CSCI 335 Software Design and Analysis III

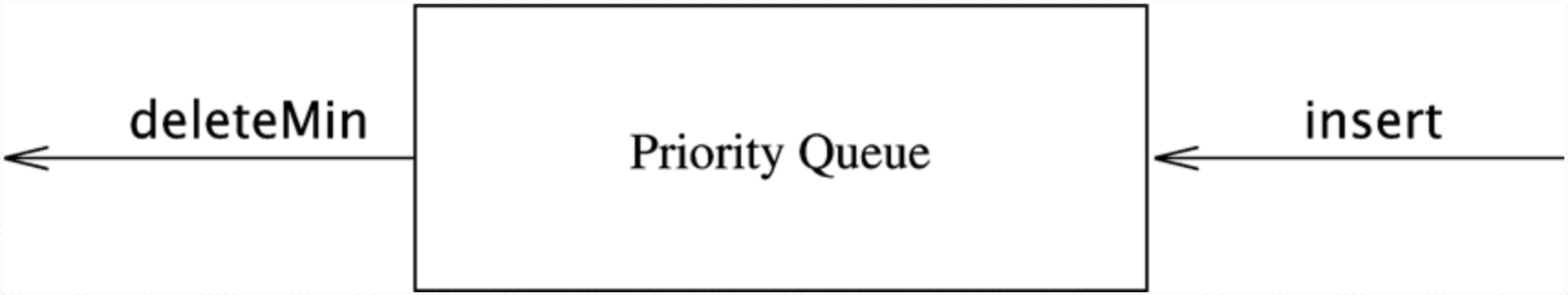
Priority Queues

(Binary Heaps, d-Heaps, Leftist Heaps)

Priority Queues

- Queue where elements have priorities
- Efficient Implementation
- Advanced Implementations
- Uses of PQs
  - Implementation of **greedy** algorithms

Priority Queue: basic operations



Possible implementations

- List
- Sorted List
- Binary search tree implementation
  - Appropriate for any priority queue
  - Largest item is rightmost and has at most one child

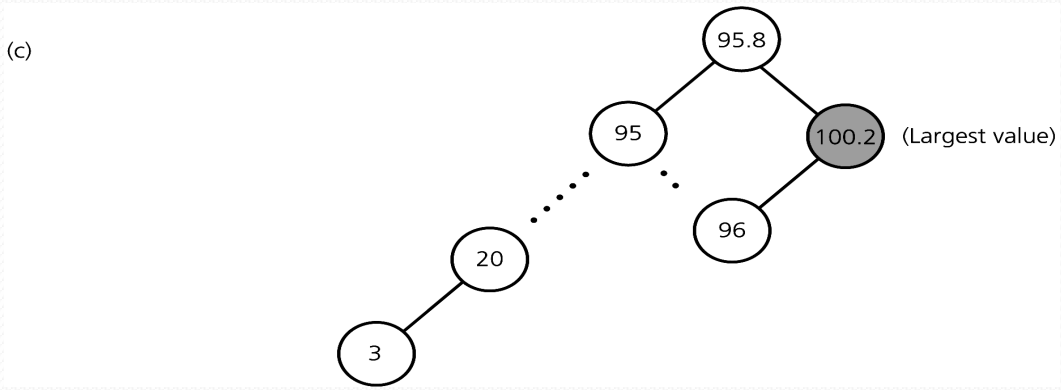


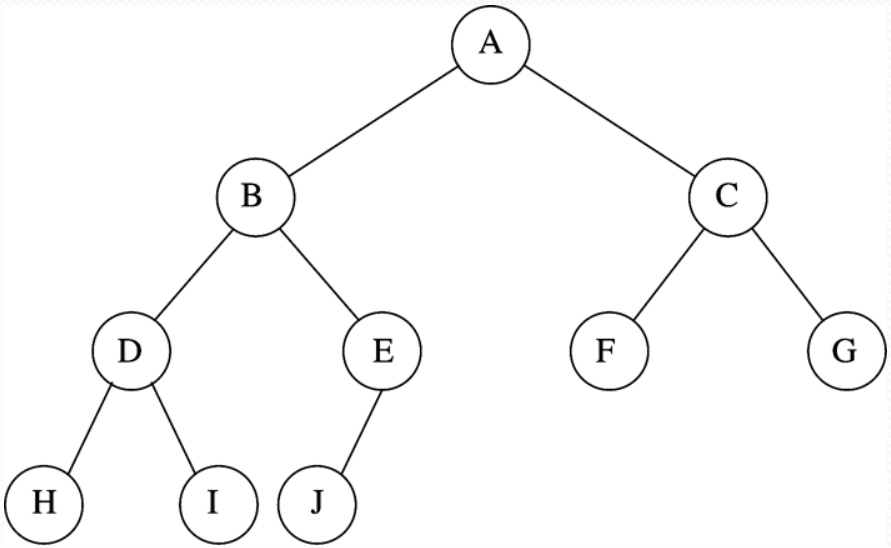
Figure 11-9c A binary search tree implementation of the ADT priority queue

First efficient implementation

- No links (pointers) required
- Very easy to implement
- O(logN) worst-case time for insert/deleteMin
- O(1) to access the min elements
- O(1) on average for insertions
- O(N) for building a queue of N items

Binary Heap

(1): Structure property -> Complete Binary Tree



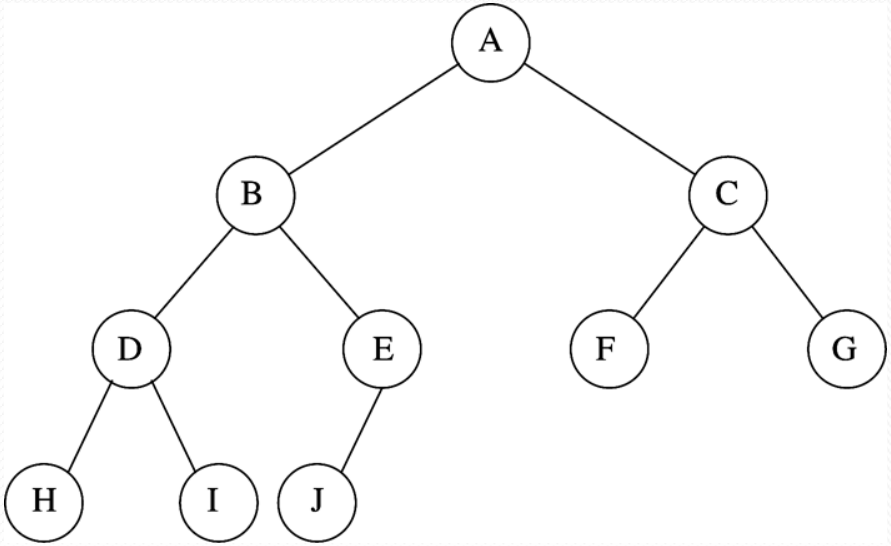
Height is floor(logN)

Can be implemented using an array !

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Binary Heap

(1): Structure property -> Complete Binary Tree



Height is floor(logN)

Left Child = 2 * Parent
Right Child = Left Child + 1

Parent = floor(Child / 2)

Can be implemented using an array !

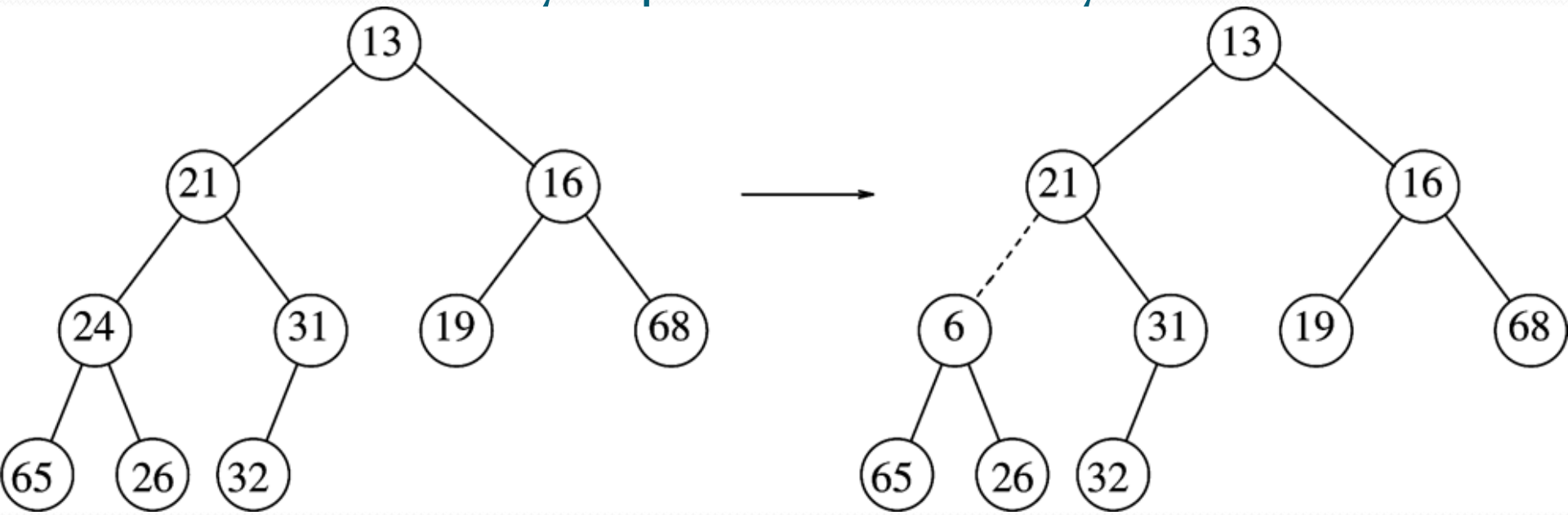
	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Binary Heap

(2): Heap-order property

For every node X (except root):

key of parent of X <= key of X

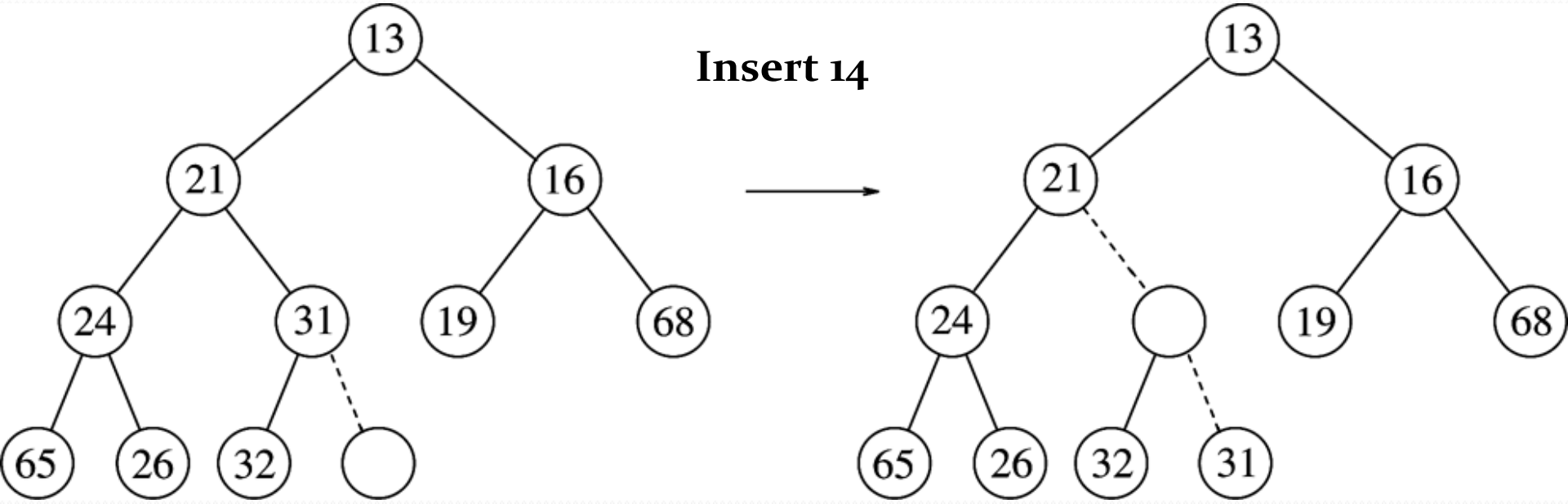


Heap **not a Heap**

Example: Insert 6, 5, 4, 3, 2, 1 into an empty heap

Basic Heap Operations

- Insert (percolate up)

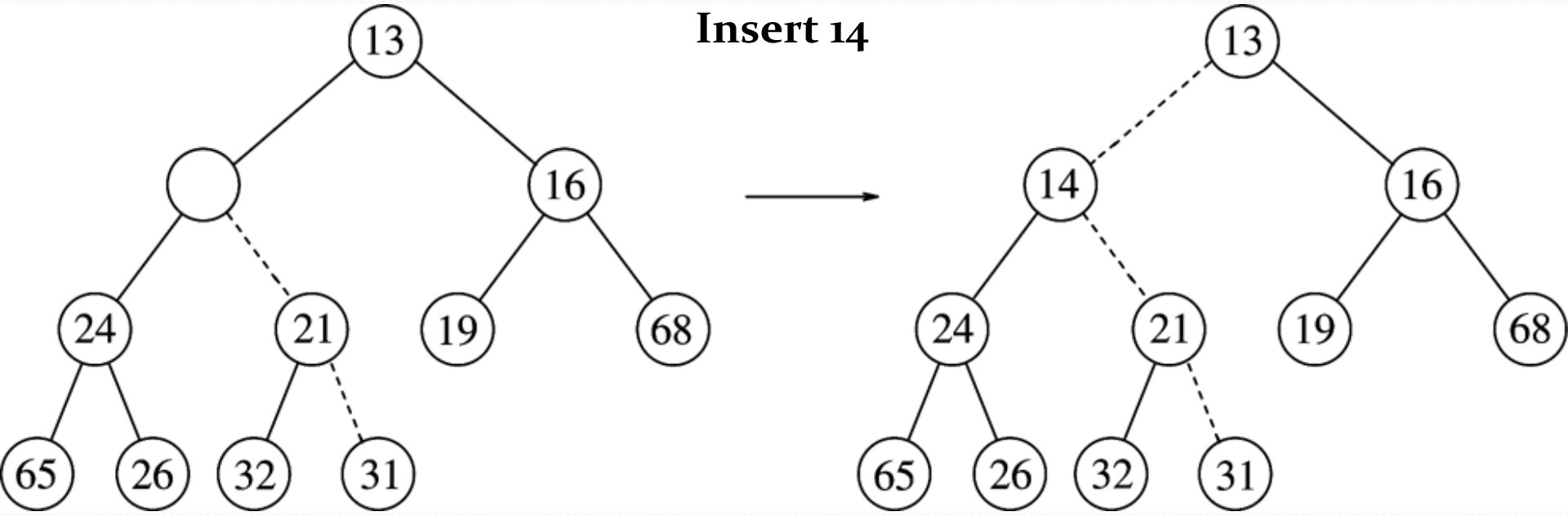


Index: 0 1 …. … 10
Array: -, 13, 21, 16, 24, 31, 19, 68, 65, 26, 32

0 1 …. 5 … 10 11
-, 13, 21, 16, 24, H, 19, 68, 65, 26, 32, 31

Basic Heap Operations

- Insert (percolate up)



0 1 2 5 … 10 11
-, 13, H, 16, 24, 21, 19, 68, 65, 26, 32, 31

0 1 2 5 … 10 11
-, 13, 14, 16, 24, 21, 19, 68, 65, 26, 32, 31

Basic Heap Operations

- Insert: O(logN)
  - Worst-case: key to be inserted is the new minimum

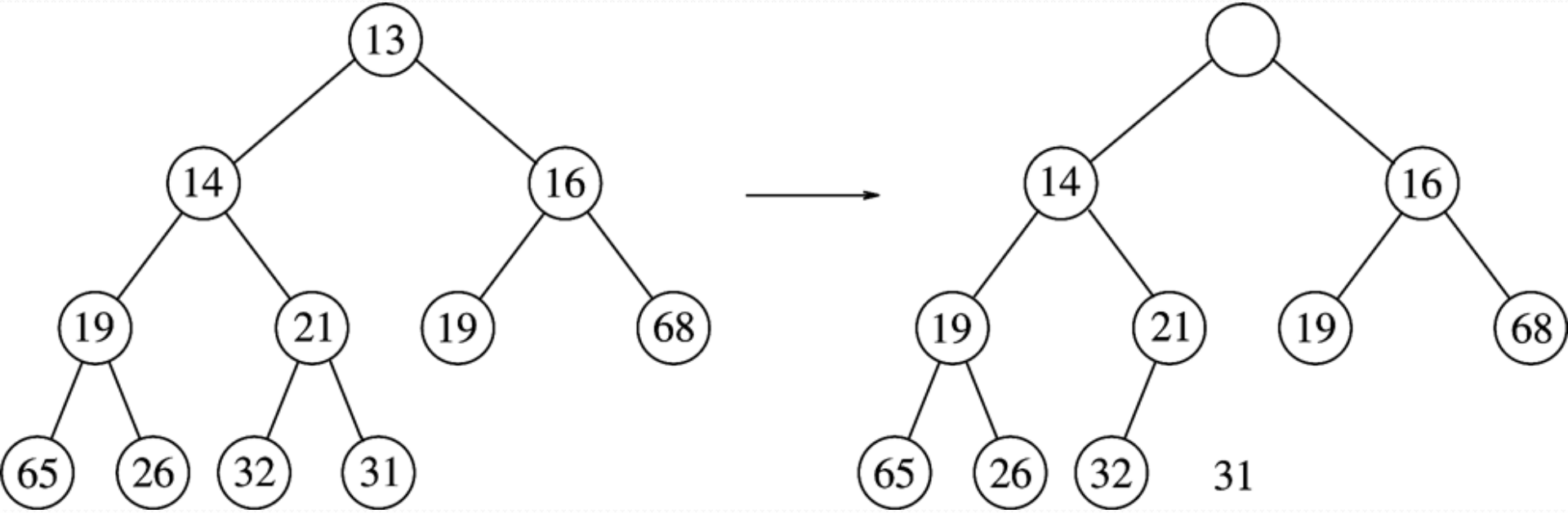
    => will be percolated up all the way to the root.

