Building A Heap – A Naïve Algorithm

Building A Heap – A Naïve Algorithm

- Take *N* items from an array and build a heap.

- Naïve algorithm
  - For each item, insert it into a heap, which is initially empty

```
public void buildHeap(T[ ] array, int size )
{
    < initialize the heap here …>
    for( int i = 0; i < size; i++ )
        insert(array[i]);
}
```

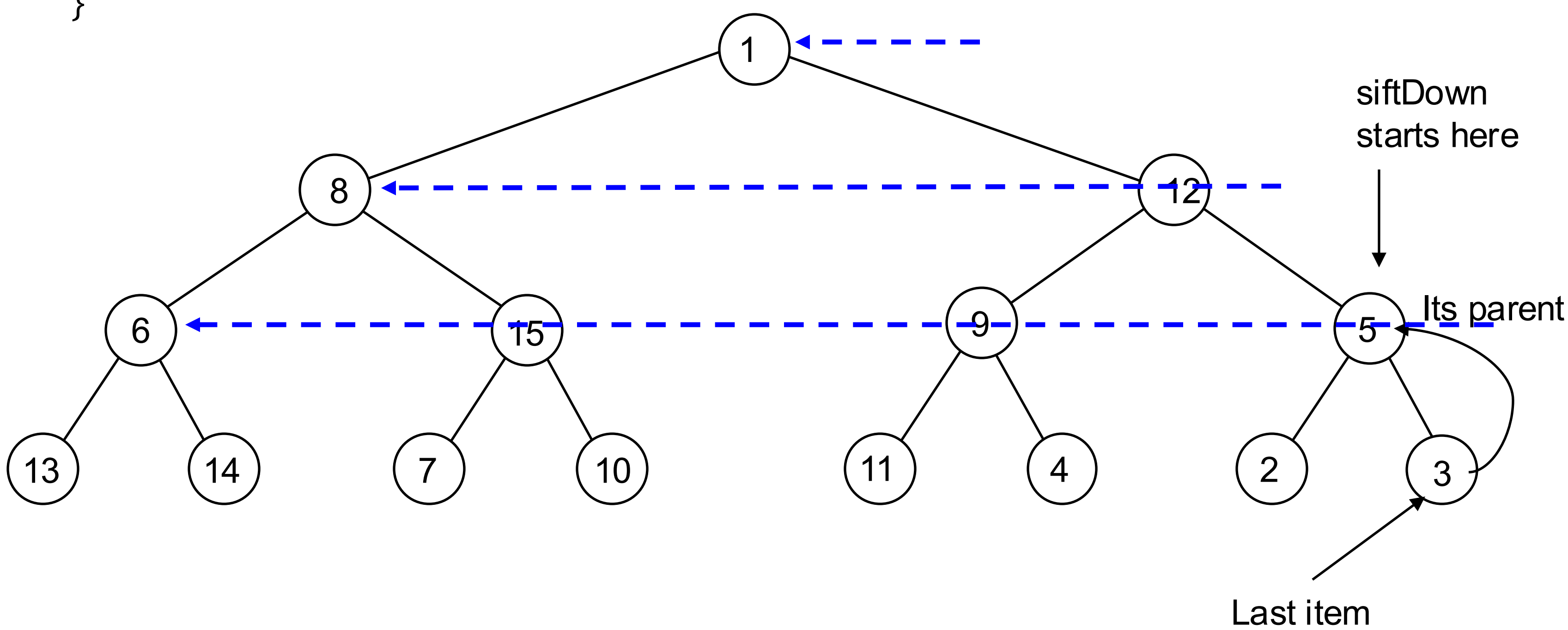
  - Running time: O(log(1) + log(2) + … + log(N)), or O(NlogN)

An Efficient Algorithm: Build the Heap in Place

1. Position = (numItems-2)/2; // initial position – the parent of the last item
2. siftDown item at that position.
3. Decrement position by one.
4. Repeat steps 2,3,4 until position is 0.

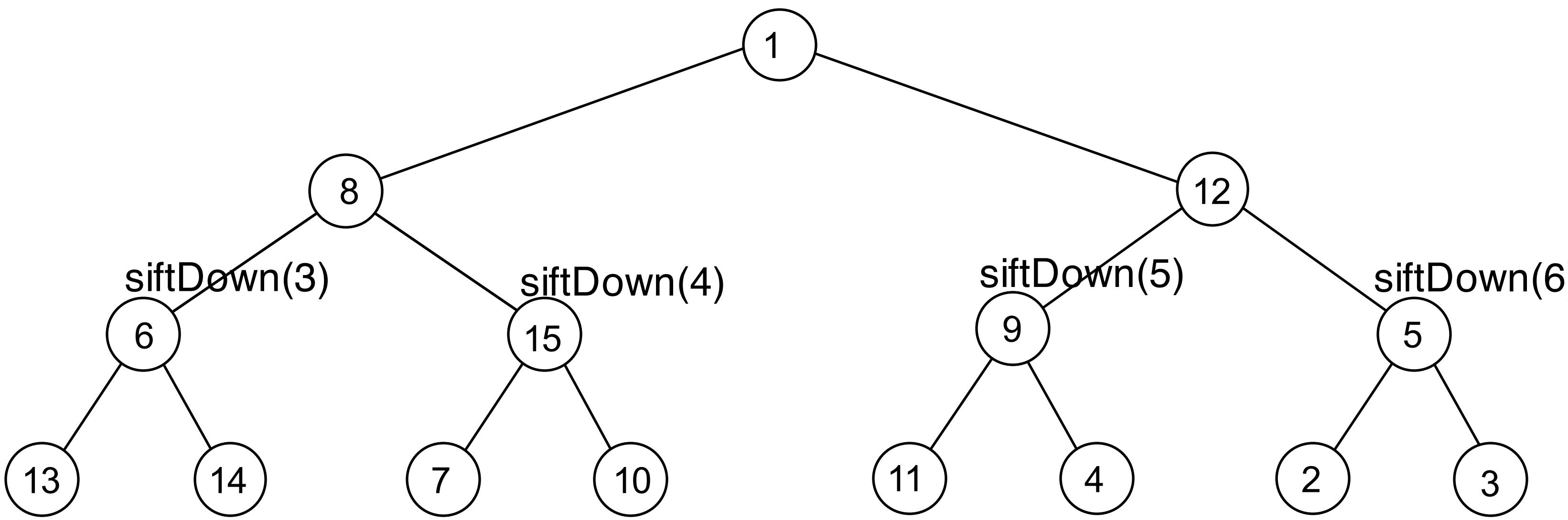
1	8	12	6	15	9	5	13	14	7	10	11	4	2	3

```
public void buildHeap( )
{
    for( int i = (numItems -2)/2; i >= 0; i-- )
        siftDown( i );
}
```



