Fall 2019

Sorting

CMPE 250 - Data Structures & Algorithms

Presenter: Meriç Turan

7 NOVEMBER 2019

Sorting Algorithms

- There are several sorting algorithms. In this PS, we will investigate the followings:
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
  - Bucket Sort
  - Radix Sort

Insertion Sort

- One of the simplest sorting algorithms is the insertion sort.
- Insertion sort consists of ***N*–1 passes**.
- **For pass p**=1 through *N*–1, insertion sort ensures that the elements in positions **0 through p are in sorted order.**
- Insertion sort makes use of the fact that elements in positions 0 through p – 1 are already known to be in sorted order.

34	8	64	51	32	21

Insertion Sort

- **In pass p**, we move the element in position **p left until its correct place** is found among the first p+1 elements.
- The element in position **p is moved to tempVariable**, and all **larger elements** (prior to position p) are **moved one spot to the right**.
- Then **tmp is moved** to the correct spot.
- This is the same technique that was used in the implementation of binary heaps.
- Because of the nested loops, each of which can take *N* iterations, insertion sort is $O(N^2)$.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure: Insertion sort after each pass.

Merge Sort

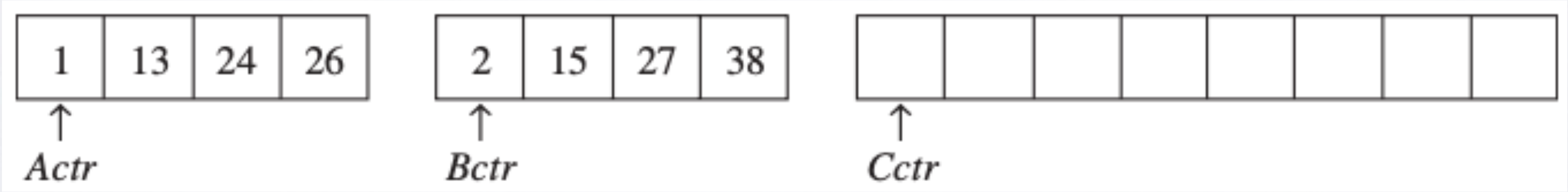
- Merge sort runs in **O(*N*log*N*) worst-case running time**, and the number of comparisons used is nearly optimal.
- It is a fine example of a **recursive algorithm**.
- The **fundamental operation** in this algorithm is **merging two sorted lists**.
- Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

Merge Sort

- Basic merging algorithm takes **two input arrays A and B**, an **output array C**, and **three counters**, ***Actr***, ***Bctr***, and ***Cctr***, which are initially set to the beginning of their respective arrays.

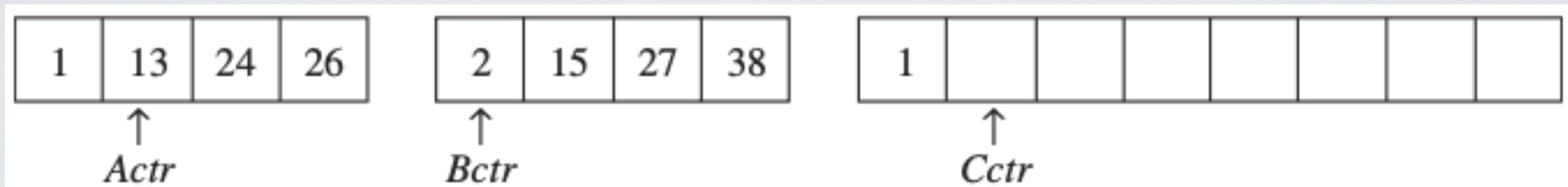
- The **smaller of A[*Actr*] and B[*Bctr*]** is copied to the next entry in C, and the appropriate **counters are advanced**.

- When either input list is exhausted, the remainder of the other list is copied to C.

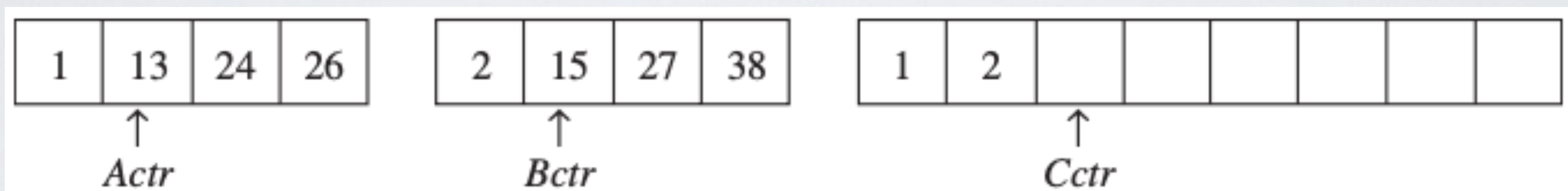


Merge Sort

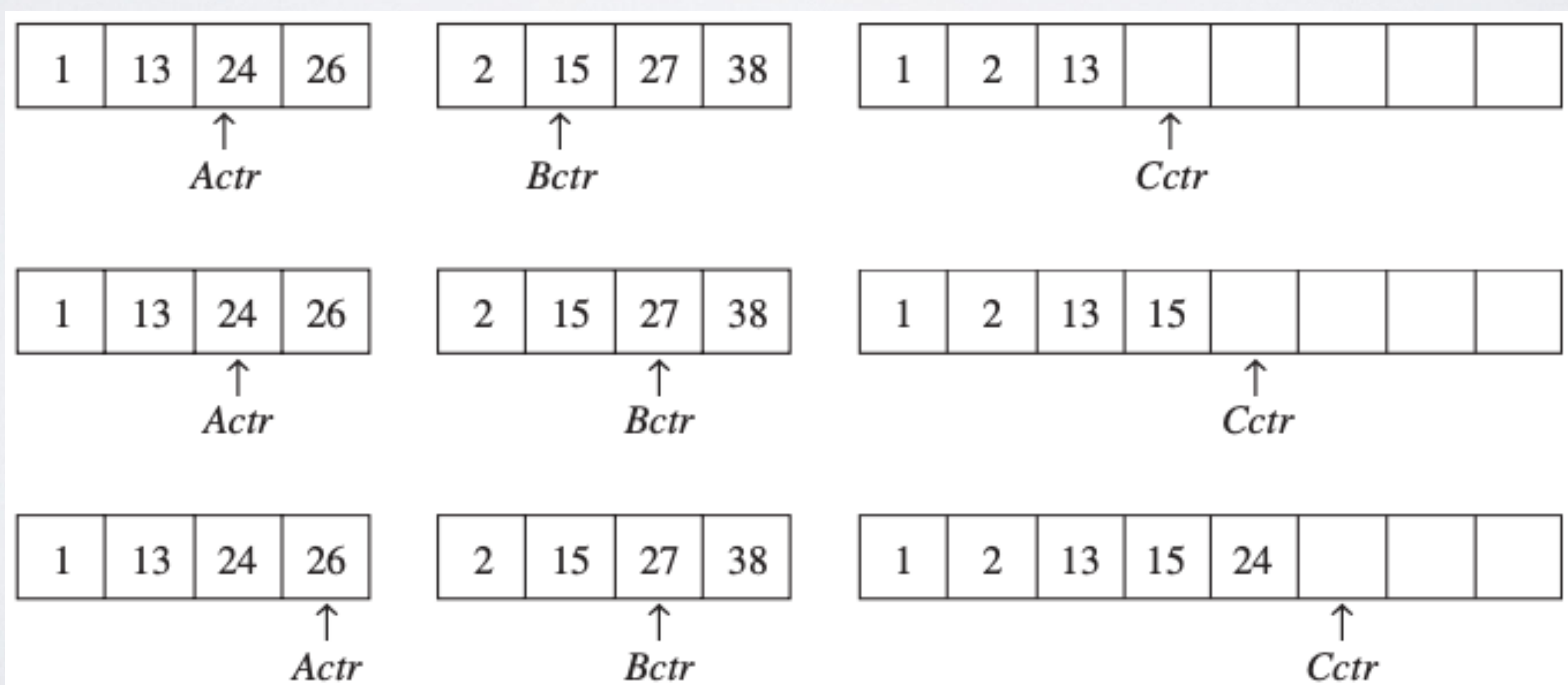
First, a comparison is done between 1 and 2. 1 is added to C, and then 13 and 2 are compared.



