Priority Queues and Heaps

- The ***priority queue*** is an ADT that associates a priority value with each element.
    - The element with the highest priority is the first one dequeued.
        - Typically, corresponds to the element with the lowest priority value.

- The priority queue ADT has an interface that looks like this:
    - `insert()` – insert an element with a specified priority value
    - `first()` – return the element with the lowest priority value (the "first" element in the priority queue)
    - `remove_first()` – remove (and return) the element with the lowest priority value

- The user's view of a priority queue is indeed something like a queue, ordered by priority value:

- A priority queue is typically implemented using a data structure called a ***heap***.

Heaps

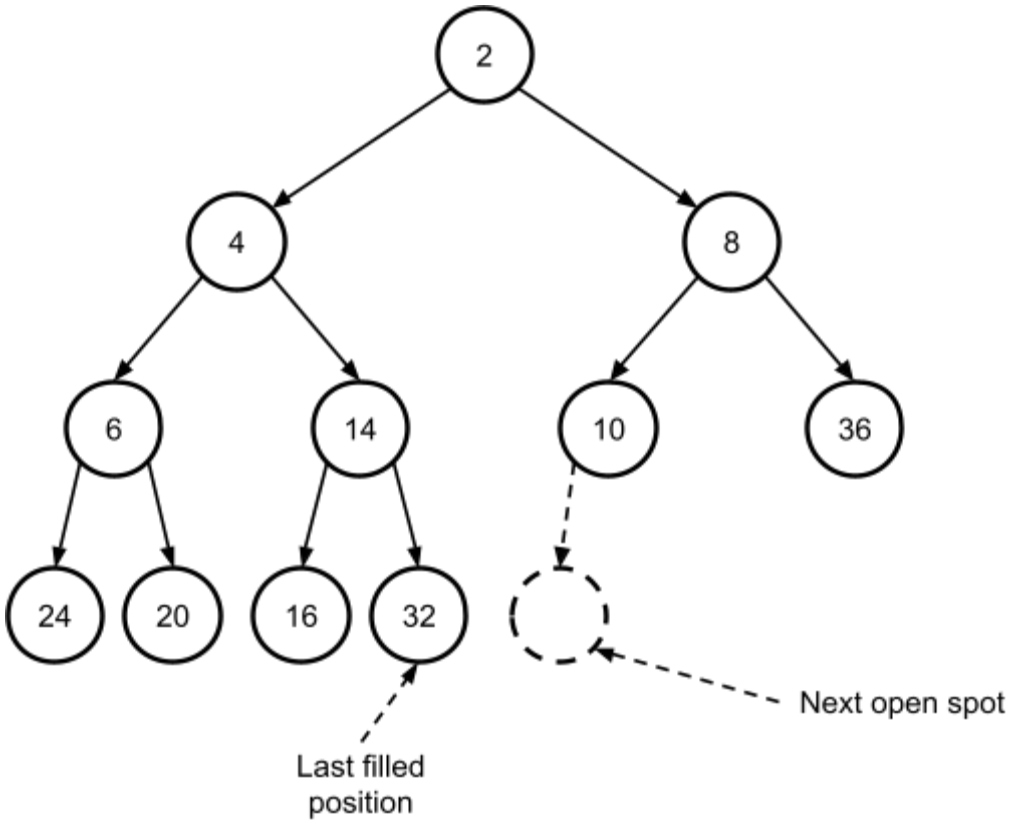
- We've already seen the term "heap" used to refer to the memory space used for dynamic memory allocation (i.e. `malloc()`).
    - The heap data structure is unrelated to this other kind of heap.

- A heap data structure is typically a *complete* binary tree in which every node's value is less than or equal to the values of its children.
    - This version of the heap is specifically called a ***minimizing binary heap***, or just "min heap".
    - We can also have a "max heap", where each node's value is greater than or equal to the values of its children.

- Remember, a complete binary tree is one that is filled, except for the bottom level, which is filled from left to right.

    - The longest path from root to leaf in such a tree is *O(log n)*.

- Here's an example of a min heap, with only priority values displayed:

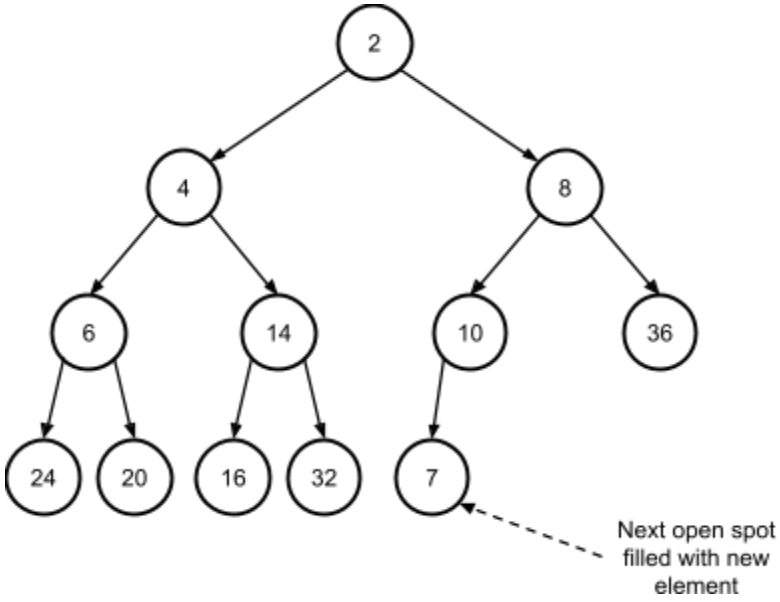


- Note that we keep track of the last filled position and the next open spot in the complete binary tree.

Adding a node to a heap

- A min (or max) heap is maintained through the addition and removal of nodes via ***percolations***, which move nodes up and down the tree according to their priority values.