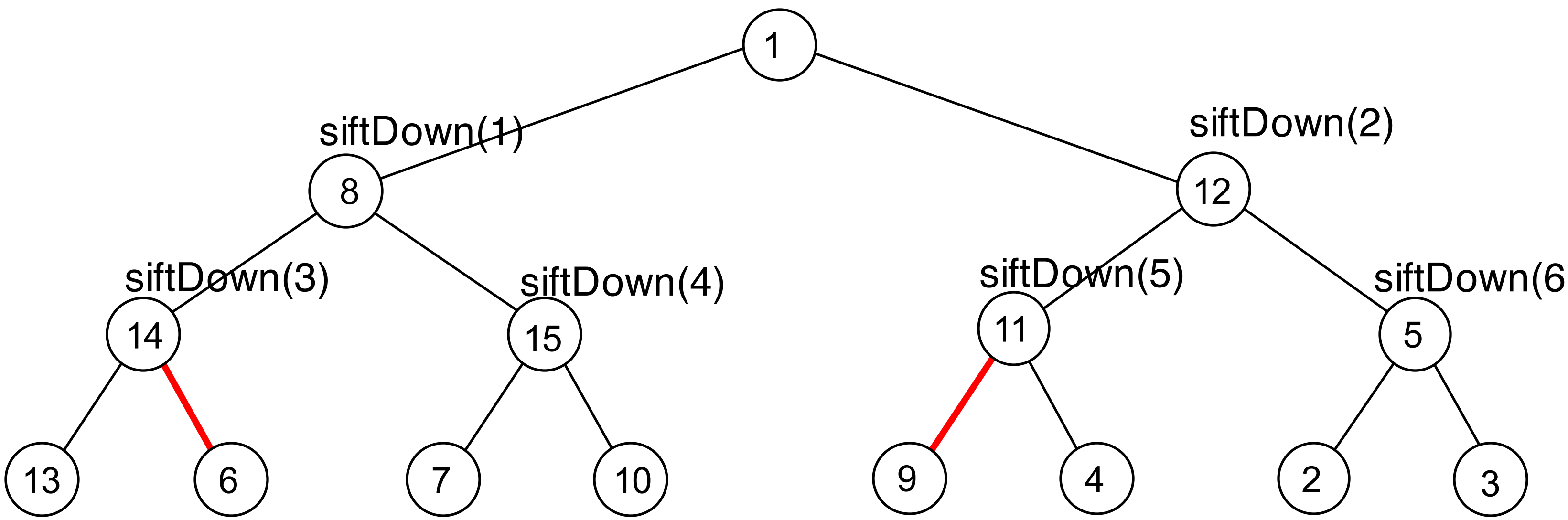
An Efficient Algorithm

```
for( int i = (numItems -2) / 2; i >= 0; i-- )
    siftDown( i );
```



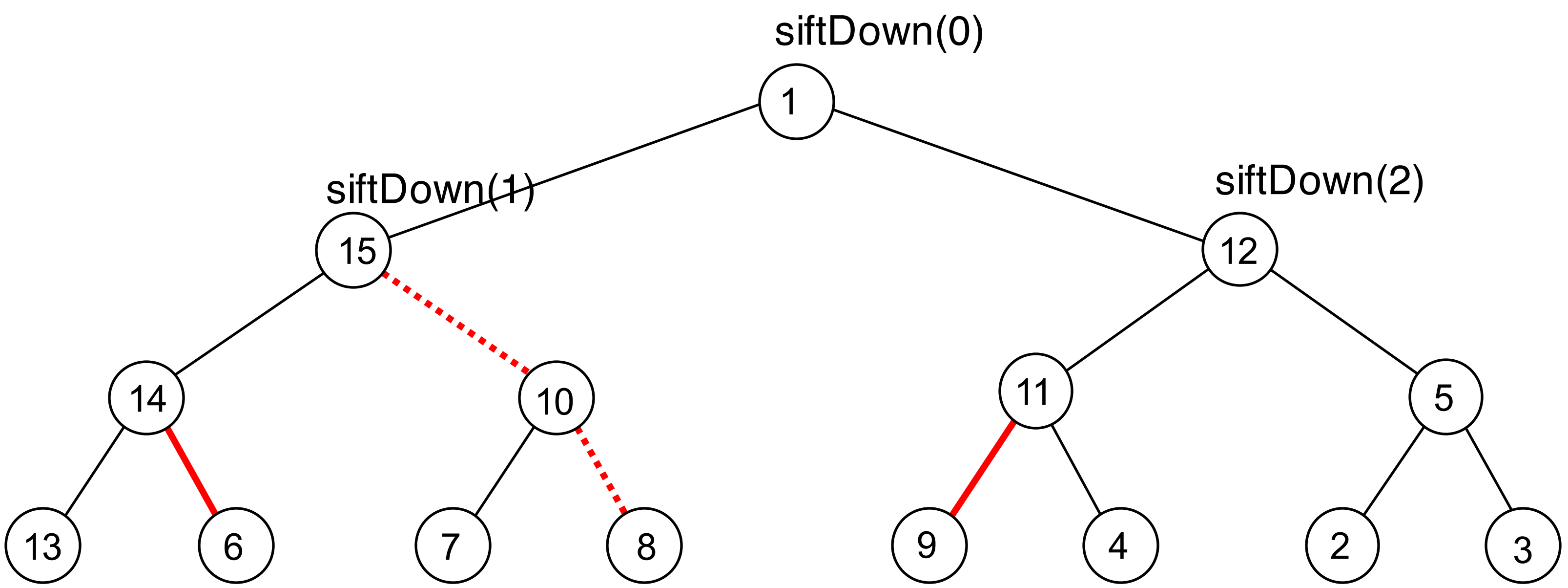
An Efficient Algorithm

```
for( int i = (numItems -2) / 2; i >= 0; i-- )
    siftDown( i );
```



