SORTING

Chapter 8

Chapter Objectives

- To learn how to use the standard sorting methods in the Java API
- To learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - Shell sort
  - merge sort
  - heapsort
  - quicksort
- To understand the differences in performance of these algorithms, and which to use for small, medium, and large arrays

Introduction

- Sorting entails arranging data in order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight into
  - problem solving techniques such as *divide and conquer*
  - the analysis and comparison of algorithms which perform the same task

Using Java Sorting Methods

- The Java API provides a class `Arrays` with several overloaded sort methods for different array types
- The `Collections` class provides similar sorting methods for `Lists`
- Sorting methods for arrays of primitive types are based on the quicksort algorithm
- Sorting methods for arrays of objects and `Lists` are based on the merge sort algorithm
- Both algorithms are O(n log n)

Method sort in Class Arrays	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] 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 and must be mutually comparable.
public static void sort(Object[] items, int fromIndex, int toIndex)	Sorts array elements items[fromIndex] to items[toIndex] in ascending order using their natural ordering (defined by method compareTo). All objects must implement the Comparable interface and must be mutually comparable.
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.
Method sort in Class Collections	**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 compareTo). 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.

Declaring a Generic Method

SYNTAX **Declaring a Generic Method**

FORM:

methodModifiers *<genericParameters>* *returnType* *methodName*(*methodParameters*)

EXAMPLE:

```
public static <T extends Comparable<T>> int binarySearch(T[] items,
                                                          T target)
```

MEANING:

To declare a generic method, list the *genericParameters* inside the symbol pair <> and between the *methodModifiers* (e.g., `public static`) and the return type. The *genericParameters* can then be used in the specification of the *methodParameters*.

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

`T` represents the generic parameter for the sort method

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

`T` should also appear in the method parameter list

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

The second method parameter means that `comp` must be an object that implements the `Comparator` interface for type `T` or for a superclass of type `T`

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

For example, you can define a class that implements `Comparator<Number>` and use it to sort an array of `Integer` objects or an array of `Double` objects

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

`<T extends Comparable<T>>` means that generic parameter `T` must implement the interface `Comparable<T>`

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

The method parameter `list` (the object being sorted) is of type `List<T>`

```
public class Person implements Comparable<Person> {
     private String lastName;
     private String firstName;
     /* Birthday represented by an integer from 1 to 366 */
     private int birthDay;

     // Methods
     /** Compares two Person objects based on names. The result is based on the
         last names if they are different; otherwise, it is based on first names.
         @param obj The other Person
         @return A negative integer if this person's name
              precedes the other person's name;
              0 if the names are the same;
              a positive integer if this person's name follows other person's name.
     */
     @Override
     public int compareTo(Person other) {
         // Compare this Person to other using last names.
         int result = lastName.compareTo(other.lastName);
         // Compare first names if last names are the same.
         if (result == 0)
          return firstName.compareTo(other.firstName);
         else
          return result;
     }

     // Other methods
     . . .
}
```

Example 8.3

```
import java.util.Comparator;

public class ComparePerson implements Comparator<Person> {
     /** Compare two Person objects based on birth date.
         @param left The left-hand side of the comparison
         @param right The right-hand side of the comparison
         @return A negative integer if the left person's birthday
                 precedes the right person's birthday;
                 0 if the birthdays are the same;
                 a positive integer if the left person's birthday
                 follows the right person's birthday.
     */
     @Override
     public int compare(Person left, Person right) {
          return left.getBirthDay() - right.getBirthDay();
     }
}
```

Selection Sort

Section 8.2

Selection Sort

- Selection sort is relatively easy to understand
- It sorts an array by making several passes through the array, selecting the next smallest item in the array each time and placing it where it belongs in the array
  - While the sort algorithms are not limited to arrays, throughout this chapter we will sort arrays for simplicity
- All items to be sorted must be `Comparable` objects, so, for example, any `int` values must be wrapped in `Integer` objects

Trace of Selection Sort

`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`

0	1	2	3	4
35	65	30	60	20

n	5
fill	
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`

0	1	2	3	4
35	65	30	60	20

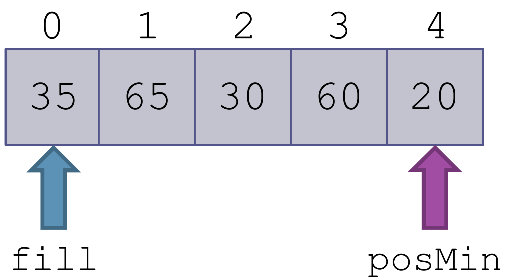
fill

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`

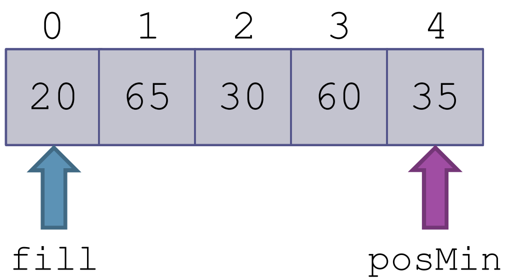


n	5
fill	0
posMin	4

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`

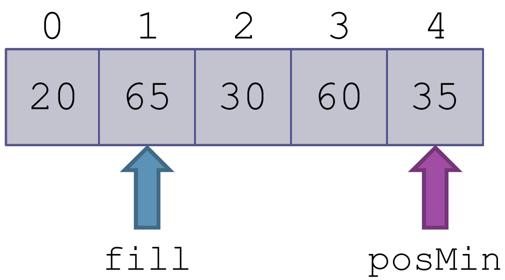


n	5
fill	0
posMin	4

Trace of Selection Sort (cont.)

`n` = number of elements in the array

1. **for** `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`



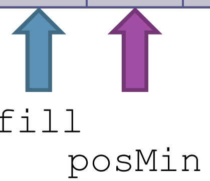
n	5
fill	1
posMin	4

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`

0	1	2	3	4
20	65	30	60	35

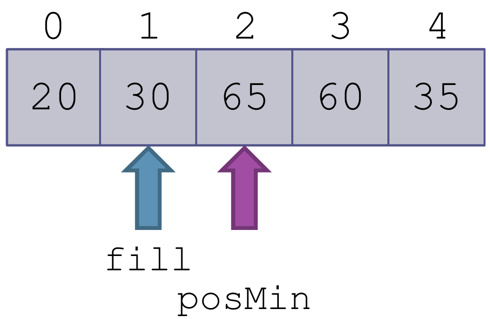


n	5
fill	1
posMin	2

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`

0	1	2	3	4
20	30	65	60	35

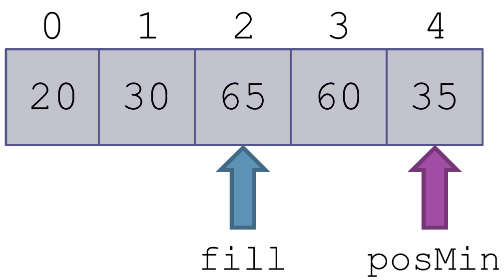
fill
posMin

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`

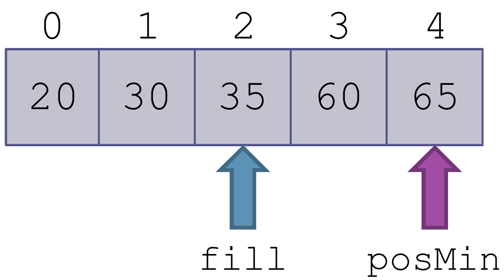


n	5
fill	2
posMin	4

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`

0	1	2	3	4
20	30	35	60	65

fillposMin

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`

0	1	2	3	4
20	30	35	60	65

`fill`

`posMin`

n	5
fill	3
posMin	3

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`

0	1	2	3	4
20	30	35	60	65

fill

posMin

n	5
fill	3
posMin	3

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`

0	1	2	3	4
20	30	35	60	65

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

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 (cont.)

`n`	5
`fill`	0
`posMin`	0
`next`	1

0	1	2	3	4
35	65	30	60	20

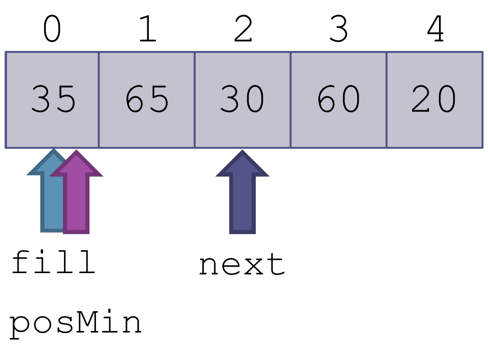
`fill` `next`

`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 (cont.)

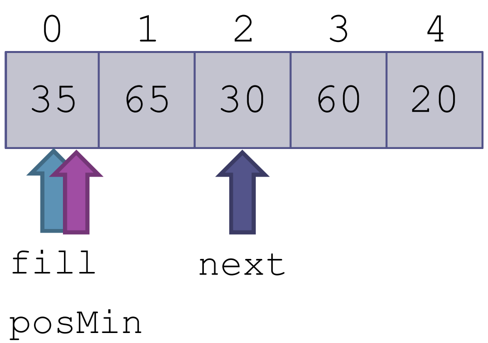
n	5
fill	0
posMin	0
next	2



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 (cont.)

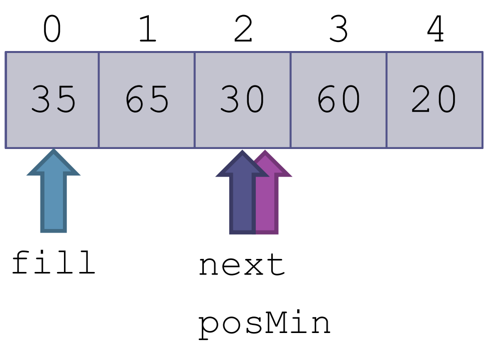
`n`	5
`fill`	0
`posMin`	0
`next`	2



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 (cont.)

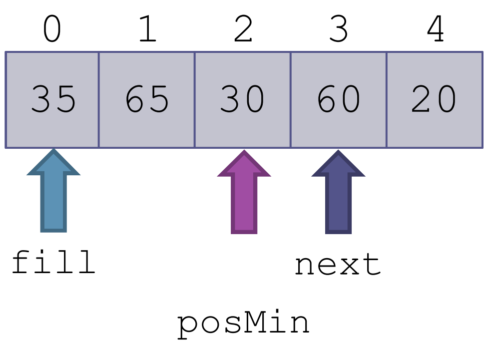
`n`	5
`fill`	0
`posMin`	2
`next`	2



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 (cont.)

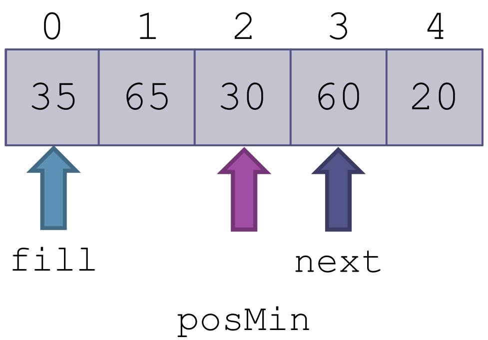
n	5
fill	0
posMin	2
next	3



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 (cont.)

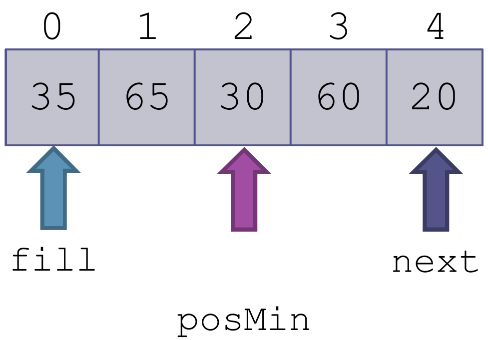
n	5
fill	0
posMin	2
next	3



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 (cont.)

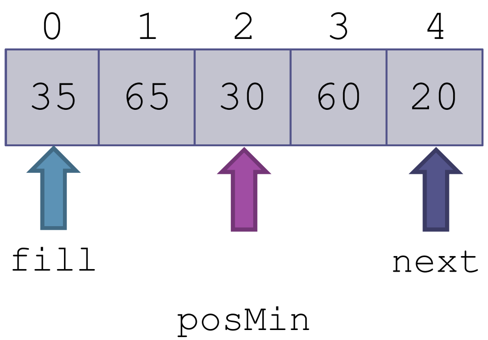
n	5
fill	0
posMin	2
next	4



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 (cont.)

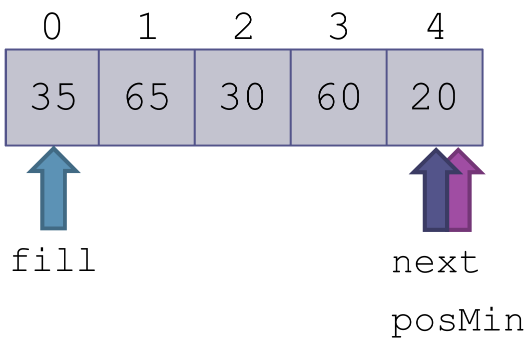
n	5
fill	0
posMin	2
next	4



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 (cont.)

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 (cont.)

`n`	5
`fill`	0
`posMin`	4
`next`	4

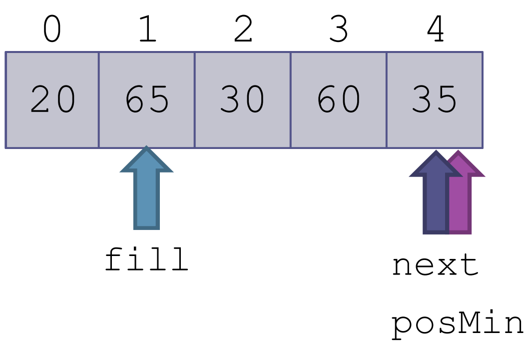
0	1	2	3	4
20	65	30	60	35

`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 (cont.)

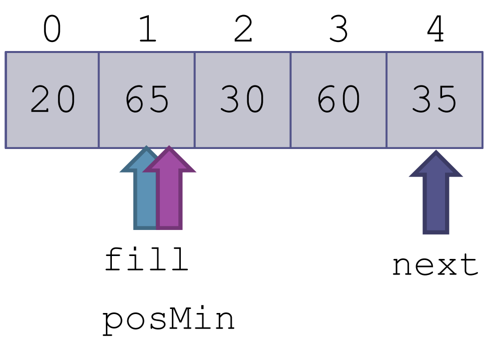
`n`	5
`fill`	1
`posMin`	4
`next`	4



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 (cont.)

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 (cont.)

`n`	5
`fill`	1
`posMin`	1
`next`	2

0	1	2	3	4
20	65	30	60	35

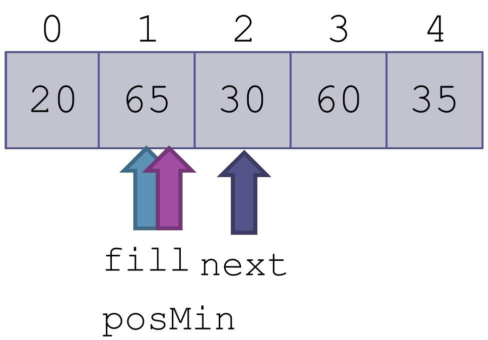
`fill` `next`

`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 (cont.)

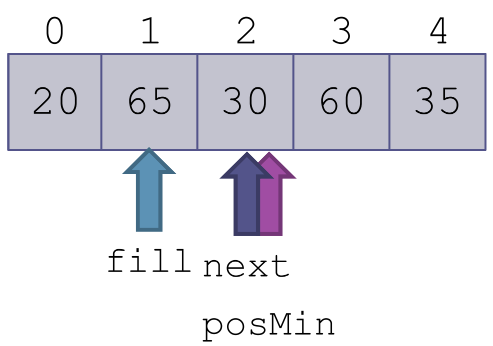
n	5
fill	1
posMin	1
next	2




```
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 (cont.)

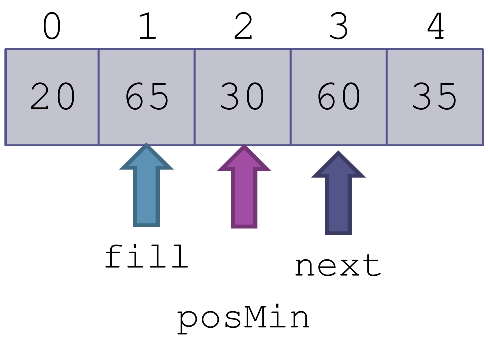
`n`	5
`fill`	1
`posMin`	2
`next`	2



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 (cont.)

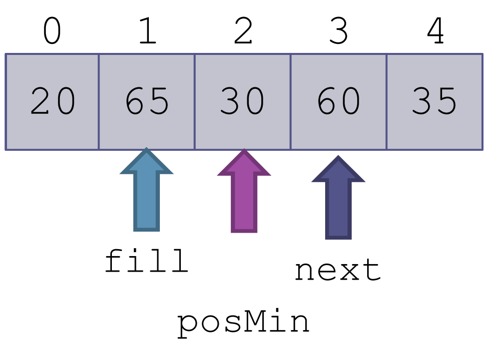
`n`	5
`fill`	1
`posMin`	2
`next`	3



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 (cont.)

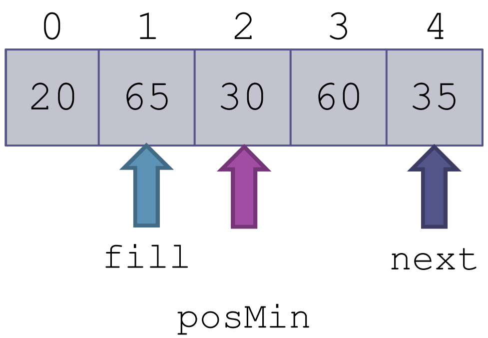
n	5
fill	1
posMin	2
next	3



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 (cont.)

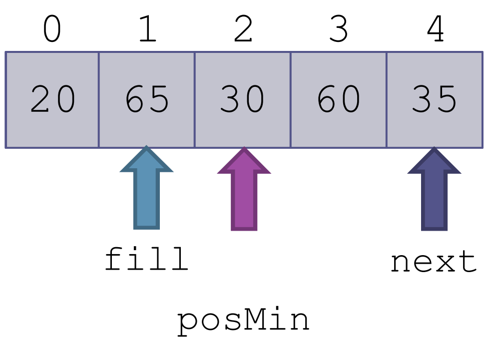
n	5
fill	1
posMin	2
next	4



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 (cont.)

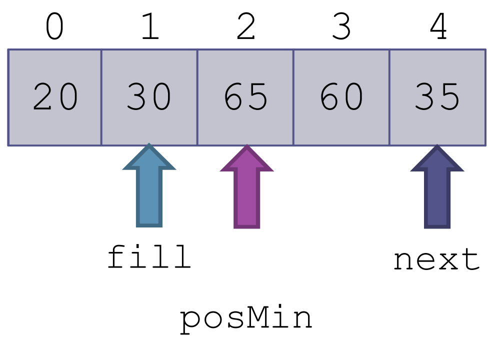
n	5
fill	1
posMin	2
next	4



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 (cont.)

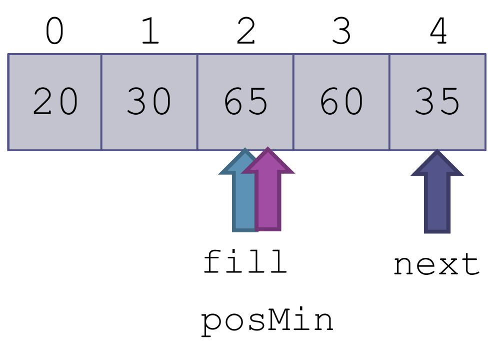
`n`	5
`fill`	1
`posMin`	2
`next`	4



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 (cont.)

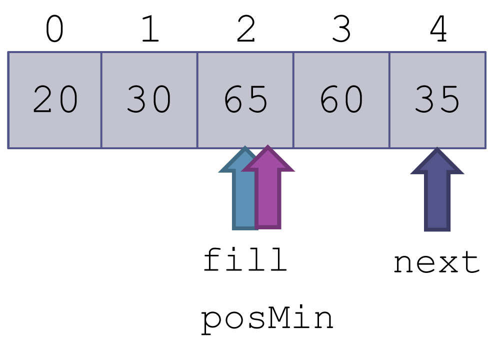
n	5
fill	2
posMin	2
next	4



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 (cont.)

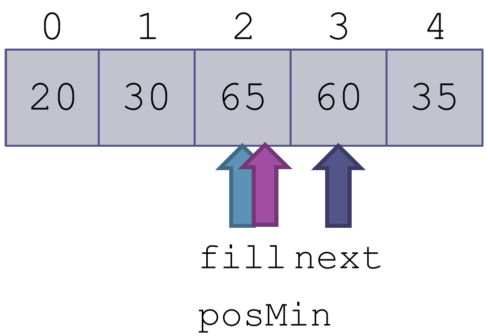
`n`	5
`fill`	2
`posMin`	2
`next`	4



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 (cont.)

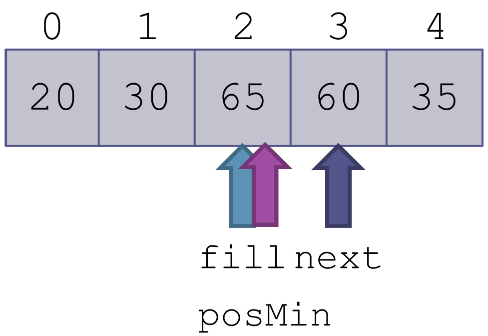
`n`	5
`fill`	2
`posMin`	2
`next`	3



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 (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 (cont.)

n	5
fill	2
posMin	3
next	3

0	1	2	3	4
20	30	65	60	35

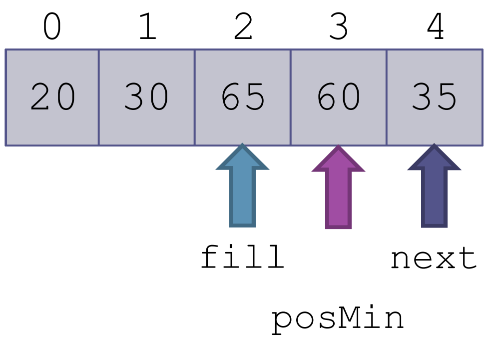
fill next

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 (cont.)

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 (cont.)

`n`	5
`fill`	2
`posMin`	3
`next`	4

0	1	2	3	4
20	30	65	60	35

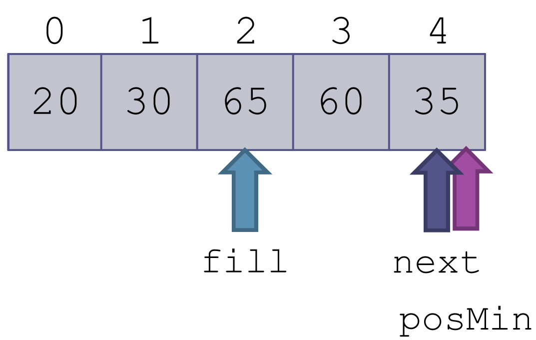
`fill` `next`

`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 (cont.)

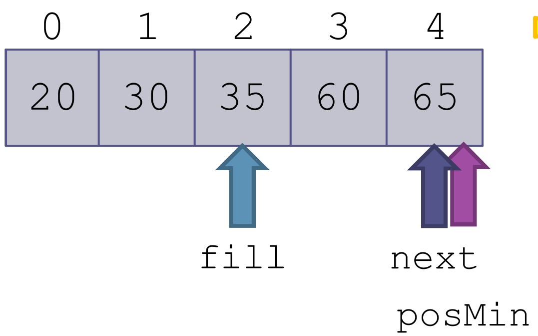
`n`	5
`fill`	2
`posMin`	4
`next`	4



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 (cont.)

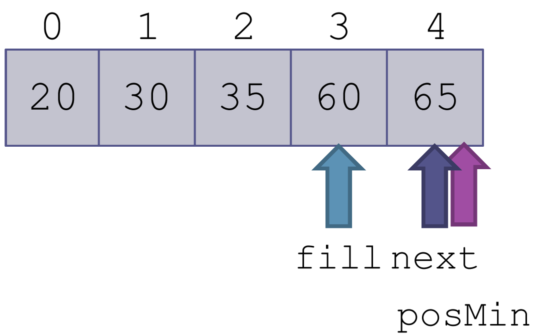
`n`	5
`fill`	2
`posMin`	4
`next`	4



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 (cont.)

`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 (cont.)

n	5
fill	3
posMin	3
next	4

0	1	2	3	4
20	30	35	60	65

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 (cont.)

`n`	5
`fill`	3
`posMin`	3
`next`	4

0	1	2	3	4
20	30	35	60	65

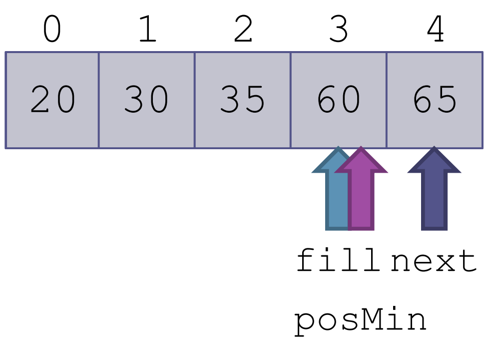
`fill` `next`

`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 (cont.)

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 (cont.)

n	5
fill	3
posMin	3
next	4

0	1	2	3	4
20	30	35	60	65

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 (cont.)

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.)

There are n-1 exchanges

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.)

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) + ... + 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
$(n\text{-}1) + (n\text{-}2) + \ldots + 3 + 2 + 1$
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`

Analysis of Selection Sort (cont.)

For very large n we can ignore all but the significant term in the expression, so the number of
- comparisons is $O(n^2)$
- exchanges is $O(n)$

An $O(n^2)$ sort is called a *quadratic sort*

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`

```java
public class SelectionSort implements SortAlgorithm {

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

Making Sort Methods Generic

- To avoid a warning message about an unchecked call to `compareTo`, change the method heading to

```
public static <T extends Comparable<T>> void sort(T[] table {
```

and change the variable `temp` from `Comparable` to type `T`

```
T temp = table[fill];
```

Bubble Sort

Section 8.3

Bubble Sort

- Also a quadratic sort
- Compares adjacent array elements and exchanges their values if they are out of order
- Smaller values *bubble* up to the top of the array and larger values sink to the bottom; hence the name

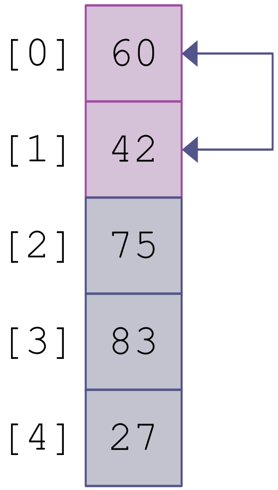
Trace of Bubble Sort

[0]	60
[1]	42
[2]	75
[3]	83
[4]	27

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	1
exchanges made	0




```
1. do
2.     for each pair of adjacent array elements
3.         if the values in a pair are out of order
4.             Exchange the values
5. while the array is not sorted
```

Trace of Bubble Sort (cont.)

pass	1
exchanges made	1

[0]	42
[1]	60
[2]	75
[3]	83
[4]	27

```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

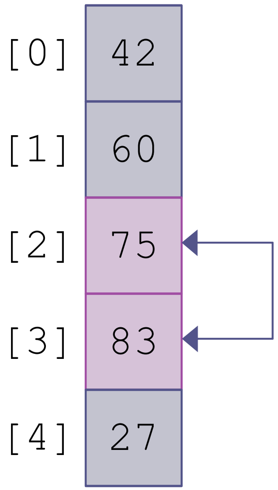
pass	1
exchanges made	1

[0]	42
[1]	60
[2]	75
[3]	83
[4]	27

```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

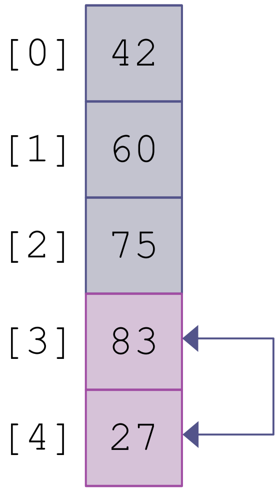
pass	1
exchanges made	1



1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

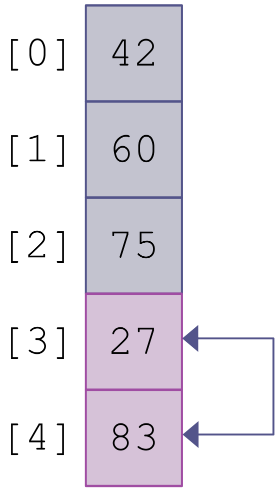
pass	1
exchanges made	1




```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

pass	1
exchanges made	1



1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	1
exchanges made	2

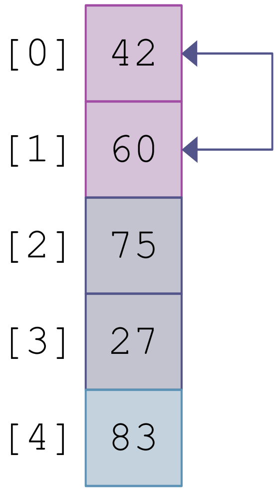
[0]	42
[1]	60
[2]	75
[3]	27
[4]	83

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

At the end of pass 1, the last item (index [4]) is guaranteed to be in its correct position. There is no need to test it again in the next pass

Trace of Bubble Sort (cont.)

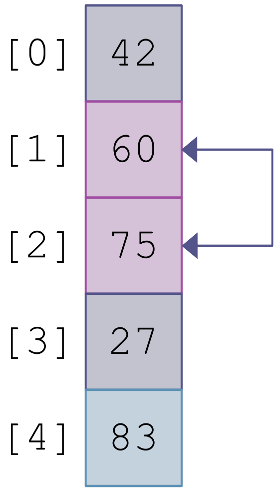
pass	2
exchanges made	0




```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

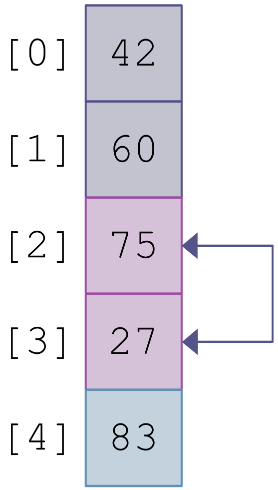
pass	2
exchanges made	0



1. **`do`**
2.   **`for`** each pair of adjacent array elements
3.   **`if`** the values in a pair are out of order
4.   Exchange the values
5. **`while`** the array is not sorted

Trace of Bubble Sort (cont.)

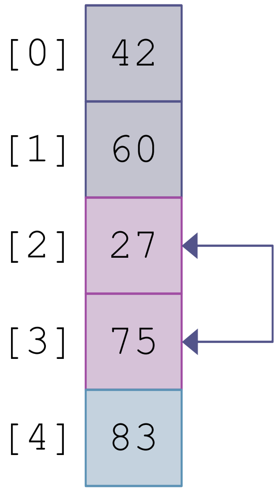
pass	2
exchanges made	0




```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

pass	2
exchanges made	1




```
1. do
2.     for each pair of adjacent array elements
3.         if the values in a pair are out of order
4.             Exchange the values
5. while the array is not sorted
```

Trace of Bubble Sort (cont.)

pass	2
exchanges made	1

[0]	42
[1]	60
[2]	27
[3]	75
[4]	83

1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

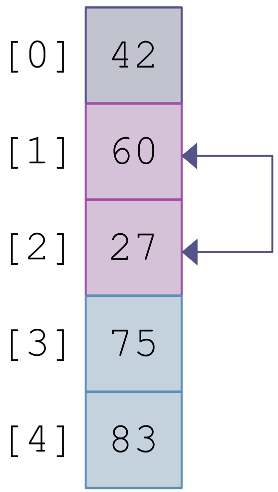
pass	3
exchanges made	0

[0]	42
[1]	60
[2]	27
[3]	75
[4]	83

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	3
exchanges made	0



1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	3
exchanges made	1

[0]	42
[1]	27
[2]	60
[3]	75
[4]	83

```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

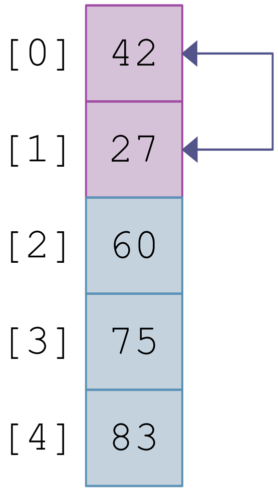
pass	3
exchanges made	1

[0]	42
[1]	27
[2]	60
[3]	75
[4]	83

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	4
exchanges made	0



1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

```
do
    for each pair of adjacent array elements
        if the values in a pair are out of order
            Exchange the values
while the array is not sorted
```

Trace of Bubble Sort (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

Trace of Bubble Sort (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted

Where n is the length of the array, after the completion of $n – 1$ passes (4, in this example) the array is sorted

Trace of Bubble Sort (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted

Sometimes an array will be sorted before
n – 1 passes. This can be detected if there are no exchanges made during a pass through the array

Trace of Bubble Sort (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. **do**
2.   **for** each pair of adjacent array elements
3.   **if** the values in a pair are out of order
4.   Exchange the values
5. **while** the array is not sorted

The algorithm can be modified to detect exchanges (next)

Trace of Bubble Sort (cont.)

`pass`	4
`exchanges made`	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. **do**
2. Initialize `exchanges` to **`false`**
3. **`for`** each pair of adjacent array elements
4. **`if`** the values in a pair are out of order
5. Exchange the values
6. Set `exchanges` to **`true`**
7. **`while`** `exchanges` is **`true`**

The algorithm can be modified to detect exchanges

Analysis of Bubble Sort

- The number of comparisons and exchanges is represented by

$$(n - 1) + (n - 2) + ... + 3 + 2 + 1$$

- Worst case:
  - number of comparisons is $O(n^2)$
  - number of exchanges is $O(n^2)$
- Compared to selection sort with its $O(n^2)$ comparisons and $O(n)$ exchanges, bubble sort usually performs worse
- If the array is sorted early, the later comparisons and exchanges are not performed and performance is improved

Analysis of Bubble Sort (cont.)

- The best case occurs when the array is sorted already
  - one pass is required (O(*n*) comparisons)
  - no exchanges are required (O(1) exchanges)
- Bubble sort works best on arrays nearly sorted and worst on *inverted* arrays (elements are in reverse sorted order)

```java
public class BubbleSort implements SortAlgorithm {

public <T extends Comparable<T>>  void sort(T[] table) {
        int pass = 1;
        boolean exchanges = false;
        do {
            // Invariant: Elements after table.length - pass + 1
            // are in place.
            exchanges = false; // No exchanges yet.
            // Compare each pair of adjacent elements.
            for (int i = 0; i < table.length - pass; i++) {
                if (table[i].compareTo(table[i + 1]) > 0) {
                    // Exchange pair.
                    T temp = table[i];
                    table[i] = table[i + 1];
                    table[i + 1] = temp;
                    exchanges = true; // Set flag.
                }
            }
            pass++;
        } while (exchanges);

        // assert: Array is sorted.
    }
}
```

Insertion Sort

Section 8.4

Insertion Sort

- Another quadratic sort, *insertion* sort, is based on the technique used by card players to arrange a hand of cards
  - 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

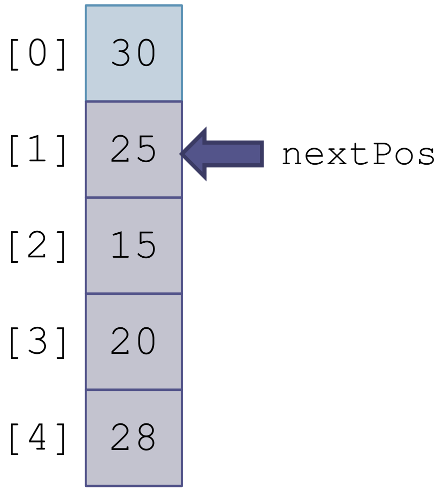
[0]	30
[1]	25
[2]	15
[3]	20
[4]	28

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

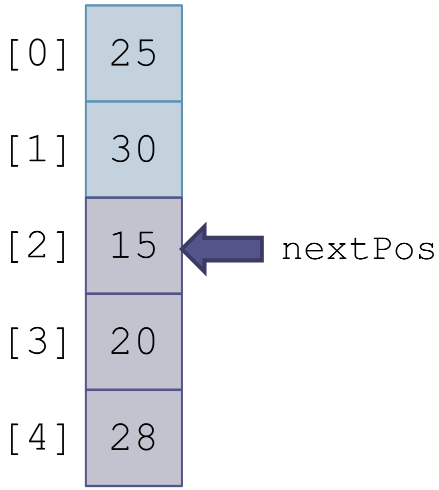
[0]	25
[1]	30 ← nextPos
[2]	15
[3]	20
[4]	28

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

Trace of Insertion Sort (cont.)

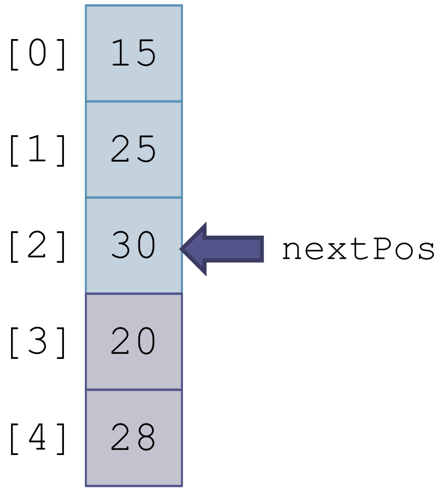
nextPos	2

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

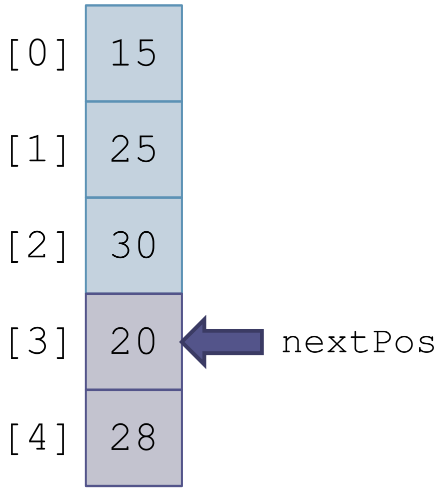


`1.` **`for`** each array element from the second (**`nextPos`** = 1) to the last

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

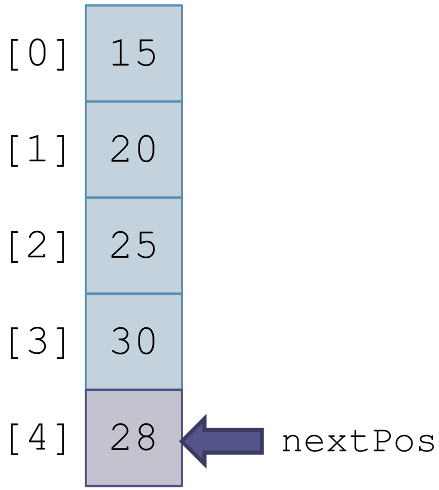
[0]	15	
[1]	20	
[2]	25	
[3]	30	⇐ nextPos
[4]	28	

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

Trace of Insertion Sort (cont.)

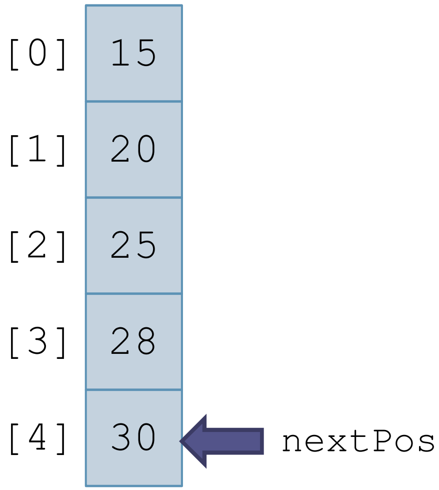
nextPos	4



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

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

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.)

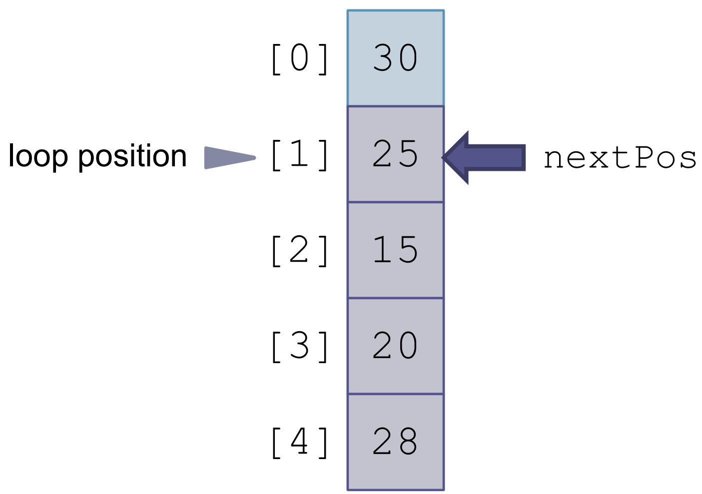
`nextPos`	1
`nextVal`	

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

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.)

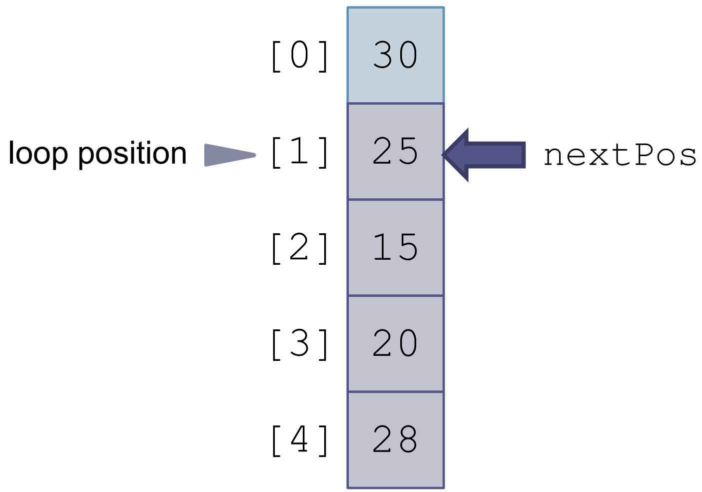
`nextPos`	1
`nextVal`	



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.)

`nextPos`	1
`nextVal`	25



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.)

`nextPos`	1
`nextVal`	25

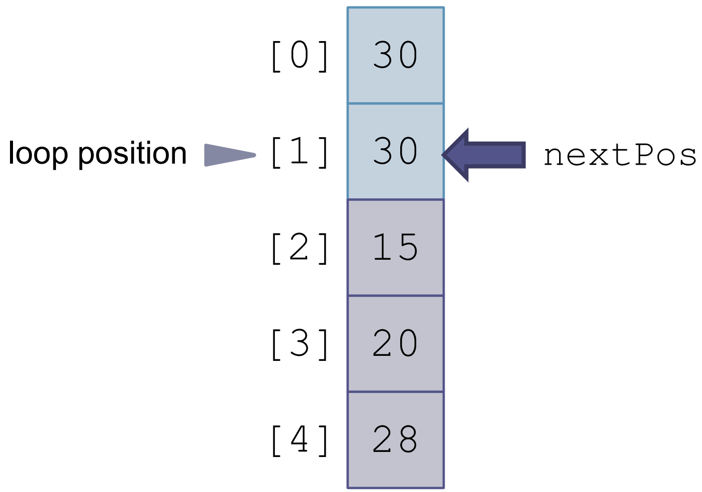
[0]	30	
[1]	25	nextPos
[2]	15	
[3]	20	
[4]	28	

loop position → [1]

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.)

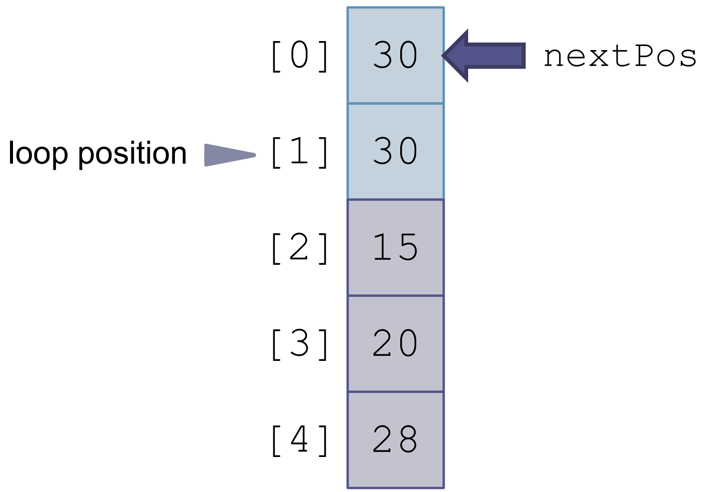
`nextPos`	1
`nextVal`	25



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.)

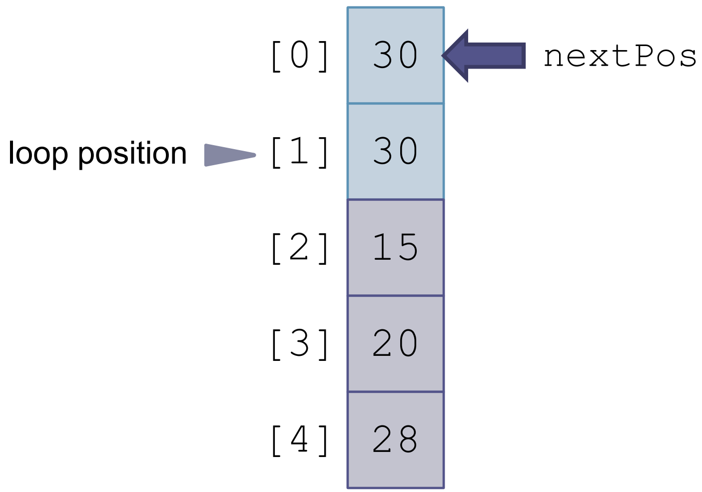
`nextPos`	0
`nextVal`	25



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.)

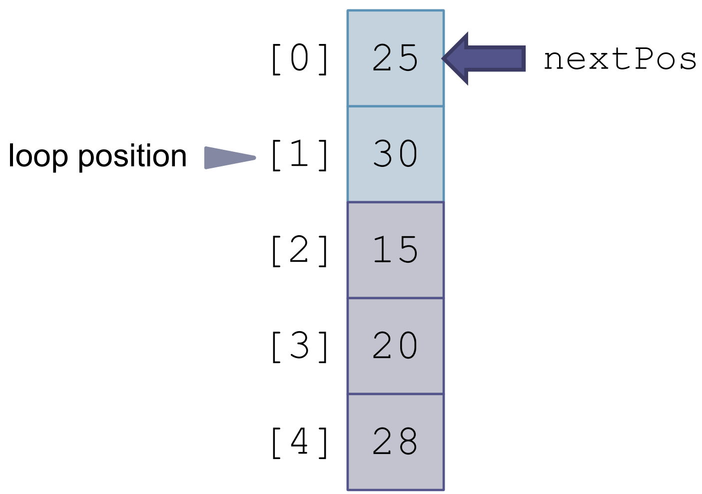
`nextPos`	0
`nextVal`	25



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.)

`nextPos`	0
`nextVal`	25



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.)

