Reminders and Topics

- This lecture:
  - **Merge Sort**
  - **Heap Sort**
  - **Quick Sort**

iClicker Question #0

- If you convert Ada's birthday from MMDDYY in binary (they are all 0s and 1s) and you get 22 decimal, how old is she?

(a) 6

(b) 7

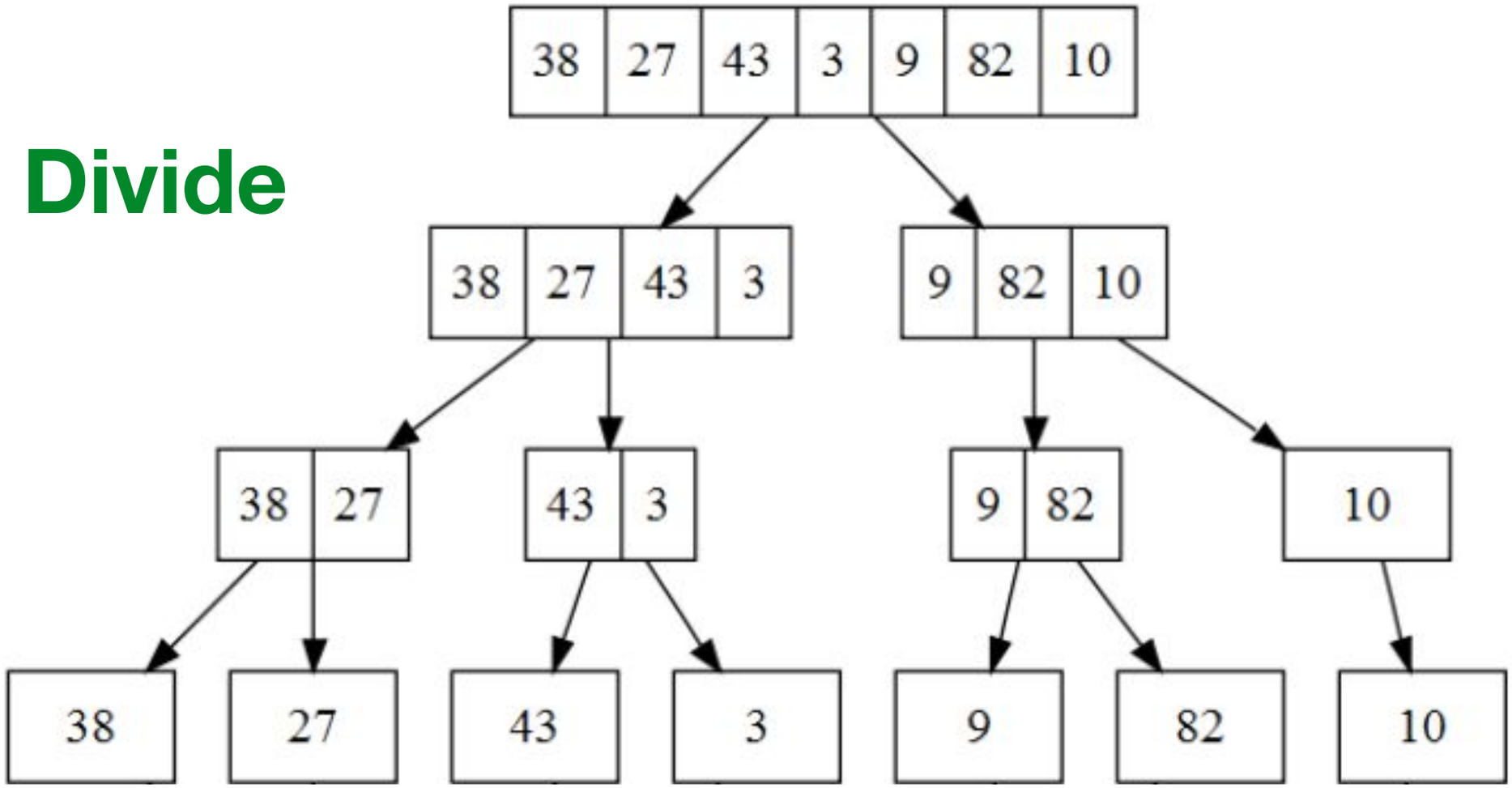
(c) 8

(d) 9

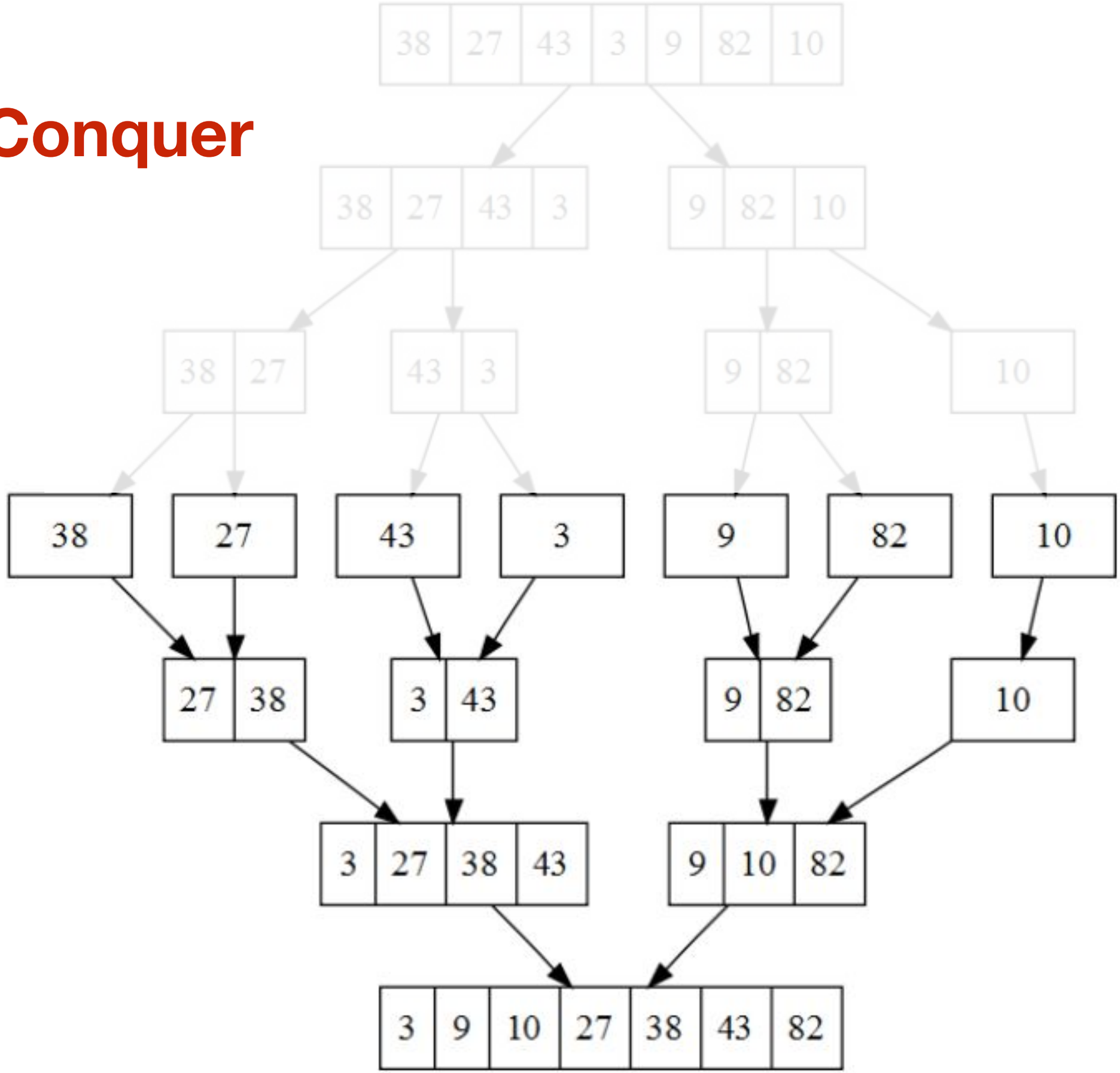
(e) 10

Answer on next slide

Divide



Conquer


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

- Merge Sort can also be implemented iteratively:
  - Merge every two adjacent 1-element blocks
  - Merge every two adjacent 2-element blocks
  - Merge every two adjacent 4-element blocks
  - Merge every two adjacent 8-element blocks
  - …..
  - Merge the first half and second half
  - You are done!

```
// temp: scratch buffer to stored merged elements
// k: the length of subarrays we are merging
public void mergeSort() {
  T[] temp = (T[])new Object[nElems];
  for(int k = 1; k < nElems; k = k*2) {
    for(int i = 0; i < nElems; i += (2*k)) {
      merge(a, temp, i, i+k, i+k+k-1);
    }
  }
}
/* merge(a, temp, startA, startB, endB) will merge
the two sub-arrays defined by a[startA, startB-1]
and a[startB, endB] to the output temp[startA, endB]
*/
```

