CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 17

Tuesday, November 2nd 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Introduction to data structures
- Linked lists
- Iterators
- Binary trees

OUTLINE

- Binary tree traversal
- Priority queues and heaps

BINARY TREE TRAVERSAL

Recall: the distinction between ordinary trees and binary trees

1. An ordinary tree can not be empty, that is, it always has a *root* node. Each node in this tree can have zero or more children.
2. A binary tree ***can be empty*** and each of its nodes can have 0, 1 or 2 children. We further distinguish between the *left* child and *right* child.

- Binary trees are one of the most fundamental data structures in Computer Science.
- Binary trees appear in many places and it is more likely that they will meet you rather than you will meet them.
- It is therefore important to have a good understanding of this data structure.
- Their "difficulty" lies in their *recursive* nature.

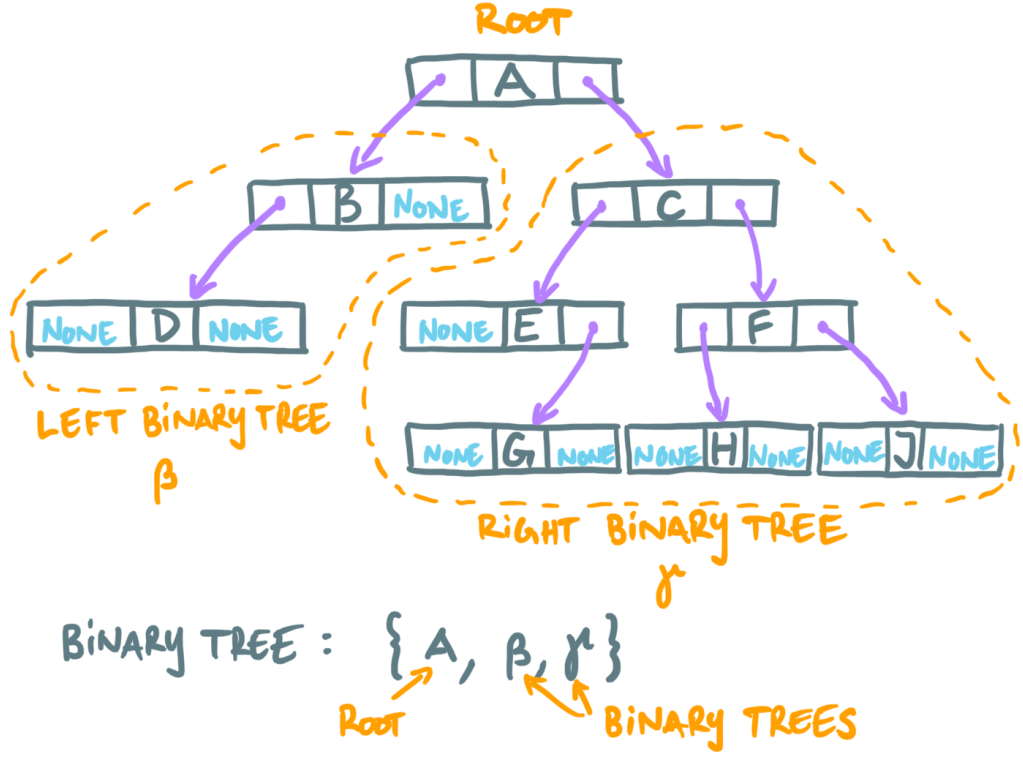
BINARY TREE TRAVERSAL

Another way to look at a binary tree:

A binary tree is a finite set of nodes that is either *empty* or consists of a *root together with two binary trees*.

Note: this definition is *recursive*!

Example binary tree:

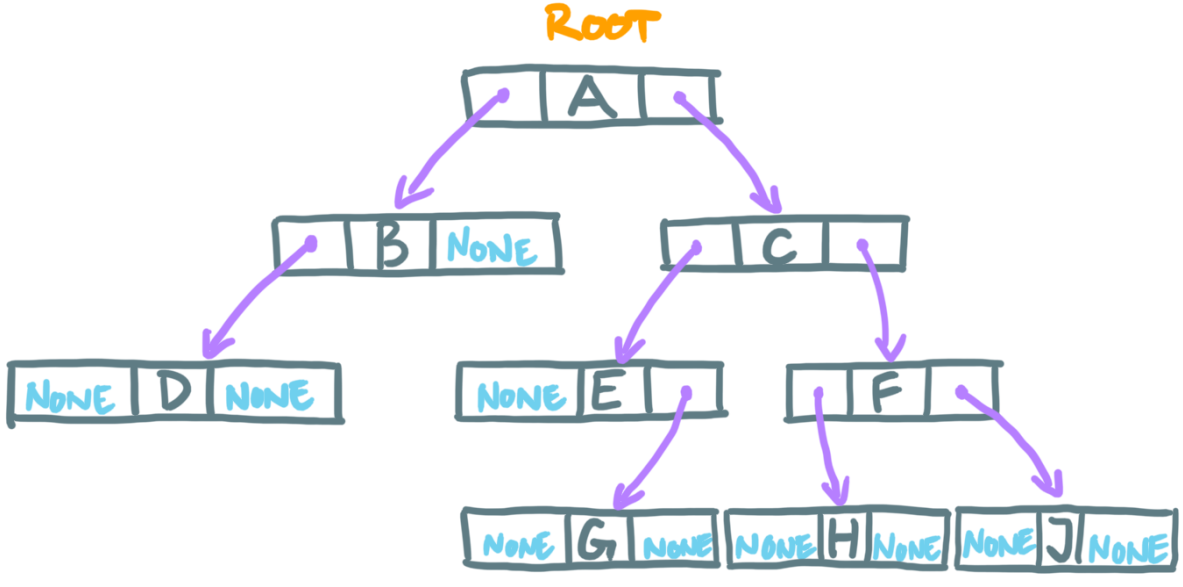


The set $\{A, \beta, \gamma\}$ defines a binary tree. How does the set look like for the binary tree with root B?

BINARY TREE TRAVERSAL

There are *three* principal ways to traverse a binary tree:

1. *Preorder* traversal:
    i. Visit the root
    ii. Traverse the left subtree
    iii. Traverse the right subtree
2. *Inorder* traversal:
    i. Traverse the left subtree
    ii. Visit the root
    iii. Traverse the right subtree
3. *Postorder* traversal:
    i. Traverse the left subtree
    ii. Traverse the right subtree
    iii. Visit the root



- Preorder:
- Inorder:
- Postorder:

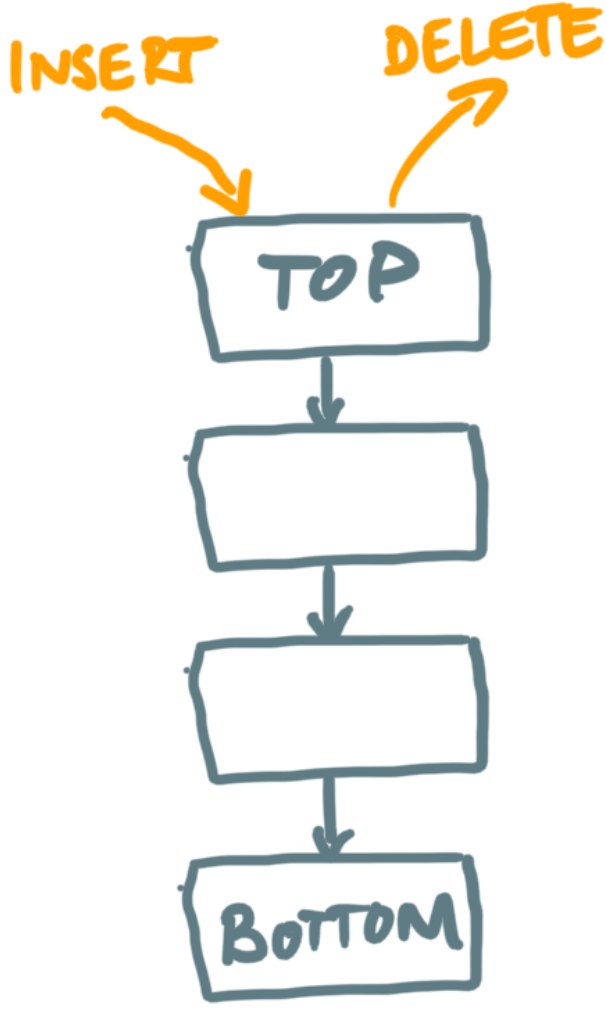
BINARY TREE TRAVERSAL

- In each of the three principal ways of tree traversal, we have visited each node once.
- In the previous exercise we have just printed the node ID when we visited it.
- For more useful applications a tree node may hold a reference to other data that we can operate on. An example includes evaluation of dual numbers.