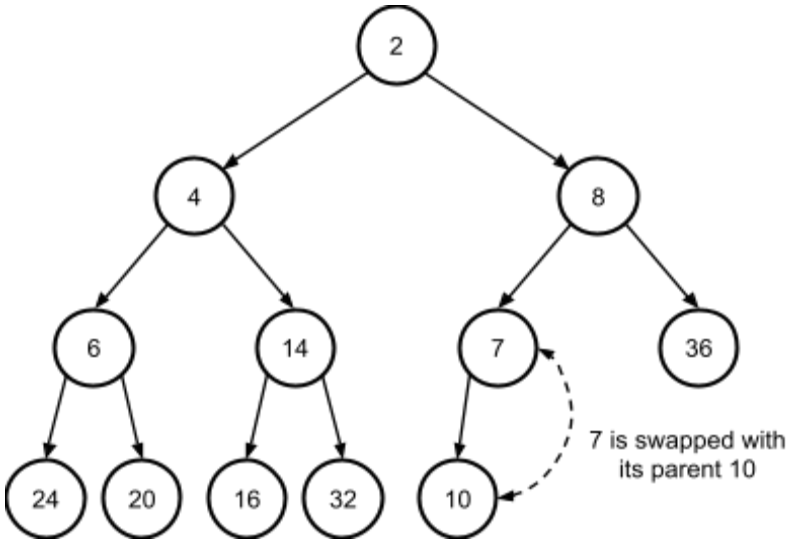
- When we add a value to a heap, we place it into the next open spot and then percolate it up the heap until its priority value is less than (or greater than, for a max heap) both of its children.
- As an example, consider adding the value 7 to the min heap above. We'd first place it in the next open spot:



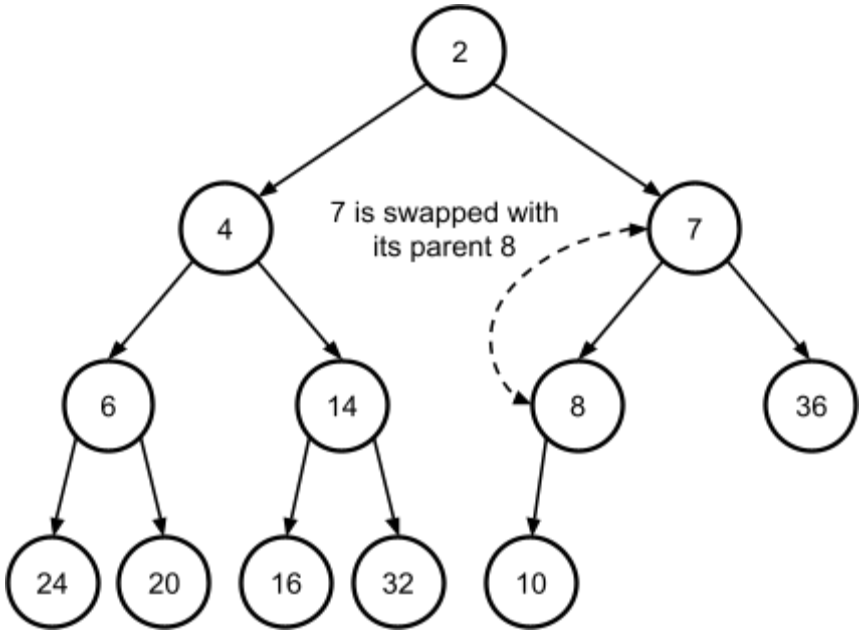
- Then, we'd percolate the new element up the tree, following this method:

```
while new priority value < parent's priority value:
     swap new node with parent
```

- This, we'd first compare the new node (7) with its parent (10) and see that they needed to be swapped to maintain the min heap property:



- We'd then compare the new node (7) with its new parent (8) and see that they too needed to be swapped:



- After this swap, the min heap property is restored, and the percolation can end.

- What is the complexity of this percolation operation?
    - *O(log n)*

Removing a node from a heap

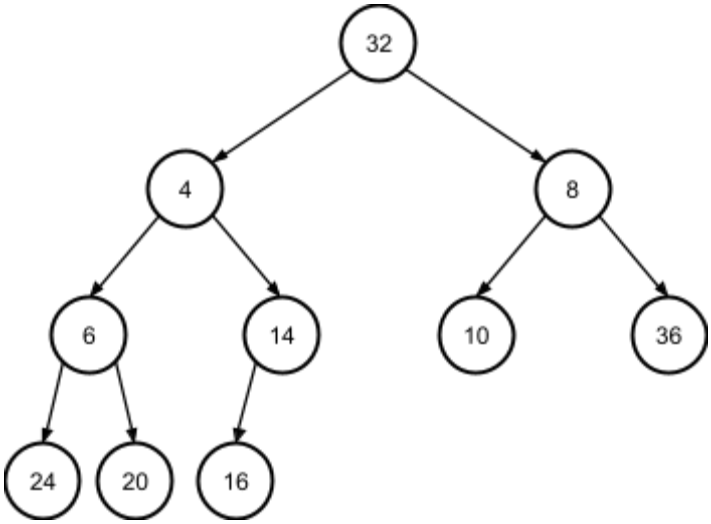
- In a min heap, the root node's priority value is always the lowest.
    - Similar for a max heap.

- That means the `first()` and `remove_first()` operations always respectively access and remove the root node.

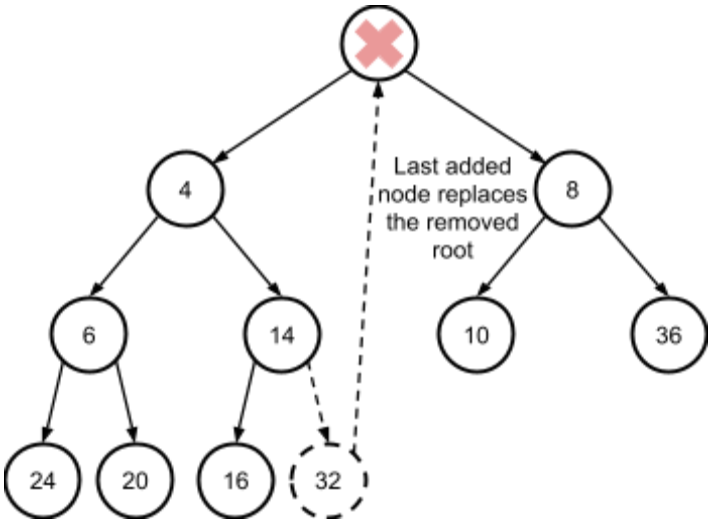
- If we always remove the root node, how do we replace it?
    - Remember, we need to maintain the completeness of the binary tree.

