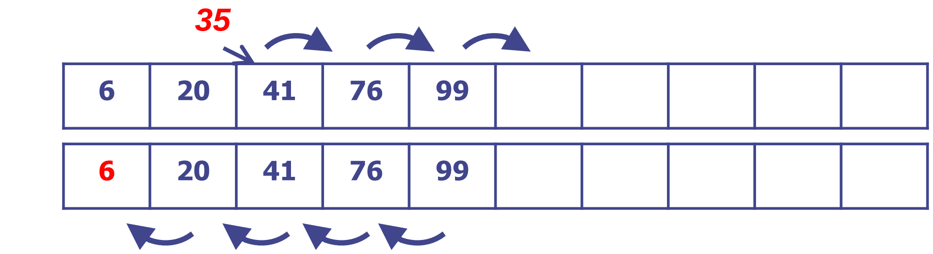
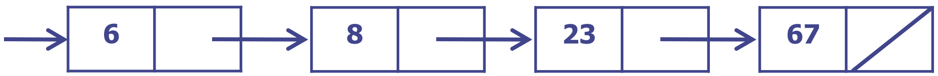
Lecture 7: Heaps

Why Heaps ?

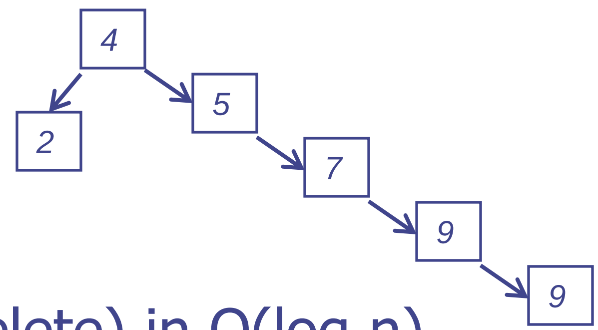
- Remember from Algorithms and Data Structure course:
  - Array
    - Linear search: O(n)
    - Insertion, deletion: O(n)



  - Linked list:
    - Search: O(n)
    - Insertion, deletion: O(n)



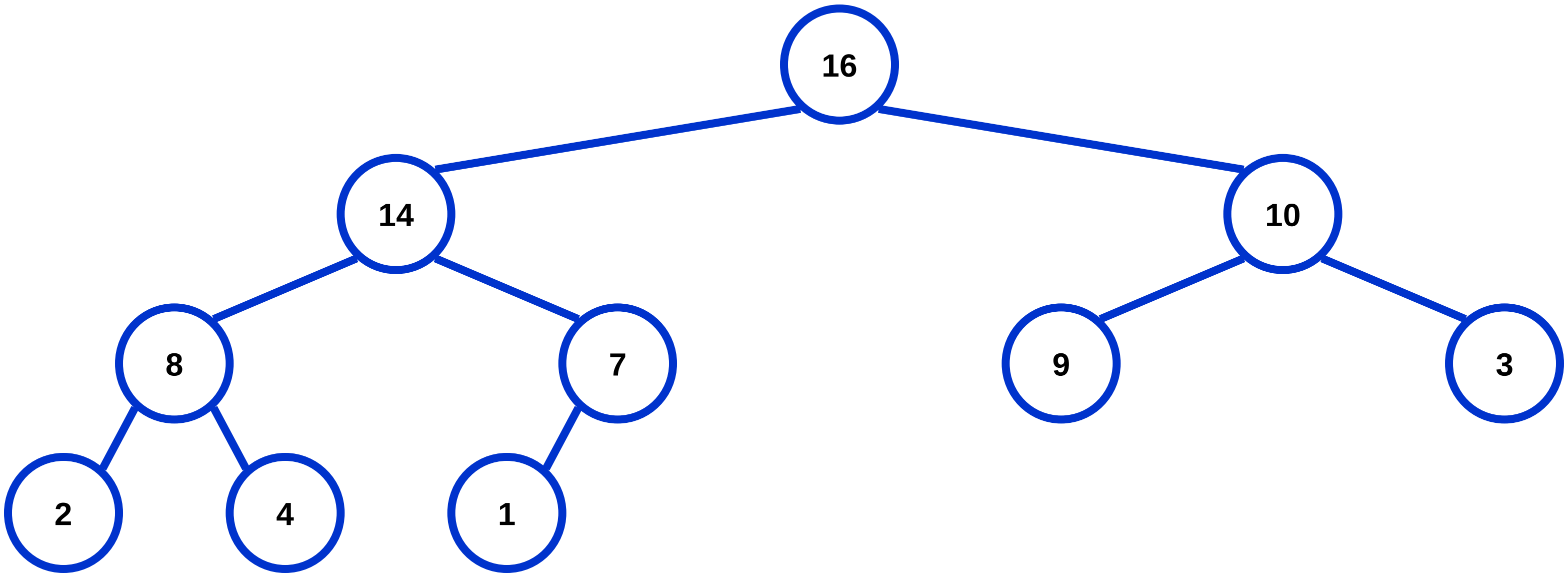
  - Binary search in Array:
    - Search: O(log n)
    - Insert, delete in Array: O(n)
  - Binary search tree
    - Height may be O(n)
    - So, search, insert, delete: O(n)



- Can we do everything (search, insert, delete) in O(log n) time?
  - Yes, by heaps (this lecture), different Height-balanced search trees
  - by Skip list (next lecture)

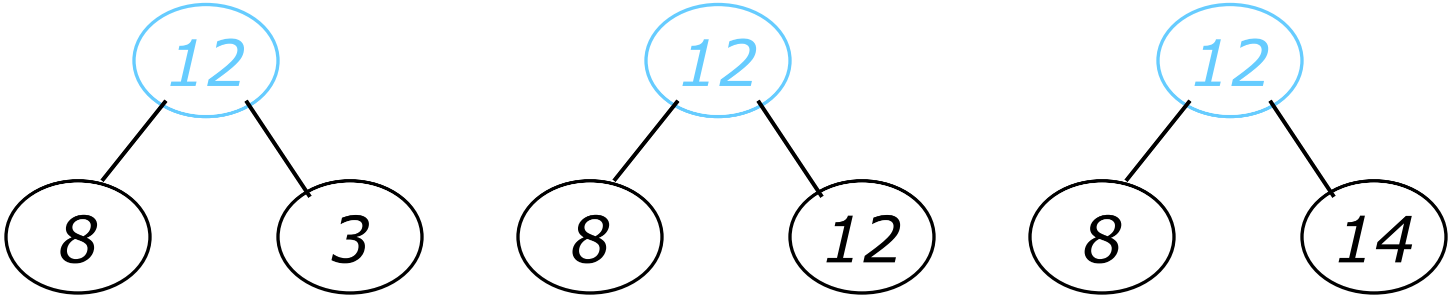
Heaps

- A *heap* can be seen as a complete binary tree (complete means: last level may be partially complete from left to right.)



The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children

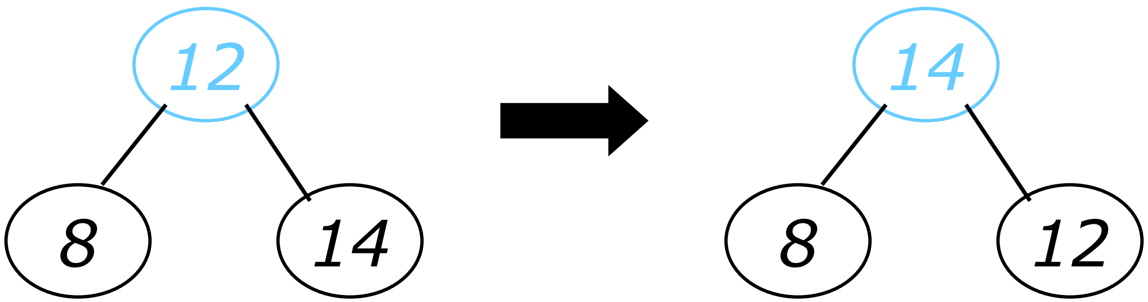


Blue node has heap property *Blue node has heap property* *Blue node does not have heap property*

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting up
- Notice that the child may have *lost* the heap property

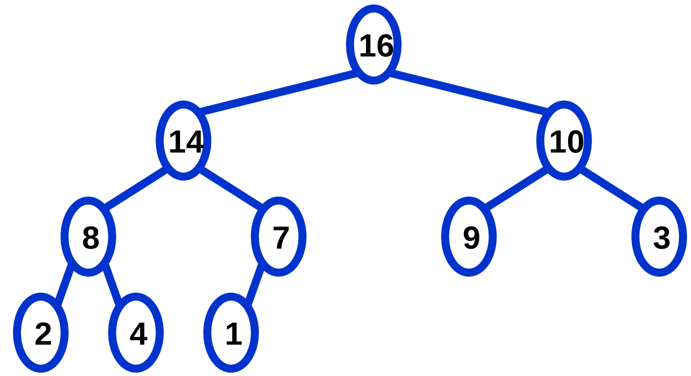
Heaps

- In practice, heaps are usually implemented as arrays (array may not be sorted):

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

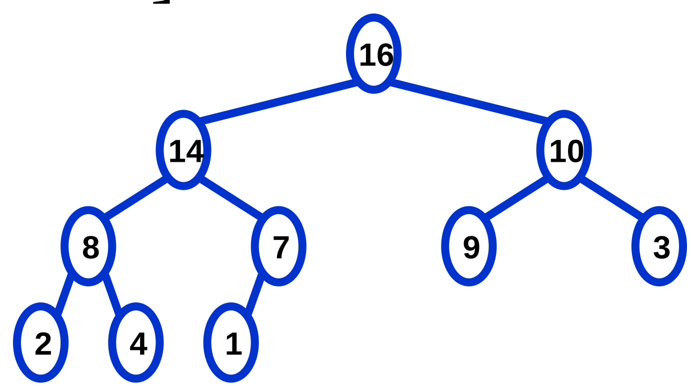
=



Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A = | *16* | *14* | *10* | *8* | *7* | *9* | *3* | *2* | *4* | *1* | =



Referencing Heap Elements

- So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Heap Property, Height

- **Heap property:**
  - ***Parent*** **≥ left and right,** for all nodes
  - *Where is the largest element in a heap stored?*
    - **Answer:** in the root.

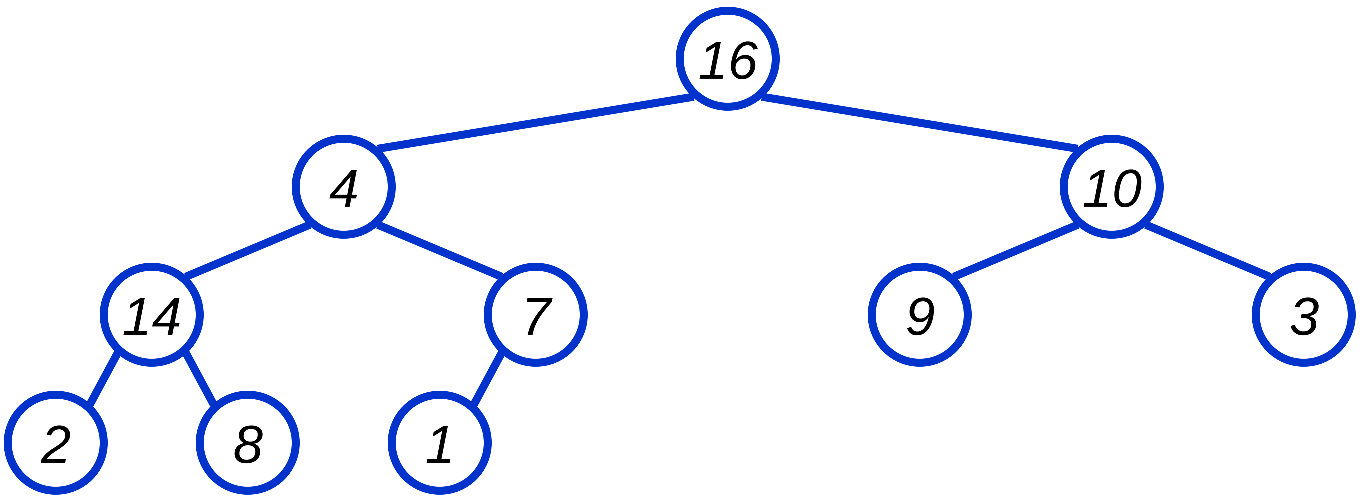
***** Important***:** This is Max heap. We shall see Min heap at the end.

- **Heap Height:** *What is the height of an n-element heap? Why?*
  - **Answer:** O(log n), because complete binary tree.

Heap Operations: Heapify() (*** Very Important***)

- **`Heapify()`**: *maintain the heap property*
  - If a node violates the heap property, then parent node “goes down” as long as required

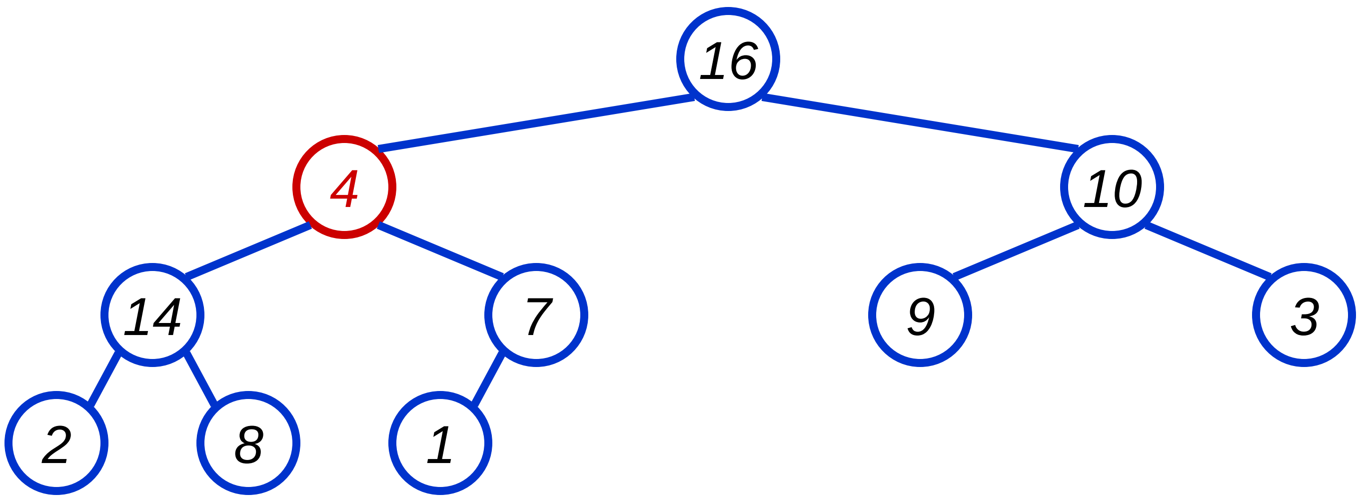
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

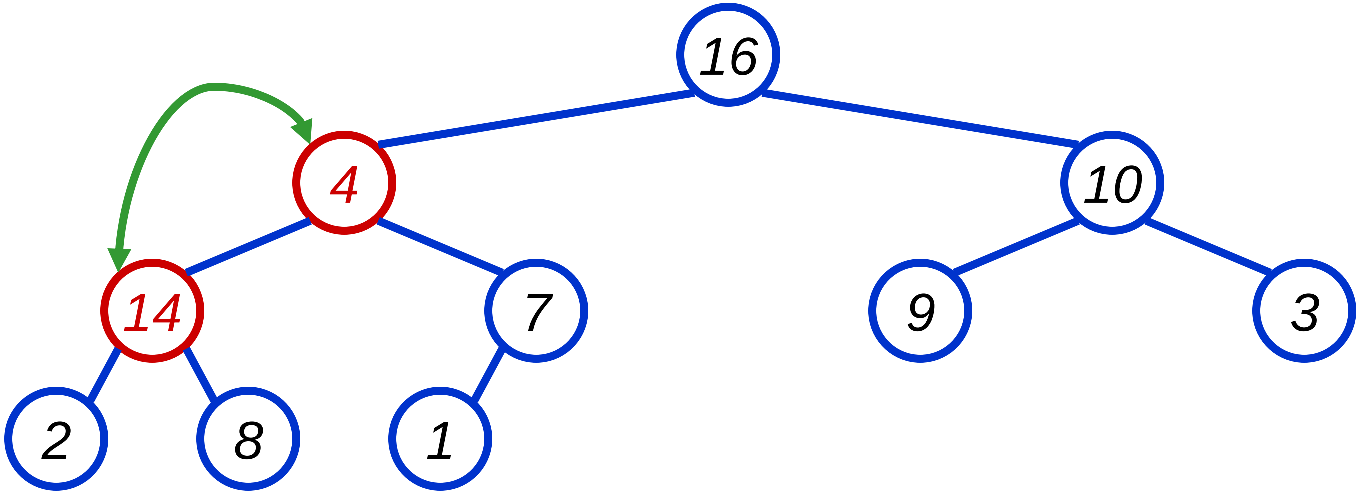
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

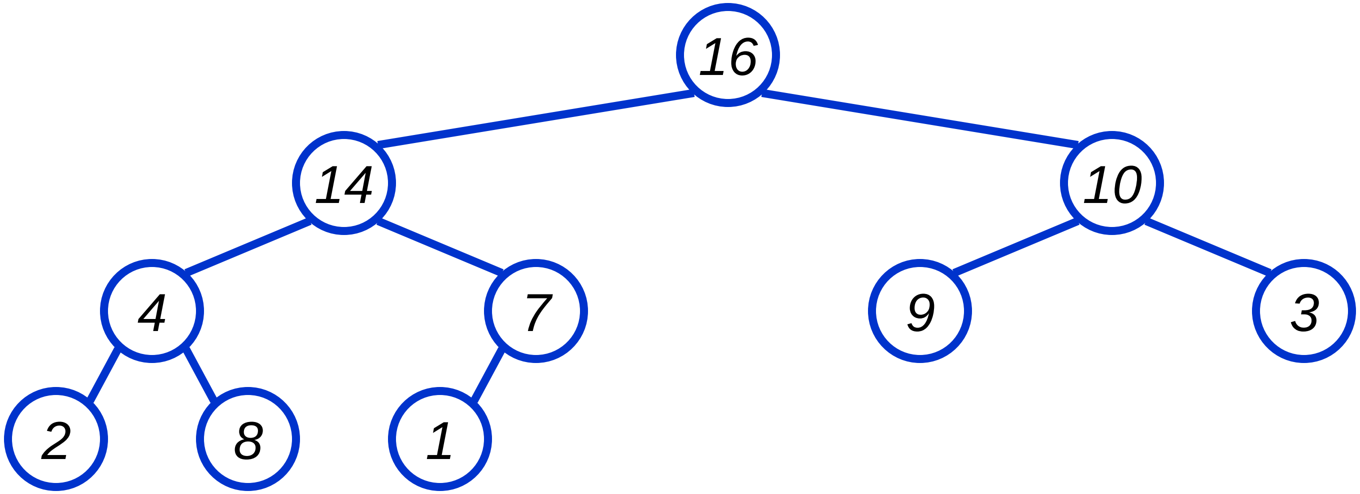
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

