ALGORITHMS

Definitions

- Informally, an ***algorithm is any well-defined computational procedure that takes*** some value, or set of values, as ***input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the*** input into the output.
- We can also view an algorithm as a tool for solving a well-specified ***computational problem. The statement of the problem specifies in general terms the desired*** input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Characteristics of an Algorithm

- Input: Requires some input which it will process
- Output: Result of processing the input
- Definiteness: Consistent. Always acts in the same way for the same input
- Effectiveness: Gives the expected output
- Termination: Ends after a finite period of time

ALGORITHM ANALYSIS

- In computer science, the **analysis of algorithms** is the process of finding the computational complexity of **algorithms** – the amount of time, storage, or other resources needed to execute them.

- The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application.
- The analysis of an algorithm can help us understand it better, and can suggest informed improvements.
- *Algorithms tend to become shorter, simpler, and more elegant during the analysis process.*

Importance of Performance Analysis

1) To estimate how long a program will run.
2) To estimate the largest input that can reasonably be given to the program.
3) To compare the efficiency of different algorithms.
4) To choose an algorithm for an application.
5) To help focus on the parts of code that are executed the largest number of times.

Analysis of Algorithms

We can say that we are looking for the most suitable algorithm for a specific purpose. For this, we need to *analysis* the algorithm under specific constraints. *An* algorithm can be analysed under three specific case

- **Best case analysis**.
- ***Average* case analysis**.
- **Worst case analysis**.

Best case analysis.

- We analyse the performance of the algorithm under the circumstances on which it works best.
- In that way, we can determine the upper-bound of its performance. However, you should note that we may obtain these results under very unusual or special circumstances and it may be difficult to find the optimum input data for such an analysis.

Average case analysis.

- This gives an indication on how the algorithm performs with an average data set.
- It is possible that this analysis is made by taking all possible combinations of data, experimenting with them, and finally averaging them.
- However, such an analysis may not reflect the exact behaviour of the algorithm you expect from a real-life data set.
- Nevertheless, this analysis gives you a better idea how this algorithm work for your problem.

Worst case analysis.

- In contrast to the best-case analysis, this gives you an indication on how bad this algorithm can go, or in other words, gives a lower-bound for its performance.
- Sometimes, this could be useful in determining the applicability of an algorithm on a mission-critical application.
- However, this analysis may be too pessimistic for a general application, and even it may be difficult to find a test data set that produces the worst case

Algorithms Performance

- Two important ways to characterize the effectiveness of an algorithm are its ***space complexity*** and ***time complexity***
- **Time complexity** of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. (Problem size= # of inputs)
- Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count, Instead attempts only to get asymptotic bounds on the step count.
- Asymptotic analysis makes use of the O (Big Oh) notation.
- Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation.

Time Complexity

- It is a measure of how long a computation takes to execute.
- The measurement of complexity is usually described in terms of how the time (or possibly space) requirements increase as the size of the input to the program increases.
- The most common measure of complexity is the amount of time the program takes to run.
- “The difference between time and space is that you can re-use space”.

Space Complexity

- Its the amount of memory space required by the algorithm, during the course of its execution.
- Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.
- An algorithm generally requires space for following components :
  - **Instruction Space :** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
  - **Data Space :** Its the space required to store all the constants and variables value.
  - **Environment Space :** Its the space required to store the environment information needed to resume the suspended function.

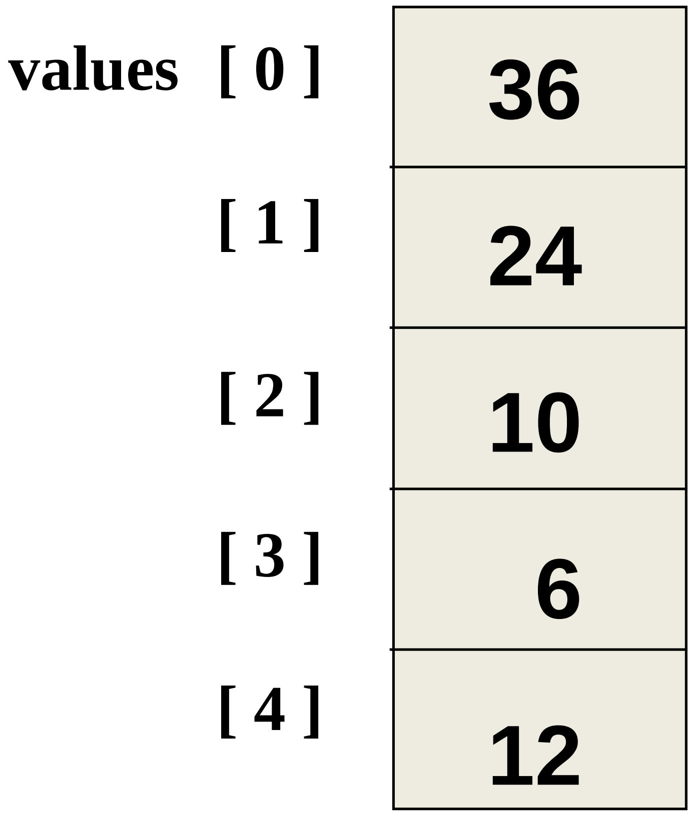
Sorting: *Agenda*

- Define sorting
- Learn the basics and how to implement some of the following sorting algorithms:
    - selection sort,
    - bubble sort,
    - heapsort,
    - quicksort

What is sorting

- Sorting rearranges the elements into either ascending or descending order within the data structure.

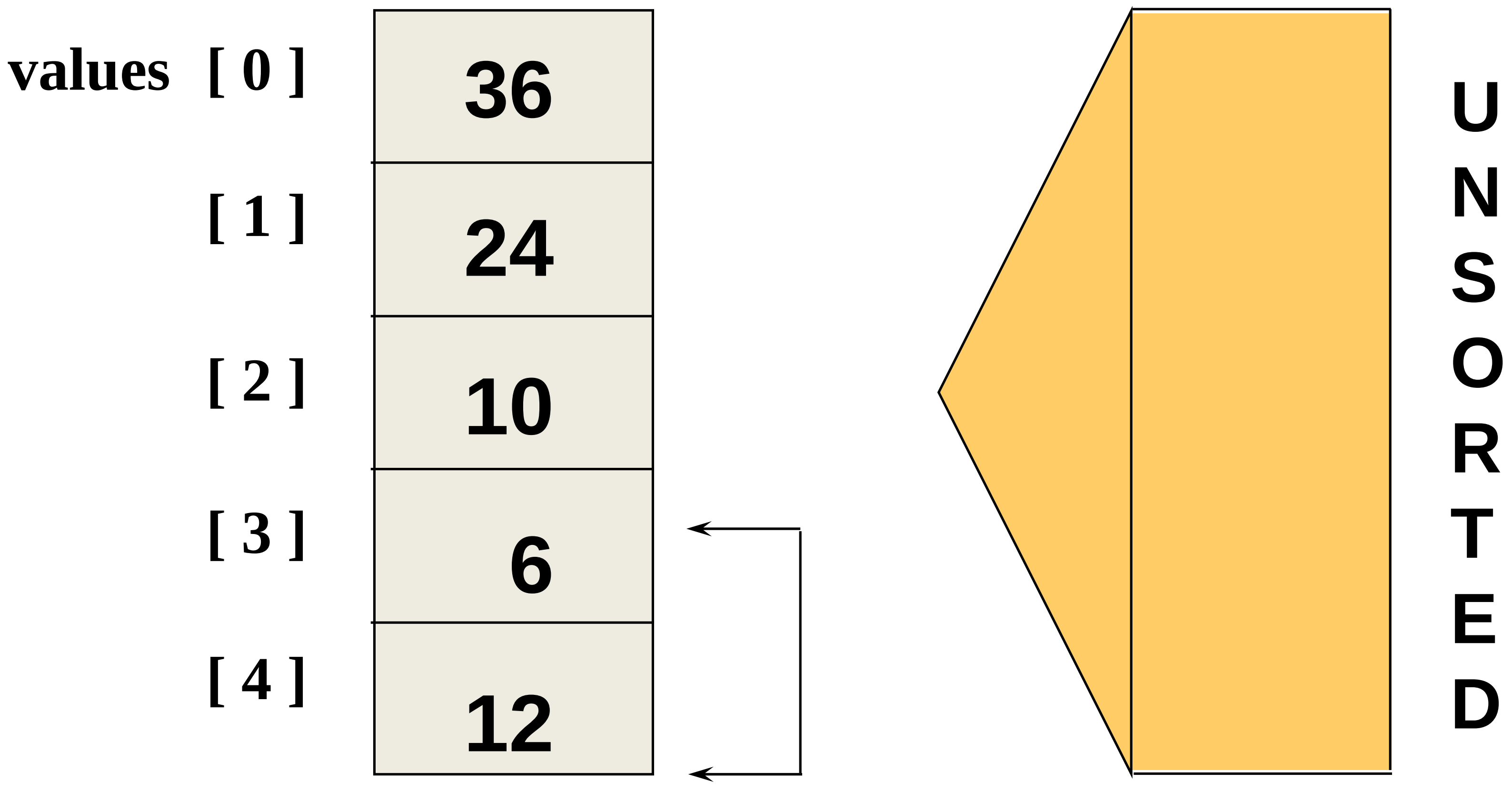
Bubble Sort



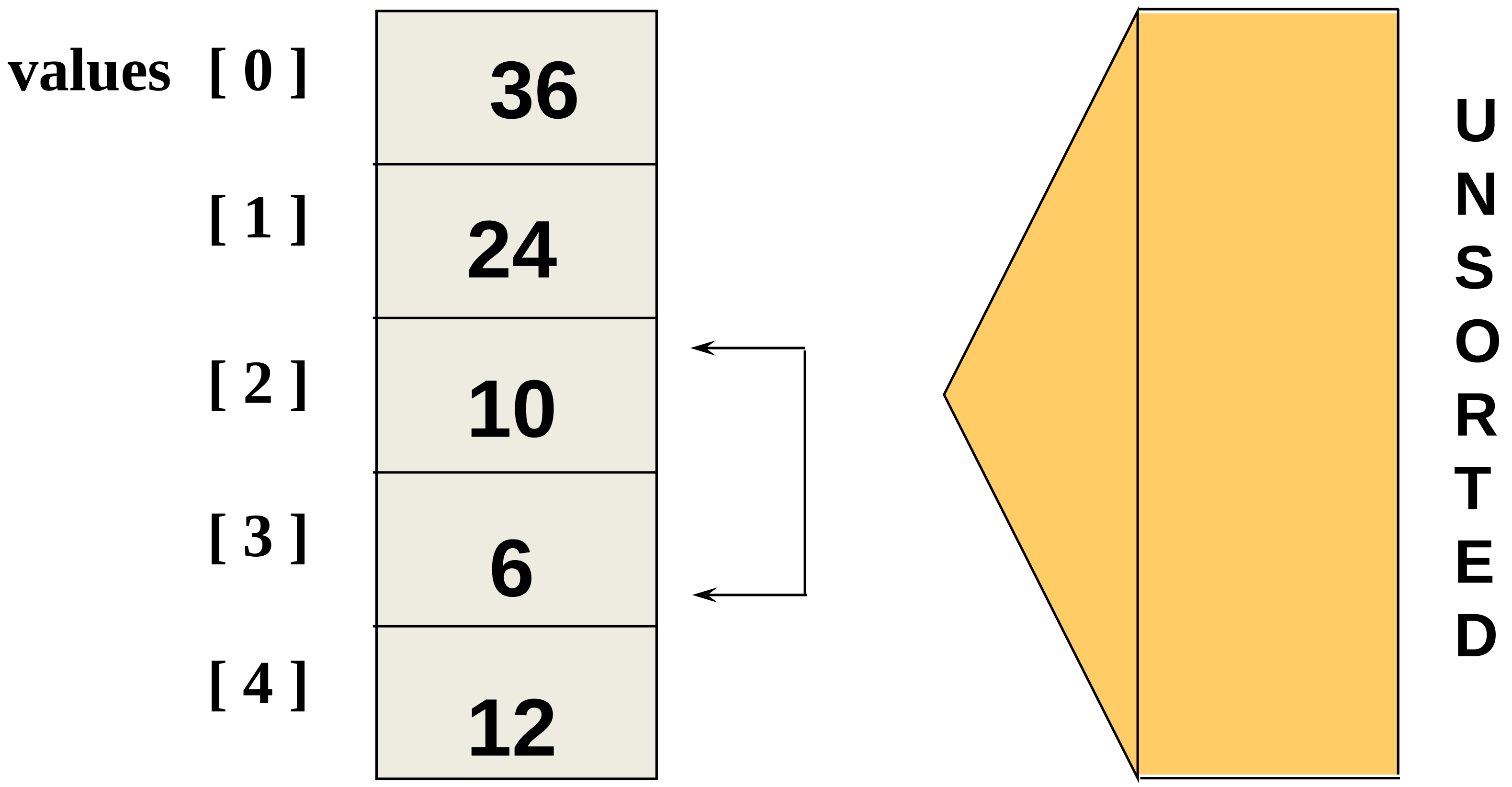
Compares neighboring pairs of array elements, and swaps neighbors whenever they are not in correct order.

