Binary Heaps & Heapsort

CSE 122 ~ Algorithms & Data Structures

Binary Heaps

- ➔ A heap is defined to be a **binary tree** with a key in each node such that:
  - All the leaves of the trees are adjacent to each other
  - All the leaves of the lower level occur from left to right
  - The key in the root is at least as large (or as small) as the keys in its children (if any) and the left and right children (if they exist) are again heaps
- ➔ The first two requirements of heaps are the definition of a **complete binary tree**
- ➔ A heap is NOT a binary search tree
- ➔ Heaps are used to implement priority queues ~ Prims and Dijsktra’s algorithms

Min/Max Heaps

- Minimum heap
  - Elements at every node will either be less than or equal to the element at its left and right child
  - The root must have the **smallest** (or equal to) key in the heap
- Maximum heap
  - Elements at every node will either be greater than or equal to the element at its left and right child
  - The root must have the **largest** (or equal to) key in the heap

Insertion in a heap

- Consider a **max** heap *H* with *n* elements. Insertion is done in two steps:
  - Add the new value at the bottom of the heap in such a way that *H* is still a **complete binary tree** but not necessarily a heap
  - Let the new value rise to its appropriate place in *H* so that *H* now becomes a heap.
    - To do this, compare the new value with the parent to see if they are in correct order. If not, swap and repeat.

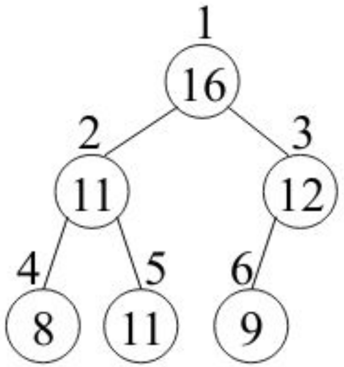
Algorithm to Insert Nodes into a Heap

```
// Add new value to a heap of size n
// pass in n and value
pos = n   // set position
// separate case if heap is empty
// if heap is not empty
if(!empty)
     // let the new value rise to the top
     while (pos != 0)
          parent = (pos - 1) /2;
          if (heap[pos] <= heap[parent])
               break;
          else
               swap(heap[pos], heap[parent])
     pos = parent
// move n
n = n + 1
```

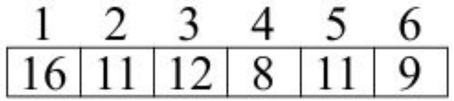
Complexity of Insertion to Heap

- Worst Case
  - So O(logn) for worst case
- Best Case
  - The while loop executes once and so does the if statement
    - So O(1)
- Average Case
  - Average case complexity O(nlogn)
- Building a heap is a **O(nlogn)**

Insertion: Example

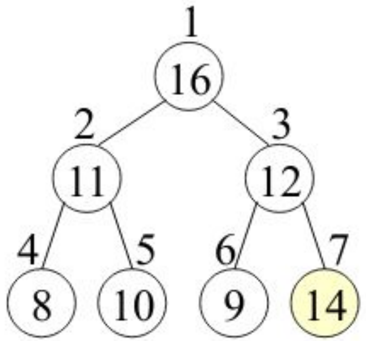


viewed as a binary tree

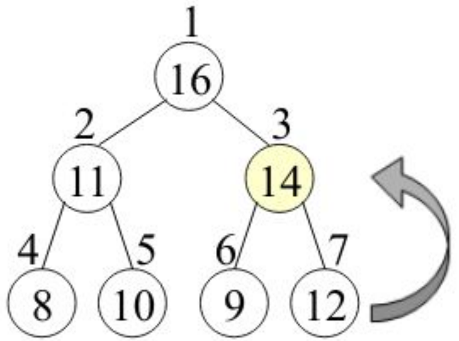


viewed as an array

Insertion ~ Example Cont.