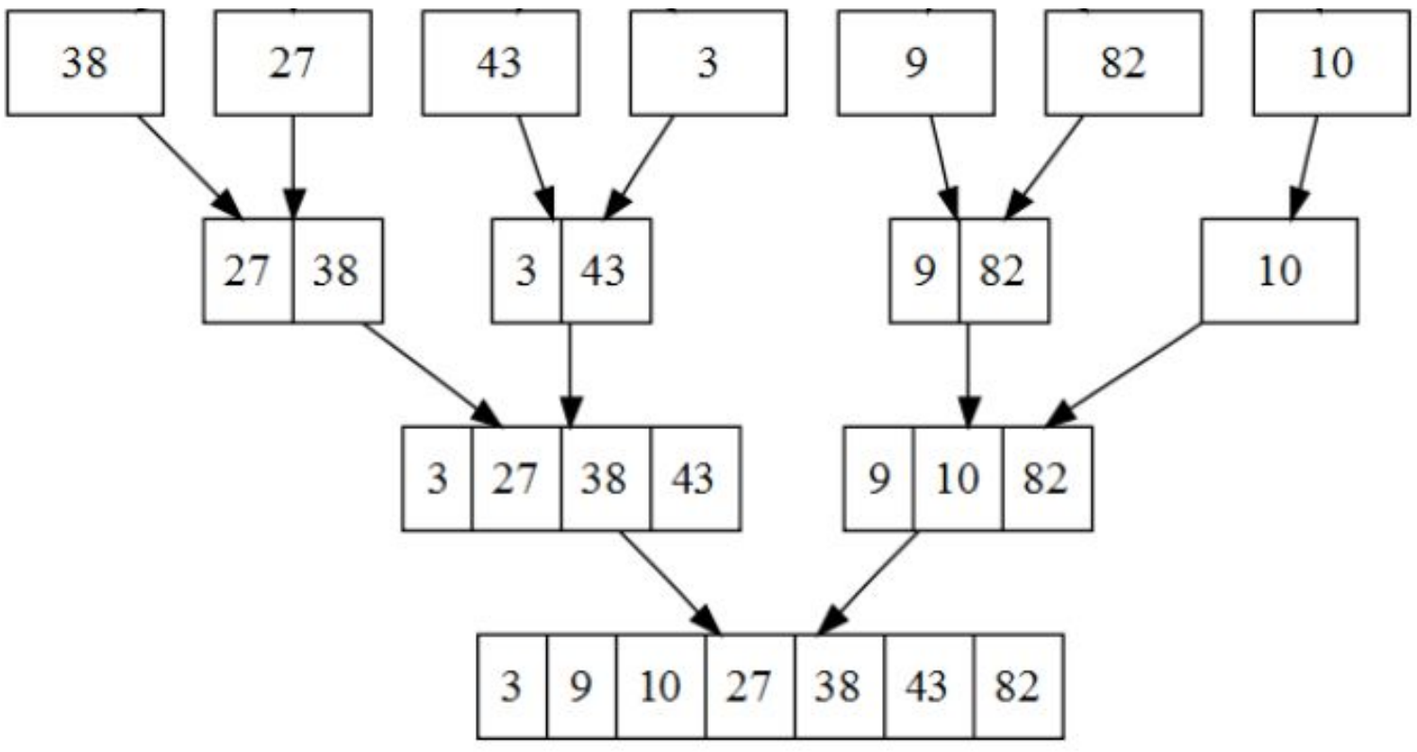
Merge Sort Cost Analysis

- The input array has N elements (just assume N is a power of 2). What's the cost of Merge Sort?
  - How much work (i.e. comparisons + copies) is involved in each round, in terms of N?
  - How many rounds are there?

Round 1

Round 2

Round …



Merge Sort Cost Analysis

- The input array has N elements (just assume N is a power of 2). What's the cost of Merge Sort?
  - How much work (i.e. comparisons + copies) is involved in each round, in terms of N?
    - **O(N)**
  - How many rounds are there?
    - **$\log_2$ N**
- Therefore the total cost of Merge Sort is: **O(N logN)**

Heap Sort

