CS-1203-2 : Monsoon 2023 : Quiz 1 - Take Home

Divyam Chaudhary : 1020221520 : divyam.chaudhary_ug25@ashoka.edu.in

Question 1

(a) For `heapInsert()` :

There is an outer loop which runs n (the number of elements in the inital array) number of times and calls a function `heapInsert()` every time. Since the outer loop uses an iterator variable `i` and the `heapInsert()` function defines its own local `i`, in the dry run below, I will refer to the outer loop's `i` as `i` (global) and the function's `i` as `i` (local).

Note if the value of a variable (represented in columns) doesn't change after a step (represented in rows) then, the dry run tables have a blank in the cell representing that variable and the current step. In such a case, the current value of that varibale is the most recent value from the first non-blank cell above the current cell in the column representing that varibale.

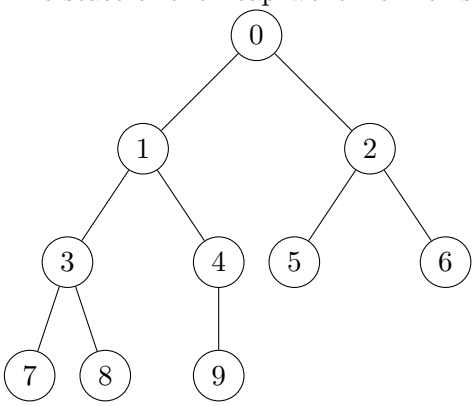
Initial state of data :
`arr[]` = $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
`n` = 10
$0 \leq$ `i` (global) $<$ `n`

→ For function call `heapInsert(arr, i, arr[i])`; where `i` = 0 and `arr[i]` = 0 :

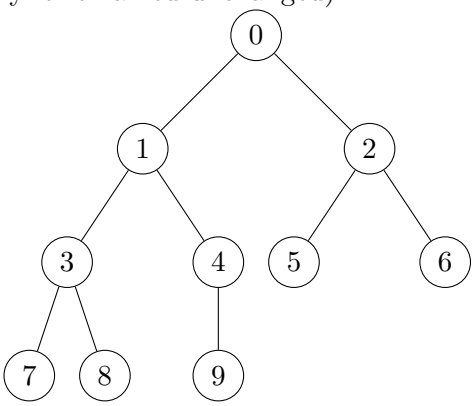
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
0	0			adding the new element at the
	0	0		bottom of the current heap
no loop triggered!				
——return call——				

After this function call, the total number of comparasions up until now are 0, and swaps are 0.

The state of the heap went from this (basically it remained unchanged) :



to this,

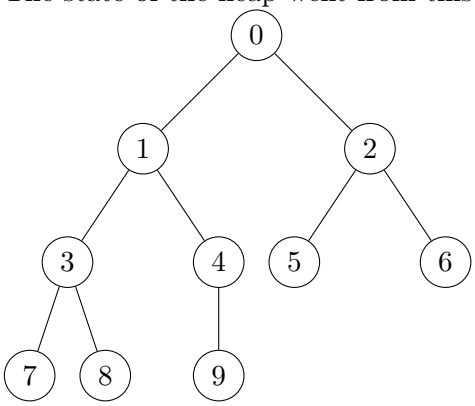


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 1` and `arr[i] = 1` :

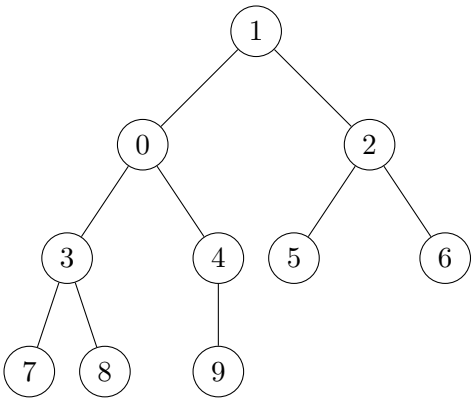
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
1	1			adding the new element at the
	1	1		bottom of the current heap
			0	1st iteration
after swap → `arr[`1, 0, 2, 3, 4, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 1, and swaps are 1.

