Chapter 12:
A Menagerie of Sorts

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

Sorting Applications

- E-mail, songs, contact book…
- Graphics: sorting the elements of a scene so that they are painted in the right order (painter's algorithm).
- Computational biology.
- Data compression.
- …

History

- The first application of sorting aided by machines was in the 1880s.
- Herman Hollerith designed a tabulating machine to assist in the counting of the U.S. census in 1890.
- At that point population growth had ensured that manual counting would require 13 years to complete—well into the next scheduled census of 1900.

Outline

General Idea

- The simplest sorting method is probably *selection sort*.
- In that method, we set out to find the minimum element.
- When we find it, we put it in the first position.
- Following that, we look for the minimum element of the remaining items.
- When we find it, we put it into the second position.
- We proceed like this until all elements are in the right position.

Selection Sort

Algorithm: Selection sort.

`SelectionSort`(A)

Input: A, an array of items to be sorted

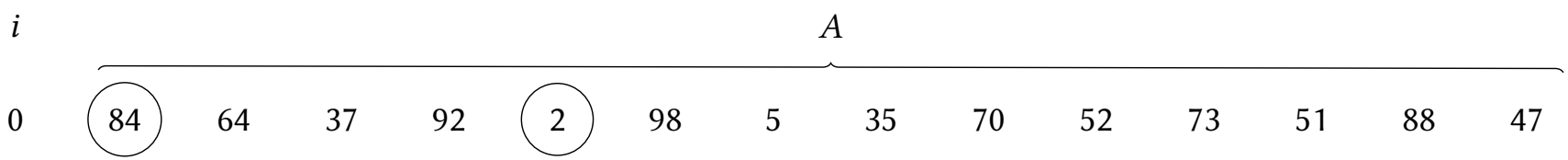
Result: A is sorted

```
1  for i ← 0 to |A| − 1 do
2      m ← i
3      for j ← i + 1 to |A| do
4          if Compare(A[j], A[m]) < 0 then
5              m ← j
6      Swap(A[i], A[m])
```

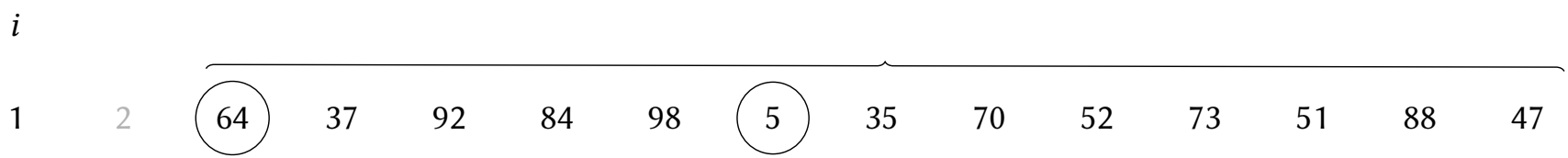
Explanation

- In each iteration we seek the ith smallest element of the array A (lines 1–6).
- To find the ith smallest element of A we use an inner loop, going from element $i + 1$ to the end of the array (lines 3–5).
- We store the position of the ith smallest element in variable m.
- When the loop of lines 3–5 is finished, we exchange the ith smallest element, which is at position m, with the element at position i.

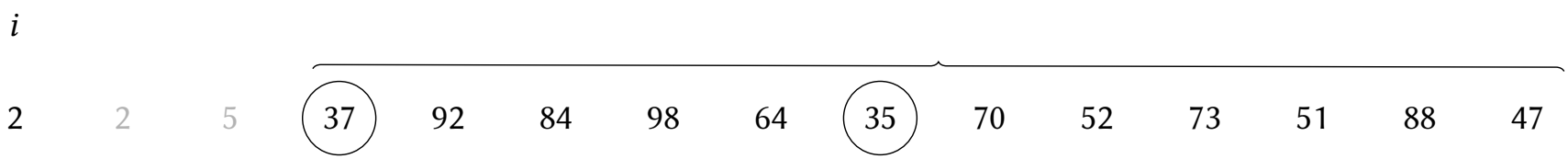
Selection Sort Example (1)



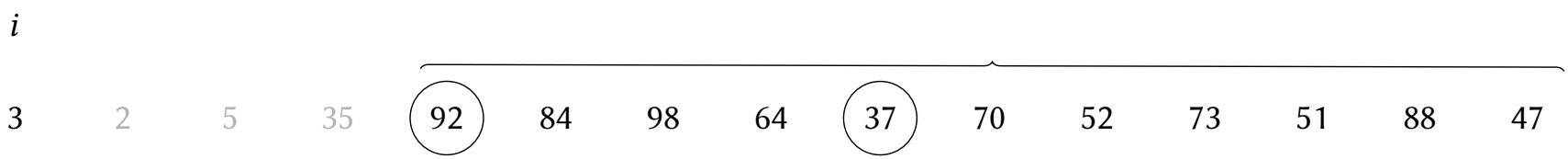
Selection Sort Example (2)



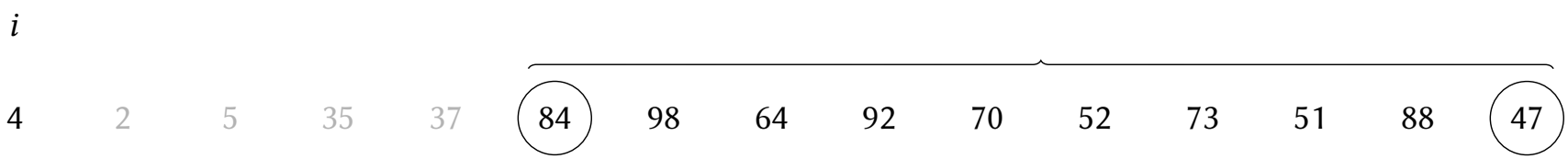
Selection Sort Example (3)



Selection Sort Example (4)

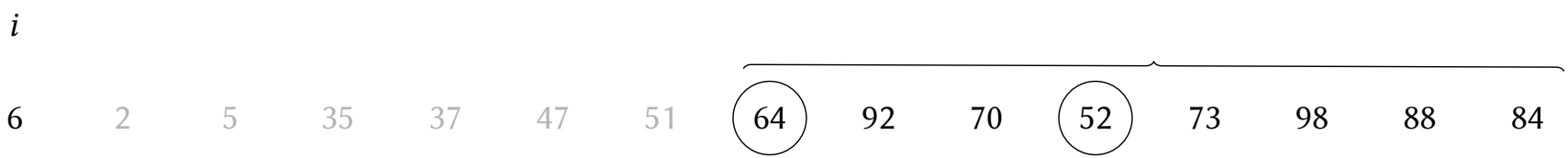


Selection Sort Example (5)



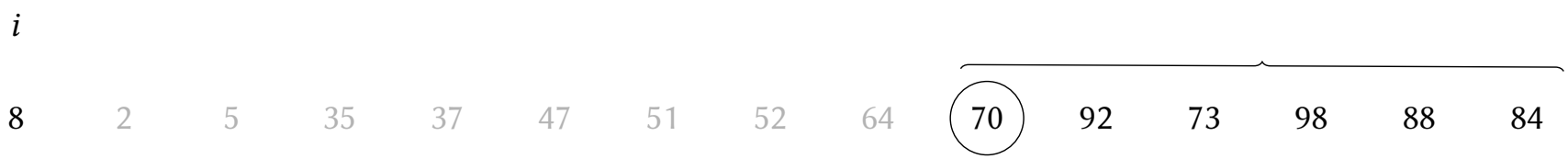
Selection Sort Example (6)

Selection Sort Example (7)

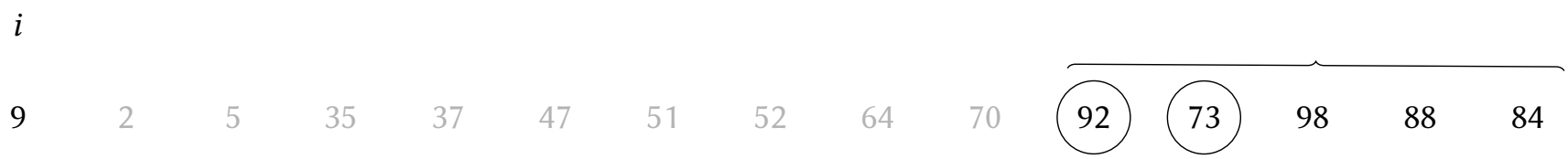


Selection Sort Example (8)

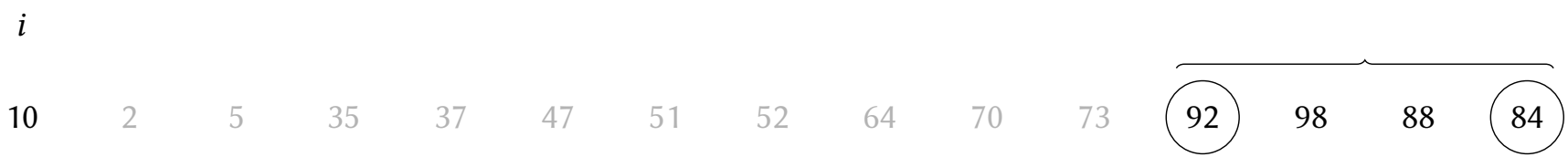
Selection Sort Example (9)



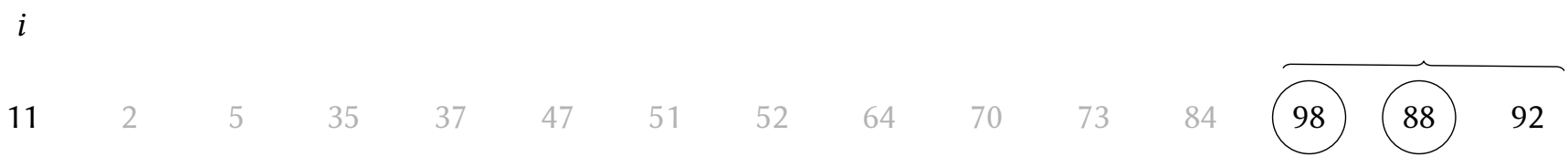
Selection Sort Example (10)



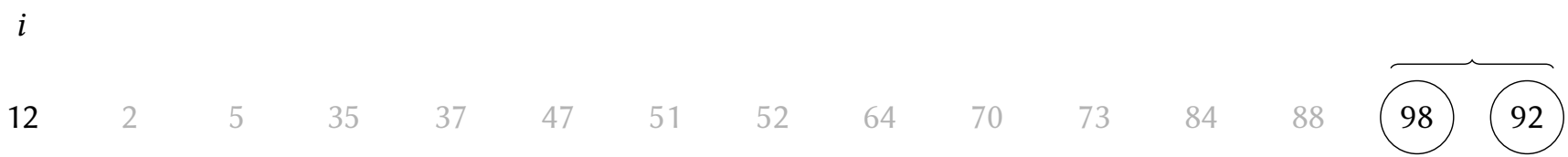
Selection Sort Example (11)



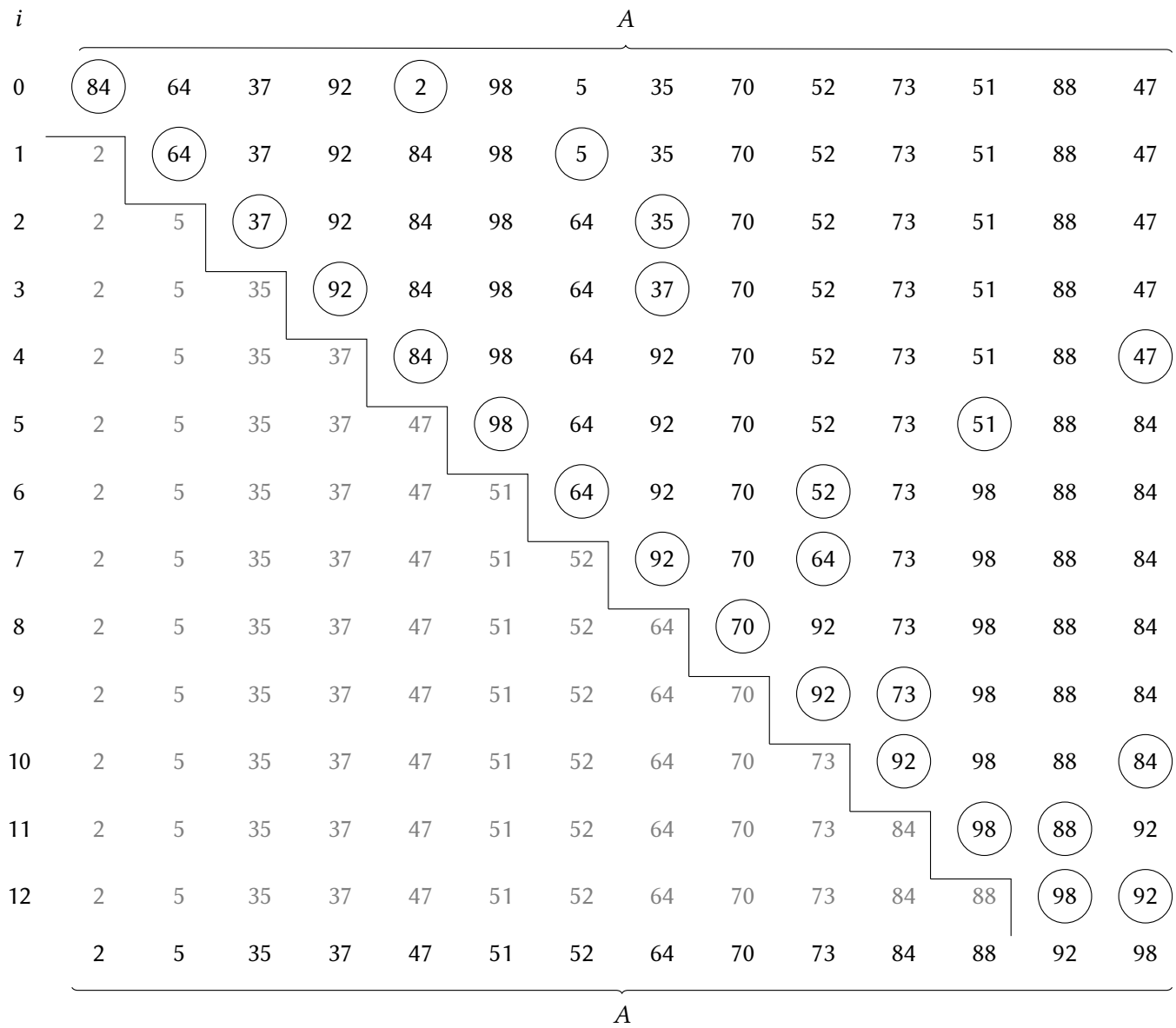
Selection Sort Example (12)



Selection Sort Example (13)



Selection Sort Example



Selection Sort Complexity (1)

- Suppose we have $|A| = n$ items.
- The outer loop is executed $n - 1$ times, so we have $n - 1$ exchanges.
- In the inner loop we compare all items from $i + 1$ to the end of A.
- In the first pass we compare all elements from $A[1]$ to $A[n - 1]$, so we have $n - 1$ comparisons.
- In the second pass we compare all items from $A[2]$ to $A[n - 1]$, so we have $n - 2$ comparisons.
- In the last pass we compare item $A[n - 2]$ with item $A[n - 1]$, so we have 1 comparison.

Selection Sort Complexity (2)

The total number of comparisons is:

$$1 + 2 + \cdots + (n-1) = 1 + 2 + \cdots + (n-1) + n - n$$
$$= \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}$$

Selection Sort Complexity (3)

- Therefore, the complexity of selection sort is:

$$\Theta(n-1) = \Theta(n)$$

exchanges and

$$\Theta(n(n-1)/2) = \Theta(n^2)$$

comparisons.

- In general we treat comparisons and exchanges separately because they can have very different computational cost.
- Comparisons are usually faster than exchanges, because they do not involve moving data around.

Selection Sort with no Unnecessary Exchanges

Algorithm: Selection sort.

`SelectionSort`(A)

Input: A, an array of items to be sorted

Result: A is sorted

```
1  for i ← 0 to |A| − 1 do
2      m ← i
3      for j ← i + 1 to |A| do
4          if Compare(A[j], A[m]) < 0 then
5              m ← j
6      Swap(A[i], A[m])
```

We can observe that in line 6 we perform an exchange even when this is not required (because $i = m$). We can avoid that with a simple check.

Selection Sort with no Unnecessary Exchanges Algorithm

Algorithm: Selection sort with no unnecessary exchanges.

`SelectionSortCheckExchanges`(A)

Input: A, an array of items to be sorted

Result: A is sorted

```
1  for i ← 0 to |A| − 1 do
2      m ← i
3      for j ← i + 1 to |A| do
4          if Compare(A[j], A[m]) < 0 then
5              m ← j
6      if i ≠ m then
7          Swap(A[i], A[m])
```

Is it a Better Algorithm?

- Now the complexity is $O(n)$ exchanges and $\Theta(n^2)$ comparisons.
- But this does not mean that the algorithm runs better in practice.
- The reason is that we add n comparisons in line 6.
- Therefore if the time spent in the extra comparisons is greater than the time saved from the unnecessary exchanges, we will not see an improvement.

Features of Selection Sort

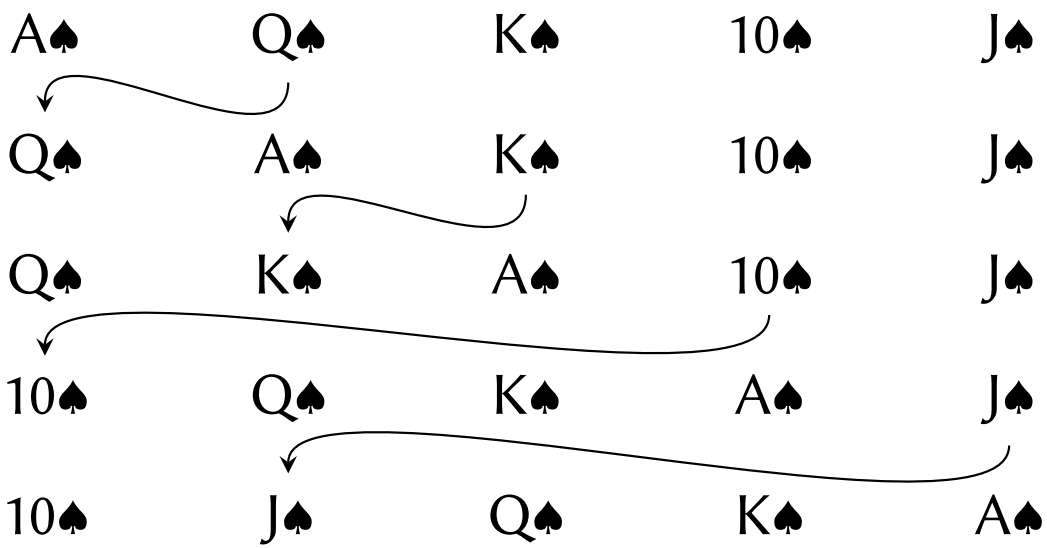
- Simple algorithm.
- Straightforward to implement.
- Does not require many exchanges. That may be important if moving data around is computationally expensive
- It is suitable for small arrays, where it will perform fast enough, but it is not used for large arrays, where better sorting algorithms will deliver much better results.

Outline

Basic Idea

- The basic idea of insertion sort is the one we used by card players to sort their hand.
- We take the first card.
- Then we take the second card.
- We put it in the correct place with respect to the first card.
- We take the third card.
- We put it in the correct place with respect to the second card.
- ...

Insertion Sort Example



Insert Sort

Algorithm: Insertion sort.

