SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 17

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, October 27th 2022

LAST TIME

- Documenting Python code
- What are virtual environments and how to use them
- Configuration of Docker containers
- Using custom containers in CI

TODAY

Main topics: ***Data structures intro, iterators, binary trees***

Details:

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

AGENDA CHECK:

- Milestone 2A is due Tuesday. You will receive feedback for your M1 submission by 6:00pm today via the *Issues* tab in your project repository in https://code.harvard.edu/CS107. Milestone 2A provides a revision of your M1 proposal based on the feedback in the GitHub issue.

INTRODUCTION TO DATA STRUCTURES

Lecture 7: motivation for object oriented programming

> *Programs = Algorithms + Data Structures*
>
> *Niklaus Wirth*

Examples:

- Algorithms: Newton's method, least squares regression, AD
- 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.
- Dual or complex numbers are also abstract data types. They have two scalar `float` data structures, a real part and a dual/imaginary part.

INTRODUCTION TO DATA STRUCTURES

Data structures may have different memory layouts:

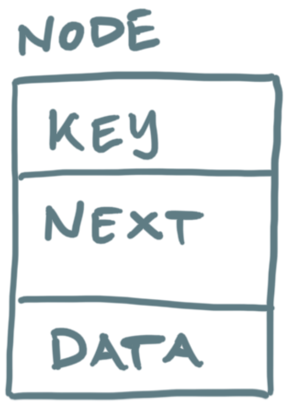
- Performance critical applications should have the *data close by in memory* (coalesced memory layout). An *array* is an example of a coalesced data structure. All elements are next to each other and we can access individual elements in $\mathcal{O}(1)$ complexity (*computing the address of an array 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. $\mathcal{O}(n)$ for a linked list with n nodes) but insertion or deletion of elements may be cheaper.

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 LIST → NODE

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

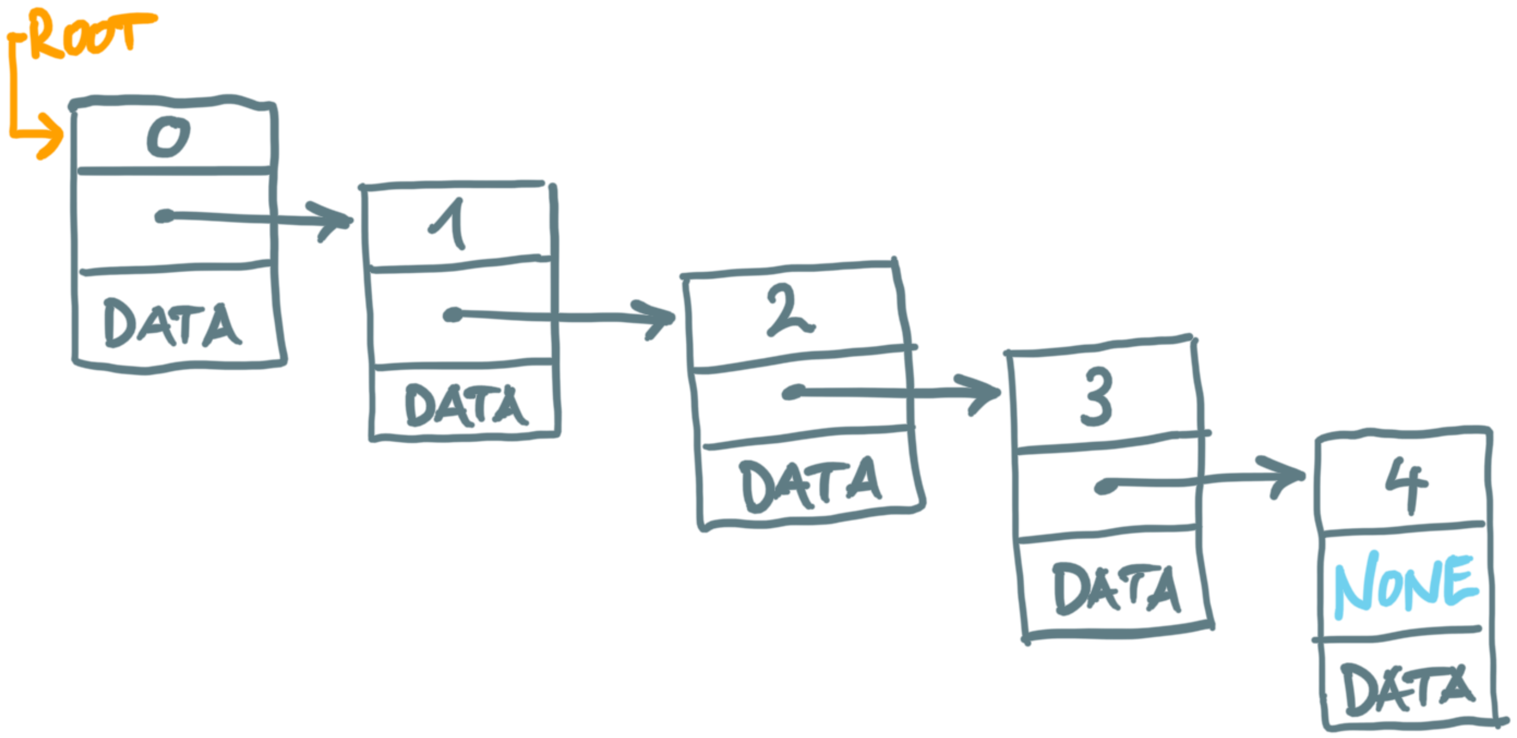


```python
class Node:
    """Node for a linked list."""
    def __init__(self, key, *,
                 next=None,
                 data=None):
        self.key = key
        self.next = next # node
        self.data = data
```

- 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.

