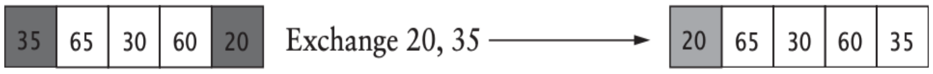
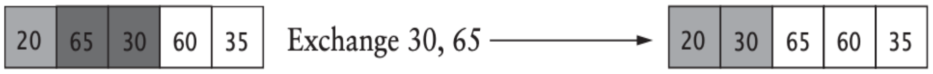
Lesson-7- Class Notes

Selection Sort(pick min and swap, then next min and swap,…)

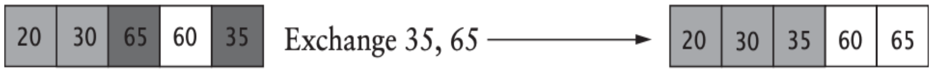
First the selection sort algorithm finds the smallest item in the array (smallest is 20) and moves it to position 0 by exchanging it with the element currently at position 0. At this point, the sorted part of the array consists of the new element at position 0. The values to be exchanged are shaded dark in all diagrams. The sorted elements are in light gray.



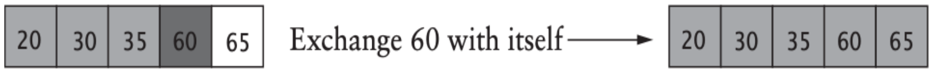
Next, the algorithm finds the smallest item in the subarray starting at position 1 (next smallest is 30) and exchanges it with the element currently at position 1:



At this point, the sorted portion of the array consists of the elements at positions 0 and 1. Next, the algorithm selects the smallest item in the subarray starting at position 2 (next smallest is 35) and exchanges it with the element currently at position 2:



At this point, the sorted portion of the array consists of the elements at positions 0, 1, and 2. Next, the algorithm selects the smallest item in the subarray starting at position 3 (next smallest is 60) and exchanges it with the element currently at position 3:



The element at position 4, the last position in the array, must store the largest value (largest is 65), so the array is sorted.

Sorting

Worst case Scenario: If the is reversed order, you want to sort in natural order
Best case Scenario: If the inputs are already in sorter order, how your algorithm performs.

Insertion Sort (Remember playing a card game)

Think of a number as cards on a table that picked up one at a time in the order, they appear it on the array. In the Insertion sort, generally the element you picked up on the left is sorted and the right is unsorted.

Input Array

57	48	79	65	15	33	52
Index 0	1	2	3	4	5	6

When you pickup 57, only one number in hand and sorted.

57

Step 1: Pick up 48, you need to compare with the left element 57, if 48 is less than 57 shift the elements. Add it in front of 57 so your hand contains the following:

48 57

Step 2: Pick up 79, you need to compare with the left element 57 and 48. 79 is greater than the left elements. So, no need to shift elements, keep 79 in the same position. Your hand contains the following:

48 57 79

Step 3: Pick up 65, you need to compare with the left element 79, 57 and 48. 65 is less than the left element 79. So, shift elements 79 and 65. Other elements are in the right position. Your hand contains the following:

48 57 65 79

Step 4: Pick up 15, you need to compare with the left element 79, 65, 57 and 48. 15 is less than the of all the left elements. So, shift elements and place 15 before 48. Your hand contains the following:

15 48 57 65 79

Step 5: Pick up 33, you need to compare with the left element 79, 65, 57, 48 and 15. 33 is less than the left elements 79,65,57 and 48. So, shift elements and place 33 before 48. Your hand contains the following:

15 33 48 57 65 79

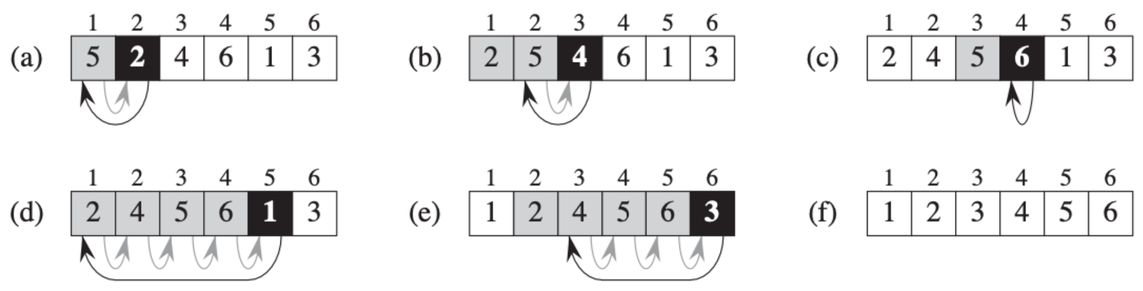
Step 6: Pick up 52, you need to compare with the left element 79, 65, 57, 48, 33 and 15. 52 is less than the left all the left elements 79,65,57. So, shift elements and place 42 before 57. Your hand contains the following:

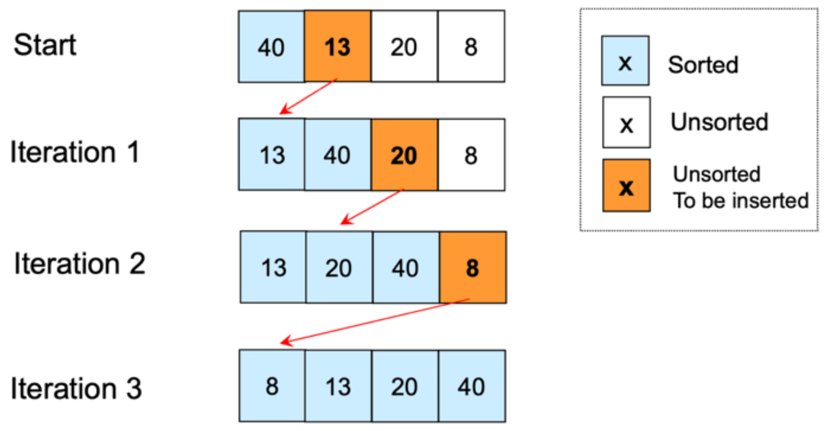
15 33 48 52 57 65 79

The numbers have been sorted in ascending order.

The method described illustrates the idea behind *insertion sort*. The numbers in the array will be processed one at a time, from left to right. This is equivalent to picking up the numbers from the table one at a time. Since the first number, by itself, is sorted, we will process the numbers in the array starting from the second.

When we come to process num[h], we can assume that num[0] to num[h-1] are sorted. We insert num[h] among num[0] to num[h-1] so that num[0] to num[h] are sorted. We then go on to process num[h+1]. When we do so, our assumption that num[0] to num[h] are sorted.





3 5 2 7 8

Insertion sort will do shifting of elements instead of swap.
Pass 1: 3 5 in sorted position, no shifting required

Left part 3 5 2
Store temp = 2, shift 5 to the position 2
Shift 3 to the position 1
Store temp in the zero position

<u>Insertion Sort – Average Case</u>

average case: each element is about halfway in order.

$$\sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2}\left(1+2+3...+(N-1)\right) = \frac{(N-1)N}{4}$$
$$= O\left(N^2\right)$$

<u>Idea of Bubble Sort</u>

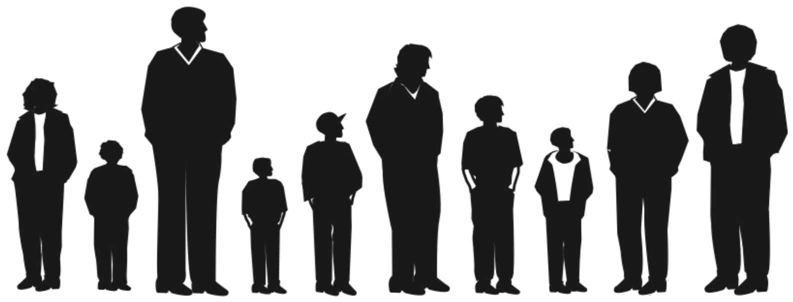


FIGURE 3.1 The unordered baseball team.

You want to arrange according to their height

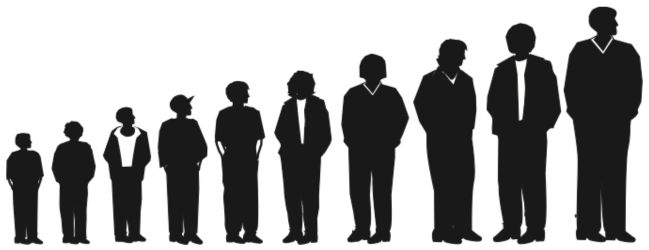
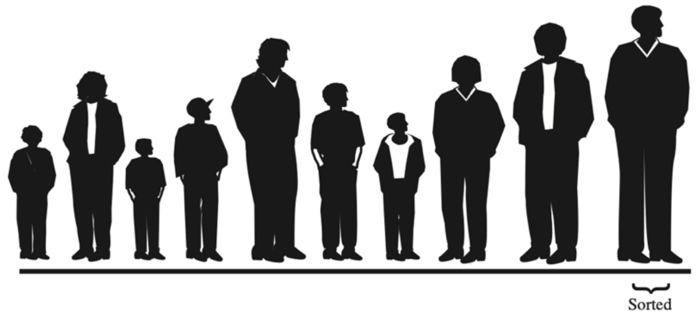


