CS 240: Data Structures and Data Management **Spring 2025**

Tutorial 02: May 23

1. **Amorzed insertion in array Analysis**
Prove that inserting n elements into a dynamic array (= std::vector) takes $O(n)$. After proving that, try breif explain why a single insertion takes $O(1)$ time when "amortized" (averaged over operations)
Solution.

Proof. We assume dynamic arrays (= std::vector):

- Keep track of size and capacity of array.
- If size = capacity, copy items over to new array (twice as big). This takes $\Theta(n)$ time but happens after $\Theta(n)$ "cheap" insertions.

When inserting up to n elements, the number of times we resize is about $\log n$. So the total number of elements copied is:

$$\underbrace{1+2+4+8+\cdots+2^{\lfloor \log n \rfloor}}_{\sum_{i=0}^{\lfloor \log n \rfloor} 2^i} = 2^{\lfloor \log n \rfloor + 1} - 1 = 2n - 1$$

So the total cost of all copying is $O(n)$. For simple insertion without resizing, the cost is $O(1)$ each time, so intotal the cost is $O(n)$ as there's n elements. In conclusion, copying takes $O(n)$, and simple insertion takes $O(n)$, so the total cost of inserting n elements into a dynamic array is also $O(n)$. □

Inserting n elements takes $O(n)$, we have average cost equal to $\frac{O(n)}{n} = O(1)$. (unformal)

2. **Average Runtime Analysis**
Suppose A is an array containing n distinct elements. In addition, assume each element is in between 1 and n, inclusive. Analyze this pseudo-code to determine a tight bound on the average number of question mark (?) that are printed, rather than a runtime. You may assume n is divisible by 2.

```
mystery(A, n)
    count = 1
    for i = 1 to n-1
        if A[i] is divisible by A[0]
            count++
    for i = 1 to count
        print("?")
```

Solution.
We know that $\mathcal{I}_n$ is finite, given by the condition. Then, we can consider following cases. Note that $T(n)$ represents number of ? marks printed.
First, `A[0]` may be 1. In that case, `count` will be set to n and there are $(n-1)!$ instances where `A[0]` is 1.
Similarly, `A[0]` may be 2. In that case, `count` will be set to $\frac{n}{2}$ and there are $(n-1)!$ instances where `A[0]` is 2.

When `A[0]` is $\frac{n}{2}$, then count will be set to 2 and there are $(n-1)!$ instances where `A[0]` is $\frac{n}{2}$.
When `A[0]` is greater than $\frac{n}{2}$, then we know that count will then be set to 1.

Then, what we want to notice is that when `A[0]` is x, number of ? is written as $\lfloor \frac{n}{x} \rfloor$. Then, we can write summation as follows.

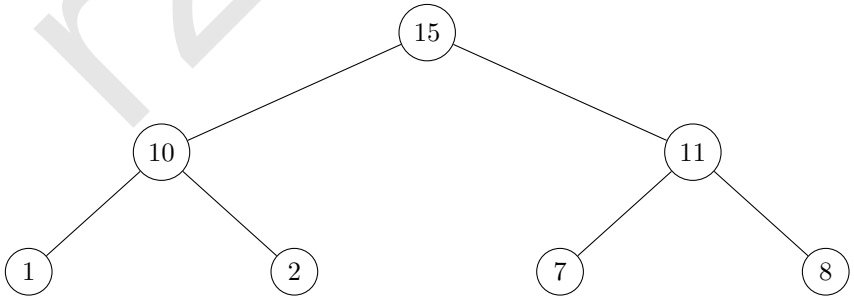
$$\sum_{I \in \mathcal{I}_n} T(I) = \sum_{x=1}^{n} \lfloor \frac{n}{x} \rfloor (n-1)!$$
$$= (n-1)! \sum_{x=1}^{n} (\lfloor \frac{n}{x} \rfloor)$$

To make our life a little easier, let us drop floor. Then, putting it back to $T^{avg}(n)$, we get

$$T(n) = \frac{1}{n!} \sum_{x=1}^{n} (n-1)! \frac{n}{x}$$
$$= \frac{1}{n} \sum_{x=1}^{n} \frac{n}{x}$$
$$= \sum_{x=1}^{n} \frac{1}{x}$$

We know that above summation is in $\Theta(logn)$.