- While Merge Sort is easy to implement and fast, it requires **`O(N)`** additional storage (the temp buffer).
- Now let's look at **Heap Sort** and **Quick Sort**, both of which can be done 'in place'.
- First, think about a trivial implementation of Heap Sort:

```
for(i=0; i<nElems; i++) heap.add(array[i]);
for(i=nElems-1; i>=0; i--) array[i] = heap.remove();
```

- Assuming heap is a maxHeap, this sorts elements in order. What's the cost? **O(N logN)**
- Is there additional storage involved here? **Yes, the heap!**

Heap Sort

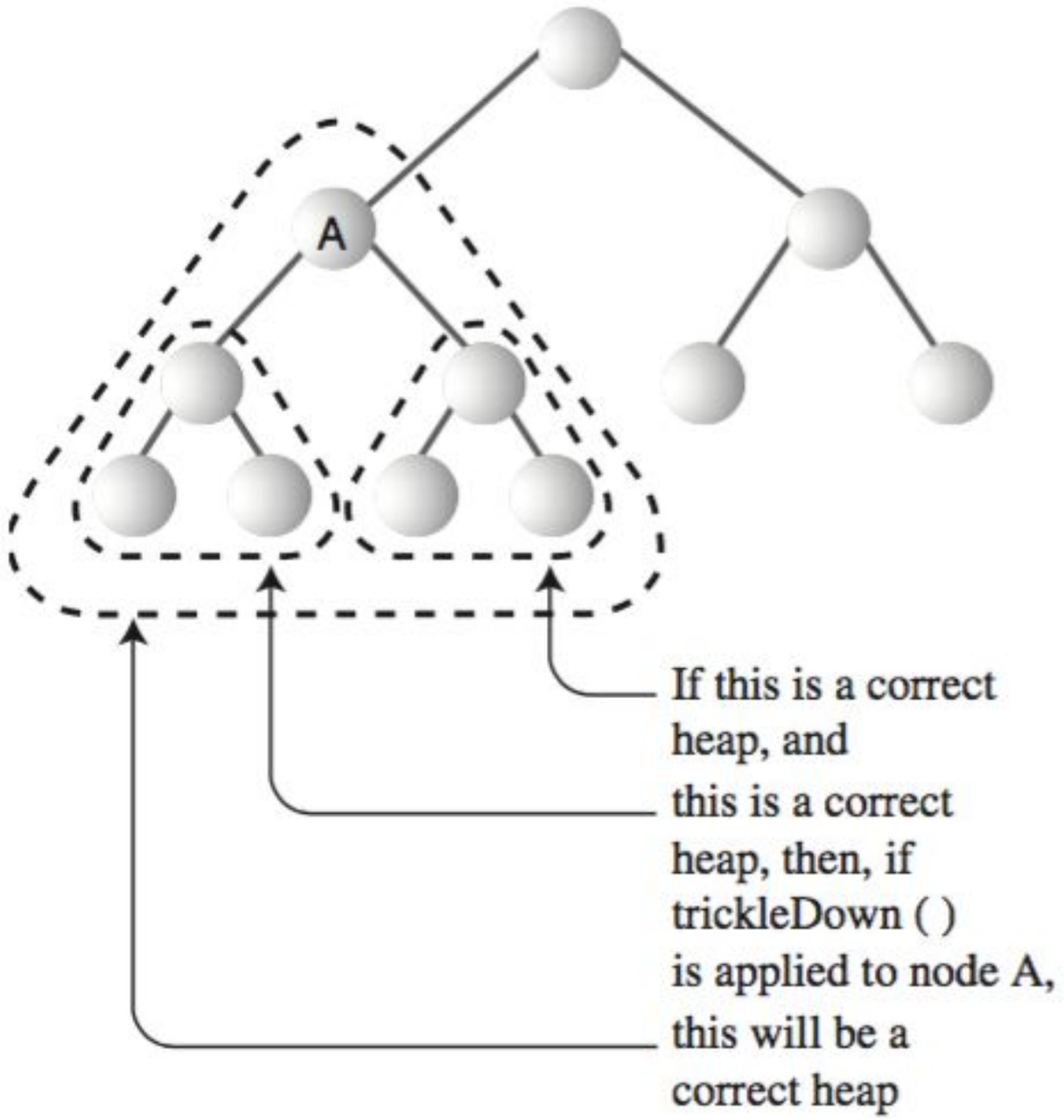
- To eliminate the additional storage, recall that a heap is typically stored in an array. Since we already have an input array, the same array will be used as a heap. How?

