Priority Queue

Mark Allen Weiss

Priority

- In many real-world scenarios one job is more important than other
  - There is a long queue of printing jobs some are one-page while other are 100 pages long. It is desirable to print short jobs first

  - In grocery store, we have separate counter for people who bought less than 5 items.

Priority

- We may want to prioritize on the basis of
  - First come First serve –FCFS (Time stamp)
  - Minimum item first
  - Maximum item first

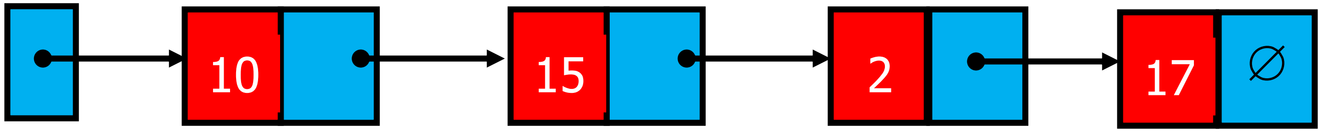
- Which data structure would be the best choice
  - FCFS
    - Queue
  - Minimum First ???

Priority Queue

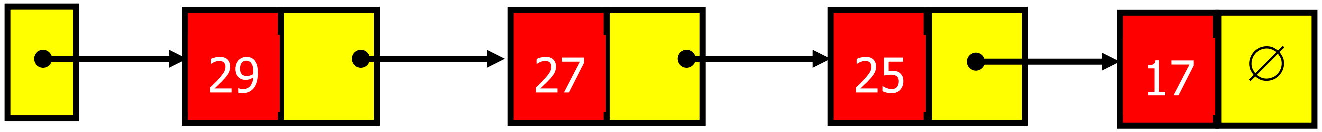
- A priority queue should allow at least the following two operations:
  - **Insert (enqueue)**
  - **deleteMax**, which finds, returns, and removes the maximum element in the priority queue (**dequeue**)

Simple Implementations

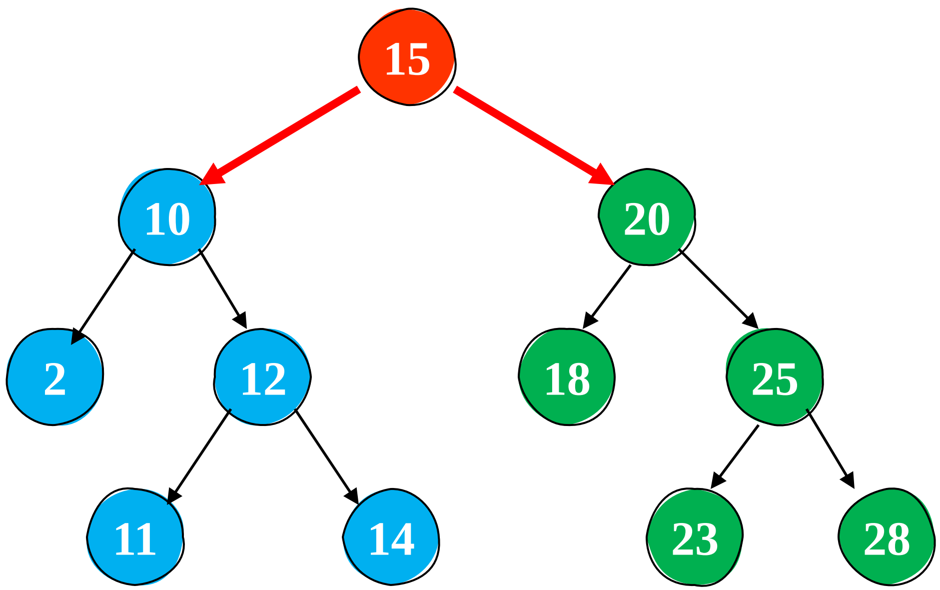
- Use a simple linked list for Priority Queue
  - perform insertions at the front in $O(1)$
  - DeleteMAX in $O(N)$ time. **WHY?**



- Alternatively,
  - Keep List sorted
    - this makes insertions expensive ($O(N)$) and
    - deleteMax cheap ($O(1)$).

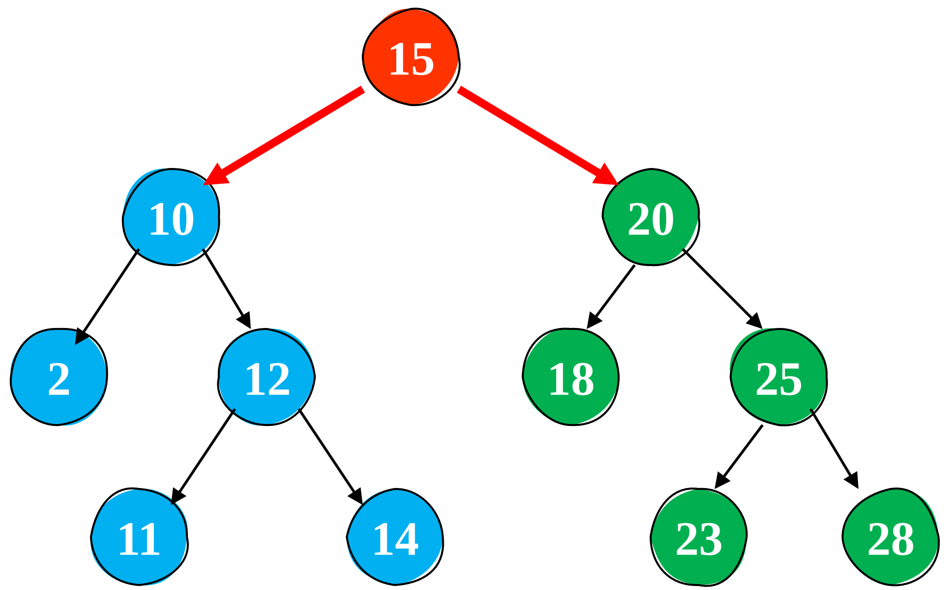


BST as Priority Queue



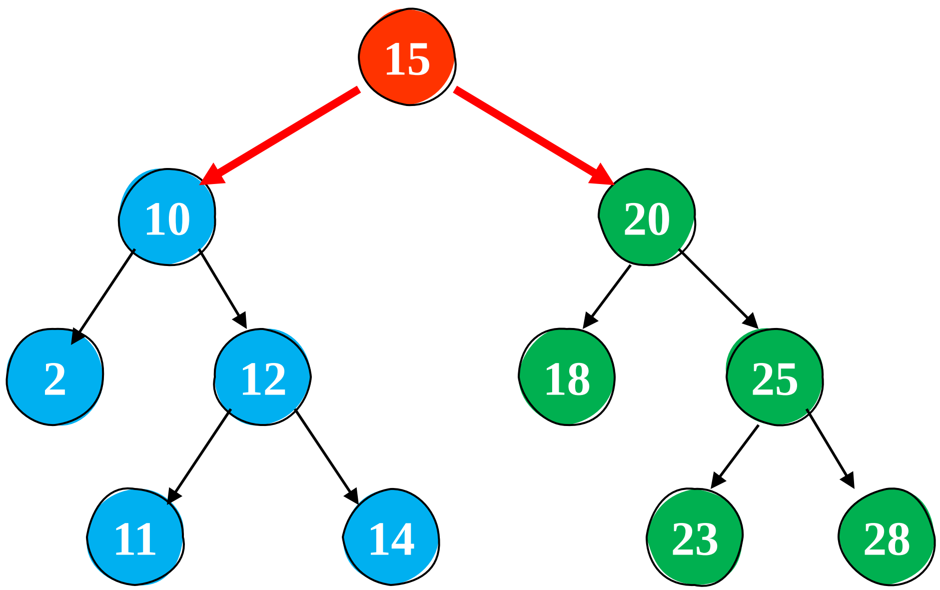
- What much time insert and deleteMax Operations will take?
  - *O*(log *N*) on average

BST as Priority Queue



- The only element we ever delete is the maximum
  - What issue can repeat deletion from right subtree create?
    - It can hurt the balance of the tree by making the left subtree heavy

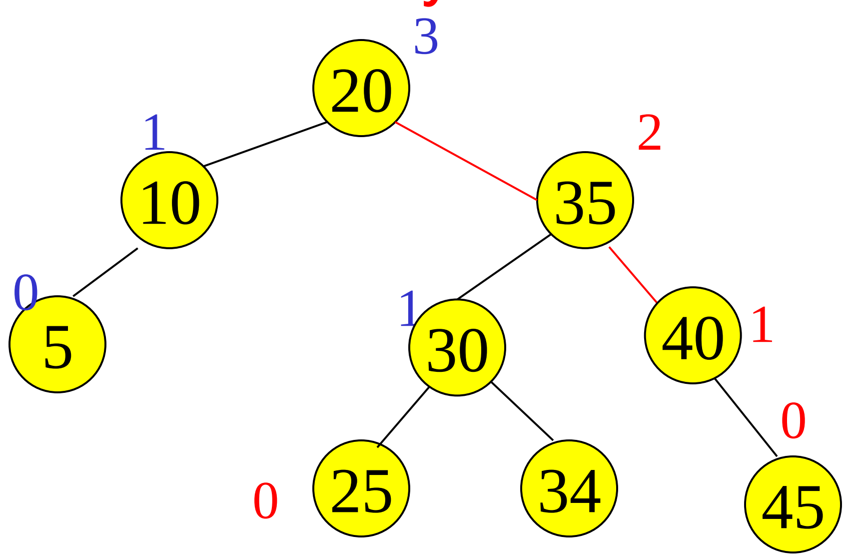
BST as Priority Queue



- In the worst case, where the deleteMaxs have depleted the right subtree,
  - the left subtree would have at most twice as many data as it should.

Balanced BST as Priority Queue

- Using a search tree could be overkill
  - because it supports a host of operations that are not required.
  - Search
  - Delete a particular key
  - And many other functionality



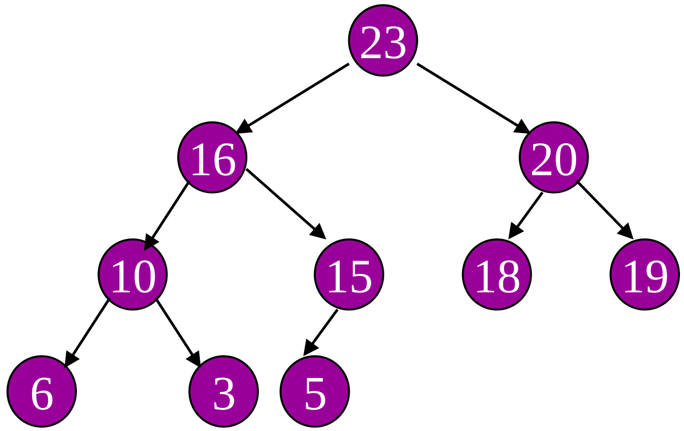
Another DS as Priority Queue

- We need another data structure
  - The basic data structure we will use will support both operations in **$O(\log N)$ worst-case time.**
  - **Insertion will take constant time on average**
  - Our implementation will build a priority queue of N items in linear time, if no deletions intervene
  - **BINARY HEAP**

Max Heap

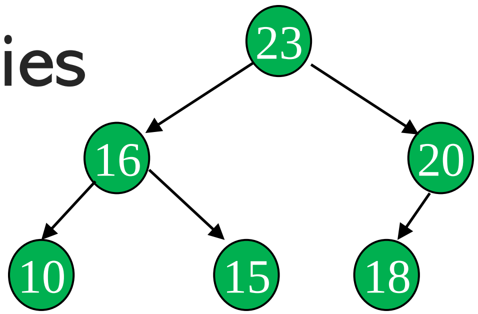
It is a binary tree with the following properties:

1. It is a complete binary tree.
2. The value stored in a node is >= to values stored in the children (heap-property)



BINARY HEAP'S Property

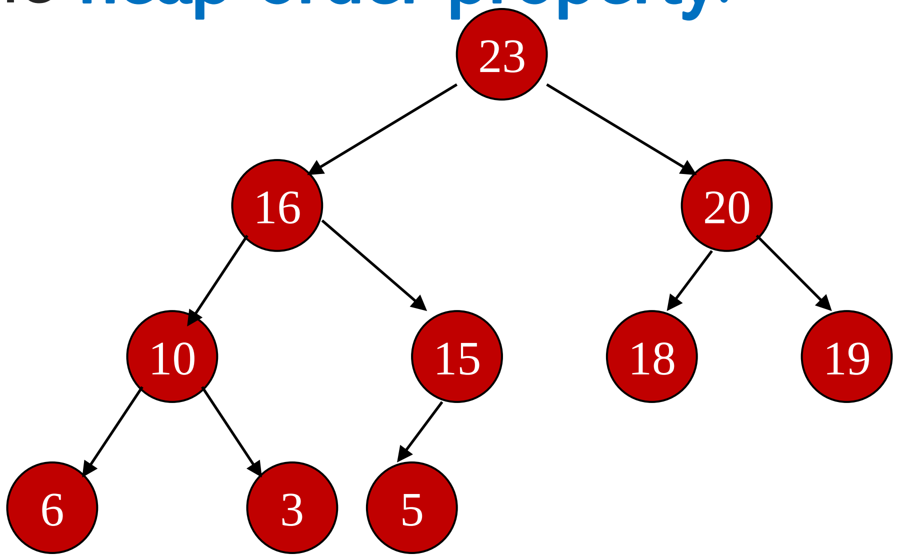
- Like BST, heaps have two properties
  - a structure property and
  - a heap order property.



- As with AVL trees, an operation on a heap can destroy one of the properties
  - a heap operation must not terminate until all heap properties are in order.
- This turns out to be simple to do.

WHY Heap-Order Property

- The property that allows operations to be performed quickly is the **heap-order property.**

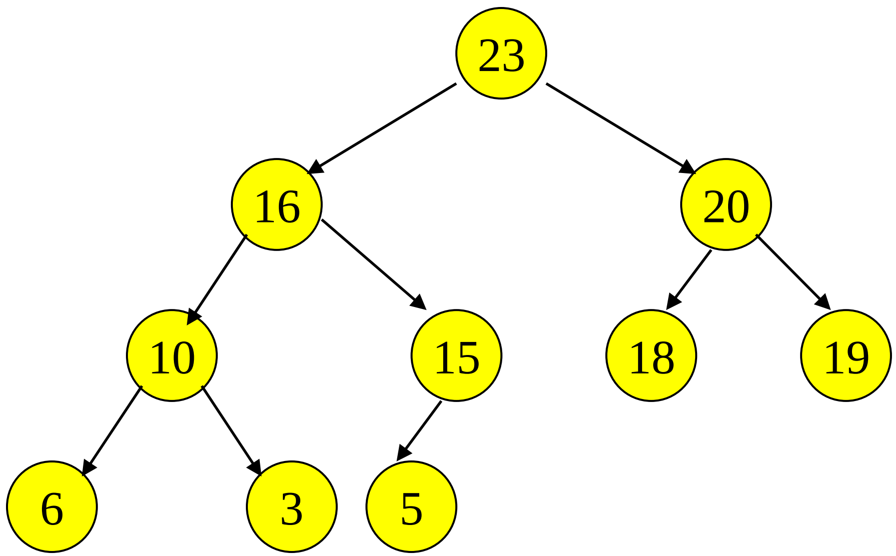


- We want to be able to find the maximum (or minimum) quickly

➢ So, it makes sense that the largest (or smallest) element should be at the root

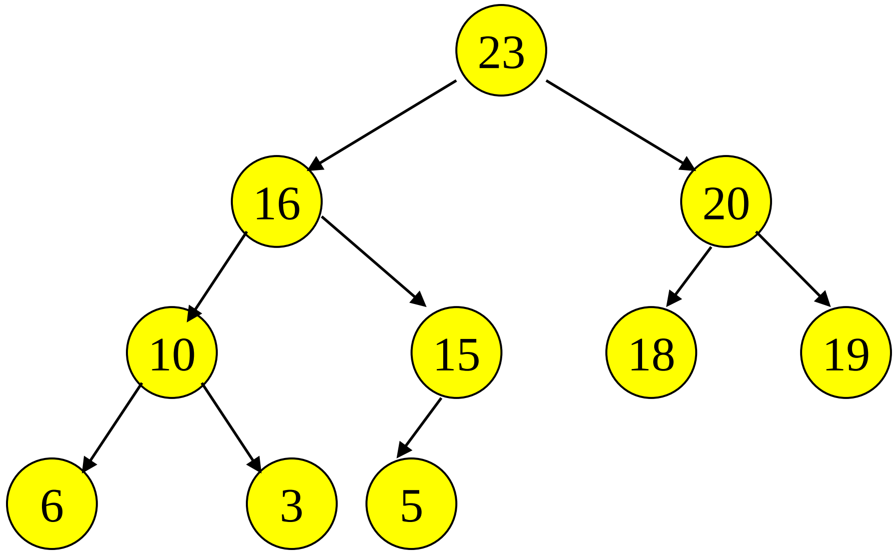
WHY Heap-Order Property

- Any subtree in a heap should also be a **heap,**
  - Every node should be larger than its descendants



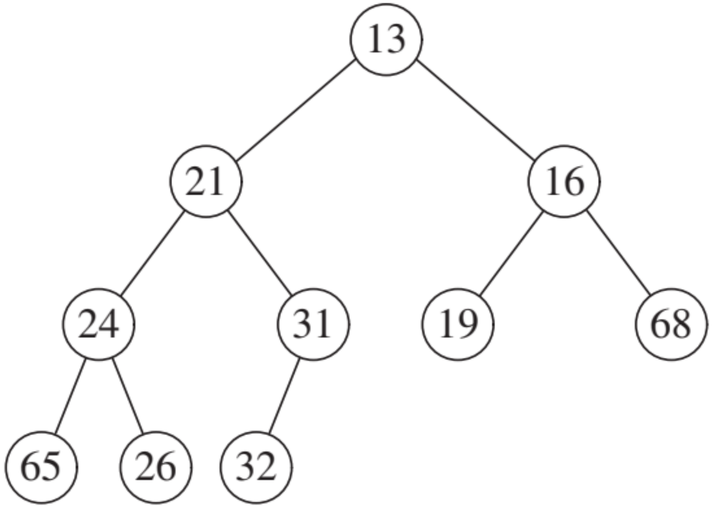
MAX Heap

- In a MAX heap, for every node *X*,
  - key(parent of *X*) $\geq$ key(*X*)
  - except for the root (which has no parent)

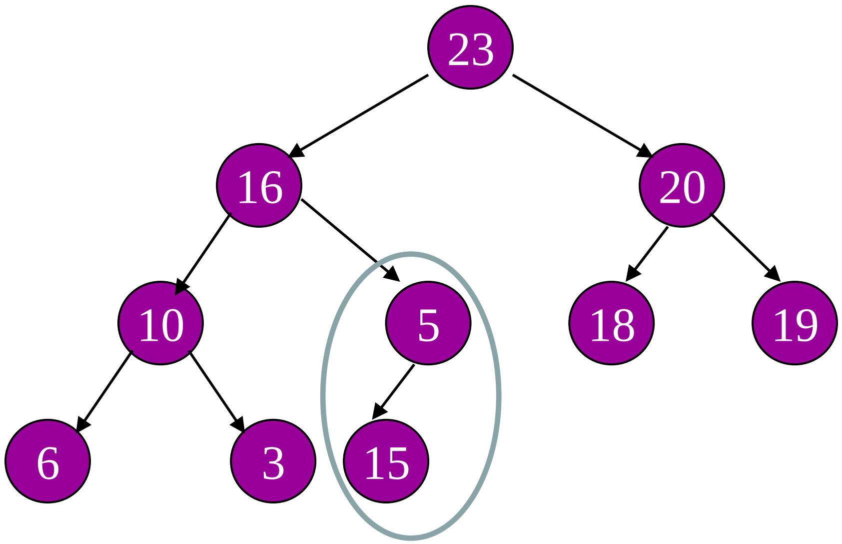


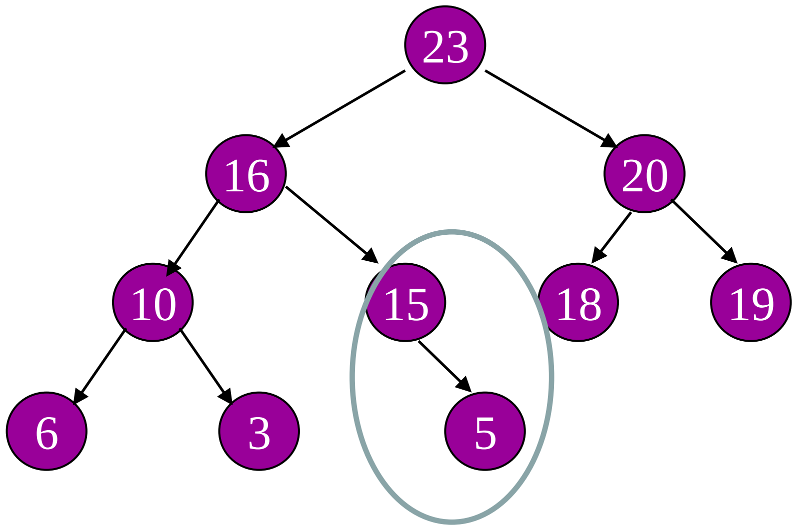
MIN Heap

- In a MIN heap, for every node *X*,
  - key(parent of *X*) ≤key(*X*)
  - except for the root (which has no parent)



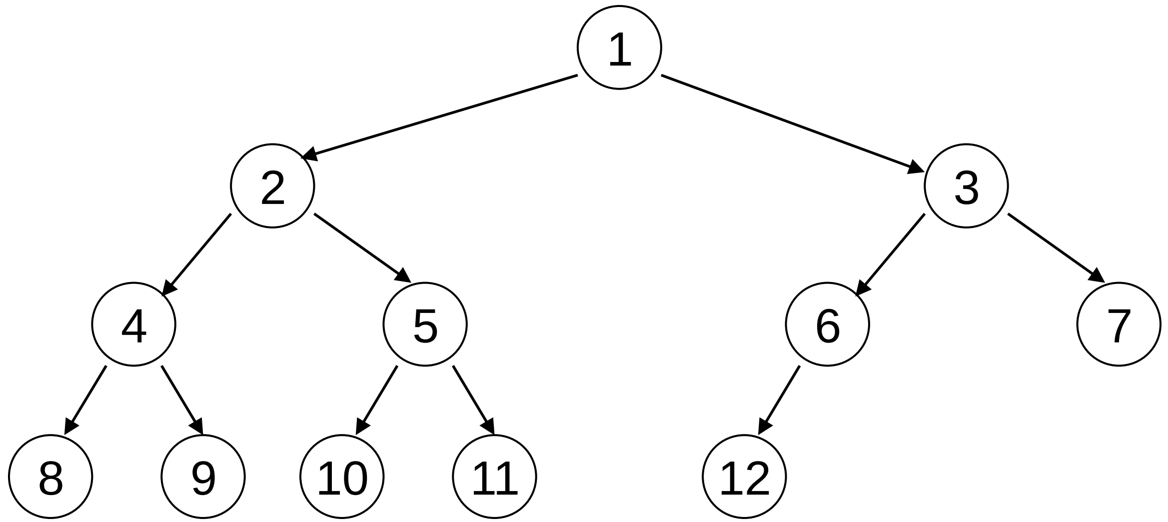
Is it a Max heap?





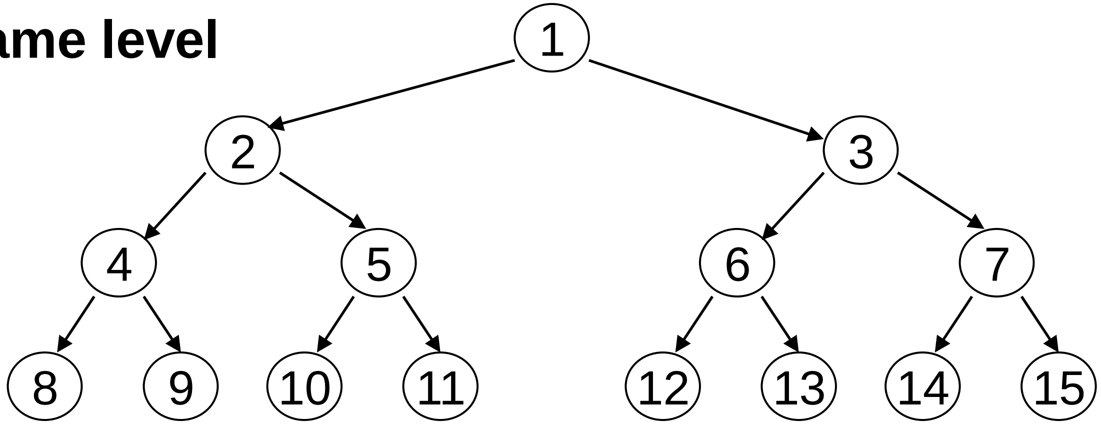
Complete Binary Tree

A binary tree that is completely filled except the last level, which is filled from left to right, is called a *complete* binary tree



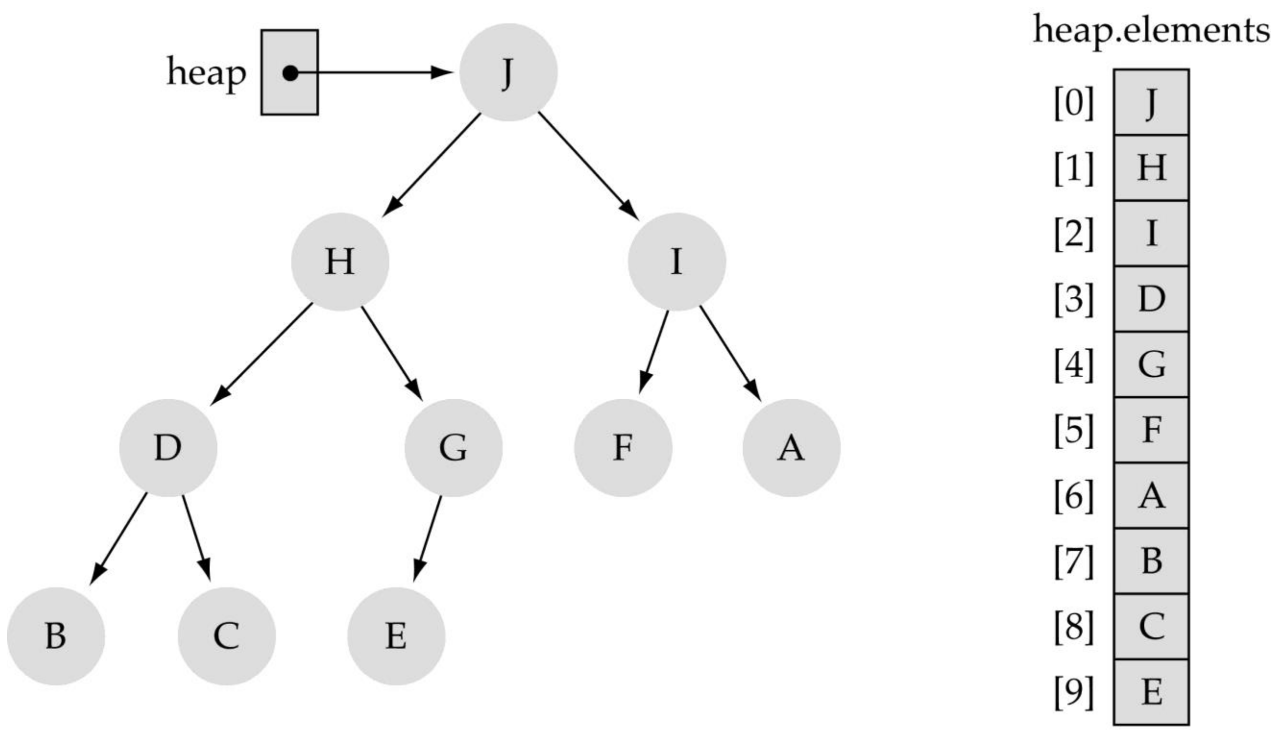
Perfect Binary Tree

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



Heap implementation using array

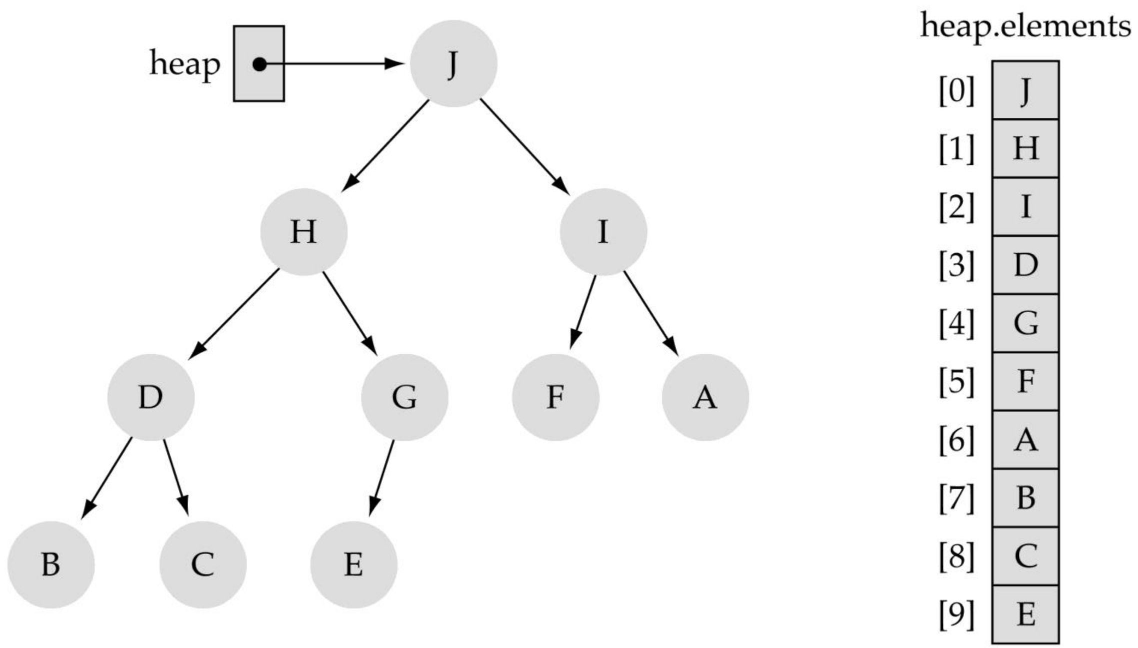
A heap is a **complete binary tree**, so it is easy to be implemented using an array representation