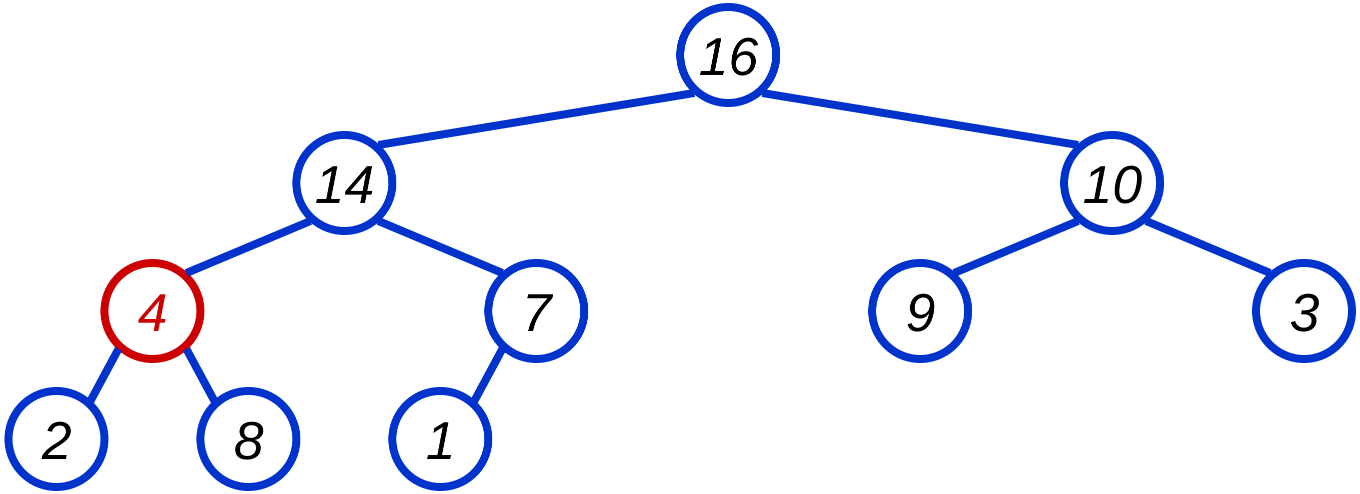
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

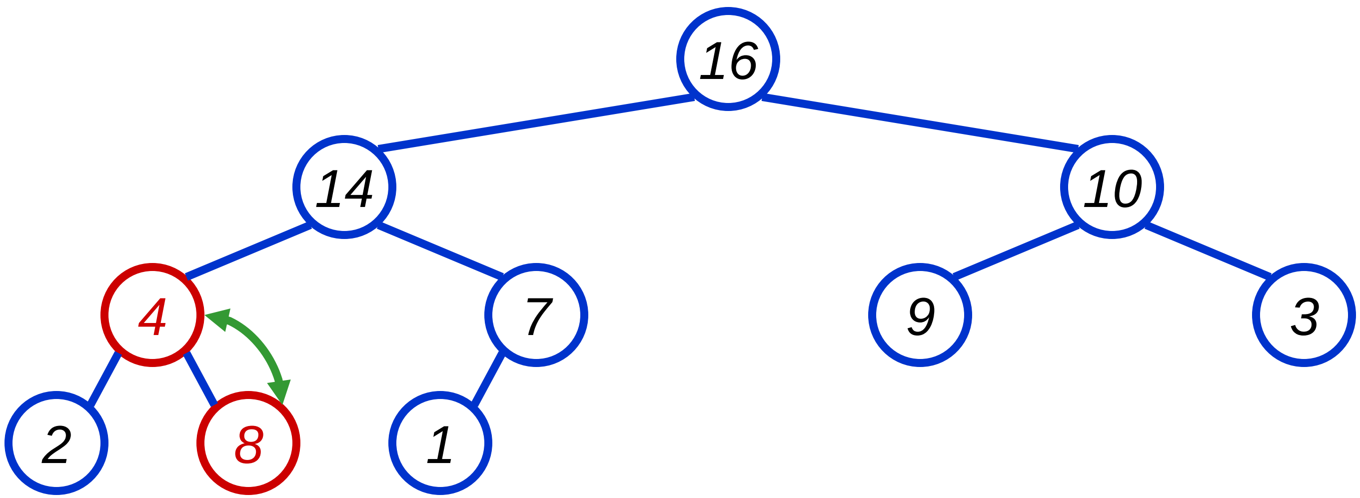
Heapify() Example



A =

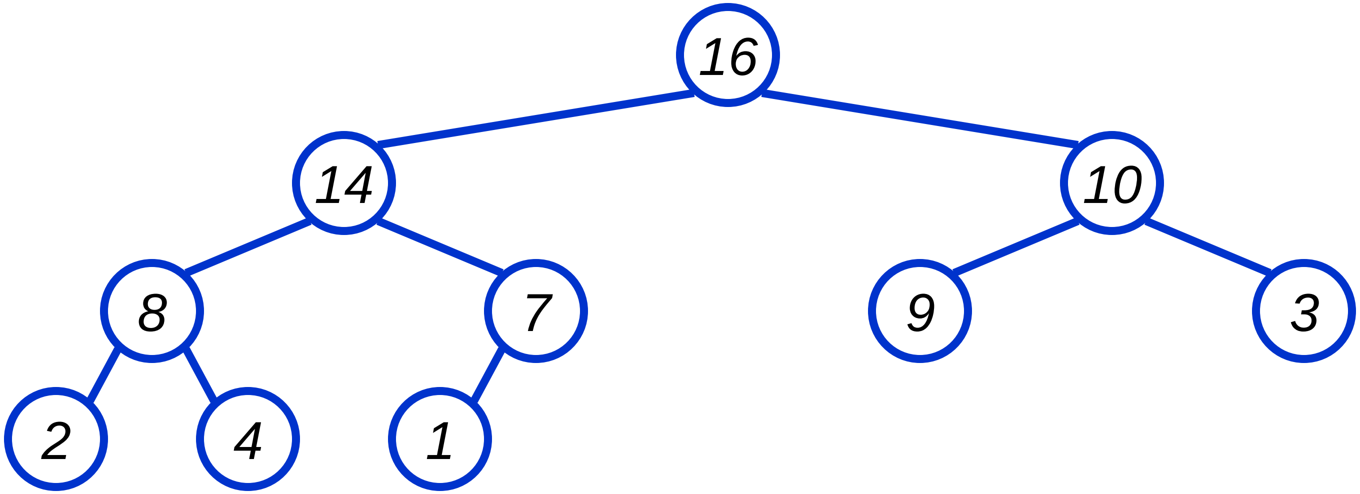
16	*14*	*10*	*4*	*7*	*9*	*3*	*2*	*8*	*1*

Heapify() Example



A =	16	14	10	4	7	9	3	2	8	1

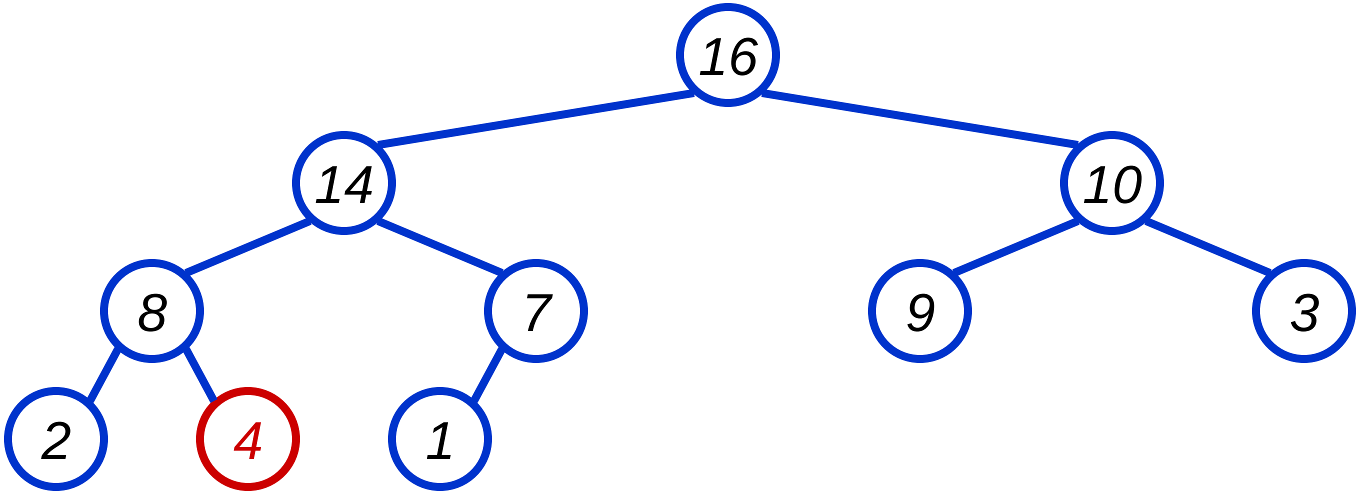
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

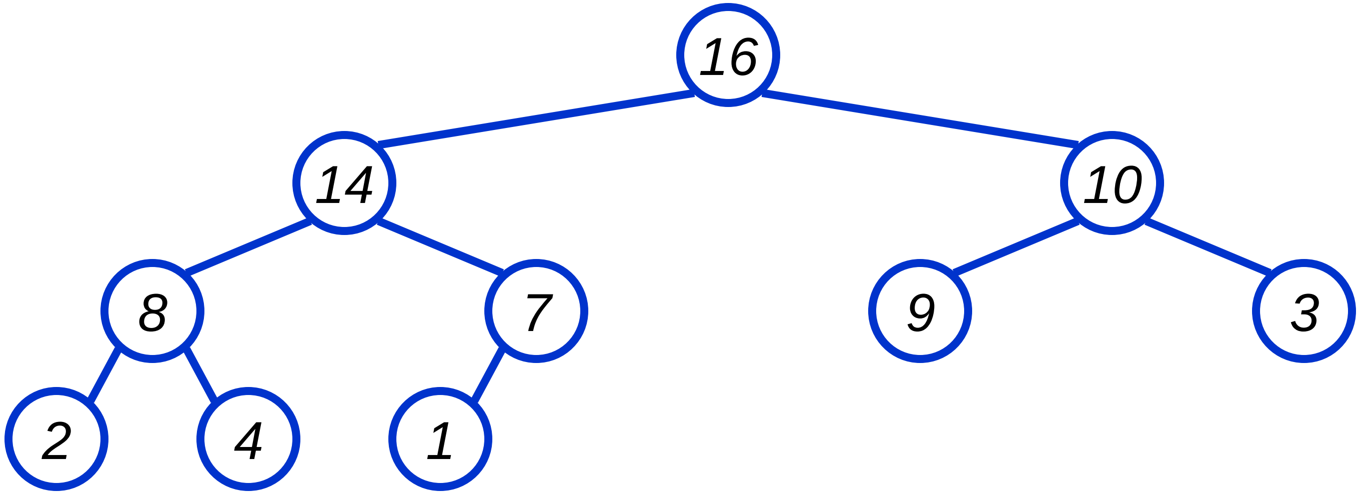
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heap Operations: Heapify()

```
Heapify(A, i)
{
  if (l <= heap_size(A) && A[l] > A[i])
      largest = l;
  else largest = i;
  if (r <= heap_size(A) && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      Heapify(A, largest);
}
```

Running time of Heapify()

- O(log n)

```
Heapify(A, i)
{
   if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
   else  largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
        largest = r;
   if (largest != i)
        Swap(A, i, largest);
        Heapify(A, largest);
}
```

Total: O(1)

How many times? Height = O(log n)

Total = O(1)*O(log n) = O(log n)

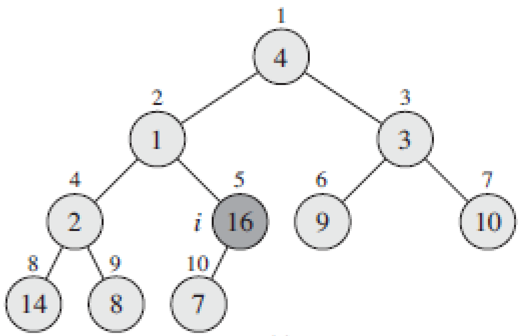
Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - For array of length *n*, all elements in range A[n/2 + 1 .. n] are already heaps, because they have no childer.
  - So, walk backwards through the array from the remaining nodes n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

BuildHeap() Example

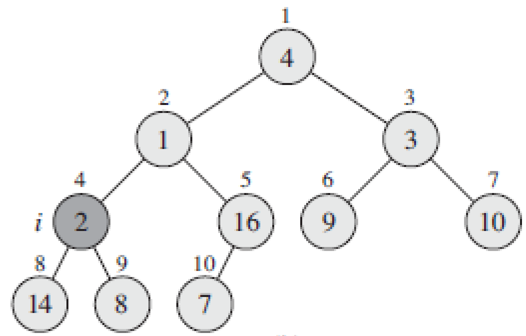
Starting array:

4	1	3	2	16	9	10	14	8	7



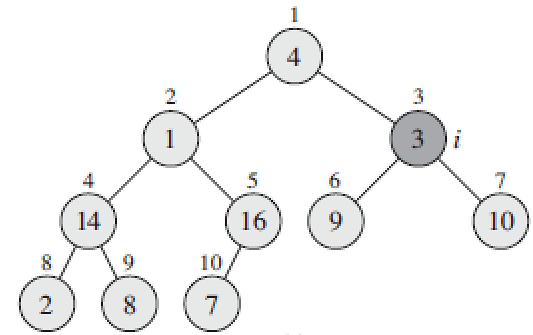
(a)

4	1	3	2	16	9	10	14	8	7



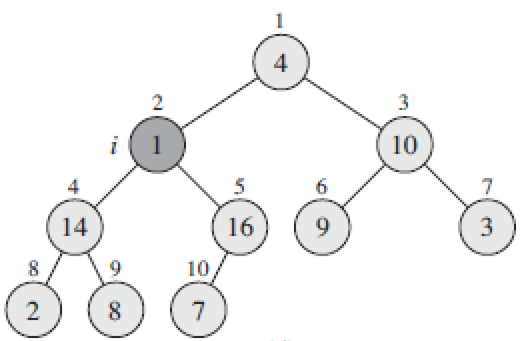
(b)

4	1	3	14	16	9	10	2	8	7



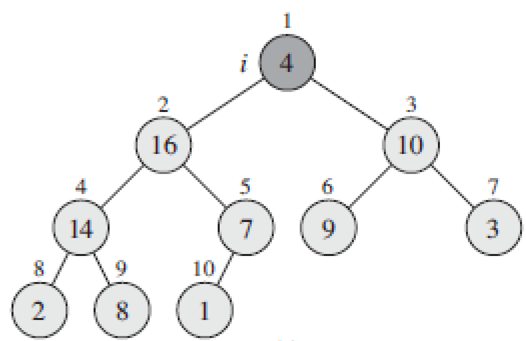
(c)

4	1	10	14	16	9	3	2	8	7



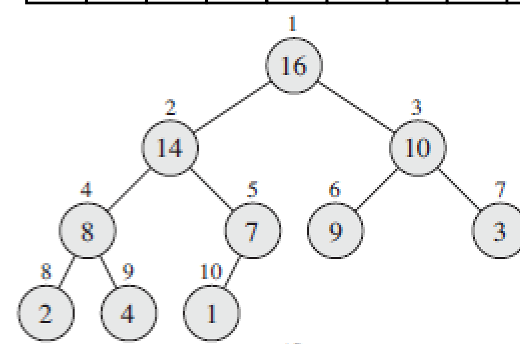
(d)

4	16	10	14	7	9	3	2	8	1



(e)

16	14	10	8	7	9	3	2	4	1



(f)

BuildHeap(): Pseudo code

```
BuildHeap(A)
{
  heap_size = n;
  for (i = n/2; i >= 1; i--)
      Heapify(A, i);
}
```

BuildHeap(): Running time

```
BuildHeap(A)
{
  heap_size = n;
  for (i = n/2; i >= 1; i--)
      Heapify(A, i);
}
```

n/2 times

O(log n)