1. Convert the input array to a heap **in place** (`heapify`)
2. Repeatedly remove the root element from the heap and move it towards the end of the array.

- This is conceptually similar to Selection Sort, but each round costs only `O(log N)` due to heap operations.

Heapify

- How to convert an unsorted array to a heap in place?
- Imagine node A's left subtree and right subtree are both valid heaps, except A may be out of order. We can apply a **`bubbleDown`** on A to fix it and transform it to a correct heap.
- Apply this repeatedly from the end of the heap, and work our way towards the root. This makes sure that at every element the subtrees below it are already correct heaps.



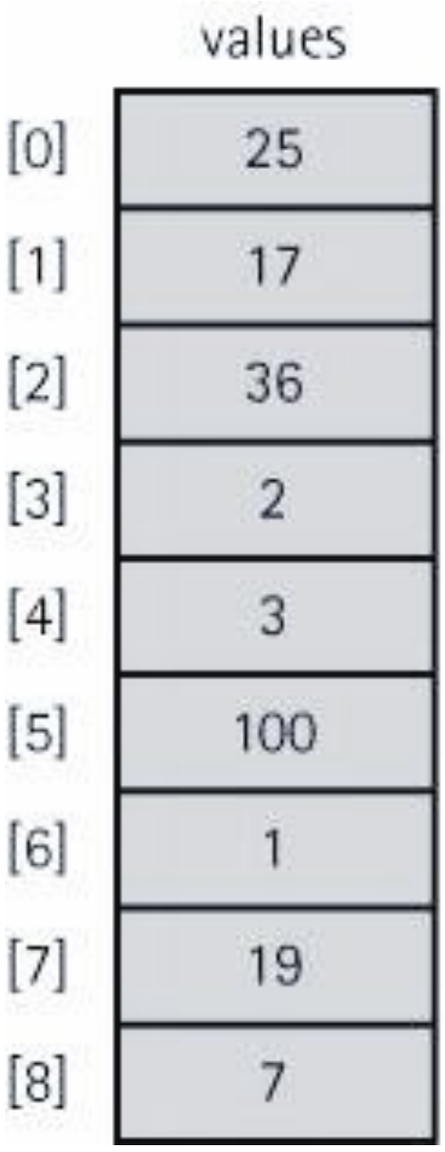
Heapify

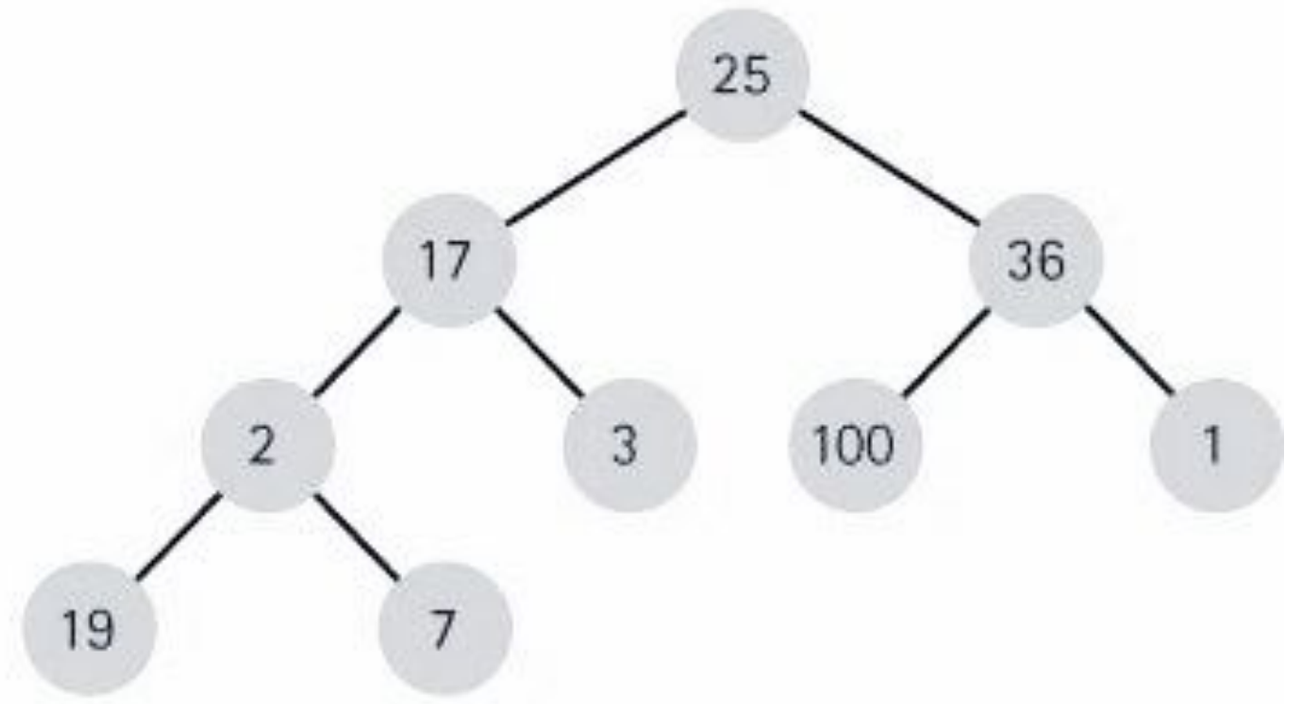
- One optimization: each leaf node is already a correct heap, because it does not have any child!
- Therefore the heapify process only needs to start from the **last interior node** (i.e. one before the first leaf).

```
public void heapify() {
   for(int i=last_interior;i>=0;i--) {
     bubbleDown(i, nElems-1);
   }
}
/* the second parameter in bubbleDown
   indicates index of the last element.
   Later will see why it's useful. */
```

Heapify Example

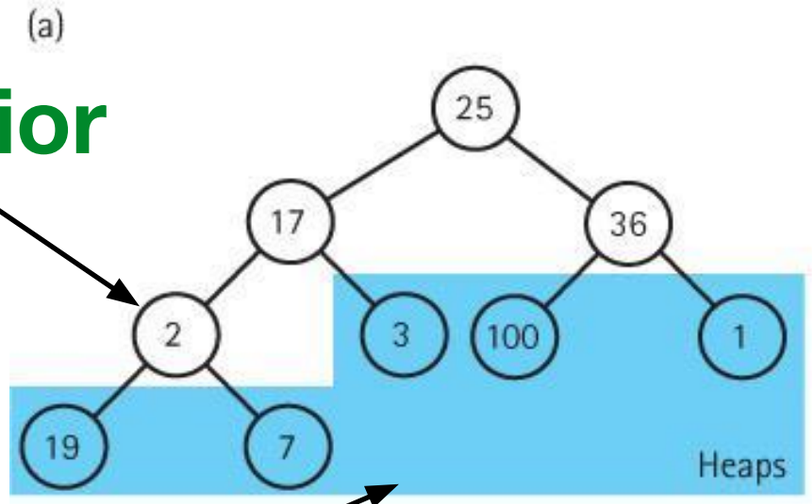
Input Array and how it looks like as a complete binary tree) — this is not a valid heap yet





