ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 7.2.
Assistances and comments will be acknowledged.

7.2 Binary Min-Heaps

A *heap* is a tree-based structure, but it doesn't use the binary-search differentiation between the left and right sub-trees to create a linear ordering. Instead, a binary heap only specifies the relationship between a parent and its children. For a *min-heap*, a node must be less than all of its children and it, in turn, must be greater than its parent (if any). Thus, a *binary min-heap* is a binary tree that satisfies the *min-heap* property.

Important: there is no other relationship between the children of a node other than they are all greater than their common parent. The failure to comprehend this has previously been the greatest source of errors.

Thus, it is reasonable to compare binary search trees and binary heaps as is shown in Table 1.

Table 1. Properties of binary search trees versus binary heaps.

Binary Search Tree	Binary Heap
Given a node, all objects in the left sub-tree are less than the node, all objects in the right sub-tree are greater than the node, and both sub-trees are also binary search trees.	Given a node, all strict descendants are greater than the node, and both sub-trees are also binary heaps.

Normally, for a priority queue, you would store objects and their associated priorities. For simplicity, our heaps will store only the priority—the associated object with that priority could be integrated into the heaps that we will be implementing.

Figure 1 gives an example of a heap. You will notice that the four largest values are in the left sub-tree which also contains the next larger value from 3.

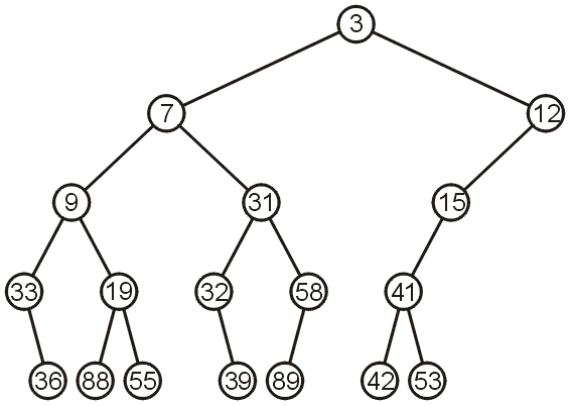


Figure 1. A binary heap.

7.2.1 Operations

The operations we will look at include *top*, *pop*, and *push*.

7.2.1.1 Top

Finding the smallest entry in Figure 1 is an $\Theta(1)$ operation—just return the root node.

7.2.1.2 Pop

Removing the top element could be implemented as easily as removing the top (in this case, 3) and then recursively promoting the smaller of the children. Thus, removing 3 from Figure 1 results in the heap shown in Figure 2. Because we are always promoting the smaller of the two children, we are guaranteed that the min-heap structure is maintained.

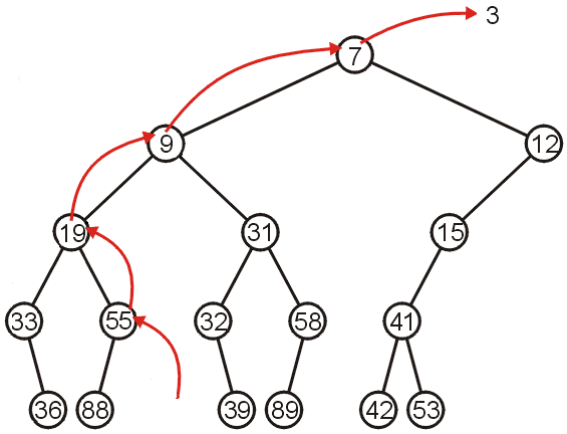


Figure 2. The binary min-heap resulting from a pop operation being performed on the heap in Figure 1.

Removing the minimum from the heap in Figure 2 produces the heap in Figure 3.