BRIEF RECAP

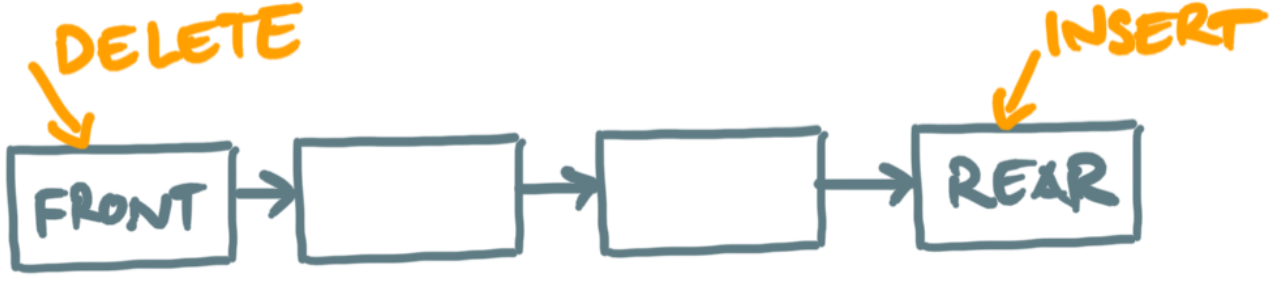
- We have discussed linked lists, a linear data structure, and (binary) trees, a nonlinear data structure, in more detail so far.
- Trees and in particular binary trees are a fundamental data structure that appears in many places in Computer Science.
- Other linear data structures are stacks, queues and deques.

STACK



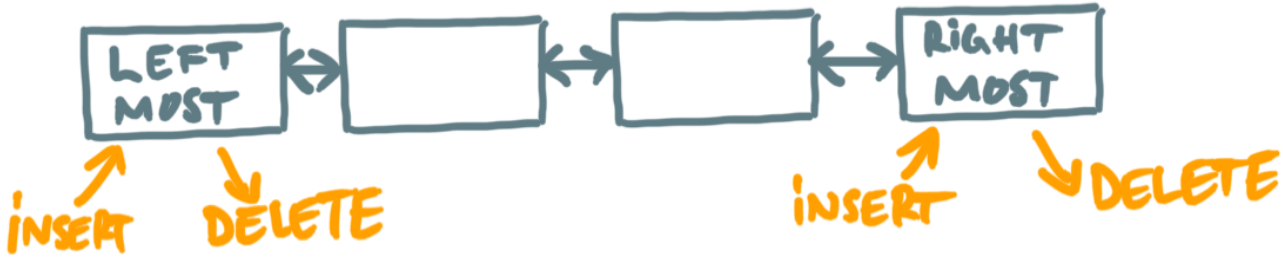
- A ***stack*** is a linear list for which all insertions, deletions and usually all accesses are made at one end of the list.
- This is often referred to as Last-In-First-Out (LIFO) stack or list.
- An example where this data structure is used is for executing threads on your computer or similarly when we execute `python` functions with pythontutor.com. Is the stack a useful data structure for *recursive* function calls?

QUEUE



- A ***queue*** is a linear list for which all insertions are made at one end of the list and all deletions are made at the other end. Usually accesses are made where we delete elements.
- This is often referred to as a First-In-First-Out (FIFO) queue or list.
- A queue keeps the order of how elements arrive.

DEQUE



- A ***deque*** (double-ended-queue) is a linear list for which all insertions and deletions are made at the ends of the list. Accesses are usually made at both ends as well.
- A deque is more general than a stack or a queue. It has some properties in common with a deck of cards which is why it is pronounced as "deck".

