

# Heapsort

CS 112 Spring 2020 – April 23

Sesh Venugopal

# Flashback: Sorting using a heap

# Sorting using a heap

To sort a set of items, insert them one at a time in a heap. When all inserts are done, perform a sequence of deletes. This gives back all the items in descending order.

Input: 4, 2, 15, 0, -10 inserted one at a time into a heap:

[4] , [4, 2], [15, 2, 4], [15, 2, 4, 0], [15, 2, 4, 0, -10]

Delete from the heap one at a time:

The deleted items can be stored  
in an output array, back to front

15 [4, 2, -10, 0]  
4 [2, 0, -10]  
2 [0, -10]  
0 [-10]  
-10 []

Sorted Output:

[-10, 0, 2, 4, 15]

## Sorting using a heap: worst case big O running time

What we know:

- Worst case time to insert in a heap:  $O(\log n)$
- Worst case time to delete from a heap:  $O(\log n)$

Inserting  $n$  items one at a time into a heap, running time:

$$\log(1) + \log(2) + \log(3) + \dots + \log(n-1) + \log(n)$$

(we'll put the big O back in at the end)

$$\Rightarrow \log(1*2*3*\dots*(n-1)*n)$$

$$\Rightarrow \log(n!)$$

There is a result called **Stirling's formula** that approximates  $n!$  like this:

$$n! \approx (n/e)^n \sqrt{2\pi n}$$

Which essentially implies that

$$\log(n!) = O(n \log n)$$

So the running time to insert all  $n$  items is  $O(n \log n)$

## Sorting using a heap: worst case big O running time

Deleting n items one at a time into a heap, running time:

$$\log(n) + \log(n-1) + \log(n-2) + \dots + \log(2) + \log(1)$$

This is exactly the same series as the inserts, just written in reverse order.  
So the running time to delete all n items is  $O(n \log n)$

So the total running time for the sort is

$$O(n \log n) + O(n \log n) = O(n \log n)$$

```
int[] arr = {4,2,15,0,-10};

Heap<Integer> sortHeap = new Heap<Integer>(n);

// insert one at a time
for (int val: arr) {
    sortHeap.insert(val);
}

// delete one at a time, and store in result
for (int i=arr.length-1; i >= 0; i--) {
    arr[i] = sortHeap.delete();
}
```

# Heapsort

A faster (in real time) algorithm

## Sorting using a heap

Building a heap out of  
input sequence



Getting sorted result  
with repeated deletes  
(which uses sift downs)



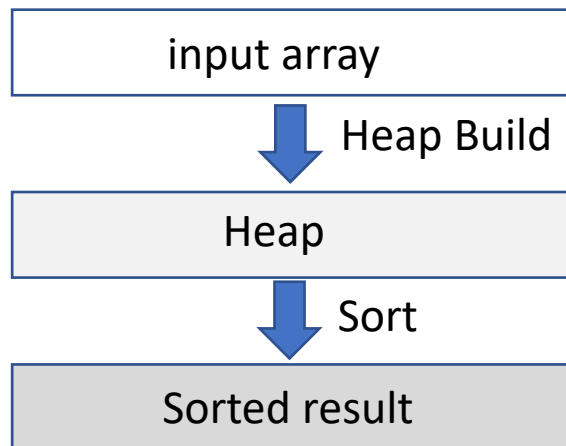
```
int[] arr = {4,2,15,0,-10};

Heap<Integer> sortHeap = new Heap<Integer>(n);

// insert one at a time
for (int val: arr) {
    sortHeap.insert(val);
}

// delete one at a time, and store in result
for (int i=arr.length-1; i >= 0; i--) {
    arr[i] = sortHeap.delete();
}
```

