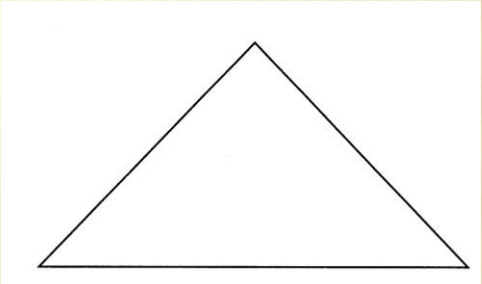
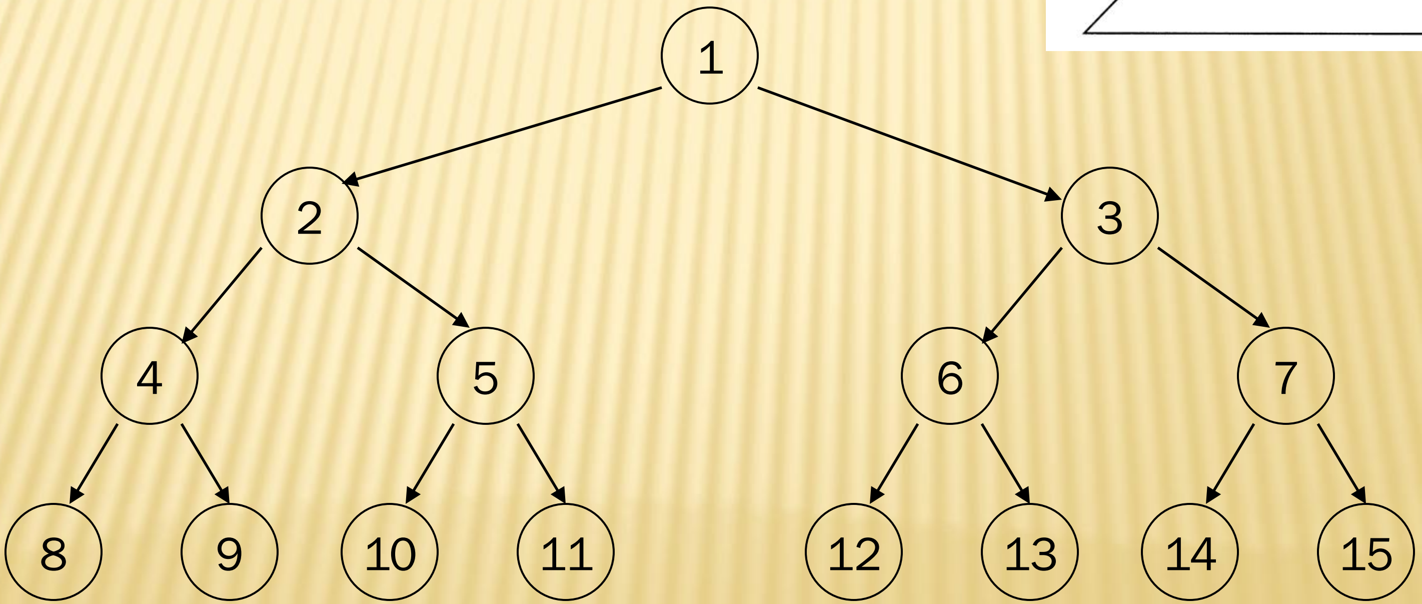
HEAP

FULL BINARY TREE

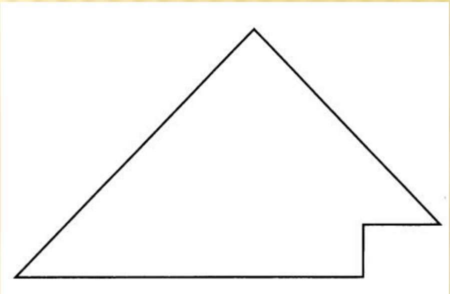
- A binary tree of height *k* having $2^k - 1$ nodes is called a *full* binary tree
- Every non-leaf node has two children
- All the leaves are on the same level

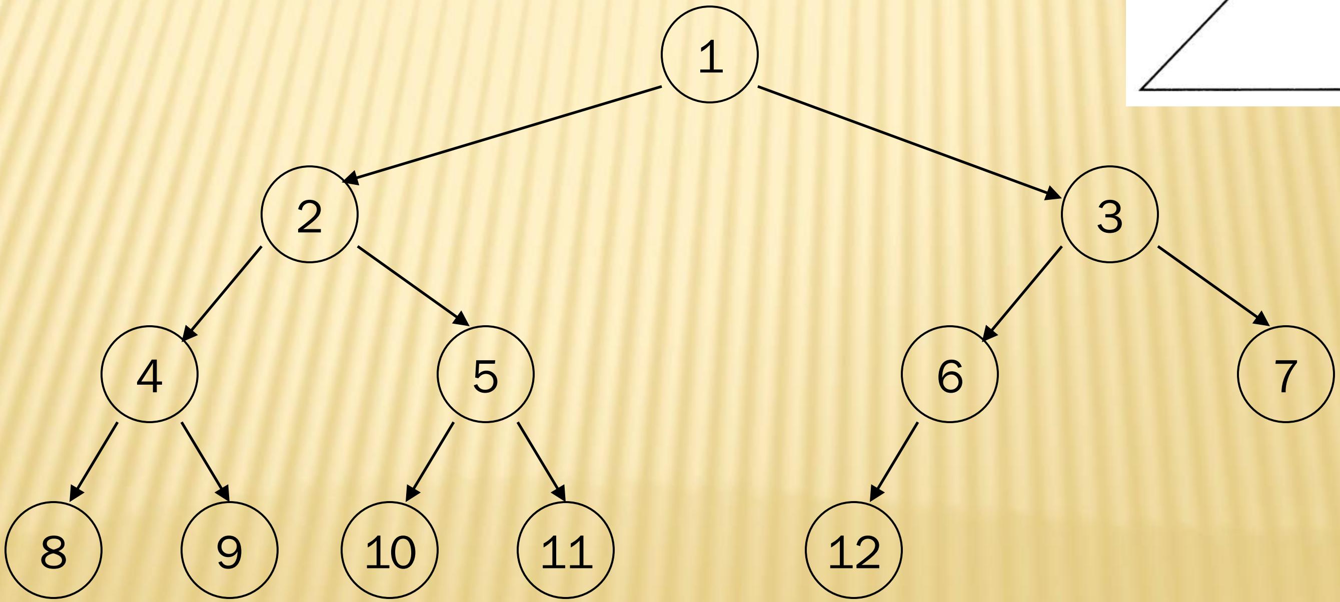




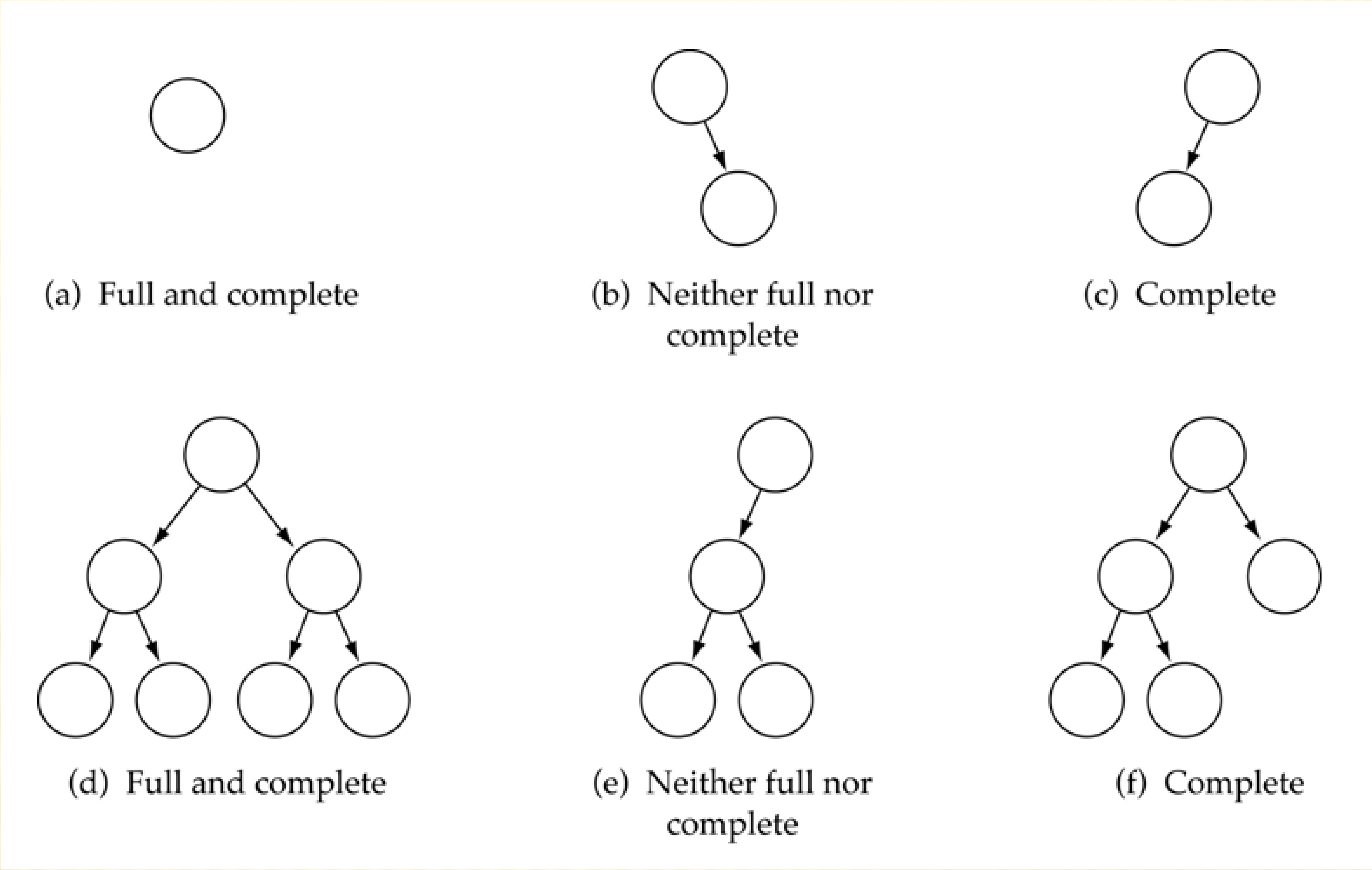
COMPLETE BINARY TREE

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





COMPLETE BINARY TREE



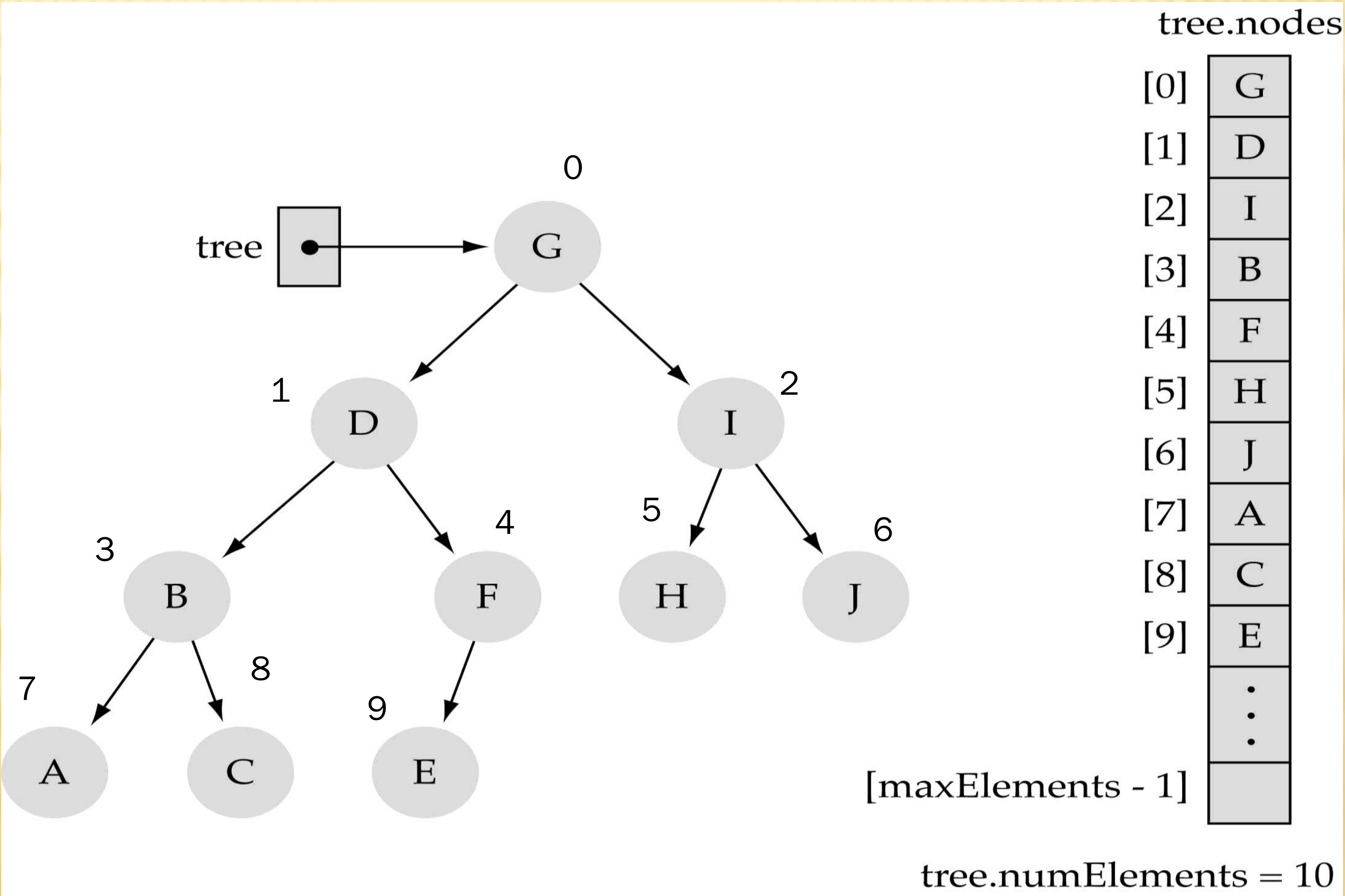
ARRAY-BASED REPRESENTATION OF BINARY TREES

- Memory space can be saved (no pointers are required)
- Preserve parent-child relationships by storing the tree elements in the array

    (i) level by level,

    (ii) left to right

ARRAY-BASED REPRESENTATION OF BINARY TREES



ARRAY-BASED REPRESENTATION OF BINARY TREES (CONT.)

- Parent-child relationships:
  - left child of *tree.nodes[index]*

    = *tree.nodes[2*index+1]*
  - right child of *tree.nodes[index]*

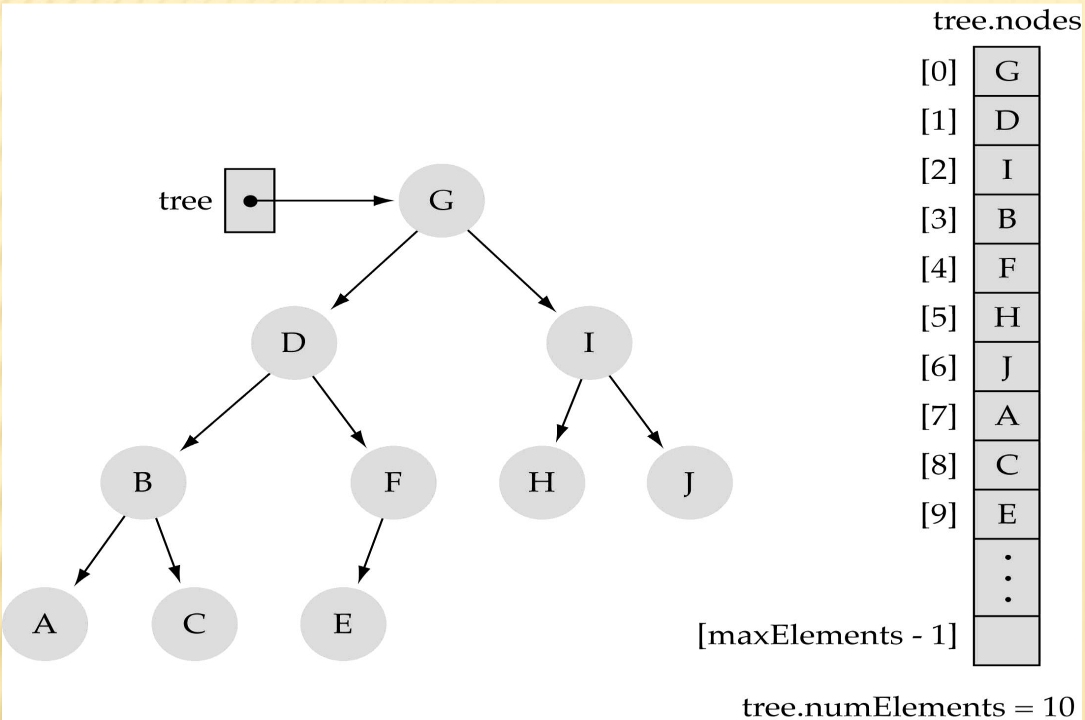
    = *tree.nodes[2*index+2]*
  - parent node of *tree.nodes[index]*

    = *tree.nodes[(index-1)/2]*

    (integer division-truncate)
- Leaf nodes:
  - *Exist between*

    tree.nodes[numElements/2] to *tree.nodes[numElements - 1]*

ARRAY-BASED REPRESENTATION OF BINARY TREES (CONT.)



B is at index 3

Left child:

2*3 +1 = 7

7 is A

Right child:

2*3 +2 = 8

8 is C

Parent:

(3-1)/2= 1

1 is D

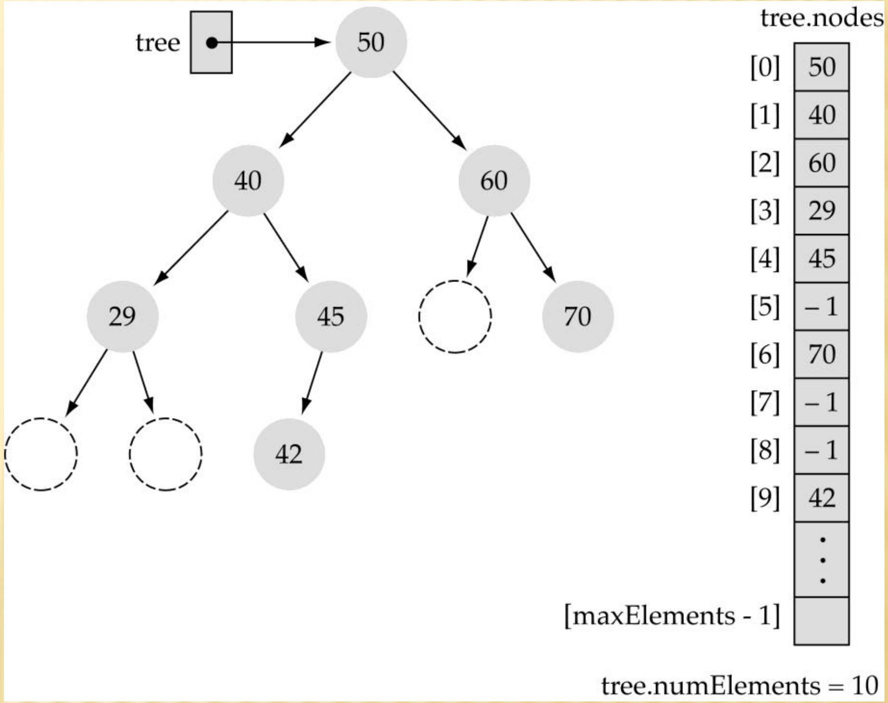
Leaf nodes:

Between (10/2) & 9 => 5 & 9

They are H,J,A,C,E

ARRAY-BASED REPRESENTATION OF BINARY TREES (CONT.)