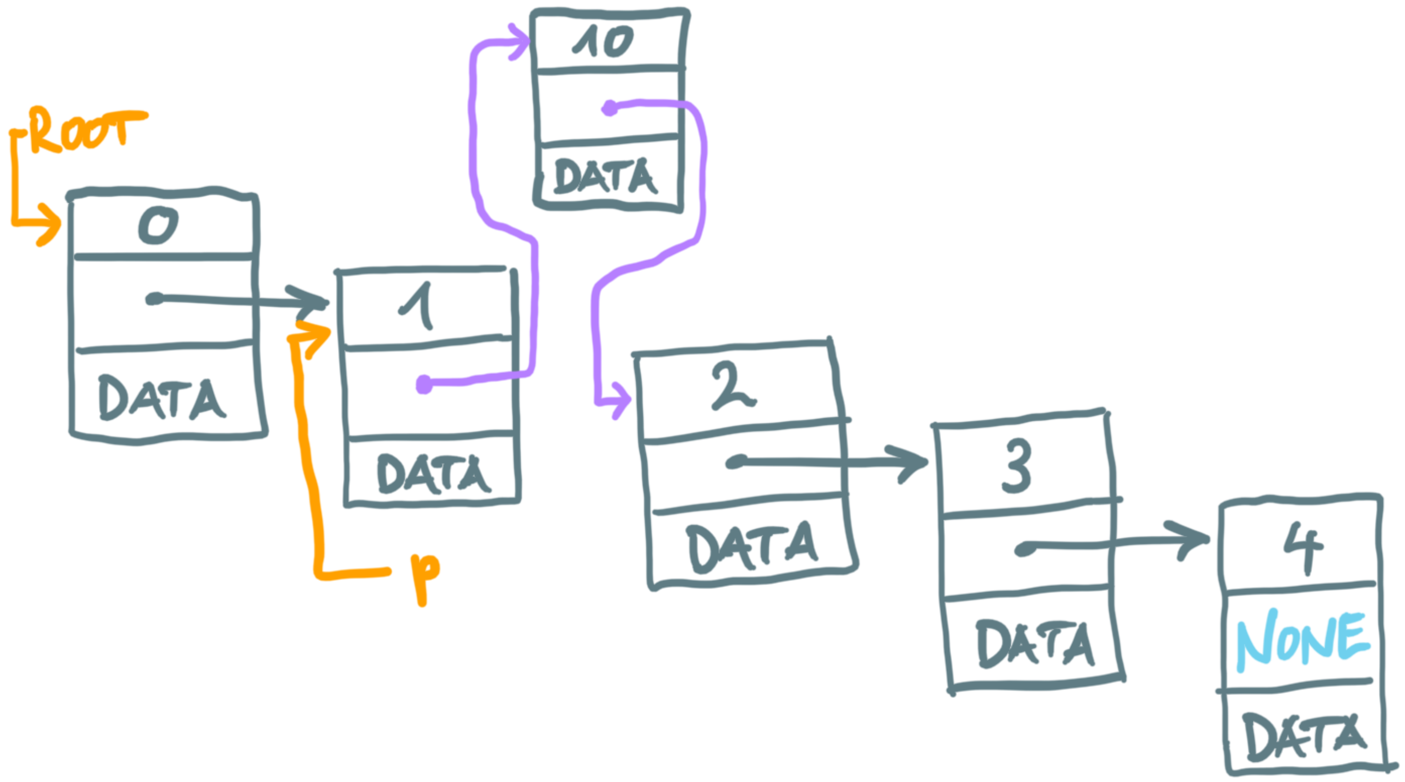
LINKED LIST

Simple example of a linked list:



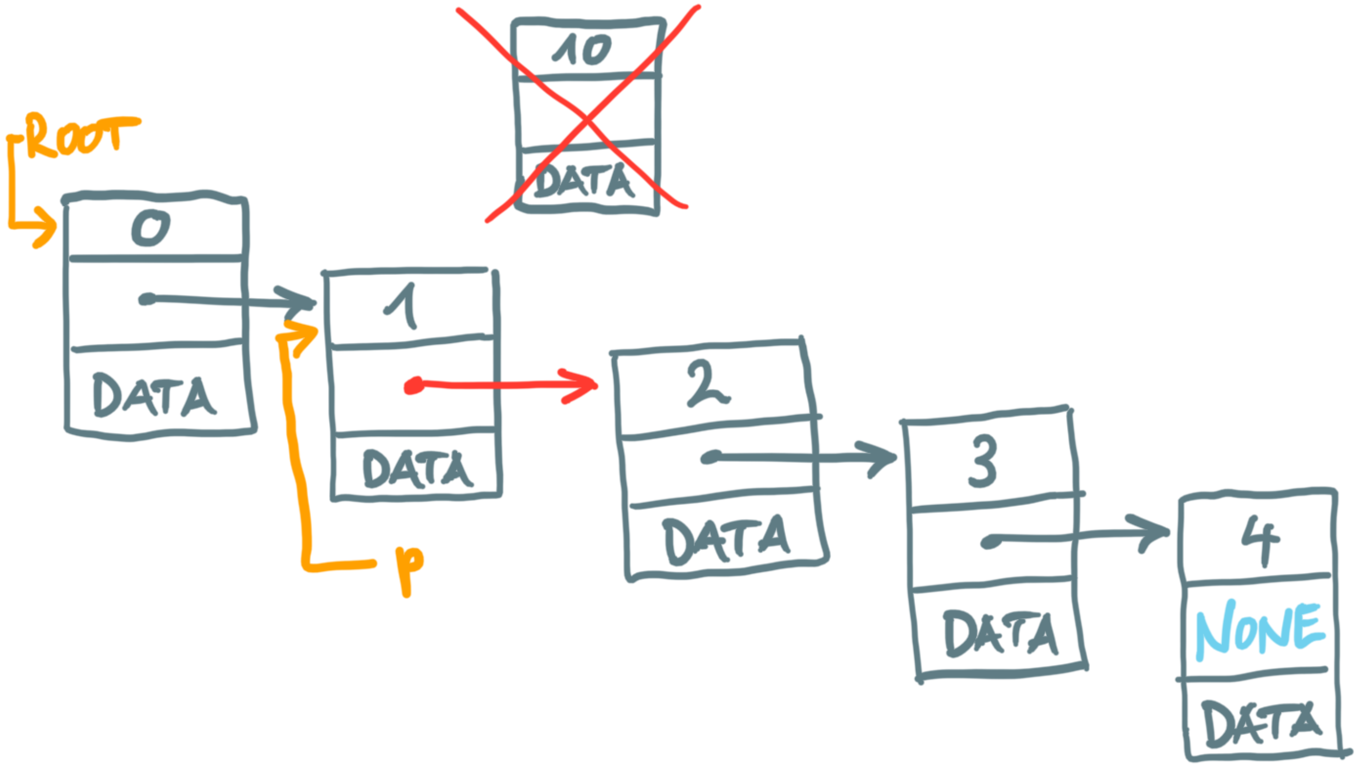
- The first node (in this example with key `0`) is the ***root*** node of the list.
- The `next` attribute of the last node has a `NULL` value (e.g. None in Python).
- List traversal is of complexity $\mathcal{O}(n)$ if there are n nodes.