`InsertionSort`(A)

Input: A, an array of items to be sorted

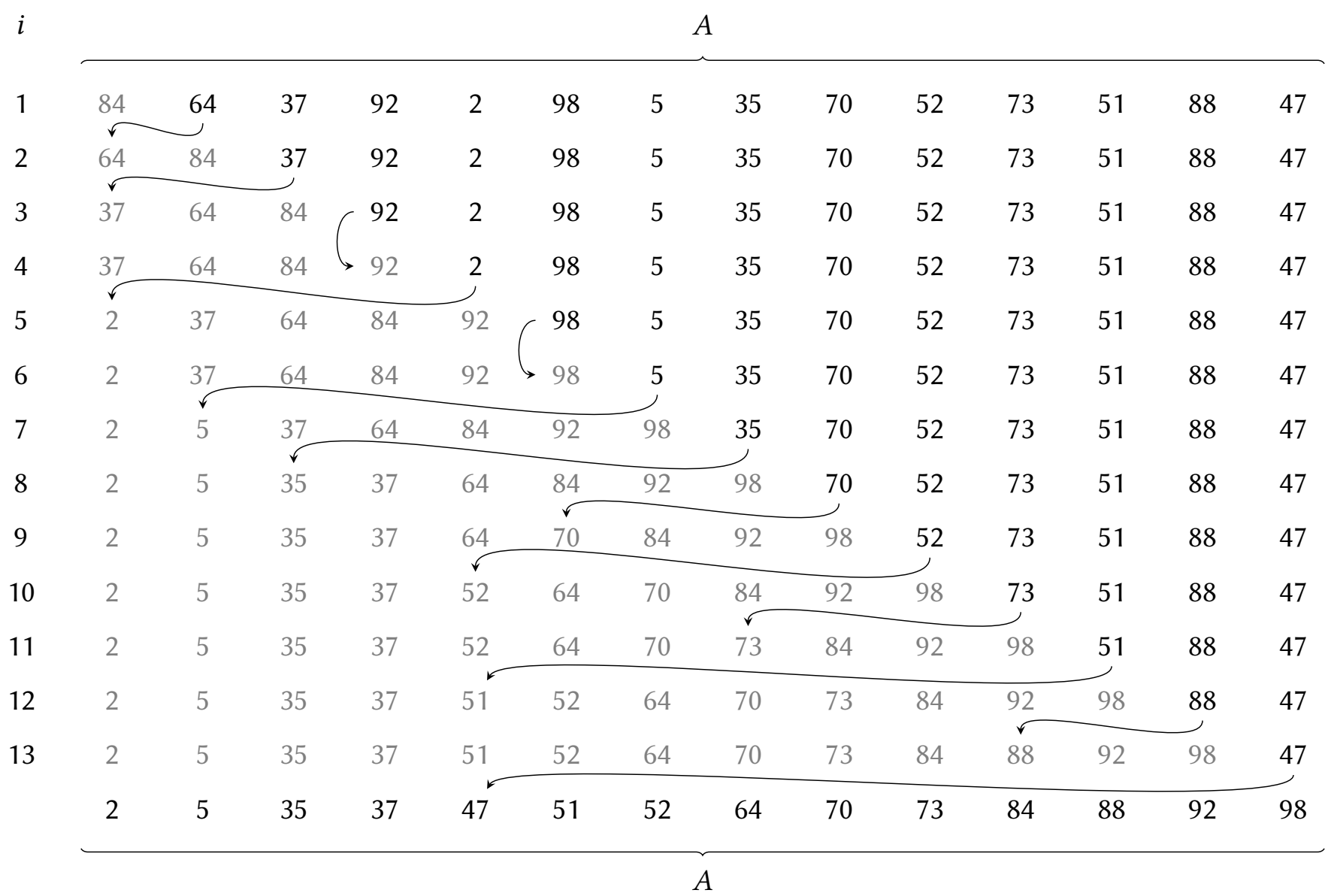
Result: A is sorted

```
1  for i ← 1 to |A| do
2      j ← i
3      while j > 0 and Compare(A[j − 1], A[j]) > 0 do
4          Swap(A[j], A[j − 1])
5          j ← j − 1
```

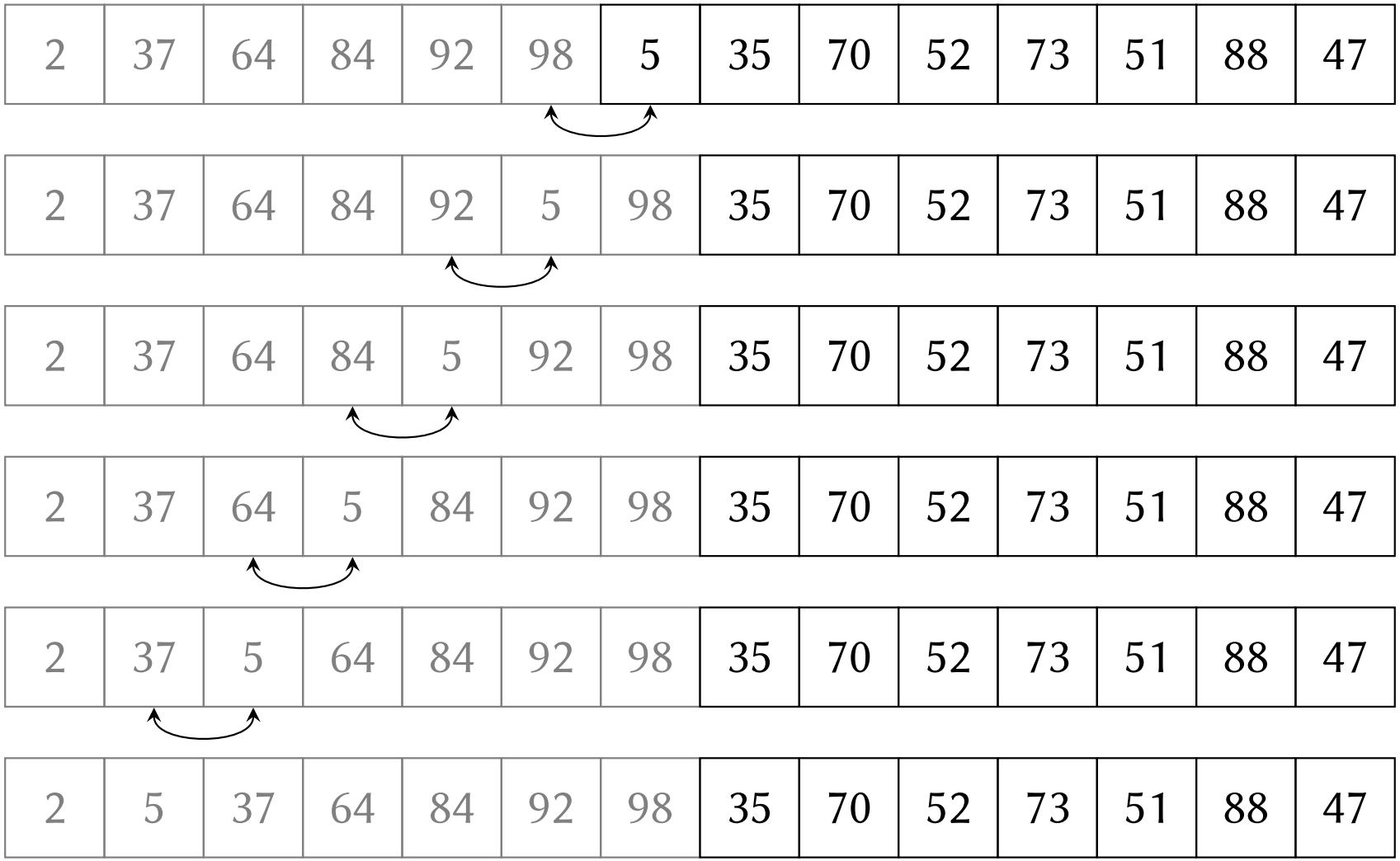
Explanation

- For each element at position i of array A, we go towards the beginning.
- Every time an element is greater than its predecessor, we exchange them.

Insertion Sort Example



Zoom in $i = 6$


2 37 64 84 92 98 5 35 70 52 73 51 88 47
2 37 64 84 92 5 98 35 70 52 73 51 88 47
2 37 64 84 5 92 98 35 70 52 73 51 88 47
2 37 64 5 84 92 98 35 70 52 73 51 88 47
2 37 5 64 84 92 98 35 70 52 73 51 88 47
2 5 37 64 84 92 98 35 70 52 73 51 88 47

Insertion Sort Complexity

- If the elements are already sorted, we need 0 exchanges and $\Theta(n-1) = \Theta(n)$ comparisons.
- If the elements are sorted in reverse order, we need $\Theta(n(n-1)/2) = \Theta(n^2)$ exchanges and $\Theta(n(n-1)/2) = \Theta(n^2)$ comparisons.
- If the elements are in random order, then we need $\Theta(n(n-1)/4) = \Theta(n^2)$ exchanges and $\Theta(n(n-1)/4) + O(n) = \Theta(n(n-1)/4) = \Theta(n^2)$ comparisons.

Features of Insertion Sort

- Simple algorithm.
- Straightforward to implement.
- Can be used as an online algorithm, as it sorts the elements without requiring to have them all available.
- However, it is slow for large amounts of data.

Sorting in n^2 time

- If we have one million elements, n^2 is one trillion
- If we have 100,000 elements, n^2 is 10 billion.

Outline

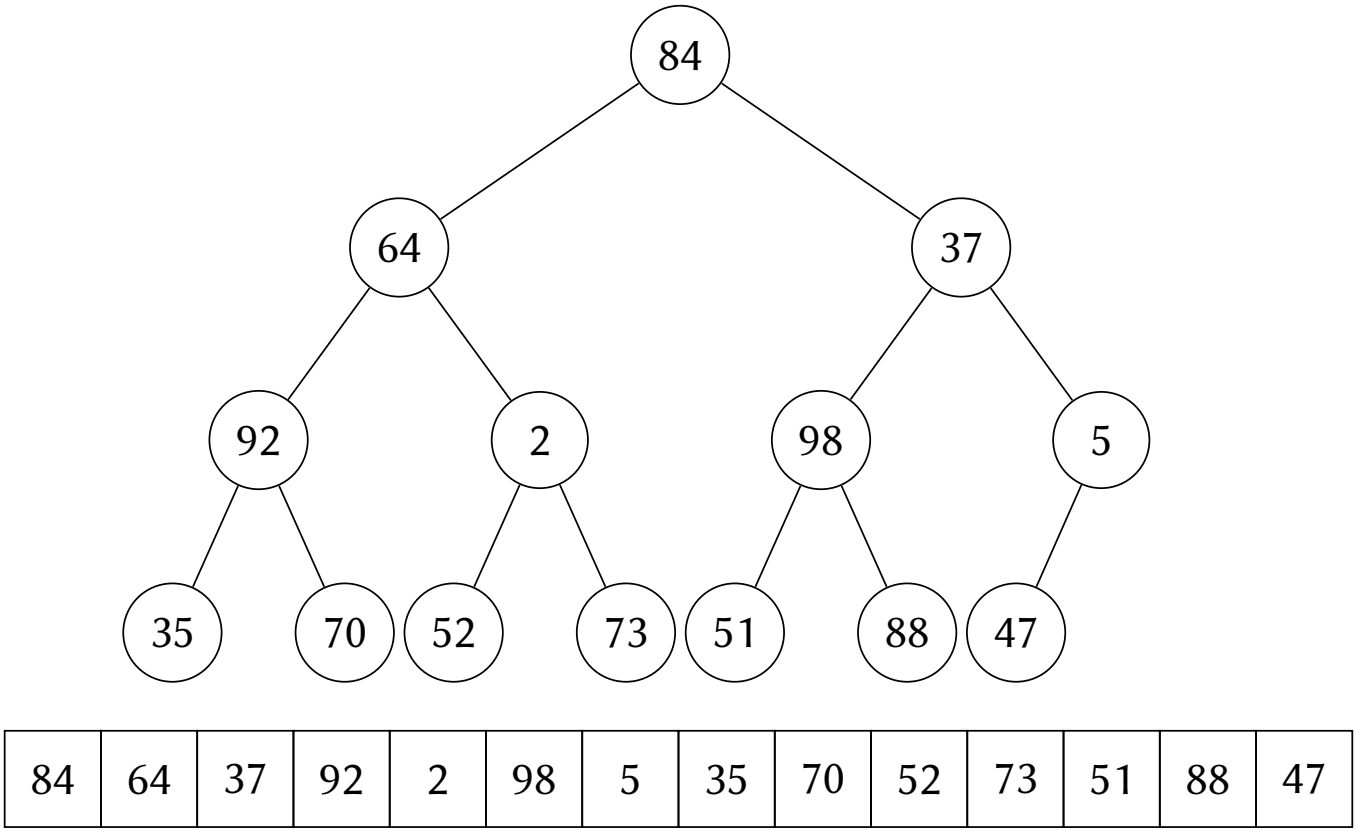
Basic Idea

- In selection sort, each time we search for the smallest of the unsorted items.
- In *heapsort* we search each time for the maximum element and we put it at the end.
- To do that efficiently we need a way to find the maximum among the unsorted elements.

Basic Idea (Contd.)

- Suppose we have arranged the elements of A so that we have $A[0] \geq A[i]$, for all $i < |A|$.
- Then $A[0]$ is the maximum element.
- We exchange it with the $A[n-1]$ element; now the maximum element is at its correct position.
- If we manage to have again $A[0] \geq A[i]$, for all $i < |A| - 1$, then we can proceed in the same way.
- We exchange $A[0]$, which is the maximum of $A[0], A[1], \ldots, A[n-2]$, with $A[n-2]$.
- We do the same thing for the remaining elements.

Correspondence between Array and Tree

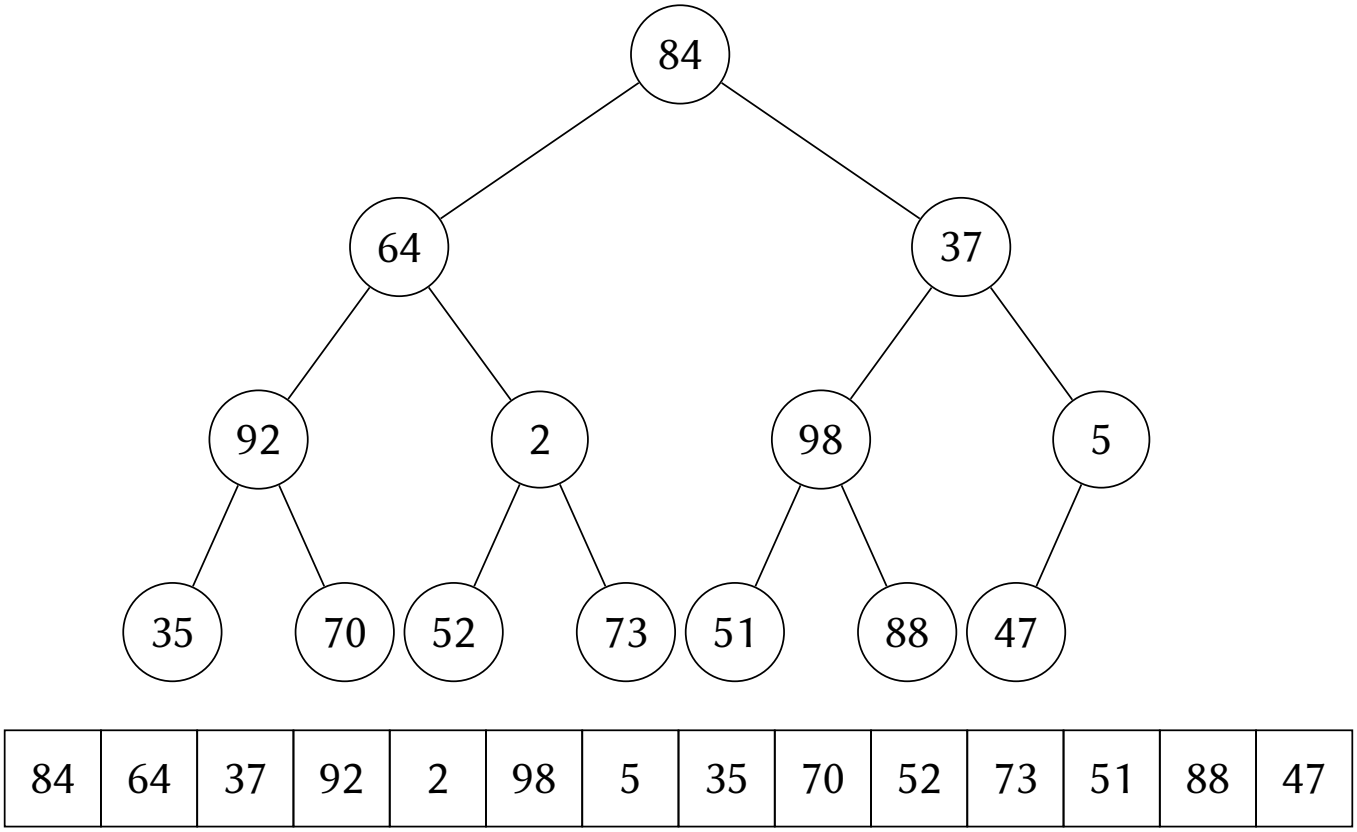


We will treat array A as a tree, where the element $A[i]$ corresponds to a node whose children are elements $A[2i + 1]$ and $A[2i + 2]$.

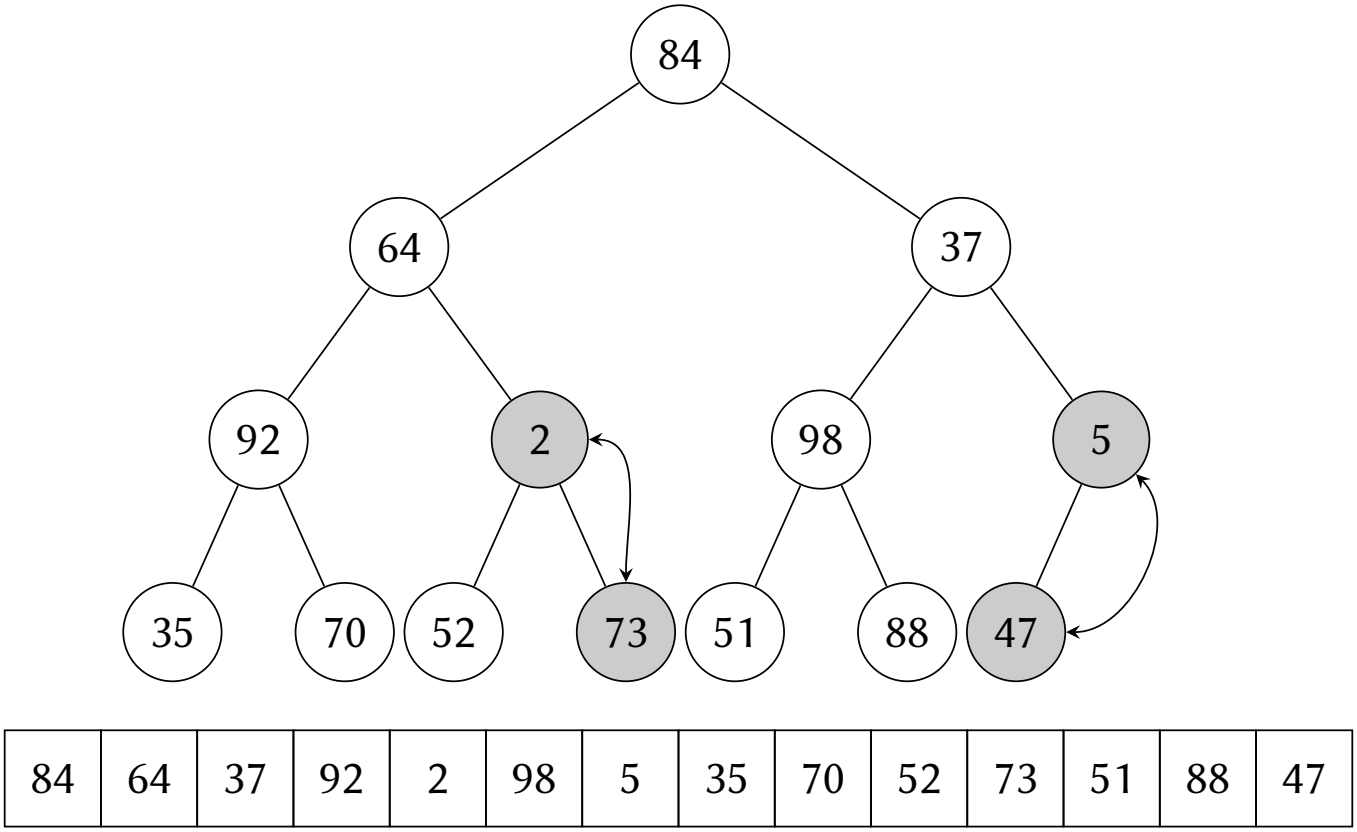
Heaps

- If for all i we have $A[i] \geq A[2i + 1]$ and $A[i] \geq A[2i + 2]$, each node of the tree is greater than or equal to its children
- Then the root of the tree is the maximum element, which is what we wanted: $A[0] \geq A[i]$, for $i < |A|$.
- Such a structure is a *heap*, in particular a *max-heap*.
- How do we make a heap?
- The construction of a max-heap is the first phase of heapsort.

