Sorting Algorithms

Sorting Definition

It is a process for arranging a collection of items in an order. Sorting can arrange both numeric and alphabetic data in increasing or decreasing order.

I. Bubble Sort

It is one of the oldest sorts known. The bubble sort got its name because of the way the biggest elements "bubble" to the top. It based on the property of a sorted list that any two adjacent elements are in sorted order.

1. Compare each element (except the last one) with its neighbor to the right
2. If they are out of order, swap them
3. This puts the largest element at the very end
4. The last element is now in the correct and final place
5. Compare each element (except the last *two*) with its neighbor to the right
6. If they are out of order, swap them
7. This puts the second largest element next to last
8. The last two elements are now in their correct and final places
9. Compare each element (except the last *three*) with its neighbor to the right
10. Continue as above until you have no unsorted elements on the left

Example

Consider a set of data: 5 9 2 8 4 6 3.

Bubble sort first compares the first two elements, the 5 and the 9.
Because they are already in sorted order, nothing happens.
The next pair of numbers, the 9 and the 2, are compared.
Because they are not in sorted order, they are swapped and the data becomes: 5 2 9 8 4 6 3.
To better understand the "bubbling" nature of the sort, watch how the largest number, 9, bubbles" to the top in the first iteration of the sort.

First iteration

(5 9) 2 8 4 6 3 --> compare 5 and 9, no swap
5 (9 2) 8 4 6 3 --> compare 9 and 2, swap
5 2 (9 8) 4 6 3 --> compare 9 and 8, swap
5 2 8 (9 4) 6 3 --> compare 9 and 4, swap
5 2 8 4 (9 6) 3 --> compare 9 and 6, swap
5 2 8 4 6 (9 3) --> compare 9 and 3, swap
5 2 8 4 6 3 9 --> first iteration complete

Notice that in the example, the largest element, 9, got swapped all the way into its correct position at the end of the list. This happens because in each comparison, the larger element is always pushed towards its place at the end of the list.

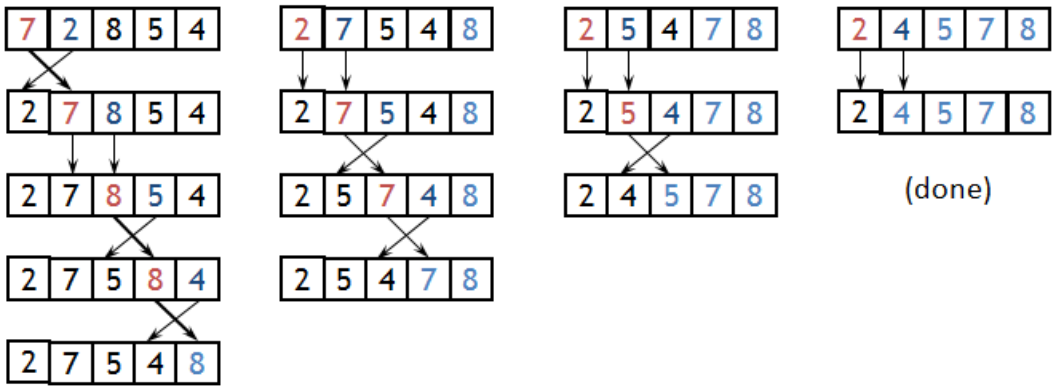
Second iteration
In the second iteration, the second-largest element will be bubbled up to its correct place in the same manner:
(5 2) 8 4 6 3 9 --> compare 5 and 2, swap
2 (5 8) 4 6 3 9 --> compare 5 and 8, no swap
2 5 (8 4) 6 3 9 --> compare 8 and 4, swap
2 5 4 (8 6) 3 9 --> compare 8 and 6, swap
2 5 4 6 (8 3) 9 --> compare 8 and 3, swap
2 5 4 6 3 8 9 --> no need to compare last two because the last element is known to be the largest.

Third iteration
(2 5) 4 6 3 8 9 -->compare 2 and 5 no swap,
2 (5 4) 6 3 8 9 -->compare 5 and 4 swap,
2 4 (5 6) 3 8 9 -->compare 5 and 6 no swap,
2 4 5 (6 3) 8 9 -->compare 6 and 3 swap,
2 4 5 3 6 8 9 -->no need to compare the last three elements

Fourth iteration
(2 4) 5 3 6 8 9 --> compare 2 and 4, no swap
2 (4 5) 3 6 8 9 --> compare 4 and 5, no swap
2 4 (5 3) 6 8 9 --> compare 5 and 3, swap
2 4 3 5 6 8 9 --> no need to compare the last four elements.

