CSCI 136: Fall 2019
Handout Heapsort Example

# Heapsort

This handout goes through an example of how heapsort functions on the following array $A$:*

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |

Heapsort on an array of size $n$ works in two steps. First, `heapify()` is called on the array to turn it into a heap. Then, for $i = n - 1$ to 0, `remove()` is called, and the resulting element is stored in slot $i$; the heap is then considered to be one element smaller.

## 1 Heapify

As we discussed in class, we want to use Bottom-Up Heapify, as it runs in $O(n)$ time. To accomplish this, for $j = n-1$ to 0, we call `pushDownRoot(`$j$`)`. On the right we show what the array looks like *after* the corresponding step, with arrows to indicate any swaps that were performed *during* that step. We highlight in red the nodes from $j$ to $n - 1$; these nodes satisfy the heap property.

`pushDownRoot(14)`: The children of 14 are 29 and 30; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(13)`: The children of 13 are 27 and 28; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(12)`: The children of 12 are 25 and 26; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(11)`: The children of 11 are 23 and 24; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(10)`: The children of 10 are 21 and 22; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(9)`: The children of 9 are 19 and 20; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

`pushDownRoot(8)`: The children of 8 are 17 and 18; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

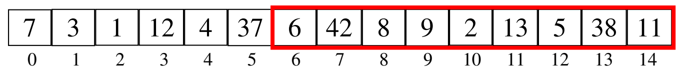
`pushDownRoot(7)`: The children of 7 are 15 and 16; both are not stored in the heap, so the heap property is satisfied.

| 7 | 3 | 1 | 12 | 4 | 37 | 6 | 42 | 8 | 9 | 2 | 13 | 5 | 38 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

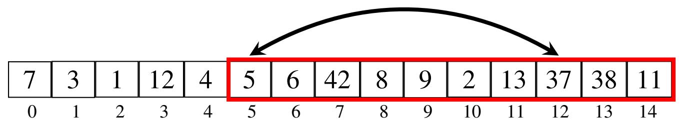
*We use an array in this example, but heapsort would work just as well on a vector.

A side note: at this point you may have noticed that we've wasted a lot of time looking at elements that do not have children. You're right! A good heapify implementation would skip right to elements with children: it would call `pushDownRoot(`$j$`)` for $j = (n-2)/2$ to 0.

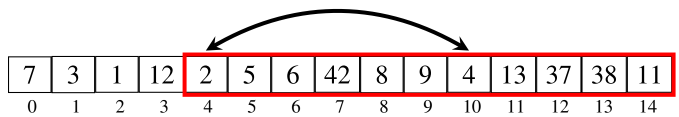
`pushDownRoot(6)`: The children of 6 are 13 and 14. $A[6] = 6$ is smaller than $A[13] = 38$ and $A[14] = 11$, so no swaps are necessary.



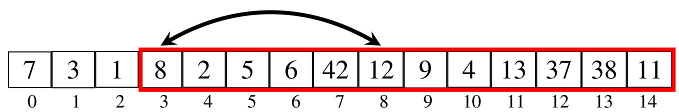

`pushDownRoot(5)`: The children of 5 are 11 and 12. $A[5] = 37$ is not smaller than $A[11] = 13$ (or $A[12] = 5$), so we need to swap. `Pushdown()` always swaps with the smaller child. Since $A[12] < A[11]$, we swap $A[5]$ with $A[12]$. The children of 12 are not in the heap so we are done.



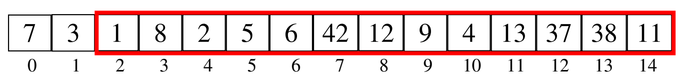

`pushDownRoot(4)`: The children of 4 are 9 and 10. $A[4] = 4$ is smaller than $A[9] = 9$, but is larger than $A[10] = 2$. We swap with the smaller child. The children of 10 are not in the heap so we are done.



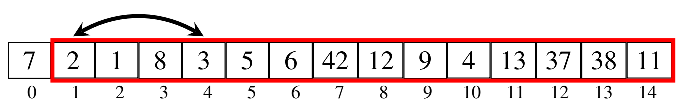

`pushDownRoot(3)`: The children of 3 are 7 and 8. $A[3] = 12$ is larger than $A[8] = 8$; we swap with the smaller child. The children of 8 are not in the heap so we are done.



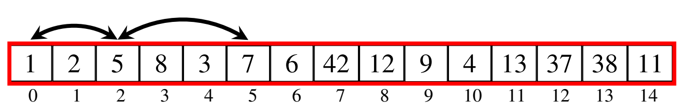

`pushDownRoot(2)`: The children of 2 are 5 and 6. $A[2]$ is smaller than $A[5]$ and $A[6]$, so we don't swap.




`pushDownRoot(1)`: The children of 1 are 3 and 4. $A[1]$ is larger than $A[4]$ and $A[4] < A[3]$ so we swap with $A[4]$. The children of 4 are 9 and 10. $A[4]$ is smaller than $A[9]$ and $A[10]$ so we don't swap.