2 is added to C, and then 13 and 15 are compared.

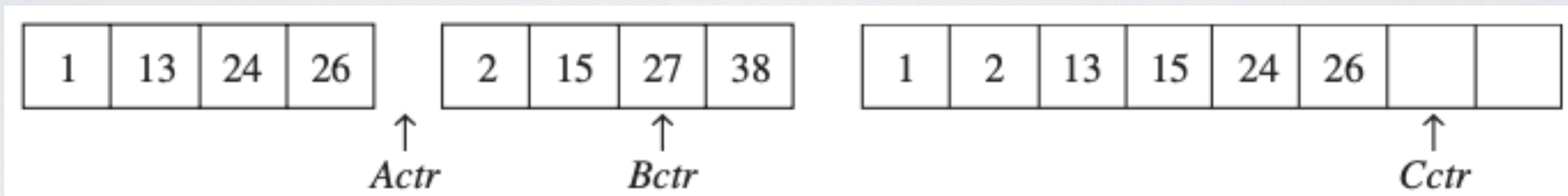


13 is added to C, and then 24 and 15 are compared. This proceeds until 26 and 27 are compared.

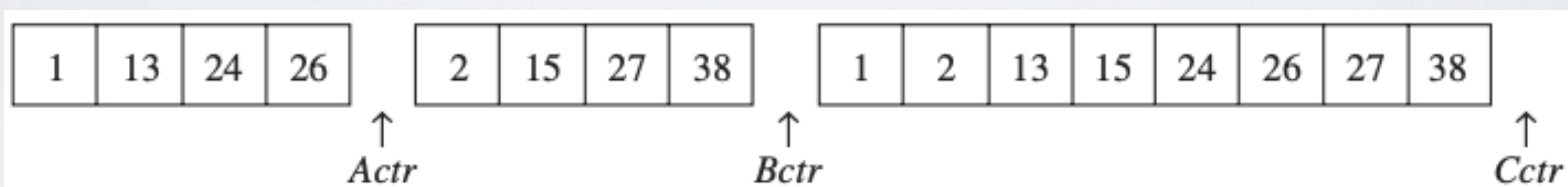


Merge Sort

26 is added to C, and the A array is exhausted.

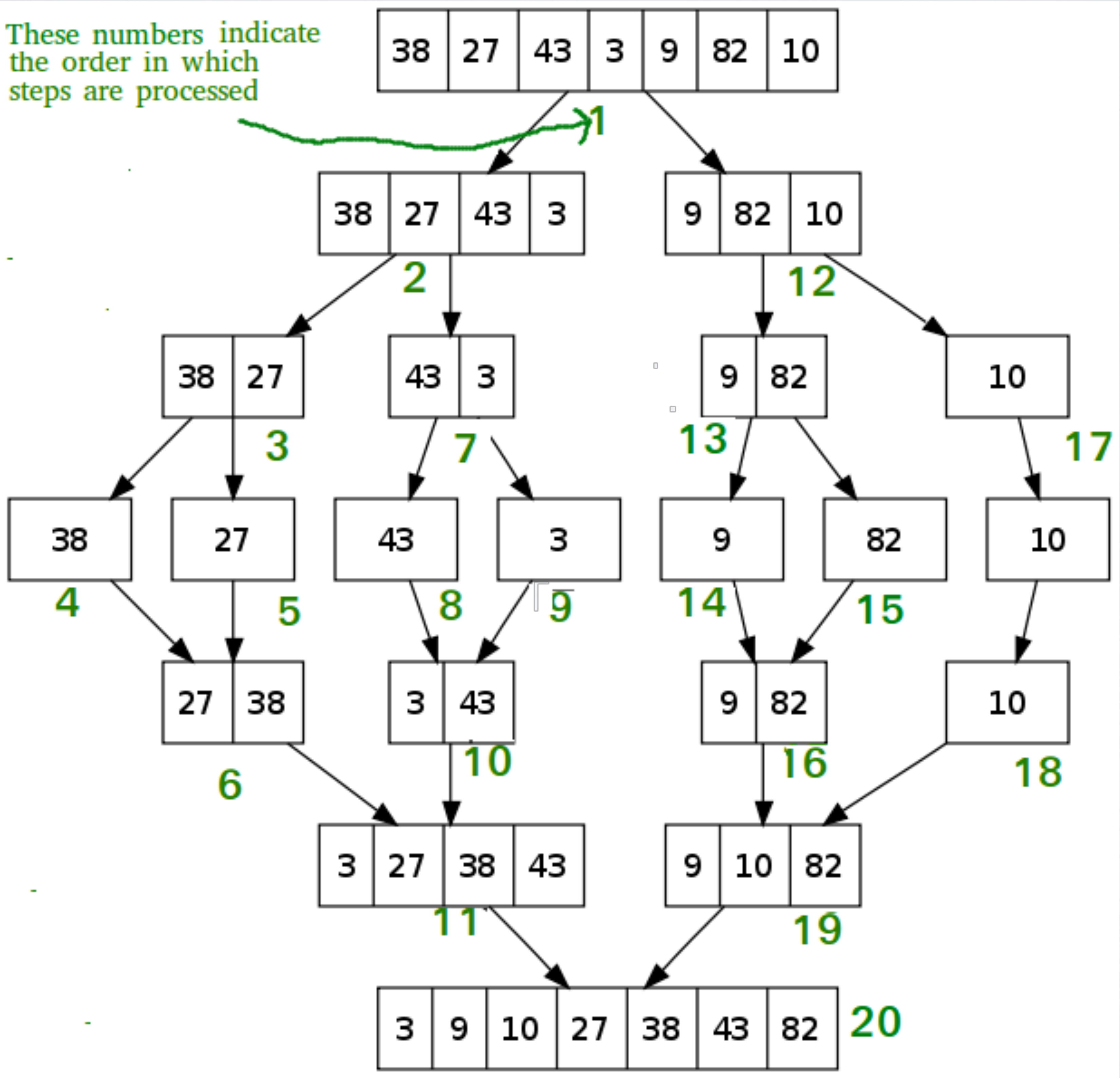


The remainder of the B array is then copied to C.