viewed as a binary tree



viewed as a binary tree

1	2	3	4	5	6	7
16	11	12	8	10	9	14

viewed as an array

1	2	3	4	5	6	7
16	11	14	8	10	9	12

viewed as an array

Deletion from a Maximum Heap

- Consider a max heap *H* having *n* elements. An element is always deleted from the root of the heap. So deleting an element from the heap is done in the following two steps.
  - Replace the root's node value with the last node's value so that *h* is still a complete binary tree but not necessarily a heap
  - Delete the last node
  - Sink the new root node so *H* satisfies the heap property.
    - In this step interchange the root node value with its child node's value (whichever is **larger**)

Algorithm to Delete from a Max Heap

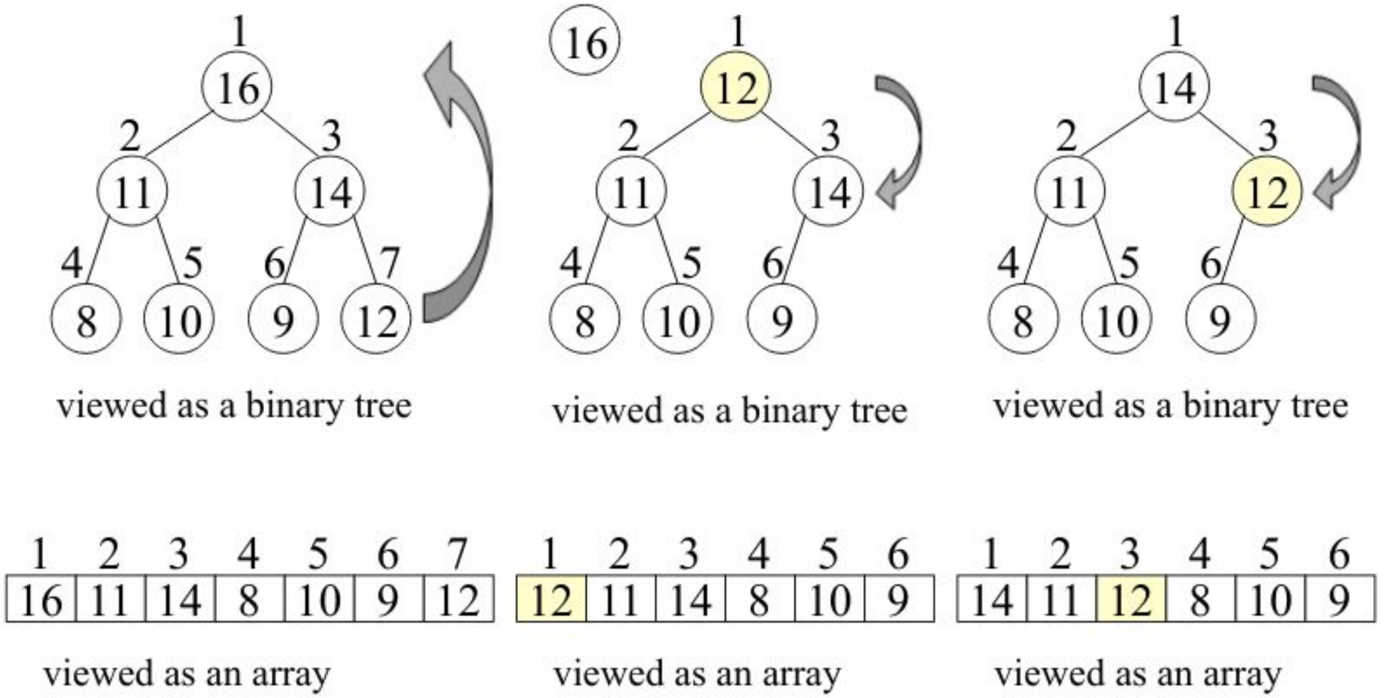
```
// Assume you have a heap H with n
elements
// pass in n
last = heap[n]
// these are needed if you sort
// top = heap[0]
// heap[n] = top; // in=place sort
n = n - 1       // delete last node
Parent = 0
left = 1
right = 2
heap[parent] = last;
```

```
if(n > 1)
     // sink new root node to reheapify
     while( left <= n)
          if(heap[parent] >= heap[left])
               &&
            (heap[parent] >= heap[right])
               Break;
          if(heap[right] <= heap[left])
               swap(head[parent],
                    heap[left])
               parent = left
          else
               swap(head[parent],
                    heap[right])
               parent = right
          left = 2* parent + 1
          right = 2 * parent + 2
```