The state of the heap went from this :



to this,

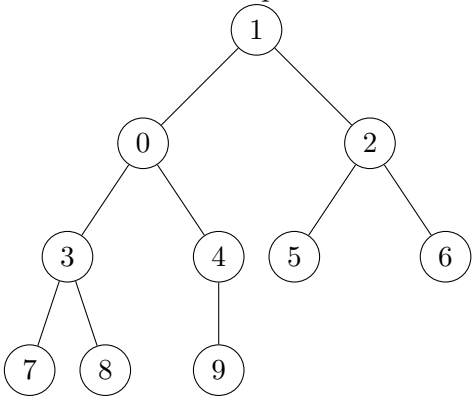


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 2` and `arr[i] = 2` :

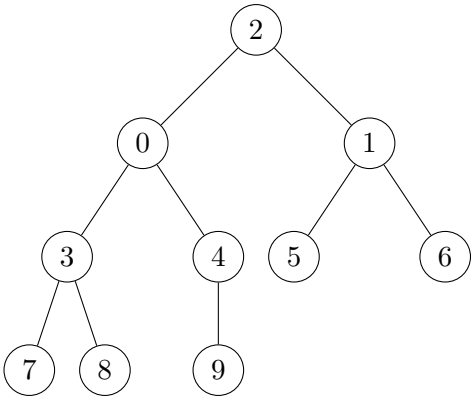
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
2	2			adding the new element at the
	2	2		bottom of the current heap
			0	1st iteration
after swap → `arr[`2, 0, 1, 3, 4, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 2, and swaps are 2.

The state of the heap went from this :



to this,

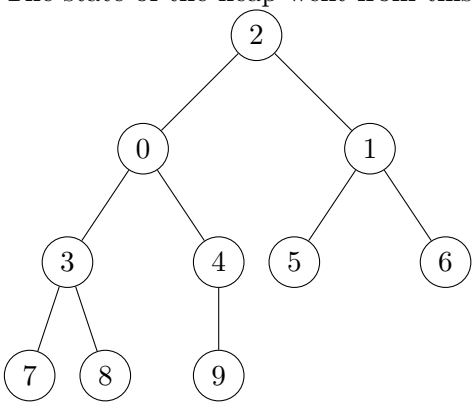


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 3` and `arr[i] = 3` :

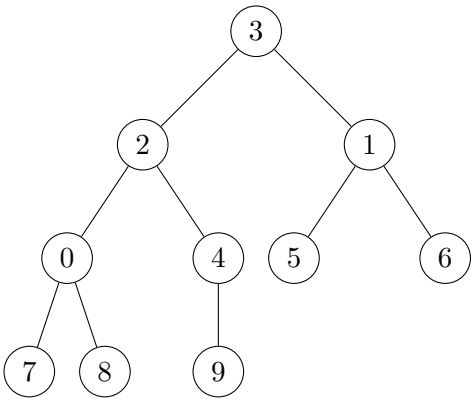
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
3	3			adding the new element at the
	3	3		bottom of the current heap
			1	1st iteration
after swap → `arr[`2, 3, 1, 0, 4, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		1		
			0	2nd iteration
after swap → `arr[`3, 2, 1, 0, 4, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 4, and swaps are 4.

