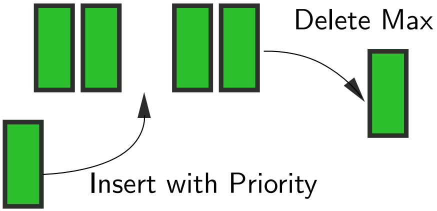
# Priority Queues

## (Max) Priority Queue = Max-In-First-Out

Priority Queue is an Abstract Data Type defined by its operations:

- `insert(value, priority)` – inserts a value with a certain priority into the data structure
- `deleteMax()` – retrieves the value with the highest priority
- `update(value, priority)` – changes the priority of the value
- `isEmpty()` – returns `true` if the priority queue is empty



Although considering such data type might seem unmotivated now, Priority Queues appear again and again in the applications and we will see later in the module how they could be used.

In this lecture we explain Max Priority Queues. Similarly, one can consider Min Priority Queues defined correspondingly.

Note that, unlike in normal queues, two things with the same priority might not be retrieved in the order in which they were inserted. The order in which they are retrieved is not specified and will depend on the implementation.

If we only wanted to compute the maximum once, it wouldn't be worth the effort to build the data structures. Priority queues are useful when we need to get the maximum repeatedly.

## Example

Starting from an empty priority queue, `insert(A, 1)`, `insert(B, 5)`, `insert(C, 3)`, followed by `deleteMax()` gives `B` and then another `deleteMax()` gives `C`.

## Usage

When designing a print queue where teachers have a higher priority than students. Then, students always have to wait for teachers' jobs currently in the print queue to finish, even if they've sent their file to print earlier.

In general, priority queues are useful whenever we repeatedly need the maximum of a (slightly) changing collection.

## Naive implementation: store in an array

**(1)** Store the elements in the array *unordered*

- `insert(value, priority)` – store the element at the end of the array $\approx \mathcal{O}(1)$
- `deleteMax()` – find the element with the highest priority, delete it, shift elements to the left by 1 $\approx \mathcal{O}(n)$

**(2)** Store in an array, *ordered* by priority (the highest at the end):

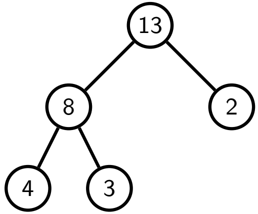
- `insert(value, priority)` – find the position where to store the element, shift to the right the elements starting from that position $\approx \mathcal{O}(n)$
- `deleteMax()` – return the last stored element $\approx \mathcal{O}(1)$

## Binary Heaps

Better solution: store the elements in a binary tree! However, unlike in Binary Search Trees, the highest-priority node is at the top!

*Invariants* of **heap trees** are as follows:

1. The priority of every node is higher than (or equal to) that of its children.
2. All levels are completely filled, except possibly the last one, which is partially filled from the left.



$\Longrightarrow$ the height of a heap tree is therefore $\mathcal{O}(\log n)$.

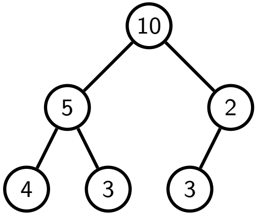
In the nodes we store other things than just the priority but for the sake of understanding of the behaviour of heaps we only show the priorities. In fact, the priorities are the only thing that matters for the behaviour of binary heaps.

To calculate the height, the same as in perfect binary trees, the number of elements is exponential in the number of nodes. Therefore, the number of levels is logarithmic in the number of nodes.

(If we know that the heap tree has $h$ levels, then it has one more element than the perfect binary tree with $h - 1$ levels, or is exactly perfect binary tree of height $h$, or something in between.)

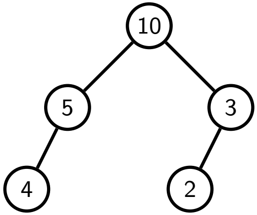
Recall that invariants, are like the tidiness rules. When we're cooking the dinner, the kitchen can be a mess but we have to clean it afterwards.

## Examples: Are the following heap trees?



No, because the left child of 2 is 3, that is, a bigger value.