- To replace the root node after it is removed, we replace it with the element last added to the heap and then fix the heap by percolating that node down.

- Let's look at our original min heap above as an example. When we remove the root node from that heap, we initially replace it with the last added node (32):



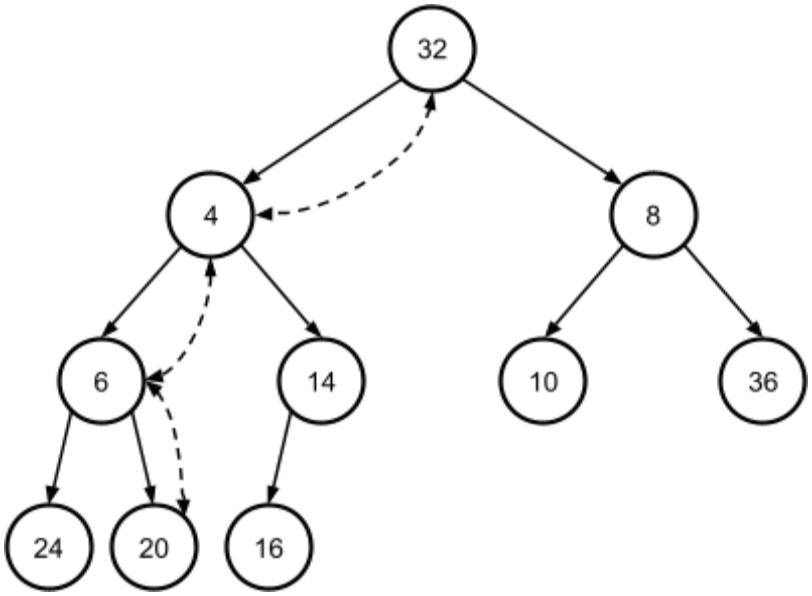
- The last added node is removed after its value replaces the root:



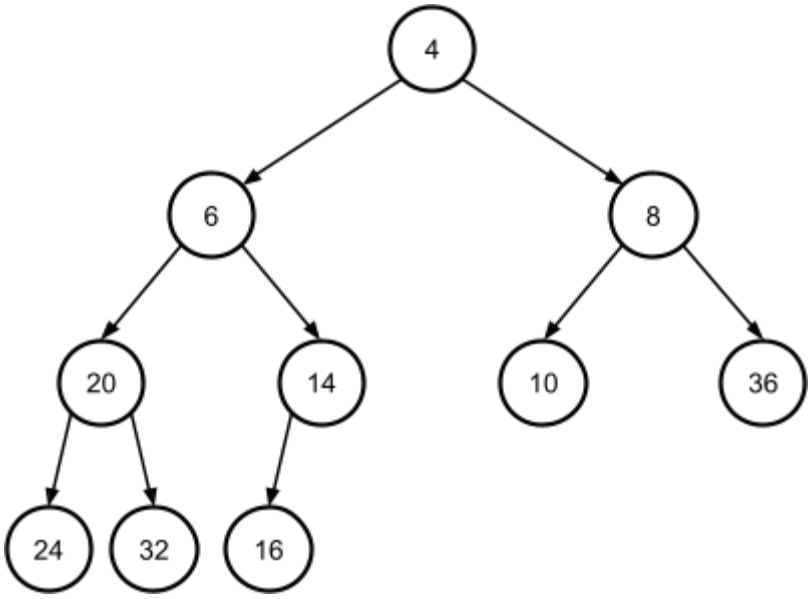
- Then, we percolate the replacement node down the tree, following this method:

```
while priority > smallest child priority:
     swap with smallest child
```

- Thus, 32 would follow this path down the heap:



- And we'd end up with a heap that looked like this:



Implementing a heap

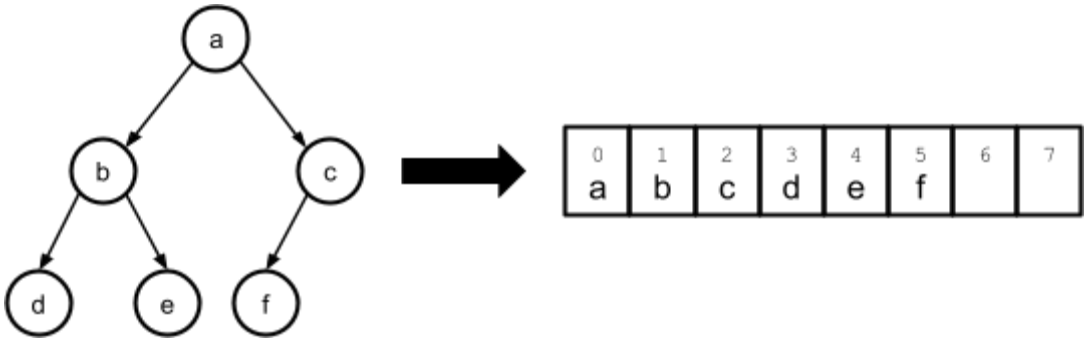
- We can efficiently implement the complete binary tree representation of a heap using an array (or a dynamic array).

- In the array representation, the root node of the heap is stored at index 0.