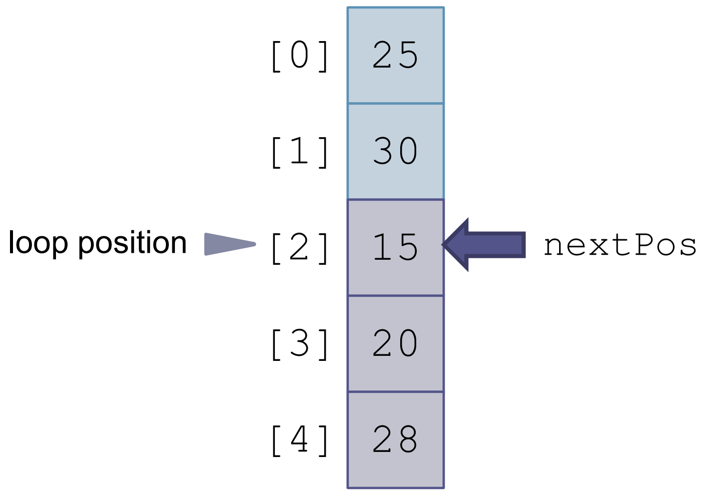
`nextPos`	0
`nextVal`	25

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

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.)

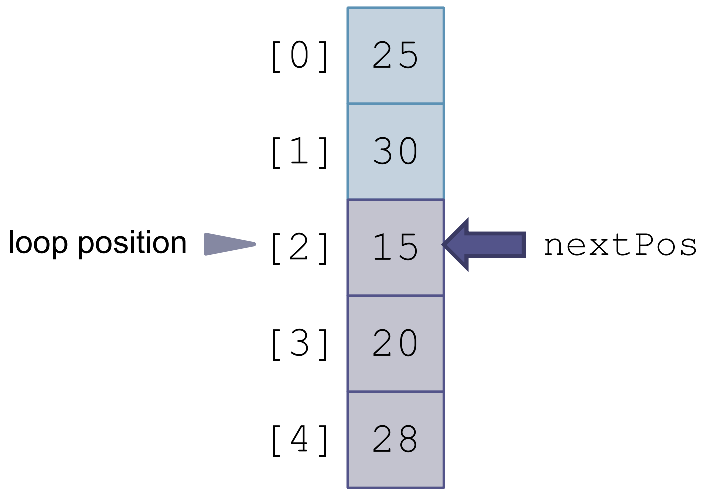
nextPos	2
nextVal	25



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.)

`nextPos`	2
`nextVal`	15



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.)

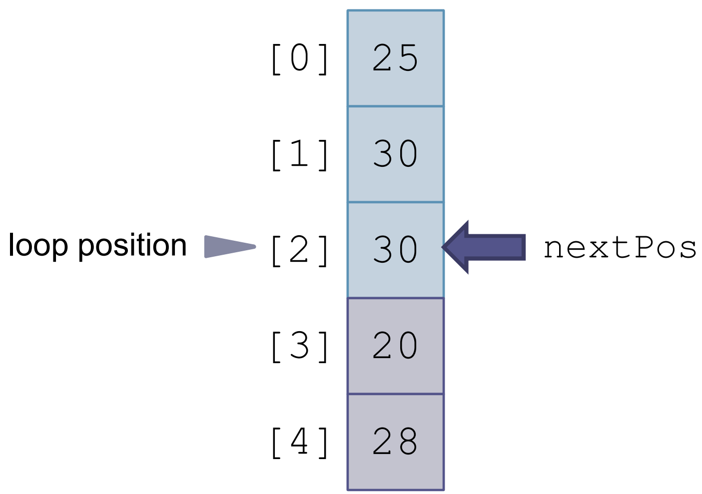
`nextPos`	2
`nextVal`	15

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

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.)

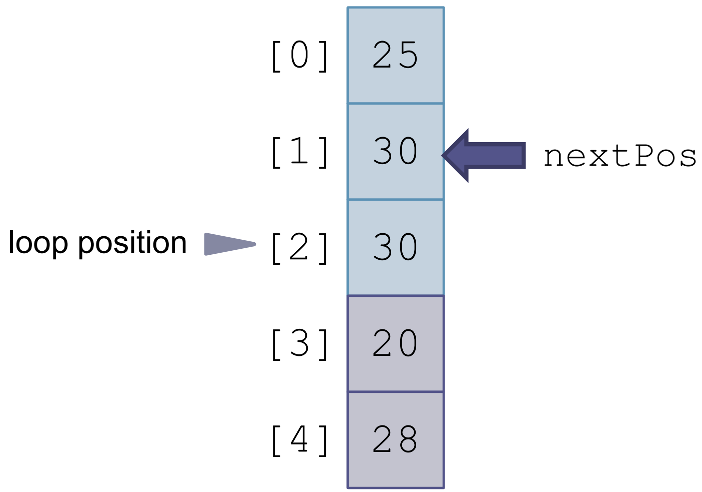
nextPos	2
nextVal	15



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.)

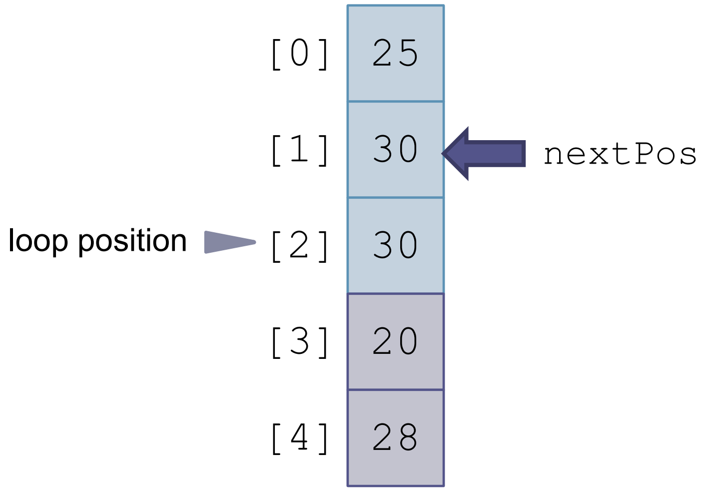
`nextPos`	1
`nextVal`	15



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.)

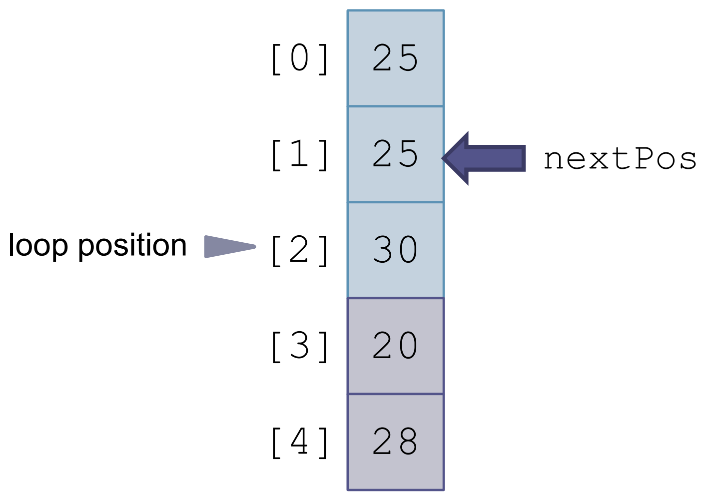
`nextPos`	1
`nextVal`	15



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.)

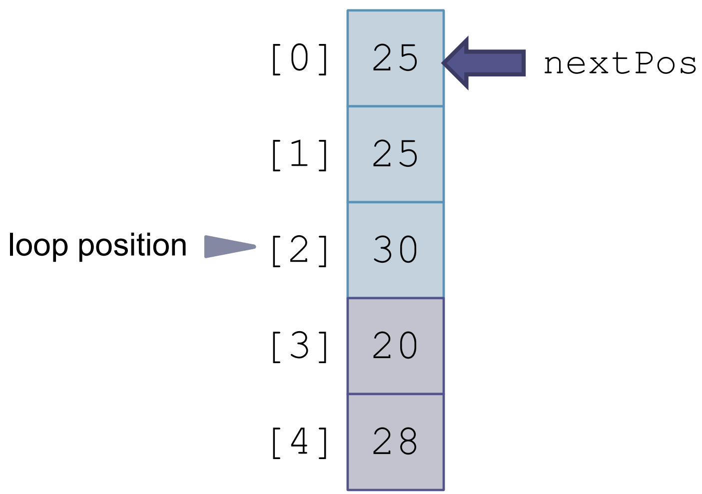
`nextPos`	1
`nextVal`	15



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.)

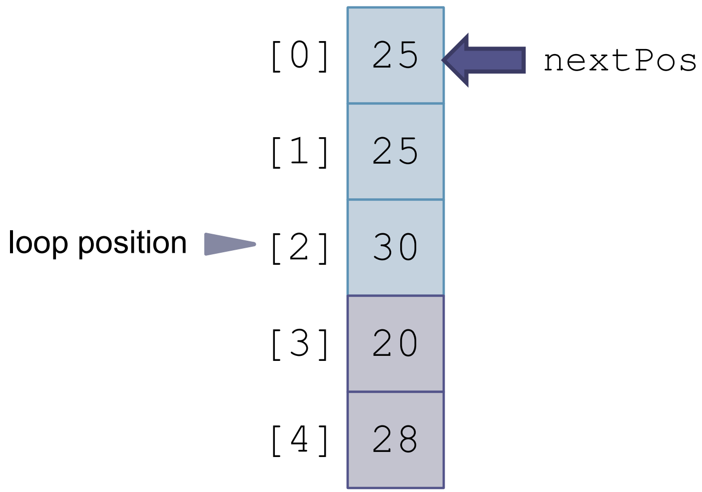
nextPos	0
nextVal	15



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.)

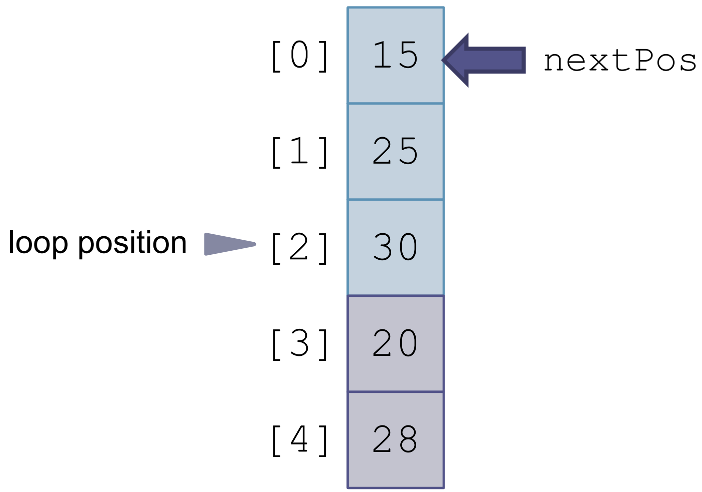
`nextPos`	0
`nextVal`	15



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.)

`nextPos`	0
`nextVal`	15



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.)

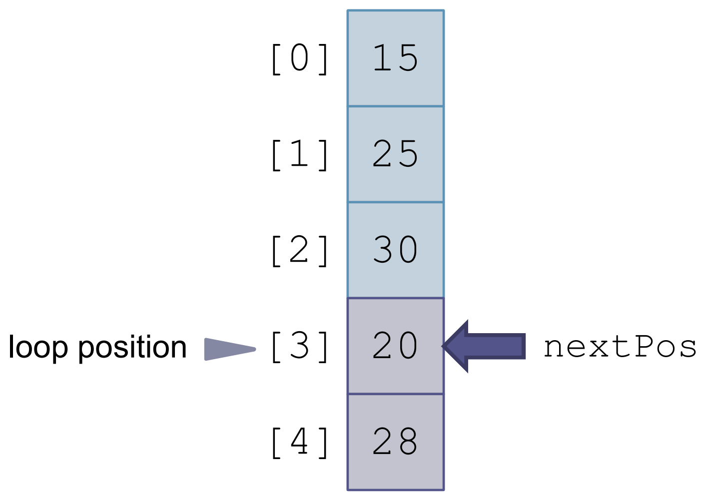
`nextPos`	0
`nextVal`	15

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

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.)

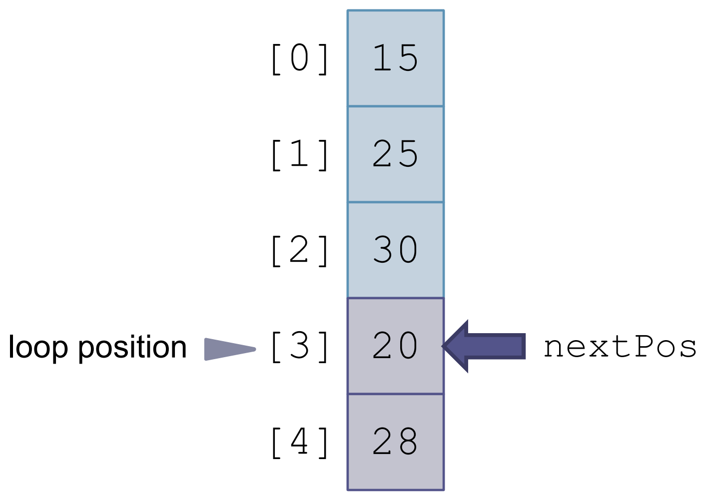
`nextPos`	3
`nextVal`	15



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.)

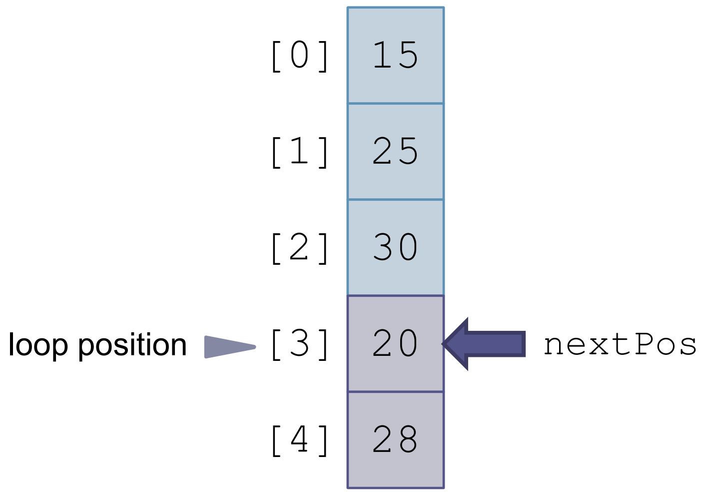
`nextPos`	3
`nextVal`	20



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.)

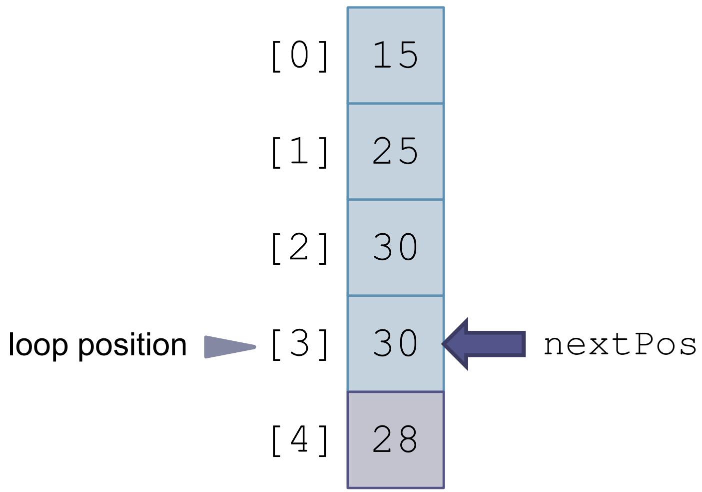
`nextPos`	3
`nextVal`	20



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.)

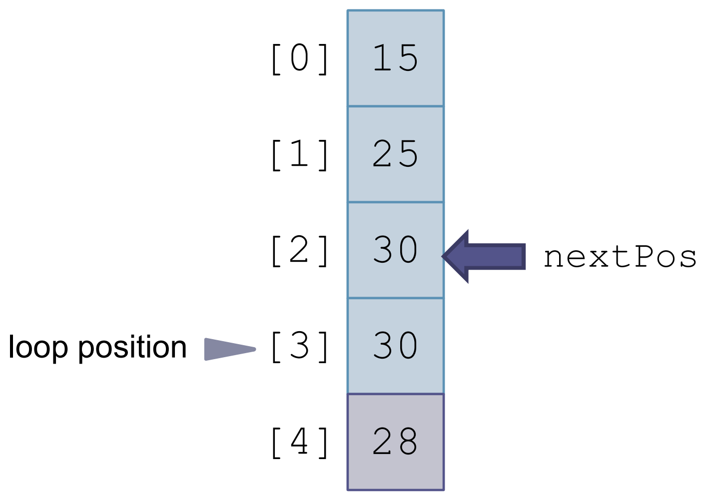
`nextPos`	3
`nextVal`	20



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.)

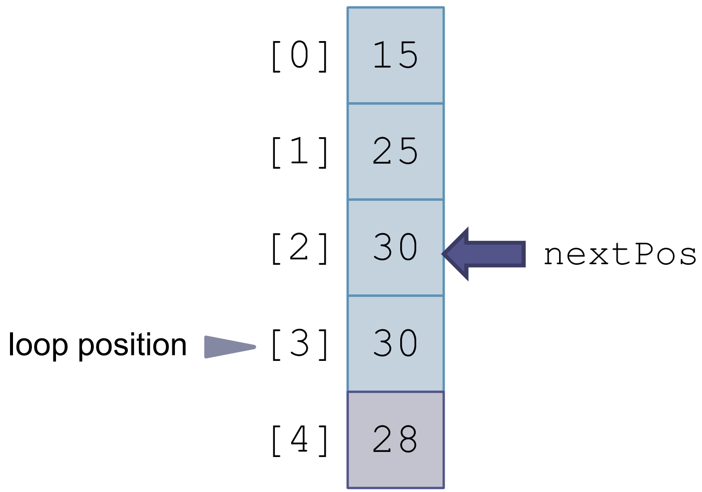
nextPos	2
nextVal	20



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.)

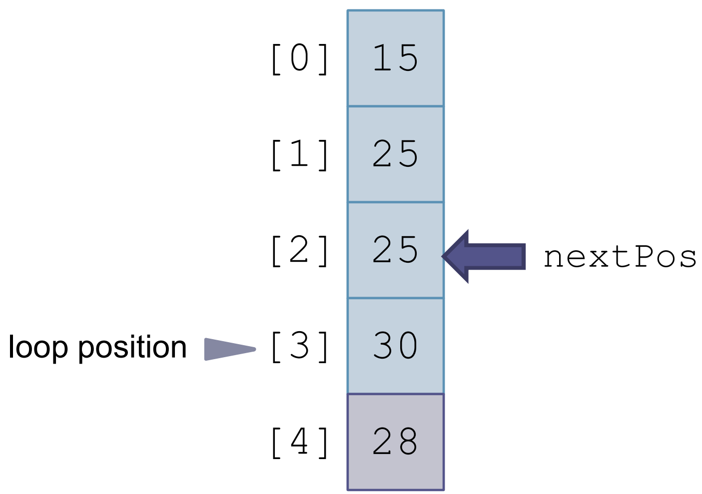
`nextPos`	2
`nextVal`	20



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.)

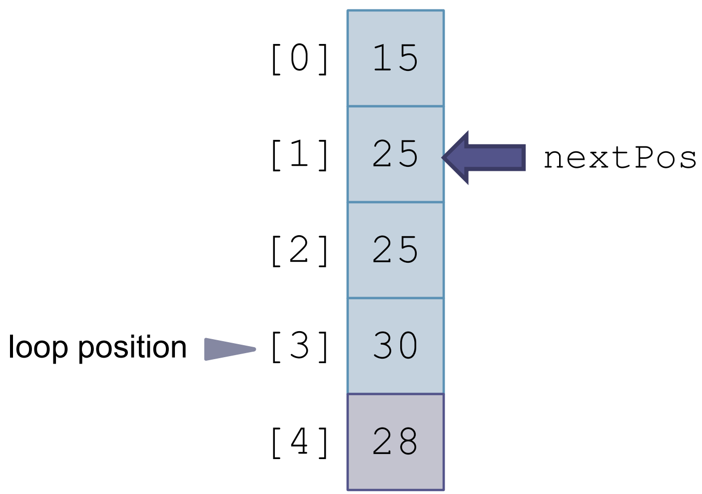
nextPos	2
nextVal	20



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.)

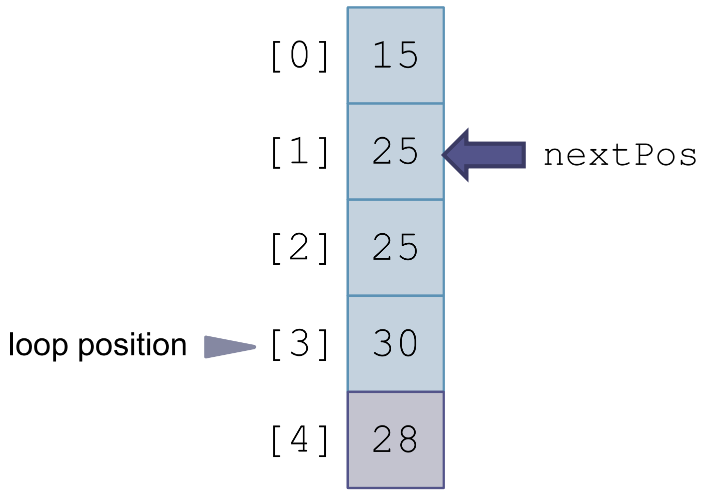
`nextPos`	1
`nextVal`	20



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.)

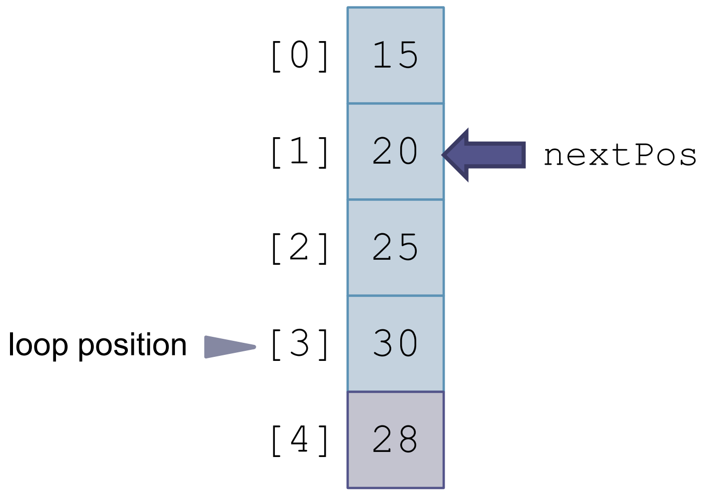
`nextPos`	1
`nextVal`	20



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.)

`nextPos`	1
`nextVal`	20



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.)

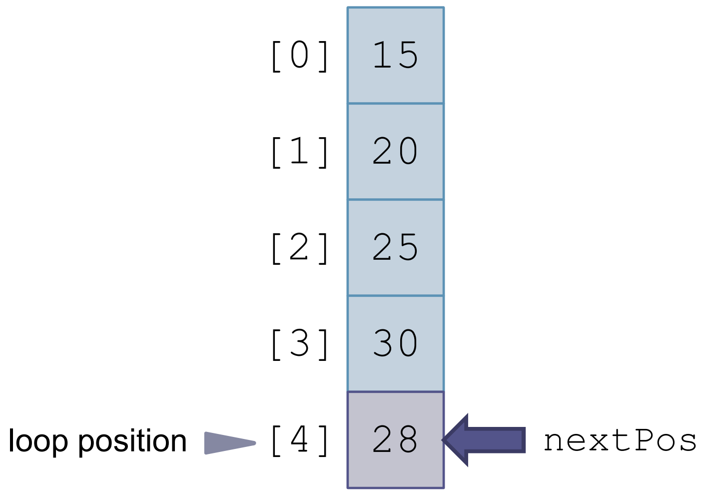
nextPos	1
nextVal	20

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

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.)

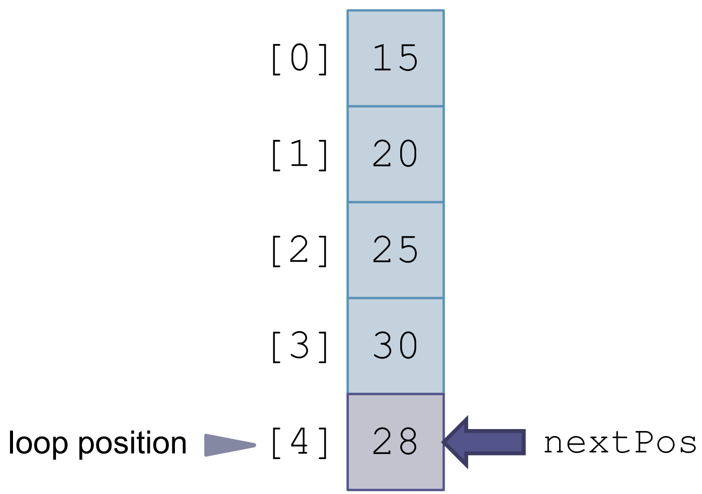
`nextPos`	4
`nextVal`	20



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.)

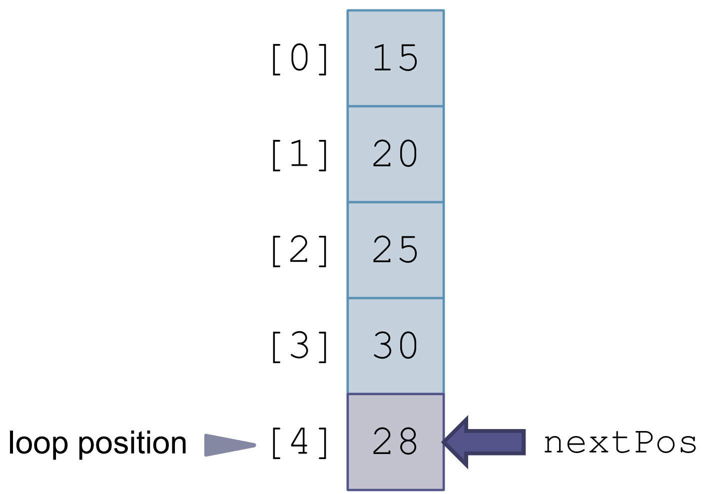
nextPos	4
nextVal	28



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.)

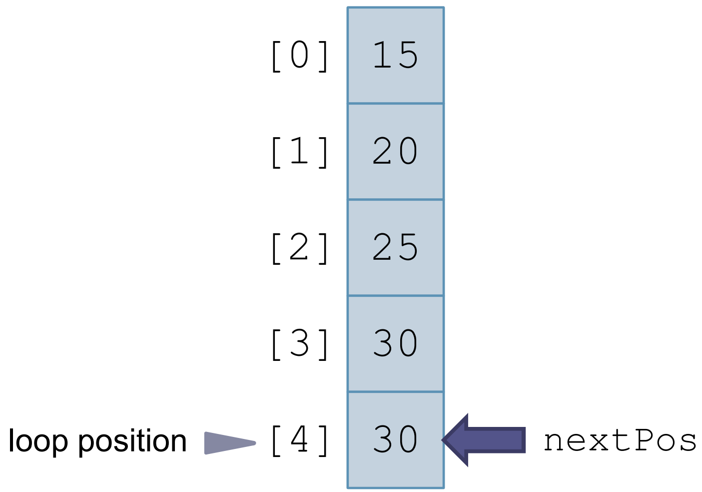
`nextPos`	4
`nextVal`	28



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.)

nextPos	4
nextVal	28



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.)

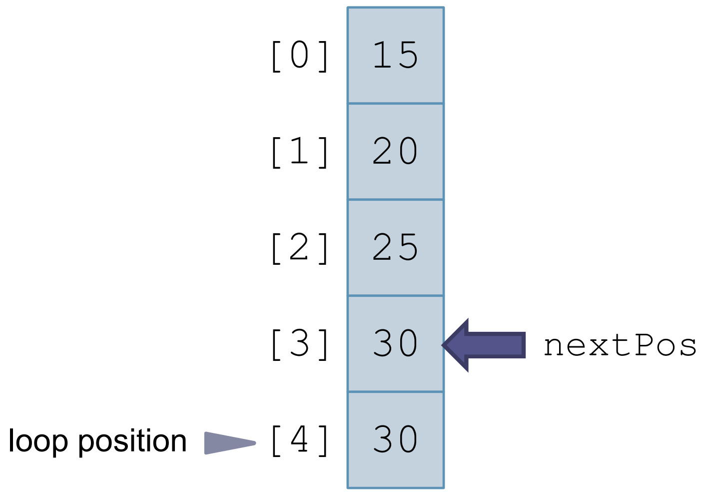
`nextPos`	3
`nextVal`	28

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

1. **`for`** each array element from the second (`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.)

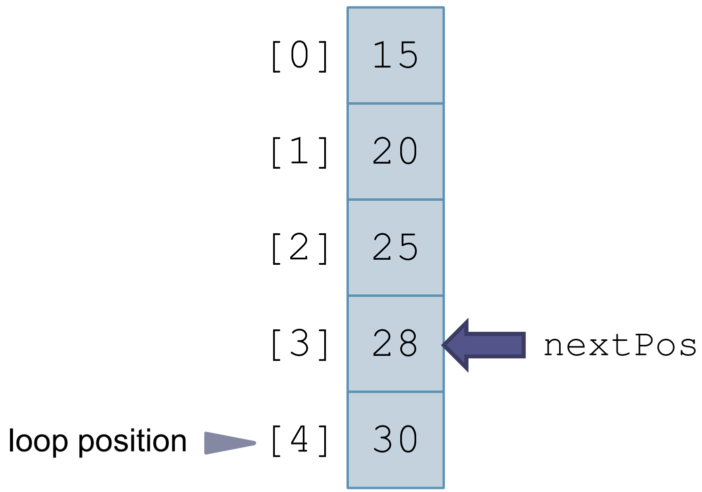
nextPos	3
nextVal	28



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.)

nextPos	3
nextVal	28



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.)

`nextPos`	3
`nextVal`	28

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

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`

Analysis of Insertion Sort

- The insertion step is performed $n - 1$ times
- 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:

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

- which is $O(n^2)$

Analysis of Insertion Sort (cont.)

- 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*)
- The number of shifts performed during an insertion is one less than the number of comparisons
- Or, when the new value is the smallest so far, it is the same as the number of comparisons
- A shift in an insertion sort requires movement of only 1 item, while an exchange in a bubble or selection sort involves a temporary item and the movement of three items
  - The item moved may be a primitive or an object reference
  - The objects themselves do not change their locations

```
public class InsertionSort implements SortAlgorithm {

    public <T extends Comparable<T>>  void sort(T[] table) {
        for (int nextPos = 1; nextPos < table.length; nextPos++) {
            // Invariant: table[0 . . . nextPos - 1] is sorted.
            // Insert element at position nextPos
            // in the sorted subarray.
            insert(table, nextPos);
        } // End for.
    } // End sort.

    /* Insert the element at nextPos where it belongs
     * in the array.
     * @pre  table[0 . . . nextPos - 1] is sorted.
     * @post table[0 . . . nextPos] is sorted.
     * @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]; // Shift down.
            nextPos--; // Check next smaller element.
        }
        // Insert nextVal at nextPos.
        table[nextPos] = nextVal;
    }
}
```

Comparison of Quadratic Sorts

Section 8.5

Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

Comparison of Quadratic Sorts (cont.)

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.)

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts
- Bubble sort generally gives the worst performance—unless the array is nearly sorted
  - Big-O analysis ignores constants and overhead
- None of the quadratic search algorithms are particularly good for large arrays (n > 1000)
- The best sorting algorithms provide $n \log n$ average case performance

Comparison of Quadratic Sorts (cont.)

- All quadratic sorts require storage for the array being sorted
- However, the array is sorted in place
- While there are also storage requirements for variables, for large *n*, the size of the array dominates and extra space usage is O(1)

Comparisons versus 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` method
- The cost of a comparison depends on its complexity, but is generally more costly than an exchange
- For some other languages, an exchange may involve physically moving information rather than swapping object references. In these cases, an exchange may be more costly than a comparison

Shell Sort: A Better Insertion Sort

Section 8.6

Shell Sort: A Better Insertion Sort

- A Shell sort is a type of insertion sort, but with O($n^{3/2}$) or better performance than the O(n^2) sorts
- It is named after its discoverer, Donald Shell
- A Shell sort can be thought of as 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

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

[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

[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 2

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 3

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

[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 5

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

[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 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

subarray 4

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

[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 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

[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 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	34	45	62	57	55	90	85	60	90	70	75	65

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

[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

[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 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	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

subarray 2

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	80	75	35	45	62	57	55	90	85	60	90	70	75	65

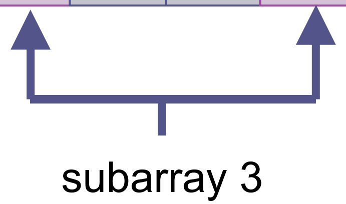
subarray 2

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	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

subarray 3

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1

[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

subarray 1

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1

[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 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

[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 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	55	75	57	80	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	55	75	57	80	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	60	90	85	80	90	70	75	65

subarray 3

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1

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

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	55	75	57	60	90	85	80	90	70	75	65

subarray 2

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	55	75	57	60	90	70	80	90	85	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	55	75	57	60	90	70	80	90	85	75	65

subarray 3

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 3

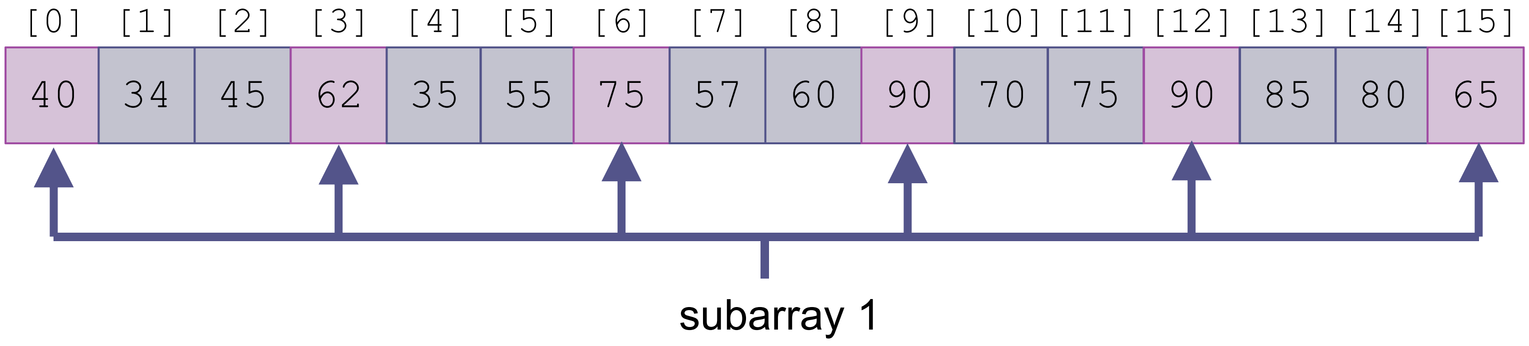
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	55	75	57	60	90	70	75	90	85	80	65

subarray 3

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1



Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1

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

subarray 1

Trace of Shell Sort (cont.)

gap value	3

Sort subarray 1

[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

subarray 1

Trace of Shell Sort (cont.)

gap value	3

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)

[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)

[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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	40	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	62	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	57	62	65	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)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	57	62	65	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)

[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	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)

[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	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)

[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	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)

[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	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)

[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	90	85	80	90

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	90	85	80	90

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	90	85	80	90

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	90	85	80	90

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	90	85	80	90

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	90	85	80	90

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	90	80	90

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	90	80	90

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)

[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

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***

Shell Sort Algorithm (cont.)

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

- 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 are used for `gap`, performance is O(n^2)
- If successive values for `gap` are based on *Hibbard's sequence*,

  $2^k - 1$ (i.e. 31, 15, 7, 3, 1)

  it can be proven that the performance is O($n^{3/2}$)
- Other sequences give similar or better performance

Analysis of Shell Sort (cont.)

- Our algorithm selects the initial value of `gap` as $\frac{n}{2}$ and then divides by 2.2 and truncates the result
- Empirical studies of this approach show that the performance is $O(n^{5/4})$ or even $O(n^{7/6})$, but there is no theoretical basis for this result

```
public class ShellSort implements SortAlgorithm {

   public <T extends Comparable<T>>  void sort(T[] table) {
        // Gap between adjacent elements.
        int gap = table.length / 2;
        while (gap > 0) {
            for (int nextPos = gap; nextPos < table.length;
                    nextPos++) {
                // Insert element at nextPos in its subarray.
                insert(table, nextPos, gap);
            } // End for.

            // Reset gap for next pass.
            if (gap == 2) {
                gap = 1;
            } else {
                gap = (int) (gap / 2.2);
            }
        } // End while.
    } // End sort.

    . . . .

}
```

```
public class ShellSort implements SortAlgorithm {

. . . .

    /**
     * Inserts element at nextPos where it belongs in array.
     * @pre  Elements through nextPos - gap in subarray are sorted.
     * @post Elements through nextPos in subarray are sorted.
     * @param table The array being sorted
     * @param nextPos The position of element to insert
     * @param gap The gap between elements in the subarray
     */
    private static <T extends Comparable<T>>  void insert(T[] table,
            int nextPos,
            int gap) {
        T nextVal = table[nextPos]; // Element to insert.
        // Shift all values > nextVal in subarray down by gap.
        while ((nextPos > gap - 1) // First element not shifted.
                && (nextVal.compareTo(table[nextPos - gap]) < 0)) {
            table[nextPos] = table[nextPos - gap]; // Shift down.
            nextPos -= gap; // Check next position in subarray.
        }
        table[nextPos] = nextVal; // Insert nextVal.
    }
}
```

Merge Sort

Section 8.7

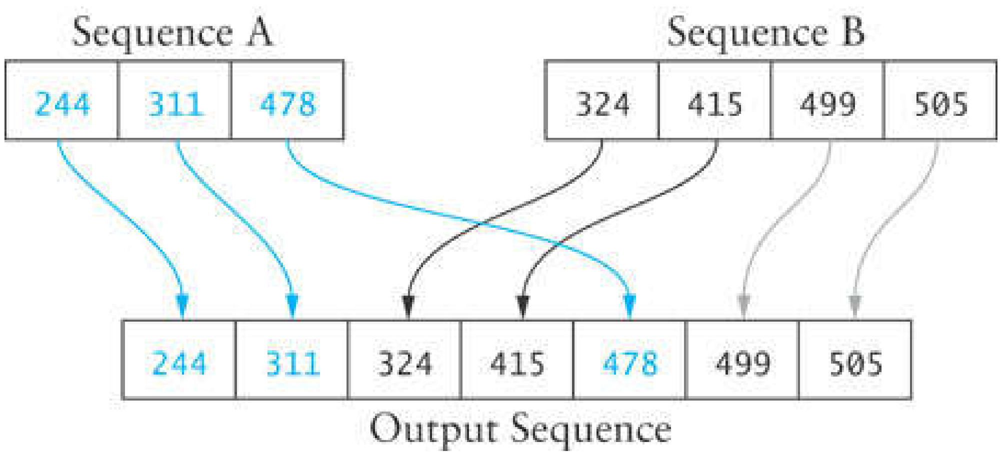
Merge

- A *merge* is a common data processing operation performed on two sequences of data with the following characteristics
  - Both sequences contain items with a common `compareTo` method
  - The objects in both sequences are ordered in accordance with this `compareTo` method
- 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 two 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*)

```java
public class MergeSort implements SortAlgorithm {

    private static <T extends Comparable<T>>  void merge(T[] outputSequence,
                                  T[] leftSequence, T[] rightSequence) {
        int i = 0; // Index into the left input sequence.
        int j = 0; // Index into the right input sequence.
        int k = 0; // 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++];
            }
        }
        // assert: one of the sequences has more items to copy.
        // 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++];
        }
    }
}
```

Merge Sort

- We can modify merging to sort a single, unsorted array
  1. Split the array into two halves
  2. Sort the left half
  3. Sort the right half
  4. Merge the two

- This algorithm can be written with a recursive step

(recursive) Algorithm for Merge Sort

Algorithm for Merge Sort

1. `if` the `tableSize` is > 1
2. Set `halfSize` to `tableSize` divided by 2.
3. Allocate a table called `leftTable` of size `halfSize`.
4. Allocate a table called `rightTable` of size `tableSize – halfSize`.
5. Copy the elements from `table[0 ... halfSize - 1]` into `leftTable`.
6. Copy the elements from `table[halfSize ... tableSize]` into `rightTable`.
7. Recursively apply the merge sort algorithm to `leftTable`.
8. Recursively apply the merge sort algorithm to `rightTable`.
9. Apply the merge method using `leftTable` and `rightTable` as the input and the original table as the output.

Trace of Merge Sort

50	60	45	30	90	20	80	15

50	60	45	30

90	20	80	15

Trace of Merge Sort (cont.)

50	60	45	30	90	20	80	15

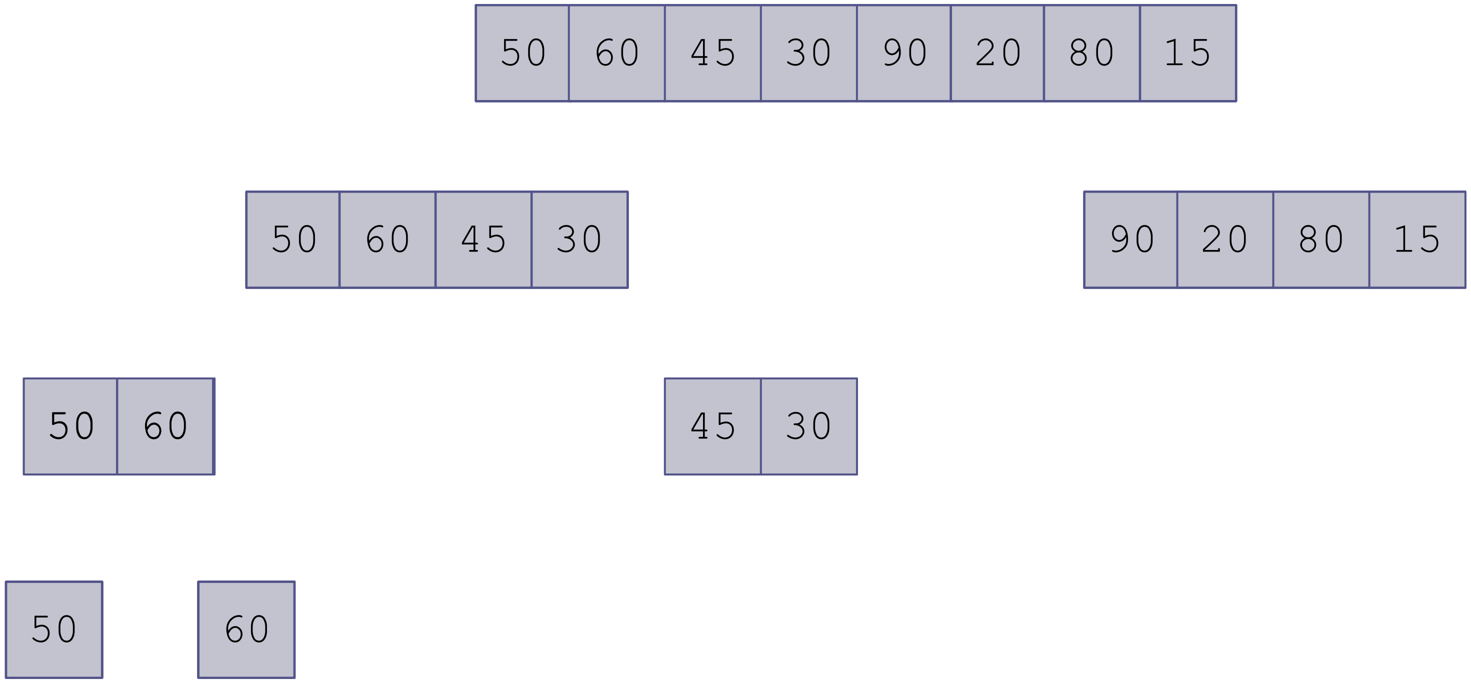
50	60	45	30

90	20	80	15

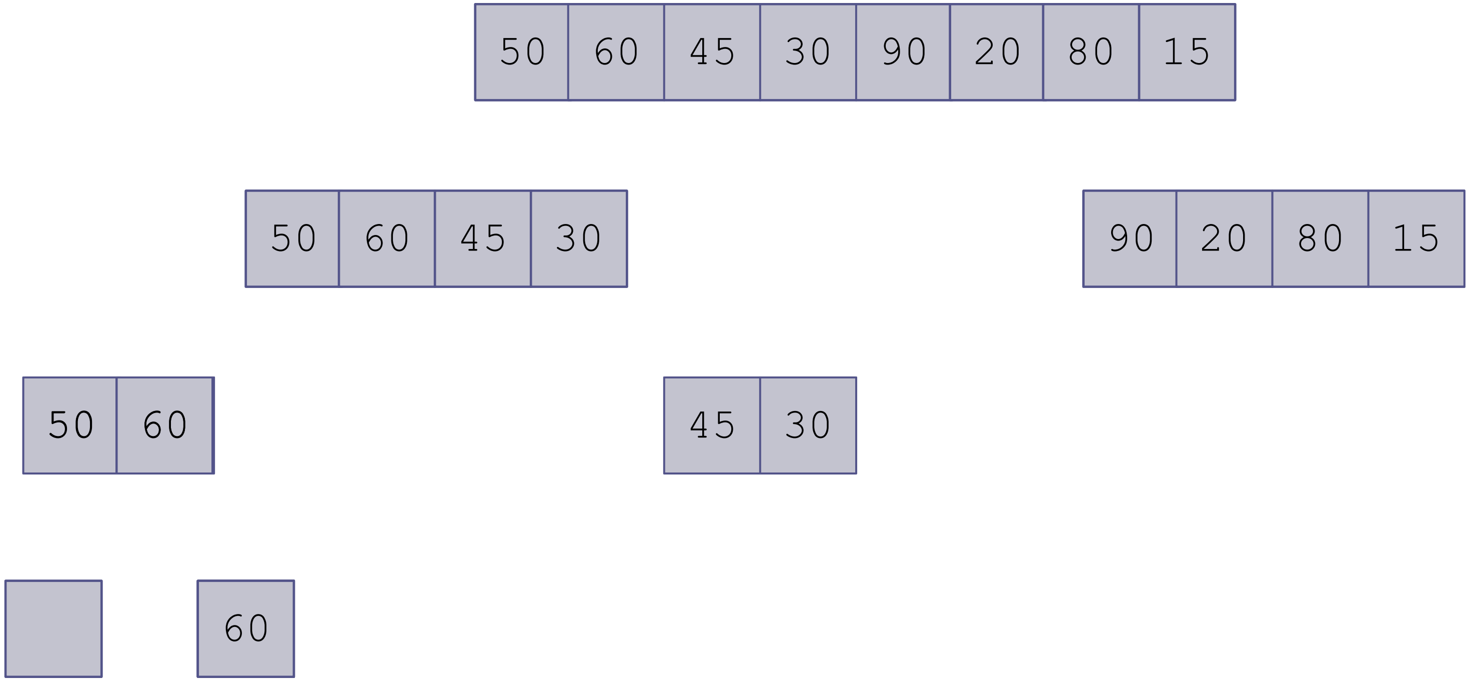
50	60

45	30

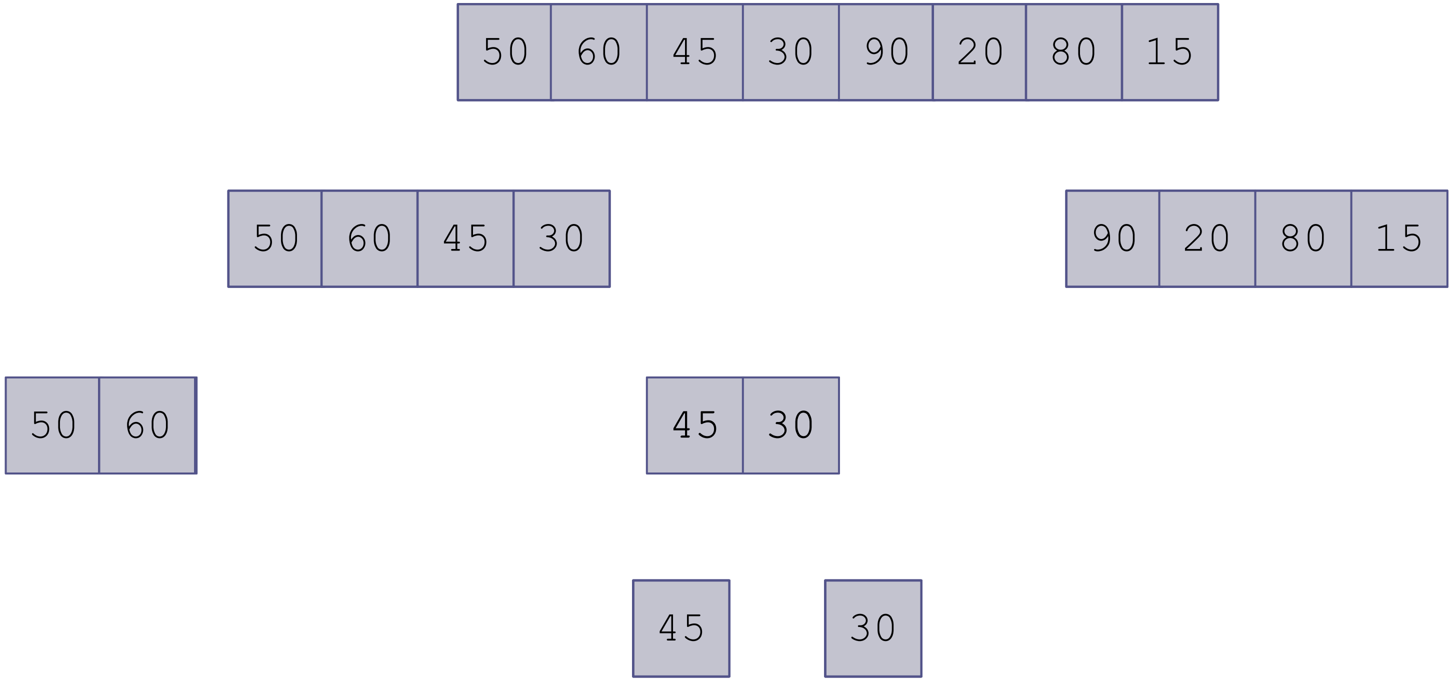
Trace of Merge Sort (cont.)