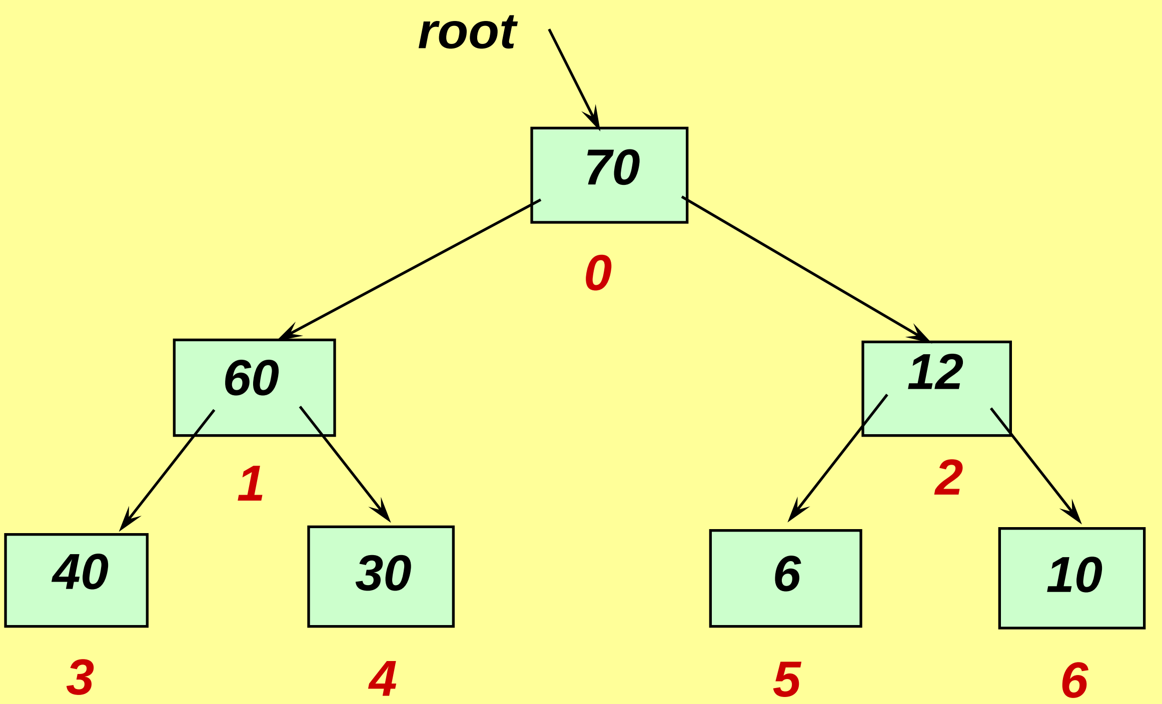
Total time: n/2*O(log n) = O(n log n)

Heap Sort

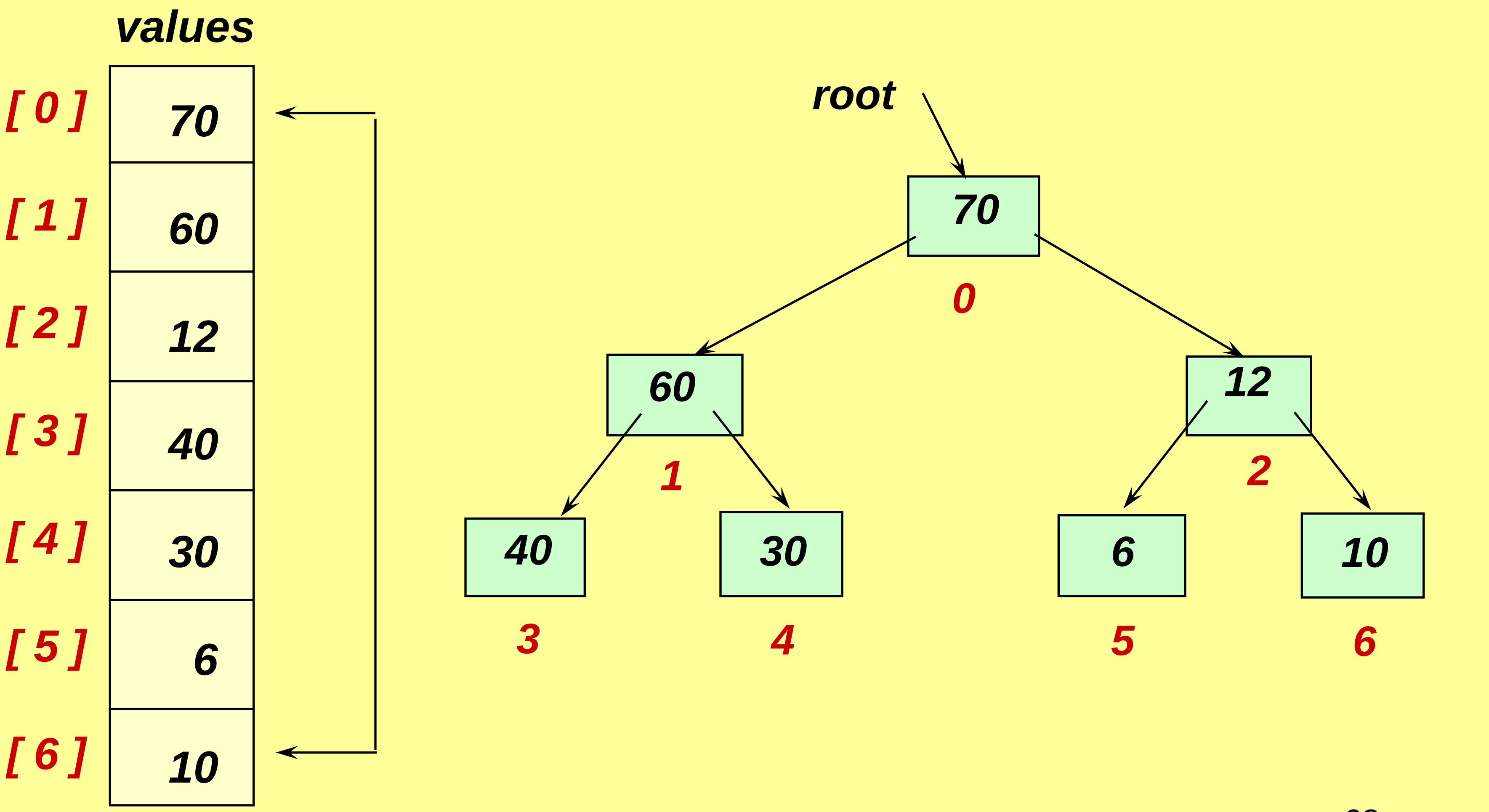
- **<u>Step 1:</u>** make the unsorted array into a heap by **`BuildHeap()`** function.
- **<u>Step 2:</u>** Swap the root (maximum) with the last unsorted element.
- **<u>Step3:</u>** Reheap by **`Heapify()`** function.

After BuildHeap()

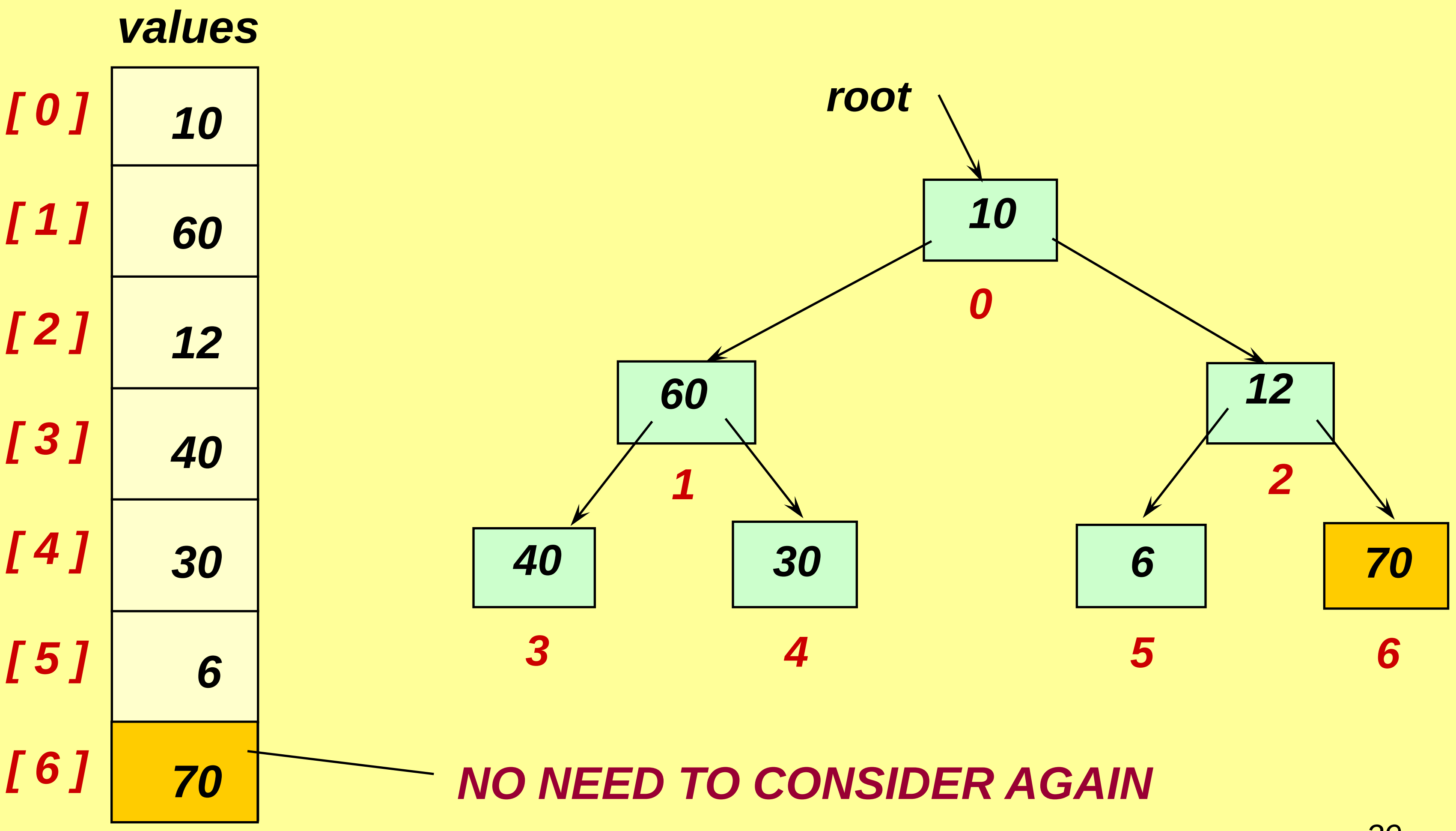
	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	6
[6]	10



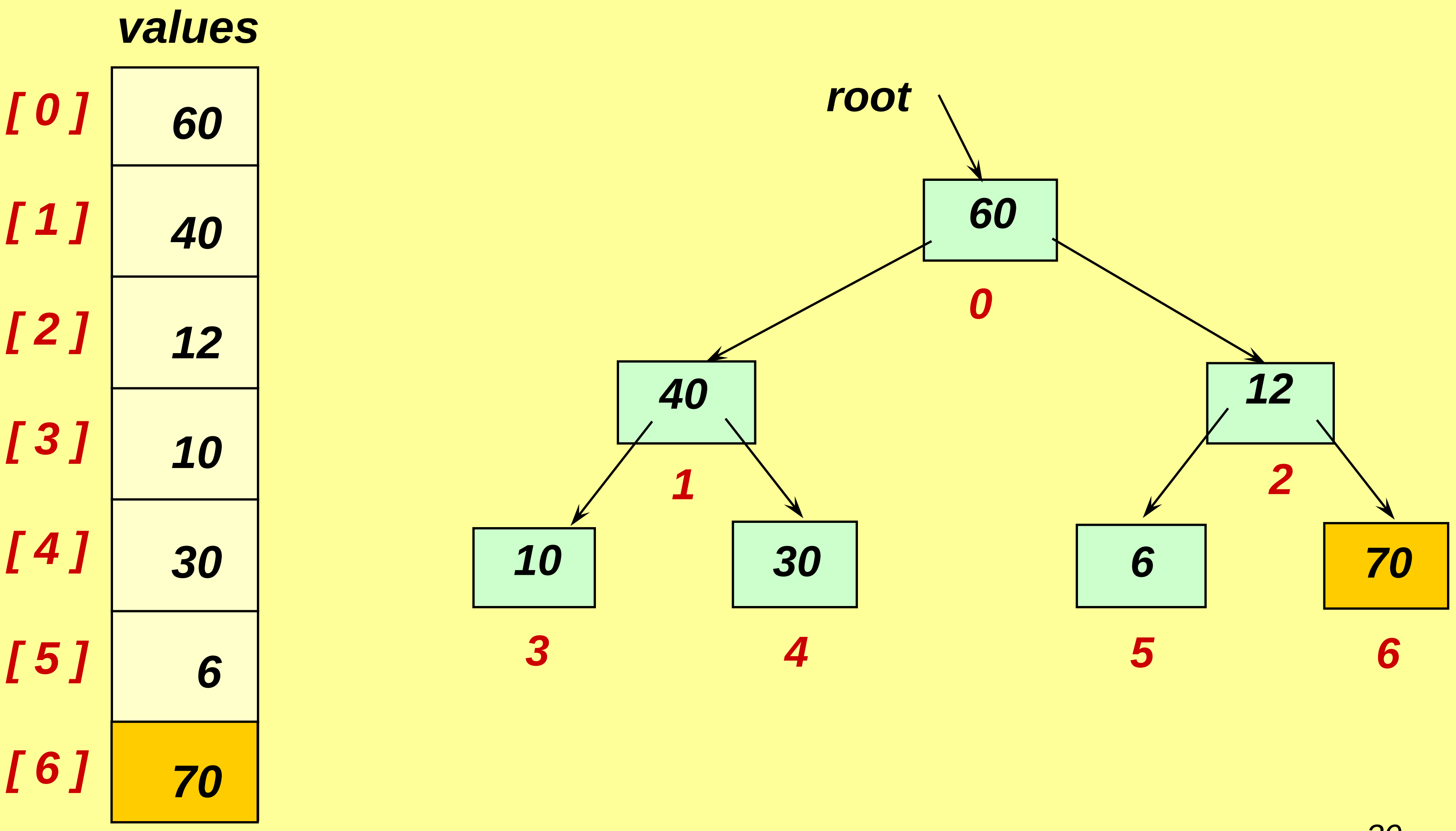
Heap sort: Swap root element into last place in unsorted array



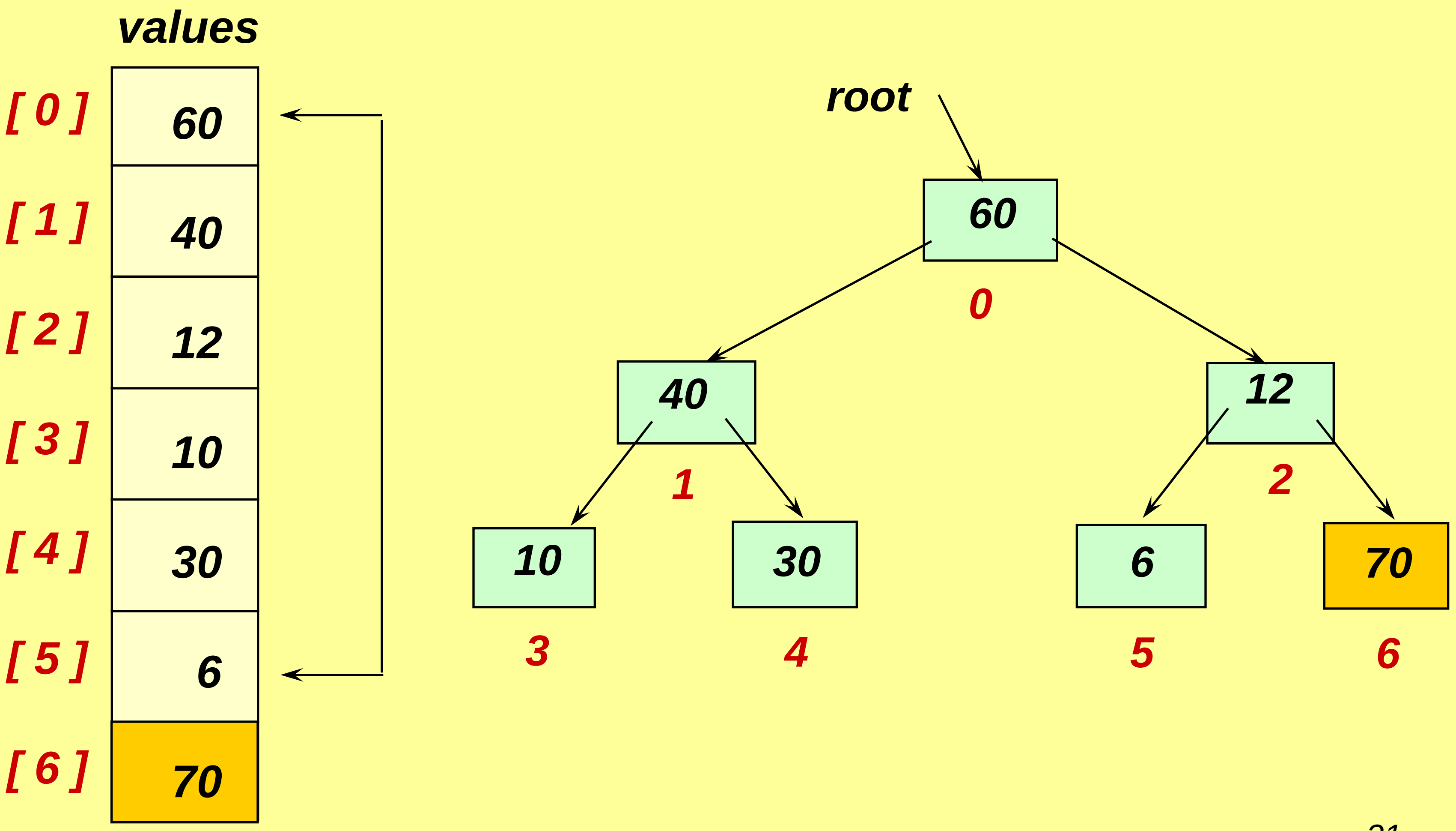
Heap sort: After swapping root element into its place



