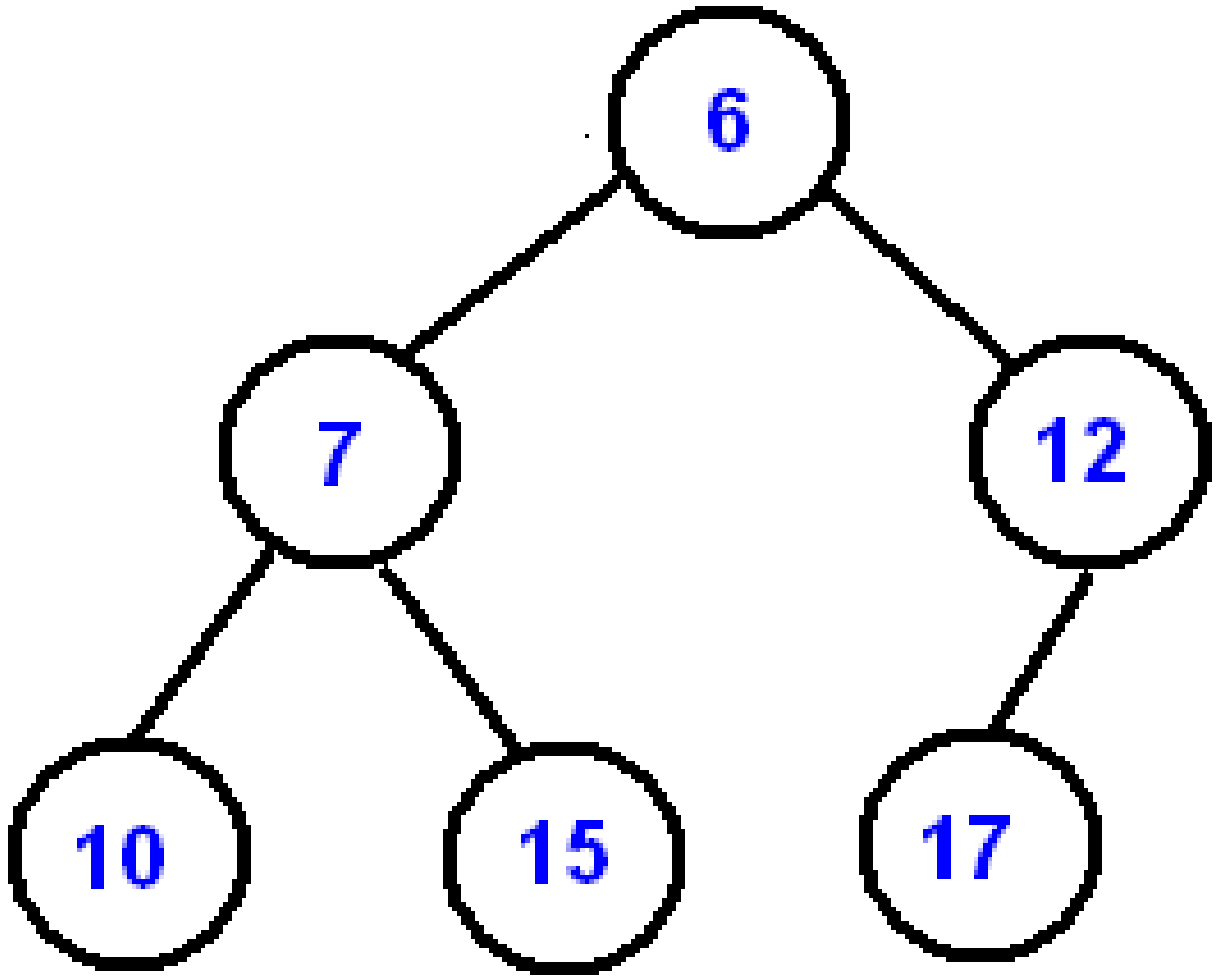
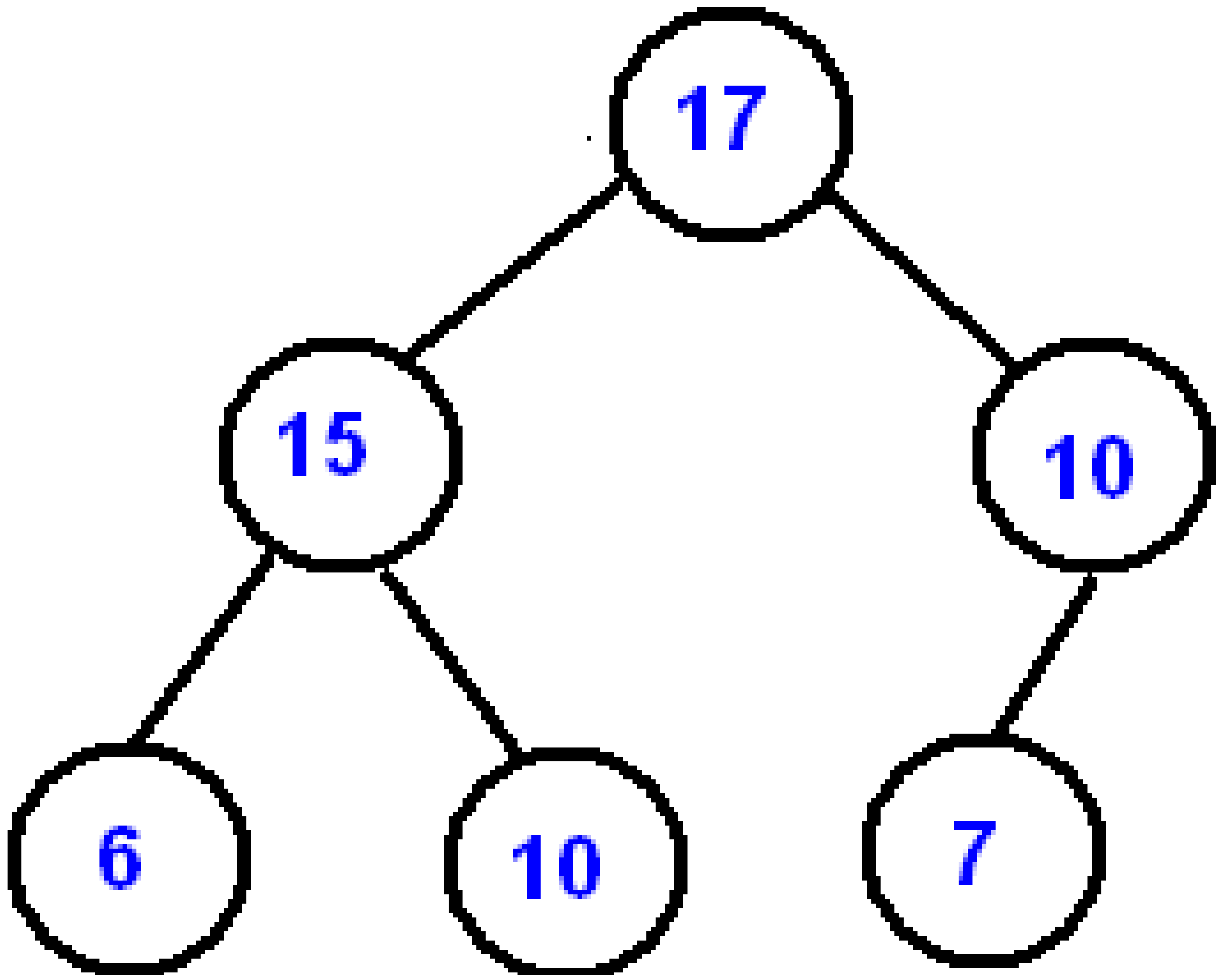
Binary heap

1. Structural property : Must be complete binary tree
2. Ordering property : Root must be minimum /max as compared to children.