(We violate condition 1)



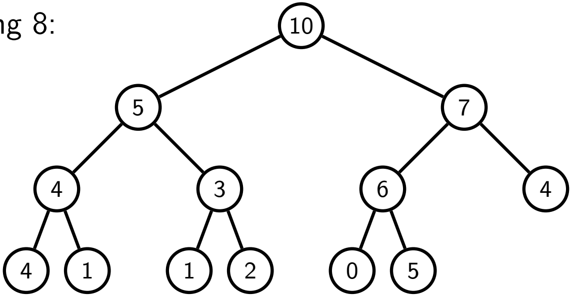
No, because the last layer is not filled from the left.

(We violate condition 2)

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

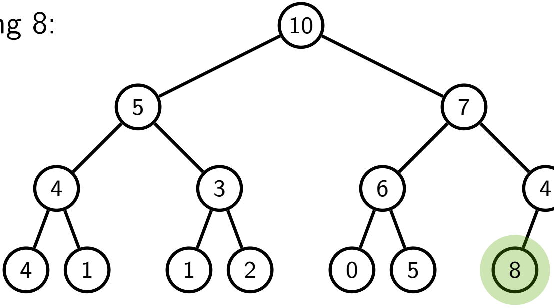


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

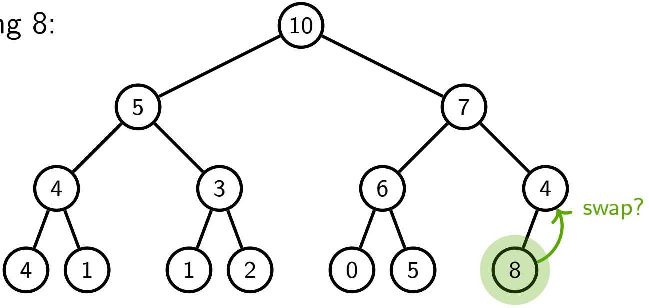


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

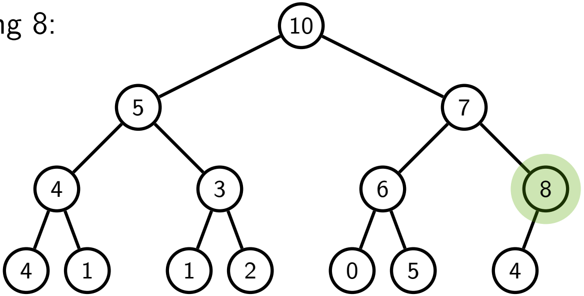


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

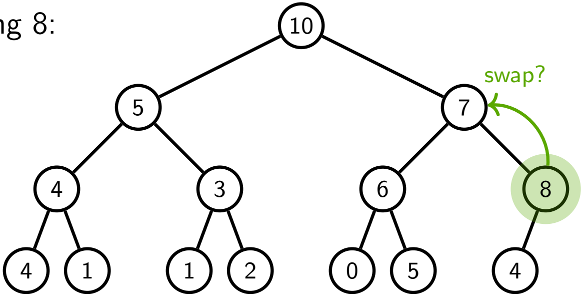


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

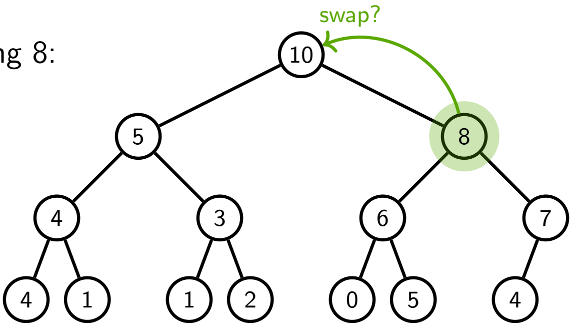


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:

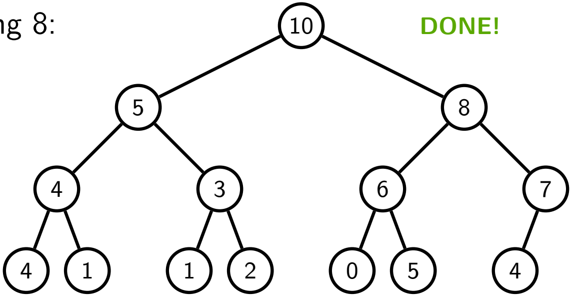


Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

## Insertion

**Idea:** Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

Inserting 8:



**DONE!**

Initially, we make sure the second condition of heap trees holds, then we fix the first one by bubbling up.

