

Sorting: Equality and Comparison

Dr. Youna Jung
Northeastern University
yo.jung@northeastern.edu



EQUALITY

equals () Method

- The `equals()` method compares the contents of two objects.

- ✓ The default implementation of the `equals()` method in the `Object` class is as follows:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

- ✓ The `equals()` method is overridden in the `Circle` class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius; }  
    else  
        return false; }
```

`==` vs `equals()`

□ The `==` comparison operator

- ✓ for comparing two **primitive data type values** OR
- ✓ for determining whether two **objects have the same references**.

□ `equals()`

- ✓ test whether **two objects have the same contents**, provided that the method is modified in the defining class of the objects.

COMPARISON

Comparable Interface

- Classes that **implement** the **Comparable** interface must **define** a **compareTo()** method
 - Implementing the **Comparable** interface is **an efficient way** to compare objects during a search
 - Method call **obj1.compareTo(obj2)** returns an **integer** with the following values
 - **negative** if **obj1 < obj2**
 - **0** if **obj1 == obj2**
 - **positive** if **obj1 > obj2**

Let us write a recursive binary search with
a **comparable** target in a paper.

```
binarySearch(Object[] items, Comparable target, int first, int last)
```

Implementation of Binary Search

```
/** Recursive binary search method (in RecursiveMethods.java).  
 * @param items The array being searched  
 * @param target The object being searched for  
 * @param first The subscript of the first element  
 * @param last The subscript of the last element  
 * @return The subscript of target if found; otherwise -1.  
 */  
private static int binarySearch(Object[] items, Comparable target,  
                                int first, int last) {  
  
Base Case 1    if (first > last)  
        return -1;      // Base case for unsuccessful search.  
    else {  
        int middle = (first + last) / 2; // Next probe index.  
        int compResult = target.compareTo(items[middle]);  
        if (compResult == 0)  
            return middle; // Base case for successful search.  
        else if (compResult < 0)  
            return binarySearch(items, target, first, middle - 1);  
        else  
            return binarySearch(items, target, middle + 1, last);  
    }  
}
```

Implementation of Binary Search (cont.)

```
/** Wrapper for recursive binary search method (in RecursiveMethods.java).
 * @param items The array being searched
 * @param target The object being searched for
 * @return The subscript of target if found; otherwise -1.
 */
public static int binarySearch(Object[] items, Comparable target) {
    return binarySearch(items, target, 0, items.length - 1);
}
```

SORTING

- Selection Sort
- Insertion Sort
- Shell Sort
- Merge Sort
- Timsort
- Heap Sort
- Quick Sort

Sorting

- Sorting entails **arranging data in order**
- In **Java**,
 - provides the **Arrays** class with several **sort methods**
 - Sorting methods for **arrays of primitive types** are based on the **quicksort** algorithm
 - Sorting methods for **arrays of objects** are based on the **Timsort** algorithm
 - The **Collections** class provides similar **sorting methods** for **Lists**
 - Sorting methods for **Lists** are based on the **Timsort** algorithm
 - **Quicksort, Merge sort and Timsort** are **$O(n \log n)$**

Sort() in java.util.Arrays

Method Arrays.sort	Behavior
public static void sort(int[] items)	Sorts the array items in ascending order
public static void sort(int[] items, int fromIndex, int toIndex)	Sorts array elements items [fromIndex] to items [toIndex] (but not including toIndex) in ascending order
public static void sort(Object[] items)	Sorts the objects in array items in ascending order using their natural ordering (defined by method compareTo()). All objects in items must implement the Comparable interface
public static void sort(Object[] items, int fromIndex, int toIndex)	Sorts objects items[fromIndex] to items[toIndex] in ascending order using their natural ordering (defined by method compareTo). All objects must implement the Comparable interface
public static <T> void sort(T[] items, Comparator<? super T> comp)	Sorts the objects in items in ascending order as defined by method comp.compare . All objects in items must be mutually comparable using method comp.compare
public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp)	Sorts the objects in items[fromIndex] to items[toIndex] in ascending order as defined by method comp.compare . All objects in items must be mutually comparable using method comp.compare

Examples of Arrays . sort

□ Sort an int [] array

```
int[] items = {3, 7, 6, 4, -1}; // declare and initialize items
Arrays.sort(items, 1, 4);           // items becomes {3, 4, 6, 7, -1}
Arrays.sort(items);                // items becomes {-1, 3, 4, 6, 7}
```

□ Sort Integer [] array

```
Integer[] iItems = {-1, -6, 3, 4, -9, 7};
Arrays.sort(iItems); // iItems becomes {-9, -6, -1, 3, 4, 7}
```

□ Sort String [] array using natural ordering

```
String[] names = {"alice", "John"}; // initialize names
Arrays.sort(names); // names becomes {"John", "alice"};
```

ASCII Code for 'J' = 74
ASCII Code for 'a' = 97

Creating a Comparator Class

□ Creating Comparator Class CompareIgnoreCase

```
import java.util.Comparator;
public class CompareIgnoreCase implement Comparator<String> {
    // Compares uppercase equivalents of its arguments
    @Override
    public int compare(String m, String n) {
        return m.toUpperCase().compareTo(n.toUpperCase());
    }
}
```

□ Using Comparator Class CompareIgnoreCase

```
...
CompareIgnoreCase comp = new CompareIgnoreCase();
Arrays.sort(names, comp);
```

Sort() in java.util.Collections

- The **Collections** class provides **sorting** methods for **Lists** similar to the ones for arrays

Method Collections.sort	Behavior
public static <T extends Comparable<T>> void sort(List<T> list)	Sorts the objects in list in ascending order using their natural ordering (defined by method <code>compareTo</code>). All objects in list must implement the Comparable interface and must be mutually comparable
public static <T> void sort (List<T> list, Comparator<? super T> comp)	Sorts the objects in list in ascending order as defined by method comp.compare() . All objects must be mutually comparable

SELECTION SORT

Selection Sort

- A quadratic sort
 - Worst case $O(n^2)$
- It sorts an array by making several passes through the array, selecting a next smallest item in the array each time and placing it where it belongs in the array
 - not limited to arrays
 - All items to be sorted must be Comparable objects

Trace of Selection Sort

0	1	2	3	4
35	65	30	60	20

n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ **do**
2. Set **posMin** to the subscript of a **smallest** item in the **subarray starting at subscript fill**
3. **Exchange** the item at **posMin** with the one at **fill**

n	5
$fill$	
$posMin$	

Trace of Selection Sort (cont.)

0	1	2	3	4
35	65	30	60	20

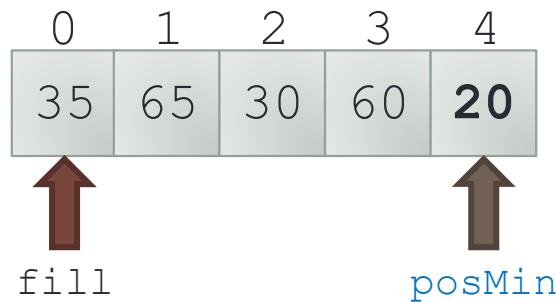
 fill

n = number of elements in the array

- 1. **for** $fill = 0$ to $n - 2$ **do**
- 2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
- 3. Exchange the item at $posMin$ with the one at $fill$

n	5
fill	0
posMin	

Trace of Selection Sort (cont.)

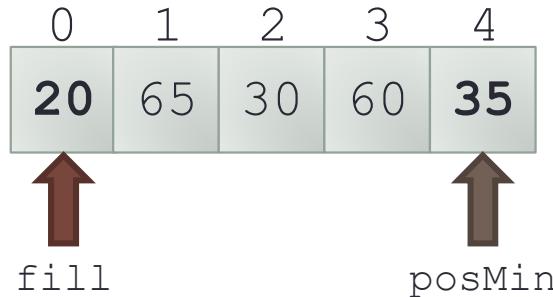


n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ **do**
2. Set **posMin** to the subscript of a smallest item in the **subarray starting** at subscript **fill**
3. Exchange the item at **posMin** with the one at **fill**

n	5
$fill$	0
posMin	4

Trace of Selection Sort (cont.)

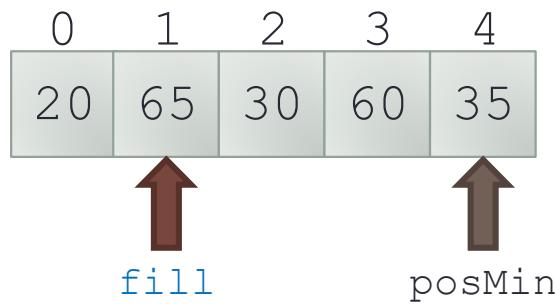


n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ **do**
2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
3. **Exchange** the item at $posMin$ with the one at $fill$

n	5
$fill$	0
$posMin$	4

Trace of Selection Sort (cont.)

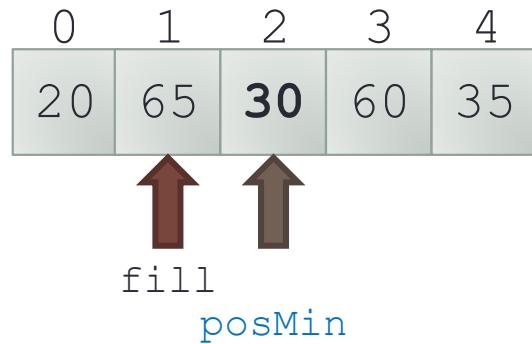


n = number of elements in the array

- 1. **for** $\text{fill} = 0$ to $n - 2$ **do**
- 2. Set posMin to the subscript of a smallest item in the subarray starting at subscript fill
- 3. Exchange the item at posMin with the one at fill

n	5
fill	1
posMin	4

Trace of Selection Sort (cont.)

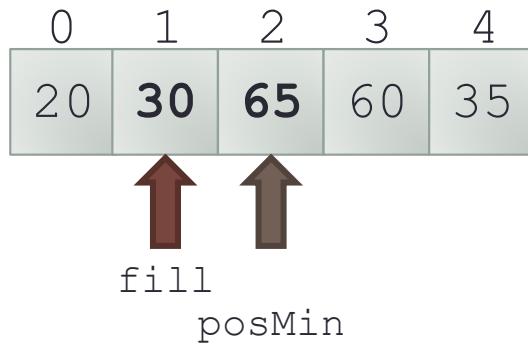


n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ **do**
2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
3. Exchange the item at $posMin$ with the one at $fill$

n	5
$fill$	1
$posMin$	2

Trace of Selection Sort (cont.)

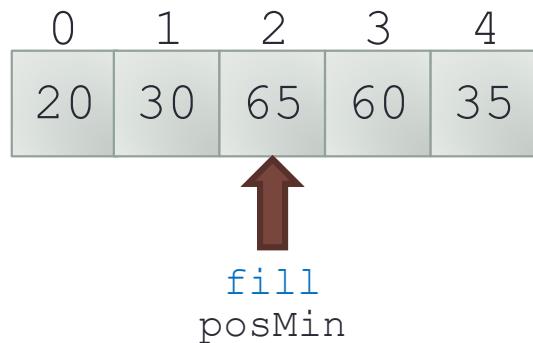


n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ do
2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
3. **Exchange** the item at $posMin$ with the one at $fill$

n	5
$fill$	1
$posMin$	2

Trace of Selection Sort (cont.)

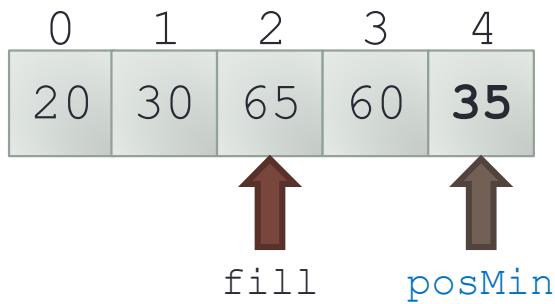


n = number of elements in the array

- 1. **for** fill = 0 to n - 2 do
- 2. Set posMin to the subscript of a smallest item in the subarray starting at subscript fill
- 3. Exchange the item at posMin with the one at fill

n	5
fill	2
posMin	2

Trace of Selection Sort (cont.)

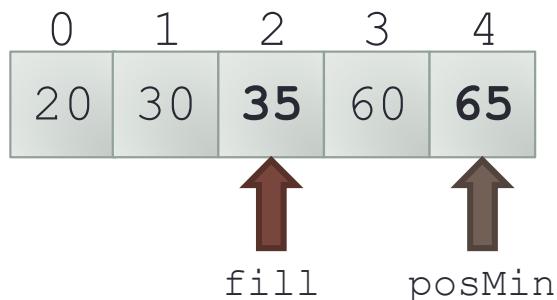


n = number of elements in the array

- 1. **for** $fill = 0$ to $n - 2$ **do**
- 2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
- 3. Exchange the item at $posMin$ with the one at $fill$

n	5
$fill$	2
$posMin$	4

Trace of Selection Sort (cont.)

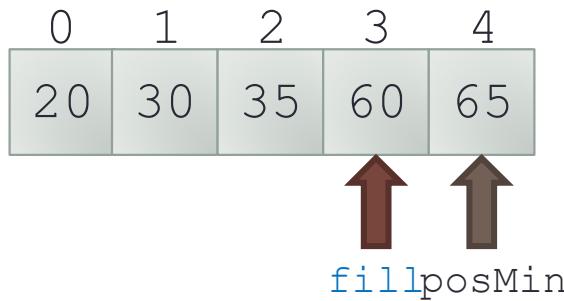


n = number of elements in the array

1. **for** $fill = 0$ to $n - 2$ **do**
2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
3. **Exchange** the item at $posMin$ with the one at $fill$

n	5
$fill$	2
$posMin$	4

Trace of Selection Sort (cont.)

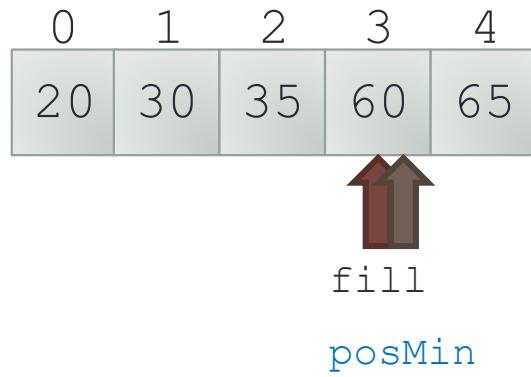


n = number of elements in the array

- 1. **for** $fill = 0$ **to** $n - 2$ **do**
- 2. Set $posMin$ to the subscript of a smallest item in the subarray starting at subscript $fill$
- 3. Exchange the item at $posMin$ with the one at $fill$

n	5
$fill$	3
$posMin$	4

Trace of Selection Sort (cont.)

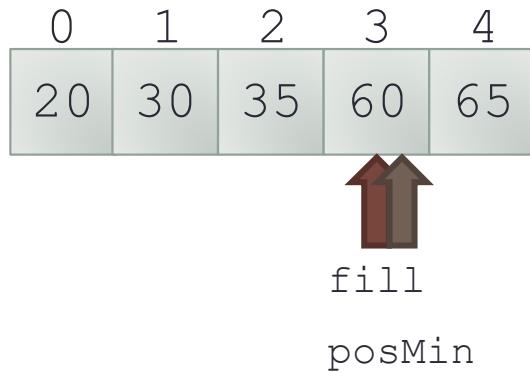


n = number of elements in the array

- 1. **for** $fill = 0$ to $n - 2$ **do**
- 2. Set **posMin** to the subscript of a smallest item in the subarray starting at subscript $fill$
- 3. Exchange the item at **posMin** with the one at **fill**

n	5
$fill$	3
posMin	3

Trace of Selection Sort (cont.)



n = number of elements in the array

1. **for** $\text{fill} = 0$ to $n - 2$ **do**
2. Set posMin to the subscript of a smallest item in the subarray starting at subscript fill
3. **Exchange** the item at posMin with the one at fill

n	5
fill	3
posMin	3

Trace of Selection Sort (cont.)

0	1	2	3	4
20	30	35	60	65

n = number of elements in the array

1. **for** fill = 0 to $n - 2$ do
2. Set posMin to the subscript of a smallest item in the subarray starting at subscript fill
3. Exchange the item at posMin with the one at fill

n	5
fill	3
posMin	3

Trace of Selection Sort Refinement

n	5
fill	
posMin	
next	

0	1	2	3	4
35	65	30	60	20

1. **for** fill = 0 to n - 2 **do**
2. **Initialize posMin** to **fill**
3. **for** next = fill + 1 to n - 1
 do
4. **if** the item at **next** is less than
 the item at **posMin**
5. **Reset posMin** to **next**
6. **Exchange** the item at **posMin** with the
 one at **fill**

Trace of Selection Sort Refinement (cont.)

n	5
fill	0
posMin	
next	

0	1	2	3	4
35	65	30	60	20


fill

- 1. **for** `fill` = 0 to $n - 2$ **do**
- 2. Initialize `posMin` to `fill`
- 3. **for** `next` = `fill` + 1 to $n - 1$
 do
- 4. **if** the item at `next` is less than
 the item at `posMin`
- 5. Reset `posMin` to `next`
- 6. Exchange the item at `posMin` with the
 one at `fill`

Trace of Selection Sort Refinement (cont.)

n	5
fill	0
posMin	0
next	

0	1	2	3	4
35	65	30	60	20



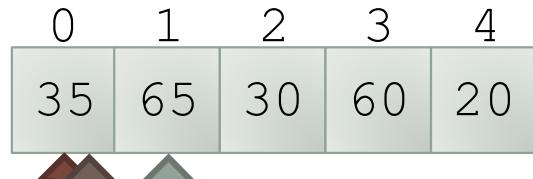
fill

posMin

- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement (cont.)

n	5
fill	0
posMin	0
next	1



fill next
posMin

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. if the item at **next** is less than
 the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill



Trace of Selection Sort Refinement

n	5
fill	0
posMin	0
next	1

0	1	2	3	4
35	65	30	60	20

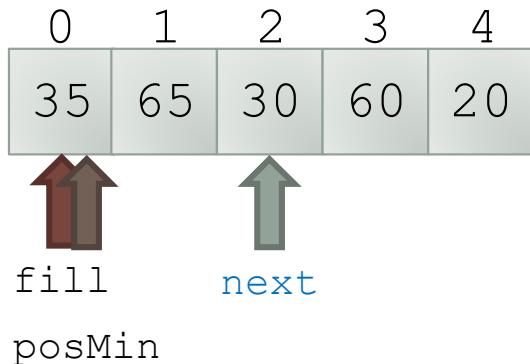

fill next
posMin

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4. **if** the item at **next** is less than the item at **posMin**
5. Reset **posMin** to **next**
6. Exchange the item at **posMin** with the one at **fill**



Trace of Selection Sort Refinement

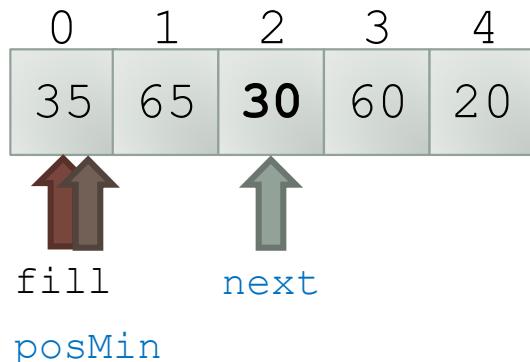
n	5
fill	0
posMin	0
next	2



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

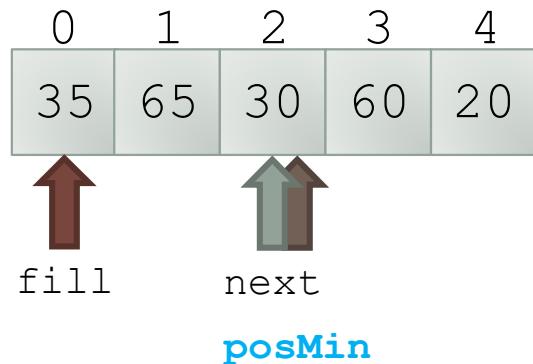
n	5
fill	0
posMin	0
next	2



-
1. **for** fill = 0 to n - 2 do
 2. Initialize posMin to fill
 3. **for** next = fill + 1 to n - 1
do
 4. **if** the item at **next** is less than
 the item at **posMin**
 5. Reset posMin to next
 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

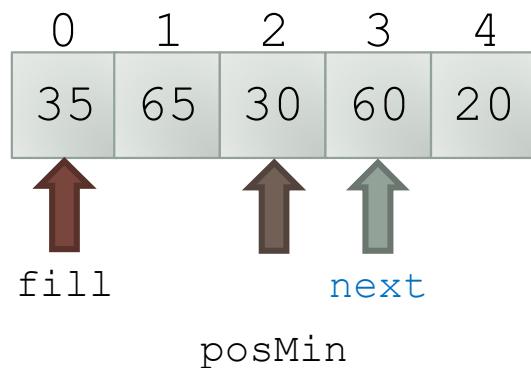
n	5
fill	0
posMin	2
next	2



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

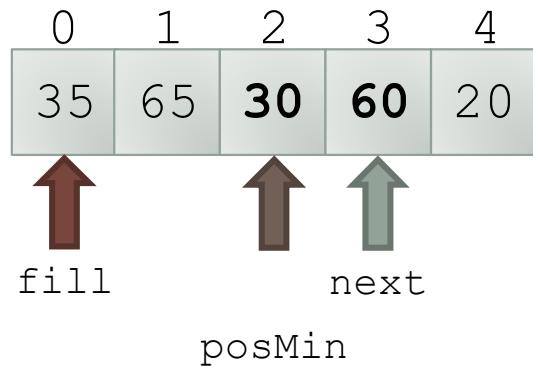
n	5
fill	0
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

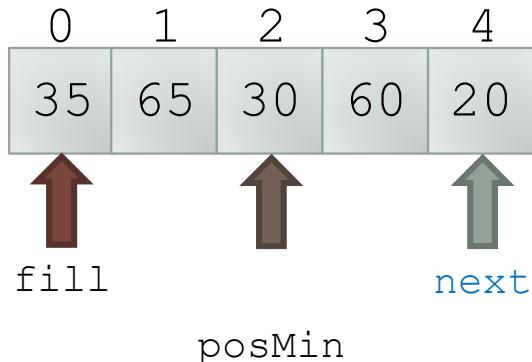
n	5
fill	0
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

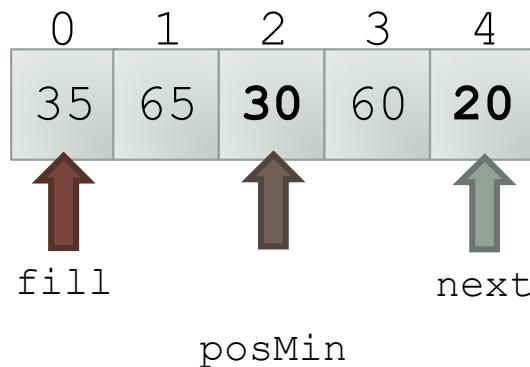
n	5
fill	0
posMin	2
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

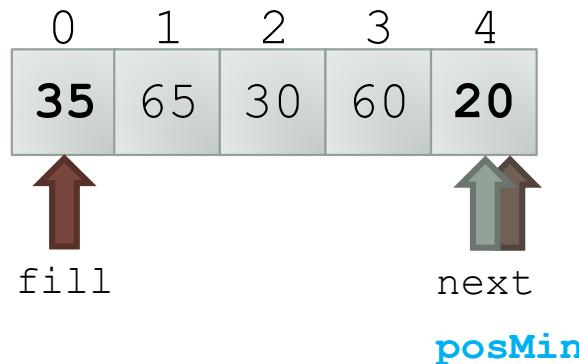
n	5
fill	0
posMin	2
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4. **if** the item at **next** is less than the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill

Trace of Selection Sort Refinement

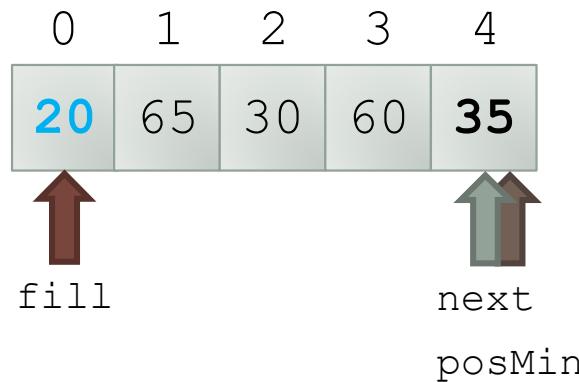
n	5
fill	0
posMin	4
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

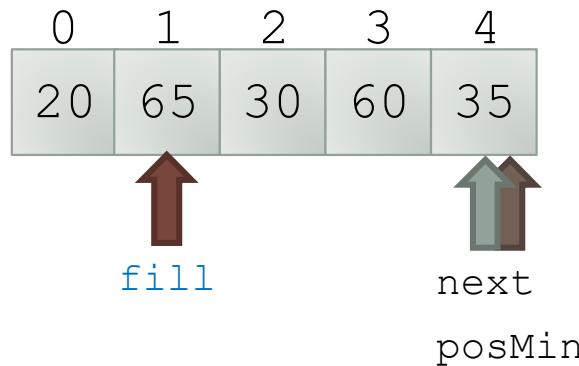
n	5
fill	0
posMin	4
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to **n - 1** do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. **Exchange** the item at posMin with the one at fill

Trace of Selection Sort Refinement

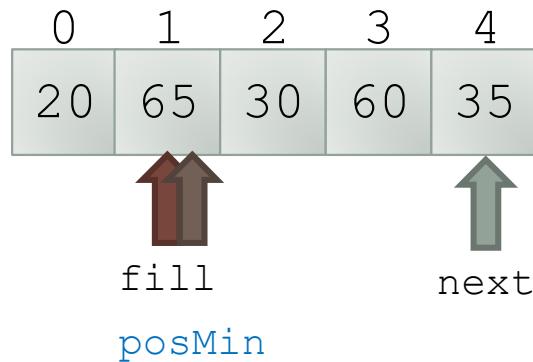
n	5
fill	1
posMin	4
next	4



- 1. **for** fill = 0 to n - 2 **do**
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
 do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

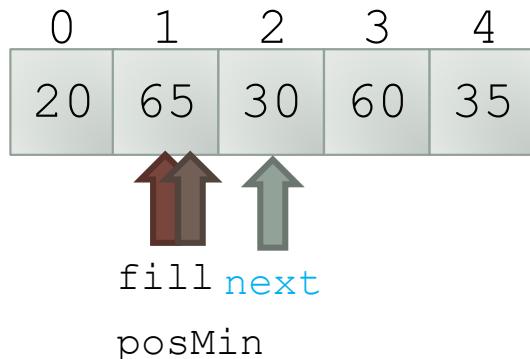
n	5
fill	1
posMin	1
next	4



- 1. **for** fill = 0 to n - 2 do
- 2. Initialize **posMin** to fill
- 3. **for** next = fill + 1 to n - 1 do
- 4. **if** the item at next is less than the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the one at fill

Trace of Selection Sort Refinement

n	5
fill	1
posMin	1
next	2



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

n	5
fill	1
posMin	1
next	2

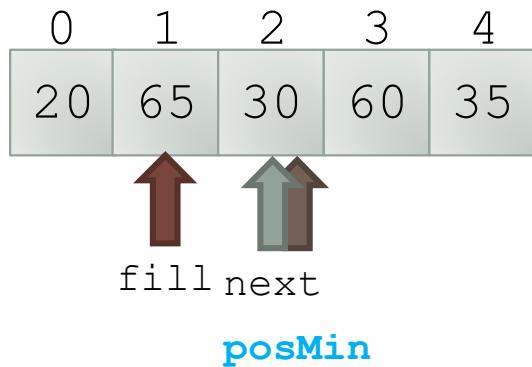
0	1	2	3	4
20	65	30	60	35

fill next
posMin

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

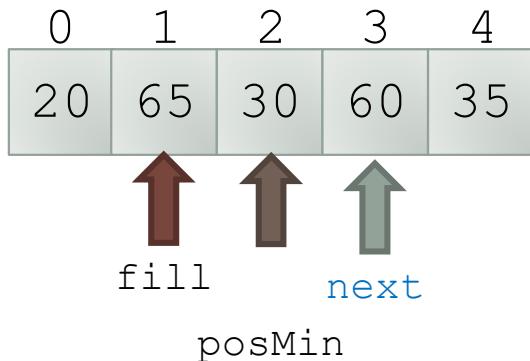
n	5
fill	1
posMin	2
next	2



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

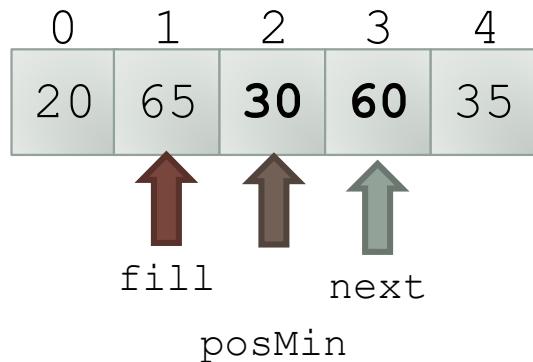
n	5
fill	1
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

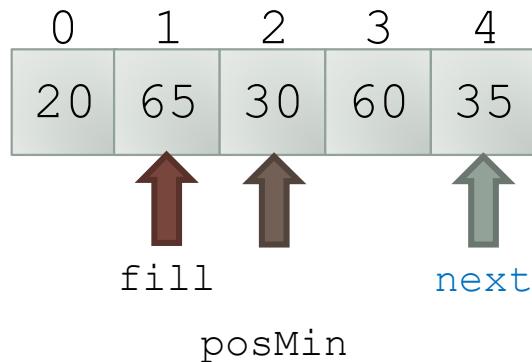
n	5
fill	1
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

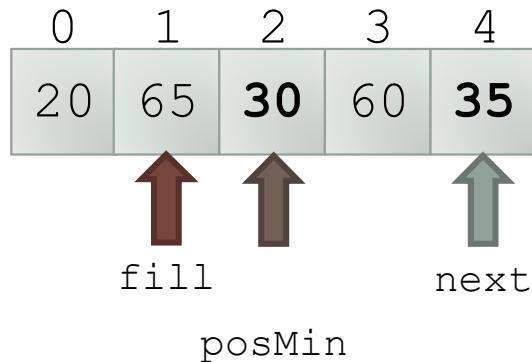
n	5
fill	1
posMin	2
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

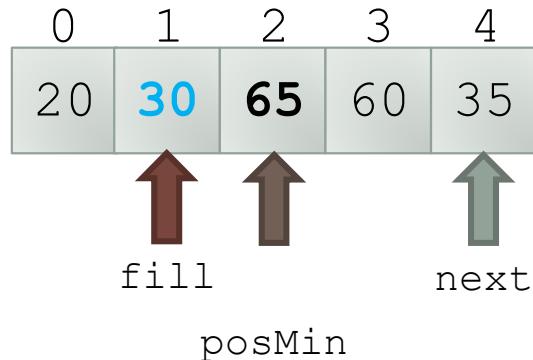
n	5
fill	1
posMin	2
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

n	5
fill	1
posMin	2
next	4

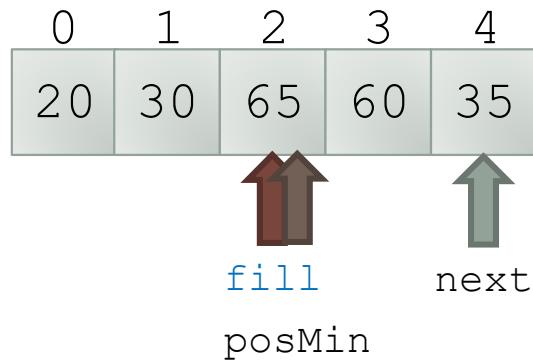


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to **n - 1** do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill



Trace of Selection Sort Refinement

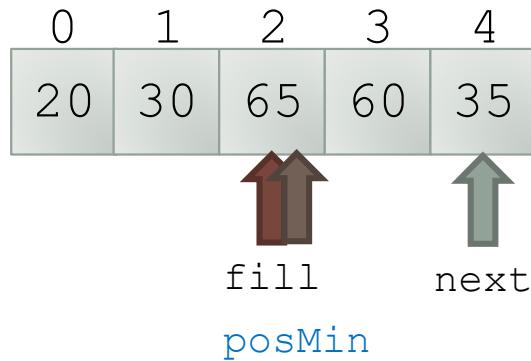
n	5
fill	2
posMin	2
next	4



- ▶ 1. **for** fill = 0 to n - 2 **do**
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
 do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

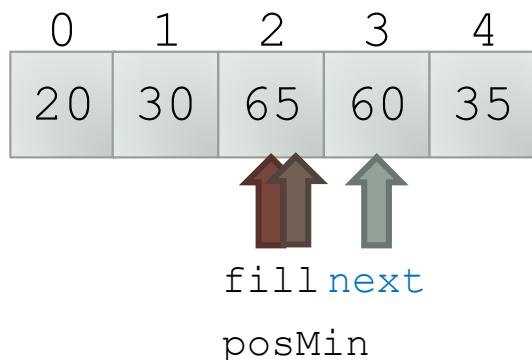
n	5
fill	2
posMin	2
next	4



- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

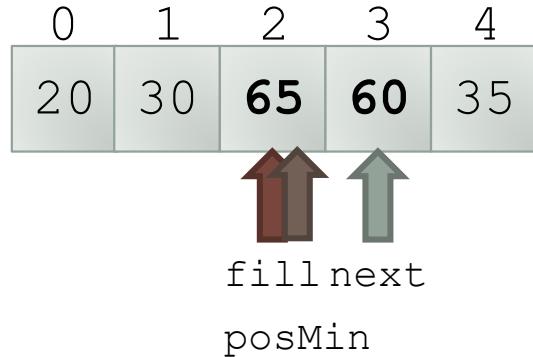
n	5
fill	2
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement (cont.)

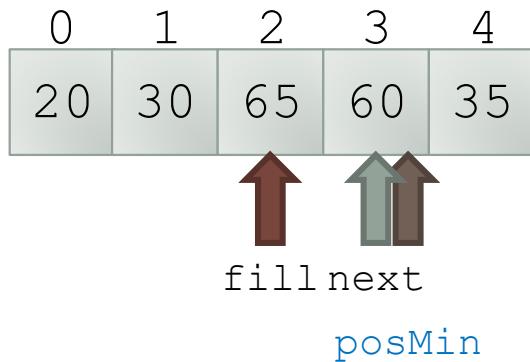
n	5
fill	2
posMin	2
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4. **if** the item at **next** is less than the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill

Trace of Selection Sort Refinement

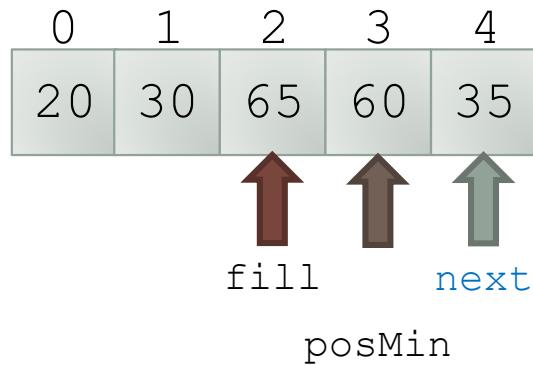
n	5
fill	2
posMin	3
next	3



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

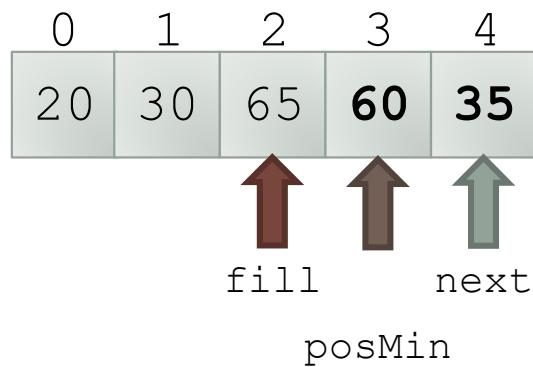
n	5
fill	2
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

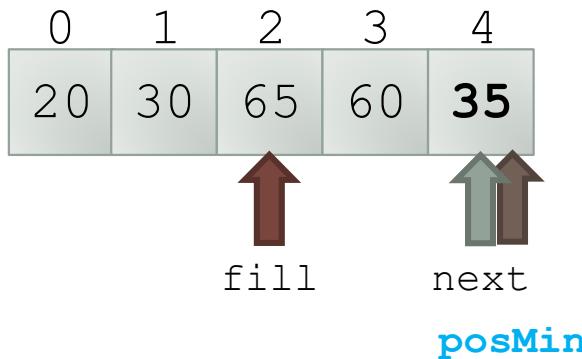
n	5
fill	2
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

n	5
fill	2
posMin	4
next	4

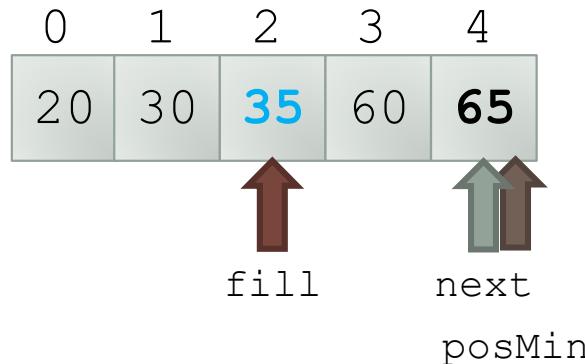


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill



Trace of Selection Sort Refinement

n	5
fill	2
posMin	4
next	4

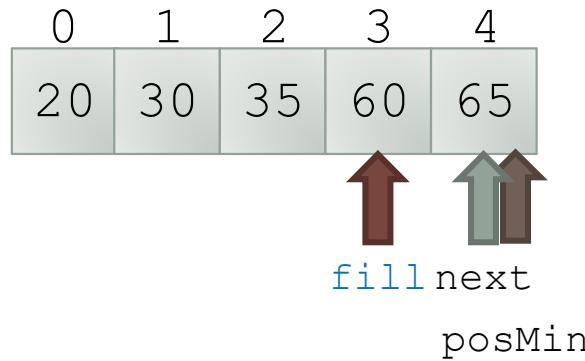


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. **Exchange** the item at posMin with
 the one at fill



Trace of Selection Sort Refinement

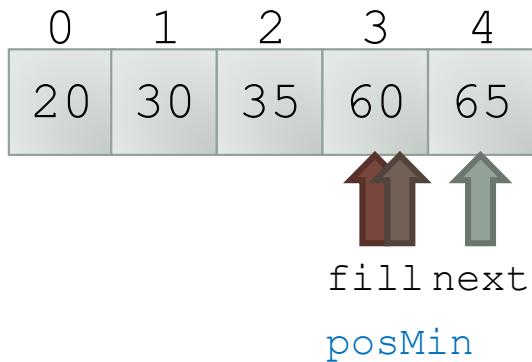
n	5
fill	3
posMin	4
next	4



- ▶ 1. **for** fill = 0 to n - 2 **do**
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
 do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

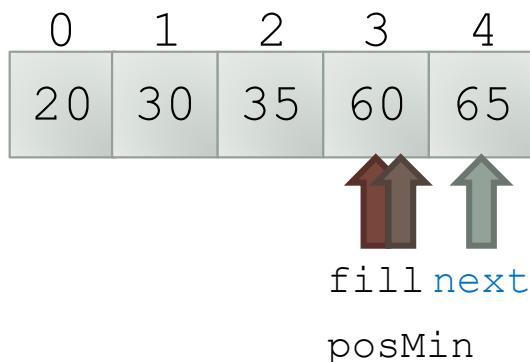
n	5
fill	3
posMin	3
next	4



- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1
do
- 4. **if** the item at next is less than
 the item at posMin
- 5. Reset posMin to next
- 6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

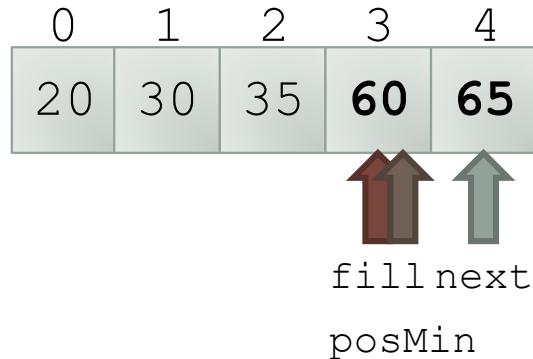
n	5
fill	3
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
one at fill

Trace of Selection Sort Refinement

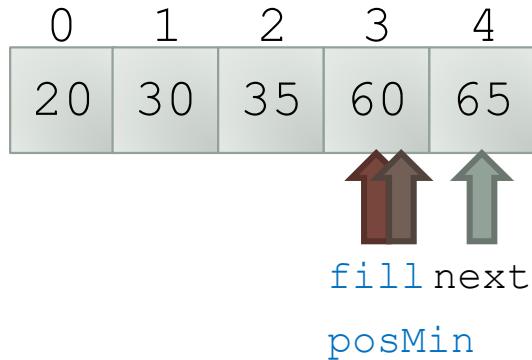
n	5
fill	3
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at **next** is less than
 the item at **posMin**
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Trace of Selection Sort Refinement

n	5
fill	3
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to **n - 1** do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. **Exchange** the item at posMin with the one at fill



Trace of Selection Sort Refinement

n	5
fill	3
posMin	3
next	4

0	1	2	3	4
20	30	35	60	65

1. **for** fill = 0 to **n - 2** do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Analysis of Selection Sort

This loop is performed $n-1$ times

1. **for** fill = 0 to $n - 2$ do
2. Initialize posMin to fill
3. **for** next = fill + 1 to $n - 1$ do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill

Analysis of Selection Sort (cont.)

1. **for** fill = 0 to n - 2 **do**
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
 do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

There are **n-1** exchanges



Analysis of Selection Sort (cont.)

This **comparison** is performed $(n - 1 - fill)$ times for each value of *fill* and can be represented by the following series:
 $(n-1) + (n-2) + \dots + 3 + 2 + 1$

1. **for** *fill* = 0 to $n - 2$ **do**
2. Initialize *posMin* to *fill*
3. **for** *next* = *fill* + 1 to $n - 1$ **do**
 4. **if** the item at *next* is less than the item at *posMin*
 5. Reset *posMin* to *next*
 6. Exchange the item at *posMin* with the one at *fill*

Analysis of Selection Sort (cont.)

The series
 $(n-1) + (n-2) + \dots + 3 + 2 + 1$
is a well-known series and can
be written as

$$\frac{n \times (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$



1. **for** fill = 0 to n - 2 **do**
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1
 do
4. **if** the item at next is less than
 the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the
 one at fill

Selection Sort

