

Data Structures

Part X: Sorting and Searching Algorithm

BeomSeok Kim

Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

Sorting means...



- 배열에 저장된 값에는 관계 연산자가 정의된 유형의 키가 있음
The values stored in an array have keys of a type for which the relational operators are defined.
 - ✓ 고유키를 가정함
We also assume unique keys.
- 정렬은 배열 내에서 요소를 오름차순 또는 내림차순으로 재 배열
Sorting rearranges the elements into either ascending or descending order within the array.
 - ✓ 우선 오름차순을 사용
We'll use ascending order.

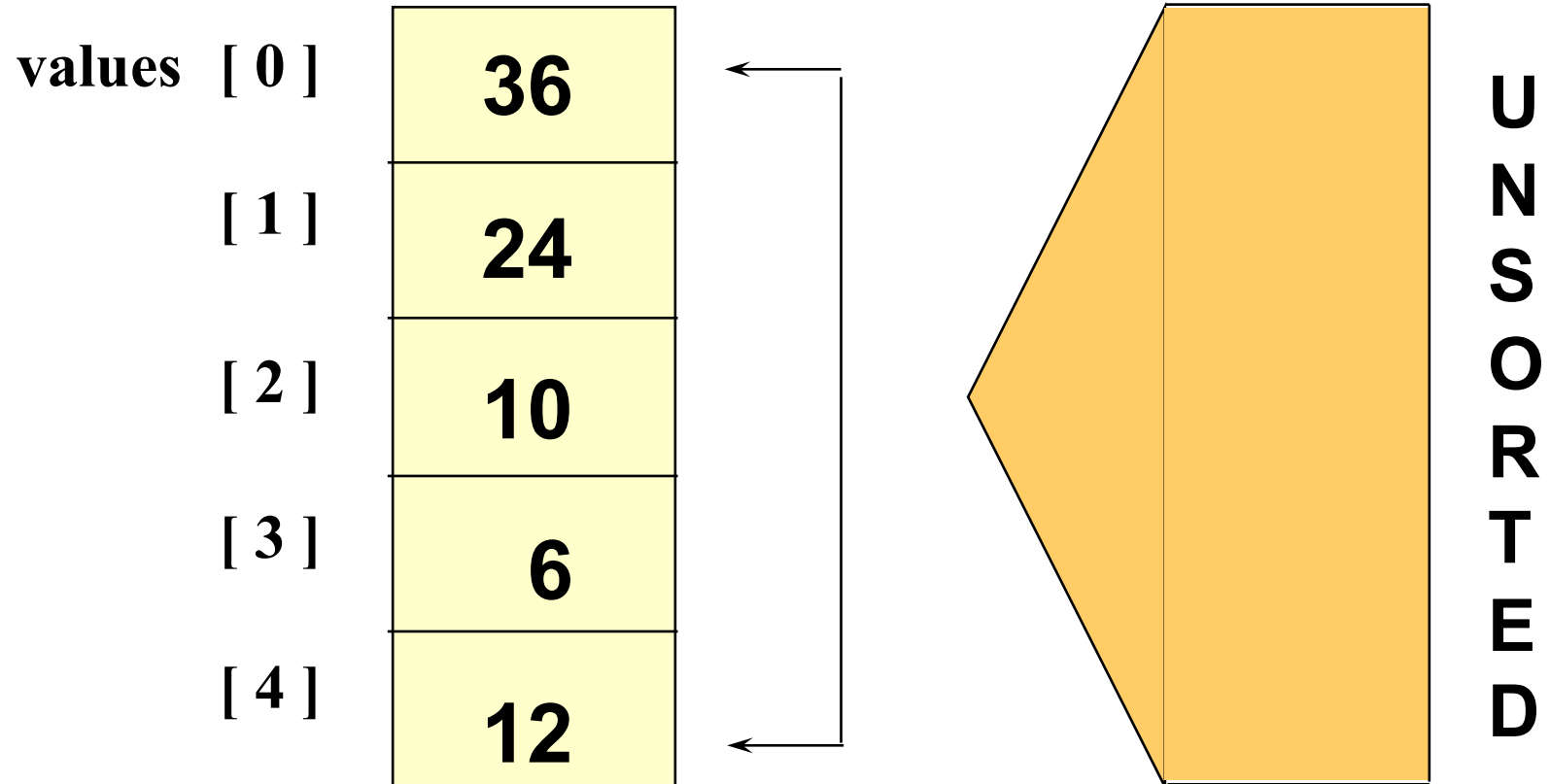
Straight Selection Sort



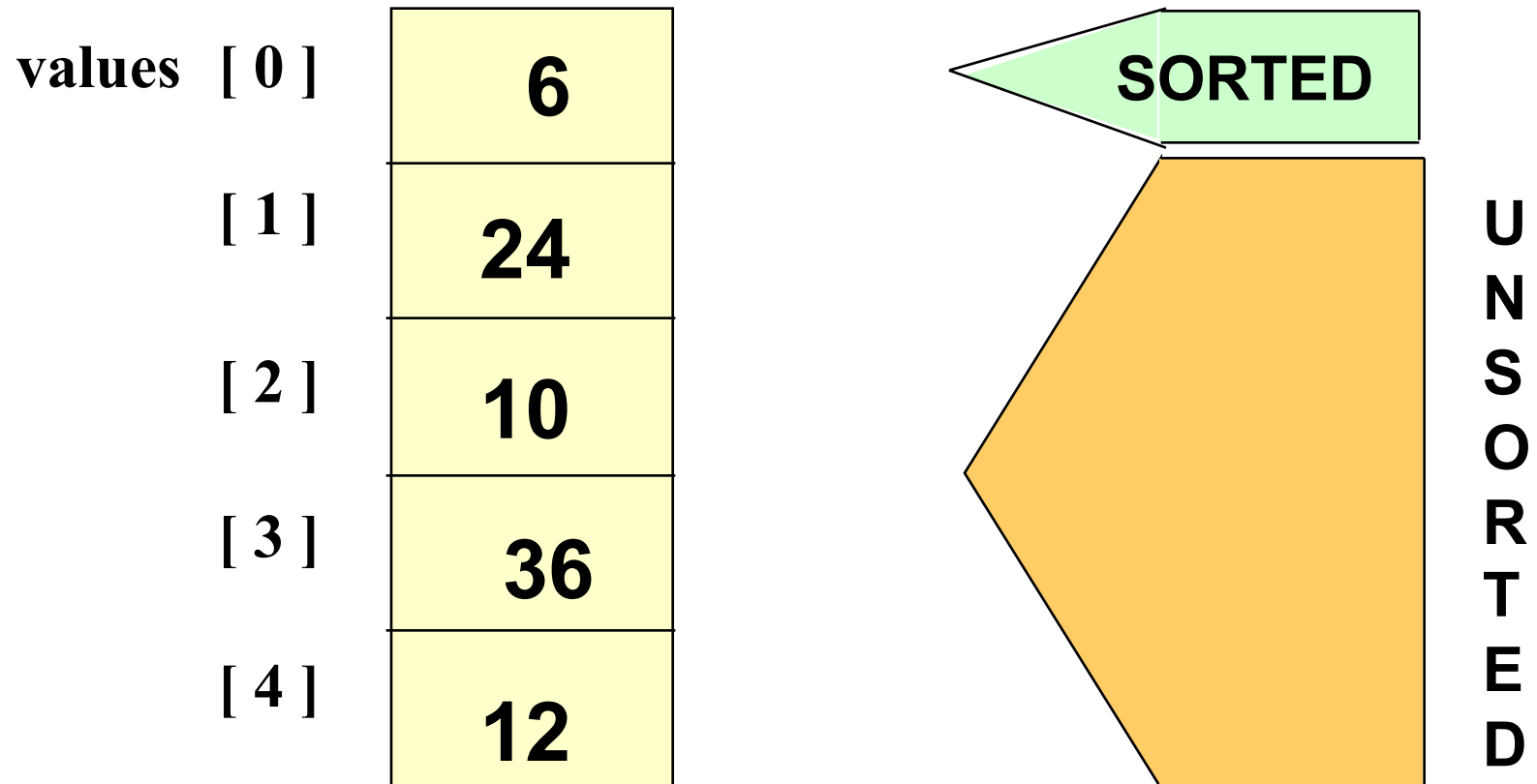
values [0]	36
[1]	24
[2]	10
[3]	6
[4]	12

- 배열을 두 개의 부분으로 나눔
Divides the array into two parts:
 - ✓ 정렬된 부분과 그렇지 않은 부분
already sorted, and not yet sorted.
- 각 패스에서 정렬되지 않은 요소 중 가장 작은 요소를 찾아 올바른 위치로 바꾸면 정렬된 요소 수가 하나씩 증가
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.