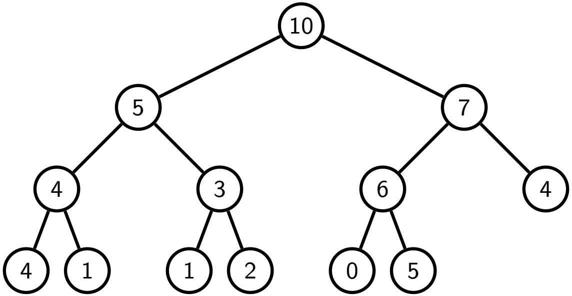
Initially, we make sure that the second invariant of heap trees is satisfied. Then, to fix the second invariant, we keep swapping the inserted node with its parent as long as its parent has a smaller priority (= we are bubbling up the node).

Because of the specifics of our implementation, we will see that it is actually fast to add/remove the last node of the last level. Similarly, it is fast to access the parent node (unlike in ordinary binary trees).

## Delete Max

**Idea:**

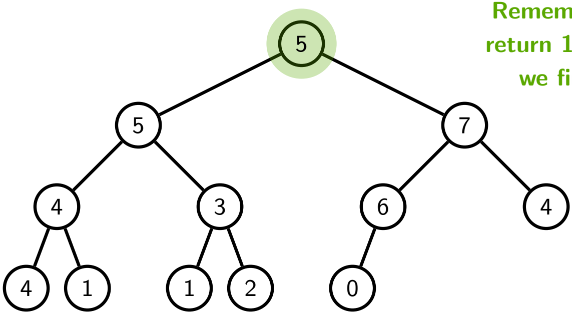
1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Delete Max

**Idea:**

1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.

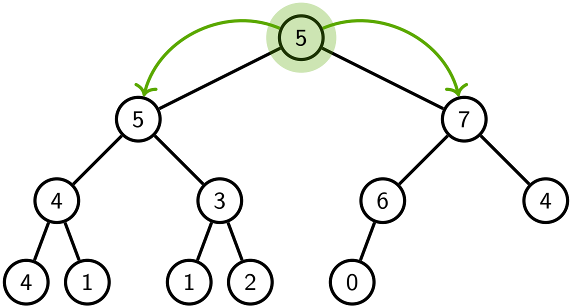


**Remember to return 10 after we finish**

## Delete Max

**Idea:**

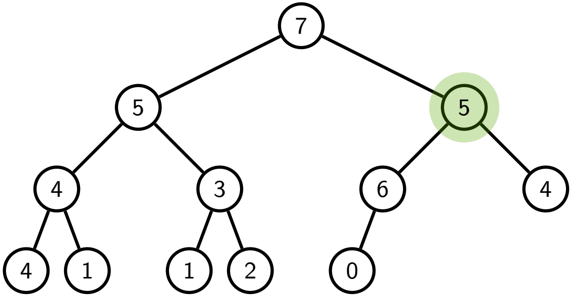
1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Delete Max

**Idea:**

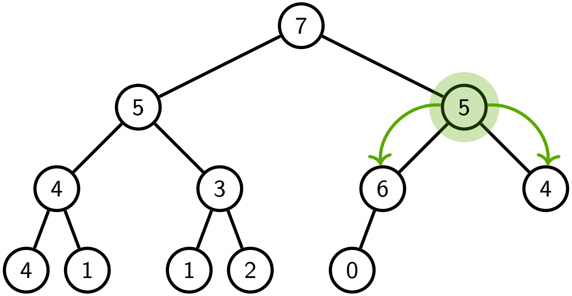
1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Delete Max

