

# Reminders and Topics

- **Project 10 is due this Friday**
- **This week, Discussion sections are Wednesday**
- This lecture:
  - **Merge Sort**
  - **Heap Sort**
  - **Quick Sort**

# Merging Two Sorted Arrays

- A **key step** in Merge Sort is ‘merging two sorted arrays’.
  1. Start from the first elements of A and B.
  2. Compare and copy the smaller of the two to output C.
  3. Increment C’s index as well as the array where you picked the smaller element from.
  4. Repeat until either A or B has reached the end.
  5. Append the remaining elements, if any, to C.
- As we’ve discussed in the last lecture: the cost of merging is:  **$O(\text{sizeA} + \text{sizeB})$**

# Merge Sort

- With the merge() method, here is how Merge Sort works:
  - **Divide** the array into two halves
  - Sort each half (**Conquer**). How? **Recursions!**
  - Call merge() to merge the two halves.
- The base case of the recursion?

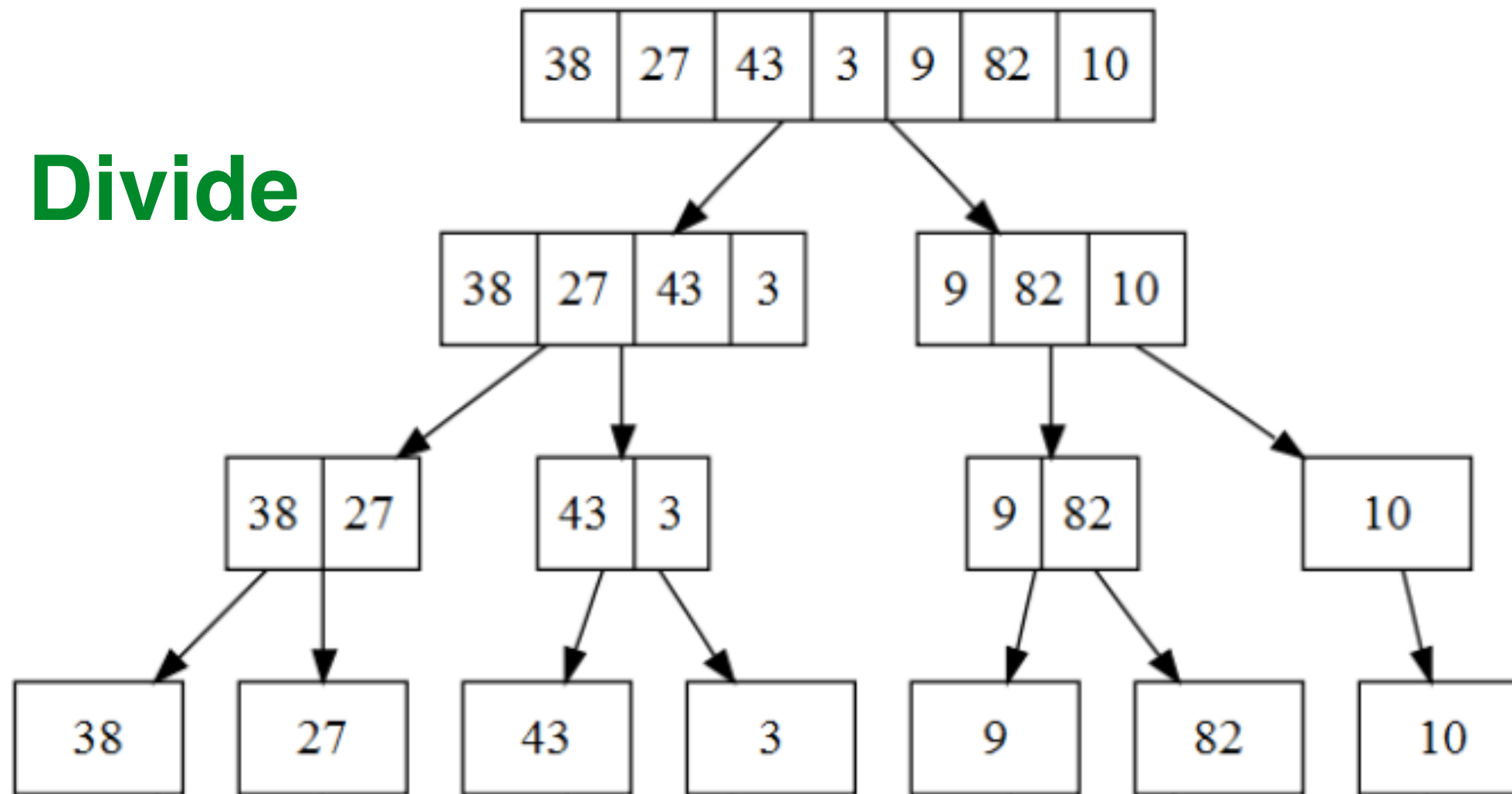
When there is only 1 element left to sort, it's trivially sorted, so return immediately.

```
// assume elements are stored in T[] elems
// temp: scratch buffer to store merged elements

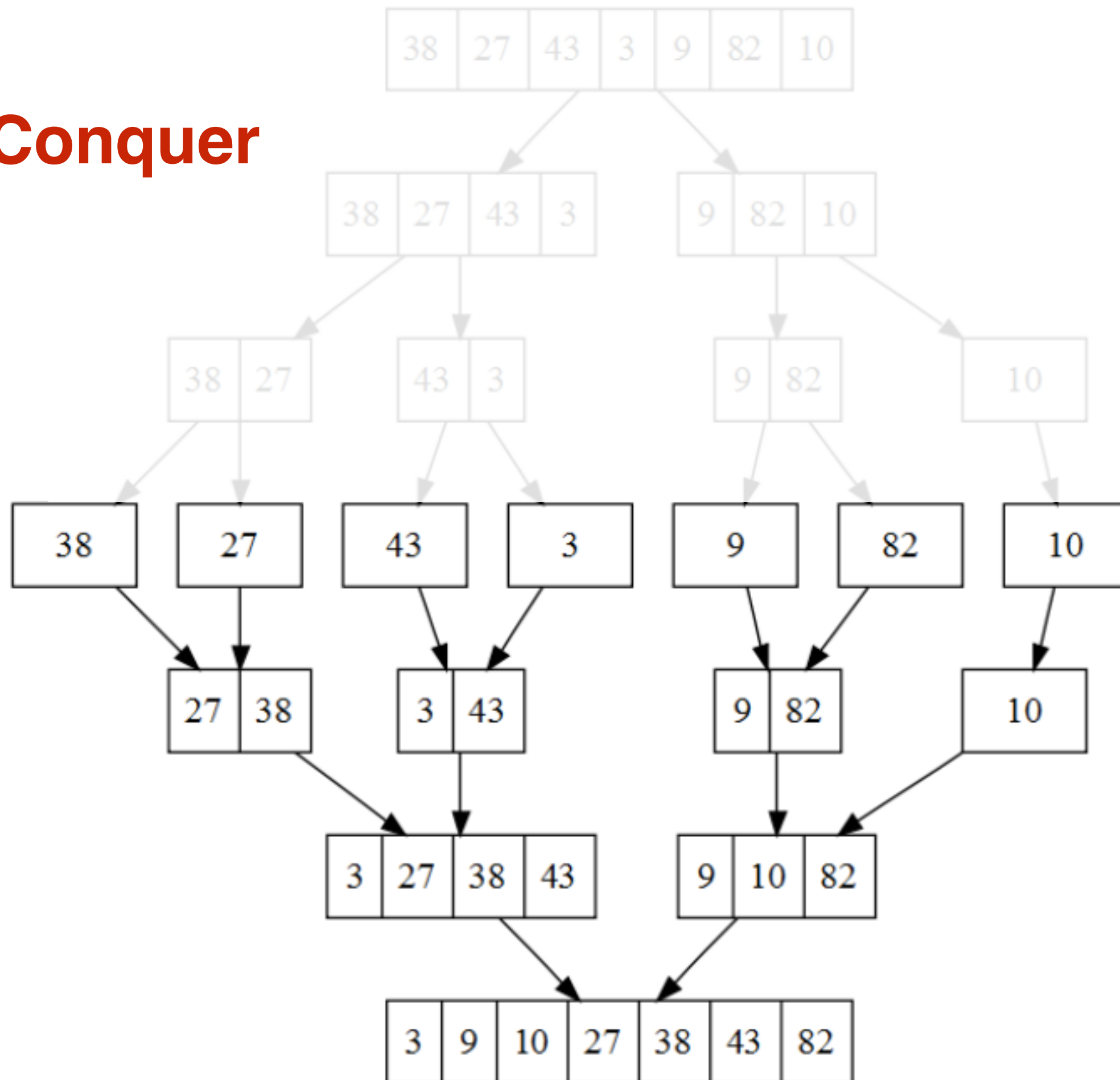
public void mergeSort() {
    T[] temp = (T[])new Object[nElems];
    recMergeSort(temp, 0, nElem-1);
}

public void recMergeSort(T[] temp,
                        int low, int high) {
    if(low == high) return;
    int mid = (low + high) / 2;
    recMergeSort(temp, low, mid);
    recMergeSort(temp, mid+1, high);
    merge(temp, low, mid+1, high);
}
```