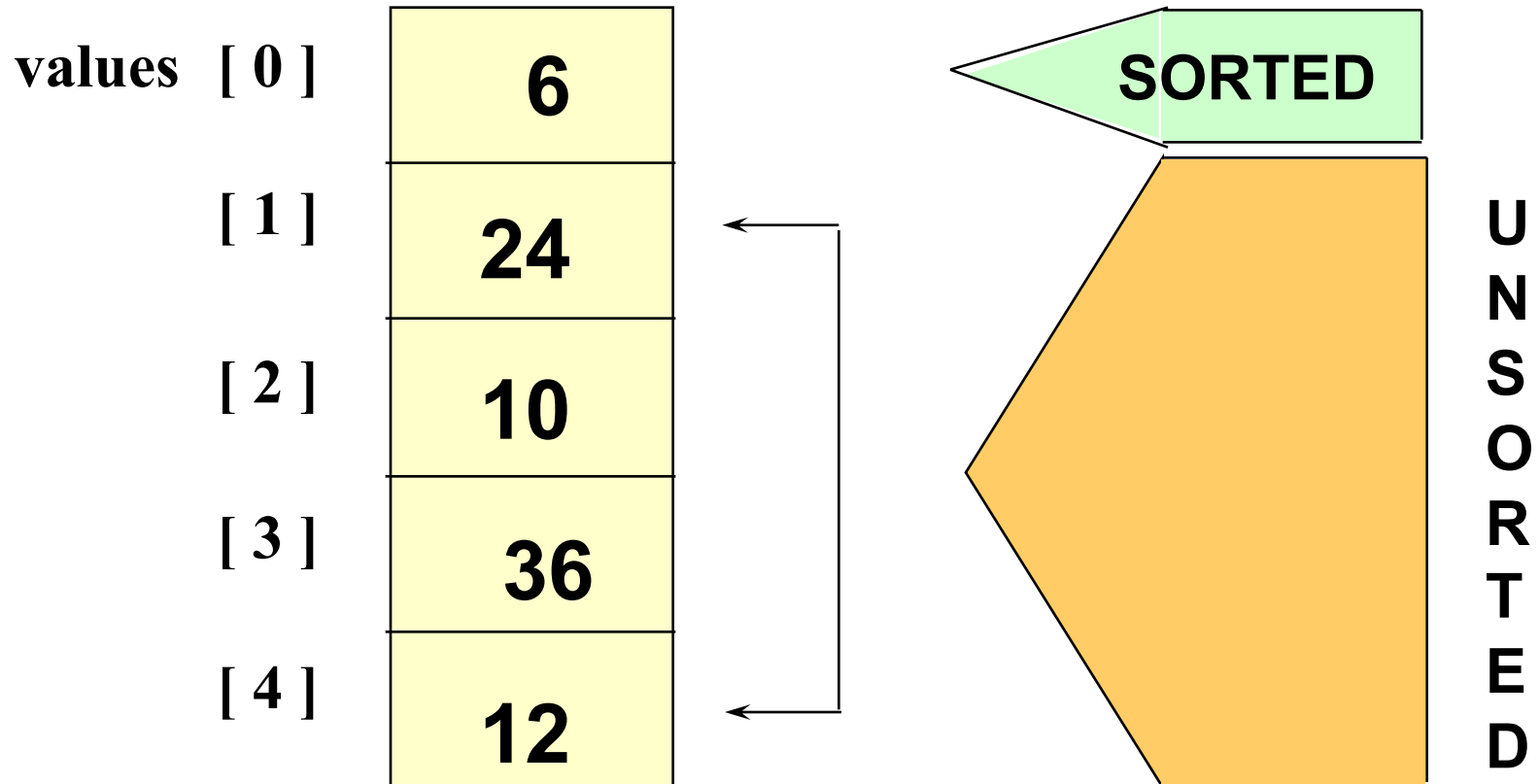
Selection Sort: Pass One



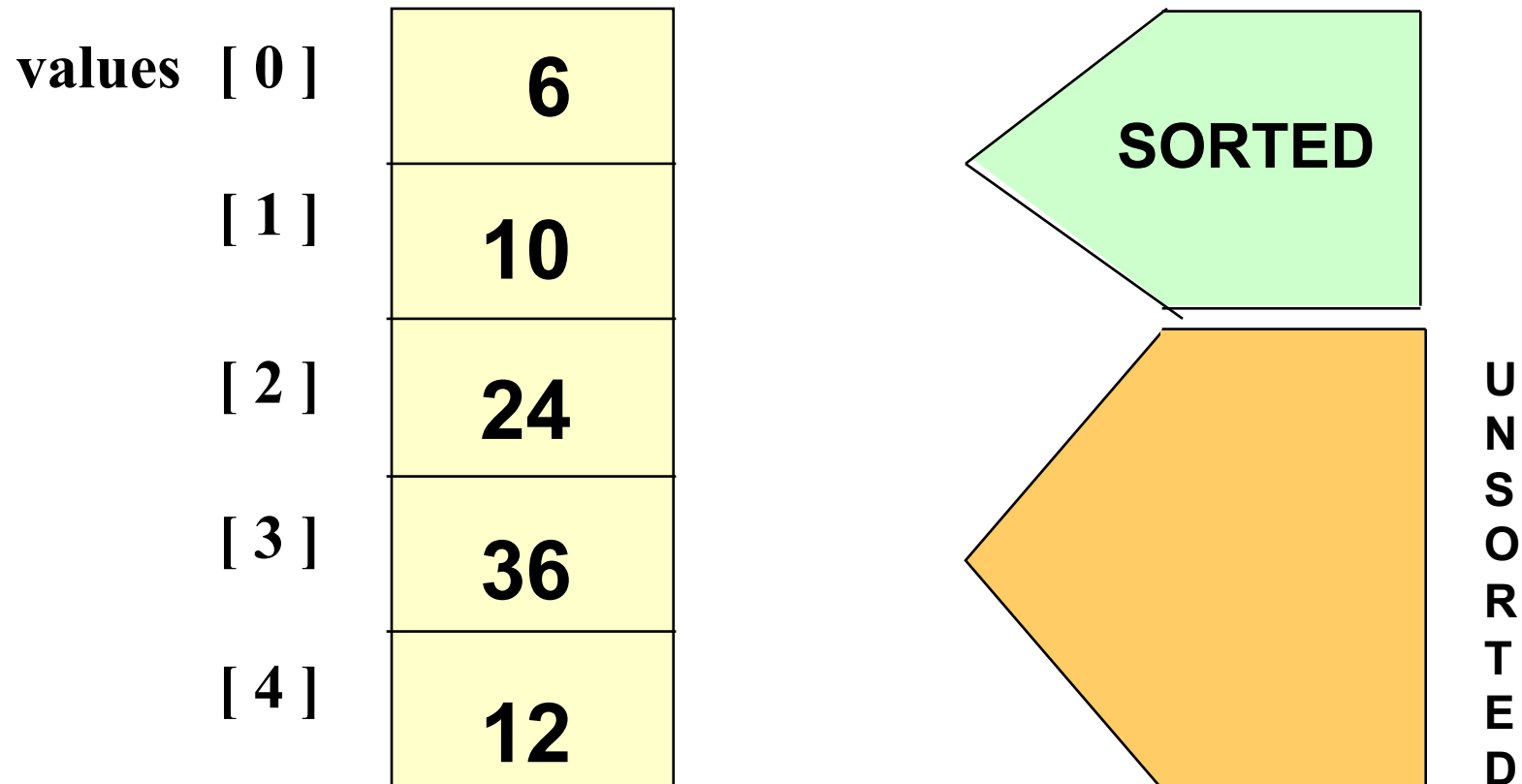
Selection Sort: End Pass One



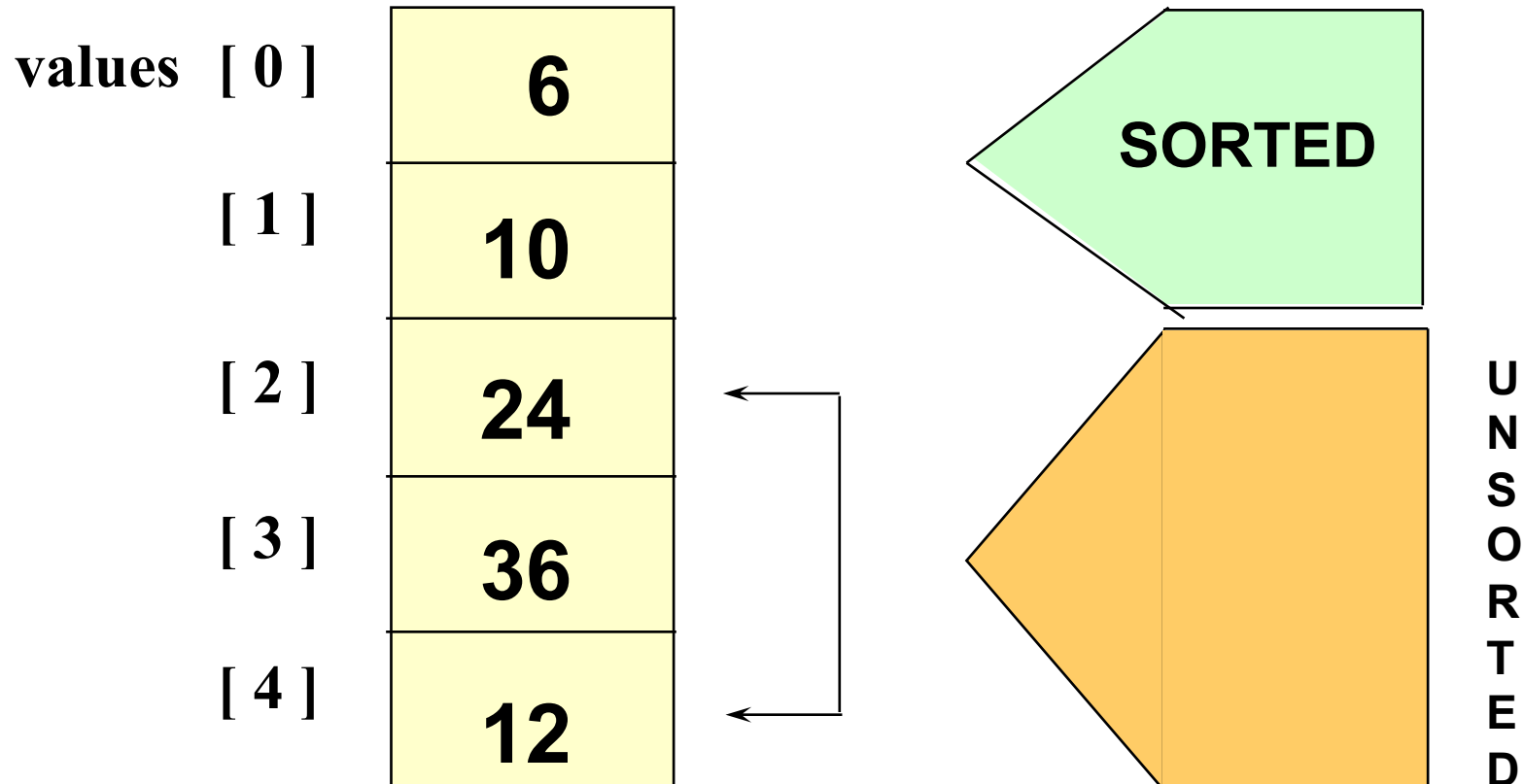
Selection Sort: Pass Two



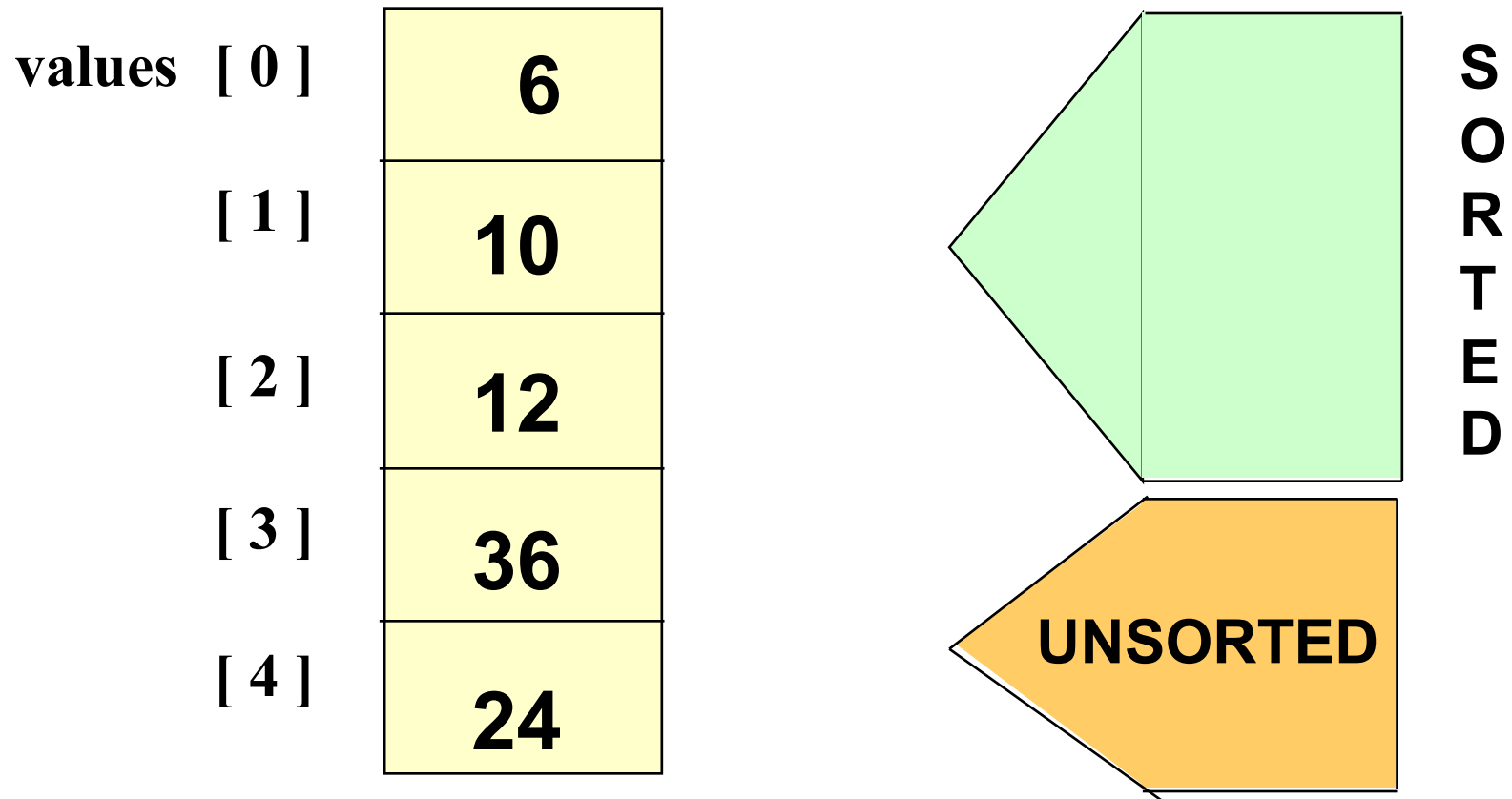
Selection Sort: End Pass Two



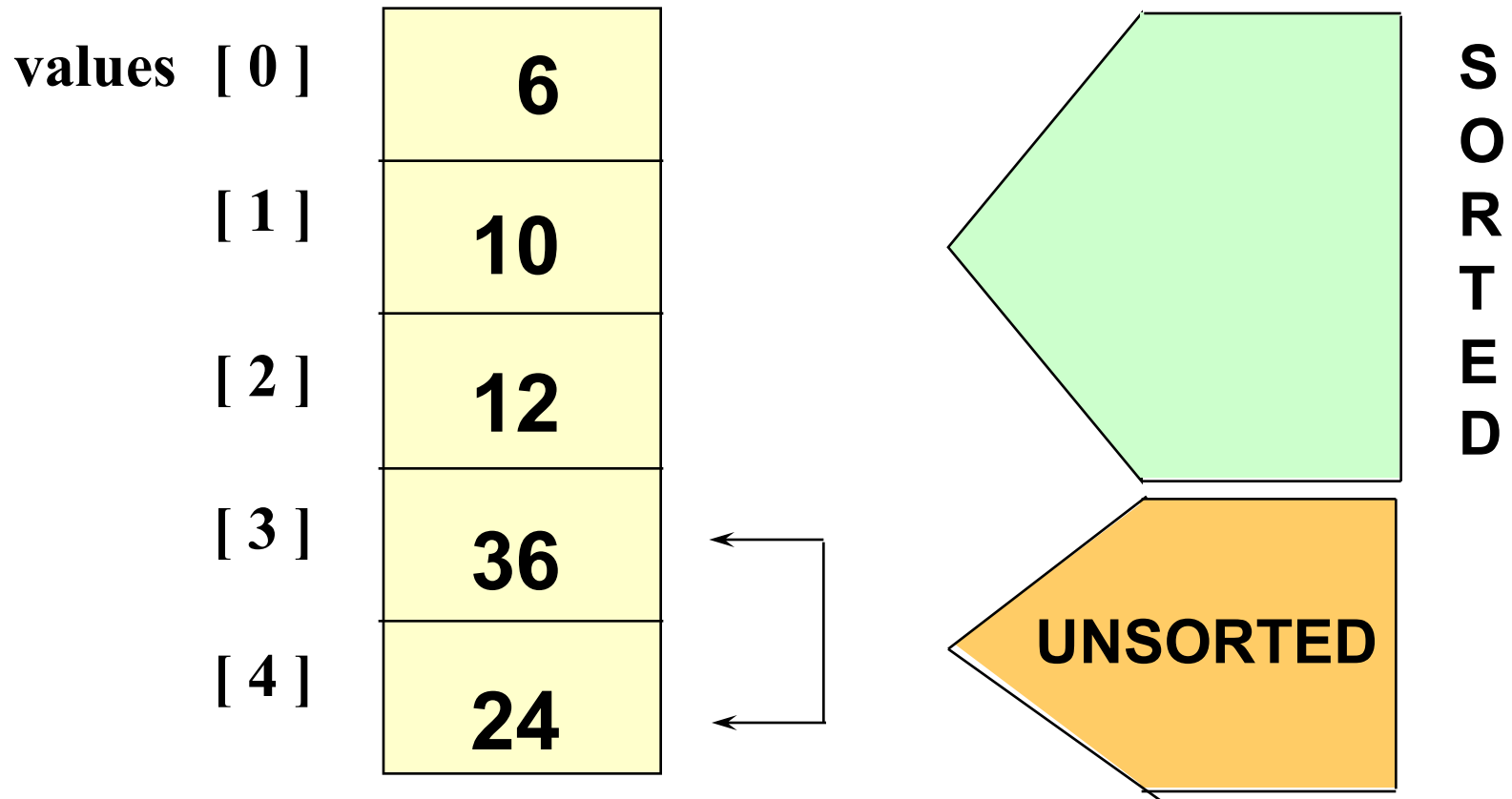
Selection Sort: Pass Three



Selection Sort: End Pass Three



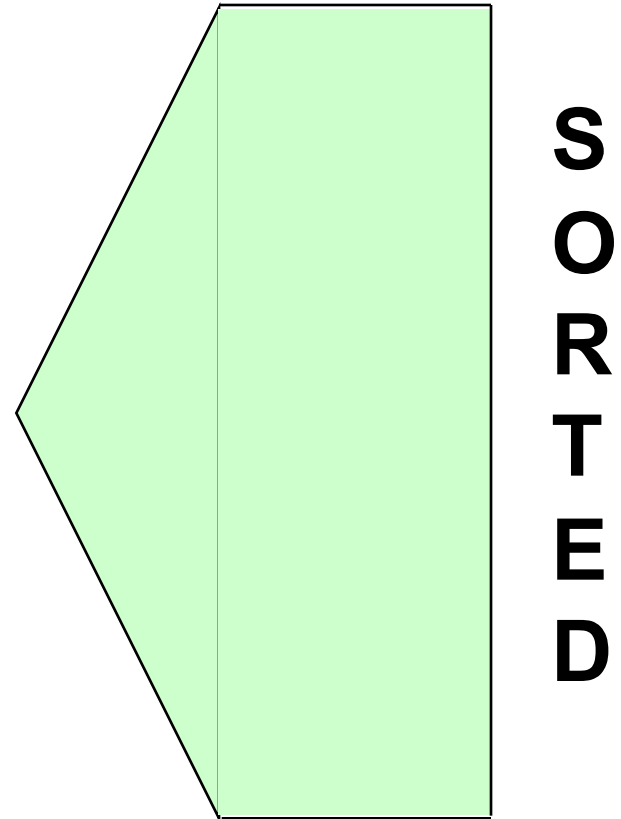
Selection Sort: Pass Four



Selection Sort: End Pass Four



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

- N개의 요소로 이루어진 배열을 정렬할 때 비교 횟수는
The number of comparisons when the array contains N elements is

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

Notice that . . .

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

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

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

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

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

For selection sort in general

- N개의 요소로 이루어진 배열을 정렬할 때 비교 횟수는
The number of comparisons when the array contains N elements is

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

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

$$\text{Sum} = .5 N^2 - .5 N$$

$$\text{Sum} = O(N^2)$$

Selection sort: Implementation



```
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;
}
```


Selection sort: Implementation



```
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

- 마지막 배열 요소부터 시작하여 인접 배열 요소 쌍을 비교하고, 올바른 순서가 아닐 때 마다 이웃을 교환
Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.
- 각 패스에서 가장 작은 요소가 배열의 올바른 위치로 “버블 업”됨
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

....

Bubble Sort: Implementation



```
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 ] ) ;
}
```

Bubble Sort: Implementation



```
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++ ;
    }
}
```

Bubble Sort: Implementation (Other ver.)

