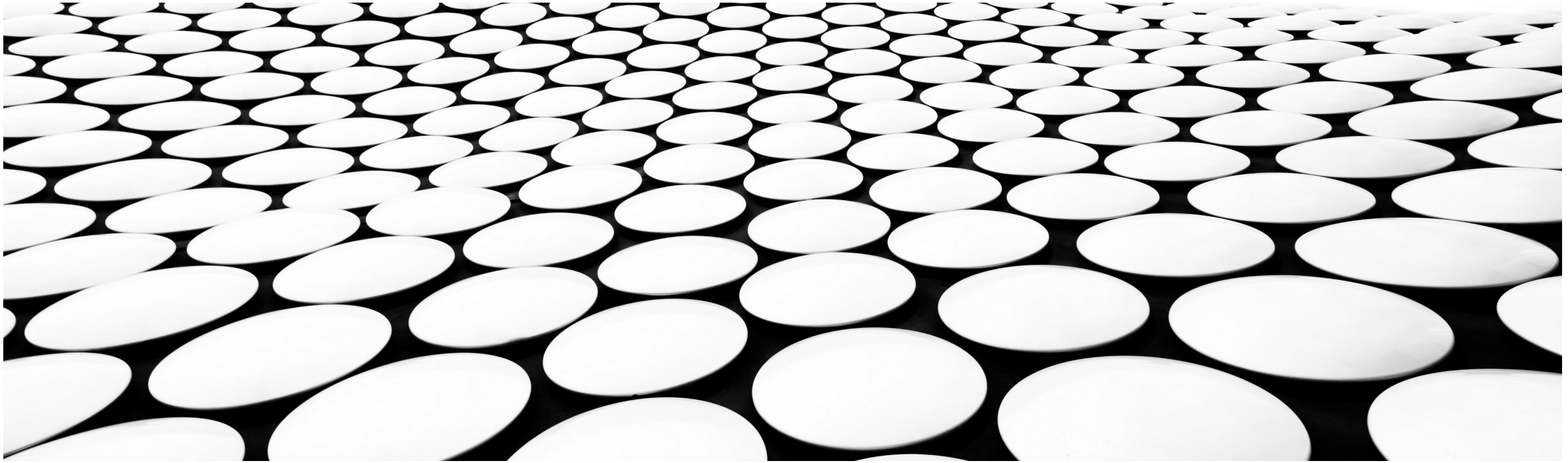
DSA 456 – WEEK 10

HEAPIFY AND HEAP SORT



HEAPIFY AND HEAP SORT

A complete binary tree with n nodes means that at most there are log n nodes from the root (top) to a leaf (a node at the bottom of the tree)

Insertion may require the percolate up process. The number of times a node needs to percolate up can be no more than the number of nodes from the root to the leaf. Therefore, it is pretty easy to see that this percolate up process is O(log n) for a tree with n nodes. This means that to add a value to a Heap is a process that is O(log n). In order for us to build a heap we need to insert into it n times. Therefore, the worst case runtime for this process is O(n log n)

The deletion process may require a percolate down process. Like the percolate up process this also is O(log n). Thus, to remove a value from the heap is O(log n). We need to remove n values so this process is O(n log n).

To heap sort we build heap O(n log n) then destroy the heap O(n log n). Therefore the whole sorting algorithm has a runtime of O(n log n)

HEAPIFY AND HEAP SORT

```
def heap_sort(mylist):
    the_heap=Heap()      # we assume we have created some Heap Object
    size = len(mylist)
    for i in range(0,size):
        the_heap.insert(mylist[i])

    for i in range(0,size):
        mylist[i]=the_heap.front()
        the_heap.delete()
```

but this is not actually a good implementation. The above would create a heap as it goes meaning that we need to actually double the data storage as the heap itself would be an array that is as big as data. Thus, the function would have the same drawback as merge sort. Furthermore building a heap by multiple insertions is actually slower than making a heap in place. Thus, instead of building an actual heap with a heap object, we will implement heap sort by turning the array into a heap in place then removing from it in place.

HEAPIFY AND HEAP SORT

Heapify

Since our heap is actually implemented with an array, it would be good to have a way to actually create a heap in place starting with an array that isn't a heap and ending with an array that is heap. While it is possible to simply "insert" values into the heap repeatedly, the faster way to perform this task is an algorithm called Heapify.

