Sorting

Sorting using Priority Queues
Selection Sort, Insertion Sort, Heap Sort

Why worry about sorting?

Sorting as a task is everywhere:

- *physical*: mail sorting, cards in your hand during card games, books on the shelf in the library, A&E patients by severity
- *interfaces*: searching for books on Amazon by cost, holidays by cost, financial results in a spreadsheet
- *software*: Google sorts query answers by likelihood that they answer our queries, in data structures for ease of searching, for assigning jobs to processors, ...

Sorting is a fundamental technique, which you must understand in order to write efficient software.
It also illustrates fundamental problems and solution techniques in complexity analysis.

Bubble Sort

```
def bubble_sort(mylist):
    n = len(mylist)
    for i in range(n-1):
        for j in range(0,n-i-1):
            if mylist[j] > mylist[j+1]:
                mylist[j],mylist[j+1] = mylist[j+1], mylist[j]
```

Worst case comparisons: (n-1) + (n-2) + ... + 1 = 0.5*(n-1)*n which is $O(n^2)$

Worst case swaps: initial list ordered in reverse, requires a swap for each comparison, so $O(n^2)$

Sorting using Priority Queues

A Priority Queue is a data structure to which we can add items, and from which we can remove the item with the top priority.

Outline sorting algorithm: for each item in our list, add it to the PQ. Then repeatedly remove the top item from the PQ and put in successive cells in the list.

```
def pq_sort(mylist):
    pq = PriorityQueue()
    for i in range(len(mylist)):
        pq.add(mylist[i],None)
    for i in range(len(mylist)):
        mylist[i],x = pq.remove_min()
```

Efficiency will depend on the implementation of the PQ

Sorting with an *unsorted* linked list PQ

Selection sort: main task is selecting the next smallest item

Adding an item to the PQ is O(1), so cost of first loop is O(n)

Removing top item requires linear search over all items in the list, so n-1 comparisons.
Each time we remove an item, the list shrinks by 1, and we make 1 assignment back into our original (array-based) list

So removing n items takes (n-1) + (n-2) + ... + 1 comparisons.
O(n^2) comparisons, and n assignments.

Best case?

O(n^2) comparisons, and n assignments

In-place sorting

Sorting using a separate Priority Queue requires extra space.

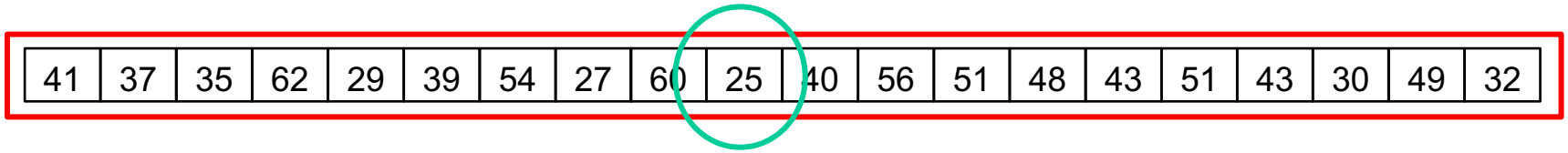
An input list of size *n* will require a PQ implementation of size *n* as it operates.

In many applications, space in memory is restricted, and so we want to sort using only a small amount of extra space over the original input array.