```
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

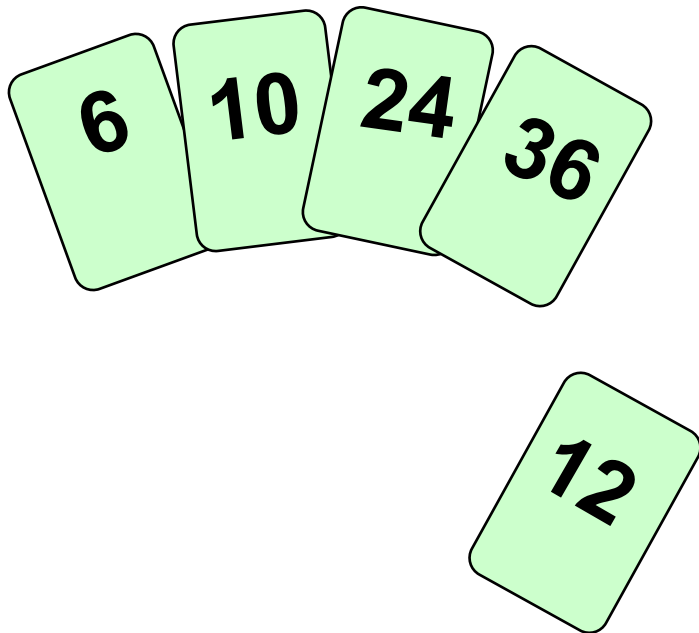
values [0]	36
[1]	24
[2]	10
[3]	6
[4]	12

- 정렬되지 않은 각 배열 요소는 하나씩 정렬되어 이미 정렬된 요소와 관련하여 적절한 위치에 삽입
One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.
- 각 패스에서 이미 정렬된 요소의 수가 1씩 증가
On each pass, this causes the number of already sorted elements to increase by one.

Insertion Sort



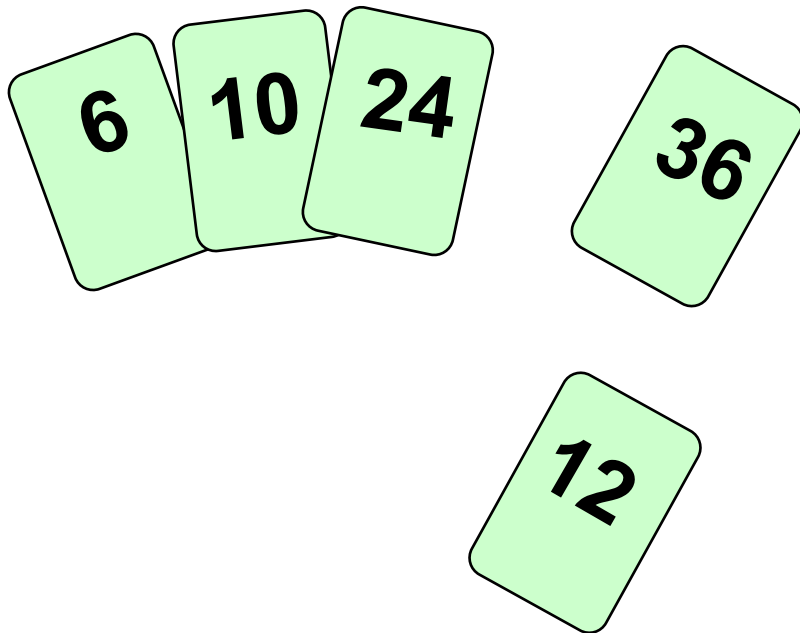
- 한번에 하나 이상의 카드를 이미 분류된 카드에 “삽입”하는 사람처럼 작동
Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.
- 12를 삽입하려면 먼저 36을 이동한 다음 24를 이동하여 공간을 확보해야 함
To insert 12, we need to make room for it by moving first 36 and then 24.



Insertion Sort



- 한번에 하나 이상의 카드를 이미 분류된 카드에 “삽입”하는 사람처럼 작동
Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

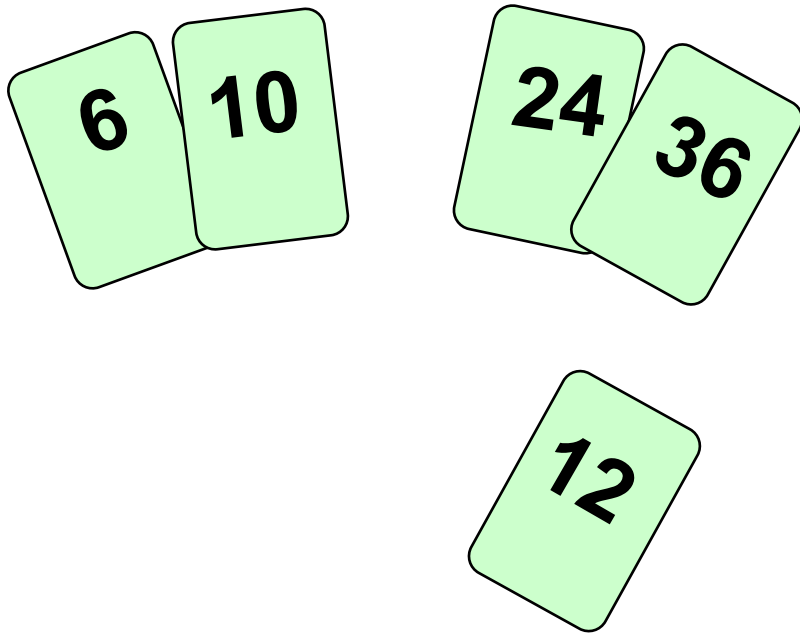


- 12를 삽입하려면 먼저 36을 이동한 다음 24를 이동하여 공간을 확보해야 함
To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



- 한번에 하나 이상의 카드를 이미 분류된 카드에 “삽입”하는 사람처럼 작동
Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

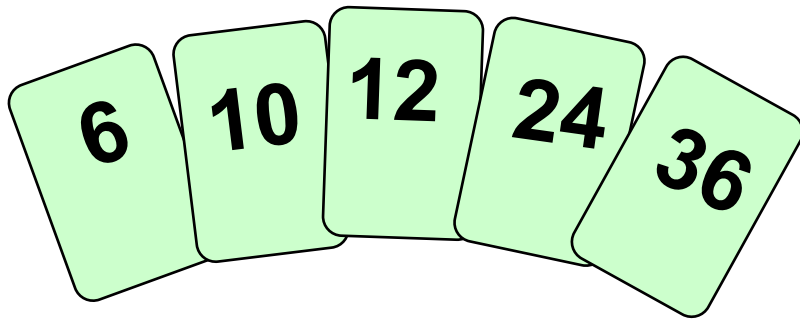


- 12를 삽입하려면 먼저 36을 이동한 다음 24를 이동하여 공간을 확보해야 함
To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



- 한번에 하나 이상의 카드를 이미 분류된 카드에 “삽입”하는 사람처럼 작동
Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.



- 12를 삽입하려면 먼저 36을 이동한 다음 24를 이동하여 공간을 확보해야 함
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

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

4

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

Insertion Sort: Implementation



```
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 ;
    }
}
```

Insertion Sort: Implementation



```
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 ) ;

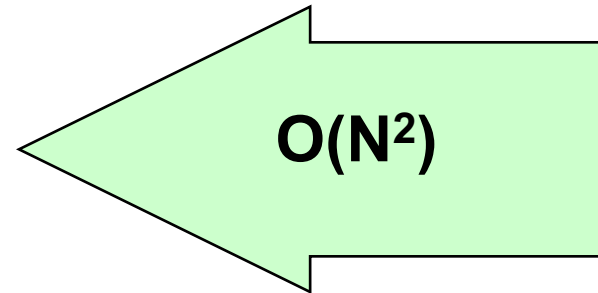
}
```

Sorting Algorithms and Average Case Number of Comparisons



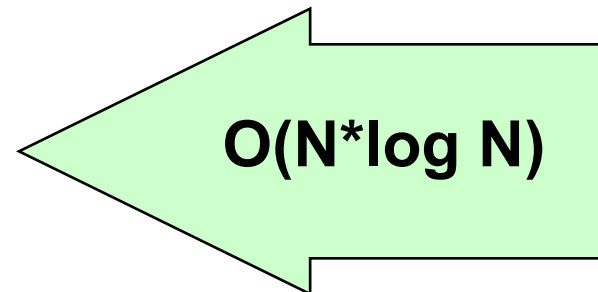
Simple Sorts

- Straight Selection Sort
- Bubble Sort
- Insertion Sort



More Complex Sorts

- Quick Sort
- Merge Sort
- Heap Sort

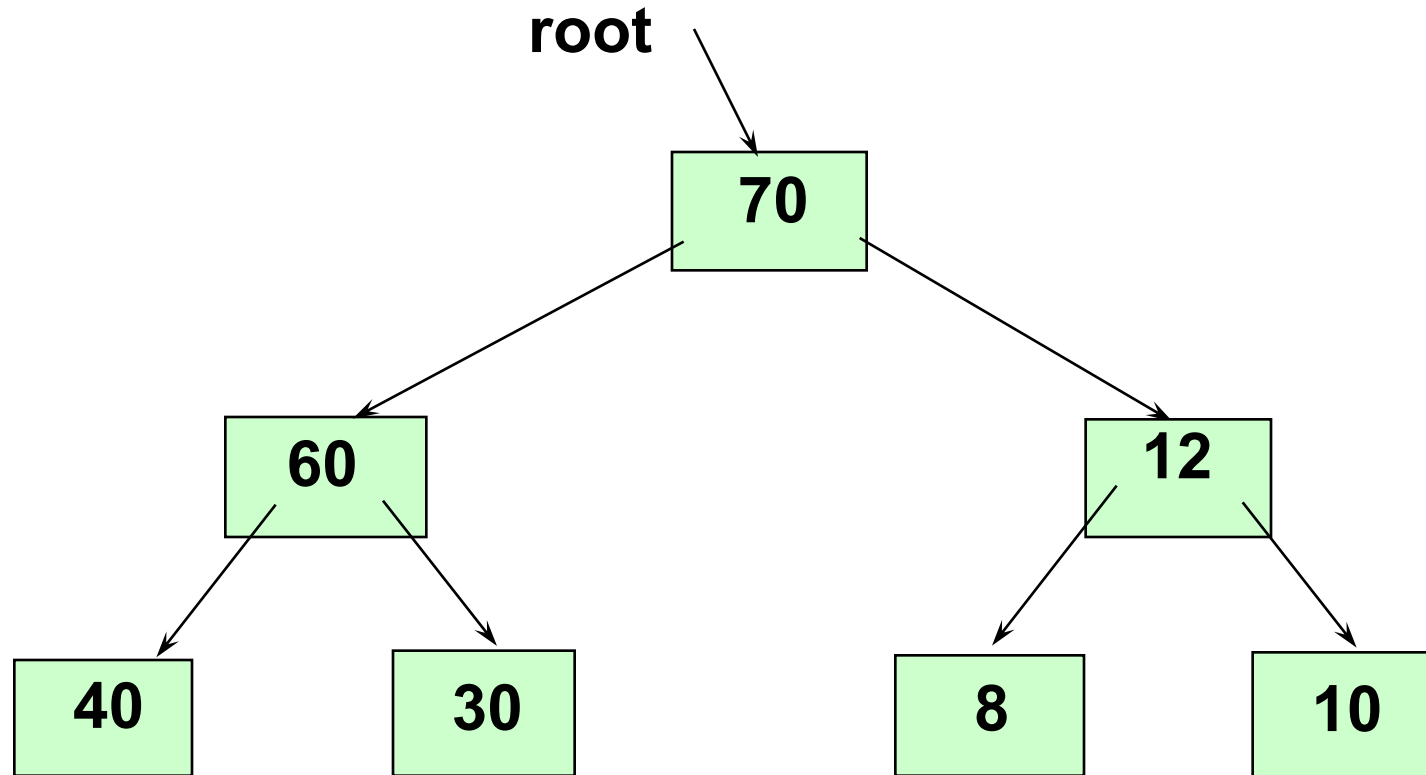


Recall that . . .



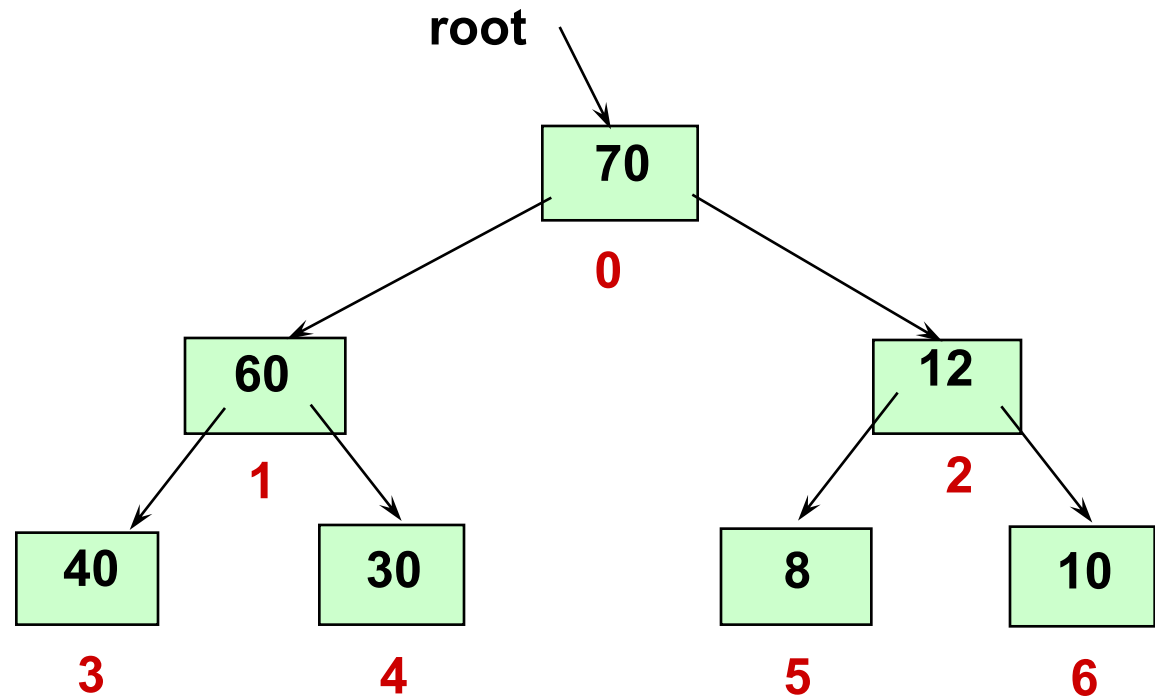
- 힙은 아래의 특수한 Shape 및 Order 속성을 만족하는 이진트리
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:
 - ✓ 모양은 완전 이진트리이어야 함
Its shape must be a complete binary tree.
 - ✓ 힙의 각 노드에 대해 해당노드에 저장된 값이 각 하위의 값보다 크거나 같음
For each node in the heap, the value stored in that node is greater 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

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



Heap Sort Approach



- 먼저, order 속성을 만족시켜 정렬되지 않은 배열을 힙으로 만들며, 정렬되지 않은 요소가 더 이상 없을 때까지 아래 단계를 반복

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.