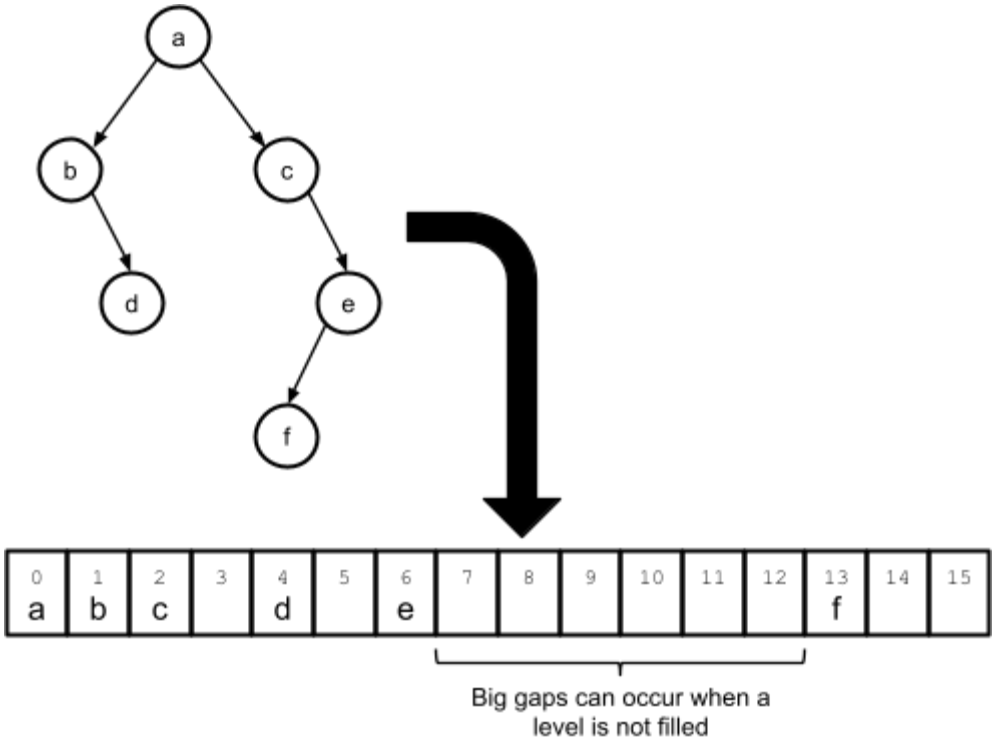
- The left and right children of a node at index `i` are stored respectively at indices `2 * i + 1` and `2 * i + 2`.

- The parent of a node at index `i` is at `(i - 1) / 2` (using the floor that results from integer division).

- Here's an example of this array representation:

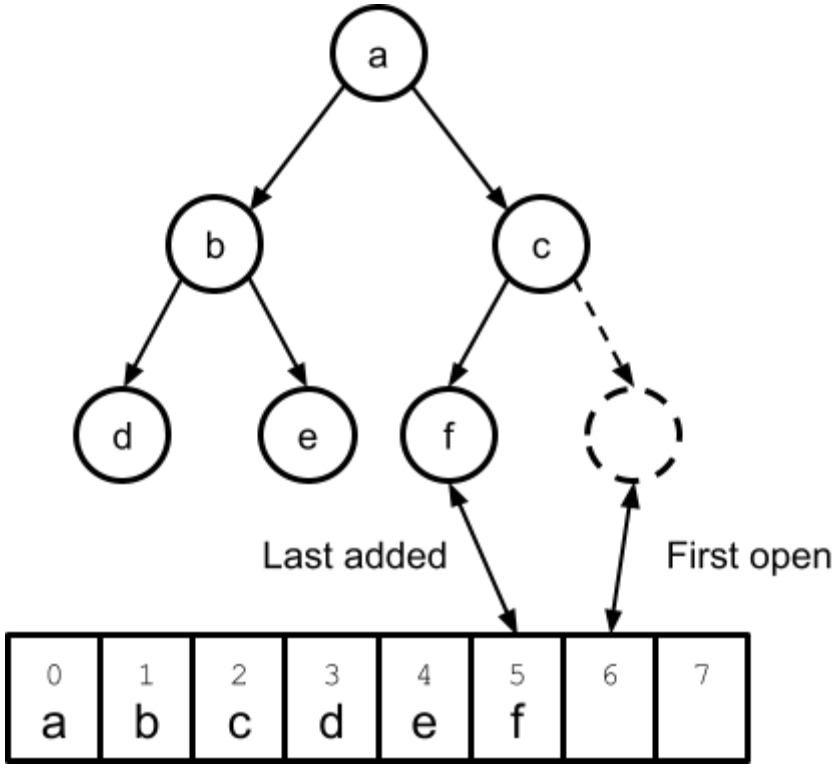


- This array implementation works well because the tree is complete. If the binary tree was not complete, it would be difficult to implement with an array:



- Keeping track of the last added element and the first open spot in the array representation of the heap is simple. These are simply the last element in the

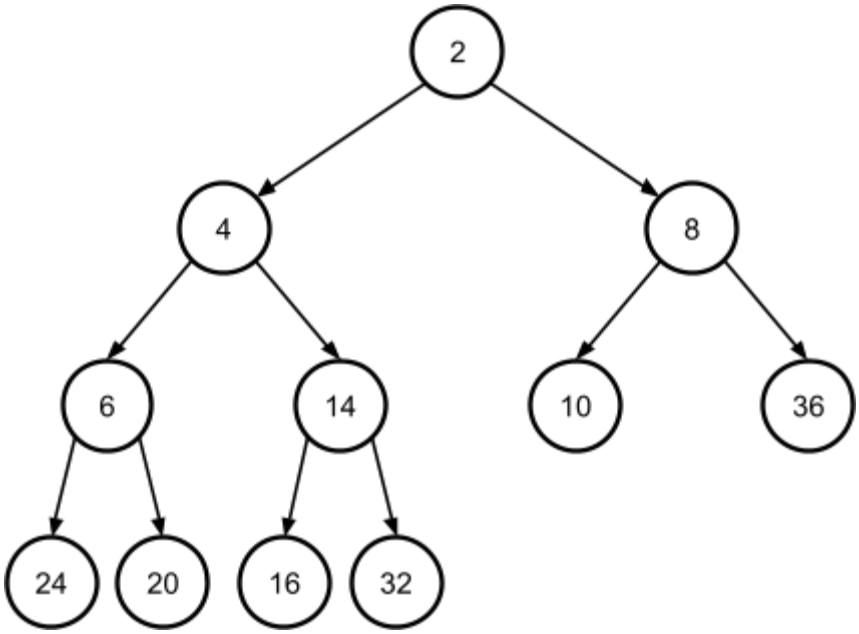
array and the following empty spot, respectively:



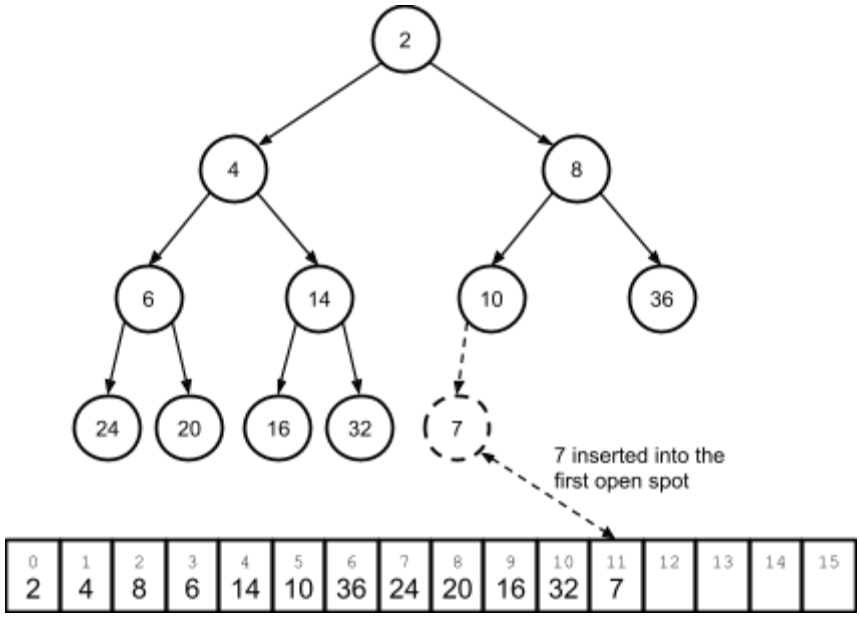
Inserting into an array-based heap

- Inserting an element into the array representation of the heap follows this procedure:
    1. Put new element at the end of the array.
    2. Compute the inserted element's parent index (`(i - 1) / 2`).
    3. Compare the value of the inserted element with the value of its parent.
    4. If the value of the parent is greater than the value of the inserted element, swap the elements in the array and repeat from step 2.
        - Do not repeat if the element has reached the beginning of the array.

- Let's see what this looks like for the insertion example we looked at above, where we added 7 to this heap:

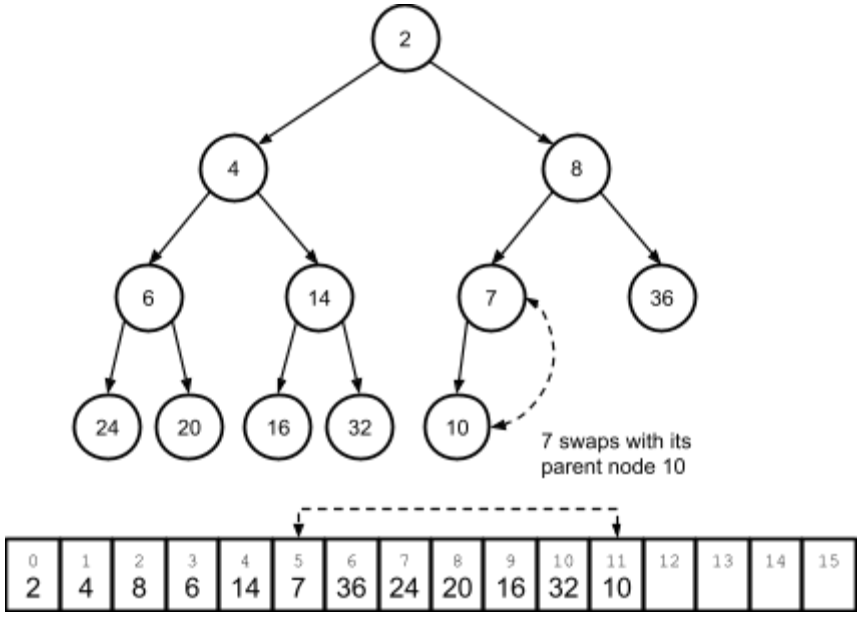


- We'd first insert the new element into the end of the array:

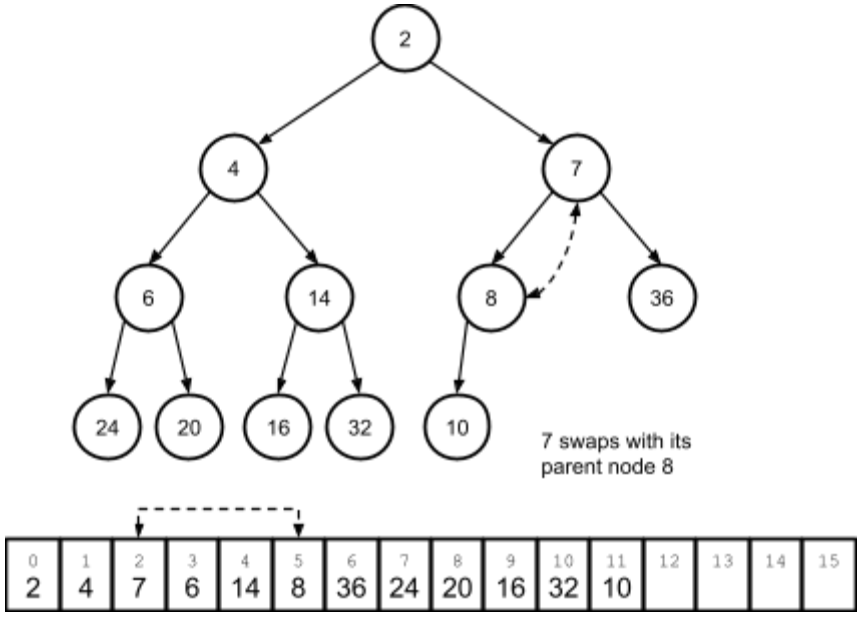


- We'd compute the index of 7's parent node (`(11 - 1) / 2` → `5`), and we'd compare 7 with the value we found there.