```
public static void sort(Object[] table) {
    int n = table.length;
    for (int fill = 0; fill < n - 1; fill++) {
        int posMin = fill;
        for (int next = fill + 1; next < n; next++) {
            // Invariant: table[posMin] is smallest item in
            // table[fill...next - 1].
            if (((Comparable) table[next]).compareTo(table[posMin]) < 0)
                posMin = next;
        }
        // assert: table[posMin] is smallest item in table[fill...n - 1]
        // Exchange table[fill] and table[posMin].
        var temp = table[fill];
        table[fill] = table[posMin];  table[posMin] = temp;
        // assert: table[fill] is smallest item in table[fill...n - 1]
    }
    // assert: table[0...n - 1] is sorted.
}
```

INSERTION SORT

Insertion Sort

- Another **quadratic sort**

- The player **keeps** the **cards** that have been picked up so far **in sorted order**
- When the player **picks** up a **new card**, the player makes room for the new card and then **inserts** it in **its proper place**



Trace of Insertion Sort

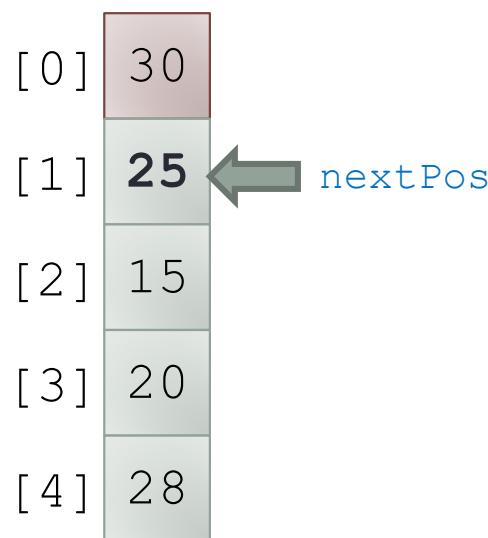
[0]	30
[1]	25
[2]	15
[3]	20
[4]	28

1. **for each** array element **from the second** (`nextPos = 1`) **to the last**
2. **Insert** the element at `nextPos` where it **belongs** in the array, **increasing the length of the sorted subarray** by 1 element

To adapt the insertion algorithm to an array that is filled with data, we start with a sorted subarray consisting of only the first element

Trace of Insertion Sort (cont.)

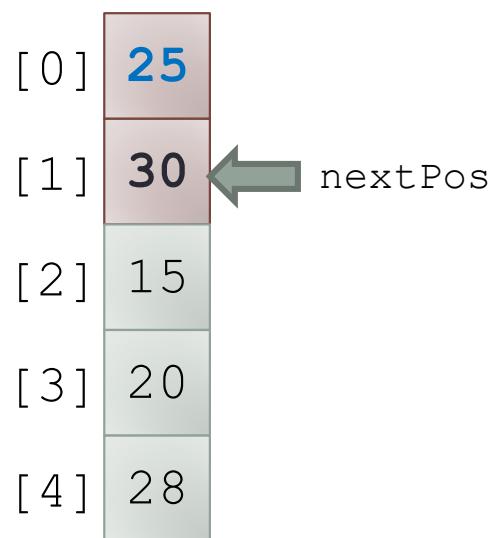
nextPos	1
---------	---



1. **for** each array element **from the second** (`nextPos = 1`) to the last
2. Insert the element at `nextPos` where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

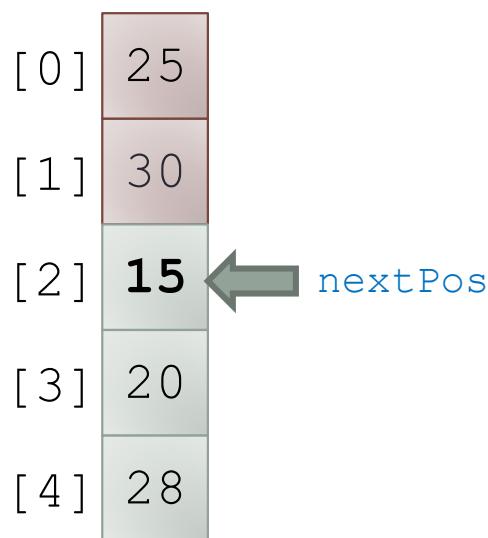
nextPos	1
---------	---



1. **for** each array element from the second (**nextPos** = 1) to the last
2. **Insert** the element at **nextPos** where it **belongs** in the array, increasing the **length** of the **sorted** subarray by 1 element

Trace of Insertion Sort (cont.)

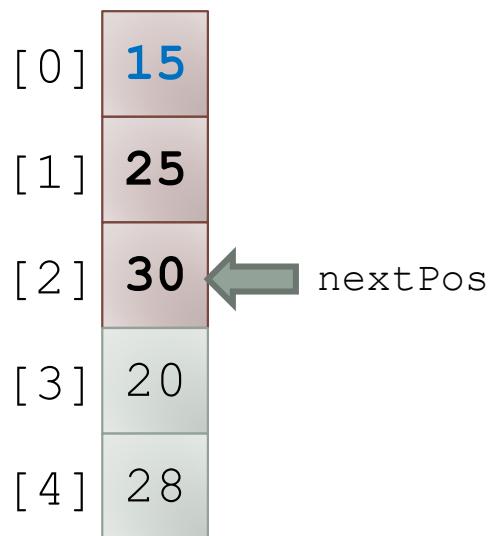
nextPos	2
---------	---



1. **for** each array element from the second (`nextPos = 1`) to the last
2. Insert the element at `nextPos` where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

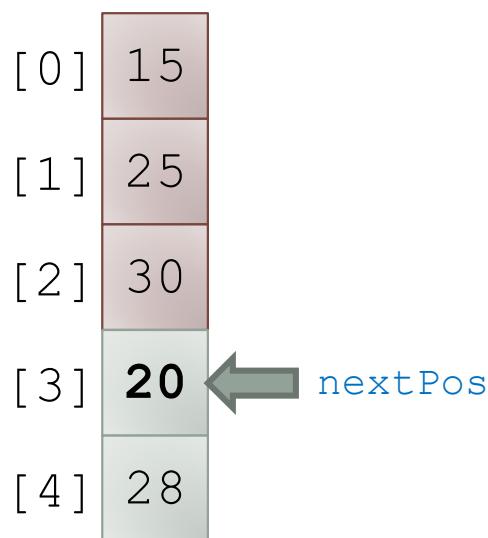
nextPos	2
---------	---



1. **for** each array element from the second (`nextPos = 1`) to the last
2. **Insert** the element at `nextPos` where it **belongs** in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

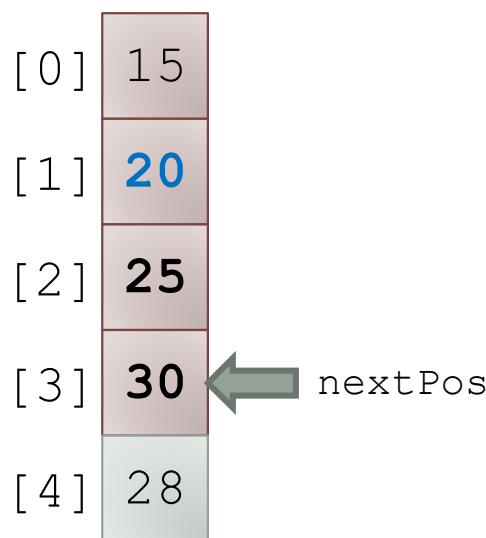
nextPos	3
---------	---



1. **for** each array element from the second (`nextPos = 1`) to the last
2. Insert the element at `nextPos` where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

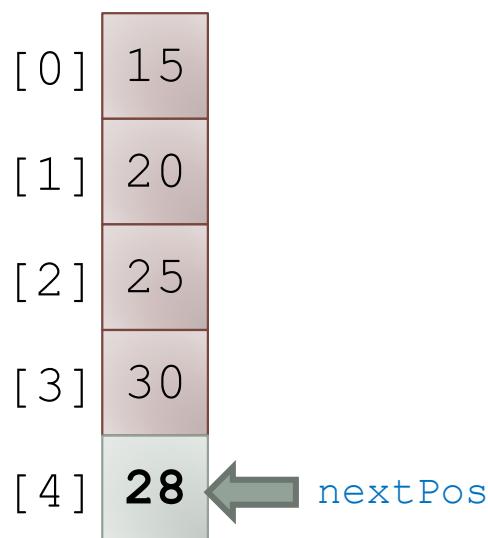
nextPos	3
---------	---



1. **for** each array element from the second (`nextPos = 1`) to the last
2. **Insert** the element at `nextPos` where it **belongs** in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

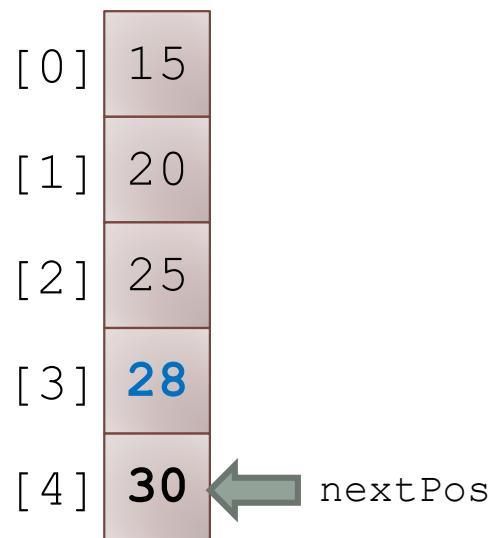
nextPos	4
---------	---



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

nextPos	4
---------	---



1. **for** each array element from the second (`nextPos = 1`) to the last
2. **Insert** the element at `nextPos` where it **belongs** in the array, increasing the length of the sorted subarray by 1 element

Trace of Insertion Sort (cont.)

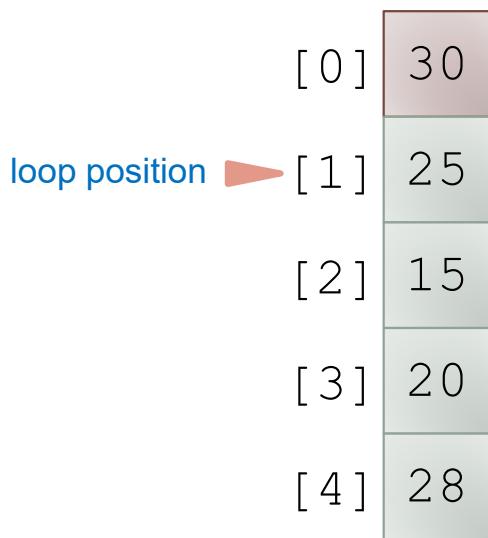
nextPos	-
---------	---

[0]	15
[1]	20
[2]	25
[3]	28
[4]	30

1. **for** each array element from the second (`nextPos = 1`) to **the last**
2. Insert the element at `nextPos` where it belongs in the array, increasing the length of the sorted subarray by 1 element

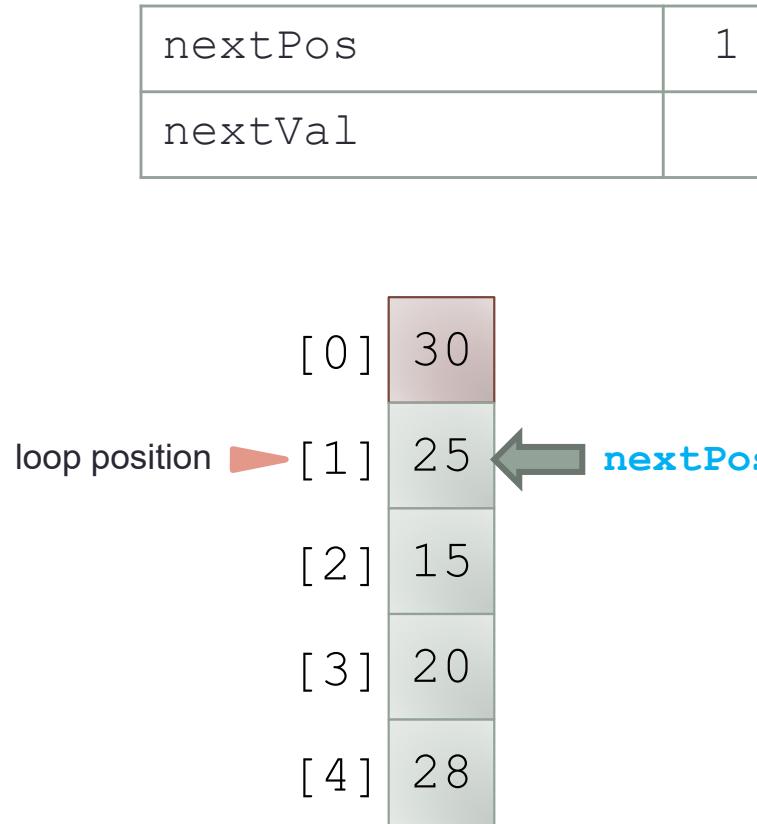
Trace of Insertion Sort Refinement

nextPos	1
nextVal	



- 1. **for each array element** from the **second** (**nextPos = 1**) to the last
- 2. **nextPos** is the position of the element to insert
- 3. Save the value of the element to insert in **nextVal**
- 4. **while** **nextPos > 0** and the element at **nextPos - 1 > nextVal**
- 5. Shift the element at **nextPos - 1** to position **nextPos**
- 6. Decrement **nextPos** by 1
- 7. Insert **nextVal** at **nextPos**

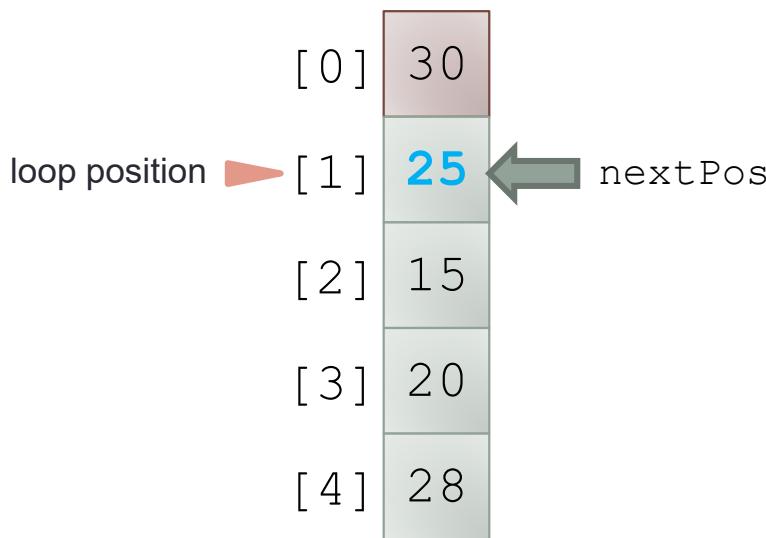
Trace of Insertion Sort Refinement (cont.)



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. **nextPos** is the **position** of the element **to insert**
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement **nextPos** by 1
7. Insert **nextVal** at **nextPos**

Trace of Insertion Sort Refinement (cont.)

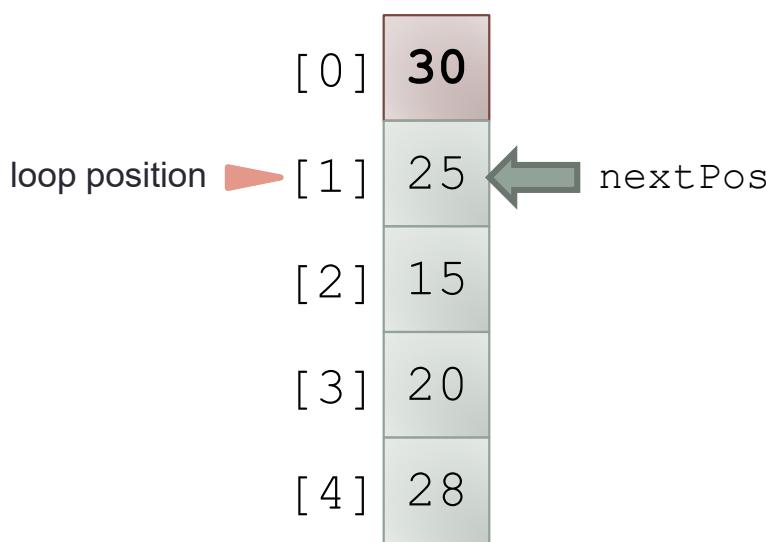
nextPos	1
nextVal	25



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the **value** of the element **to insert** in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

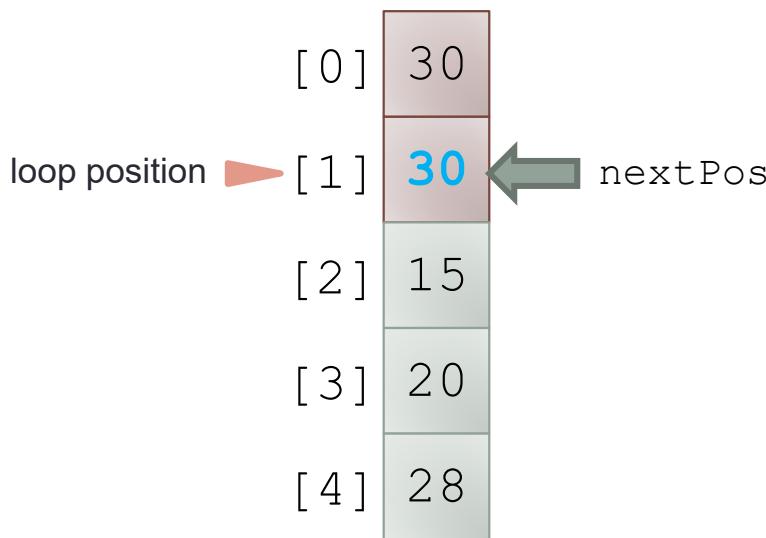
nextPos	1
nextVal	25



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

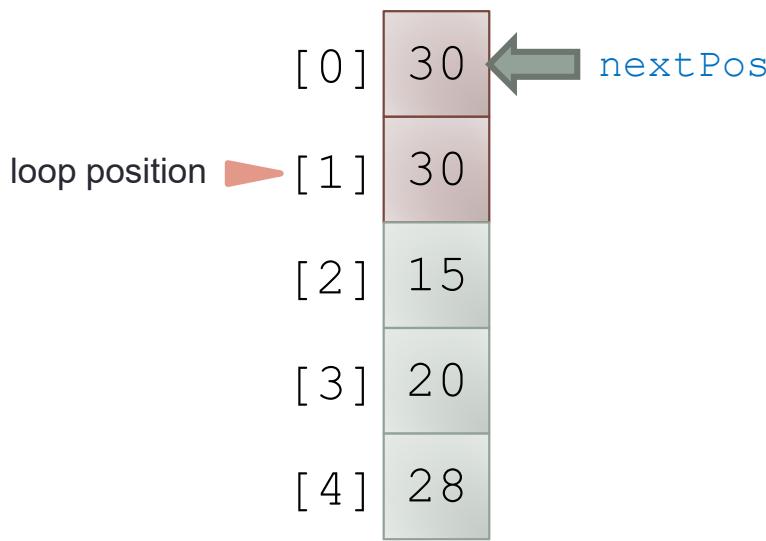
nextPos	1
nextVal	25



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. **Shift** the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

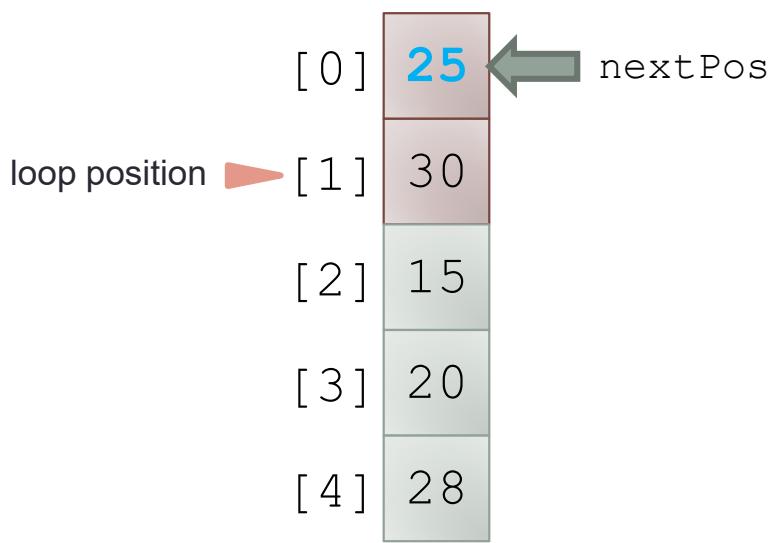
nextPos	0
nextVal	25



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. **Decrement nextPos by 1**
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

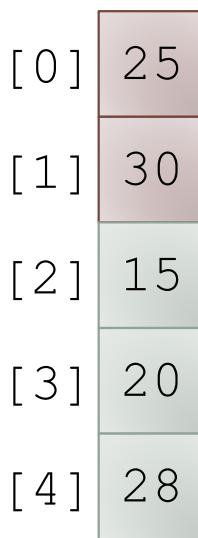
nextPos	0
nextVal	25



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. **Insert nextVal at nextPos**

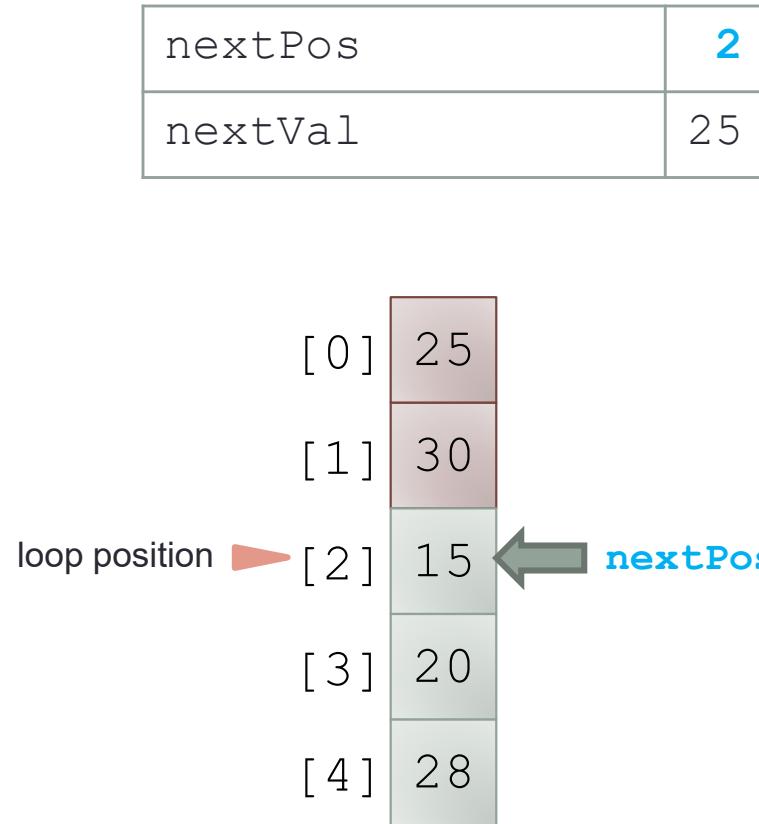
Trace of Insertion Sort Refinement (cont.)

nextPos	0
nextVal	25



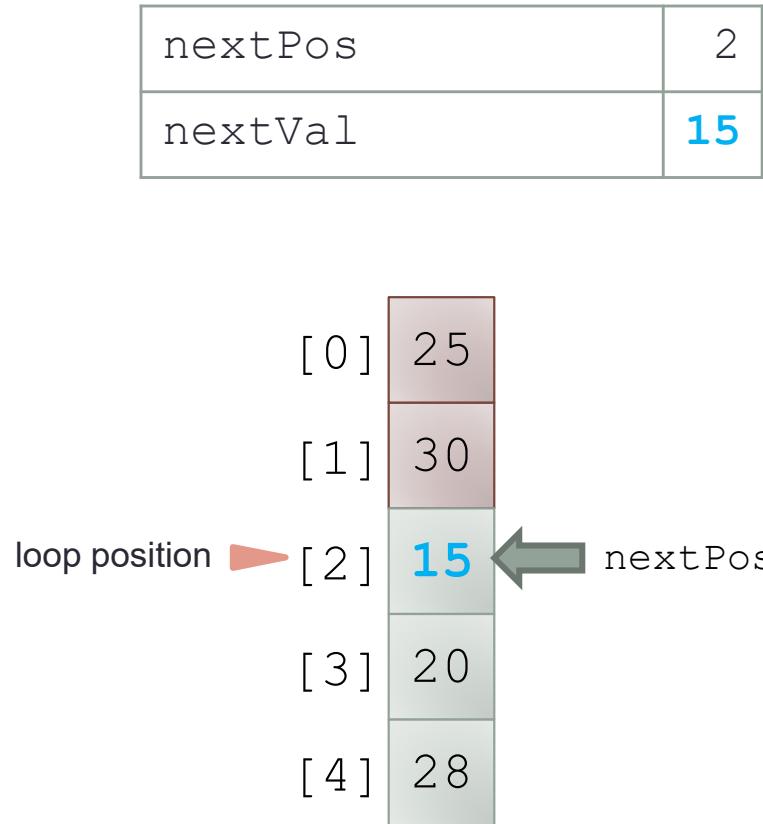
- ▶ 1. **for each array element** from the second ($\text{nextPos} = 1$) to the last
- 2. nextPos is the position of the element to insert
- 3. Save the value of the element to insert in nextVal
- 4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at $\text{nextPos} - 1$ to position nextPos
- 6. Decrement nextPos by 1
- 7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. **nextPos** is the position of the element to insert
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position **nextPos**
6. Decrement **nextPos** by 1
7. Insert **nextVal** at **nextPos**

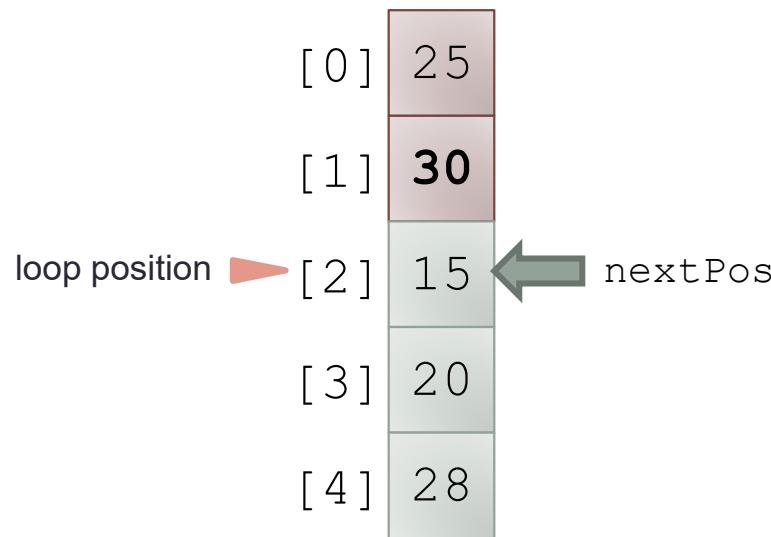
Trace of Insertion Sort Refinement (cont.)



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. **Save** the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. **Shift** the element at $\text{nextPos} - 1$ to position nextPos
6. **Decrement** nextPos by 1
7. **Insert** nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

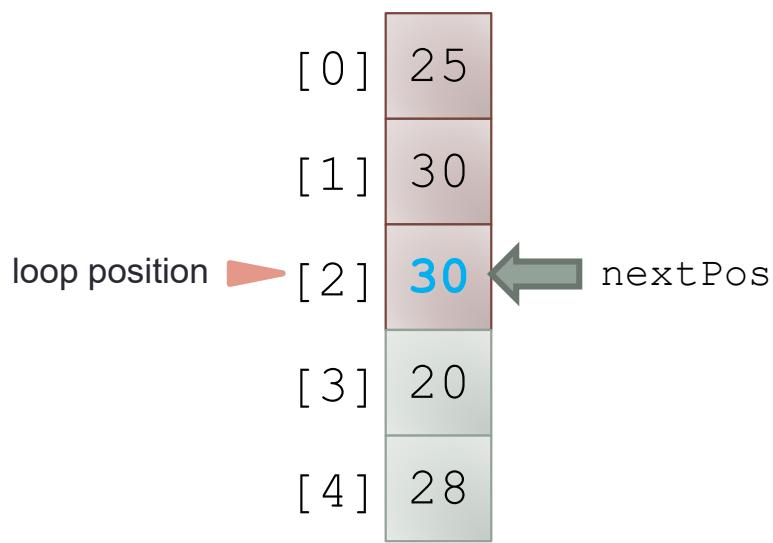
nextPos	2
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

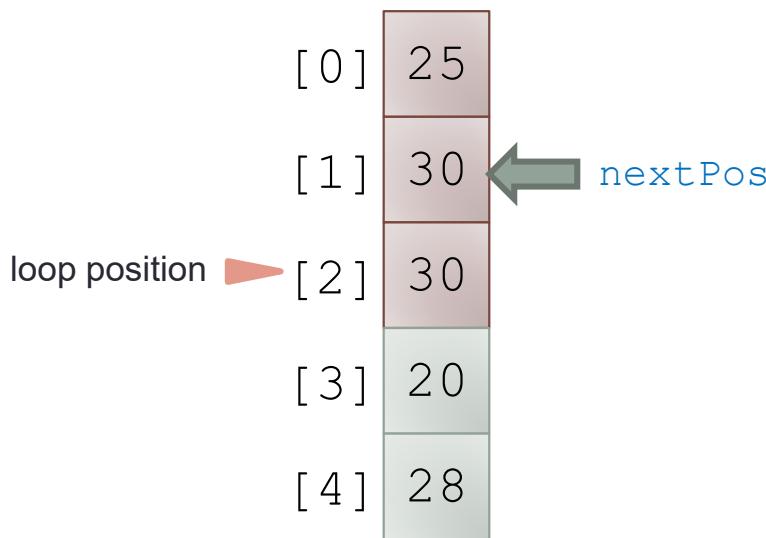
nextPos	2
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

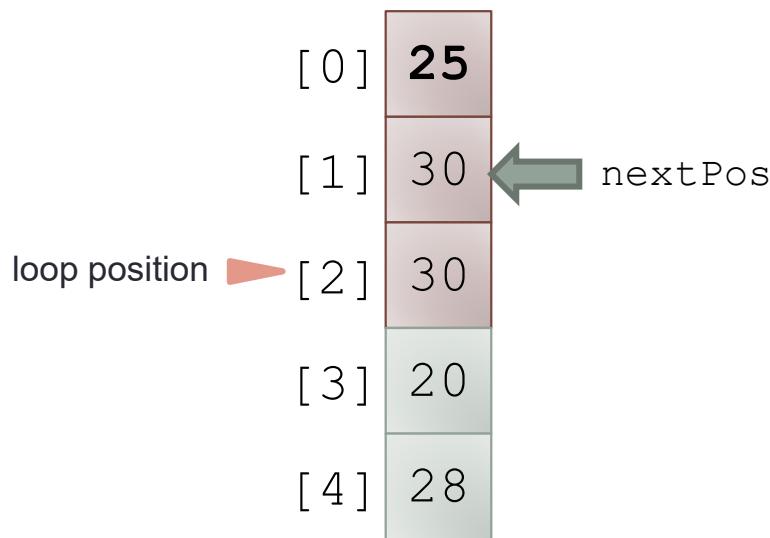
nextPos	1
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

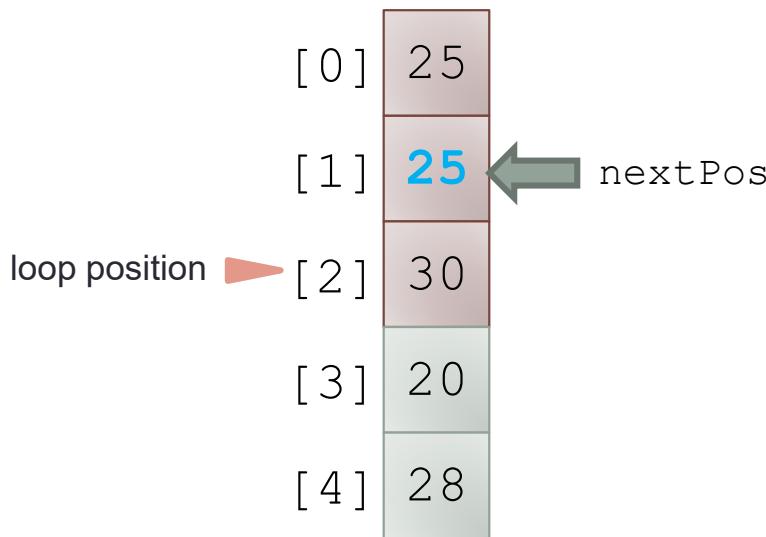
nextPos	1
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

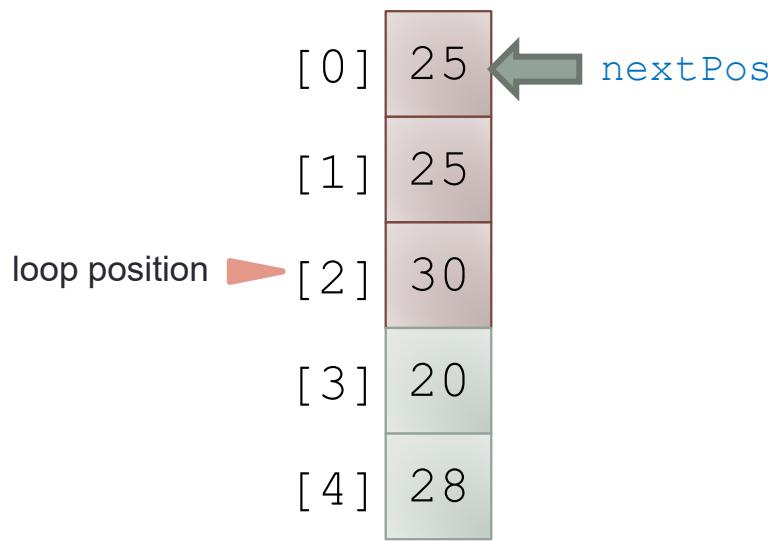
nextPos	1
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. **Shift** the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

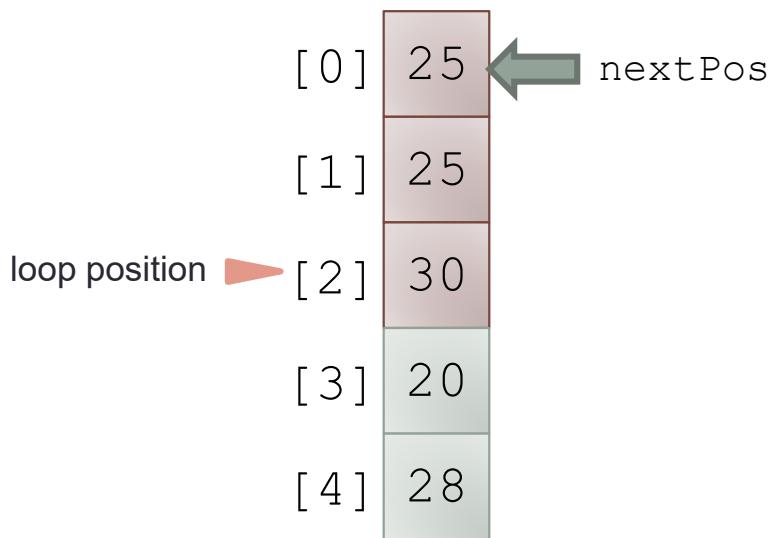
nextPos	0
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement **nextPos** by 1 ➔
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

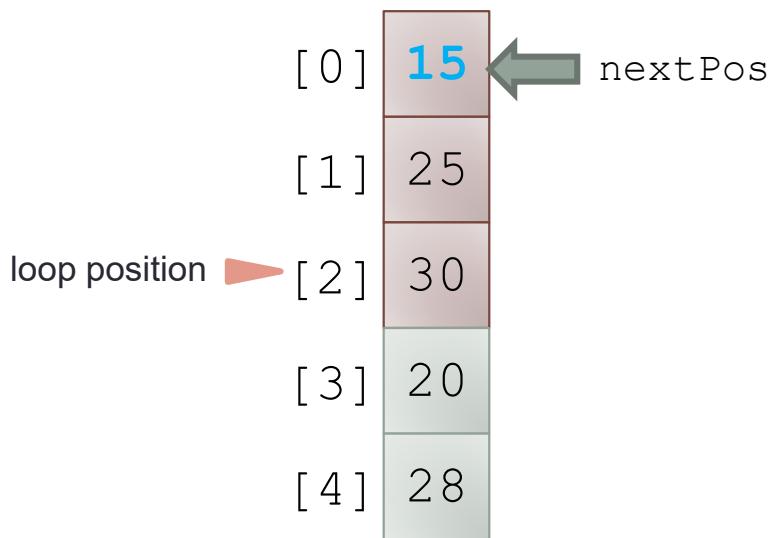
nextPos	0
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

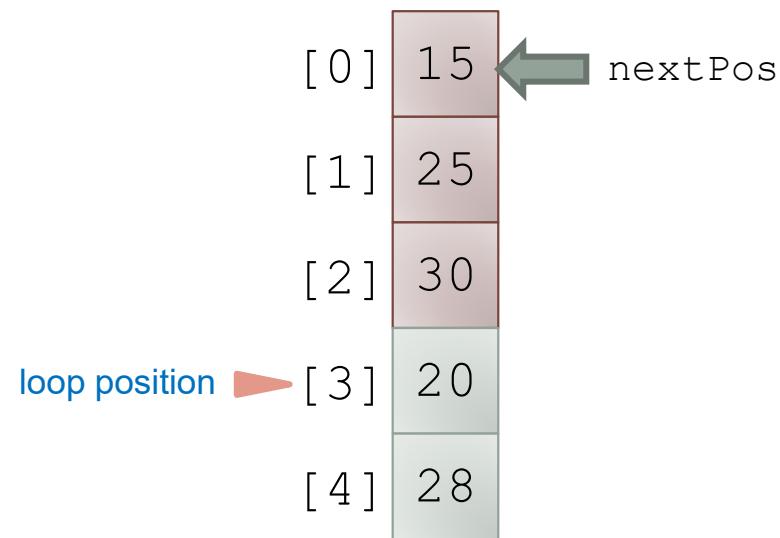
nextPos	0
nextVal	15



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. **Insert nextVal at nextPos**

Trace of Insertion Sort Refinement (cont.)

nextPos	0
nextVal	15



- 1. **for each array element** from the second ($\text{nextPos} = 1$) to the last
- 2. nextPos is the position of the element to insert
- 3. Save the value of the element to insert in nextVal
- 4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at $\text{nextPos} - 1$ to position nextPos
- 6. Decrement nextPos by 1
- 7. Insert nextVal at nextPos

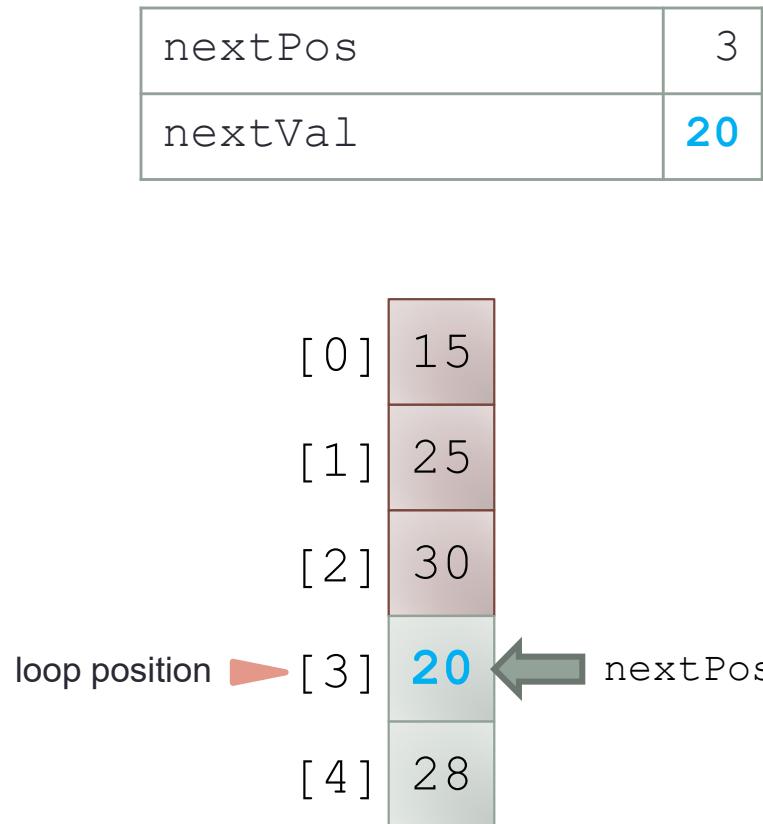
Trace of Insertion Sort Refinement (cont.)

nextPos		3
nextVal		15

[0] 15
[1] 25
[2] 30
loop position [3] 20
[4] 28

1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. **nextPos** is the position of the element to insert
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position **nextPos**
6. Decrement **nextPos** by 1
7. Insert **nextVal** at **nextPos**

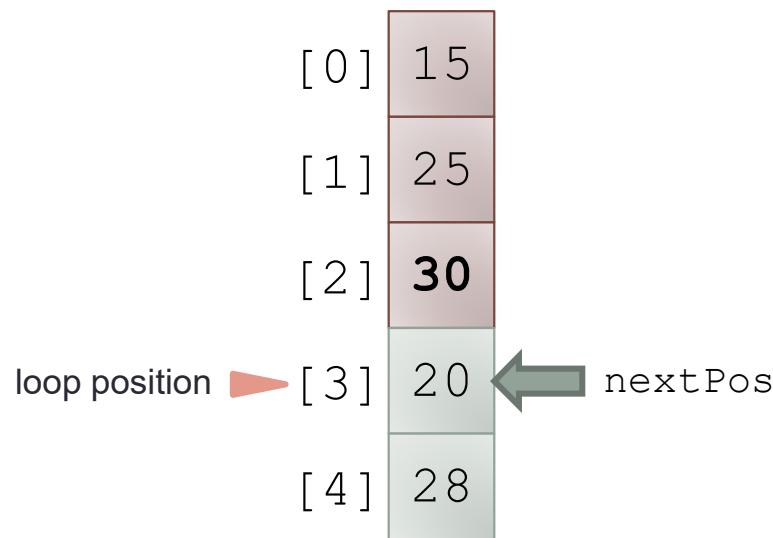
Trace of Insertion Sort Refinement (cont.)



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert **nextVal** at nextPos

Trace of Insertion Sort Refinement (cont.)

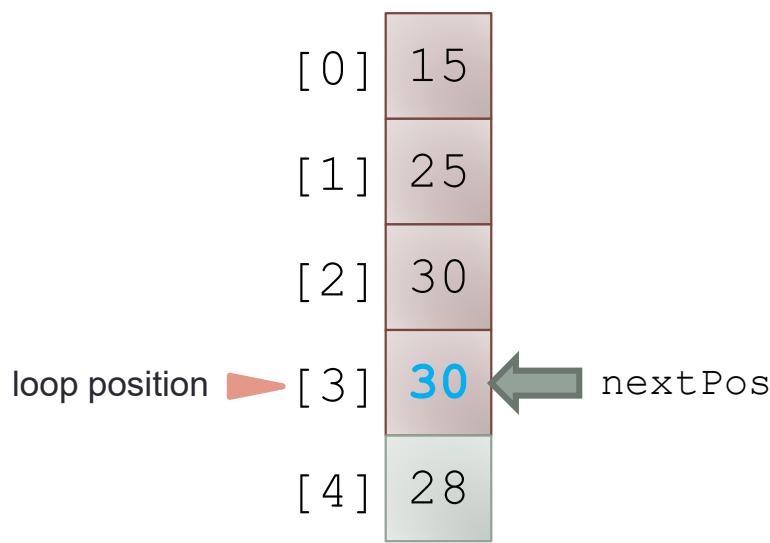
nextPos	3
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

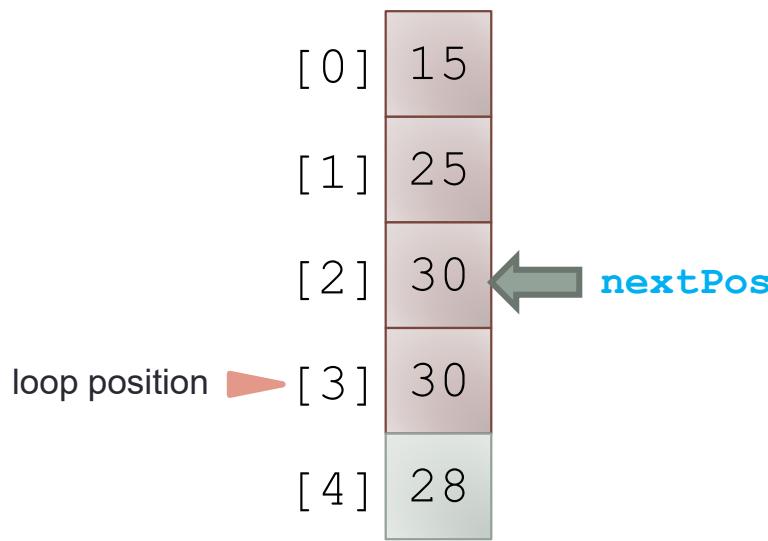
nextPos	3
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. **Shift** the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

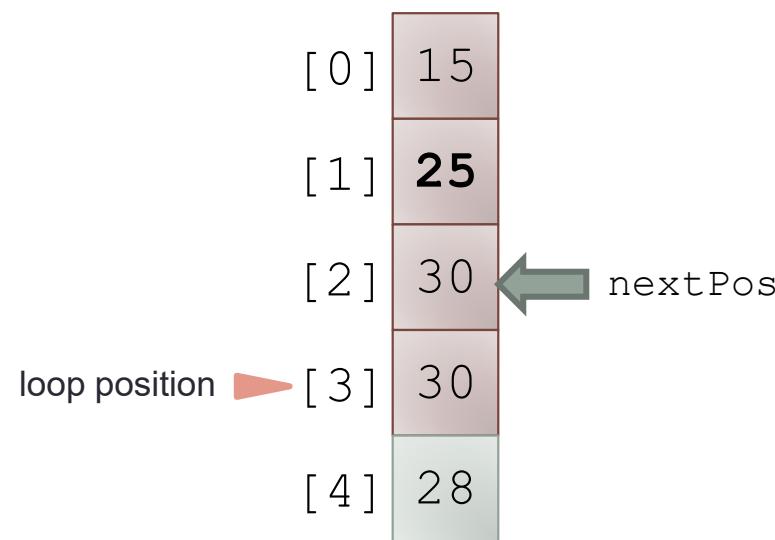
nextPos	2
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

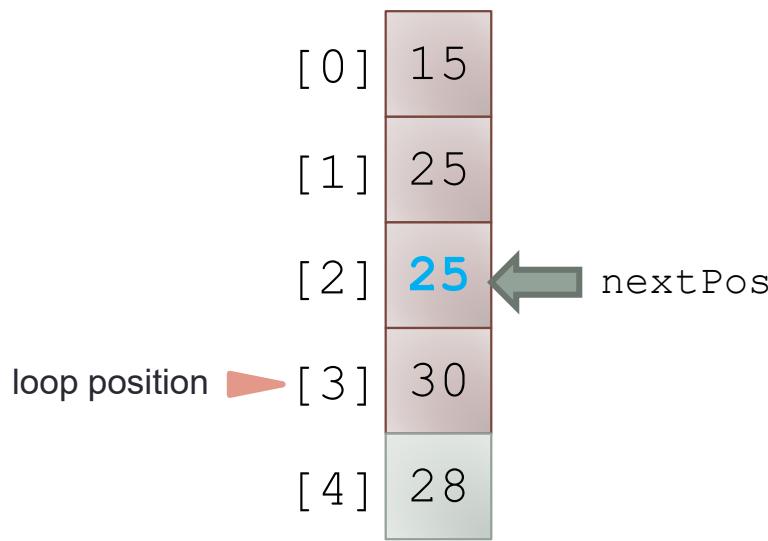
nextPos	2
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

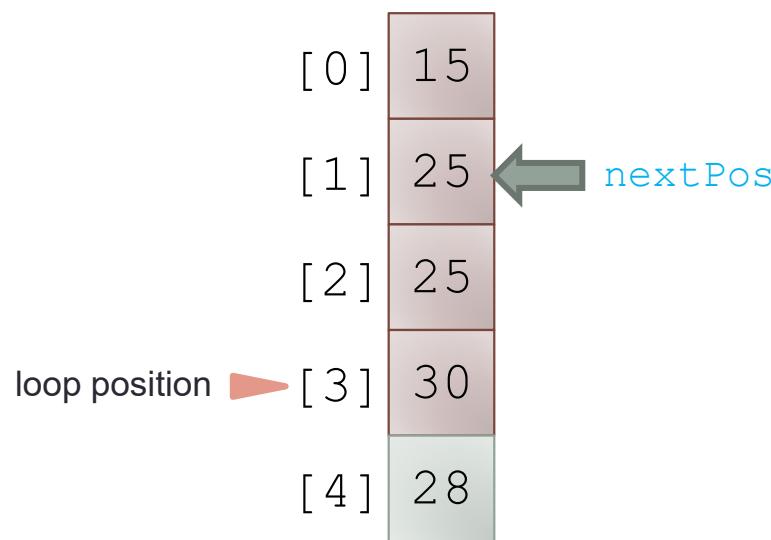
nextPos	2
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

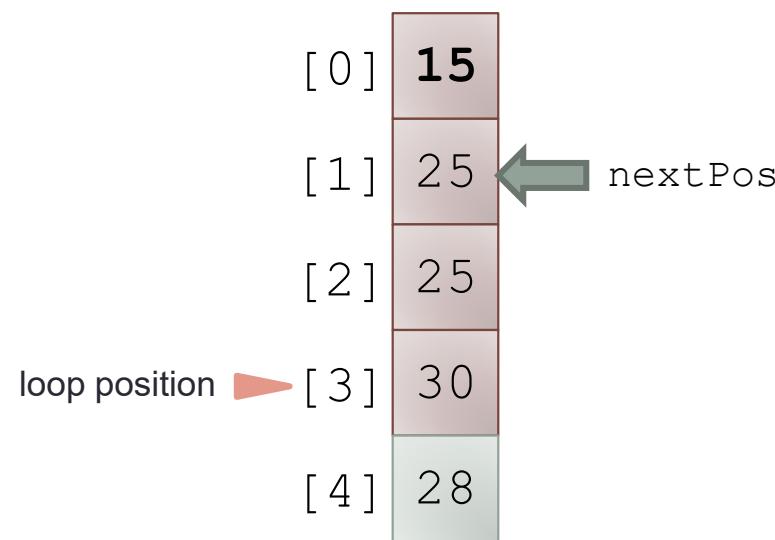
nextPos	1
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

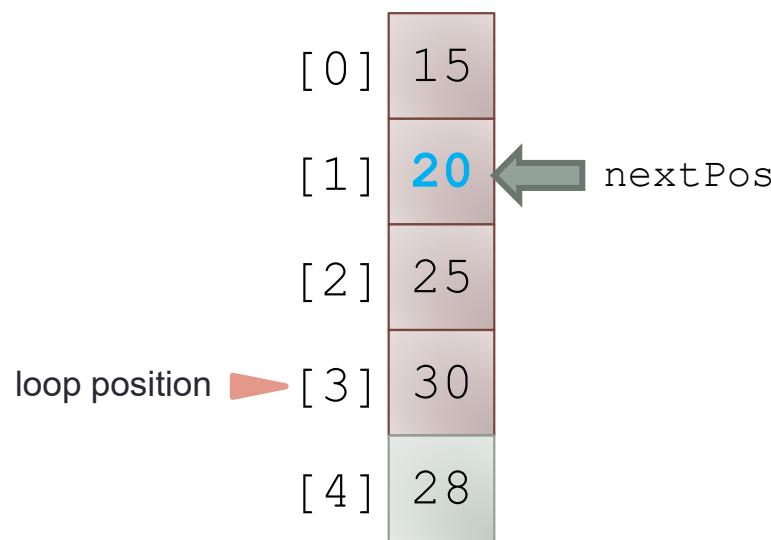
nextPos	1
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

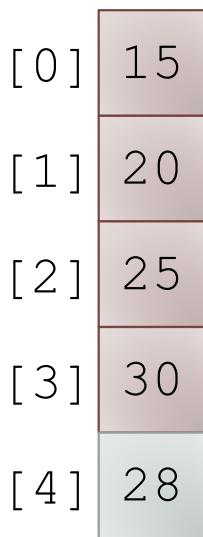
nextPos	1
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. **Insert nextVal at nextPos**

Trace of Insertion Sort Refinement (cont.)

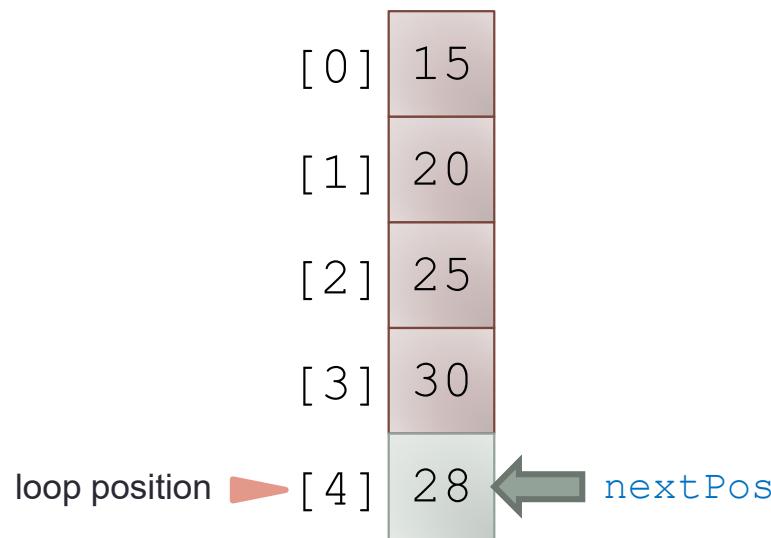
nextPos	1
nextVal	20



- 1. **for each array element** from the second ($\text{nextPos} = 1$) to the last
- 2. nextPos is the position of the element to insert
- 3. Save the value of the element to insert in nextVal
- 4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at $\text{nextPos} - 1$ to position nextPos
- 6. Decrement nextPos by 1
- 7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

nextPos	4
nextVal	20



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. **nextPos** is the position of the element to insert
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position **nextPos**
6. Decrement **nextPos** by 1
7. Insert **nextVal** at **nextPos**

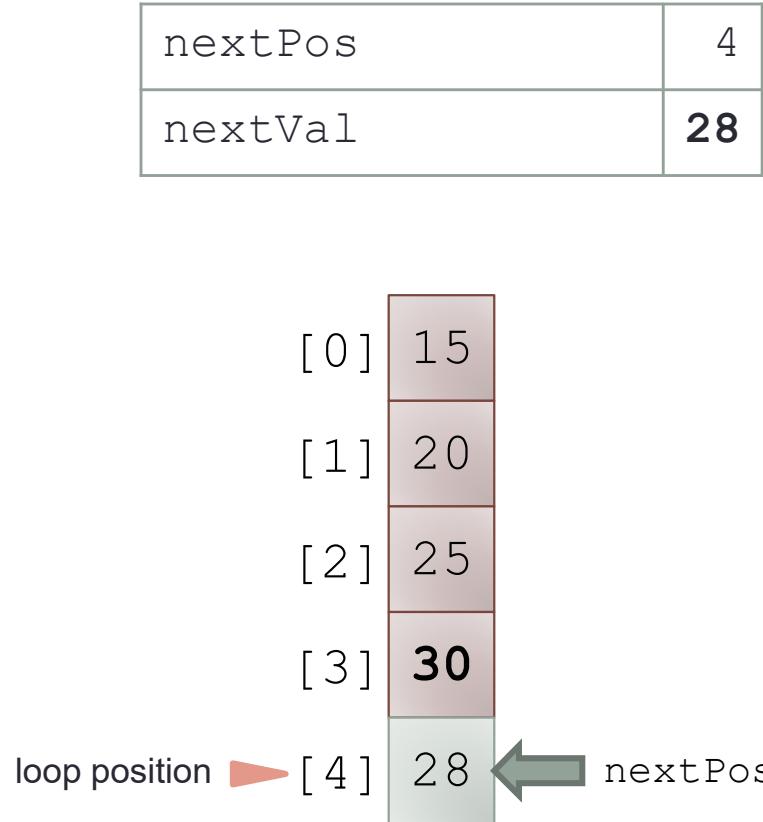
Trace of Insertion Sort Refinement (cont.)

nextPos	4
nextVal	28

[0] 15
[1] 20
[2] 25
[3] 30
loop position ► [4] 28 ← nextPos

1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in **nextVal**
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

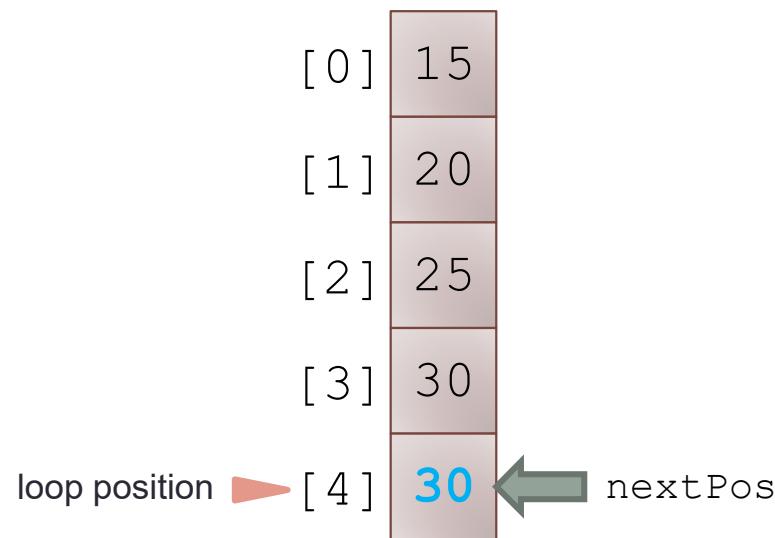
Trace of Insertion Sort Refinement (cont.)



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

nextPos	4
nextVal	28



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. **Shift** the element at $\text{nextPos}-1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

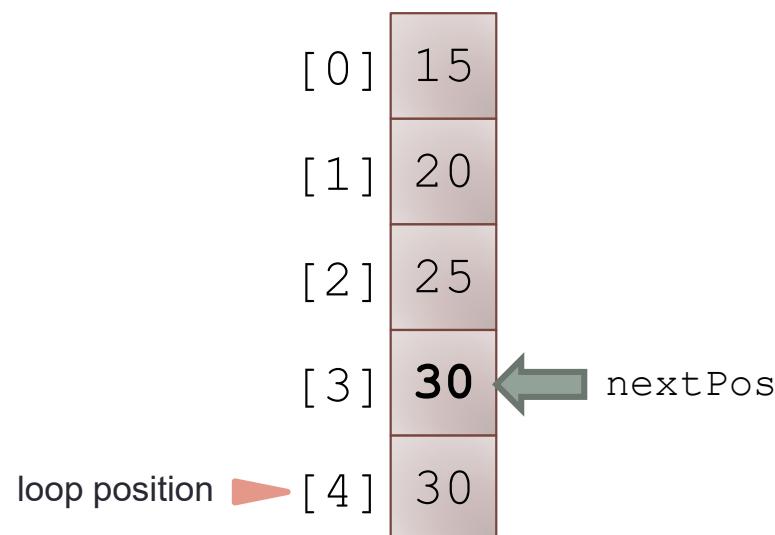
nextPos	3
nextVal	28

[0] 15
[1] 20
[2] 25
[3] 30 ← nextPos
loop position ► [4] 30

1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

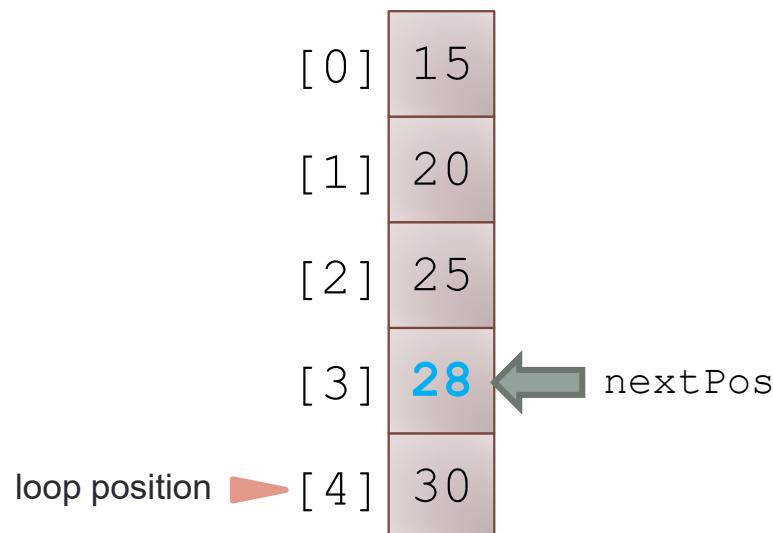
nextPos	3
nextVal	28



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos}-1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Trace of Insertion Sort Refinement (cont.)

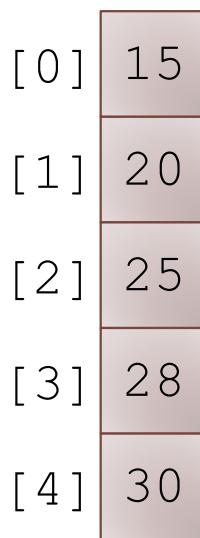
nextPos	3
nextVal	28



1. **for** each array element from the second ($\text{nextPos} = 1$) to the last
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. **Insert nextVal at nextPos**

Trace of Insertion Sort Refinement (cont.)

nextPos	3
nextVal	28



1. **for** each array element from the second ($\text{nextPos} = 1$) to **the last**
2. nextPos is the position of the element to insert
3. Save the value of the element to insert in nextVal
4. **while** $\text{nextPos} > 0$ and the element at $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at $\text{nextPos} - 1$ to position nextPos
6. Decrement nextPos by 1
7. Insert nextVal at nextPos

Analysis of Insertion Sort

- The **insertion** step is performed $n - 1$ times
- In **the best case** (when the array is sorted already), only **one comparison** is required for each insertion
 - In the best case, the number of comparisons is **$O(n)$**
- In **the worst case**, **all elements** in the sorted subarray are **compared** to **nextVal** for each insertion
 - The maximum number of comparisons then will be:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1)$$

which is **$O(n^2)$**