On average 2.607 comparisons are required…

```
// @x: item to be inserted into binary heap.
// Heaps starts from index 1. Location 0 is unused.
void Insert(const Comparable &x) {
     if (current_size_ == array_.size() - 1)  // Heap full.
        array_.resize(array_.size() * 2);
     int hole = current_size_++;
     Comparable copy = x;
     // Save item to dead space of heap.
     // Used also as marker to stop the loop.
     array_[0] = std::move(copy);
     for ( ; x < array_[hole / 2]; hole /= 2)
       array_[hole] = std::move(array_[hole / 2]);
     array_[hole] = std::move(array_[0]);
}
```

Basic Heap Operations

- DeleteMin: Percolate down

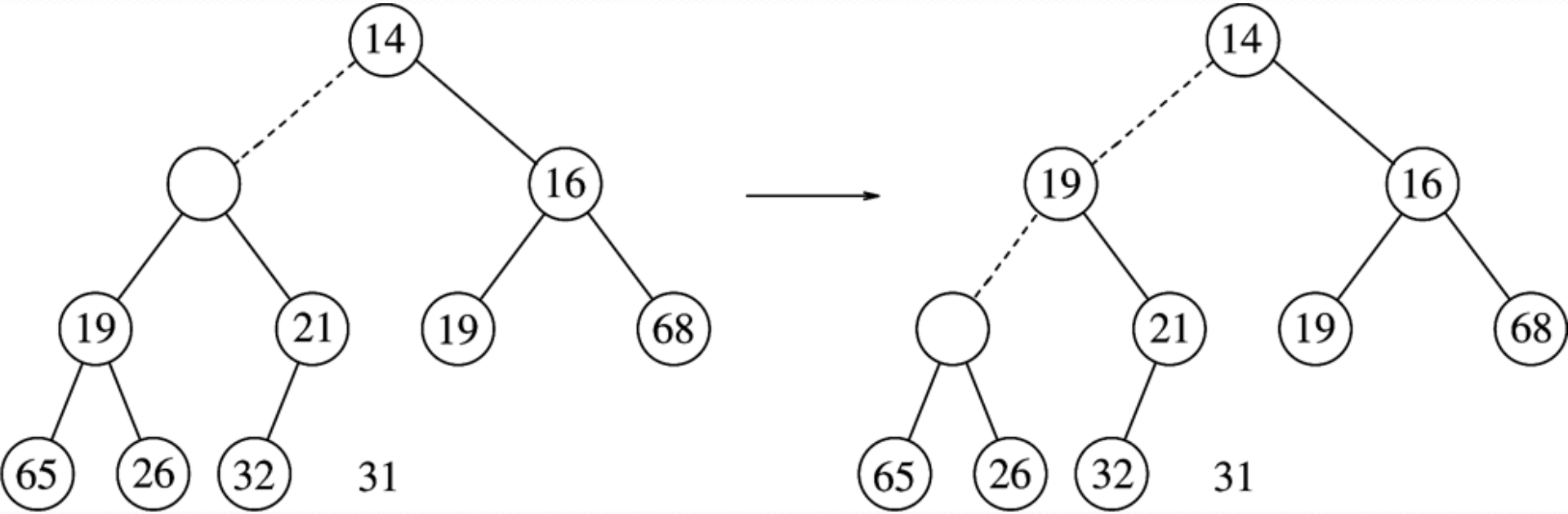


0 1 2 5 … 10 11
-, 13, 14, 16, 19, 21, 19, 68, 65, 26, 32, 31

0 1 2 5 10
-, 31, 14, 16, 19, 21, 19, 68, 65, 26, 32

Basic Heap Operations

- DeleteMin: Percolate down



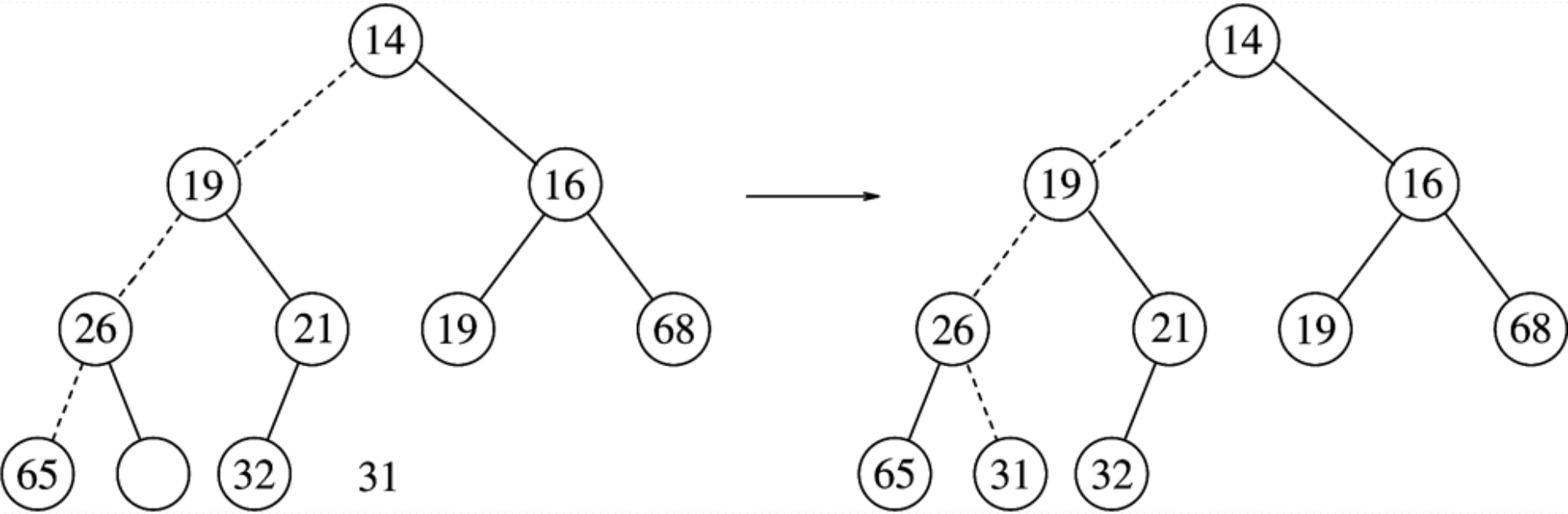
tmp = 31

0	1	2	3		5				10
-	14	14	16	19	21	19	68	65	26	32

0	1	2	3	4	5			8	9	10
-	14	19	16	19	21	19	68	65	26	32

Basic Heap Operations

- DeleteMin: Percolate down



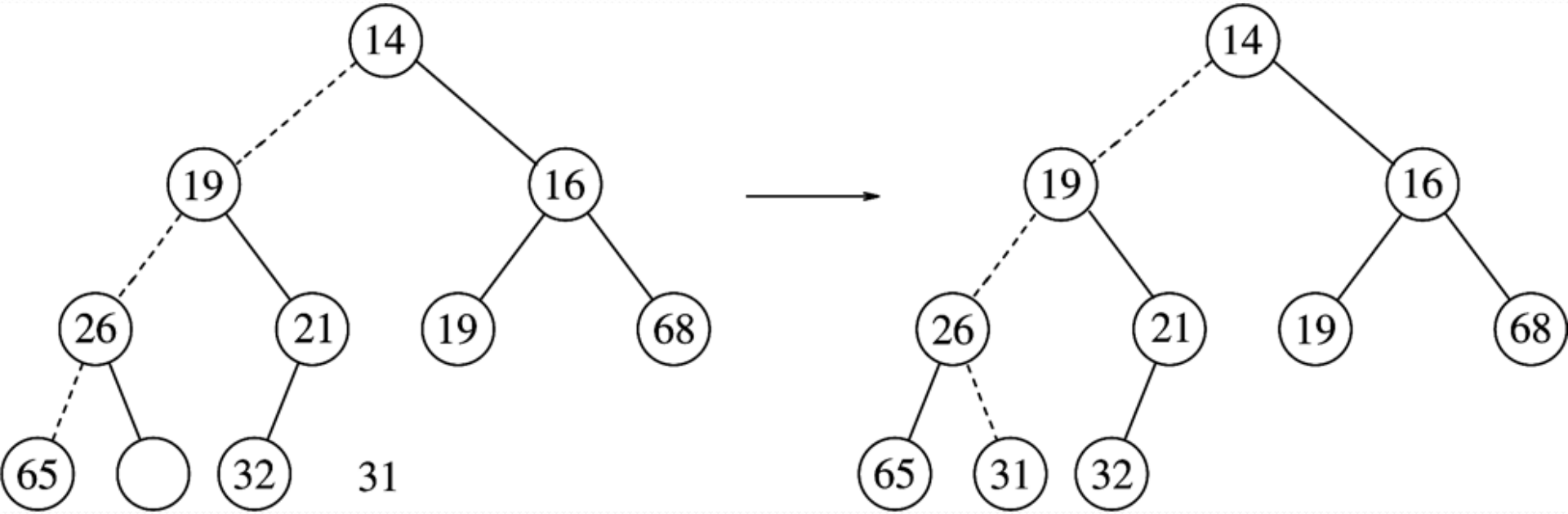
0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, 26, 32

0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, 31, 32

Basic Heap Operations

- DeleteMin: Percolate down

Continue: one more DeleteMin



0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, 26, 32

0 1 2 3 4 5 8 9 10
-, 14, 19, 16, 26, 21, 19, 68, 65, 31, 32

Basic Heap Operations

- DeleteMin: O(logN) worst- and average-case

Other Operations

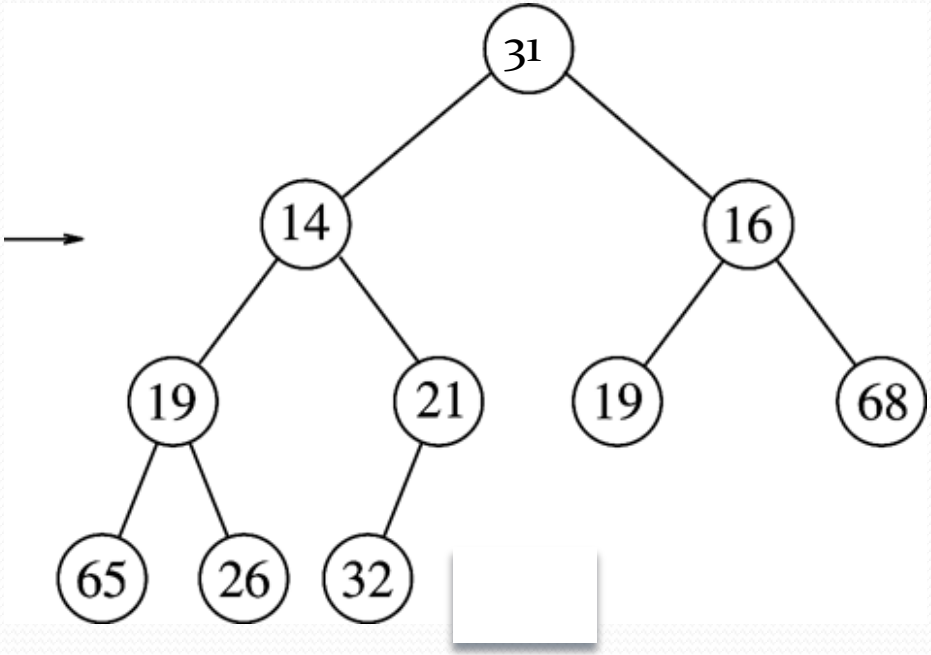
- decreaseKey(p, Δ): lower key at position p by positive Δ
- increaseKey(p, Δ): increase key at position p by positive Δ
- remove(p):

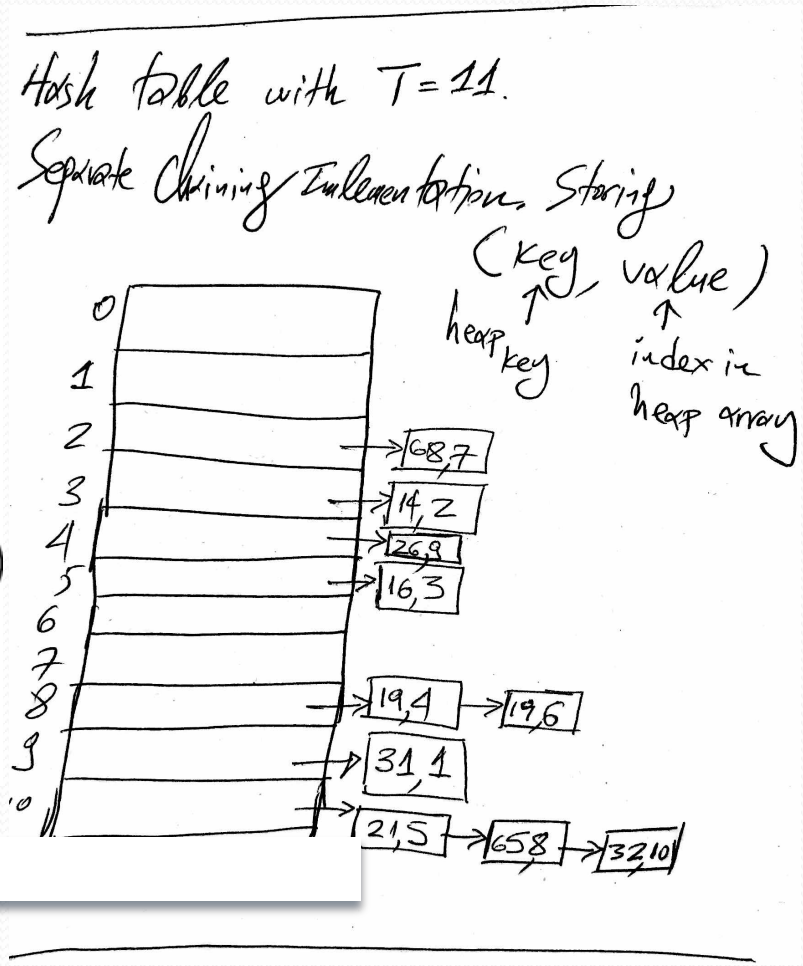
    decreaseKey(p, ∞)

    deleteMin

Question: How to find position p of key?

Heap and hash table





0 1 2 5 10

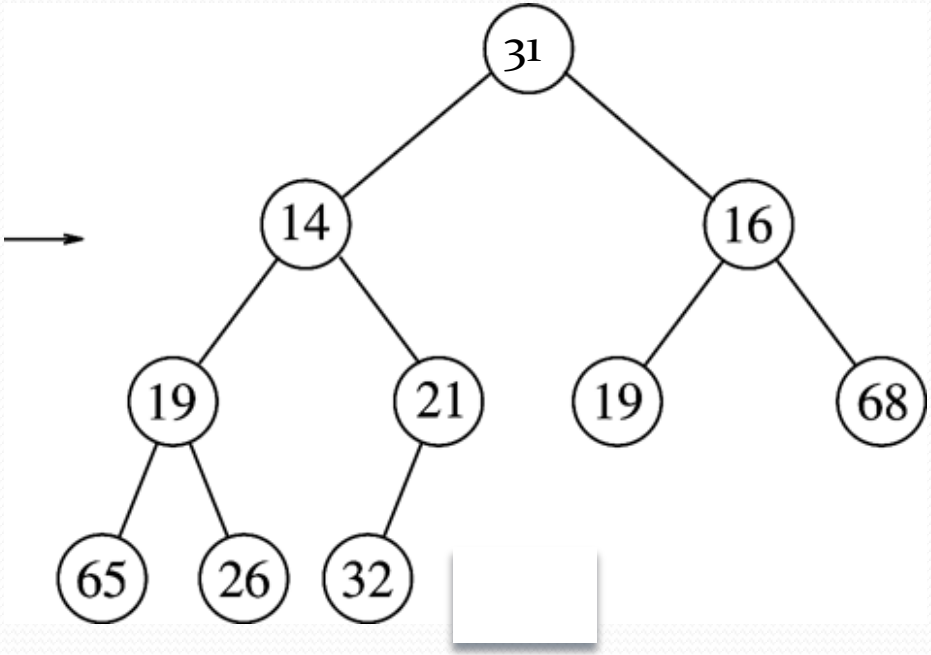
-, 31, 14, 16, 19, 21, 19, 68, 65, 26, 32

```
// @param hole: index of element on array_
// Percolates down element stored in array_[hole].
void PercolateDown(int hole) {
     Comparable tmp = std::move(array_[hole]);
     int child;
     for(; hole * 2 <= current_size_; hole = child) {
        child = hole * 2;  // Index of left child.
        if (child != current_size_ &&
            array_[child + 1] < array_[child]) ++child;
        // child is the index of the minimum of the two children.
        if (array_[child] < tmp)
            array_[hole] = std::move(array_[child]);
        else
            break;  // Stop percolating down.
      }  // End for
      array_[hole] = std::move(tmp);
}
----

For DeleteMin() need to call PercolateDown(1)
```

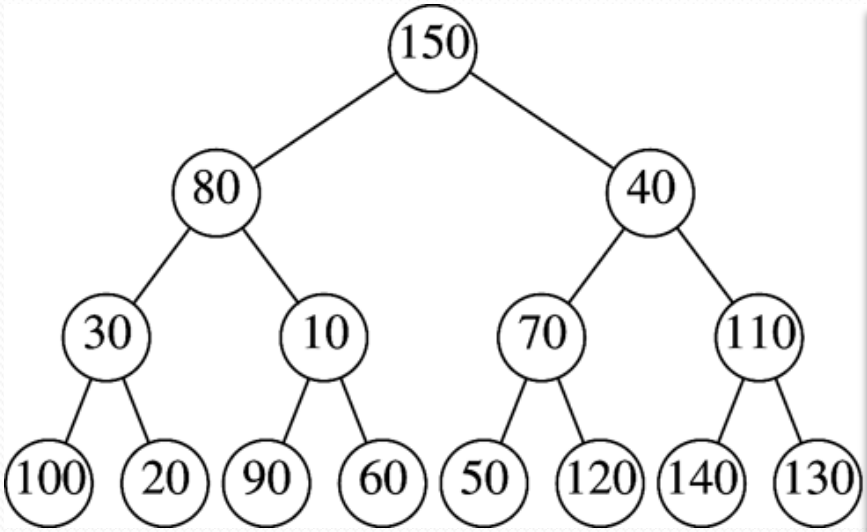
Run PercolateDown code

- PercolateDown(1)