## Heapsort

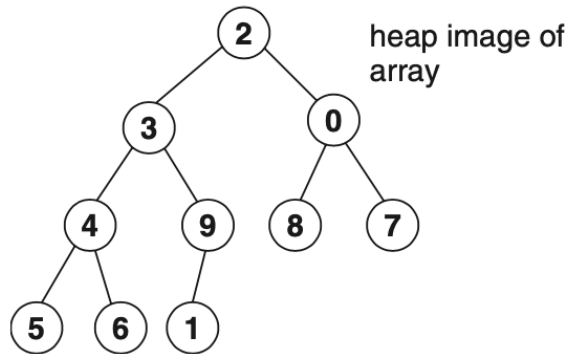


1. We will NOT use a Heap class. Instead, sorting is done directly on the array storage
2. The most significant difference is in the way we build the heap out of the input array
3. Once the heap is built, the process to get a sorted result is similar to the previous version, as in using repeated sift downs

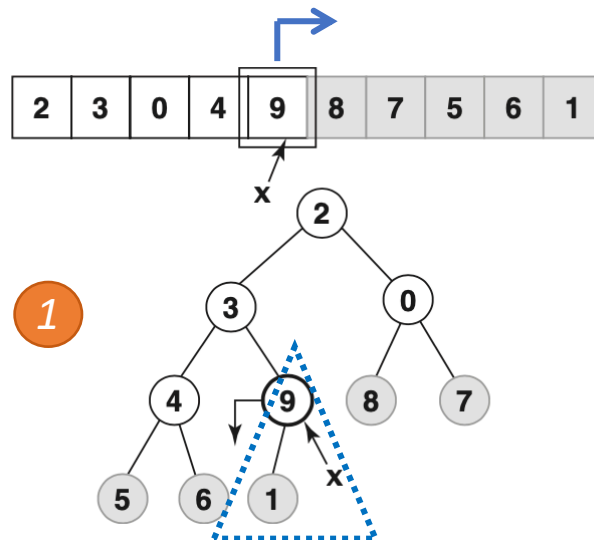
# Heapsort Step 1 - Building a heap out of the input array

Input array

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 0 | 4 | 9 | 8 | 7 | 5 | 6 | 1 |
|---|---|---|---|---|---|---|---|---|---|



The build algorithm goes up from last level to the top, building bigger and bigger heaps



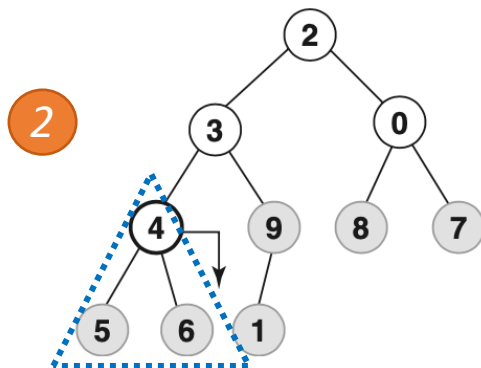
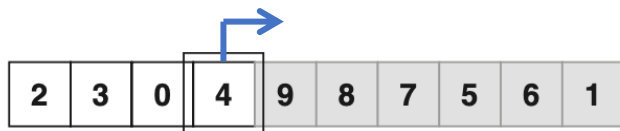
To start with, all leaf nodes are trivially heaps

The build algo begins its real work at the "last" non-leaf (internal) node. Here, this is the node 9. It needs to build a bigger heap at 9, on top of the subtrees which are already heaps (here, just the node 1) – this means moving 9 to its correct position by sifting down.

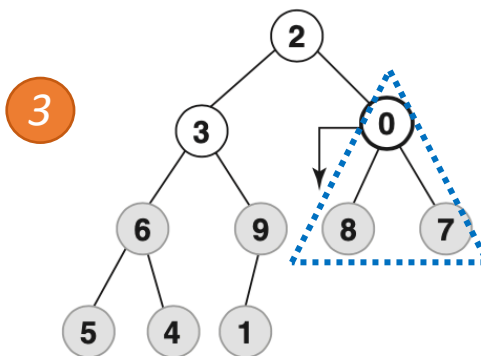
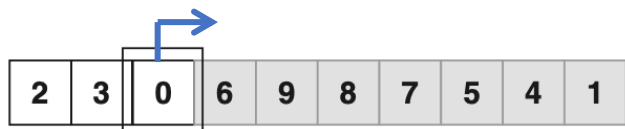
The subtree at 9 is said to now be "heapified"