6
7
12
10
15
17


17
15
10
6
10
7

1. **if we want to merge both binary heap :**

Time complexity = (m+n) + (m+n) *log(m+n)

To solve this higher time complexity of binary heap (disadvantage) we use leftist heap.

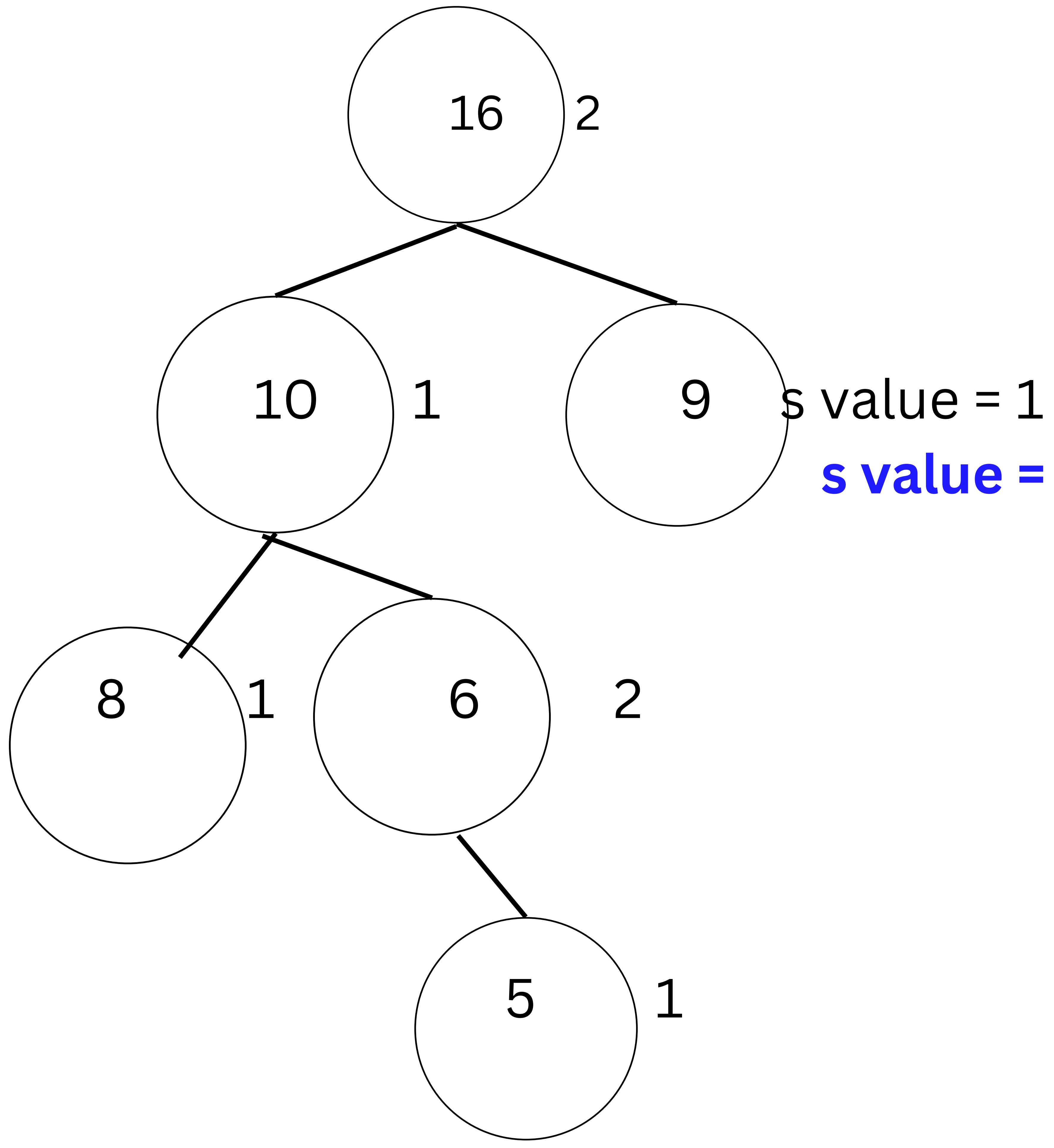
Properties of leftist heap :

A. S value : length of path(shortest one) from any node to leaf/external node.

s value (left child) >= s value(right child)

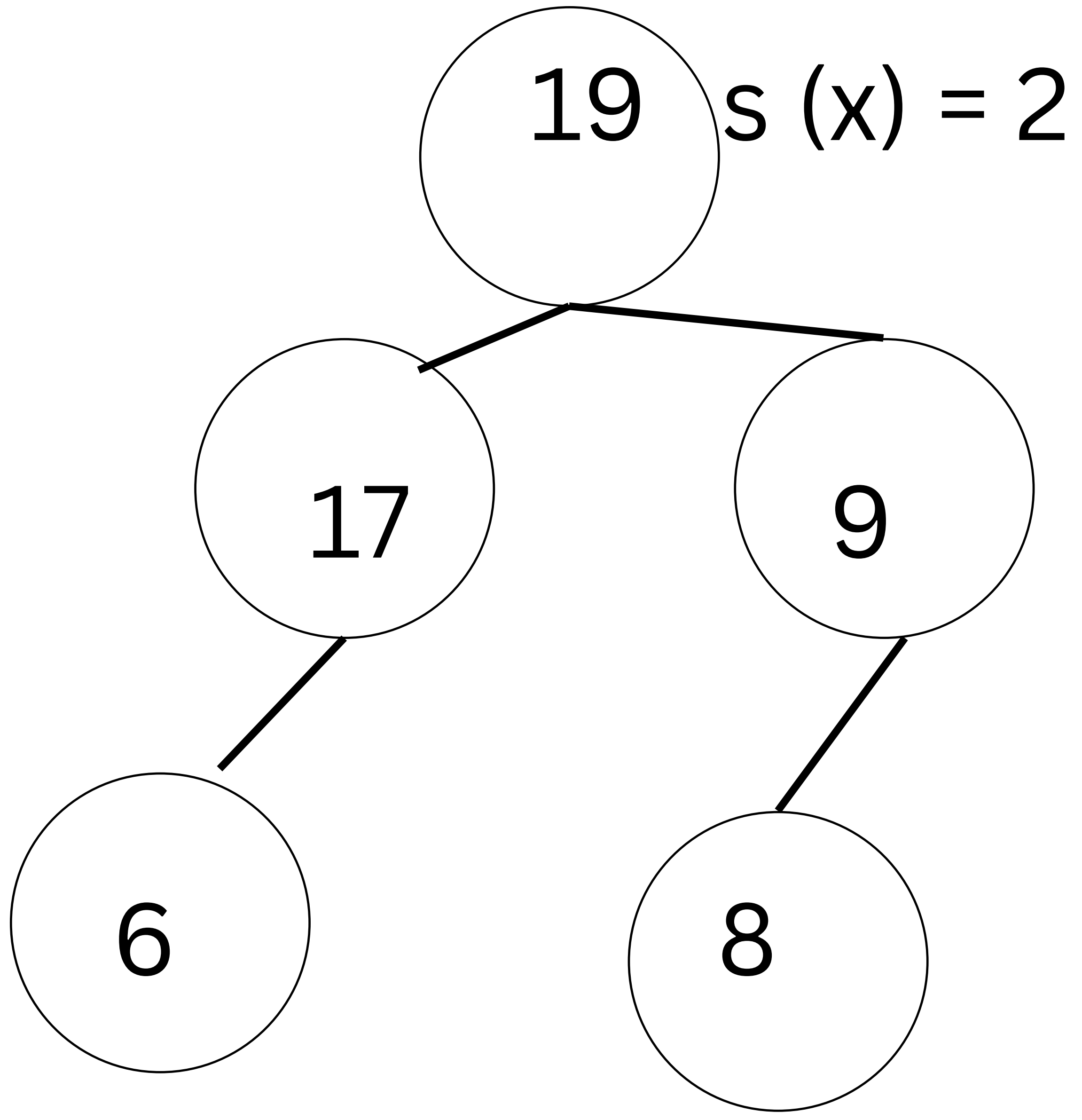
B. **ORDERING PROPERTY must be maintained.**