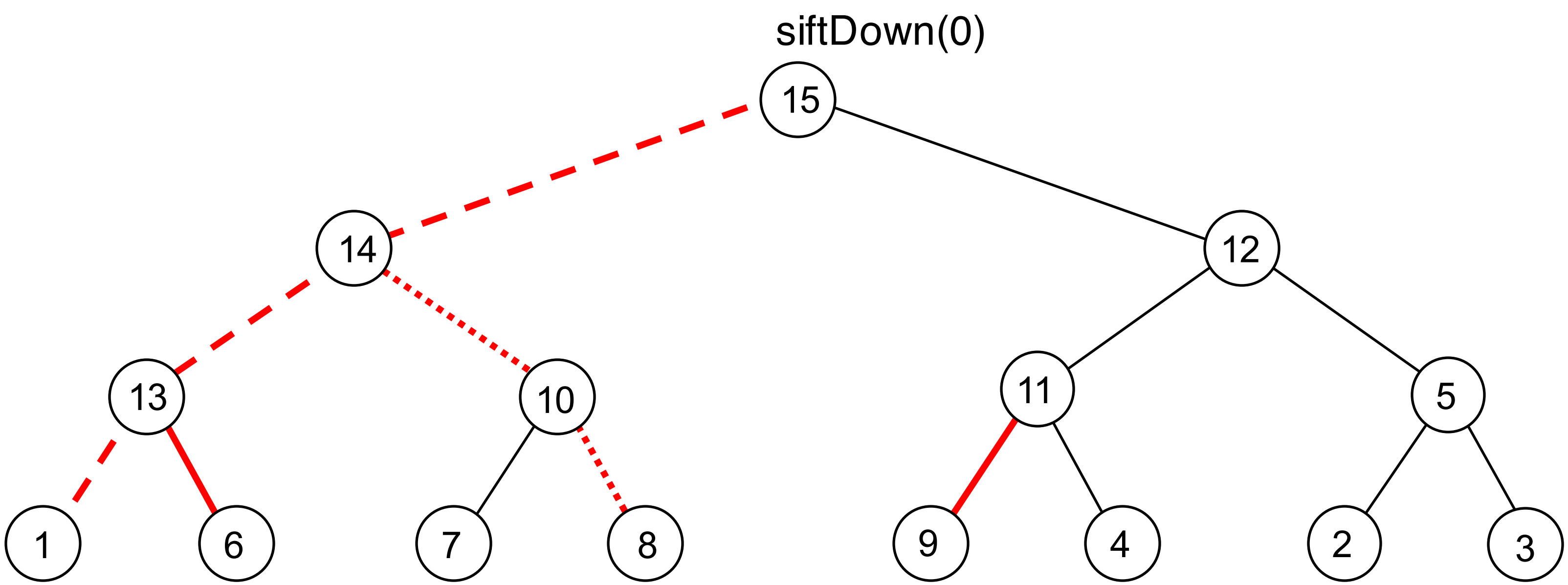
An Efficient Algorithm

```
for( int i = numItems / 2 - 1; i >= 0; i-- )
    siftDown( i );
```



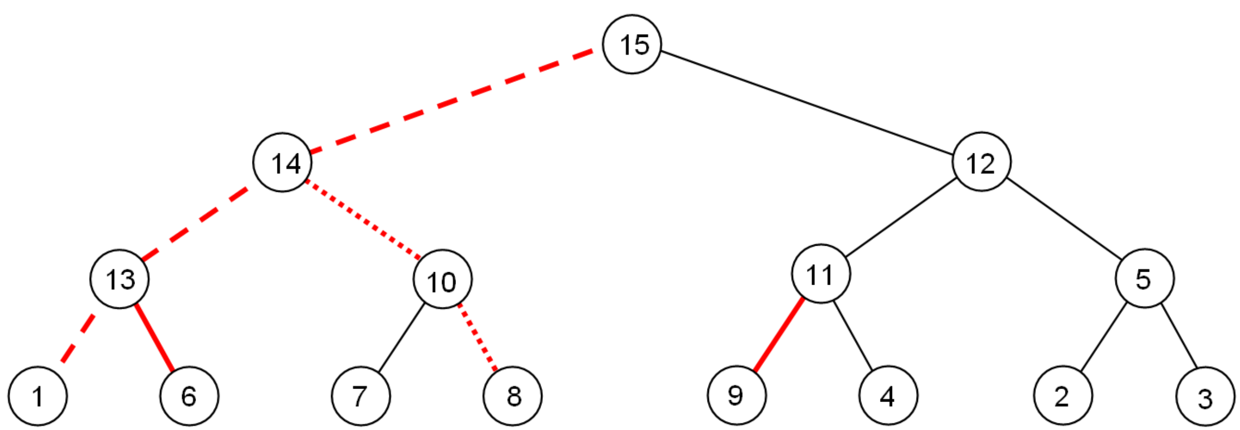
An Efficient Algorithm

```
for( int i = numItems / 2 - 1; i >= 0; i-- )
        siftDown( i );
```



