Lecture 25
Heap Sort

November 09, 2021
Tuesday

BUILDING THE HEAP ALGORITHM

```
Heapify (array, size, i)

    set i as largest

    leftChild = 2i + 1
    rightChild = 2i + 2

    if leftChild > array[largest]
        set leftChildIndex as largest
    if rightChild > array[largest]
        set rightChildIndex as largest

    swap array[i] and array[largest]
```

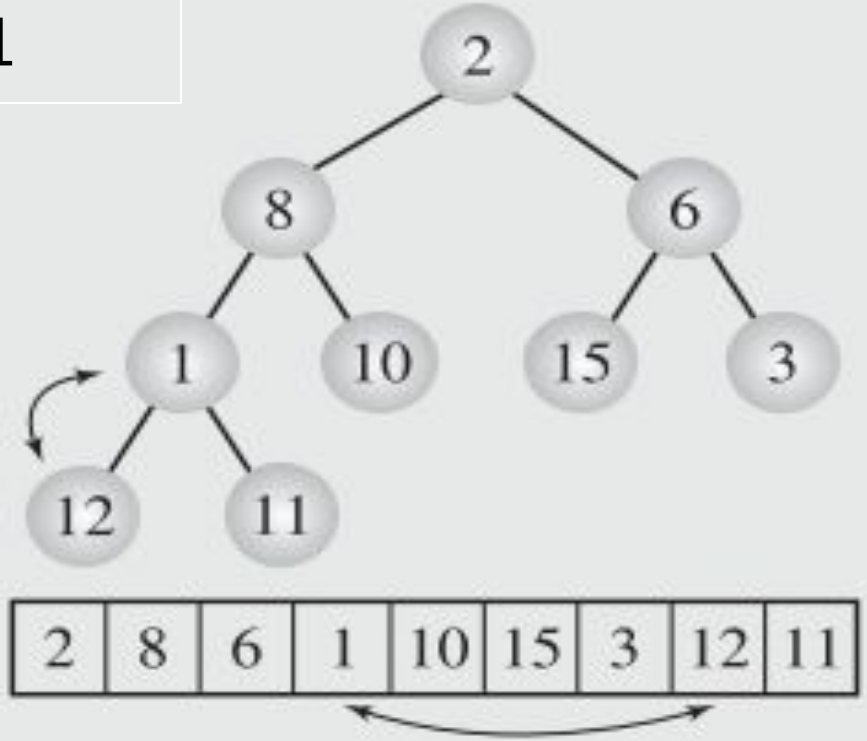
HEAPIFY IMPLEMENTATION

```
void heapify(int heap [ ], int size, int i)
{
        int largest = i;          int left = 2 * i + 1;       int right = 2 * i + 2;
        if (left < size && heap [ left] > heap [ largest ] )
             largest = left;
        if (right < size && heap [ right ] > heap [ largest ] )
             largest = right;

        if (largest != i)
        {
             swap ( heap [ i ], heap [ largest ] );
             heapify (heap, size, largest);
  }
}
```