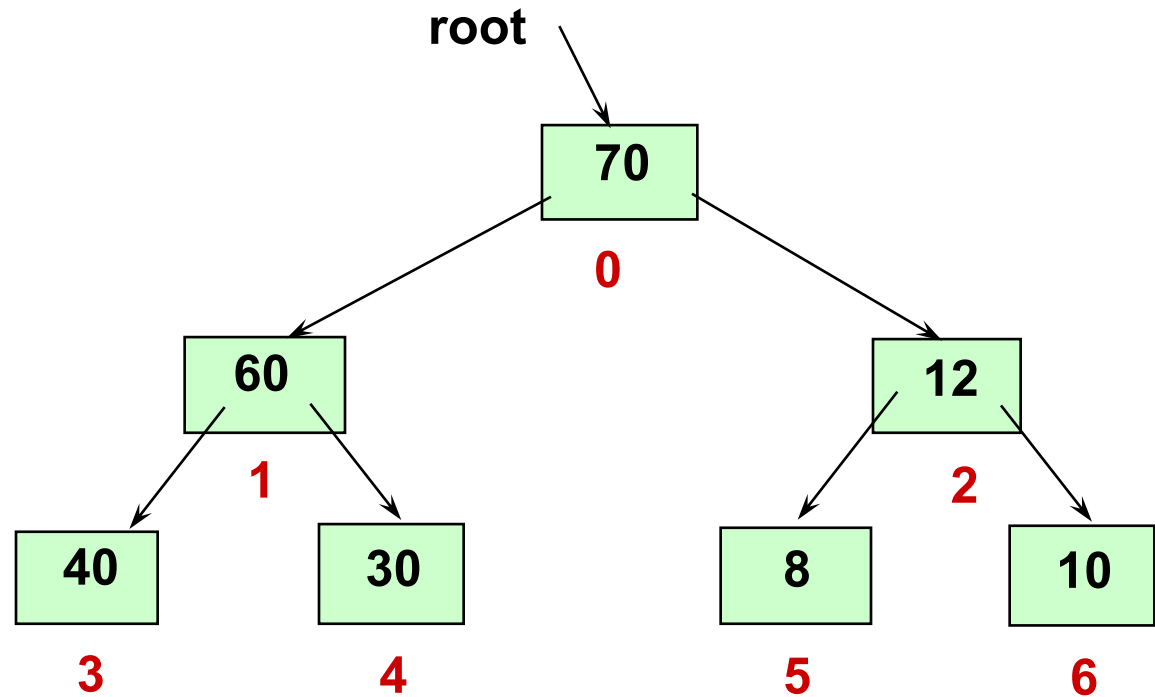
- ✓ 정렬되지 않은 요소의 끝에 배열의 올바른 위치로 바꾸어 힙(최대) 요소를 힙에서 제거

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.

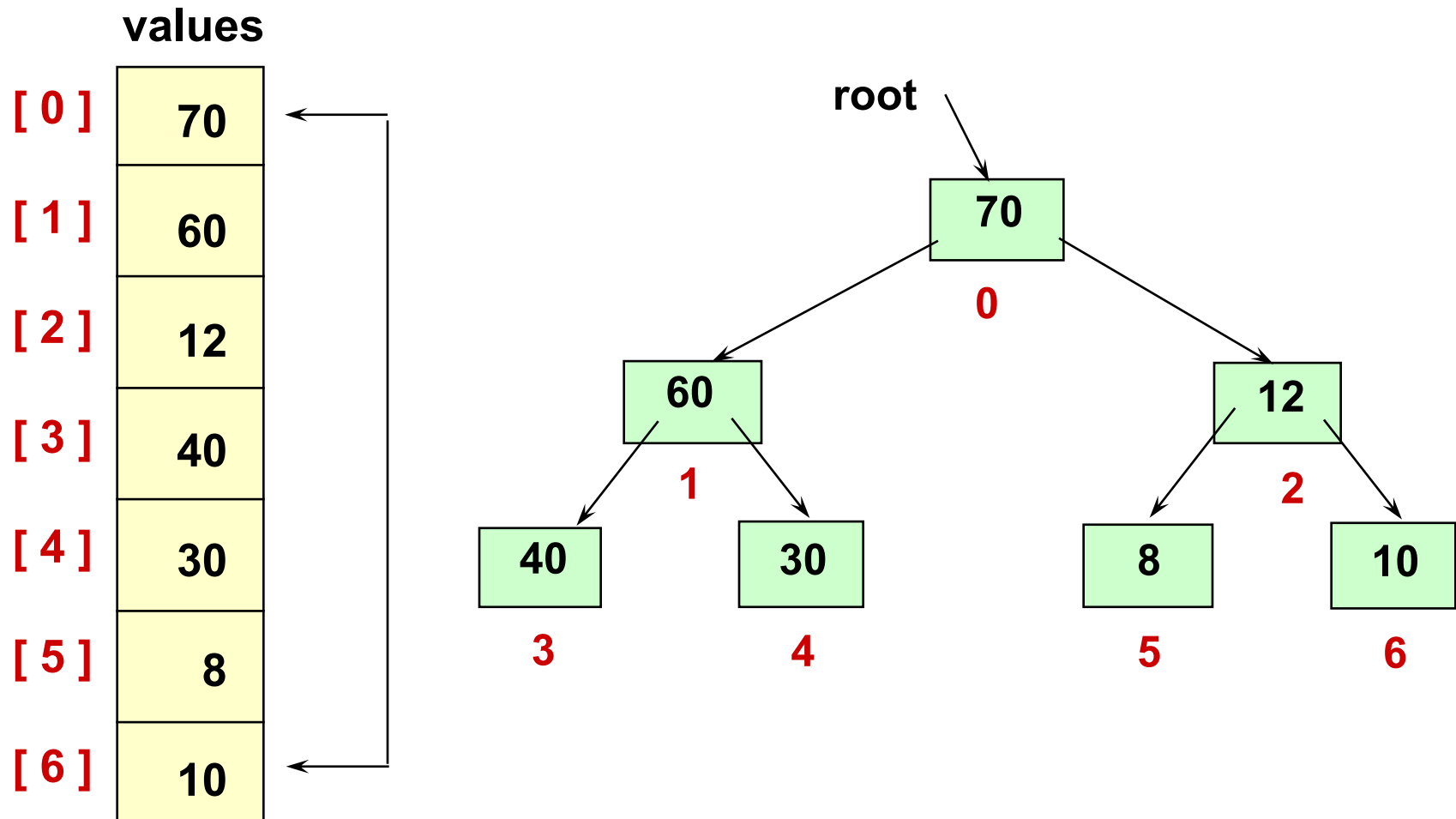
- ✓ 나머지 정렬되지 않은 요소를 리힙 함 (다음으로 큰 요소가 루트 위치에 놓음)
Reheap the remaining unsorted elements. (This puts the next-largest element into the root position).

Heap Sort: Procedure

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

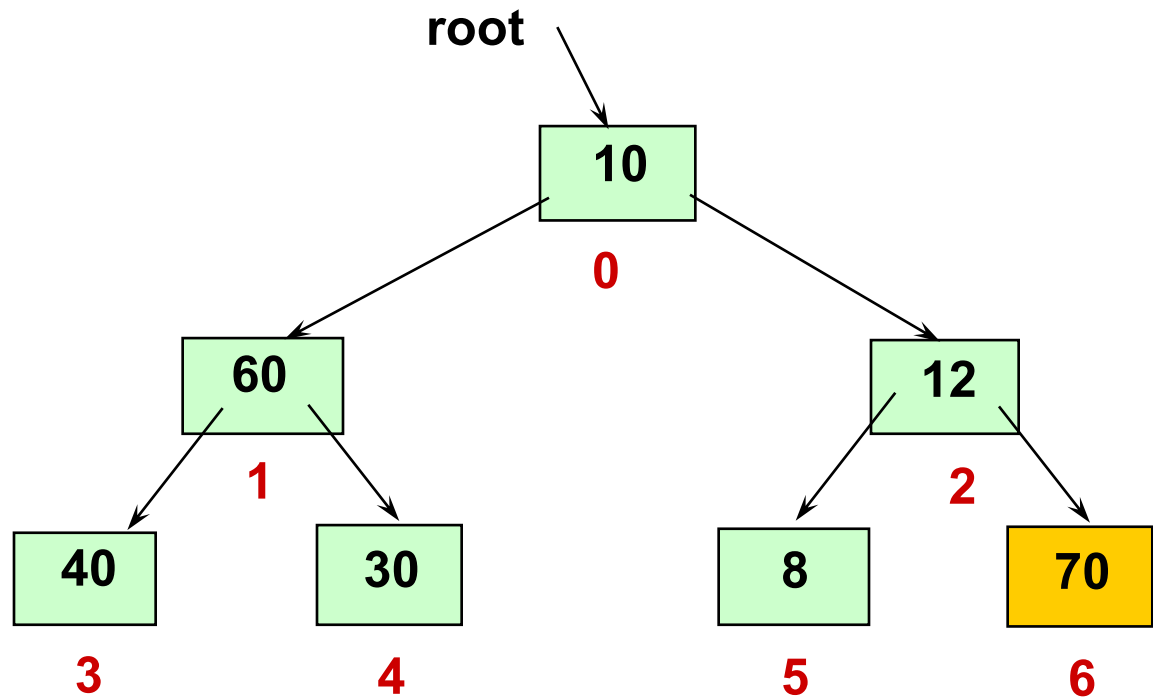


Heap Sort: Procedure



Heap Sort: Procedure

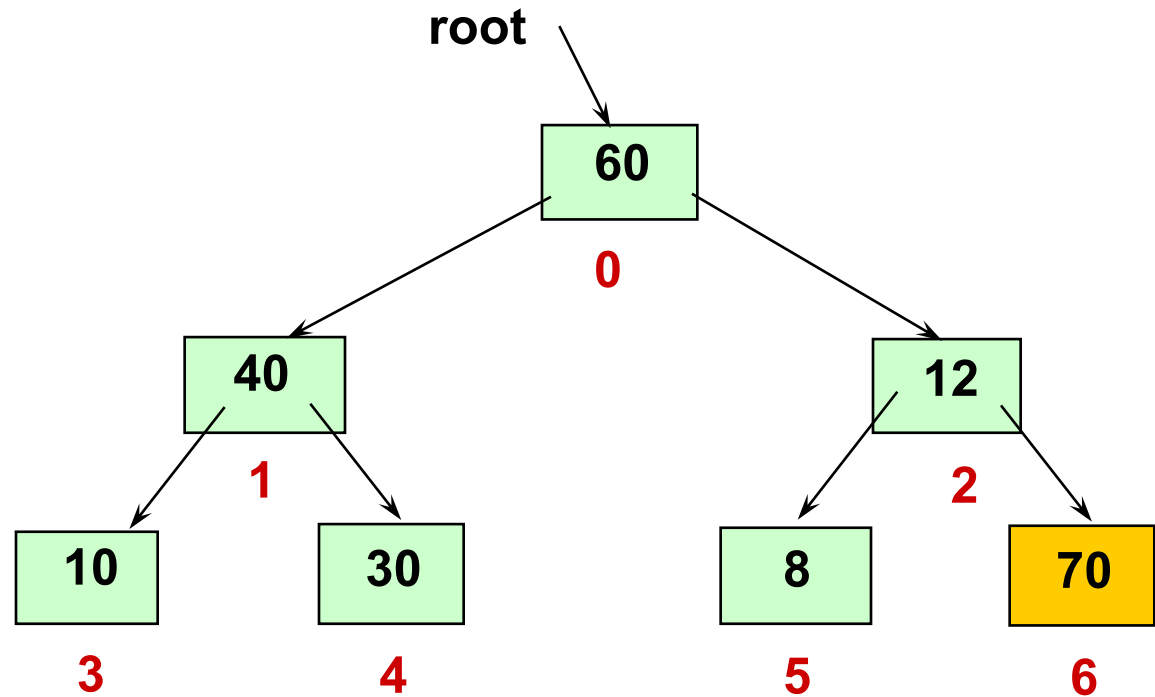
	values
[0]	10
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	70