Time to merge two sorted lists **is clearly linear**, because **at most *N*–1 comparisons** are made, where *N* is the total number of elements. To see this, note that every comparison adds an element to C, except the last comparison, which adds at least two.

Merge Sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Quick Sort

- As its name implies for C++, quick sort has historically been the fastest known generic sorting algorithm in practice.
- Its **average running time** is **O(*N*log*N*)**.
- It is very fast, mainly due to a very tight and highly optimized inner loop.
- It has **O(N^2) worst-case performance**, but this can be made exponentially unlikely with a little effort.
- Like merge sort, quick sort is a **divide-and-conquer recursive algorithm**.

Quick Sort - Basic Algorithm

- Arbitrarily choose any item, and then **form three groups**:
  - those **smaller than** the chosen item,
  - those **equal to** the chosen item,
  - and those **larger than** the chosen item.
- Recursively **sort the first and third groups**, and then **concatenate the three groups**.
- **Result** is **guaranteed** by the basic principles of recursion **to be** a **sorted** arrangement of the original list.

Quick Sort

```
template <typename Comparable>
void SORT( vector<Comparable> & items )
{
    if( items.size( ) > 1 )
    {
        vector<Comparable> smaller;
        vector<Comparable> same;
        vector<Comparable> larger;

        auto chosenItem = items[ items.size( ) / 2 ];

        for( auto & i : items )
        {
            if( i < chosenItem )
                smaller.push_back( std::move( i ) );
            else if( chosenItem < i )
                larger.push_back( std::move( i ) );
            else
                same.push_back( std::move( i ) );
        }

        SORT( smaller );     // Recursive call!
        SORT( larger );      // Recursive call!

        std::move( begin( smaller ), end( smaller ), begin( items ) );
        std::move( begin( same ), end( same ), begin( items ) + smaller.size( ) );
        std::move( begin( larger ), end( larger ), end( items ) - larger.size( ) );
    }
}
```

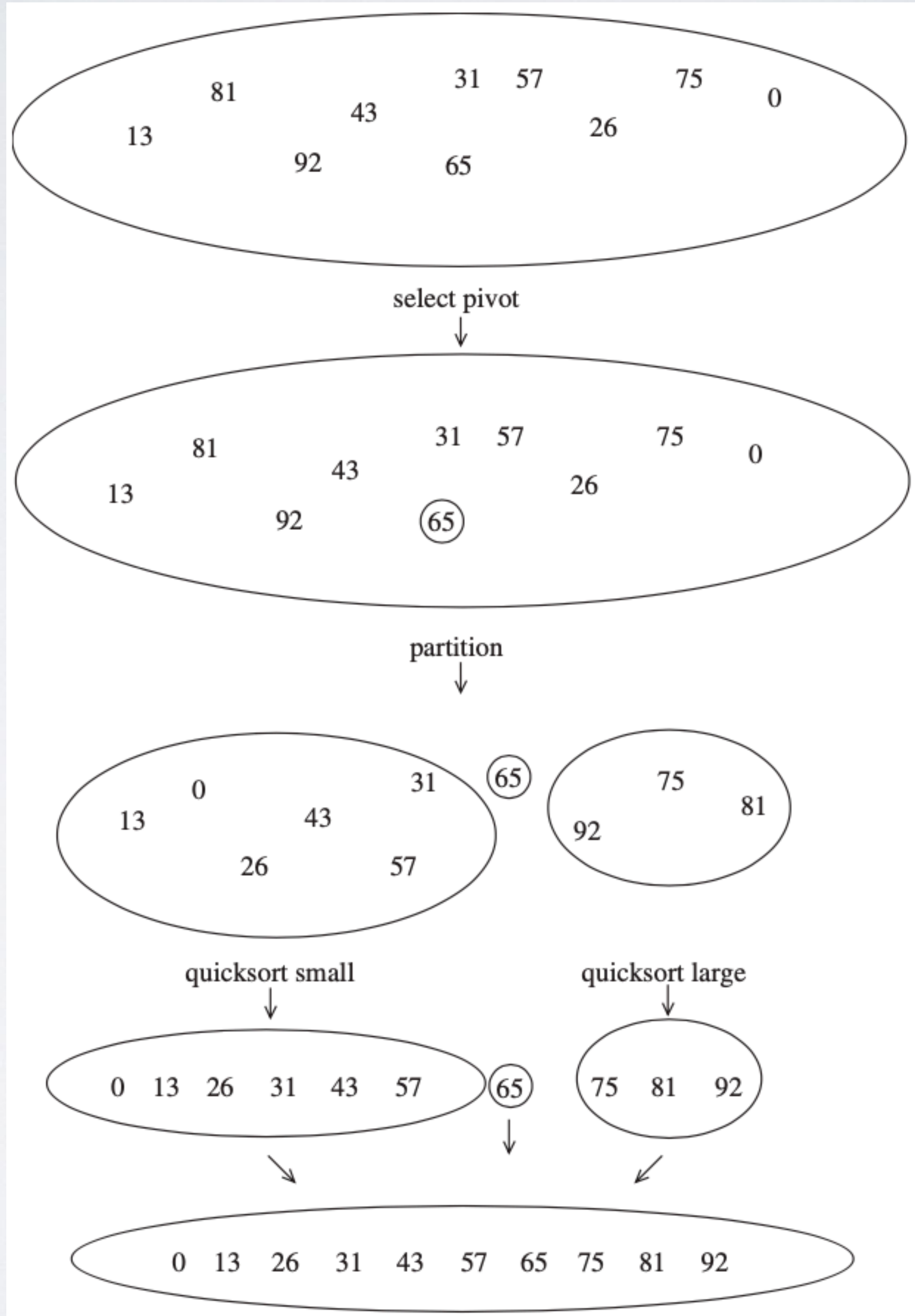
Quick Sort - Algorithm

- Below, I describe the most common implementation of quick sort "classic quick sort" in which the input is an array, and in which **no extra arrays** are created by the algorithm.
- The **classic quick sort algorithm** to sort an array S consists of the following four steps:

  1. If the number of elements in S is 0 or 1, then return.
  2. Pick any element v in S. This is called the pivot (we will see the details and importance of pivot selection later).
  3. Partition S − {v} (remaining elements in S) into two disjoint groups: $S_1 = \{x \in S-\{v\} \mid x \leq v\}$, and $S_2 = \{x \in S-\{v\} \mid x \geq v\}$.
  4. Return {quicksort(S_1) followed by v followed by quicksort(S_2)}.

