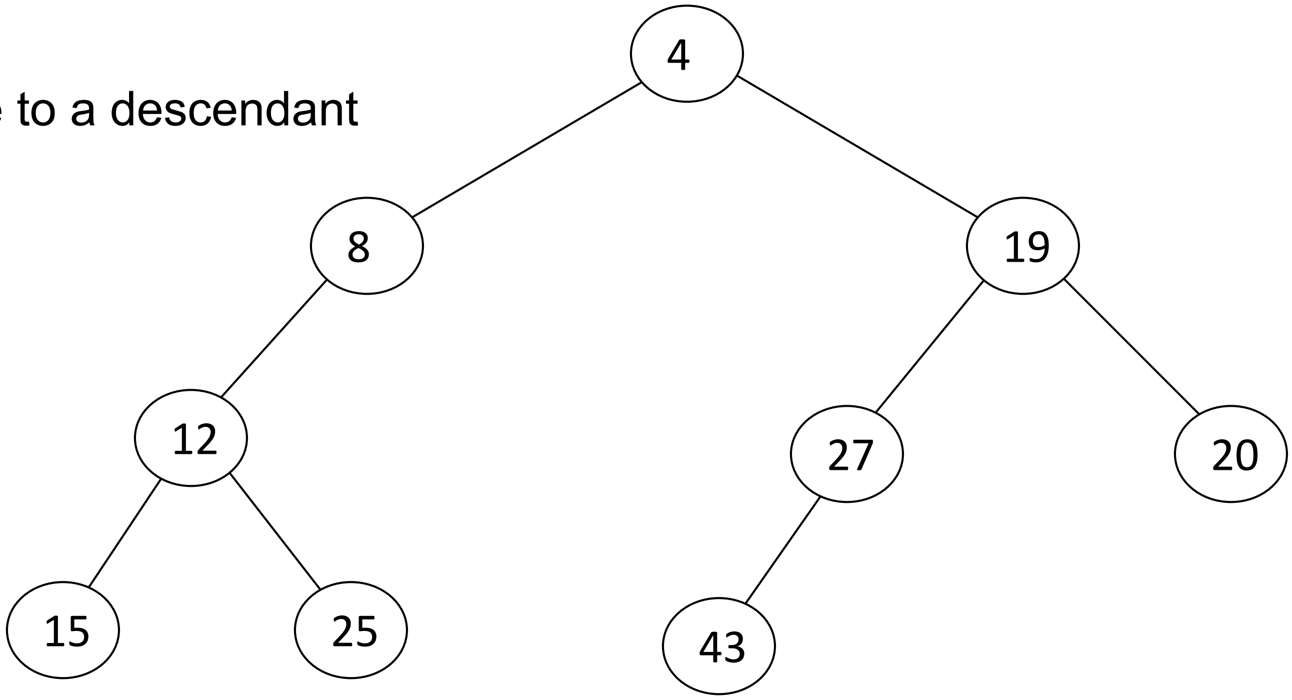
But we give up completeness

back to the idea of merging to the right
null path

- *null path length (npl)* of a node *x* = the number of nodes between *x* and a null in its subtree

OR

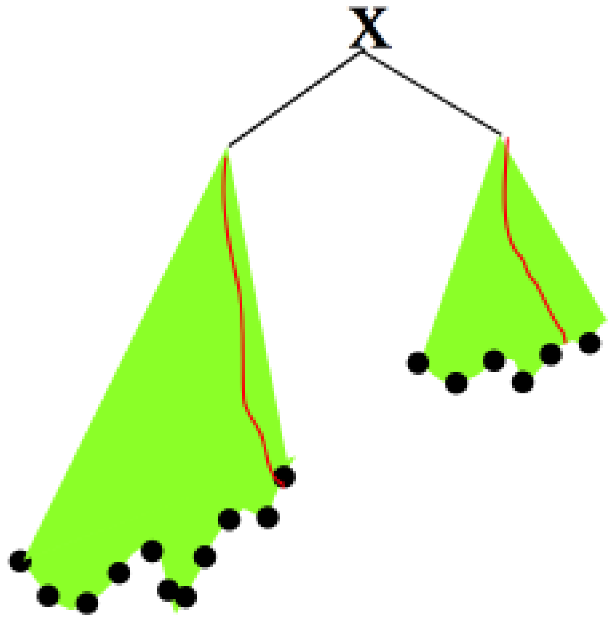
- npl(x) = min distance to a descendant with 0 or 1 children
- Npl(null) = −1



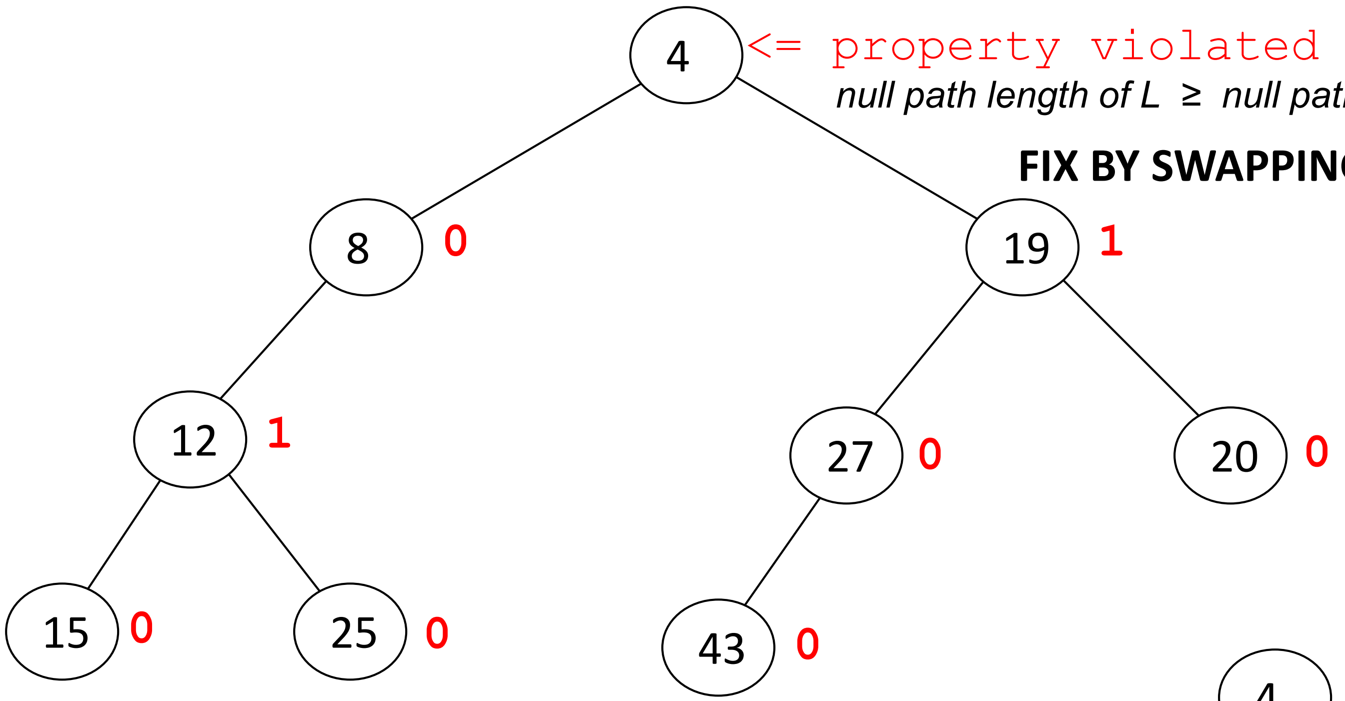
node	4	8	19	12	15	25	27	20	43
npl	1	0	1	1	0	0	0	0	0

Leftist heap: most nodes on left

- A Leftist (min)Heap is a binary tree that satisfies the following conditions.
- If X is a node and L and R are its left and right children, then:
  1. X.value ≤ L.value
  2. X.value ≤ R.value
  3. null path length of L ≥ null path length of R



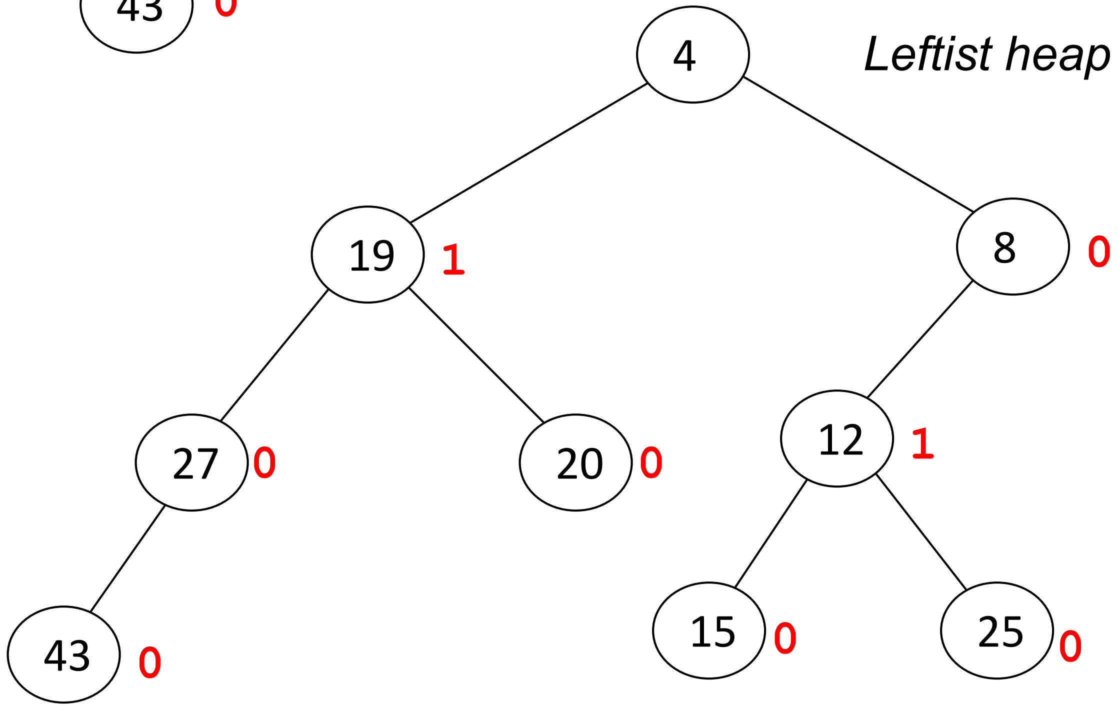
- Observations:
- The rightmost null path is the shortest
- Every subtree in a leftist tree is leftist