LINKED LIST (NODE INSERTION)



- We can insert a new node with $\mathcal{O}(1)$ complexity at a reference node p (If p is known → you can get this reference by $\mathcal{O}(n)$ search otherwise).
- Insertion after p is trivial → *simply relink the* `next` *attributes*.
- How can we insert the new node *before* p?

LINKED LIST (NODE REMOVAL)



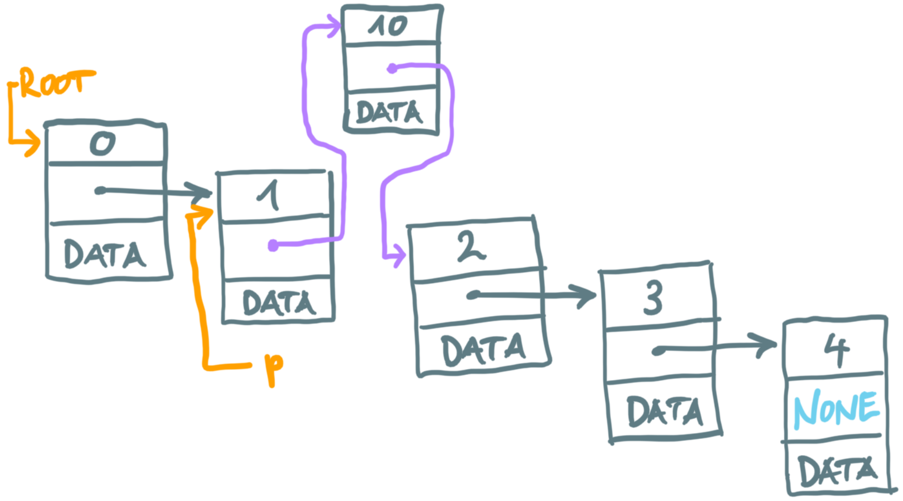
- Removing the *successor* of a node with reference p is trivial and is of $\mathcal{O}(1)$ complexity (if p is known).
- It is not straight forward to remove the node where reference p points to because in a singly linked list we have no knowledge about the *predecessor*.

LINKED LIST: EXAMPLE IMPLEMENTATION

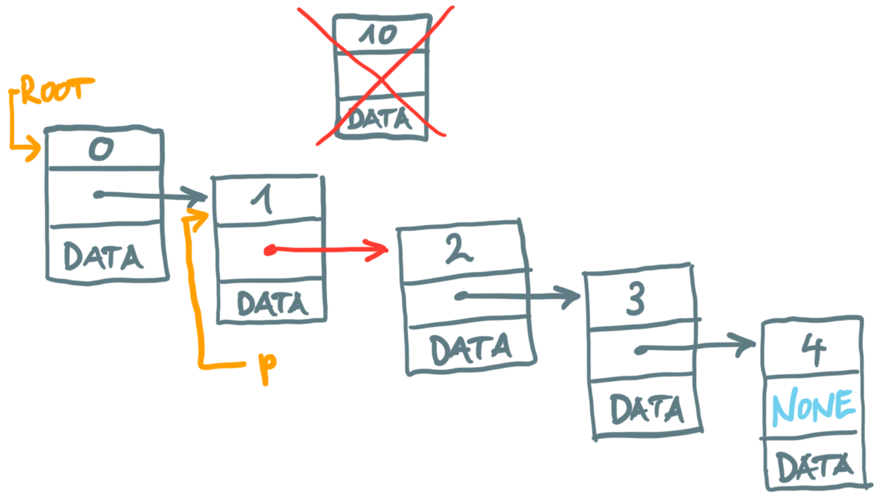
Example linked list implementation that addresses the following:

- 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. (***Recall:*** french deck custom sequence of lecture 8.)
- Insert a new node before or after the list node identified with `key`.
- Remove a list node identified by `key`.

