More Examples of Operations on Priority Queue

Tony Gong

ITEE
University of Queensland

In the lectures last week we studied the priority queue data structure. Recall:

A priority queue stores a set S of n integers and supports the following operations:

- `Insert(e)`: Adds a new integer to S.
- `Delete-min`: Removes the smallest integer in S, and returns it.

We saw that the binary heap provides an efficient implementation of a priority queue, supporting both of the operations above in $O(\log n)$ time and consuming $O(n)$ space (where n is the number of elements stored on the priority queue).

Recall also that we looked at two different ways of representing a binary heap:

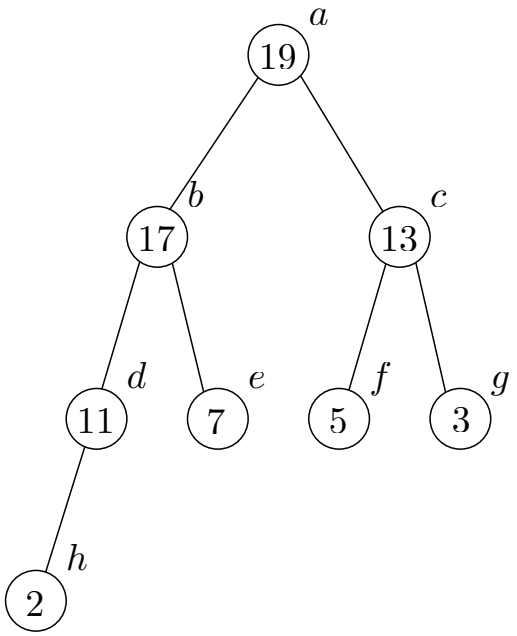
1. Using nodes linked together with pointers.
2. Using a dynamic array.

We saw that it was possible to construct a heap of n elements using only $O(n)$ time, which is an improvement upon the naive $O(n \log n)$.

Today we will look at some examples of running various operations on both of these representations of a binary heap to reinforce our understanding.

Heap Construction

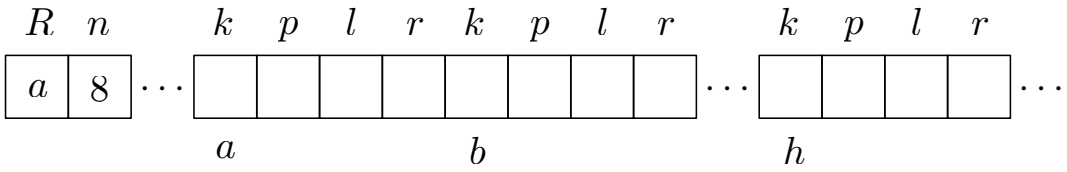
We saw how to construct a heap in $O(n)$ time with the dynamic array representation already, so let's look at the pointer-based version: Suppose we are tasked with building a heap containing $19, 17, 13, 11, 7, 5, 3, 2$. We will employ the same strategy, i.e. to build the following complete binary tree:



(where the letters denote the address of each node) and then "fix" it up so that it becomes a binary heap.

Heap Construction

To that end, we will take the following as our starting position:

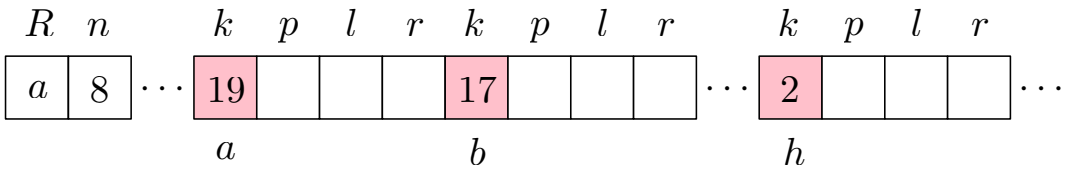


On the far left is our binary heap: we store a pointer to the root node and the number of nodes in our tree. We then contiguously allocate all of the nodes we need (creating essentially an array of nodes), where each node occupies 4 cells storing the key and pointers to the parent, left child and right child.

Heap Construction

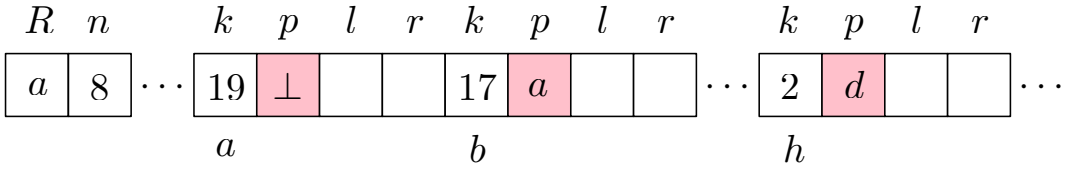
The reason why we allocate all nodes contiguously is so that we can fill out the fields using what we already know about the dynamic array representation.

The key will simply be sequentially allocated like so:



Heap Construction

The parent pointers can be computed using that the parent of the ith node will be at index $\lfloor \frac{i}{2} \rfloor$ in the array of nodes:



- The node at a has index 1, thus the parent is at index 0. This index is invalid, so we know that this node is the root.
- The node at b has index 2, thus the parent is at index 1, hence the value should be set to a.
- The node at h has index 8, thus the parent is at index 4, which corresponds to the address d.