<= property violated

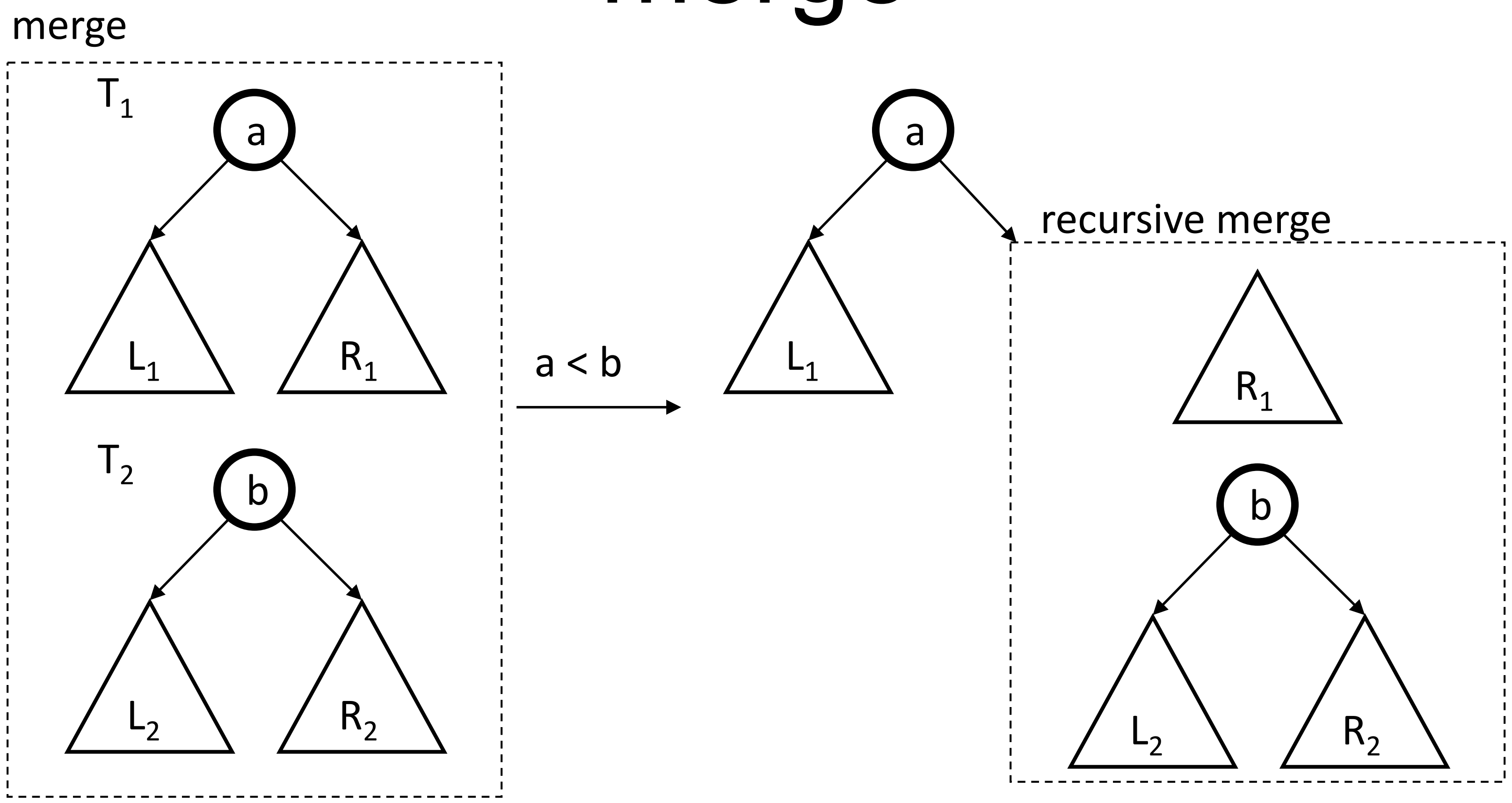
null path length of L ≥ null path length of R

FIX BY SWAPPING CHILDREN SUBTREES



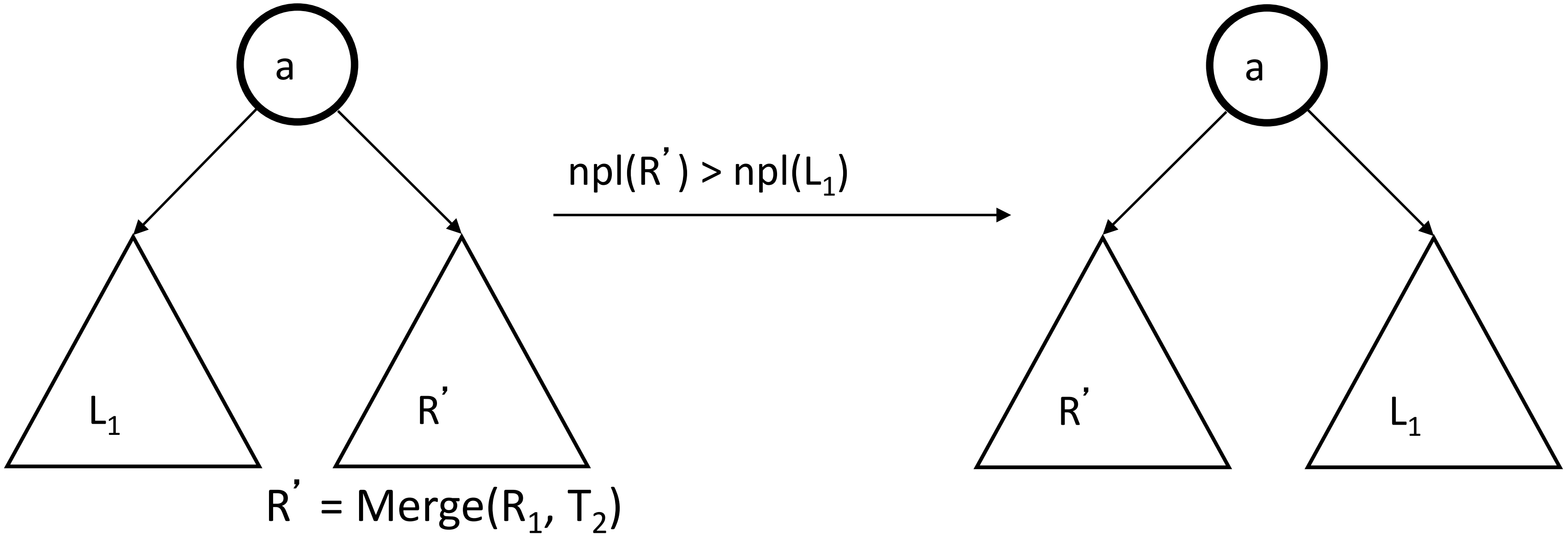
Leftist heap

merge

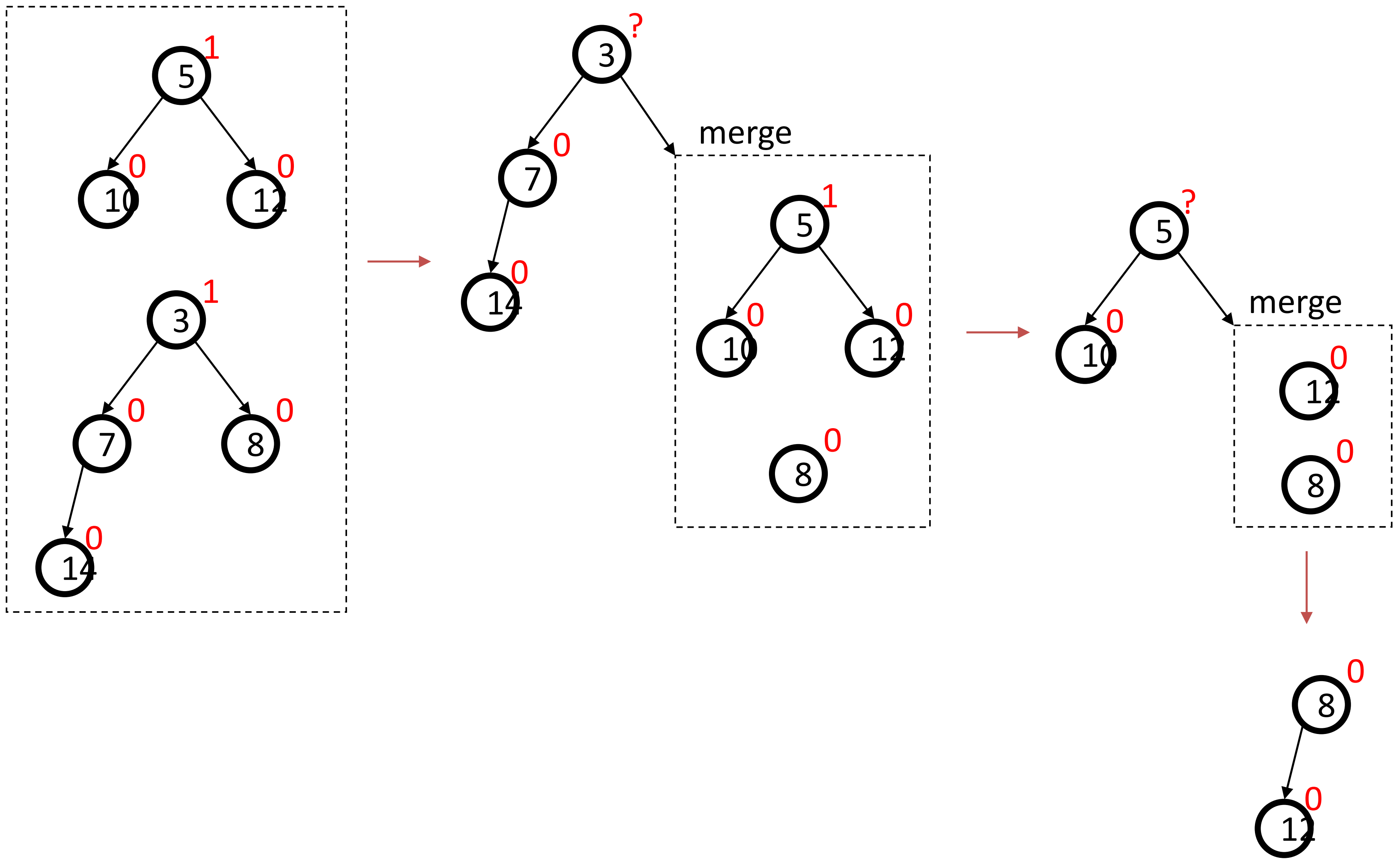


- Put the smaller root as the new root
- Hang its left subtree on the left.
- Recursively merge its right subtree and the other tree.

