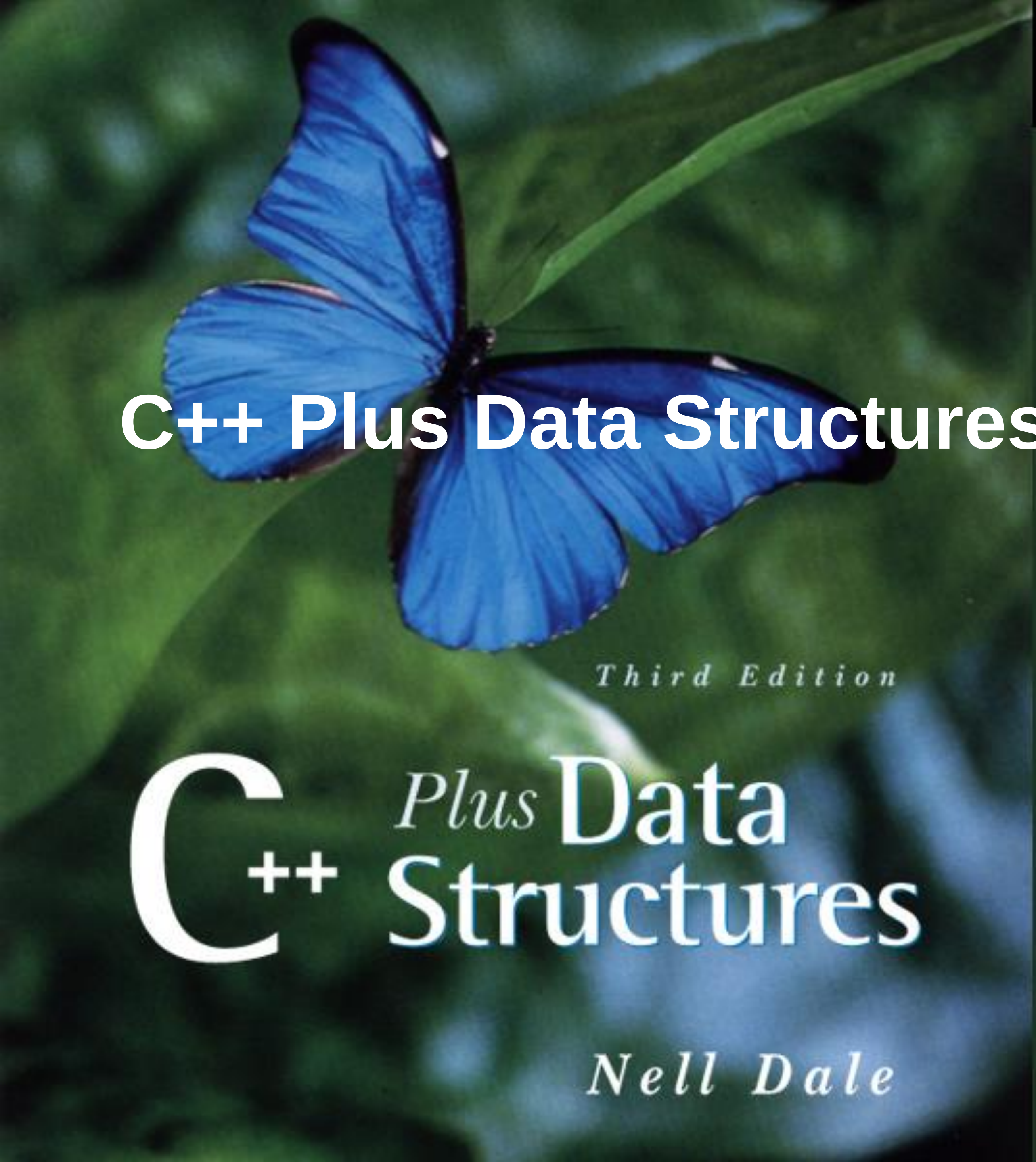
# C++ Plus Data Structures




Chapter

***10***
***Sorting and Searching Algorithms***

# Sorting means . . .

The values stored in an array have keys of a type for which the relational operators are defined. (We also assume unique keys.)

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

# Straight Selection Sort

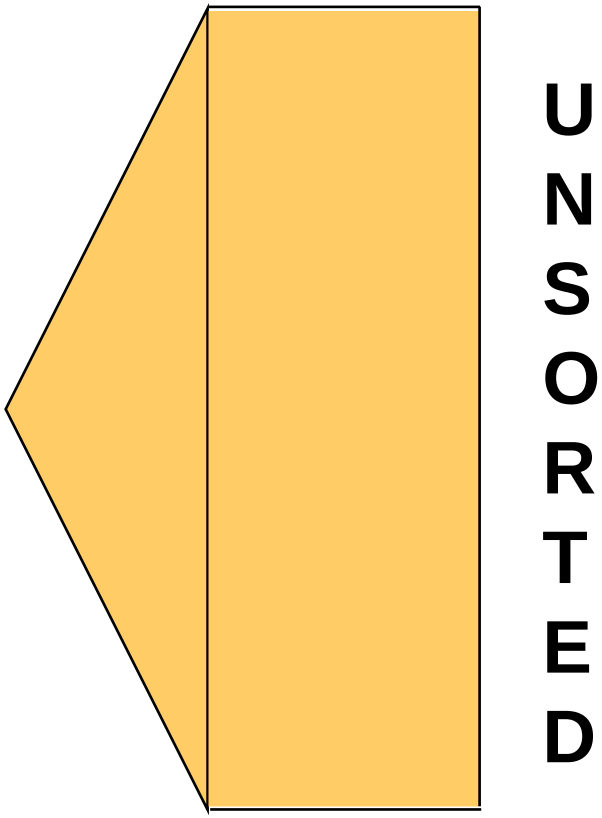
| | | |
|---|---|---|
| values | [ 0 ] | 36 |
| | [ 1 ] | 24 |
| | [ 2 ] | 10 |
| | [ 3 ] | 6 |
| | [ 4 ] | 12 |

**Divides the array into two parts: already sorted, and not yet sorted.**

**On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.**

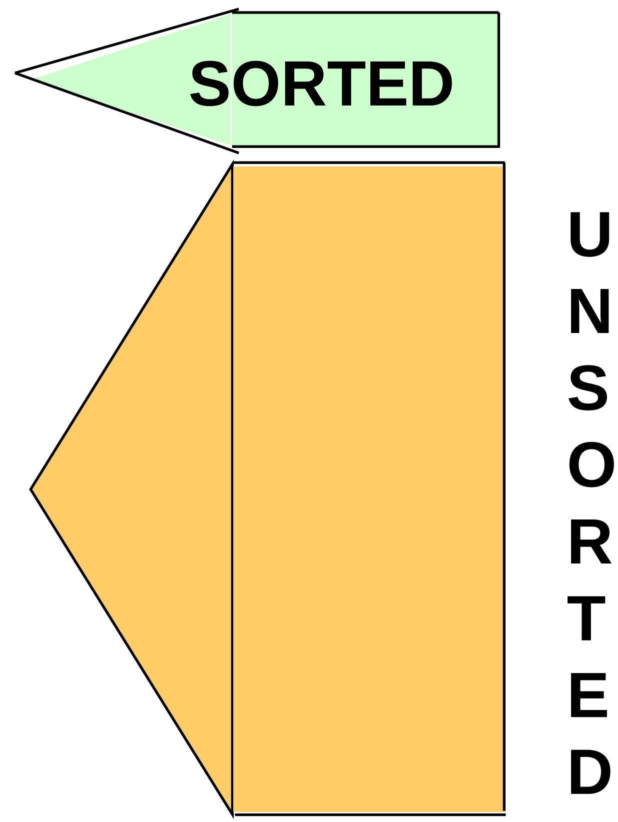
# Selection Sort: Pass One

| | | |
|---|---|---|
| values | [ 0 ] | 36 |
| | [ 1 ] | 24 |
| | [ 2 ] | 10 |
| | [ 3 ] | 6 |
| | [ 4 ] | 12 |

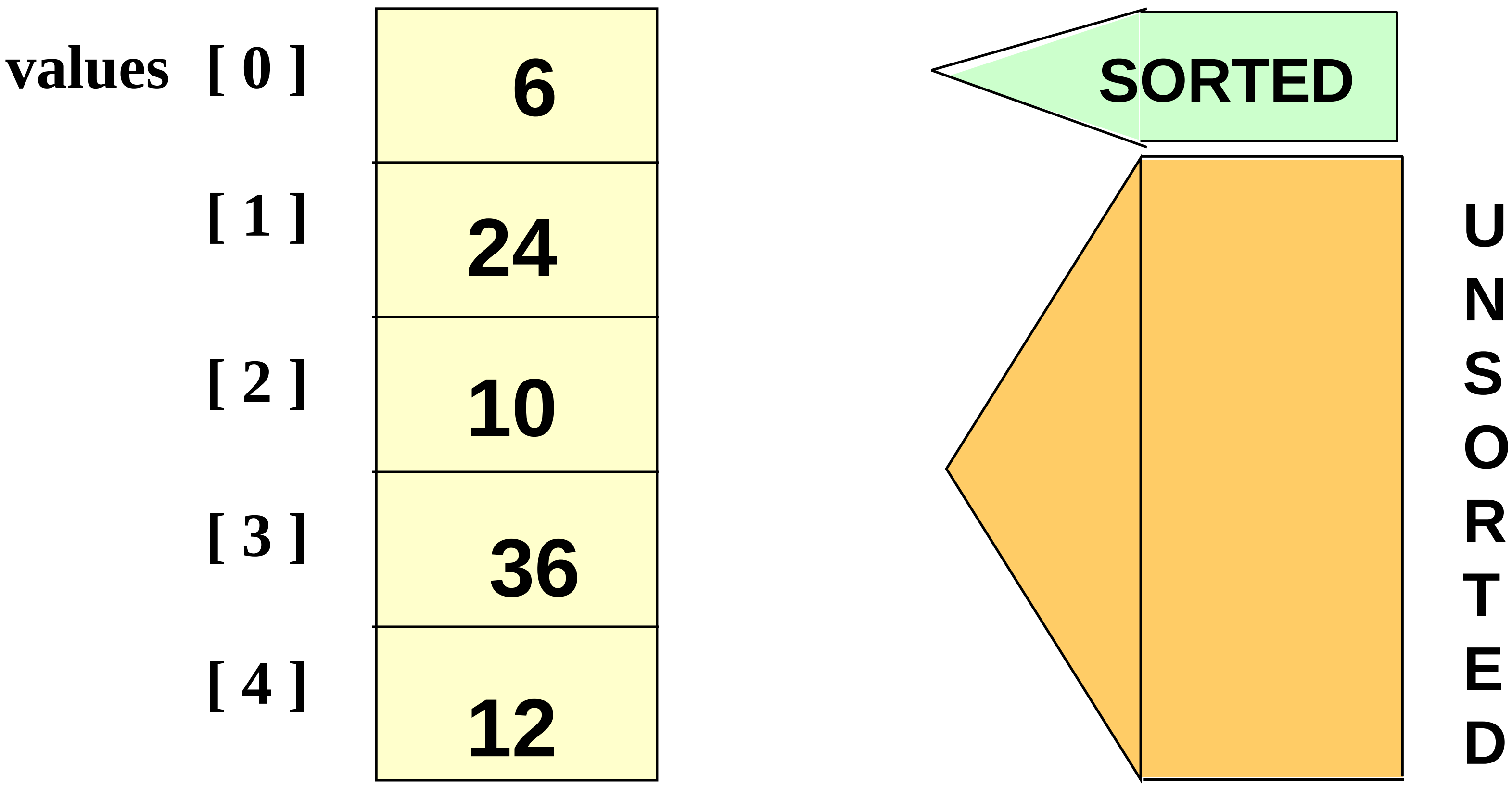


# Selection Sort: End Pass One

| values | |
|---|---|
| [ 0 ] | 6 |
| [ 1 ] | 24 |
| [ 2 ] | 10 |
| [ 3 ] | 36 |
| [ 4 ] | 12 |

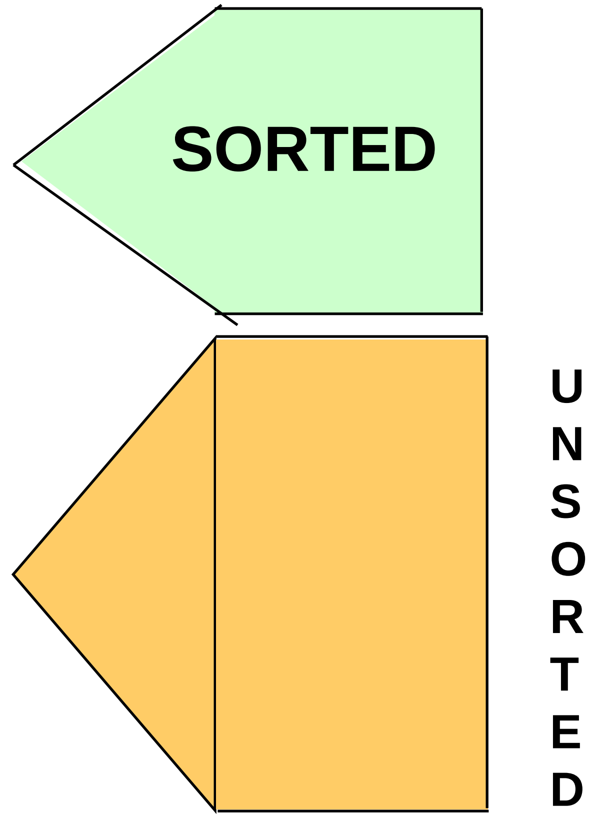


# Selection Sort: Pass Two



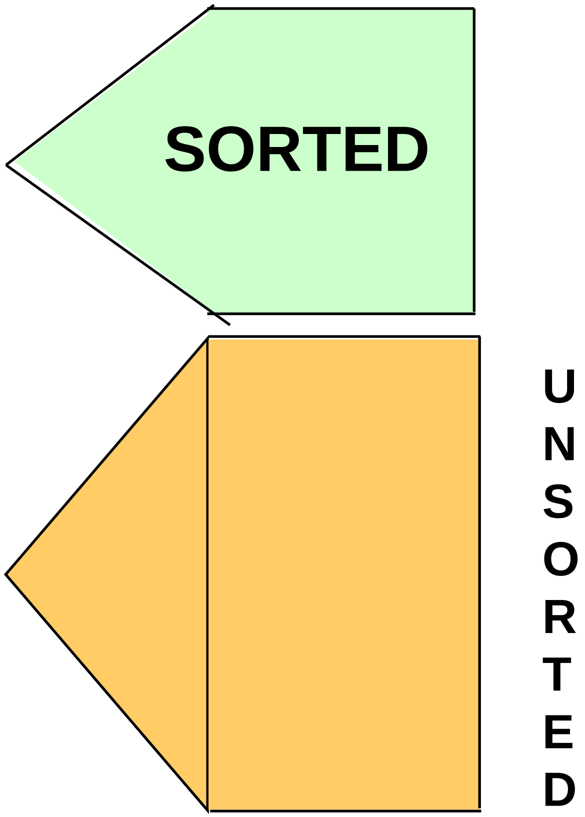
# Selection Sort: End Pass Two

| | | |
|---|---|---|
| values | [ 0 ] | 6 |
| | [ 1 ] | 10 |
| | [ 2 ] | 24 |
| | [ 3 ] | 36 |
| | [ 4 ] | 12 |