# Heapification

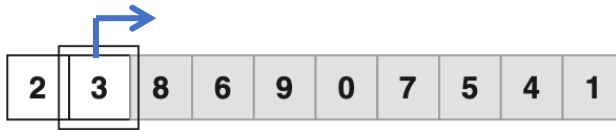


Next up for heapification is the node 4. The subtrees at 5 and 6 are already heaps, so 4 might be the only value out of position, and is sifted down, resulting in swapping with 6



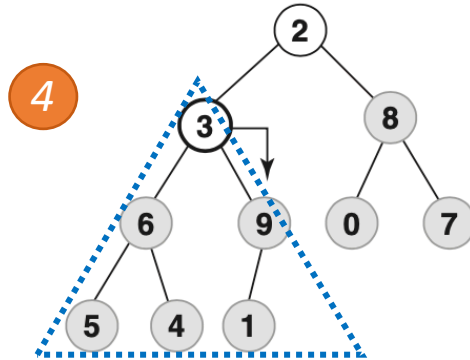
The algo proceeds to the next level up, where it starts with the last node, 0, for heapification  
0 is sifted down, resulting in swapping with 8

# Heapification



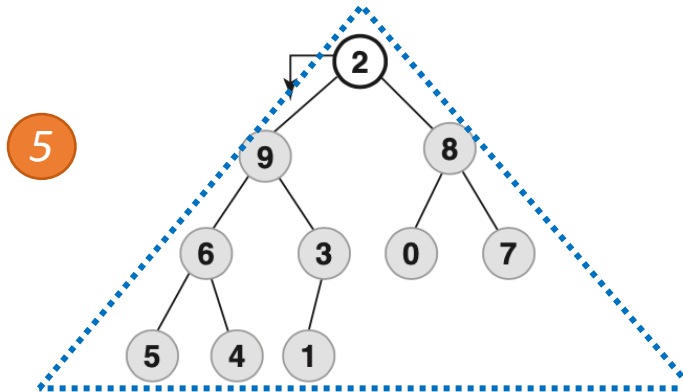
Then the algo moves on to node 3.

This node has two subheaps under it. 3 sifts down, resulting a swap with 9.



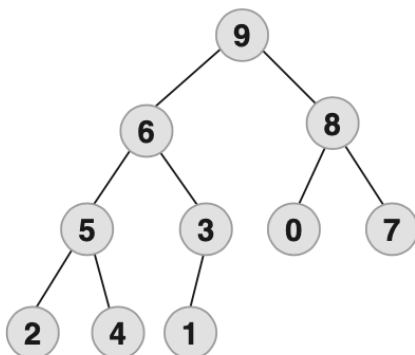
Finally, heapification moves to the top of the heap, node 2.

2 is sifted down, swapping with 9, then swapping with 6, and finally swapping with 5



# Heapification

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 6 | 8 | 5 | 3 | 0 | 7 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|



This completes the heapification process.  
The array now holds a heap-ordered sequence of items.


How is this heapification better than building a heap by inserting one item a time, like before, which took  $O(n \log n)$  time?

It just seems like we are doing a bunch of sift downs, instead of a bunch of sift ups, and both sift ups and sift downs are  $O(\log n)$  time in the worst case.

