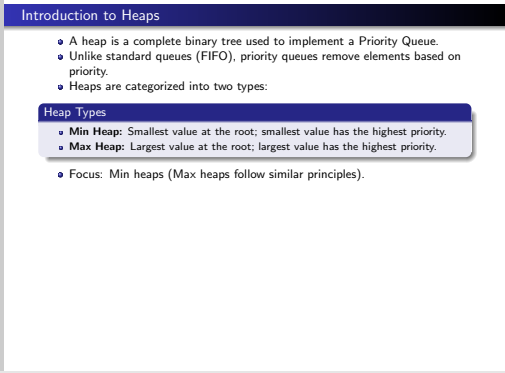
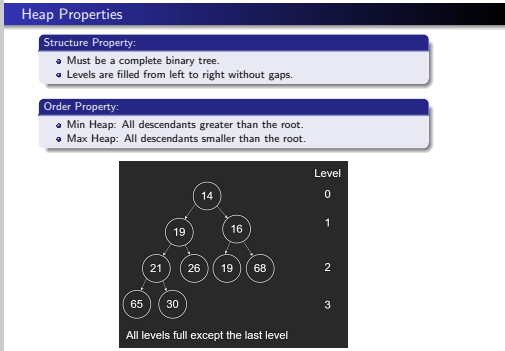
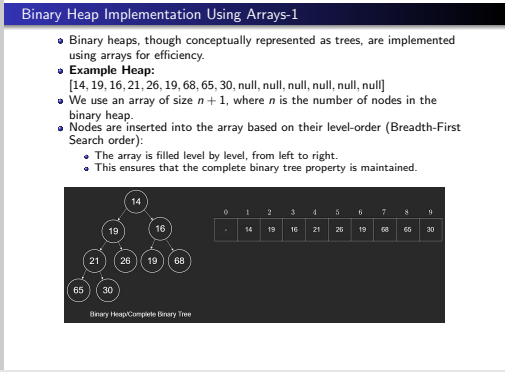


A heap is a specialized, tree-based data structure, which is a complete binary tree. It implements an abstract data type called the Priority Queue, but sometimes 'Heap' and 'Priority Queue' are used interchangeably. We already learned that queues operate with a first-in-first-out basis but with a priority queue, the values are removed based on a given priority. The element with the highest priority is removed first.

Min heaps have the smallest value at the root node and when deleting, the smallest value has the highest priority. Max heaps have the largest value at the root node and when deleting, the largest value has the highest priority. In this chapter, we will be focusing on min heaps, but the implementation is exactly the same for max heap, except you would prioritize the maximum value instead of the minimum.

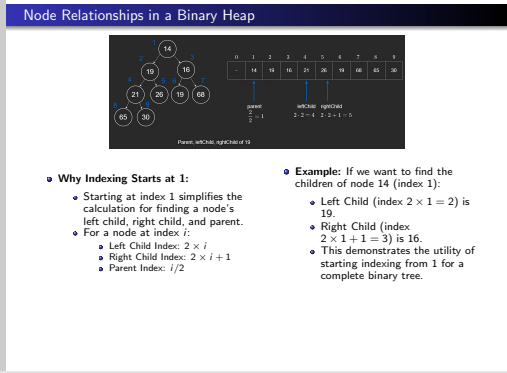


For a binary tree to qualify as a heap, it must satisfy the following properties:
1. Structure Property A binary heap is a binary tree that is a complete binary tree, where every single level of the tree is filled completely, except the lowest level nodes, which are filled contiguously from left to right.
2. Order Property
The order property for a min-heap is that all of the descendents should be greater than their ancestor. In other words, if we have a tree rooted at y, every node in the right and the left sub-tree should be greater than or equal to y. This is a recursive property, similar to binary search trees.
In a max-heap, every node in the right and the left sub-tree is smaller than or equal to y.



Binary heaps are drawn using a tree data structure but under the hood, they are implemented using arrays. Let's show how we can do this by using the given binary heap: [14,19,16,21,26,19,68,65,30,null,null,null,null,null,null]

We will take an array of size n+1 where n is the number of nodes in our binary heap. This will make sense soon. We will visit our nodes in the same order as we visit nodes in breadth-first search - level by level, from left to right. We will insert these into our array in a contiguous fashion. However, we will start filling them from index 1 instead of 0, for reasons we will discuss soon.



The reason why we start filling up our array from index 1 is because it helps us figure out the index at which a node's left child, right child, or the parent resides. Because binary heaps are complete binary trees, no space is required for pointers. Instead, a node's left child, right child and parent can be calculated using the following formulas, where i is the index of a given node; leftChild = 2i, rightChild = 2i+1, parent = i/2. Now, suppose we wanted to find the above properties of node 19. The following visual demonstrates how using the formulas helps us figure them out. The number within the circle at each node in the tree is the value stored at that node. The number above a node (in blue) is the corresponding index in the array. It is important to note that these formulas only work when the tree is a complete binary tree. We can also now appreciate why we start at index 1. Suppose we wanted to find 14's left and right child and 14 was at 0. Well, any number multiplied by a 0 is 0, and would tell us that the left child resides at the 0th index, which is of course not the case.