Fifth iteration
(2 4) 3 5 6 8 9 --> compare 2 and 4, no swap
2 (4 3) 5 6 8 9 --> compare 4 and 3, swap
2 3 4 5 6 8 9 --> compare 4 and 5, no swap
2 3 4 5 6 8 9 --> no need to compare the last five elements.

Sixth iteration
(2 3) 4 5 6 8 9 --> compare 2 and 3, no swap. No need to compare the last six elements.



Bubble sort Algorithm (input array of size N):

```
for(i=n-1;i>=1;i--)
{
     for(y=1;y<=i; y++)
      {
        if(x[y-1]>x[y])
              {
        temp=x[y-1];
        x[y-1]=x[y];
        x[y]=temp;
            }
      }
}
```

Output sorted array of Size N

Code for bubble sort

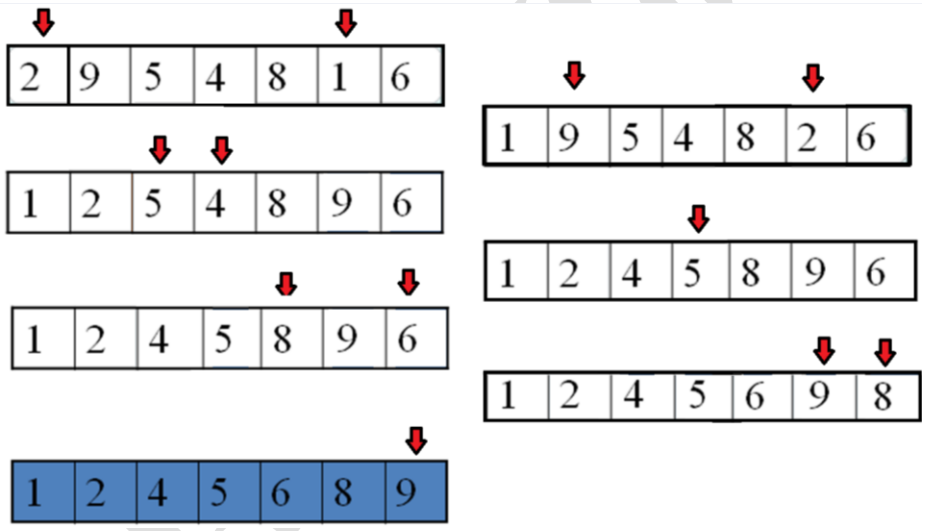
```
public static void bubbleSort(int[] a)
 {
  int outer, inner;
  for (outer = a.length - 1; outer > 0; outer--)
    { // counting down
      for (inner = 0; inner < outer; inner++)
        { // bubbling up
           if (a[inner] > a[inner + 1])
             { // if out of order...
               int temp = a[inner];         // ...then swap
               a[inner] = a[inner + 1];
               a[inner + 1] = temp;
         }
    }
  }
}
```

II. Selection sort

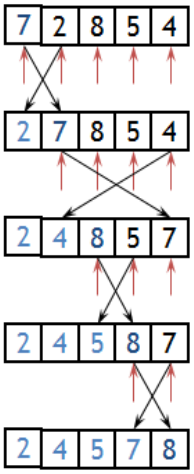
1. Given an array of length n,
2. Search elements 0 through n-1 and select the smallest
3. Swap it with the element in location 0
4. Search elements 1 through n-1 and select the smallest
5. Swap it with the element in location 1
6. Search elements 2 through n-1 and select the smallest
7. Swap it with the element in location 2
8. Search elements 3 through n-1 and select the smallest
9. Swap it with the element in location 3
10. Continue in this fashion until there is nothing left to search

Example of selection sort

1. int [] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted



2.



The selection sort might swap an array element with itself--this is harmless, and not worth checking for.

3.

35	65	30	60	20	scan 0-4, smallest 20→ swap 35 and 20
20	65	30	60	35	scan 1-4, smallest 30→swap 65 and 30
20	30	65	60	35	scan 2-4, smallest 35→swap 65 and 35
20	30	35	60	65	scan 3-4, smallest 60→swap 60 and 60
20	30	35	60	65	done

Selection Sort Algorithm

for **fill** = 0 to n-1 do
  set **posMin** to **fill**
  for **next** = **fill**+1 to n-1 do
        if item at **next** < item at **posMin**
        set **posMin** to **next**
  Exchange item at **posMin** with one at **fill**