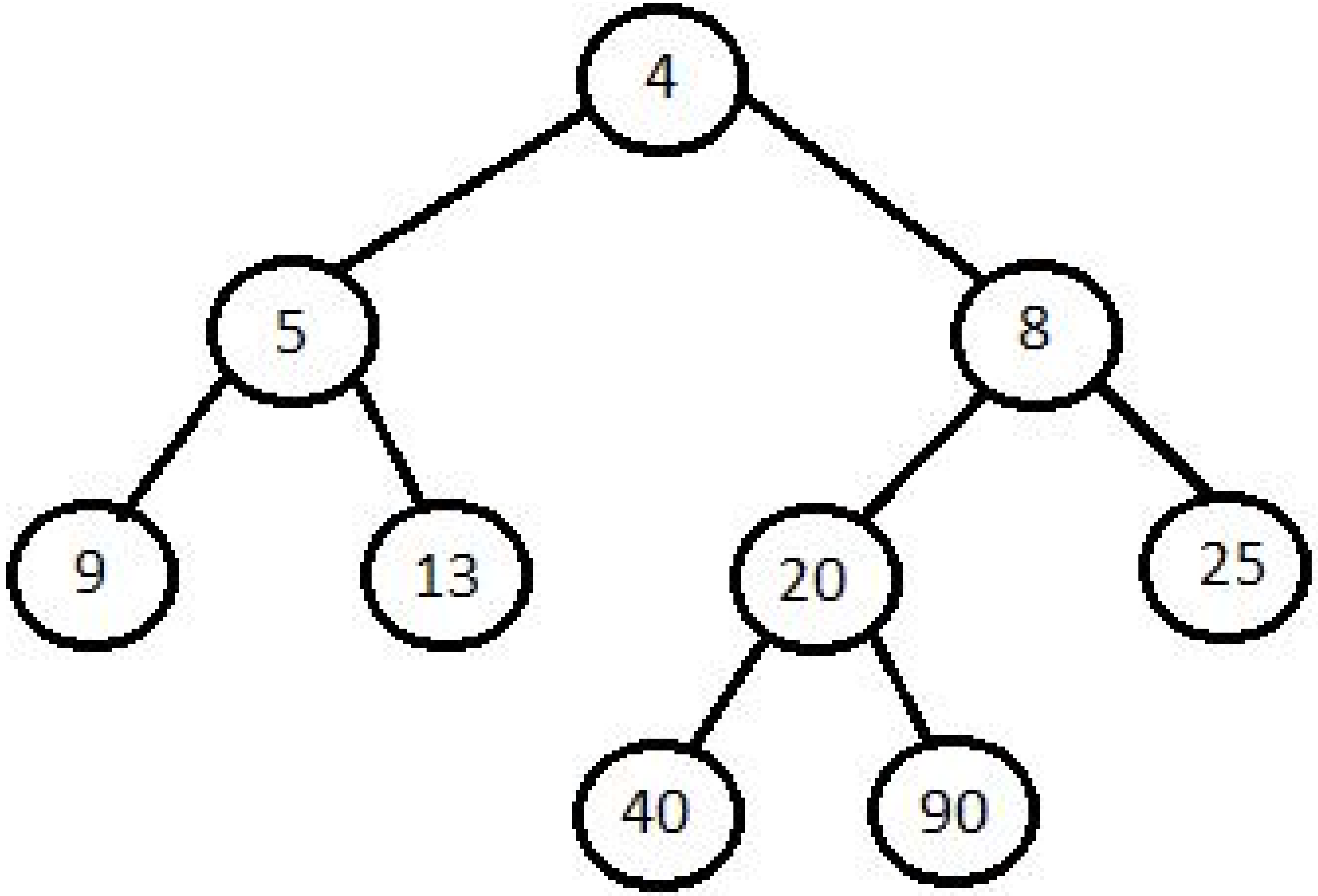
c. leftist tree must be min/max heap.



s value = min(s value(right)+s value(left))+1

1. leftist heap must be left heavy.
2. no of internal nodes in the subtree with root x : $n >= 2^{s(x)} - 1$
3. if subtree with root x has n nodes then s(x) is almost log(n+1).