```

public class InsertionSort {

    /** Sort the table using insertion sort algorithm.
     * @param table The array to be sorted */
    public static <T extends Comparable<T>> void sort(T[] table) {
        for (int nextPos = 1; nextPos < table.length; nextPos++) {
            insert(table, nextPos); // Insert element at position nextPos in the sorted subarray
        }
    }

    /** Insert the element at nextPos where it belongs in the array.
     * @param table The array being sorted
     * @param nextPos The position of the element to insert */
    private static <T extends Comparable<T>> void insert(T[] table, int nextPos) {
        T nextVal = table[nextPos]; // Element to insert.
        while (nextPos > 0 && nextVal.compareTo(table[nextPos - 1]) < 0) {
            table[nextPos] = table[nextPos - 1];
            nextPos-- ; // Shift down.
        }
        table[nextPos] = nextVal; // Insert nextVal at nextPos.
    }
}

```

COMPARISON OF QUADRATIC SORTS

Comparison of Quadratic Sorts

Sort kind	Comparisons	Comparisons	Exchanges	Exchanges
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$

Comparison of growth rates

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Comparison of Quadratic Sorts (cont.)

- The **best sorting** algorithms provide **$O(n \log n)$** **average** case performance
- **Insertion** sort
 - ✓ gives the **good performance** for **most** arrays
 - takes advantage of any **partial sorting** in the array and uses less costly shifts

Comparisons VS Exchanges

- In Java, an **exchange** requires a **switch** of **two** object references using a third object reference as an intermediary
- A **comparison** requires an execution of a **compareTo()**
- The cost of a **comparison** depends on its **complexity**, but is generally **more costly** than an exchange

SHELL SORT

A Better Insertion Sort

Shell Sort: A Better Insertion Sort

- A **divide-and-conquer** approach to **insertion sort**
 - ✓ Instead of **sorting the entire array**, Shell sort **sorts many smaller subarrays** using **insertion sort before** sorting the **entire array**

- A **type** of **insertion sort**
 - ✓ but with **$O(n^{3/2})$** or better performance than the **$O(n^2)$**

Trace of Shell Sort

gap value	7
-----------	---

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65

Trace of Shell Sort (cont.)

gap value	7	= N-1 / 2
-----------	---	-----------

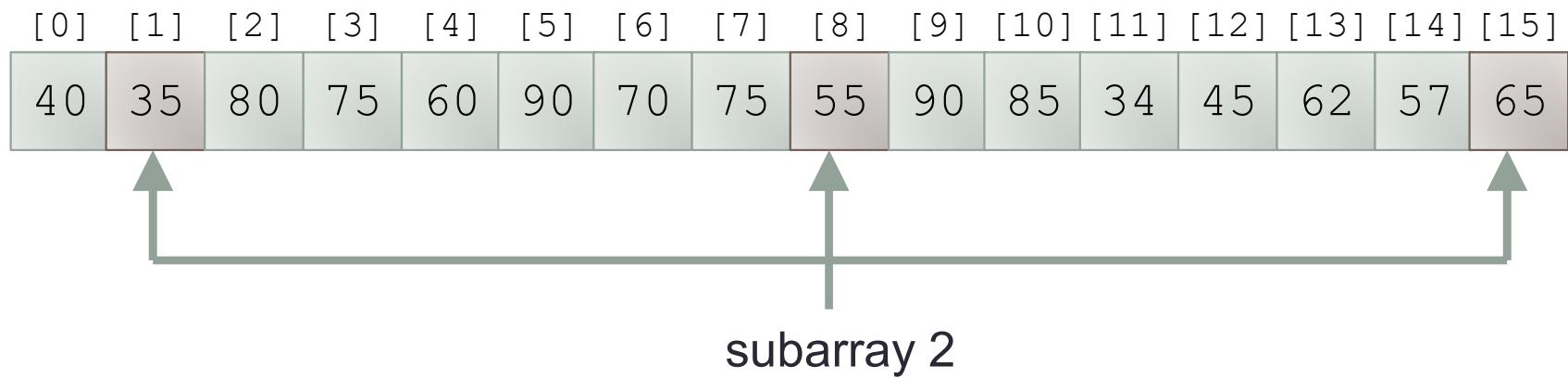
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65



subarray 1

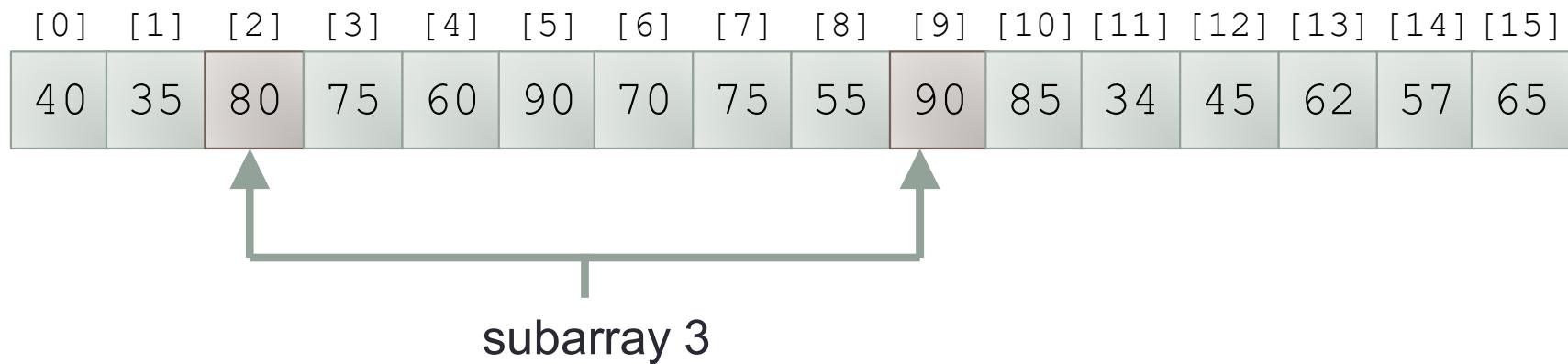
Trace of Shell Sort (cont.)

gap value	7
-----------	---



Trace of Shell Sort (cont.)

gap value	7
-----------	---



Trace of Shell Sort (cont.)

gap value	7
-----------	---

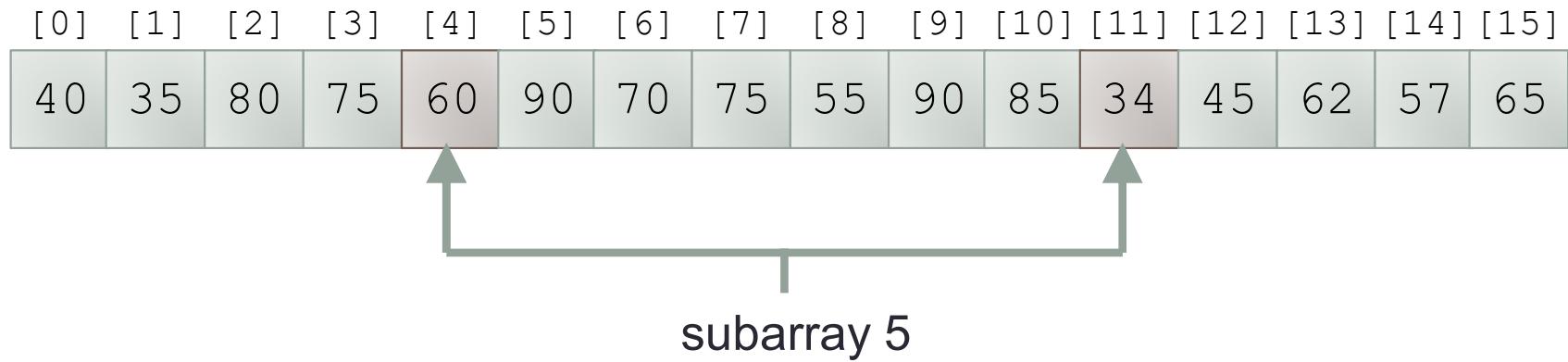
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65



subarray 4

Trace of Shell Sort (cont.)

gap value	7
-----------	---



Trace of Shell Sort (cont.)

gap value	7
-----------	---

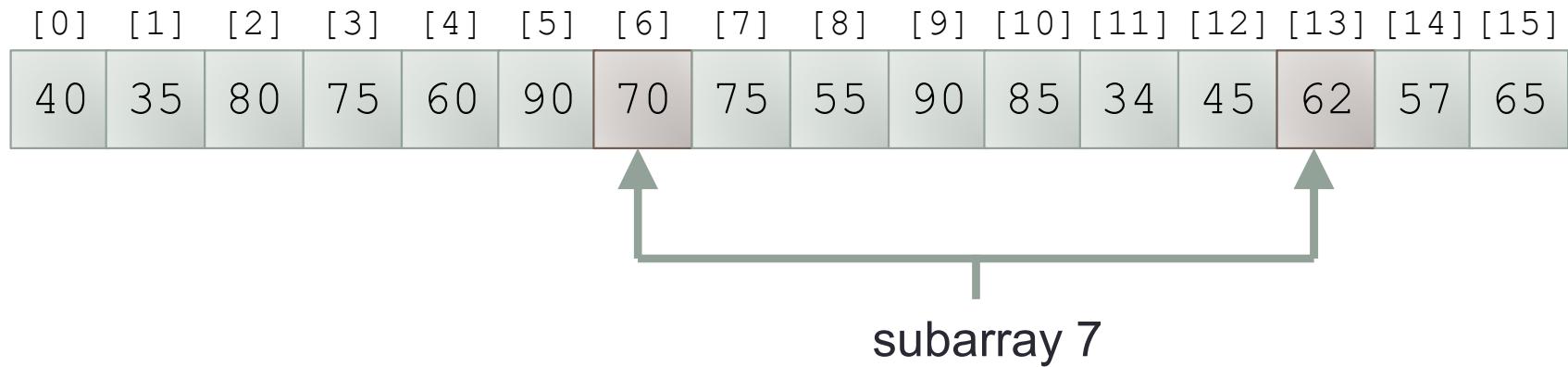
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65



subarray 6

Trace of Shell Sort (cont.)

gap value	7
-----------	---



Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65



subarray 1

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	57	55	90	85	34	45	62	75	65



subarray 2

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	57	55	90	85	34	45	62	75	65



subarray 3

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	57	55	90	85	34	45	62	75	65



Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	57	55	90	85	34	45	62	75	65

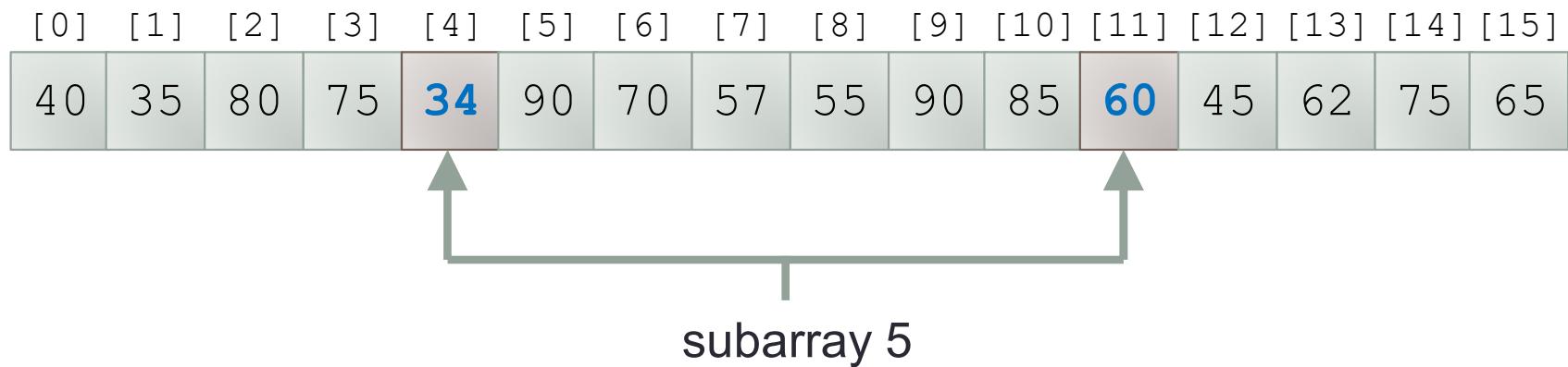


subarray 5

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 5



Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	90	70	57	55	90	85	60	45	62	75	65



subarray 6

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	70	57	55	90	85	60	90	62	75	65



subarray 6

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	70	57	55	90	85	60	90	62	75	65



subarray 7

Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

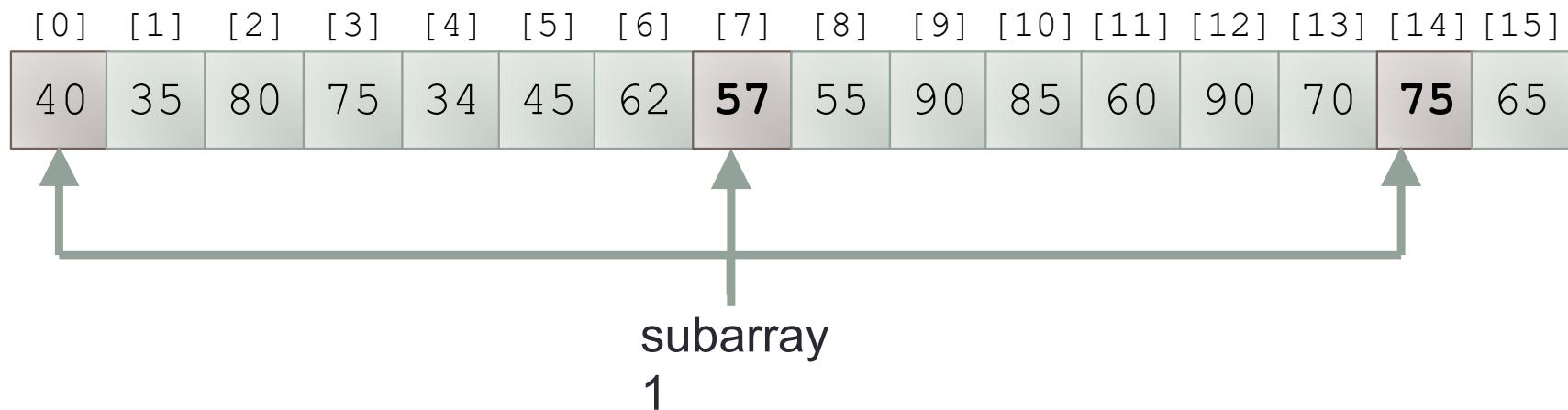


subarray 7

Trace of Shell Sort (cont.)

gap value	7
-----------	---

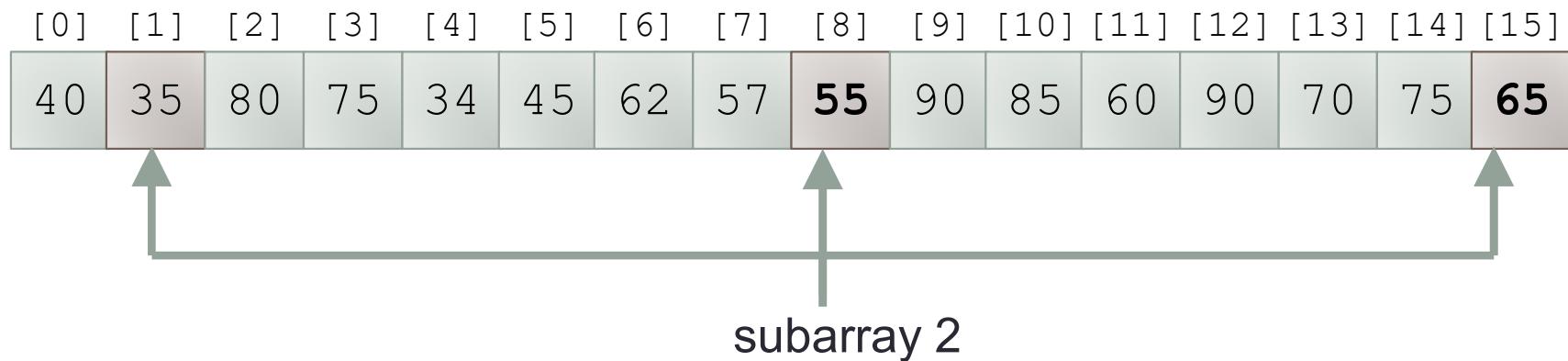
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort subarray 2



Trace of Shell Sort (cont.)

gap value	7
-----------	---

Sort on **smaller gap** value **next**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

Trace of Shell Sort (cont.)

gap value	3
-----------	---

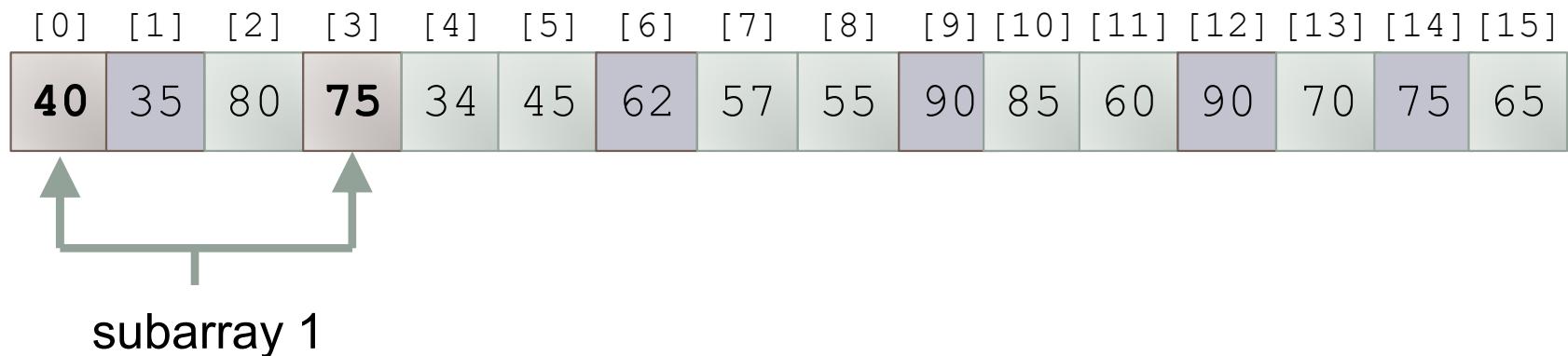
Sort on smaller gap value
 $= \text{gap}/2.2 = 7/2.2 = 3.18$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

Trace of Shell Sort (cont.)

gap value	3
-----------	---

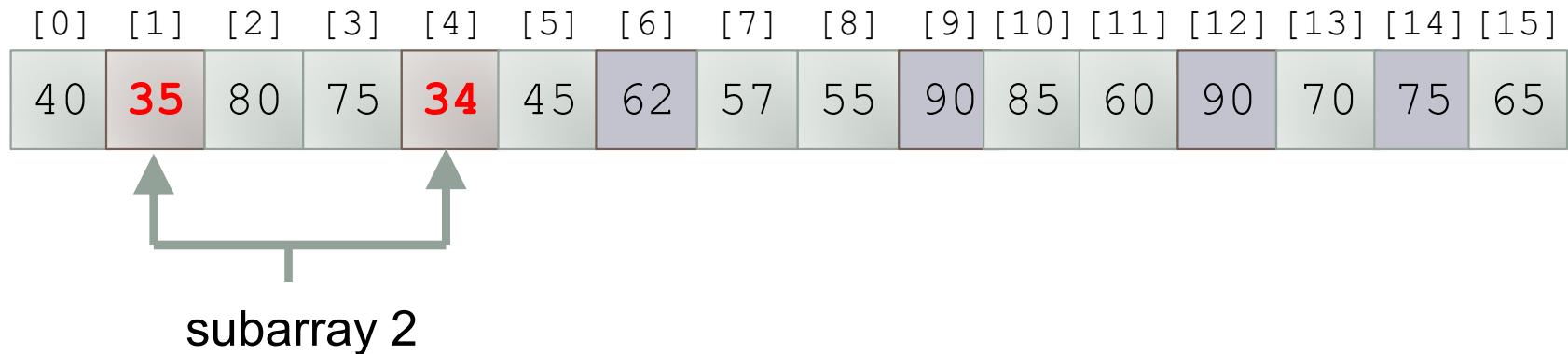
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

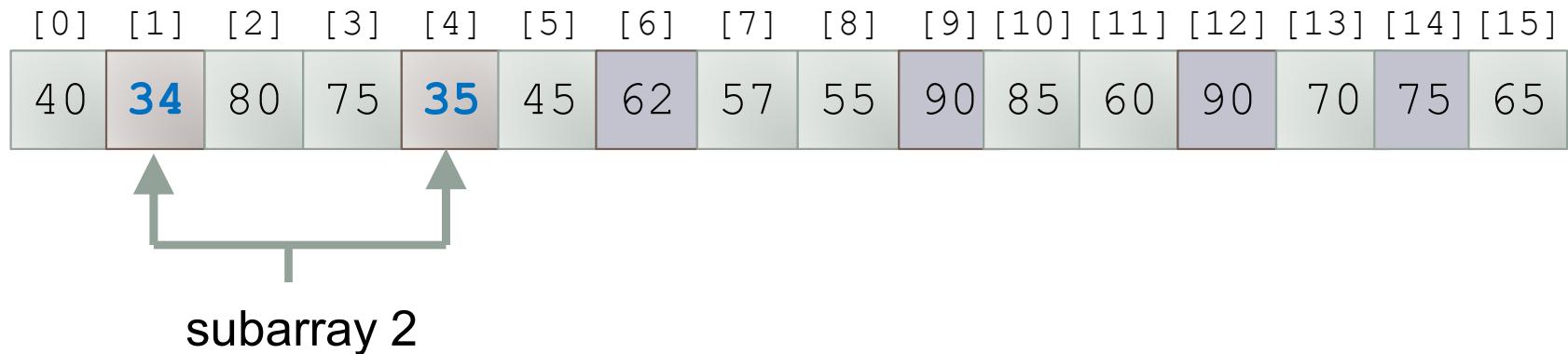
Sort subarray 2



Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 2



Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	80	75	35	45	62	57	55	90	85	60	90	70	75	65



Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 3

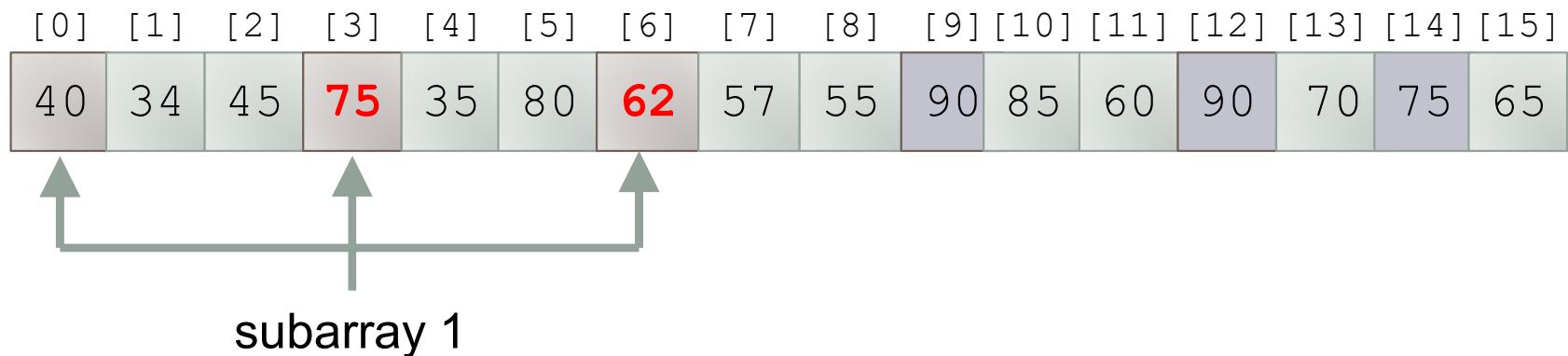
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	75	35	80	62	57	55	90	85	60	90	70	75	65



Trace of Shell Sort (cont.)

gap value	3
-----------	---

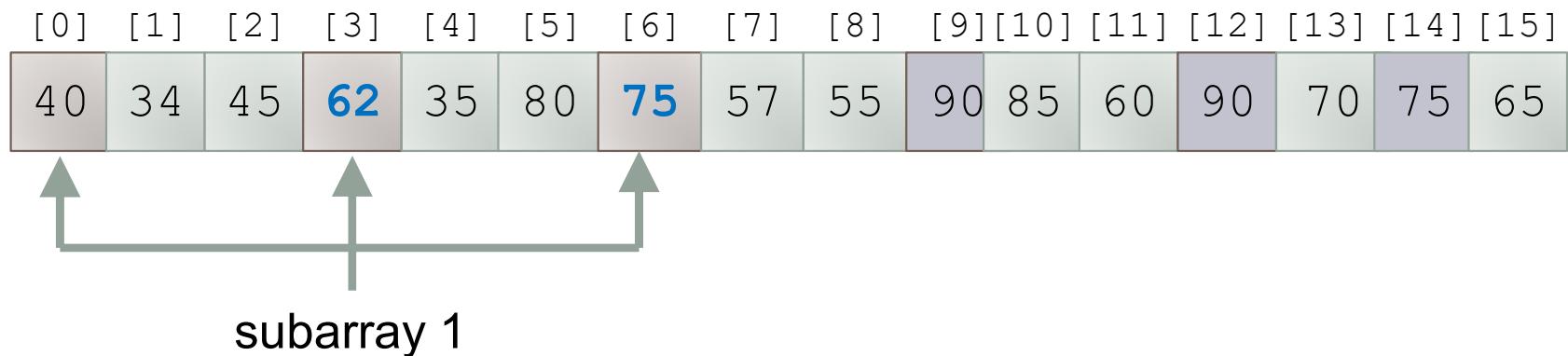
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 1

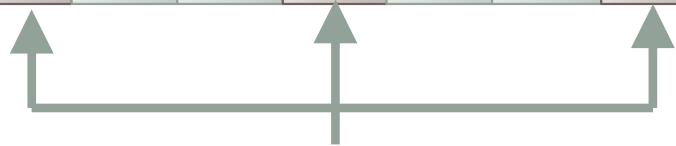


Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	80	75	57	55	90	85	60	90	70	75	65



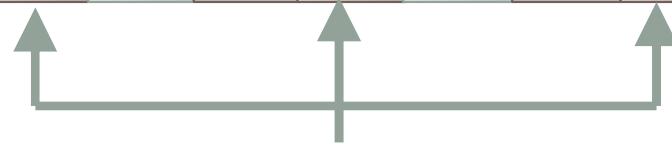
subarray 2

Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	80	75	57	55	90	85	60	90	70	75	65



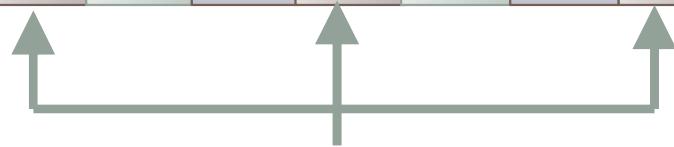
subarray 3

Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	55	75	57	80	90	85	60	90	70	75	65

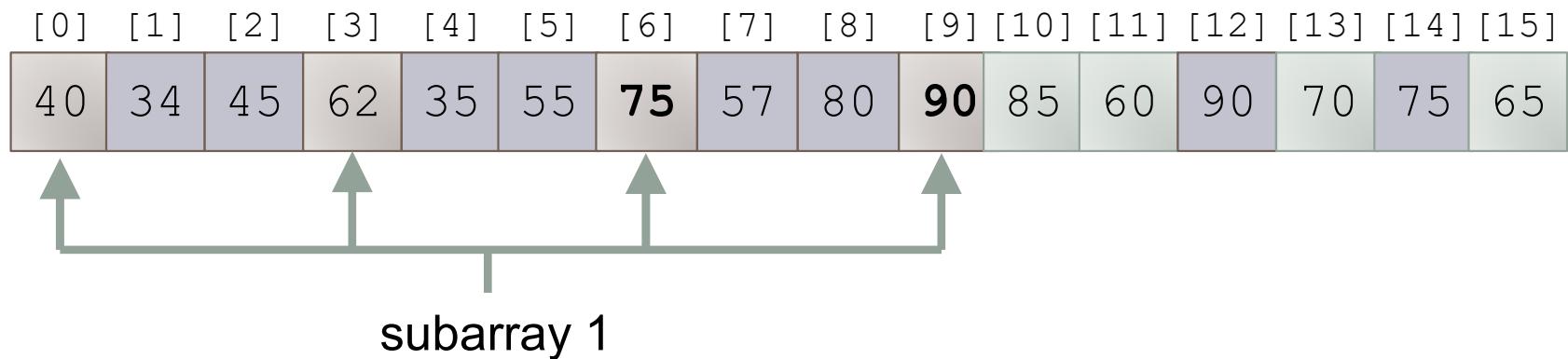


subarray 3

Trace of Shell Sort (cont.)

gap value	3
-----------	---

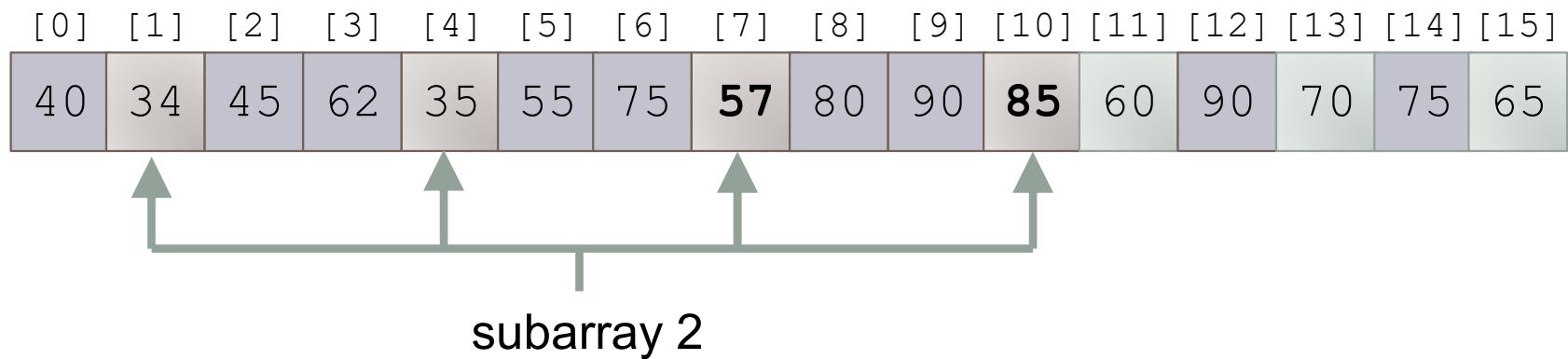
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

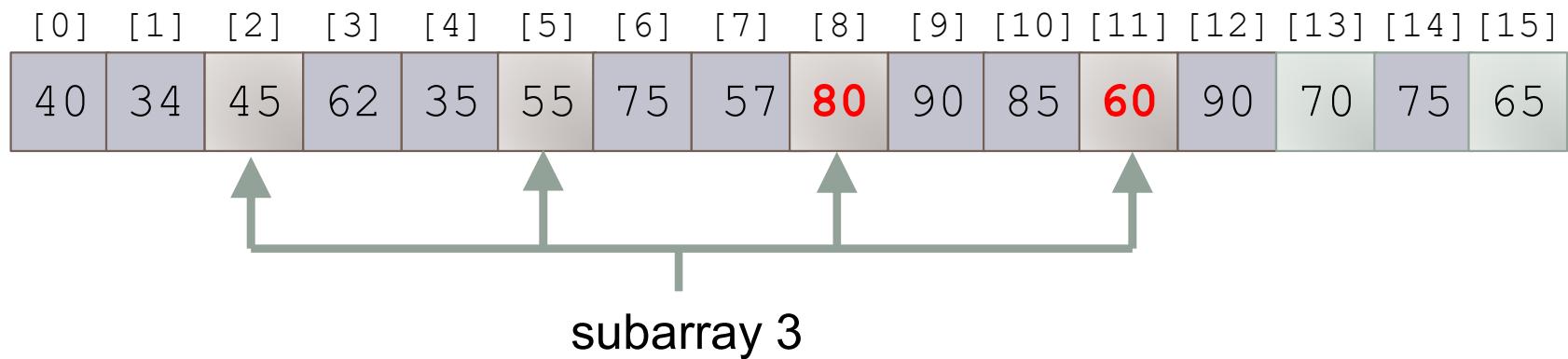
Sort subarray 2



Trace of Shell Sort (cont.)

gap value	3
-----------	---

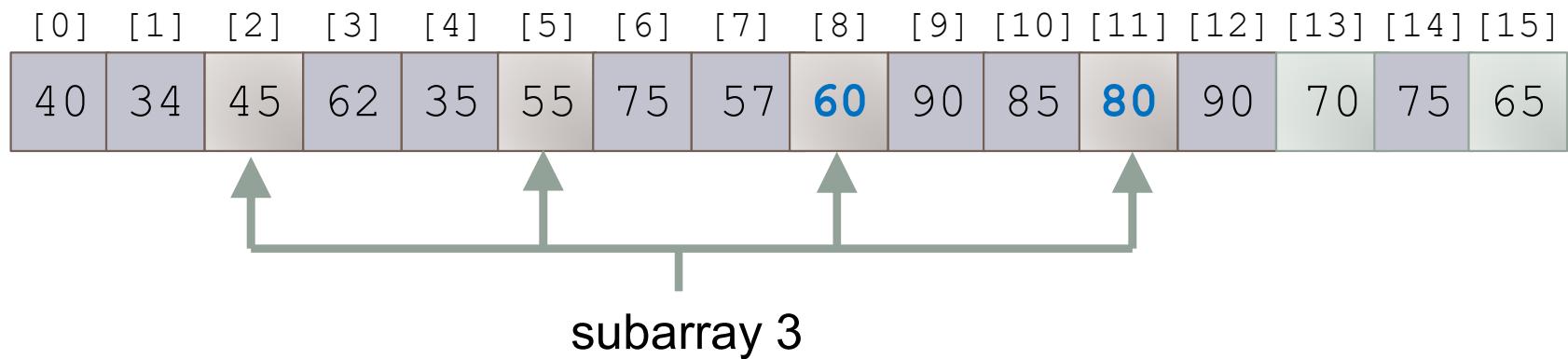
Sort subarray 3



Trace of Shell Sort (cont.)

gap value	3
-----------	---

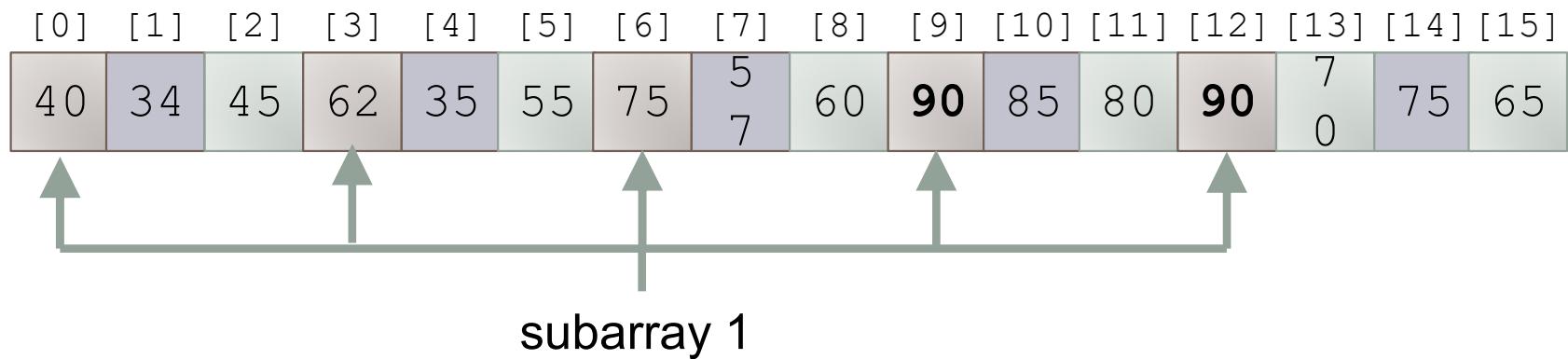
Sort subarray 3



Trace of Shell Sort (cont.)

gap value	3
-----------	---

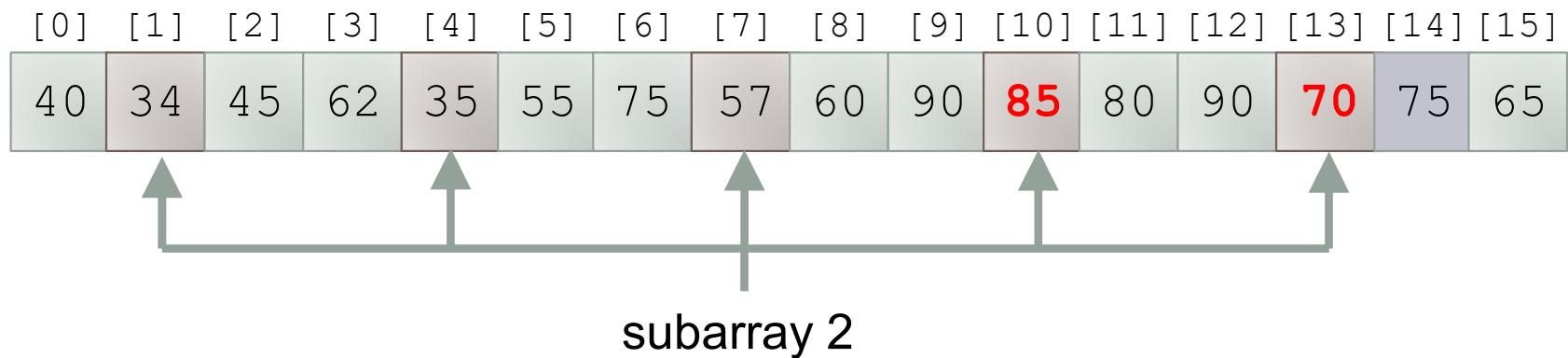
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

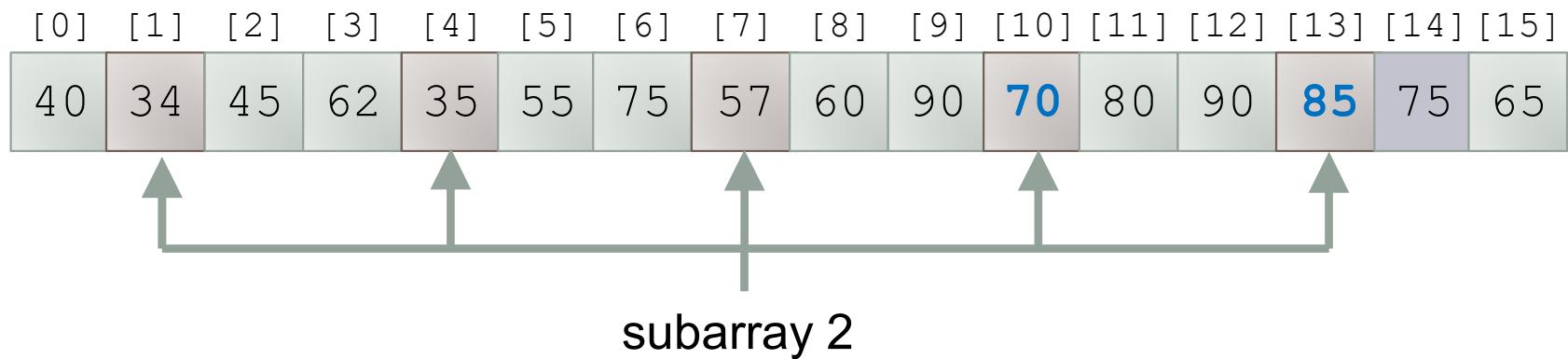
Sort subarray 2



Trace of Shell Sort (cont.)

gap value	3
-----------	---

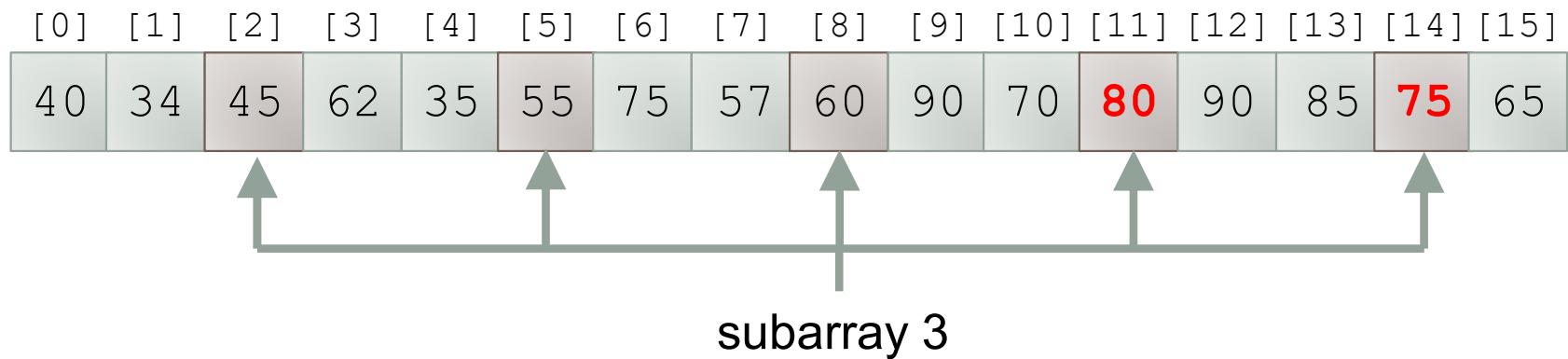
Sort subarray 2



Trace of Shell Sort (cont.)