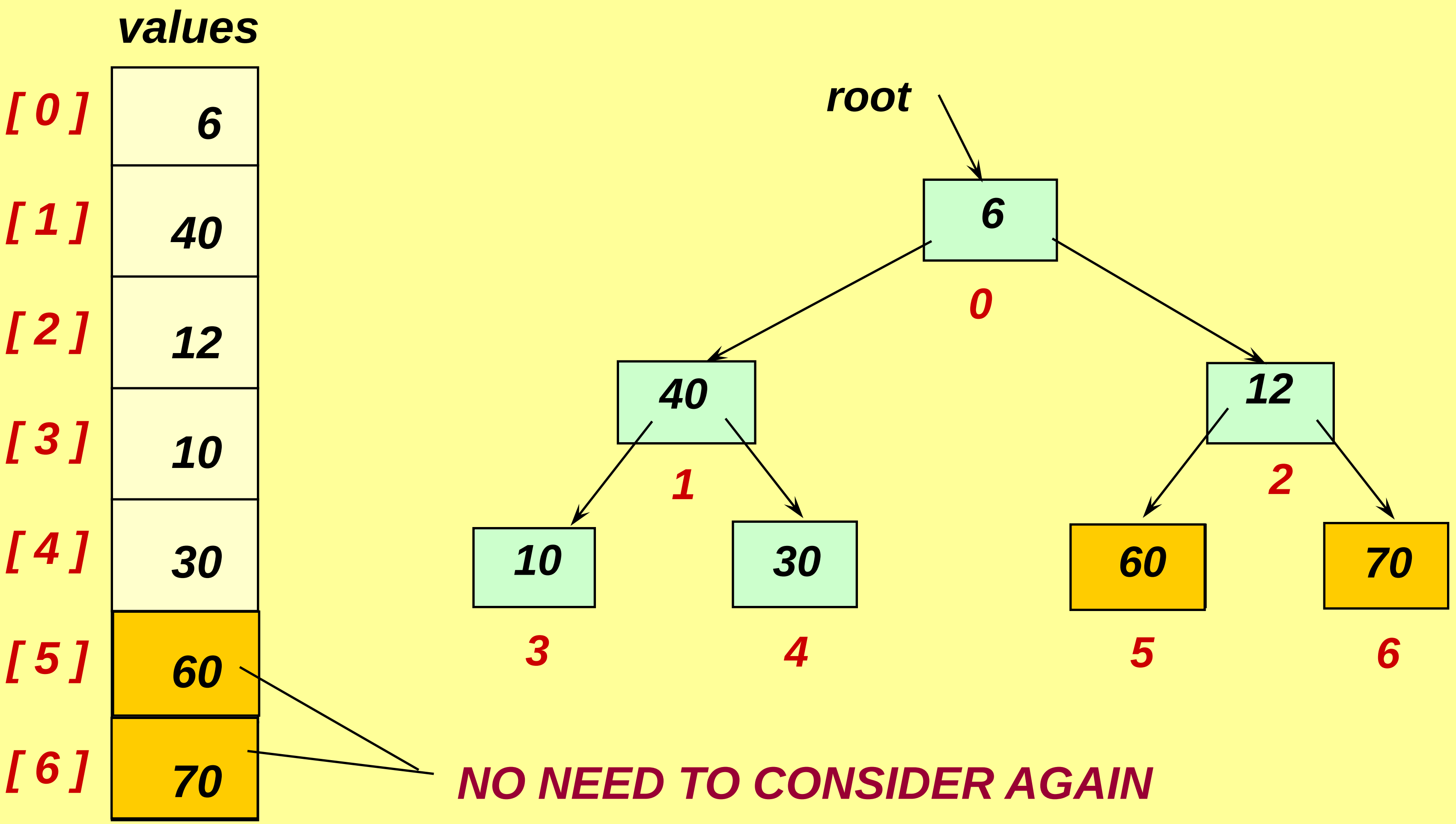
Heap sort: After reheaping remaining unsorted elements



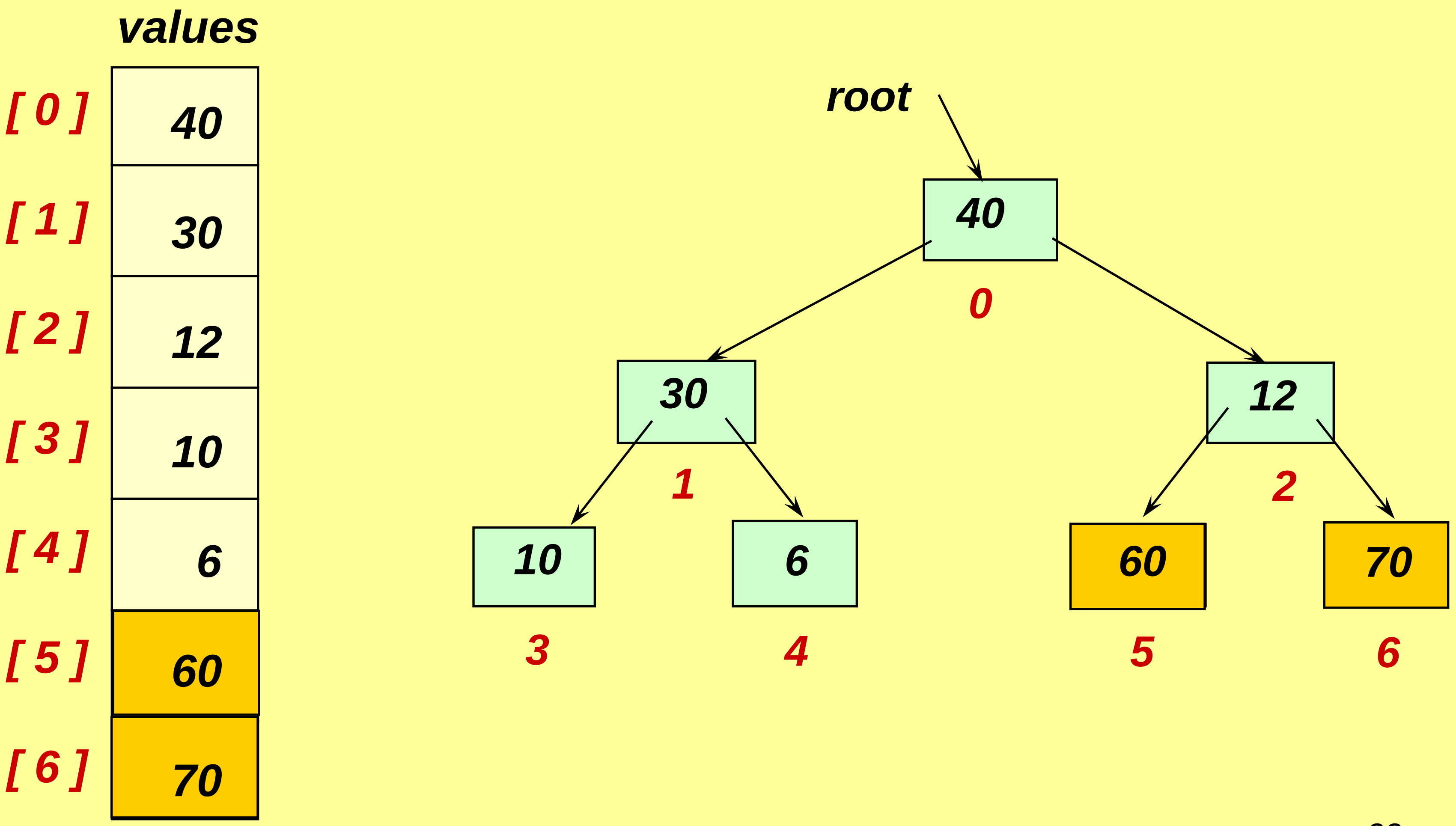
Heap sort: Swap root element into last place in unsorted array



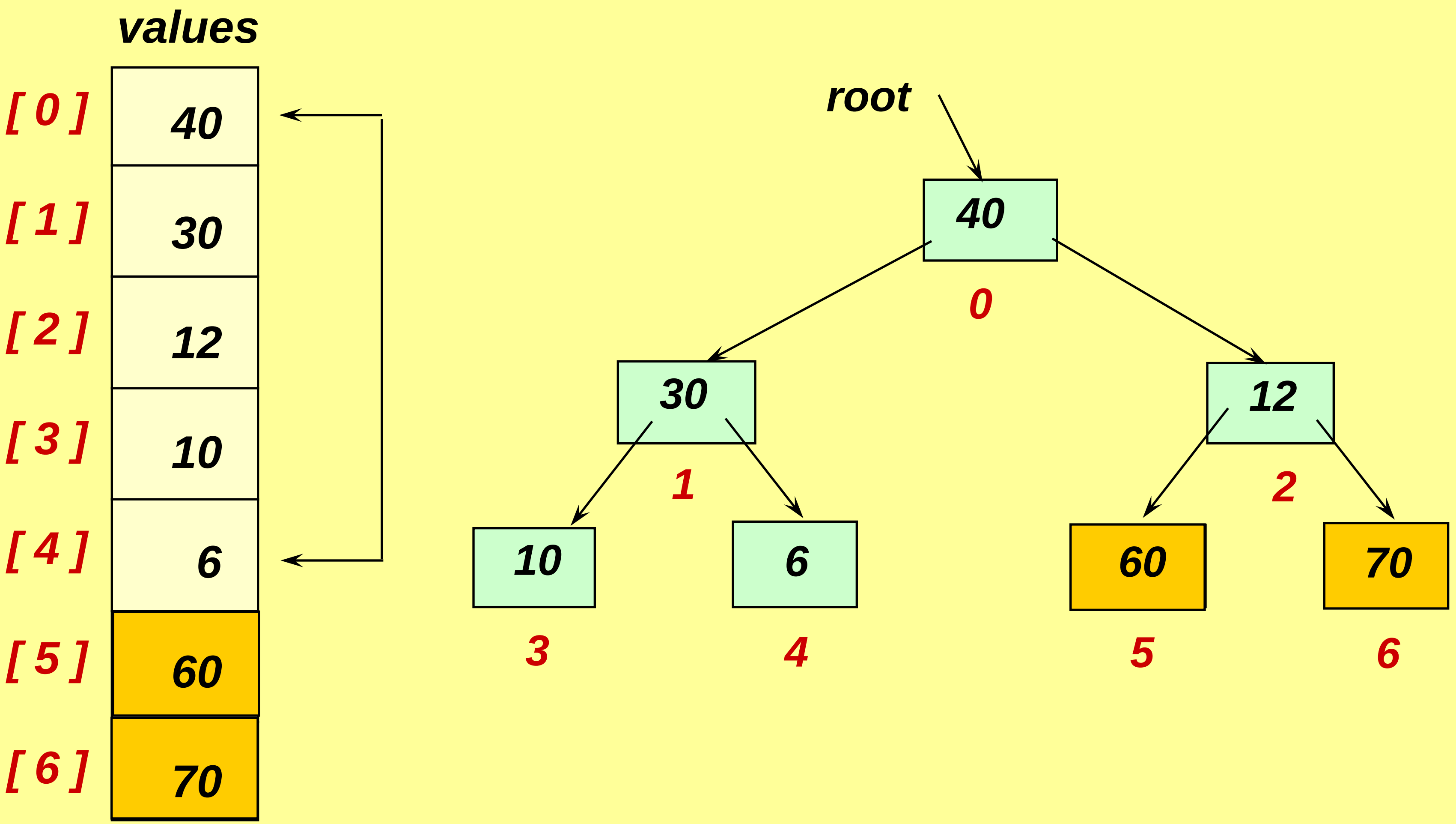
Heap sort: After swapping root element into its place



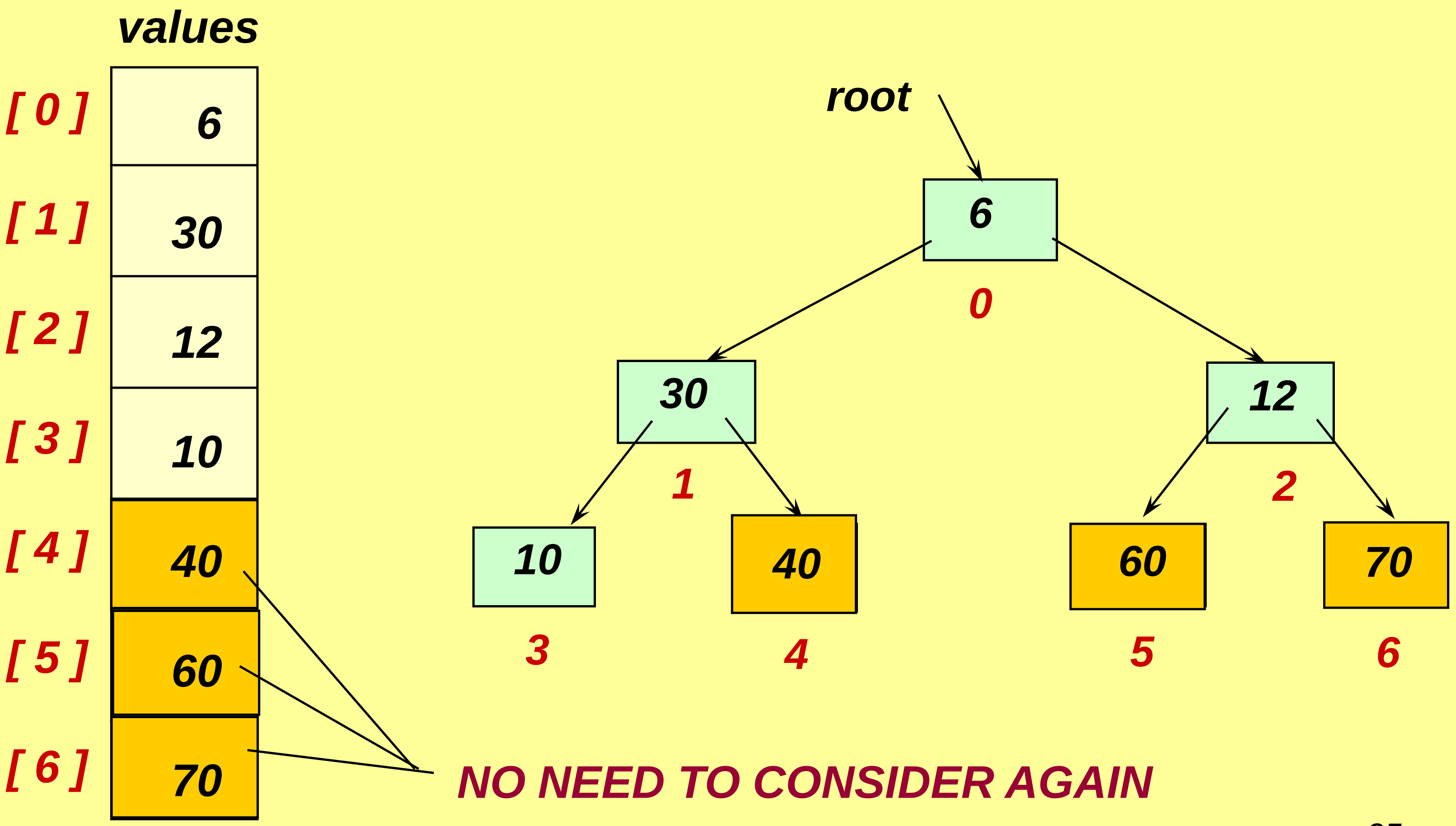
Heap sort: After reheaping remaining unsorted elements



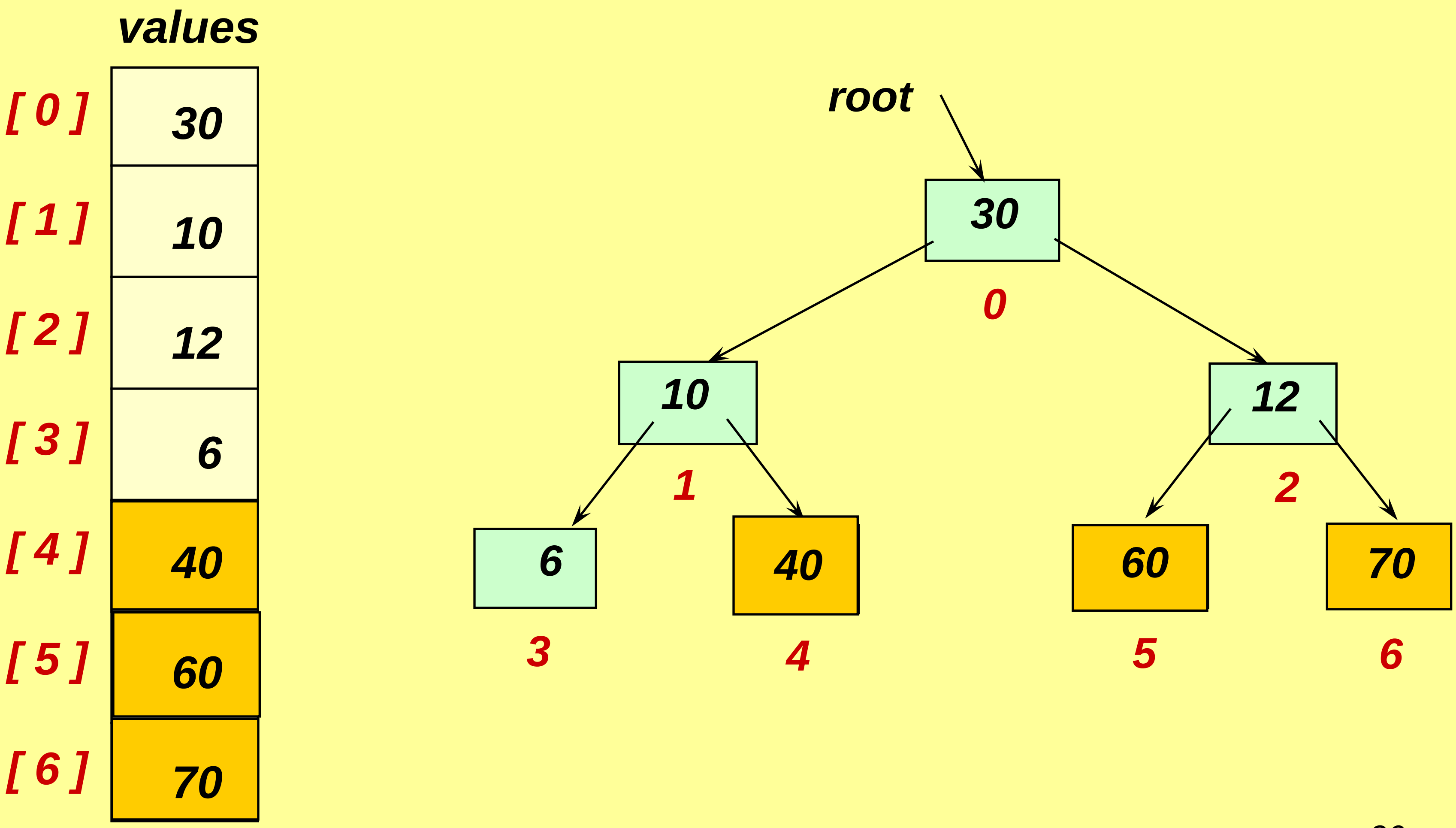
Heap sort: Swap root element into last place in unsorted array



