CSCI203

Algorithms and Data Structures

Improving Sorting I

Lecturer: Dr. Fenghui Ren
Room 3.203
Email: fren@uow.edu.au

Sorting

- Insertion Sort (review)
- Merge Sort
- The Heap, a new data structure
- Heap Sort

Insertion Sort.

- You should already be familiar with insertion sort from CSIT113.
- Insertion sort uses the following strategy:
  - Start with the second element in the list.
  - Insert it in the right place in the preceding list.
  - Repeat with the next unsorted element.
  - Keep going until we have placed the last element in the list.
- We can see how this works with an example.

Insertion Sort Example

Starting list:

7	8	14	20	5	15	18	17	6	13

Start at the next element.

7	8	14	20	5	15	18	17	6	13

Put it in the right place

5	7	8	14	20	15	18	17	6	13

We have finished the step.

Repeat with the next element.

5	7	8	14	20	15	18	17	6	13

Insertion Sort Example...

- Just showing the result at the end of each iteration

5	7	8	14	20	15	18	17	6	13
5	7	8	14	15	20	18	17	6	13
5	7	8	14	15	18	20	17	6	13
5	7	8	14	15	17	18	20	6	13
5	6	7	8	14	15	17	18	20	13
5	6	7	8	13	14	15	17	18	20

- And we have finished

Looking Deeper

- To sort a list of n numbers requires $n-1$ iterations.
- Each iteration, however, requires that the selected entry be compared with the already sorted list until its correct location can be found.
- In the worst case this means comparing with every such element.
  - So, to sort the ith element requires i comparisons.
- Typically, the entire sort will require around $n^2/2$ comparisons.

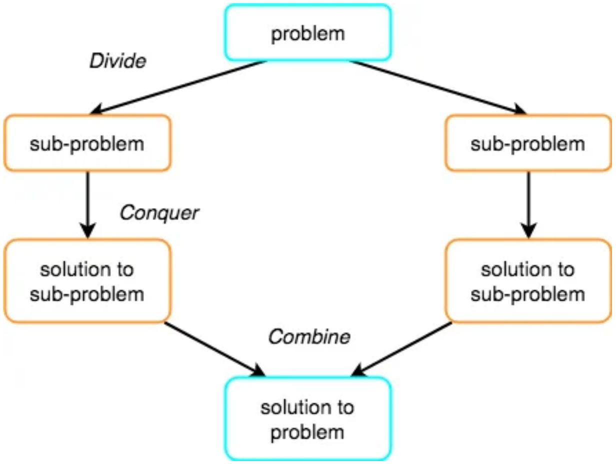
Merge Sort

- Many different sorting algorithms you have seen have one common feature:
  - They all sort the values in place, the sorted array is the same array as the unsorted one.
- Merge sort takes a different approach:
  - It uses a second array to hold the intermediate results.
  - It works recursively by dividing the unsorted array into two parts and merging them in order.

Merge Sort...

- Because this procedure divides all the way down before merging back up, the final result is a sorted array.

- The pseudocode representation of the merge sort algorithm is as follows:

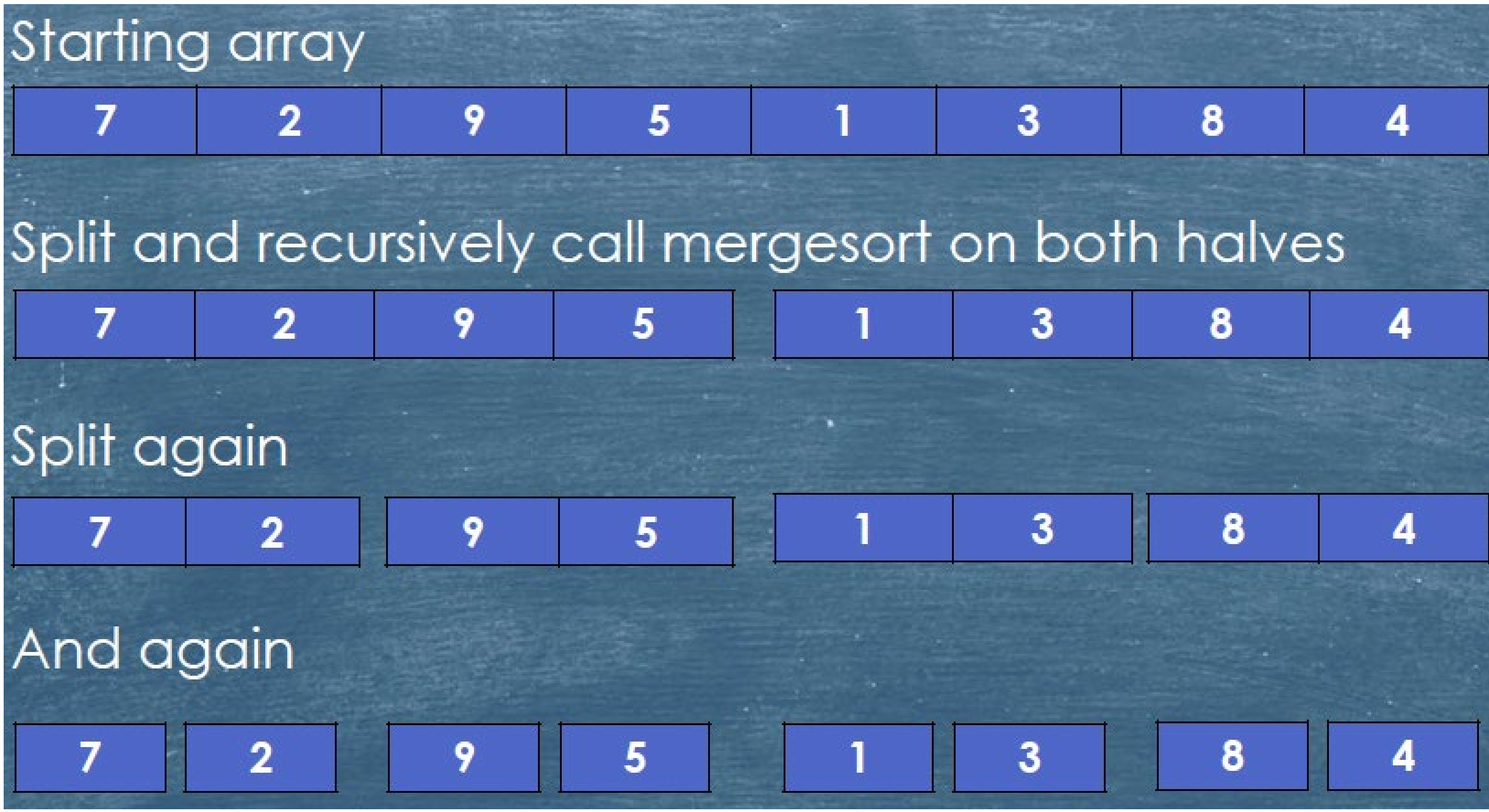


The Merge Sort Algorithm

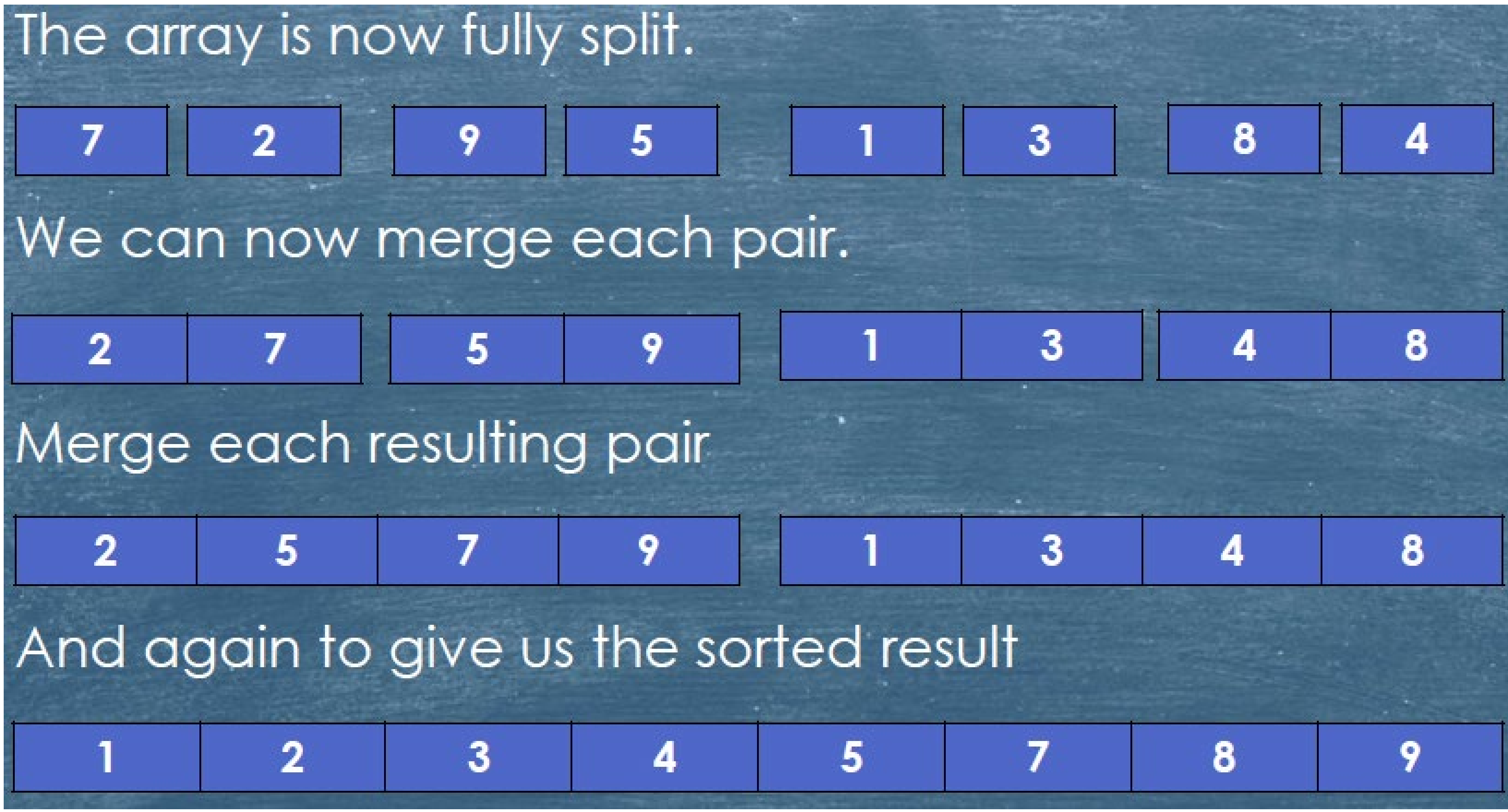
```
// temporary array used in merge procedure
global X[1..n]

procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])// sort the left half
    mergesort(T[centre+1..right]) // sort the right half
    merge(T[left..centre],T[centre+1..right],T[left..right])
      // join the halves in sorted order
```

Merge Sort: an example



Merge Sort: an example…



The Merge Sort Algorithm

```
// temporary array used in merge procedure
global X[1..n]

procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])// sort the left half
    mergesort(T[centre+1..right]) // sort the right half
    merge(T[left..centre],T[centre+1..right],T[left..right])
      // join the halves in sorted order
```

The Merge Sort Algorithm

```
procedure merge(A[1..a], B[1..b], C[1..a + b])
    apos = 1; bpos = 1; cpos = 1
    while apos < a and bpos < b do
        if A[apos] < B[bpos] then
          X[cpos] = A[apos]
          apos = apos + 1; cpos = cpos + 1
        else
          X[cpos] = B[bpos]
          bpos = bpos + 1; cpos = cpos + 1
    while apos < a do
          X[cpos] = A[apos]
          apos = apos + 1; cpos = cpos + 1
    while bpos < b do
          X[cpos] = B[bpos]
          bpos = bpos + 1; cpos = cpos + 1
    for cpos = 1 to a + b do
          C[cpos] = X[cpos]
```

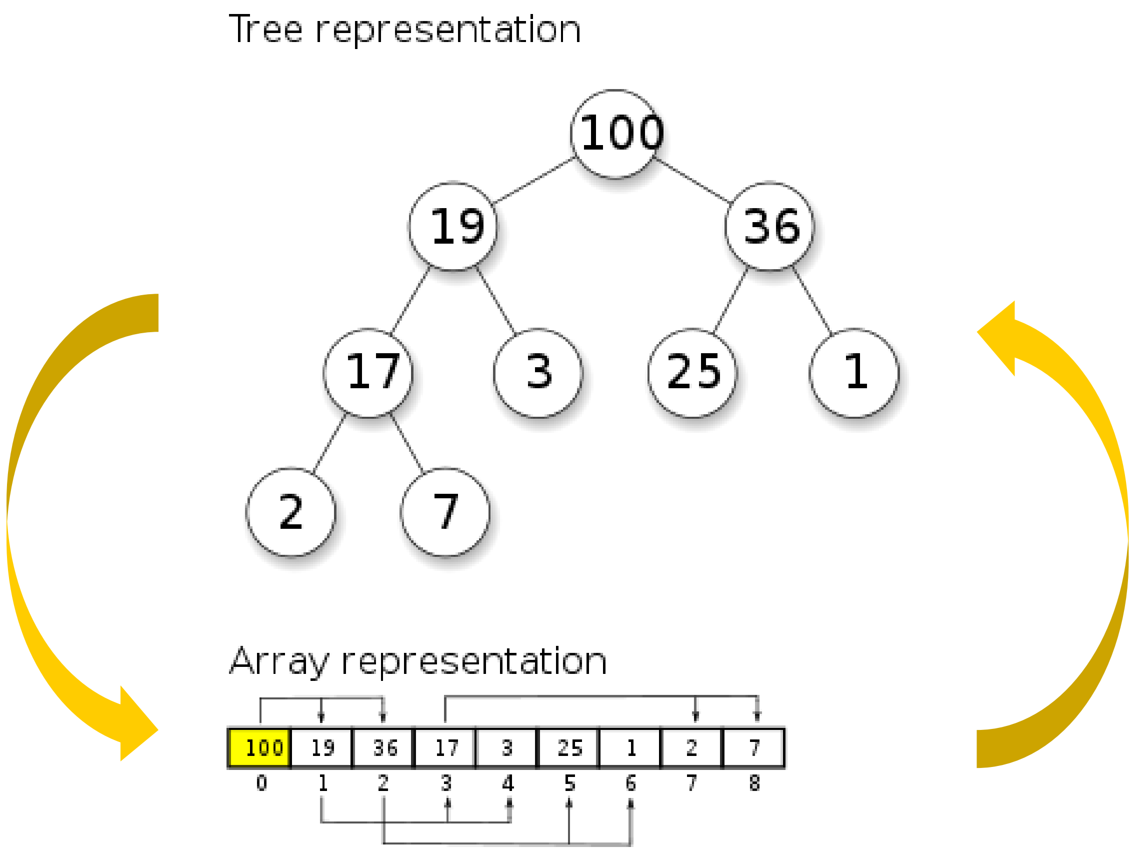
Some Analysis

- At each level, merge operates on all n items in the array.
- As each level divides the array in two, there are $\log n$ levels.
- Overall, `mergesort` requires $n \times \log n$ operations.

Heaps

- A heap is an essentially complete binary tree with an additional property
- Max-heap: the value in any node is less than or equal to the value in its parent node. (except for the root node).
- Min-heap: the value in any node is greater than or equal to the value in its parent node. (except for the root node).
- We can store a heap (or any other binary tree) in an array:
  - Heap[1] is the root of the tree
  - Heap[2] and Heap[3] are the children of Heap[1]
  - In general, Heap[i] has children Heap[2i] and Heap[2i+1] for root's index is 1
  - Heap[i] has children Heap[2i+1] and Heap[2i+2] for root's index is 0

An Example - Max-heap

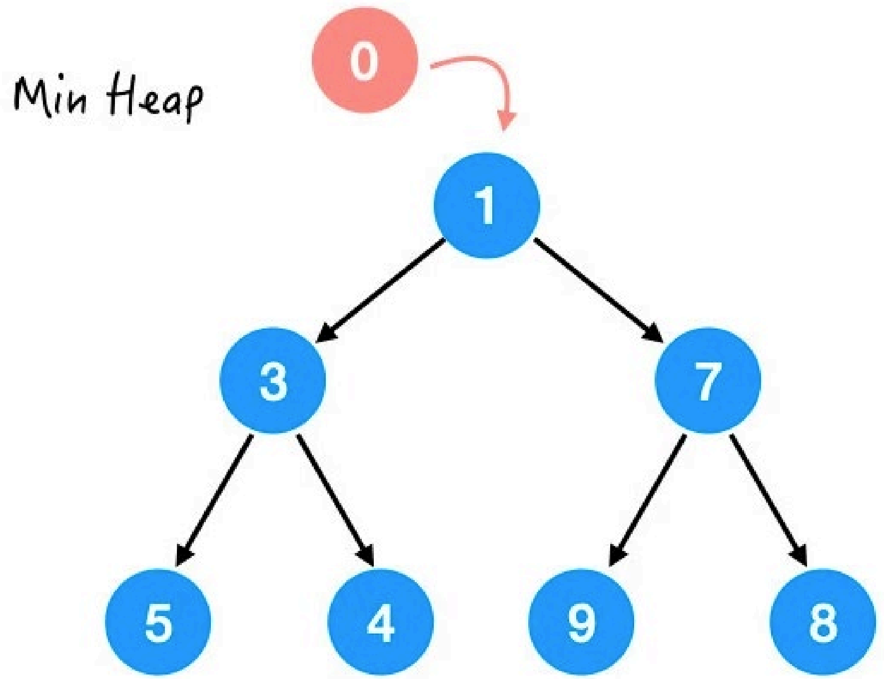


Operations on Heaps

- If we have a non-heap how can we convert it into one?
  - `Makeheap/Heapify`: create a heap data structure
- If we have a heap and add a new element at the next leaf how can we restore the heap property?
- If we have a heap and change the root element how can we restore the heap property?
- We need two basic functions to manage heaps:
  - `Siftup`: Insert a new leaf into the correct position;
  - `Siftdown`: Insert a new root element into the correct position.
  - Each compares an element of the heap with other elements, either its parent or its children.

