MA3 introduction

Tom Smedsaas

This module is about *data structures*, mostly *linked lists* and *trees*.

We will also discuss some new features:

- Operator overloading
- Iterators
- Generators

However this short lecture is just a repetition of what classes and objects are.

Everything in Python is an object

Examples:

- Numbers
- Strings
- Lists
- Dictionaries
- Functions
- Modules
- and more …

Exceptions: `+`, `<`, `in`, `not`, `and`, `for`, `if`, `[`, `)`, …

Objects have properties

The function `dir` can be used to see the properties. Example:

```
>>> dir(4.6)
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getformat__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__set_format__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real’]
>>> 4.6.__round__()
5
>>> round(4.6)
5
```

Python list properties

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort’]
```

Here we can see methods that we actually use, e.g. `append`, `reverse` and `sort`.

We can also see methods defining some list operators e.g. `__add__` for +, `__eq__` for `==` and `__le__` for `<=`.

These are called *dunder, magic* or *special methods.*

How create new types of objects?

Write a class!

Example from the traffic simulation in Prog 1:

```
class Vehicle:
    def __init__(self, destination, borntime):
        self.destination = destination
        self.borntime = borntime

    def __str__(self):
        return f'Vehicle({self.destination}, {self.borntime})'
```

Class name

Initializer or *constructor*

String representation

Usage:

```
car1 = Vehicle('Uppsala', 25)
car2 = Vehicle('Stockholm', 47)
print(car1)
print(car2)
print(car1.destination)
print(car2.destination)
print(car2.borntime)
```

Output:

```
Vehicle(Uppsala, 25)
Vehicle(Stockholm, 47)
Uppsala
Stockholm
47
```

Another example from the traffic simulation

```
class Light:

    def __init__(self, period, green_period):
        self._period = period
        self._green_period = green_period
        self._time = 0

    def is_green(self):
        return self._time < self._green_period

    def __str__(self):
        if self.is_green():
            return "(G)"
        else:
            return "(R)"

    def step(self):
        self._time = (self._time+1) % self._period
```

The constructor.
Three *instance variables*

Predicate: `True` or `False`

Special method for converting to string

Time stepping

Usage of the `Light` class

Code:

```
s1 = Light(5,2)
s2 = Light(7,3)
s3 = s2
for i in range(8):
    print(i, s1, s2, s3)
    s1.step()
    s2.step()
```

Output:

```
0 (G) (G) (G)
1 (G) (G) (G)
2 (R) (G) (G)
3 (R) (R) (R)
4 (R) (R) (R)
5 (G) (R) (R)
6 (G) (R) (R)
7 (R) (G) (G)
```

Link to the traffic simulation lesson in English and in Swedish

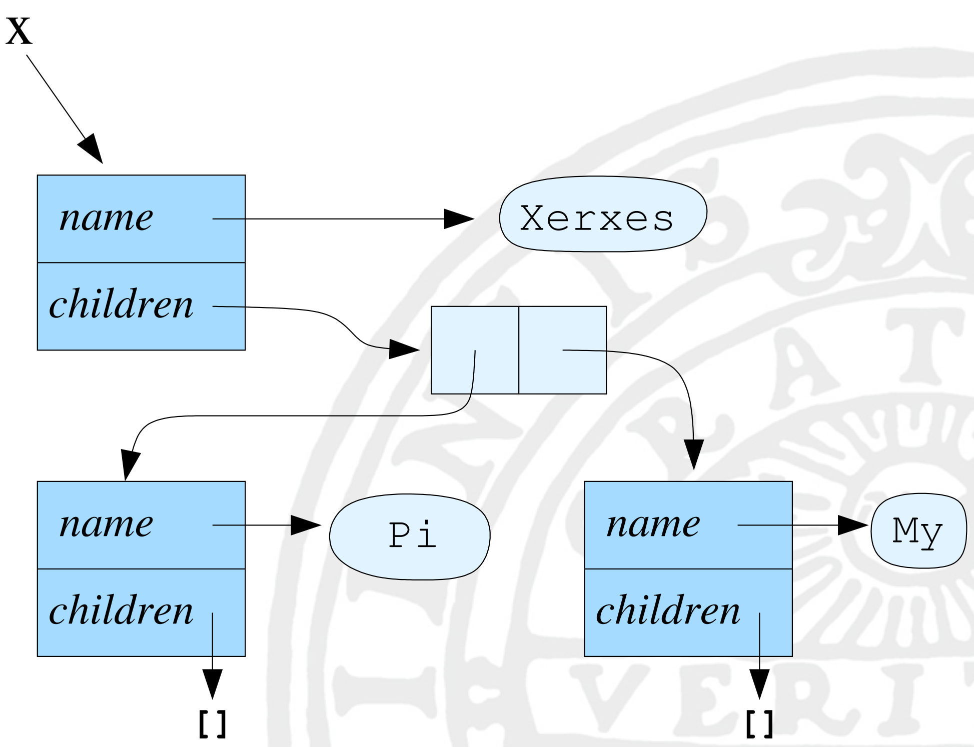
Another example

```
class Person:
    def __init__(self, name):
        self.name = name
        self.children = []

    def __str__(self):
        return f"{self.name} : {str([s.name for s in self.children])}"

    def add_child(self, child):
        self.children.append(child)
```

Usage of the class `Person`

Code:

```
x = Person("Xerxes")
x.add_child(Person("Pi"))
x.add_child(Person("My"))
print(x)
```



Output:

```
Xerxes : ['Pi', 'My']
```

Summary

- Classes are used to define new types of objects
- A class contains methods and data attributes
- The `__init__` method is used to initilize an object
- The method definitions must have `self` as the first parameter
- The method `__str__` is used to define a string representation of the object
- Other special methods can define other operations on the objects (`__lt__`, `__eq__`, …)

You will see more examples in the coming material.

The end

MA3: Linked Lists

Tom Smedsaas

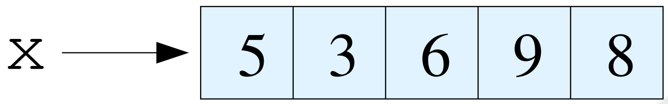
In this lecture we will discuss basic techniques for handling *linked lists*.
It covers the first 11 pages in the MA3.pdf document i.e. up to but not including "Iterators and generators".

Python lists

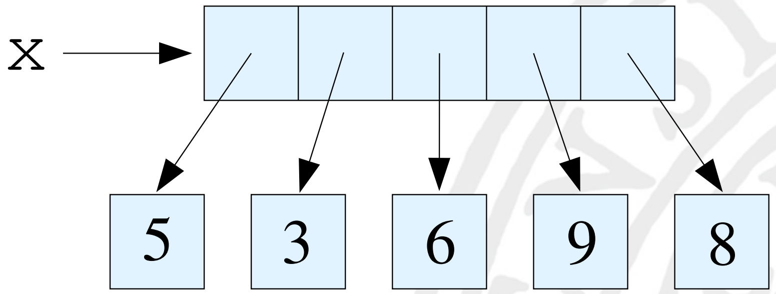
Lists are the most fundamental structure in Python.

We can, for example, write `x = [5, 3, 6, 9, 8]`

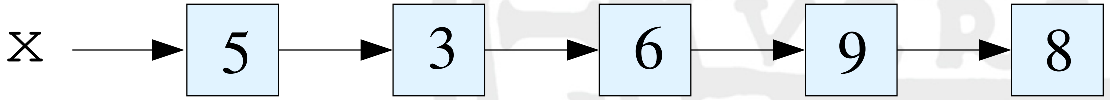
which can be illustrated:



Remember that it really is more like this:



We are now going to make a *linked list*:



Why?

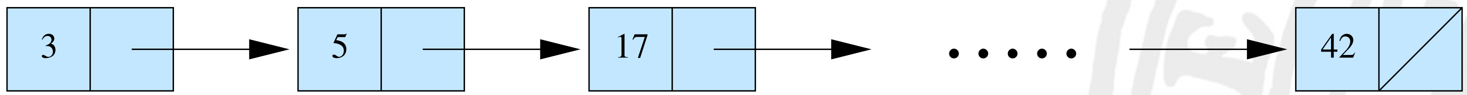
Why are we making an alternative to the Python `list`?

- Study basic techniques for handling linked structures.
- Study *iterators*, *generators* and *operator overloading*.
- See more examples of recursive methods.
- Practice algorithm analysis.
- Discover that there are situations where these lists are more efficient than the built in lists.

A linked list

- We shall create a class `LinkedList` that can store a number of data items.
- The data shall be stored in *increasing order* so they must be comparable.
- We will use integers in our examples.

A linked list can be illustrated like this:

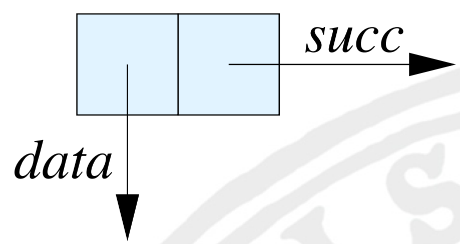


Each element has a reference to its follower.

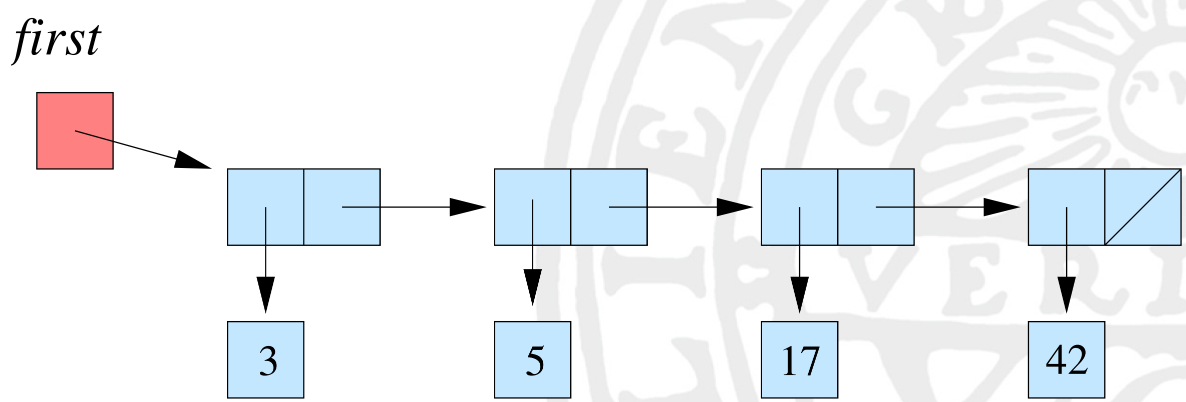
Python representation

We define a class for the nodes in the list:

```
class Node:
    def __init__(self, data, succ):
        self.data = data
        self.succ = succ
```



We have to keep track of the first node:



A class for linked lists

We would like to be able to write like this:

```
ll = LinkedList()
print(ll)
for x in [5, 2, 3, 7]:
    ll.insert(x)
    print(ll)
```

```
()
(5)
(2, 5)
(2, 3, 5)
(2, 3, 5, 7)
```

Sketch:

```
class LinkedList:
    def __init__(self):
        pass

    def __str__(self):
        pass

    def insert(self, data):
        pass
```

We will also write the methods `remove_first` and `get_last` as examples of simple methods.

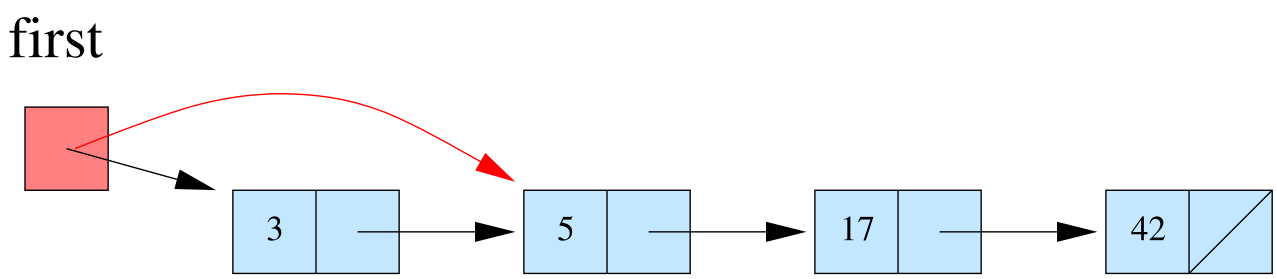
The LinkedList class

```
class LinkedList:

    class Node:
        def __init__(self, data, succ):
            self.data = data
            self.succ = succ


    def __init__(self):
        self.first = None

    . . .
```

The `Node` class is inside the `LinkedList` class

Creates an empty list

The remove_first method