- Full or complete trees can be implemented easily using an array-based representation (elements occupy contiguous array slots)
- "Dummy nodes" are required for trees which are not full or complete

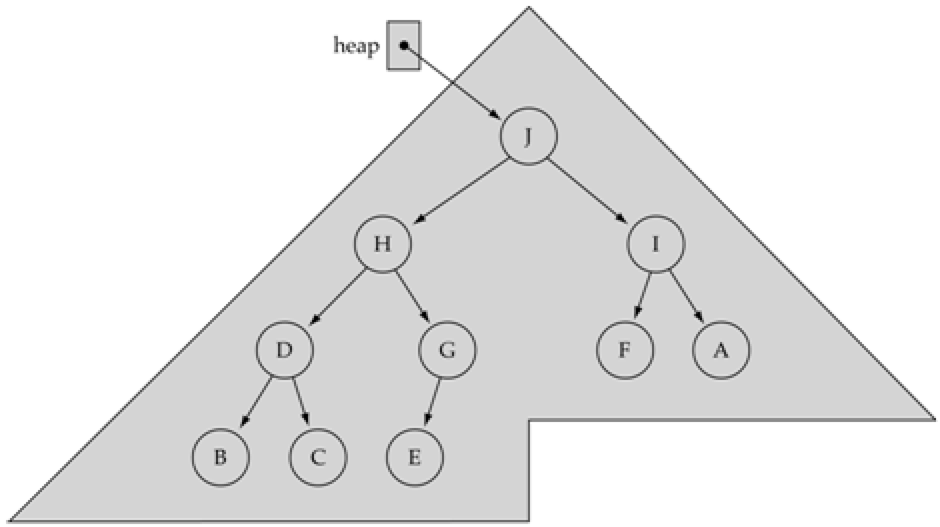


A HEAP

- It is a binary tree with the following properties:

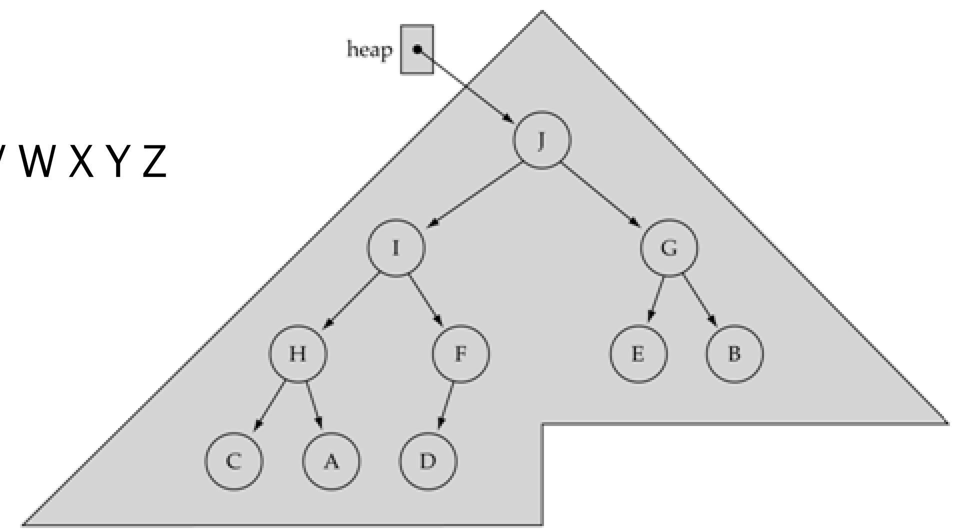
1. It is a complete binary tree.
2. The value stored at a node is greater or equal to the values stored at the children (heap property)

WHAT IS A HEAP? (CONT.)



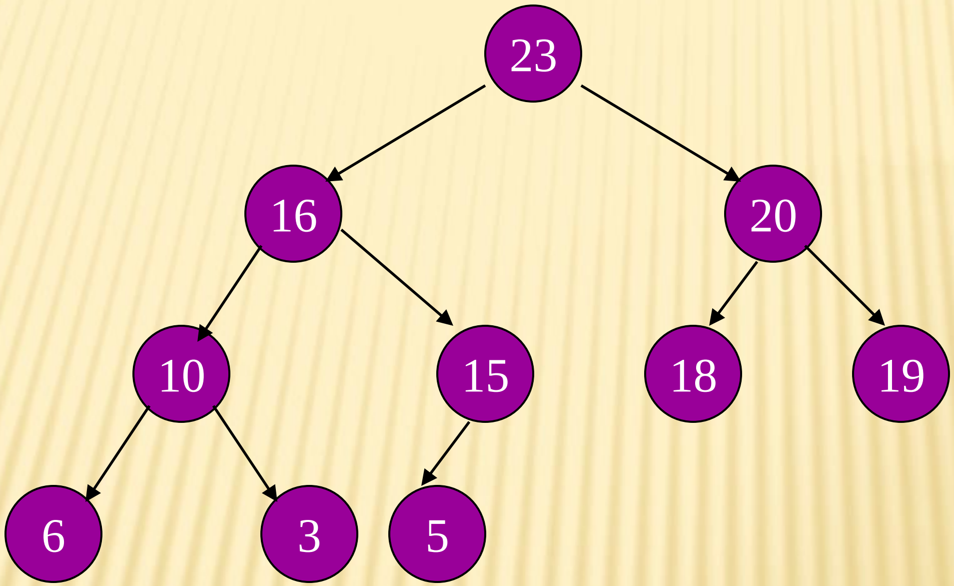
(a)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

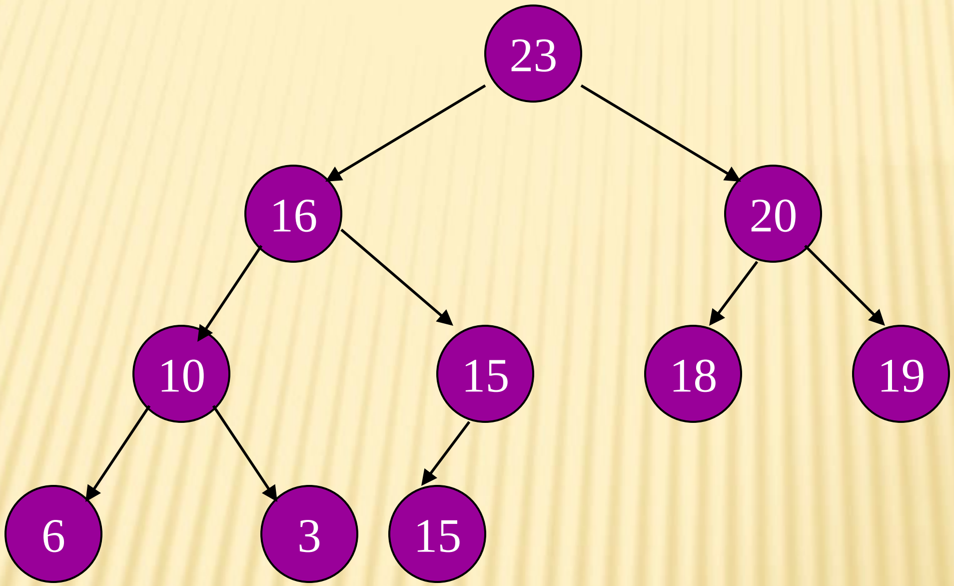


(b)

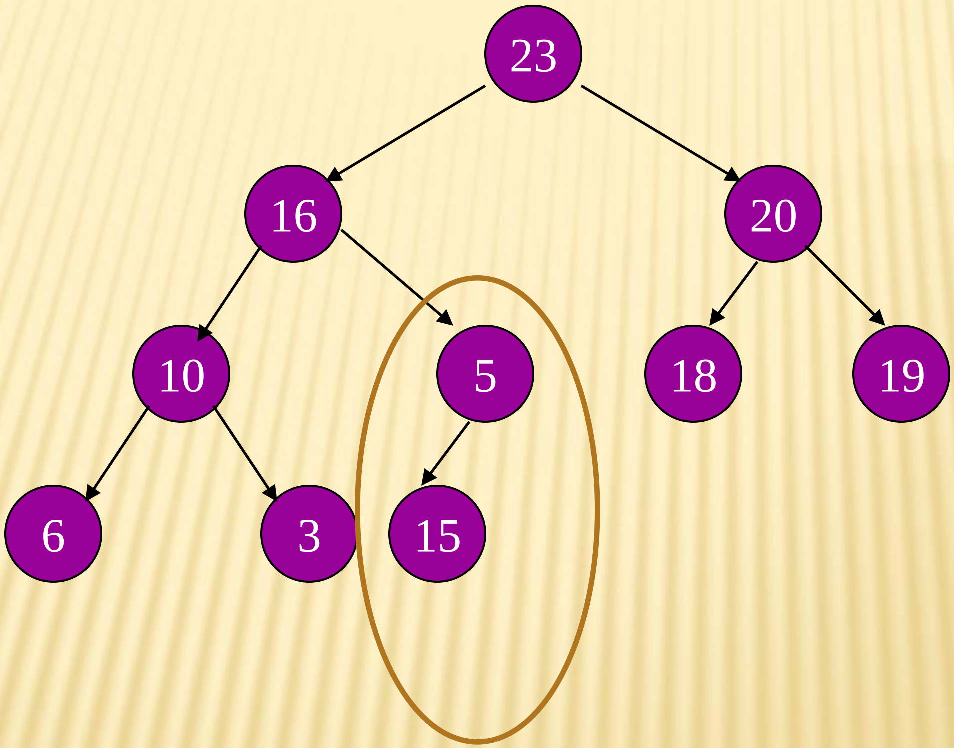
WHAT IS A HEAP? (CONT.)



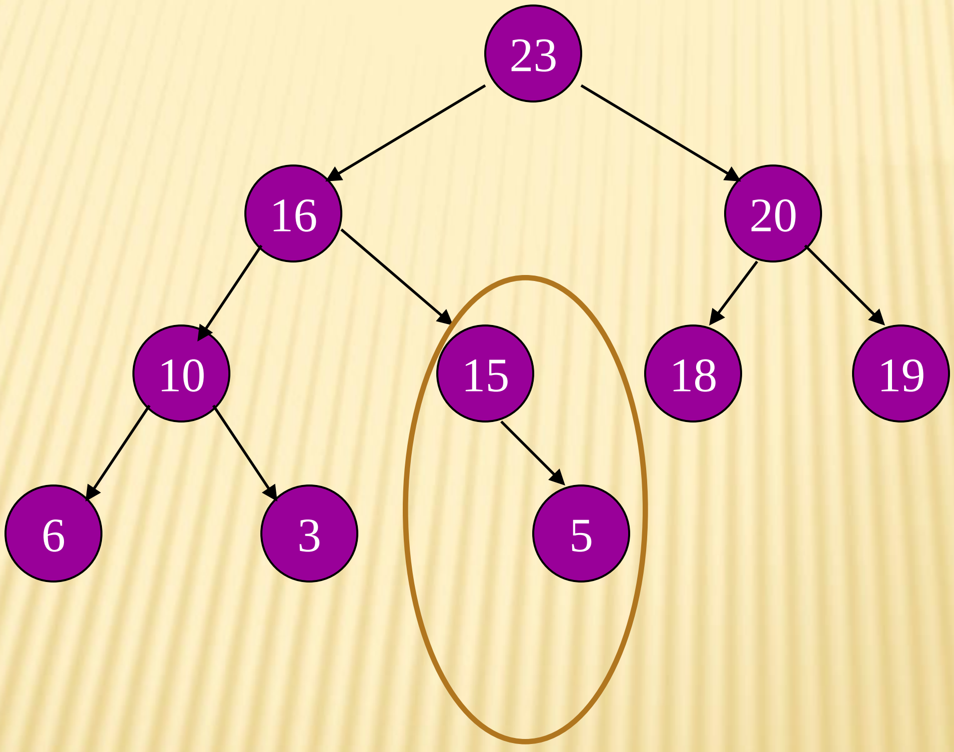
IS IT A HEAP?



IS IT A HEAP?

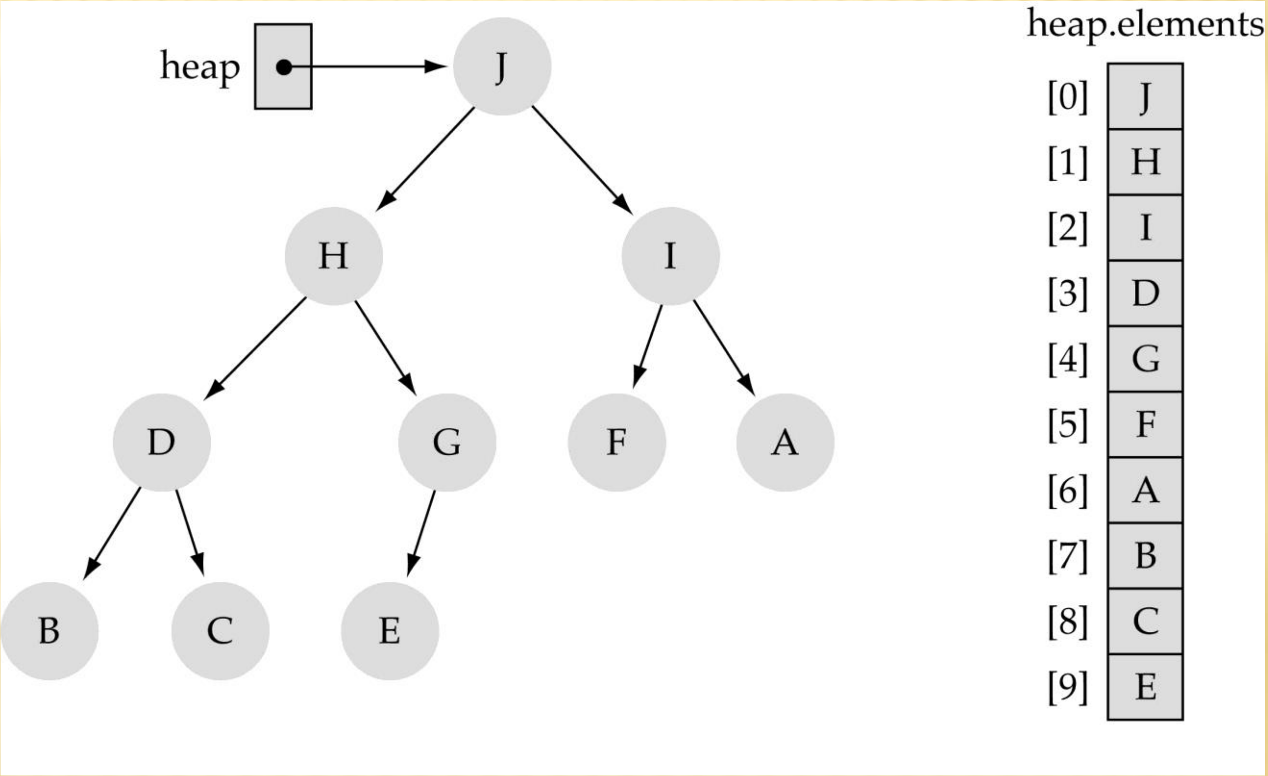


IS IT A HEAP?



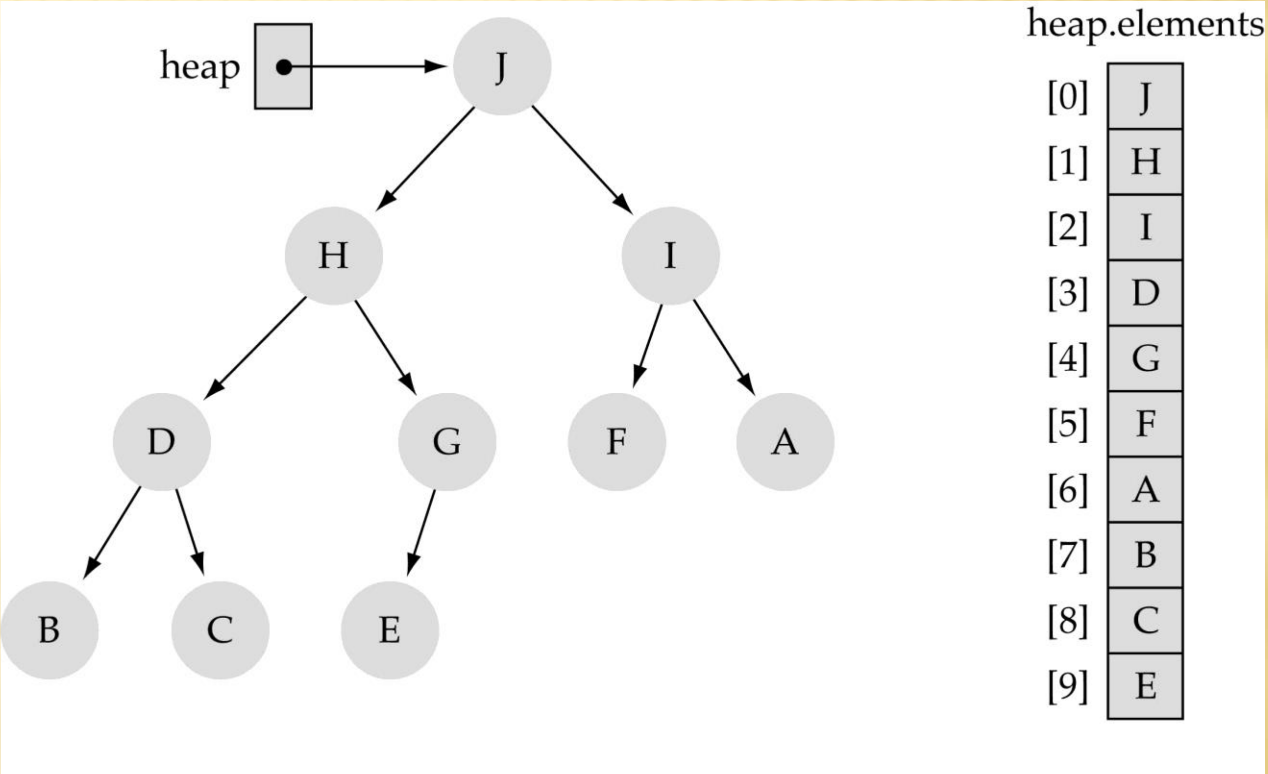
LARGEST HEAP ELEMENT

- From *Property 2*, the largest value of the heap is always stored at the root



HEAP IMPLEMENTATION USING ARRAY REPRESENTATION

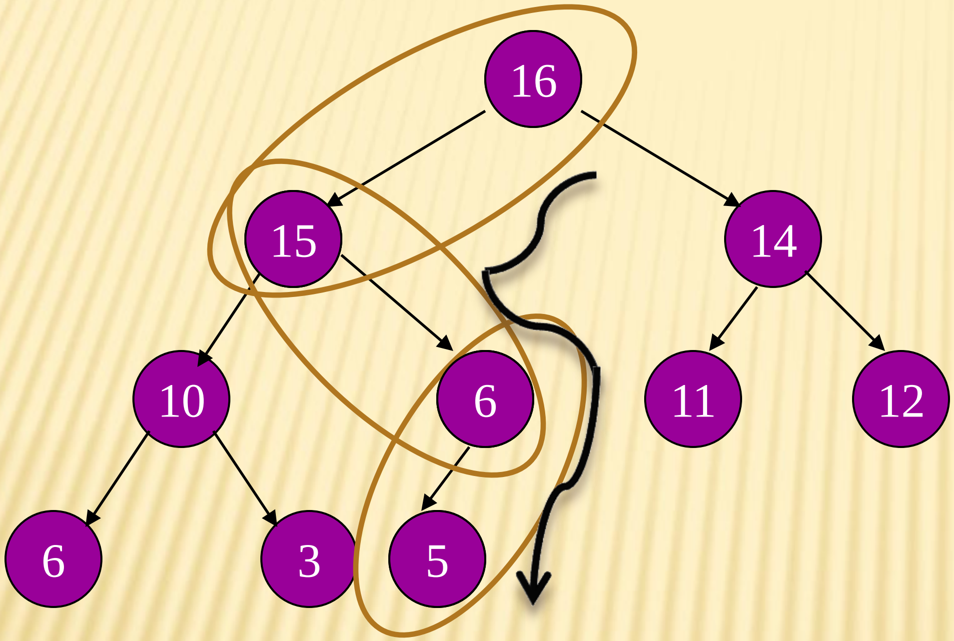
- A heap is a complete binary tree, so it is easy to be implemented using an array representation



HEAP SPECIFICATION

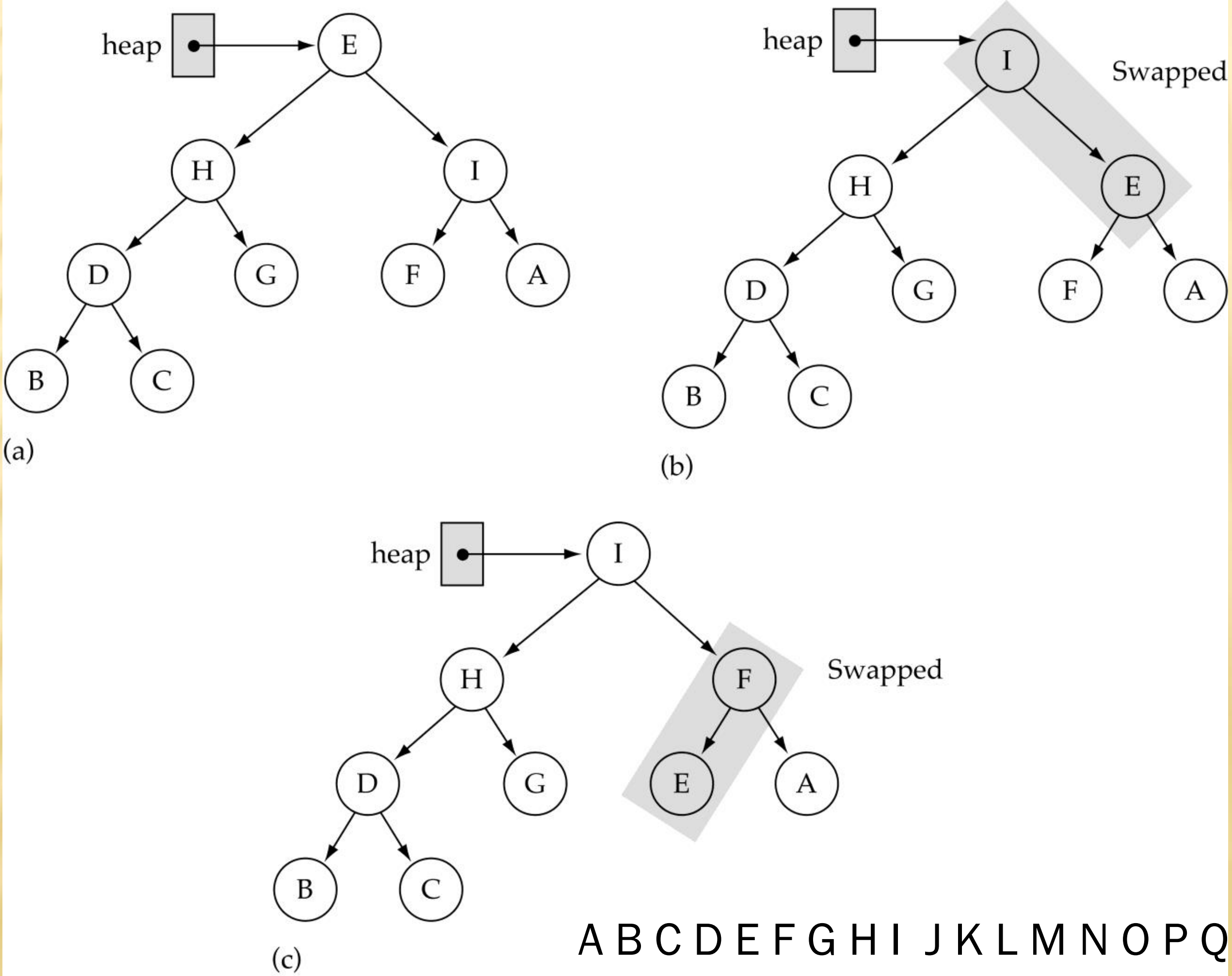
```
template<class ItemType>
struct HeapType {
    void ReheapDown(int, int);
    void ReheapUp(int, int);
    ItemType *elements;
    int numElements; // heap elements
};
```