The state of the heap went from this :



to this,

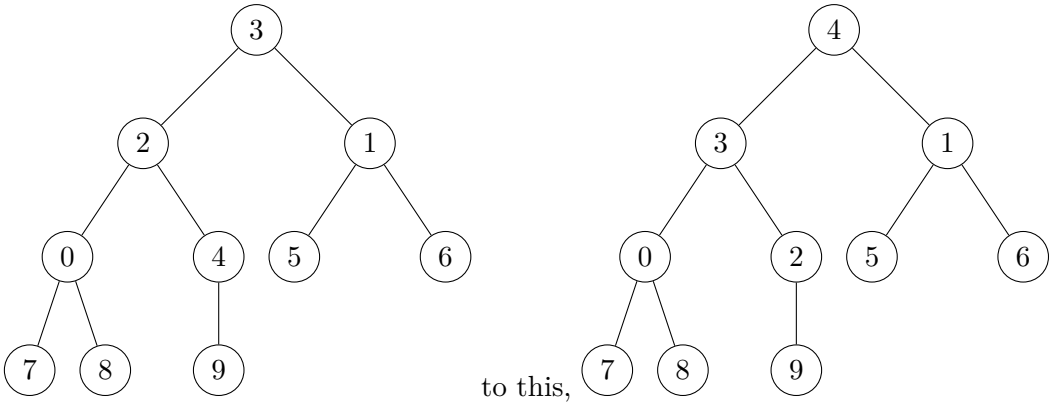


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 4` and `arr[i] = 4` :

`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
4	4			adding the new element at the
	4	4		bottom of the current heap
			1	1st iteration
after swap → `arr[`3, 4, 1, 0, 2, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		1		
			0	2nd iteration
after swap → `arr[`4, 3, 1, 0, 2, 5, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 6, and swaps are 6.

The state of the heap went from this :

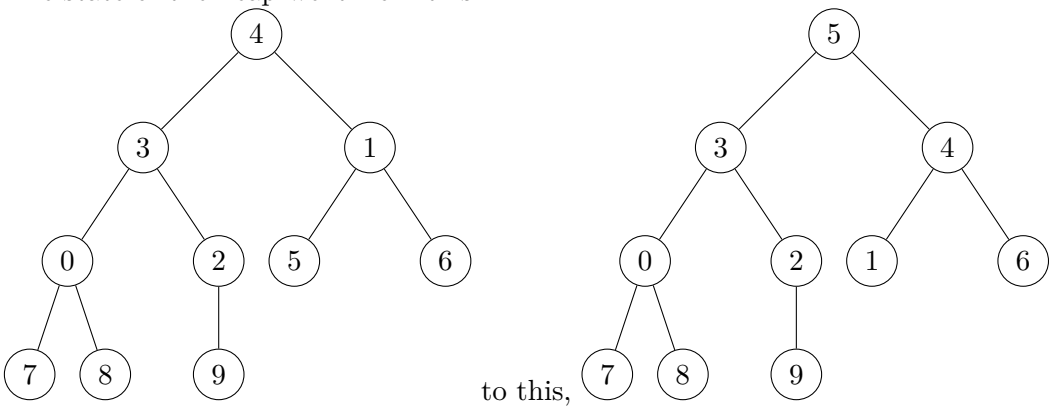


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 5` and `arr[i] = 5` :

`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
5	5			adding the new element at the
	5	5		bottom of the current heap
			2	1st iteration
after swap → `arr[`4, 3, 5, 0, 2, 1, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		2		
			0	2nd iteration
after swap → `arr[`5, 3, 4, 0, 2, 1, 6, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 8, and swaps are 8.

The state of the heap went from this :

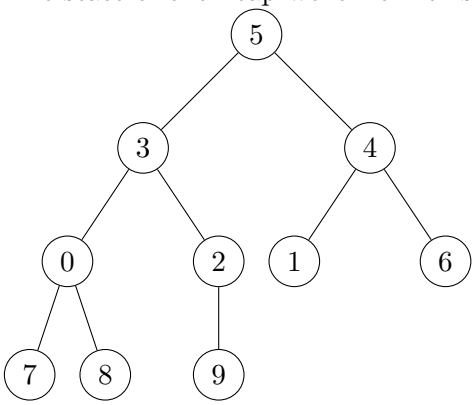


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 6` and `arr[i] = 6` :

