C++ Plus Data Structures

Nell Dale

David Teague

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.)

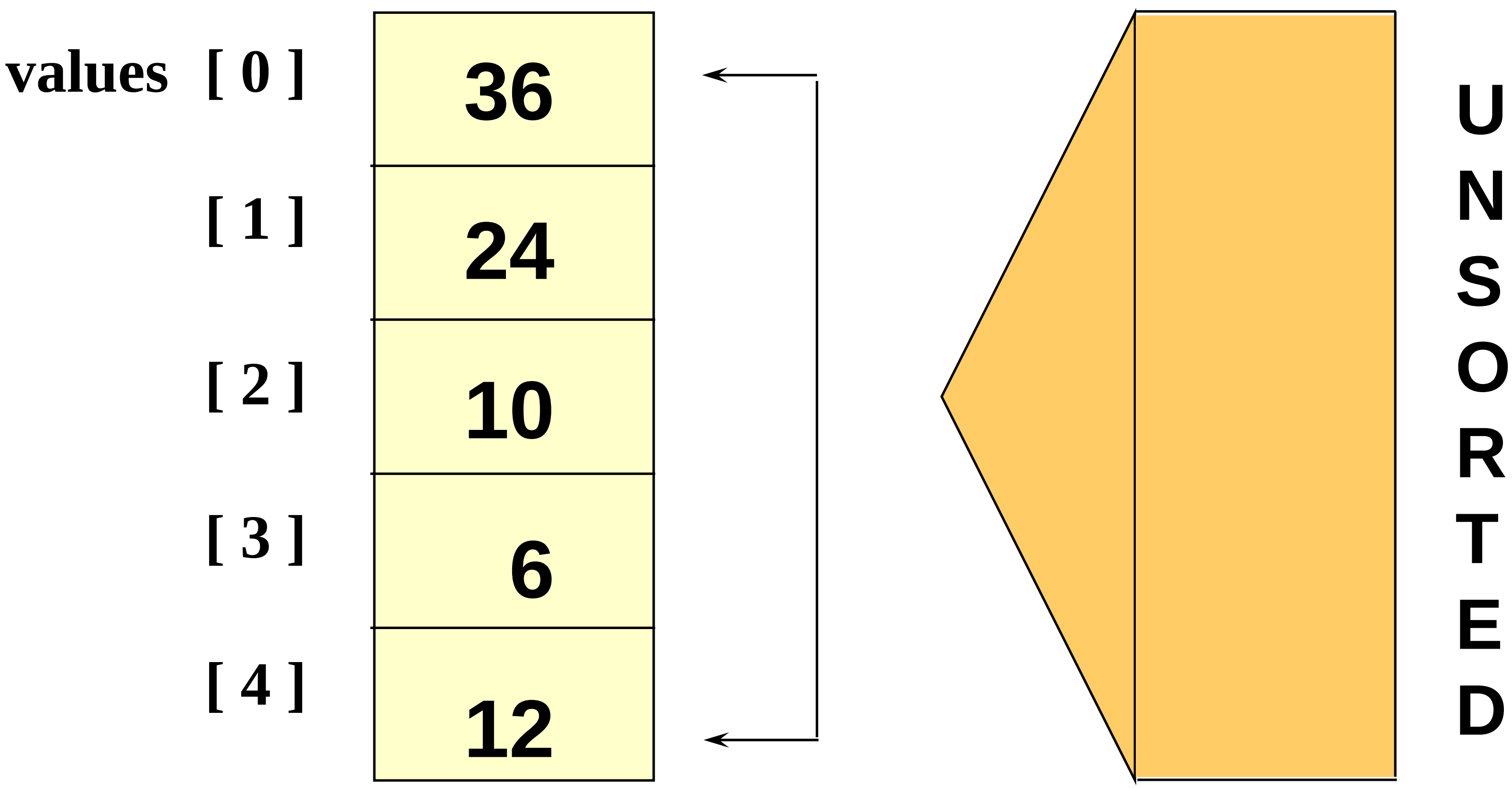
Straight Selection Sort

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

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

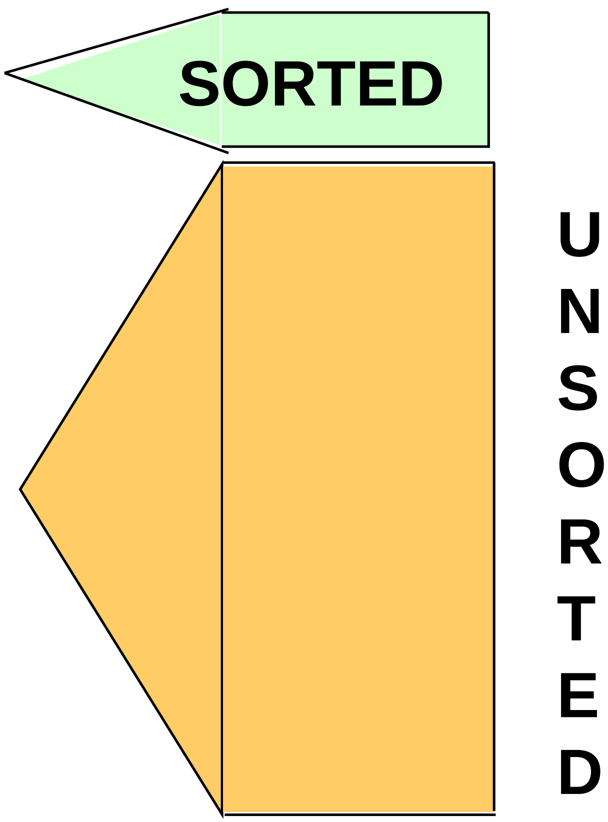
On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