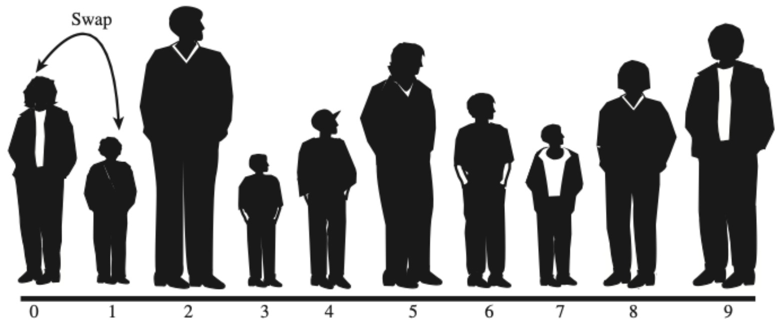
FIGURE 3.2 The ordered baseball team.

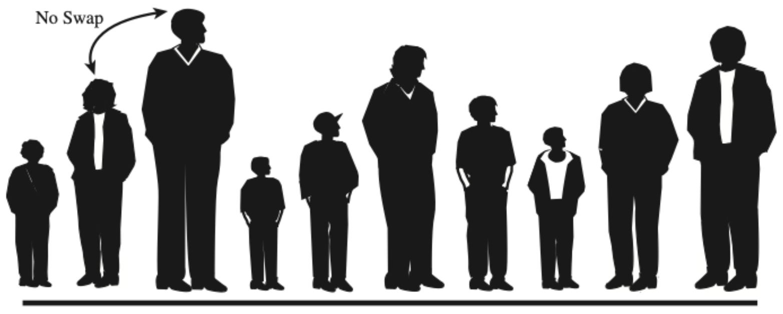
Let's assume there are N players, and the positions they're standing in are numbered from 0 on the left to N-1 on the right.

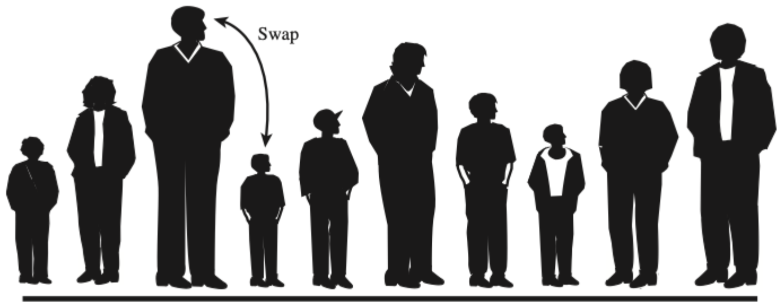
Here are the rules you're following:

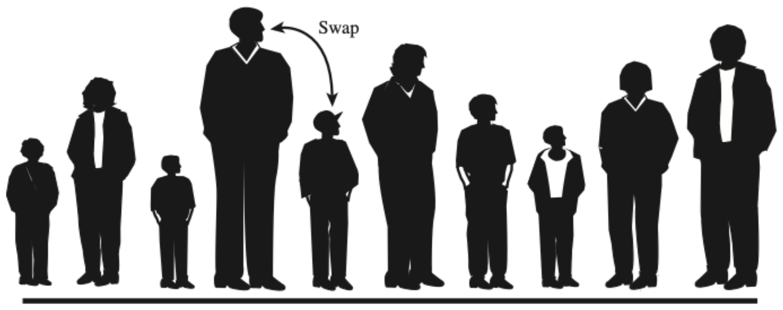
1. **Compare two players.**
2. **If the one on the left is taller, swap them otherwise don't do anything**
3. **Move one position right.**











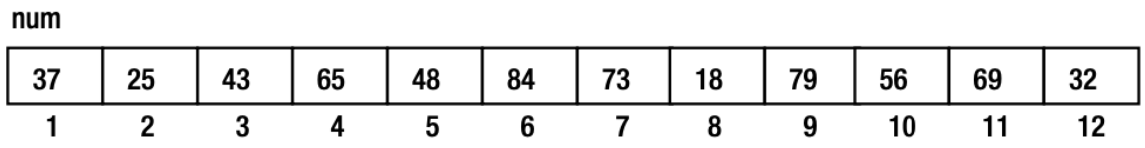
Continued like this, after the pass1, the n-1 position got sorted

after the pass2, the n-2 position got sorted
after the pass3, the n-3 position got sorted and so on.