Complexity of Deletion from Max Heap

- Worst Case
  - O(logn)

Deletion ~ Example



1	2	3	4	5	6	7
16	11	14	8	10	9	12

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

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

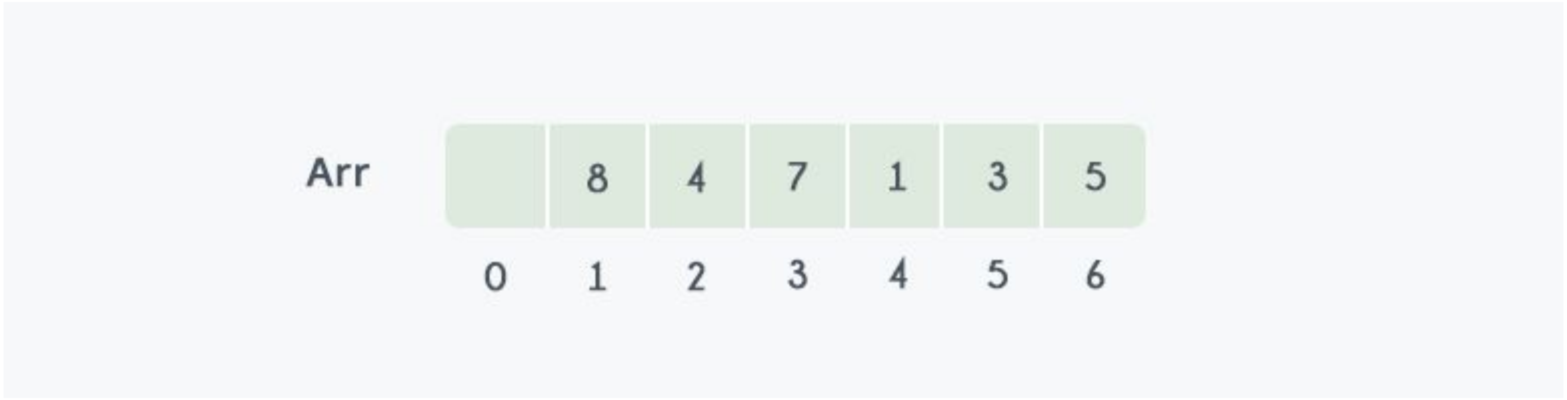
Heapsort

- ➔ Developed in 1964 by J.W.J Williams
  - Published in the Communications of the ACM
- ➔ The sort has complexity of O(nlogn)
- ➔ Basic Idea:
  - Largest element at top of heap. So swap with last element of heap and sink new root to restore heap and repeat
- ➔ Two-step process
  - Build heap (insert nodes into heap)
  - Delete root - swap root and last element of heap and sink new root to restore heap. Elements come off in descending order