The data occupy contiguous array slots

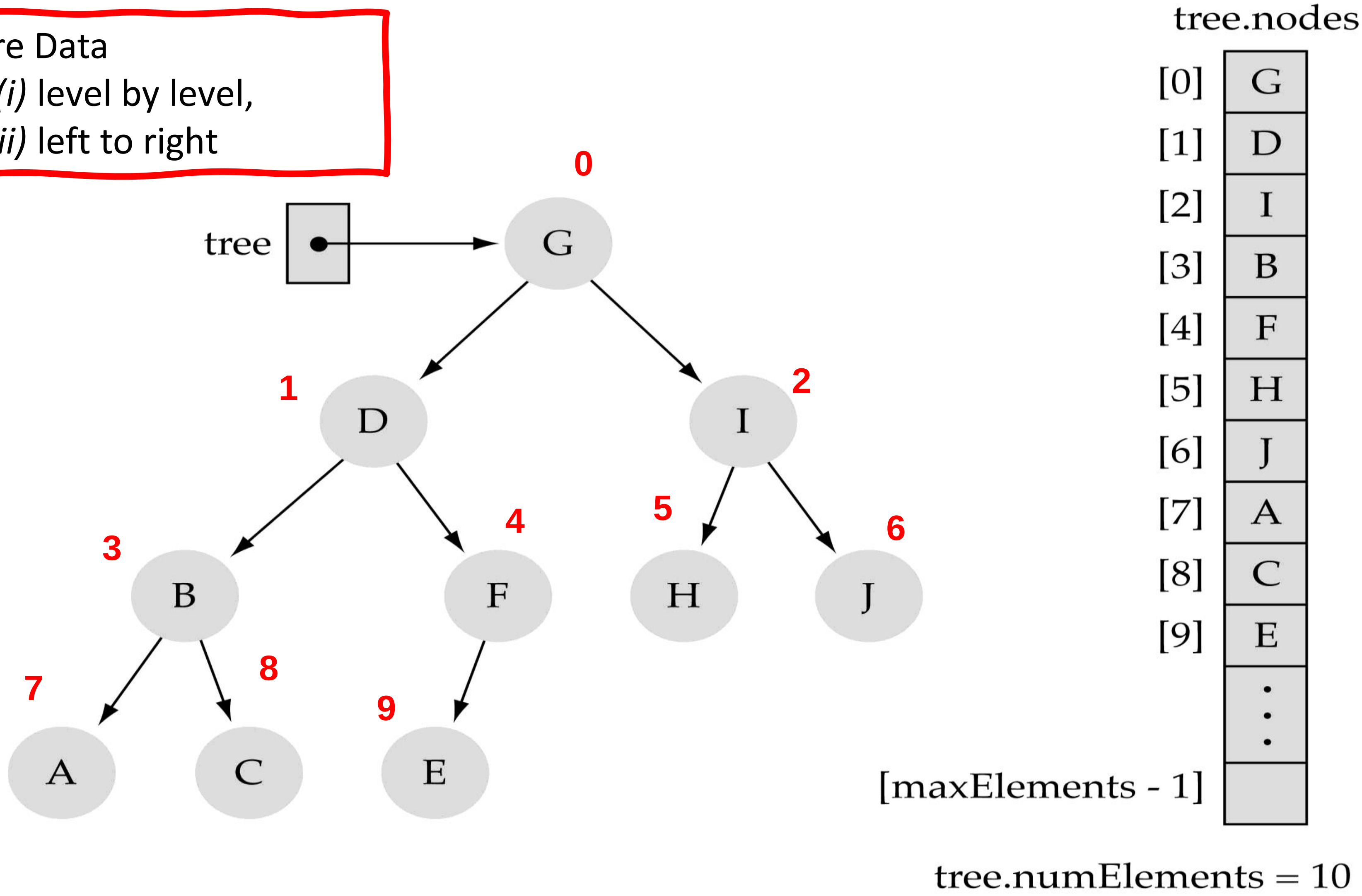
Heap implementation using array

- Memory space can be saved (no pointers are required)
- Preserve parent-child relationships by storing the tree data in the array
  - *(i)* **level by level,**
  - *(ii)* **left to right**

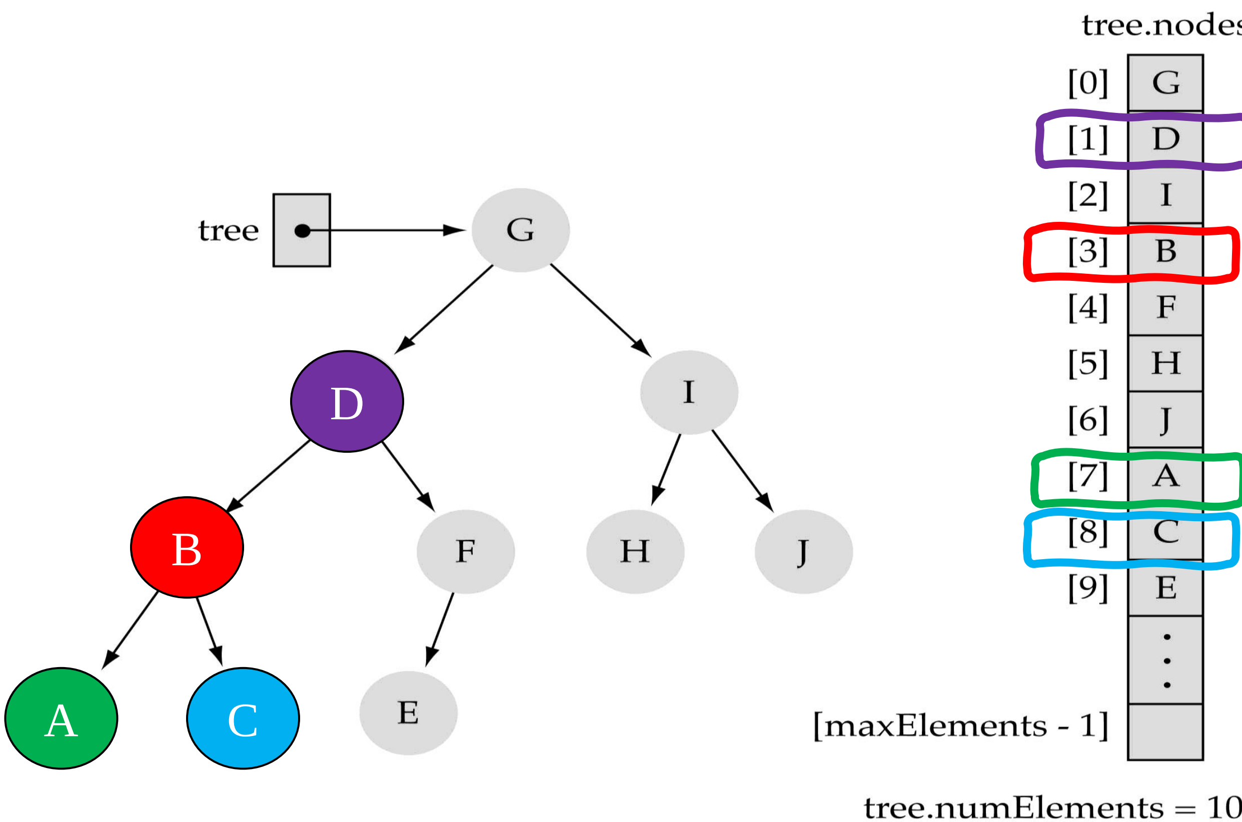


Heap implementation using array

Store Data
(i) level by level,
(ii) left to right



Heap using array– some Properties



B is at index 3

Left child:

*nodes[2*index+1]*

2*3 +1 = 7

Right child:

*nodes[2*index+2]*

2*3 +2 = 8

Parent:

nodes[(index-1)/2]

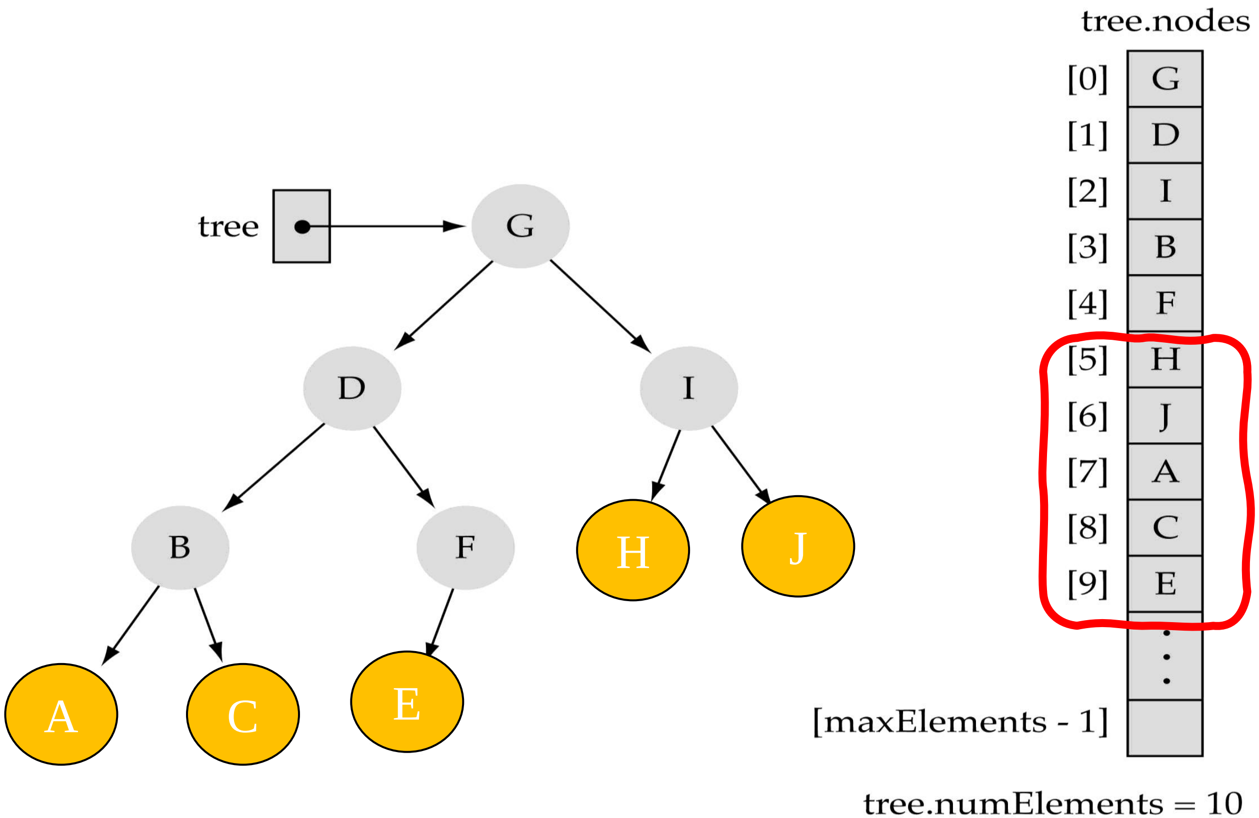
(3-1)/2= 1

Leaf nodes of tree: H, J, A, C, E

Between index 5 & 9

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

Heap using array– some Properties



Leaf nodes:
nodes[N/2]
to
nodes[N - 1]

Leaf nodes of tree: H, J, A, C, E
Between index 5 & 9
Between (10/2) & 9 => *5 & 9*

Heap implementation using array

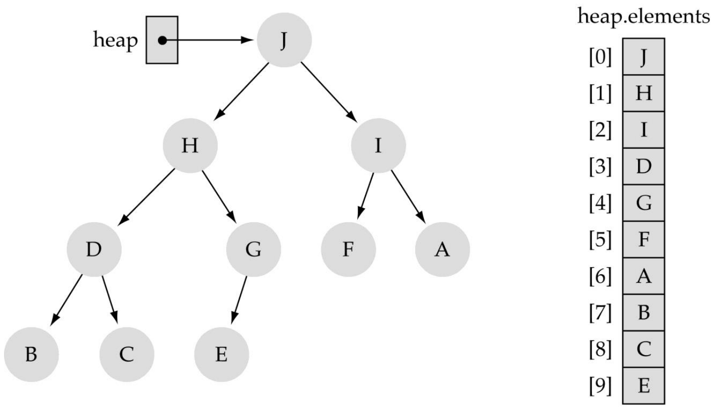
- **Parent-child relationships:**
  - left child of *tree.nodes[index]*

    *= 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)

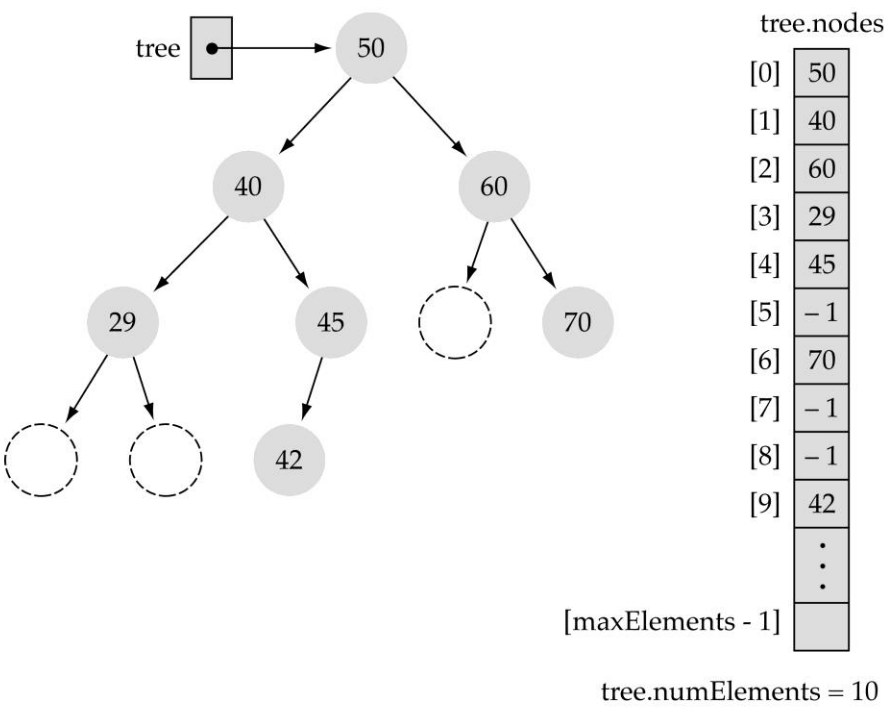


- **Leaf nodes:**
  - *Exist between*

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

Array-based representation of binary trees

- **Note for binary trees that are not complete Array is not a good representation**
- "Dummy nodes" are required for trees which are not full or complete



Binary Heap

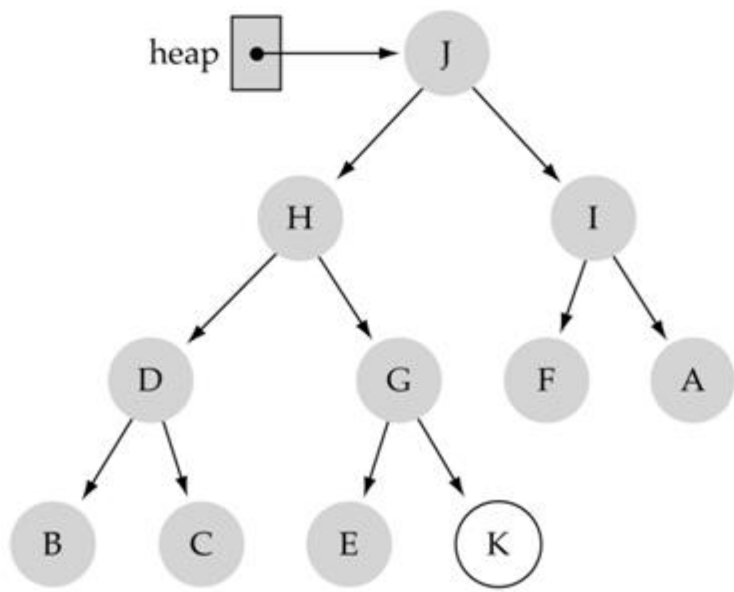
```
template <typename T>
class BinaryHeap{
    public:
        void BinaryHeap(int capacity = 100);
        bool isEmpty() const;
        const T & findMax() const;
        void insert(const T & x);
        void deleteMax(T & maxItem);
        void makeEmpty();

   private:
        int currentSize; // Number of data in heap
        vector<T> data; // The heap array
        int capacity;
        void buildHeap();
        void ReheapDown(int hole);
        void ReheapUp(int hole);

};
```

Insert X in the heap

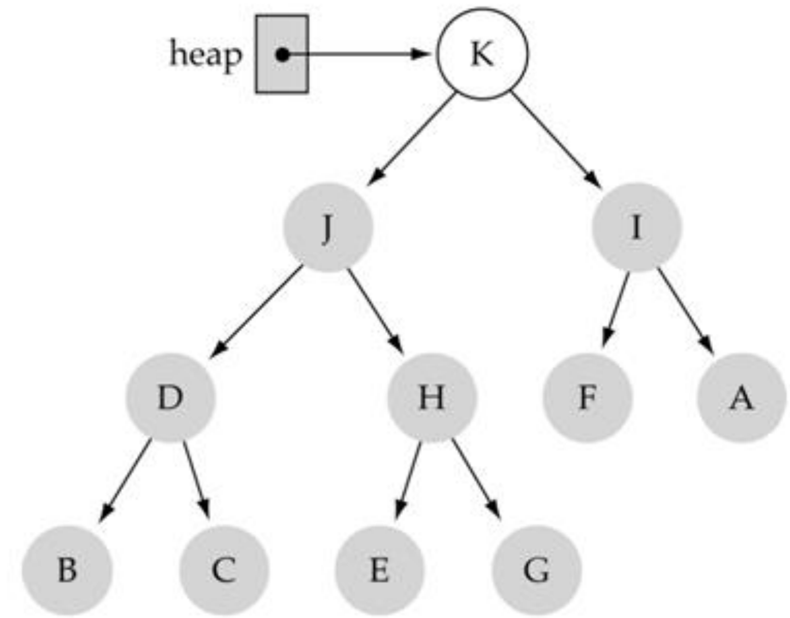
1) Insert the new element at the end of the heap
2) Fix the heap property by calling *ReheapUp*

```
PseudoCode

heapEnqueue(el)
    put el  at the end of heap;
    // ReheapUP
    while el  is not in the root and el  >  parent(el)
        swap el  with its parent;
```

