Data Structures and Algorithms

Lecture 11 – Priority Queues-Binary Heaps

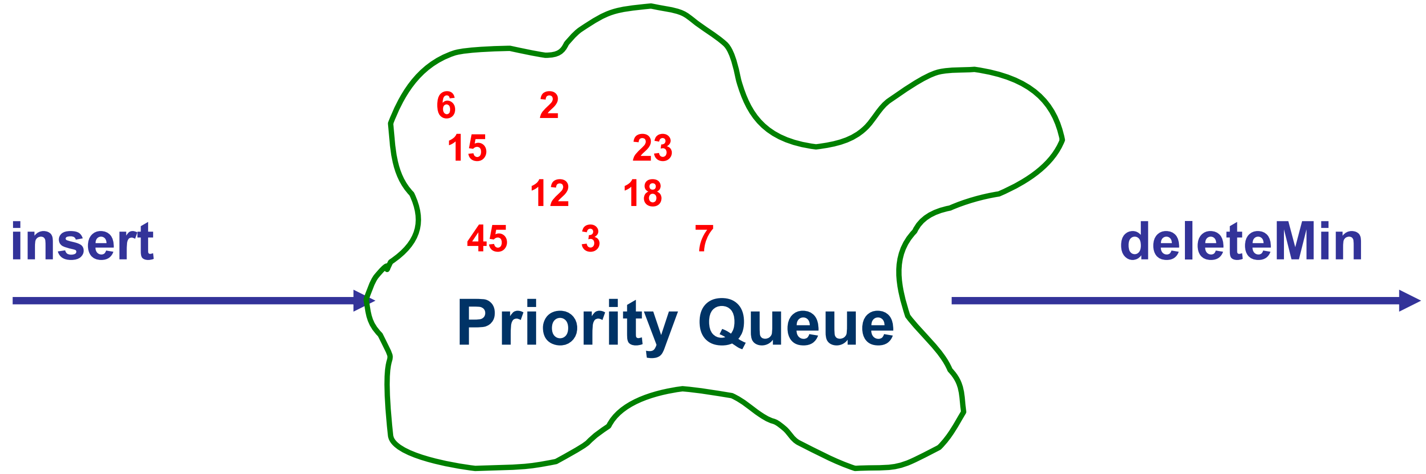
Miao Zhang

Priority Queues

- **Many applications require that we process records with keys in order, but not necessarily in full sorted order.**
- **Often we collect a set of items and process the one with the current minimum value.**
  - **e.g. Cases in a hospital emergency room**
  - **Operating system job scheduler in a multi-user environment**
  - **Simulation environments**
- **An appropriate data structure is called a *priority queue*.**

Definition

- A priority queue is a data structure that supports two basic operations: <u>insert a new item</u> and <u>remove the highest priority item.</u>



Priority Queues

- **A priority queue is a collection of zero or more elements → each element has a priority or value**
- **Unlike the FIFO queues, the order of deletion from a priority queue (e.g., who gets served next) is determined by the element priority**
- **Elements are deleted by increasing or decreasing order of priority rather than by the order in which they arrived in the queue**

Priority Queues

- Operations performed on priority queues
  1) Find an element,
  2) insert a new element,
  3) delete an element, etc.
- Two kinds of (Min, Max) priority queues exist
- In a Min priority queue, find/delete operation finds/deletes the element with minimum priority
- In a Max priority queue, find/delete operation finds/deletes the element with maximum priority

Simple Implementations

- A simple linked list:
  - Insertion at the front (O(1)); delete minimum (O(N)), or
  - Keep list sorted; insertion O(N), deleteMin O(1)
- A binary search tree:
  - This gives an O(log N) average for both operations.
  - But BST class supports a lot of operations that are not required.
- Binary Heap
  - will support both operations in O(logN) wost-case time.

Binary Heap

- The binary heap is the classic method used to implement priority queues.
- We use the term heap to refer to the binary heap.

- Heap has two properties:
  - Structure property
  - Ordering property

Structure Property

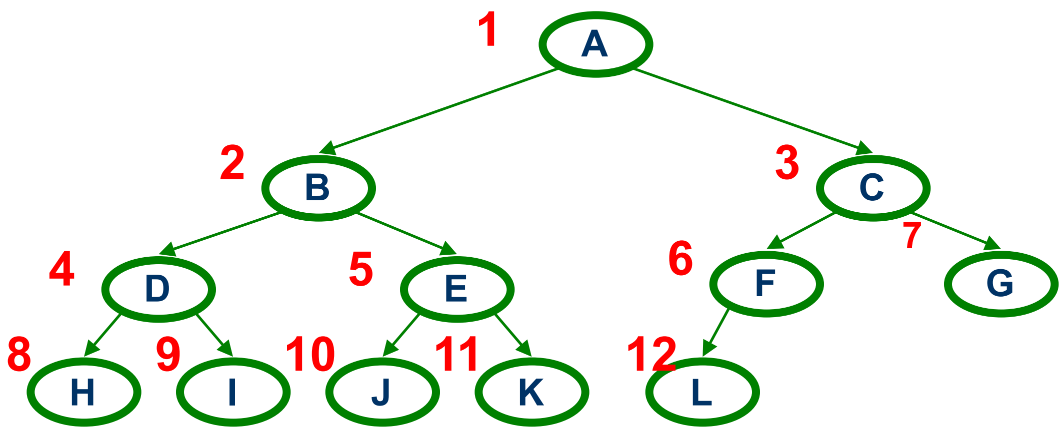
- A heap is a ***complete binary tree***, represented as an array.

A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Properties of a complete binary tree

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
- The height of a complete binary tree is $\lfloor \log N \rfloor$.
- It can be implemented as an array such that:
  - For any element in array position i :
    - the left child is in position $2i$,
    - the right child is in the cell after the left child $(2i + 1)$, and
    - the parent is in position $\lfloor i/2 \rfloor$.

Array Representing of A Complete Binary Tree



From node i:

left child:
right child:
parent:

implicit (array) implementation:

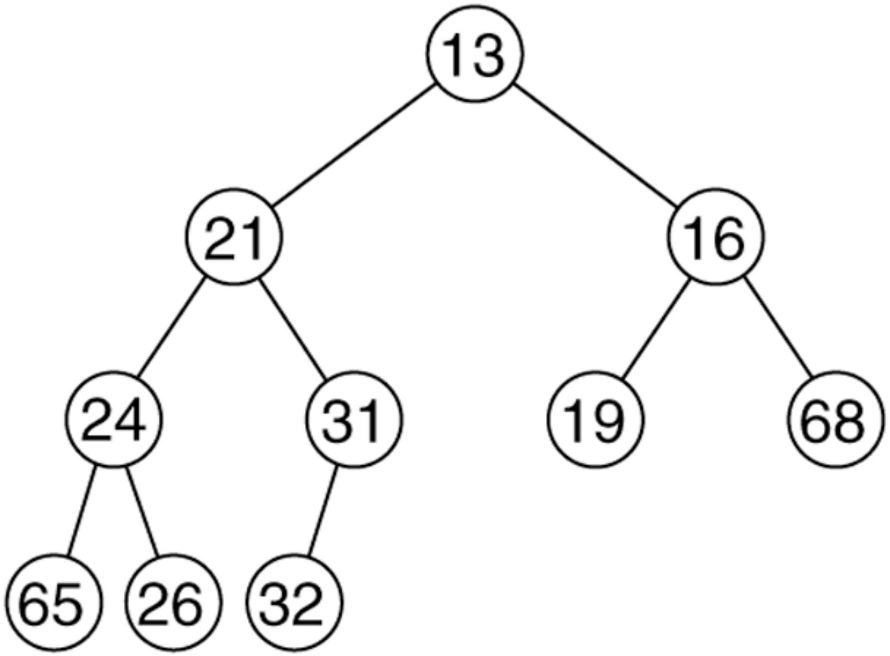
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-Order Property

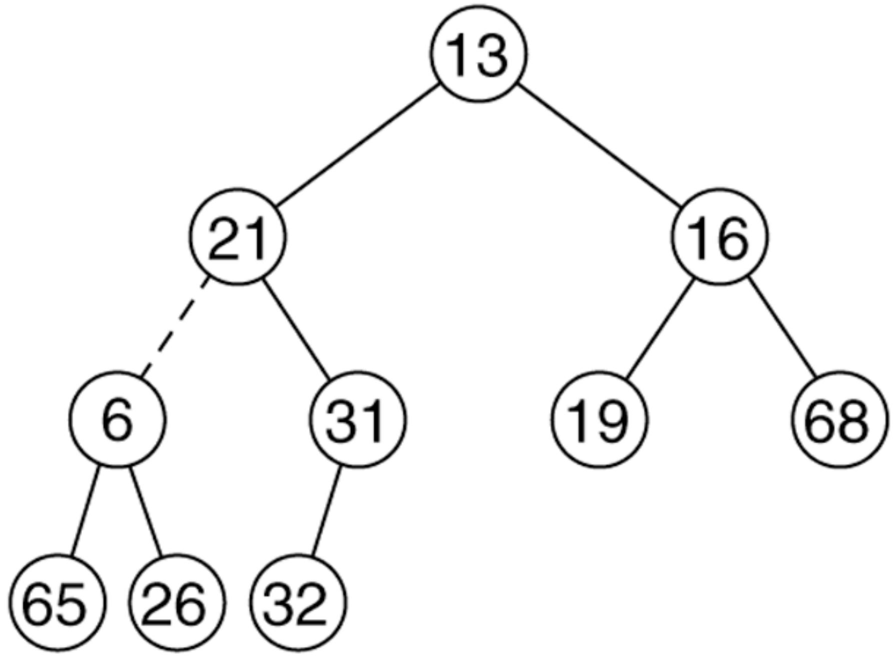
- In a min-heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.
- Thus the minimum element is always at the root.
  - Thus we get the extra operation findMin in constant time.
- A max-heap supports access of the maximum element instead of the minimum, by changing the heap property slightly.

Examples

Two complete trees: (a) a min-heap; (b) not a heap



(a)



(b)