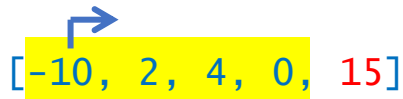
Surprisingly, it turns out that heapification only takes  $O(n)$  time !!!  
In other words, heapification (or heapify) with repeated sift downs runs in LINEAR time

## Sorting the heap

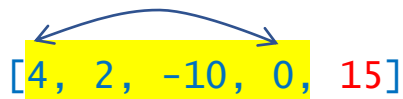
[15, 2, 4, 0, -10]



[-10, 2, 4, 0, 15]



[4, 2, -10, 0, 15]



[0, 2, -10, 4, 15]



[2, 0, -10, 4, 15]



[-10, 0, 2, 4, 15]



[0, -10, 2, 4, 15]



[-10, 0, 2, 4, 15]

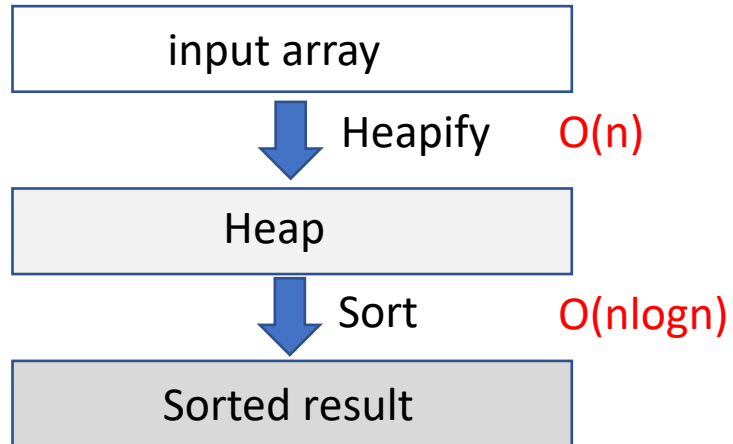
Top of heap is the max value

Swap with last, then sift down from top with an effective heap that excludes the last value

Repeat this process until one value is left in effective heap

This is identical to repeatedly deleting from a heap, so worst case running time is  $O(n \log n)$

## Heapsort Running Time Worst Case



Total time is  $O(n\log n)$  with the real time saving coming in the heapify step

# Heapsort: Implementation

# Heapsort Implementation

See Resources -> Apr 23 ->

[Heapsort.java](#) for the implementation.

A few things to observe:

1. Since both the heapify and the sorting steps use sift down, there is no need to implement sift up.
2. In the heapify process, sift down is done from various points in the array, so there is an additional parameter **k** to the sift down method that tells from where to start sifting down
3. Since the sort process keeps accumulating sorted items at the end of the array, and the heap size effectively keeps decreasing after every iteration, there is an additional parameter **n** to sift down that tells how many items are effectively in the heap
4. The “last” internal node (where heapify starts doing sift downs) is at index  $(n-2)/2$

```
private static <T extends Comparable<T>>
void siftDown(T[] list, int k, int n) {
    ...
}

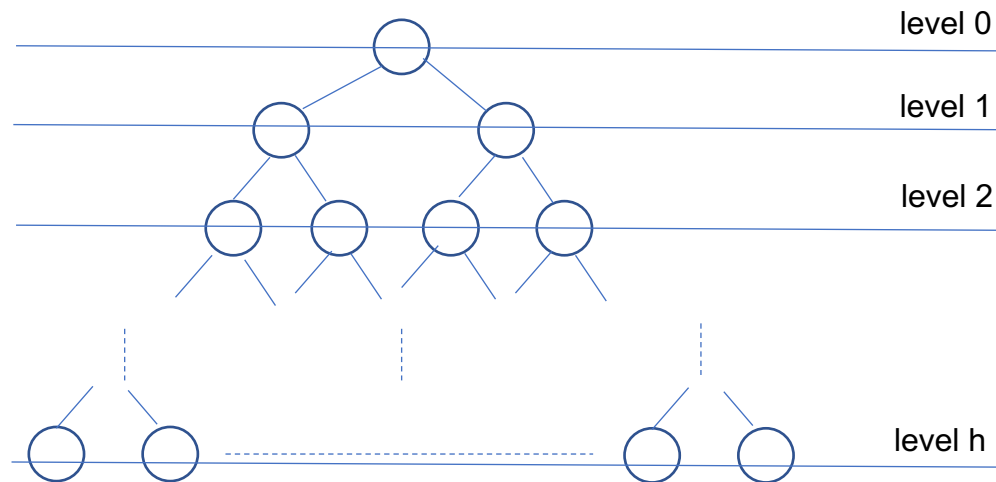
public static <T extends Comparable<T>>
void sort(T[] list) {
    // build heap, linear [O(n)] time
    for (int k=(list.length-2)/2; k >= 0; k--) {
        siftDown(list, k, list.length);
    }
    // sort (n log n time)
    for (int n=list.length; n > 1; n--) {
        // swap max and last
        T max = list[0];
        list[0] = list[n-1];
        list[n-1] = max;
        // sift down from 0,
        // in sub array of length n-1
        siftDown(list, 0, n-1);
    }
}
```