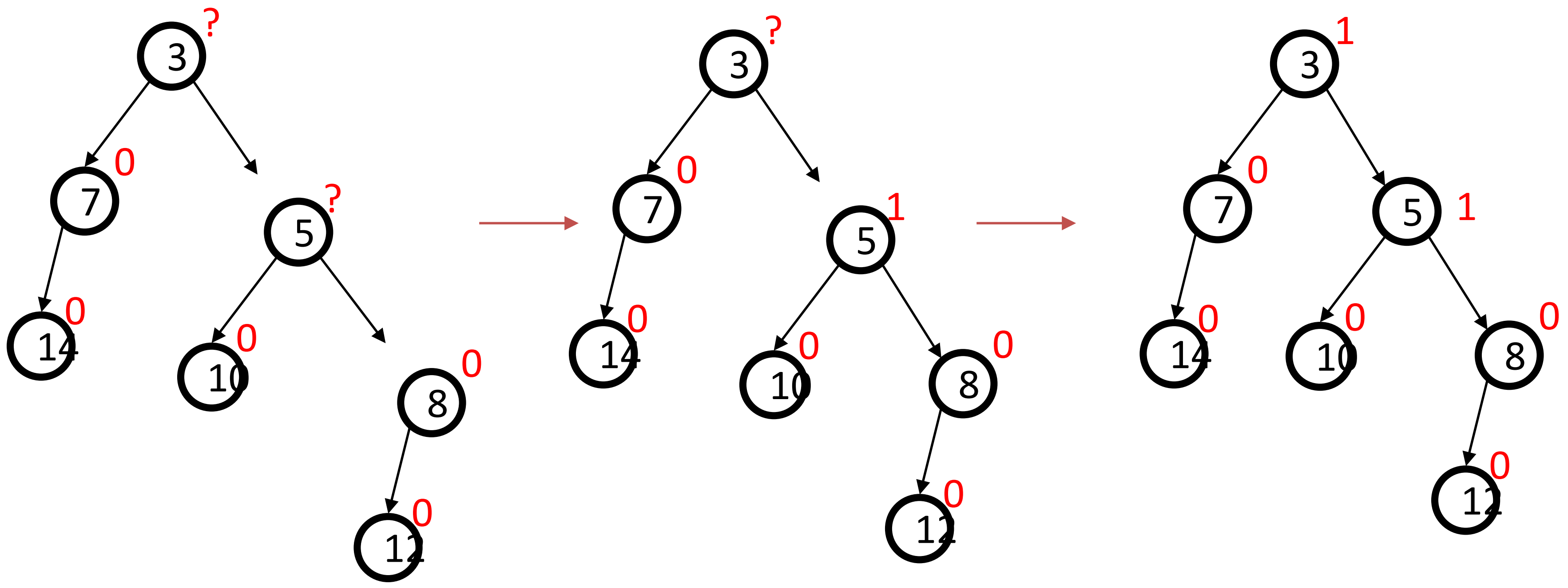
merge (cnt'd)

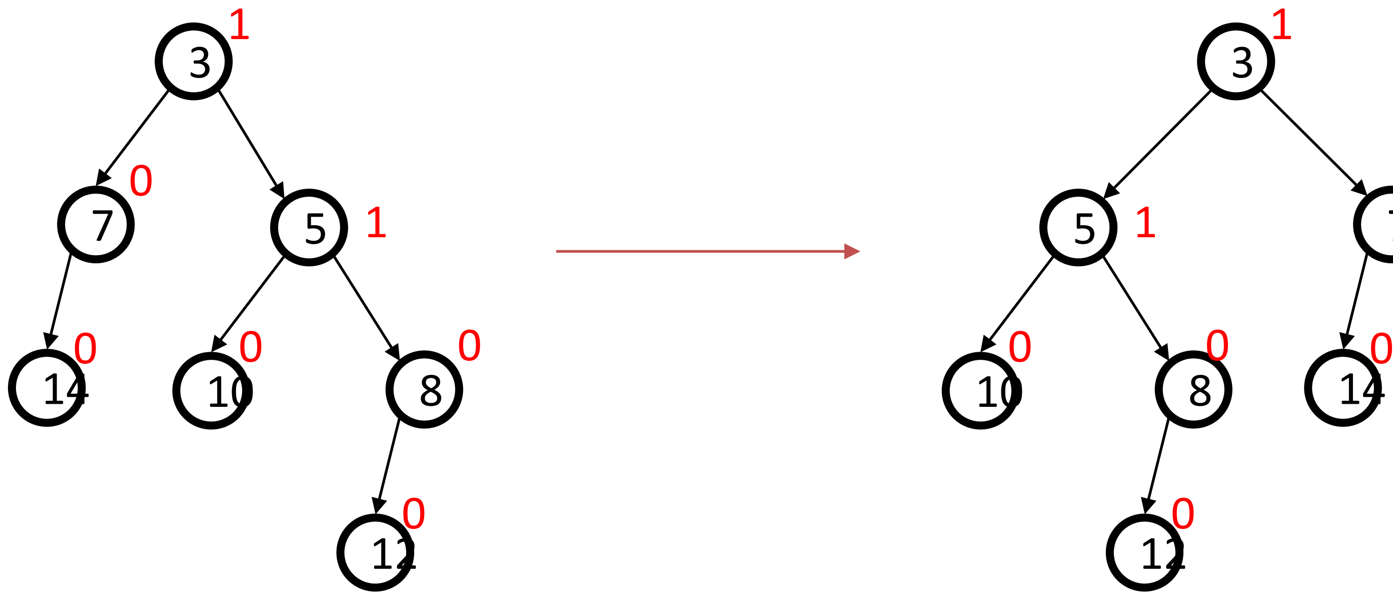


example

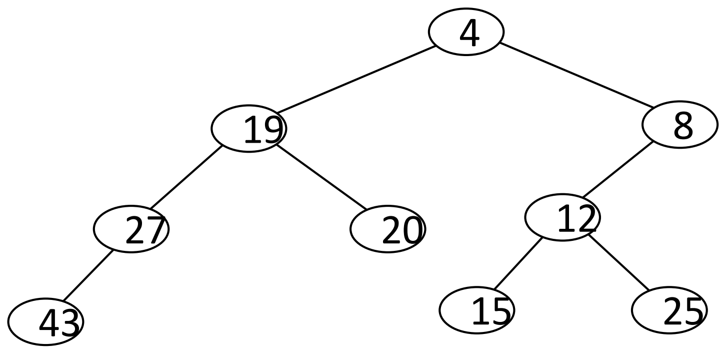


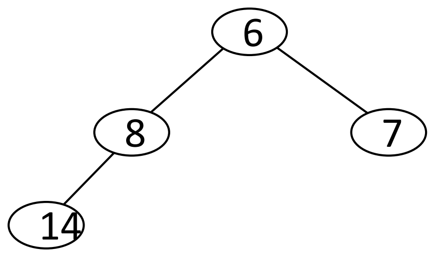
5
1
10
0
12
0
3
1
7
0
8
0
14
0
3
?
7
0
14
0
merge
5
1
10
0
12
0
8
0
5
?
10
0
merge
12
0
8
0
8
0
12
0

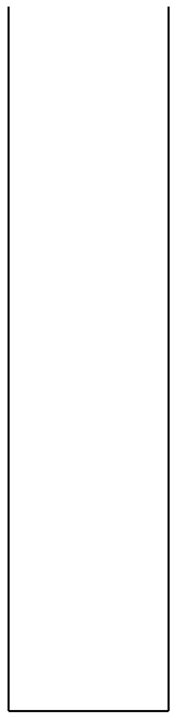

3 ?
7 0
5 ?
14 0
10 0
8 0
12 0
3 ?
7 0
5 1
14 0
10 0
8 0
12 0
3 1
7 0
5 1
14 0
10 0
8 0
12 0


