SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 18

Fabian Wermelinger
Harvard University
CS107 / AC207

Tuesday, November 1st 2022

LAST TIME

- Introduction to data structures
- Linked lists
- Iterators
- Binary (search) trees

TODAY

Main topics: ***Binary search trees, tree traversal, priority queues and heaps***

Details:

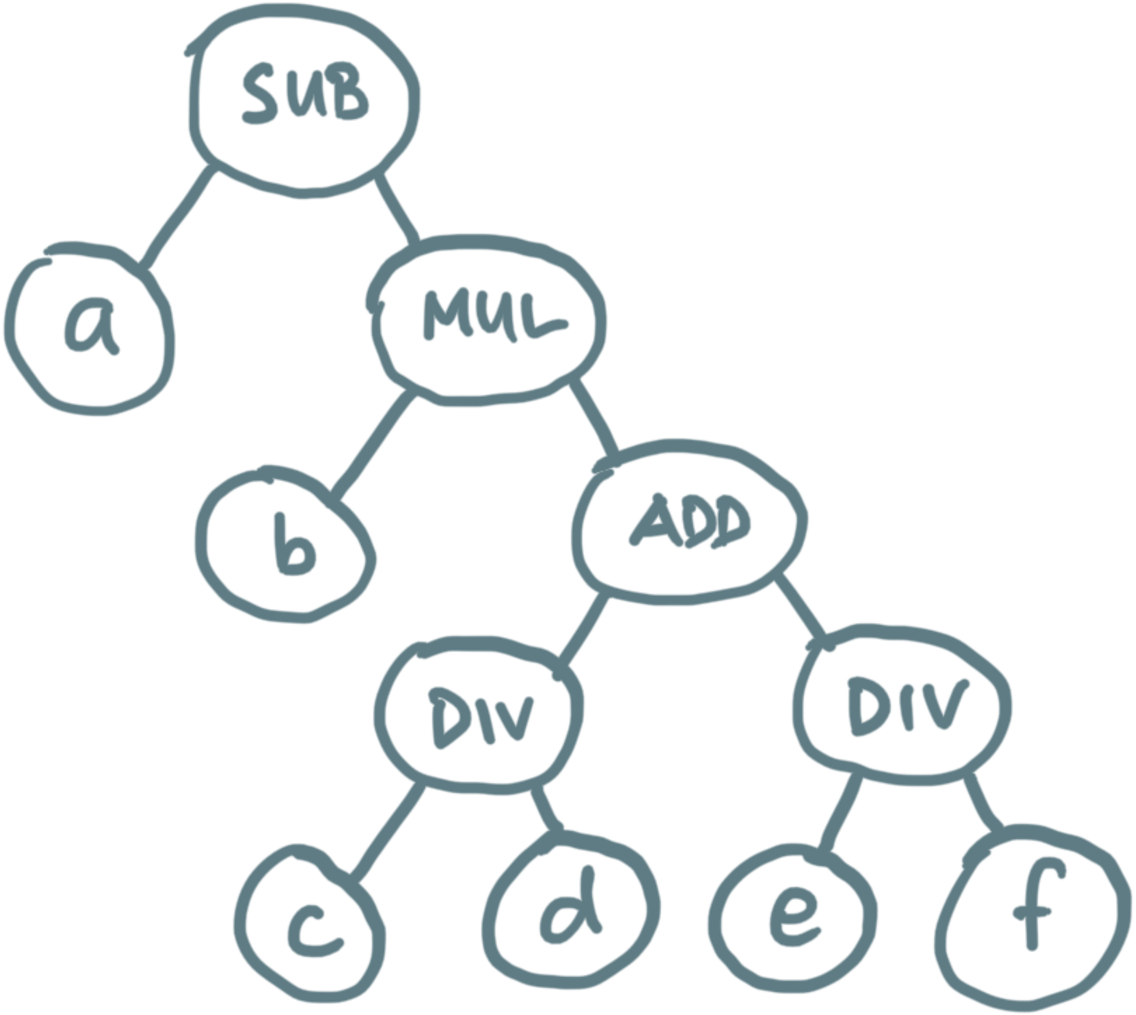
- Continuation with binary search trees
- Binary tree traversal
- Other common data structures
- Priority queues and heaps

BINARY TREES

- Binary trees are an important type of tree structure.
- Each node in a binary tree has *at most* ***two*** subtrees → the highest degree possible in such a tree is 2.
- If only one subtree is present, we *distinguish* whether it is a ***left*** or ***right*** subtree.
- A binary tree *is not* a special case of an ordinary tree → *a binary tree is a different concept but there are many relations between ordinary trees.*
- In class and in the homework we will focus on binary trees.

BINARY TREES

Example: parse an expression tree



Given the expression:

$$a - b\left(\frac{c}{d} + \frac{e}{f}\right)$$

the corresponding binary tree is given on the left. This connection between formulas and trees is very important in applications and naturally finds application in AD as well. The tree reflects the *precedence* of parentheses as well as multiplication or division operations before addition and subtraction.