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

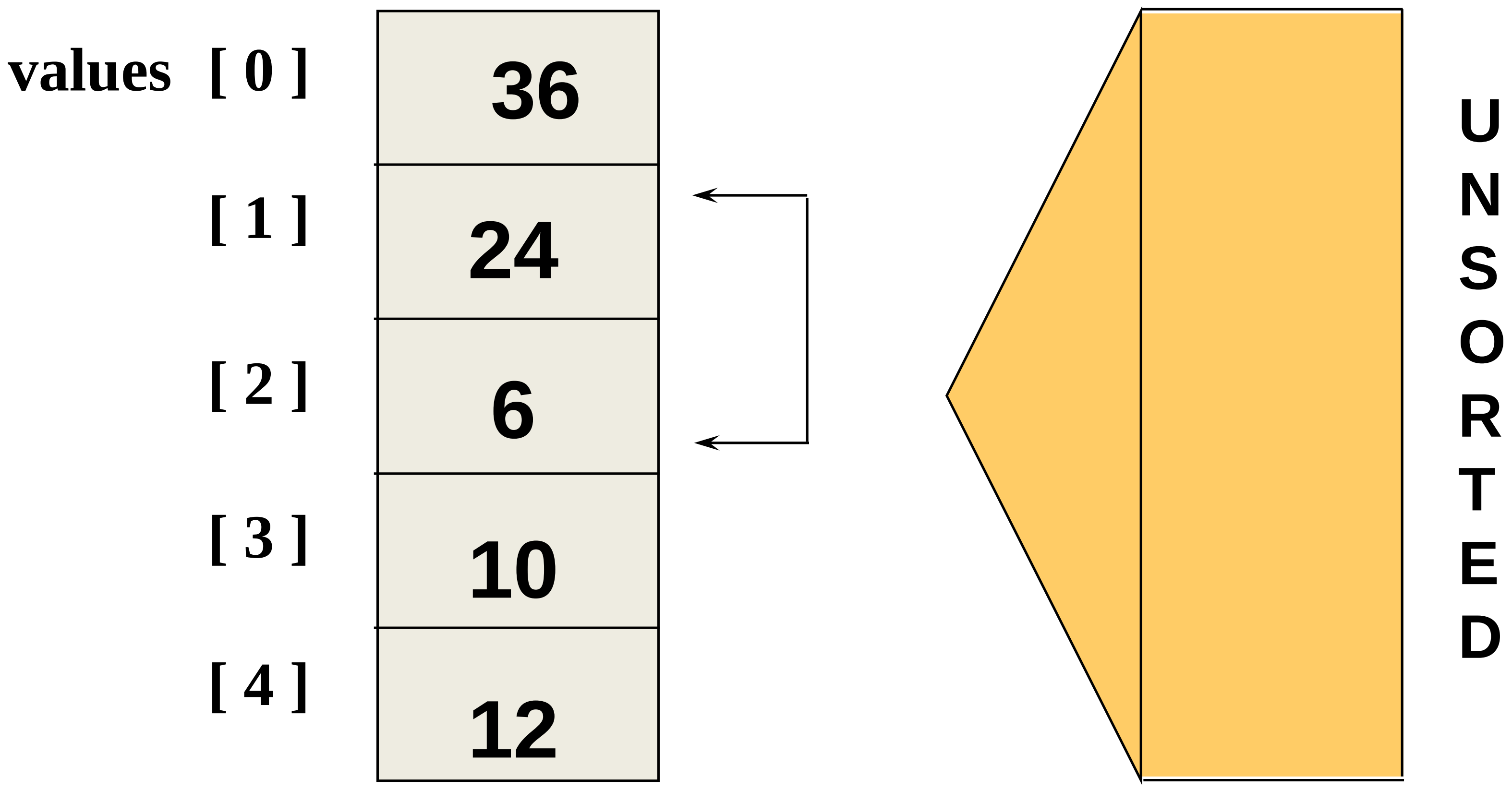
Bubble Sort: Pass One



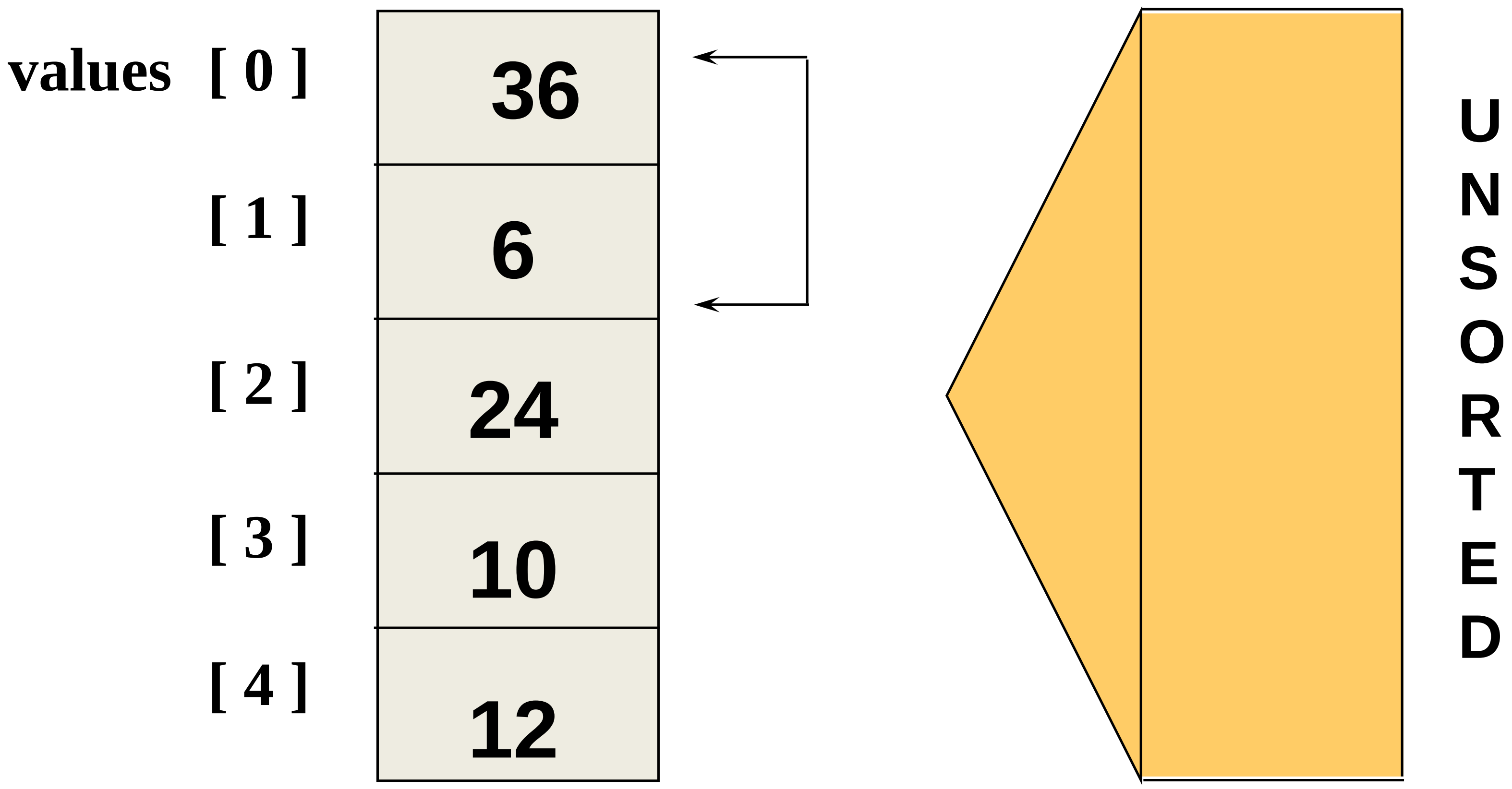
Bubble Sort: Pass One



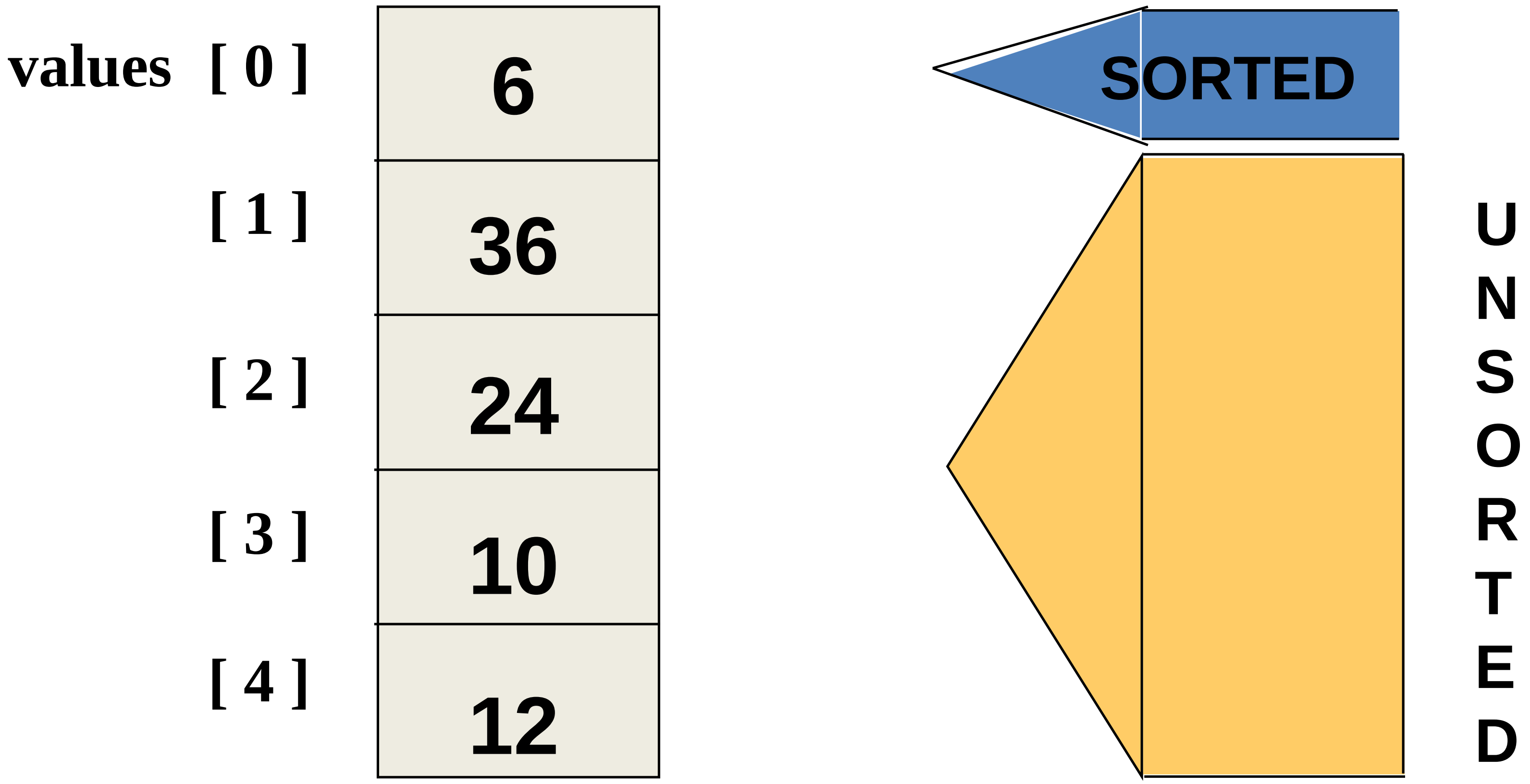
Bubble Sort: Pass One



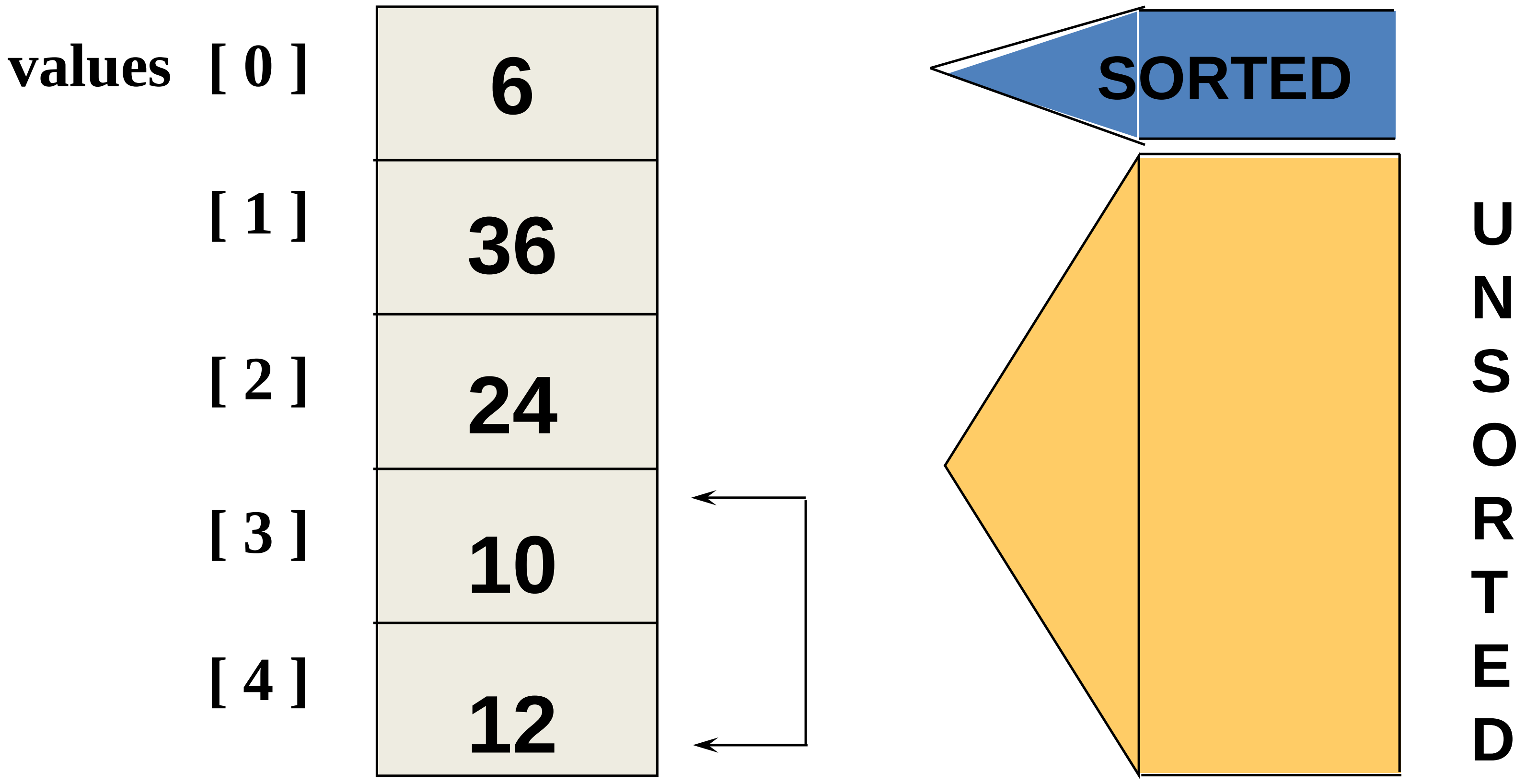
Bubble Sort: Pass One



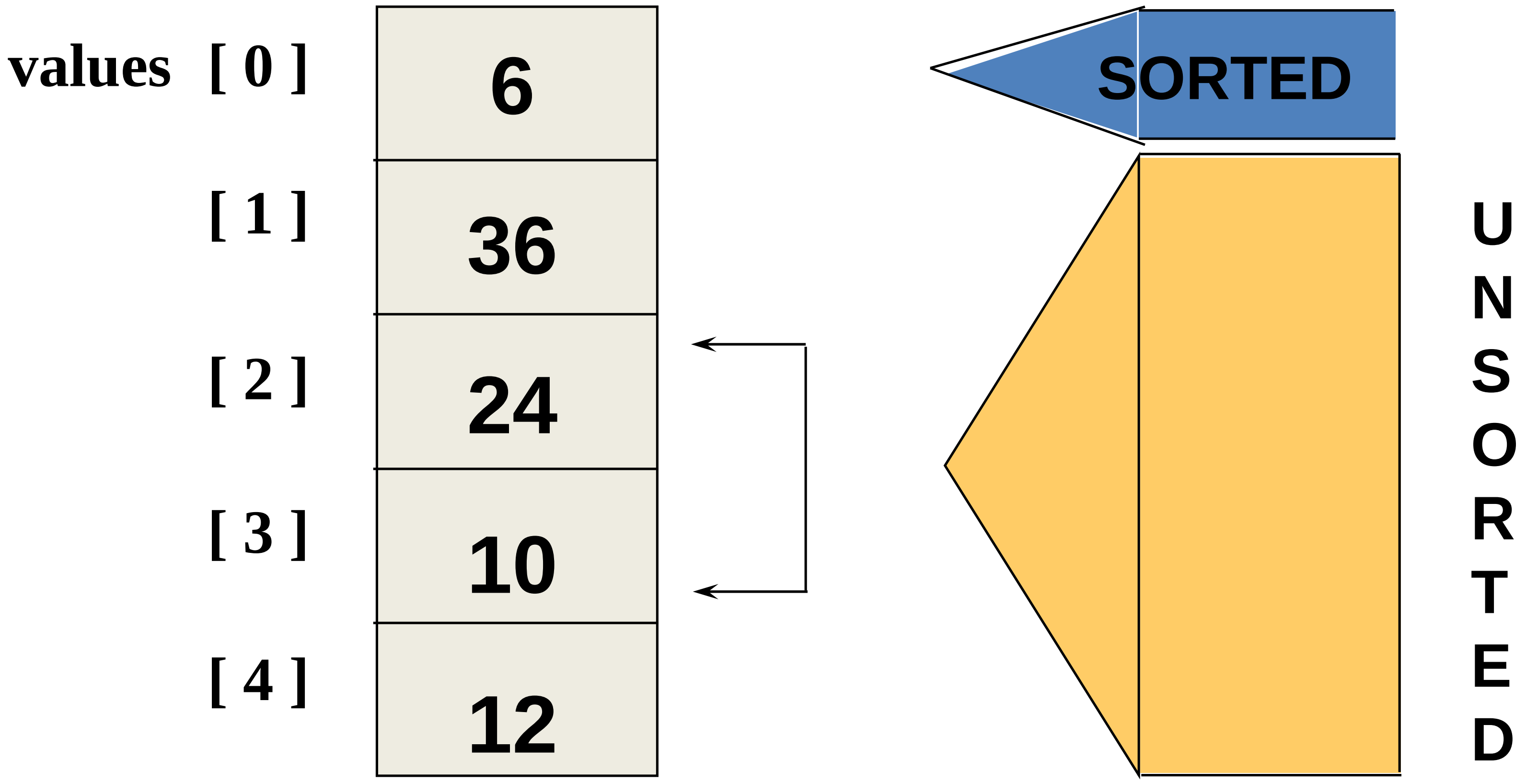
Bubble Sort: End Pass One



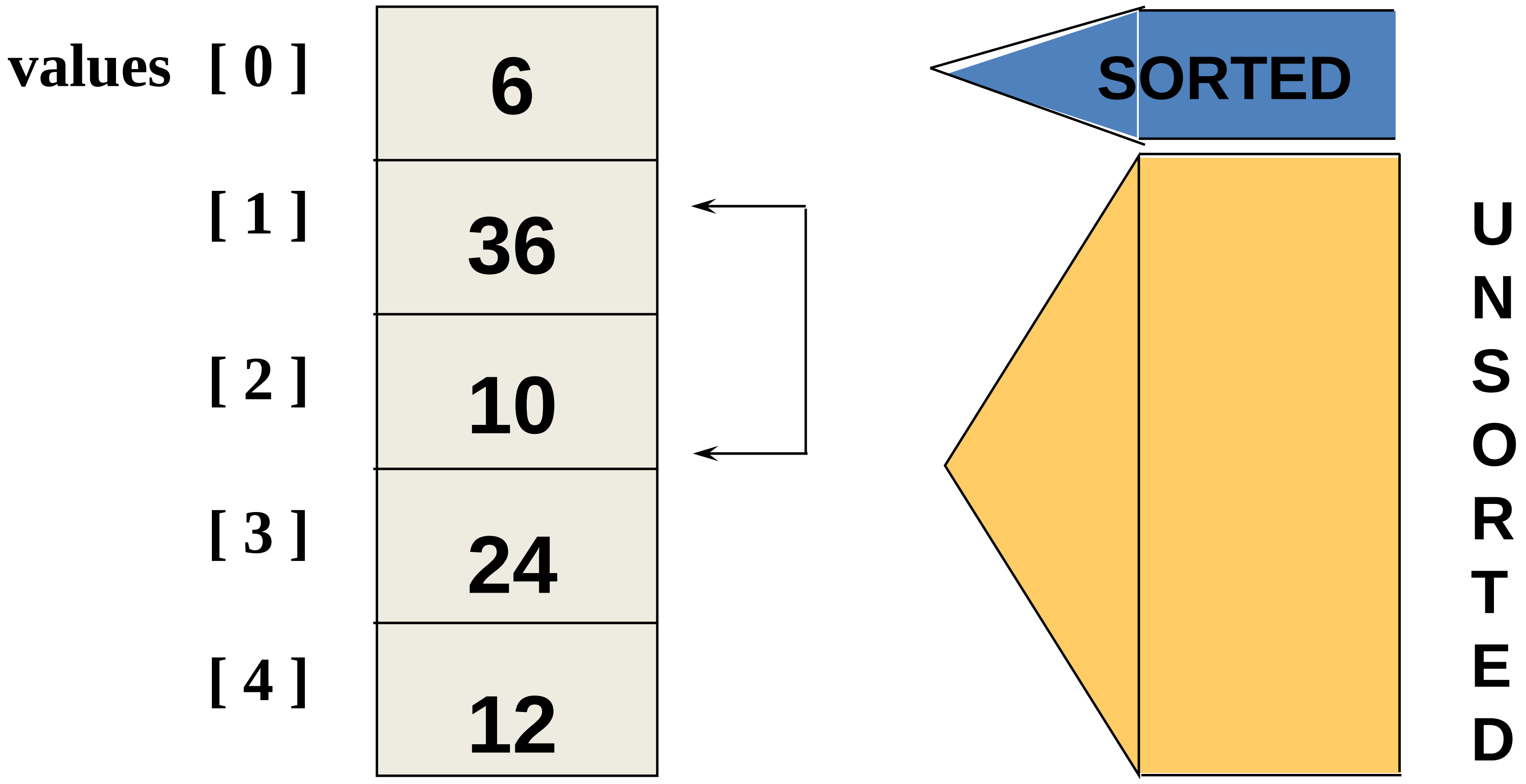
Bubble Sort: Pass Two



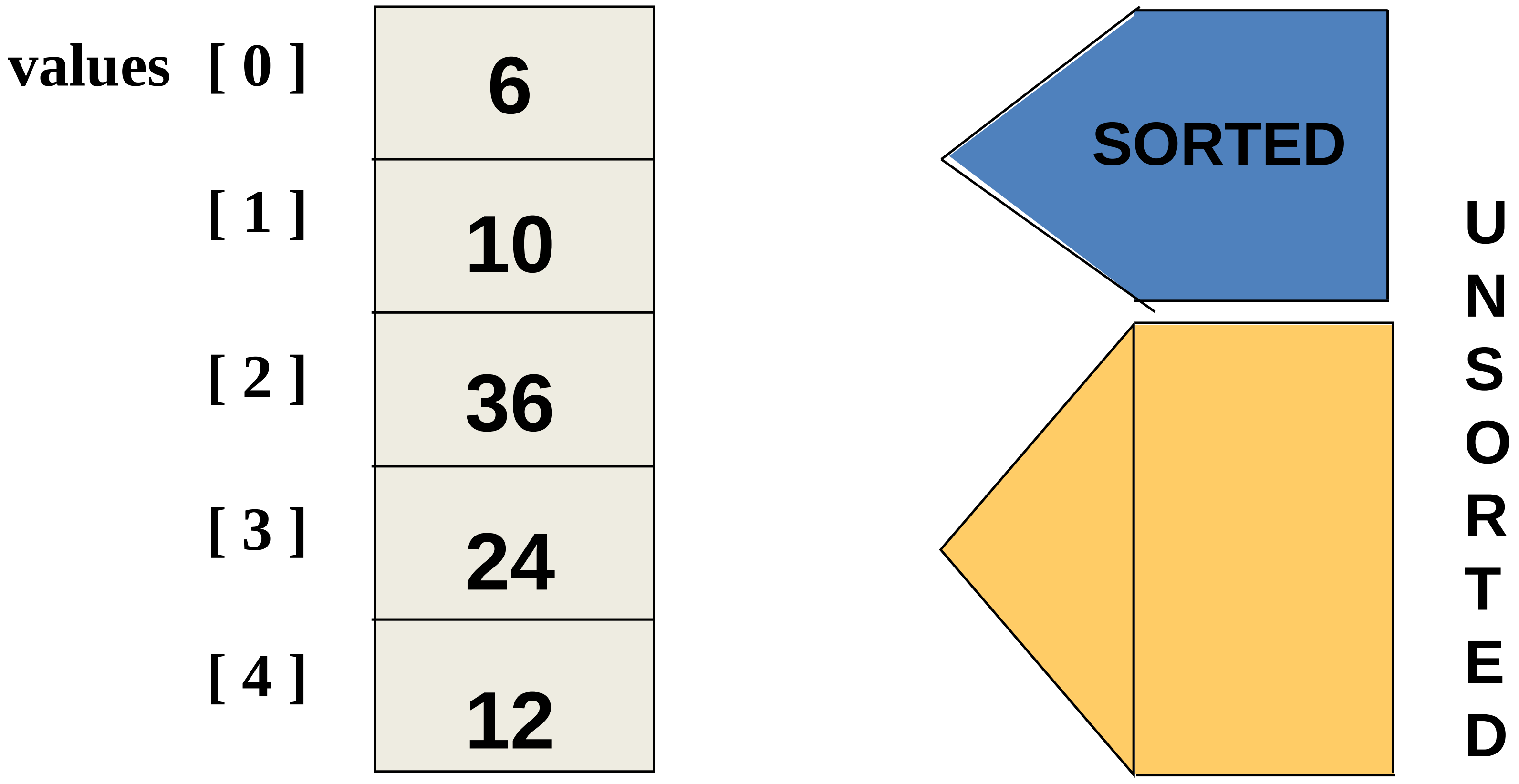
Bubble Sort: Pass Two



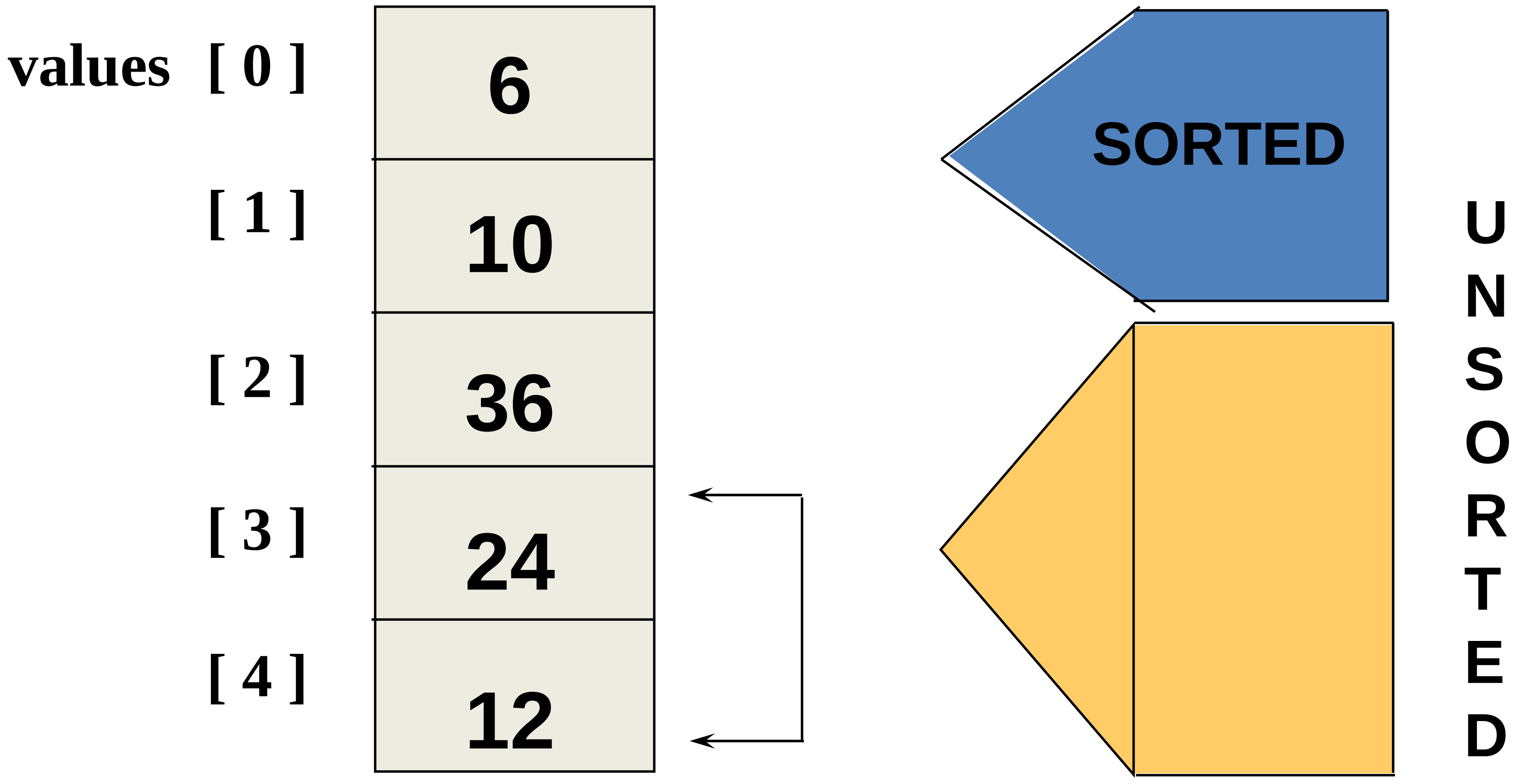
Bubble Sort: Pass Two