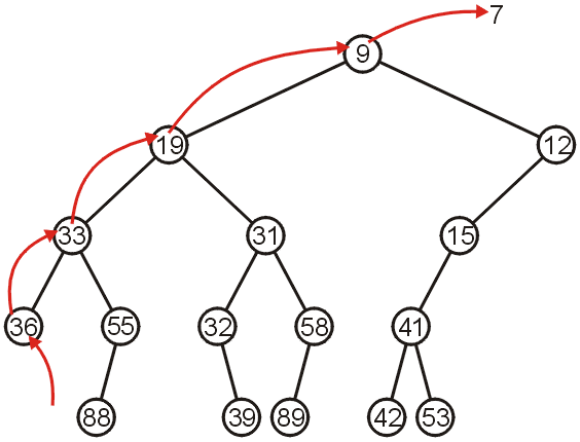


Figure 3. The binary min-heap resulting from a pop operation being performed on the heap in Figure 2.

Finally, Figure 4 shows the result of popping 9 from the binary min-heap in Figure 3.

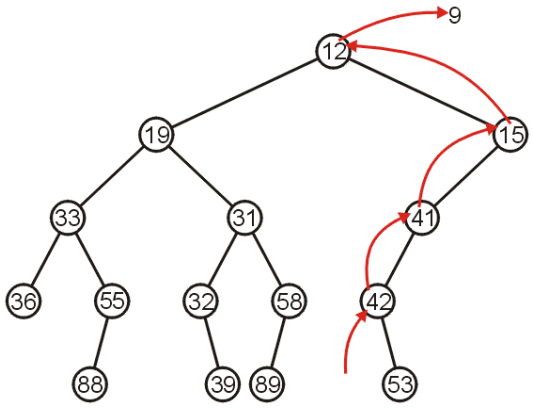


Figure 4. The binary min-heap resulting from a pop operation being performed on the heap in Figure 3.

7.2.1.3 Push

When we insert a new object into a min-heap, we have one of two options:

1. Insert the object at the root, or
2. Insert the object at an empty node.

In the first case, we would recursively apply a rule such as *replace the node with minimum of the current node and the value being inserted and insert the larger of the two into one of the two sub-heaps*.

In the second case, you would recursively apply a rule such as *swap the inserted node with its parent until the parent is less than the inserted value or we are at the root.*

We will look at a few examples of the second option. Inserting 17 into the tree in Figure 4 could occur at any location, so we will choose one arbitrary empty node. We compare it with its parent, 32, and swap these two, for 17 < 32. We then examine the new parent of 17 and notice 31 > 17, so we continue swapping. We continue until finally 12 < 17. This process is called *percolating up* (similar to the actions of a coffee percolator). Figure 5 shows this percolation.

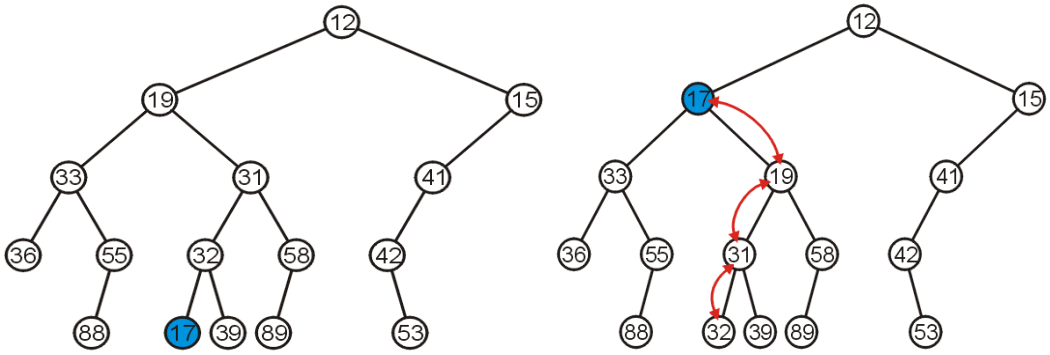


Figure 5. Inserting 17 at an arbitrary location and percolating it up.

One issue with such an insertion is: how do we know where the empty nodes are located? We will solve this in the following implementation.

7.2.2 Complete-tree Implementation

There are numerous data structures that implement min-heaps, including using a complete binary tree, leftist heaps, skew heaps, binomial heaps, and Fibonacci heaps. While some of the run-time characteristics of using complete binary trees are sub-optimal as compared to others, it is most efficient with respect to memory usage.