THE REHEAPDOWN FUNCTION (USED BY DELETEITEM)



Assumption:
heap property is violated at the root of the tree

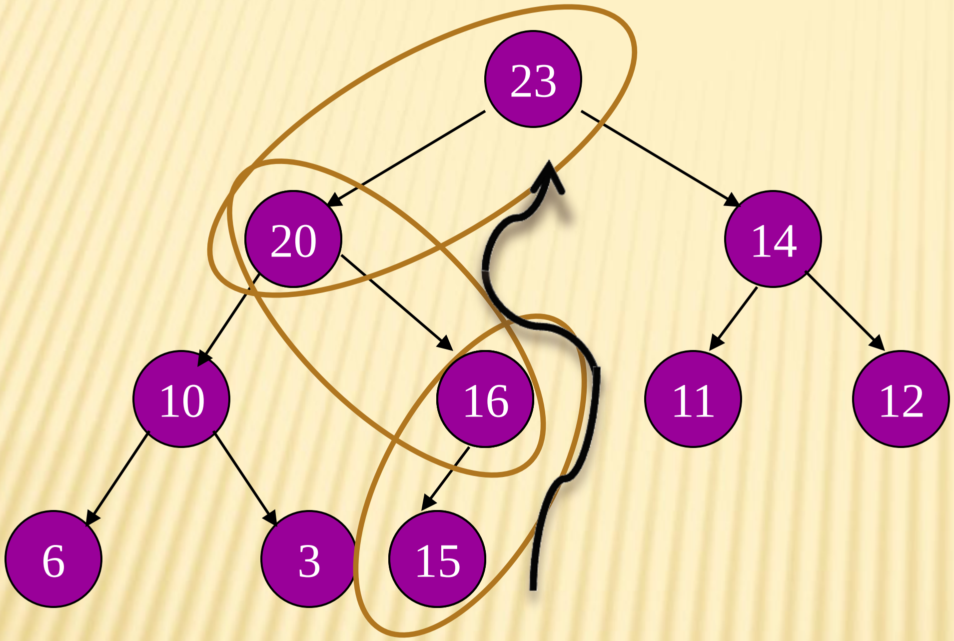
THE REHEAPDOWN FUNCTION (USED BY DELETEITEM)



A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

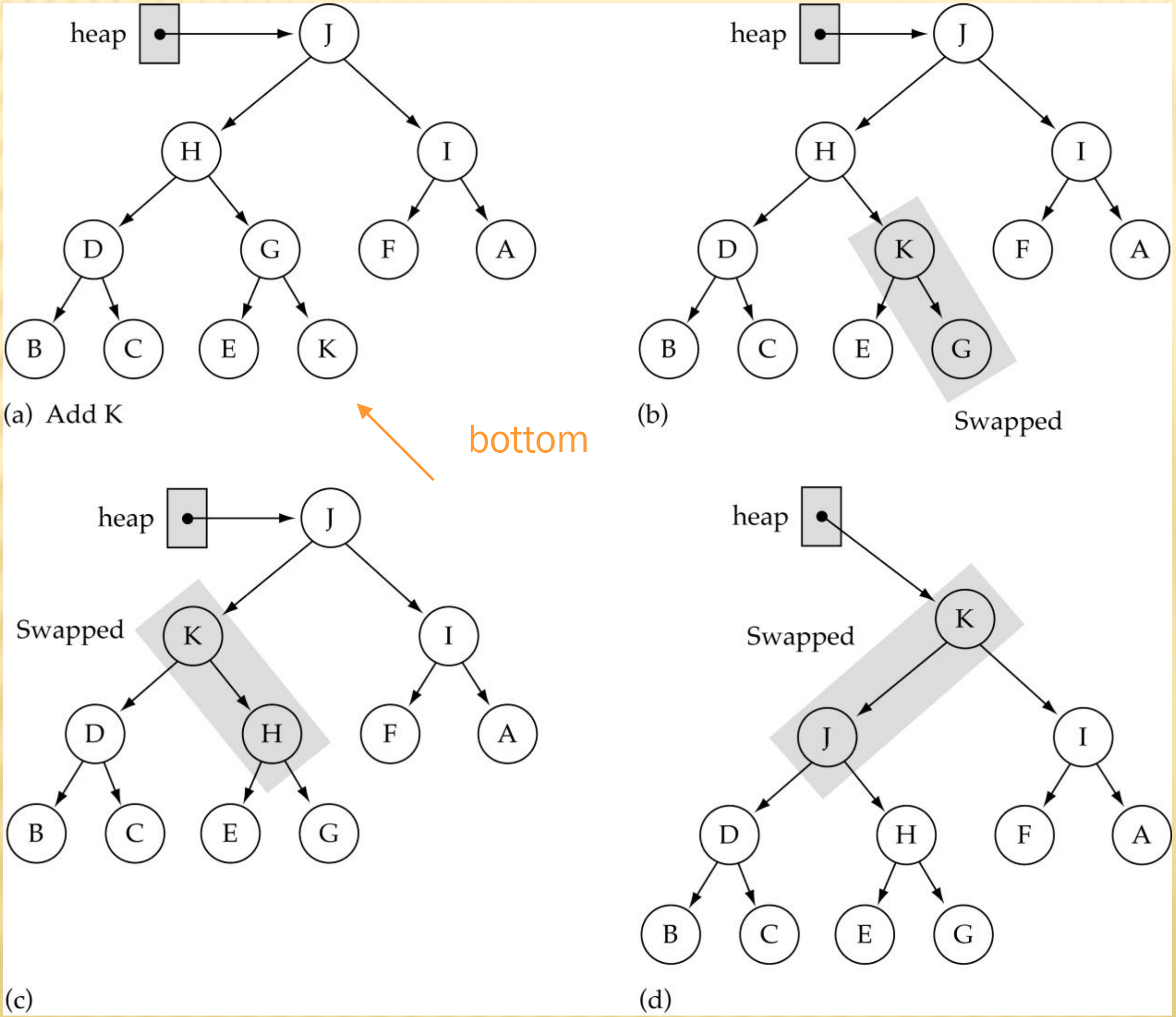
Assumption: heap property is violated at the root of the tree

THE REHEAPUP FUNCTION (USED BY INSERTITEM)



Assumption: heap property is violated at the rightmost node at the last level of the tree

THE REHEAPUP FUNCTION (USED BY INSERTITEM)



Assumption: heap property is violated at the rightmost node at the last level of the tree

REHEAPDOWN FUNCTION

```
template<class ItemType>
void HeapType<ItemType>::ReheapDown(int root, int bottom)
{
    int maxChild, rightChild, leftChild;

    leftChild = 2*root+1;
    rightChild = 2*root+2;

    if(leftChild <= bottom) {  // left child is part of the heap
        if(leftChild == bottom) // only one child
            maxChild = leftChild;
        else {
            if(elements[leftChild] <= elements[rightChild])
                maxChild = rightChild;
            else
                maxChild = leftChild;
        }
        if(elements[root] < elements[maxChild]) {
            Swap(elements, root, maxChild);
            ReheapDown(maxChild, bottom);
        }
    }
}
```

rightmost node
in the last level

REHEAPUP FUNCTION

Assumption:
heap property
is violated at bottom

```
template<class ItemType>
void HeapType<ItemType>::ReheapUp(int root, int bottom)
{
    int parent;

    if(bottom > root) { // tree is not empty
        parent = (bottom-1)/2;
        if(elements[parent] < elements[bottom]) {
            Swap(elements, parent, bottom);
            ReheapUp(root, parent);
        }
    }
}
```

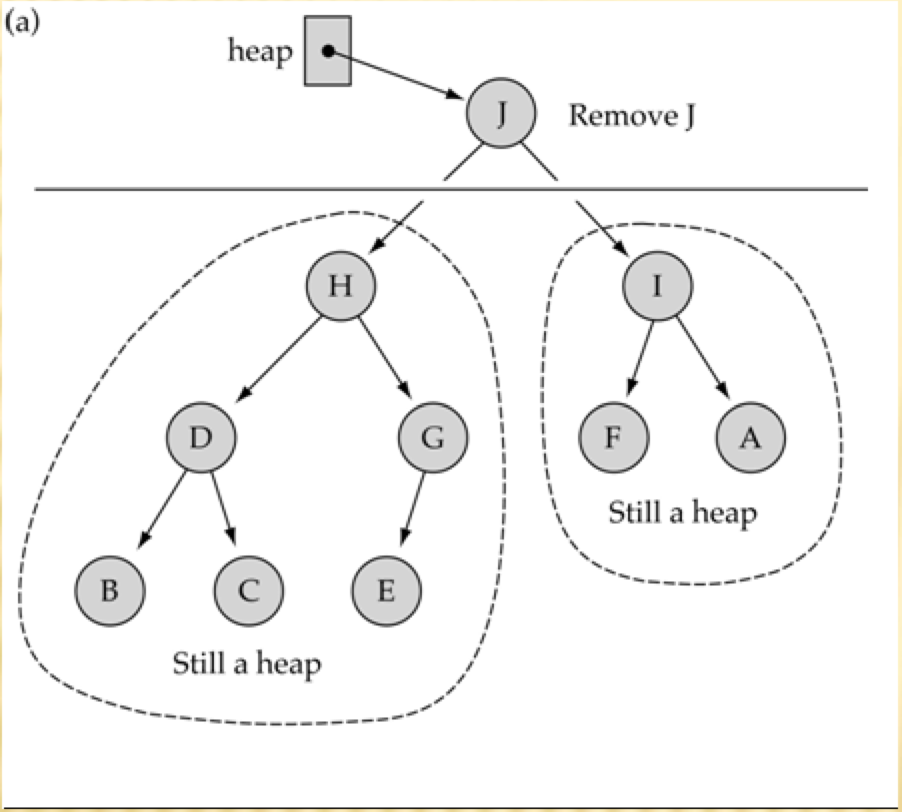
REMOVING THE LARGEST ELEMENT FROM THE HEAP

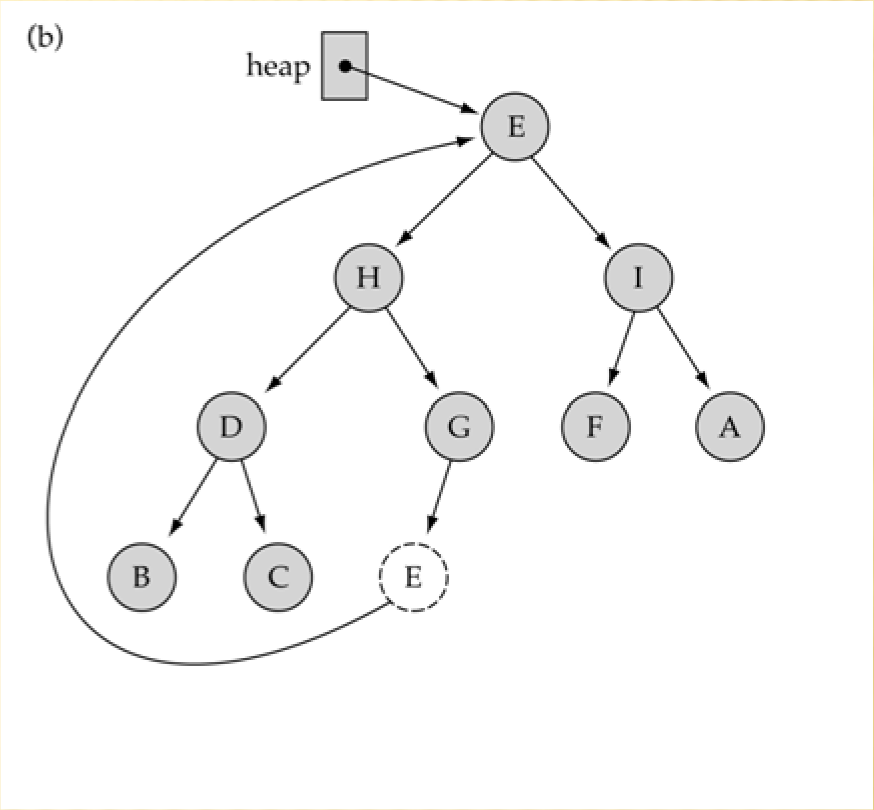
1) Copy the bottom rightmost element to the root

2) Delete the bottom rightmost node

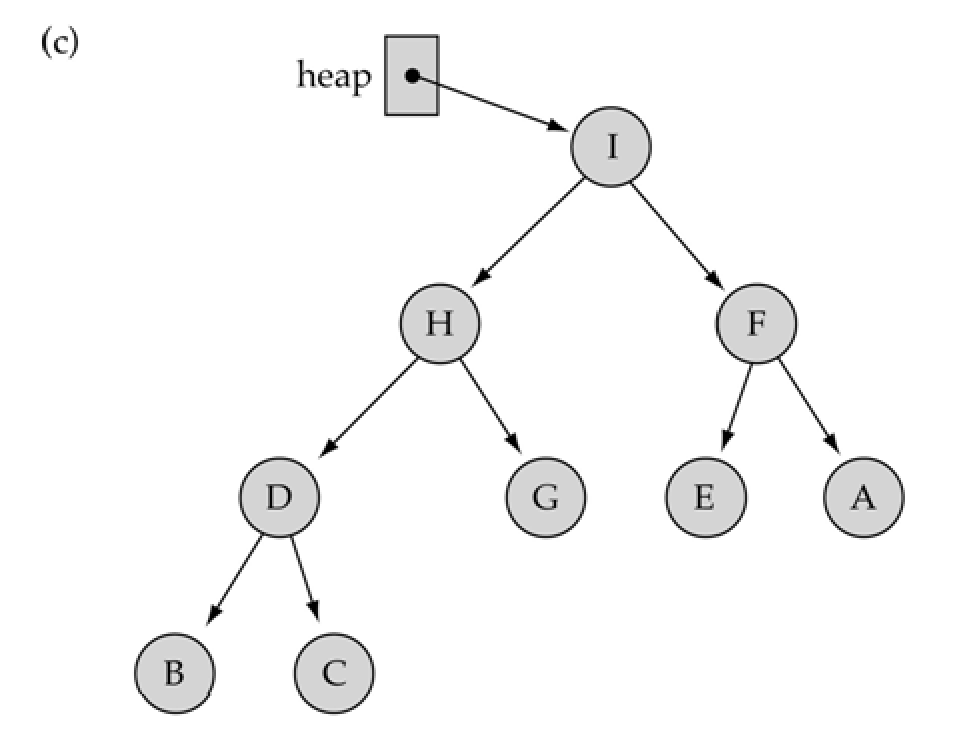
3) Fix the heap property by calling *ReheapDown*

REMOVING THE LARGEST ELEMENT FROM THE HEAP (CONT.)





REMOVING THE LARGEST ELEMENT FROM THE HEAP (CONT.)

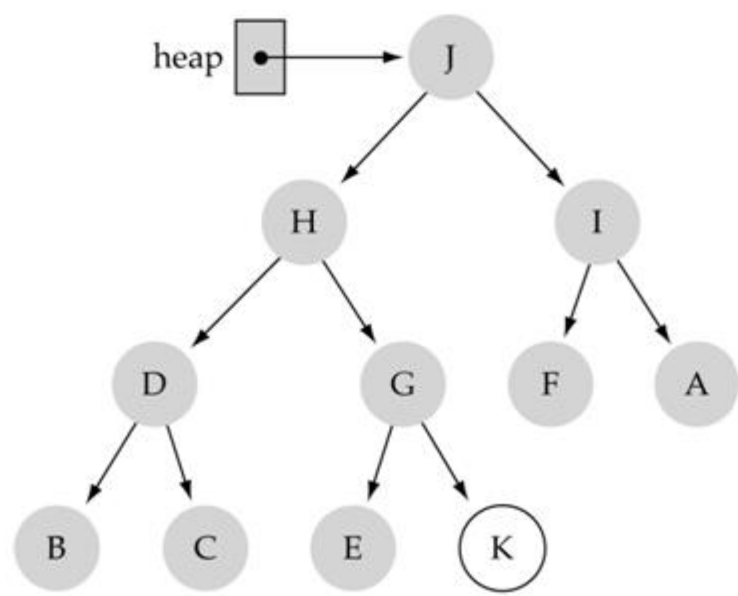


INSERTING A NEW ELEMENT INTO THE HEAP

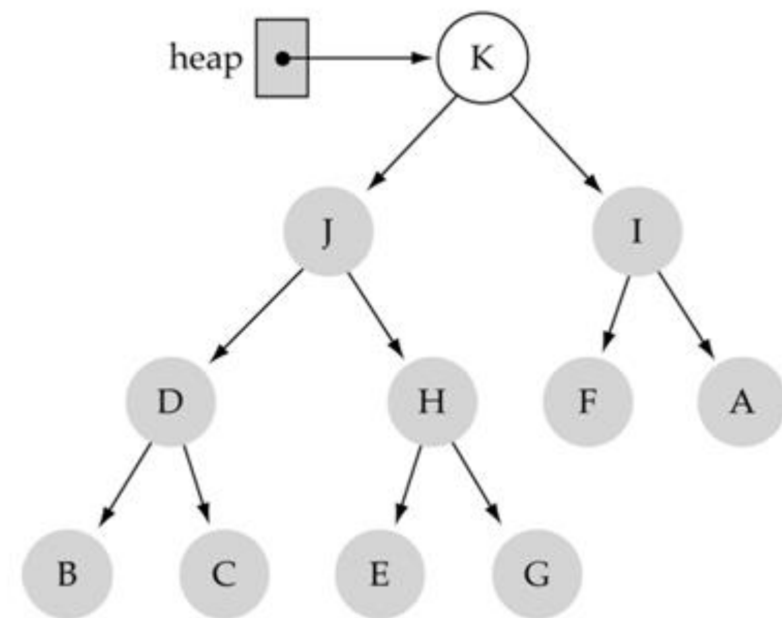
1) Insert the new element in the next bottom leftmost place

2) Fix the heap property by calling *ReheapUp*

INSERTING A NEW ELEMENT INTO THE HEAP (CONT.)



(a) Add K



(b) ReheapUp

SORTING

- Sorting rearranges the elements into either ascending or descending order within the array. (we'll use ascending order.)

2	3	6	7	10	20	22	45	90

210	193	186	177	110	20	12	5	0

HEAP SORT APPROACH

- First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

  - Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.

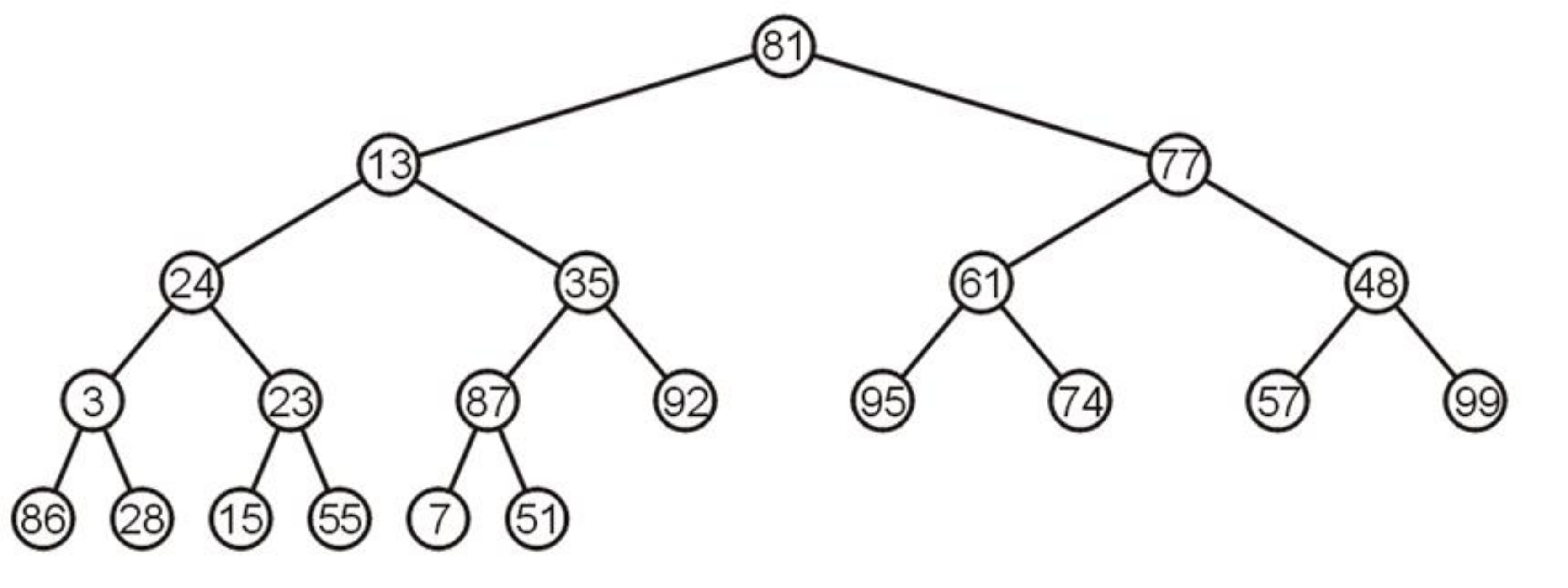
  - Reheap the remaining unsorted elements.