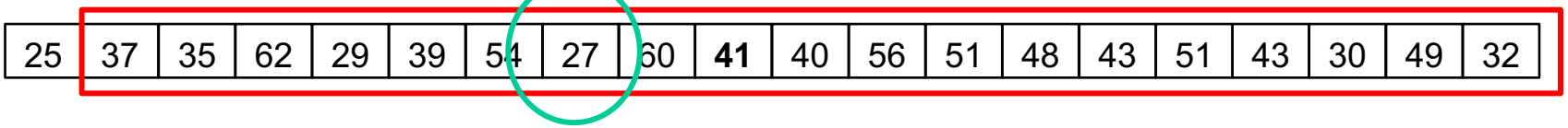
- this is called *in-place* sorting

In-place Selection Sort

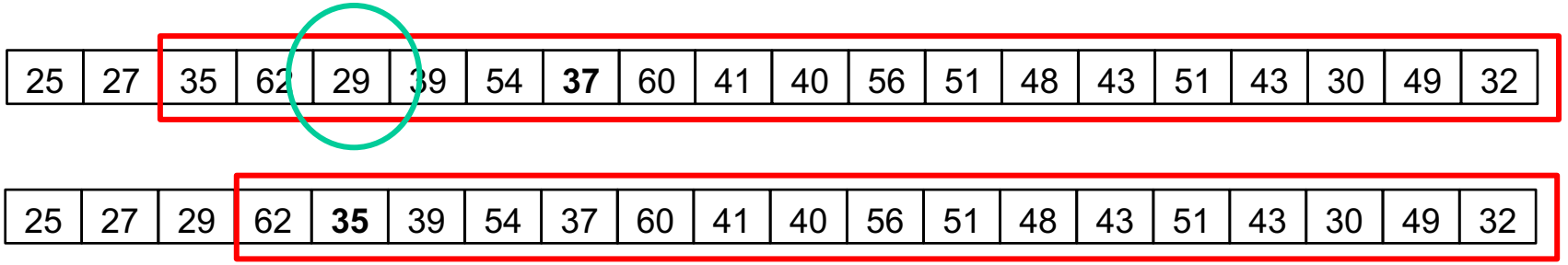
Treat the unsorted input array as the PQ list implementation (so no build cost).
Instead of removing the top item, swap it into the correct cell, and shrink the 'view' of the PQ



Find the smallest item, swap it with cell 0.



Find the smallest item in the remainder, swap it with cell 1, etc.



```
def selection_sort(mylist):
    n = len(mylist)
    i = 0
    while i < n:
        smallest = i
        j = i+1
        while j < n:
            if mylist[j] < mylist[smallest]:
                smallest = j
            j += 1
        mylist[i], mylist[smallest] = mylist[smallest], mylist[i]
        i += 1
```

Note: this Python code is using as few of the helpful Python features as possible, to make it easier to understand the complexity

Worst case: $O(n^2)$ comparisons, and n swaps.
Best case: $O(n^2)$ comparisons, and n swaps.

Sorting with an *sorted* linked list PQ

Insertion sort: main task is inserting each item in right place

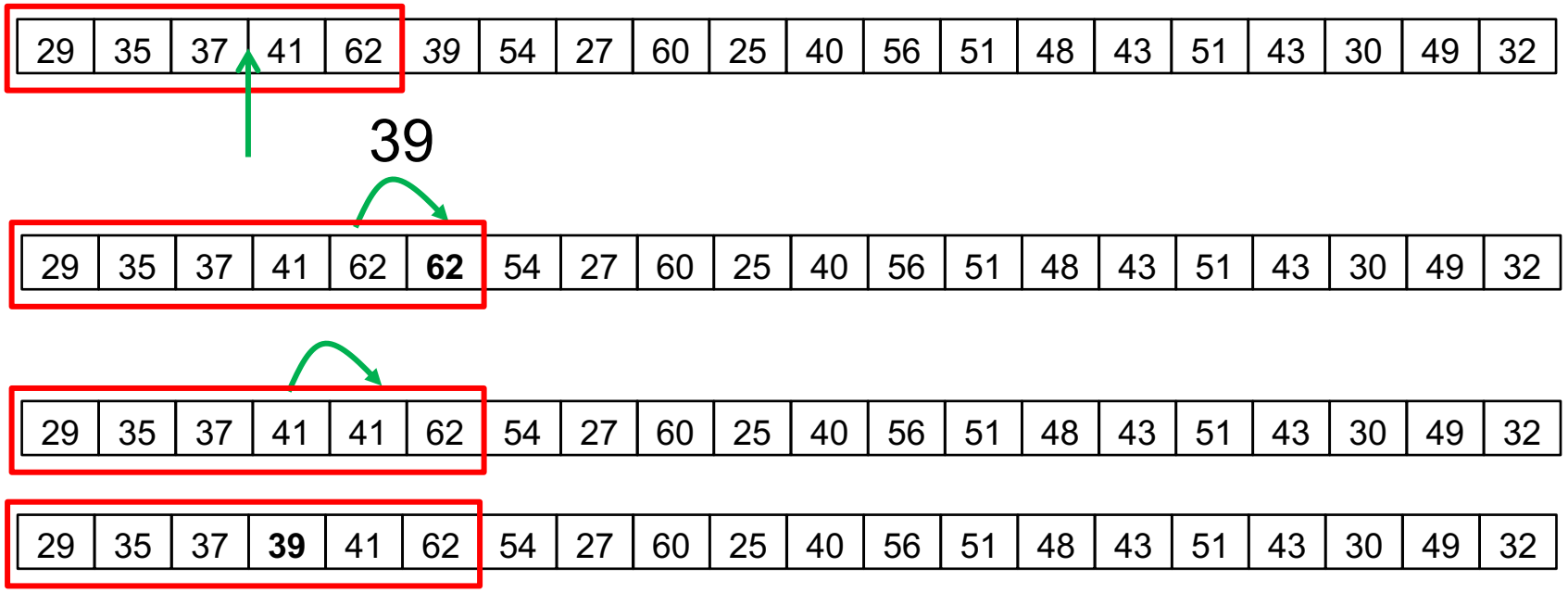
Adding an item to sorted linked list of length *n* takes at most *n* comparisons, and if we are lucky, just 1 comparison. It takes 1 assignment (since this is a *linked list)*.
So adding *n* items into an initially empty PQ takes $0+1+2+...+(n-1) = 0.5*(n-1)*n = O(n^2)$ comparisons worst case (and $O(n^2)$ assignments).

