# MATHEMATICAL MODELLING FOR COMPUTATIONAL SCIENCE

## LECTURE 9

AM215

Harvard University

# OUTLINE

- Data structures
- Linked lists
- Trees
- Binary trees
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

# OUTLINE

- **Data structures**
- Linked lists
- Trees
- Binary trees
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

# INTRODUCTION TO DATA STRUCTURES

*Programs = Algorithms + Data Structures*

***Examples:***

Algorithms:

Newton's method, least squares regression

Data structures:

- Python built-in data structures: `list`, `tuple`, `dict`
- Singly or doubly linked lists
- Stacks or queues
- Trees
- Heaps (priority queues)

# INTRODUCTION TO DATA STRUCTURES

## *What is an Abstract Data Type*

Defines *behavior* from the point of view of a user. A `class` in Python implements an abstract data type.

The *implementation is hidden* from the user and utilizes some *data structures* to realize the expected behavior (*abstraction*).

The *data structures* used can be as simple as a built-in integer or float or more complex data structures like lists, dictionaries or trees.

***Examples:***

- A directed acyclic graph is an abstract data type.
- Complex numbers are also abstract data types. They have two scalar `float` data structures, a real part and an imaginary part.

# INTRODUCTION TO DATA STRUCTURES

***Data structures may have different memory layouts:***

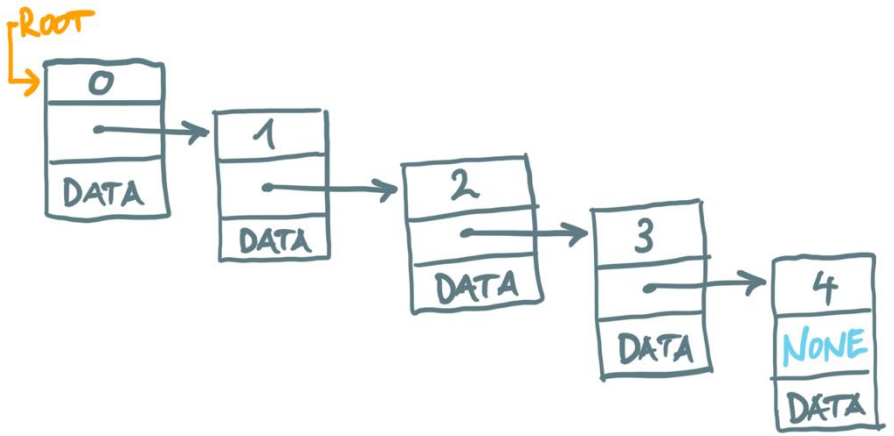
Performance critical applications should have the *data close by in memory* (coalesced memory layout).
An *array* is an example of a coalesced data structure. All elements are next to each other and we can access individual elements in `O(1)` complexity *(computing the address of an array element is constant)*.

Other data structures, like a *linked list* for example, allocate memory for new elements dynamically and are *chained together using pointers/references*.
Individual elements may not be close in memory and the *address computation of an element depends on previous elements*. This usually means the complexity of element access is higher (e.g. `O(`*n*`)` for a linked list with *n* nodes) but insertion or deletion of elements may be cheaper.

# OUTLINE

- Data structures
- **Linked lists**
- Trees
- Binary trees
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

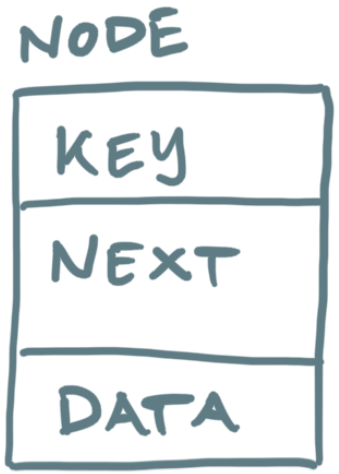
# LINKED LIST




- A ***linked list*** *(or linear list)* is one of the simplest data structures.
- It consists of a series of ***nodes*** which have a *reference to the next node* in the sequence.
- A node is identified by a key or ID and may additionally be associated with data.
- List traversal is simply achieved by following the references to the next node, given that we have a reference to the ***root*** node (the first node in the list).

# LINKED LISTS: NODES

A node in a linked list is the foundation of the structure. A simple layout of a node may look like this:

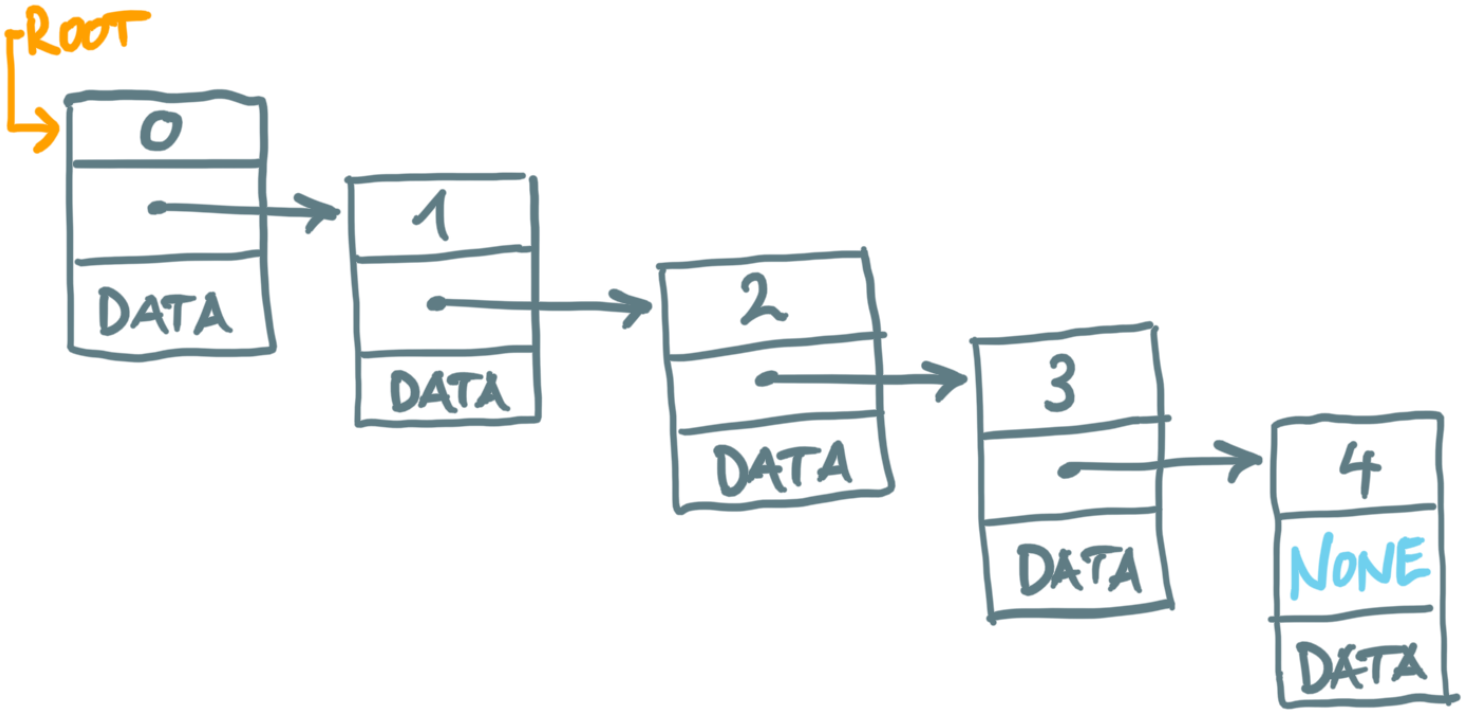



```
class Node:
    """Node for a linked list."""
    def __init__(self, key, *,
                 next=None,
                 data=None):
        self.key = key
        self.next = next # node
        self.data = data
```

A node is identified by a `key`. This can be an integer or string for example. *It must uniquely identify the node*.
The `next` attribute points to the next node in the sequence.
The `data` attribute is optional and can be used for attaching data to the node.