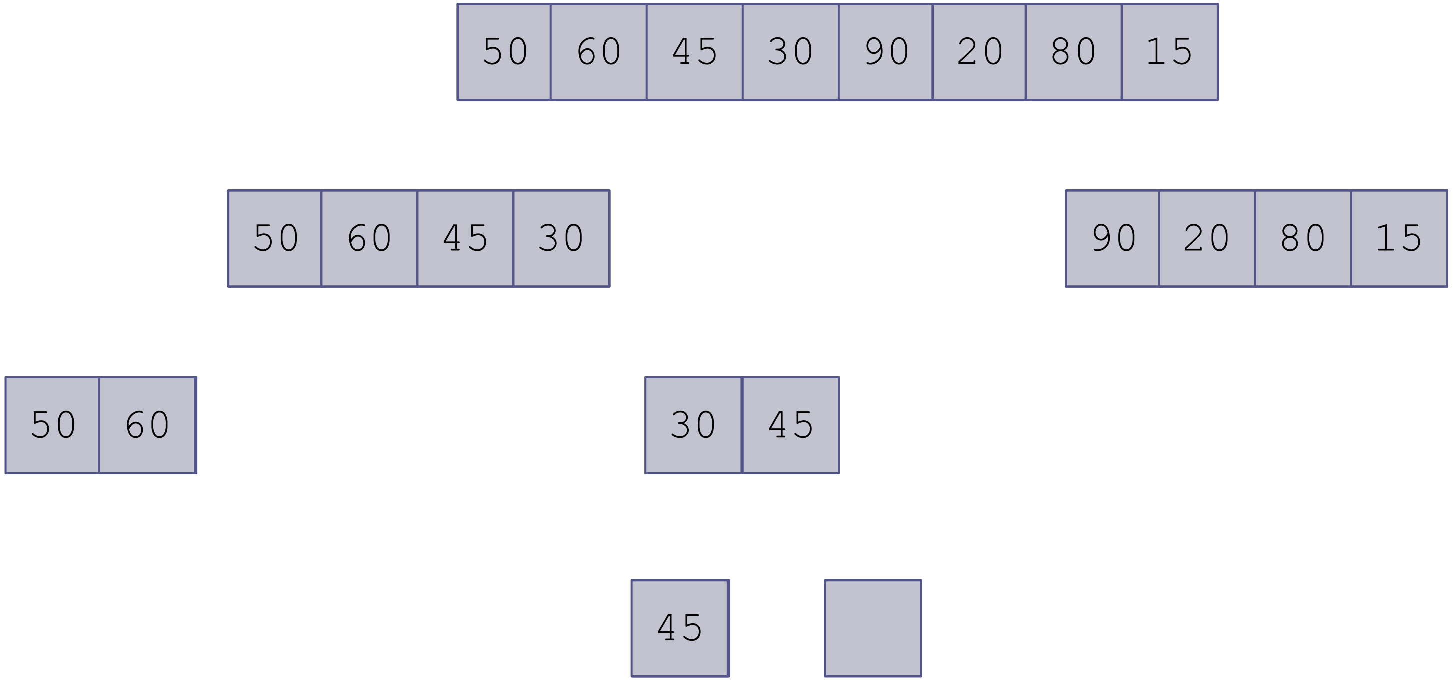
Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)

50	60	45	30	90	20	80	15

30	45	50	60

90	20	80	15

50	60

30	45

Trace of Merge Sort (cont.)

50	60	45	30	90	20	80	15

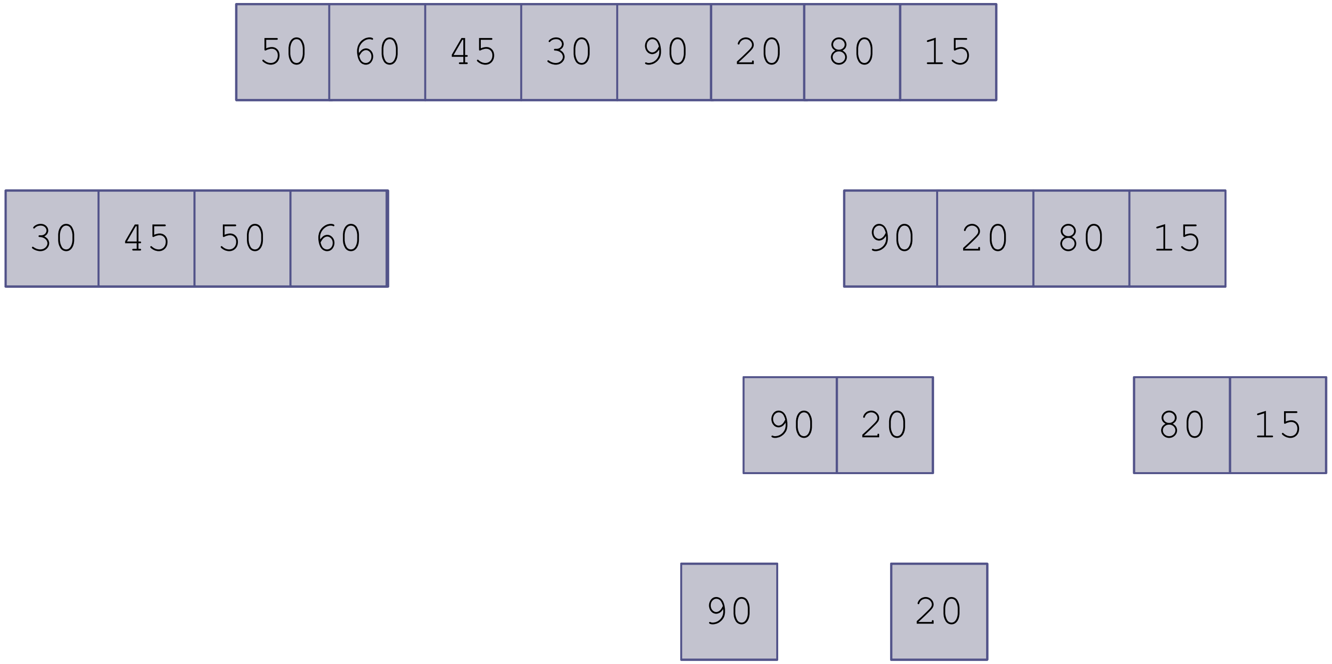
30	45	50	60

90	20	80	15

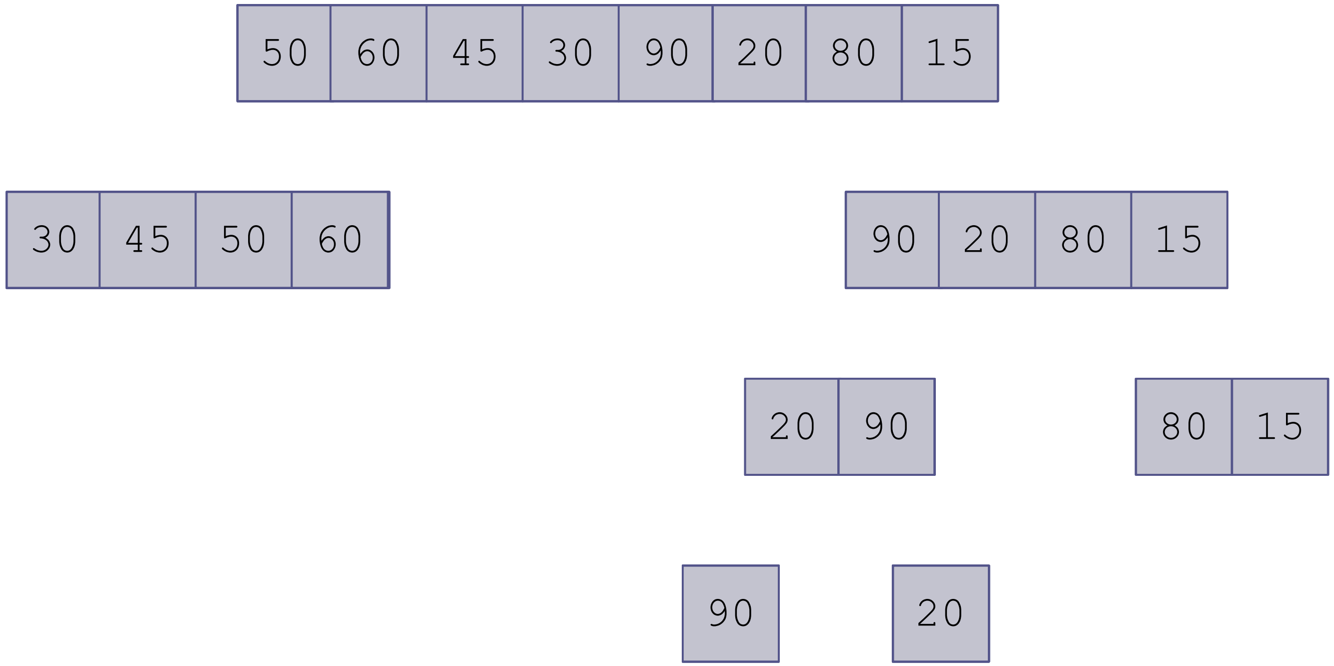
90	20

80	15

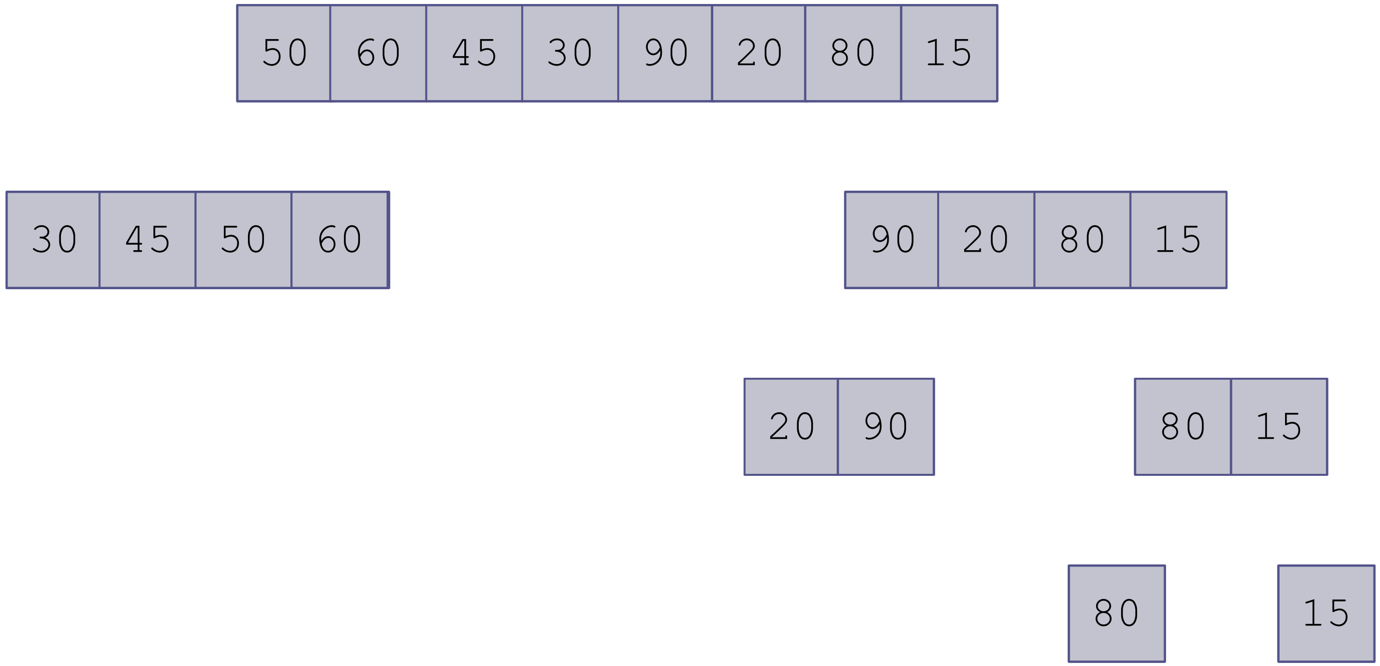
Trace of Merge Sort (cont.)



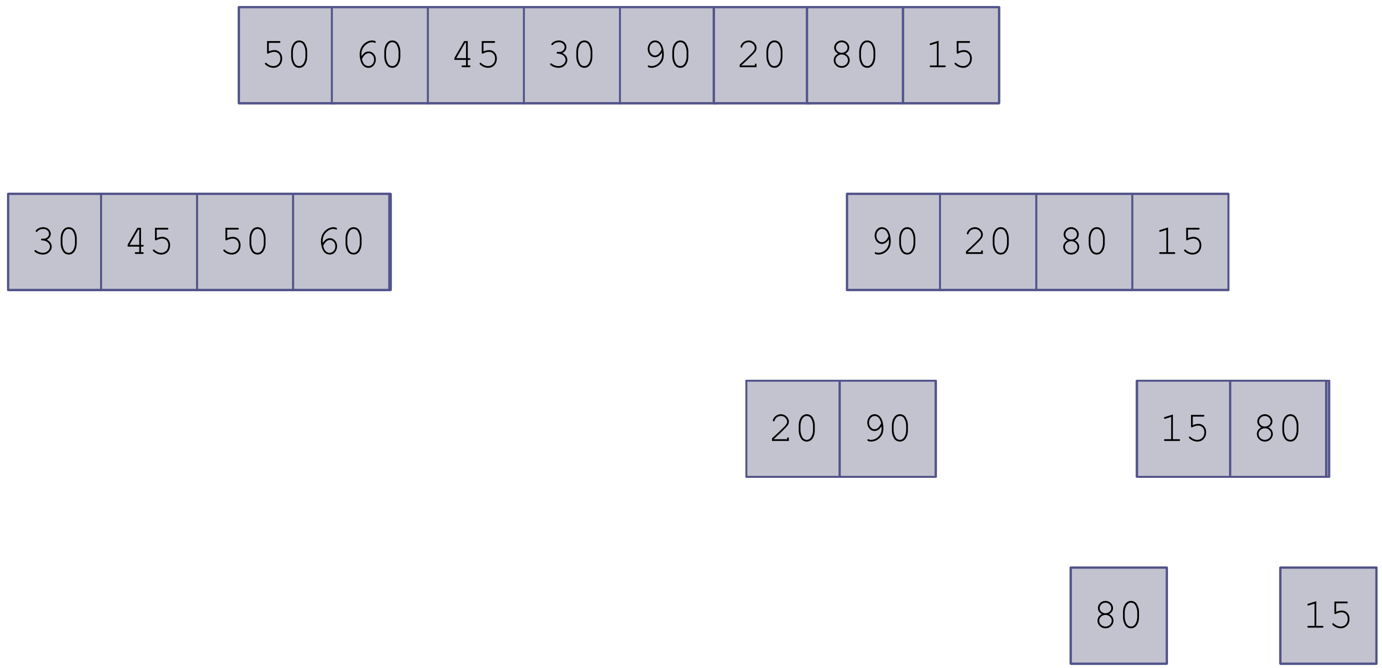
Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)



Trace of Merge Sort (cont.)

50	60	45	30	90	20	80	15

30	45	50	60

15	20	80	90

20	90

15	80

Trace of Merge Sort (cont.)

15	20	30	45	50	60	80	90

30	45	50	60

15	20	80	90

Analysis of Merge Sort

- Each backward step requires a movement of n elements from smaller-size arrays to larger arrays; the effort is O(n)
- The number of steps which require merging is log n because each recursive call splits the array in half
- The total effort to reconstruct the sorted array through merging is O(n log n)

Analysis of Merge Sort (cont.)

- Going down through the recursion chain, sorting the left tables, a sequence of right tables of size

$$\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k}$$

  is allocated

- Since

$$\frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1 = n - 1$$

  a total of *n* additional storage locations are required

```java
public class MergeSort implements SortAlgorithm {

    /**Sort the array using the merge sort algorithm.
     * @pre table contains Comparable objects.
     * @post table is sorted.
     * @param table The array to be sorted
     */
    @Override
    public <T extends Comparable<T>>  void sort(T[] table) {
        // A table with one element is sorted already.
        if (table.length > 1) {
            // Split table into halves.
            int halfSize = table.length / 2;
            T[] leftTable = (T[]) new Comparable[halfSize];
            T[] rightTable =
                    (T[]) new Comparable[table.length - halfSize];
            System.arraycopy(table, 0, leftTable, 0, halfSize);
            System.arraycopy(table, halfSize, rightTable, 0,
                    table.length - halfSize);

            // Sort the halves.
            sort(leftTable);
            sort(rightTable);

            // Merge the halves.
            merge(table, leftTable, rightTable);
        }
    }
}
```

Heapsort

Section 8.8

Heapsort

- Merge sort time is O(n log n) but still requires, temporarily, n extra storage locations
- *Heapsort* does not require any additional storage
- As its name implies, heapsort uses a heap to store the array

First Version of a Heapsort Algorithm

- When used as a priority queue, a heap maintains a 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

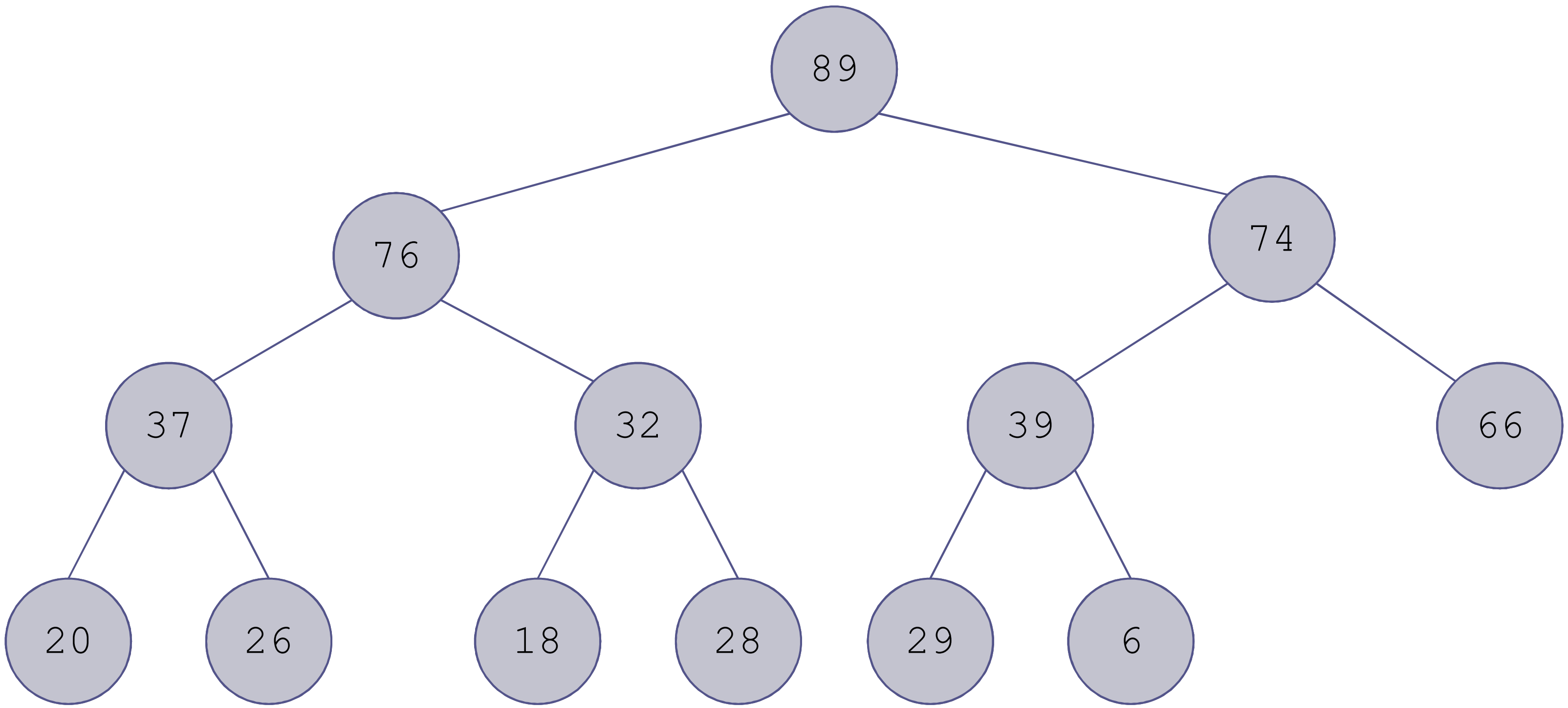
Heapsort Algorithm: First Version

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`**

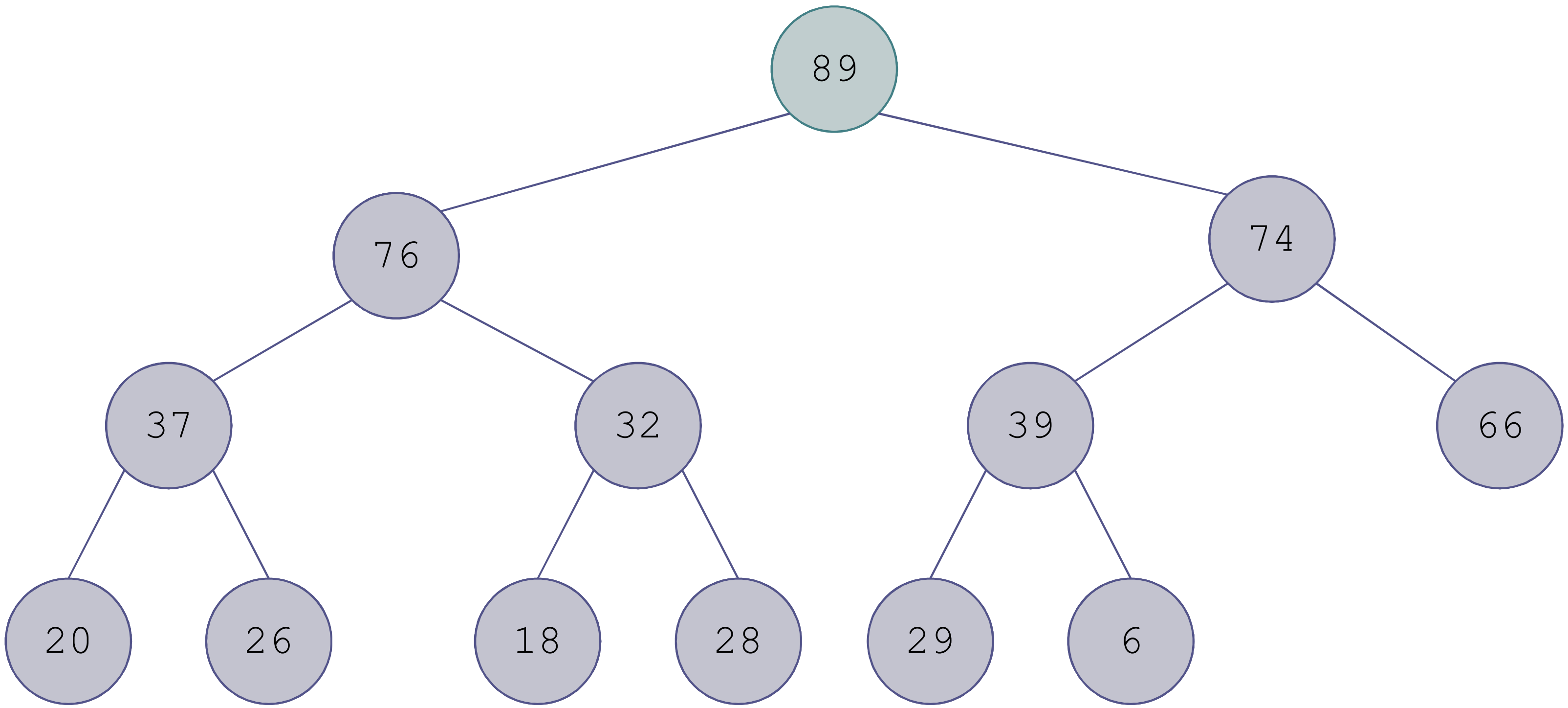
Revising the Heapsort Algorithm

- In heaps we've used so far, each parent node value was not greater than the values of its children
- We can build a heap so that each parent node value is *not less* than its children
- Then,
  - move the top item to the bottom of the heap
  - reheap, ignoring the item moved to the bottom

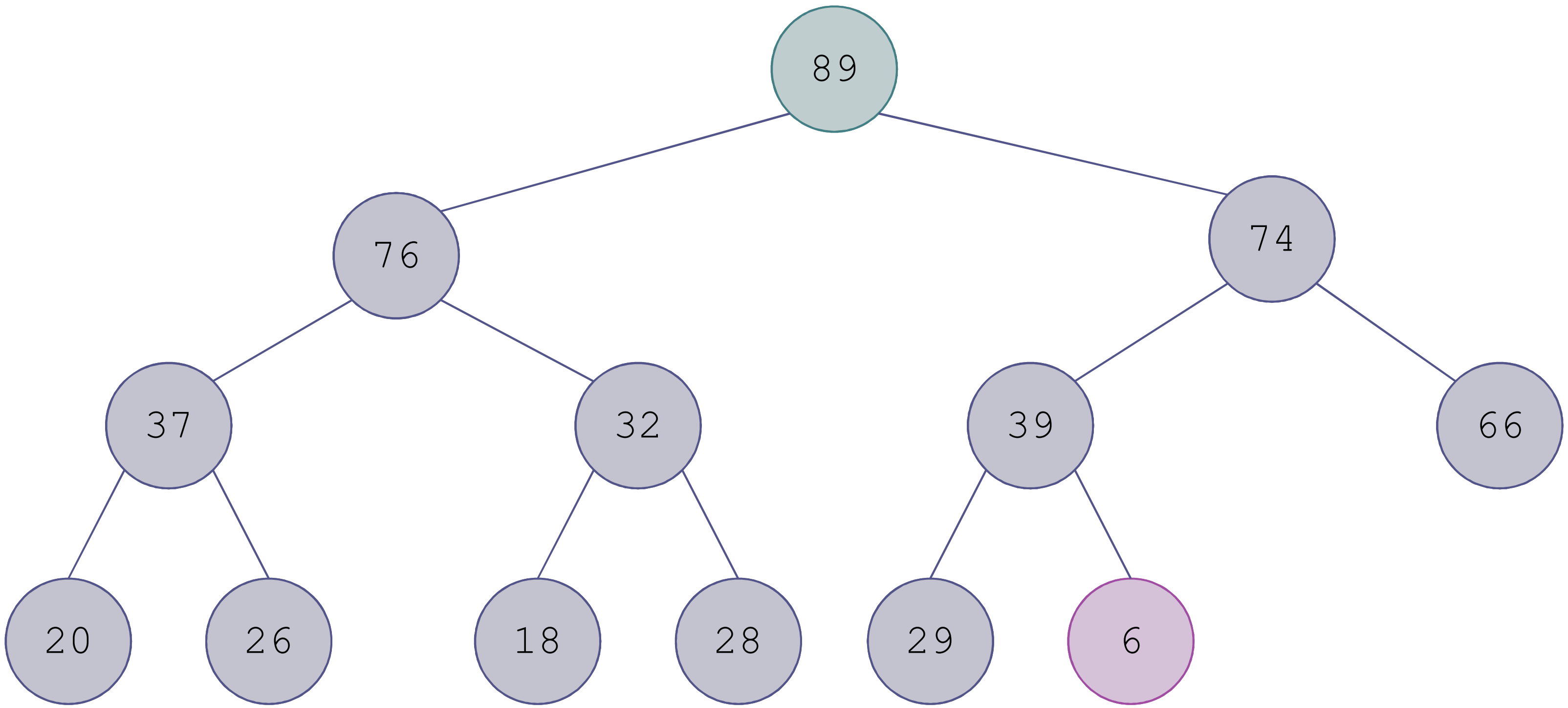
Trace of Heapsort



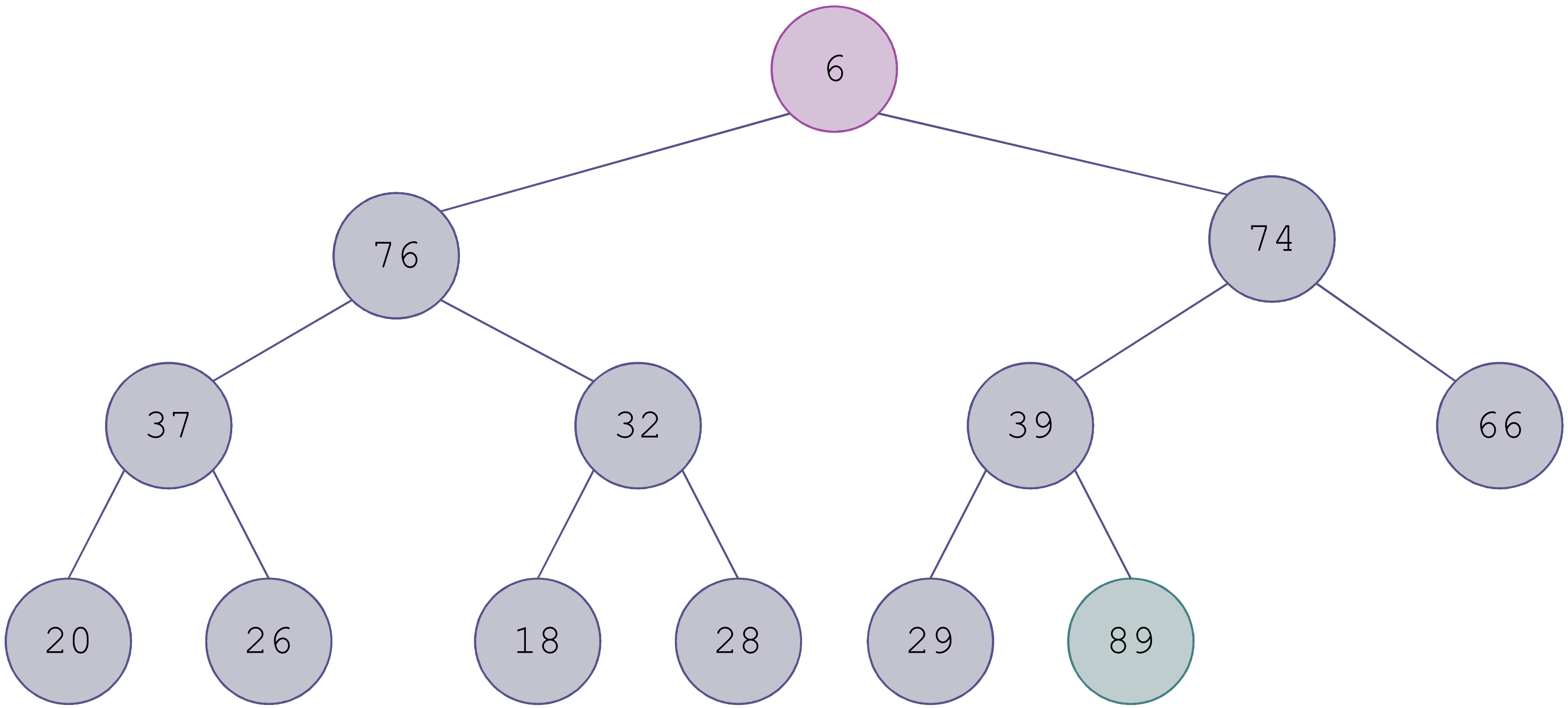
Trace of Heapsort (cont.)



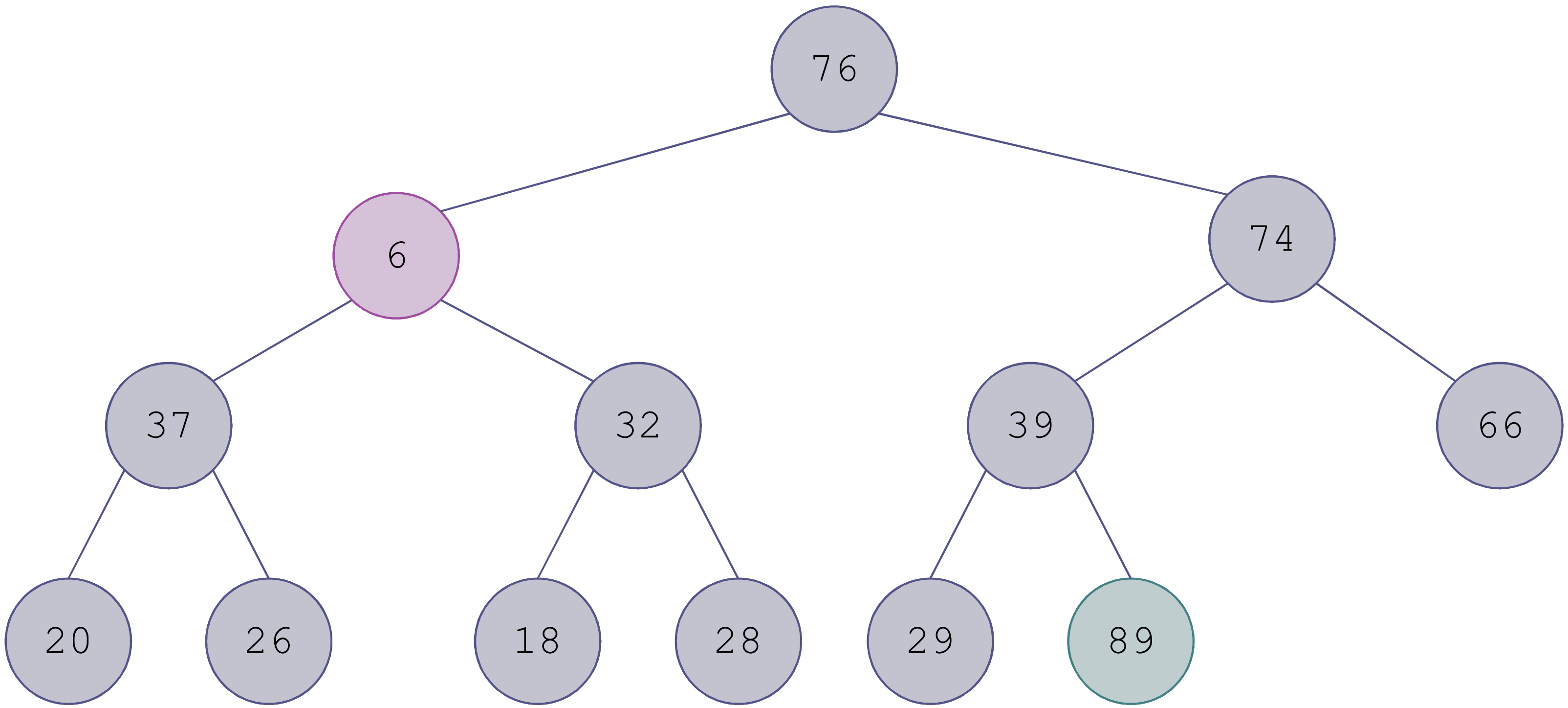
Trace of Heapsort (cont.)



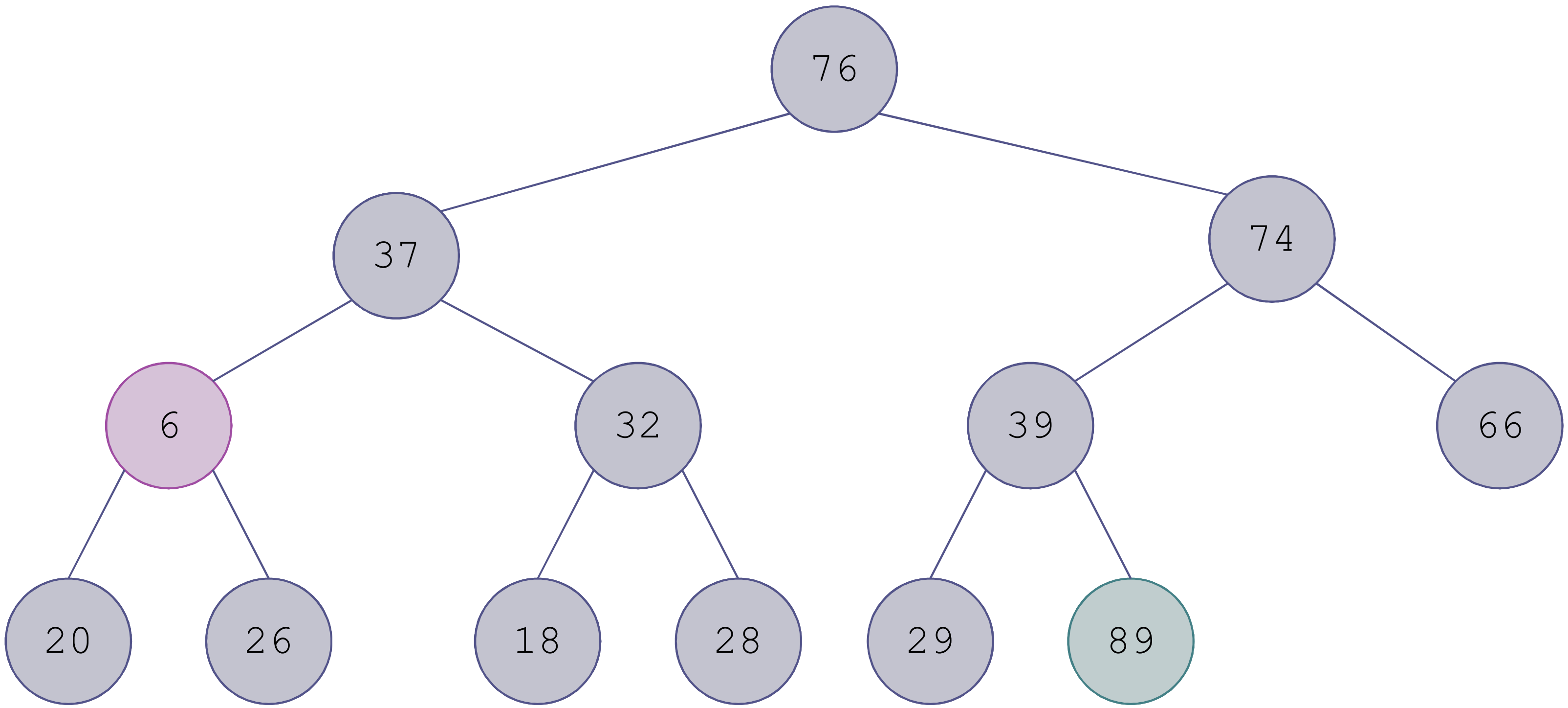
Trace of Heapsort (cont.)



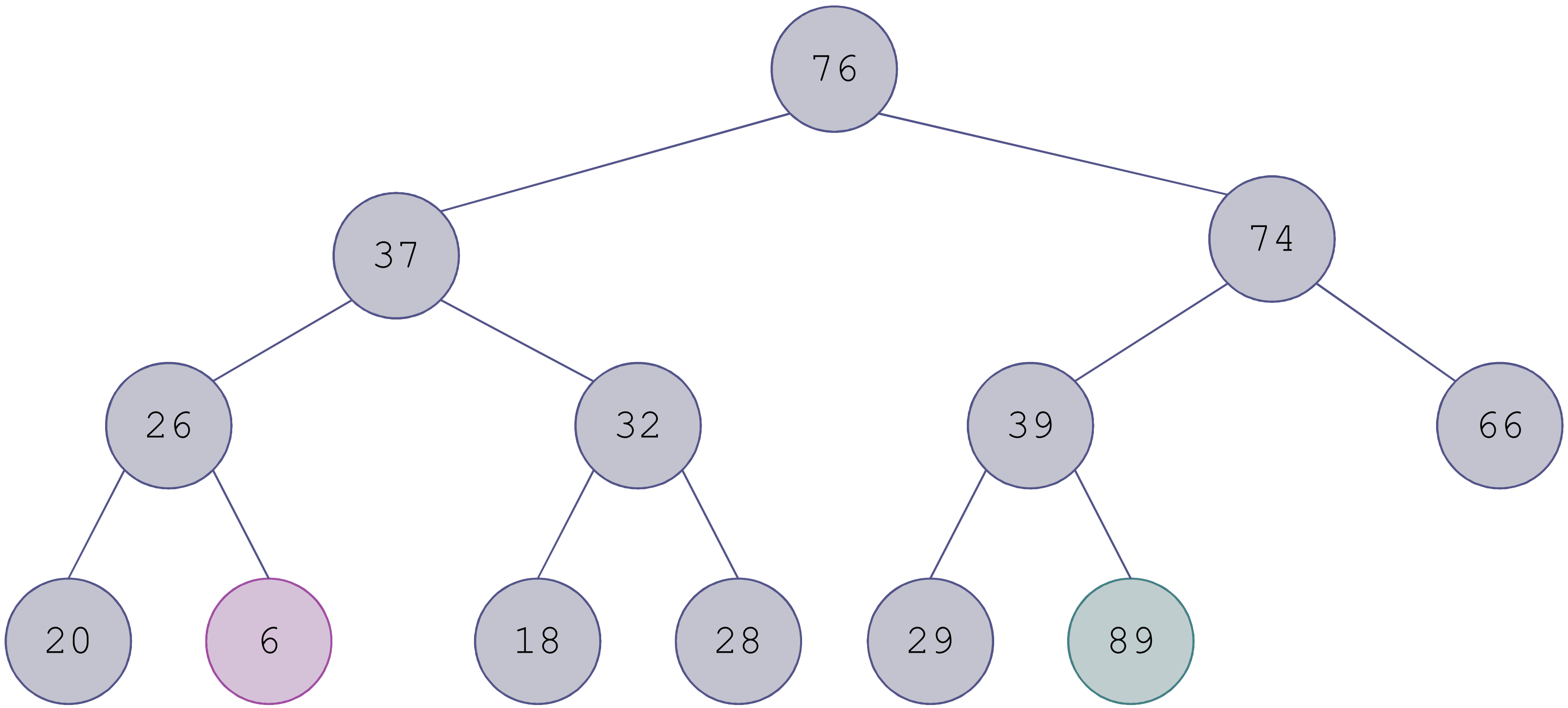
Trace of Heapsort (cont.)



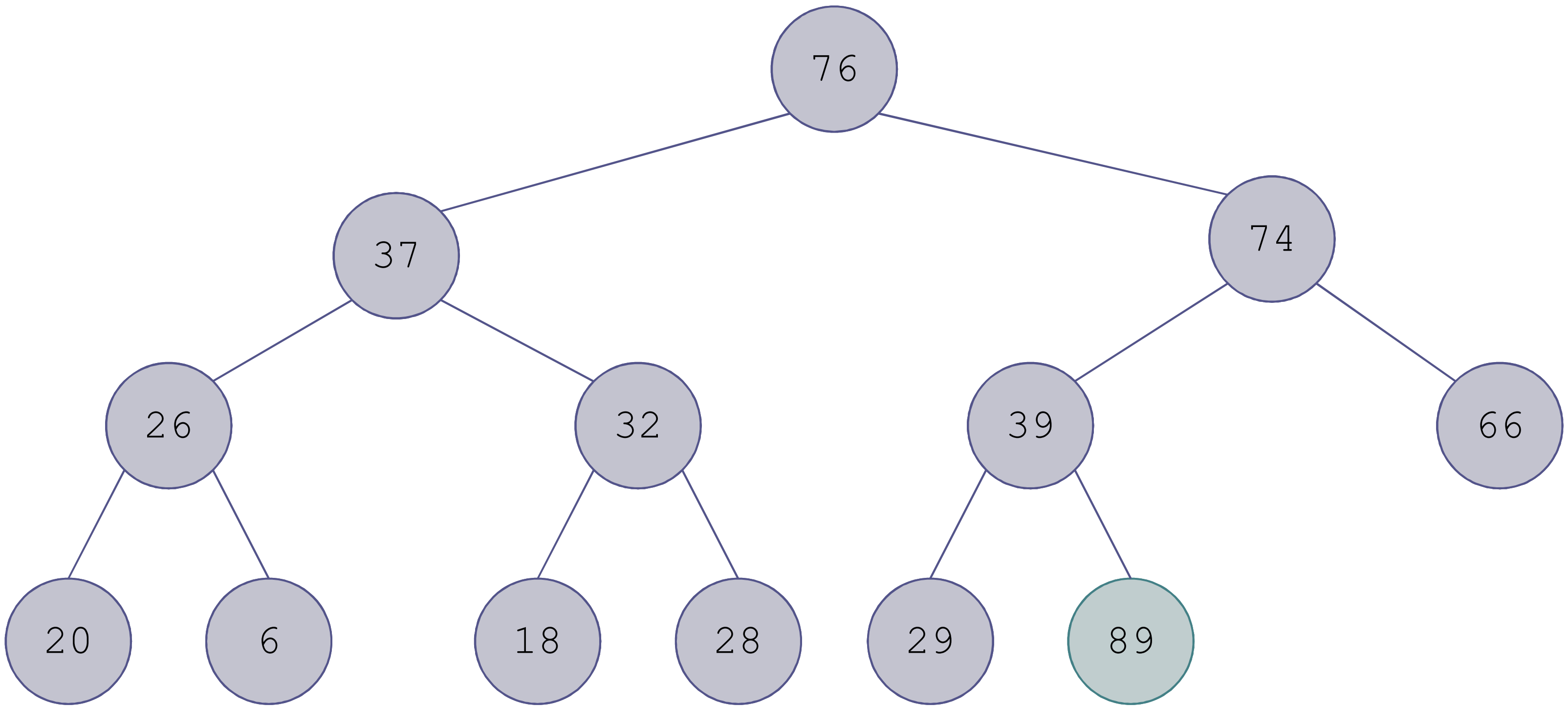
Trace of Heapsort (cont.)