Heapify:  
Derivation of worst case  $O(n)$  running time  
(Not required for final exam)



## Heap – number of nodes in terms of height, h

Heap is a complete tree, so all levels except last must be full.  
Let's assume that the last level is full as well – it won't make a difference for the big O result



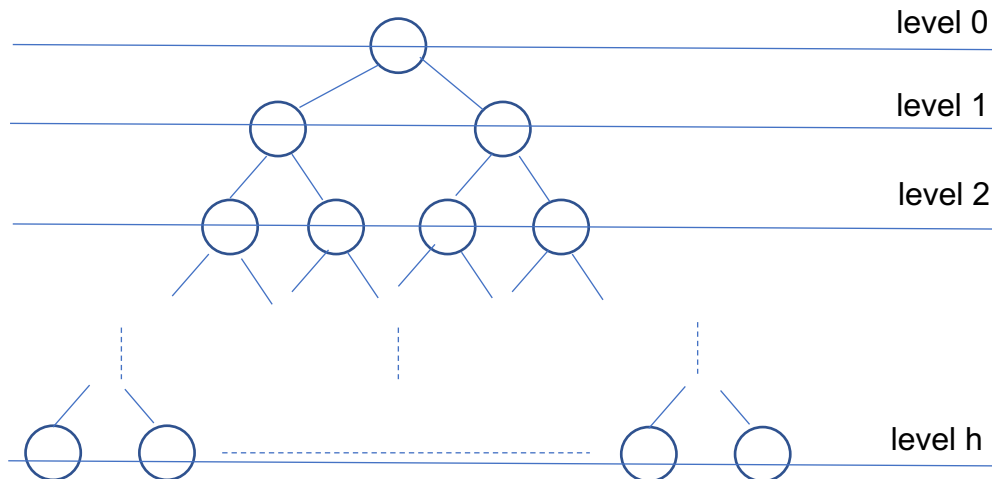
Assume the height of the heap is  $h$

The number of nodes/items,  $n$ , in this heap is computed by adding up the number of nodes at each level, and we can then write  $h$  in terms of  $n$ :

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

(We saw this earlier when we derived running time for sift up and sift down in a heap)

# Heapify: Sift downs



Sifting down each level requires 2 comparisons: one between the children, and one between the larger child and the parent

Sifting down is done for all nodes at each non-leaf level.

- For each node at level 0, in the worst case sift down would go all the way to the bottom, and would require  $2 \cdot (h)$  comparisons
- Similarly, for each node at level 1, sifting down would require  $2 \cdot (h-1)$  comparisons
- We can do this for every level up to the last but one: at level  $(h-1)$ , sift down for each node would take  $2 \cdot (h-(h-1)) = 2$  comparisons
- There are  $2^0$  nodes at level 0,  $2^1$  nodes at level 1, ...,  $2^{h-1}$  nodes at level  $(h-1)$

# Heapify: Sift downs

Sifting down is done for all nodes at each non-leaf level.