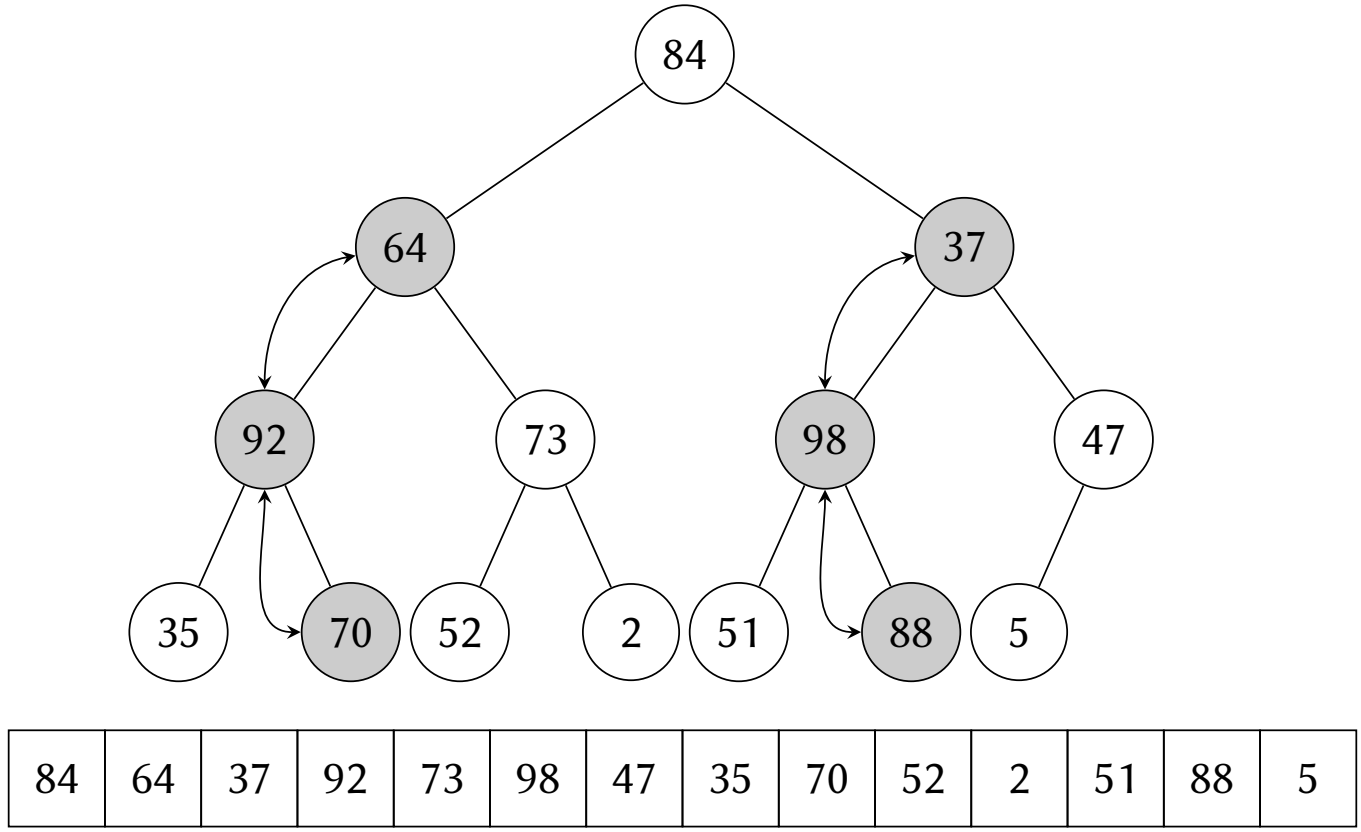
Heap Construction Example (1)



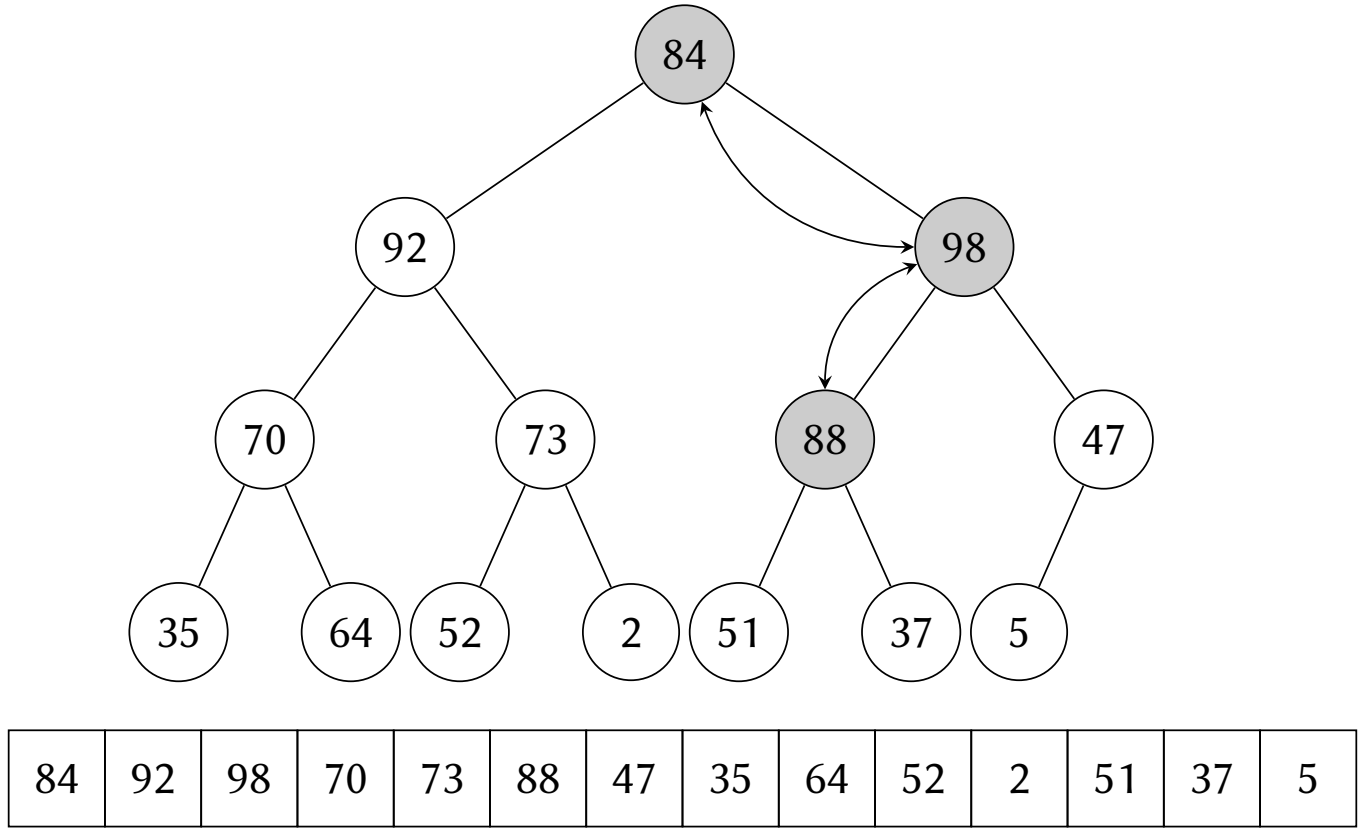
Heap Construction Example (2)



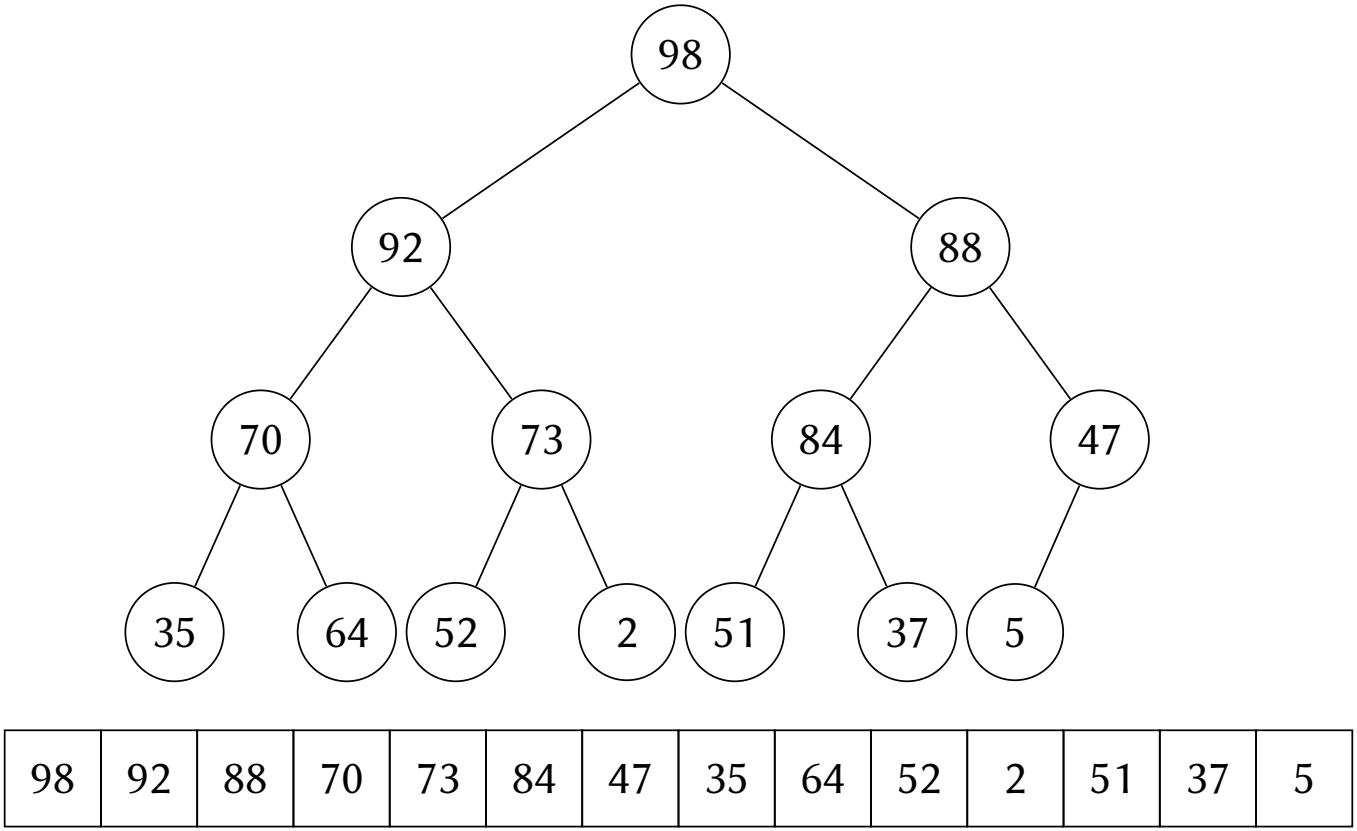
Heap Construction Example (3)



Heap Construction Example (4)



Heap Construction Example (5)



Sink

- We see that every element that is moved sinks to a lower point in the tree, until it is greater than its children.
- For that reason this procedure is called *sink*.

Sink Algorithm

Algorithm: Sink.

`Sink`(A, i, n)

Input: A, a array of items
i, the index of the item to sink into place
n, the number of items we will consider

Result: A with item $A[i]$ sunk into place

```
1  k = i
2  placed ← FALSE
3  j ← 2k + 1
4  while not placed and j < n do
5      if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then
6          j ← j + 1
7      if Compare(A[k], A[j]) ≥ 0 then
8          placed = TRUE
9      else
10         Swap(A[k], A[j])
11     k ← j
12     j ← 2k + 1
```

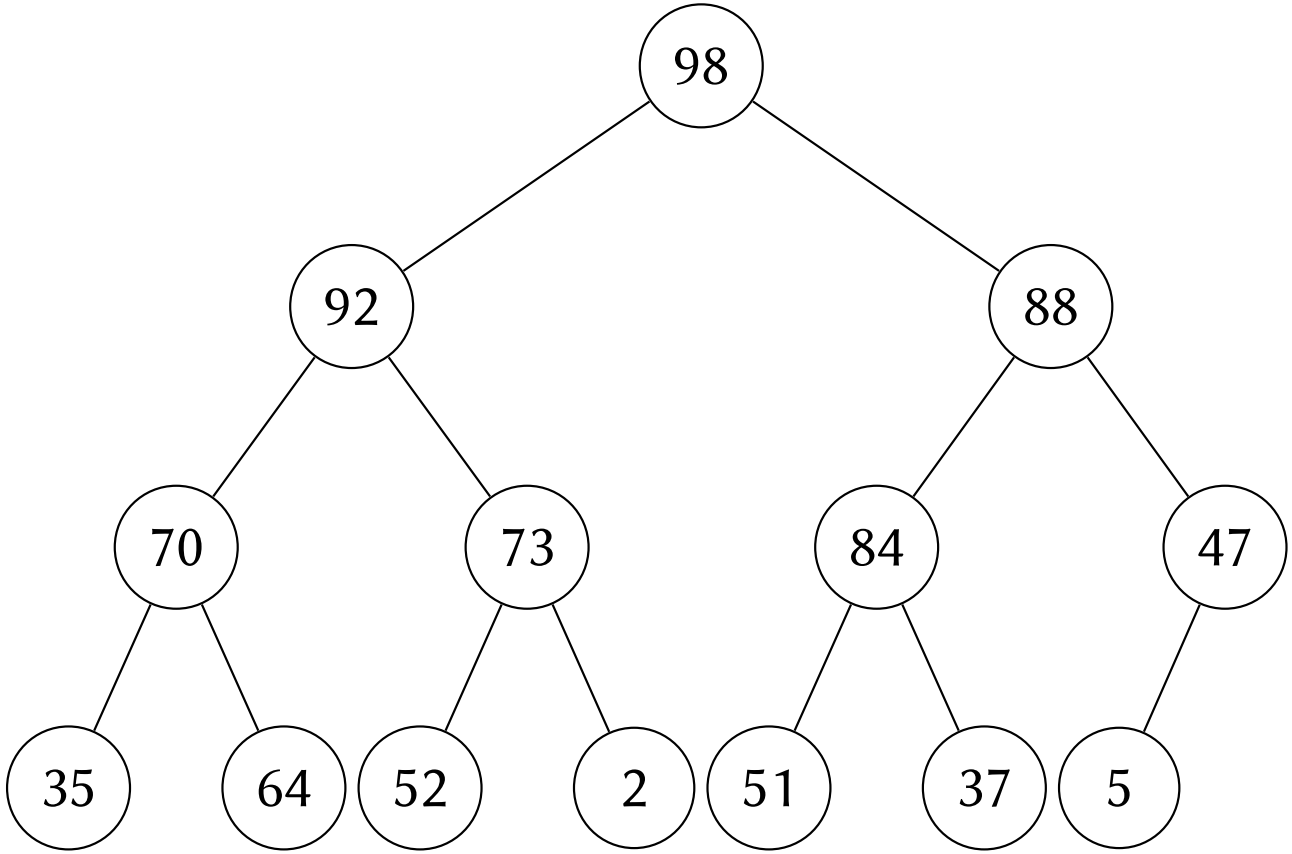
Explanation

- We use k for the position of the element.
- The variable *placed* indicates whether the element has sunk to its right place.
- The children of the element at position k are at positions $2k + 1$ and $2k + 2$, if they exist.
- In lines 5–6 we find the maximum of the children.
- In lines 7–8 we compare the maximum of the children with the element we sink. If it is greater than the maximum of the children, we are done. Otherwise we exchange them in line 10.
- In lines 11–12 we calculate the position of the first child of the node, so that we can repeat the loop (if required).

Second Phase of Heapsort

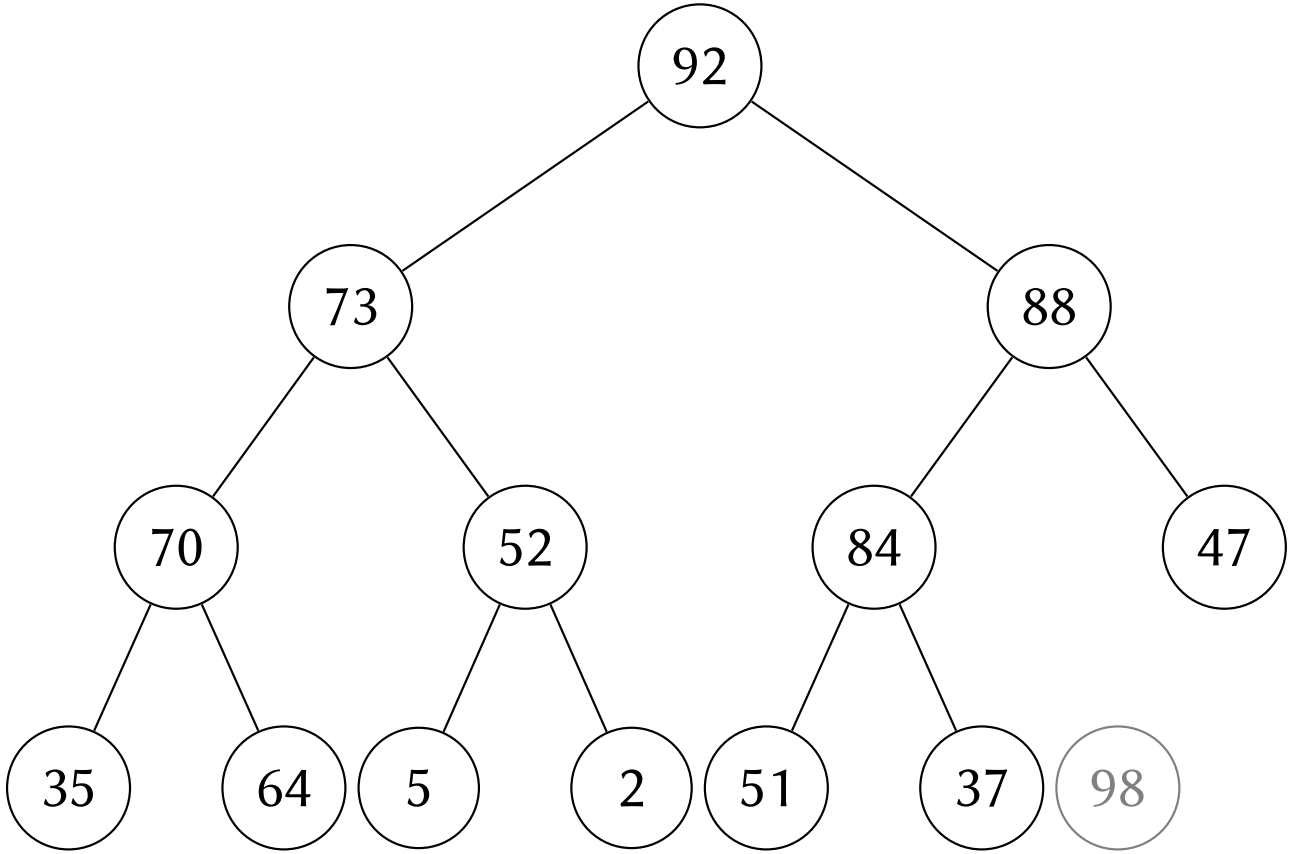
- Once we have a max-heap in place, we proceed to the second phase of heapsort.
- In this phase we take the maximum of the heap, element $A[0]$, and we exchange it with the element at the end of A.
- We rebuild the heap with the first $n-1$ elements of A.
- We take the new $A[0]$ element and we exchange it with the element $A[n-2]$.
- We rebuild the heap with the first $n-2$ of A.
- ...