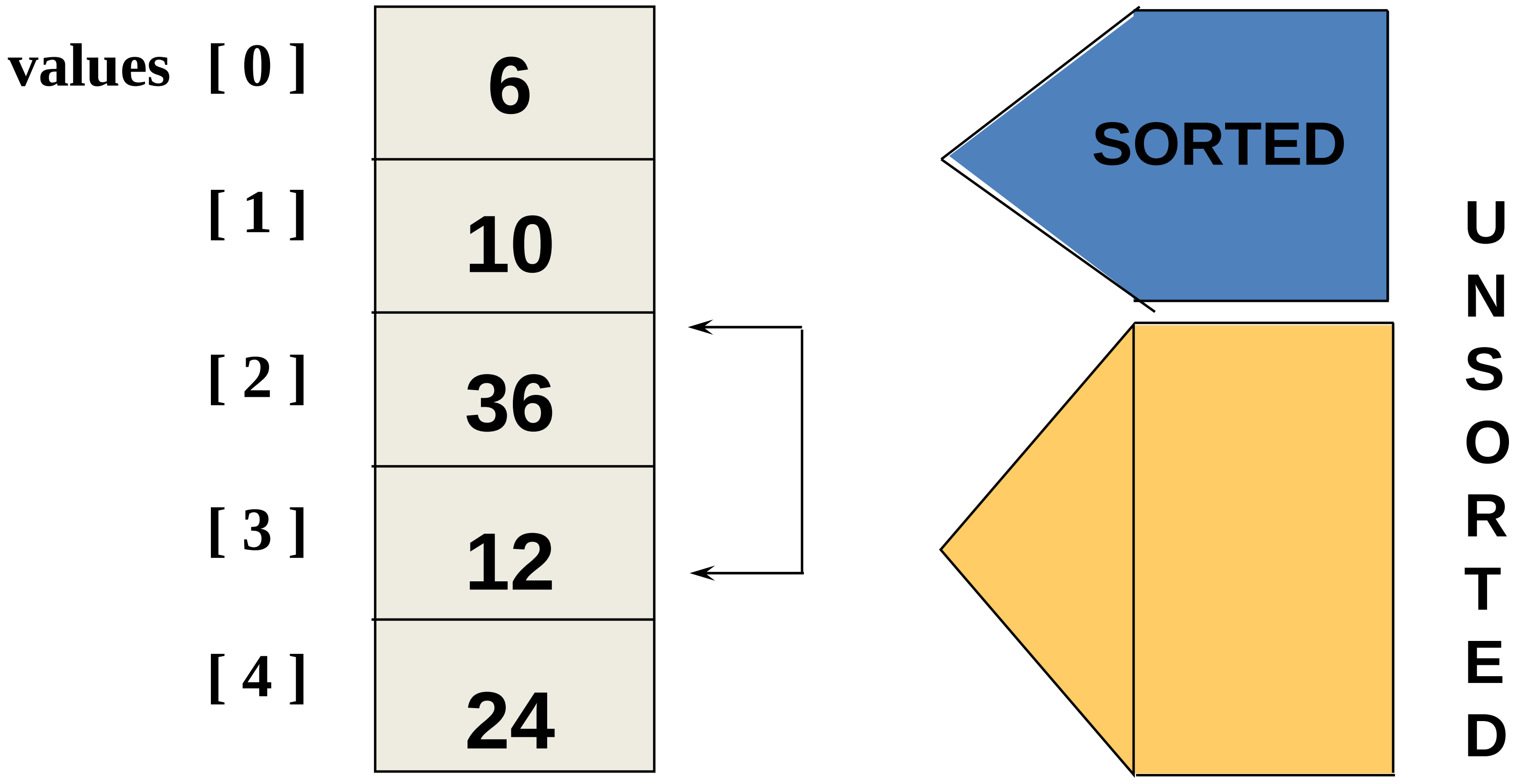
Bubble Sort: End Pass Two



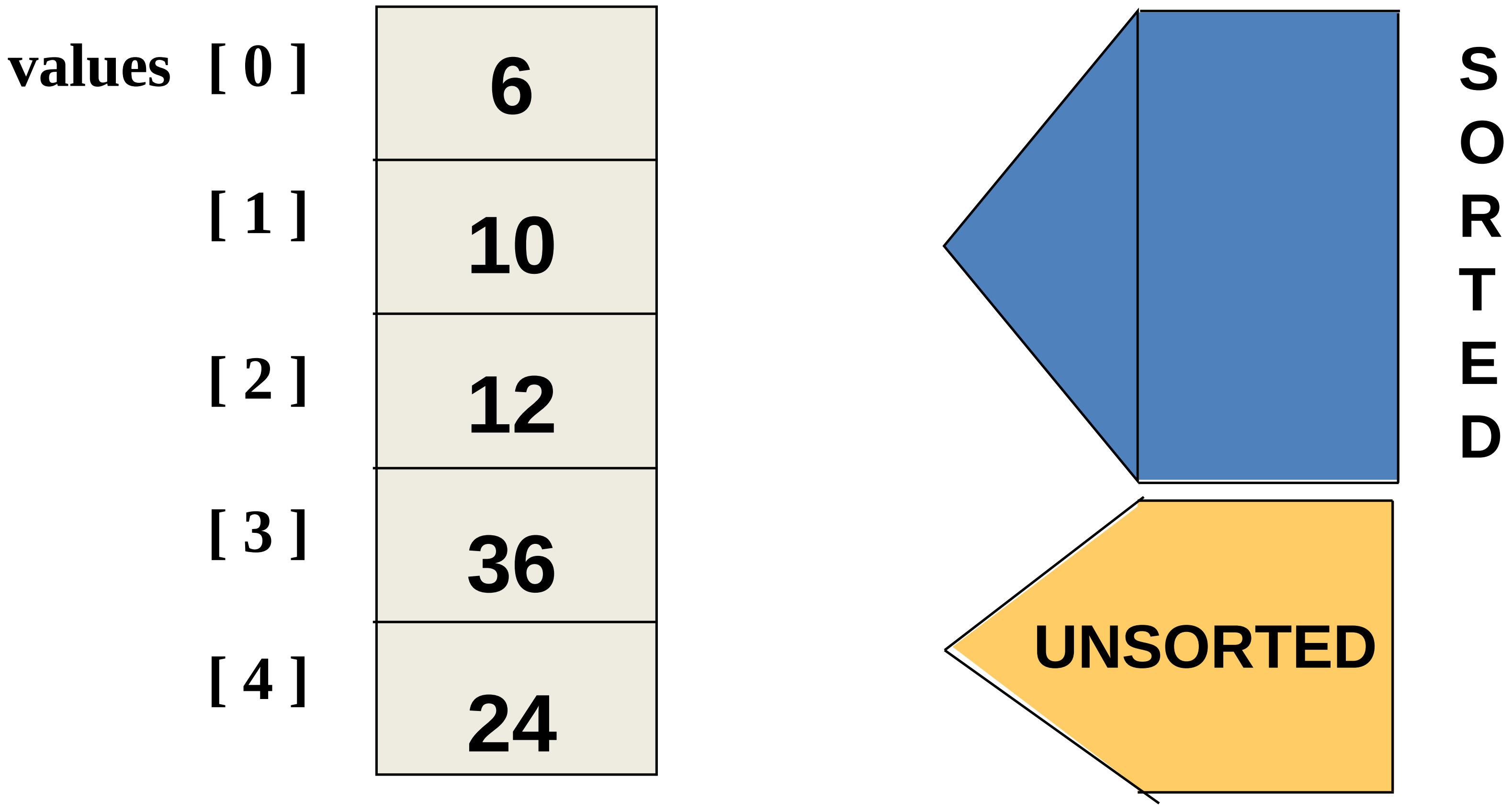
Bubble Sort: Pass Three



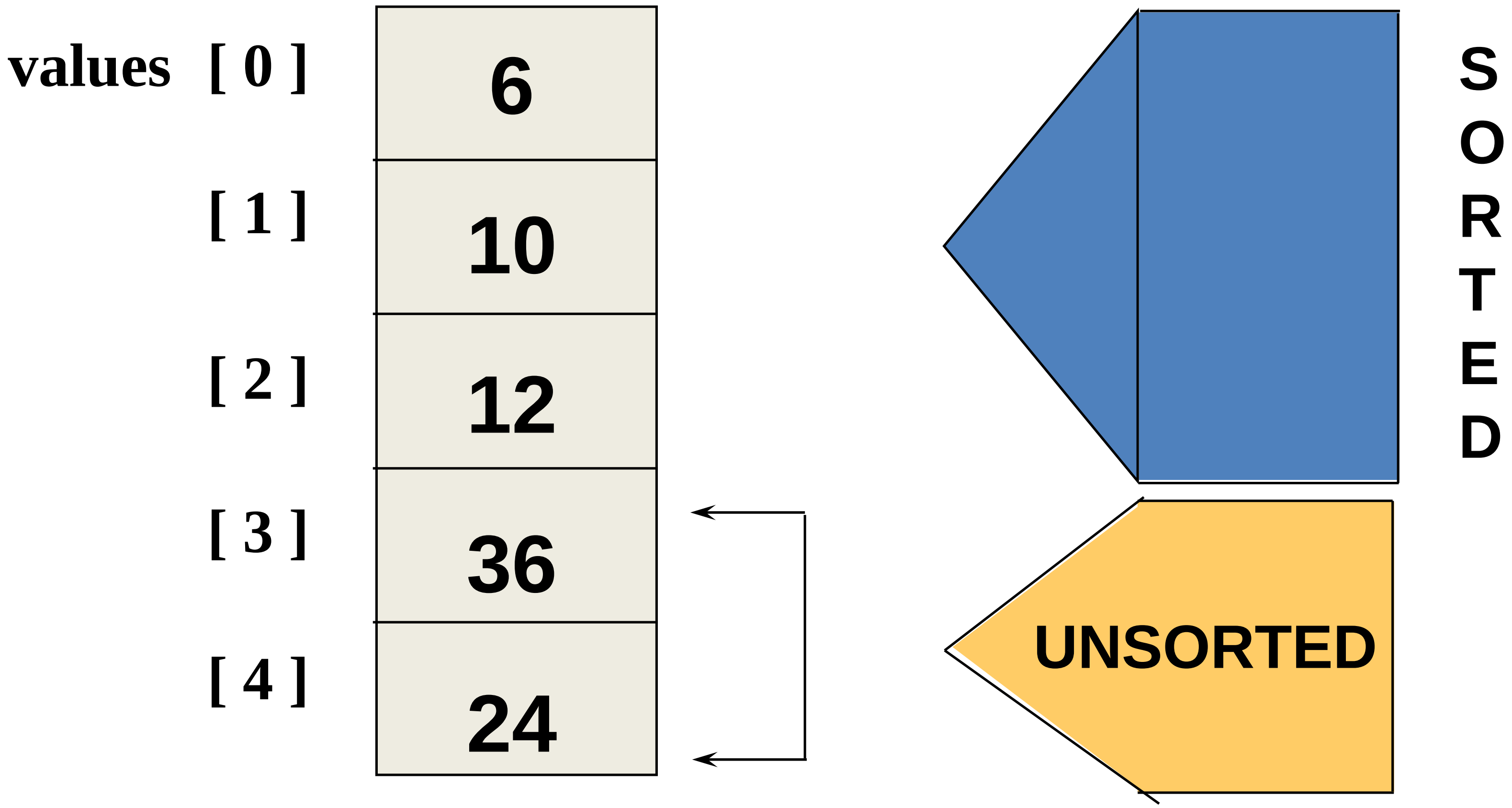
Bubble Sort: Pass Three



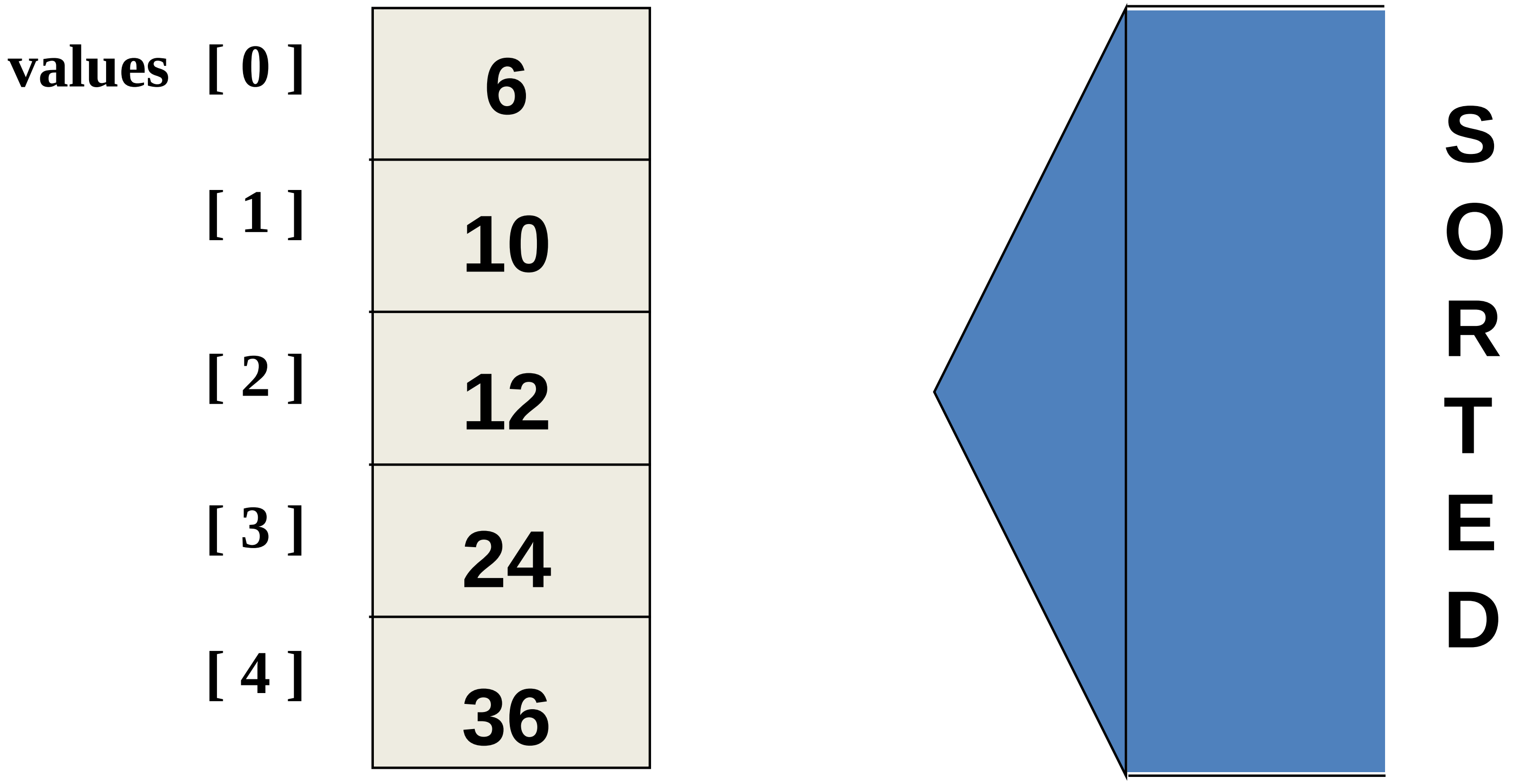
Bubble Sort: End Pass Three



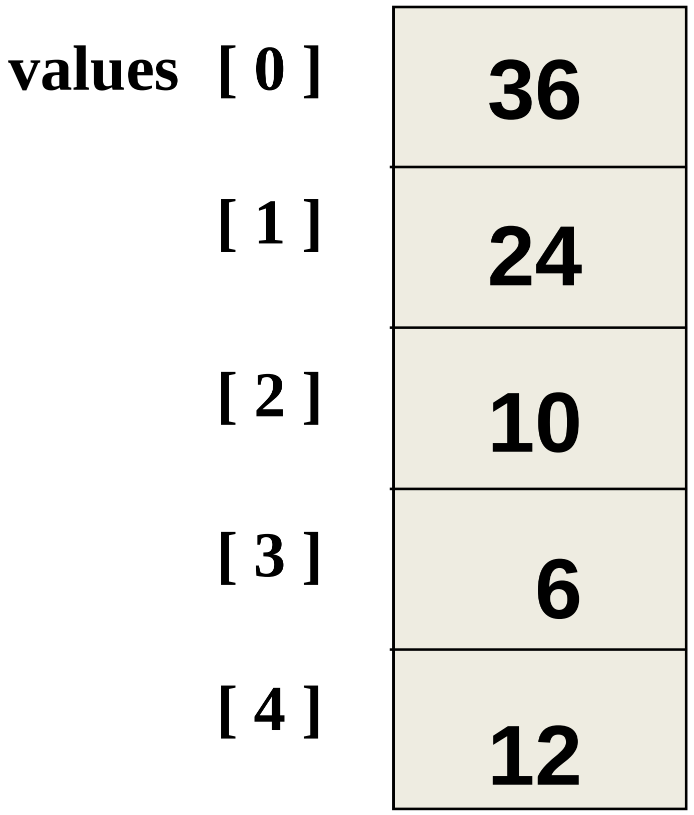
Bubble Sort: Pass Four



Bubble Sort: End Pass Four



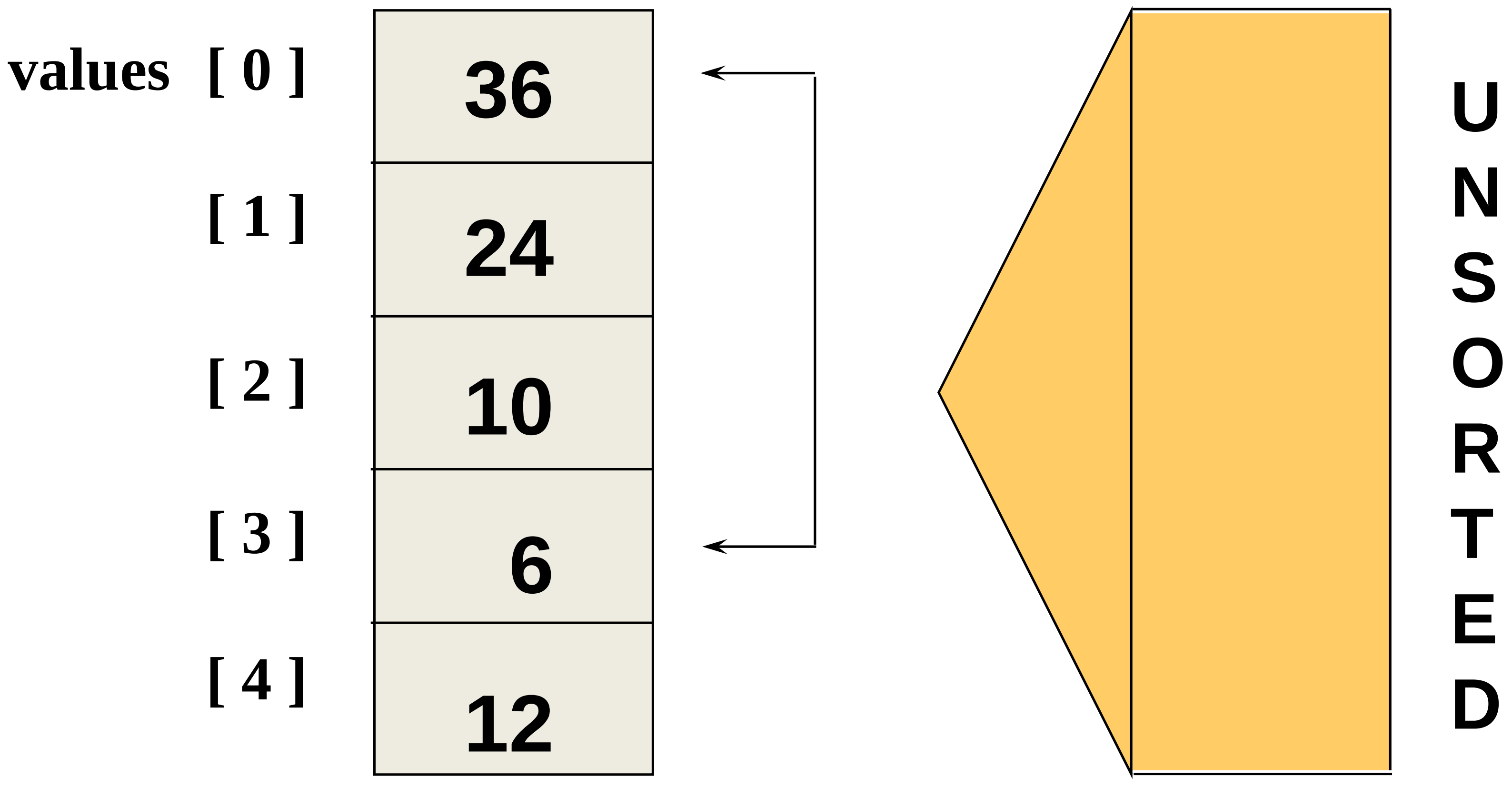
Selection Sort



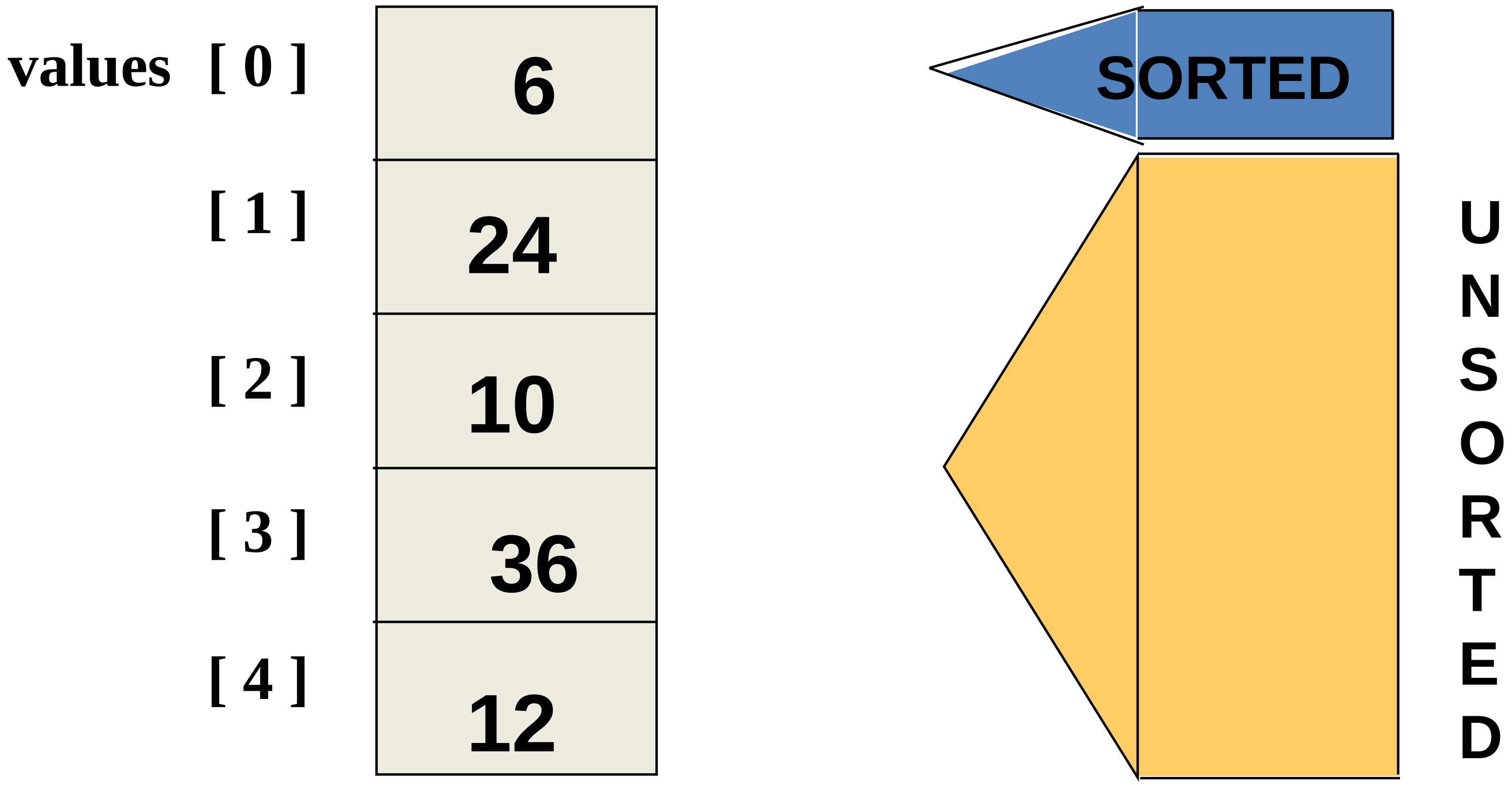
Divides the array into two parts: already sorted, and not yet sorted.

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.