Inserting a new element into the heap

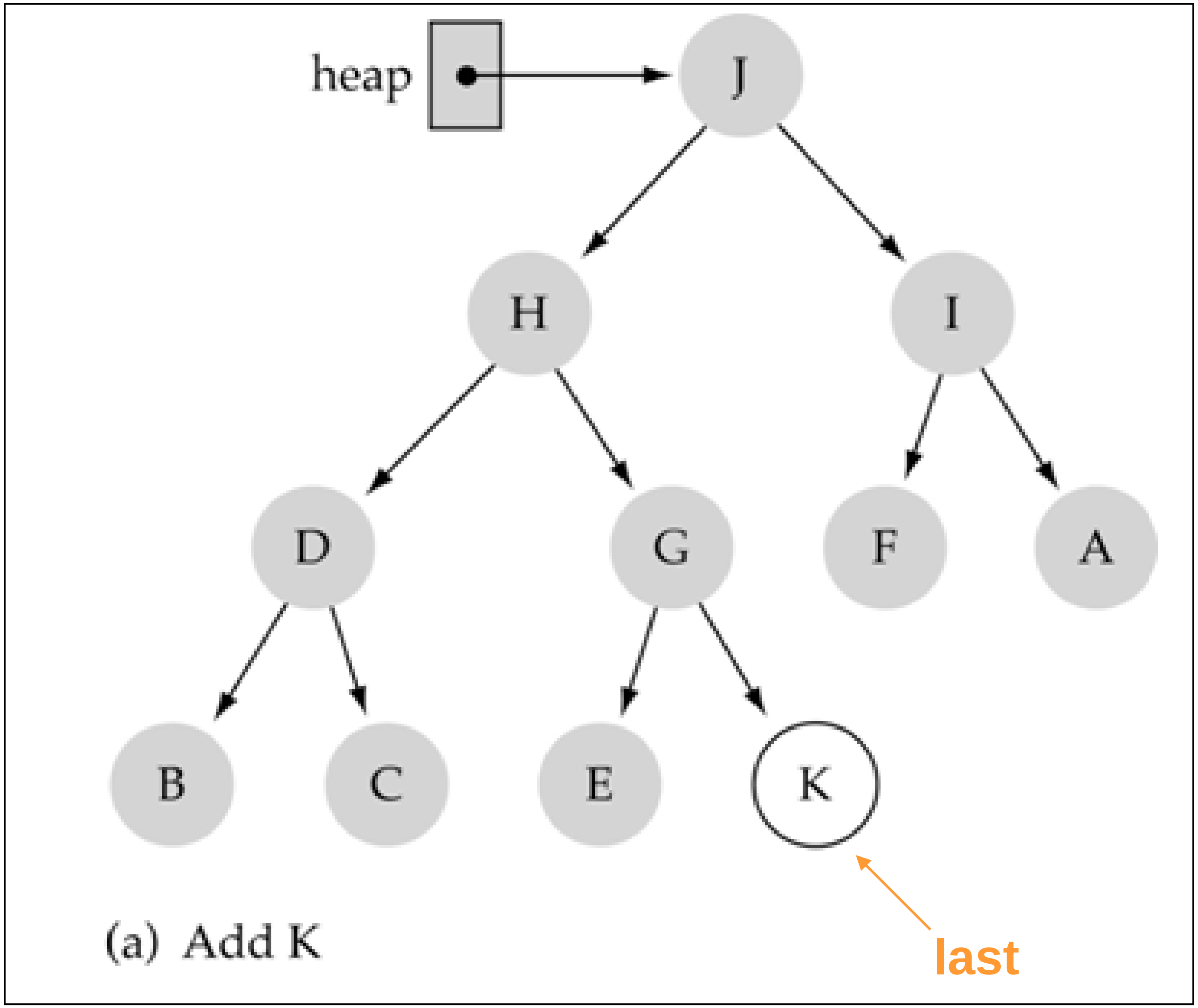


(a) Add K

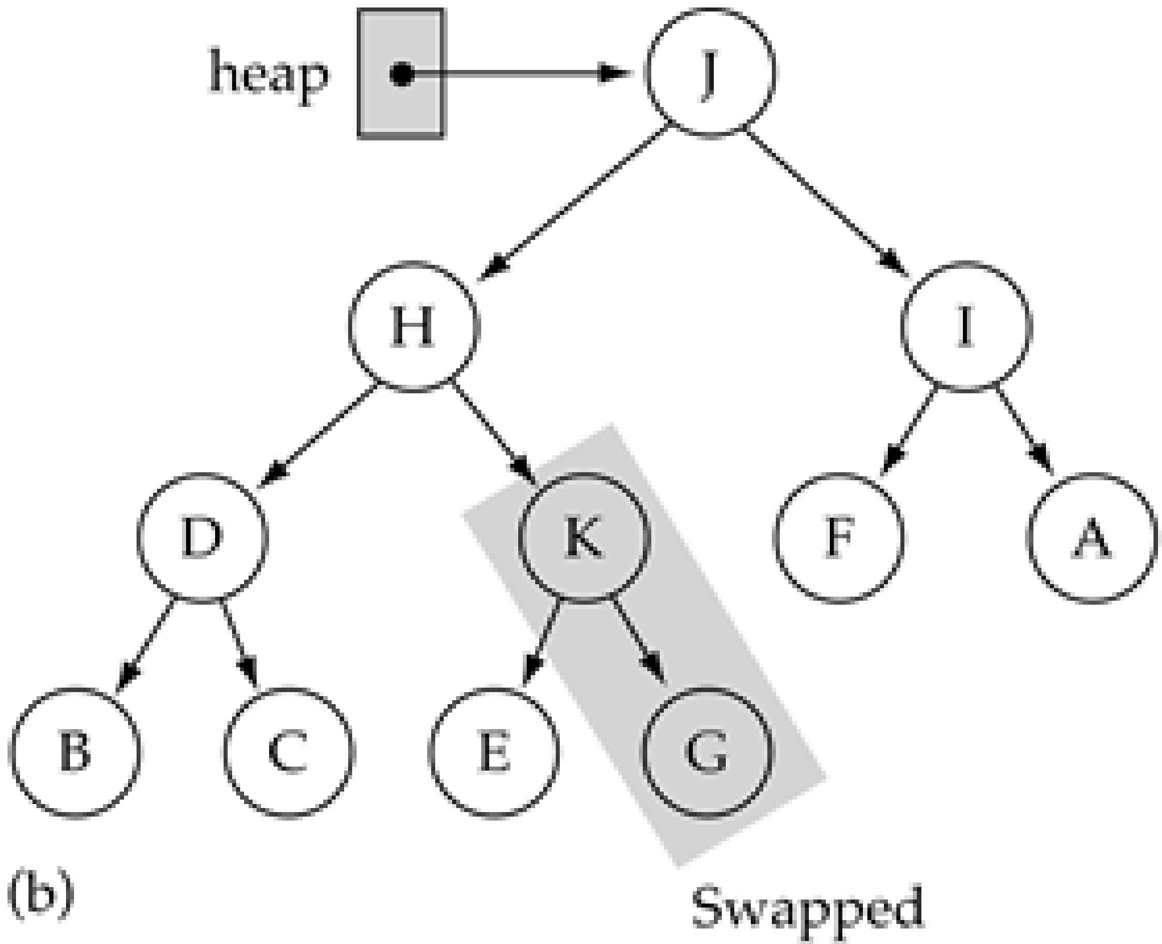


(b) ReheapUp

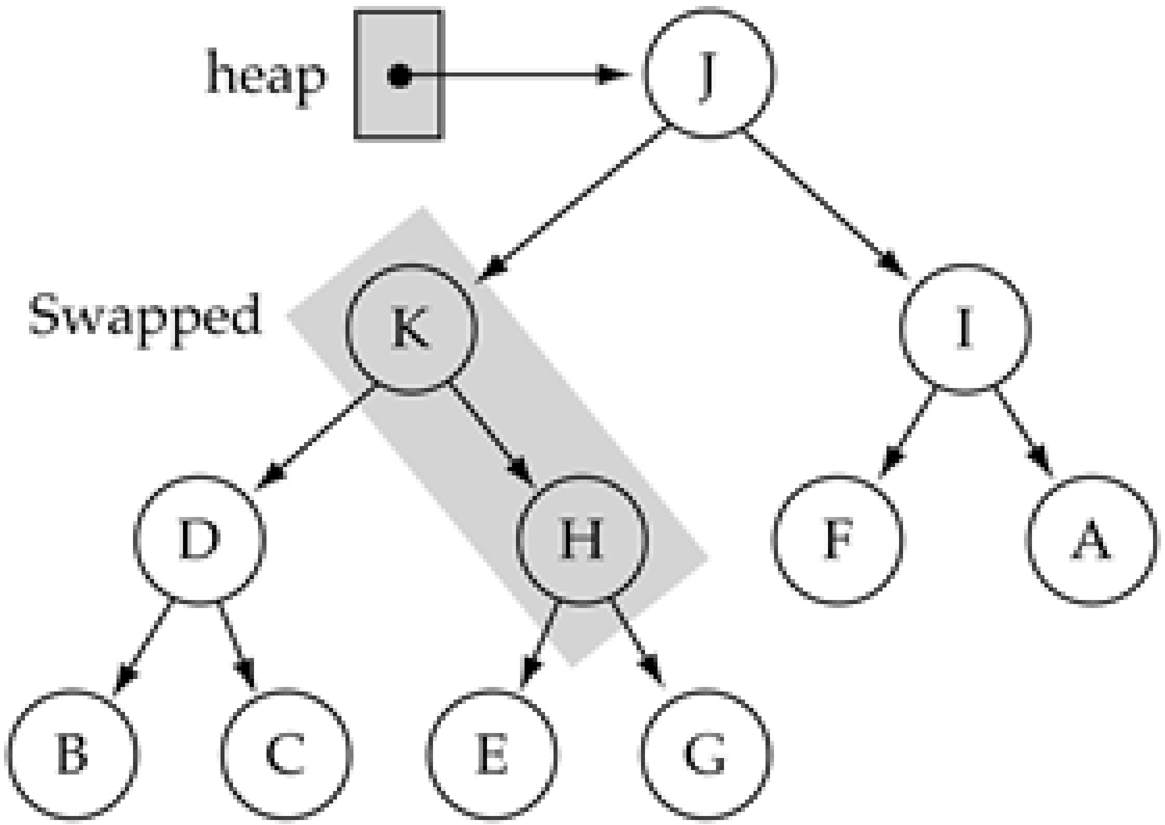
Inserting a new element into the heap



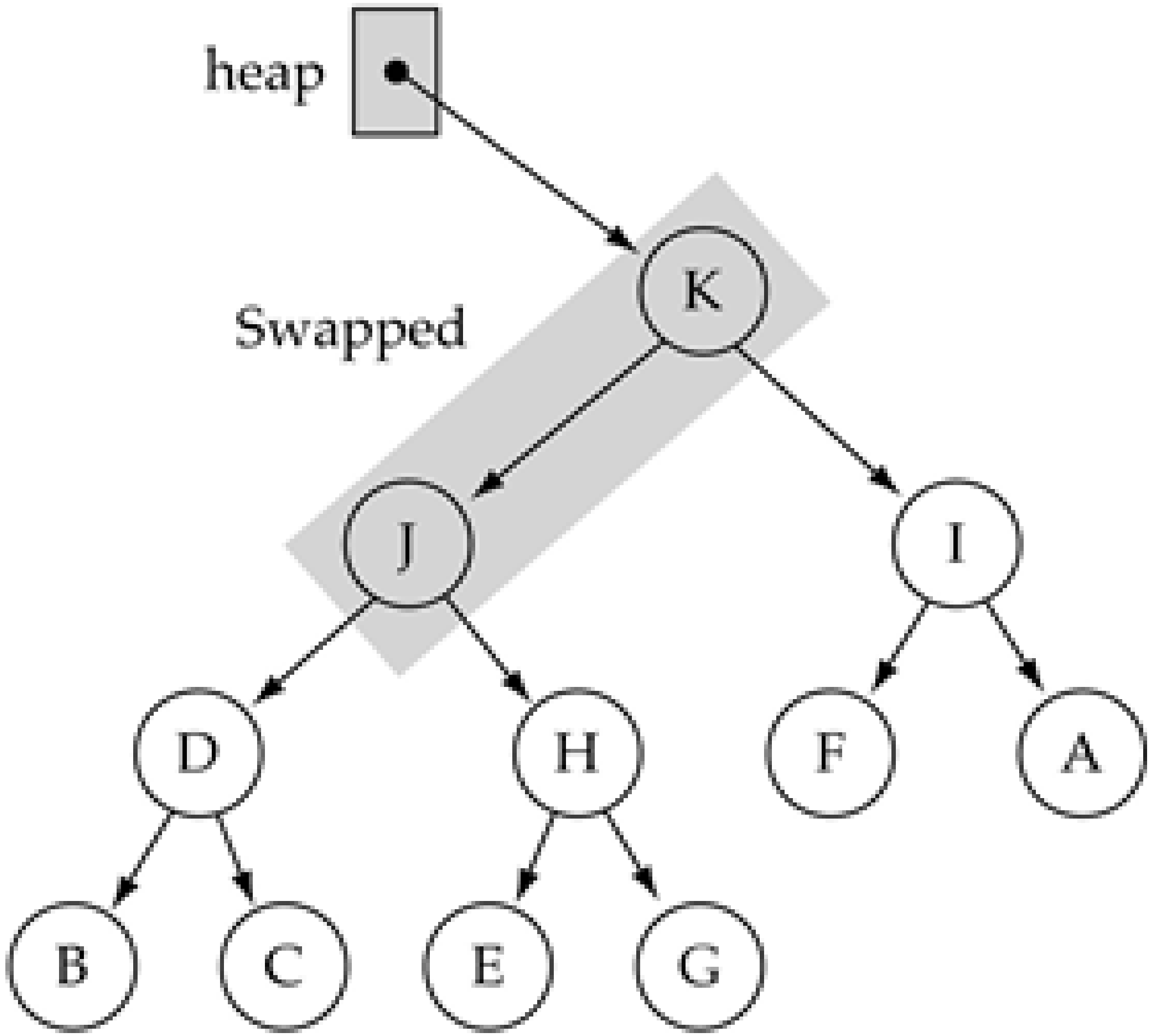
The ReheapUp function
(used by insertItem)



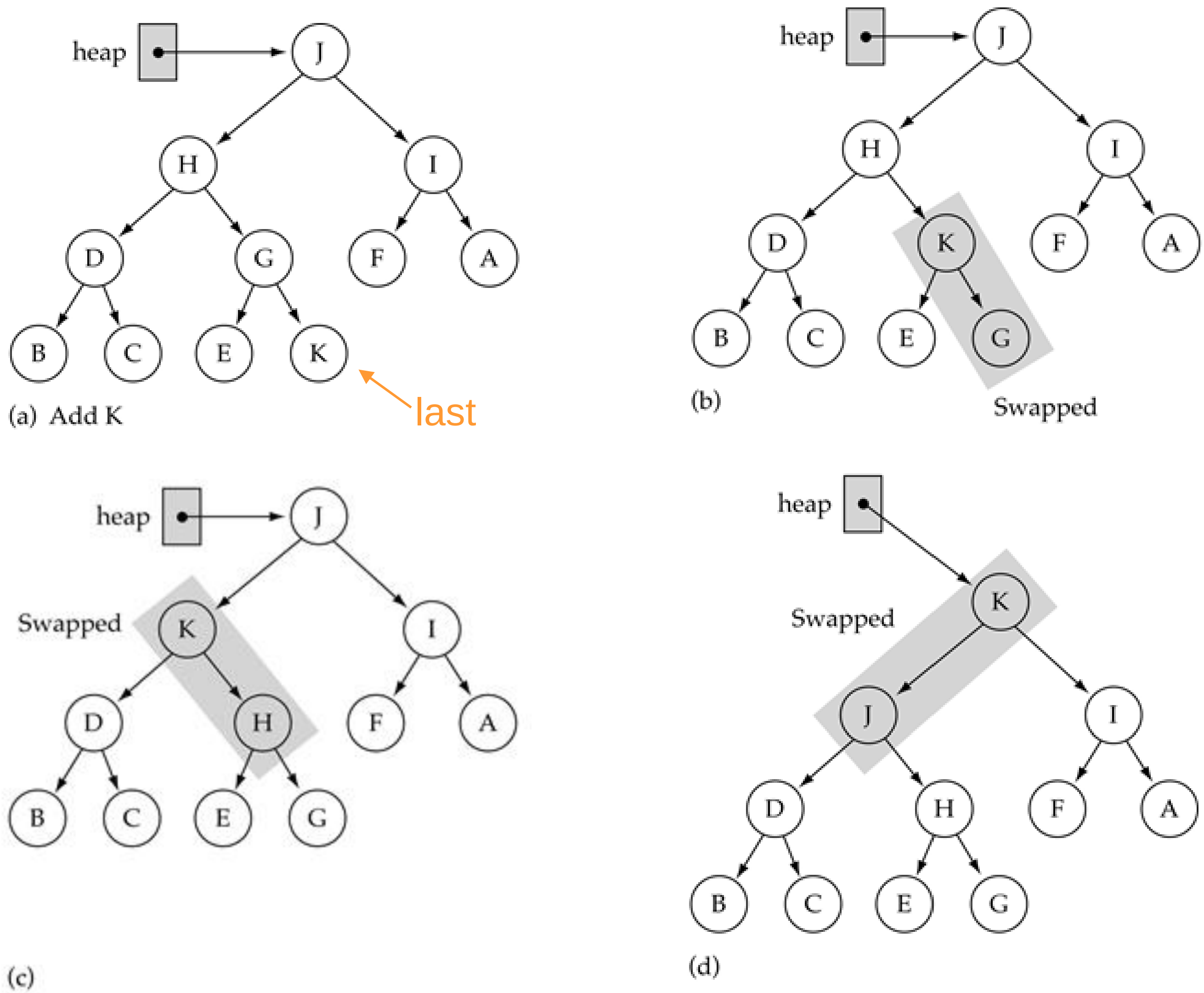
The ReheapUp function (used by insertItem)



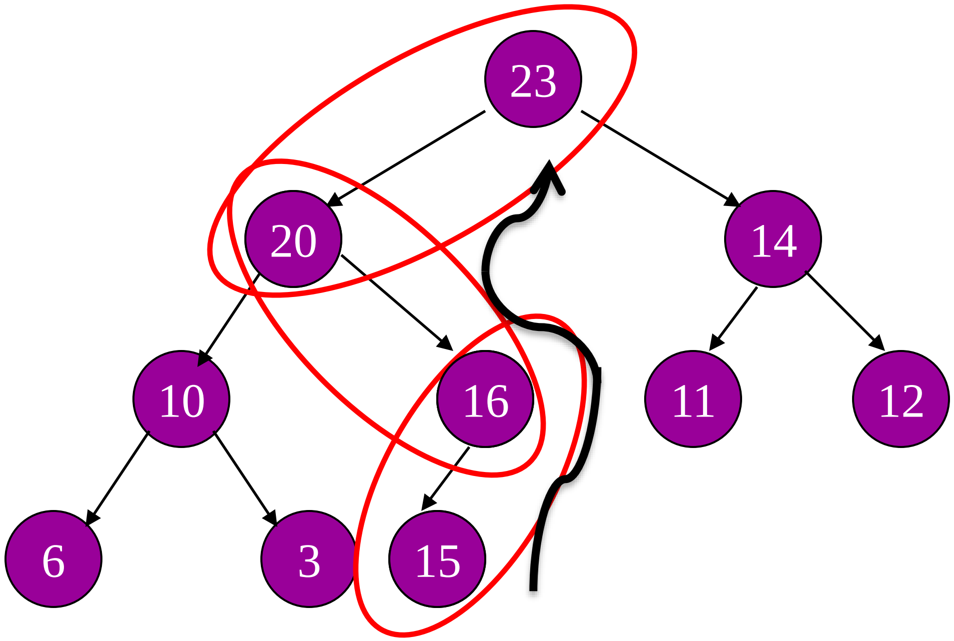
The ReheapUp function (used by insertItem)



The ReheapUp function (used by insertItem)



The ReheapUp function
(used by insertItem)

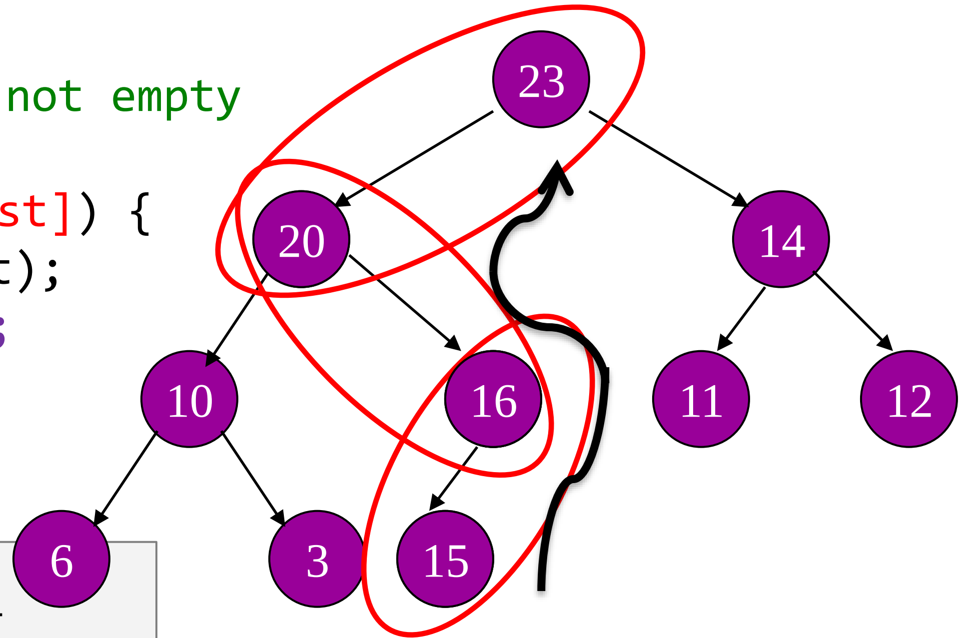


Assumption:

Heap property is violated at the rightmost node at the last level of the tree

Insert & Recursive ReheapUP

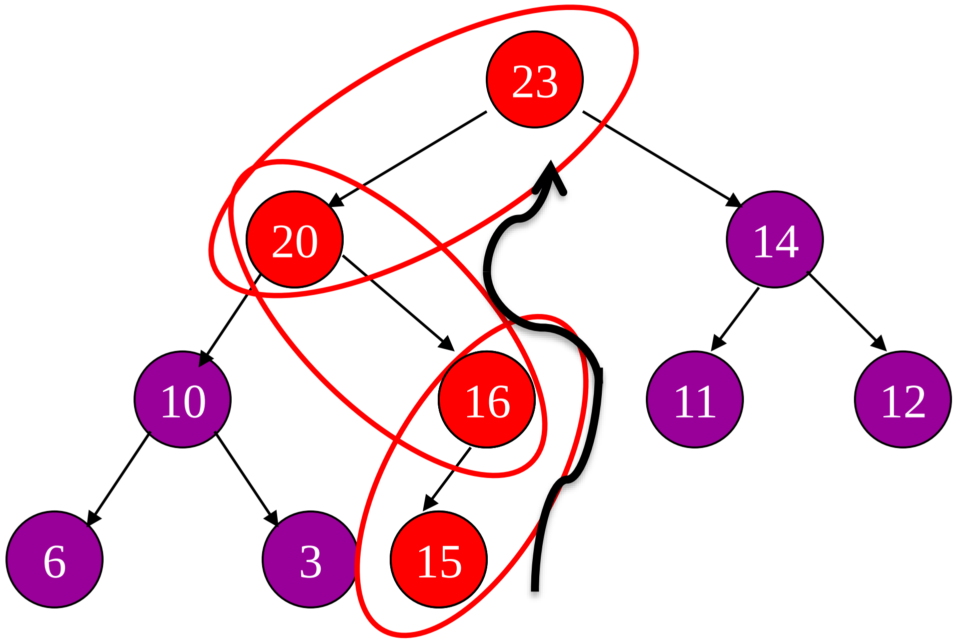
```
template<class T>
void BinaryHeap<T>::ReheapUp(int root, int last)
{
    int parent;
    if (last > root) { // tree is not empty
        parent = (last - 1) / 2;
        if (data[parent] < data[last]) {
            Swap(data, parent, last);
            ReheapUp(root, parent);
        }
    }
}
```



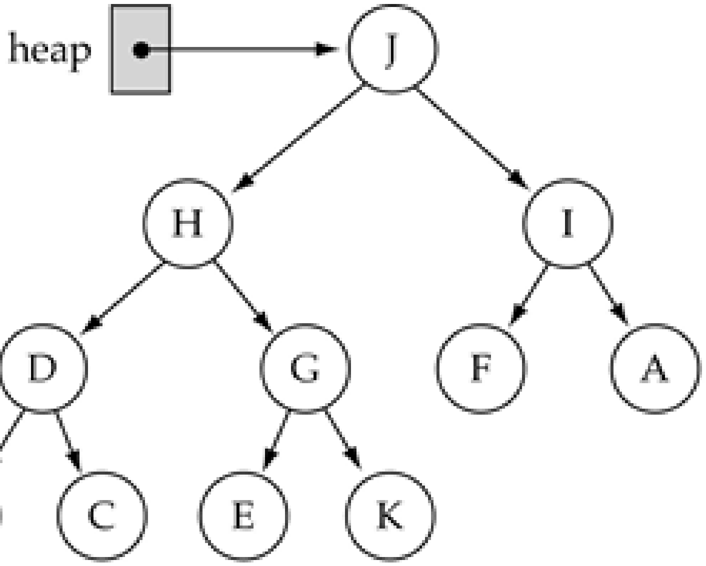

```
template<class T>
void BinaryHeap<T>::Insert(T newItem){
    if(currentsize < capacity){
        currentSize++;
        data[currentSize - 1] = newItem;
        ReheapUp(0, currentSize - 1]);
    }
}
```

Insert Iterative

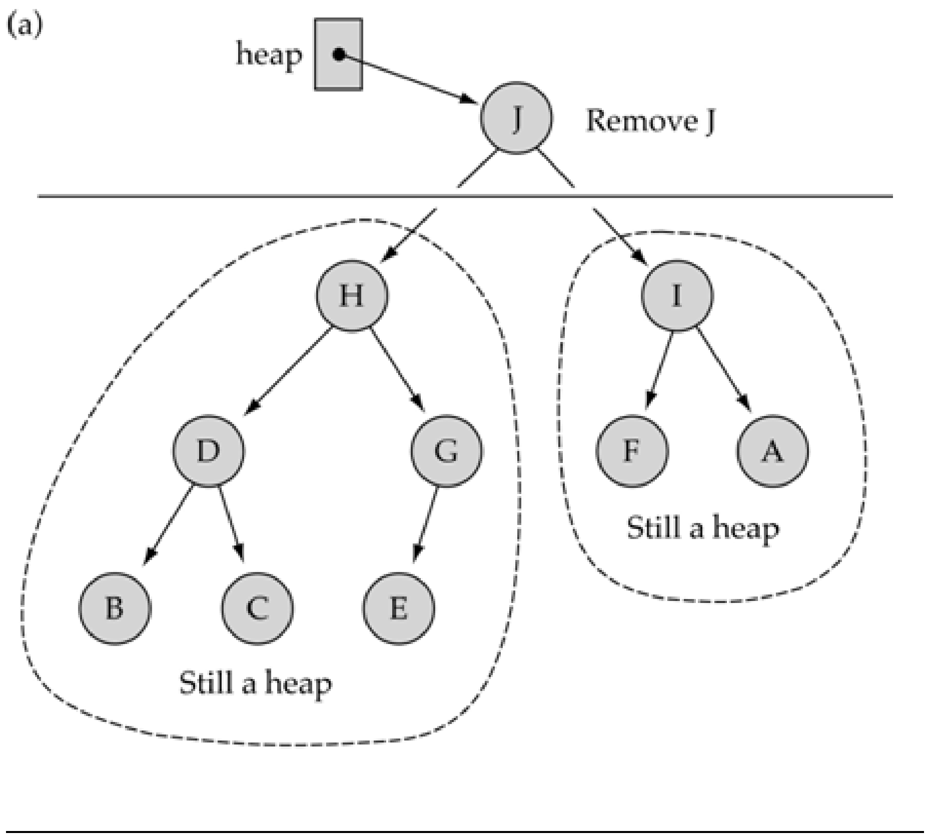
```
void insert(const T & x){
    if(currentsize < capacity){
        // ReheapUP
        int hole = ++currentSize;
        for (; x > data[hole / 2] && hole>=0; hole /= 2)
            data[hole] = data[hole / 2];
        data[hole] = x; // assumption =operator overloaded
    }
}
```

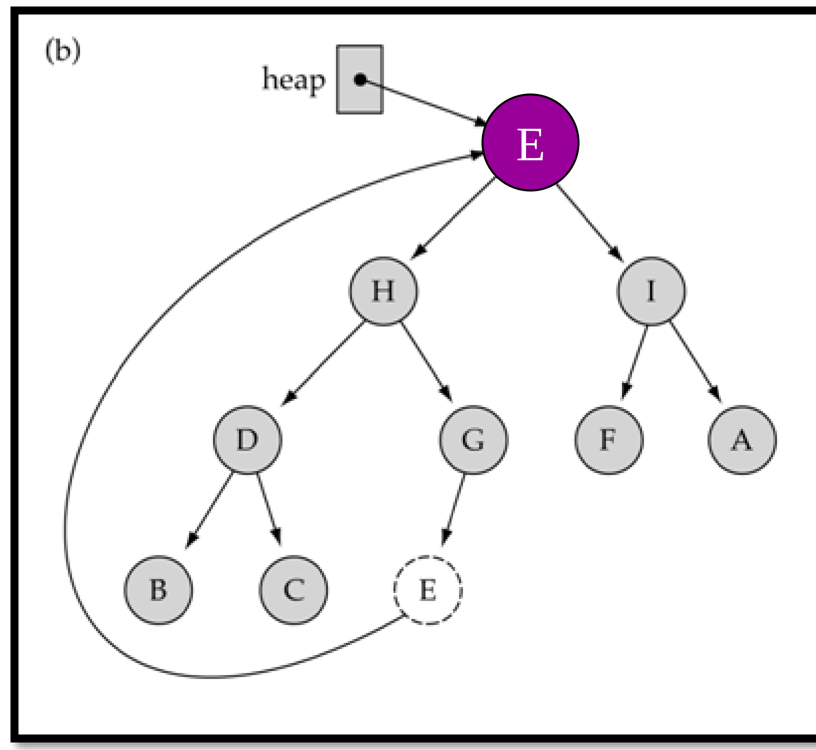


Removing the largest element from the heap

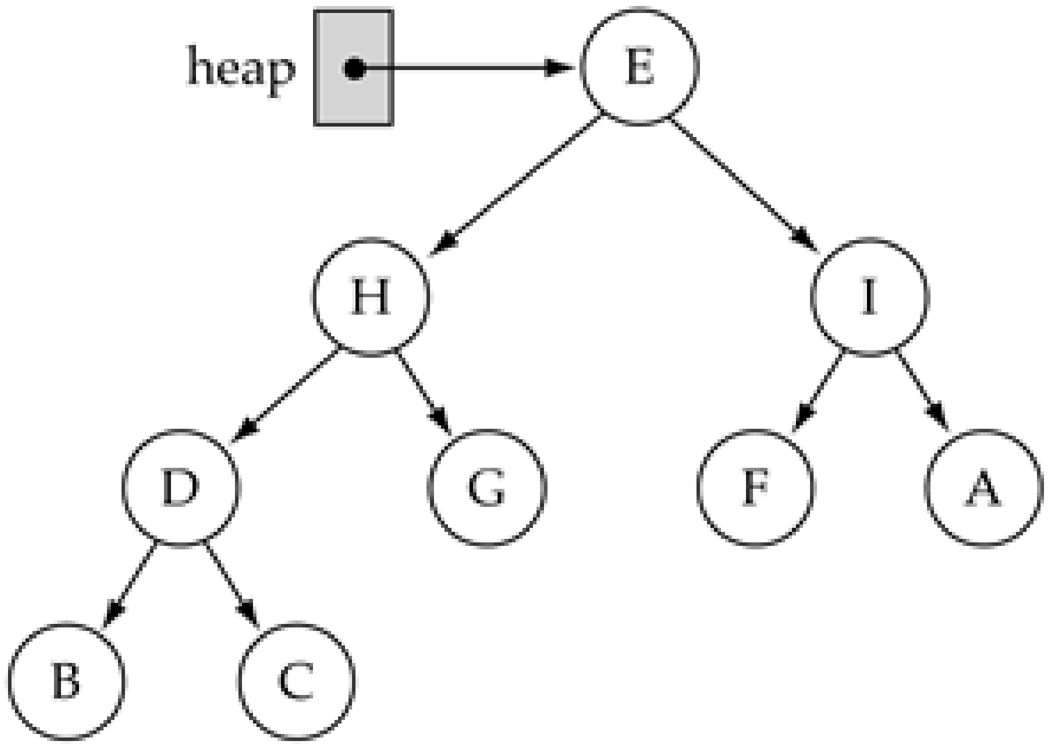


Removing the largest element from the heap



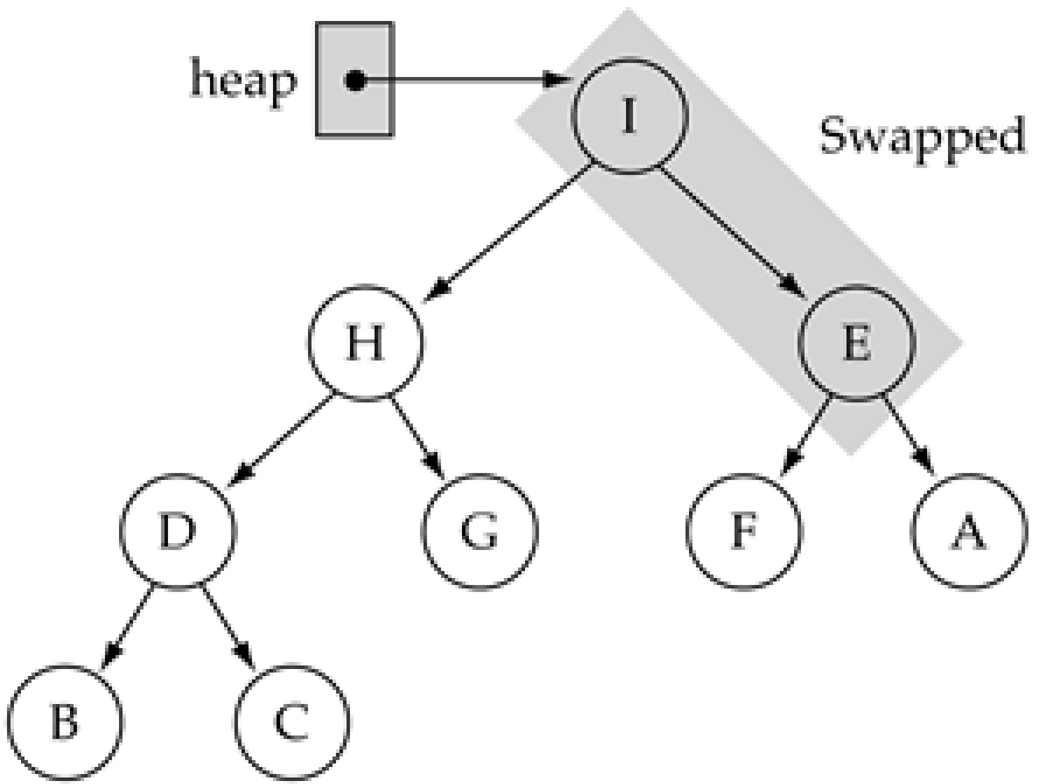


The ReheapDown function



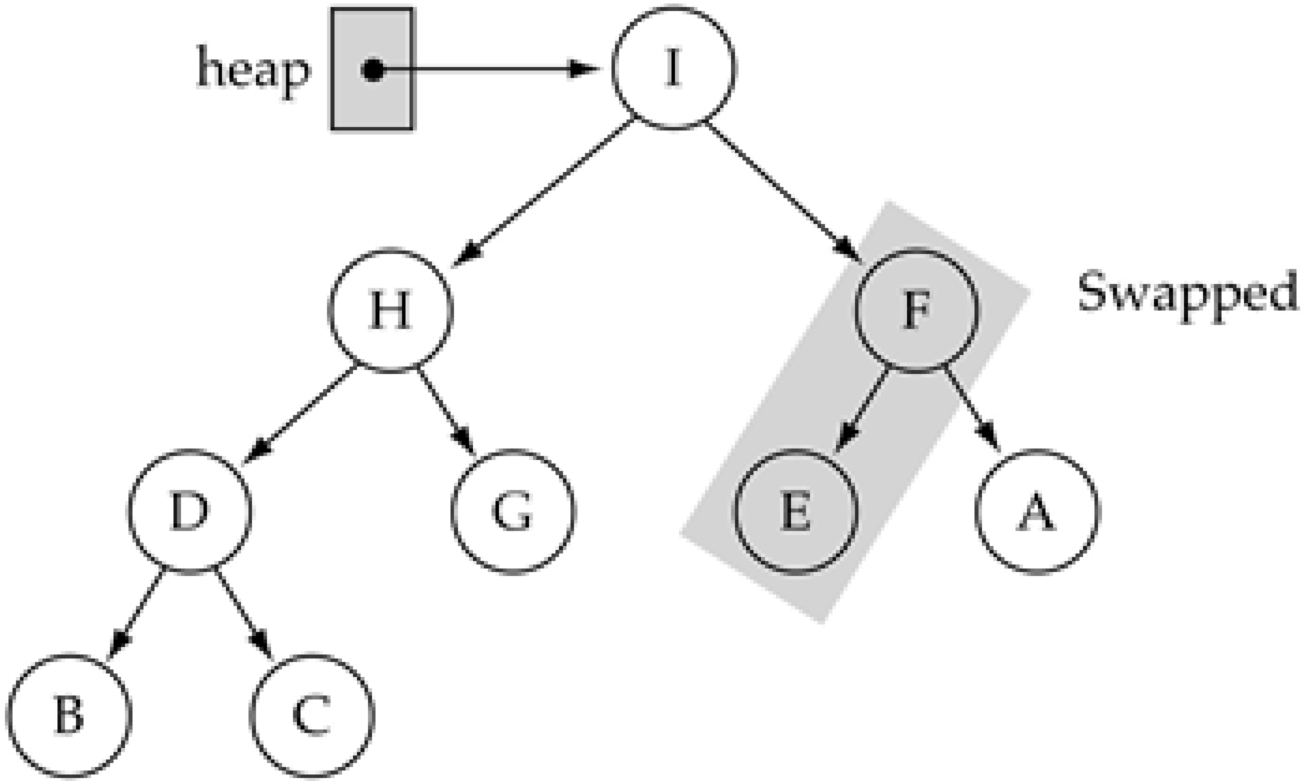
(a)