Heap Construction

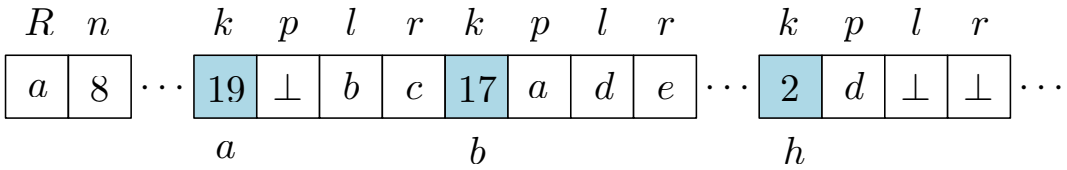
The left and right child pointers are set similarly using information encoded by the ordering of the array:

R	n		k	p	l	r	k	p	l	r		k	p	l	r	
a	8	$\cdots$	19	$\perp$	b	c	17	a	d	e	$\cdots$	2	d	$\perp$	$\perp$	$\cdots$
			a				b					h				

- The node at b has index 2, so its children are at $2 \times 2 = 4$ and $4 + 1 = 5$ respectively hence d and e.
- The node at h has index 8, which tells us that its children should be at 16 and 17. Because these are out of bounds, i.e. greater than n, we know that h is a leaf node.

Heap Construction

Now that we have the complete binary tree constructed:



If we singled out just the key of every node, we get an identical structure to when we construct the dynamic array representation, thus we can use the exact same algorithm to reshuffle the keys.

However even if the nodes are arbitrarily placed throughout memory, we can still traverse the tree in an appropriate order to turn our tree into a binary heap in $O(n)$ time.

Heap Construction

Recall that the `root-fix` function when called on a node r turns the subtree rooted at r into a binary heap under the premise that both the left and right subtrees of r are already binary heaps.

With our pointer-based representation, we will make use of the following recursive function:

Root Fix (Recursive)

At any node:

1. If the left child exists, make recursive call on it.
2. If the right child exists, make recursive call on it.
3. Call `root-fix` on self.

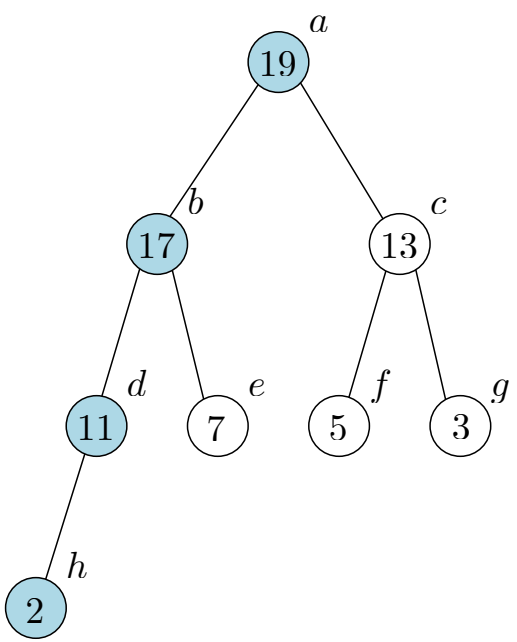
Heap Construction

It should be completely obvious why this recursive function traverses the tree in the right order: the two recursive calls ensure that the left and right subtrees are binary heaps, and only then do we call `root-fix`.

Let's see how this recursive algorithm runs on our example.

Heap Construction

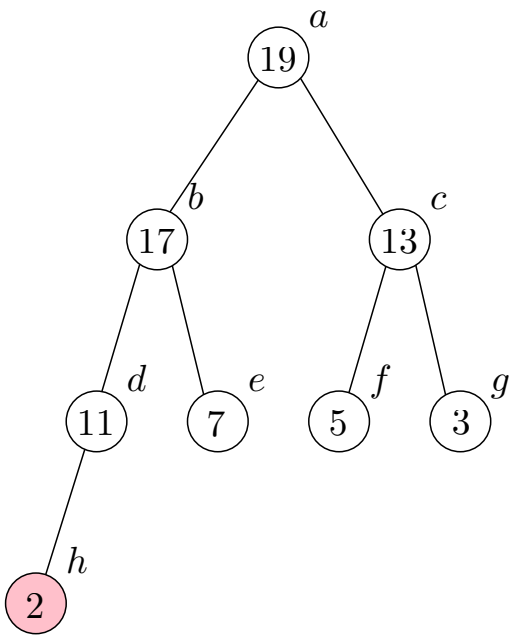
To turn the whole tree into a binary heap, we call `root-fix-recursive` on the root node:



Since it does have a left child, it will recurse into b, which will then subsequently recurse into d and then into h.

Heap Construction

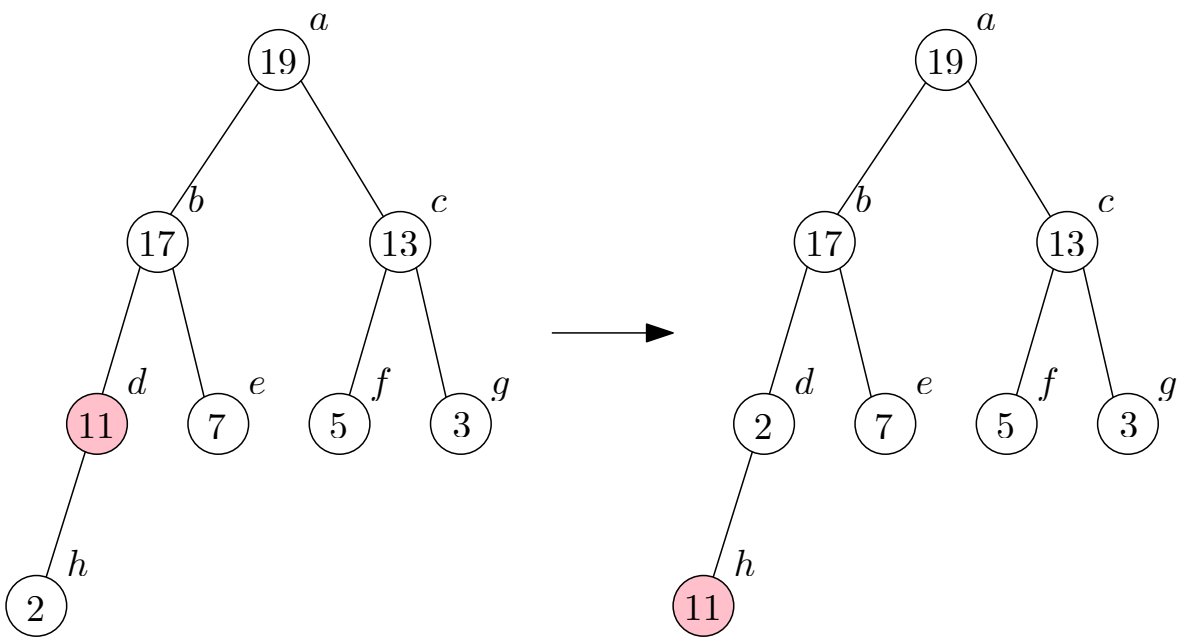
h has neither a left nor a right child, thus this is the first time `root-fix` gets called:



It does nothing because h is a leaf node, and so it is trivially already a binary heap.

Heap Construction

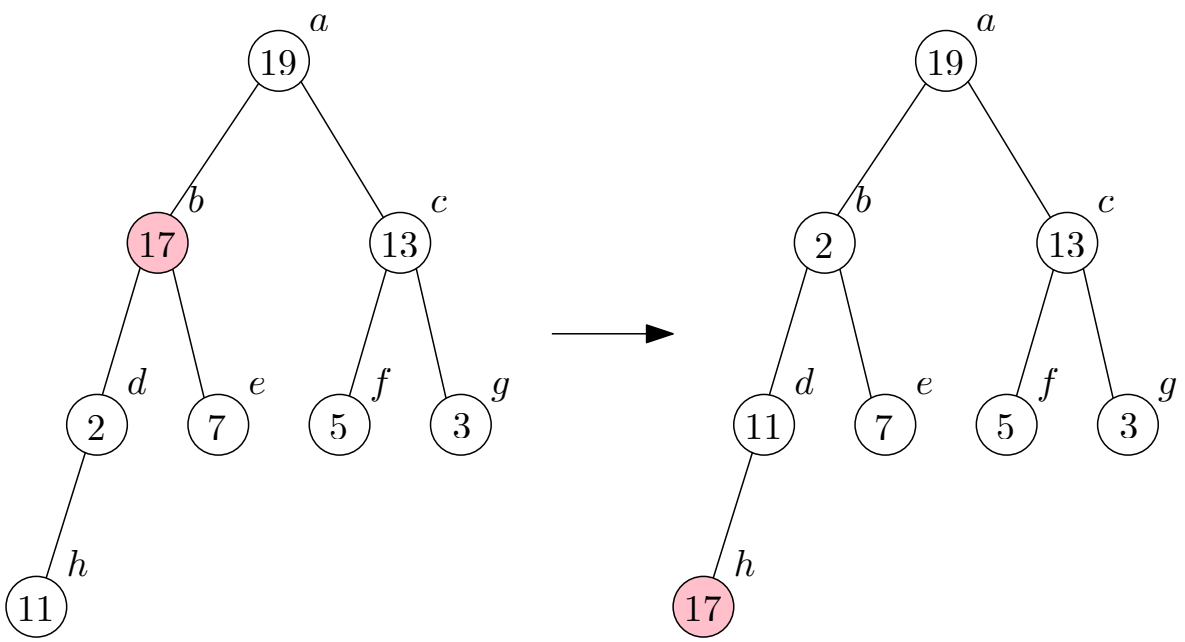
We then return to d, and upon finding that d does not have a right child, `root-fix` is called on d:



At d we find that a child, namely h, has a key that's smaller than us, so we make a swap (and this is the only swap we need to make).

Heap Construction

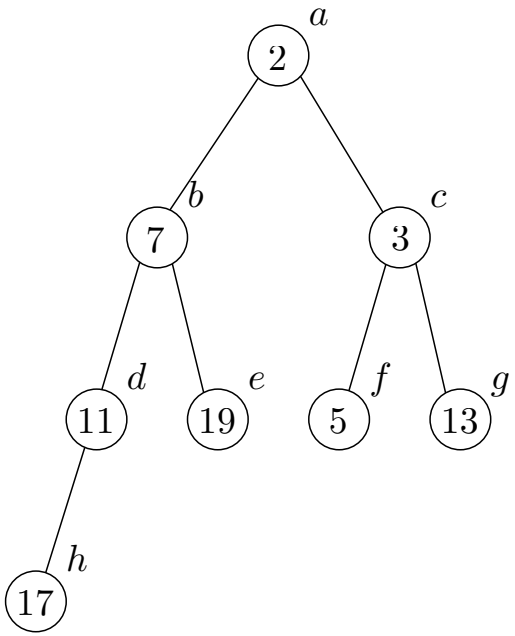
After returning to b and the recursive call on e does nothing, `root-fix` is called on b:



This time two swaps were required (between b and d, and between d and h).

Heap Construction

The process continues until eventually when we're done:



We have a binary heap!