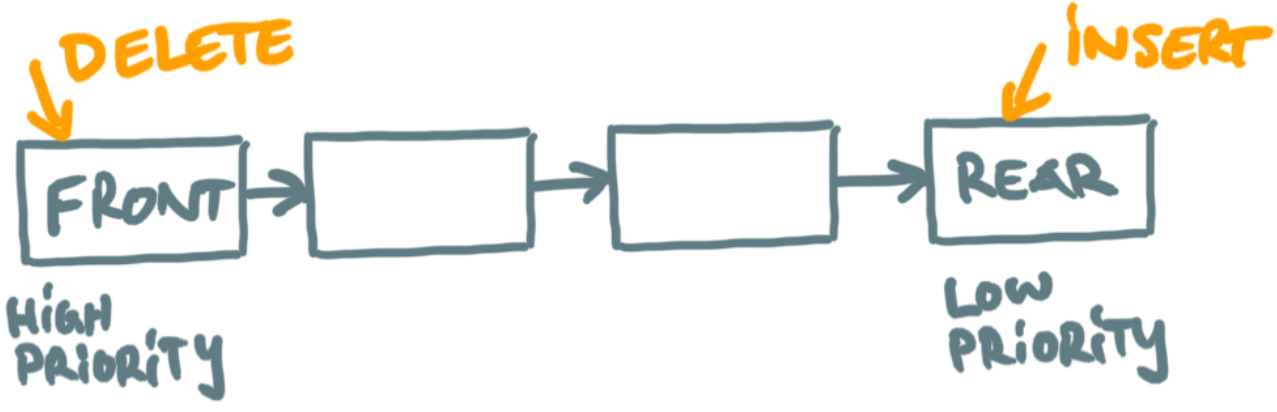
PRIORITY QUEUE

- Assume items in a list have a `key` that is comparable.
- Often a data structure that behaves like "smallest-in-first-out" (or equivalently "largest-in-first-out") is useful.
- In the case of "smallest-in-first-out" every deletion removes the element with the *smallest* `key`.
- A list that gives certain elements priority is called a ***priority queue***.
- This implies a certain *order* that must be maintained.

PRIORITY QUEUE EXAMPLES

- Operating systems or job schedulers on compute clusters use priority queues to schedule jobs.
- If you need to store data according to "least recently used", priority queues are useful.
- Maintain a priority order among your customers.

PRIORITY QUEUE IMPLEMENTATION



- We need a method for element insertion that maintains the priority order.
- We need a method to remove the element with highest priority.
- We need to be able to obtain the element with highest priority (essentially the same as removal without actually removing the element).

How would you go about this?

PRIORITY QUEUE IMPLEMENTATION

- You could simply use a sorted list. Insertion of new elements is $\mathcal{O}(n)$, removal and access are $\mathcal{O}(1)$.
- You could keep a reference to the element with highest priority (e.g. a pointer). Insertion and access are $\mathcal{O}(1)$, removal is $\mathcal{O}(n)$.

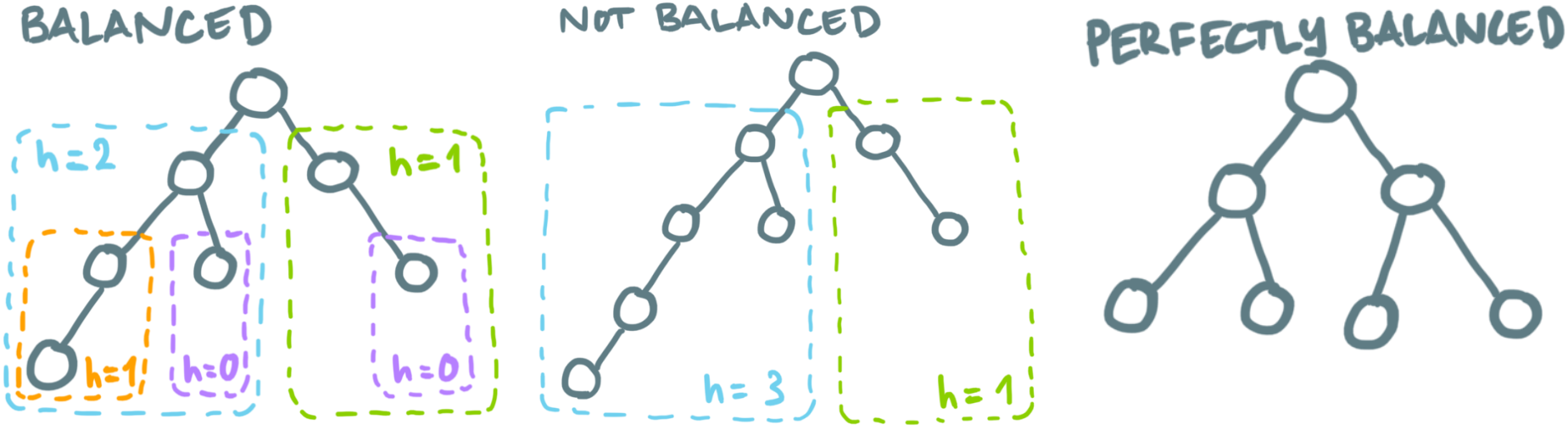
Both of these approaches are not efficient when the number of elements n is large.