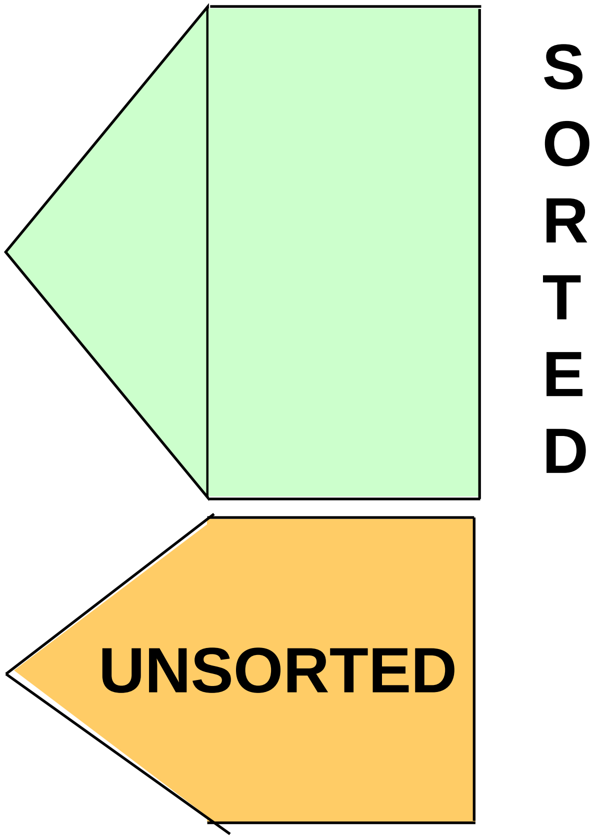
# Selection Sort: Pass Three

| | | |
|---|---|---|
| values | [ 0 ] | 6 |
| | [ 1 ] | 10 |
| | [ 2 ] | 24 |
| | [ 3 ] | 36 |
| | [ 4 ] | 12 |

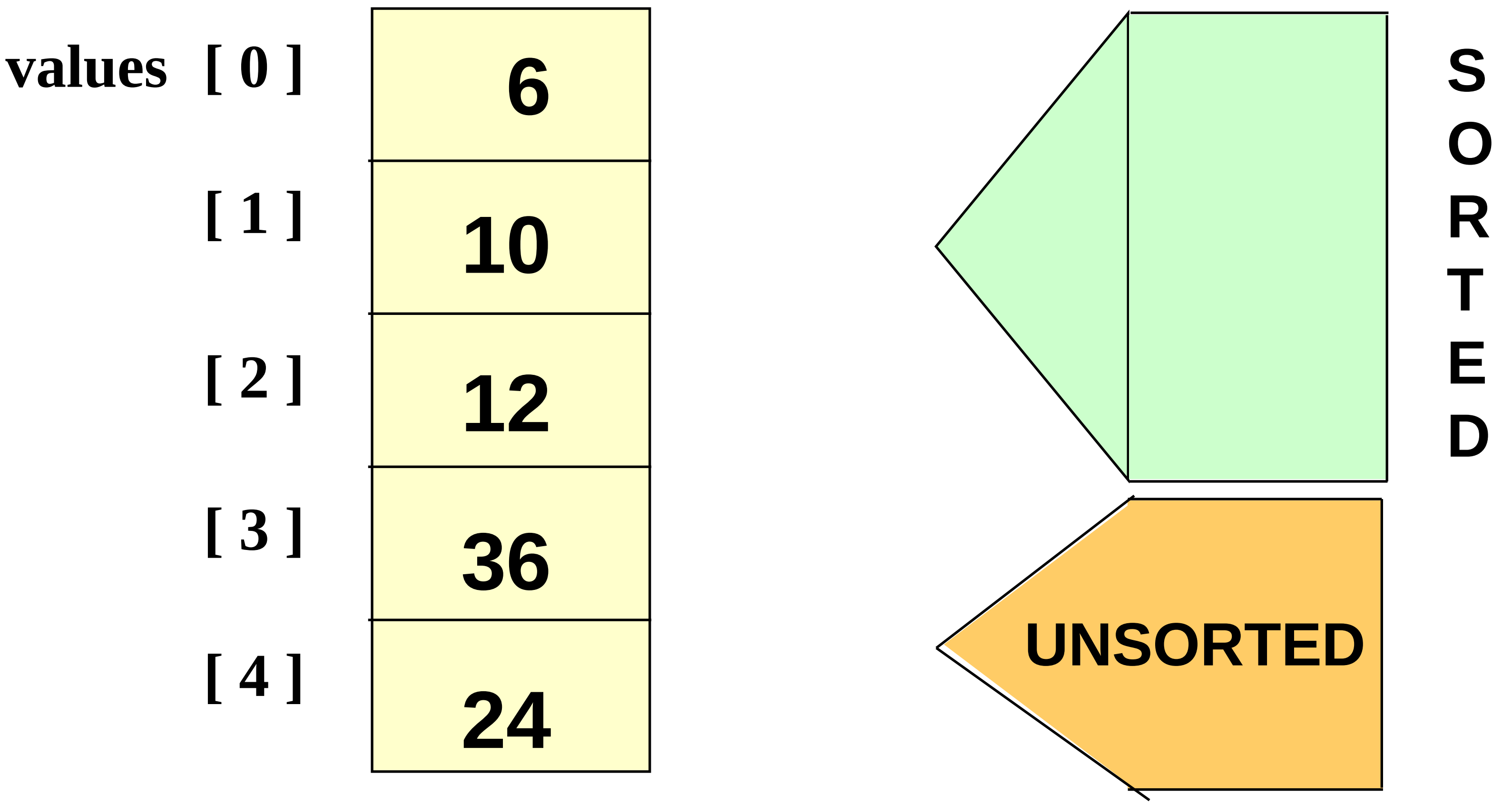


# Selection Sort: End Pass Three

| | | |
|---|---|---|
| values | [ 0 ] | 6 |
| | [ 1 ] | 10 |
| | [ 2 ] | 12 |
| | [ 3 ] | 36 |
| | [ 4 ] | 24 |

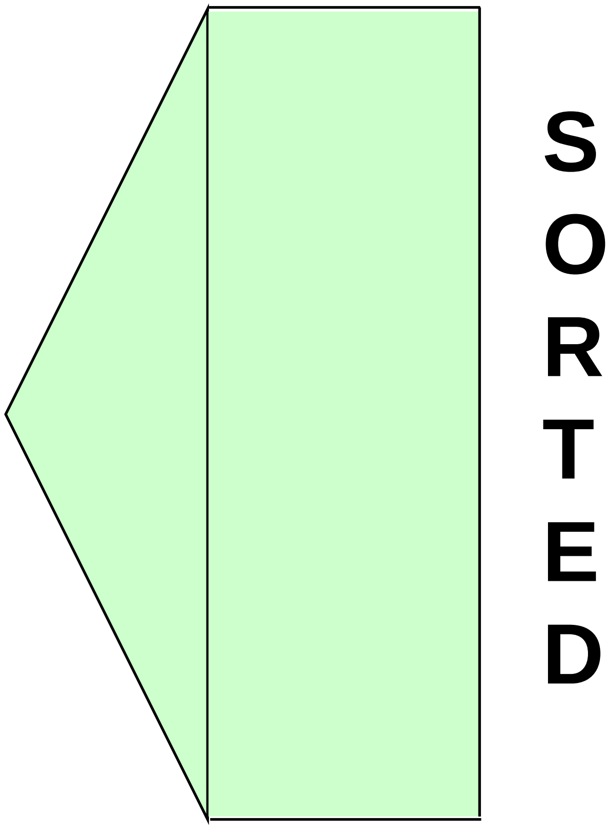


# Selection Sort: Pass Four



# Selection Sort: End Pass Four

| values | |
|---|---|
| [ 0 ] | 6 |
| [ 1 ] | 10 |
| [ 2 ] | 12 |
| [ 3 ] | 24 |
| [ 4 ] | 36 |



# Selection Sort: How many comparisons?

| values | |
|---|---|
| [ 0 ] | 6 |
| [ 1 ] | 10 |
| [ 2 ] | 12 |
| [ 3 ] | 24 |
| [ 4 ] | 36 |

4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

1 compare for values[3]

= 4 + 3 + 2 + 1

# For selection sort in general

**The number of comparisons when the array contains N elements is**

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$

# Notice that . . .

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$

$$+ \ \text{Sum} = 1 + 2 + \ldots + (N-2) + (N-1)$$

---

$$2^* \ \text{Sum} = N + N + \ldots + N + N$$

$$2 * \text{Sum} = N * (N-1)$$

$$\boxed{\text{Sum} = \frac{N * (N-1)}{2}}$$

# For selection sort in general

**The number of comparisons when the array contains N elements is**

**Sum = (N-1) + (N-2) + . . . + 2 + 1**

**Sum = N * (N-1) /2**

**Sum = .5 $N^2$ - .5 N**

**Sum = O($N^2$)**

```
template <class  ItemType >
int  MinIndex(ItemType values [ ], int  start, int end)
//  Post: Function value = index of the smallest value
//  in values [start]  . . values [end].
{
   int  indexOfMin = start ;

   for(int index = start + 1 ; index <= end ; index++)
     if  (values[ index] < values [indexOfMin])
         indexOfMin = index ;

   return   indexOfMin;

}
```

```
template <class  ItemType >
void  SelectionSort (ItemType values[ ],
  int  numValues )

// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key
{
   int  endIndex = numValues - 1 ;

   for (int current = 0 ; current < endIndex;
     current++)

     Swap (values[current],
       values[MinIndex(values,current, endIndex)]);

}
```

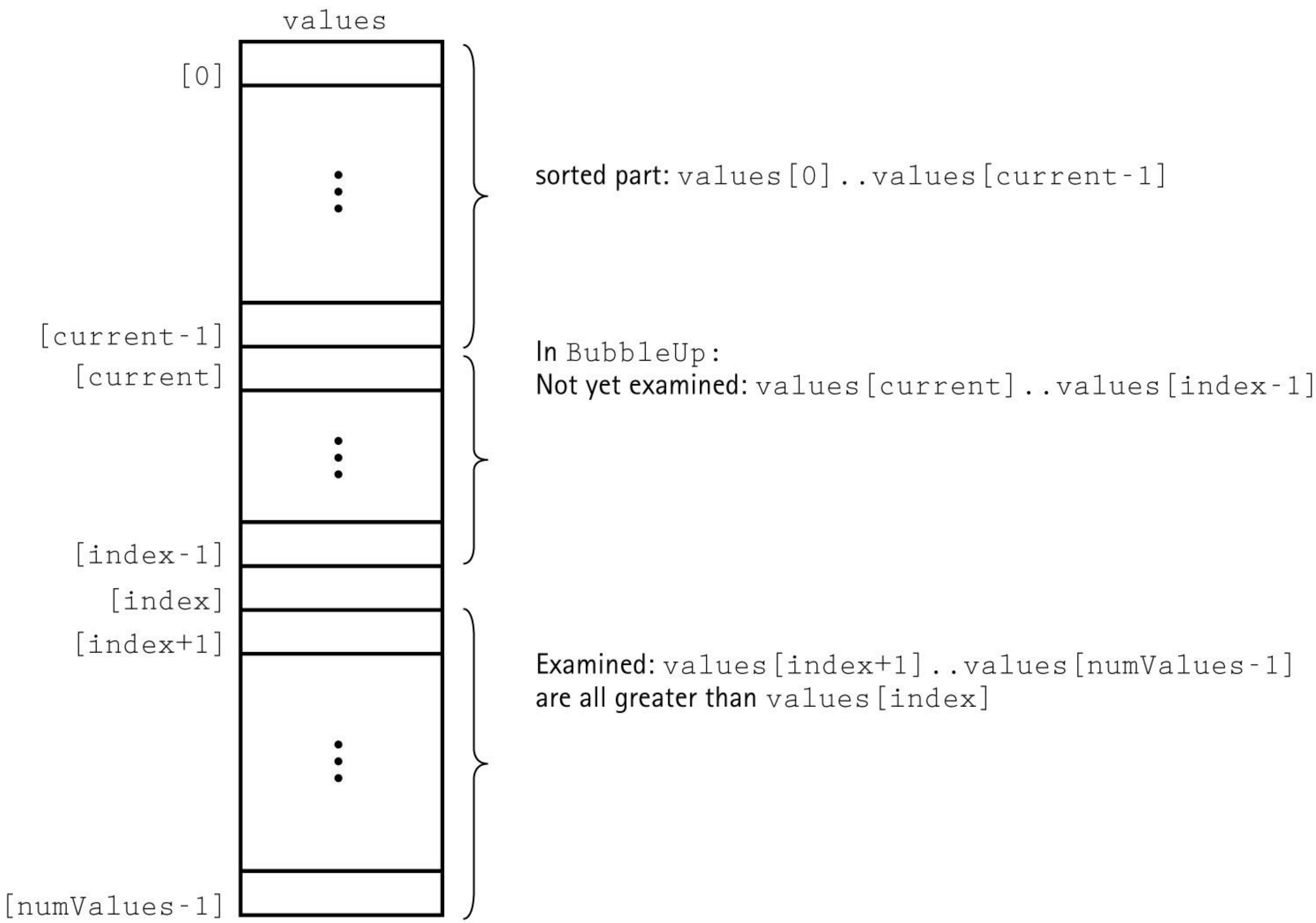
# Bubble Sort

| | | |
|---|---|---|
| values | [ 0 ] | 36 |
| | [ 1 ] | 24 |
| | [ 2 ] | 10 |
| | [ 3 ] | 6 |
| | [ 4 ] | 12 |

**Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.**

**On each pass, this causes the smallest element to “bubble up” to its correct place in the array.**

# Snapshot of BubbleSort



# Code for BubbleSort

```
template<class ItemType>
void BubbleSort(ItemType values[],
  int numValues)
{
  int current = 0;
  while (current < numValues - 1)
  {
    BubbleUp(values, current, numValues-1);
    current++;
  }
}
```

# Code for BubbleUp

```
template<class ItemType>
void BubbleUp(ItemType values[],
  int startIndex, int endIndex)
// Post: Adjacent pairs that are out of
//    order have been switched between
//    values[startIndex]..values[endIndex]
//    beginning at values[endIndex].

{
  for (int index = endIndex;
    index > startIndex; index--)
    if (values[index] < values[index-1])
      Swap(values[index], values[index-1]);
}
```

# Observations on BubbleSort

This algorithm is *always* $O(N^2)$.

There can be a large number of intermediate swaps.

**Can this algorithm be improved?**

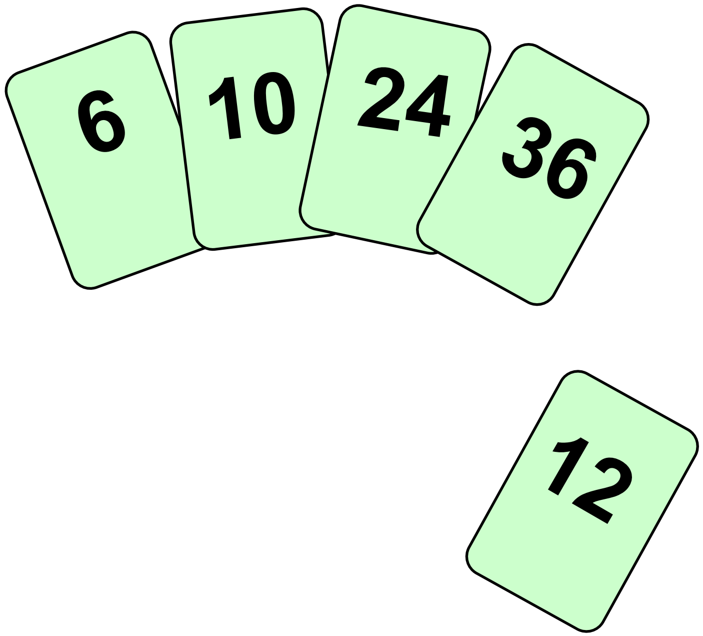
# Insertion Sort

| | | |
|---|---|---|
| values | [ 0 ] | 36 |
| | [ 1 ] | 24 |
| | [ 2 ] | 10 |
| | [ 3 ] | 6 |
| | [ 4 ] | 12 |