# LINKED LISTS

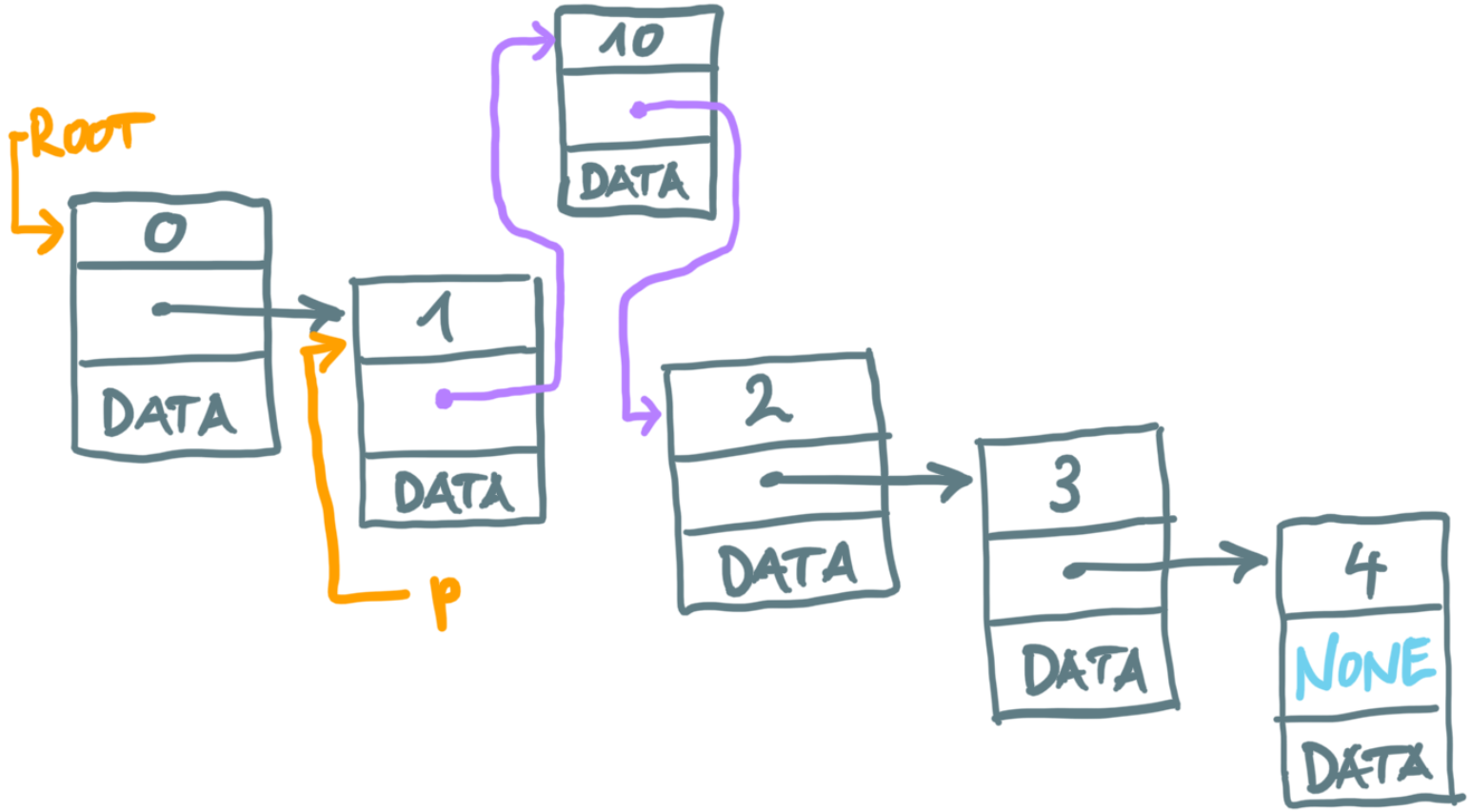
*Simple example of a linked list*




The first node (in this example with key `0`) is the ***root*** node of the list.
The `next` attribute of the last node has a `NULL` value (e.g. `None` in Python).

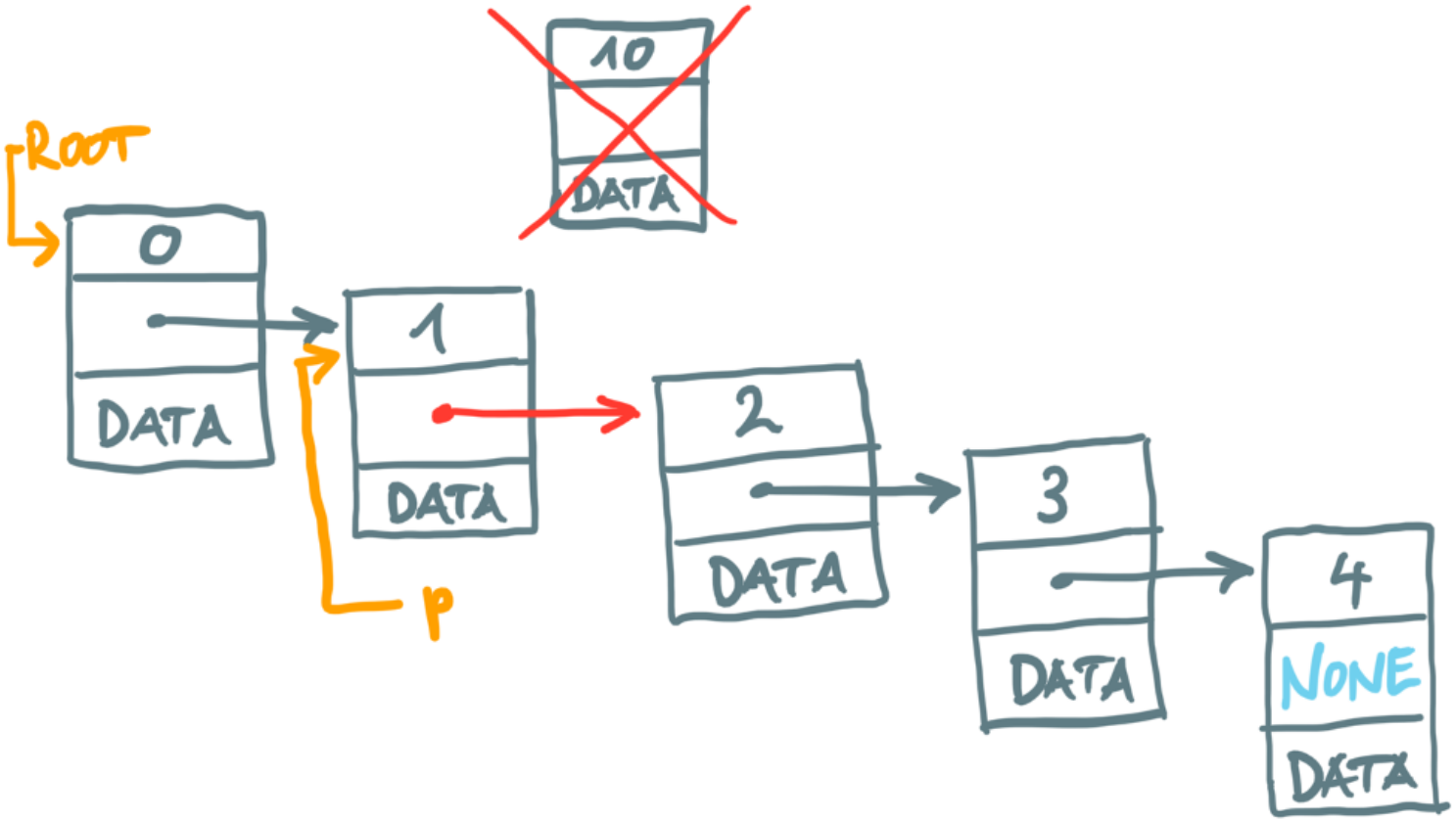
List traversal is of complexity $O(n)$ if there are $n$ nodes.

# LINKED LISTS: NODE INSERTION




We can insert a new node with $O(1)$ complexity at a reference node p (If p is known → you can get this reference by $O(n)$ search otherwise).
Insertion after p is trivial → *simply relink the* next *attributes*.

# LINKED LISTS: NODE INSERTION




Removing the *successor* of a node with reference `p` is trivial and is of `O(1)` complexity (if `p` is known).
It is not straightforward to remove the node where reference `p` points to because in a singly linked list we have no knowledge about the *predecessor*.

# LINKED LISTS: EXAMPLE IMPLEMENTATION

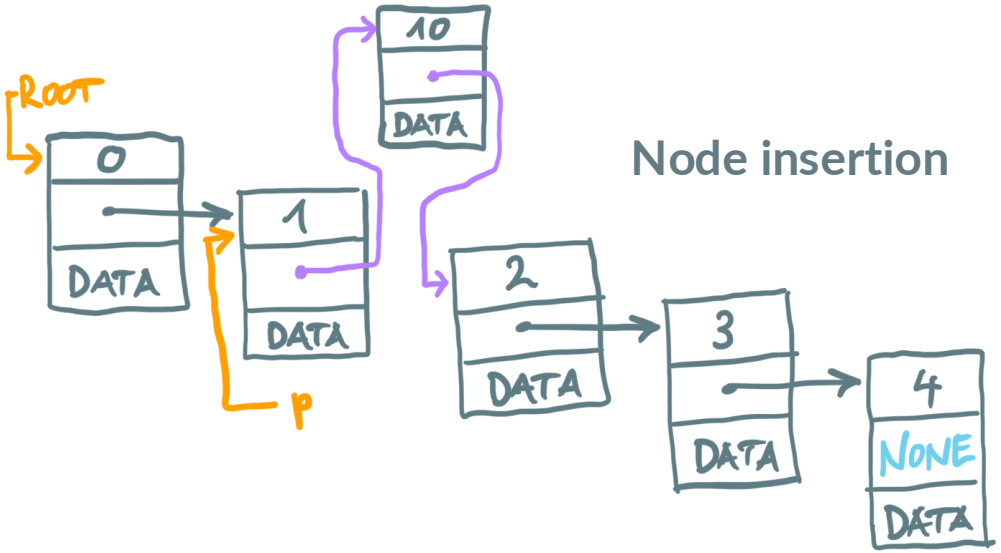
***Example linked list implementation that addresses the following:***