Code for selection sort

```
public static void selectionSort(int[] a)
{
  int outer, inner, min;
   for (outer = 0; outer < a.length - 1; outer++)
    {
      min = outer;
      for (inner = outer + 1; inner < a.length; inner++)
       {
         if (a[inner] < a[min])
             min = inner;     // Invariant: for all i, if outer <= i <= inner, then a[min] <= a[i]
        }
      // a[min] is least among a[outer]..a[a.length - 1]
      int temp = a[outer];
```

```
        a[outer] = a[min];
        a[min] = temp;
        // Invariant: for all i <= outer, if i < j then a[i] <= a[j]
      }
}
```

III. Insertion Sort

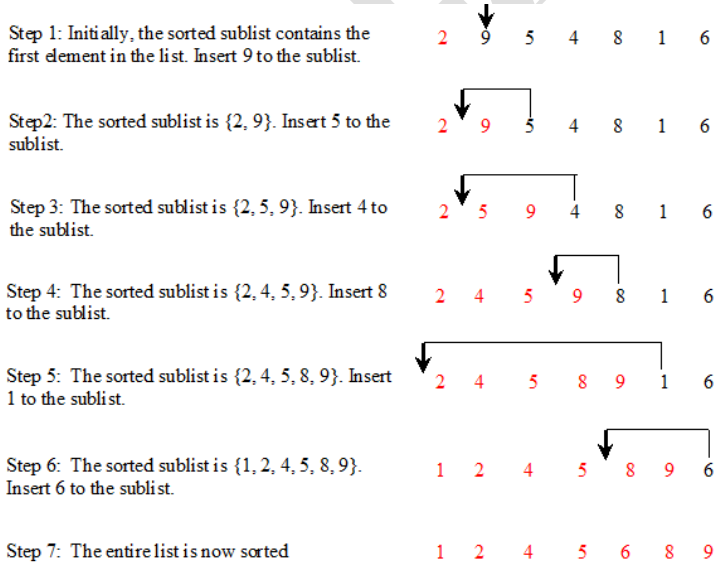
One of the simplest methods to sort an array is an insertion sort, widely used for small data sets, if the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place.

An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

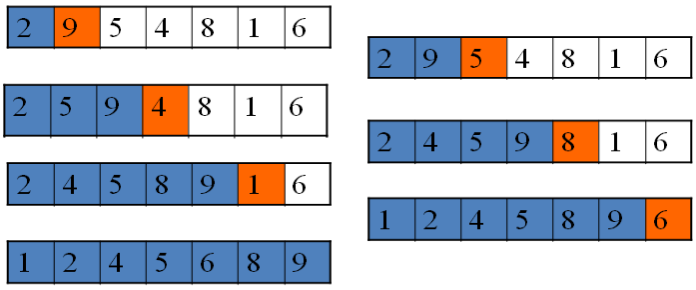
Based on technique of card players to arrange a hand, player keeps cards picked up so far in *sorted order.* When the player picks up a new card, he makes room for the new card, then inserts it in its proper place

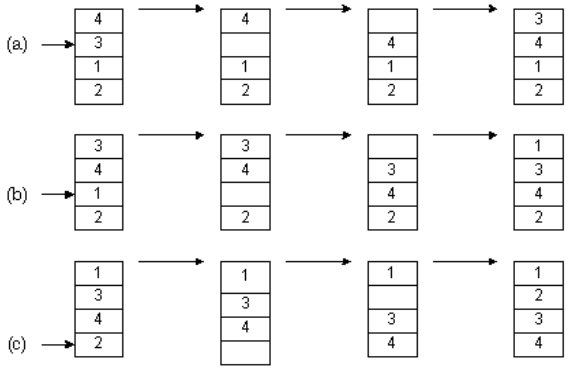
Example:

int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted



The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.





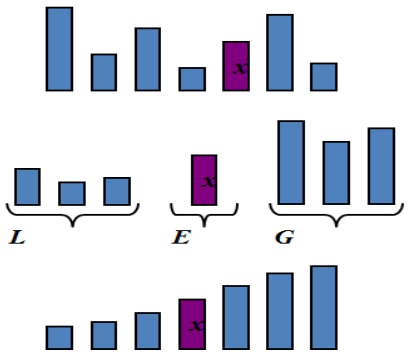
Starting near the top of the array in the Figure (a) we extract the 3. Then the above elements are shifted down until we find the correct place to insert the 3. This process repeats in Figure (b) with the next number. Finally, in Figure (c), we complete the sort by inserting 2 in the correct place.

Insertion Sort Algorithm

```
for i = 1 to N-1 do
    newElement = list[ i ]
    location = i - 1
     while (location ≥ 0) and (list[ location ] > newElement) do
       list[ location + 1 ] = list[ location ]
       location = location - 1
     end while
    list[ location + 1 ] = newElement
end for
```

IV. Quick Sort

As the name implies **quicksort** is the fastest known sorting algorithm in practice. Quick-sort is a randomized sorting algorithm based on the divide-and-conquer prototype Select a pivot and split the data into two groups: (< pivot) and (> pivot): Recursively apply Quicksort to the subgroups.