gap value	3
-----------	---

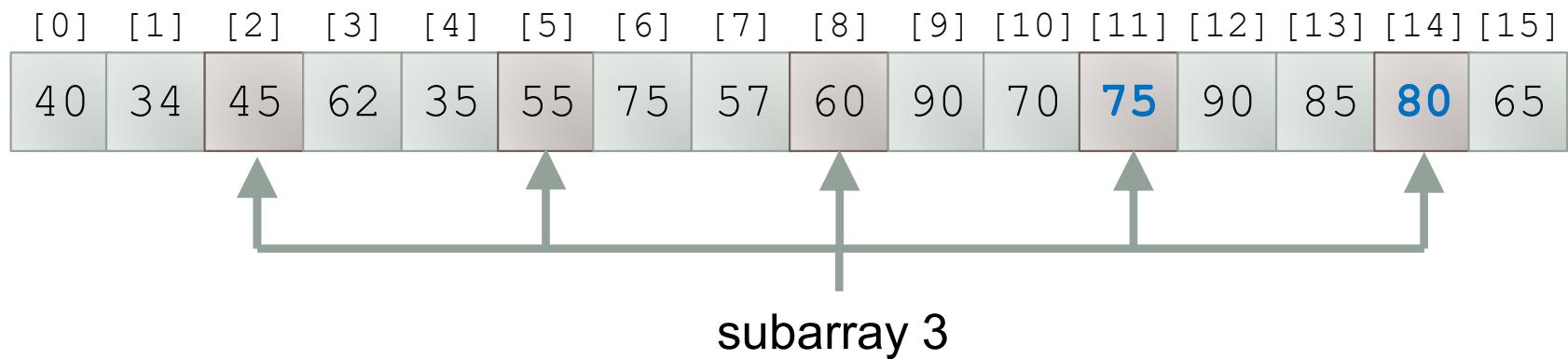
Sort subarray 3



Trace of Shell Sort (cont.)

gap value	3
-----------	---

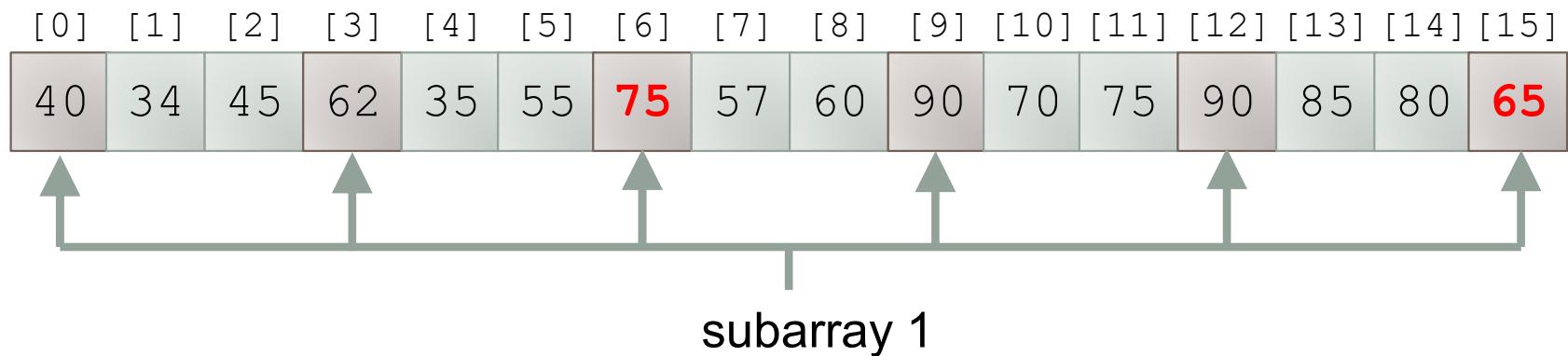
Sort subarray 3



Trace of Shell Sort (cont.)

gap value	3
-----------	---

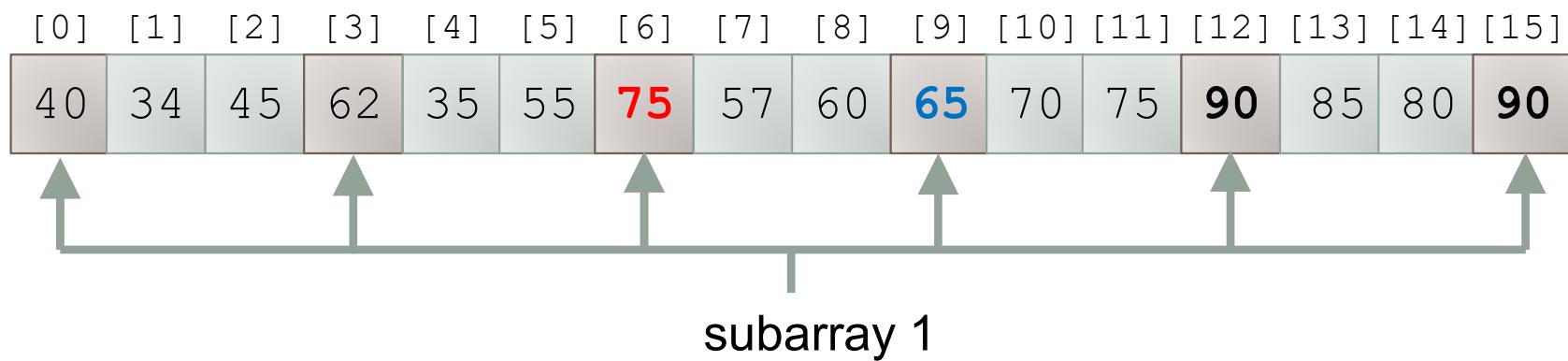
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

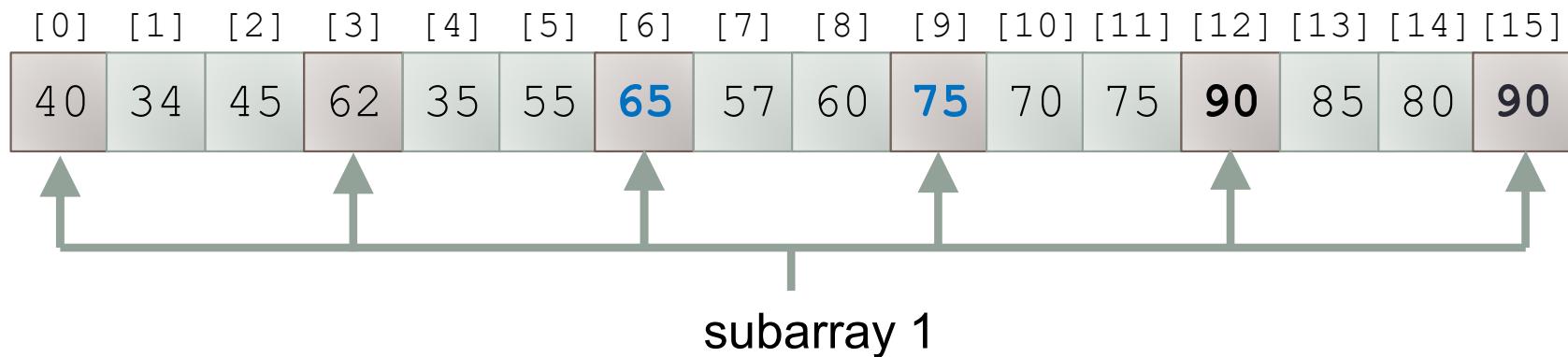
Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3
-----------	---

Sort subarray 1



Trace of Shell Sort (cont.)

gap value **1**

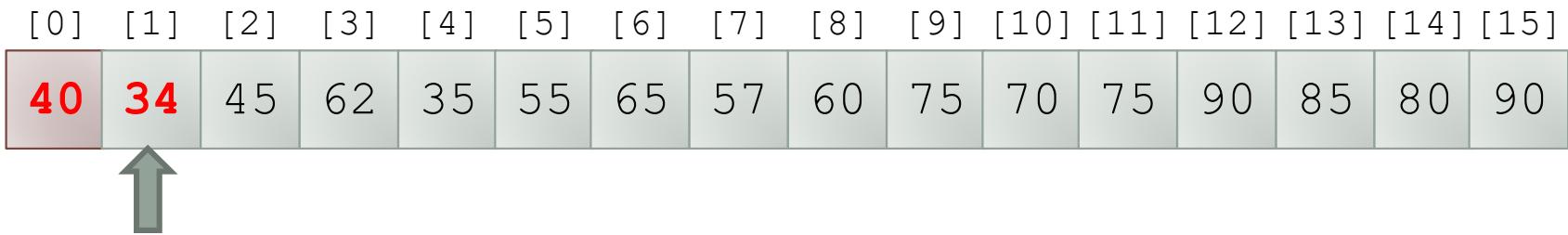
Sort on **gap value** of **1**
(a regular insertion sort)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	55	65	57	60	75	70	75	90	85	80	90

Trace of Shell Sort (cont.)

gap value	1
-----------	---

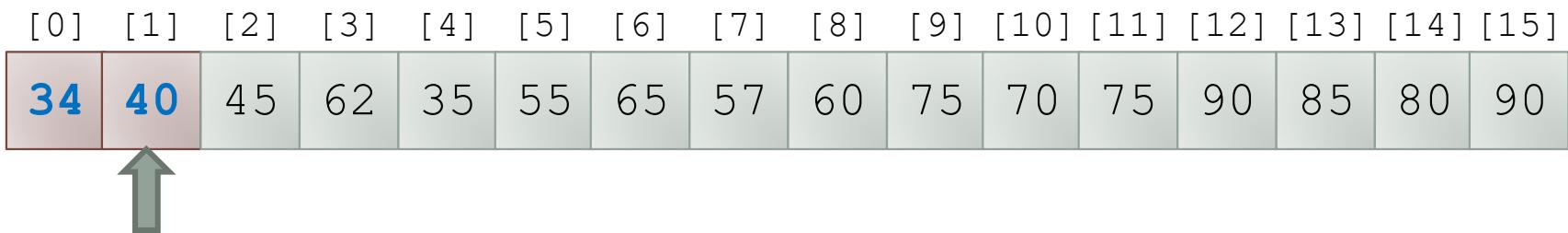
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

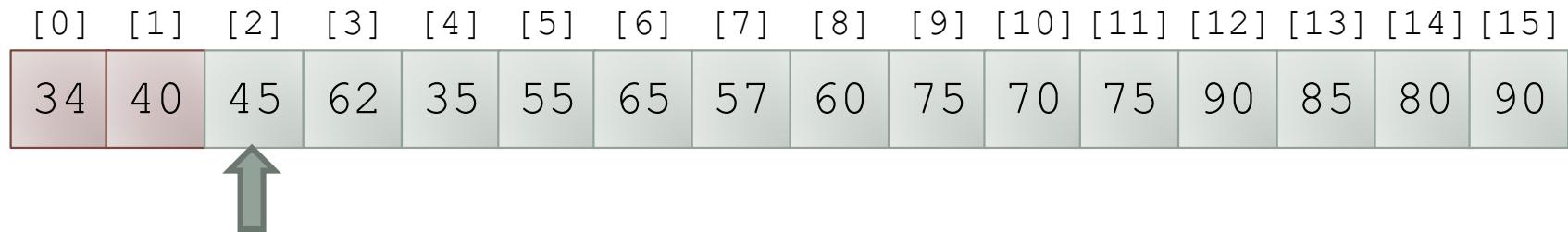
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

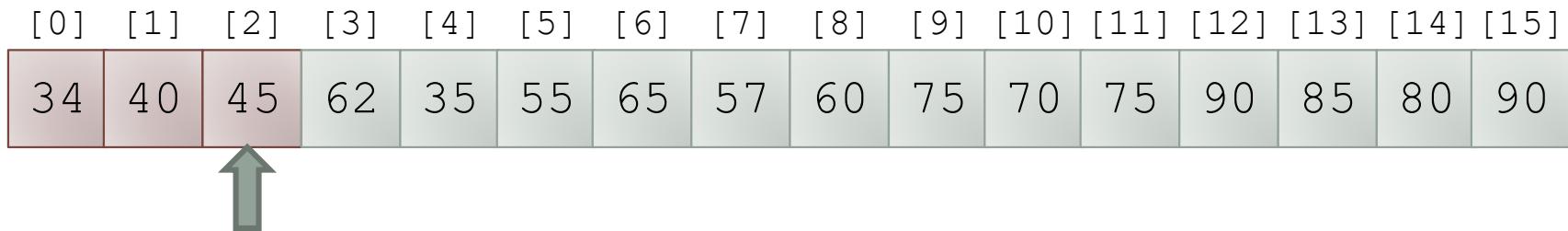
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

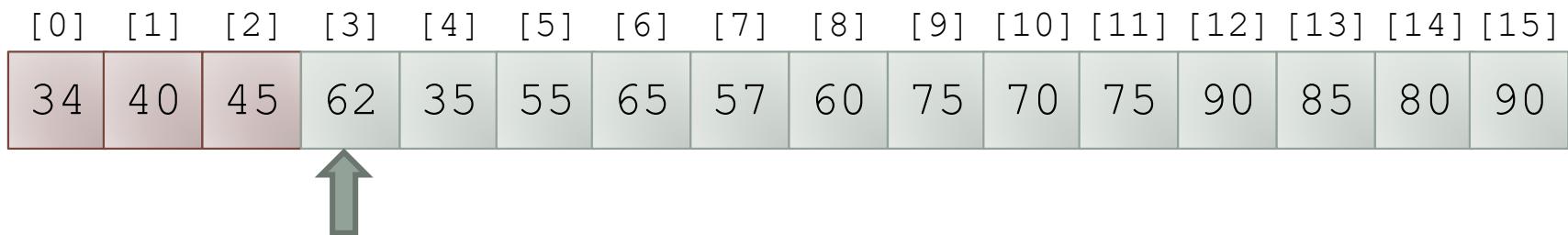
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

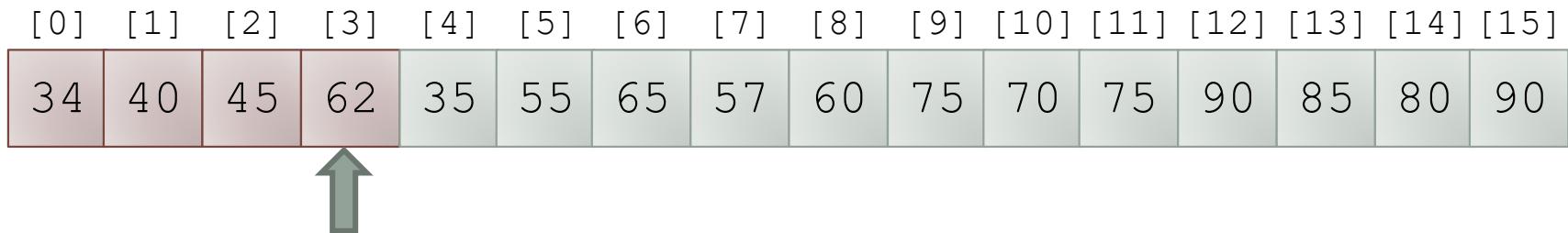
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

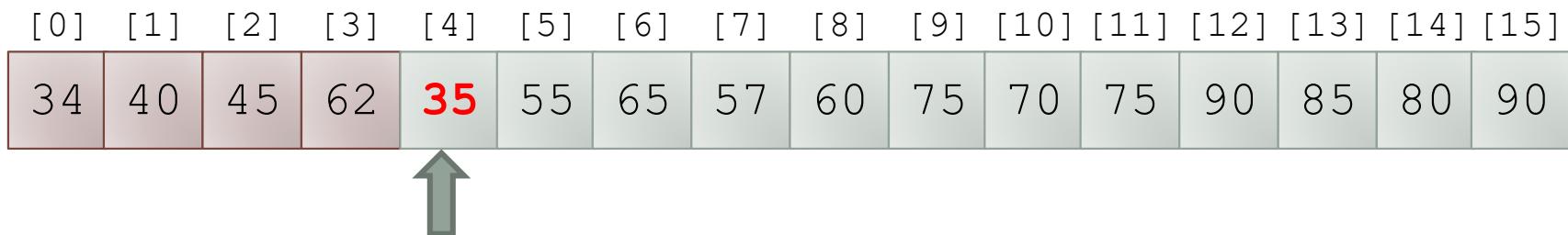
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

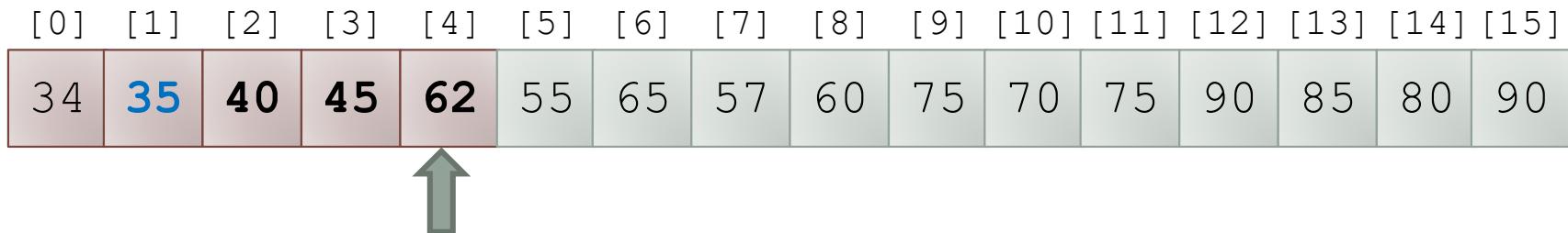
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

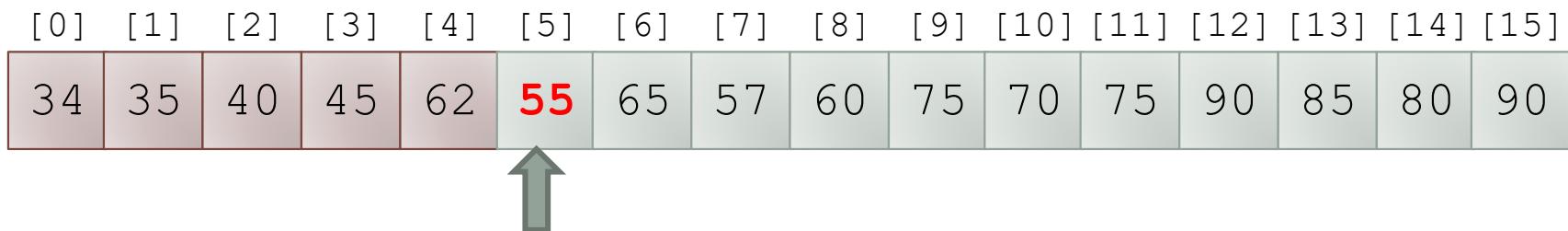
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

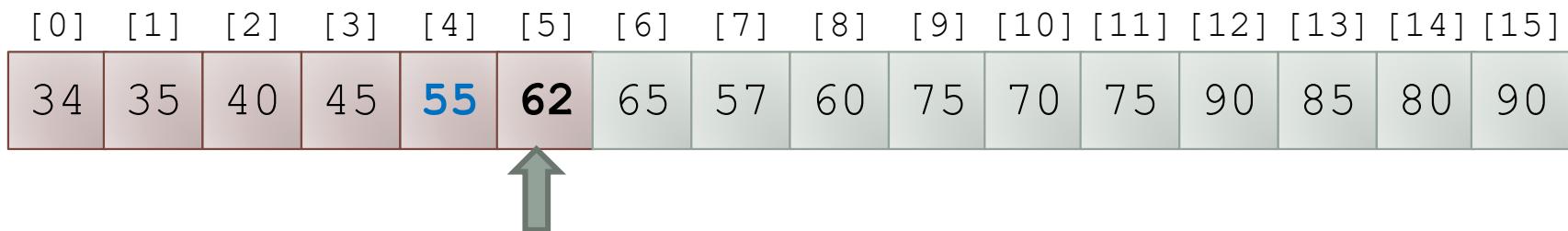
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

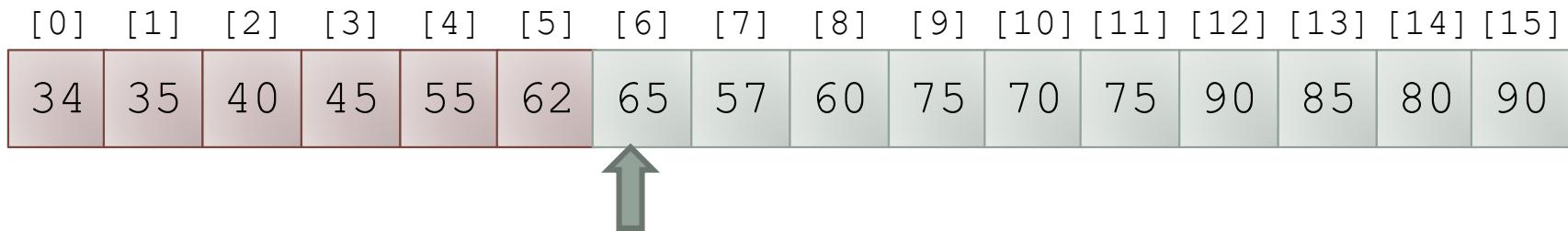
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

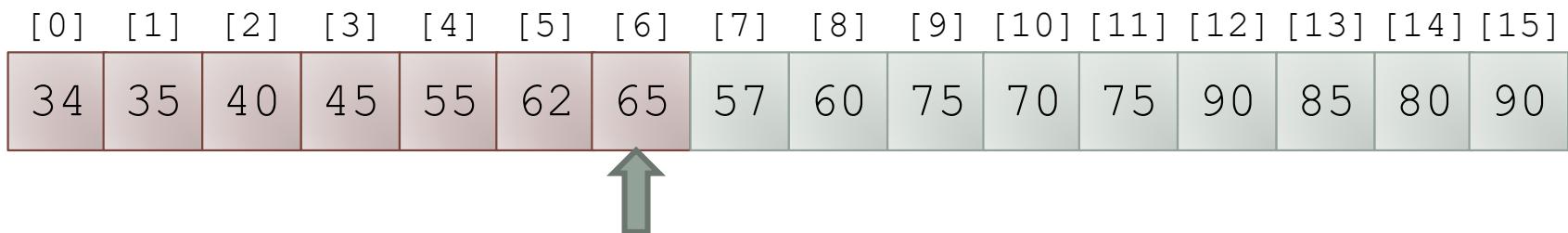
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

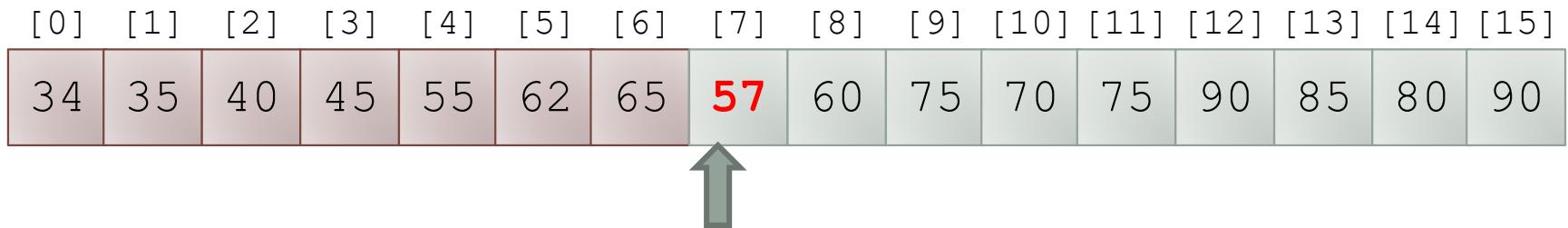
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

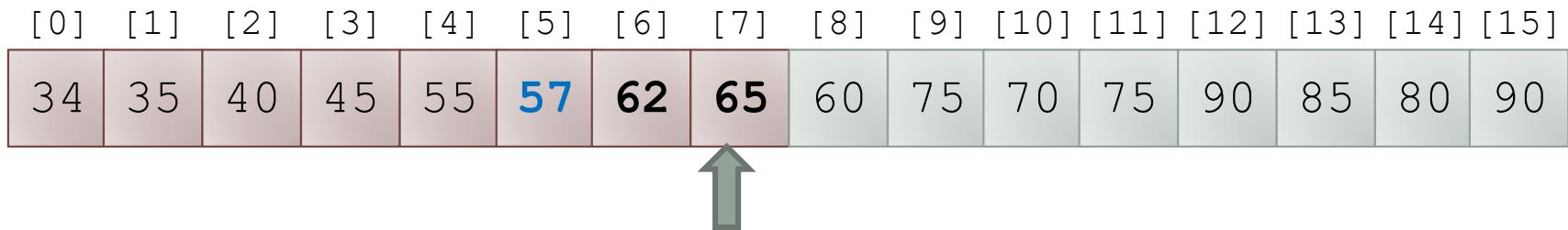
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

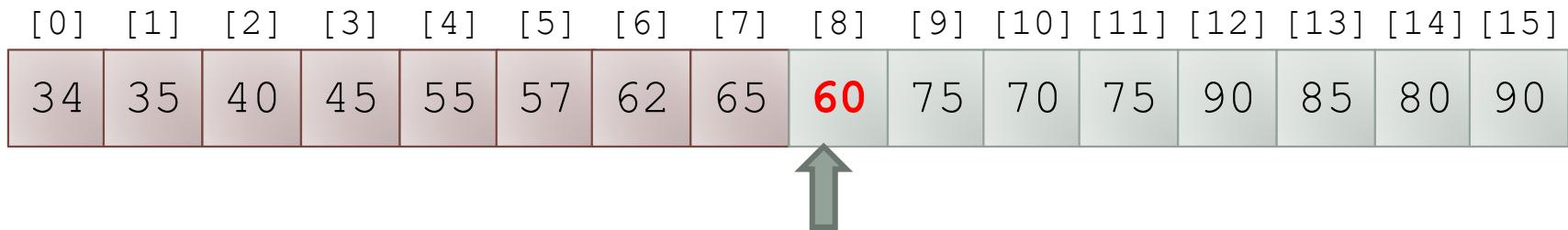
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

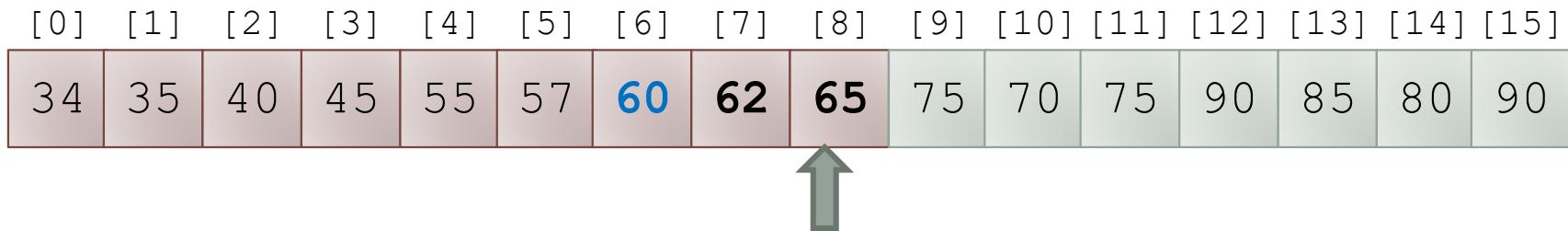
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

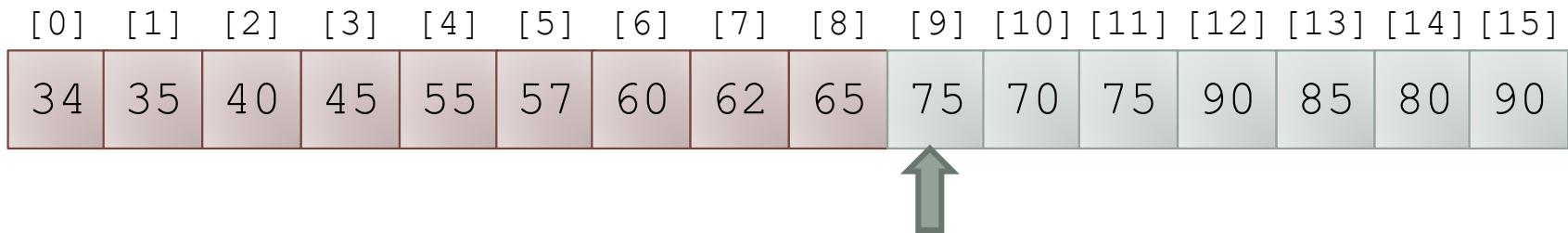
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

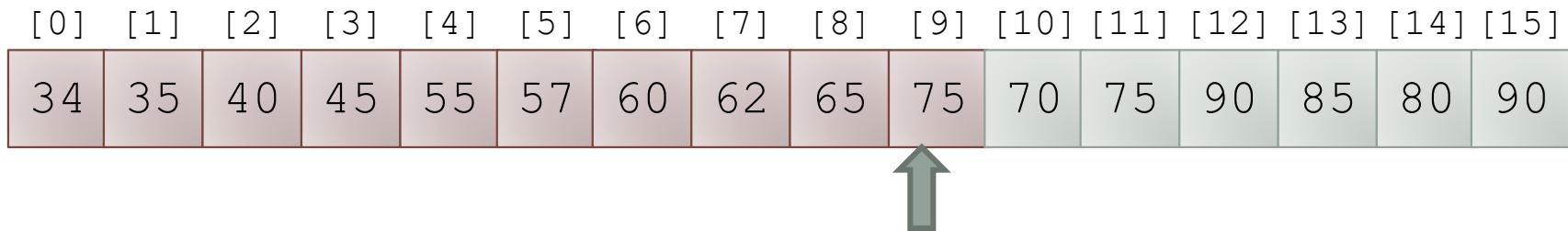
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

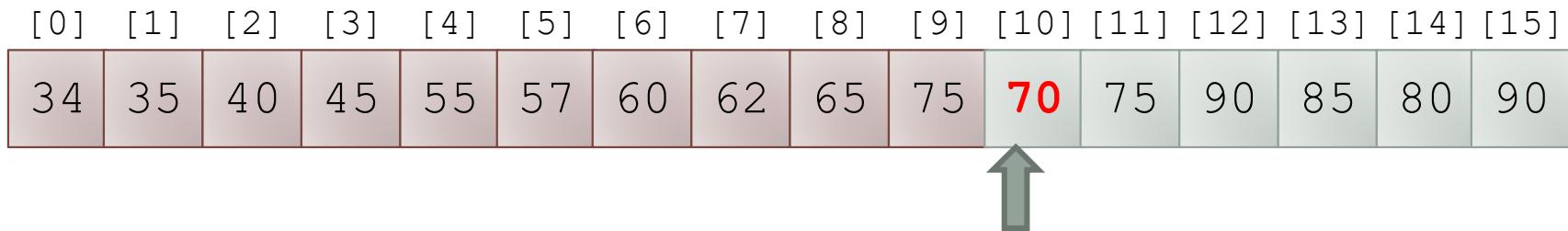
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

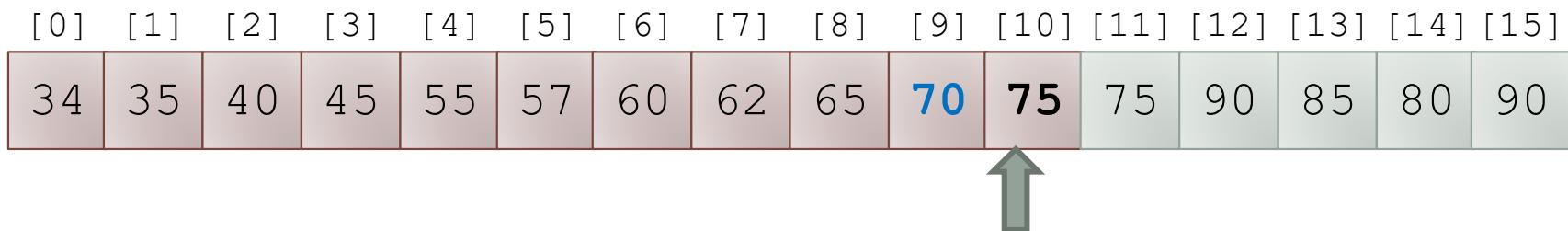
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

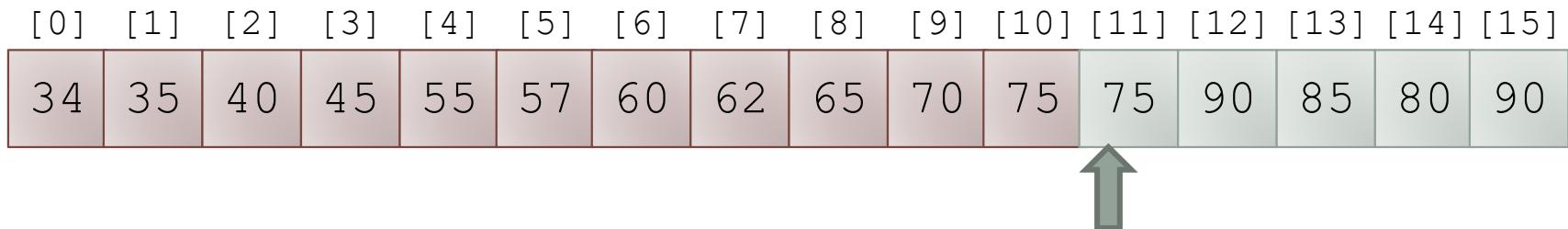
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

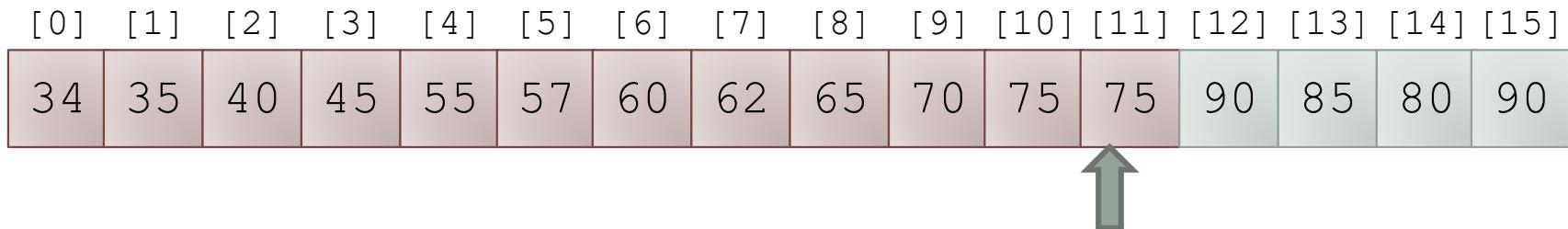
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

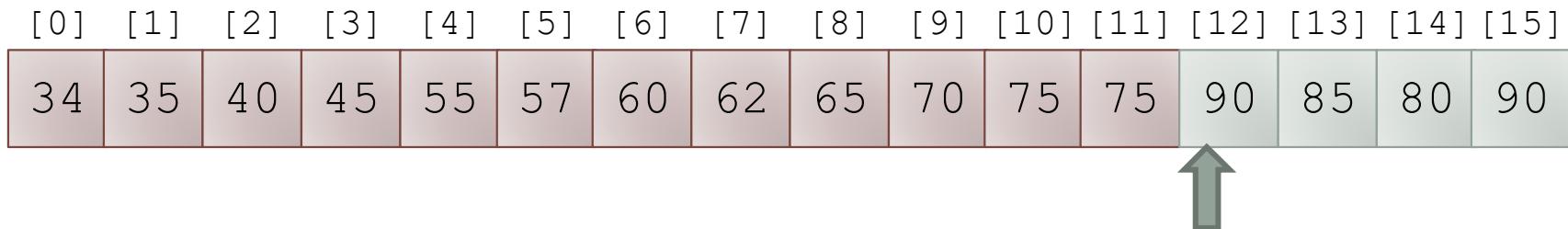
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

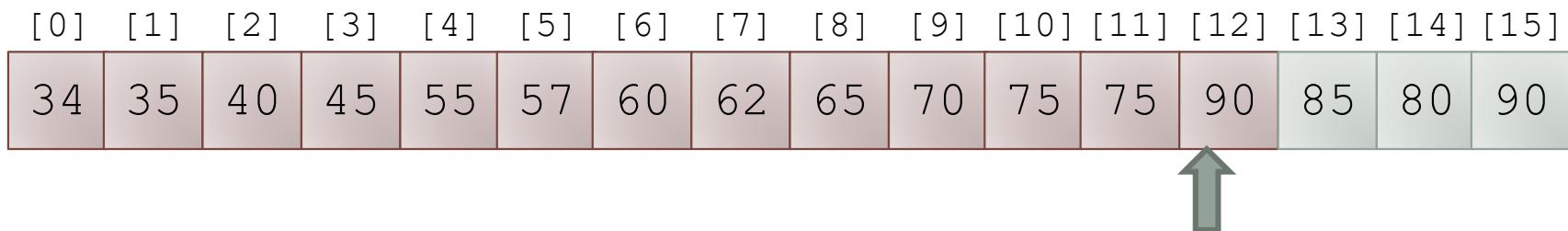
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

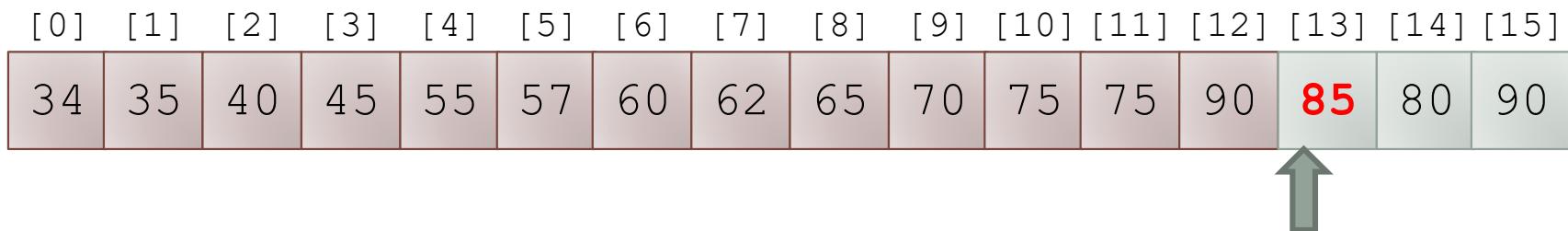
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

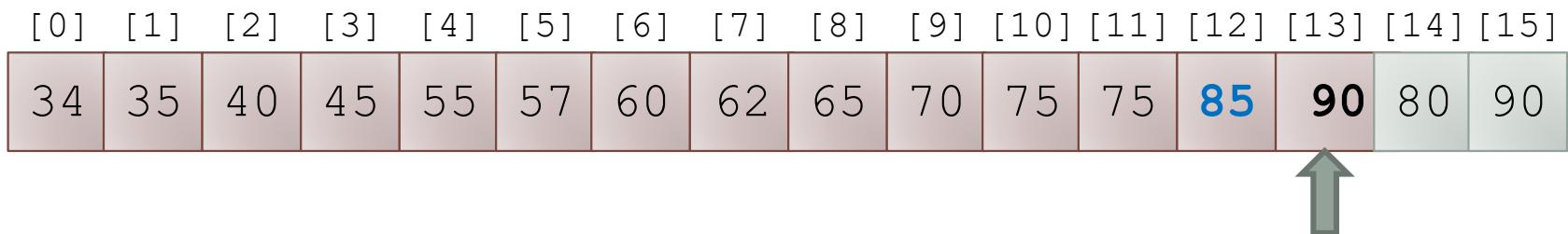
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

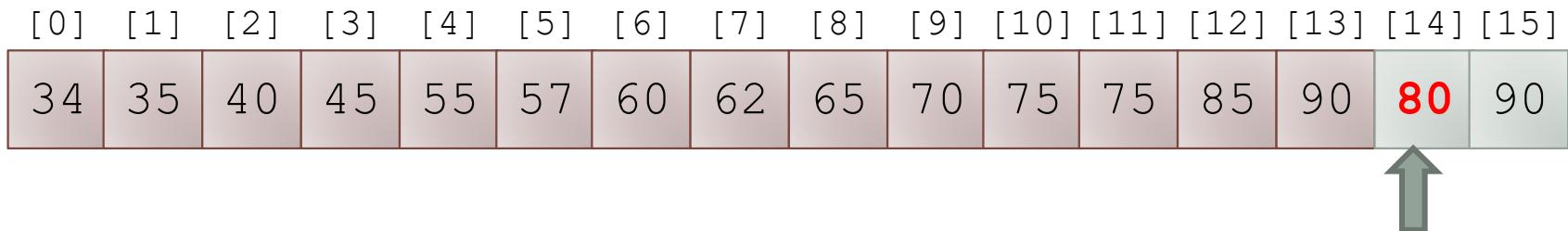
Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

Sort on gap value of 1
(a regular insertion sort)

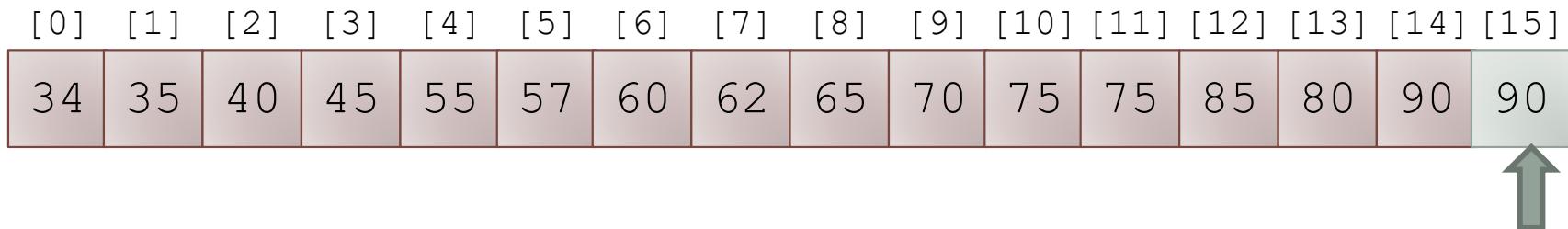
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	57	60	62	65	70	75	75	80	85	90	90



Trace of Shell Sort (cont.)

gap value	1
-----------	---

Sort on gap value of 1
(a regular insertion sort)



Trace of Shell Sort (cont.)

gap value	1
-----------	---

Sort on gap value of 1
(a regular insertion sort)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	57	60	62	65	70	75	75	85	80	90	90

Shell Sort Algorithm

Shell Sort Algorithm

1. Set the **initial value of gap** to $n / 2$
2. **while** $gap > 0$
 3. **for each array, element** from position **gap** to the **last** element
 4. **Insert** this element **where it belongs** in its **subarray**
 5. if gap is 2, set it to 1
 6. else $gap = gap / 2.2$ *// chosen by experimentation*

Refinement of Step 4, the Insertion Step

- 4.1 **nextPos** is the **position** of the element **to insert**
- 4.2 **Save the value** of the element **to insert** in **nextVal**
- 4.3 **while** **nextPos>gap** and the element at **nextPos-gap > nextVal**
 - 4.4 **Shift** the element at **nextPos-gap** to position **nextPos**
 - 4.5 **Decrement** **nextPos** **by** **gap**
 - 4.6 **Insert** **nextVal** at **nextPos**

Analysis of Shell Sort

- ❑ Because the behavior of insertion sort is **closer to $O(n)$** than $O(n^2)$ **when an array is nearly sorted**
→ **presorting speeds up** later sorting
- ❑ This is **critical** when **sorting large arrays** where the $O(n^2)$ performance becomes significant

Analysis of Shell Sort (cont.)

- A general analysis of Shell sort is **an open research problem** in computer science
- **Performance depends** on **how** the decreasing sequence of values for **gap** is **chosen**
 - If successive powers of 2, 2^k , are used for **gap** → **$O(n^2)$**
 - If successive values for **gap** are based on *Hibbard's sequence*,
 $2^k - 1$ (i.e. 31, 15, 7, 3, 1) → **$O(n^{3/2})$**

MERGE SORT

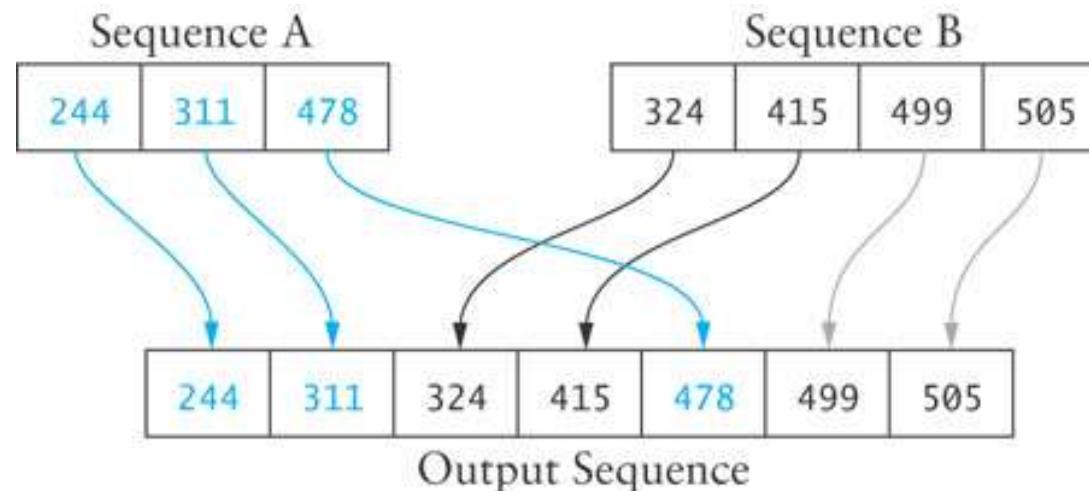
Merge

- A merge is a common **data processing operation** performed on **2 sequences** of **data** with the following characteristics
 - Both sequences contain **items** with a common **compareTo()**
 - The **objects** in both sequences are **pre-ordered** in accordance with this **compareTo()**
- The **result** is a **third sequence** containing **all** the **data** from the first two sequences

Merge Algorithm

Merge Algorithm

1. Access the **first** item **from both** sequences.
2. **while** not finished with **either** sequence
3. **Compare** the current items from the two sequences, **copy** the **smaller** current item to the **output sequence**, and **access** the **next** item **from** the input **sequence whose item was copied**.
4. **Copy** any **remaining** items **from** the **first** sequence to the output sequence.
5. **Copy** any **remaining** items **from** the **second** sequence to the output sequence.



Analysis of Merge

- ❑ For 2 input **sequences** each containing **n** elements, each element needs to **move** from its input sequence to the output sequence → Merge time is **$O(n)$**
- ❑ **Space** requirements
 - ✓ The array cannot be merged in place
 - ✓ **Additional space** usage is **$O(n)$**

dest allows the merge result to start at a position > 0 in the output array