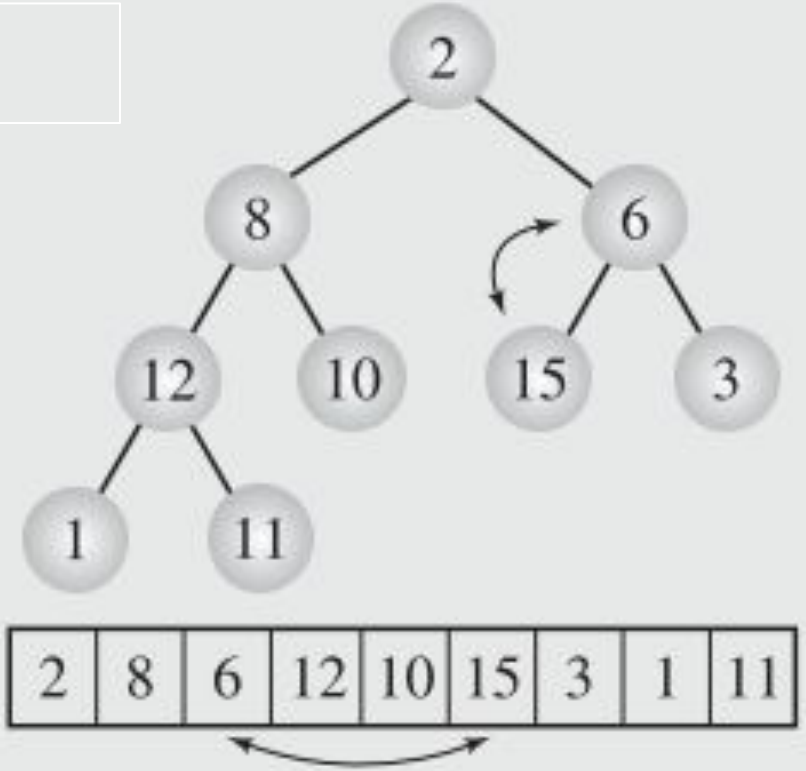
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



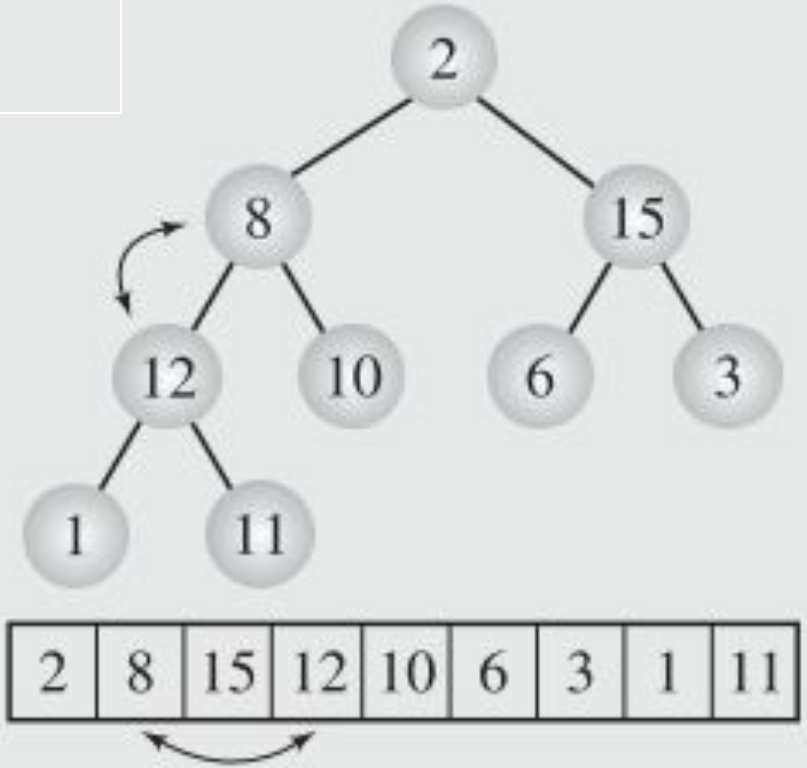
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