1. Divide: pick a random element ***x*** (called pivot) and partition ***S*** into
   - ***L*** elements less than ***x***

- ***E*** elements equal ***x***
- ***G*** elements greater than ***x***

2. Recur: sort ***L*** and ***G***
3. Conquer: join ***L***, ***E*** and ***G***

Basic Algorithm:

quicksort(S)

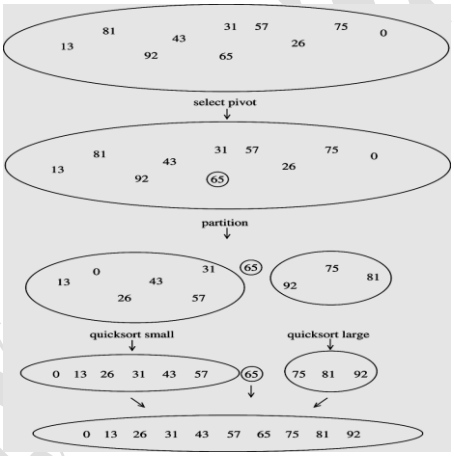
{

If the number of elements in *S* is 0 or 1, then return S
Pick any element *v* in *S*. This is called the **pivot**.
Partition *S-{v}* into 2 disjoint groups:
$L=\{x \epsilon S-\{v\} | x<=v\}$ and $G=\{x \epsilon S-\{v\} | x>v\}$
return ({quicksort(L)} ∪ {v} ∪ {quicksort(G)})

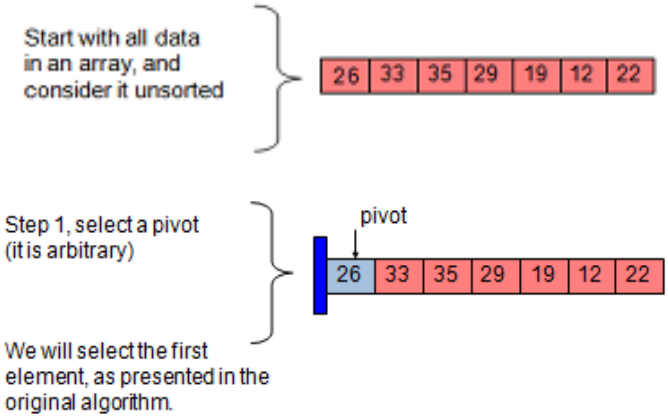
}

Partition step ambiguous for elements equal to the pivot.

Example:



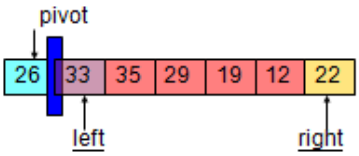
Example:



Step 2, start process of dividing data into LEFT and RIGHT groups:

The LEFT group will have elements less than the pivot.
The RIGHT group will have elements greater that the pivot.

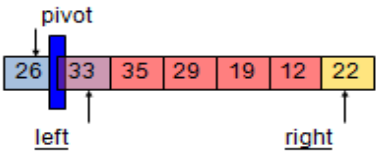
Use markers left and right



Step 3,
If left element belongs to LEFT group, then increment left index.

If right index element belongs to RIGHT, then decrement right.

Exchange when you find elements that belong to the other group.

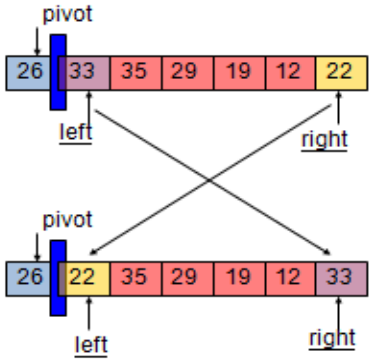


Step 4:

Element 33 belongs to RIGHT group.

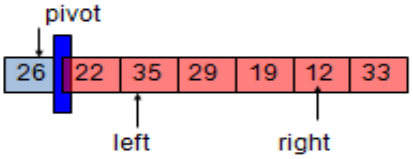
Element 22 belongs to LEFT group.

Exchange the two elements.



Step 5:

After the exchange, increment left marker, decrement right marker.

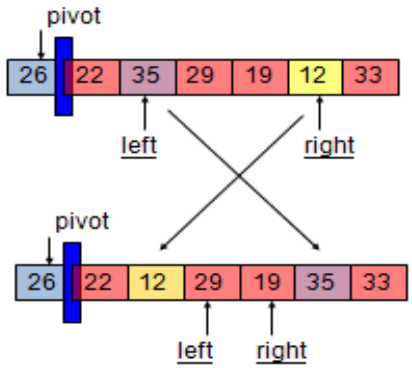


Step 6:

Element 35 belongs to RIGHT group.