3
1
7
0
5
1
14
0
10
0
8
0
12
0
3
1
5
1
7
0
10
0
8
0
14
0
12
0

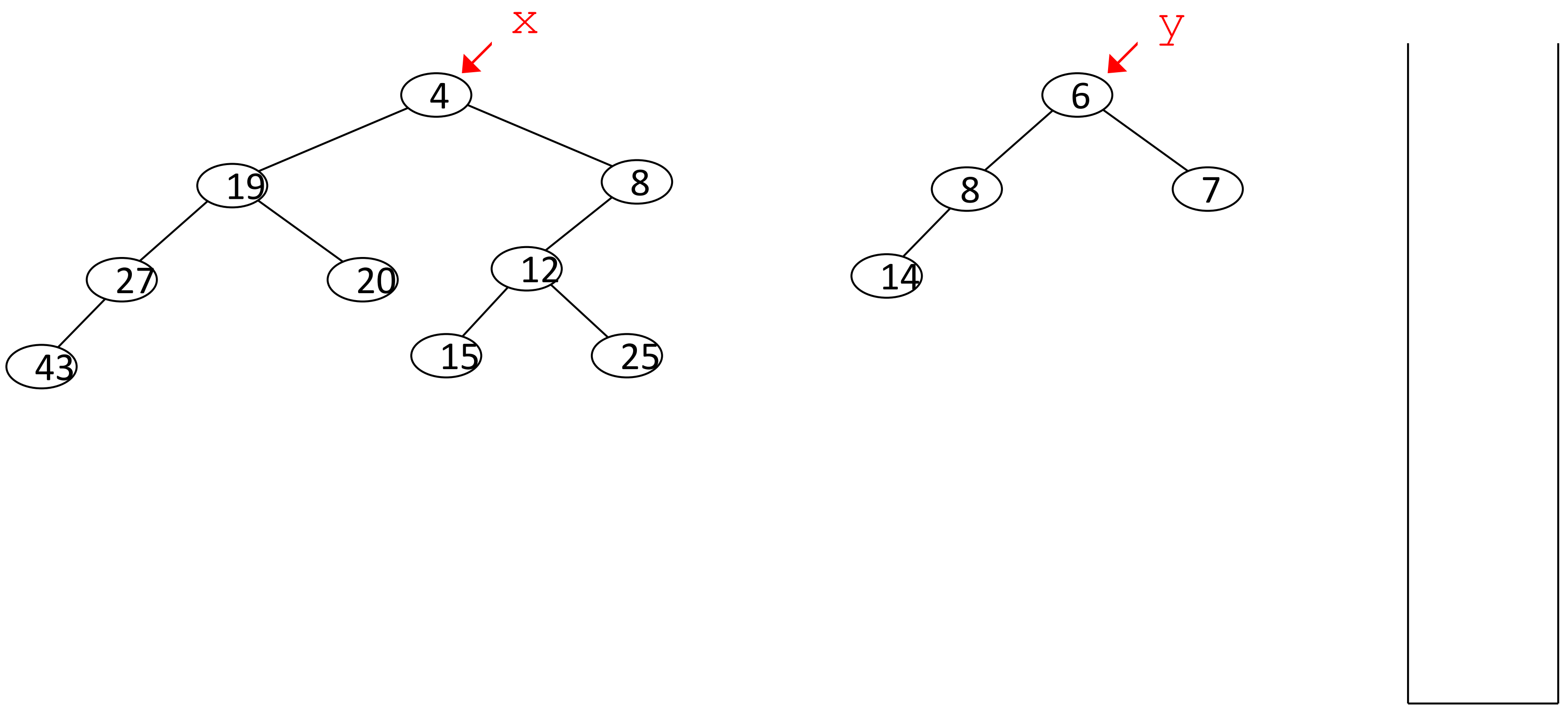
implementation with stack





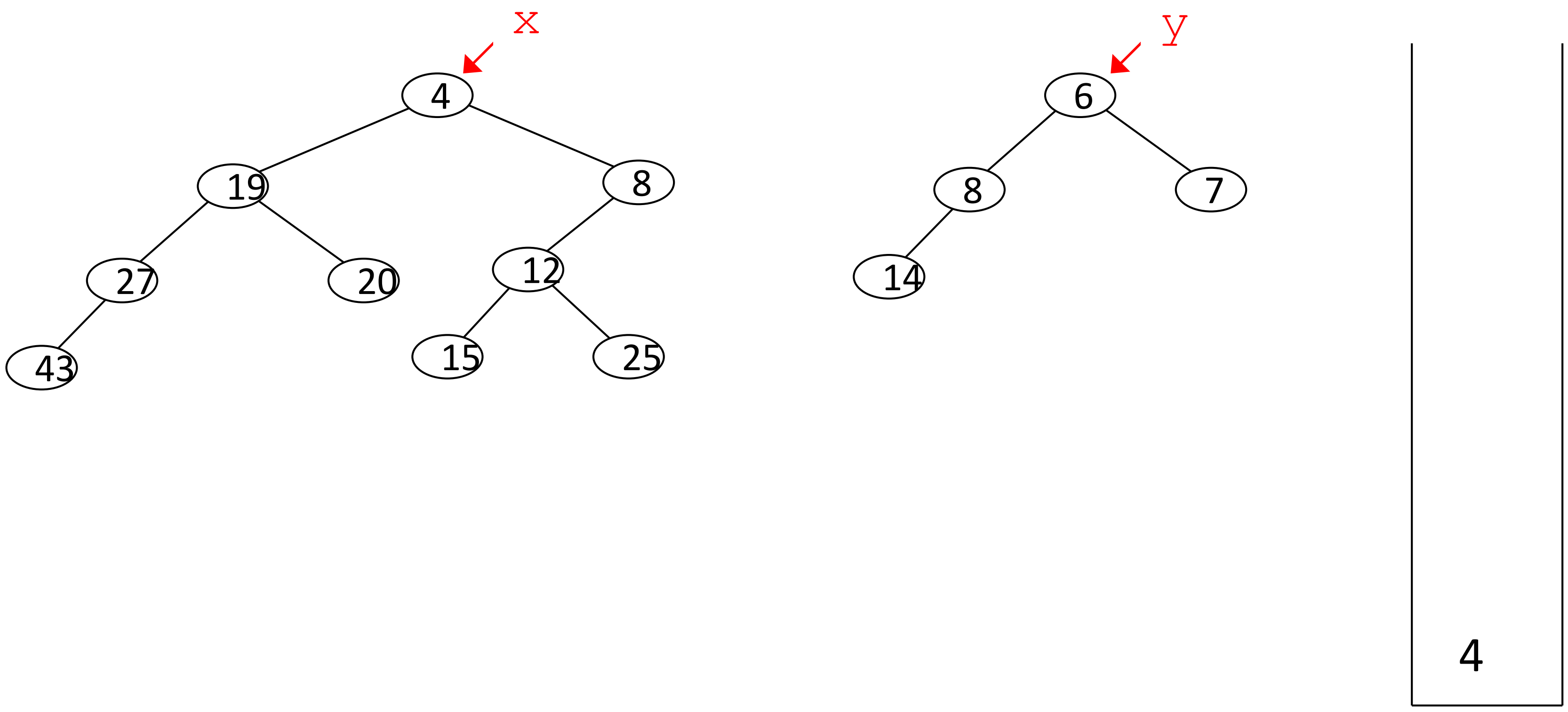


initiate stack

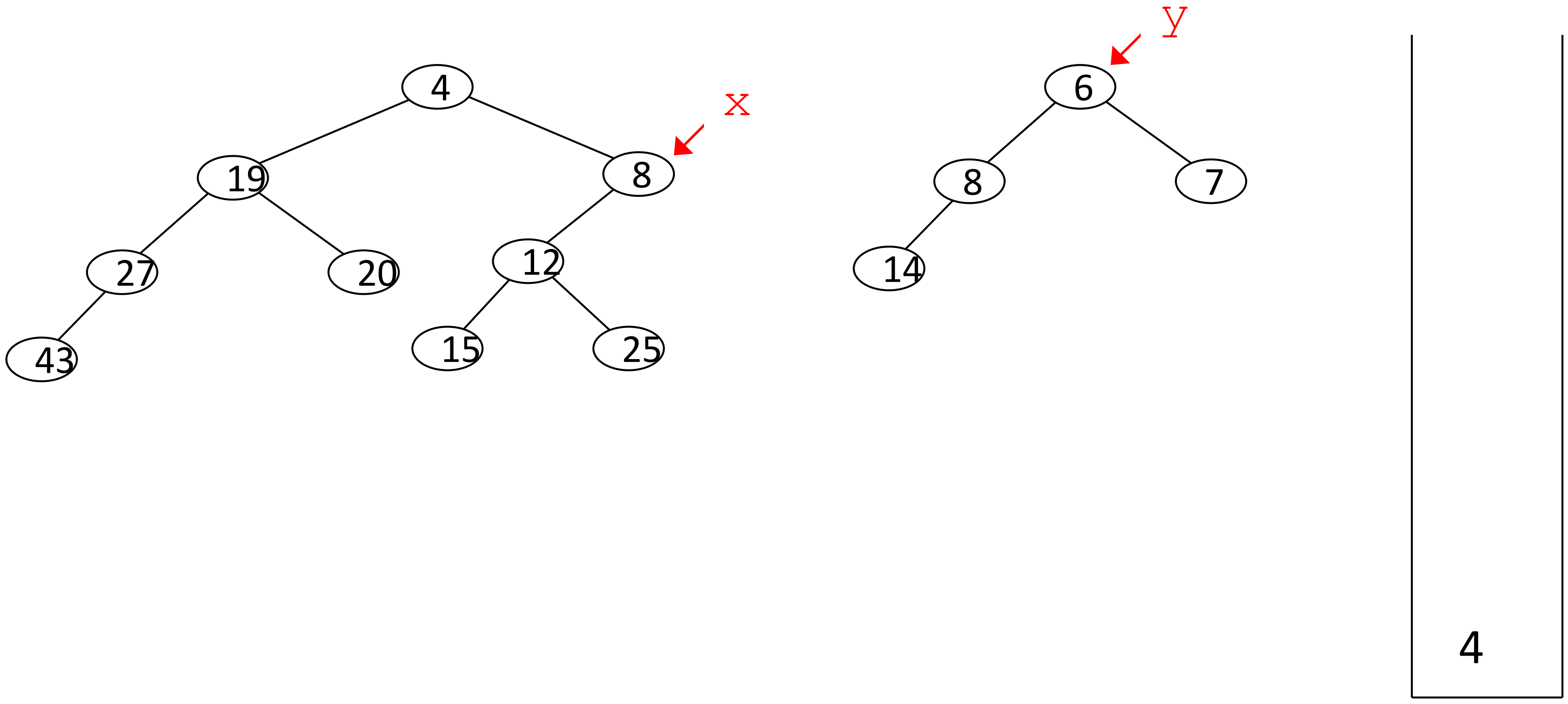