3. **Max-Heap Operations**
Insert 27 and 9 into the following heap, and then perform a delete-max operation on the resulting heap.



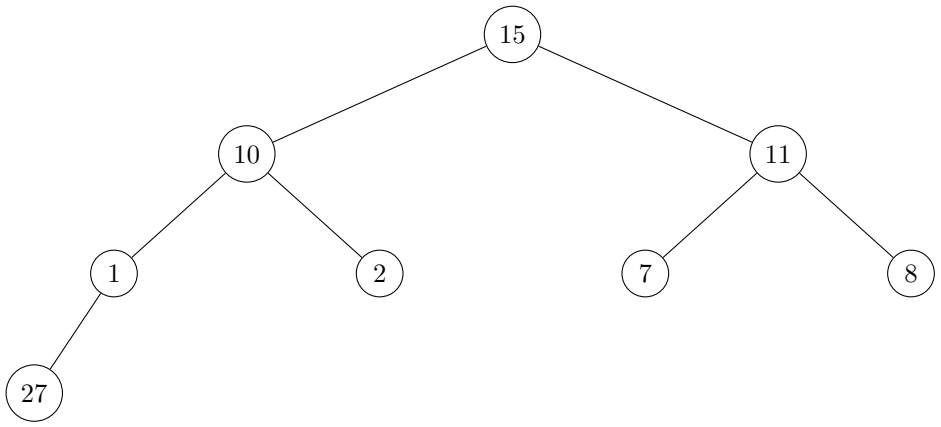
Solution.

A (max) heap is a binary tree with two properties:

- Structural Property: All levels of the heap are full, except possibly the last level, where the elements are left-justified. There should be no empty nodes at a position further left at the same level as a non-empty node.
- Heap-Order Property: The key for each node is smaller or equal to the key of its parent.

Insert(27):

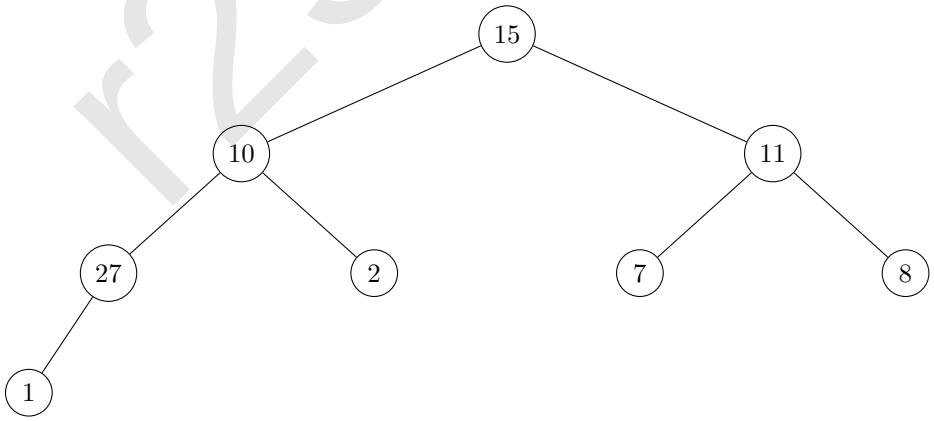
In order to maintain the structural property, the element should initially be inserted into the last level of the heap, at the leftmost available position. If the last level is full, then the element should be inserted at a new level at the leftmost position.