- Since 7 is less than 10, we'd swap them:



- We'd repeat, comparing 7 to its new parent 8 at index `(5 - 1) / 2` → `2`, and we'd have to swap again:



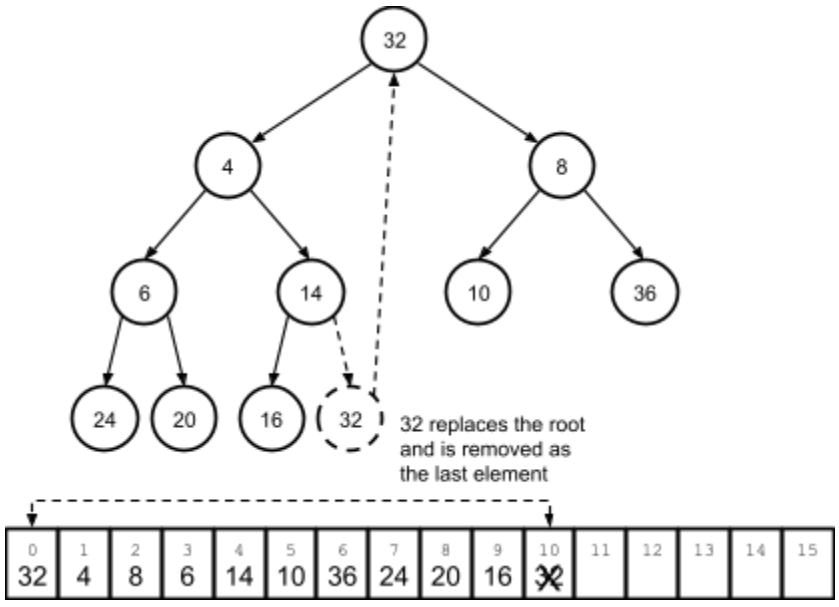
- Finally, we'd compare to 7's new parent node 2 at index `(2 - 1) / 2` → `0`, and we'd stop, since 2 is less than 7.

Removing from an array-based heap

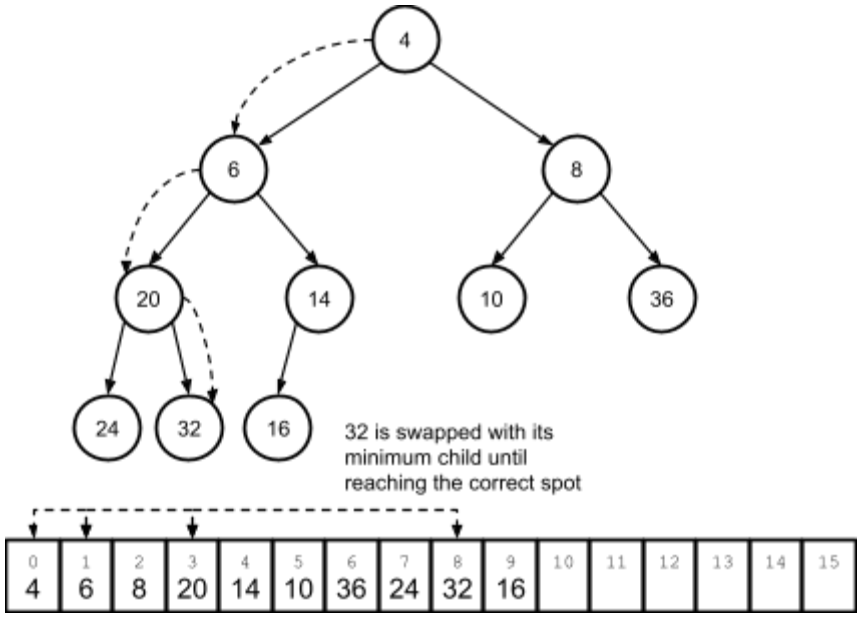
- Accessing the minimum element in an array-based heap representation is easy: just return the value of the first element in the array.

- Removing the minimum element is slightly more involved. It follows this procedure:
    1. Remember the value of the first element in the array (to be returned later).
    2. Replace the value of the first element in the array with the value of the last element and remove the last element.
    3. If the array is not empty (i.e. it started with more than one element), compute the indices of the children of the replacement element (`2 * i + 1` and `2 * i + 2`).
        - If both of these elements fall beyond the bounds of the array, we can stop here.
    4. Compare the value of the replacement element with the minimum value of its two children (or possibly one child).
    5. If the replacement element's value is less than its minimum child's value, swap those two elements in the array and repeat from step 3.

- Looking at our example from before, removing the root in this heap would first involve replacing the root (the first element in the array) with the last element and then removing the last element:



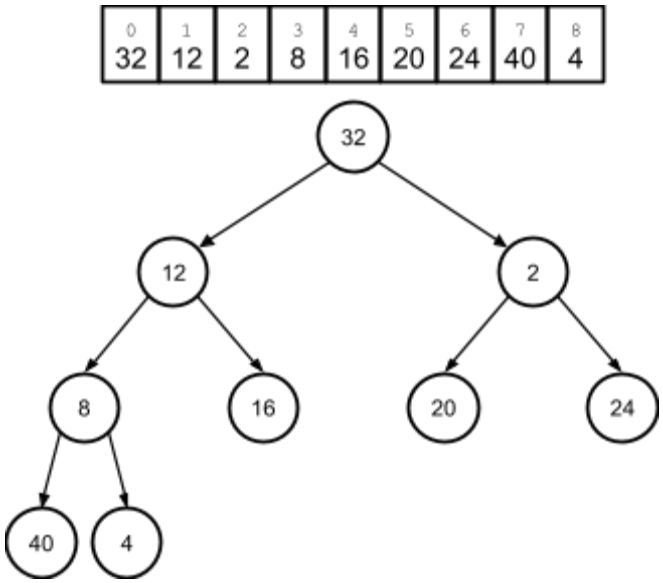
- Then, we would percolate 32 down the array, comparing it to its minimum-value child and swapping values in the array until 32 reached its correct place:



Building a heap from an arbitrary array

- We can use a procedure similar to the downward percolation portion of our removal method to build a heap from an arbitrary array of values.