Heap Construction

Remarks:

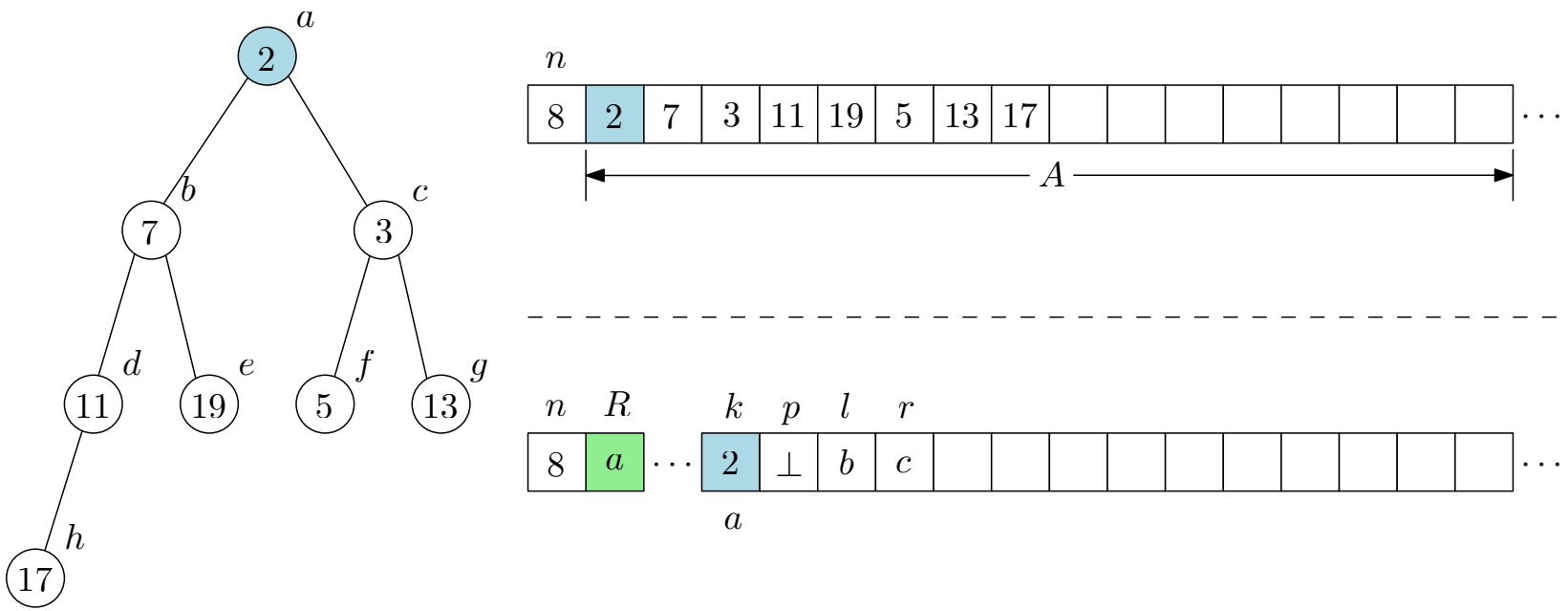
- The recursive function we gave is a post-order traversal on the tree, "post" because the node we are calling the function on is visited after its subtrees have been visited.
- The order in which `root-fix` is called on each node over the course of the recursive calls is:

  $$h, d, e, b, f, g, c, a$$

  Note that this is not the same ordering as the dynamic array representation (which is simply reverse alphabetical).
- Even if we do choose to put all of our initial nodes in one contiguous segment initially, once we start performing `delete-min` and `insert` operations they will no longer be in one block.

Delete-Min

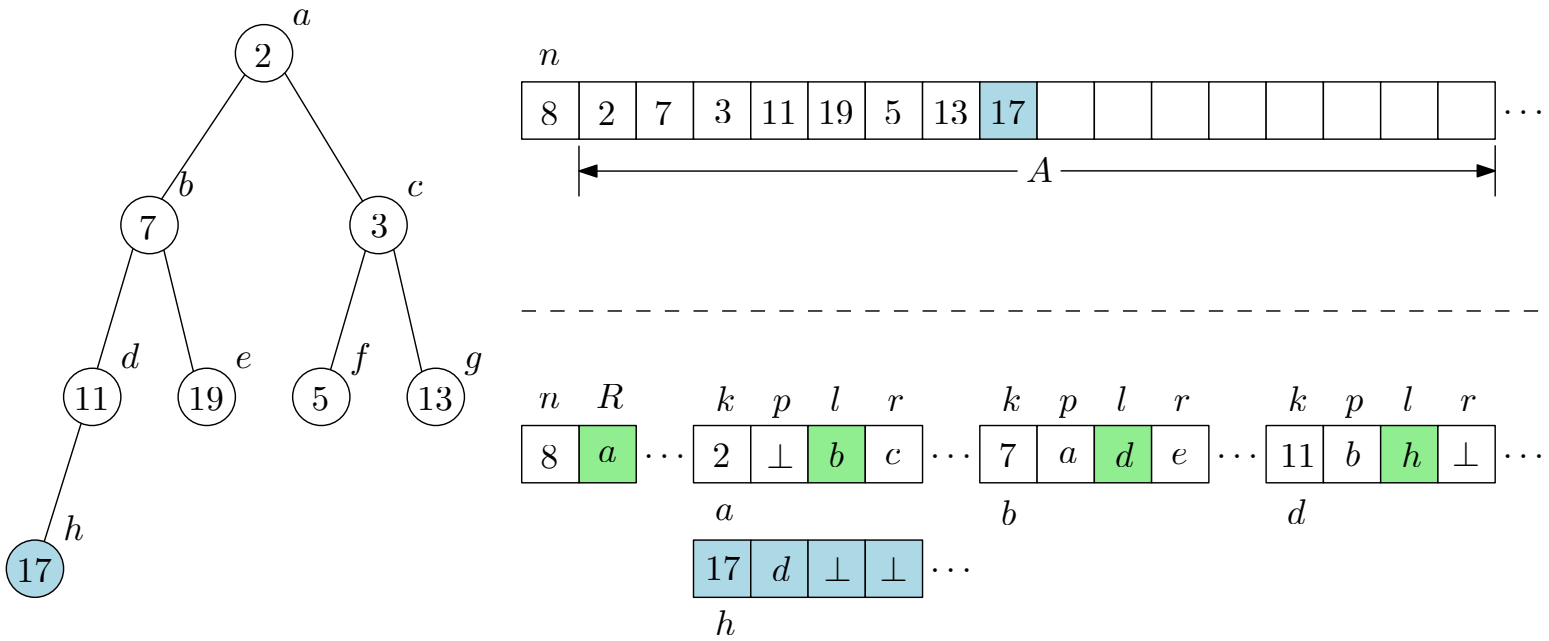
Now that we finally have a binary heap, let's see how `delete-min` runs. We start with the tree we had before. First we are interested in reporting the minimal key:



In the dynamic array representation, we simply index into the array for the first element. In the pointer-based representation, since we always keep a pointer to the root node we simply follow it to get the key.

Delete-Min

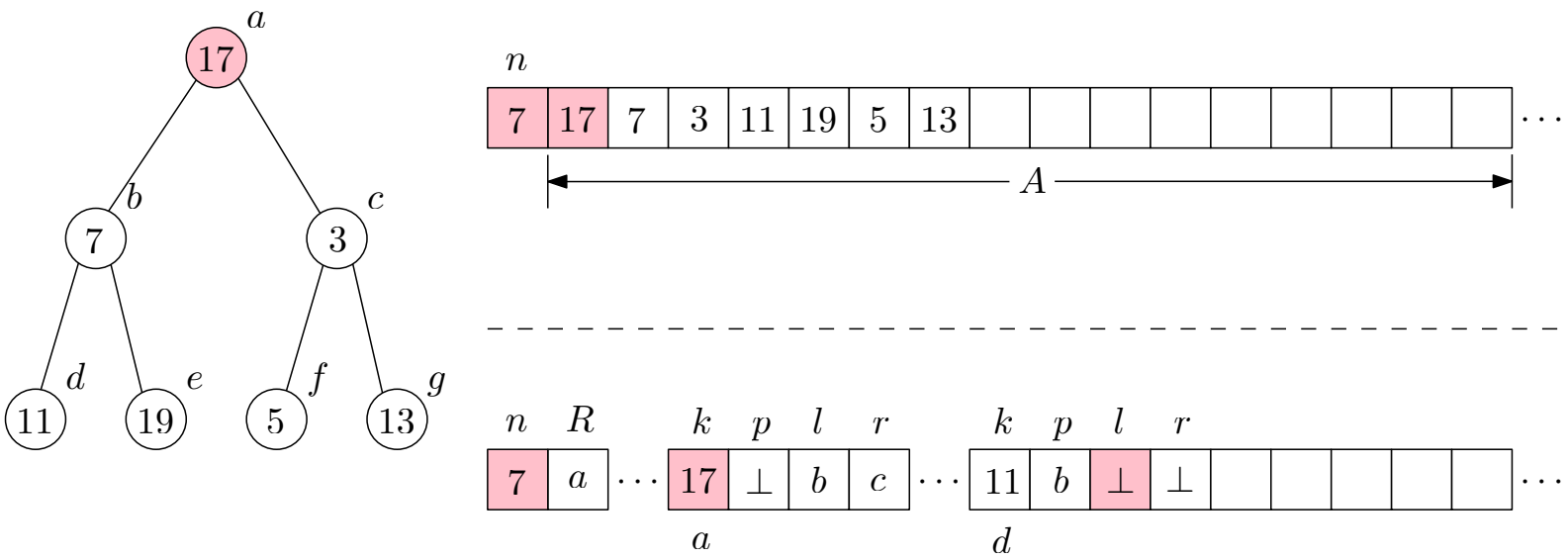
The next step is to locate the "last" node in the binary heap, i.e. the rightmost leaf node on the bottom level:



For the dynamic array, we simply index using the value of n; for pointer-based we need to write n in binary (in this case 0b1000), which tells us to go left three times from the root.

Delete-Min

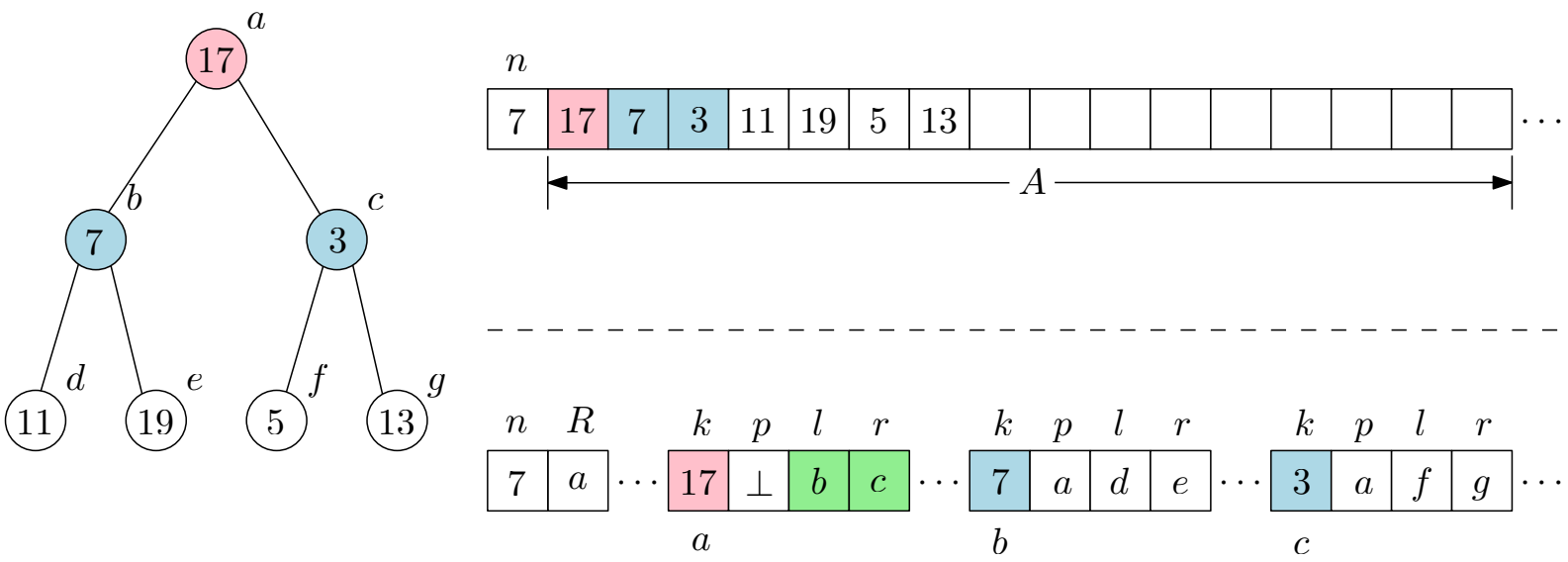
We then swap the key of the root node with the last node and delete the last node (updating the count of elements while we are at it):