Heapsort Second Phase Example (1)



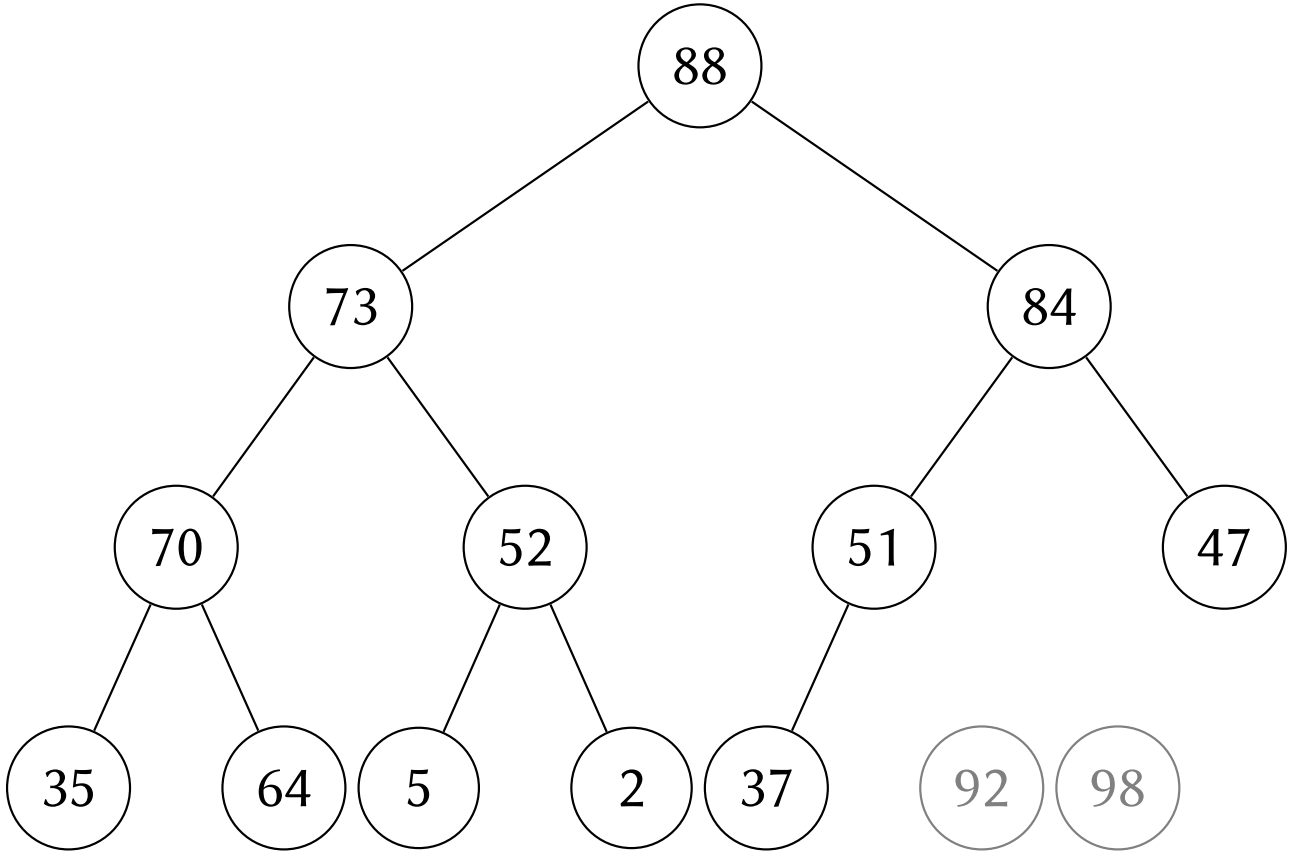
98	92	88	70	73	84	47	35	64	52	2	51	37	5

Heapsort Second Phase Example (2)



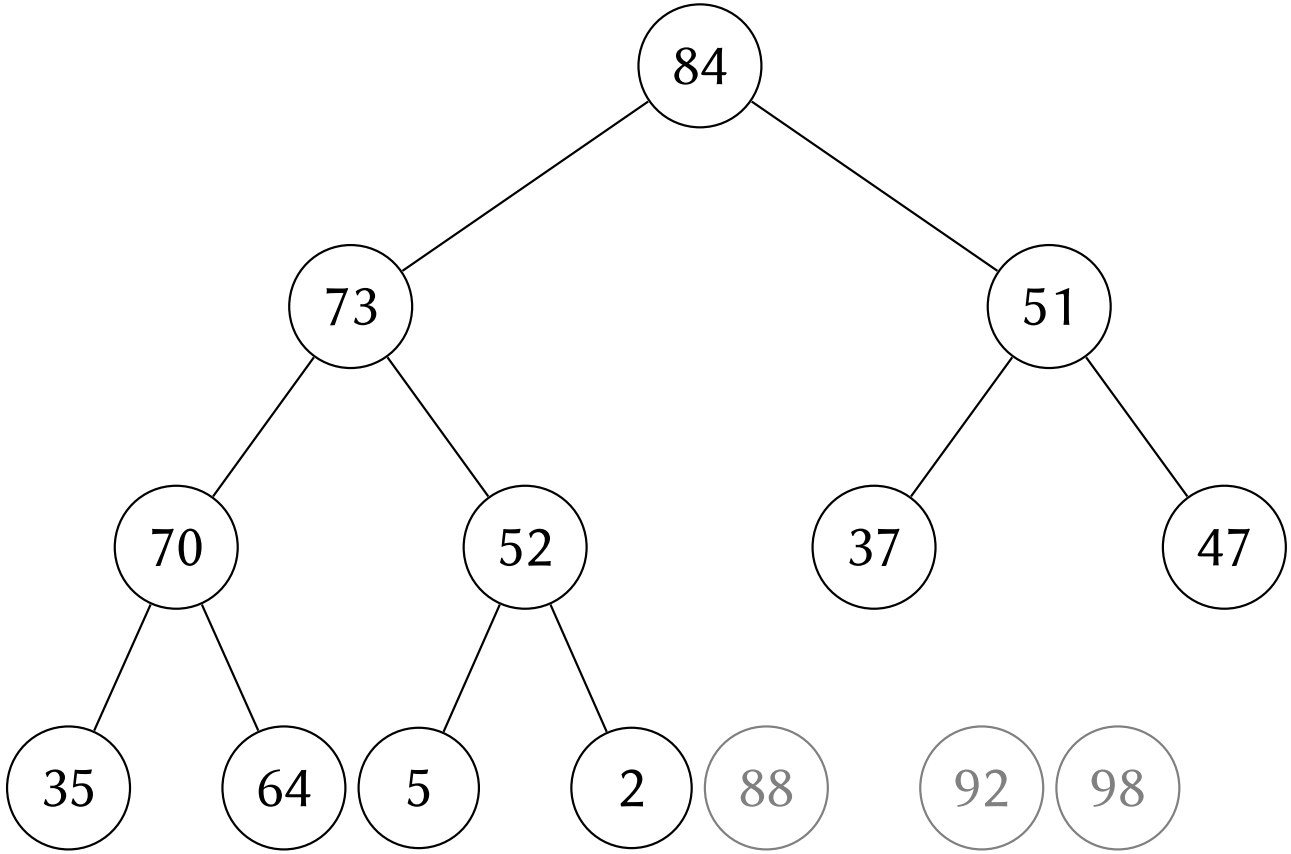
92	73	88	70	52	84	47	35	64	5	2	51	37	98

Heapsort Second Phase Example (3)



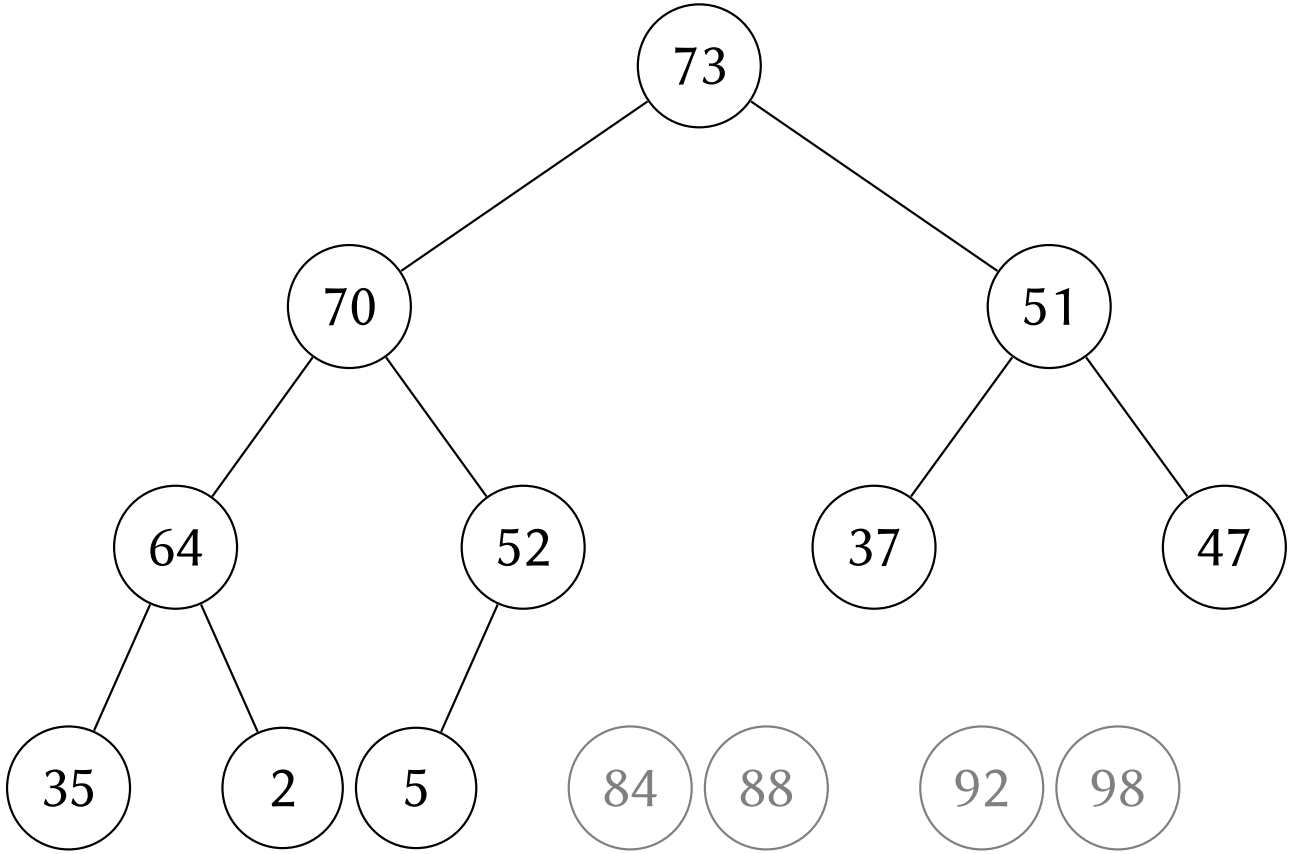
88	73	84	70	52	51	47	35	64	5	2	37	92	98

Heapsort Second Phase Example (4)



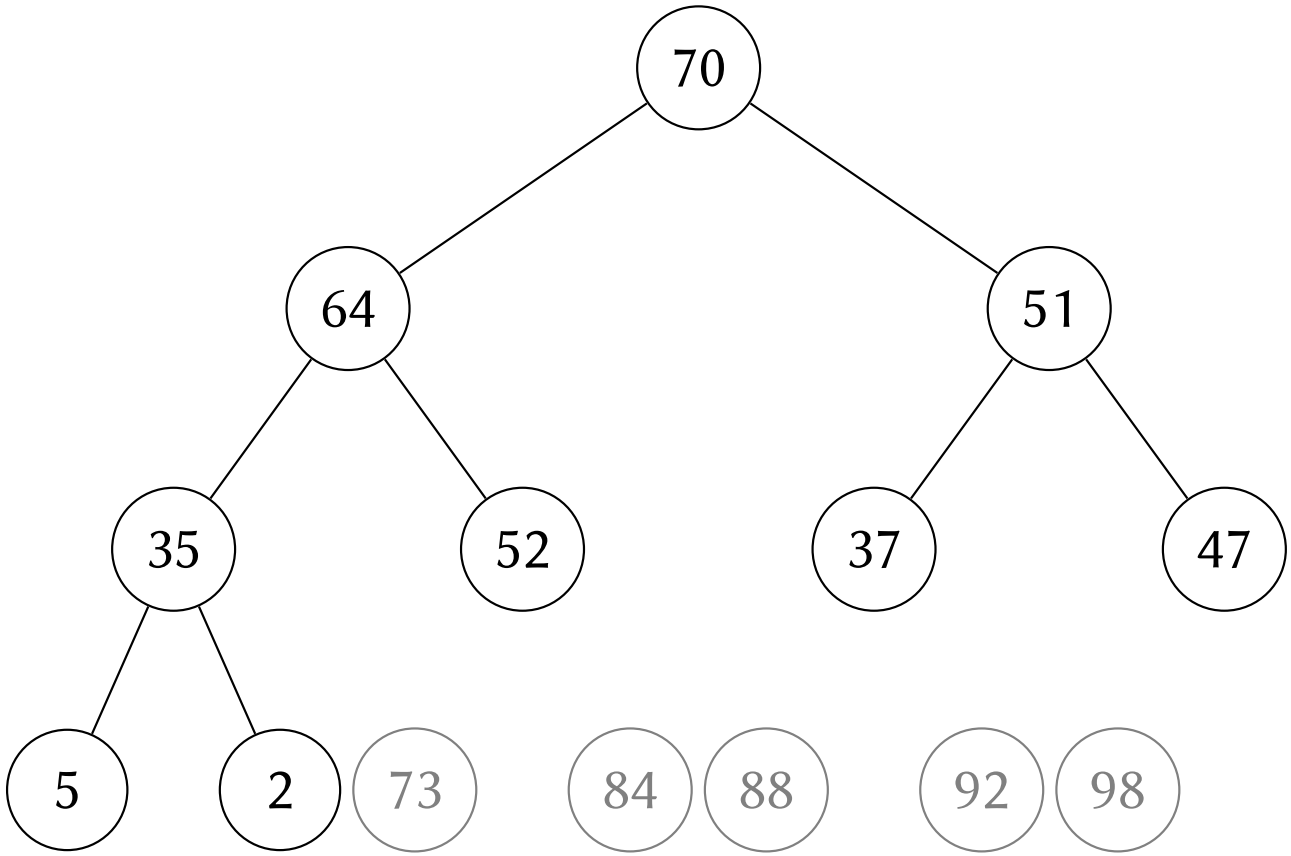
84	73	51	70	52	37	47	35	64	5	2	88	92	98

Heapsort Second Phase Example (5)



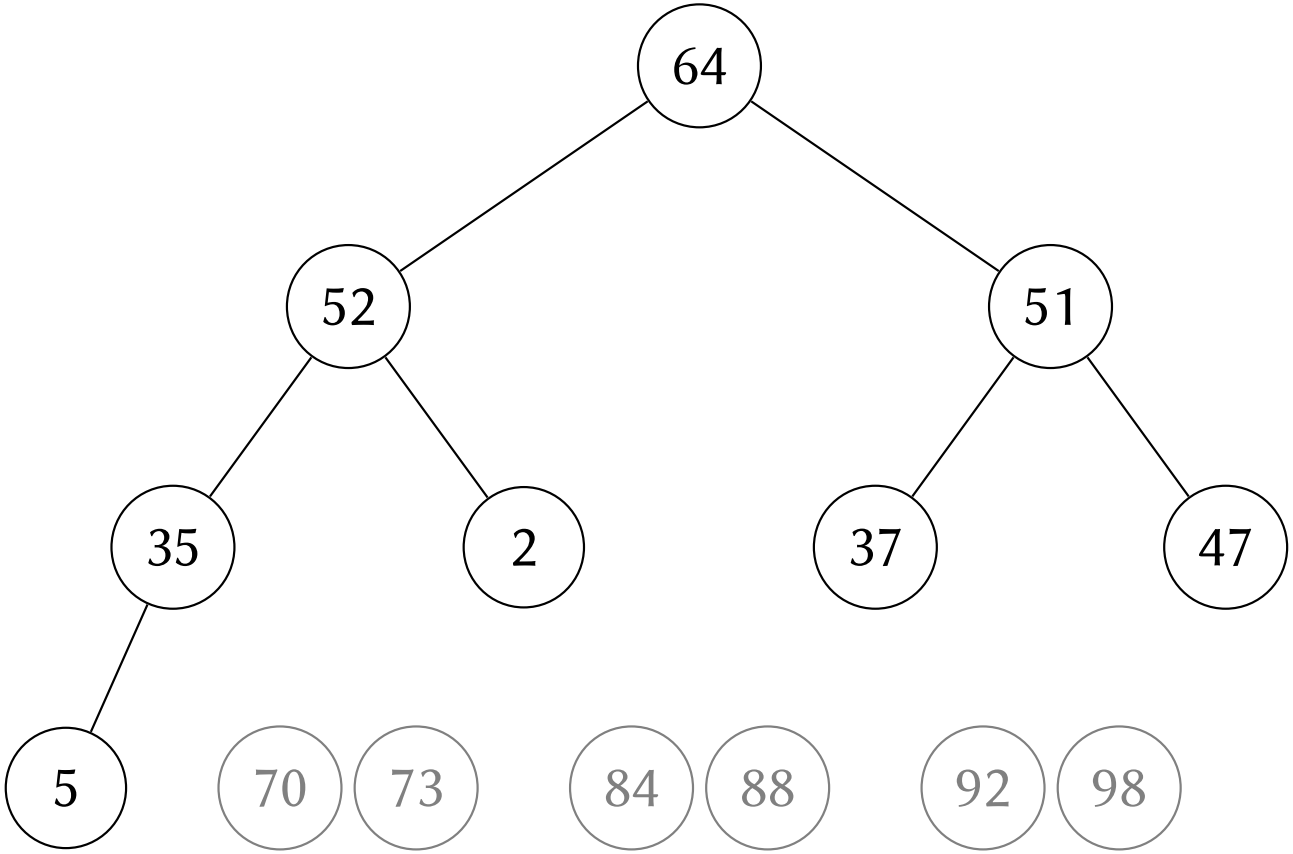
73	70	51	64	52	37	47	35	2	5	84	88	92	98

Heapsort Second Phase Example (6)



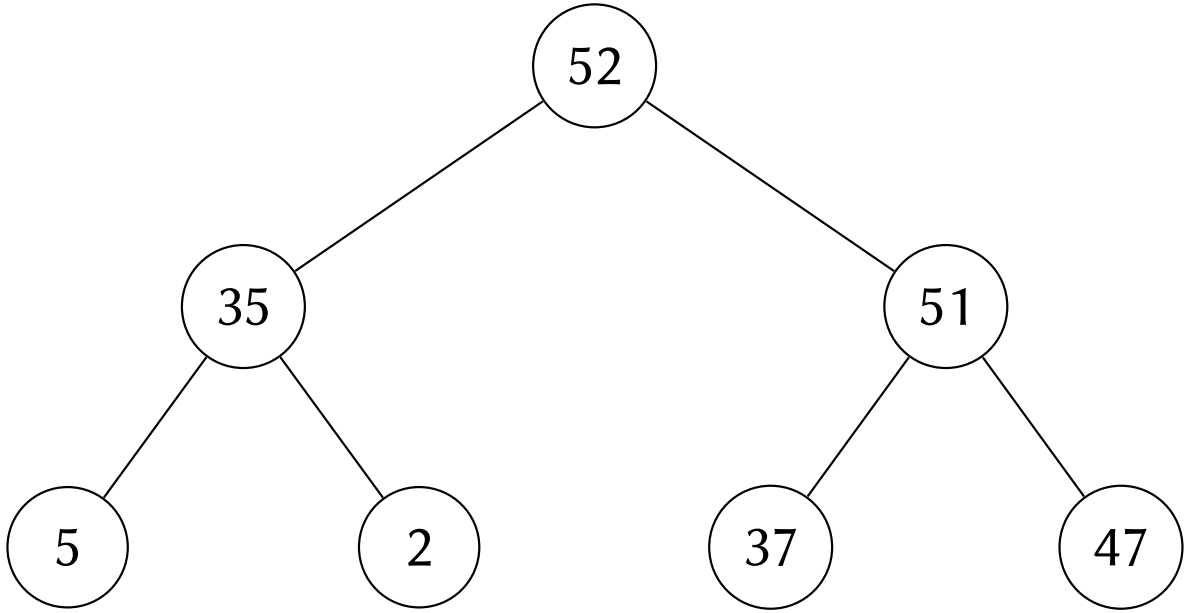
70	64	51	35	52	37	47	5	2	73	84	88	92	98

Heapsort Second Phase Example (7)



64	52	51	35	2	37	47	5	70	73	84	88	92	98

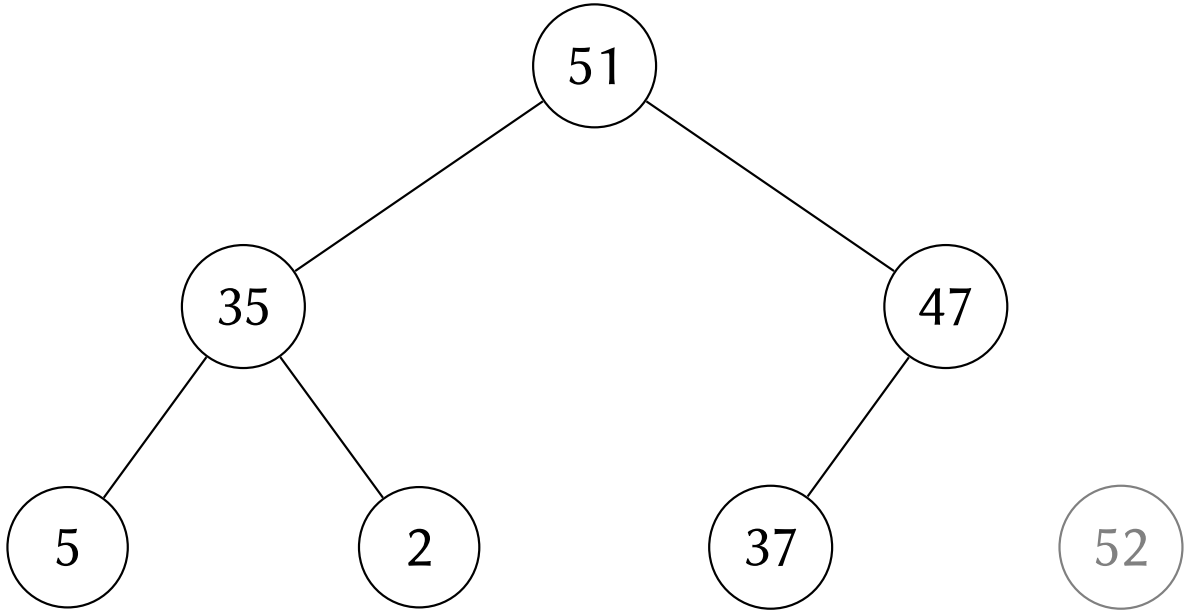
Heapsort Second Phase Example (8)