Divide



# Conquer



# 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 store 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(temp, i, i+k, i+k+k-1);
        }
    }
}

/* merge(temp, startA, startB, endB) will merge the
   two sub-arrays defined by
   elem[startA, startB-1] and elem[startB, endB]
   to the output temp[startA, endB]
*/
```



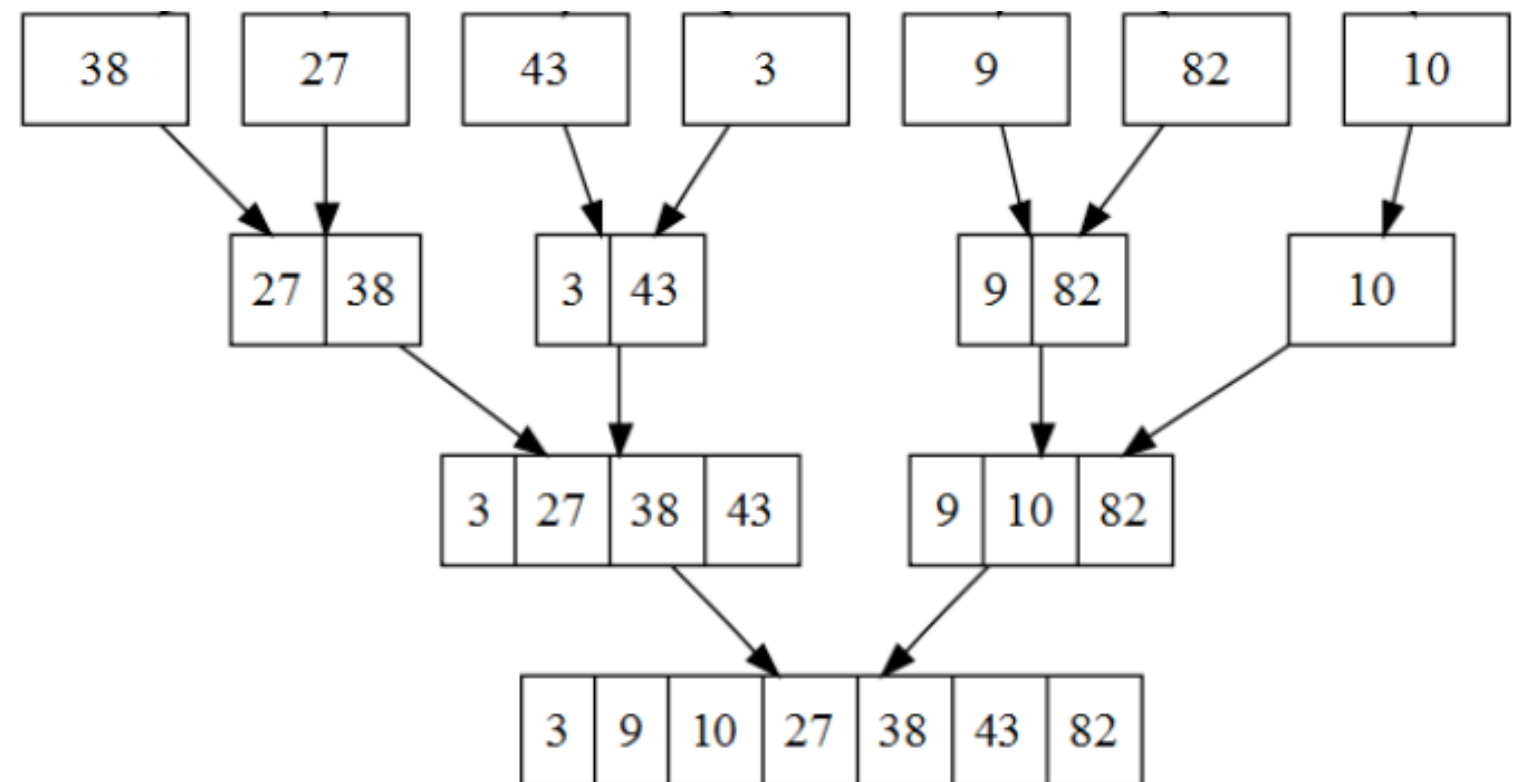
# Merge Sort Cost Analysis

- 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:  **$O(N \log N)$**

# Heap Sort

- While Merge Sort is easy to implement and fast, it requires additional storage (the temp buffer).
- Now let's look at **Heap Sort** and **Quick Sort**, both of which are  $O(N \log N)$  and 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();
```