It is possible to use a *balanced* binary tree which can be represented in a compact form using an array of keys only (no extra overhead required for the bookkeeping of nodes in the tree). This will lead us to the notion of a (binary) ***heap***.

Note: a heap is a *special arrangement of values*, it is completely unrelated to a dynamic pool of memory that is often referred to as "heap".

BALANCED BINARY TREES

- The height h of a tree is given by the maximum level of the tree (see sketch in slides of previous lecture).
- A binary tree is balanced if the *height difference* between left and right subtrees is no larger than 1.
- For a *perfectly* balanced binary tree the relation $\lfloor \log_2(n) \rfloor = h$ holds, where n is the number of nodes in the tree.



HEAP

A ***heap*** is defined as a sequence of n keys

$$h_1, h_2, \ldots, h_n$$

such that

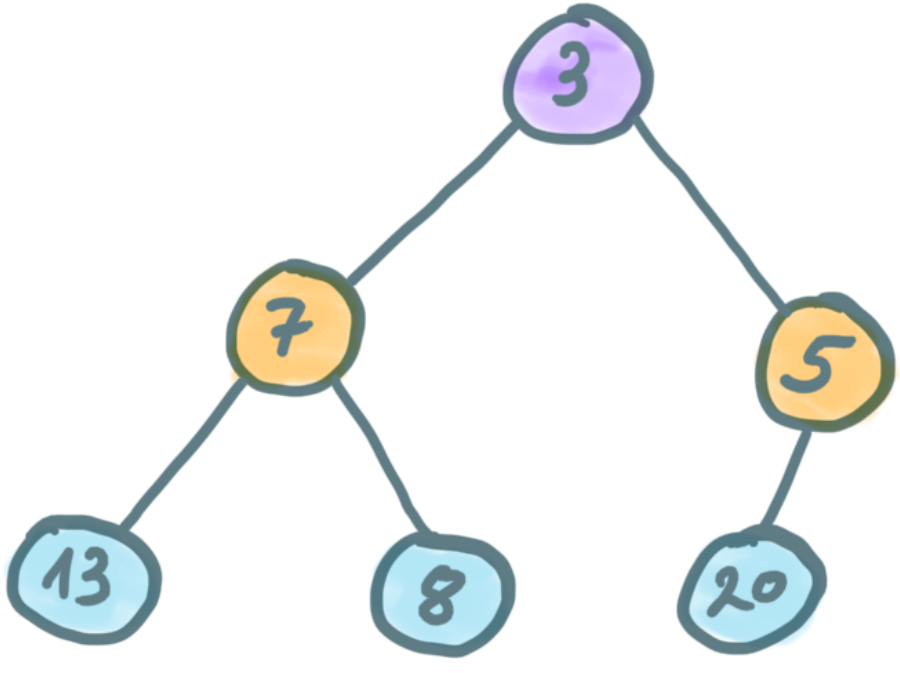
$$h_i \leq h_{2i}$$
$$h_i \leq h_{2i+1}$$

for all $i = 1, \ldots, n/2$. Here the *least* element is $h_1 = \min(h_1, h_2, \ldots, h_n)$.
We can just as well define the heap with "$\geq$" instead. The greatest element then is $h_1 = \max(h_1, h_2, \ldots, h_n)$. We refer to a min-heap for the former and max-heap for the latter definition, respectively.

A heap can therefore be cast into a binary tree, where the key h_i of a tree node satisfies the properties above relative to its two children.