**One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.**

**On each pass, this causes the number of already sorted elements to increase by one.**

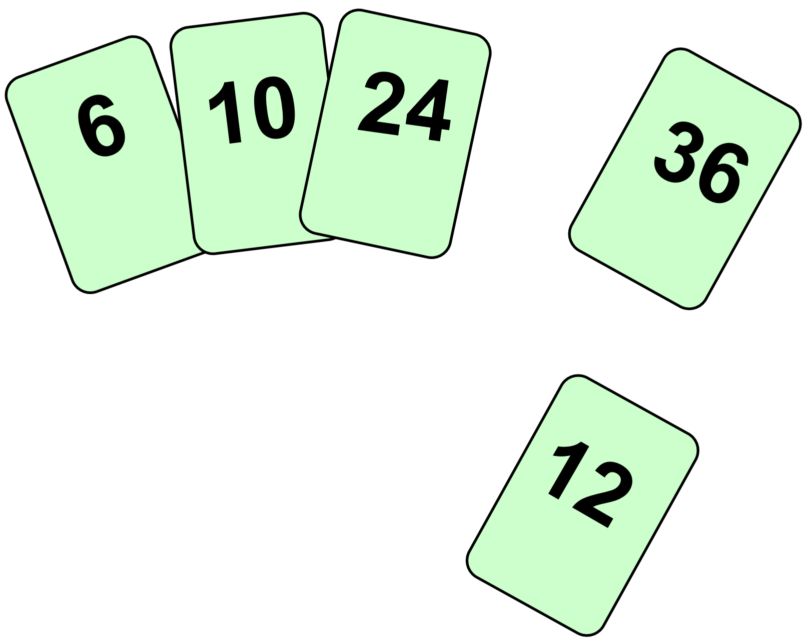
# Insertion Sort




**Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.**

**To insert 12, we need to make room for it by moving first 36 and then 24.**

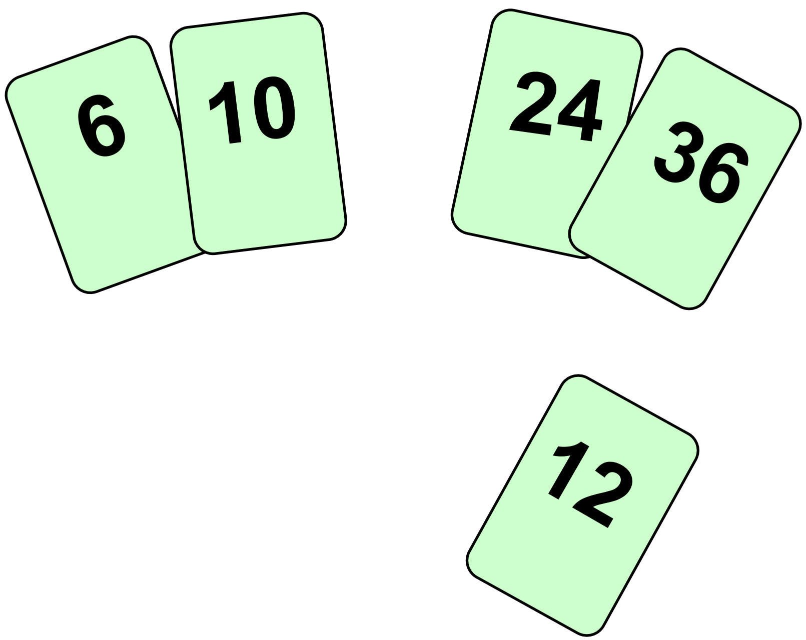
# Insertion Sort




Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

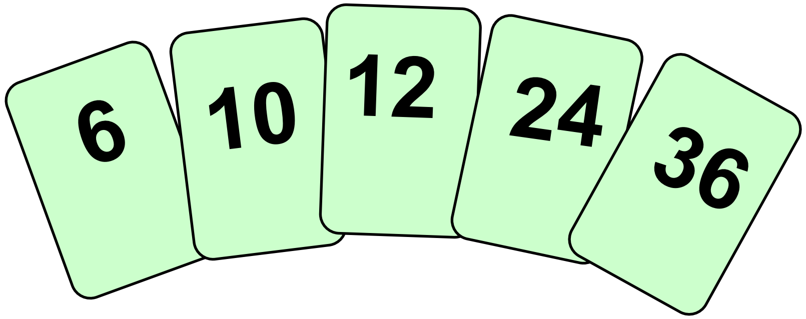
# Insertion Sort




Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

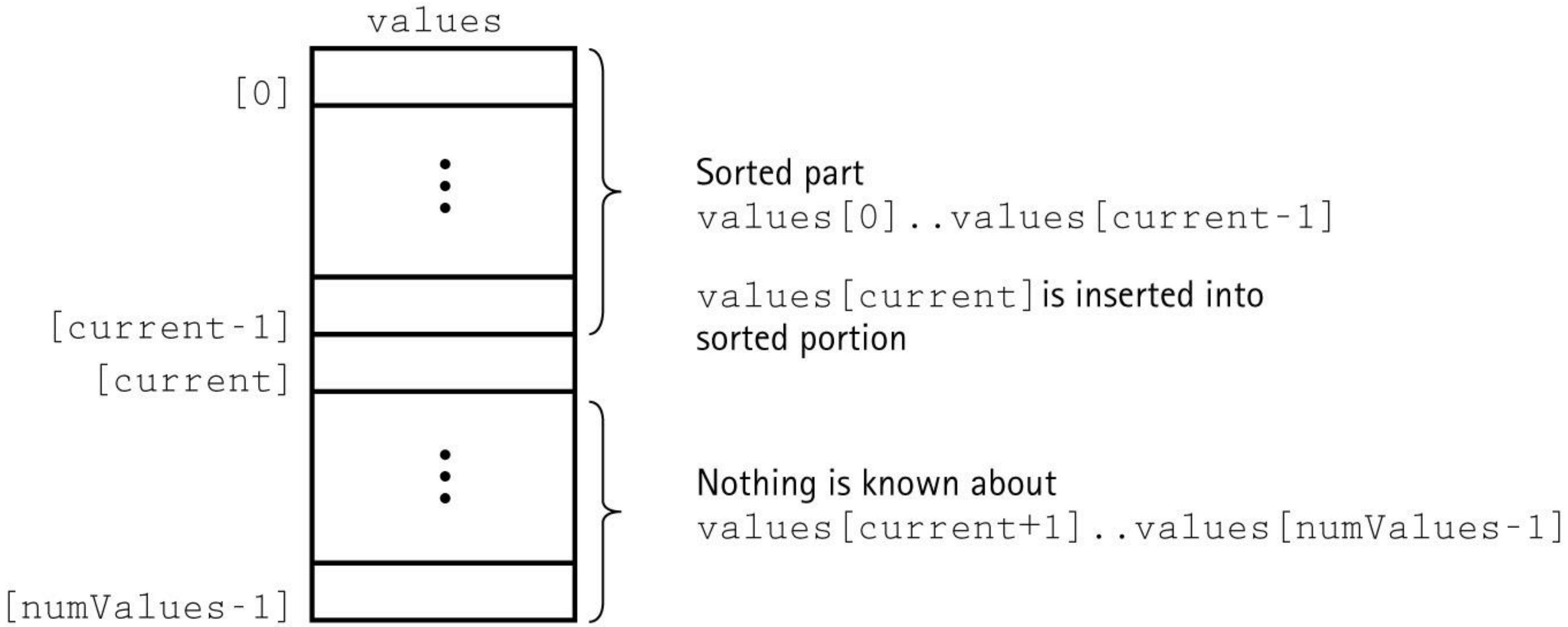
# Insertion Sort




**Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.**

**To insert 12, we need to make room for it by moving first 36 and then 24.**

# A Snapshot of the Insertion Sort Algorithm



```
template <class  ItemType >
void InsertItem  ( ItemType  values [ ] ,   int  start ,
   int end )
//  Post: Elements between values[start] and  values
//   [end] have been sorted into ascending order by key.
{
   bool  finished = false ;
   int   current  =  end ;
   bool  moreToSearch = (current != start);

   while (moreToSearch  &&  !finished )
   {
     if  (values[current] < values[current - 1])
       {
        Swap(values[current], values[current - 1);
        current--;
        moreToSearch = ( current != start );
       }
     else
        finished = true ;
   }
}
```

```
template <class  ItemType >
void  InsertionSort  ( ItemType  values [ ] ,
  int  numValues )

//  Post: Sorts array values[0 . . numValues-1 ] into
//   ascending order by key
{
   for (int count = 0 ; count < numValues; count++)

     InsertItem ( values , 0 , count ) ;
}
```

# Sorting Algorithms and Average Case Number of Comparisons

**Simple Sorts**

- Straight Selection Sort
- Bubble Sort
- Insertion Sort

$O(N^2)$

**More Complex Sorts**

- Quick Sort
- Merge Sort
- Heap Sort

$O(N*\log N)$

# Recall that . . .

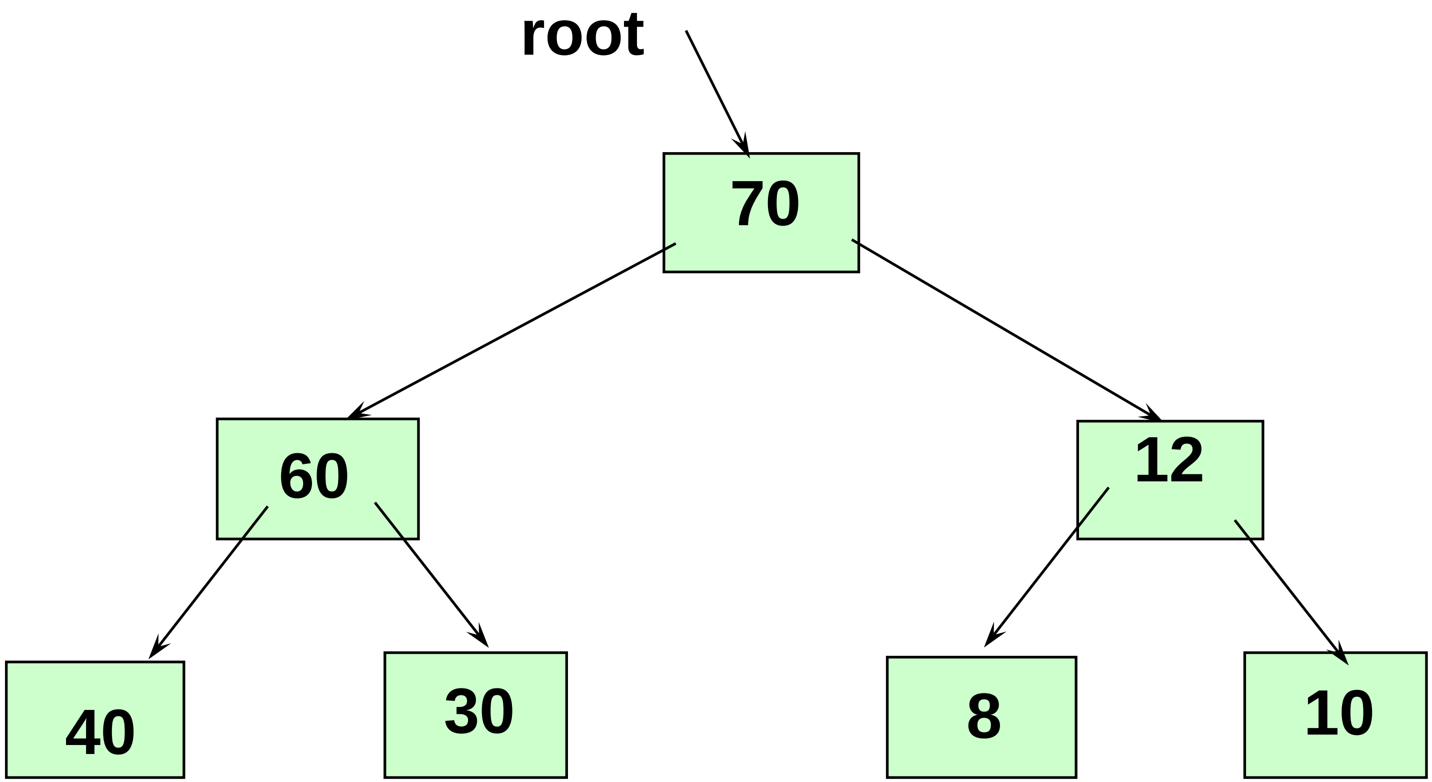
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

**Its shape must be a complete binary tree.**

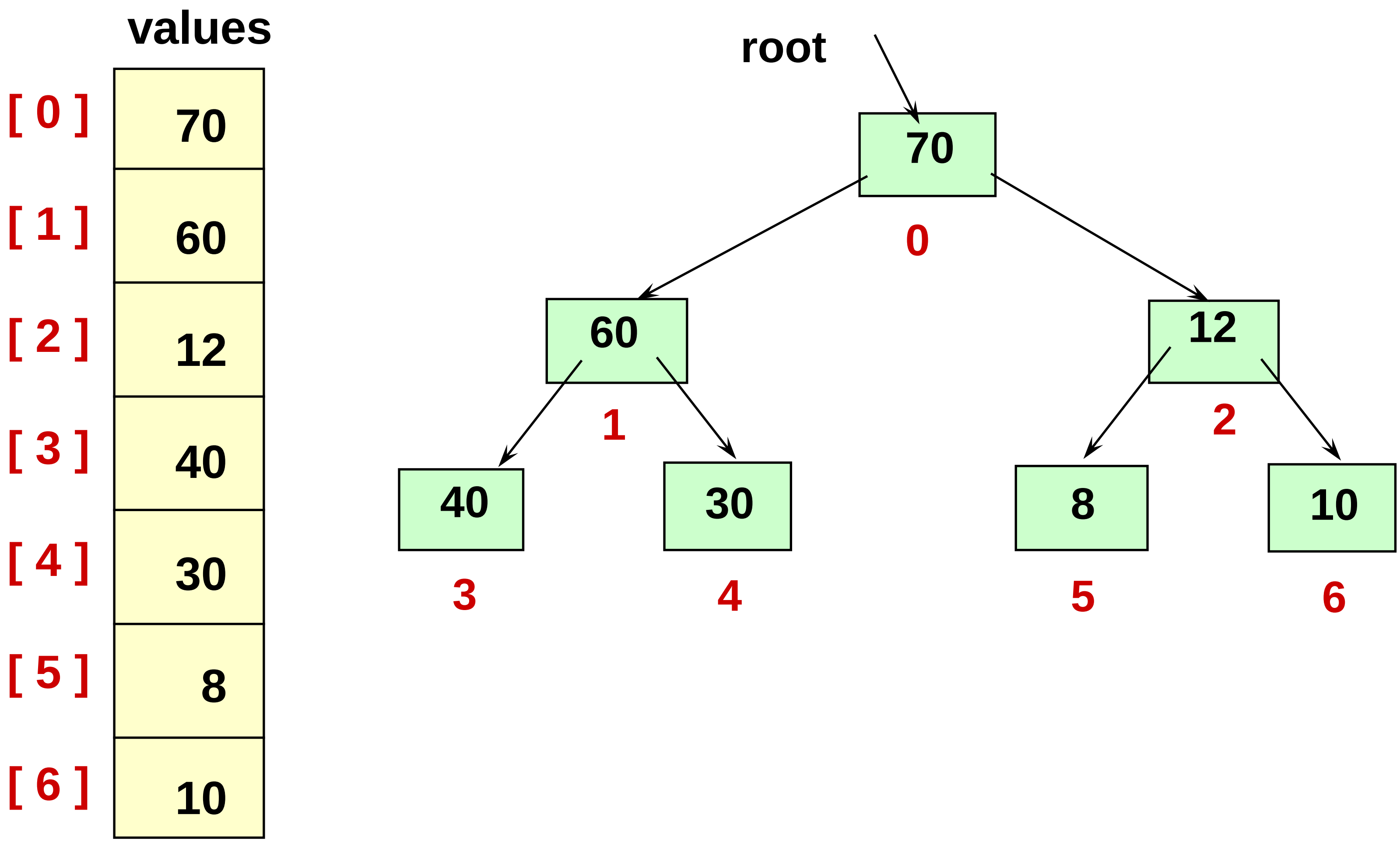
**For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