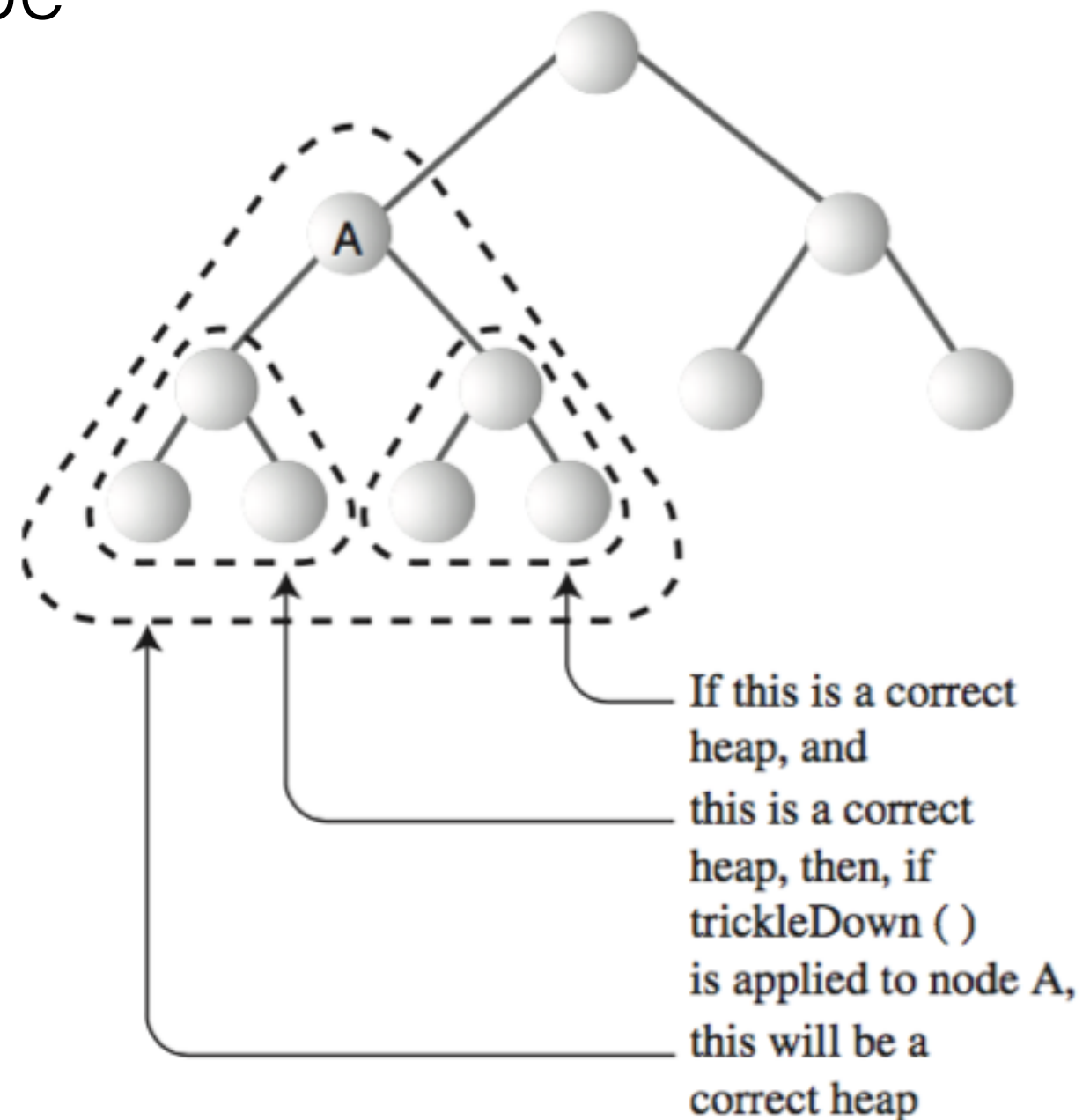
- Assuming heap is a maxHeap, this sorts elements in order. What's the cost?  **$O(N \log N)$**
- Is there additional storage involved here? **Yes, the heap!**

# 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 can 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

- Convert an unsorted array to a heap in place.
- Imagine node A's left subtree and right subtree are both correct 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 bottom 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**.

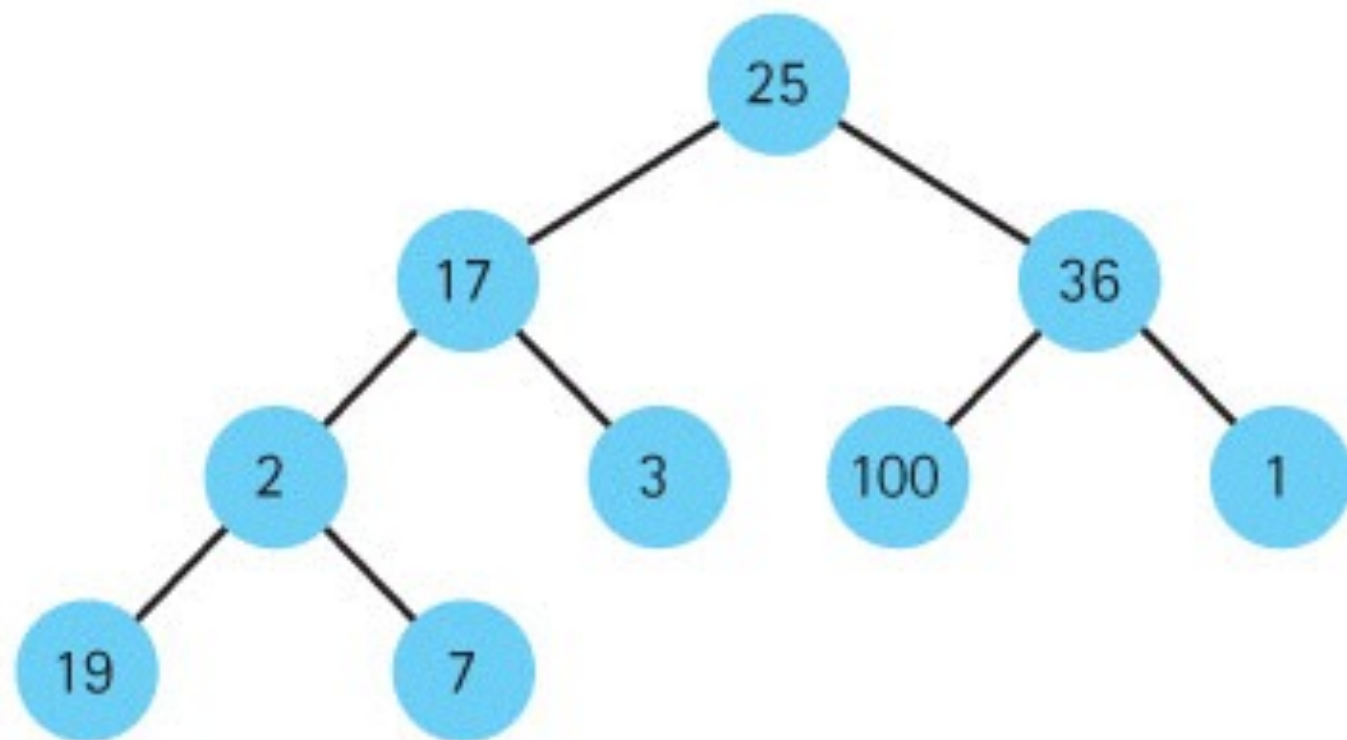
```
public void heapify() {  
    for(int i=last_interior; i >= 0; i--) {  
        bubbleDown(i, nElems-1);  
    }  
}
```

```
/* the second parameter in bubbleDown  
   indicates the end of the heap. Later you  
   will see why it's useful. */
```