```python
def remove_first(self):
    '''Removes the first element and returns its value'''
    result = self.first.data
    self.first = self.first.succ
    return result
```

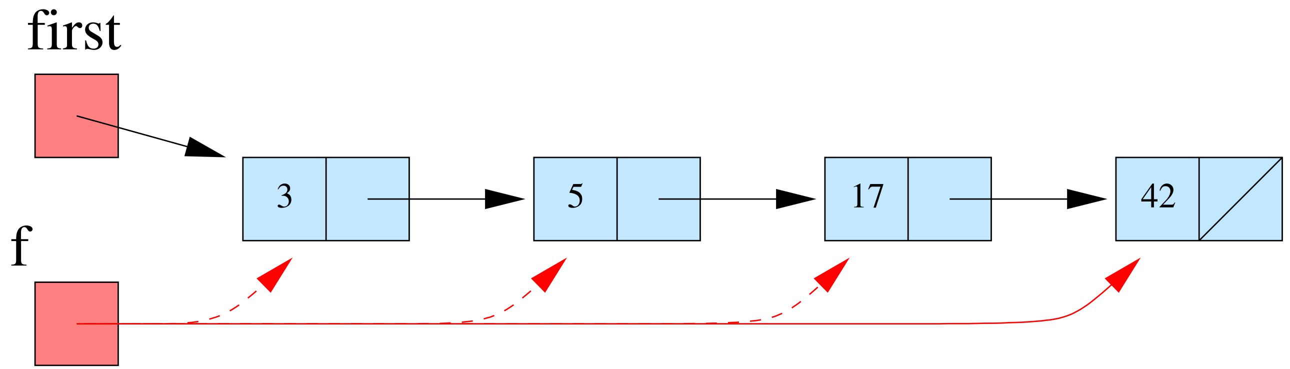
There is actually a problem in the code. What is that?

What happens if the list is empty?

Better `remove_first`

```
def remove_first(self):
    '''Removes the first element and returns its value'''
    if self.first == None:
        return None                 # Or raise an exception
    result = self.first.data
    self.first = self.first.succ
    return result
```

The `get_last` method



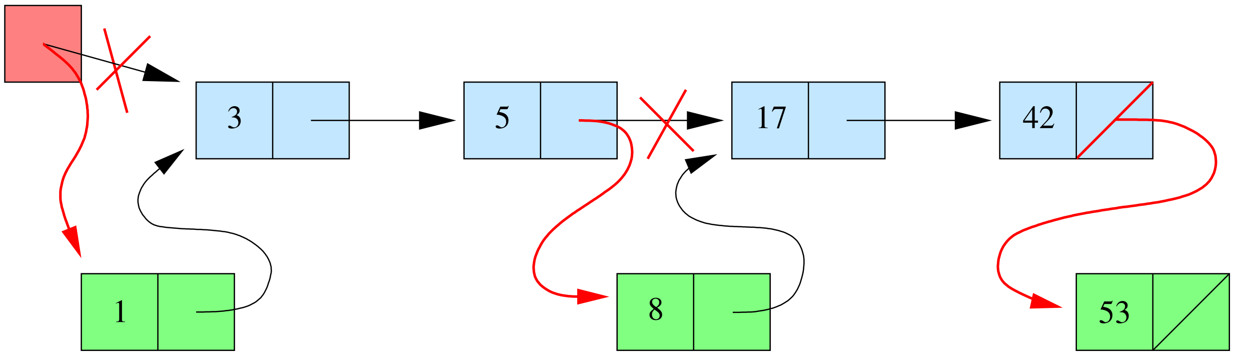

```python
def get_last(self):
    '''Returns the last stored data item'''
    if self.first == None:
        return None                # Or raise an exception
    f = self.first
    while f.succ:
        f = f.succ
    return f.data
```

The `__str__` method

```
def __str__(self):
    result = ''
    f = self.first
    while f:
        result += str(f.data)
        f = f.succ
        if f:
            result += ', '
    return '(' + result + ')'
```

Comma *between* elements

An *iterative* `insert` method




```python
def insert(self, x):
    if self.first is None or x < self.first.data:
        self.first = self.Node(x, self.first)
    else:
        f = self.first
        while f.succ and x >= f.succ.data:
            f = f.succ
        f.succ = self.Node(x, f.succ)
```

A *recursive* insert: method 1.

In the Node class:

```
def insert(self, x):
    if self.succ is None or x < self.succ.data:
        self.succ = self.Node(x, self.succ)
    else:
        self.succ.insert(x)
```

and the LinkedList class:

```
def insert(self, x):
    if self.first is None or x < self.first.data:
        self.first = self.Node(x, self.first)
    else:
        self.first.insert(x)
```

A *recursive* insert: method 2.

With a *recursive help method* in `LinkedList`

```
def insert(self, x):
    self.first = self._insert(x, self.first)

def _insert(self, x, f):
    if f is None or x < f.data:
        return self.Node(x, f)
    else:
        f.succ = self._insert(x, f.succ)
        return f
```

The end

MA3: Some Python facilities

This lecture discusses

- *iterators*,
- *generators* and
- *operator overloading*

applied to the `LinkedList` class

A common pattern

Suppose we want to

- sum all values in our linked list or
- compute the mean and standard deviation of the values in the list or
- find the first prime number in the list or
- …

For ordinary lists this could be done with a for-statement and we would like to be able to it for our linked list also

A for-loop for the linked lists

```
ll = LinkedList()
. . .  # Build up the list

sum = 0
n = 0
for x in ll:
    sum += x
    n += 1
print(f'The mean value is {sum/n}')
```

This code is written *without any knowledge of the internal structure* of `LinkedList` class!

Make the list *iterable*!

Add two special methods:

```
def __iter__(self):
    self.current = self.first
    return self

def __next__(self):
    if self.current:
        result = self.current.data
        self.current = self.current.succ
        return result
    else:
        raise StopIteration
```

Now we can iterate over our lists with a `for` statement

An easier way

We can write the `__iter__` method as a *generator* :

```
def __iter__(self):
    current = self.first
    while current:
        yield current.data
        current = current.succ
```

Note:

- The `yield` statement
- No `__next__` method
- `current` is a local variable

Operator overloading

By operator overloading we mean to give existing operators like +, ==, <=, … a meaning and definition for new data types.

For an ordinary list we can use the operator `in` for example in an expression like

```
if w in lista:
    print('Oui!')
else:
    print ('Non!')
```

We can get this to work by implementing the `__in__` method in our `LinkedList` class.

In the `LinkedList` class

```
def __in__(self, x):
    for d in self:          # Use generator/iterator
        if x == d:
            return True
        elif x < d:         # No point in searching more
            return False
    return False
```

Another example: indexing

```
def __getitem__(self, index):
    i = 0
    for x in self:
        if i == index:
            return x
        i += 1
    raise IndexError(f'LinkedList index {index} out of range’)
```

Now we can write code like:

```
print(ll[0] + ll[2])
```

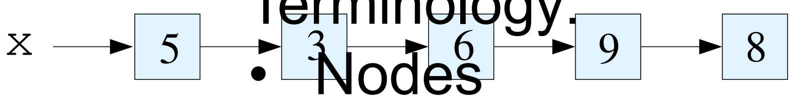
but **not**:

```
ll[3] = ll[0] + ll[2])
```

The end

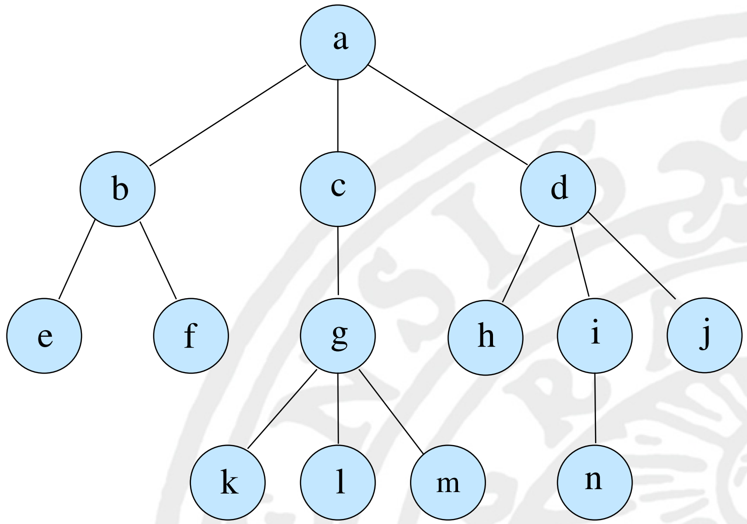
MA3: Tree structures

Tom Smedsaas



Terminology:

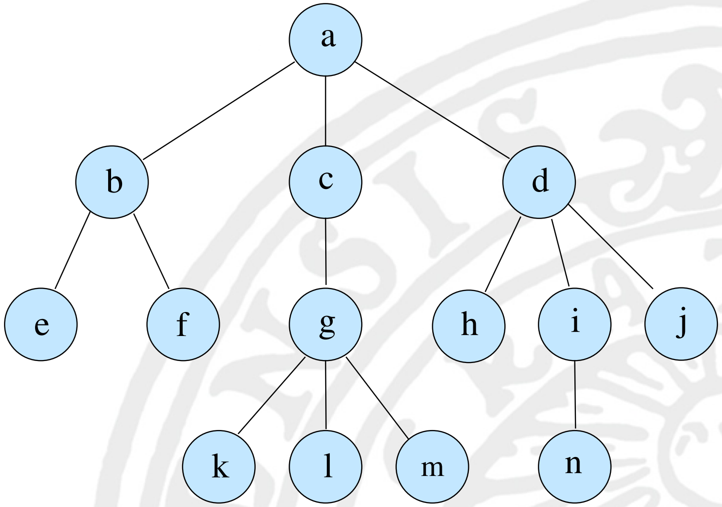
- Nodes
- Root
- Leefs
- Children
- Siblings
- Height
- Size



Operations in trees

Operations on trees:

- Enter new nodes
- Remove nodes
- Traverse the tree
- Search a node with a particular contents
- Search a node in a particular position
- Merge trees
- Measurements:
  - Height
  - Size
  - Path length

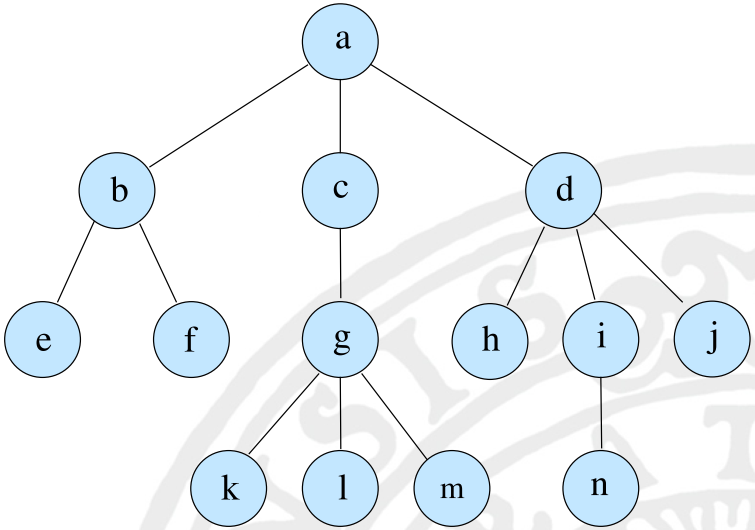


Tree traversals

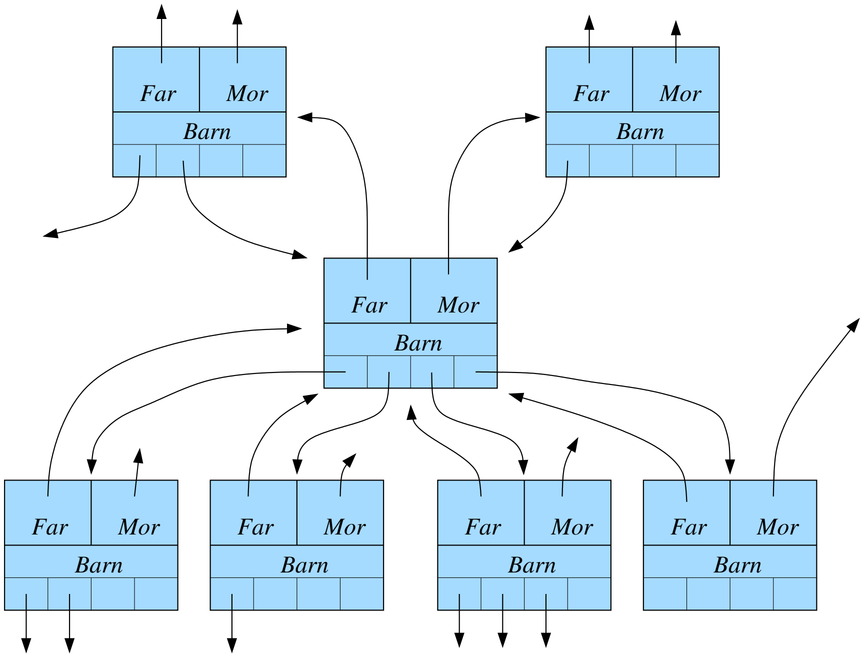
Preorder: a b e f c g k l m d h i n j

Postorder: e f b k l m g c h n i j d a

Level order: a b c d e f g h i j k l m n



More complicated structures