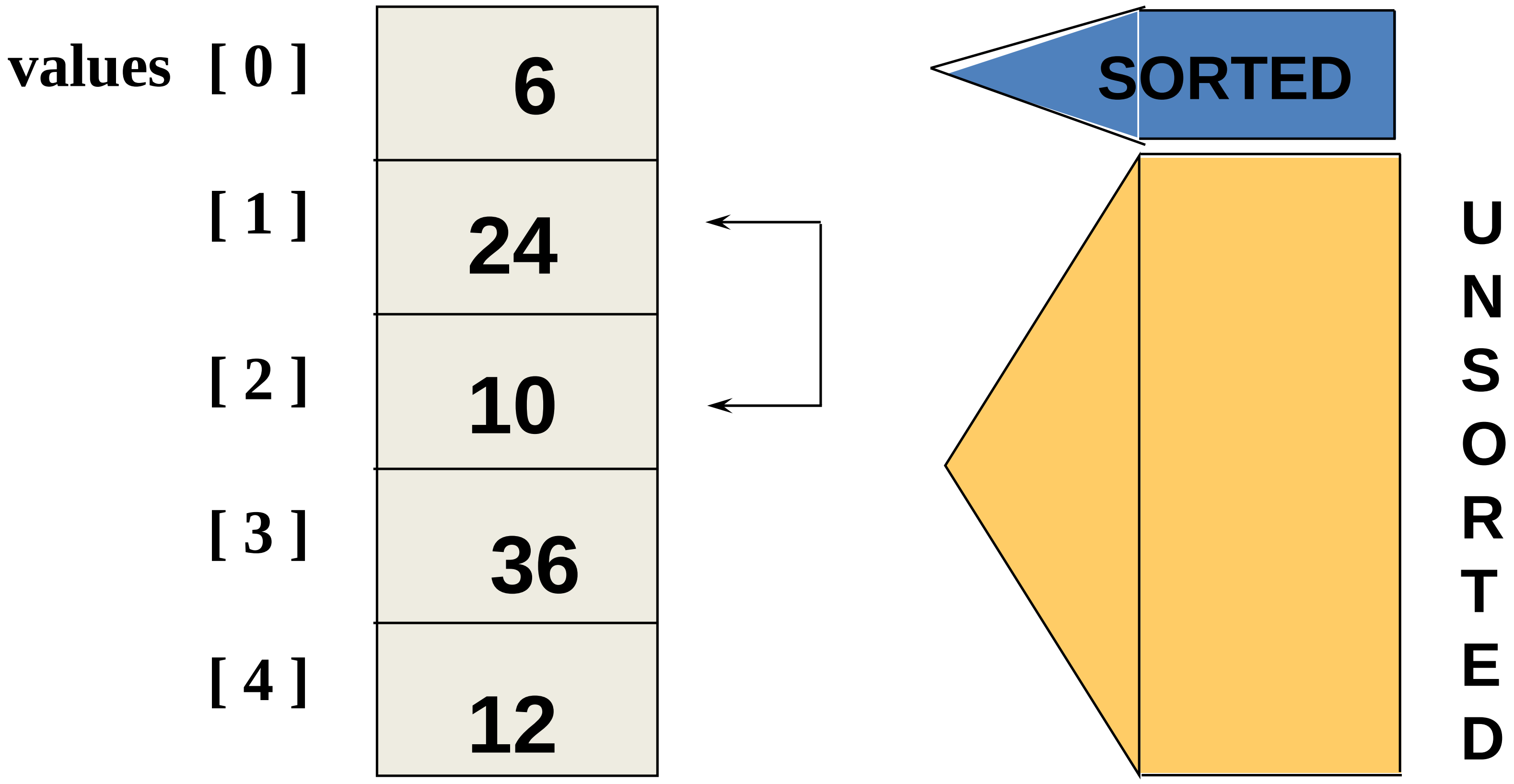
Selection Sort: Pass One



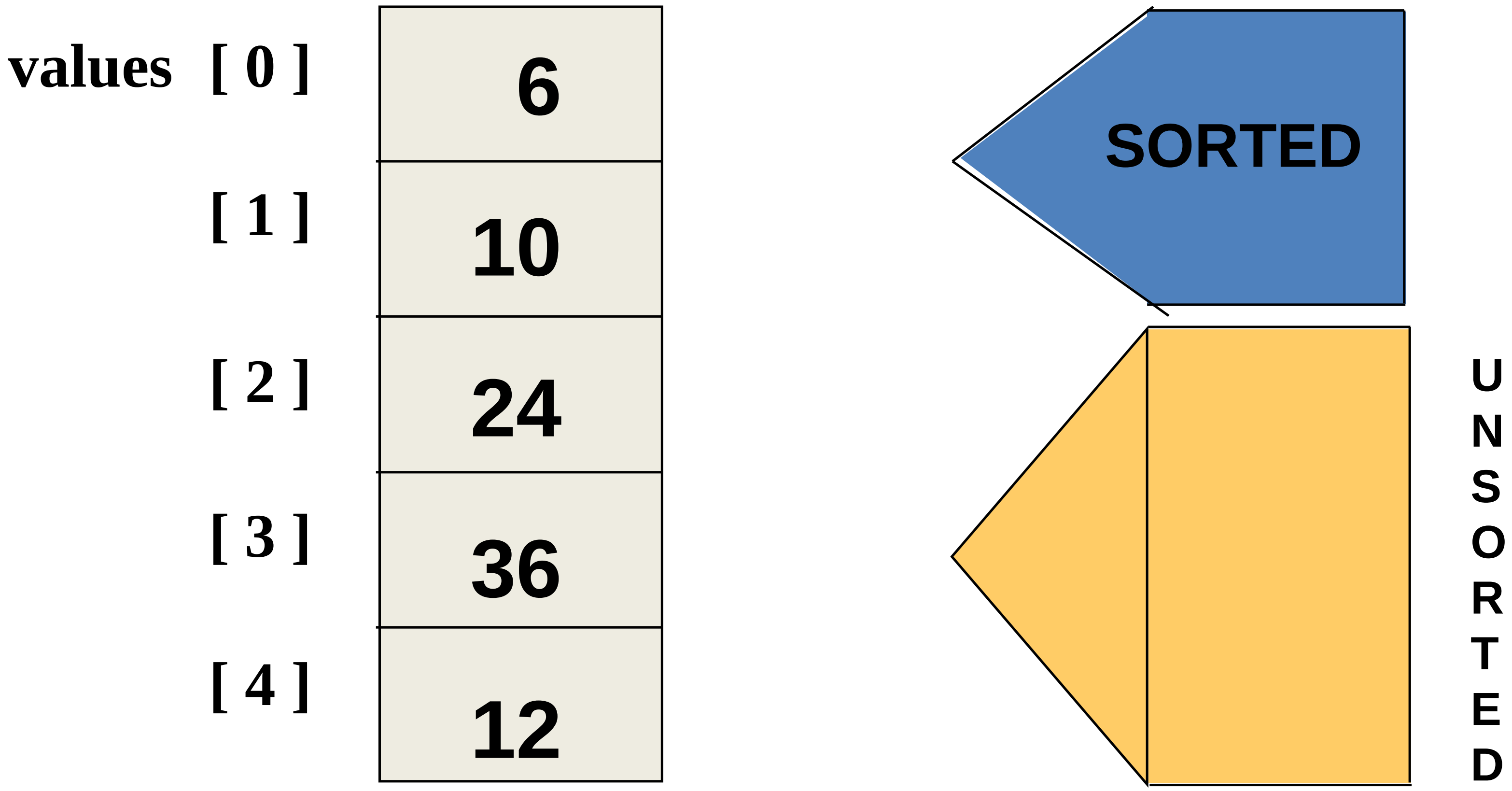
Selection Sort: End Pass One



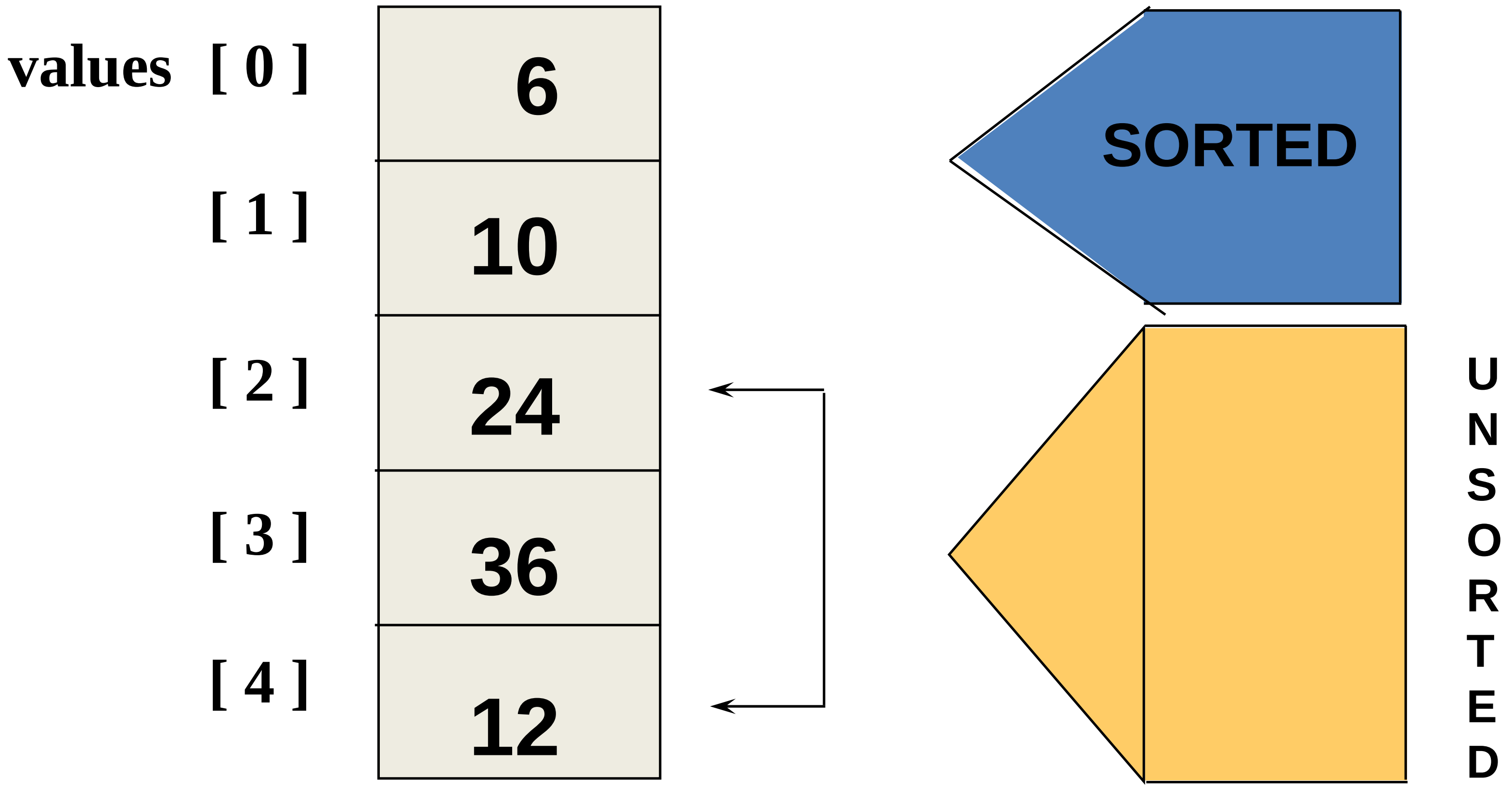
Selection Sort: Pass Two



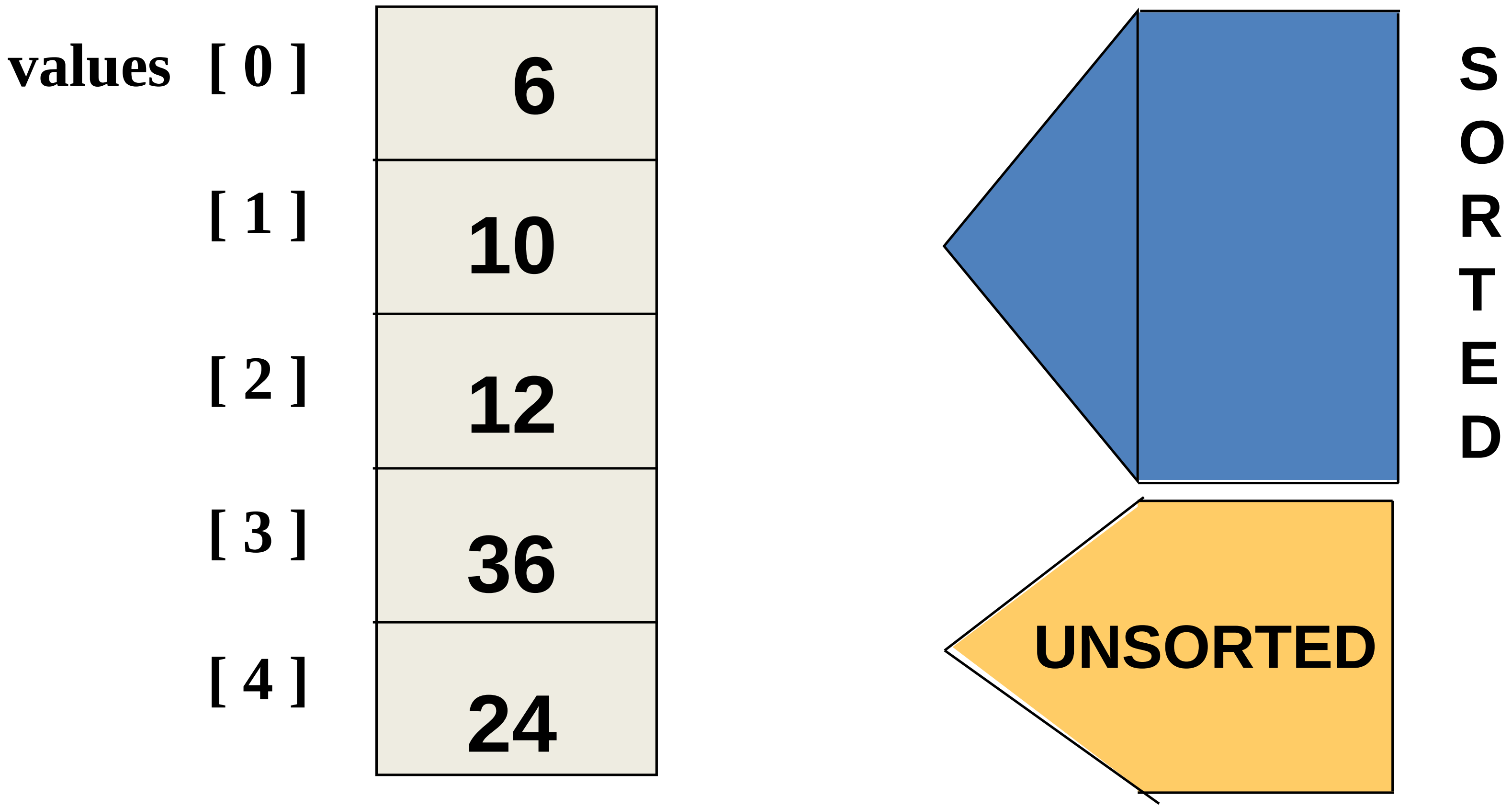
Selection Sort: End Pass Two



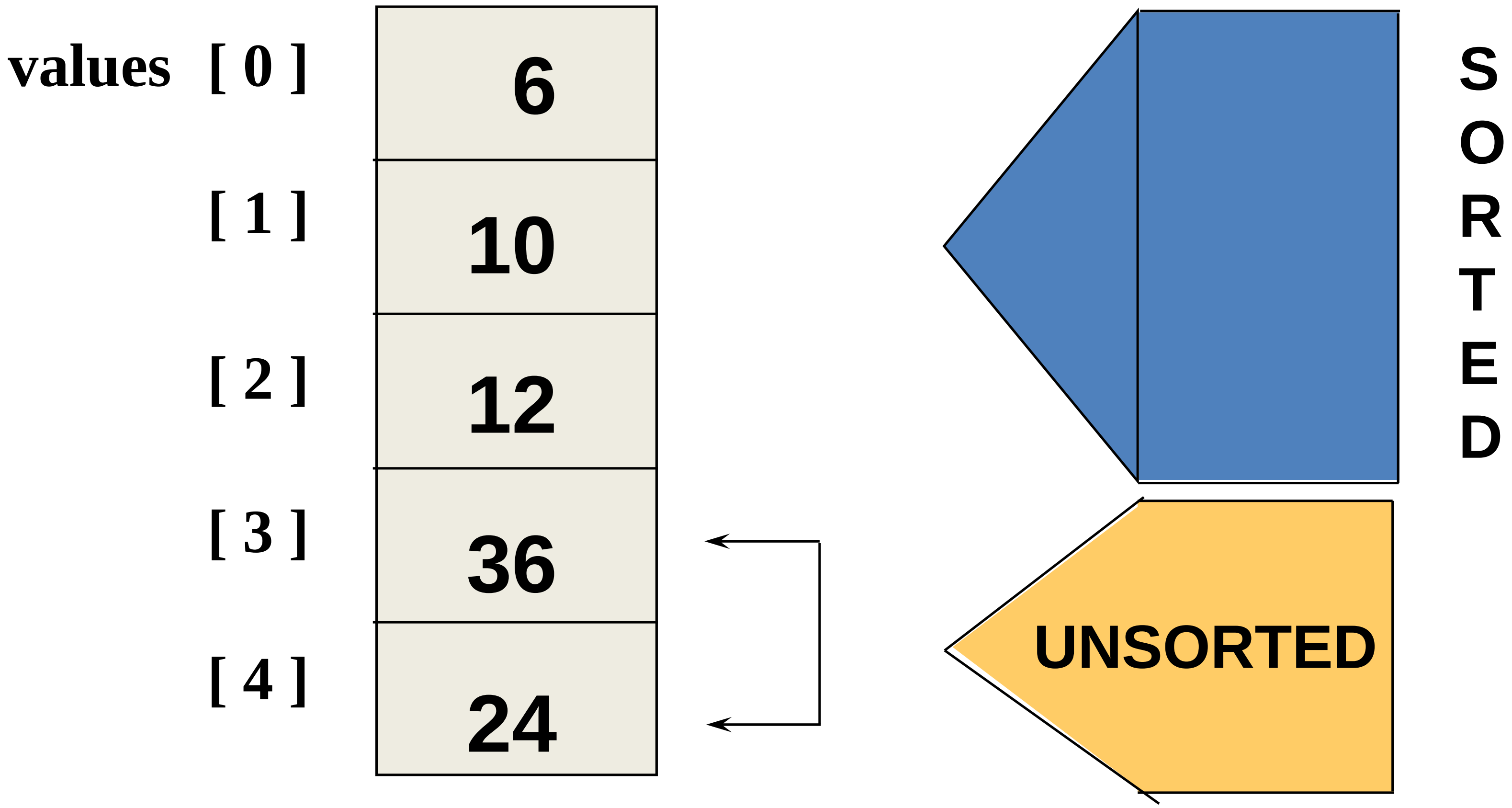
Selection Sort: Pass Three



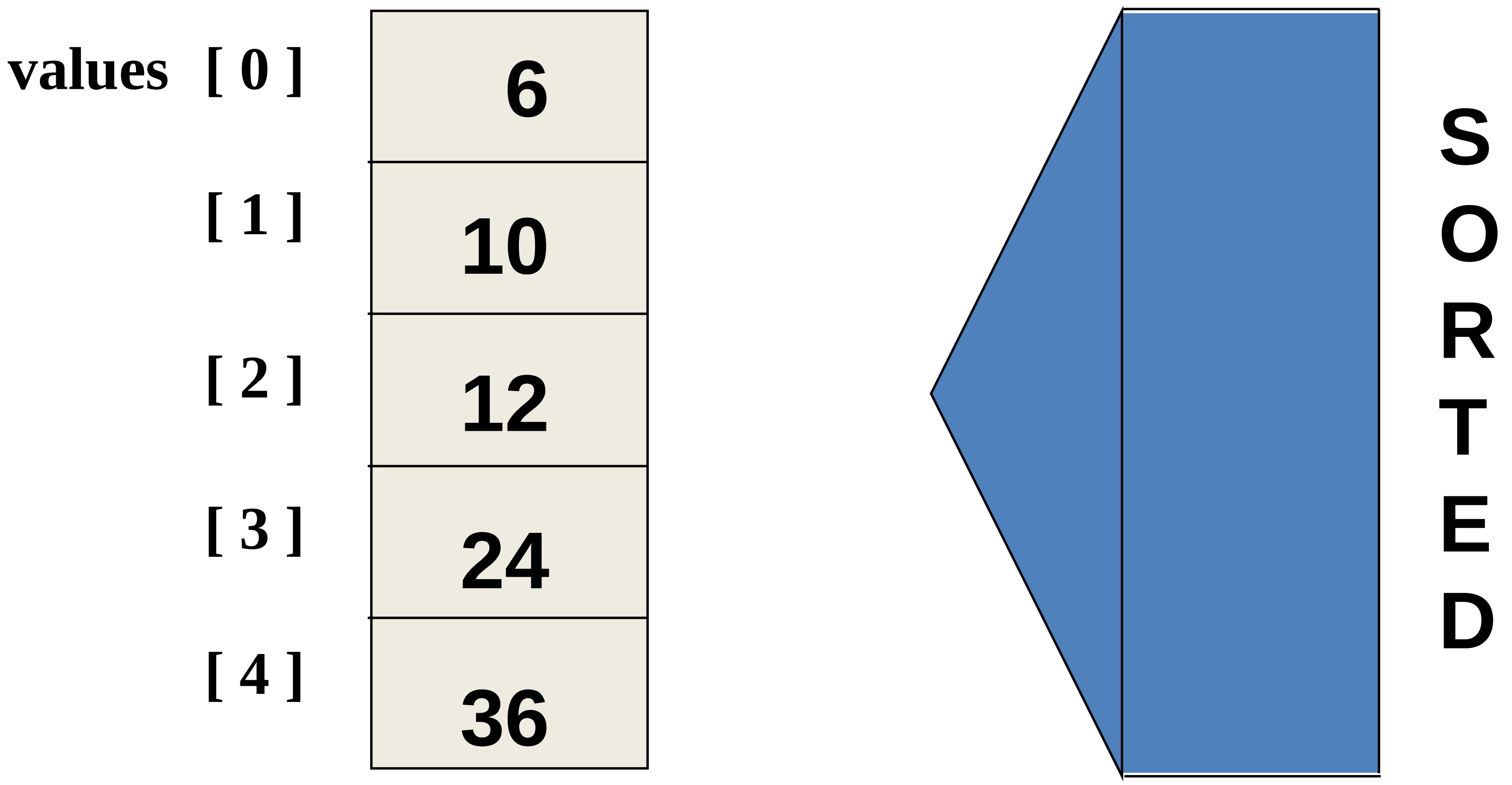
Selection Sort: End Pass Three



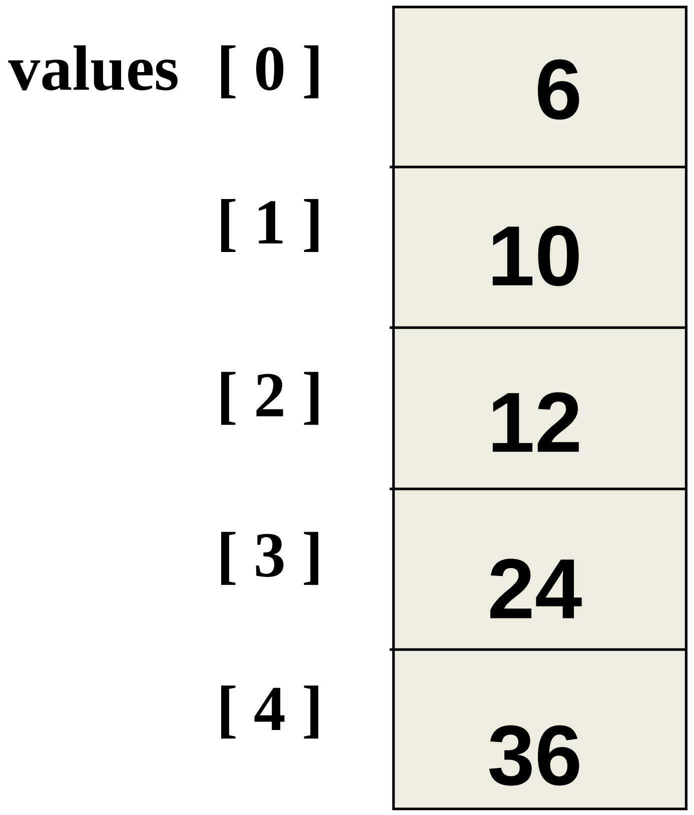
Selection Sort: Pass Four



Selection Sort: End Pass Four



Selection Sort: How many comparisons?



4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

1 compare for values[3]

= 4 + 3 + 2 + 1

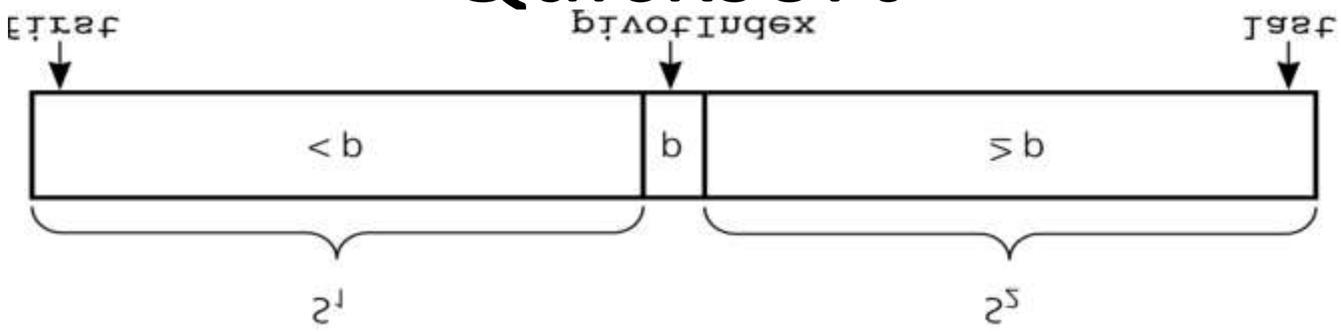
Quicksort

Another divide-and-conquer sorting algorihm

To understand quick-sort, let’s look at a high-level description of the algorithm

1) **Divide** : If the sequence S has 2 or more elements, select an element x from S to be your **pivot**. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:
   - L, holds S’s elements less than x
   - E, holds S’s elements equal to x
   - G, holds S’s elements greater than x
2) **Recurse**: Recursively sort L and G
3) **Conquer**: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.

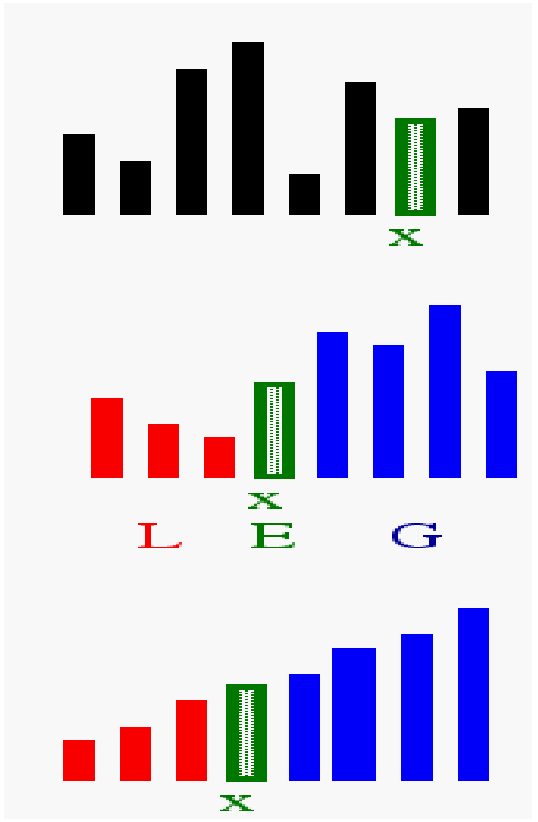
Quicksort



1) **Select**: pick an element

2) **Divide**: rearrange elements so that x goes to its final position E

3) **Recurse and Conquer**: recursively sort



Quicksort

- Analysis
  - quicksort is usually extremely fast in practice
  - Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property (the key in the root is larger than that in either child and both subtrees have the heap 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).**