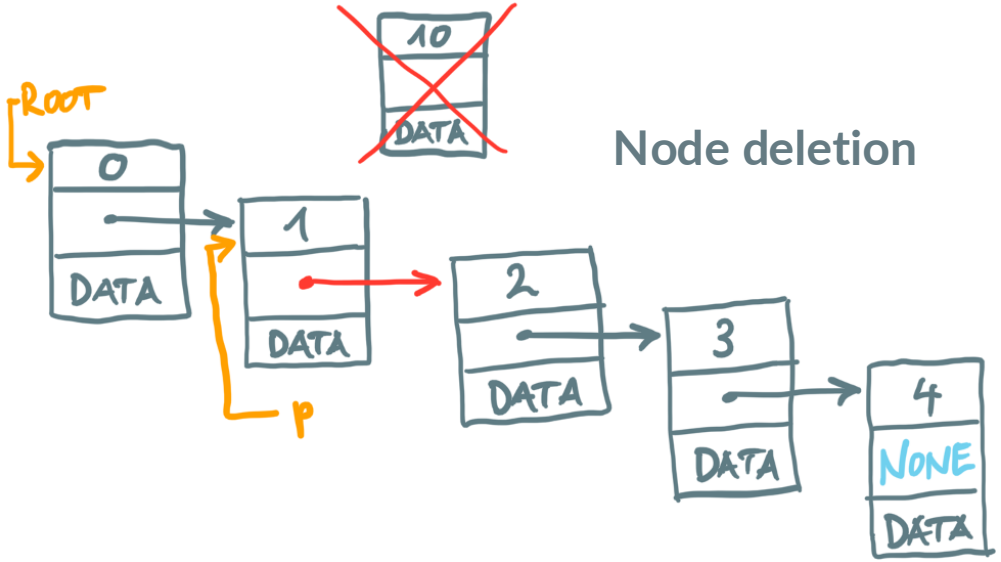
Create a linked list with `key/data` pairs. Optionally we want to be able to create the list in reverse order.

A linked list is a sequence, it should support the sequence protocol. (`__len__, __getitem__`)

Insert a new node before or after the list node identified with `key`.

Remove a list node identified by `key`.



# OUTLINE

- Data structures
- Linked lists
- **Trees**
- Binary trees
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

# TREES

Trees are the most important *nonlinear* structures that arise in computer algorithms.

***Linear lists:***

- Stacks, queues
- Double-ended queues (deques)
- Singly and doubly linked lists
- Circular lists
- Arrays

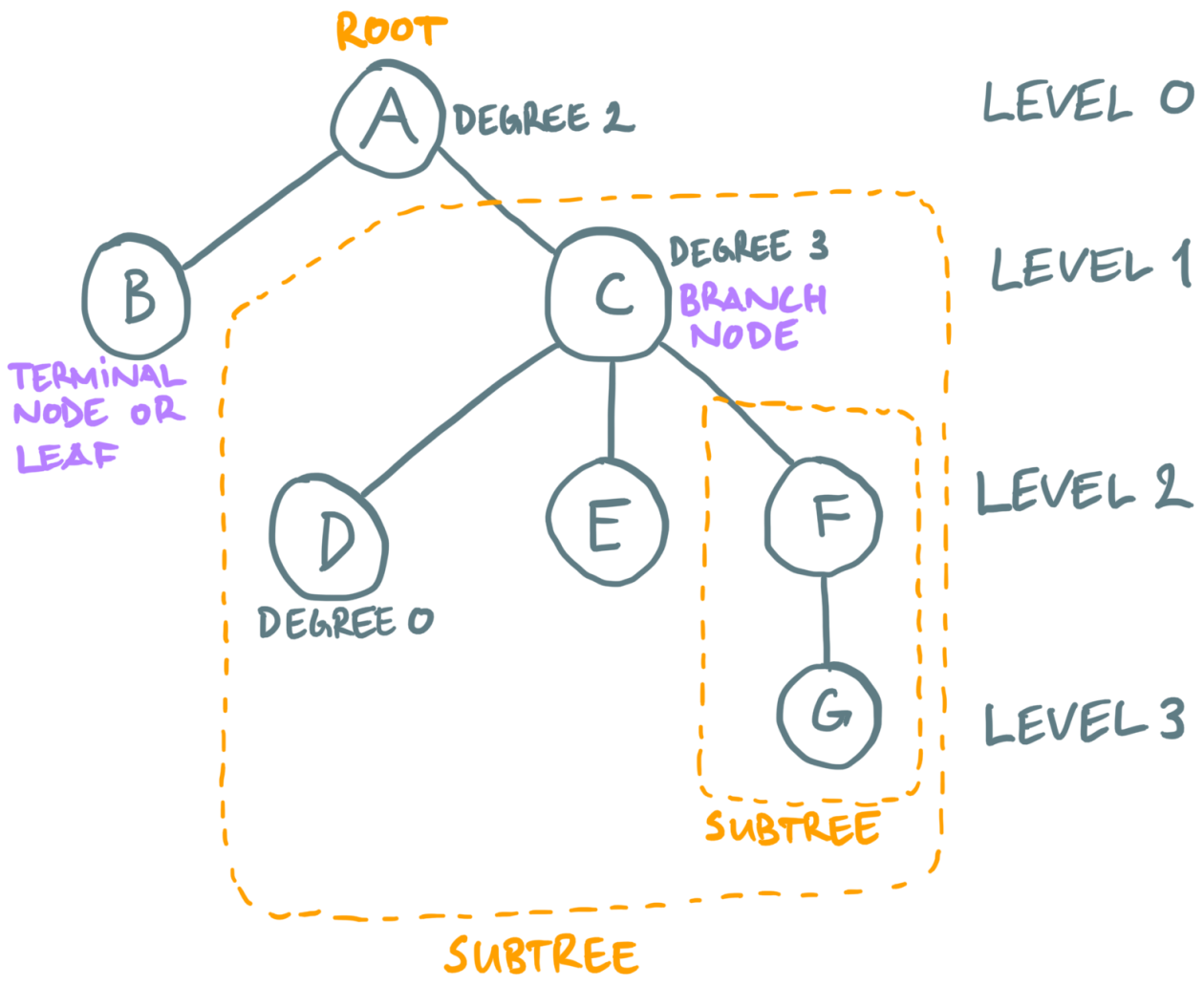
***Trees:***

- Ordinary trees
- Binary trees
- *Ordered* and oriented trees
- Forests and subtrees

All tree structures have a ***recursive*** nature in common. Just like in real trees its branches are again little trees which have little branches and so on.

# ORDINARY TREES

- The node at level 0 is called *root node* (node `A`).
- Every (sub)tree has a root node.
- Node `C` is called a *parent* node and nodes `D`, `E` and `F` are called *children* of `C`.
- The ***degree*** of a node tells you how many subtrees the node has.
- A node with degree 0 is called *terminal node* or *leaf*.
- If the tree is ***ordered***, the subtrees have a *relative order* to each other.



# OUTLINE

- Data structures
- Linked lists
- Trees
- **Binary trees**
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- 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.*

# BINARY TREES

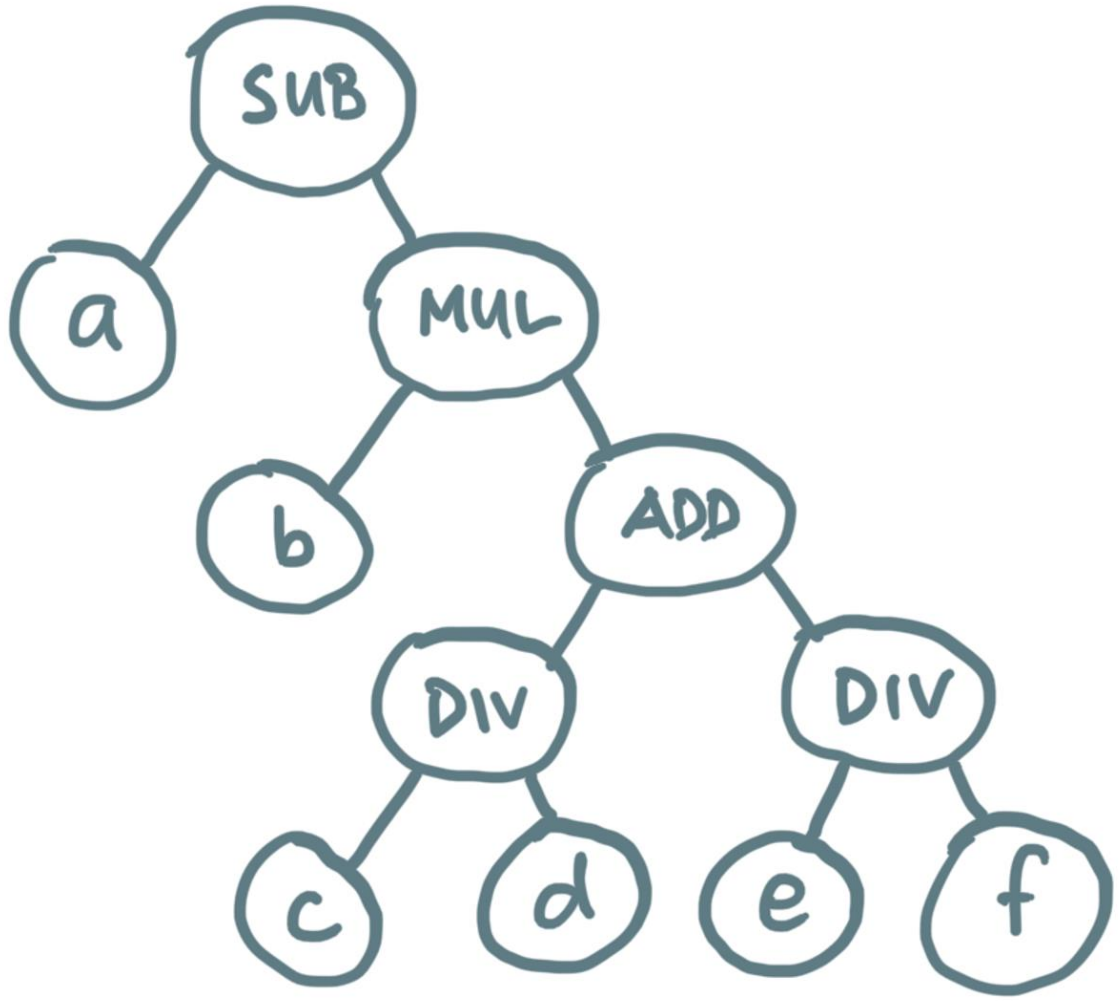
Given the expression:

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

the corresponding binary tree is given on the left.

This connection between formulas and trees is very important in applications.

The tree reflects the *precedence* of parentheses as well as multiplication or division operations before addition and subtraction.



# BINARY TREES: PARSING

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))))
```

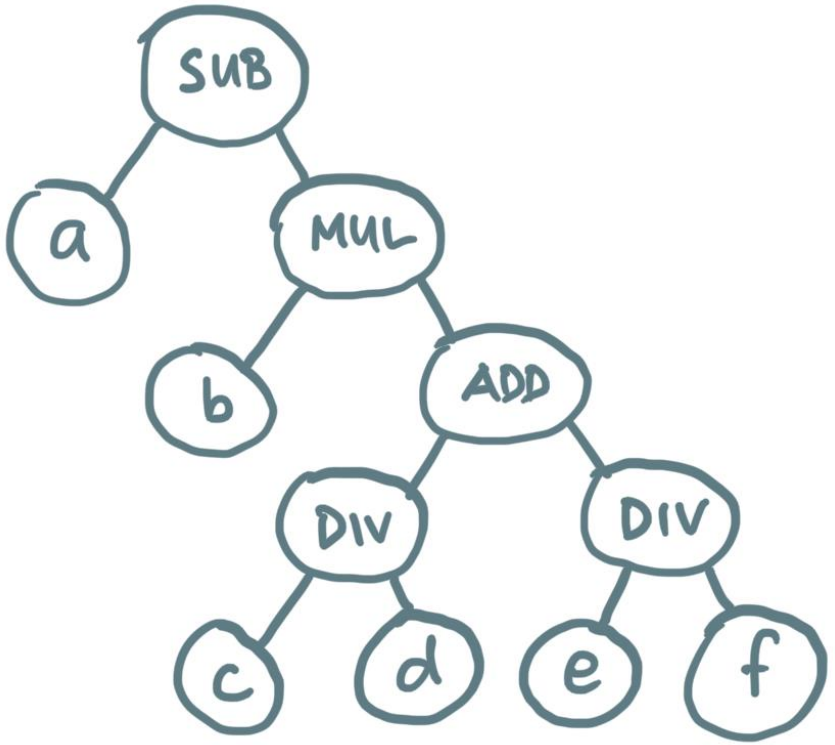
The recursion is implied by operator precedence:

```
>>> 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:**



# OUTLINE

- Data structures
- Linked lists
- Trees
- Binary trees
- **Binary search trees**
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

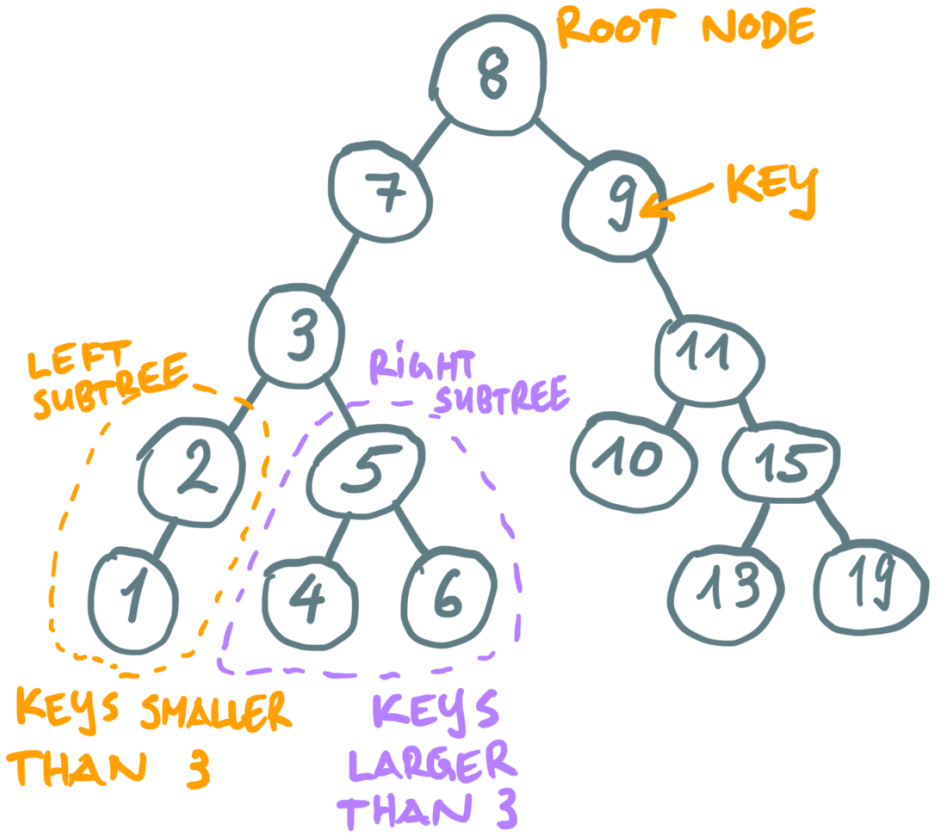
# BINARY SEARCH TREES

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 in the subtree* (similarly, no right child means it's the *largest value*).

We are only concerned with a *single* occurrence of key values.



# BINARY SEARCH TREES

## *Searching a BST:*

Searching a BST is a ***recursive*** algorithm.

If there is a search *hit*, return the associated node value.

If there is a search *miss*, return `NULL` (e.g. `None` in Python)

***Algorithm:*** Start at the root node and compare the search value with the `key` of the node.

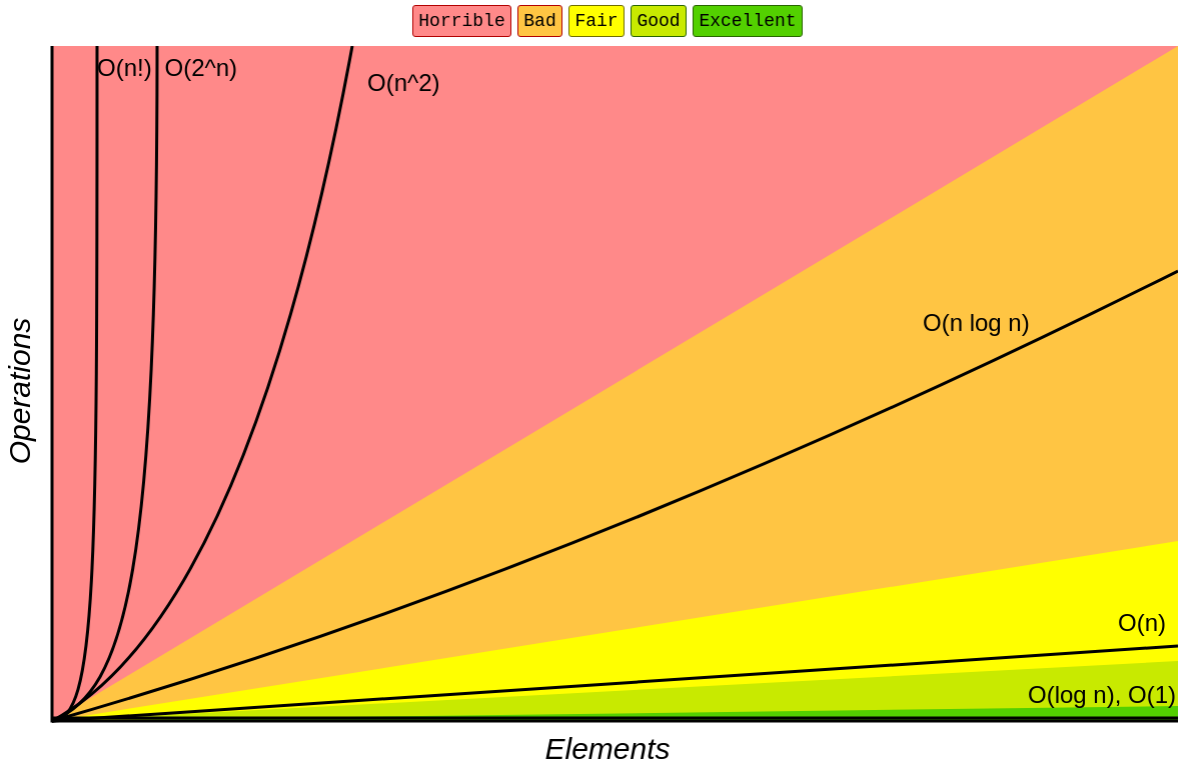
1. If the search value is *less* than the `key` of the node, recursively search the left subtree.
2. If the search value is *greater* than the `key` of the node, recursively search the right subtree.
3. If the search value is equal to the node `key` return the corresponding value.
4. If you reached a terminal node without a hit, you can return `NULL` or handle an exception.