# The largest element in a heap

## is always found in the root node



# The heap can be stored in an array



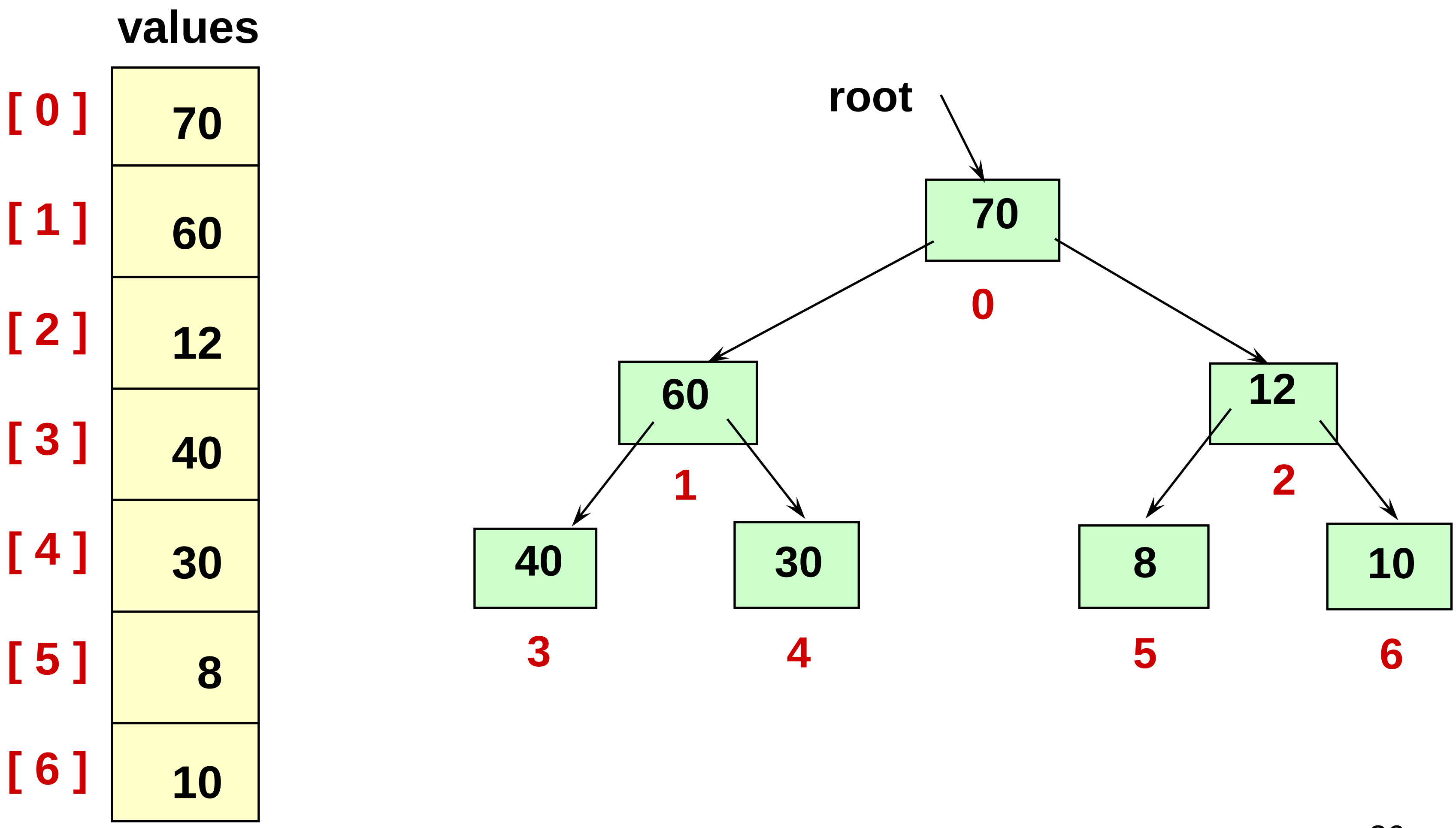
# Heap Sort Approach

**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.**

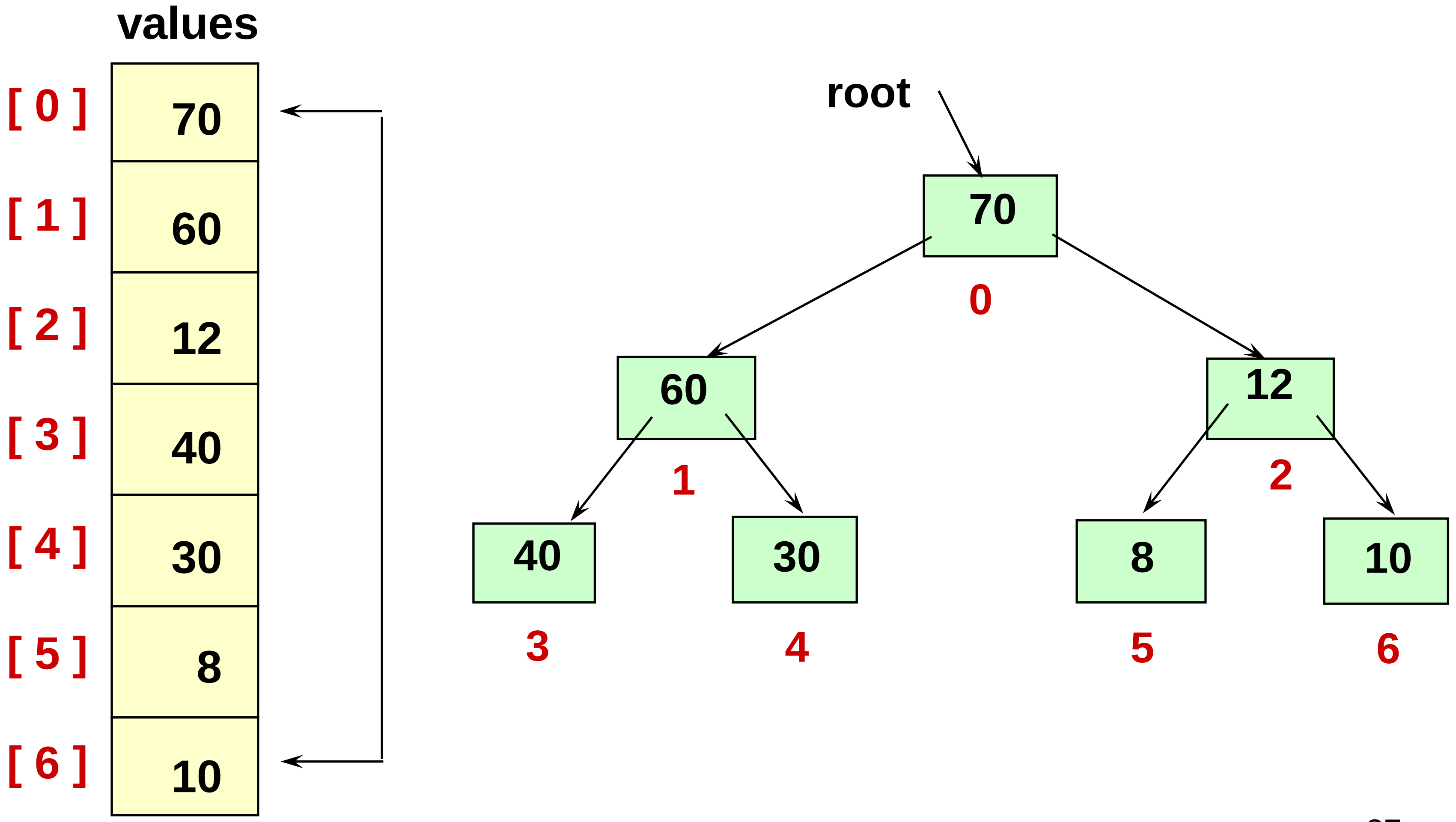
**Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.**