BINARY TREES

Example: parse an expression tree

- The expression tree is parsed by exploiting operator precedence built into Python.
- It allows to build the tree automatically.
- ***Example code:***

```
>>> tree = a - b * (c / d + e / f)
>>> print(tree)
sub(a, mul(b, add(div(c, d), div(e, f))))
```

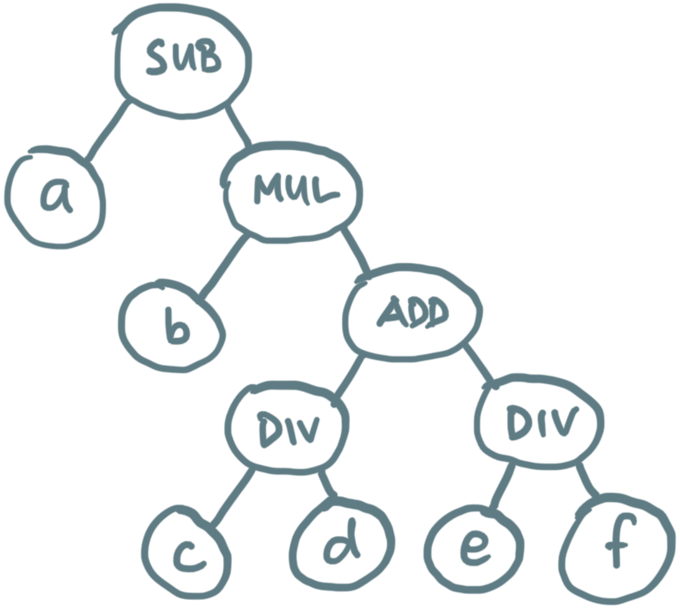
- The recursion is implied by operator precedence:

```
>>> tree = TreeNode.__sub__(a,
            TreeNode.__mul__(b,
                TreeNode.__add__(
                    TreeNode.__truediv__(c, d),
                    TreeNode.__truediv__(e, f)
                )
            )
        )
```

Expression:

$$a - b\left(\frac{c}{d} + \frac{e}{f}\right)$$

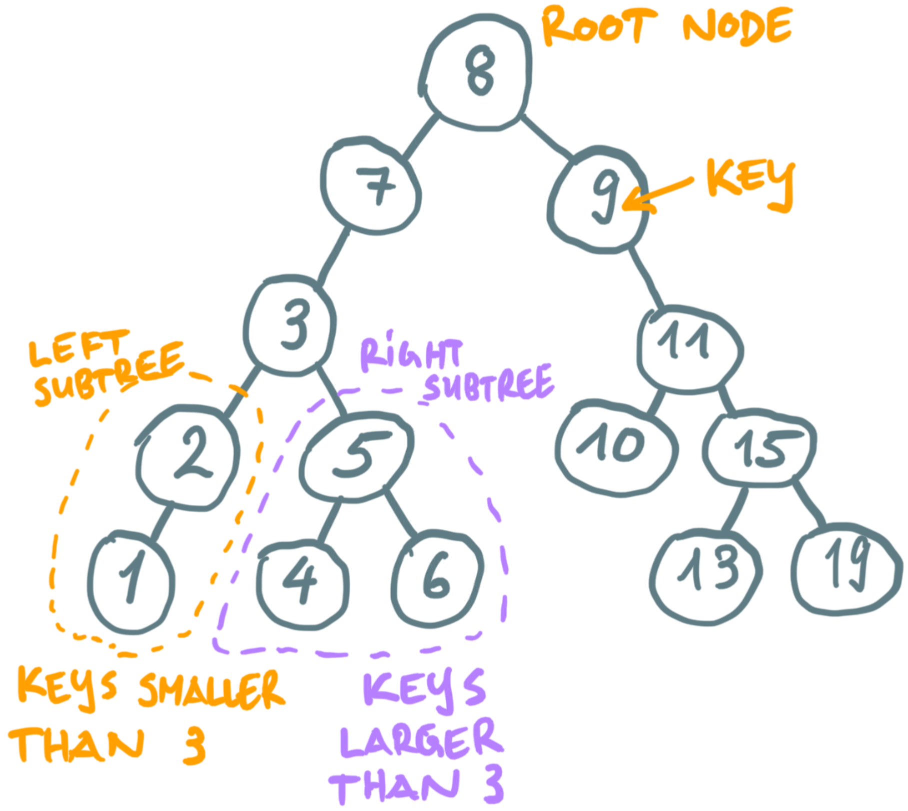
Expression tree:



BINARY SEARCH TREE

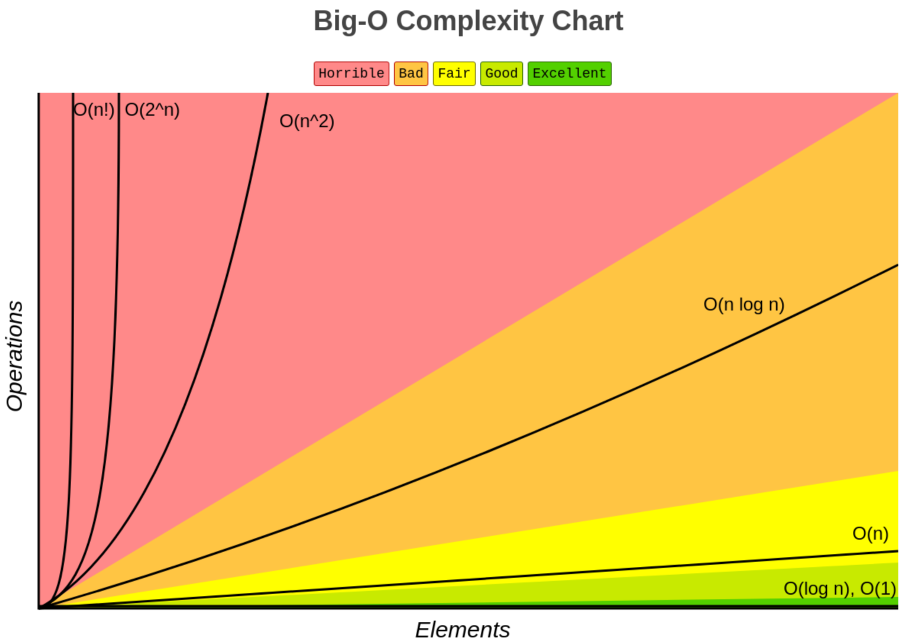
A binary search tree (BST) is an ***ordered*** binary tree with key values *comparable* with each other. A BST has the property that any key in the nodes contained in the ***left*** subtree of the root node v are strictly ***smaller*** than the key in v and any key in nodes contained in the ***right*** subtree are strictly ***larger*** than the key in v.