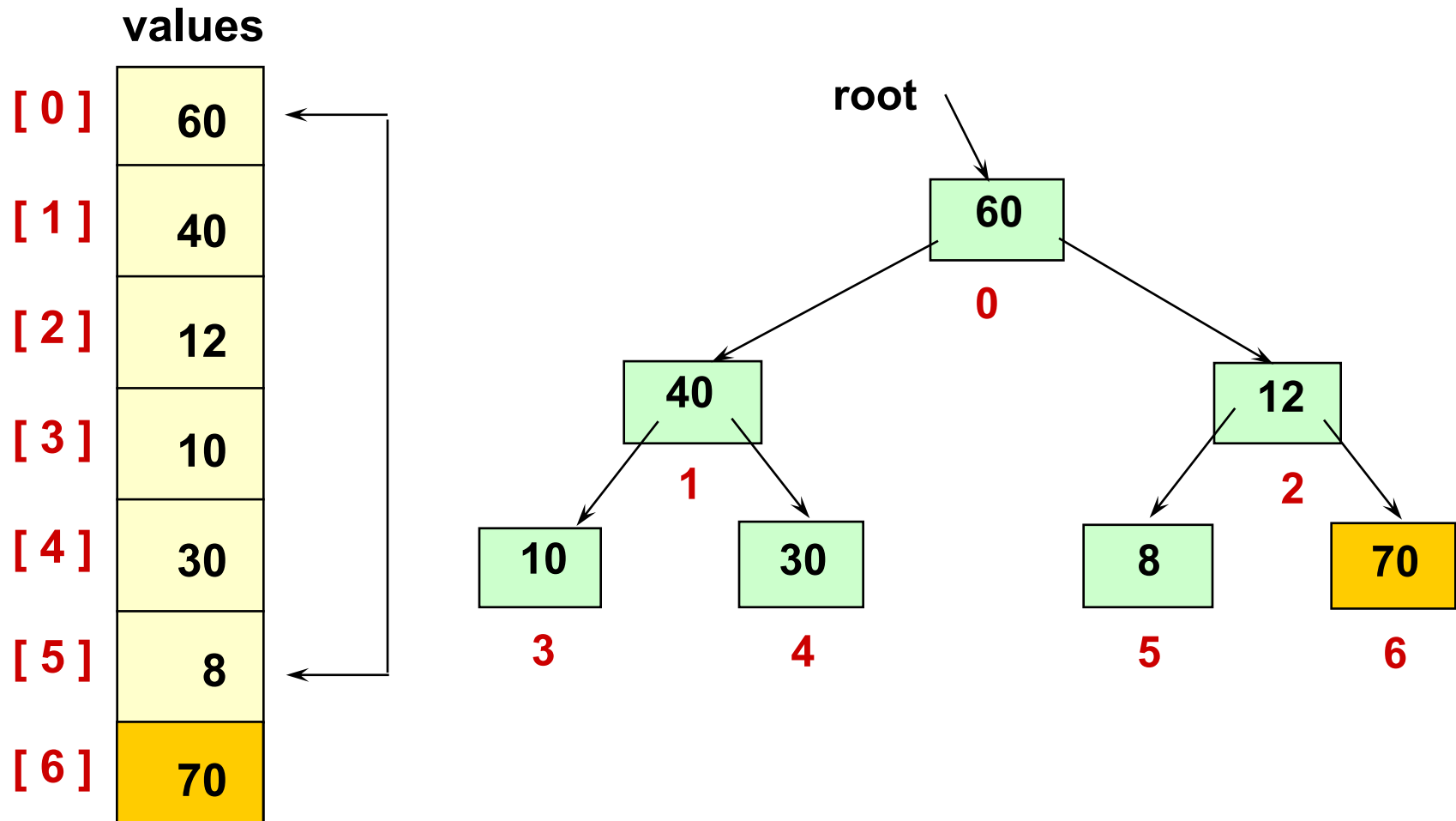
NO NEED TO CONSIDER AGAIN

Heap Sort: Procedure

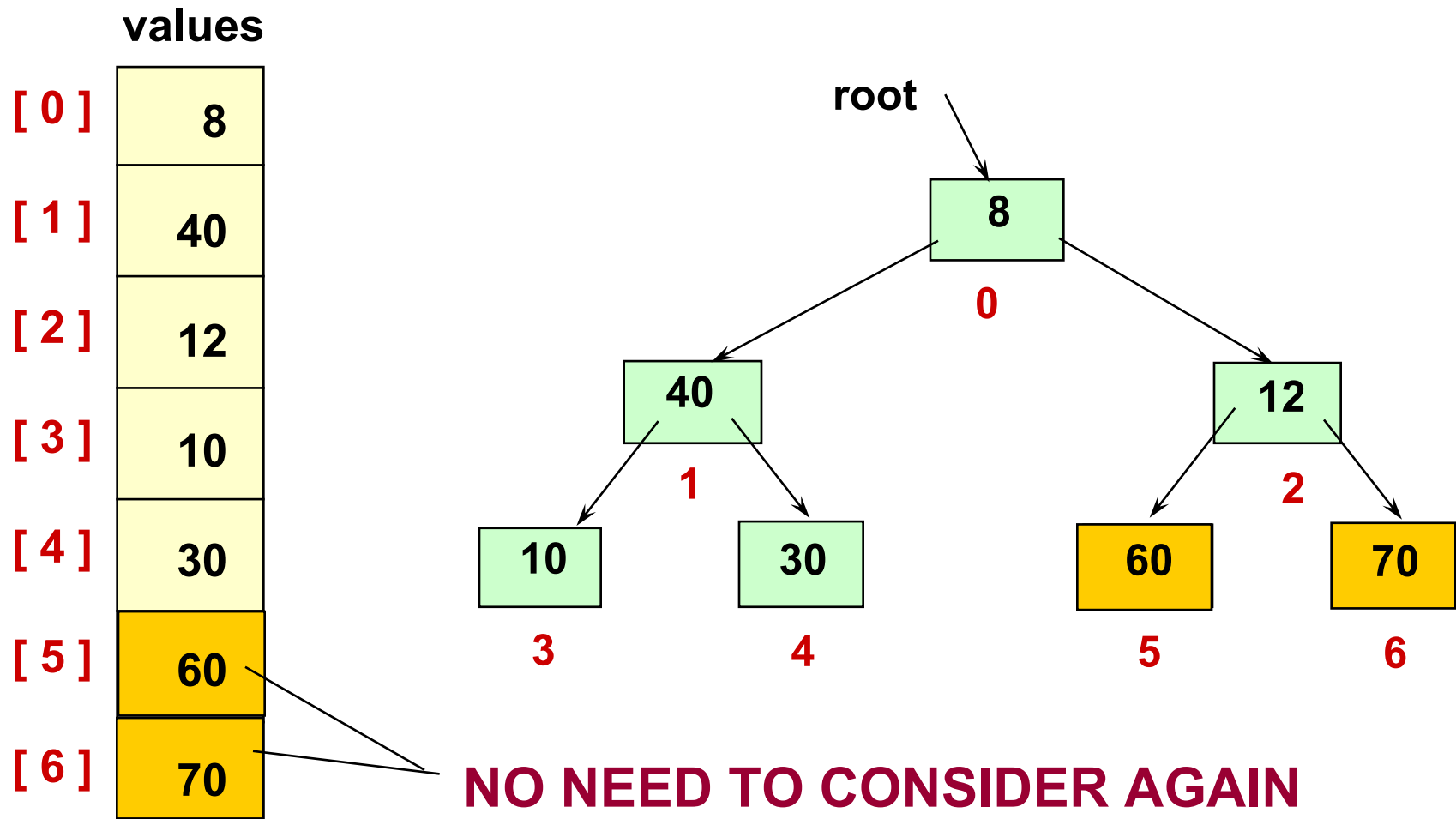
values	
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	8
[6]	70



Heap Sort: Procedure

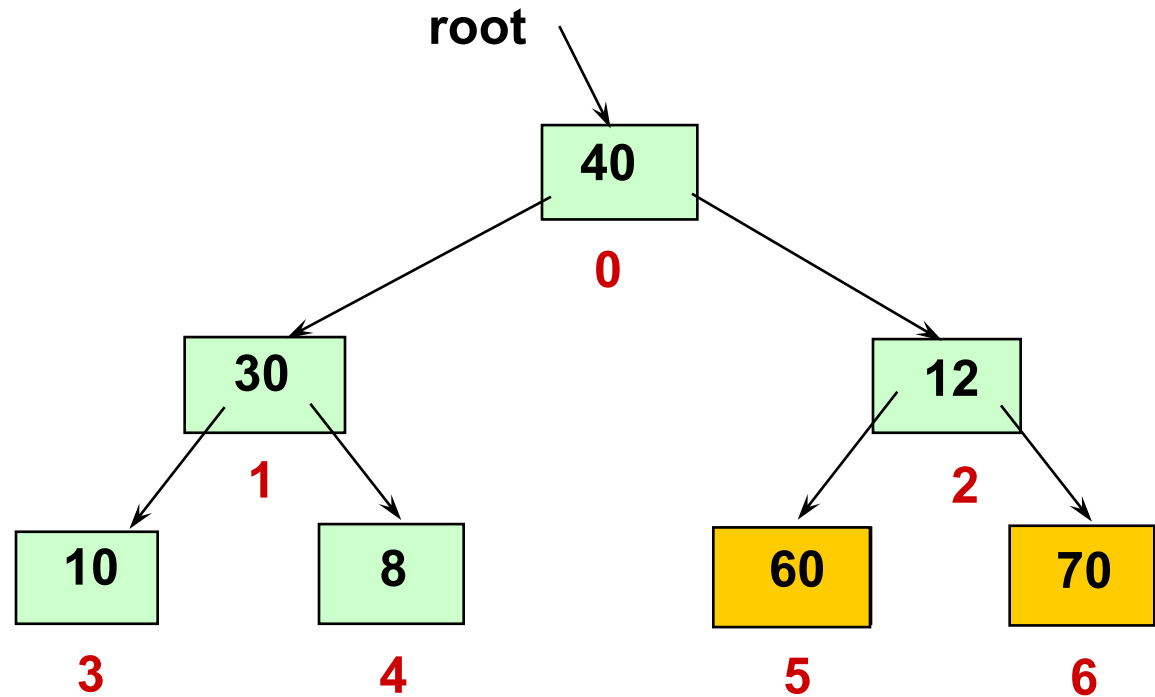


Heap Sort: Procedure

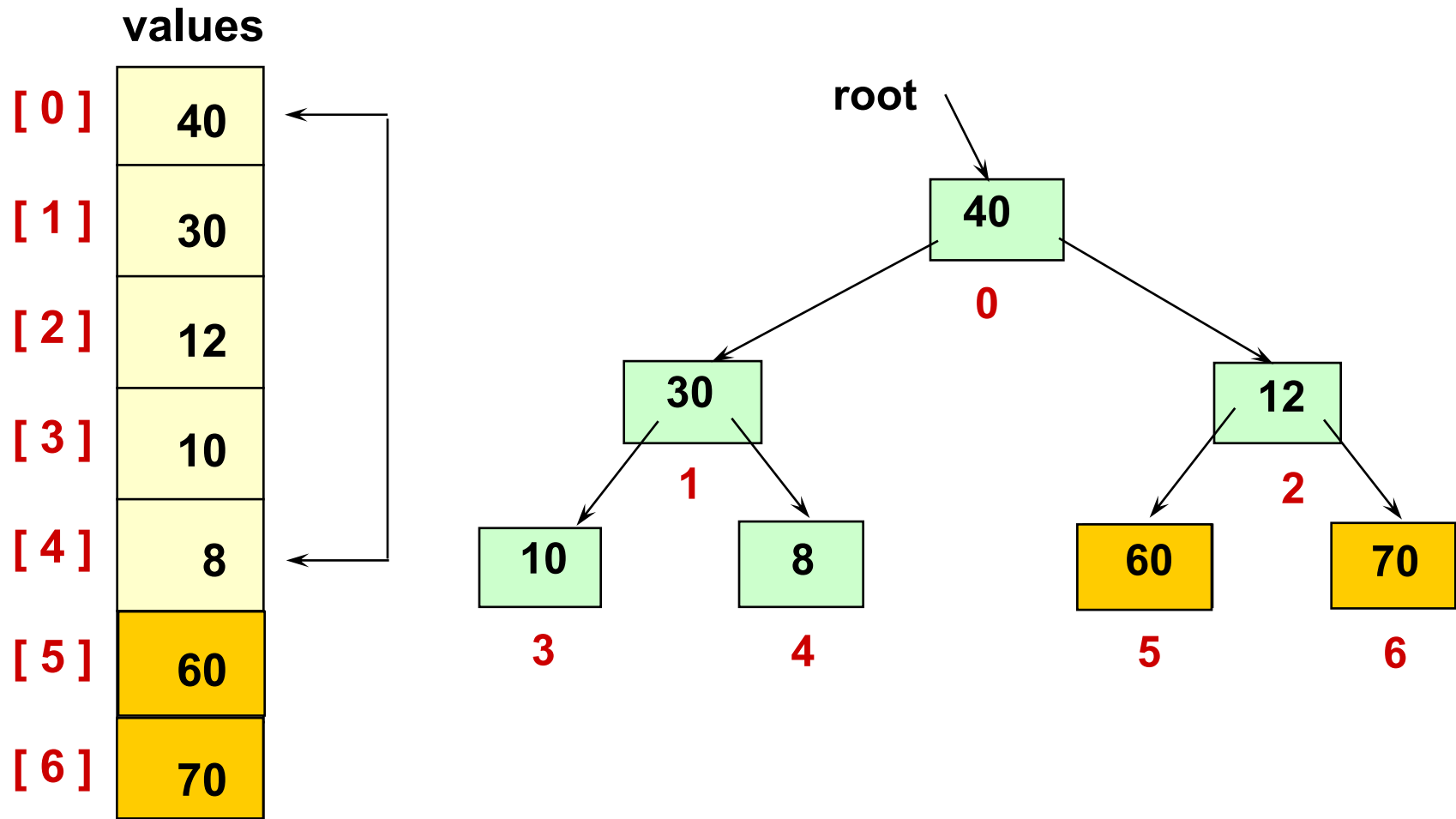


Heap Sort: Procedure

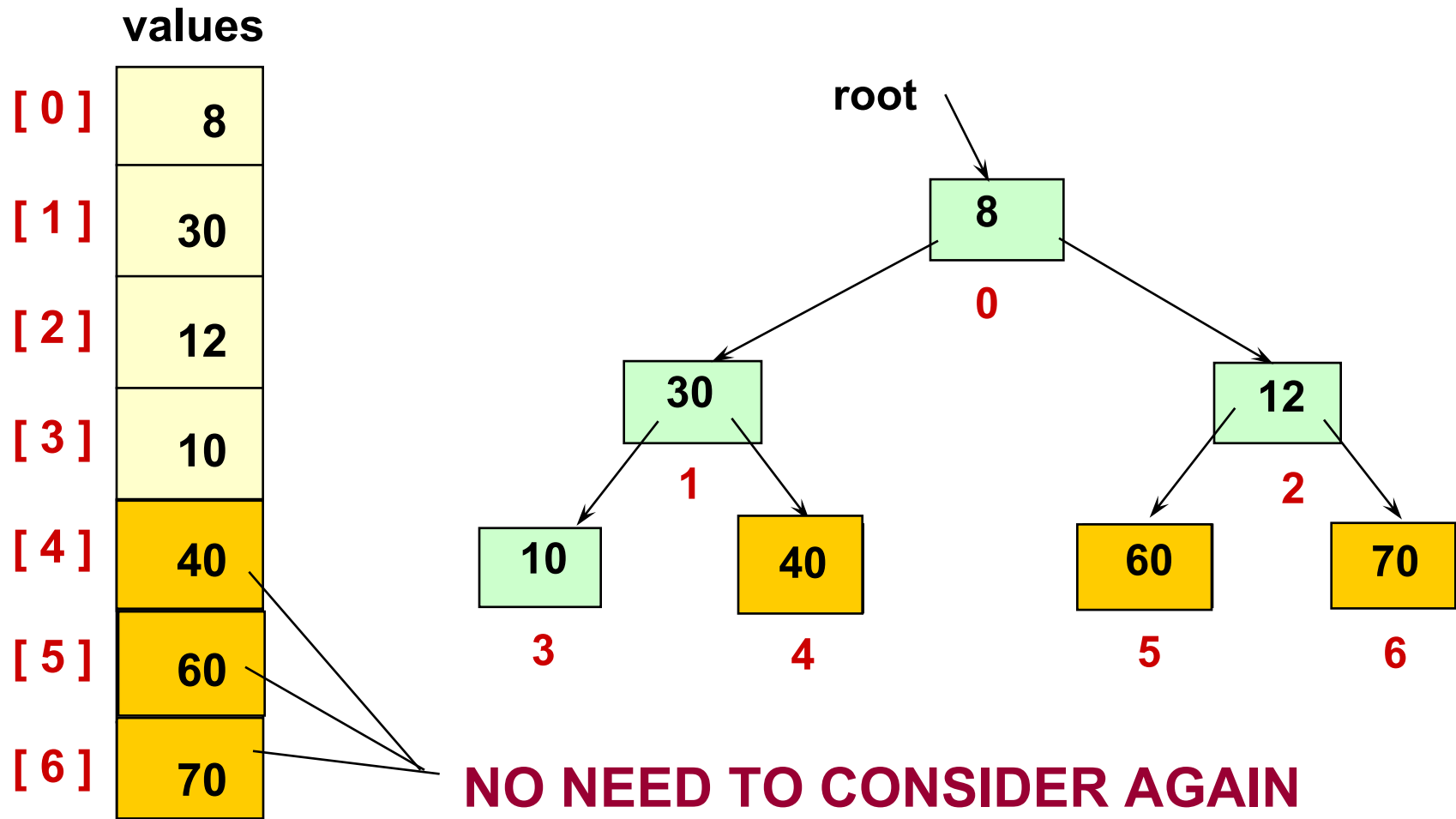
values	
[0]	40
[1]	30
[2]	12
[3]	10
[4]	8
[5]	60
[6]	70