```
class Person:
    def __init__(self, name,
                 mother=None,
                 father=None):
        self.name = name
        self.mother = mother
        self.father = father
        self.children = []

    def add_child(self, child):,
        self.children.append(child)
```

The end

MA3: Binary trees

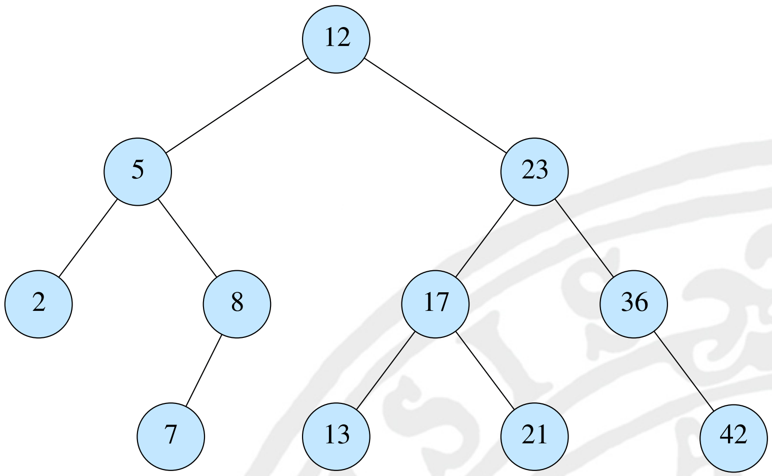
Tom Smedsaas

Binary trees

Definition:

A set of nodes that is either empty or consists of three three disjoint sets:

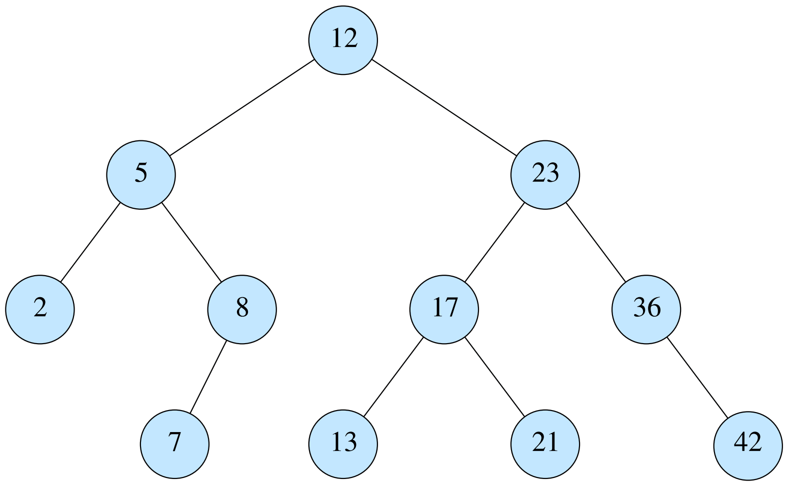
- One with one node called *the root*
- One called *the left subtree* which is a binary tree
- One called *the right subtree* which is a binary tree



Inorder traversal: 2, 5, 7, 8, 12, 13,17, 21, 23, 36, 42

Binary *search* trees

A *binary search tree* is a binary tree where the nodes are ordered:



The data in every node is greater than all data in its left subtree and less than all data in its right subtree.

Binary search trees

This is a powerful structure for storing data with an order relationship.

The operations

- searching data
- inserting
- removing data

can be done in $\Theta(\log n)$ time on the average.

We will here look at these algorithms.

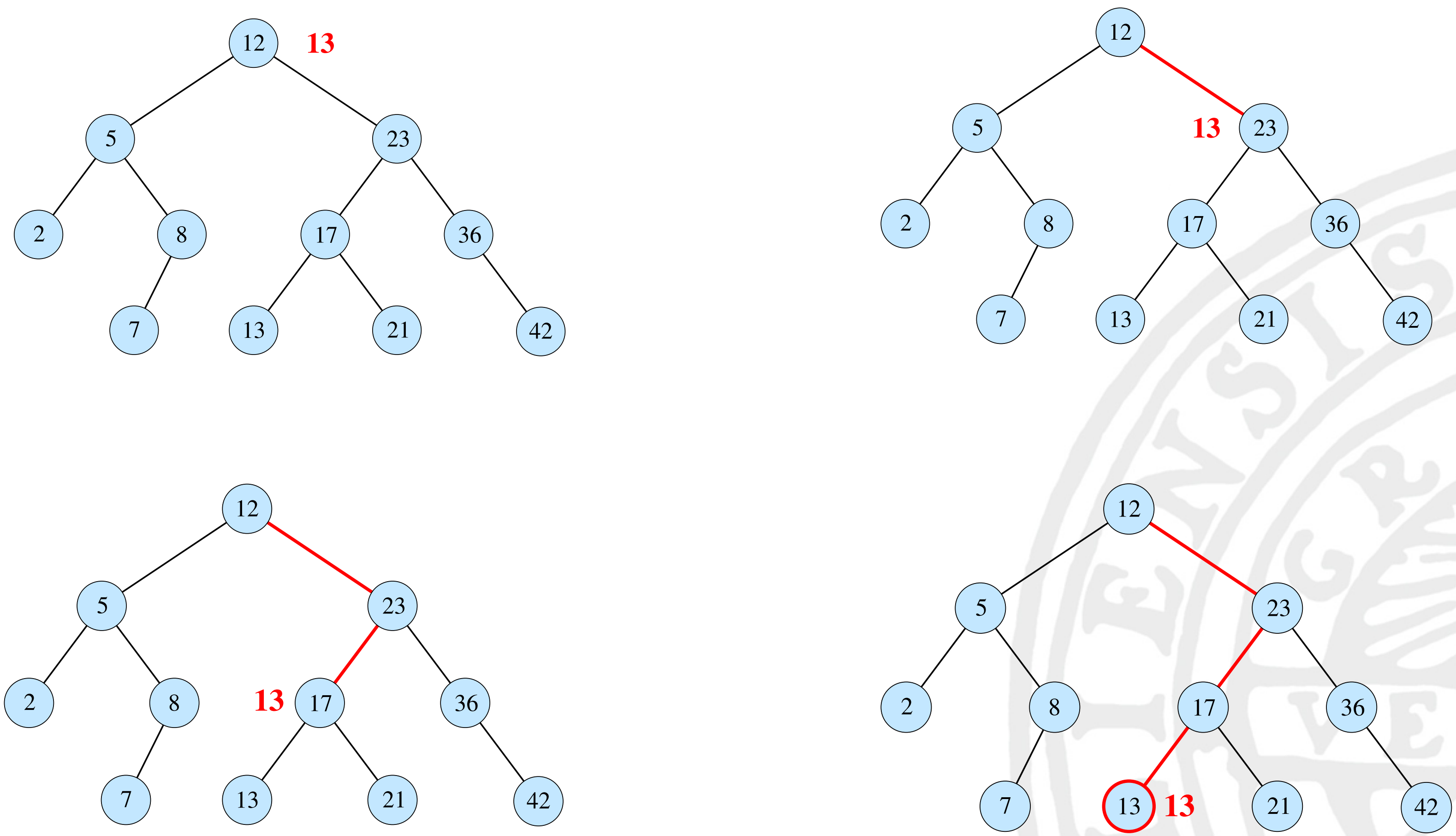
A class for binary search trees