In the Heapify Algorithm, works like this:

Given a node within the heap where both its left and right children are proper heaps (maintains proper heap order) , do the following:

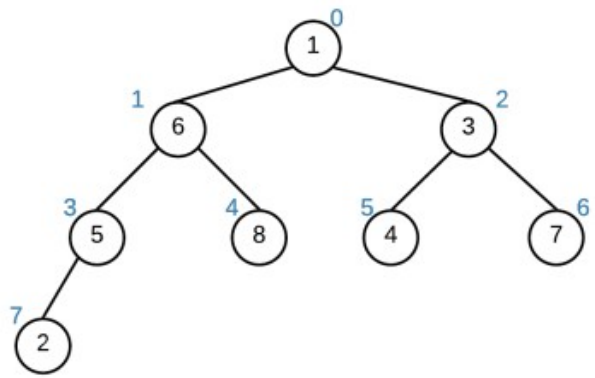
- If the node has higher priority than both children, we are done, the entire heap is a proper heap
- Otherwise
  - swap current node with higher priority child
  - heapify() that subtree

Effectively what is happening is that we already know that both the left and right children are proper heaps. If the current node is higher priority than both children then the entire heap must be proper

However if its not then we do a swap, this means that the current node's value has now gone down into one of the subtrees. This could cause that subtree to no longer be a heap because the root of that subtree now has a value of lower priority than it use to have. so we heapify the subtree.

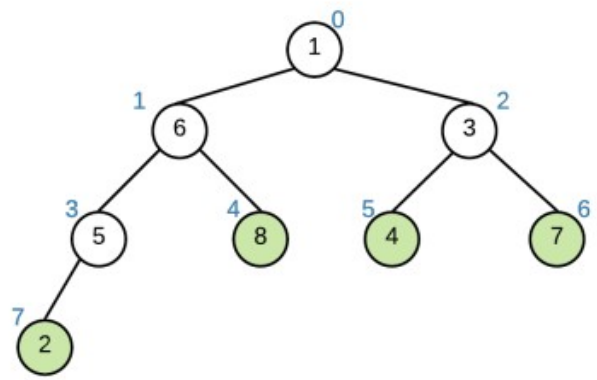
Building a maxheap in place

This example will start with an array that is not heap. It will perform heapify() on it to form a max heap (bigger values have higher priority). In each of the diagrams below, the argument to heapify() is the index corresponding to the node



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

Firstly, all leaf nodes are valid heaps. Since they have no subtree, we don't need to deal with those nodes. They are highlighted in green in this next picture. Our algorithm therefore starts at the first non-leaf node from the bottom. This node is at index (n-2)/2 where n is the total number of values in our heap.

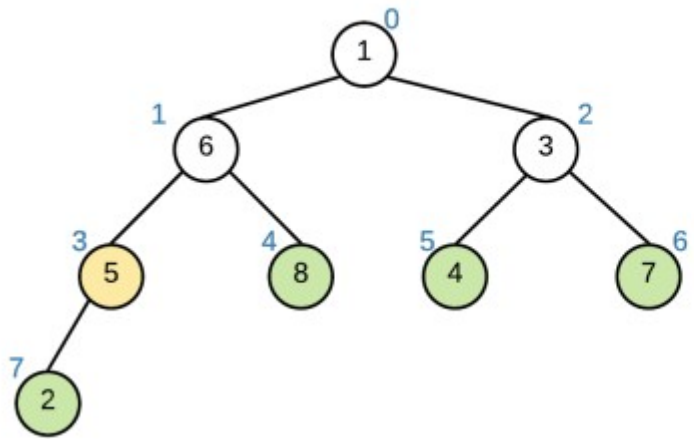


1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

The heapify function takes the ***index*** of the root of the heapify routine (ie we know that nodes children are heaps, and we are looking at it from that node down.

heapify(3)

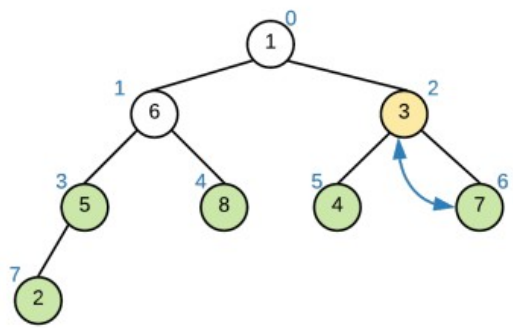
First node to consider is the node with 5 (index 3). the left subtree is lower priority and the right subtree doesn't exist so we do nothing.



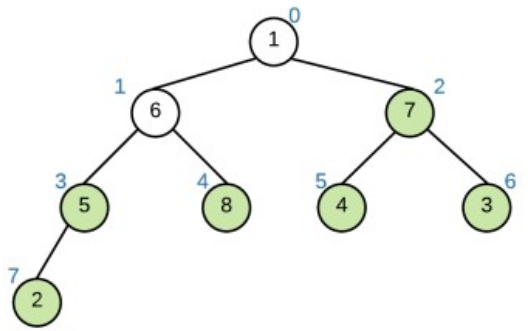
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

heapify(2)

Second node to consider is 3 (index 2). both 4 and 7 are bigger than 3 and thus have higher priority. However, 7 is higher priority than 4, so we swap these two values. 3 is now in a leaf node and thus we can stop.