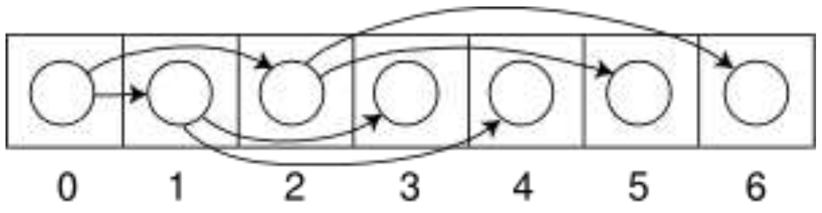
Heap Sort Approach: Binary heap revisited

- Binary heaps are a particularly simple kind of heap data structure created using a binary tree.
- It can be seen as a binary tree with two additional constraints:
  - The *shape property*: the tree is either a perfectly balanced binary tree (all leaves are at the same level), or, if the last level of the tree is not complete, the nodes are filled from left to right.
  - The *heap property*: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.
- Heaps where the comparison function is mathematical greater than are called max-heaps; those where the comparison function is mathematical less than are called min-heaps
- Note that the ordering of siblings in a heap is not specified by the heap property, so the two children of a parent can be freely interchanged (as long as this does not violate the shape property)

Heap Sort Approach: Binary heap revisited

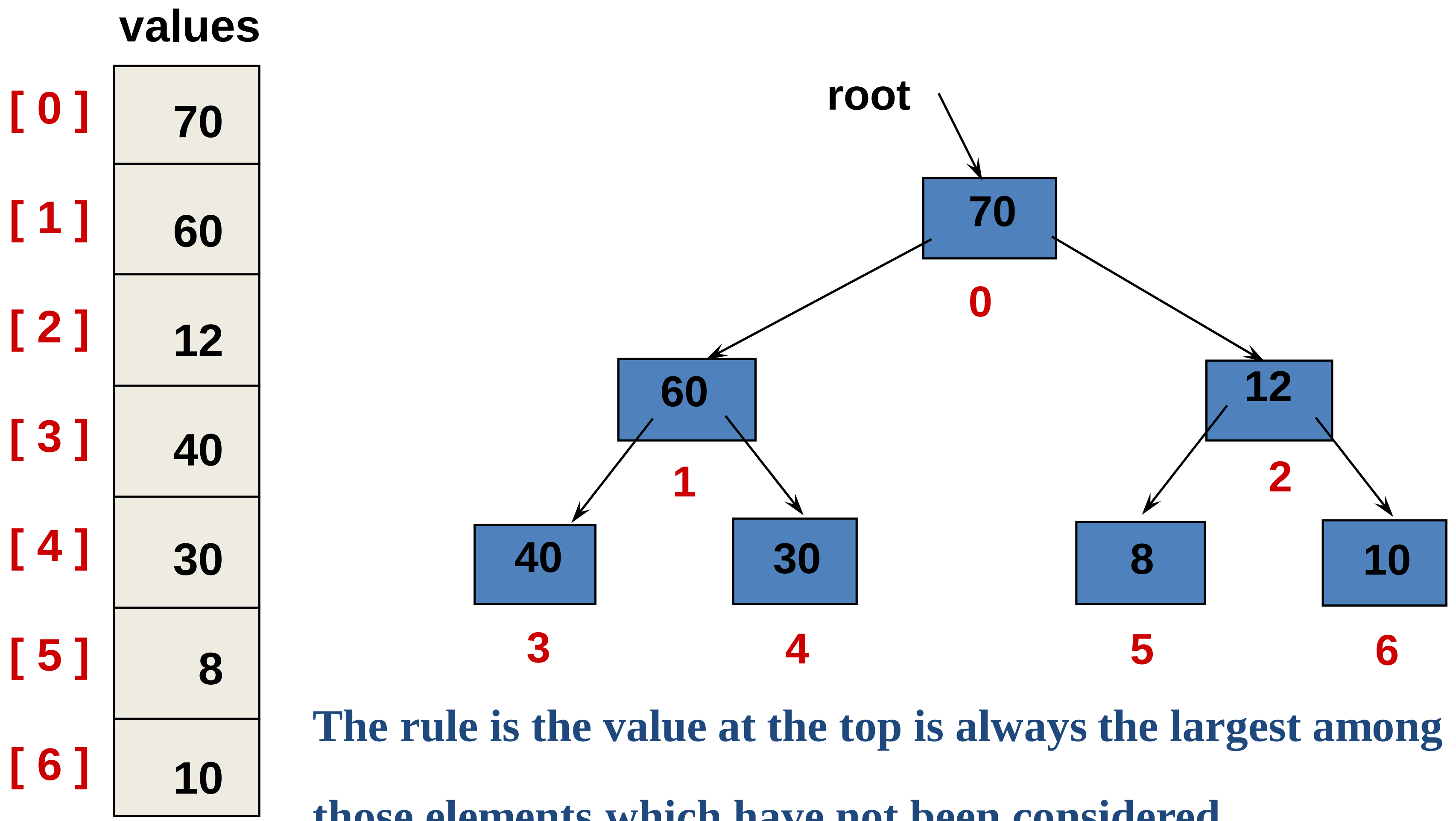
Implementation using arrays

- Because a heap is always an almost complete binary tree, it can be stored compactly.
- For each index *i*, element *a*[*i*] is the parent of two children *a*[2*i*+1] and *a*[2*i*+2, as shown in the figure below.



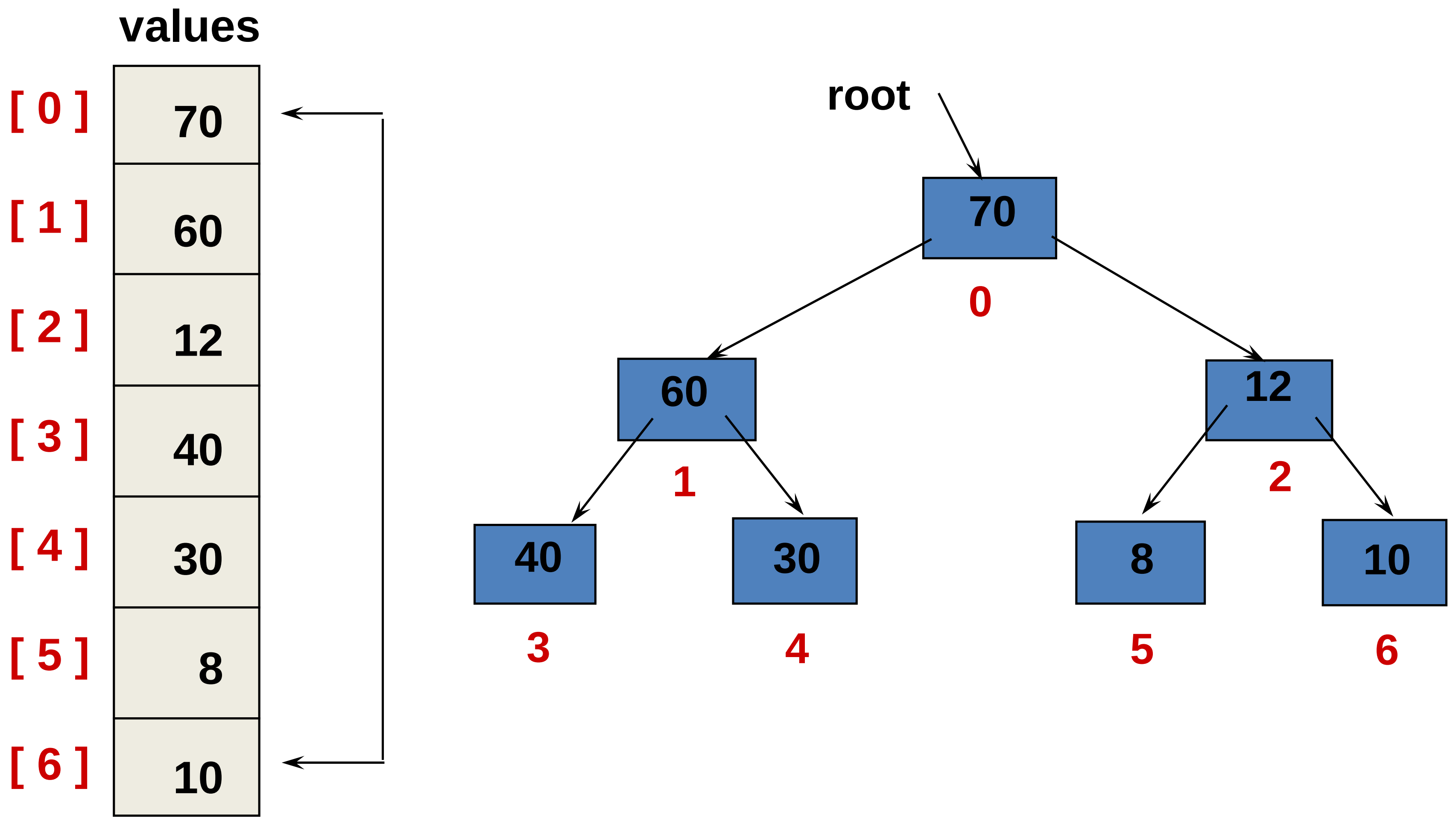
- Note that with an implementation starting at 0 for the root, the parent of *a*[*i*] is *a*[floor((*i*–1)/2)].
- Note that in the figure above, the node at position 2 has children at position 5 and 6.
- All the other following nodes can't have children because they would lie outside the array.