Heap Sort: Procedure

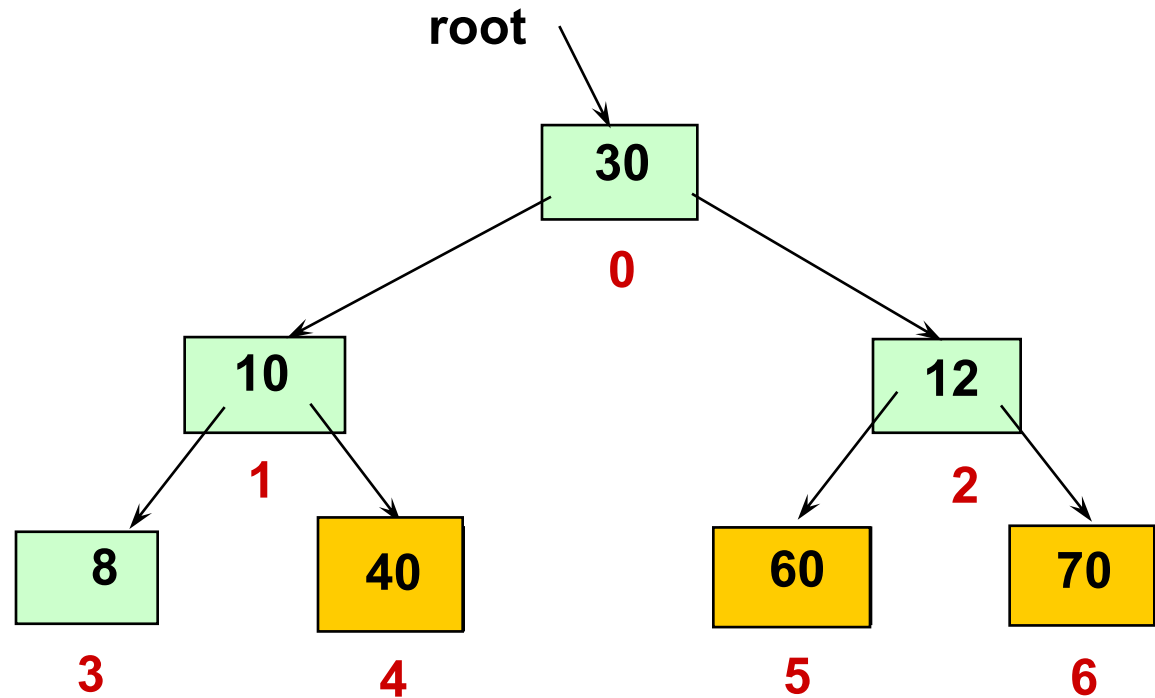


Heap Sort: Procedure



Heap Sort: Procedure

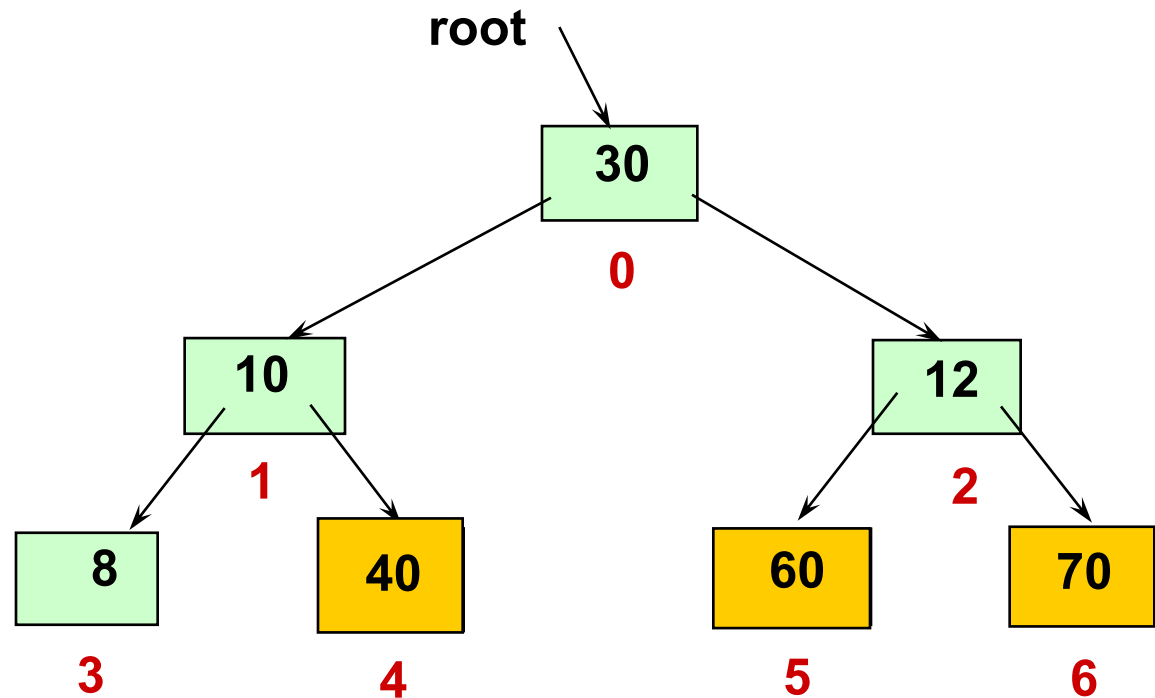
values	
[0]	30
[1]	10
[2]	12
[3]	8
[4]	40
[5]	60
[6]	70