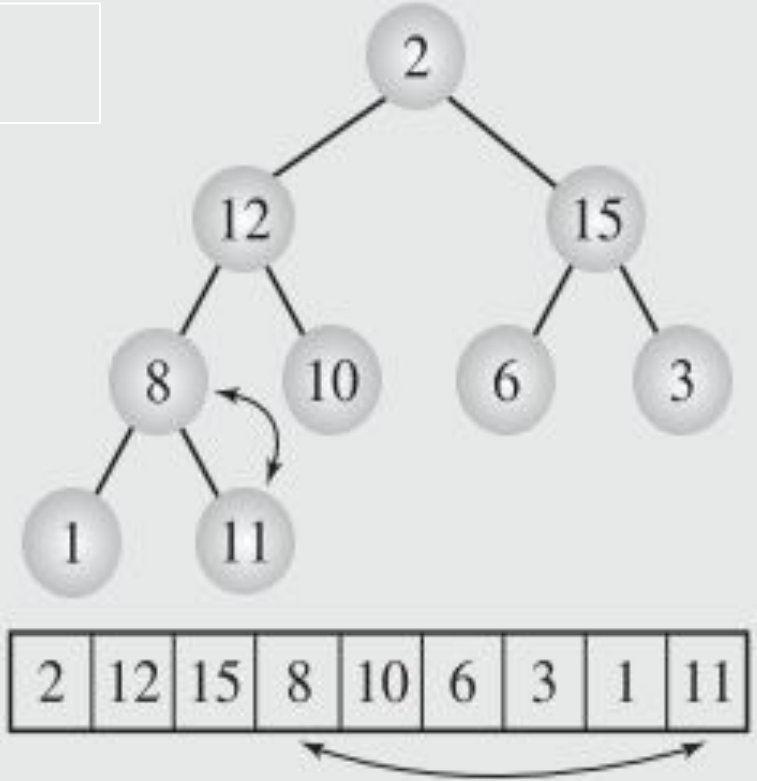
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



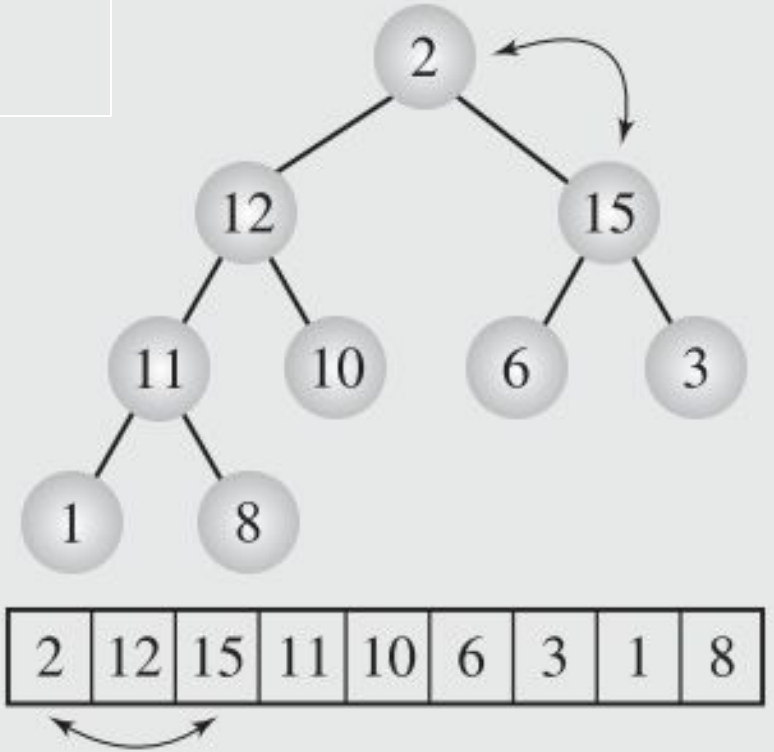
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