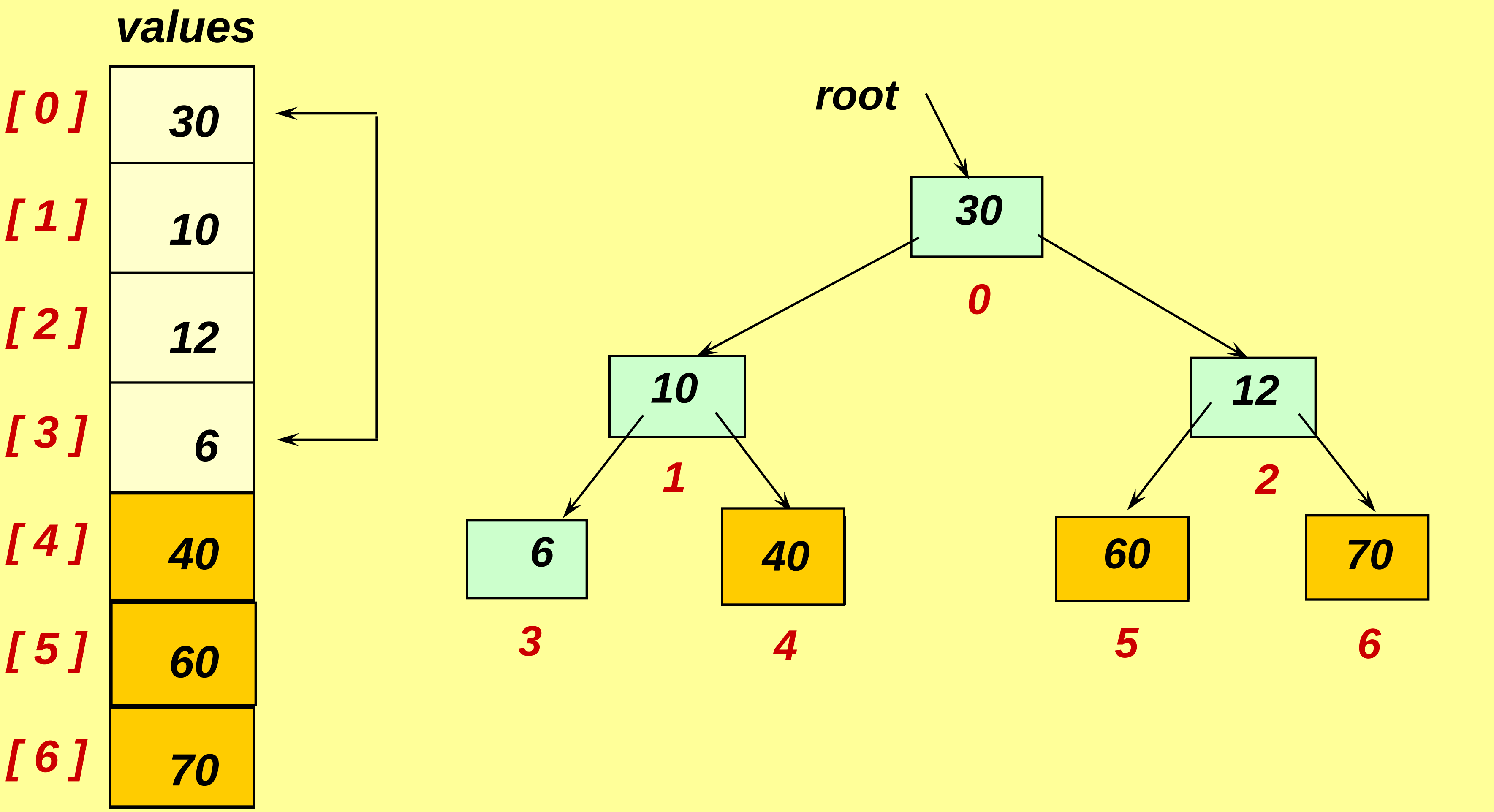
Heap sort: After swapping root element into its place



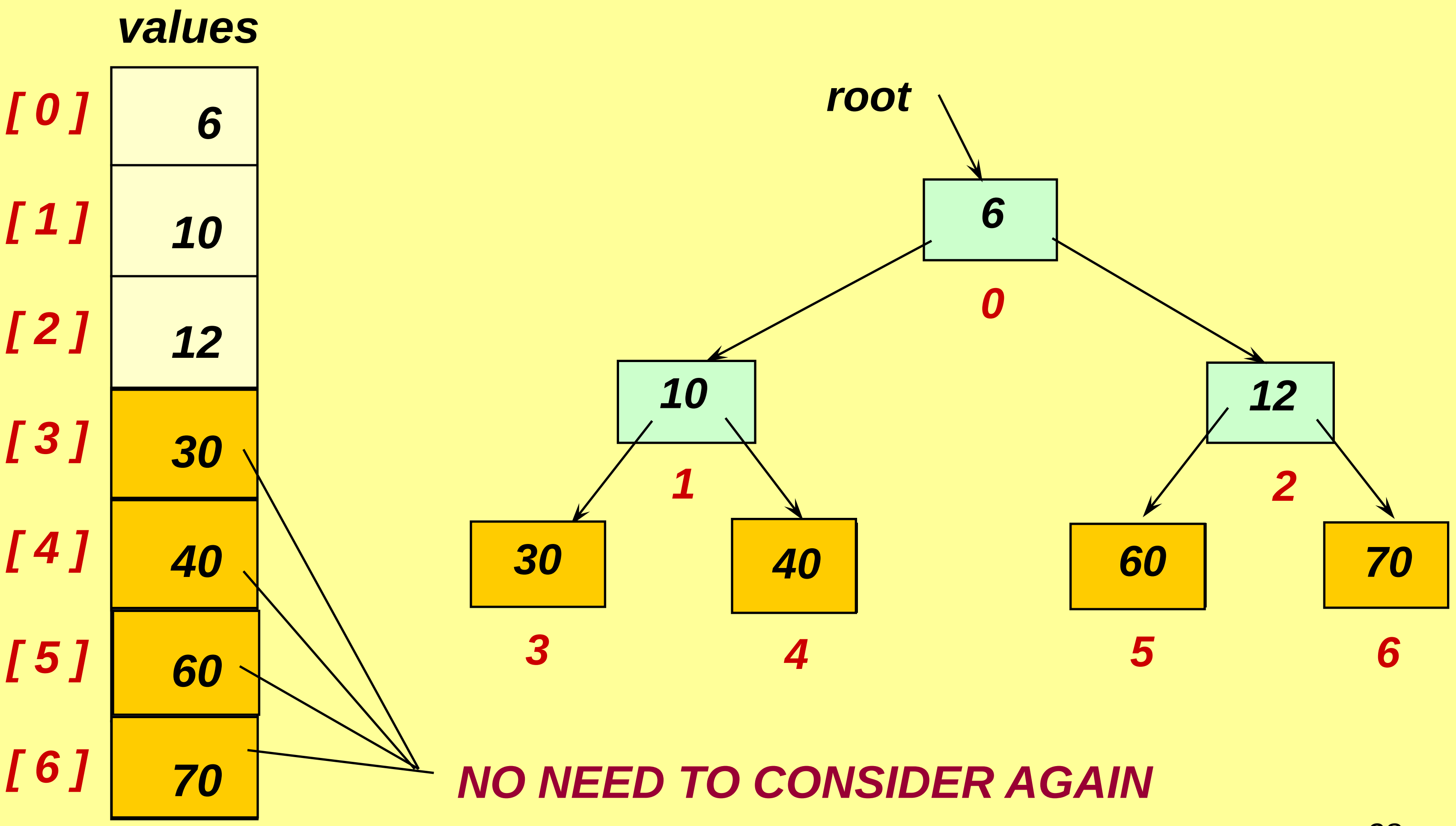
Heap sort: After reheaping remaining unsorted elements



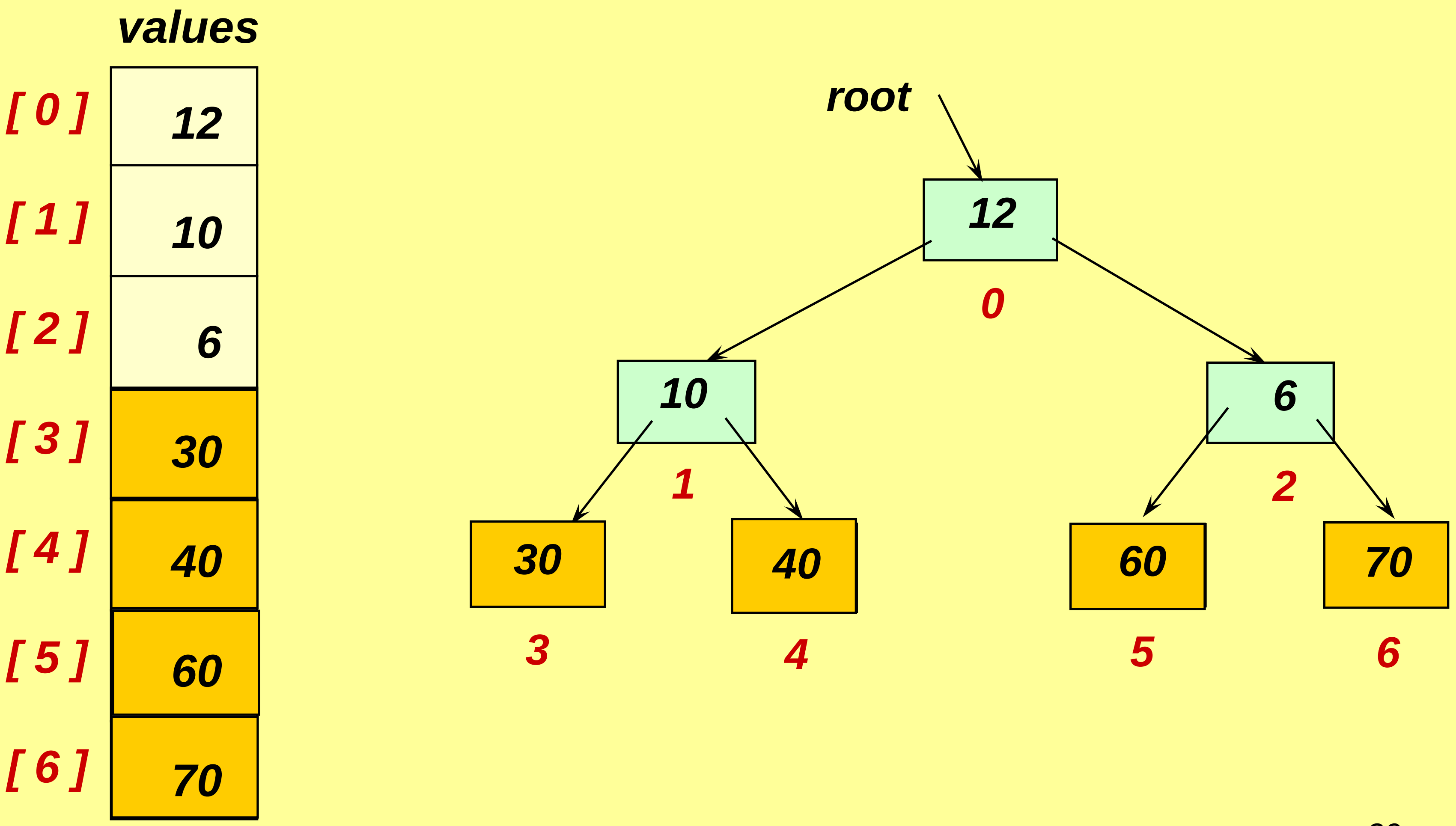
Heap sort: Swap root element into last place in unsorted array



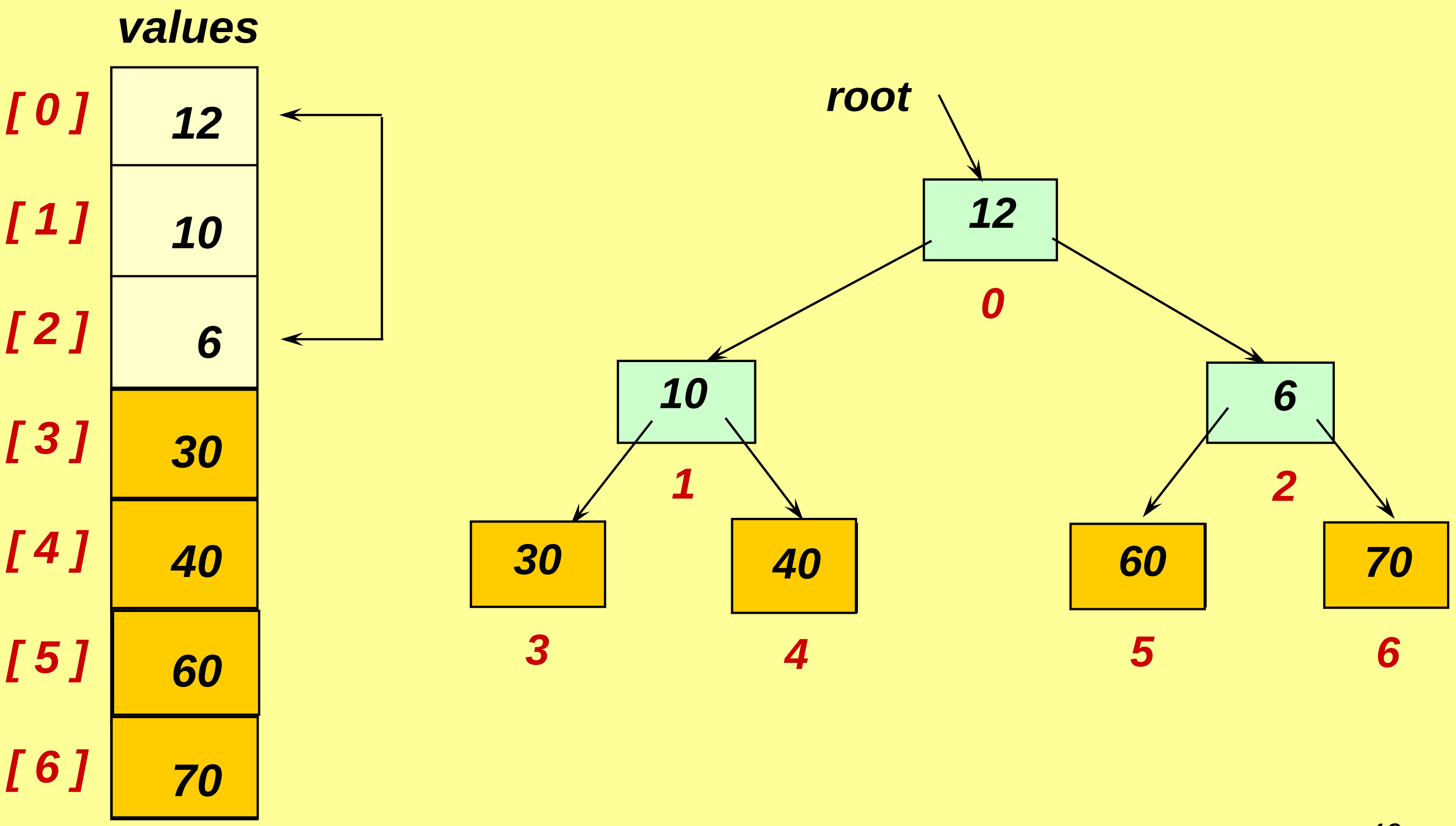
Heap sort: After swapping root element into its place



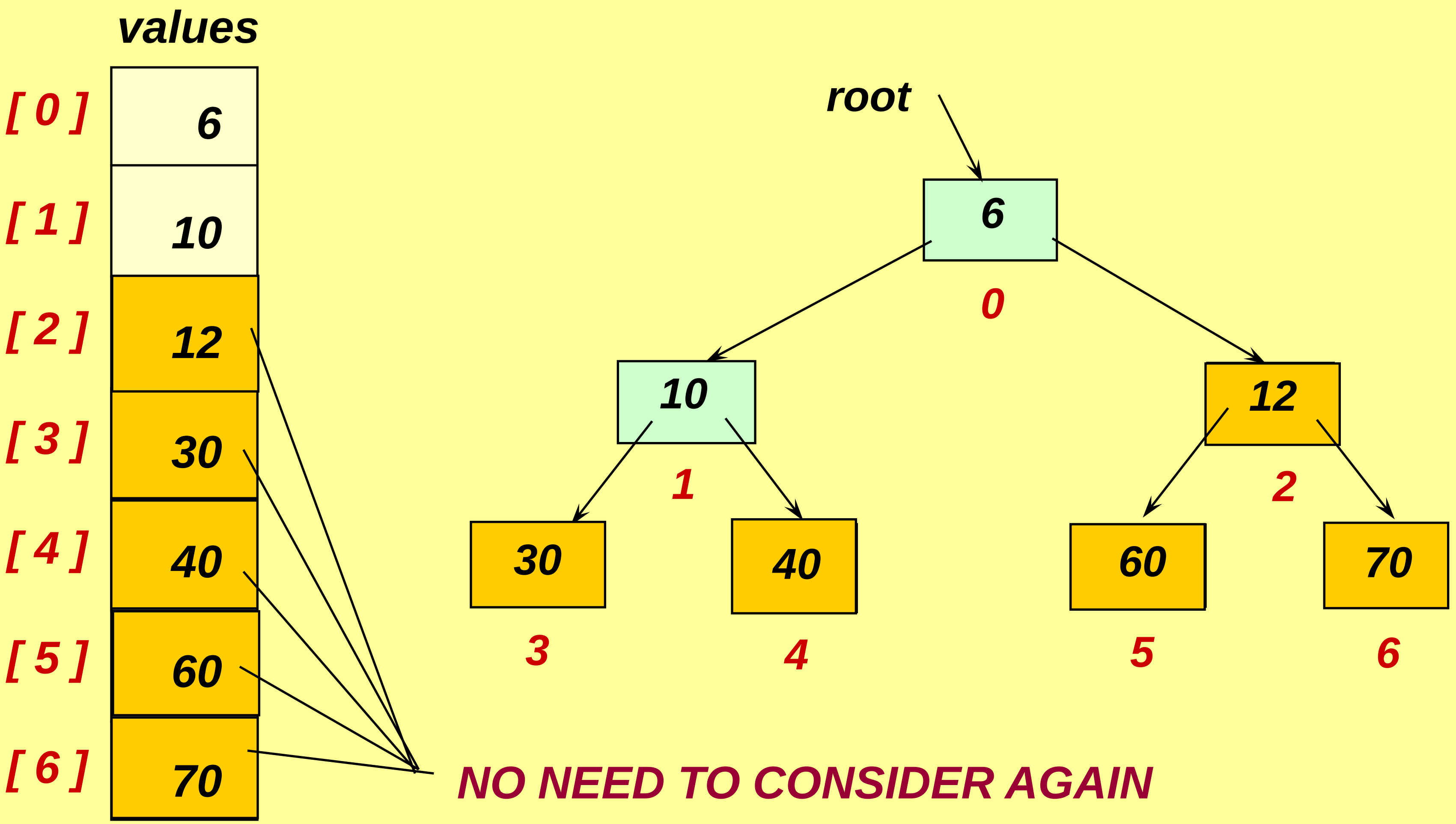
Heap sort: After reheaping remaining unsorted elements