If you recall the definition of a *complete binary tree* in Topic 4.6, you will recall the entries of a complete tree are filled in breadth-first traversal order. Figure 6 shows a min-heap that contains the same nodes as Figure 5 but in the structure of a complete tree.

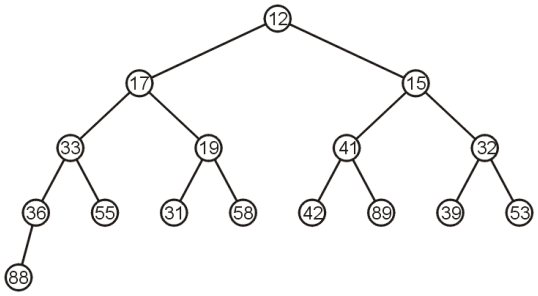


Figure 6. The min-heap of Figure 5 rearranged to satisfy the shape of a complete tree.

At this point, determining where one can insert a new node is trivial: there is only one place that a new can be inserted: to the right of 88. Suppose we are inserting 25. We would insert 25 at this location and then percolate it up to the appropriate location within the heap.

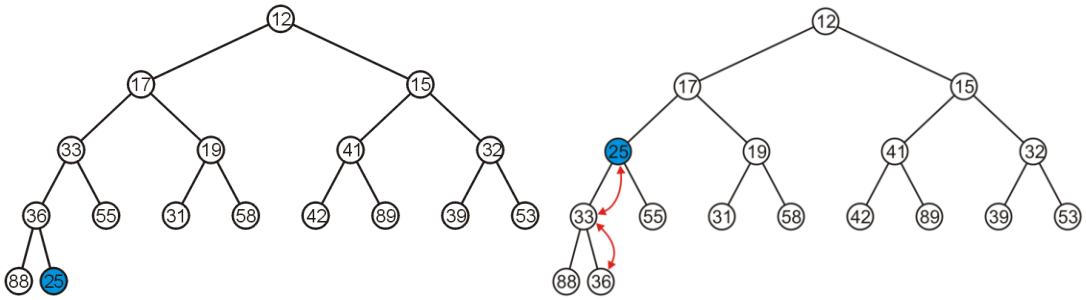


Figure 7. The insertion of and percolating up of 25 into the min-heap in Figure 6.

Thus, the use of a complete tree makes the implementation of Option 2 for performing pushes into a binary min-heap described in §7.2.1.3 straight-forward. Unfortunately, if we use the same rule for performing a pop, as described in §7.2.1.2, we run into a problem. Suppose we pop the top off of the tree in Figure 7. In order to maintain the min-tree structure, we would promote 15, 32, 39, and this would leave a gap in the tree as shown in Figure 8.

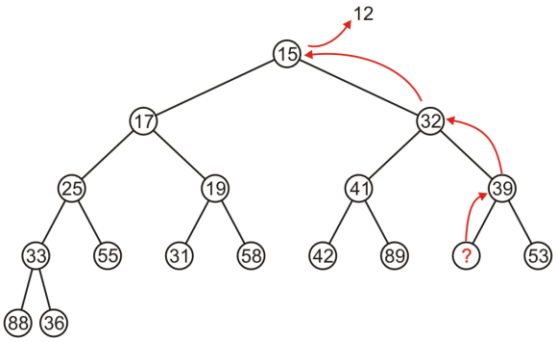


Figure 8. Popping the top off of the heap in Figure 7.

With the empty node at that location, the tree is no longer complete. Instead, consider the following strategy for popping the top:

> Replace the root node with the last entry in the complete tree. Then proceed to percolate that value down into the heap until the resulting tree is again a min heap.

For example, in Figure 9, we replace 12 with the last entry, 36, and then we proceed to swap 36 with the minimum of the two children until 36 is greater than both its children.