64 70 73 84 88 92 98

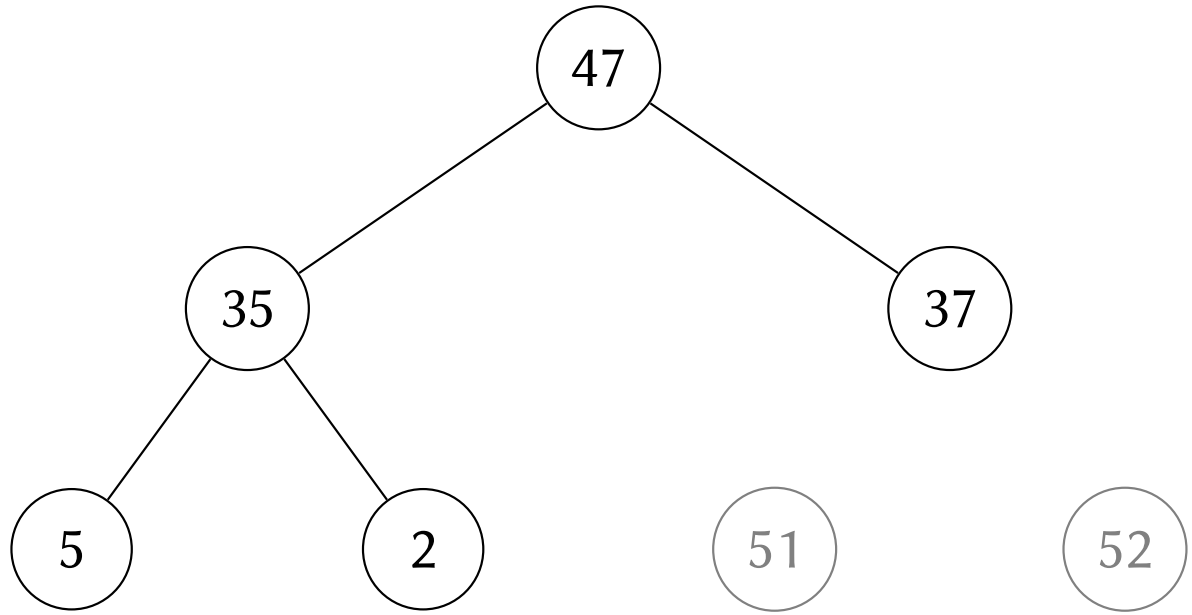
52	35	51	5	2	37	47	64	70	73	84	88	92	98

Heapsort Second Phase Example (9)



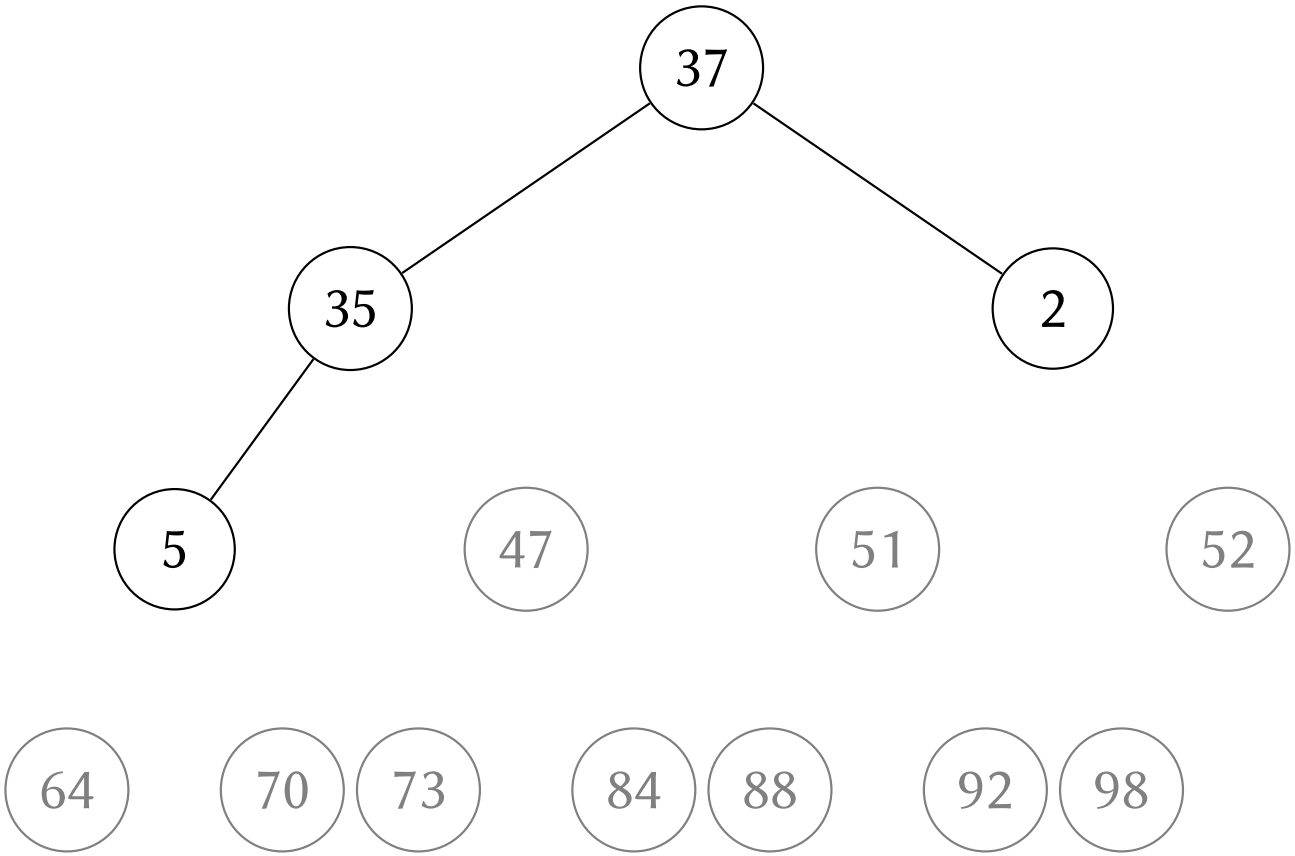
51	35	47	5	2	37	52	64	70	73	84	88	92	98

Heapsort Second Phase Example (10)



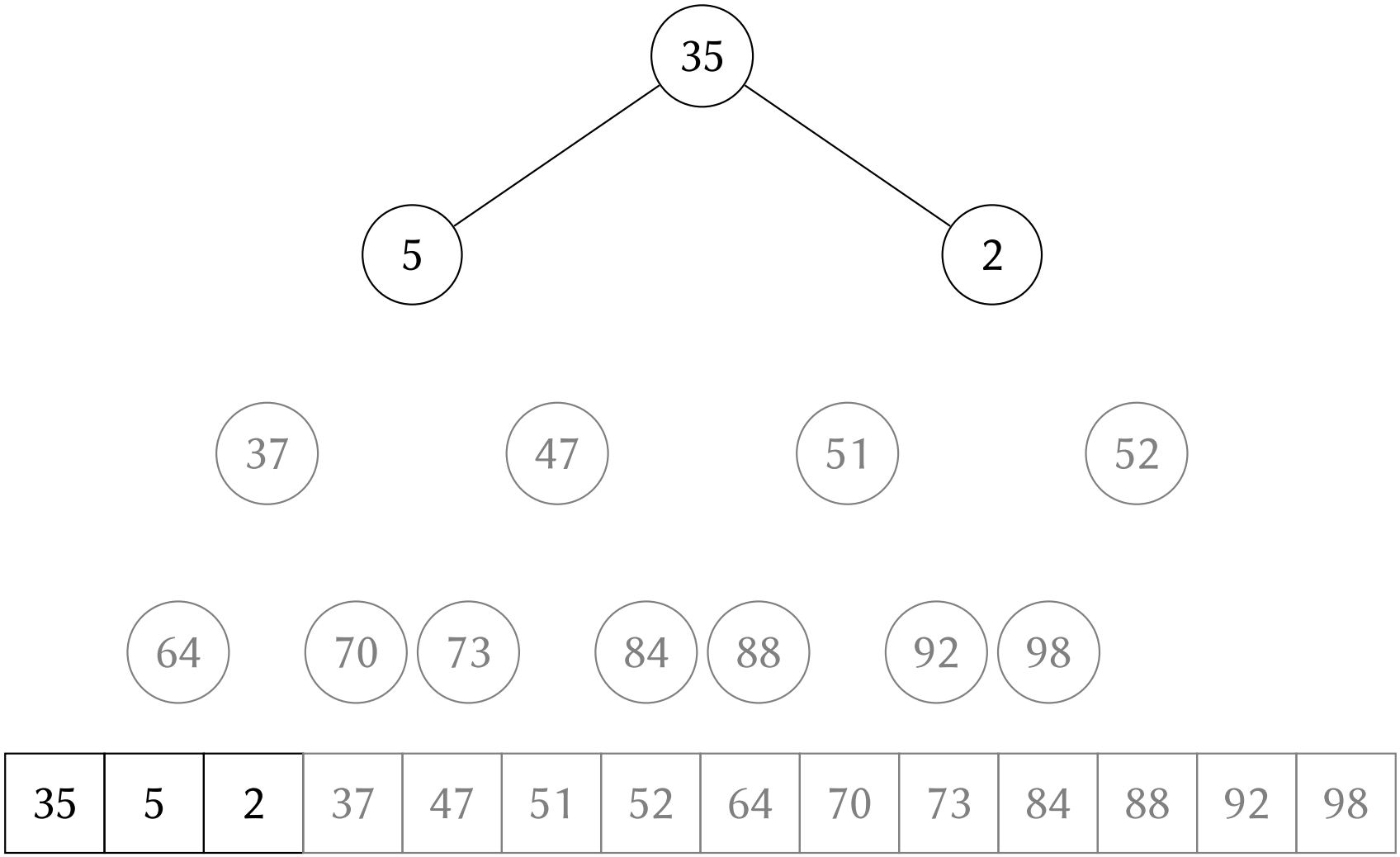
47	35	37	5	2	51	52	64	70	73	84	88	92	98

Heapsort Second Phase Example (11)



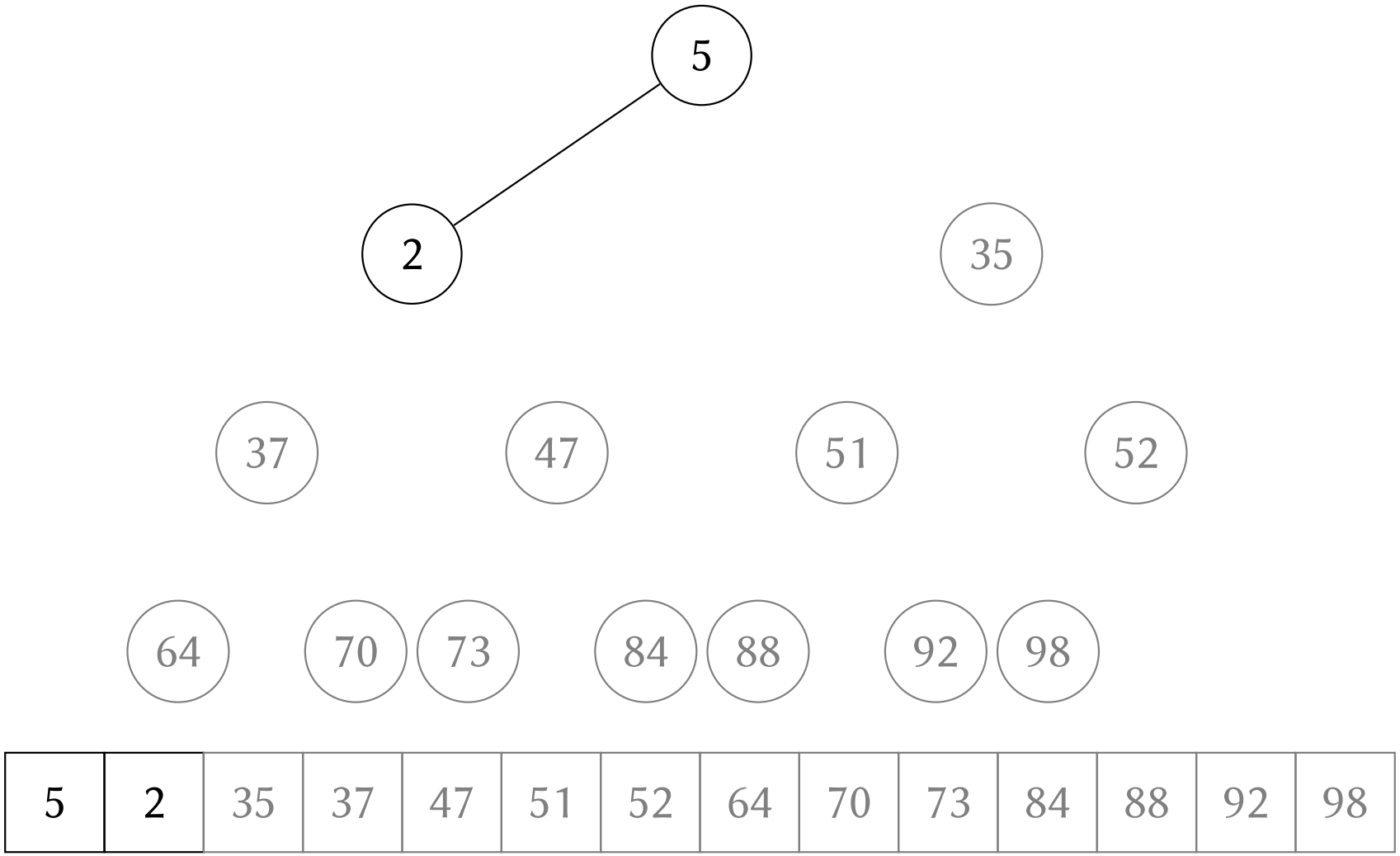
37	35	2	5	47	51	52	64	70	73	84	88	92	98

Heapsort Second Phase Example (12)



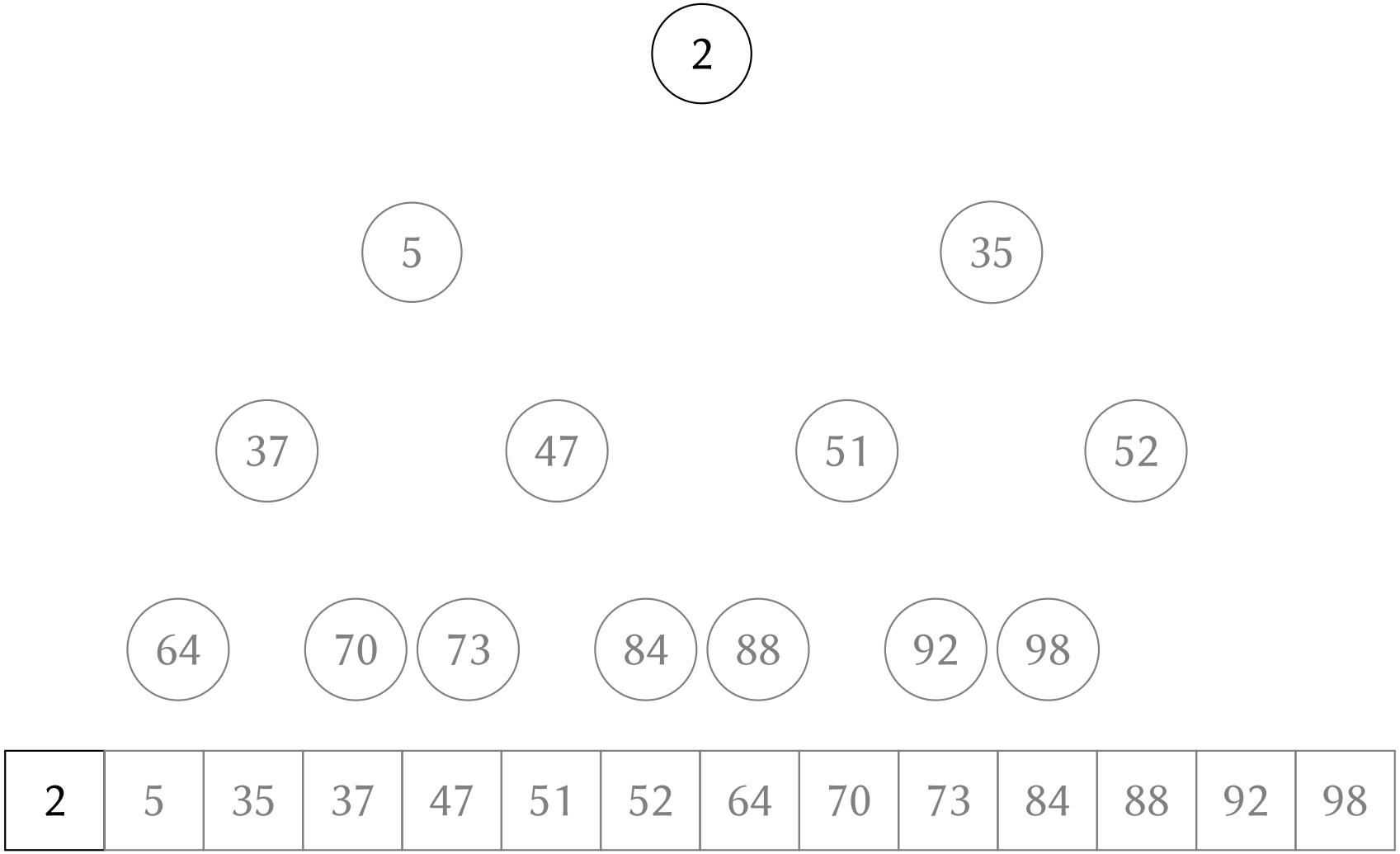
35	5	2	37	47	51	52	64	70	73	84	88	92	98

Heapsort Second Phase Example (13)



5	2	35	37	47	51	52	64	70	73	84	88	92	98

Heapsort Second Phase Example (14)



The Complete Heapsort Algorithm

- The complete heapsort algorithm is the combination of the first and the second phase.
- In the first phase we build a max-heap.
- In the second phase we take the maximum of the heap, we exchange it with the last item, we shorten the heap by one element, and we repeat.

Heapsort Algorithm

Algorithm: Heapsort.

HeapSort(A)

Input: A, an array of items

Result: A is sorted

```
1  n ← |A|
2  for i ← ⌊(n − 1)/2⌋ to −1 do
3      Sink(A, i, n)
4  while n > 0 do
5      Swap(A[0], A[n − 1])
6      n ← n − 1
7      Sink(A, 0, n)
```

Heapsort Complexity

- If we have n elements we need less than $2n \lg n + 2n$ comparisons and $n \lg n + n$ exchanges.
- So we have $O(2n \lg n + 2n) = O(n \lg n)$ comparisons and $O(n \lg n + n) = O(n \lg n)$ exchanges.

Combined Heapsort and Selection Sort

- Heapsort is based on selection sort, but instead of plowing through the elements looking for the minimum, it selects the maximum from an elegant data structure.
- This shows the importance of choosing the correct data structure when we have to solve a problem.

Outline

Basic Idea

- Suppose we have two ordered lists.
- Can we create an ordered list with the elements of the two lists, taking advantage of the fact that they are already ordered?

Basic Idea (Contd.)

- Suppose we have two piles of playing cards.
- We assume that each pile is sorted.
- We put the two piles side-by-side.
- We take the minimum card from the two piles and we put it aside, starting a third pile.
- We take again the minimum card from the two piles (we have taken the minimum from one of them) and we put it in the third pile, after the first card we took out.
- We go on like this until we use up one of the two piles. When this happens, all the cards of the other piles are greater than the ones we have already put aside, so we just put them to the end of the third pile.

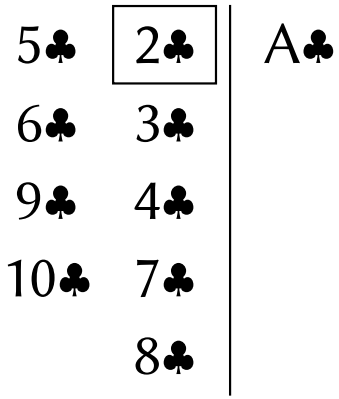
Merge Sort

- This is the basic idea behind *merge sort*.
- To sort a set of elements, we split it in half, we sort each half (we'll see how), and then we merge the two sorted halves.