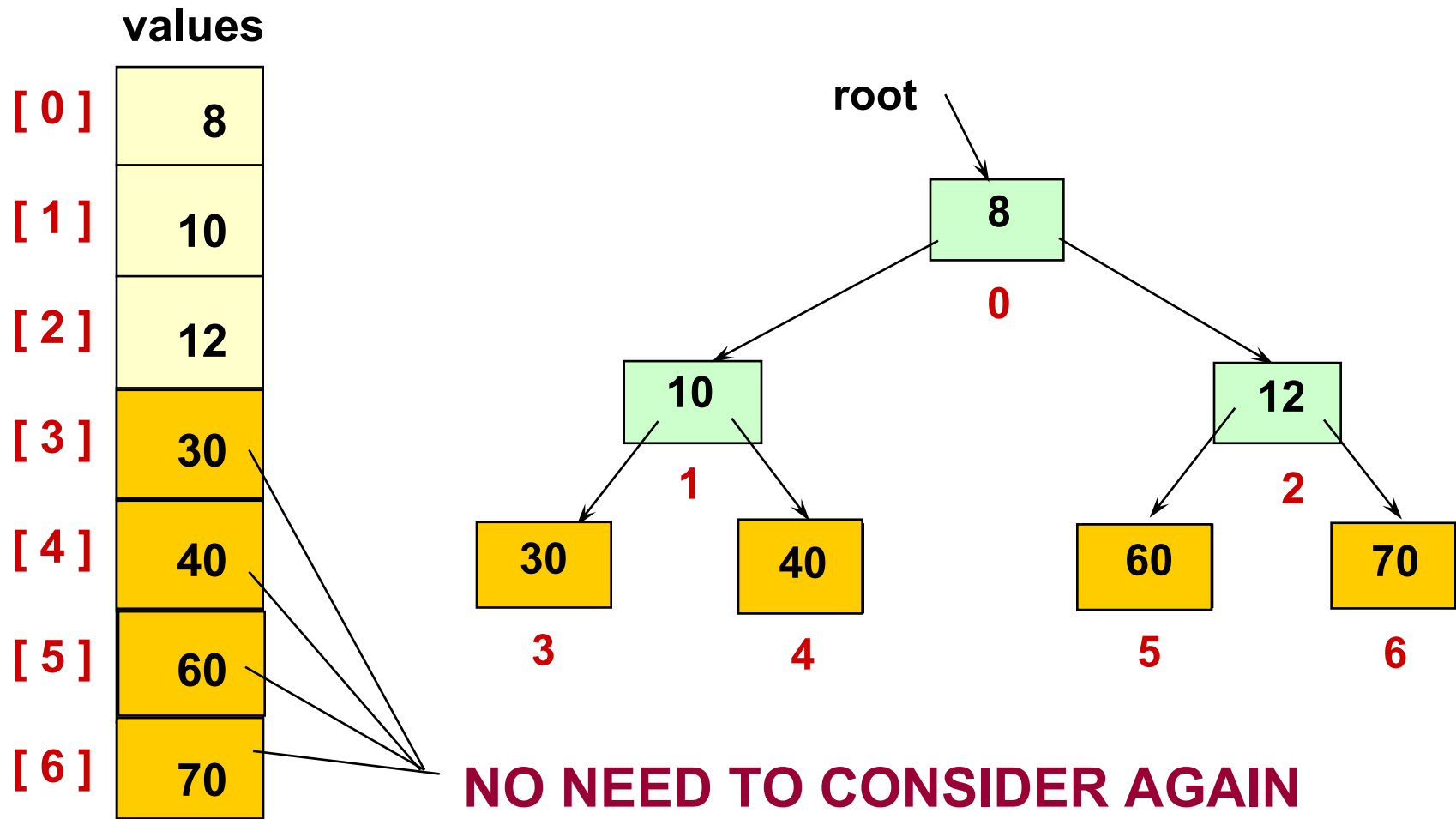
Heap Sort: Procedure

values

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

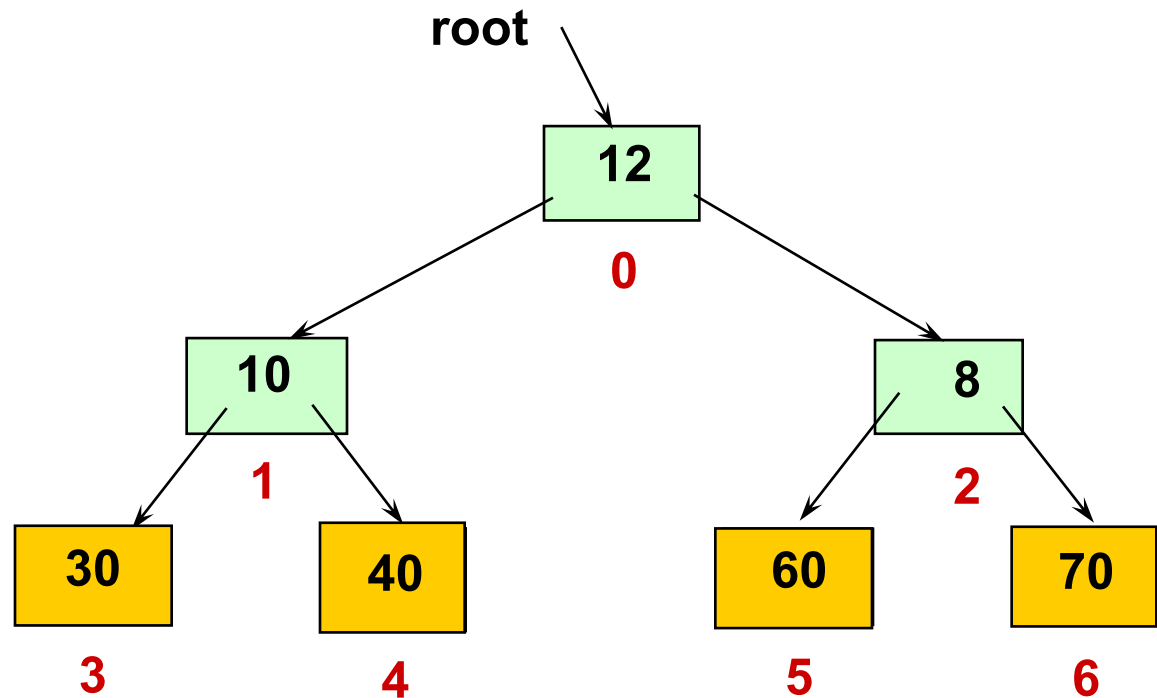


Heap Sort: Procedure



Heap Sort: Procedure

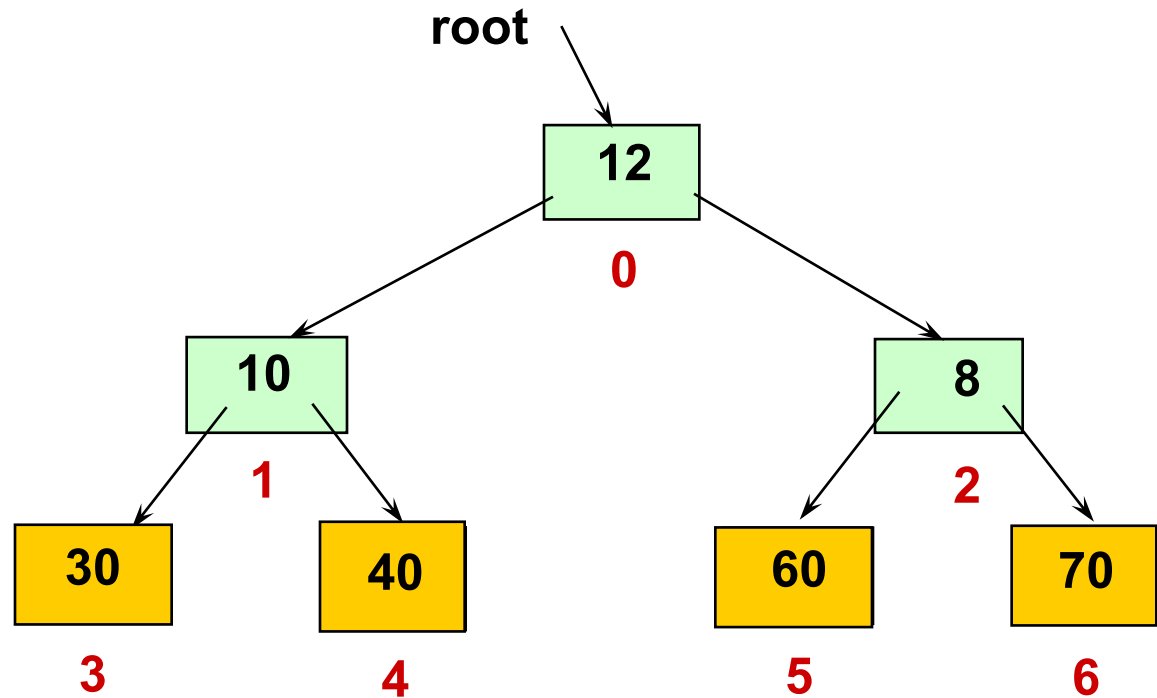

values	
[0]	12
[1]	10
[2]	8
[3]	30
[4]	40
[5]	60
[6]	70



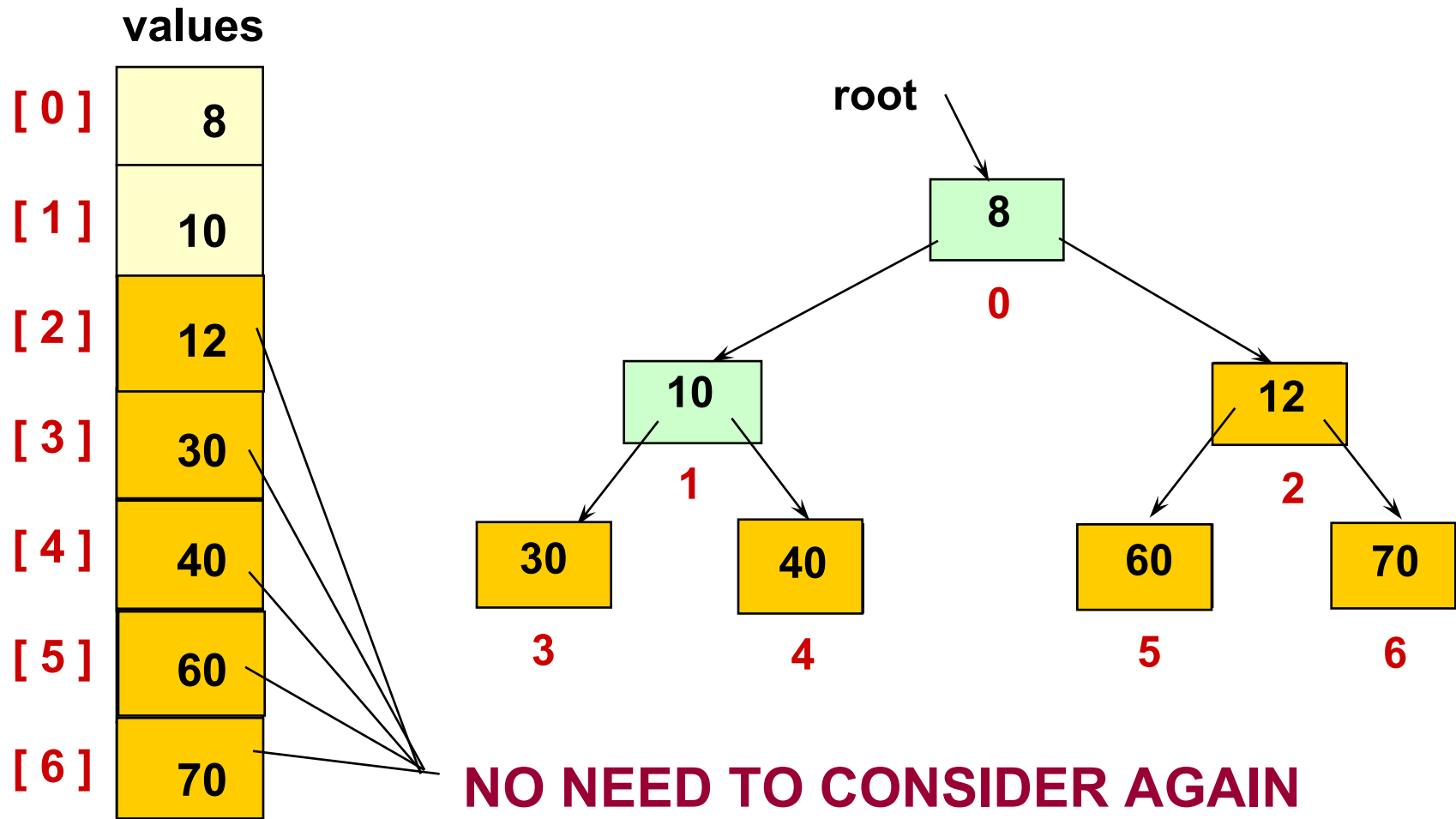
Heap Sort: Procedure

values

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

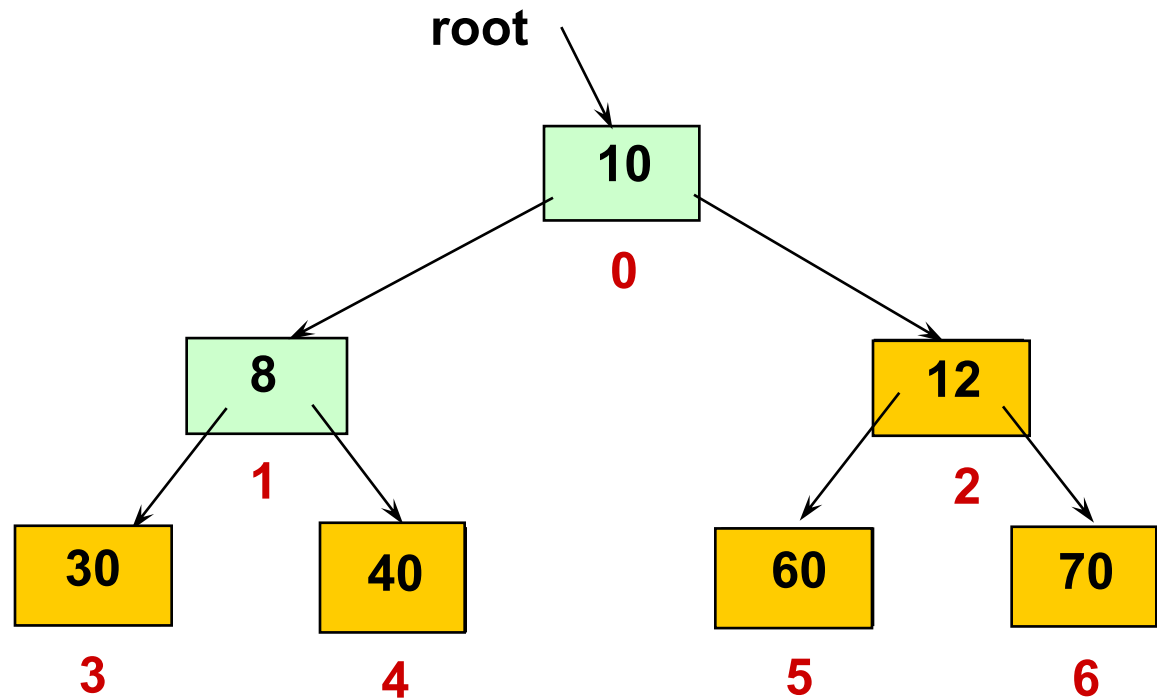


Heap Sort: Procedure



Heap Sort: Procedure

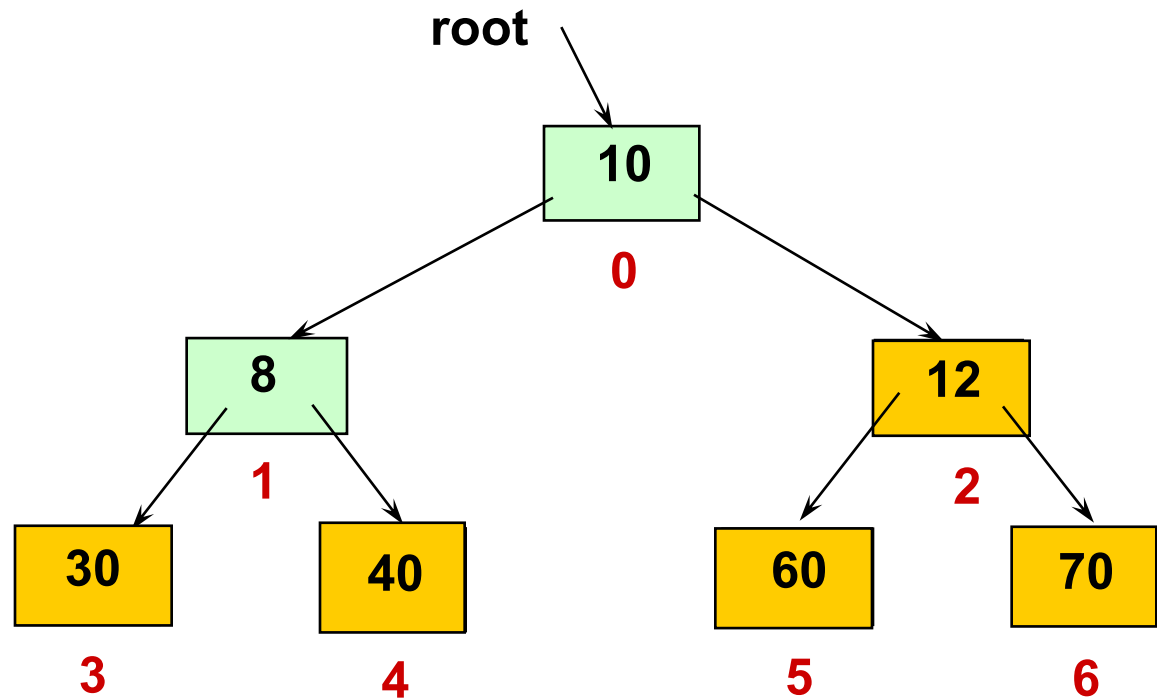
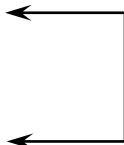
values	
[0]	10
[1]	8
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70



Heap Sort: Procedure

values

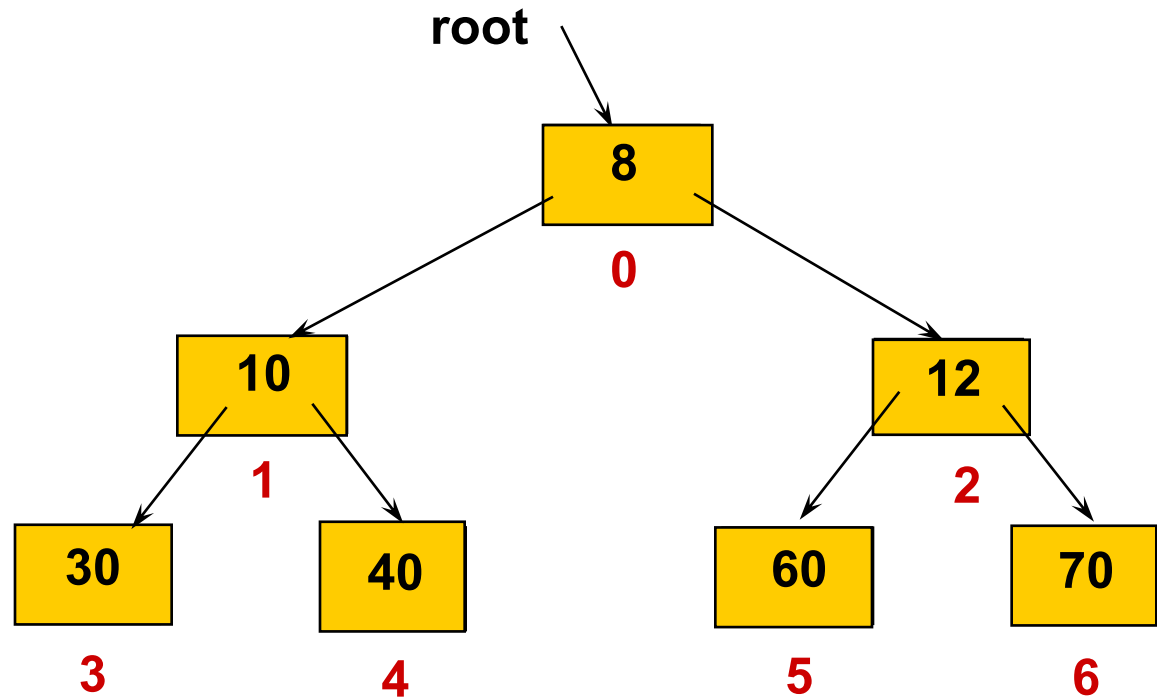
[0]	10
[1]	8
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70



Heap Sort: Procedure



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



ALL ELEMENTS ARE SORTED

Heap Sort: Implementation



```
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 ) ;
    }
}
```

Heap Sort: Implementation



```
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 ;
```


