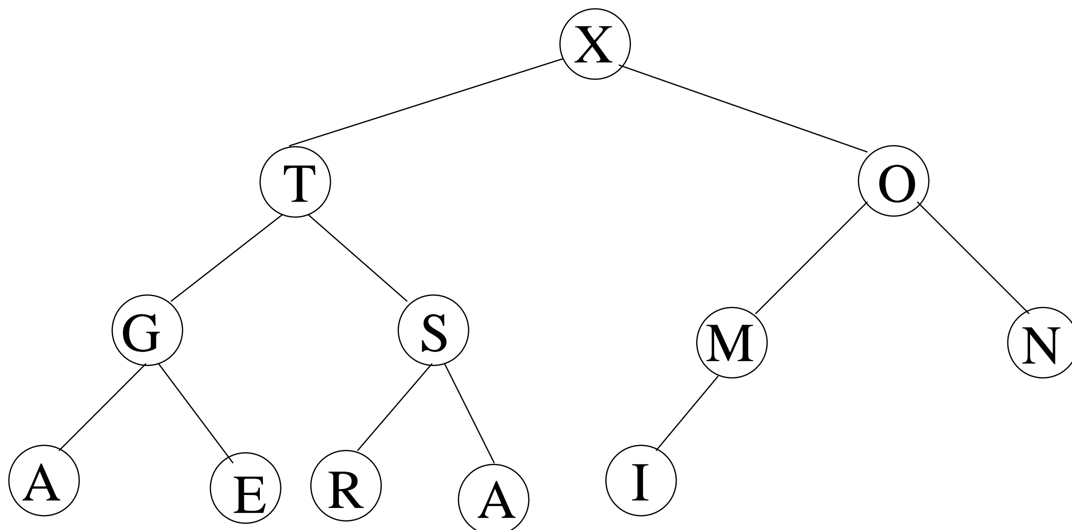
Heaps

- The next sorting algorithm we will talk about is **Heapsort**.
- Before we talk about **Heapsort**, we need to discuss the **heap** data structure.
- A **heap** is a complete binary tree such that for every node, $key(child) \leq key(parent)$
- Recall that a **complete binary tree** is full at every level except possibly the last, which is left-filled.
- **Example:**



Height of a Heap

- **Theorem:** A heap with n nodes has height $\Theta(\log n)$
- **Proof:**
  - Let n be the number of nodes of a heap of height h.
  - Since a binary tree of height h has at most $2^h - 1$ nodes, it is not too difficult to see that