  (This puts the next-largest element into the root position).

BUILDING THE HEAP

numElements = 21

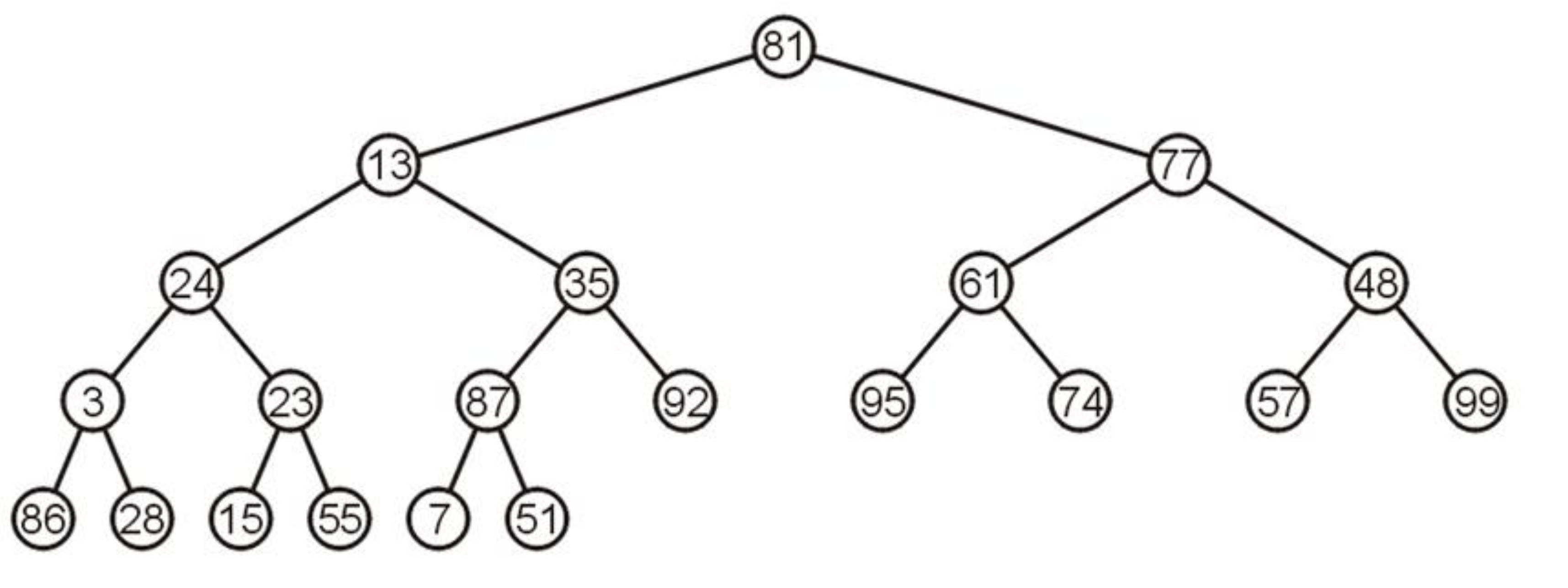
- Consider the following unsorted array



81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

BUILDING THE HEAP

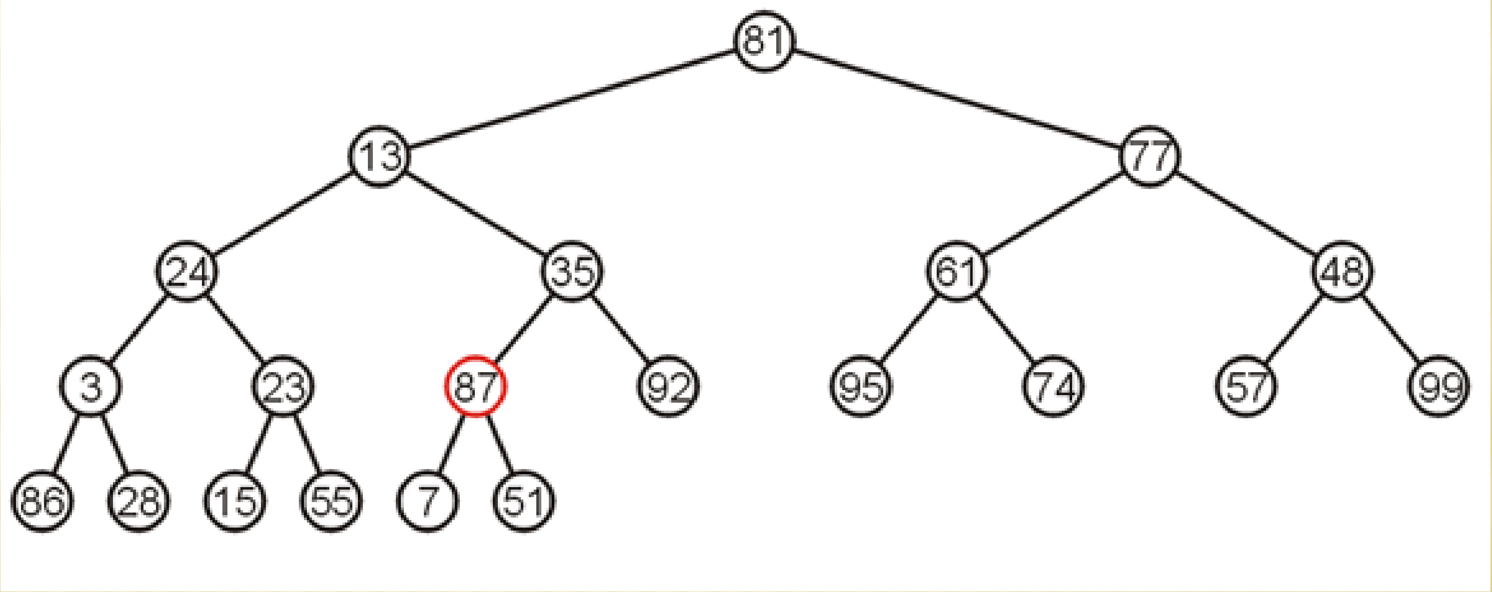
- all leaf nodes are trivial heaps
- Leaf nodes between: 21/2 to 20 → 10 to 20



81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

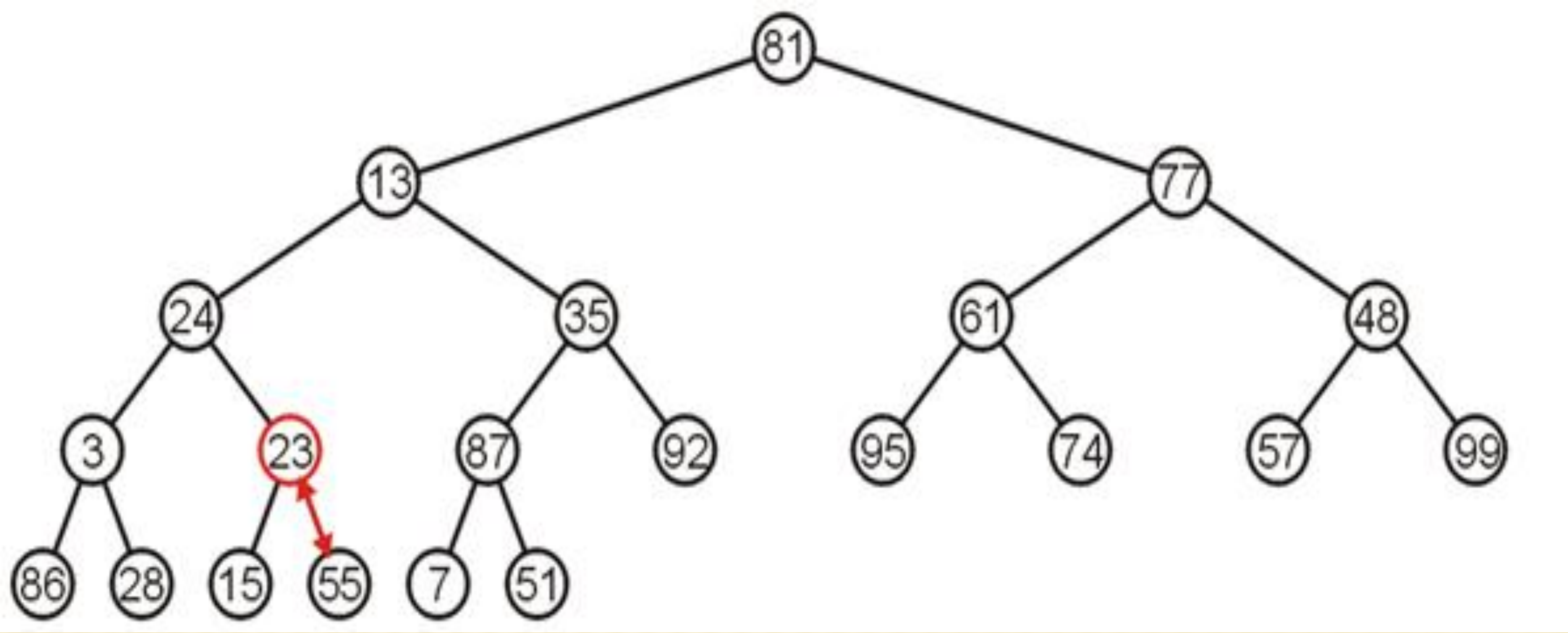
BUILDING THE HEAP

- Reheapdown every non leaf node (starting from 2nd last level (right to left))
- The subtree with 87 as the root is a max-heap



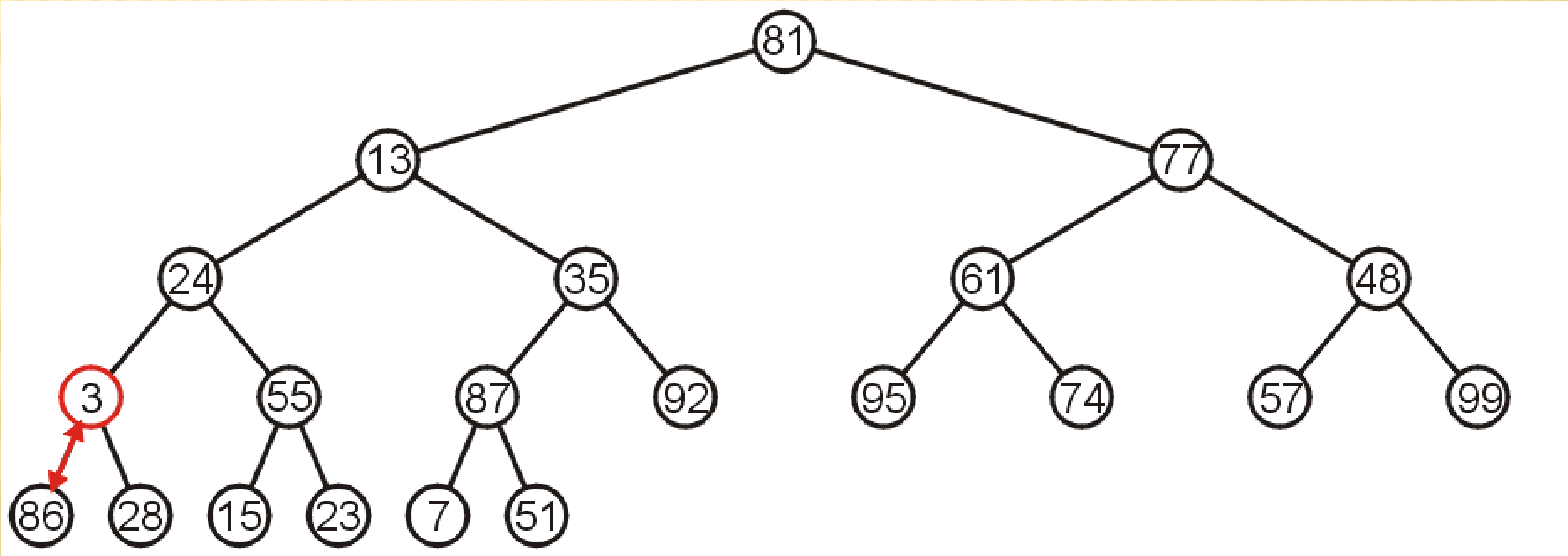
BUILDING THE HEAP

- The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap



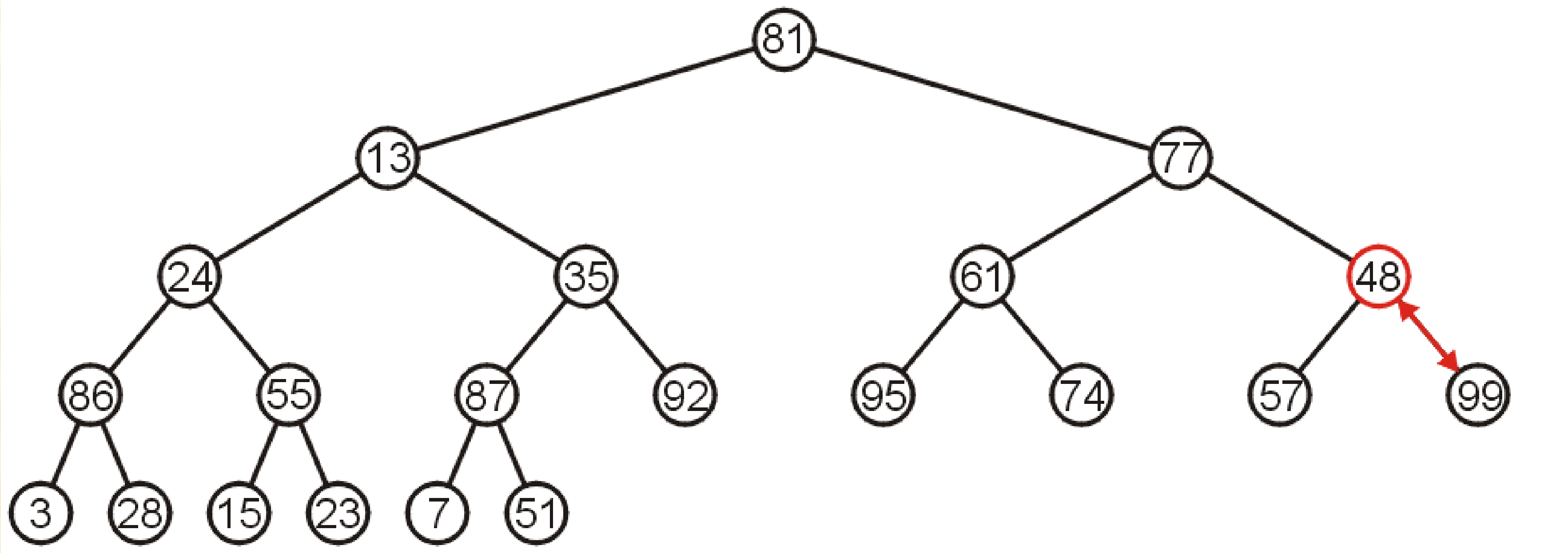
BUILDING THE HEAP

- The subtree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86



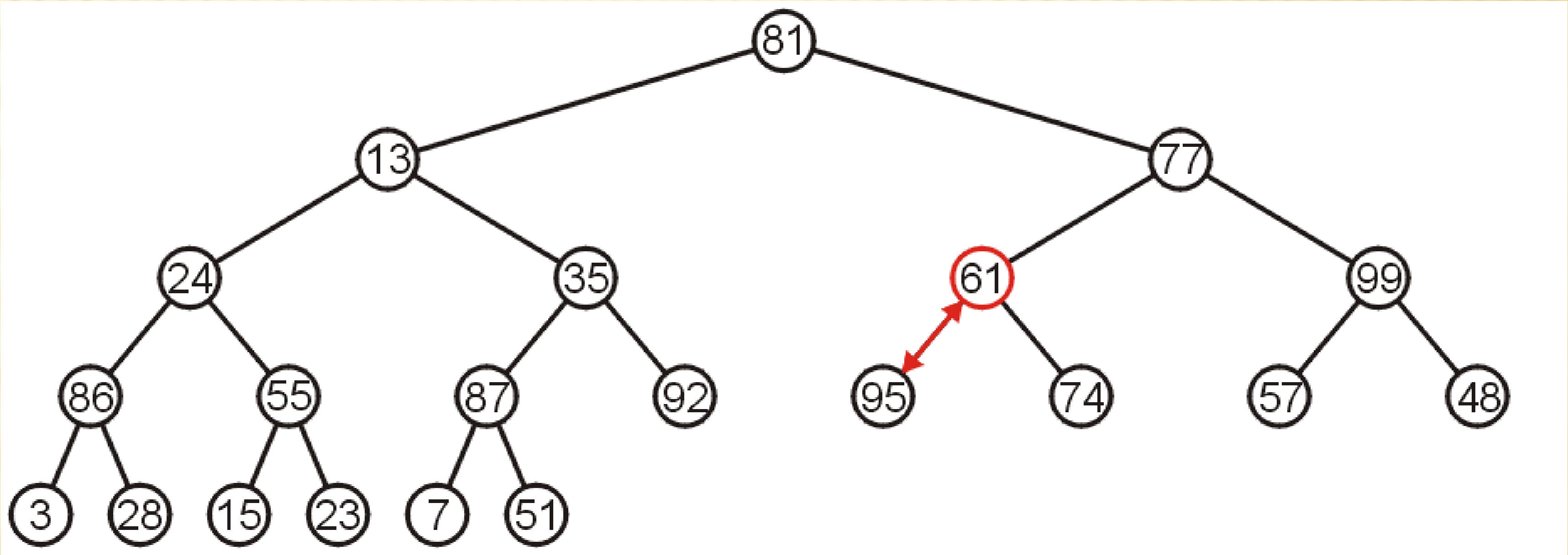
BUILDING THE HEAP

- Starting with the next higher level, the subtree with root 48 can be turned into a max-heap by swapping 48 and 99