**Reheap the remaining unsorted elements. (This puts the next-largest element into the root position).**

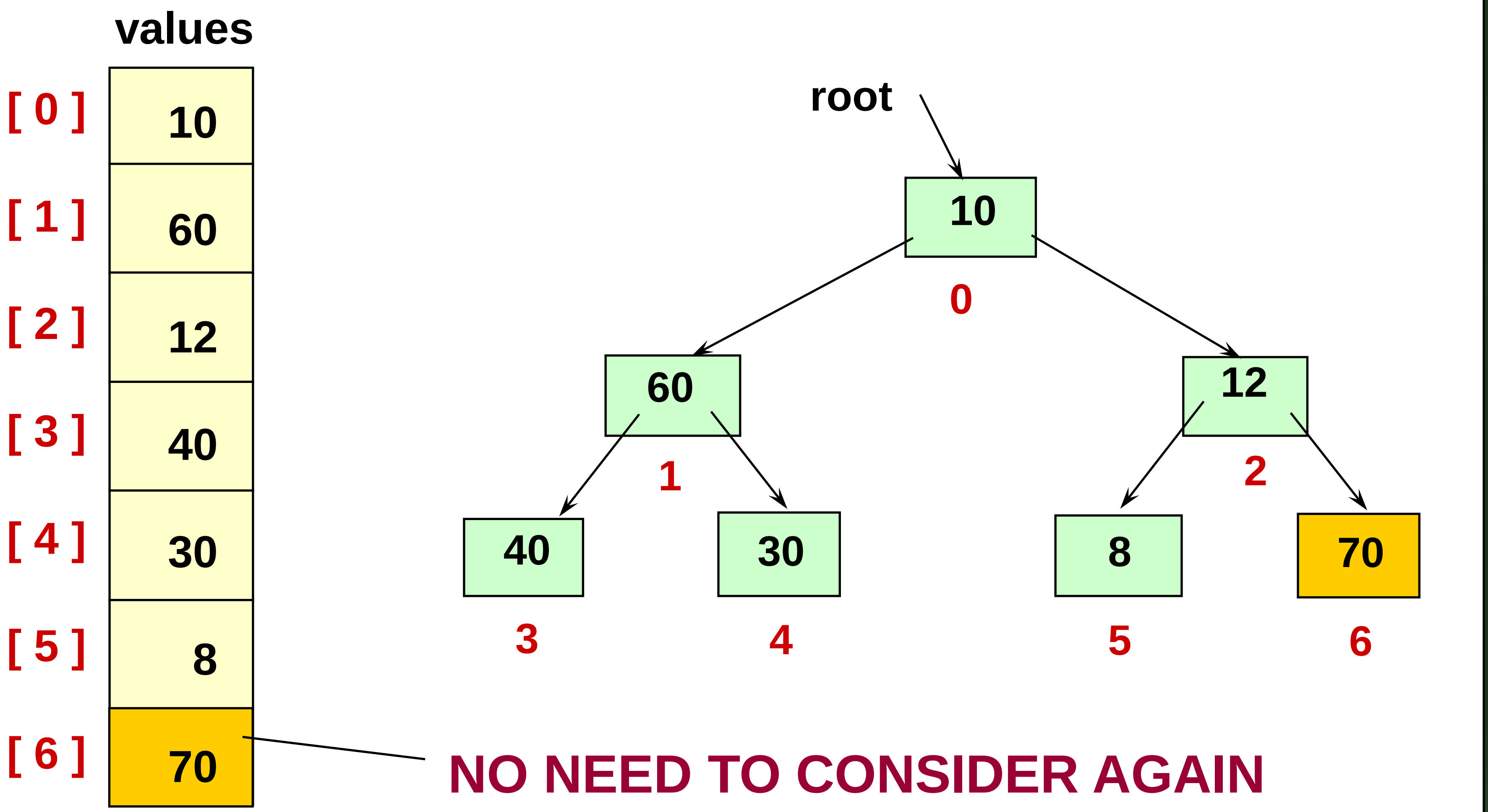
# After creating the original heap



# Swap root element into last place in unsorted array

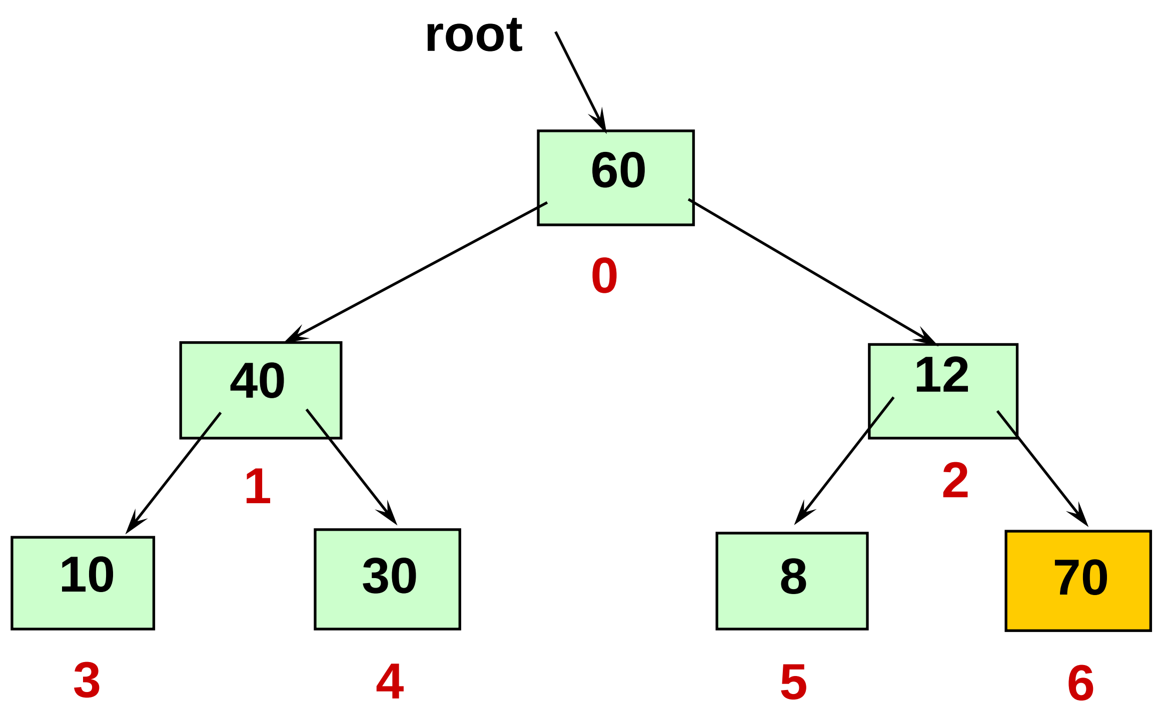


# After swapping root element into it place

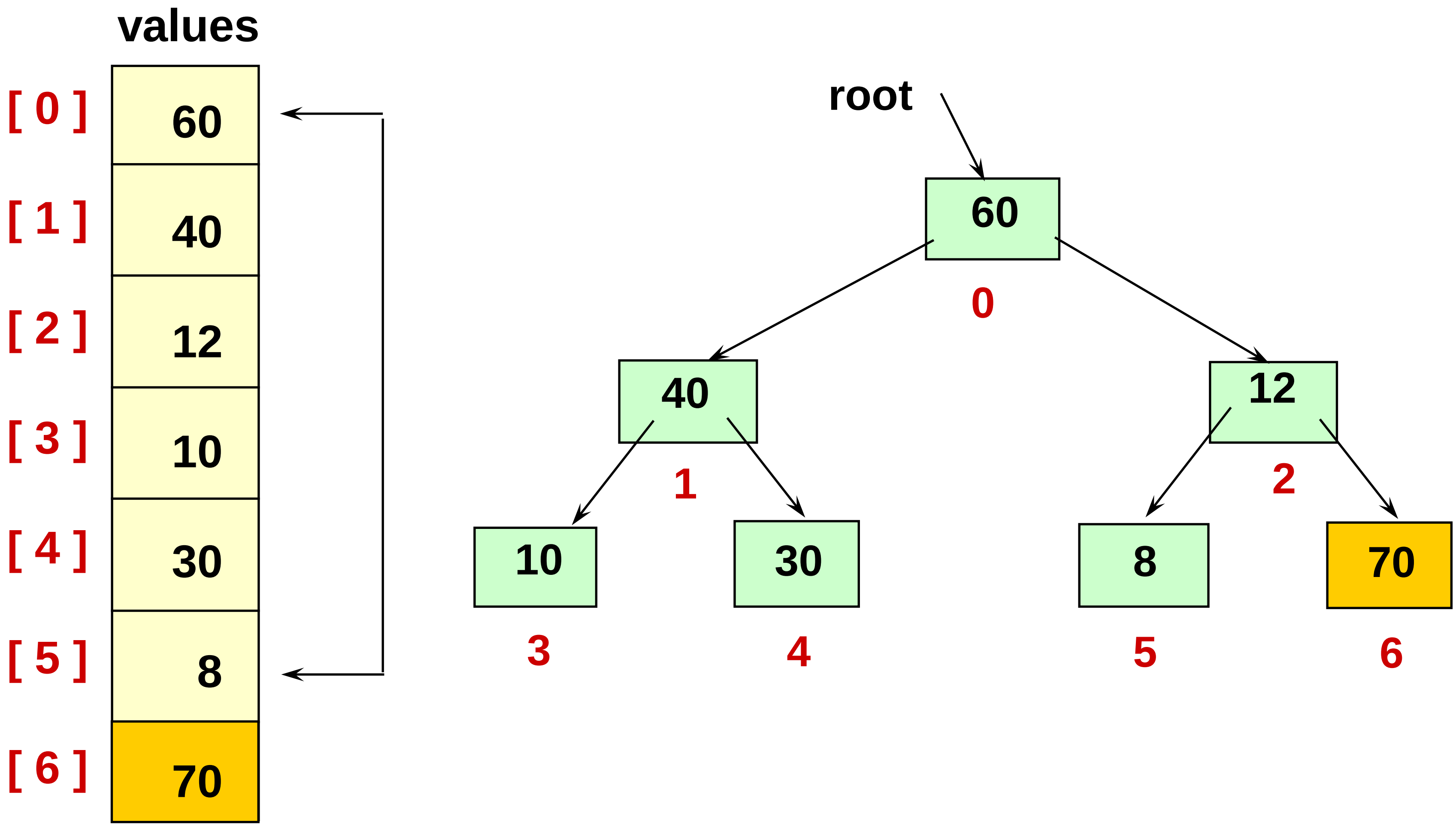


# After reheaping remaining unsorted elements

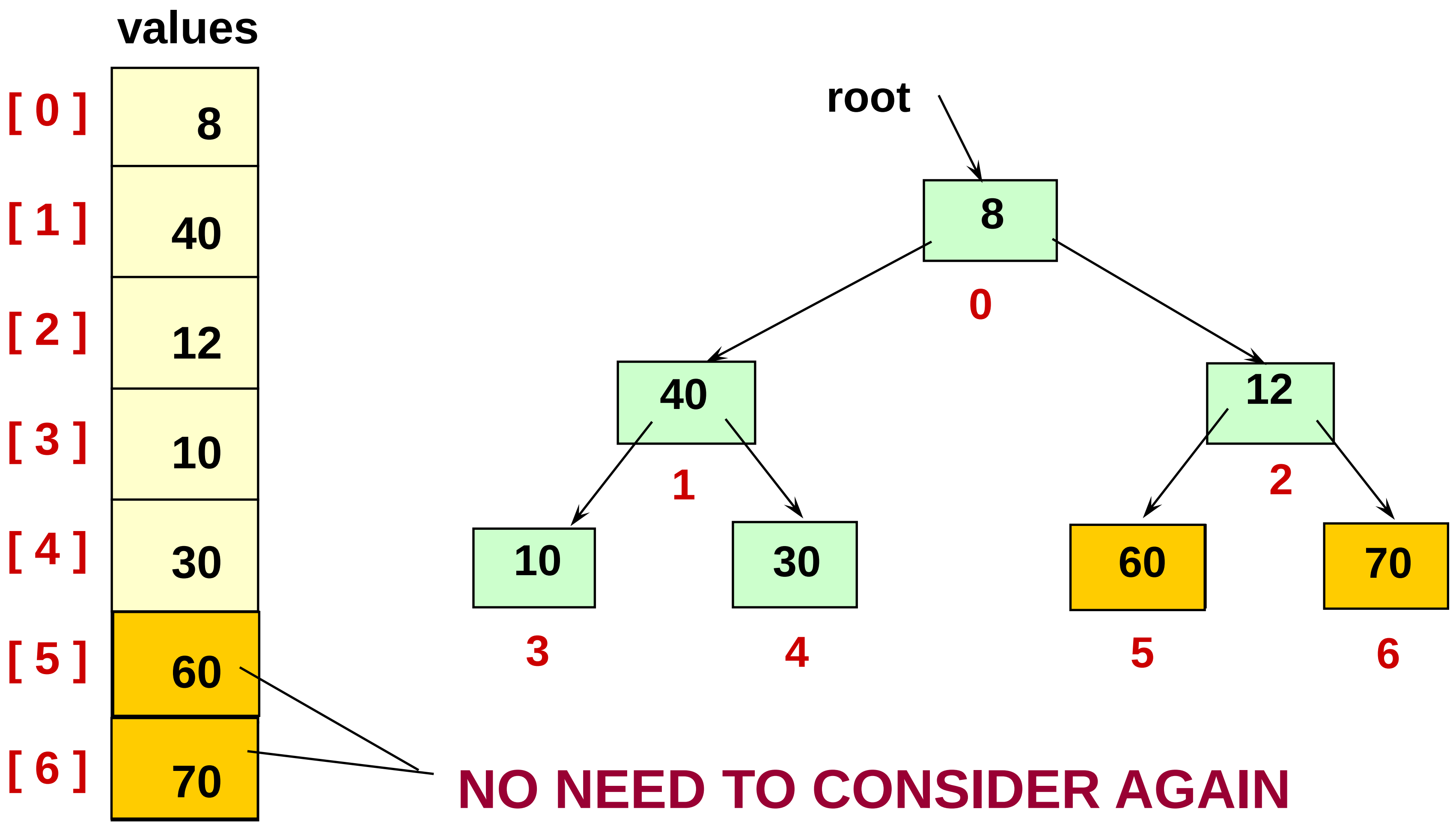
| | values |
|---|---|
| [ 0 ] | 60 |
| [ 1 ] | 40 |
| [ 2 ] | 12 |
| [ 3 ] | 10 |
| [ 4 ] | 30 |
| [ 5 ] | 8 |
| [ 6 ] | 70 |



# Swap root element into last place in unsorted array

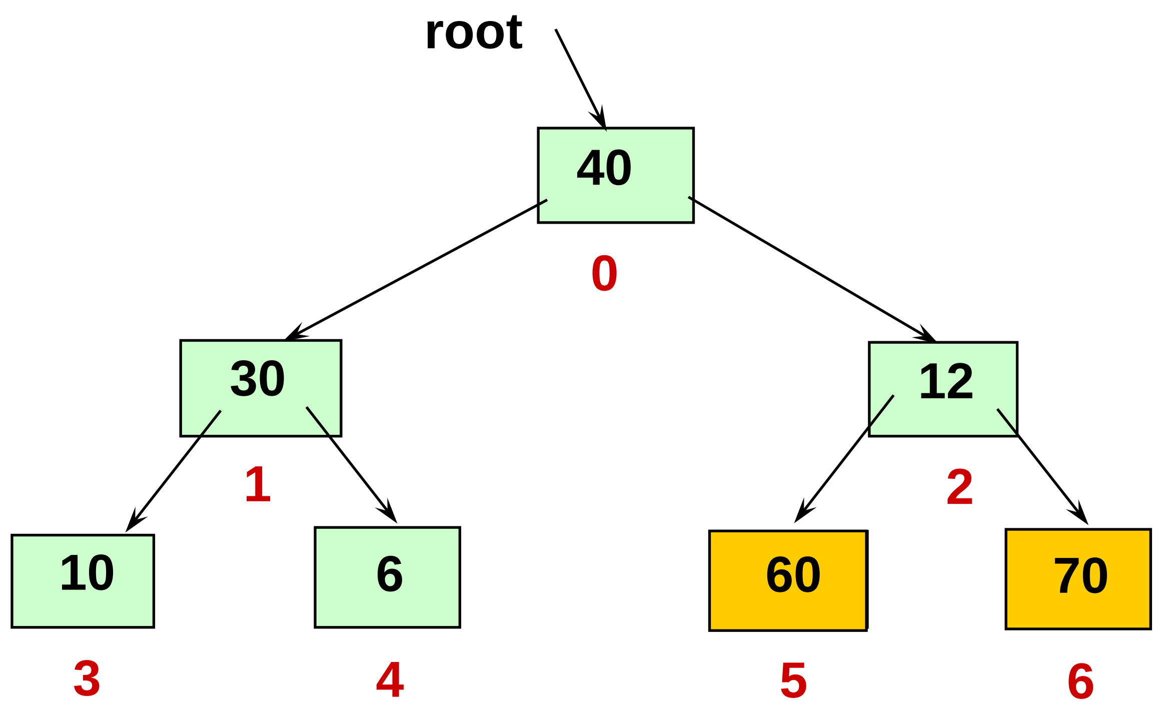


# After swapping root element into its place

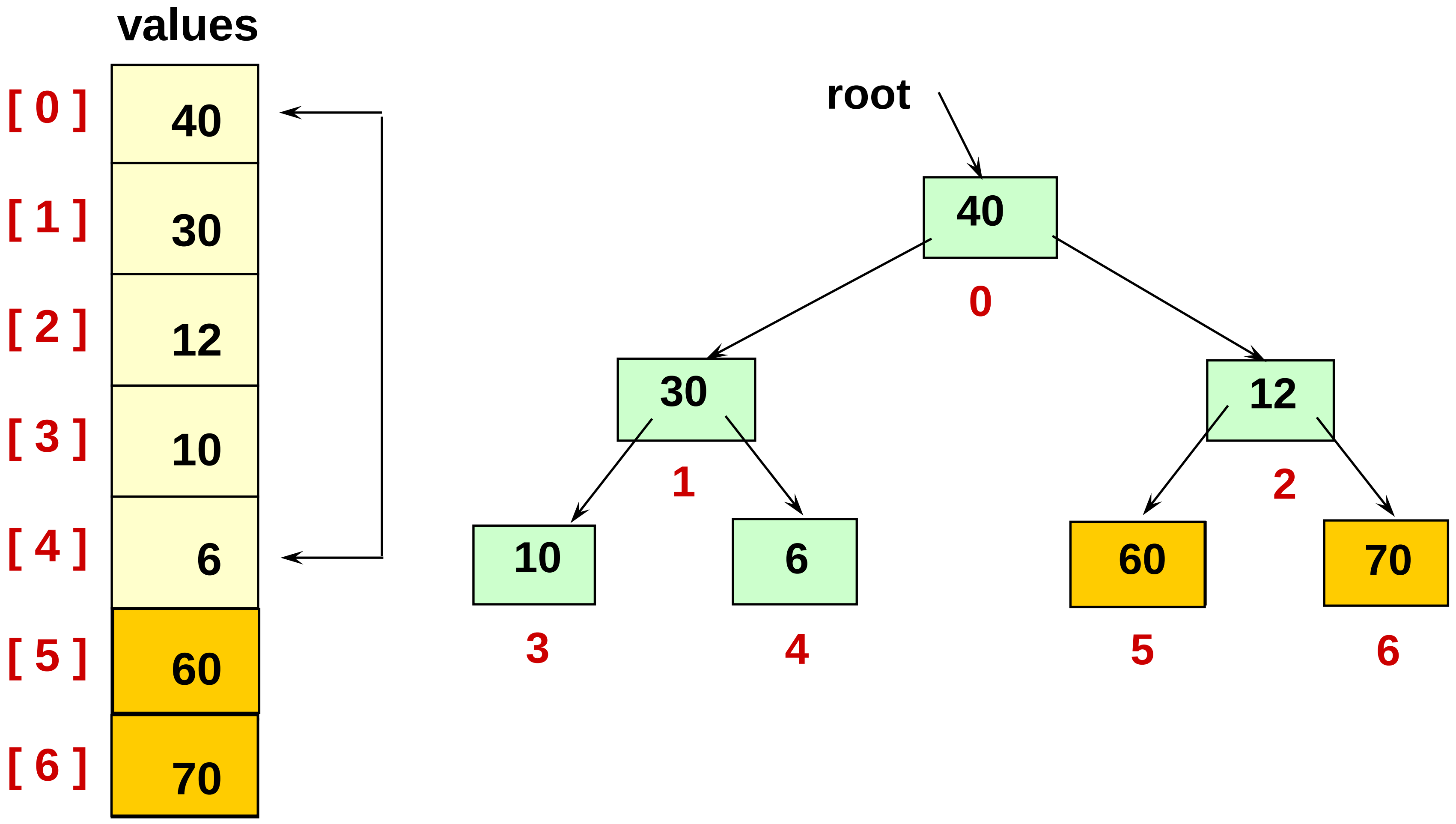


# After reheaping remaining unsorted elements

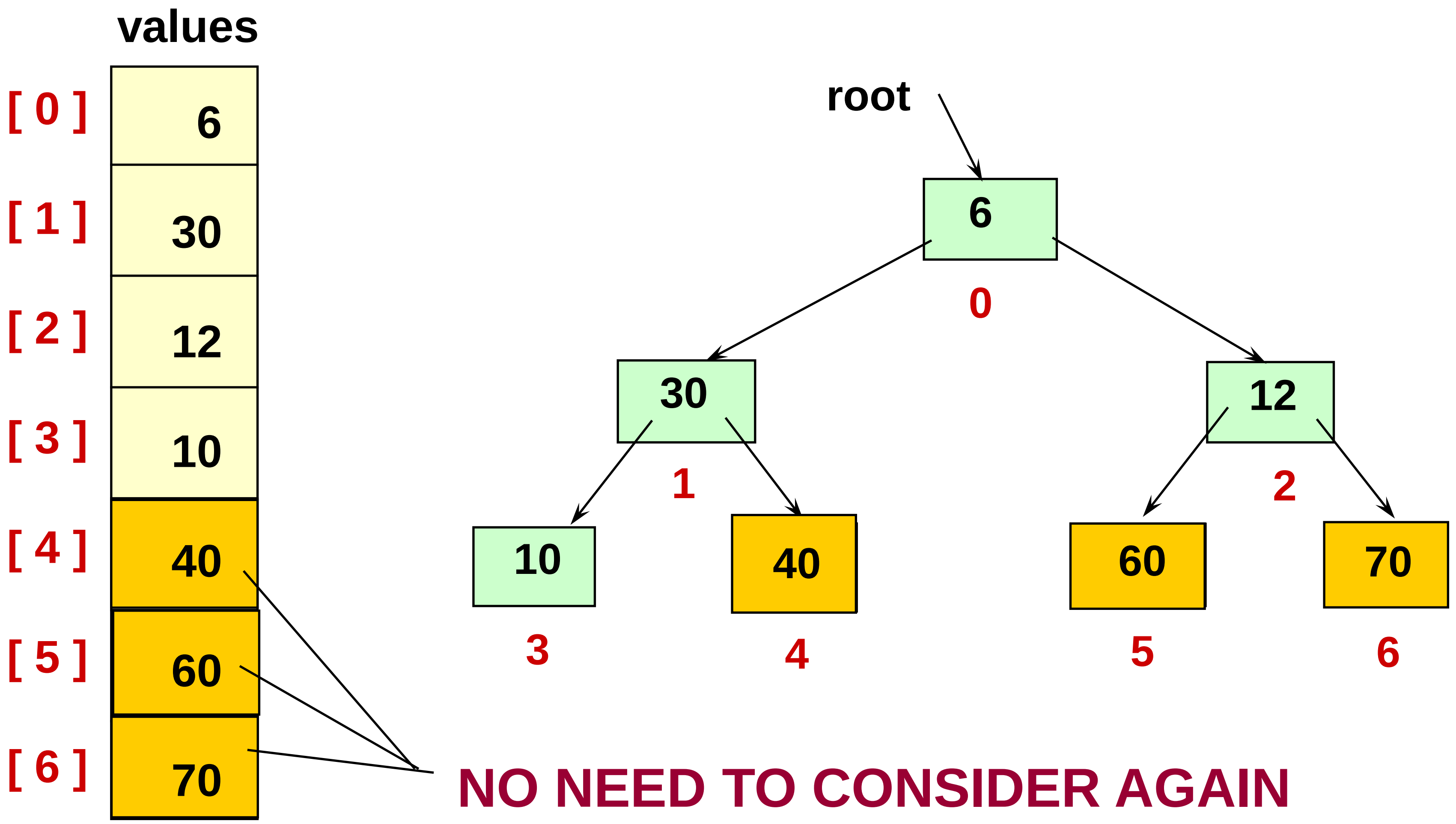
| | values |
|---|---|
| [ 0 ] | 40 |
| [ 1 ] | 30 |
| [ 2 ] | 12 |
| [ 3 ] | 10 |
| [ 4 ] | 6 |
| [ 5 ] | 60 |
| [ 6 ] | 70 |