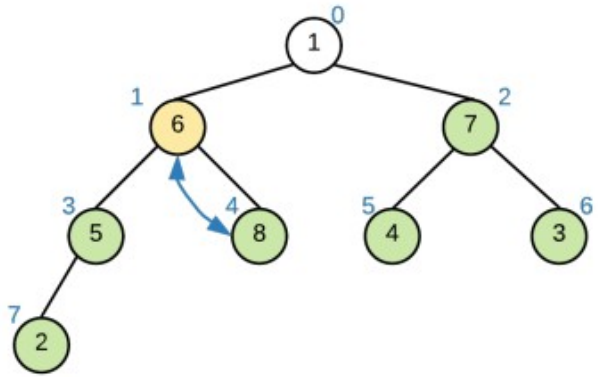
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7



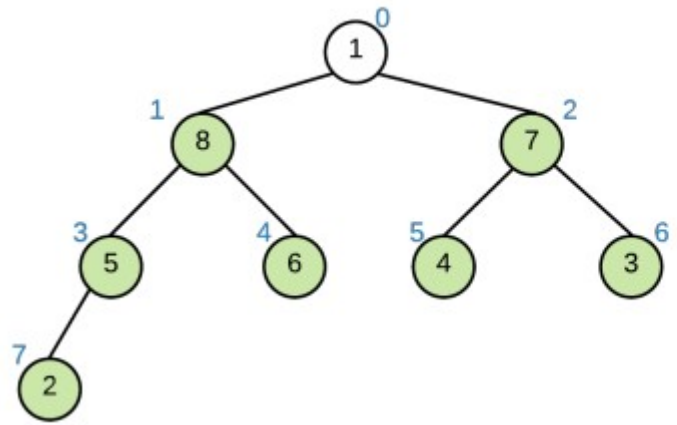
1	6	7	5	8	4	3	2
0	1	2	3	4	5	6	7

heapify(1)

Next consider the node with 6. 6 is higher priority than 5 but not higher priority than 8, so we swap them. 6 is now in a leaf node and thus we can stop.



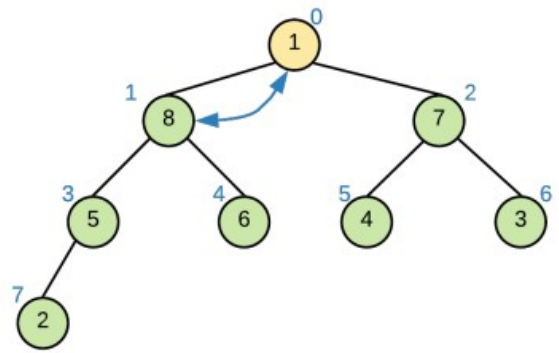
1	6	7	5	8	4	3	2
0	1	2	3	4	5	6	7



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

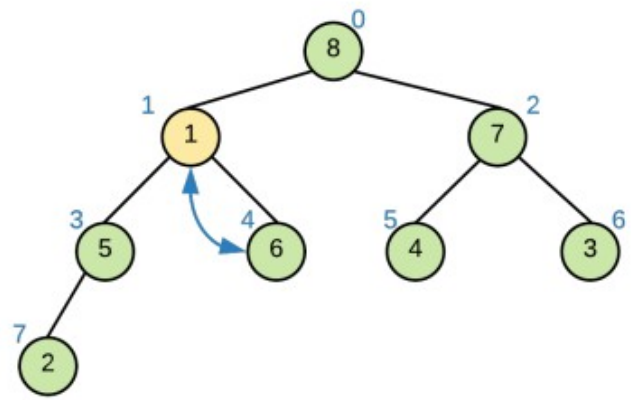
heapify(0)

Lastly we consider the node 1. both 8 and 7 have higher priority than 1, but 8 is higher priority than 7 and thus we swap 8 and 1 first.



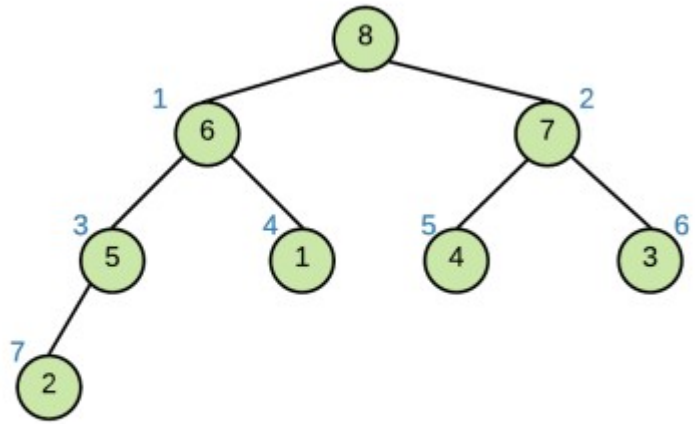
1	8	7	5	6	4	3	2
0	1	2	3	4	5	6	7

At this point we need to heapify the subtree that now has 1 as it's root (index 1). We swap it with the higher priority child (6 in this case). Notice that because we won't need to do anything to the right subtree with 7 as root, as that subtree was not modified.