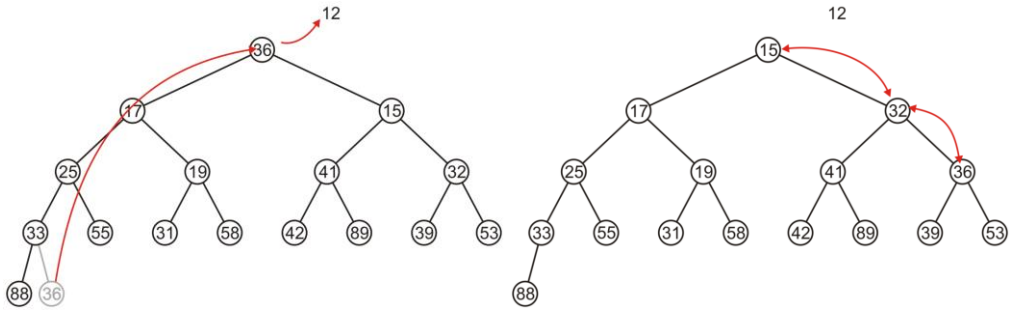


Figure 9. Popping the top of Figure 7 by replacing 12 with the last entry and percolating it down until it is greater than both its children.

This maintains the complete tree structure. As another example, suppose we pop the top again. Replace 15 with the last entry, 88, and percolate 88 down until it is less than both its children. In this case, we continue percolating until 88 ends up in a leaf node, as shown in Figure 10.

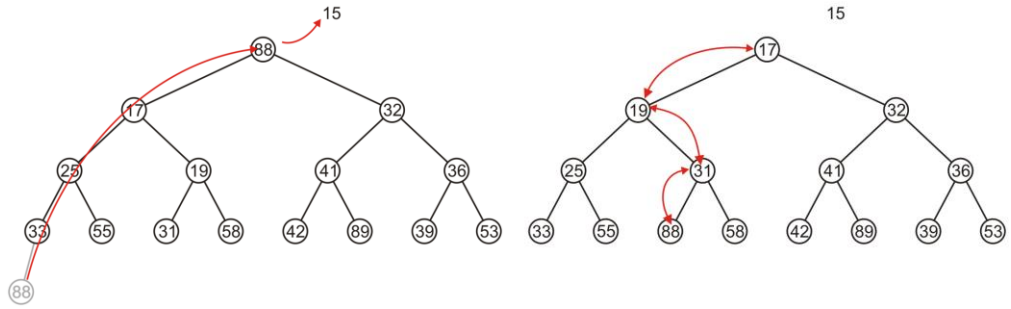


Figure 10. Popping the top from the final min-heap in Figure 9.

Two further pops are shown in Figures 11 and 12.

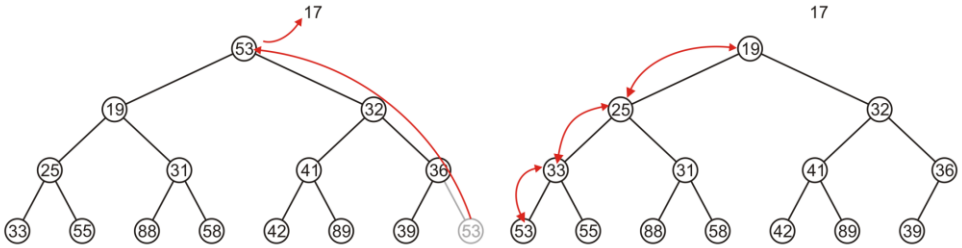


Figure 11. Popping the top from the final tree in Figure 10.

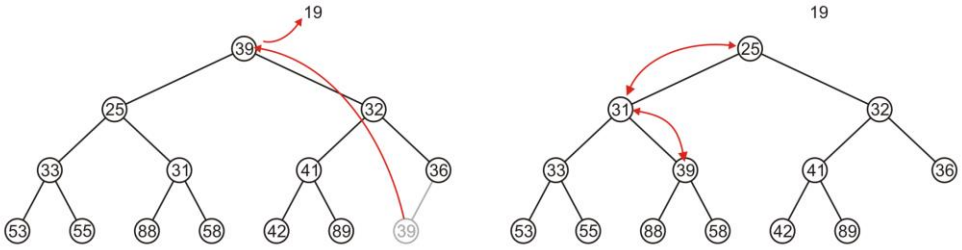


Figure 12. Popping the top from the final tree in Figure 11.