Min-Heap – Two views

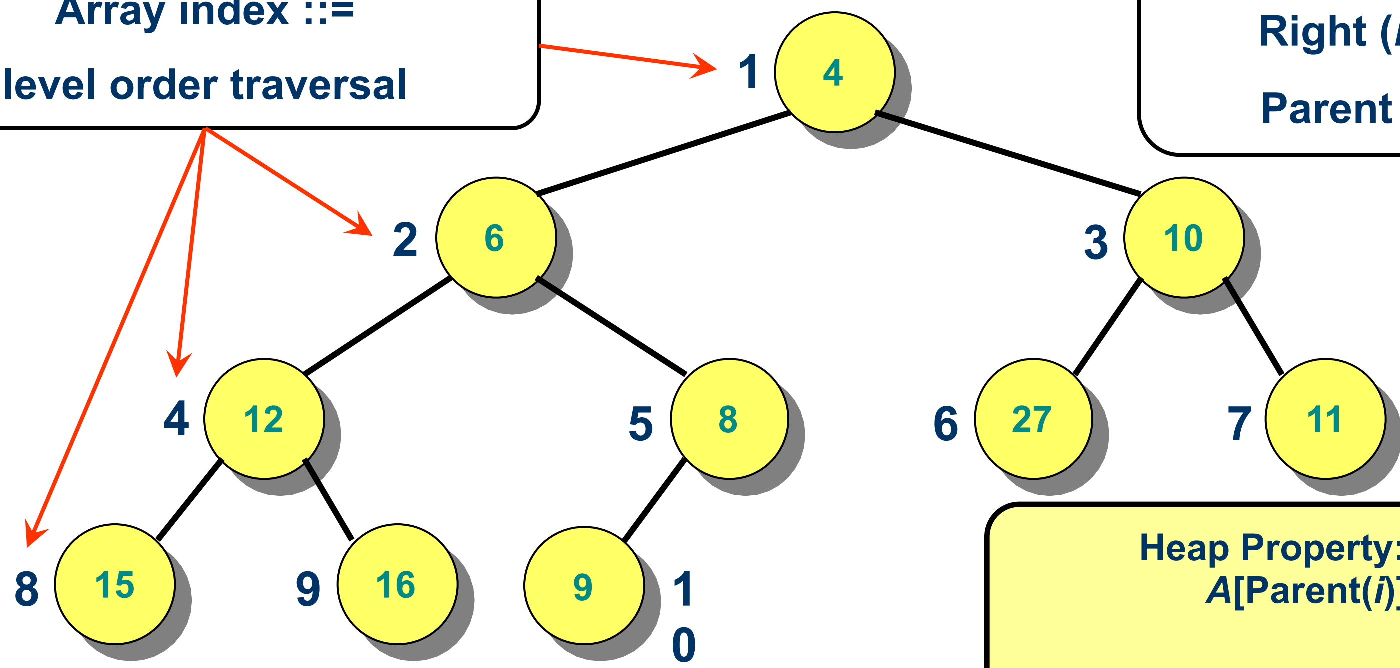
Array index ::=

level order traversal

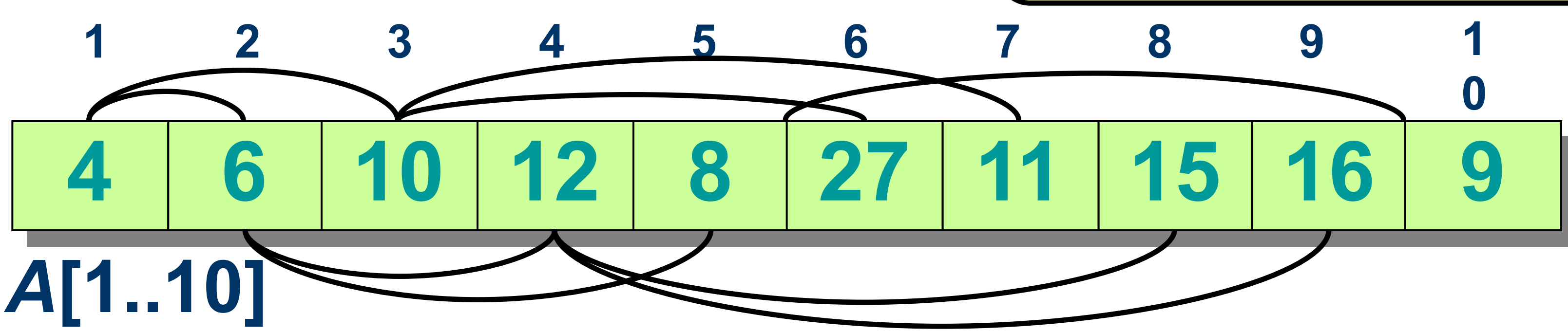
Left (i) = $2i$

Right (i) = $2i + 1$

Parent (i) = $\lfloor i/2 \rfloor$

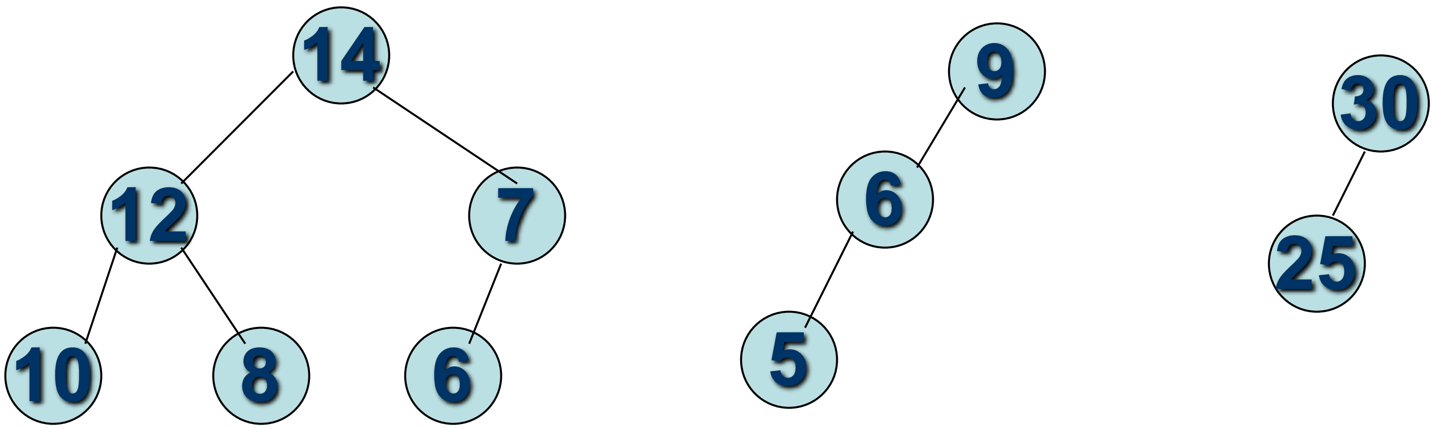


Heap Property: for all i
$A[\text{Parent}(i)] \le A[i]$

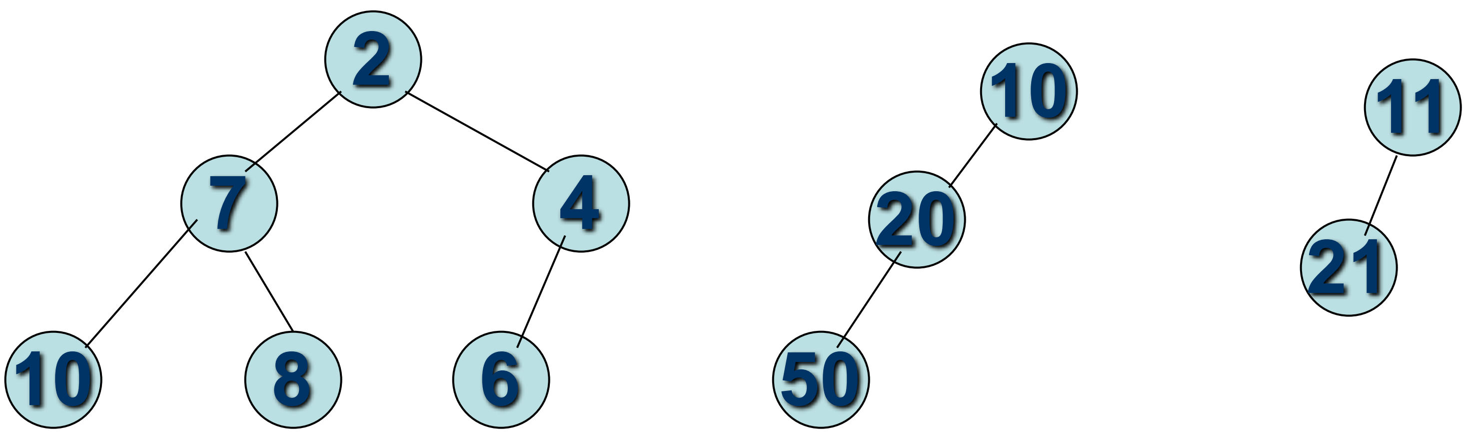


$A[1..10]$

Heaps



Max-Heap?



Min-Heap?

Heap Operations

- Insert
- Delete Min
- Build Heap

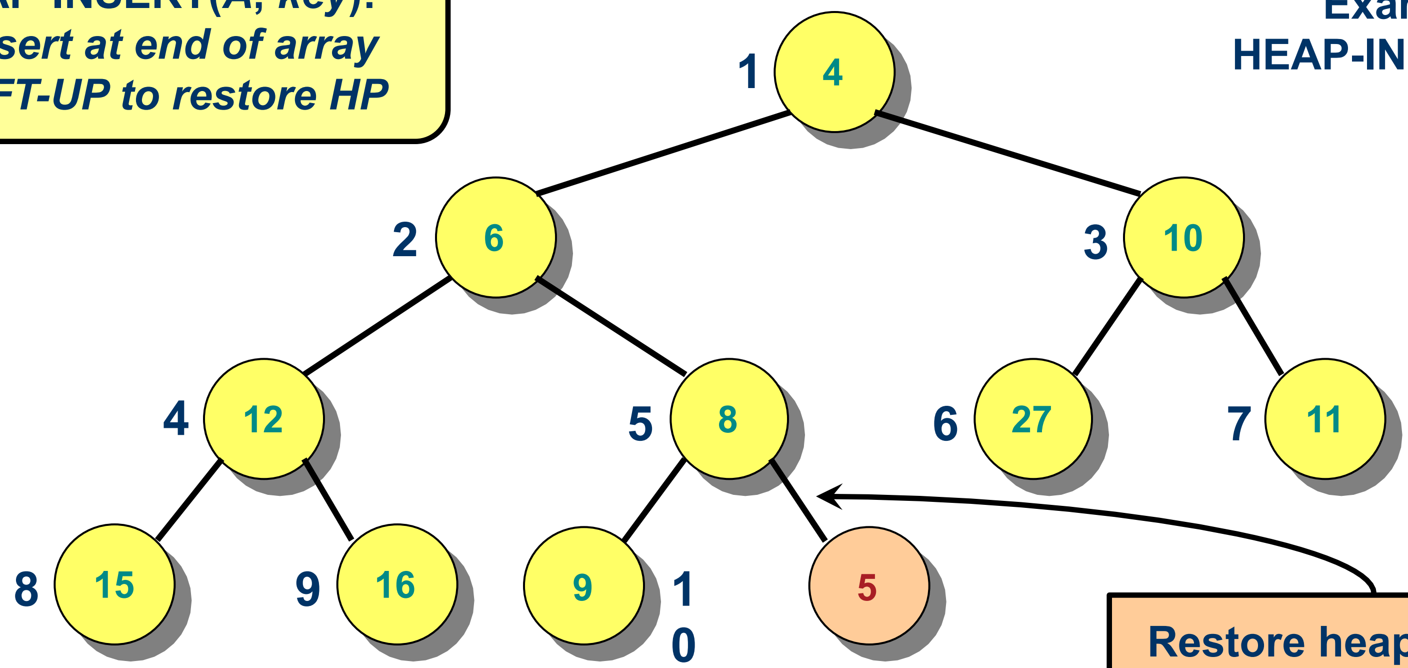
Basic Heap Operations: Insert

- To insert an element X into the heap:
  - We create a hole in the next available location.
  - If X can be placed there without violating the heap property, then we do so and are done.
  - Otherwise
    - we bubble up the hole toward the root by sliding the element in the hole's parent down.
    - We continue this until X can be placed in the hole.
- This general strategy is known as a *percolate up*.

Min-Heap – Insert Operation (1)

HEAP-INSERT(*A*, *key*):
insert at end of array
SIFT-UP to restore HP

Example:
HEAP-INSERT(*A*, 5)



Restore heap property with SIFT-UP

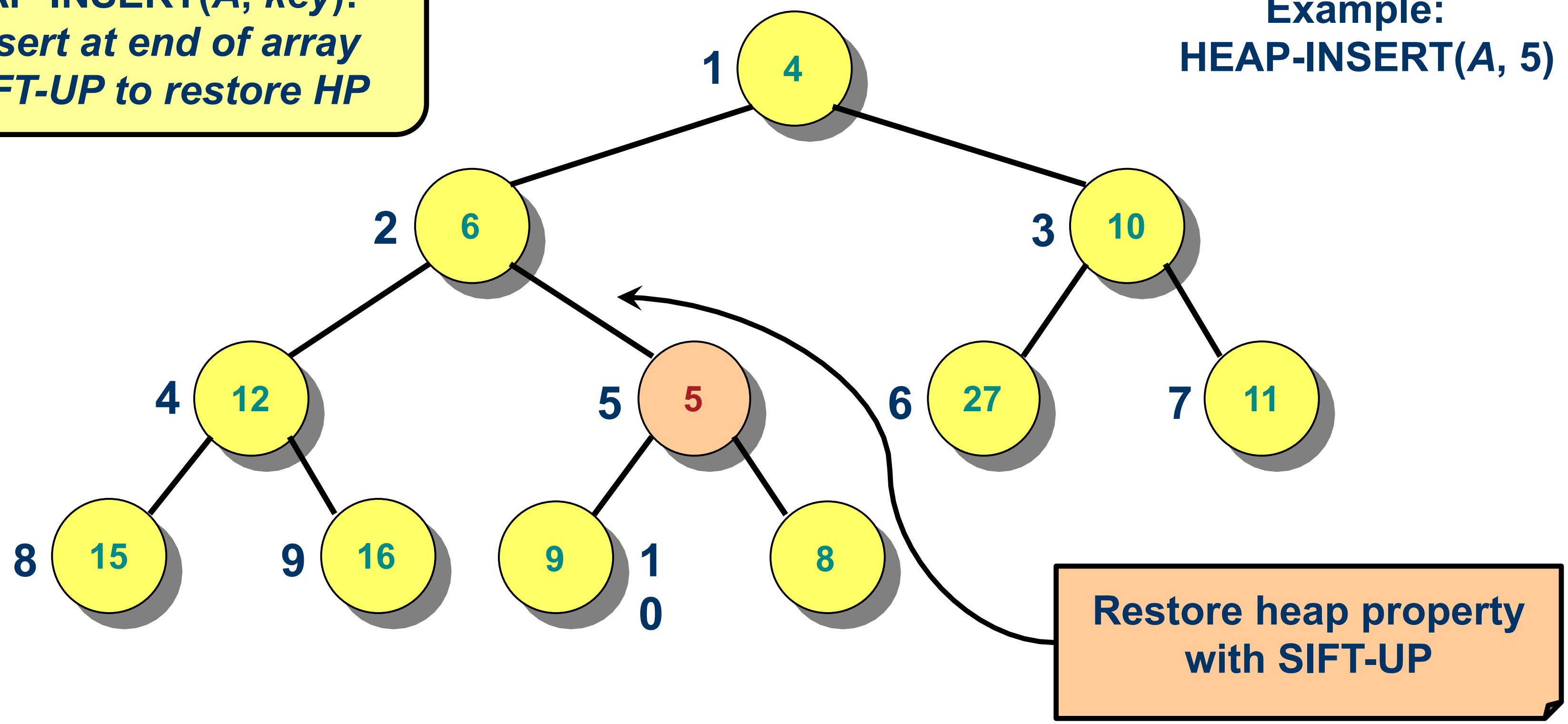
1	2	3	4	5	6	7	8	9	10	11
4	6	10	12	8	27	11	15	16	9	5

A[1..10]

MinHeap – Insert operation (2)

HEAP-INSERT(*A*, *key*):
insert at end of array
SIFT-UP to restore HP

Example:
HEAP-INSERT(*A*, 5)



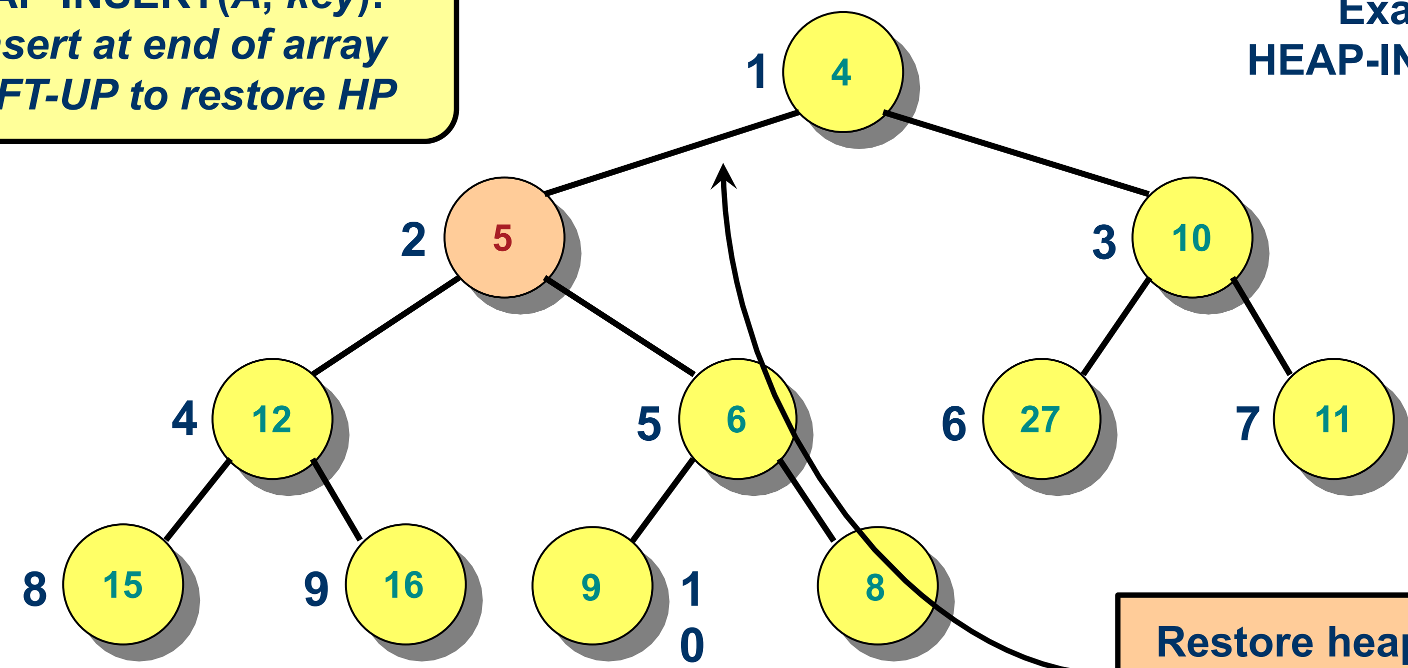
1	2	3	4	5	6	7	8	9	10	11
4	6	10	12	5	27	11	15	16	9	8

A[1..10]

MinHeap – Insert operation (3)

HEAP-INSERT(*A*, *key*):
insert at end of array
SIFT-UP to restore HP

Example:
HEAP-INSERT(*A*, 5)