Element 12 belongs to LEFT group.

Exchange, increment left, and decrement right.

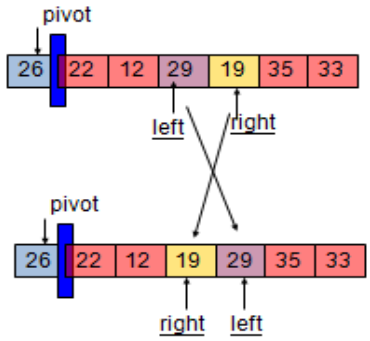


Step 7:

Element 29 belongs to RIGHT.

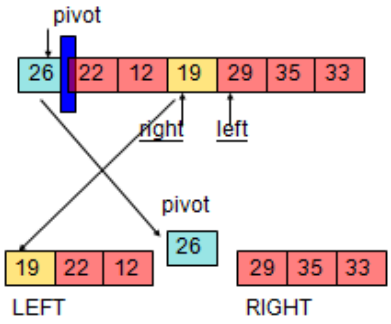
Element 19 belongs to LEFT.

Exchange, increment left, decrement right.



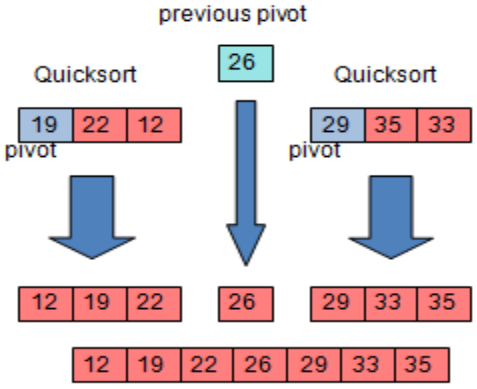
Step 8:
When the left and right markers pass each other, we are done with the partition task.

Swap the right with pivot.



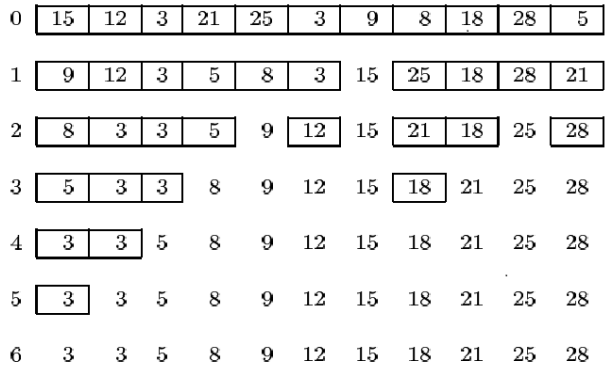
Step 9:
Apply Quicksort to the LEFT and RIGHT groups, recursively.

Assemble parts when done



Example

Recursive implementation with the left most array entry selected as the pivot element.



Quick sort code (p: first element, r: last element

```
Quicksort (A as array, L as int, R as int)
{
   if (L < R)
    {
      q = Partition(A, L, R)
      Quicksort(A, 0, L)
      Quicksort(A, q+1, R)
     }
}

Int Partition (A as array, low as int, high as int)
{
    pivot = A[low]
    leftwall = low
    for i = low + 1 to high
          {
             if (A[i] < pivot) then
             {
                leftwall = leftwall + 1
                swap(A[i], A[leftwall])
             }
          }
    swap(A[low],A[leftwall])
   return (leftwall)
}
```

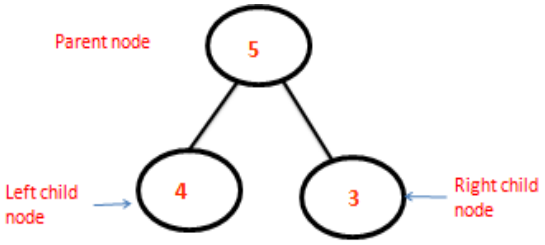
- **Initial call:** Quicksort (A, 0, n-1), where n in the length of A
- **H.W:** write swap (Element1, Element2) algorithm

V. *Heap*

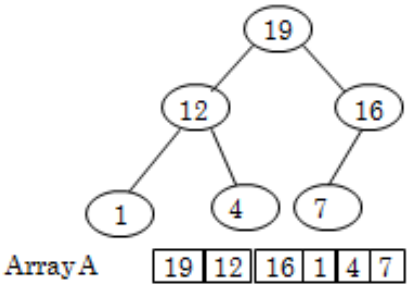
A heap is a data structure that stores a collection of objects (with keys), and has the following properties: ***Complete Binary Tree and has Heap Order***. It is implemented as an array where each node in the tree corresponds to an element of the array.