For the dynamic array, we simply delete the last element; for pointer-based we free the node at h and update its parent d's child pointer.

Delete-Min

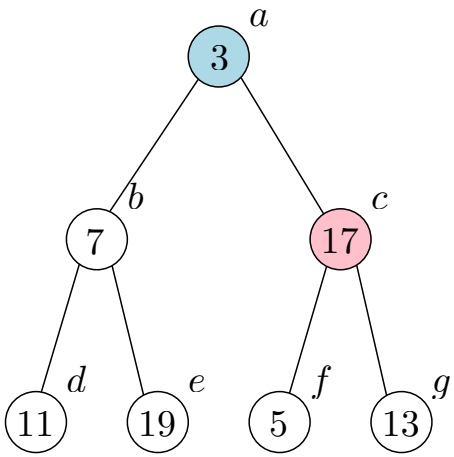
Now we need to re-establish the binary heap by swapping keys if necessary:

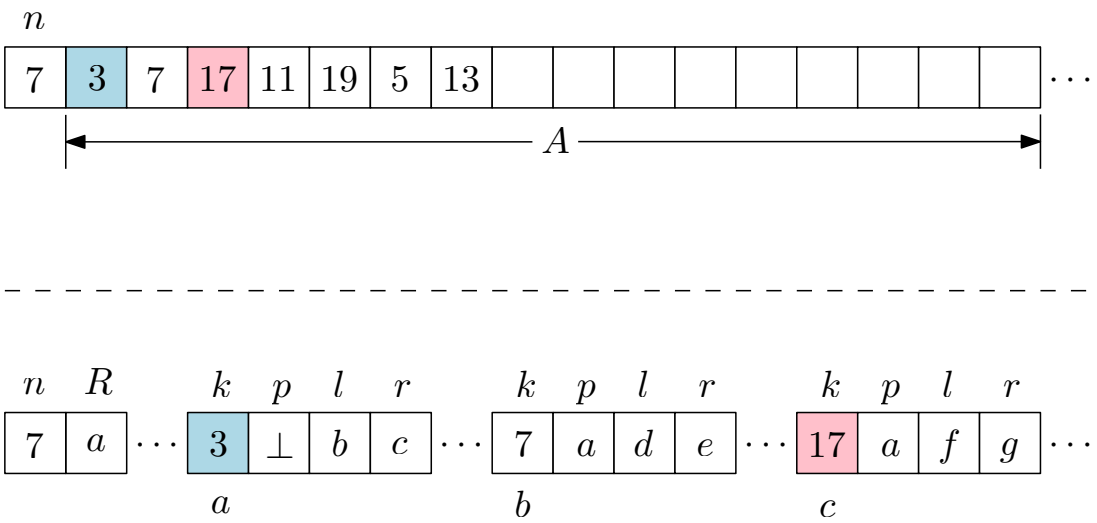


For the dynamic array, we can calculate the indices of the children by multiplying the current index by 2 for the left child (and adding 1 for the right child); for pointer-based we simply follow the pointers to find the left and right child nodes. We do need a swap in this case (with c since it has the smaller key of the two child nodes).