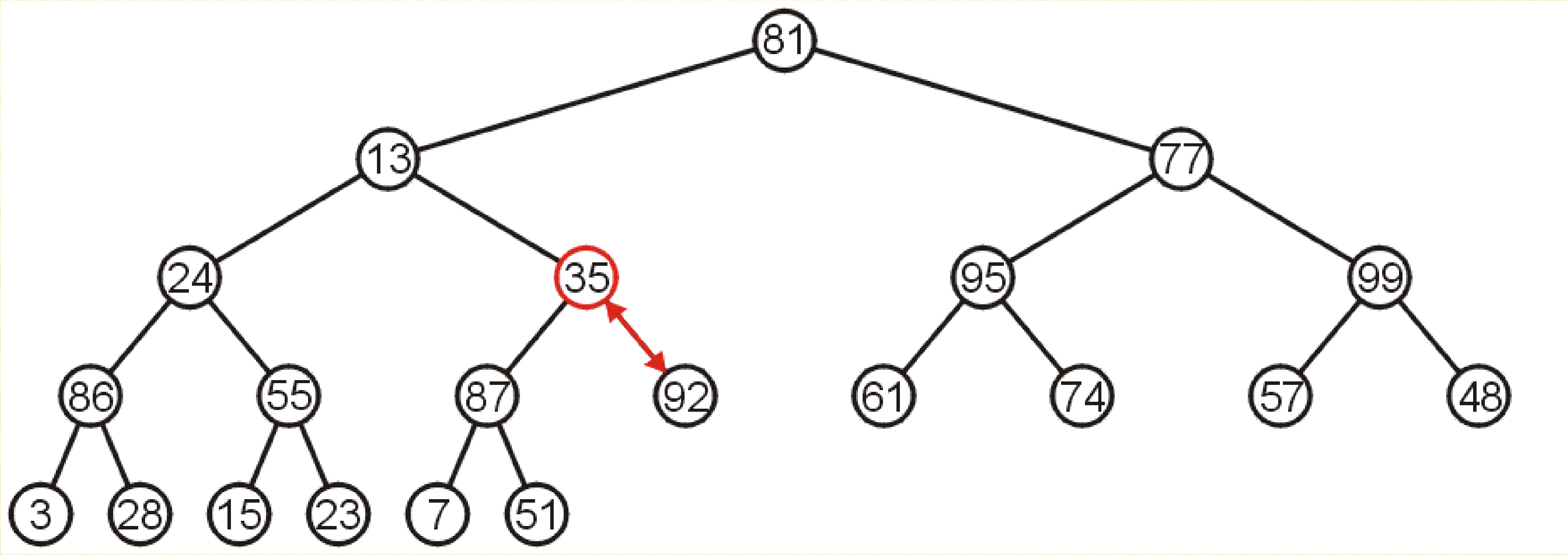
BUILDING THE HEAP

- Similarly, swapping 61 and 95 creates a max-heap of the next subtree



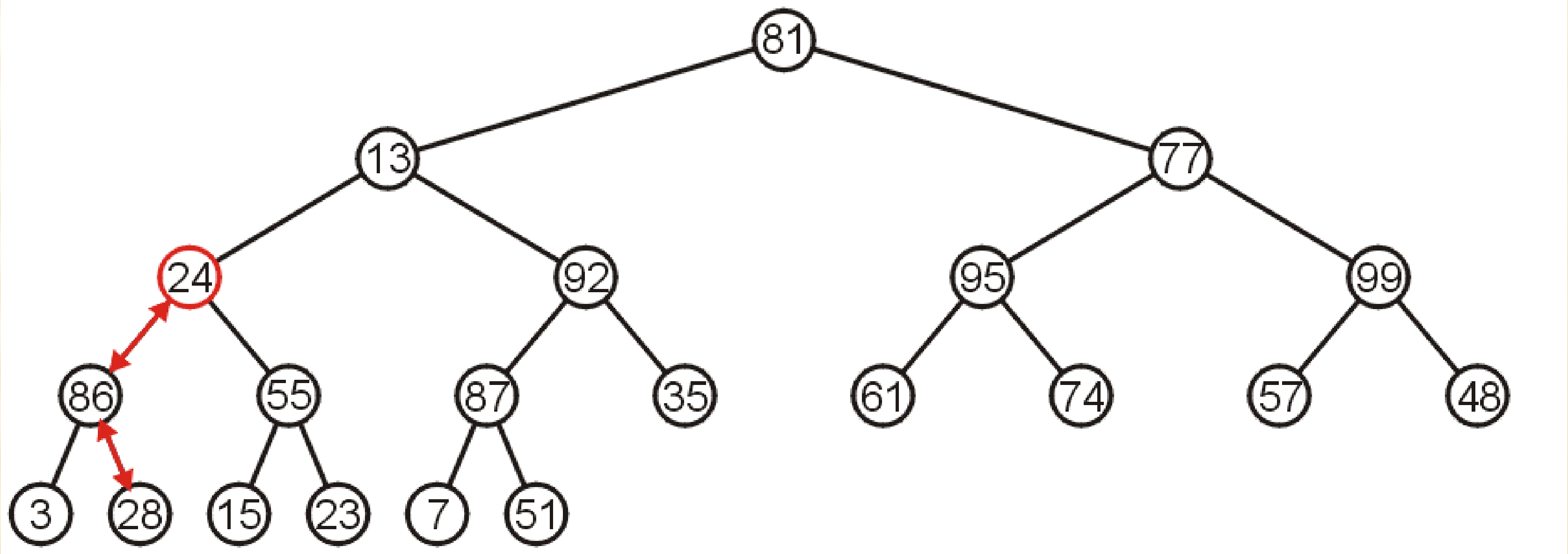
BUILDING THE HEAP

- As does swapping 35 and 92



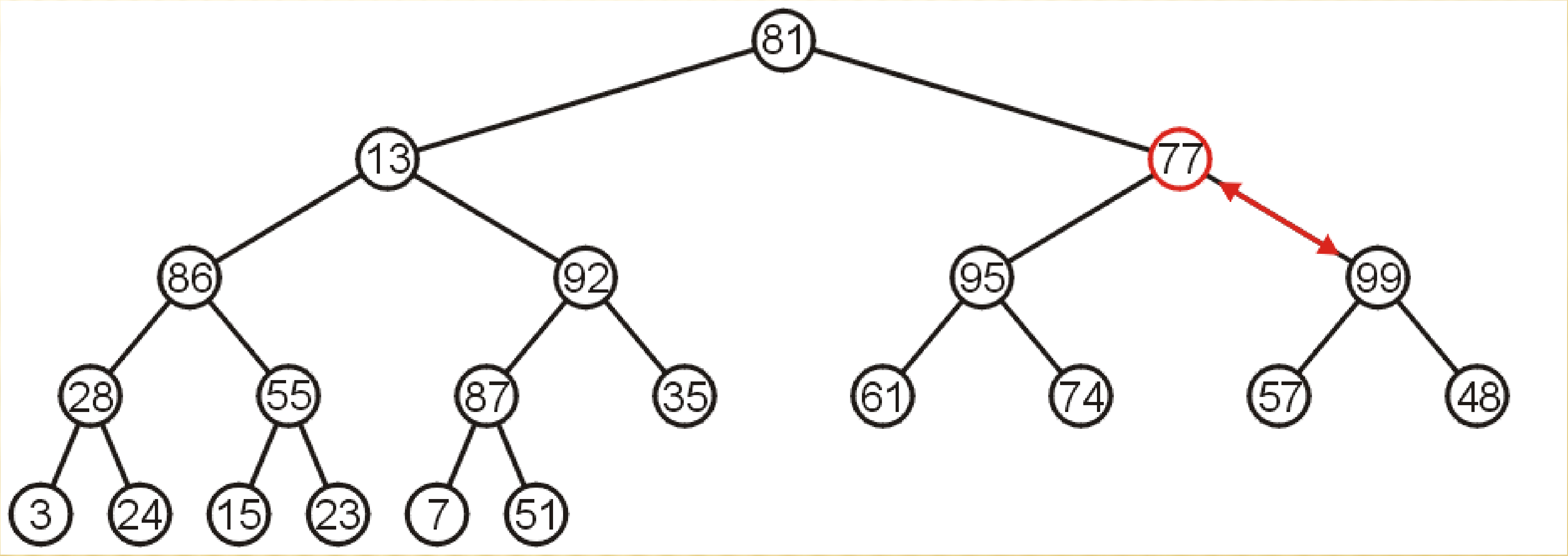
BUILDING THE HEAP

- The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28



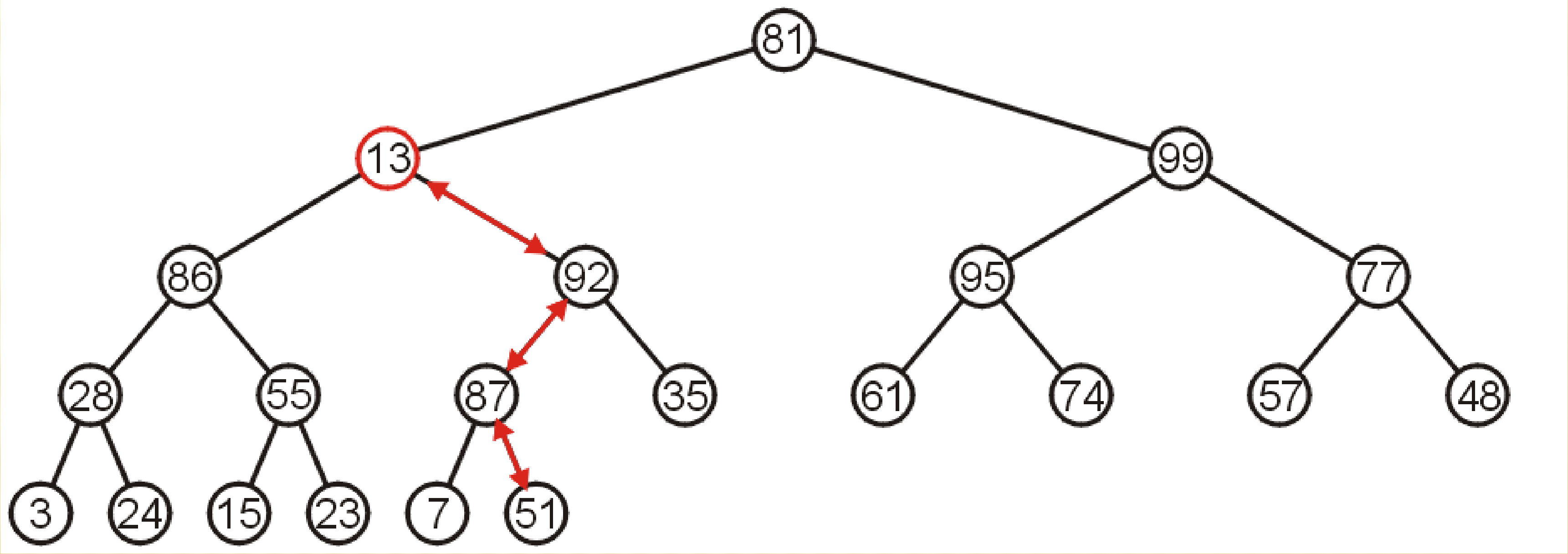
BUILDING THE HEAP

- The right-most subtree of the next higher level may be turned into a max-heap by swapping 77 and 99



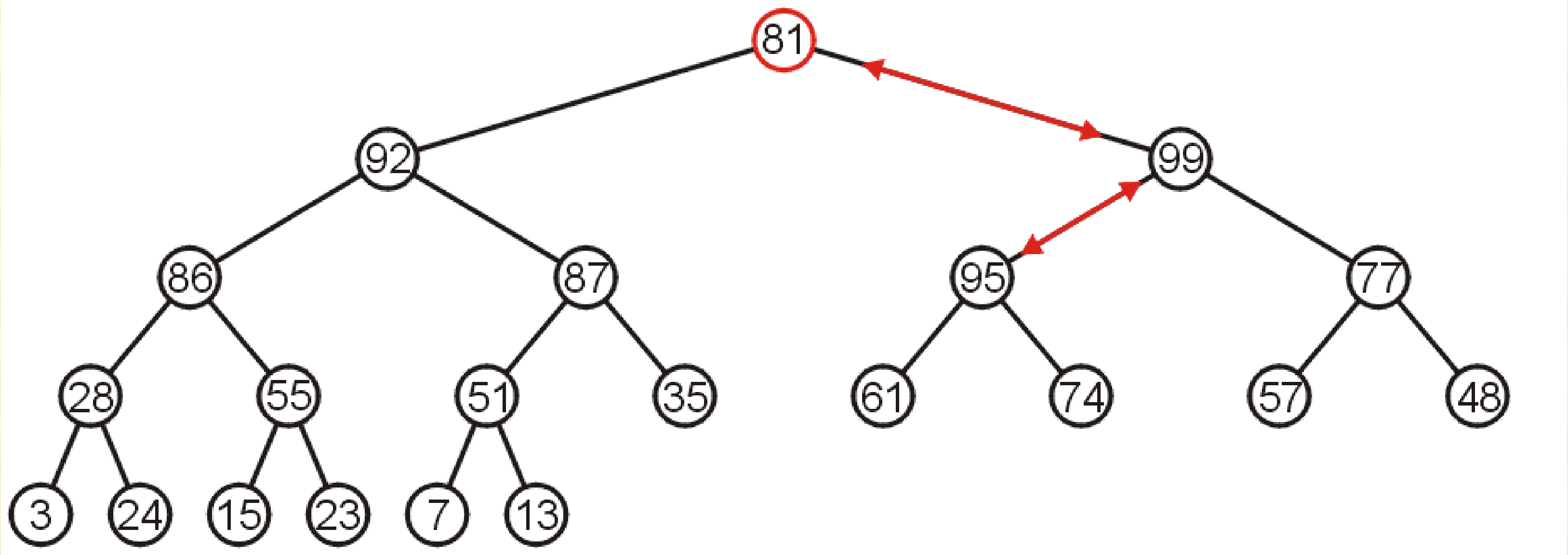
BUILDING THE HEAP

- However, to turn the next subtree into a max-heap requires that 13 be percolated down to a leaf node



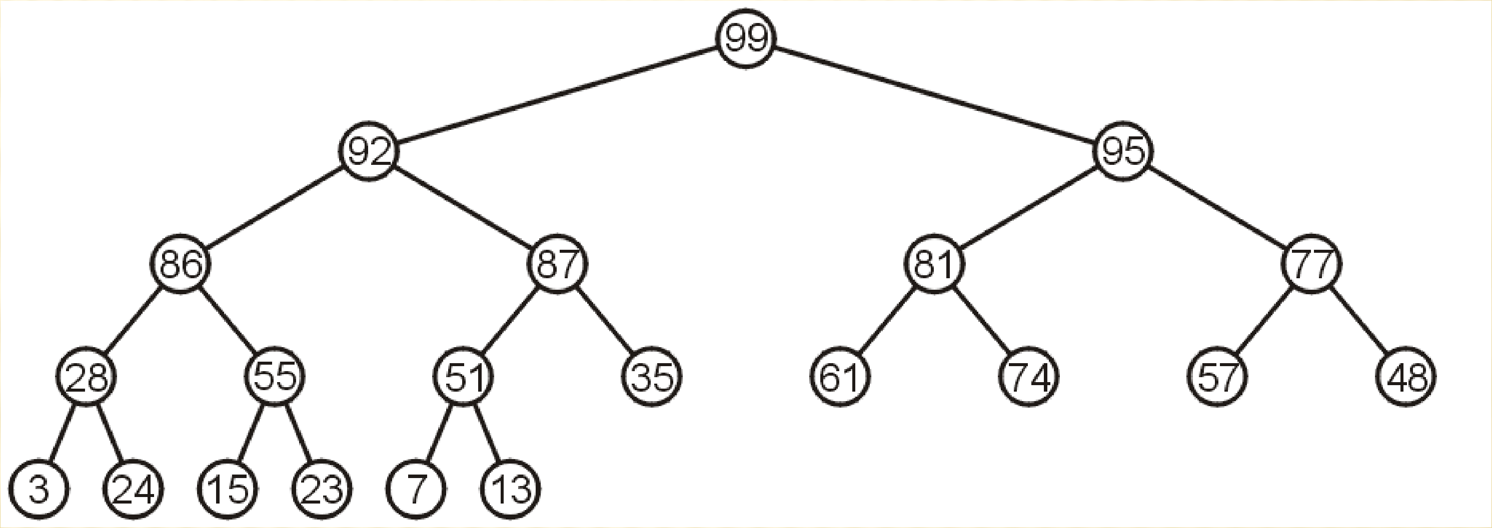
BUILDING THE HEAP

- The root need only be percolated down by two levels



BUILDING THE HEAP

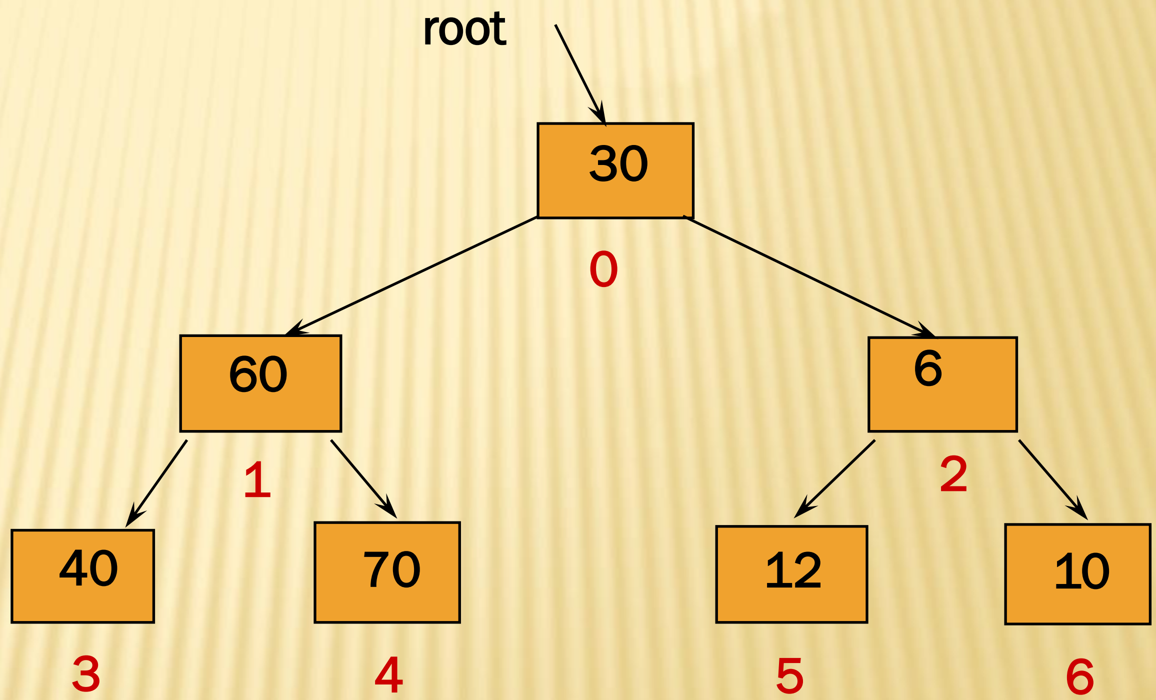
The final product is a max-heap



BUILD HEAP

Build heap of the following unsorted array

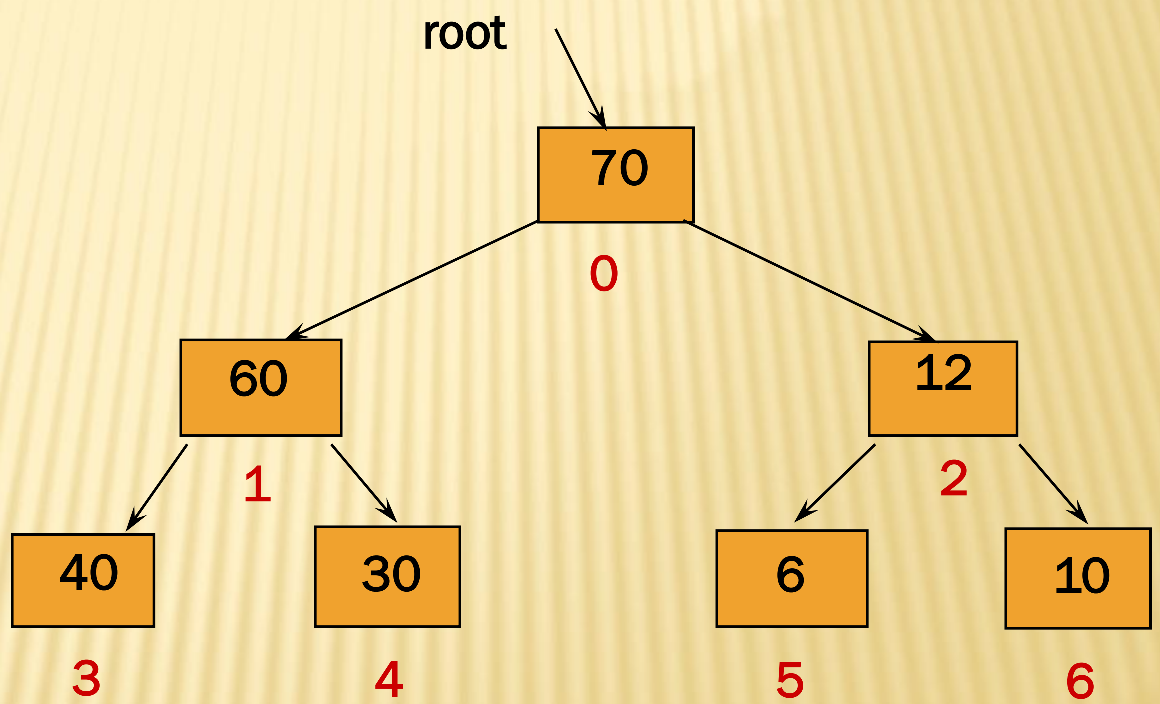
	values
[0]	30
[1]	60
[2]	6
[3]	40
[4]	70
[5]	12
[6]	10