```
private static <T extends Comparable<T>> void merge(T[] outputSequence, int dest;  
                                         T[] leftSequence, T[] rightSequence)  
{  
    int i = 0; // Index into the left input sequence.  
    int j = 0; // Index into the right input sequence.  
    int k = dest; // Index into the output sequence.  
    // While there is data in both input sequences  
    while (i < leftSequence.length && j < rightSequence.length) {  
        // Find the smaller and insert it into the output sequence.  
        if (leftSequence[i].compareTo(rightSequence[j]) < 0) {  
            outputSequence[k++] = leftSequence[i++];  
        } else {  
            outputSequence[k++] = rightSequence[j++];  
        }  
    }  
    // Copy remaining input from left sequence into the output.  
    while (i < leftSequence.length) {  
        outputSequence[k++] = leftSequence[i++];  
    }  
    // Copy remaining input from right sequence into output.  
    while (j < rightSequence.length) {  
        outputSequence[k++] = rightSequence[j++];  
    }  
}
```

TIMSORT

Timsort

- ❑ A **modification of merge sort** that takes advantage of **sorted subsets** in an array.
 - ✓ It **identifies** sequences that are **already sorted** in ascending or descending order (called a ***run***) and does not split them, unlike merge sort.
 - ✓ developed by **Tim Peters** in **2002** as the **library sorting algorithm** for Python.
 - ✓ **Java** uses **Timsort** as the **sorting** algorithm for **arrays of Objects** and for **Lists**.

Timsort (cont.)

- ❑ Newly identified **runs** are pushed onto a **stack** and may be **merged with adjacent runs** on the stack to form **longer runs**.
- ❑ Before placing it on the stack, a **descending run** is **converted** in-place **to** an **ascending run**.
- ❑ Timsort **maintains** the **invariants** below where i is the **index** of a **new run** pushed onto a **stack**.
 1. The **run** at index $i - 2$ is **longer than** the **sum** of the **length** of the **runs** at indexes i and $i - 1$.
 2. The **run** at index $i - 1$ is **longer than** the **run** at index i
- ❑ If there are **2 runs** on the **stack** AND the **run** at index $i - 1$ is **not longer than** the **new run** at i , **merge** the **runs** to **satisfy invariant 2**.

Trace of Timsort

- The array to be sorted is shown below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	8	6	4	12	14	15	16	17	11	10

- The **first run** from [0] to [4] is **identified** and its description is **pushed** onto the **stack** at stack **index 0**. The stack entry shows the starting position of the run (0) and its length (5).

index	start	length
0	0	5

Trace of TimSort (cont.)

- The **next run** from [5] to [7] is a **descending** sequence, so it is first **converted** in-place to an **ascending** sequence

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

- and is then pushed onto the stack at index 1
(note: the run with the largest index is at the top of the stack)

index	start	length
0	0	5
1	5	3

The **new run** at 1 is **shorter than** the run at index 0 so **invariant 2** is **satisfied**

Trace of TimSort (cont.)

The next run from [8] through [12] is an ascending run.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

Its description is pushed onto the stack:

index	start	length
0	0	5
1	5	3
2	8	5

$3 + 5 > 5$

$5 > 3$

Run at 0 is **not longer than** the **sum** of the **lengths** of the **runs** at 1 and 2, and **run** at length of the **run** at 2 is **not less than** the length of the **run** at 1 → **Merge** the runs at 1 and 2.

Trace of TimSort (cont.)

The array and stack after the merge are:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

index	start	length
0	0	5
1	5	8

8 > 5

Invariant not satisfied.
Merge the runs.

Trace of TimSort (cont.)

The array and stack after the merge are:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	12	14	15	16	17	11	10

index	start	length
0	0	13

Invariant 1 is satisfied

Trace of TimSort (cont.)

The **next run** starting at position [13] is **identified** and converted to an **ascending** sequence

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	12	14	15	16	17	10	11

and pushed onto the stack:

index	start	length
0	0	12
1	13	2

2 < 12

Both invariants are **satisfied**

Trace of TimSort (cont.)

There are **no more runs**. We **merge** the runs at 0 and 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	10	11	12	14	15	16	17

this leaves us with one run and we are **done**.

index	start	length