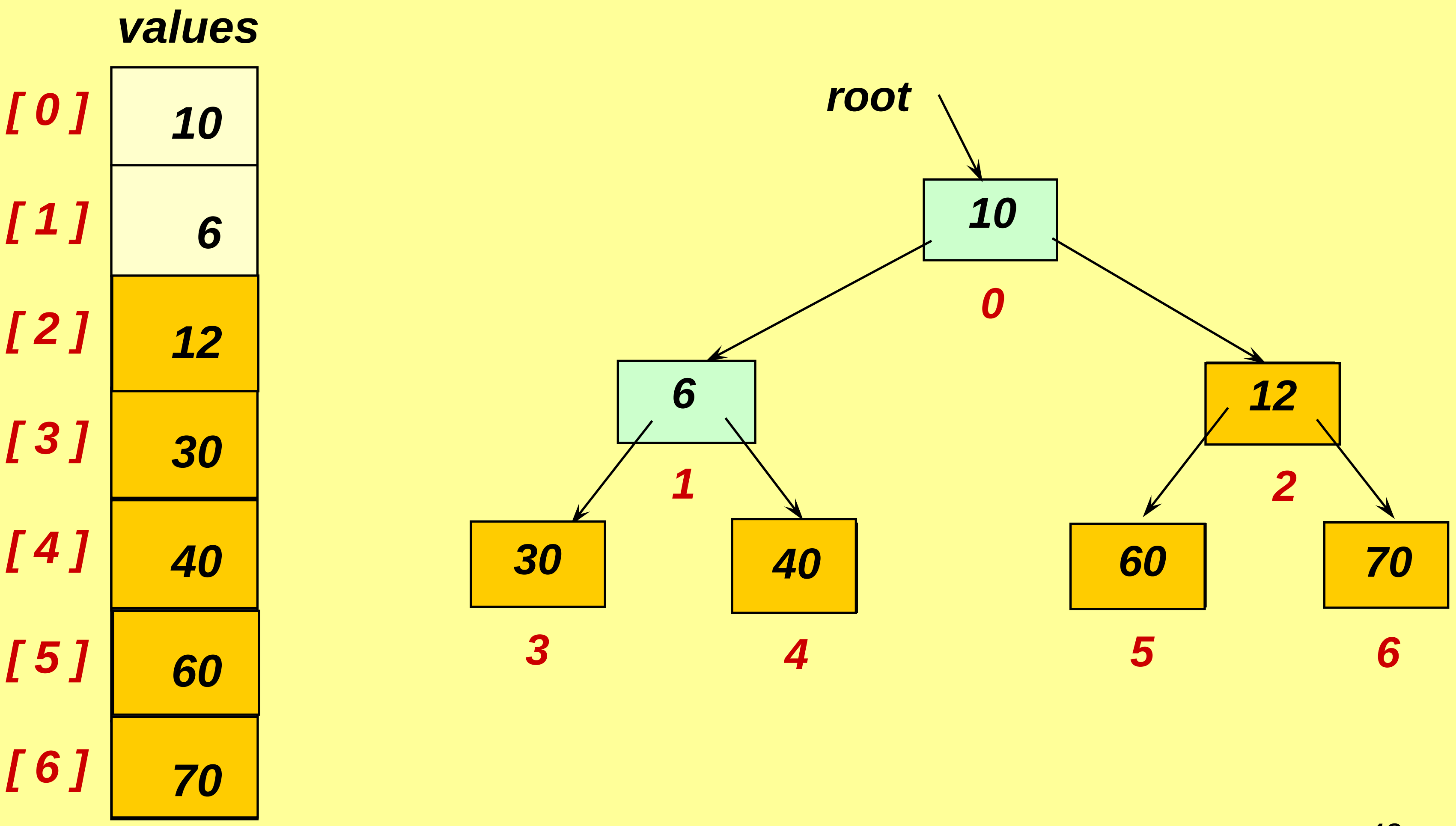
Heap sort: Swap root element into last place in unsorted array



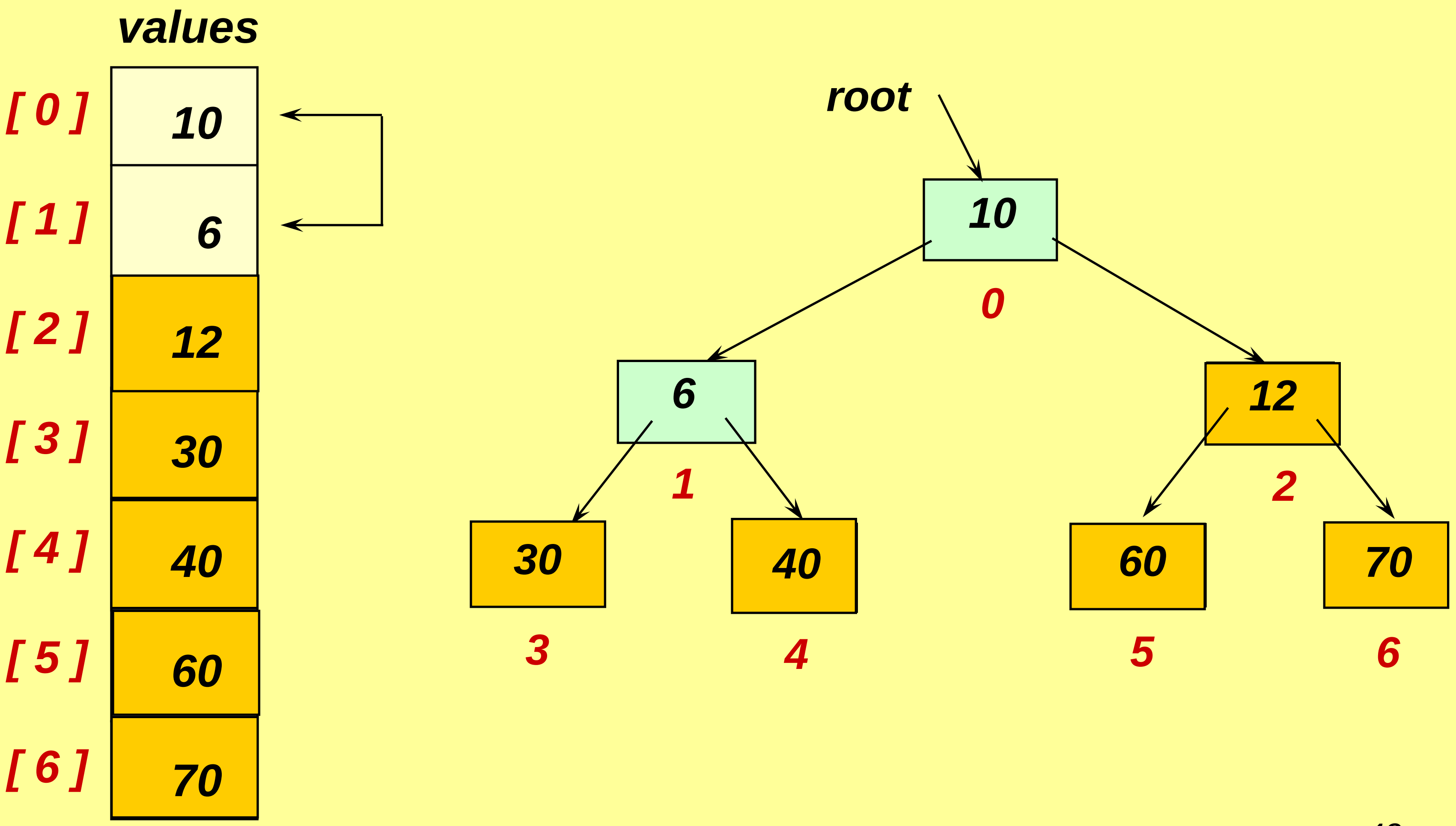
Heap sort: After swapping root element into its place



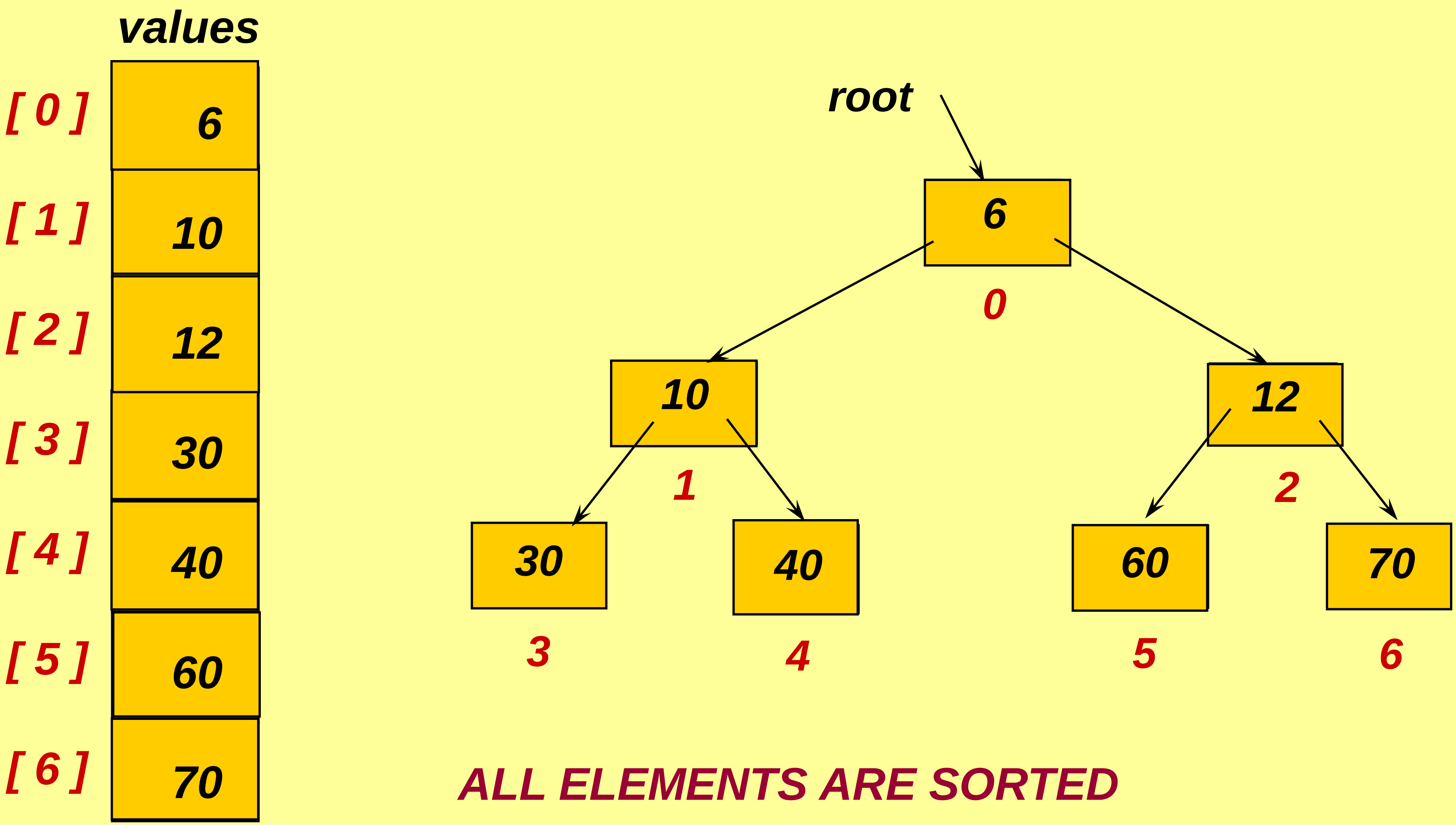
Heap sort: After reheaping remaining unsorted elements



Heap sort: Swap root element into last place in unsorted array

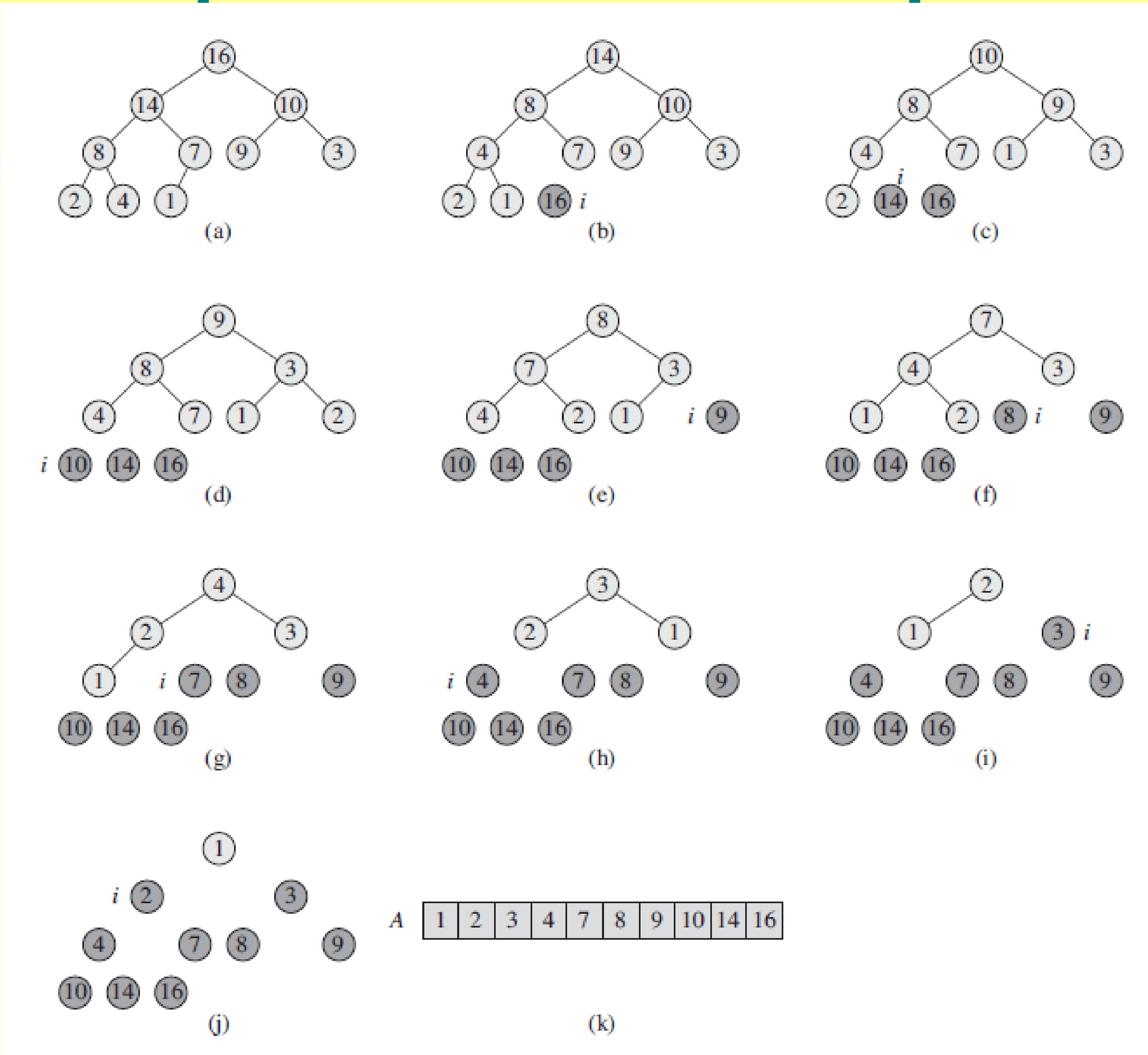


Heap sort: After swapping root element into its place



ALL ELEMENTS ARE SORTED

Heap sort: Another example



Heapsort: Pseudo code

```
Heapsort(A)
{
     heap_size = n;
     BuildHeap(A);
     for (i = n; i >= 2; i--)
     {
          Swap(A[1], A[i]);
          heap_size -= 1;
          Heapify(A, 1);
     }
}
```

Heapsort: Running time

```
Heapsort(A)
{
        heap_size = n;
        BuildHeap(A);                              O(n log n)
        for (i = n; i >= 2; i--)                   n-1 times
        {
                Swap(A[1], A[i]);
                heap_size -= 1;                    O(1)
                Heapify(A, 1);                     O(log n)
        }
}
```

Total: O(n log n)+(n-1)*(O(1)+O(log n)) = O(n log n)+O(n log n) = O(n log n)