Thus, we can maintain the min-heap in the shape of a complete tree with both push and pop operations. We will now proceed to consider how we can implement a binary min-heap using an array.

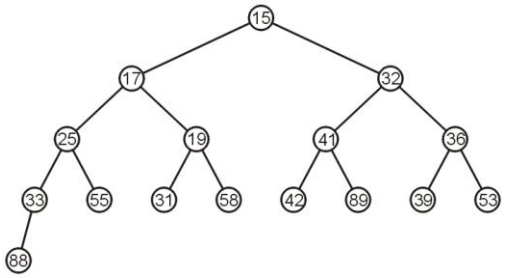
7.2.3 Array Implementation of a Binary Min-Heap

In §4.6.3, we discussed how we could store a complete tree in an array by using the entries 1 through n. In this case, the children of the object at index k are located in indices $2k$ and $2k + 1$ while the parent is located at index $k/2$.

ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 7.2.
Assistances and comments will be acknowledged.

7.2.3.1 Example 1

For example, the tree resulting in Figure 9 would be stored in the array shown in Figure 13.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

Figure 13. The complete tree stored as an array using array indices 1 through 16.

Thus, for example, we see that:

- The children of the root, 15 at the root, are located at indices 2 and 3 (17 and 32),
- The children of 17 (index 2) are located at indices 4 and 5 (25 and 19), and
- The children of 19 (index 5) are located at indices 10 and 11 (31 and 58).

If we wanted to pop the minimum element, we would remove 15 and replace it with 88:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	88	17	32	25	19	41	36	33	55	31	58	42	89	39	53	~~88~~

We would then compare 88 with its children in entries 2 and 3. Of these, 17 < 32 < 88 so we swap 88 and 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	**88**	32	25	19	41	36	33	55	31	58	42	89	39	53	

Again, we compare 88 and its two children at entries 4 and 5 and determine 19 < 25 < 88, and therefore we swap 88 and 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	19	32	25	**88**	41	36	33	55	31	58	42	89	39	53	

Again, we compare 88 and its two children at entries 10 and 11 and determine 31 < 58 < 88, and therefore we swap 31 and 88.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	19	32	25	31	41	36	33	55	88	58	42	89	39	53	

At this point, 20 and 21 > 15, and thus, 88 has no children to compare to, and thus we are left with a min heap in the shape of a complete tree stored as an array. This results in the complete tree shown in Figure 9.

7.2.3.2 Example 2

Consider the binary min-heap shown in Figure 14.

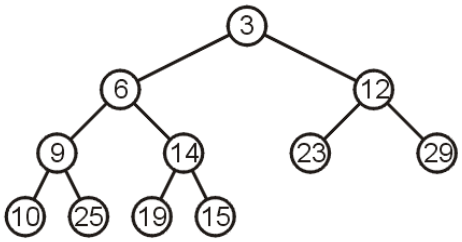


Figure 14. A binary min-heap in the shape of a complete tree.

The array representation of this complete would be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	12	9	14	23	29	10	25	19	15					

7.2.3.2.1 Two Insertions (Pushes) into Figure 14

If we were to insert 26, we would first place 26 into the next available index, 12, and compare 26 with its parent, 23 at location 12/2 = 6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	12	9	14	23	29	10	25	19	15	**26**				

Because 26 > 23, we leave 26 in index 12, as shown in Figure 15.

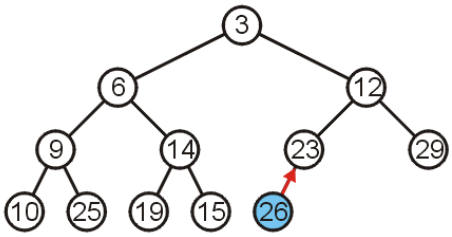


Figure 15. Inserting 26 into the min-heap in Figure 14.

