CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 16

Thursday, October 28th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- `docker` containers and containerization
- Building `docker` containers

OUTLINE

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

INTRODUCTION TO DATA STRUCTURES

Lecture 6:

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

Examples:

- Algorithms: Newton's method, least squares regression, AD
- Data structures:
  - `python` built-in: `list`, `tuple`, `dict`
  - Singly or doubly linked lists
  - Stacks or queues
  - Trees
  - Heaps (priority queues)

INTRODUCTION TO DATA STRUCTURES

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 *data structures* to realize the expected user behavior.
- These data structures can be as simple as a built-in integer or float or more complex data structures like lists, dictionaries or trees.

Examples:

- A graph is an abstract data type.
- A dual number is also an abstract data type. It has two `float` data structures, a real part and a dual part.

INTRODUCTION TO DATA STRUCTURES

Data structures 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.
- Other data structures, like a linked list for example, allocate memory for new elements dynamically and are chained together using pointers. Individual elements may not be close by in memory, which usually means the complexity of element access is higher but insertion or deletion of elements may be cheaper.

INTRODUCTION TO DATA STRUCTURES

References:

- N. Wirth, "*Algorithms + Data Structures = Programs*", Prentice-Hall, 1976.
- D. Knuth, "*The Art of Computer Programming*", Volume 1, 3rd Edition, Addison-Wesley Professional, 1997.

LINKED LIST

- Linked lists (or linear lists) are one of the simplest data structures.
- They consist 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:

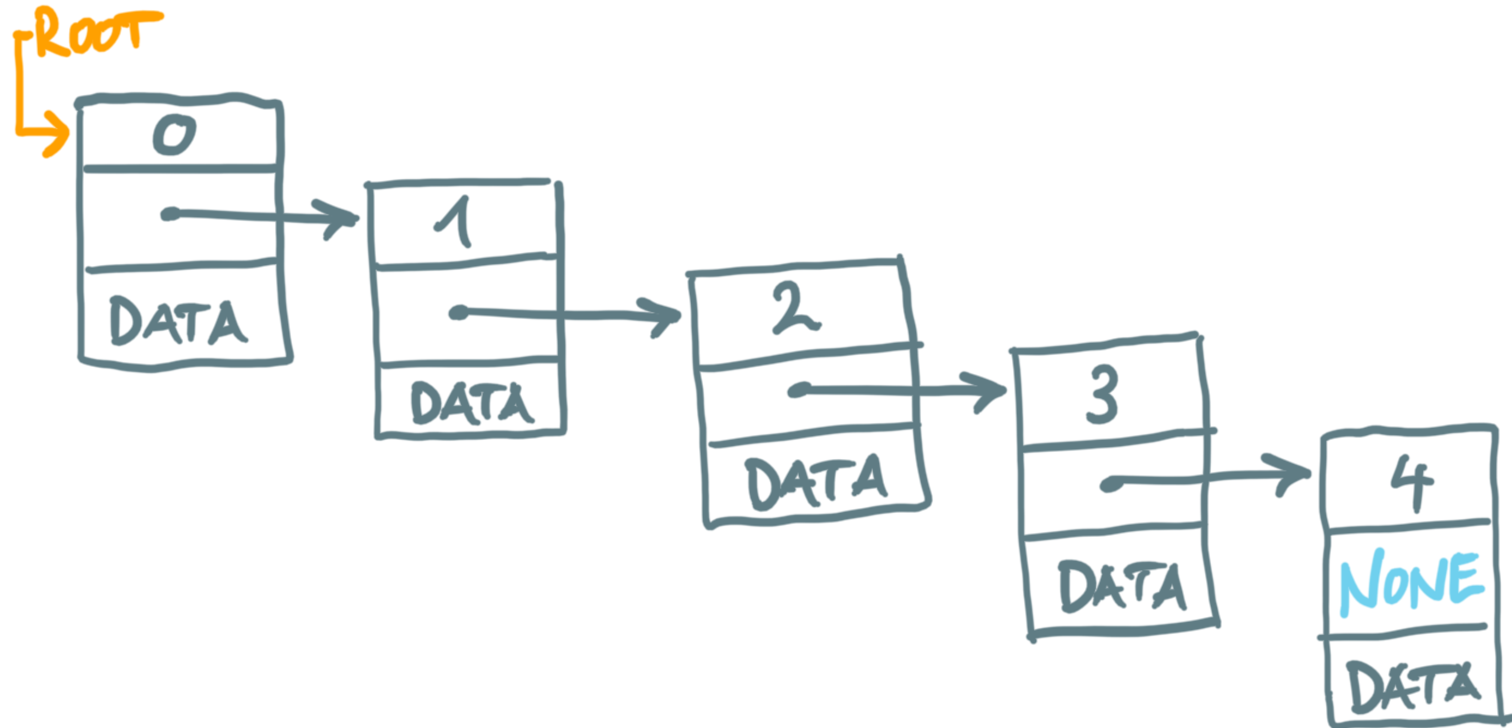
```python
class Node:
    def __init__(self,
                 key, *,
                 data=None,
                 next=None):
        self.key = key
        self.data = data
        self.next = next
```

python class to model a node in a singly linked list.