$$2^{h-1} \leq n \leq 2^h - 1$$

  - Taking logs on both sides of the first inequality, we get

$$h - 1 \leq \log n.$$

  - Adding one and taking logs on both sides of the second inequality, we get

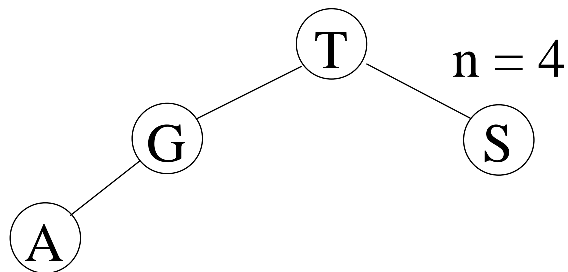
$$\log(n + 1) \leq h.$$

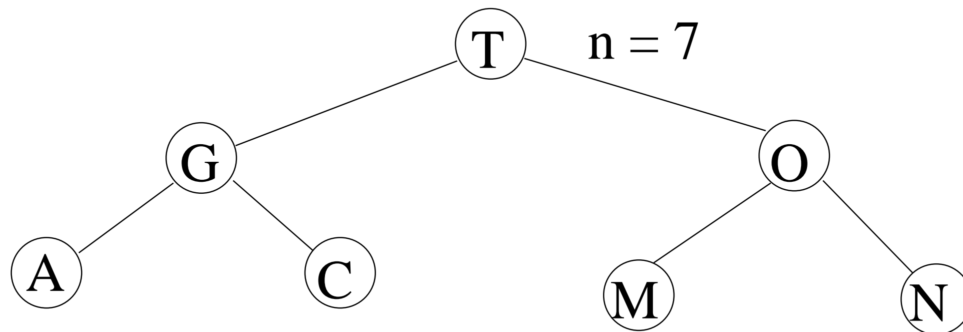
  - Thus,

$$\log(n + 1) \leq h \leq \log n + 1.$$

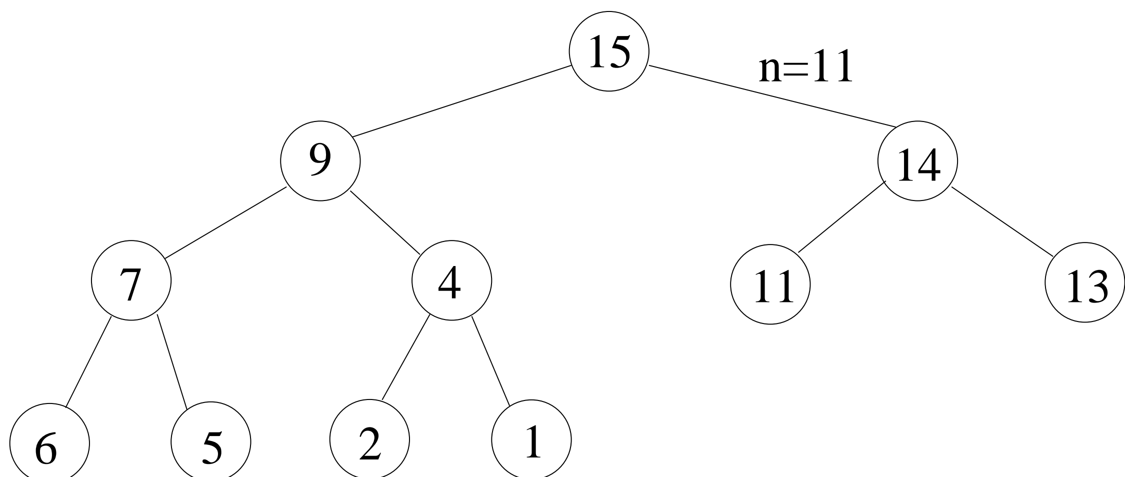
Heap Examples

- **Example:** Heaps with height $h = 3$



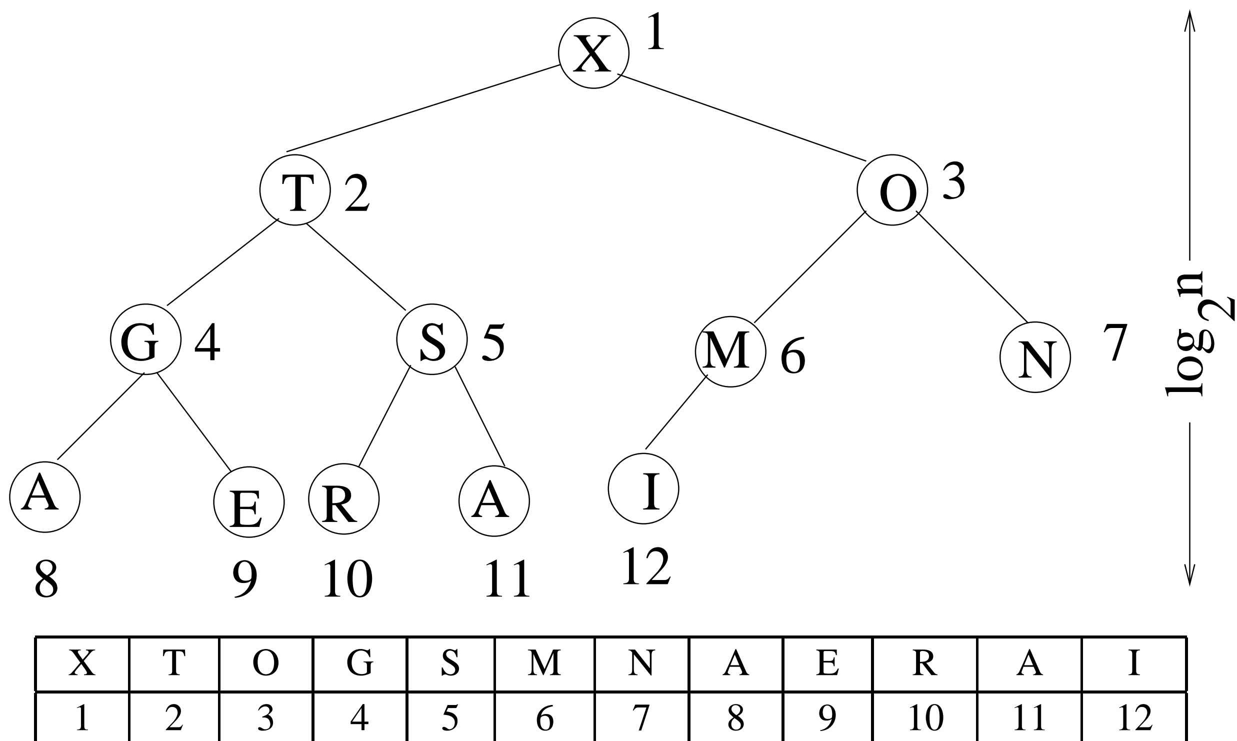


- **Example:** Heap with height $h = 4$



Heap Representation

- We have already seen how to represent binary trees, so we know how to represent heaps.
- Using arrays, we find parents/children as follows (Assuming we start indexing at 1):
  - $Left(i) = 2i$
  - $Right(i) = 2i + 1$
