ADVANCED ANALYSIS OF ALGORITHMS
CPS 5440

SORTING AND SEARCHING: MERGE SORT, HEAPSORT, AND QUICKSORT REVIEW.

HEAP SORT

HEAP – DEFINITION

- A heap is a tree-based data structure in which all nodes follow a specific order.

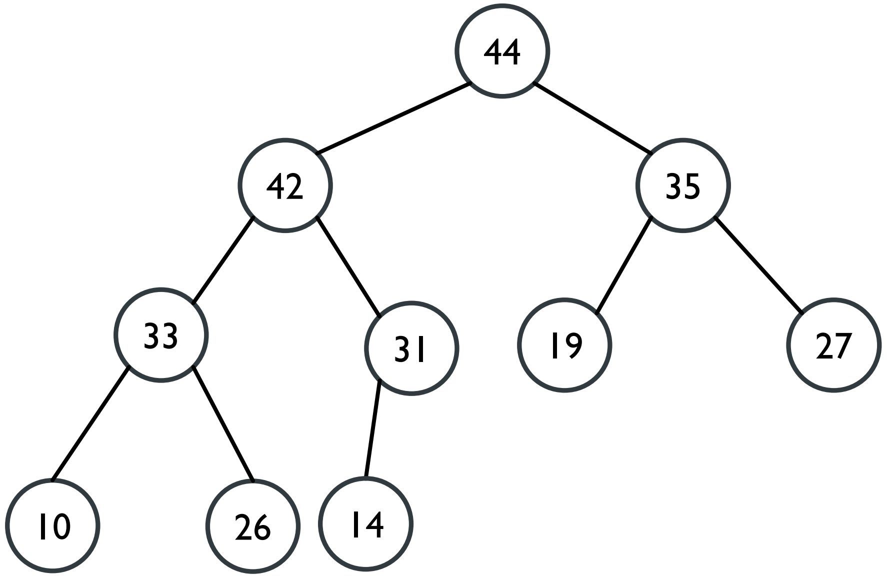
- If X is the parent node of Y, then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.

- Generally, heaps can be of two types:
  - **max-heap**: for any given node C, if P is a parent node of C, then the key of P is greater than or equal to the key of C.
  - **min-heap**: the key of P is less than or equal to the key of C.

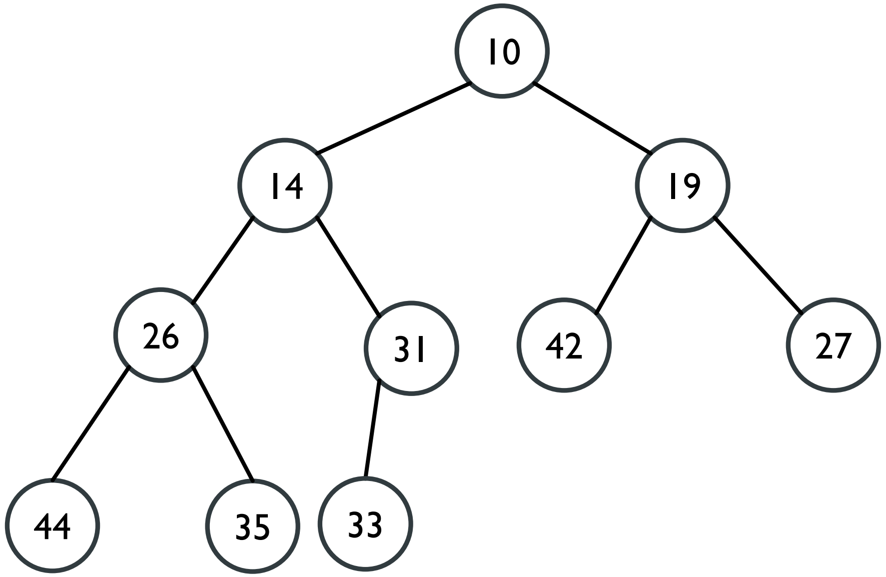
BINARY HEAP – DEFINITION

- A binary heap is defined as a **binary tree** with two additional constraints:
  - **Shape property:** a binary heap is a **complete binary tree**; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
  - **Heap property:** the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.

BINARY HEAP – EXAMPLE



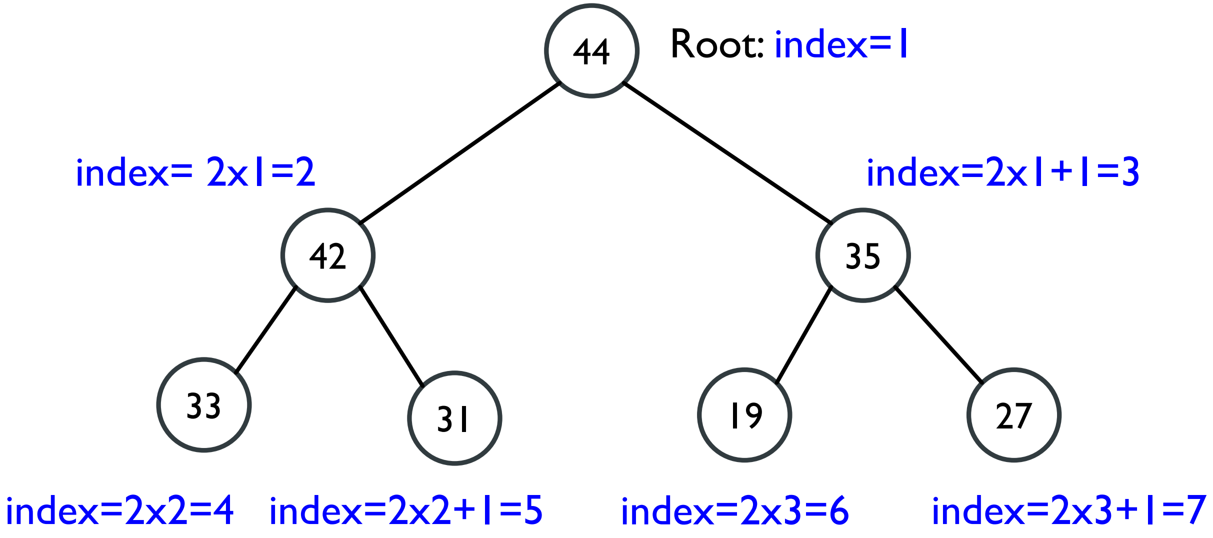
Max-Heap



Min-Heap

BINARY HEAP – REPRESENTATION

- An array can be used to simulate a tree in the following way.
- If we are storing one element at index i in array Arr, then its parent will be stored at index $\frac{i}{2}$ (unless it's a root, as root has no parent) and can be accessed by $Arr[\frac{i}{2}]$, and its left child can be accessed by $Arr[2 * i]$ and its right child can be accessed by $Arr[2 * i + 1]$.
- Index of root will be 1 in an array.



	44	42	35	33	31	19	27
	1	2	3	4	5	6	7

❑Max Heap Insertion Algorithm

Step 1 − Create a new node at the bottom level of the heap at the most left.

Step 2 − Assign new value to the node.

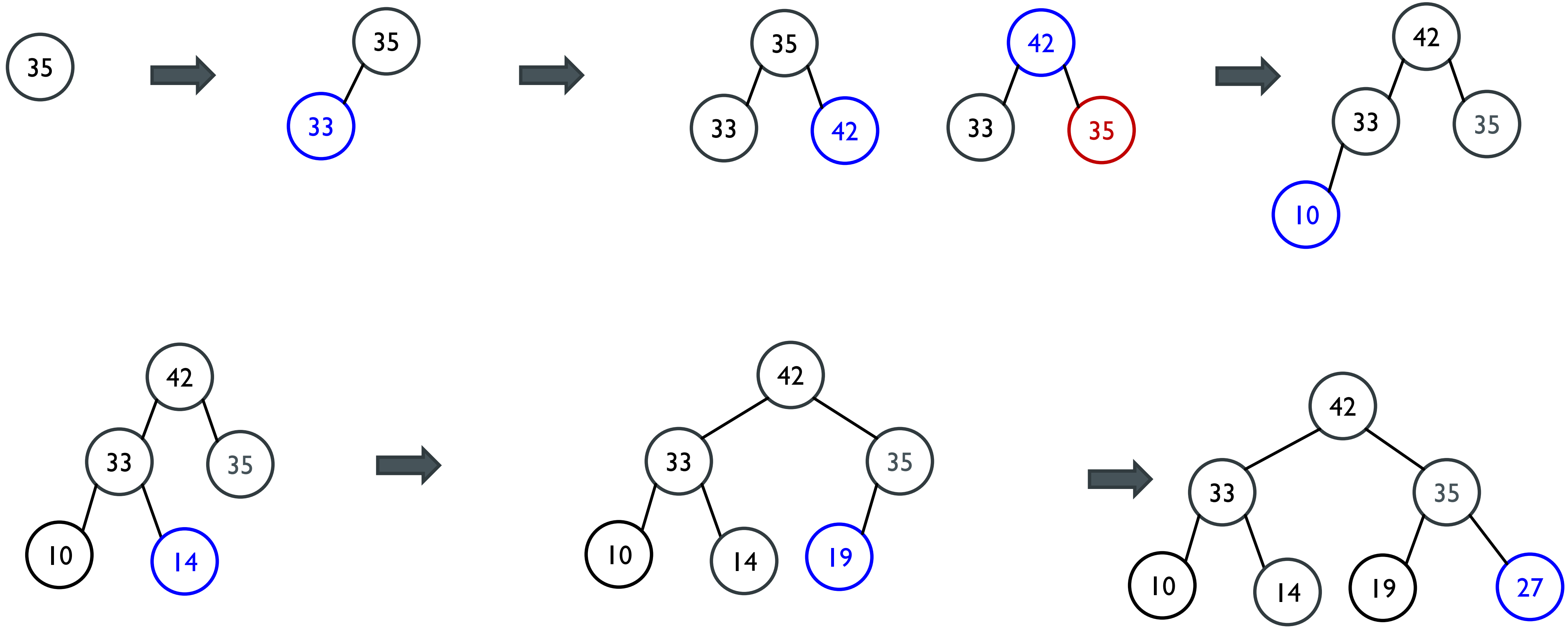
Step 3 − Compare the added element with its parent; if they are in the correct order, stop.

Step 4 − If not, swap the element with its parent and return to the previous step.