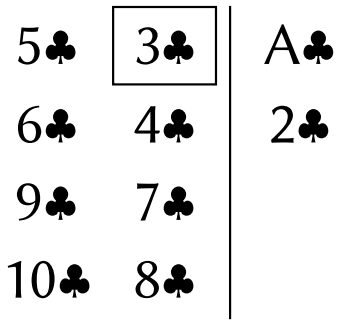
Merge Sort Example (1)

A♣	2♣
5♣	3♣
6♣	4♣
9♣	7♣
10♣	8♣

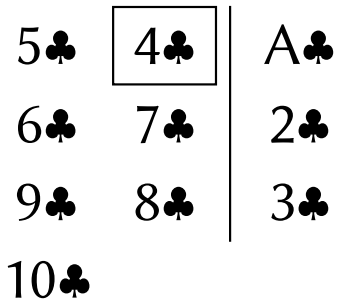
Merge Sort Example (2)



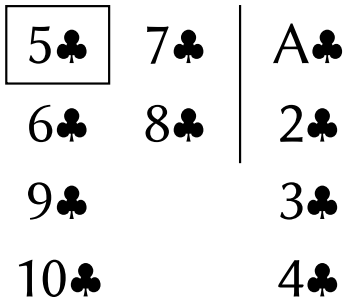
Merge Sort Example (3)



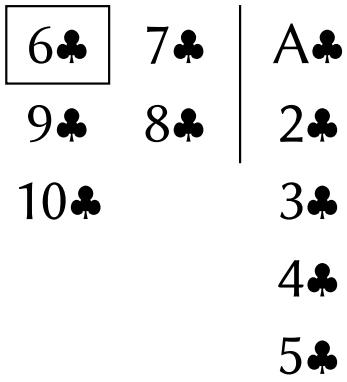
Merge Sort Example (4)



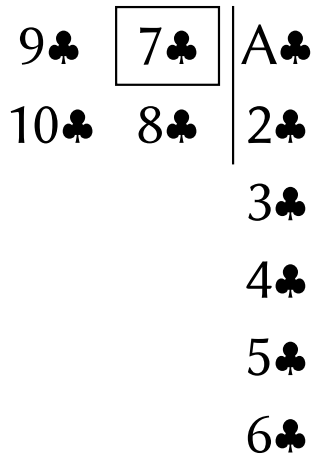
Merge Sort Example (5)



Merge Sort Example (6)

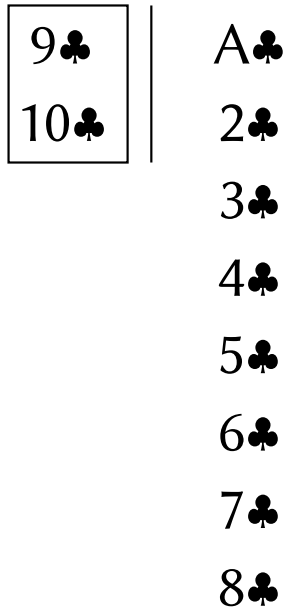


Merge Sort Example (7)



Merge Sort Example (8)

Merge Sort Example (9)



Merge Sort Example (10)

A♣

2♣

3♣

4♣

5♣

6♣

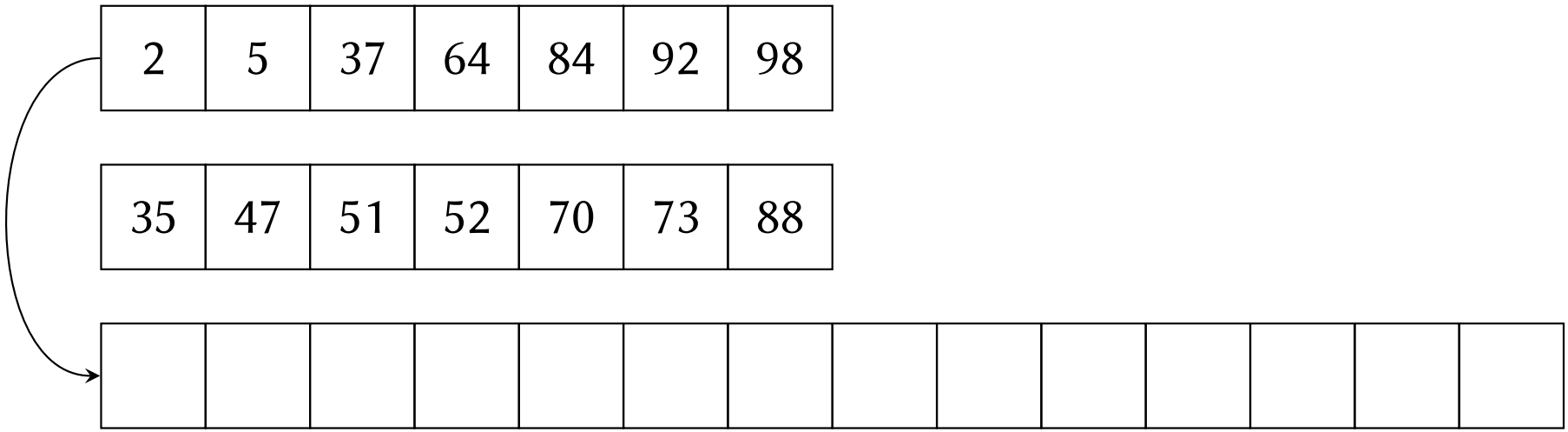
7♣

8♣

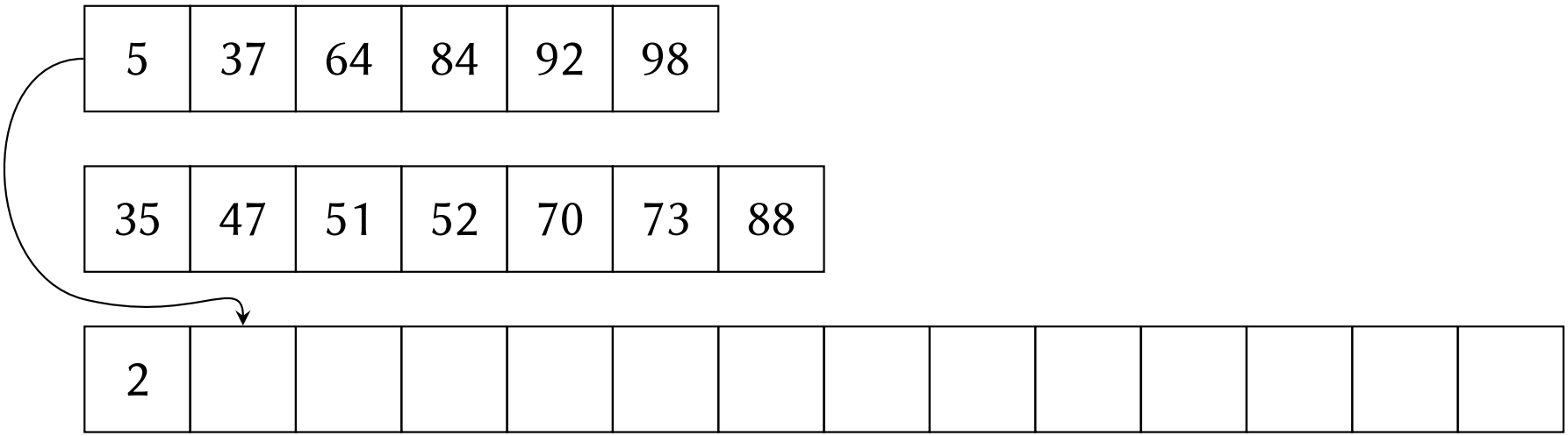
9♣

10♣

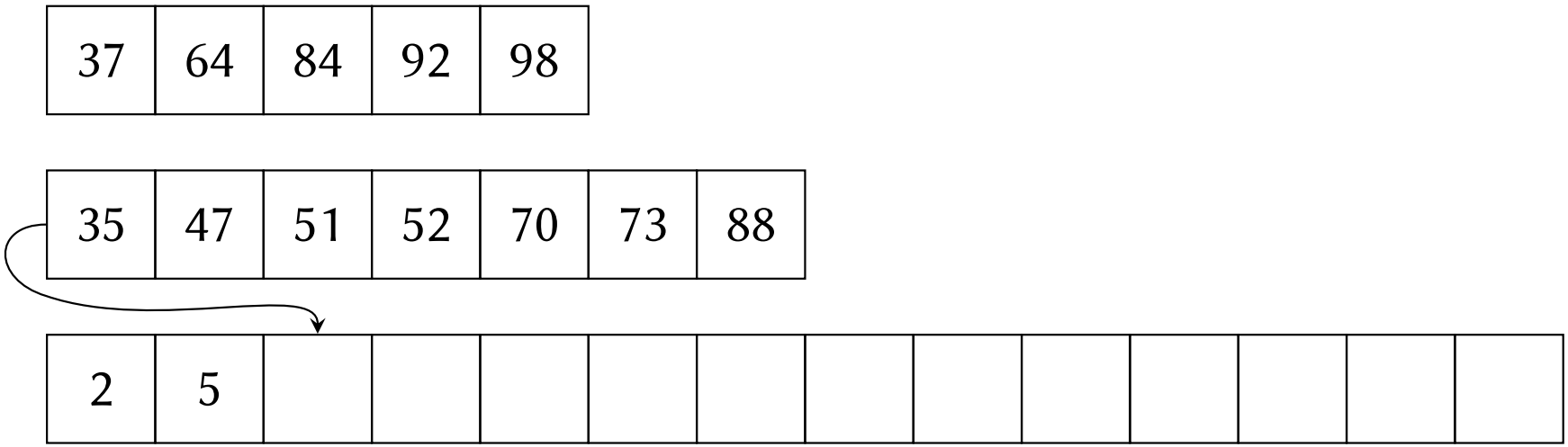
Array Merge Sort Example (1)



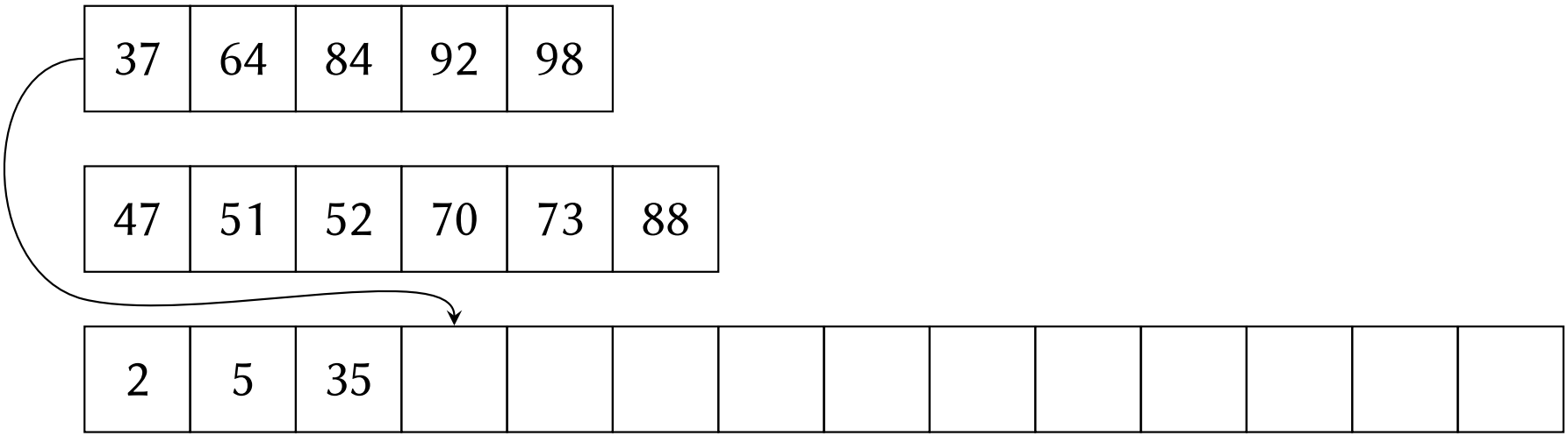
Array Merge Sort Example (2)



Array Merge Sort Example (3)



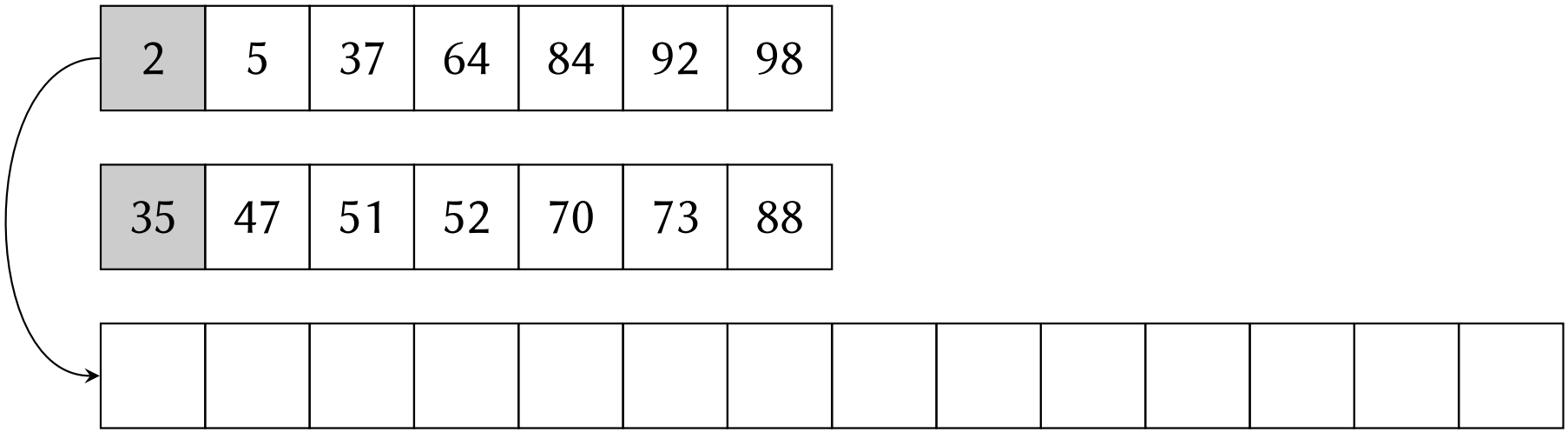
Array Merge Sort Example (4)



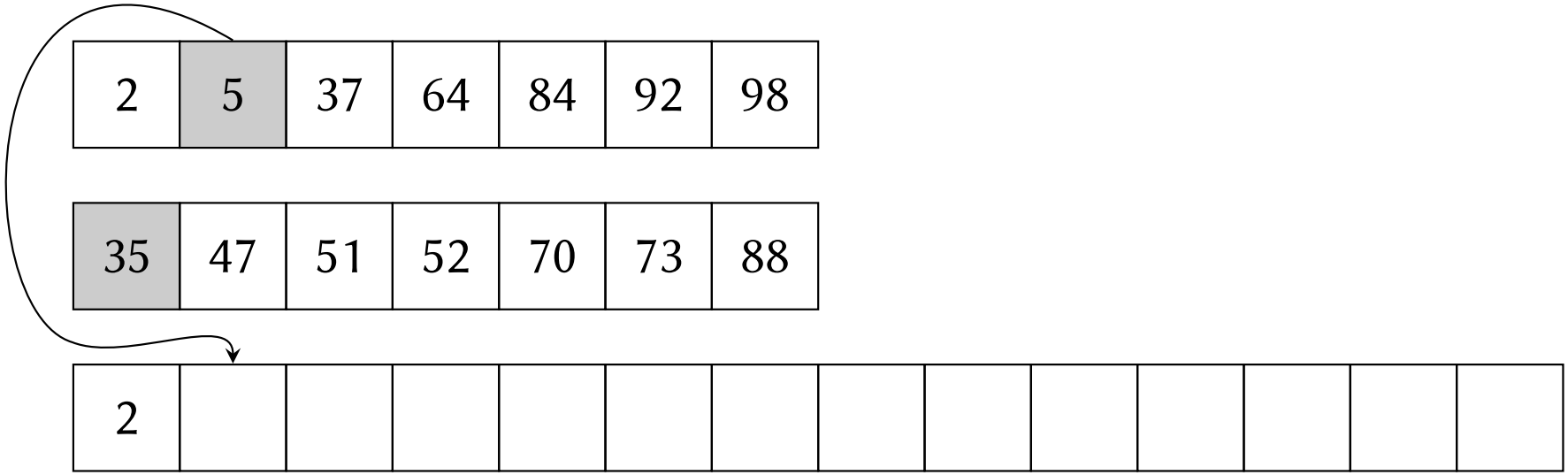
Efficient Array Merge

- In reality we do not take out elements from the arrays.
- That is because removing elements from an array is a complicated process (we may need to move all elements that follow one position forward).
- For that reason we use pointers to indicate the position we are in the array.

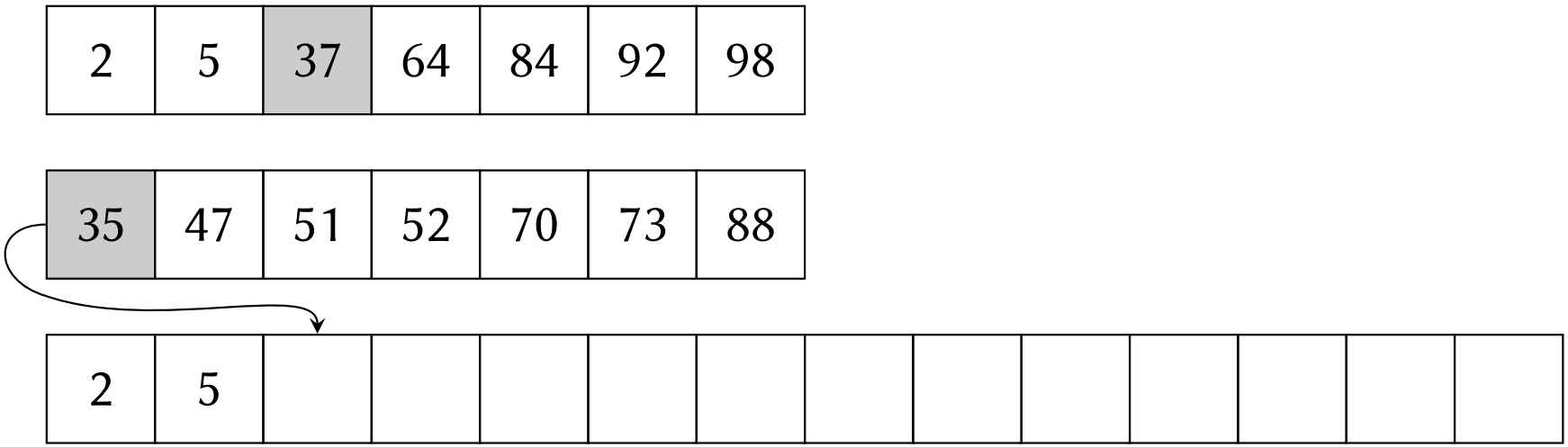
Array Merge with Pointers Example (1)



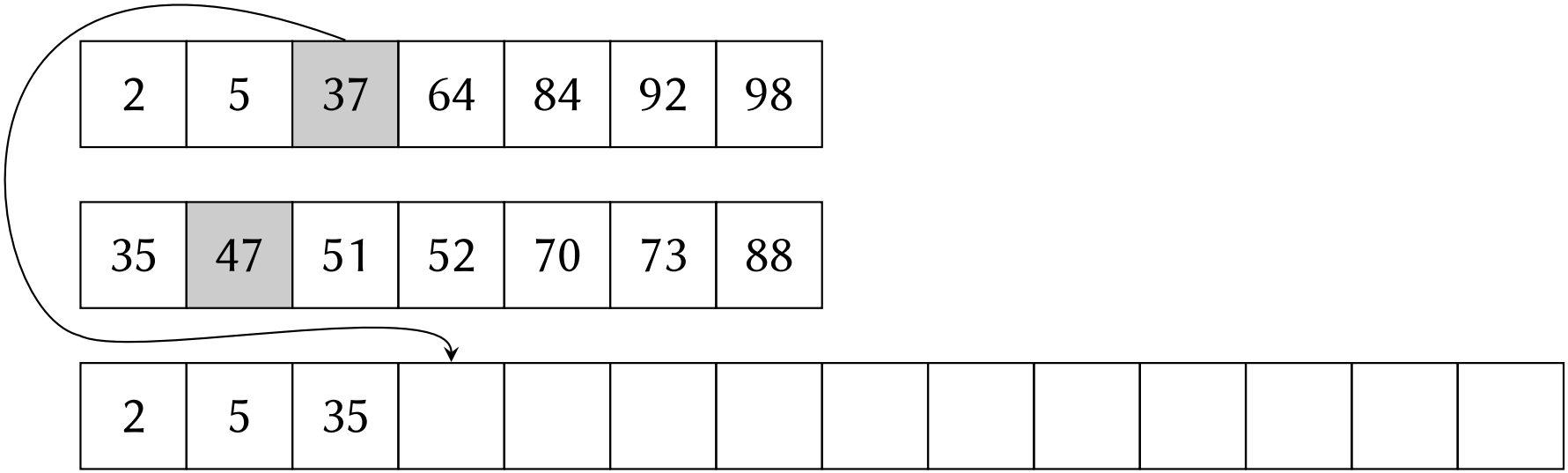
Array Merge with Pointers Example (2)



Array Merge with Pointers Example (3)



Array Merge with Pointers Example (4)



Array Merge

Algorithm: Array merge.