  - $Parent(i) = i/2$
- **Example:**



X	T	O	G	S	M	N	A	E	R	A	I
1	2	3	4	5	6	7	8	9	10	11	12

The Heapify Routine

- The **Heapify** routine is the basis of all other routines needed to use heaps.
- **Heapify** details:
  - **Input:** An array A and index i into the array.
  - **Assumption:** The subtrees rooted at Left(i) and Right(i) are heaps.
  - **Problem:** $A[i]$ may violate the heap property.
  - **Output:** The array A, where the tree rooted at i is a heap.
- It is not hard to see that **Heapify** runs in $O(\log n)$ time.
- Why is this useful? We will see.
- For now, just imagine that if we change the value of some key in the heap, it may violate the heap property, and thus must be fixed.

Heapify

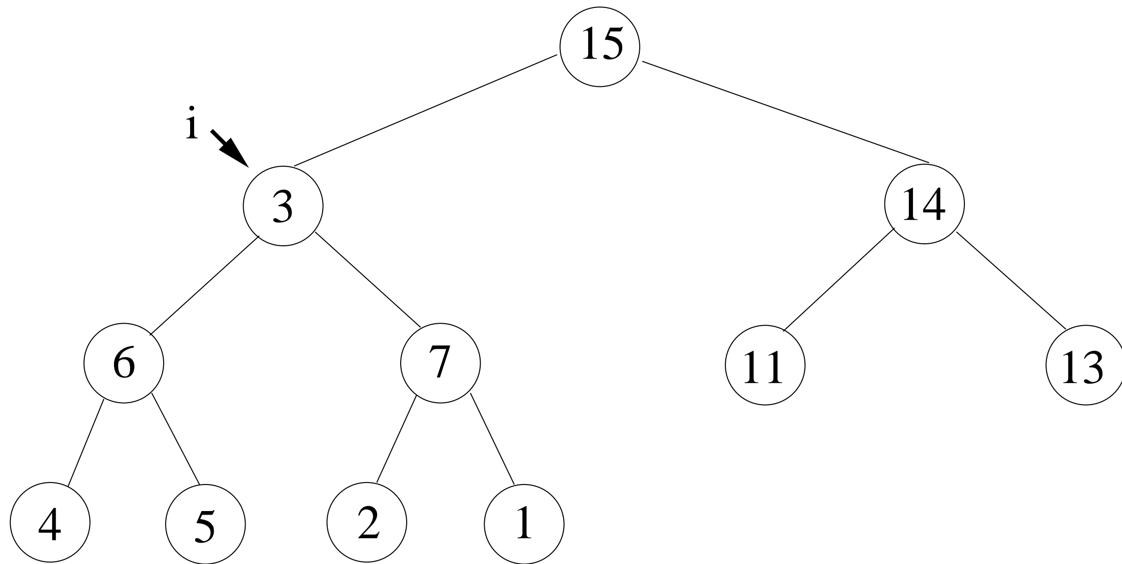
- Here is a C++ implementation of **Heapify**:

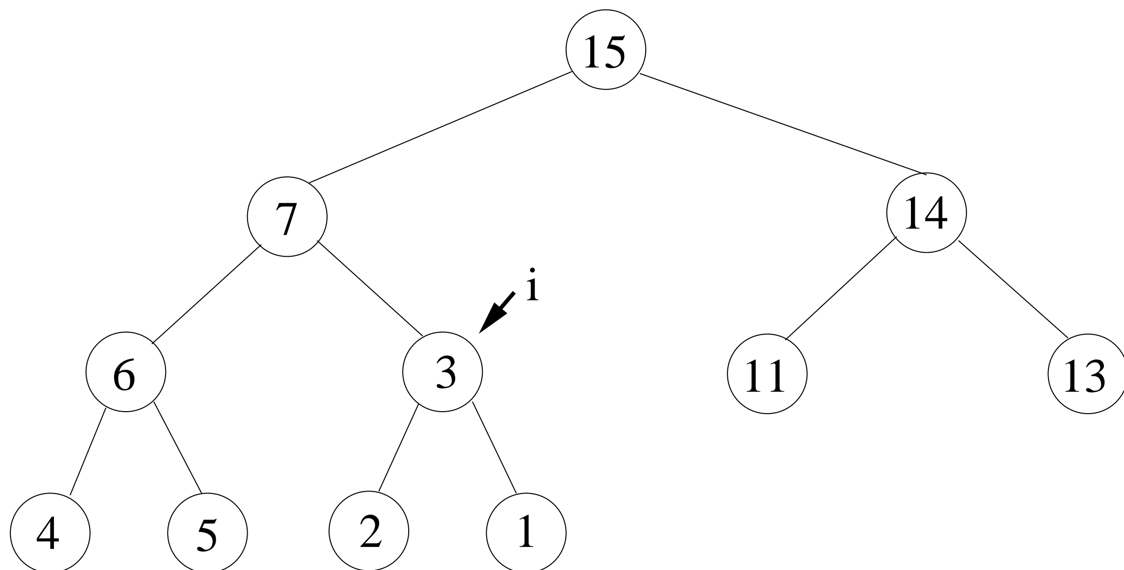
```
void Heapify(int *A,int i,int n) {
     l=Left(i);
     r=Right(i);
     if(l<= n && A[l] > A[i])
          largest=l;
     else largest=i;
     if(r<= n && A[r] > A[largest])
        largest=r;
     if(largest !=i) {
       Swap(A[i],A[largest])
       Heapify(A,largest,n);
       }
}
```

This is much simpler than it looks.
Heapify simply:

- Determines the largest of $A[i]$, $A[Left(i)]$, and $A[Right(i)]$
- If $A[i]$ is not the largest, then
  - Swap $A[i]$ with the $A[largest]$ (where $largest$ is either $Right(i)$ or $Left(i)$)
  - Calls heapify on node $largest$.

Heapify Example





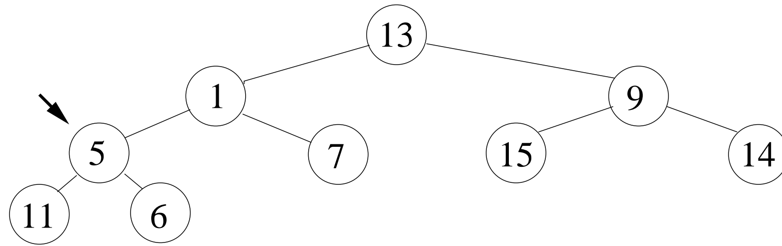
Build Heap

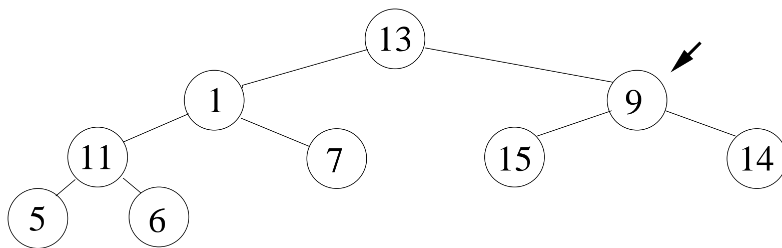
- The routine **BuildHeap** converts a regular array into a heap.
- Essentially, it runs **Heapify** on the nodes in reverse order.
- Since we are going in reverse order, we are know that the subtrees rooted at the children are heaps.
- The last half of the array corresponds to leaf nodes, so we don't need to heapify them.
- Here is the C++ code for Heapify:

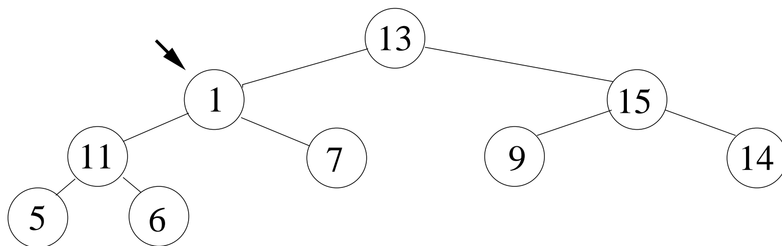
```
void BuildHeap(int *A,int n) {
     for(i=n/2;i>=1;i--)
        Heapify(A,i,n);
}
```