Siftup: min-heap

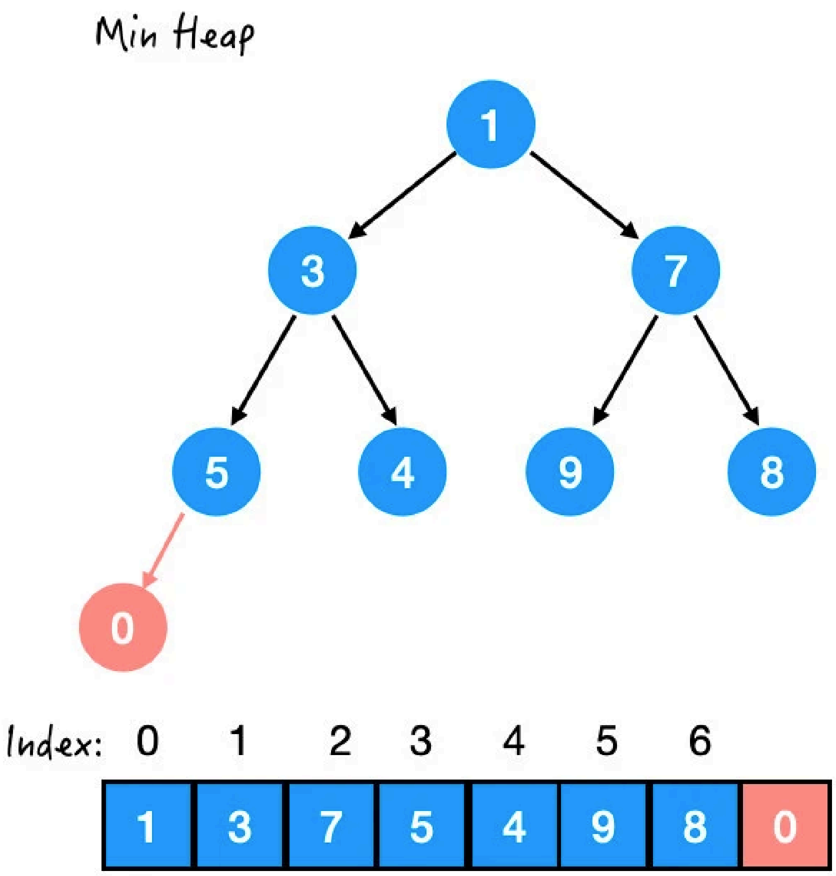
We can't add the new leaf as the root directly. Why?



It will destroy the binary tree structure.

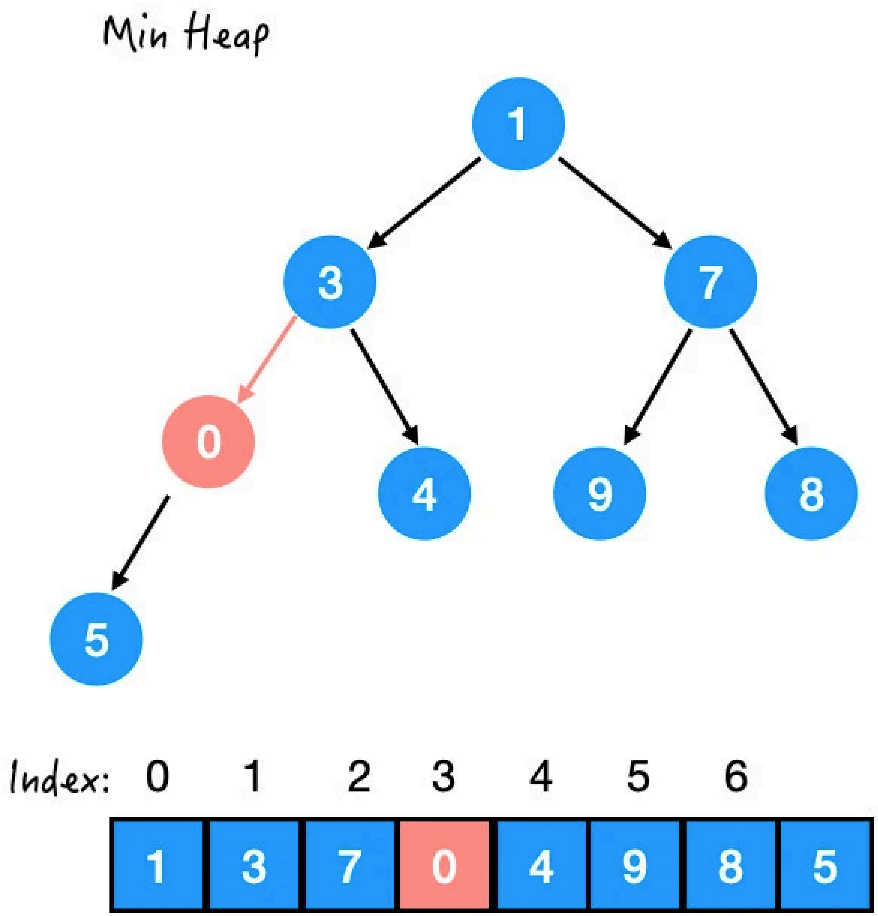
Siftup: min-heap

- We must make sure it is still a binary tree after the operation. How?
- Add new leaf to the end than swap with the parents



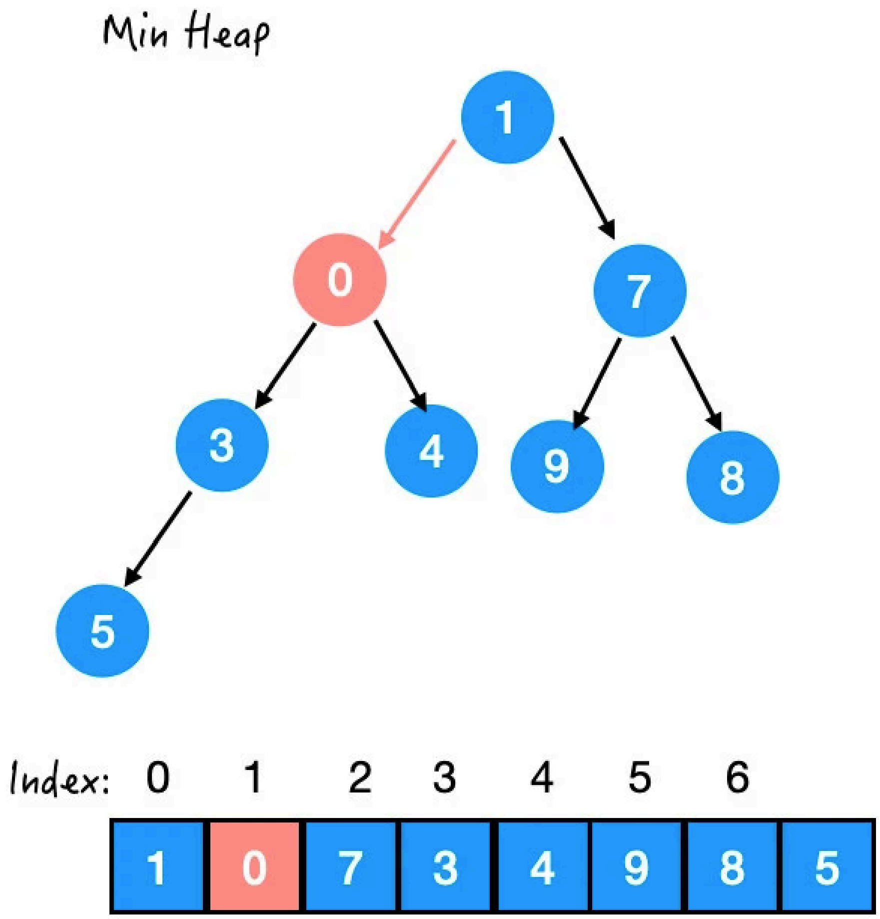
Siftup: min-heap

- Swap with its parent if its no bigger
- Because in a heap, the smaller numbers on top of bigger numbers