- 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.

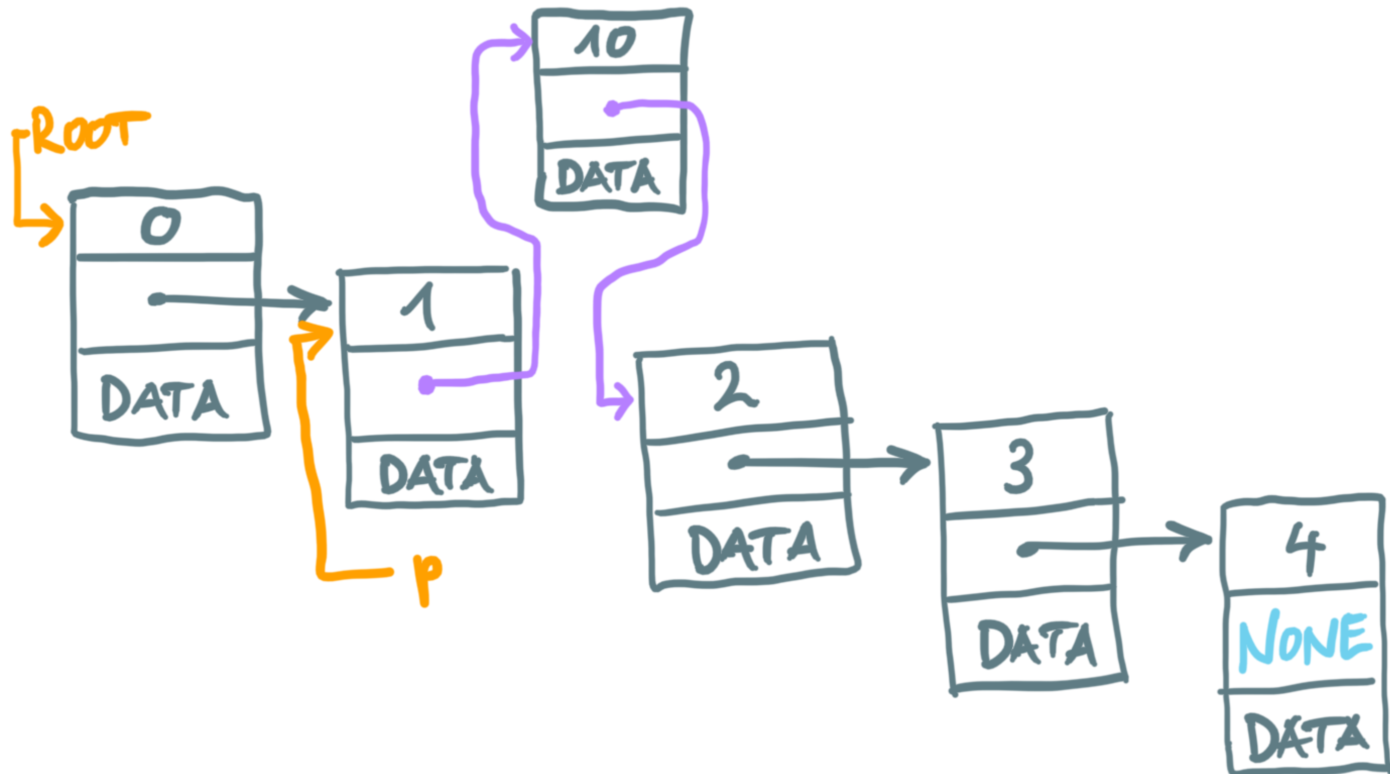
LINKED LIST

Simple example of a linked list:



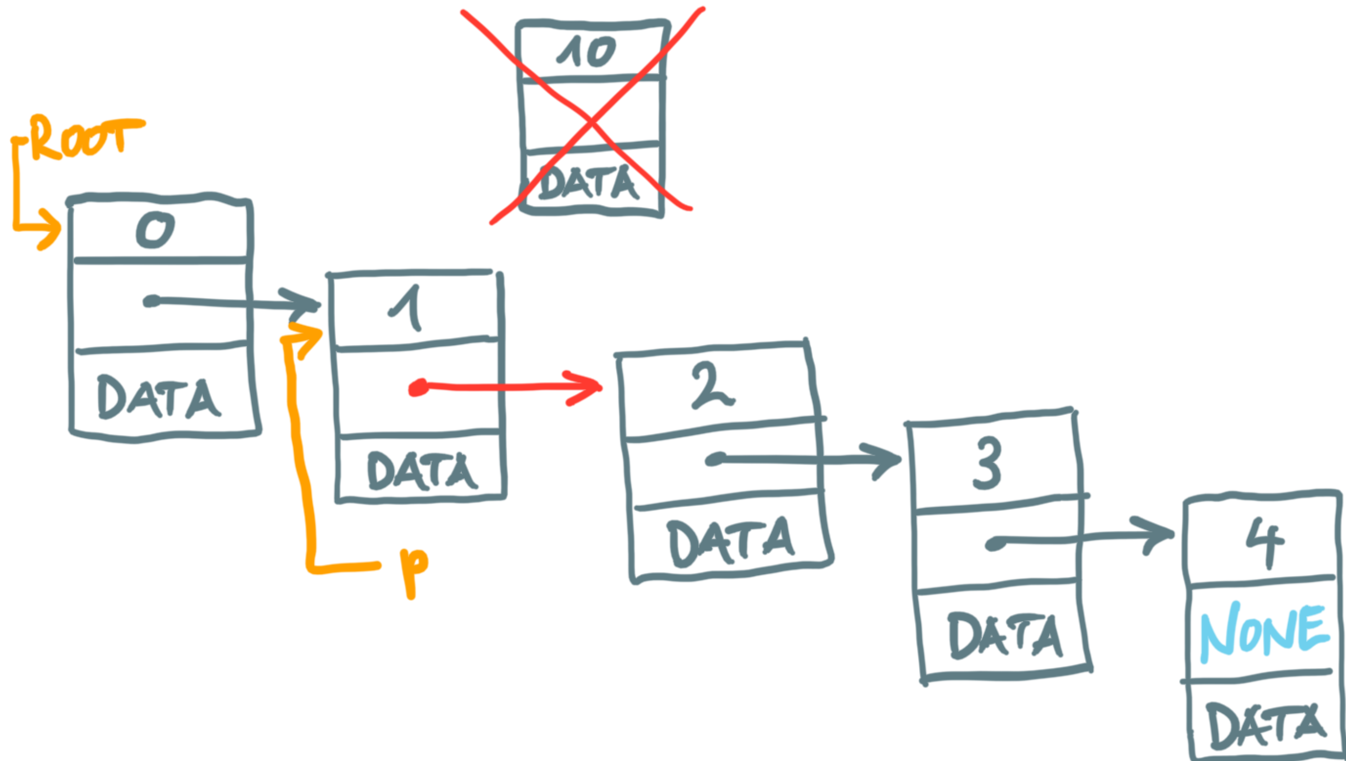
- The first node, here with key `0`, is the root node of the list.
- The `next` attribute of the last node has a `NULL` value, e.g. None in `python`.
- List traversal is of complexity $\mathcal{O}(n)$ if there are n nodes.

LINKED LIST: NODE INSERTION



- We can simply insert a new node with $\mathcal{O}(1)$ complexity if we have a reference `p` to a node, e.g. node `1`. Insertion after `p` is trivial, we simply relink the `next` attributes.
- How can we insert the new node *before* `p`?

LINKED LIST: NODE REMOVAL



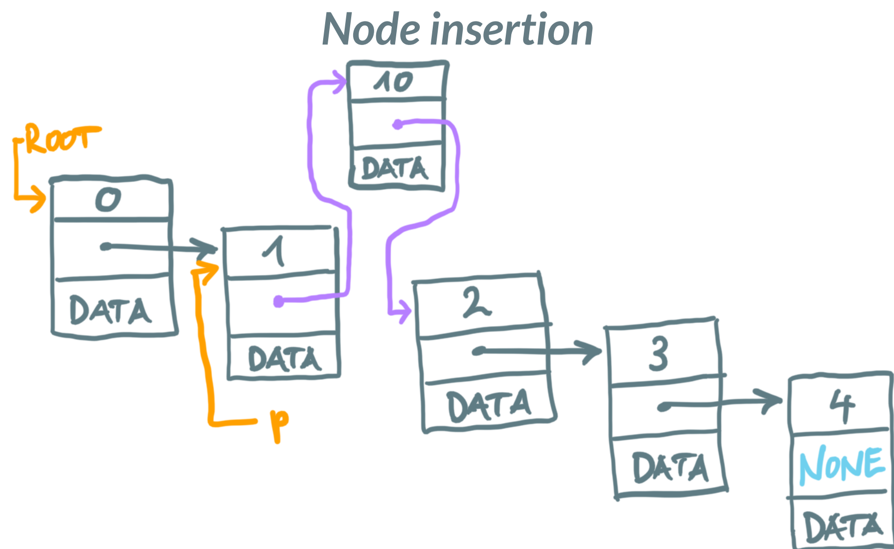
- Removing the *successor* of a node with reference p is trivial and is of $\mathcal{O}(1)$ complexity.
- It is not straight forward to remove the node with reference p in a singly linked list as we do not know about its *predecessor*.