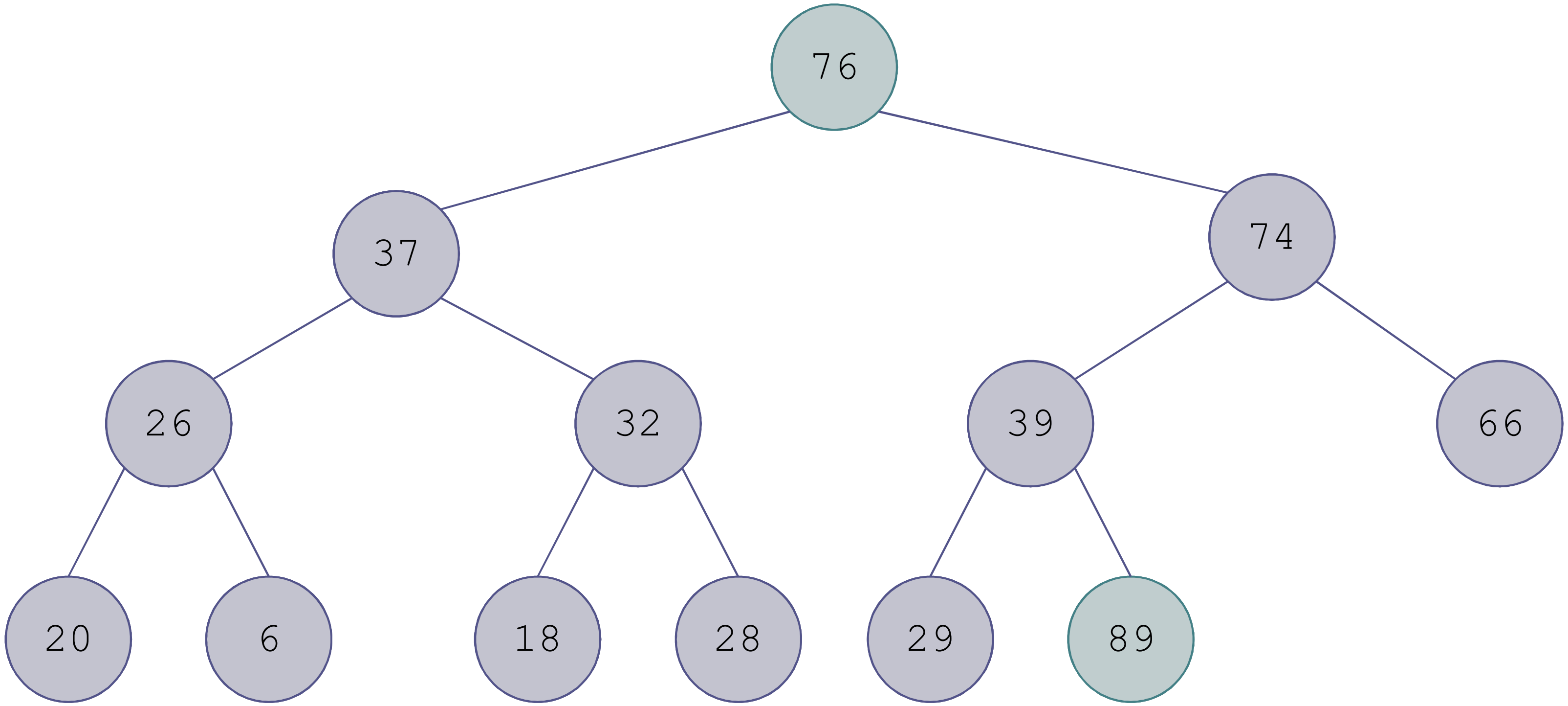
Trace of Heapsort (cont.)



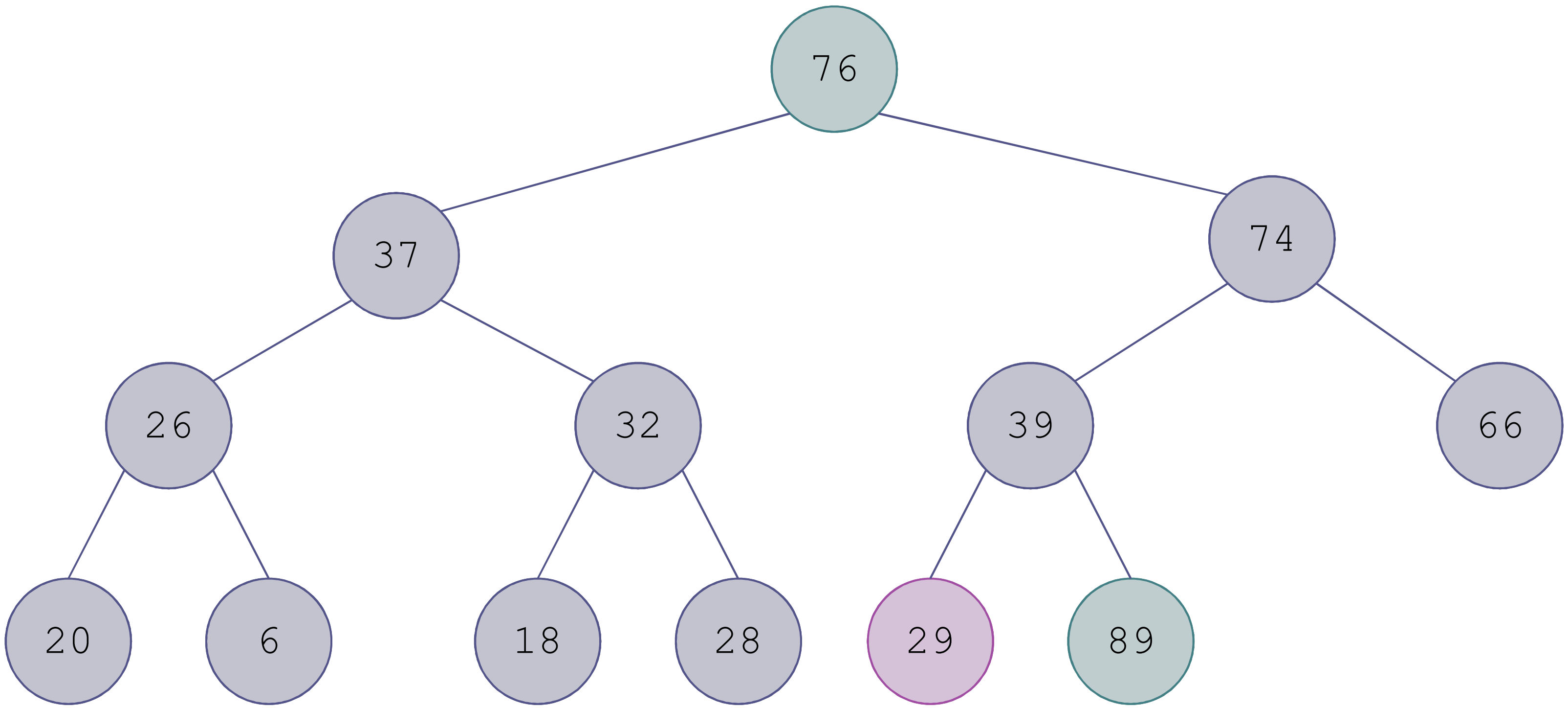
Trace of Heapsort (cont.)



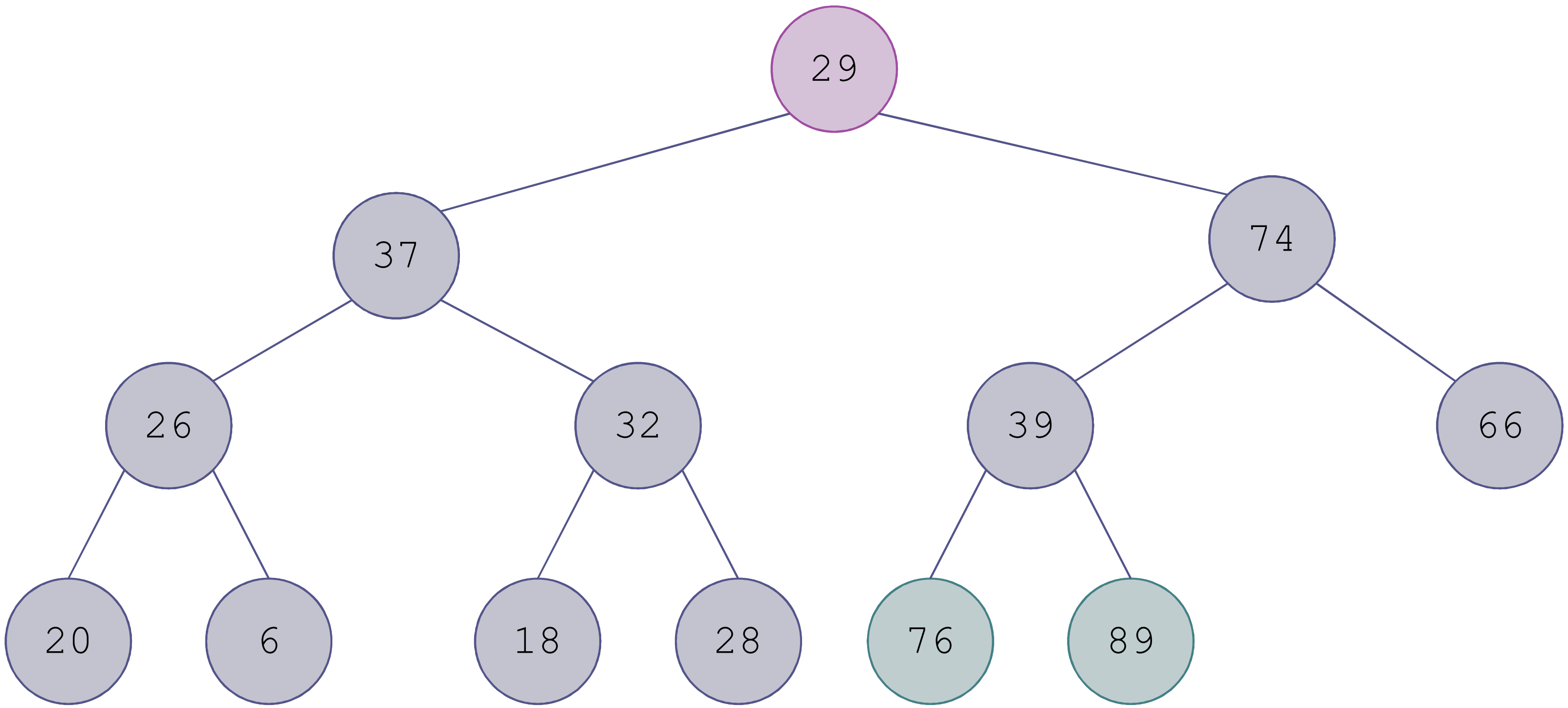
Trace of Heapsort (cont.)



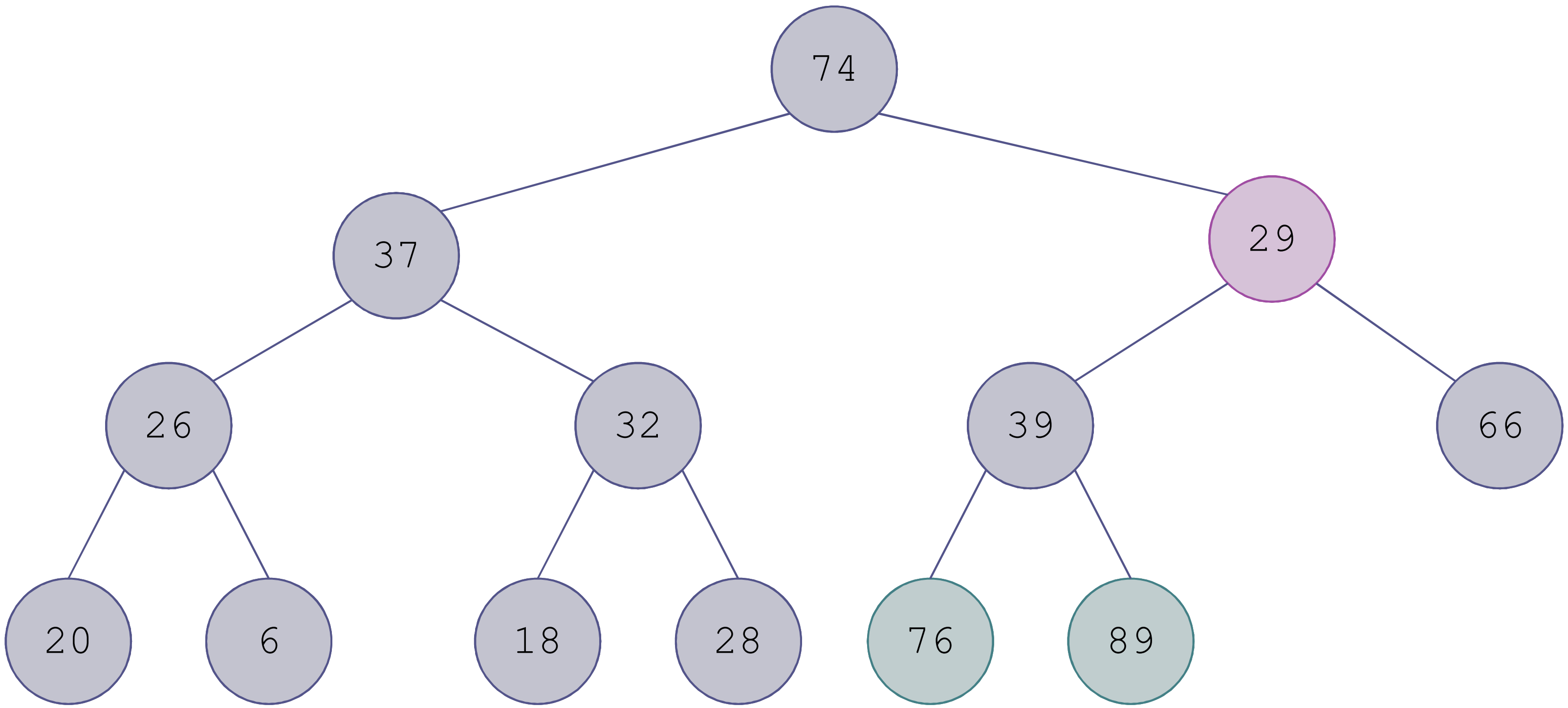
Trace of Heapsort (cont.)



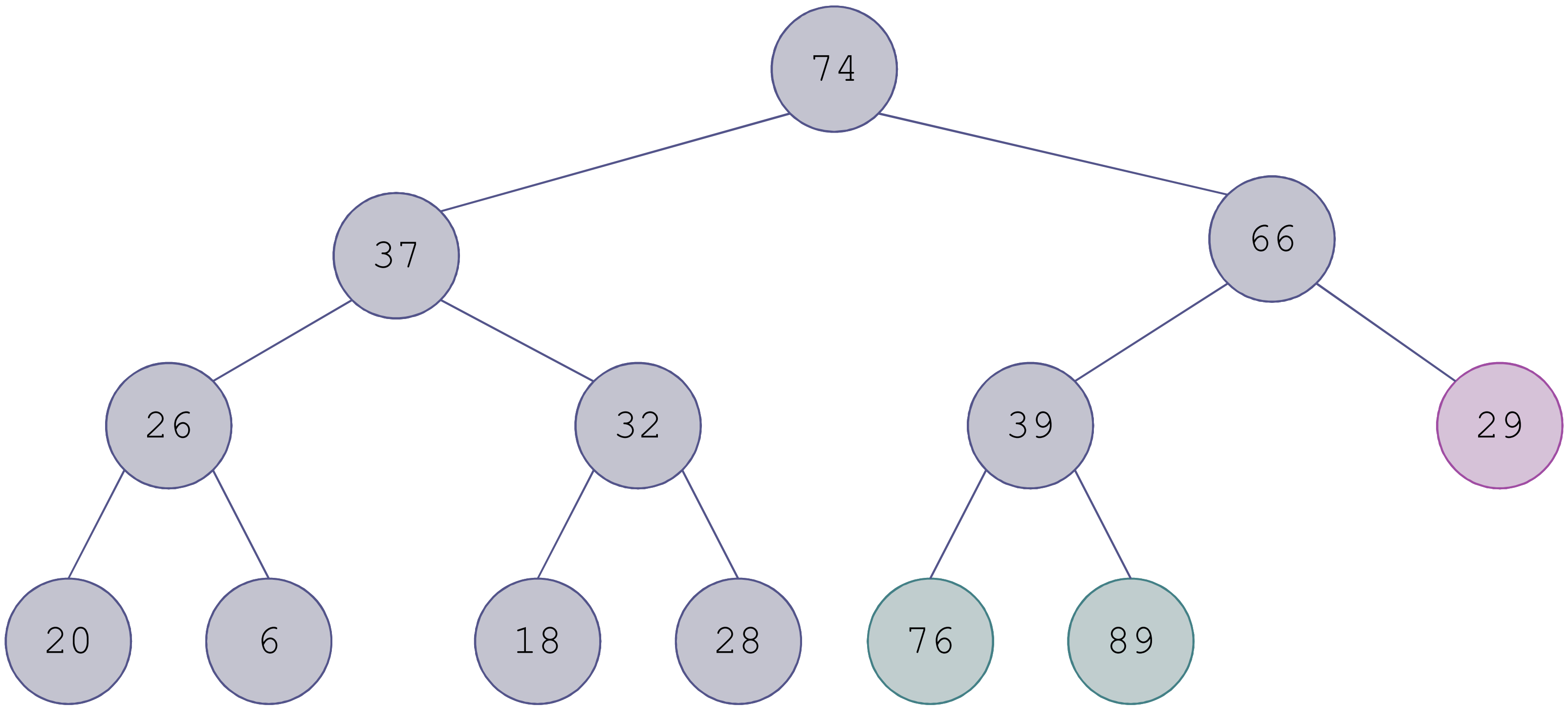
Trace of Heapsort (cont.)



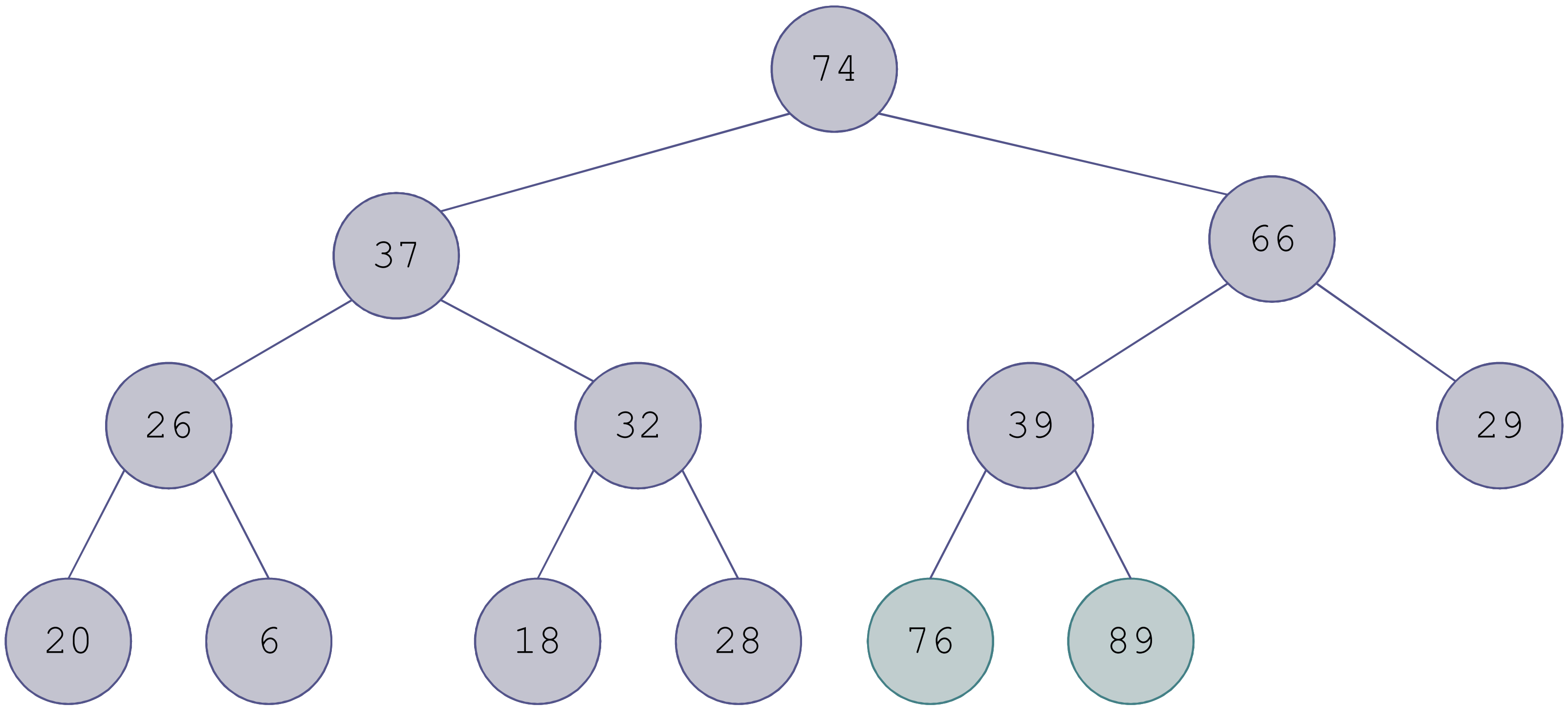
Trace of Heapsort (cont.)



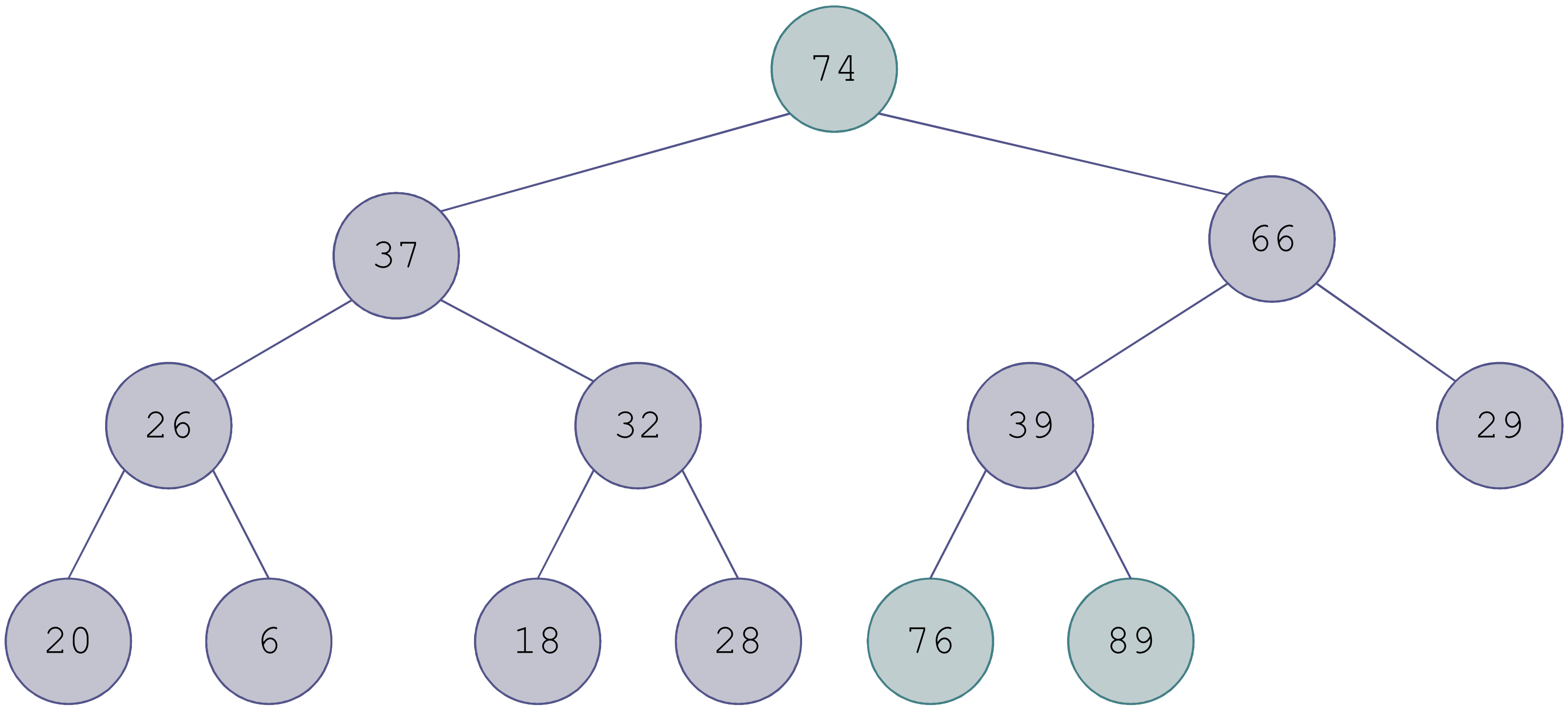
Trace of Heapsort (cont.)



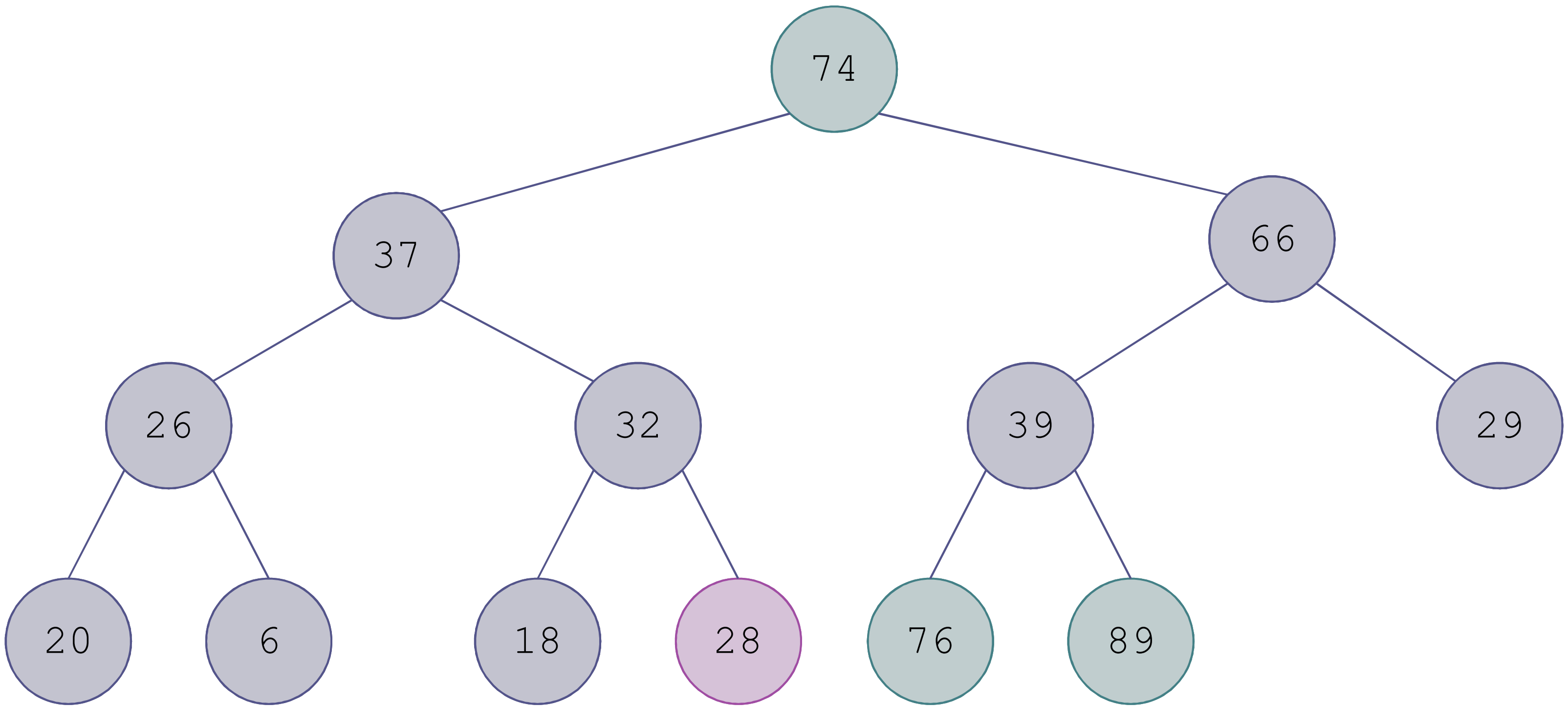
Trace of Heapsort (cont.)



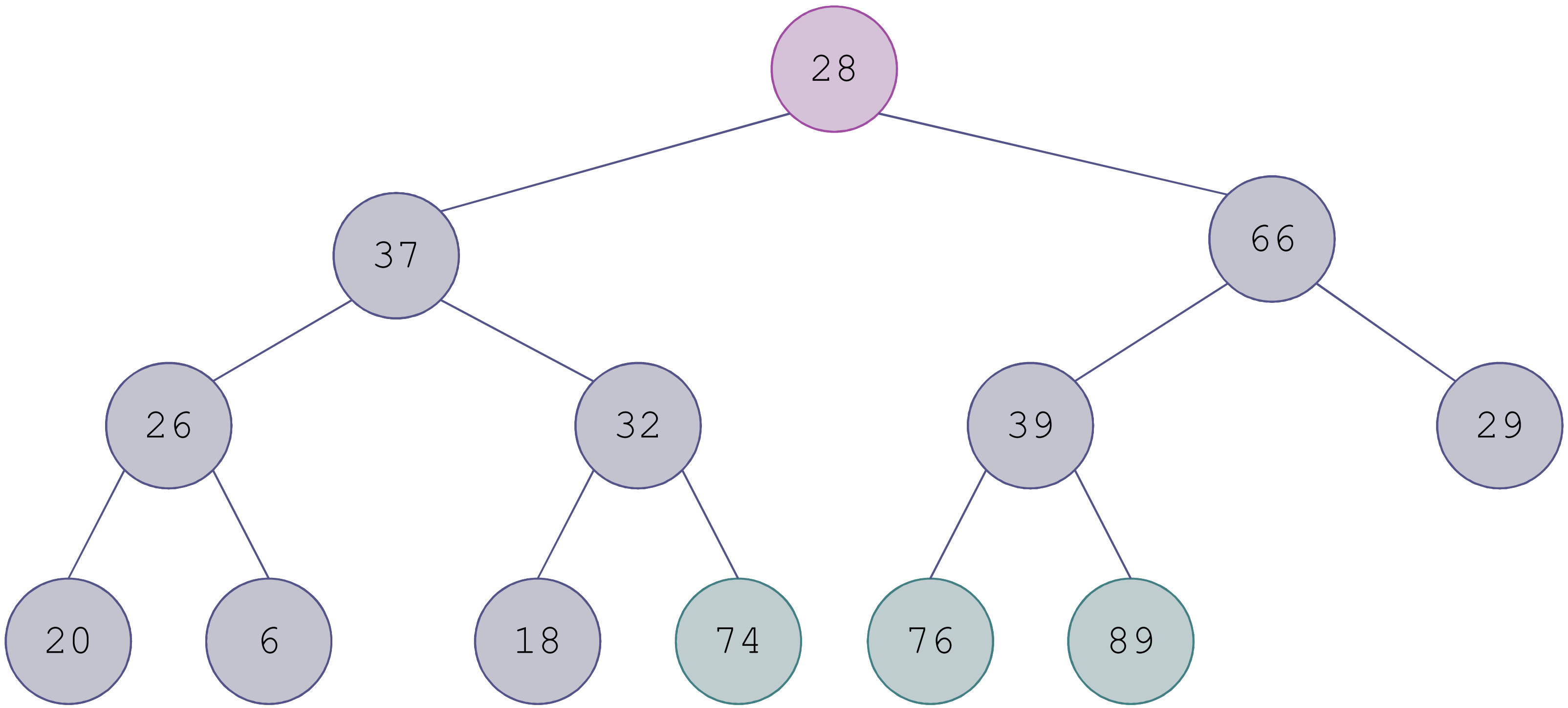
Trace of Heapsort (cont.)



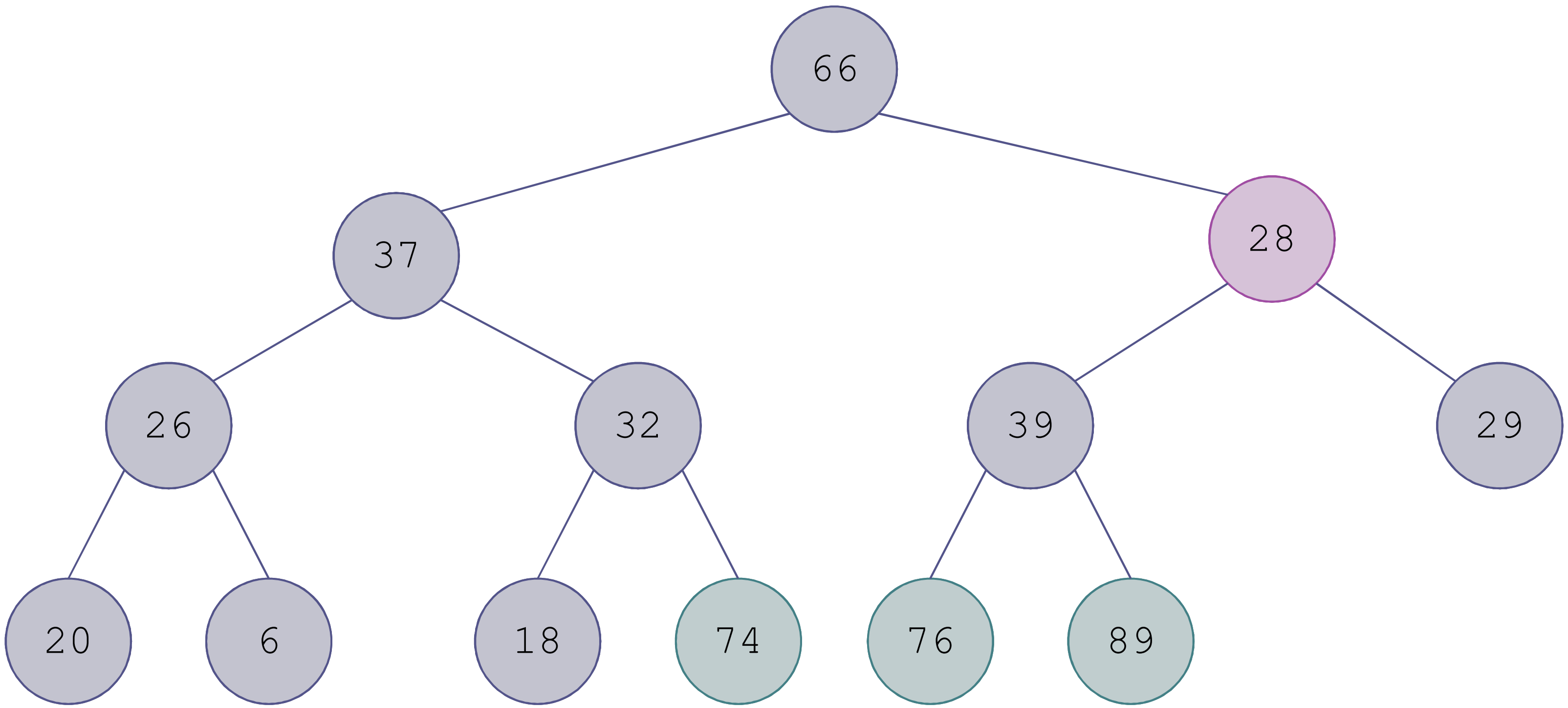
Trace of Heapsort (cont.)



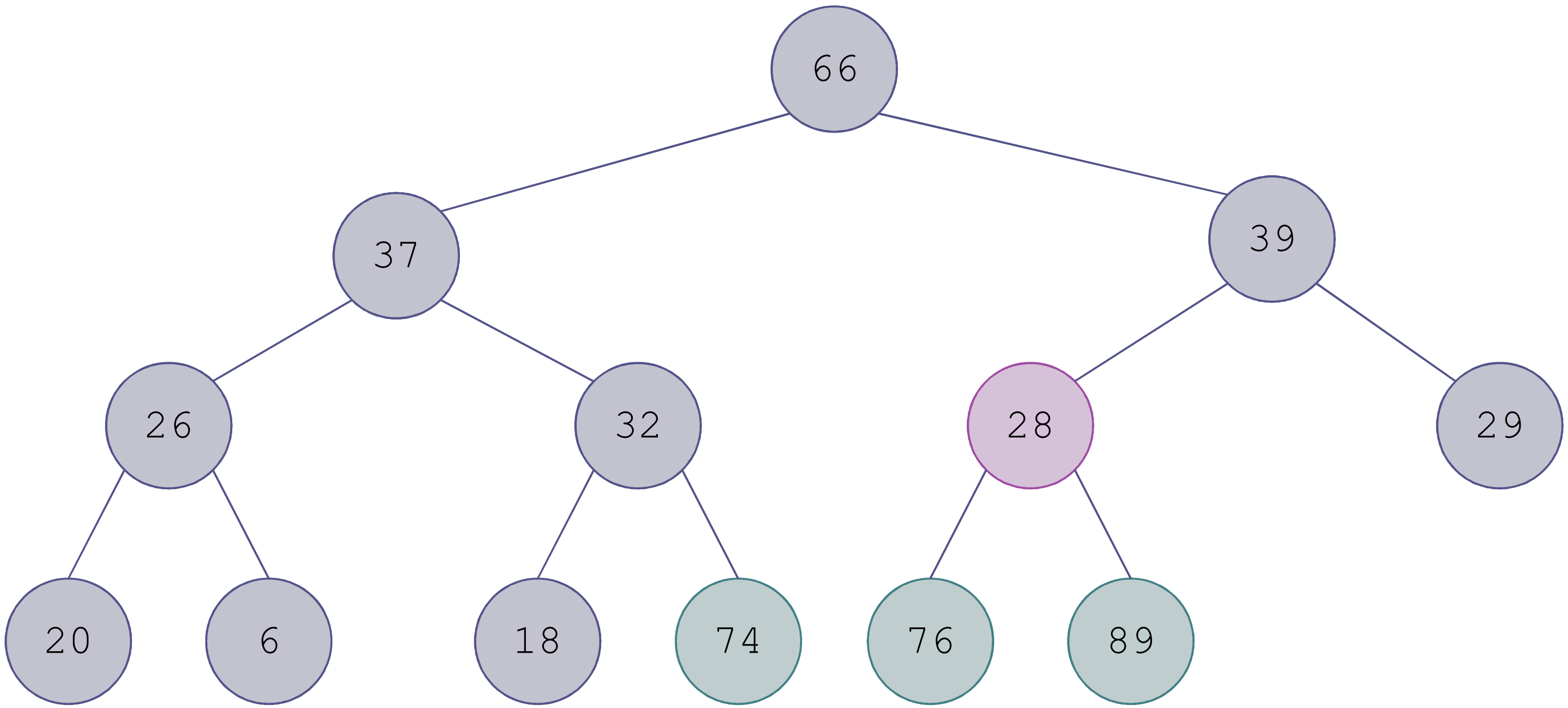
Trace of Heapsort (cont.)



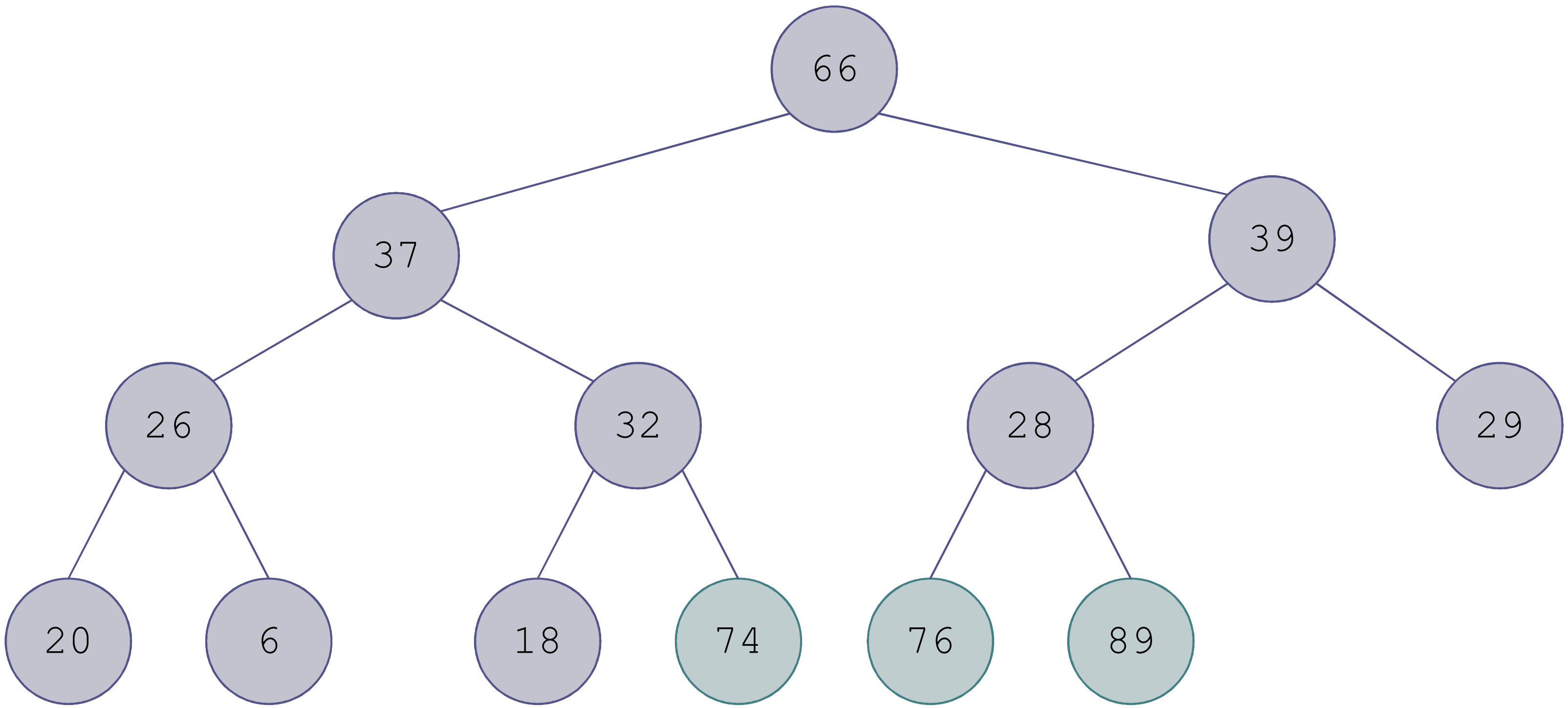
Trace of Heapsort (cont.)



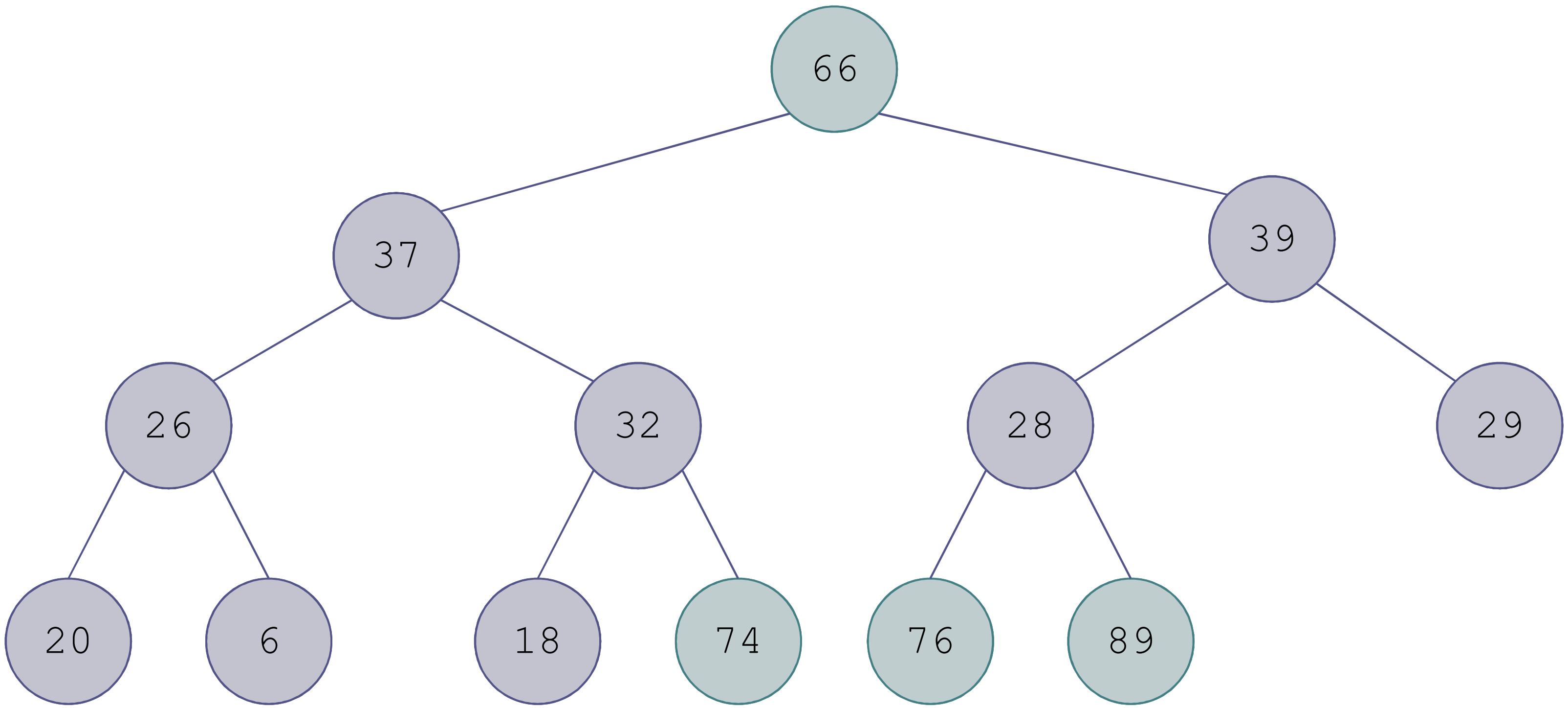
Trace of Heapsort (cont.)