0	1	2			5					10
-,	31,	14,	16,	19,	21,	19,	68,	65,	26,	32

Build Heap from array

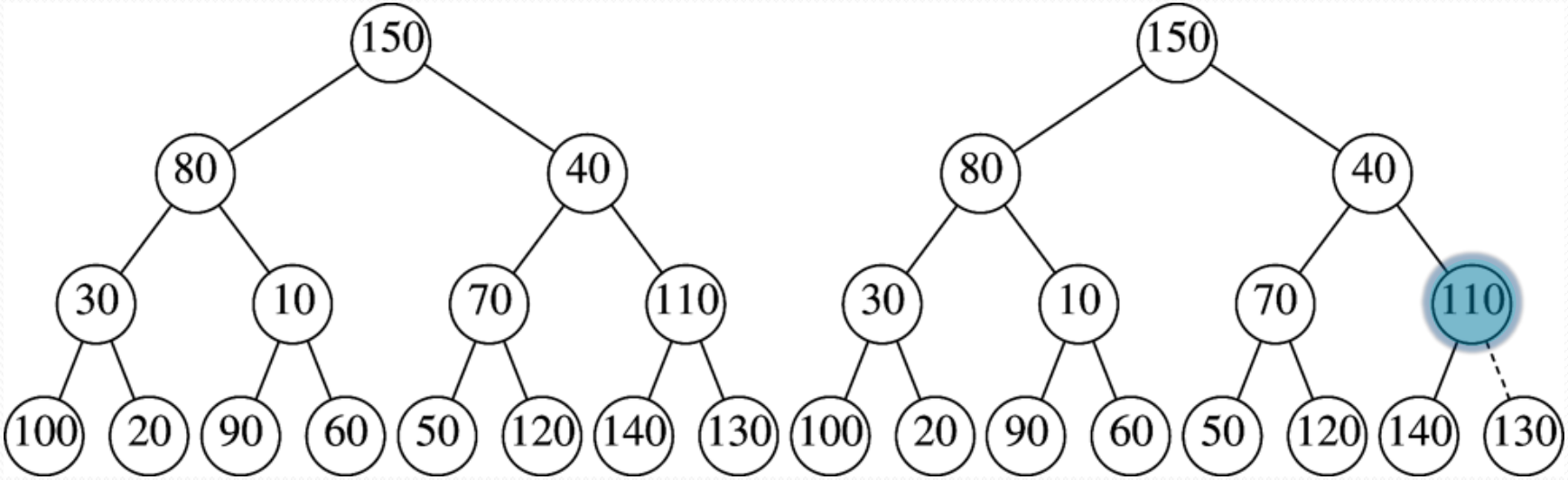


Initial array :

| | 150 | 80 | 40 | 30 | 10 | 70 | 110 | 100 | 20 | 90 | 60 | 50 | 120 | 140 | 130 |

current_size_ = 15

Build Heap from array



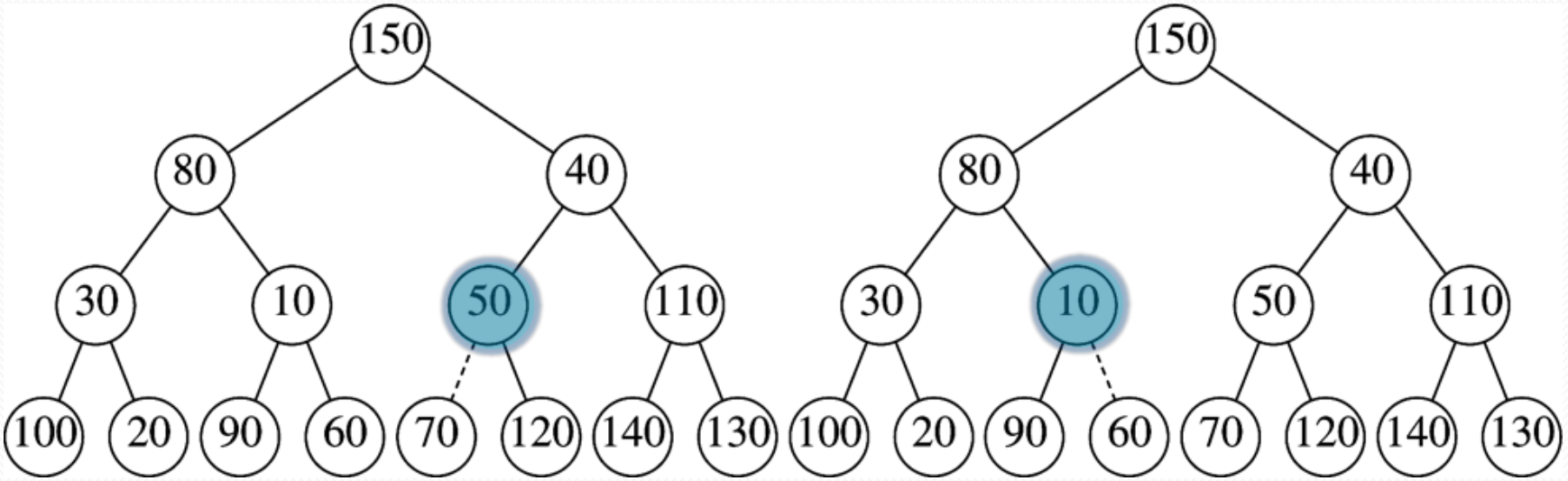
PercolateDown(7)

Initial array :

| | 150 | 80 | 40 | 30 | 10 | 70 | 110 | 100 | 20 | 90 | 60 | 50 | 120 | 140 | 130 |

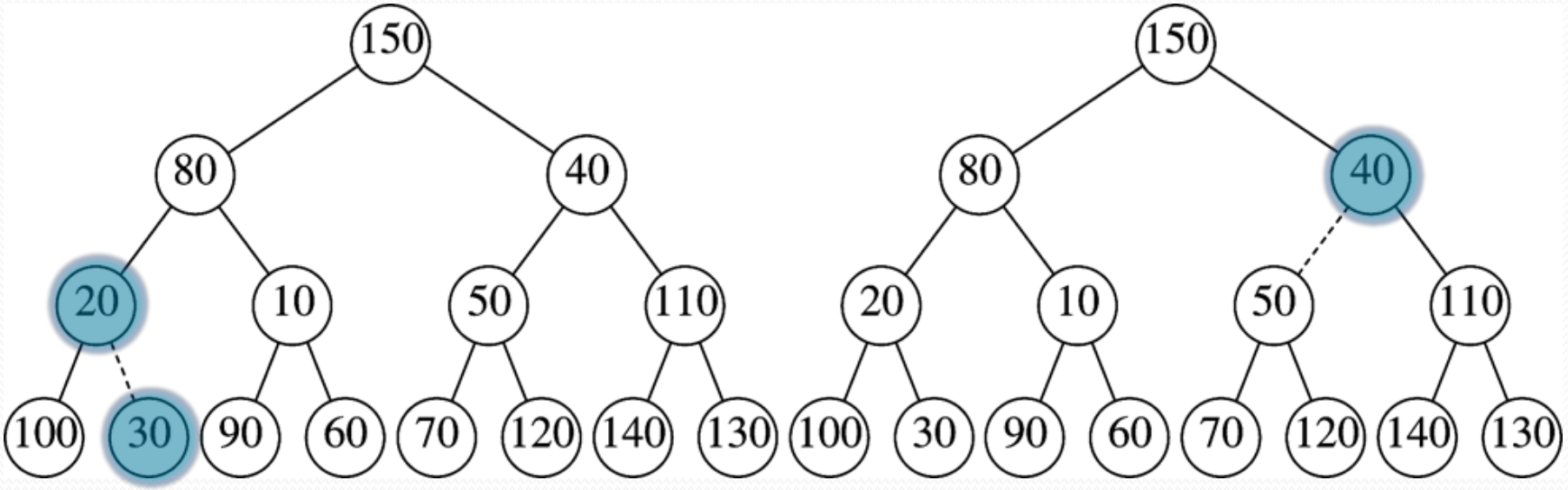
current_size_ = 15 (=> current_size_ / 2 = 7)

Build Heap from array



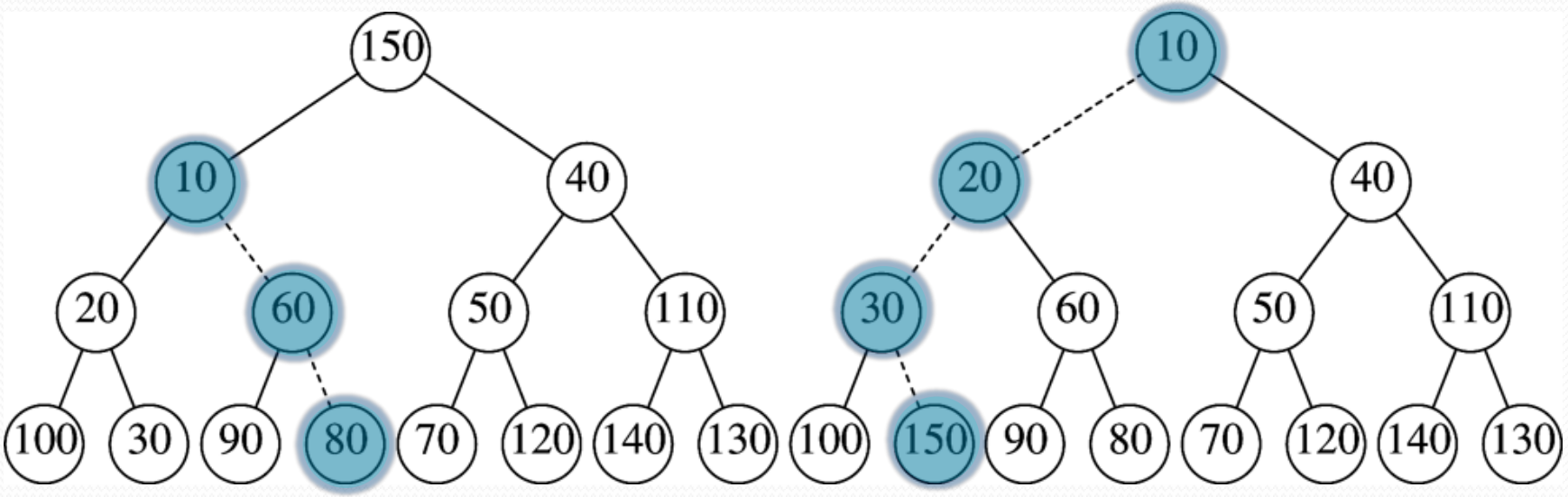
PercolateDown(6) **PercolateDown(5)**

Build Heap from array



PercolateDown(4) **PercolateDown(3)**

Build Heap from array



PercolateDown(2) **PercolateDown(1)**

BuildHeap

```
explicit BinaryHeap(const vector<Comparable> &items)
  : array_(items.size() + 10), current_size_{items.size()} {
      for (int i = 0; i < items.size(); i++ )
         array_[i + 1] = items[i];  // Create initial array
      BuildHeap();  // Make it a heap.
}

void BuildHeap() {
    for (int i = current_size_ / 2; i > 0; --i)
        PercolateDown(i);
}
```

buildHeap: running time?

- Each dash line corresponds to 2 comparisons
  - => 20 comparisons in previous example
- In worst-case: find total number of dash lines

  i.e. find the sum of heights of all nodes

buildHeap: running time?

Theorem:

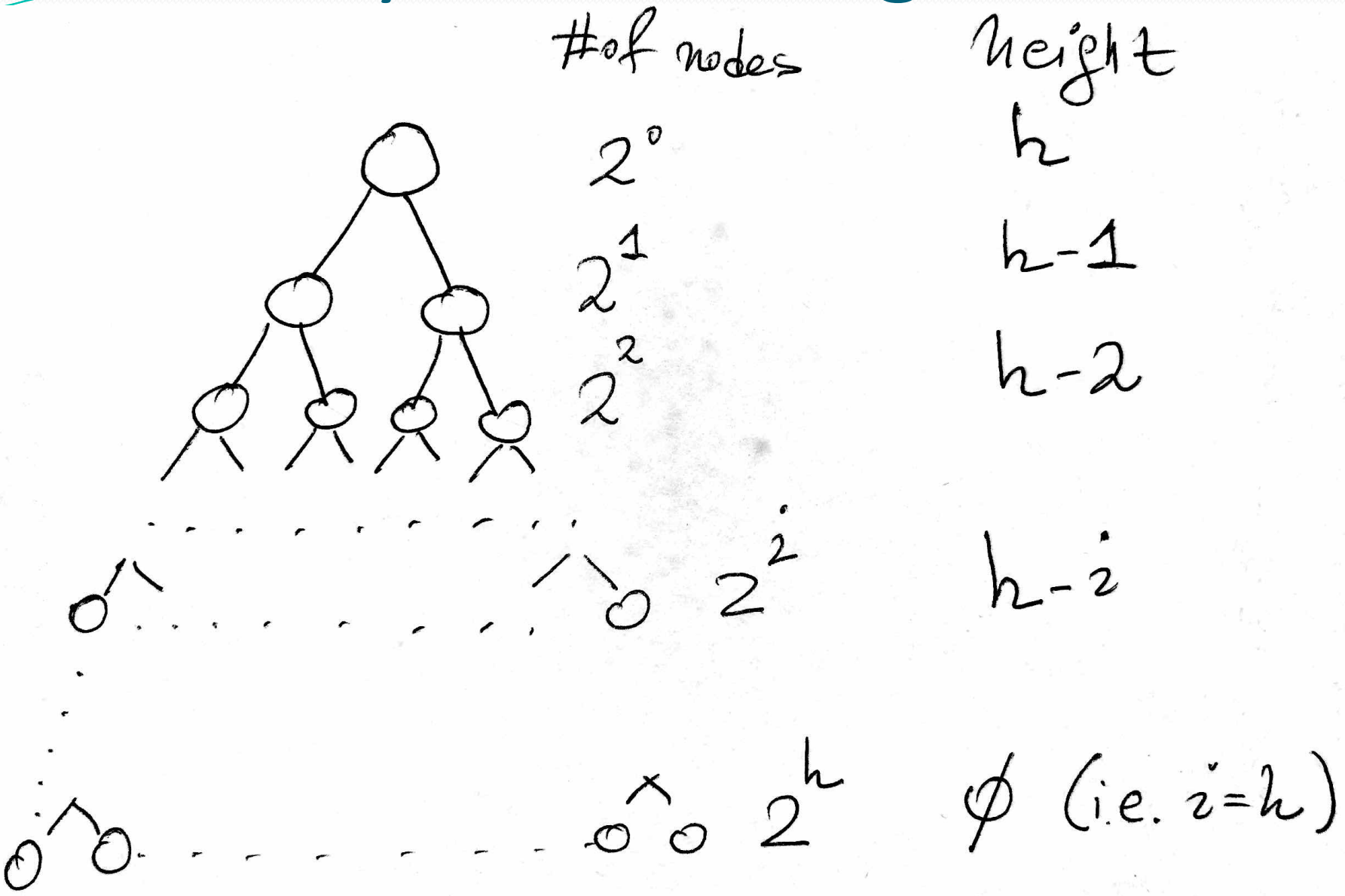
For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum S of the heights of the nodes is $2^{h+1} - 1 - (h+1)$.

Proof: … (next slide)

Running Time: A complete tree has between 2^h and 2^{h+1} nodes

=> **buildHeap is O(N)** where N is number of nodes

Full Binary Tree of height h



buildHeap Running Time

Theorem

For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 + (h + 1)$.

Proof

There is 1 node at height h, 2 nodes at height $h - 1$, ..., 2^i at $h - i$.

$$S = \sum_{i=0}^{h} 2^i(h - i) = h + 2(h - 1) + 4(h - 2) + \cdots + 2^{h-1}$$ (h-(h-1))

$$2S = 2h + 4(h - 1) + 8(h - 2) + \cdots + 2^h$$

$$2S - S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h$$

$$S = -h + 2^{h+1} - 1 - 1$$

$$S = \left(2^{h+1} - 1\right) - (h + 1)$$

buildHeap Running Time

- buildHeap is $O(N)$ worst-case
- Complete binary tree has between 2^h and 2^{h+1} nodes
- $S = \left(2^{h+1} - 1\right) - (h + 1) = O(N) - O(\log N) = O(N)$

The Selection Problem

Given a list of N elements, and integer k

Find the k^{th} largest (or smallest) element.

Ideas?