Node insertion

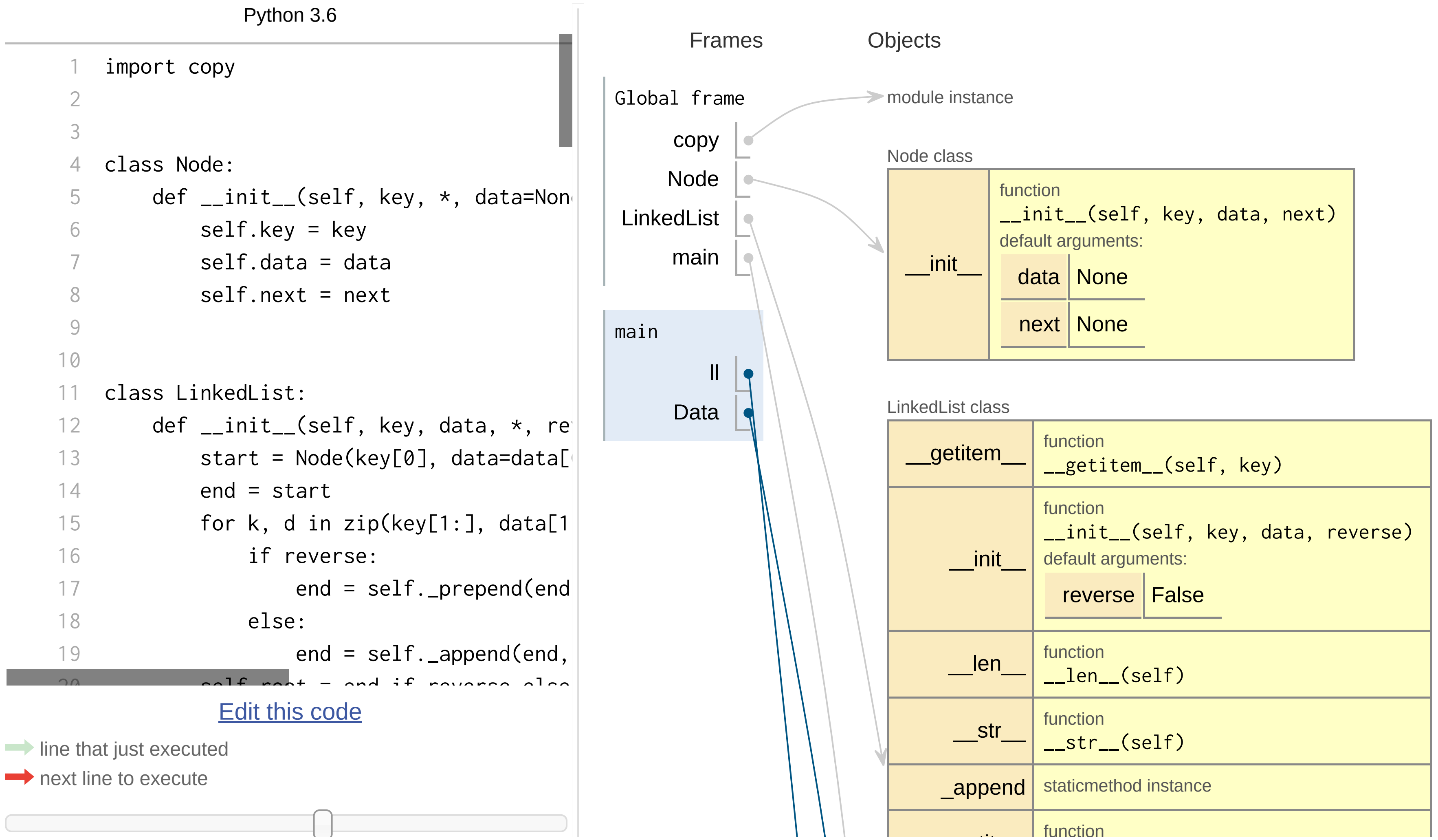


Node removal



LINKED LIST: EXAMPLE IMPLEMENTATION (TUTOR)

Example on pythontutor


Python 3.6

```
import copy


class Node:
    def __init__(self, key, *, data=Non
        self.key = key
        self.data = data
        self.next = next


class LinkedList:
    def __init__(self, key, data, *, re
        start = Node(key[0], data=data[
        end = start
        for k, d in zip(key[1:], data[1
            if reverse:
                end = self._prepend(end
            else:
                end = self._append(end,
```

Edit this code

LINKED LIST

Summary:

- A linked list is a *sequence of nodes* connected together with a `next` attribute which is a reference to the successor node.
- Insertion or removal of nodes is a $\mathcal{O}(1)$ operation. (For an array with fixed size (e.g. Python list), insertion or removal is $\mathcal{O}(n)$ on average with n array elements → elements must be moved).
- We discussed singly-linked lists, there are also *doubly linked lists* with `prev` and `next` attributes.
- You must traverse the list to find a particular node → this is an $\mathcal{O}(n)$ operation in the worst case.
- Compared to arrays → nodes need additional memory for the bookkeeping (`key` and `next` attributes) and there is no spatial locality (in memory) for linked lists.

ITERATORS

Iterator: provides a way to access the elements of an aggregate object *sequentially* *without exposing* its underlying representation.

- Note the word "*sequence*" in "*sequentially*".
- In the French deck custom sequence example of lecture 8 we defined `__getitem__` and `__len__` which caused our object to become ***iterable***.
- ***We did the same for our linked list → can we loop over it?***
- It worked for the custom sequence because we have ***delegated*** these operations to the underlying Python list (Python data model) → *we are not delegating anymore in the linked list data structure!*

ITERATORS

Python performs the following steps for iteration over an object:

1. Python automatically calls `iter(x)` on the object `x`. This requires that the special `__iter__` method is implemented in the type of `x`. See `iter()` for the documentation of this built-in function.
2. If the previous attempt fails with an `AttributeError`, `__getitem__` is called starting at *integer* index `0`. For our linked list, this only works if the `key` is an integer (and we should raise an `IndexError` instead of a `KeyError` in that case).
3. If both of the above fail, raise a `TypeError`.

ITERATORS

Example: Python list iterator

- Create a list:

```
>>> l = list('abc')
>>> type(l)
<class 'list'>
>>> dir(l)
['__add__', '__class__', other special methods..., '__iter__',
other special methods..., 'append', 'insert', 'pop', 'remove', 'sort']
```

→ list objects implement the `__iter__` special method!

- Create a list iterator:

```
>>> i = iter(l)
>>> type(i)
<class 'list_iterator'>  # iterators are class types!
>>> dir(i)
['__add__', '__class__', other special methods..., '__iter__',
other special methods..., '__next__', other special methods...]
```

→ iterator objects implement `__iter__` and `__next__` special methods!

ITERATORS

Example: Python list iterator

- Create a list iterator:

```
>>> i = iter(l)
>>> type(i)
<class 'list_iterator'>  # iterators are class types!
>>> dir(i)
['__add__', '__class__', other special methods..., '__iter__',
other special methods..., '__next__', other special methods...]
```

→ iterator objects implement `__iter__` and `__next__` special methods!

- ***Observe:***
  1. An iterator is a `class` (class types *encode state*) → ***why is that important for iterators? Or think about it from this perspective: why are iterators not implemented in the `list` class directly?***
  2. Iterators itself implement the `__iter__` method → ***what should it return?***
  3. Iterators implement the `__next__` special method.