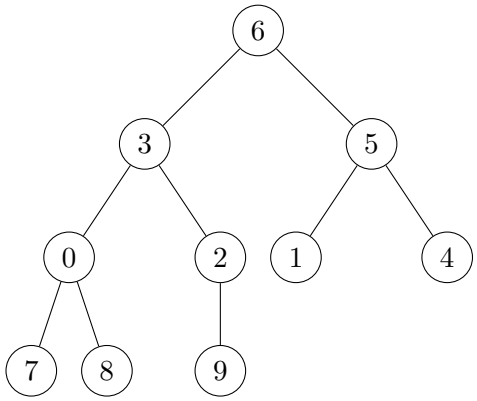
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
6	6			adding the new element at the
	6	6		bottom of the current heap
			2	1st iteration
after swap → `arr[`5, 3, 6, 0, 2, 1, 4, 7, 8, 9`]`				comparasions = 1; swaps = 1
		2		
			0	2nd iteration
after swap → `arr[`6, 3, 5, 0, 2, 1, 4, 7, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 10, and swaps are 10.

The state of the heap went from this :



to this,

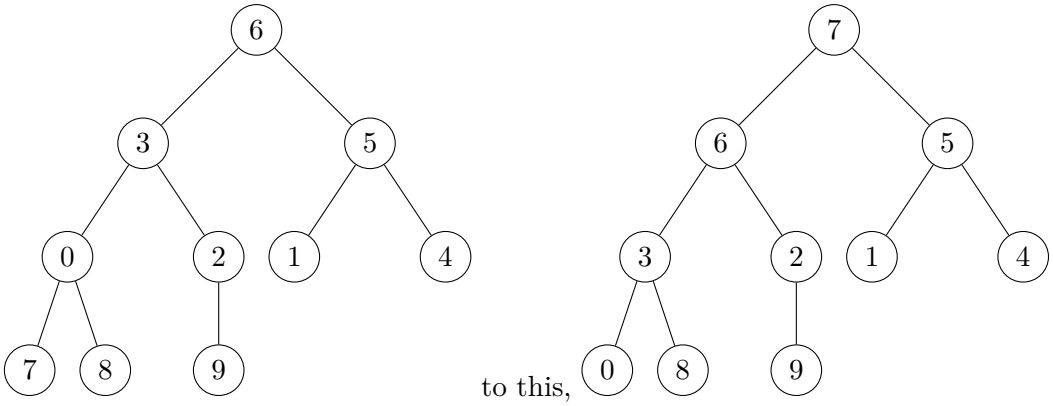


→ For function call `heapInsert(arr, i, arr[i])`; where `i = 7` and `arr[i] = 7` :

`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
7	7			adding the new element at the
	7	7		bottom of the current heap
			3	1st iteration
after swap → `arr[`6, 3, 5, 7, 2, 1, 4, 0, 8, 9`]`				comparasions = 1; swaps = 1
		3		
			1	2nd iteration
after swap → `arr[`6, 7, 5, 3, 2, 1, 4, 0, 8, 9`]`				comparasions = 1; swaps = 1
		1		
			0	3rd iteration
after swap → `arr[`7, 6, 5, 3, 2, 1, 4, 0, 8, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 13, and swaps are 13.

The state of the heap went from this :