Selection Sort: Pass One

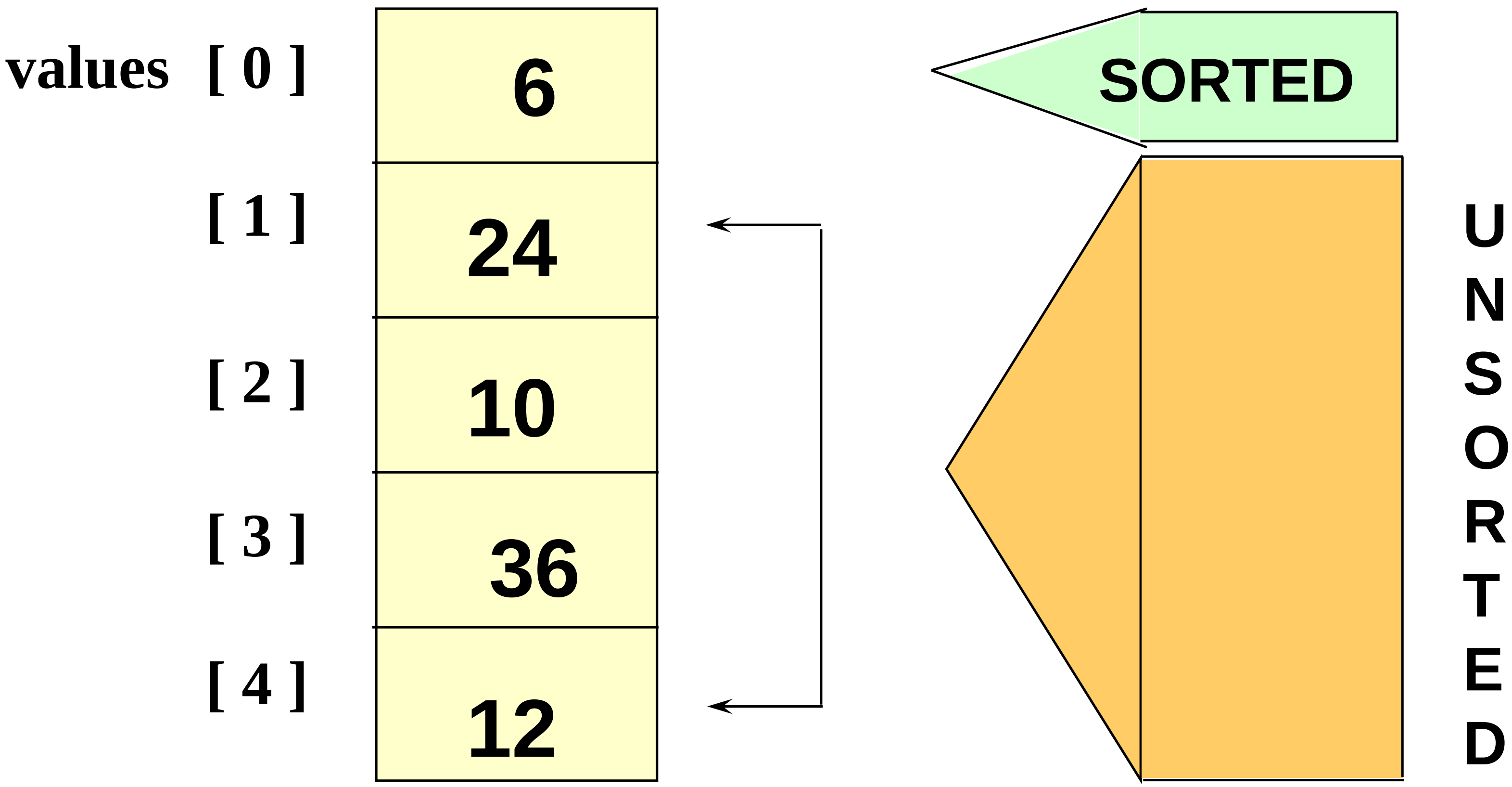


Selection Sort: End Pass One

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

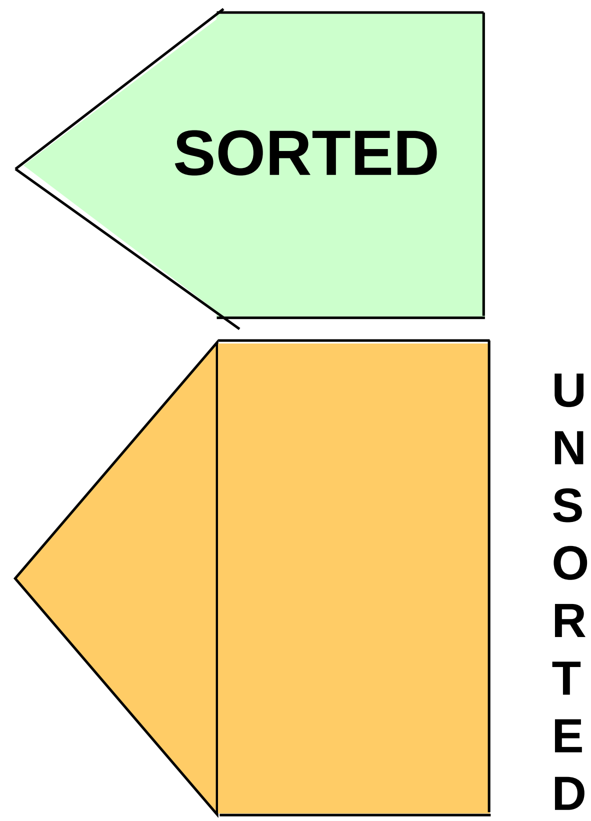


Selection Sort: Pass Two

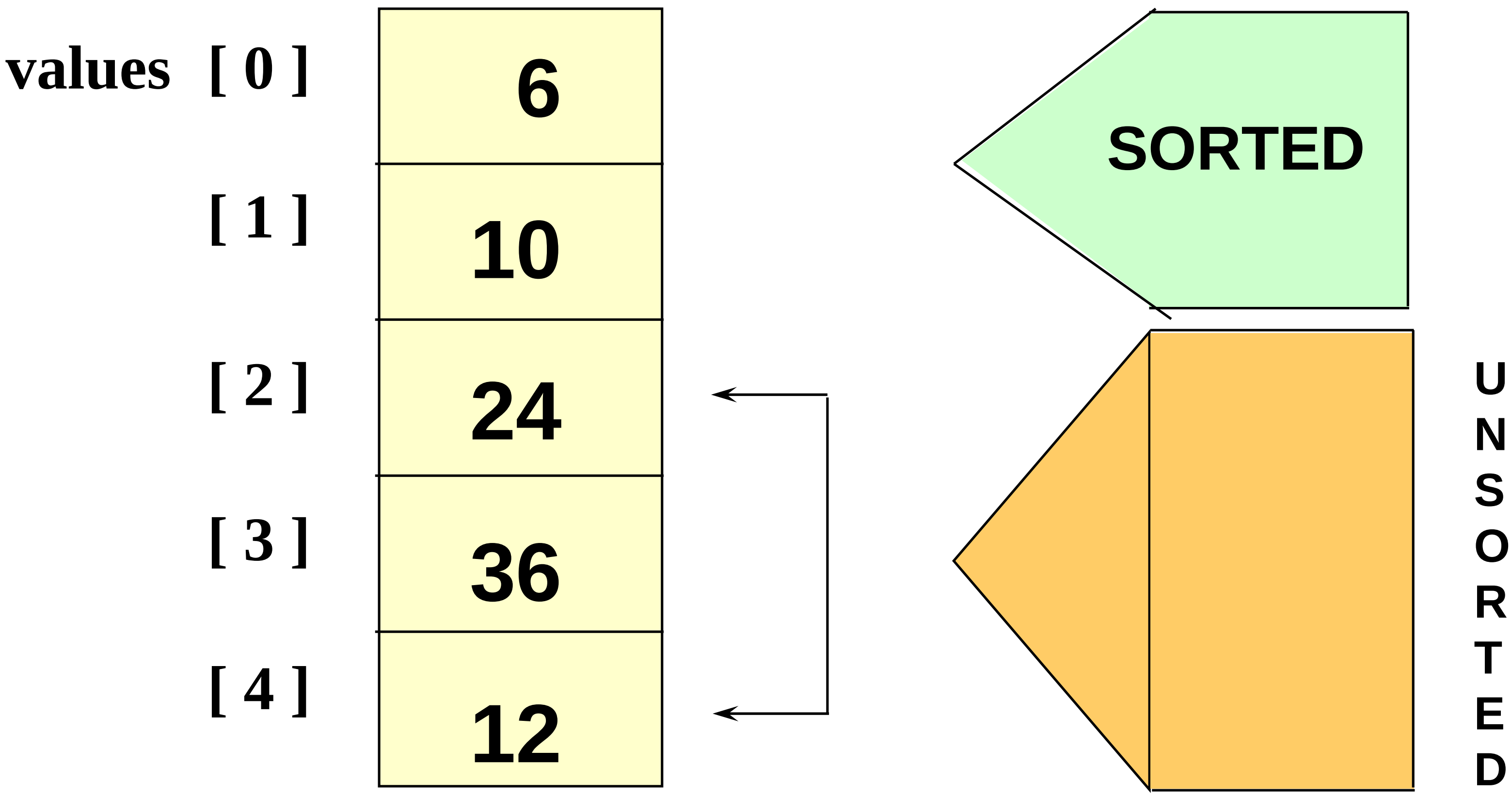


Selection Sort: End Pass Two

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

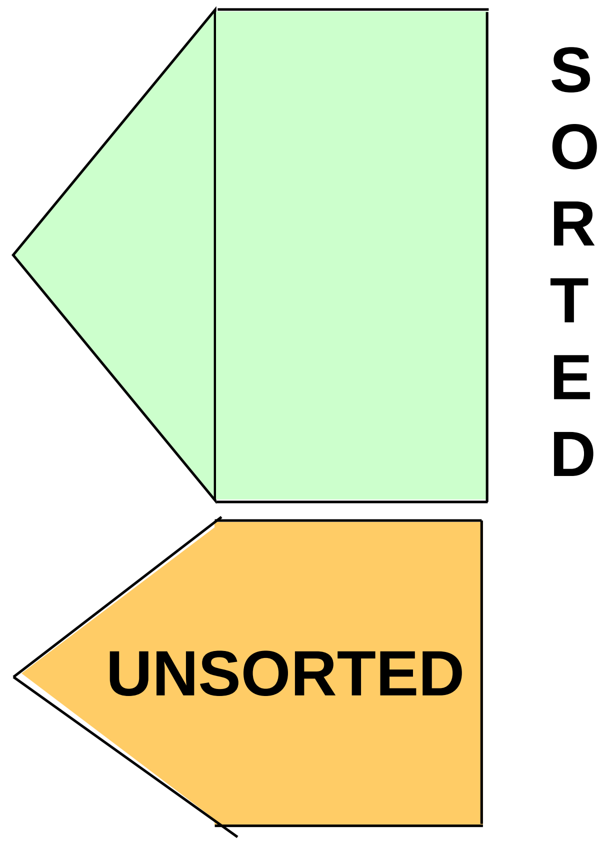


Selection Sort: Pass Three

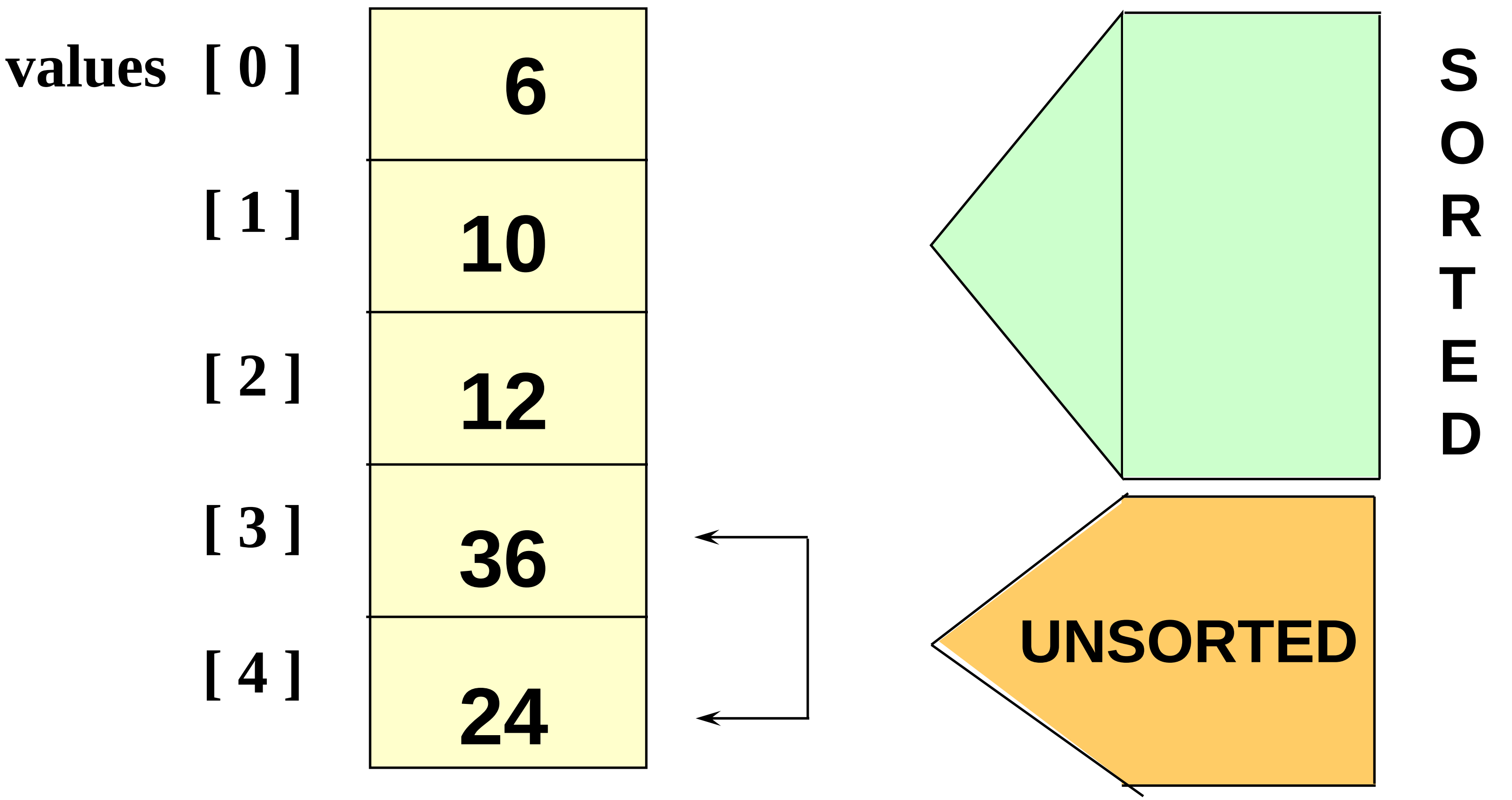


Selection Sort: End Pass Three

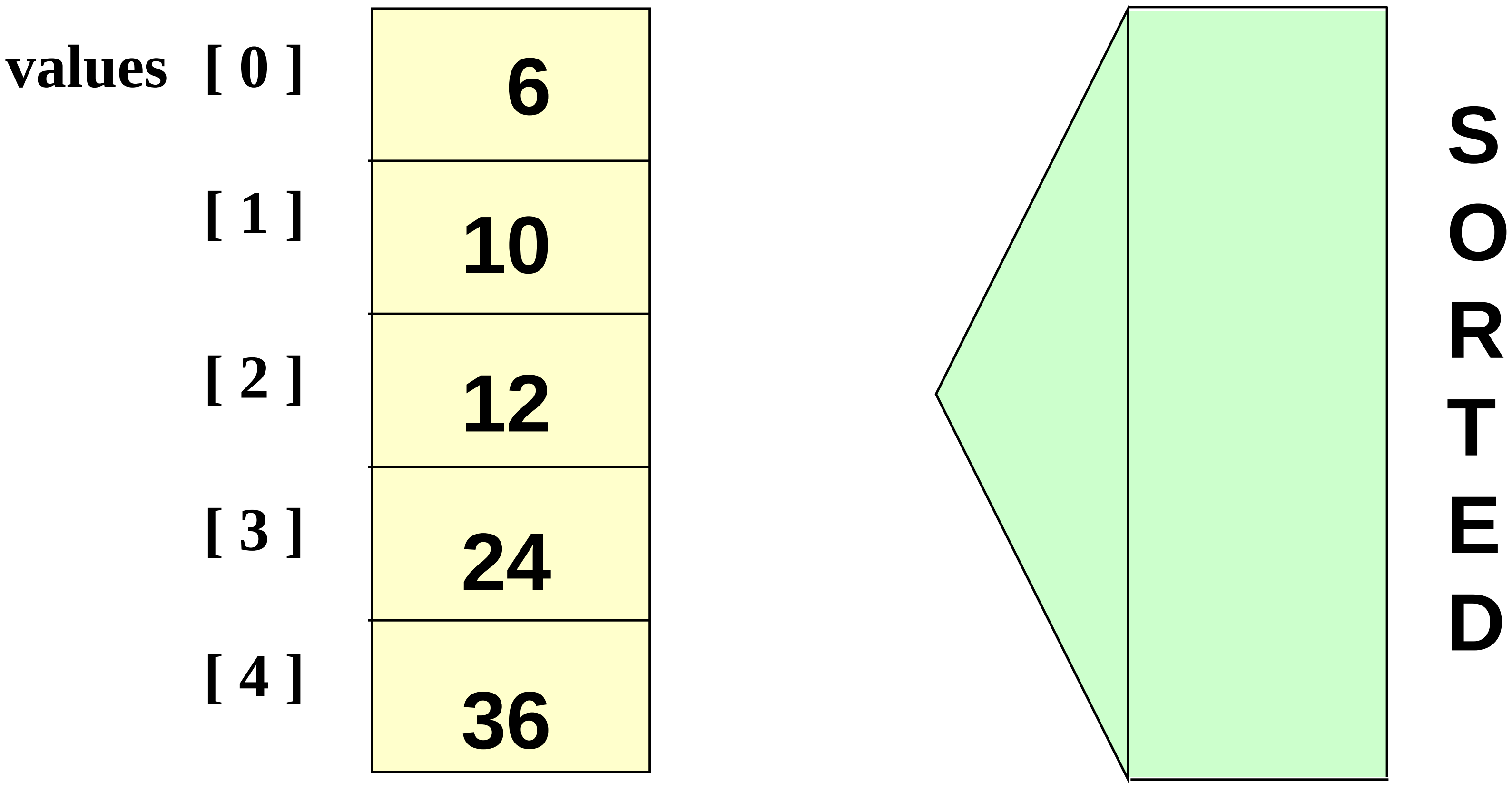
values	[0]	6
	[1]	10
	[2]	12
	[3]	36
	[4]	24



Selection Sort: Pass Four



Selection Sort: End Pass Four



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 . . .

$$
\begin{array}{rl}
& \text{Sum} = (N-1) + (N-2) + \;.\;.\;.\; + 2 + 1 \\
+ & \text{Sum} = 1 + 2 + \;.\;.\;.\; + (N-2) + (N-1) \\
\hline
& 2^* \text{Sum} = \underbrace{N + N + \;.\;.\;.\; + N + N} \\
& 2 * \text{Sum} = N * (N-1)
\end{array}
$$

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

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.

Bubble Sort

36	36	36	6	6	6	
24	24	6	36	36	10	
10	6	24	24	10	36	....
6	10	10	10	24	24	
12	12	12	12	12	12	

```
template <class  ItemType >
void BubbleUp  ( ItemType  values [ ] ,   int  start ,  int end )

//  Post: Neighboring elements that were out of order have been
//           swapped between values [start] and  values [end],
//           beginning at values [end].
{
    for  ( int  index = end ;  index > start ; index-- )

        if  (values [ index ] < values [ index - 1 ] )

            Swap ( values [ index ], values [ index - 1 ] ) ;

}
```

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

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

    while  ( current < numValues - 1 )
    {

           BubbleUp ( values , current , numValues - 1 ) ;

           current++ ;
    }

}
```

Modified Bubble Sort

```
template <class  ItemType >
int BubbleUp  ( ItemType  values [ ] ,   int  start ,  int end )

{
      int flag=0;

      for  ( int  index = end ;  index > start ; index-- )
            if  (values [ index ] < values [ index - 1 ] )
            {
                     Swap ( values [ index ], values [ index - 1 ] ) ;
                     flag++;
            }
      return flag;
}

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

{
      int  current = 0  ;
      while  ( (current < numValues – 1) && BubbleUp ( values , current , numValues - 1 ) )
      {

               current++ ;
      }

}
```

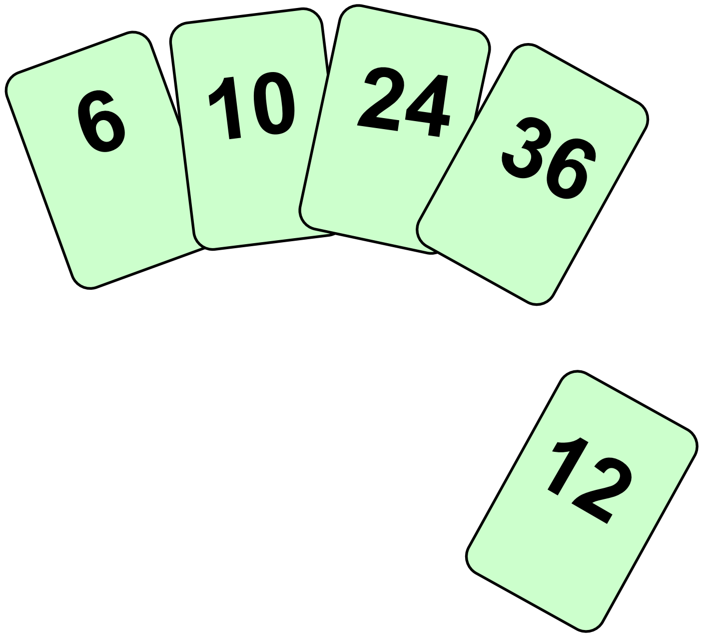
Insertion Sort

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.

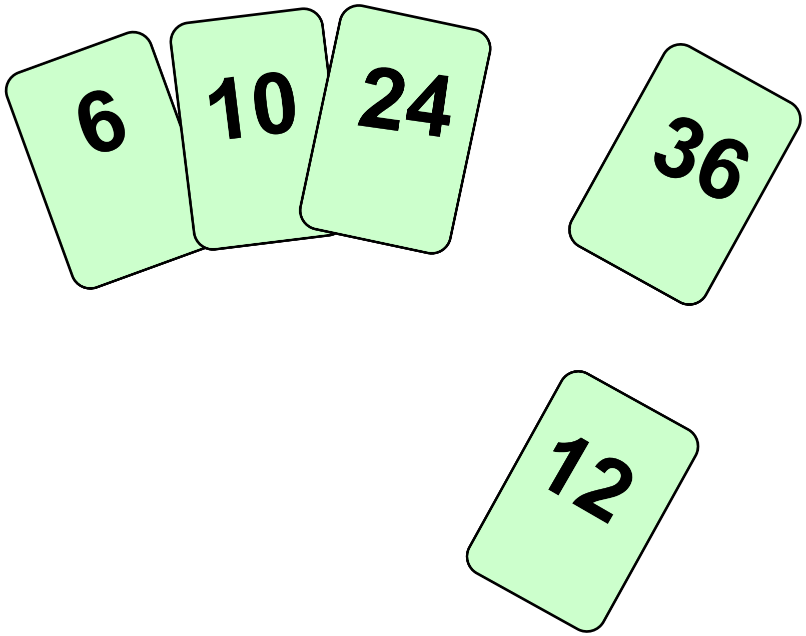
Insertion Sort



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.

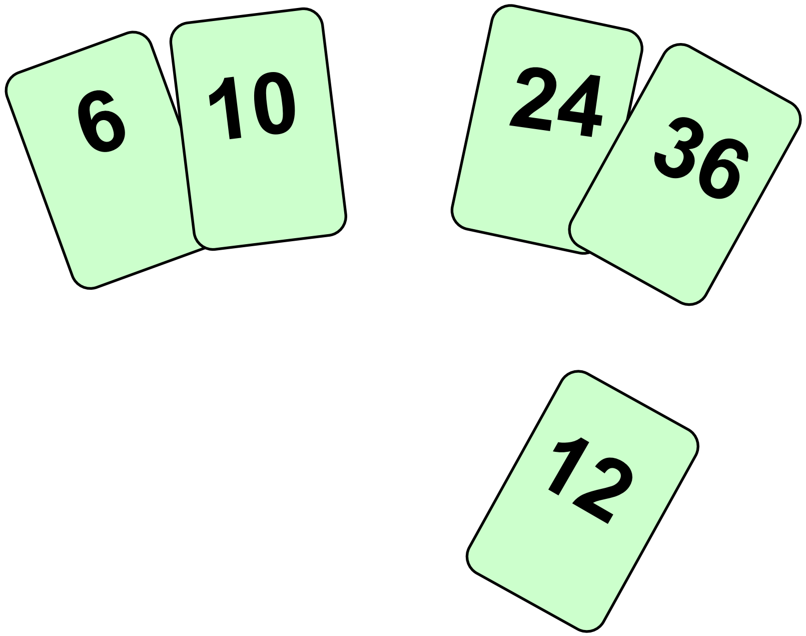
Insertion Sort



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.

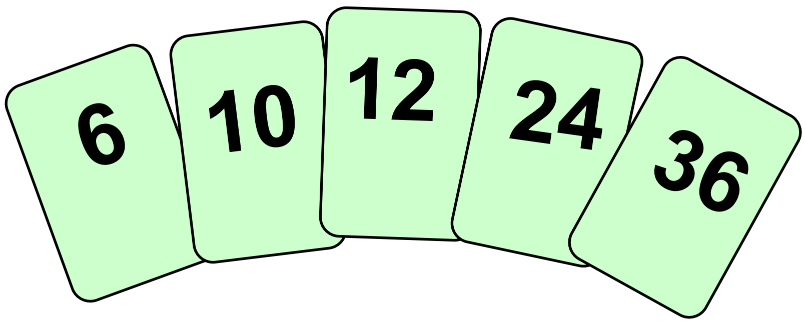
Insertion Sort



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.

Insertion Sort

end =	0	1		2		
values [0]	36	36	24	24	24	10
[1]	24	24	36	36	10	24
[2]	10	10	10	10	36	36
[3]	6	6	6	6	6	6
[4]	12	12	12	12	12	12

Insertion Sort

	end = 3				4		
[0]	10	10	10	6	6	6	6
[1]	24	24	6	10	10	10	10
[2]	36	6	24	24	24	24	12
[3]	6	36	36	36	36	12	24
[4]	12	12	12	12	12	36	36

```
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 = 1 ; count < numValues ; count++ )

         InsertItem ( values , 0 , count ) ;