**Idea:**

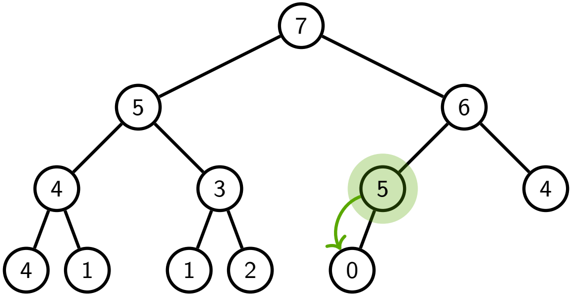
1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Delete Max

**Idea:**

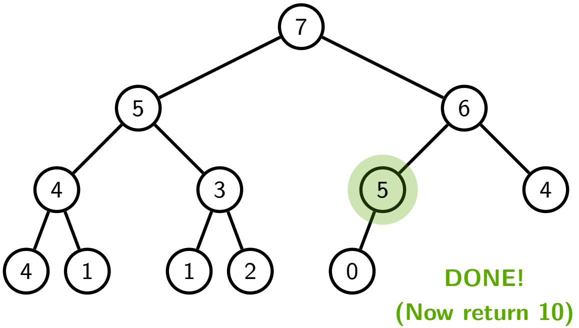
1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



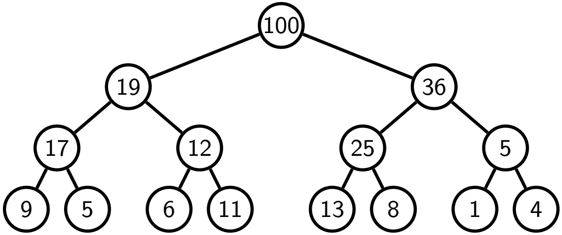
## Delete Max

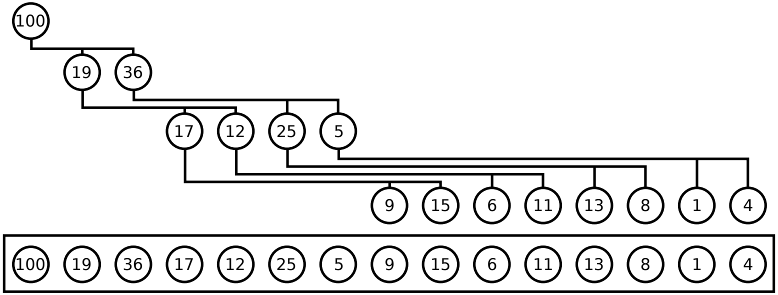
**Idea:**

1. Make the last node in the last level the new root.
2. Keep swapping it with the higher priority child as long as any of its children has a higher priority.



## Storing heap trees in arrays (the picture)





(source: wikipedia)

Idea: First skew the tree, then flatten it onto an array.

This way, we store the elements of the heap tree in an array such that

1. The root node (= level 0) is stored on the first position,
2. the nodes from level 1 are stored on the next two positions,
3. the nodes from level 2 are stored on the next four positions,
4. the nodes from level 3 are stored on the next eight positions,
5. ...

This way we can store any tree which satisfies the second condition of heap trees. (Such trees are called *complete*, in the literature.)

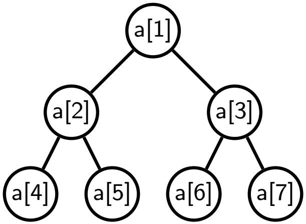
## Storing heap trees in arrays

We start counting from 1!!!

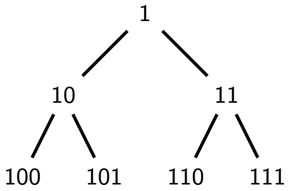
$\Longrightarrow$ The root node is stored on position `1`.

For a node stored on position `i`,

1. its left child is on position `2*i`,
2. its right child is on position `2*i + 1`, and
3. its parent is on position `i div 2`.



or in binary



To see why the parent position is computed this way, consider the node on position `i`. Its children are on positions `2*i` and `2*i+1`. Then, to compute their parent (back again), we compute either

`2*i div 2` or `(2*i + 1) div 2`

And in both cases we get `i` back because `div` throws away the remainder.