Compare root nodes

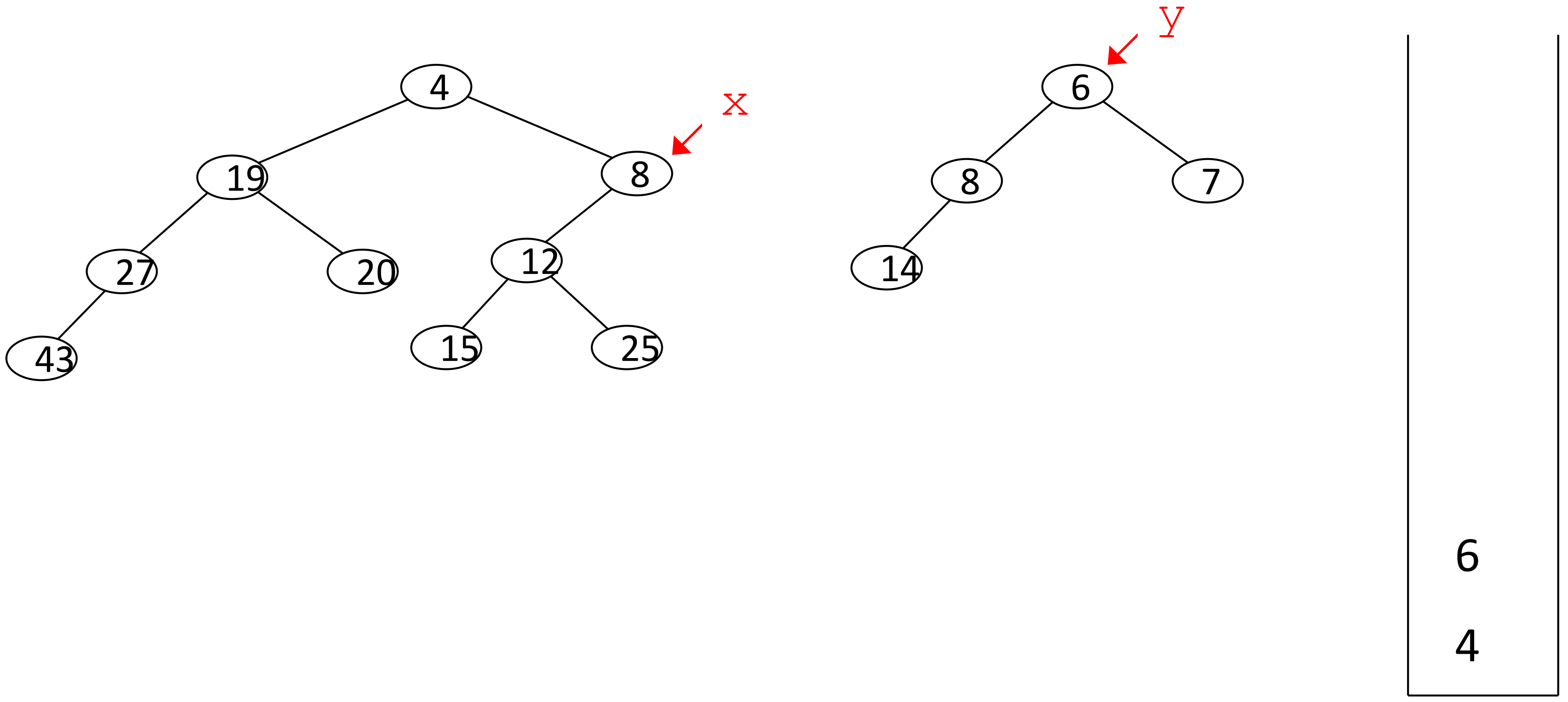
`merge(x,y)`



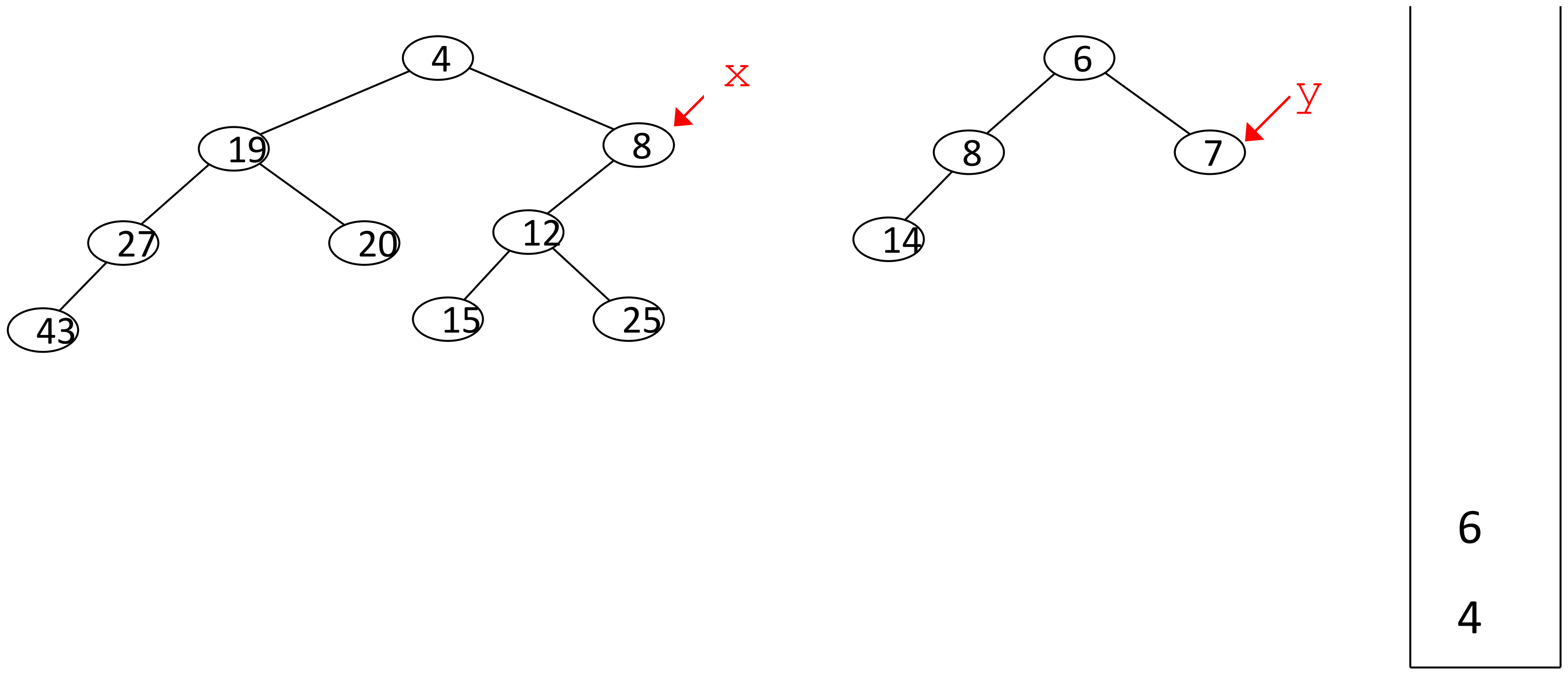
Remember smaller value



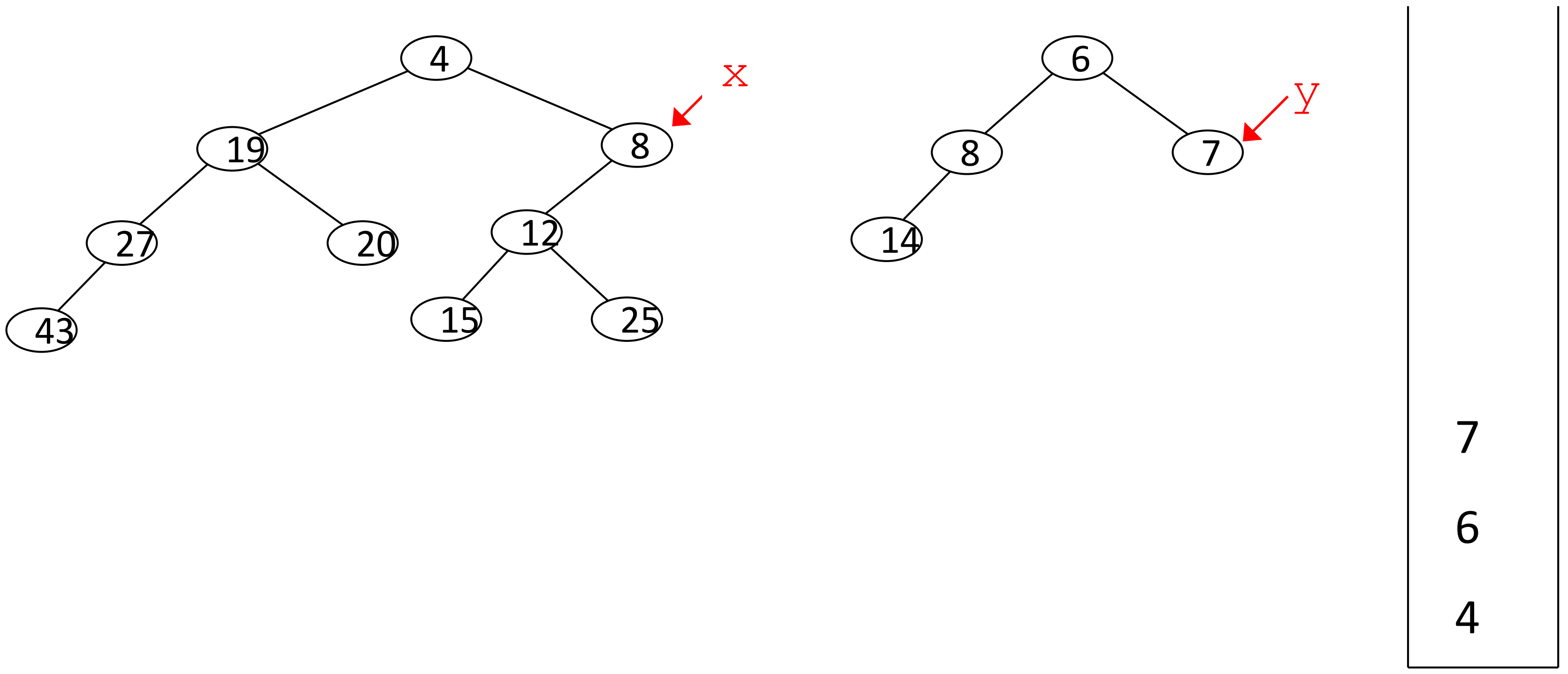
Repeat the process with the right child of the smaller value