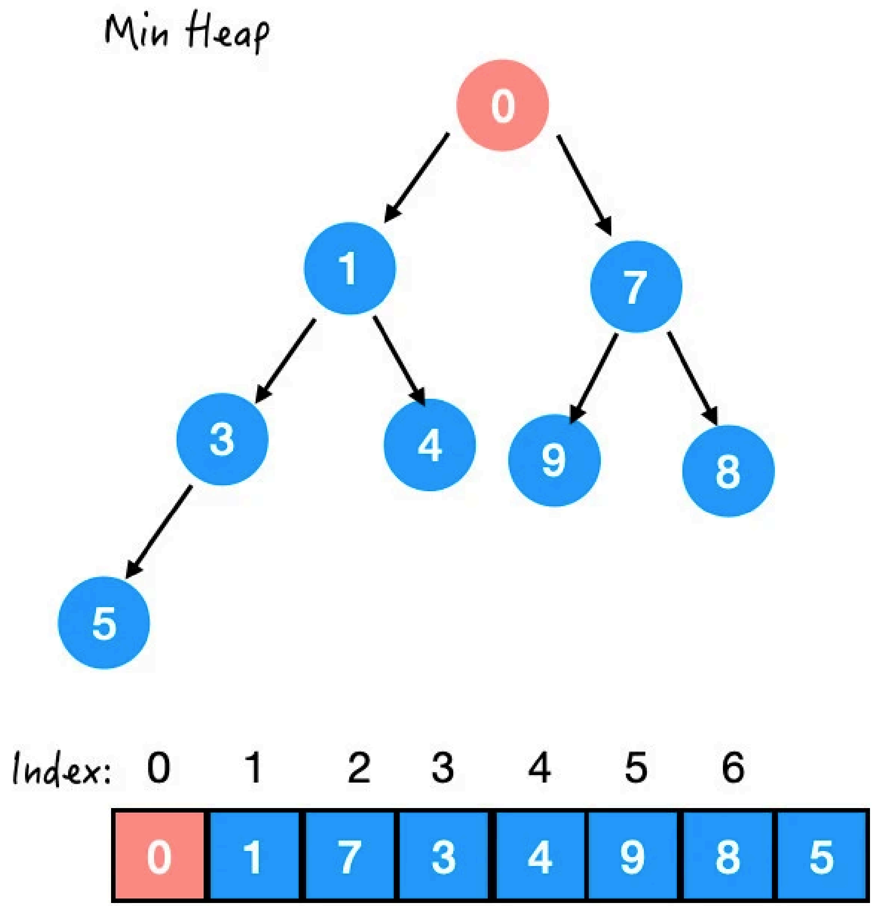
Siftup: min-heap

- Swap with its parent again
- Because in a heap, the smaller numbers on top of bigger numbers



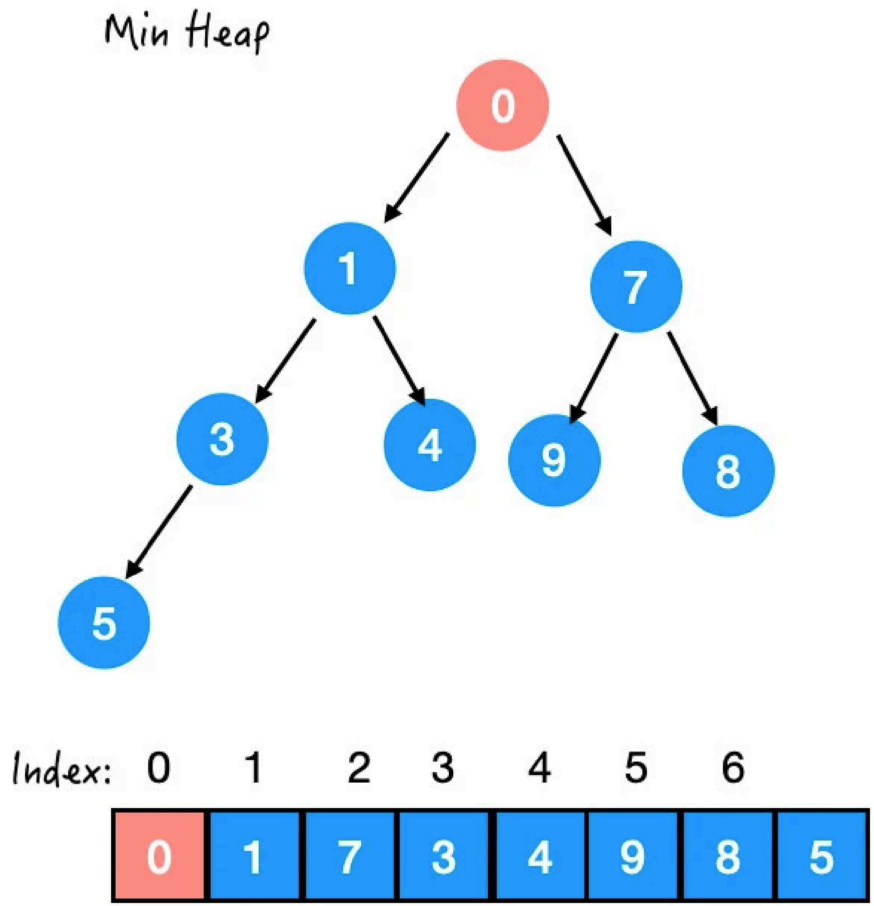
Siftup: min-heap

- Keep on swapping with its parent till to the correct position
- Finally, a new heap is created



Siftup: complexity

- Only swap nodes on a single branch
- The swap times depend on the height of the heap
- The complexity class of siftup operation is log(n)



Siftup: min-heap

```
Procedure siftup(Heap, i)
// move element i up to its correct position
    if i = 1 return
    p = i ÷ 2 // integer division
    //the condition should be changed to Heap[p]>
    //Heap[i] for a max-heap
    if Heap[p] < Heap[i]
        return
    else
        swap (Heap[i], Heap[p])
        siftup(Heap,p)
    endif
end
```

`Siftup` - an example (max-heap)

Start with the following heap:

10	7	5	6	3	1	

Add a new element:

10	7	5	6	3	1	9

Compare with its parent:

10	7	5	6	3	1	9

Swap:

10	7	9	6	3	1	5

Compare again

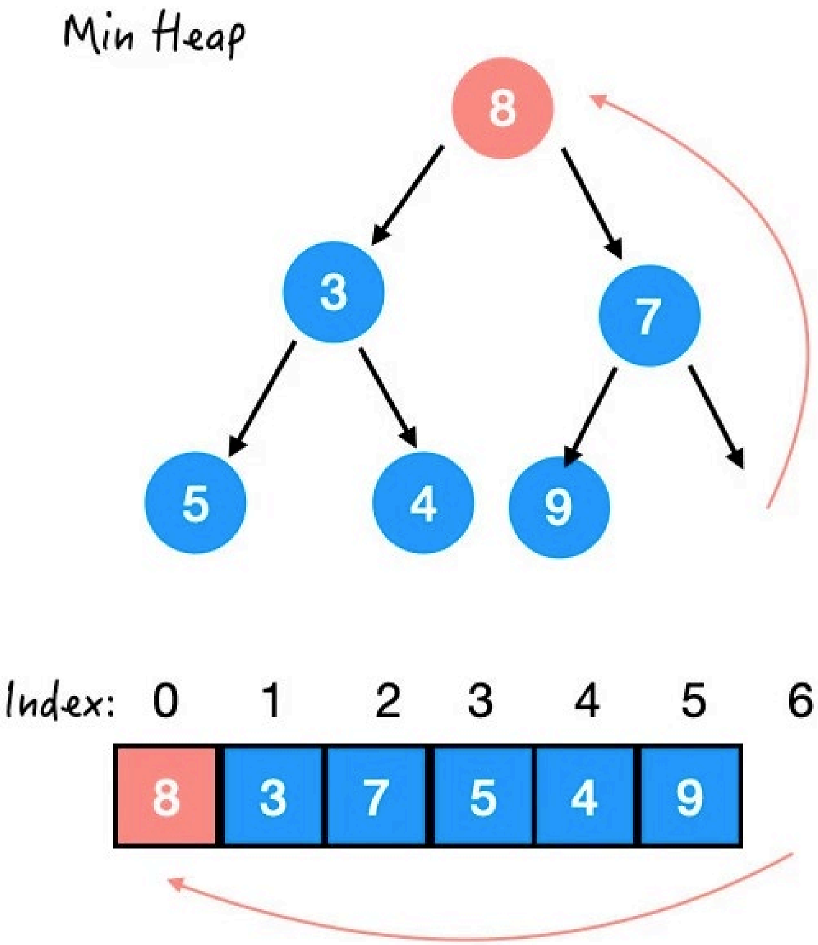
10	7	9	6	3	1	5

Done

10	7	9	6	3	1	5

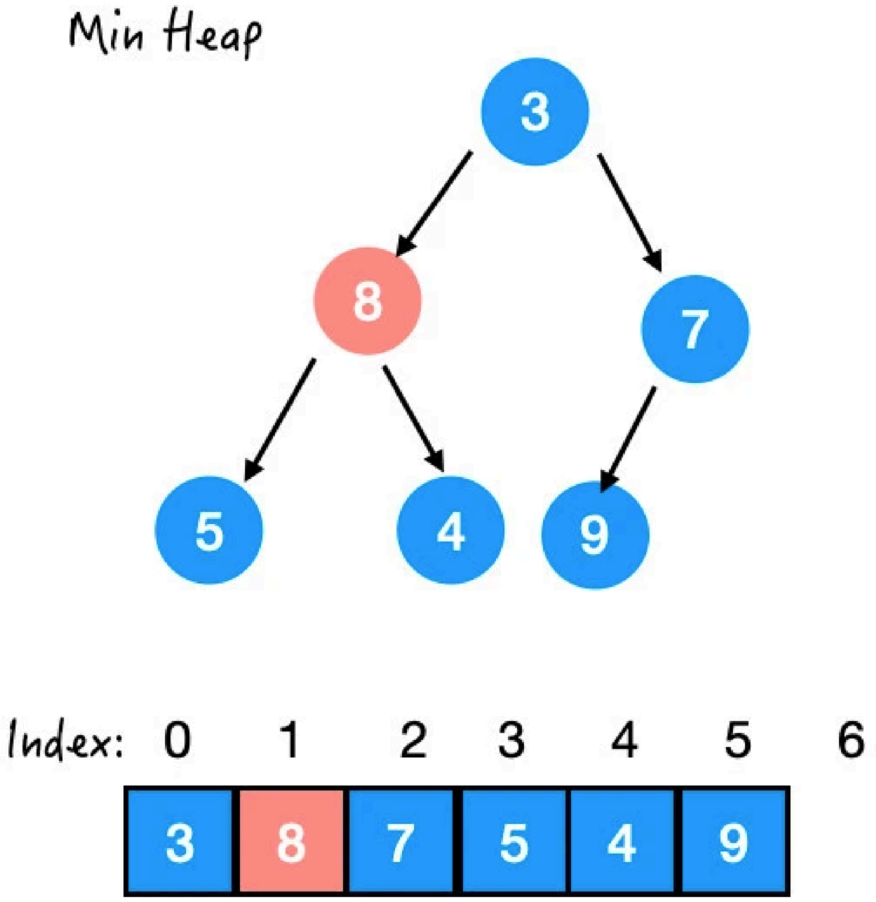
Siftdown: min-heap

- Move the last node as the new root (temporally)
- It is not a heap as 8 is bigger than both 3 and 7
- Can we swap 8 with 7?
- Yes, we can. But after that, we have to swap 7 with 3. Why?
- So we swap 8 with 3 (the smaller children) directly



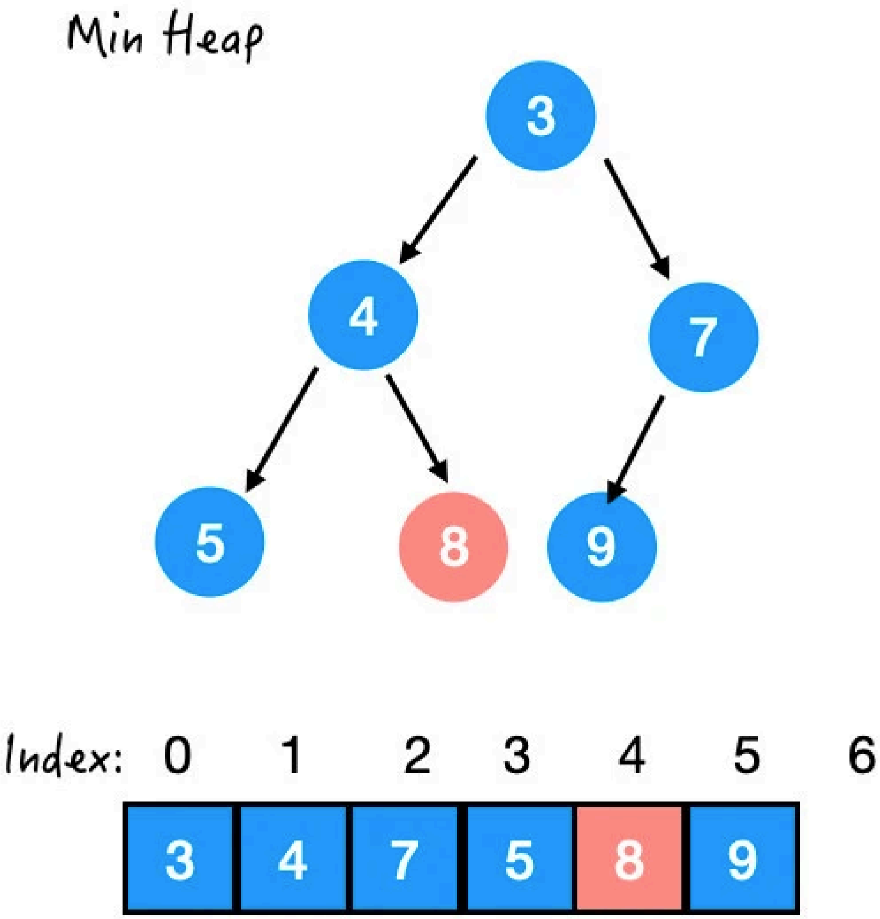
Siftdown: min-heap

- **We swap 8 with 4.**
- **Why?**



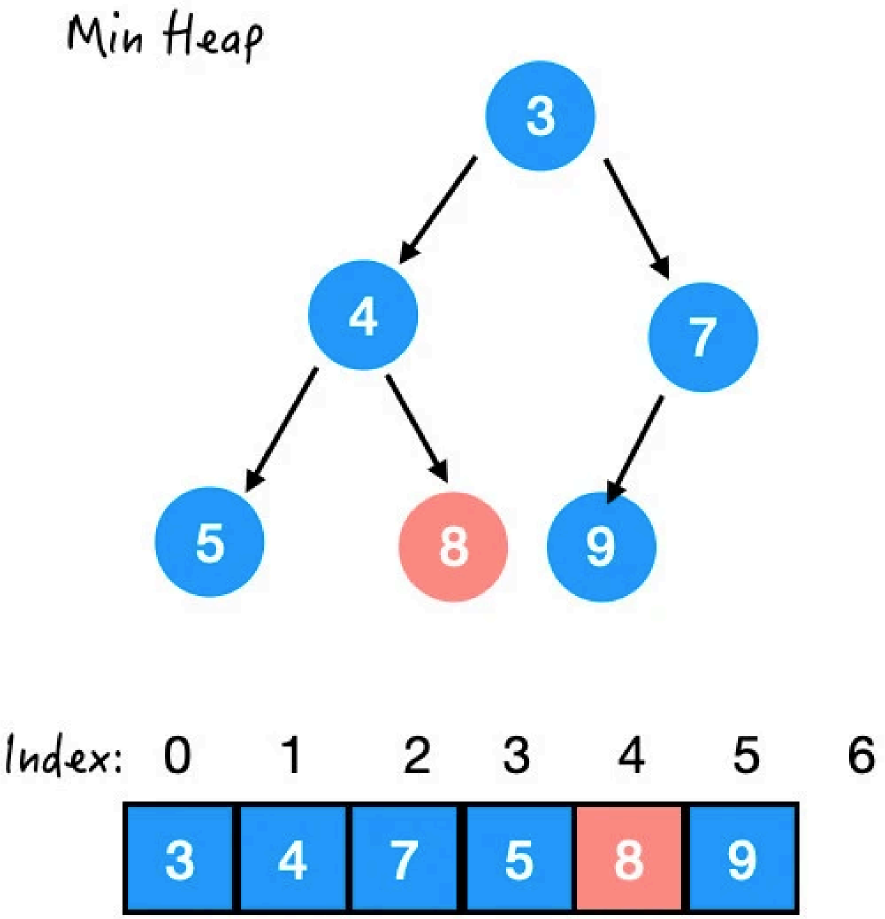
Siftdown: min-heap

- **Finally, we got a new heap**



Siftdown: complexity

- **Again, we only swap with numbers on one branch**
- **The complexity of siftdown operation is also log(n).**



Siftdown: min-heap, root's index is 1

```
procedure siftdown(Heap, i)
//move element i down to its correct position
    c = i * 2
    //Heap[c] < Heap[c+1] for a max-heap
    if Heap[c] > Heap[c + 1]
        c = c +1
    //for max-heap, the condition should be
    //changed to Heap[i] < Heap[c]
    if Heap[i] > Heap[c]
        swap (Heap[i], Heap[c])
        siftdown(Heap,c)
    endif
end
```

Note: *this procedure is not complete – we need to make sure we don't fall off the end of the array.*

Siftdown - an example (max-heap)

This array is a heap with the root missing:

	7	9	6	3	1	5

This is not a heap, fix it:

4	7	9	6	3	1	5

Is it smaller than one of its children?