`ArrayMerge`$(A, B) \rightarrow C$

Input: A, a sorted array of items
B, a sorted array of items

Output: C, a sorted array containing the items of A and B

```
1  C ← CreateArray(|A| + |B|)
2  i ← 0
3  j ← 0
4  for k ← 0 to |A| + |B| do
5      if i ≥ |A| then
6          C[k] ← B[j]
7          j ← j + 1
8      else if j ≥ |B| then
9          C[k] ← A[i]
10         i ← i + 1
11     else if Compare(A[i], B[j]) ≤ 0 then
12         C[k] ← A[i]
13         i ← i + 1
14     else
15         C[k] ← B[j]
16         j ← j + 1
17 return C
```

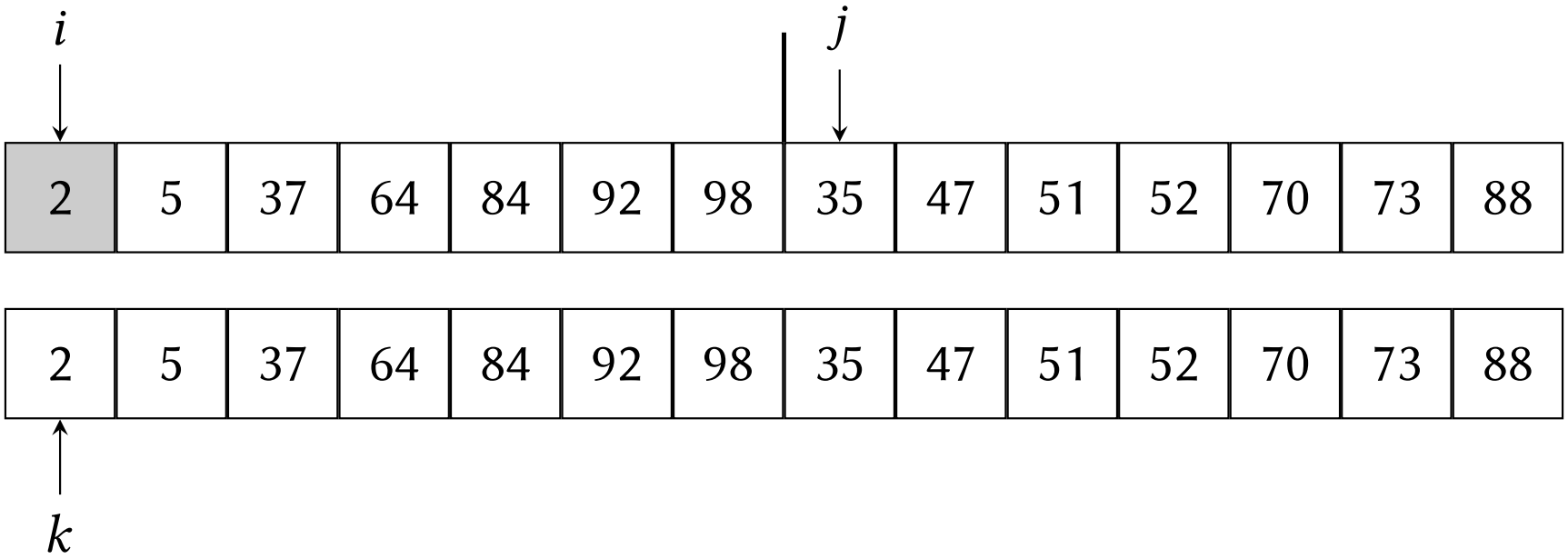
Explanation

- We use a variable i that shows where we are in array A, a variable j that shows where we are in array B, and a variable k that shows where we are in array C.
- Lines 5–7 handle the situation when array A is empty; then we put the remaining elements of B at the end of C.
- Lines 8–10 handle the situation when array B is empty; then we put remaining elements of A at the end of C.
- Otherwise, we put the minimum of $A[i]$ and $B[j]$ at the end of C and we increase i or j correspondingly.

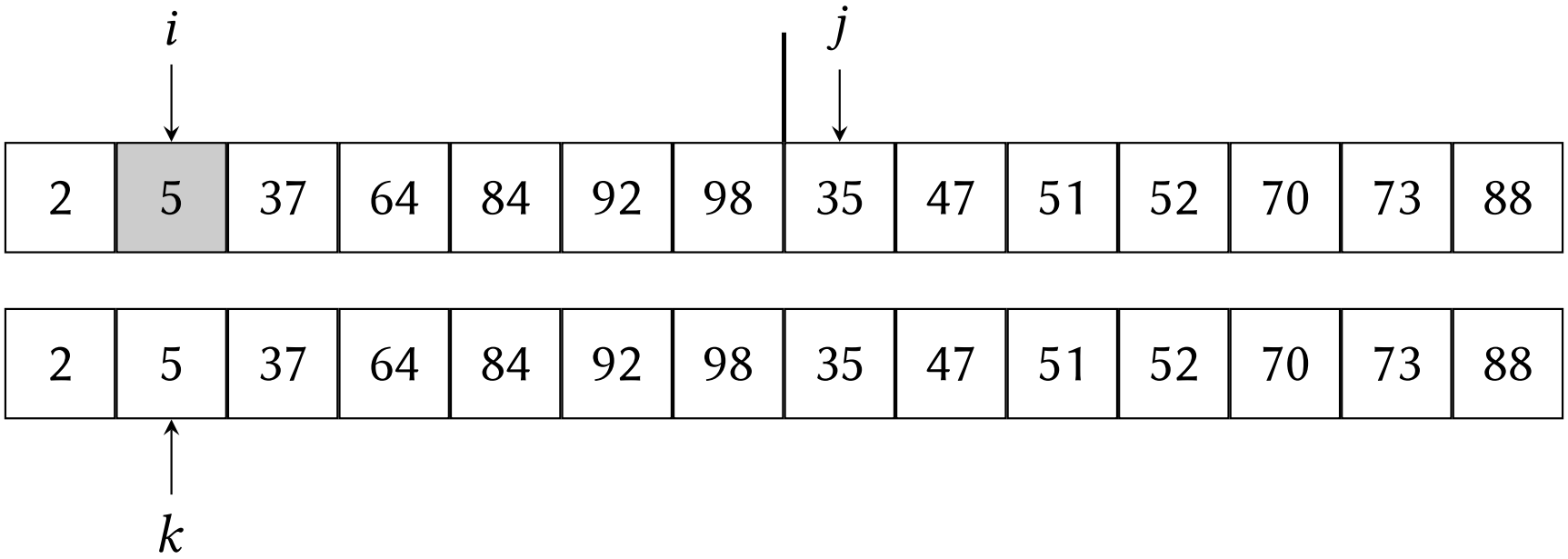
In-place Array Merge

- At this point we can make a further improvement.
- Instead of merging two arrays to create a third array, we can assume that we have divided our array to two sorted parts.
- Then we can merge these two parts, using a temporary array for help.
- In this way at the end we will have our original array sorted.
- This is an *in-place* merge.

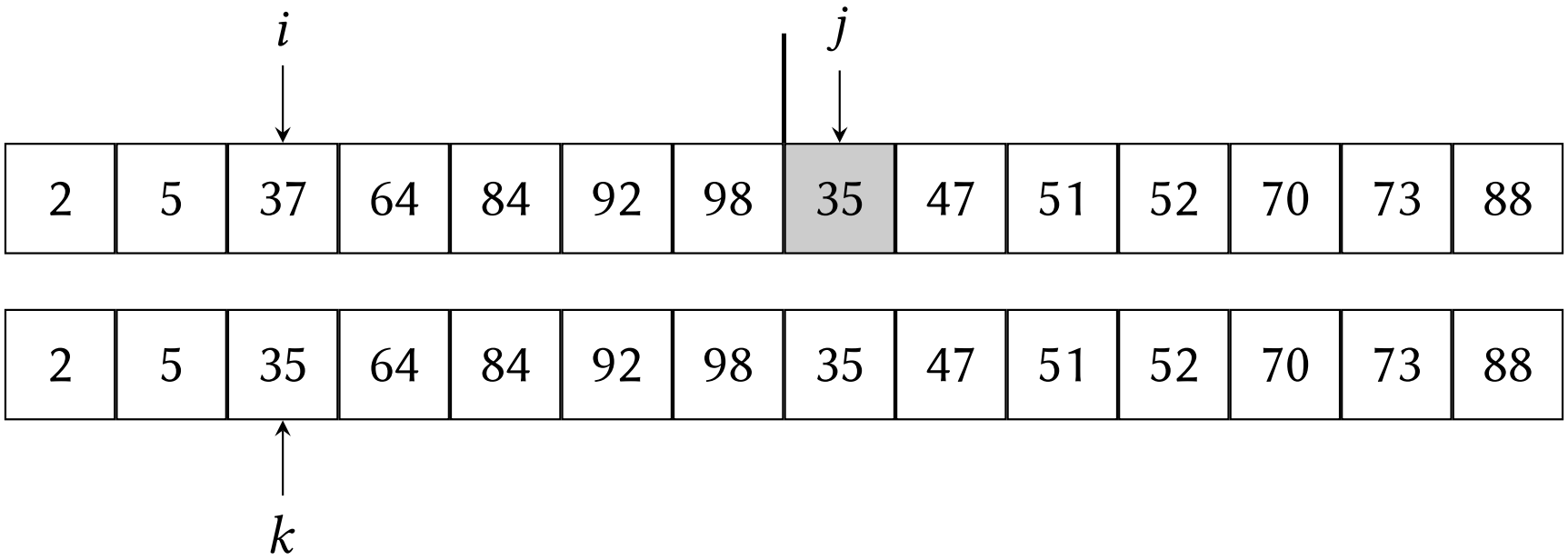
In-place Array Merge Example (1)



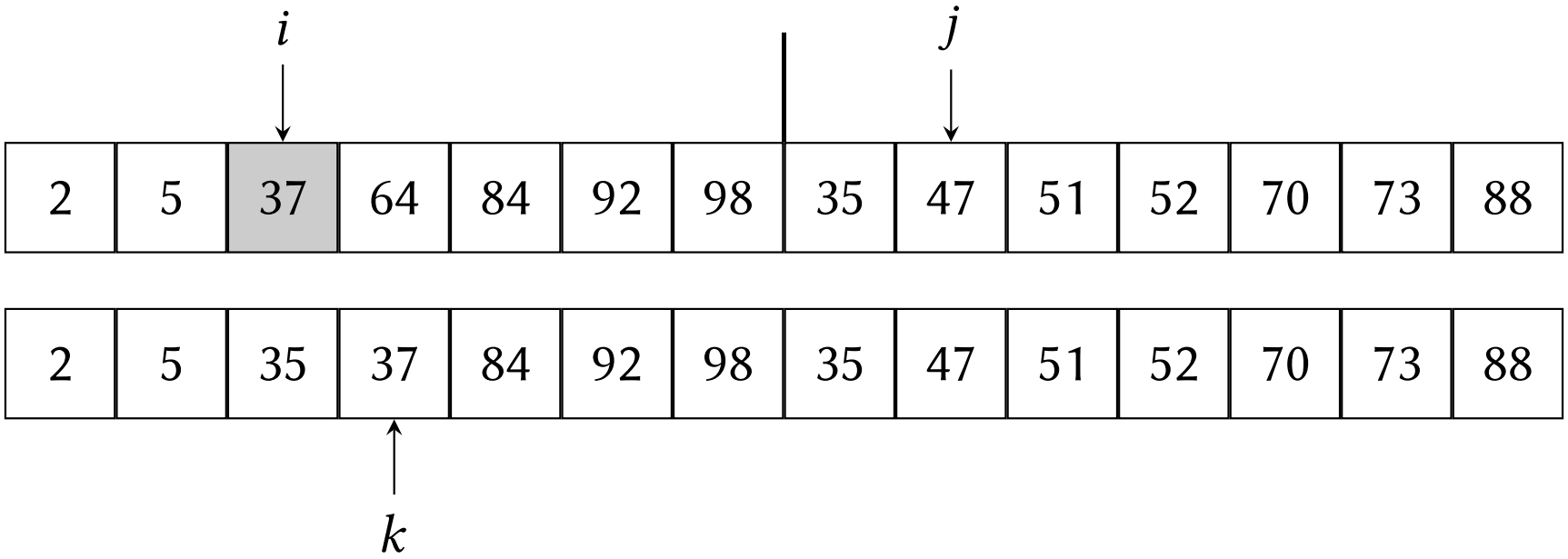
In-place Array Merge Example (2)



In-place Array Merge Example (3)



In-place Array Merge Example (4)



In-place Array Merge Algorithm

Algorithm: In-place array merge.

`ArrayMergeInPlace`(A, l, m, h)

Input: A, an array of items
l, m, h, array indices such that items $A[l], \ldots, A[m]$ and $A[m+1], \ldots, A[h]$ are sorted

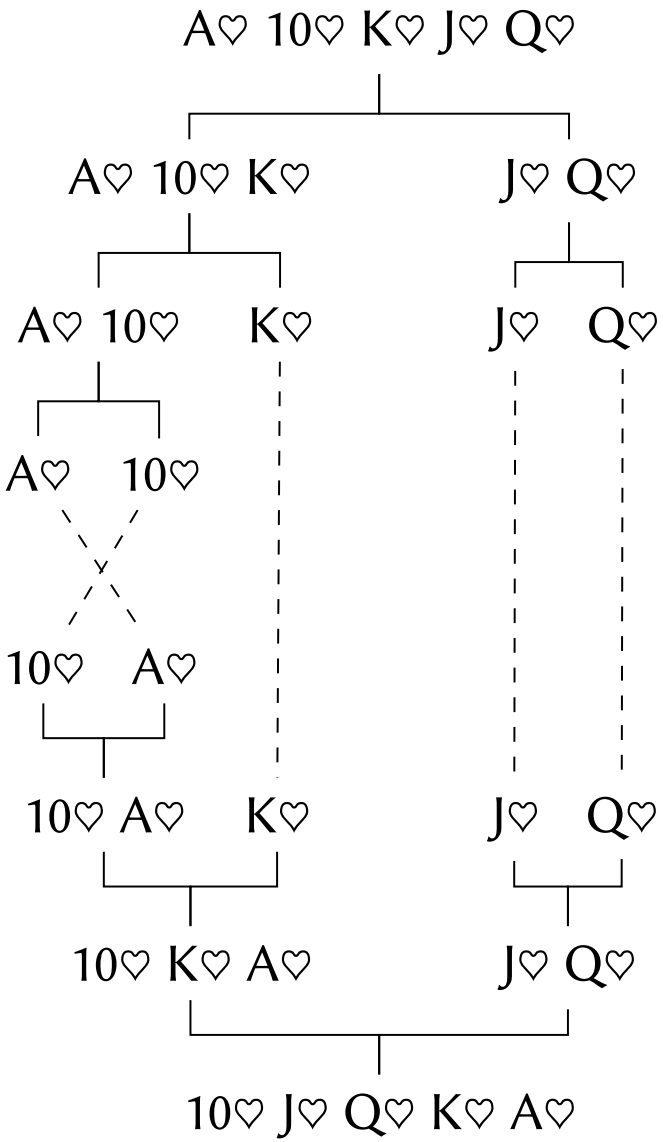
Result: A with items $A[l], \ldots, A[h]$ sorted

```
1  C ← CreateArray(h − l + 1)
2  for k ← l to h + 1 do
3      C[k − l] = A[k]
4  i ← 0
5  cm ← m − l + 1
6  ch ← h − l + 1
7  j ← cm
8  for k ← l to h + 1 do
9      if i ≥ cm then
10         A[k] ← C[j]
11         j ← j + 1
12     else if j ≥ ch then
13         A[k] ← C[i]
14         i ← i + 1
15     else if Compare(C[i], C[j]) ≤ 0 then
16         A[k] ← C[i]
17         i ← i + 1
18     else
19         A[k] ← C[j]
20         j ← j + 1
```

Merge Sort

- We now how to make a sorted array from two sorted arrays.
- So, if we have an array, we split it in half, we sort its two halves, and we merge them. Then we will have sorted our initial array.
- To sort each half, we can do the same, splitting it in half.
- We go on splitting in halves, until we arrive at pieces that contain only one element; these we just merge.
- This then is merge sort.

Merge Sort Example



Merge Sort Algorithm

Algorithm: Merge sort.

`MergeSort`(A, l, h)

Input: A, an array of items
l, h, array indices

Result: A with items $A[l], \ldots, A[h]$ sorted

```
1  if l < h then
2      m = l + ⌊(h − l)/2⌋
3      MergeSort(A, l, m)
4      MergeSort(A, m + 1, h)
5      ArrayMergeInPlace(A, l, m, h)
```

Explanation

- We start with an array A, where we want to sort elements $A[l], A[l+1], \ldots, A[h]$.
- If $l \geq h$, then A contains one element and we do not need to do anything.
- Otherwise we find the midpoint of the array in line 4 and we call `MergeSort` recursively for each one of the two parts, i.e, `MergeSort`(A, m, h) and `MergeSort`$(A, m+1, h)$.
- In each recursive call we do the same.
- The recursive calls stop when we have parts of one element each.
- When this happens, we start merging larger and larger pieces until we cover the whole array A.
- We kick-off the process with the call `MergeSort`$(A, 0, |A|-1)$.

MergeSort Call Trace (1)

- sort [84, 64, 37, 92, 2, 98, 5, 35, 70, 52, 73, 51, 88, 47]
  - sort [84, 64, 37, 92, 2, 98, 5]
    - sort [84, 64, 37, 92]
      - sort [84, 64]
        - sort [84]
        - sort [64]
        - merge [84] [64] → [64, 84]
      - sort [37, 92]
        - sort [37]
        - sort [92]
        - merge [37] [92] → [37, 92]
      - merge [64, 84] [37, 92] → [37, 64, 84, 92]
    - sort [2, 98, 5]
      - sort [2, 98]
        - sort [2]
        - sort [98]
        - merge [2] [98] → [2, 98]
      - sort [5]
      - merge [2, 98] [5] → [2, 5, 98]
    - merge [37, 64, 84, 92] [2, 5, 98] → [2, 5, 37, 64, 84, 92, 98]

MergeSort Trace Call (2)

- sort [84, 64, 37, 92, 2, 98, 5, 35, 70, 52, 73, 51, 88, 47]
  - sort [35, 70, 52, 73, 51, 88, 47]
    - sort [35, 70, 52, 73]
      - sort [35, 70]
        - sort [35]
        - sort [70]
        - merge [35] [70] → [35, 70]
      - sort [52, 73]
        - sort [52]
        - sort [73]
        - merge [52] [73] → [52, 73]
      - merge [35, 70] [52, 73] → [35, 52, 70, 73]
    - sort [51, 88, 47]
      - sort [51, 88]
        - sort [51]
        - sort [88]
        - merge [51] [88] → [51, 88]
      - sort [47]
      - merge [51, 88] [47] → [47, 51, 88]
    - merge [35, 52, 70, 73] [47, 51, 88] → [35, 47, 51, 52, 70, 73, 88]

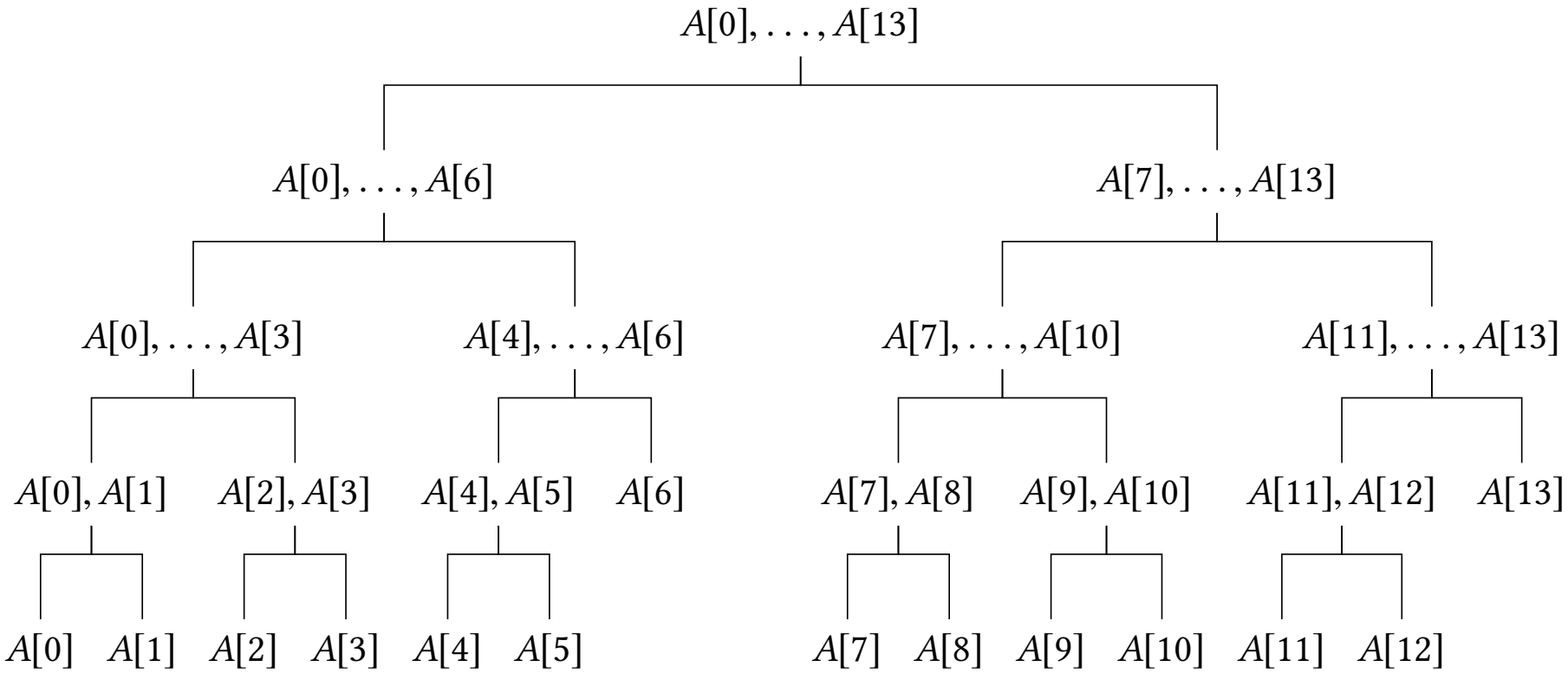
MergeSort Trace Call (3)

sort [84, 64, 37, 92, 2, 98, 5, 35, 70, 52, 73, 51, 88, 47]
└── merge [2, 5, 37, 64, 84, 92, 98] [35, 47, 51, 52, 70, 73, 88] → [2, 5, 35, 37, 47, 51, 52, 64, 70, 73, 84, 88, 92, 98]

Merge Sort and Divide and Conquer

- Merge sort is an application of *divide and conquer*.
- It sorts a set of items by dividing it in two and sorting the two halves.
- This is the key in deriving the complexity of merge sort.