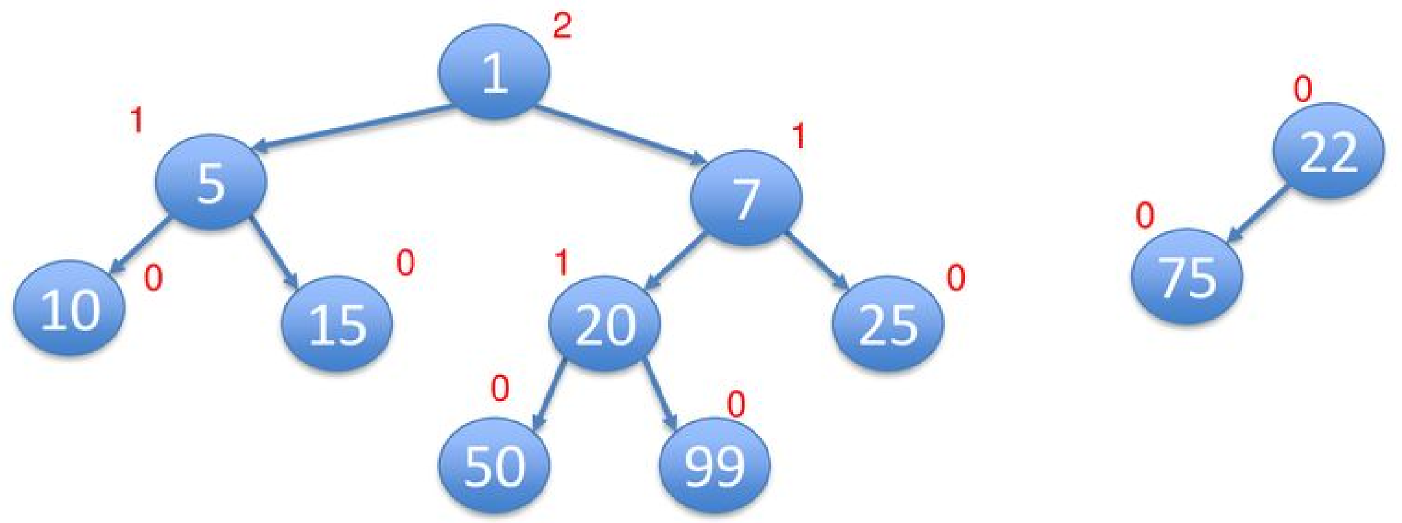




not a leftist one

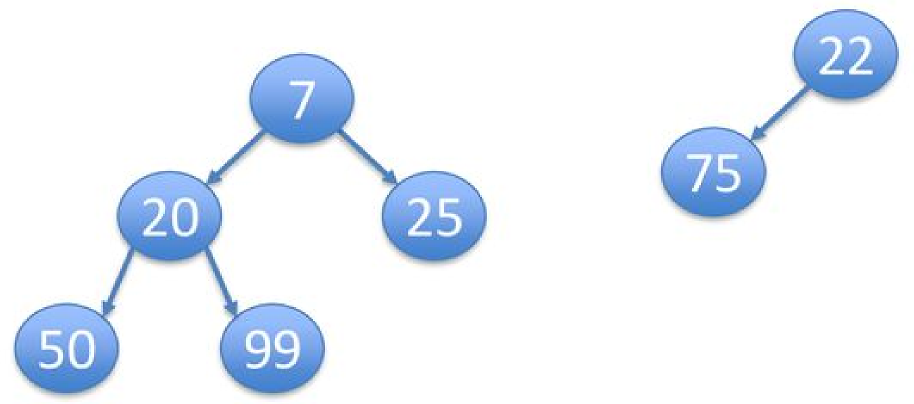
```
Merge(A,B)
{
  if A = NULL
    return B;
  if B = NULL
    return A;

  if key(A) > key(B)
    swap A, B
  right(A) = Merge(right(A),B)
  if dist(right(A)) > dist(left(A))
    swap right(A), left(A)
  if right(A) = NULL
    dist(A) = 0
  else
    dist(A) = 1 + dist(right(A))
  return A
}
```

Merging Leftist Heaps Example



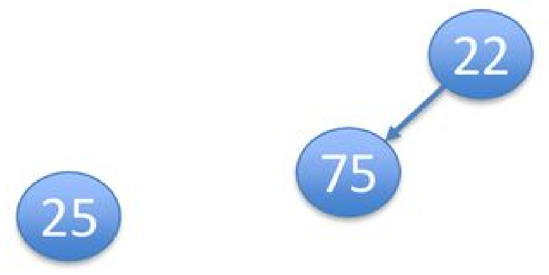
Where should we attempt to merge?

Merging Leftist Heaps Example



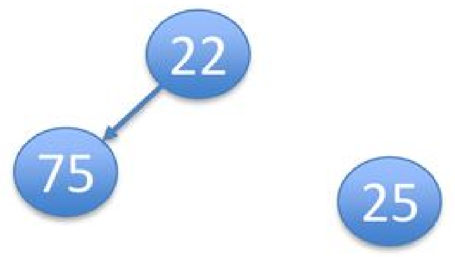
In the right sub-tree

Merging Leftist Heaps Example



All the way down to the right most node

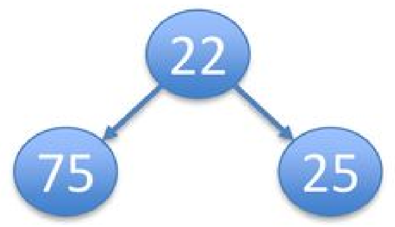
Merging Leftist Heaps Example



As there are two nodes in the right subtree, swap.

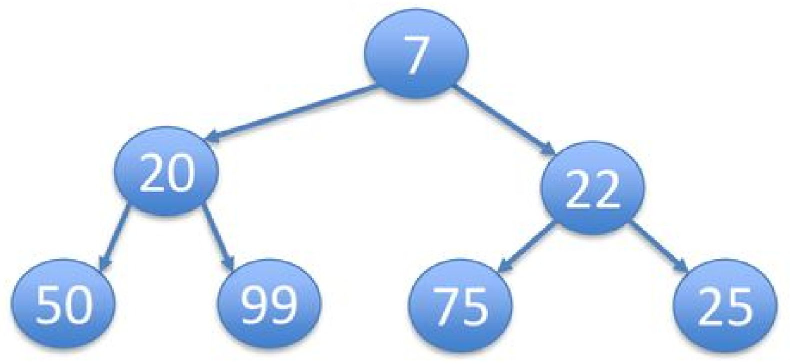
Important: We don't "split" a heap, so 22 must be the parent in this merge

Merging Leftist Heaps Example



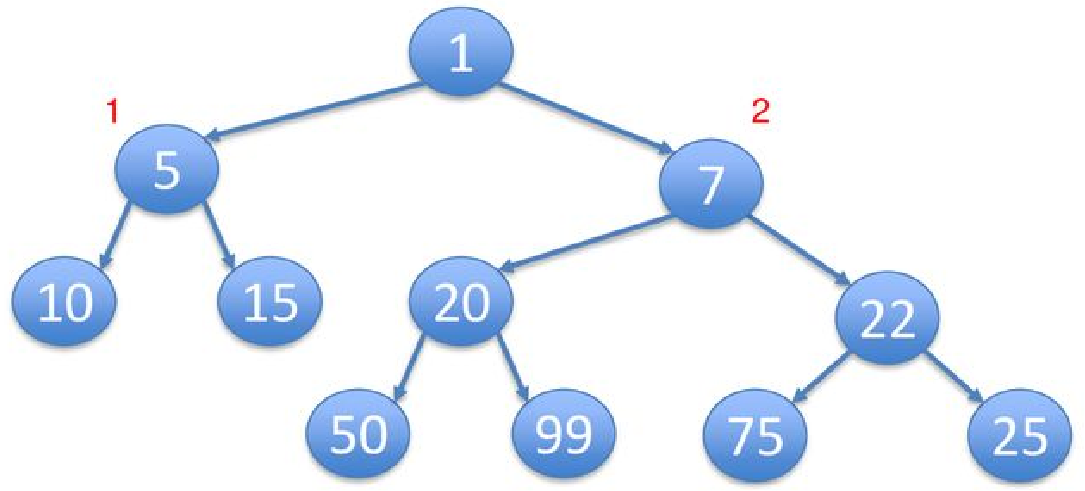
Merge two subtrees

Merging Leftist Heaps Example



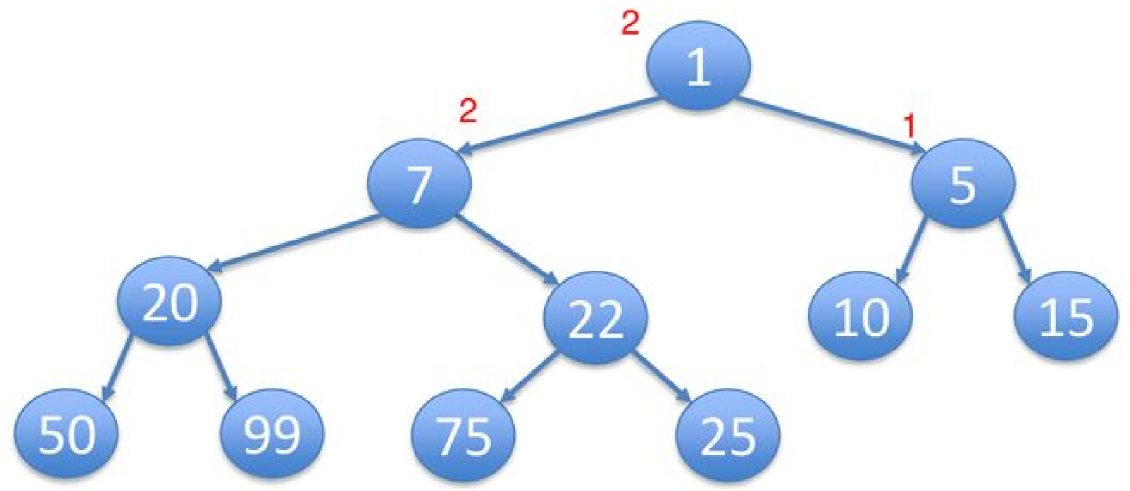
Next level of the tree

Merging Leftist Heaps Example



Right side of the tree has a npl of 2 so we need to swap

Merging Leftist Heaps Example



Now the highest npl is on the left.