Running Time Analysis

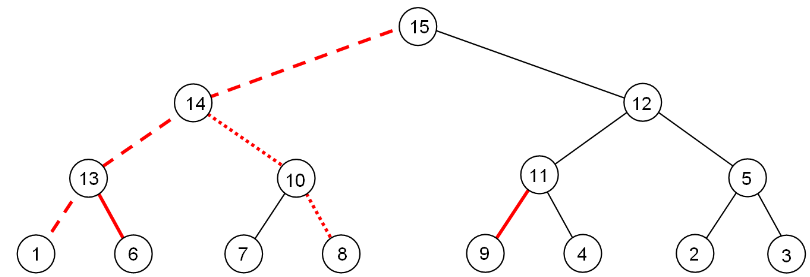
- O(NlogN) as siftDown() is called N/2 times, and each takes no more than O(logN) time.
- However, this running time is not tight. The cost of BuildHeap is bounded by the number of red lines (all red lines)
  - …which is at most the sum of heights of all nodes of the heap.
- This sum is O(*N*), where *N* is the number of nodes in the heap.



Running Time Analysis

- For a perfect binary tree of height h, $N = 2^{h+1}-1$:

1 node at height h
2 nodes at height h-1
4 nodes at height h-2
…
2^{h-1} nodes at height 1
2^h nodes at height 0



Total height of all nodes $= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(h-(h-1)) = S$

$$2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^h(h-(h-1)) = 2S$$

$$S = 2h + 4h - 4 + 8h - 16 + 16h - 48 + \ldots + 2^h(h-(h-1))$$

$$- \; (h + 2h - 2 + 4h - 8 + 8h - 24 + \ldots + 2^{h-1}h - 2^{h-1}(h-1))$$

- Although a complete tree is not a perfect binary tree, but number of nodes in a complete tree of height h is:

$$2^h \le N_{actual} \le 2^{h+1} - 1$$

- Thus the actual number of nodes is within a factor of 2 of N. Hence, $S = O(N_{actual})$

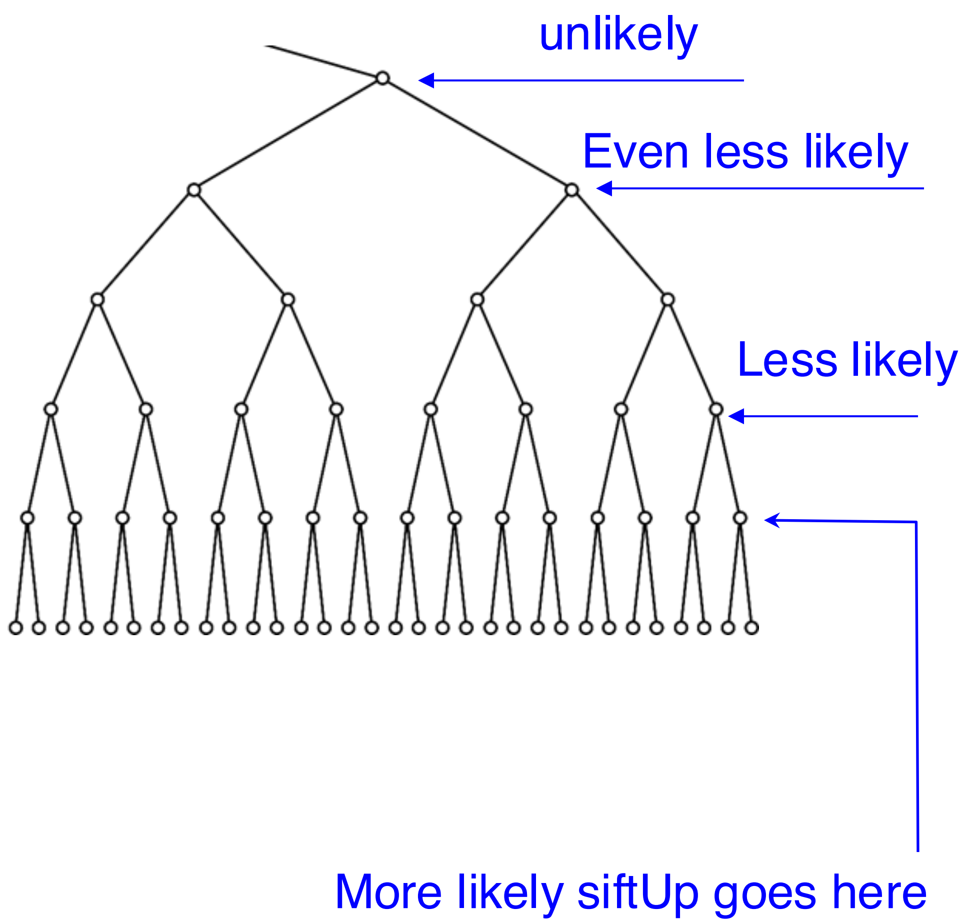
Merge Two Heaps

- Merge two heaps into a single one.
- Append one heap to the end of the other, and then just like buildHeap, sift down the interior nodes
  - Running time = O(?)