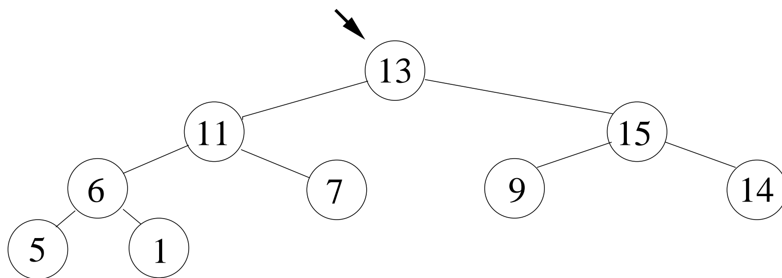
- It is clear that **BuildHeap** runs in $O(n \log n)$ time.
- In fact **BuildHeap** runs in $O(n)$ time.

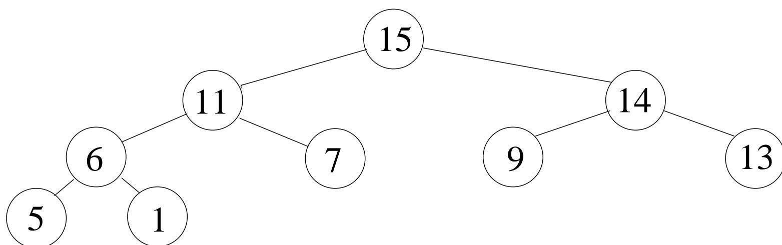
Build Heap Example











Other Heap Operations

- There are other operations one may wish to perform on a heap:
  - Insert()
  - Extract_Max()
- We won't discuss these here, but it is not too difficult to implement them given the **Heapify** procedure.
- With these operations, we can use a heap to implement a **priority queue**.
- A **priority queue** is a data structure which supports the operations *insert*, *maximum*, and *extract maximum*.
- We will use the priority queue data structure later.
- We are interested in the **Heapsort** algorithm, so that is what we will do next.

Heapsort

The idea behind **Heapsort** is simple.

- Run **BuildHeap** to turn array A into a heap.
- Since A is a heap, the largest element is $A[1]$.
- Swap $A[1]$ with $A[n]$. Now $A[n]$ is in place.
- Since $A[n]$ is in place, we can ignore it. Thus, we decrease the heap size by 1.
- Now $A[1]$ might violate the heap property, so we run **Heapify** on the (now smaller) heap.
- We repeat until only one item is left in the heap.
- Our array is sorted.

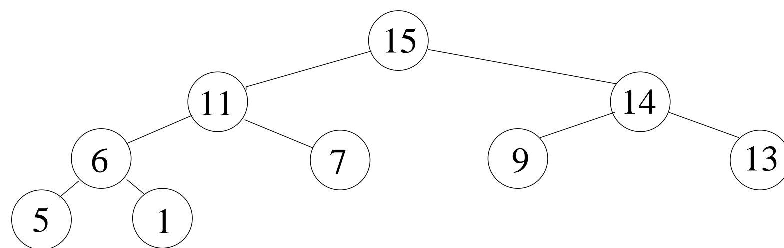
The C++ implementation:

```
void Heapsort(int *A, int n) {
     BuildHeap(A,n);
     for(int i=n;i>1;i--) {
          Swap(A[1],A[i]);
          Heapify(A,1,i-1);
          }
}
```