Delete-Min

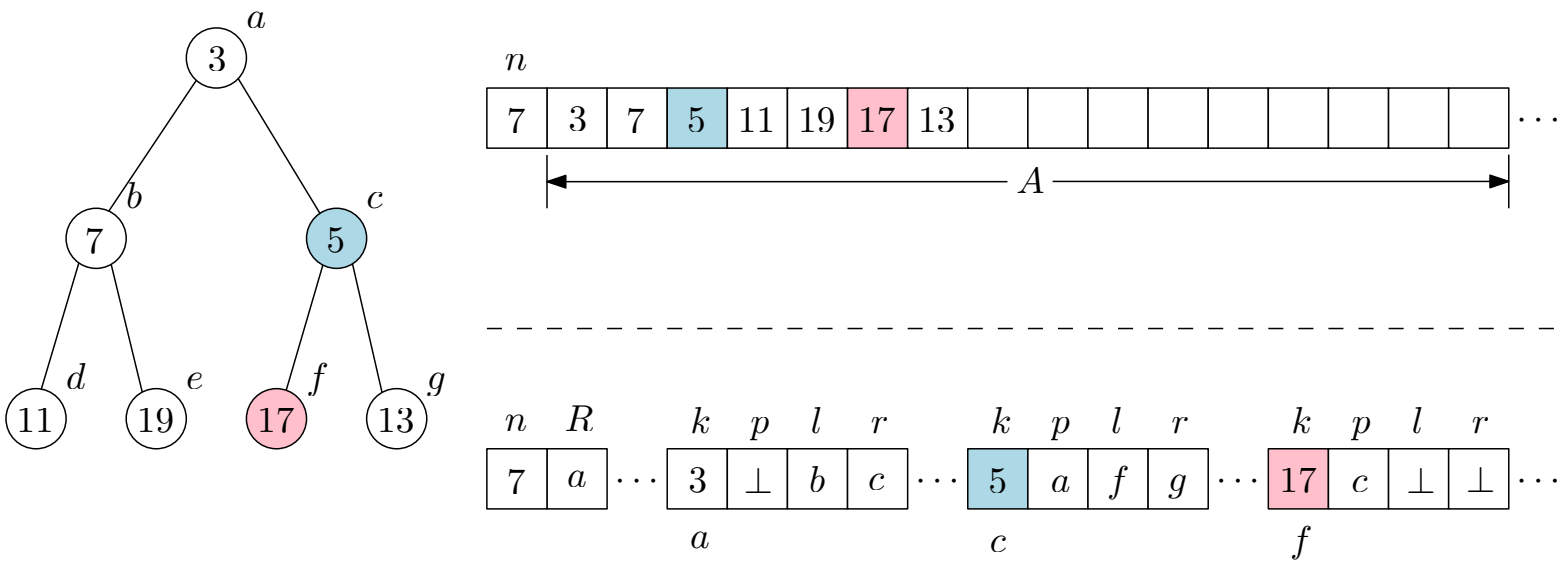
After swapping the keys of a and c:





Delete-Min

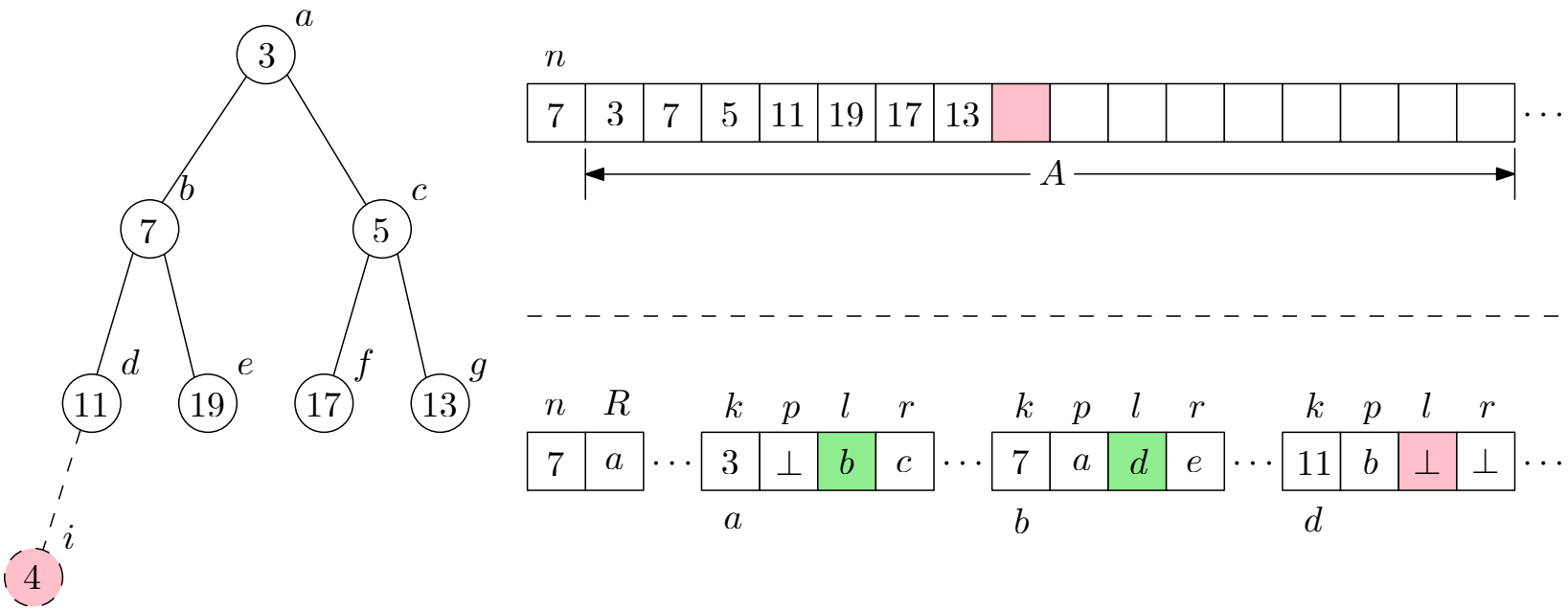
We then need another swap between c and f:



And then we're done!