Compare with Quicksort

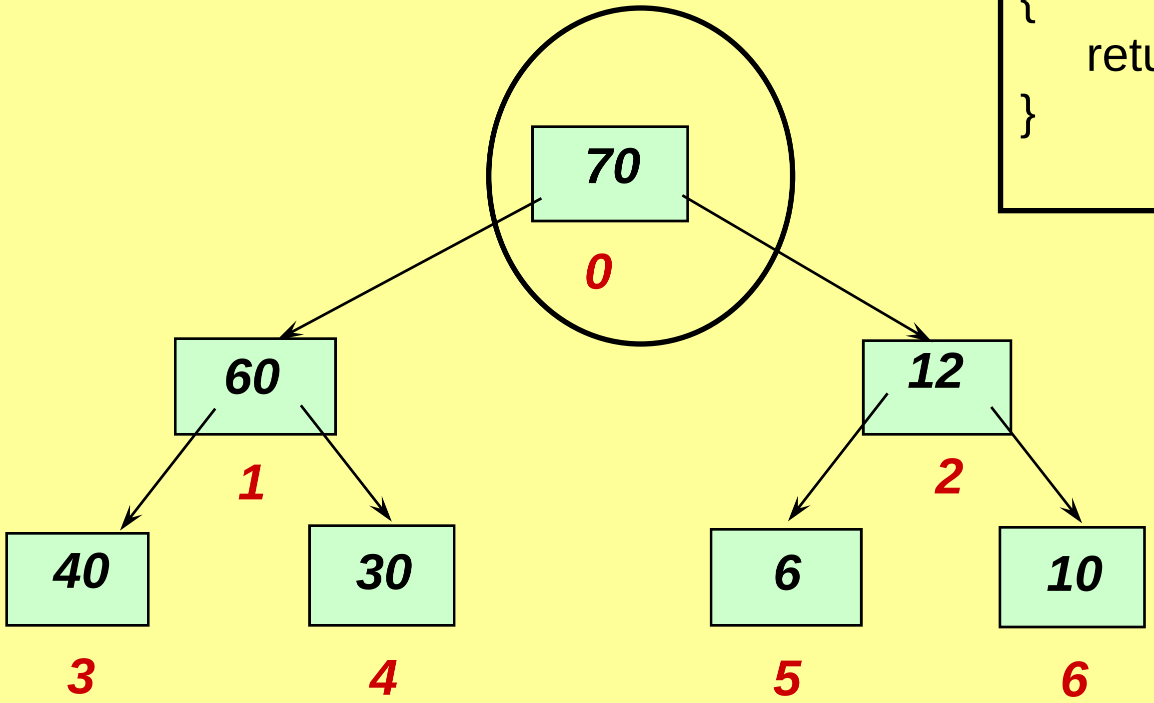
- Heapsort is *always* O(n log n)
- Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
- Quicksort is generally faster, but Heapsort has the guaranteed O(n log n) time and can be used in time-critical applications

Using Heaps: Priority Queues

- But the heap data structure is incredibly useful for implementing *priority queues*
- Priority Queue has following operations
  - **Maximum()** returns the maximum element
  - **ExtractMax()** removes and returns the maximum element
  - **Increasekey(i, new_value)** change the value of position ***i*** to new higher value (**`new_value`**)
  - **Insert(x)** inserts the element x into the queue

Heap Maximum

- Return the root element
- Cost: O(1)

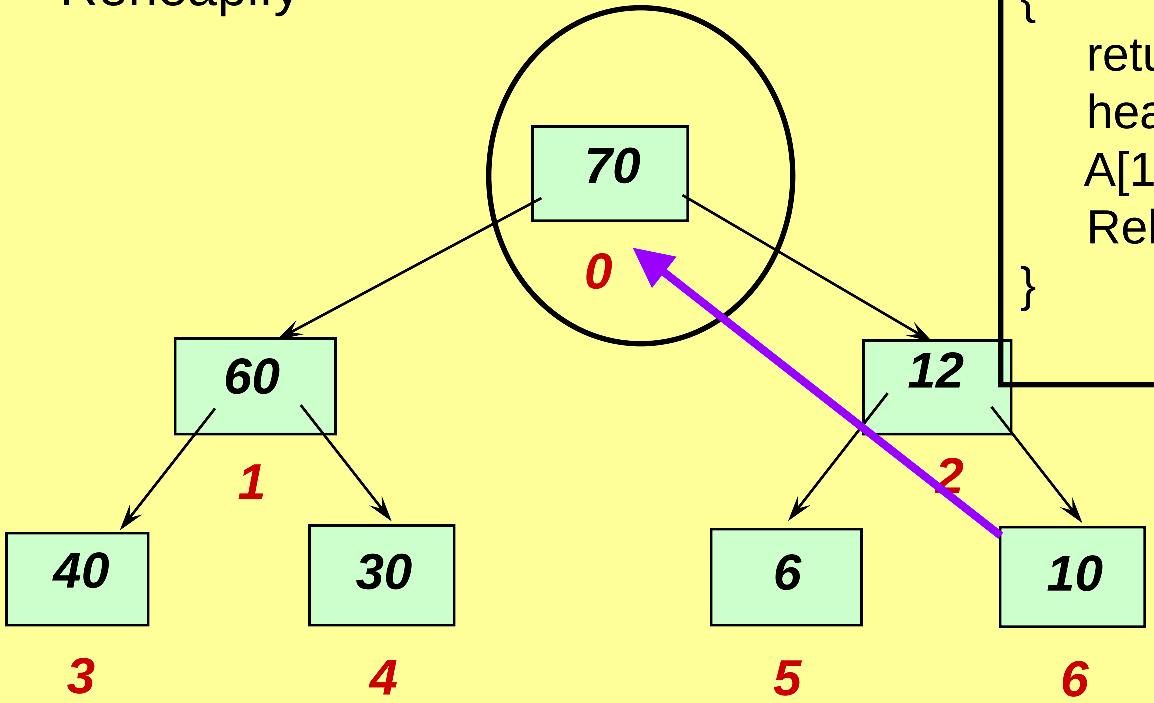


Pseudo code

```
HeapMax()
{
    return A[1];
}
```

Heap Extract Maximum

- Return the root element
- Delete the root element
- Reheapify

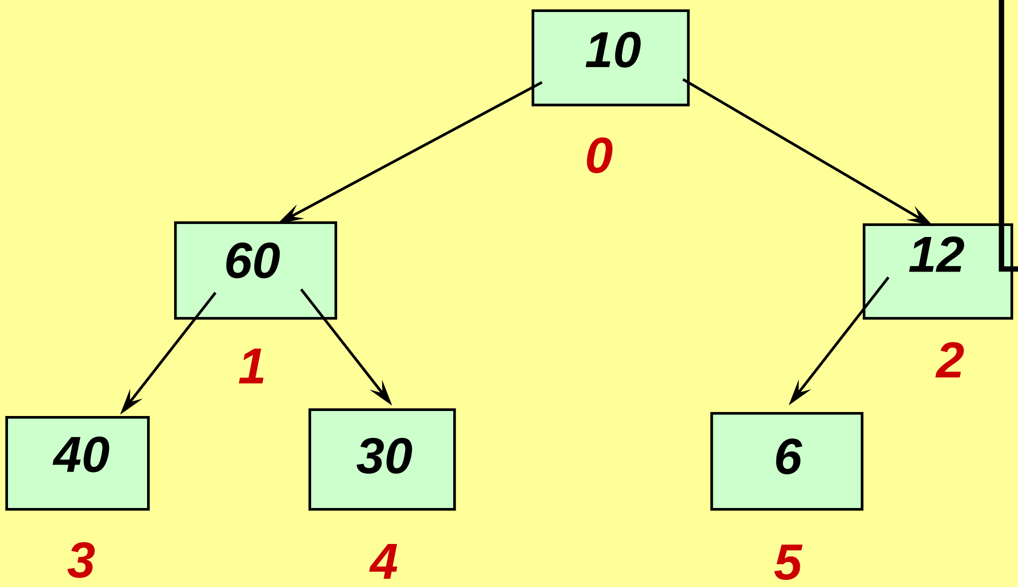


Pseudo code

```
HeapExtractMax()
{
    return A[1];
    heap_size = heap_size-1;
    A[1] = A[heap_size];
    Reheapify();
}
```

Heap Extract Maximum

- Return the root element
- Delete the root element
- Reheapify

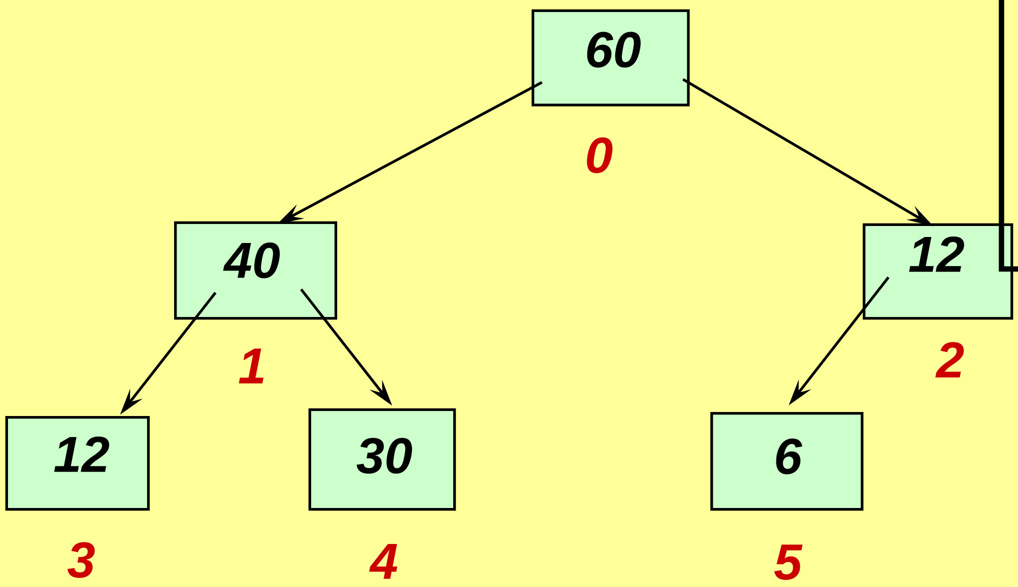


Pseudo code

```
HeapExtractMax()
{
    return A[1];
    heap_size = heap_size-1;
    A[1] = A[heap_size];
    Reheapify();
}
```

Heap Extract Maximum

- Return the root element
- Delete the root element
- Reheapify

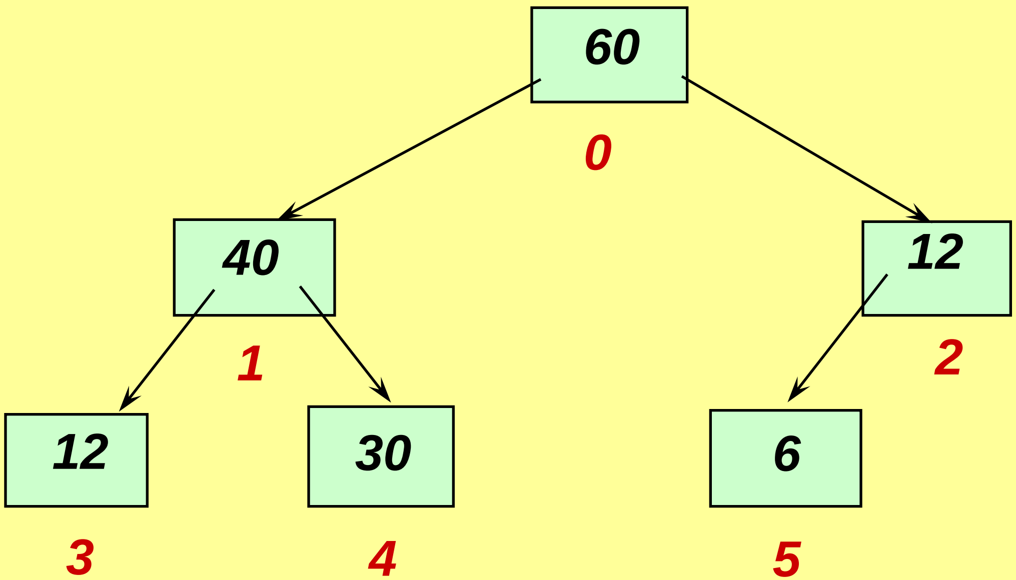


Pseudo code

```
HeapExtractMax()
{
    return A[1];
    heap_size = heap_size-1;
    A[1] = A[heap_size];
    Reheapify();
}
```

Heap ExtractMaximum

- Return the root element
- Delete the root element
- Reheapify



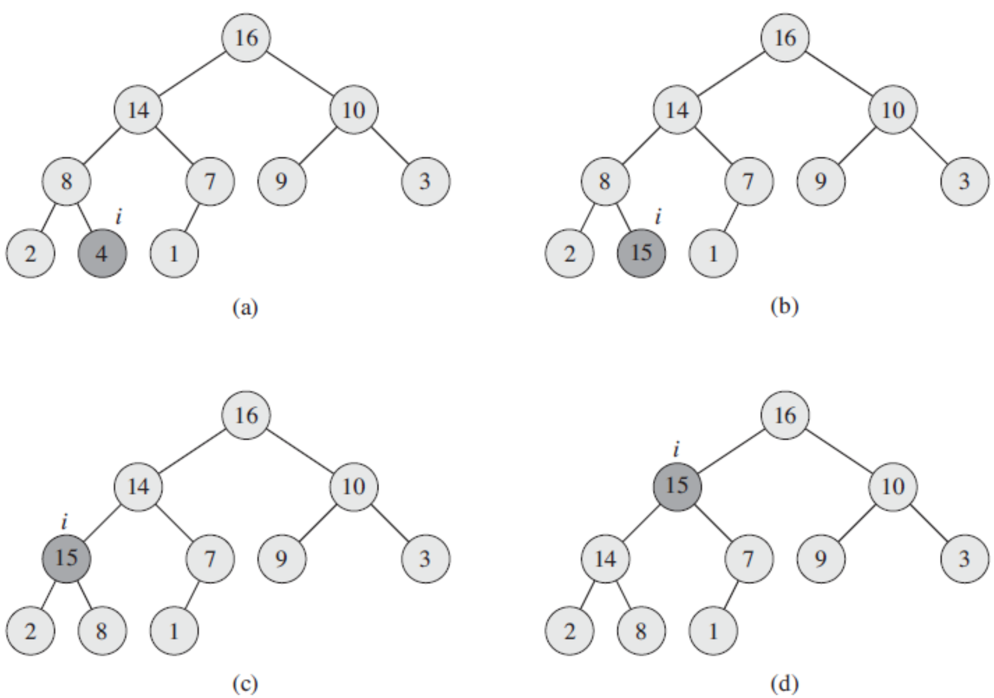
Pseudo code

```
HeapExtractMax()
{
    return A[1];
    heap_size = heap_size-1;
    A[1] = A[heap_size];
    Reheapify();
}
```

- Cost: O(1)+O(1)+O(1)+O(log n) = O(log n)

Priority Queue Operations: IncreaseKey

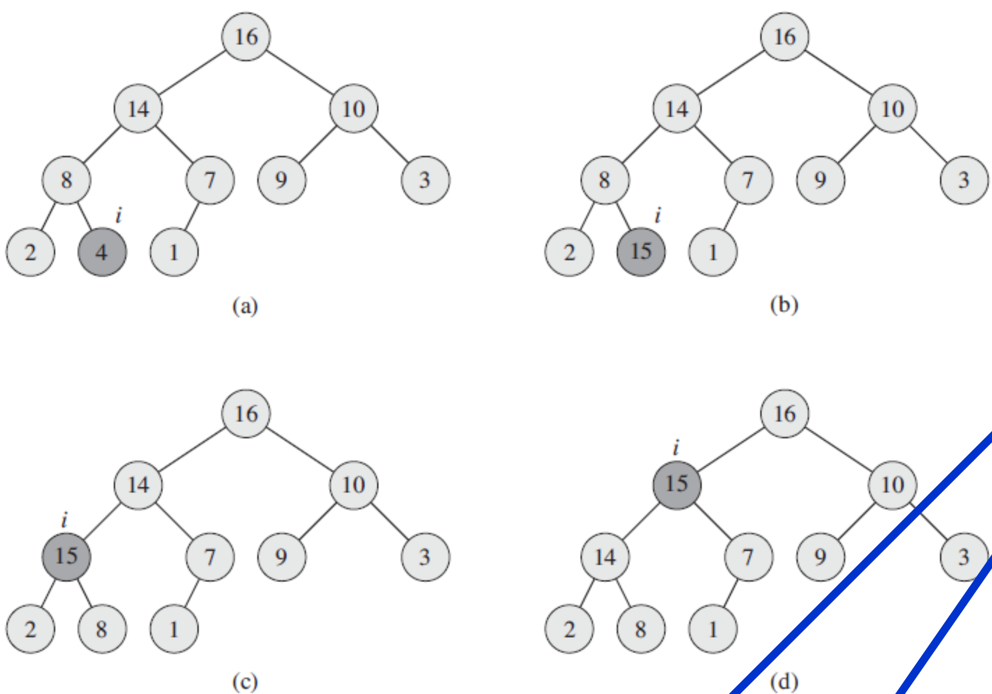
-Increase a value to **new_value**
- do swapping as long as necessary to maintain the heap property



<u>Pseudo code</u>

```
IncreaseKey(i, new_value)
{
    A[i] = new_value;
    while (i>1 and A[parent(i)] < A[i])
        swap(A[parent(i)], A[i]);
        i = parent(i);
    }
```

Priority Queue Operations: IncreaseKey



Pseudo code

```
IncreaseKey(i, new_value)
{
    A[i] = new_value;
    while (i>1 and A[parent(i)] < A[i])
        swap(A[parent(i)], A[i]);
        i = parent(i);
    }
```