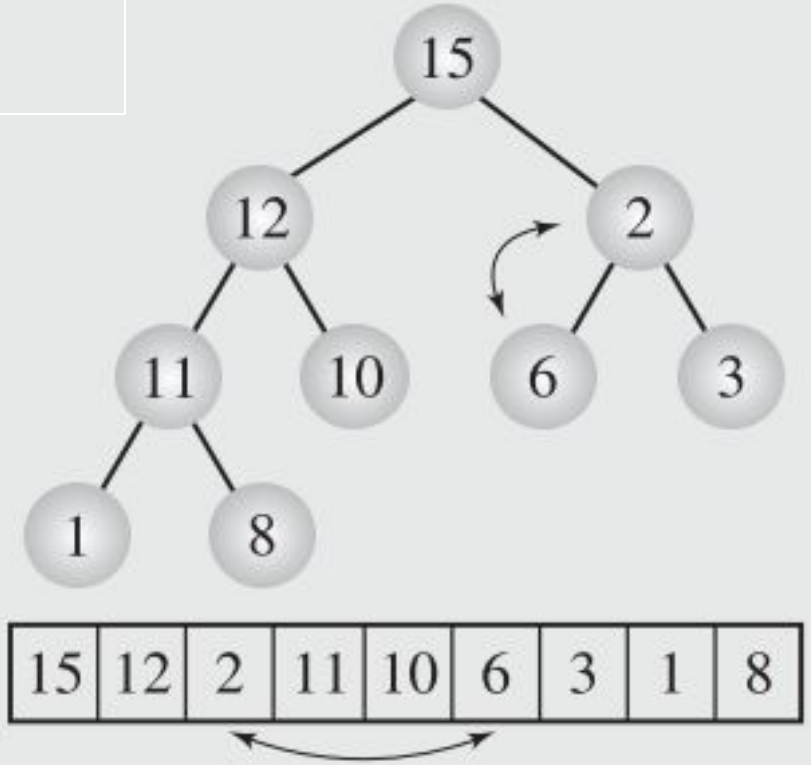
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