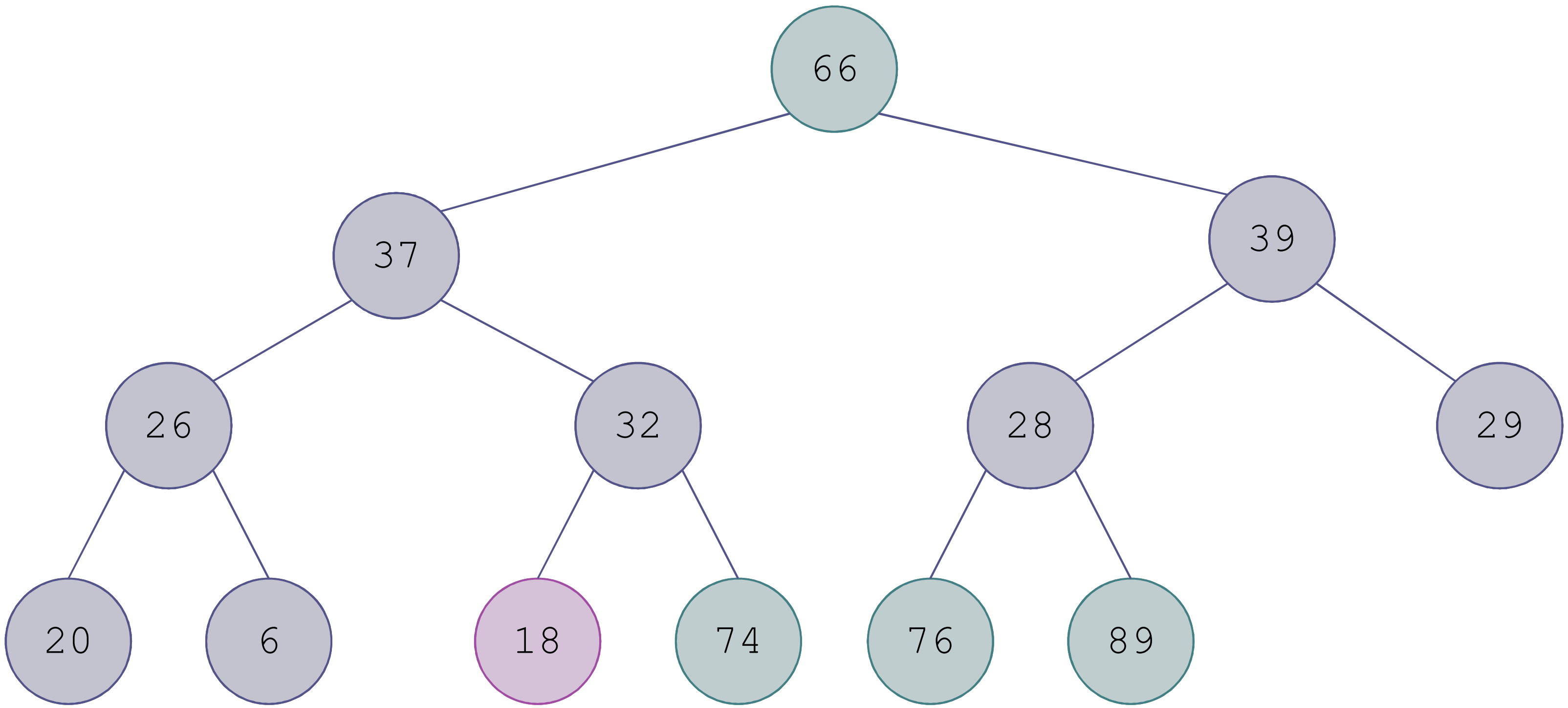
Trace of Heapsort (cont.)



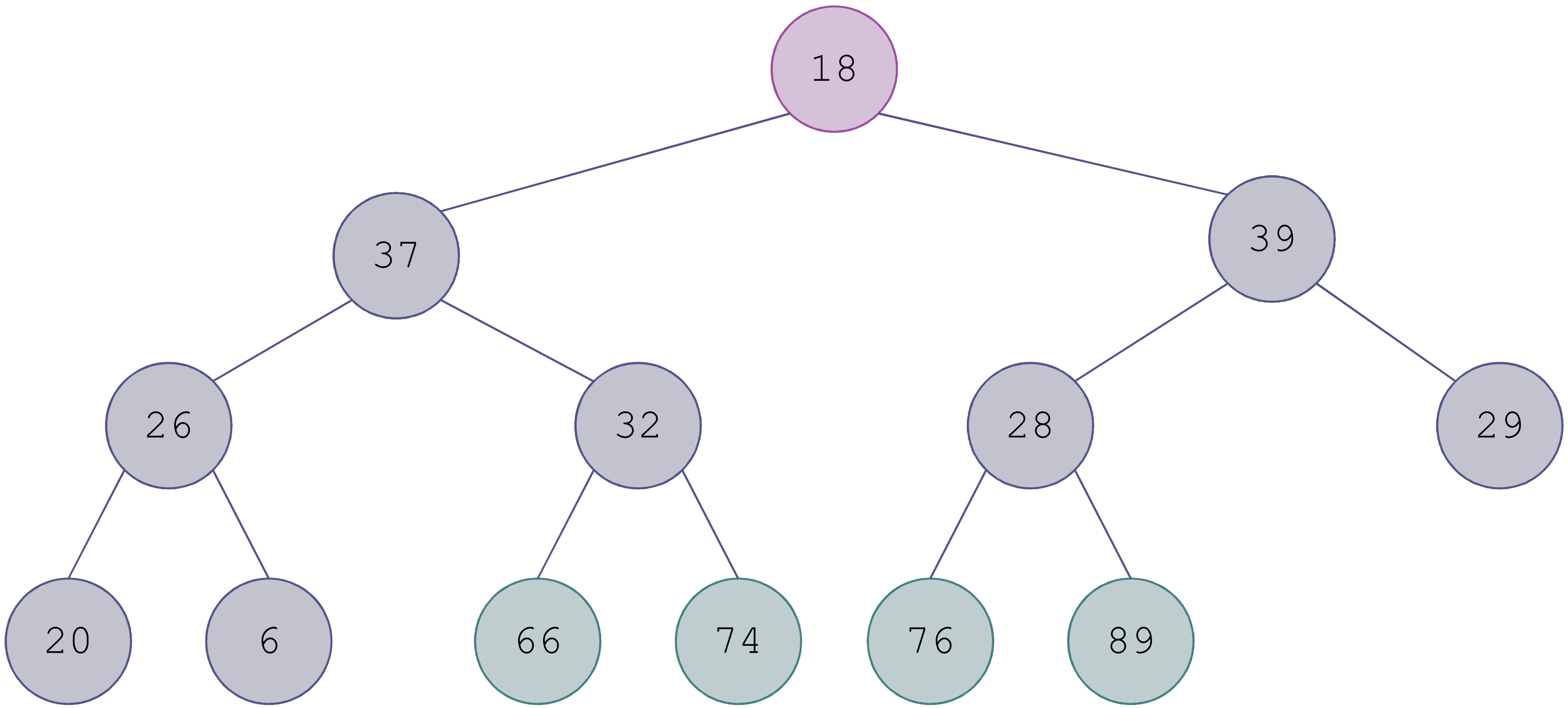
Trace of Heapsort (cont.)



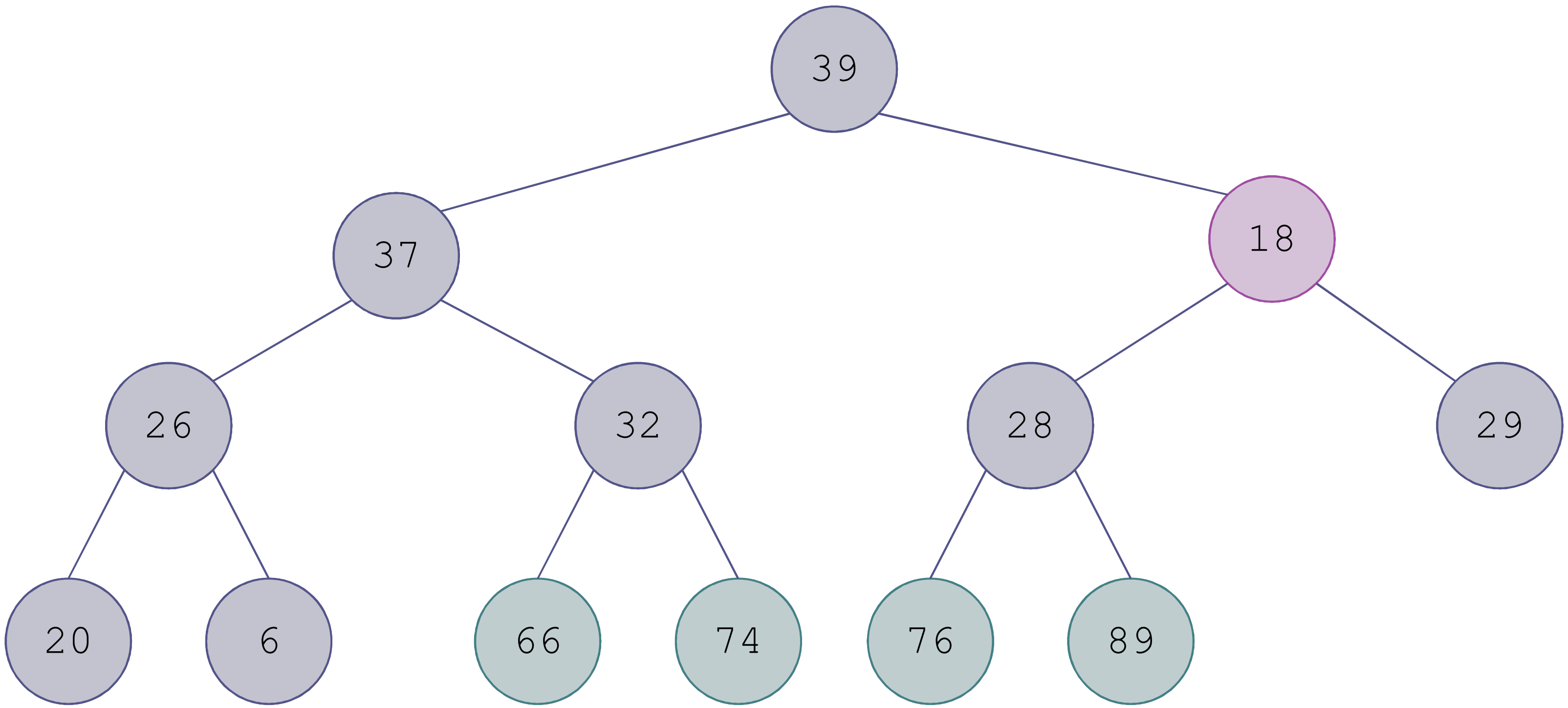
Trace of Heapsort (cont.)



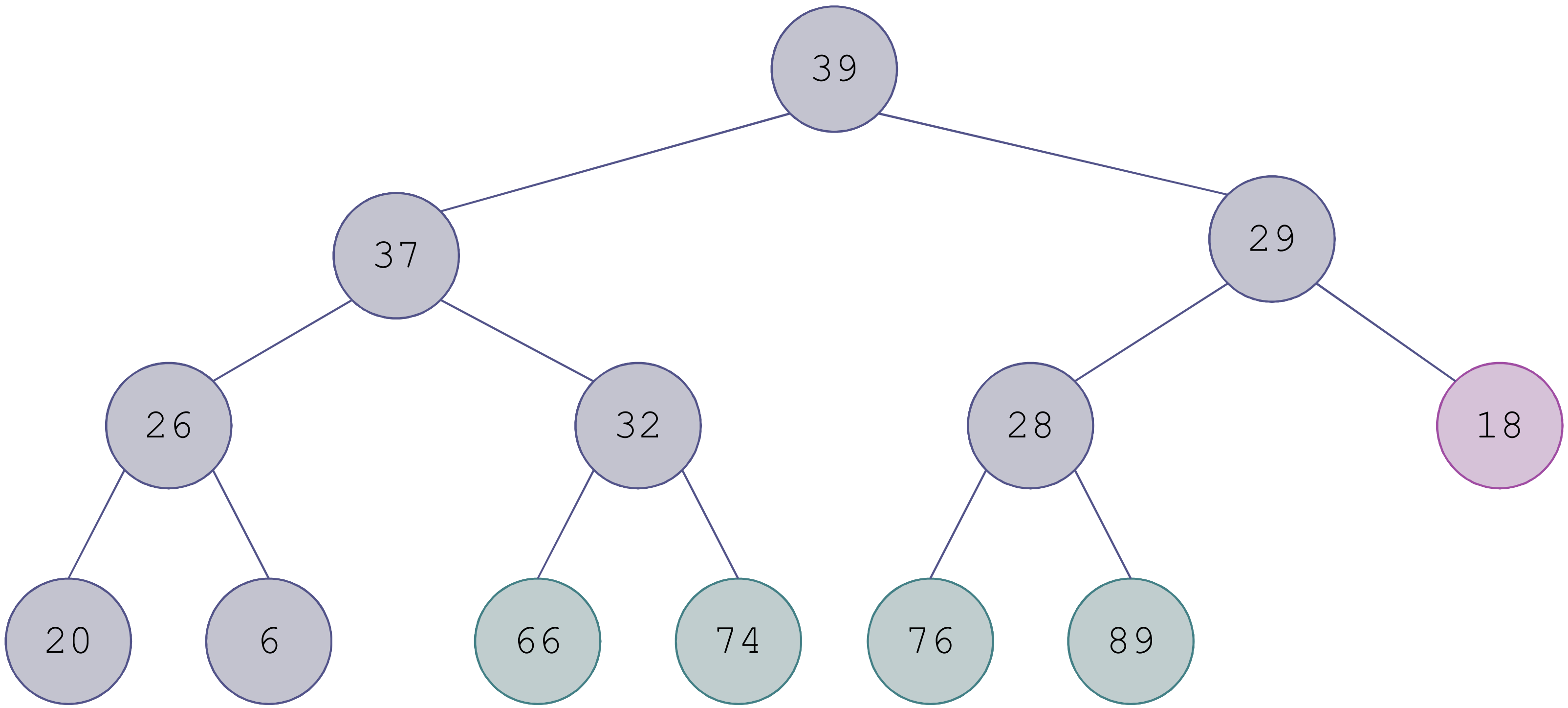
Trace of Heapsort (cont.)



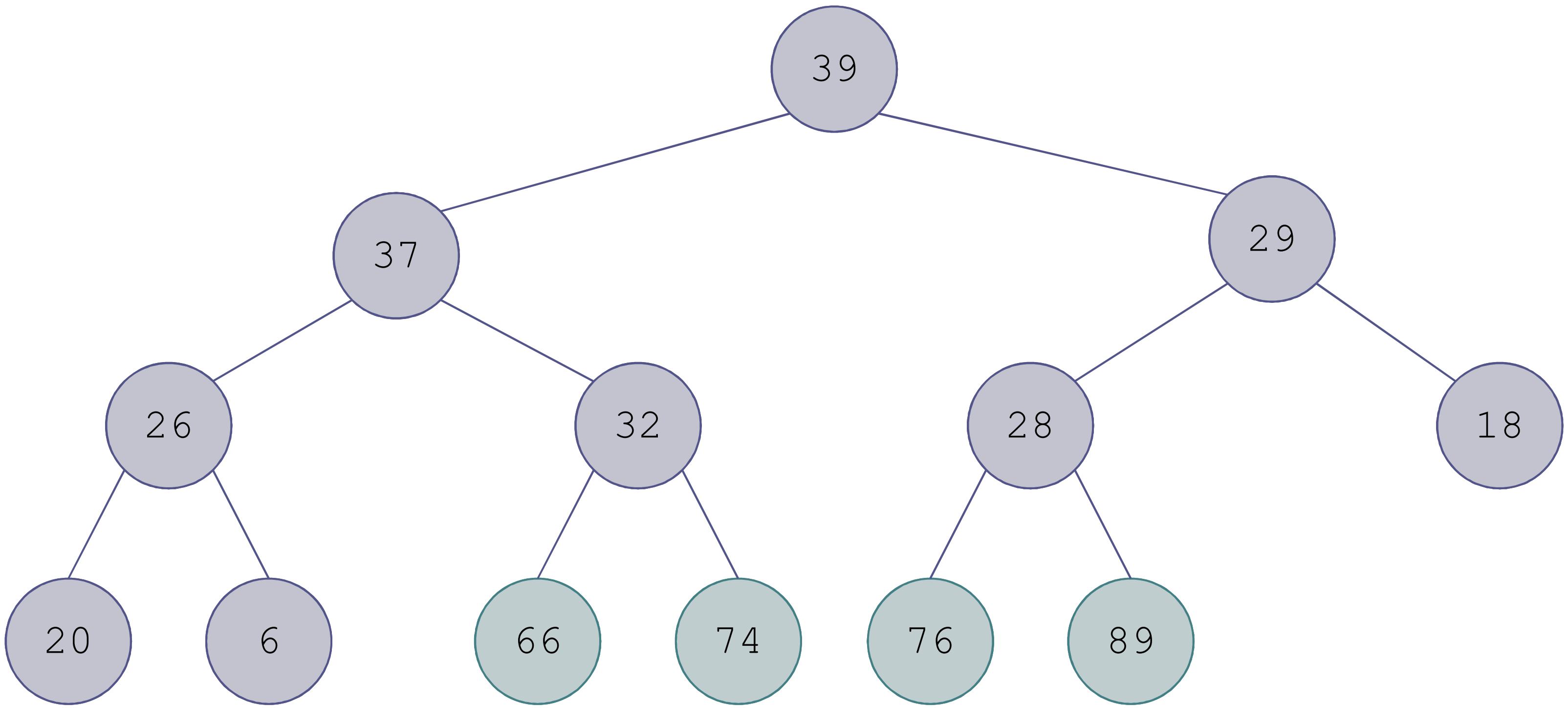
Trace of Heapsort (cont.)



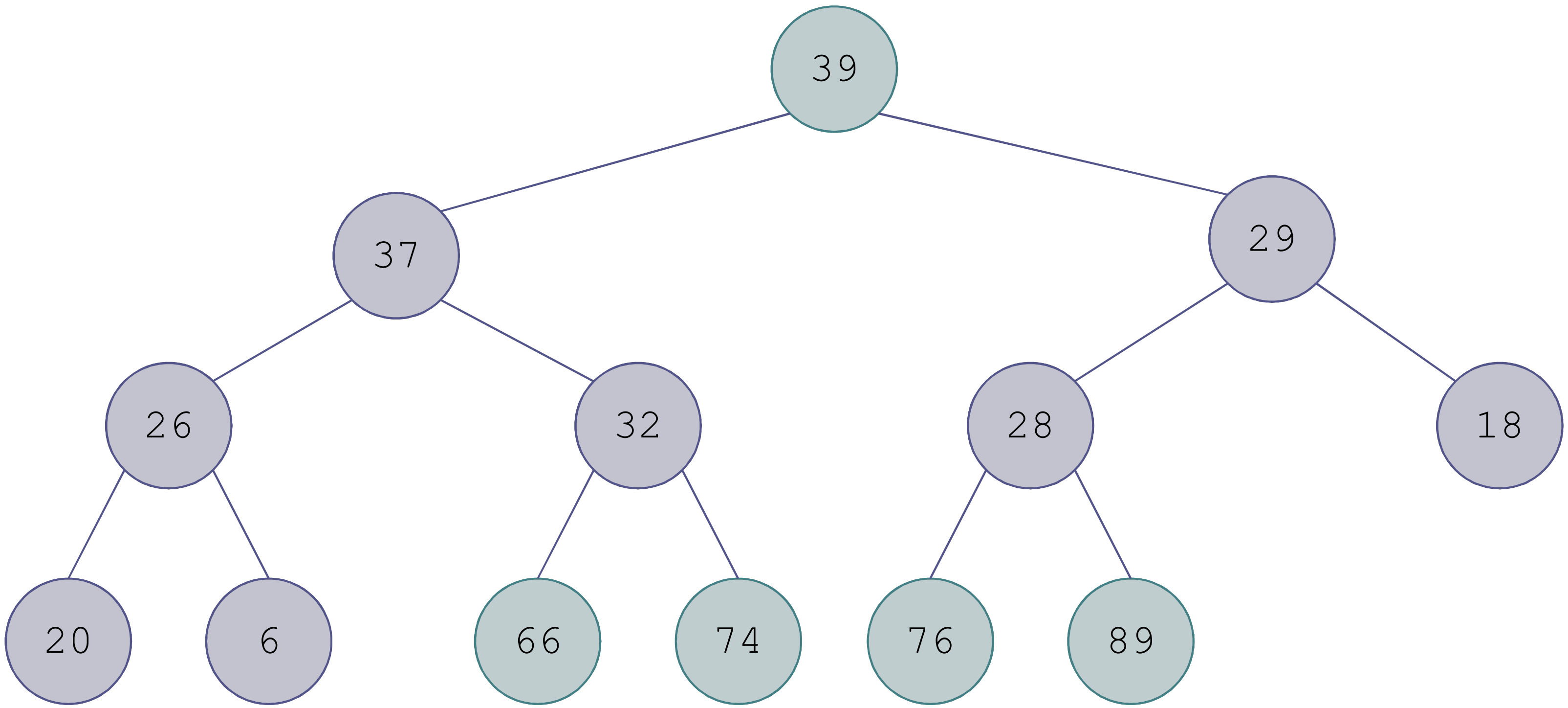
Trace of Heapsort (cont.)



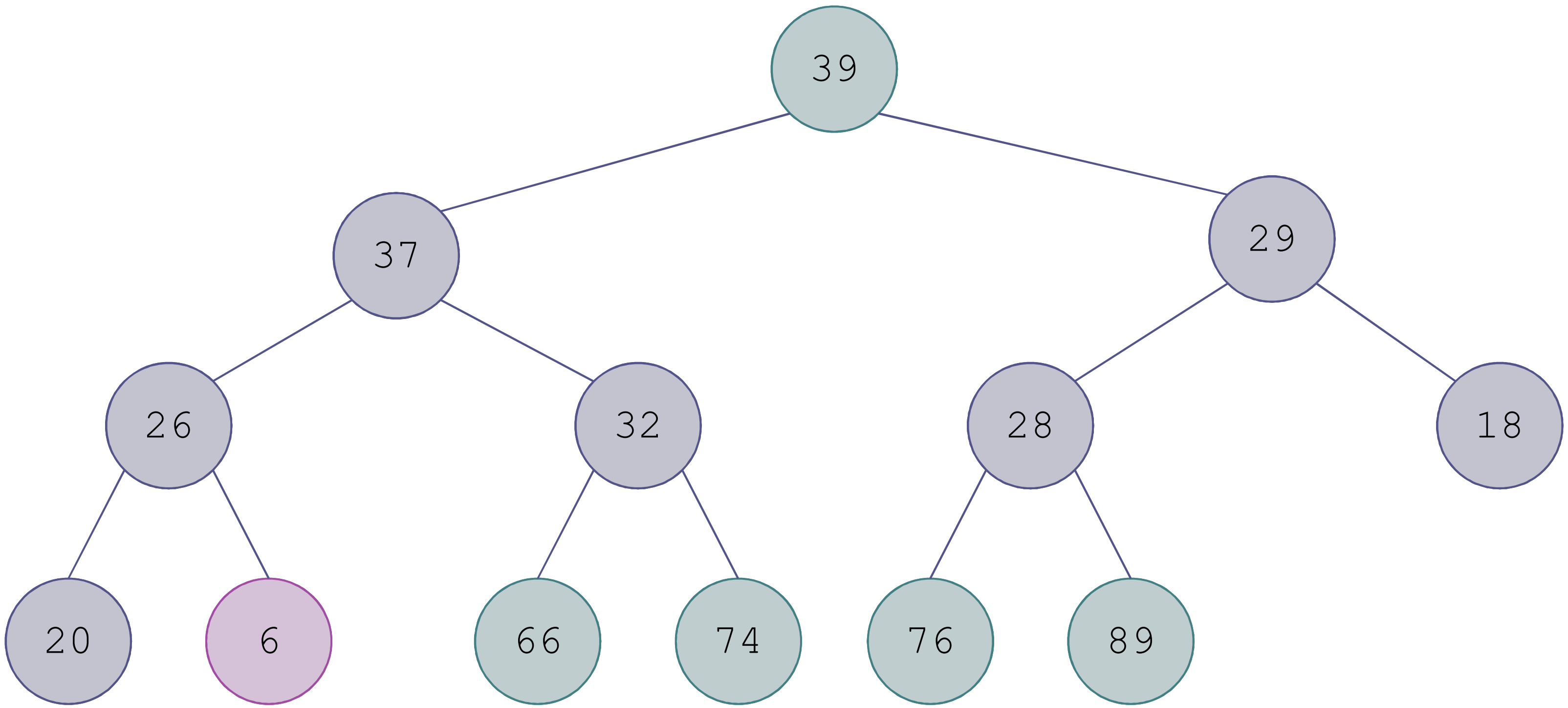
Trace of Heapsort (cont.)



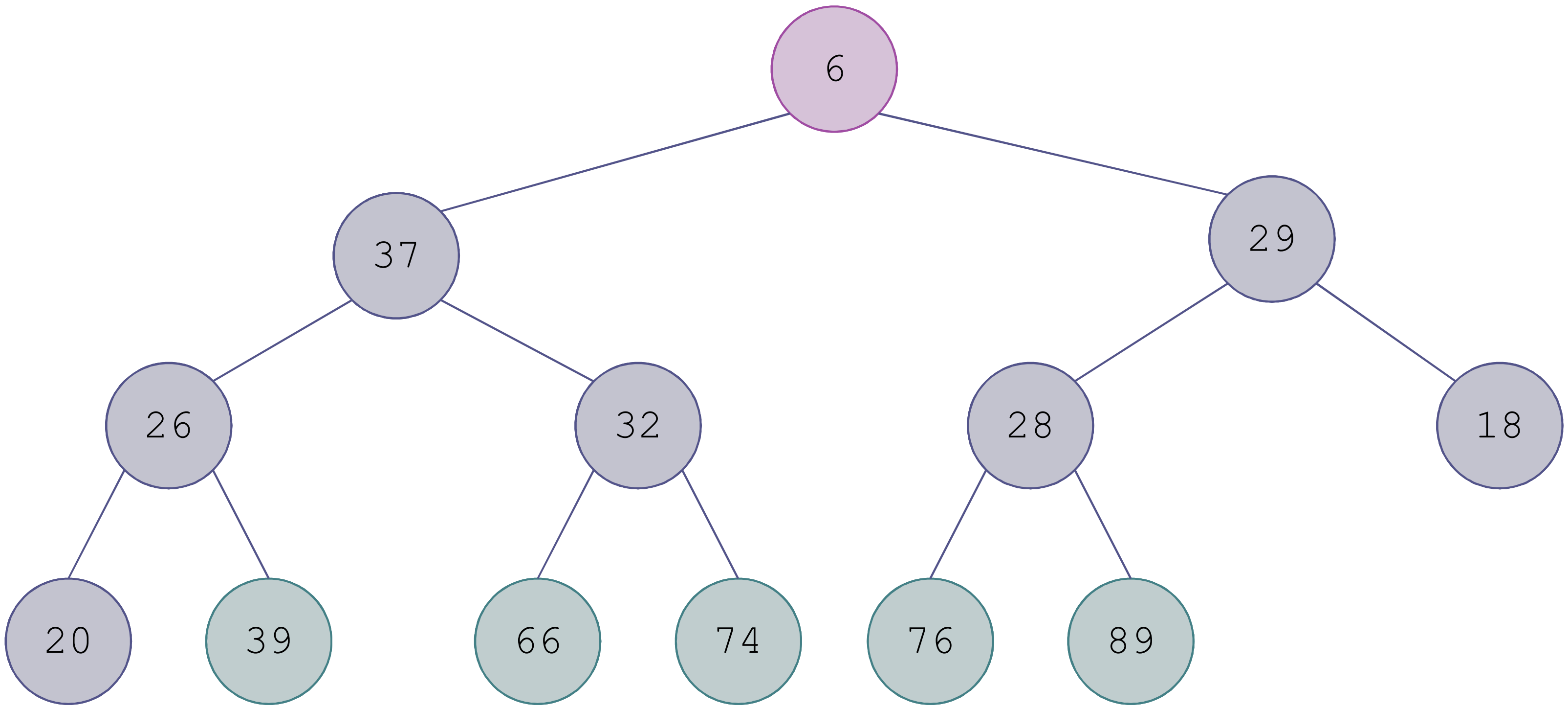
Trace of Heapsort (cont.)



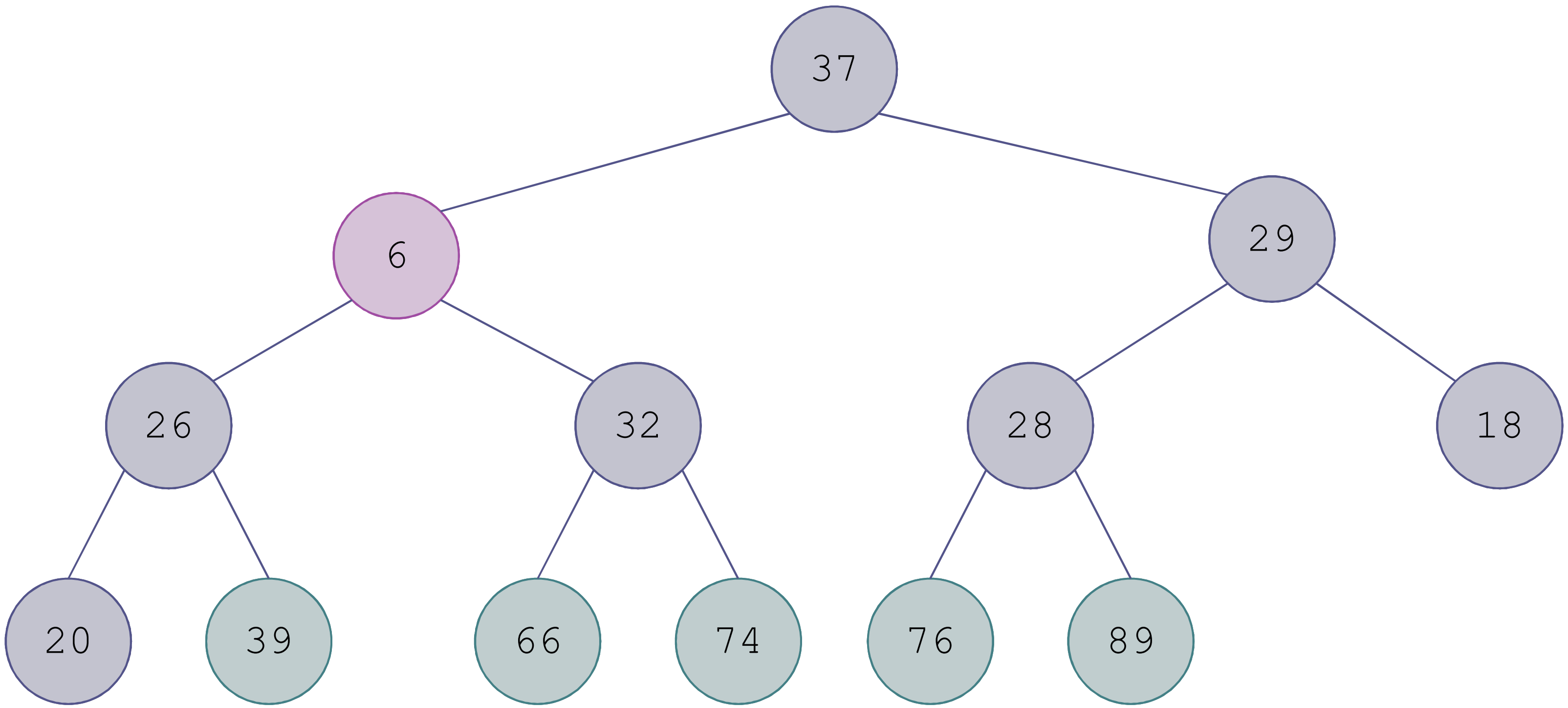
Trace of Heapsort (cont.)



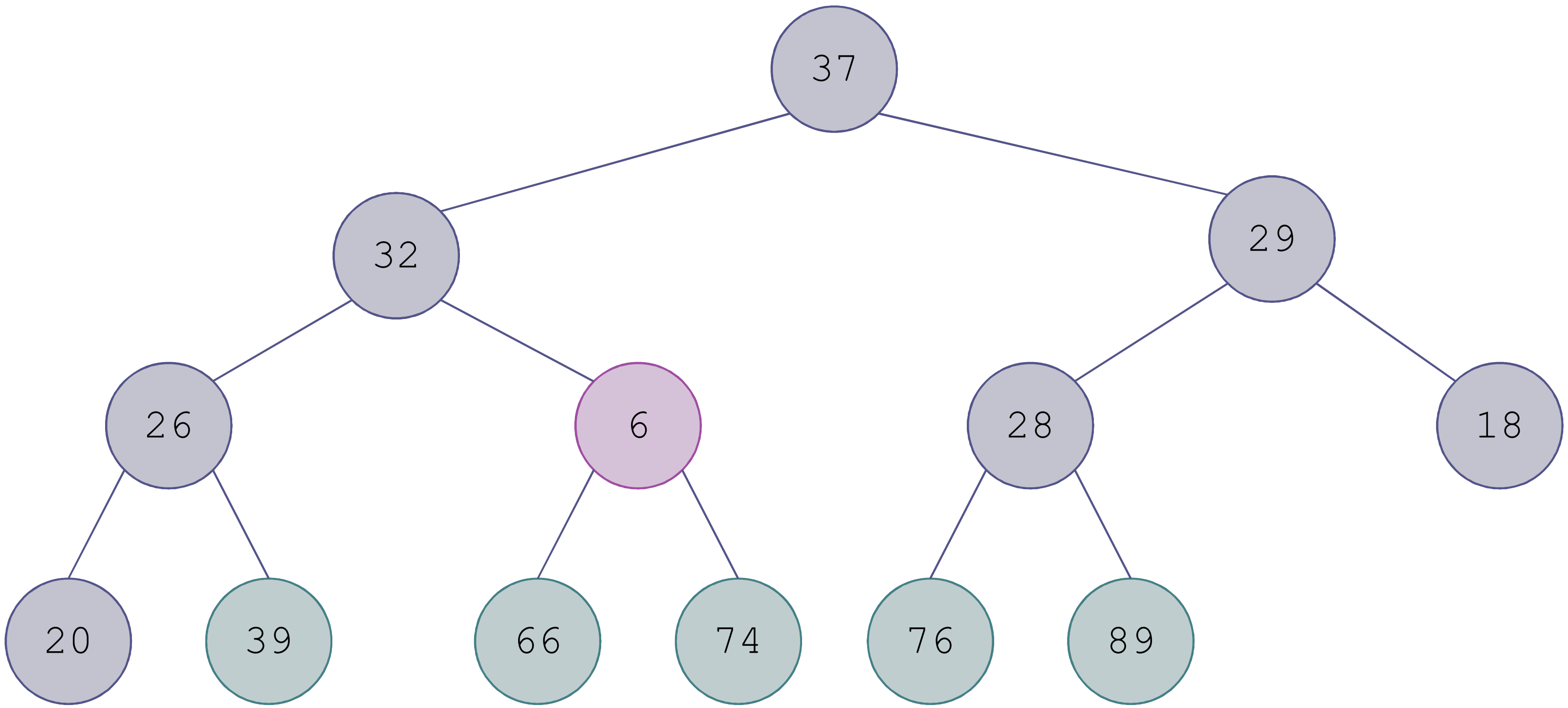
Trace of Heapsort (cont.)



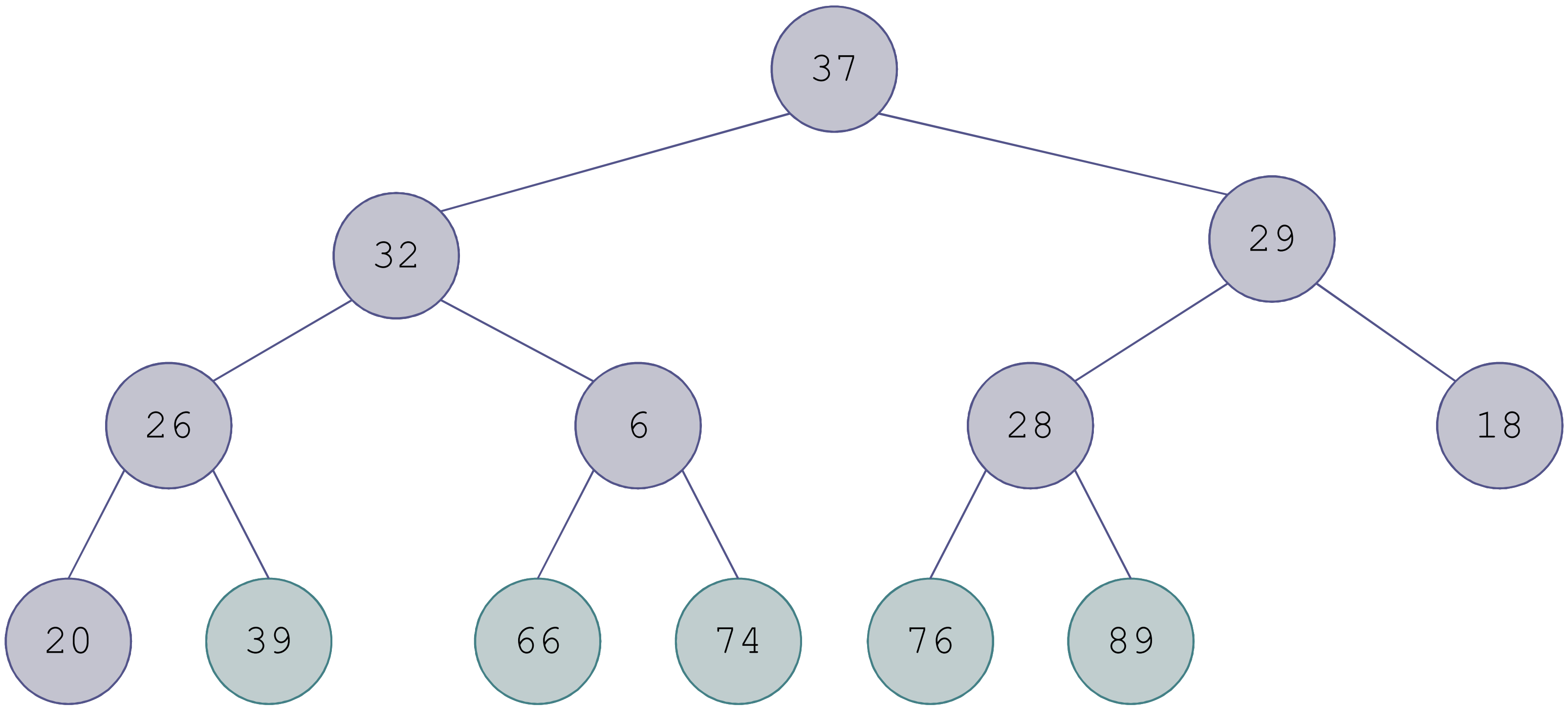
Trace of Heapsort (cont.)



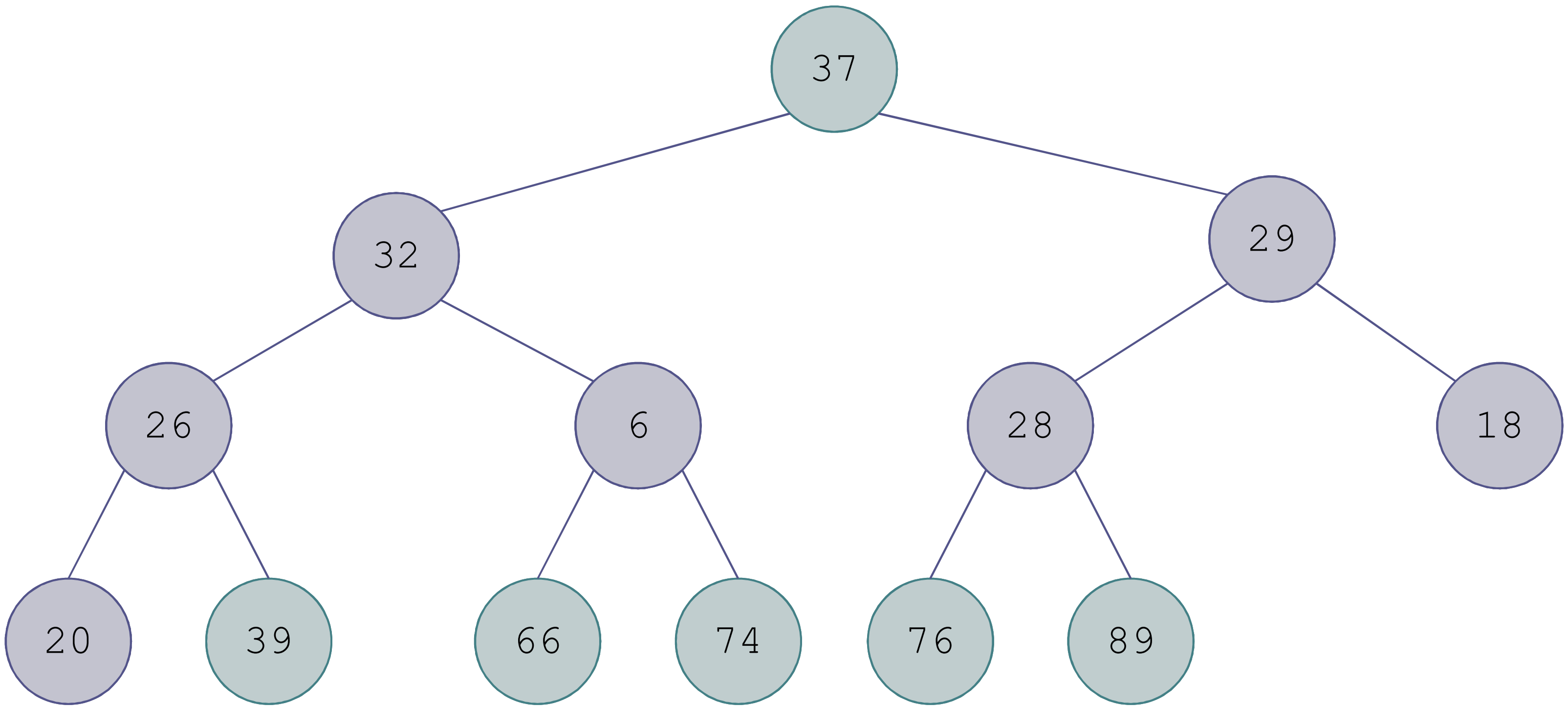
Trace of Heapsort (cont.)



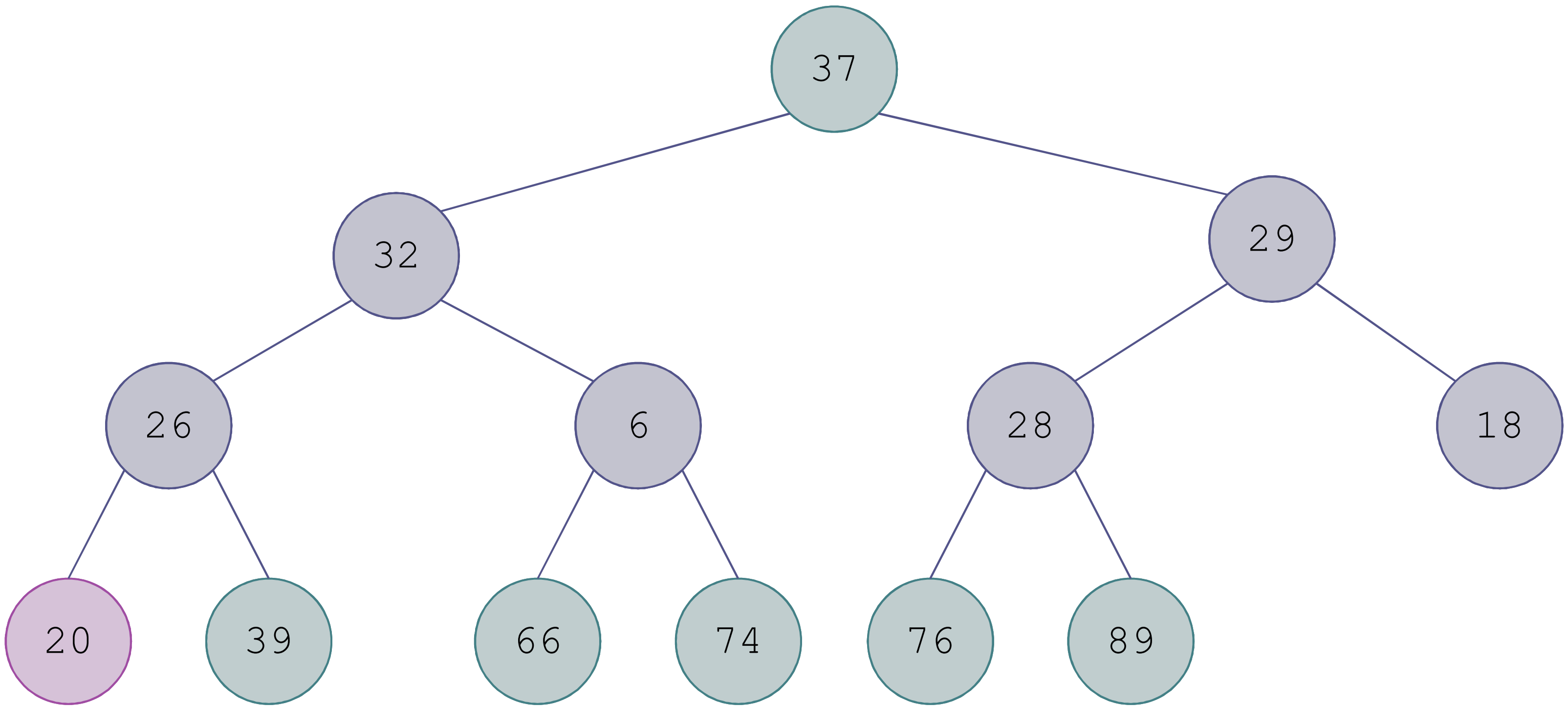
Trace of Heapsort (cont.)



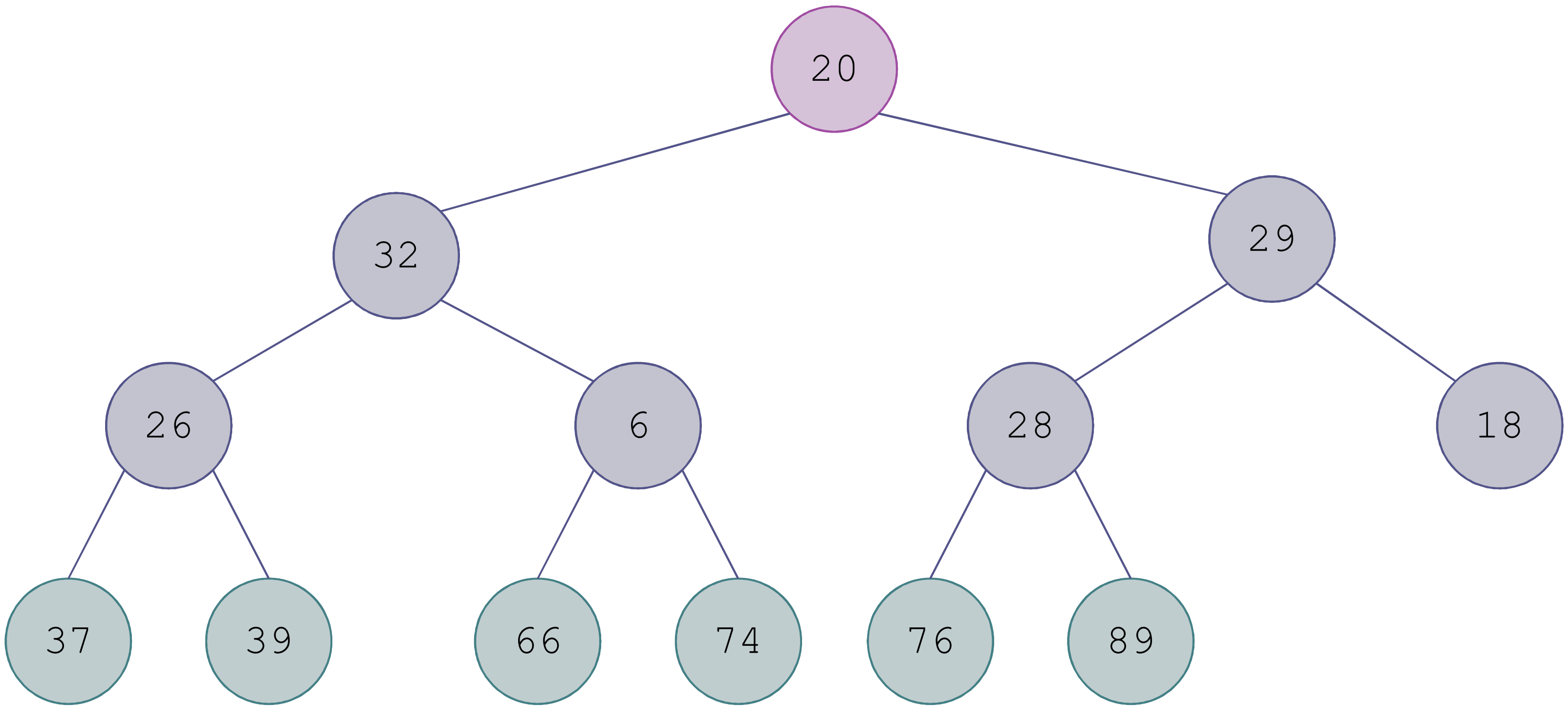
Trace of Heapsort (cont.)