- For each node at level 0, in the worst case sift down would go all the way to the bottom, and would require  $2^*(h)$  comparisons
- Similarly, for each node at level 1, sifting down would require  $2^*(h-1)$  comparisons
- We can do this for every level up to the last but one: at level  $(h-1)$ , sift down for each node would take  $2^*(h-(h-1)) = 2$  comparisons
- There are  $2^0$  nodes at level 0,  $2^1$  nodes at level 1, ...,  $2^{h-1}$  nodes at level  $(h-1)$

So the total number of comparisons,  $S$ , in the worst case, to sift down all nodes at all levels except the last is:

$$S = 2^0 * 2h + 2^1 * 2(h-1) + 2^2 * 2(h-2) + \dots + 2^{h-2} * 2(h-(h-2)) + 2^{h-1} * 2(h-(h-1))$$

This is a difficult series to sum because each term has two varying parts

So, we're going to use a little trick to eliminate of the variations, and get a simple geometric series

# Heapify: Sift downs

The trick is to multiple the sum by 2, and write the result underneath the original:

$$S = 2^0 * 2h + 2^1 * 2(h-1) + 2^2 * 2(h-2) + \dots + 2^{h-2} * 2(h-(h-2)) + 2^{h-1} * 2(h-(h-1))$$

$$2S = 2^1 * 2h + 2^2 * 2(h-1) + 2^3 * 2(h-2) + \dots + 2^{h-1} * 2(h-(h-2)) + 2^h * 2(h-(h-1))$$

Basically every term on the LHS and RHS are multiplied by 2. On the RHS, this adds 1 to the power of each of the original powers of 2

This doesn't really seem to do much, but what if we right shifted every term in the second series and lined it up under the first, like this:

$$S = 2^0 * 2h + 2^1 * 2(h-1) + 2^2 * 2(h-2) + \dots + 2^{h-2} * 2(h-(h-2)) + 2^{h-1} * 2(h-(h-1))$$

$$2S = \qquad 2^1 * 2h \qquad + 2^2 * 2(h-1) + 2^3 * 2(h-2) + \dots \qquad + 2^{h-1} * 2(h-(h-2)) + 2^h * 2(h-(h-1))$$

And then subtracted the first series from the second, term by term:

$$S = -2^0 * 2h + 2^1 * 2 \qquad + 2^2 * 2 \qquad + 2^3 * 2 + \dots \qquad + 2^{h-1} * 2 \qquad + 2^h * 2$$

In every term except the first,  $2^i * 2$  is the common factor, and the  $(h-i+1)-(h-i)$  results in 1, so we have effectively taken out the multiplier that varied in  $h$  – except for the solitary first term

# Heapify: Sift downs

$$S = -2^0 * 2h + 2^1 * 2 + 2^2 * 2 + 2^3 * 2 + \dots + 2^{h-1} * 2 + 2^h * 2$$

Let's move the first term to the end, and from the other terms, extract 2 as a common factor:

$$S = 2(2^1 + 2^2 + 2^3 + \dots + 2^{h-1} + 2^h) - 2^0 * 2h$$

To the power series inside parentheses, we'll add a  $2^0$  term, and to compensate subtract it out:

$$\Rightarrow S = 2(2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{h-1} + 2^h - 2^0) - 2^0 * 2h$$

$$\Rightarrow S = 2(2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{h-1} + 2^h) - 2 * 2^0 - 2^0 * 2h$$

The power series inside the parenthesis is exactly  $n!!$  (slide 17)

$$\Rightarrow S = 2(n) - 2 * 2^0 - 2^0 * 2h$$

And since  $h$  is  $O(\log n)$ , the result is  $O(n)$

# What just happened?

So back when we first looked at heapify, it wasn't clear as to why it would run any faster than doing a bunch of inserts and sift ups. It just seems like we are doing a bunch of sift downs instead, and both sift ups and sift downs are  $O(\log n)$  time in the worst case. So how come heapify works out to  $O(n)$  running time?

The math proves it beyond doubt, but what is the intuition?

The intuition is that when you sift up, you always start from the lowest level. But the heapify sift downs don't have to deal with the leaf nodes at all, and the leaf nodes are half of the entire set of nodes!

So basically you are only sifting down about  $n/2$  nodes, while with insertions and sift ups, you would be working on  $n$  nodes. This is why heapify with sift downs beats building a heap with inserts/sift ups.