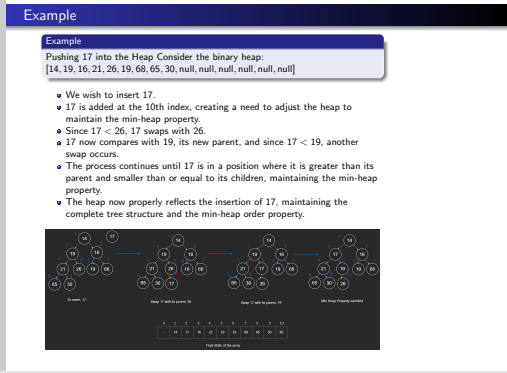
Maintaining Heap Properties

- **Maintaining Min-Heap Properties:** Whenever operations such as add or remove are performed on the heap, it's crucial to ensure that the min-heap properties are preserved.
- This involves adjusting the heap to maintain the structural and order properties, ensuring the smallest value remains at the root.
- The integrity of the parent-child relationship, defined by the index formulas, must also be preserved.

```
class Heap:
    heap = a list of integers, objects etc.
    constructor():
        heap = intialize the list
        heap.add(0)
```

Concept
Push Operation in Heaps
- To insert a new element into a heap (push operation), we must ensure that the heap's structural and order properties are maintained.
- The element is initially added at the next available position to keep the tree complete.
- We then adjust the heap to restore the heap property (percolate up).

- To insert a new element into a heap (push operation), we must ensure that the heap's structural and order properties are maintained.
- The element is initially added at the next available position to keep the tree complete.
- We then adjust the heap to restore the heap property (percolate up).



Taking the same binary heap from before: [14,19,16,21,26,19,68,65,30,null,null,null,null,null,null] let's say we wish to push 17. We need to make sure we push 17 such that we maintain our structure and order property.

Since a binary heap is a complete binary tree, and we are required to fill nodes in a contigious fashion, pushing 17 should happen at the 10th index. However, this might violate the min heap property, which means we will have to percolate 17 up the tree until we find its correct position.

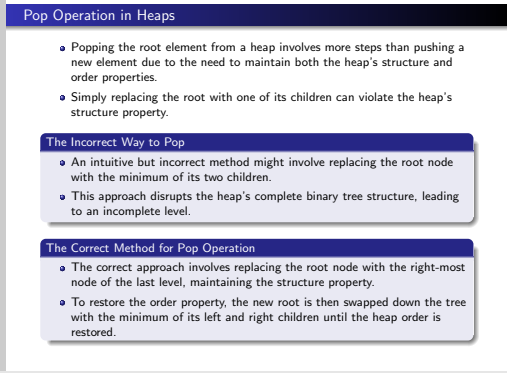
In this case, because 17 is less than its parent, 26, it needs to percolate up until it is no longer less than its parent. So, we swap 17 with 26 and now 17's parent is 19, which again violates the min-heap property. We perform another swap. After that 17 is now greater than its new parent, which is 14. 17 is also smaller than all of its descendents because 19 was already smaller than all of its descendents.

Example Pseudo code

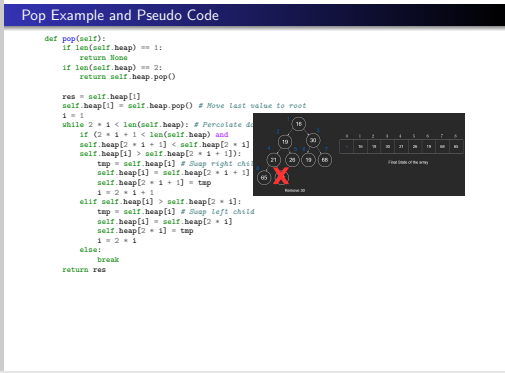
```
def push(self, val):
    self.heap.append(val)
    i = len(self.heap) - 1

    # Percolate up
    while i > 1 and self.heap[i] < self.heap[i // 2]:
        tmp = self.heap[i]
        self.heap[i] = self.heap[i // 2]
        self.heap[i // 2] = tmp
        i = i // 2
```

- The "//" indicates taking the floor of the resulting answer so we can round down, e.g. 5.5 becomes 5 since indices are whole numbers.
- Since we know the tree will always be balanced, the time complexity of the push operation is O(log n)