The operation has a worst-case time complexity of $O(\log n)$

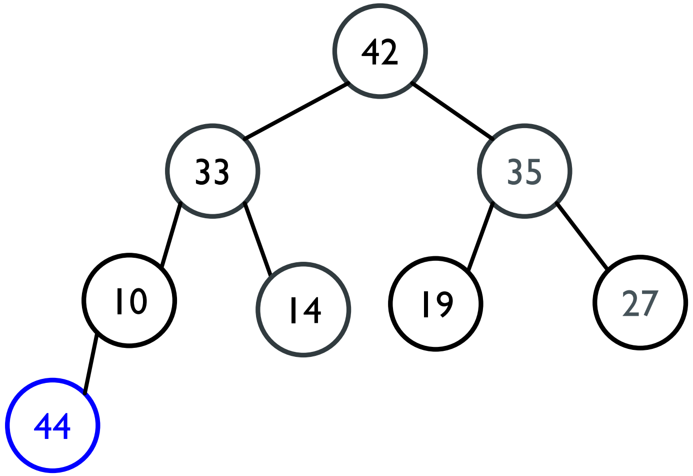
BINARY HEAP – INSERTION

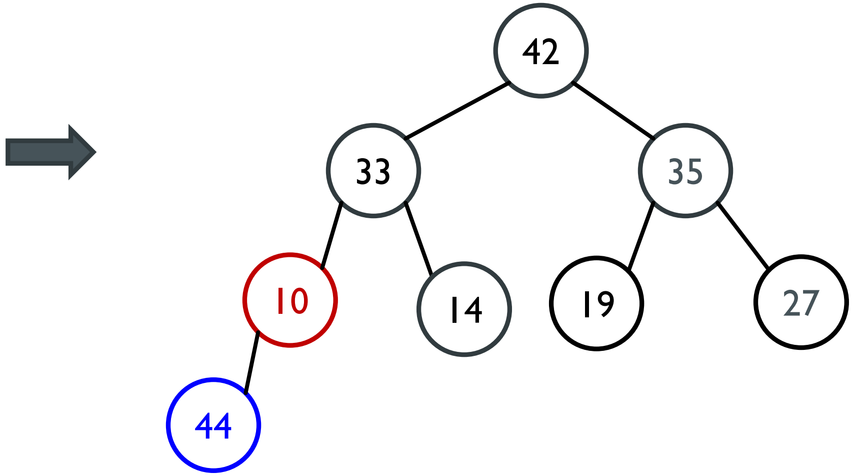
Example: For Input → 35 33 42 10 14 19 27 44 26 31

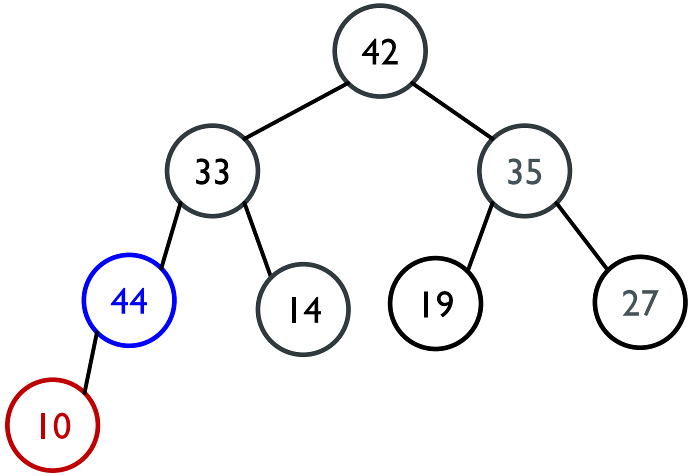


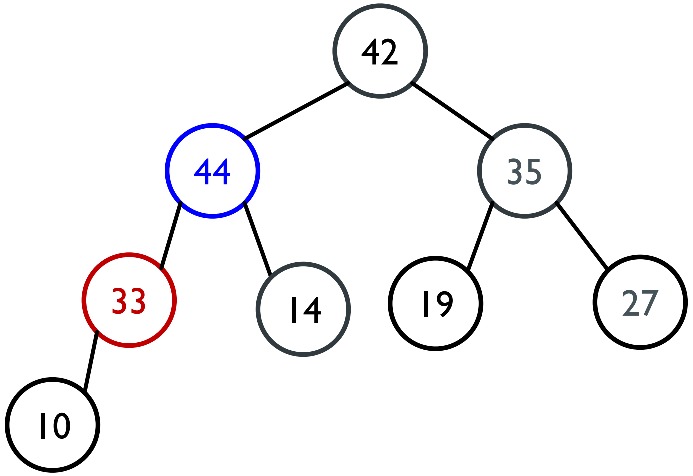
BINARY HEAP – INSERTION

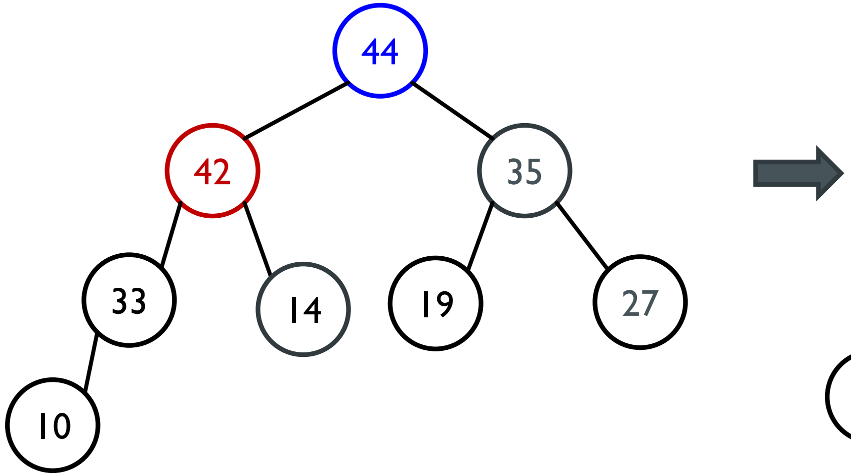
Example: (cont….) For Input → 35 33 42 10 14 19 27 44 26 31

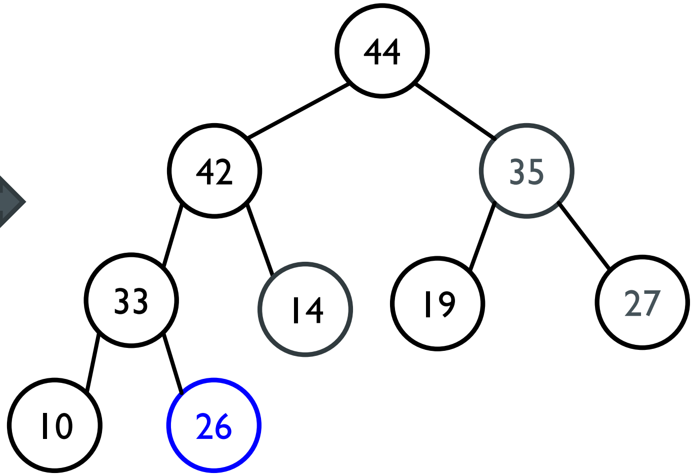






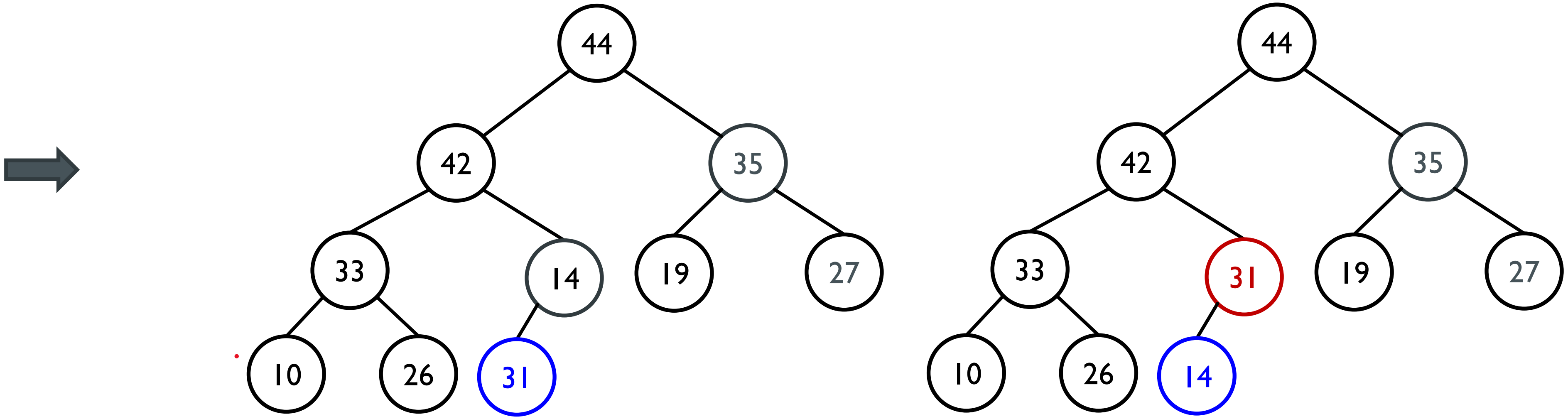






BINARY HEAP – INSERTION

Example: (cont.…) For Input → 35 33 42 10 14 19 27 44 26 31



- Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.

Step 2 − Replace the root of the heap with the last element on the last level.

Step 3 − Compare the new root with its children; if they are in the correct order, stop.

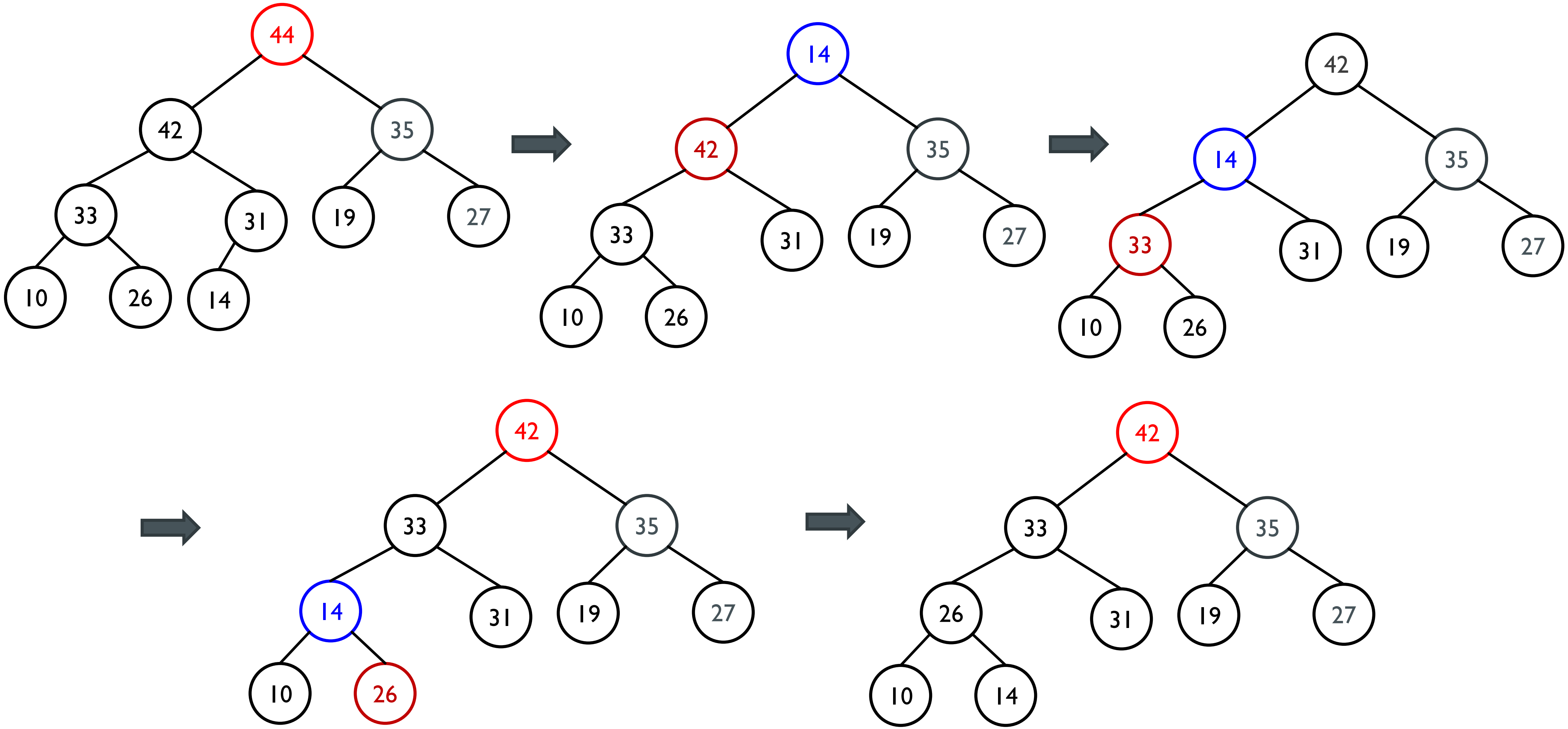
Step 4 − If not, swap the element with one of its children and return to the previous step.

(Swap with its smaller child in a min-heap and its larger child in a max-heap.)

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or $O(\log n)$.