Insertion

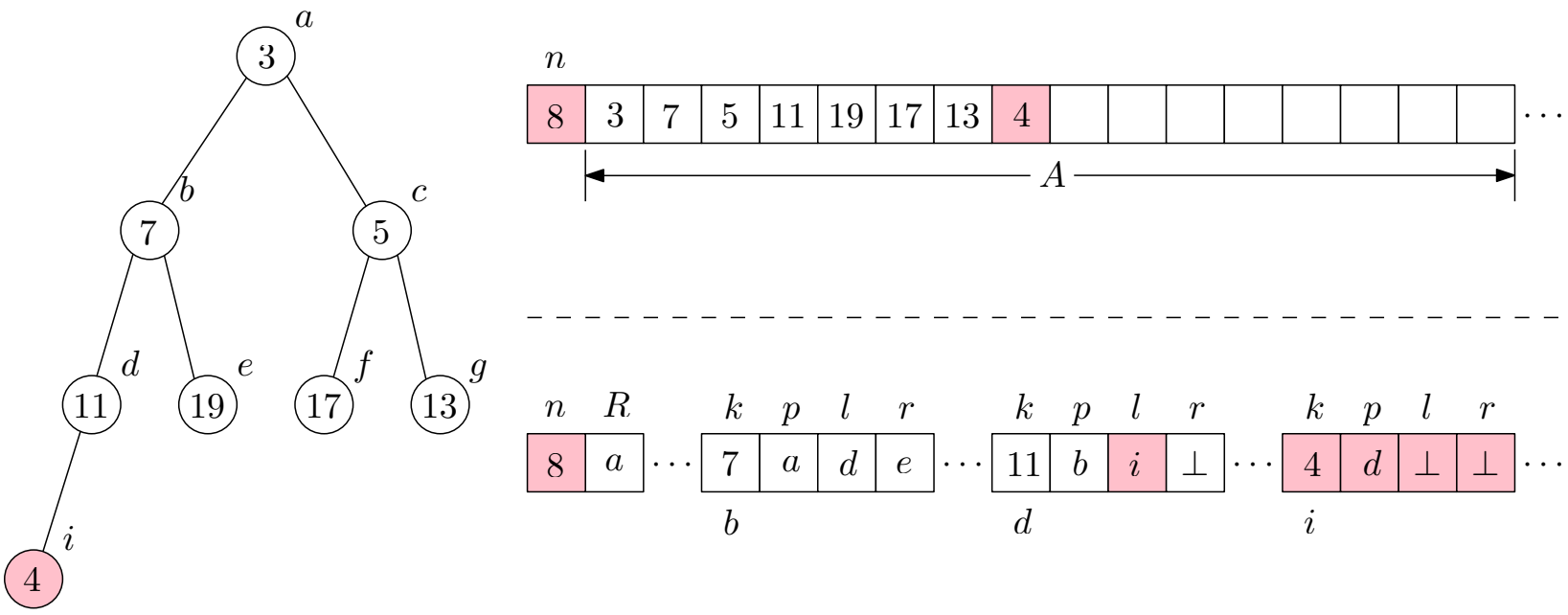
Let's look now at inserting say a key of 4:



To figure out where to add the node, for dynamic array simply append to the end of the array; while for pointer-based we need to again look at the binary representation, this time of $n + 1$. We will again get 3 zeroes, telling us to go left twice and then add a node as the left child.

Insertion

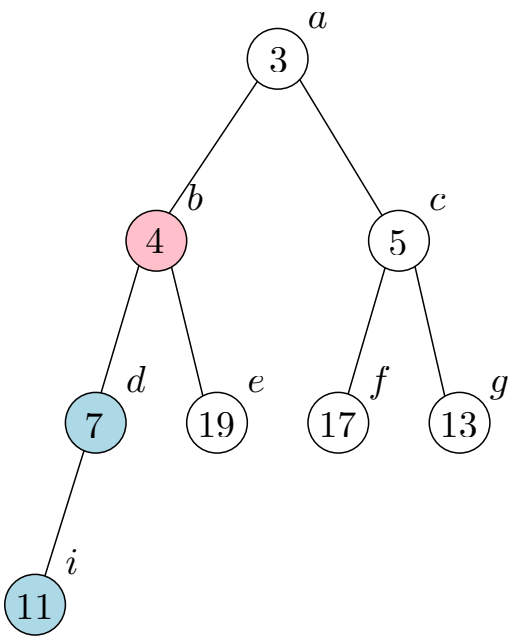
This is what it looks like after the node has been inserted:

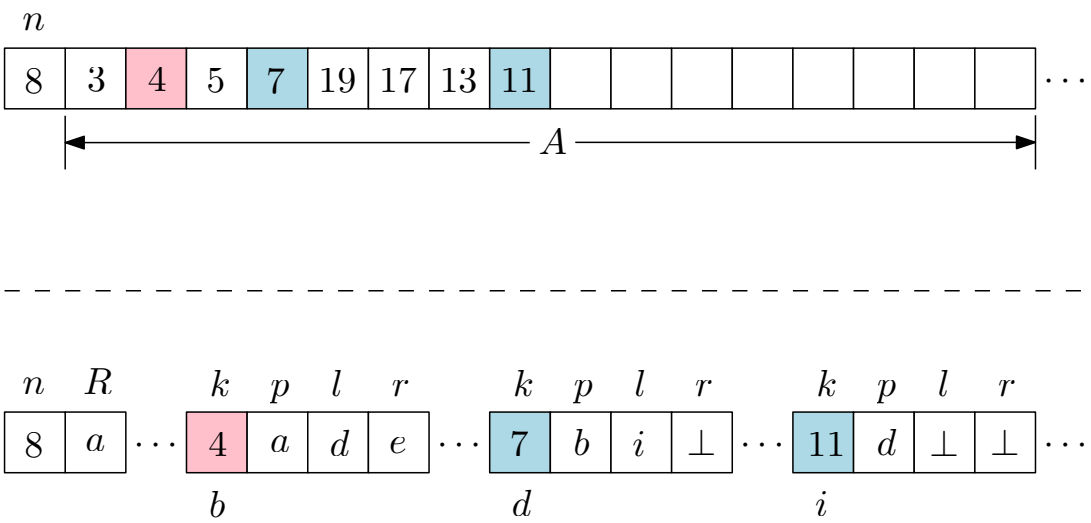


Notice we also adjusted the total count of elements. Again now the task is to re-establish the binary heap, and we do this by swapping upwards.

Insertion

After the two swaps:





And the insertion is complete!