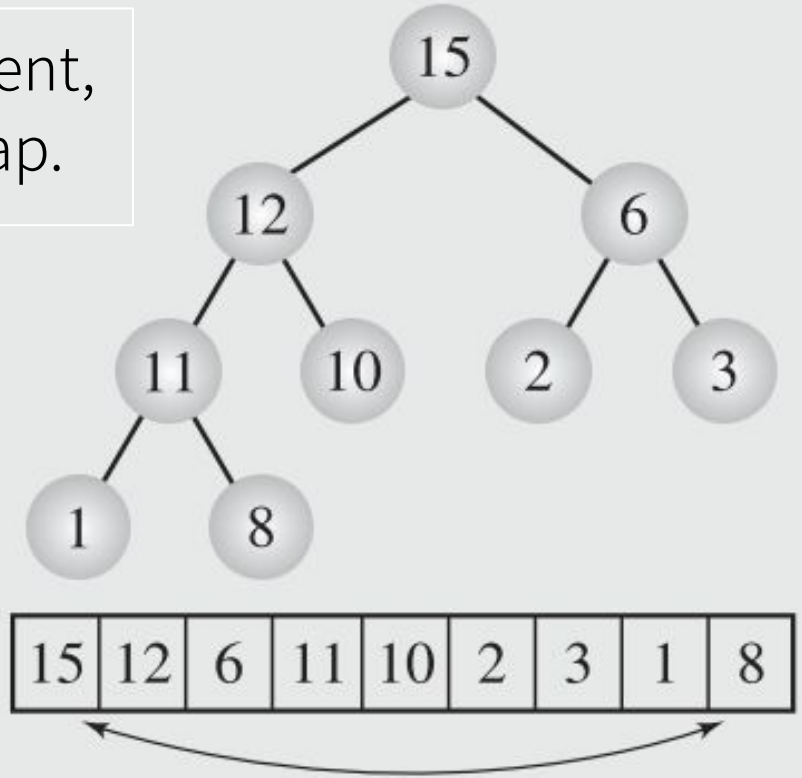
BUILDING THE HEAP FIRST

2, 8, 6, 1, 10, 15, 3, 12, 11



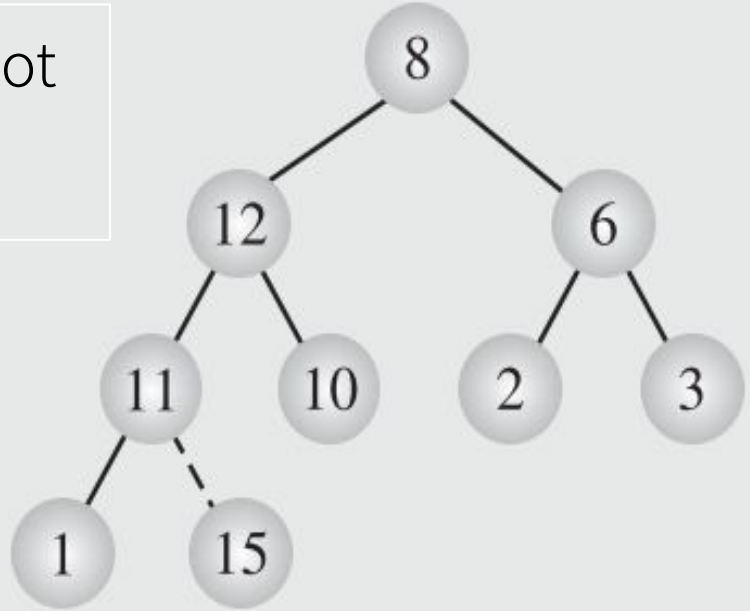
EXECUTION OF THE HEAP SORT

Swap the root with the last element, and decrease the size of the heap.



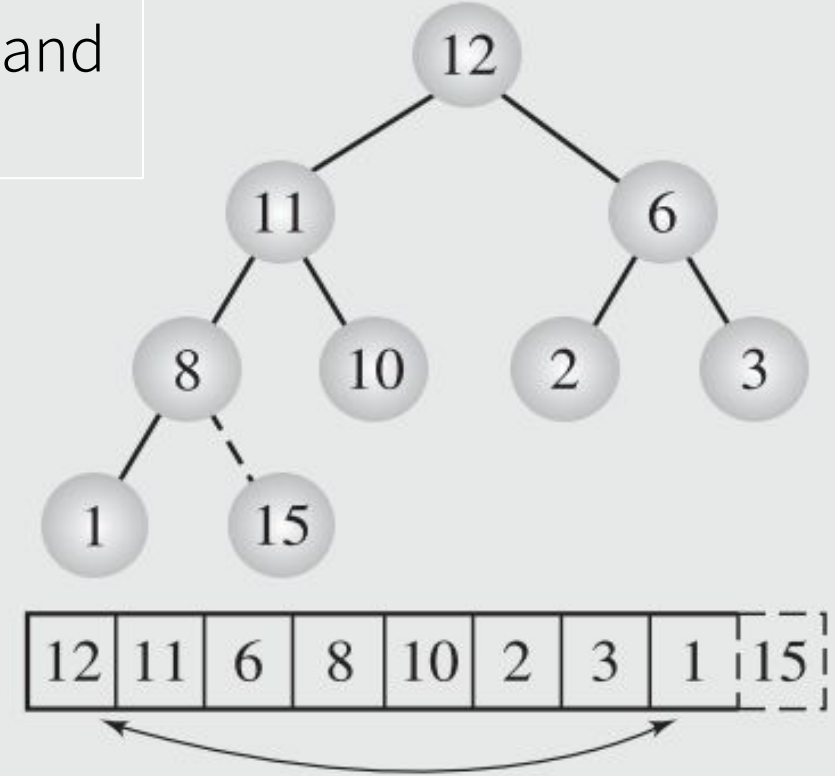
EXECUTION OF THE HEAP SORT

Last element (15) of the array is not the part of the heap any more.