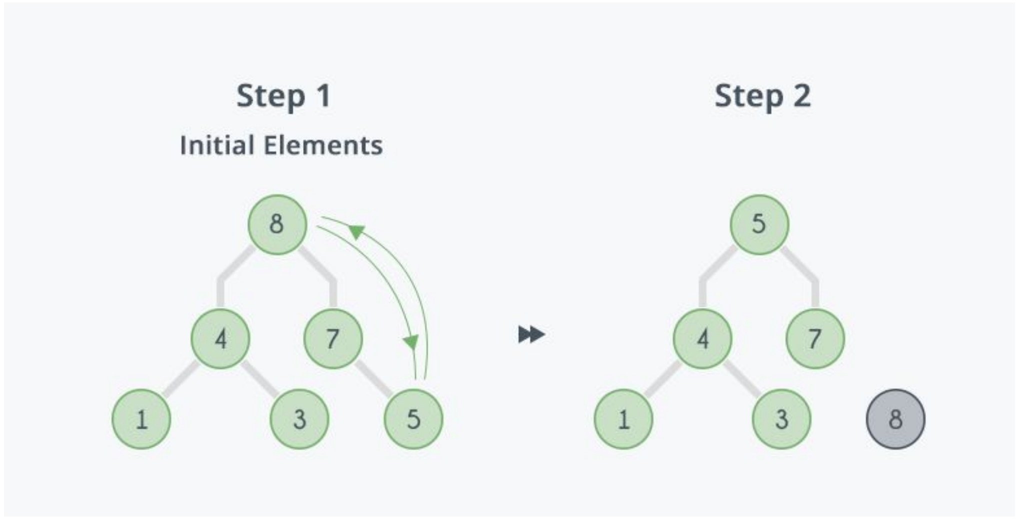
Algorithm for Sorting

- ➔ Similar to algorithm for deletion
  - Add swap and keep “deleted” item in the last element’s place on the heap
- ➔ Complexity O(nlogn)
  - Why? - insertion - comparisons max *logn* the height of the tree and do this *n* times so insertions is O(nlogn)
  - Deletion is O(logn) - do this for *n* elements so O(nlogn)