4	7	9	6	3	1	5

Yes: swap it with its larger child

9	7	4	6	3	1	5

Compare again:

9	7	4	6	3	1	5

Swap

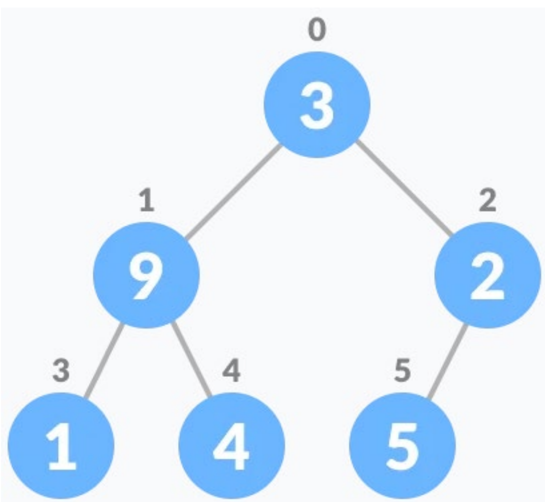
9	7	5	6	3	1	4

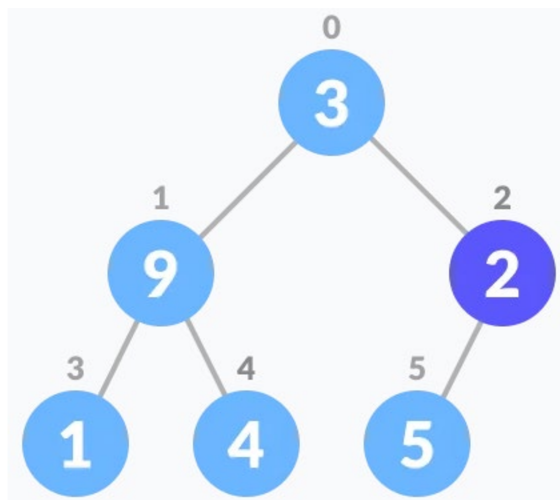
Done

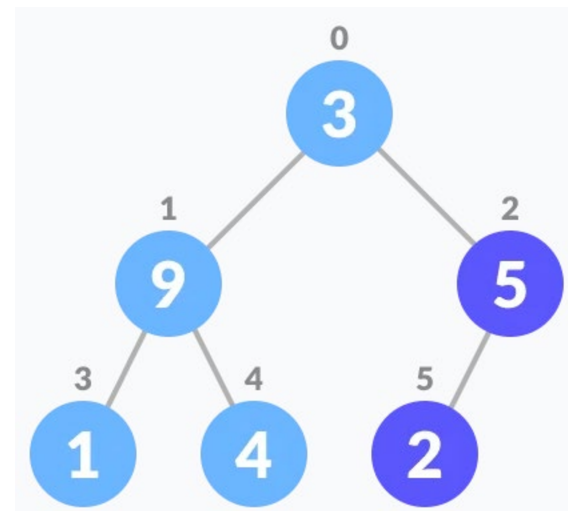
Makeheap/Heapify

1. Create a complete binary tree from the array (not a heap yet)
2. Start from the first index of non-leaf note whose index is given by n/2 -1
3. Set current element i as the largest
4. Siftdown
5. Repeat steps 2 ~ 5

Makeheap/Heapify: max-heap







Makeheap/Heapify

```
procedure makeheap(T[1..n])
    for i = n ÷ 2 to 1 step -1 do
        siftdown(T, i)
    end for
end
```

- The `makeheap` procedure also uses siftdown
- Each element, other than the leaves, is progressively moved into the correct location.
- The complexity of Heapify operation is n. Why?

Makeheap/Heapify

It is because siftdown starts from position n/2, so

- At position n/2, the maximum swap is 1;
- At position n/4, the maximum swap is 2;
- At position n/8, the maximum swap is 3;
- At position n/16, the maximum swap is 4;
- ....
- At position 1, the maximum swap is logn

Total: 1*n/2+2*n/4+3*n/8+4*n/16+..+1*logn = n

`makeheap` in action max-heap

Start with the following array:

7	2	9	5	1	3	8	4

Siftdown 5, compare with 4 no swap needed

7	2	9	5	1	3	8	4

7	2	9	5	1	3	8	4

Siftdown 9, no swaps needed

7	2	9	5	1	3	8	4

7	2	9	5	1	3	8	4

`makeheap` in action max-heap

Siftdown 2, swap with 5

7	2	9	5	1	3	8	4
7	5	9	2	1	3	8	4

Siftdown 2, swap with 4

7	4	9	2	1	3	8	4
7	5	9	4	1	3	8	2

Siftdown 7, swap with 9

7	5	9	4	1	3	8	2
9	5	7	4	1	3	8	2

Siftdown 7, swap with 8

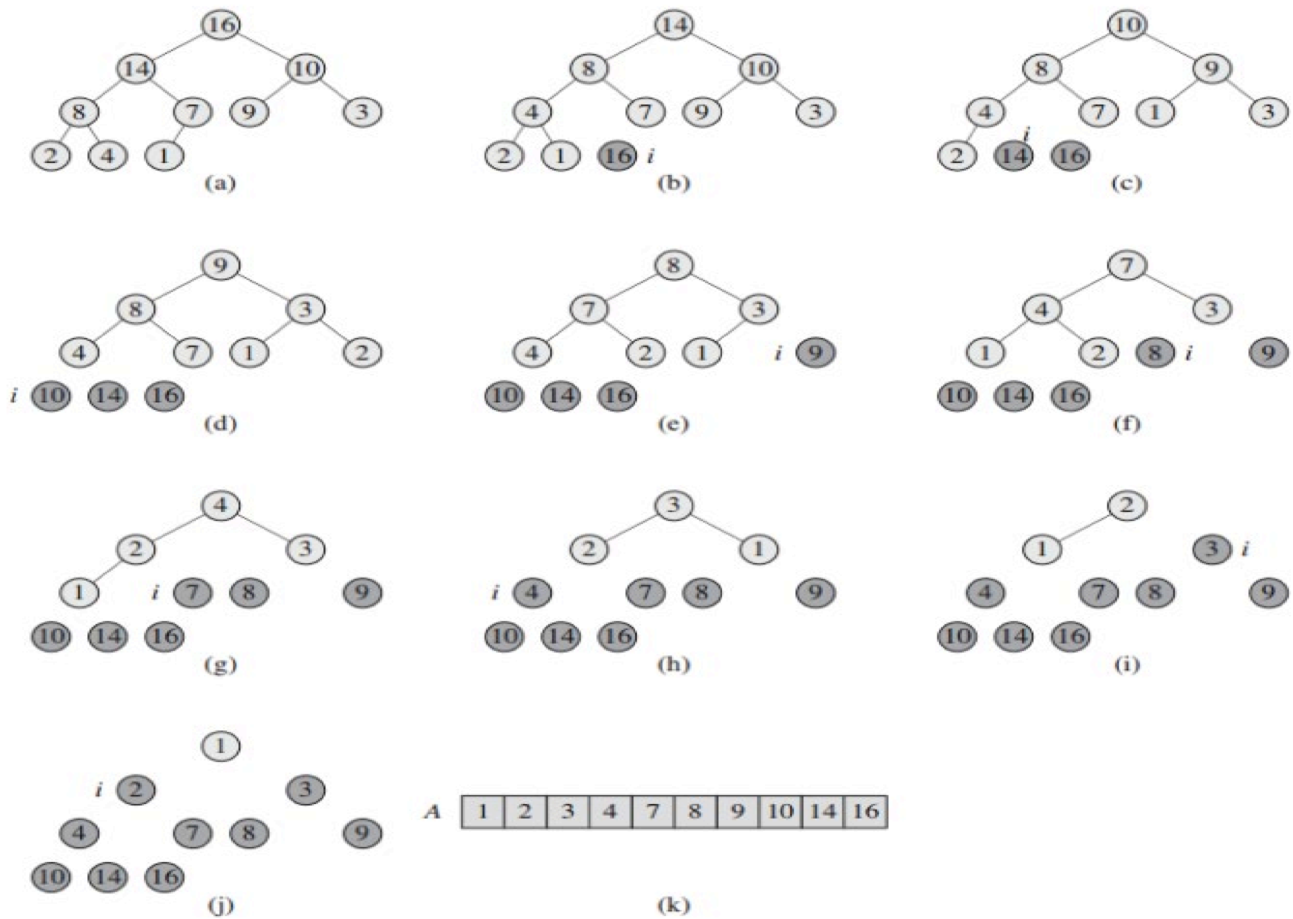
9	5	7	4	1	3	8	2
9	5	8	4	1	3	7	2

Done

Heapsort

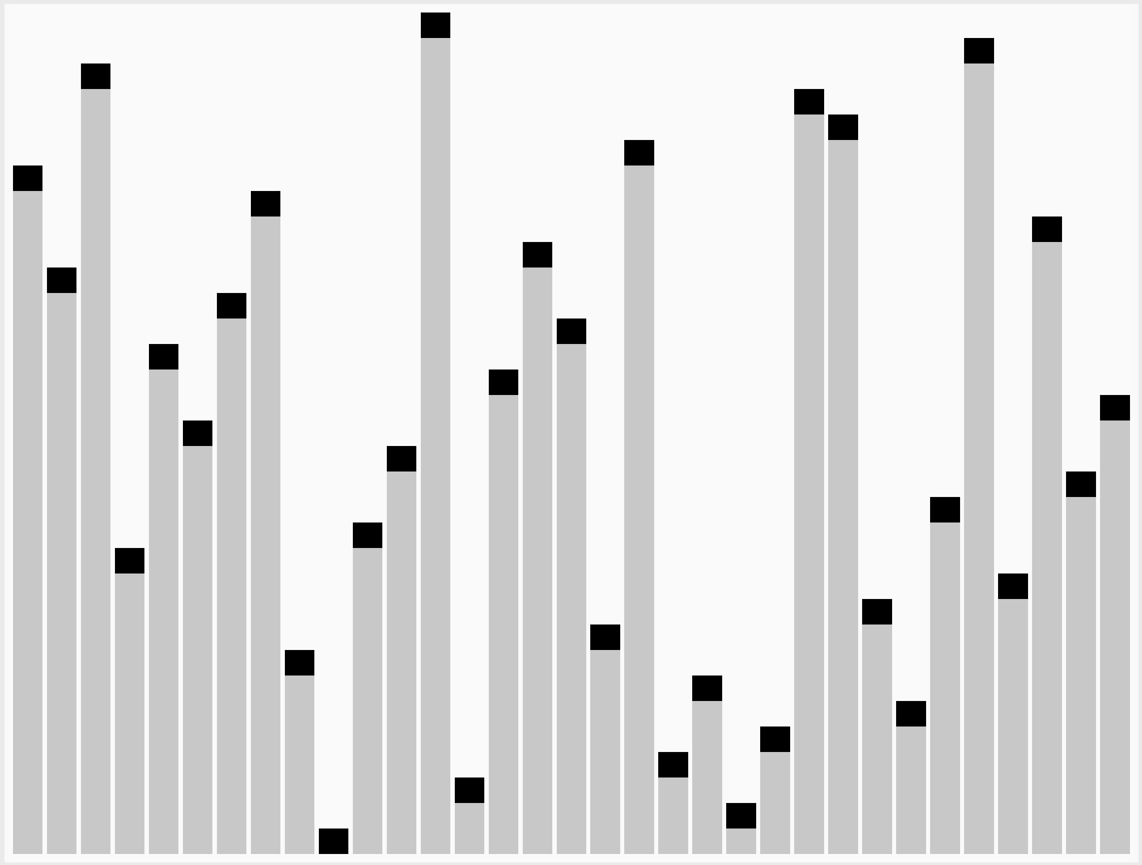
- Heapsort uses the properties of a heap to sort an array.
- It proceeds as follows:
  1. Convert the array into a heap (heapify)
  2. Repeatedly:
     a. Swap the first and last elements of the heap
     b. Reduce the size of the heap by 1
     c. Restore the heap property of the smaller heap (siftdown)
  3. Until the heap contains a single element
- The array is now sorted.
  - Max-heap will sort the list in ascending order, and
  - Min-heap will sort the list in descending order
  - Why?

Heapsort: max-heap -> ascending order



Heapsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$O(n \log n)$ (distinct keys) or $O(n)$ (equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ total $O(1)$ auxiliary



The Algorithm

```
Procedure heapsort(T[1..n])
    makeheap(T)
    for i = n to 2 step -1 do
        swap T[1] and T[i]
        siftdown(T[1 .. i - 1], 1)
```

- This uses two functions:
  - `makeheap` which converts the array into a heap
  - `siftdown` which restores the heap property

Analysis

- 1st step
  - `makeheap` requires roughly n operations.
- 2nd Step
  - `siftdown` requires roughly $\log n$ operations.
  - `siftdown` is repeated $n - 1$ times.
- Overall, heapsort requires roughly $\mathrm{n} + (n - 1) \times \log n$ operations.

Heapsort: max-heap -> ascending order

Starting array:

7	2	9	5	1	3	8	4

After makeheap:

9	5	8	4	1	3	7	2

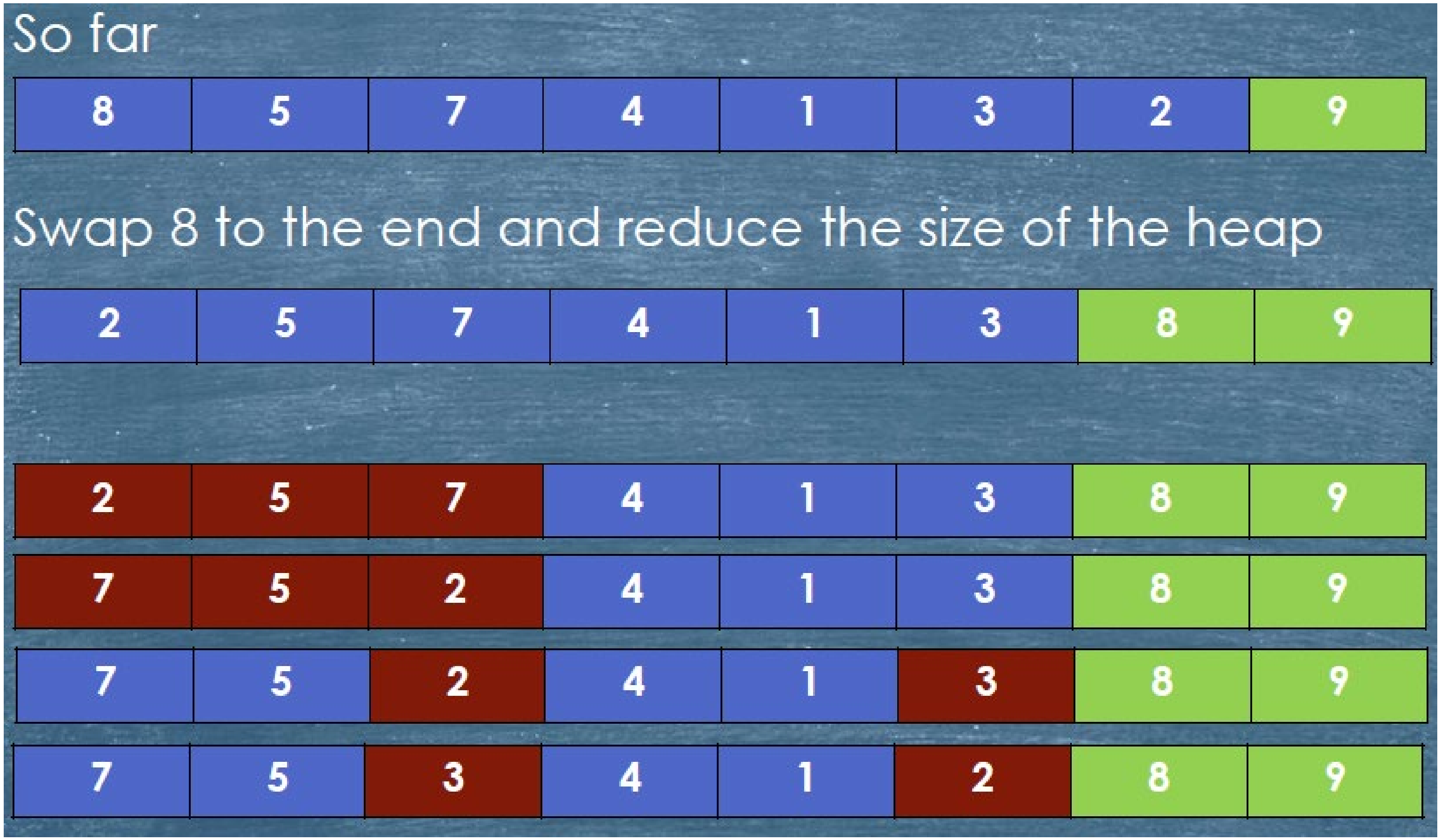
Swap 9 to the end and reduce the size of the heap