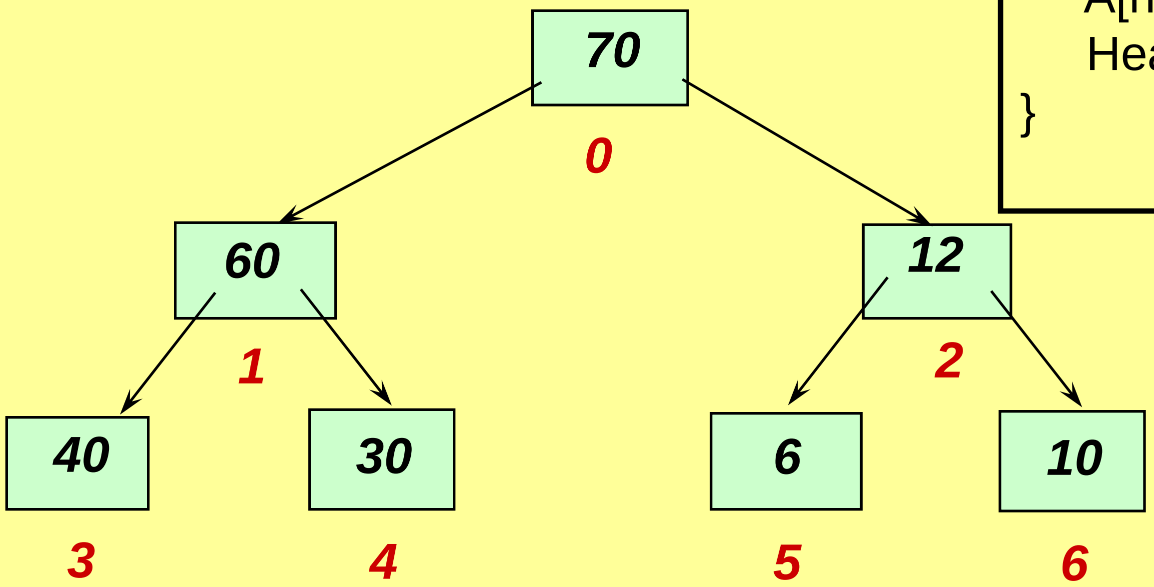
- Cost: O(1)+height*(O(1)+O(1)) = O(log n)*O(1) = O(log n)

Heap Insert

- Insert a node with -∞ at the end
- then, increase the node to x by HeapIncrease()
- HeapIncrease() will do the necessary heapify

Pseudo code

```
HeapInsert(x)
{
    hear_size = heap_size+1;
    A[heap_size] = x;
    HeapIncrease(heap_size,x);
}
```

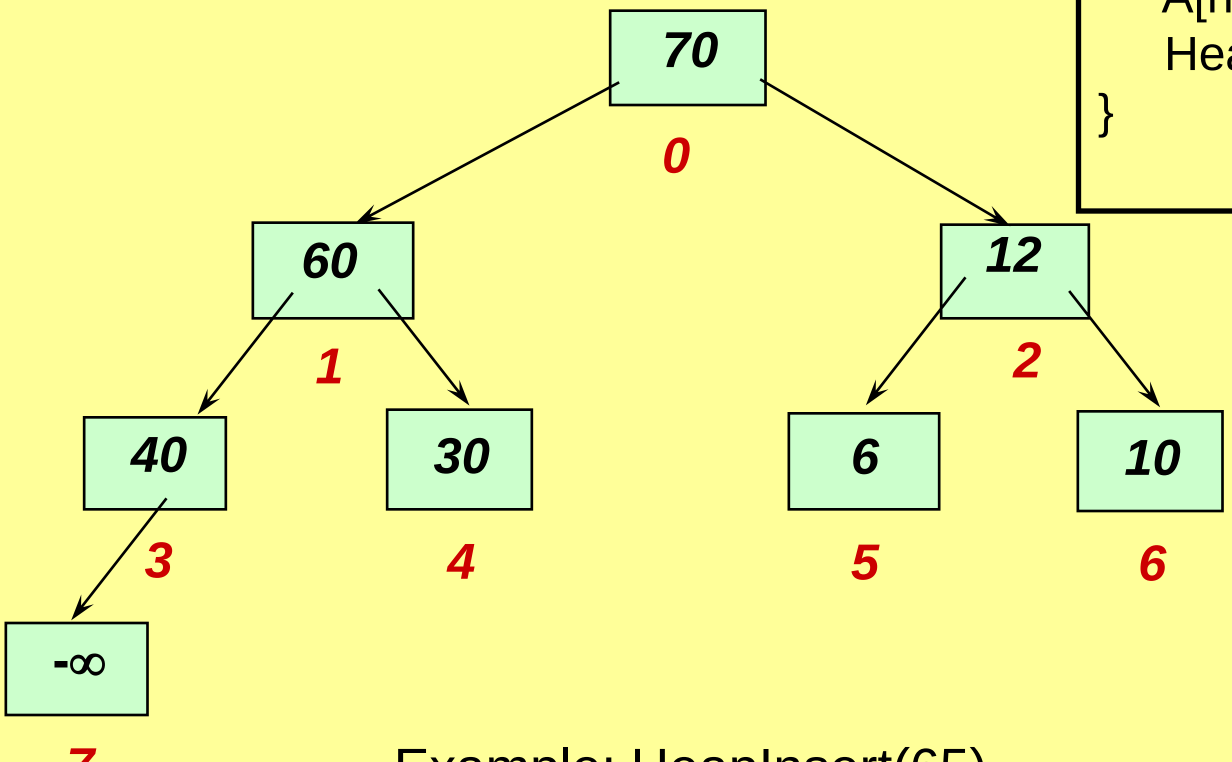


Heap Insert

- Insert a node with -∞ at the end
- then, increase the node to x by HeapIncrease()
- HeapIncrease() will do the necessary heapify

Pseudo code

```
HeapInsert(x)
{
    hear_size = heap_size+1;
    A[heap_size] = x;
    HeapIncrease(heap_size,x);
}
```



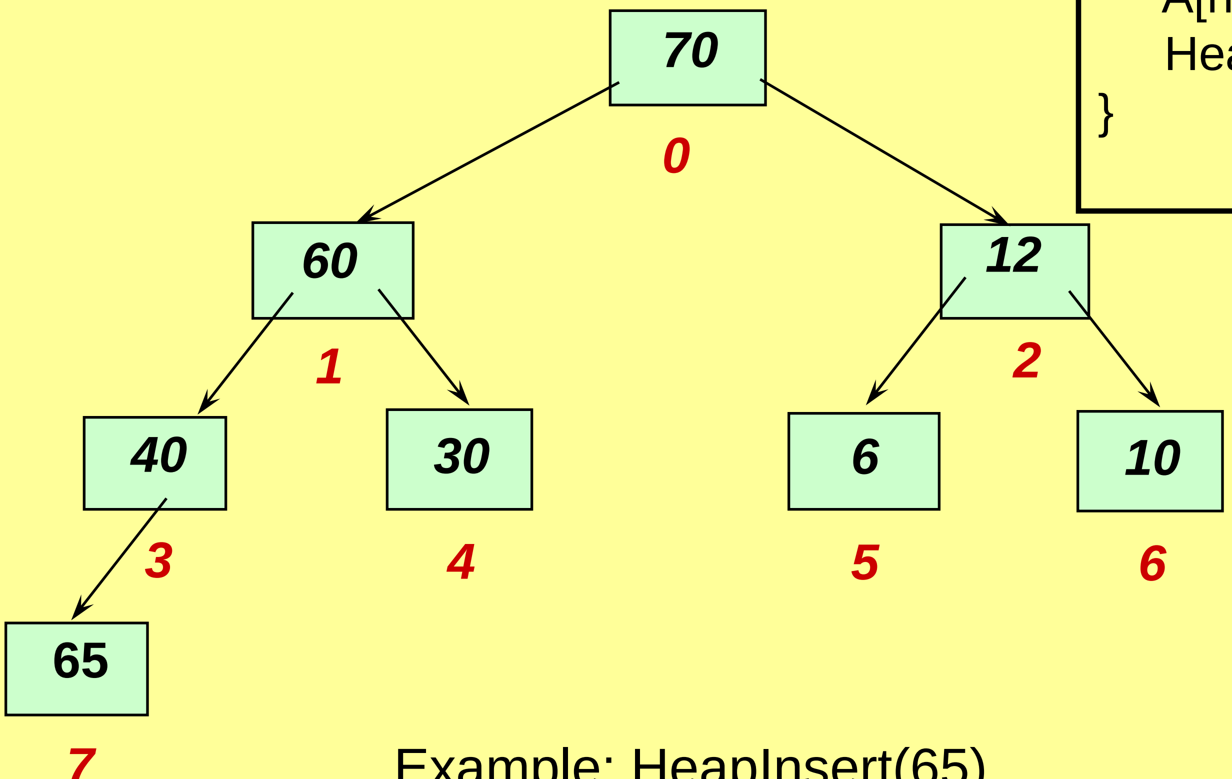
Example: HeapInsert(65)

Heap Insert

- Insert a node with -∞ at the end
- then, increase the node to x
  by HeapIncrease()
- HeapIncrease() will do the necessary
  heapify

Pseudo code

```
HeapInsert(x)
{
    hear_size = heap_size+1;
    A[heap_size] = x;
    HeapIncrease(heap_size,x);
}
```



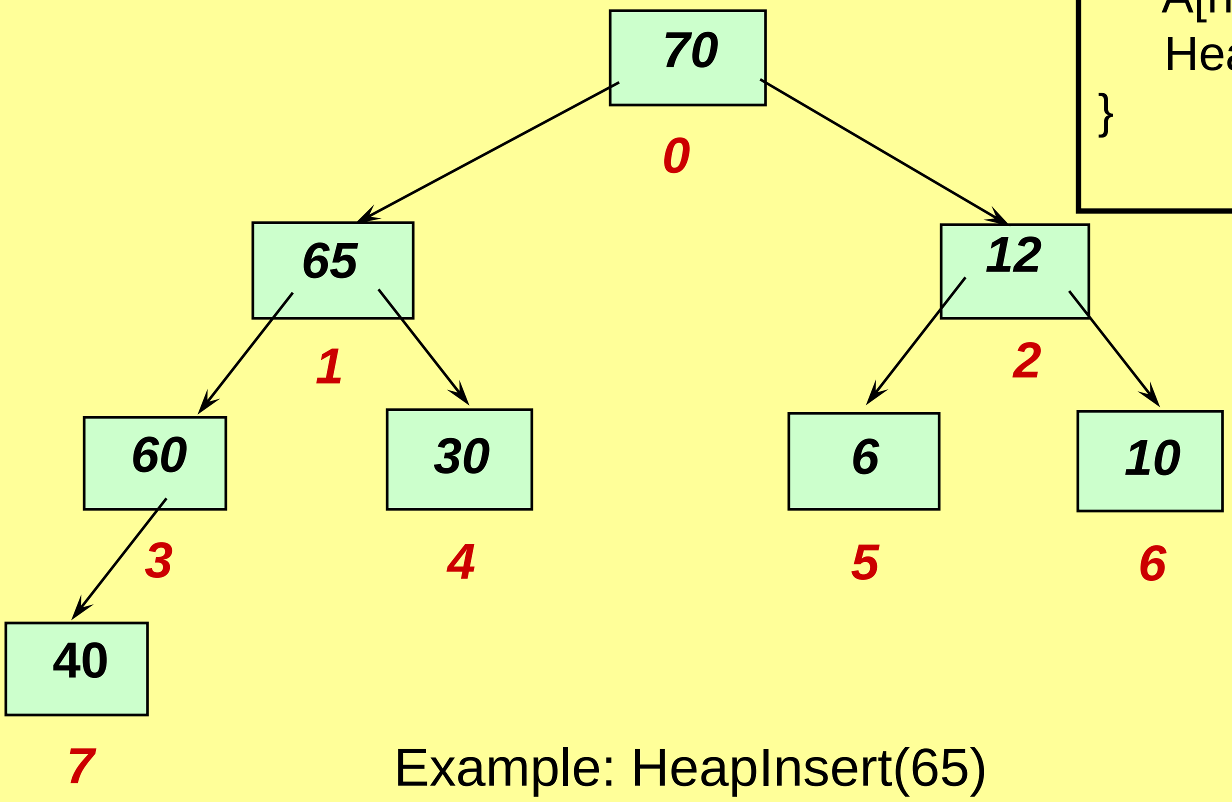
Example: HeapInsert(65)

Heap Insert

- Insert a node with -∞ at the end
- then, increase the node to x by HeapIncrease()
- HeapIncrease() will do the necessary heapify

Pseudo code

```
HeapInsert(x)
{
    hear_size = heap_size+1;
    A[heap_size] = x;
    HeapIncrease(heap_size,x);
}
```



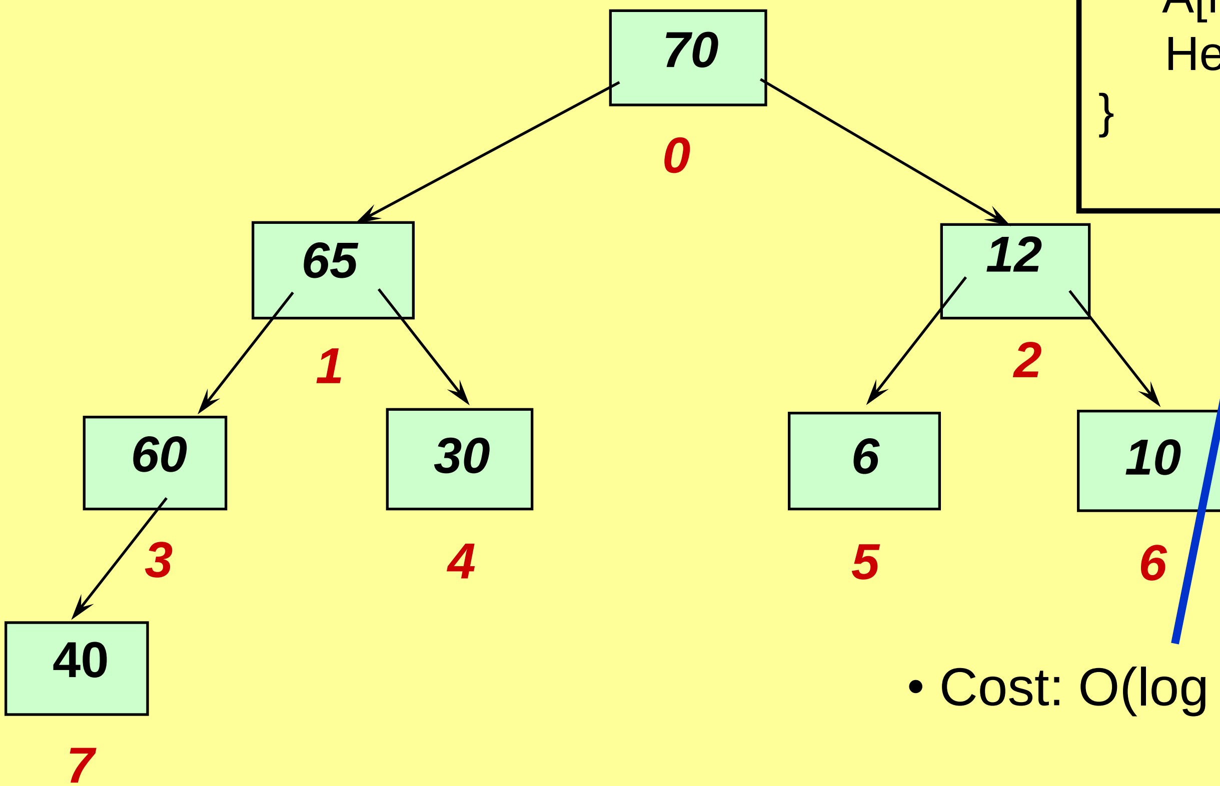
Example: HeapInsert(65)

Heap Insert

- Insert a node with -∞ at the end
- then, increase the node to x by HeapIncrease()
- HeapIncrease() will do the necessary heapify

Pseudo code

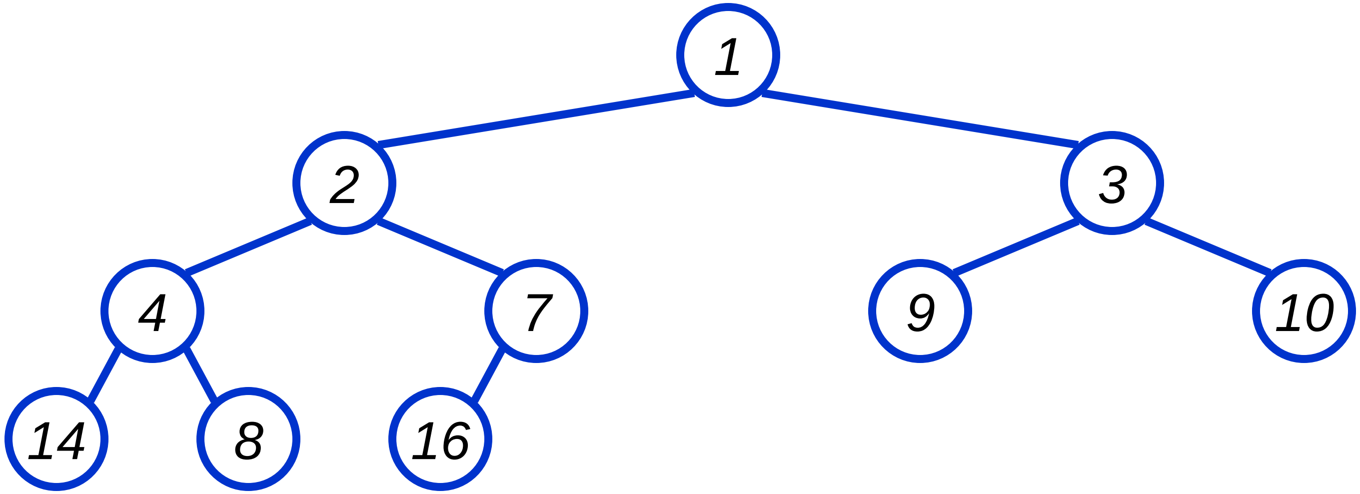
```
HeapInsert(x)
{
     hear_size = heap_size+1;
     A[heap_size] = x;
     HeapIncrease(heap_size,x);
}
```



- Cost: O(log n)+O(1)+O(1)) = O(log n)

Min Heap

- ***Parent <= left, right*** for all nodes
- All operations similar to Max Heap: Heapify(), BuildHeap(), HeapSort(), HeapMin(), HeapInsert(), HeapExtractMin(), DecreaseKey().



A =	1	2	3	4	7	9	10	14	8	16