}
```

Insertion Sort

- **Advantages:**
  - **Relatively easy to implement**
  - **Fast when:**
    - ✓ **the data is almost sorted**
    - ✓ **the size of the data is small.**
- **Disadvantages:**
  - **If the data is already sorted in reverse order, each insertion requires a large number of comparisons.**

Shell Sort

- Essentially a segmented insertion sort
  - Divides an array into several smaller non-contiguous segments
  - The distance between successive elements in one segment is called a “gap”.
  - Each segment is sorted within itself using insertion sort.
  - Then resegment into larger segments (smaller gaps) and repeat sort.
  - Continue until only one segment (gap = 1)

Shell Sort

- It works because:
  - It always deals with a small number of elements.
  - Elements are moved a long way through array with each swap and this leaves it more nearly sorted.

Shell Sort Example

Iteration 1

80	90	50	10	40	31	55	81	9

Gap = 8/2=4. If even number, make it odd by adding 1. Gag=5

80					31			
	90					55		
		50					81	
			10					9
				40				

31					80			
	55					90		
		50					81	
			9					10
				40				

Shell Sort Example

Iteration 2

31	55	50	9	40	80	90	81	10

Gap = 5/2+1=3.

31			9			90		
	55			40			81	
		50			80			10

9			31			90		
	40			55			81	
		10			50			80

Shell Sort Example

Iteration 3

9	40	10	31	55	50	90	81	80

Gap = 3/2=1.

9	40	10	31	55	50	90	81	80

9	10	31	40	50	55	80	81	90

Sorting Algorithms
and Average Case Number of Comparisons

Simple Sorts

- Straight Selection Sort
- Bubble Sort
- Insertion Sort
- Shell Sort

$O(N^2)$

More Complex Sorts

- Quick Sort
- Merge Sort
- Heap Sort

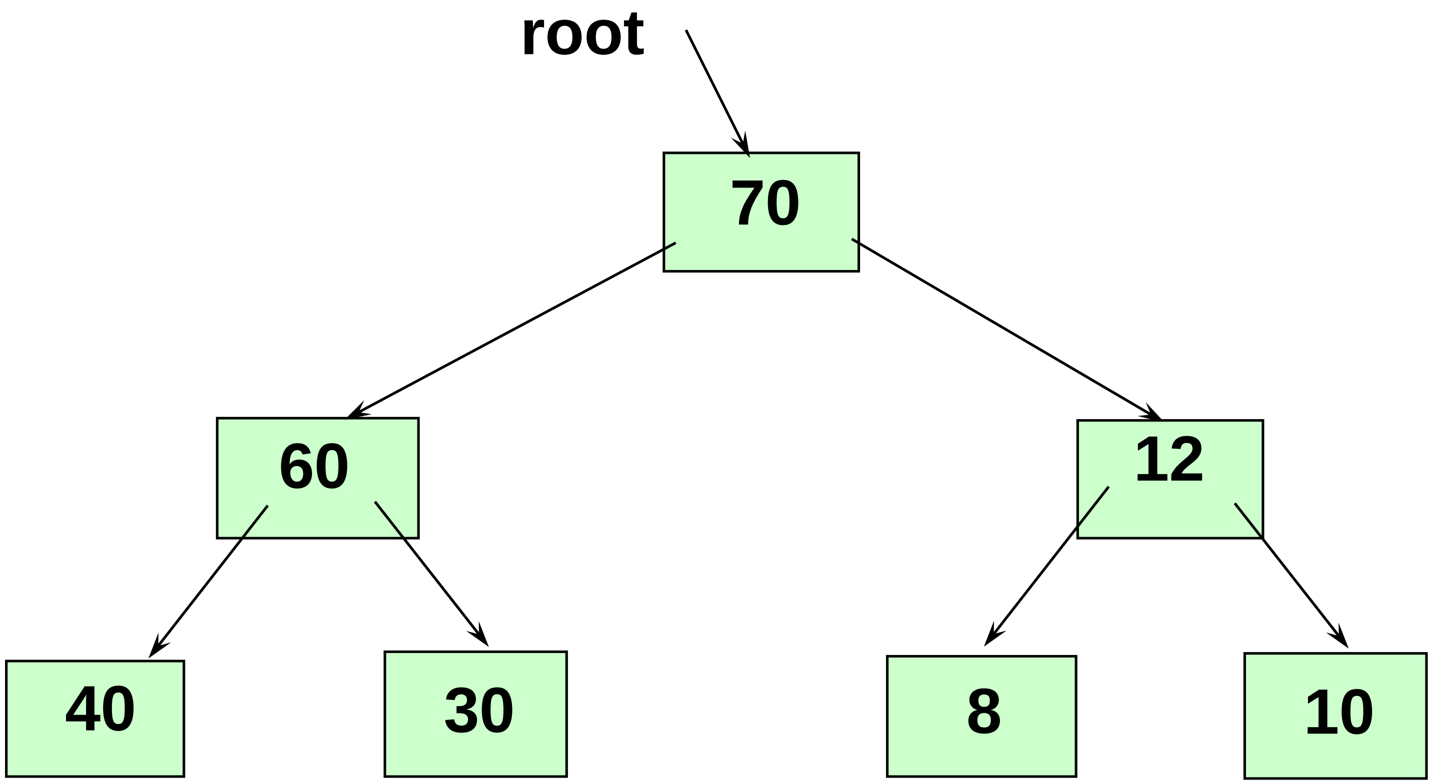
$O(N*\log N)$

Recall that . . .

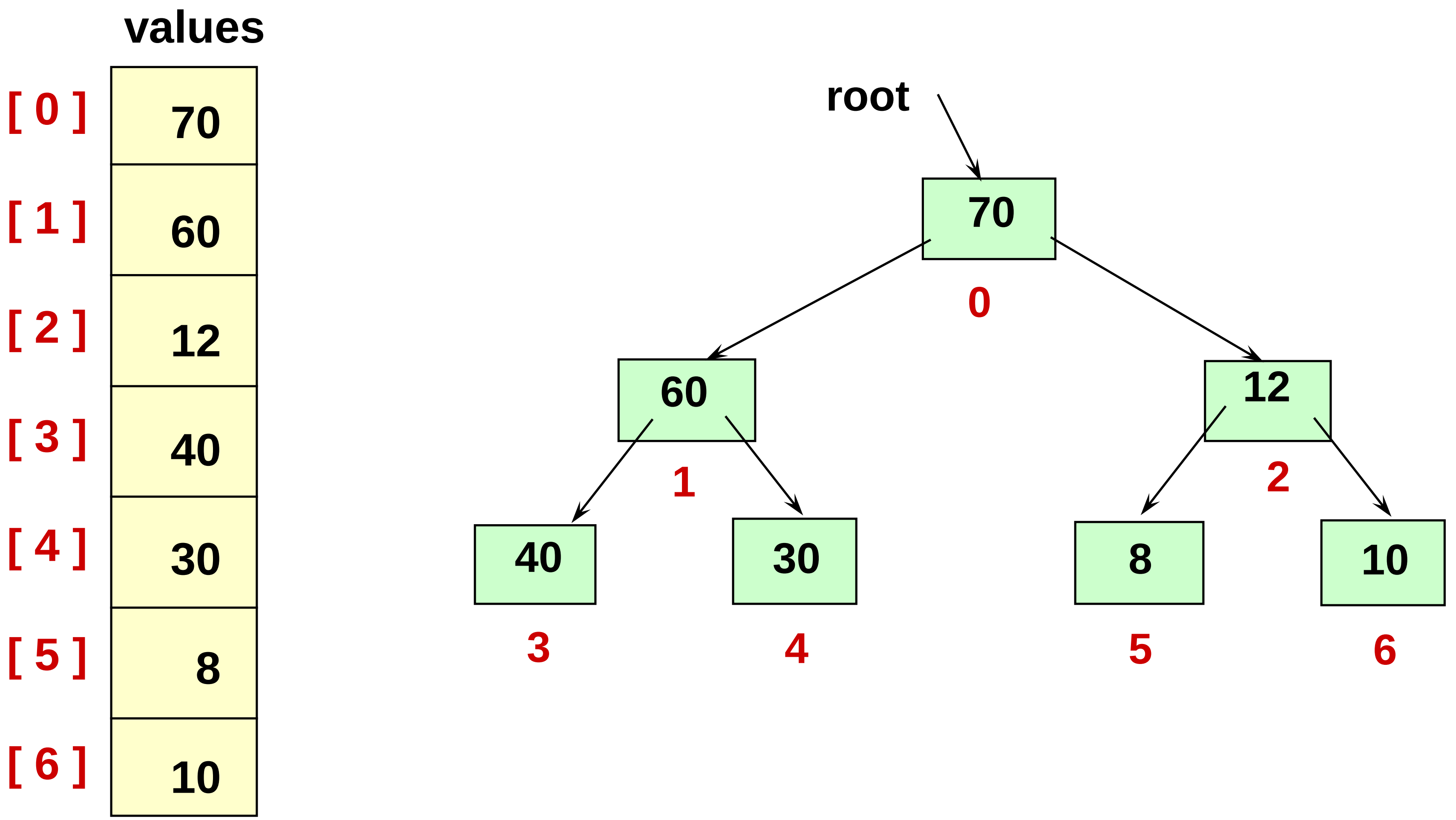
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- **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

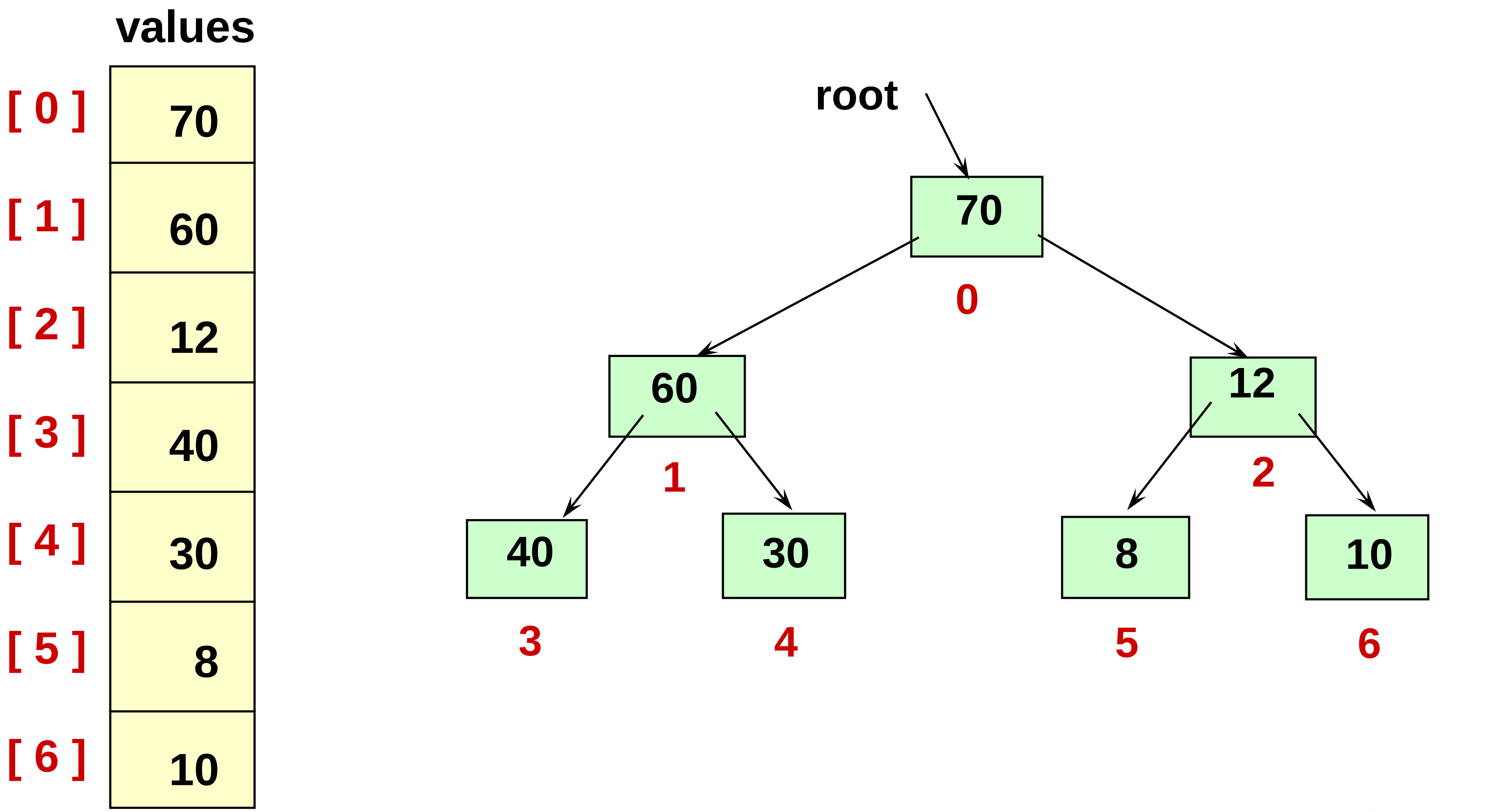


Heap Sort Approach

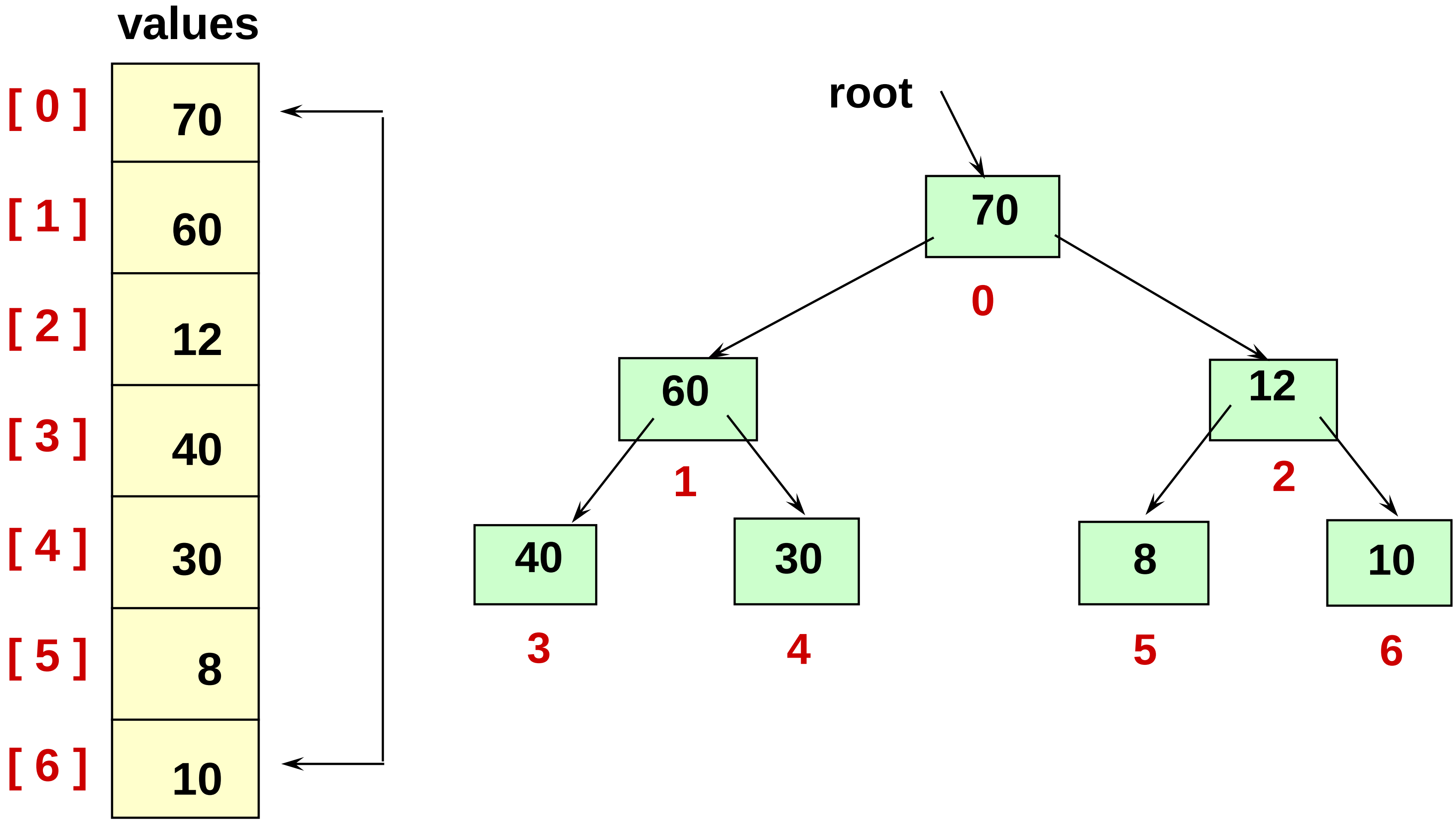
First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- 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).

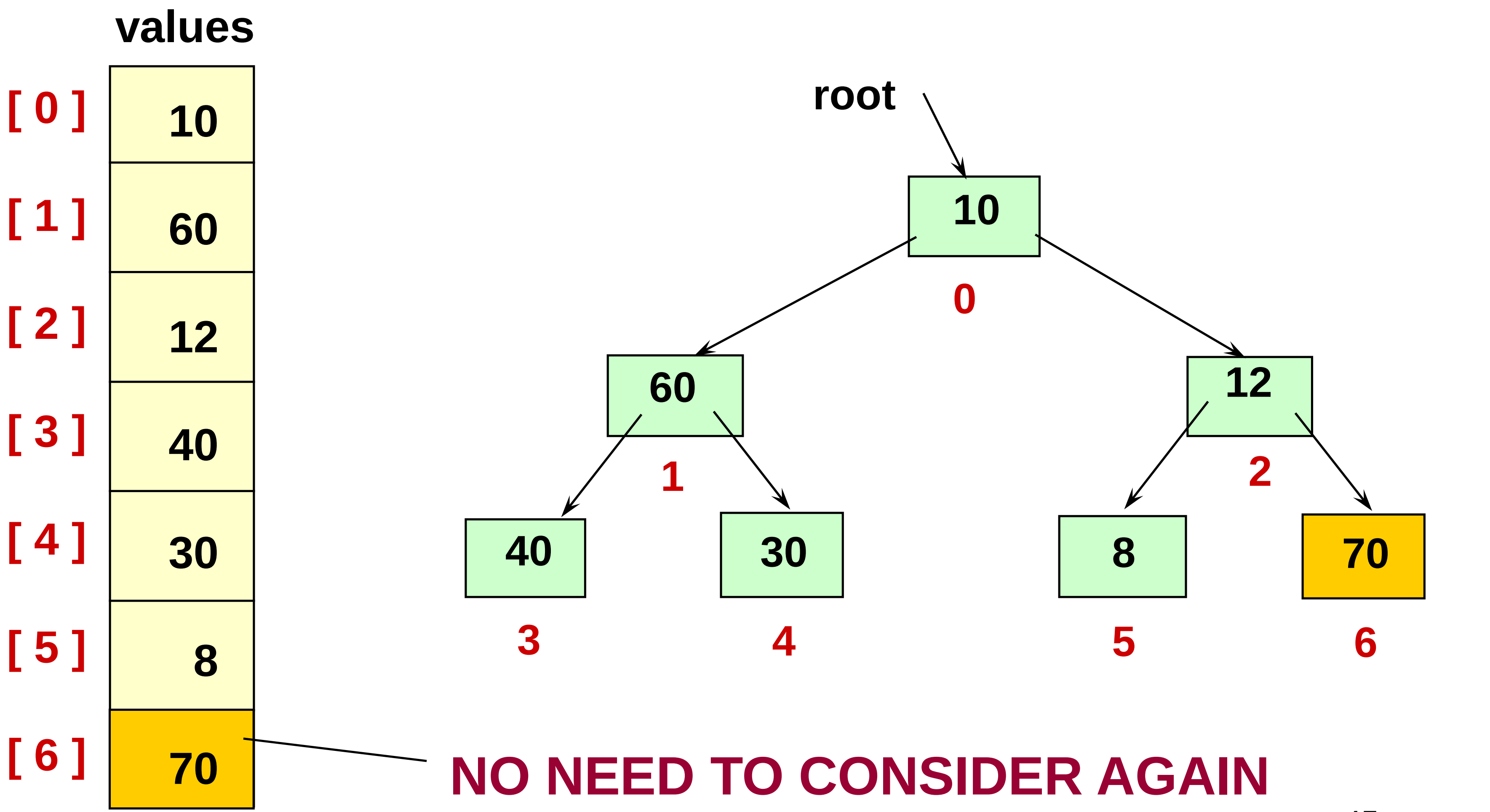
After creating the original heap



Swap root element into last place in unsorted array

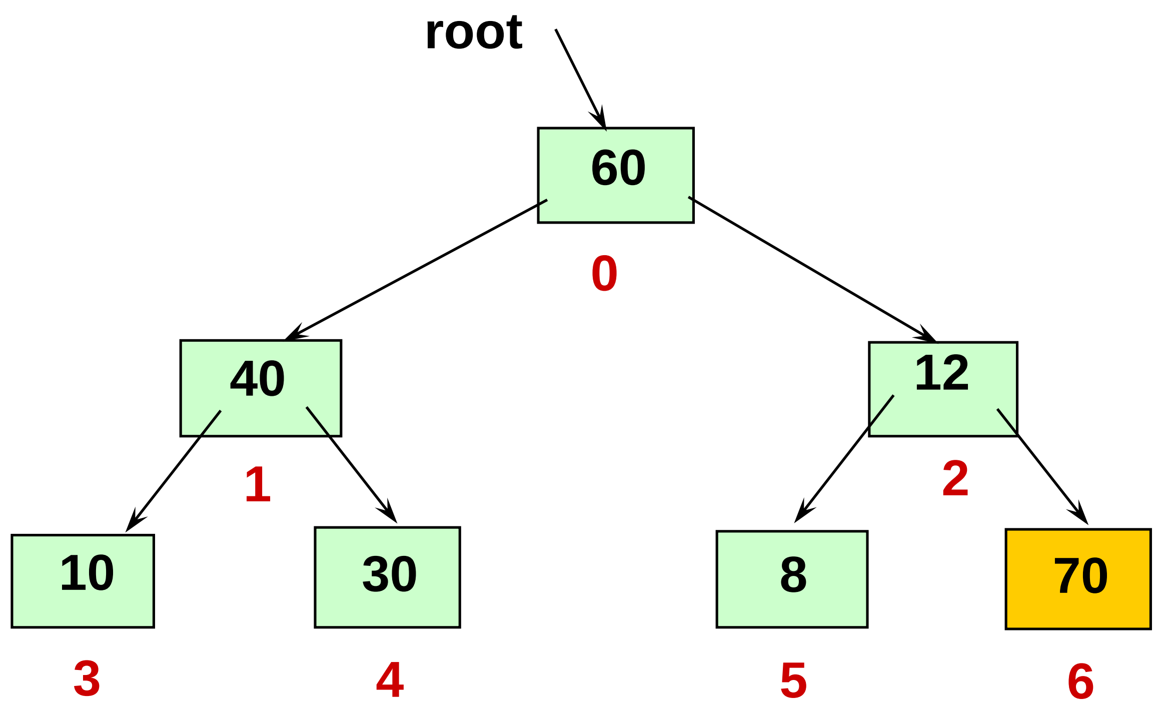


After swapping root element into its place

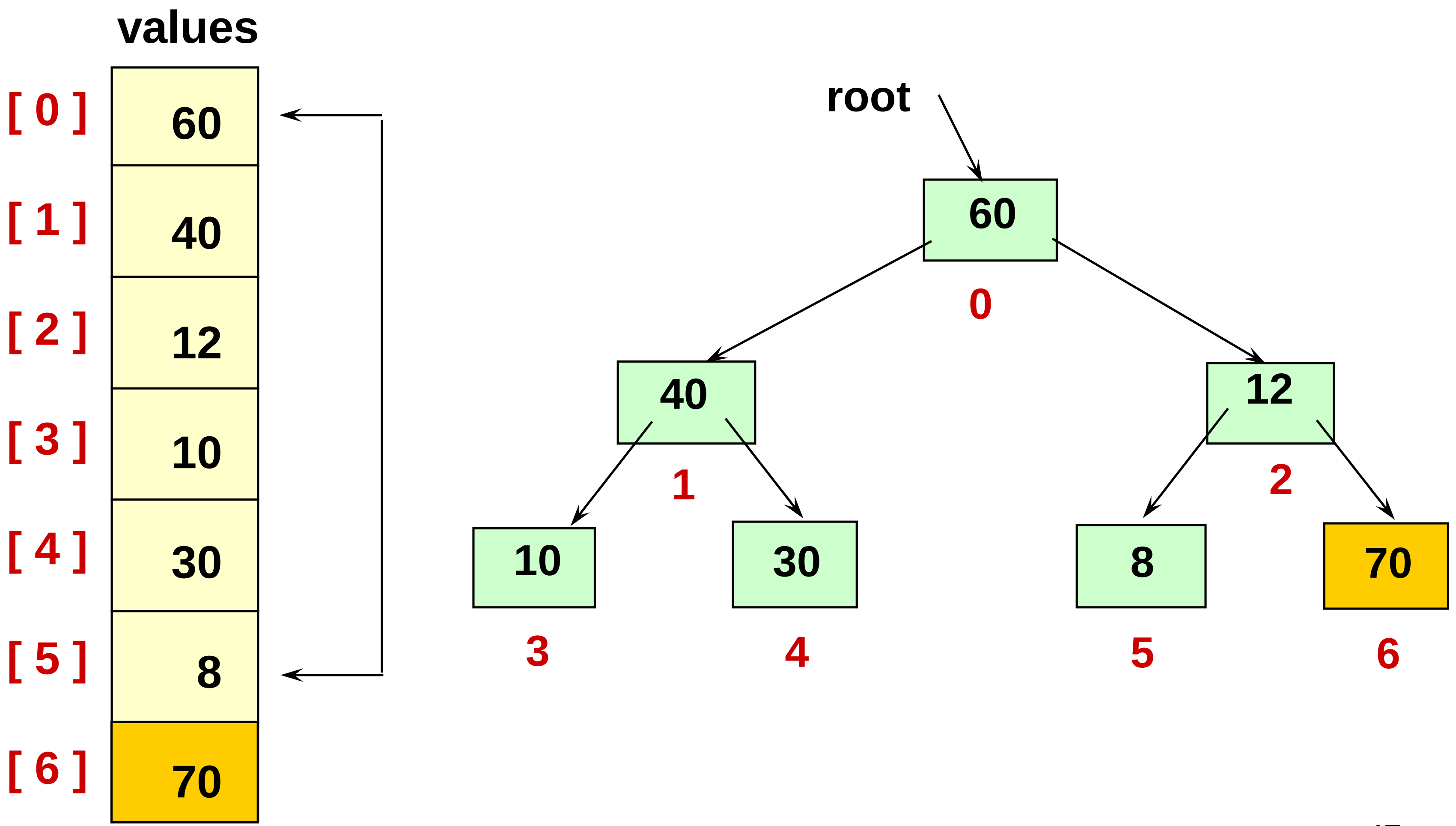


After reheaping remaining unsorted elements

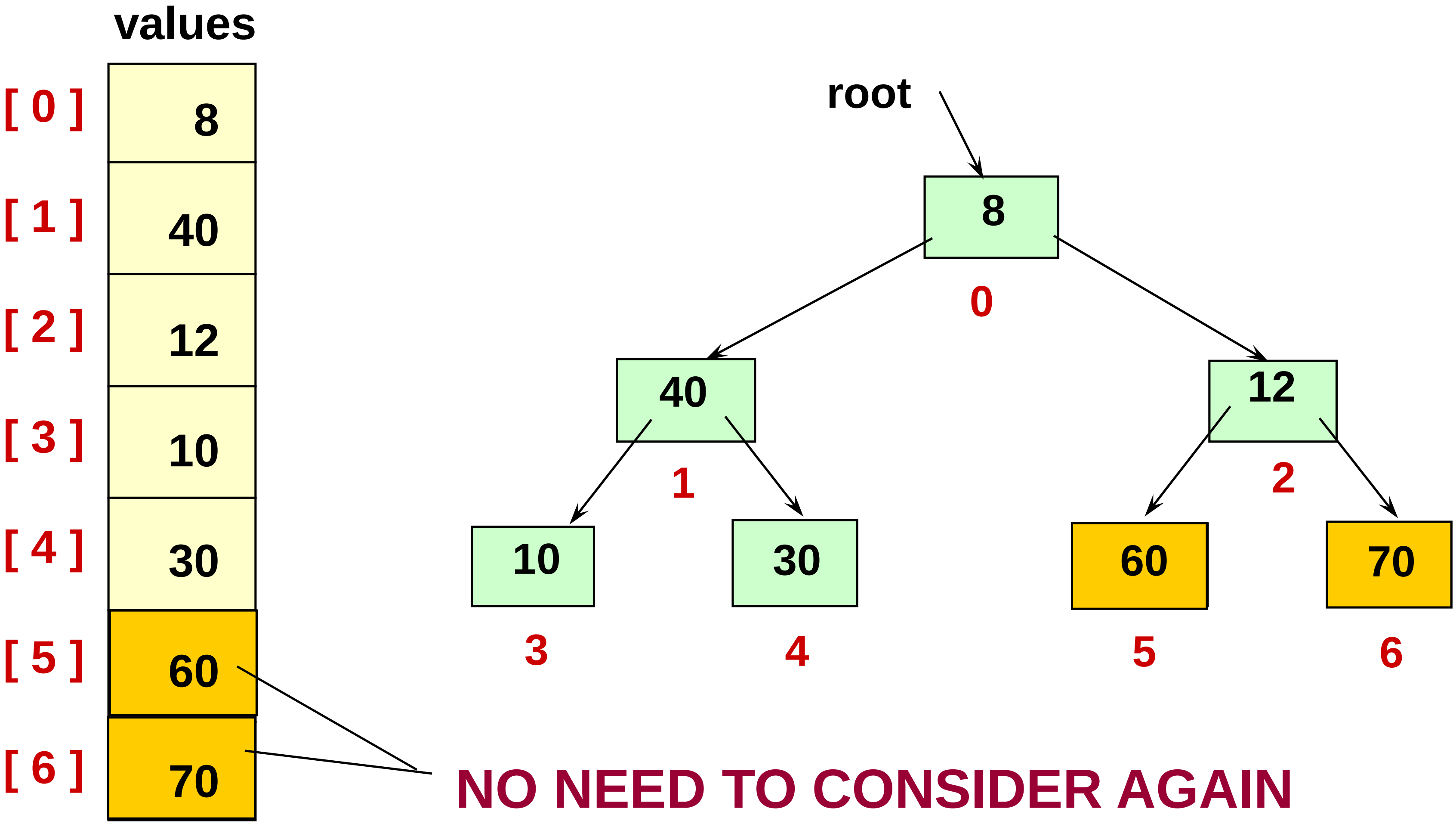
	values
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	8
[6]	70