- A BST is one of the most fundamental algorithms in computer science.
- If the root node of a BST does not have a *left* subtree, it means that the key of the root node is the *smallest value* (similarly for the *largest value*).
- We are only concerned with a *single* occurrence of key values.



BINARY SEARCH TREE

Time complexity of a binary search tree:



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

https://www.bigocheatsheet.com/

If the BST is *balanced*, the time complexity for a search is $\mathcal{O}(\log_2 n)$

BINARY SEARCH TREE

Searching a BST:

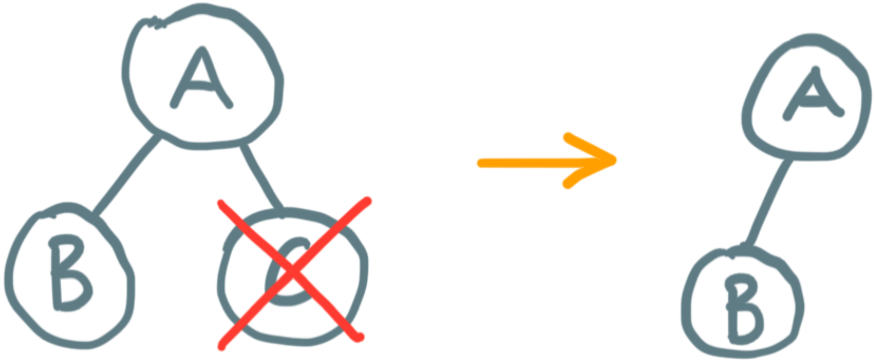
- Searching a BST is a ***recursive*** algorithm → you search for a *key* match.
- If there is a search *hit*, return the associated node value.
- If there is a search *miss*, return `NULL` (e.g. `None` in Python or `nullptr` in C++).
- ***Algorithm:*** start at the root node and compare the search value with the `key` of the node.
  1. If the search value is *less* than the `key` of the node, recursively search the left subtree.
  2. If the search value is *greater* than the `key` of the node, recursively search the right subtree.
  3. If the search value is equal to the node `key` return the corresponding value.
  4. If you reached a terminal node without a hit you can return `NULL` or handle an exception.
- → *node insertion is almost identical to a tree search.*

BINARY SEARCH TREE

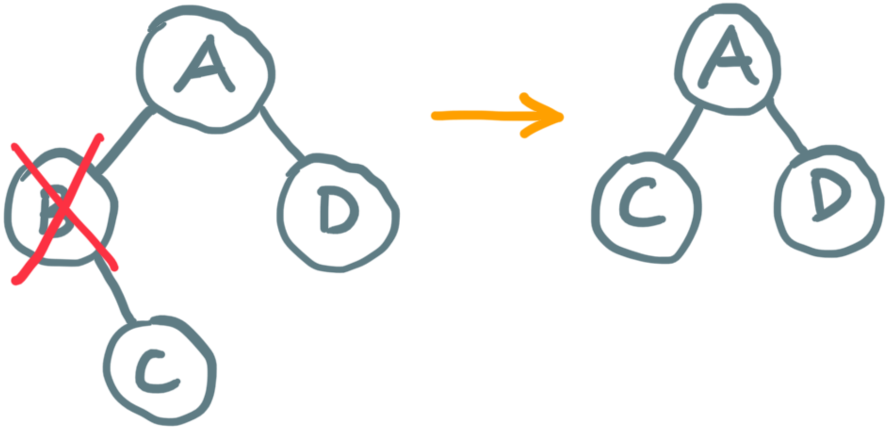
Node deletion for degrees 0 and 1:

- If the node to be deleted has *no children* it can just be removed.
- If the node to be deleted has *only one child*, replace the node to be deleted with its child, then delete the node that is no longer needed.
- ***How do you delete the smallest key in the tree? What about the largest key?***

Removal of terminal node:



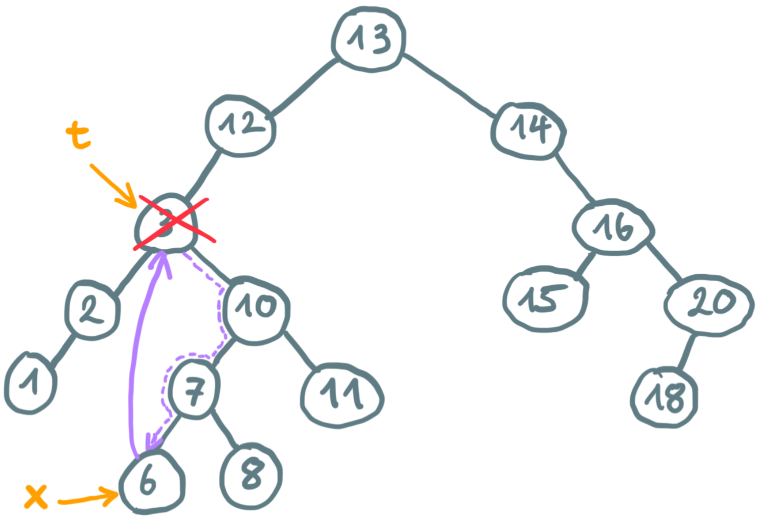
Node removal with single child:

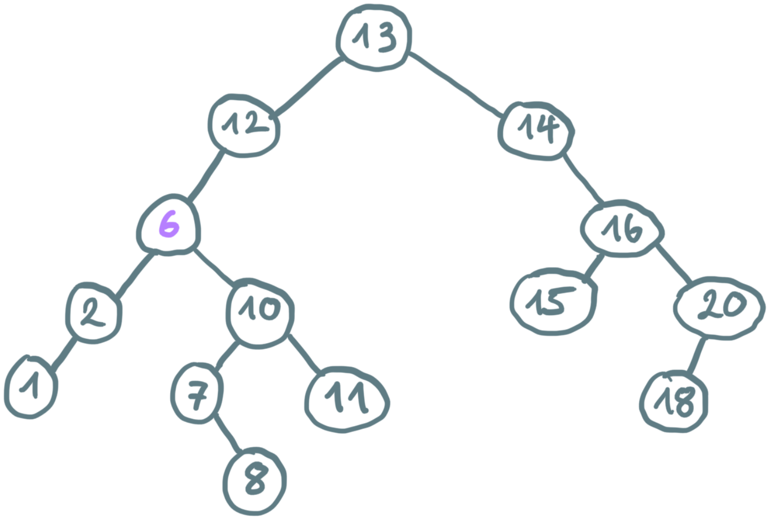


BINARY SEARCH TREE

Node deletion for degree 2: *(example deletes node 3)*

1. Keep a reference (or pointer) of the node to be deleted in `t`
2. Set `x` to point to the successor node `min(t.right)` (`x` points to node 6)
3. Update `t.key` with `x.key` (and possibly other node data).
4. If `x` has a *right subtree* it becomes the left subtree in the parent of `x`.
5. Delete node `x`





What happens when we delete node 6?

BINARY TREE TRAVERSAL

Distinction between ordinary trees and binary trees:

1. *An **ordinary tree** cannot 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 in applications and it is very likely that you will meet them in one form or another.
- It is therefore important to have a good understanding of this data structure.
- The difficulty lies mostly in the ***recursive*** nature of trees.

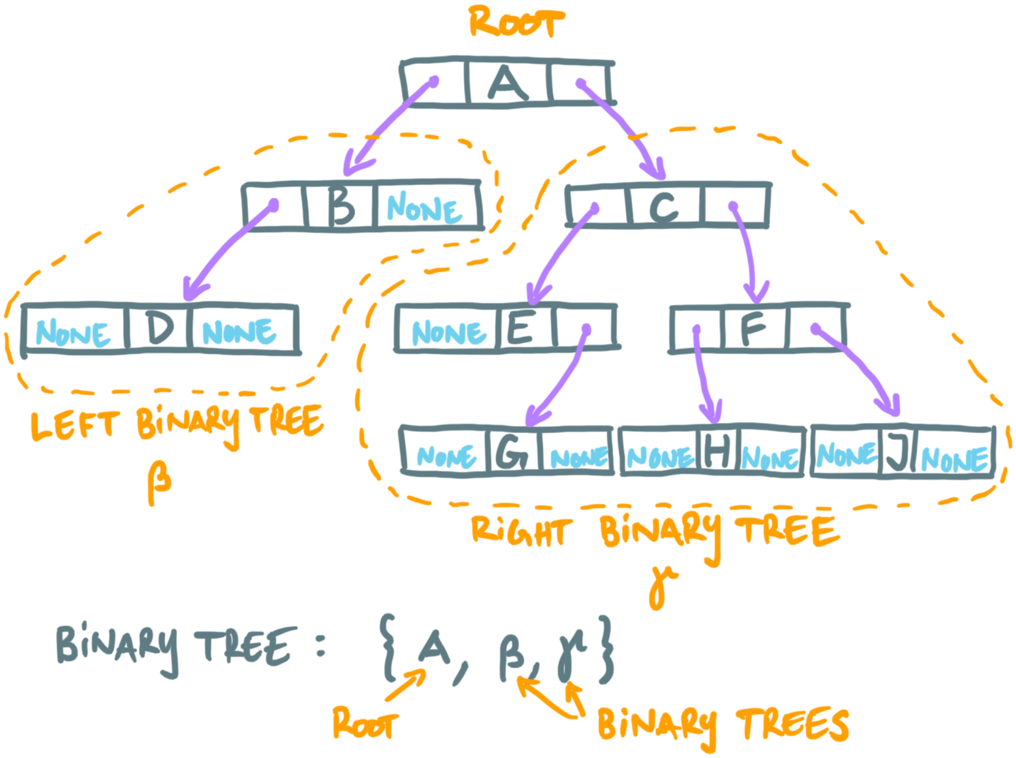
BINARY TREE TRAVERSAL

Another way to look at a binary tree using set notation:

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

→ ***Note:*** this definition is *recursive*!

Example binary tree:

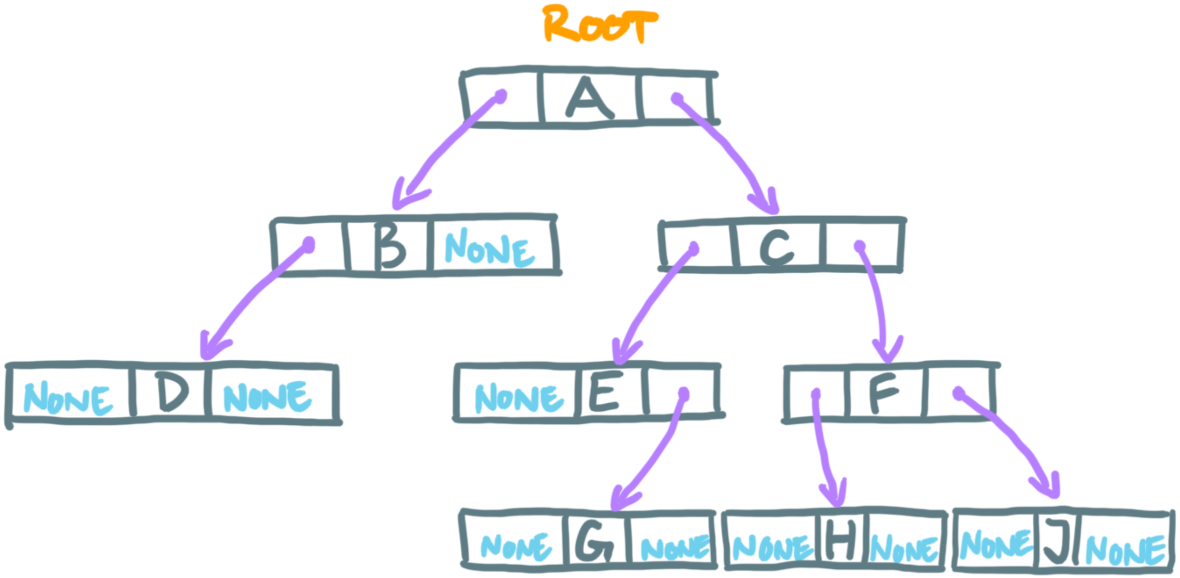


The set $\{A, \beta, \gamma\}$ defines a binary tree. How does the set look like for the binary tree with root B?

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:

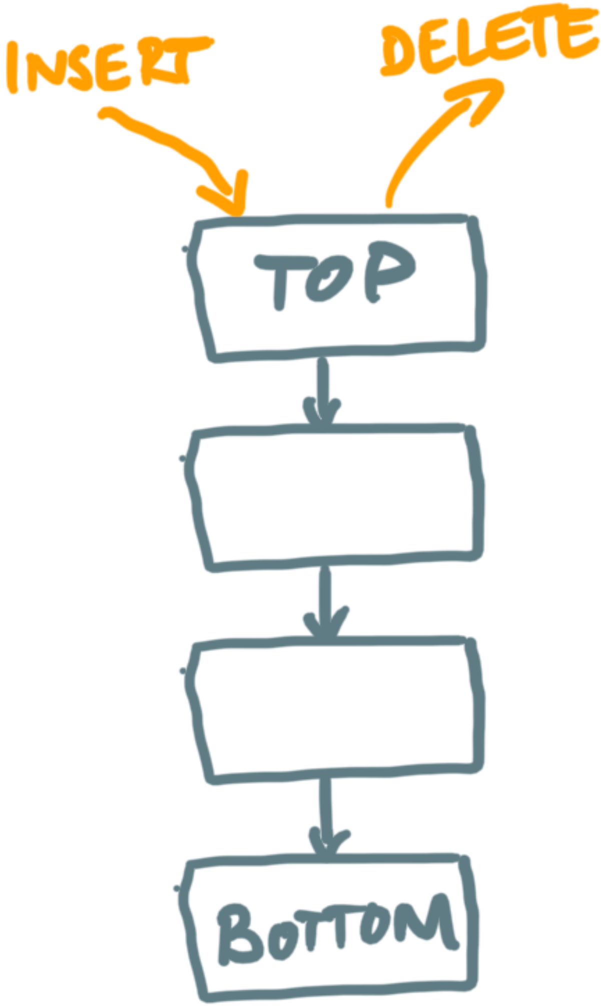
BINARY TREE TRAVERSAL

- In each of the three principal ways of tree traversal, ***we have visited each node exactly once***.
- In the previous exercise we have just printed the node ID when we visited it to visualize the traversal order.
- In more useful applications, a tree node may hold a reference to other data that we can operate on *when we visit the node*.
  Example: → 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 appear in many places in Computer Science.
- Other linear data structures are stacks, queues and deques.
- These data structures follow similar ideas we have seen so far:
  - A structure of nodes linked together.
  - We ask the question what is the time complexity for operations on the data structure such as node insertion, deletion, search, and so on.

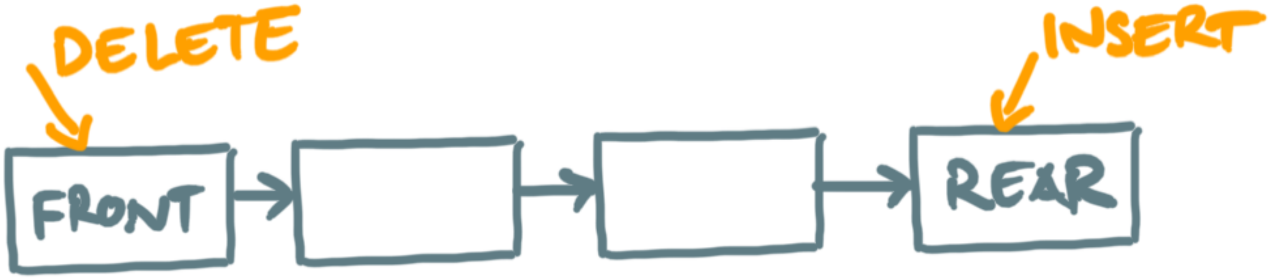
STACK



https://en.cppreference.com/w/cpp/container/stack

- A ***stack*** is a linear list for which all insertions, deletions and usually all accesses are made at one end of the list only.
- 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 Python tutor. ***Question:*** is the stack a useful data structure for *recursive* function calls?