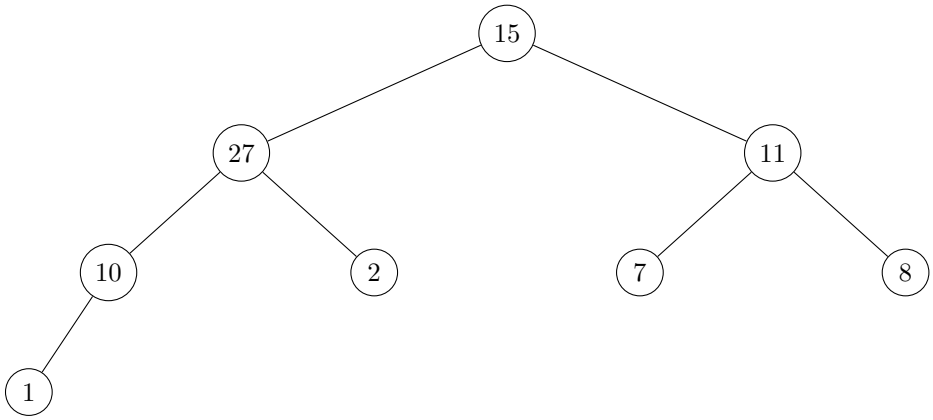
The new element is inserted while preserving the structural property. However, the new element might violate the heap-order property due to being larger than some of its ancestors. To fix this, we call fix-up on this newly inserted element.

For fix-up, the inserted element's key is compared with its parent's key. If the parent has a smaller key, then the two elements will swap places. The process is repeated with the new parent of the inserted element, and continues until the inserted element has a parent with a larger key, or if the inserted element becomes the root.