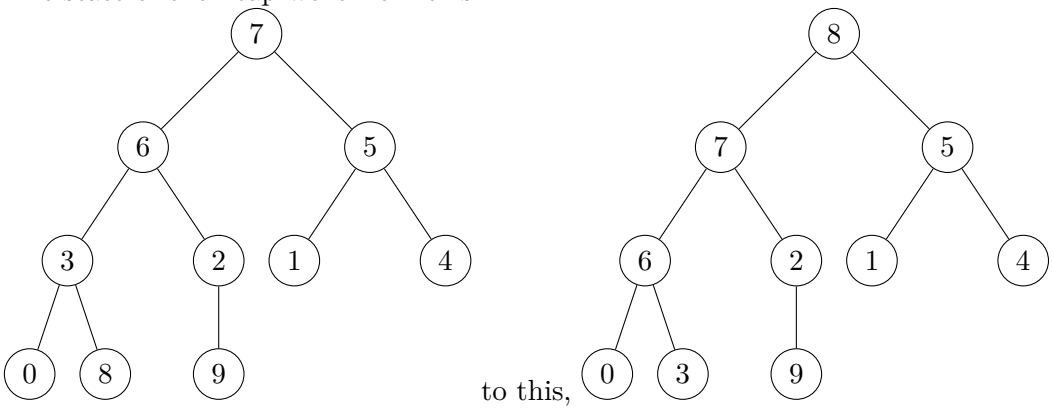


→ For function call `heapInsert(arr, i, arr[i])`; where `i` = 8 and `arr[i]` = 8 :

`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
8	8			adding the new element at the
	8	8		bottom of the current heap
			3	1st iteration
after swap → `arr[`7, 6, 5, 8, 2, 1, 4, 0, 3, 9`]`				comparasions = 1; swaps = 1
		3		
			1	2nd iteration
after swap → `arr[`7, 8, 5, 6, 2, 1, 4, 0, 3, 9`]`				comparasions = 1; swaps = 1
		1		
			0	3rd iteration
after swap → `arr[`8, 7, 5, 6, 2, 1, 4, 0, 3, 9`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 16, and swaps are 16.

The state of the heap went from this :

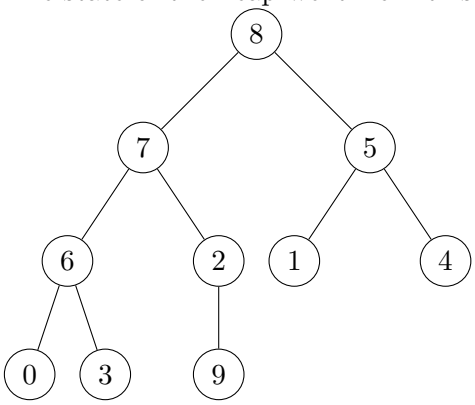


→ For function call `heapInsert(arr, i, arr[i])`; where `i` = 9 and `arr[i]` = 9 :

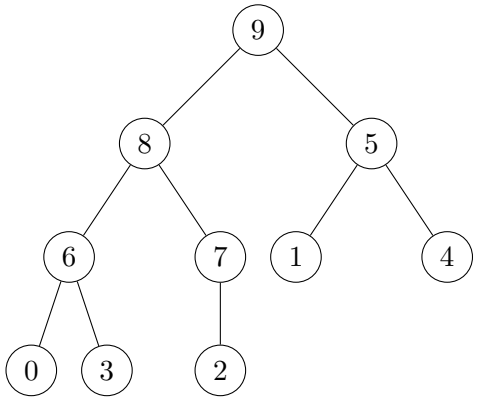
`i` (global)	`arr[`i (global)`]`	i (local)	parent	Comments
9	9			adding the new element at the
	9	9		bottom of the current heap
			4	1st iteration
after swap → `arr[`8, 7, 5, 6, 9, 1, 4, 0, 3, 2`]`				comparasions = 1; swaps = 1
		4		
			1	2nd iteration
after swap → `arr[`8, 9, 5, 6, 7, 1, 4, 0, 3, 2`]`				comparasions = 1; swaps = 1
		1		
			0	3rd iteration
after swap → `arr[`9, 8, 5, 6, 7, 1, 4, 0, 3, 2`]`				comparasions = 1; swaps = 1
		0		
——return call——				

After this function call, the total number of comparasions up until now are 19, and swaps are 19.

The state of the heap went from this :



to this,



At this point, the outer loop terminates and we have a valid heap stored in `arr[]`. And the total number of comparasion it took to get here were 19, whereas swaps were also 19.