EXECUTION OF THE HEAP SORT

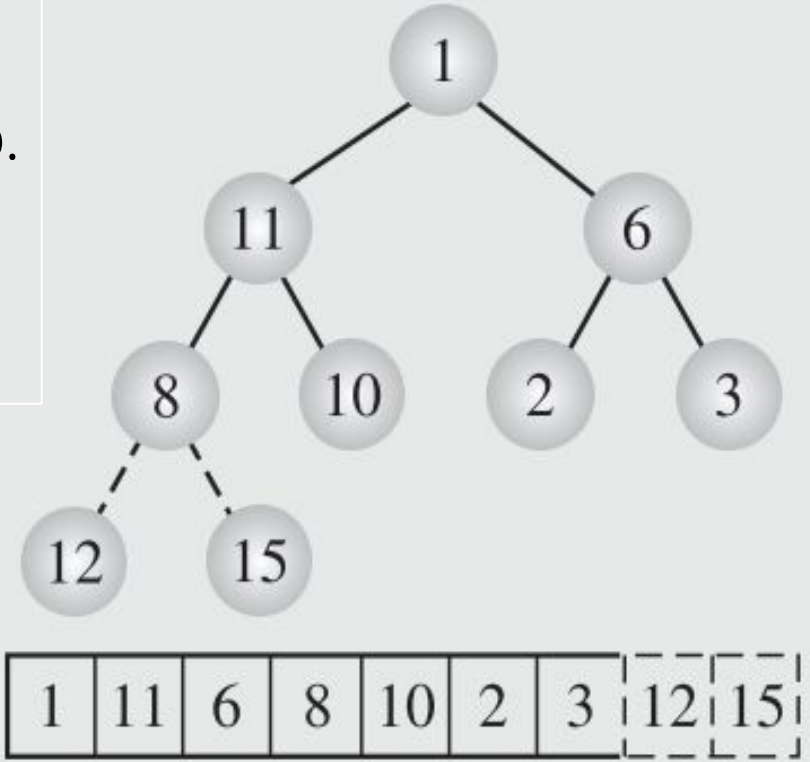
Repeat the Process for next root and last element of the heap.



EXECUTION OF THE HEAP SORT

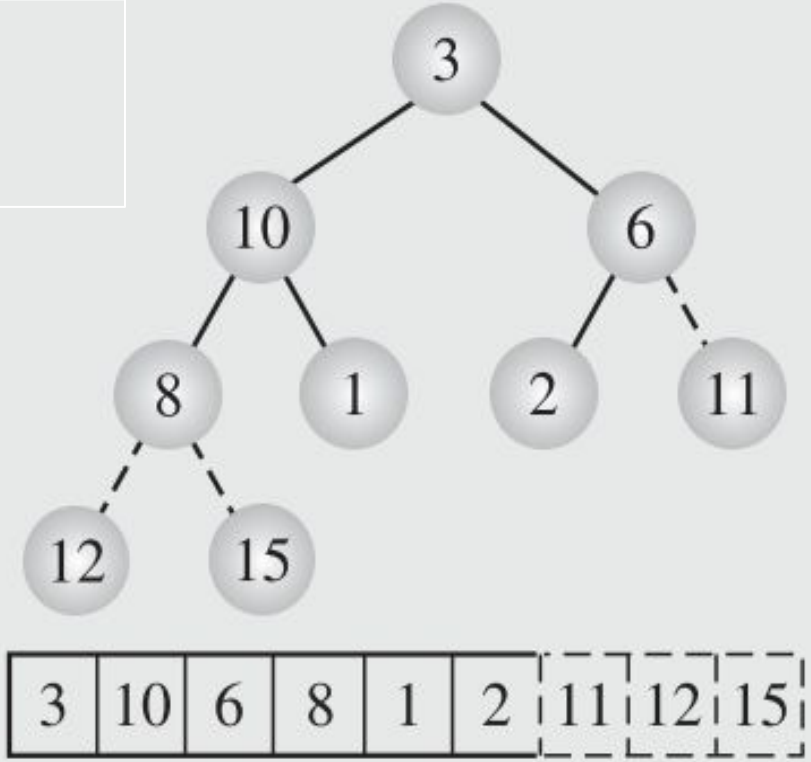
Now last two elements of the array are not the part of the heap.

Both are sorted with respect to each other.