The ReheapDown function



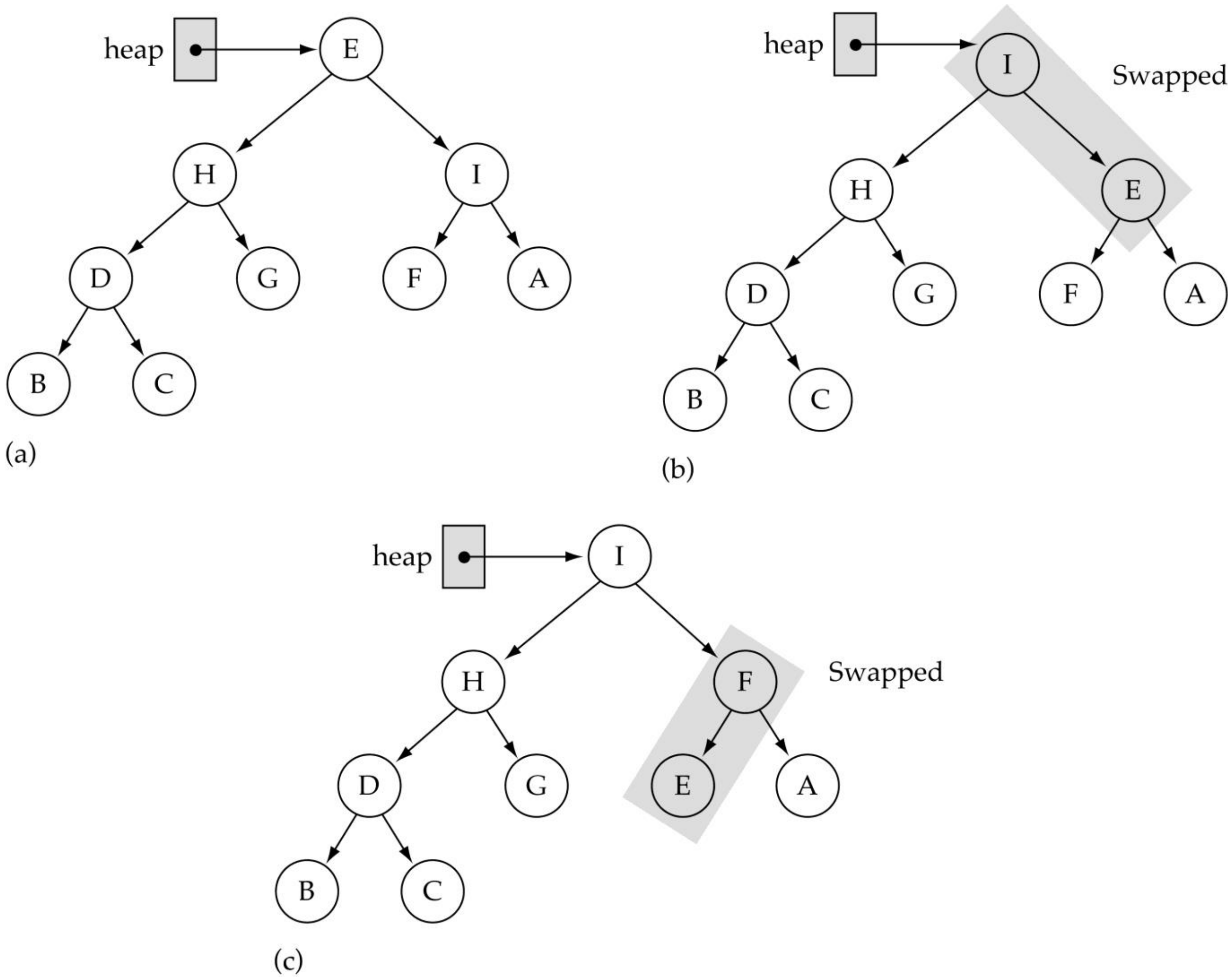
(b)

The ReheapDown function



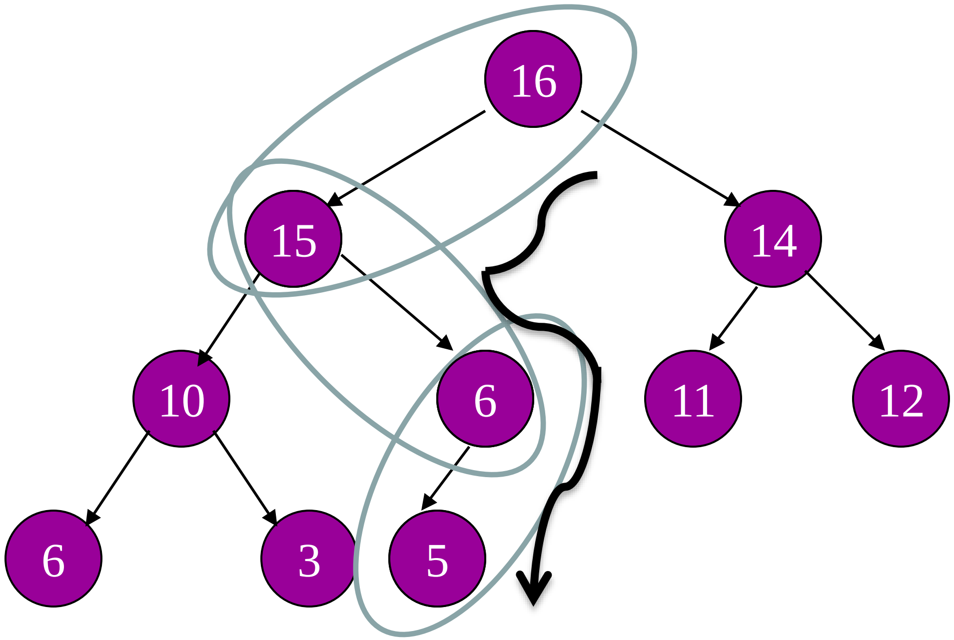
(c)

The ReheapDown function (used by deleteItem)



heap property is violated at the root of the tree

The ReheapDown function (used by deleteItem)



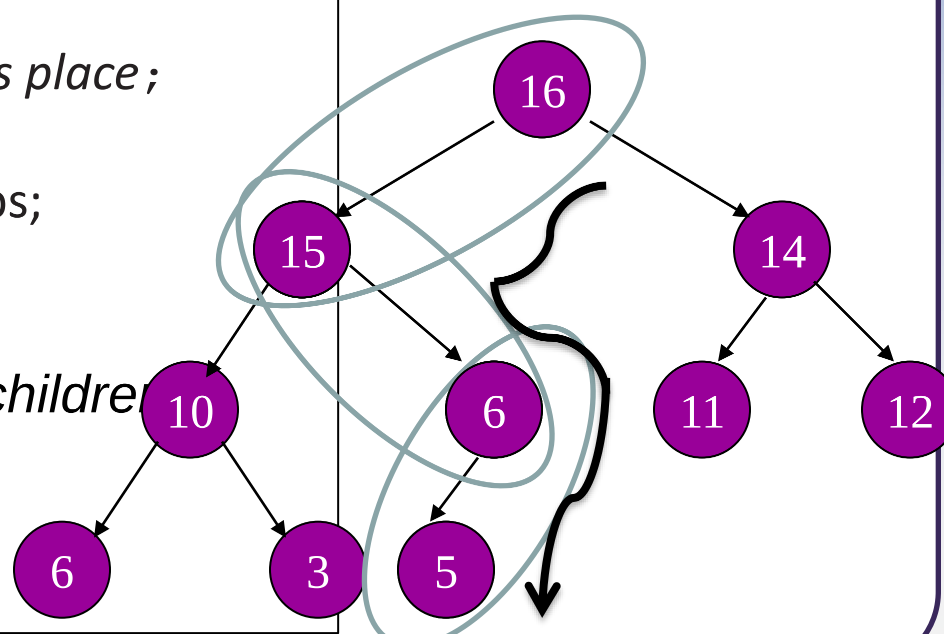
Assumption:
heap property is violated at the root of the tree

DeleteMax

1) **Copy the last rightmost element to the root**
2) **Delete the last rightmost node**
3) **Fix the heap property by calling *ReheapDown***

heapDequeue()

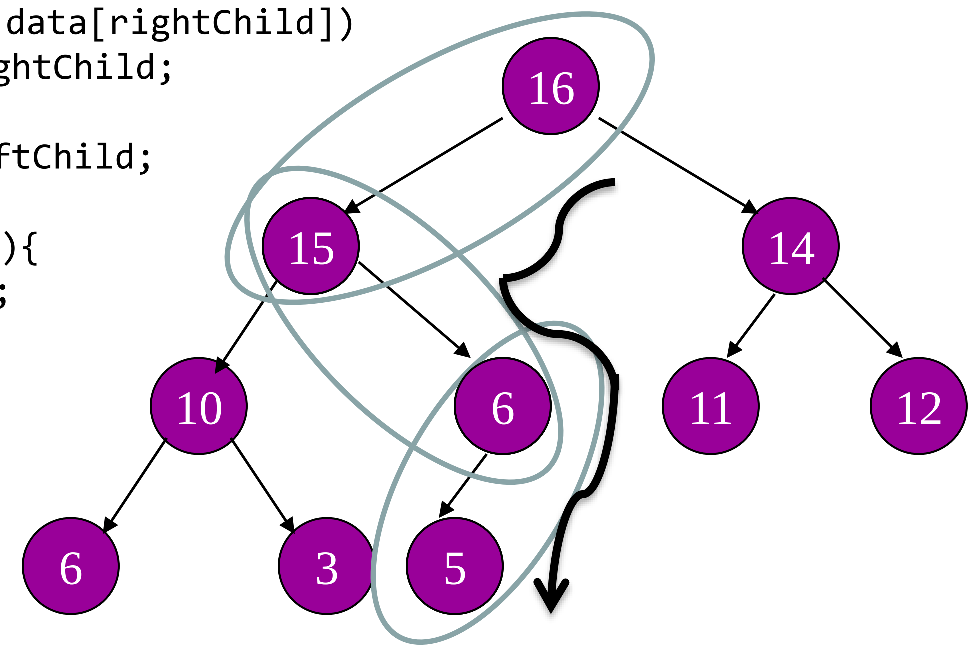
extract the element from the root;
put the element from the last leaf in its place;
remove the last leaf;
// both subtrees of the root are heaps;
//Reheap DOWN
p = *the root*;
while p *is not a leaf &* p<*any of its childre*
    swap p *with the larger child*;



DeleteMax & ReheapDown Recursive

```
template<class T>
void BinaryHeap<T>::ReheapDown(int cnode, int last){
     int maxChild, rightChild, leftChild;

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

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

rightmost node in the last level



DeleteMax & ReheapDown Recursive

```
template<class T>
void BinaryHeap<T>::ReheapDown(int cnode, int last){
     int maxChild, rightChild, leftChild;

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

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

rightmost node
in the last level

```
void BinaryHeap<T>::DeleteMax(T& item){
          item = data[0];
          data[0] = data[currentSize-1];
          currentSize--;
          ReheapDown(0, currentSize-1);
}
```

DeleteMax Iterative

DO IT YOURSELF

Quick Review

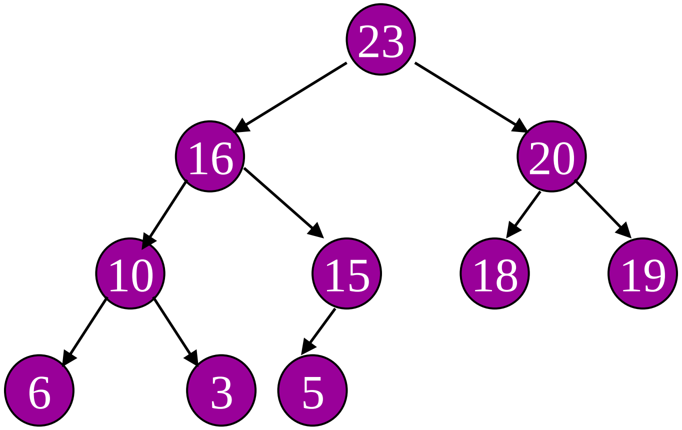
Priority Queue

- A priority queue should allow at least the following two operations:
  - **Insert (enqueue)**
  - **deleteMax**, which finds, returns, and removes the maximum element in the priority queue (**dequeue**)

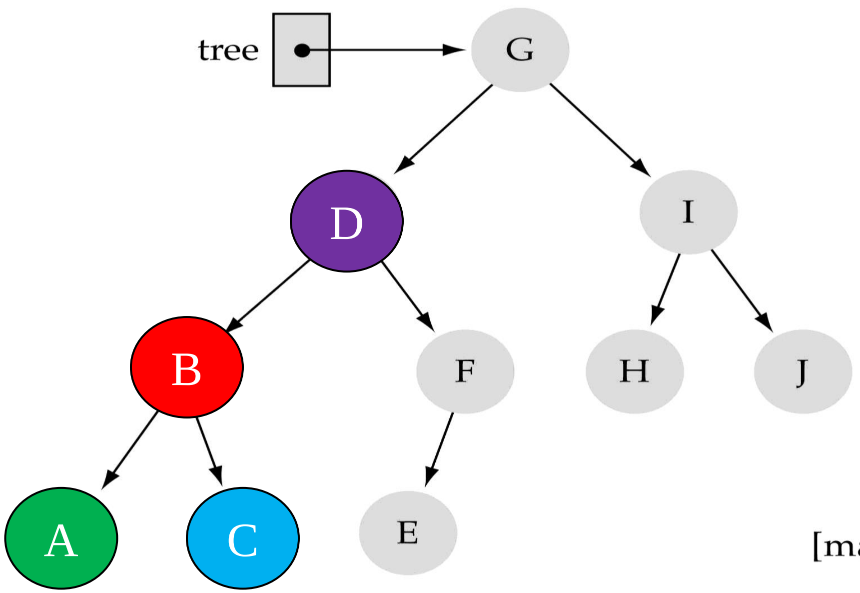
Max Heap

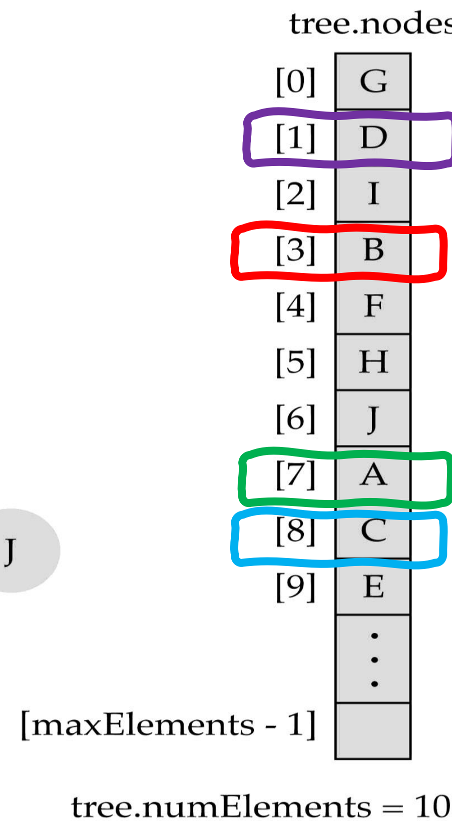
It is a binary tree with the following properties:

1. It is a complete binary tree.
2. The value stored in a node is >= to values stored in the children (heap-property)



Heap using array– some Properties





B is at index 3

Left child:

*nodes[2*index+1]*

2*3 +1 = 7

Right child:

*nodes[2*index+2]*

2*3 +2 = 8

Parent:

nodes[(index-1)/2]

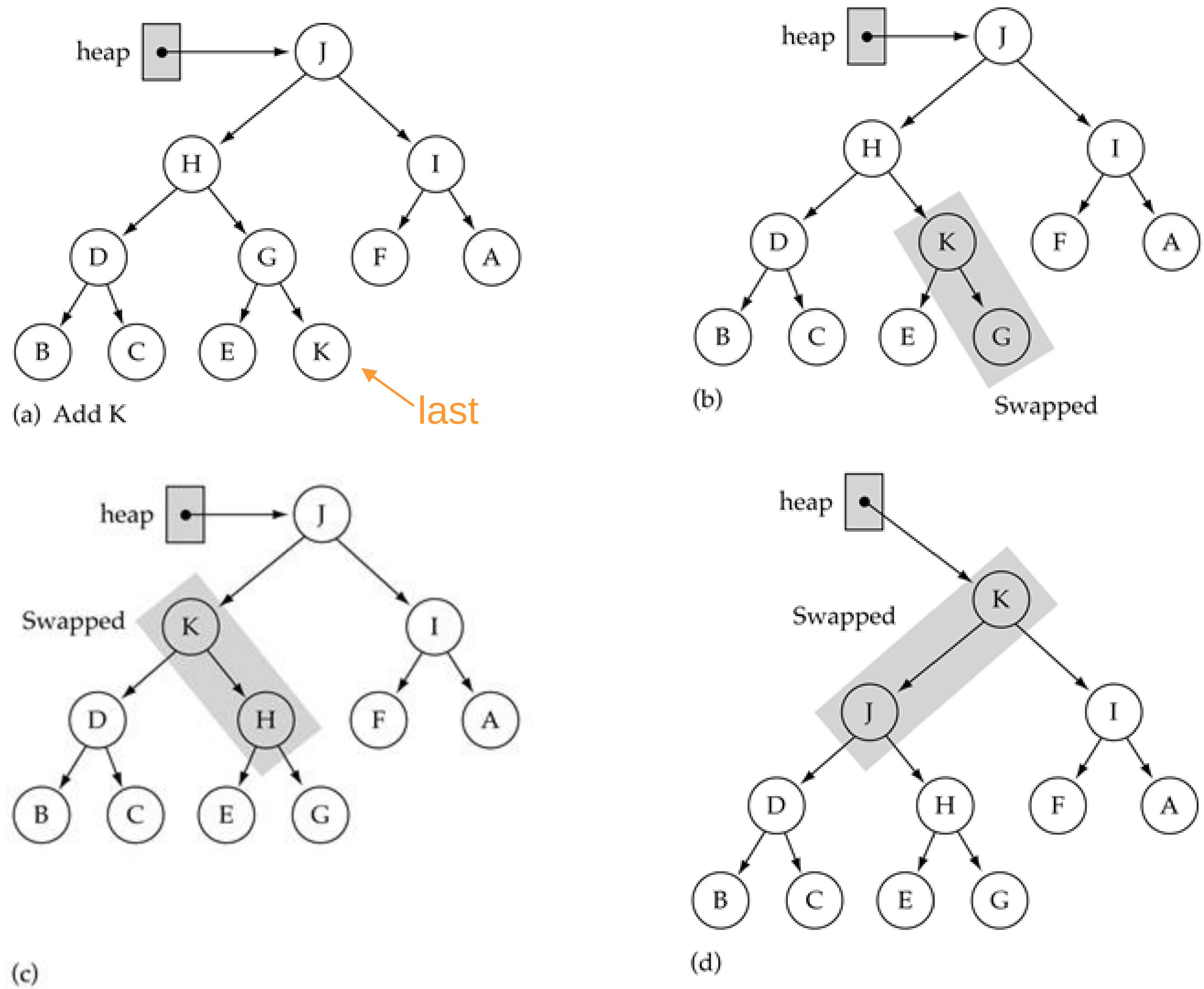
(3-1)/2= 1

Leaf nodes of tree: H, J, A, C, E

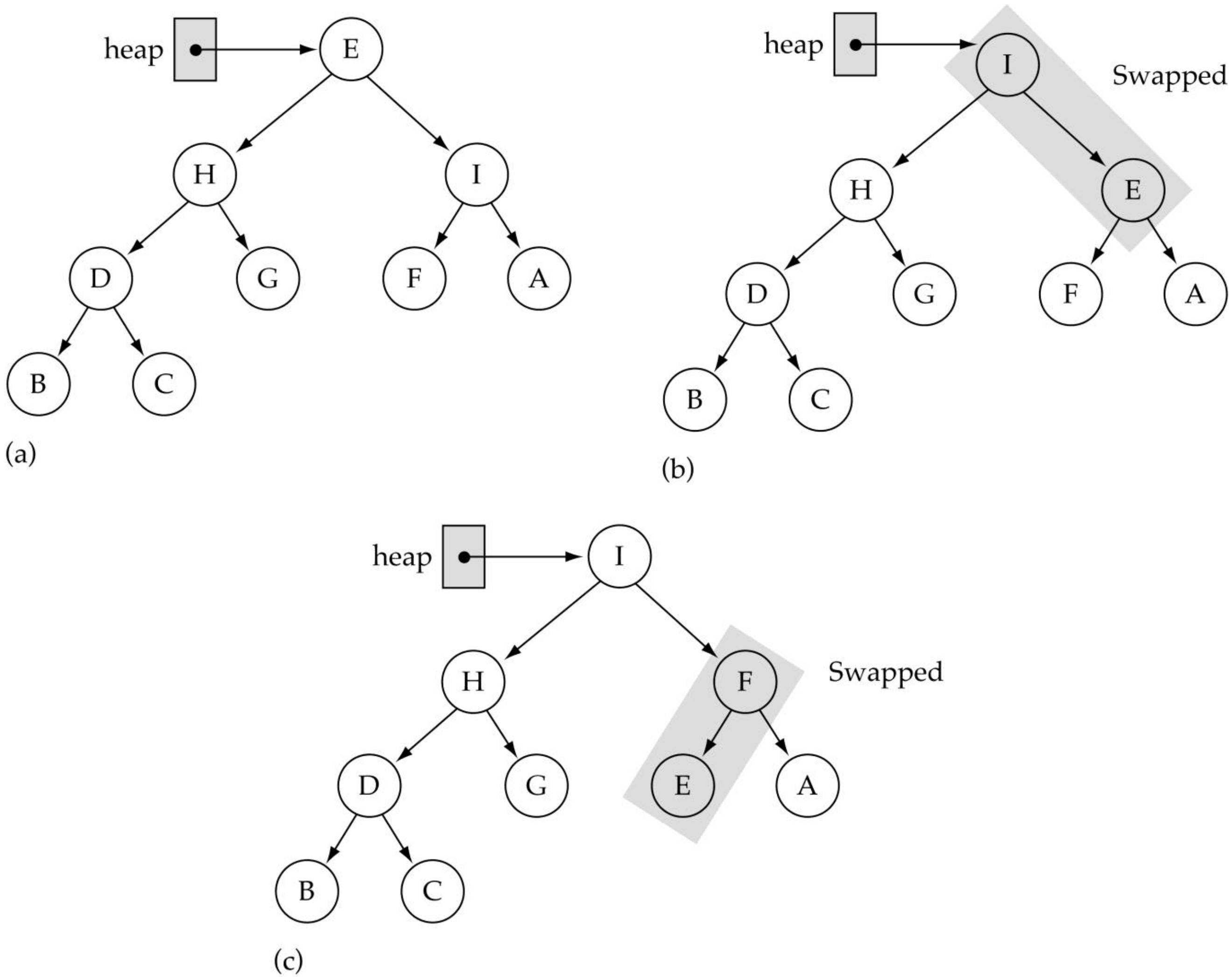
Between index 5 & 9

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

The ReheapUp function (used by insertItem)



The ReheapDown function (used by deleteItem)



heap property is violated at the root of the tree

HEAPSORT

HeapSort

BUILDHEAP

- First, make the unsorted array into a heap by satisfying the order property..