Running Time of Binary Heap Operations

- Summary of the worst-case running time of binary heap operations (max-at-top)

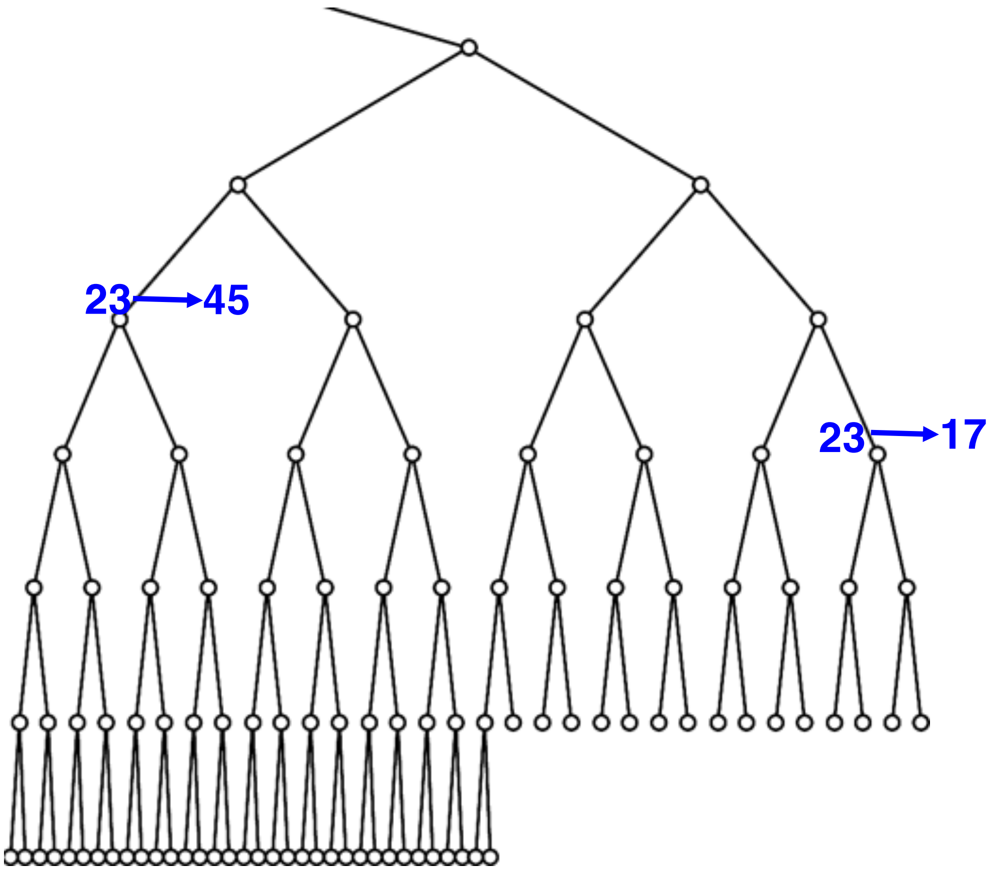
findMax	O(1)
removeMax()	O(logN)
insert()	O(logN)
buildHeap()	O(N)
merge()	O(N)



Exercise: Update An Item

- Update an item and return the old one

```
public T update(int i, T item) {
    T oldItem = items[i];
?       items[i] = item;
        if (item.compareTo(oldItem) == 1)
            siftUp(i);
        else
            siftDown(i);
    return oldItem;
}
```



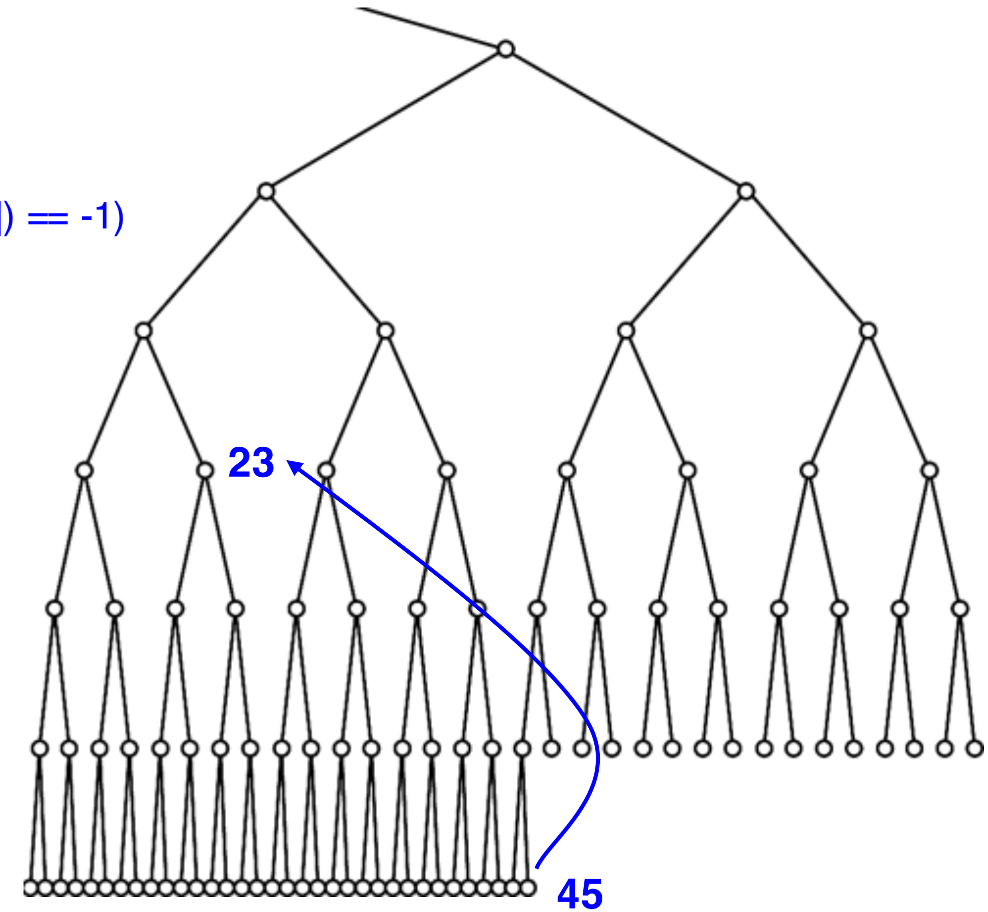
- Running time?