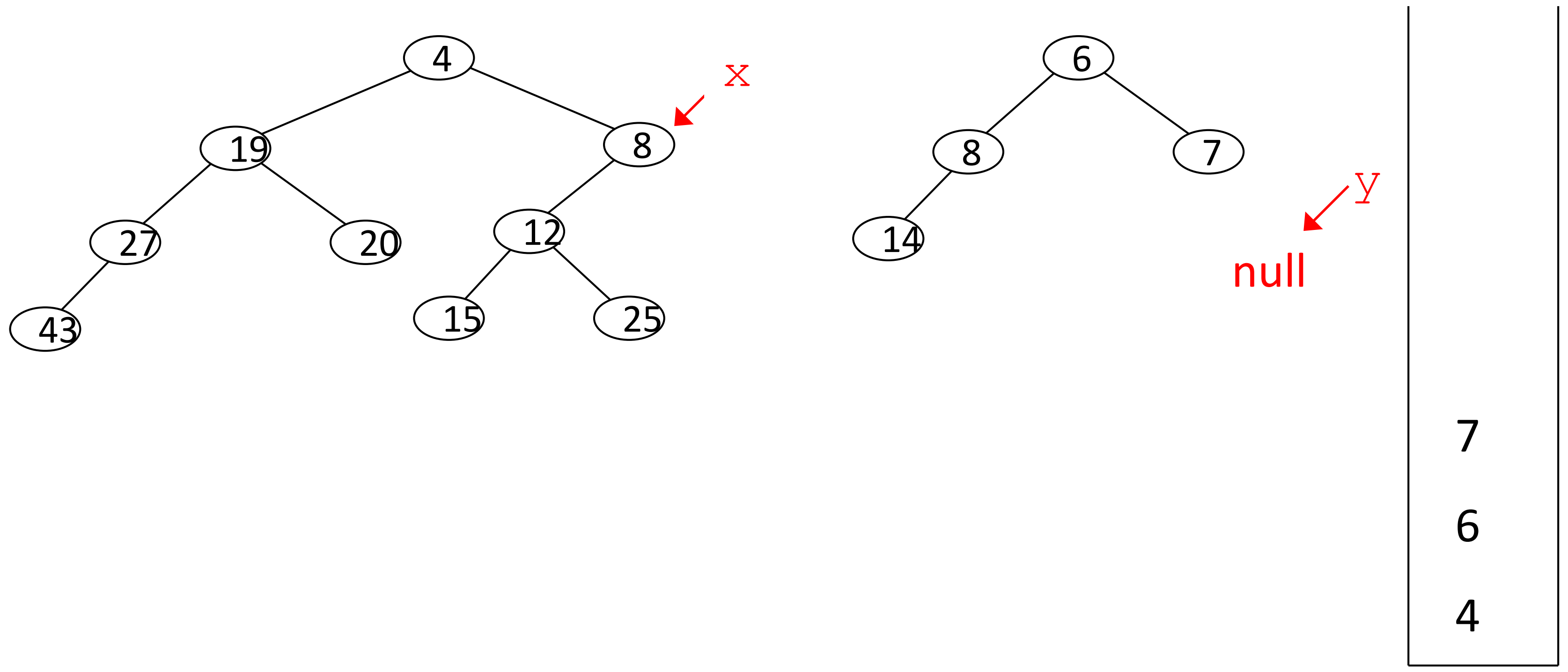
Remember smaller value



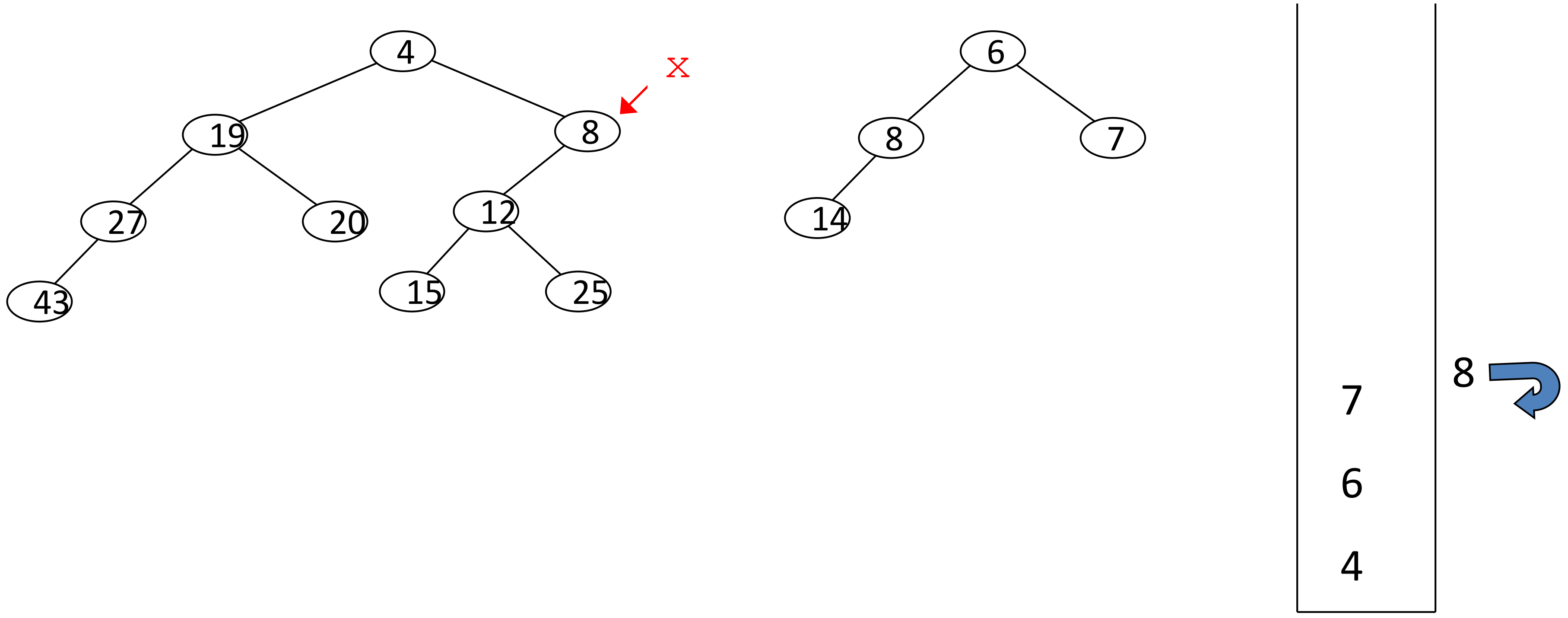
Repeat the process with the right child of the smaller value



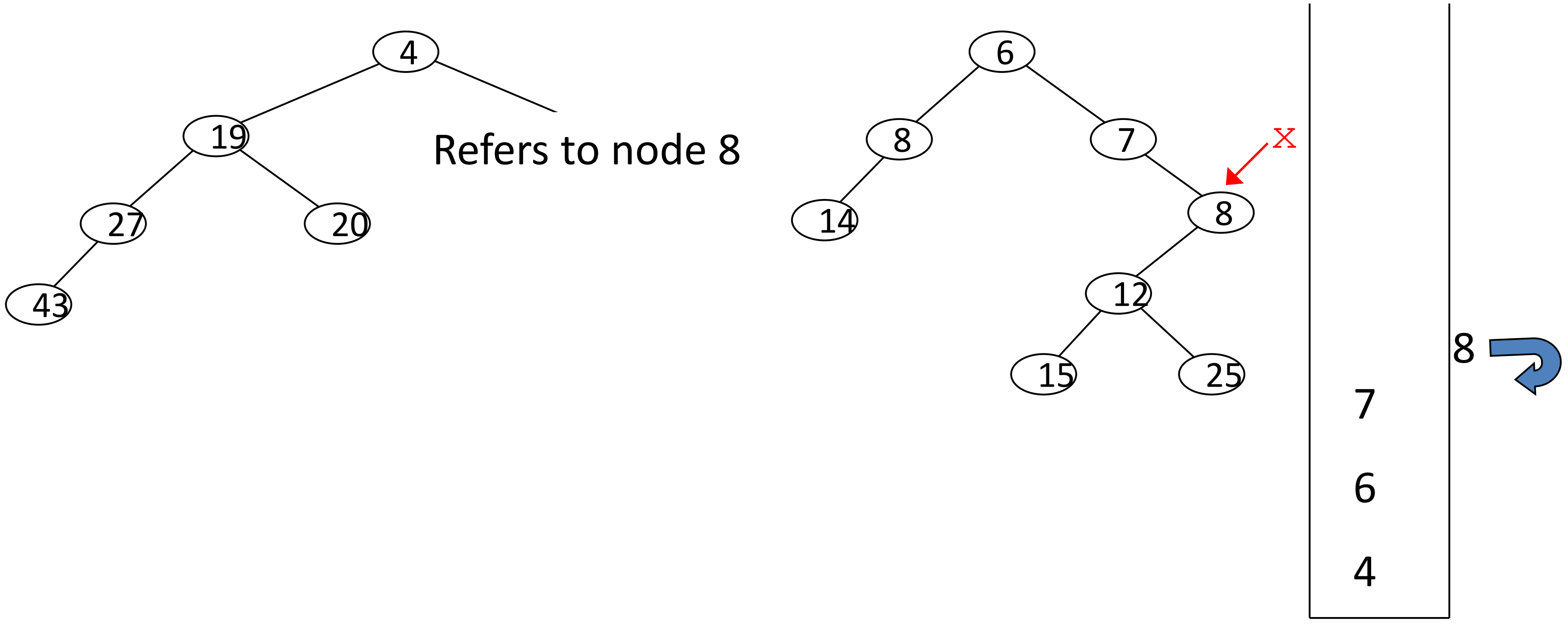
Remember smaller value



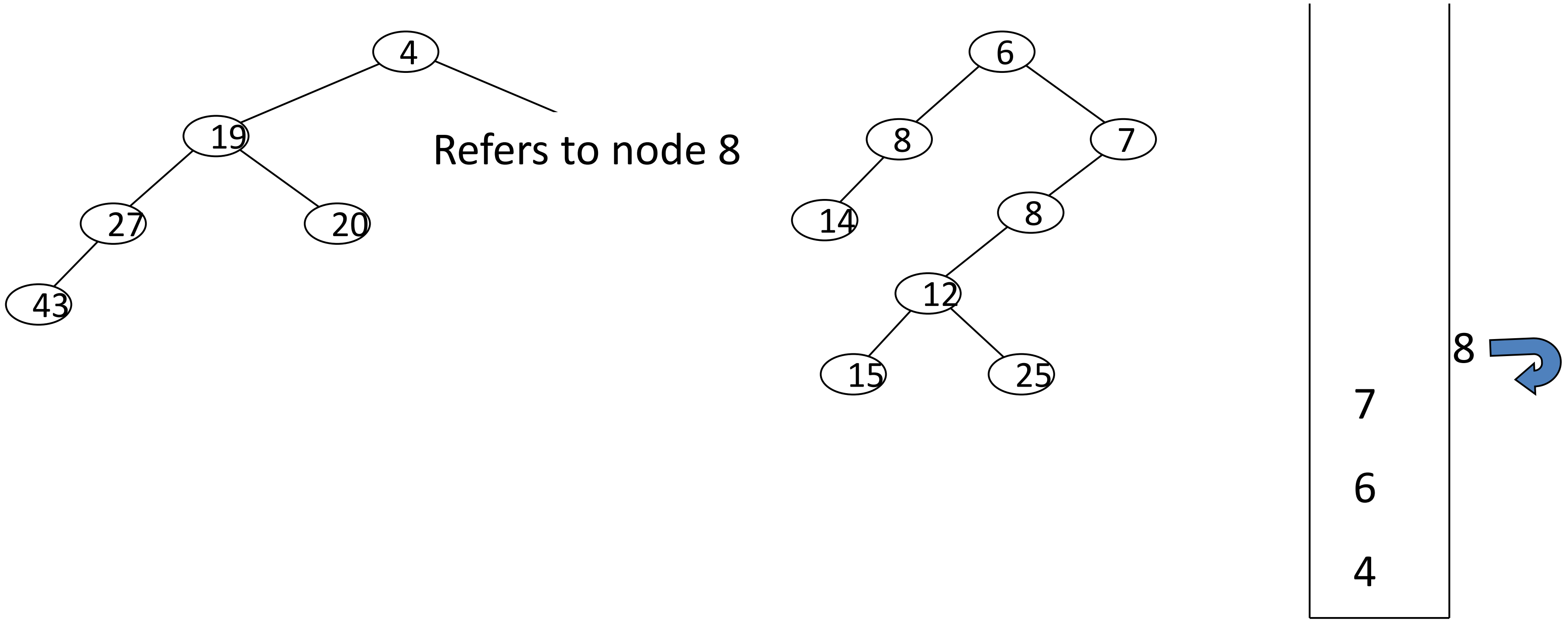
Repeat the process with the right child of the smaller value