```
class BST:
    class Node:
        def __init__(self, key,
                         left = None,
                         right = None):
            self.key = key
            self.left = left
            self.right = right

    def __init__(self, root = None)
        self.root = root

     . . . .
```

UPPSALA UNIVERSITET

Searching

Searching starts at the root and is guided by the values in the nodes:



A method for searching

```
def contains(self, k):
    n = self.root
    while n and n.key != k:
        if k < n.key:
            n = n.left
        else:
            n = n.right
    return n
```

Traversal: Counting nodes

Use a recursive help method:

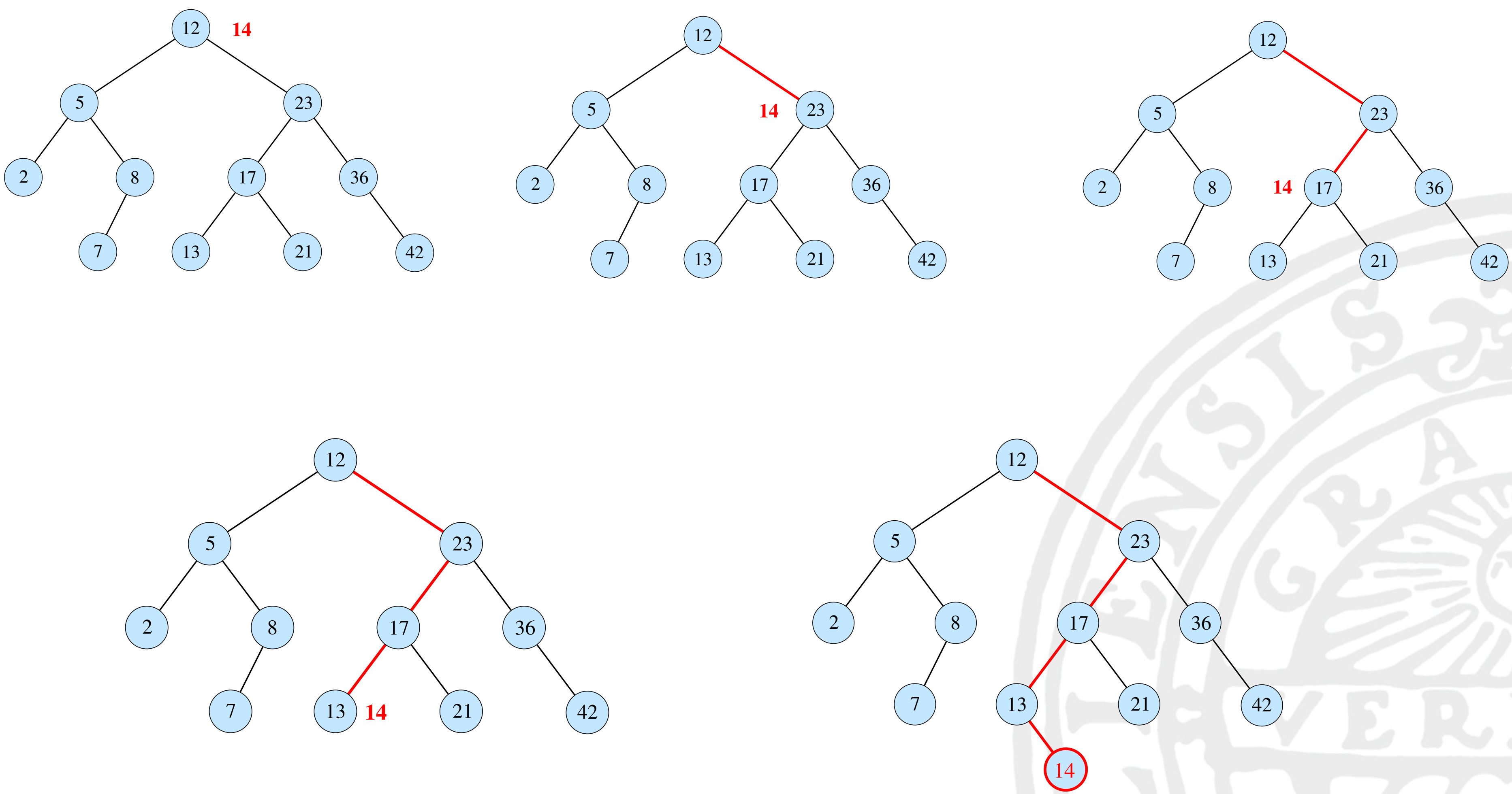
```
def size(self):
    return self._size(self.root)

def _size(self, r):
    if r:
        return 1 + self._size(r.left) + self._size(r.right)
    else:
        return 0
```

Note that `r == None` is a much better base case than `r.left == None` and `r.right == None`.

We will later see how this could be done using a *generator*.

UPPSALA UNIVERSITET

Inserting in binary search tree



Code

```
def insert(self, key):
    self.root = self._insert(self.root, key)

def _insert(self, r, key):
    if r is None:
        return self.Node(key)
    elif key < r.key:
        r. left = self._insert(r.left, key)     # Insert in the left subtree
    elif key > r.key:
        r.right = self._insert(r.right, key)   # Insert in the right subtree
    else:
            pass
    return r
```

Modify the (sub-)tree with root in `r` *and return the root of the modified (sub-)tree.*

Note that the help method always have to return the root of the modified subtree regardless if it is the new node or not.

Modified insertion code

Suppose we want to know if a new node was inserted or not.

```
def insert(self, key):
    self.root = self._insert(self.root, key)

def _insert(self, r, key):

    if r is None:
           return self.Node(key)

    elfi key < r.key:
        r. left = self._insert(r.left, key)    # Insert in the left subtree
    elif key > r.key:
        r.right = self._insert(r.right, key)   # Insert in the right subtree
    else:
           pass