SORTING

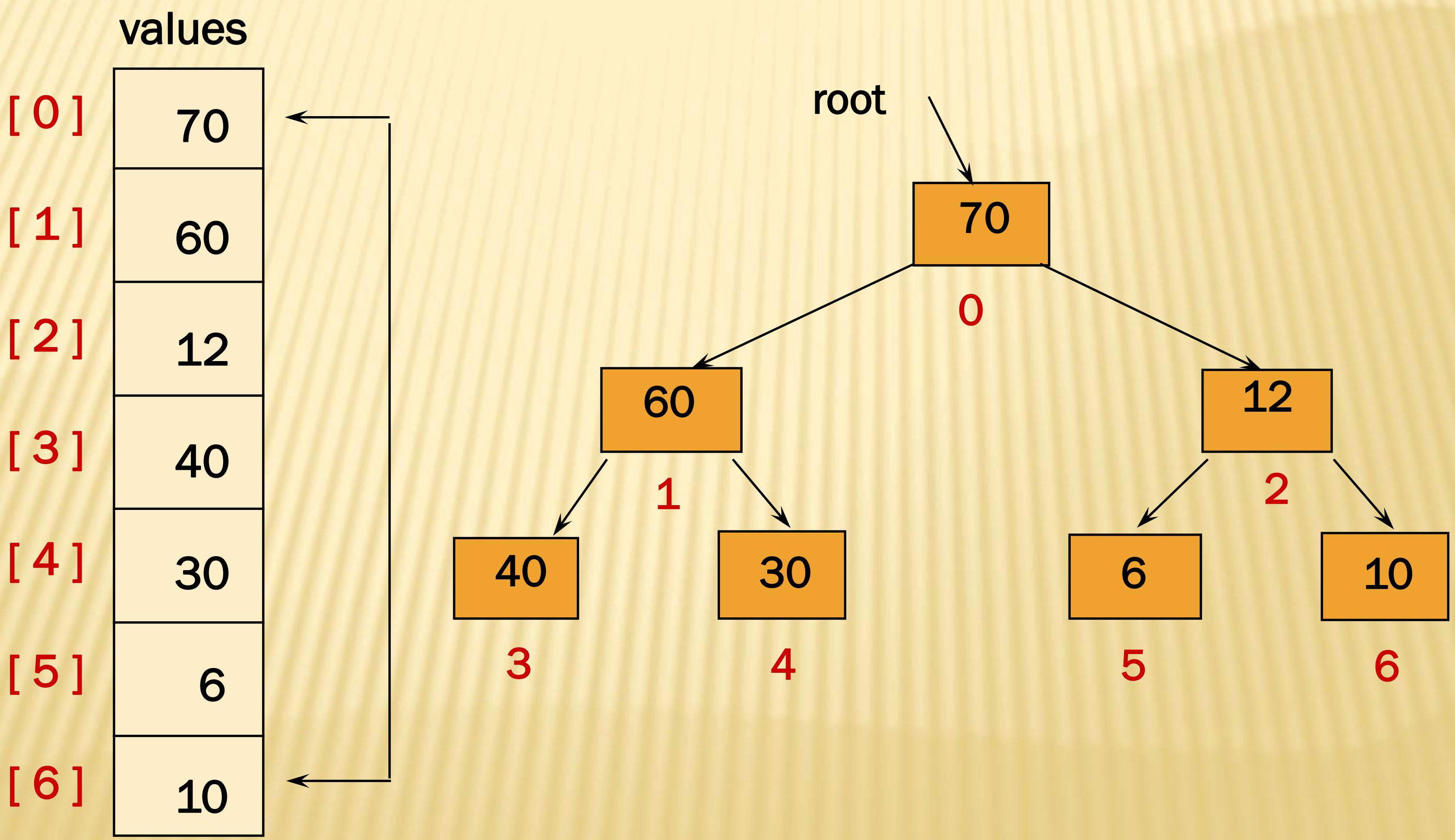
After creating the original heap

	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	6
[6]	10



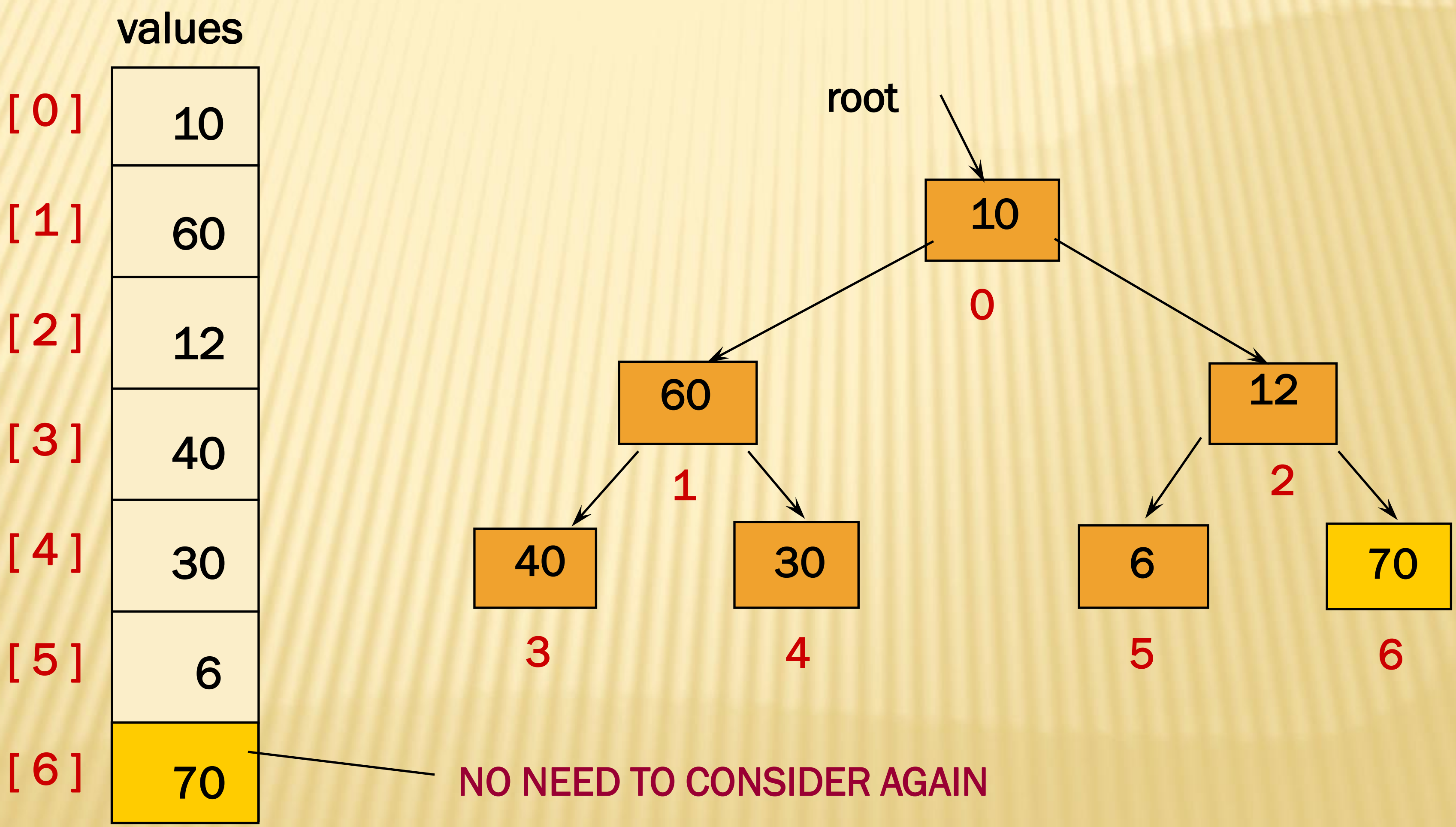
SORTING

Swap root element into last place in unsorted array



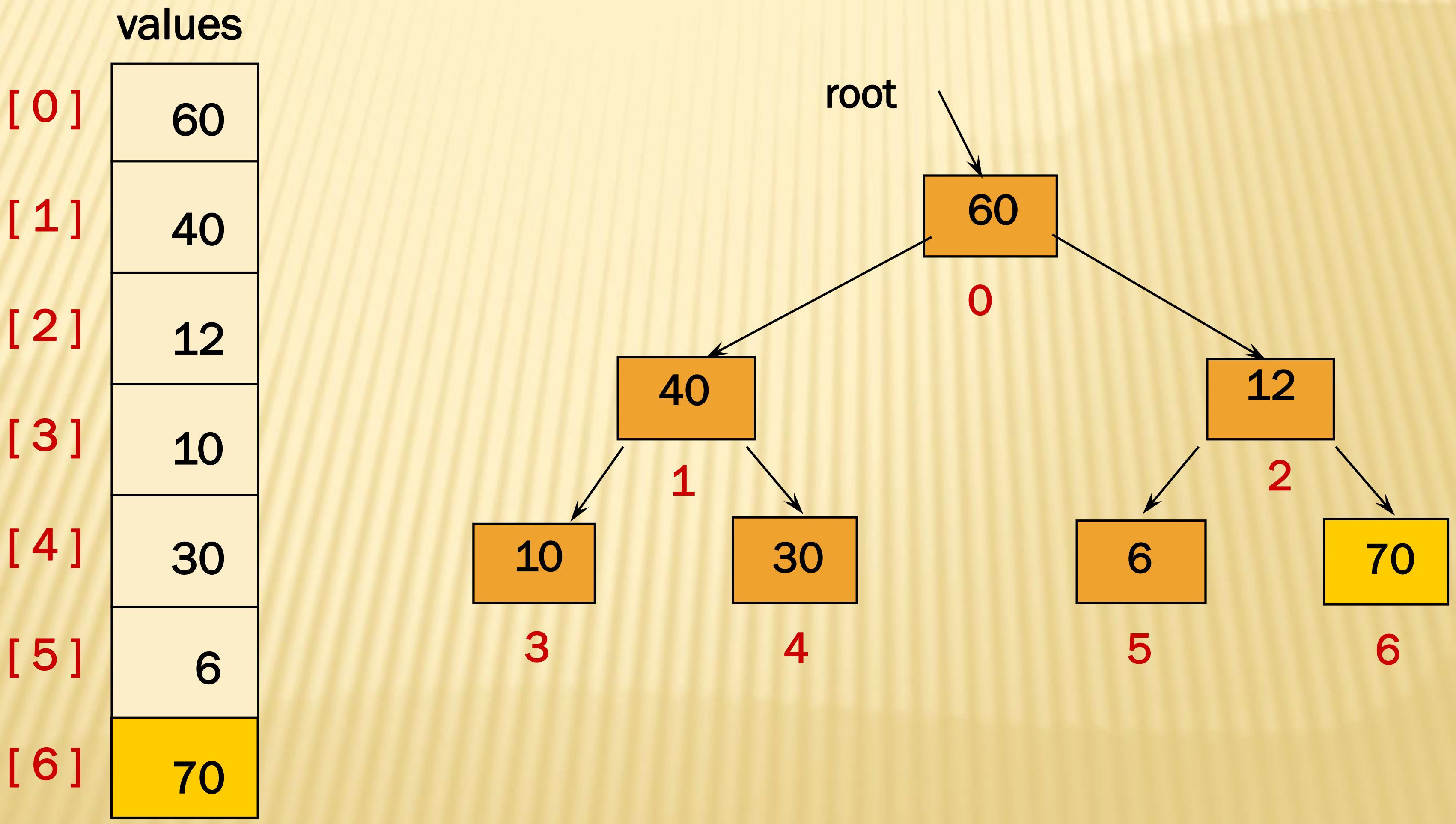
SORTING

After swapping root element into its place



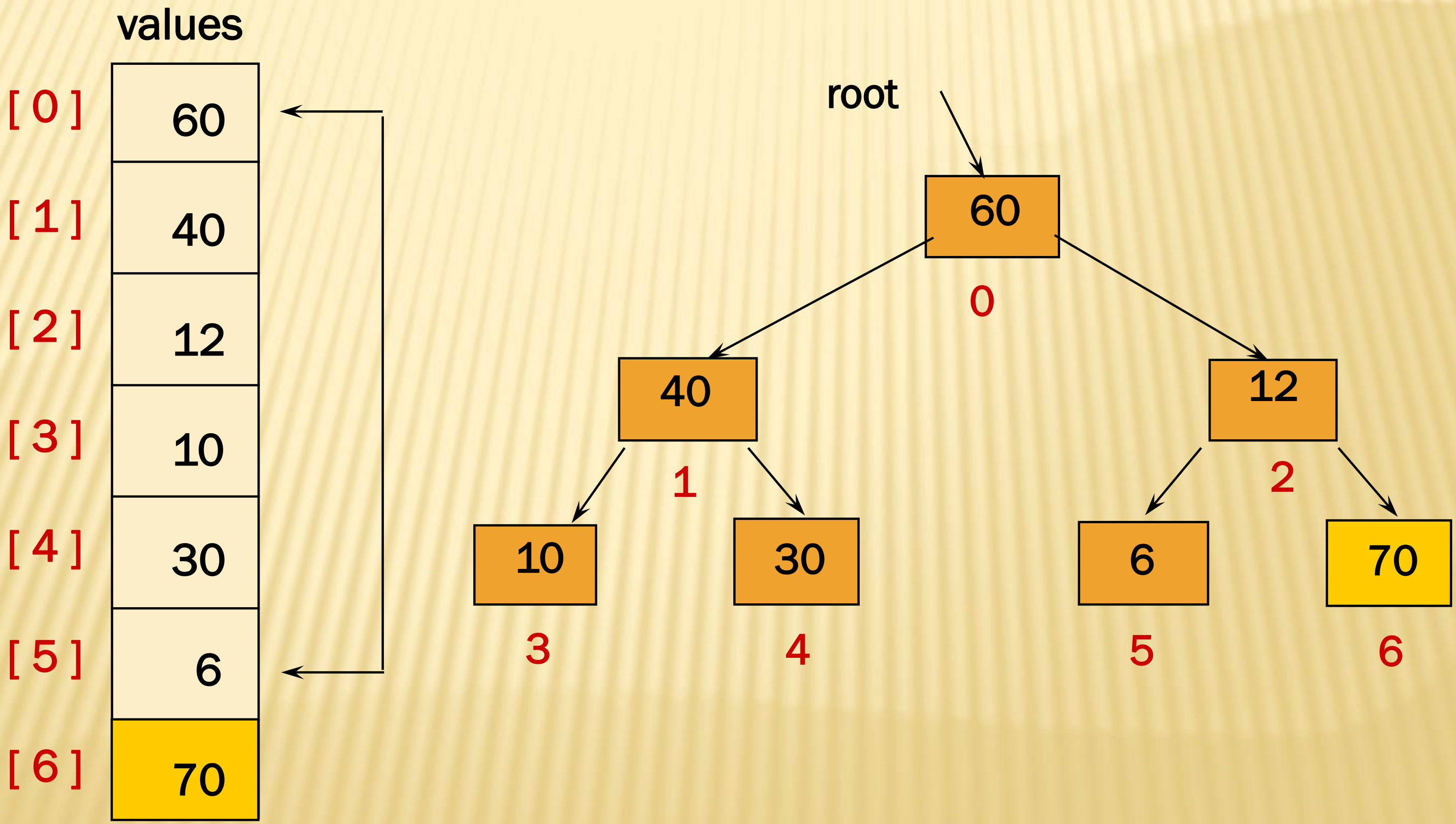
SORTING

After reheaping remaining unsorted elements



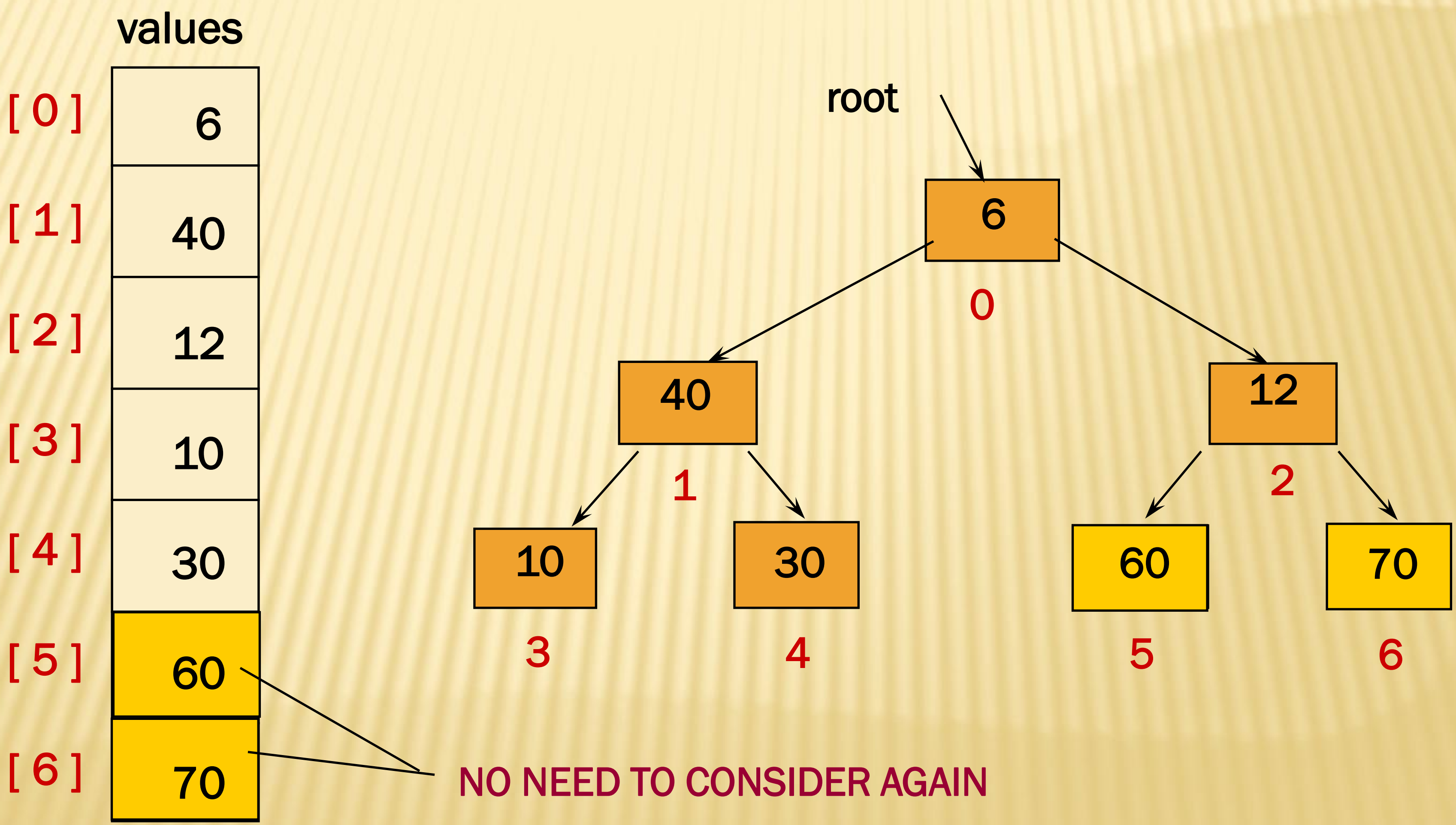
SORTING

Swap root element into last place in unsorted array



SORTING

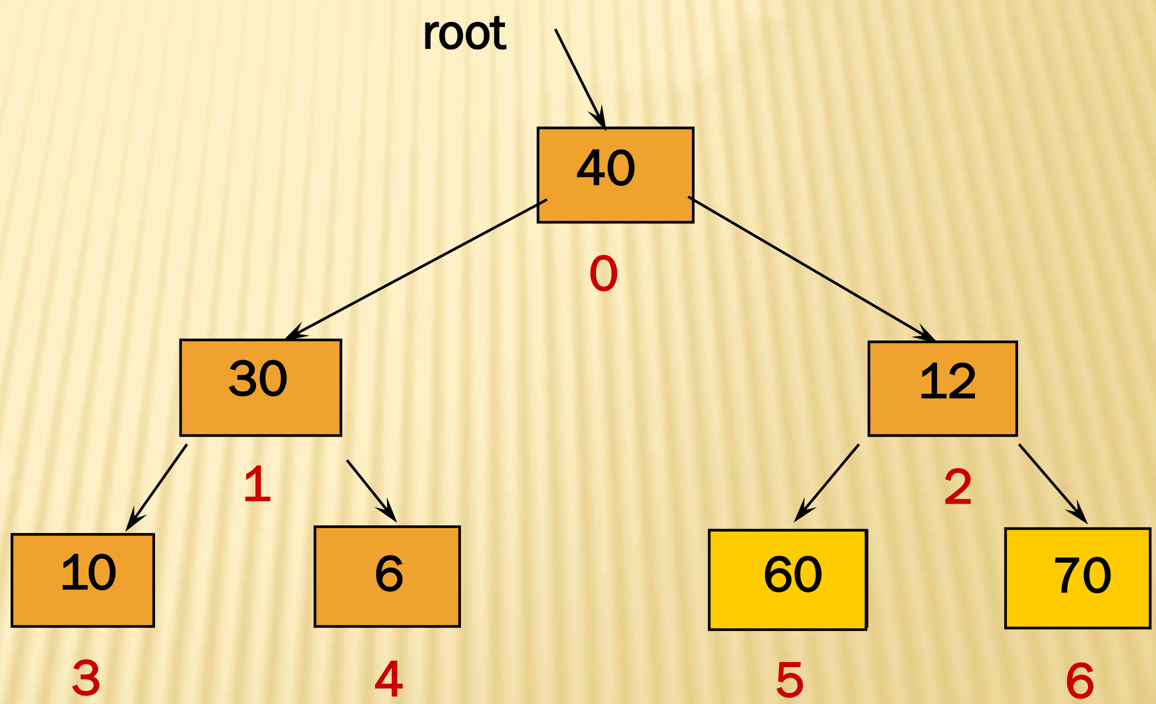
After swapping root element into its place