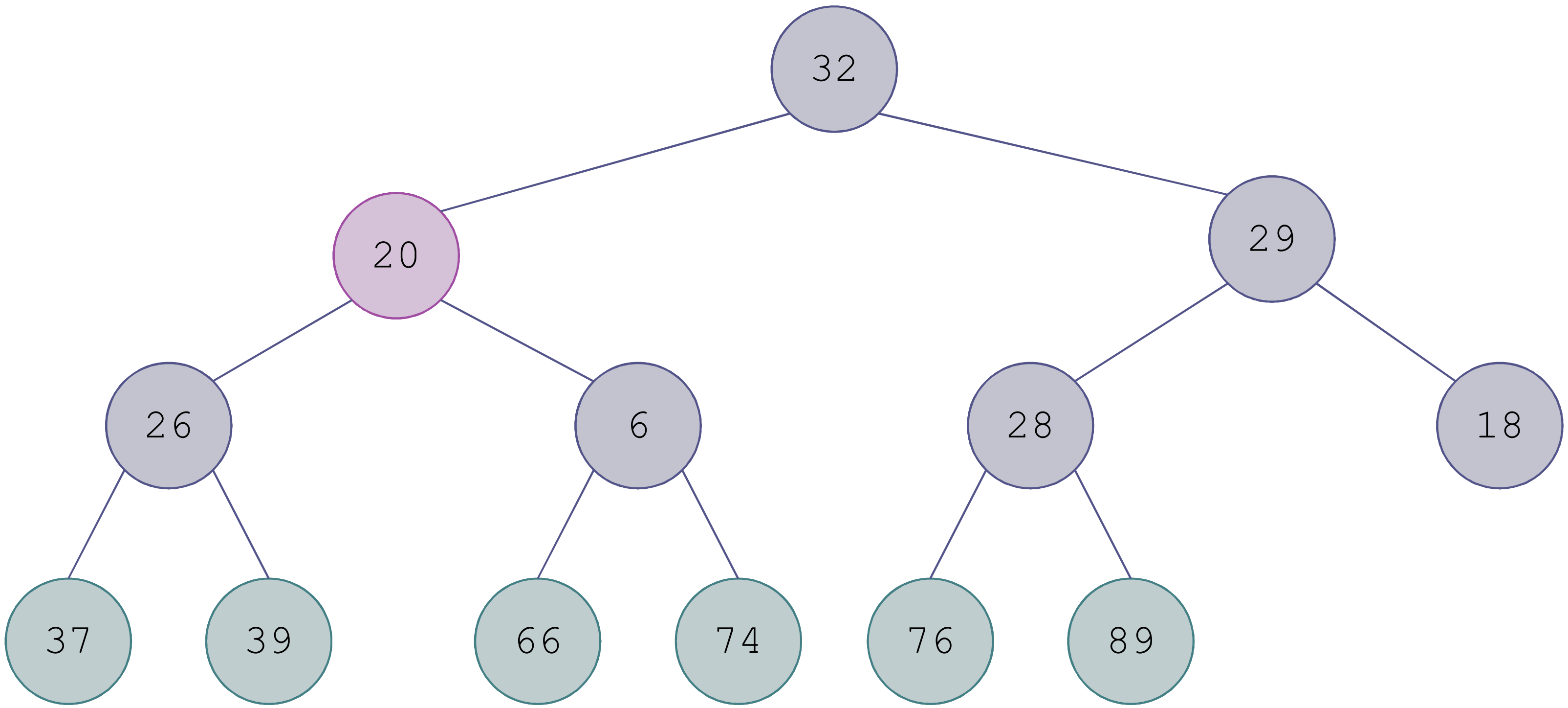
Trace of Heapsort (cont.)



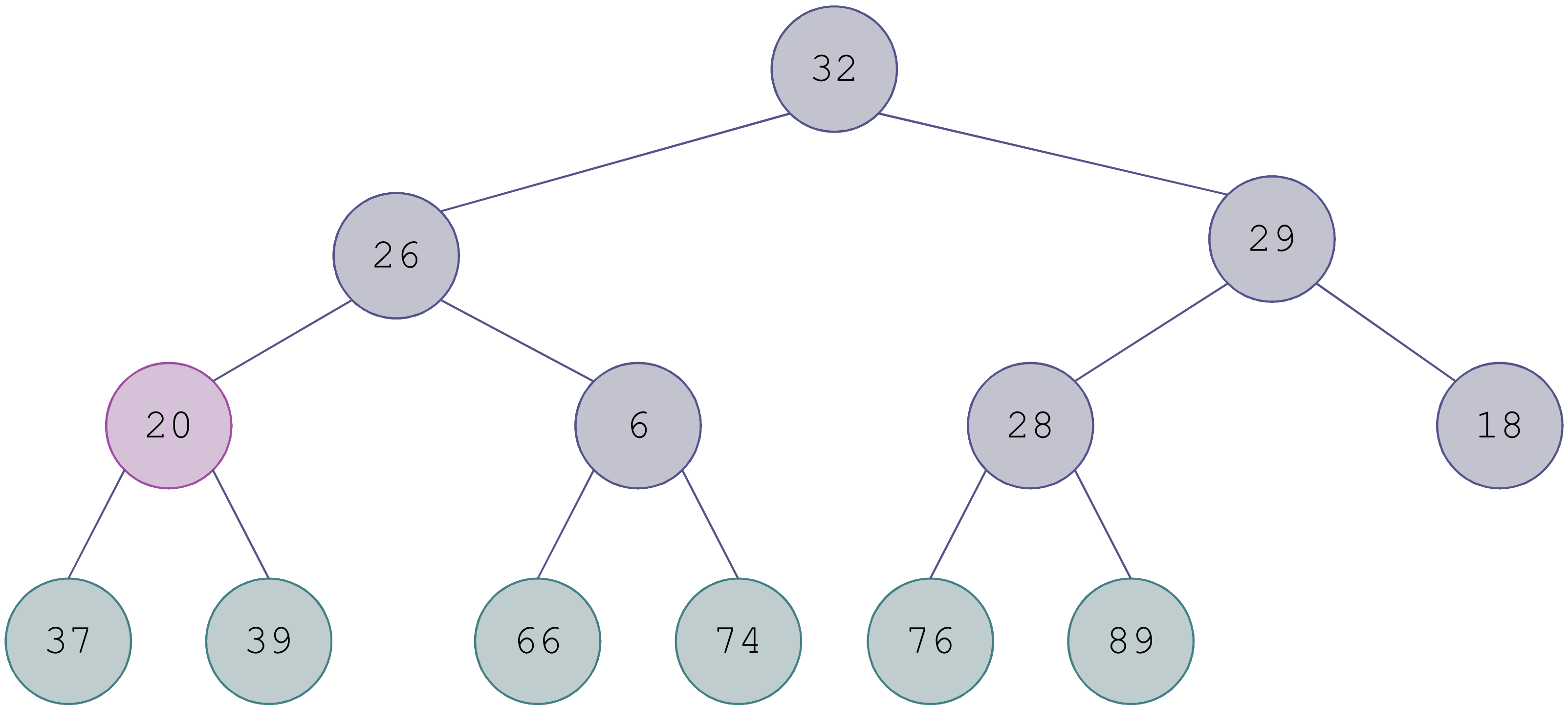
Trace of Heapsort (cont.)



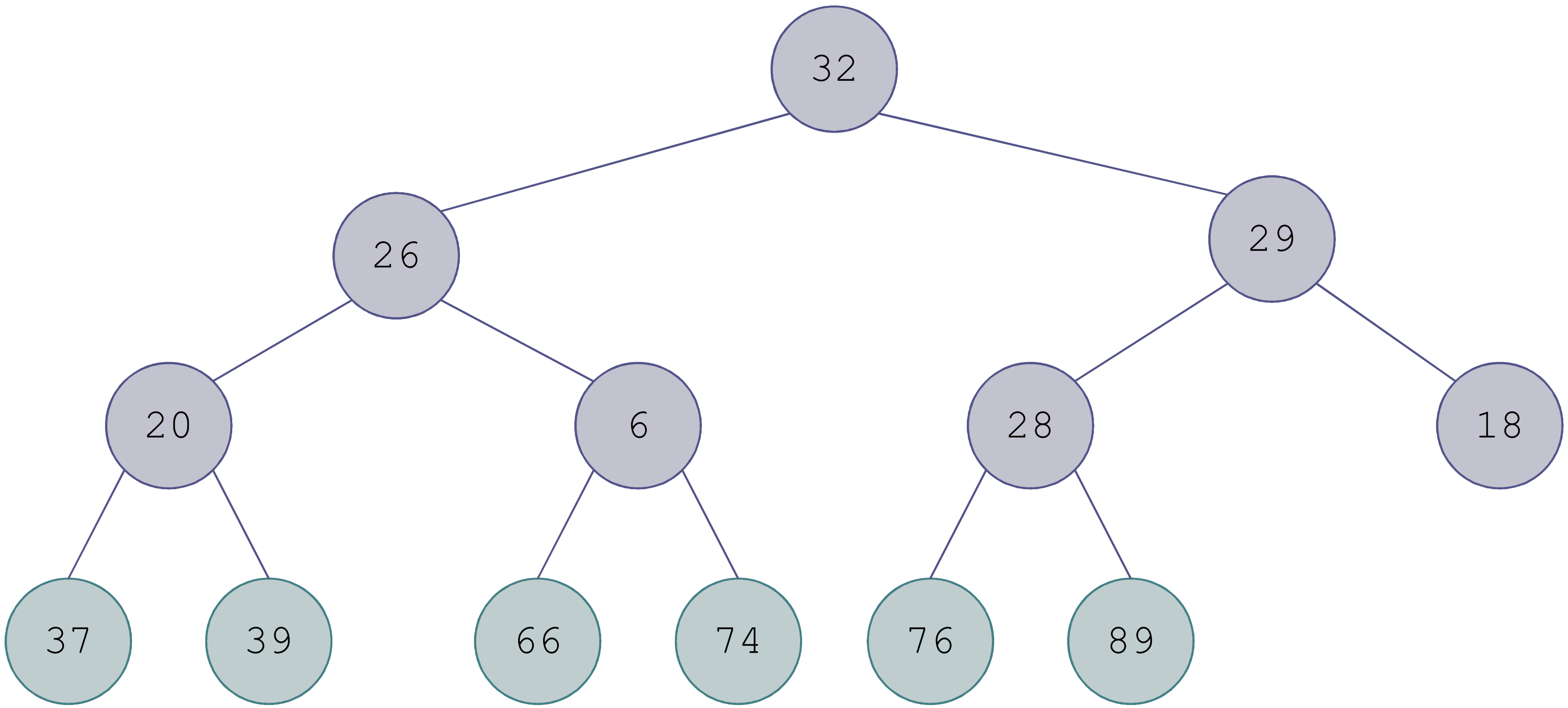
Trace of Heapsort (cont.)



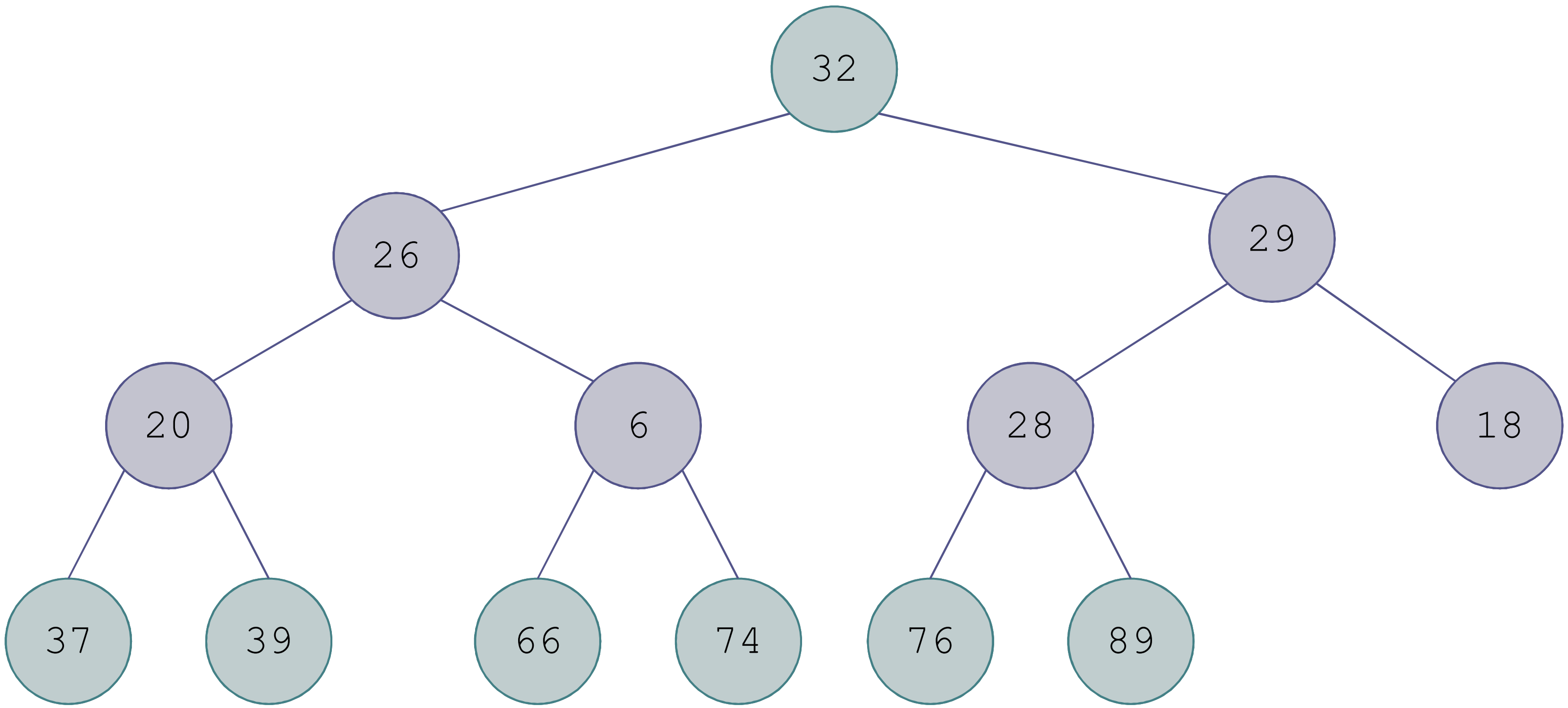
Trace of Heapsort (cont.)



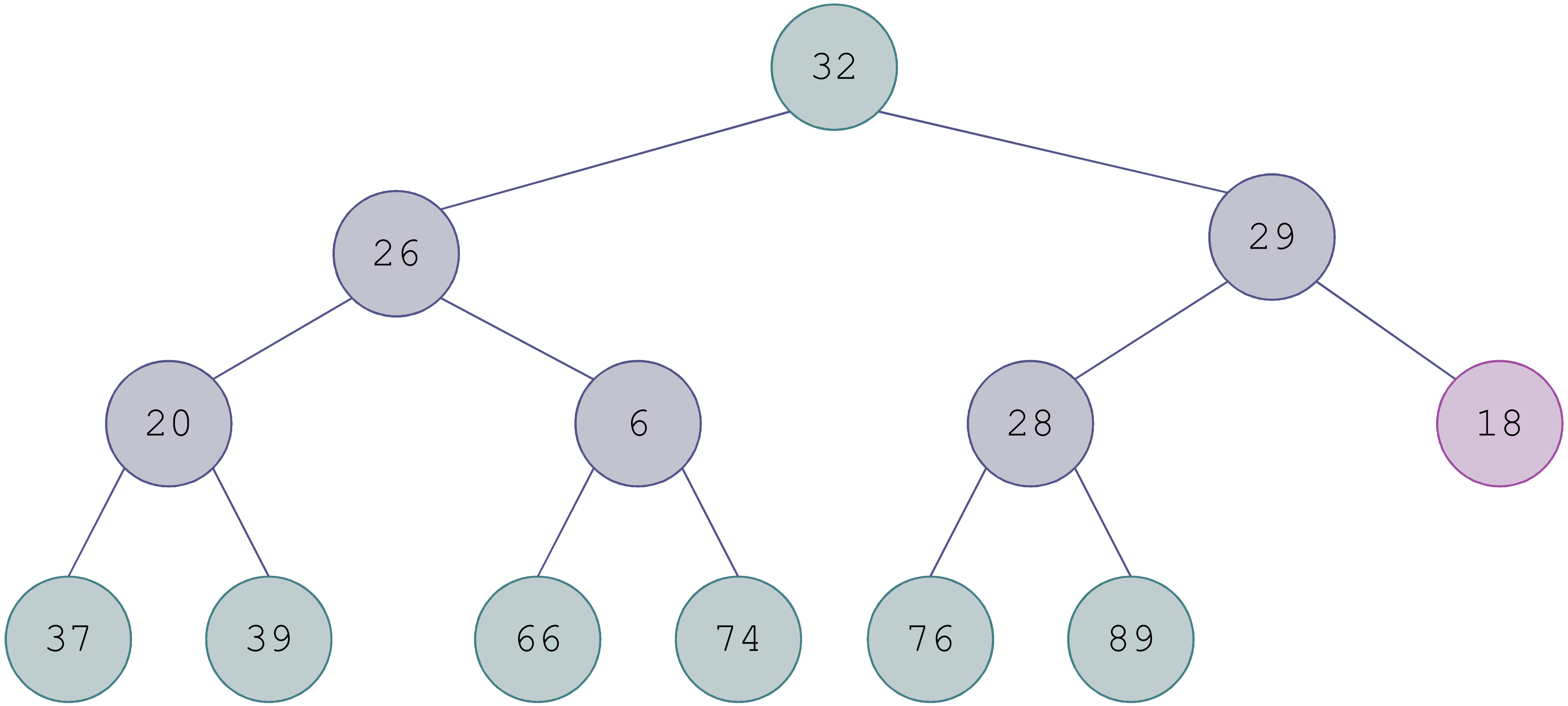
Trace of Heapsort (cont.)



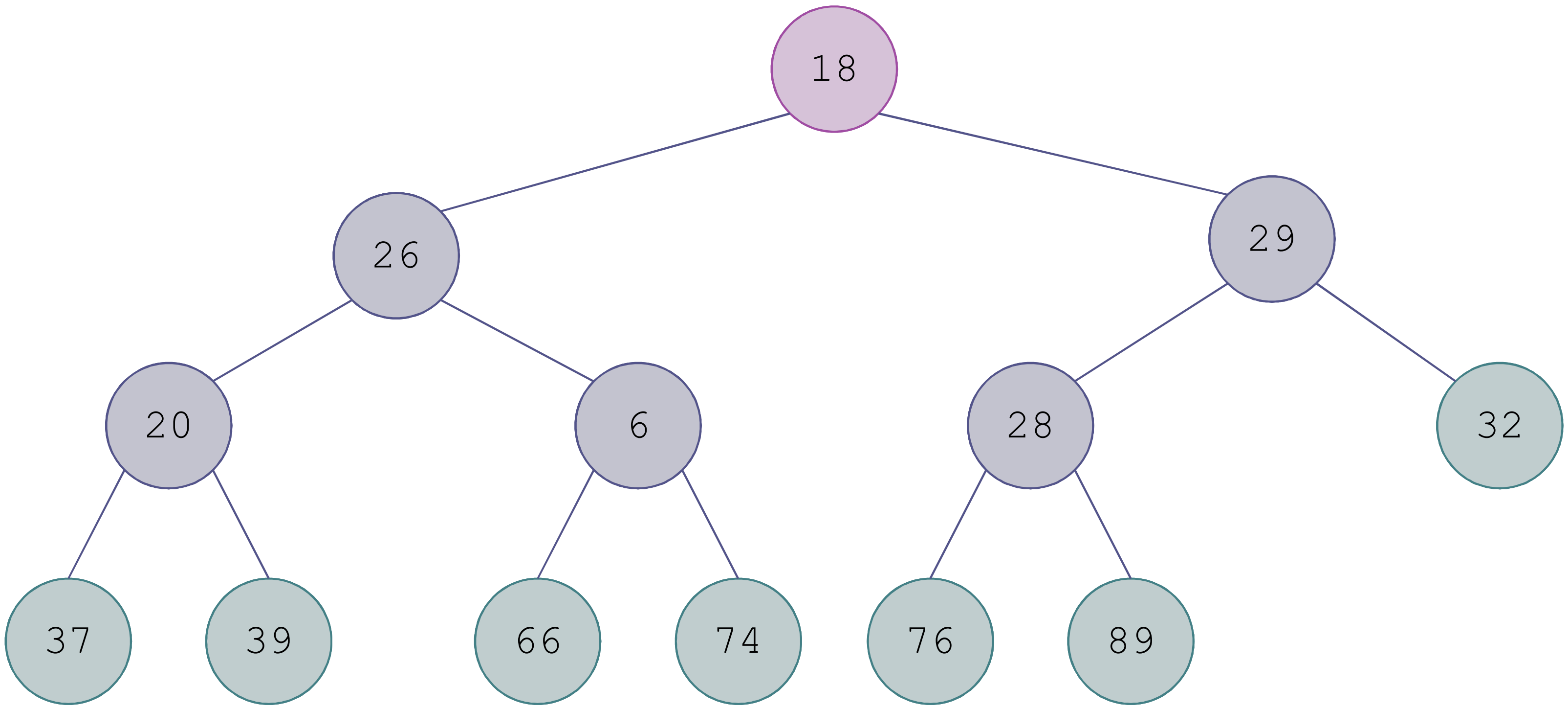
Trace of Heapsort (cont.)



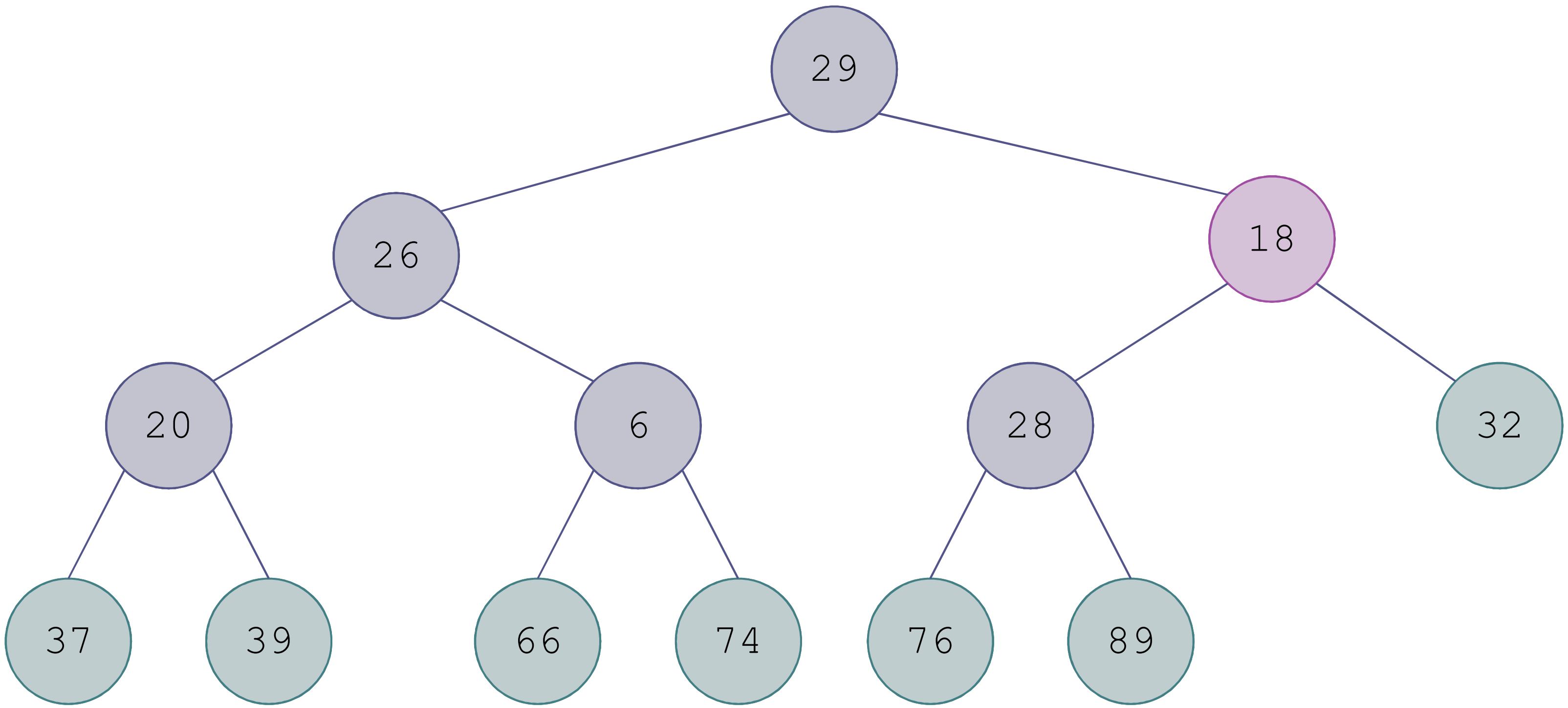
Trace of Heapsort (cont.)



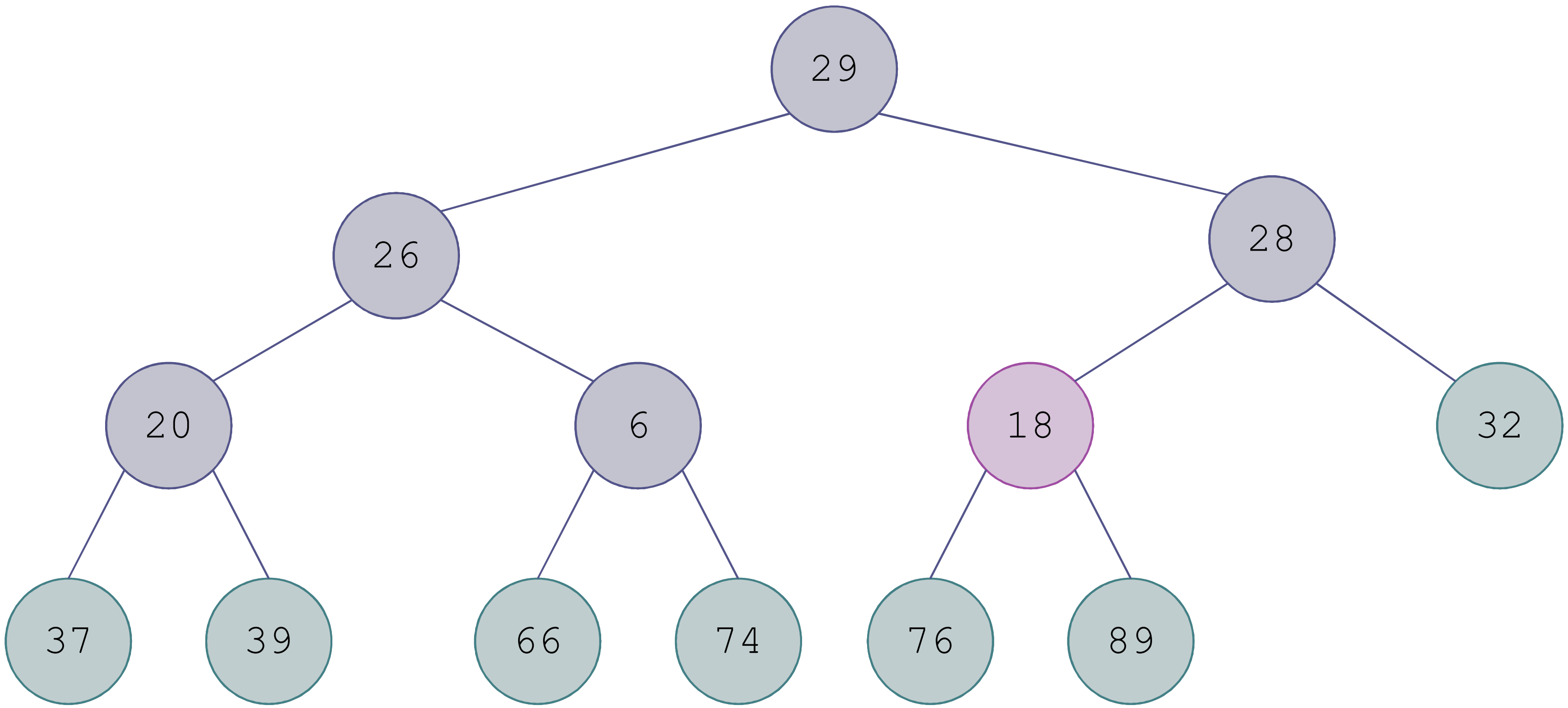
Trace of Heapsort (cont.)



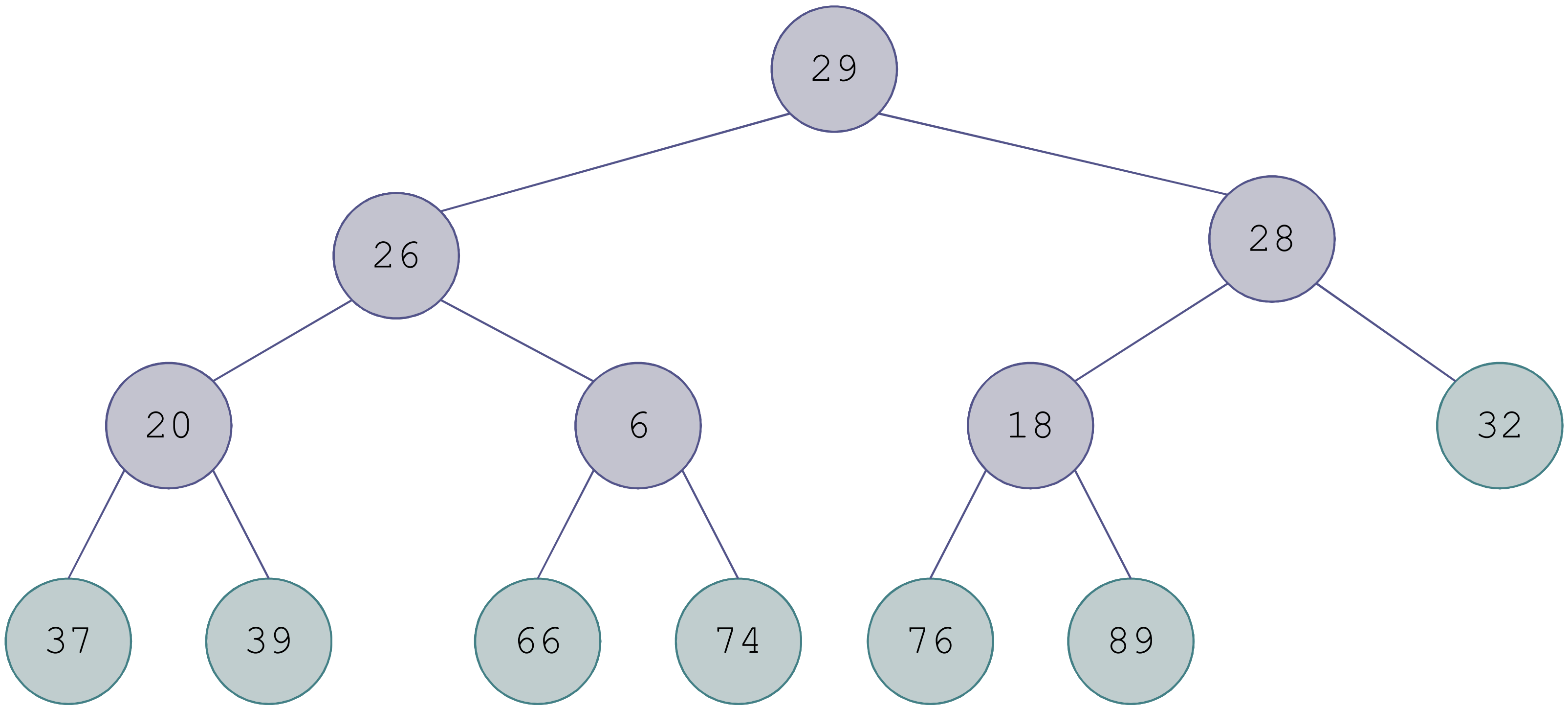
Trace of Heapsort (cont.)



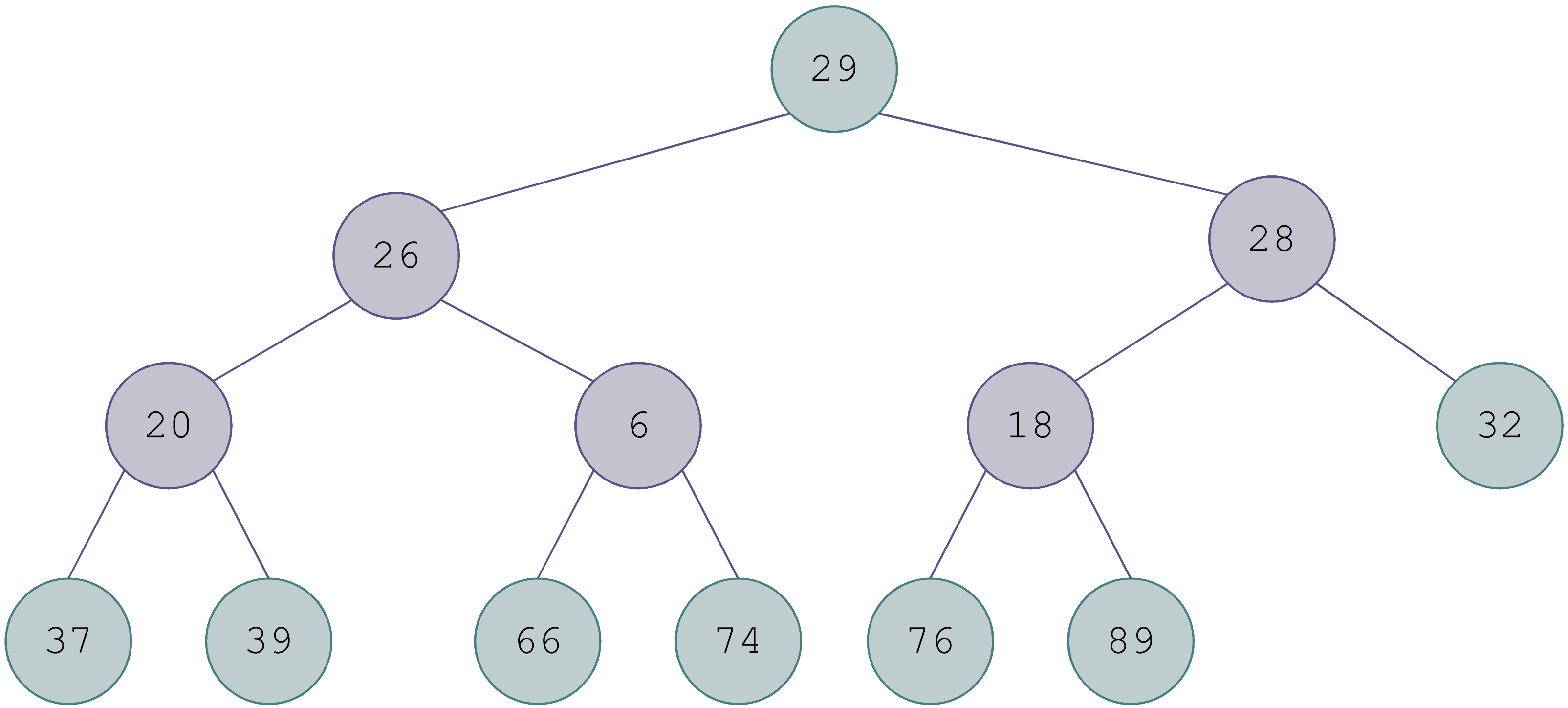
Trace of Heapsort (cont.)



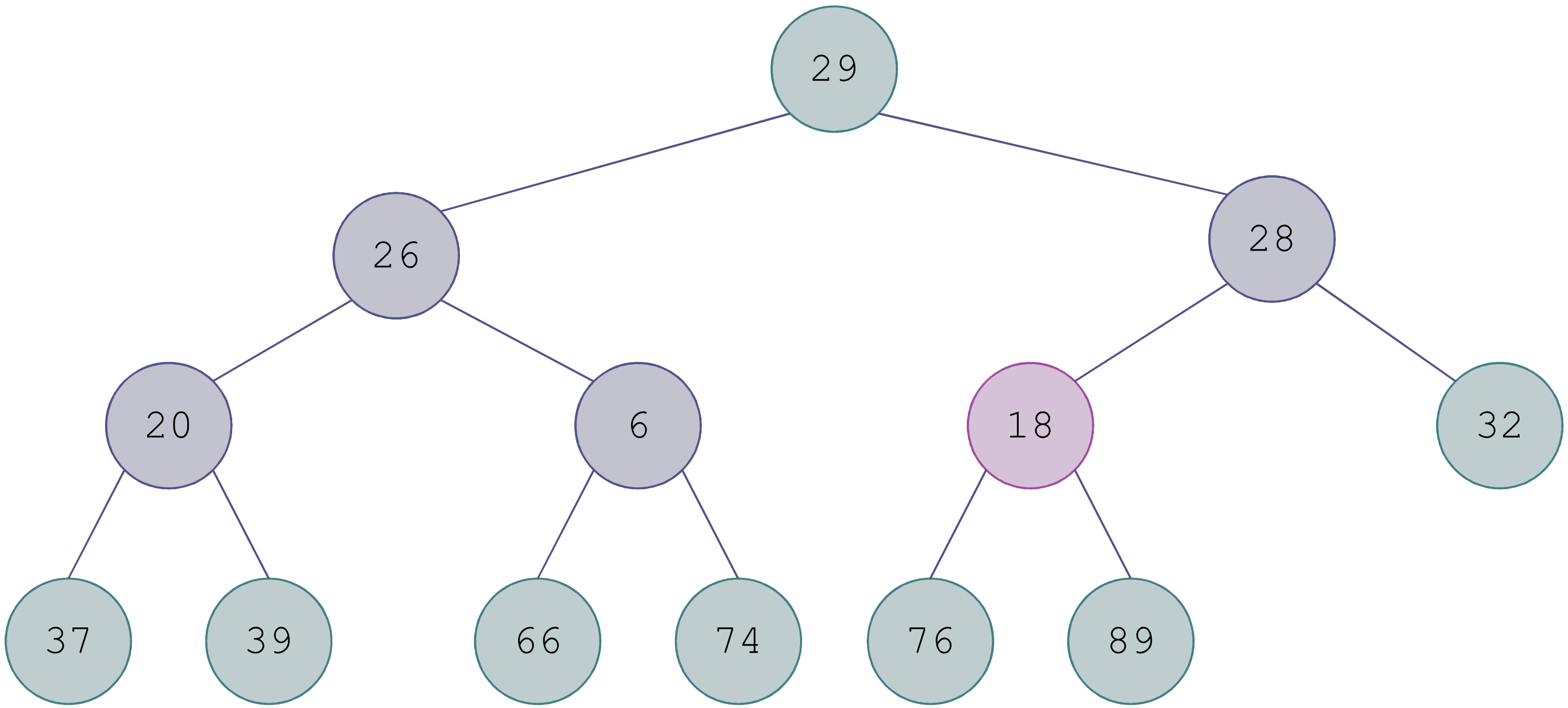
Trace of Heapsort (cont.)



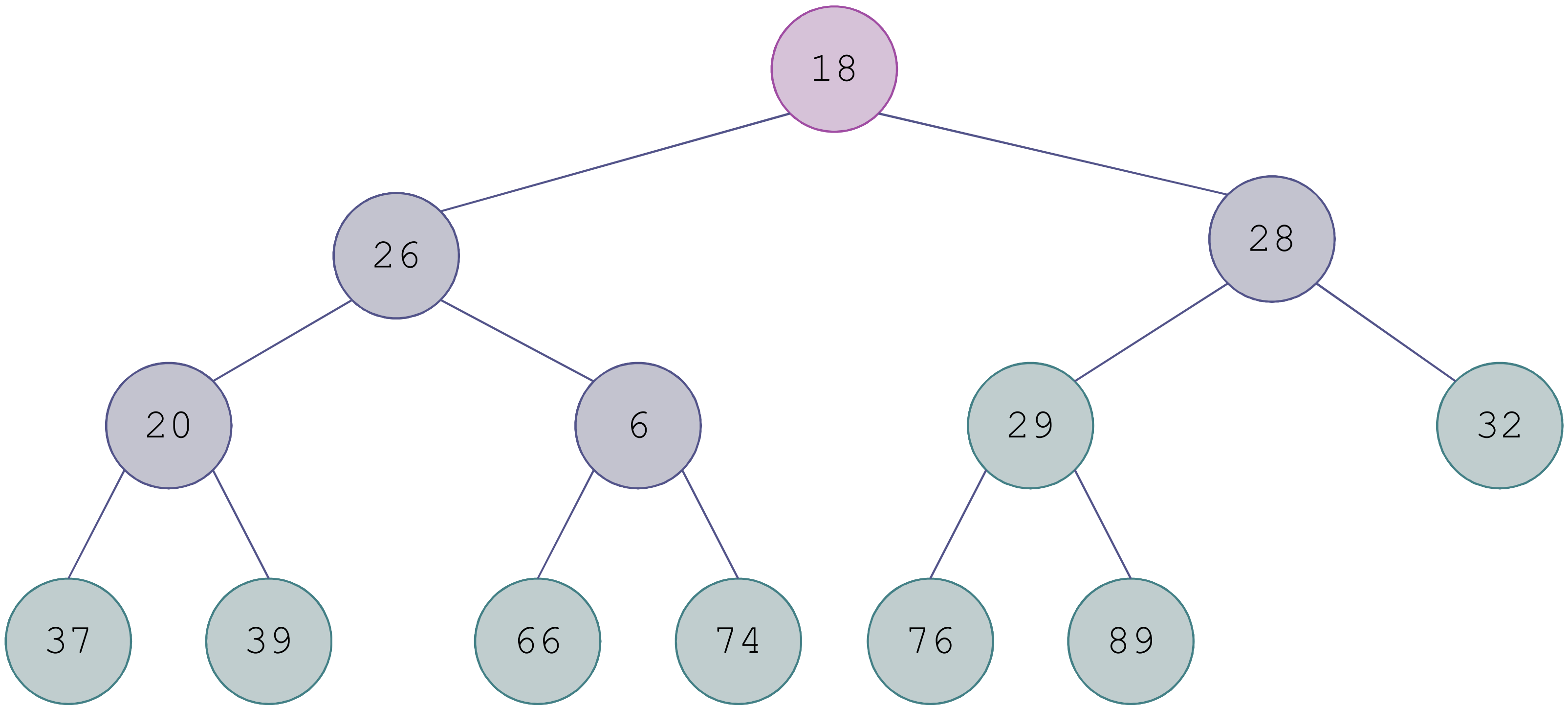
Trace of Heapsort (cont.)



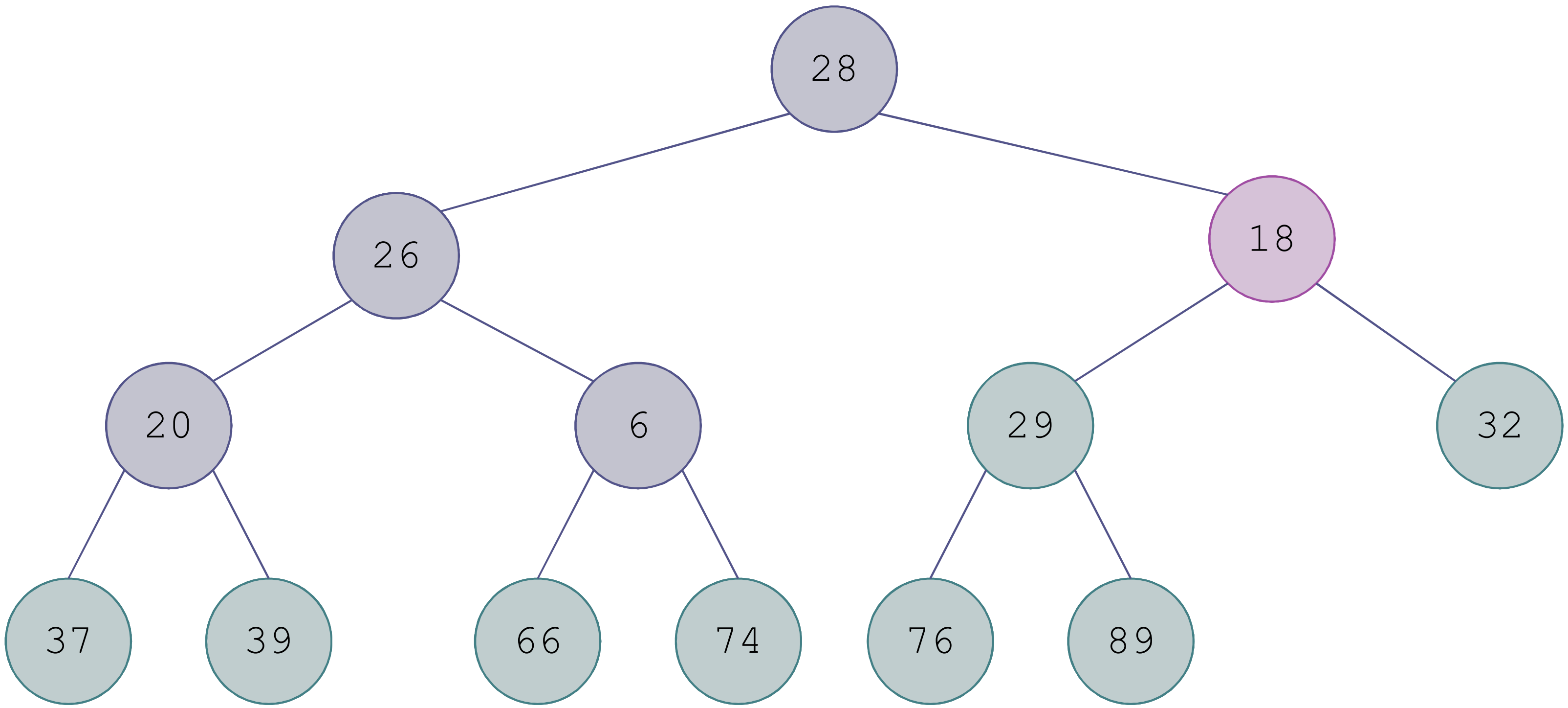
Trace of Heapsort (cont.)