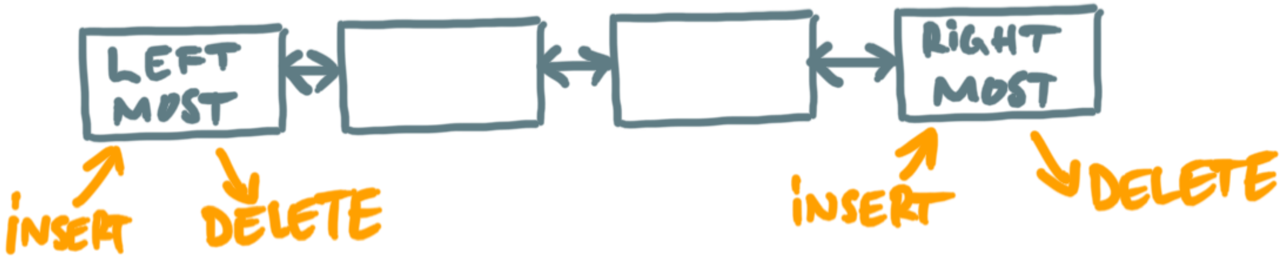
QUEUE



https://en.cppreference.com/w/cpp/container/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



https://en.cppreference.com/w/cpp/container/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".

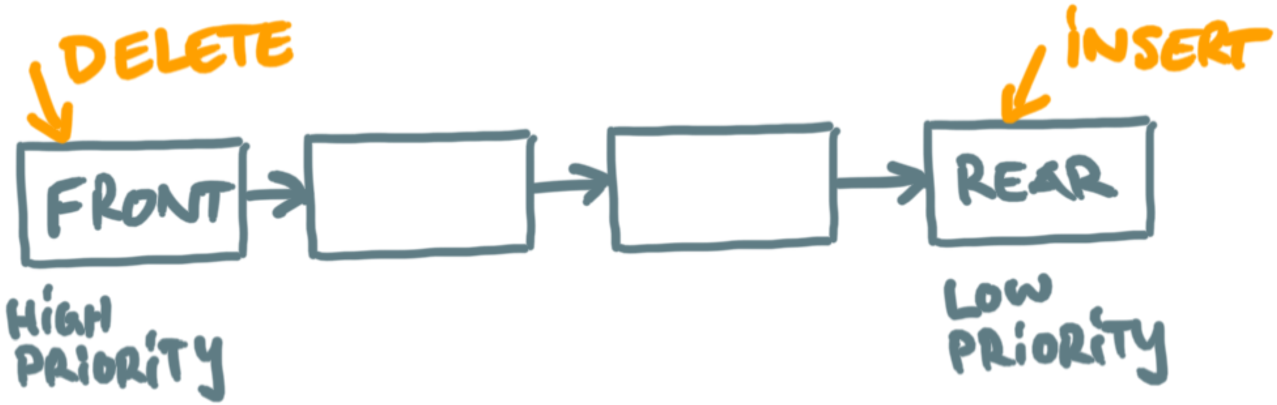
PRIORITY QUEUE

- 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` (and the ***largest*** `key` in the case of "*largest-in-first-out*").
- A list that assigns certain elements ***priority*** is called a ***priority queue***.
- This implies an *order* among list elements 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 based on a *"least recently used"* policy, priority queues are the correct data structure to use.
- Maintain a priority order among your customers.

PRIORITY QUEUE IMPLEMENTATION



https://en.cppreference.com/w/cpp/container/priority_queue

- We need a method for *element insertion that maintains the priority order*.
- We need a method to *remove the element with highest priority* (or *lowest* priority, depending on use case).
- We need to be able to *obtain the element with highest priority* (same as removal without removing the element).
- → ***ideas how to go about this with data structures you have seen so far?***

PRIORITY QUEUE IMPLEMENTATION

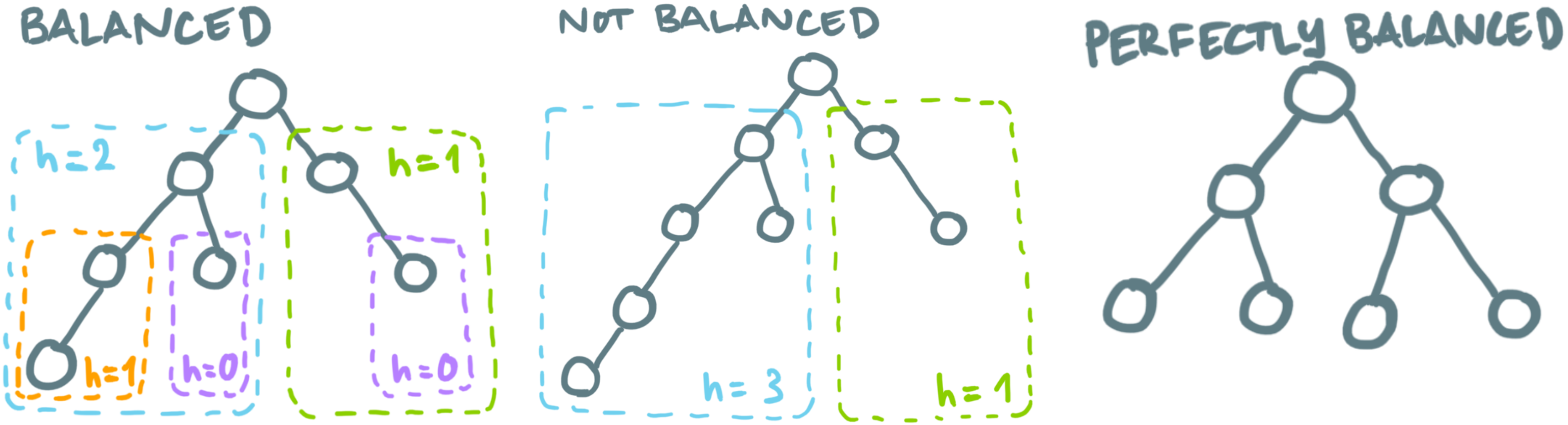
Thoughts on implementation:

1. ***You could use a sorted list:*** insertion of new elements is $\mathcal{O}(n)$, removal and access are $\mathcal{O}(1)$.
2. ***You could keep a reference to the element with highest priority:*** 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

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

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

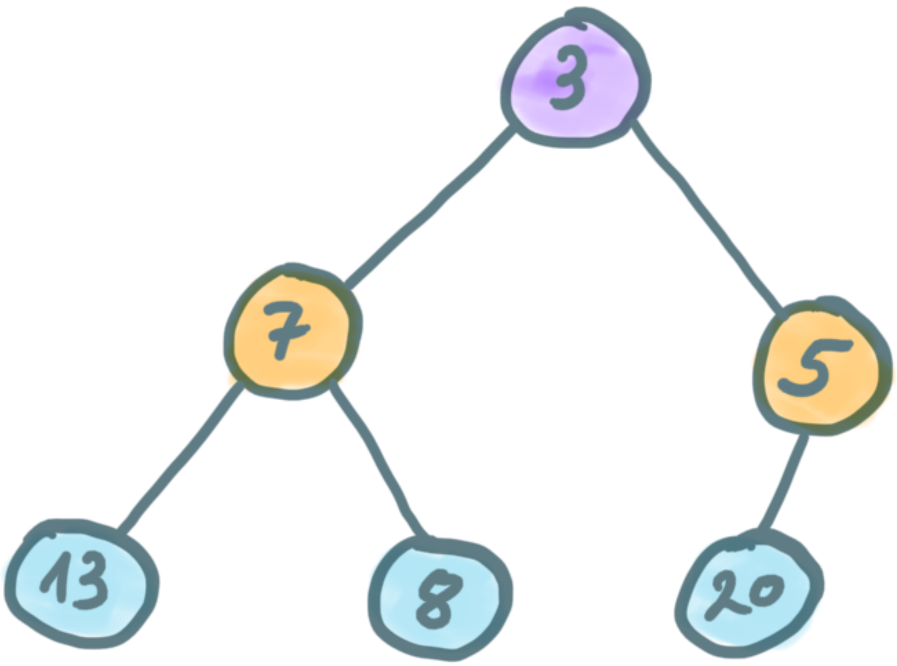
for all $i = 1, \ldots, n/2$. The ***least element*** is $h_1 = \min(h_1, h_2, \ldots, h_n)$.

- We can just as well define the heap with the "$\geq$" operator instead. The ***greatest element*** is then given by $h_1 = \max(h_1, h_2, \ldots, h_n)$.
- We refer to a ***min-heap*** for the former ($\leq$) and ***max-heap*** for the latter ($\geq$) definition, respectively.

→ ***a heap can therefore be cast into a binary tree***, where the key h_i of a tree node satisfies the ***heap properties*** (1) and (2), relative to its two children h_{2i} and h_{2i+1}.

HEAP

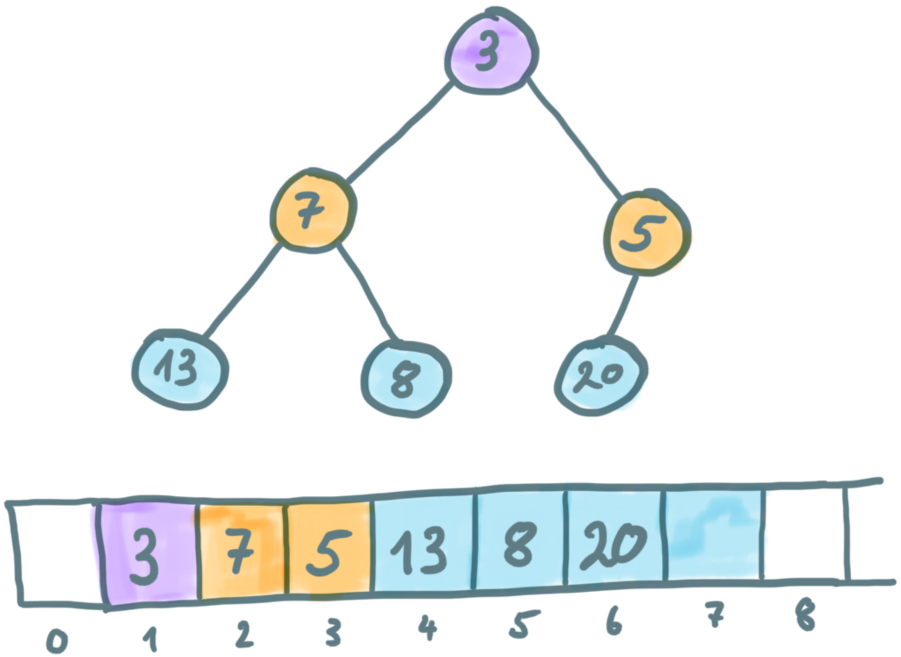
Heap ordered tree:



- 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 (*key 3 in the example on the left*).
- If you change $\leq$ to $\geq$ in the heap property, the key in the root node corresponds to the greatest element (*min-heap* or *max-heap*).