Restore heap property with SIFT-UP

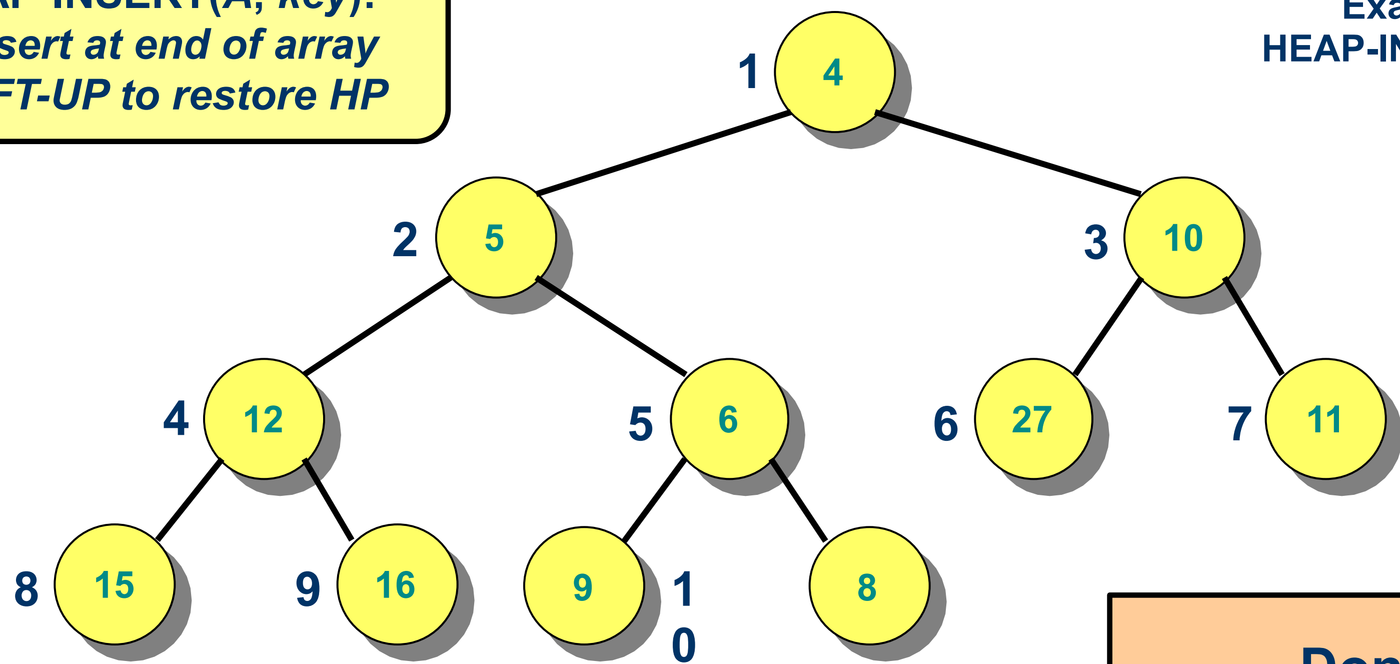
1	2	3	4	5	6	7	8	9	10	11
4	5	10	12	6	27	11	15	16	9	8

A[1..10]

MinHeap – Insert operation (4)

HEAP-INSERT(*A*, *key*):
insert at end of array
SIFT-UP to restore HP

Example:
HEAP-INSERT(*A*, 5)



Done!

1	2	3	4	5	6	7	8	9	10	11
4	5	10	12	6	27	11	15	16	9	8

A[1..10]

MinHeap – Insert operation

```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    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 ] );
}
```

Delete Minimum

- **deleteMin is handled in a similar manner as insertion:**
- **Remove the minimum; a hole is created at the root.**
- **The last element X must move somewhere in the heap.**
  - **If X can be placed in the hole then we are done.**
  - **Otherwise,**
    - **We slide the smaller of the hole's children into the hole, thus pushing the hole one level down.**
    - **We repeat this until X can be placed in the hole.**
- **deleteMin is logarithmic in both the worst and average cases.**

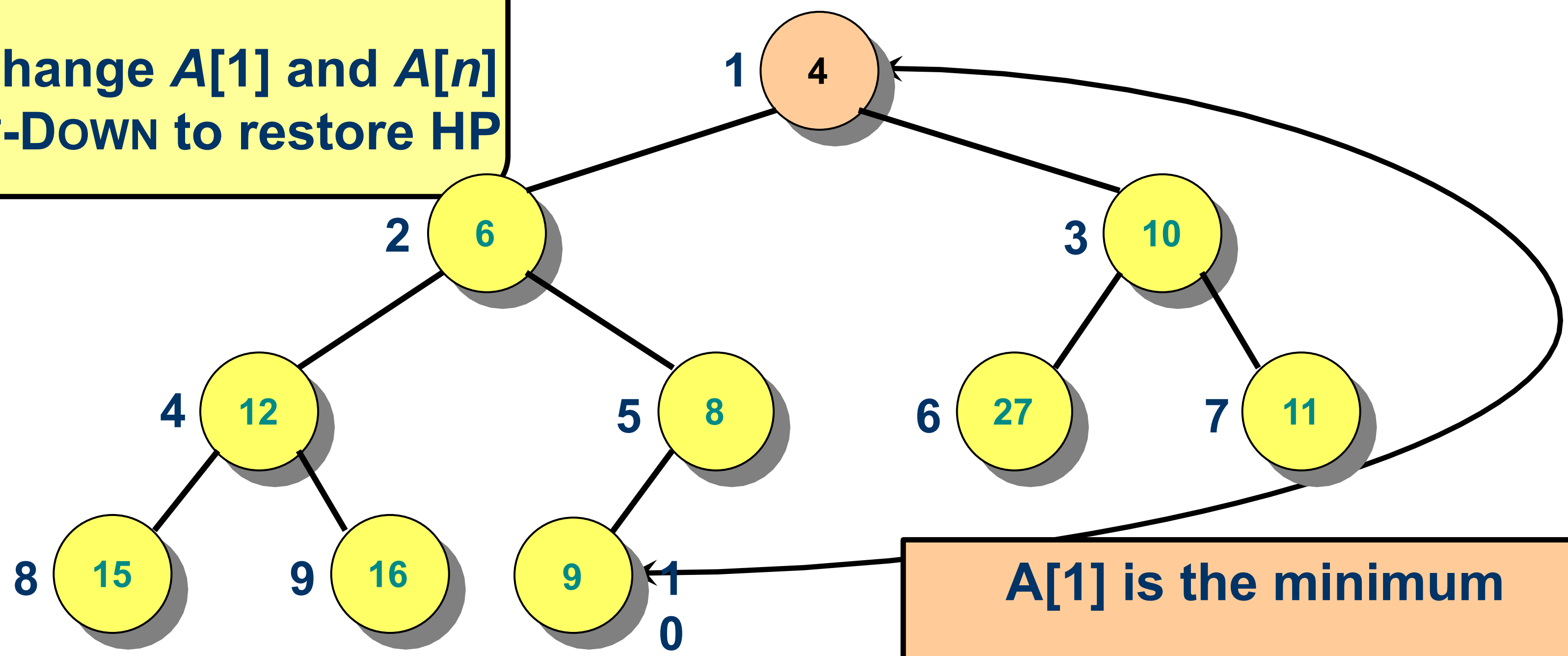
Heap – RemoveFirst

Basic Idea:

1. **Remove root (that is always the min!)**
2. **Put “last” leaf node at root**
3. **Find smallest child of node**
4. **Swap node with its smallest child if needed.**
5. **Repeat steps 3 & 4 until no swaps needed.**

MinHeap – RemoveFirst operation (1)

Exchange *A*[1] and *A*[*n*]
SIFT-DOWN to restore HP



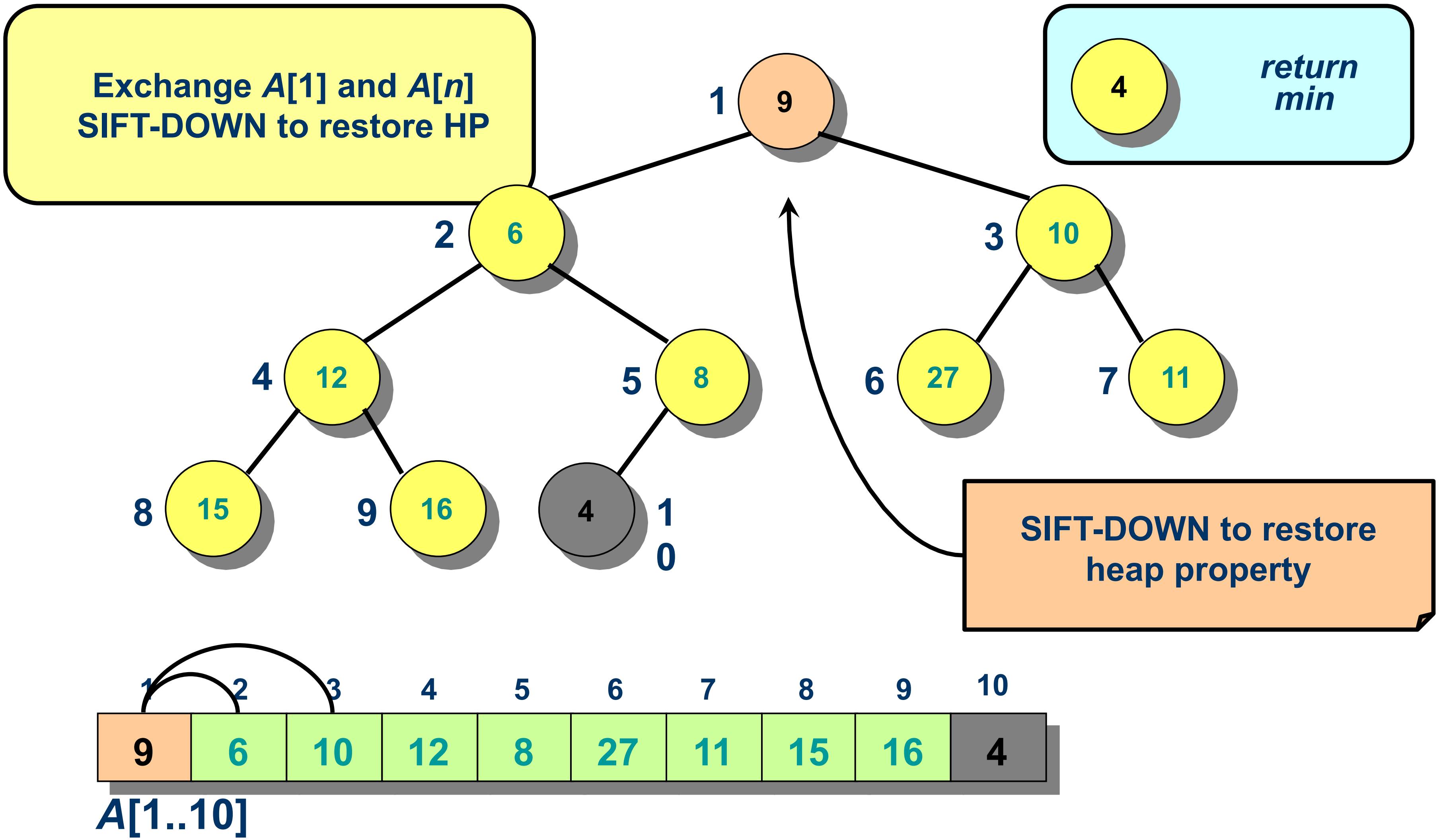
A[1] is the minimum

Exchange *A*[1] with *A*[*n*]

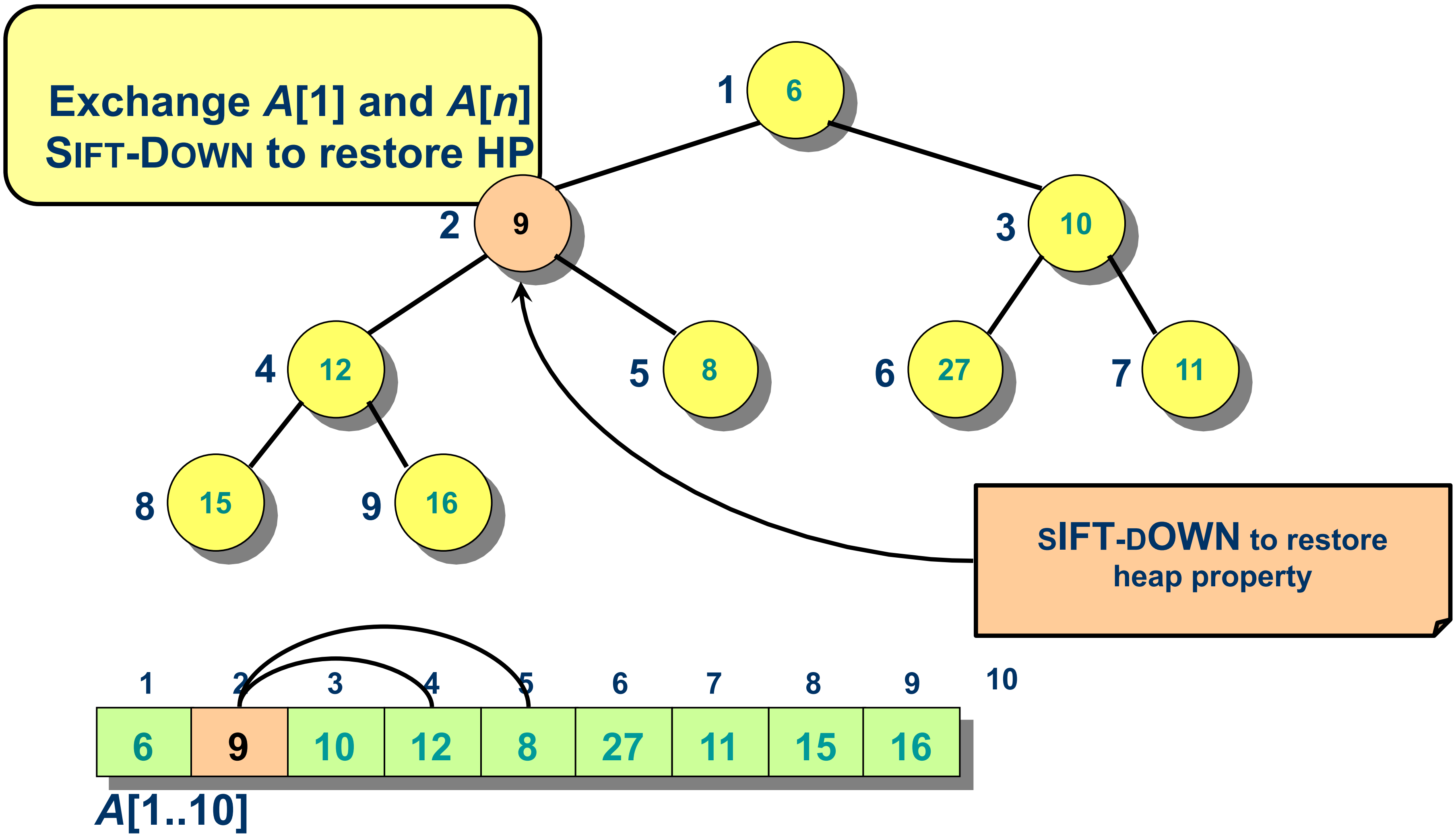
1	2	3	4	5	6	7	8	9	10
4	6	10	12	8	27	11	15	16	9