Merging Leftist Heaps

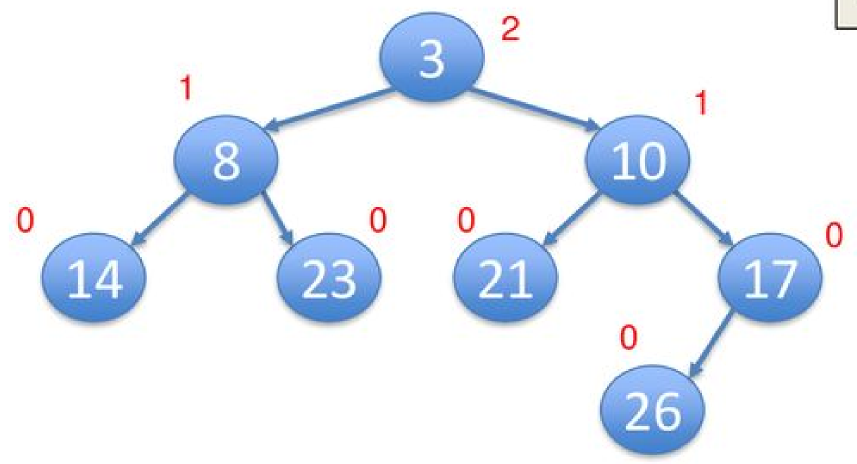
- Compare the roots. Smaller root becomes new root of subheap
- "Hang up" the left child of the new root
- Recursively merge the right subheap of the new root with the other heap.
- Update the npl
- Verify a leftist heap! (left npl >= right npl)
    - if not, swap troubled node with sibling

Deleting from Leftist Heap

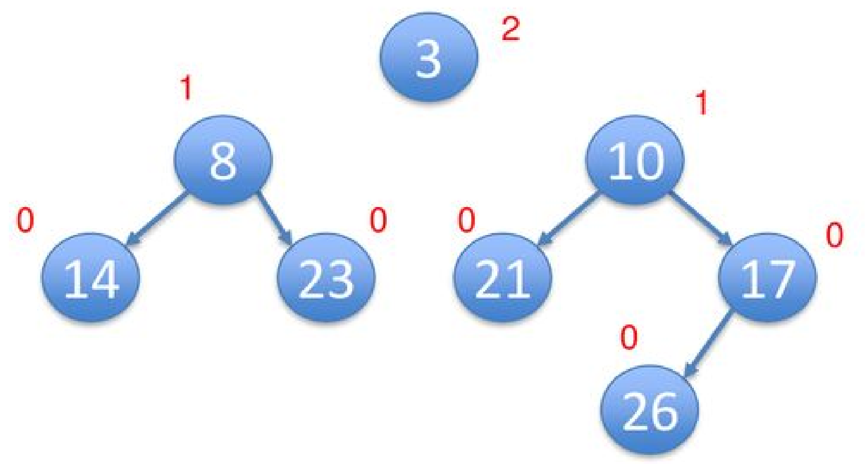
- Simple to just remove a node (since at top)
  - this will make two trees
- Merge the two trees like we just did

Deleting from Leftist Heap

We remove the root.

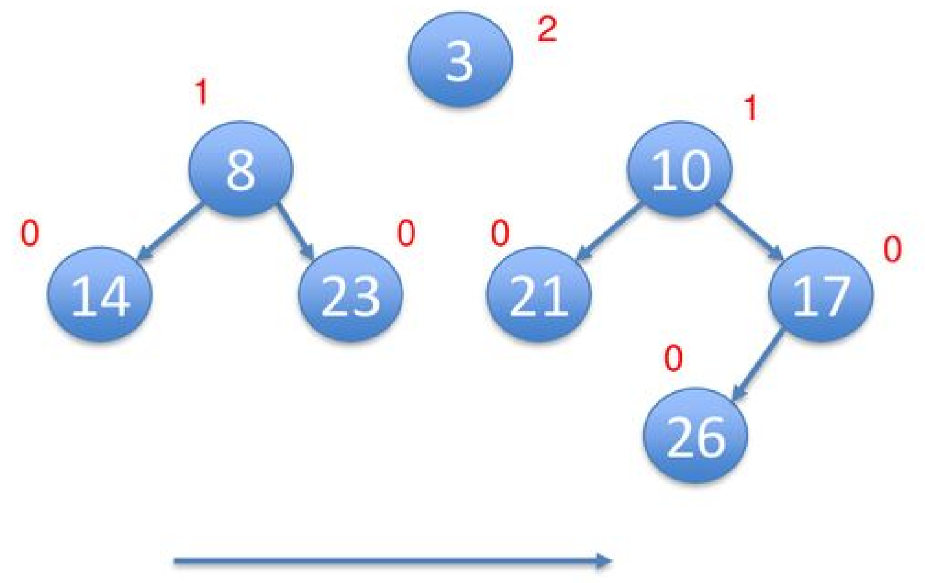


Deleting from Leftist Heap



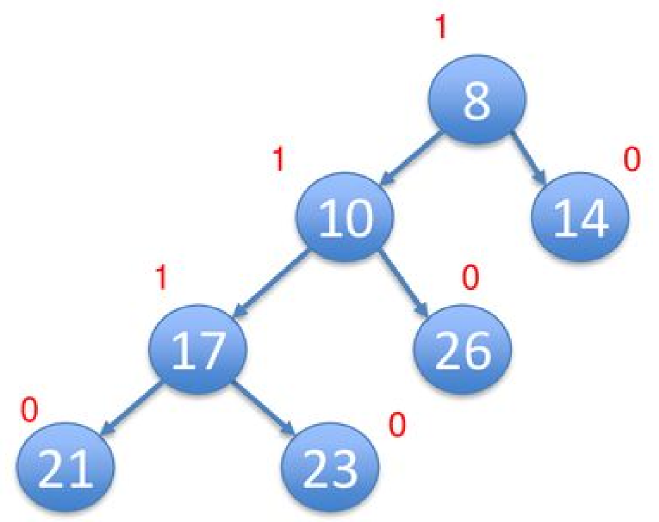
Then we do a merge and because min is in left subtree, we recursively merge right into left

Deleting from Leftist Heap



Then we do a merge and because min is in left subtree, we recursively merge right into left