SORTING

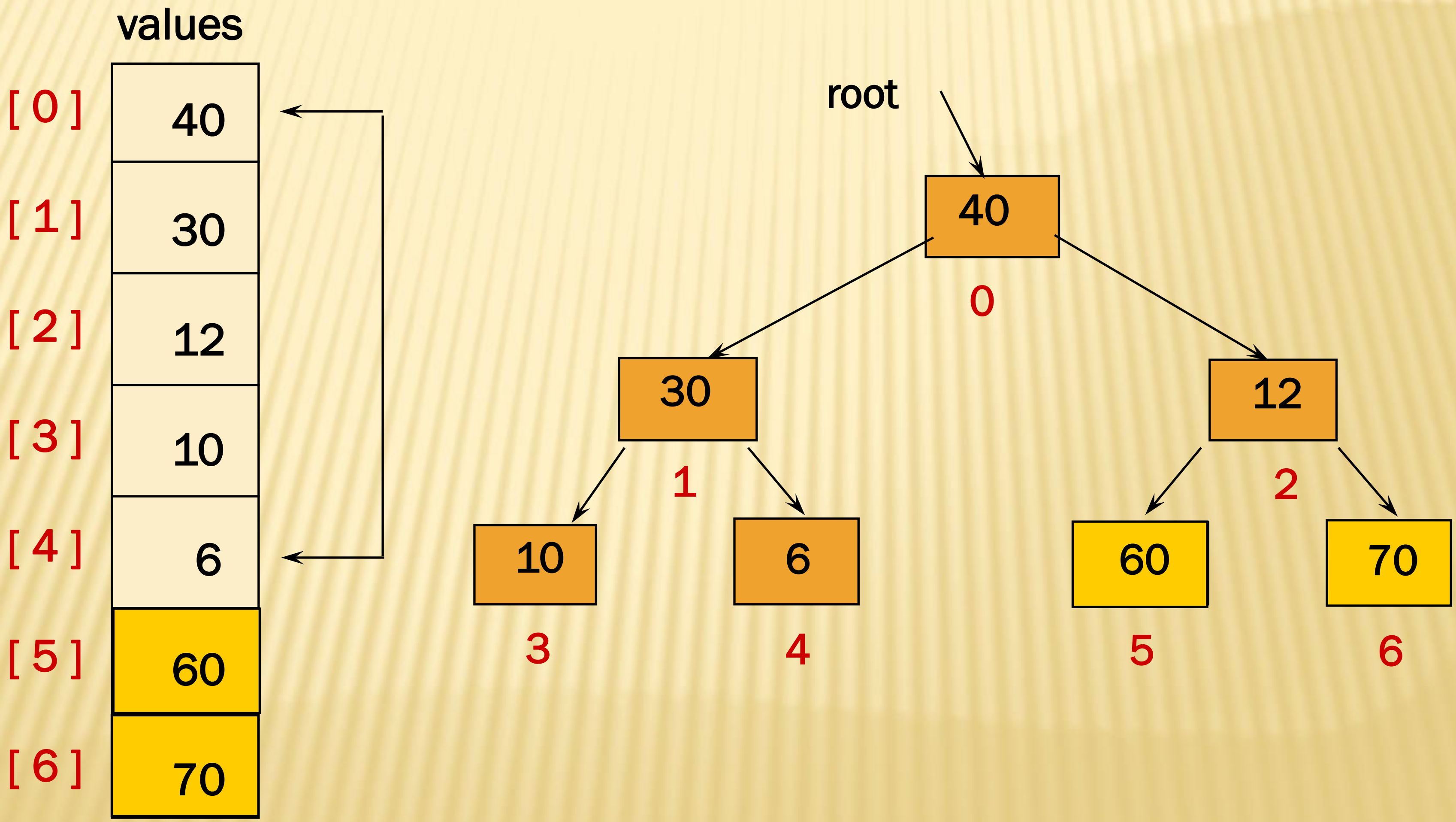
After reheaping remaining unsorted elements

	values
[0]	40
[1]	30
[2]	12
[3]	10
[4]	6
[5]	60
[6]	70



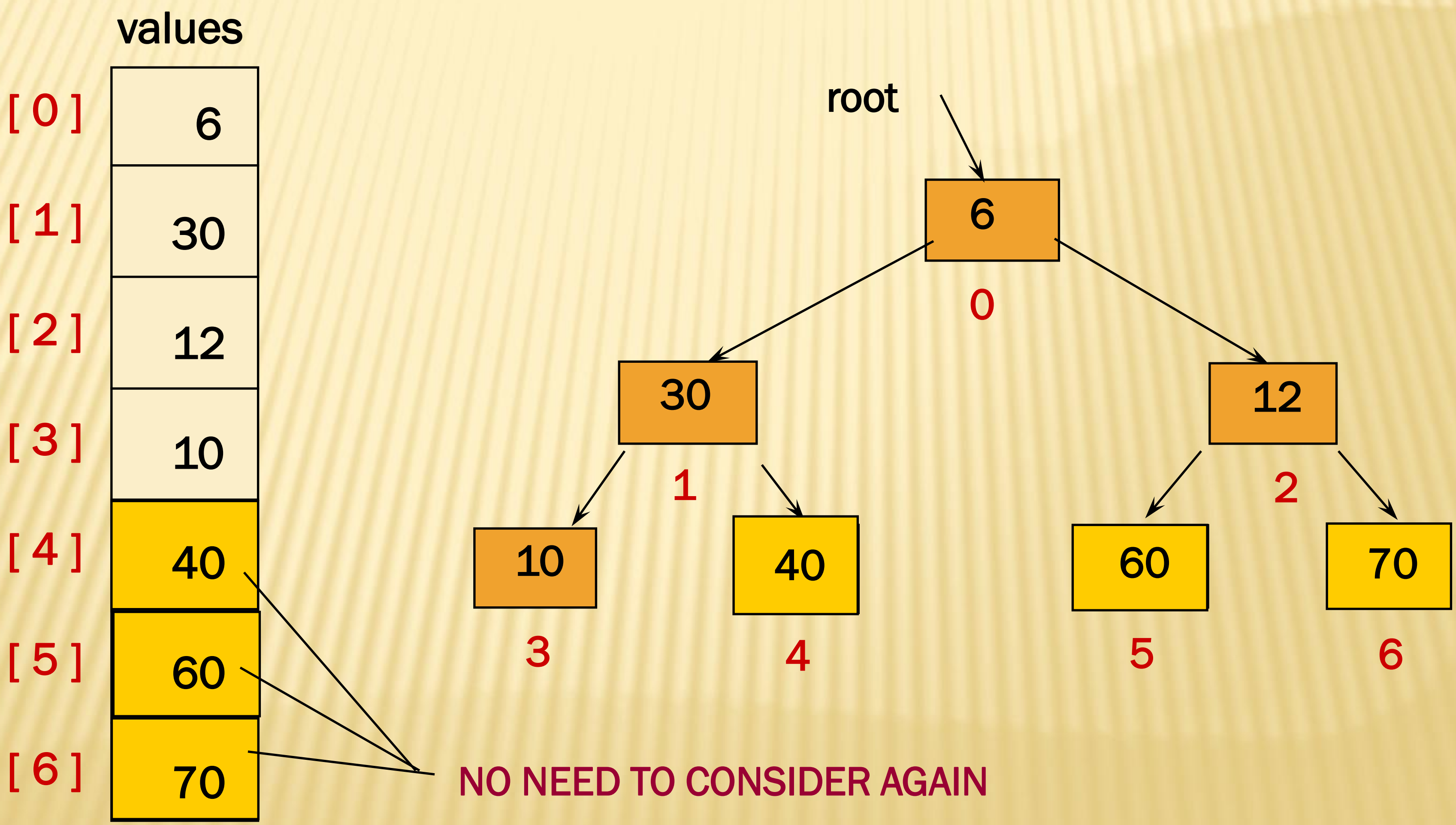
SORTING

Swap root element into last place in unsorted array



SORTING

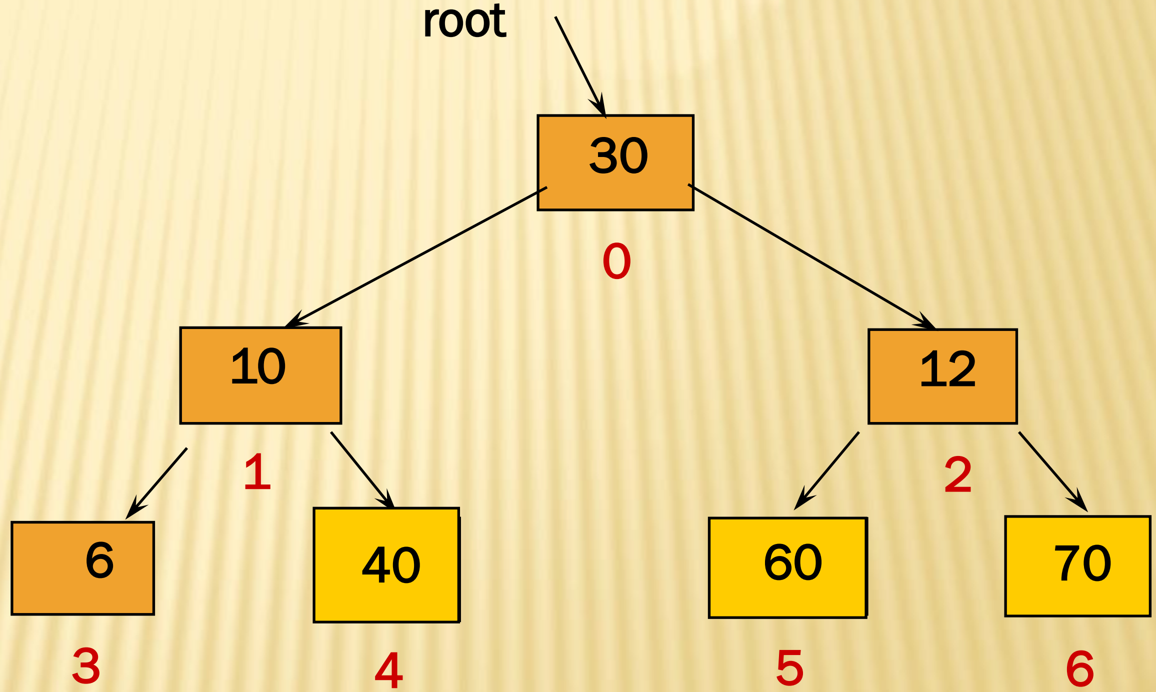
After swapping root element into its place



SORTING

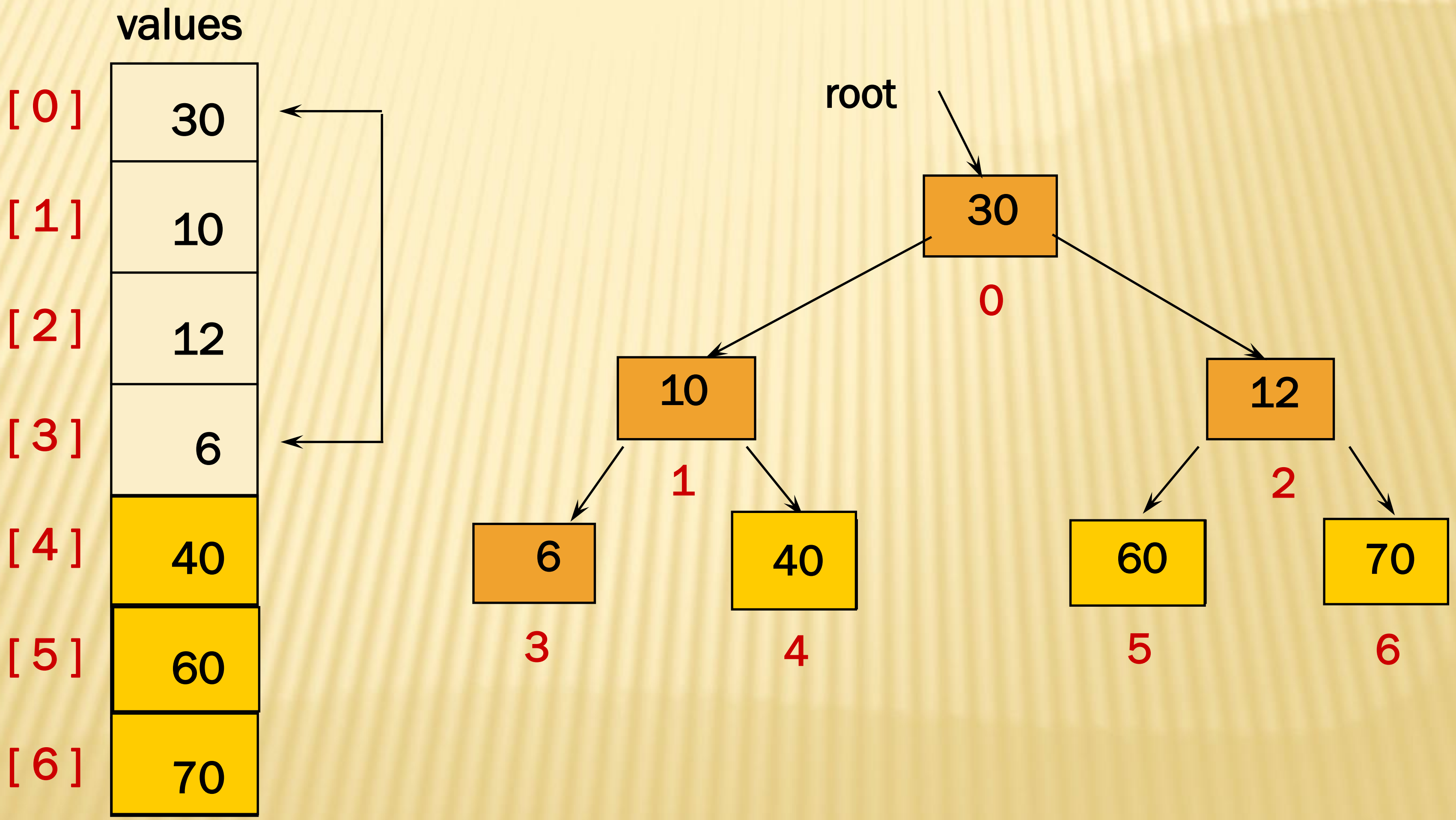
After reheaping remaining unsorted elements