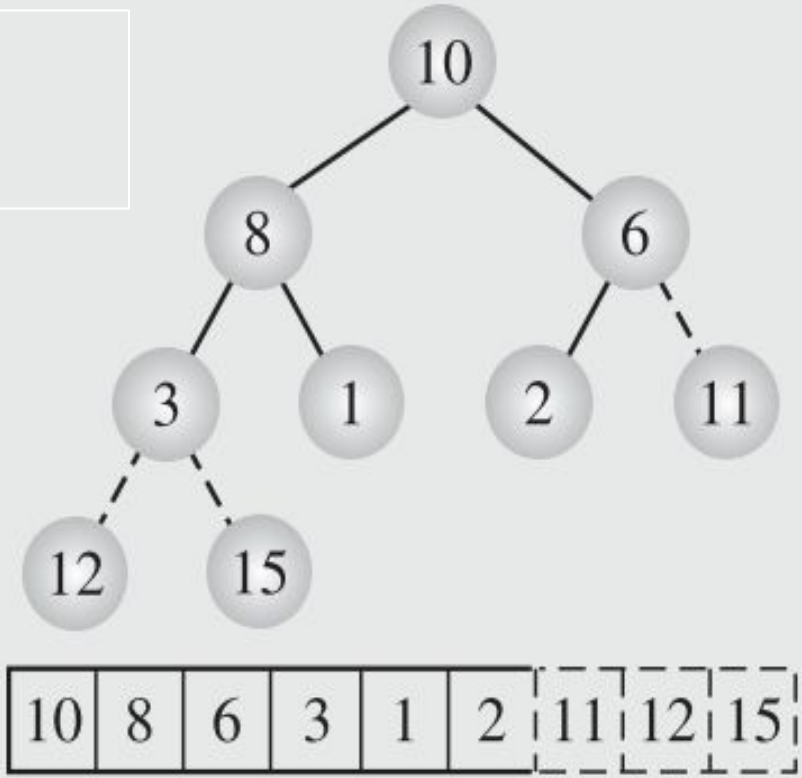
EXECUTION OF THE HEAP SORT

11 is swapped with 3



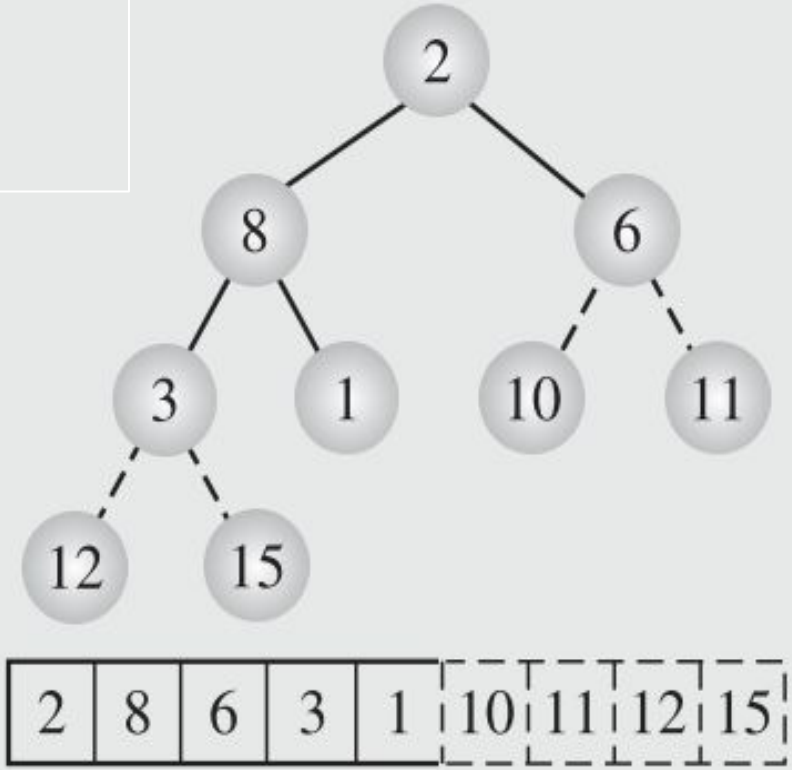
EXECUTION OF THE HEAP SORT

Heap Property restored.
11, 12 & 15 are sorted.