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

After the final swap we have a proper max heap



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

Once the heap is built, we continuously remove the highest priority value from the heap and place it at the back of the array.

END OF WEEK 10

- Please work on your lab 6 and assignment 3

INTRODUCTION TO HEAPS

- A heap is a special tree-based data structure.
- It is a complete binary tree where the parent node satisfies the heap property.
- Two types: Min-Heap (parent ≤ children), Max-Heap (parent ≥ children).
- Used in Priority Queues, Scheduling, Graph Algorithms, etc.

HEAP OPERATIONS

- Insertion: Insert at the last position and bubble up.
- Deletion: Remove root and replace it with last element, then heapify.
- Heapify: Process of ensuring heap property is maintained.
- Time Complexity: Insertion (O(logN)), Deletion (O(logN)), Heapify (O(N)).

HEAPIFY IN HEAPS

- Heapify ensures the heap structure is maintained.
- Bottom-up approach: Used when inserting elements.
- Top-down approach: Used when deleting the root element.
- Efficient way to build a heap in O(N) time.

HEAP SORT ALGORITHM

- Step 1: Build a max heap from the array.
- Step 2: Swap the root (largest element) with the last item.
- Step 3: Reduce heap size and heapify the root.
- Step 4: Repeat until the heap size is 1.
- Time Complexity: O(N log N)

HEAP SORT EXAMPLE

- Given array: [4, 10, 3, 5, 1]
- Step 1: Convert to max heap: [10, 5, 3, 4, 1]
- Step 2: Swap 10 with 1, heapify: [5, 4, 3, 1]
- Step 3: Repeat until sorted: [1, 3, 4, 5, 10]

HEAP SORT VS OTHER SORTING ALGORITHMS

- Heap Sort: O(N log N), worst-case O(N log N), in-place sorting.
- Quick Sort: Average O(N log N), worst-case O(N^2), fast but unstable.
- Merge Sort: O(N log N), stable but requires extra space.
- Heap Sort is useful for large datasets with priority-based sorting.

PRACTICE QUESTIONS

- 1. What is the difference between a min-heap and a max-heap?
- 2. How does heapify ensure the heap property?
- 3. Perform heap sort on the array: [8, 3, 10, 5, 1, 6].
- 4. Compare heap sort with merge sort in terms of space complexity.
- 5. Implement a max-heap insertion step by step.

PRACTICE QUESTION

- What is the difference between a Min-Heap and a Max-Heap?

SOLUTION

- In a **Min-Heap**, the parent node is always smaller than its children.
- In a **Max-Heap**, the parent node is always larger than its children.
- Example:
- Min-Heap: [1, 5, 3, 10, 8] (1 is the smallest)
- Max-Heap: [10, 8, 5, 3, 1] (10 is the largest)

PRACTICE QUESTION

- How does Heapify ensure the heap property?

SOLUTION

- Heapify works by moving elements up or down to maintain the heap structure.
- Two approaches:
- - **Bottom-Up Heapify**: Used when inserting elements, percolates up.
- - **Top-Down Heapify**: Used when deleting the root, percolates down.
- Time Complexity: O(logN)

PRACTICE QUESTION

- Perform Heap Sort on the array: [8, 3, 10, 5, 1, 6]

SOLUTION

- **Step 1:** Build Max-Heap: [10, 5, 8, 3, 1, 6]
- **Step 2:** Swap root (10) with last element → [6, 5, 8, 3, 1, 10]
- **Step 3:** Heapify root → [8, 5, 6, 3, 1, 10]
- **Step 4:** Repeat until sorted: [1, 3, 5, 6, 8, 10]

PRACTICE QUESTION

- Compare Heap Sort with Merge Sort in terms of space complexity.

SOLUTION

- Heap Sort: **O(1)** (in-place sorting, does not require extra memory)
- Merge Sort: **O(N)** (requires additional space for merging)
- **Heap Sort is more space-efficient but less stable than Merge Sort.**

PRACTICE QUESTION

- Implement a Max-Heap insertion step-by-step for [4, 10, 3, 5, 1, 6].

SOLUTION

- **Step 1:** Insert elements one by one.
- **Step 2:** After inserting 10 → [10, 4, 3, 5, 1, 6]
- **Step 3:** Bubble up 10 to the correct position.
- **Step 4:** Resulting Max-Heap: [10, 5, 6, 4, 1, 3]