# Heapify Example

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

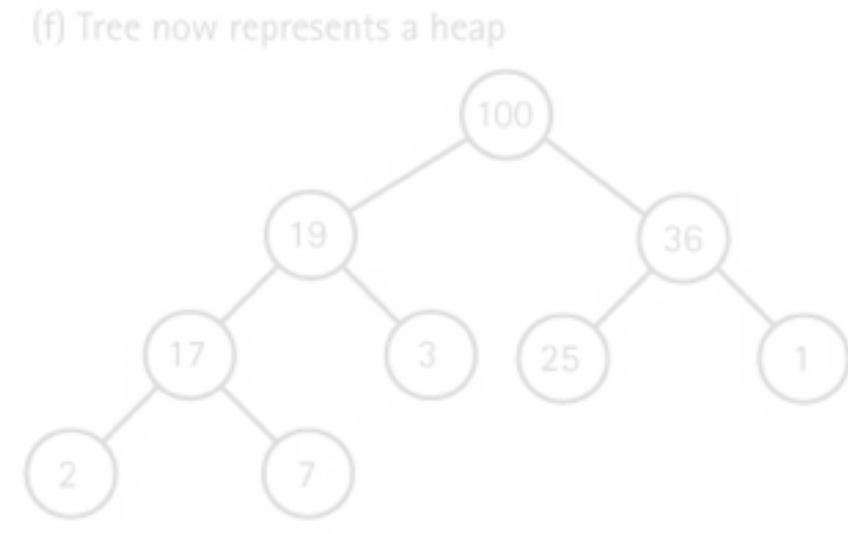
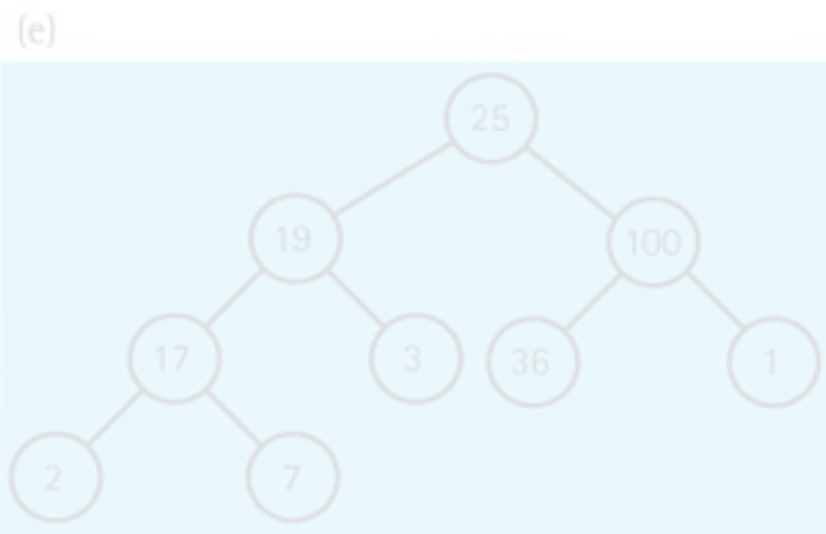
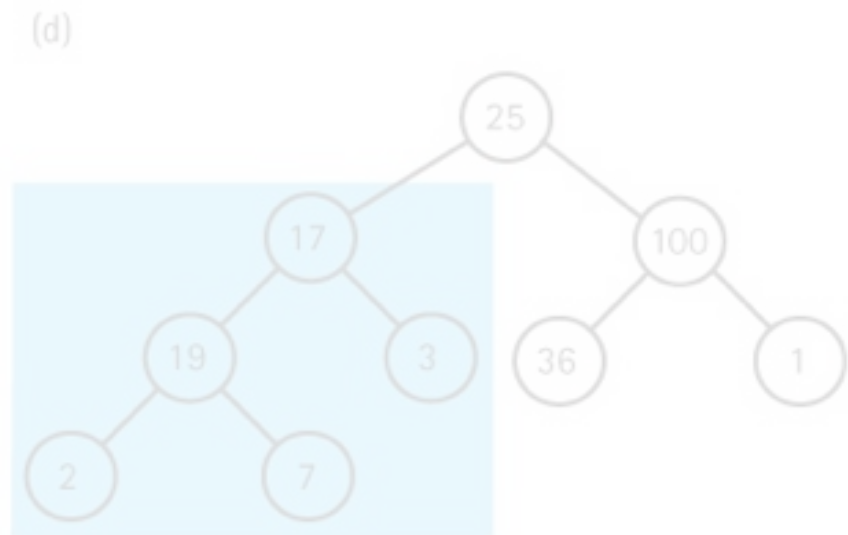