Removing top item is O(1)

Best case? O(*n*) comparisons, and *n* assignments

In-place Insertion Sort

Treat the unsorted input array as the stream of items to be added to the PQ list implementation, and gradually expand the sorted list from the front (i.e. growing the 'view' of the PQ).
Search the PQ to find the insertion place.
Copy the new item, shuffle the others down one place, insert the new item



```
def insertion_sort(mylist):
    n = len(mylist)
    i = 1
    while i < n:
        j = i-1
        while mylist[i] < mylist[j] and j > -1:
            j -= 1
        #insert i in the cell after j
        temp = mylist[i]
        k = i-1
        while k > j:
            mylist[k+1] = mylist[k]
            k -= 1
        mylist[k+1] = temp
        i += 1
```

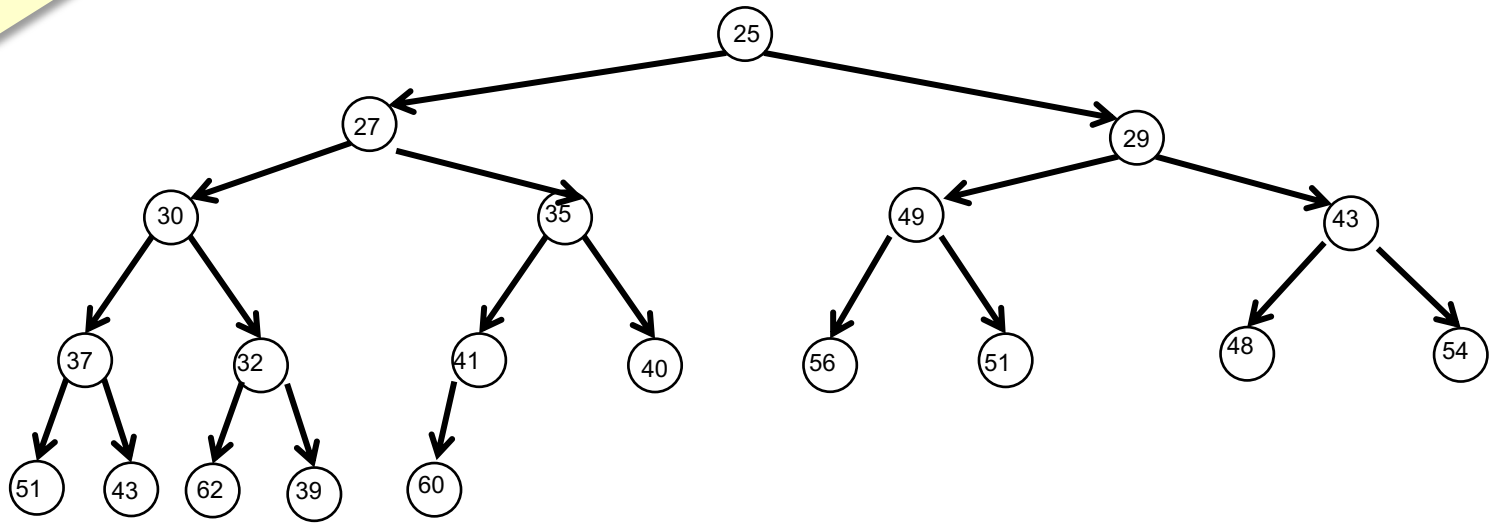
Worst case is a completely reverse sorted list: O(n^2) comparisons, and O(n^2) swaps.
Best case is a completely sorted list: O(n) comparisons, and *0* swaps.