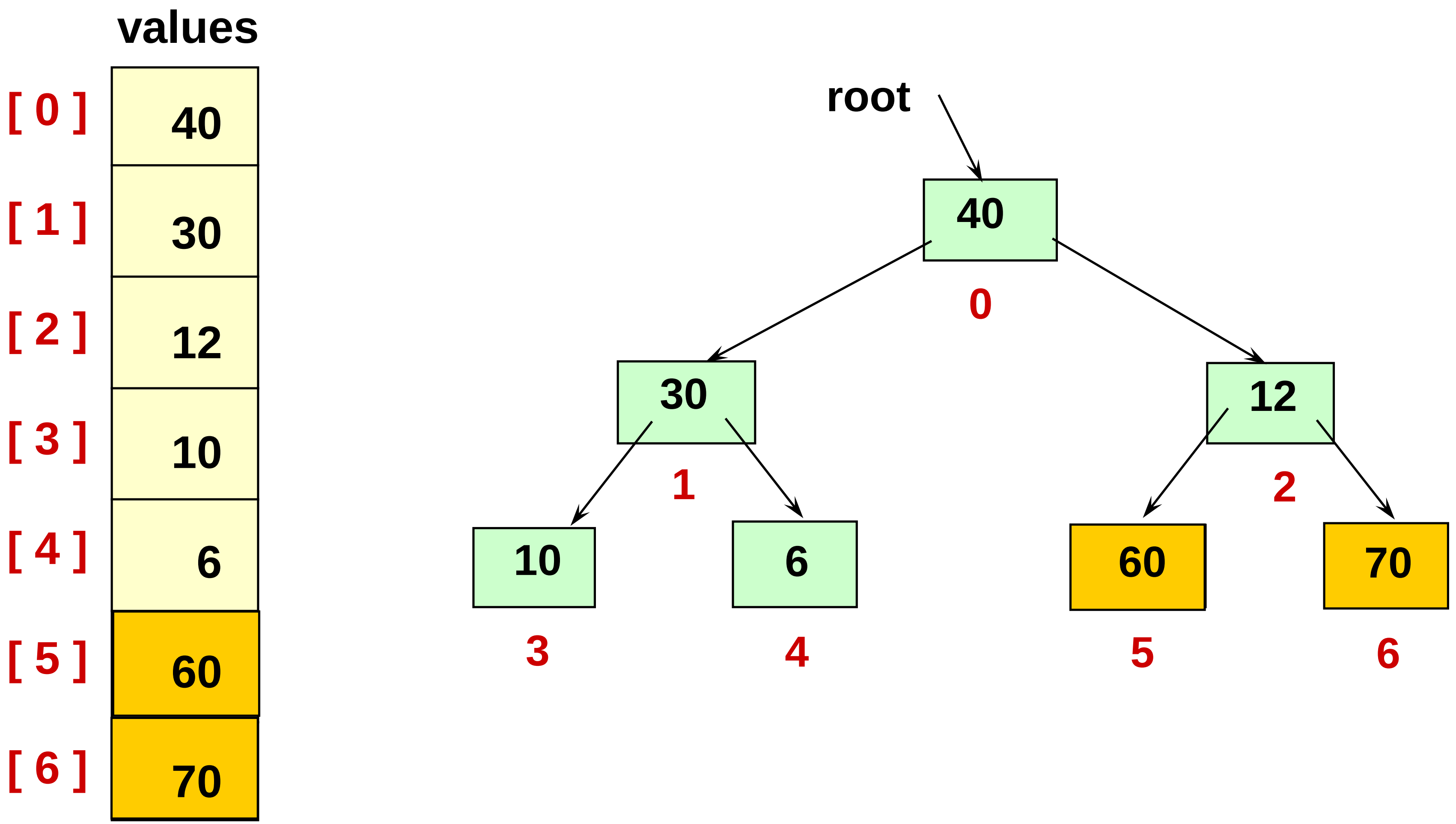
Swap root element into last place in unsorted array



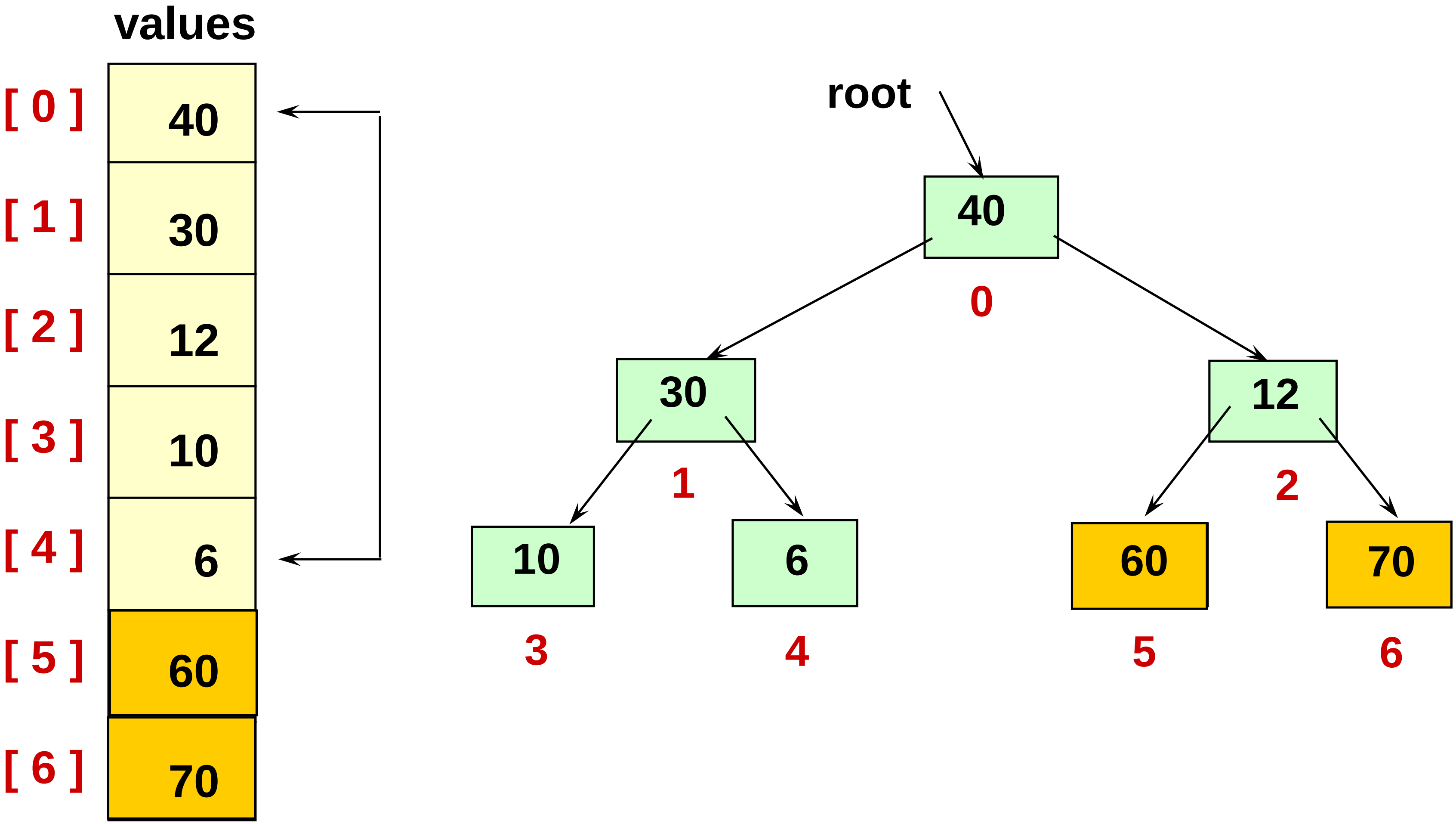
After swapping root element into its place



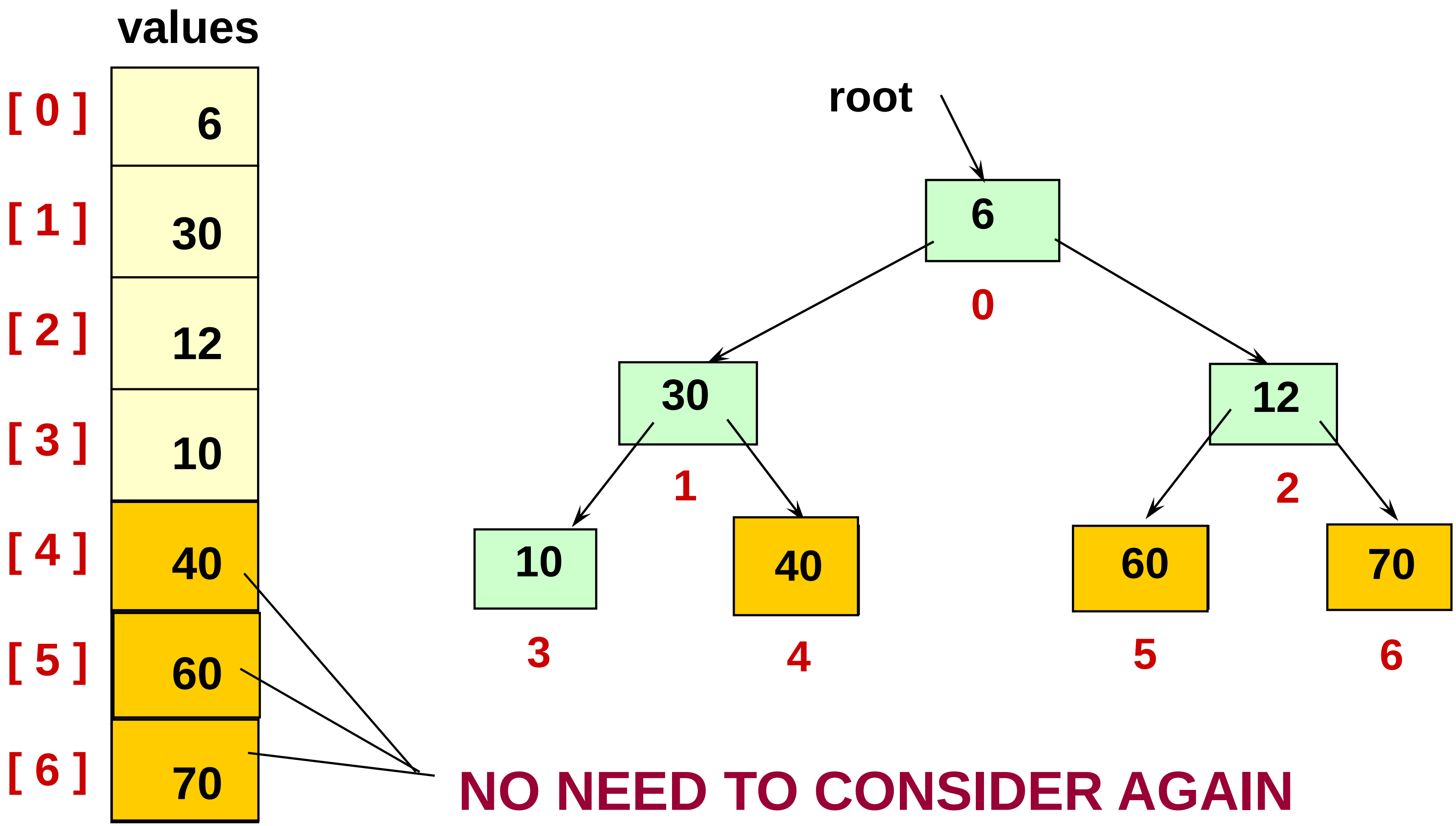
After reheaping remaining unsorted elements



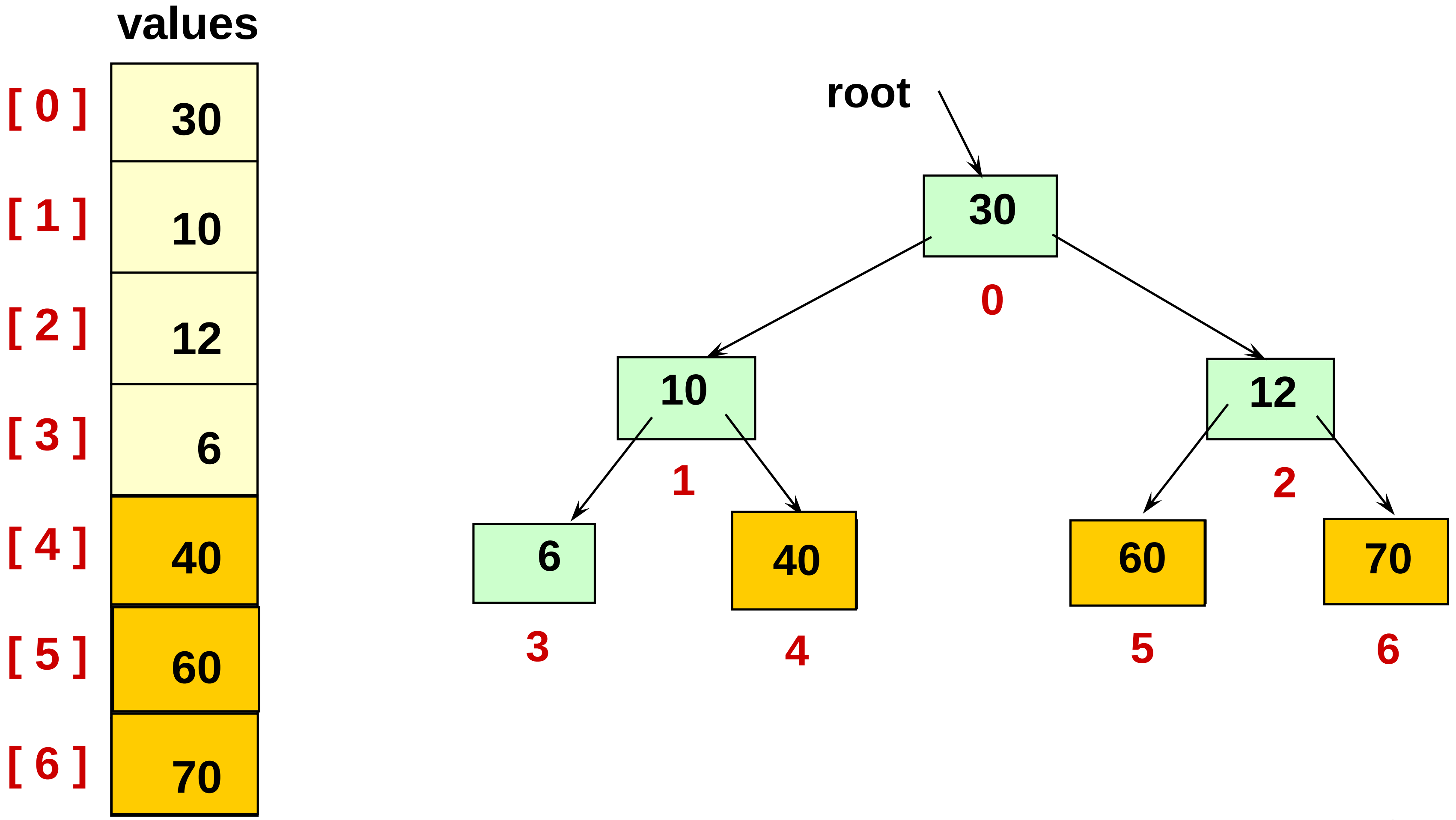
Swap root element into last place in unsorted array



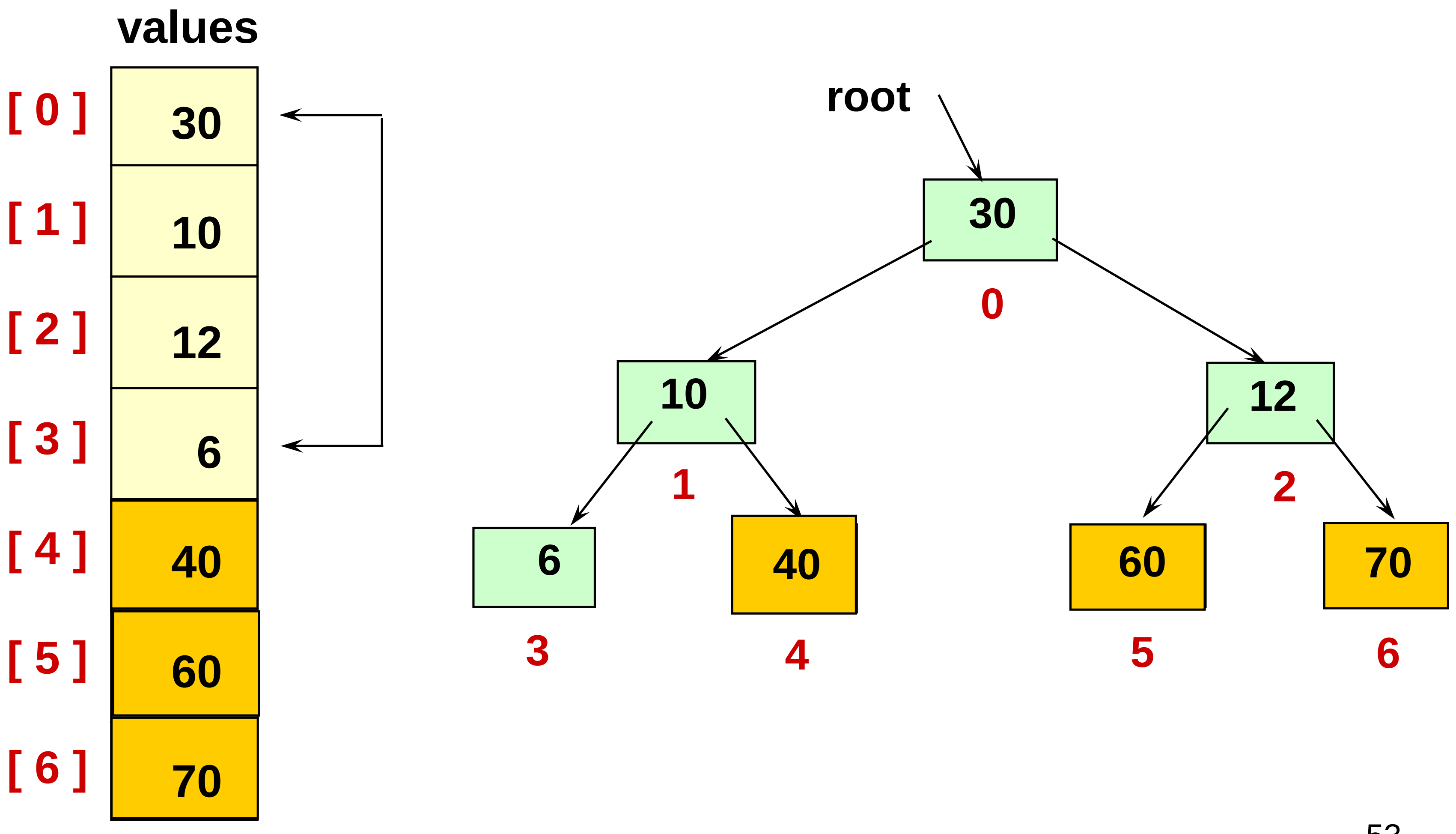
After swapping root element into its place



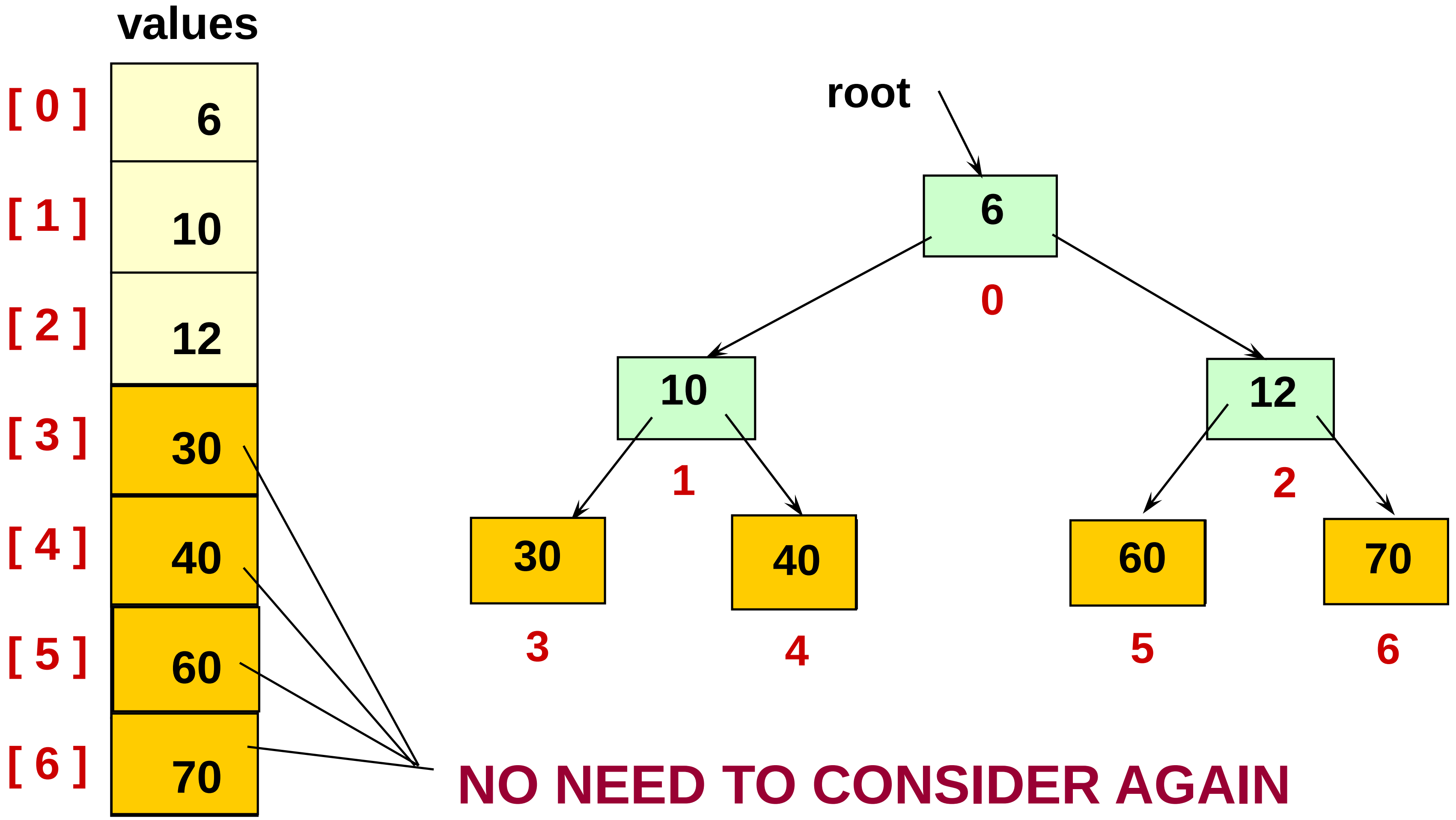
After reheaping remaining unsorted elements



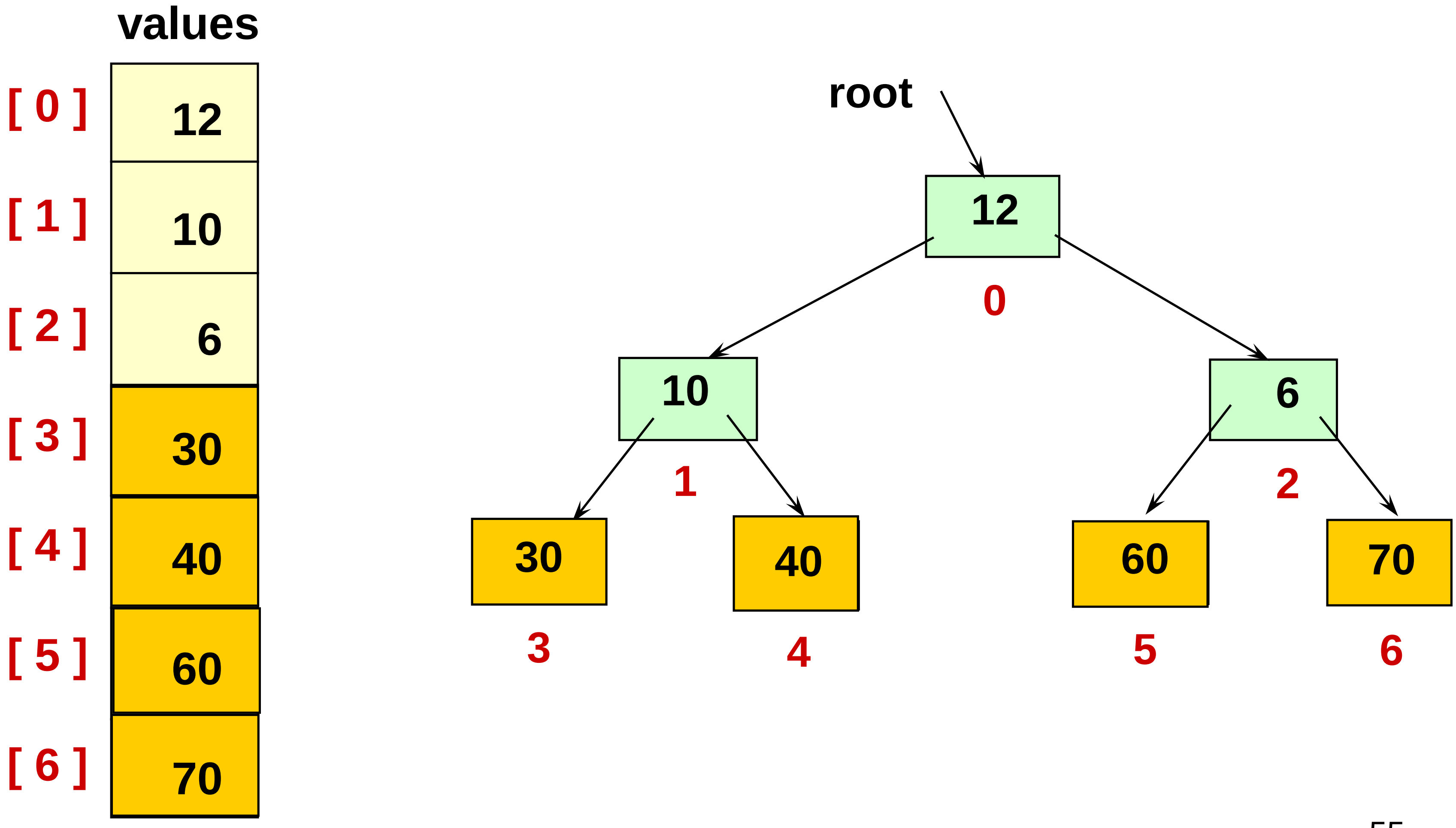
Swap root element into last place in unsorted array



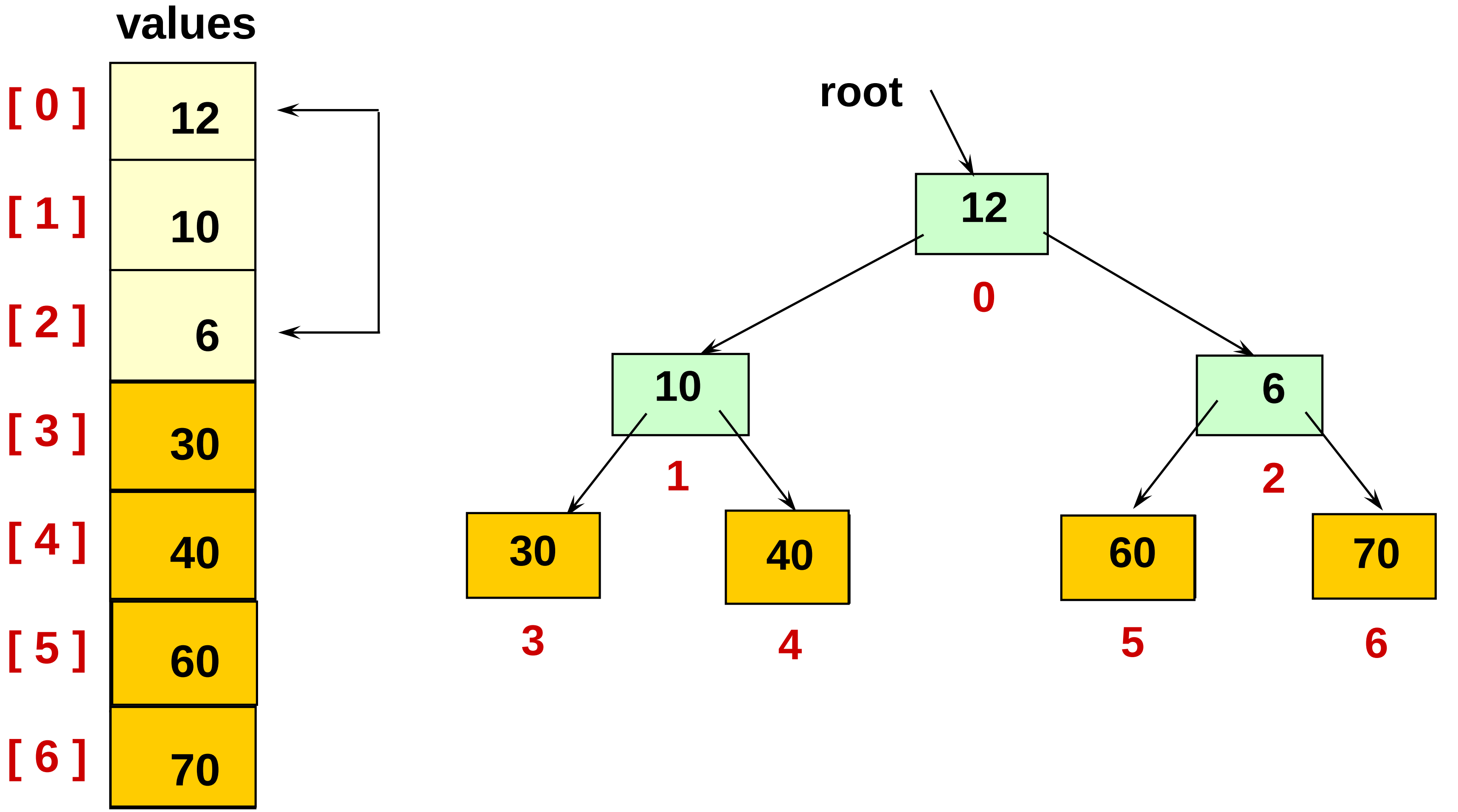
After swapping root element into its place



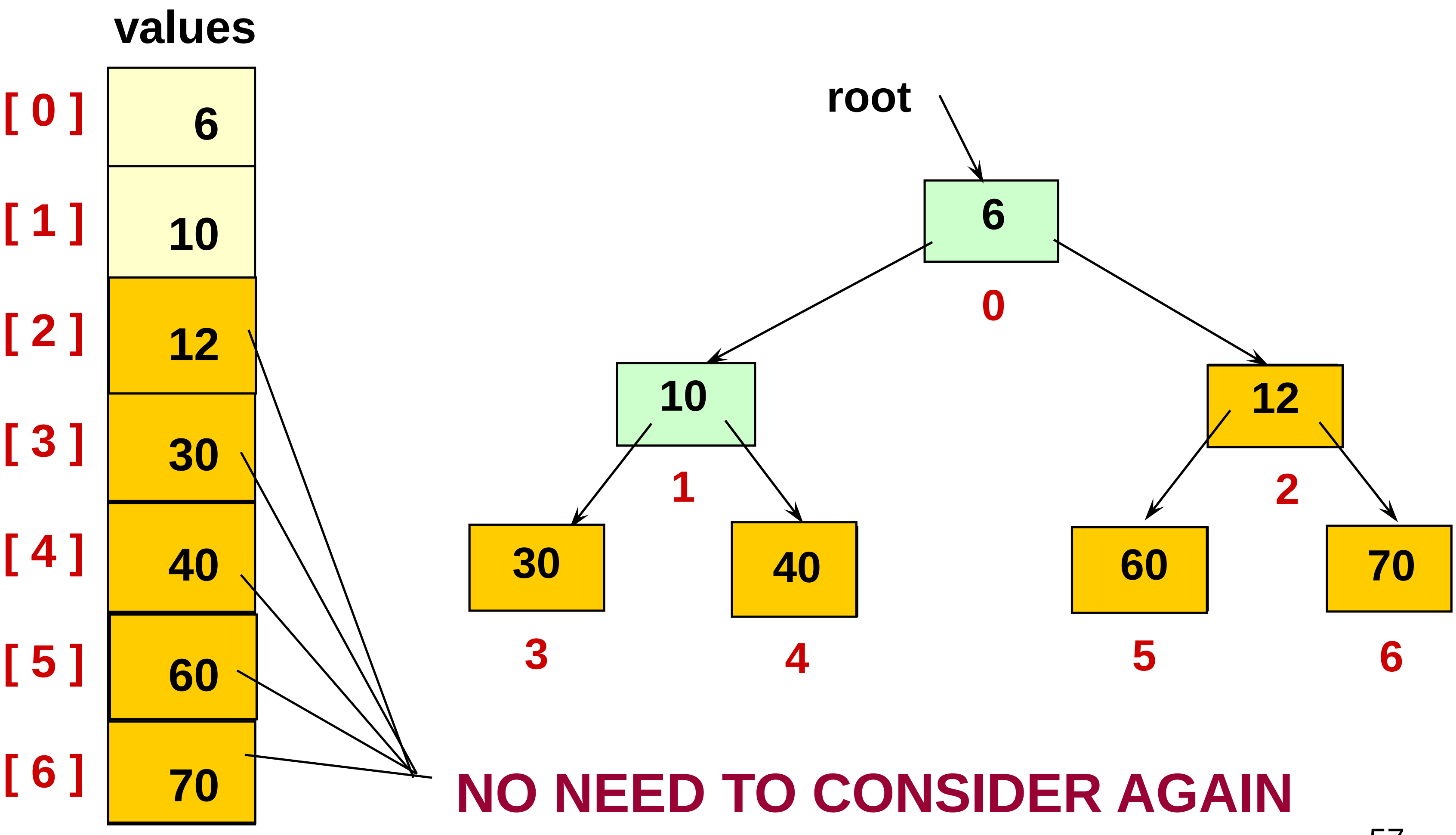
After reheaping remaining unsorted elements



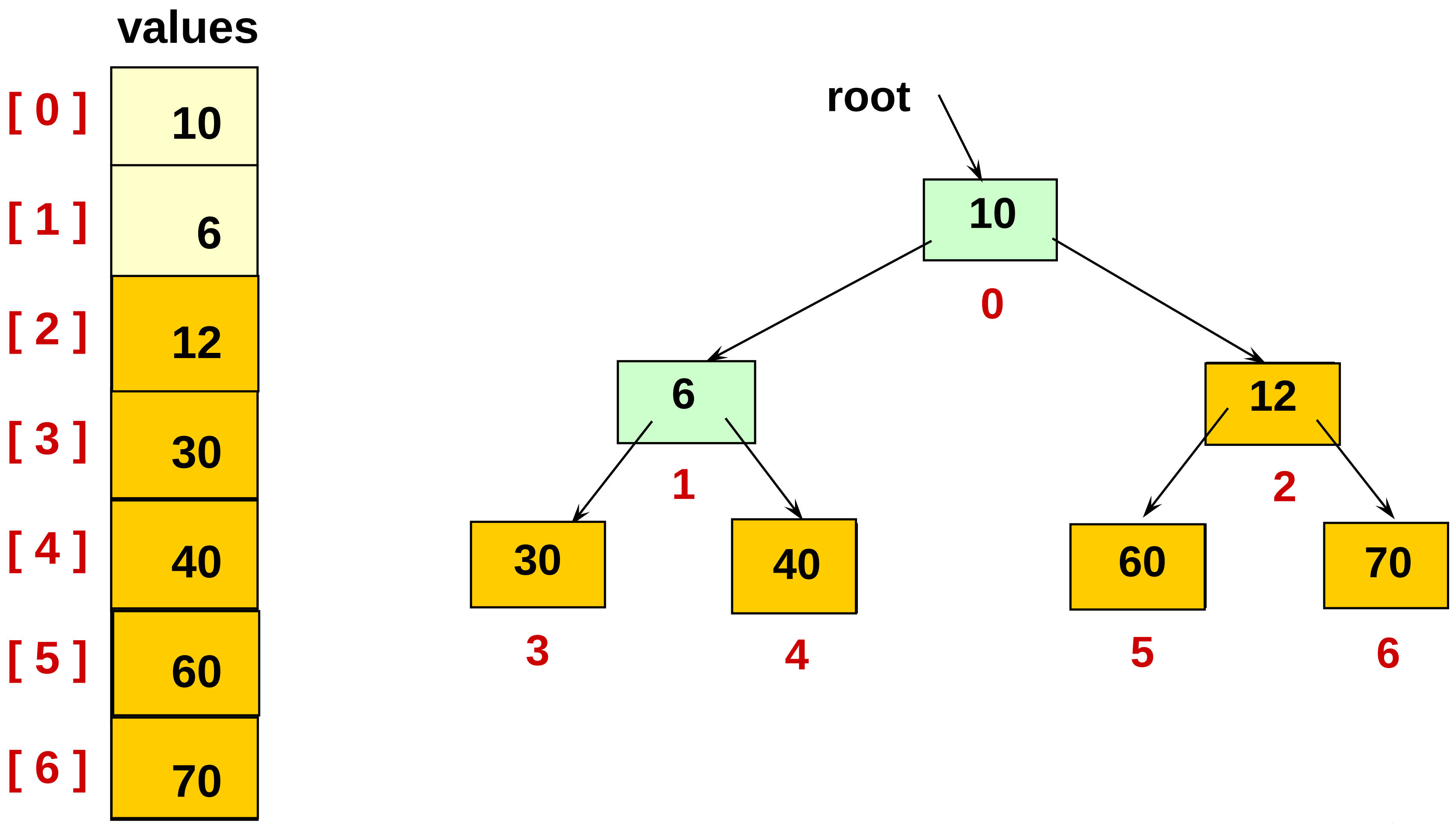
Swap root element into last place in unsorted array



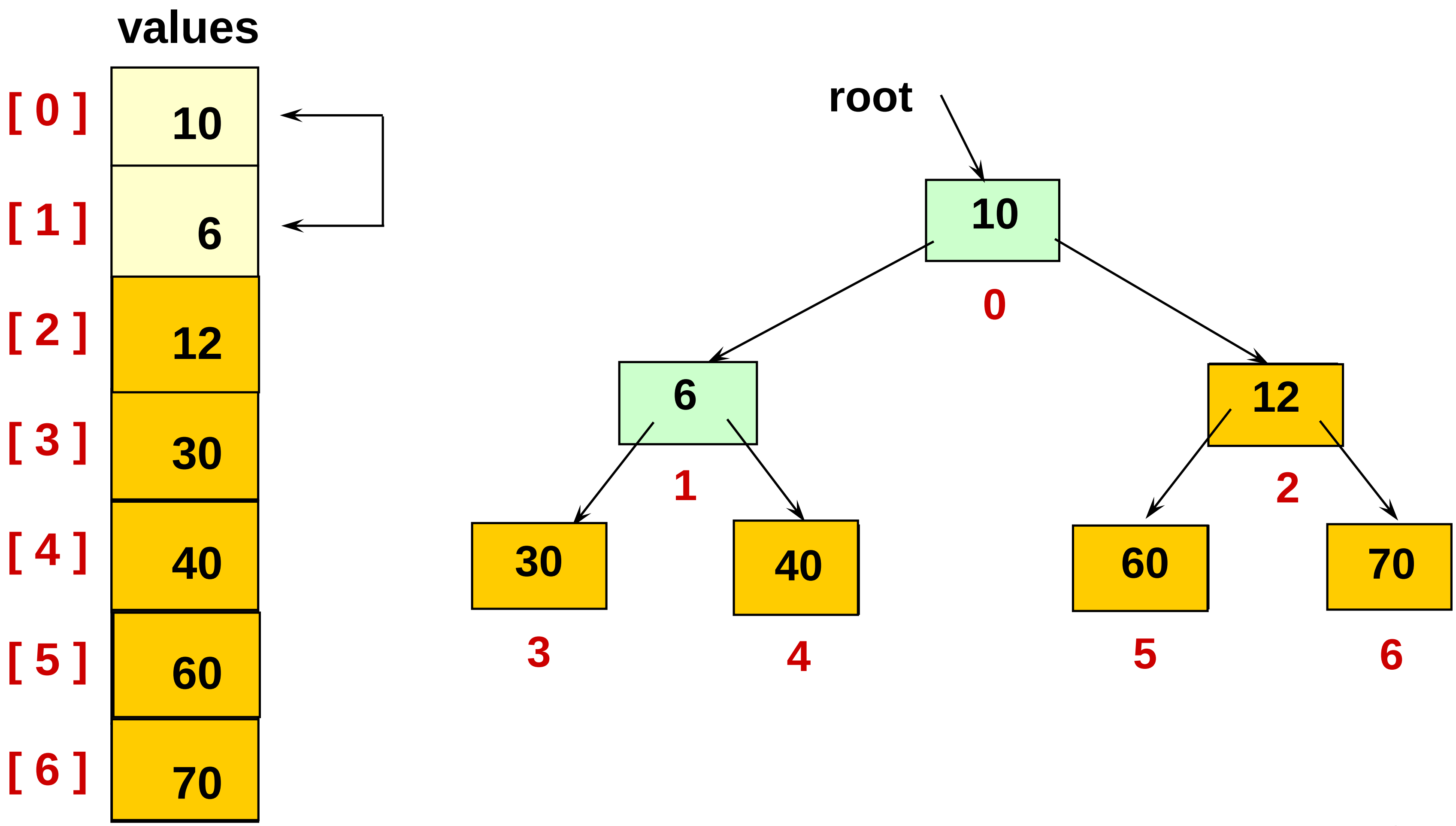
After swapping root element into its place



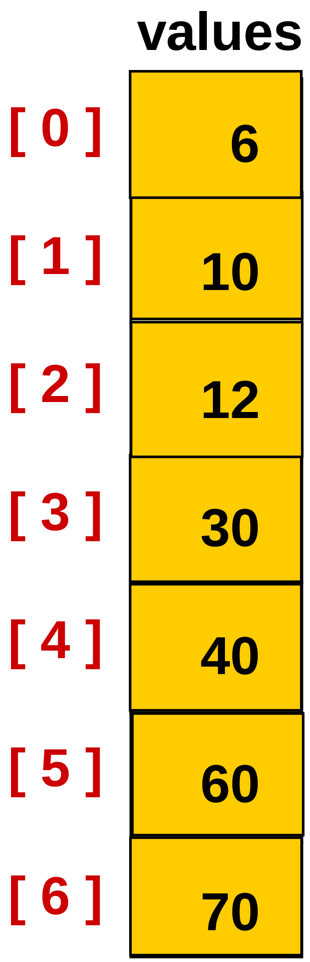
After reheaping remaining unsorted elements

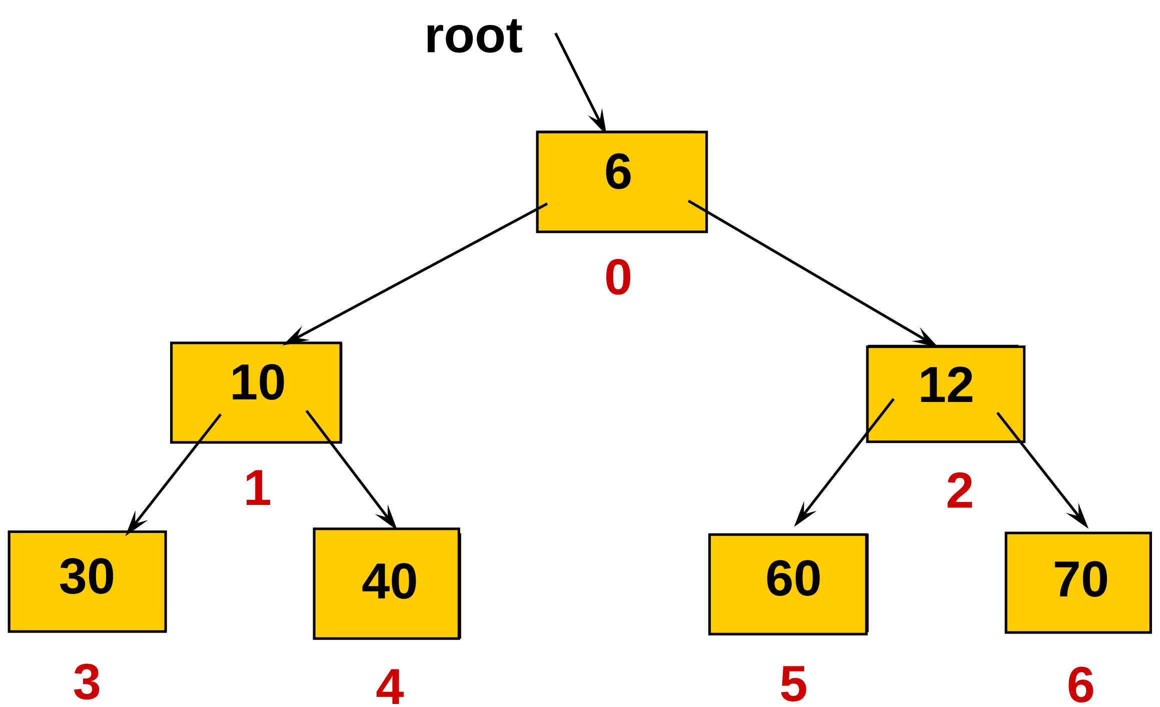


Swap root element into last place in unsorted array



After swapping root element into its place





ALL ELEMENTS ARE SORTED