Then repeat the steps below until there are no more unsorted data

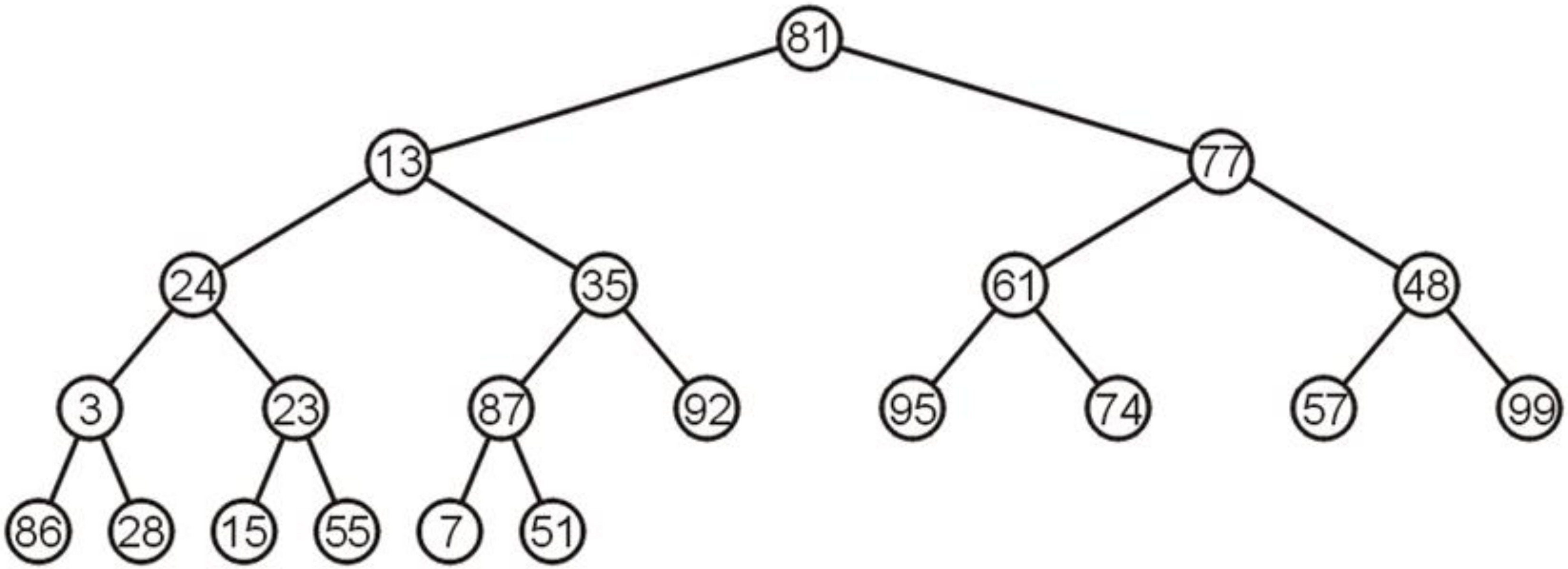
- **Take the root (maximum) element off the heap by swapping it into its correct place in the array (at the end of the unsorted data).**
- **Reheap the remaining unsorted data.**

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

Building the heap

numdata = 21

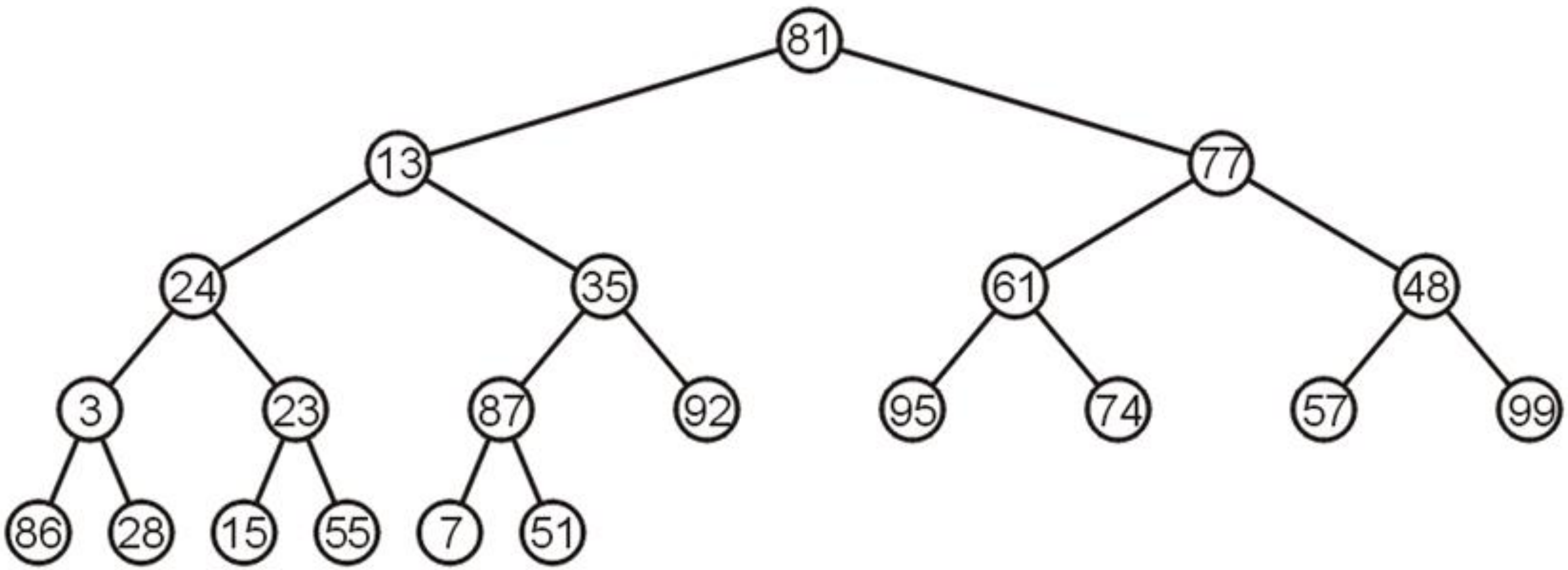
Consider the following unsorted array



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

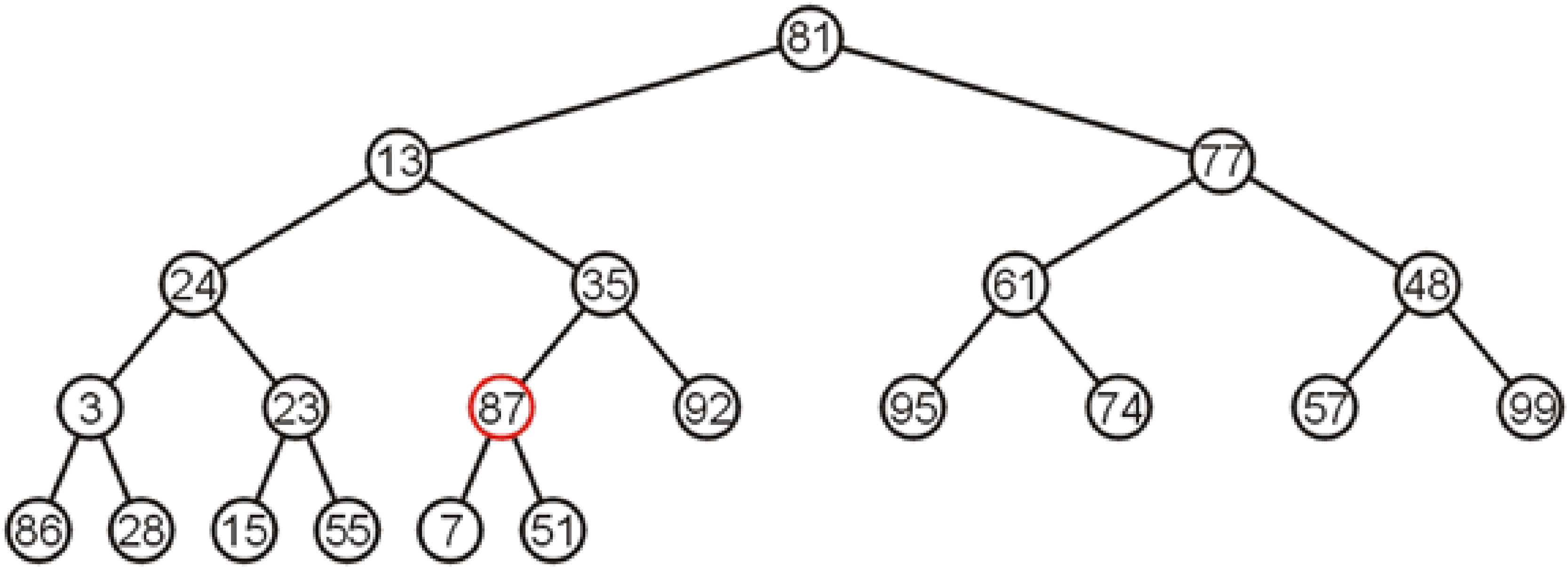
- All leaf nodes are trivial heaps
- Leaf nodes: 21/2 to 20 → 10 to 20