- Before we see how to do this, let's look at an arbitrary array as a heap:



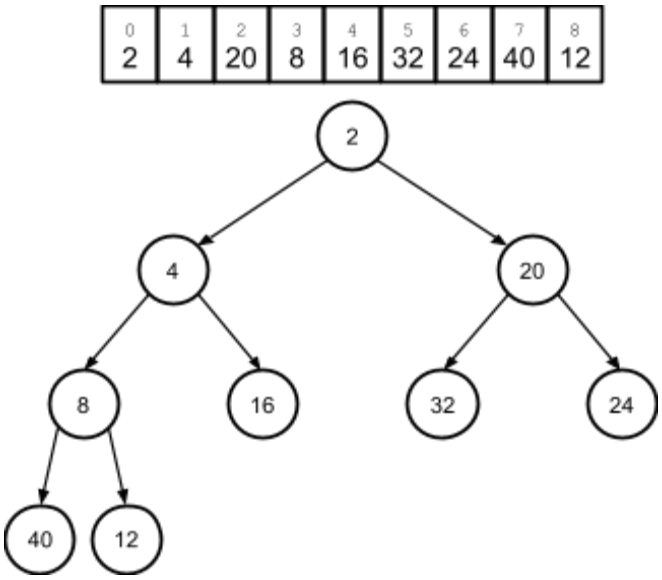
- As we can see, this array does not correspond to a proper heap.

- But, we can break the problem down: each individual leaf subtree is a proper heap (of one element).

- Thus, if we percolate down the first non-leaf element, then the subtree rooted at that element's original position will be a proper heap.
    - The first non-leaf element (from the back of the array) is at `n / 2 - 1` (using the floor from integer division).

- We can repeat this, moving backwards one element at a time from the first non-leaf element, and each time we percolate an element down, the subtree rooted at that element's original position will be a proper heap.

- Once we percolate down the root element, the entire array will represent a proper heap:



- What is the time complexity of this operation?
    - We perform *n / 2* downward percolation operations.
    - Each of these operations is *O(log n)*.
    - This means the total complexity is *O(n log n)*.

- What is the space complexity?
    - No additional space needed and no recursive calls: *O(1)*.

Heapsort

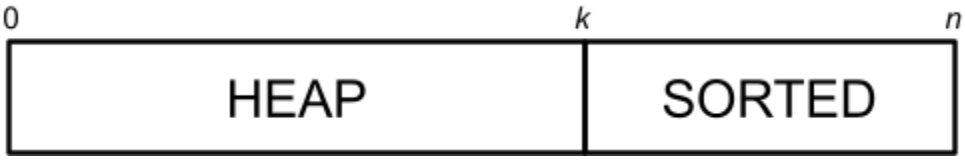
- Given the heap and its operations described above, we can implement an efficient (*O(n log n)*), in-place sorting algorithm called ***heapsort***.

- The first thing heapsort does is builds a heap out of the array using the procedure described above.

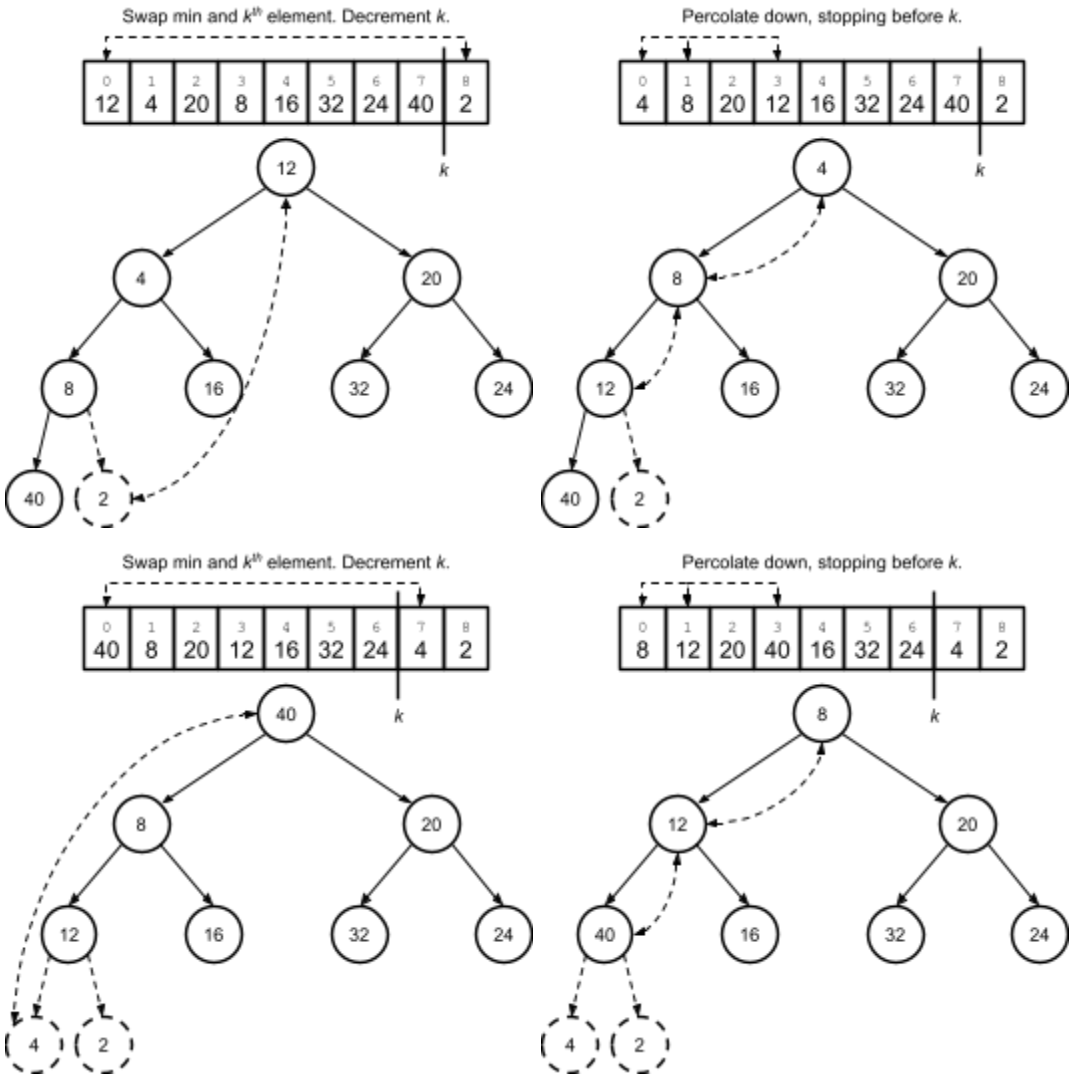
- Then, to complete the sort, we use a procedure like our heap removal operation above, with a few small tweaks:
    - Keep a running counter *k* that is initialized to one less than the size of the array (i.e. the last element).
    - Instead of *replacing* the first element in the array (the min) with the last element (the k^{th} element), we *swap* those two elements in the array.
        - The array itself remains the same size, and we decrement *k*.
    - When percolating the replacement value down to its correct place in the array, we stop at the k^{th} element.
        - Thus, the heap is effectively shrinking by 1 at each iteration.