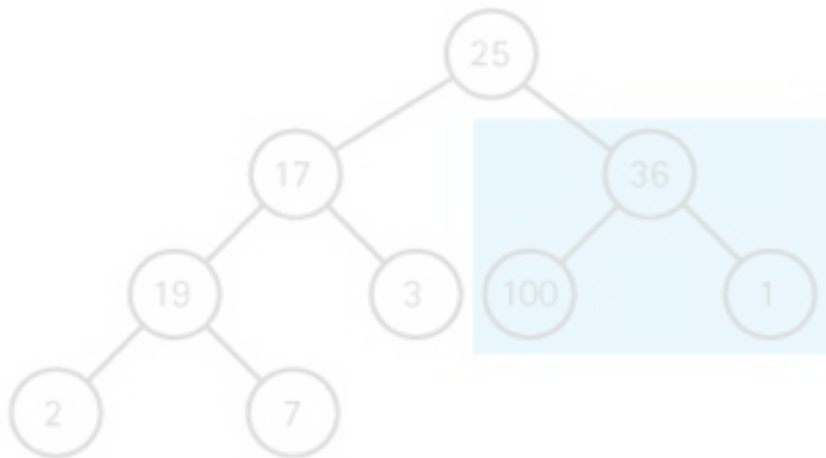
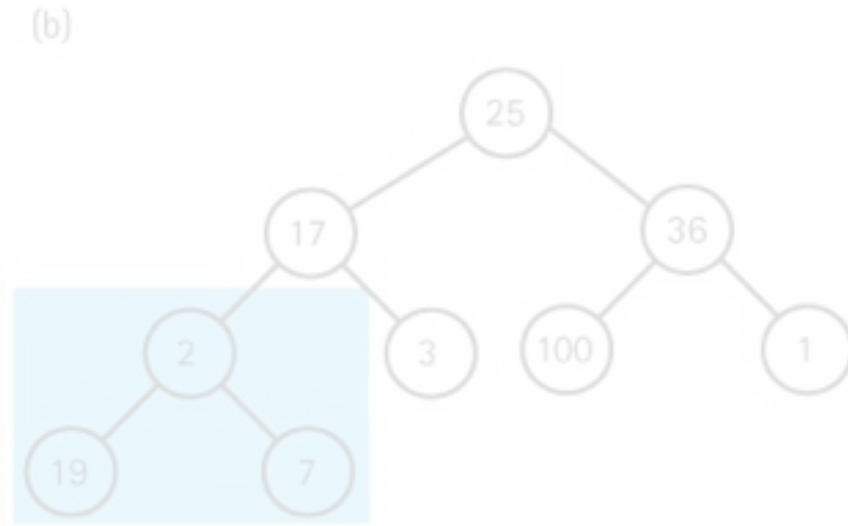
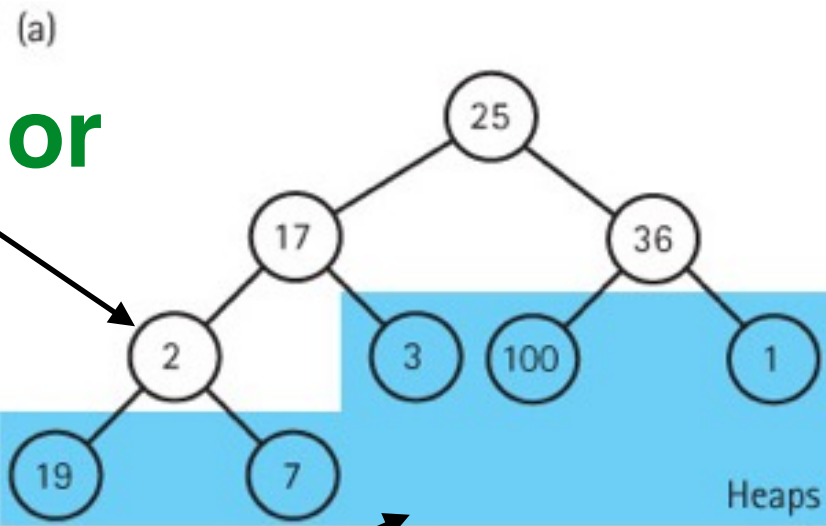
	values
[0]	25
[1]	17
[2]	36
[3]	2
[4]	3
[5]	100
[6]	1
[7]	19
[8]	7