The total number of comparasions = the total number of swaps =

$$\begin{aligned}
&\lfloor log_2(1) \rfloor + \lfloor log_2(2) \rfloor + \lfloor log_2(3) \rfloor + \lfloor log_2(4) \rfloor + \lfloor log_2(5) \rfloor + \lfloor log_2(6) \rfloor \\
&+ \lfloor log_2(7) \rfloor + \lfloor log_2(8) \rfloor + \lfloor log_2(9) \rfloor + \lfloor log_2(10) \rfloor \\
&= 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 \\
&= 19
\end{aligned}$$

Using this method, one new element is added to the heap in $log(n)$ time (for simplicity's sake, just ignore the lower bounds above), and we are adding n elements, so the worst case time complexity of `heapInsert()` is $O(n\ log(n))$.

(*b*) For `heapify()` :

There is an outer loop which runs $\sim log(n)$ (n is the number of elements in the inital array) number of times and calls a function `heapify()` every time. Here the function takes in the value of i from the outer loop but modifies it locally hence the value of i changes in the below dry run tables.

Note if the value of a variable (represented in columns) doesn't change after a step (represented in rows) then, the dry run tables have a blank in the cell representing that variable and the current step. In such a case, the current value of that varibale is the most recent value from the first non-blank cell above the current cell in the column representing that varibale.

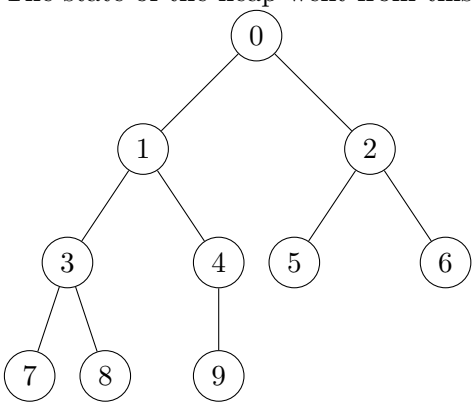
Initial state of data :
`arr[]` = $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
`n` = 10
$\frac{n}{2} - 1 \geq$ `i` $\geq$ `0`

→ For function call `heapify(arr, n, i)`; where `n` = 10 and `i` = 4 :

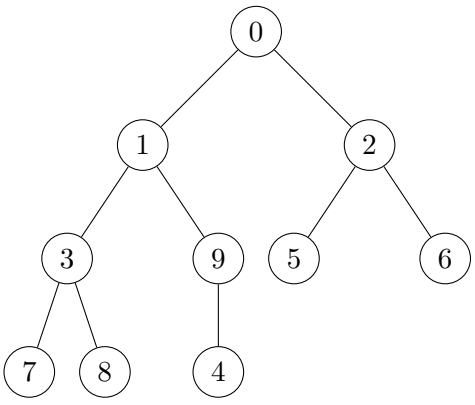
largest	left	right	i	n	Comments
4	9	10	4	10	1st iteration
9			9		
after swap → `arr`$[0, 1, 2, 3, 9, 5, 6, 7, 8, 4]$					comparasions = 1; swaps = 1
9	19	20			2nd iteration
——loop terminates and function returns——					

After this function call, the total number of comparasions up until now are 1, and swaps are 1.

The state of the heap went from this :



to this,

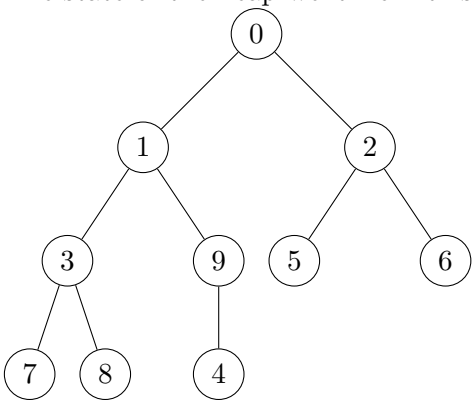


→ For function call `heapify(arr, n, i)`; where `n` = 10 and `i` = 3 :