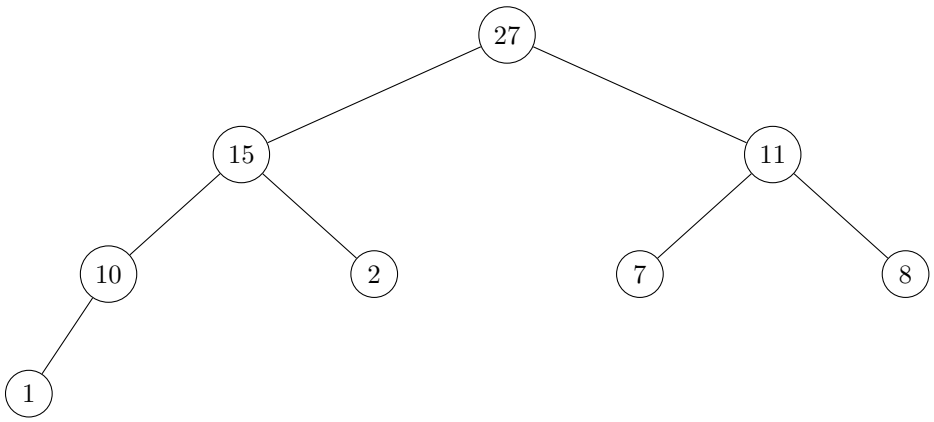
$27 > 1$, so the two will swap



$27 > 10$, so the two will swap



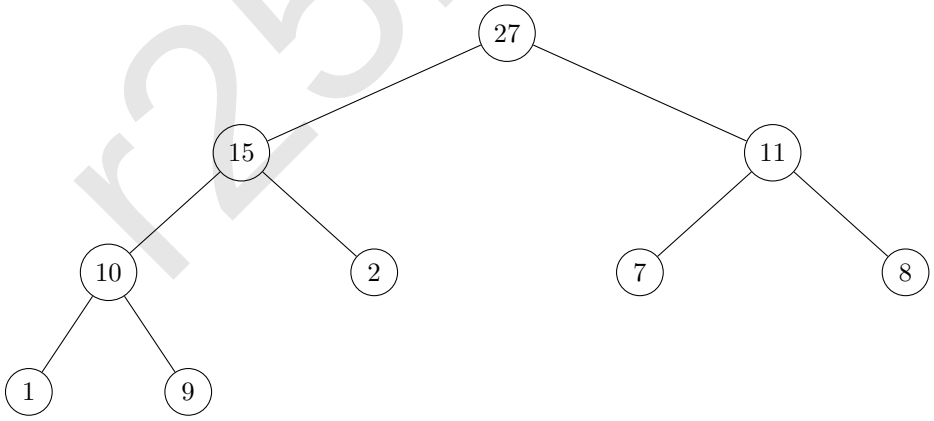
$27 > 15$, so the two will swap



The node with key 27 is now the root, so the fix-up is complete. This is the resulting heap from inserting 27.

Insert(9):

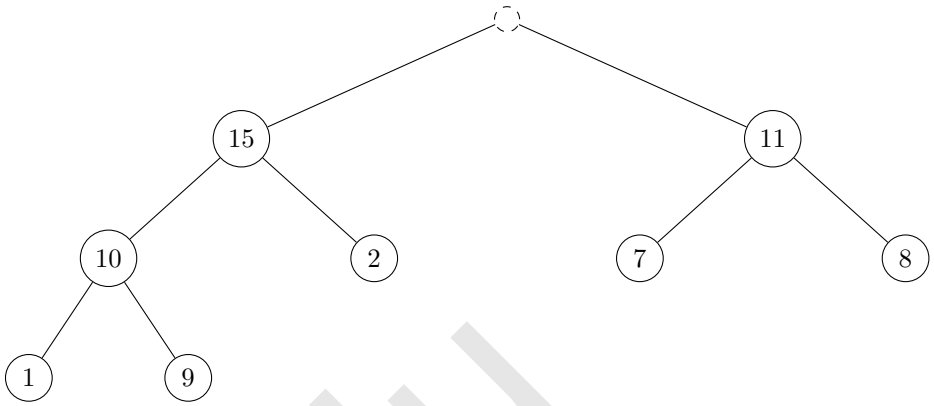
We insert 9 at the leftmost available spot of the lowest level.



We still call fix-up on the 9, but this time its parent already has a larger key, thus satisfying the heap-order property. So the insertion is completed without any swaps.

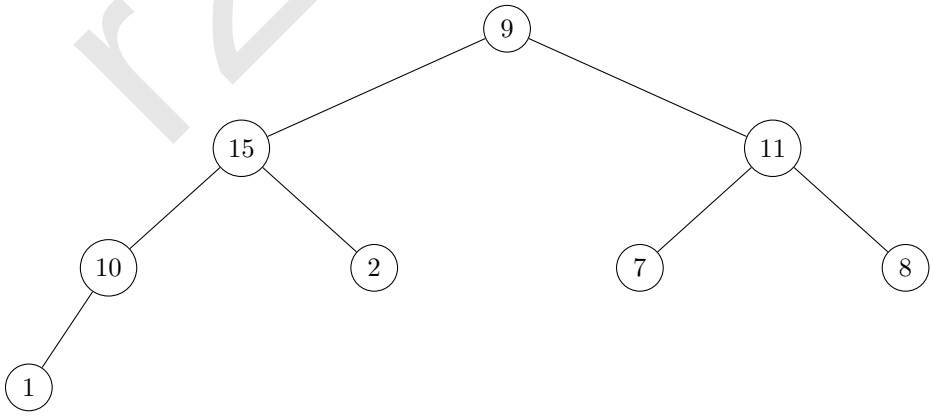
deleteMax:

The other operation of (max) heaps is the deleteMax operation, consuming no arguments, but it removes and returns the element of the heap with the largest key. Due to the heap-order property, this is always the root of the heap, which is 27 in this case.