Suppose now we insert 8 in the min heap in Figure 14. It would first be placed into index 13, it would then be compared to the entry 23 at index 13/2 = 6 and because 8 < 23, we would swap them. We would then compare 8 to the entry 12 at 6/2 = 3 and because 8 < 12, we would swap them. Finally, we note that 8 is greater than its parent, the root, so we stop.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	**8**	9	14	**12**	29	10	25	19	15	26	**23**			

The resulting tree is shown in Figure 16.

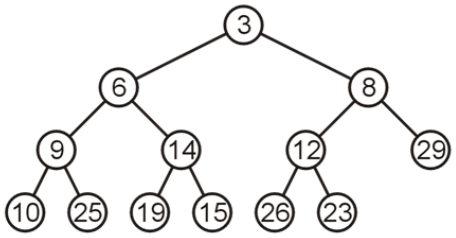


Figure 16. The tree in Figure 15 after pushing the value 8.

7.2.3.2.1 Three Removals (Pops) From Figure 16

To pop the top off of Figure 16, we remove 3 and copy 23 to entry 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	23	6	8	9	14	12	29	10	25	19	15	26	~~23~~			

We now percolate 23 down by comparing it with its children: $6 < 8 < 23$, so we swap 23 and 6:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	**23**	8	9	14	12	29	10	25	19	15	26				

We compare 23 with its two children at 4 and 5 and swap 23 and 9:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	**9**	8	**23**	14	12	29	10	25	19	15	26				

We compare 23 with its two children at 8 and 9 and swap 23 and 10:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	9	8	**10**	14	12	29	**23**	25	19	15	26				

A this point, $2 \cdot 8 = 16$ and $2 \cdot 8 + 1 = 17$ are both greater than 12 (the last entry), so we are finished.

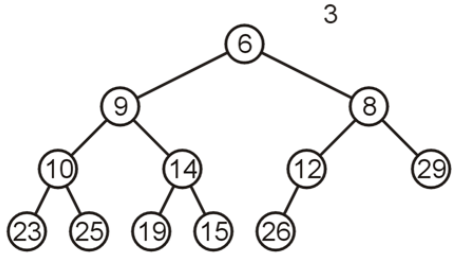


Figure 17. The result of popping the top of Figure 16.

If we pop the minimum of Figure 17, we remove 6 from the tree and replace it with 26:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	26	9	8	10	14	12	29	23	25	19	15	~~26~~				

ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 7.2.
Assistances and comments will be acknowledged.

Next, we compare it with its children and swap 26 and 8:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	8	9	**26**	10	14	12	29	23	25	19	15					

We compare 26 with its children 12 and 29 and swap 26 with 12:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	8	9	**12**	10	14	**26**	29	23	25	19	15					

A this point, $2 \cdot 6 = 12$ and $2 \cdot 6 + 1 = 13$ are greater than 11 (the last entry), so we are finished. The result is in Figure 18.

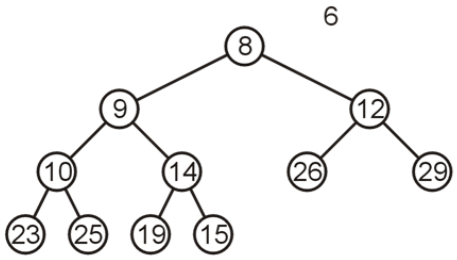


Figure 18. The result of popping the top of Figure 17.

Performing one final pop, we remove 8 and replace it with 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	15	9	12	10	14	26	29	23	25	19	~~15~~					

We compare it with its children and swap 9 and 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	9	**15**	12	10	14	26	29	23	25	19						

We compare 15 with its children 10 and 14 and swap 15 and 10:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	9	**10**	12	**15**	14	26	29	23	25	19						

We compare 15 with its children 23 and 25 (at indices 8 and 9) and realize 15 < 23 < 25, so we are finished. Figure 19 shows the resulting min heap.