ITERATORS

Example: Python list iterator

- ***Observe:***
  1. An iterator is a `class` (class types *encode state*) → ***why is that important for iterators? Or think about it from this perspective: why are iterators not implemented in the `list` class directly?***
  2. Iterators itself implement the `__iter__` method → ***what should it return?***
  3. Iterators implement the `__next__` special method.
- The interface for the iterator object is the `next()` built-in:

```
>>> next(i)
'a'
>>> next(i)
'b'
>>> next(i)
'c'
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration  # when the iterator is exhausted → StopIteration is raised!
```

ITERATORS

Example: Python list iterator

- The interface for the iterator object is the `next()` built-in:

```
>>> next(i)
'c'
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration  # when the iterator is exhausted → StopIteration is raised!
```

When the iterator is exhausted a `StopIteration` exception is raised.

- A `for`-loop in Python works like this:

1. Obtain iterator for an *iterable*.
2. Call `next()` on the iterator at the start of every loop iteration.
3. Terminate the loop when `StopIteration` is raised.

```
>>> for i in l: # it = iter(l)
...     i       #  i = next(it)
...
'a'
'b'
'c'
```

ITERATORS

- ***The iterator API in Python consists of two parts:***
  1. The `next()` built-in returns the next element of the iterable.
  2. If the iterator is exhausted a `StopIteration` exception must be raised.
- The return value of `iter()` *called on an iterator must be the iterator itself*.
- The `next()` interface *hides the internal details of the iterable*. The user only cares about the next element, regardless of how it is obtained. *How* elements are returned from a linked list is different than for a built-in Python list, *and it is a detail*.
- Iterators must encode state. *You can have multiple iterators for the same iterable at any given time.*
- Some objects allow for *reverse iteration* with the `reversed()` built-in. This requires an implementation of the `__reversed__` method.

LINKED LIST REVISITED WITH ITERATORS

We now return to our linked list implementation:

- Note that the *sequence protocol* with `__getitem__` and `__len__` only works if the sequence can be indexed with *integers*. *Internally, Python creates an iterator from this protocol*.
- → *We do not want to limit our node* `key`*'s to integers only!*

Furthermore: if your data type is supposed to be *iterable*, then you should implement iterators *and not rely on the sequence protocol*. The Python data model works with the `iter()` built-in for iterables (*the sequence protocol is just converted to an iterator and acts as a fallback*).

- For our linked list we need the `__iter__` method ***and*** a *forward iterator class*.
- For *a doubly-linked list* we could implement a *reverse iterator class* using the `__reversed__` special method in addition.

TREES

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

Linear lists:

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

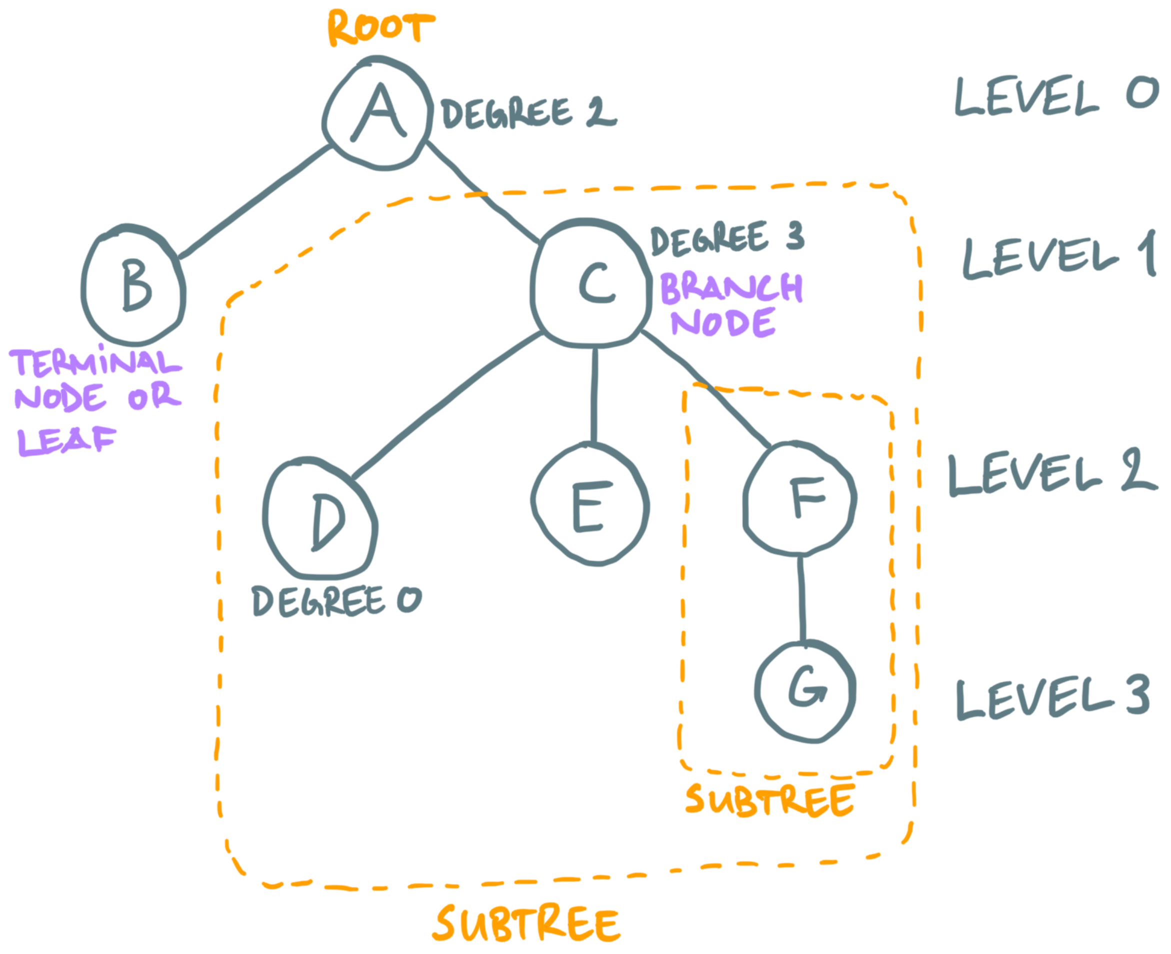
Trees:

- 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.