Node insertion is almost identical to a tree search.

# BINARY SEARCH TREES

## *Time complexity of a binary search tree:*

### Big-O Complexity Chart




### Common Data Structure Operations

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

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

# BINARY SEARCH TREES

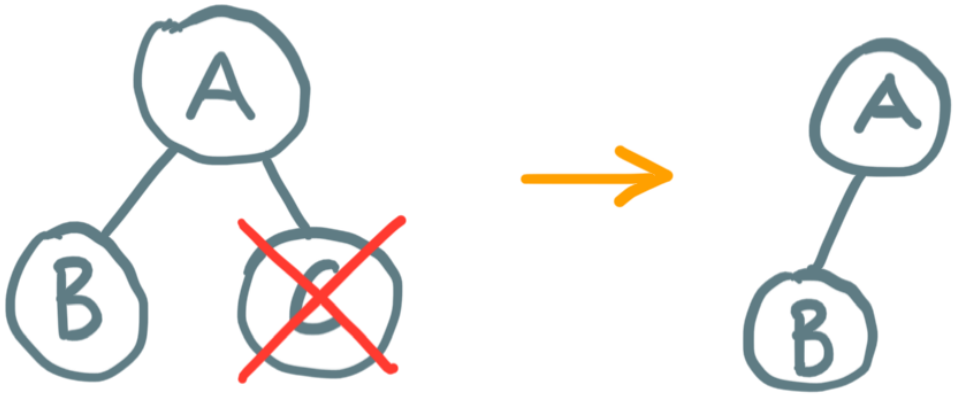
***Node deletion for degrees 0 and 1:***

If the node to be deleted has *no children*, it can just be removed.

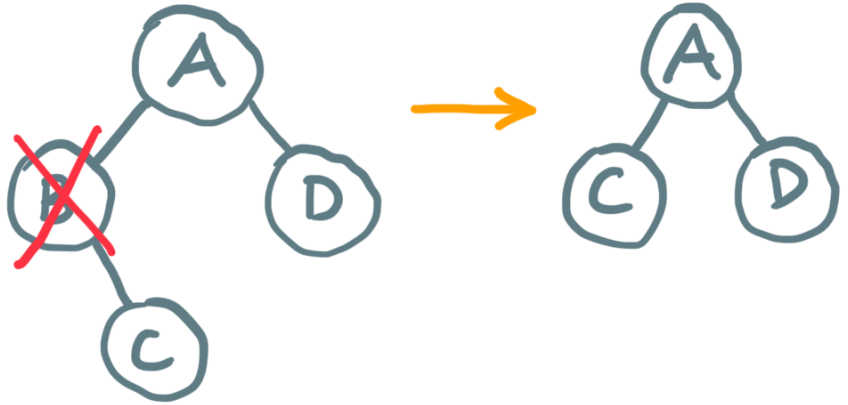
If the node to be deleted has *only one child*, replace the node to be deleted with its child, then delete the node that is no longer needed.

***How do you delete the smallest key in the tree? What about the largest key?***

***Removal of terminal node:***



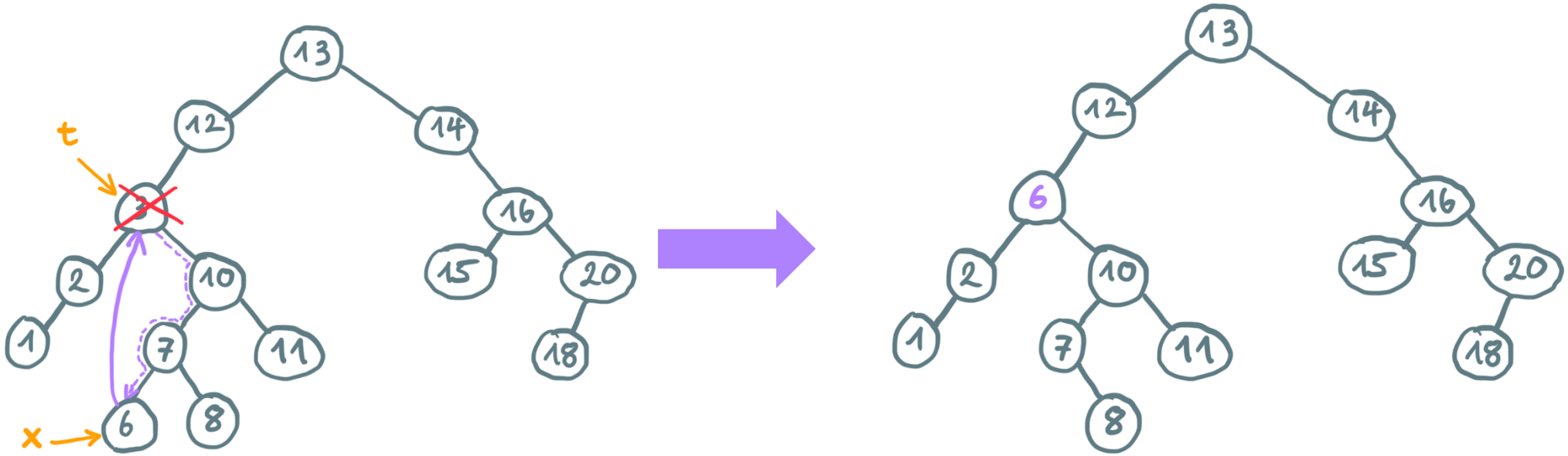

***Node removal with single child:***



# BINARY SEARCH TREES

***Node deletion for degrees 2 and 3:*** *(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`.



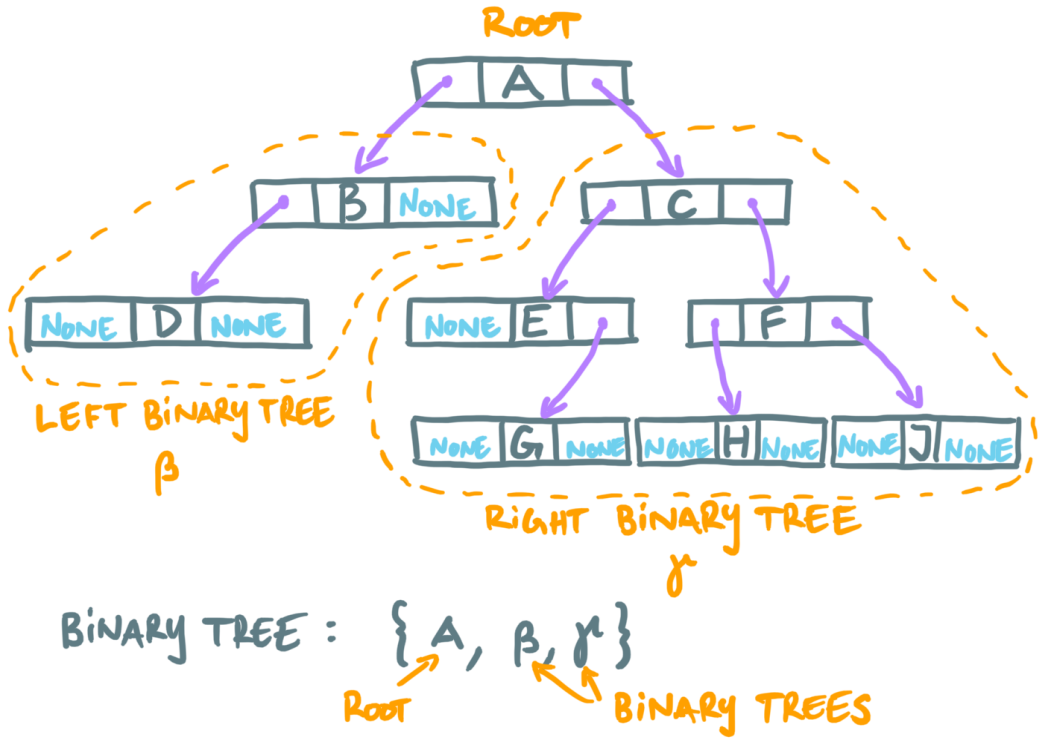
# BINARY TREE TRAVERSAL

***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 and 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:***

Preorder traversal:

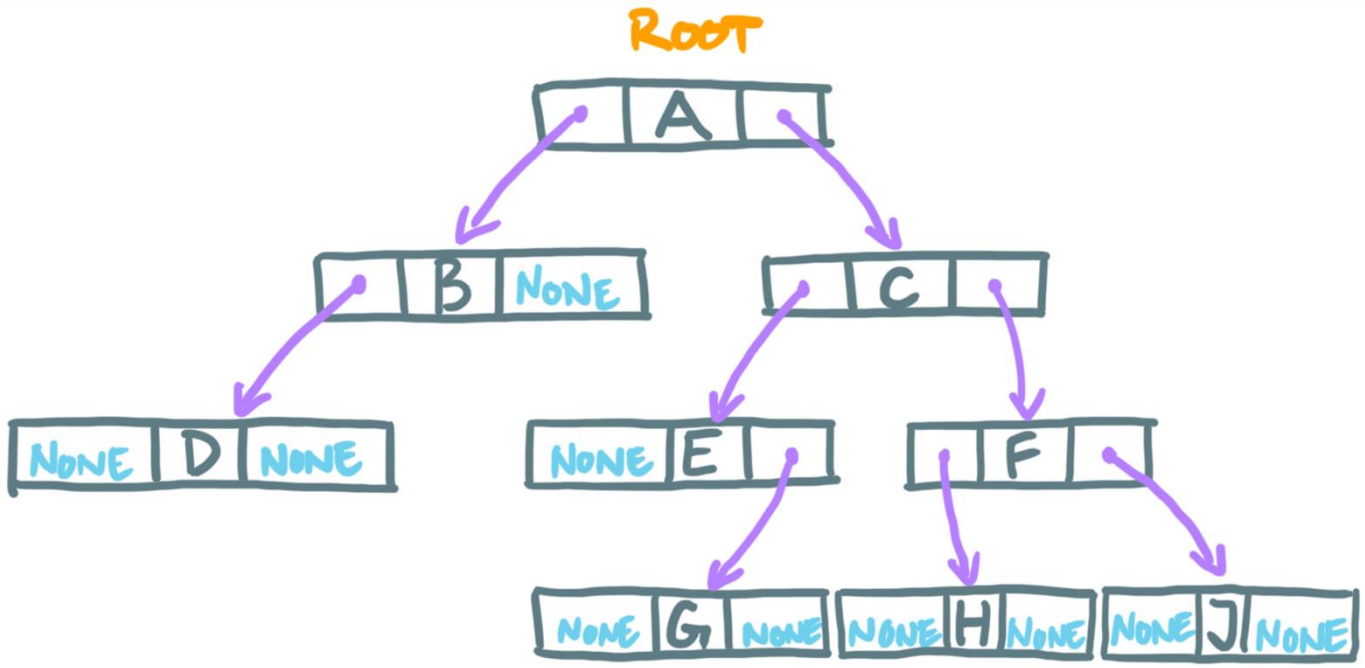
1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

Inorder traversal:

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

Postorder traversal:

1. Traverse the left subtree
2. Traverse the right subtree
3. 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*.

# OUTLINE

- Data structures
- Linked lists
- Trees
- Binary trees
- Binary search trees
  - Traversal
- **Other data structures**
  - Stack
  - Queue
  - Deque
  - Priority queue
- Heaps

# OTHER DATA STRUCTURES

We have discussed linked lists, a linear data structure, and (binary) trees, a nonlinear data structure.

Trees, and in particular binary trees, are a fundamental data structure that appear in many places in Computer Science.

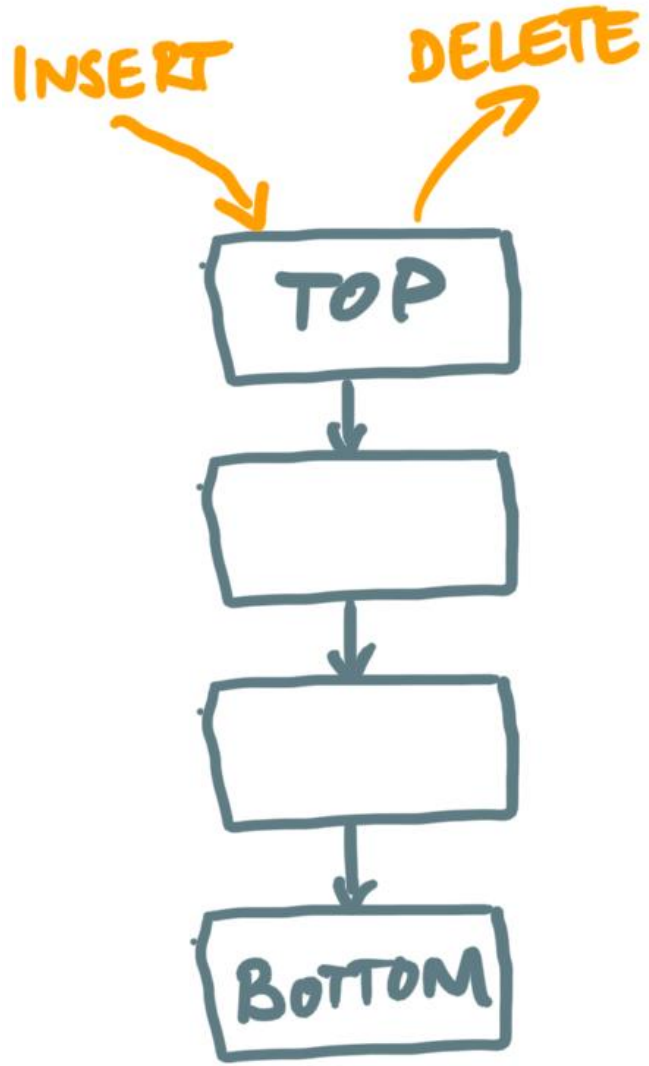
Other linear data structures are stacks, queues and deques.

These data structures follow similar ideas we have seen so far:

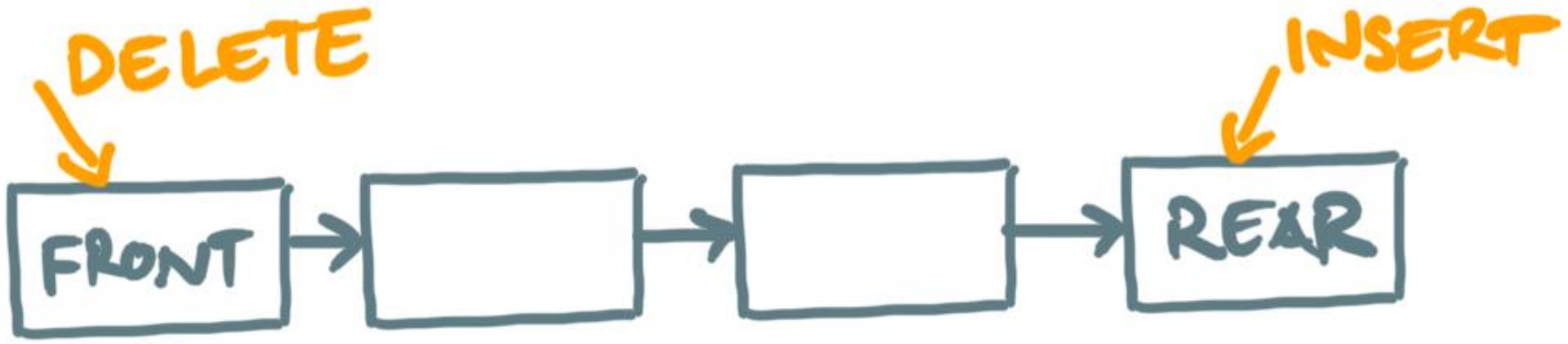
- A structure of nodes linked together.
- We ask the question: what is the time complexity for operations on the data structure such as node insertion, deletion, search, and so on?

# OTHER DATA STRUCTURES: STACK

- A ***stack*** is a linear list for which all insertions, deletions and usually all accesses are made only at *one end* of the list.
- This is often referred to as ***Last-In-First-Out (LIFO)***.
- 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:*** where else did we just see an implicit application of a stack?



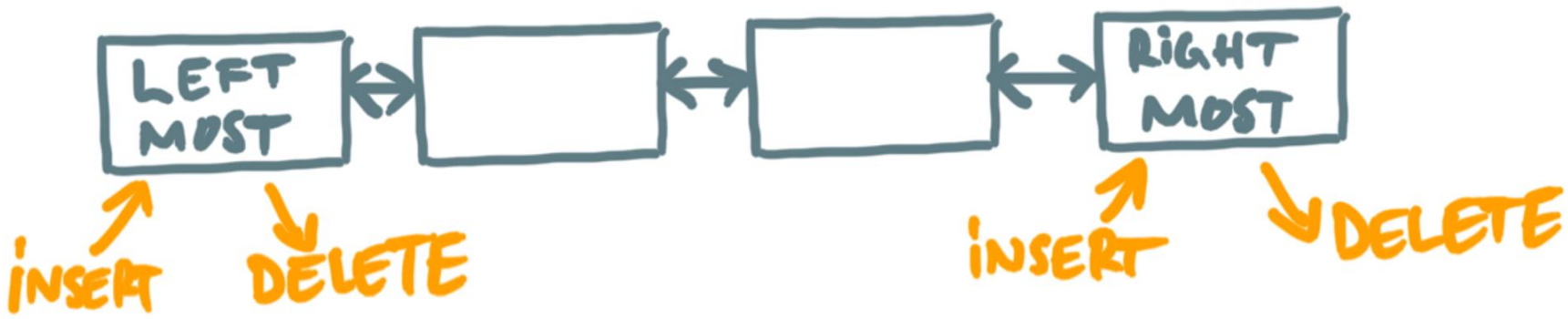
# OTHER DATA STRUCTURES: QUEUES




- 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)***.