LINKED LIST: EXAMPLE IMPLEMENTATION

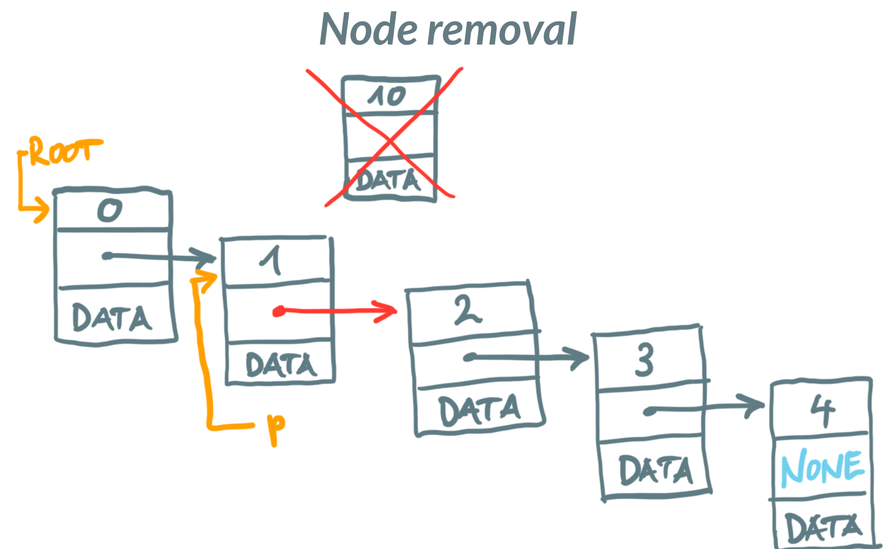
We now go through an example implementation that addresses the following:

- Create a linked list with `key/data` pairs. Optionally we want to create the list in reverse order.
- A linked list is a sequence (***recall:*** French deck custom sequence example). The linked list should support the sequence protocol.
- 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

```
1  import copy
2
3
4  class Node:
5      def __init__(self, key, *, data=Non
6          self.key = key
7          self.data = data
8          self.next = next
9
10
11 class LinkedList:
12     def __init__(self, key, data, *, re
13         start = Node(key[0], data=data[
14         end = start
15         for k, d in zip(key[1:], data[1
16             if reverse:
17                 end = self._prepend(end
18             else:
19                 end = self._append(end,
```

line that just executed

next line to execute

< Prev   Next >

Frames   Objects

LINKED LIST: SUMMARY

- A linked list is a sequence of nodes connected together with a `next` attribute that holds a reference to the successor node.
- Insertion or removal of nodes is a $\mathcal{O}(1)$ operation. An array with fixed size (e.g. `python` list) insertion or removal is $\mathcal{O}(n)$ on average with n elements in the array (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 node you are looking for which is a $\mathcal{O}(n)$ operation in the worst case.
- Nodes need additional memory for the bookkeeping (`key` and `next` attributes). There is no spatial memory locality in linked lists.

ITERATORS

Optional reading: *Iterators* in Chapter 5 of *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison Wesley Professional, 1995.

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

- The word "sequence" shows up again.
- In our custom sequence example of lecture 7 (French deck), we defined `__getitem__` and `__len__` which caused our object to become *iterable*.
- We did the same for our linked list before, so we can loop over it, right? Let's see.
- It worked for our custom sequence because we have ***delegated*** these operations to the underlying `python` list. We are not delegating anymore in our linked list data structure.

ITERATORS

`python` performs the following steps when it is asked to iterate over an object:

1. `python` automatically calls `iter(x)` on the object `x`. This requires that the dunder method `__iter__` 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`.
3. If both of the above fail, raise a `TypeError`.

ITERATORS

Example for a python list iterator:

- We create a list:

```
1  >>> l = list('abc')
2  >>> type(l)
3  <class 'list'>
4  >>> dir(l)
5  ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__deli
6  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__g
7  '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__le
8  '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__revers
9  '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',
10 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- It implements the `__iter__` method. We create an iterator:

```
1 >>> i = iter(l)
2 >>> type(i)
3 <class 'list_iterator'>
4 >>> dir(i)
5 ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
6 '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lengt
7 '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setatt
8 '__sizeof__', '__str__', '__subclasshook__']
```

ITERATORS

Example for a python list iterator:

- It implements the `__iter__` method. We create an iterator:

```
1 >>> i = iter(l)
2 >>> type(i)
3 <class 'list_iterator'>
4 >>> dir(i)
5 ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
6 '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lengt
7 '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setatt
8 '__sizeof__', '__str__', '__subclasshook__']
```

- ***Notice:***

1. An iterator is a `class` (a *type*). 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?
2. Iterators themself implement the `__iter__` method.
3. Iterators implement the `__next__` method.

ITERATORS

Example for a `python` list iterator:

- ***Notice:***
    1. An iterator is a `class` (a *type*). 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?
    2. Iterators themself implement the `__iter__` method.
    3. Iterators implement the `__next__` method.
- The interface for an iterator is the `next()` built-in:

```
1 >>> next(i)
2 'a'
3 >>> next(i)
4 'b'
5 >>> next(i)
6 'c'
7 >>> next(i)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 StopIteration
```

ITERATORS

Example for a `python` list iterator:

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

```
1 >>> next(i)
2 'c'
3 >>> next(i)
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 StopIteration
```

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

- A `for`-loop in `python` works like this: obtain an iterator for the object, call `next()` on it at the start of every loop iteration, terminate when `StopIteration` is raised.

```
1 >>> for i in l:
2 ...     i
3 ...
4 'a'
5 'b'
6 '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.
- 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 are now returning 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 and the sequence protocol is just converted to an iterator.

- 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 as well.)