- Although the algorithm as described works no matter which element is chosen as pivot, some choices are obviously better than others.

Quick Sort - Picking the Pivot



Quick Sort - Picking the Pivot (A Wrong Way)

- The popular choice is to use the **first element as the pivot**.
- This is **acceptable if the input is random**, but if the input is presorted or in reverse order, then the pivot provides a poor partition, because either all the elements go into S_1 or they go into S_2.
- Worse, this **happens consistently throughout the recursive calls**.
- Practical effect is that if the **first element** is used as the **pivot** and the **input** is **presorted**, then quick sort will take **quadratic time** to do essentially nothing at all, which is quite embarrassing.
- Moreover, **presorted input is quite frequent**, so using the first element as pivot is an **absolutely horrible idea** and should be discarded immediately.

Quick Sort - Picking the Pivot (A Safe Maneuver)

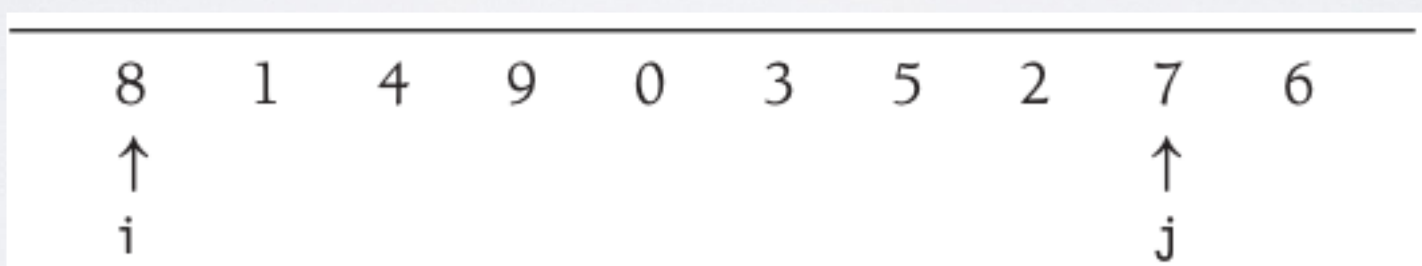
- A safe course is merely to **choose the pivot randomly**.
- This strategy is generally perfectly safe, unless the **random number generator has a flaw** (which is **not** as **uncommon** as you might think), since it is very unlikely that a random pivot would consistently provide a poor partition.

Quick Sort - Picking the Pivot (Median-of-Three Partitioning)

- Median of a group of N numbers is the $\lceil N/2 \rceil$th largest number.
- **Best choice** of pivot would be the **median of the array**. Unfortunately, this is **hard to calculate** and would **slow down quick sort** considerably.
- A good estimate can be obtained by **picking three elements randomly** and using the **median of these three as pivot**. R**andomness turns out not to help much**, so the common course is to use as pivot the **median of the left, right, and center** elements.
- For instance, with input {8, 1, 4, 9, 6, 3, 5, 2, 7, 0}, left element is 8, right element is 0, and center (in position $\lfloor(\text{left} + \text{right})/2\rfloor$) element is 6. Thus, the pivot would be v = 6.
- Using median-of-three partitioning **eliminates the bad case for sorted input** and reduces the number of comparisons.

Quick Sort - Partitioning Strategy

- There are **several partitioning strategies** used in practice, but the one described here is known to give good results.
- First step is to get the **pivot element** out of the way by **swapping** it with the **last element**.
- **i** starts at the **first element** and **j** starts at the **next-to-last element**.



- What our partitioning stage wants to do is to **move all the small elements to the left part** of the array and **all the large elements to the right part**. “Small” and “large” are, of course, relative to the pivot.

Quick Sort - Partitioning Strategy

- While i is to the left of j, we move i right, skipping over elements that are smaller than the pivot.
- We move j left, skipping over elements that are larger than the pivot.

8	1	4	9	0	3	5	2	7	6
↑								↑	
i								j	

- In the example above, i would not move and j would slide over one place.

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

- We then swap the elements pointed to by i and j and repeat the process until i and j cross.

Quick Sort - Partitioning Strategy

After First Swap

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

Before Second Swap

2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

After Second Swap

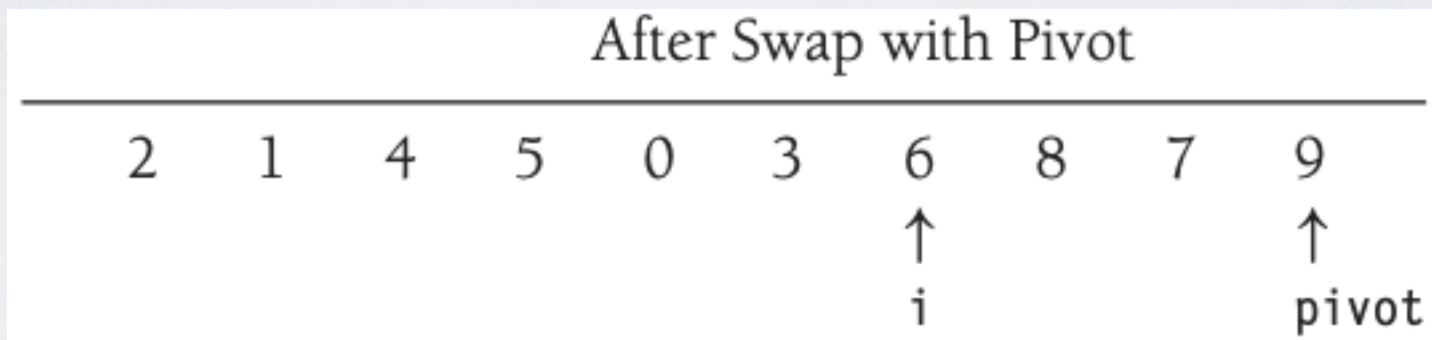
2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