A queue keeps the order of how elements arrive.

# OTHER DATA STRUCTURES: 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".

# OTHER DATA STRUCTURES: 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 queue that 'respects' each element's ***priority*** is called a ***priority queue***.

This implies an *order* among list elements that must be maintained.

**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

***You could use a sorted list:*** insertion of new elements is $O(n)$ removal and access are $O(1)$.

***You could keep a reference to the element with highest priority:*** insertion and access are $O(1)$, removal is $O(n)$.

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

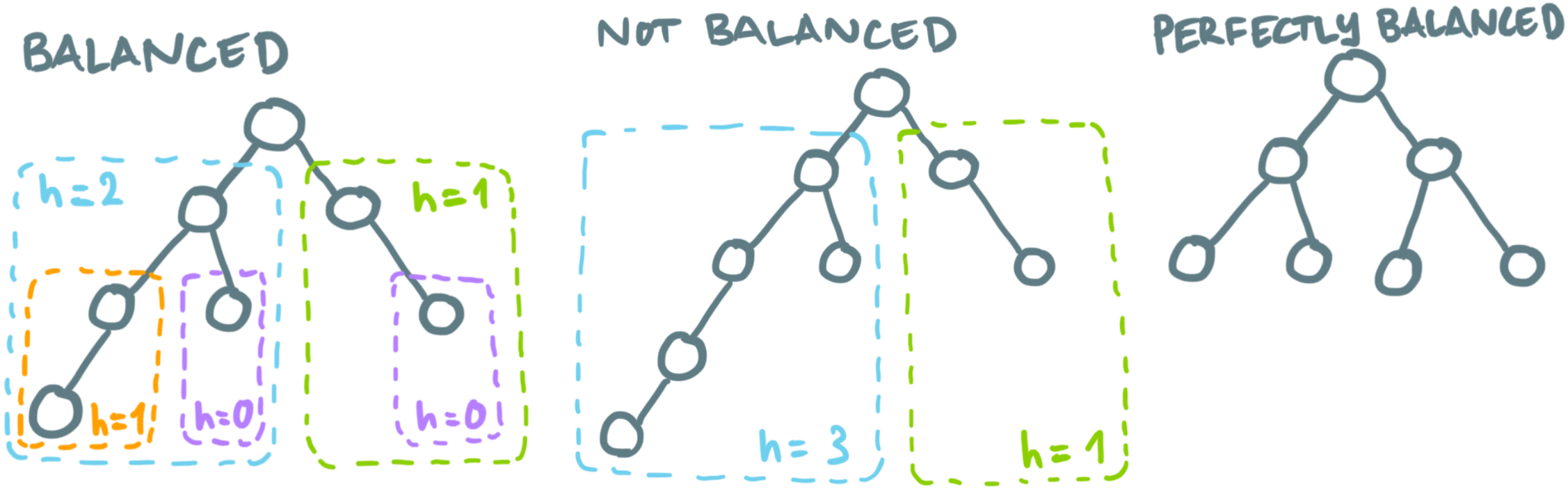
It is possible to use a ***balanced binary tree*** which can be *represented in a compact form using an array of keys only* (no extra overhead required for the bookkeeping of nodes in the tree). *This will lead us to the notion of a (binary)* ***heap***.

# BALANCED BINARY TREES

The *height h* of a tree is given by the maximum level of the tree.

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 $\lceil \log_2(n) \rceil = h$ holds, where $n$ is the number of nodes in the tree.



# OUTLINE

- Data structures
- Linked lists
- Trees
- Binary trees
- Binary search trees
  - Traversal
- Other data structures
  - Stack
  - Queue
  - Deque
  - Priority queue
- **Heaps**

# HEAP

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

$$h_1, h_2, \dots, h_n$$

such that

$$h_i \leq h_{2i} \quad (1)$$

$$h_i \leq h_{2i+1} \quad (2)$$

For all $i = 1, \dots, n/2$. The ***least element*** is $\mathrm{h}_1 = \min(h_1, h_2, \dots, h_n)$

We can just as well define the heap with the $\geq$ operator instead. The ***greatest element*** is then given by $\mathrm{h}_1 = \max(h_1, h_2, \dots, h_n)$ .

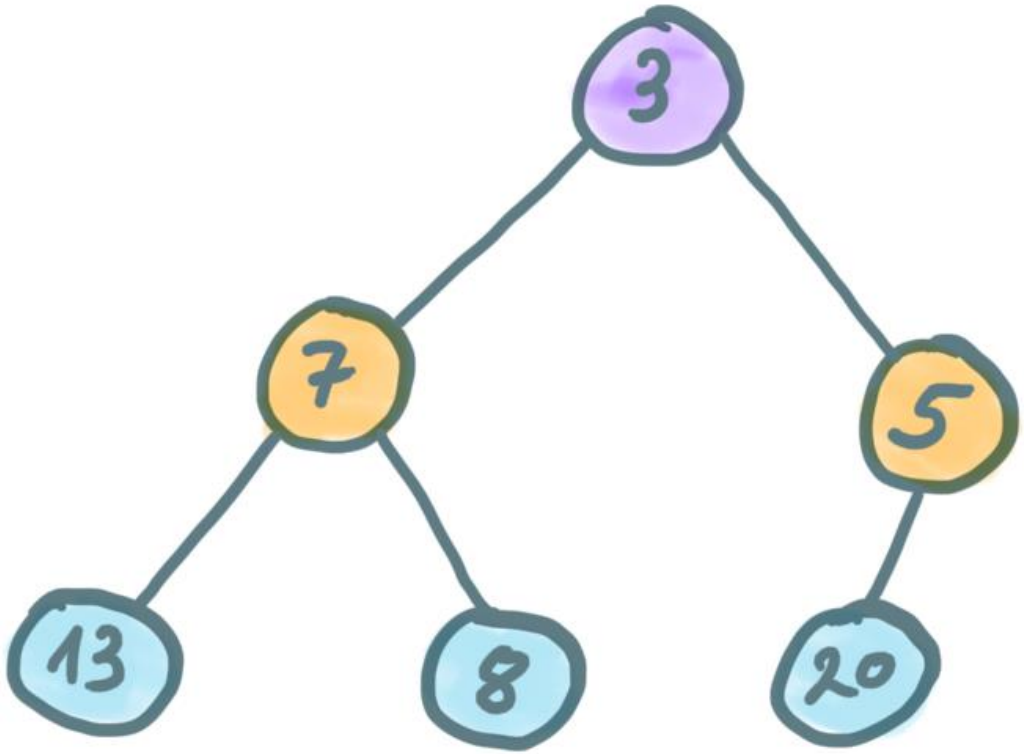
We refer to a ***min-heap*** for the former ($\leq$) and ***max-heap*** for the latter ($\geq$) definition, respectively.

***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 right)*.
- 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*).

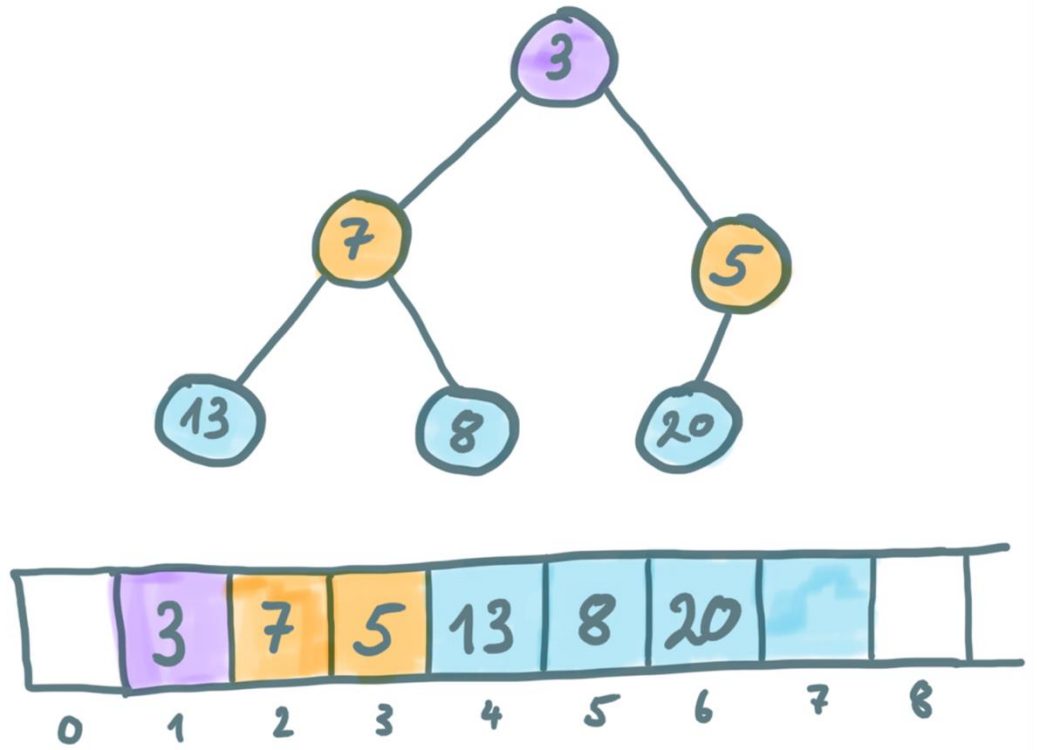


# HEAP

***Binary heap (or just "heap"):***

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.