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

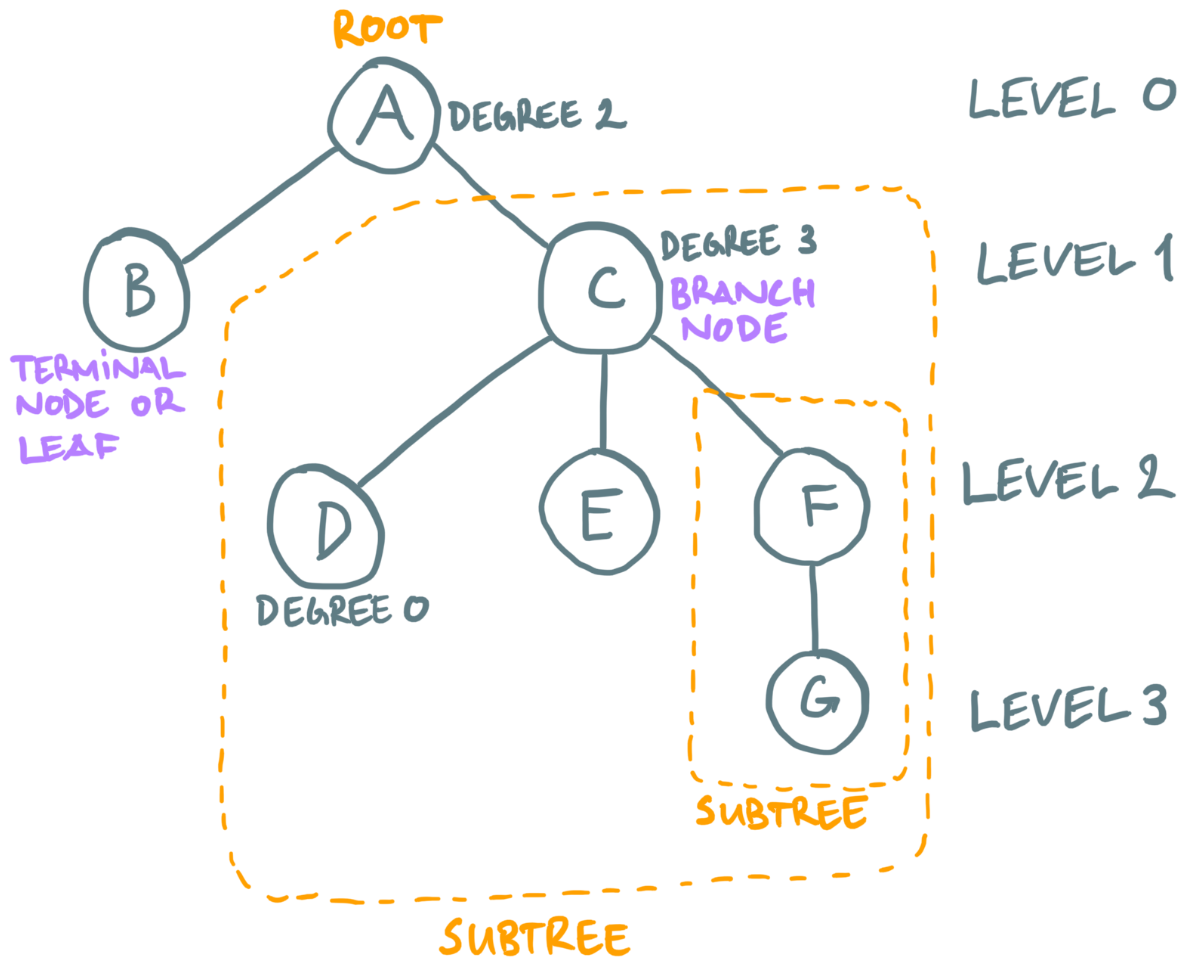
Trees:

- 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*.
- 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:

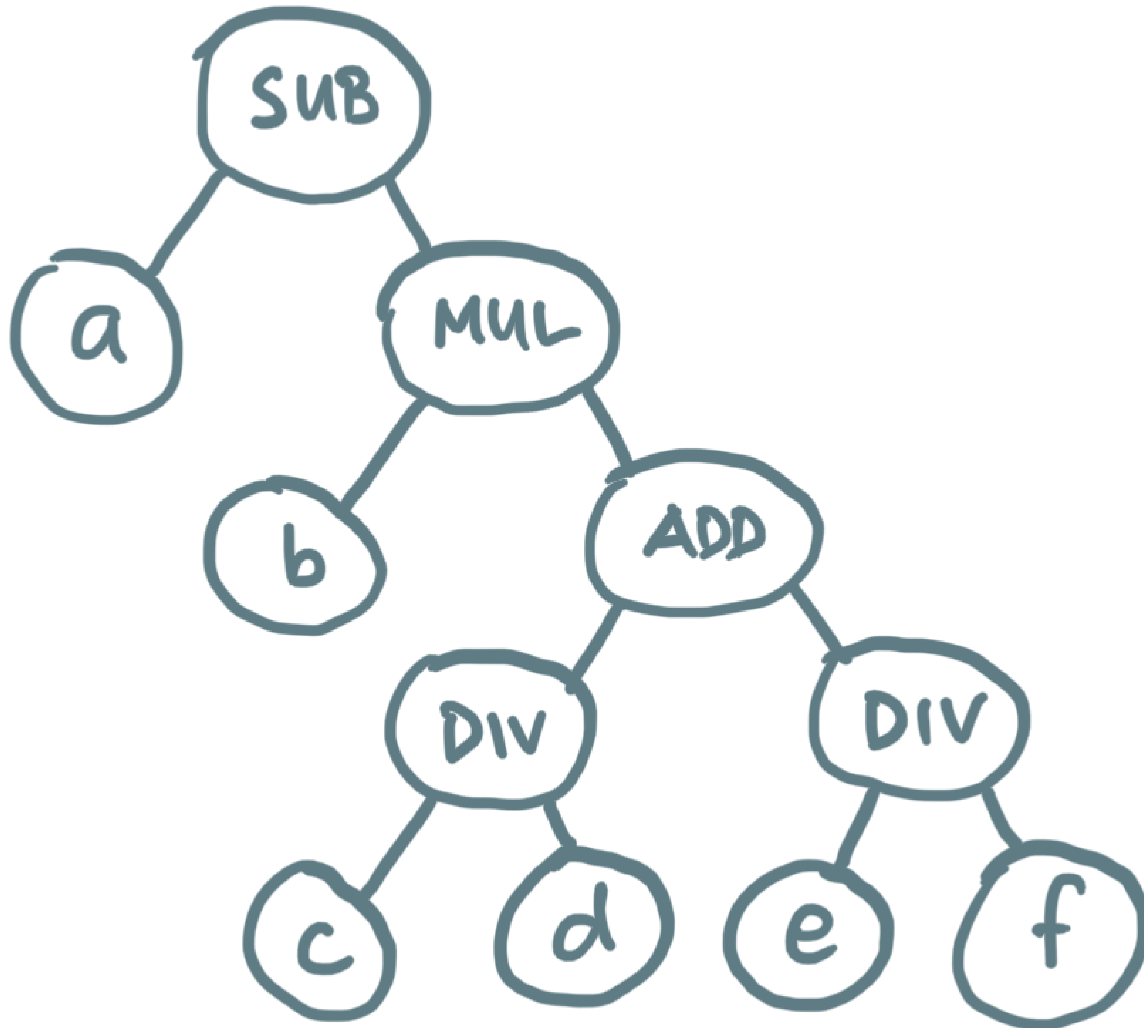
- Parsing of code
- Evolutionary trees in biology
- Unix file system
- Parallel computing (communication patterns)
- Graph theory

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

BINARY TREES

- Binary trees are an important type of tree structure
- In a binary tree, each node has *at most* ***two*** subtrees. What is the highest degree in such a tree?
- 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 TREE EXAMPLE



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 and multiplication or division over addition and subtraction.

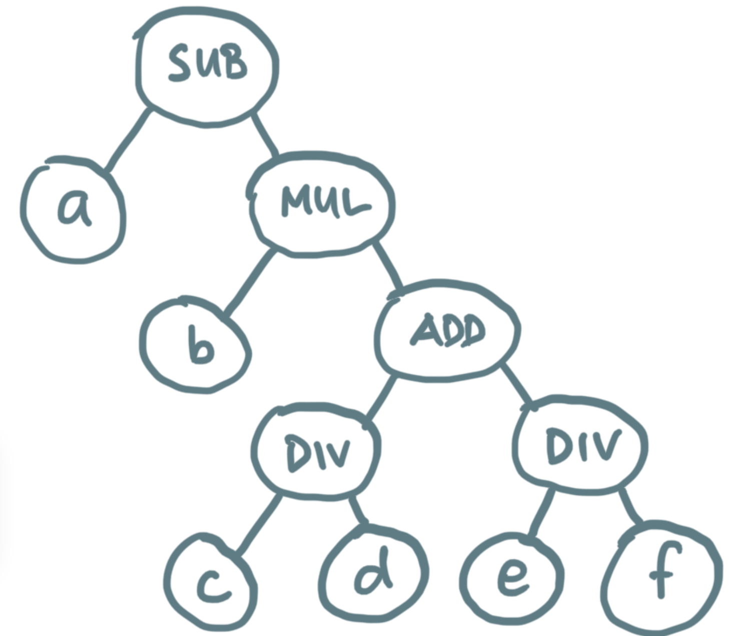
BINARY TREE EXAMPLE

- Let us go through an example code that generates the expression tree on the right.
- We build the tree by exploiting the operator precedence built into `python`. It allows us to build the tree automatically.
- We want to print the expression tree like this:

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

- The recursion for building the tree looks like this:

```
1 >>> tree = TreeNode.__sub__(a,
2                 TreeNode.__mul__(b,
3                     TreeNode.__add__(
4                         TreeNode.__truediv__(c, d),
5                         TreeNode.__truediv__(e, f)
6                     )
7                 )
8             )
```