Try: 4^{th} smallest (1^{st} algo) or 4^{th} largest (2^{nd} algo)

2, 1, 3, 5, 10, 4, 7, 0

The Selection Problem

Ideas:

1. Sort N elements and find index k.
2. ~~Sort k elements, insert/delete N - k of them~~.
3. Build a heap of N elements and do k deleteMin.
4. Build a heap of k elements and do N - k insert/ deleteMin pairs.

k DeleteMin

- Build heap of n items O(n)
- k DeleteMins O(k logn)
- Total: O(n + k logn)

Example: 4^{th} smallest in : 2, 1, 3, 5, 10, 4, 7, 0

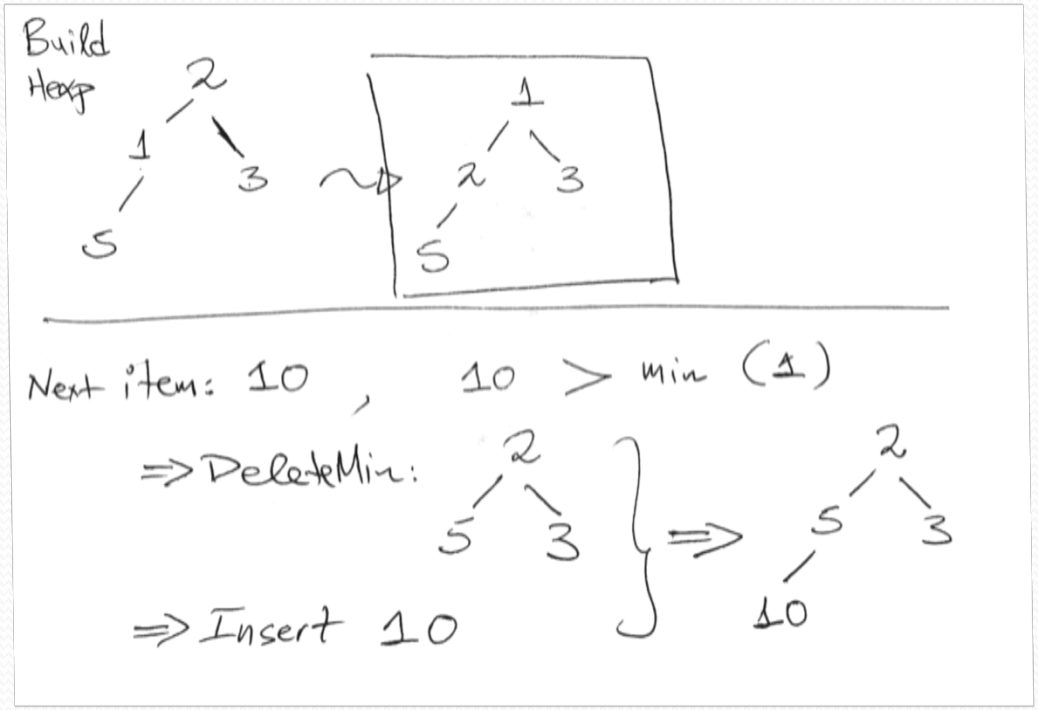
1^{st} DeleteMin -> 0

2^{nd} -> 1

3^{rd} -> 2

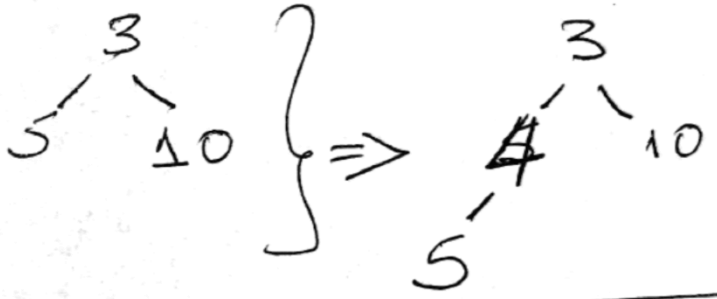
4^{th} -> 3

Alternative (4th largest)



Next item: 4 > min(2)

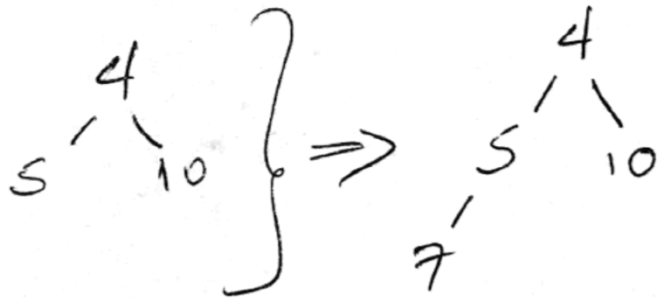
⇒ DeleteMin:



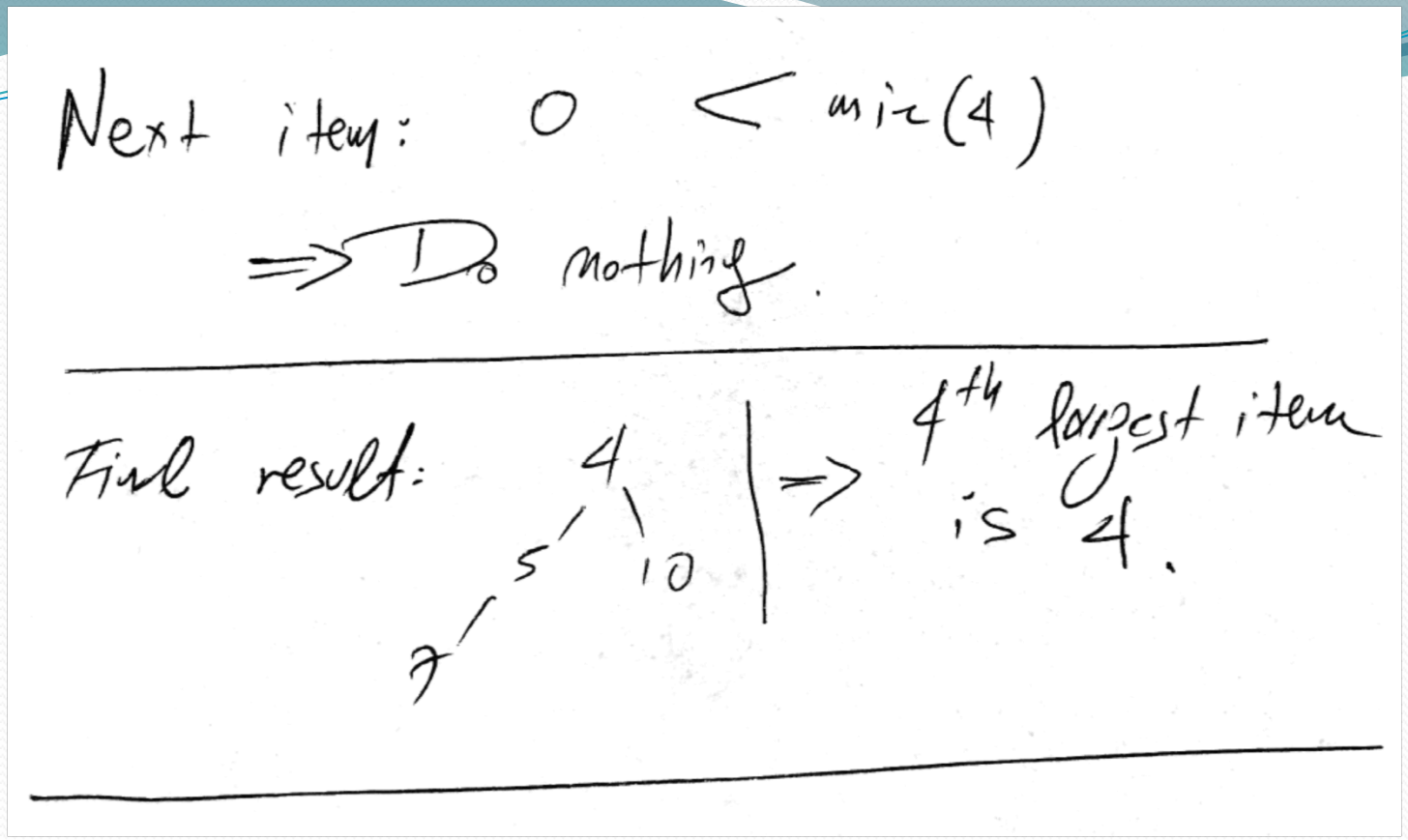
⇒ Insert 4

Next item: 7 > min(3)

⇒ Deletemin:



⇒ Insert 7



Online algorithm: Answer (i.e. 4th largest) as items are being read. Complexity?

The Selection Problem

Analysis

1. $O(n^2)$ or $O(n \log n)$
2. $O(n \cdot k)$
3. $O(n + k \log n)$
4. $O(k + (n - k) \log k) = O(n \log k)$

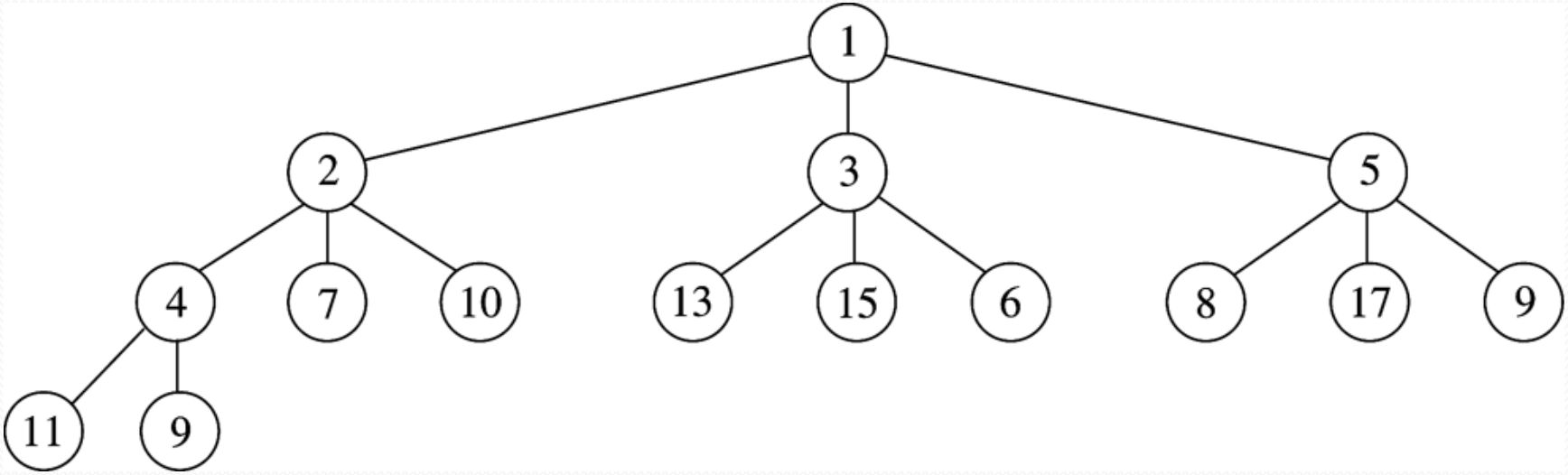
- In the worst-case, $k = \left\lceil \frac{n}{2} \right\rceil$ is the median.
- We will revisit this problem with a better solution based on the Quicksort algorithm.

d-Heaps

d children

Height is now $O(\log_d N)$=>less deep

Is it better than a 2-heap?



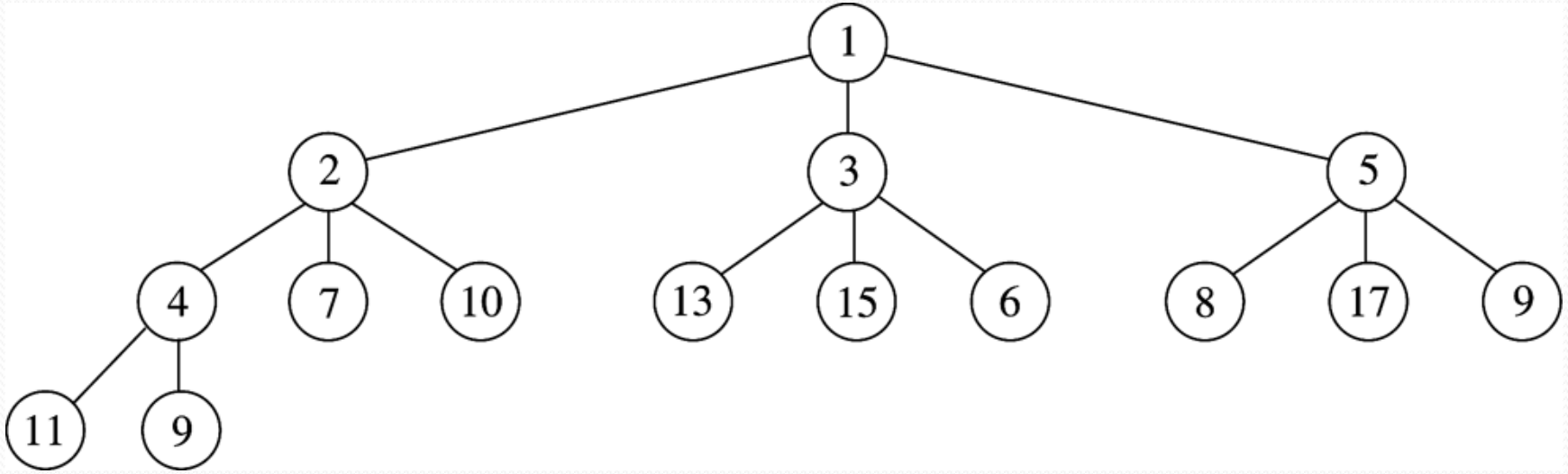
d-Heaps

d children

Height is now O($\log_d N$)=>less deep

Is it better than a 2-heap?

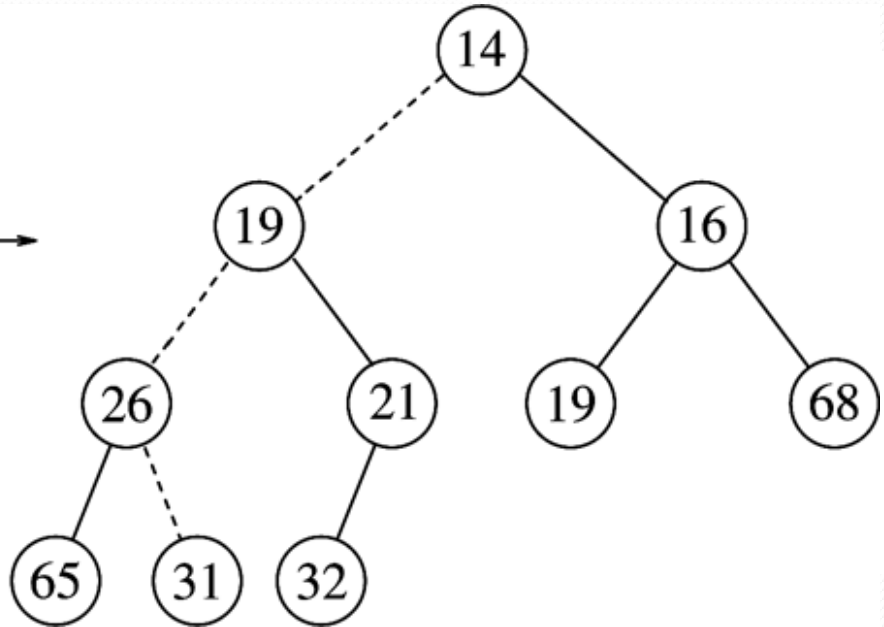
- Good for external (disk) implementation
- Good when more inserts than deleteMins
- In general 4-heaps outperform 2-heaps



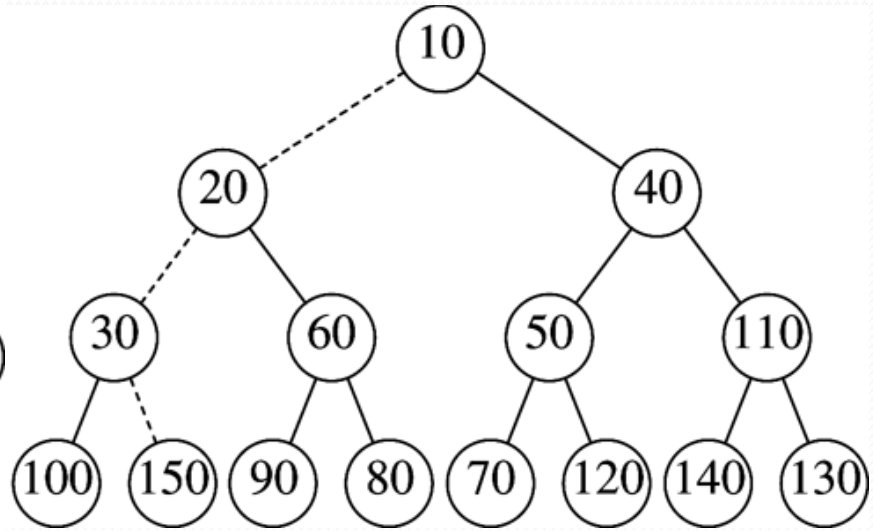
The Merge operation

- Given two heaps H1 and H2, merge them into heap H
- In a binary heap, what is the complexity of merge?
- Need efficient merge

H1



H2



Leftist Heap

- Supports o(N) merge, in particular logarithmic
- **Ordering property**: as in regular heaps
- **Structural property**: not enforced explicitely
  - Null path length of node X: **npl(X)** = length of shortest path from node to a node without 2 children

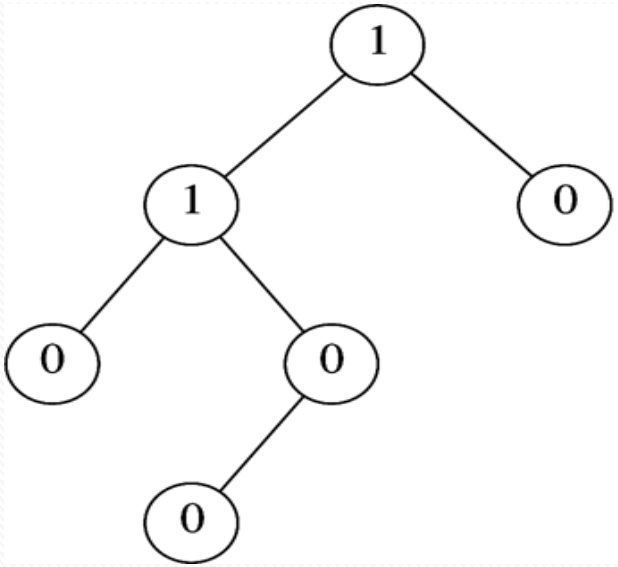
    $$npl(X) = 1 + \min\{npl(children)\}$$
  - npl(empty tree)=-1

In leftist heaps (definition):

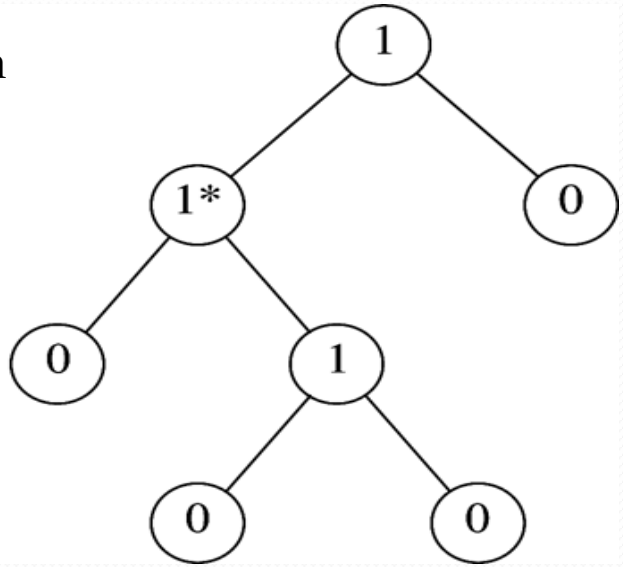
For every node X, the npl() of left child is greater or equal to npl() of right child, therefore:

npl(X)=1+npl(right child)

Leftist heaps



npl() of nodes shown



Leftist

not leftist

Leftist heaps: implementation

- Keep value of npl() at each node.
- Update npl() as necessary.