A[1..10]

MinHeap – RemoveFirst operation (2)

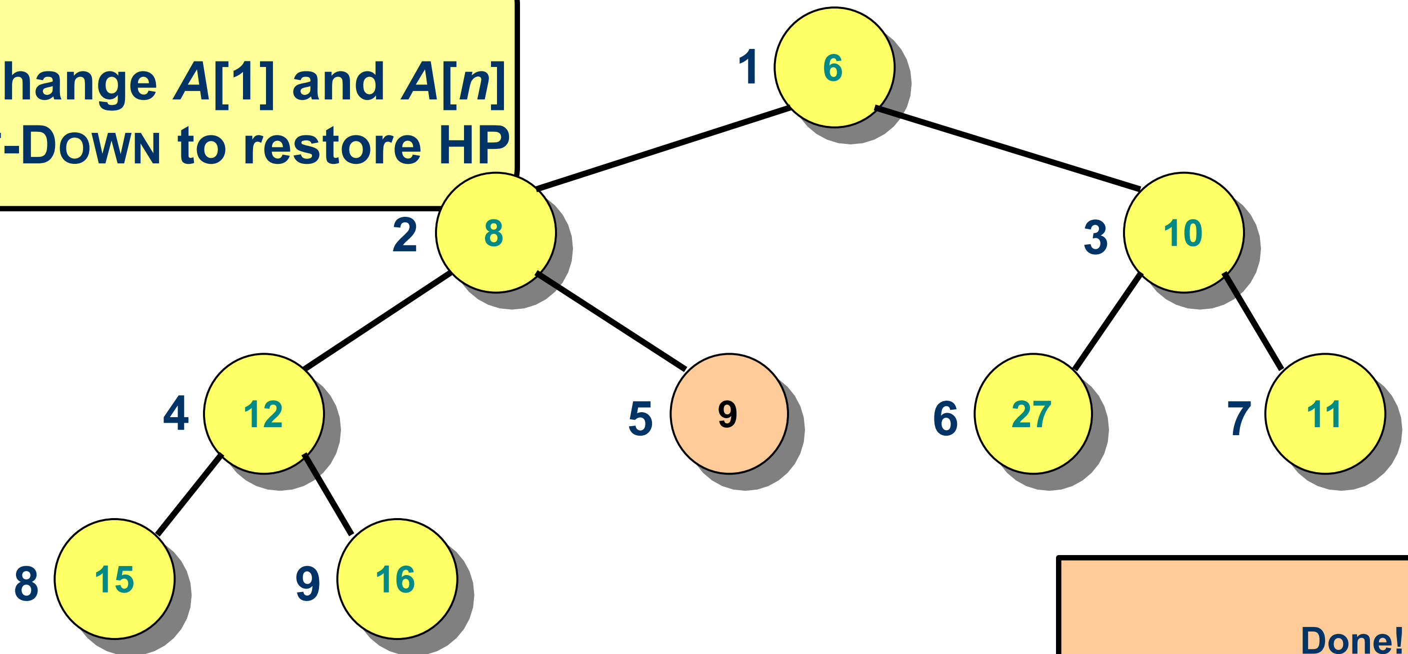


MinHeap – RemoveFirst operation (3)



MinHeap – RemoveFirst operation (4)

Exchange *A*[1] and *A*[*n*]
SIFT-DOWN to restore HP

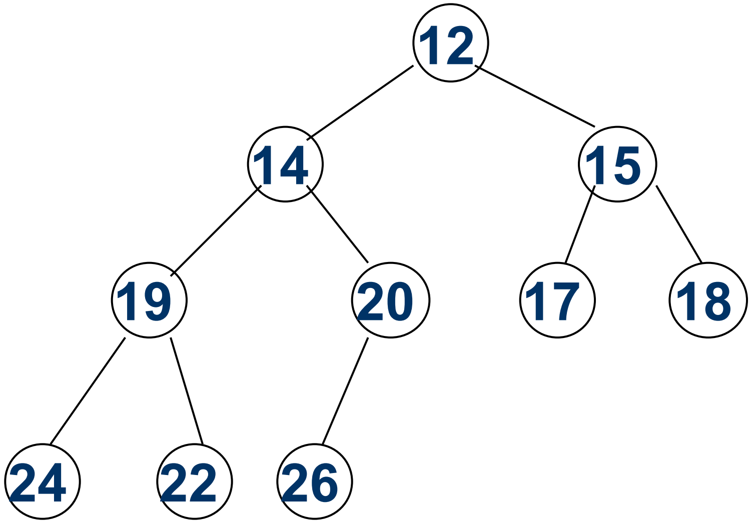


Done!

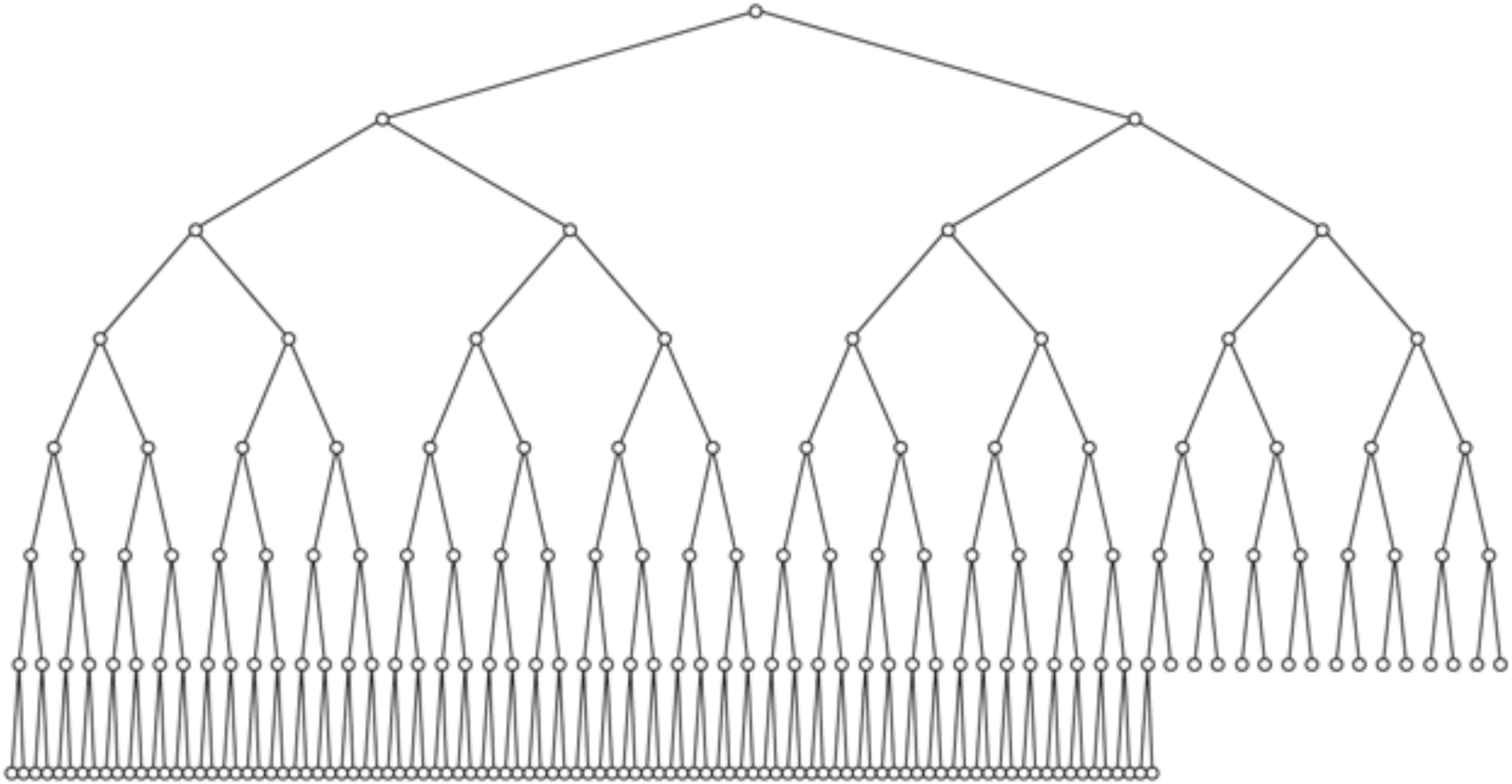
1	2	3	4	5	6	7	8	9	10
6	8	10	12	9	27	11	15	16	

A[1..10]

Heap – Remove



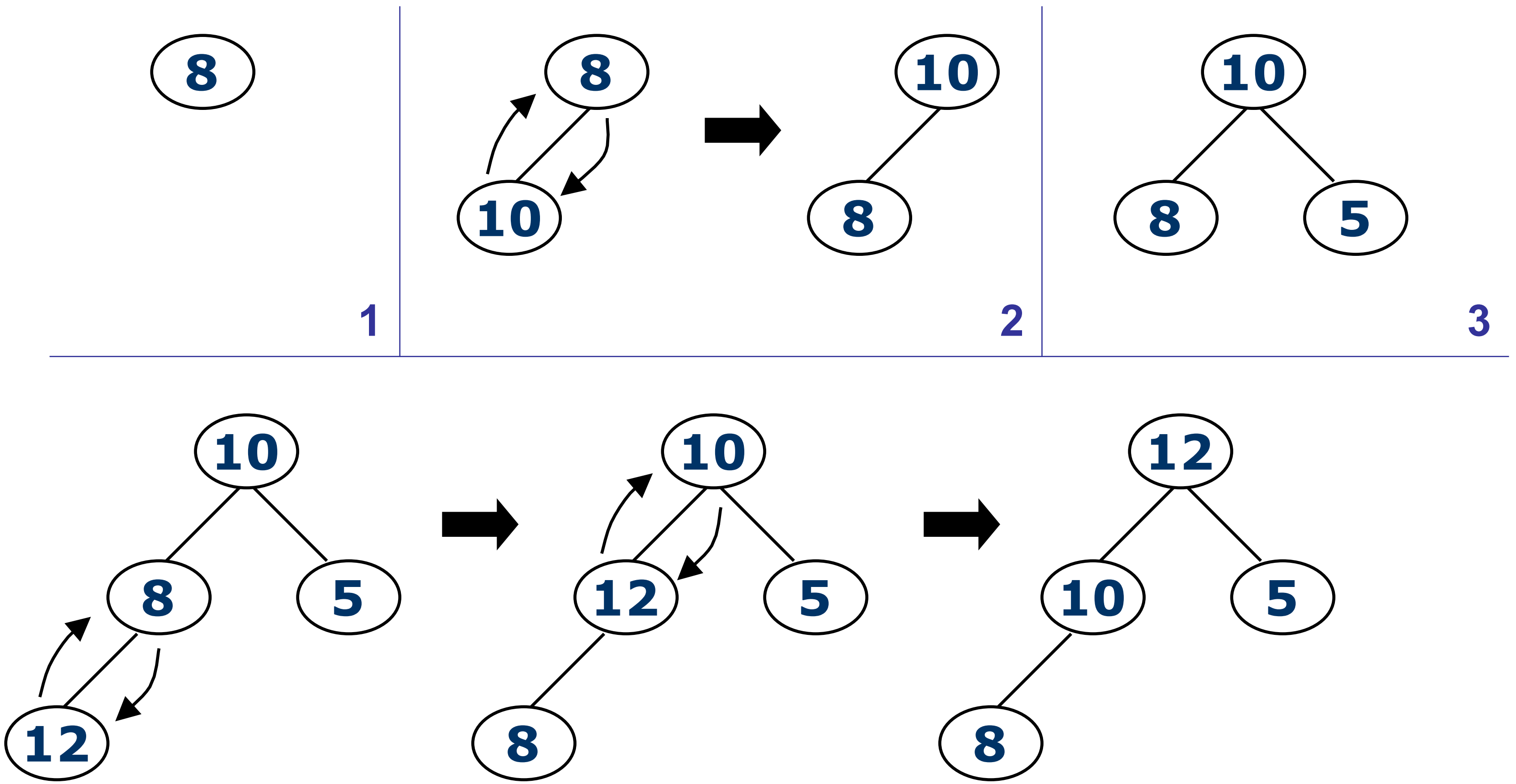
```
/**
 * Remove the minimum item.
 * Throws UnderflowException if empty.
 */
void deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException{ };

    array[ 1 ] = std::move( array[ currentSize-- ] );
    percolateDown( 1 );
}

/**
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */
void deleteMin( Comparable & minItem )
{
    if( isEmpty( ) )
        throw UnderflowException{ };

    minItem = std::move( array[ 1 ] );
    array[ 1 ] = std::move( array[ currentSize-- ] );
    percolateDown( 1 );
}

/**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */
void percolateDown( int hole )
{
    int child;
    Comparable tmp = std::move( array[ hole ] );

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize && array[ child + 1 ] < array[ child ] )
            ++child;
        if( array[ child ] < tmp )
            array[ hole ] = std::move( array[ child ] );
        else
            break;
    }
    array[ hole ] = std::move( tmp );
}
```



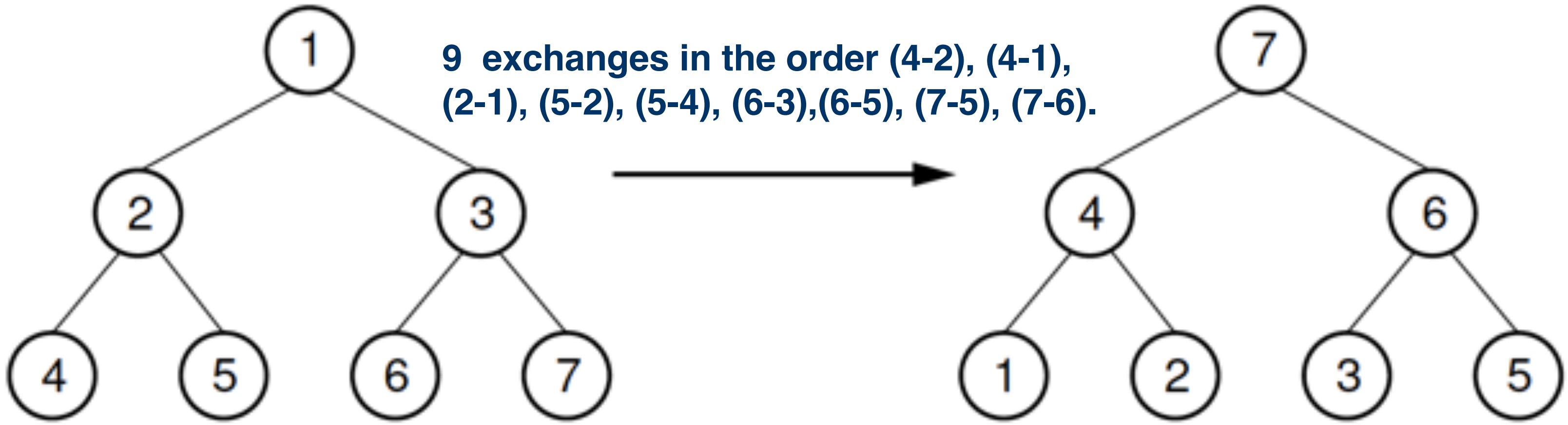
Building a Heap

- Construct heap from initial set of N items
- Solution 1
  - Perform N inserts
  - $O(N \log_2 N)$ worst-case
- Solution 2 (use *buildHeap()*)
  - Randomly populate initial heap with structure property
  - Perform a sift-down from each internal node (H[size/2] to H[1])
  - O(N) worst-case
    - To take care of heap order property