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

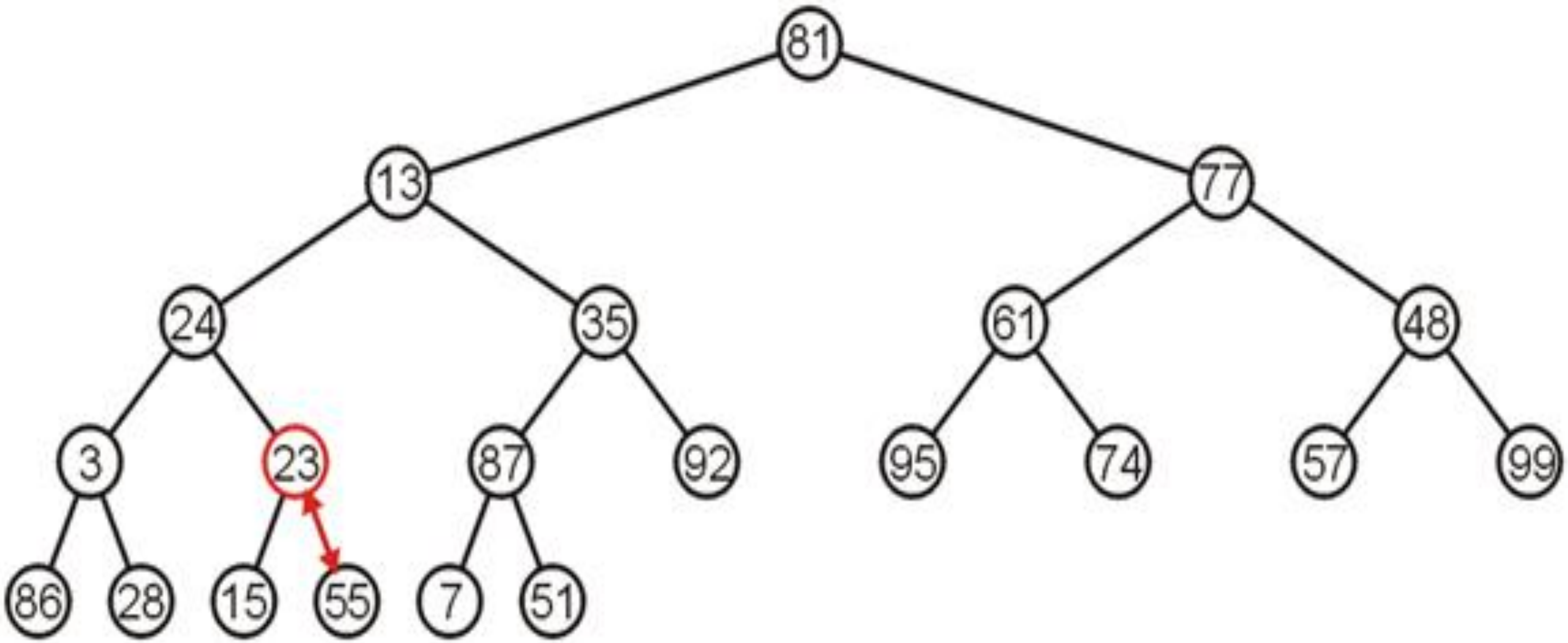
Building the heap

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



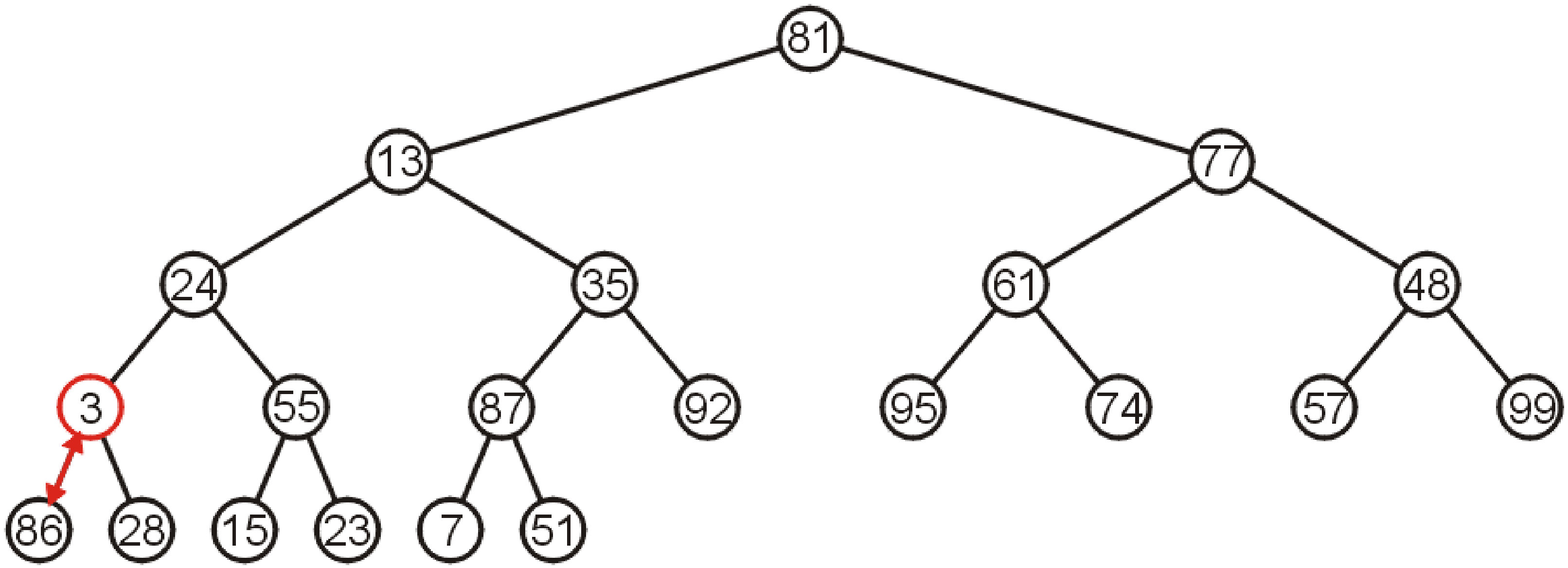
Building the heap

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



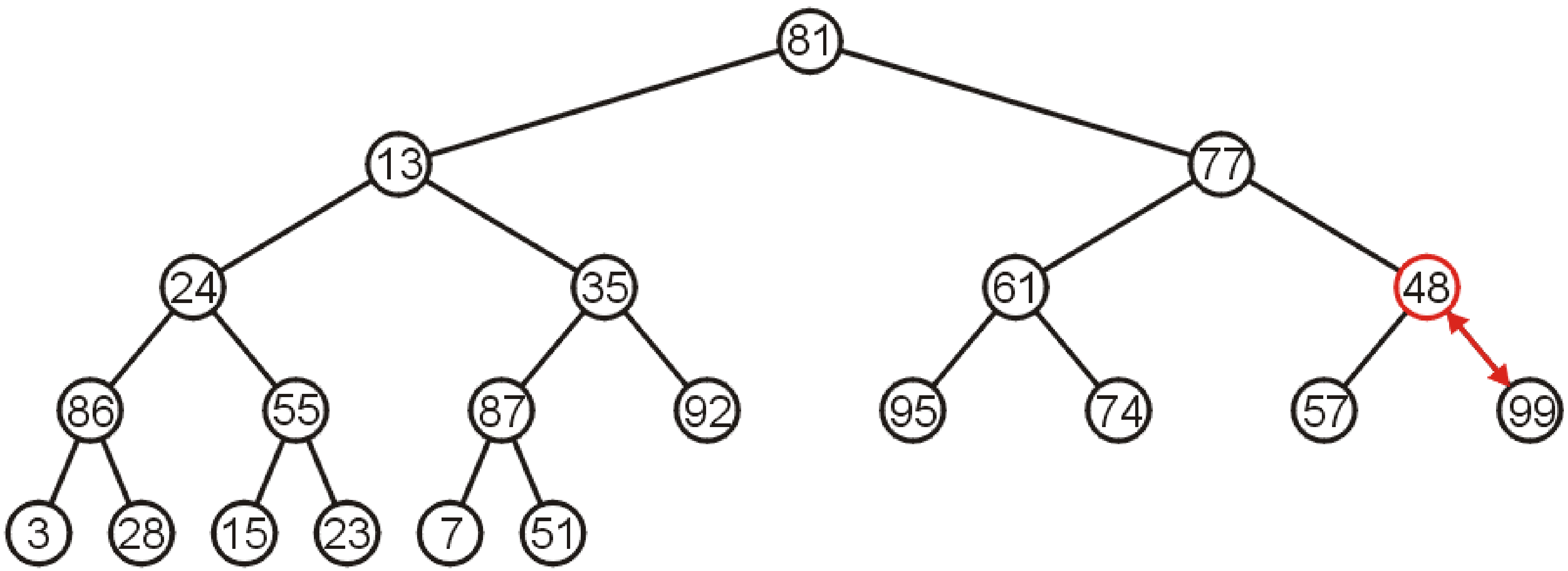
Building the heap

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



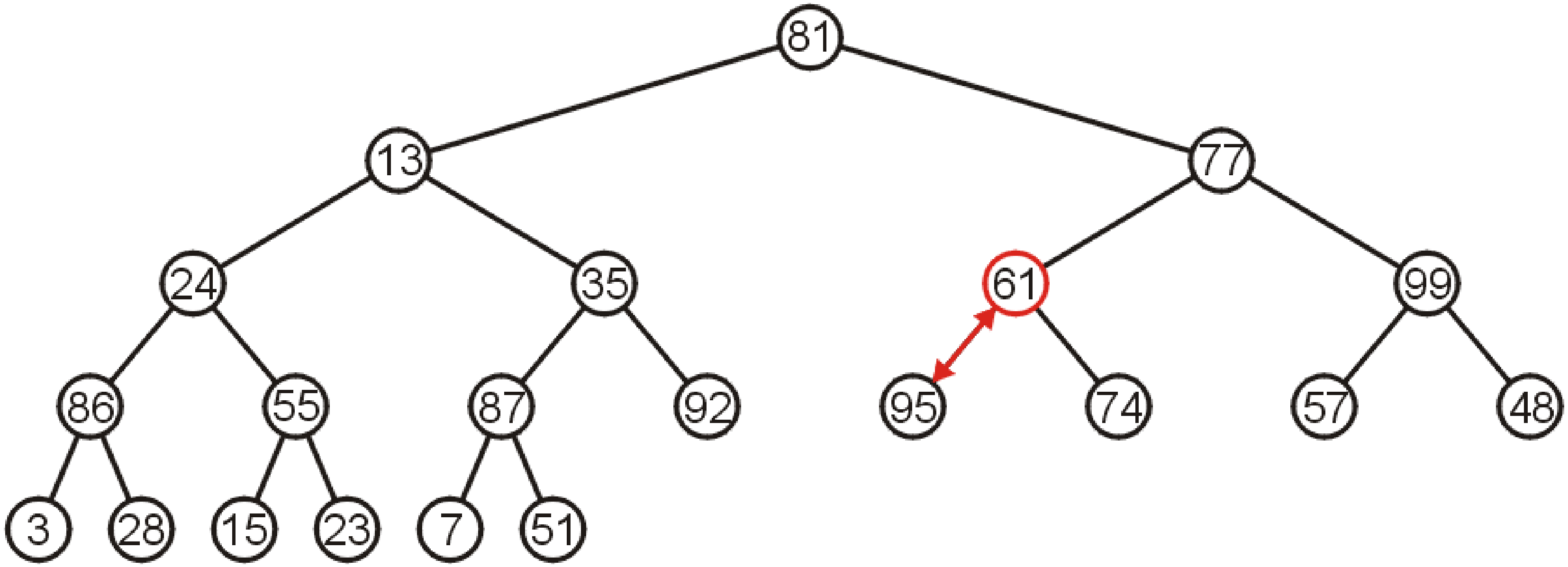
Building the heap

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



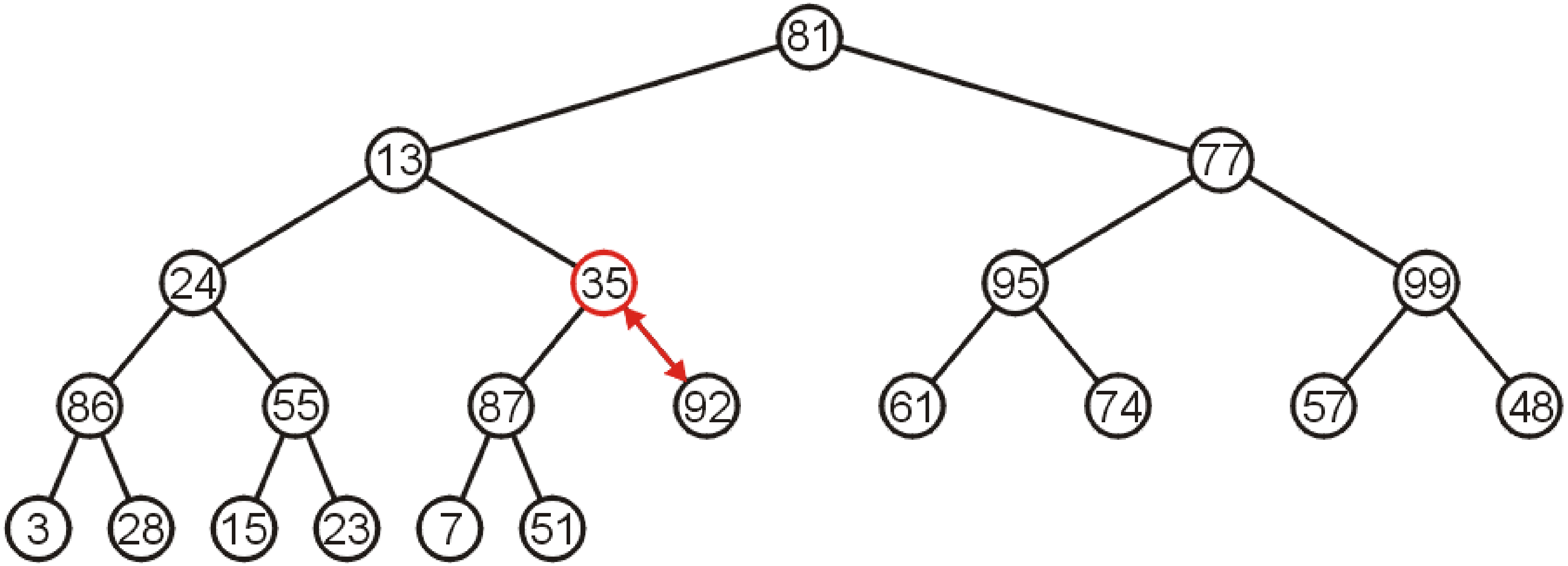
Building the heap

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



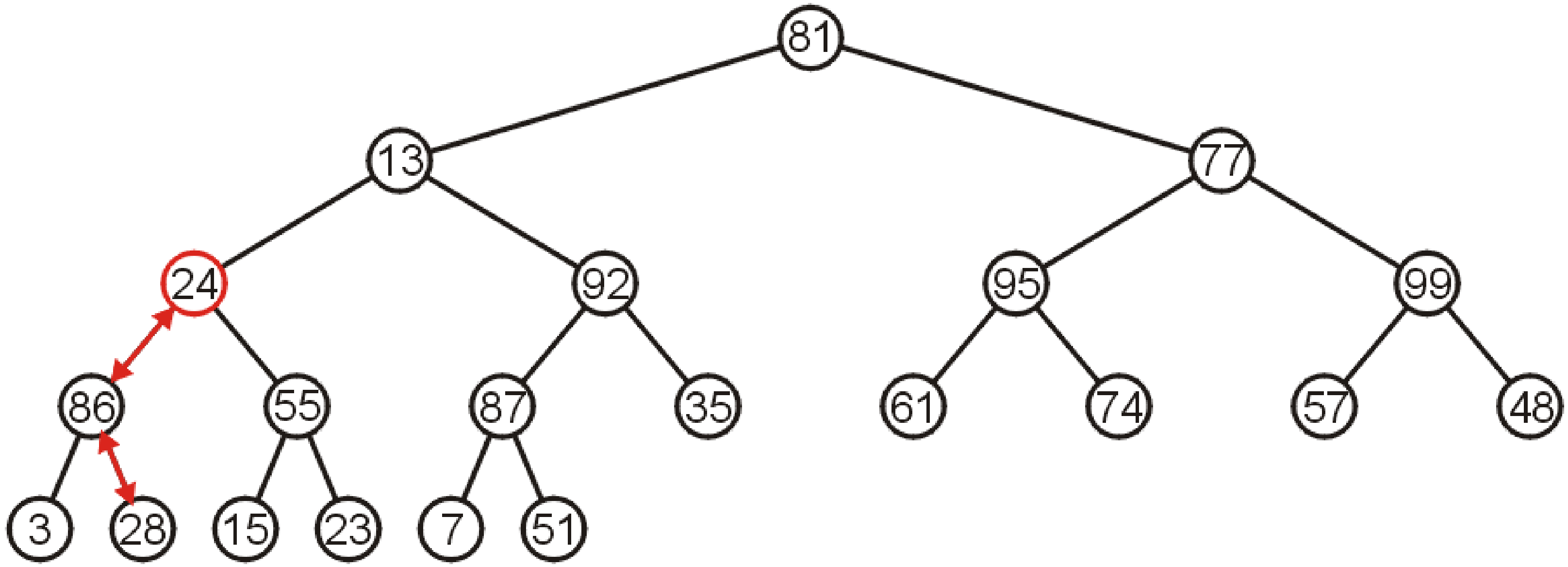
Building the heap

- As does swapping 35 and 92



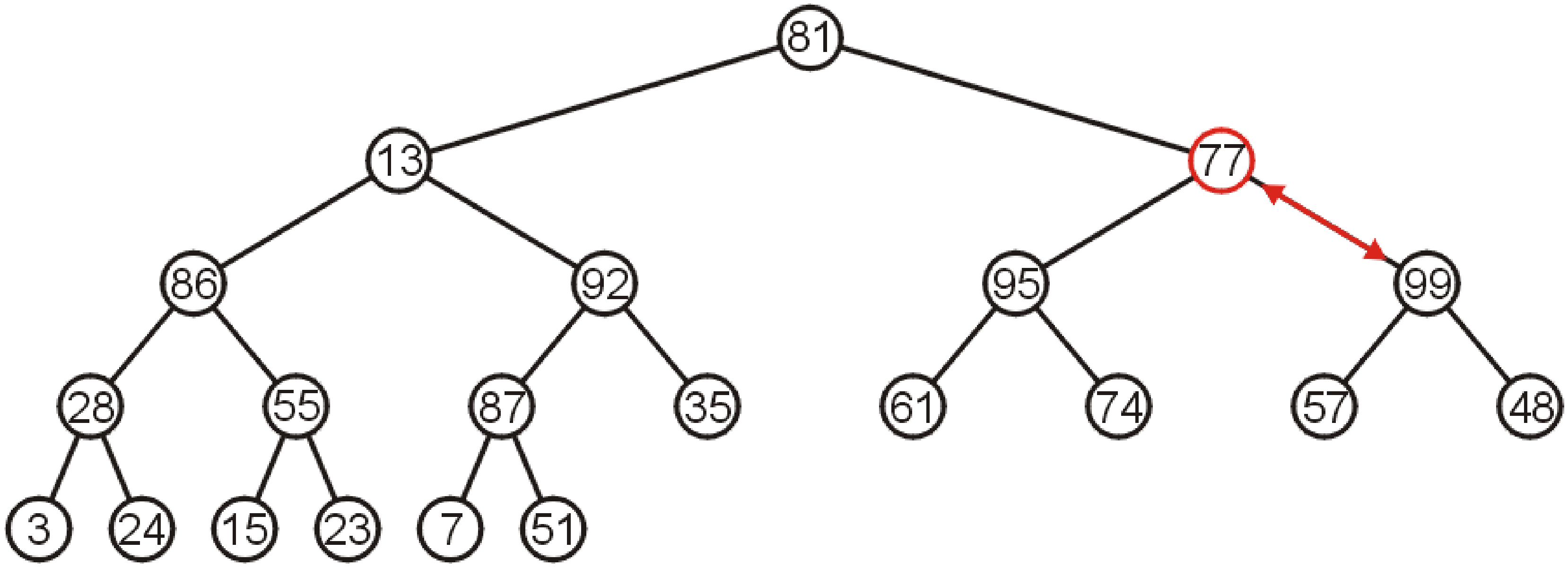
Building the heap

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



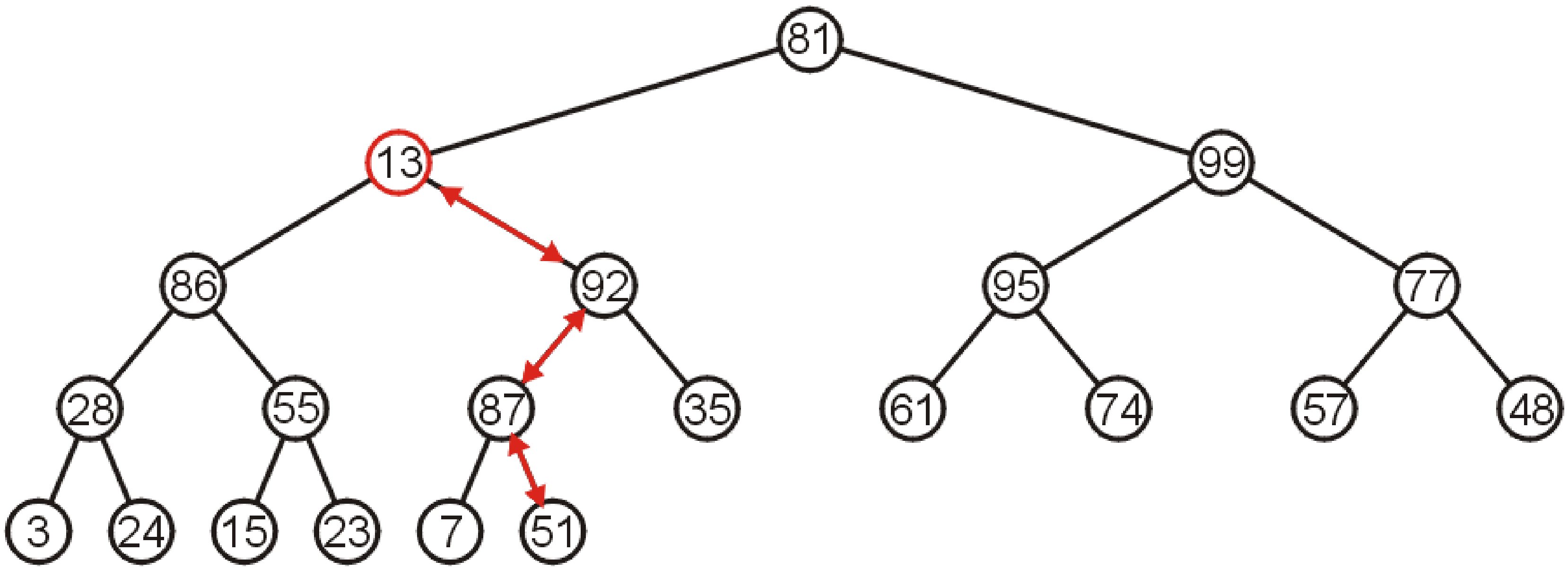
Building the heap

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



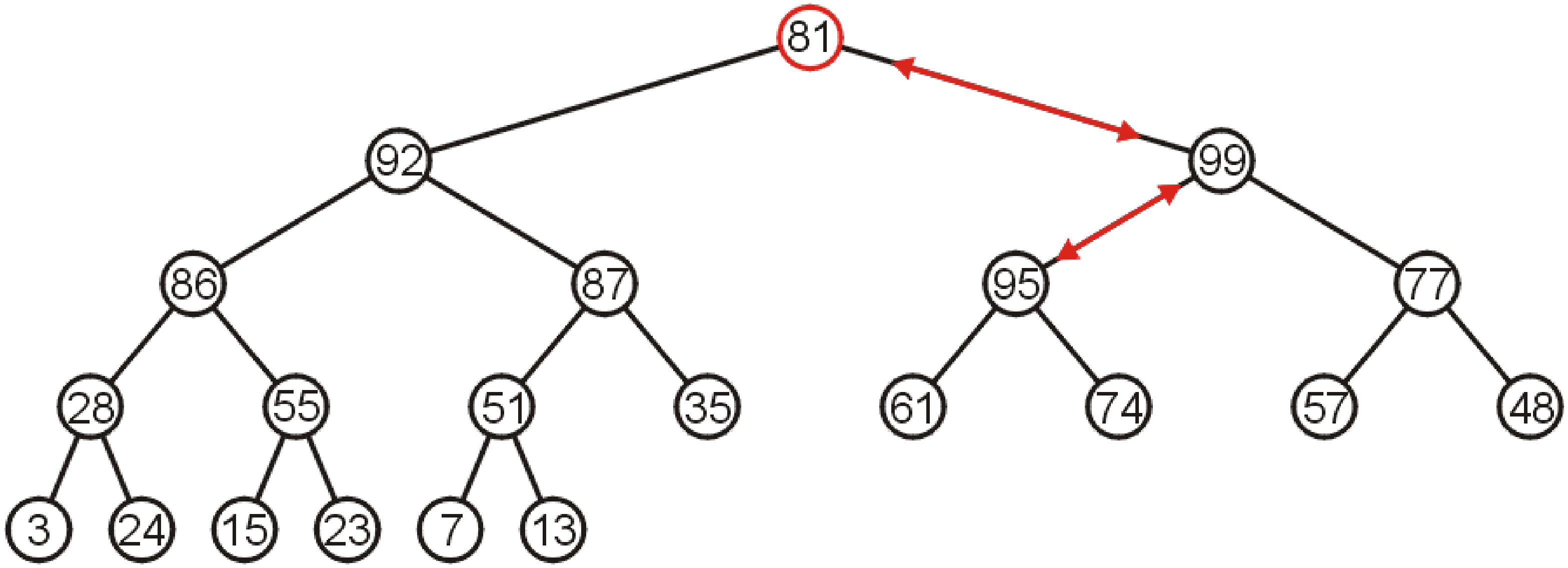
Building the heap

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



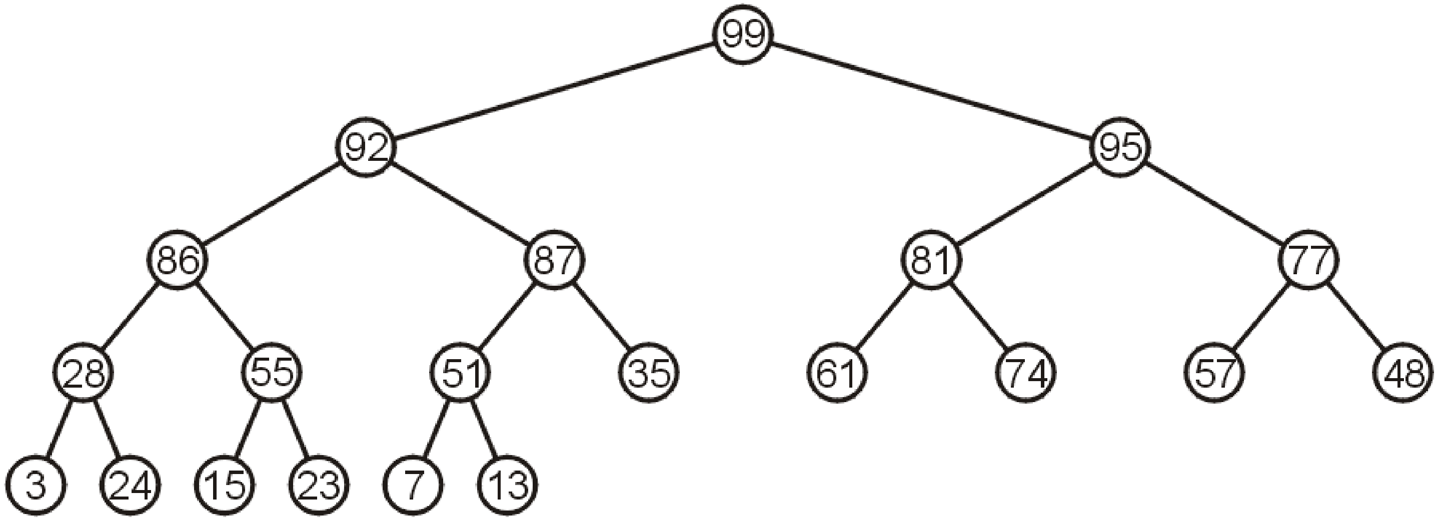
Building the heap

- The root need only be percolated down by two levels



Building the heap

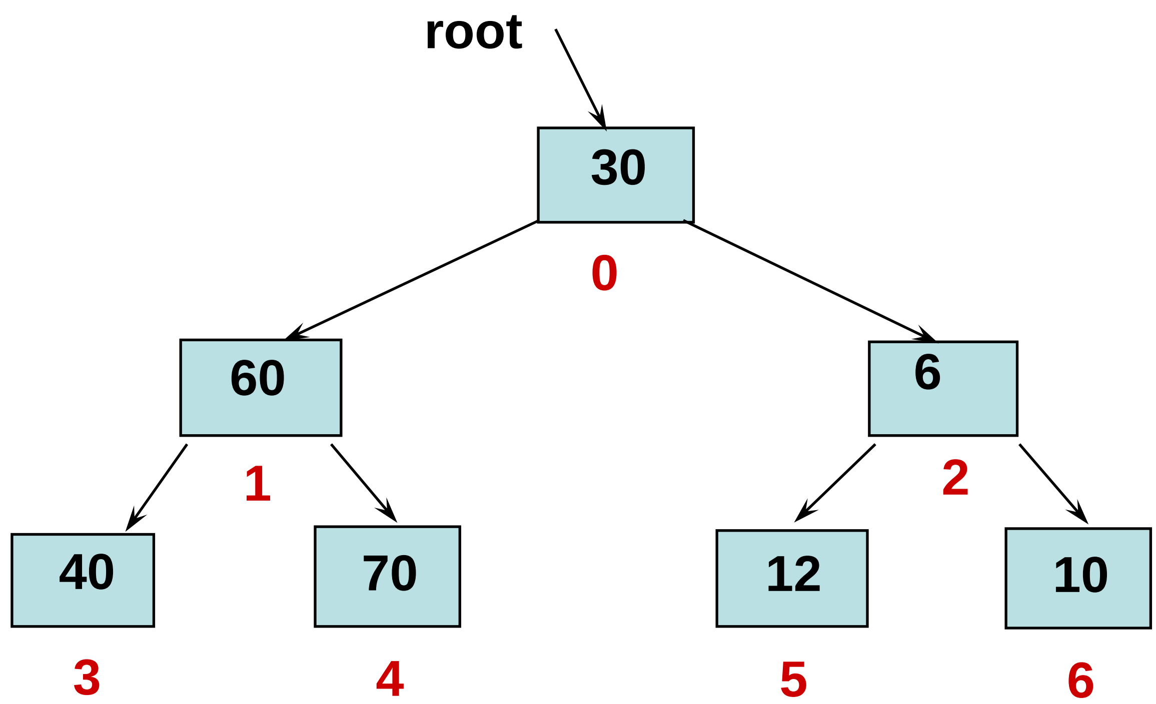
The final product is a max-heap



Build heap

Build heap of the following unsorted array

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

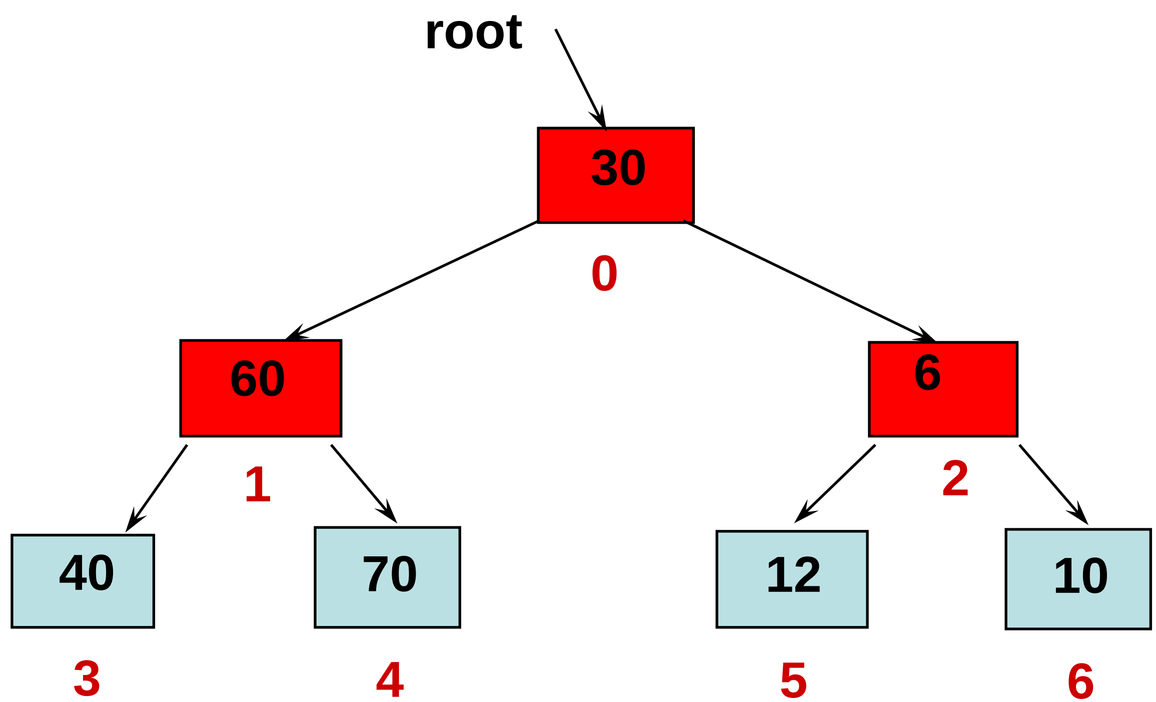


72

Build heap

Build heap of the following unsorted array

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



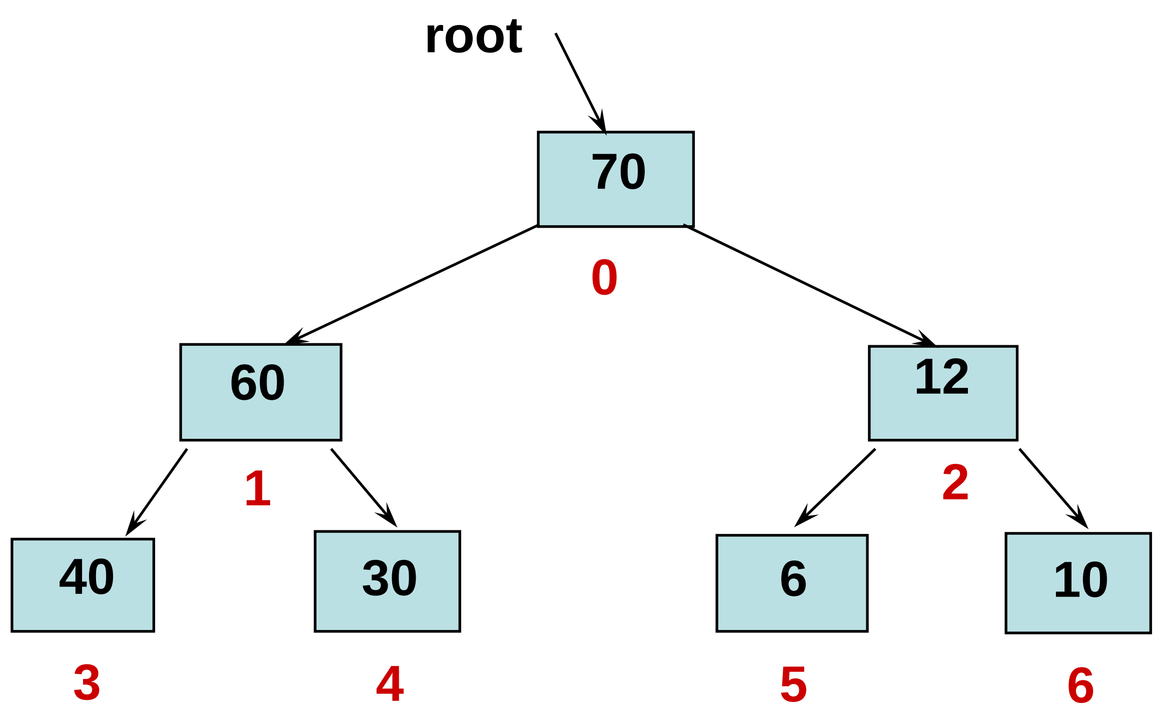
73

sorting

After creating the original heap

values

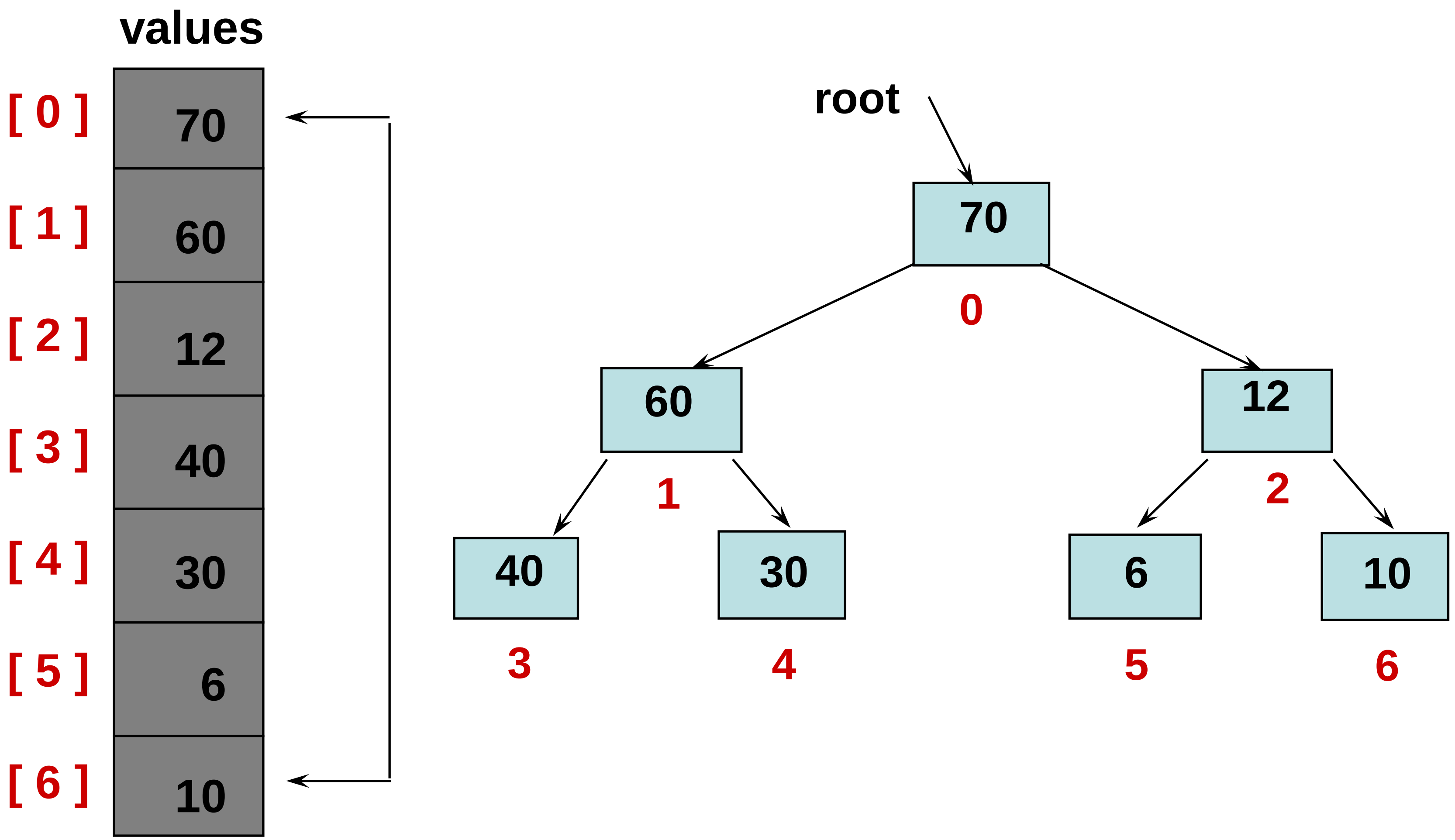
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	6
[6]	10



74

sorting

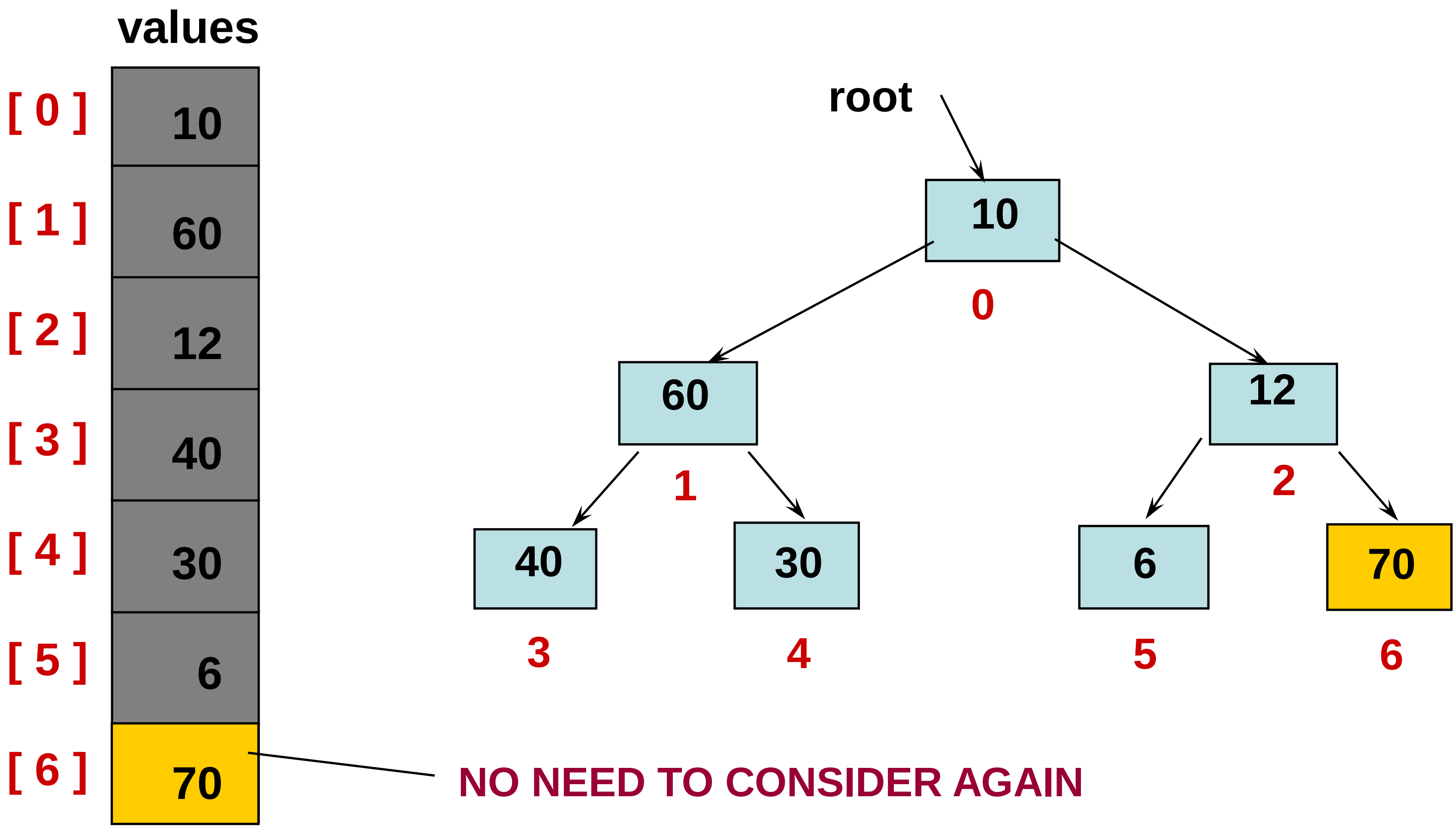
Swap root element into last place in unsorted array



75

sorting

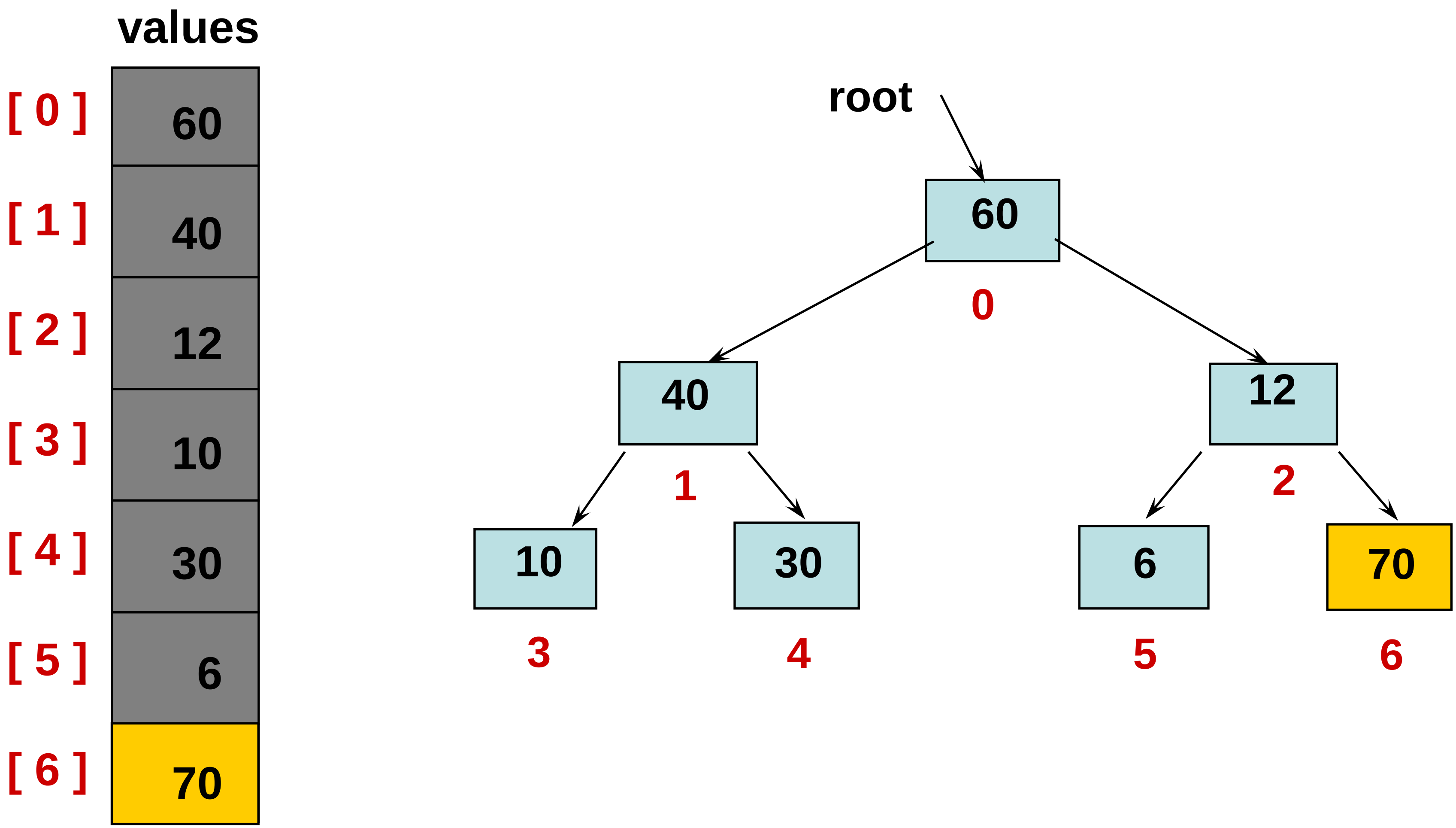
After swapping root element into its place