What other options do we have for implementing the PQ?

Could we use any of those for an in-place sort?

CS2515

The Binary Heap



Complete tree, each node has lower value than its children.
Add a value in last place, then bubble up to restore property.
Remove top item, copy last into place, bubble down, always choosing the lowest value child to swap with.

Complexity: O(log n) to add and remove, O(1) to find

Sorting with a *binary heap* PQ

Heap sort: all the action takes place with heap operations

Adding an item to a binary heap which has *n* items takes O(*log n*) comparisons and swaps. So adding all *n* items takes log(2) + log(3) + ... + log(n-1).
Each of these $\leq$ log(n), and there are $\leq$ *n*, so
*n*O(log n*) = O(*n log n)* to build the PQ.

Removing the top item from a binary heap with *n* items takes O(*log n*), so by the same argument, to rebuild the array takes *O(n log n)* comparisons and swaps.

So O(*n log n*) to sort using a binary heap.

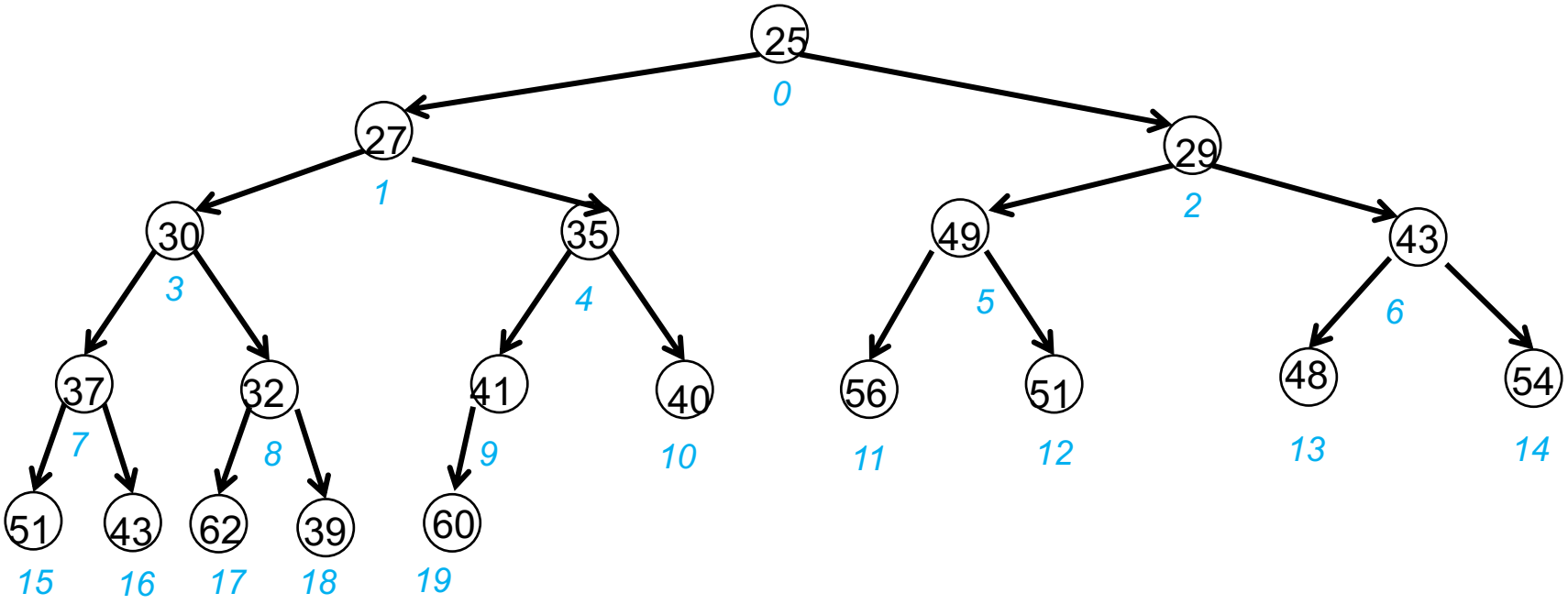
O*(n log n*) is a lot better than O(n^2)