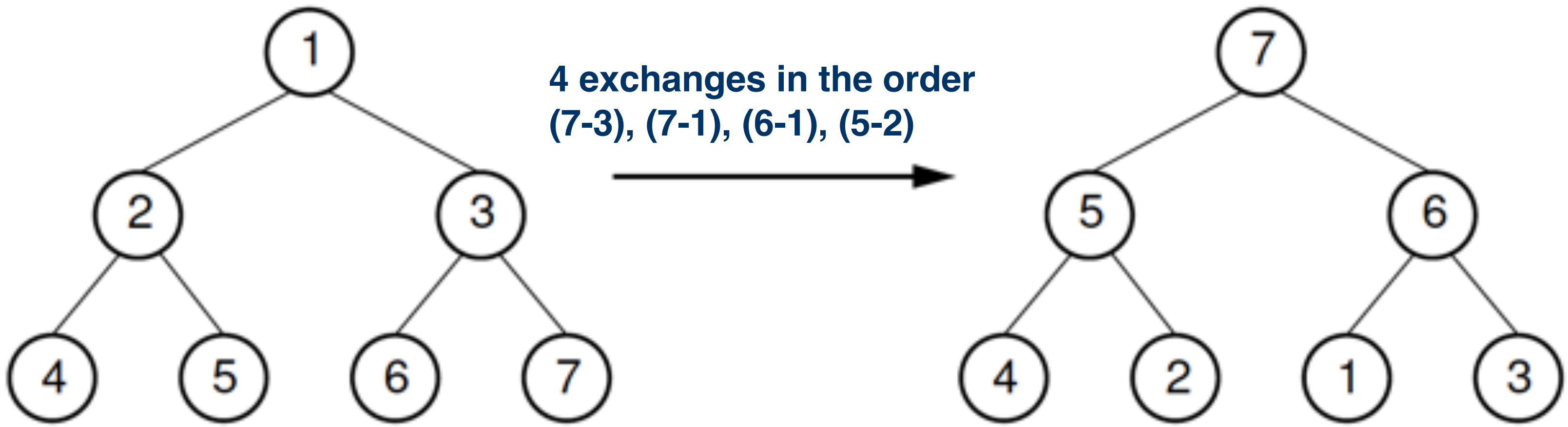
Build a max-heap by insertion (1)



Two series of exchanges to build a max-heap

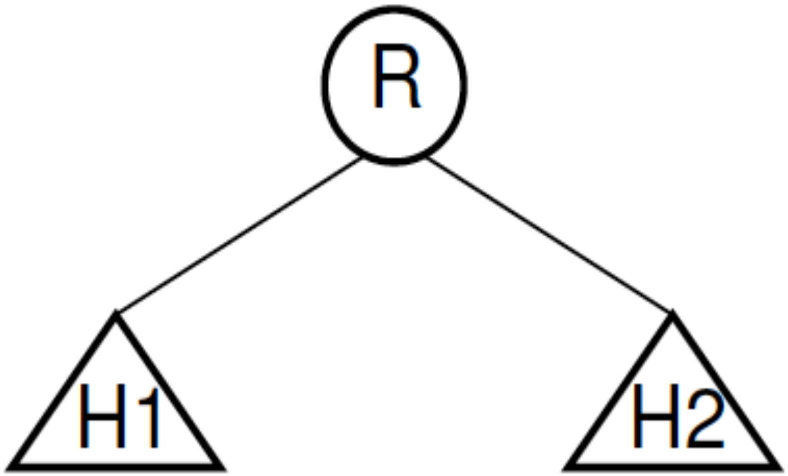


(a)



(b)

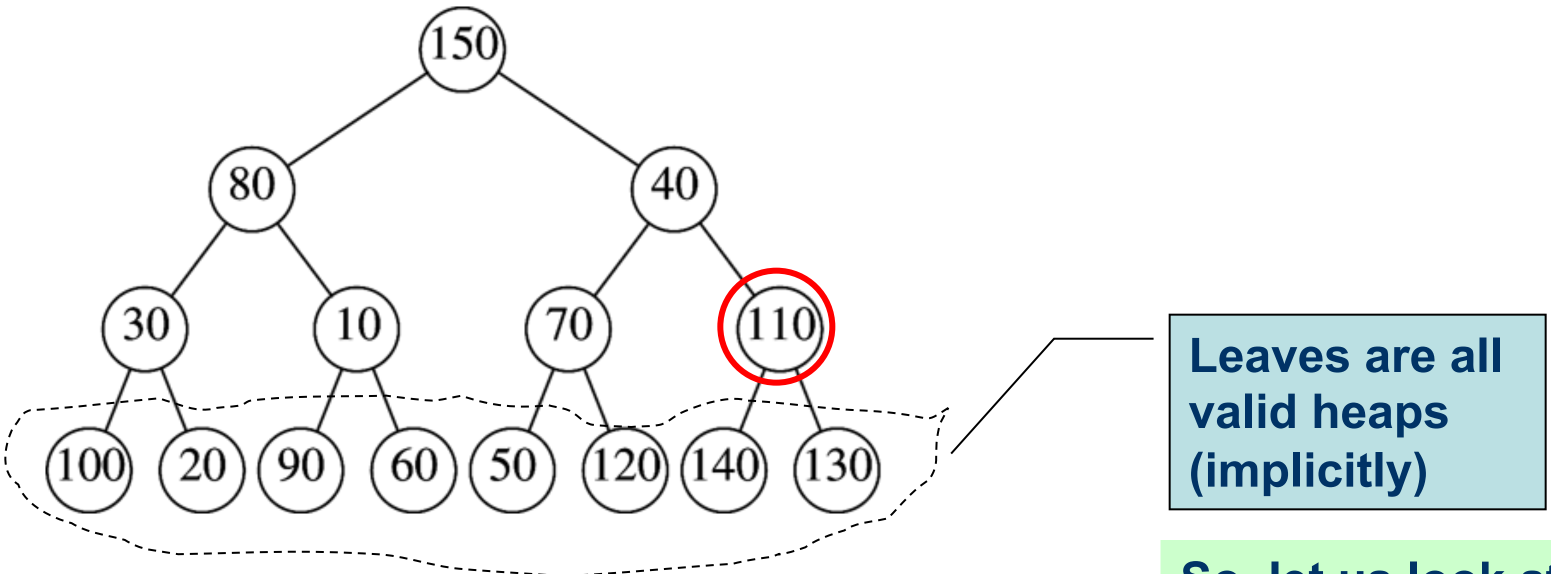
Pick a good rearrangement



Final stage in the heap-building algorithm. Both subtrees of node R are heaps. All that remains is to push R down to its proper level in the heap.

Build Min-Heap Example (2)

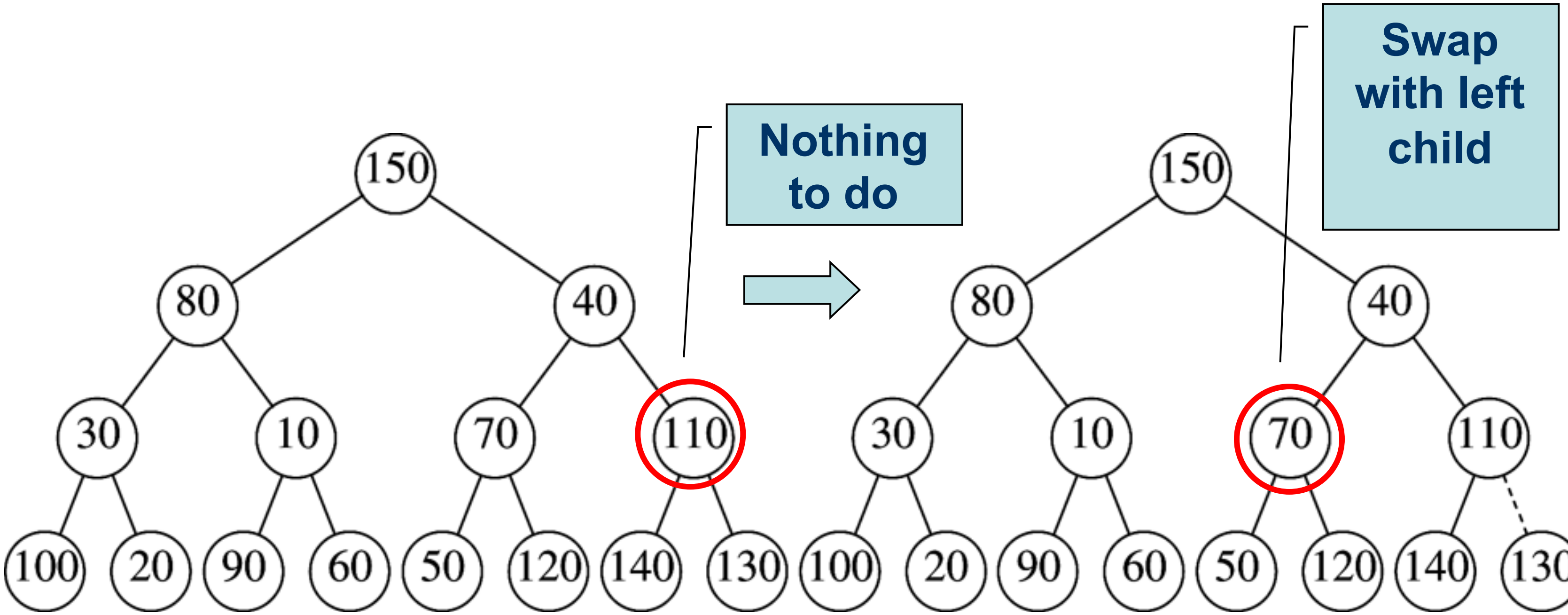
Input: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }



- Arbitrarily assign elements to heap nodes
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

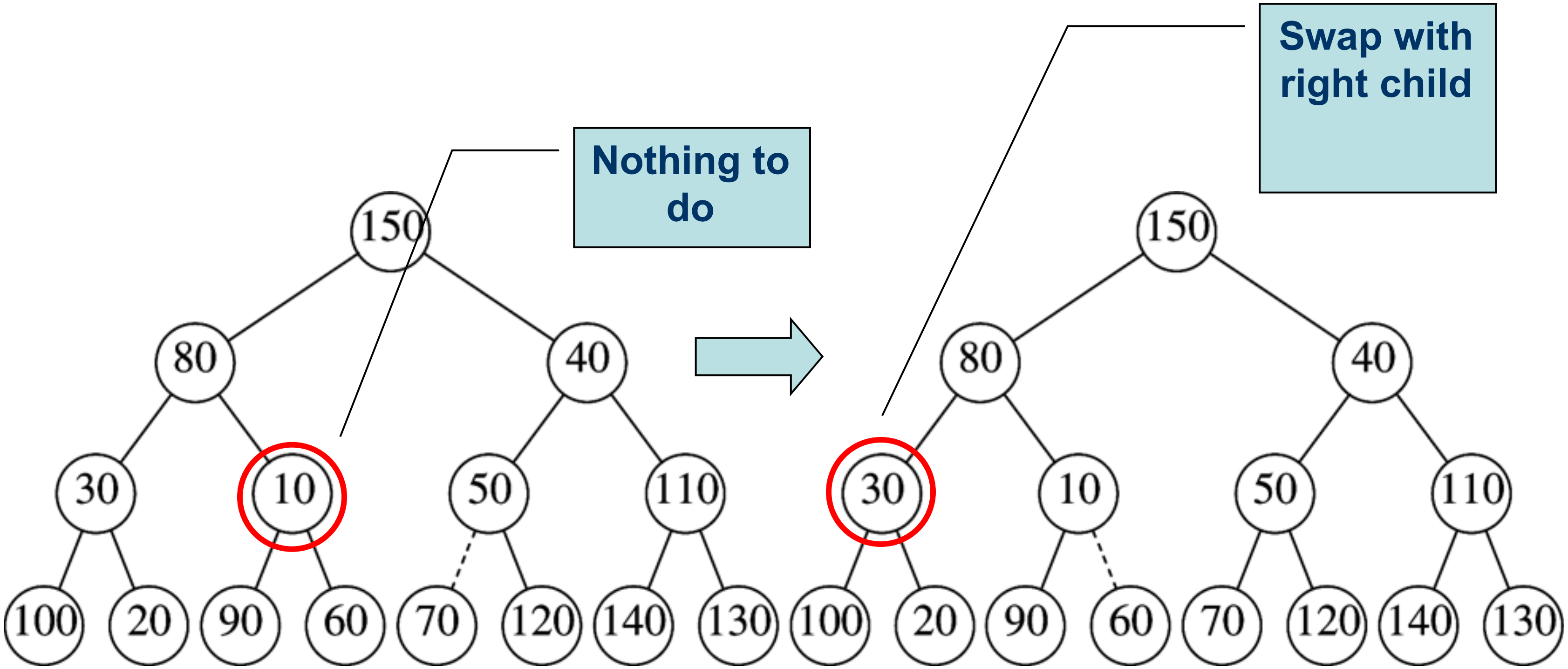
So, let us look at each internal node, from bottom to top, and fix if necessary

Build Min-Heap Example (2)