Heap Sort: Implementation



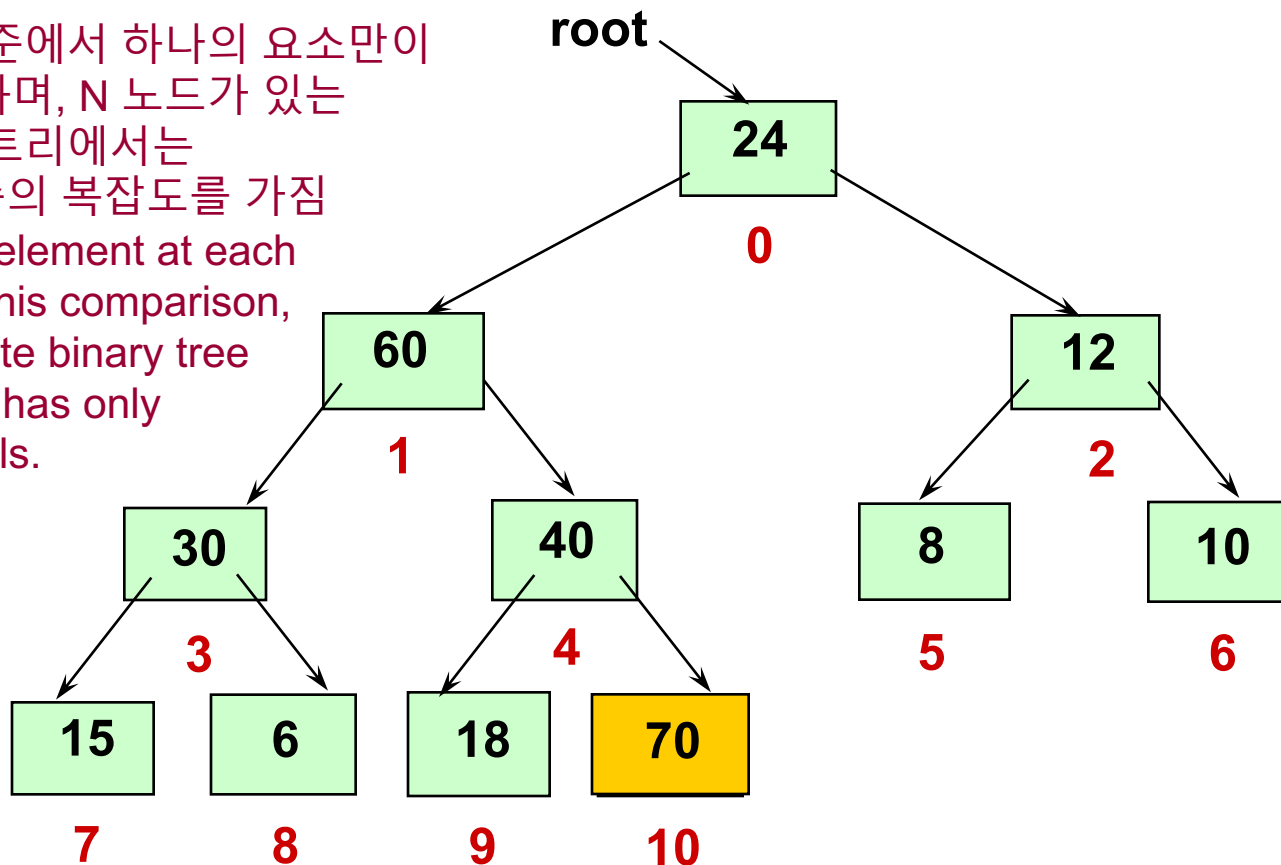
```
if ( leftChild <= bottom )           // ReheapDown continued
{
    if ( leftChild == bottom )
        maxChild = leftChild;
    else
    {
        if ( values [ leftChild ] <= values [ rightChild ] )
            maxChild = rightChild ;
        else
            maxChild = leftChild ;
    }
    if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

Heap Sort: How many comparisons?

- 리힙 (heap down)에서 요소는 2개의 하위 요소와 비교되고, 더 큰 요소와 교체됨

In reheap down, an element is compared with its 2 children (and swapped with the larger).

- ✓ 하지만 각 수준에서 하나의 요소만이 비교를 수행하며, N 노드가 있는 완전한 이진 트리에서는 $O(\log_2 N)$ 수준의 복잡도를 가짐
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?

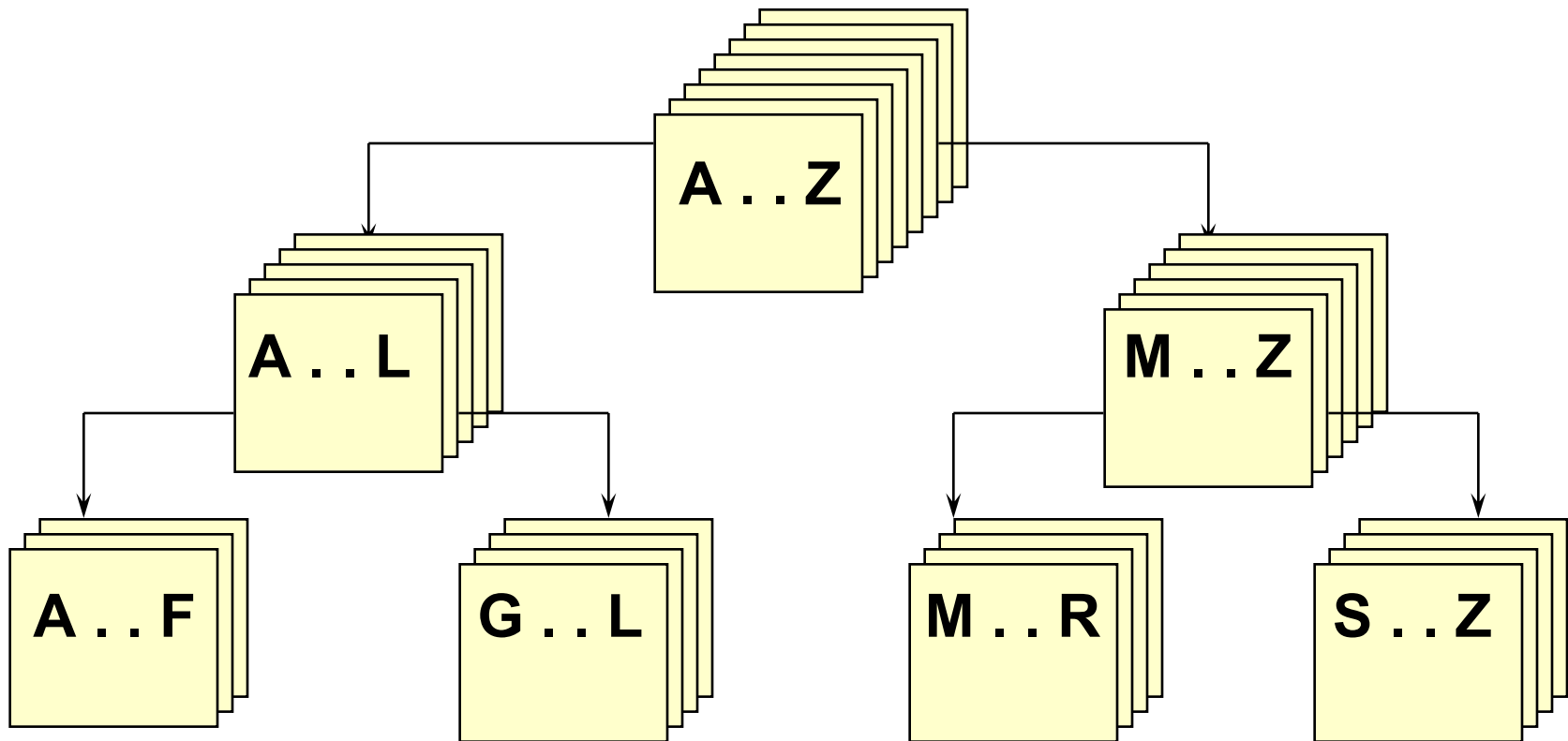


$(N/2) * O(\log N)$ compares to create original heap

$(N-1) * O(\log N)$ compares for the sorting loop

= $O(N * \log N)$ compares total

Using quick sort algorithm



Before call to function Split



splitVal = 9

GOAL: splitVal을 왼쪽에 splitVal 이하의 모든 값과 오른쪽에 더 큰 모든 값으로 위치할 수 있도록 배치
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]

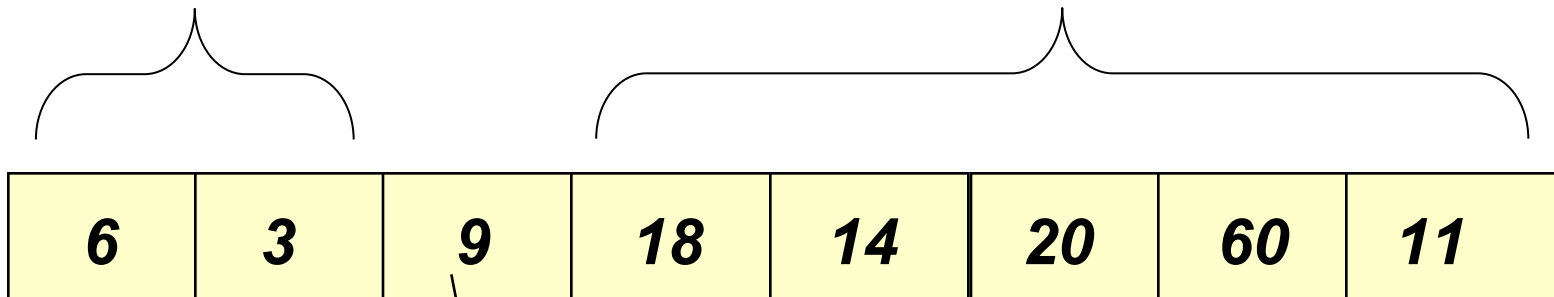
After call to function Split



splitVal = 9

**smaller values
in left part**

**larger values
in right part**



values[first]

[last]

splitVal in correct position

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?

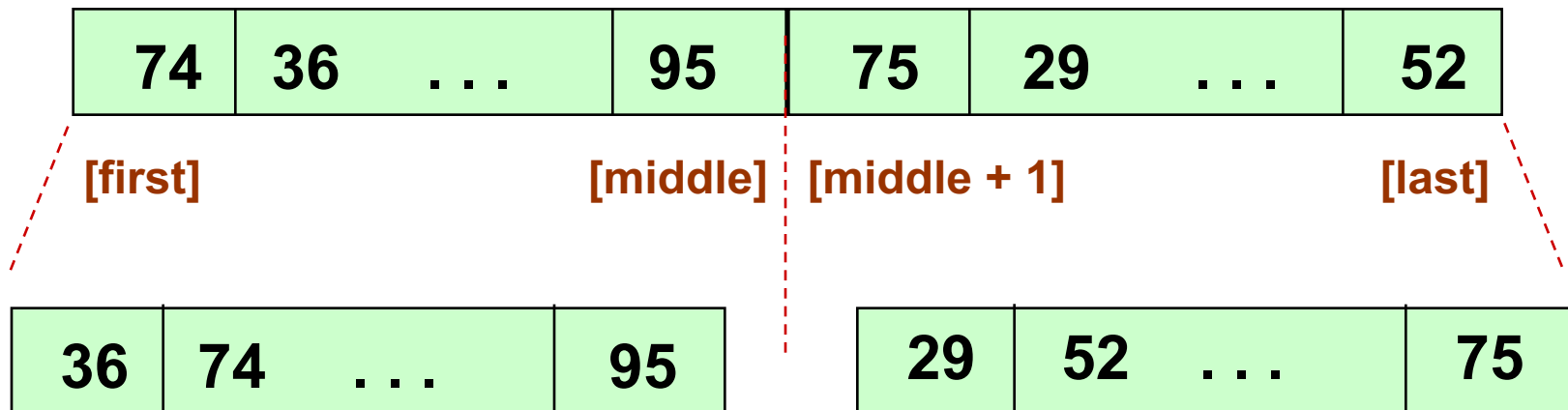
Quick Sort of N elements: How many splits can occur?



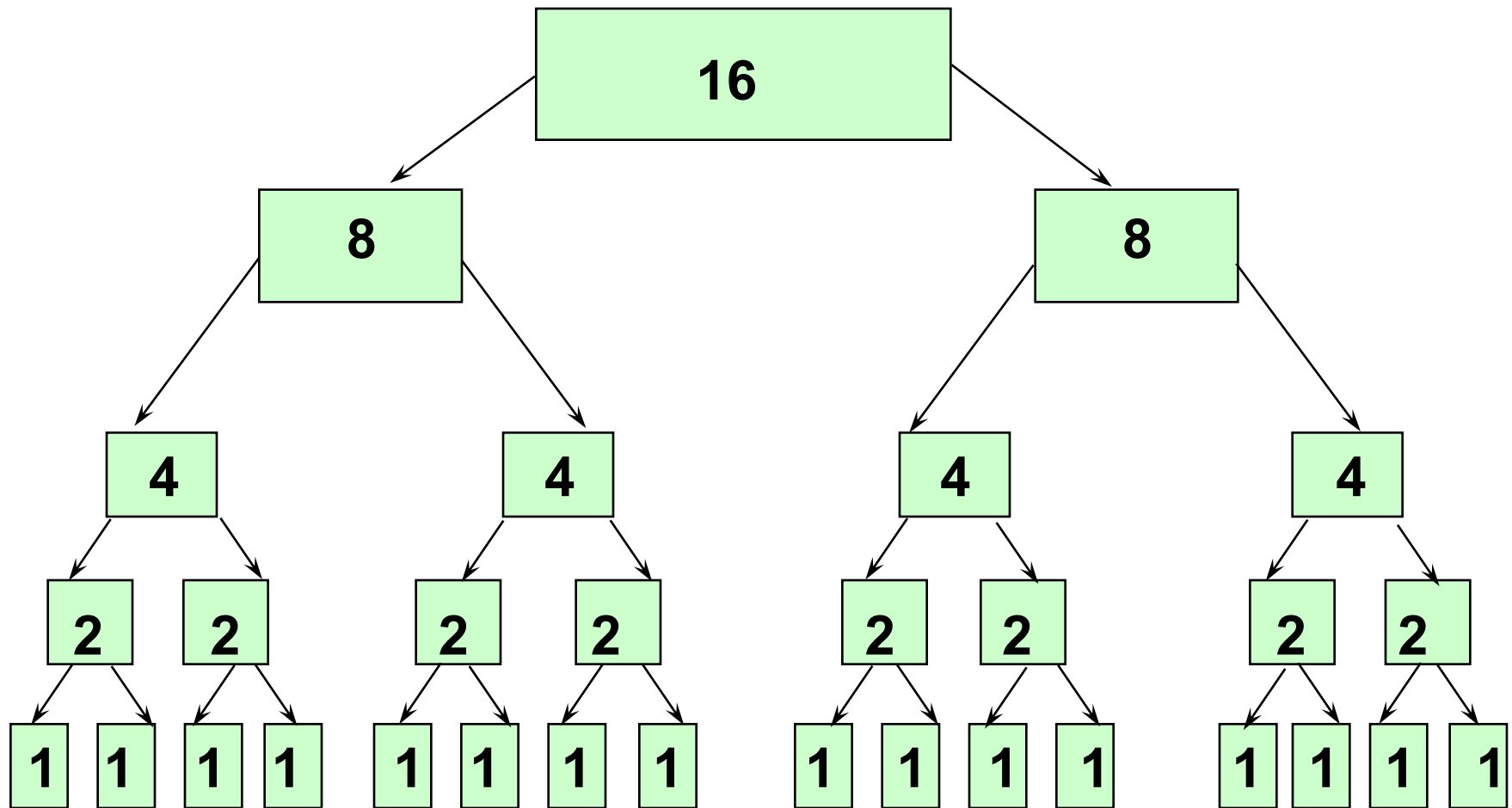
- 원래 배열 요소의 순서에 따라 다름!
It depends on the order of the original array elements!
- 각 분할이 하위 배열을 약 절반으로 나누면 $\log_2 N$ 분할만 일어나며, QuickSort는 $O(N \cdot \log_2 N)$ 의 복잡도를 가짐
If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N \cdot \log_2 N)$.
- 하지만, 원래 배열이 시작되도록 정렬된 경우 재귀 호출은 배열을 길이가 같지 않은 부분으로 분할하고, 한 부분은 비어있고 다른 부분은 분할 값 자체를 제외한 나머지 배열을 모두 포함함
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.
 - ✓ 이 경우 $N-1$ 분할만큼 있을 수 있으며, QuickSort는 $O(N^2)$ 가 됨
In this case, there can be as many as $N-1$ splits, and QuickSort is $O(N^2)$.

Merge Sort Algorithm

- 배열을 반으로 나누고,
Cut the array in half
- 왼쪽을 정렬하고,
Sort the left half
- 오른쪽을 정렬한 후,
Sort the right half
- 두 정렬된 배열을 하나의 정렬된 배열로 병합
Merge the two sorted halves into one sorted array



Using Merge Sort Algorithm with $N = 16$



Merge Sort: Implementation

// 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?



- 전체 배열은 $\log_2 N$ 번만 절반으로 분할
The entire array can be subdivided into halves only $\log_2 N$ times.
- 분할 할 때마다 merge 함수가 호출되어 절반을 다시 결합
Each time it is subdivided, function Merge is called to re-combine the halves
 - ✓ Merge 함수는 임시 배열을 사용하여 병합된 요소를 저장
Function Merge uses a temporary array to store the merged elements.
 - ✓ 병합은 하위 배열의 각 요소를 비교하기 때문에 $O(N)$ 임
Merging is $O(N)$ because it compares each element in the subarrays.
- 임시 배열에서 값을 원 배열로 다시 복사하는 것도 $O(N)$ 임
Copying elements back from the temporary array to the values array is also $O(N)$.
- 따라서 병합정렬은 $O(N \cdot \log_2 N)$ 의 복잡도를 가짐
MERGE SORT IS $O(N \cdot \log_2 N)$.

Function BinarySearch ()



- 이진탐색은 정렬된 배열 정보와 두 개의 첨자 fromLoc 및 toLoc과 항목을 인수로 취함

BinarySearch takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments.

- ✓ info[fromLoc...toLoc] 요소에 항목이 없으면 false를 리턴
It returns false if item is not found in the elements info[fromLoc...toLoc].
- ✓ 그렇지 않으면 true 리턴
Otherwise, it returns true.

- 이진탐색은 $O(\log_2 N)$ 임
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

BinarySearch: Implementation



```
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



- 해싱은 키 값의 함수를 사용하여 목록에서 해당 위치를 식별함으로써 목록의 요소를 빠르게 정렬하고 접근하는데 사용되는 수단 –목표는 $O(1)$ 시간–
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.
- 키 값의 기능을 해시 기능이라 함
The function of the key value is called a hash function.

Using a hash function



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

- HandyParts 회사는 100개 이하의 다른 부품을 생산
HandyParts company makes no more than 100 different parts.
 - ✓ 하지만 부품에는 모두 4자리 숫자가 존재
But the parts all have four digit numbers.
- 이 해시 함수는 배열에 부품을 저장하고 검색하는데 사용될 수 있음
This hash function can be used to store and retrieve parts in an array.
- $\text{Hash}(\text{key}) = \text{partNum} \% 100$

Placing elements in the array



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

- 해시함수 $\text{Hash}(\text{key}) = \text{partNum} \% 100$ 을 사용하여 부품번호 5502인 요소를 배열에 배치
Use the hash function $\text{Hash}(\text{key}) = \text{partNum} \% 100$ to place the element with part number 5502 in the array.

Placing elements in the array



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

- 다음으로 부품번호 6702를 배열에 배치
Next place part number 6702 in the array.

- $\text{Hash}(\text{key}) = \text{partNum} \% 100$

$$6702 \% 100 = 2$$

- 그러나 value[2]는 이미 사용되고 있음
But values[2] is already occupied.
- 충돌 발생!
COLLISION OCCURS

How to resolve the collision?



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

- 한가지 방법은 linear probing임
One way is by linear probing.

✓ 이것은 rehash 기능을 사용
This uses the rehash function

$$(\text{HashValue} + 1) \% 100$$

- 부품번호 6702에 대해 빈 위치가 발견될 때까지 반복
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

- 이 기능을 사용하여 6702의 장소를 여전히 찾아야 함
Still looking for a place for 6702 using the function

$$(\text{HashValue} + 1) \% 100$$

Collision resolved



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

- 부품 6702는 index 4에 배치할 수 있음
Part 6702 can be placed at the location with index 4.

Collision resolved



	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

- HandyParts 회사는 100개 이하의 다른 부품을 생산

HandyParts company makes no more than 100 different parts.

✓ 하지만 부품에는 모두 4자리 숫자가 존재
But the parts all have four digit numbers.

- 이 해시 함수는 배열에 부품을 저장하고 검색하는데 사용될 수 있음

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

- $\text{Hash}(\text{key}) = \text{partNum} \% 100$

Thank You!
Q&A