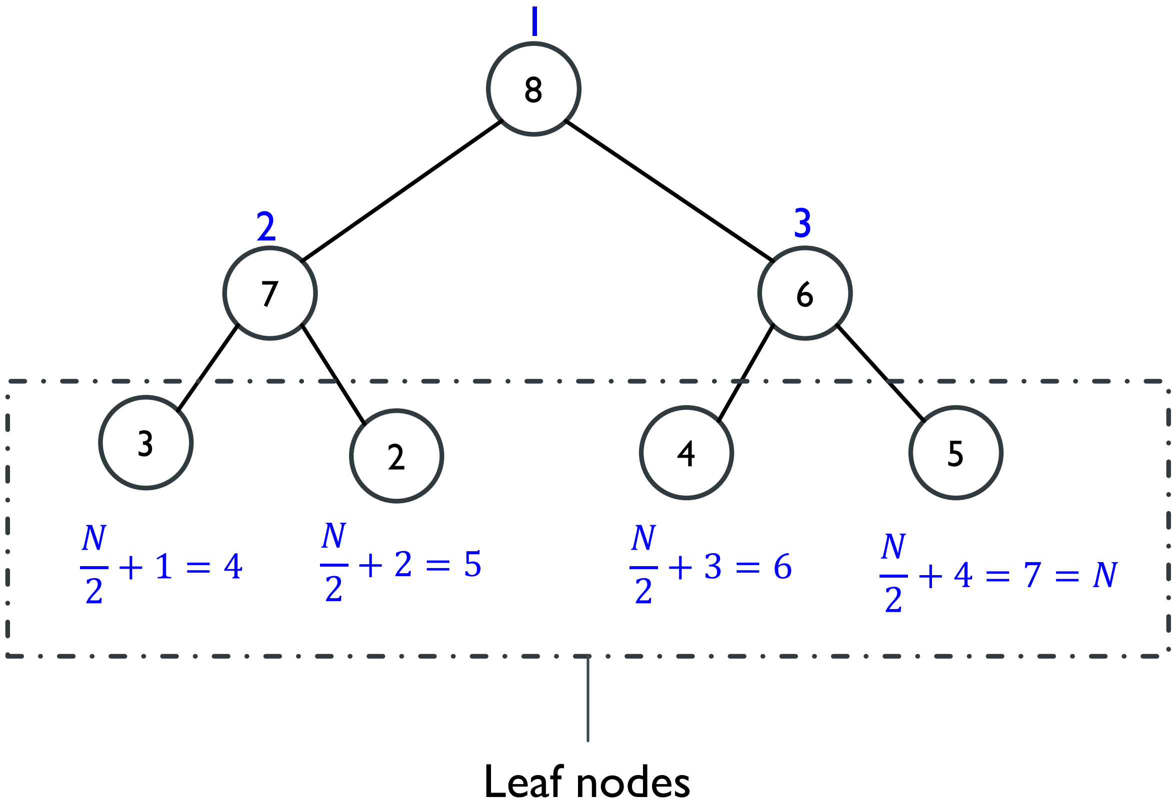
BINARY HEAP – DELETION

Example



BINARY HEAP – BUILD

- Building a heap from an array of n input elements can be done by starting with an empty heap, then successively inserting each element. This approach is easily seen to run in $O(n \log n)$ time: it performs n insertions at $O(\log n)$ cost each
- But, a N element heap stored in an array has leaves indexed by $\frac{N}{2} + 1$, $\frac{N}{2} + 2$, $\frac{N}{2} + 3$ …. Up to N.
- Lets take above example of 7 elements having values $\{8, 7, 6, 3, 2, 4, 5\}$.



	8	7	6	3	2	4	5
	1	2	3	4	5	6	7

$N = 7$

BINARY HEAP – BUILD

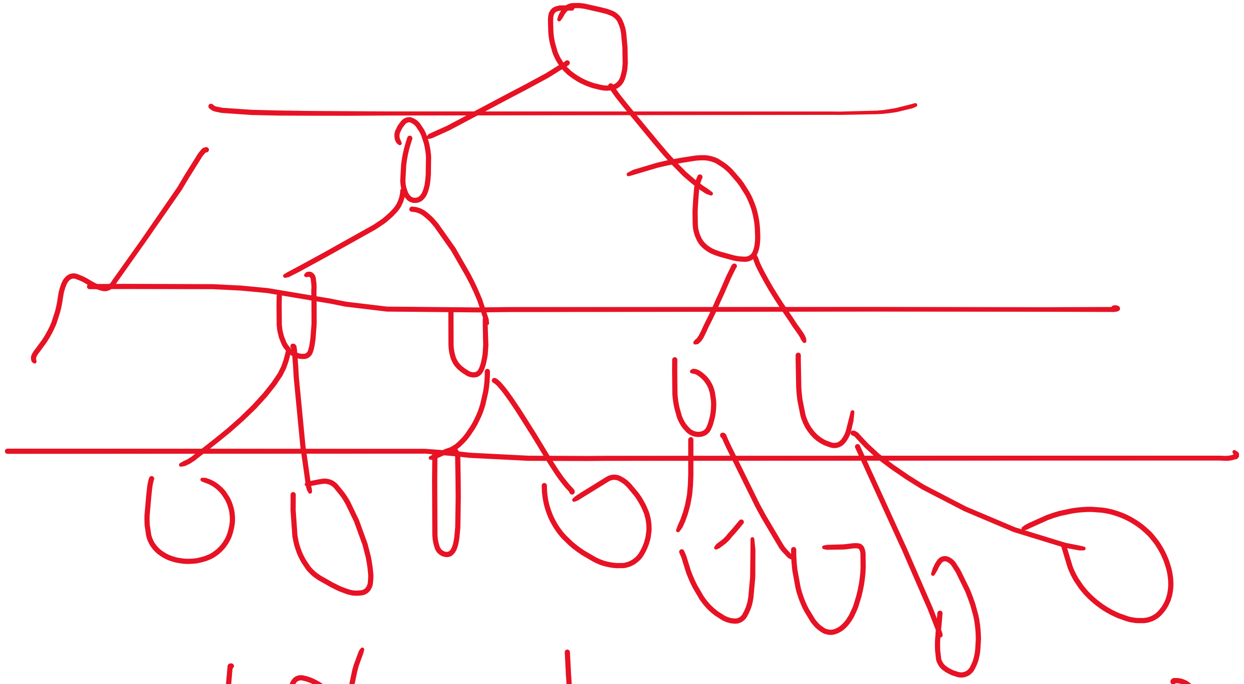
```
void build_maxheap (int Arr[ ], int N) {
        for(int i = N/2 ; i >= 1 ; i-- ){
            max_heapify (Arr, i, N) ;
        }
 }
```

```
void max_heapify (int Arr[ ], int i, int N){
        int left = 2*i                    //left child
        int right = 2*i +1            //right child

       if(left<= N and Arr[left] > Arr[i] ) largest = left;
        else  largest = i;

       if(right <= N and Arr[right] > Arr[largest] )
            largest = right;

        if(largest != i ) {
            swap (Arr[i] , Arr[largest]);
            max_heapify (Arr, largest,N);
        }
    }
```

Complexity: $O(\frac{n}{2}(\log n))$.

- **max_heapify** function has complexity $O(\log n)$ and the
- **build_maxheap** functions runs only $O(\frac{n}{2})$ times

BINARY HEAP – BUILD



$$\left(\frac{n}{2} * 0\right) + \left(\frac{n}{4} * 1\right) + \left(\frac{n}{8} * 2\right) + \cdots + (1 * h)$$

$$h = \log n$$

$$\sum_{k=1}^{h} \frac{k \cdot n}{2^{k+1}} = \Theta(n)$$

BINARY HEAP – BUILD

```
(0 * n/2) + (1 * n/4) + (2 * n/8) + ... + (h * 1).
```

$$
\begin{aligned}
1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + \ldots + (h \cdot 1) &= \sum_{k=1}^{h} \frac{kn}{2^{k+1}} = \frac{n}{4} \sum_{k=1}^{h} \frac{k}{2^{k-1}} \\
&< \frac{n}{4} \sum_{k=1}^{\infty} \frac{k}{2^{k-1}} = \frac{n}{4} \sum_{k=1}^{\infty} k x^{k-1}, \qquad x = \frac{1}{2} \\
&= \frac{n}{4} \frac{d}{dx} \left[\sum_{k=0}^{\infty} x^k \right] = \frac{n}{4} \frac{d}{dx} \left[\frac{1}{1-x} \right] \\
&= \frac{n}{4} \frac{1}{(1-x)^2} = \frac{n}{4} \frac{1}{(1-1/2)^2)} = n.
\end{aligned}
$$

If you aren't sure why each of those steps works, here is a justification for the process in words:

- The terms are all positive, so the finite sum must be smaller than the infinite sum.
- The series is equal to a power series evaluated at *x=1/2*.
- That power series is equal to (a constant times) the derivative of the Taylor series for *f(x)=1/(1-x)*.
- *x=1/2* is within the interval of convergence of that Taylor series.
- Therefore, we can replace the Taylor series with *1/(1-x)*, differentiate, and evaluate to find the value of the infinite series.

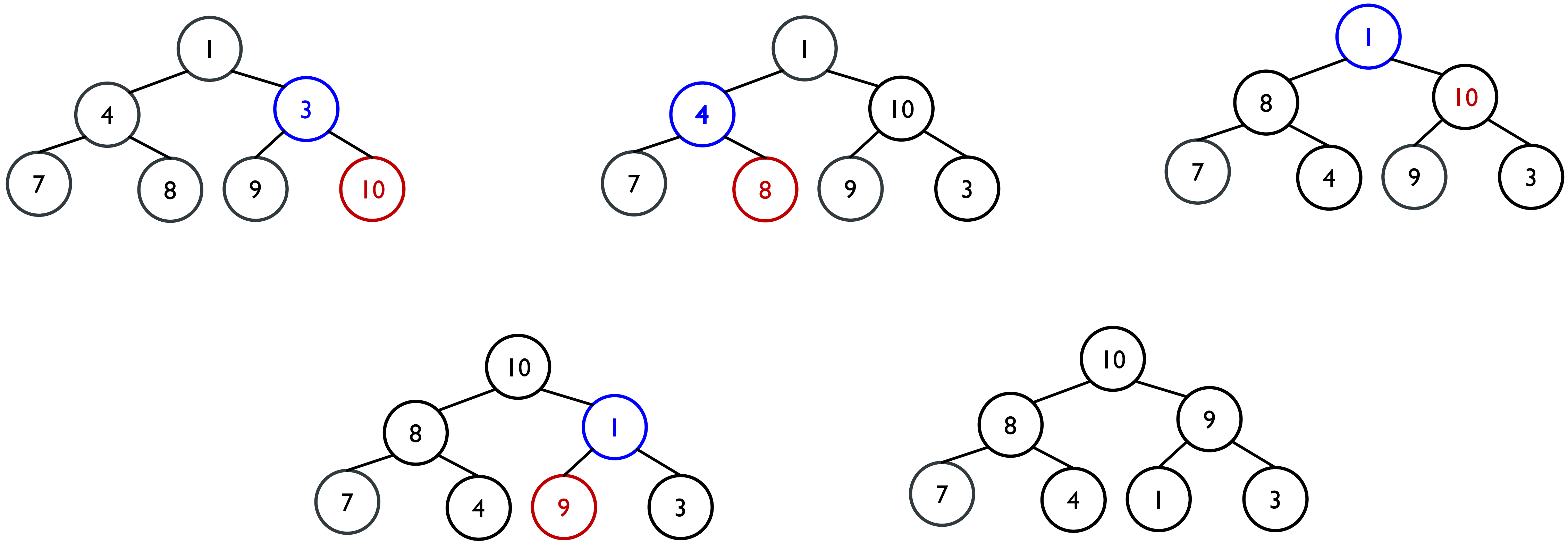
Since the infinite sum is exactly *n*, we conclude that the finite sum is no larger, and is therefore, *O(n)*.

[How can building a heap be O(n) time complexity?]