    return r
```

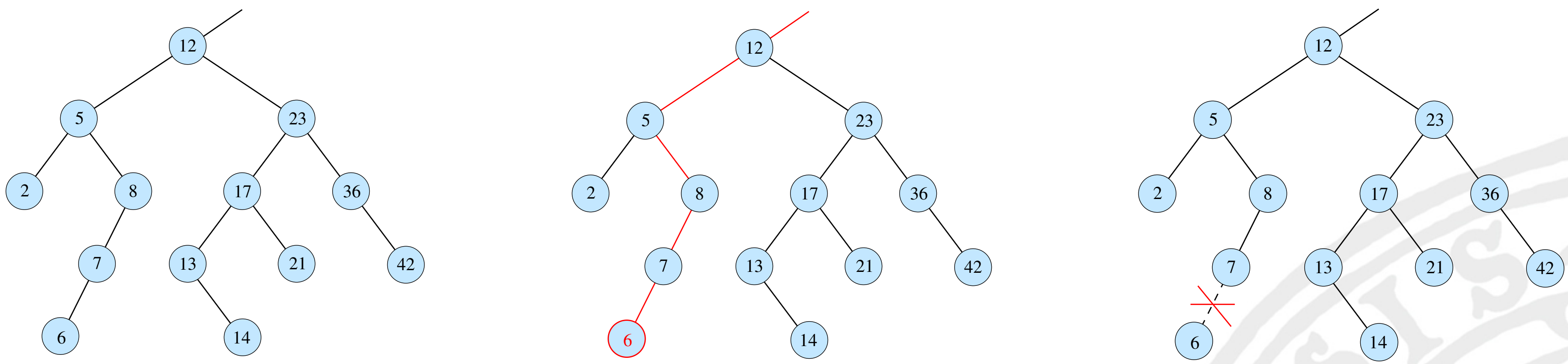
UPPSALA UNIVERSITET

Modified insertion code

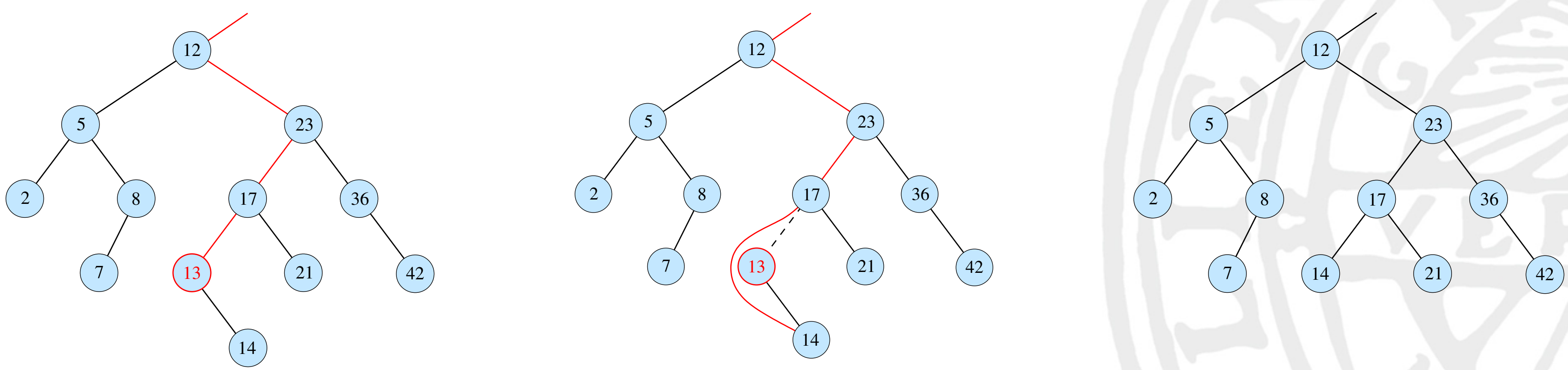
```
def insert(self, key):
        self.root, result = self._insert(self.root, key)
        return result

    def _insert(self, r, key):
        if r is None:
            return self.Node(key), True
        elif key < r.key:
            r.left, result = self._insert(r.left, key)
        elif key > r.key:
            r.right, result = self._insert(r.right, key)
        else:
            result = False # Already there
        return r, result
```

Remove

If the node to be removed has no children. Example: remove 6.

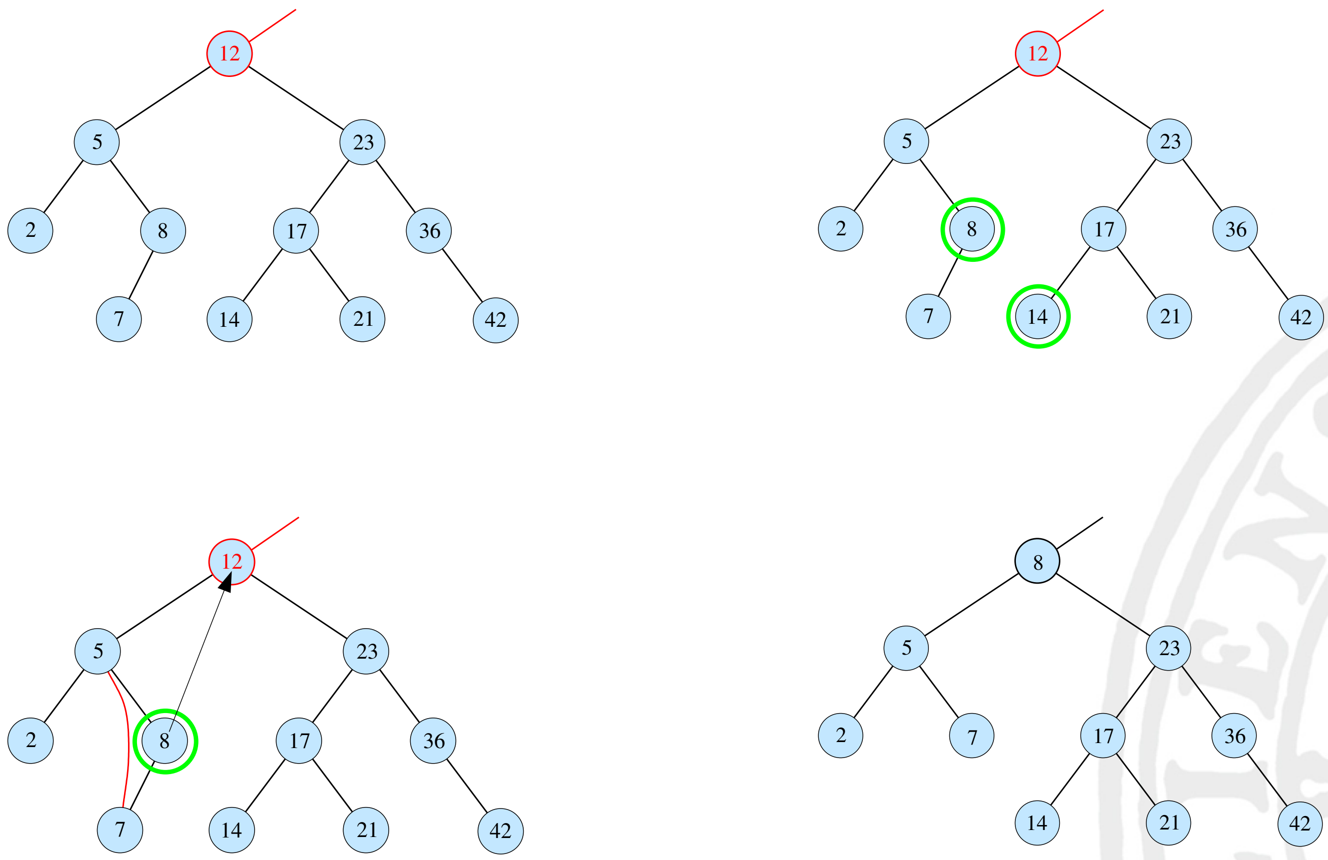


If the node to be removed has one child. Example: remove 13.

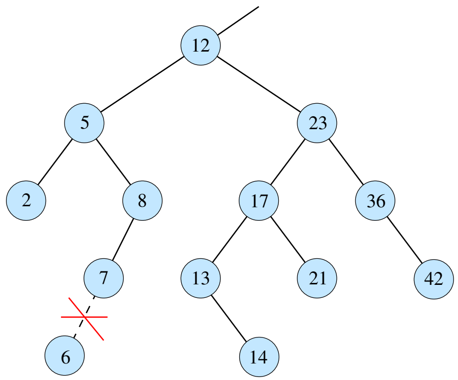


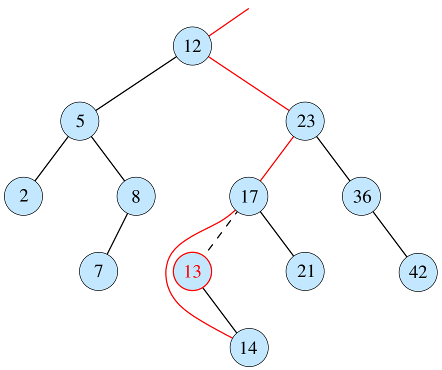
Remove – the harder case

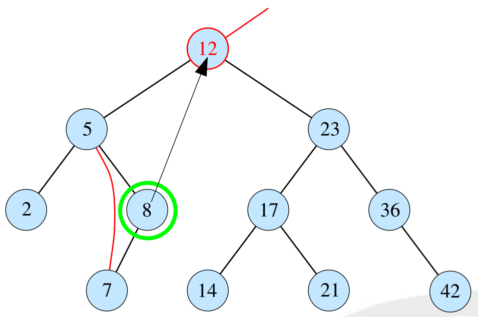
Removing a key in a node with two children. Example: remove 12.



Coding hints for remove