Binary Tree

A **binary tree** is a tree data structure in which each node has at most two children (referred to as the *left* child and the *right* child). In a binary tree, the *degree* of each node can be at most two. Binary trees are used to implement binary search and heap sort.

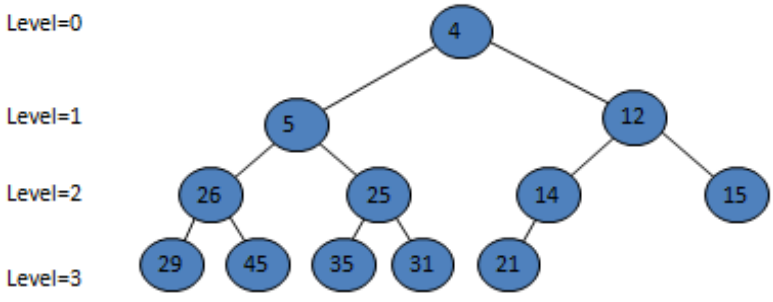


Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest as follows



I	0	1	2	3	4	5	6	7	8	9	10	11
array	4	5	12	26	25	14	15	29	45	35	31	21

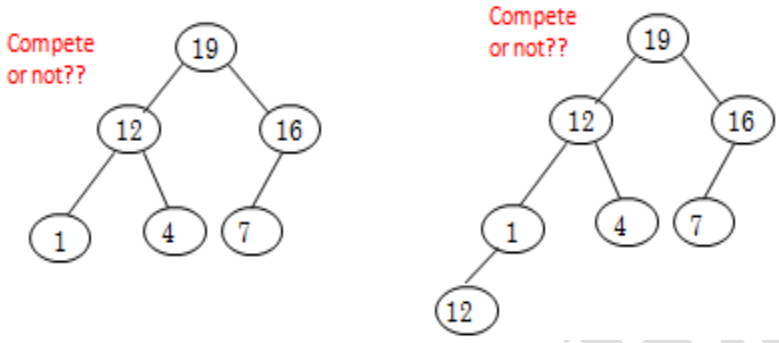
size = 12



Binary tree of Size 12 and depth 3

PARENT (i) = ceiling (i/2)-1, LEFT (i) = 2i+1, RIGHT (i) =2i + 2

A **complete binary tree** is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible (left justified). This type of tree is used as a specialized data structure called a heap.

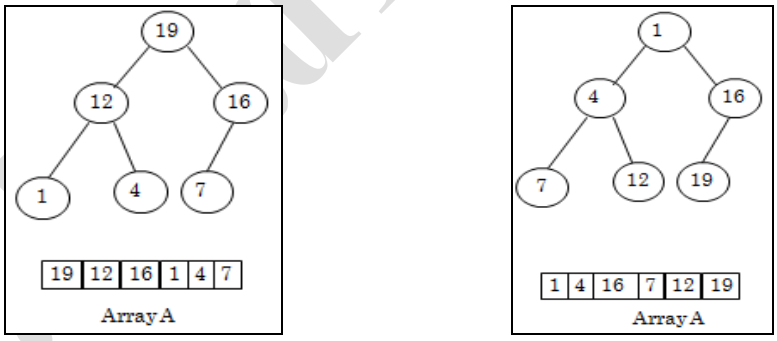


Heap Order

A **heap** is a *complete binary tree* with one (or both) of the following heap order properties:

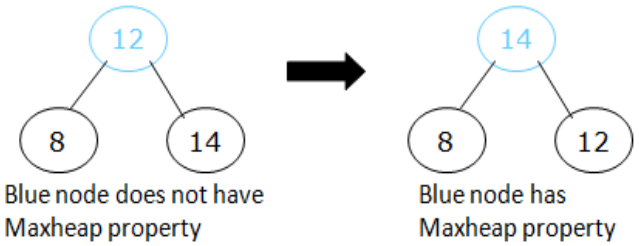
- **MinHeap property:** Each node must have a key that is less or equal to the key of each of its children.
- **MaxHeap property:** Each node must have a key that is greater or equal to the key of each of its children.

A binary heap satisfying the MinHeap property is called a MinHeap, while a binary heap satisfying the MaxHeap property is called a MaxHeap. Recall, that a complete binary tree may have missing nodes only on the right side of the lowest level.