Deleting from Leftist Heap



After Merge

Leftist Heaps

- Merge with two trees of size n
  - O(log n), we are not creating a totally new tree!!
  - some was used as the LEFT side!
- Inserting into a left-ist heap
  - O(log n)
  - same as before with a regular heap
- `deleteMin` with heap size n
  - O(log n)
  - remove and return root (minimum value)
  - merge left and right subtrees

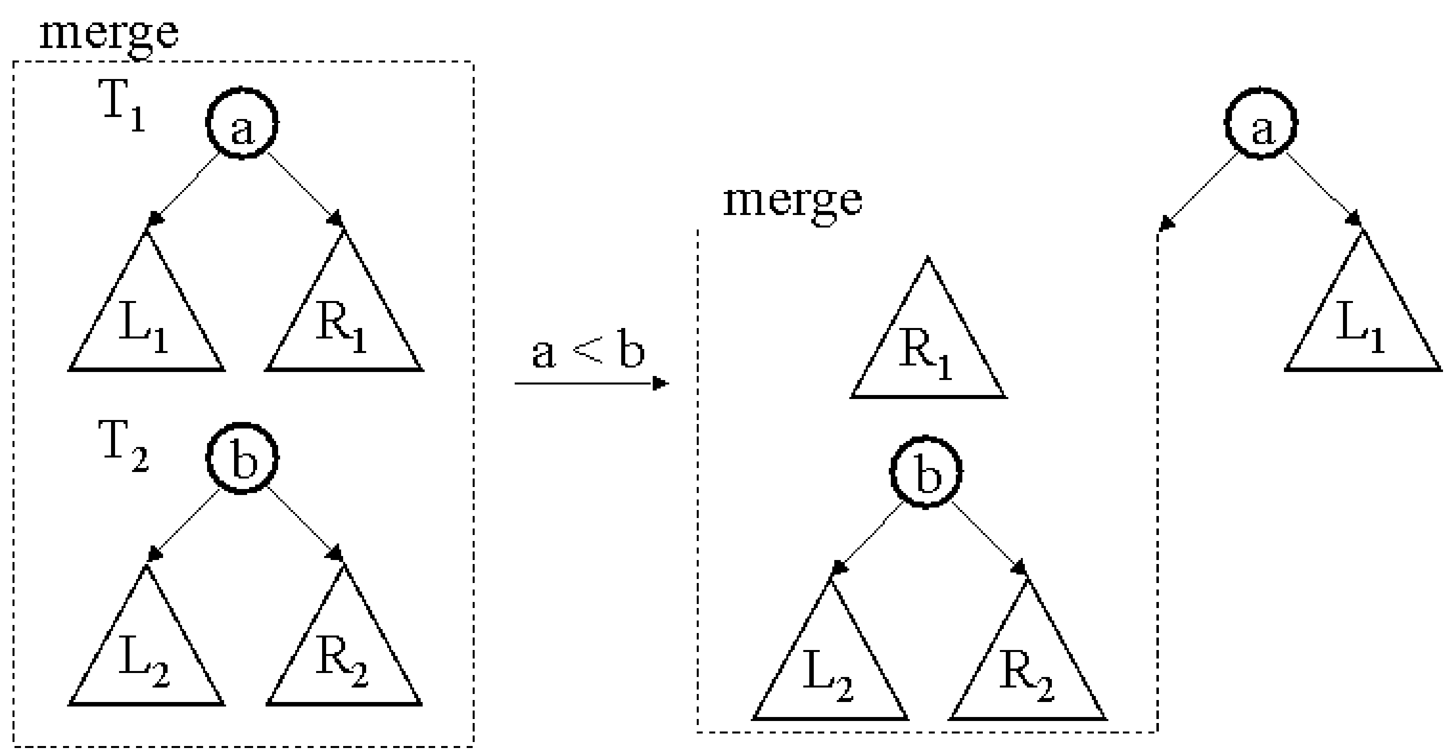
skew heap

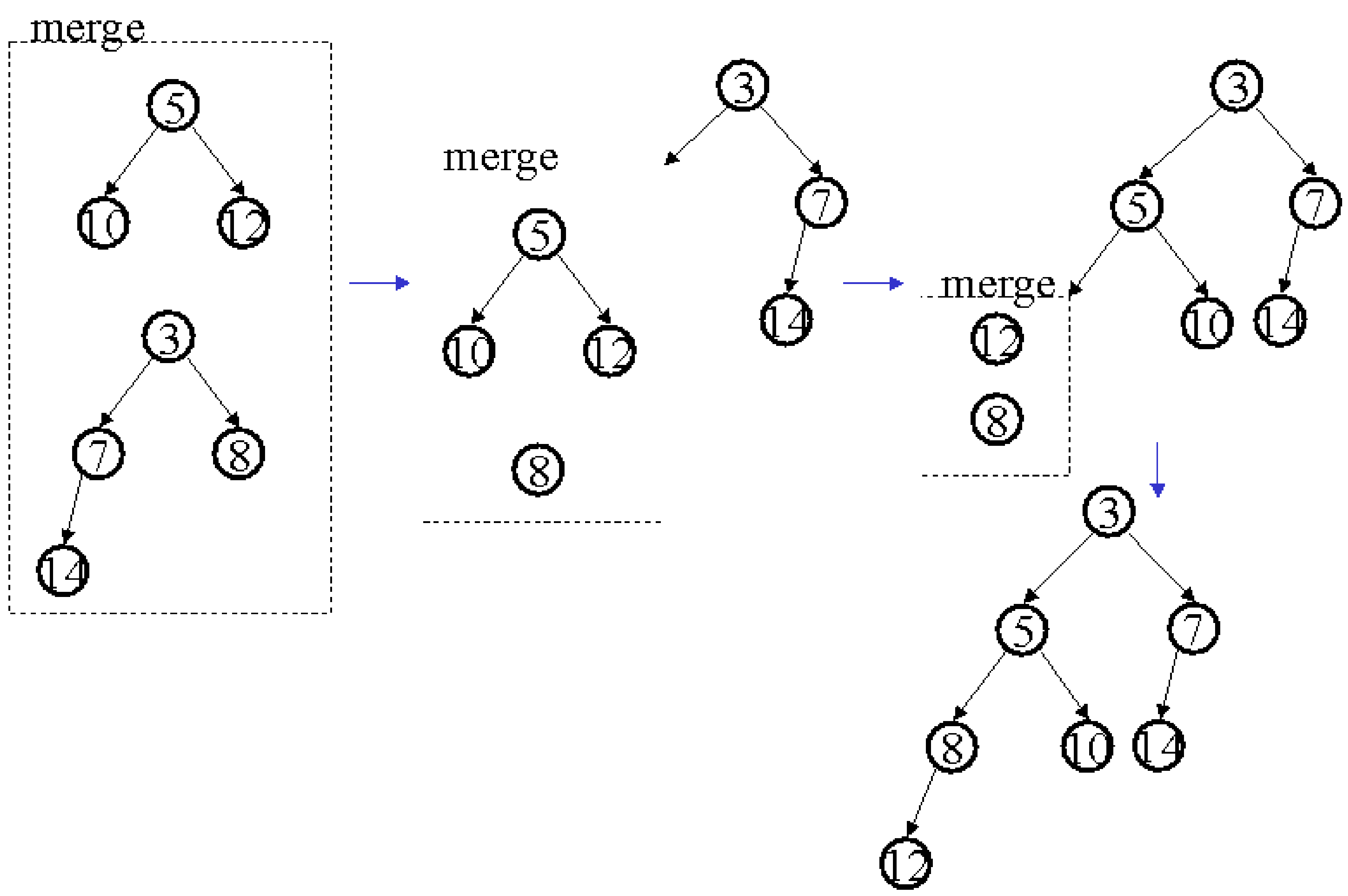
1. binary trees with heap order, but no structural constraint.
2. Skew heaps offer faster merge time as they are a special case of leftist trees.
3. Unlike leftist heaps, no information is maintained about null point length(s value) .