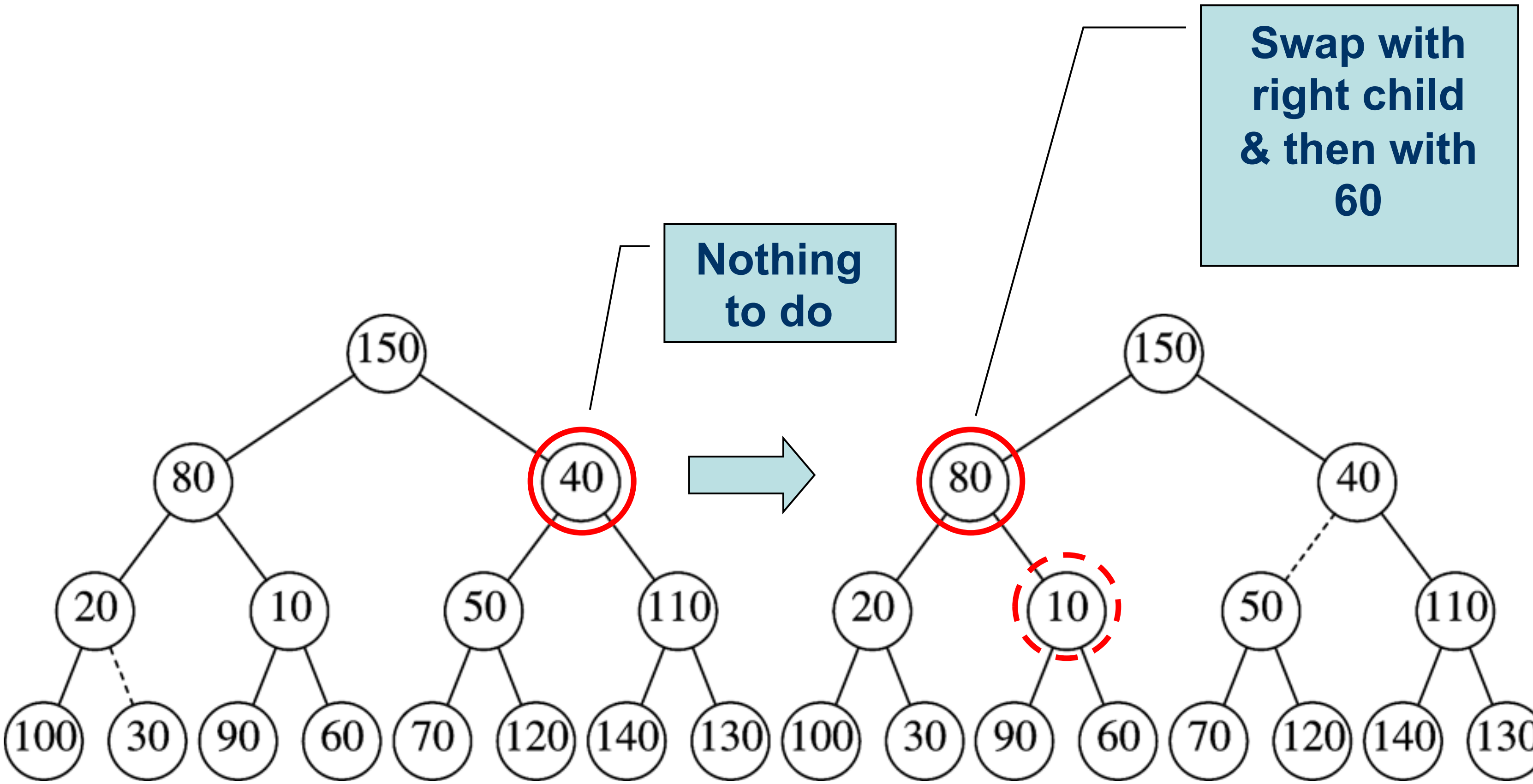
- Randomly initialized heap
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

Build Min-Heap Example (2)



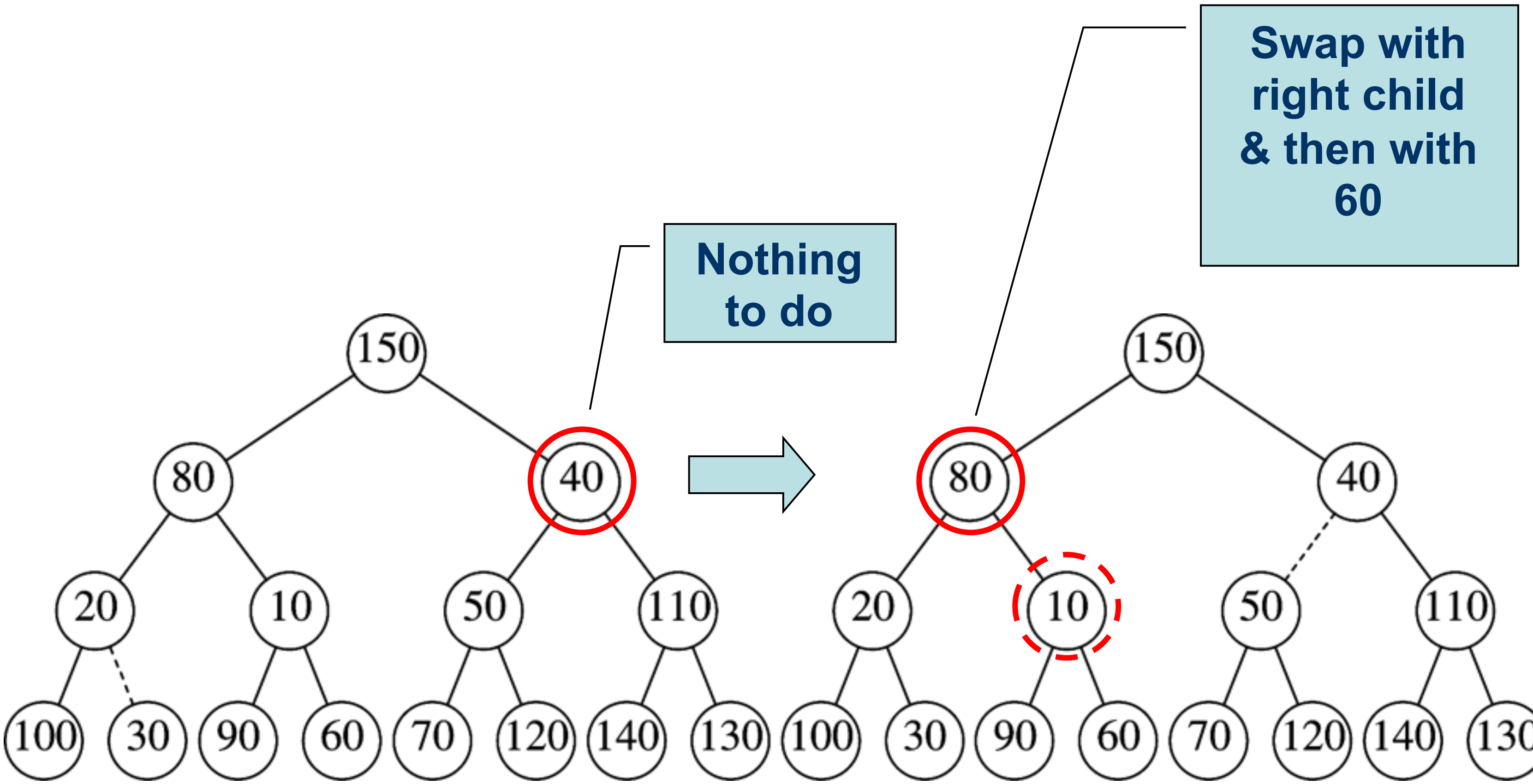
Dotted lines show path of percolating down

Build Min-Heap Example (2)



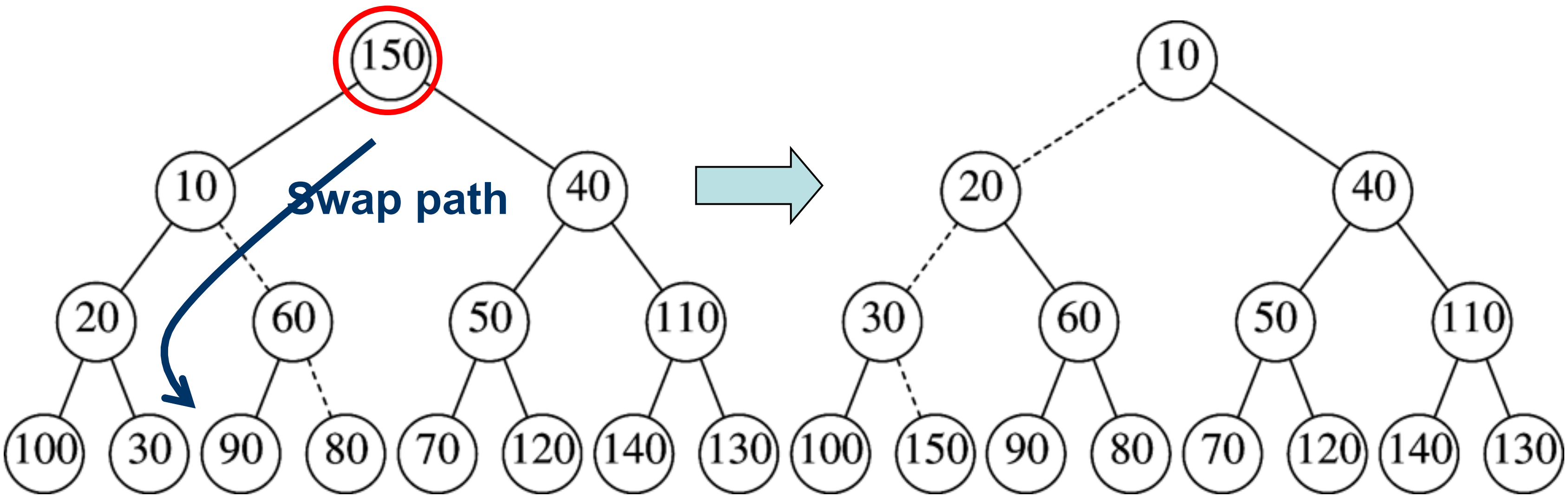
Dotted lines show path of percolating down

Build Min-Heap Example (2)



Dotted lines show path of percolating down

Build Min-Heap Example (2)