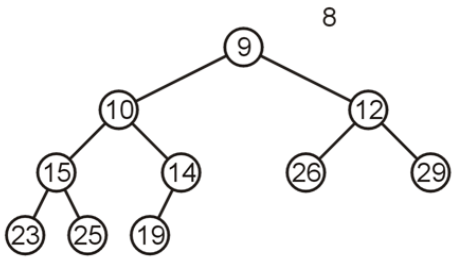


Figure 19. The result of popping the top from Figure 18.

7.2.4 Run-time Analysis

Accessing the minimum is accessing the 2nd array entry: $\Theta(1)$.

When we pop the top object, we replace it with an object at the lowest depth—it is very likely that this object will be percolated down to the bottom again, and thus, we may assume the run time will be $O(\ln(n))$.

How about push? Intuitively, it may appear that the average run time is $O(\ln(n))$ and if the object being inserted is less than the top, then the run time will be $\Theta(\ln(n))$. However, if we assume that a new object could occur at any location, what is the average run time assuming a random insertion?

We have already seen that at least half the entries of a complete tree are leaf nodes, as is demonstrated in Figure 20.

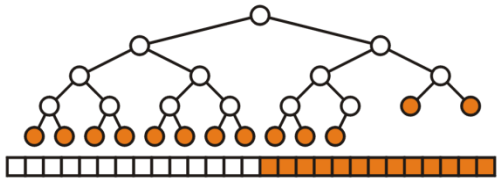


Figure 20. A complete tree noting the leaf nodes.

To simplify the analysis, let us assume that the heap is in the shape of a perfect tree and a balanced distribution of values in the heap. In this case, any object that is inserted that has a priority lower than the mean priority will not percolate up even once. Those in the next quarter will percolate once, those in the next eighth will percolate twice, and so on. Therefore, on average, there are 2^k nodes that will require $h - k$ percolations where k runs from 0 to h.

Thus, we can compute the average using

$$\sum_{k=0}^{h}\left(\frac{1}{n}\cdot(h-k)\,2^k\right) = \frac{h}{n}\sum_{k=0}^{h}2^k - \frac{1}{n}\sum_{k=0}^{h}\left(k\,2^k\right)$$

$$= \frac{2^{h+1}-h-2}{n} = \frac{n-h-1}{n} = 1 - \frac{\lg(n+1)}{n} - \frac{1}{n} = \Theta(1)$$

Therefore, the average insertion time assuming a random distribution of incoming entries, is Θ(1). Thus, if we compare the three techniques, using multiple queues, an AVL tree, and binary heaps, these run times are shown in Table 2.

Table 2. Run times for various implementations of priority queues.

	Multiple (*m*) Queues	**AVL Tree**	**Binary Heap**
top	O(*m*)	Θ(ln(*n*))	Θ(1)
push	Θ(1)	Θ(ln(*n*))	Θ(1)
pop	O(*m*)	Θ(ln(*n*))	O(ln(*n*))

Merging two binary heaps is, however, expensive: Θ(*n*). It should be noted that other data structures such as leftist, skew, binomial, and Fibonacci heaps have better run-time characteristics but are node based and therefore require Θ(*n*) additional memory. For Fibonacci heaps, all run times (including merging two Fibonacci heaps) except for pop are Θ(1).

7.2.5 Binary Max-Heaps

A binary max-heap is one that is ordered so that a node is always greater than all of its strict descendants. All other operations are similar to that of a binary min-heap.

7.2.6 Implementing stable priority queues

Suppose that the values A, B, C, D, E, F, and G are inserted into a binary min-heap in that order and assume that they all have the same priority. Thus, the heap, as represented by an array, will be represented by the heap in Figure 21.

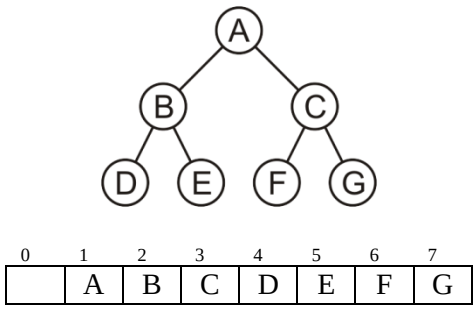


Figure 21. A heap after inserting seven objects at equal priority.