Delete An Arbitrary Item (Not RemoveMax())

```
public T delete(int i) {
    T toDelete = items[i];

        items[i] = item[numItems - 1];
        numItems--;
?       if (toDelete.compareTo(items[i]) == -1)
            siftUp(i);
        else
            siftDown(i);

    return toDelete;
}
```



- Running time?

Applications of Heaps

EECS 233

Application #1: Heap-sort

- Sorting algorithms: given an arbitrary array, re-position the items in the array in ascending or descending order

15	7	13	5	4	6	11	16	9

- A naïve algorithm (we will learn more later)

4	7	13	5	15	6	11	16	9

4	**5**	13	7	15	6	11	16	9

4	**5**	**6**	7	15	13	11	16	9

- It isn't efficient ($O(n^2)$), because it performs a linear scan to find the smallest remaining item ($O(n)$ steps per scan).

Heap-Sort

- Heap-sort is a sorting algorithm that repeatedly finds the *largest* remaining item from a heap and puts it in place.

```
void heapSort(T[] arr, int numItems) {
        // Turn the array into a max-at-top heap.
        buildHeap(arr, numItems);

        int endUnsorted = numItems - 1;
        while (endUnsorted > 0) {
                // Get the largest remaining item and put it to the end
                T largestRemaining = removeMax(arr);
                arr[endUnsorted] = largestRemaining;
                endUnsorted--;
        }
}
```

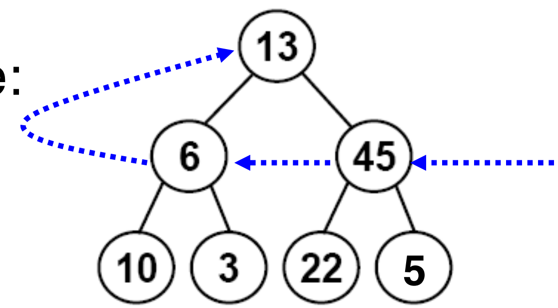
- It *is* efficient:
  - O(N) to turn the array into a heap
  - O(log N) to remove the largest remaining item
  - N above removals
  - *O*(N log N) overall

Heap-Sort Example: (1) Build the Heap

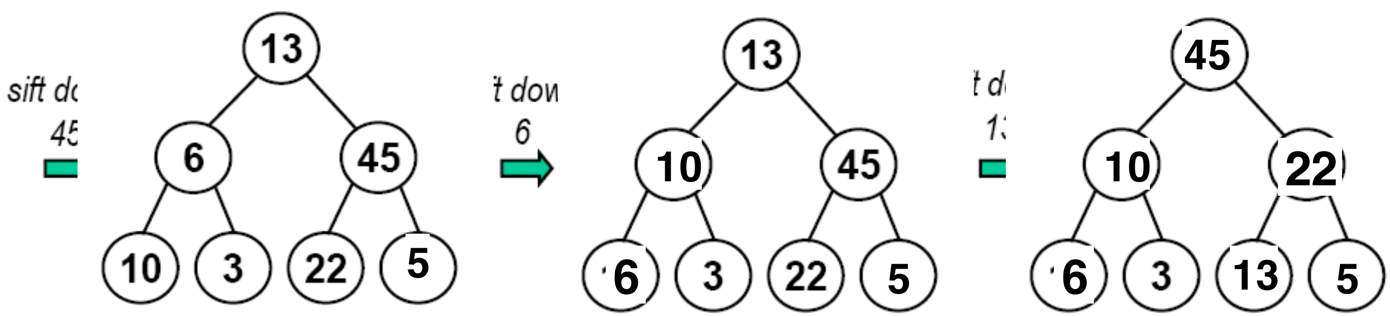
- Sort this array:

13	6	45	10	3	22	5

- The corresponding complete tree:

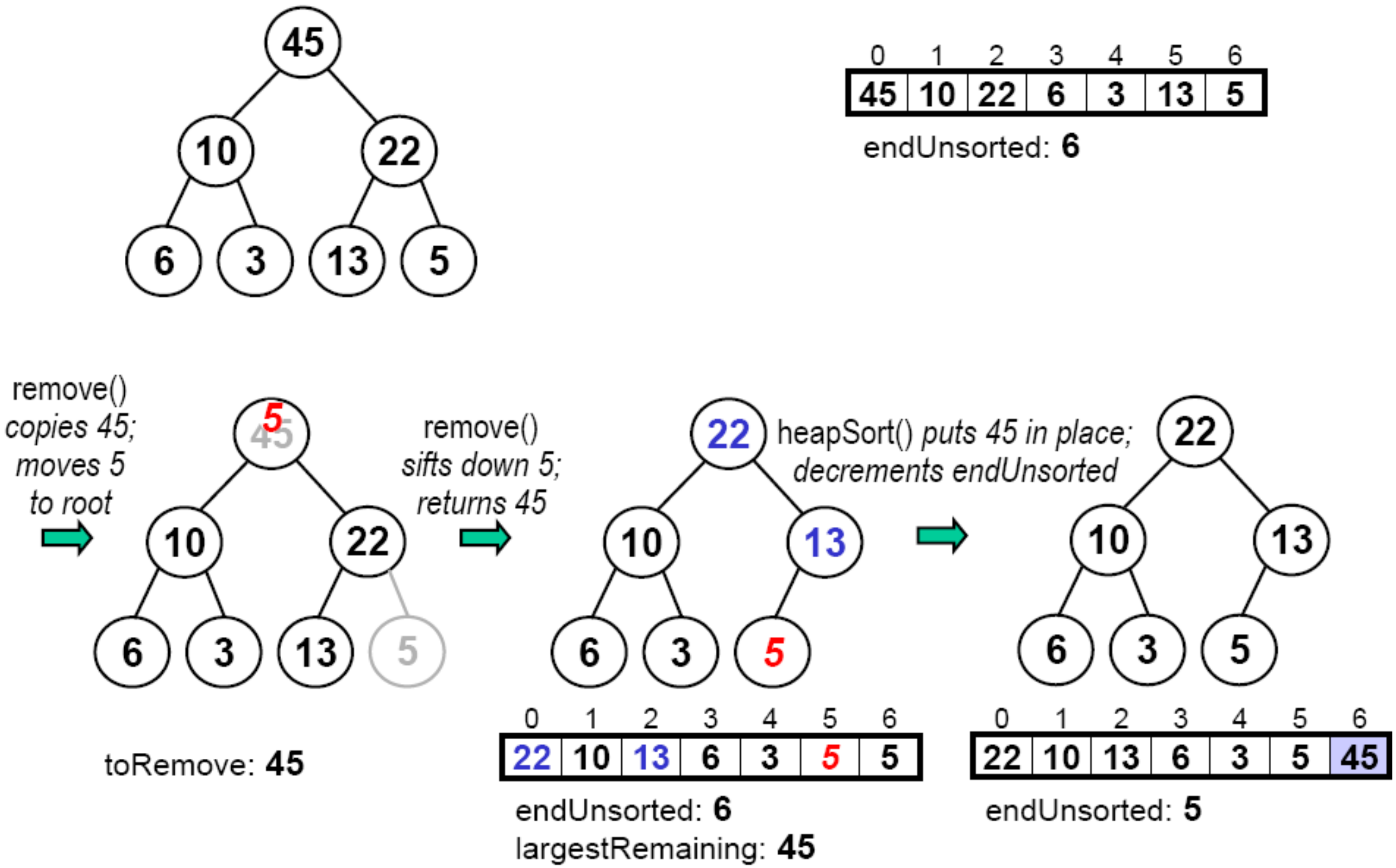


- First call BuildHeap():