Merge Sort Tree



Merge Sort Complexity

- In the worst case, merge sort requires $O(n \lg n)$ copies and $O(n \lg n)$ comparisons.
- That means that it can be used for sorting large amounts of data.
- Its main disadvantage is the amount of space it requires.
- Merge sort requires an amount of n extra space for sorting an array with n elements.

Outline

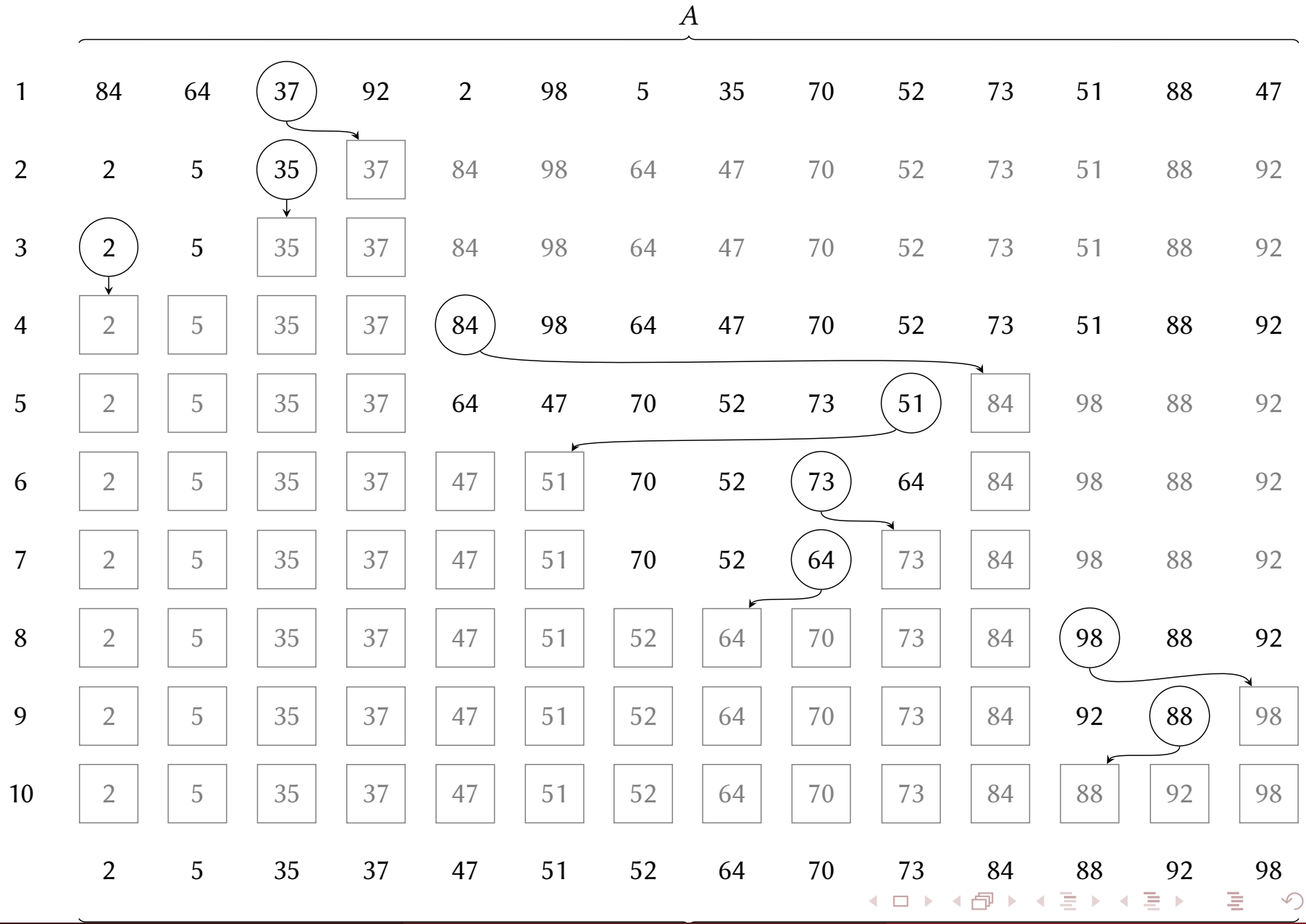
Basic Idea

- We pick an item among those we want to sort.
- We want to put it in its final, correct position.
- To do that, we only need to put all elements that are smaller in front of it, and all other elements after it.

Basic Idea (Contd.)

- Suppose we want to sort array A.
- Also suppose that $A[p]$ is in its correct, sorted position.
- Then we have $A[0], A[1], \ldots, A[p-1] < A[p]$ and $A[p] \leq A[p+1], A[p+2], \ldots, A[n-1]$.
- Now we need to put each one of $A[0], A[1], \ldots, A[p-1]$ and $A[p+1], A[p+2], \ldots, A[n-1]$ in its final position.
- We can repeat the same process for $A[0], A[1], \ldots, A[p-1]$ and for $A[p+1], A[p+2], \ldots, A[n-1]$.
- Then we will have three elements in their final positions and we can repeat the process for each of the two unsorted segments of $A[0], A[1], \ldots, A[p-1]$ and $A[p+1], A[p+2], \ldots, A[n-1]$.

Quicksort Example



Quicksort Algorithm

Algorithm: Quicksort.

`Quicksort`(A, l, h)

Input: A, an array of items
l, h, array indices

Result: A with items $A[l], \ldots, A[h]$ sorted

1 **if** $l < h$ **then**
2 $p \leftarrow$ `Partition`(A, l, h)
3 `Quicksort`$(A, l, p-1)$
4 `Quicksort`$(A, p+1, h)$

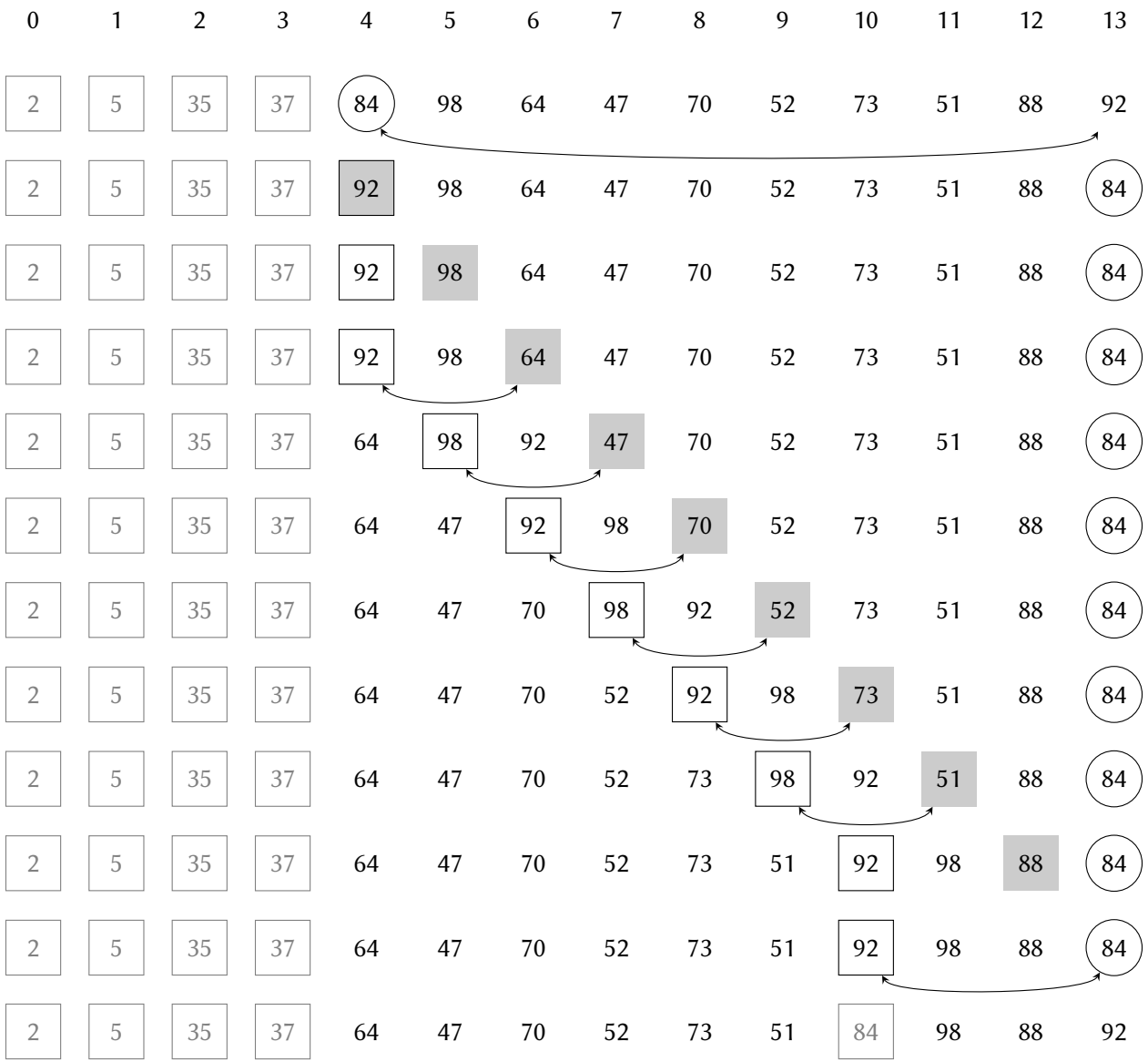
Explanation

- The algorithm is recursive.
- In line 2 we partition A in two parts, $A[0], A[1], \ldots, A[p-1]$ and $A[p+1], A[p+2], \ldots, A[n-1]$.
- The element at position p, around which we partition the array, is called *pivot*.
- Elements $A[0], A[1], \ldots, A[p-1]$ are smaller than the pivot.
- Elements $A[p+1], A[p+2], \ldots, A[n-1]$ are greater than or equal to the pivot.
- After partitioning A, we apply `QuickSort` to each partition.
- The recursive calls go on as long as the partition to be sorted has at least one element, as checked in line 1 of the algorithm.

Partitioning

- To complete the algorithm we must define the `Partition` function.
- We pick an element at random from the partition that we want to split in two.
- As we don't know its final position, we exchange it temporarily with the last element.
- We also assume that its final position will be at the start of the current partition.
- We traverse the elements of the current partition.
- Every time we encounter an element greater than or equal to the pivot, we know we must put it before the pivot's final position; and the pivot's final position will be one greater than what we had assumed.

Partition Example



Partitioning Algorithm

Algorithm: Partitioning.

`Partition`$(A, l, h) \rightarrow b$

Input: A, an array of items
l, h, array indices

Result: A is partitioned so that $A[0], \ldots, A[b-1] < A[b]$ and $A[b+1], \ldots, A[n-1] \geq A[b]$, for $n = |A|$

Output: b, the index of the final position of the pivot element

```
1  p ← PickElement(A)
2  Swap(A[p], A[h])
3  b ← l
4  for i ← l to h do
5      if Compare(A[i], A[h]) < 0 then
6          Swap(A[i], A[b])
7          b ← b + 1
8  Swap(A[h], A[b])
9  return b
```

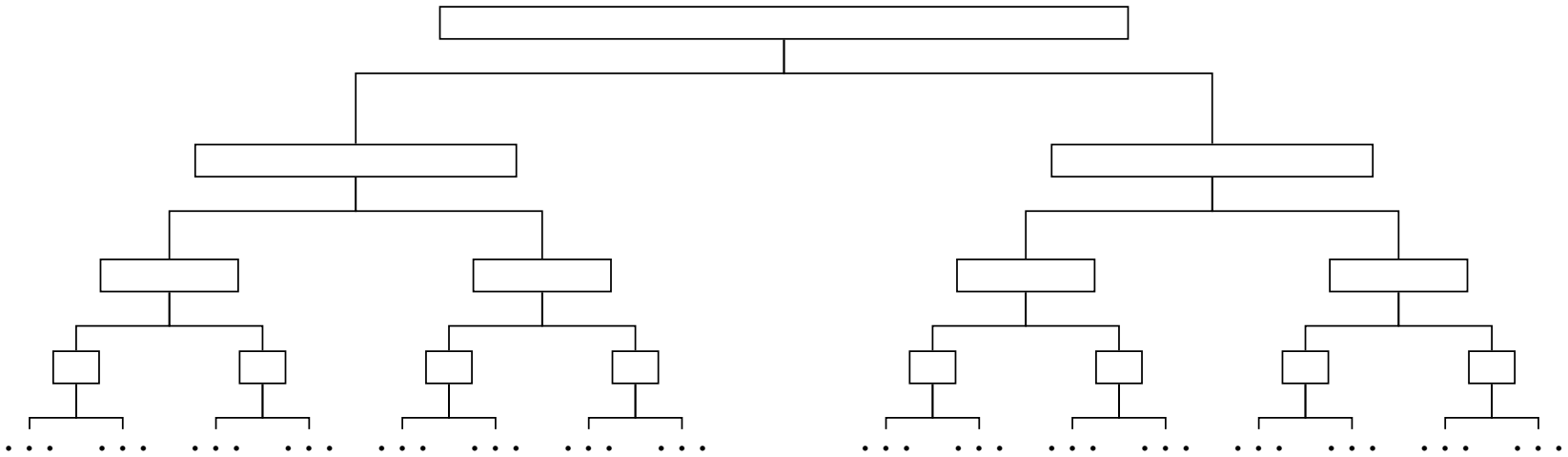
Explanation

- In line 1 we pick the pivot element.
- We exchange the pivot with last of our elements in line 2.
- In line 3 we set b to show the boundary between those elements that are smaller than the pivot element and those that are not.
- In the loop of lines 4–7 we traverse the items that will be partitioned.
- Each time we encounter an item smaller than the pivot, we exchange it with the item at position b and we increase b by one.
- At the end, we put the pivot to its final position and we return that position.

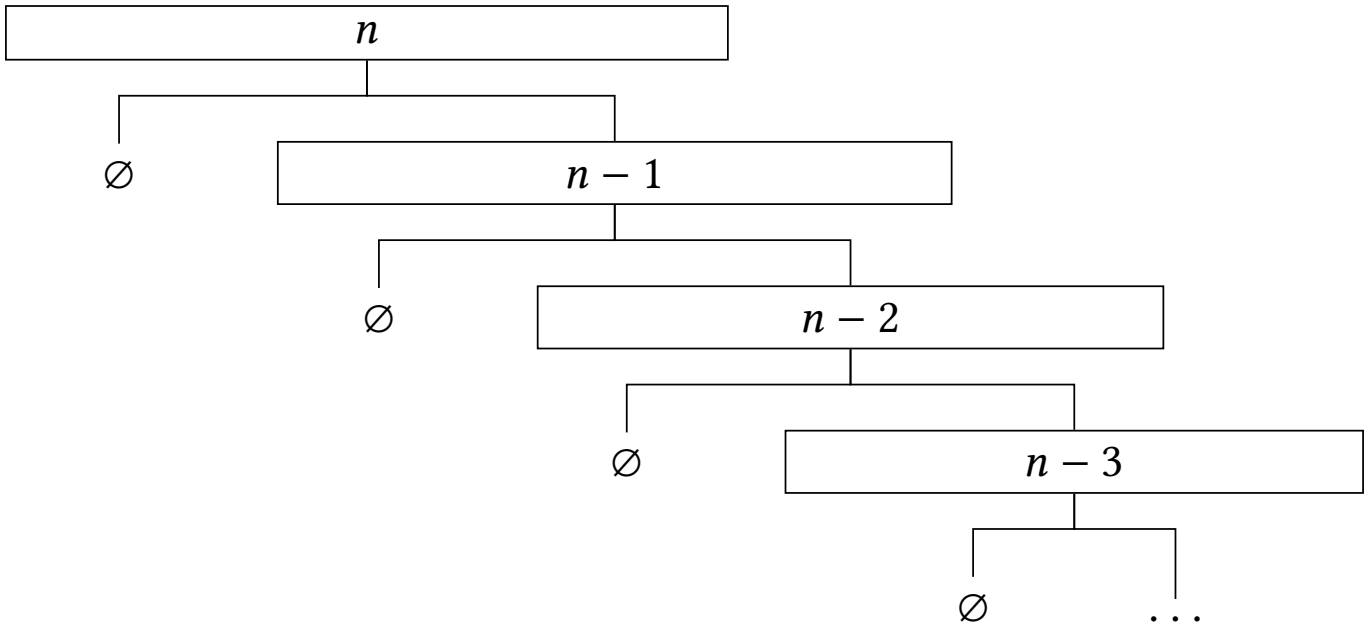
Picking the Pivot

- We pick the pivot *at random*.
- We do that to ensure that the performance of quicksort is close to its optimal.
- In the optimal case, the pivot must partition to two equal halves.
- In the worst case, the pivot is the smallest element, so in effect it does not really do any partitioning.

Optimum Quicksort Tree



Worst Case Quicksort Tree



Quicksort Complexity

- If each time we select as pivot an element that partitions an element to two equal halves, the complexity of quicksort is $O(n \lg n)$ exchanges and comparisons.
- If each time we select as pivot the smallest of the elements we want to partition, the complexity of quicksort is $O(n^2)$.
- If each time we select the pivot at random, the *expected* complexity of quicksort is again $O(n \lg n)$.
- Usually, quicksort implementations are very optimized, because the loop of lines 4–7 of the partitioning algorithm can be implemented in very efficient code.

How Probable is Pathological Behavior?

- If we pick the pivot at random, the probability of selecting the smallest or the largest element the first time, if we have n elements, is $2/n$.
- The second time the probability will be $2/(n-1)$.
- The probability that this will happen all the times is:

$$\frac{2}{n} \times \frac{2}{n-1} \times \cdots \times \frac{2}{3} = \frac{2^{n-2}}{3 \times \cdots \times n} = \frac{2^{n-1}}{1 \times 2 \times 3 \times \cdots \times n} = \frac{2^{n-1}}{n!}$$

- The value $2^{n-1}/n!$ is extremely small. For just 15 elements we have $2^{14}/15! \approx 1/79{,}814{,}109$.

Randomized Algorithms

- Quicksort is an example of a *randomized algorithm*.
- Randomized algorithms depend on chance and they give a guarantee on their expected performance.
- They have gained a lot in popularity over the last years.
- A basic requirement for them to work is to be able to produce random numbers with a good *random number generator*.

Outline

Quicksort and Merge Sort

- Quicksort needs to access elements in a completely arbitrary order.
- Merge sort processes elements sequentially.
- Therefore, merge sort is better suited for arrays than quicksort.
- On the other hand, quicksort requires less space than merge sort.

Quicksort and Heapsort

- They usually have the same complexity, but in problematic situations situations heapsort is faster, as it is guaranteed that it will run in time $O(n \lg n)$.
- If we do not encounter a problematic situation, quicksort is usually faster.
- If we do require $O(n \lg n)$ performance, for example in systems with critical response times where we cannot tolerate $O(n^2)$, then we have to use heapsort.

Merge sort and Heapsort

- Merge sort can be parallelized (each segment can be handled by a different processor or core).
- Merge sort can also be used for *external sorting*, i.e., sorting data in external storage, such as disks.

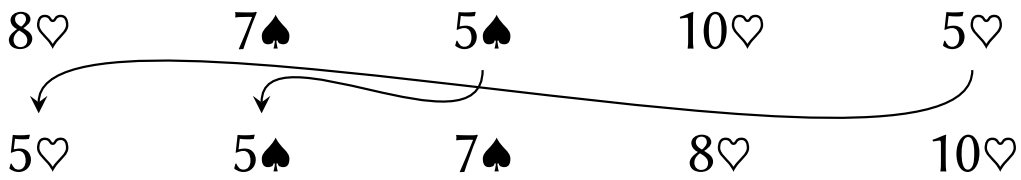
Selection Sort and Insertion Sort

- Selection sort and insertion sort are slow, but they are acceptable for a small number of elements.
- Sometimes they are used in combination with other algorithms: for example, we may use quicksort in the beginning and when the number of data falls below a threshold, say 20, we switch to the simpler algorithm.

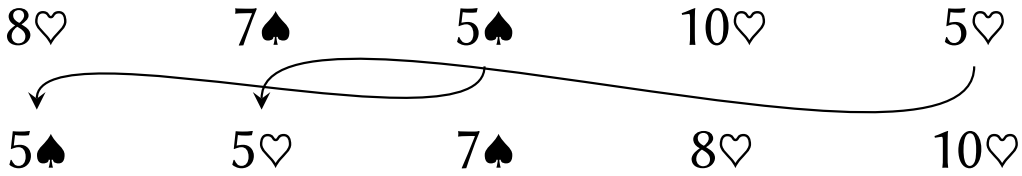
Stable Sorting

- A sorting algorithm is called *stable* when it preserves the relative order of records that have the same keys.
- That is, if we have records R_i and R_j with keys K_i and K_j respectively, with $\texttt{Compare}(K_i, K_j) = 0$ and, moreover, R_i is before R_j in the input to our algorithm, R_i must come before R_j in the output.
- Merge sort and insertion sort are stable, the other ones are not.

Examples of Stable and Unstable Sorting



Heapsort: unstable sorting.



Insertion sort: stable sorting.