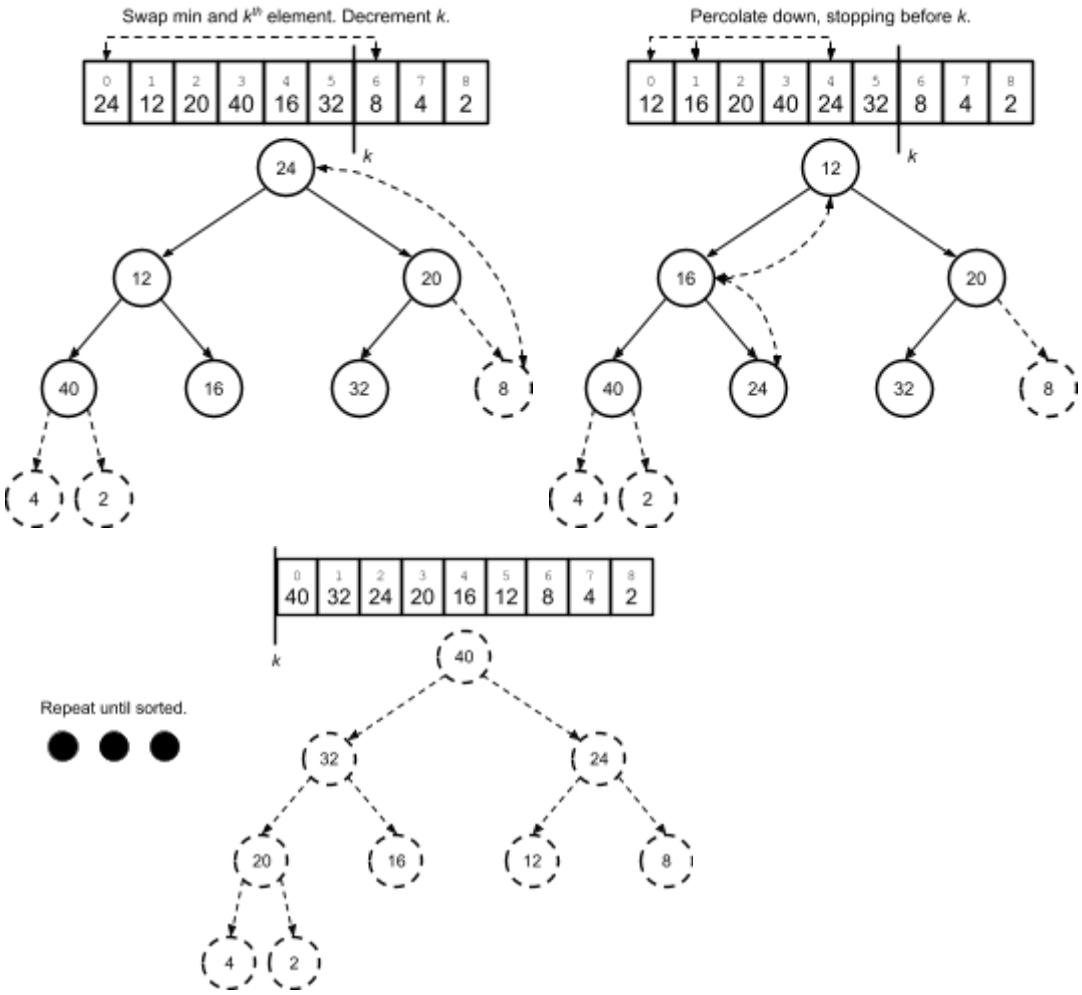
- We repeat this procedure until *k* reaches the beginning of the array.

- As this sorting procedure runs, it maintains two properties:
    - The elements of the array beyond *k* are sorted, with the minimum element at the end of the array.
    - The array through element *k* always forms a heap, with the minimum remaining value at the beginning of the array.



- Taking the heap array we built above, heapsort will run as follows, with *k* initialized to the last element in the array:




Swap min and k^{th} element. Decrement k.
Percolate down, stopping before k.
Repeat until sorted.
k