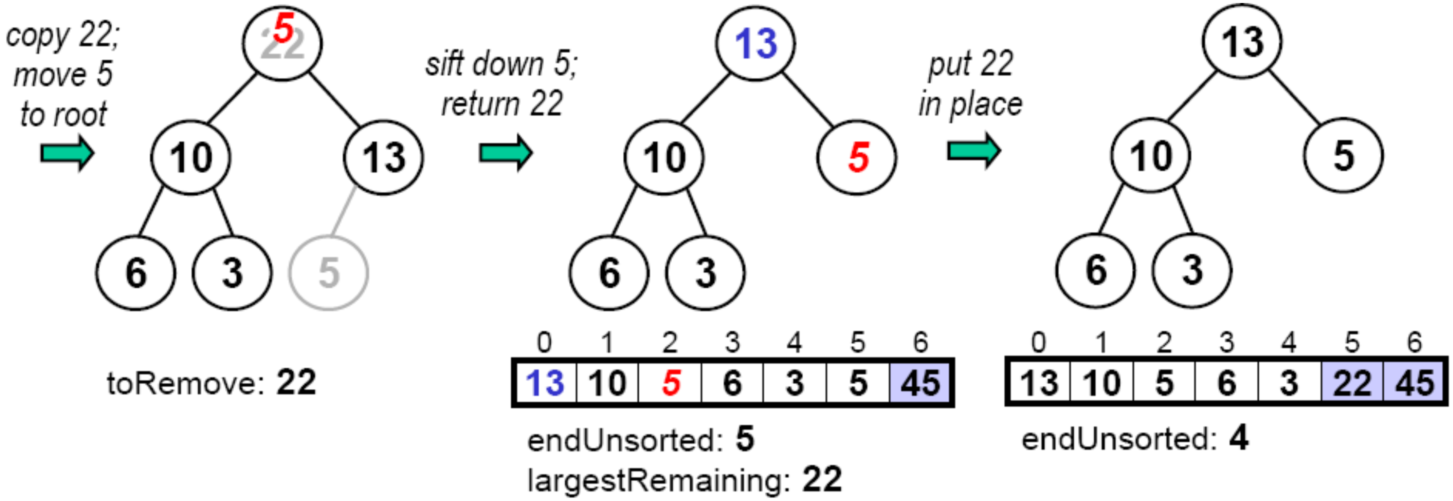


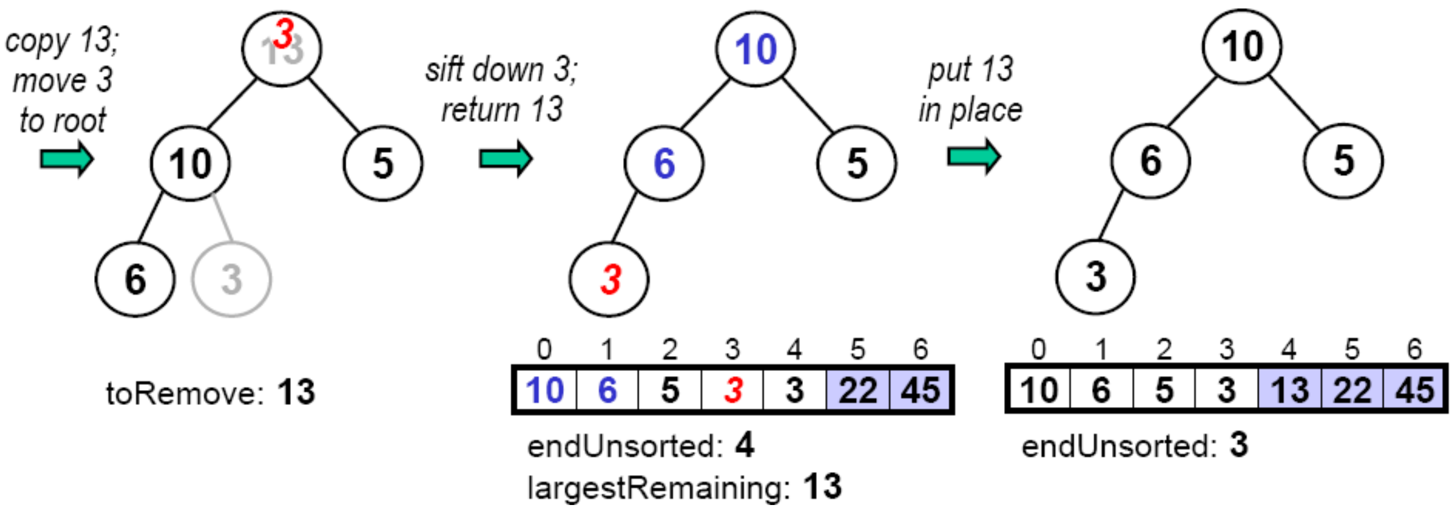
Heap-Sort Example: (2) Drain the Heap

- Remove the largest item from heap:

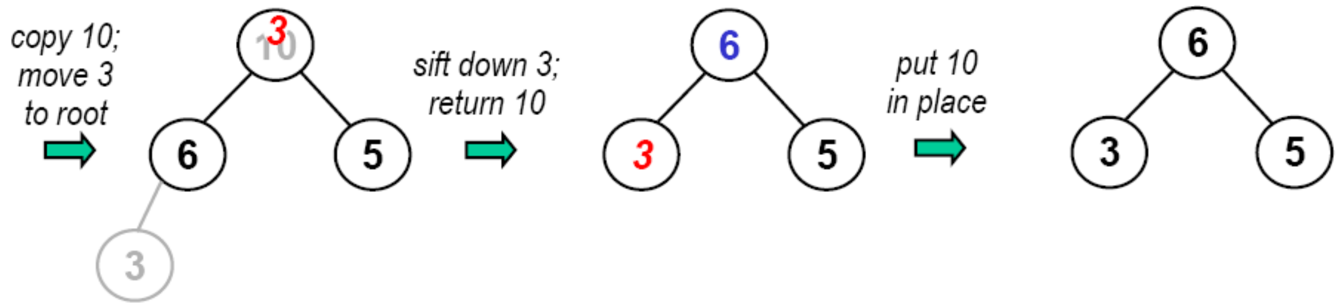


Remove the Next Largest Items





Keep Going



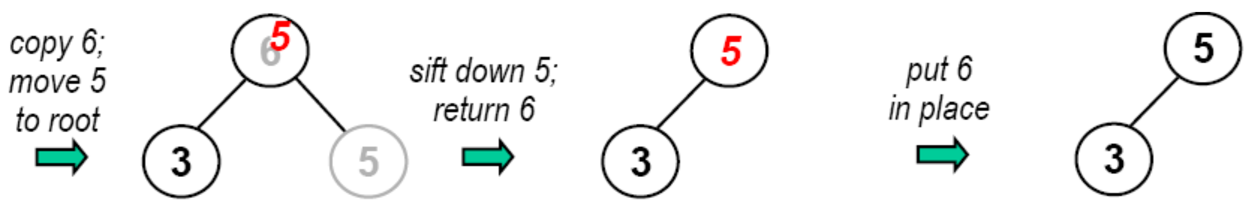
toRemove: **10**

0	1	2	3	4	5	6
6	*3*	5	3	13	22	45

endUnsorted: **3**
largestRemaining: **10**

0	1	2	3	4	5	6
6	3	5	10	13	22	45

endUnsorted: **2**



toRemove: **6**

0	1	2	3	4	5	6
5	3	5	10	13	22	45

endUnsorted: **2**
largestRemaining: **6**

0	1	2	3	4	5	6
5	3	6	10	13	22	45

endUnsorted: **1**

Application #2: The Selection Problem

- Given a list of N items and an integer k (1 <= k <= N), find out the k-th largest item (or k largest items) in the list. E.g., N=1,000,000 and k=100, or k=N/2

- Example:
  - List of items: 4, 9, 0, 3, 5, 7, 10, 12, 2, 8

    12 10 9 8 7 5 4 3 2 0 (sorted order of items)

  - 1st largest item is: 12
  - 10th largest item is: 0
  - 6th largest item is: 5

- What is your method?

The Selection Problem: Naïve Methods

- Naïve method 1
  - Sort N items: $O(N^2)$ (for a simple sort algorithm), O(NlogN) for heap-sort (and a number of other algorithms we will learn)
  - Retrieve the k-th largest item: O(1)

- No so naïve method 2
  - Read k items into an array A: O(k)
  - Sort the items in the array A: $O(k^2)$ (actually O(k log k) but no matter)
  - For each of the remaining items: (N-k) of them (*new_item*)
    - Compare *new_item* to the last (smallest) item in A, *last_item*. If *new_item* is larger than *last_item*, replace *last_item* with *new_item* and put the latter into correct spot in the array: O(k)
  - The final array contains the top-k items

  Total Running time: $O(k + k^2 + (N-k) * k) = O(Nk)$.

The Selection Problem: Efficient Method 1

- Use heaps!

- To find the k-th largest item.
  - Read N items into an array   O(?)
  - Apply buildHeap() to the array   O(?)
  - Perform k removeMax() operations. O(?)
    - Last operation will give us the k-th largest one.

- What is the total running time? O(?)

O(N + klogN)

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest item.

- To find the k-th largest item.
  - Read k items into a min-on-top heap S (of size k). O(?)
  - For each remaining item (N-k) of them
    - Compare it with the smallest item (root) in heap S
    - If item is larger than root, then put it into S instead of root.
    - Sift down the root if necessary. O(?)

- Running time: O(k+(N-k)logk) = O(?)
  - Compared to O(N + klogN) of Efficient Method 1
  - Which is better?