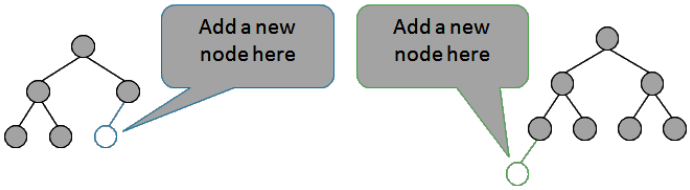
Max Heap Min heap

siftUp: Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child. This is sometimes called **sifting up**. Notice that the child may have *lost* the heap property.



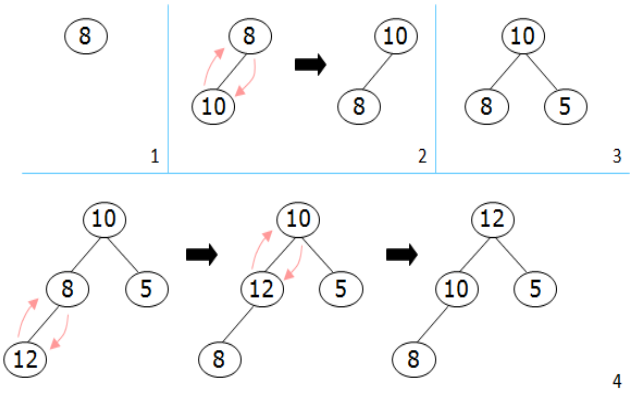
1. ***Constructing a heap***

A tree consisting of a single node is automatically a heap. We construct a heap by adding nodes one at a time as follows: "Add the node just to the right of the leftmost node in the deepest level. If the deepest level is full, start a new level



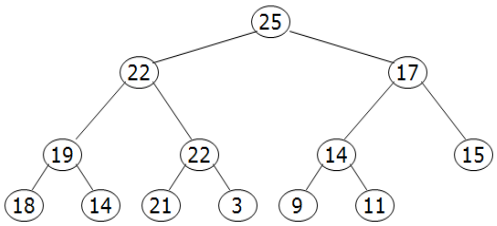
Each time we add a node, we may destroy the heap property of its parent node. To fix this, we siftup. But each time we siftup, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node. We repeat the sifting up process, moving up in the tree, ***until either*** we reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or we reach the root.

Example:

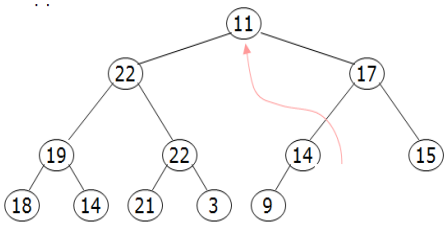


A sample heap

Here's a sample binary tree after it has been heapified. Notice that heapified does *not* mean sorted. Heapifying does *not* change the shape of the binary tree;

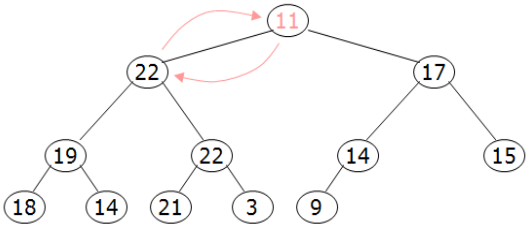


2. *Removing the root*

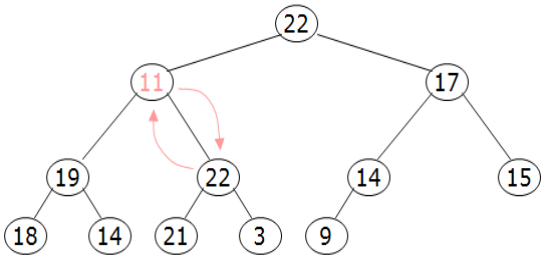


Question: How can we fix the binary tree so it is heap once again?
Solution: remove the rightmost leaf at the deepest level and use it for the new root

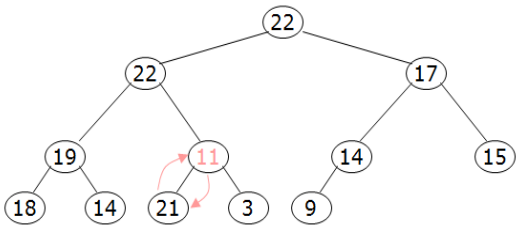
Our binary tree is no longer having the Max heap property. However, *only the root* lacks the Maxheap property. We can siftUp() the root. After doing this, one and only one of its children may have lost the Maxheap property