# Swap root element into last place in unsorted array

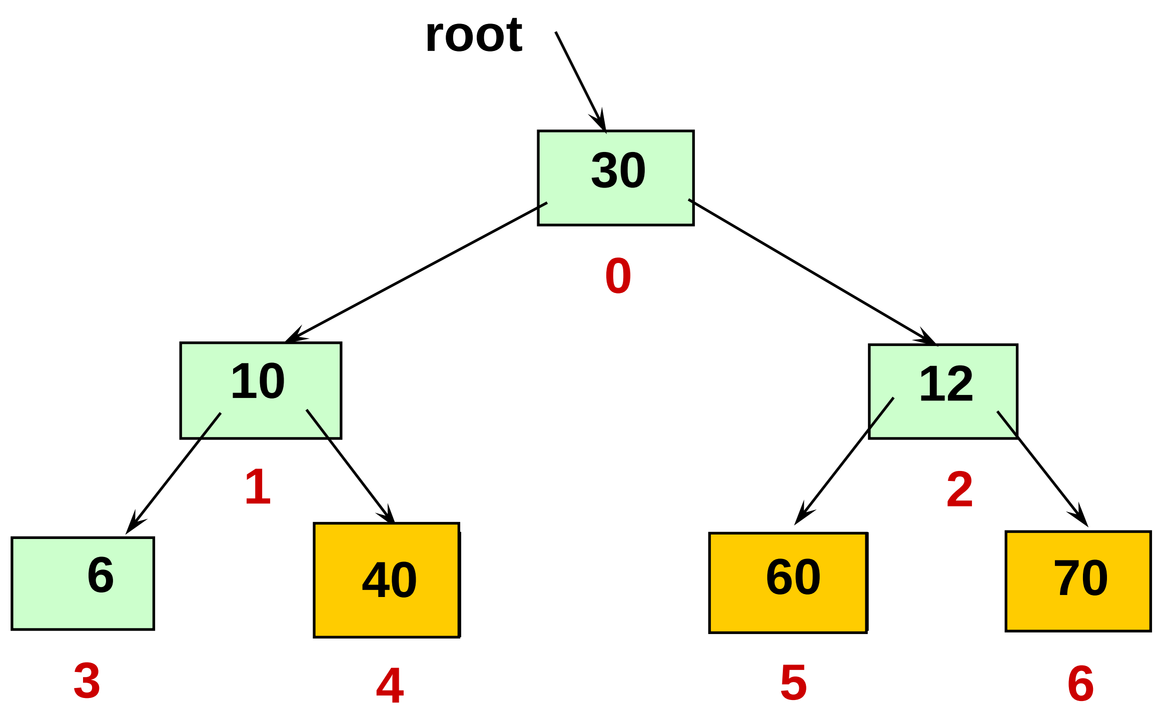


# After swapping root element into its place

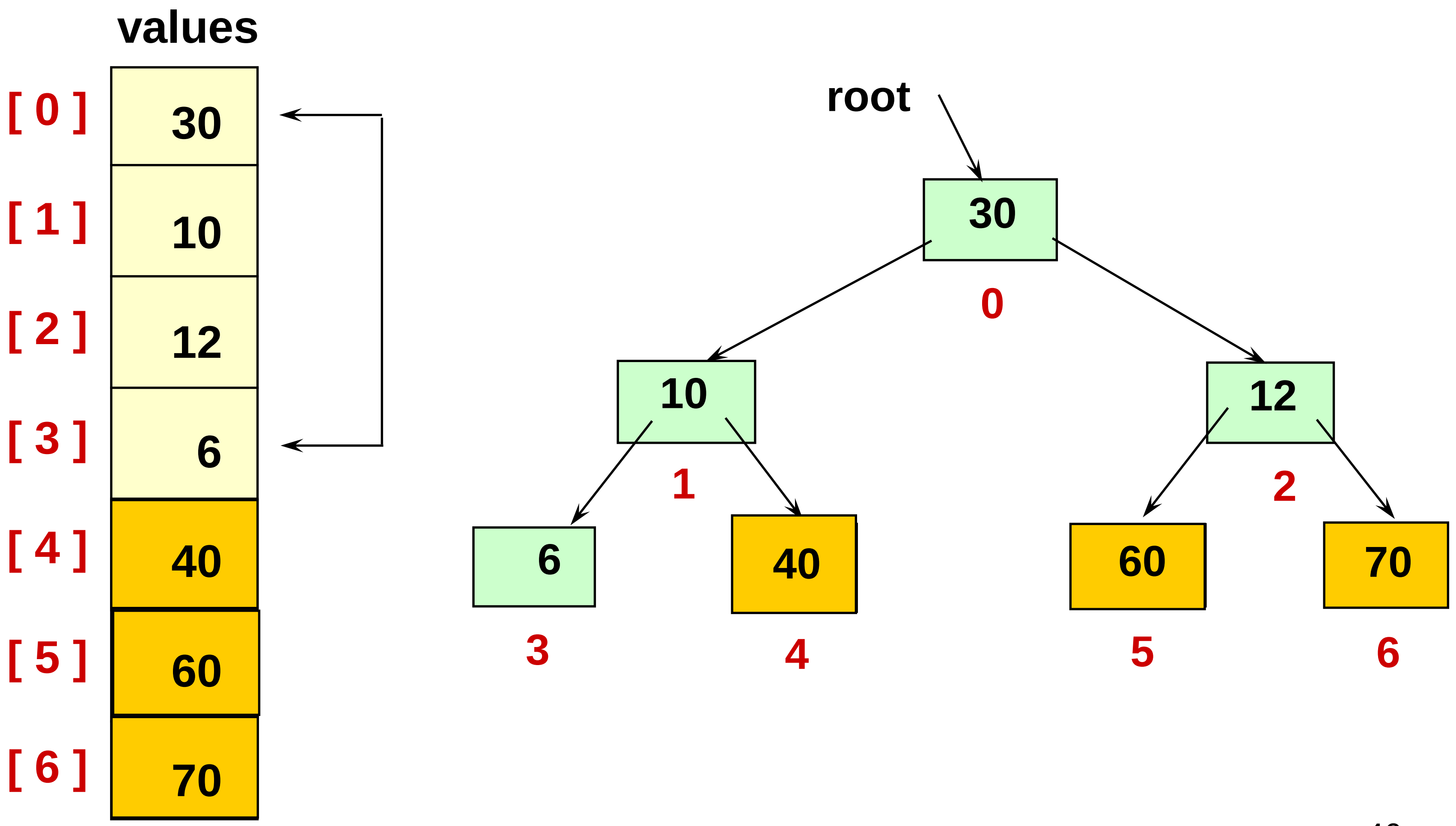


# After reheaping remaining unsorted elements

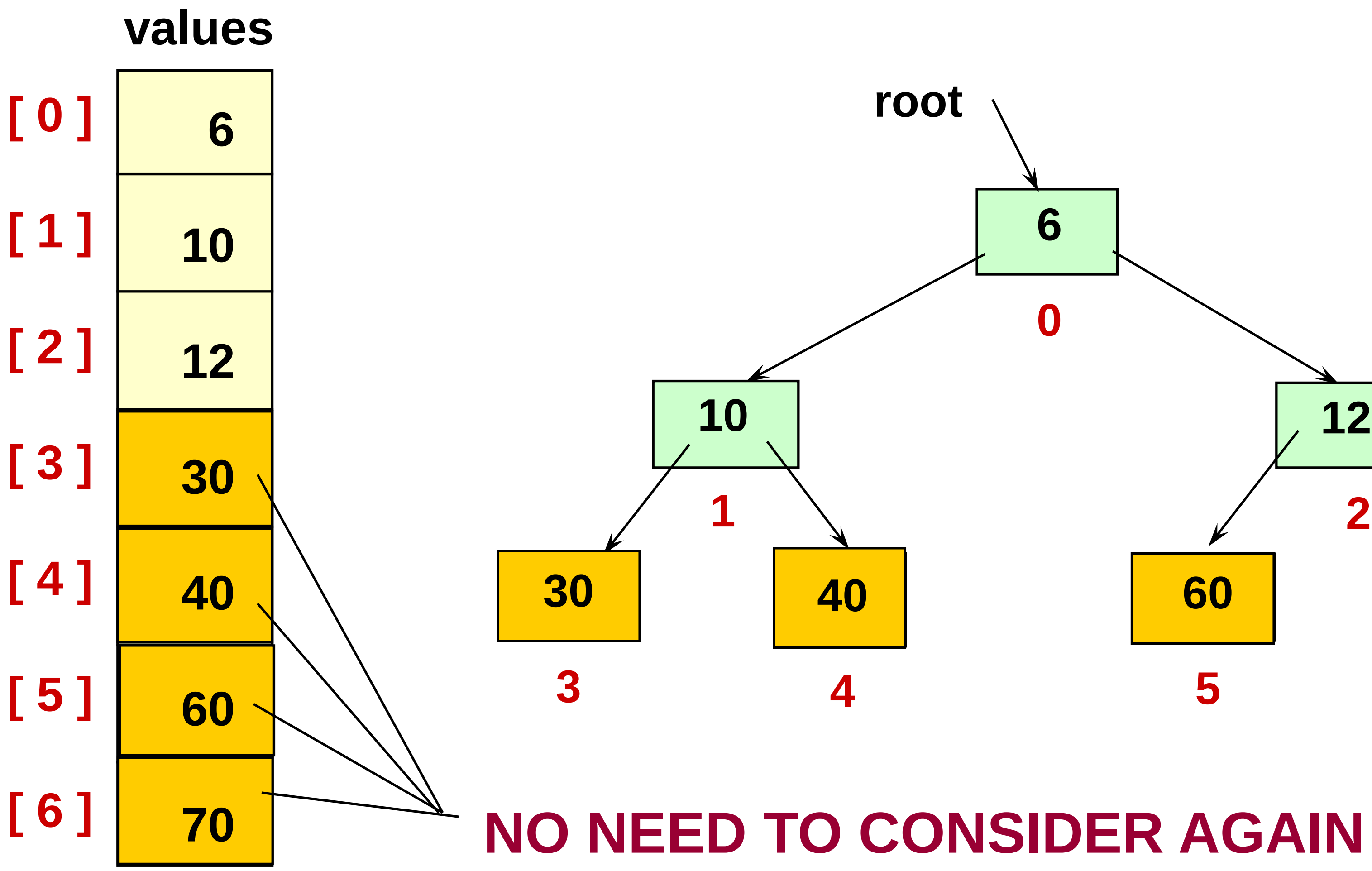
| | values |
|---|---|
| [ 0 ] | 30 |
| [ 1 ] | 10 |
| [ 2 ] | 12 |
| [ 3 ] | 6 |
| [ 4 ] | 40 |
| [ 5 ] | 60 |
| [ 6 ] | 70 |



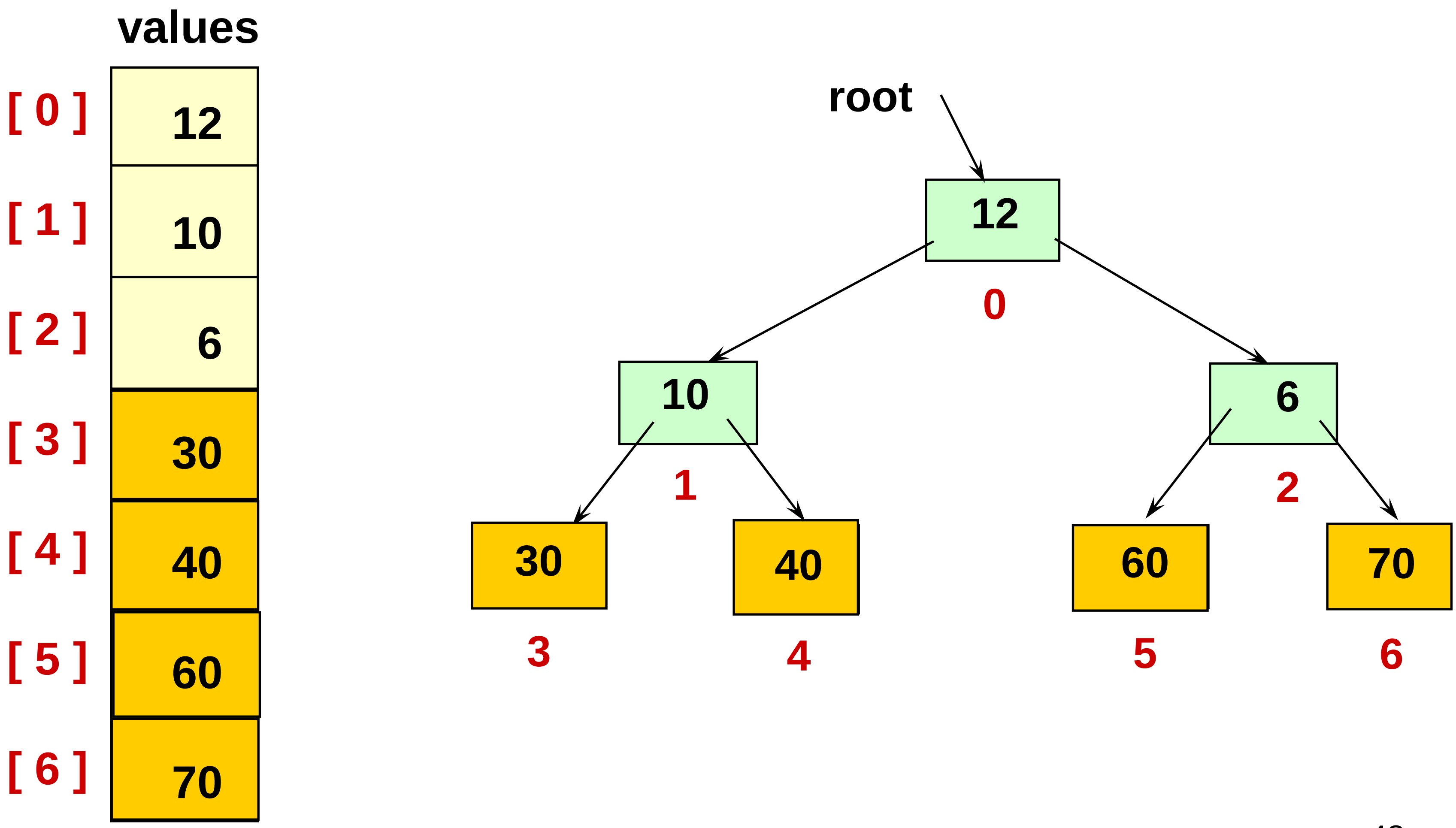
# Swap root element into last place in unsorted array



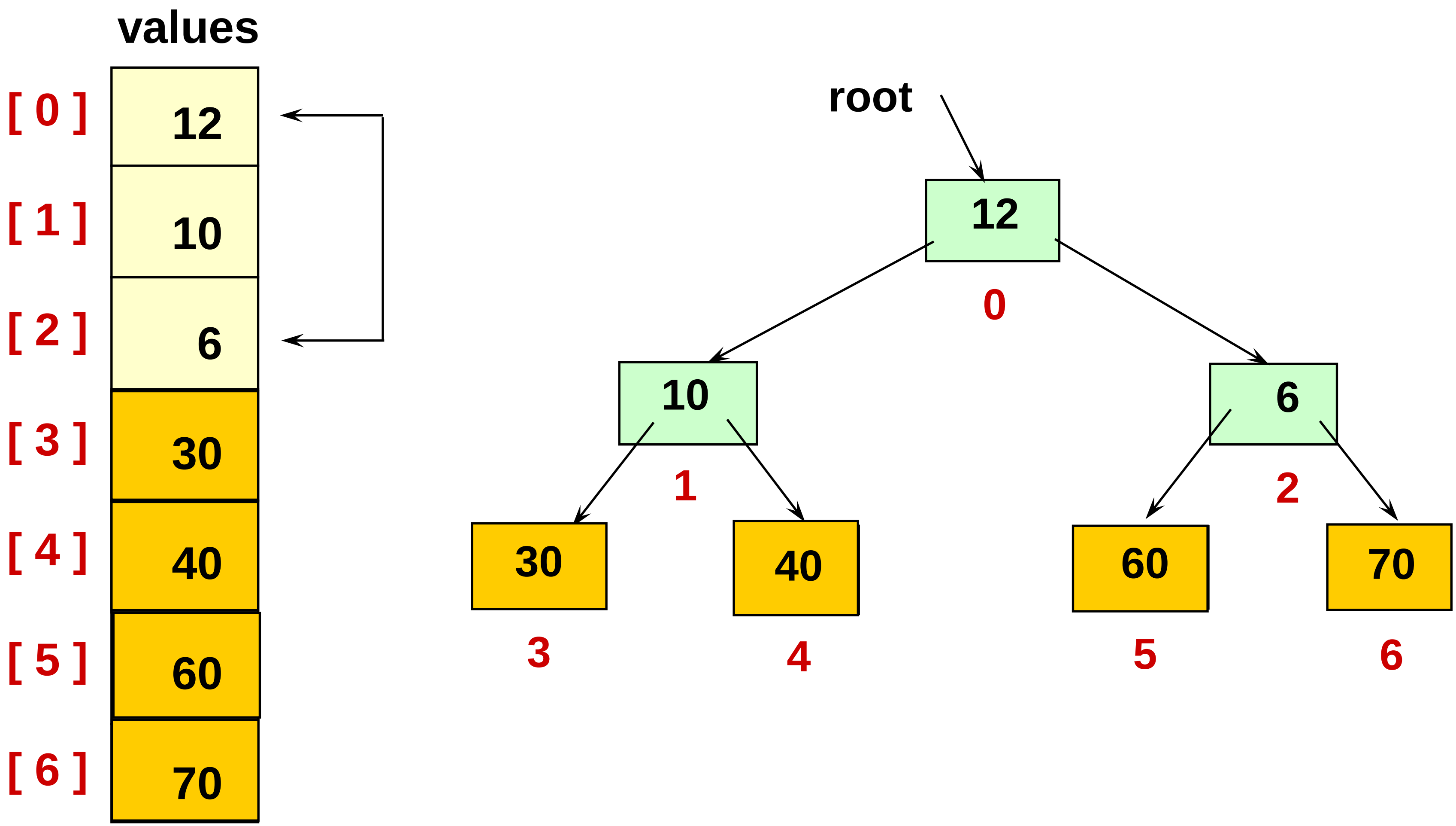
# After swapping root element into its place