Heapify Example

Last interior node



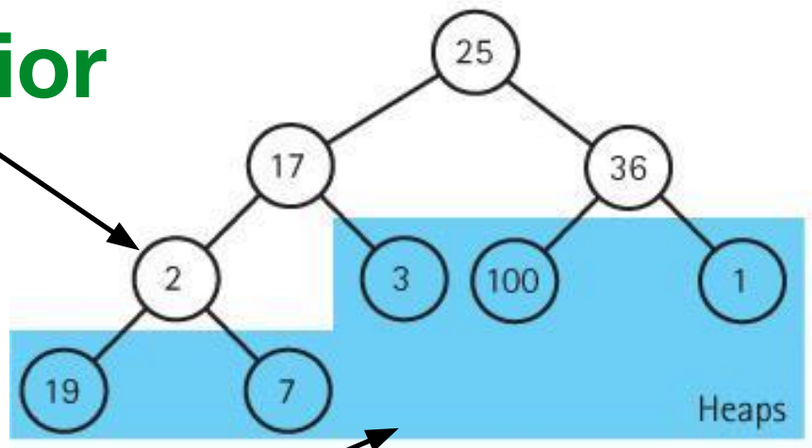
Leaf nodes

Heapify Example

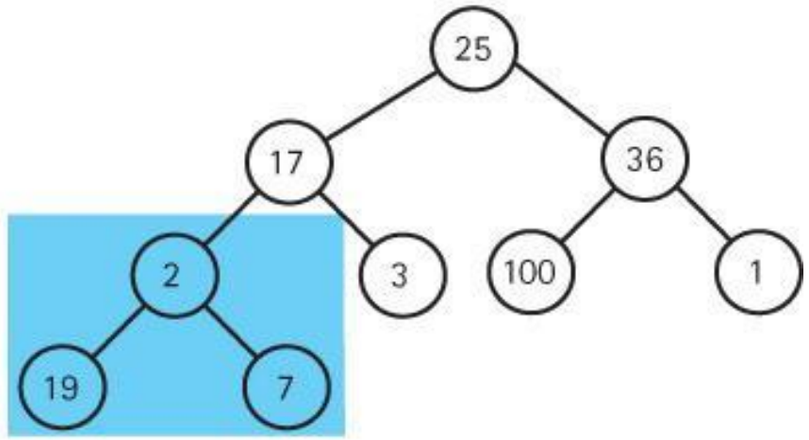
Last interior node

Leaf nodes

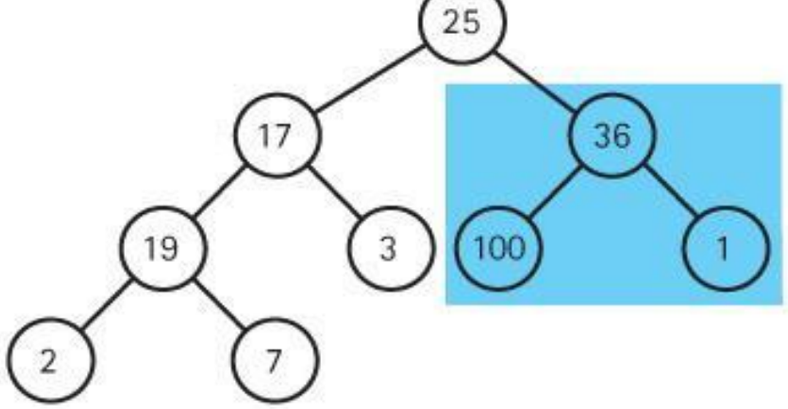
(a)



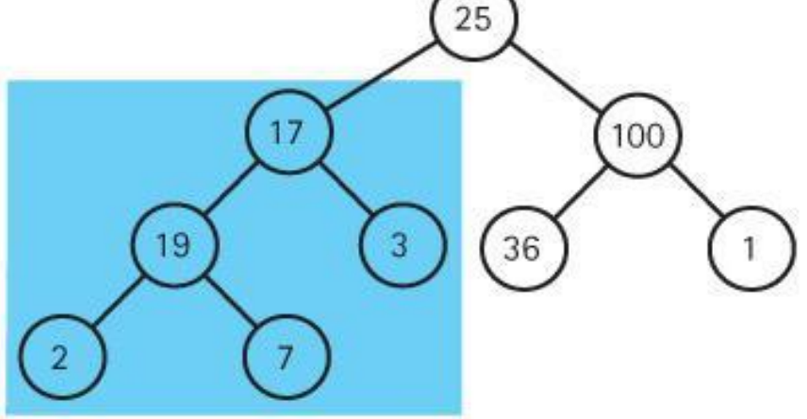
(b)



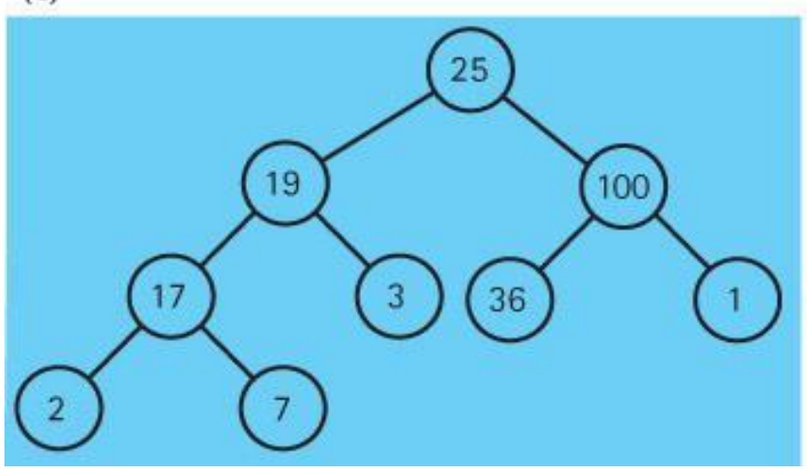
(c)



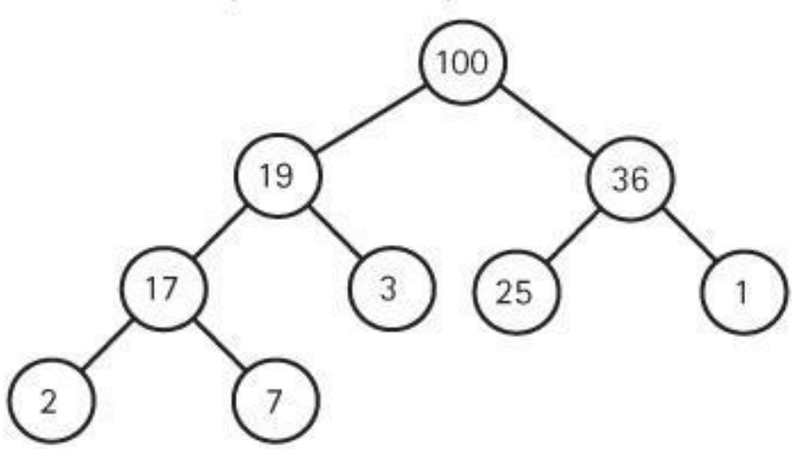
(d)



(e)



(f) Tree now represents a heap



iClicker Question #1

- If an array contains N elements, and we think of it as a heap (i.e. a complete binary tree), what's the index of the **last <u>interior</u> (non-leaf) node**?