A heap is the optimal data structure for a ***priority queue***.

Heap property:

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

# PRIORITY QUEUE WITH HEAP

### *Element insertion:*

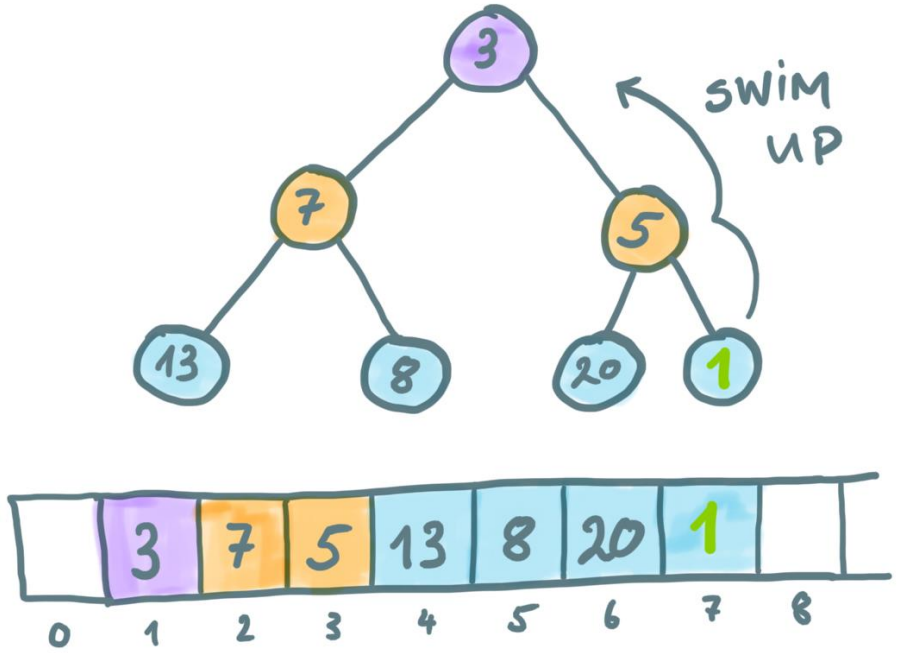
A new element is inserted at the index `n+1`.

Insertion will likely *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
```

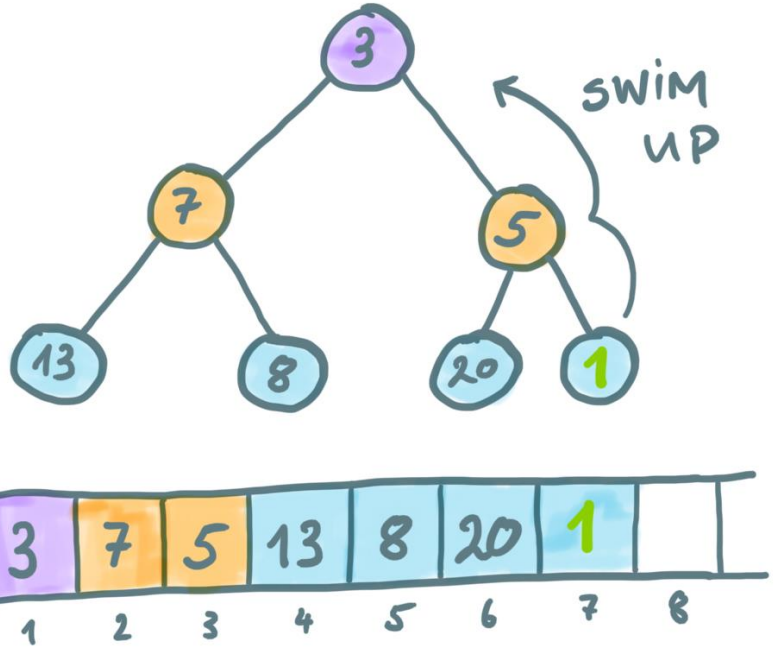



*Notes:*

- 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.

# PRIORITY QUEUE WITH HEAP

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




***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
```

***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.

# PRIORITY QUEUE WITH HEAP

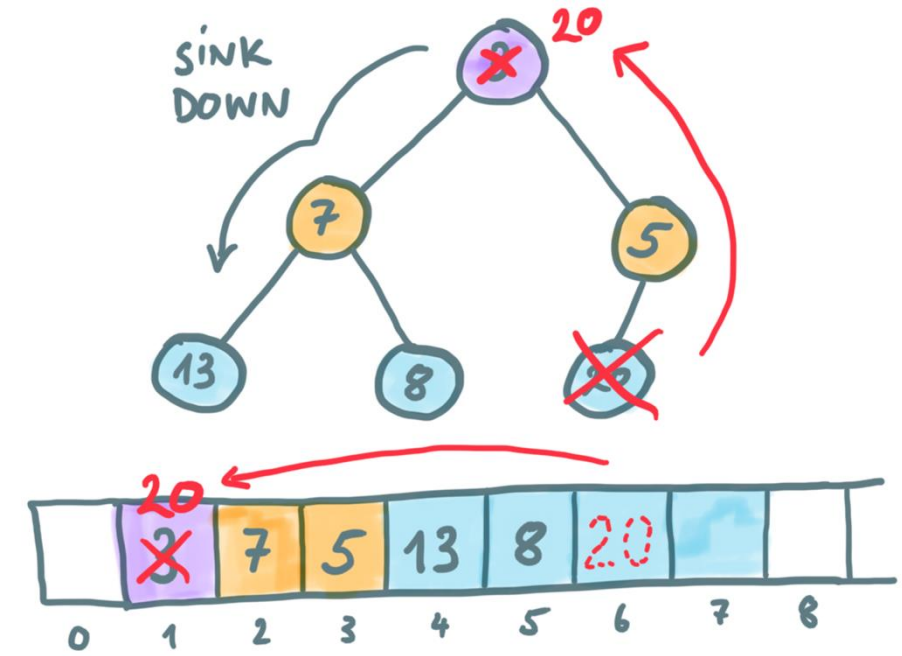
### *Element removal:*

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!***




```
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
```

# PRIORITY QUEUE WITH HEAP

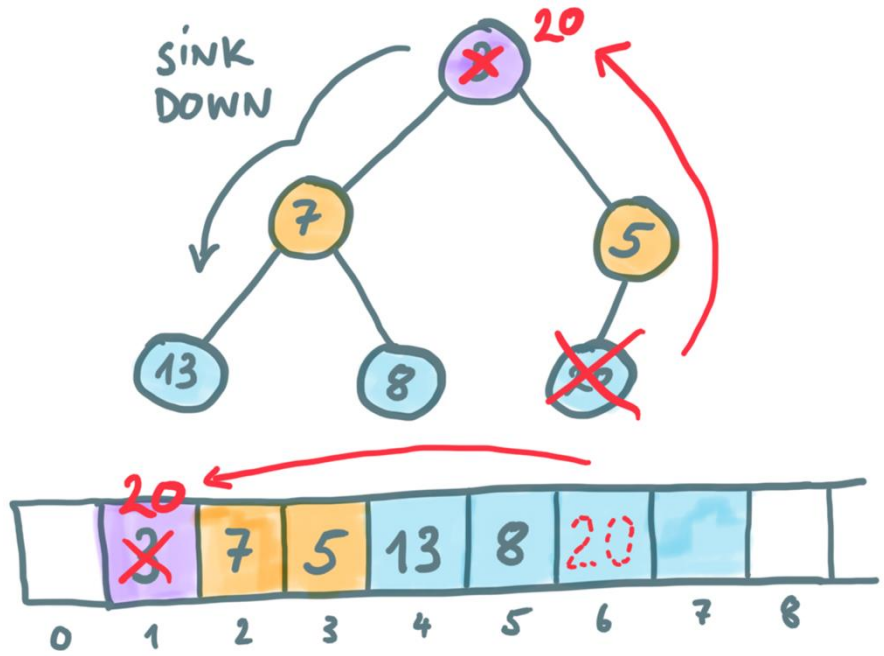
***Example:*** sift-down steps for removal of 3:

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
```




```
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
```

# THANK YOU