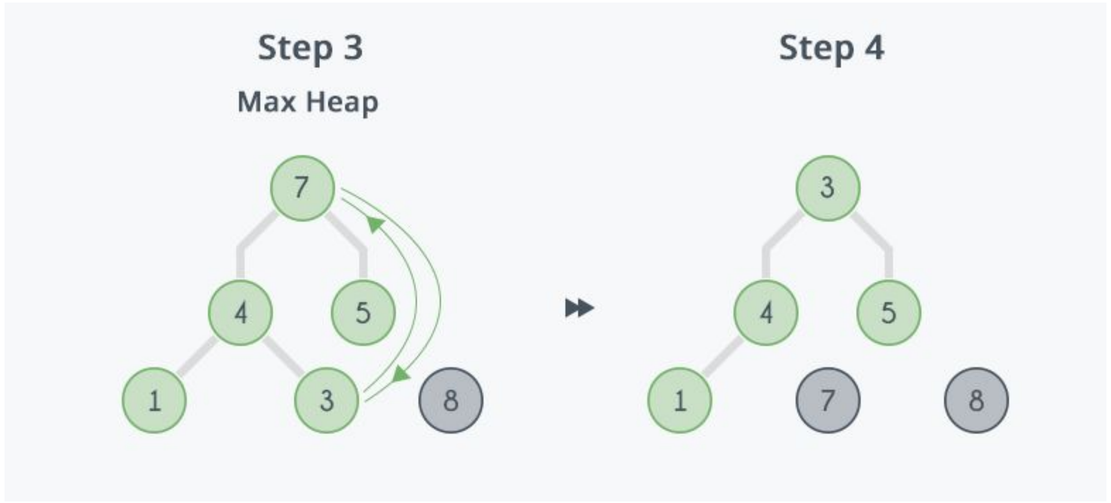
Heapsort Example



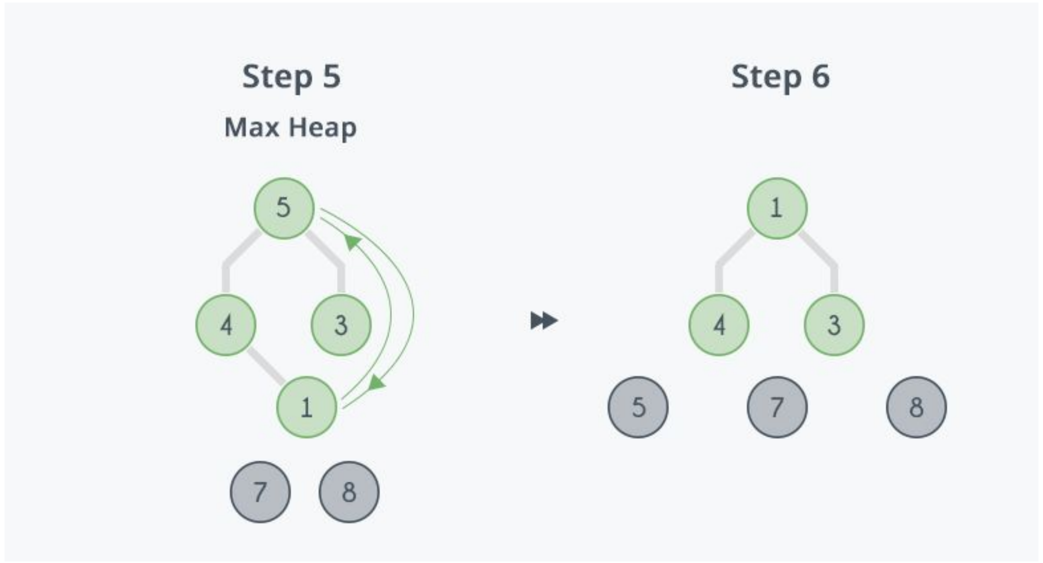
HeapSort Example ~ Cont.



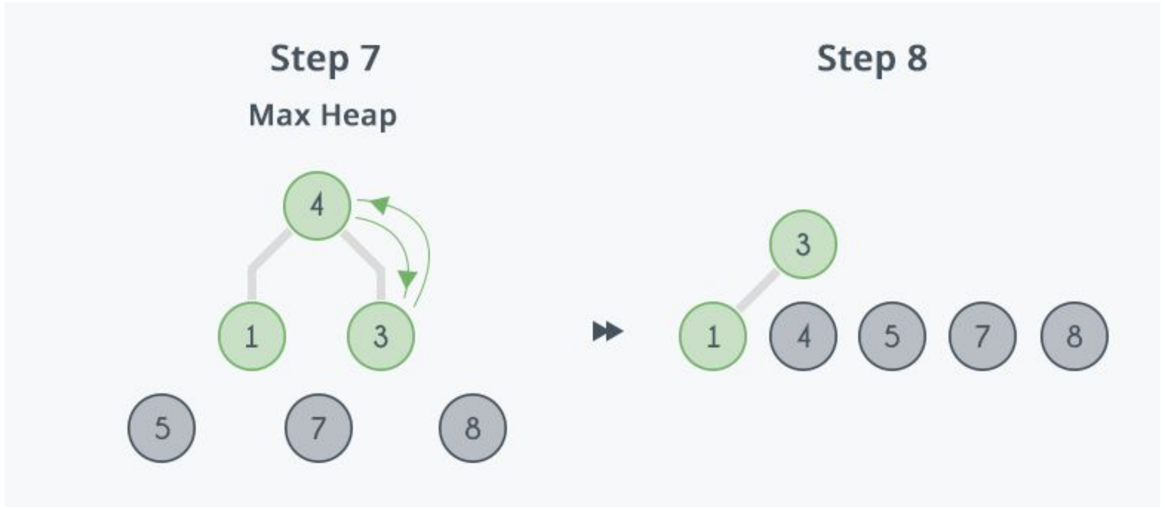
Heapsort Example ~ Cont.