```
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 ) ;
    }
}
```

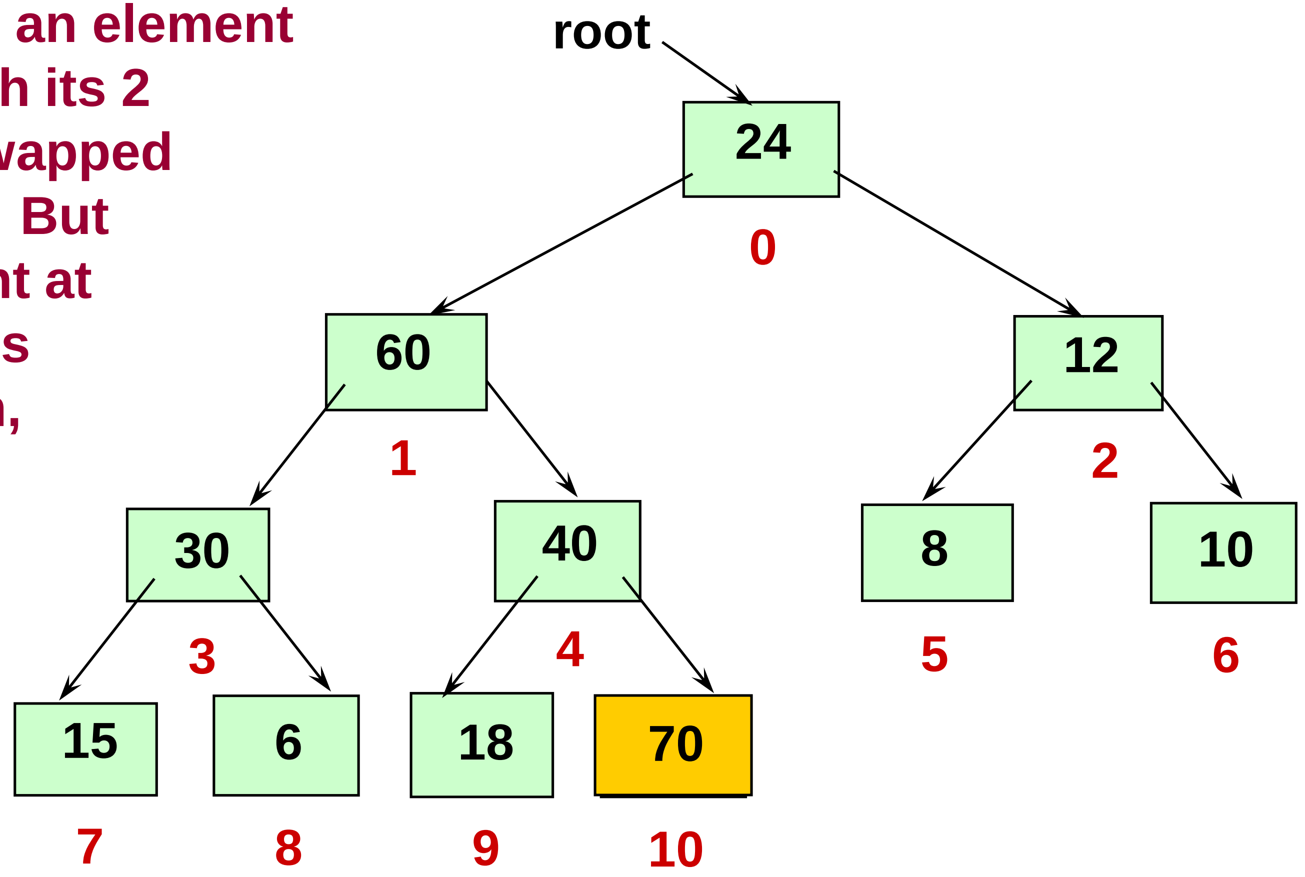
ReheapDown