HEAP

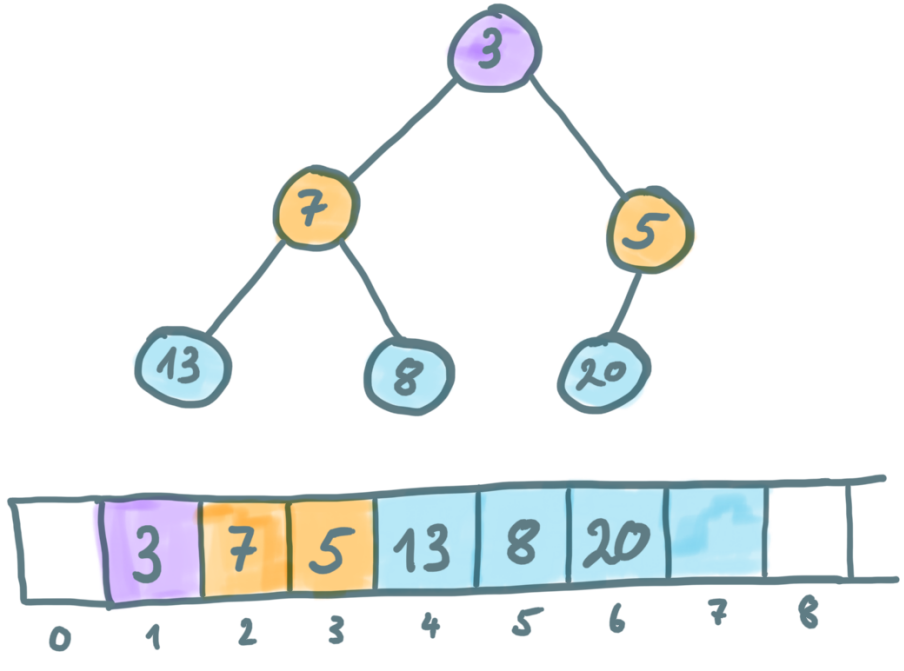
Heap ordered tree:



- A heap ordered binary tree is a balanced binary tree that satisfies the heap property.
- The key of the root node corresponds to the *least* element.
- If you change $\leq$ to $\geq$ in the heap property, the key in the root node corresponds to the greatest element.

HEAP

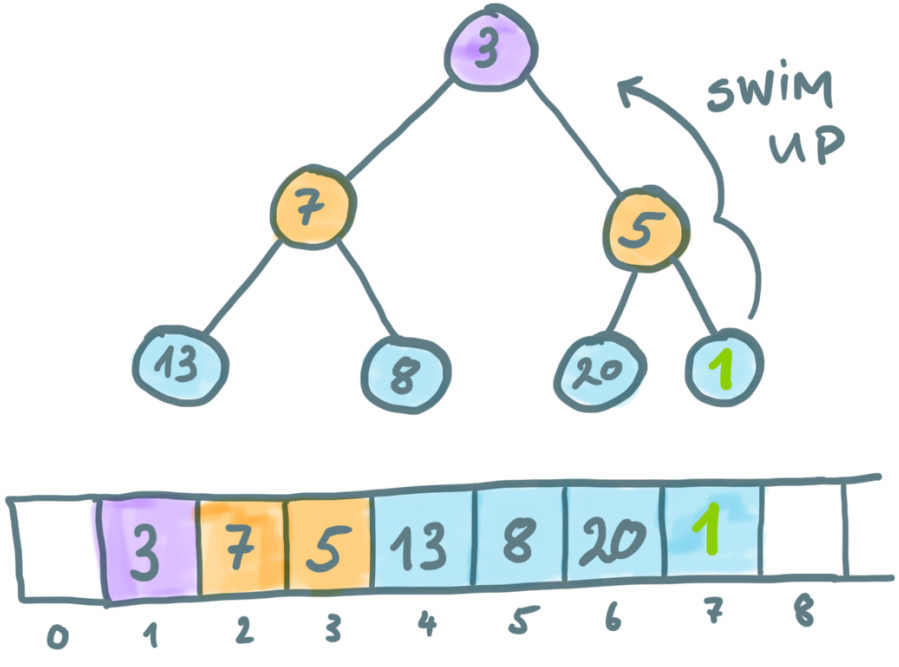
Binary heap (or just "heap"):



- A binary heap or simply "heap" is a heap ordered binary tree compactly represented with an *array*.
- If a parent node is at index i in the array, its left and right child have indices $2i$ and $2i + 1$, respectively,
- What are the corresponding child indices if the root node is at index `0` in the array?
- A heap is the optimal data structure for a ***priority queue***.