`pushDownRoot(0)`: The children of 0 are 1 and 2. $A[0]$ is larger than $A[2]$ and $A[2] < A[1]$ so we swap with $A[2]$. The children of 2 are 5 and 6. $A[2] = 7$ is larger than $A[5] = 5$ and $A[5] < A[6] = 6$ so we swap with $A[5]$. After this, $A[5] < A[11]$ and $A[5] < A[12]$ so we are done.




After $O(n)$ calls to `pushDownRoot`, we have a heap! As we showed in class, this actually takes $O(n)$ total work in the worst case. Now we need to sort.

## 2 Sort By Repeatedly Removing the Minimum

Now we sort. As discussed in class, for $i = n-1$ to 0, we remove the minimum element of the heap and place it in $A[i]$. This is particularly easy for a heap, as the first step in `remove()` is to swap the first and last item in the heap. After that, `pushDownRoot(0)` is called.
So after round $i$, the $i$ smallest elements are in reverse-sorted order from $A[i]$ to $A[n-1]$; meanwhile $A[0]$ to $A[i-1]$ is a heap. Again we go round by round. We show the state of the array *after* each round, and show the swaps made during the round using arrows. Again, cells highlighted in red form a heap; cells highlighted in blue are in reverse-sorted order. We begin with the heap obtained from before:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 8 | 3 | 7 | 6 | 42 | 12 | 9 | 4 | 13 | 37 | 38 | 11 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Now we begin our calls to `remove`.

($i = 14$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$, $A[1]$ is swapped with $A[4]$, and $A[4]$ is swapped with $A[10]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 8 | 4 | 7 | 6 | 42 | 12 | 9 | 11 | 13 | 37 | 38 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 13$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$, $A[1]$ is swapped with $A[4]$, and $A[4]$ is swapped with $A[9]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 8 | 9 | 7 | 6 | 42 | 12 | 38 | 11 | 13 | 37 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 12$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$, $A[1]$ is swapped with $A[3]$, and $A[3]$ is swapped with $A[8]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 8 | 5 | 12 | 9 | 7 | 6 | 42 | 37 | 38 | 11 | 13 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 11$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[2]$ and $A[2]$ is swapped with $A[6]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 6 | 12 | 9 | 7 | 13 | 42 | 37 | 38 | 11 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 10$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[2]$ and $A[2]$ is swapped with $A[5]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 8 | 7 | 12 | 9 | 11 | 13 | 42 | 37 | 38 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 9$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[2]$ and $A[2]$ is swapped with $A[5]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 11 | 12 | 9 | 38 | 13 | 42 | 37 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

($i = 8$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$ and $A[1]$ is swapped with $A[4]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 9 | 11 | 12 | 37 | 38 | 13 | 42 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

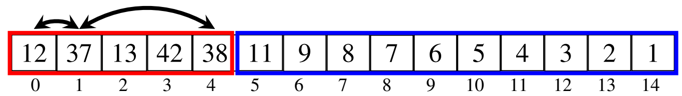
($i = 7$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$ and $A[1]$ is swapped with $A[3]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 12 | 11 | 42 | 37 | 38 | 13 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

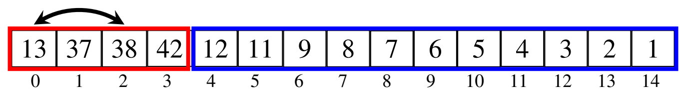
($i = 6$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[2]$.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 12 | 13 | 42 | 37 | 38 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

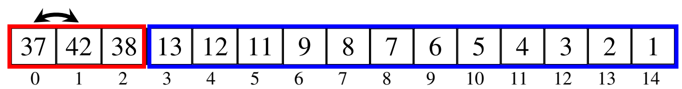
($i = 5$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$, and $A[1]$ is swapped with $A[4]$.



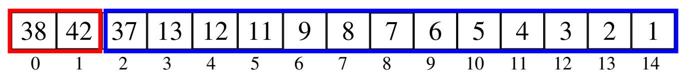

($i = 4$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[2]$.



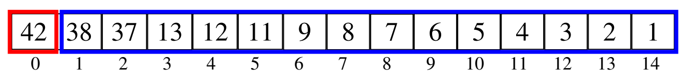

($i = 3$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. $A[0]$ is swapped with $A[1]$.



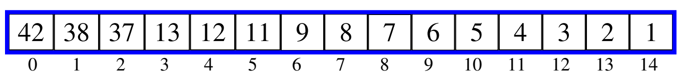

($i = 2$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. No swaps are needed.




($i = 1$): We swap $A[0]$ with $A[i]$ and call `Pushdown(0)`. No swaps are needed.




And we're done! Our array is in sorted order:




Well, it's in reverse sorted order. If we want, we can reverse it in $O(n)$ time.

**Side note:** There are techniques one can use to avoid this final step—for example, by using a "max heap," which is exactly like a heap but with the property that each node is *larger* than its children. Calling `remove()` on a max heap gives the largest node, ultimately resulting in a sorted array with no reversals needed.