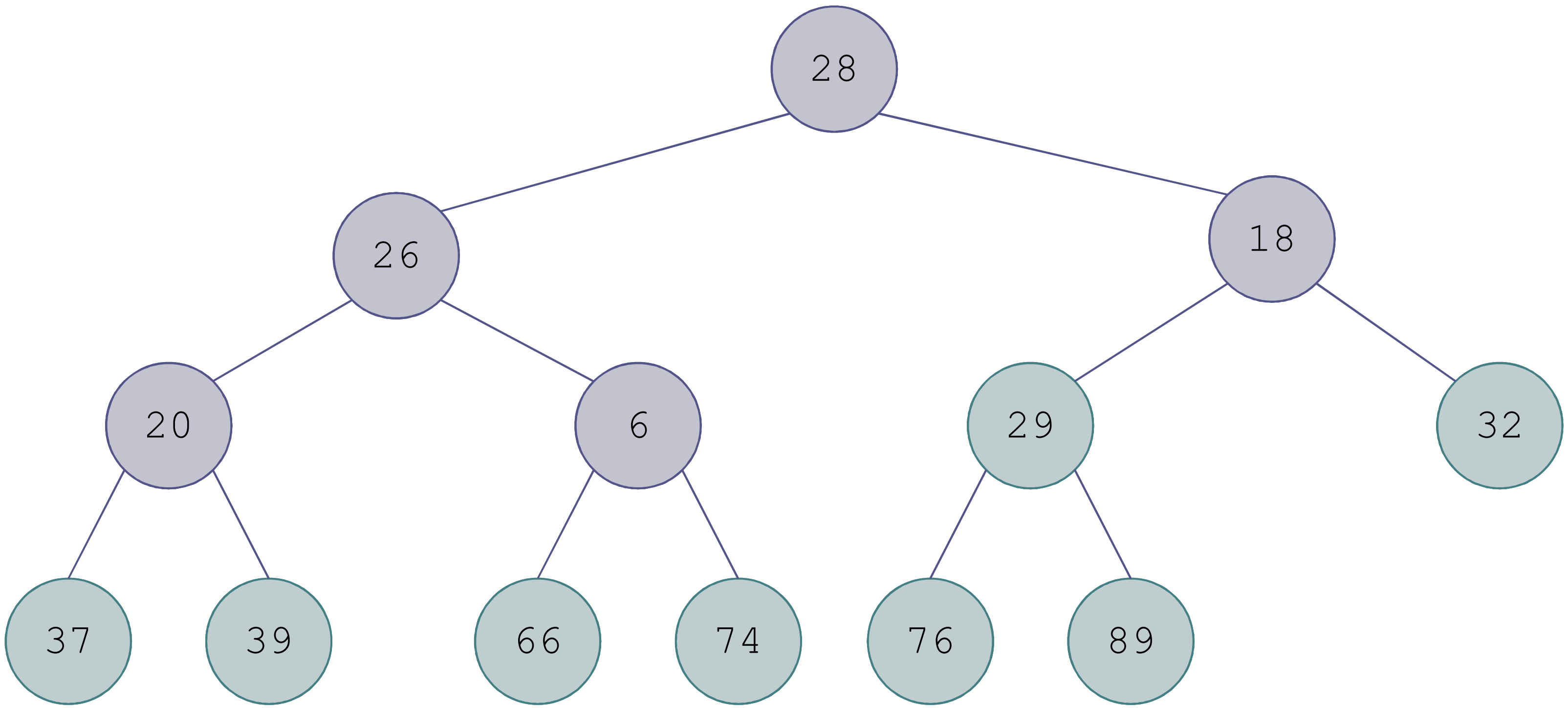
Trace of Heapsort (cont.)



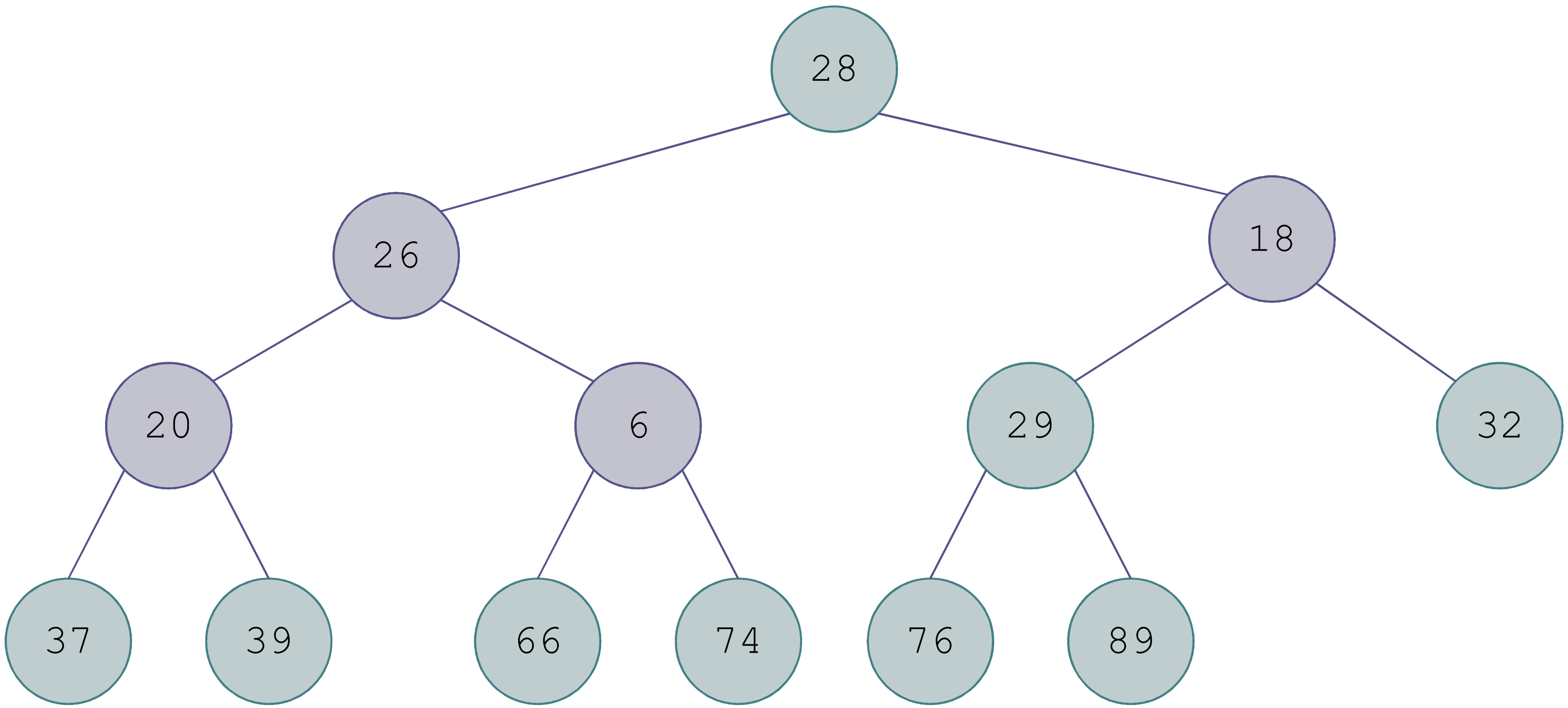
Trace of Heapsort (cont.)



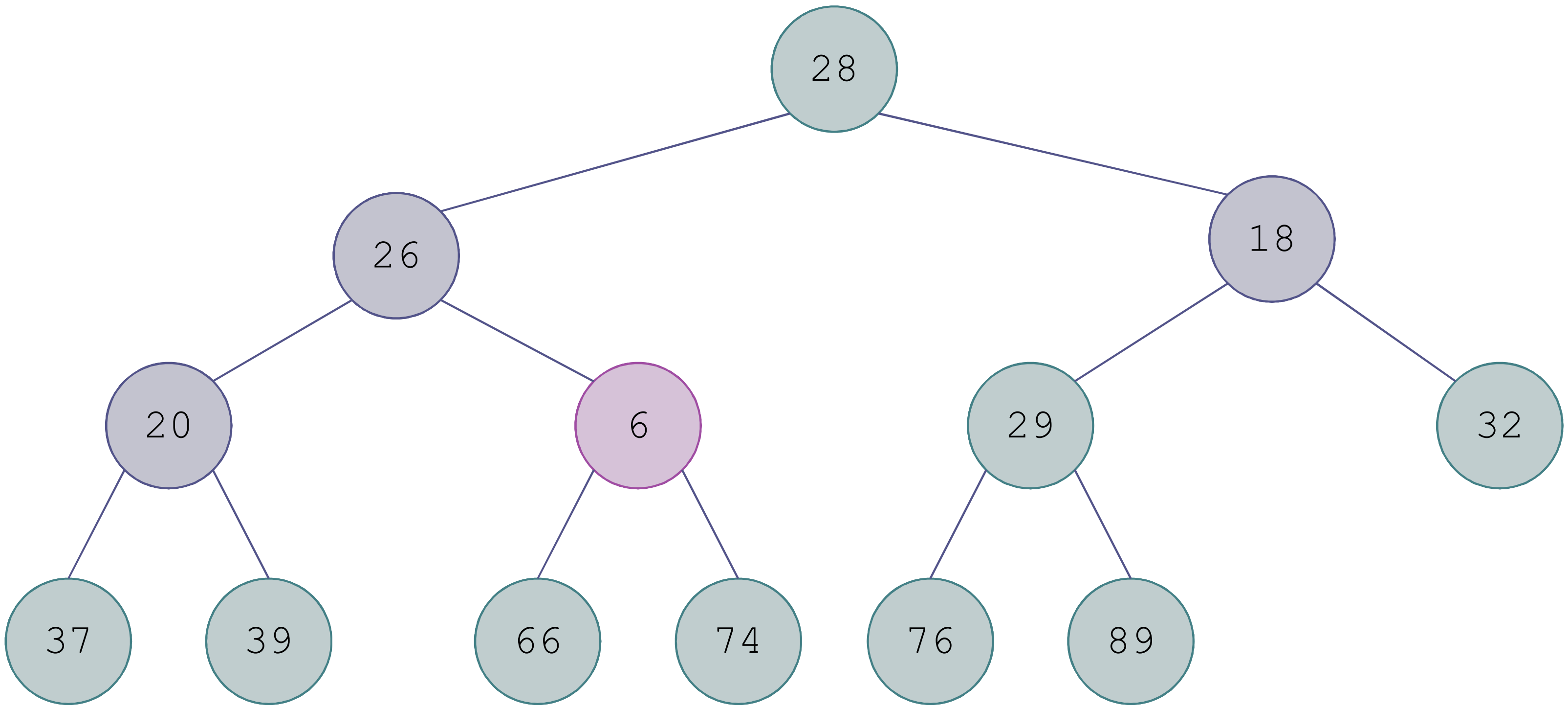
Trace of Heapsort (cont.)



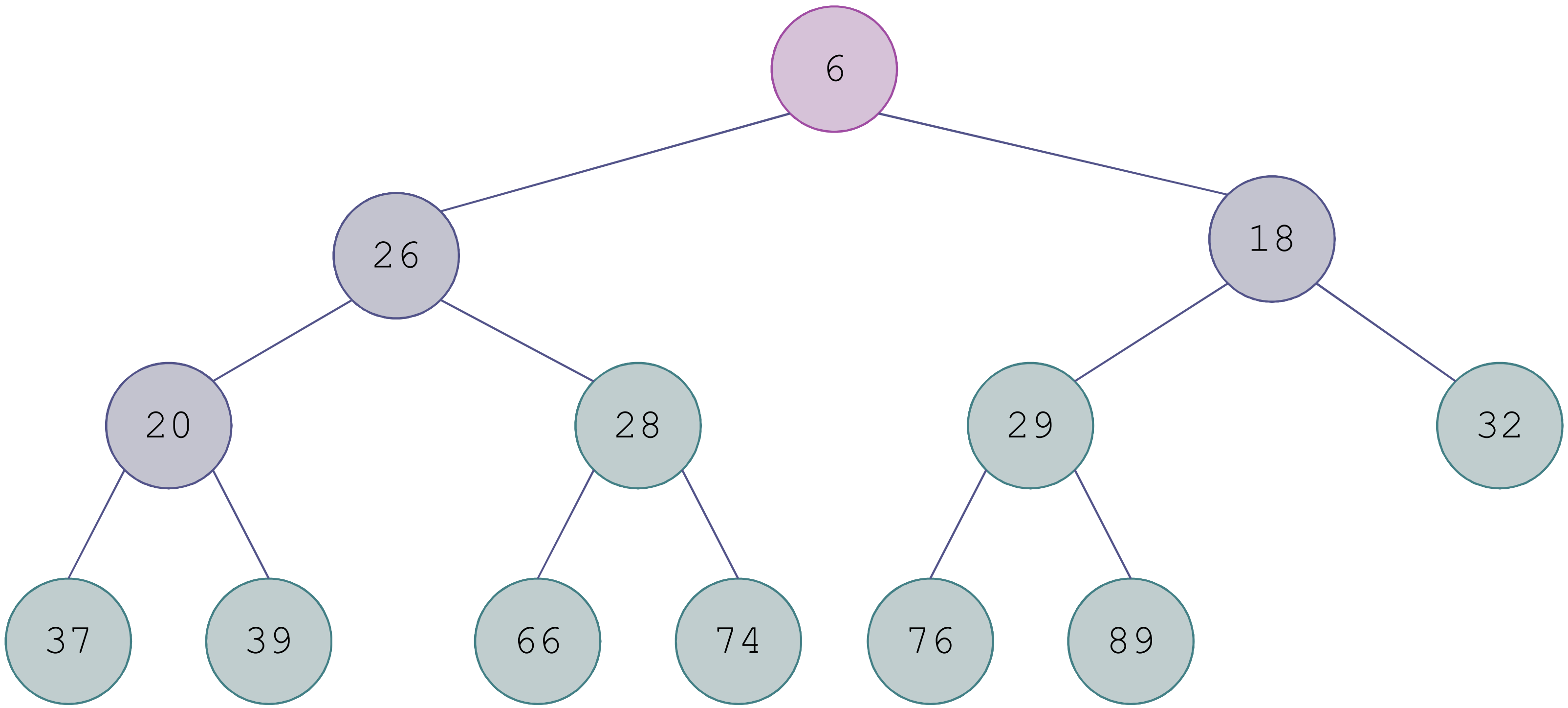
Trace of Heapsort (cont.)



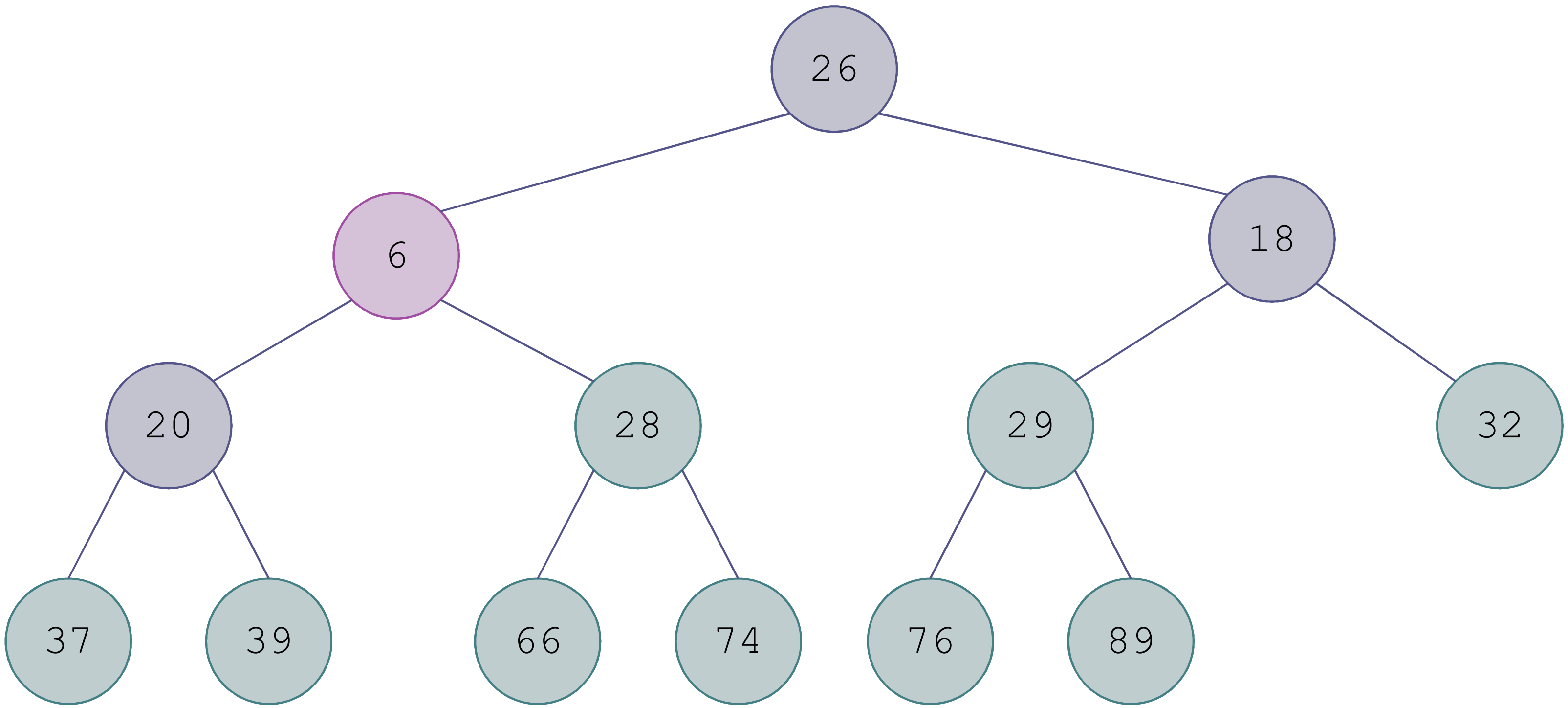
Trace of Heapsort (cont.)



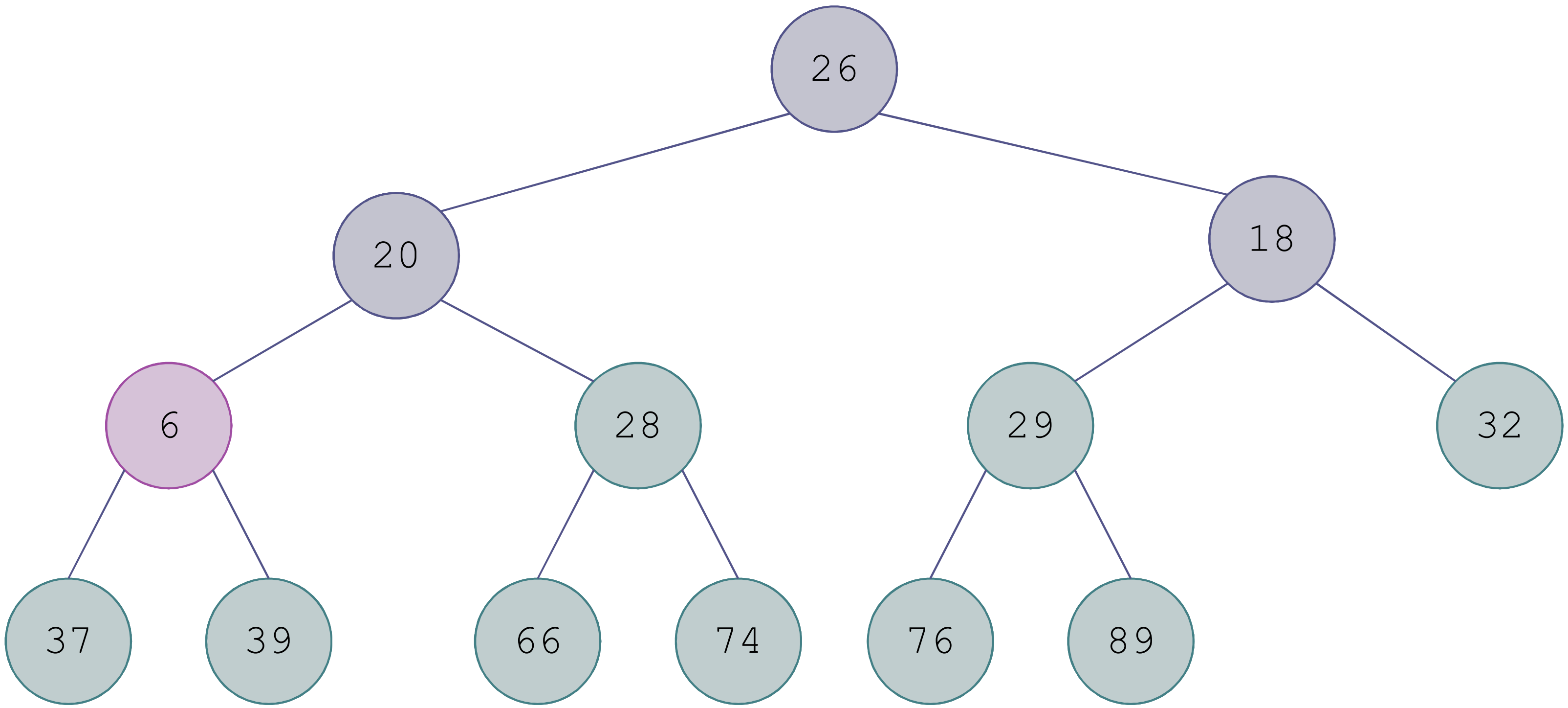
Trace of Heapsort (cont.)



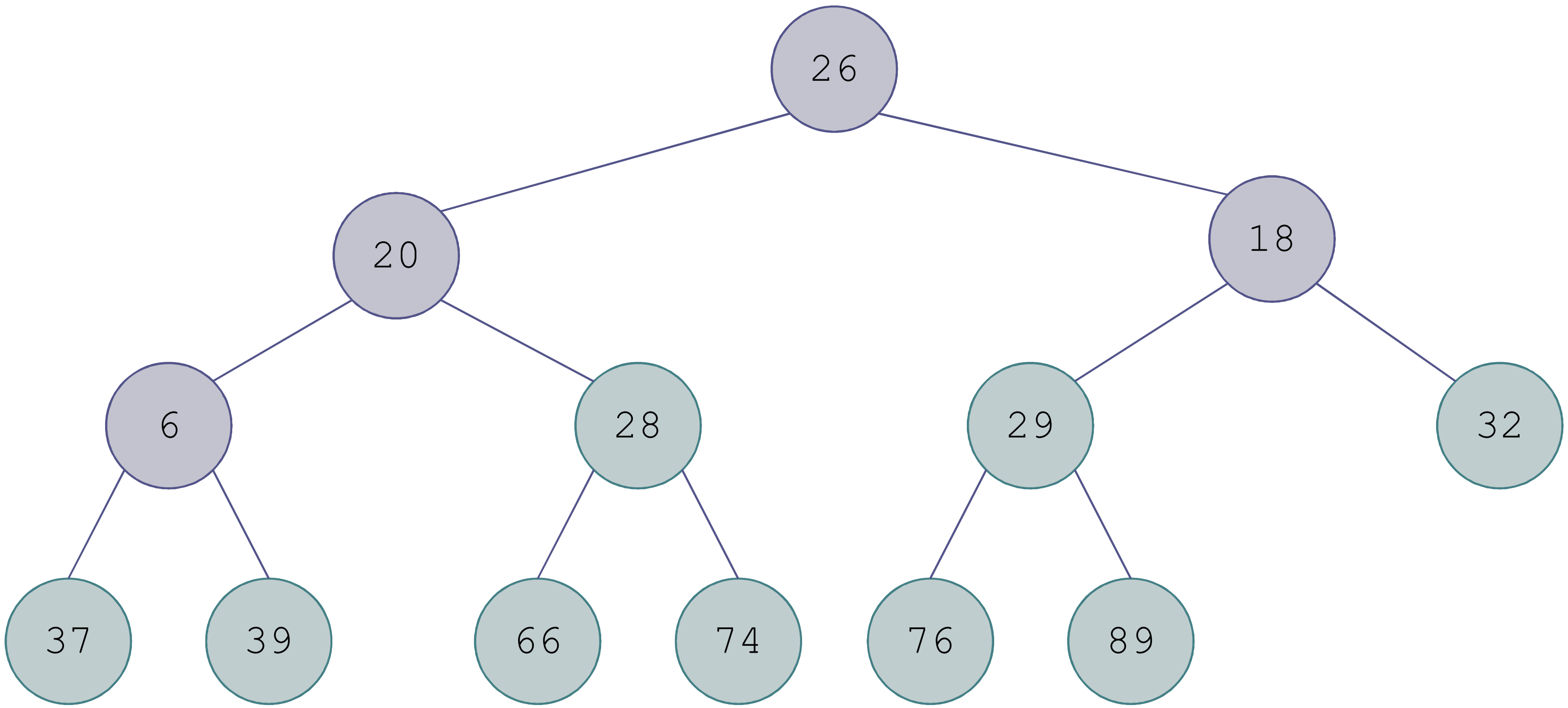
Trace of Heapsort (cont.)



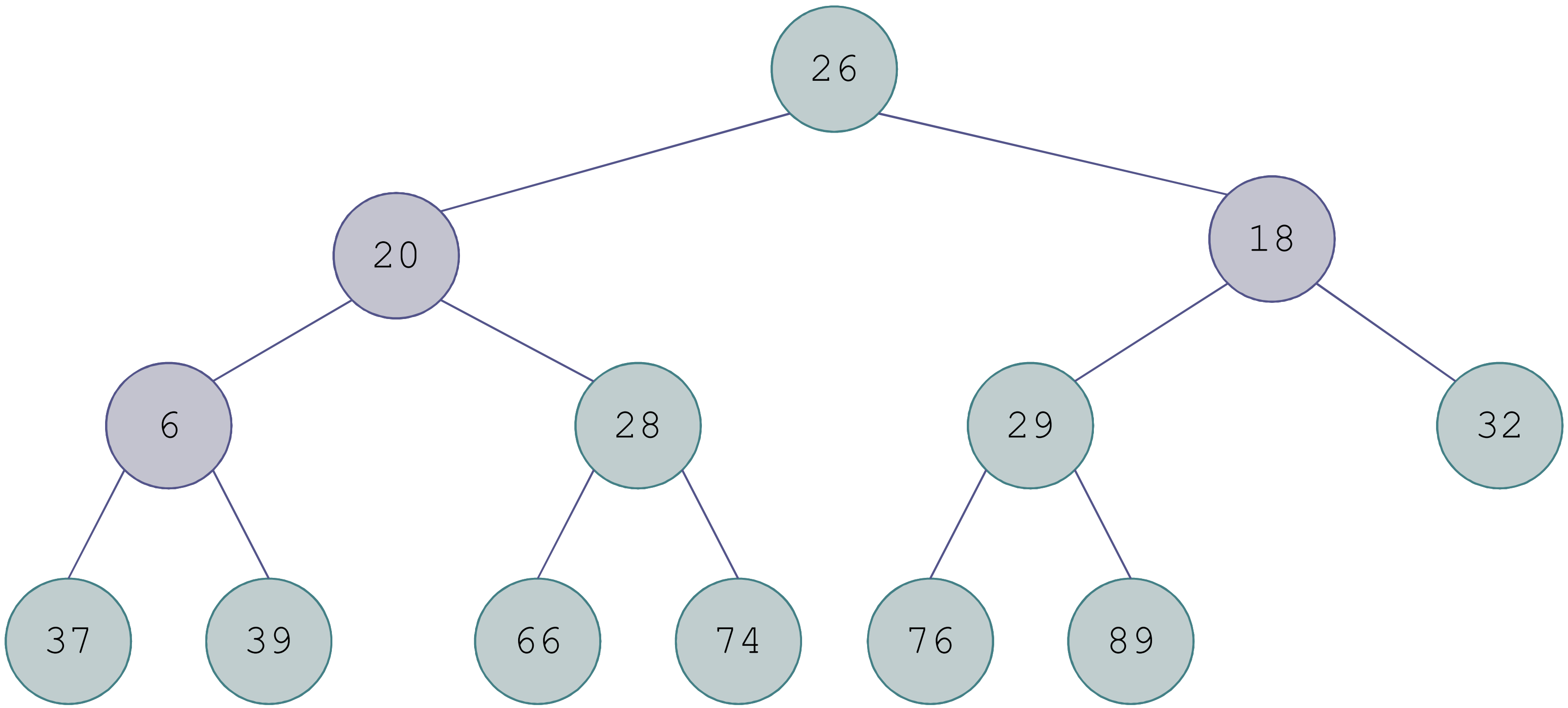
Trace of Heapsort (cont.)



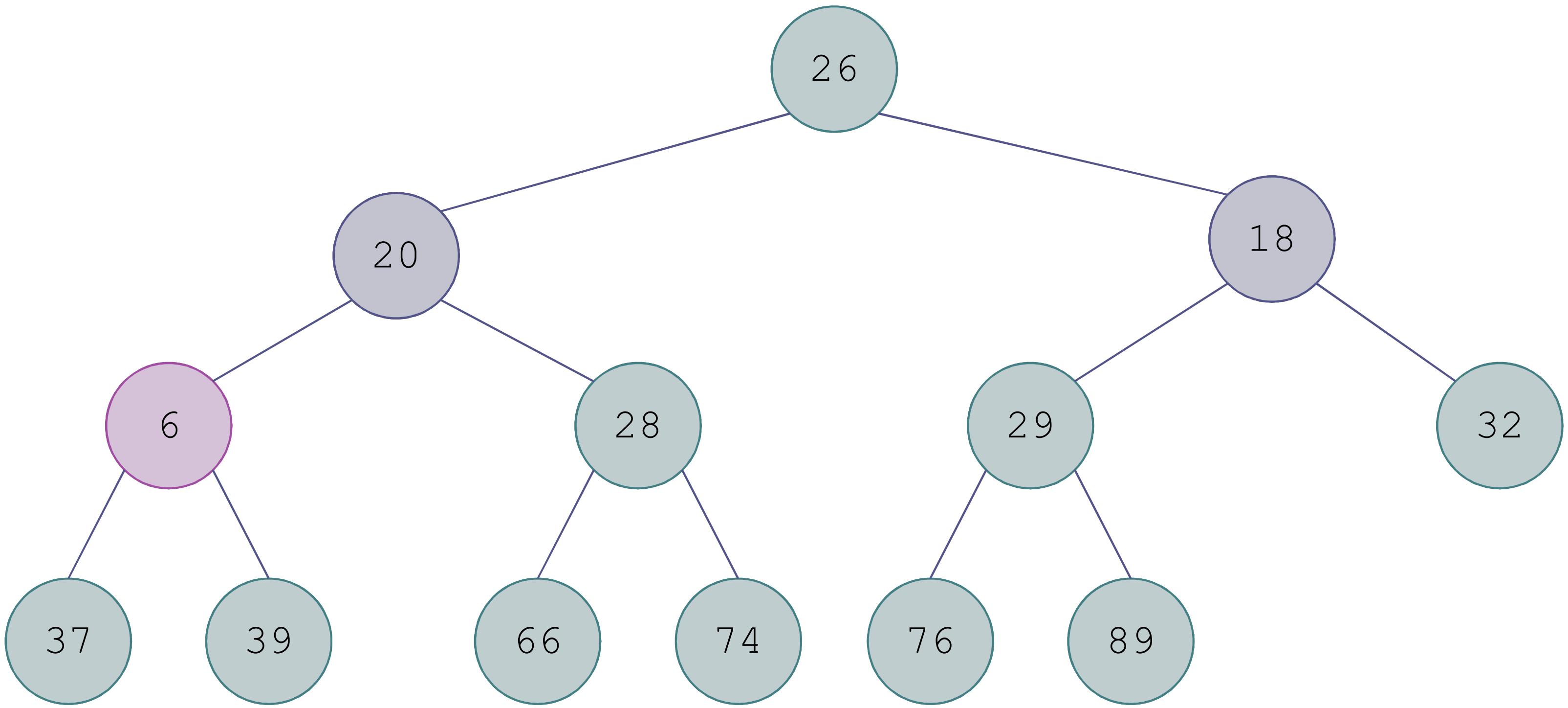
Trace of Heapsort (cont.)



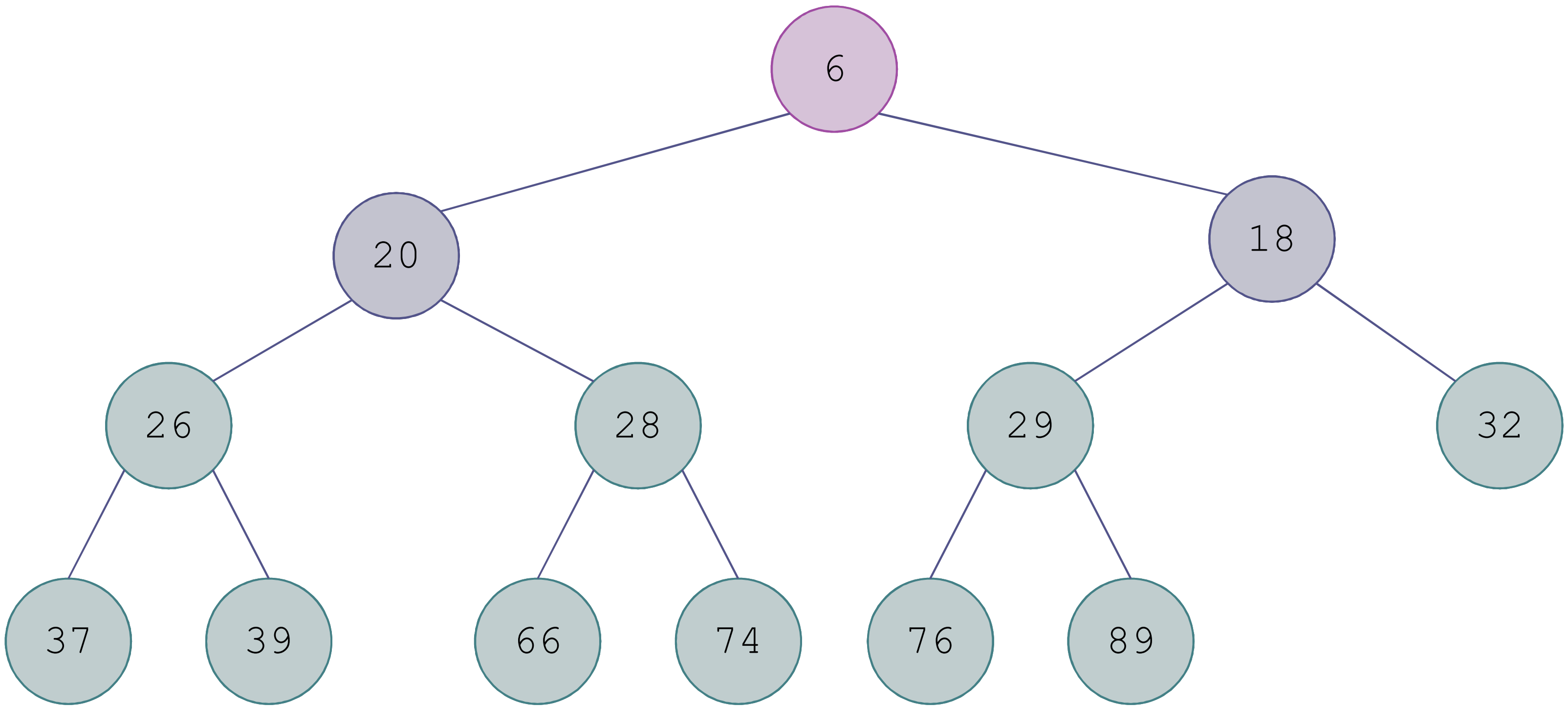
Trace of Heapsort (cont.)



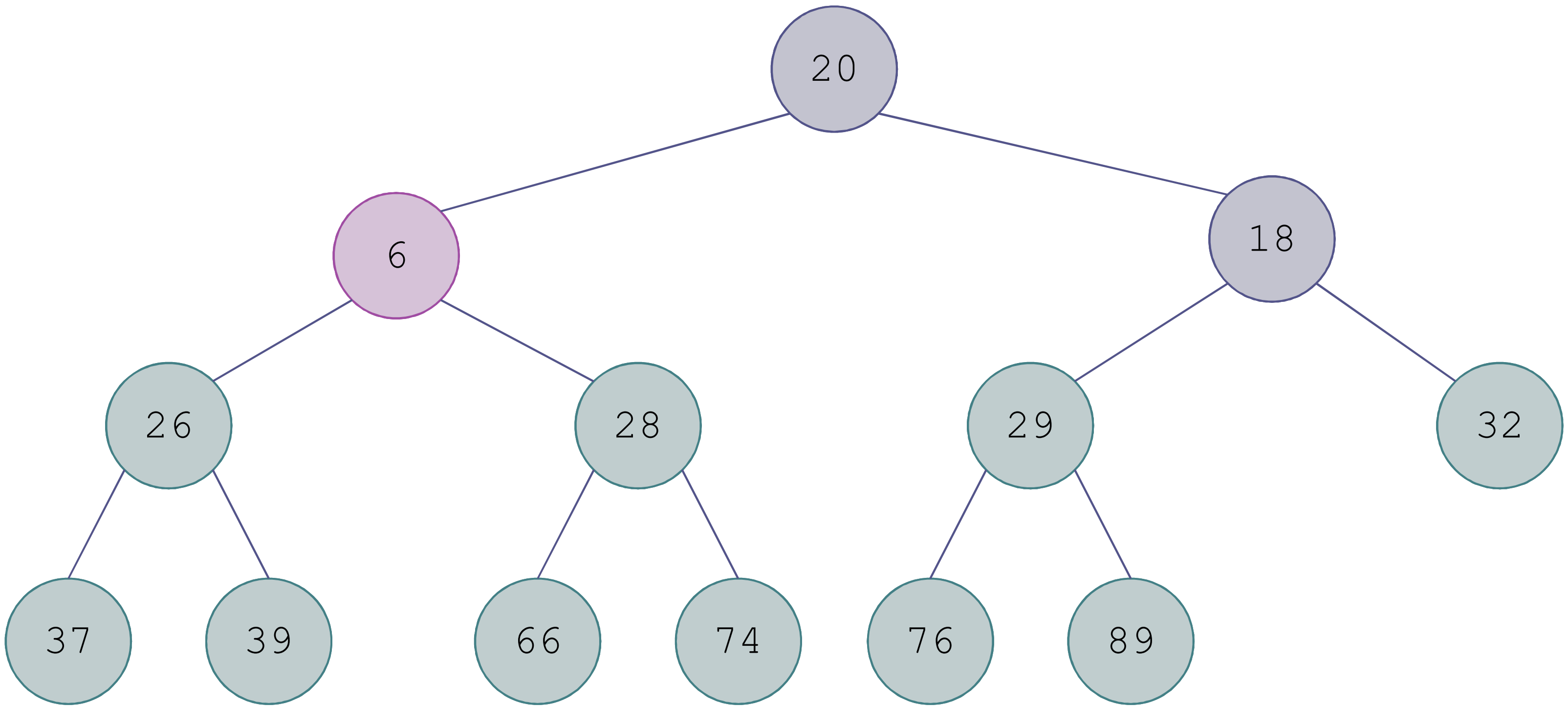
Trace of Heapsort (cont.)



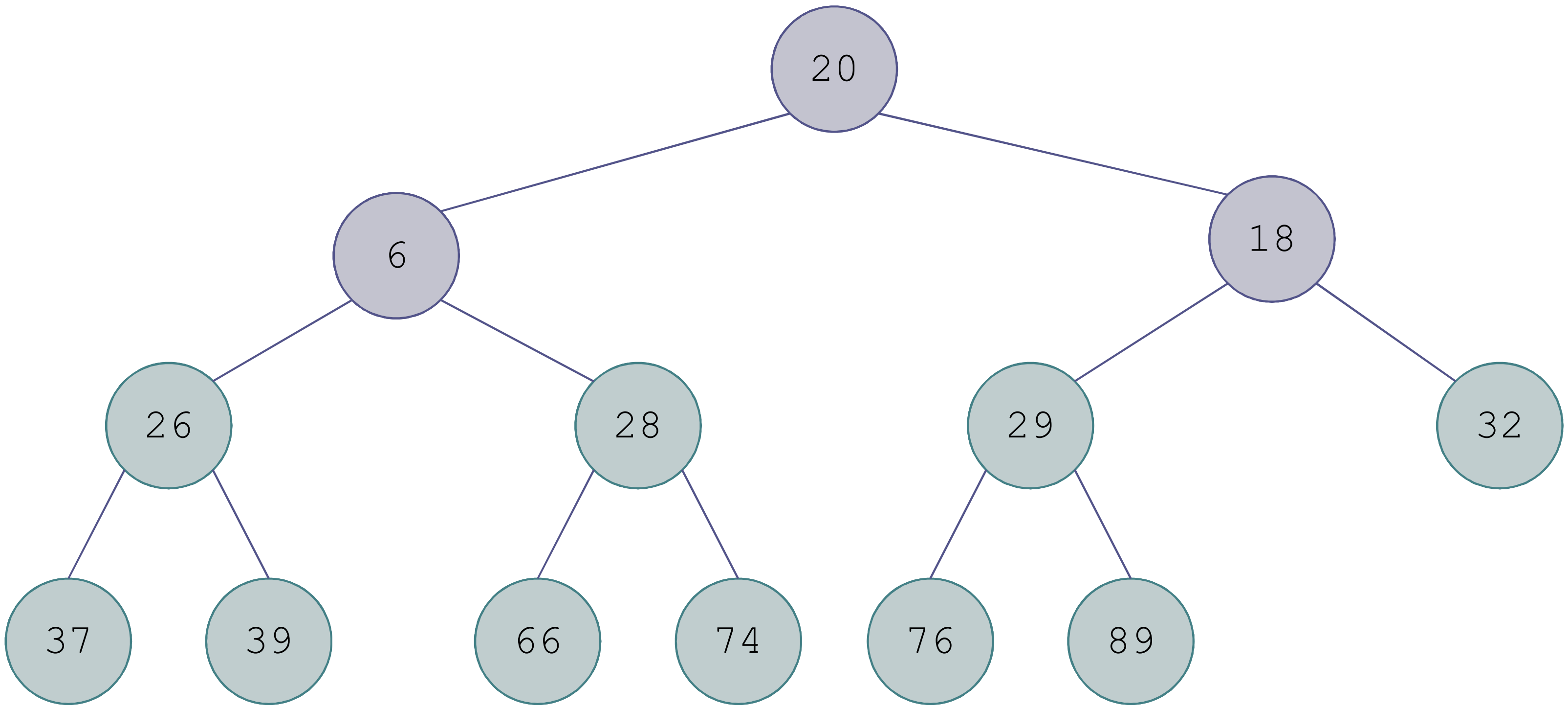
Trace of Heapsort (cont.)



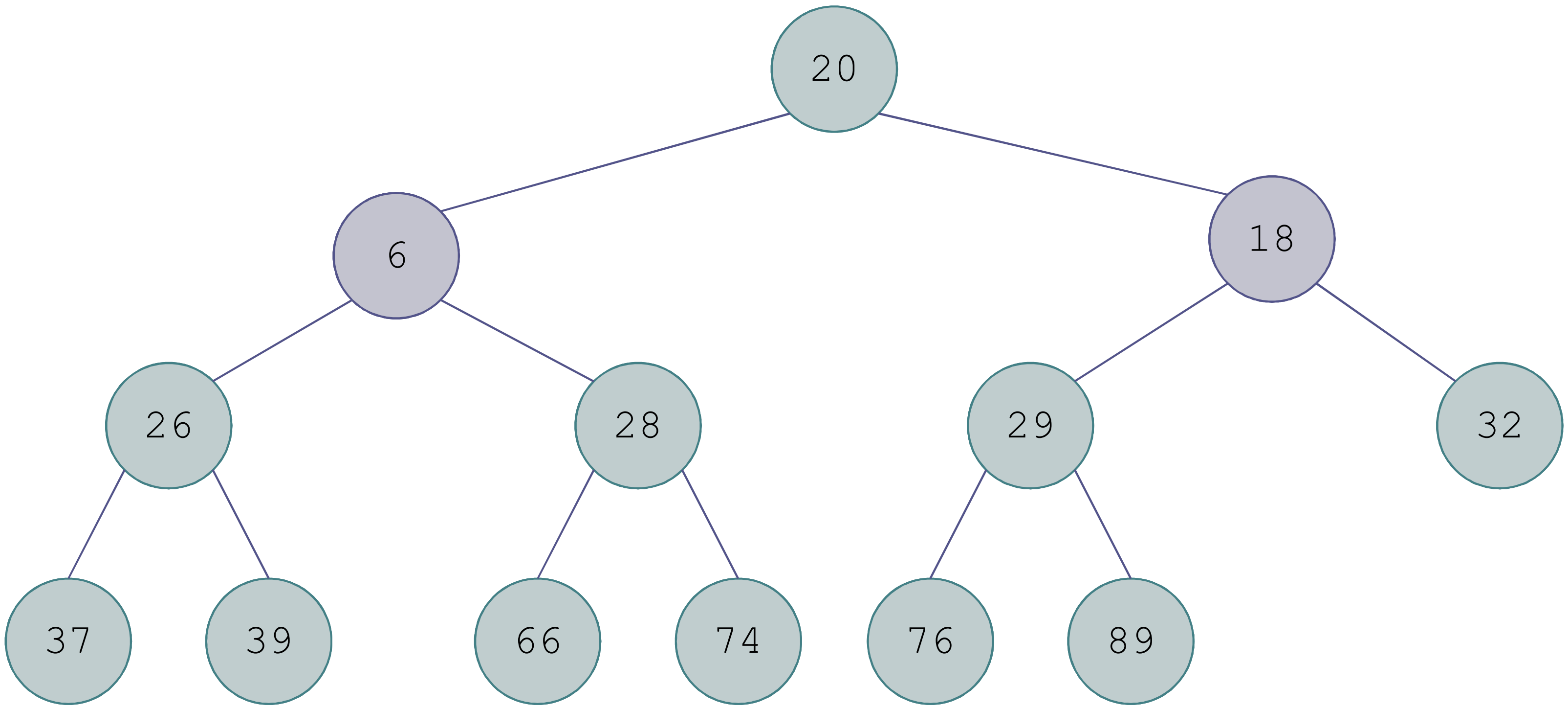
Trace of Heapsort (cont.)