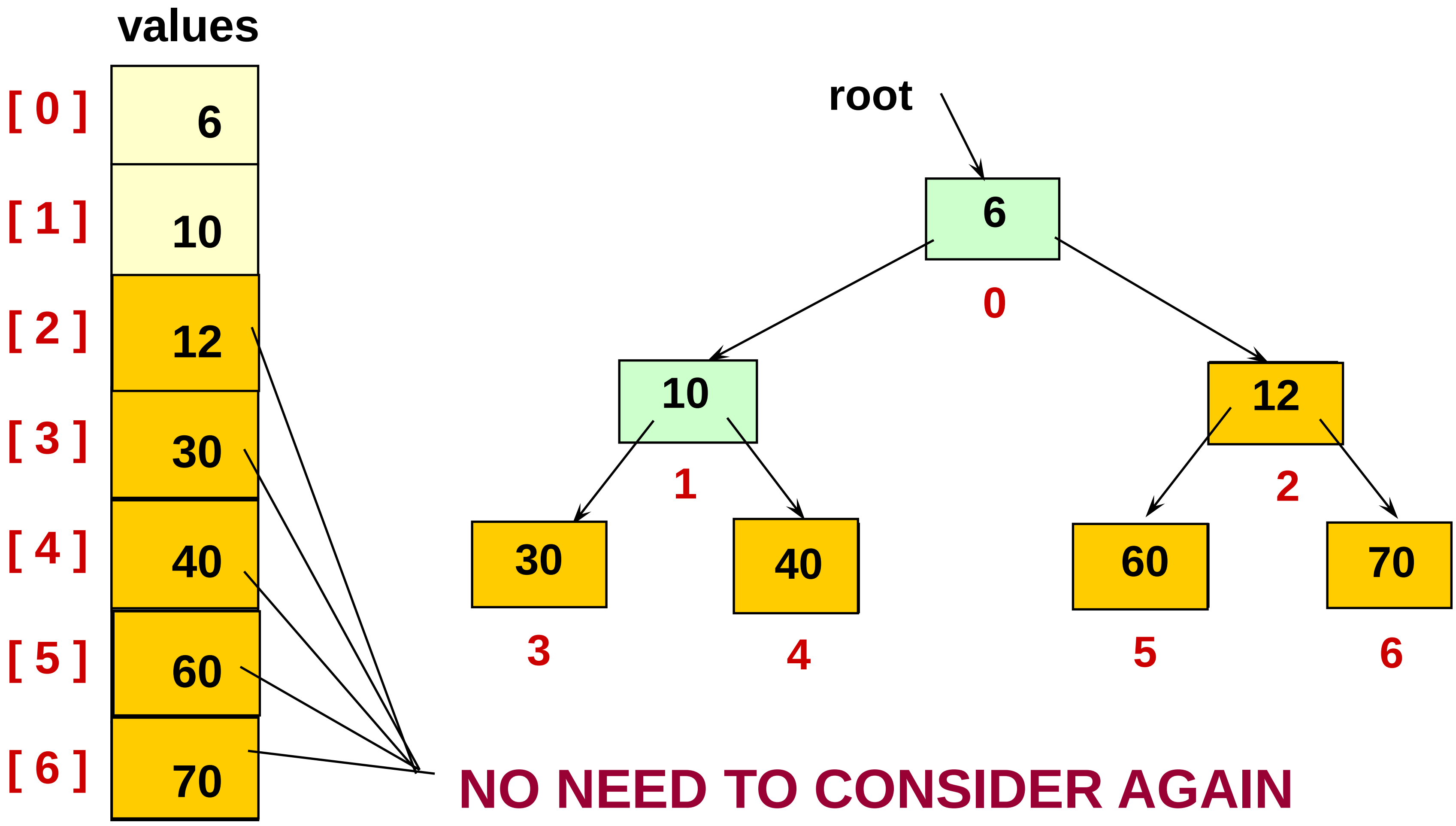
# After reheaping remaining unsorted elements



# Swap root element into last place in unsorted array



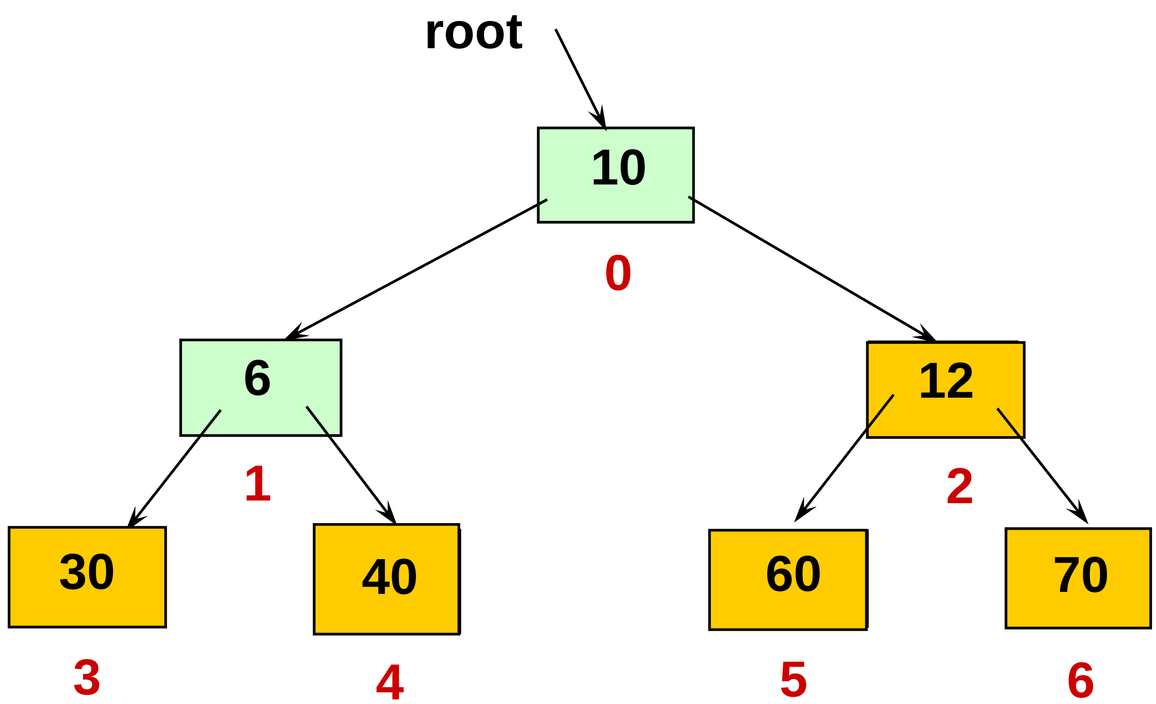
# After swapping root element into its place



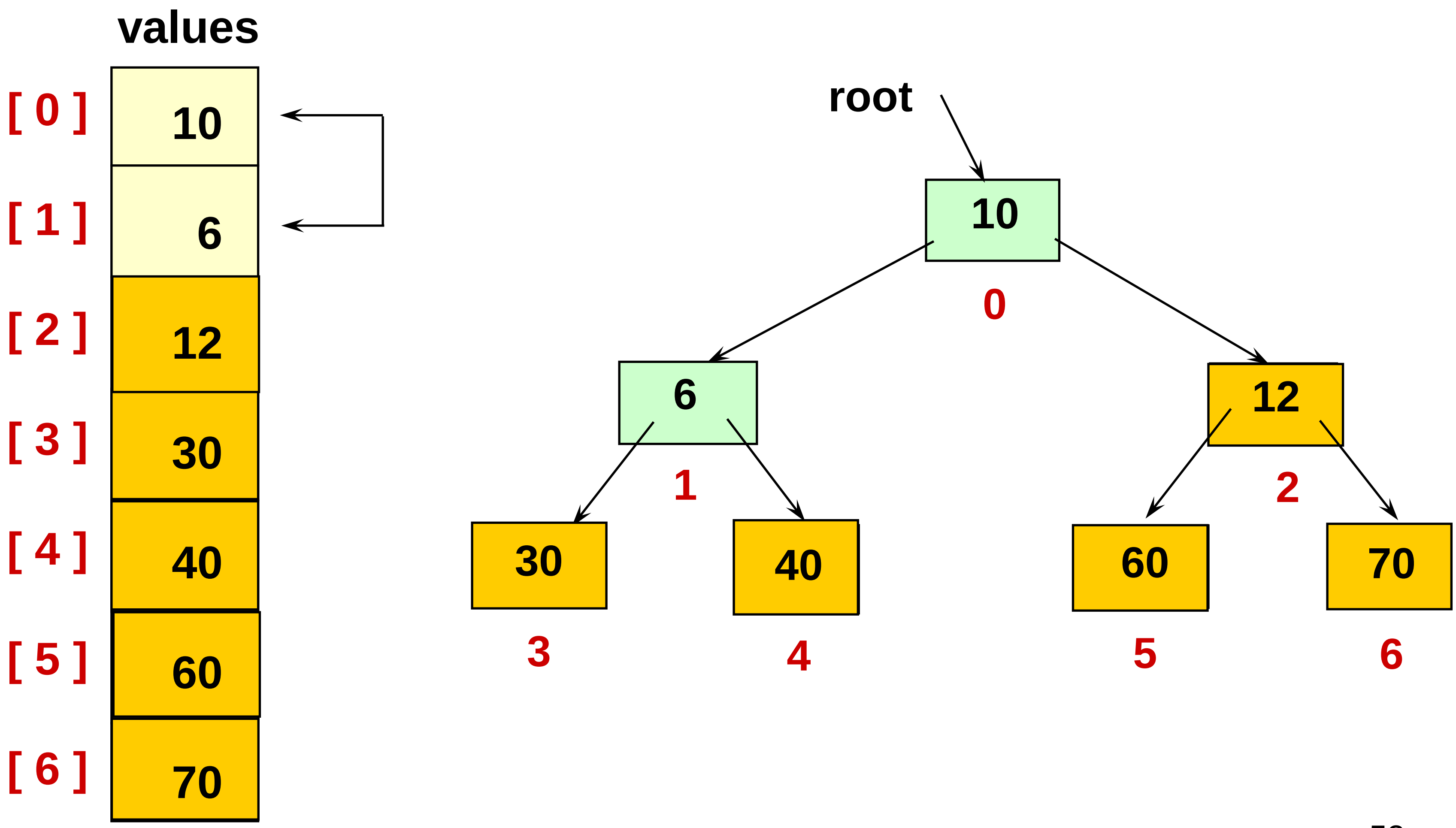
# After reheaping remaining unsorted elements

**values**

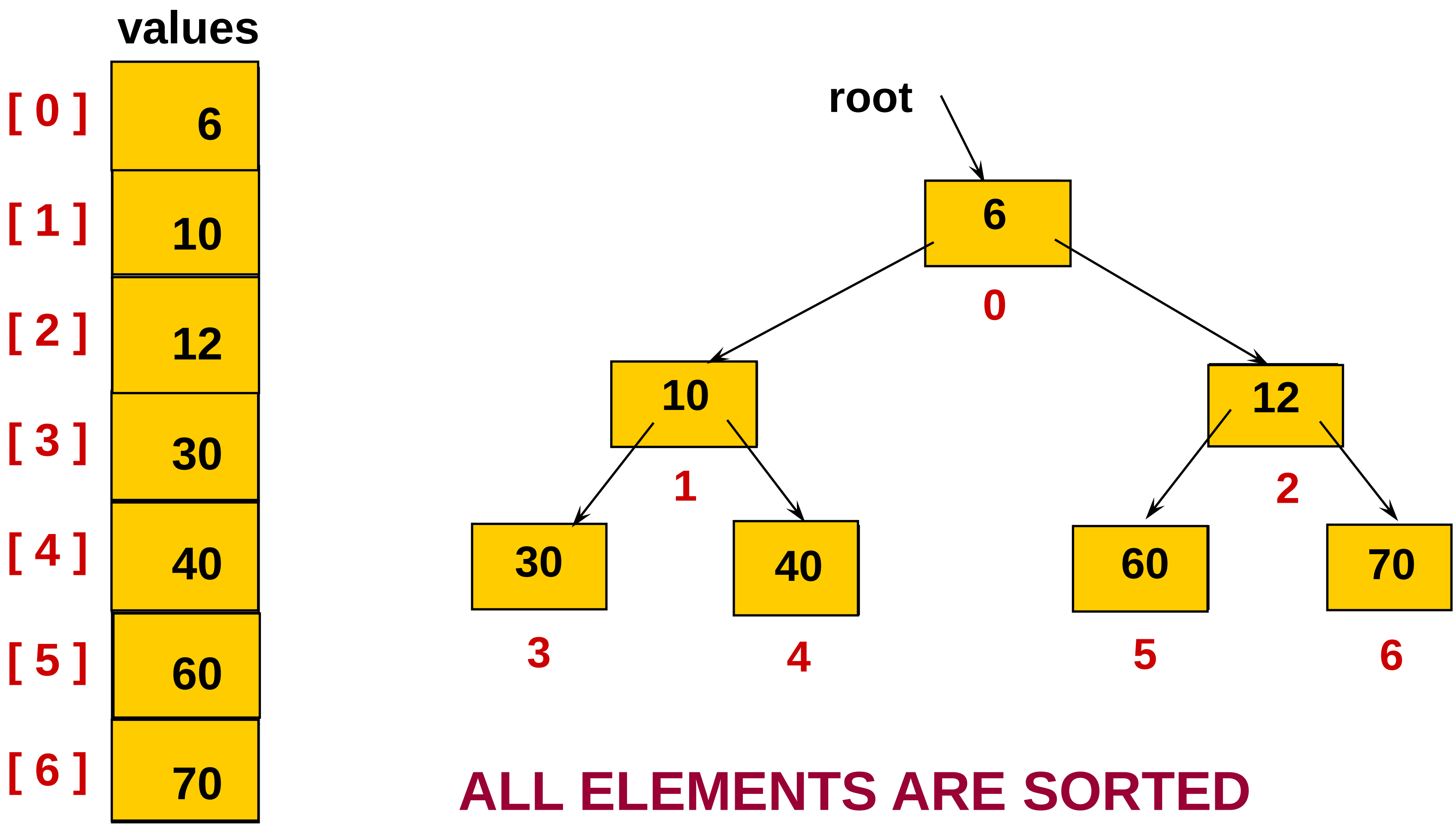
| | |
|---|---|
| [ 0 ] | 10 |
| [ 1 ] | 6 |
| [ 2 ] | 12 |
| [ 3 ] | 30 |
| [ 4 ] | 40 |
| [ 5 ] | 60 |
| [ 6 ] | 70 |



# Swap root element into last place in unsorted array



# After swapping root element into its place




**ALL ELEMENTS ARE SORTED**

```
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);
   }
}
```

# ReheapDown

```
template< class  ItemType >
void  ReheapDown ( ItemType  values [ ],  int  root,
  int  bottom )

//  Pre:  root is the index of a node that may violate the
//     heap order property
//  Post:  Heap order property is restored between root and
//     bottom

{
    int  maxChild ;
    int  rightChild ;
    int  leftChild ;

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