Before Third Swap

2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	i			

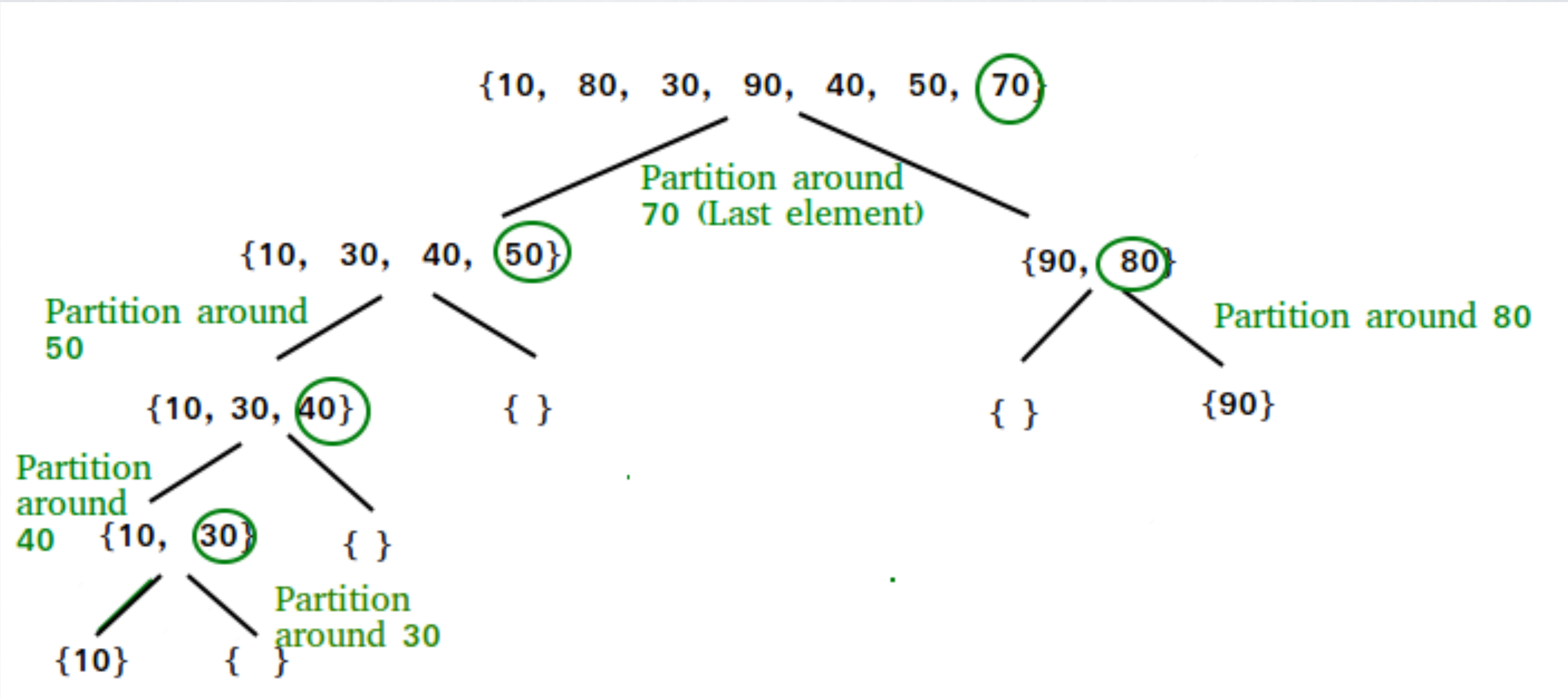
Quick Sort - Partitioning Strategy

- At this stage, i and j have crossed, so no swap is performed. Final part of the partitioning is to swap the pivot element with the element pointed to by i.



- When the pivot is swapped with i in the last step, we know that every element in a position p < i must be small.

Quick Sort - Example



Heap Sort

- Priority queues can be used to sort in **O(*N*log*N*)** time.
- For heap sort, basic strategy is to **build a binary heap of *N* elements**, which takes O(*N*) time.
- We **then perform *N* deleteMin operations**. The **smallest elements leave the heap** in sorted order.
- By **recording these elements in a second array** and then **copying the array back**, we sort N elements.
- Since each **deleteMin takes O(log*N*)** time, the total **running time is O(*N*log*N*)**.

Heap Sort

- **Main problem** with this algorithm is that it uses **an extra array**.
- Thus, the **memory requirement is doubled**.
- A clever way to avoid using a second array makes use of the fact that **after each deleteMin**, the **heap shrinks by 1**.
- Thus, the **cell that was last in the heap** can be used to **store the element that was just deleted**.
- Using this strategy, **after the last deleteMin** the array will contain the elements in **decreasing sorted order**.
- If we want the elements in the more typical **increasing sorted order**, we can change the ordering property so that the parent has a larger element than the child (**max heap**).

Heap Sort - Example 1

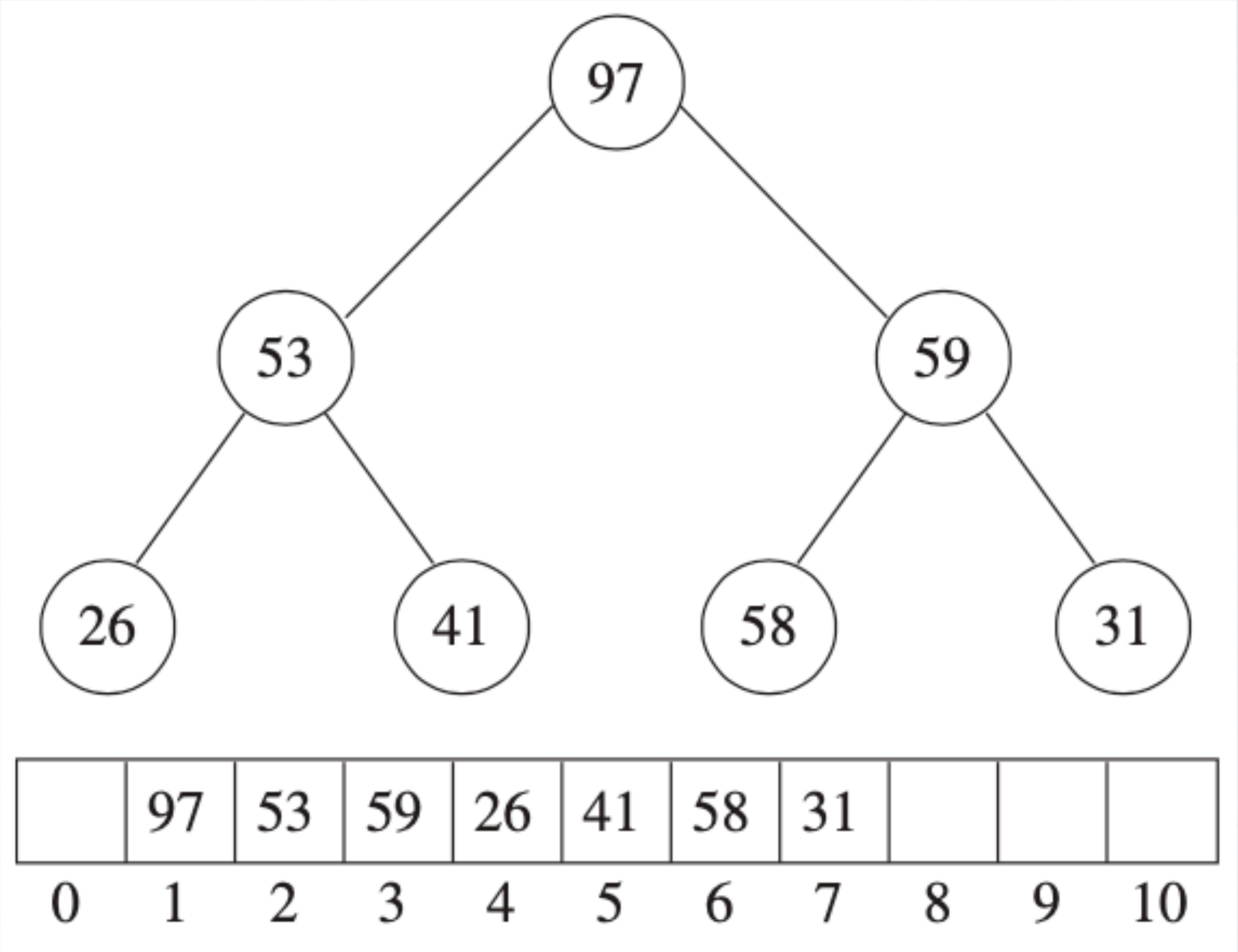


Figure: Max heap after buildHeap phase.