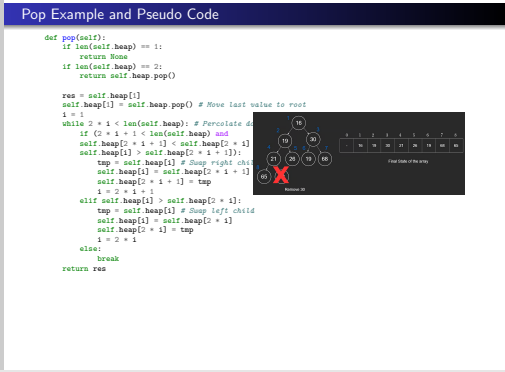
The obvious way:
Popping from a heap is more complicated than the push operation. One way that you might have already thought about is pop the root node and replace it with $\min(\text{left}_c hild, right_c hild)$. $The issue here is that while the order property is intact, we have violated the str$
$min(left_c hild, right_c hild) would require 19 to replace 16. Now, level 2 has a missing node i. e 19 is missing a$
The correct way:
The correct way is to take the right-most node of the last level and swap it with the root node. We have now maintained the structure property. However, the order property is violated. To fix the order property, we have to make sure that 30 finds its place. To do so, we will run a loop and swap 30 with $\min(\text{left}_c hild, right_c hild)$. $We swap 30 with 16, then 19 with 30. The resulting tree will look like the following$



The pseudocode shown above might seem daunting at first so let's go over it. If our heap is empty, there is nothing to pop, hence the return null. Our heap also could have just one node, in which case, we will just pop that node and don't need to make any adjustments. If the above two statements have not executed, it must be the case that we have children, meaning we need to perform a swap.

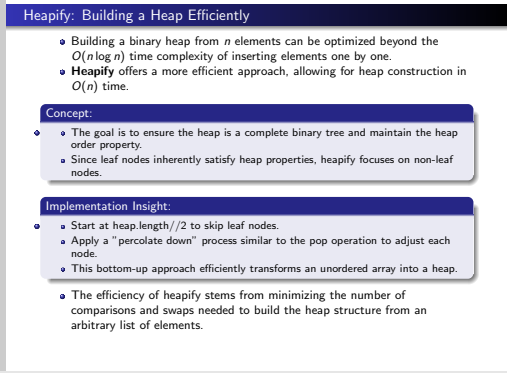
We store our 14 into a variable called res so that we don't lose it. Then, we can pop from our heap, and replace 30 to be at the root node.

Our while loop runs as long as we have a left child and we determine this by making sure 2 * i is not out of bounds. Then, there are three cases we concern ourselves with: The node has no children The node only has a left child The node has two children



When considering a binary heap, it is not possible to have only a right child because then it no longer is a complete binary tree and violates the structure property. Because we are guarenteed to have a left child in the while loop, we need to now check if the node also has a right child, which we check by 2 * i + 1. We also make sure that the current node is greater than its children because of the order property. We replace the node with the minimum of its two children.
If no right child exists and the current node's value is greater than its left child, we swap it with the left child.

If none of the above cases exectue, then it must be the case that our node is in the proper position already, satisfying both the order and the structural property.



Heapify

Recall that to build a binary search tree with n elements, the time complexity is O(n log n). If we build our heap of size n by pushing each element this would also run in O(n log n) time. But there is actually a more efficient algorithm known as Heapify, which allows us to perform this operation in O(n) time.

Concept

The idea behind using heapify to build a heap is to satisfy the structure and the order property. We need to make sure that our binary heap is a complete binary tree and that every node's value is at most its parent's value.

Because the leaf nodes can't violate the min-heap properties, there is no need to perform heapify() on them.

Since we are skipping all of the leaf nodes, we only need to start at heap.length // 2. Then, we need to percolate down the exact same way we did in the previous chapter in the pop() method. We will not be going over the code in detail as majority of it is the same as the pop() method.

Pseudo Code

```
def heapify(self, arr):
    # 0-th position is moved to the end
    arr.append(arr[0])

    self.heap = arr
    cur = (len(self.heap) - 1) // 2
    while cur > 0:
        # Percolate down
        i = cur
        while 2 * i < len(self.heap):
            if (2 * i + 1 < len(self.heap) and
            self.heap[2 * i + 1] < self.heap[2 * i] and
            self.heap[i] > self.heap[2 * i + 1]):
                # Swap right child
                tmp = self.heap[i]
                self.heap[i] = self.heap[2 * i + 1]
                self.heap[2 * i + 1] = tmp
                i = 2 * i + 1
            elif self.heap[i] > self.heap[2 * i]:
                # Swap left child
                tmp = self.heap[i]
                self.heap[i] = self.heap[2 * i]
                self.heap[2 * i] = tmp
                i = 2 * i
            else:
                break
        cur -= 1
```

Starting from the first non-leaf node, we will percolate down, the exact same way we did in the pop() function. After each iteration, we are going to decrement the index by 1 so we can perform heapify() on the next node, all the way until index 1.

The visual below demonstrates heapify() being performed on all nodes starting from index 4. The nodes in the blue rectangles are leaf nodes.