Theorem

- **Theorem:** A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes [**Proof = ?**]
- Therefore a leftist tree with N nodes total,

  and exactly r nodes on right path

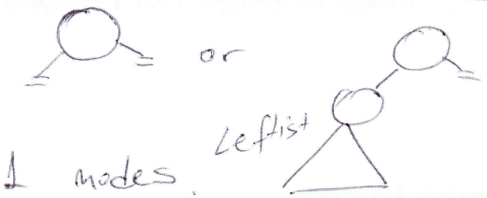
  has $r \leq \log(N+1)$ nodes on right path
- So, right path is short.

=> Merge based on the right path only.

PROOF BY INDUCTION

- A leftist tree w/ r nodes on the right path has at least ($\geq$) $2^r - 1$ nodes.

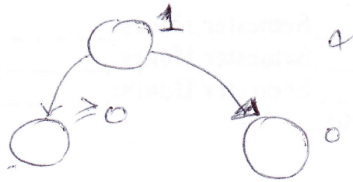
True for r = 1



at least $2^1 - 1 = 1$ nodes.

Suppose property is true for 2, 3, ..., up to r

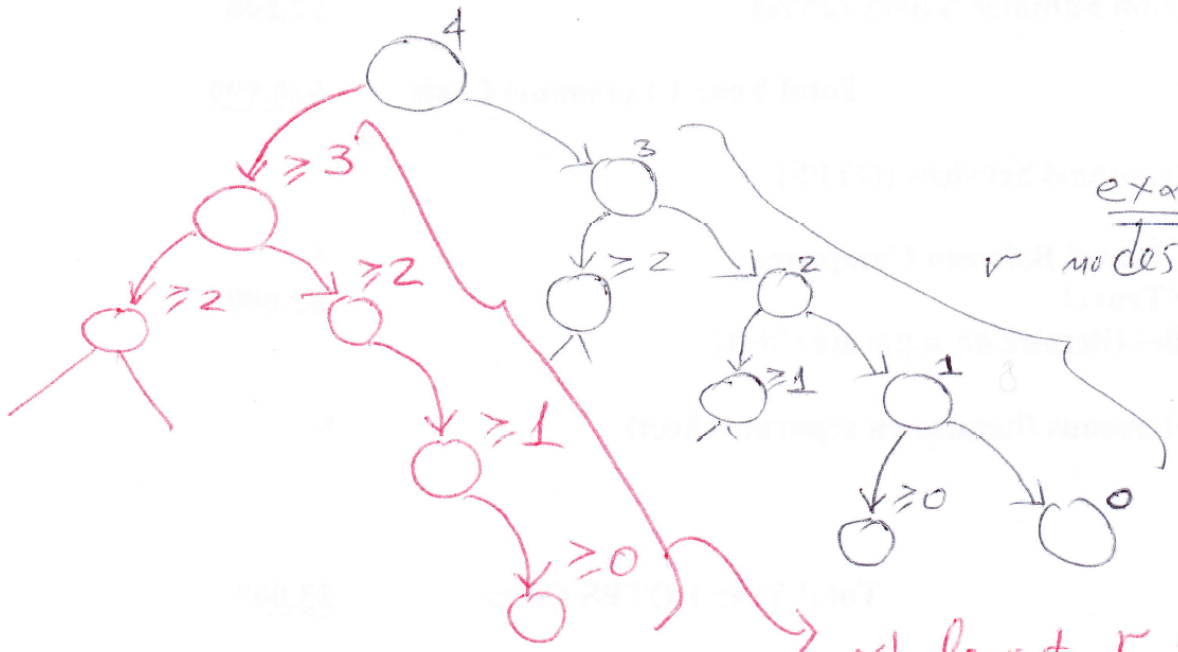
// see for r = 2



at least $2^r - 1 = 3$ nodes.

Consider a tree w/ $r+1$ nodes on right path

(example w/ $r+1 = 5$)



exactly r nodes on right path

at least r nodes on right path

Main idea: $npl(X) = npl(\text{Right child}) + 1$

for leftist trees

Therefore: Right subtree has exactly r nodes on right path $\Rightarrow T_{Right} \geq 2^r - 1$.

Left subtree has $\geq$ [redacted] nodes on right path

$\Rightarrow T_{left} \geq 2^r - 1$ (even stronger than before)

[If left subtree had exactly r nodes on right path we can use inductive hypothesis. If more than r, we can say that now it would have even more nodes]

So [redacted] the tree would have at least ($\geq$)

$1 + 2^r - 1 + 2^r - 1 = 2^{r+1} - 1$ nodes

COMPLETES PROOF

Merge leftist heaps H1, H2

- Recursive algorithm:

(1) If one of them is empty return the other one (basis)

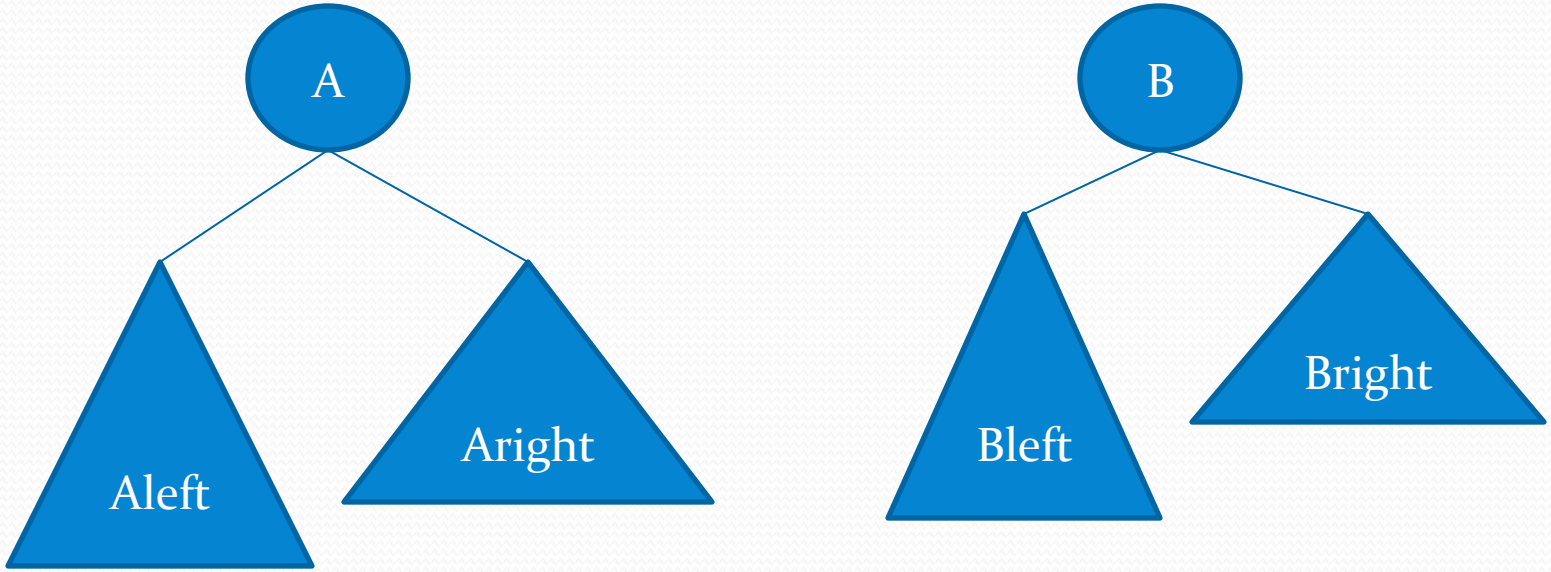
Otherwise,

(2) recursively merge heap with larger root, with the right subheap of the tree with the smaller root. [**Result is a leftist heap**]

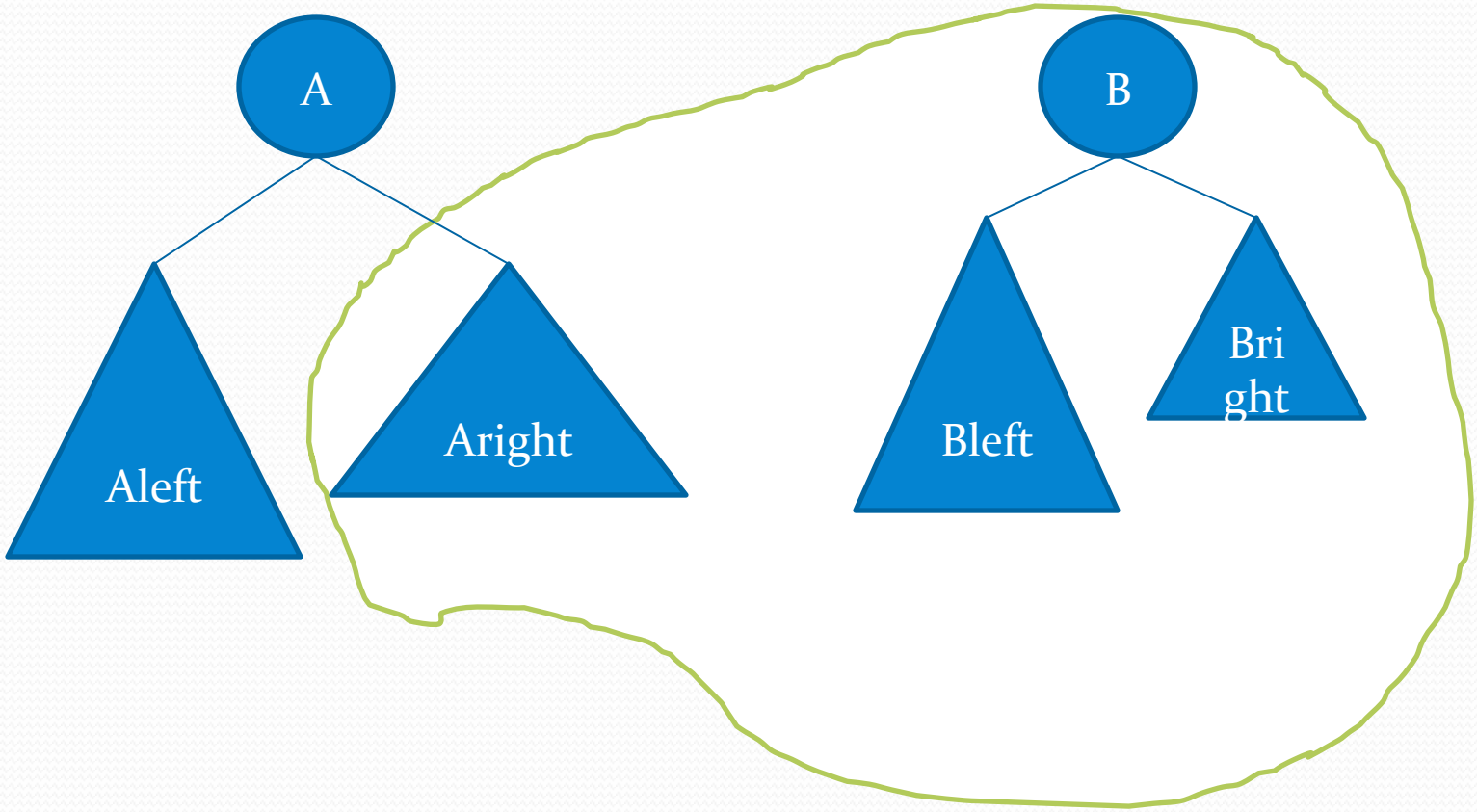
(3) finally make the right child of the heap with the smaller root, the one resulted from step (2)