Heap Sort - Example 1

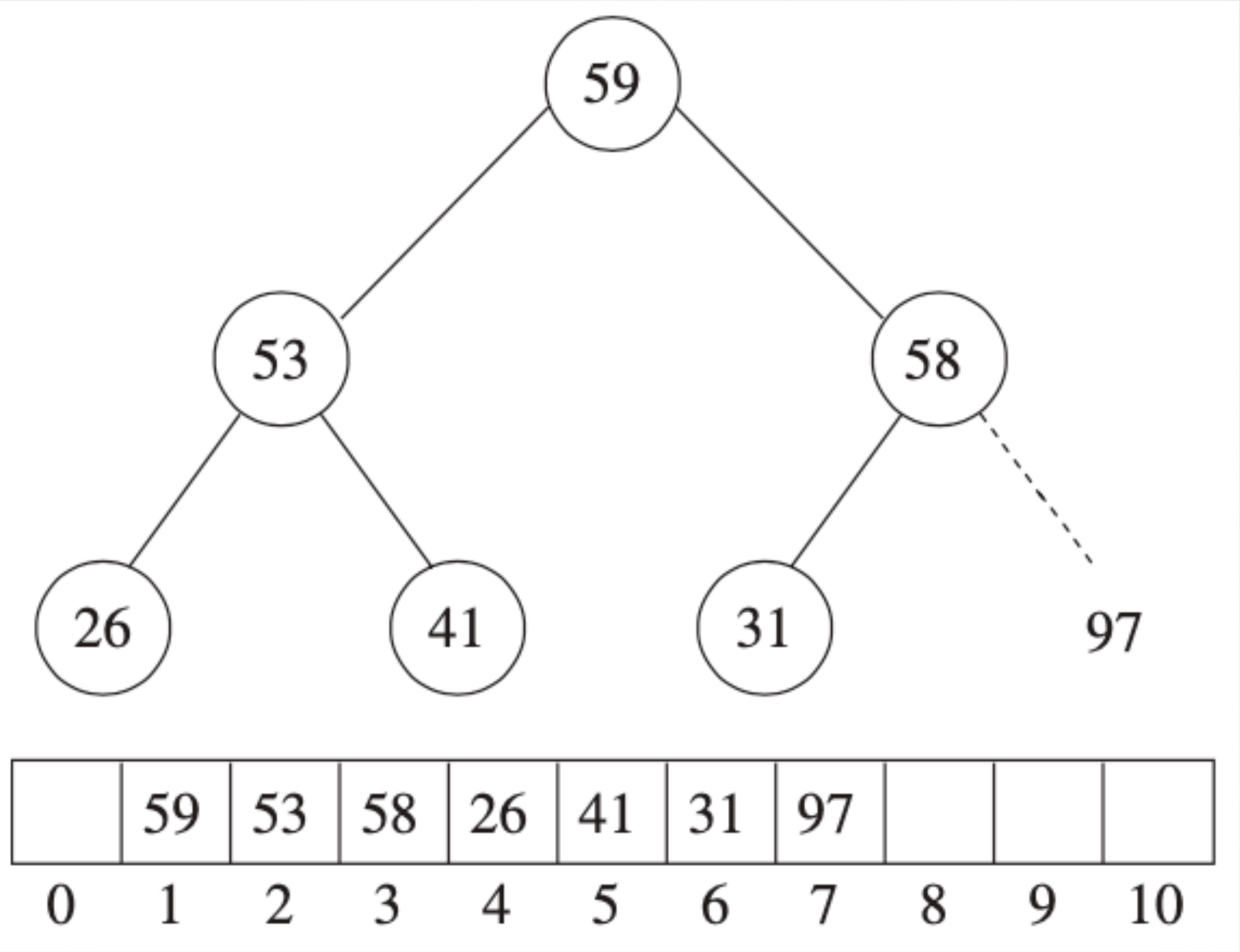
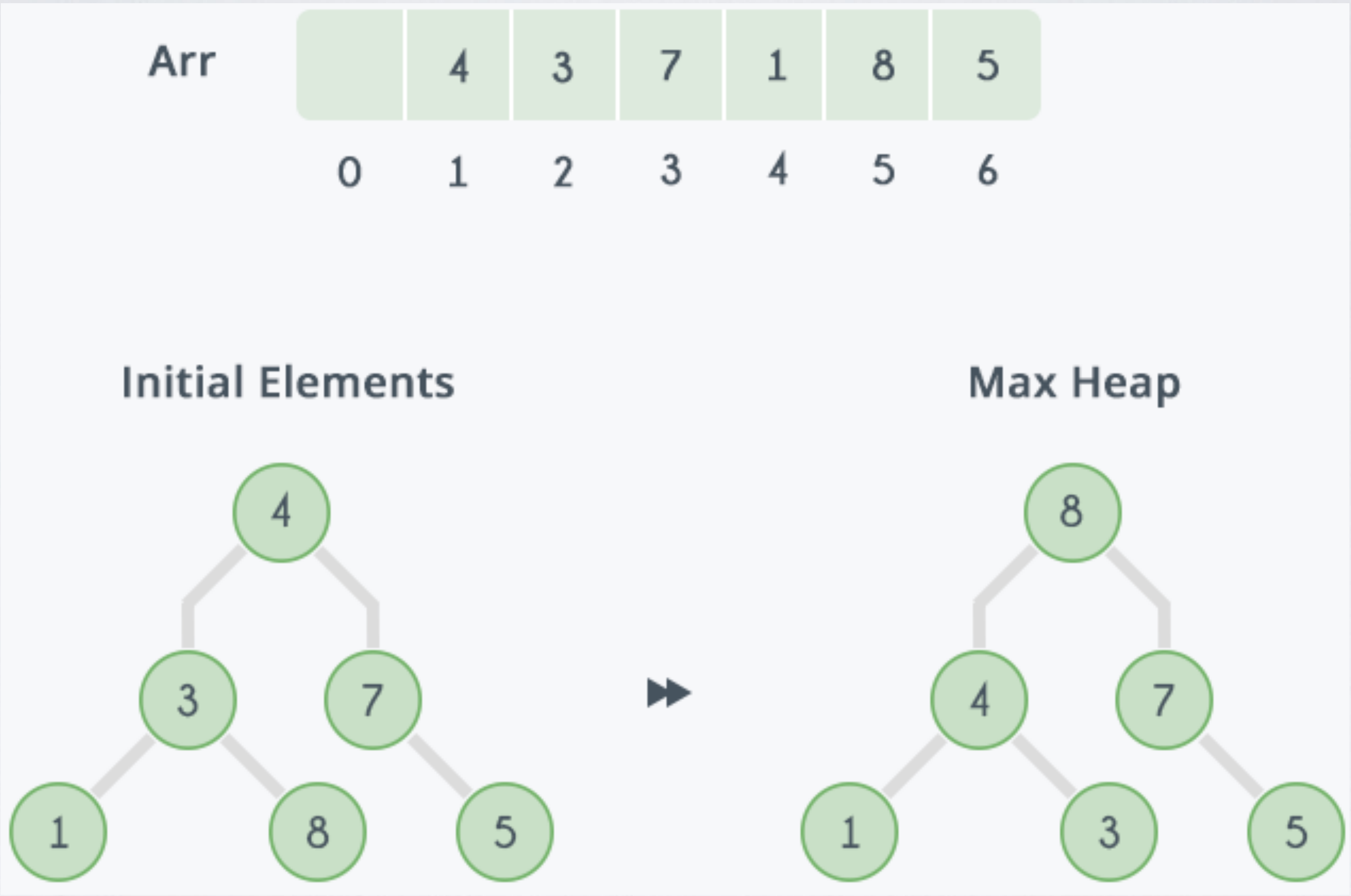
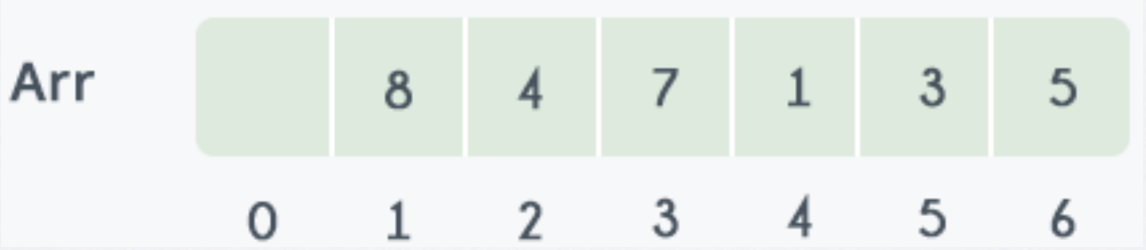