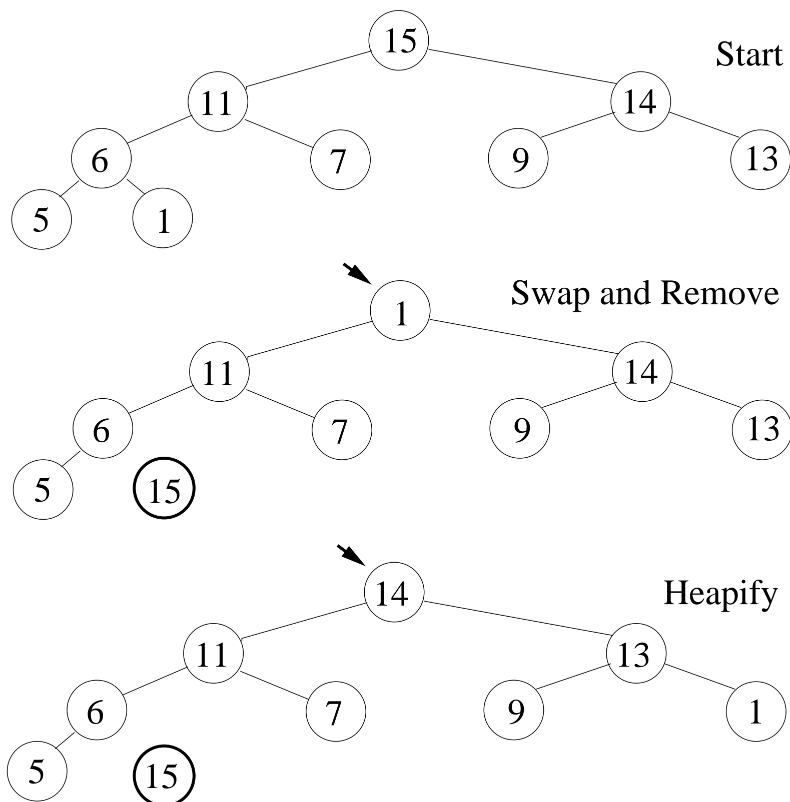
Heapsort Example

- Let $A = [13, 1, 9, 5, 7, 15, 14, 11, 6]$.
- Heapify will produce

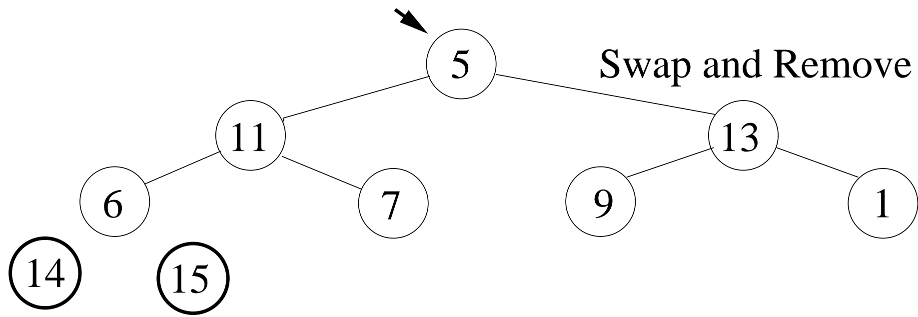
$$A = [15, 11, 14, 6, 7, 9, 13, 5, 1].$$

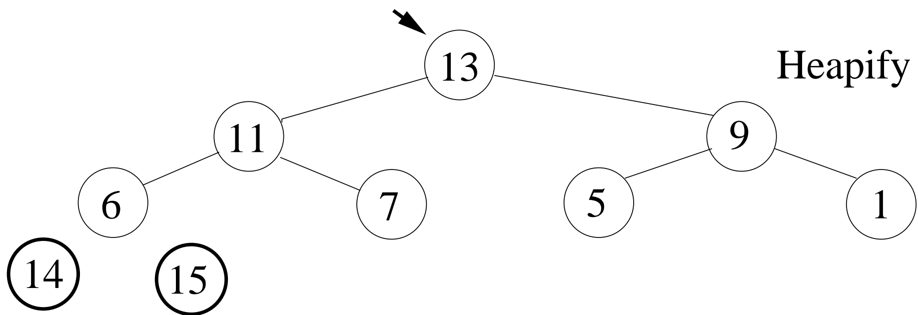


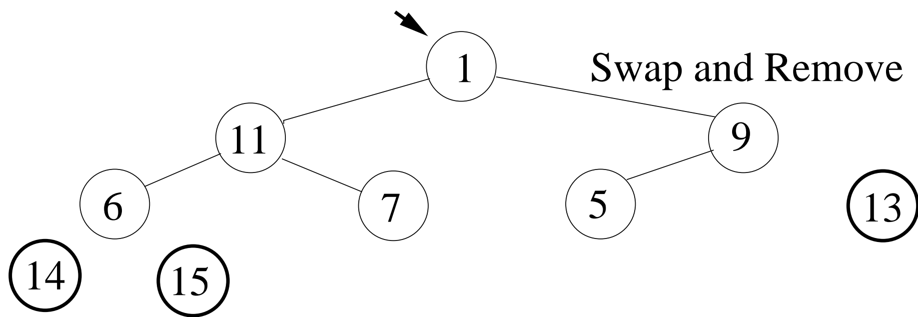
- Now, swap, "remove", and heapify until done:

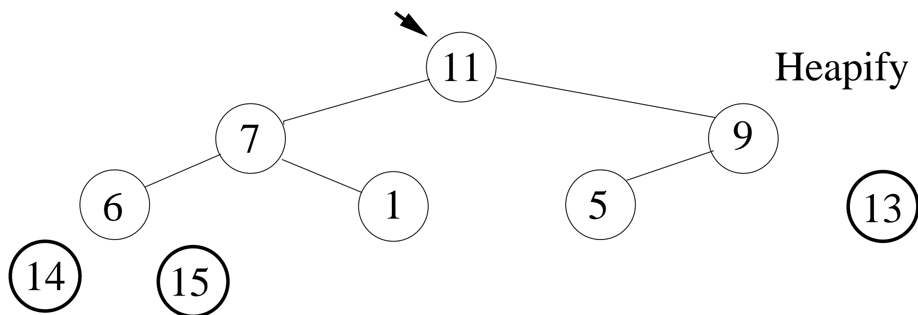


Heapsort Example Continued

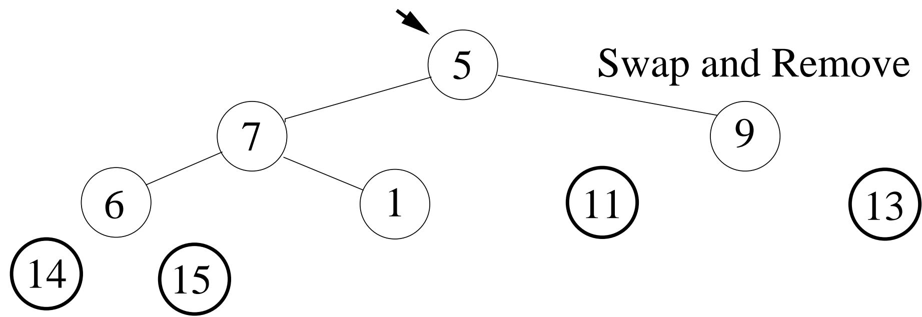


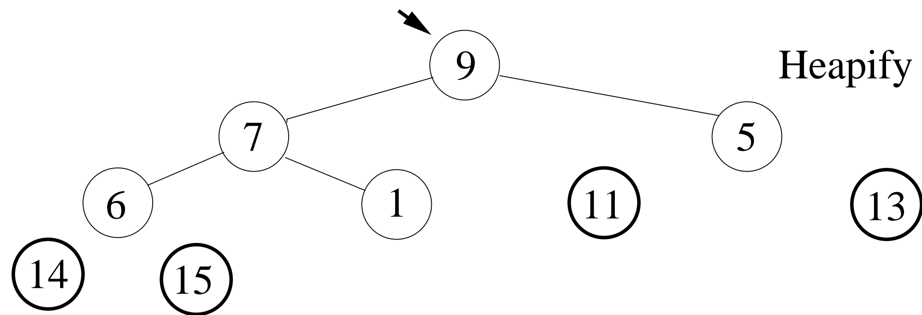


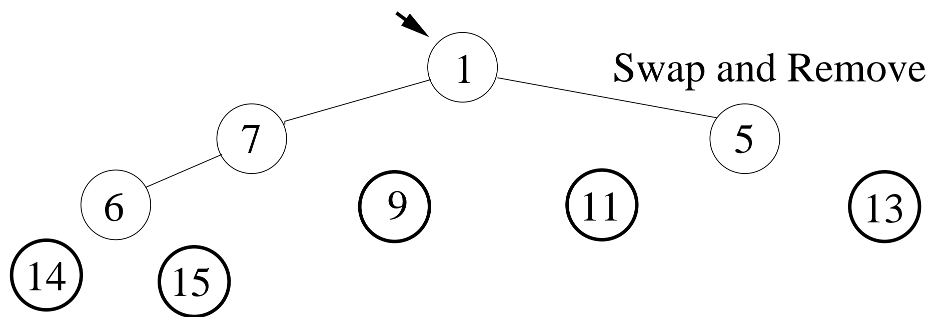


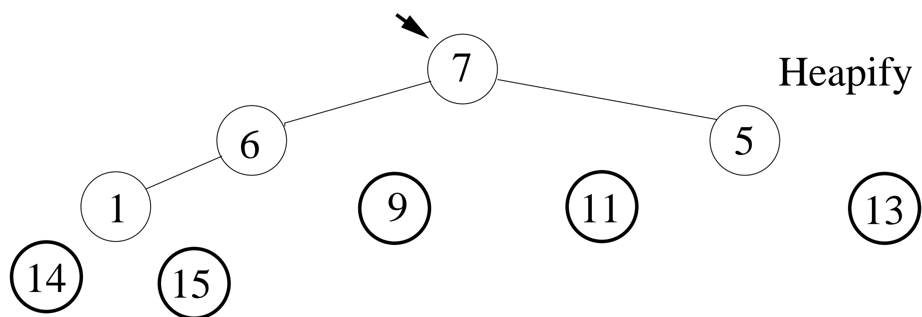


Heapsort Example Continued

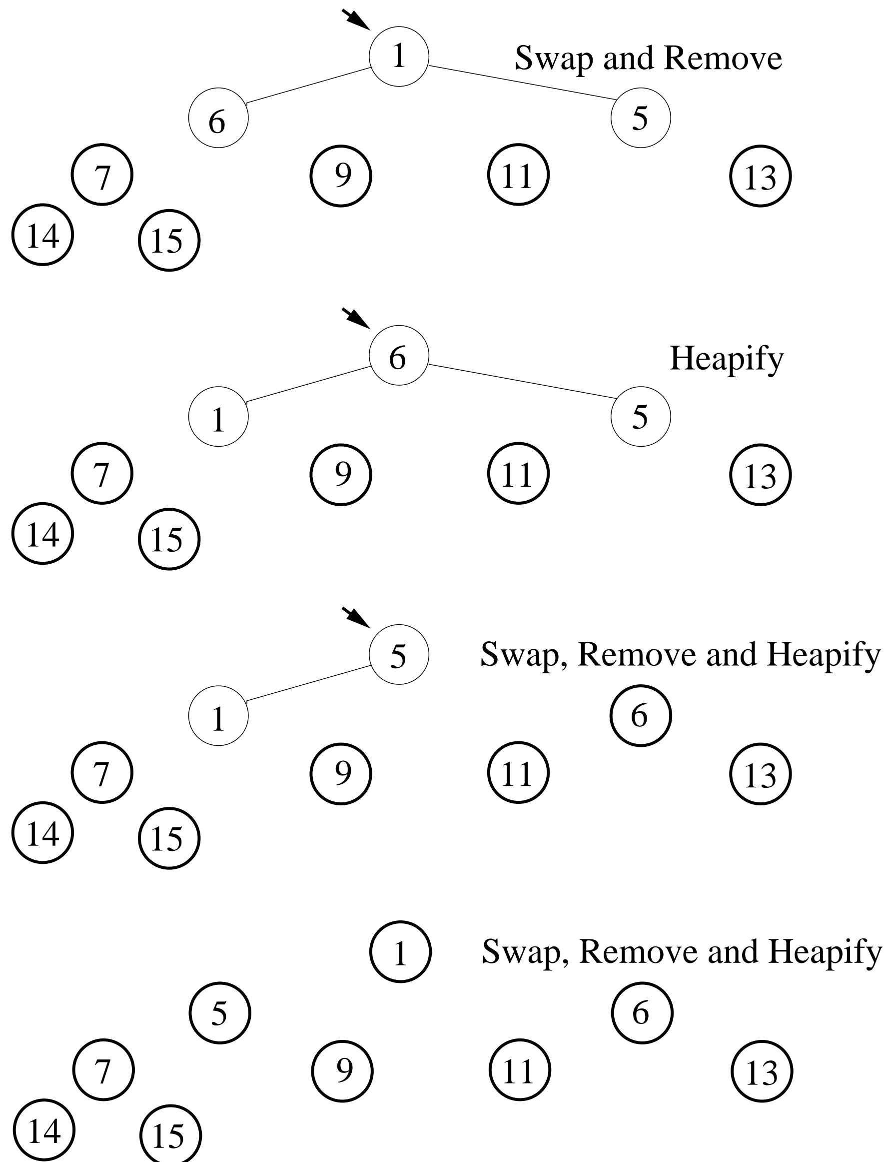








Heapsort Example Continued



We now have $A = [1, 5, 6, 7, 9, 11, 13, 14, 15]$.

Heapsort Complexity

- **BuildHeap** takes $O(n)$ time.
- **Heapify** takes $O(\log n)$ time.
- **Heapsort** makes one call to **BuildHeap** and $n-1$ calls to **Heapify**.
- Thus, the complexity of **Heapsort** is $O(n+(n-1)\log n) = O(n \log n)$.
- It is usually not as good as **Quicksort** in practice.