3. right path of a skew heap can be arbitrarily

3.Skew heap is extremely similar with leftist heap in terms of merge operation.
4. There is only one difference: for leftist heap, we check to see whether the left and right children satisfy the leftist heap order property and swap them if they do not.
5. However, for skew heaps, the swap is unconditional. In other words, we always swap the left & right subtrees at each step of merge.

Merging Two Skew Heaps

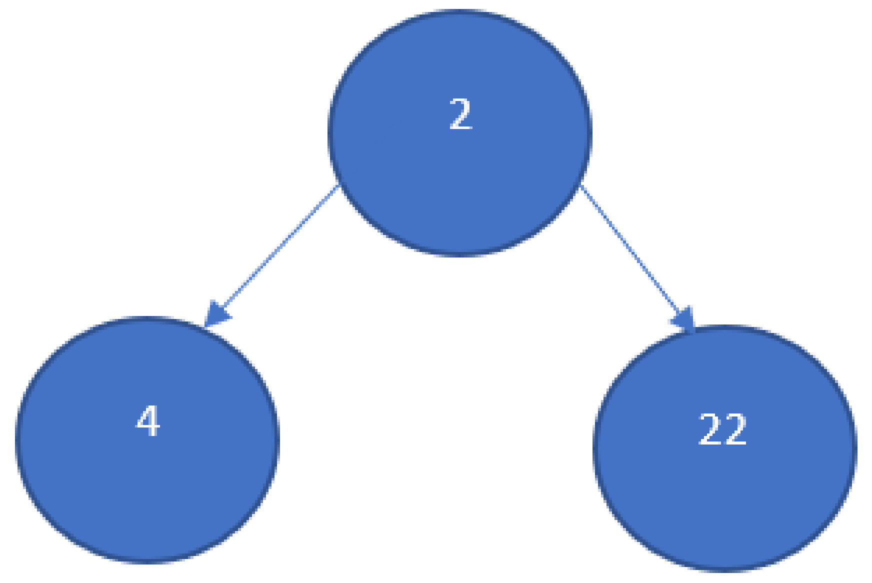


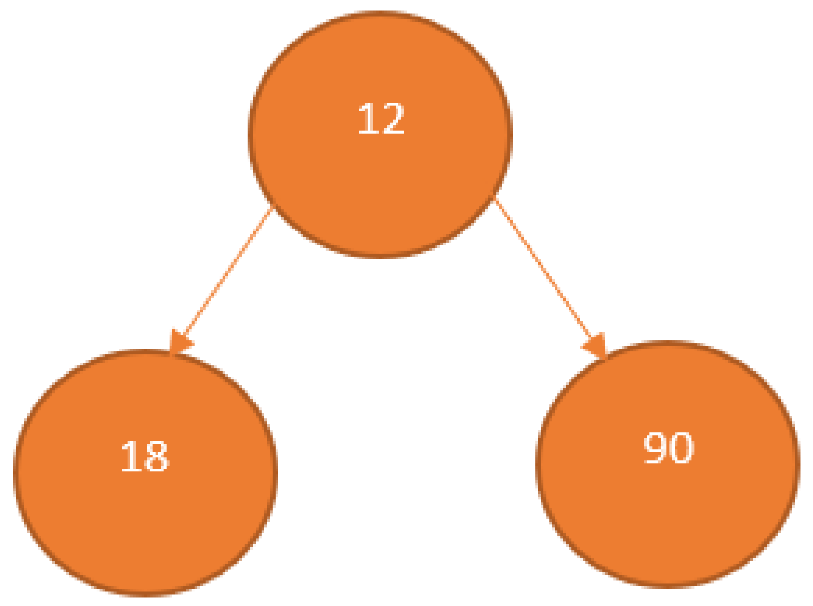

merge
merge
merge

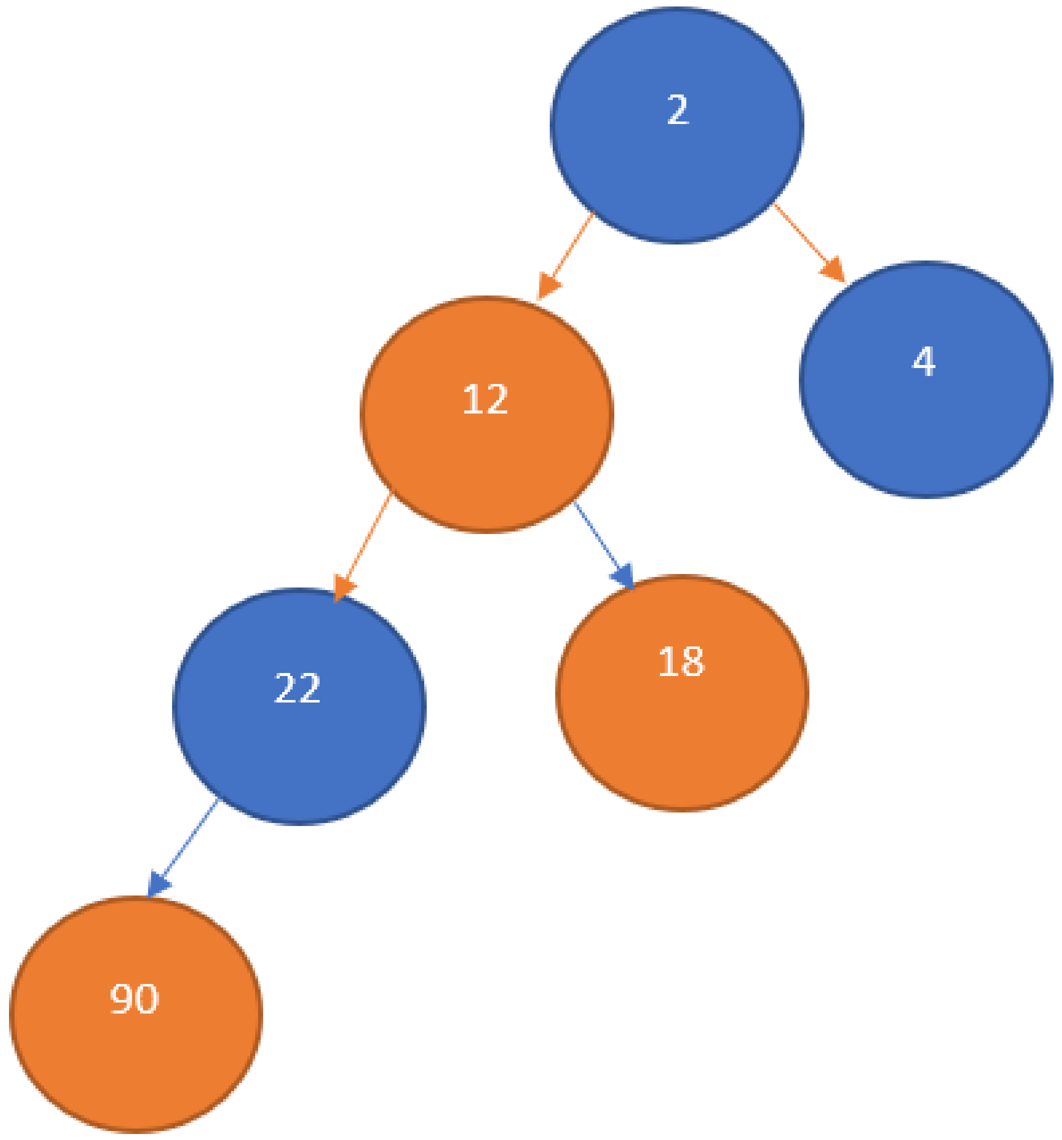
Skew Heap Code

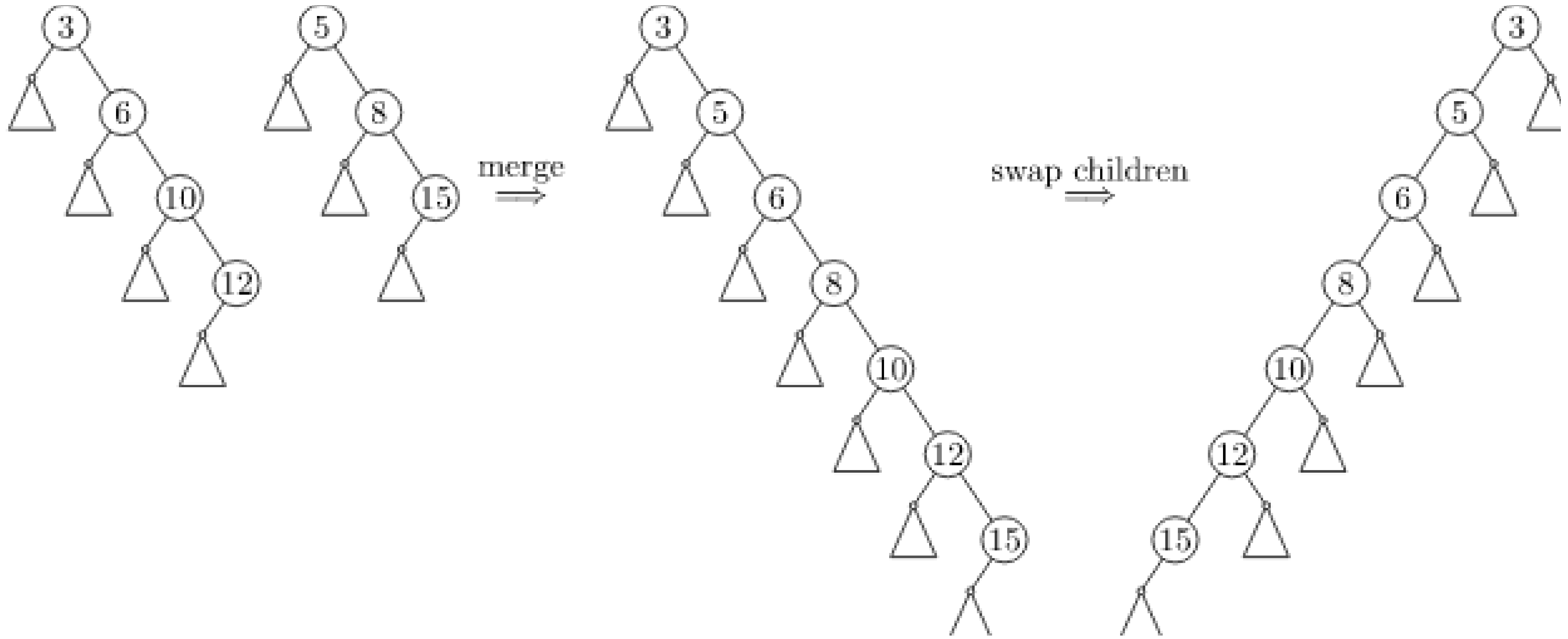
```
void merge(heap1, heap2) {
  case {
      heap1 == NULL: return heap2;
      heap2 == NULL: return heap1;
      heap1.findMin() < heap2.findMin():
            temp = heap1.right;
            heap1.right = heap1.left;