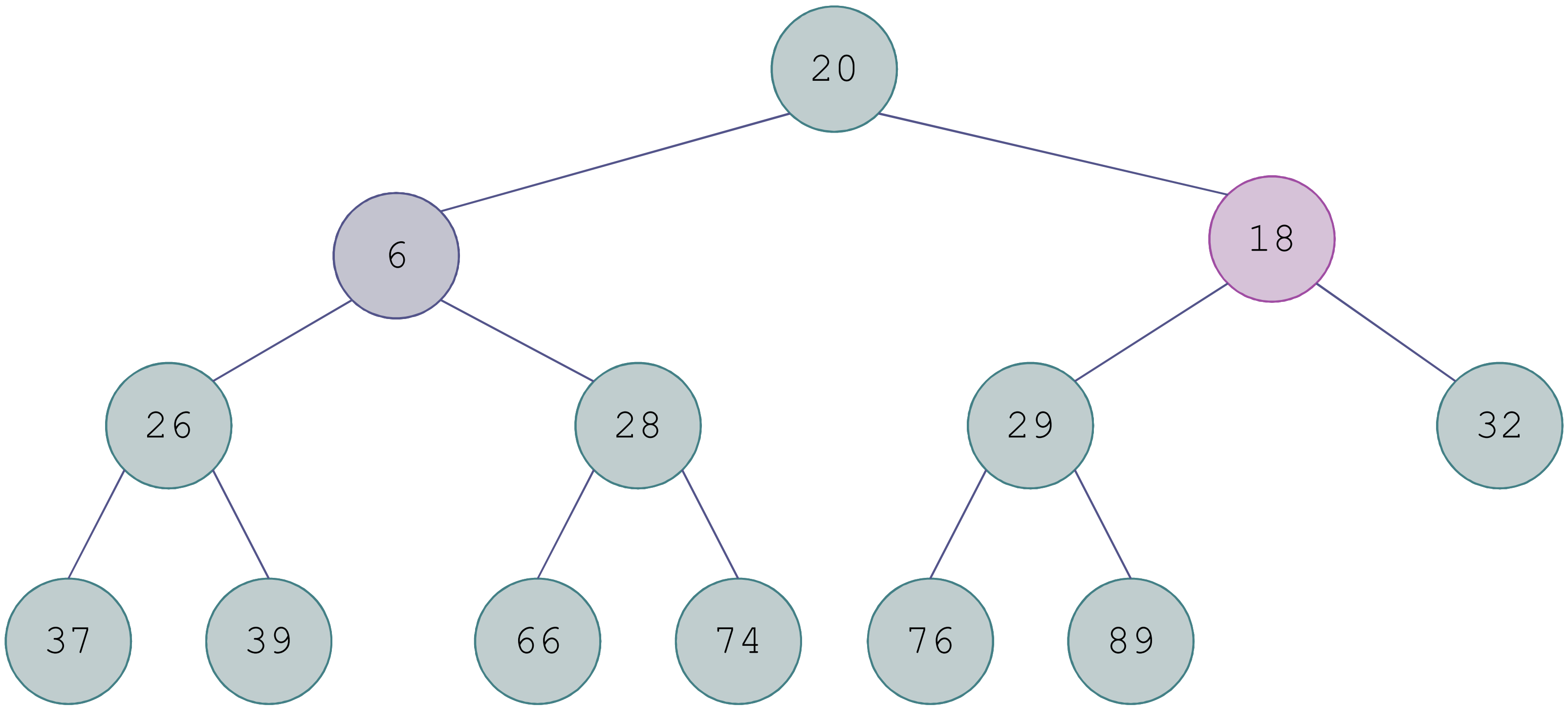
Trace of Heapsort (cont.)



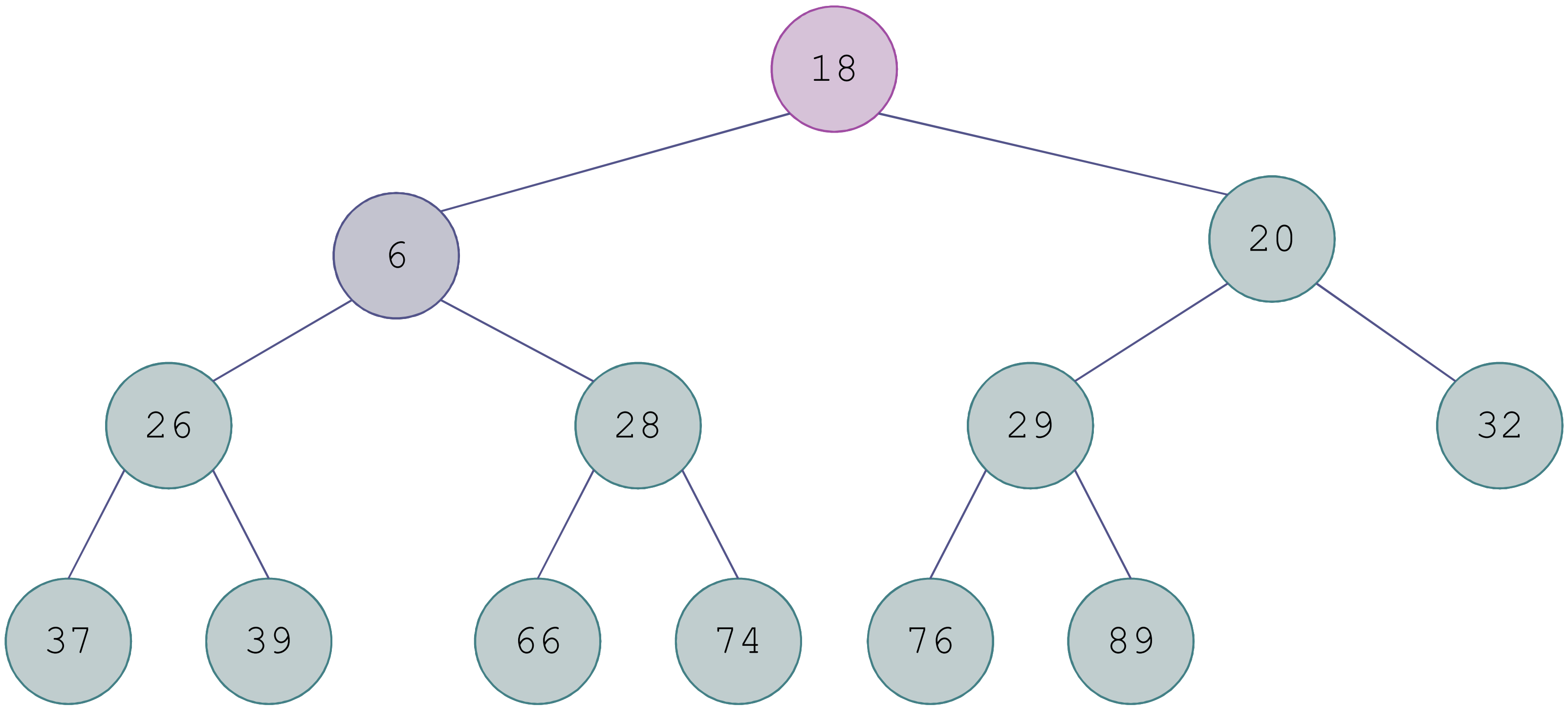
Trace of Heapsort (cont.)



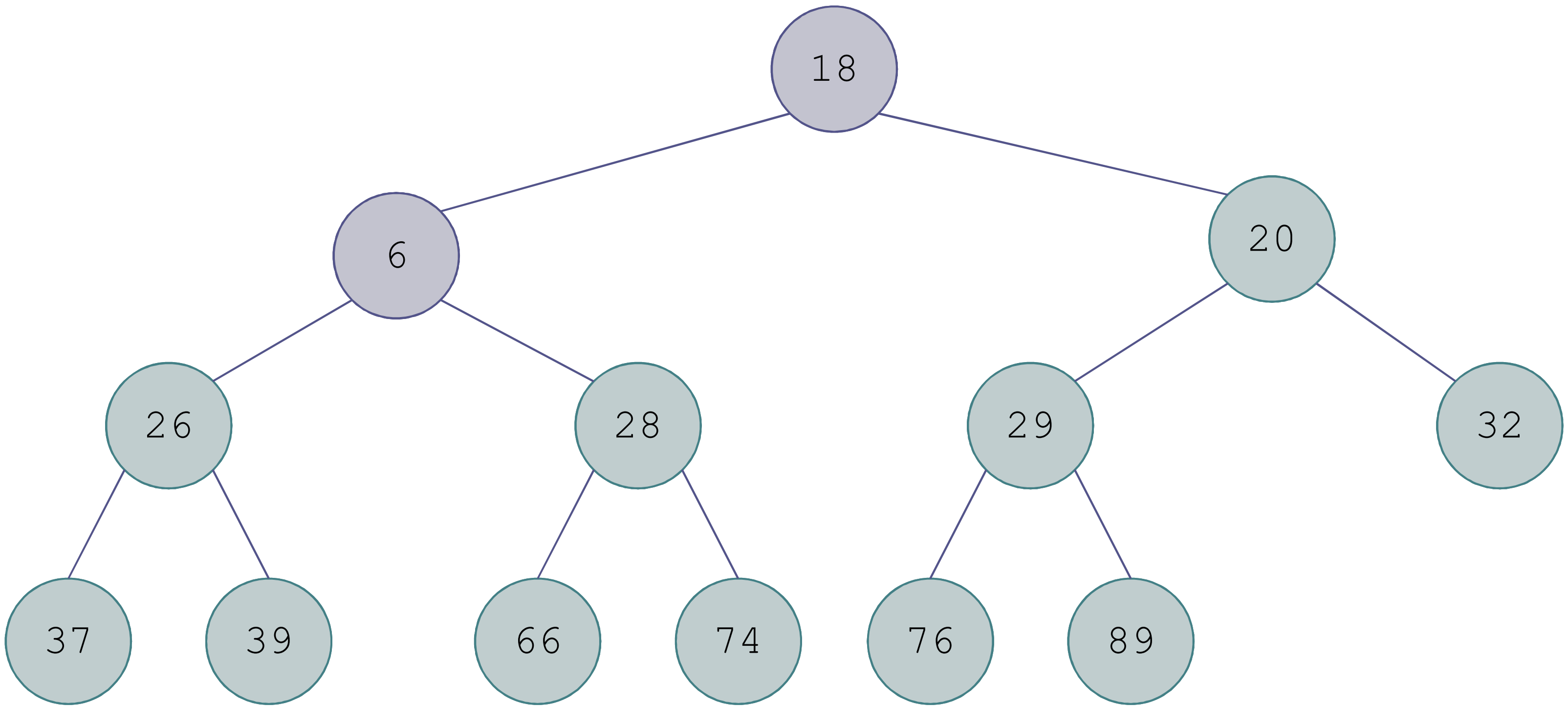
Trace of Heapsort (cont.)



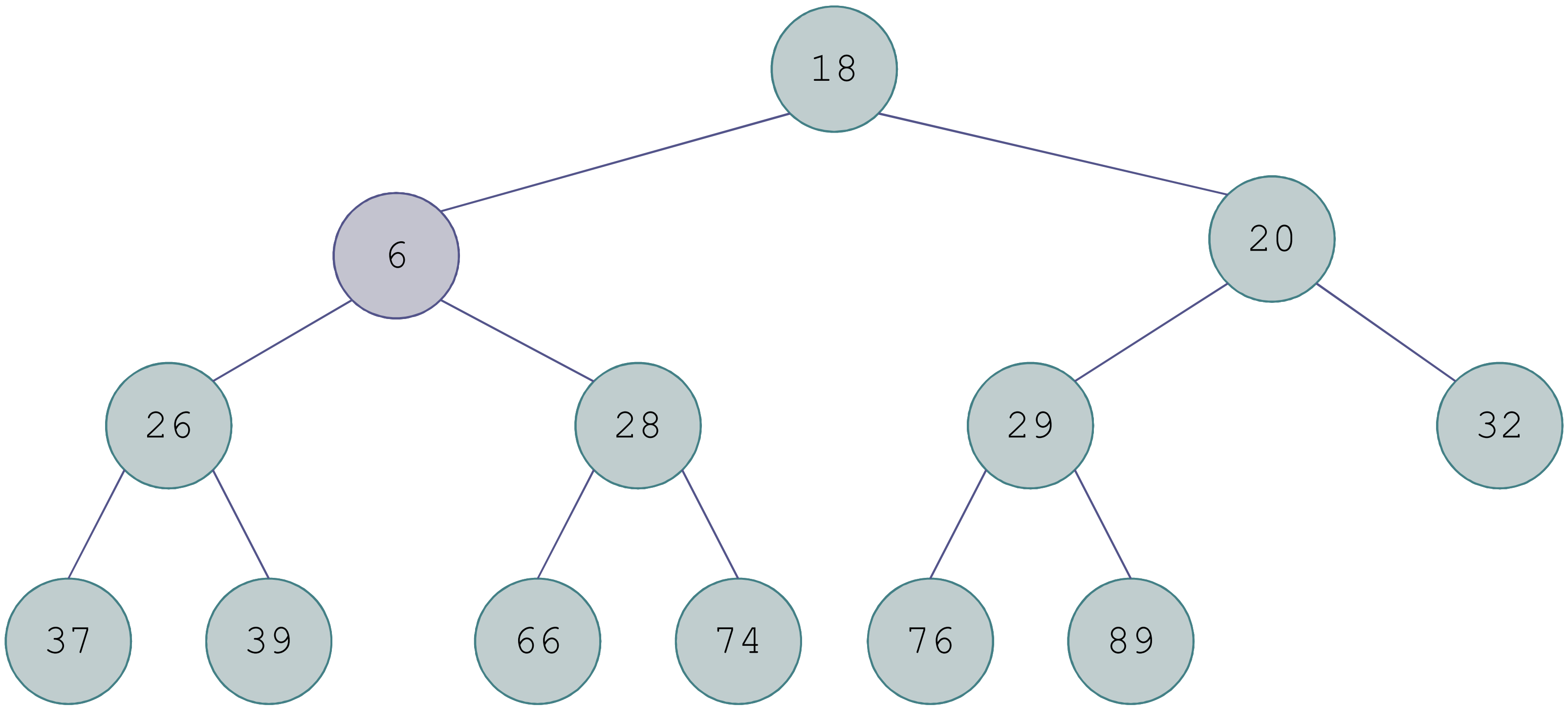
Trace of Heapsort (cont.)



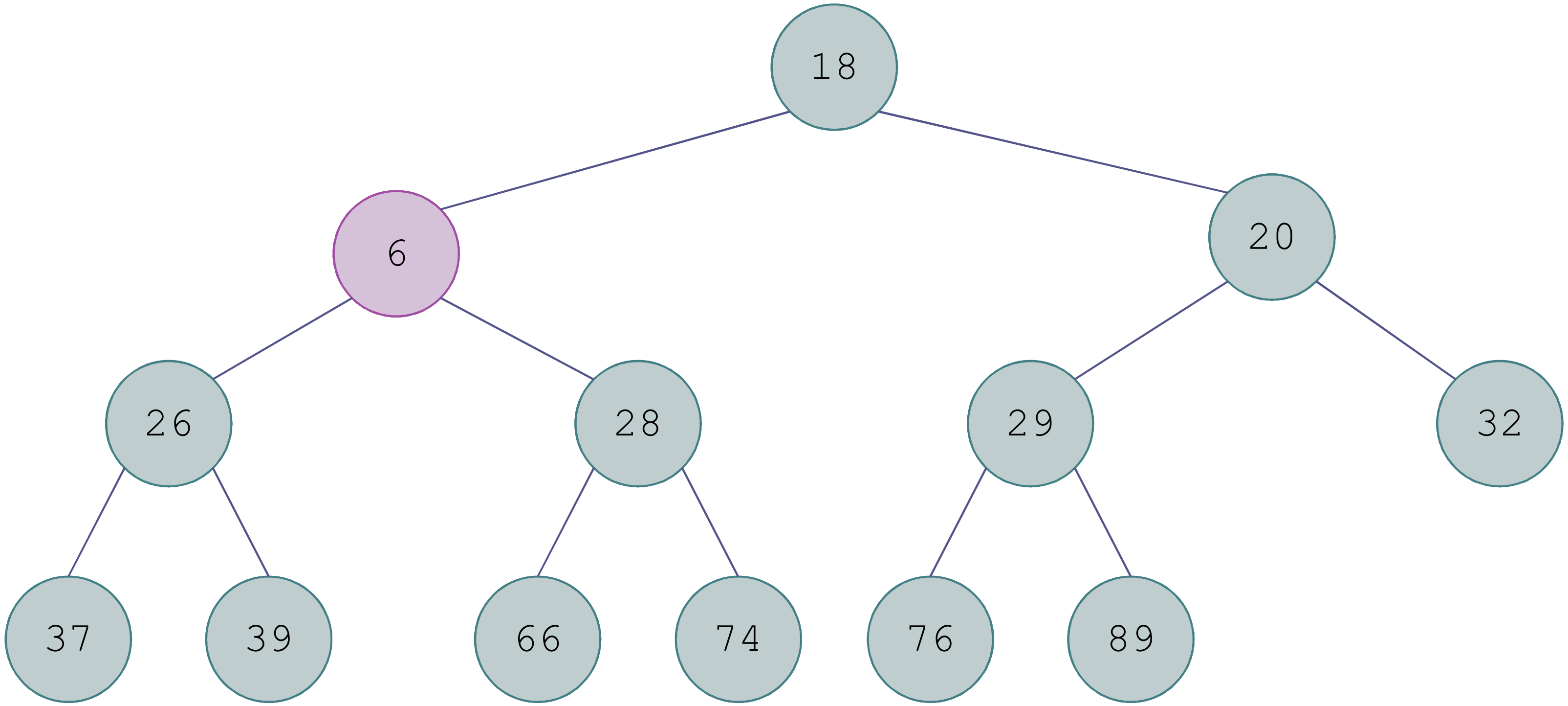
Trace of Heapsort (cont.)



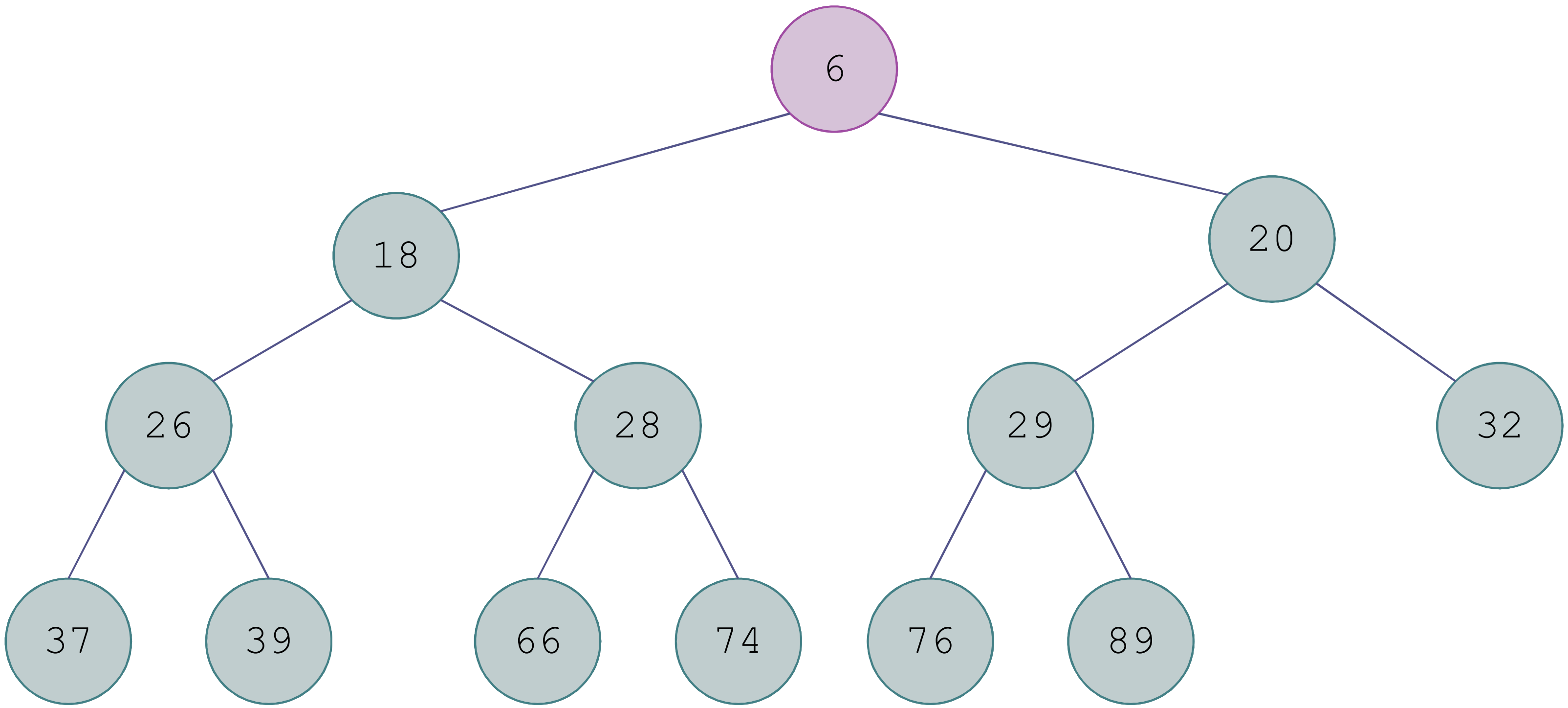
Trace of Heapsort (cont.)



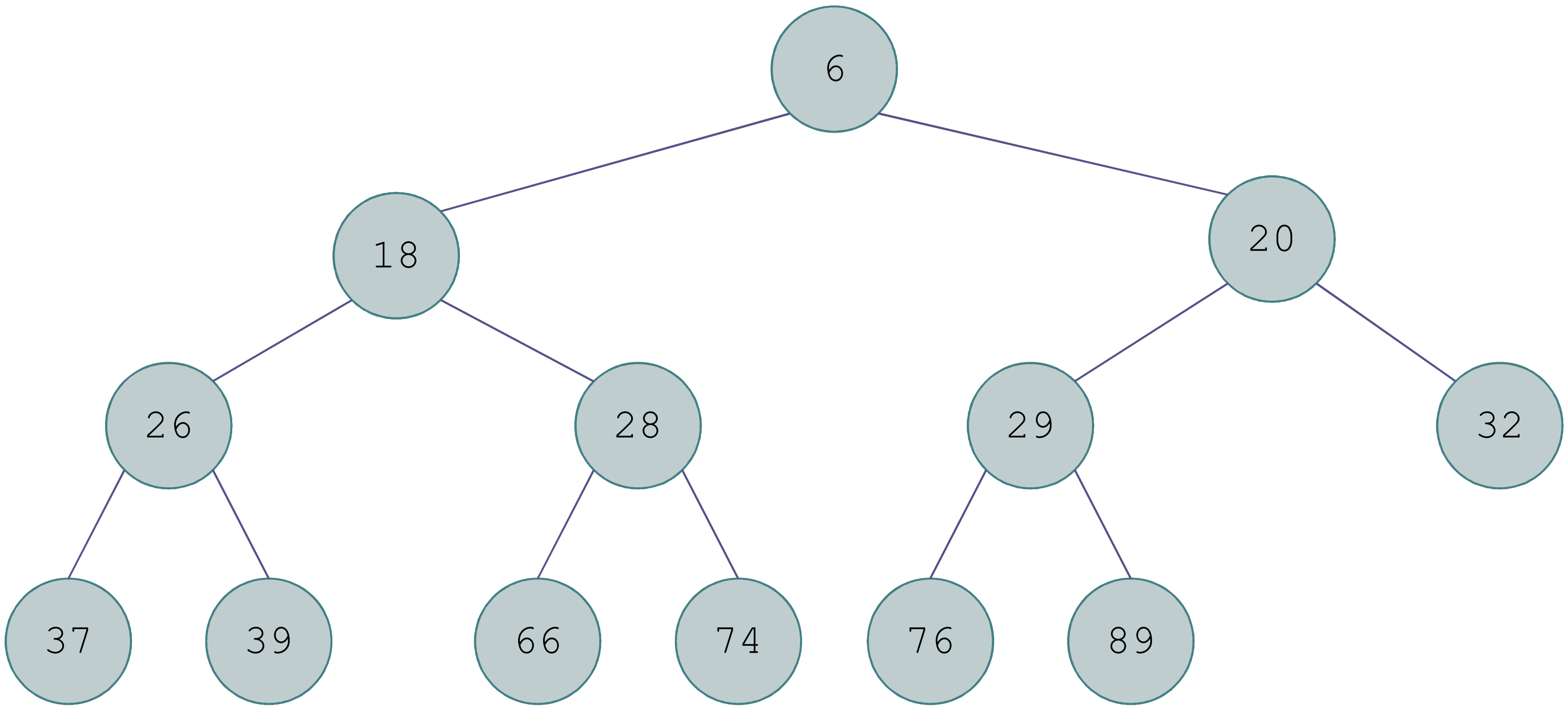
Trace of Heapsort (cont.)



Trace of Heapsort (cont.)



Trace of Heapsort (cont.)



Revising the Heapsort Algorithm

- 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

```java
public class HeapSort implements SortAlgorithm {

    public <T extends Comparable<T>>  void sort(T[] table) {
        buildHeap(table);

        shrinkHeap(table);
    }

    /**buildHeap transforms the table into a heap.
     * @pre  The array contains at least one item.
     * @post All items in the array are in heap order.
     * @param table The array to be transformed into a heap
     */
    private <T extends Comparable<T>>  void buildHeap(T[] table) {
        int n = 1;

        // Invariant: table[0 . . . n - 1] is a heap.
        while (n < table.length) {
            n++; // Add a new item to the heap and reheap.
            int child = n - 1;
            int parent = (child - 1) / 2; // Find parent.
            while (parent >= 0
                    && table[parent].compareTo(table[child]) < 0) {
                swap(table, parent, child);
                child = parent;
                parent = (child - 1) / 2;
            }
        }
    }
```

```
private <T extends Comparable<T>>  void shrinkHeap(T[] table) {
    int n = table.length;

    // Invariant: table[0 . . . n - 1] forms a heap.
    // table[n . . . table.length - 1] is sorted.
    while (n > 0) {
        n--;
        swap(table, 0, n);
        // table[1 . . . n - 1] form a heap.
        // table[n . . . table.length - 1] is sorted.
        int parent = 0;
        while (true) {
            int leftChild = 2 * parent + 1;
            if (leftChild >= n) {
                break; // No more children.
            }
            int rightChild = leftChild + 1;
            // Find the larger of the two children.
            int maxChild = leftChild;
            if (rightChild < n // There is a right child.
                    && table[leftChild].compareTo(table[rightChild]) < 0) {
                maxChild = rightChild;
            }
            // If the parent is smaller than the larger child,
            if (table[parent].compareTo(table[maxChild]) < 0) {
                // Swap the parent and child.
                swap(table, parent, maxChild);
                // Continue at the child level.
                parent = maxChild;
            } else { // Heap property is restored.
                break; // Exit the loop.
            }
        }
    }
}
```

```
/**
 * Swap the items in table[i] and table[j].
 * @param table The array that contains the items
 * @param i The index of one item
 * @param j The index of the other item
 */
private static <T extends Comparable<T>>  void swap(T[] table,
        int i, int j) {
    T temp = table[i];
    table[i] = table[j];
    table[j] = temp;
}
}
```

Quicksort

Section 8.9

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

44	75	23	43	55	12	64	77	33

Trace of Quicksort (cont.)

44	75	23	43	55	12	64	77	33

Arbitrarily select the first element as the pivot

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Swap the pivot with the element in the middle

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Quicksort Example(cont.)

44 is now in its correct position

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Now apply quicksort recursively to the two subarrays

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 55

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 64

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)

12	23	33	43	44	55	64	75	77

Left subarray has single value; it is sorted

Trace of Quicksort (cont.)

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 x *n* log *n*

Analysis of Quicksort (cont.)

- 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 $O(n^2)$
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts
  - We'll discuss a solution later

```java
public abstract class QuickSort implements SortAlgorithm {

public <T extends Comparable<T>>  void sort(T[] table) {
        // Sort the whole table.
        quickSort(table, 0, table.length - 1);
    }

    /**
     * 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
     */
    protected <T extends Comparable<T>>  void quickSort(T[] table,
            int first,
            int last) {
        if (first < last) { // There is data to be sorted.
            // Partition the table.
            int pivIndex = partition(table, first, last);
            // Sort the left half.
            quickSort(table, first, pivIndex - 1);
            // Sort the right half.
            quickSort(table, pivIndex + 1, last);
        }
    }
}
```

Algorithm for Partitioning

44	75	23	43	55	12	64	77	33

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

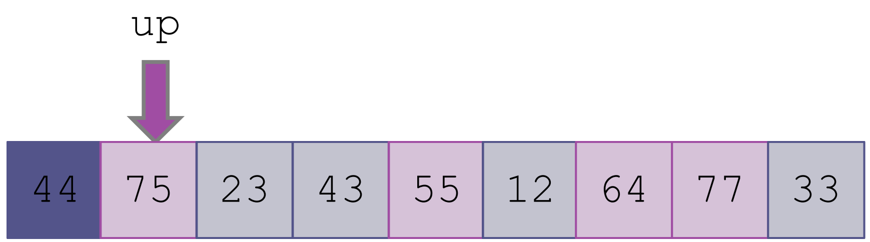
For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

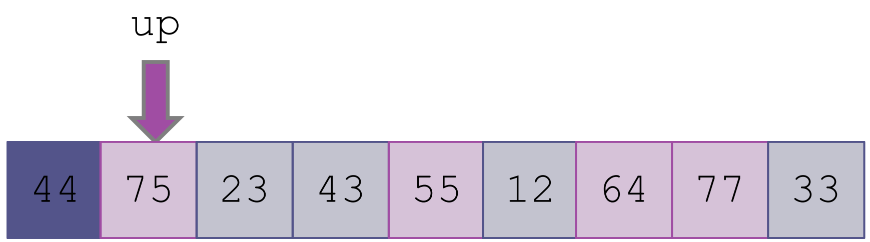
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



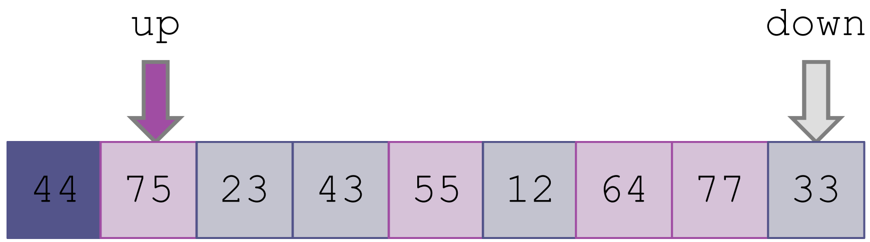
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



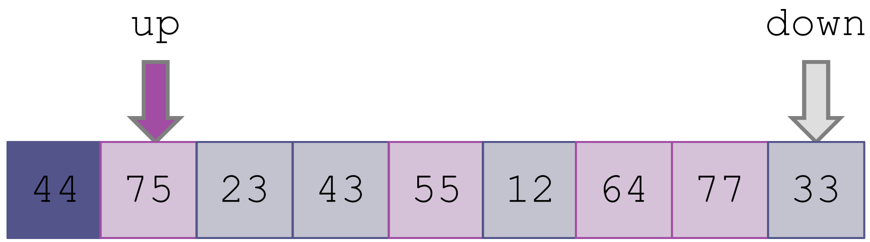
Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Exchange these values

Trace of Partitioning (cont.)

44	33	23	43	55	12	64	77	75

Exchange these values

Trace of Partitioning (cont.)

44	33	23	43	55	12	64	77	75

Repeat

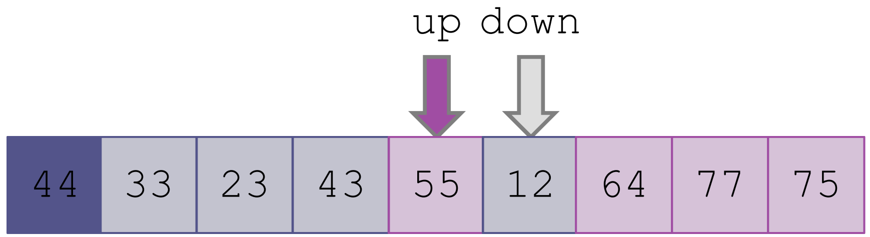
Trace of Partitioning (cont.)

up

44	33	23	43	55	12	64	77	75

Find first value at left end greater than pivot

Trace of Partitioning (cont.)



Find first value at right end less than or equal to pivot

Trace of Partitioning (cont.)

44	33	23	43	12	55	64	77	75

Exchange

Trace of Partitioning (cont.)

44	33	23	43	12	55	64	77	75

Repeat

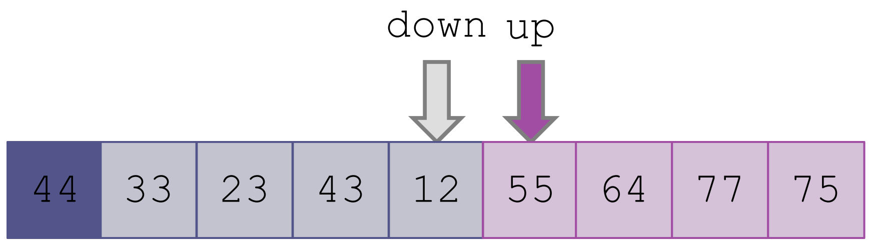
Trace of Partitioning (cont.)

up

44	33	23	43	12	55	64	77	75

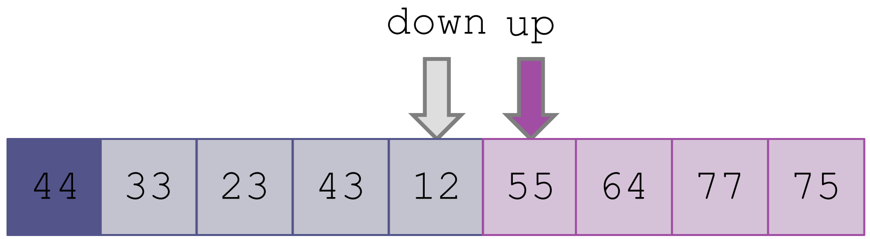
Find first element at left end greater than pivot

Trace of Partitioning (cont.)



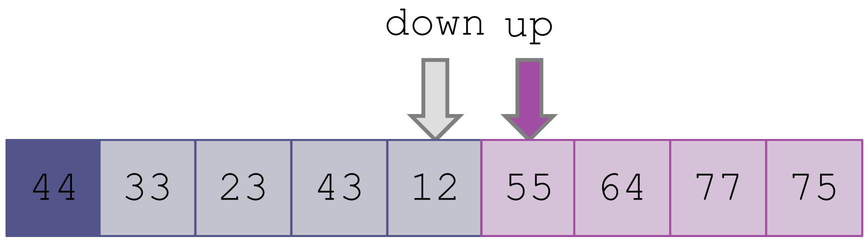
Find first element at right end less than or equal to pivot

Trace of Partitioning (cont.)



Since `down` has "passed" `up`, do not exchange

Trace of Partitioning (cont.)



Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)

down

12	33	23	43	44	55	64	77	75

Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)

down

12	33	23	43	44	55	64	77	75

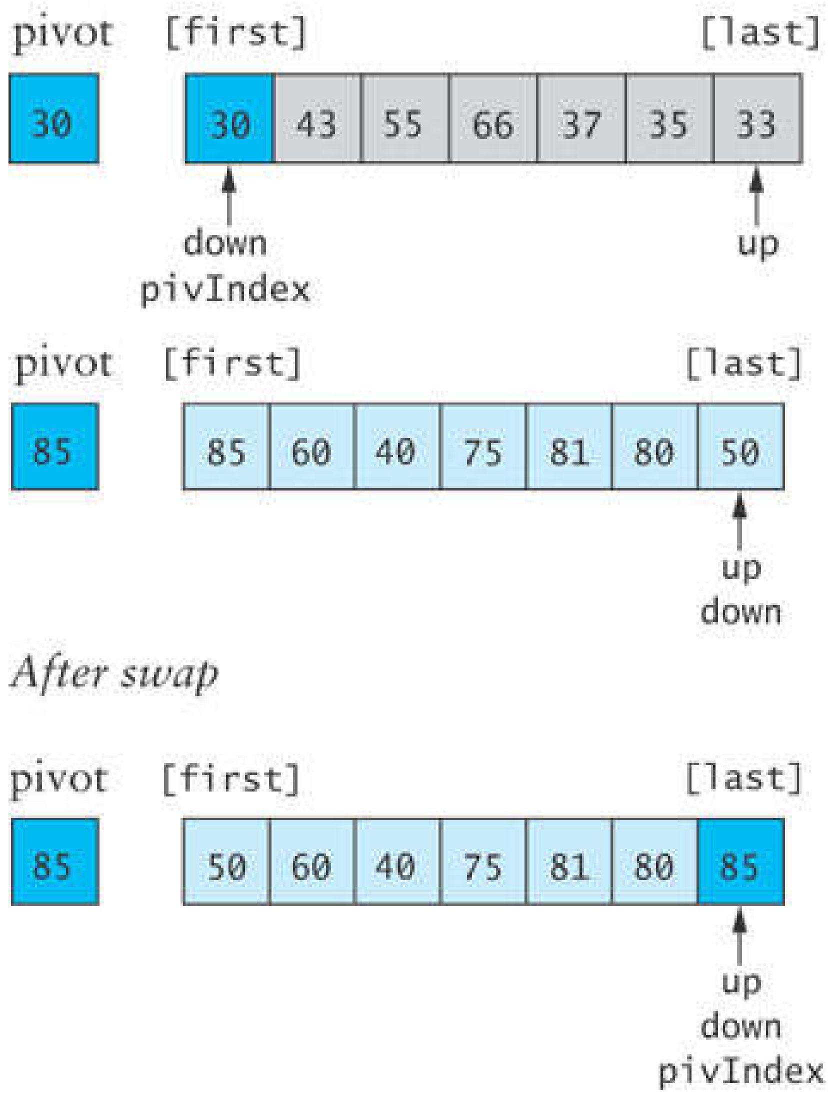
The pivot value is in the correct position; return the value of `down` and assign it to the pivot index `pivIndex`

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`.

Code for `partition` when `Pivot` is the largest or smallest value



```java
protected <T extends Comparable<T>>  int partition(T[] table,
            int first, int last) {
        // Select the first item as the pivot value.
        T pivot = table[first];
        int up = first; int down = last;
        do {
            // Invariant:
            // All items in table[first . . . up - 1] <= pivot
            // All items in table[down + 1 . . . last] > pivot
            while ((up < last) && (pivot.compareTo(table[up]) >= 0)) {
                up++;
            }
            // assert: up equals last or table[up] > pivot.
            while (pivot.compareTo(table[down]) < 0) {
                down--;
            }
            // assert: down equals first or table[down] <= pivot.
            if (up < down) { // if up is to the left of down.
                // Exchange table[up] and table[down].
                swap(table, up, down);
            }
        } while (up < down); // Repeat while up is left of down.

        // Exchange table[first] and table[down] thus putting the
        // pivot value where it belongs.
        swap(table, first, down);
        // Return the index of the pivot value.
        return down;
    }
```

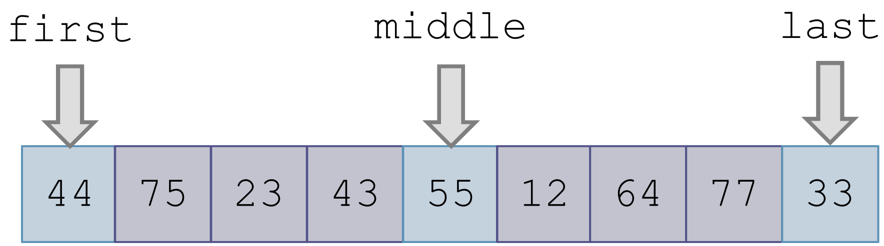
Revised Partition Algorithm

- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split
  - Use three references: `first`, `middle`, `last`
  - Select the median of the these items as the pivot

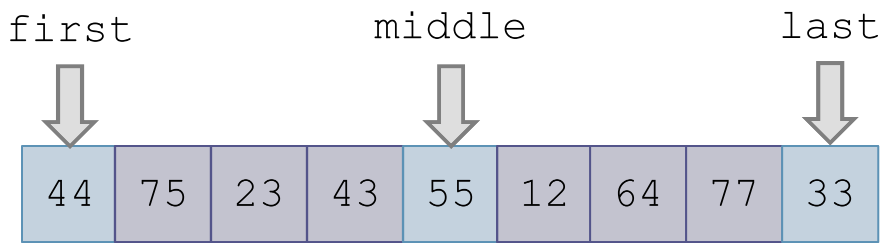
Trace of Revised Partitioning

44	75	23	43	55	12	64	77	33

Trace of Revised Partitioning (cont.)

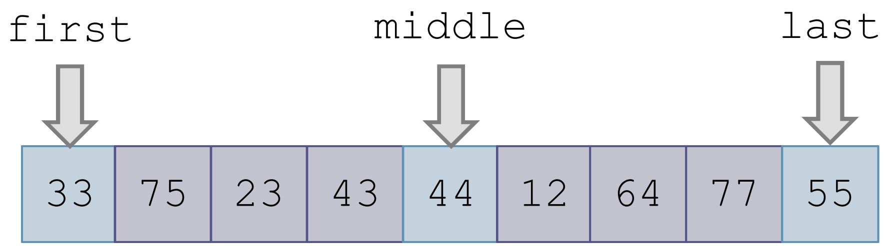


Trace of Revised Partitioning (cont.)



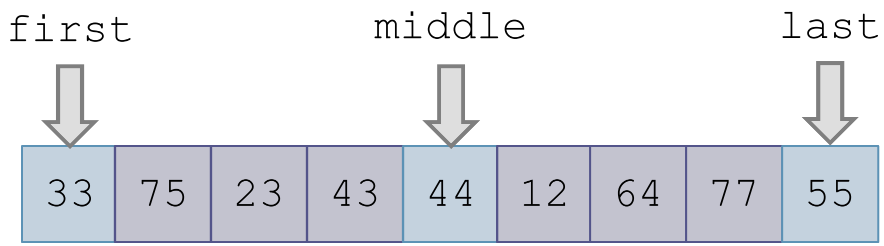
Sort these values

Trace of Revised Partitioning (cont.)



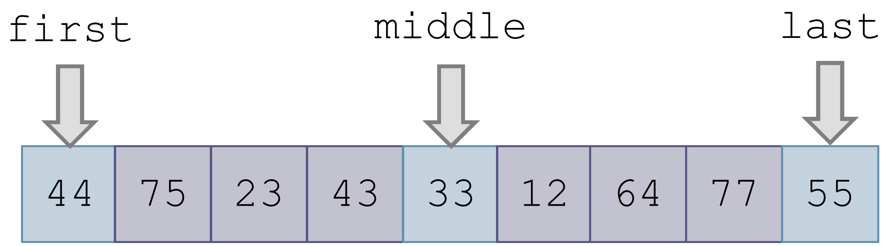
Sort these values

Trace of Revised Partitioning (cont.)



Exchange middle with first

Trace of Revised Partitioning (cont.)



Exchange middle with first

Trace of Revised Partitioning (cont.)

44	75	23	43	33	12	64	77	55

Run the partition algorithm using the first element as the pivot

Algorithm for Revised `partition` Method

Algorithm for Revised partition Method

1. Sort `table[first]`, `table[middle]`, and `table[last]`
2. Move the median value to `table[first]` (the pivot value) by exchanging `table[first]` and `table[middle]`.
3. Initialize `up` to `first` and `down` to `last`
4. `do`
5. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`
6. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`
7. `if up < down` then
8. Exchange `table[up]` and `table[down]`
9. `while up` is to the left of `down`
10. Exchange `table[first]` and `table[down]`
11. Return the value of `down` to `pivIndex`

Code for Revised `partition` Method

- Listing 8.10 (`QuickSort2`, page 459)

Testing the Sort Algorithms

Section 8.10

Testing the Sort Algorithms

- Use a variety of test cases
  - small and large arrays
  - arrays in random order
  - arrays that are already sorted
  - arrays with duplicate values
- Compare performance on each type of array

Driver to Test Sort Algorithms

- Listing 8.11(`TestSort.java`, page 461)

The Dutch National Flag Problem (optional)

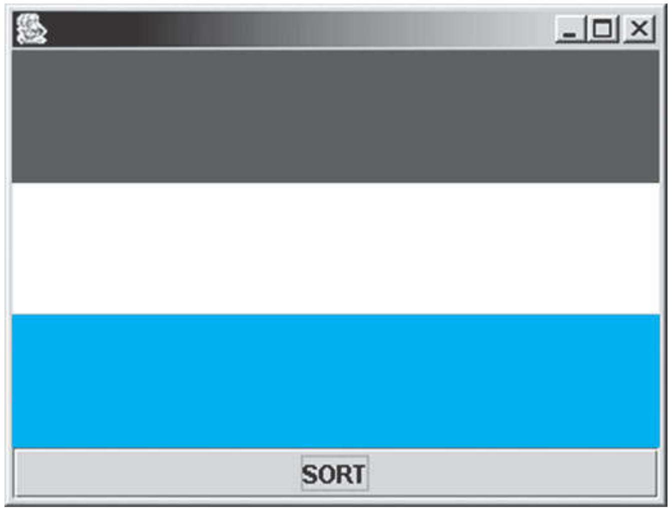
Section 8.11

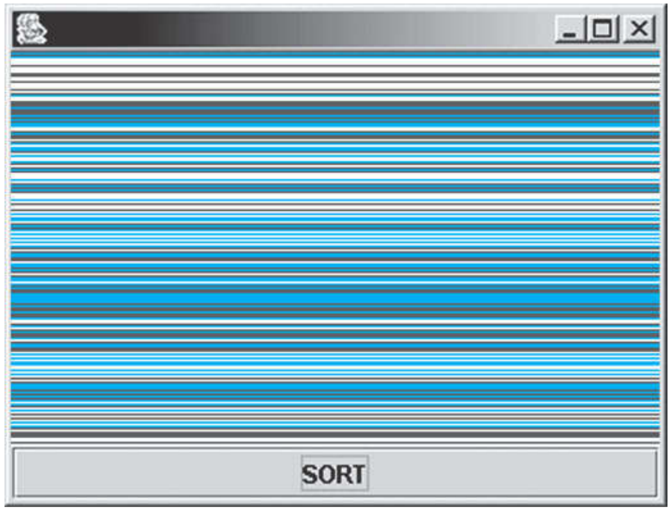
The Dutch National Flag Problem

- A variety of partitioning algorithms for quicksort have been published
- A popular variation uses a single left-to-right scan of the array
- Edsger W. Dijkstra developed a partitioning algorithm using both techniques to partition an array into three segments

Problem

- Partition a disordered three-color flag into the appropriate three segments by writing software for an unscrambling machine





Analysis

- The unscrambling machine can
  - look at one thread in the flag and determine its color
  - swap the position of two threads in the flag
  - execute `while` loops and `if` statements

Design – Loop Invariant

- As with quicksort, values between `first` and `up` are less than or equal to the pivot
- Values between `down` and `last` are greater than the pivot
- Values between `up` and `down` are unknown
- Initially,
  - The unknown region is the entire array (`first == up` and `down == last`)

Design

- Have four regions:
  - Red (light gray)
  - white
  - blue
  - unknown (dark gray

- Flag is complete:
  - if `0 <= i < red` then `threads[i]` is red
  - if `white < i <= blue` then `threads[i]` is white
  - if `blue < i < HEIGHT` then `threads[i]` is blue
- If `red <= i <= white` then the flag is not complete and the color is unknown

Algorithm

We can solve our problem by establishing the loop invariant and then executing a loop that both preserves the loop invariant and shrinks the unknown region.

1. Set `red` to `0`, `white` to `HEIGHT - 1`, and `blue` to `HEIGHT - 1`. This establishes our loop invariant with the unknown region the whole flag and the red, white, and blue regions empty.
2. `while red < white`
3. Shrink the distance between `red` and `white` while preserving the loop invariant.

Preserving the Loop Invariant

- Assume we know the value of `threads[white]`
  - either leave it where it is (in the white region if it is white) or
  - place it in the region where it belongs

Preserving the Loop Invariant

- Three cases
  1. The color of `threads[white]` is white
     - Decrement the value of `white`, increasing the size of the white region by one thread
  2. The color of `threads[white]` is red
     - The color of `threads[red]` is unknown
     - Swap the thread at `threads[red]` with the thread at `threads[white]`, adding the thread to the end of the red region and reducing the size of the unknown region by one thread
  3. The color of `threads[white]` is blue
     - The color of `threads[blue]` is white
     - Swap the thread at `threads[white]` with the thread at `threads[blue]`
     - Decrement both `blue` and `white`, inserting the thread at the beginning of the blue region and reducing the size of the unknown region by one thread

Implementation

- Listing 8.12 (`DutchNationalFlag.java`, page 465)

Comparison of Sort Algorithms

Summary

Sort Review

Number of Comparisons			
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$