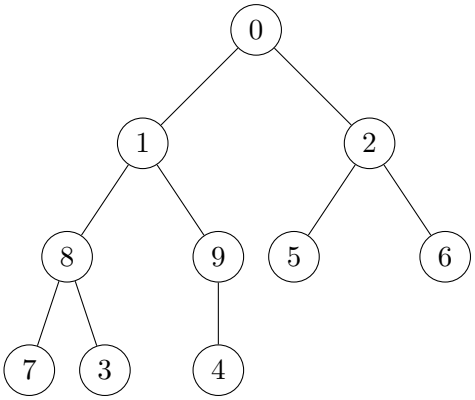
largest	left	right	i	n	Comments
3	7	8	3	10	1st iteration
7					
8			8		
after swap → `arr`$[0, 1, 2, 8, 9, 5, 6, 7, 3, 4]$					comparasions = 2; swaps = 1
8	17	18			2nd iteration
——loop terminates and function returns——					

After this function call, the total number of comparasions up until now are 3, and swaps are 2.

The state of the heap went from this :



to this,

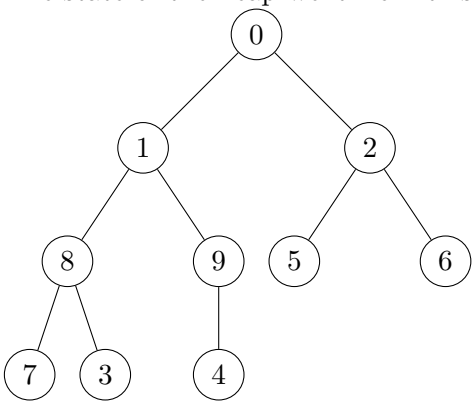


→ For function call `heapify(arr, n, i)`; where `n = 10` and `i = 2` :

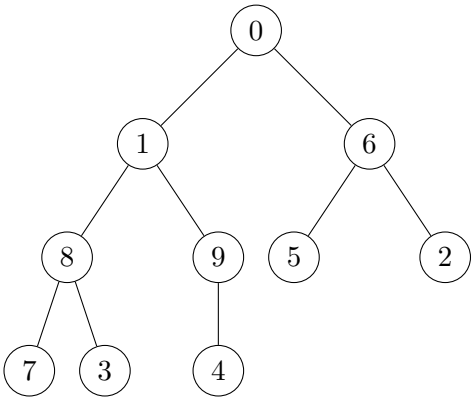
largest	left	right	i	n	Comments
2	5	6	2	10	1st iteration
5					
6			6		
after swap → `arr`$[0, 1, 6, 8, 9, 5, 2, 7, 3, 4]$					comparasions = 2; swaps = 1
6	13	14			2nd iteration
——loop terminates and function returns——					

After this function call, the total number of comparasions up until now are 5, and swaps are 3.

The state of the heap went from this :



to this,

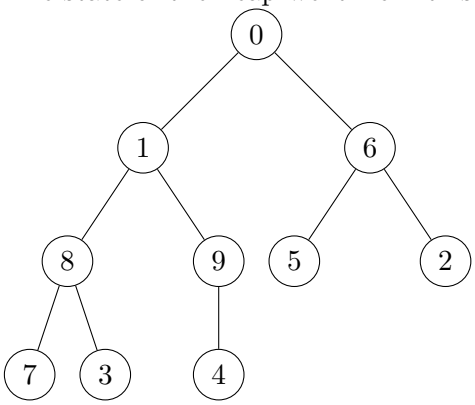


→ For function call `heapify(arr, n, i)`; where `n = 10` and `i = 1` :