	values
[0]	30
[1]	10
[2]	12
[3]	6
[4]	40
[5]	60
[6]	70



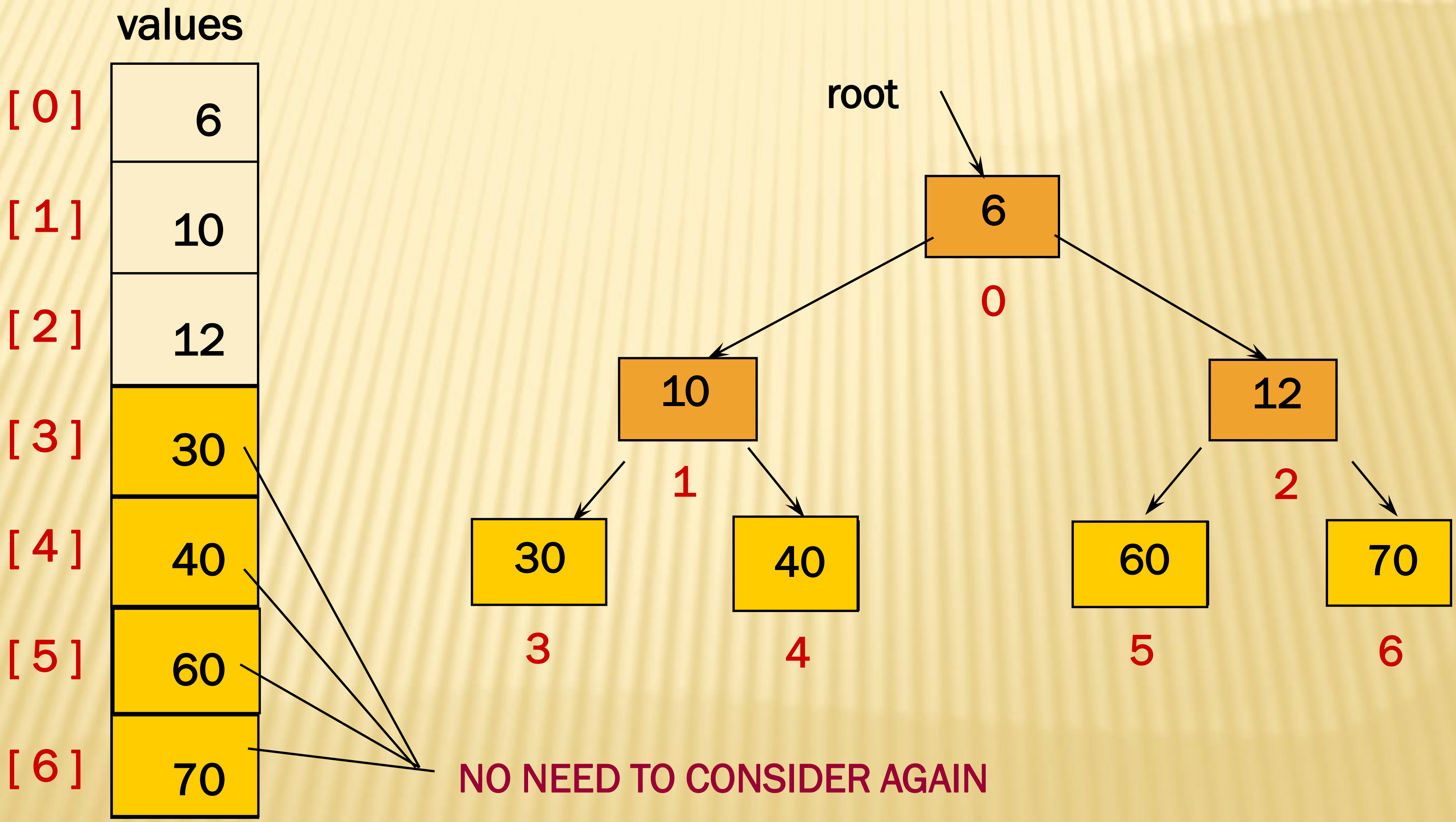
SORTING

Swap root element into last place in unsorted array



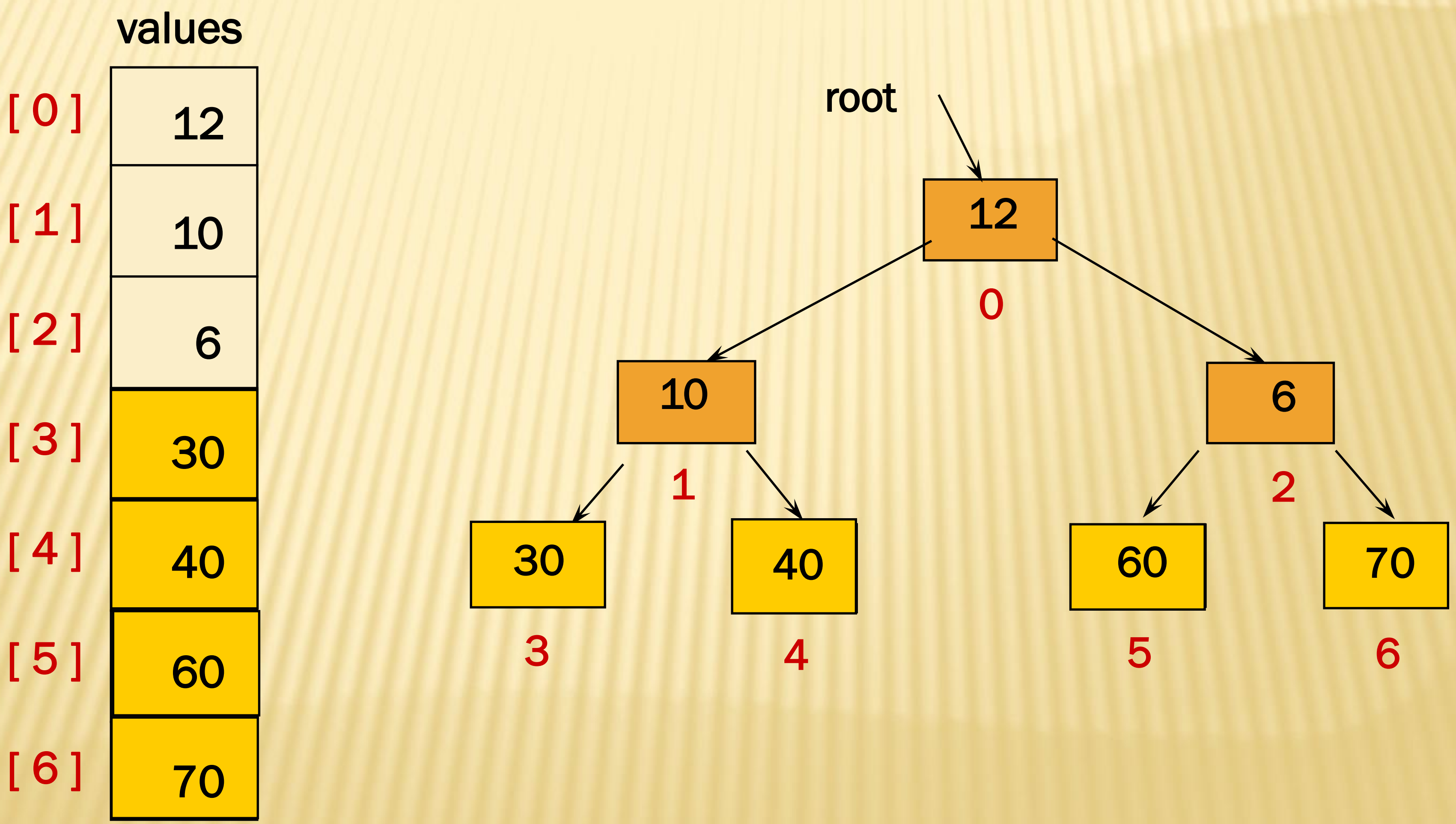
SORTING

After swapping root element into its place



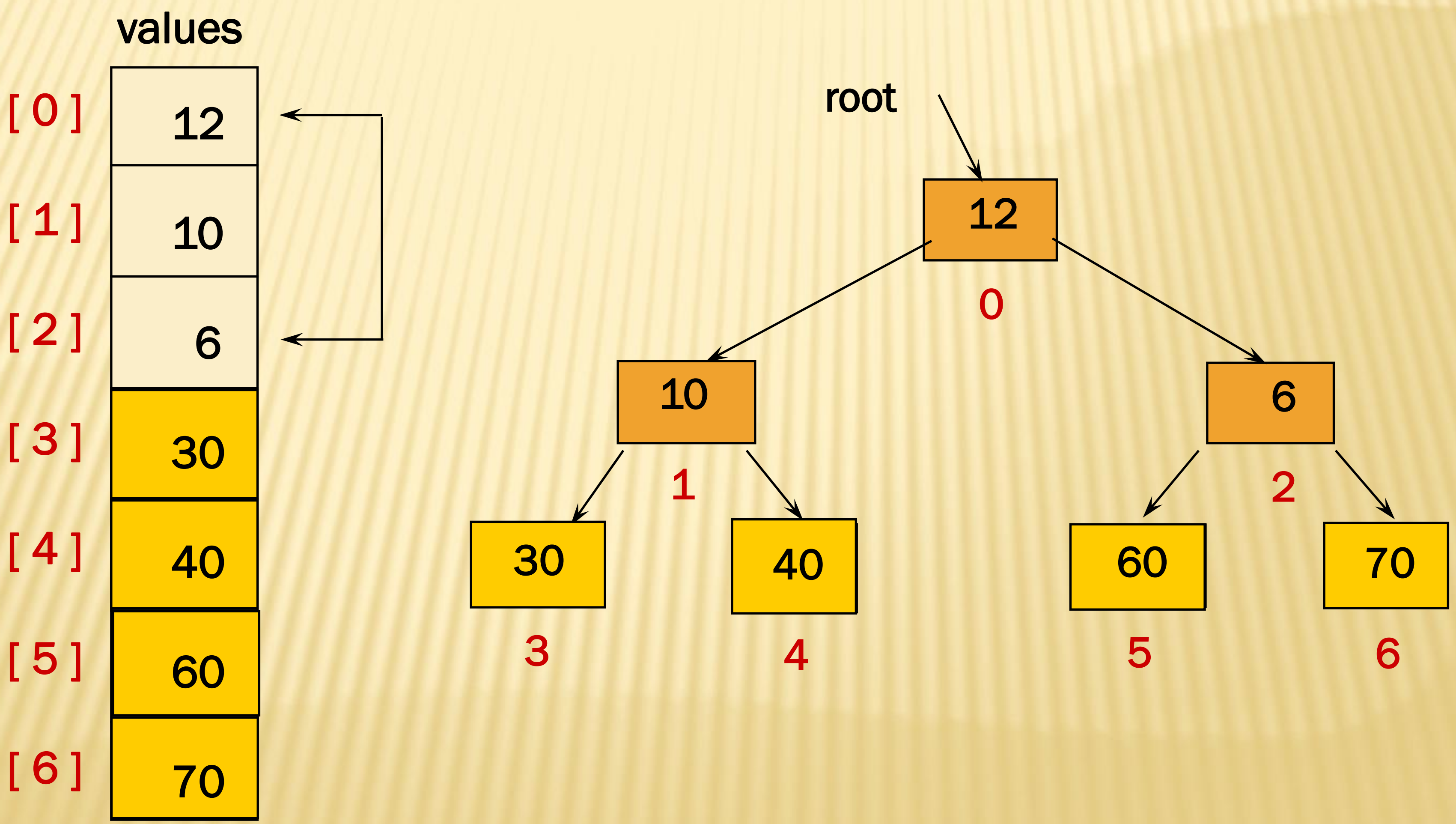
SORTING

After reheaping remaining unsorted elements



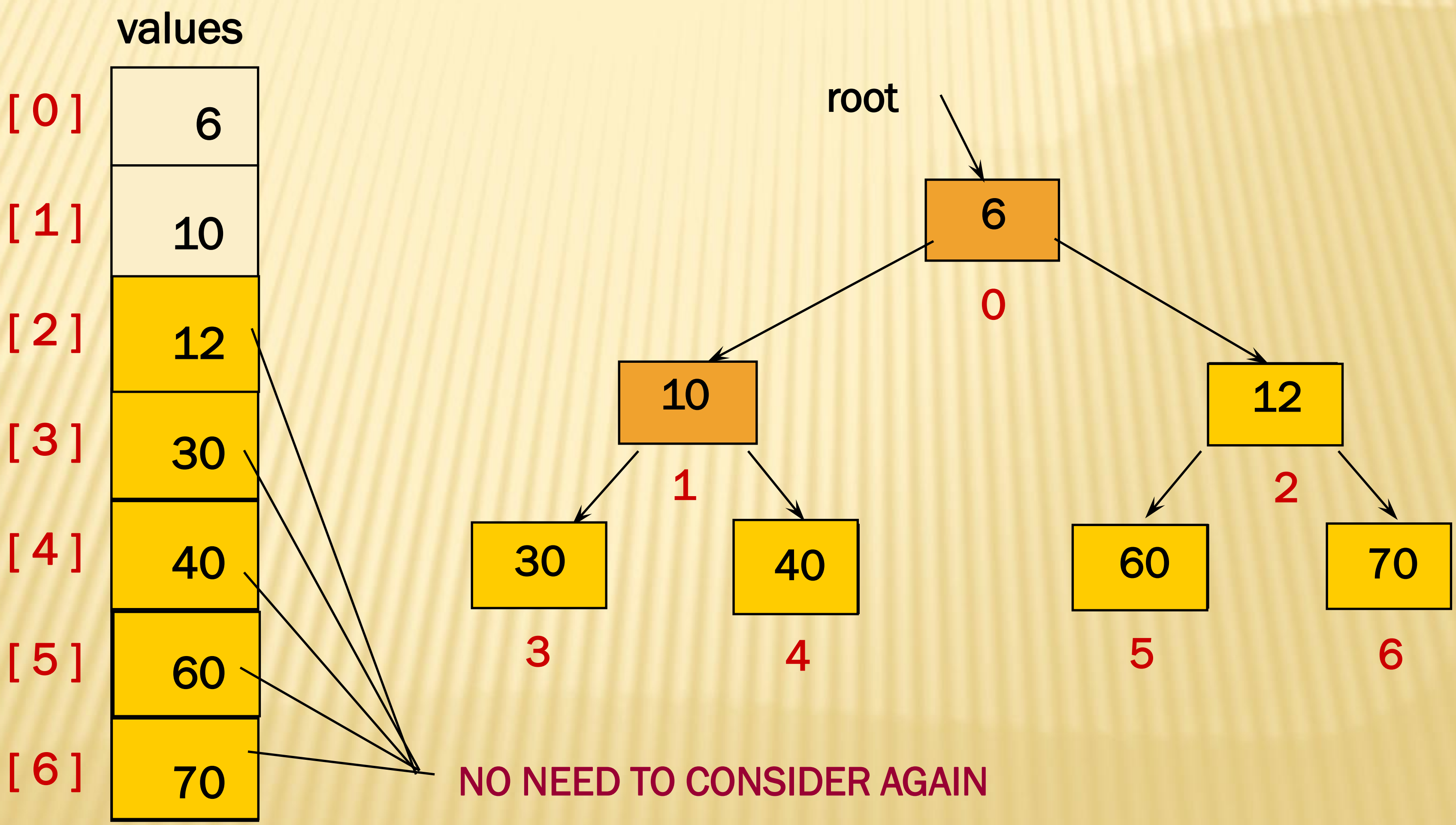
SORTING

Swap root element into last place in unsorted array



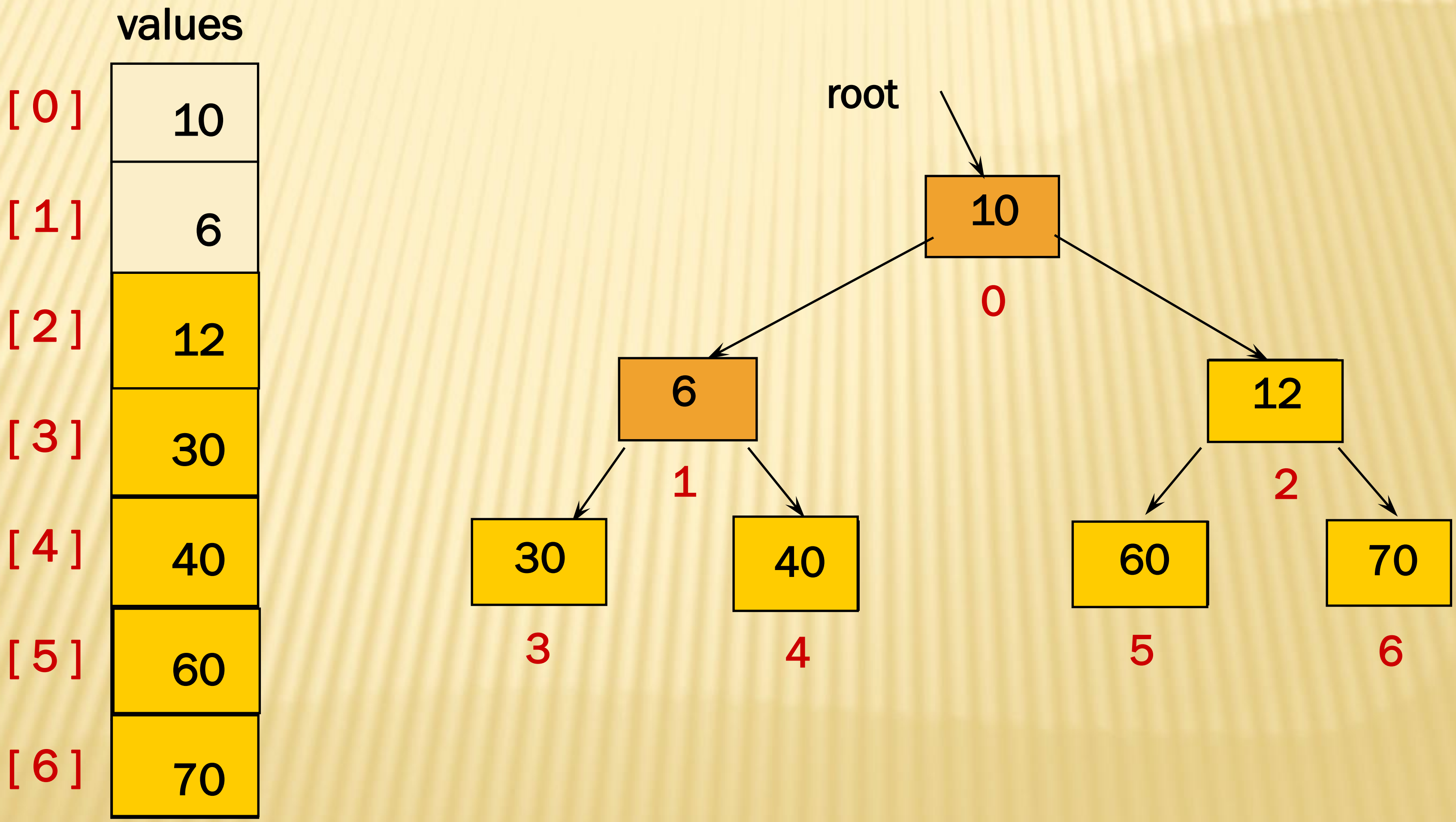
SORTING

After swapping root element into its place



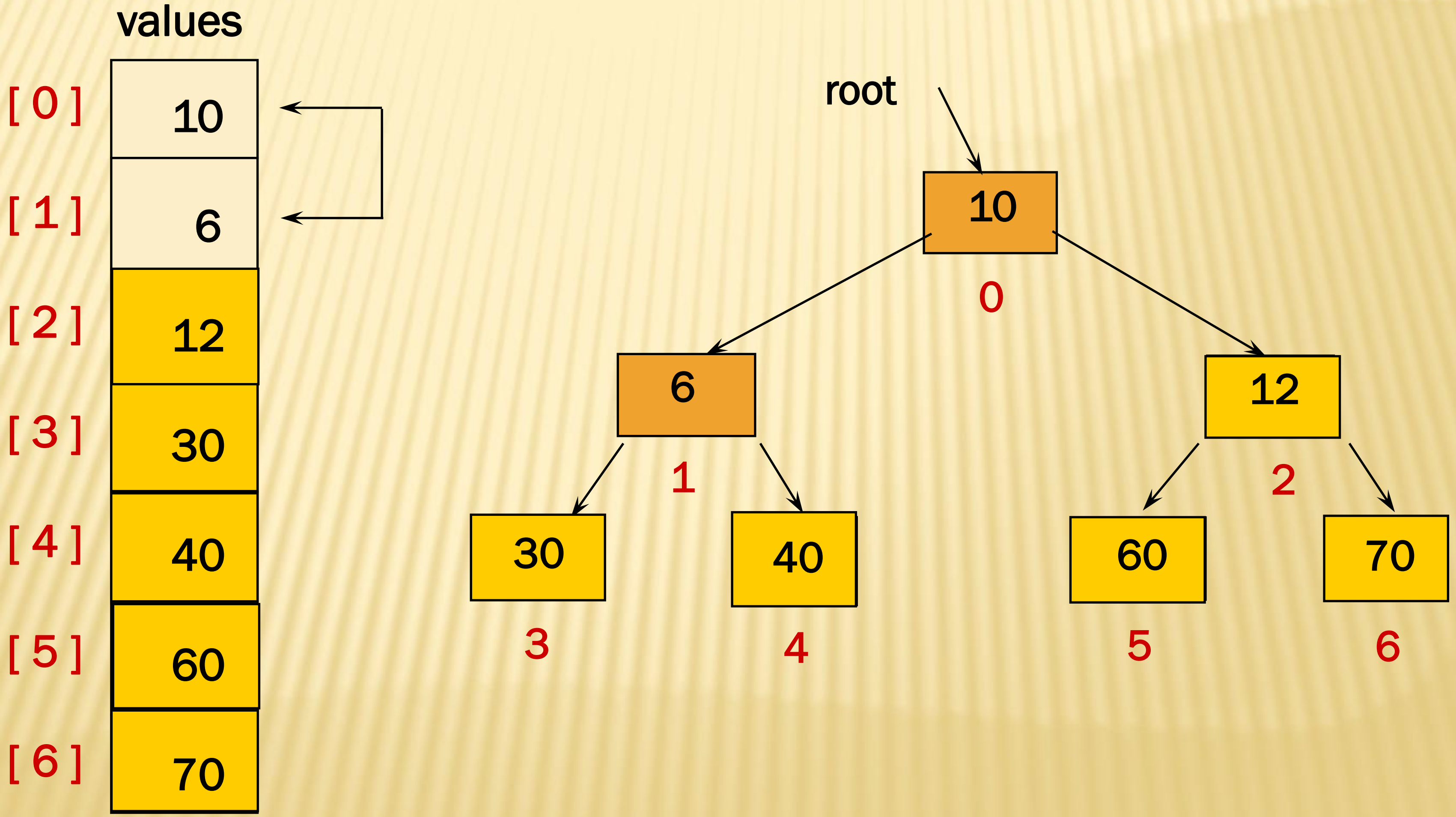
SORTING

After reheaping remaining unsorted elements



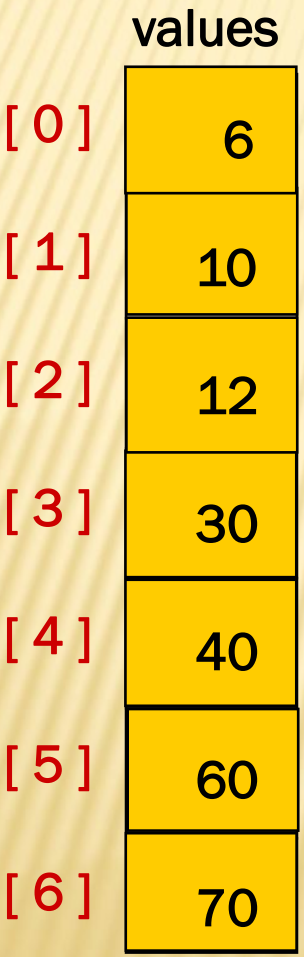
SORTING

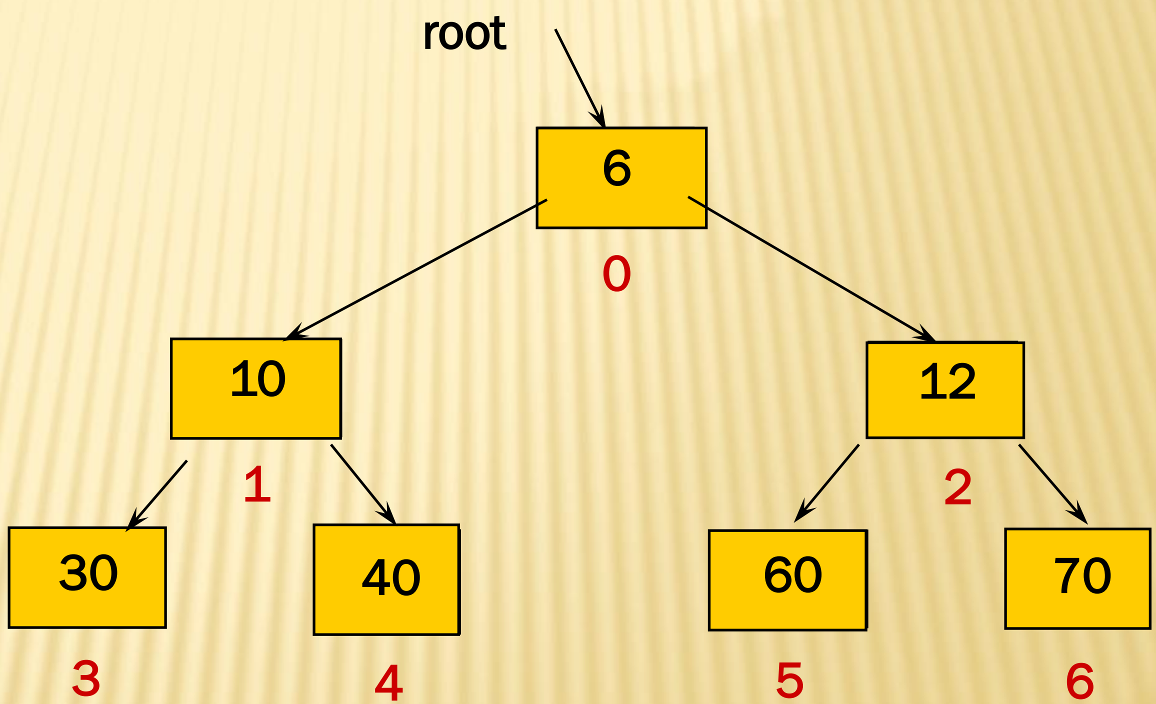
Swap root element into last place in unsorted array



SORTING

After swapping root element into its place





ALL ELEMENTS ARE SORTED

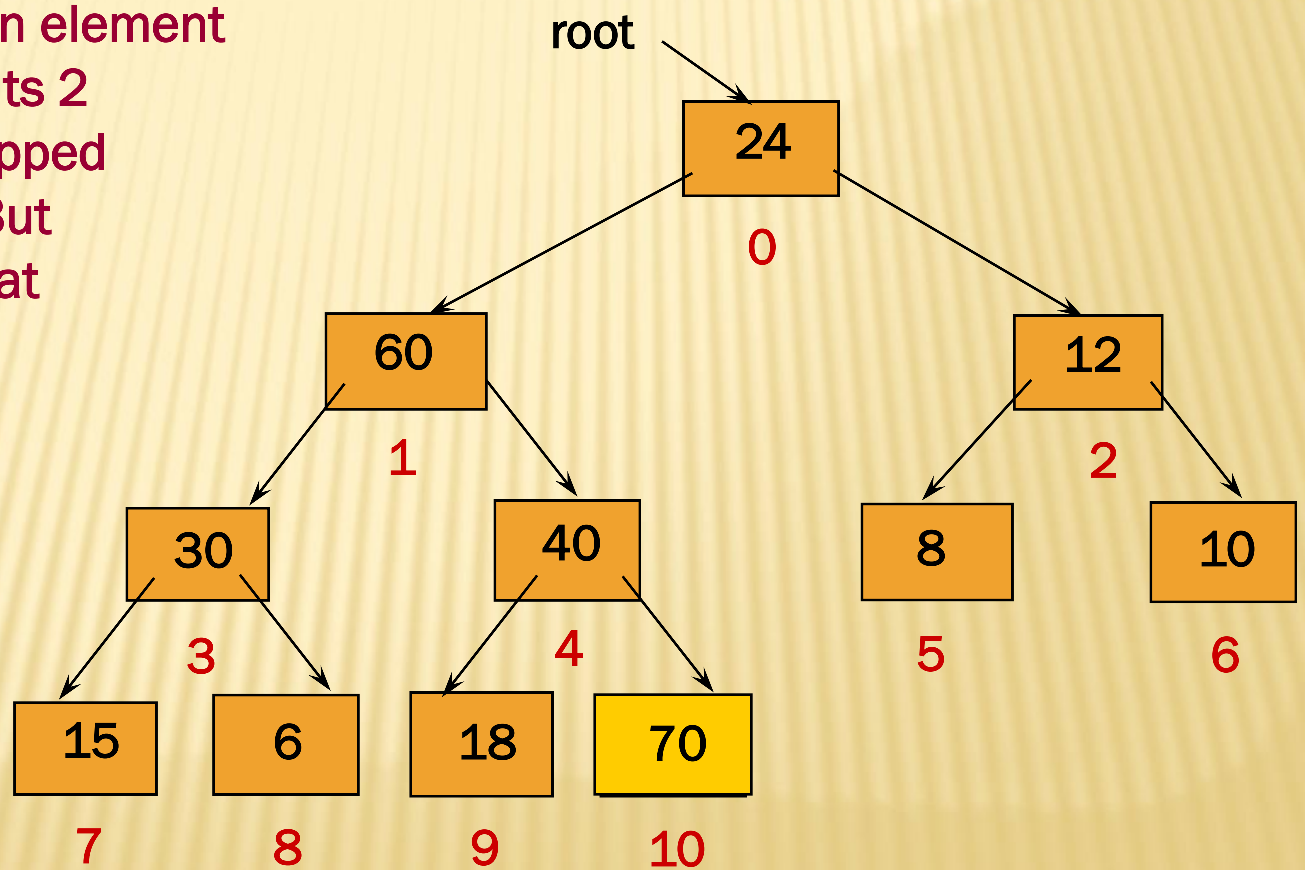
HEAP SORT

```
template <class  ItemType >
void  HeapSort  ( ItemType  values [ ] ,   int  numValues )
//  Post: Sorts array values[ 0 . . numValues-1 ] into ascending
//           order by key
{
    int  index ;

    // Convert array  values[ 0 . . numValues-1 ]  into a heap.
    for  ( index = numValues/2 - 1 ;  index >= 0 ;  index-- )
            ReheapDown ( values , index , numValues - 1 ) ;
    //  Sort the array.
    for  ( index = numValues - 1 ;  index >= 1 ;  index-- )
    {
            Swap ( values [0] , values [index] );
            ReheapDown ( values , 0 , index - 1 ) ;
    }
}
```

HEAP SORT: HOW MANY COMPARISONS?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only $O(\log_2 N)$ levels.



HEAP SORT OF N ELEMENTS: HOW MANY COMPARISONS?

(N/2) * O(log N) compares to create original heap

+

(N-1) * O(log N) compares for the sorting loop

= O (N * log N) compares total