# Heapify Example

**Last interior node**

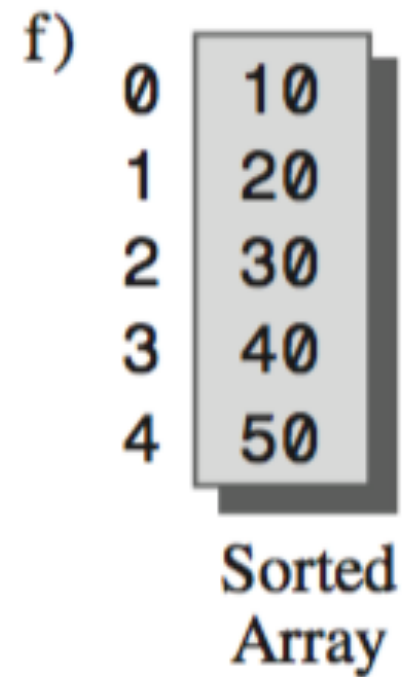
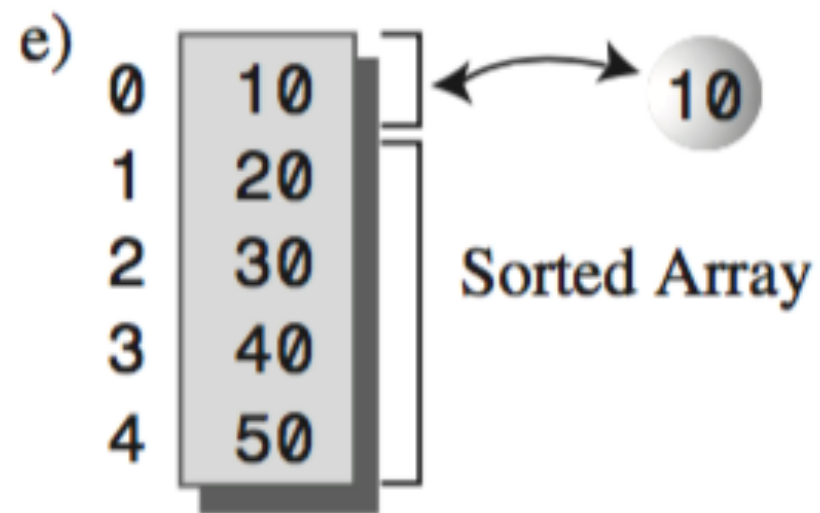
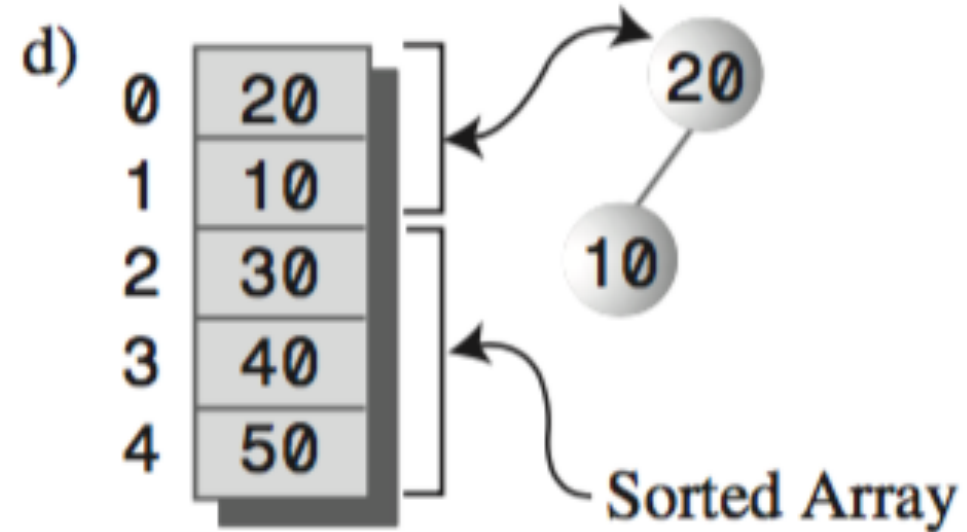
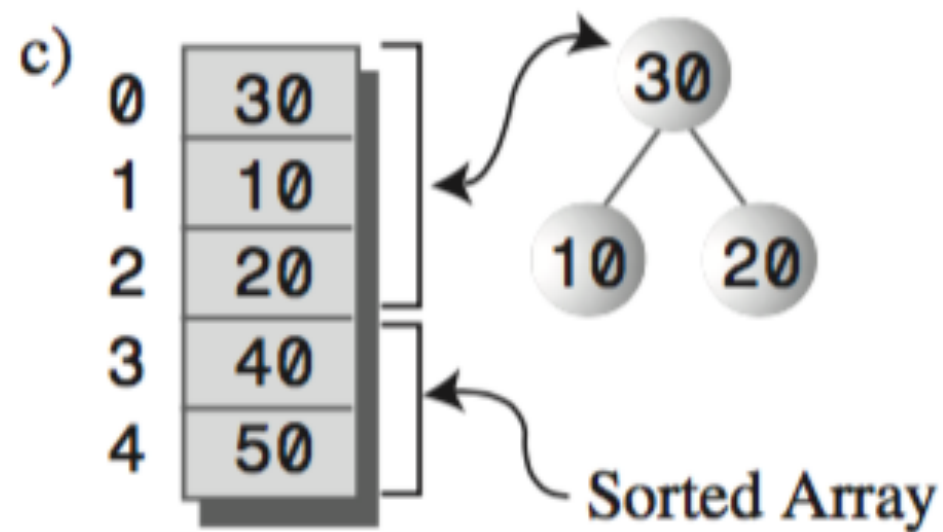
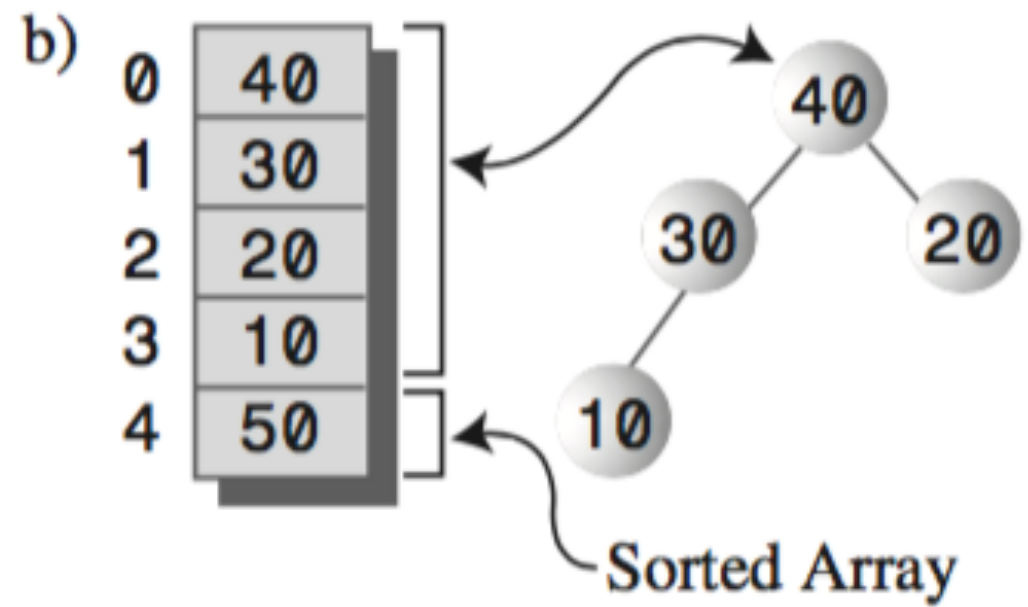
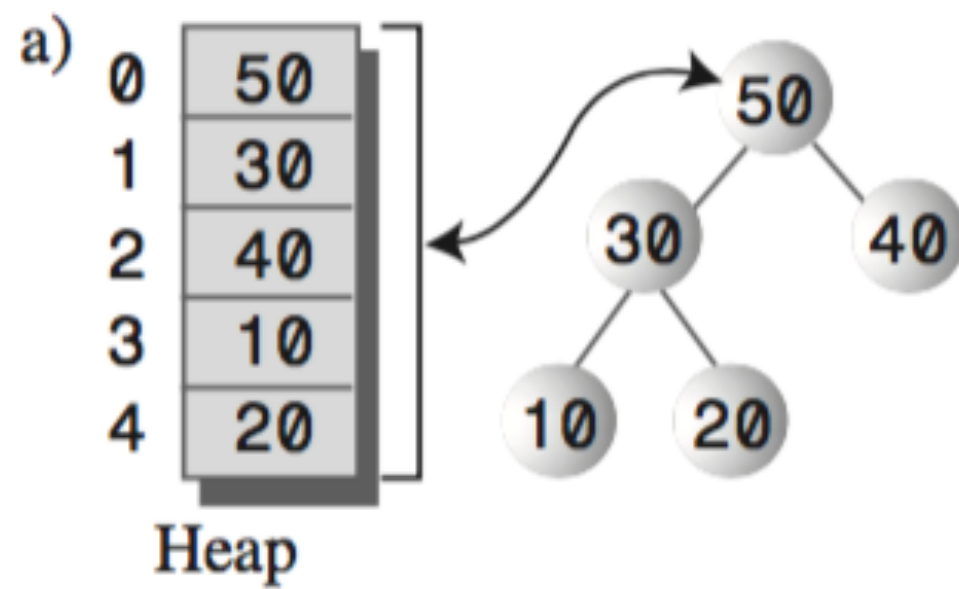
**Leaf nodes**





# 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.
- Do you 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)`



# Heap Sort

- Once we move the last element to index 0, it leaves an empty spot, and that's exactly where the removed root should go to. So we can simply do a swap.
- 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.
- Involves a heapify step (transforms an unsorted array into a heap) and a sorting step (very much like selection sort)
- Both steps are  $O(N \log N)$ 
  - `bubbleDown` on a size  $N$  array costs  $O(\log N)$
  - Both steps involve a  $O(N)$  loop
- Therefore the total cost of Heap Sort is  $O(N \log N)$ 
  - This is also called **log-linear** cost

# 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 5 are placed on the left of it, and those bigger than 5 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[storeIndex]**.
- 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.

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

```
public int partition(int low, int high) {  
    T pivot = a[high];    // 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;  
}
```



# Partition Example

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

# Quick Sort

- With the Partition operation, 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.

# $O(N \log N)$ Sorting Summary

- Merge Sort
  - A key step is to merge two sorted arrays.
  - Requires an additional storage buffer
- Heap Sort
  - Leverages heap's efficient insert / remove methods
  - The same input array is split into a heap region and sorted region.
  - Sorts in place
- Quick Sort
  - A key step is to partition around a pivot element
  - Sorts in place