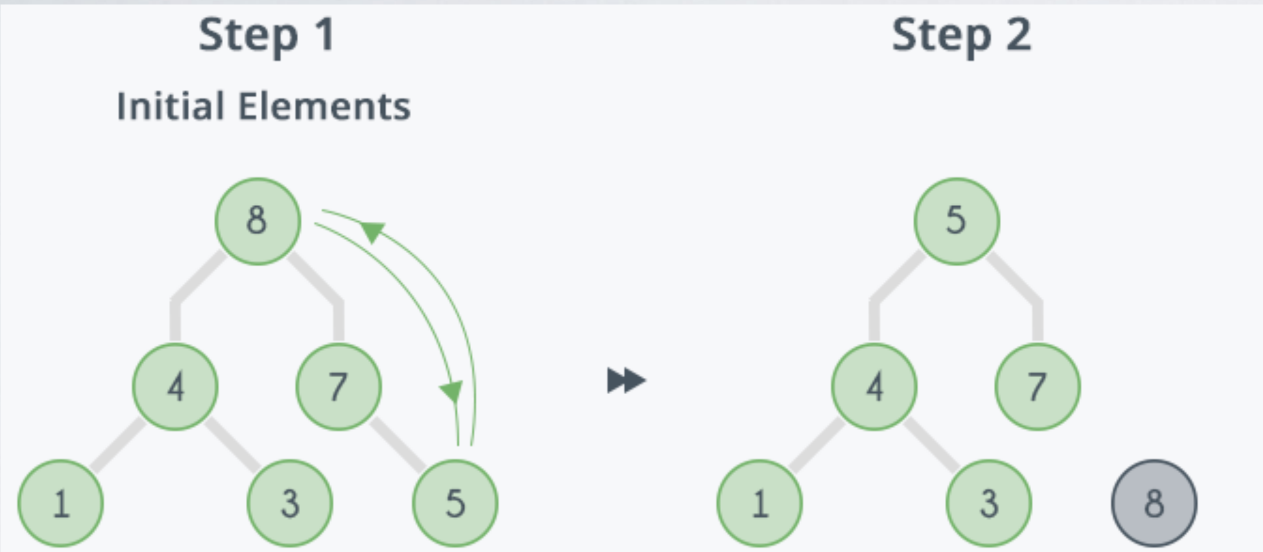
Figure: Heap after first deleteMax.