Represent the tree using a list

CS2515

...node is at index 0.
...ext item to be added is at index *size*.
Last item is at index *size-1*

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



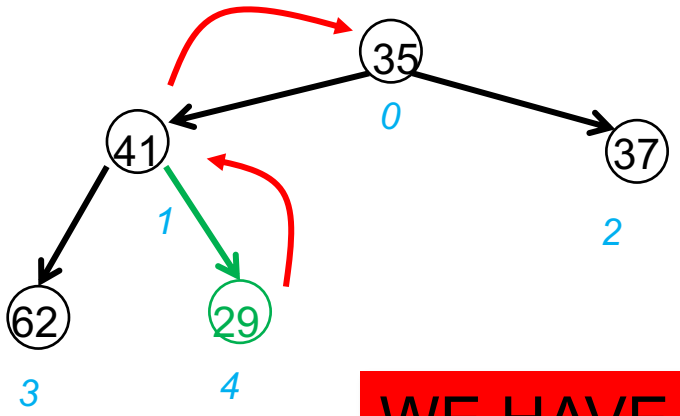
25	27	29	30	35	49	43	37	32	41	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

In-place HeapSort ?

Treat the unsorted input array as the stream of items to be added to the PQ, like insertion sort, and gradually expand an *array-based heap* from the front (growing the 'view' of the PQ).

35	41	37	62	*29*	39	54	27	60	25	40	56	51	48	43	51	43	30	49	32

Next item added to heap goes into last position, so it is already in the right starting place. Now bubble up to find its correct position.



29	35	37	62	41	*39*	54	27	60	25	40	56	51	48	43	51

WE HAVE A PROBLEM ...

Using a *max* heap

The end point of that process would be a complete array-based representation of the heap.

But the heap is *not* a sorted list.

Generating a sorted list from a min binary heap, in-place, requires a lot of searching.

Instead, we will do two phases, and in the first phase, we will build a *max* binary heap.

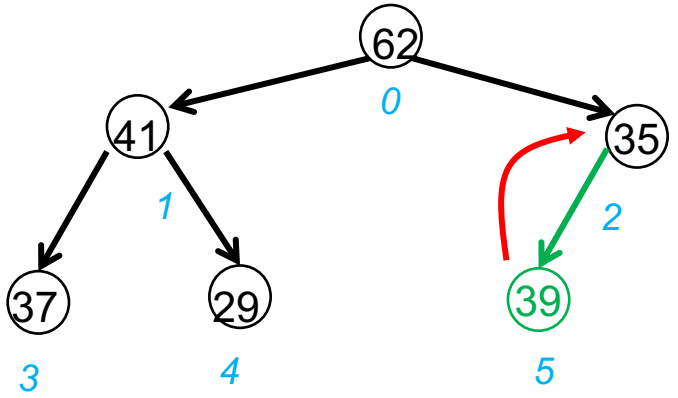
- same as before, but now each item must be larger than or equal to its children

In-place HeapSort , phase 1

Treat the unsorted input array as the stream of items to be added to the PQ, like insertion sort, and gradually expand an array-based *max* heap from the front

62	41	35	37	29	*39*	54	27	60	25	40	56	51	48	43	51	43	30	49	32

Next item added to heap goes into last position, so it is already in the right starting place. Now bubble up to find its correct position.