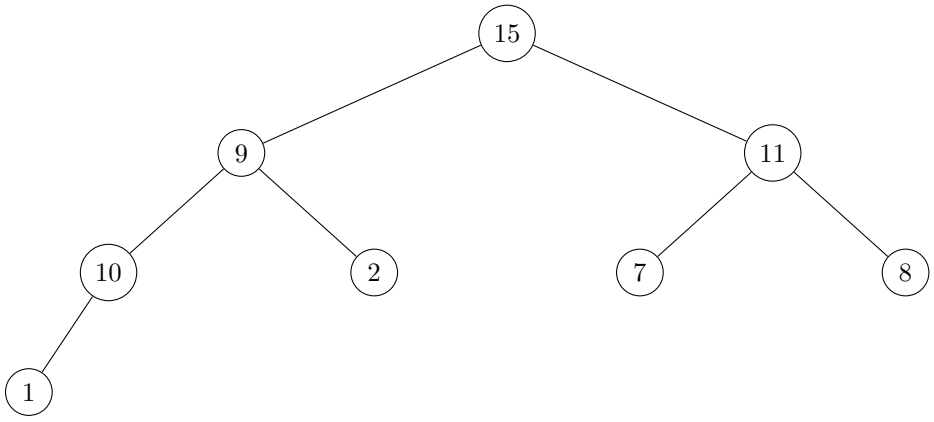
We cannot simply leave the root as empty, so another node in the heap must be moved to its place. It may be tempting to move 15 to the root, since it is now the largest key of the heap, but this will not restore the structural property.

The only node we can afford to relocate is the rightmost element of the lowest level, since its displacement will still allow the lowest level to remain left-justified. Therefore, to restore the structural property, we move this last element (rightmost element of lowest level) to the root.

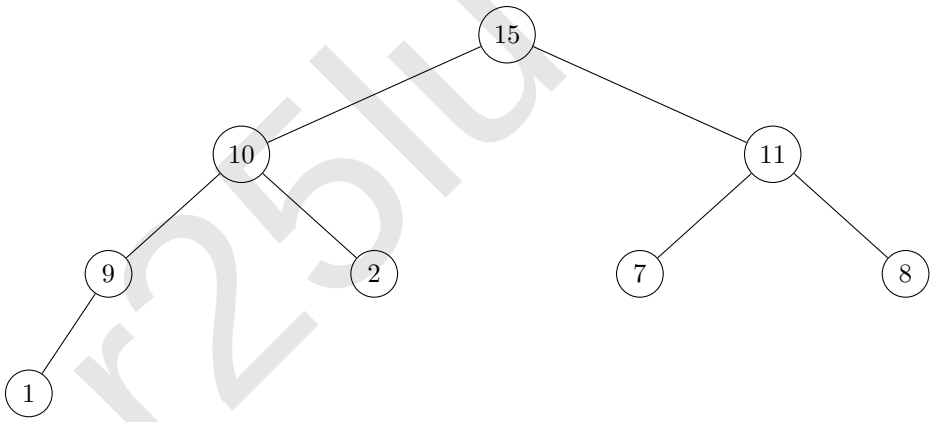


Although the structural property is restored, this new root element will almost always violate the heap-order property, since its key may be smaller than some of its descendants. This is fixed using the fix-down function
Each step of fix-down involves three keys: the violator's key (initially the new root), and the keys of its two children. The largest of these three keys should become the parent of the other two. To achieve this, we first determine which of the two children keys is larger, and compare this larger key with the key of its parent (the violator). If the parent has a smaller key, then it is swapped with this child (the child with the larger key).

Here, the children of the violator (root) are 15 and 11. The larger of these two children is 15, so it is compared with the violator key 9. Since $9 < 15$, we swap the two.



The children of 9 are 10 and 2, with 10 being larger. We have $9 < 10$, so we swap the two



The only child of 9 is 1, but we have $9 \geq 1$ (satisfying the heap-order property), so we end the fix-down without any further swaps.

This is the resulting final heap. The deleted element (with key 27) is returned by this deleteMax operation.

In general, when we ask you to perform an insert or deleteMax, the question will usually specify if you need to show the intermediate heaps or if you just need to show resulting heaps after those operations.