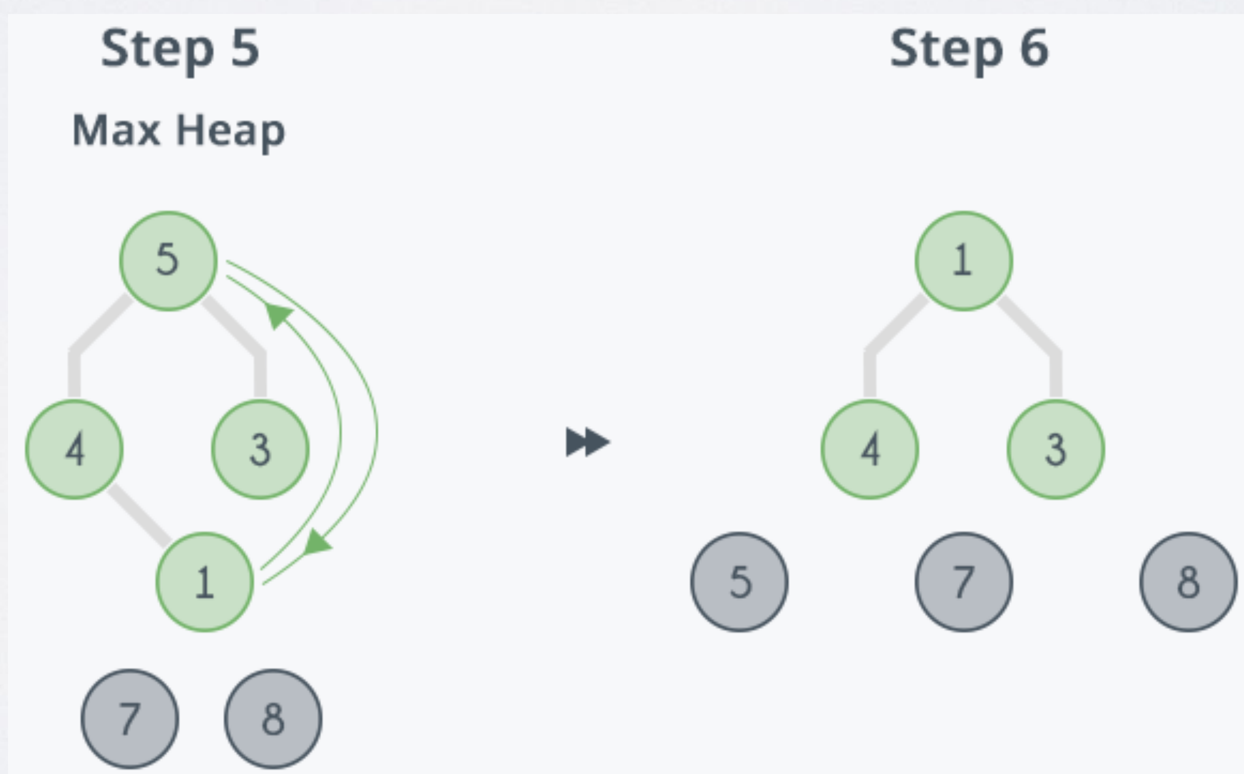
Heap Sort - Example 2



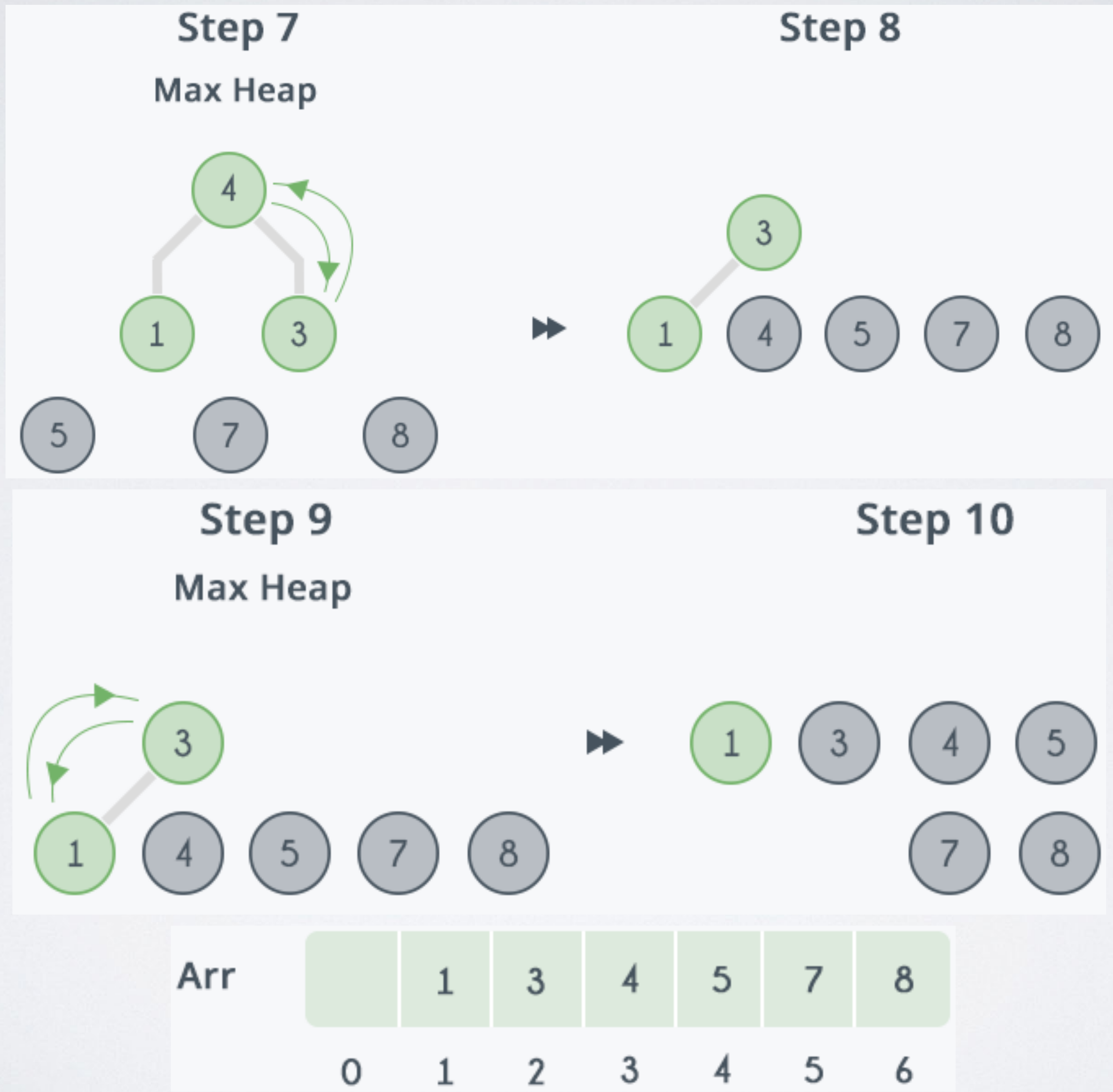


Heap Sort - Example 2





Heap Sort - Example 2



Bucket Sort (Non comparison sorting algorithm)

- For bucket sort to work, **extra information** must be available.
- The **input** A_1, A_2, . . . , A_N must consist of **only positive integers smaller than *M***.
- Keep an **array** called count, **of size *M***, which is **initialized to all 0s**.
- Thus, count has *M* cells, or buckets, which are initially empty.
- When A_i is read, **increment count[A_i] by 1**.
- After all input is read, scan the count array, printing out a representation of the sorted list.
- This algorithm takes **O(*M*+*N*)**. If M is O(N), then running time will be **O(*N*)**.

Radix Sort (Non comparison sorting algorithm)

- Although bucket sort seems like much too trivial an algorithm to be useful, it turns out that there are **many cases** where the **input size is only small integers**, so that using a method like quick sort is really overkill. One such example is radix sort.
- Suppose we have 10 numbers in the range 0 to 999 that we would like to sort.
- Obviously we cannot use bucket sort; there would be too many buckets.

Radix Sort (Non comparison sorting algorithm)

- The trick is to use several passes of bucket sort. The natural algorithm would be to bucket sort by the most significant "digit" (digit is taken to base b), then next most significant, and so on.
- But a simpler idea is to perform bucket sorts in the reverse order, starting with the least significant "digit" first.
- Of course, more than one number could fall into the same bucket, and unlike the original bucket sort, these numbers could be different, so we keep them in a list.

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

TODO

- At the end of the PS, we investigated C++ codes of four comparison based sorting algorithm (i.e., insertion sort, heap sort, merge sort, and quick sort with two different pivot selection algorithms).
- Your task is to enlarge the input array size and include reverse sorted list additional to the random and sorted input lists.
- Then, test the run time of the algorithms again as we did in the PS.