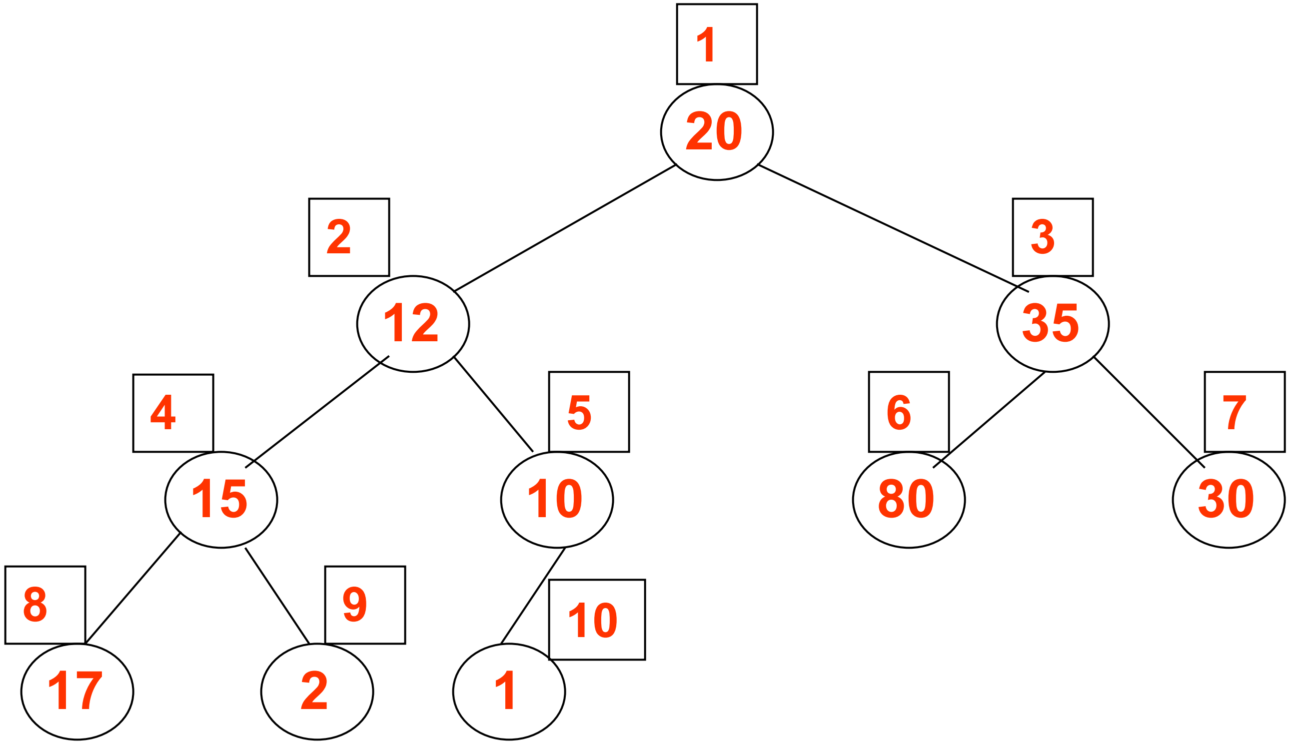


Final Heap

Dotted lines show path of percolating down

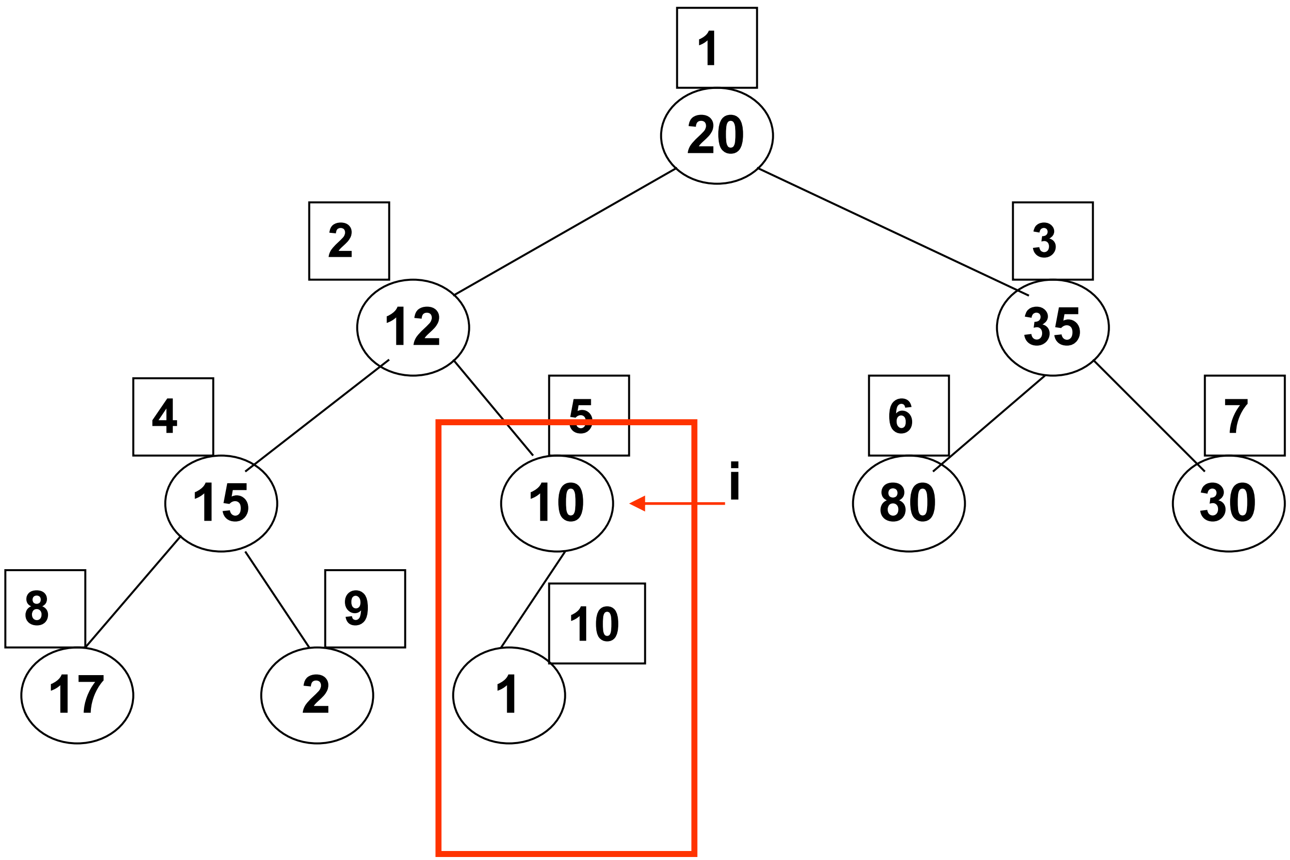
Max-Heap Example

a[10]={20,12,35,15,10,80,30,17,2,1}



Max-Heap - step1

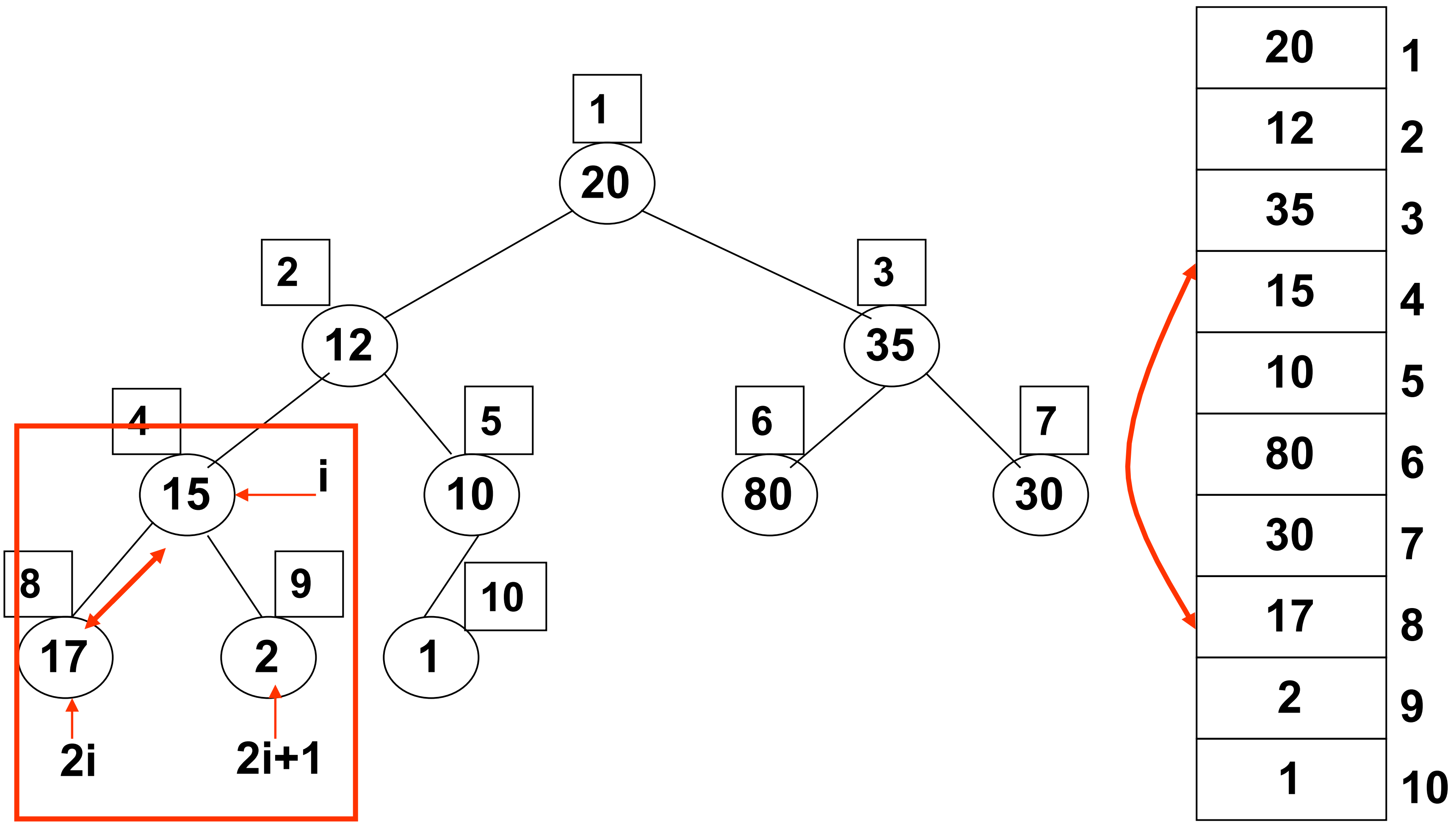
n=10; i=n/2=5;



20	1
12	2
35	3
15	4
10	5
80	6
30	7
17	8
2	9
1	10

Step2

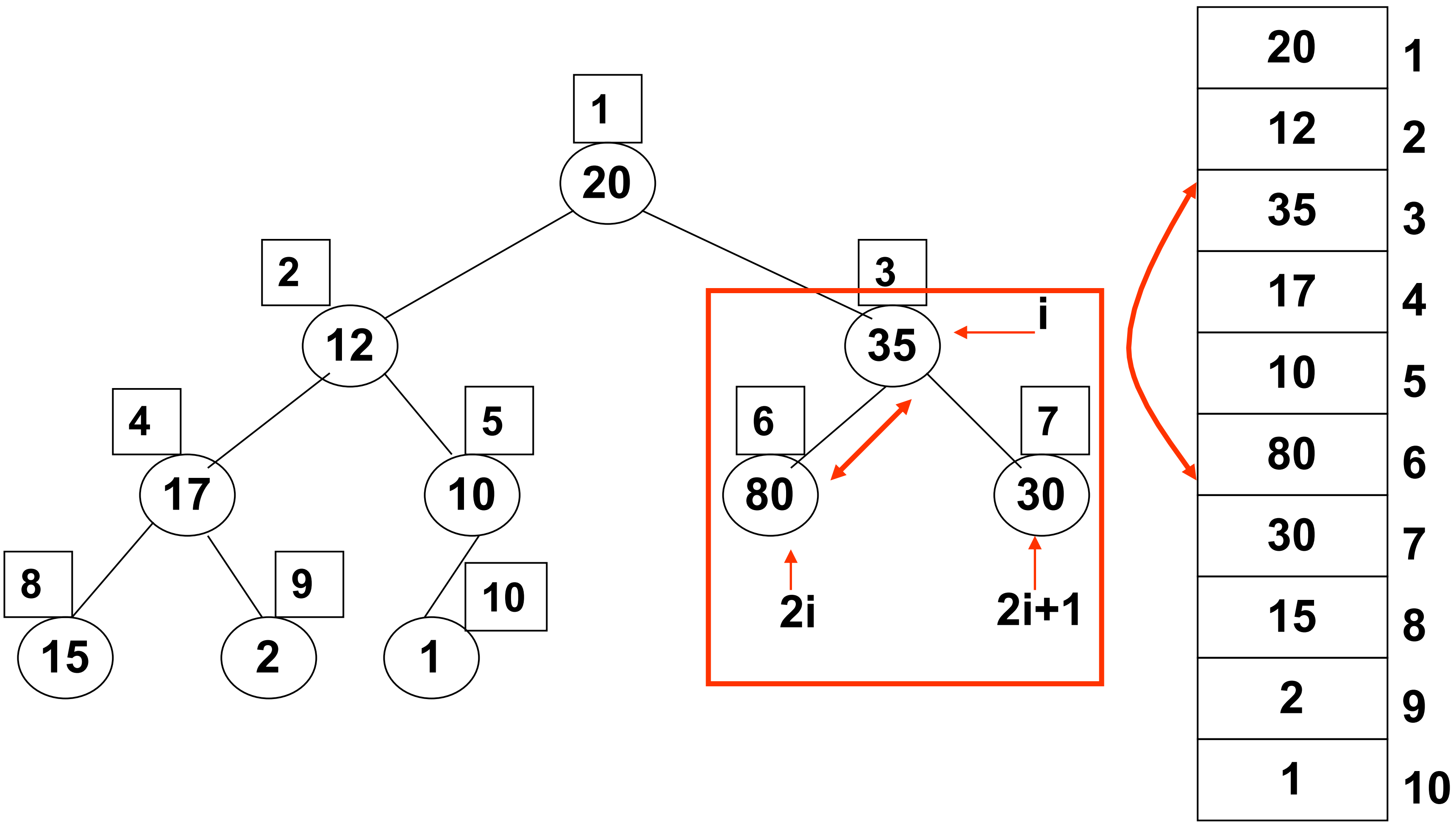
i=4



1
20
2
12
3
35
4
15
i
5
10
6
80
7
30
8
17
9
2
10
1
2i
2i+1
20 1
12 2
35 3
15 4
10 5
80 6
30 7
17 8
2 9
1 10

Step3

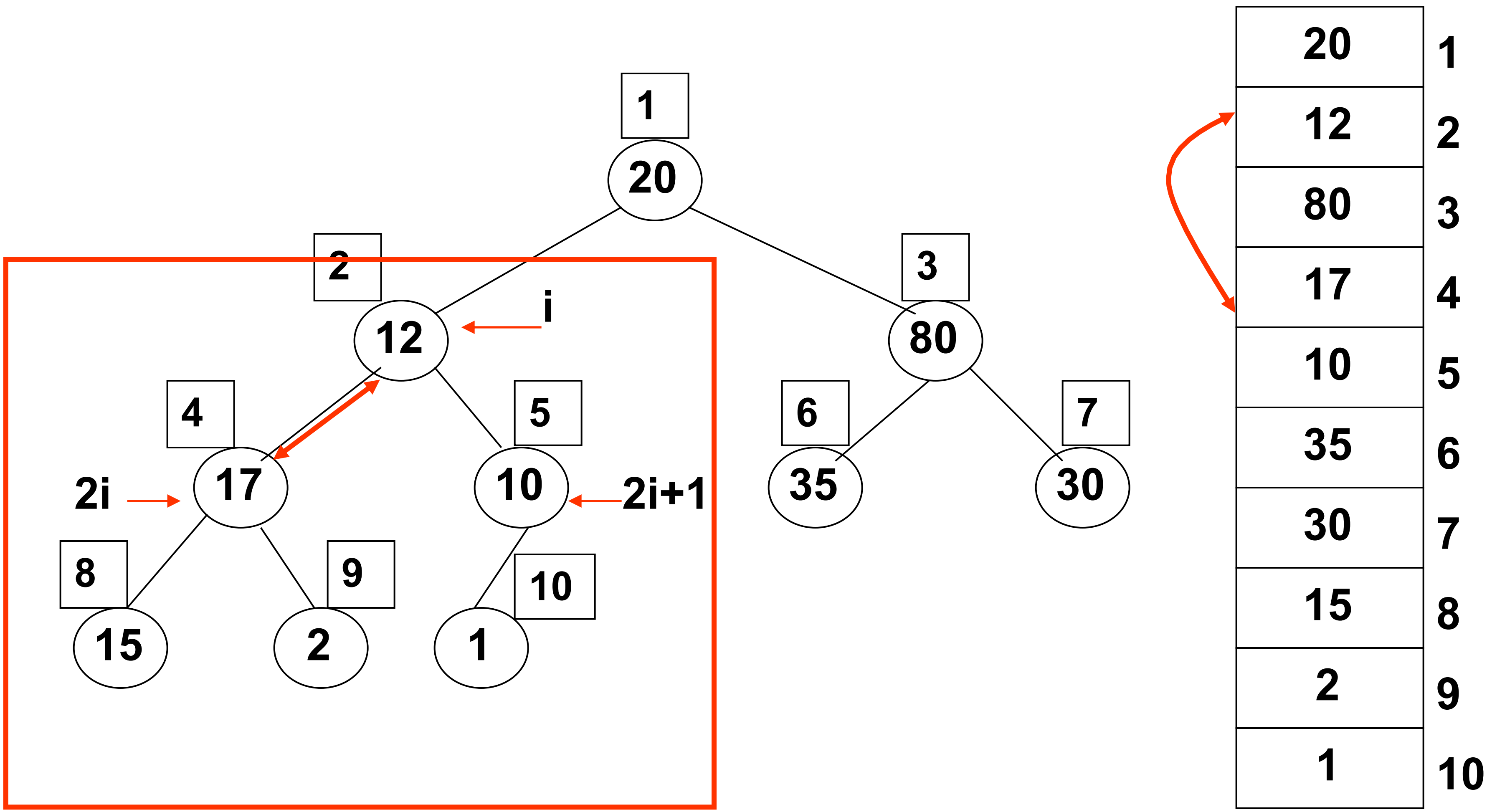
i=3



1
20
2
12
3
35
i
4
17
5
10
6
80
7
30
2i
2i+1
8
15
9
2
10
1
20 1
12 2
35 3
17 4
10 5
80 6
30 7
15 8
2 9
1 10