After creating the original heap

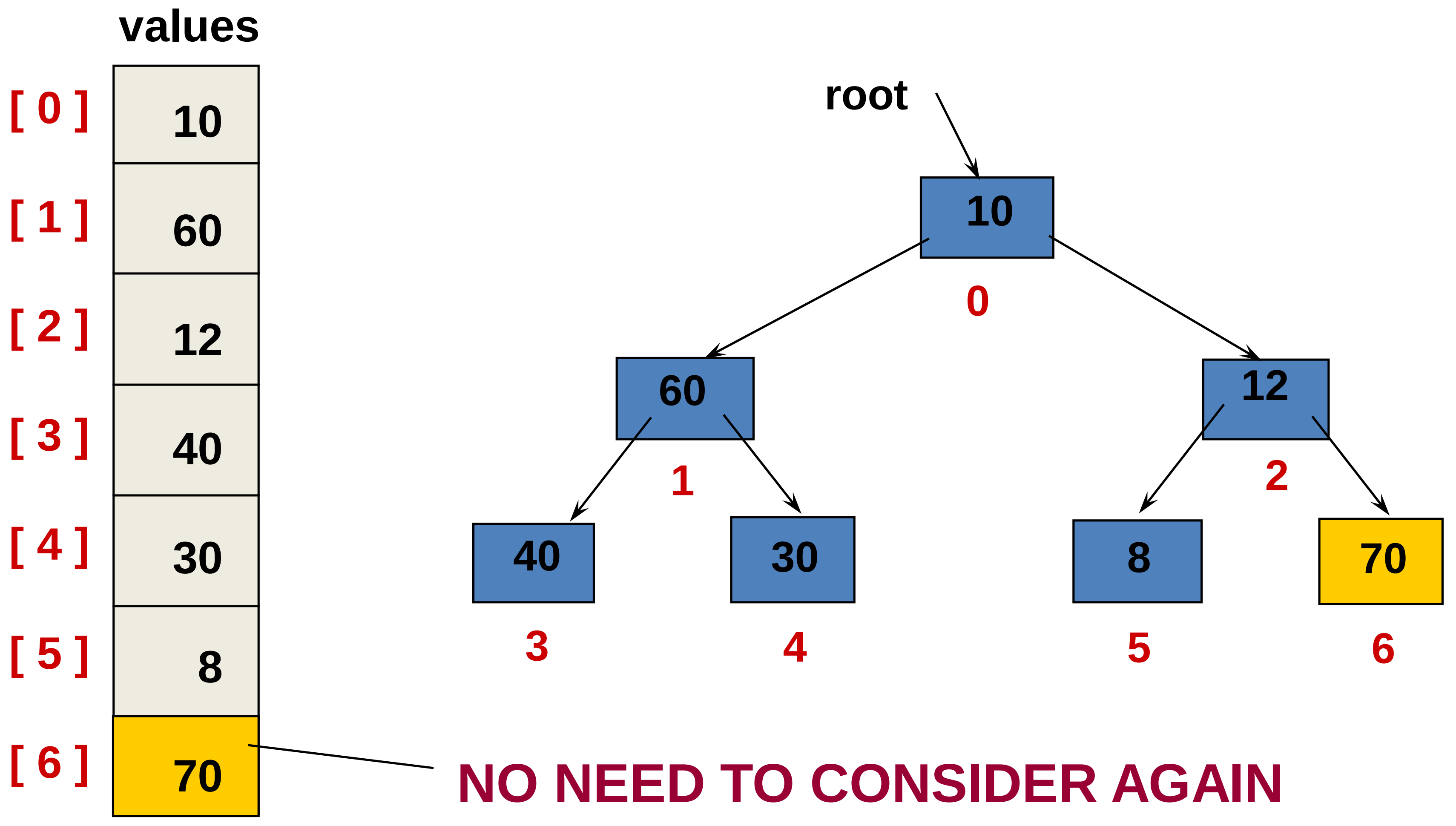


The rule is the value at the top is always the largest among those elements which have not been considered

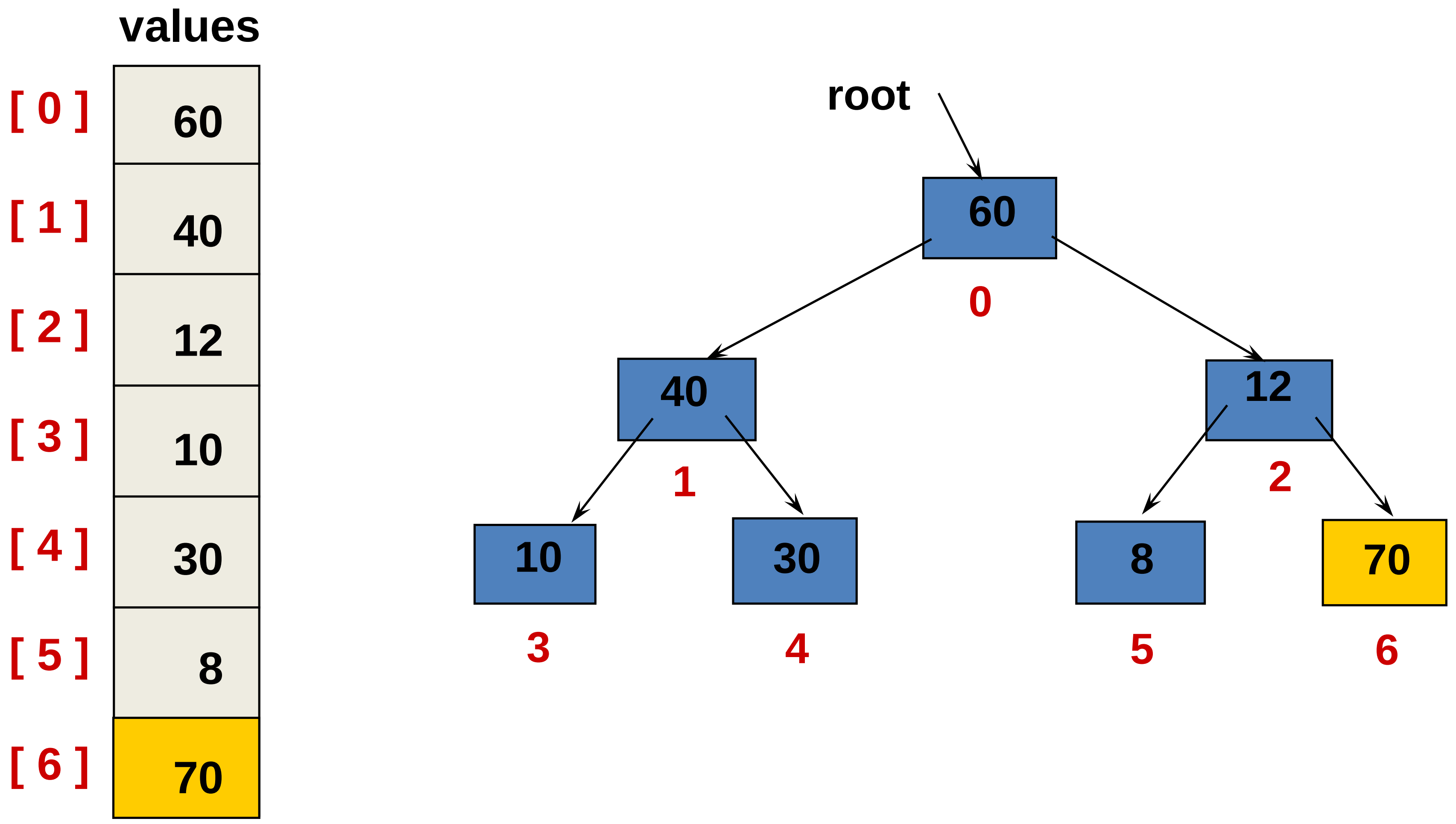
Swap root element into last place in unsorted array



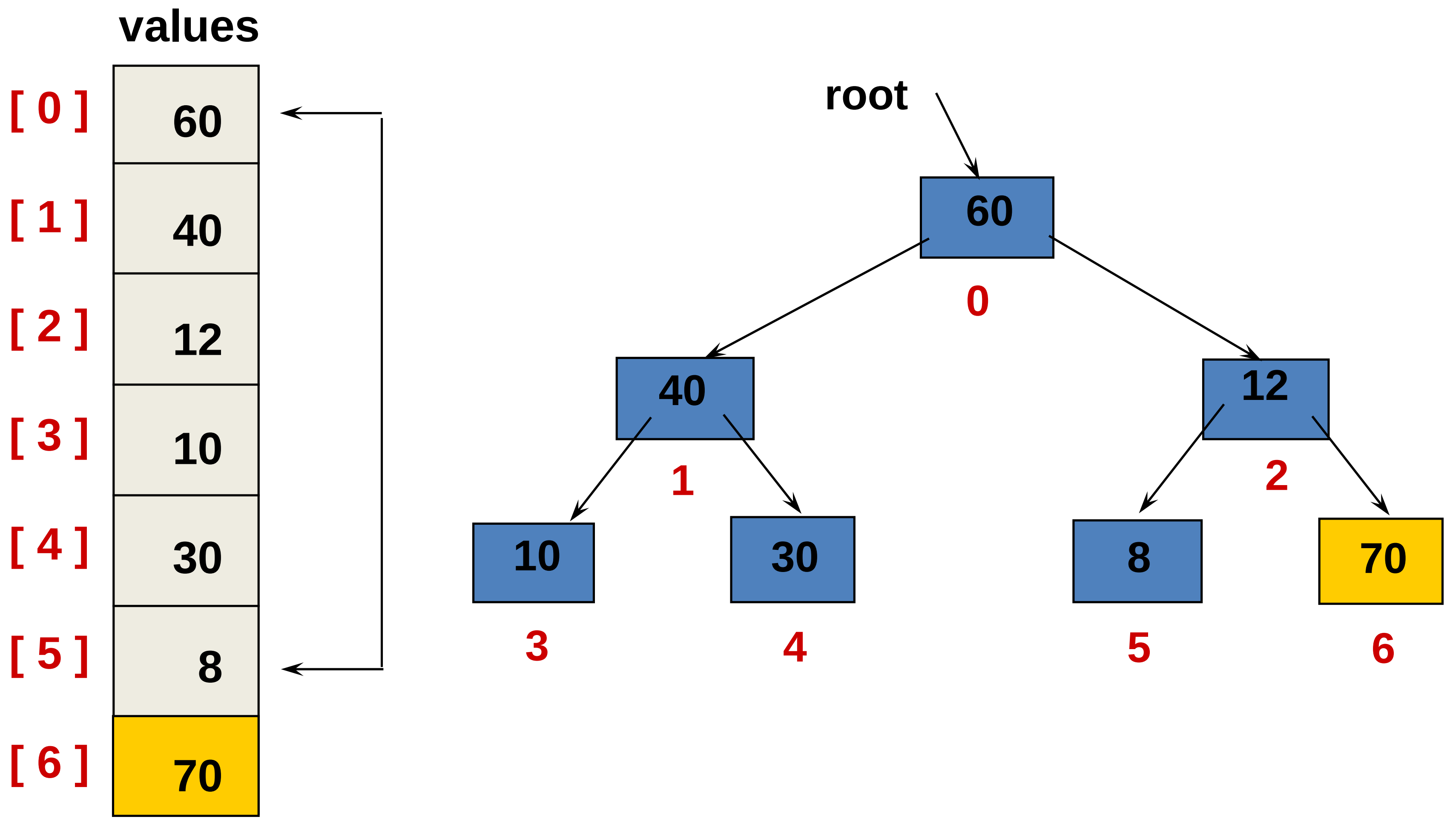
After swapping root element into its place



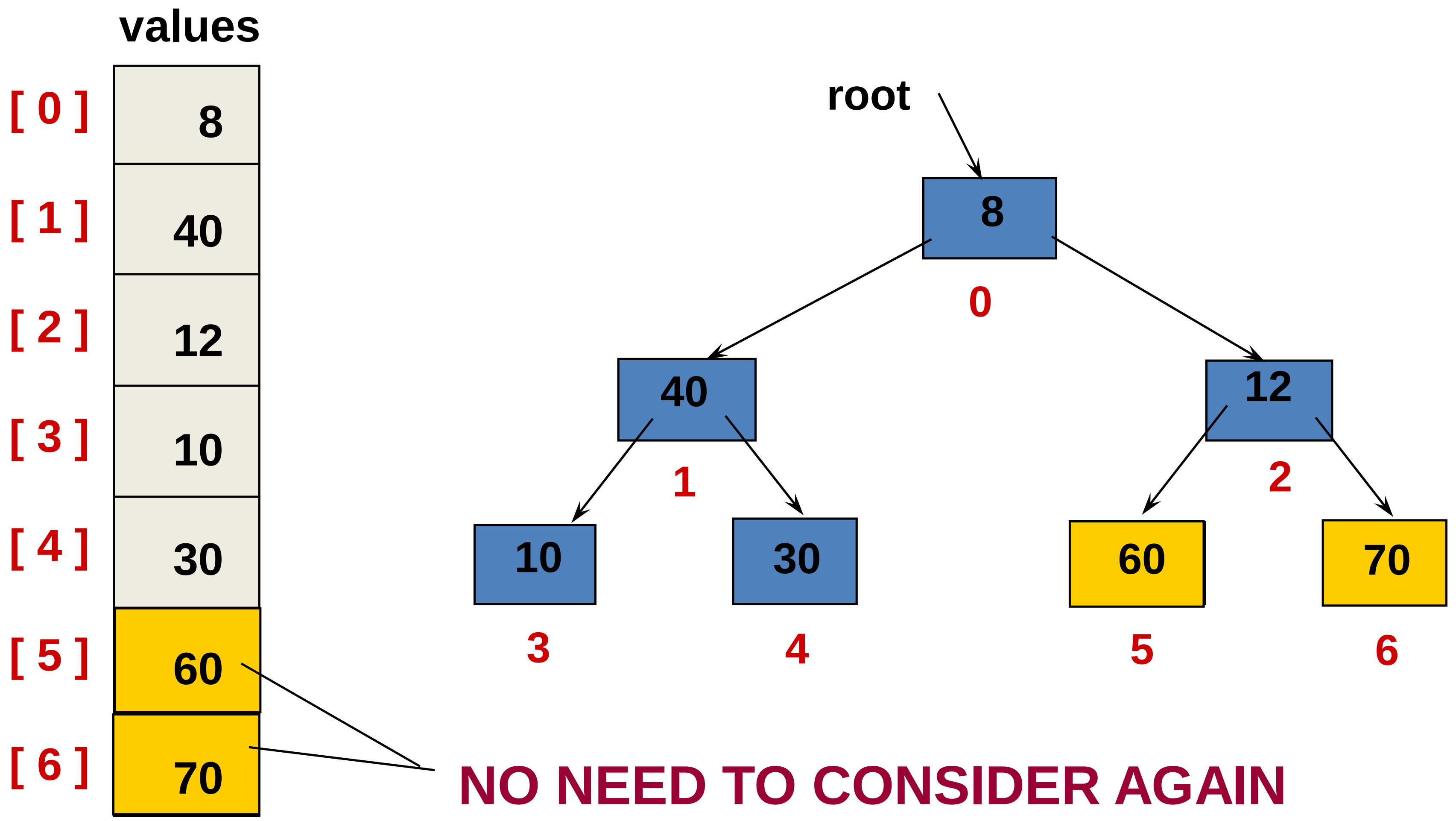
After reheaping remaining unsorted elements



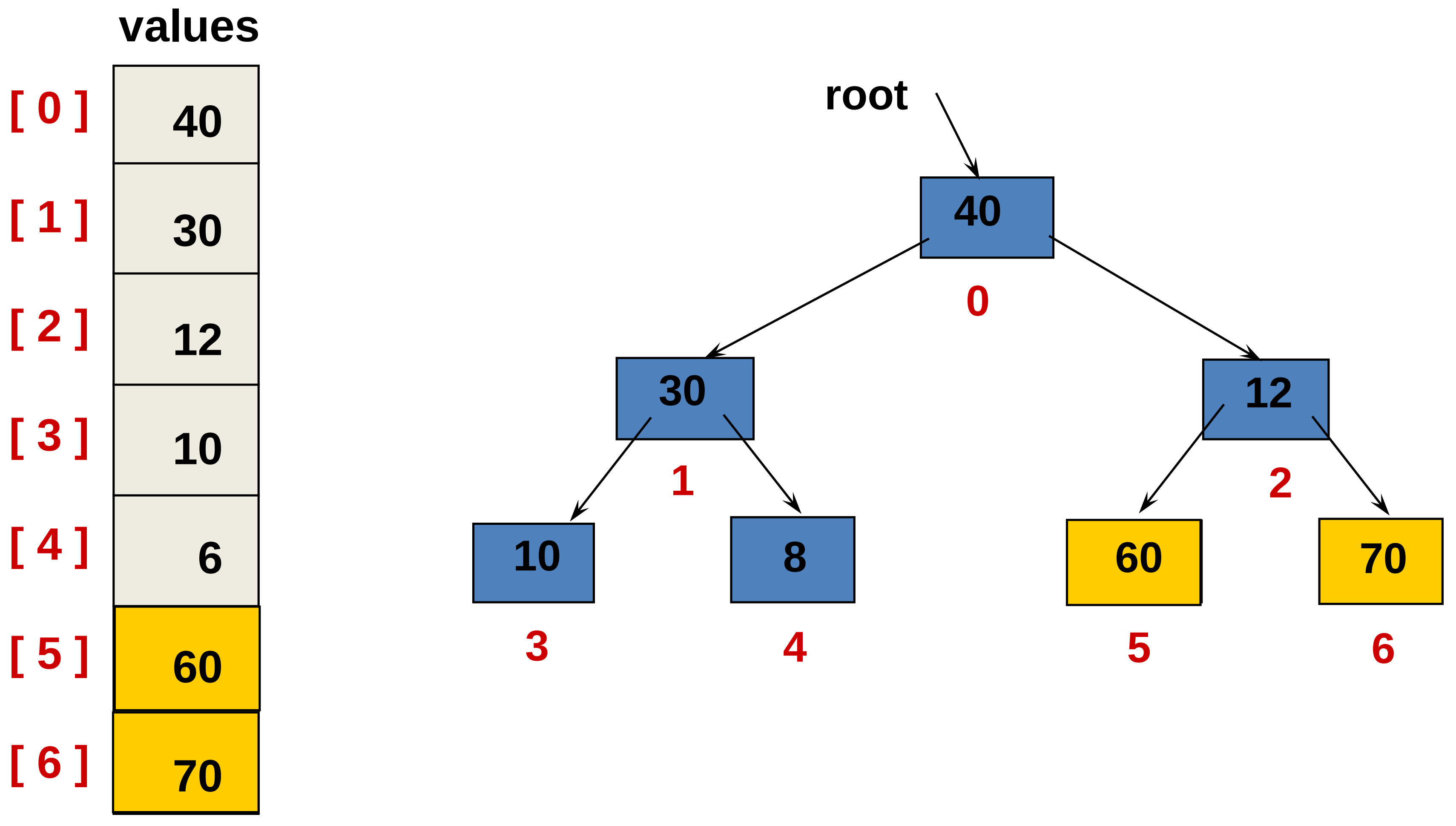
Swap root element into last place in unsorted array