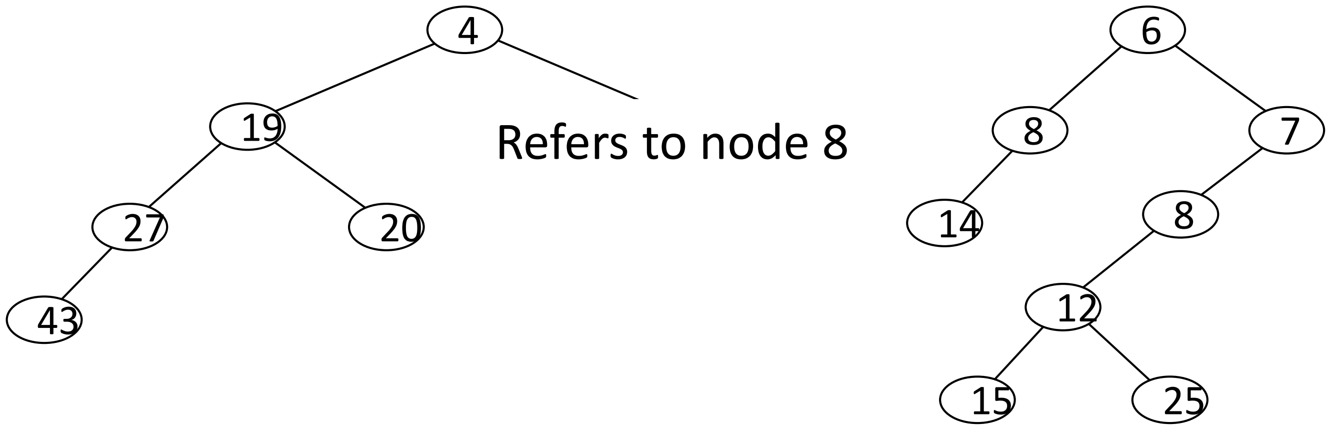
Because one of the arguments is null, return the other argument

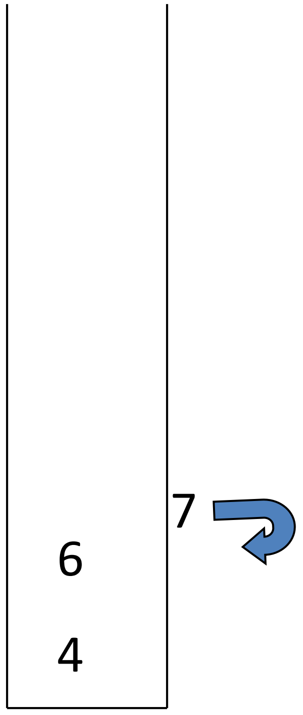


Make 8 the right child of 7

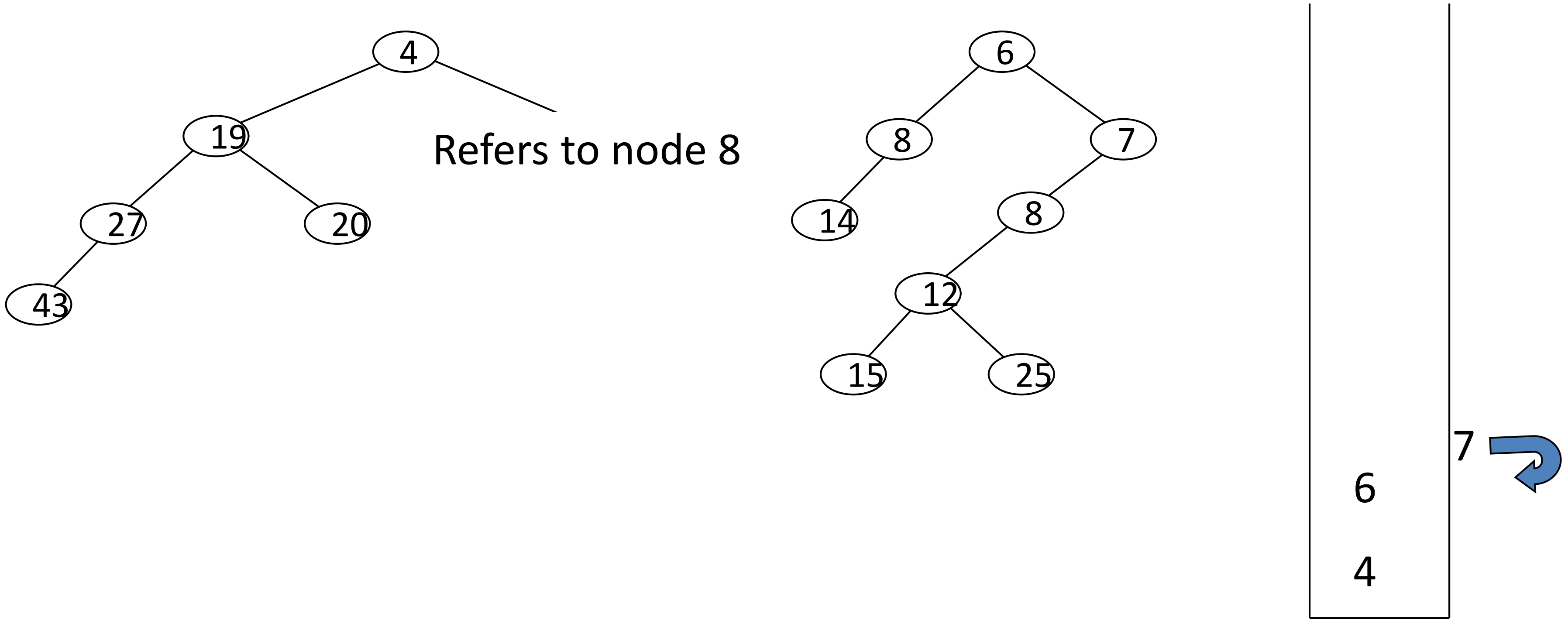


Make 7 leftist (by swapping children)