```
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 ) ;
        }
    }
}
```

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.



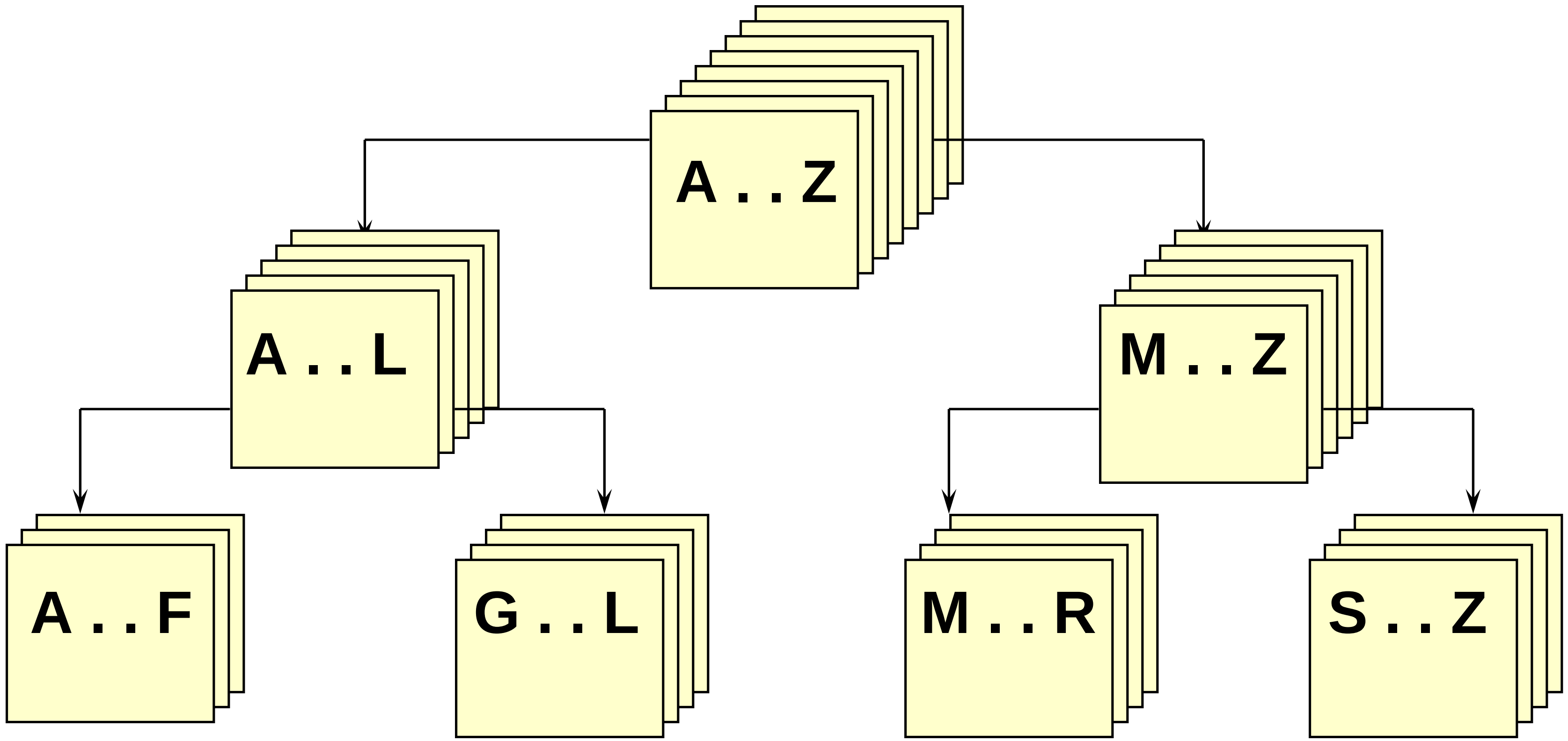
Heap Sort of N elements: How many comparisons?

(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

Using quick sort algorithm



Before call to function Split

splitVal = 9

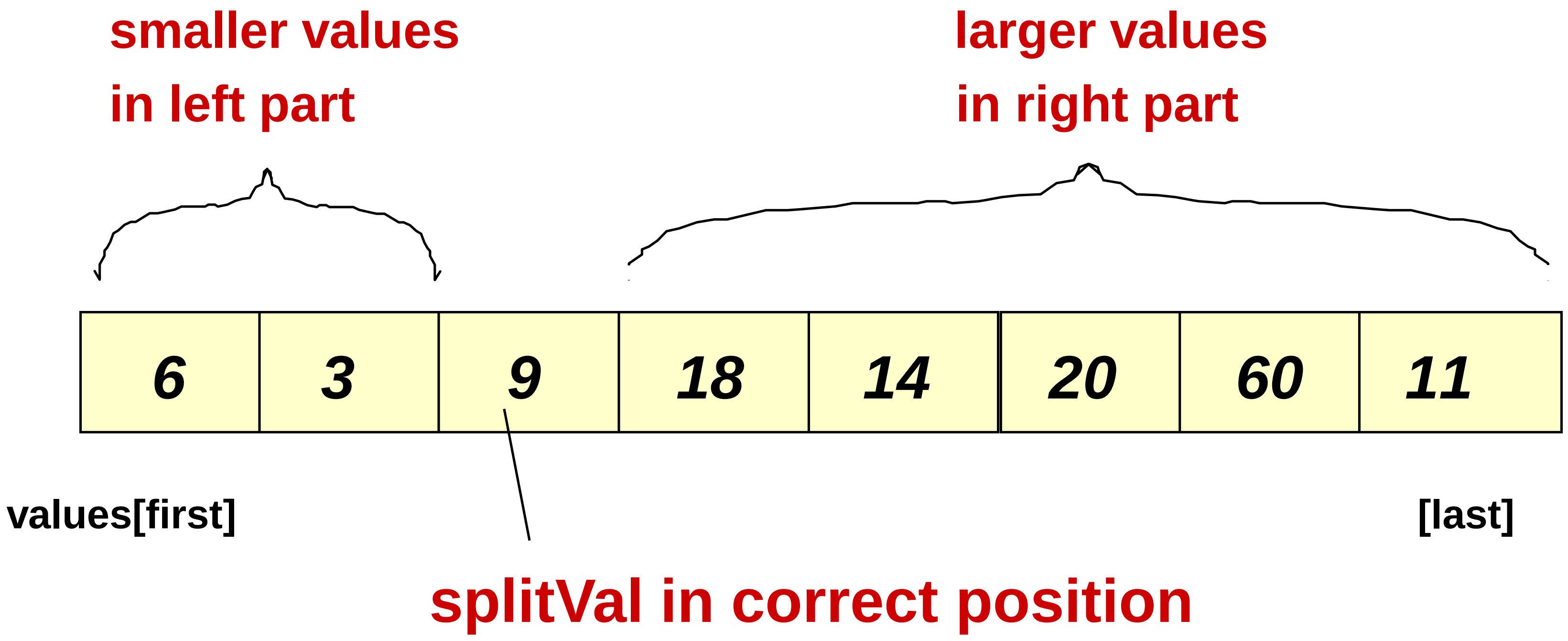
GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	*20*	*6*	*18*	*14*	*3*	*60*	*11*

values[first] **[last]**

After call to function Split

splitVal = 9



After call to function Split

6	*3*	*9*	*18*	*14*	*20*	*60*	*11*
3	*6*	*9*	*11*	*14*	*18*	*20*	*60*
3	*6*	*9*	*11*	*14*	*18*	*20*	*60*
3	*6*	*9*	*11*	*14*	*18*	*20*	*60*