76

sorting

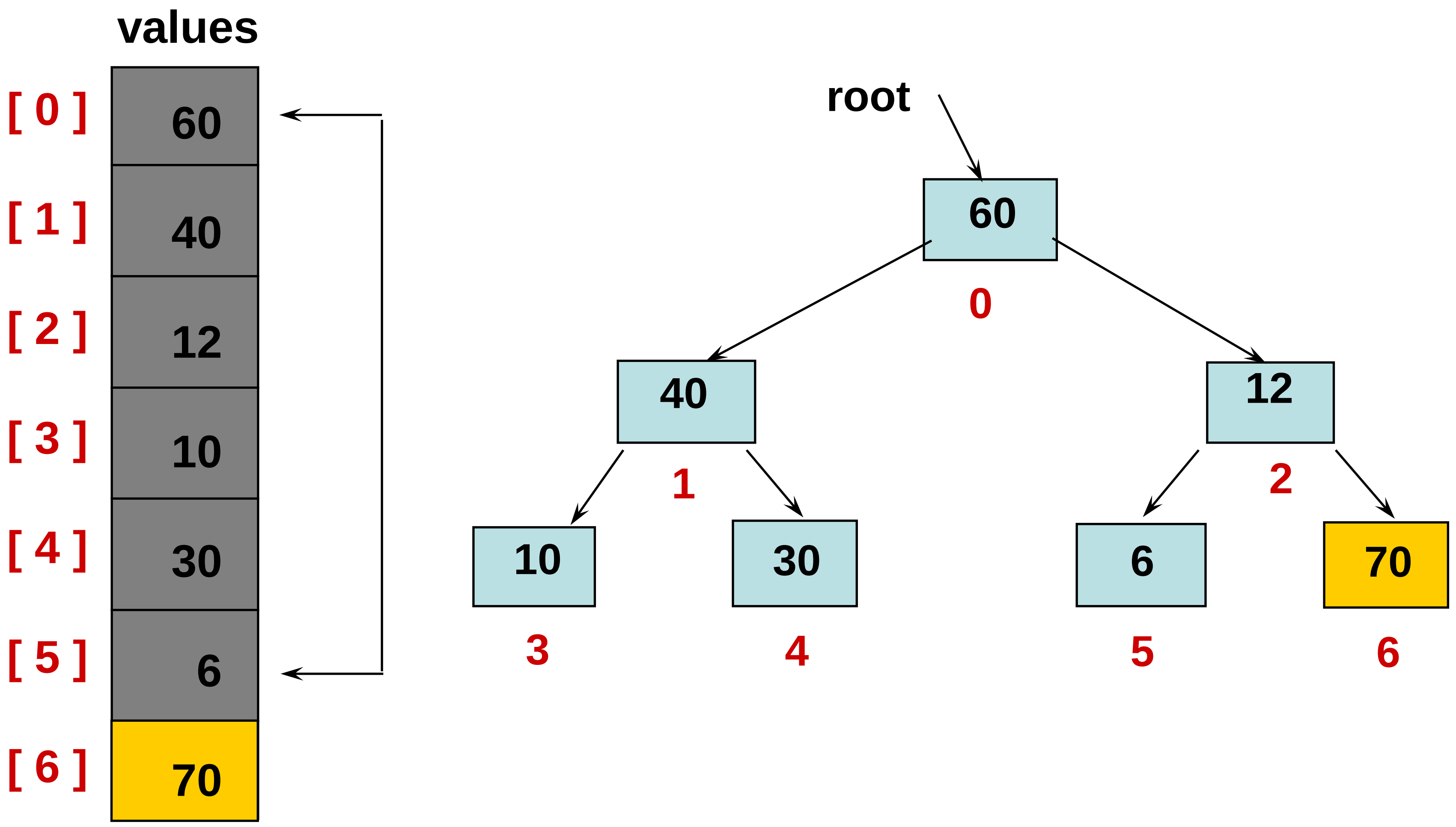
After reheaping remaining unsorted data



77

sorting

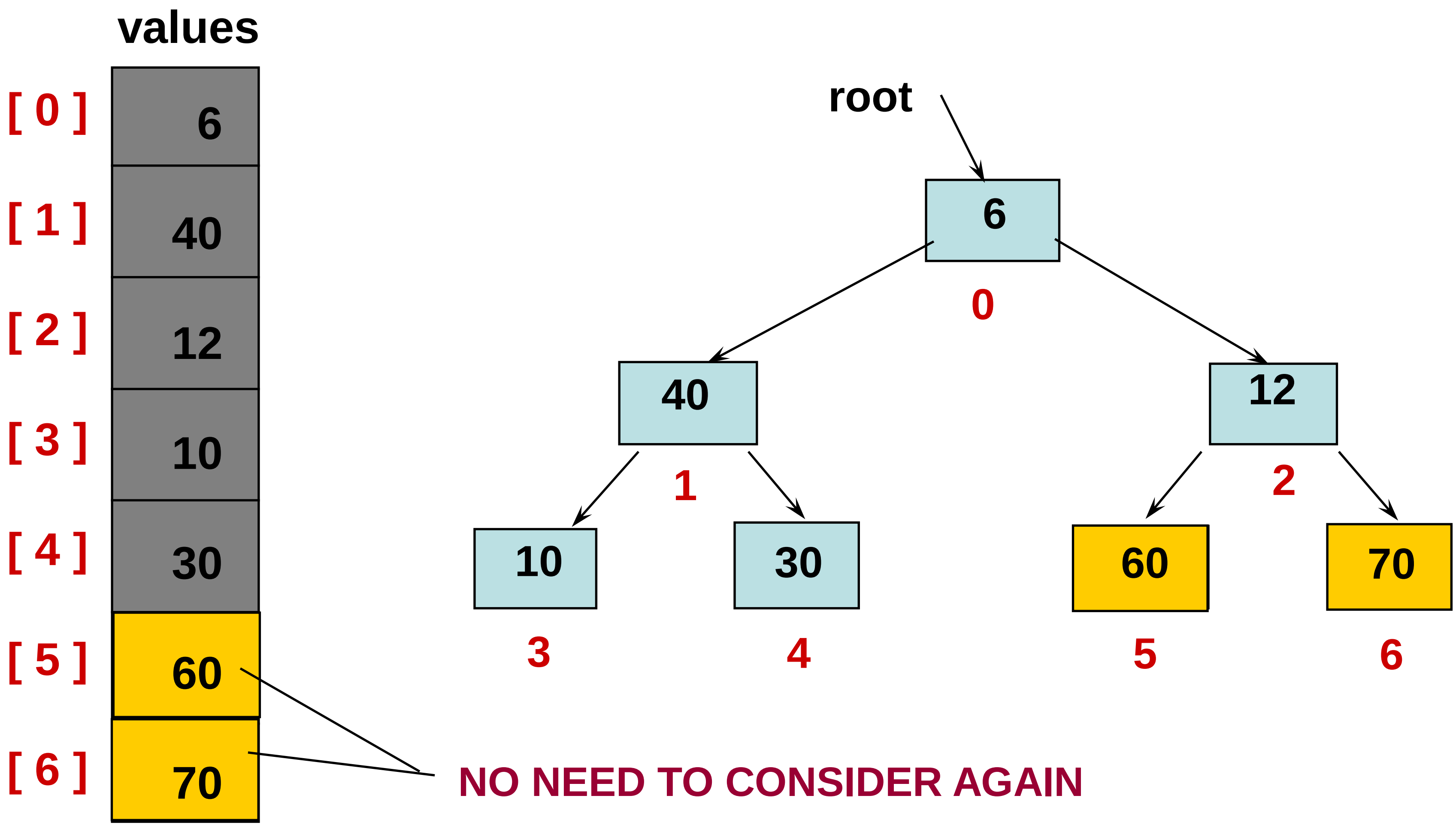
Swap root element into last place in unsorted array



78

sorting

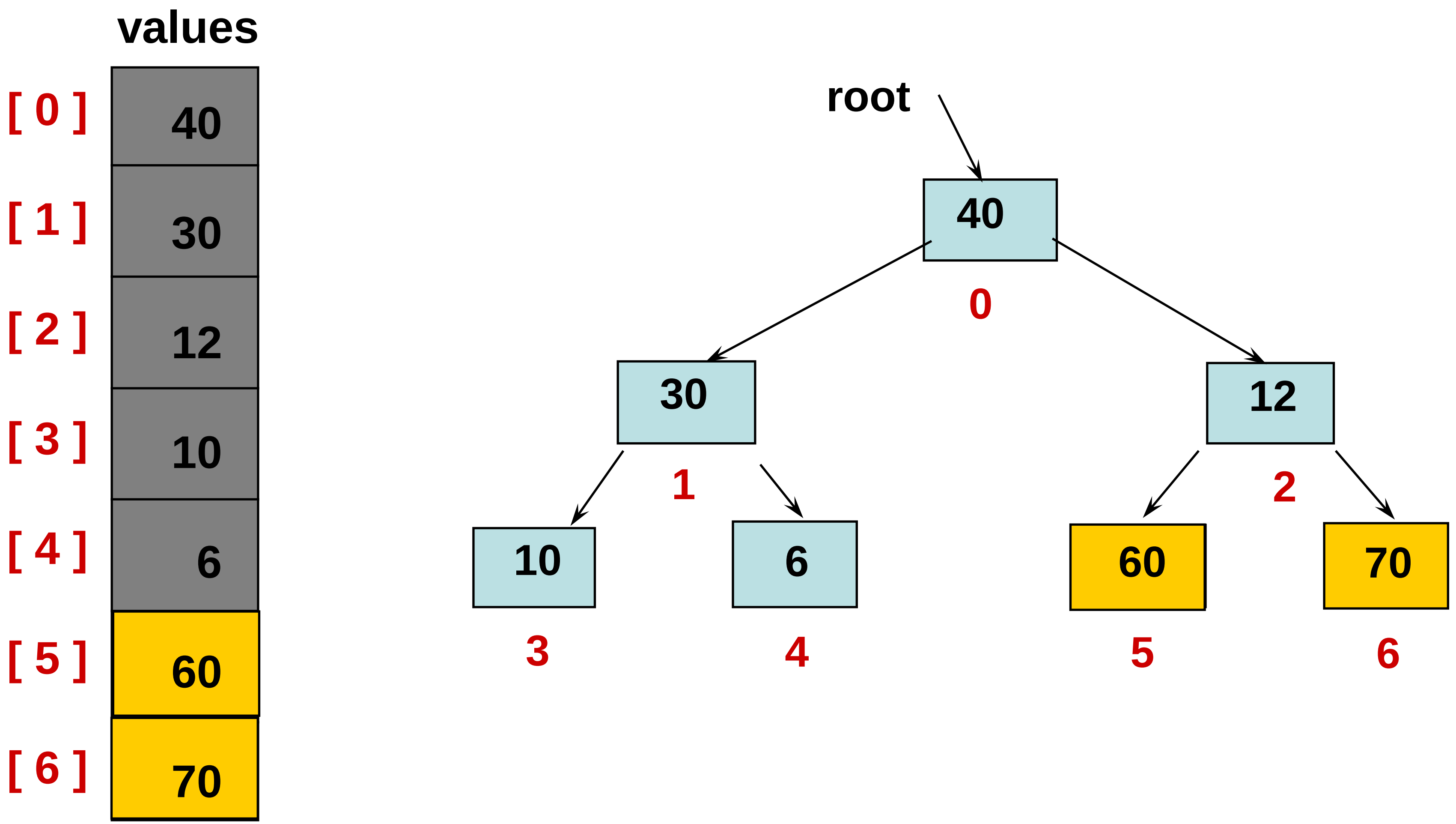
After swapping root element into its place



79

sorting

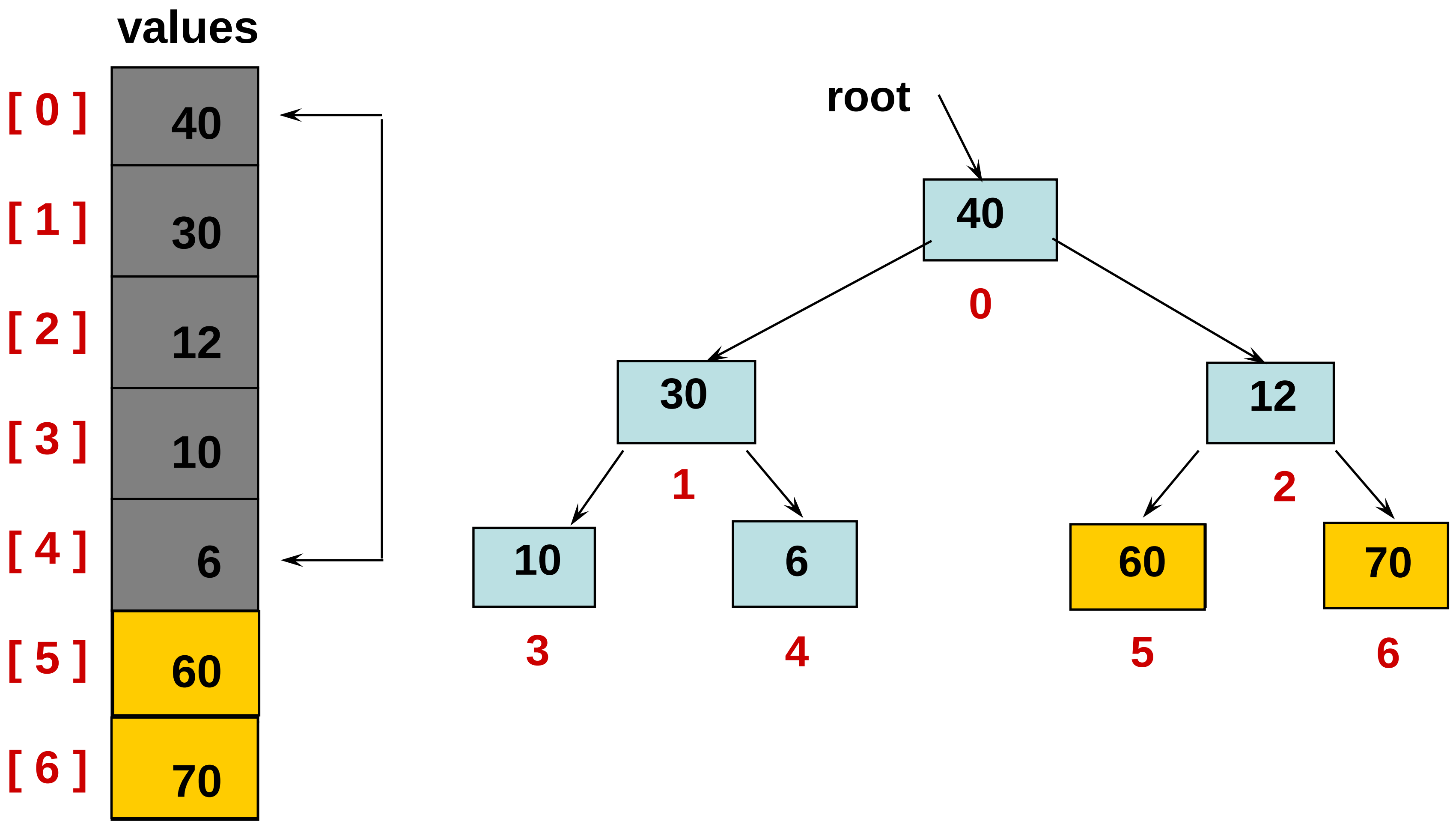
After reheaping remaining unsorted data



80

sorting

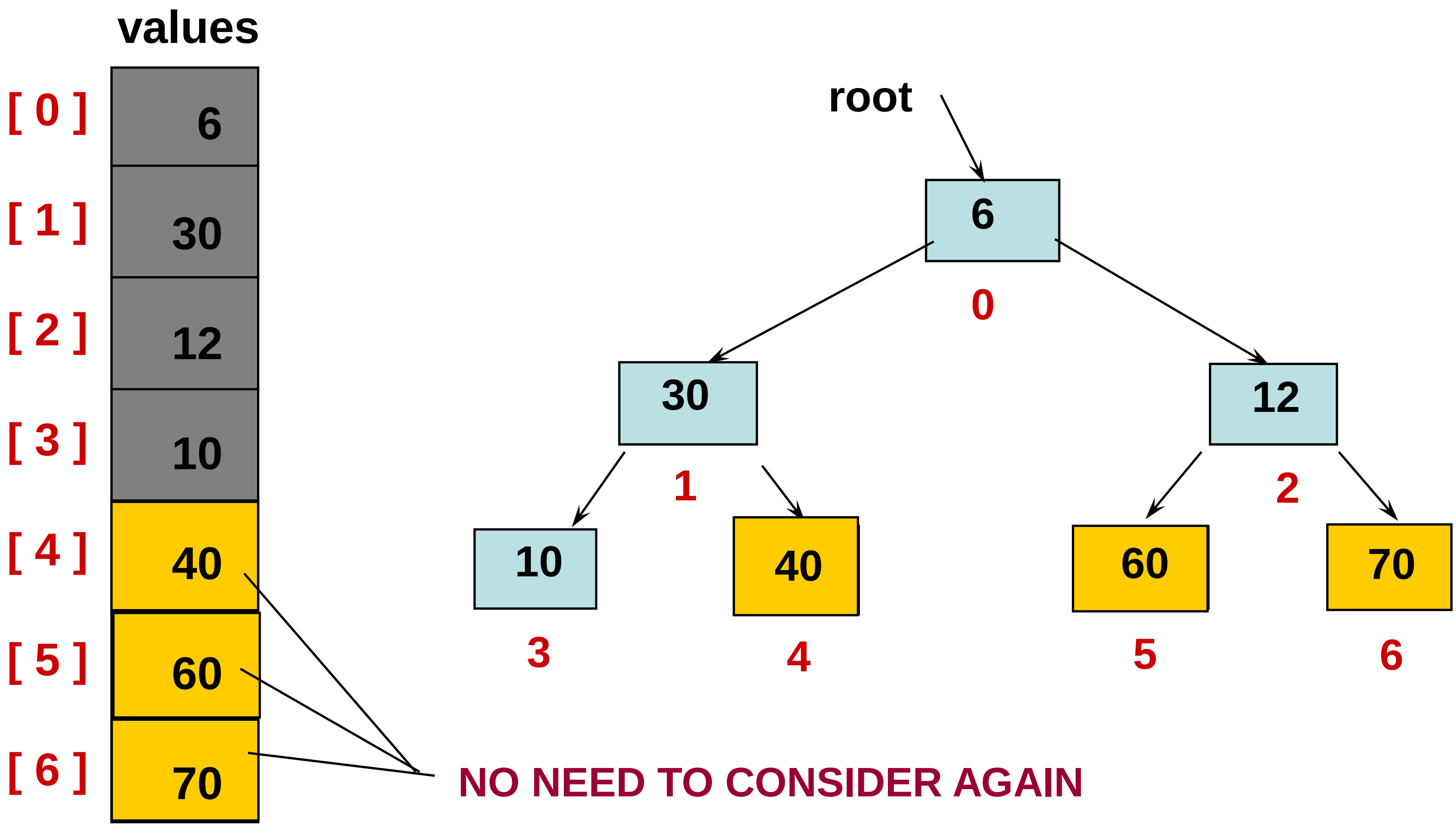
Swap root element into last place in unsorted array



81

sorting

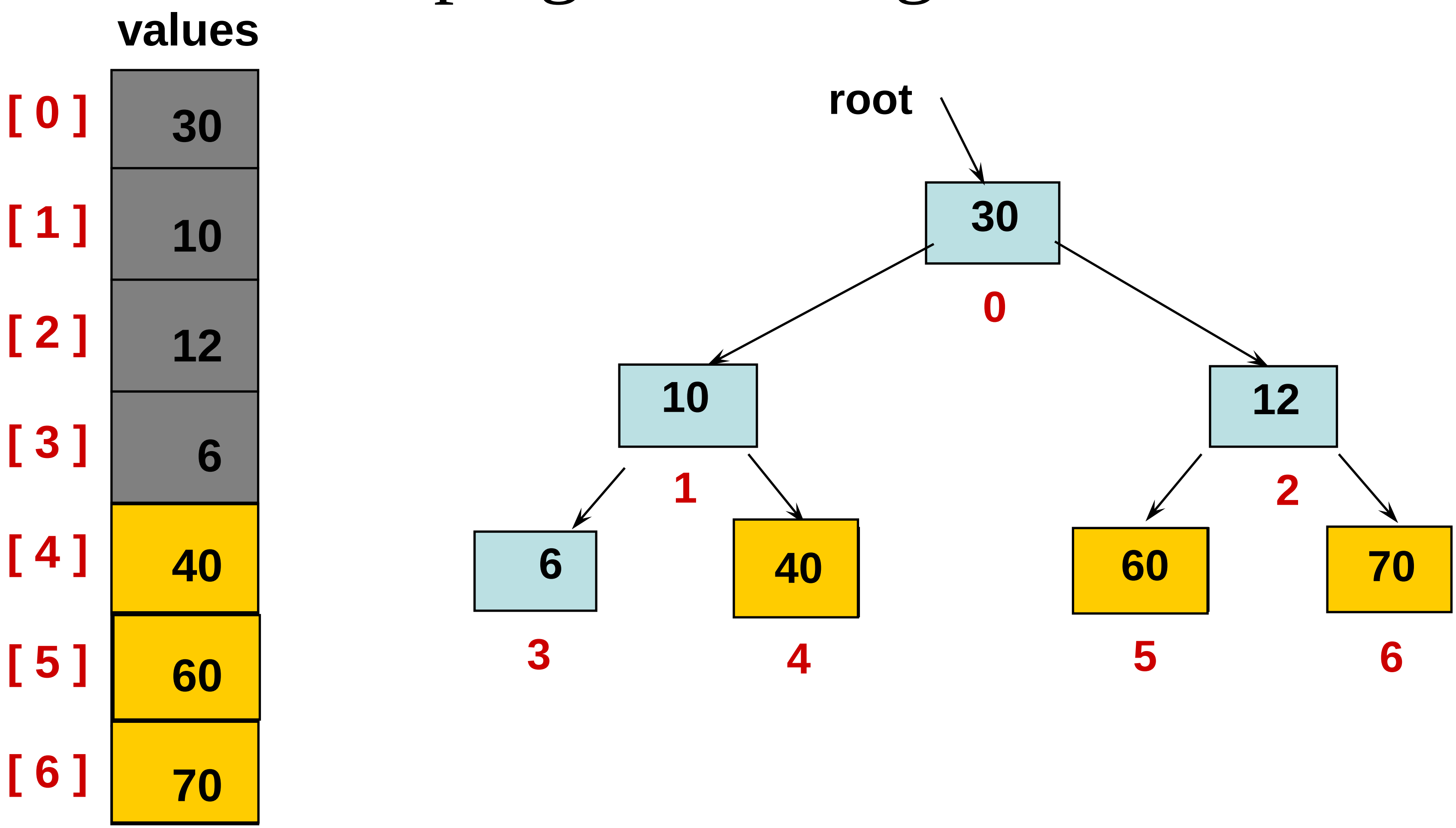
After swapping root element into its place



82

sorting

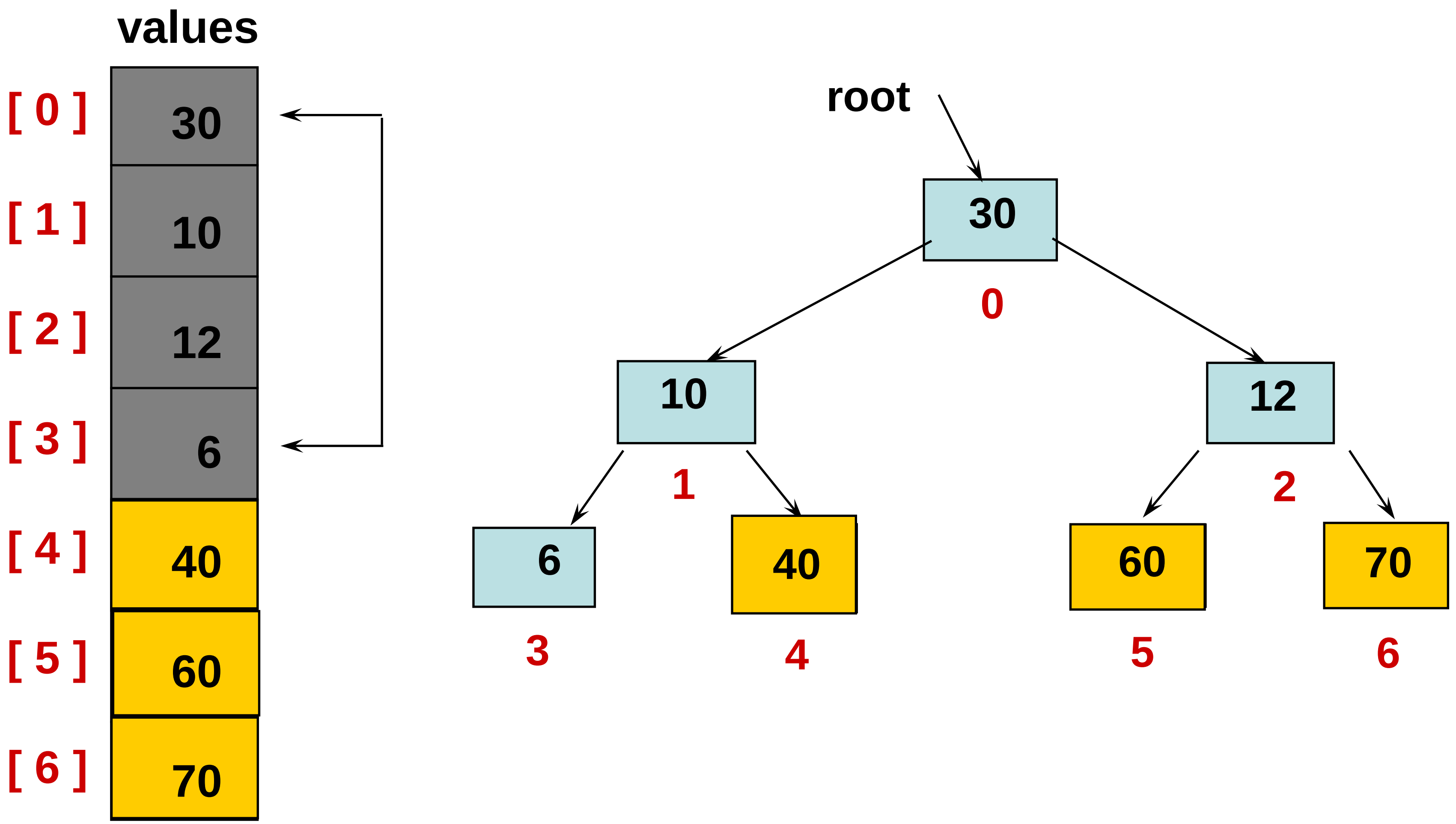
After reheaping remaining unsorted data



83

sorting

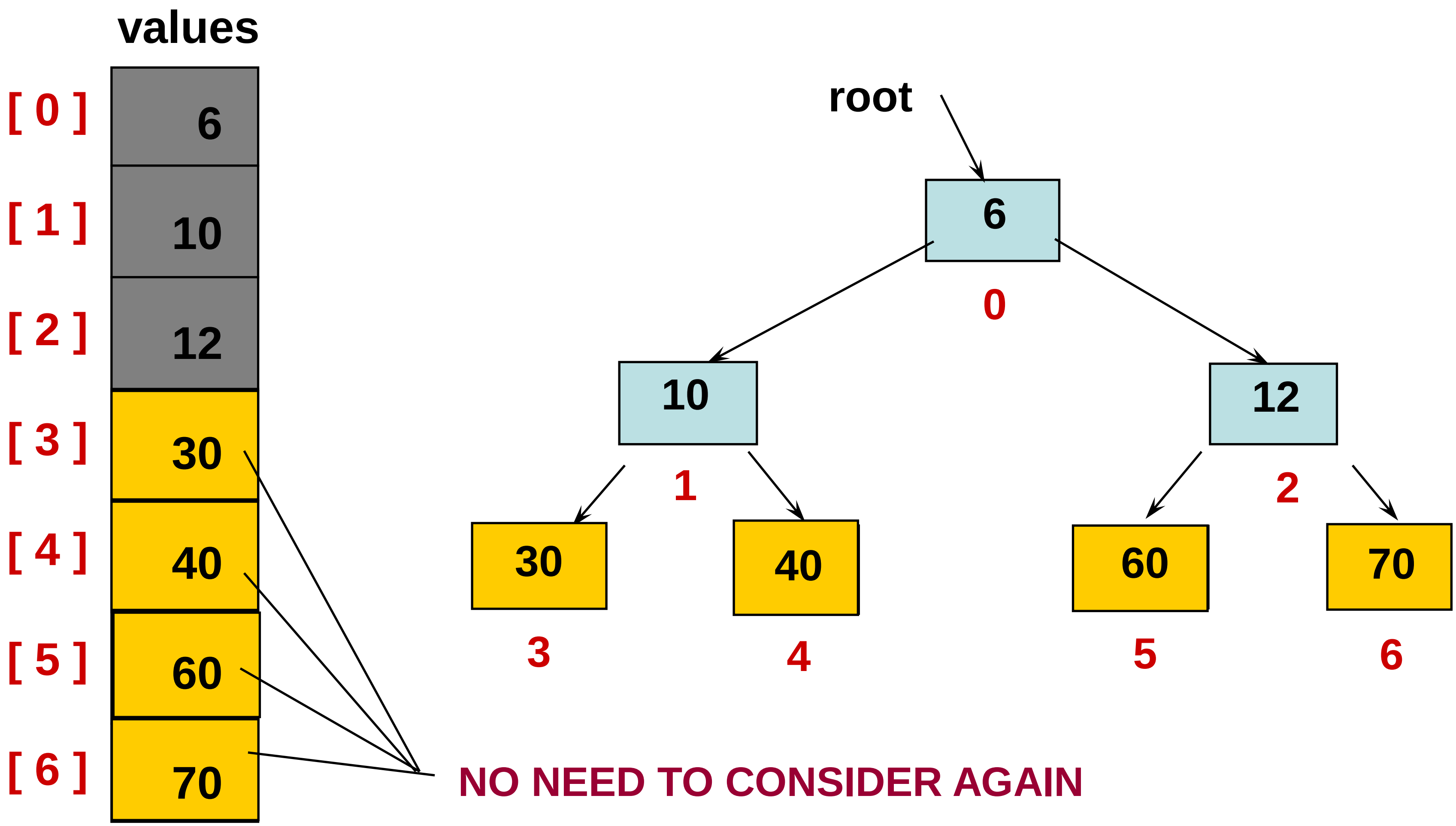
Swap root element into last place in unsorted array