<Plus: leftist heap restoration via simple swap if needed>

Two leftist heaps, A < B

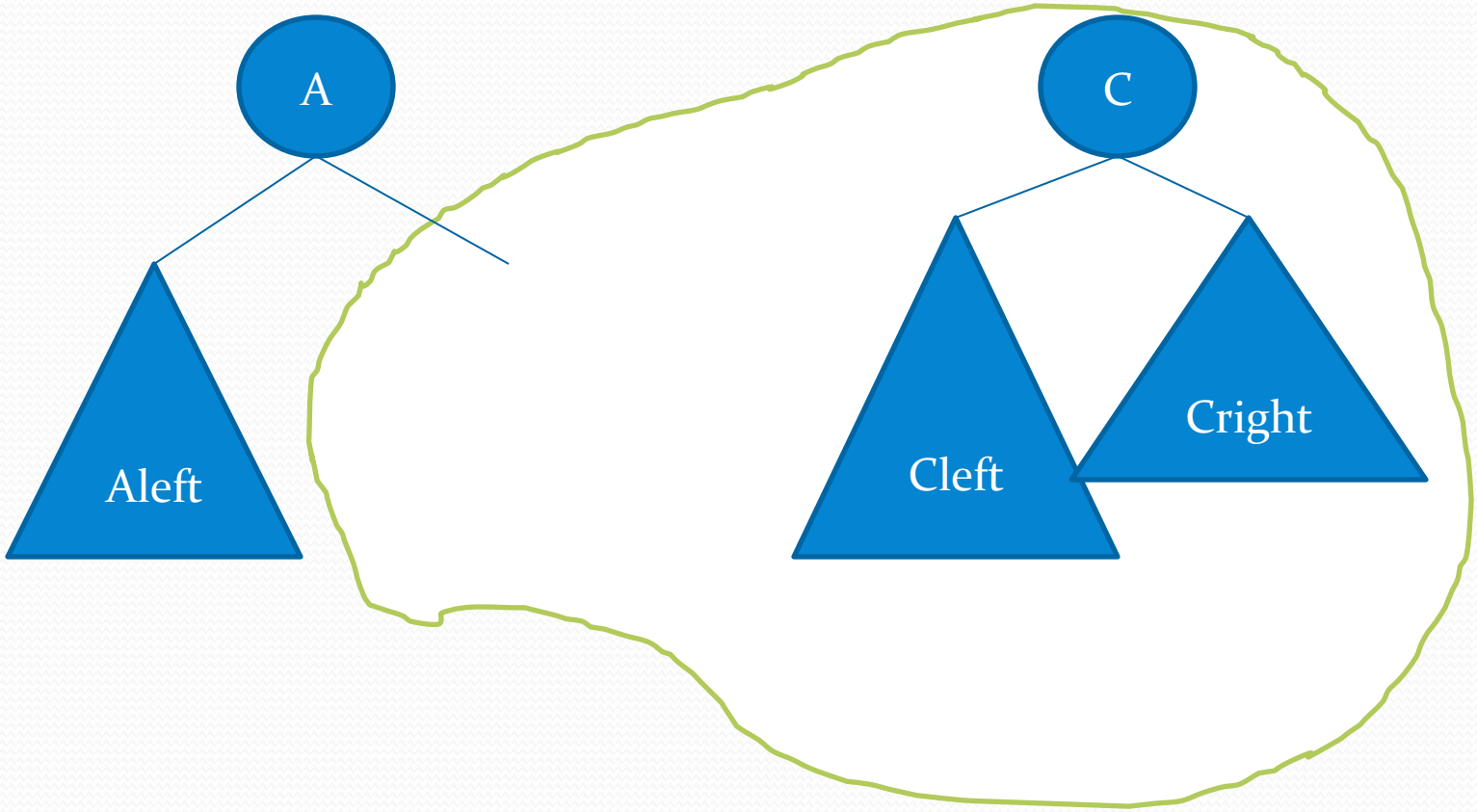


Two leftist heaps, A < B



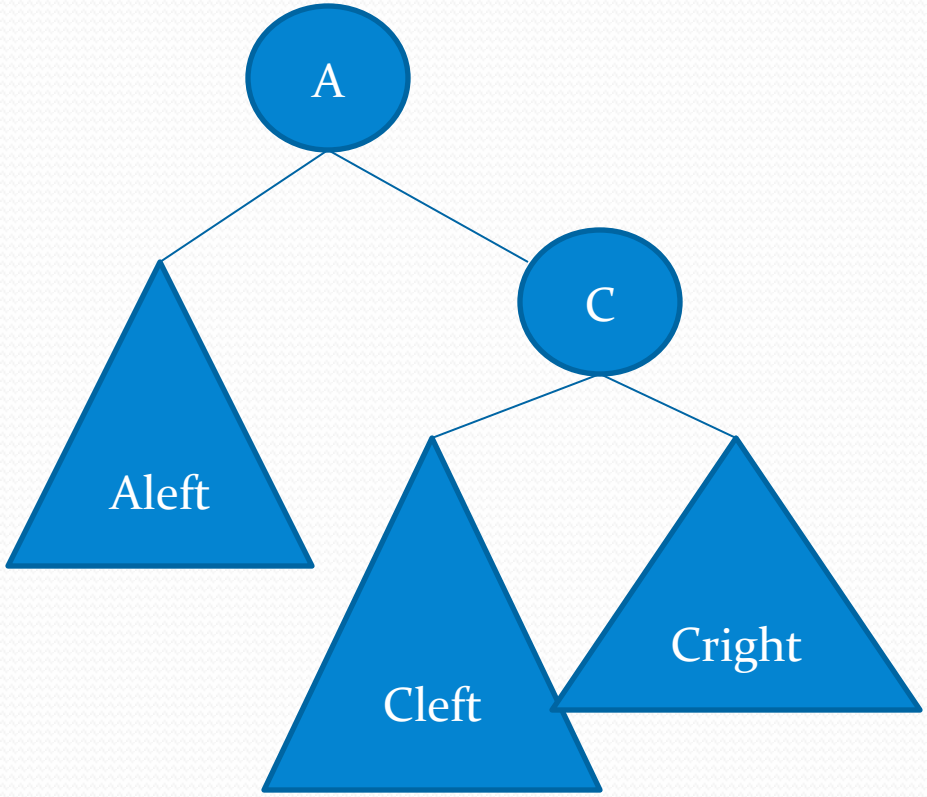
MERGE

Two leftist heaps, A < B



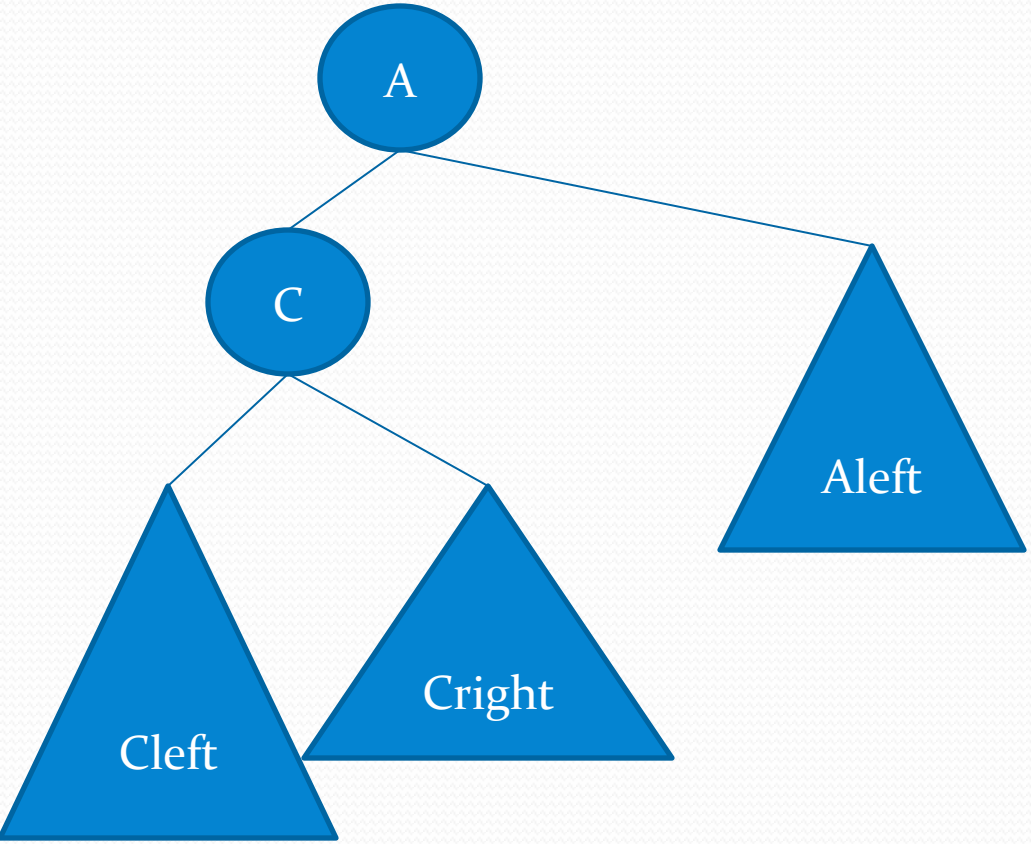
Result of Merge is a leftist heap

Two leftist heaps, A < B



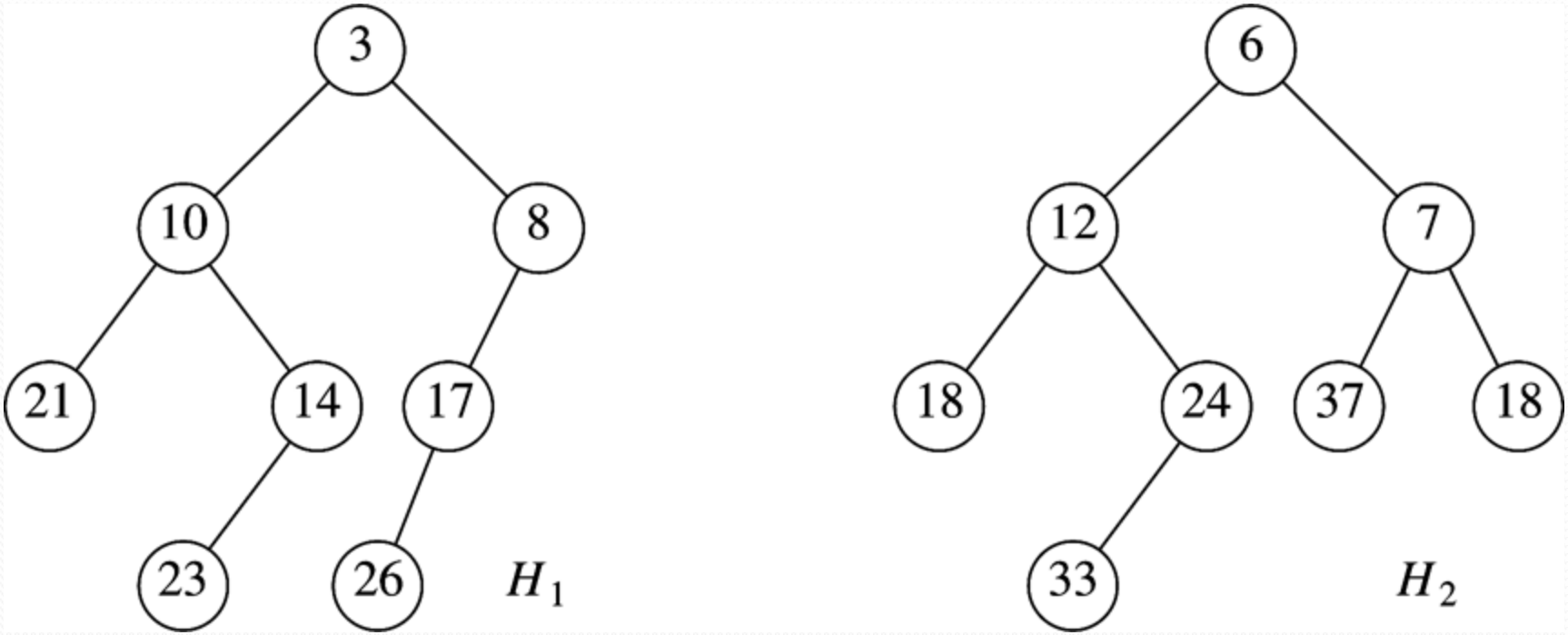
Attach result as right child of A

Two leftist heaps, A < B (root)



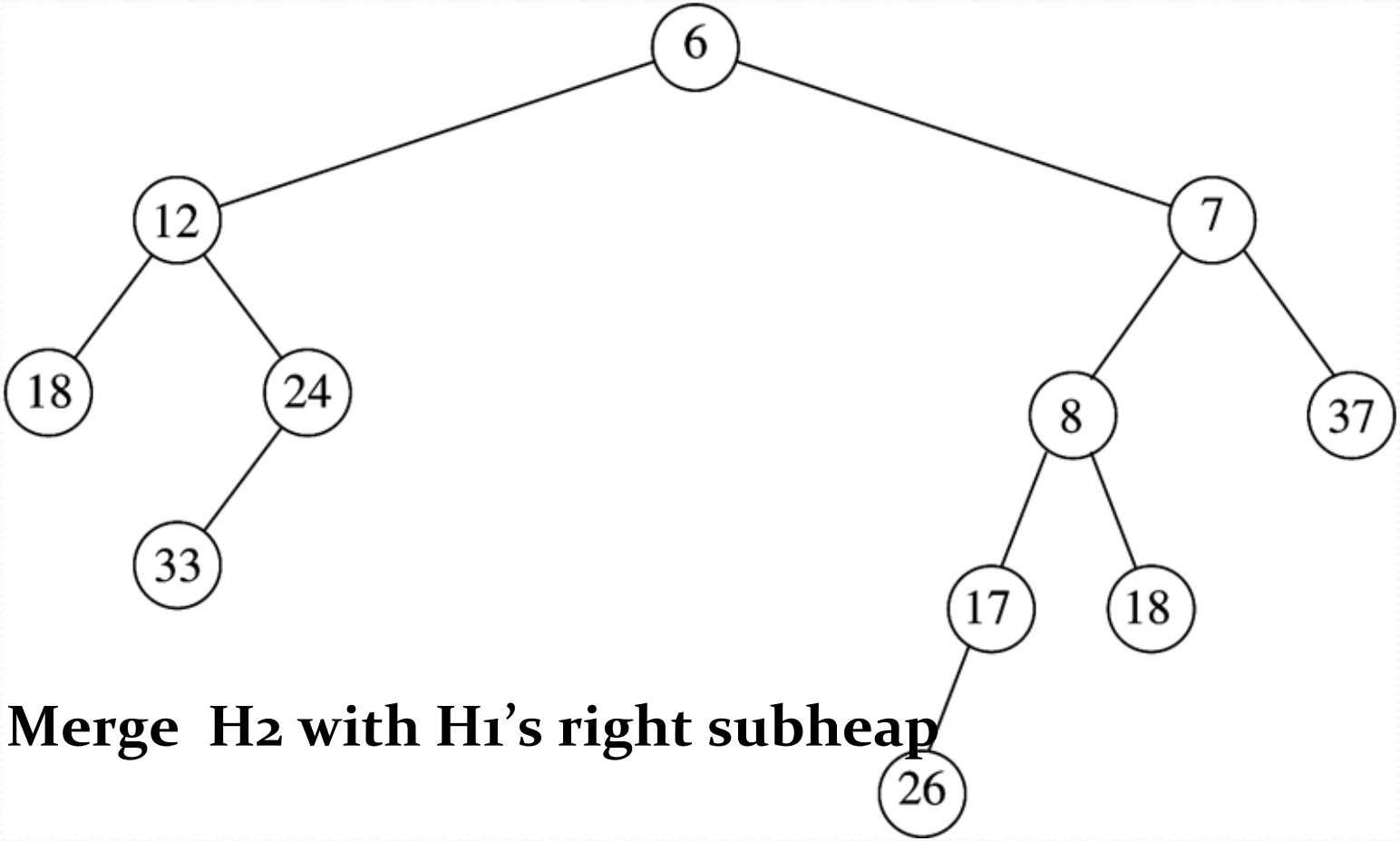
Swap children if needed. Update npl() of root

Merge leftist heaps



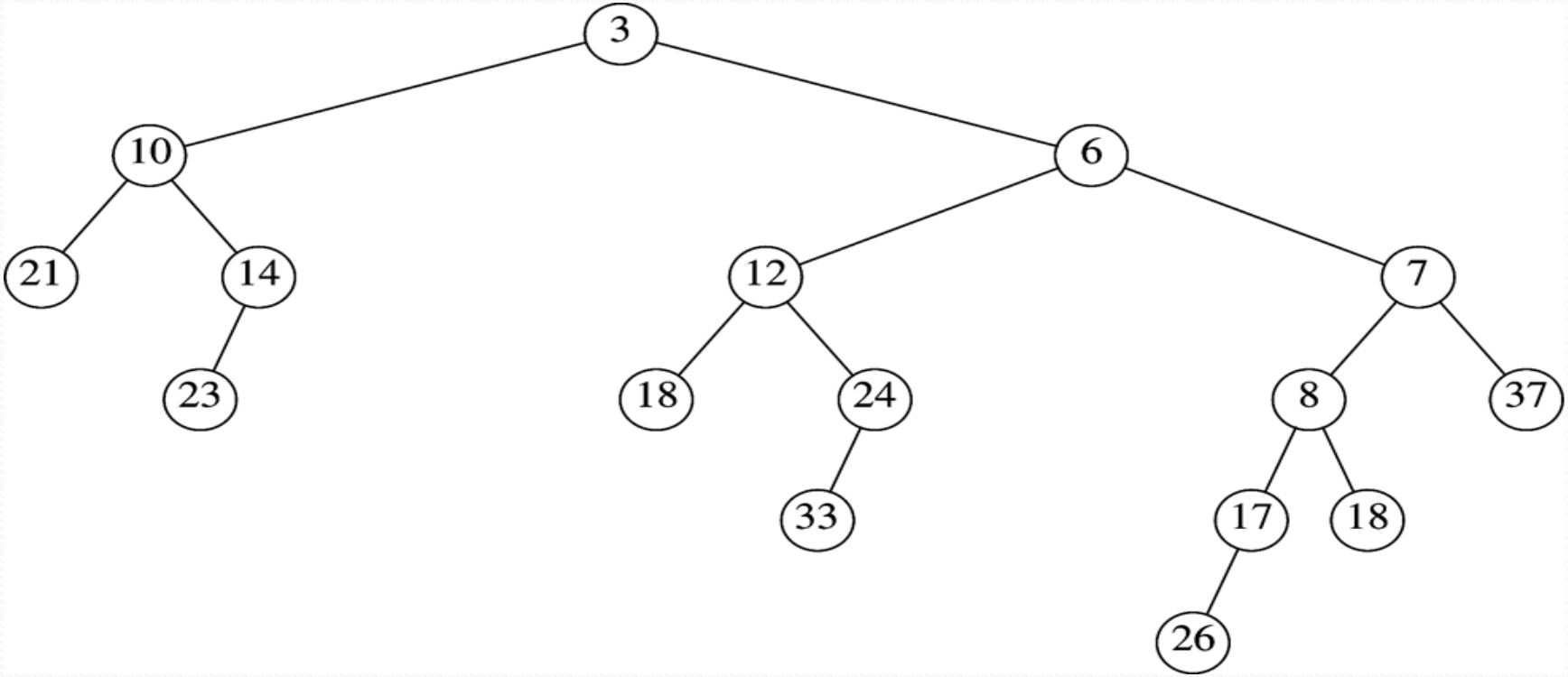
Input

Merge leftist heaps



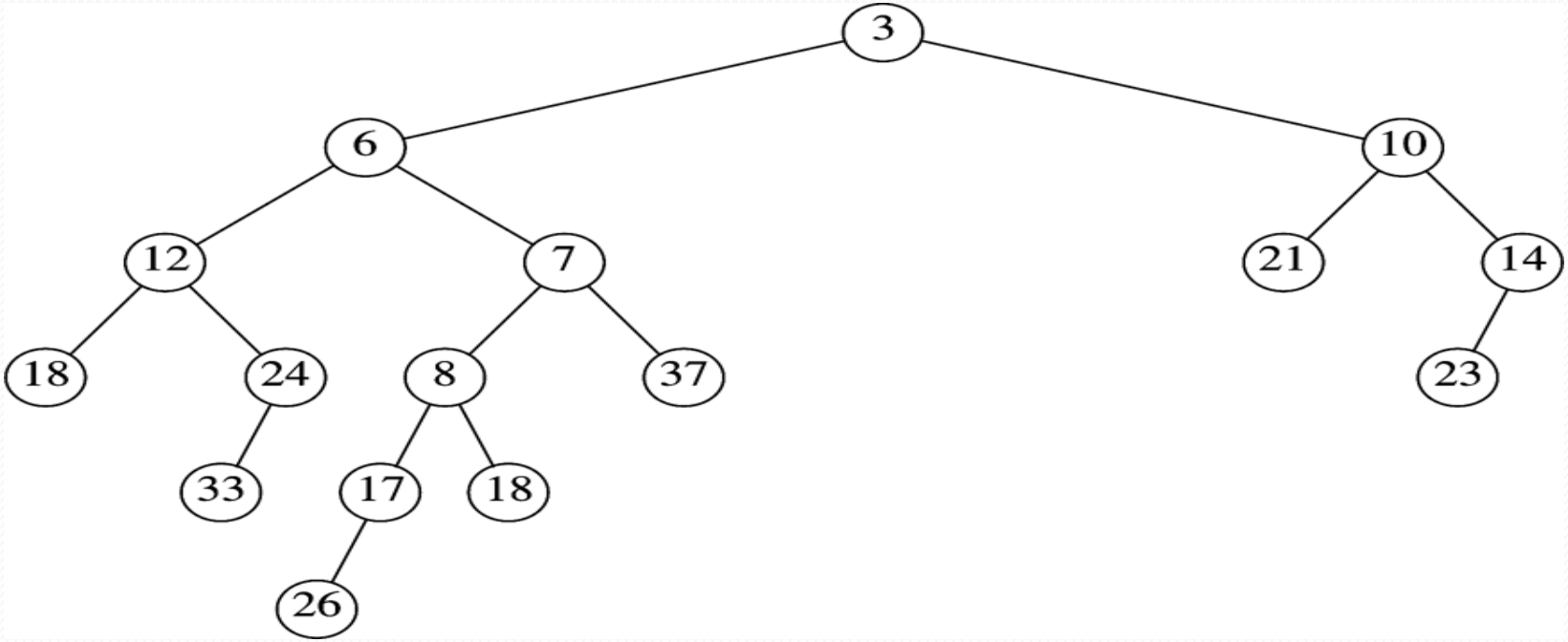
(2) Merge H2 with H1's right subheap

Merge leftist heaps



(3) Attach previous result as right child of H1

Merge leftist heaps



(3) Restore leftist property by swapping children under root
The result of (2) is a leftist heap

Code

```
void merge(LeftistHeap &rhs) {
        if (this == &rhs)    // Avoid aliasing problems
          return;
        root_ = Merge(root_, rhs.root_);
        rhs.root_ = nullptr;
}

LeftistNode *Merge(LeftistNode *h1, LeftistNode *h2)
{
        if (h1 == nullptr)      // Base cases
                return h2;
        if (h2 == nullptr)
                return h1;
        if (h1->element_ < h2->element_)
                return Merge1(h1, h2);
        else
                return Merge1(h2, h1);
}
```

Base case

Recursive merge

Code

Base case (other)