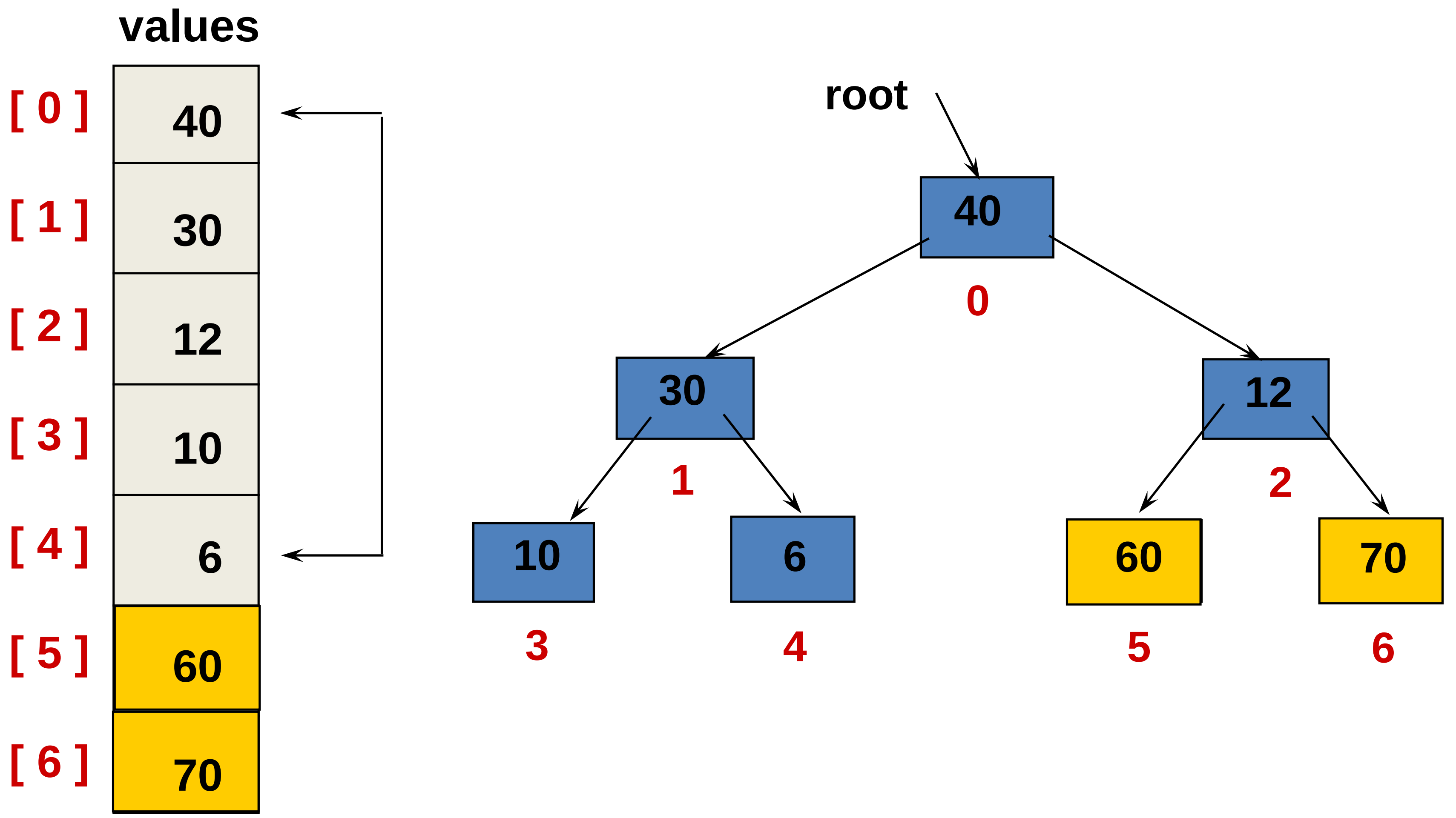
After swapping root element into its place



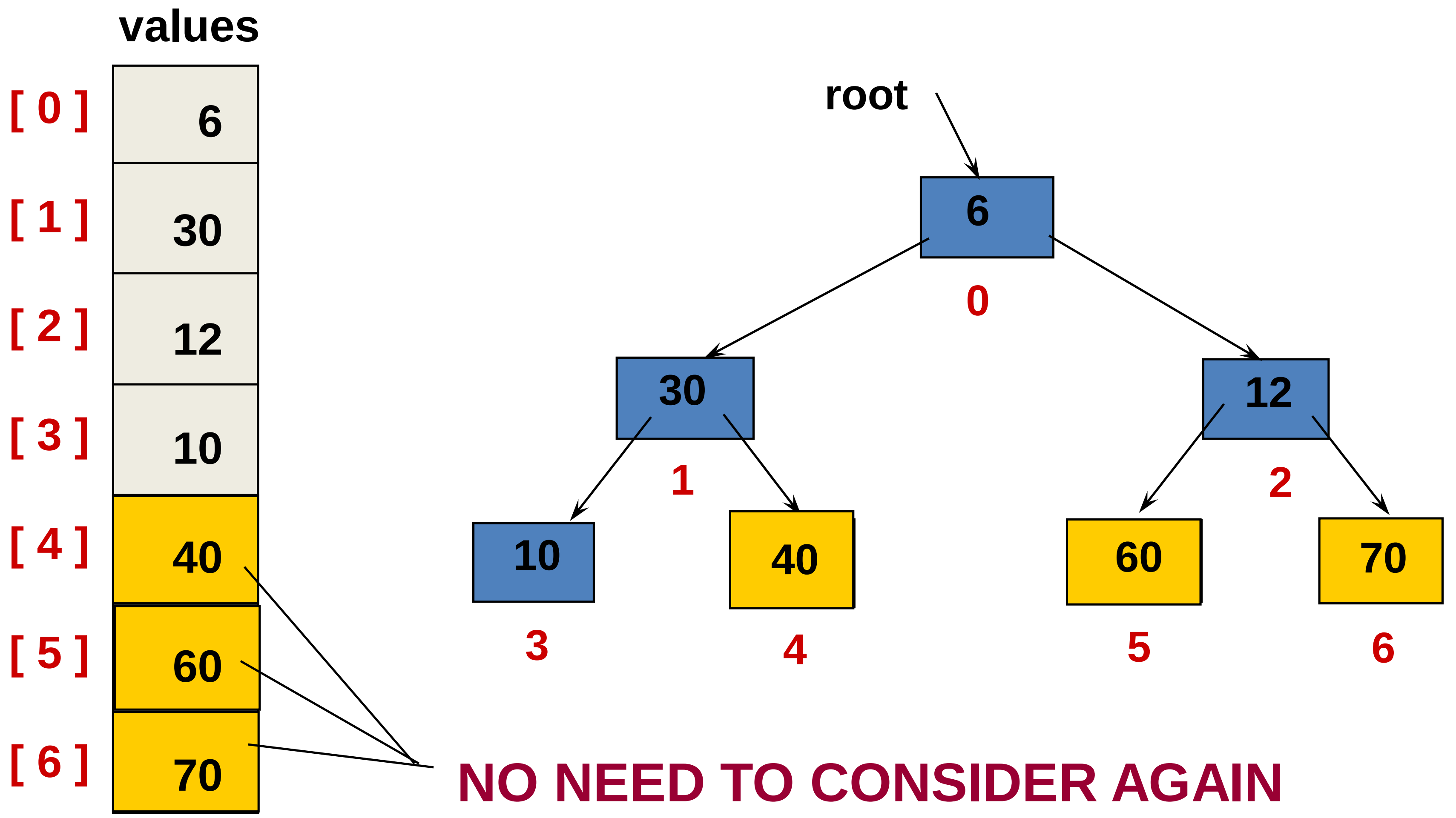
After reheaping remaining unsorted elements



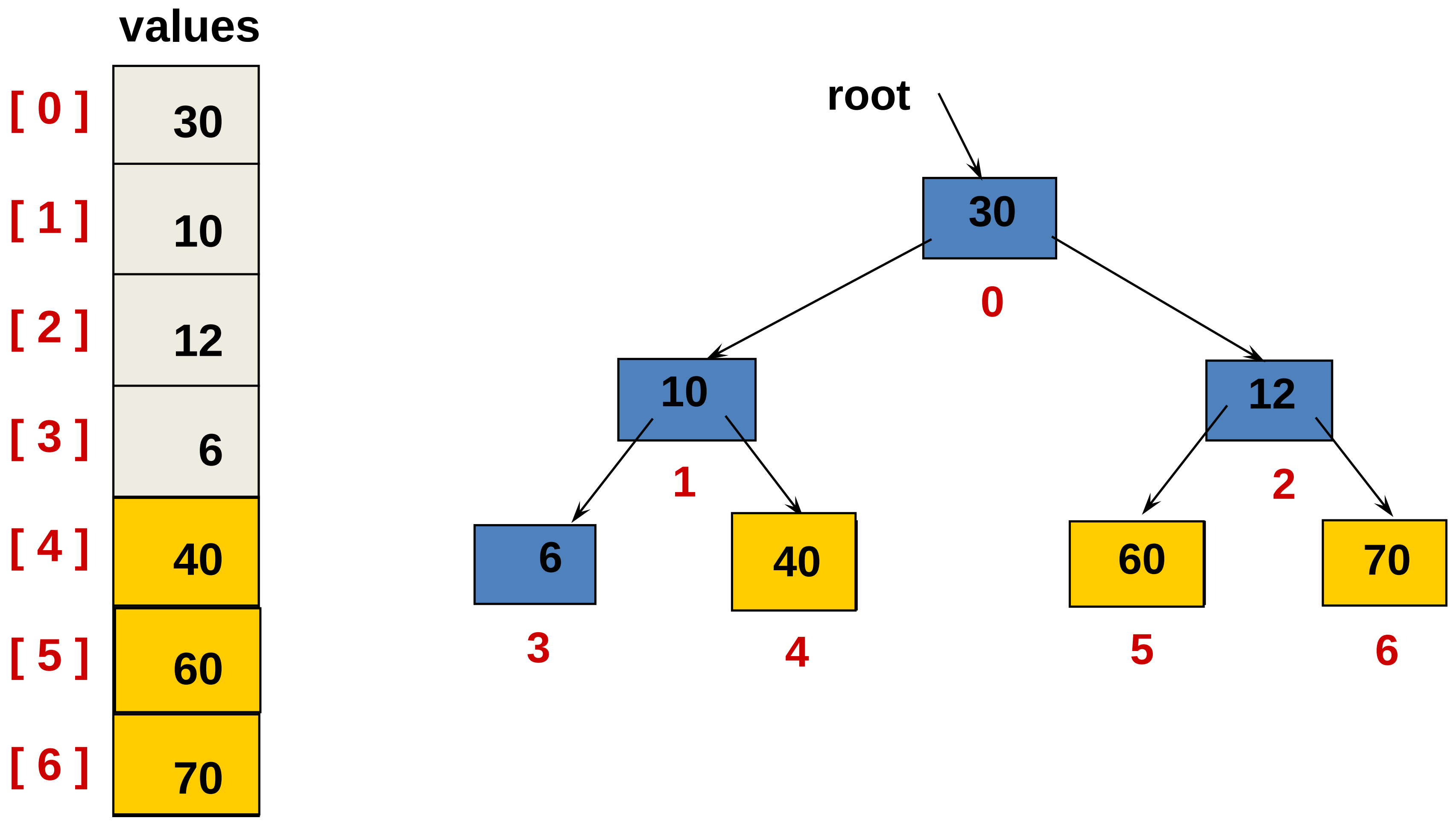
Swap root element into last place in unsorted array



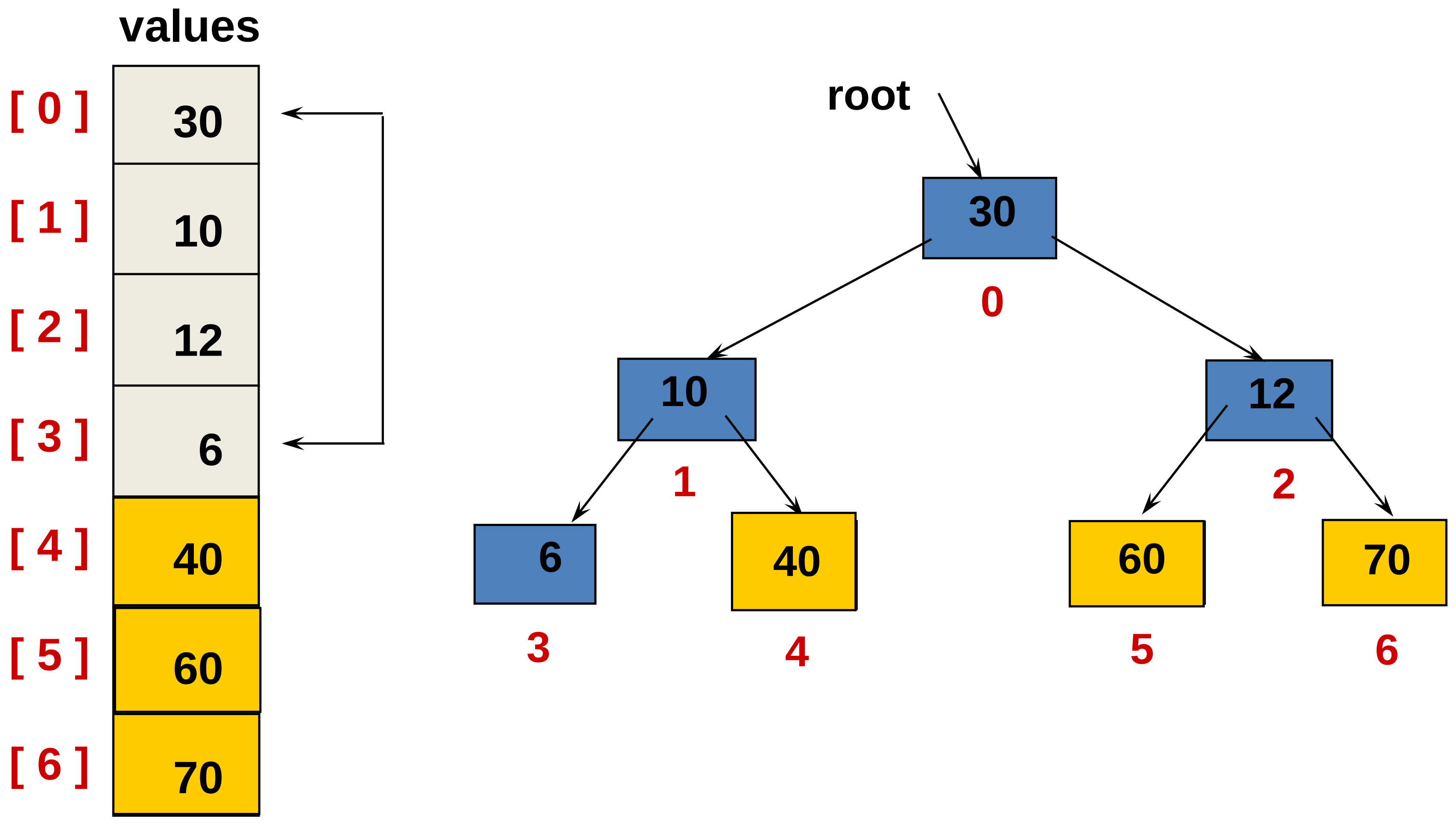
After swapping root element into its place



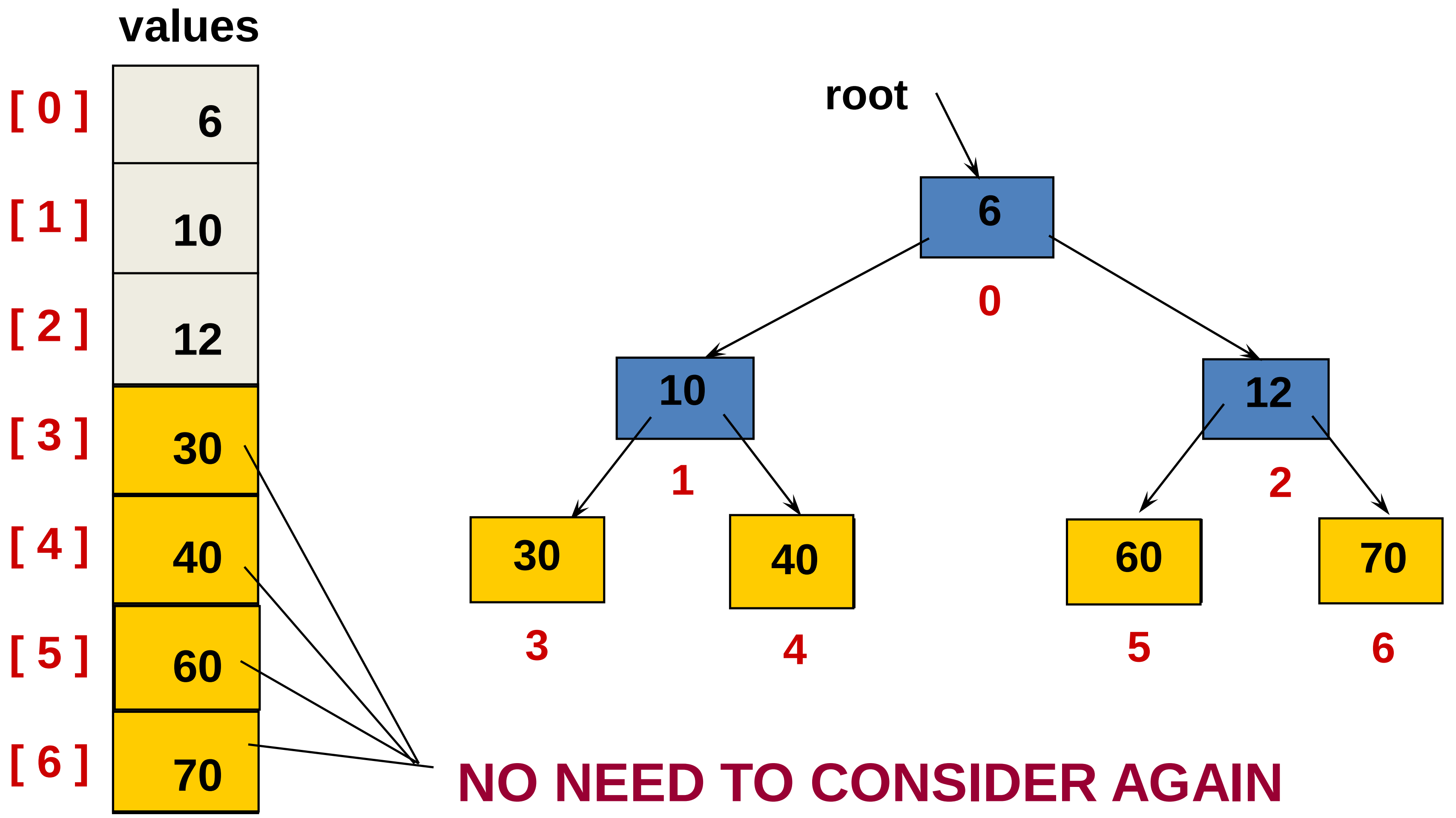
After reheaping remaining unsorted elements



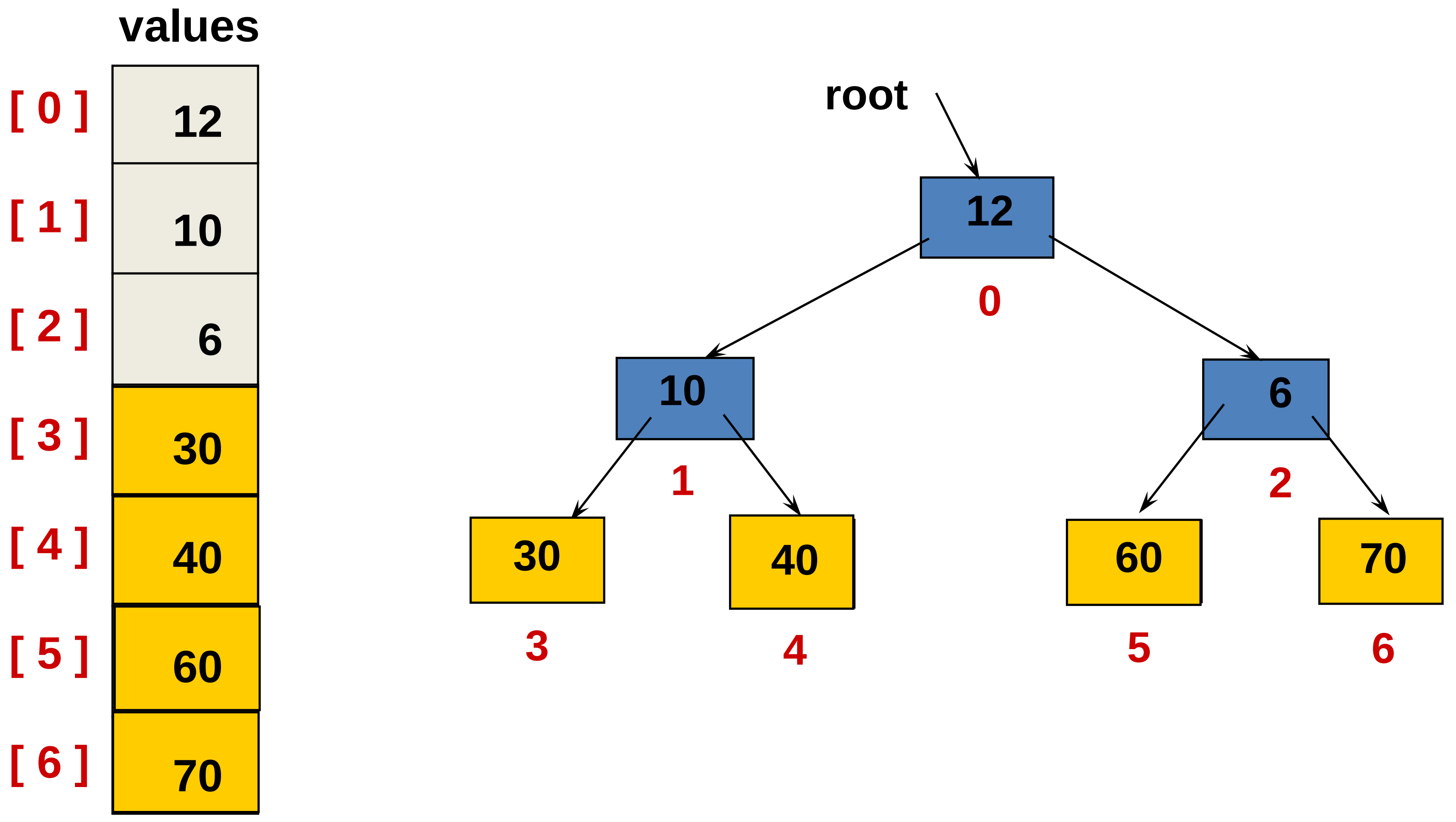
Swap root element into last place in unsorted array