BINARY HEAP – BUILD

	1	**4**	**3**	**7**	**8**	**9**	**10**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

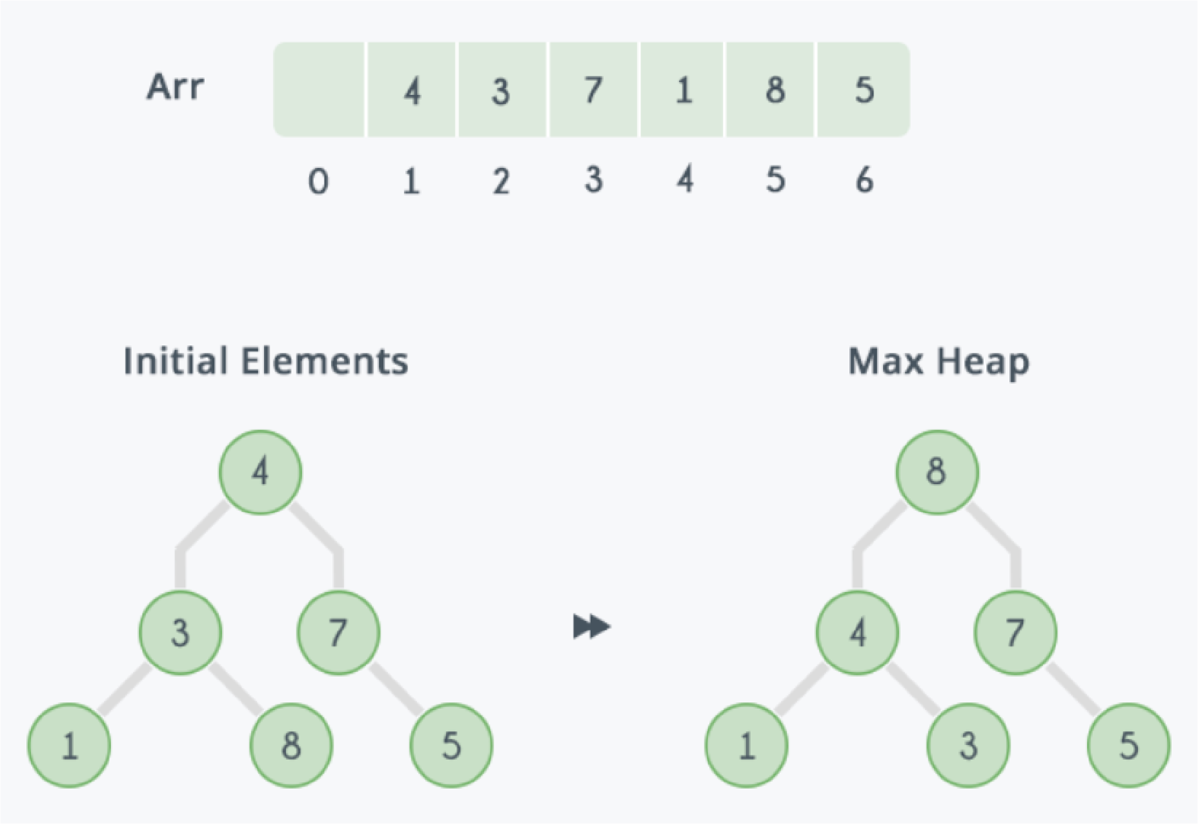
Here $N = 7$, so starting from node having index $\frac{N}{2} = 3$, (also having value 3 in the above diagram), we will call max_heapify from index $\frac{N}{2}$ $to\ 1$.



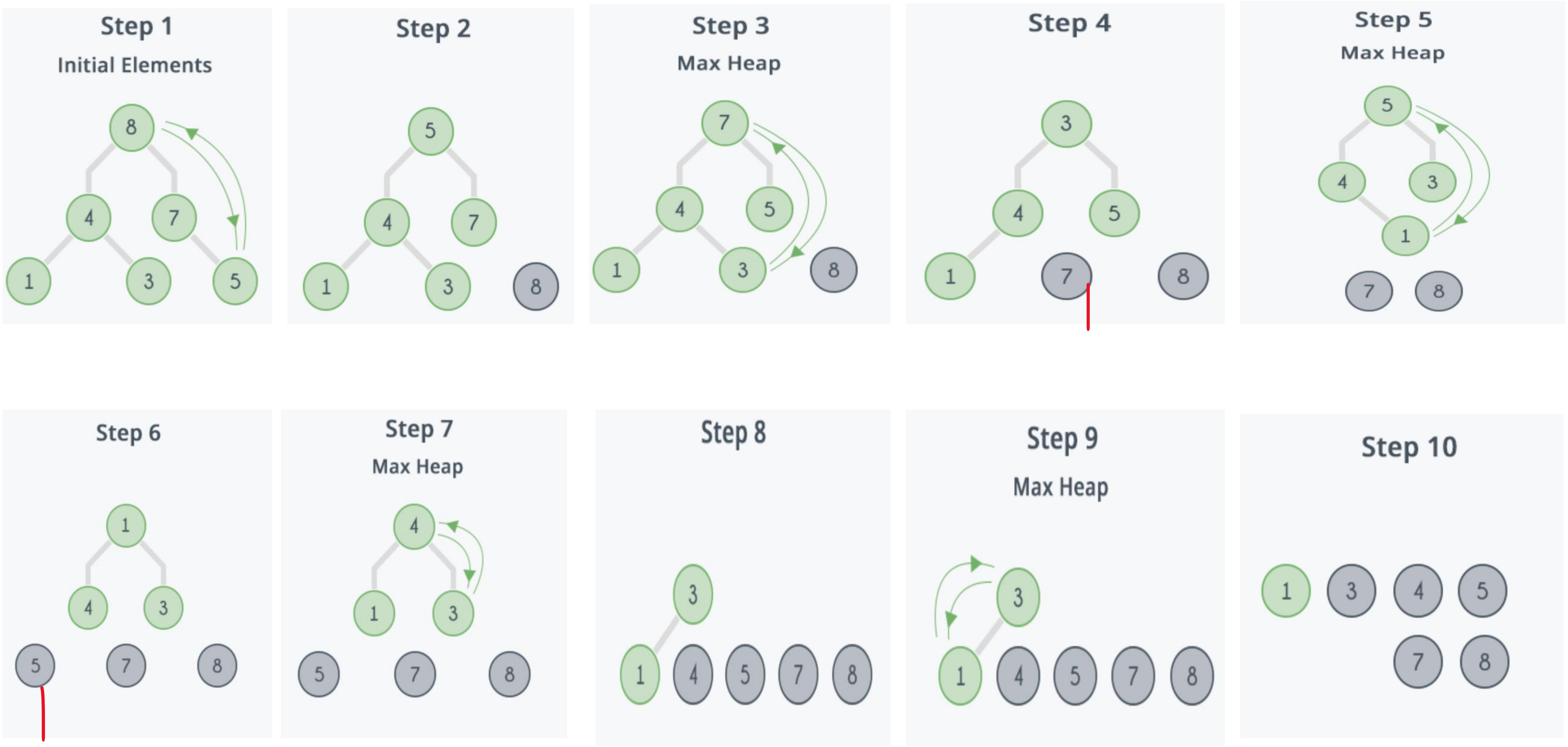
BINARY HEAP – APPLICATIONS

❑ Heap Sort: $(O(n * \log(n)))$

- Build a max heap from the input data $O(N)$
- At this point, the largest item is stored at the root of the heap. Replace it with the heap's last item, then reduce the heap's size by 1. Finally, heapify the root of the tree.
- Repeat the above steps while the heap size is greater than 1.
- **Complexity:** We run max_heapify $N - 1$ times in heap_sort function, therefore complexity of heap_sort function is $O(NlogN)$.

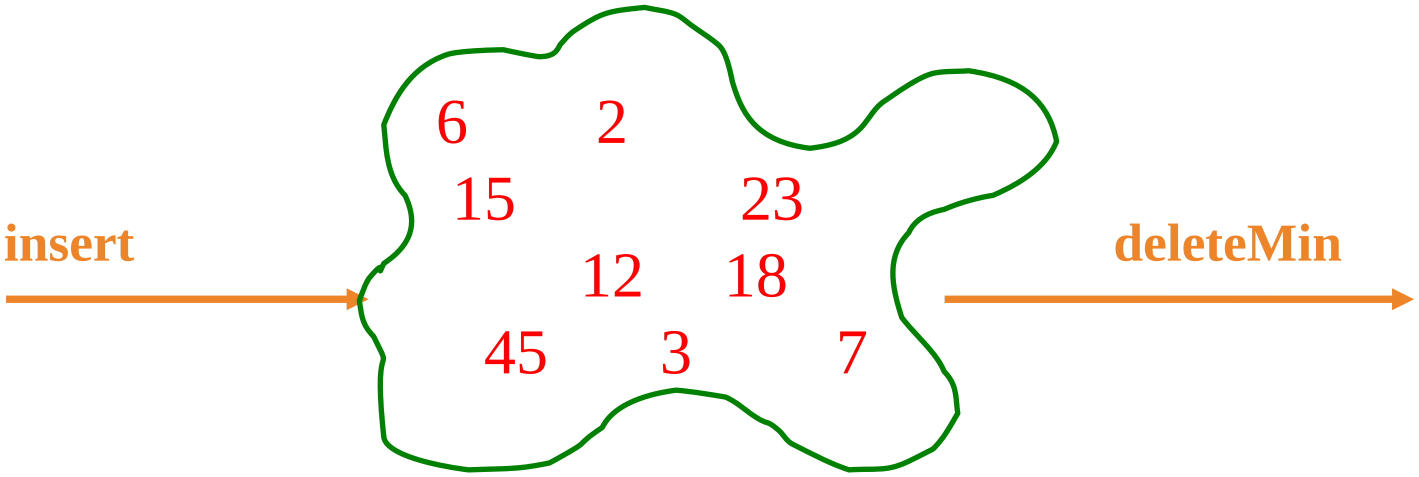


BINARY HEAP – APPLICATIONS



❑ Recall Queues

- FIFO: First-In, First-Out
- Some contexts where this seems right?
- Some contexts where some things should be allowed to skip ahead in the line?
- Queues that Allow Line Jumping
  - Need a new ADT
  - Operations: Insert an Item, Remove the "Best" Item



BINARY HEAP – APPLICATIONS

❑ PRIORITY QUEUE ADT

- **Data**: collection of data with priority
- **Operations:** insert, deleteMin
- **Property**: for two elements in the queue, x and y, if x has a **lower** priority value than y, x will be deleted before y
- **Applications:**
  - Graph searching: Dijkstra's algorithm, Prim's algorithm
  - Select print jobs in order of decreasing length
  - Forward packets on routers in order of urgency
  - Select most frequent symbols for compression
  - Sort numbers, picking minimum first
  - Anything ***greedy***

❑ PRIORITY QUEUES PERFORMANCE COST SUMMARY

Operation	Linked List (worst-case)	Binary Heap (worst-case)	Binomial Heap (worst-case)	Fibonacci Heap (amortized)	Relaxed Heap (worst-case)
Make-heap	1	1	1	1	1
Is-empty	1	1	1	1	1
Insert	1	$\log n$	$\log n$	1	1
Extract-min	n	$\log n$	$\log n$	$\log n$	$\log n$
Decrease-key	n	$\log n$	$\log n$	1	1
Delete	n	$\log n$	$\log n$	$\log n$	$\log n$
Union	1	n	$\log n$	1	1
Find-min	n	1	$\log n$	1	1

QUICK SORT

QUICK SORT ALGORITHM

Given an array of n elements (e.g., integers):

- If array only contains one element, return
- Else
  - Pick one element to use as $\boldsymbol{pivot}$.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

EXAMPLE

- We are given an array of n integers to sort:

40	20	10	80	60	50	7	30	100

PICK PIVOT ELEMENT

- There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

PARTITIONING ARRAY

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements $\geq$ pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