0	0	15

Timsort Performance

- In **the worst case**, each run will be of length **2**. In this case the performance of Timsort would be **$O(n \log n)$**
- If the **initial array is already sorted**, there would only be **one run** and the performance would be **$O(n)$**
- The number of **storage locations** required is **$O(n)$**
 - ✓ n storage locations are needed for the array being sorted and n for the runs being merged

HEAPSORT

Heapsort

- ❑ Merge sort time is $O(n \log n)$ but still requires, temporarily,
n extra storage locations
- ❑ Heapsort is also $O(n \log n)$ but it can be implemented
without any additional storage
 - ✓ As its name implies, heapsort **uses** a **heap** to store the **array**

First Version of a Heapsort Algorithm

- When implemented as a **priority queue (heap)** maintains **the smallest value at the top**
- The following algorithm
 - places an **array's data** into a **heap**,
 - then **removes** each **heap item** ($O(n \log n)$) and **moves it back** into the **array**
- This version of the algorithm requires **n extra storage** locations

Heapsort Algorithm: First Version

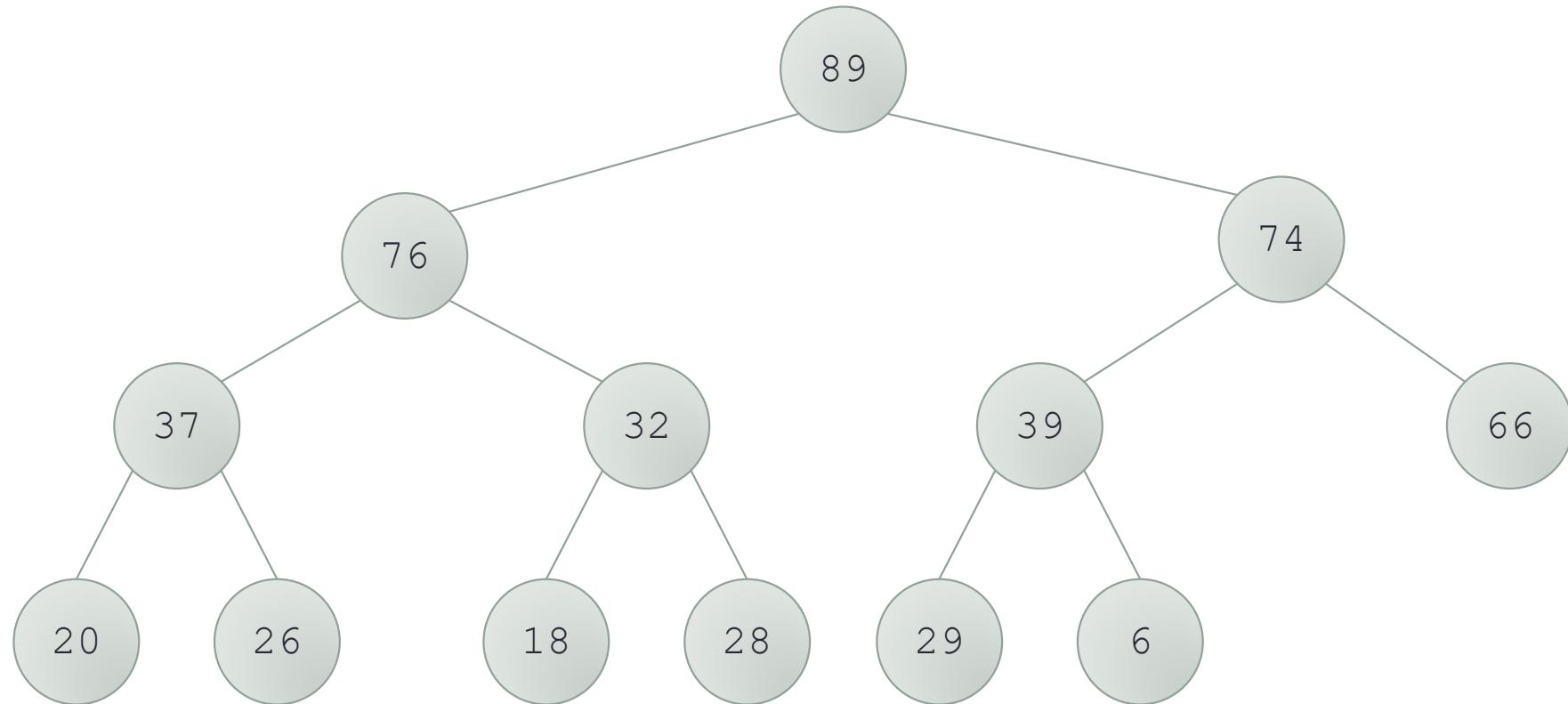
1. Insert each **value** from the **array** to be **sorted** into a **priority queue (heap)**.
2. Set ***i*** to 0
3. **while** the **priority queue** is **not empty**
 4. **Remove** an **item** from the **queue** and insert it back **into** the **array** at position ***i***
 5. Increment ***i***

In-Place Heapsort

- ❑ Although this algorithm is $O(n \log n)$, it requires *n extra storage* locations (array and heap are size *n*).
- ❑ In **heaps** we've used so far, each **parent** node value was **not greater than** the values of its **children**.
- ❑ We can implement an **in-place heapsort** by storing the **heap** in an array such that each **parent** node value is **not less than** its **children**.
- ❑ Then,
 - ✓ **exchange** the **top** item with the **one** at the **bottom** of the heap
 - ✓ **reheap**, ignoring the item moved to the bottom

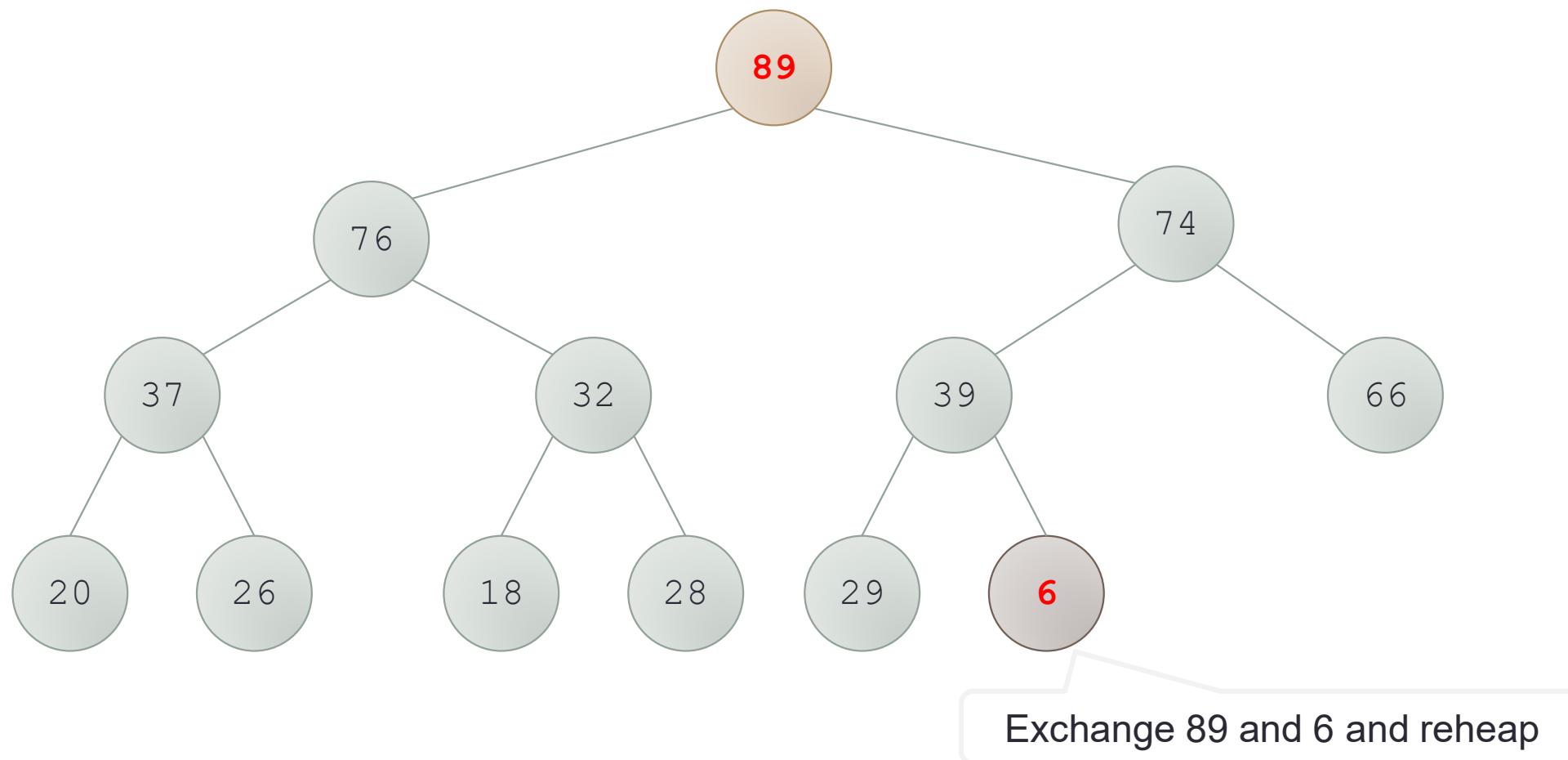
Trace of In-Place Heapsort

Initial heap representing an **unsorted array**—each item \geq its children

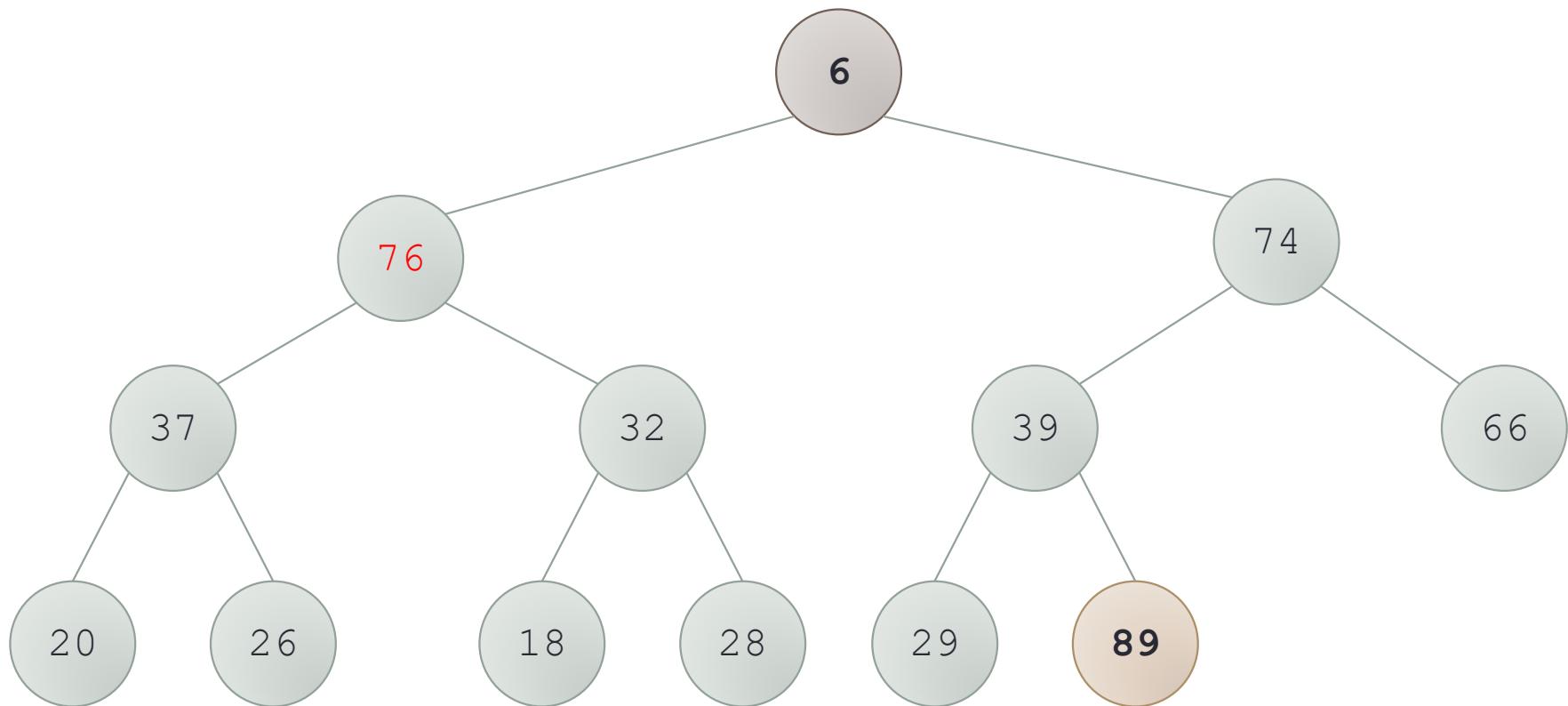


Array representation of an **unsorted array** as a **heap**:
 $\{89, 76, 74, 37, 32, 39, 66, 20, 26, 18, 28, 29, 6\}$

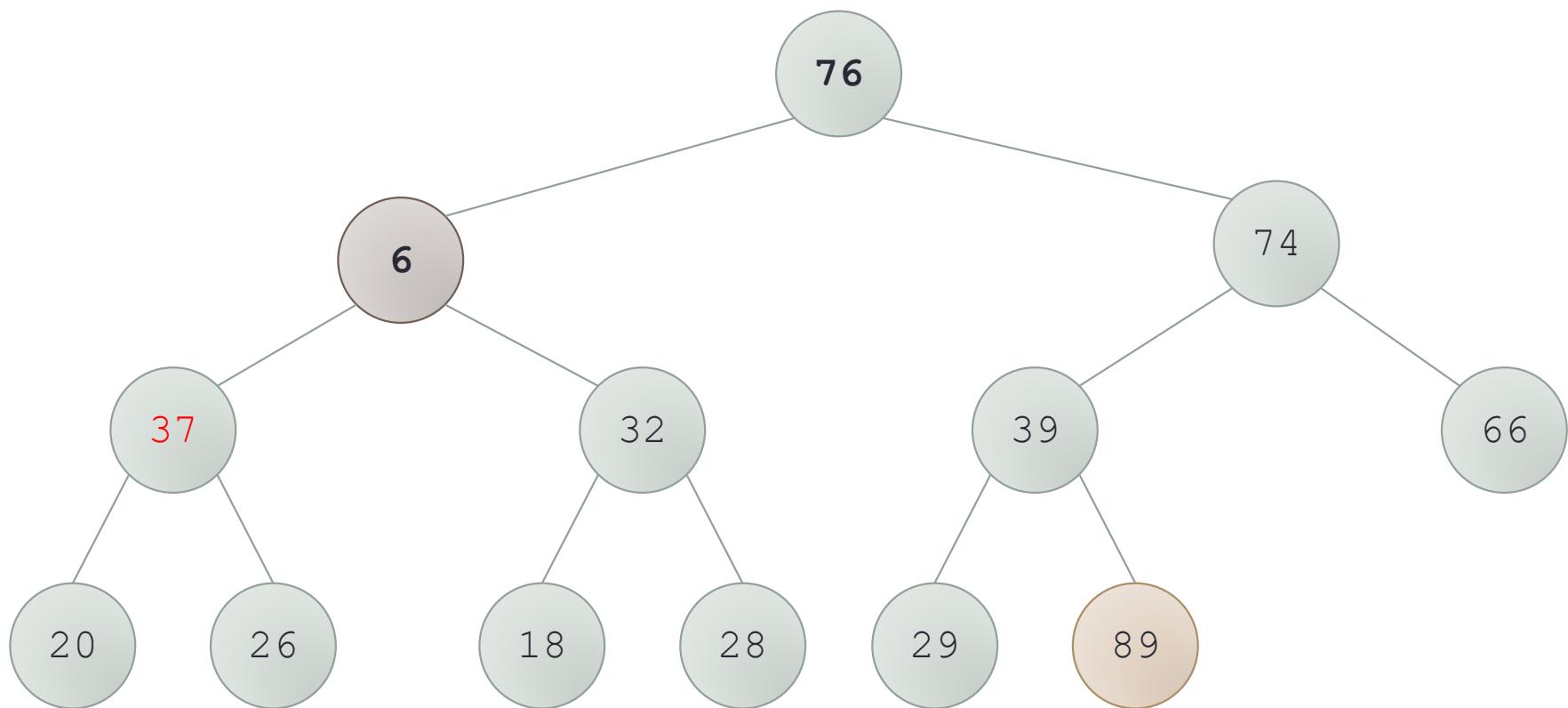
Trace of In-Place Heapsort (cont.)



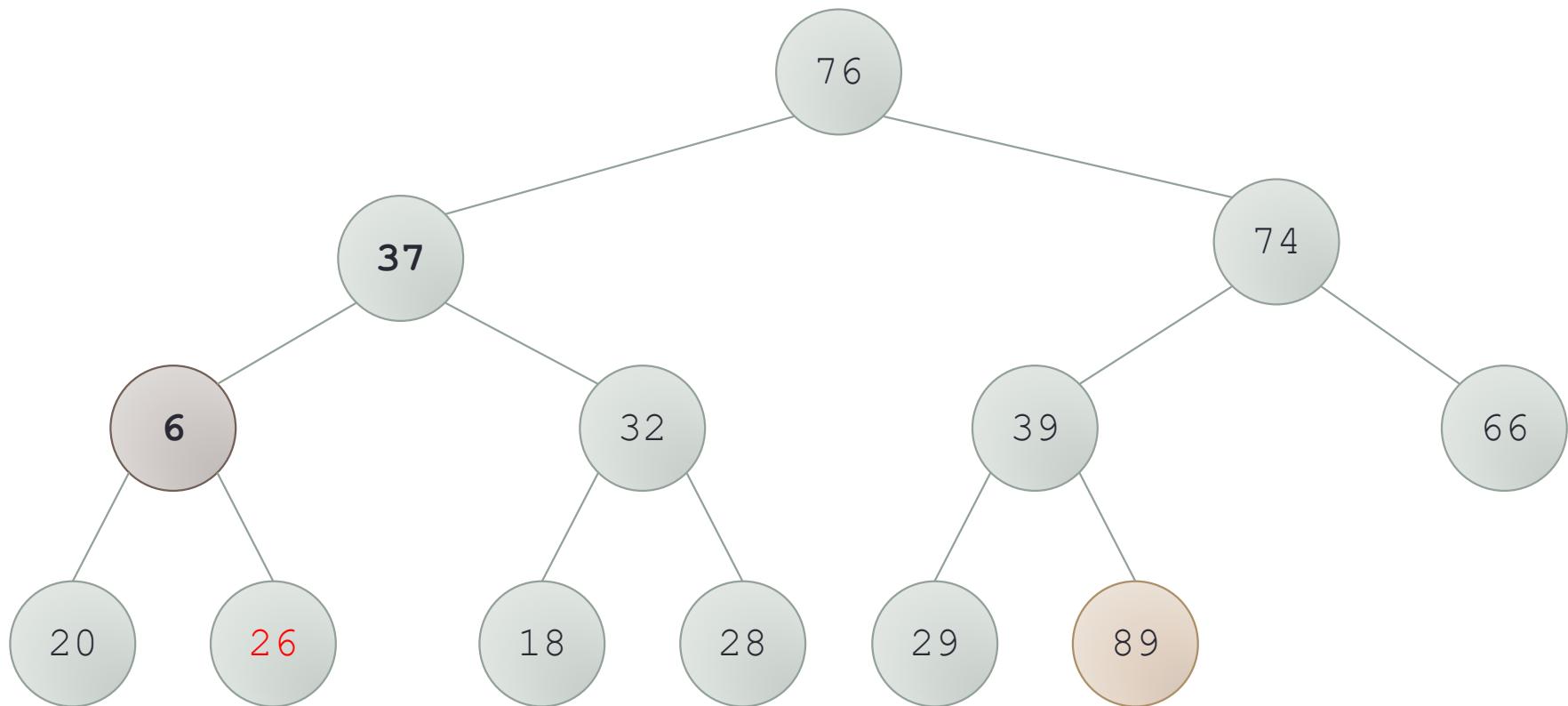
Trace of In-Place Heapsort (cont.)



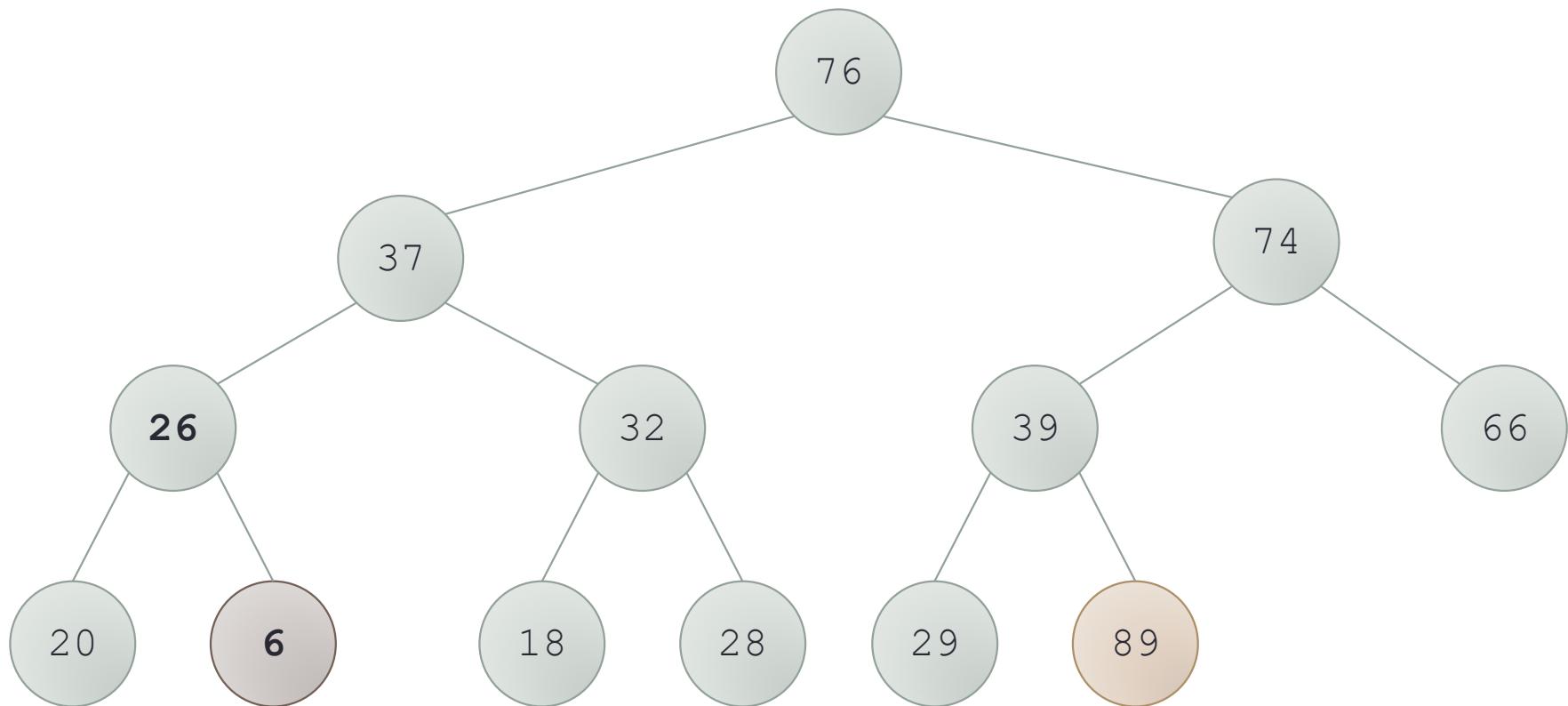
Trace of In-Place Heapsort (cont.)



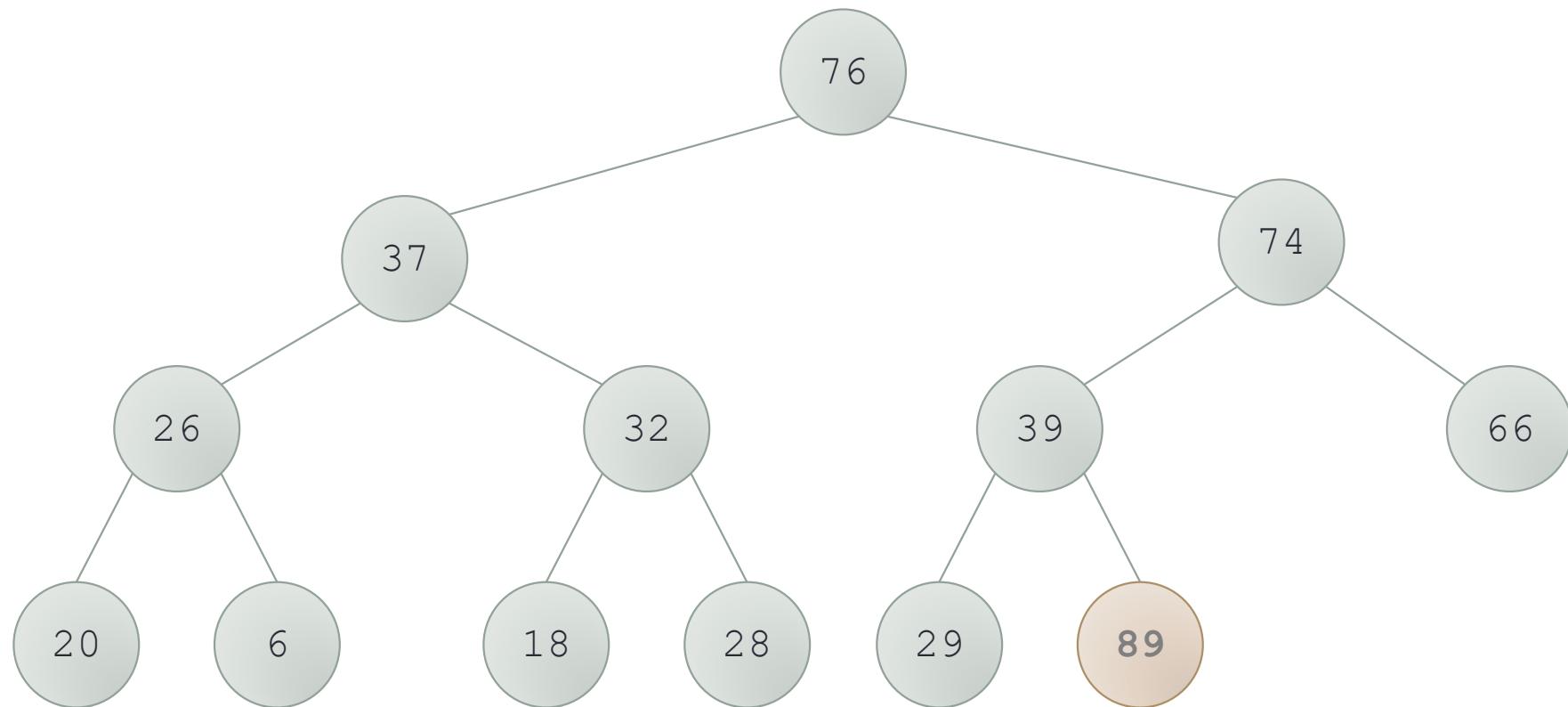
Trace of In-Place Heapsort (cont.)



Trace of In-Place Heapsort (cont.)

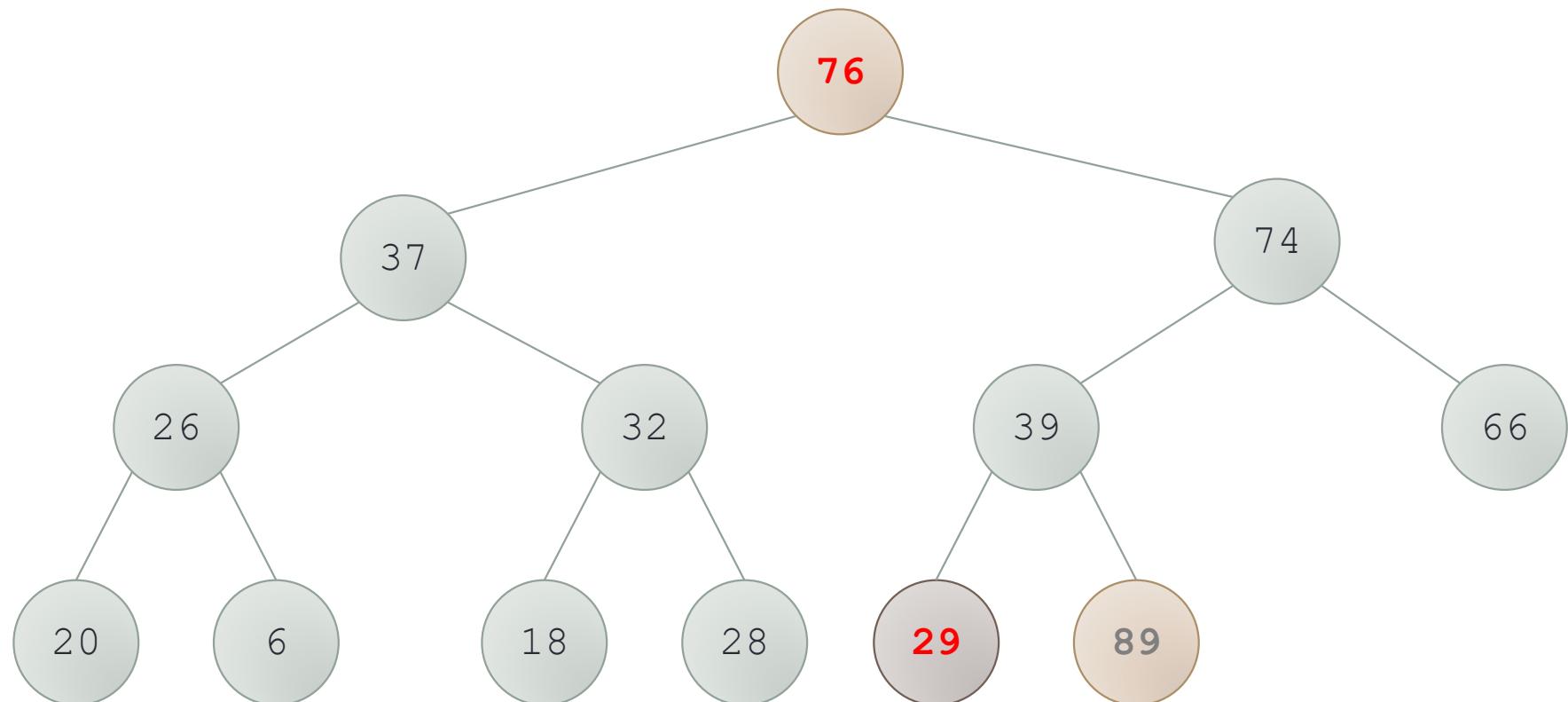


Trace of In-Place Heapsort (cont.)



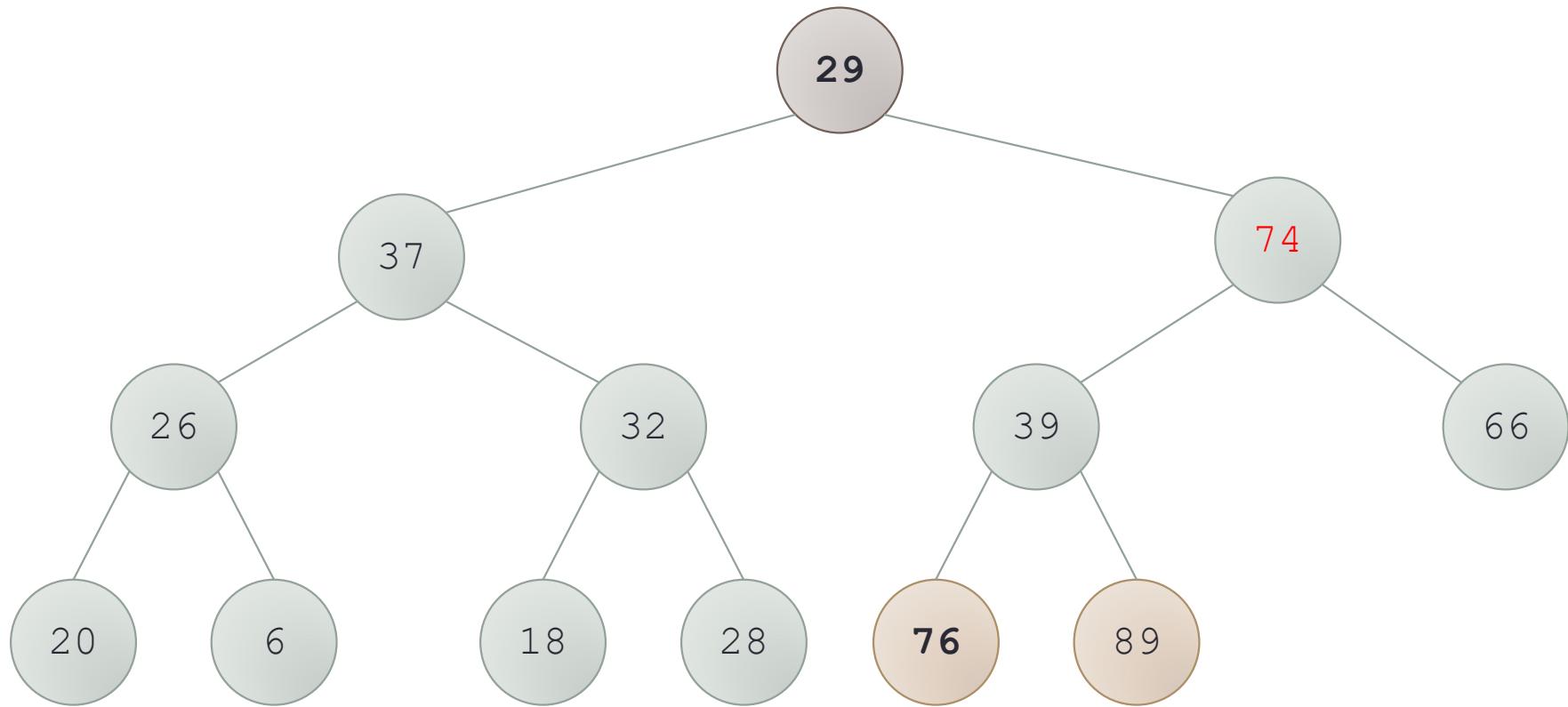
Array representation of unsorted array after largest element is removed:
{76, 37, 74, 26, 32, 39, 66, 20, 6, 18, 28, 29, 89}

Trace of In-Place Heapsort (cont.)

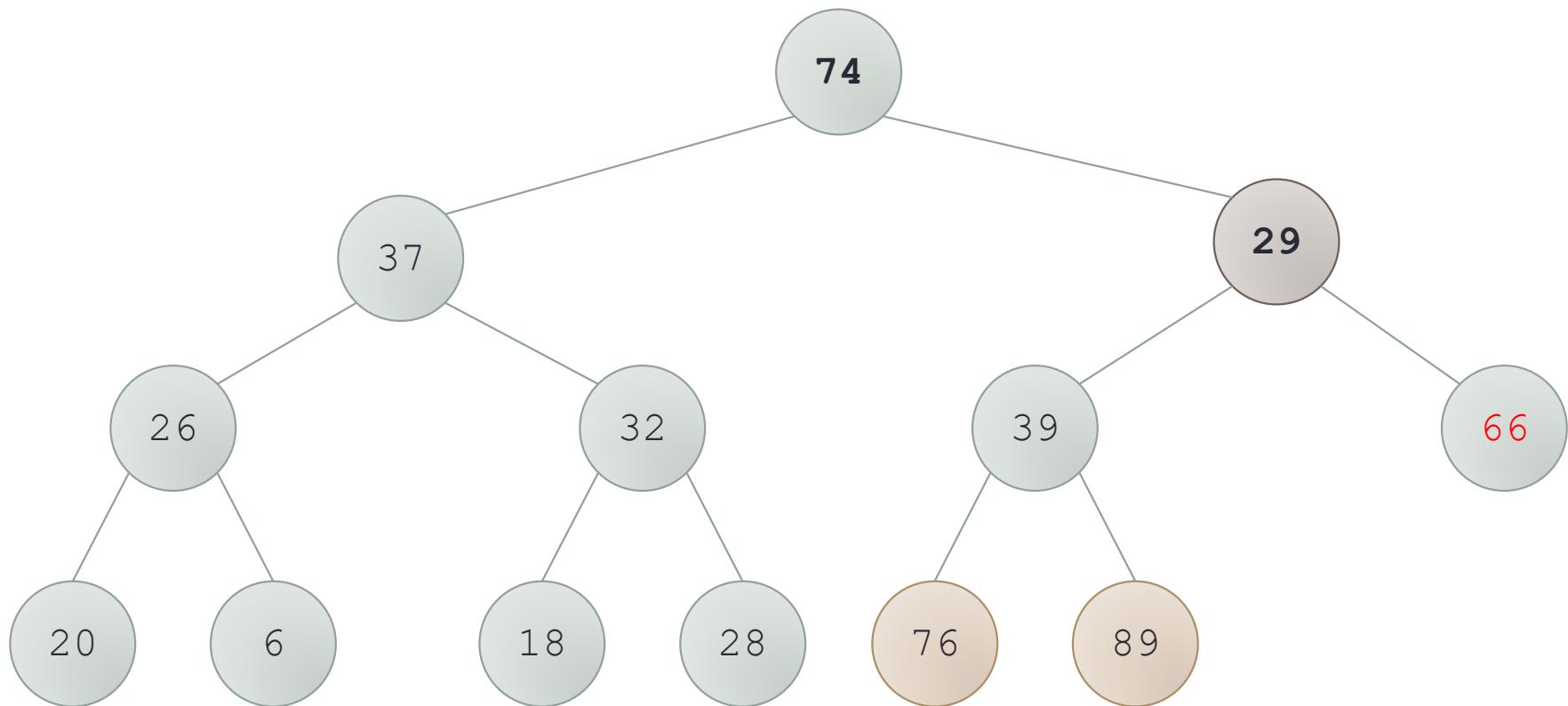


Exchange 76 and 29 and reheap

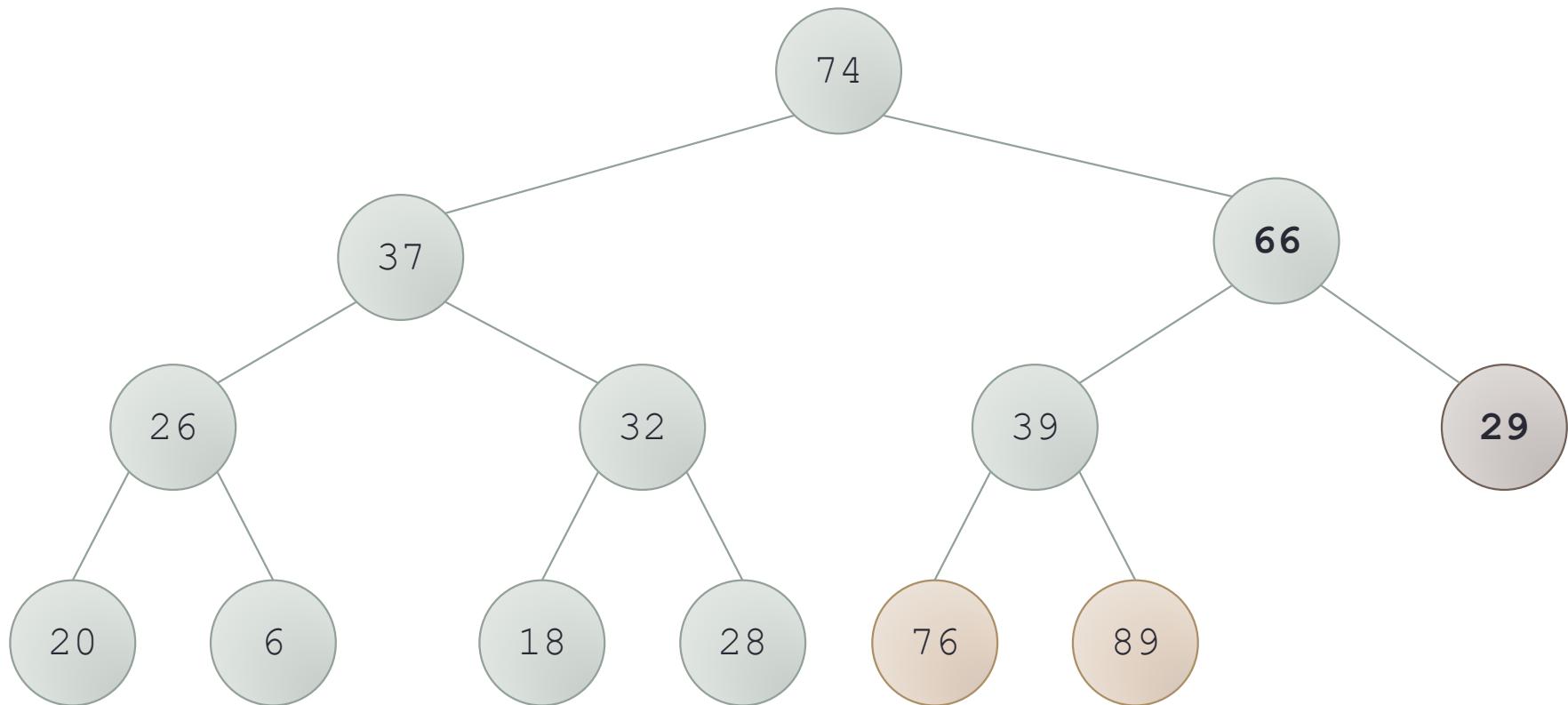
Trace of In-Place Heapsort (cont.)



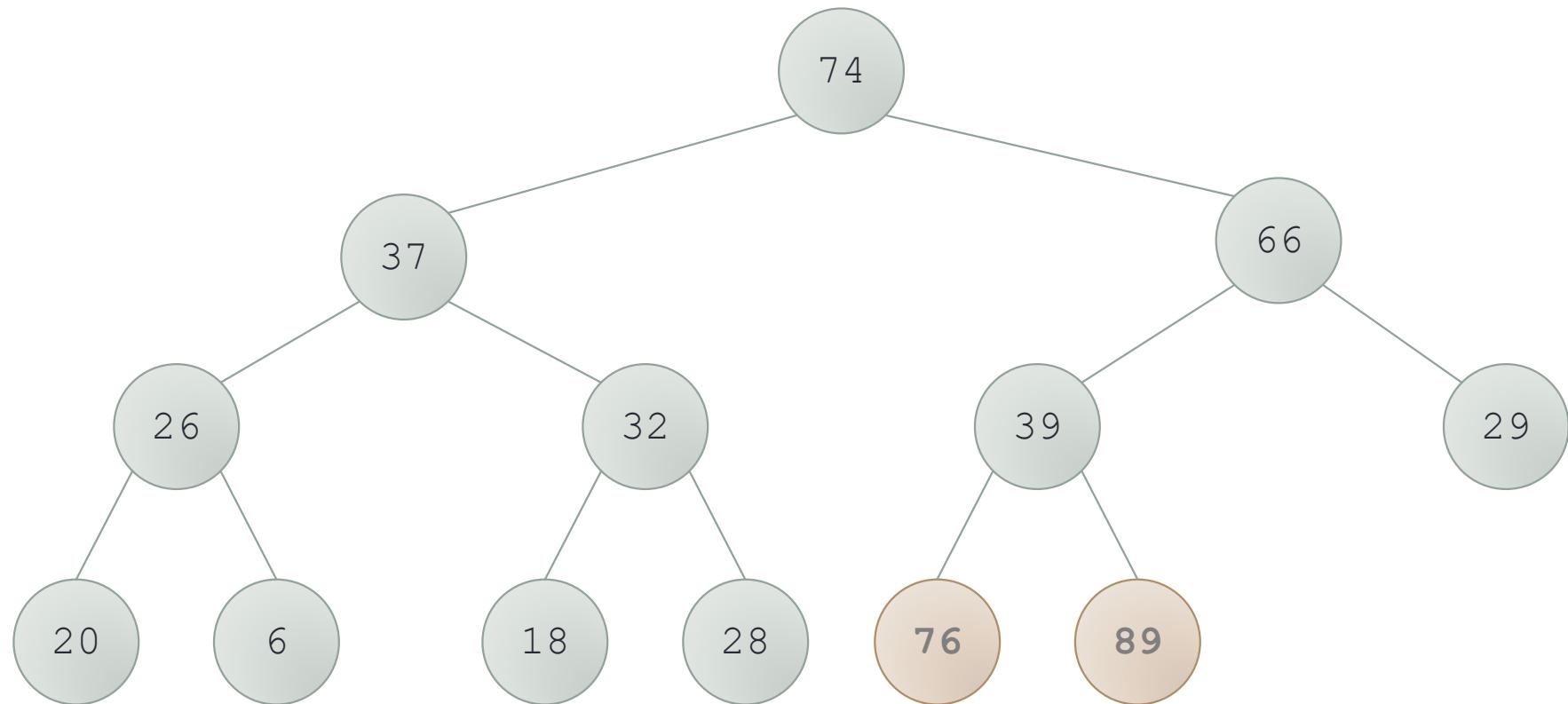
Trace of In-Place Heapsort (cont.)



Trace of In-Place Heapsort (cont.)

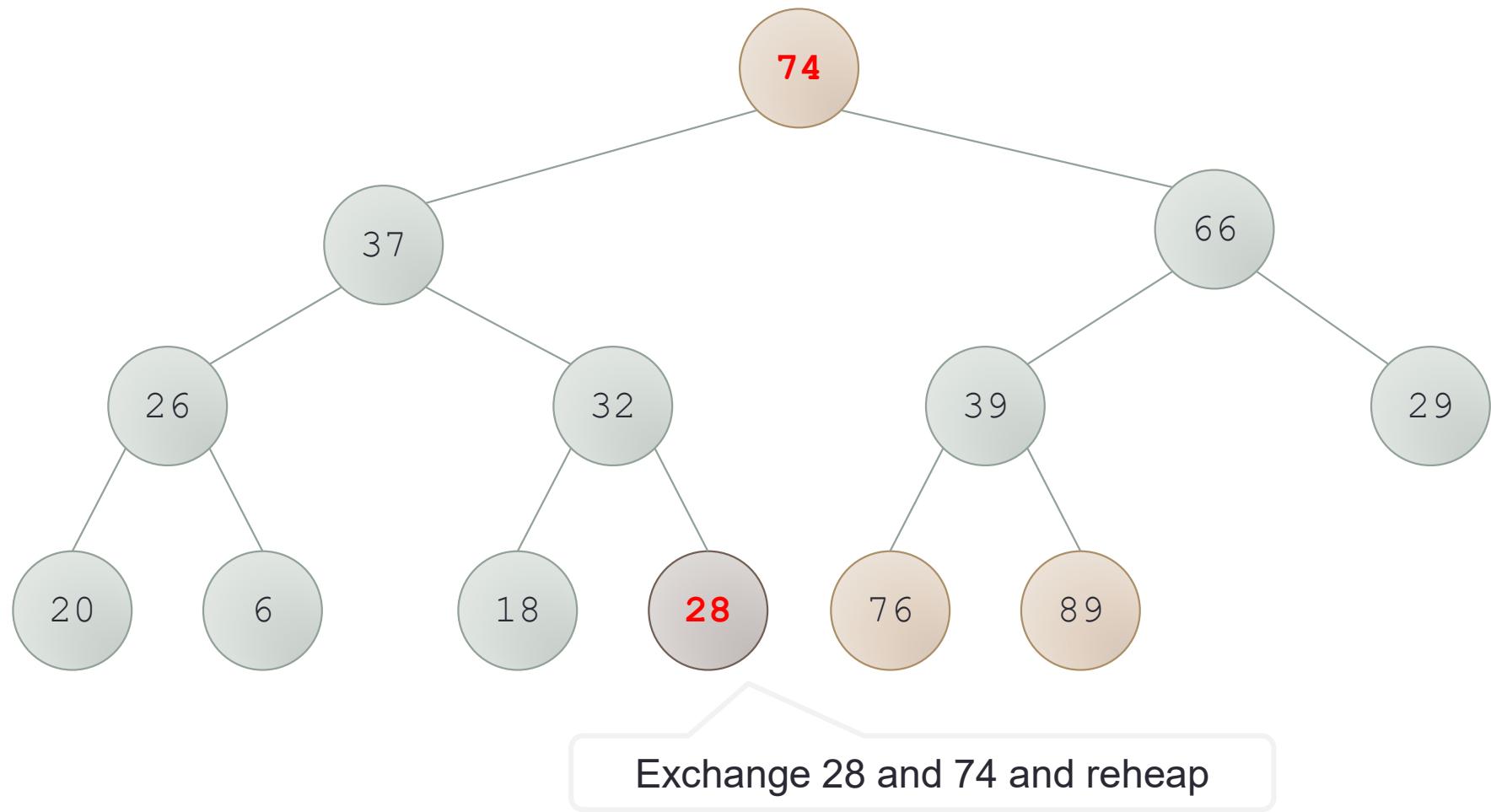


Trace of In-Place Heapsort (cont.)

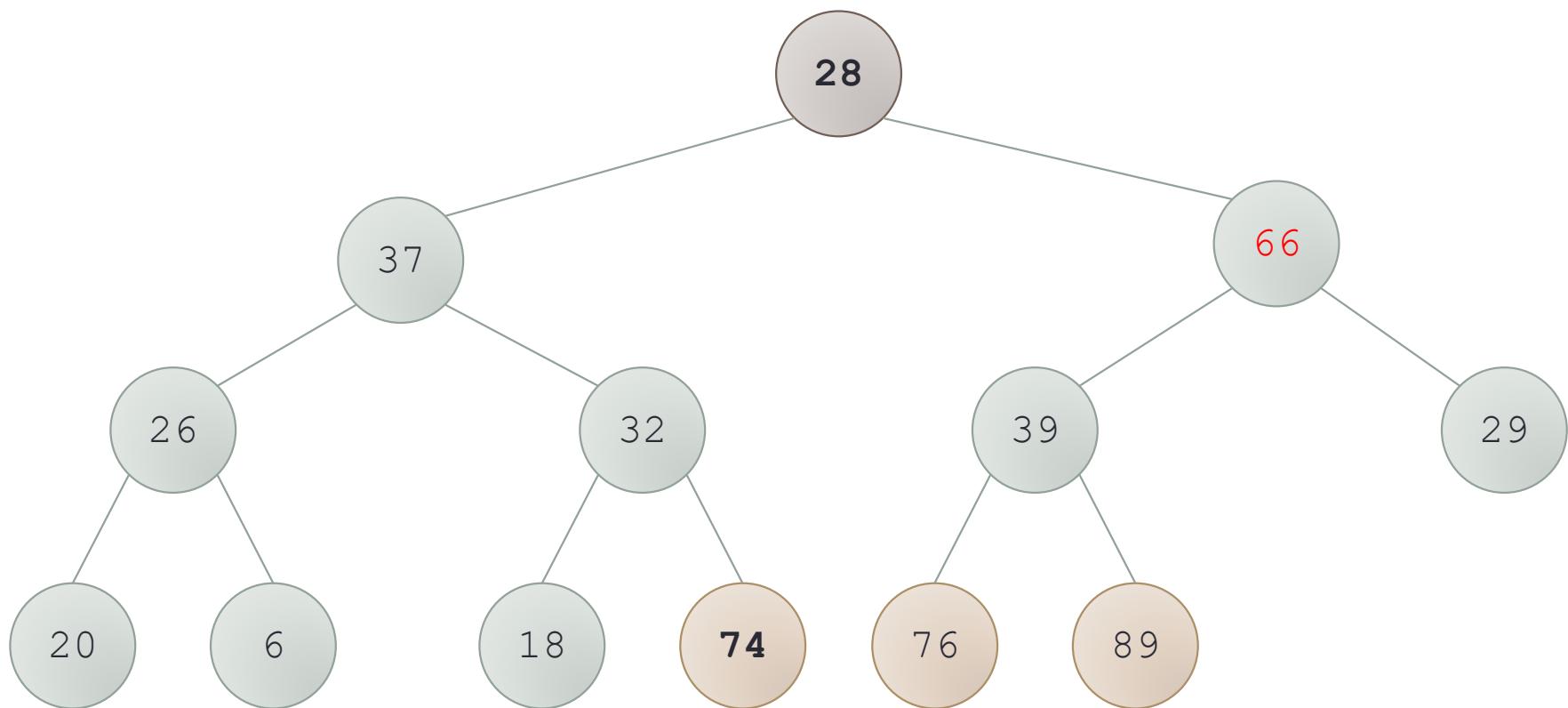


Array representation of unsorted array after two largest elements are removed:
{74, 37, 66, 26, 32, 39, 29, 20, 6, 18, 28, 76, 89}

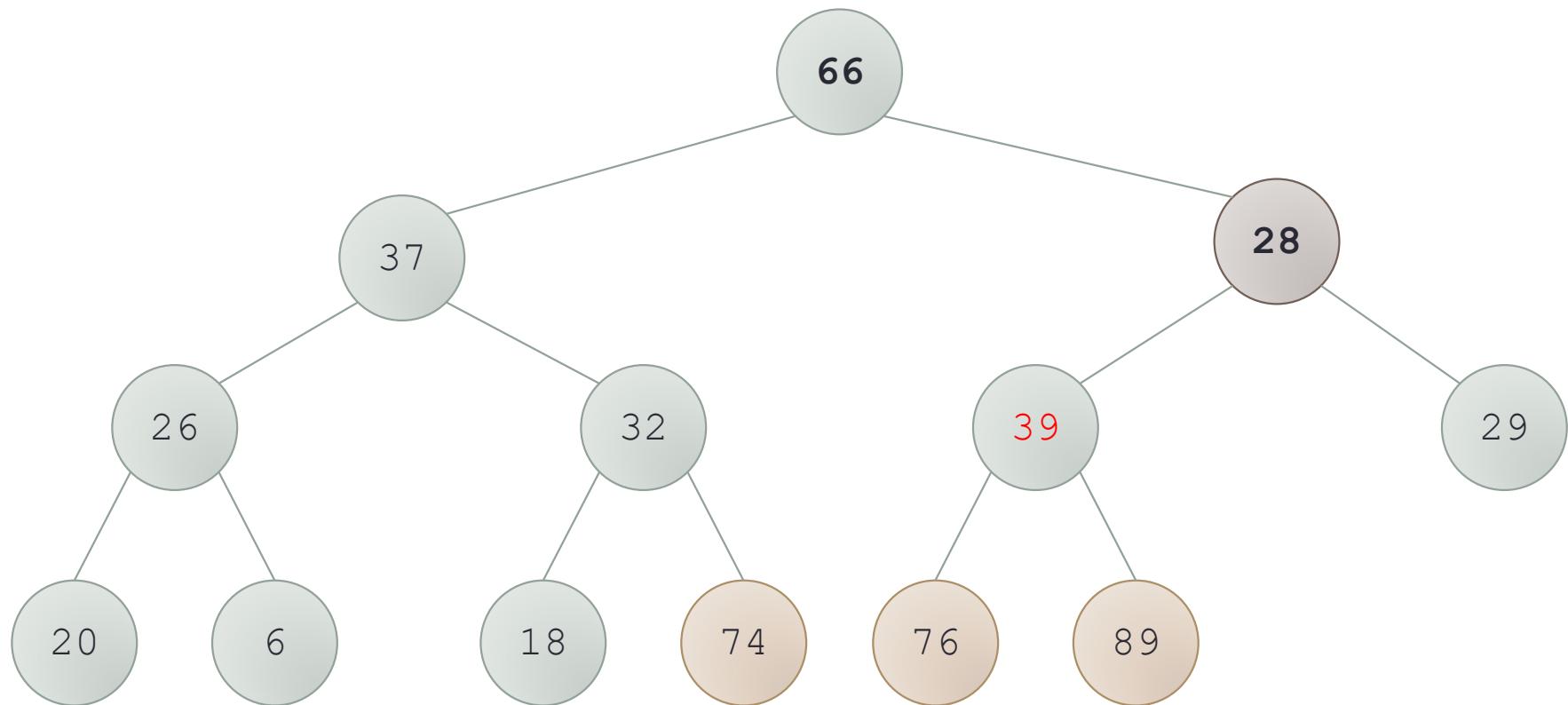
Trace of In-Place Heapsort (cont.)



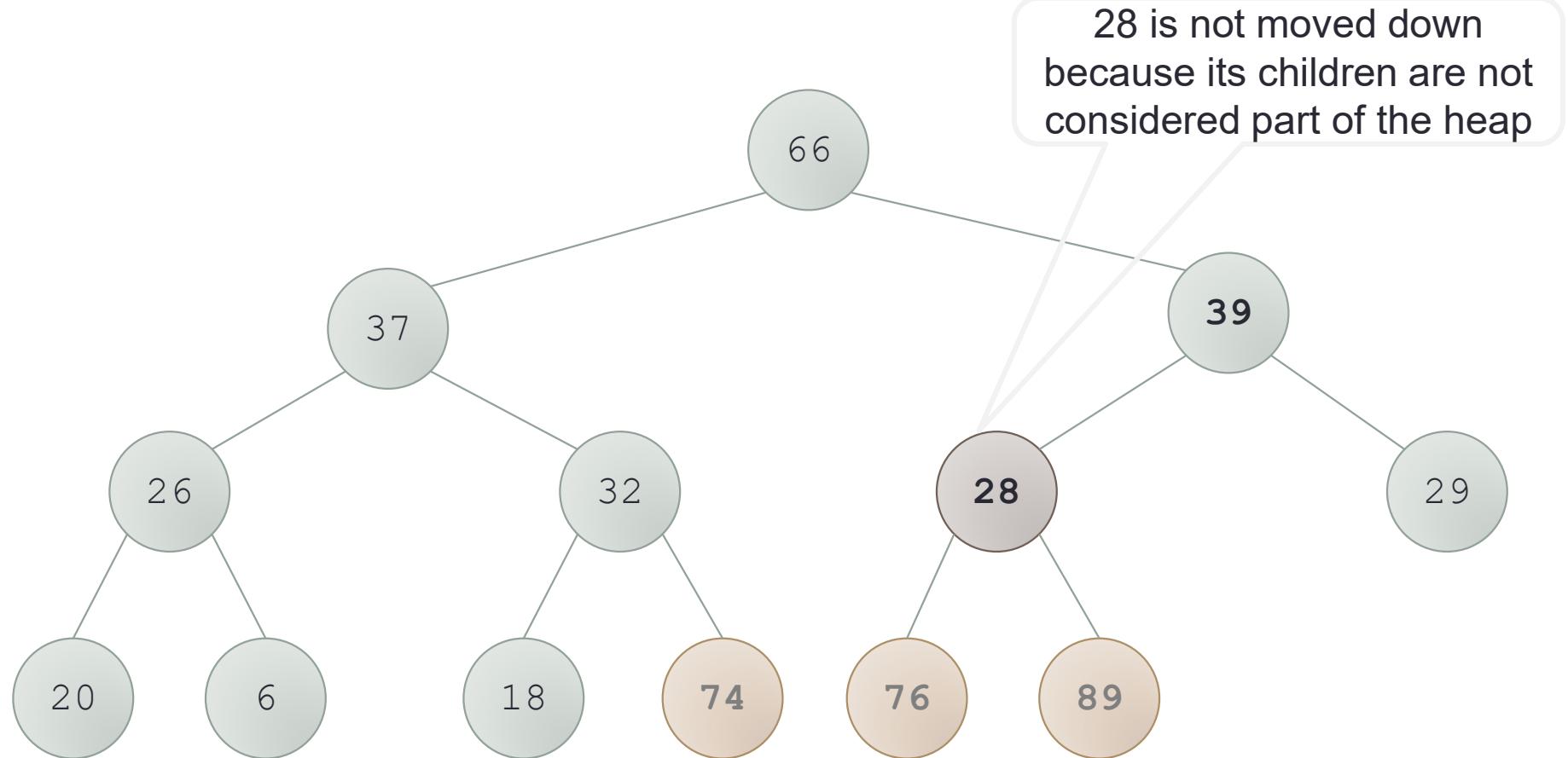
Trace of In-Place Heapsort (cont.)



Trace of In-Place Heapsort (cont.)

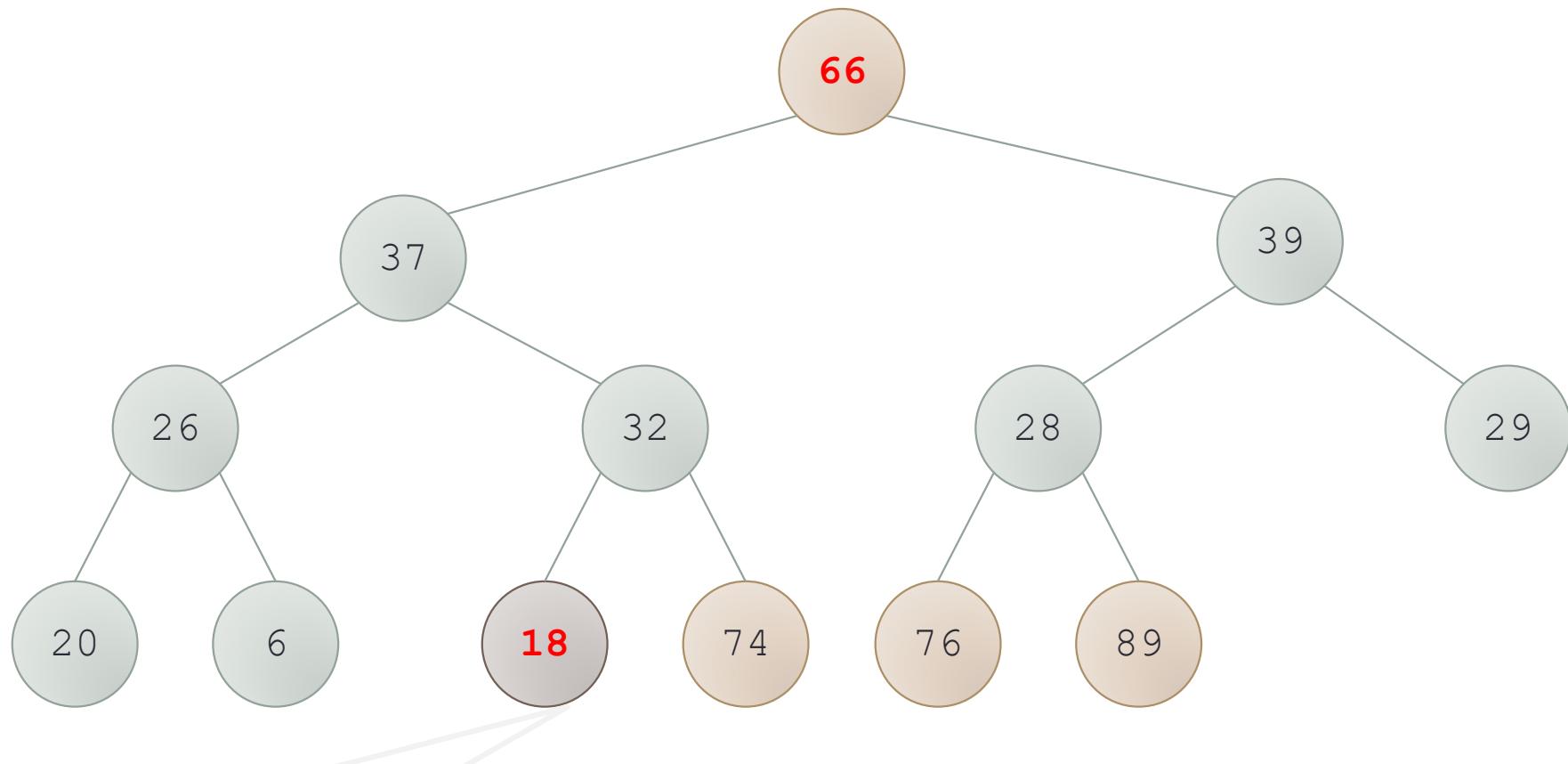


Trace of In-Place Heapsort (cont.)



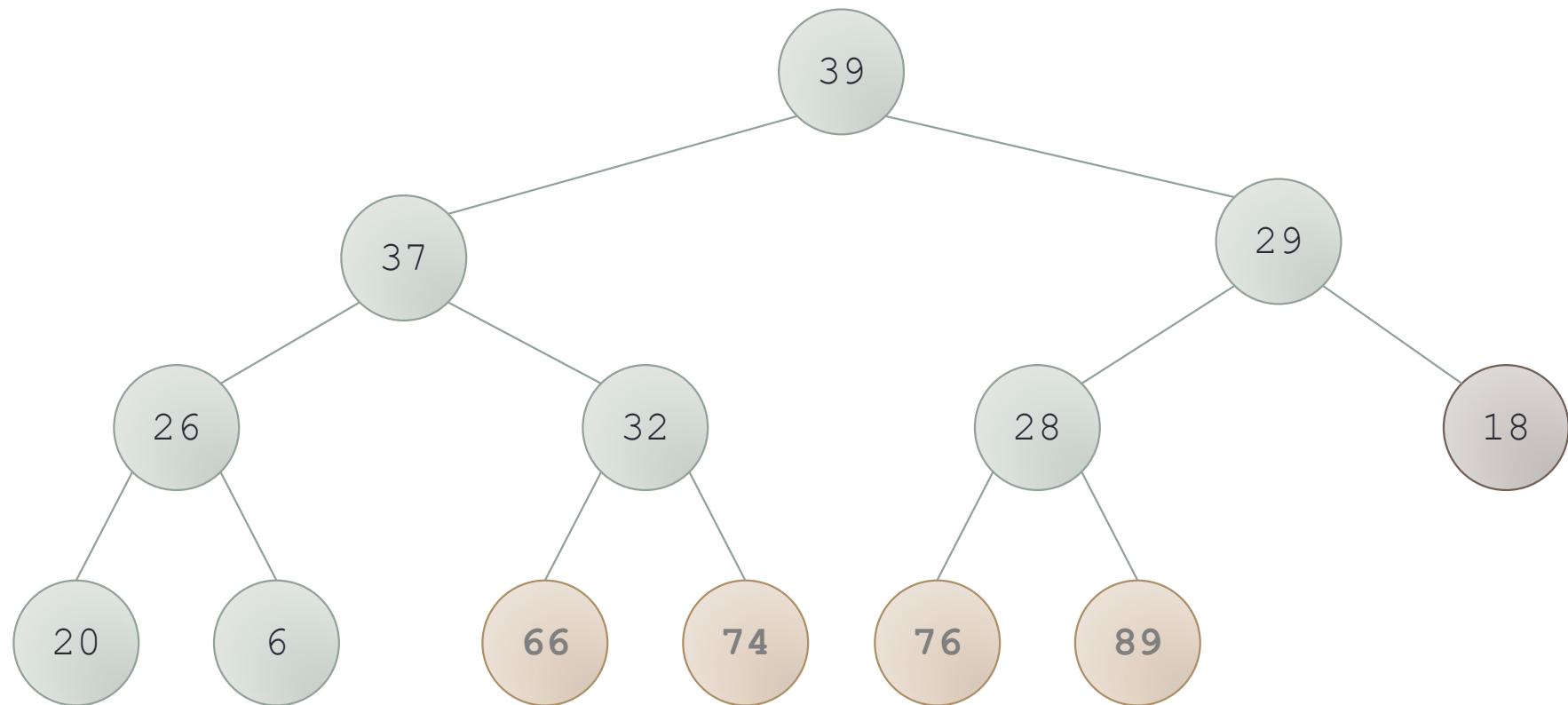
Array representation of unsorted array **after three largest elements are removed**:
{66, 37, 39, 26, 32, 28, 29, 20, 6, 18, 74, 76, 89}

Trace of In-Place Heapsort (cont.)



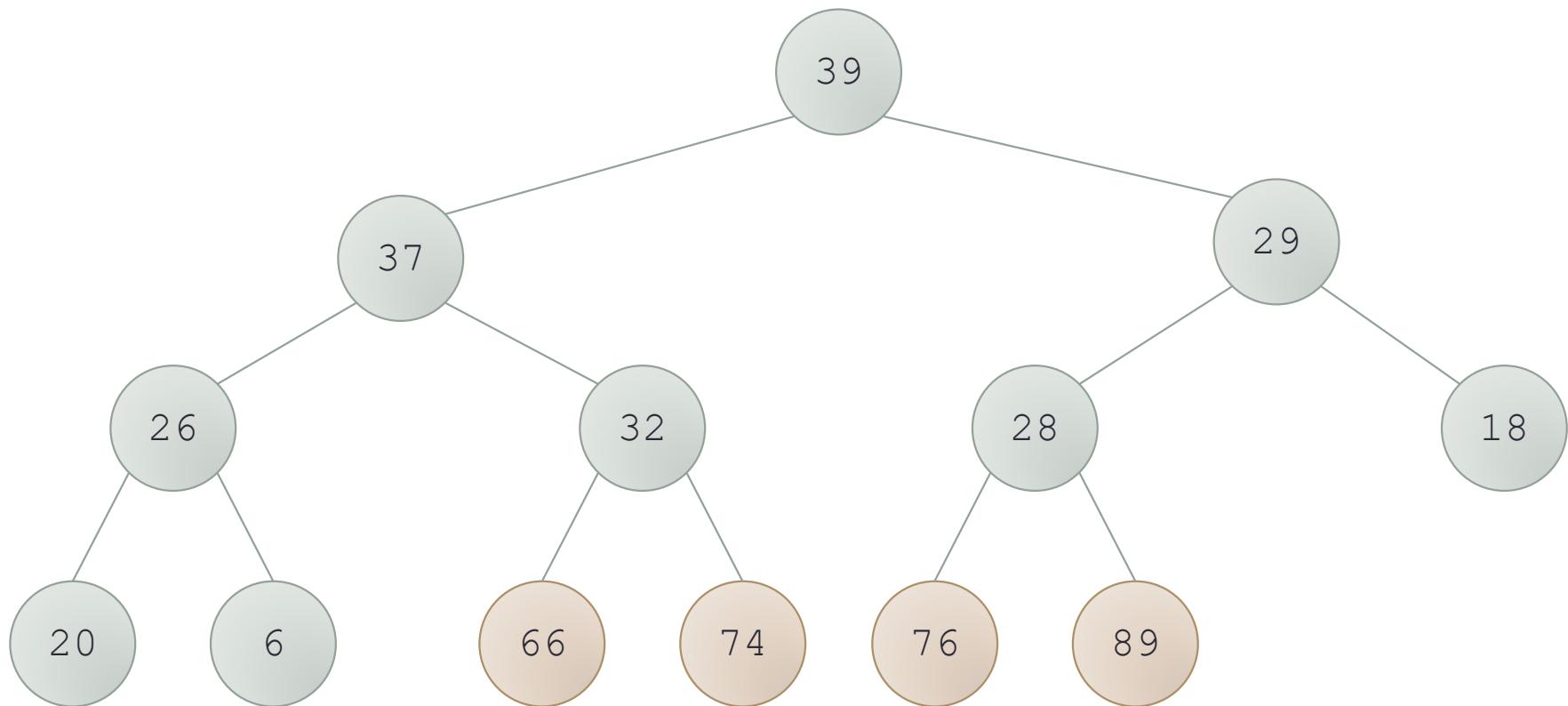
Process continues – exchange
18 and 66, and so on

Trace of In-Place Heapsort (cont.)

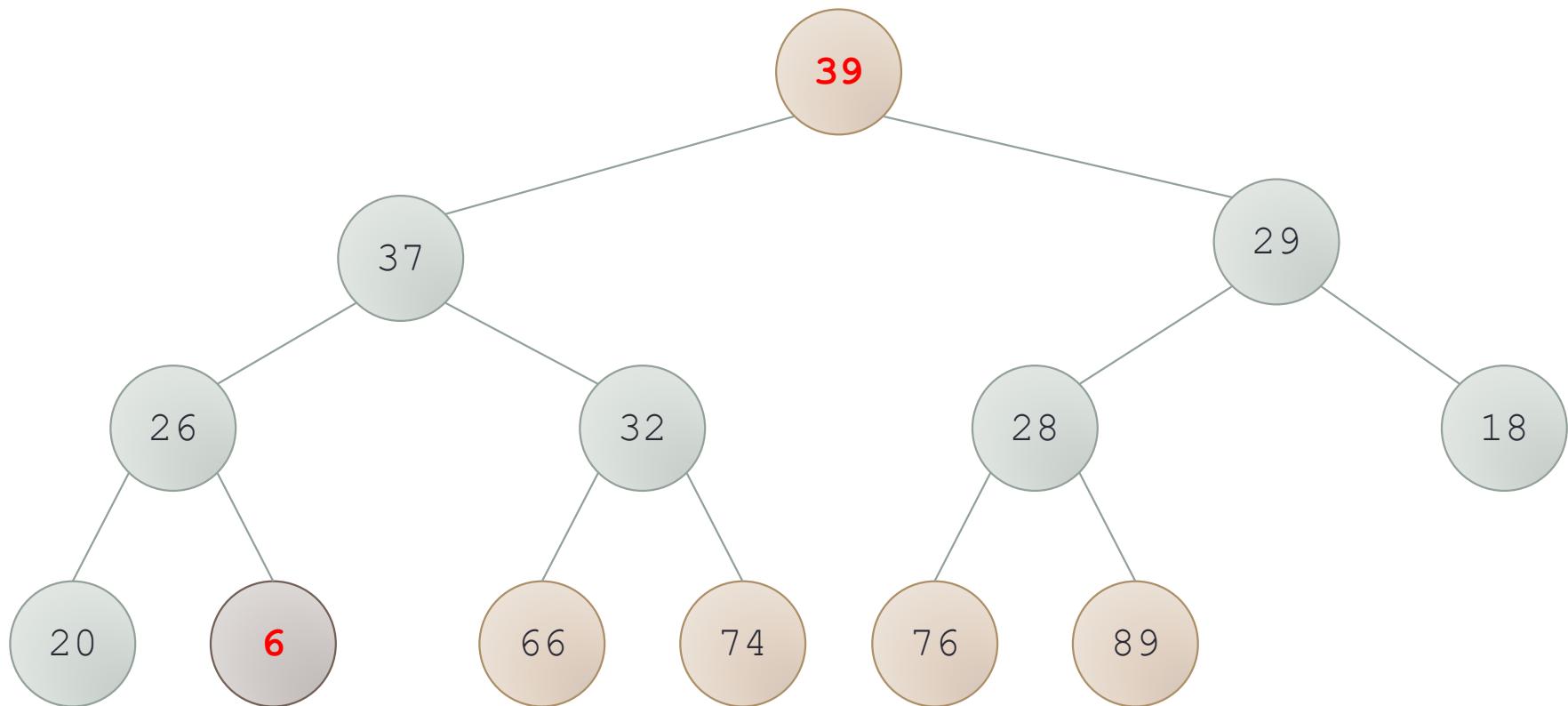


Array representation of unsorted array after four largest elements are removed:
{39, 37, 39, 26, 32, 28, 18, 20, 6, 66, 74, 76, 89}

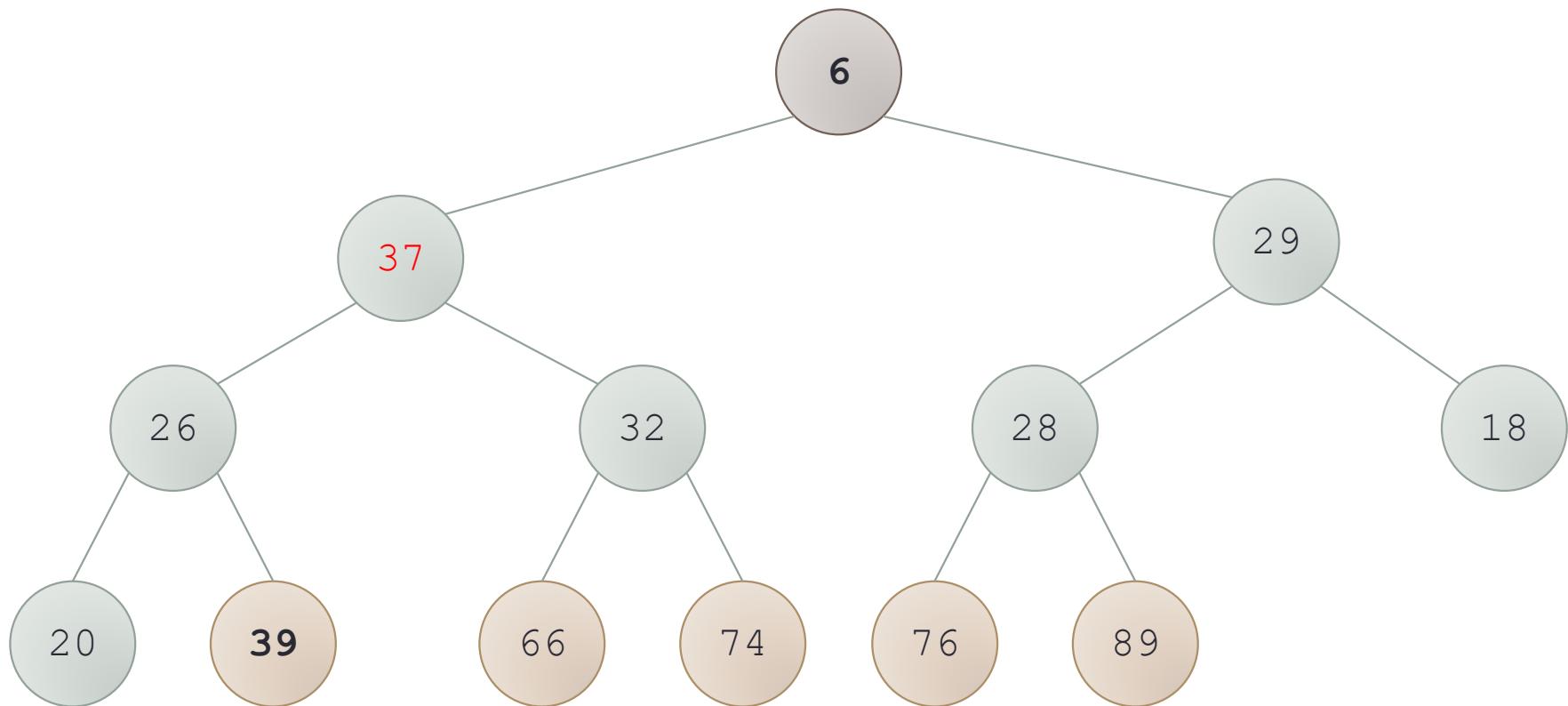
Trace of In-Place Heapsort (cont.)



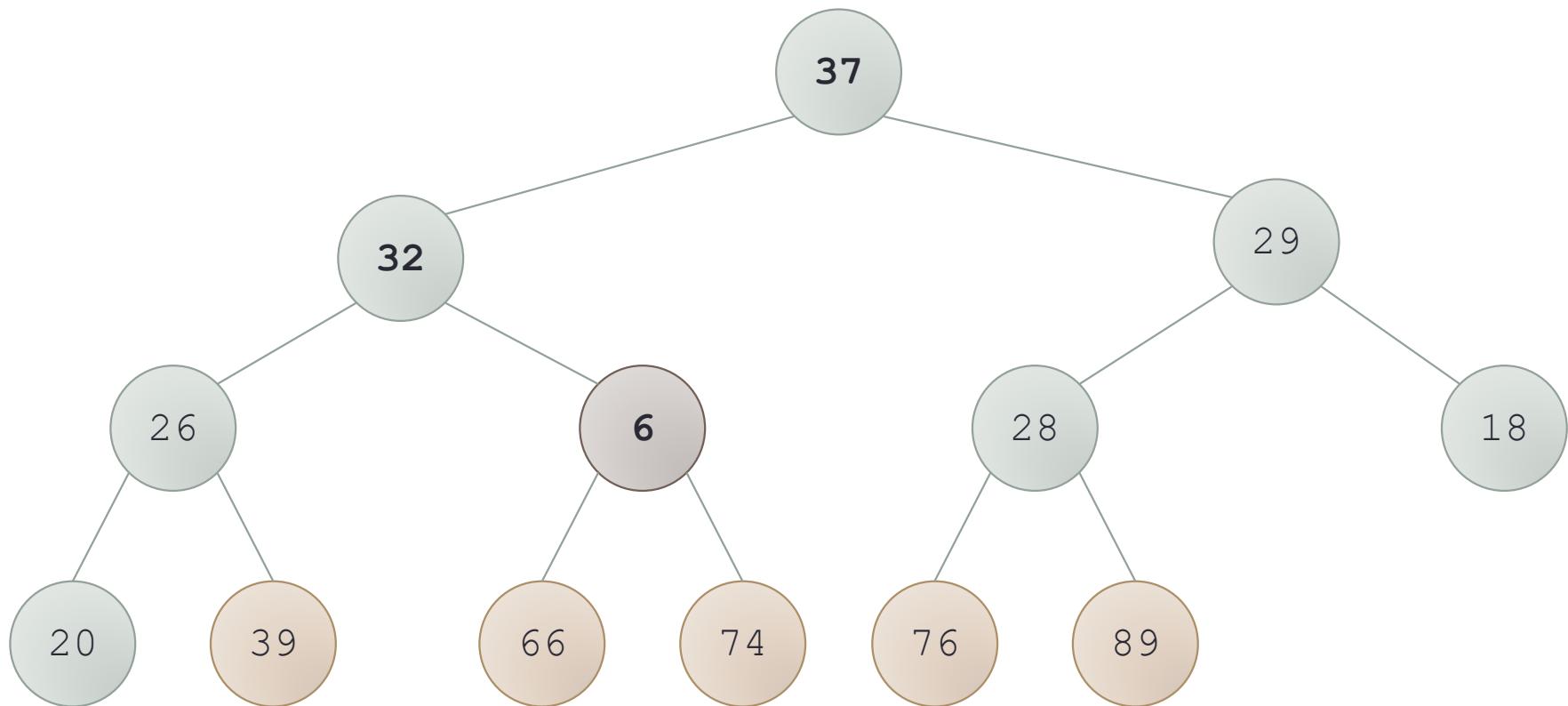
Trace of In-Place Heapsort (cont.)



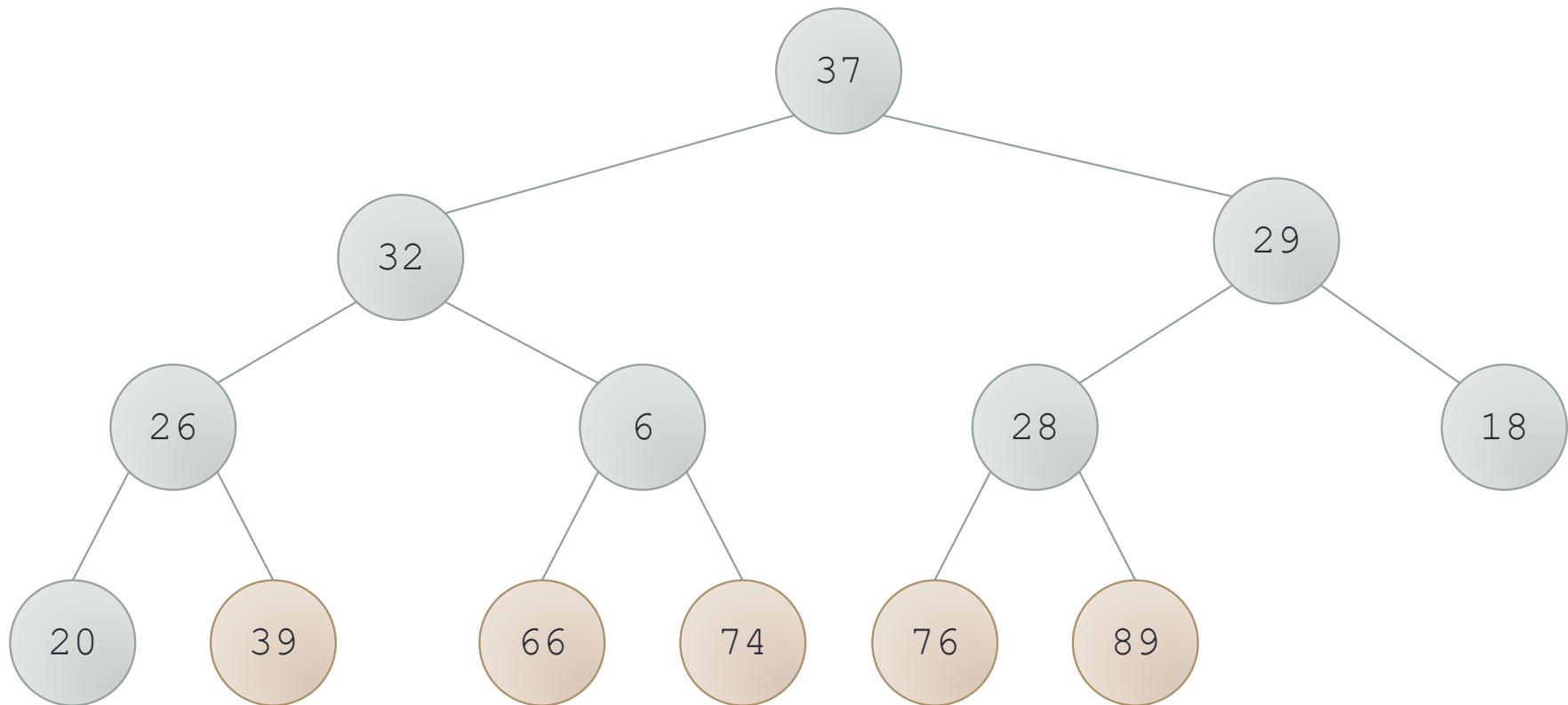
Trace of In-Place Heapsort (cont.)