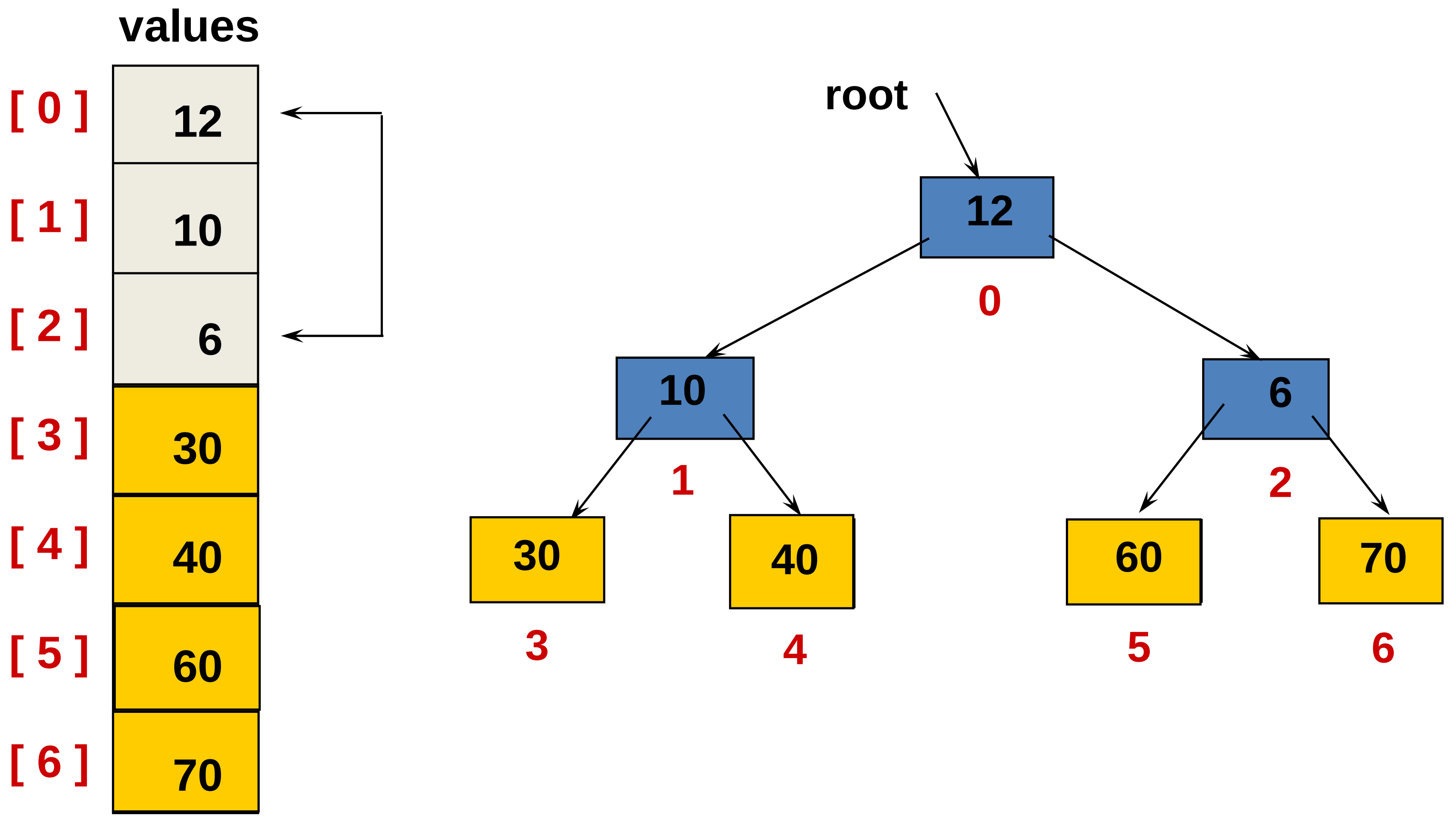
After swapping root element into its place



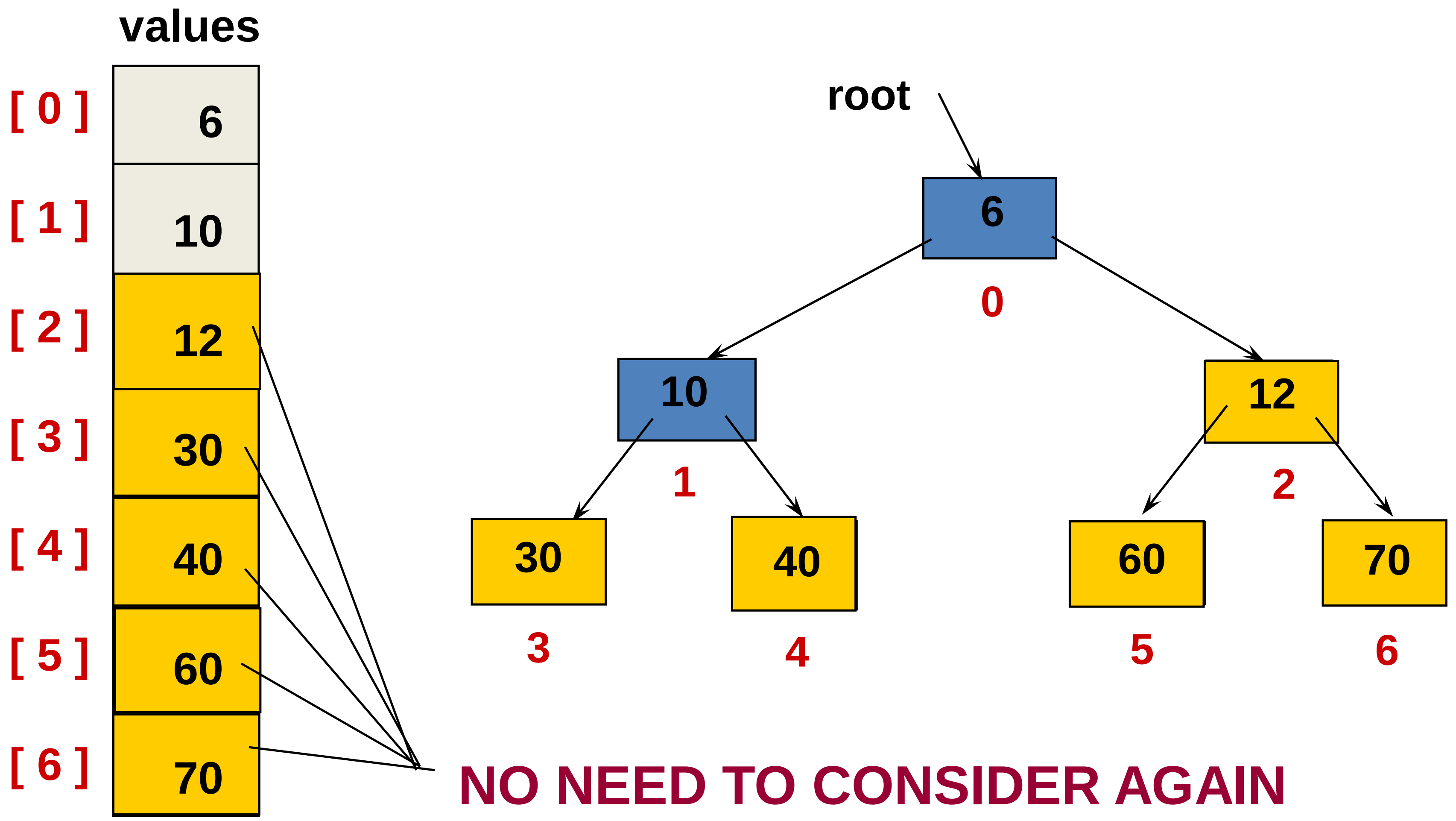
After reheaping remaining unsorted elements



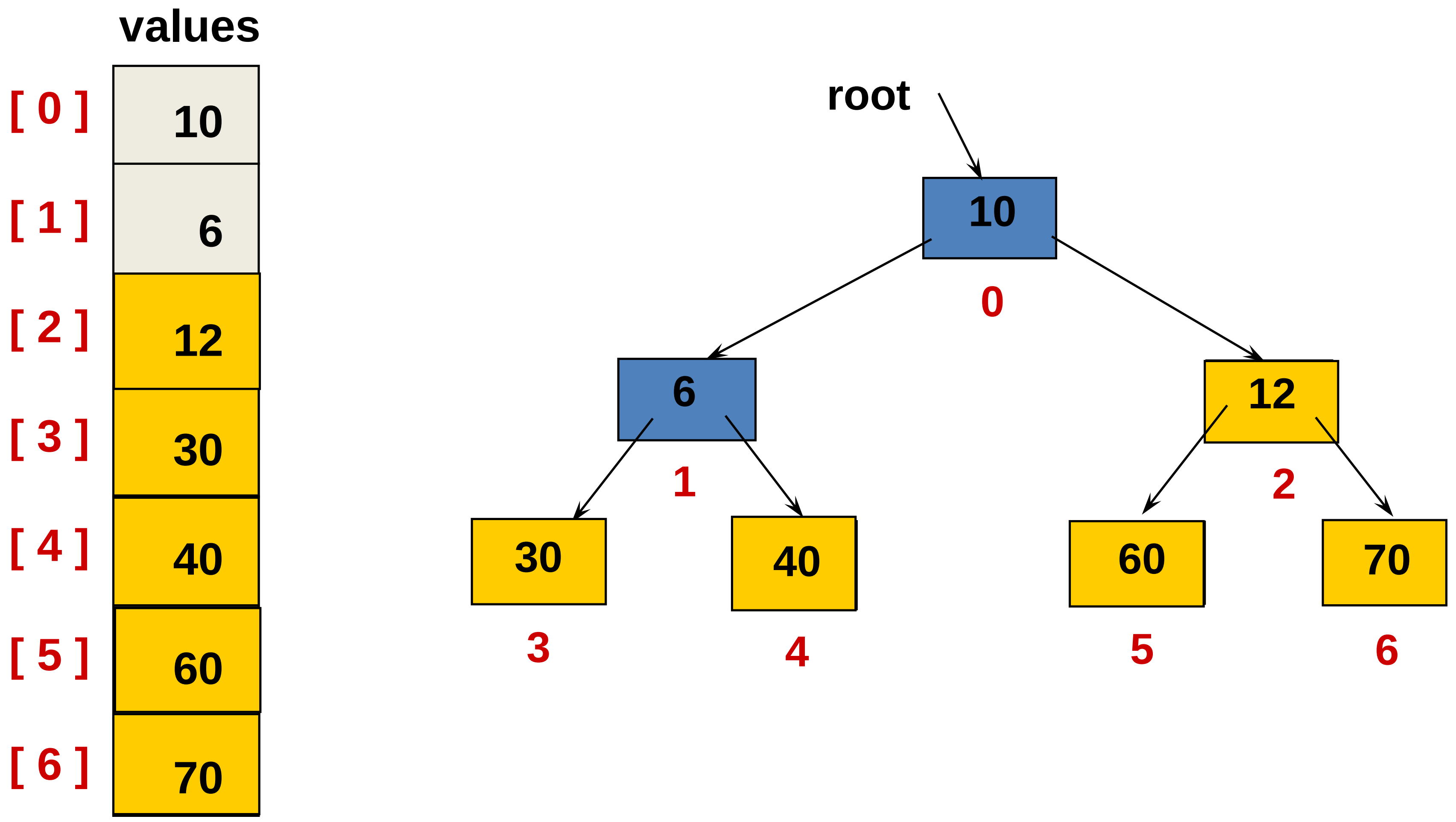
Swap root element into last place in unsorted array



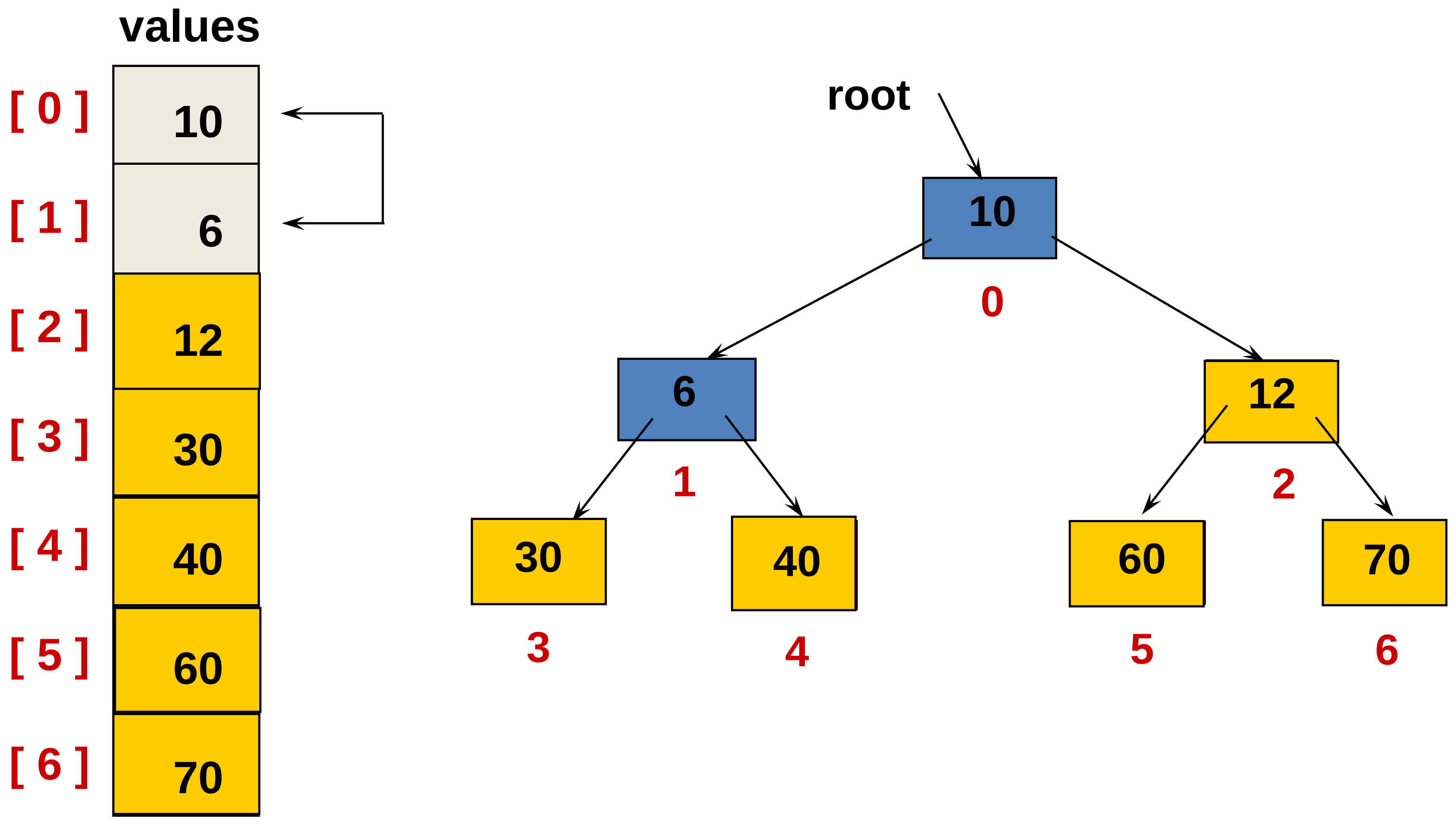
After swapping root element into its place



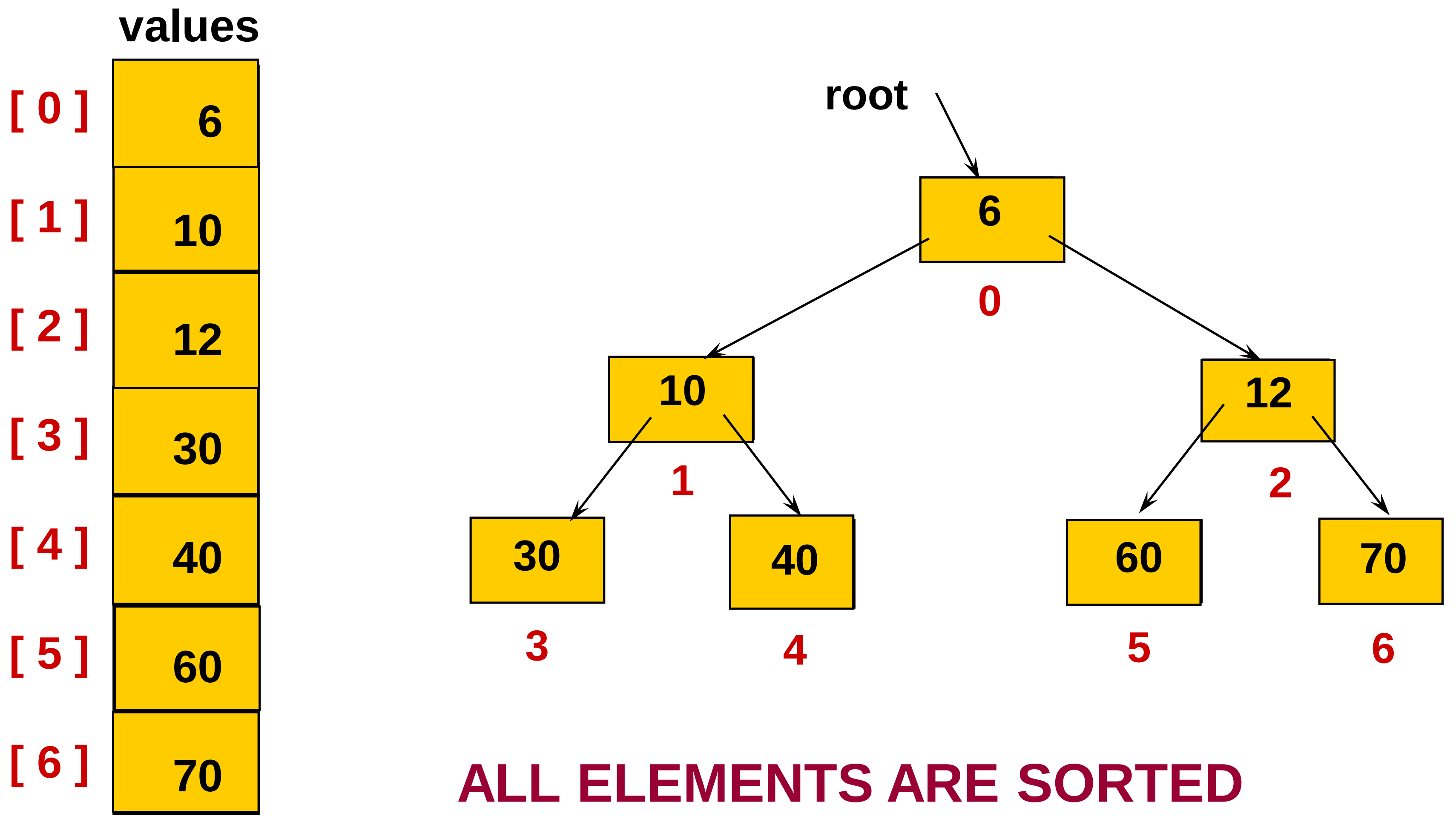
After reheaping remaining unsorted elements



Swap root element into last place in unsorted array



After swapping root element into its place



ALL ELEMENTS ARE SORTED

END.
WAIRAGU G.R.