HEAP

Binary heap (or just "heap"):



Heap property:

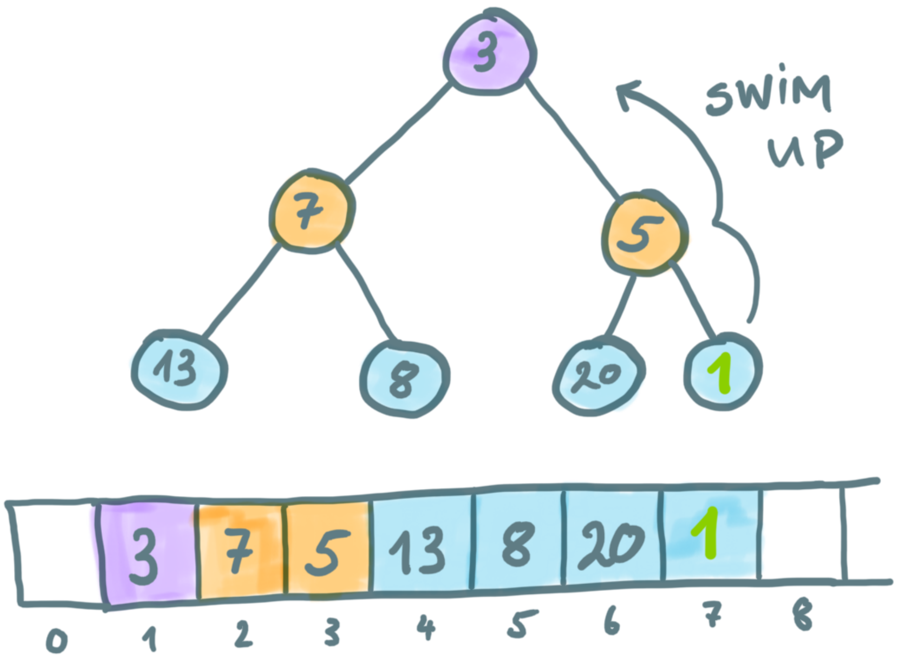
$$h_i \leq h_{2i}$$

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

- 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:



- A new element is inserted at the index $n + 1$.
- Insertion will *destroy the heap property*. We need to rebuild the heap from the bottom up → *this is called "**sift-up**".*
- ***Sift-up up the tree is simple:***

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

Notes:

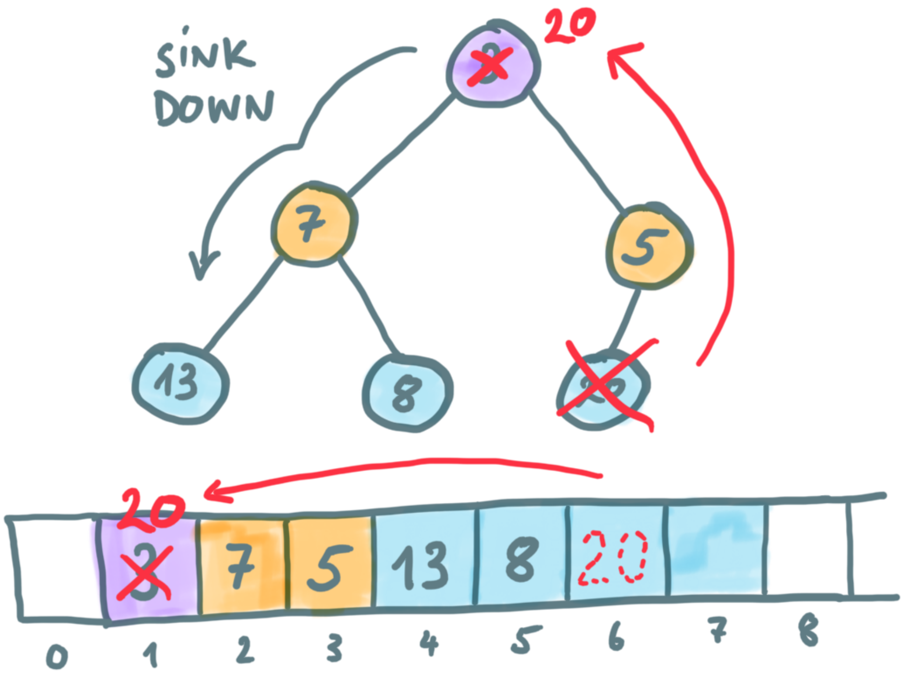
- In Python "`//`" corresponds to *integer division*.
- The `swap` function exchanges the array values at the two given indices.
- The `greater` function returns true if the array value at the first index is larger than that at the second index.

Example: sift-up steps for insertion of value 1 at index `k = n + 1 = 7`:

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`

PRIORITY QUEUE WITH HEAP

Element removal:



- 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.*
- Removal will *destroy the heap property*. We need to rebuild the heap from the top down → this is called "***sift-down***".
- ***Sift-down down the tree is simple too:***

```python
def siftdown(k):
    # n = 5 elements in the heap
    while 2 * k <= n:
        j = 2 * k
        if j < n and greater(j, j + 1):
            j += 1 # pick smaller child
        if not greater(k, j):
            break
        swap(k, j)
        k = j
```

Example: sift-down steps for removal of 3:

1.
```
k: 1 | j = 2 * k: 2 | j + 1 = 2 * k + 1: 3
array[k]: 20 | array[j]: 7 | array[j + 1]: 5
```

2.
```
k: 3 | j = 2 * k: 6 | j + 1 = 2 * k + 1: 7
while condition fails: 6 <= 5 is False
```

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 could simply build the initial heap by *inserting* the new elements in a loop.
- ***What is the time complexity for the sift-up and sift-down operations?***
- ***What is the best time complexity we can expect for this initial heap build?***
- The Python standard library provides an implementation of a heap queue https://docs.python.org/3/library/heapq.html

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 sift-up or sift-down operations.

RECAP

- Continuation with binary search trees
- Binary tree traversal
- Other common data structures
- Priority queues and heaps

Further reading:

- ***Heap and heapsort:*** Section 2.2.5 in N. Wirth, "*Algorithms + Data Structures = Programs*", Prentice-Hall, 1976.
- ***Heap and heapsort:*** Section 5.2.3 in D. Knuth, "*The Art of Computer Programming*", Volume 3, 2nd Edition, Addison-Wesley Professional, 1998.