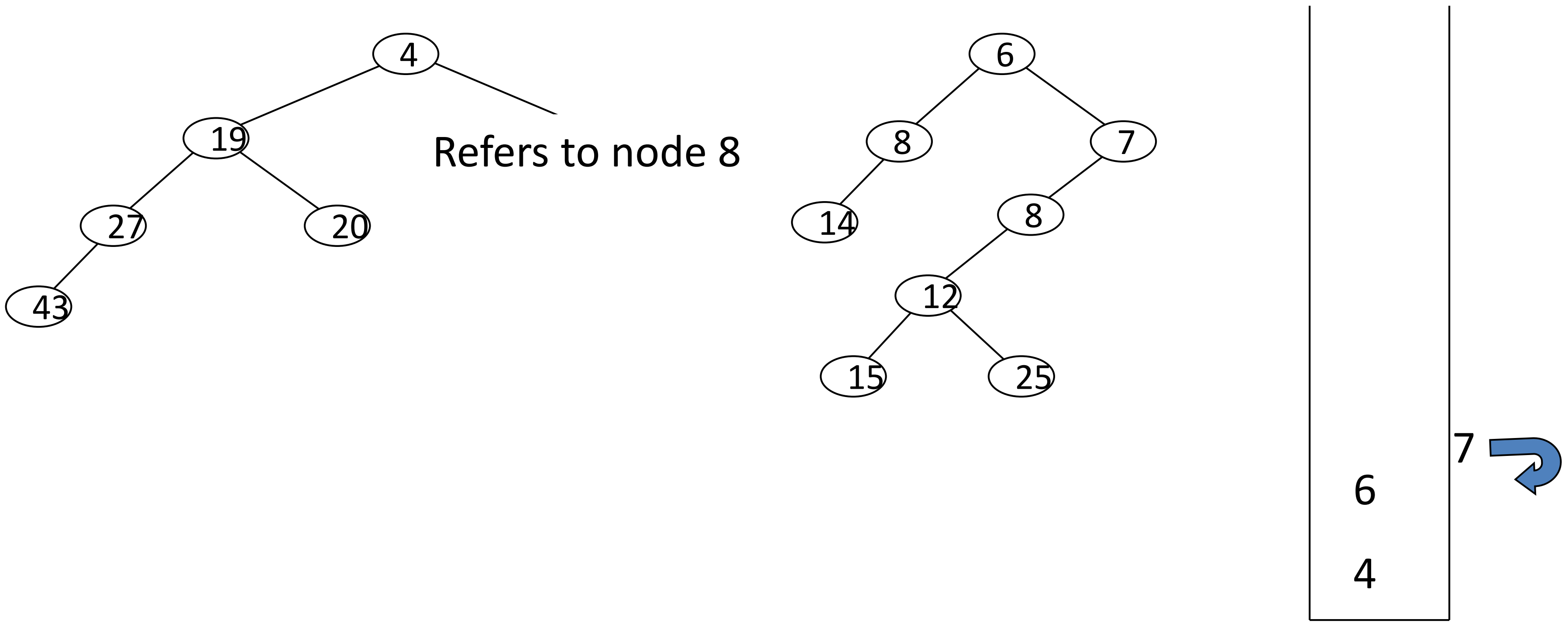




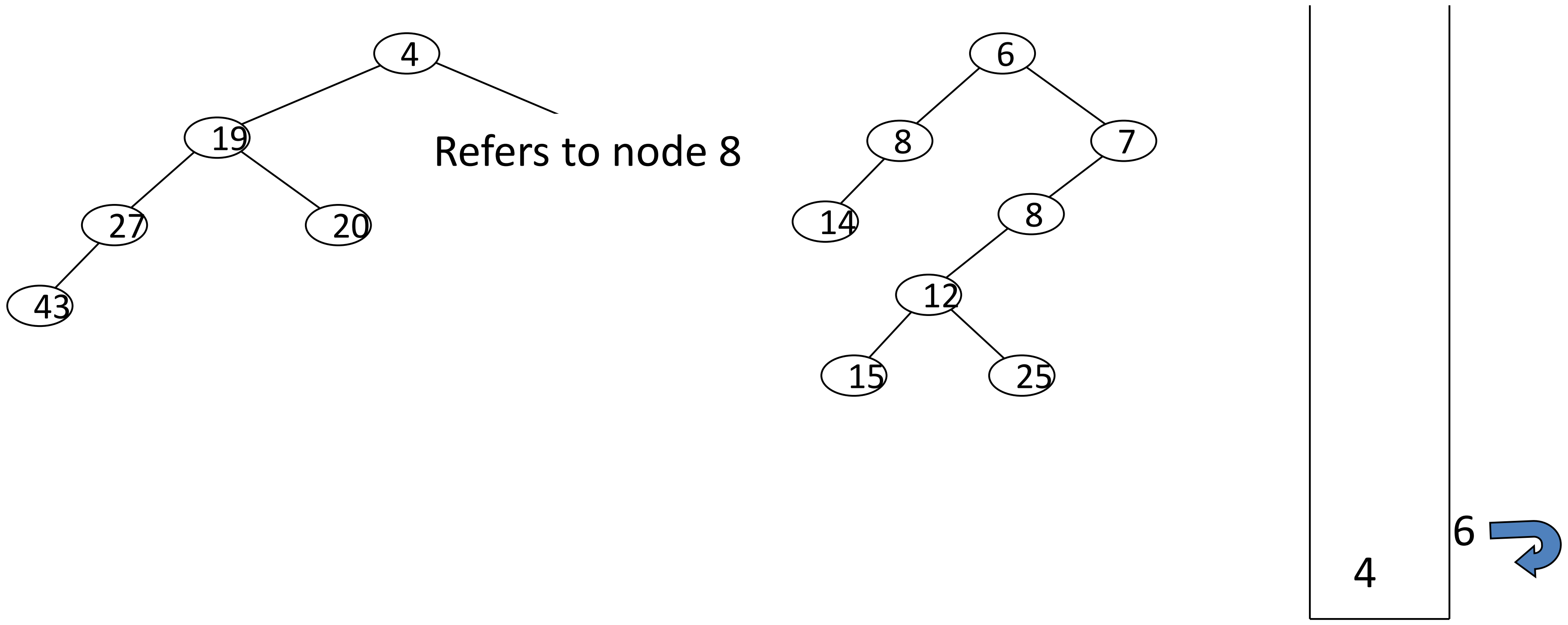
Return node 7



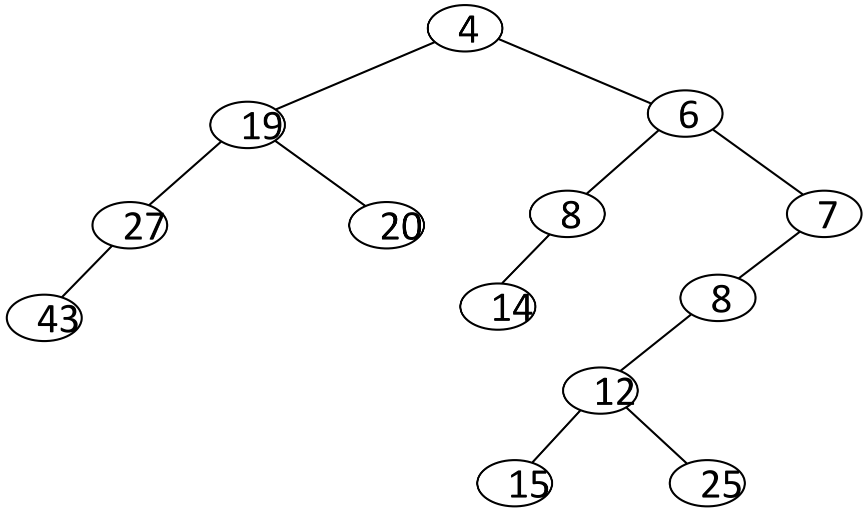
Make 7 the right child of 6 (which it already is)

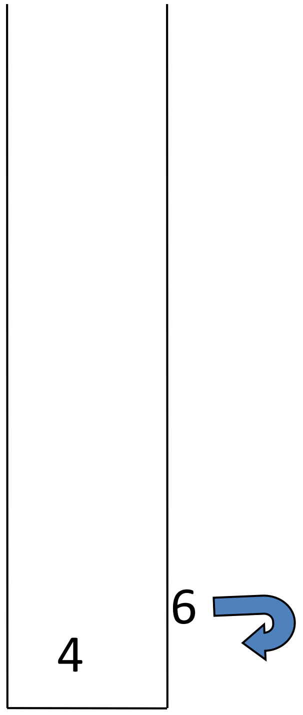


Make 6 leftist (it already is)

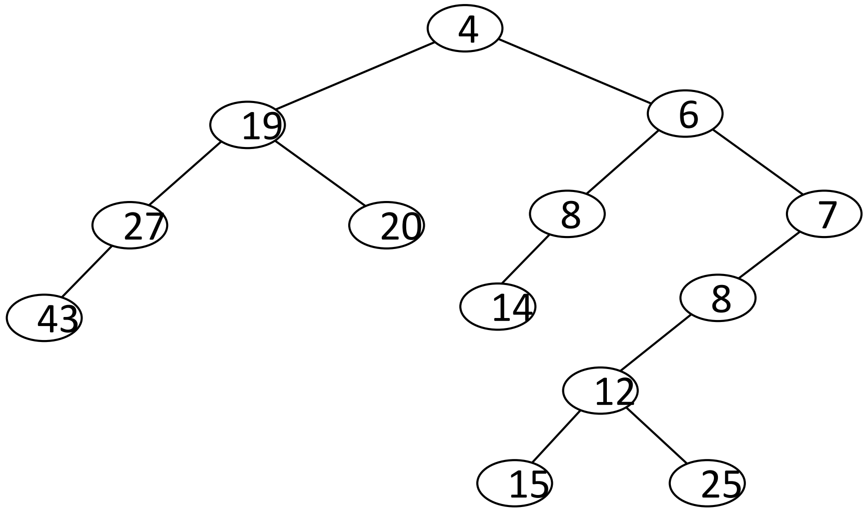


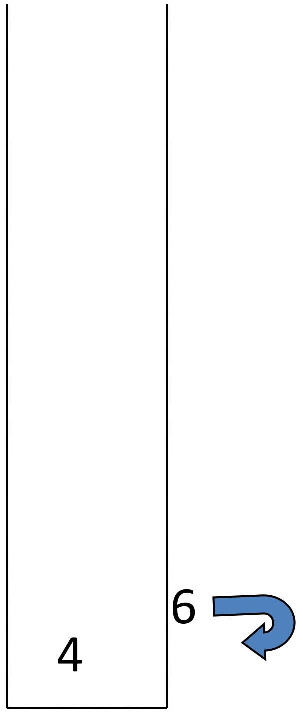
Return node 6



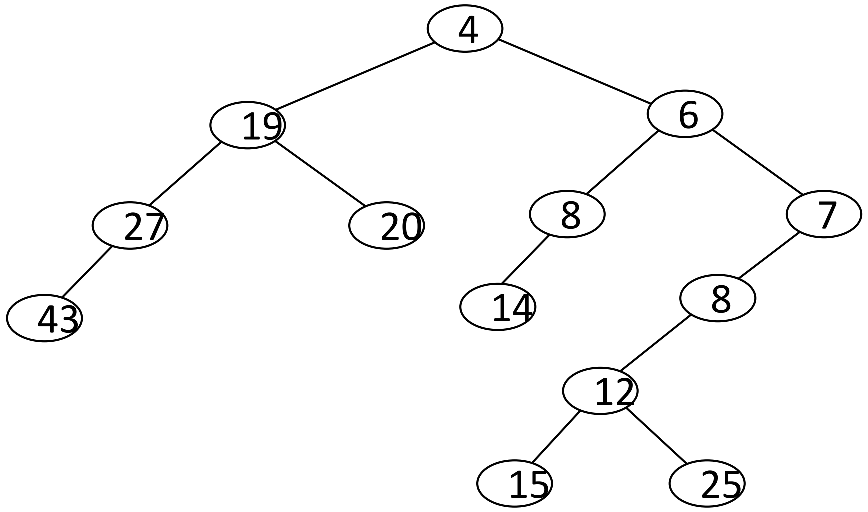


Make 6 the right child of 4

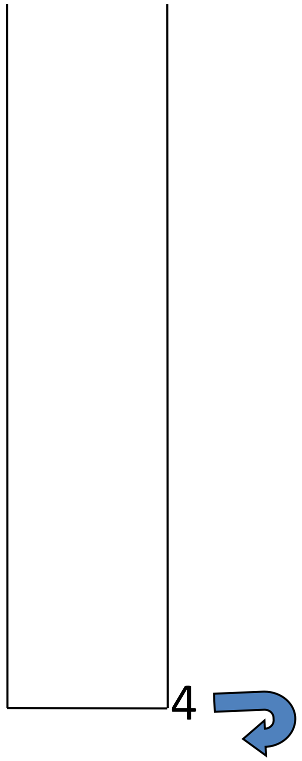




Make 4 leftist (it already is)

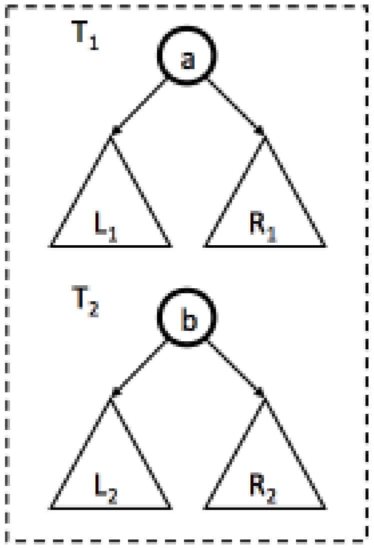


- Verify that the tree is heap
- Verify that the heap is leftist
- Total cost of merge: O(logN)



Return node 4

The schema



- remember smaller value (put in stack)
- when reach at comparison with null of a right subtree:
- Return value
- Make right child
- Make leftish