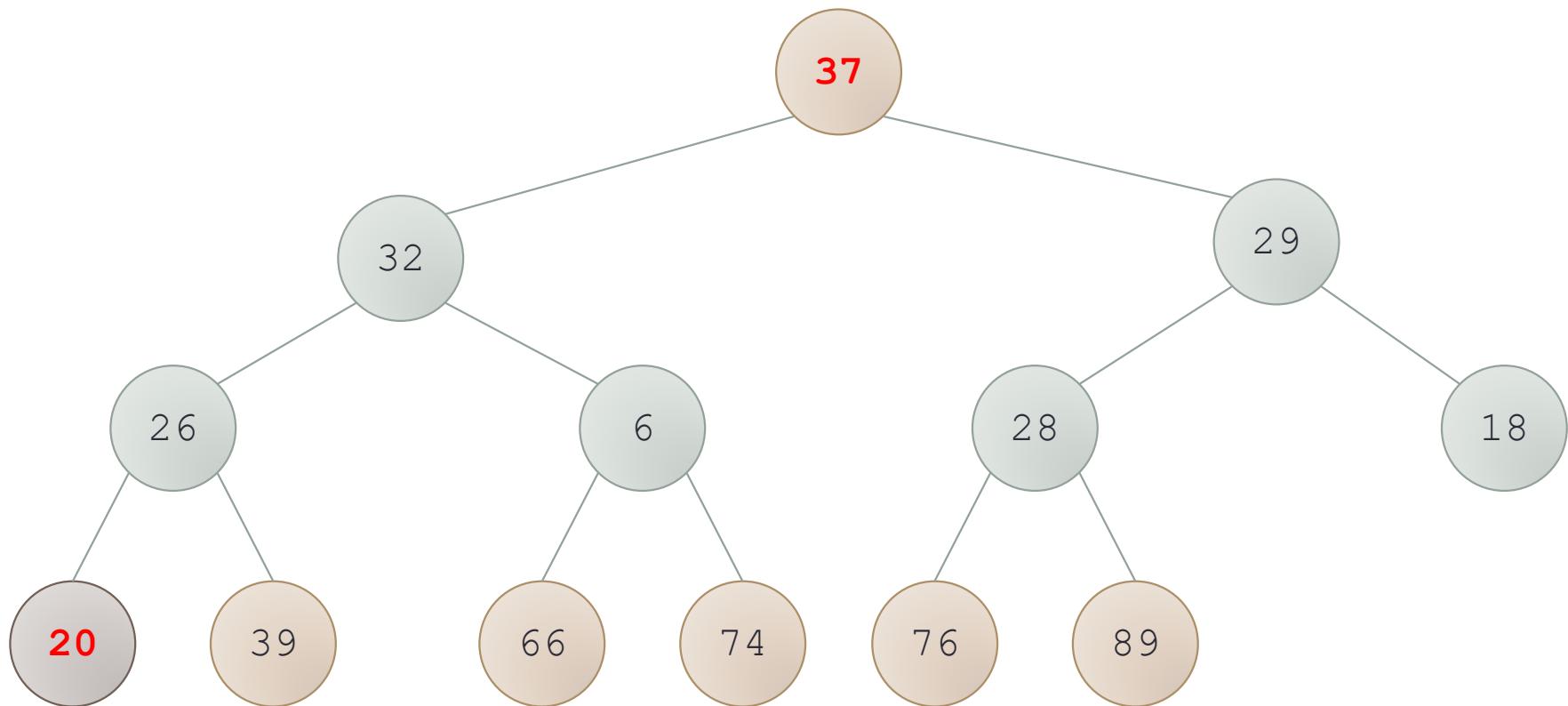
Trace of In-Place Heapsort (cont.)



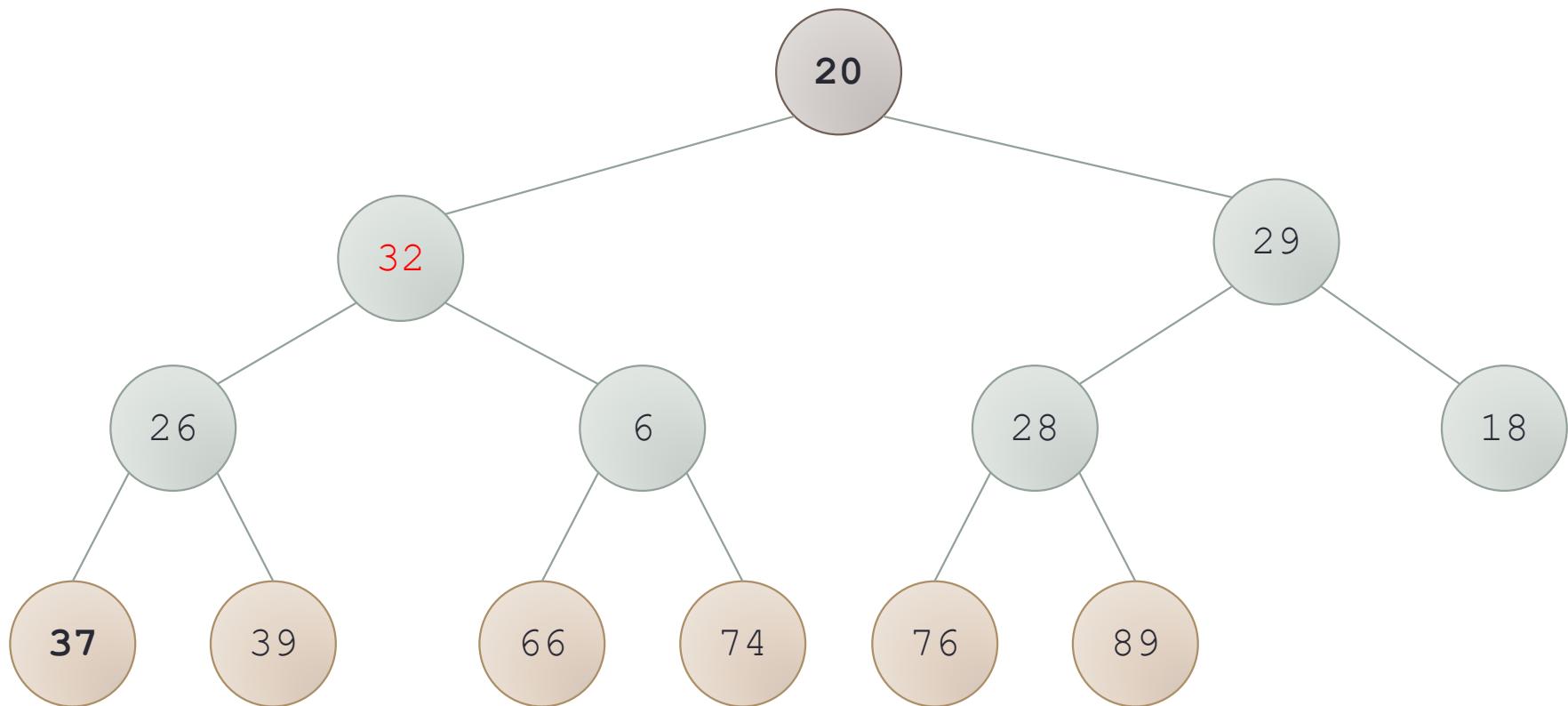
Trace of In-Place Heapsort (cont.)



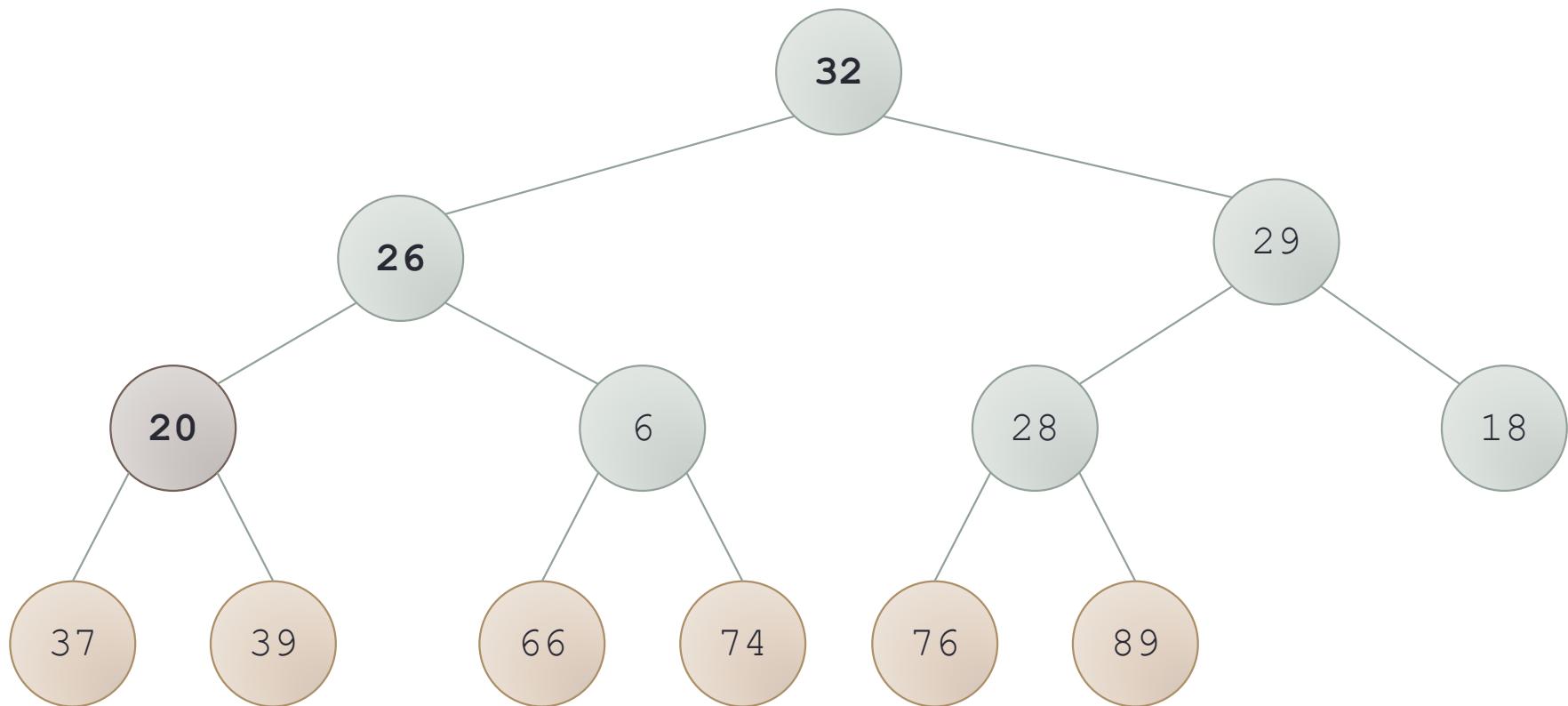
Trace of In-Place Heapsort (cont.)



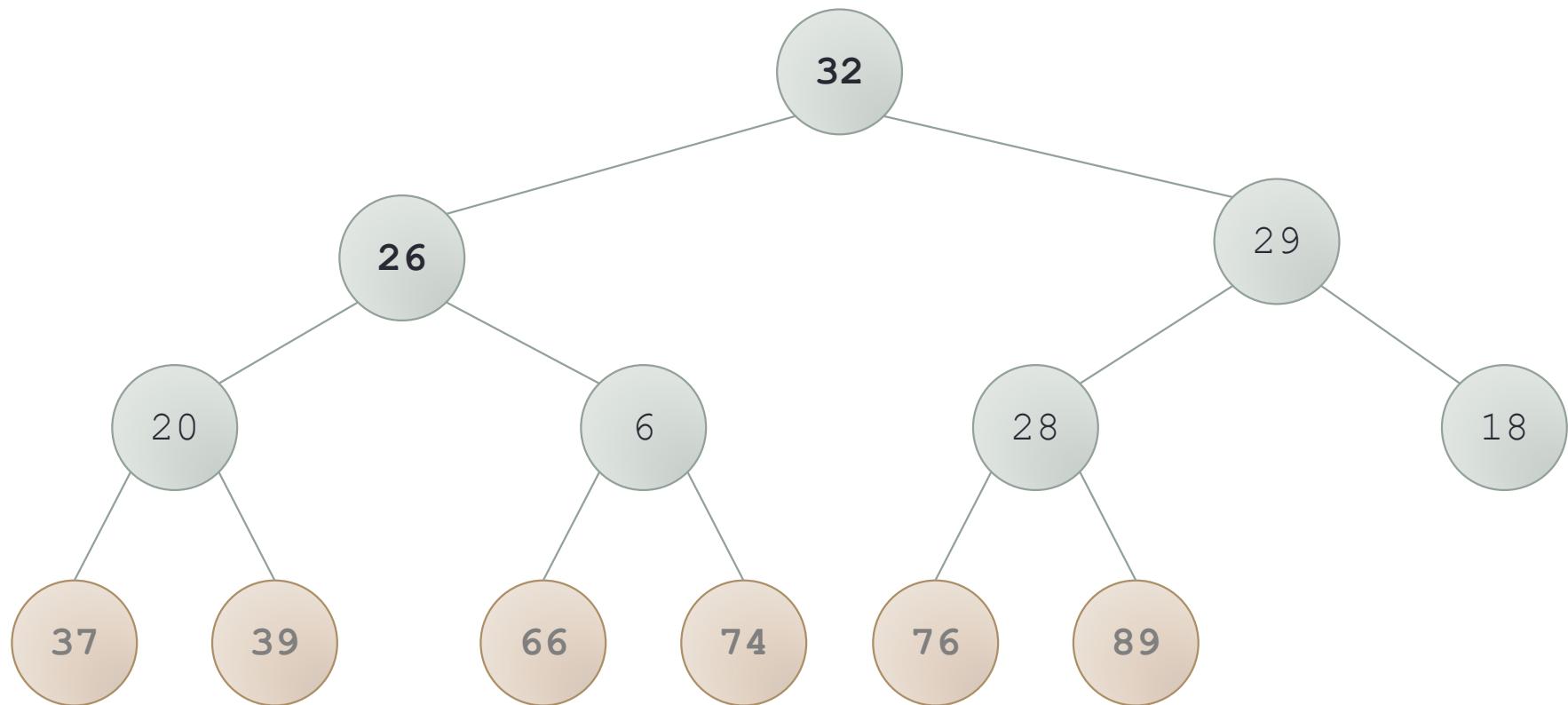
Trace of In-Place Heapsort (cont.)



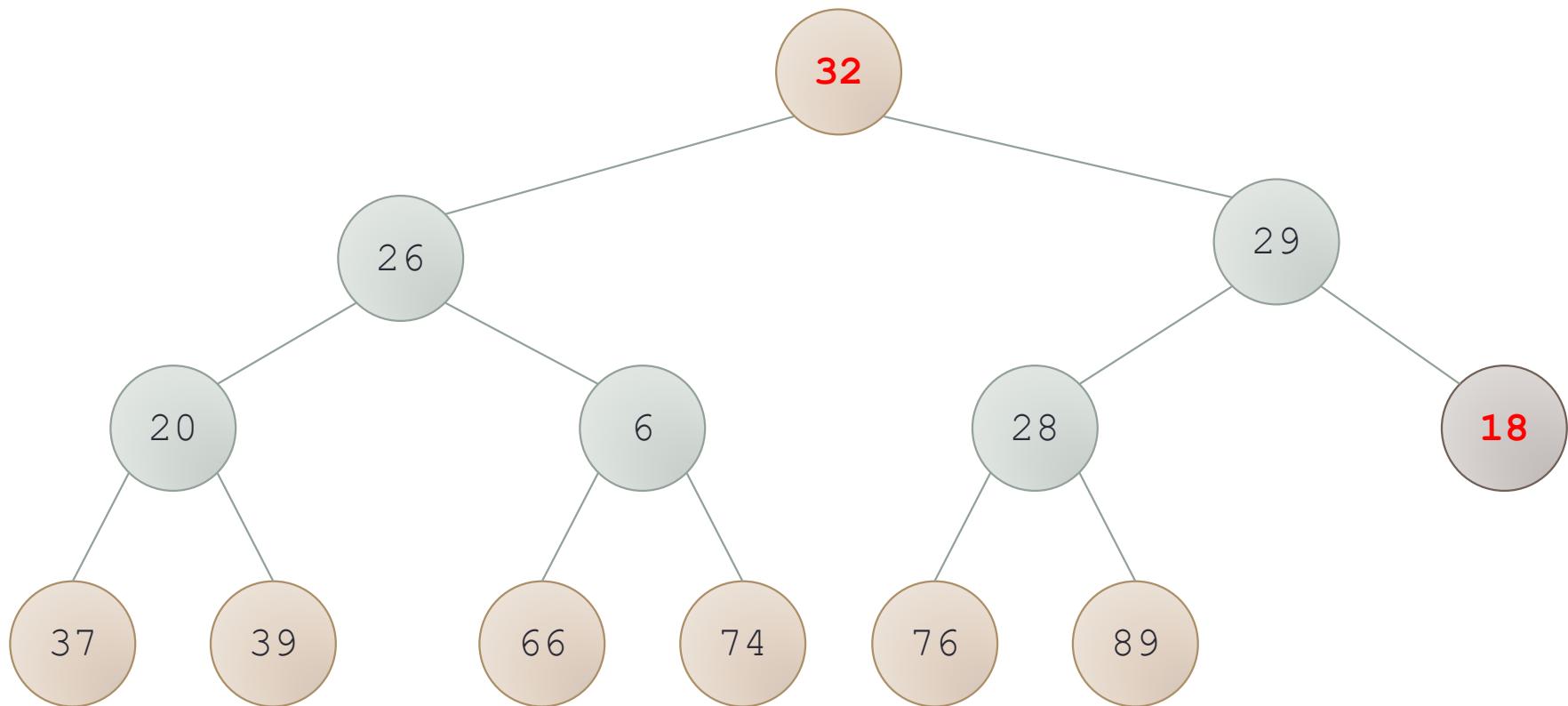
Trace of In-Place Heapsort (cont.)



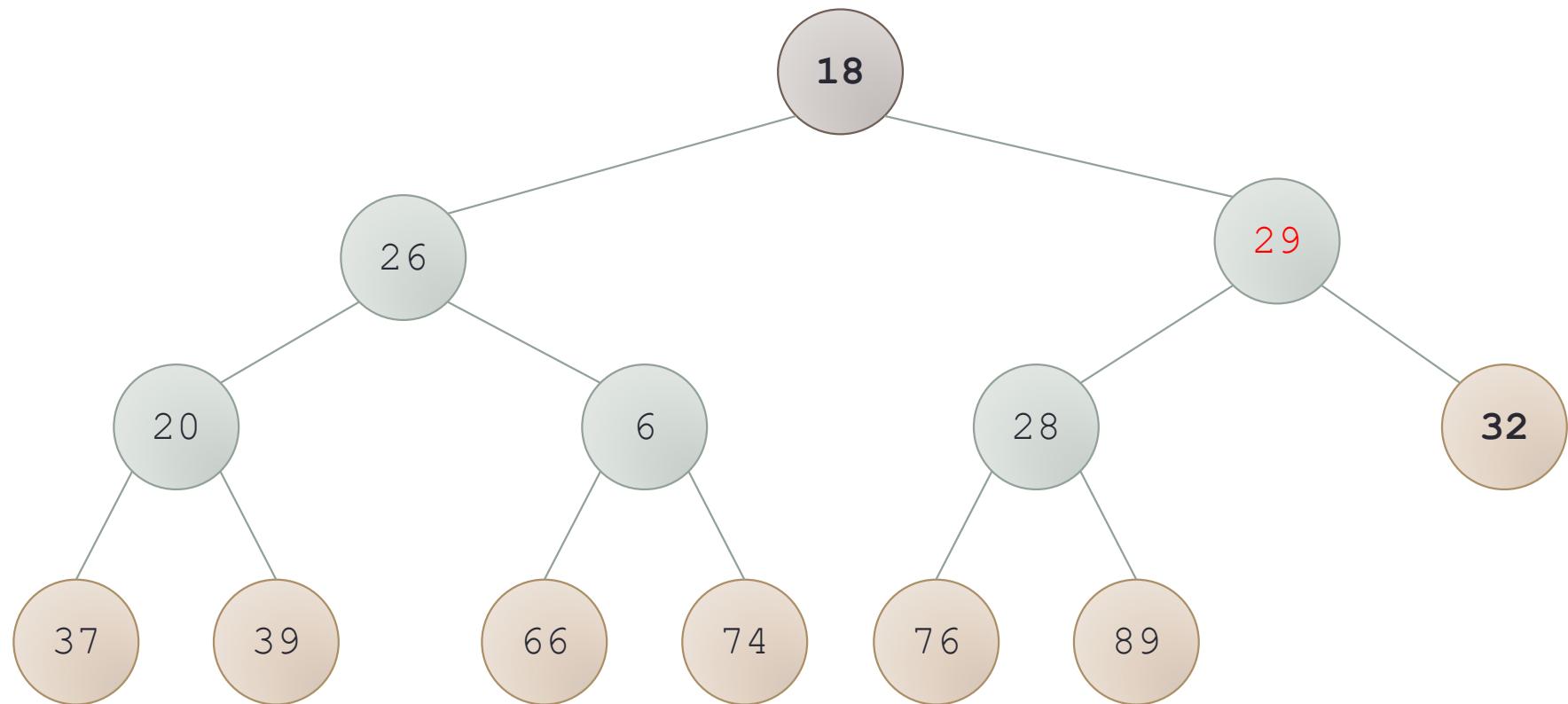
Trace of In-Place Heapsort (cont.)



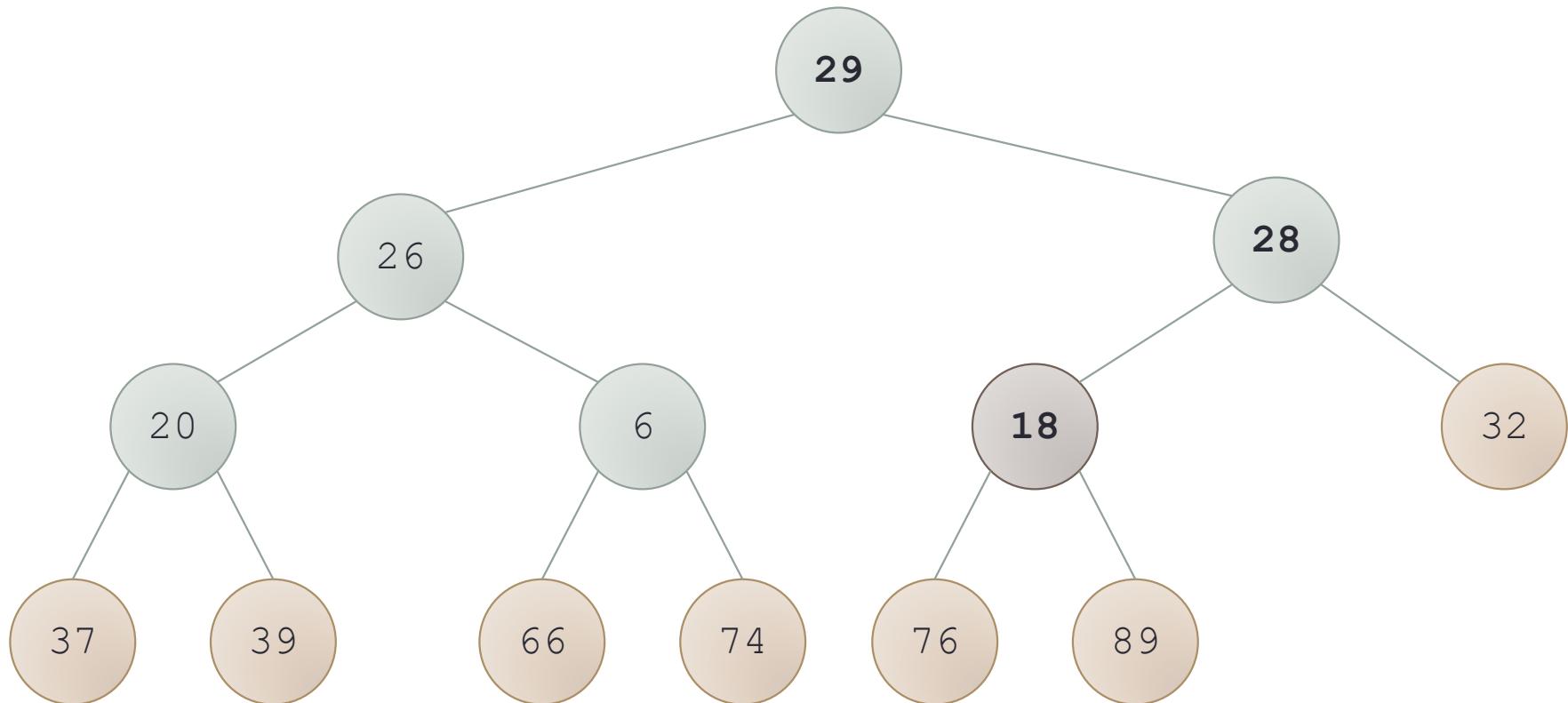
Trace of In-Place Heapsort (cont.)



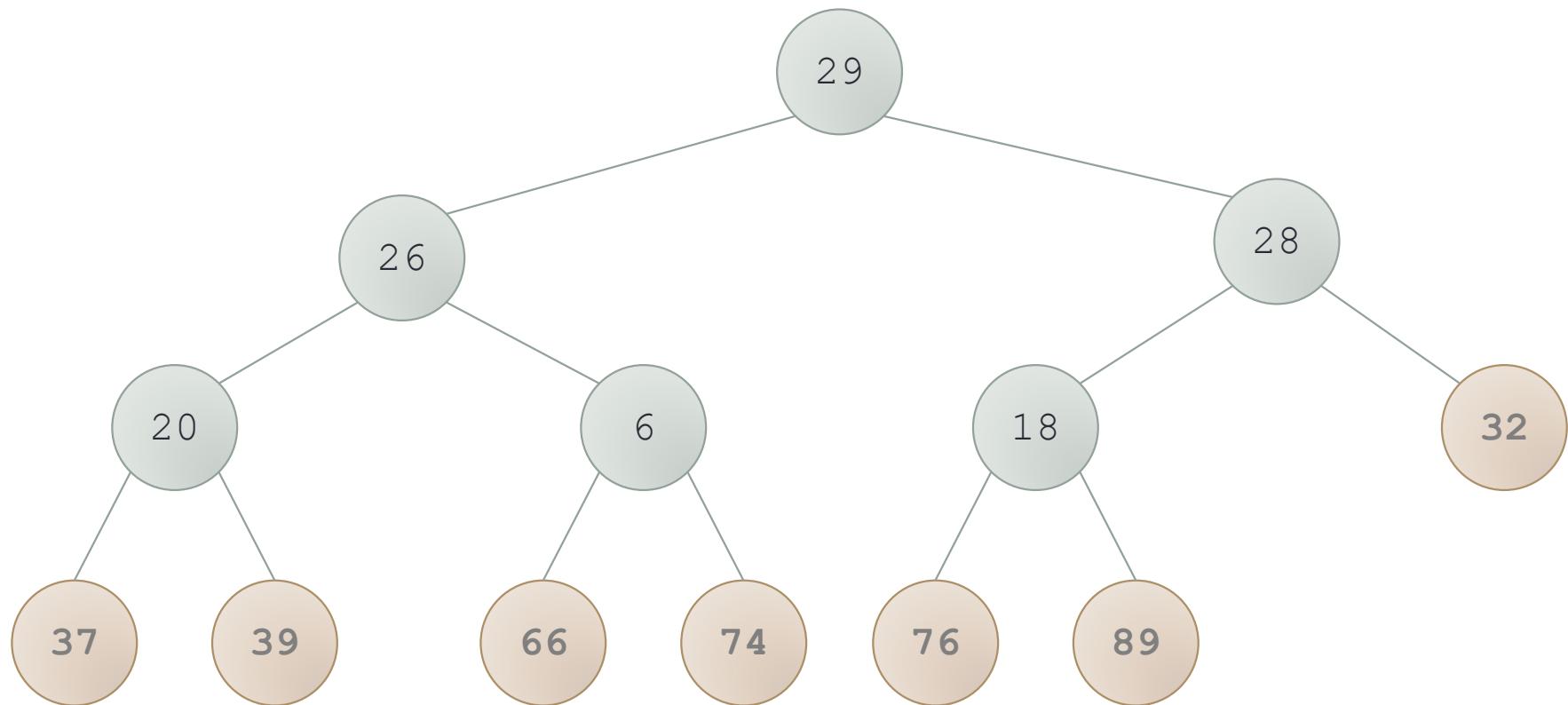
Trace of In-Place Heapsort (cont.)



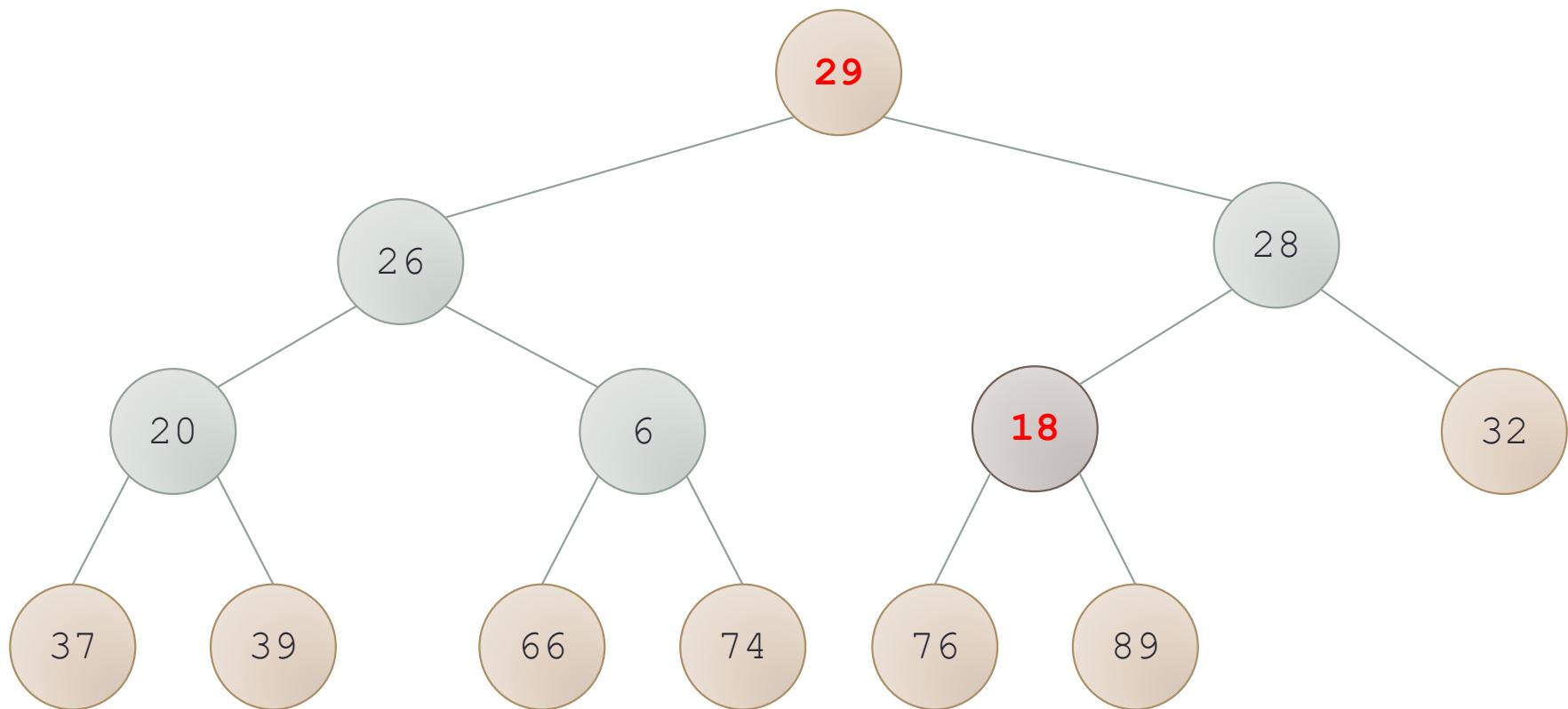
Trace of In-Place Heapsort (cont.)



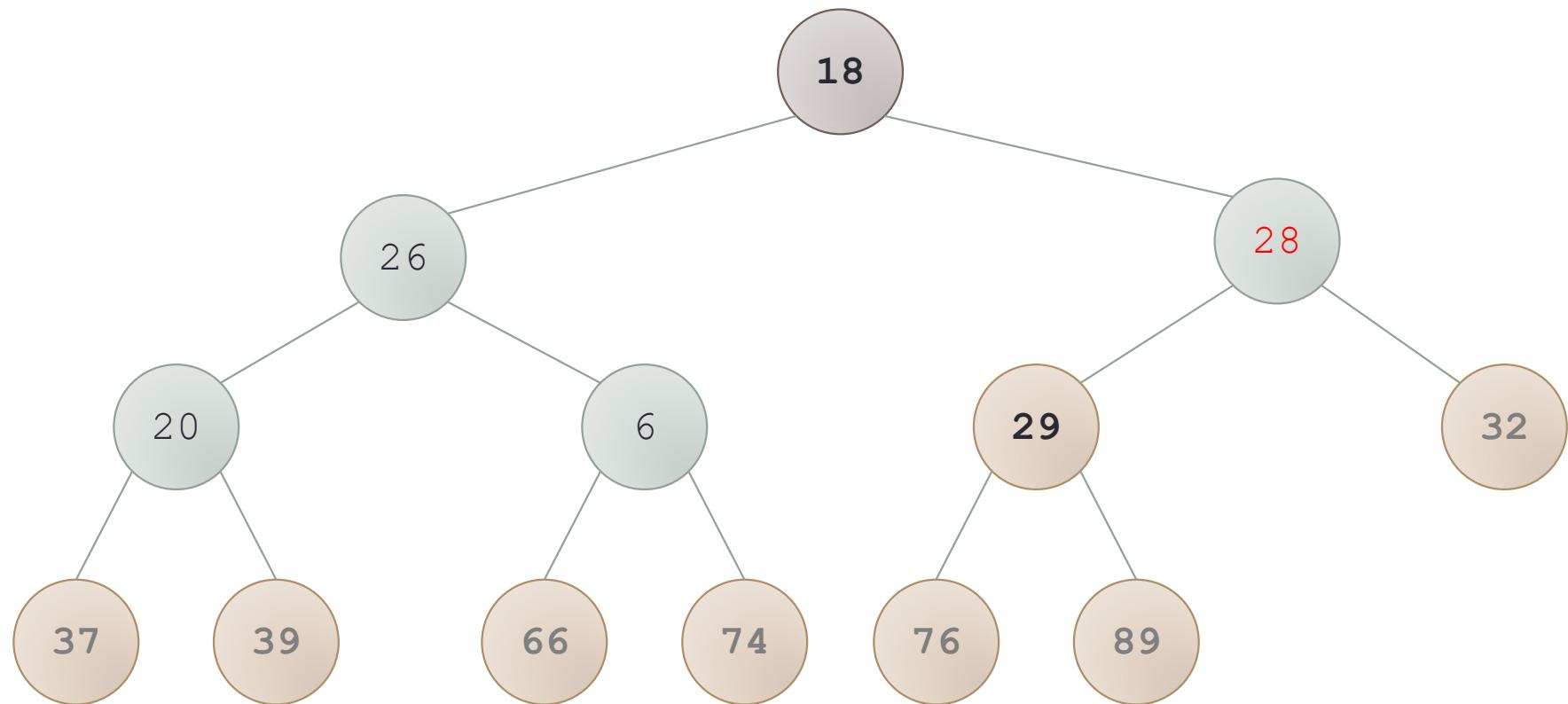
Trace of In-Place Heapsort (cont.)



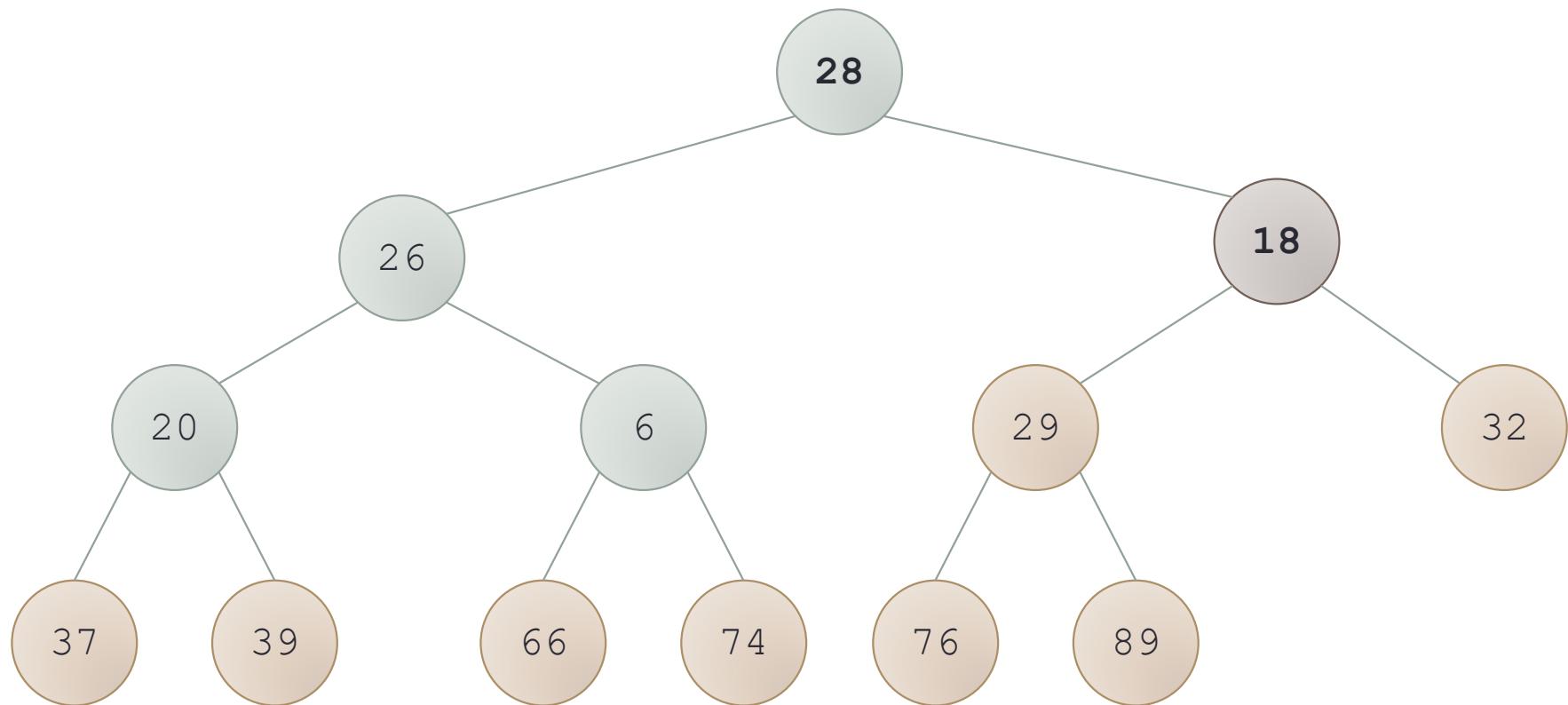
Trace of In-Place Heapsort (cont.)



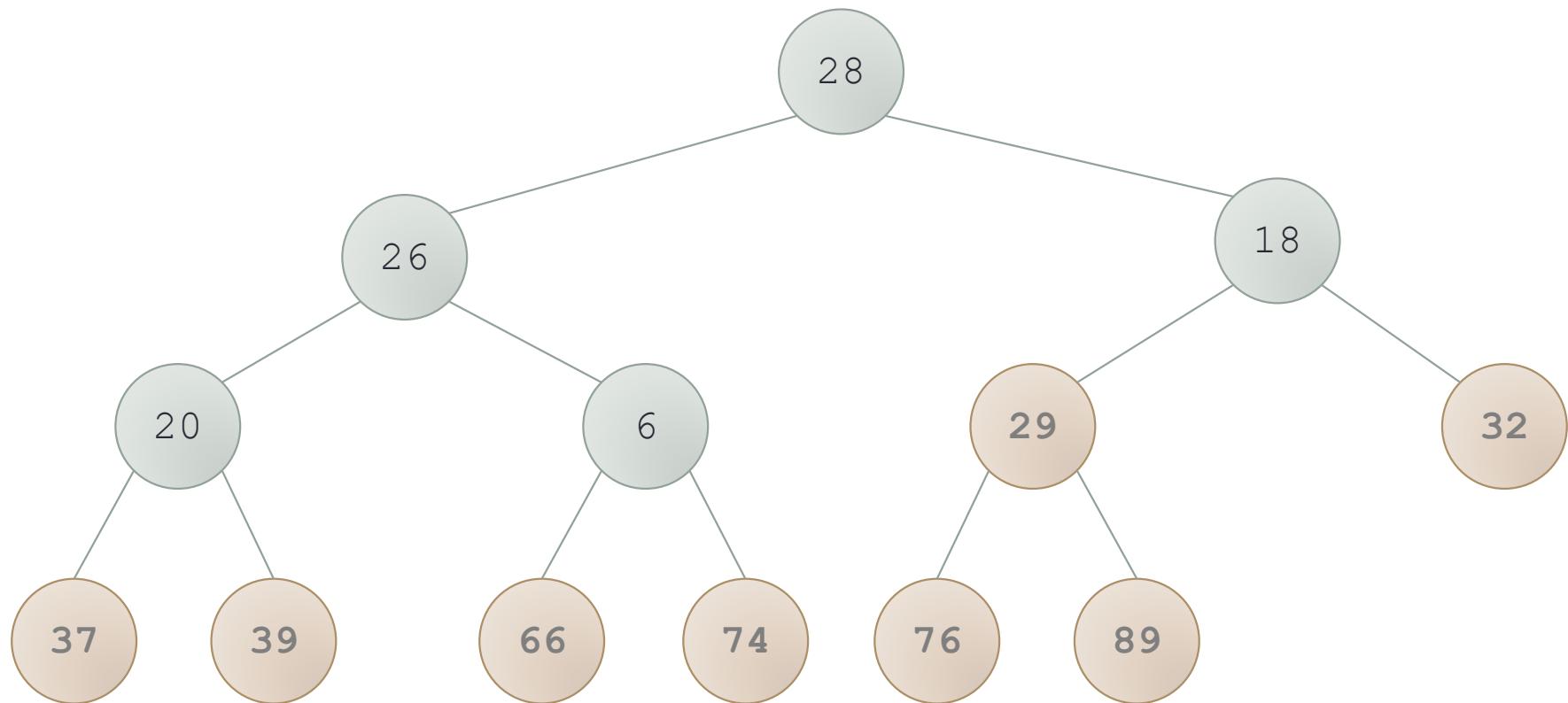
Trace of In-Place Heapsort (cont.)



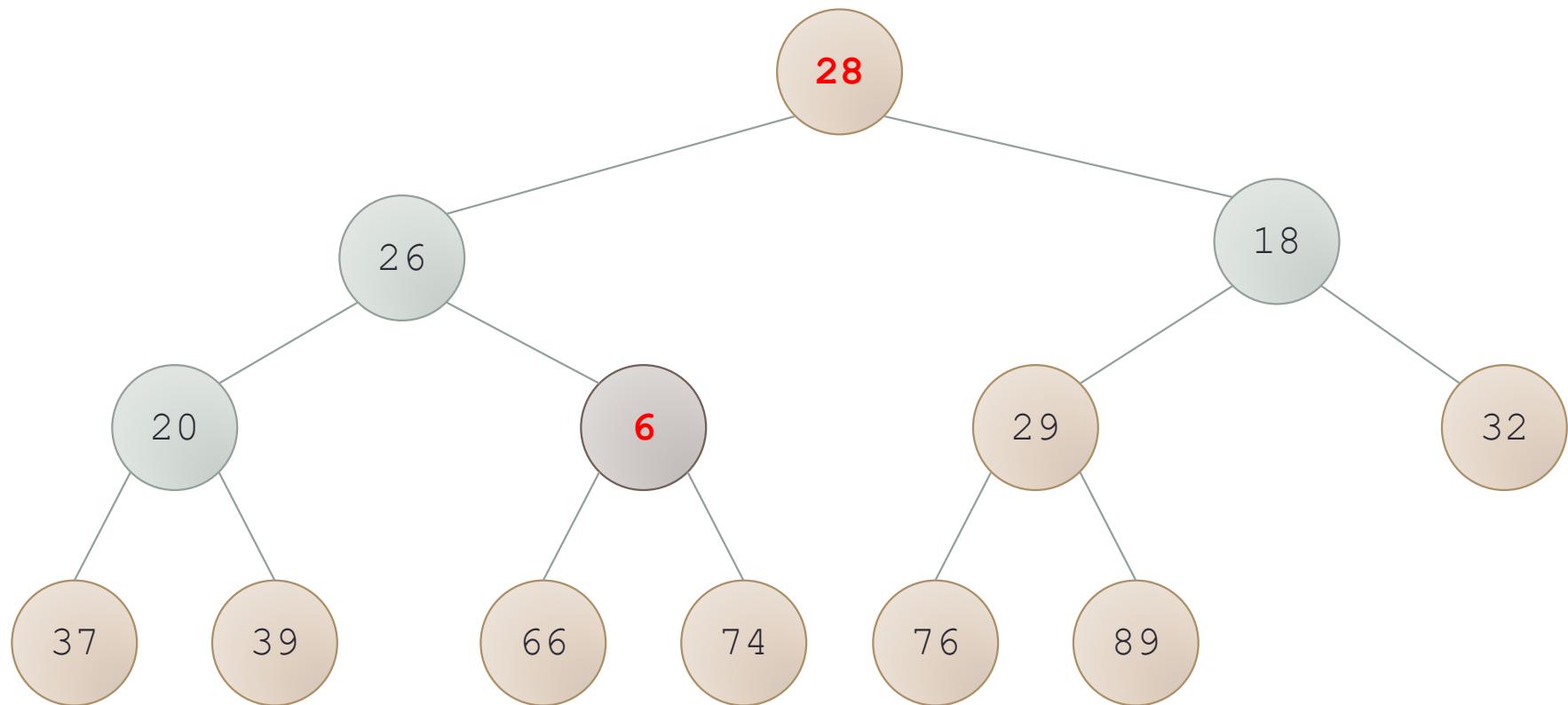
Trace of In-Place Heapsort (cont.)



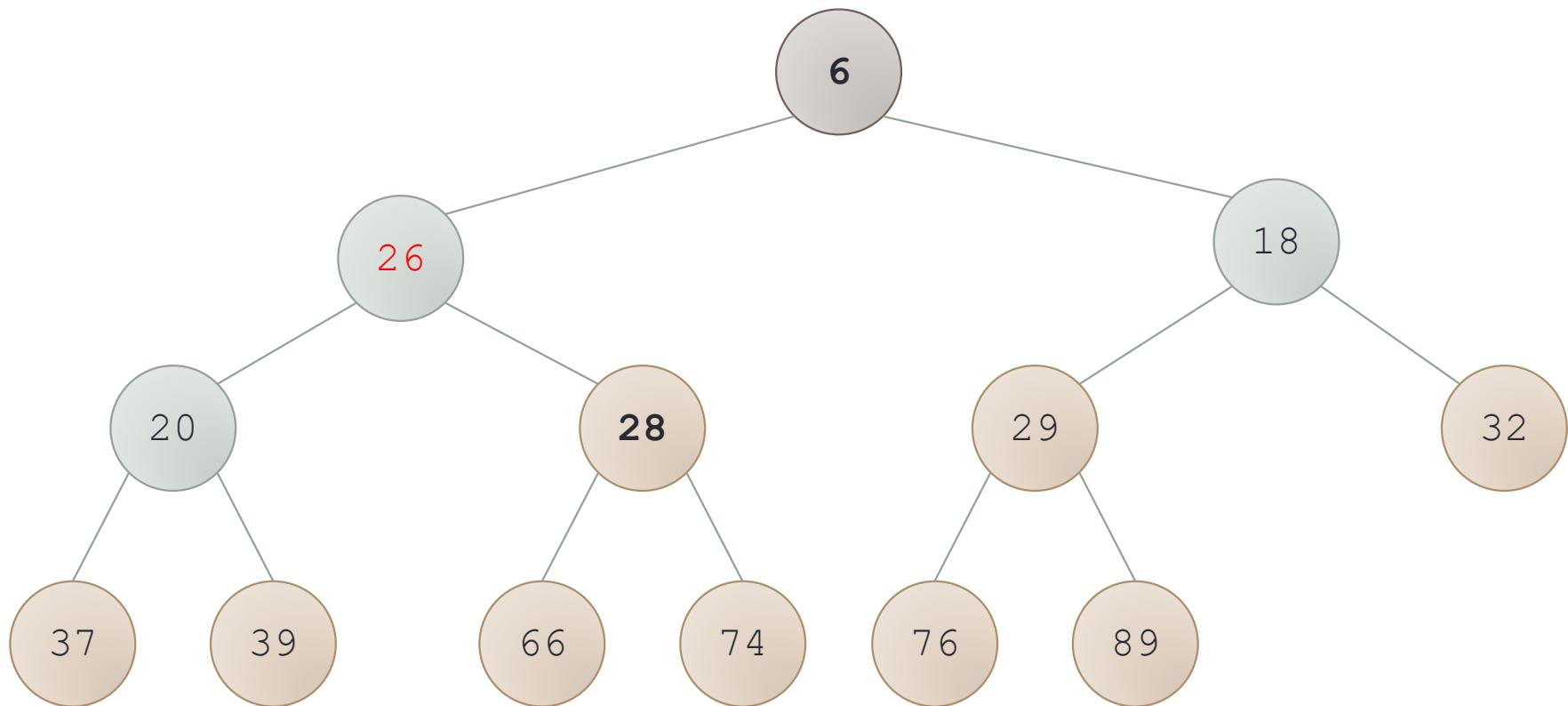
Trace of In-Place Heapsort (cont.)



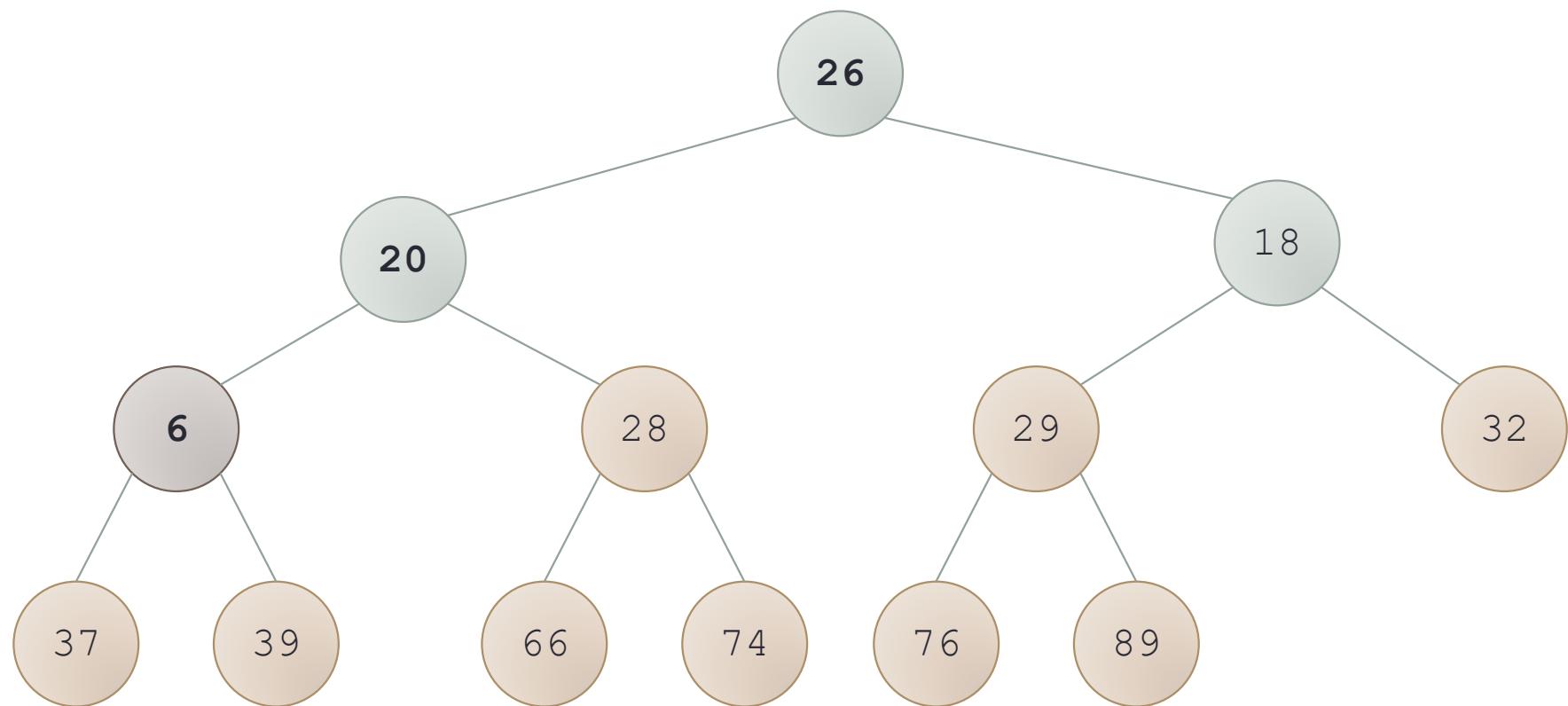
Trace of In-Place Heapsort (cont.)



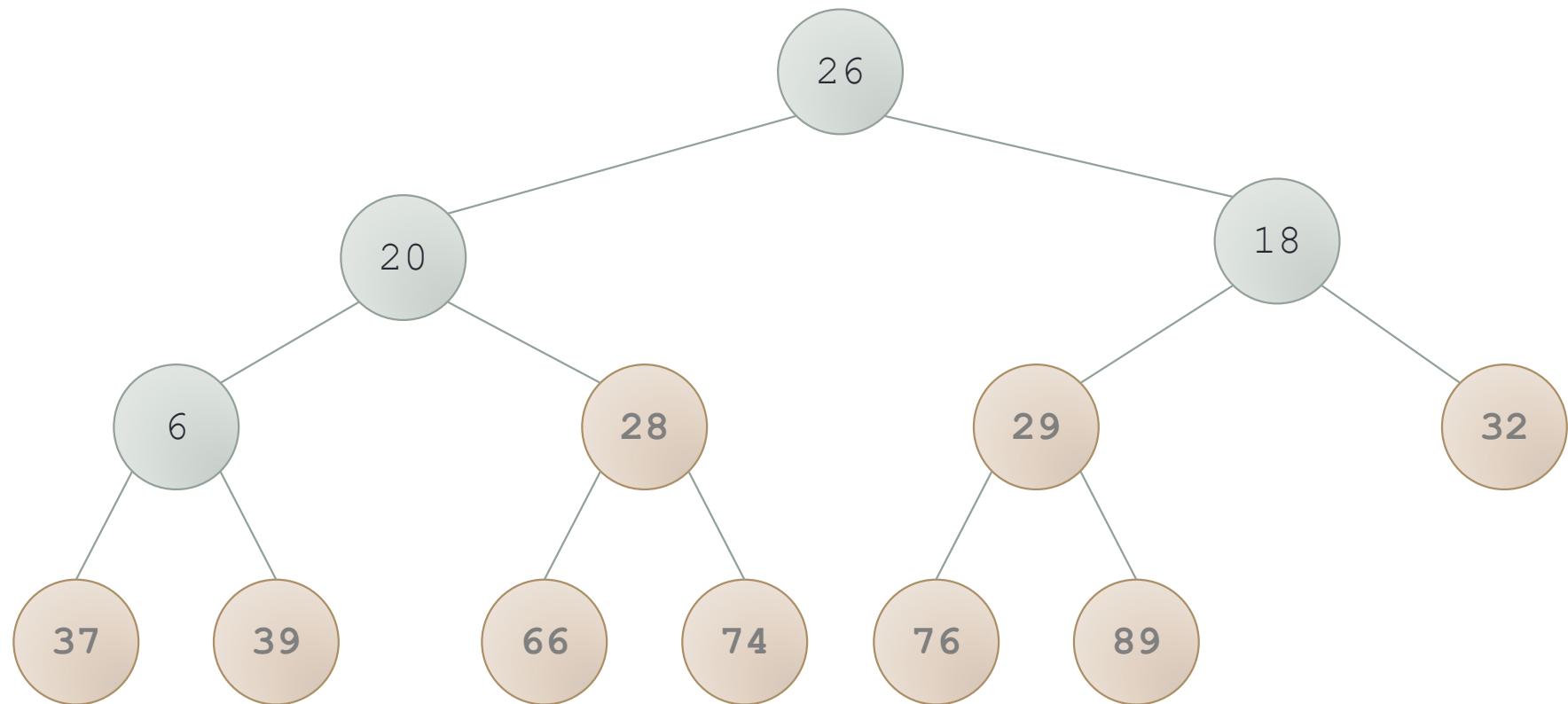
Trace of In-Place Heapsort (cont.)



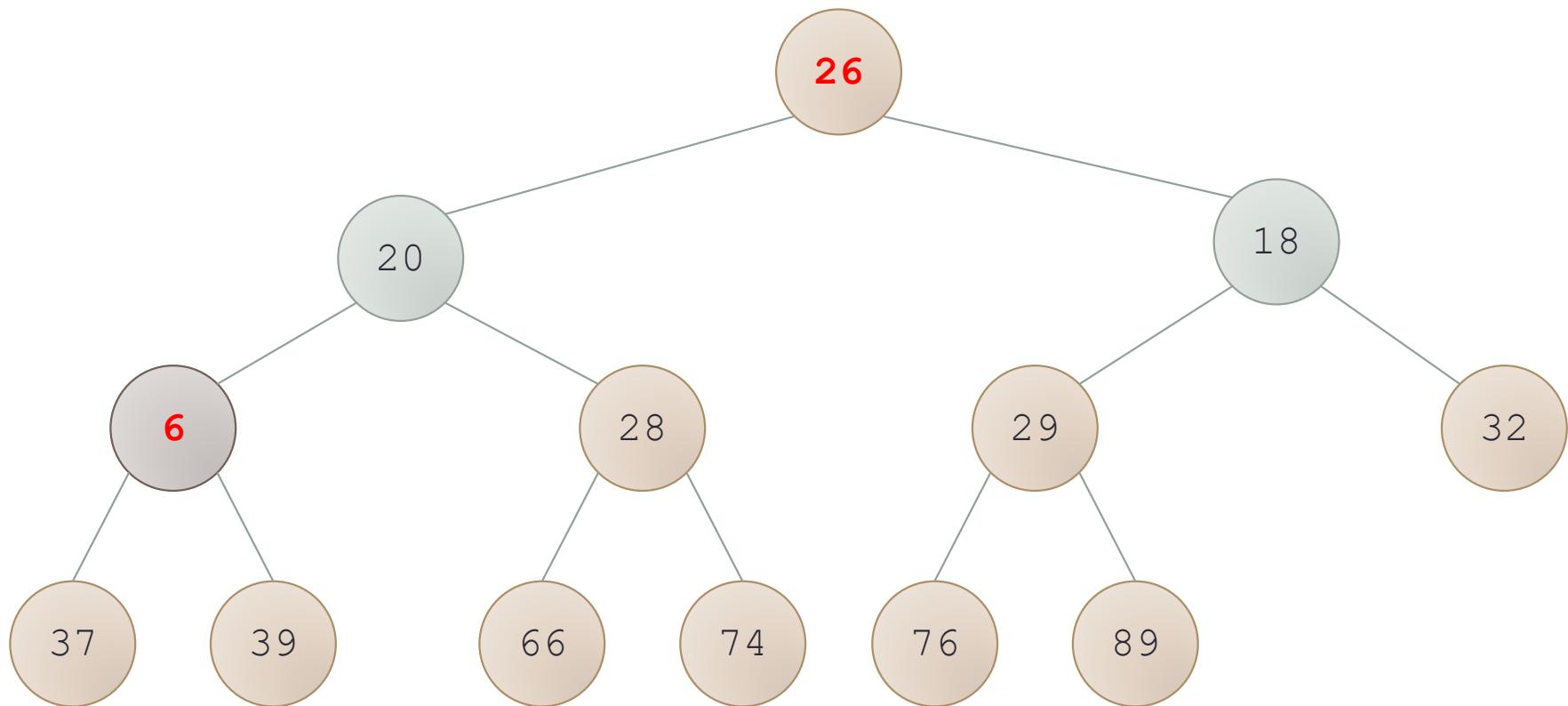
Trace of In-Place Heapsort (cont.)



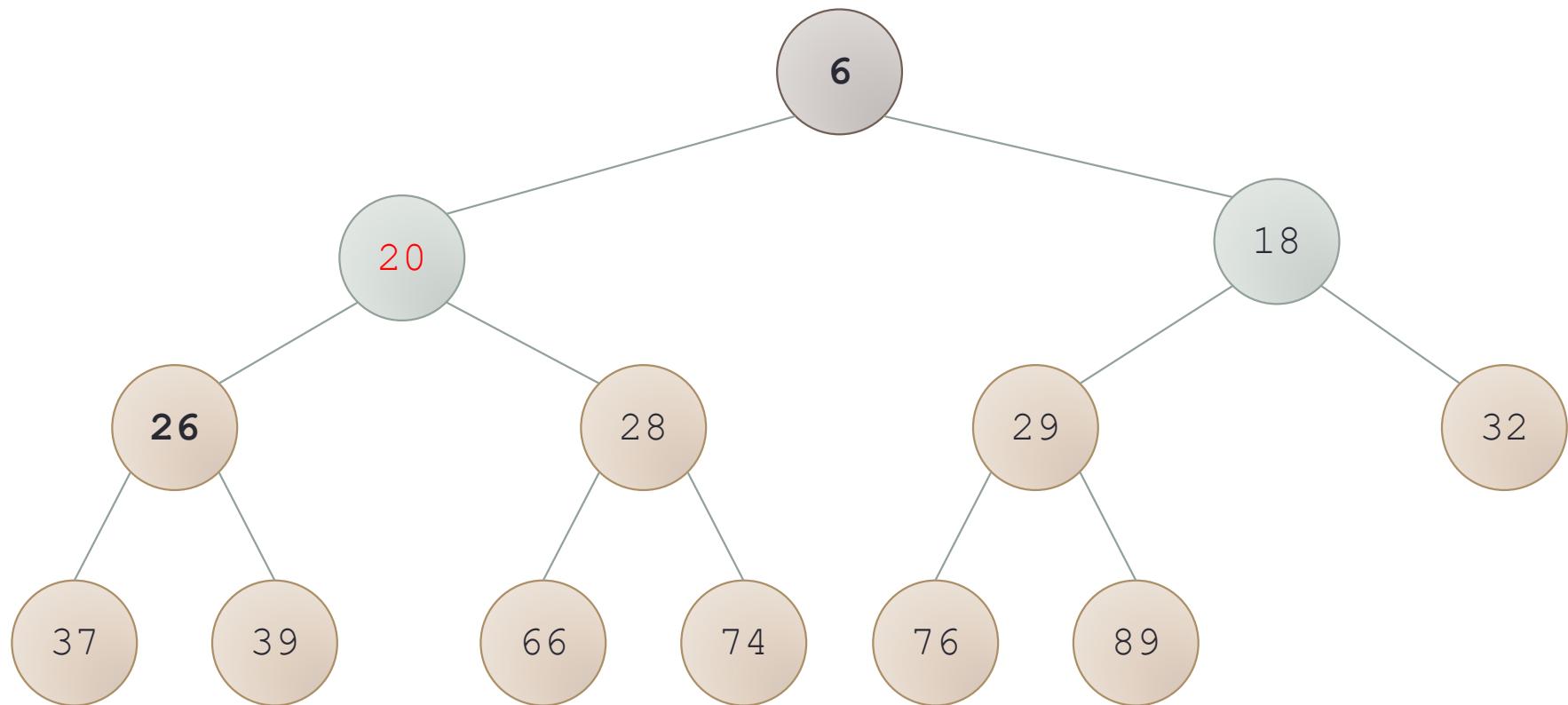
Trace of In-Place Heapsort (cont.)



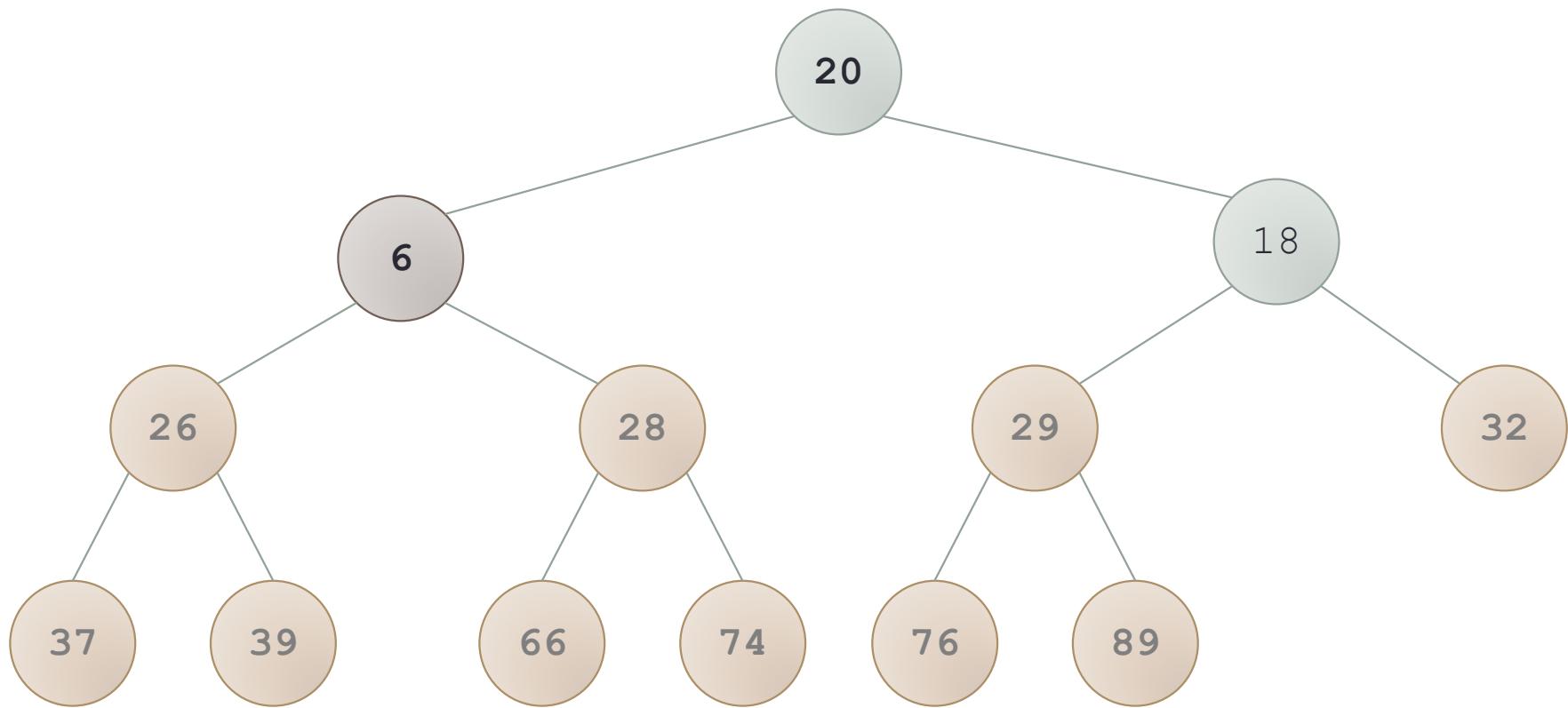
Trace of In-Place Heapsort (cont.)



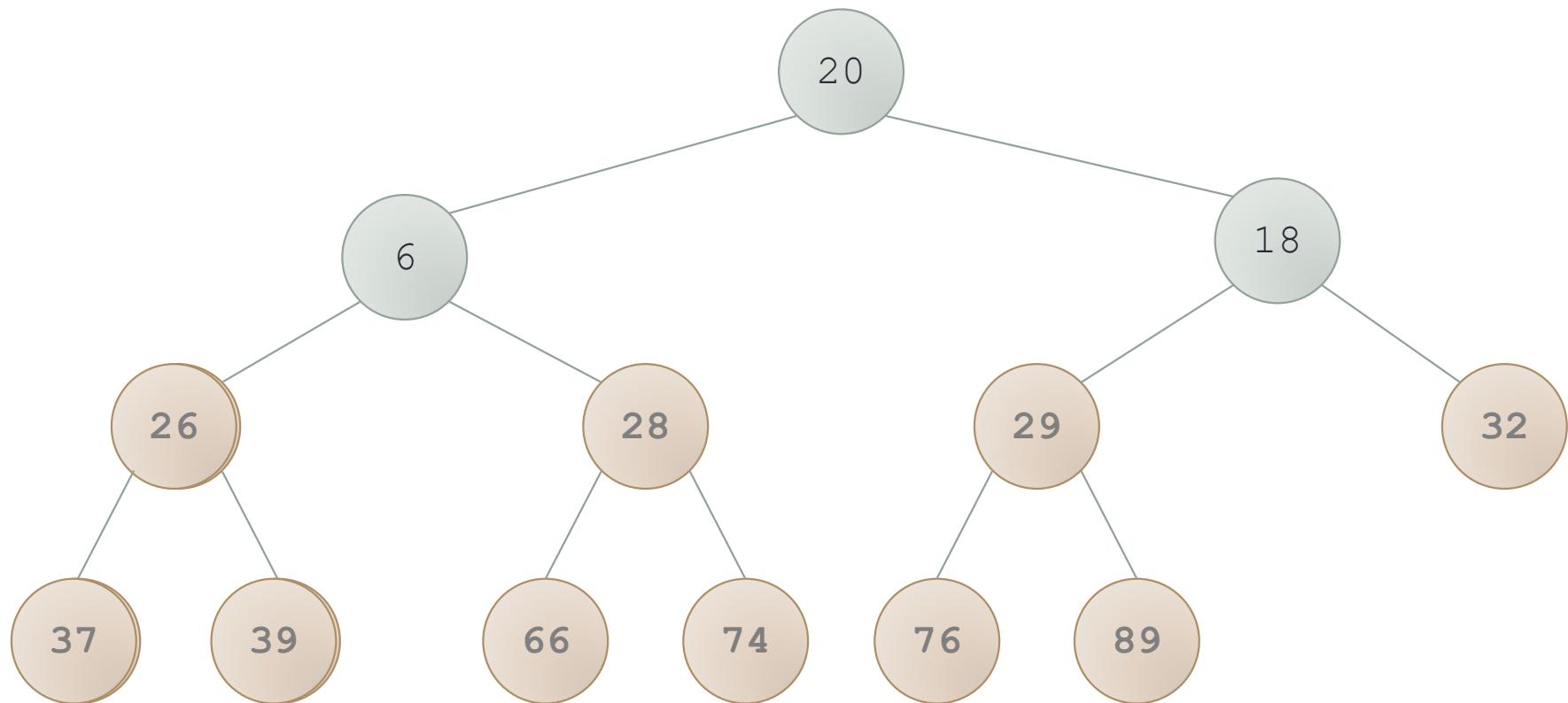
Trace of In-Place Heapsort (cont.)



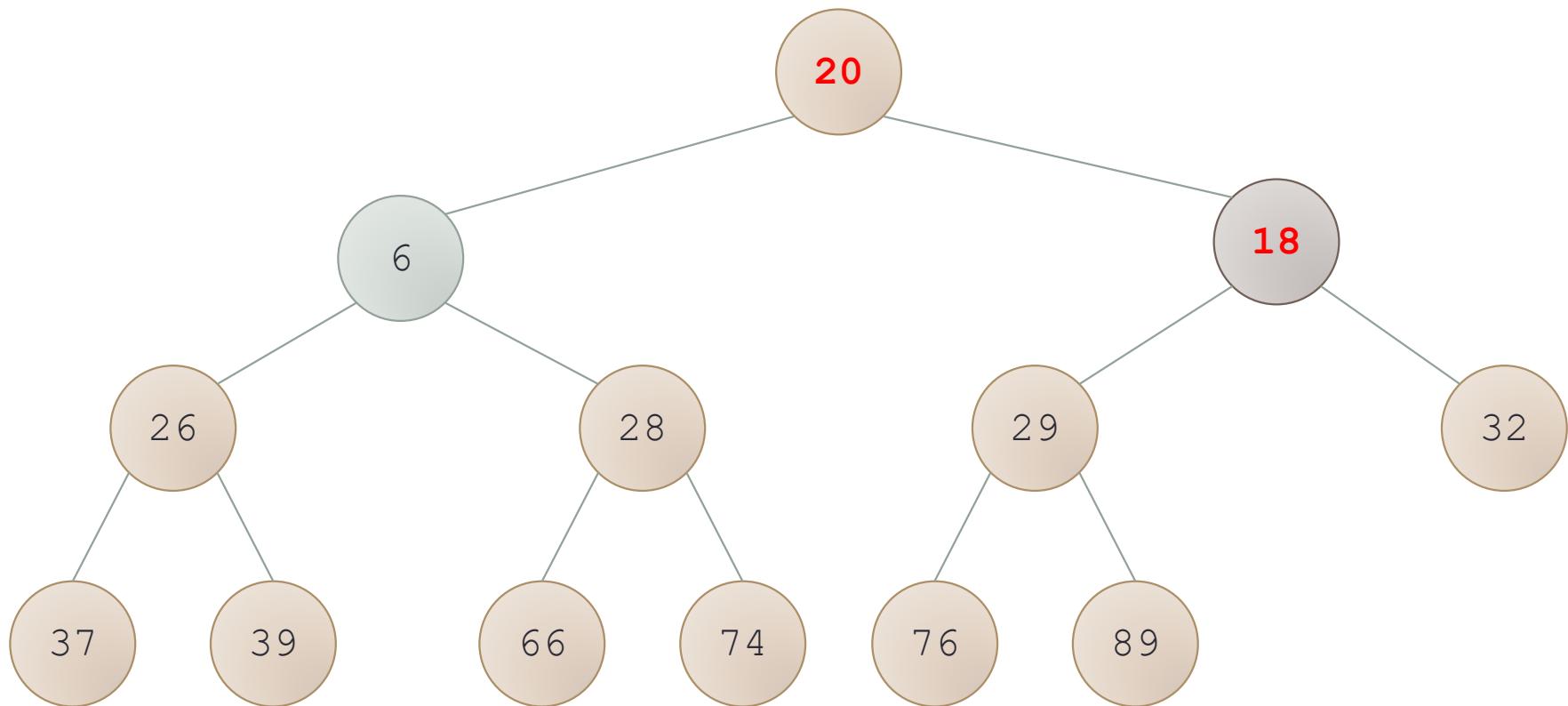
Trace of In-Place Heapsort (cont.)



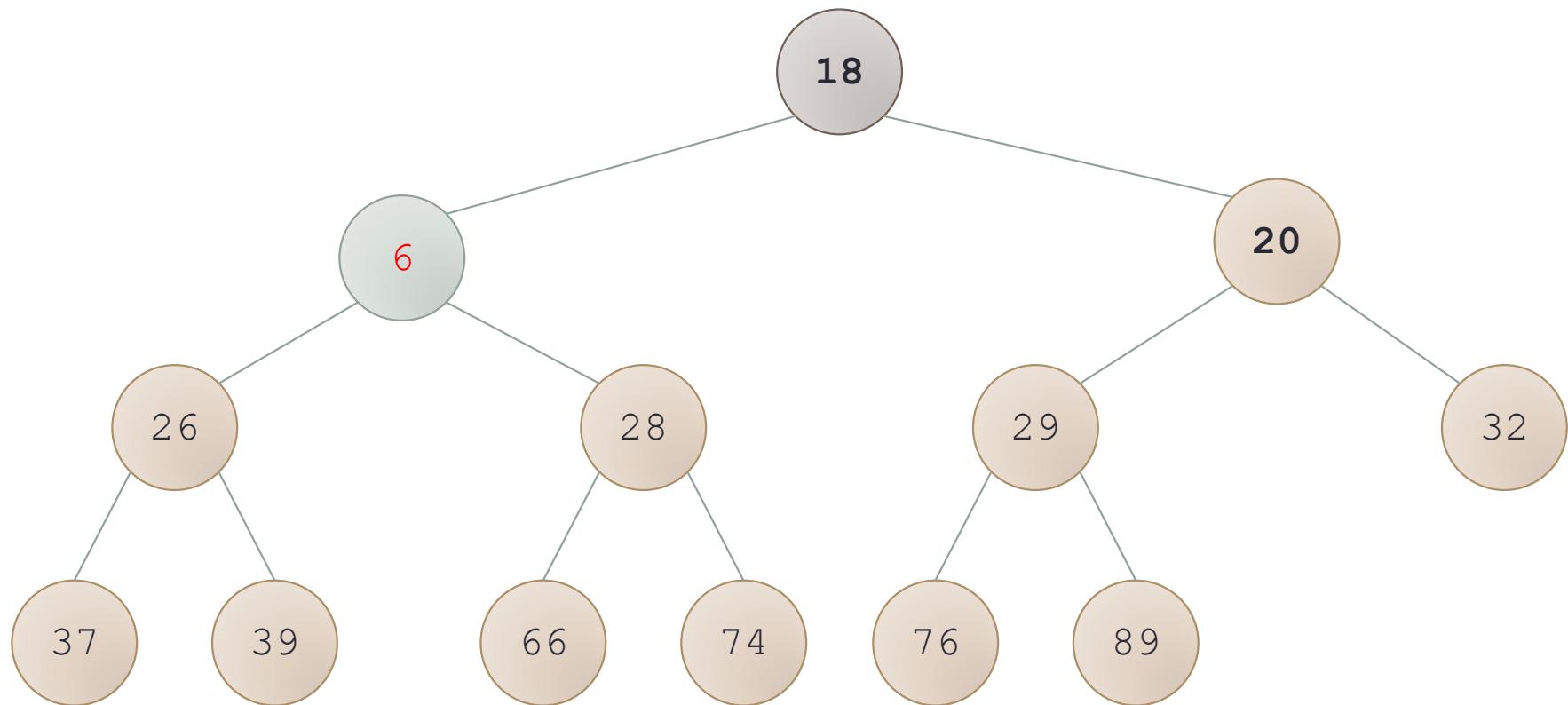
Trace of In-Place Heapsort (cont.)



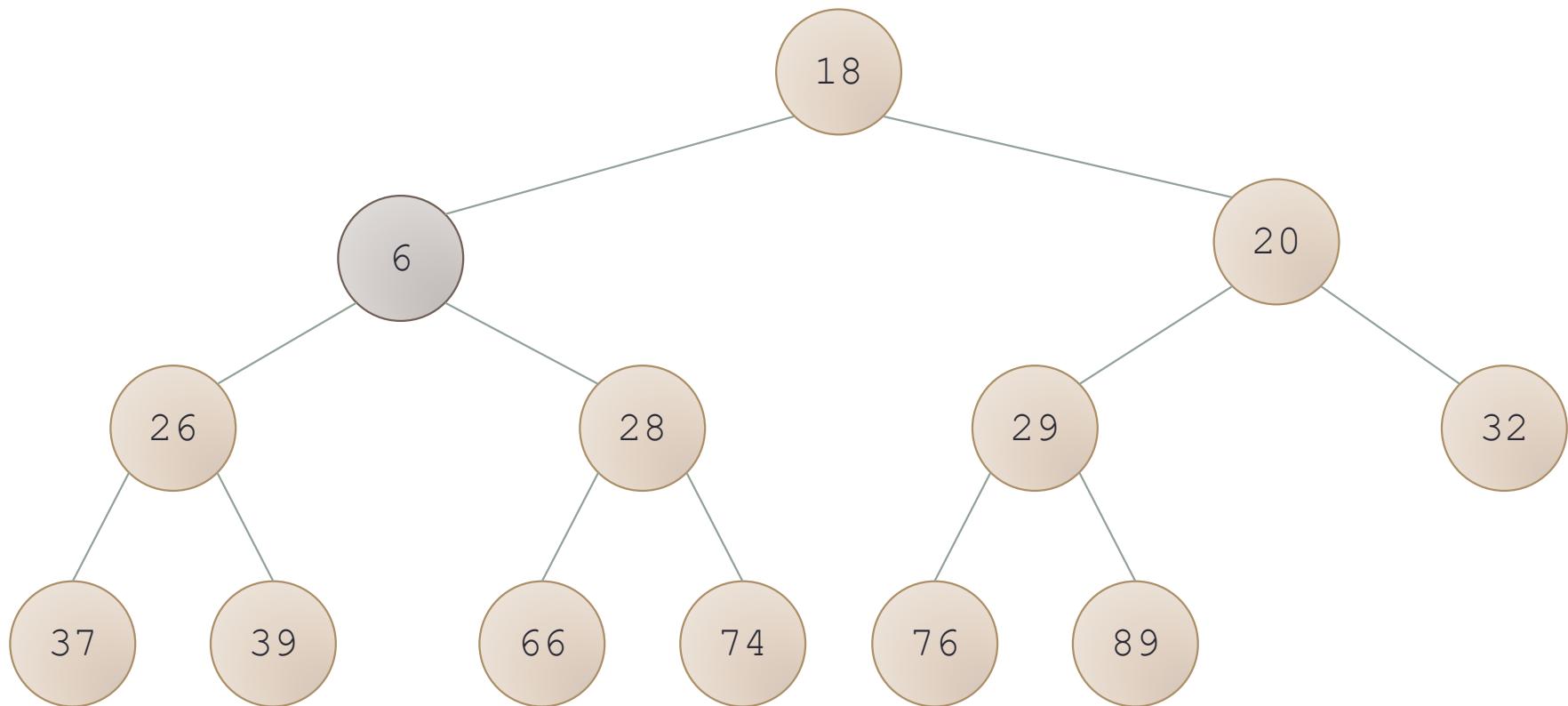
Trace of In-Place Heapsort (cont.)



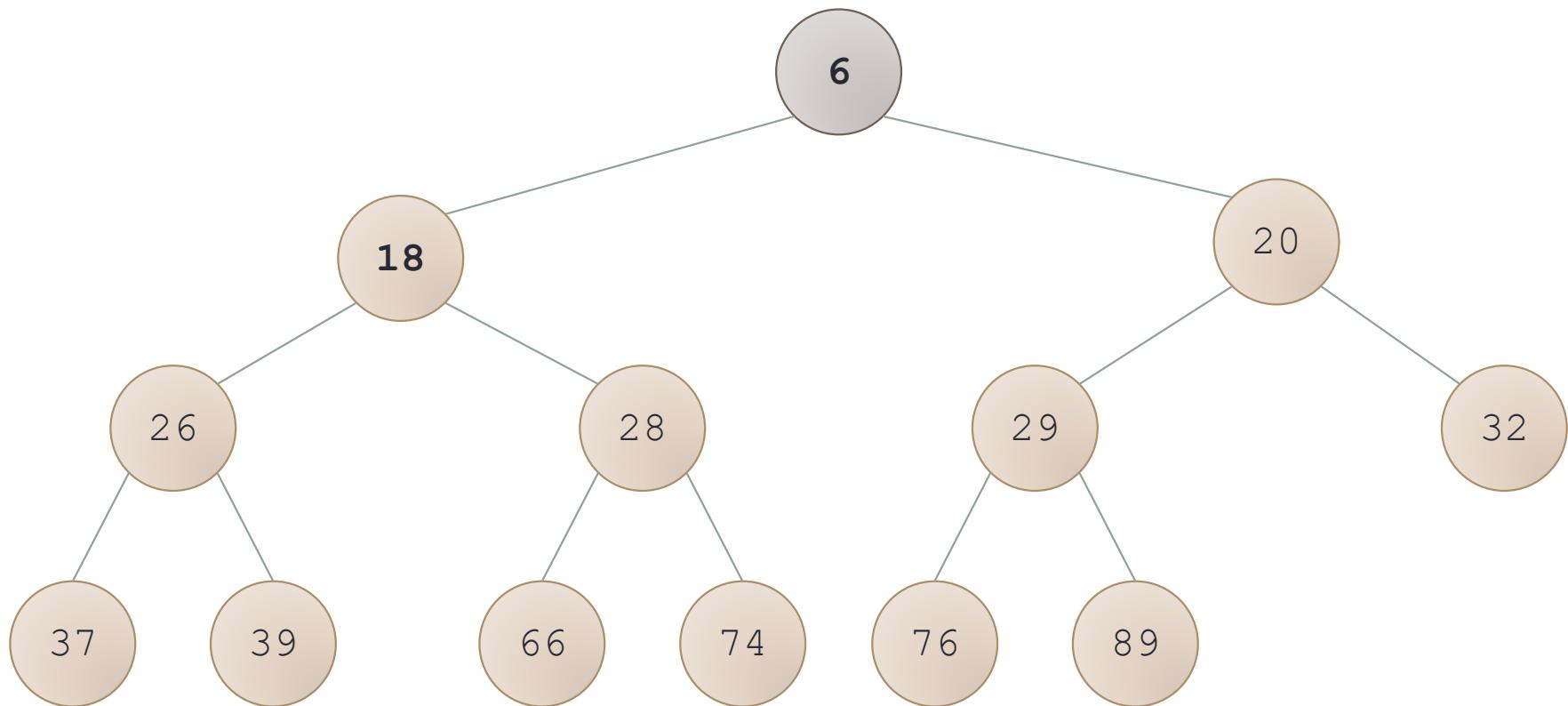
Trace of In-Place Heapsort (cont.)



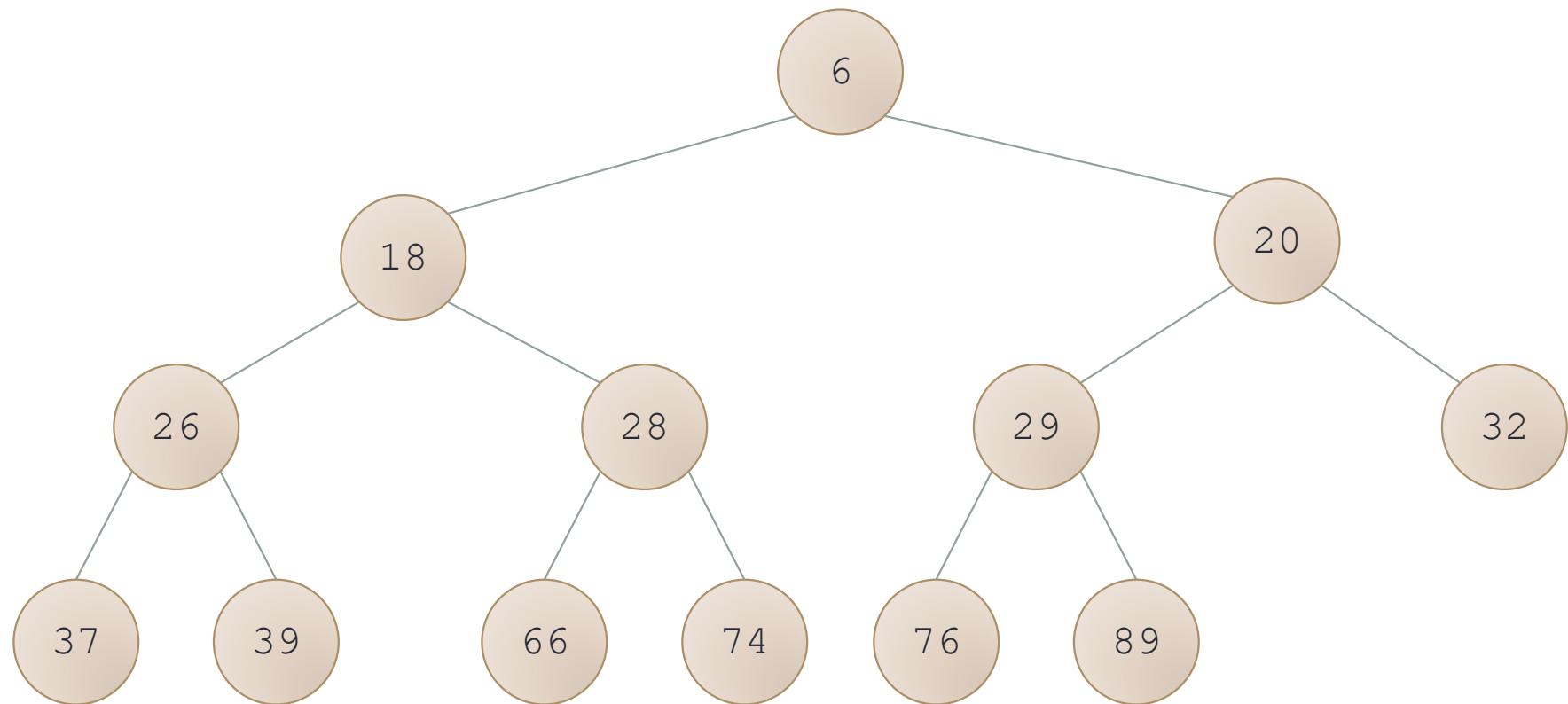
Trace of In-Place Heapsort (cont.)



Trace of In-Place Heapsort (cont.)



Trace of In-Place Heapsort (cont.)



Heap after removing all elements. Array representation of the heap is now sorted:
 $\{6, 18, 20, 26, 28, 29, 32, 37, 39, 66, 74, 76, 89\}$

Implementing the In-Place Heapsort

- If we implement the heap as an array
 - each element removed will be placed at the end of the array, and
 - the heap part of the array decreases by one element

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
89	76	74	37	32	39	66	20	26	18	28	29	6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
76	37	74	26	32	39	66	20	6	18	28	29	89

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
74	37	66	26	32	39	29	20	6	18	28	76	89

⋮

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
6	18	20	26	28	29	32	37	39	66	74	76	89

Algorithm for In-Place Heapsort

Algorithm for In-Place Heapsort

1. Build a heap by rearranging the elements in an unsorted array
2. while the heap is not empty
3. Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property

Algorithm to Build a Heap

- ❑ Start with an array `table` of length `table.length`
- ❑ Consider the first item to be a heap of one item
- ❑ Next, consider the general case where the items in array `table` from 0 through $n-1$ form a heap and the items from n through `table.length - 1` are not in the heap

Algorithm to Build a Heap (cont.)

Refinement of Step 1 for In-Place Heapsort

- 1.1 while n is less than `table.length`
- 1.2 Increment n by 1. This inserts a new item into the heap
- 1.3 Restore the heap property

Analysis of Heapsort

- ❑ Because a heap is a complete binary tree, it has $\log n$ levels
- ❑ Building a heap of size n requires finding the correct location for an item in a heap with $\log n$ levels
- ❑ Each insert (or remove) is $O(\log n)$
- ❑ With n items, building a heap is $O(n \log n)$
- ❑ No extra storage is needed

QUICKSORT

Quicksort

- Developed in 1962
- Quicksort **selects** a specific value called a **pivot** and **rearranges** the array **into two parts** (called **partitioning**)
 - all the elements **in** the **left** subarray are **less than or equal to** the **pivot**
 - all the elements **in** the **right** subarray are **larger than** the **pivot**
 - the **pivot** is placed **between** the two subarrays
- The process is **repeated until** the array is **sorted**

Trace of Quicksort

Array to be sorted

44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

Trace of Quicksort (cont.)



Arbitrarily select the first element as the **pivot**

Trace of Quicksort (cont.)