```
// Recursive quick sort algorithm

template <class  ItemType >
void  QuickSort  ( ItemType  values[ ] ,  int  first ,  int  last )

//  Pre:   first <= last
//  Post: Sorts array values[ first .  . last ] into ascending order
{
    if  ( first < last )                          //  general case

    {      int  splitPoint ;

           Split ( values, first, last, splitPoint ) ;

           // values [ first ] . . values[splitPoint - 1 ] <= splitVal
           // values  [ splitPoint ] = splitVal
           // values [ splitPoint + 1 ] . . values[ last ] > splitVal

           QuickSort( values,  first,  splitPoint - 1 ) ;
           QuickSort( values,  splitPoint + 1,  last );
    }

} ;
```

Quick Sort of N elements: How many comparisons?

N	**For first call, when each of N elements is compared to the split value**
2 * N/2	**For the next pair of calls, when N/2 elements in each “half” of the original array are compared to their own split values.**
4 * N/4	**For the four calls when N/4 elements in each “quarter” of original array are compared to their own split values.**

.

.

. **HOW MANY SPLITS CAN OCCUR?**

Quick Sort of N elements: How many splits can occur?

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N*\log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as N-1 splits, and QuickSort is $O(N^2)$.

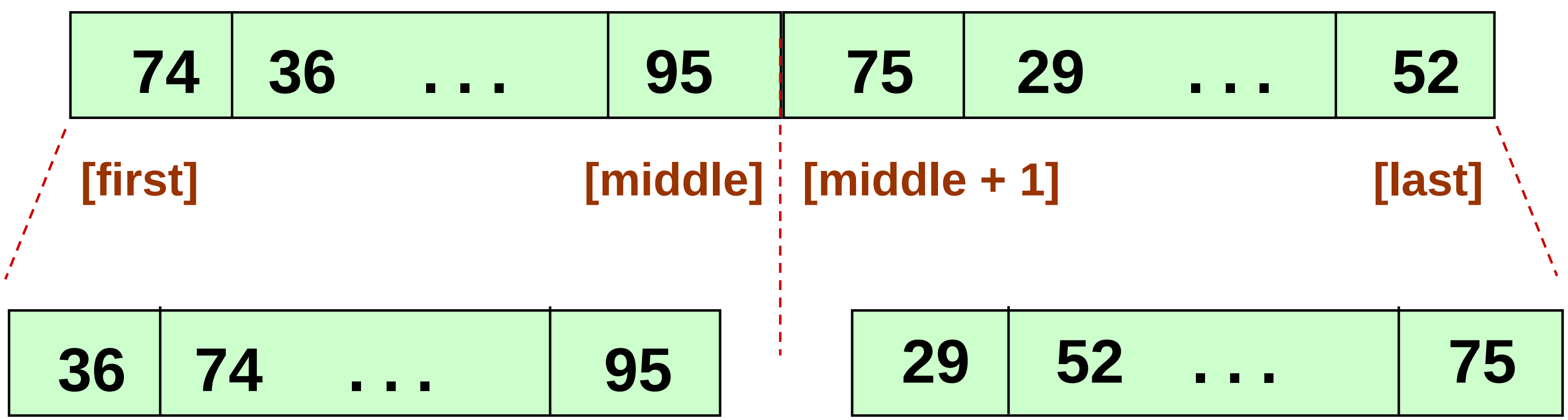
Merge Sort Algorithm

Cut the array in half.

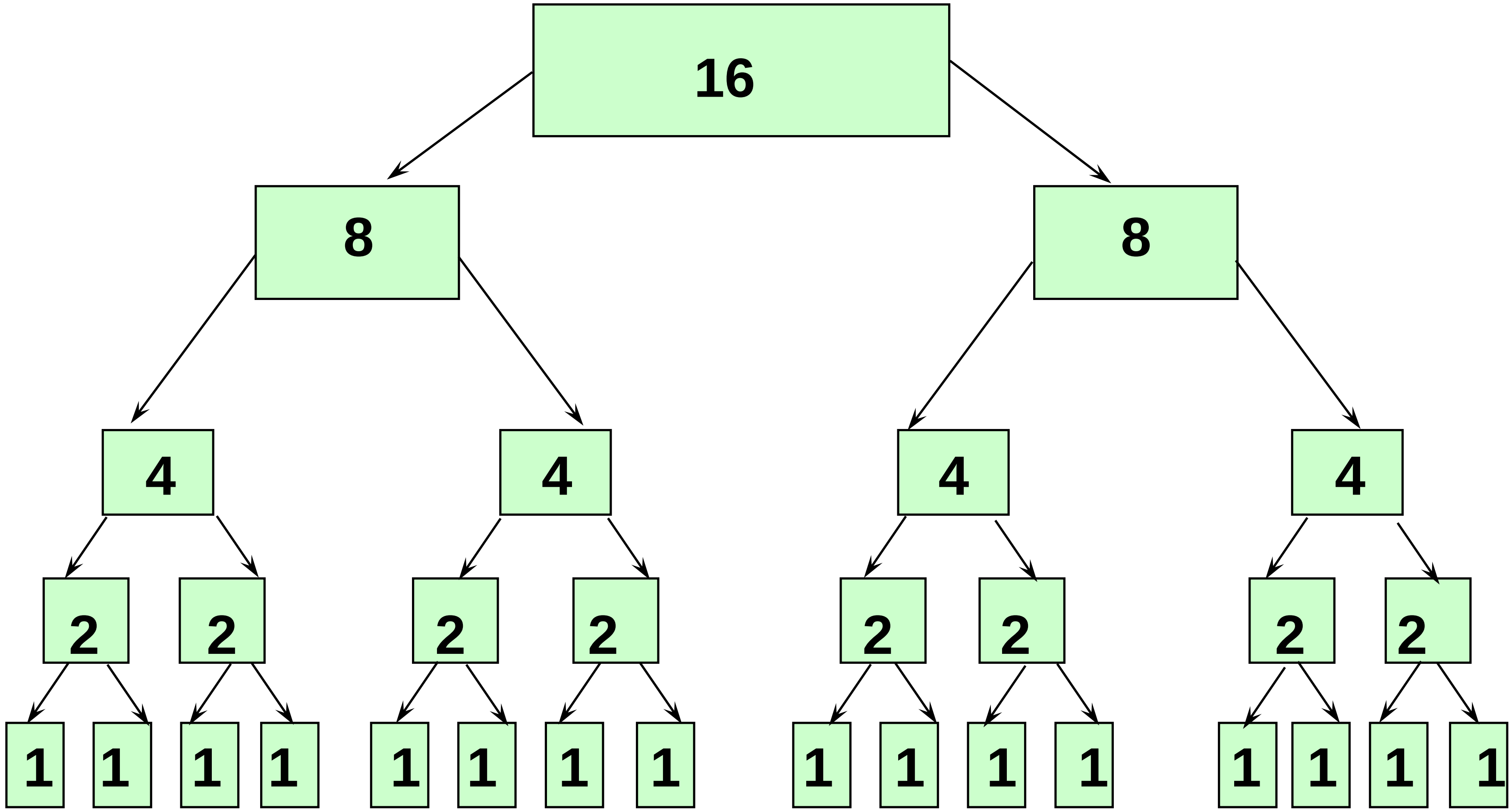
Sort the left half.

Sort the right half.

Merge the two sorted halves into one sorted array.



Using Merge Sort Algorithm with N = 16



```
// Recursive merge sort algorithm

template <class  ItemType >
void  MergeSort  ( ItemType  values[ ] ,  int  first ,  int  last )

//  Pre:   first <= last
//  Post: Array values[ first . . last ] sorted into ascending order.
{
    if  ( first < last )                          //  general case

    {      int  middle = ( first  +  last ) / 2  ;

           MergeSort ( values, first, middle ) ;

           MergeSort( values,  middle + 1, last ) ;

           // now  merge two subarrays
           // values  [ first . . . middle ] with
           // values [ middle + 1,  . . . last ].

           Merge( values,  first, middle, middle + 1, last ) ;
    }
}
```

Merge Sort of N elements: How many comparisons?

The entire array can be subdivided into halves only $\log_2 N$ times.

Each time it is subdivided, function Merge is called to re-combine the halves. Function Merge uses a temporary array to store the merged elements. Merging is O(N) because it compares each element in the subarrays.

Copying elements back from the temporary array to the values array is also O(N).

MERGE SORT IS $O(N*\log_2 N)$.

Sorting Algorithm Comparison

[Algorithm Visualization]

[Algorithm Comparison]

Function `BinarySearch( )`

- `BinarySearch` takes **sorted** array `info`, and two subscripts, `fromLoc` and `toLoc`, and `item` as arguments. It returns false if `item` is not found in the elements `info[fromLoc…toLoc]`. Otherwise, it returns true.

- `BinarySearch` is $O(\log_2 N)$.

```
found = BinarySearch(info, 25, 0, 14 );
```

item fromLoc toLoc

indexes	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
info	**0**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**

16 18 20 22 24 26 28

24 26 28

24

NOTE: denotes element examined

```
template<class  ItemType>
bool  BinarySearch   (  ItemType  info[ ] ,  ItemType  item ,
                                 int   fromLoc ,    int  toLoc )
    //    Pre:   info [ fromLoc . . toLoc ] sorted in ascending order
    //    Post:   Function value =  ( item  in info [ fromLoc . . toLoc] )

{          int  mid ;
           if  ( fromLoc > toLoc )                    //  base case -- not found
                      return  false ;
           else  {
                  mid = ( fromLoc + toLoc ) / 2 ;

                  if ( info [ mid ] == item )      // base case-- found at mid

                        return  true  ;

                 else  if ( item < info [ mid ] )          //  search lower half
                        return BinarySearch ( info, item, fromLoc, mid-1 ) ;
                 else                                  // search upper half
                        return  BinarySearch( info, item, mid + 1, toLoc ) ;
           }
}
```

Hashing

- is a means used to order and access elements in a list quickly -- the goal is O(1) time -- by using a function of the key value to identify its location in the list.

- The function of the key value is called a hash function.

FOR EXAMPLE . . .

Using a hash function

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

Placing elements in the array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

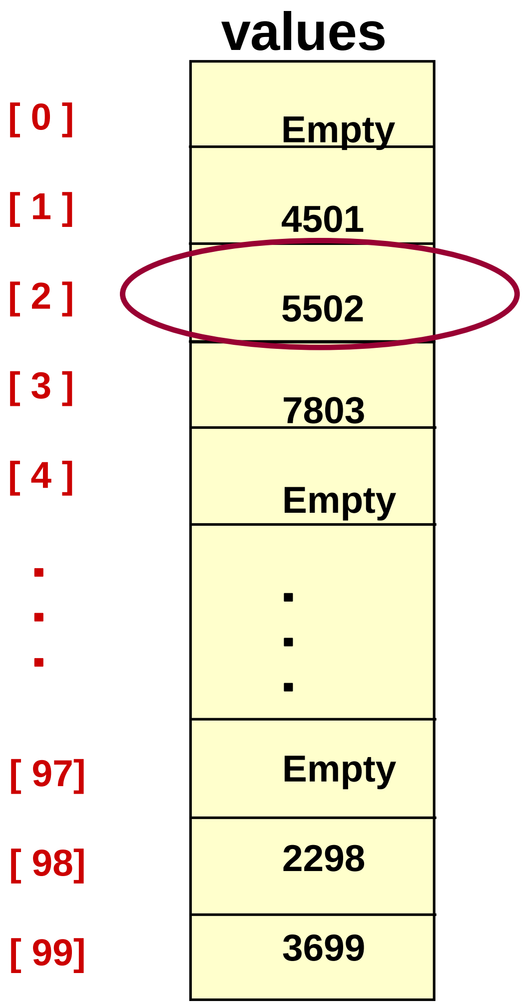
Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

Placing elements in the array



Next place part number 6702 in the array.

Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to resolve the collision?

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

One way is by linear probing. This uses the rehash function

(HashValue + 1) % 100

repeatedly until an empty location is found for part number 6702.

Resolving the collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

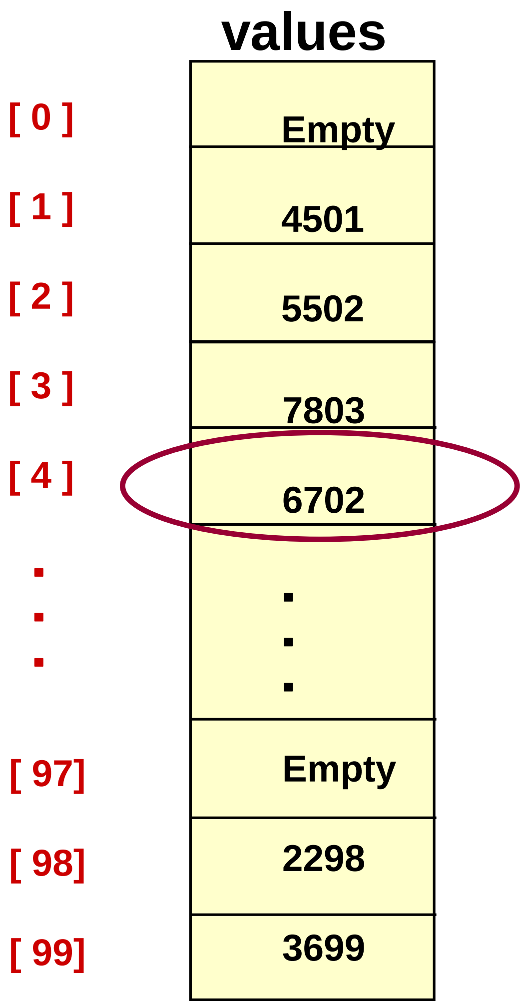
(HashValue + 1) % 100

Collision resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision resolved



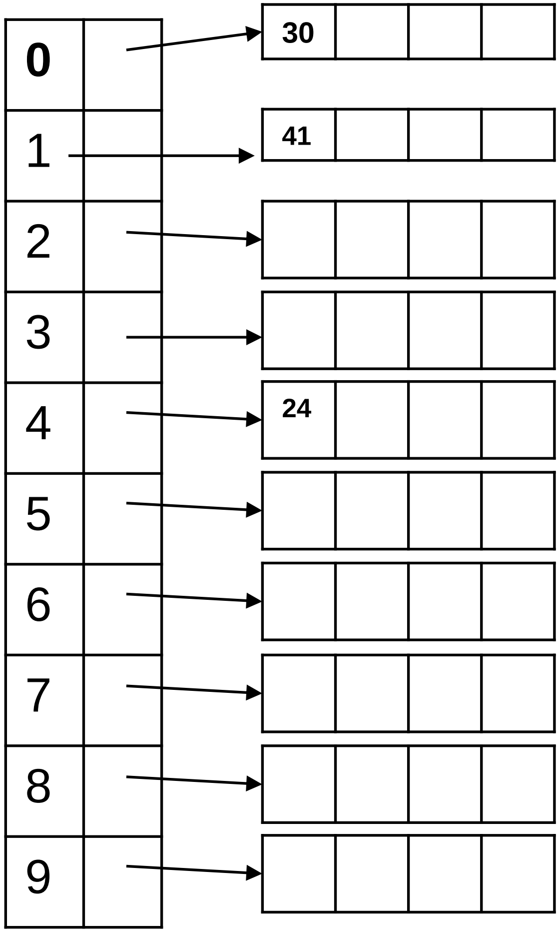
Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Radix Sort

Radix sort

- Is *not* a comparison sort
- Uses a radix-length array of queues of records
- Makes use of the values in digit positions in the keys to select the queue into which a record must be enqueued



Example of the Radix Sort

- Original Data:

  {762, 124, 432, 761, 002}

- First Pass(1digit):

  { 761, 002, 762, , 432, 124}

- Second Pass(10 digit):

  {002, 124, 432, 761, 762}

- Third Pass(100 digit) :

  {002, 124 , 432 , 761, 762}

Original Array

762
124
432
761
800
402
976
100
001
999

Queues After First Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800	761	762		124		976			999
100	001	432							
		402							

762
124
432
761
800
402
976
100
001
999

800
100
761
001
762
432
402
124
976
999

Queues After Second Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800		124	432			761	976		999
100						762			
001									
402									

800
100
761
001
762
432
402
124
976
999

800
100
001
402
124
432
761
762
976
999

Queues After Third Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
001	100			402			761	800	976
	124			432			762		999

800
100
001
402
124
432
761
762
976
999

001
100
124
402
432
761
762
800
976
999

Array After Third Pass

001
100
124
402
432
761
762
800
976
999