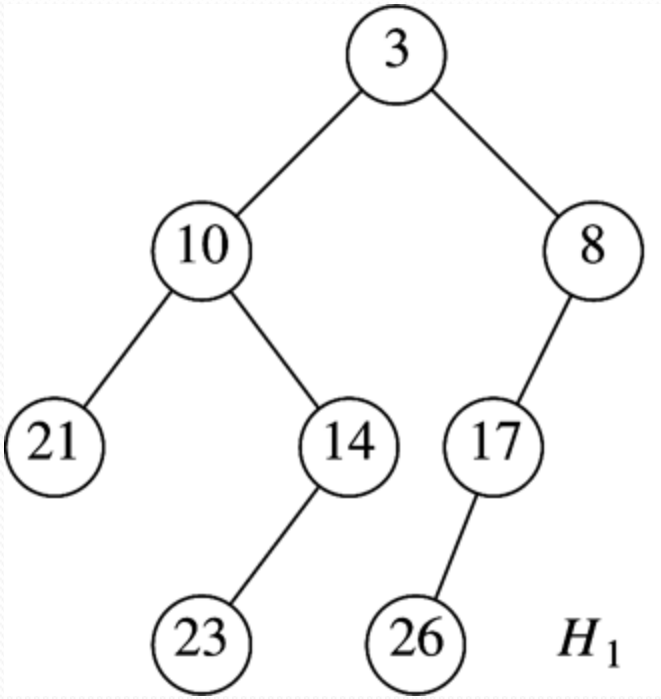
Recursive merge

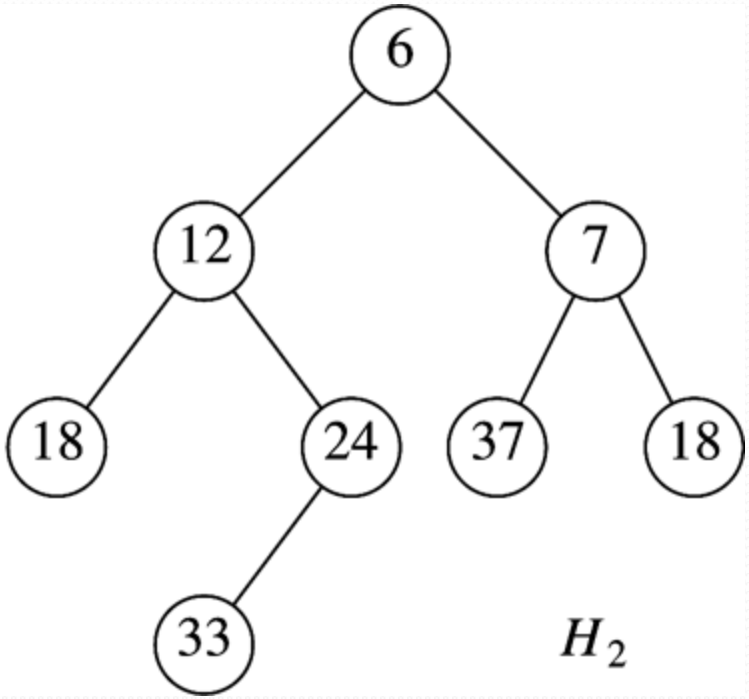
Make leftist

Update npl

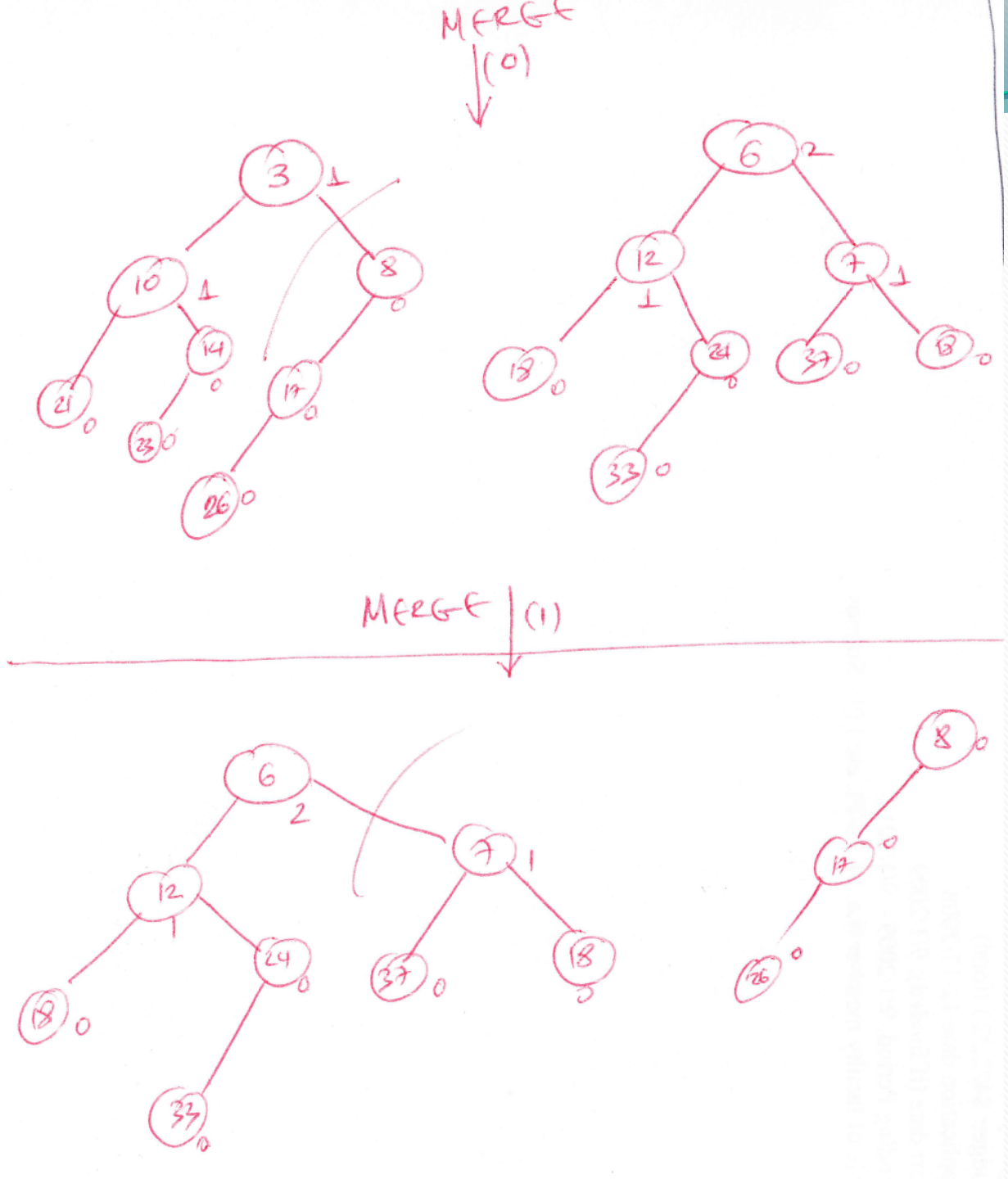
```
1       /**
2        * Internal method to merge two roots.
3        * Assumes trees are not empty, and h1's root contains smallest item.
4        */
5       LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
6       {
7           if( h1->left == NULL )   // Single node
8               h1->left = h2;       // Other fields in h1 already accurate
9           else
10          {
11              h1->right = merge( h1->right, h2 );
12              if( h1->left->npl < h1->right->npl )
13                  swapChildren( h1 );
14              h1->npl = h1->right->npl + 1;
15          }
16          return h1;
17      }
```