Observe that this representation allows us to access the parent in a constant time. Moreover, if we want to read the value stored in the last node of the last level, we just read the last value stored in the array. Similarly, if we want to add one extra node to the last level, we just store it after the last stored value in the array.

## Insert (pseudocode)

Inserting `p` to a heap which already has `n` elements stored in it:

```
void insert(int p, int[] heap, int n) {
  if (n == MAXSIZE)
    throw HeapFullException;
  n = n + 1;
  heap[n] = p;    // insert the new value as the last
                  // node of the last level
  bubbleUp(n, heap, n);  // and bubble it up
}
```

```
void bubbleUp(int i, int[] heap, int n) {
  if (i == 1) return;  // i is the root

  if (heap[i] > heap[parent(i)]) {
    swap heap[i] and heap[parent(i)];
    bubbleUp(parent(i), heap, n);
  }
}
```

The function `parent(i)` just returns `i div 2`.

Swapping two values could be done as follows

```
1. int tmp = heap[i];
2. heap[i] = heap[j];
3. heap[j] = tmp;
```

**Remark:** In fact, we do not have to throw the `HeapFullException` whenever the heap is full. Instead we can double the size of the array and insert into this bigger one. In week 4 we computed that *amortized* complexity of this operation would still be $\mathcal{O}(1)$.

## Delete Max (pseudocode)

```
int deleteMax(int[] heap, int n) {
  if (n < 1)
    throw EmptyHeapException;

  int result = heap[1];         // remember root's value
  heap[1] = heap[n];            // the last node is root now
  bubbleDown(1, heap, n-1);     // bubble down this new root
  return answer;
}
```

```
void bubbleDown(int i, int[] heap, int n) {
    // exercise!
}
```

## Comparison of priority queues implementations

| Operation | Binary Heaps | Binomial Heaps | Fibonacci Heaps |
|---|---|---|---|
| Insert | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)^\star$ | $\mathcal{O}(1)$ |
| Delete Max | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)^\star$ |
| Update | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)^\star$ |
| Merge | $\mathcal{O}(n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |
| Heapify | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

Where $\star$ means that it is the amortized complexity.

The missing operations:

1. `merge(priorityQueue1,priorityQueue2)` – merges two priority queues into one
2. `heapify(array with priorities)` – builds a priority queue out of an array of elements with priorities

Priority Queues are defined by their abstract data type specification. We showed one implementation but there are many other implementations which are, in fact, better behaved.

Moreover, there is a strict version of Fibonacci heaps which have all operations in the Average Case complexity the same as shown here in the table.

When using heaps in your favourite programming language, you should be aware which implementation is used there to determine (to know the corresponding time complexities). You'll probably find Fibonacci heaps most often but not their strict version, it is quite new.

## Heapify (bonus slide)

```
void heapify(int[] arr, int n) {
    for (i = n div 2; i > 0; i--) {
        bubbleDown(i, arr, n);
    }
}
```

In the worst case, the root node needs to swap with $h$-many nodes, the nodes on level 1 swap with $(h-1)$-many nodes and so on, where $h$ is the height of the heap tree. In total, the number of swaps is

$$C(h) = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}.$$

However, $C(h) \leq 2^{h+1}$ and, since $h$ is in $\mathcal{O}(\log n)$, $C(h)$ is in $\mathcal{O}(n)$.

We will show $C(h) \leq 2^{h+1}$. The sum

$$C(h) = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}$$

can be also written as

$$C(h) = \sum_{i=0}^{h} 2^i (h-i).$$

Then,

$$C(h) = 2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j}$$

and since $\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$ we obtain that $C(h) = 2^h \times 2 = 2^{h+1}$.

A more elementary proof of this fact can be found in Martín Escardó's notes (uploaded to Canvas). In fact, Martín proves even better estimate, i.e. that $C(h) \leq 2^{h+1} - h - 2$.