Rearrange the array elements so **all values \leq pivot** are on **left** (in pink) and **all values $>$ pivot** are on **right** (in gray)

Trace of Quicksort (cont.)



Swap the **pivot** value with **last value \leq pivot**

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75
----	----	----	----	----	----	----	----	----

Pivot value is in correct position, color is green

Trace of Quicksort (cont.)



Now apply **quicksort recursively** to the two subarrays,
starting with the **left** subarray

Trace of Quicksort (cont.)

Pivot value is 12



Trace of Quicksort (cont.)

Pivot value is **12**;
12 is **smallest** value in left subarray



Trace of Quicksort (cont.)

12 is in correct position, next pivot value is **33**



Trace of Quicksort (cont.)

values ≤ 33 are in gray, values > 33 are in pink



Trace of Quicksort (cont.)

Swap 23 and 33;
33 is in correct position



Trace of Quicksort (cont.)



Left and right subarrays have single values so they are sorted.

Trace of Quicksort (cont.)

Left subarray of original array is sorted;
sort right subarray

12	23	33	43	44	55	64	77	75
----	----	----	----	----	----	----	----	----

Trace of Quicksort (cont.)

Pivot value is 55, 55 is smallest value so it is in correct position



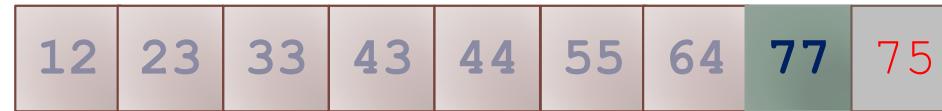
Trace of Quicksort (cont.)

Pivot value is 64; 64 is **smallest** value so it is in correct position



Trace of Quicksort (cont.)

Pivot value is 77; swap it with 75



Trace of Quicksort (cont.)

77 is in correct position



Trace of Quicksort (cont.)



Left subarray has single value;
it is sorted

Trace of Quicksort (cont.)

Sorted Array

12	23	33	43	44	55	64	75	77
----	----	----	----	----	----	----	----	----

Algorithm for Quicksort

- We describe how to do the partitioning later
- The indexes `first` and `last` are the `end points` of the `array` being sorted
- The `index` of the `pivot` after partitioning is `pivIndex`

Algorithm for Quicksort

1. if `first < last` then
2. Partition the elements in the subarray `first . . . last` so that the pivot value is in its correct place (subscript `pivIndex`)
3. Recursively apply quicksort to the subarray `first . . . pivIndex - 1`
4. Recursively apply quicksort to the subarray `pivIndex + 1 . . . Last`

Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray,
 - then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be $\log n$ levels of recursion
 - At each recursion level, the partitioning process involves moving every element to its correct position— n moves
- Quicksort is $O(n \log n)$, just like merge sort

Analysis of Quicksort (cont.)

- The array split **may not be the best case**, i.e. 50-50
 - ✓ An exact analysis is **difficult** (and beyond the scope of this class), but, the running time will be bounded by a constant $\times n \log n$
- A quicksort will give very **poor behavior** if, each time the array is partitioned, **a subarray is empty**.
 - ✓ In that case, the sort will be **$O(n^2)$**

```

public class QuickSort {
    /** Sort the table using the quicksort algorithm.
     * pre: table contains Comparable objects, post: table is sorted.
     * @param table The array to be sorted */
    public static <T extends Comparable<T>> void sort(T[] table) {
        quickSort(table, 0, table.length - 1); //Call recursive quicksort
    }

    /** Sort a part of the table using the quicksort algorithm.
     * post: The part of table from first through last is sorted.
     * @param table The array to be sorted
     * @param first The index of the low bound
     * @param last The index of the high bound */
    private static <T extends Comparable<T>> void quickSort(T[] table, int first, int last) {
        if (first < last) { // There is data to be sorted.
            int pivIndex = partition(table, first, last); // Partition the array
            quickSort(table, first, pivIndex - 1); // Sort the left half.
            quickSort(table, pivIndex + 1, last); // Sort the right half.
        }
    }
    // Insert partition()
}

```

Algorithm for Partitioning

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize up to `first` and down to `last`.
3. `do`
4. Increment up until up selects the first element greater than the pivot value or up has reached `last`.
5. Decrement down until down selects the first element less than or equal to the pivot value or down has reached `first`.
6. `if up < down then`
7. Exchange `table[up]` and `table[down]`.
8. `while up is to the left of down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.