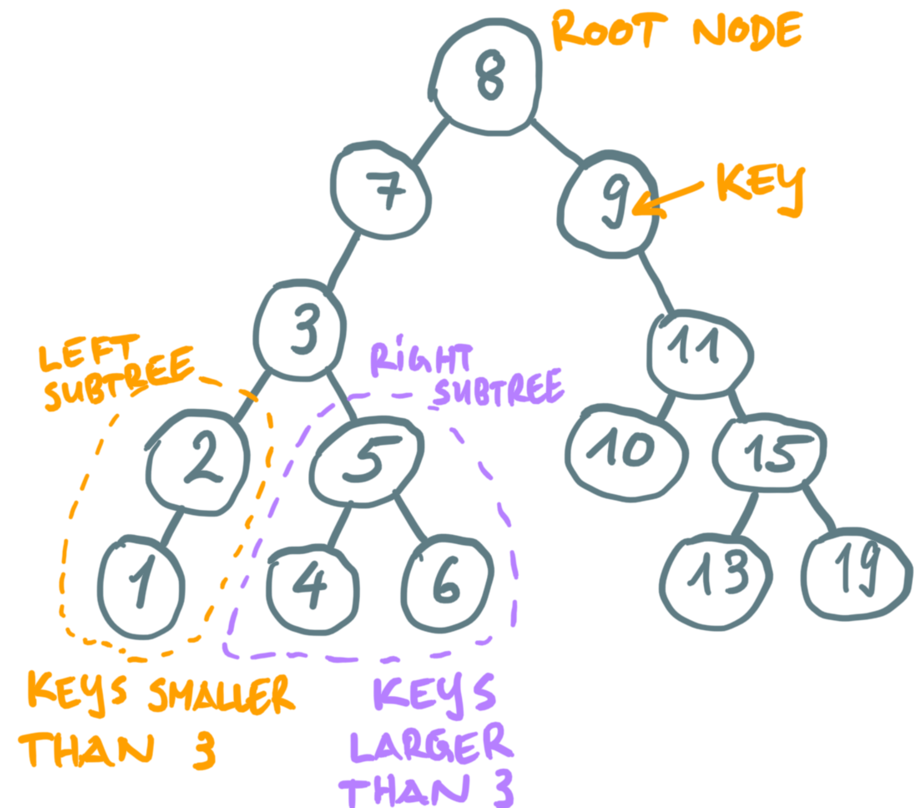


$$a - b\left(\frac{c}{d} + \frac{e}{f}\right)$$

BINARY SEARCH TREE

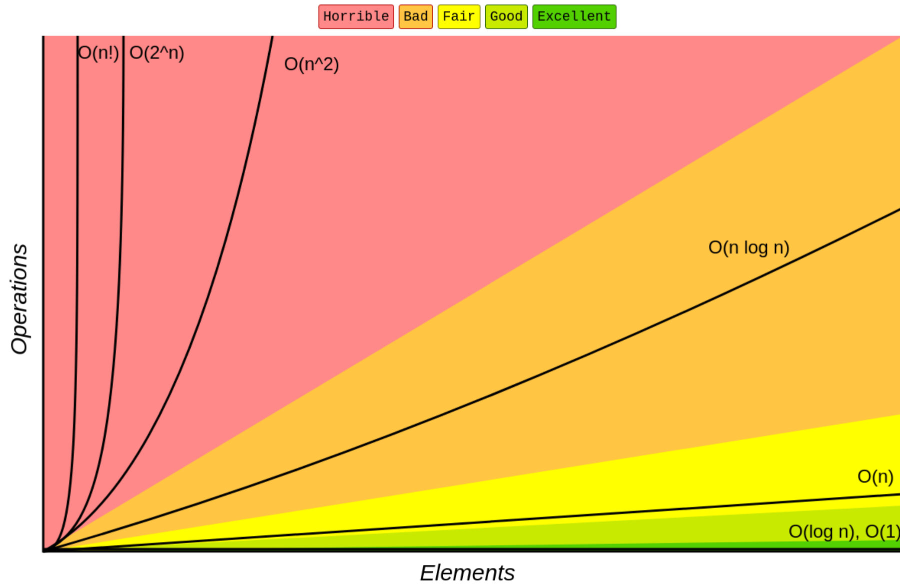
A binary search tree (BST) is an ***ordered*** binary tree with `key` values that can be compared with each other. A BST satisfies the restriction that the `key`'s in any nodes of the ***left*** subtree of a node v are ***smaller*** than the `key` in node v and any `key`'s in the ***right*** subtree are ***larger*** than the key in node 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*.
- We are only concerned with *single* occurrence of `key` values.



BINARY SEARCH TREE: COMPLEXITY

https://www.bigocheatsheet.com/



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)

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

BINARY SEARCH TREE: SEARCH

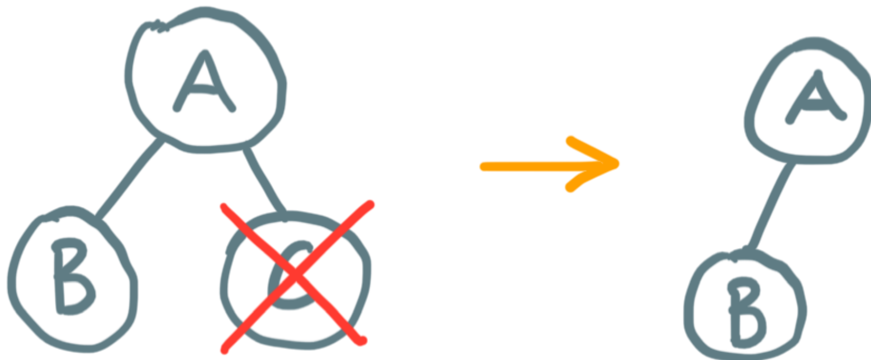
- Searching a BST is a recursive algorithm.
- If we have a search *hit*, we return the associated value.
- If we have a search *miss* we return `NULL`, e.g., `None` in `python`.
- ***Algorithm:*** start at the root node and compare the search value with the `key` of the node.
    1. If the search value 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 hit, return `NULL`.
- Node insertion is almost identical to a tree search.