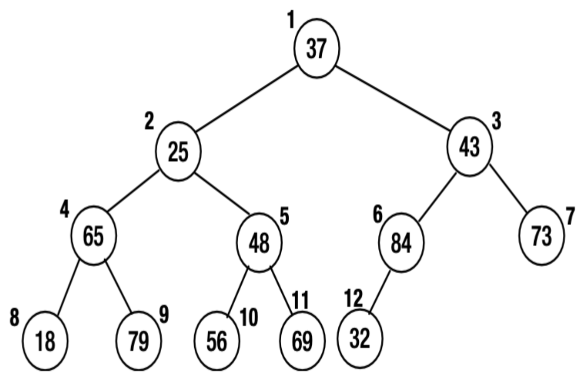
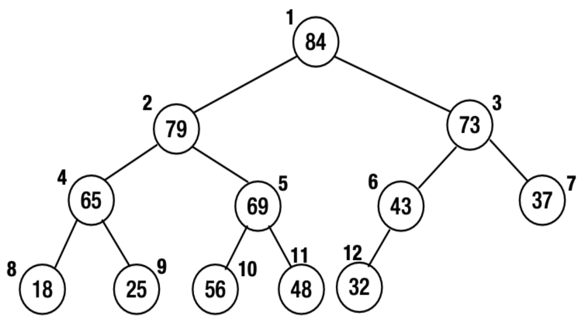
Heap Sort

- To get inputs in a natural sorted for the heap, use Max-Heap. (straight forward)
- To get inputs in a reversed sorted for the heap, use Min-Heap.

Input Array


num

37	25	43	65	48	84	73	18	79	56	69	32
1	2	3	4	5	6	7	8	9	10	11	12

Binary Tree for the given array	**Max – Heap for the given array**
 1 37 2 25 3 43 4 65 5 48 6 84 7 73 8 18 9 79 10 56 11 69 12 32	 1 84 2 79 3 73 4 65 5 69 6 43 7 37 8 18 9 25 10 56 11 48 12 32

9.1.2 The Sorting Process

After conversion to a heap, note that the largest value, 84, is at the root of the tree. Now that the values in the array form a heap, we can sort them in ascending order as follows:

- Store the last item, 32, in a temporary location. Next, move 84 to the last position (node 12), freeing node 1. Then, imagine 32 is in node 1 and move it so that items 1 to 11 become a heap. This will be done as follows:
- 32 is exchanged with its bigger child, 79, which now moves into node 1. Then, 32 is further exchanged with its (new) bigger child, 69, which moves into node 2.

Finally, 32 is exchanged with 56, giving us Figure 9-4.

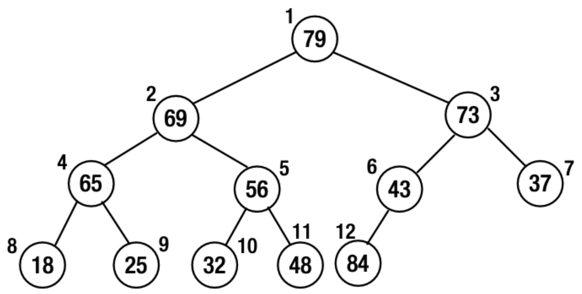


***Figure 9-4.** After 84 has been placed and the heap is reorganized*

At this stage, the second largest number, 79, is in node 1. This is placed in node 11, and 48 is "sifted down" from node 1 until items 1 to 10 form a heap. Now, the third largest number, 73, will be at the root. This is placed in node 10, and so on. The process is repeated until the array is sorted.

If you repeat the same process, all the elements in the array get sorted.