TREES

Ordinary tree:



- 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.

TREES

Some examples where tree structures are to be found:

- Parsing of code (abstract syntax tree → AST)
- Evolutionary trees in biology
- Unix file system
- Parallel computing (communication patterns)
- Graph theory

Note:

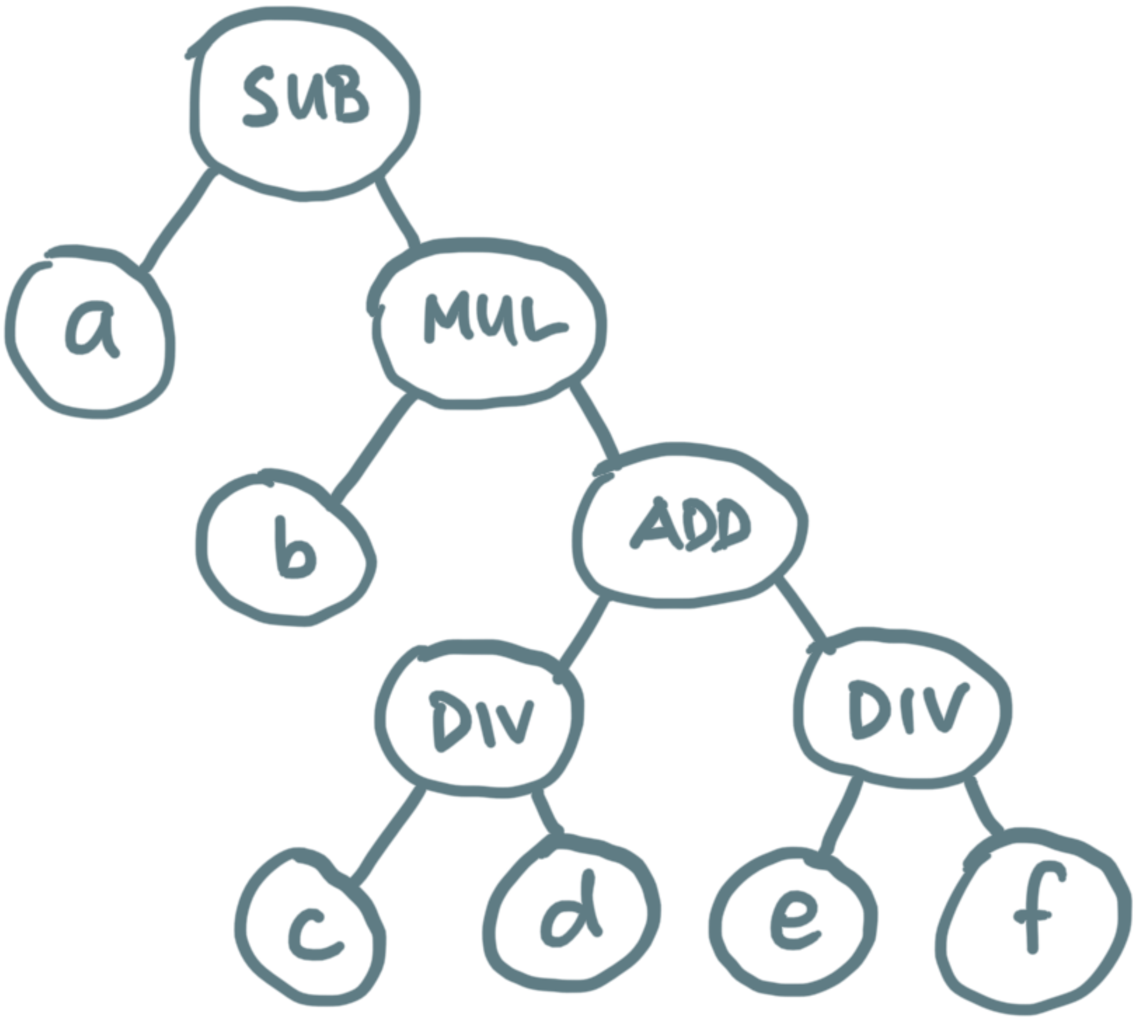
- The nodes in a tree can be modeled the same way we did for the linked list before.
- If we remove the root node of the tree (or any subtree) we get a forest of trees (multiple disjoint trees).