If you then pop these, they come out in the order

A, G, F, E, D, C, B

This is not the order in which these items were entered into the priority queue. Thus, a priority queue will allow a pop of an entry that has the highest priority; however, if there are multiple objects in the queue that have equal priorities, there is no guarantee that the object popped will be that object at the highest priority that has been in the priority queue the longest.

To fix this, we need an augmented priority:

Let $c = 0$ be a counter that is initialized with the creation of the priority queue. With each insertion of an item at priority p, it will be given the lexicographical priority (p, c) and then c will be incremented. Thus, if A and B are inserted with priorities p_A and p_B, respectively, and A was inserted before B, then the lexicographical priorities are (p_A, c_A) and (p_B, c_B), respectively, where $c_A < c_B$.

Thus, we have the possibilities:

1. if A has higher priority, $p_A < p_B$, it follows that $(p_A, c_A) < (p_B, c_B)$,
2. if B has higher priority, $p_A > p_B$, it follows that $(p_A, c_A) > (p_B, c_B)$, and
3. if A and B have equal priority, then $(p_A, c_A) < (p_B, c_B)$ because $c_A < c_B$.

Suppose A_2, B_3, C_3, D_2, E_1, F_2, G_2 are inserted in that order into a binary min-heap where the subscript indicates the priority. Thus, the resulting binary heap together with the result of popping the top seven times is shown in Figure 22. The order in which these are popped is

E_1, A_2, D_2, F_2, G_2, B_3, and C_3.

Thus, a priority queue using a lexicographical ordering where the second parameter is the insertion number will pop objects at equal priority in the same order in which they were entered: FIFO.

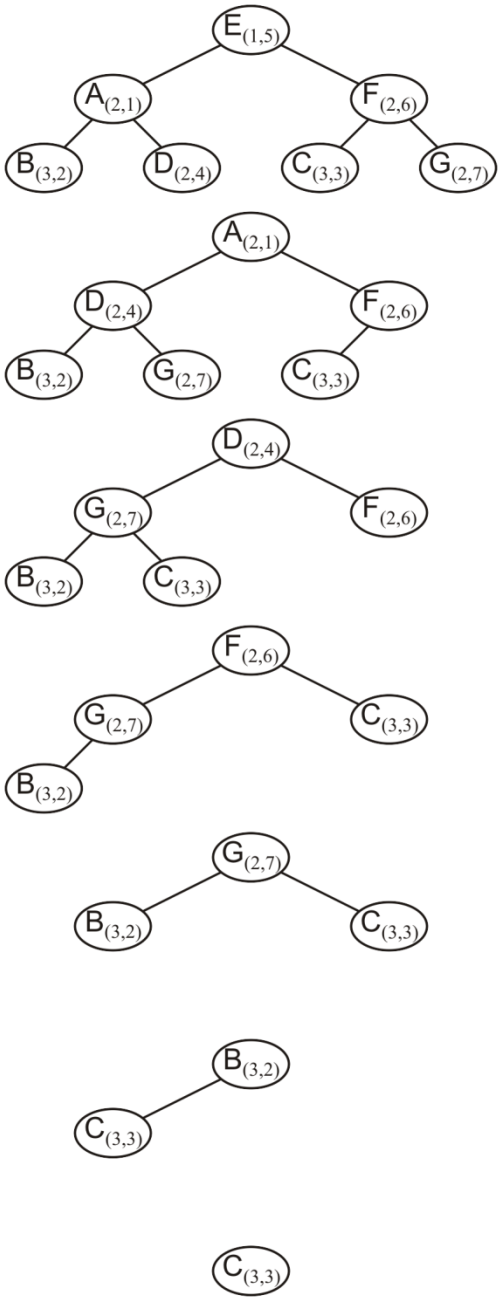


Figure 22. The binary heap resulting from the insertion and subsequent popping of seven entries, some with equal priority.