18 25 32 37 43 48 56 65 69 73 79 84

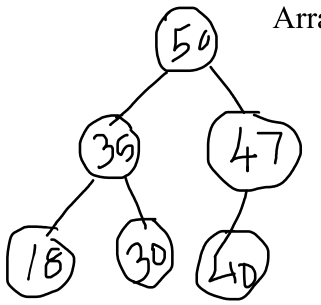
Max-heap – Heap Sort

1. Construct a Max-heap (Root is always the max value)
2. Delete the root, replace with last element.
3. Restore the heap property using downheap.

Rule : ***key***(***v***) <= ***key***(***parent***(***v***))

Index is Empty.

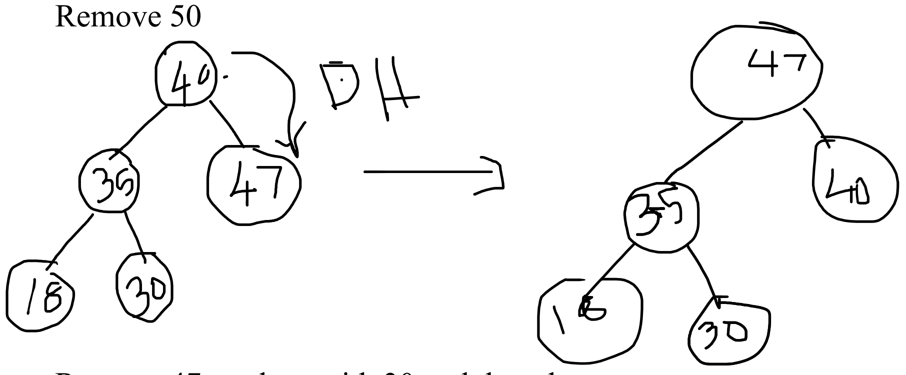
Input Heap



Array : 50 35 47 18 30 40

Remove Max and store in your end of the array(keep storing n,n-1, n-2,…1)

Remove 50



Remove 47, replace with 30 and downheap

Shell Sort – Example with 16 elements using gaps 8, 3 and 1. You can chose different gaps. The general gaps followed n/2, n/4, … 1.

num

67	90	28	84	29	58	25	32	16	64	13	71	82	10	51	57
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The increments decrease in size (hence the term *diminishing increment sort*), with the last one being 1.

Using increment 8, we eight-sort the array. This means we sort the elements that are eight apart. We sort elements 1 and 9, 2 and 10, 3 and 11, 4 and 12, 5 and 13, 6 and 14, 7 and 15, and 8 and 16. This will transform `num` into this:

16	64	13	71	29	10	25	32	67	90	28	84	82	58	51	57
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Next, we three-sort the array; that is, we sort elements that are three apart. We sort elements (1, 4, 7, 10, 13, 16), (2, 5, 8, 11, 14), and (3, 6, 9, 12, 15). This gives us the following:

16	28	10	25	29	13	57	32	51	71	58	67	82	64	84	90
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Note that, at each step, the array is a little closer to being sorted. Finally, we perform a one-sort, sorting the entire list, giving the final sorted order:

10	13	16	25	28	29	32	51	57	58	64	67	71	82	84	90
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Shell sort Visualization: https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html

<u>Shell Sort – Slide Algorithm – Execution with the data(slide-37)</u>

arr = [80, 93, 60, 12, 42, 30, 68, 85, 10]

First, we calculate maxGap as floor((arr.length - 1) / 3), which is floor((9 - 1) / 3)=floor(8 / 3)=2

Start with h = 1
h = 1 * 3 + 1 = 4 ,

// compute the maximum gap size
while (h ≤ maxGap) **do**
        h ← h * 3 + 1   = 4
	Start with h = 1
	h = 1 * 3 + 1 = 4 , 4<=2 , condition false

while (0 < h) **do,** h=4	Perform segmentInsertionSort(arr, 4) gap=4

segmentInsertionSort(arr, h)	[80, 93, 60, 12, 42, 30, 68, 85, 10] After pass 1 [10, 30, 60, 12, 42, 93, 68, 85, 80]

h = (h-1) /3 --> 4-1/3 = 1
while (0 < h) **do,** h=1
	Perform segmentInsertionSort(arr, 1) gap=1
	After pass 2
	[10, 12, 30, 42, 60, 68, 80, 85, 93]

<u>Review Questions</u>

1. Need to know the working procedure of the below five sorting algorithms and performance analysis.
    a. Selection-sort
    b. Insertion-sort
    c. Bubble –sort
    d. Heap-sort
    e. Shell-sort
2. What is In-Place sort.