(a) N/2

(b) N/2+1

(c) N/2-1

(d) N-1

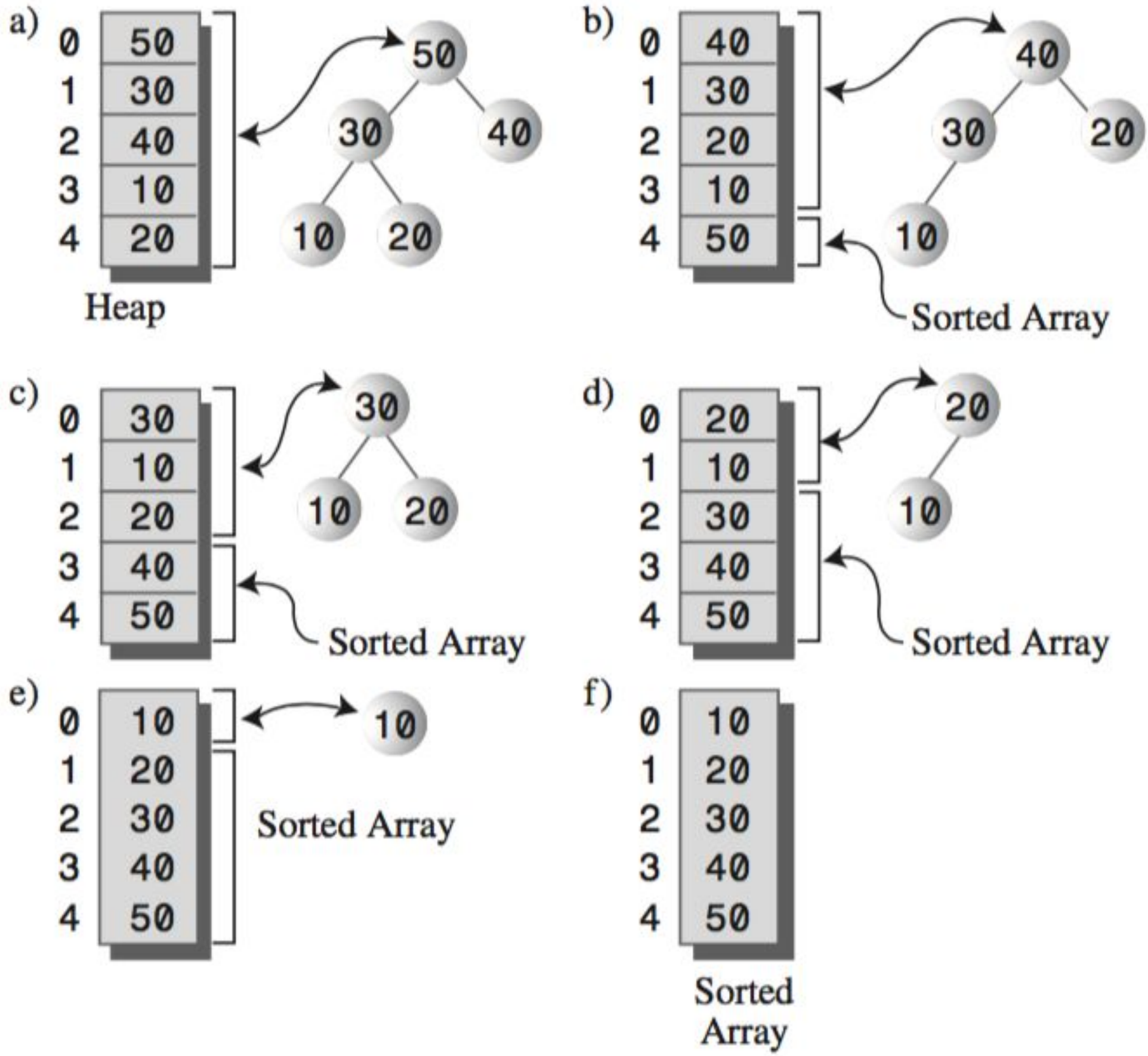
(e) N-2

Answer on next slide

Heap Sort

- Once the heap is constructed, the next step is to repeatedly remove the root element, and move it towards the end of the array.
- This is done in place, and think of the array as partitioned into a heap region and a sorted region.
- As we keep removing elements, the heap region shrinks, while the sorted region grows.
- Remember how to remove an element from the heap?