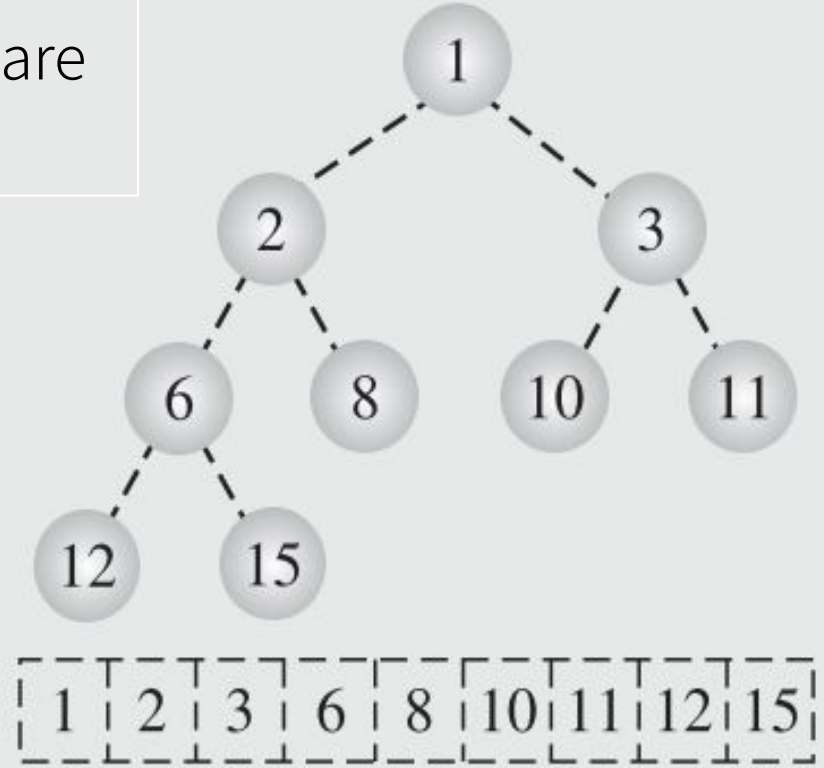
EXECUTION OF THE HEAP SORT

10 is swapped with 2, the last element of the heap.



EXECUTION OF THE HEAP SORT

Heap is empty and all elements are sorted in the array.



TIME COMPLEXITY

- Creating the Heap in *O (n).*
- Heap sort exchanges the root n - 1 times with the element at position heap.size( ).
  - Heap is restored n - 1 times as well.
  - In worst cases bringing root down to the level of leaves heapsort requires log n.
  - All the execution in this phase cost *O (n log n).*

TIME COMPLEXITY

- **In the worst case scenario Heap sort requires**
  - O (n) steps in first phase (making the heap).
  - n - 1 swaps and O (n log n) operations to restore the heap property.
- **This gives us**
  - O (n) + O (n log n) + (n - 1)
  - Hence, O (n log n) exchanges for the whole process in the worst case.

TIME COMPLEXITY

- **In the best case scenario** (when array contains identical elements)
  - Heapify process is called n / 2 times in the first phase
  - In the second phase heap sort makes one swap to move the root element to the end of the array,
    - costing only n - 1
    - Hence in best case heapsort gives O (n).
- When elements are distinct
  - number of comparisons will equal n log n - O (n).

TIME COMPLEXITY

- Best
  - *O (n log n)*
- Average
  - *O (n log n)*
- Worst
  - *O (n log n)*
- Space Complexity
  - *O (1)*
- Stability
  - No

BINARY SEARCH

Recall Binary Search

- We divide the array into two parts.
- Compare mid if equals return mid
- If key is less than mid recur on first interval
- If key is greater than mid recur on second interval.

TERNARY SEARCH

Similar to Binary Search.

- Instead we divide the array into 3 parts.
  - Now we will have two midpoints

```
mid1 = left + ( right - left ) / 3;

mid2 = right - (right - left ) / 3;
```

TERNARY SEARCH ALGORITHM

1. Compare the key with mid1, if found return mid1.
2. If not, then Compare the key with mid2, if found return mid2.
3. If not, check whether the key is less than mid1
   a. If true, recur on first interval using low & mid1.
4. If not, check whether the key is greater than the mid2
   a. If true, recur on the last interval using mid2 & right.
5. If not, then recur to the middle part using mid1 & mid2.