BINARY SEARCH TREE: NODE DELETION

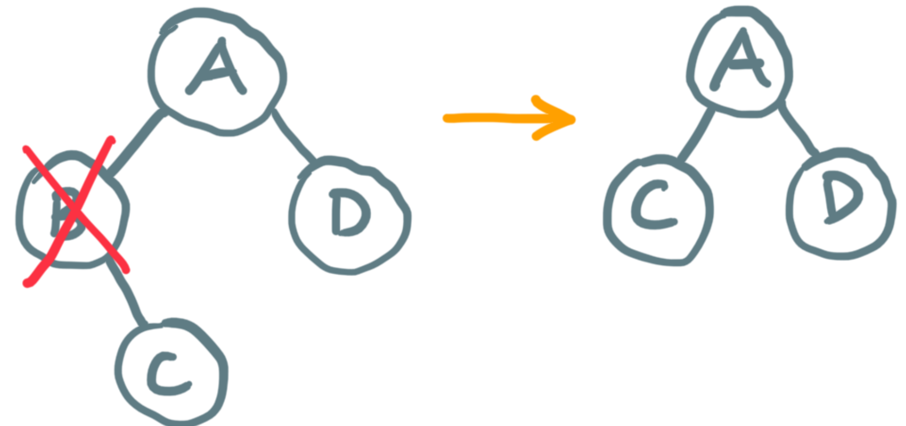
Node deletion is more difficult to implement:

- If the node to be deleted has no children, we can just remove it.
- 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:



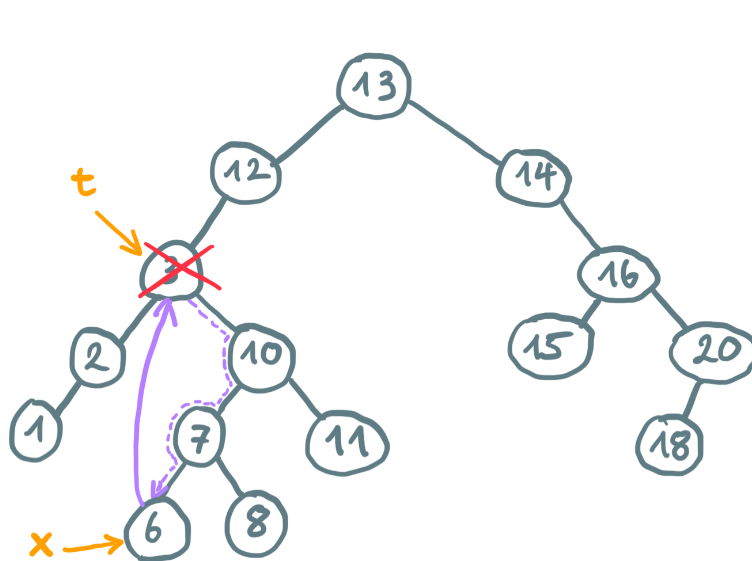
Node removal with single child:

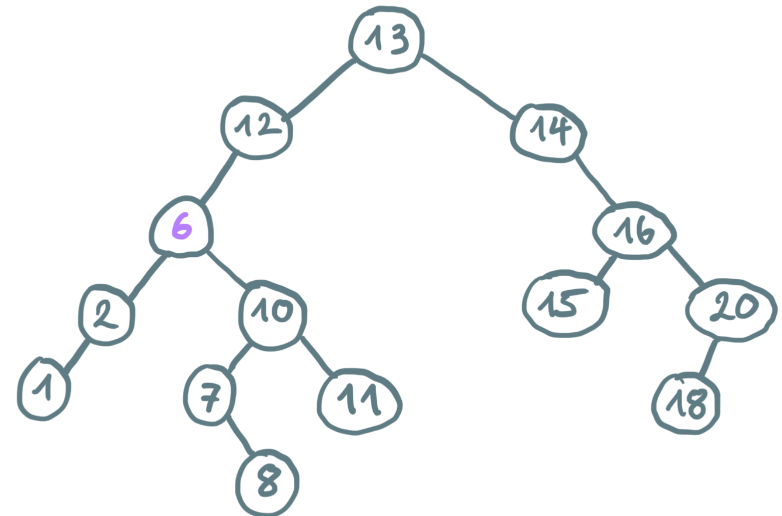


BINARY SEARCH TREE: NODE DELETION

Node deletion is more difficult to implement: node with two children

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





How does the tree on the right look like after we delete node `6` ?

RECAP

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