PRIORITY QUEUE WITH HEAP

Element insertion:



Swim steps for insert of value `1` :

1. `k: 7 | k//2: 3 | array[k]: 1 | array[k//2]: 5`
2. `k: 3 | k//2: 1 | array[k]: 1 | array[k//2]: 3`

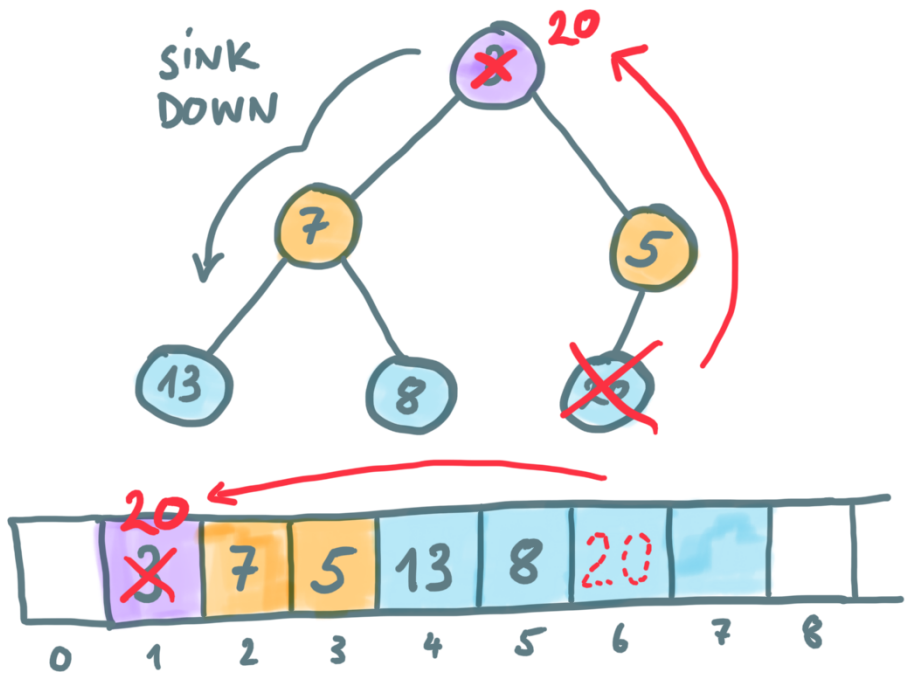
- A new element is inserted at the index $n+1$.
- This will destroy the heap property. We need to rebuild the heap from the bottom up. This is called "*swim*".
- Swimming up the tree is simple:

```python
def swim(k):
    while k > 1 and greater(k // 2, k):
        swap(k // 2, k)
        k = k // 2
```

  - Note that `//` in `python` corresponds to *integer division*
  - The `swap` function exchanges the array values at the two given indices.
  - The `greater` function returns true if the value at the first index is larger than that at the second index.

PRIORITY QUEUE WITH HEAP

Element removal:



Sink steps for removal of `3`:

1.
```
k: 1 | 2 * k: 2 | 2 * k + 1: 3
array[k]: 20 | array[2*k]: 7 | array[2*k+1]: 5
```
2.
```
k: 3 | 2 * k: 6 | 2 * k + 1: 7
while condition fails: only one step for this sink!
```

- The highest priority element is at index `1` which is where we delete elements. *The removed element is replaced with the last element in the heap.*
- This will again destroy the heap property. We need to rebuild the heap from the top down. This is called "*sink*".
- Sinking down the tree is simple too:

```
1 def sink(k):
2     while 2 * k <= n:
3         j = 2 * k
4         if j < n and greater(j, j + 1):
5             j += 1
6         if not greater(k, j):
7             break
8         swap(k, j)
9         k = j
```

  - n is the number of elements in the heap

PRIORITY QUEUE WITH HEAP

Building the initial heap:

- Initially we need to build the heap assuming we start from an array input with values at random order.
- We can simply build the initial heap by *inserting* the new elements in a loop.
- What is the time complexity for the swim and sink operations?
- What is the best time complexity we can expect for this initial heap build?
- We are now going through a simple implementation for demonstration purpose. The `python` standard library provides an implementation of a priority queue for you.

PRIORITY QUEUE WITH HEAP

Heap exercise:

You are given the following input array:

$$a = [1, 8, 5, 9, 23, 2, 45, 6, 7, 99, -5]$$

1. Draw the heap ordered binary tree and write the binary heap (array).
2. Remove -5 and rebuild the heap with a corresponding swim or sink operation.

RECAP

- BST traversal
- Priority queues and heaps