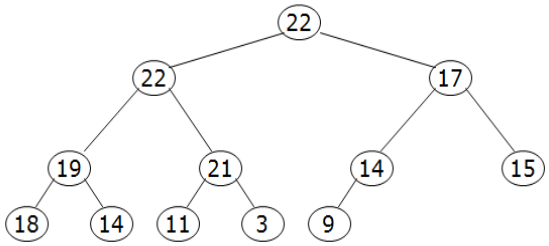
Now the left child of the root (still the number) lacks the Maxheap property



We can siftUp() this node. After doing this, one and only one of its children may have lost the Maxheap property. Now the right child of the left child of the root (still the number 11) lacks the Maxheap property:



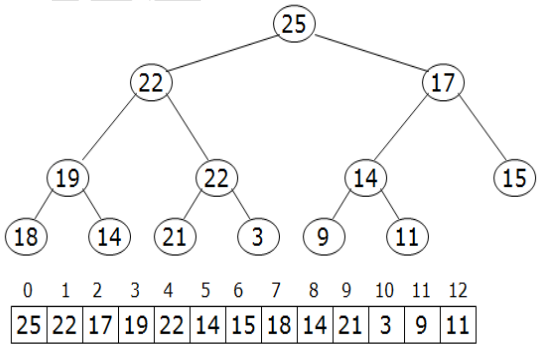
We can siftUp() this node. After doing this, one and only one of its children may have lost the Maxheap property; but it doesn't, because it's a leaf. Our tree is once again a heap, because every node in it has the Maxheap property



Once again, the largest (or *a* largest) value is in the root. We can repeat this process until the tree becomes empty. This produces a sequence of values in order largest to smallest

Mapping into an array

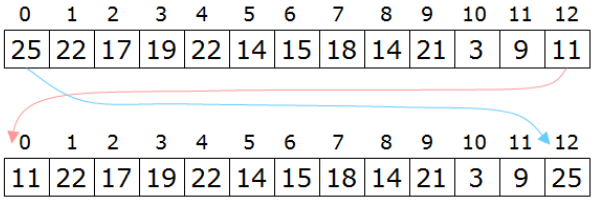
What do heaps have to do with sorting an array? Here's the neat part:
Because the compete binary tree can be represented as an array, all our operations on the complete binary trees can be represented as operations on *arrays*



Remember:
The left child of index i is at index 2*i+1
The right child of index i is at index 2*i+2
Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

- The "root" is the first element in the array.
- The "rightmost node at the deepest level" is the last element
- Swap them...



Pretend that the last element in the array no longer exists; that is, the "last index" is (9). A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap

Procedures on Heap

1. **Heapify**: picks the largest child key and compare it to the parent key. If parent key is larger then heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

```
Heapify(A, i)
{
    L ← left(i)     (using 2i+1)
    R ← right(i)   (using 2i+2)
    if L < heapsize[A] and A[L] > A[i]
       then largest ←L
       else largest ← i
    if R < heapsize[A] and A[R] > A[largest]
       then largest ← R
    if largest != i
       then swap A[i] ←→ A[largest]
          Heapify(A, largest)
}
```

2. **Build Heap**: We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . n] into a heap. Since the elements in the subarray A[n/2 +1 . . n] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A)
{
    Heapsize ←length[A]
    for i ← Heapsize / 2 down to 0 do
      Heapify(A, i)
}
```

Heap Sort Algorithm:

The heapsort algorithm consists of two phases:

- Build a heap from an arbitrary array
- Use the heap to sort the data

The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . n]. Since the maximum element of the array stored at the root A [1], it can be put into its correct final position by exchanging it with A[n] (the last element in A). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All those are needed to restore the heap property.

```
Heapsort(A)
{
    Buildheap(A)
    for i ← length[A]-1 down to 0
      do
       swap A[0] ←→ A[i]
       heapsize[A] ← heapsize[A] - 1
       Heapify(A, 0)
}
```

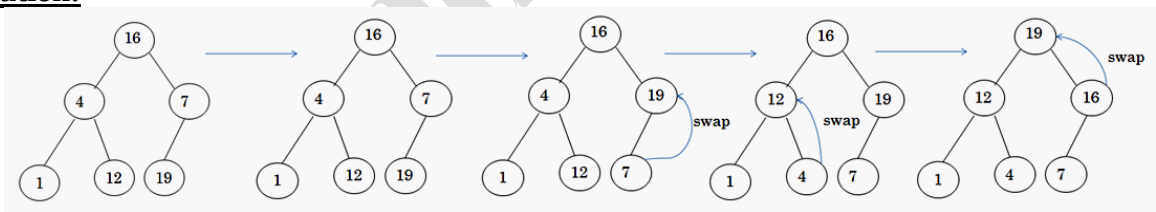
Remember:

- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap

Example:
Convert the following array to a heap

16	4	7	1	12	19

Solution:



19	12	16	1	4	7