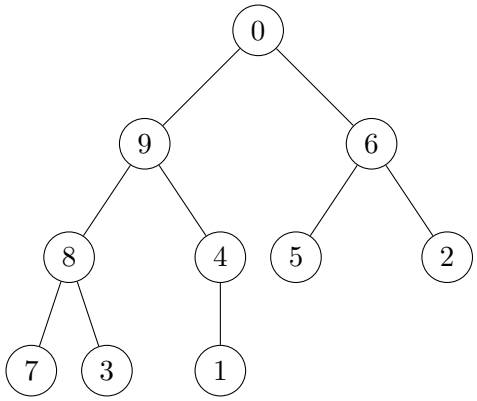
largest	left	right	i	n	Comments
1	3	4	1	10	1st iteration
3					
4			4		
after swap → `arr[0, 9, 6, 8, 1, 5, 2, 7, 3, 4]`					comparasions = 2; swaps = 1
4	9	10	4		2nd iteration
9			9		
after swap → `arr[0, 9, 6, 8, 4, 5, 2, 7, 3, 1]`					comparasions = 1; swaps = 1
9	19	20			3rd iteration
——loop terminates and function returns——					

After this function call, the total number of comparasions up until now are 8, and swaps are 5.

The state of the heap went from this :



to this,

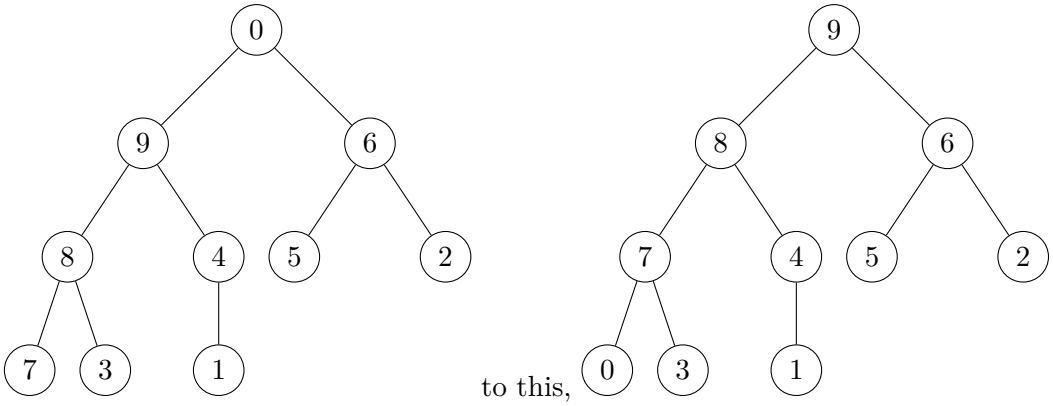


→ For function call `heapify(arr, n, i)`; where `n = 10` and `i = 0` :

largest	left	right	i	n	Comments
0	1	2	0	10	1st iteration
1			1		
after swap → `arr[9, 0, 6, 8, 4, 5, 2, 7, 3, 1]`					comparasions = 1; swaps = 1
1	3	4	1		2nd iteration
3			3		
after swap → `arr[9, 8, 6, 0, 4, 5, 2, 7, 3, 1]`					comparasions = 1; swaps = 1
3	7	8	3		3rd iteration
7			7		
after swap → `arr[9, 8, 6, 7, 4, 5, 2, 0, 3, 1]`					comparasions = 1; swaps = 1
7	15	16			4th iteration
——loop terminates and function returns——					

After this function call, the total number of comparasions up until now are 11, and swaps are 8.

The state of the heap went from this :



At this point, the outer loop terminates and we have a valid heap stored in `arr[]`. And the total number of comparasion it took to get here were 11, whereas swaps were also 8.

The total number of comparasions =

$$\begin{aligned} &\lfloor log_2(10) \times log_2(10) \rfloor \\ &= \lfloor 11.0352062676 \ldots \rfloor \\ &= 11 \end{aligned}$$

Using this method, in the average case, n new elements are added to the heap in $log(n)$ time. But since each new element we are providing is the largest one yet in the heap, it needs to be moved up in $log(n)$ time as well, hence this algorithm defaults to the worst case which runs in $log(n) \times log(n)$ time $\sim n$. Thus, the worst case time complexity of `heapify()` is $O(n)$.

Note, the number of swaps here is a little less than the number of comparasions, as in the `heapify()`'s inner loop we compare an element with both the sub-nodes of a given node but are only swapping it with one of the two sub nodes.

* * *