84

sorting

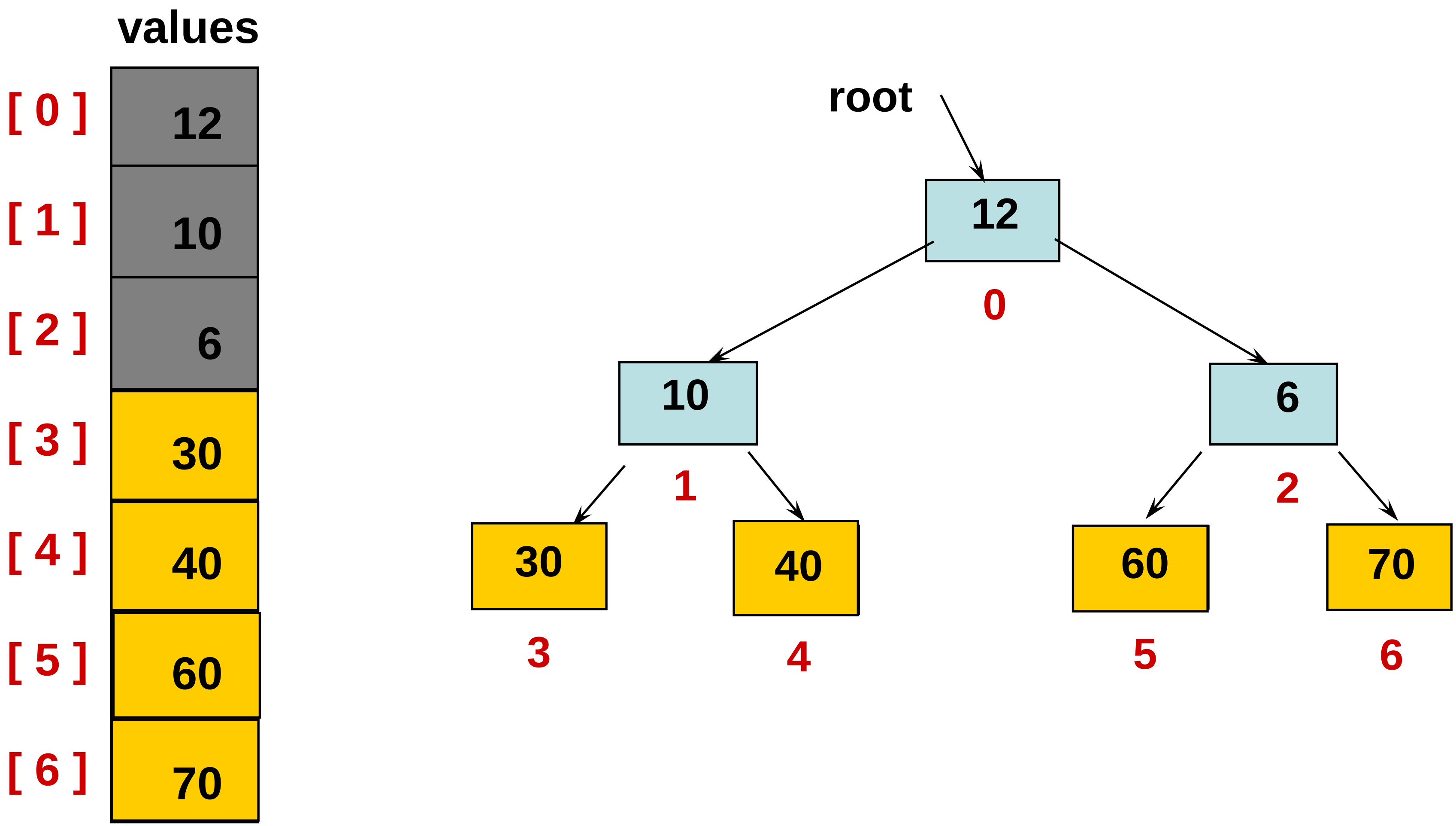
After swapping root element into its place



85

sorting

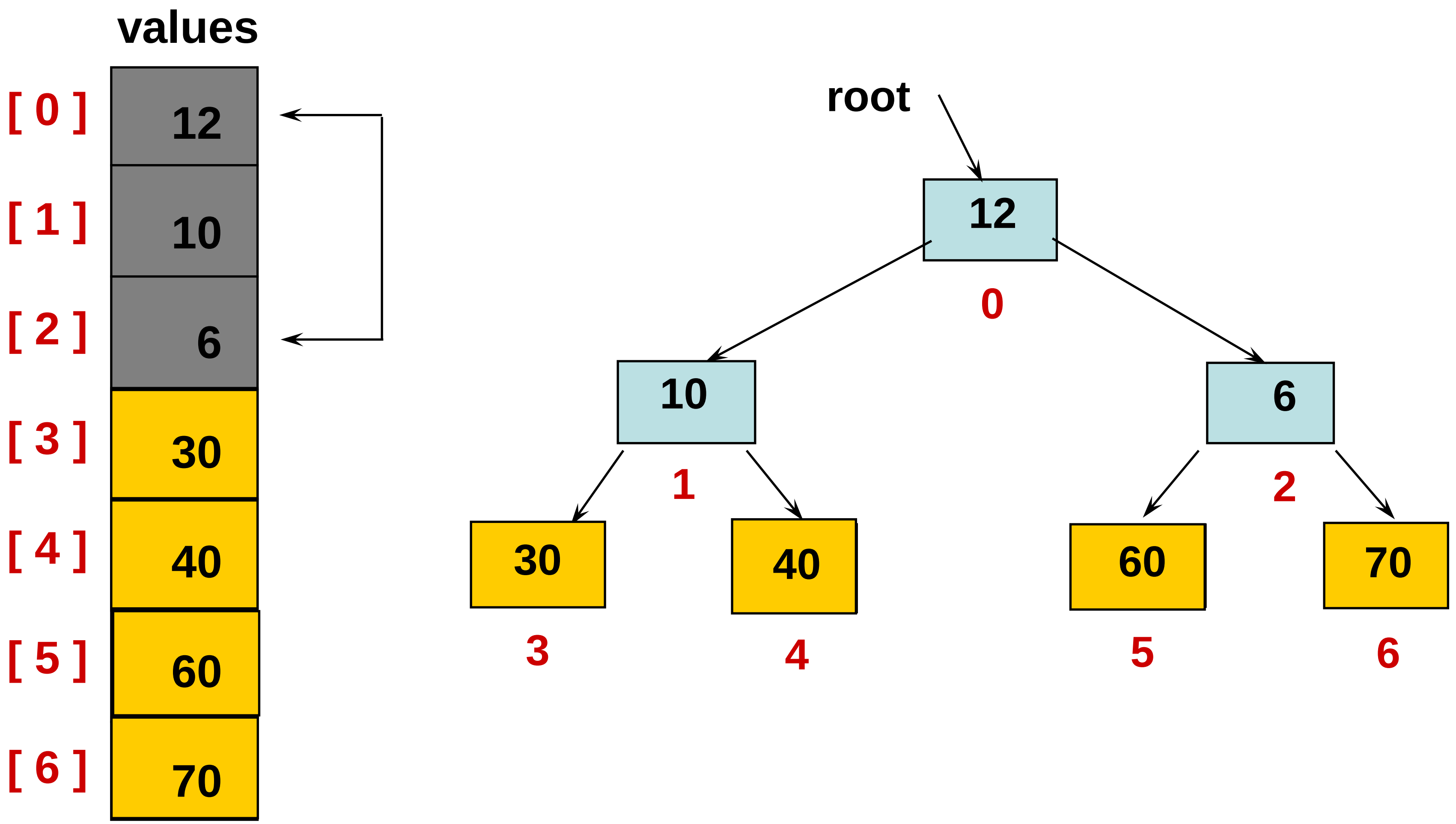
After reheaping remaining unsorted data



86

sorting

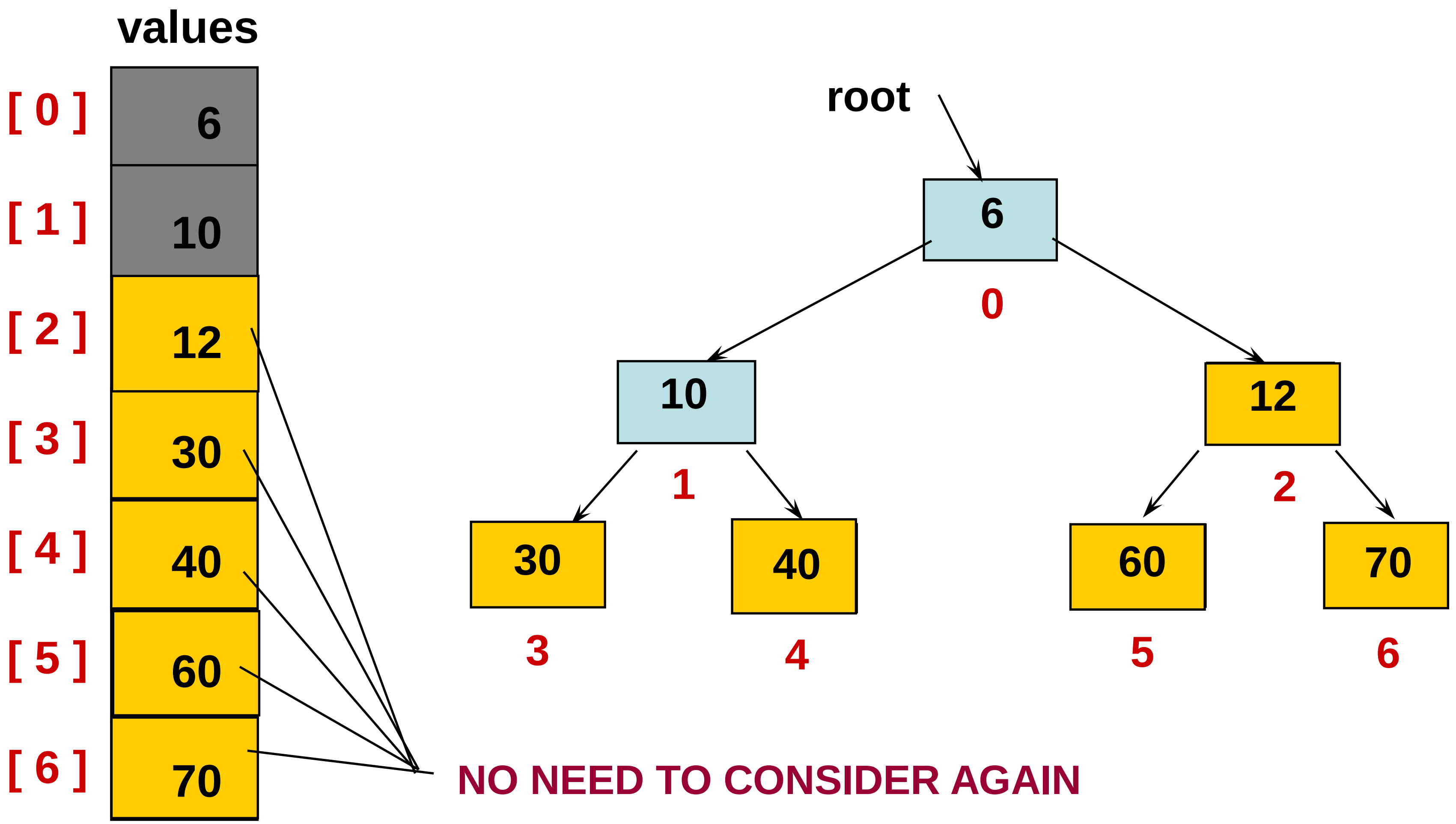
Swap root element into last place in unsorted array



87

sorting

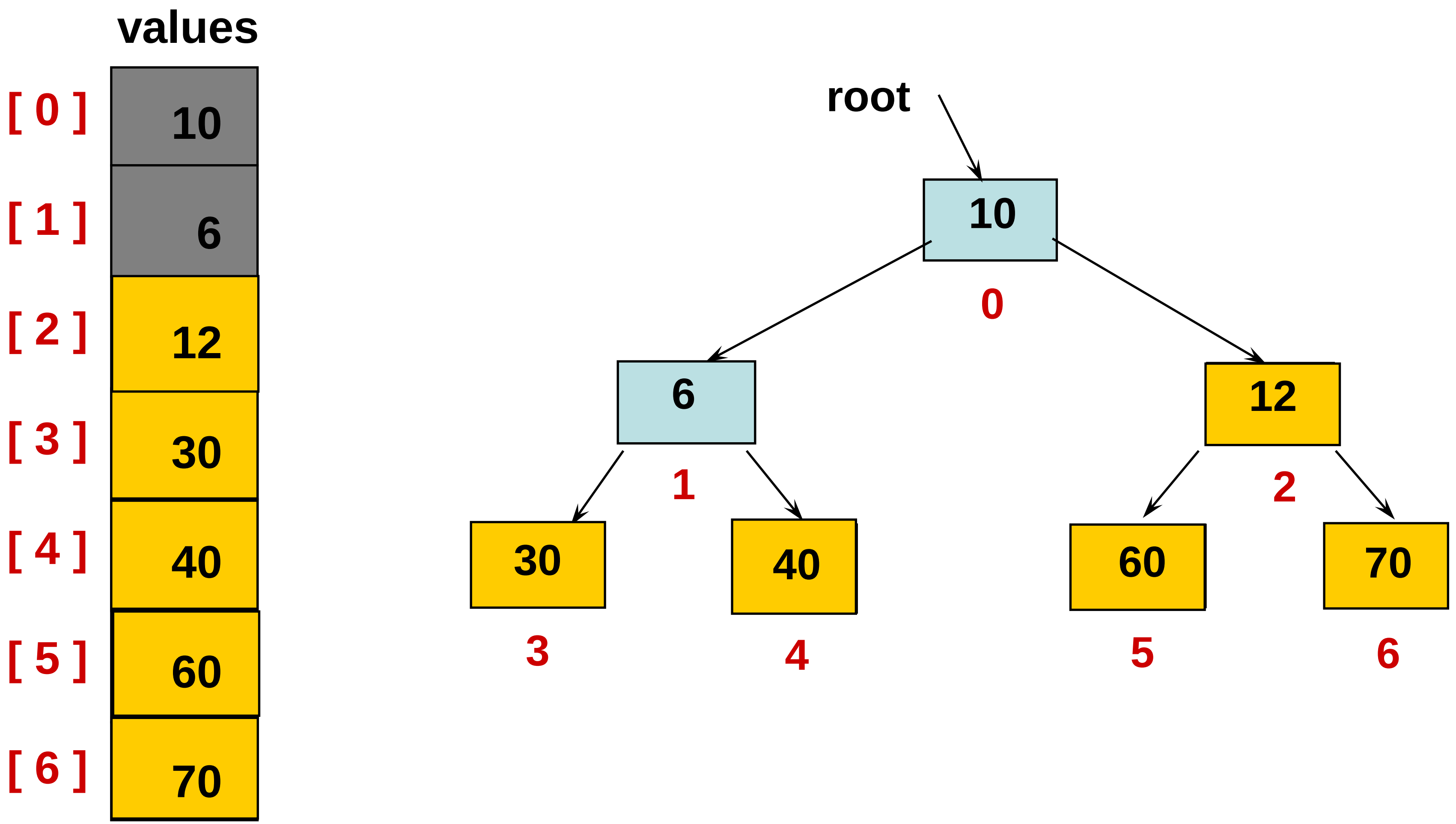
After swapping root element into its place



88

sorting

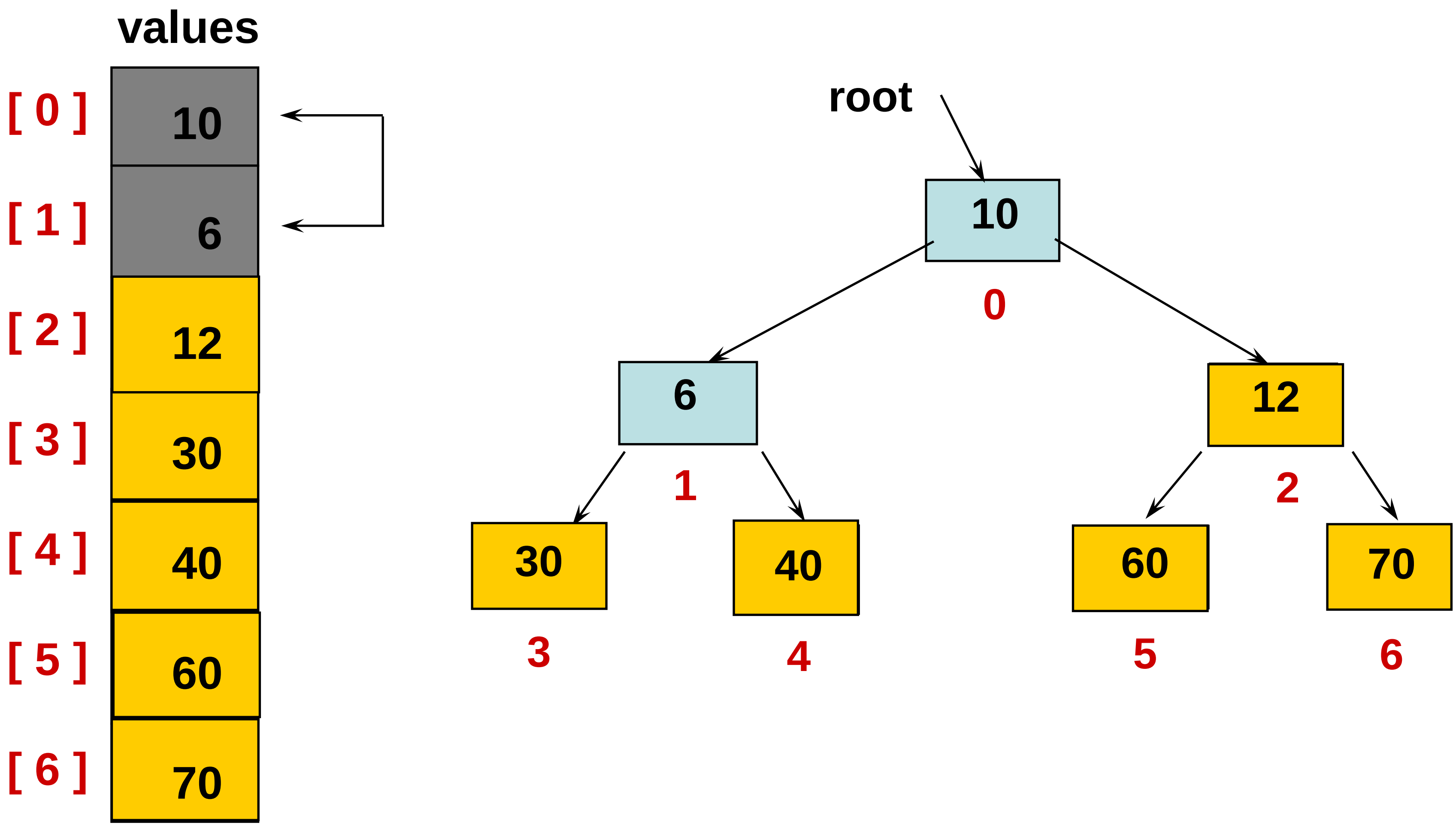
After reheaping remaining unsorted data



89

sorting

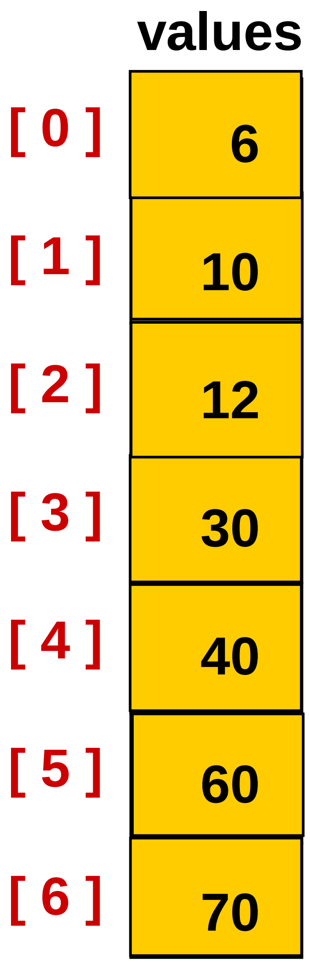
Swap root element into last place in unsorted array

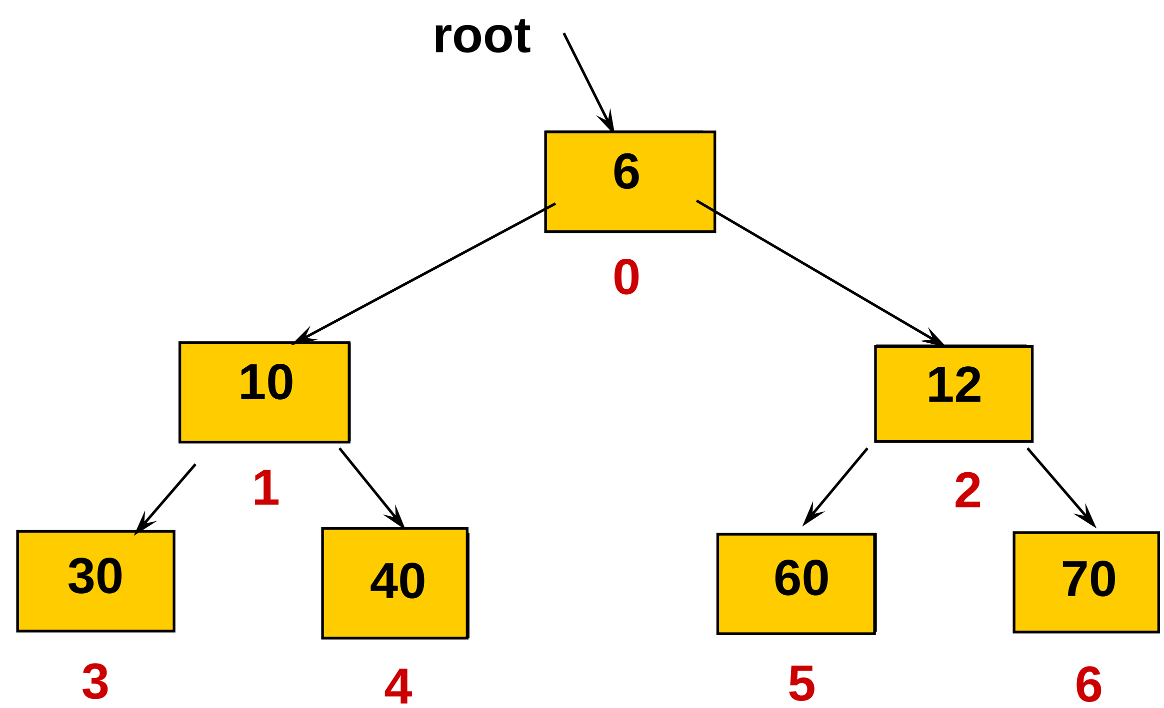


90

sorting

After swapping root element into its place





ALL data ARE SORTED

91

Heap sort

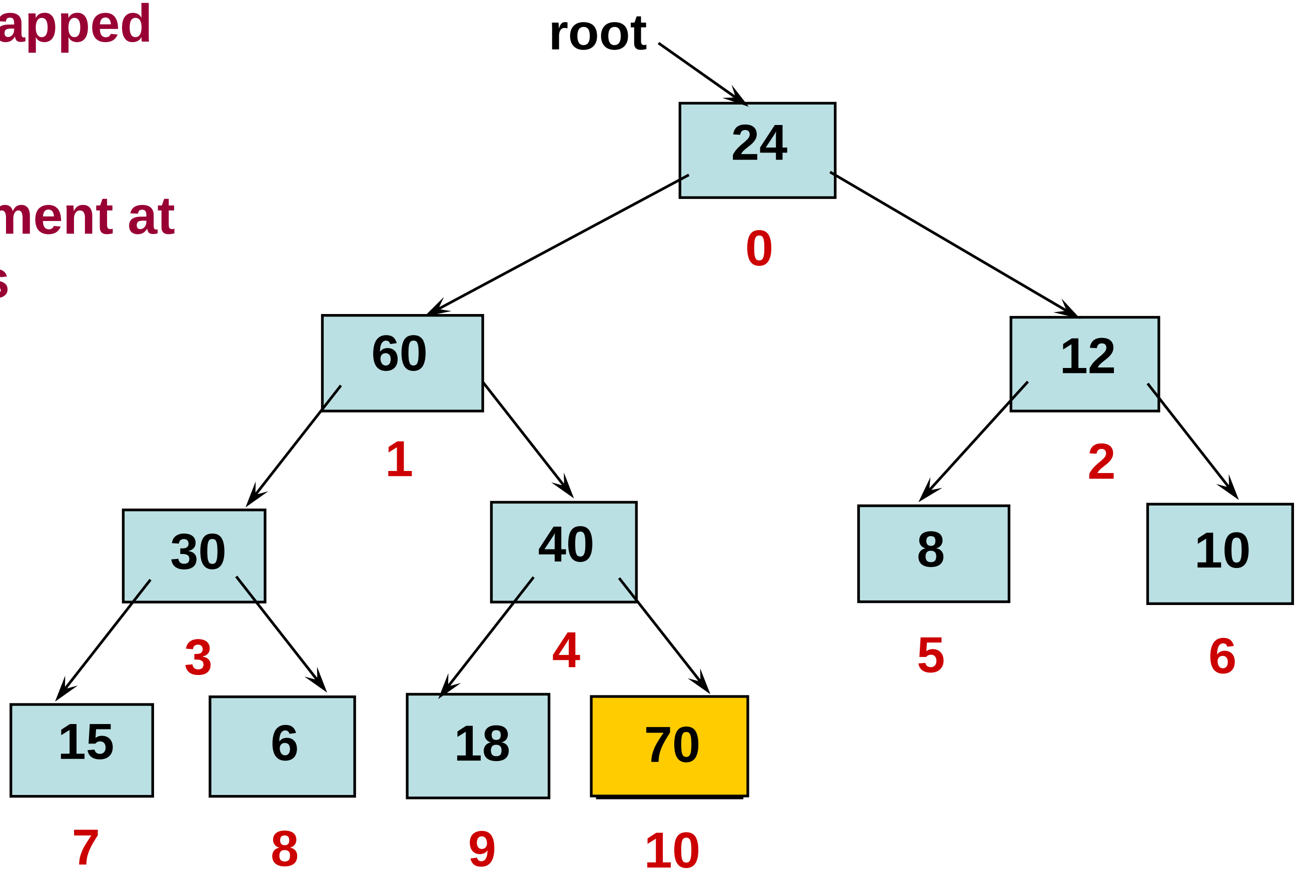
```
template <class  ItemType >
void  HeapSort  ( ItemType  values [ ] ,   int  numValues )
//  Post: Sorts array values[ 0 . . numValues-1 ] into ascending
//            order by key
{
    int  index ;

    // Convert array  values[ 0 . . numValues-1 ]  into a heap. Build 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 ) ;
    }
}
```


92

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.



93

Heap Sort of N data:

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

Operation	Linked List	Binary
make-heap	1	1
insert	1	log N
find-min	N	1
delete-min	N	log N
union	1	N
decrease-key	1	log N
delete	N	log N
is-empty	1	1

Practice Questions

- Write following functions
  - decreaseKey
    - The decreaseKey(p,) operation lowers the value of the item at position p by a positive amount .
    - This might violate the heap order, it must be fixed by a *percolate up.*
    - This operation could be useful to system administrators: They can make their programs run with highest priority
  - increaseKey
    - The increaseKey(p,) operation increases the value of the item at position p by a positive amount .
    - This is done with a *percolate down.*
    - Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Practice Questions

1. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2,
   one at a time, into an initially empty binary heap.
   1. b. Show the result of using the linear-time algorithm to build a binary heap using the same input.
   2. Show the result of performing three deleteMin operations in the heap

2. Find second min in min heap
3. Find third min in a min heap
4. Convert min heap to max heap
5. How to determine if the given array is a binary heap
6. Check if the binary tree is a binary heap

Practice Questions

- Show the following regarding the maximum item in the Minheap:
  a. It must be at one of the leaves.
  b. There are exactly *N*/2 leaves.
  c. Every leaf must be examined to find it
    - How much time do you need to visit all leaf nodes to find maximum?
- * Give an algorithm to find all nodes less than some value, *X*, in a binary heap. Your algorithm should run in $O(K)$, where *K* is the number of nodes output.
- Suppose binary heaps are represented using explicit links. Give algorithm to find a tree node that is implicit position i