EXAMPLE

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

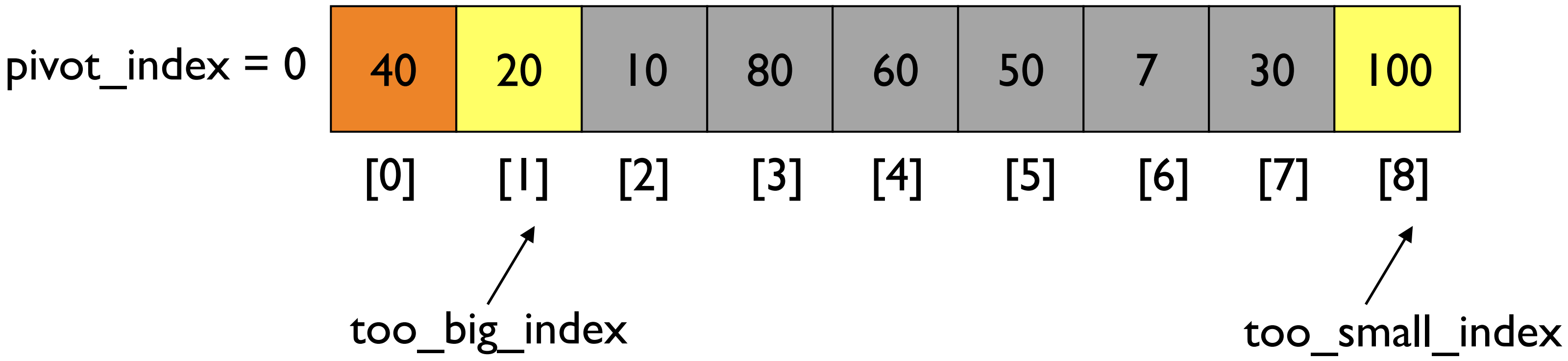
too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]

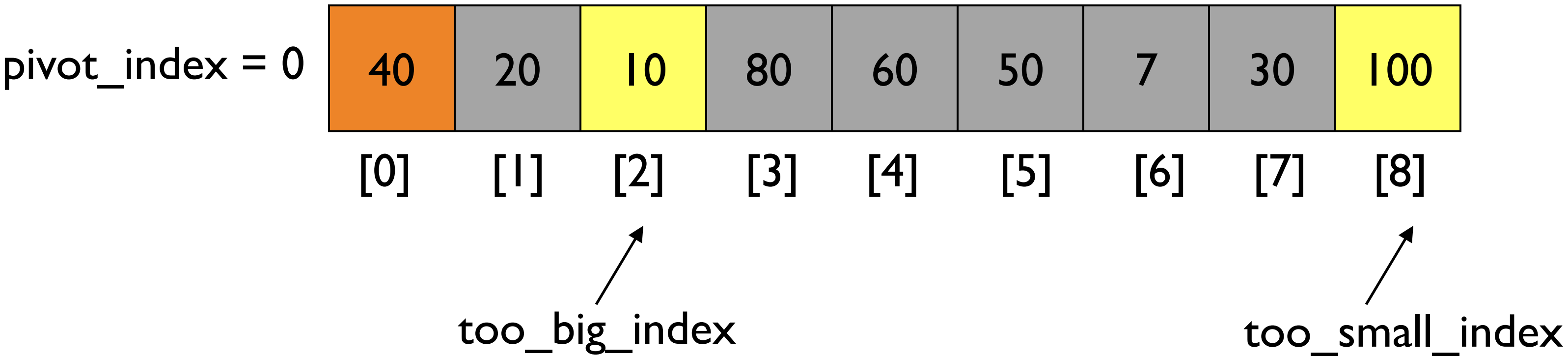
    ++too_big_index



EXAMPLE

1. While data[too_big_index] <= data[pivot]

   ++too_big_index



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index

pivot_index = 0

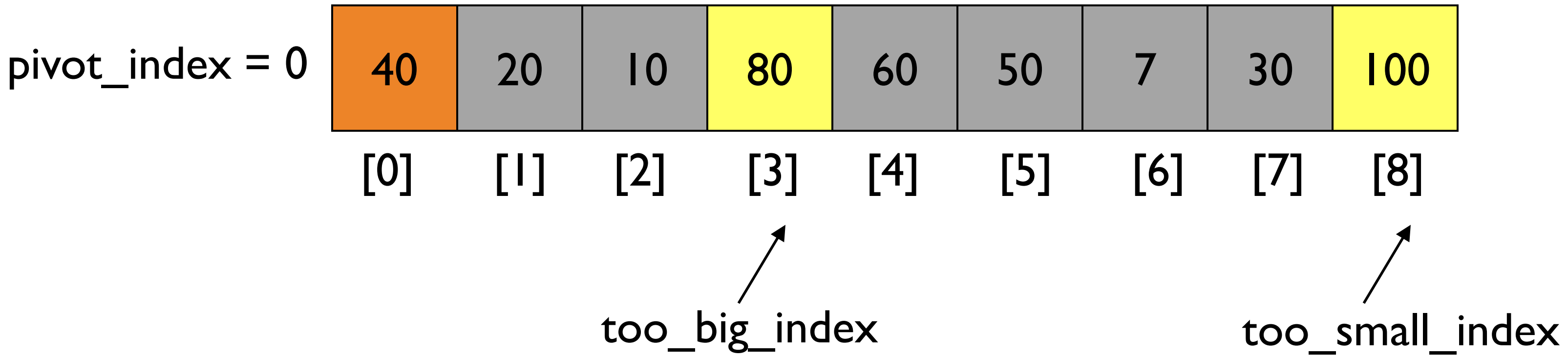
40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index

pivot_index = 0

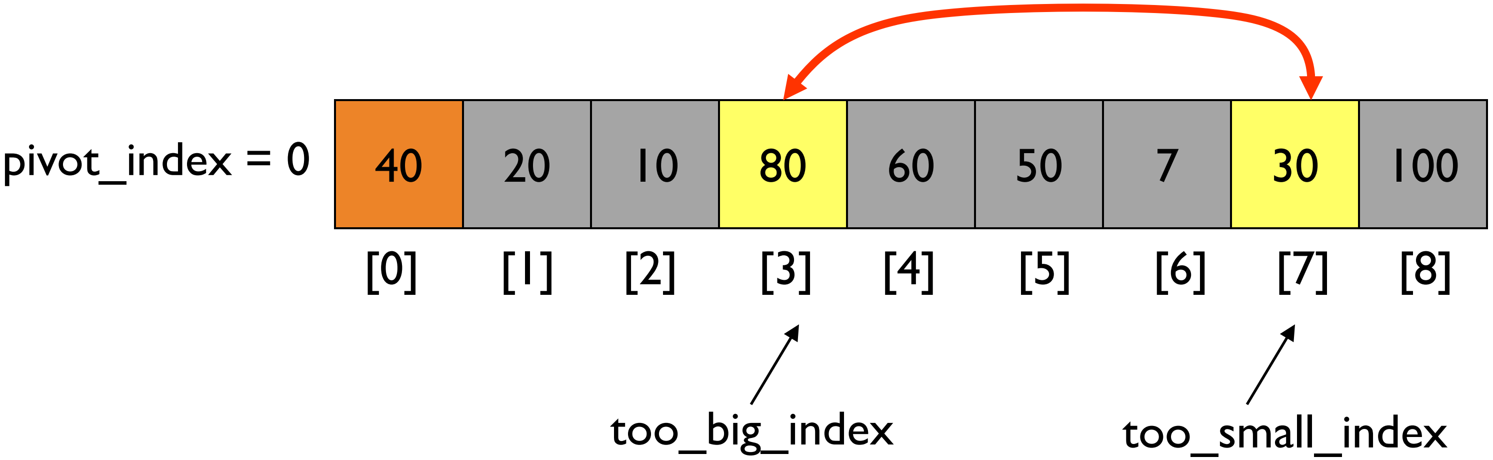
40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

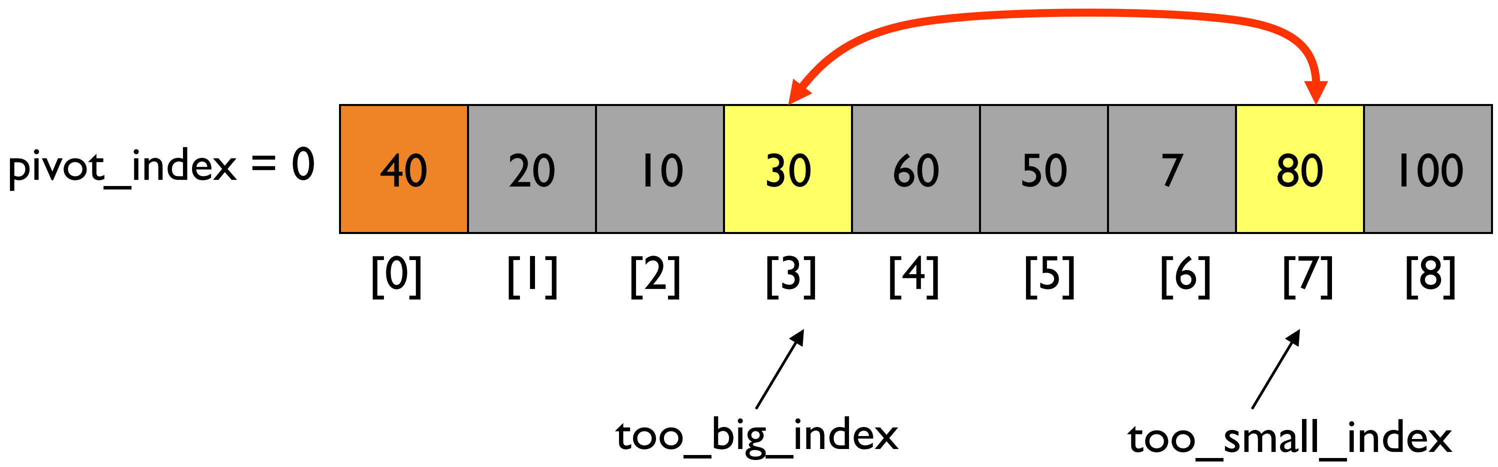
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



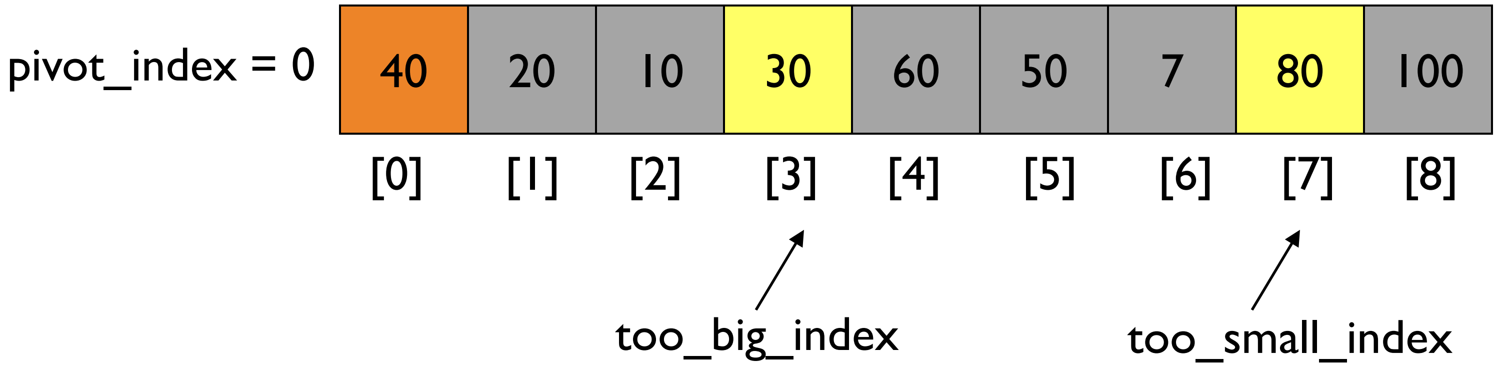
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