Step4_0

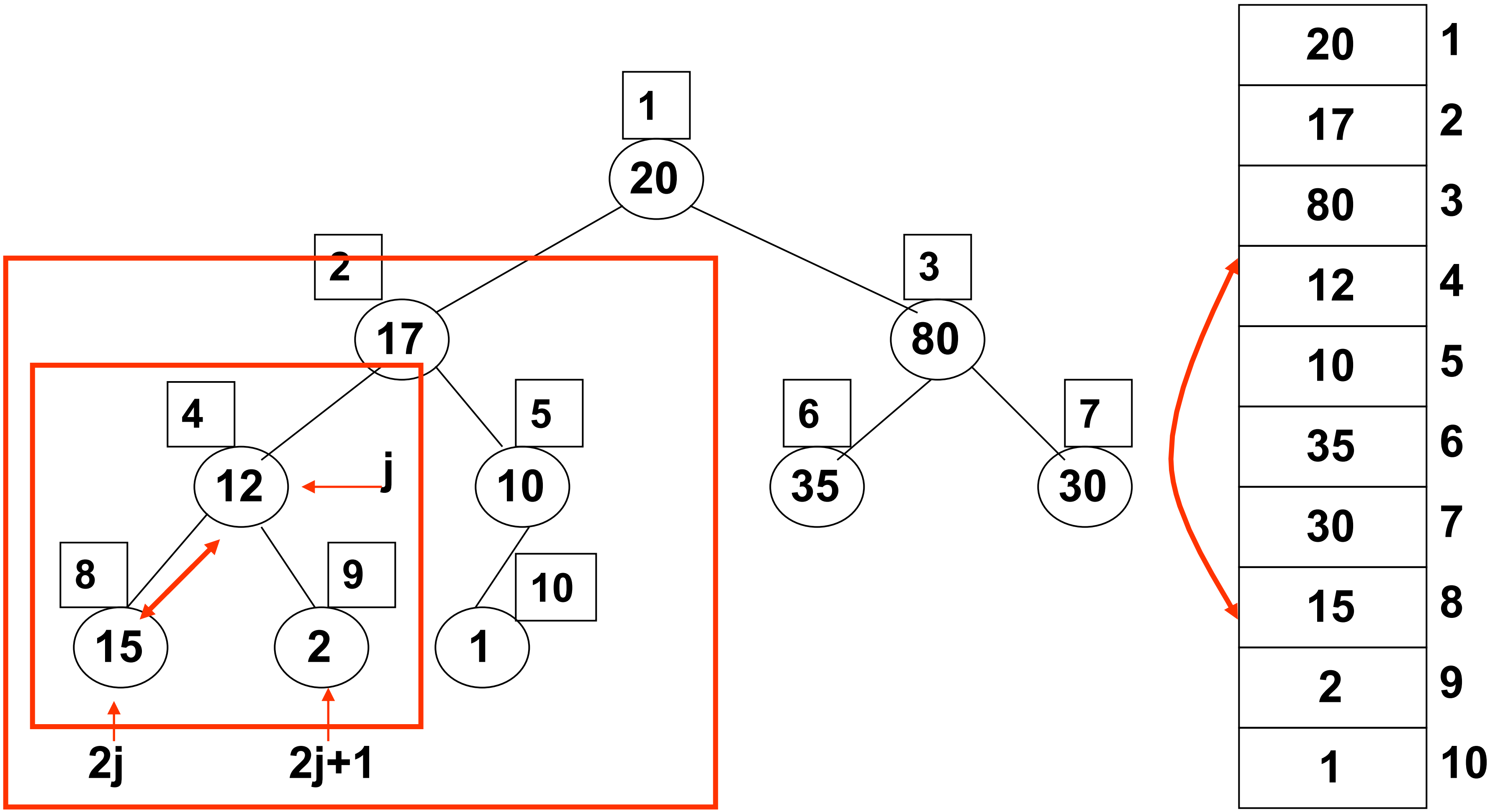
i=2



1
20
2
12
i
3
80
4
17
2i
5
10
2i+1
6
35
7
30
8
15
9
2
10
1
20 1
12 2
80 3
17 4
10 5
35 6
30 7
15 8
2 9
1 10

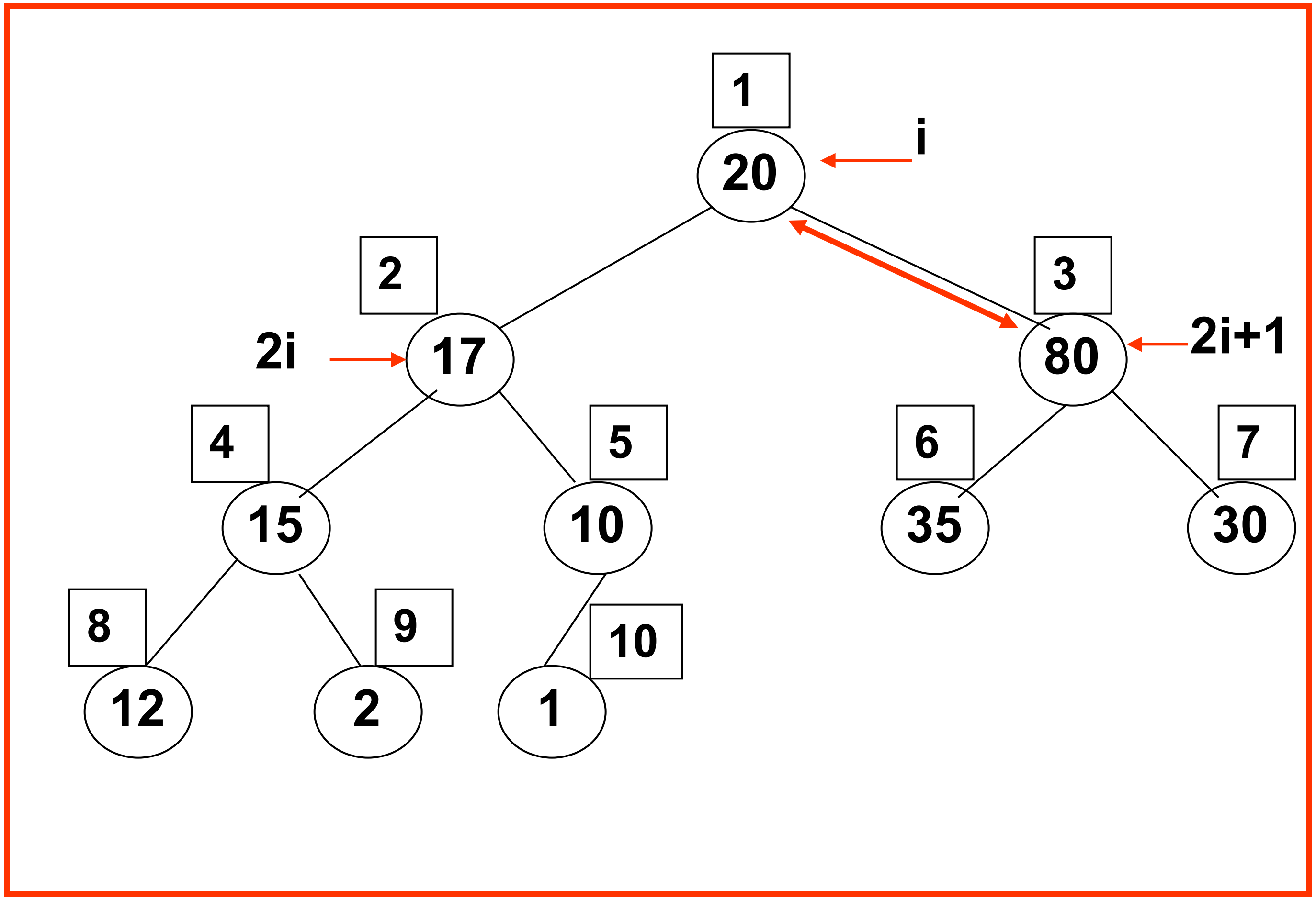
Step 4_1

i=2,j=2i=4

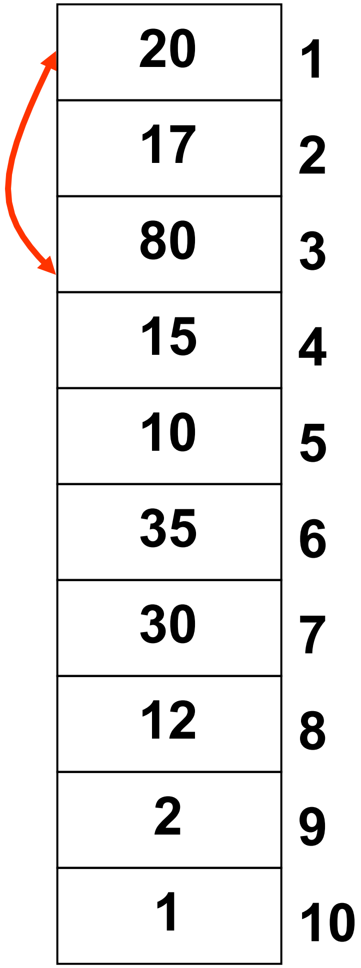


Step 5_0

i=1



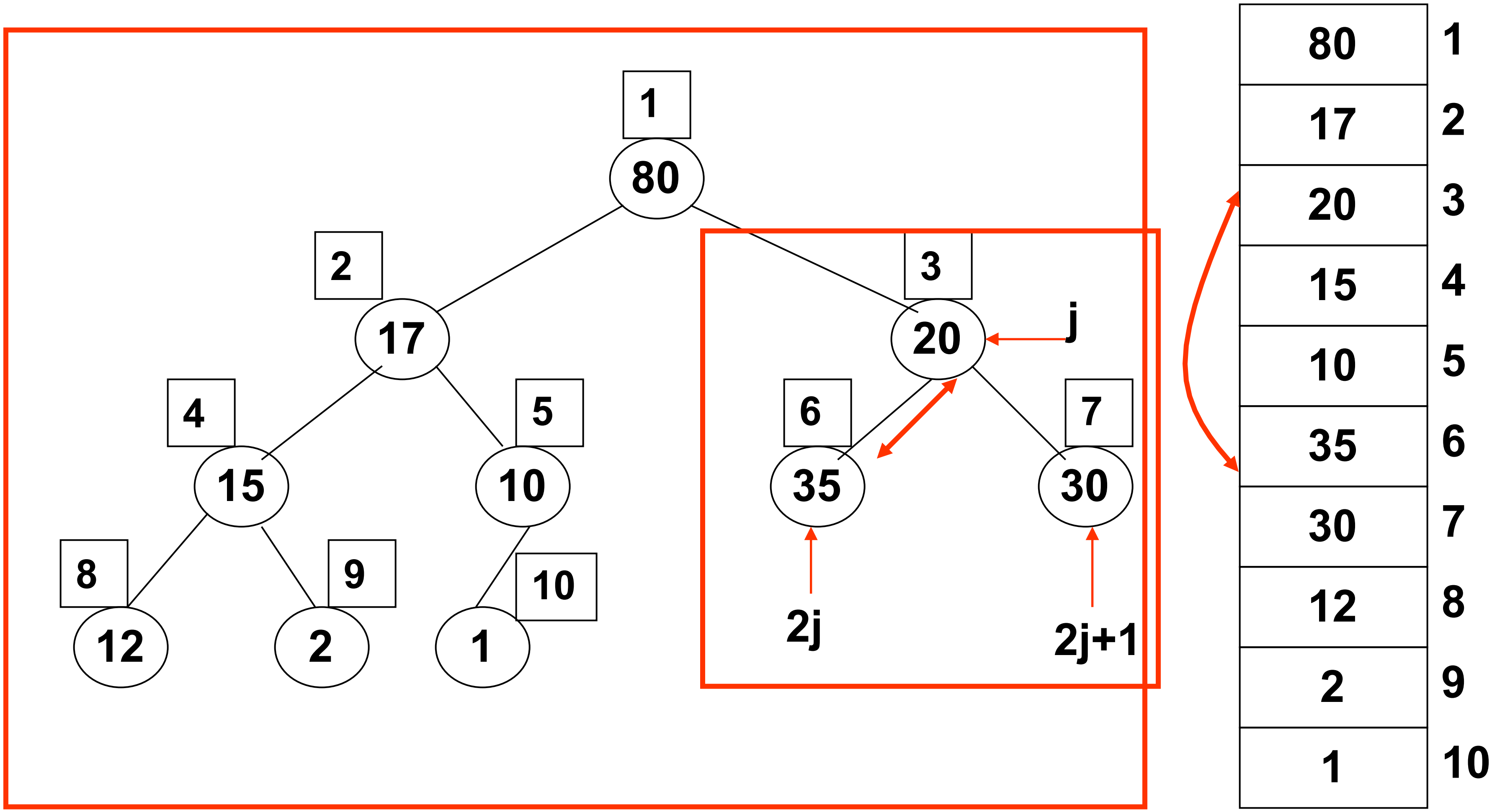
1
20
i
2
17
2i
3
80
2i+1
4
15
5
10
6
35
7
30
8
12
9
2
10
1



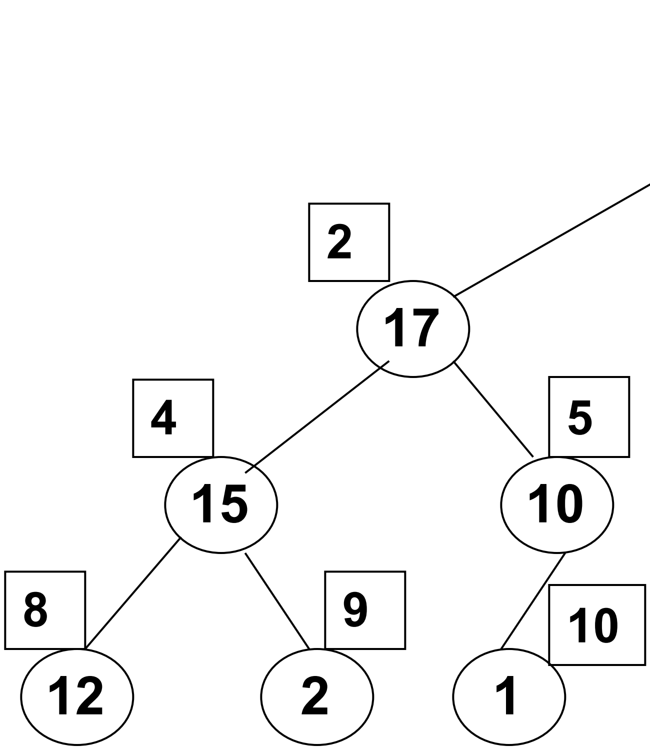
20 1
17 2
80 3
15 4
10 5
35 6
30 7
12 8
2 9
1 10

Step 5_1

i=1,j=2i+1=3



Step 5_2



80	1
17	2
35	3
15	4
10	5
20	6
30	7
12	8
2	9
1	10

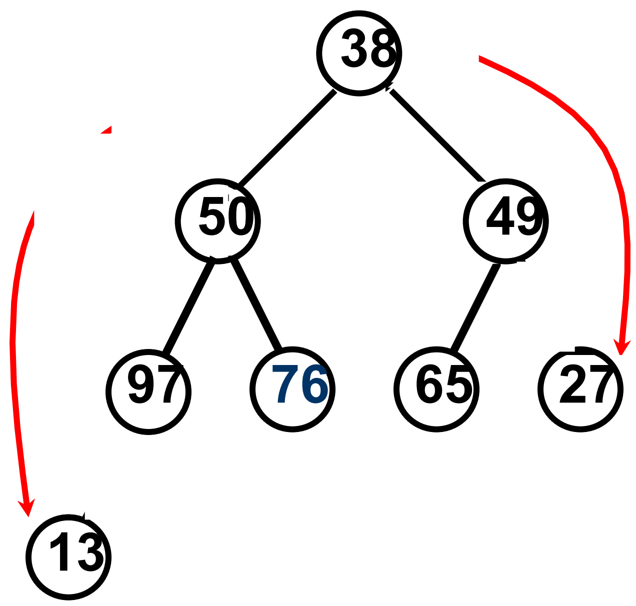
Exercises

Key : K = {12， 14， 15， 19， 20， 17， 18， 24， 22， 26} , build a max-heap with the time complexity O (n)

Heap sort (2)

Ex, 49 38 65 97 76 13 27 50

1. Build a complete binary;



2. Adjustment to a minheap;

3. Get minimum 13

4. Delete 13 , a new heap;

5. Get 27;

6. Delete 27 , a new heap;

7. Get 38;

Build Heap (2)

- Time complexity O(n)
- Insert delete O(log n)
- Search minimum/maxium O(1)

Homework

- **Please refer to Icourse，Huawei Cloud.**

- **Due date for quiz: 23:30 2022/5/10**
- **Due date for homework: 23:30 2022/5/15**
- **Due data for online lab assignment: 2022/5/15 23：30**