            heap1.left = merge(heap2, temp);
            return heap1;
      otherwise:
            return merge(heap2, heap1);
  }
}
```

Compare roots of two heaps
Recursively merge the heap that has the larger root with the right subtree of the other heap.
Make the resulting merge the right subtree of the heap that has smaller root.
Swap the children of the new heap

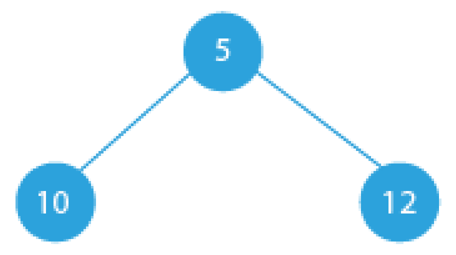

2
4
22


12
18
90

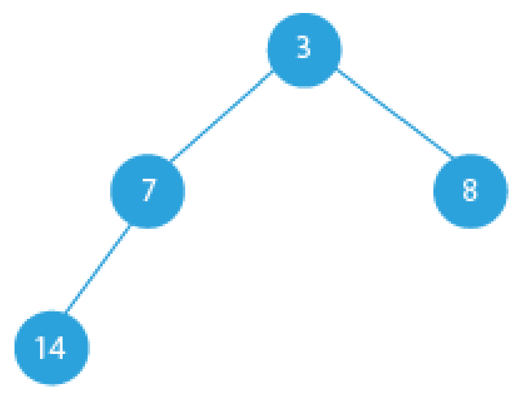

2
12
4
22
18
90


3
6
10
12
5
8
15
merge
3
5
6
8
10
12
15
swap children
3
5
6
8
10
12
15

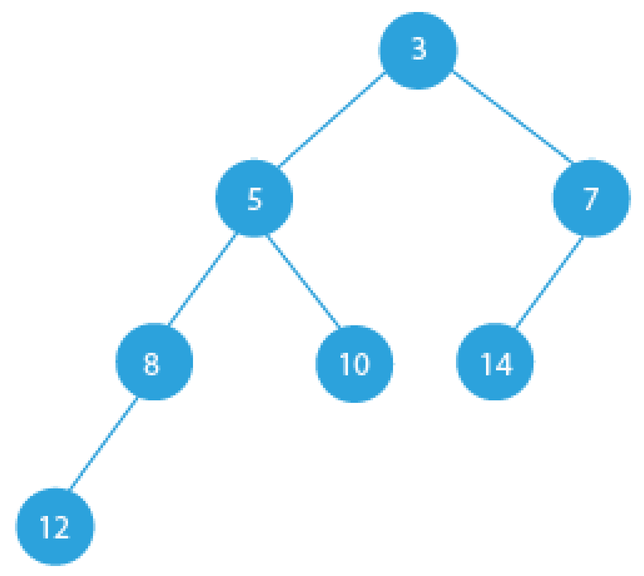
- Let us have a skew heap is (h1):



- And Second skew heap is (h2):



- The resultant merge skew heap will be (h3):



- **Insert a node**, we merge the heap with a new single-node heap.
- **Extract-min**, we delete the root node and merge the two remaining subtrees.