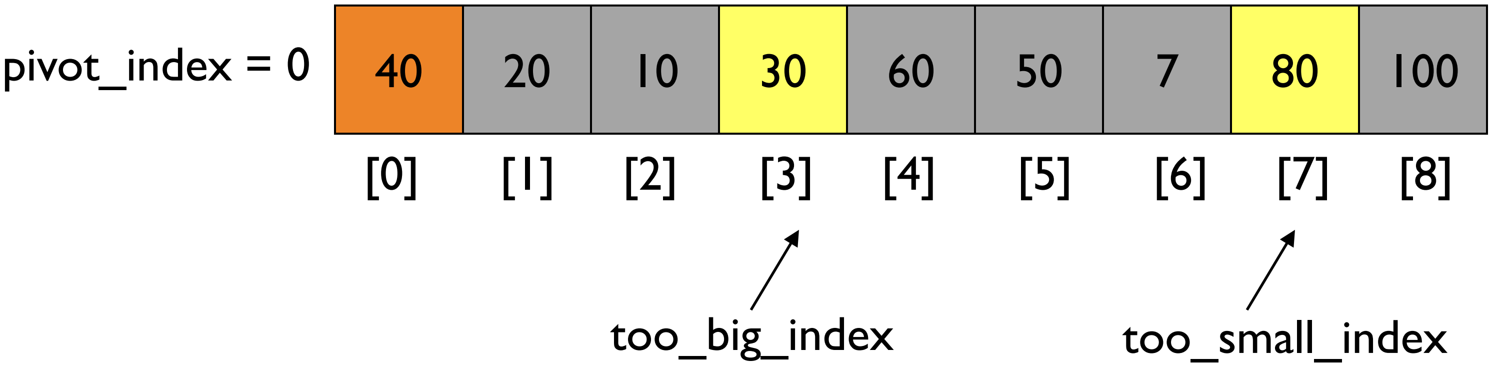
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



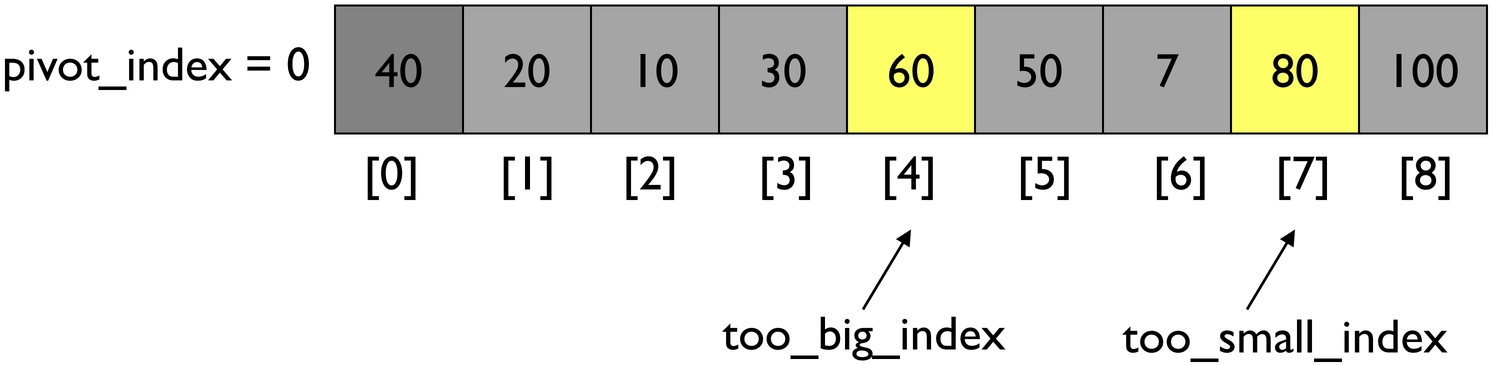
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



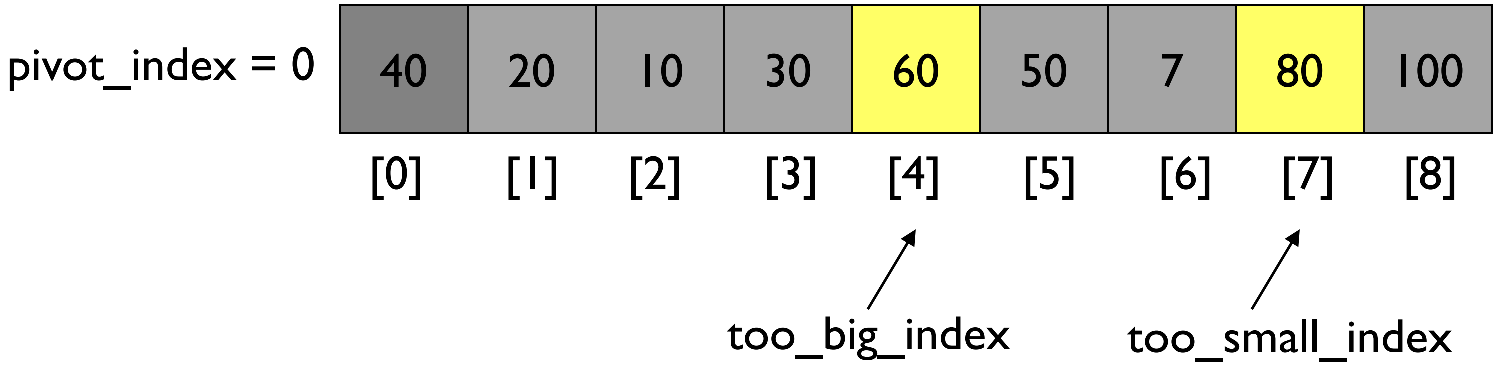
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



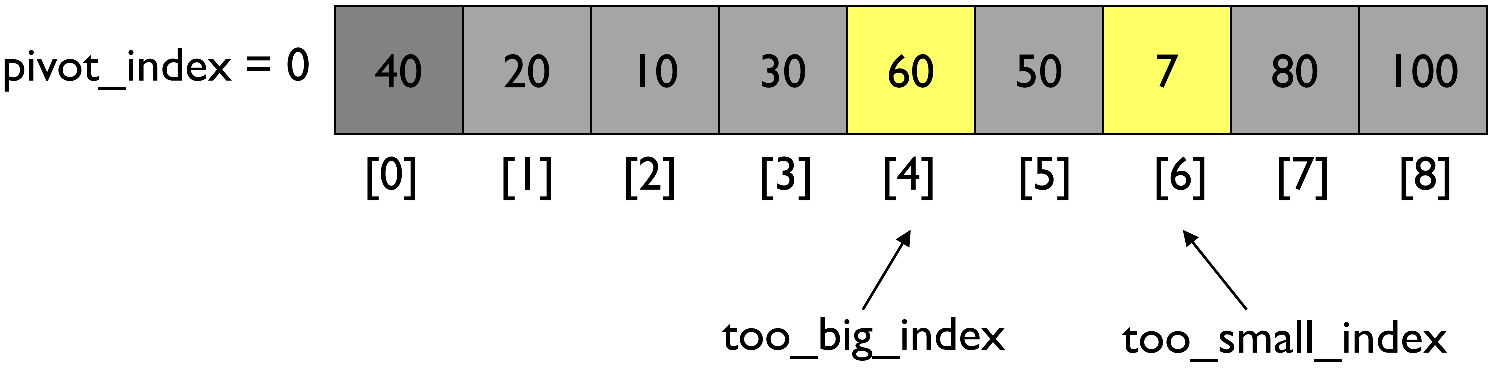
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



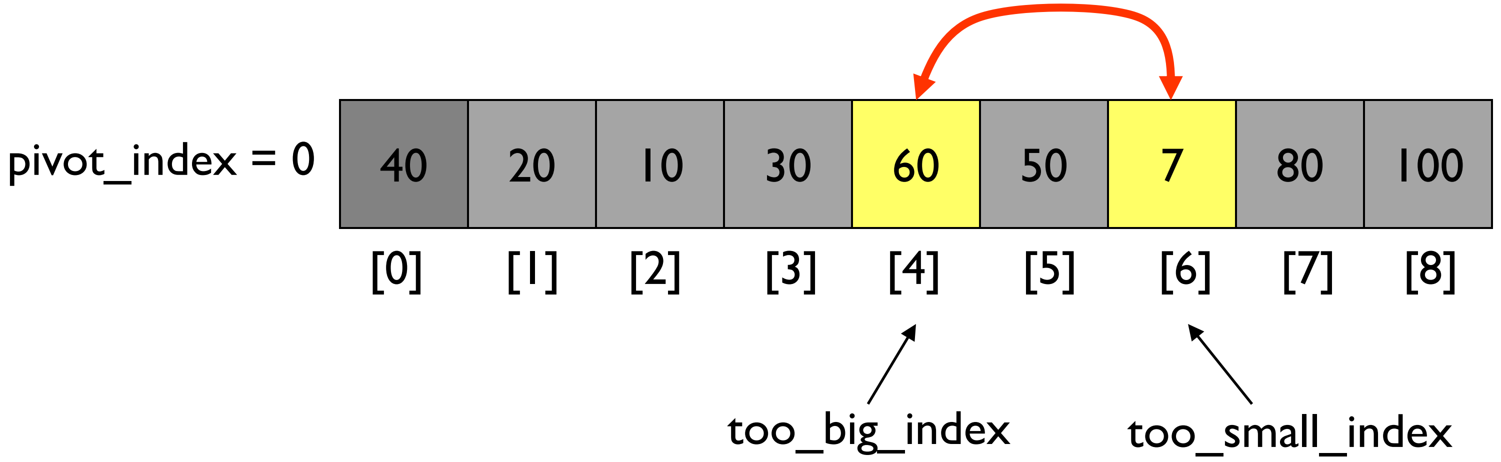
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index → [5]

too_small_index → [6]

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index → [5]

too_small_index → [6]

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index   too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

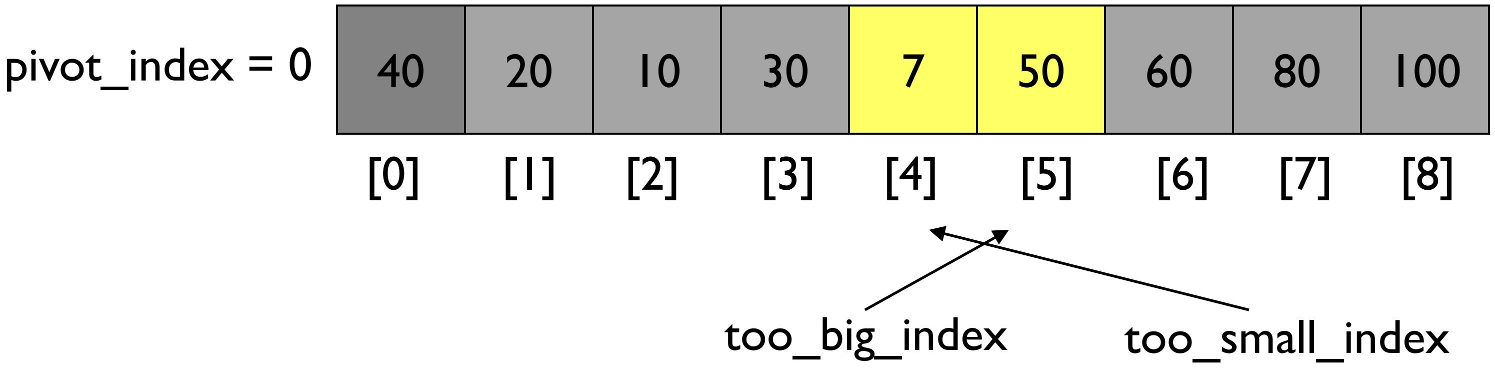
pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	40	20	10	30	7	50	60	80	100

too_big_index too_small_index

EXAMPLE

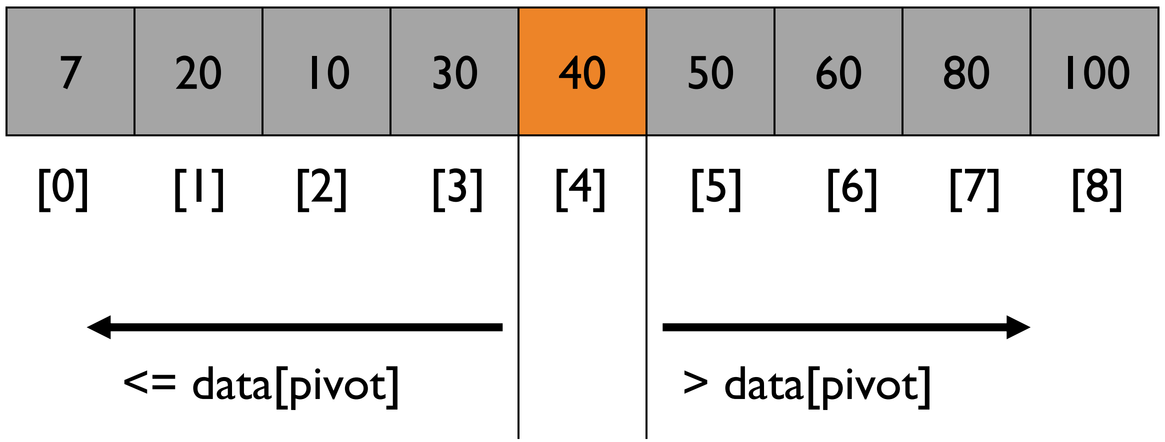
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 4

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

PARTITION RESULT



RECURSION: QUICKSORT SUB-ARRAYS

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]

> data[pivot]

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

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- What is best case running time?