```
    if (leftChild  <=  bottom)     // ReheapDown continued
    {
      if  (leftChild  ==  bottom)
        maxChild  = leftChild;
      else
      {
        if (values[leftChild] <=  values [rightChild])
          maxChild  =  rightChild ;
        else
          maxChild  =  leftChild ;
      }
      if  (values[ root ] < values[maxChild])
      {
        Swap (values[root], values[maxChild]);
        ReheapDown ( maxChild, bottom  ;
      }
     }
}
```

# Building Heap From Unsorted Array

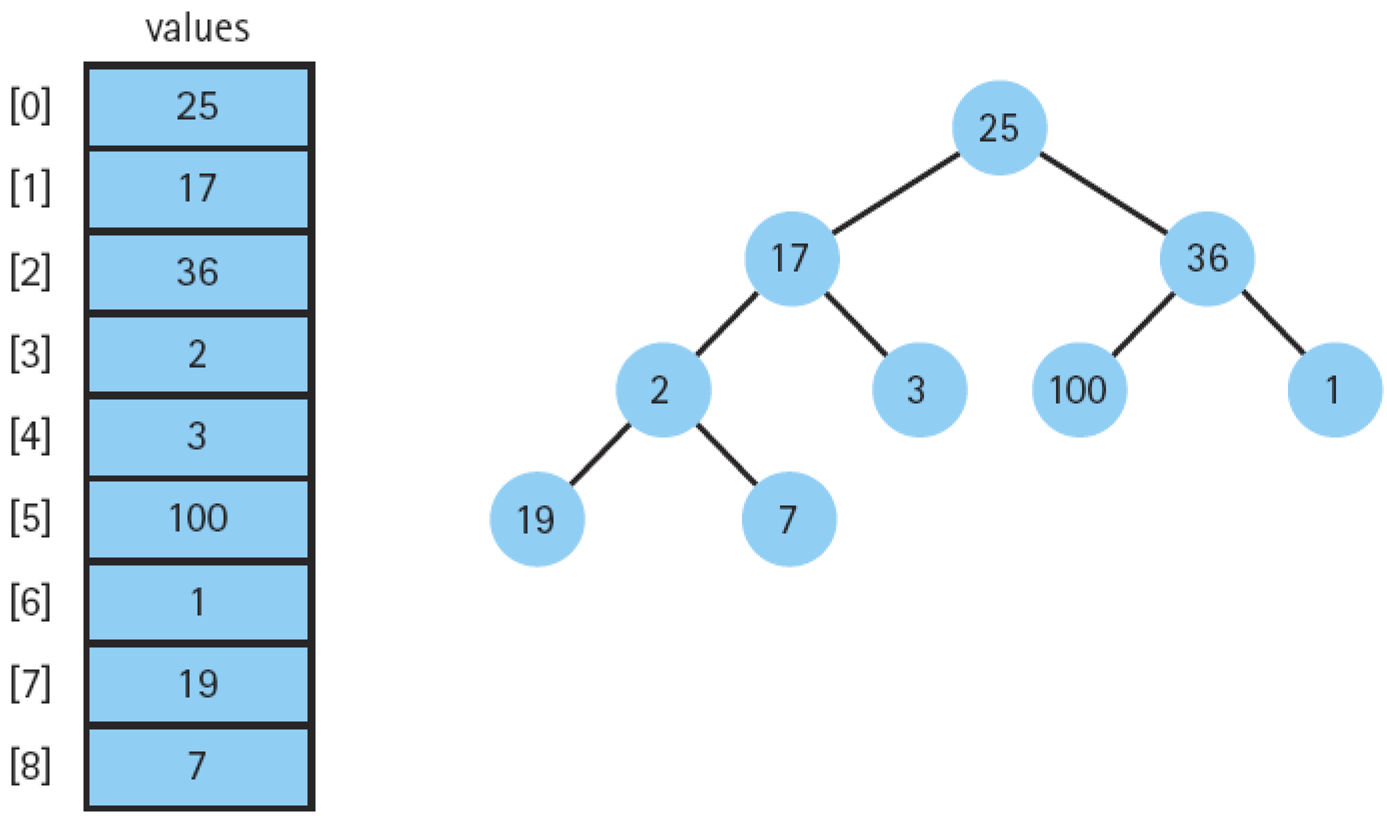
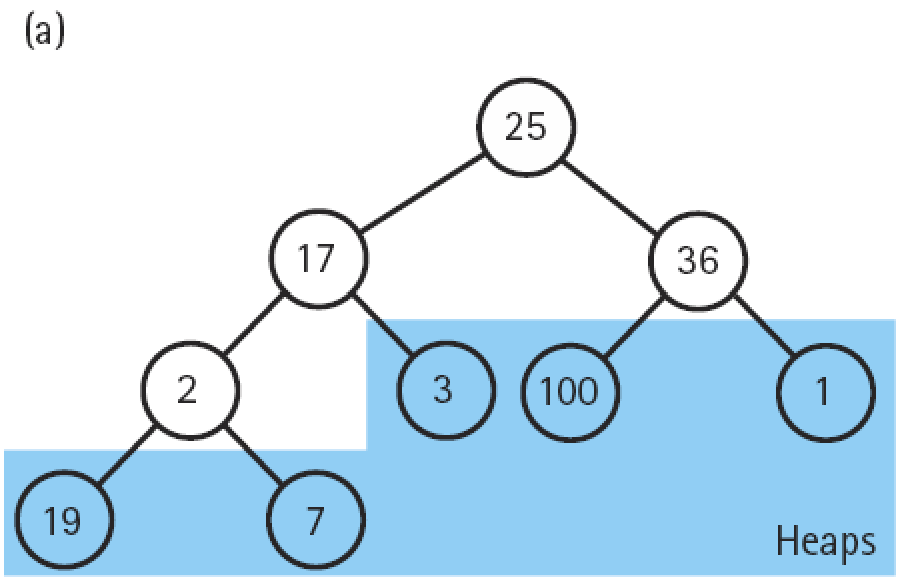



Figure 10.12 *An unsorted array and its tree*

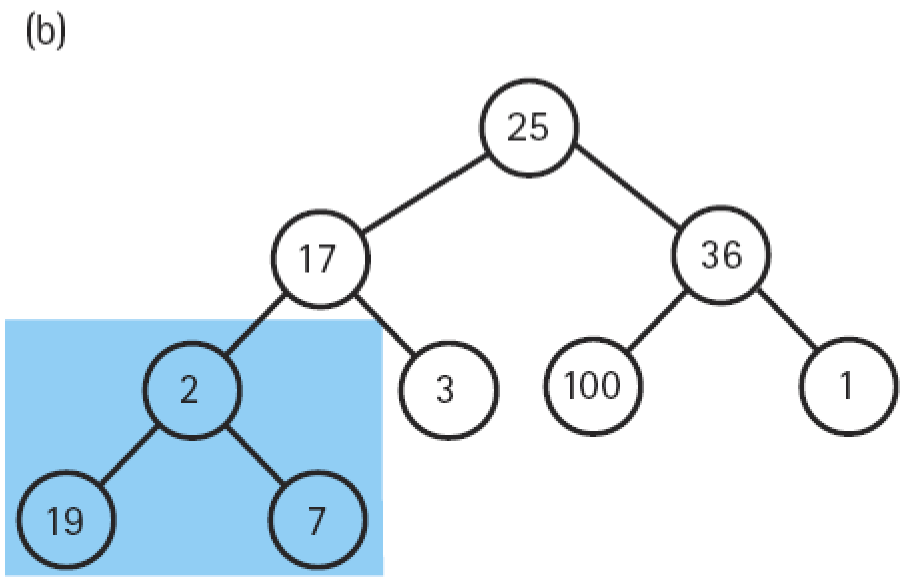
# Building Heap From Unsorted Array (cont'd)

Leaf nodes are already heaps



# Building Heap From Unsorted Array (cont'd)

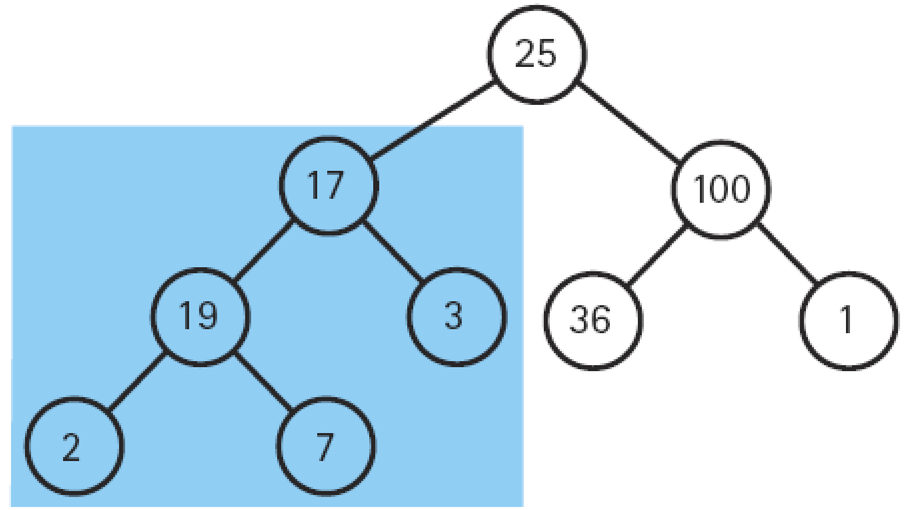
The subtrees rooted at first nonleaf nodes are almost heaps



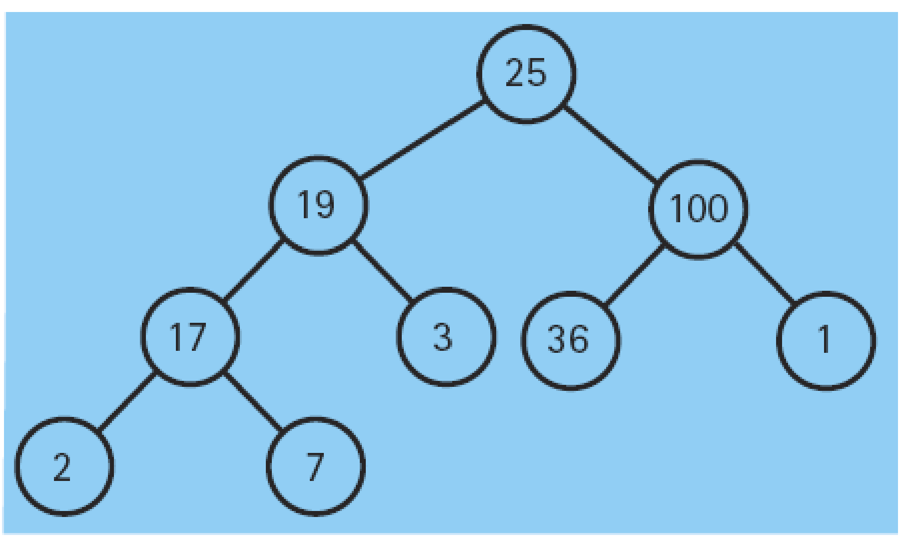
# Building Heap From Unsorted Array (cont'd)

Move up a level in the tree and continue reheaping until we reach the root node

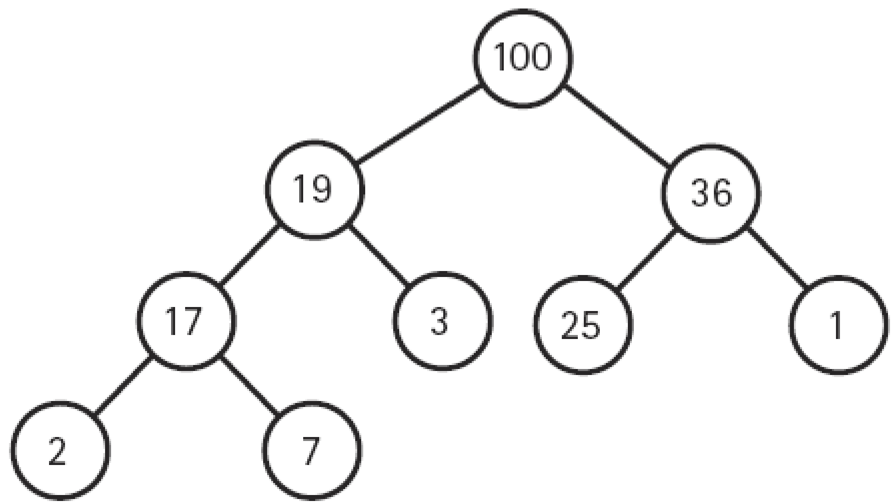
(d)




(e)

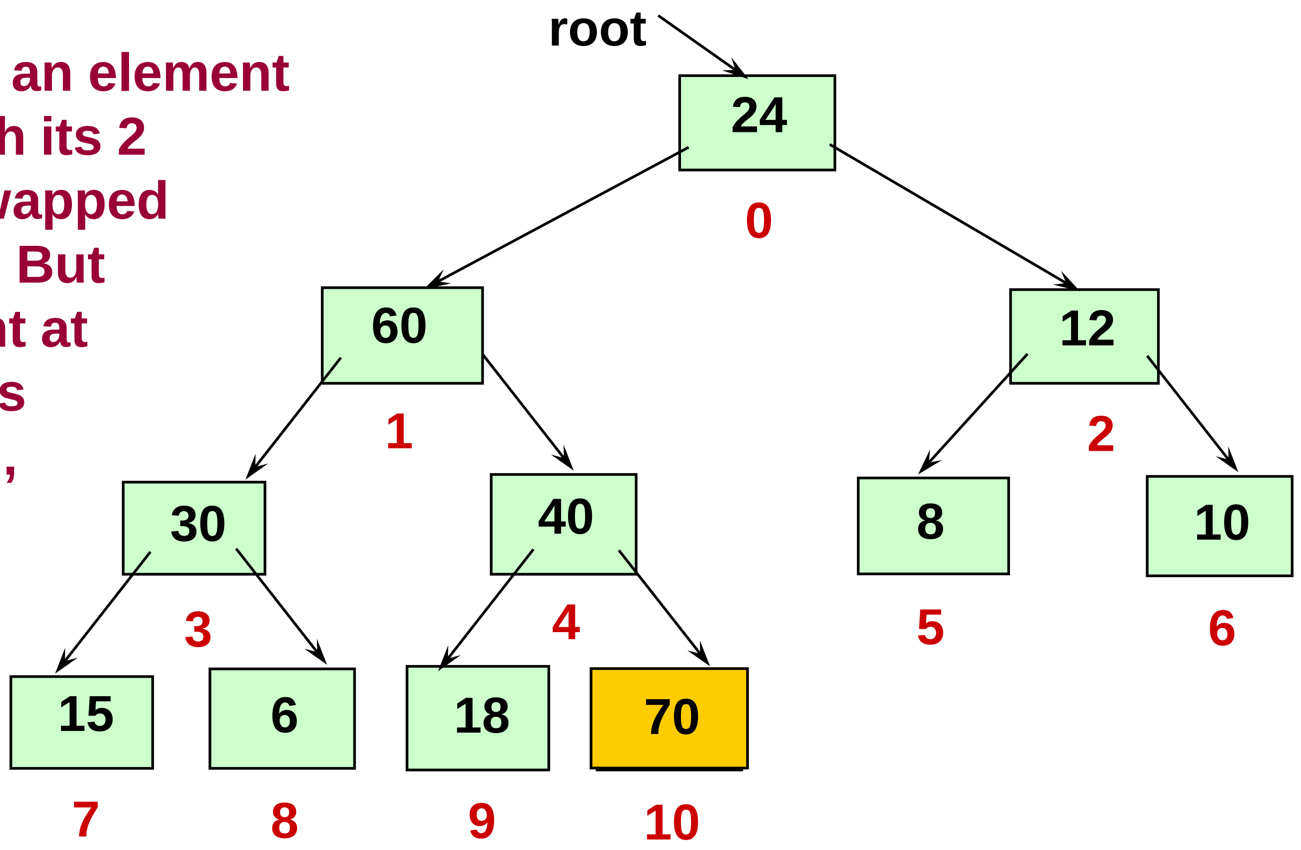



(f) Tree now represents a heap



# 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**