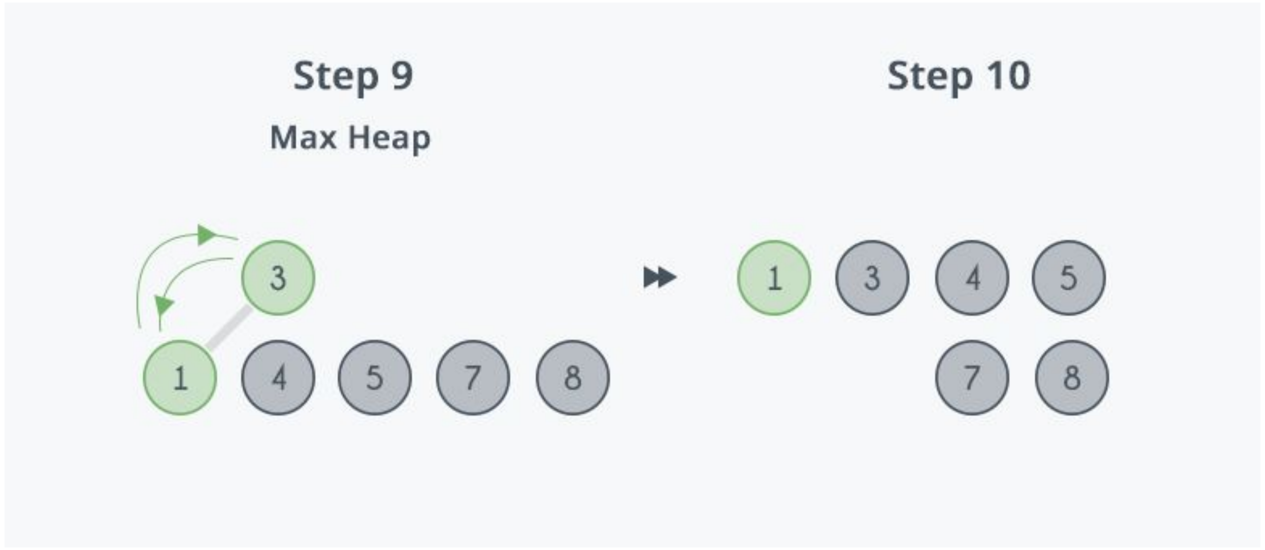
Heapsort Example ~ Cont.



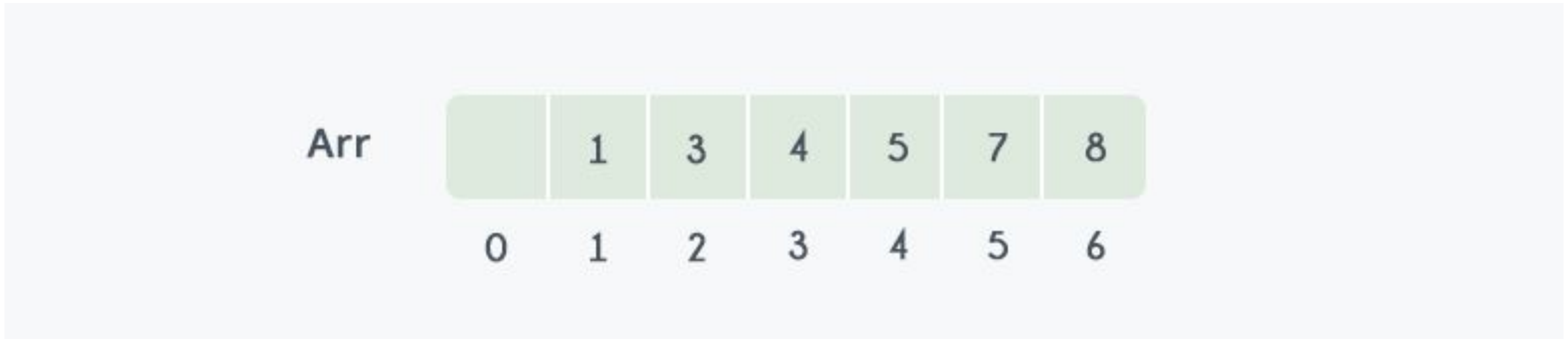
Heapsort Example ~ Cont.



Heapsort Example ~ Cont.



Heapsort Example ~ Cont.


Arr
1
3
4
5
7
8
0
1
2
3
4
5
6