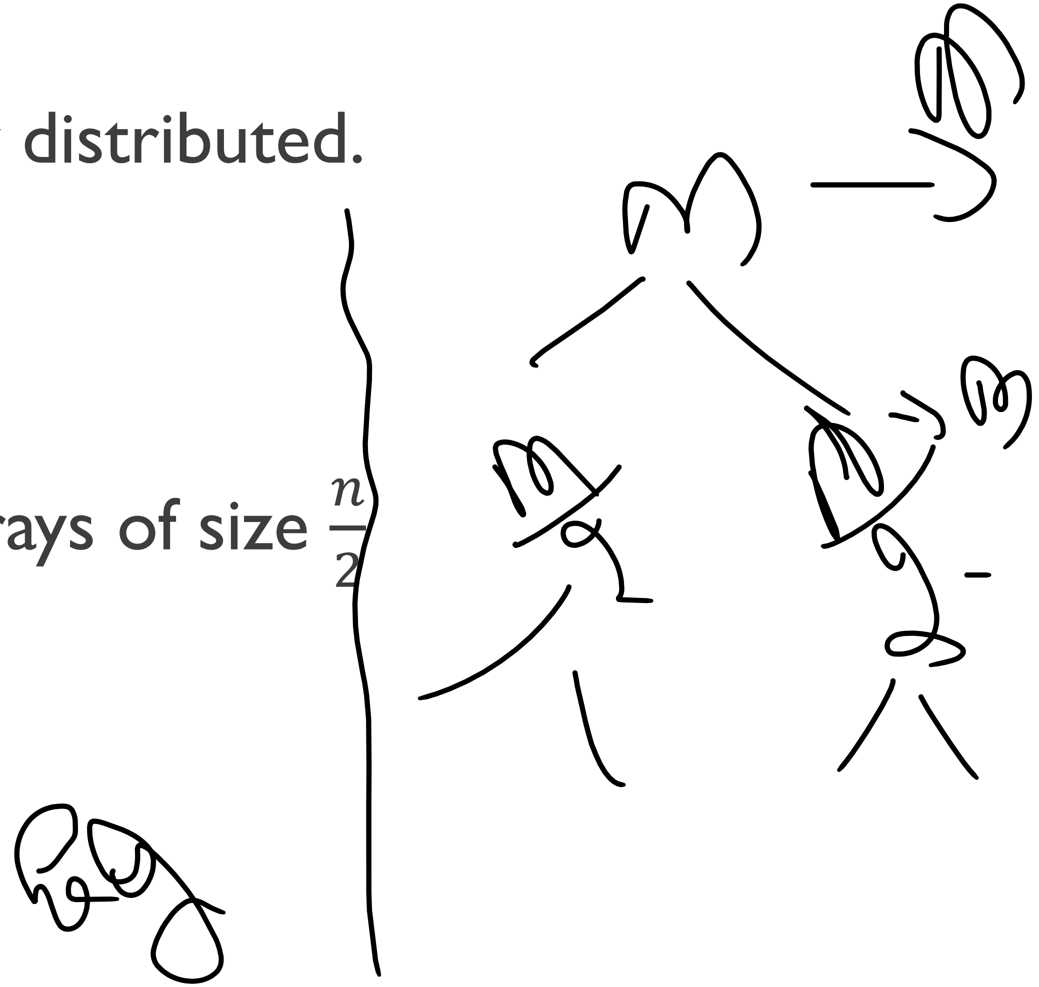
QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $\frac{n}{2}$
    2. Quicksort each sub-array

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $\frac{n}{2}$
    2. Quicksort each sub-array
  - Depth of recursion tree?

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + 1$

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $\frac{n}{2}$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(logn)$



QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $\frac{n}{2}$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(logn)$
  - Number of accesses in partition?

QUICKSORT ANALYSIS

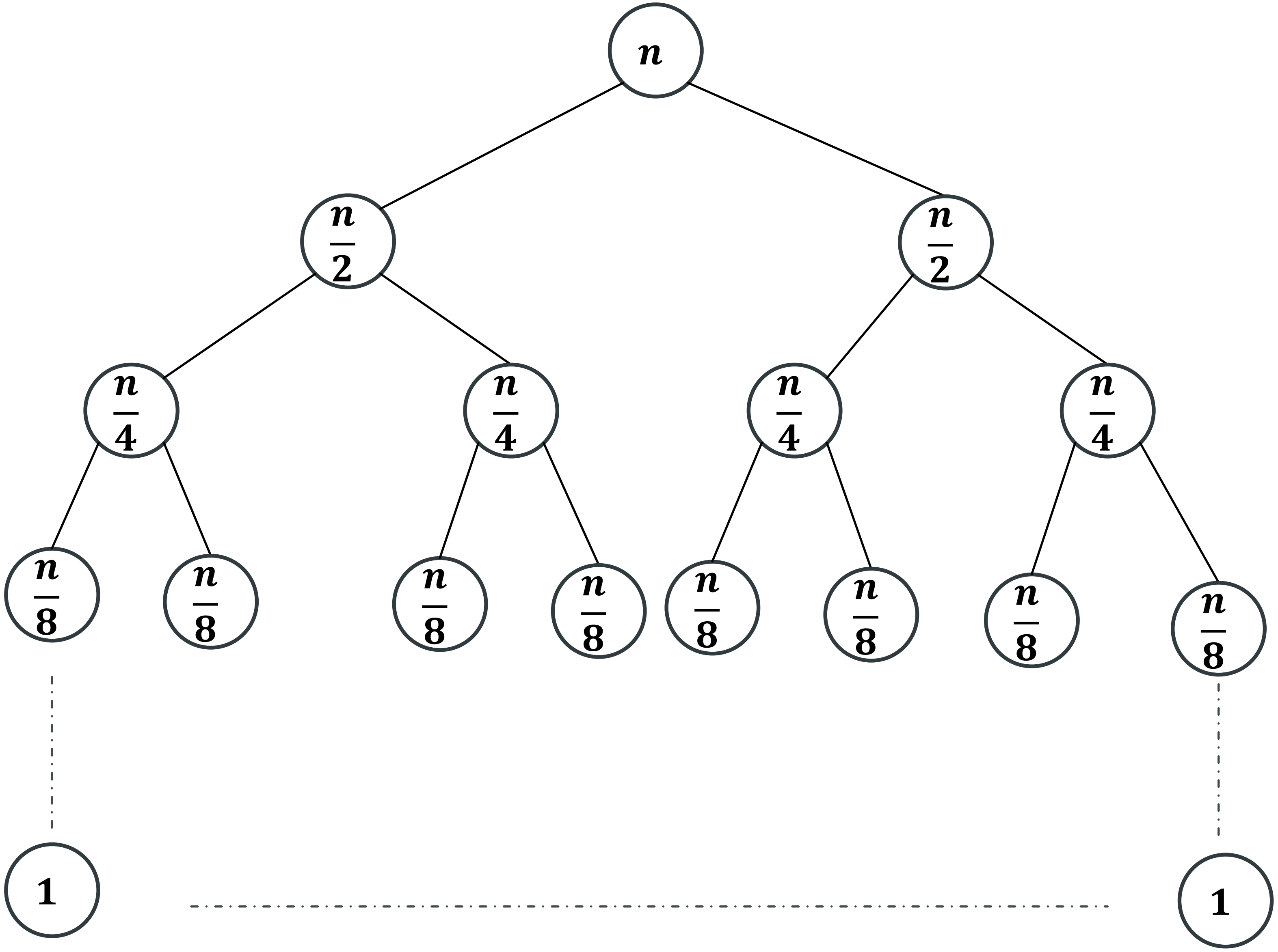
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $\frac{n}{2}$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(logn)$
  - Number of accesses in partition? $O(n)$

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time?

QUICKSORT ANALYSIS



$(n - 1)$

$2\,(\frac{n}{2} - 1)$

$4\,(\frac{n}{4} - 1)$

$8\,(\frac{n}{8} - 1)$

?

0

QUICKSORT ANALYSIS

$$(n \; - 1) + 2\left(\frac{n}{2} - 1\right) + 4\left(\frac{n}{4} - 1\right) + 8\left(\frac{n}{8} - 1\right) + \; \ldots + 0 \; =$$

$$(n \; - 1) + 2^1\left(\frac{n}{2^1} - 1\right) + 2^2\left(\frac{n}{2^2} - 1\right) + 2^3\left(\frac{n}{2^3} - 1\right) + \; \ldots + \; 2^{k-1}\left(\frac{n}{2^{k-1}} - 1\right) + 0 \leq$$

$$\sum_{i=0}^{k-1} 2^i \; \left(\frac{n}{2^i}\right) \leq$$

$$\sum_{i=0}^{\log n} n \leq$$

$$O(n * \log(n))$$

QUICKSORT: WORST CASE

- Assume first element is chosen as pivot.
- Assume we get array that is already in order:

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

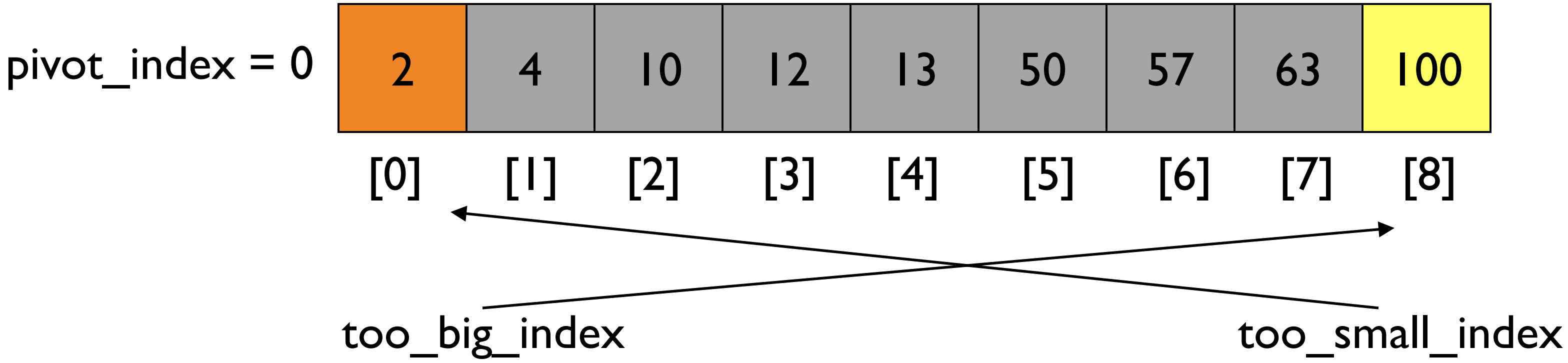
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

too_big_index

too_small_index

EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

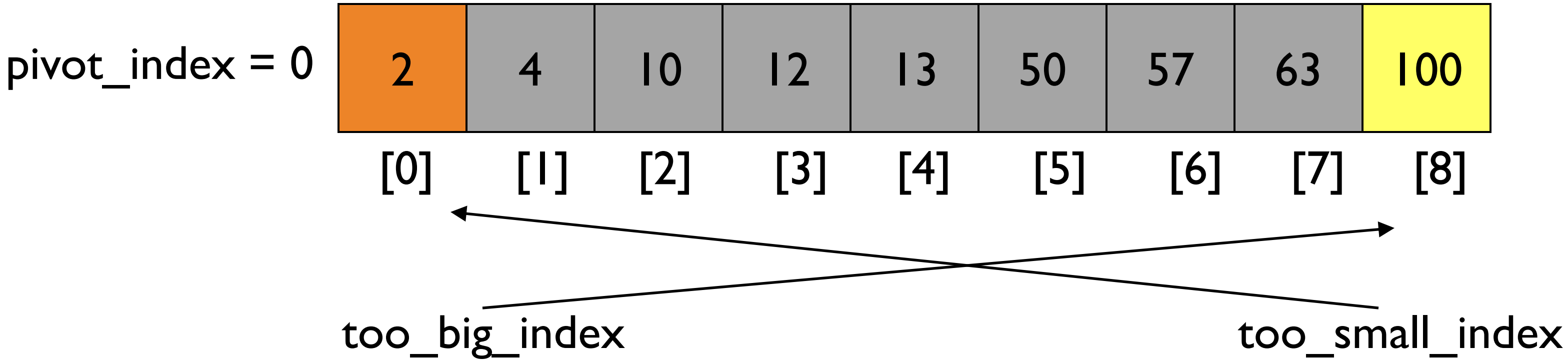
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

too_big_index

too_small_index

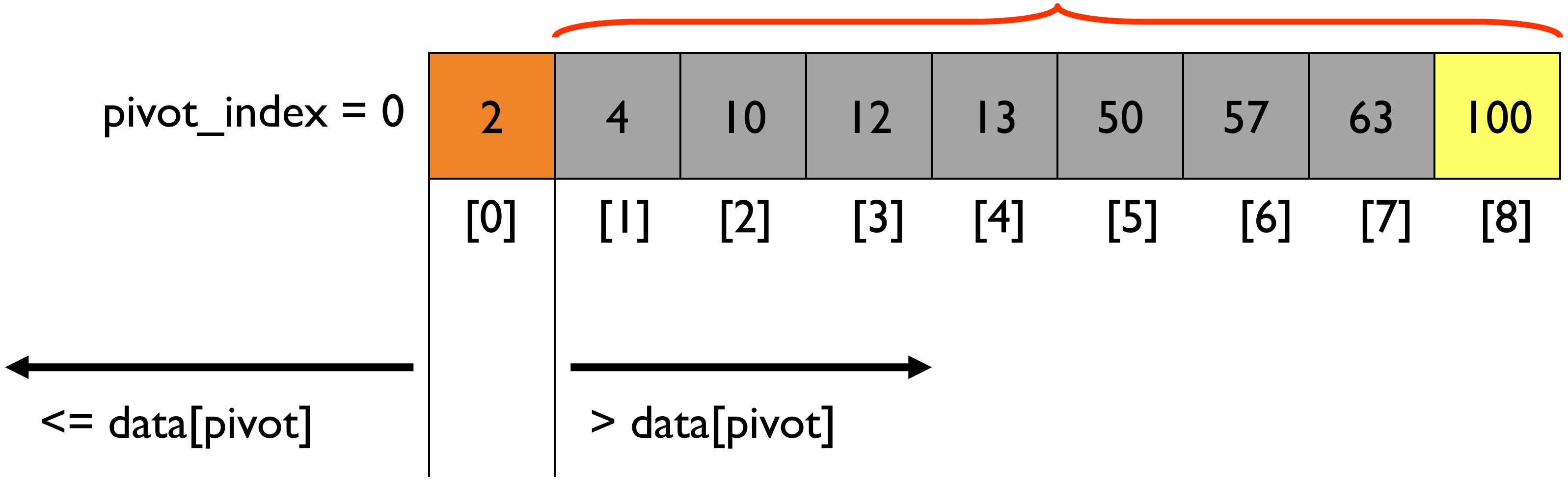
EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



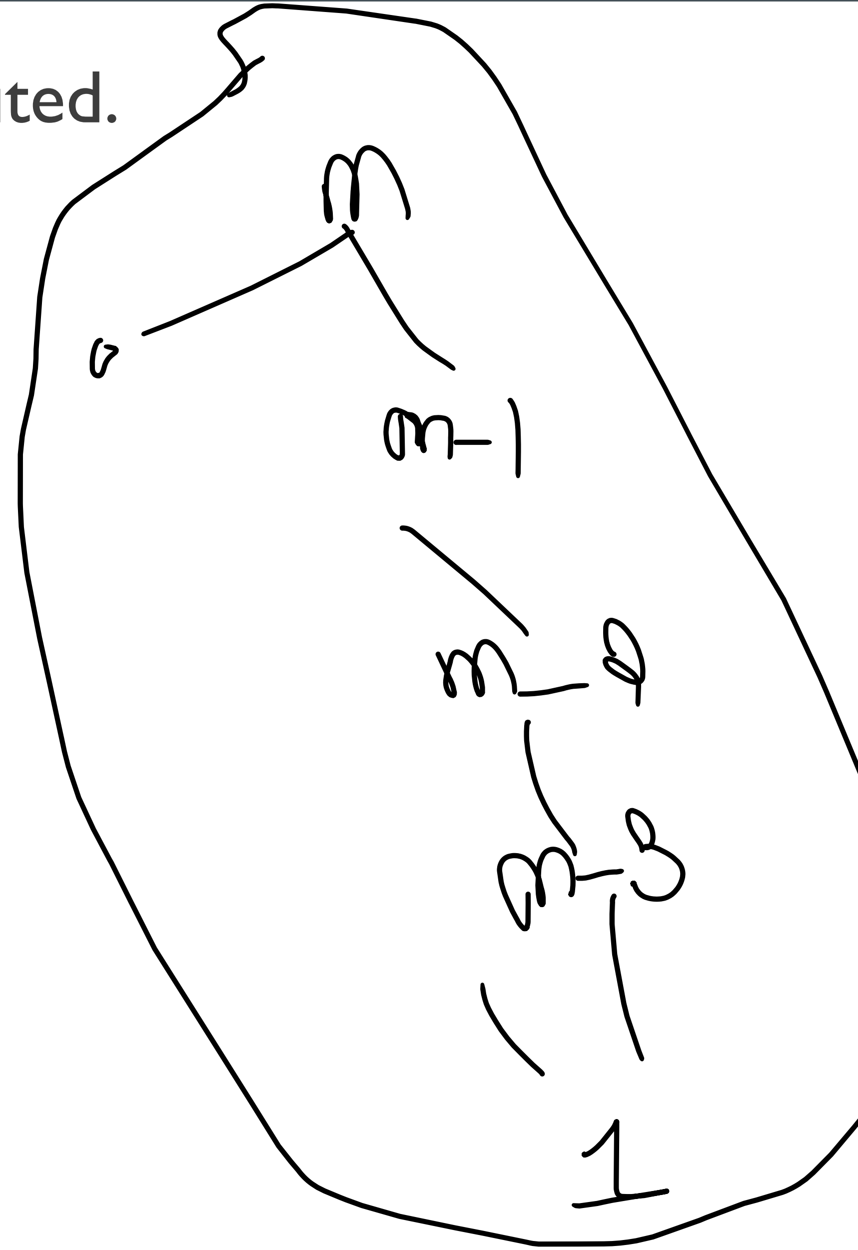
QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\,logn)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition?



- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition? $O(n)$

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time: $O(n^2)$!!!

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n\ logn)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

pivot

10% 90%

$n \log_{10/9} n$

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

$\log_{10/9} n$

n

$\frac{n}{10}$ $\frac{9n}{10}$

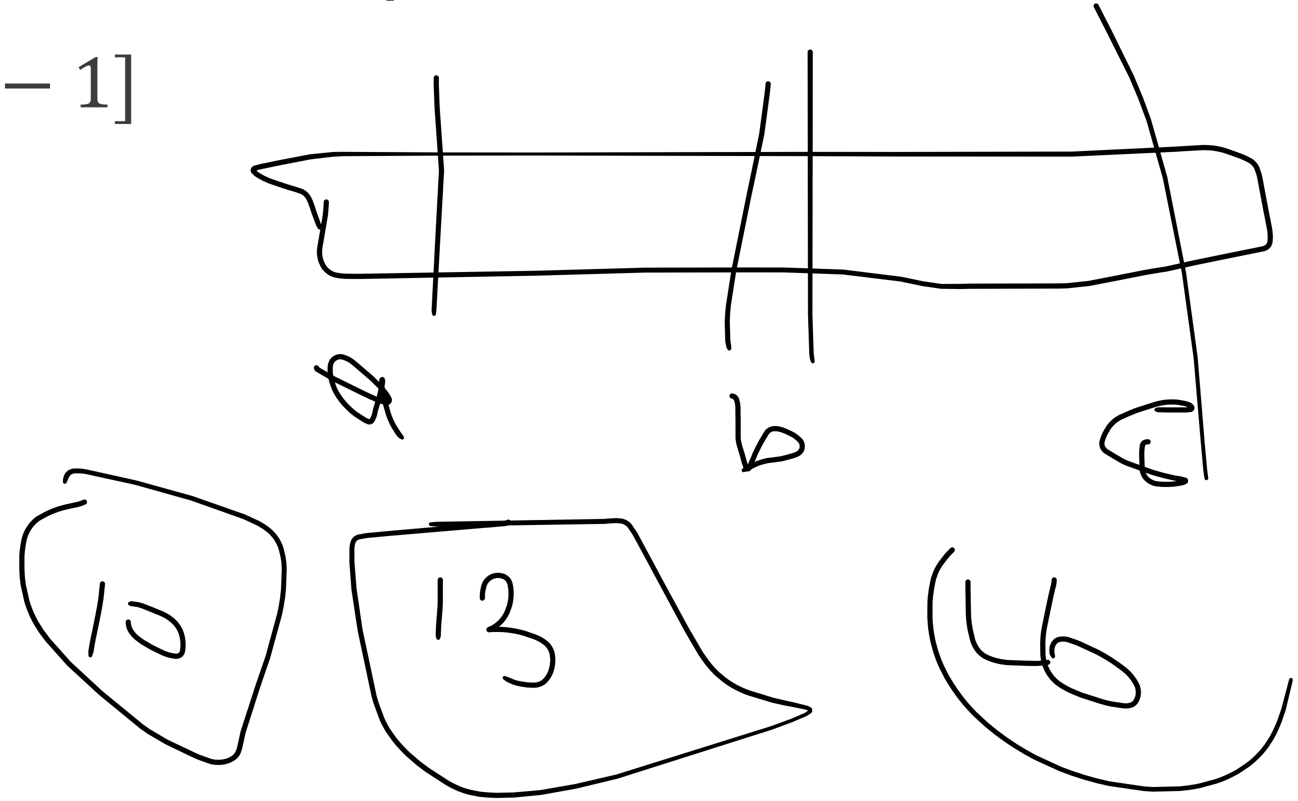
$\frac{n}{100}$ $\frac{9n}{100}$ $\frac{9n}{100}$ $\frac{81n}{100}$

IMPROVED PIVOT SELECTION

Pick median value of three elements from data array:

$$data[0], data[n/2], and\ data[n-1]$$

Use this median value as pivot.



- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 3: left as an exercise.

QUESTIONS/ANSWERS