2	5	8	4	1	3	7	9

Siftdown 2

2	5	8	4	1	3	7	9
8	5	2	4	1	3	7	9
8	5	2	4	1	3	7	9
8	5	7	4	1	3	2	9

Heapsort: max-heap -> ascending order



Heapsort: max-heap -> ascending order

So far

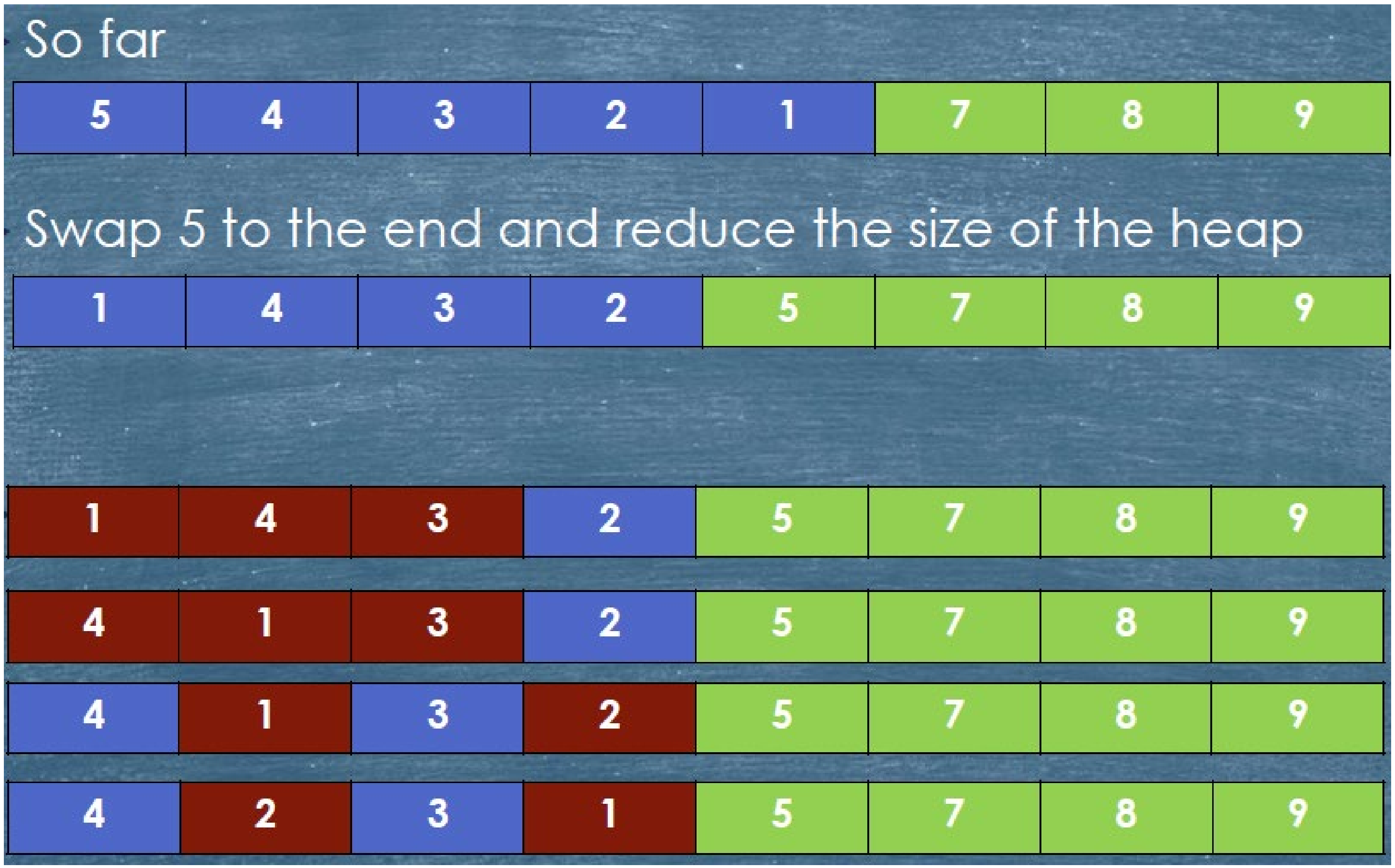
7	5	3	4	1	2	8	9

Swap 7 to the end and reduce the size of the heap

2	5	3	4	1	7	8	9

2	5	3	4	1	7	8	9
5	2	3	4	1	7	8	9
5	2	3	4	1	7	8	9
5	4	3	2	1	7	8	9

Heapsort: max-heap -> ascending order



Heapsort: max-heap -> ascending order

So far

4	2	3	1	5	7	8	9

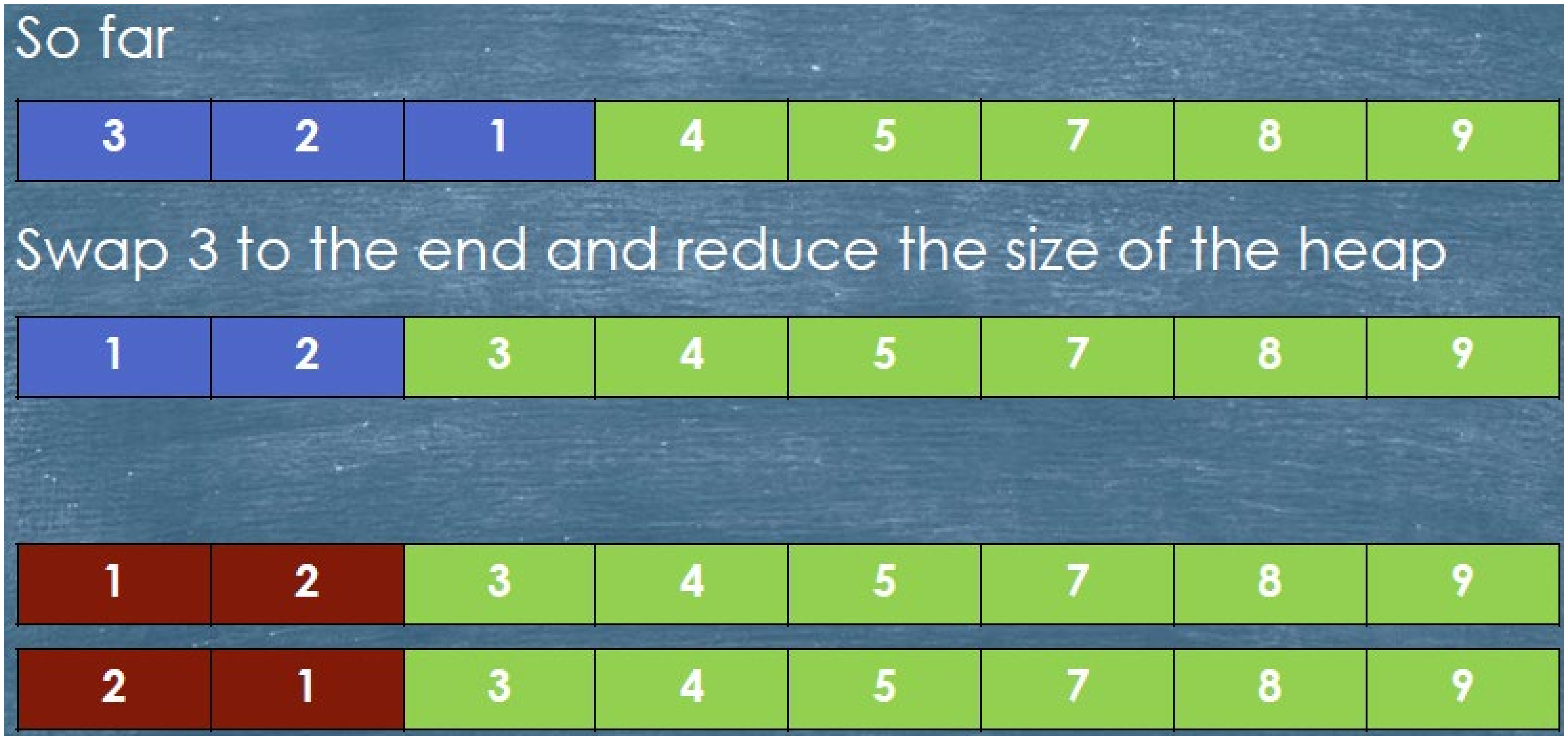
Swap 4 to the end and reduce the size of the heap

1	2	3	4	5	7	8	9

1	2	3	4	5	7	8	9

3	2	1	4	5	7	8	9

Heapsort: max-heap -> ascending order



Heapsort: max-heap -> ascending order

So far

2	1	3	4	5	7	8	9

Swap 2 to the end and reduce the size of the heap

1	2	3	4	5	7	8	9

The heap now has a single element and we are done.

1	2	3	4	5	7	8	9

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 4.1, 5.1 & 6.4
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 6