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))))
```

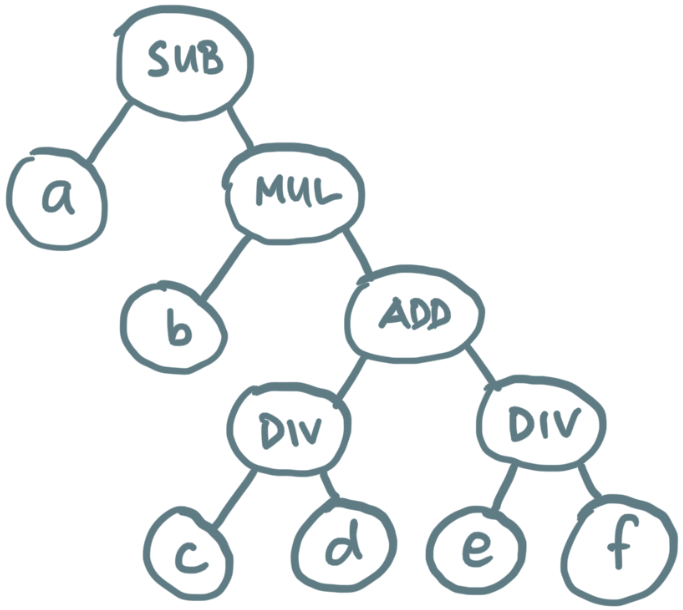
- The recursion is implied by operator precedence:

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

Expression:

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

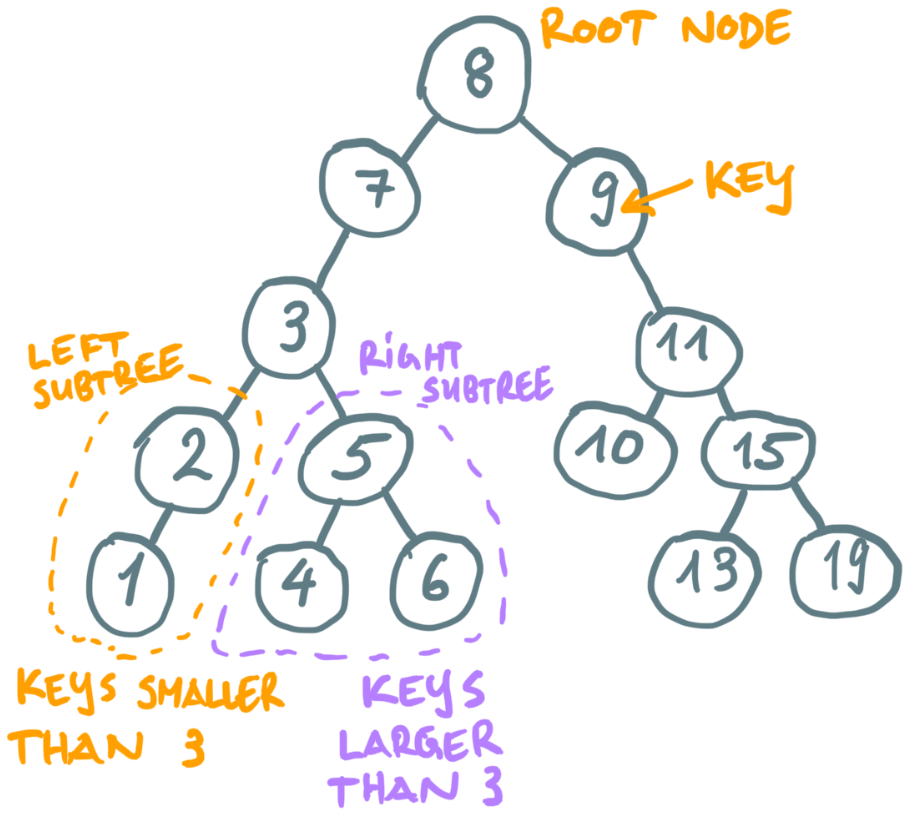
Expression tree:



BINARY SEARCH TREE

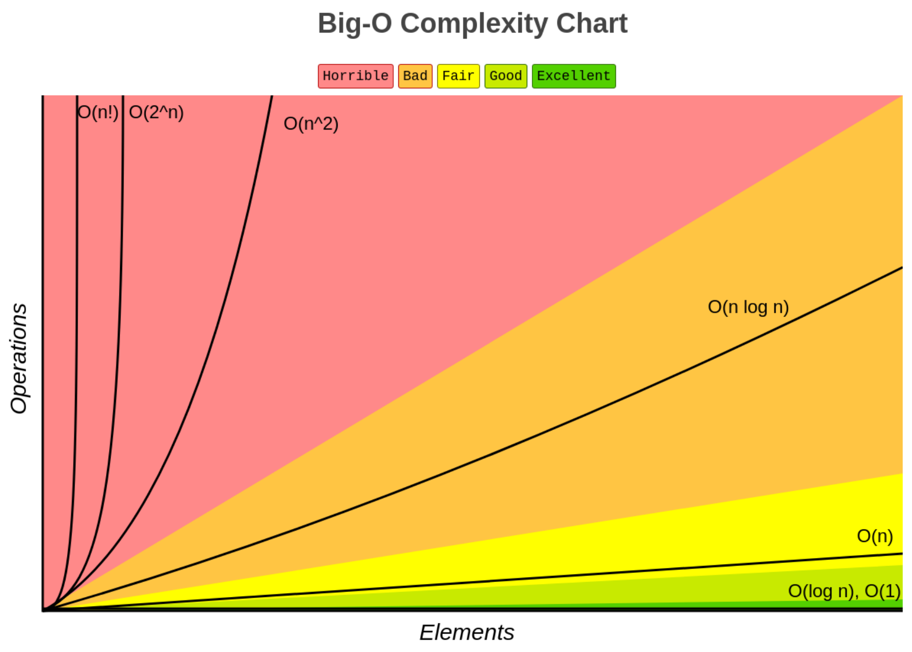
A binary search tree (BST) is an ***ordered*** binary tree with key values *comparable* with each other. A BST has the property that any key in the nodes contained in the ***left*** subtree of the root node v are strictly ***smaller*** than the key in v and any key in nodes contained in the ***right***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

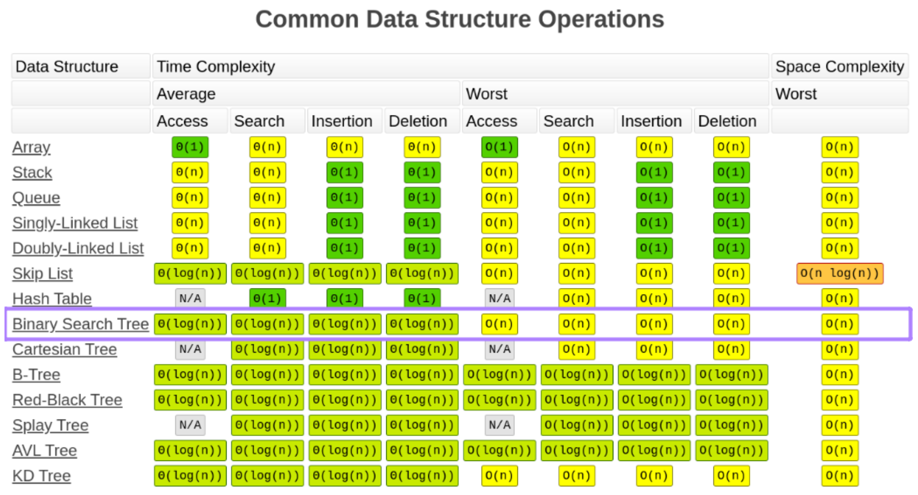
- A BST is one of the most fundamental algorithms in computer science.
- If the root node of a BST does not have a *left* subtree, it means that the key of the root node is the *smallest value* (similarly for the *largest value*).
- We are only concerned with a *single* occurrence of key values.



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	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

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:

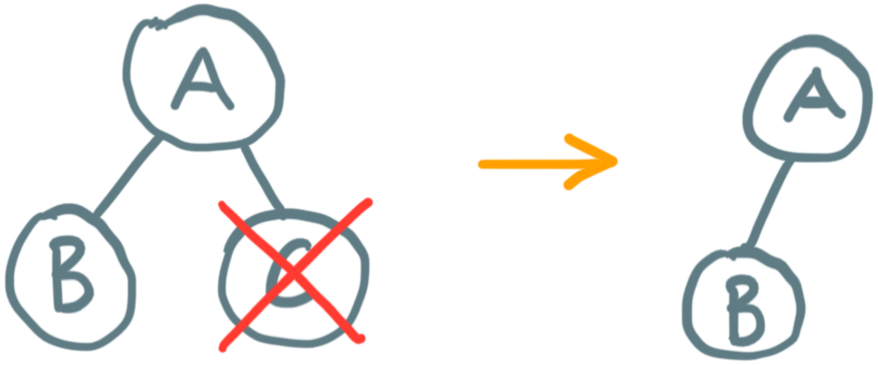
- Searching a BST is a ***recursive*** algorithm.
- If there is a search *hit*, return the associated node value.
- If there is a search *miss*, return `NULL` (e.g. `None` in Python or `nullptr` in C++).
- ***Algorithm:*** start at the root node and compare the search value with the `key` of the node.
  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

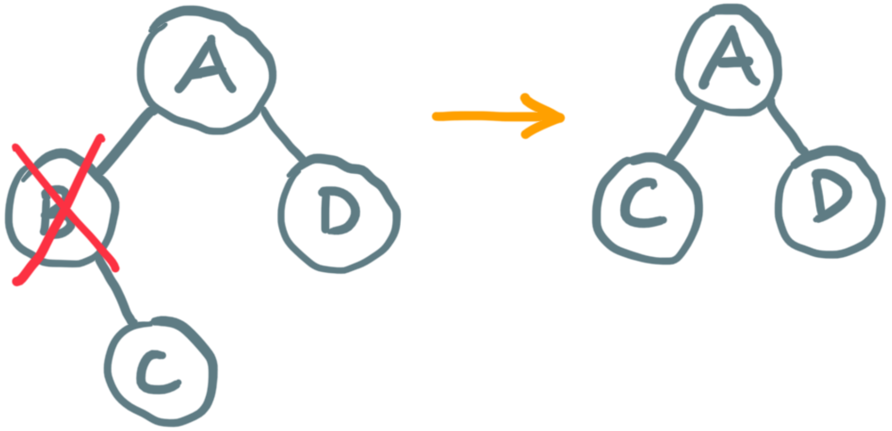
Node deletion for degrees 0 and 1:

- 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:



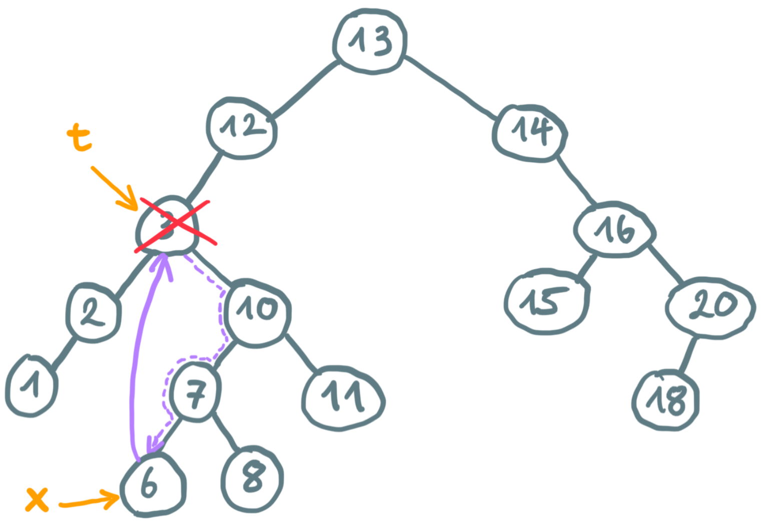
Node removal with single child:

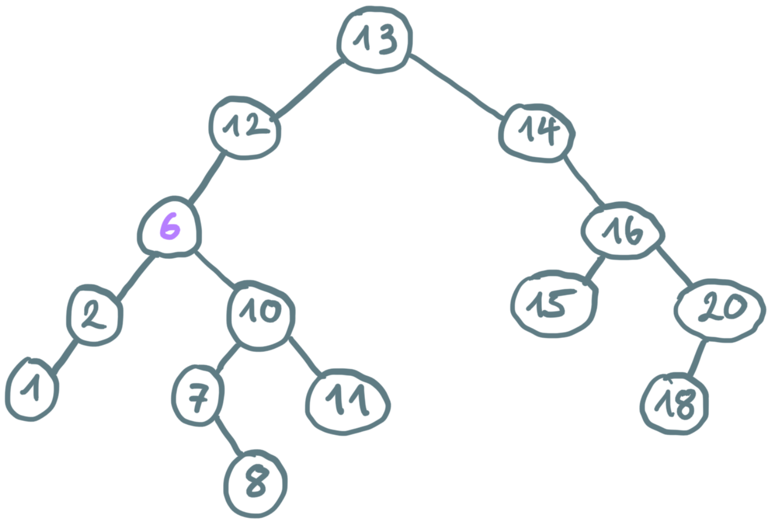


BINARY SEARCH TREE

***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?

RECAP

- Introduction to data structures
- Linked lists
- Iterators
- Trees, binary trees and binary search trees

Further reading:

- N. Wirth, "*Algorithms + Data Structures = Programs*", Prentice-Hall, 1976.
- D. Knuth, "*The Art of Computer Programming*", Volume 1, 3rd Edition, Addison-Wesley Professional, 1997.
- Iterators in Chapter 5 of E. Gamma, R. Helm, R. Johnson and J. Vlissides "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison Wesley Professional, 1995.