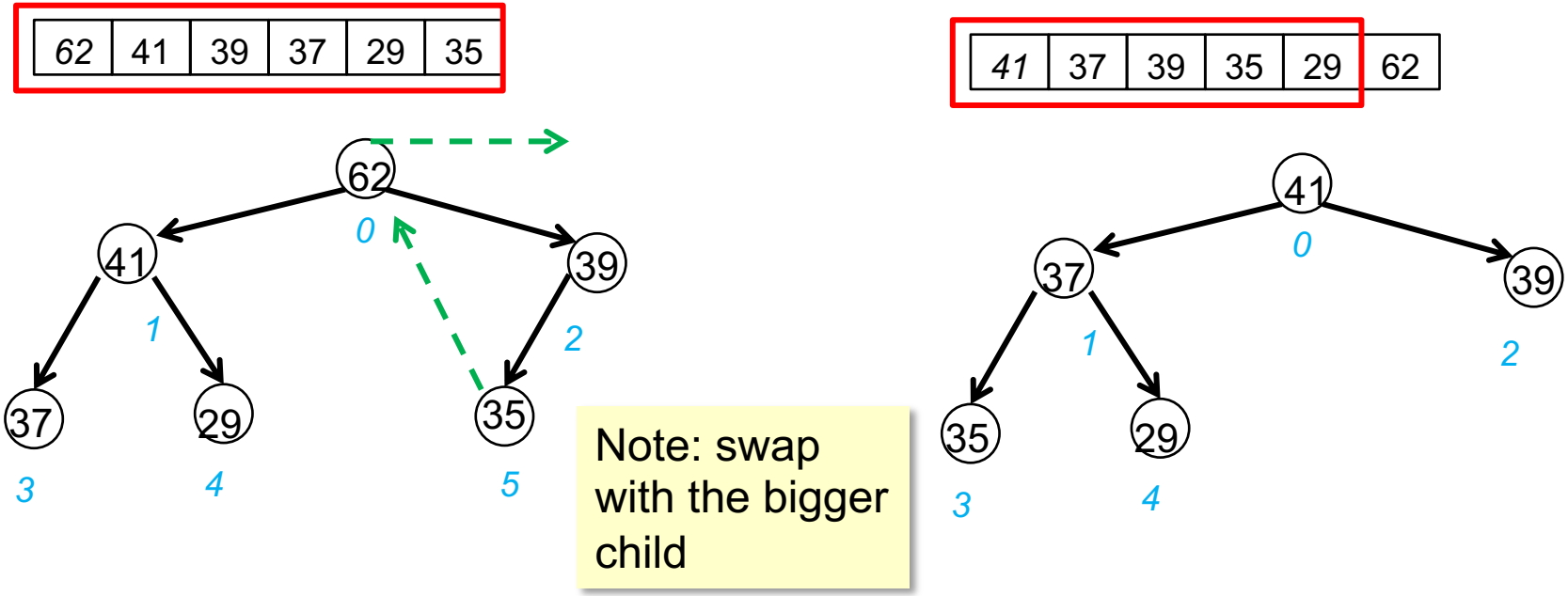


62	41	39	37	29	35	*54*	27	60	25	40	56	51	48	43	51	43	30	49	32

In-place Heap Sort, phase 2

At the end of phase 1, we have an array based max heap, with the biggest item at the front.

In the final output, we want the biggest item at the end. So remove it from the heap, copy the last item up then bubble it down, leaving the last place free to re-insert the biggest item.
Iterate until the (virtual) heap is empty, and we have a sorted list.



Heap Sort complexity

The same analysis as for the heap sort with separate PQ works again.

O(*n log n*) comparisons and swaps to build the max heap in the array.
O(*n log n*) comparisons and swaps to turn the max heap into the sorted list.

Note: if we start with a complete unsorted array, the bound on building the array-based heap can be reduced to *O(n)*, but overall complexity remains *O(n log n)* as we still have to turn it into a sorted list

Next Lecture

More sorting