    1. Remove the element at index 0 (root)
    2. Move the last element in the heap to index 0
    3. Call `bubbleDown(0)`


a)
0 50
1 30
2 40
3 10
4 20
50
30 40
10 20
Heap
b)
0 40
1 30
2 20
3 10
4 50
40
30 20
10
Sorted Array
c)
0 30
1 10
2 20
3 40
4 50
30
10 20
Sorted Array
d)
0 20
1 10
2 30
3 40
4 50
20
10
Sorted Array
e)
0 10
1 20
2 30
3 40
4 50
10
Sorted Array
f)
0 10
1 20
2 30
3 40
4 50
Sorted Array

Heap Sort

- We can simply swap the last element of the heap (recall the heap shrinks as you work) with the root (index=0)
- Putting everything together:

```
public void heapSort() {
   for(i=nElems/2-1; i>=0; i--) { // heapify
     bubbleDown(i, nElems-1);
   }
   for(i=nElems-1;  i>=1; i--) {
     swap(0, i);
     bubbleDown(0, i-1);
   }
}
```

↘ Last element index of the current heap

Heap Sort Summary

- Sorts an array ‘in place’ without additional storage.
- Heapify step is N/2 bubbleDown operations (which are log(N)
- Second step is N bubbleDown operations, also log(N)
- So the total is O(N log N): called **log-linear** cost

(Note: This analysis is not “tight”. The heapify is actually O(N), something we will leave for later classes)

Quick Sort

- Like Merge Sort, Quick Sort uses divide and conquer
- It sorts in place, hence does not require additional buffer.

1. **A key step** in Quick Sort is **Partition** (a.k.a. Split)
    - Given an input array, find a **pivot** element, then partition the array into two groups: all elements smaller than the pivot are placed on its left, all elements larger than the pivot are placed on its right.
    - This implies that the **pivot element is in its final position**.
2. Recursion on the left group and right group.

Partition

- Example: `8 5 3 7 1 9 6`
- Say the pivot element is `6`. How do you write an algorithm to partition the array, such that elements smaller than 6 are placed on the left of it, and those bigger than 6 are placed on the right?
- If you are allowed to use additional buffer, it's easy.
- What if this must be done in place?

Partition

- Idea: keep track of where the next element smaller than pivot should be stored at (call it `storeIndex`).
- Scan through the array, if an element is smaller (or equal to) the pivot element, swap it with `a[storedIndex]`.
- How do we pick the pivot element?
  - It can be any element in the array. For example, the first element, the last element, or the element in the middle.
  - Without loss of generality, we pick the last element as the pivot. (but you can as well pick the first, or the middle etc.)

```
// assume the array being considered for partition
// is in the range [low, high], inclusive on both ends

public int partition(int low, int high) {
  T pivot = a[high]; // pick the last element as pivot
  int storeIndex = low;
  int j;
  for(j=low; j<=high-1; j++) {
    if(a[j].compareTo(pivot) <= 0) {
      swap(j, storeIndex);
      storeIndex++;
    }
  }
  swap(storeIndex, high);
  return storeIndex;
}
```

Return the position of the pivot element after partition.

Partition Example

- Input: [8 5 3 7 1 9 6]
- Pivot: 6 (the last element)
- storeIndex is 0 (points to element ‘8’) to begin with.
- Walk through this example on the board
- The return value (storeIndex) marks where the pivot element is stored at.

iClicker Question #2

- The input array is `[3 9 7 5 8 2 6]`. What's the result of partitioning this array? (As before, we choose the last element as the pivot element)

(a) [3 5 2 6 9 7 8]

(b) [5 3 2 9 8 7 6]

(c) [3 5 2 6 7 8 9]

(d) [3 5 2 6 8 7 9]

(e) [5 3 2 6 8 7 9]

Answer on next slide

Another One

iClicker Question #3

- What's the cost of partition on a size N array?

(a) O(1)

(b) O(log N)

(c) O(N)

(d) O(N log N)

(e) $O(N^2)$

```
public int partition(int low,
                     int high){
  T pivot = a[high];
  int storeIndex = low;
  int j;
  for(j=low; j<=high-1; j++) {
    if(a[j].compareTo(pivot) <= 0)
    {
      swap(j, storeIndex);
      storeIndex++;
    }
  }
  swap(storeIndex, high);
  return storeIndex;
}
```

Answer on next slide

Quick Sort

- With `partition()`, Quick Sort is very easy to implement:

```
protected void recQuickSort(int low, int high) {
  if(low < high) {
    int p = partition(low, high); // p is split point
    recQuickSort(low, p-1);
    recQuickSort(p+1, high);
  }
}

public void quickSort() {
  recQuickSort(0, nElems-1);
}
```

Quick Sort Analysis

- Note that Partition does not at all guarantee that the array will be split in half (that would be ideal).
- Let's say the partition does a pretty good job at splitting the array in half each time. The cost would be O(N log N)
  - Partition takes `O(N)` time
  - Roughly `(log N)` rounds
- However, in the worst case, you may get really unlucky (say the pivot element is either the largest or smallest element). Then partition ends up creating two highly imbalanced groups, and it fails to quickly reduce the size of the problem.

O(N logN) Sorting Summary

- **Merge Sort**
  - A key step is to **merge** two already sorted sub-arrays.
  - Guarantees log-linear time, but is not in-place sorting.
- **Heap Sort**
  - Leverages **heap**'s efficient insert / remove methods
  - The same input array is split into a heap region and sorted region.
  - Guarantees log-linear time, and sorts in place
- **Quick Sort**
  - A key step is to **partition** around a pivot element
  - Sorts in place
  - Does **not** guarantee log-linear time in worst-case scenarios.

Can we do even better?

- **Counting Sort**
- **Bucket Sort**
- **Radix Sort**
  - These achieve `O(N)` cost (linear) in sorting, but they all make assumptions about the input data (i.e. from a discrete set, or are bounded integers).

- We can mathematically prove that the cost of **comparison-based sorting** is no less than `O(N log N)`.
  - See: https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/