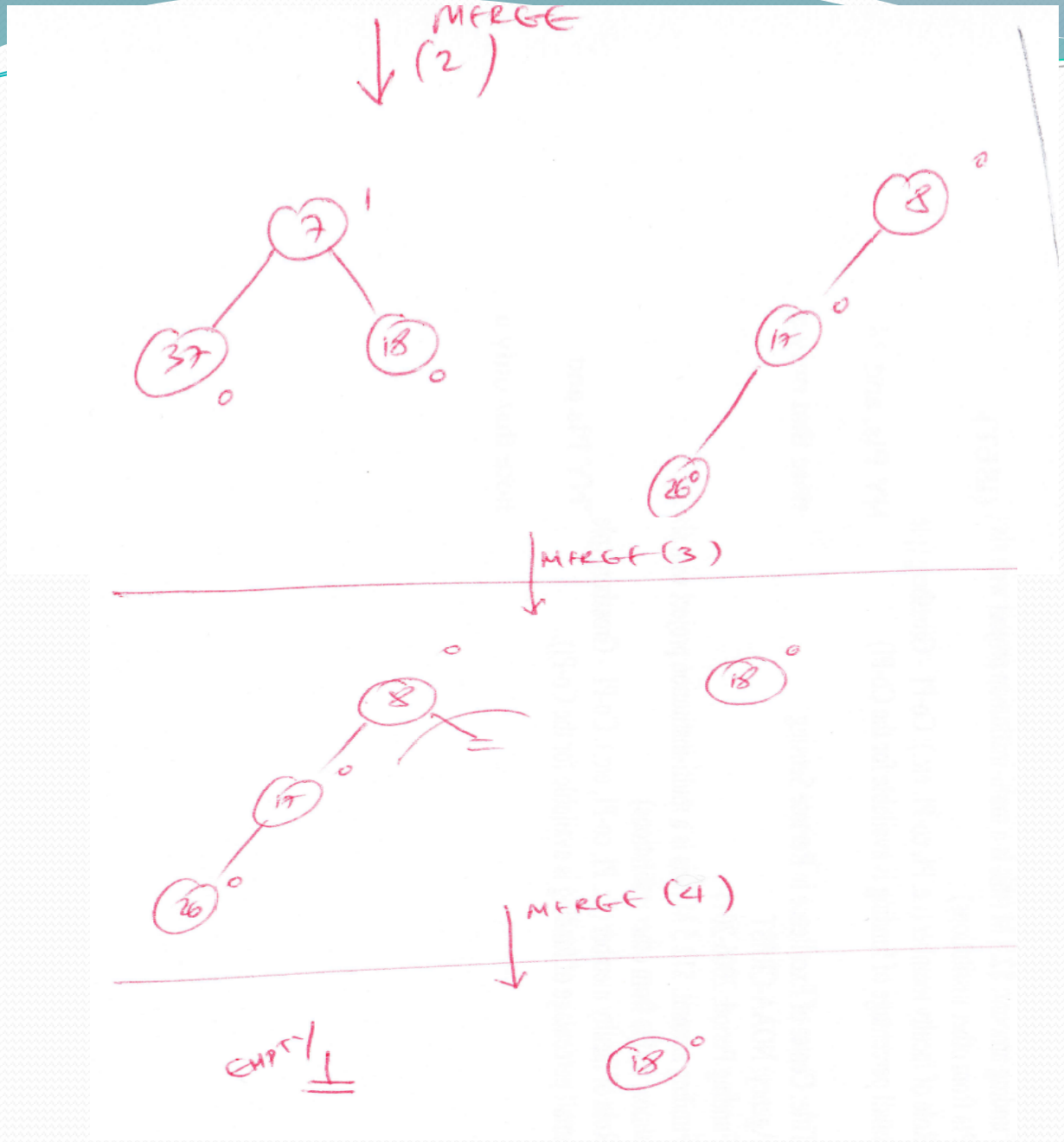
Merge leftist heaps

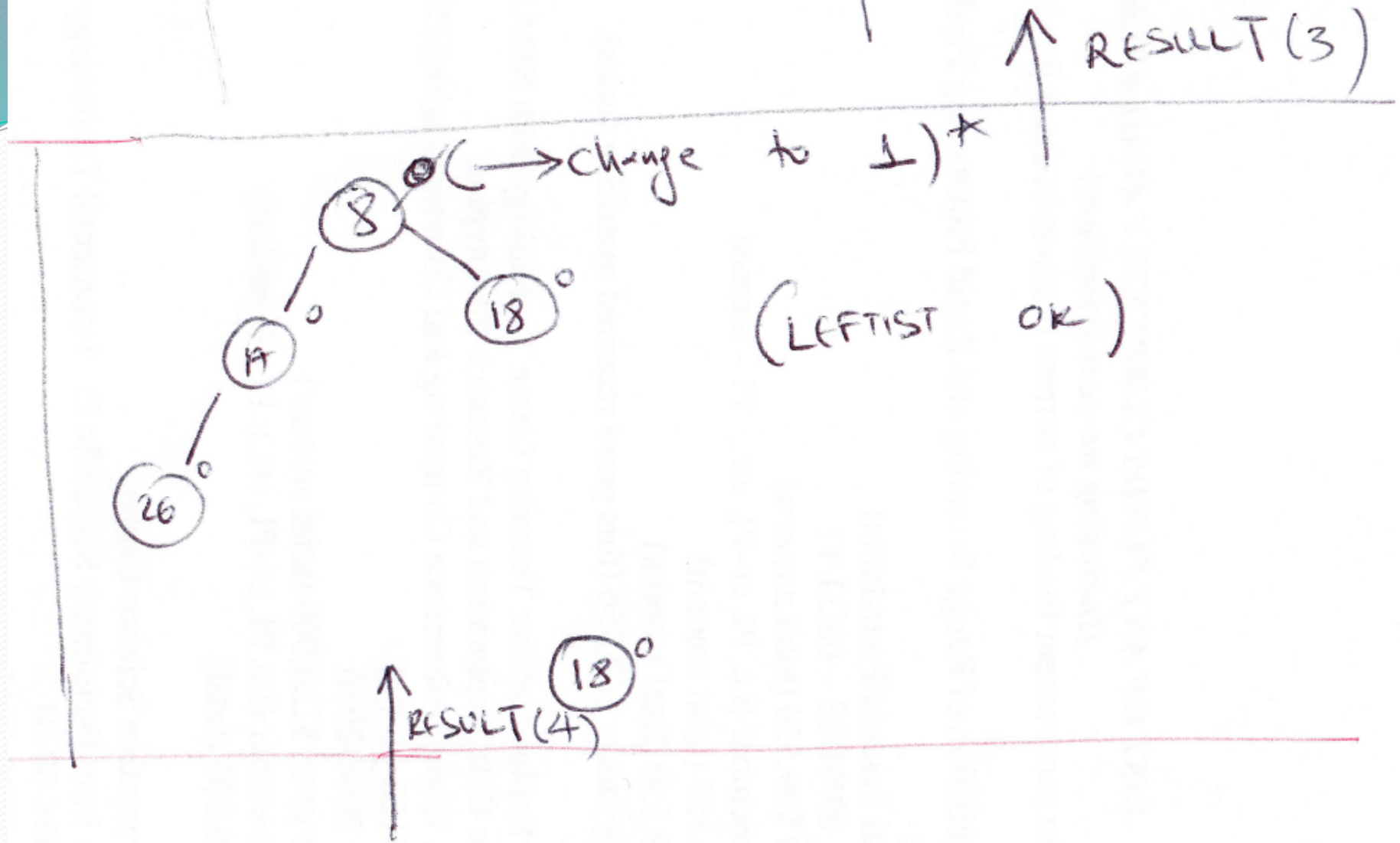


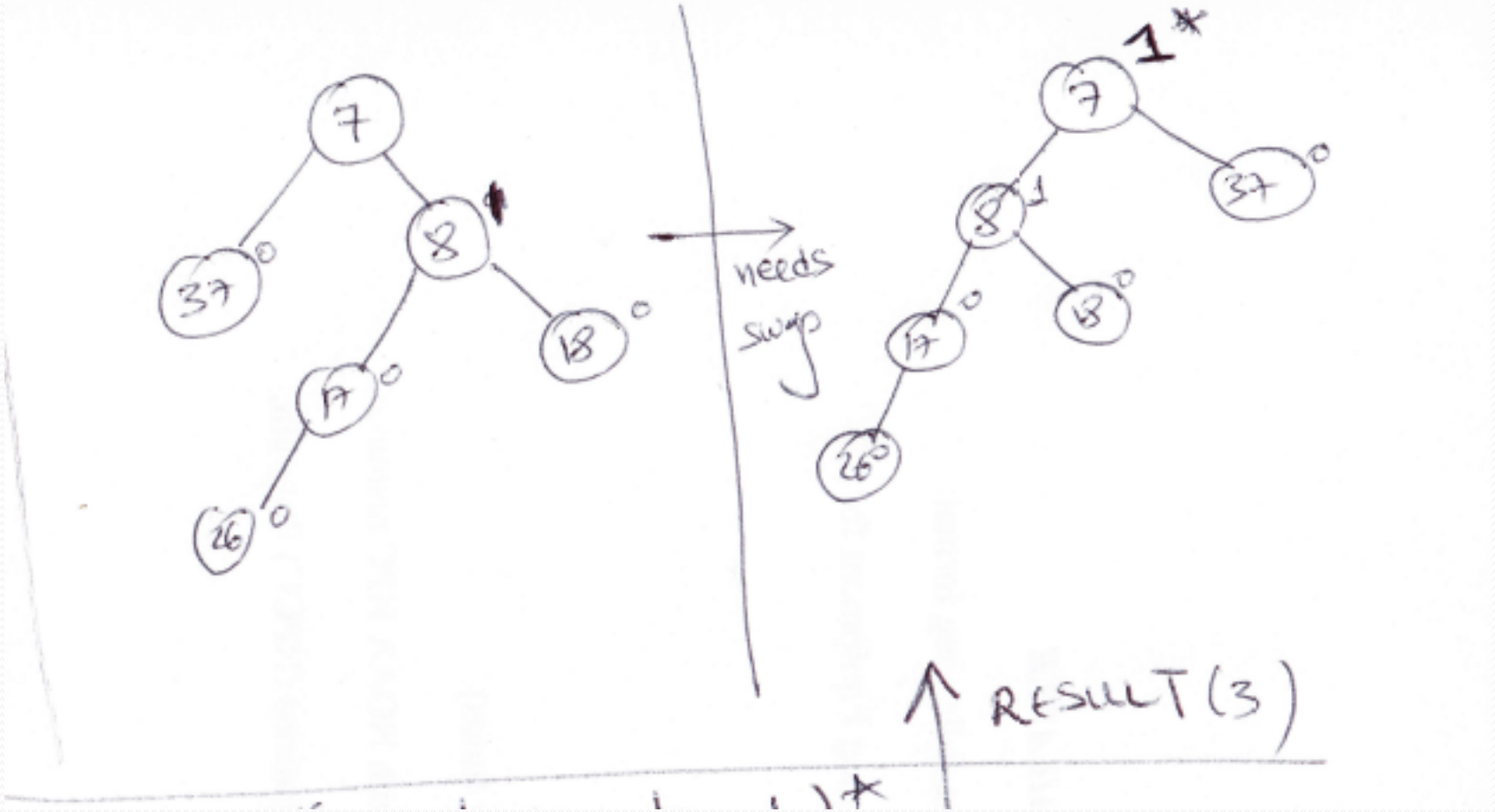


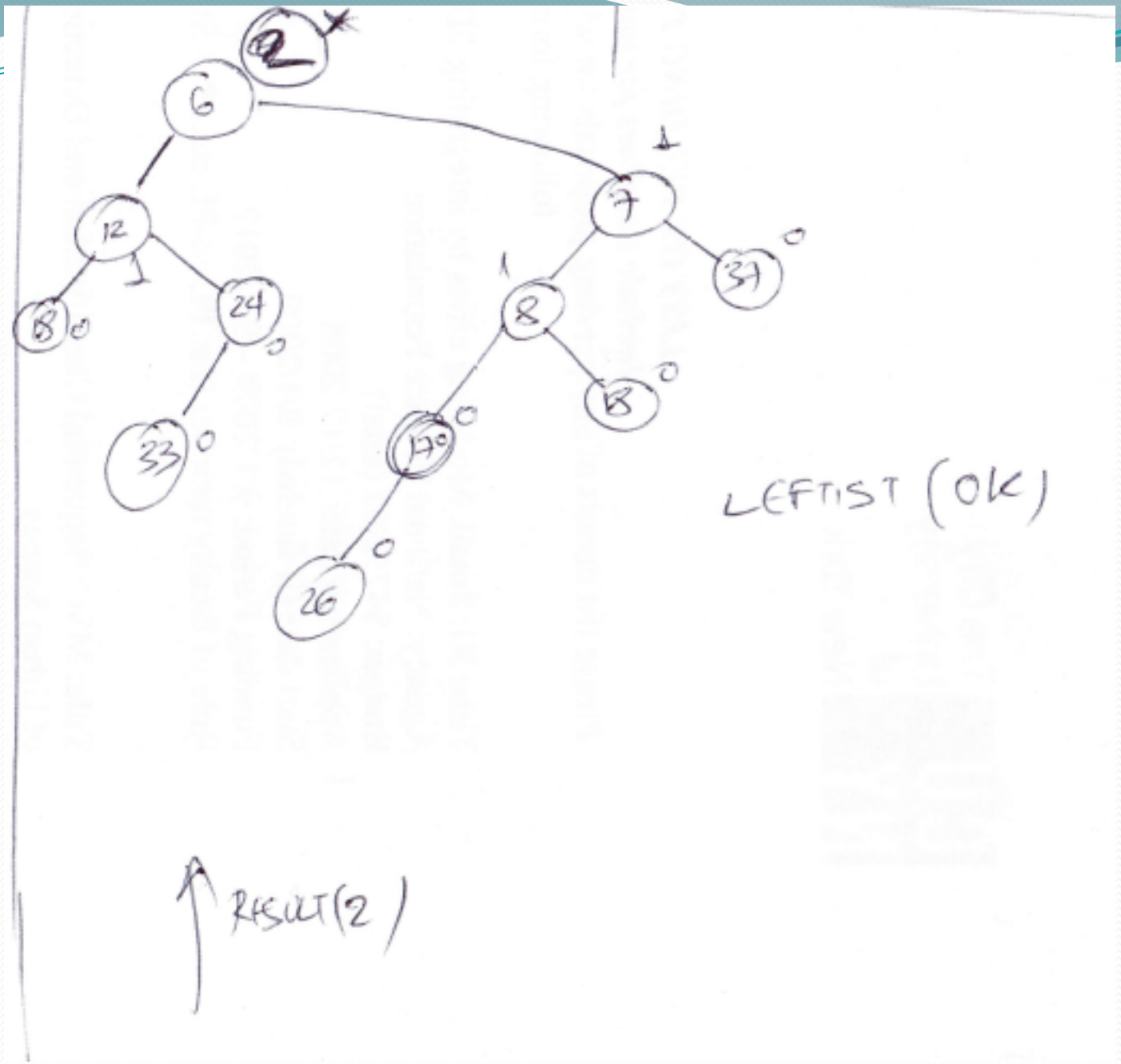
Input {Run through example}

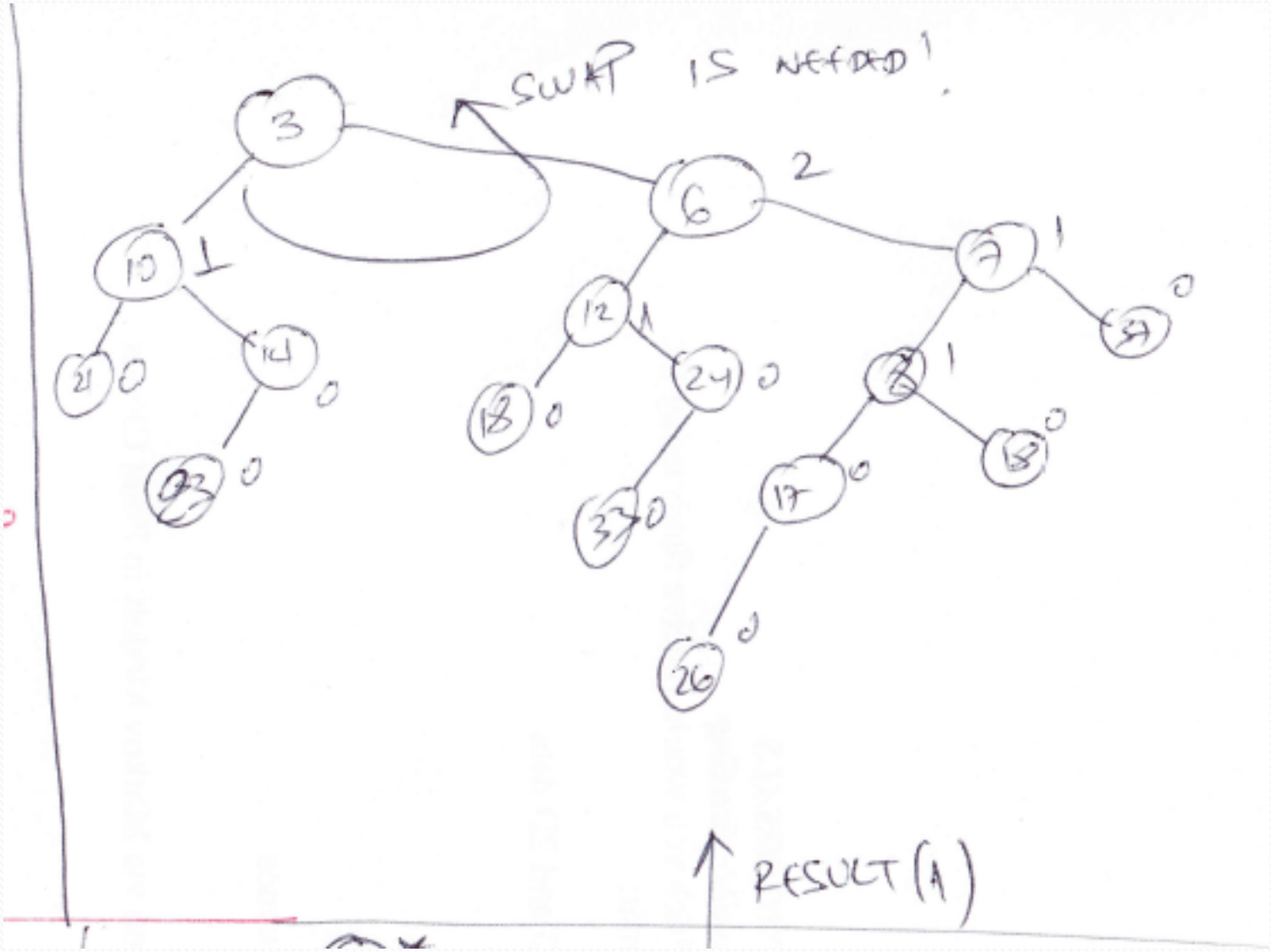

MERGE
(0)
3
1
10
1
8
0
14
0
21
0
23
0
17
0
26
0
6
2
12
1
7
1
18
0
24
0
37
0
13
0
33
0
MERGE
(1)
6
2
12
1
24
0
18
0
33
0
7
1
37
0
18
0
8
0
17
0
26
0


MERGE
(2)
7
37
18
8
17
26
MERGE (3)
8
17
26
18
MERGE (4)
EMPTY
18

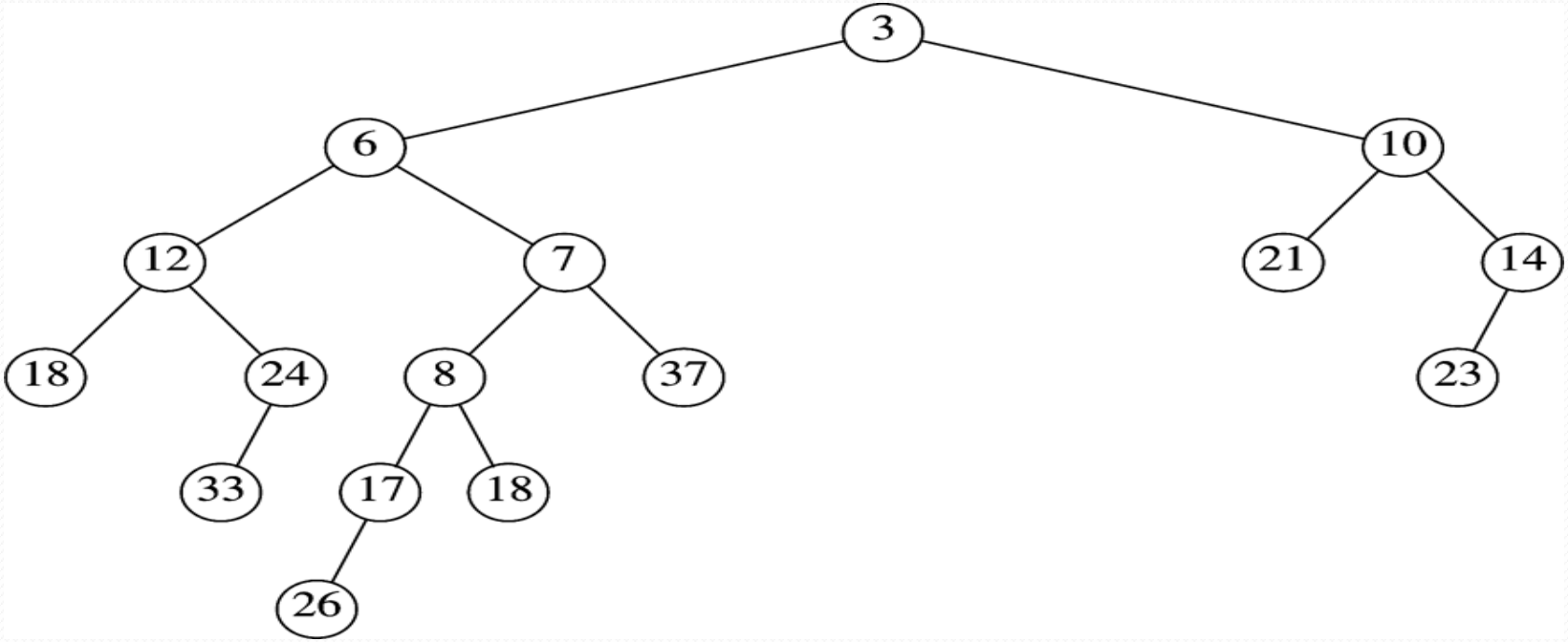

RESULT(3)
(→change to 1)*
8
18
4
26
(LEFTIST OK)
18
RESULT(4)


7
37
8
18
17
26
needs
swap
1*
7
8
37
18
17
26
RESULT (3)


2
6
7
1
12
1
37
0
8
1
8
0
24
0
13
0
33
0
19
0
26
0
LEFTIST (OK)
RESULT (2)


SWAP IS NEEDED!
3
6
2
10
1
12
1
7
1
21
0
14
0
18
0
24
0
9
0
23
0
33
0
17
0
26
0
RESULT (1)

Final Result



Merge leftist Heaps

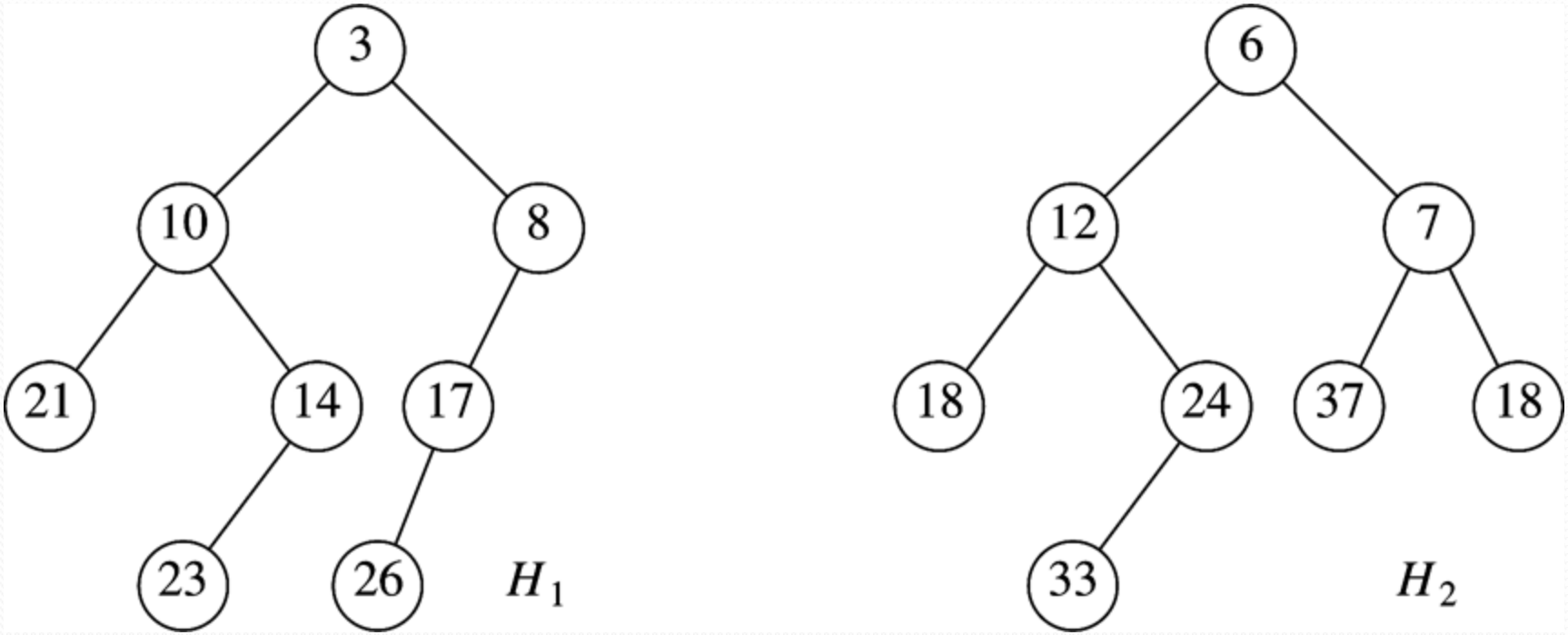
- Non-recursive implementation:

Pass 1) Arrange nodes of right paths of H1, and H2 in sorted order, keeping their respective left children

Pass 2) Bottom-up -> swap children that violate leftist property

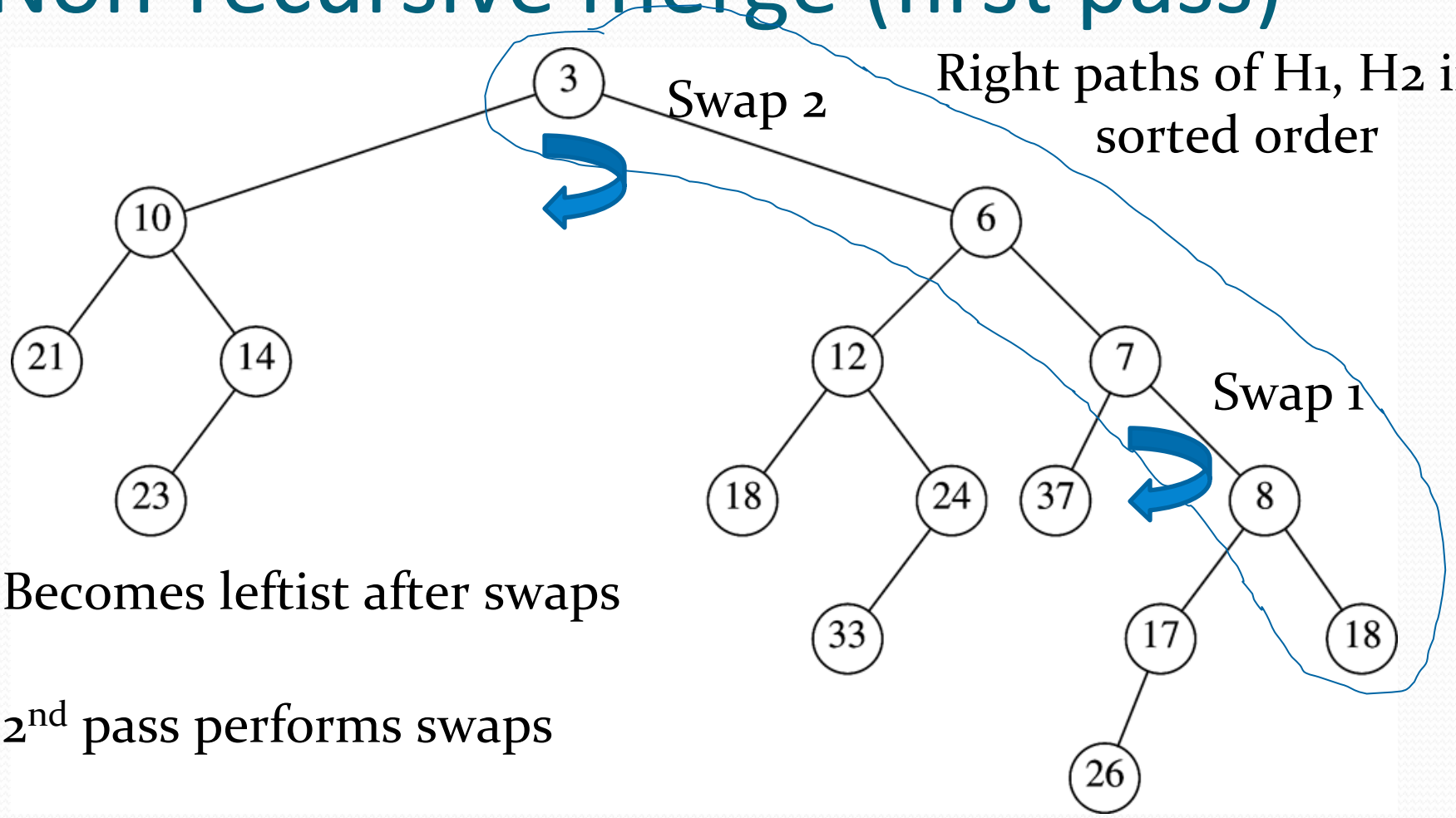
Simple to view

Merge leftist heaps



Input

Non-recursive merge (first pass)



Becomes leftist after swaps

2nd pass performs swaps

Summary (leftist heaps)

- Merge is an O(logN) operation
- Insert, DeleteMin ?

Summary (leftist heaps)

- Merge is an O(logN) operation
- Insert, deleteMin ?

  Insert: Merge current heap with a single-node heap

  deleteMin: Destroy root, merge two subheaps
- So Insert, deleteMin: O(logN)