```python
def _remove(self, r, key):
    pass
```

Takes a reference to a node (`r`) that is the root in a subtree. Removes `key` from that subtree and returns a reference to the root in the resulting subtree.

```python
def _get_largest(self, r):
    pass
```

Find the largest and then then call `_remove` recursively

```python
def _remove_largest(self, r):
    pass
```

Removes the node with the largest key and returns a tuple with that key and a reference to the root in the resulting subtree.

UPPSALA UNIVERSITET

The end

MA3: Tree Generators

Tom Smedsaas

This lecture discusses
generators for tree structures

The LinkedList generator

We noticed the the easiest way to to write the __iter__ function in the LinkedList class was to do it as an generator:

```
    def __iter__(self):
        current = self.first
        while current:
            result = current.data
            yield result
            current = current.succ
```

With this construction we could iterate over a list wit the code:

```
ll = LinkedList()
...
for n in ll:
    do_something(n)
```

The for statement will use the generator to access the elements in the list.

Tree generator

The situation in the `BST` class is more complicated since we have no easy way to say what the next element is.

However, if we write a generator in the `Node` class, we can iterate over the nodes in the left and right subtree by using the generators in the two root nodes there.

BST-classes

```
class BST:

    class Node:
        def __init__(self, key,
                     left = None,
                     right = None):
            self.key = key
            self.left = left
            self.right = right


    def __init__(self, root = None):
        self.root = root

        . . . .
```

Generator for BST

```
class BST:
    class Node:
        def __init__(. . .): . . .

        def __iter__(self):
            if self.left:
                for key in self.left:
                    yield key
            yield self.key
            if self.right:
                for key in self.right
                  yield key

    def __init__(. . .): . . .

    def __iter__(self):
        if self.root:
            for key in self.root:
                yield key
```

Using yield from

The code can be a little simplified by using the `yield from` construction:

```
def __iter__(self):                    # In the Node class
    if self.left:
        yield from self.left:
    yield self.key
    if self.right:
        yield from self.right


def __iter__(self):                        # In the BST class
    if self.root:
        yield from self.root:
```

Note how easily the generator can be changed to do other traversals!

The end

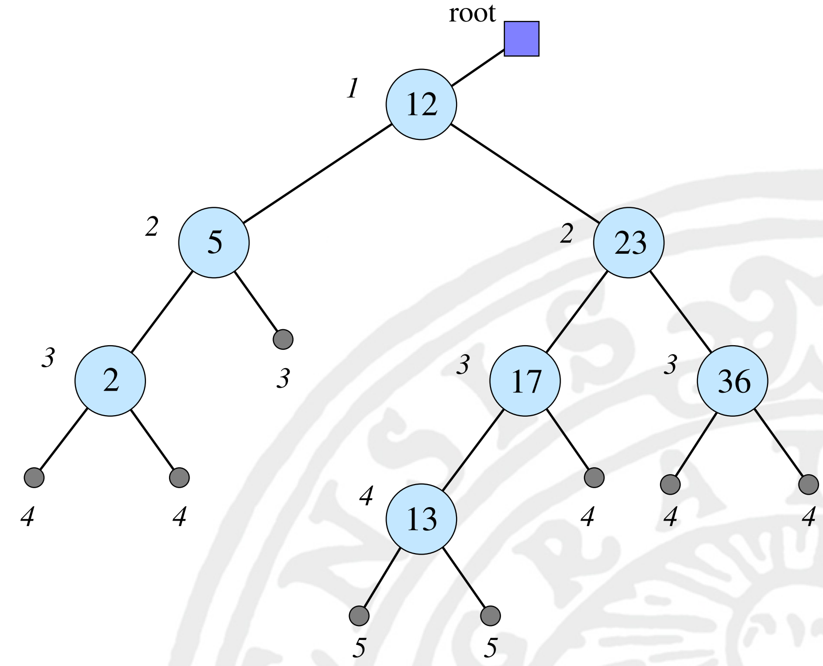
MA3: Properties of binary trees

Tom Smedsaas

This lecture discuss properties of binary trees and their implications for binary search trees.

Measurements on trees

- Size (n): 7
- Height (h): 4
- Internal path length (i or ipl): $1 + 2 + 2 + 3 + 3 + 3 + 4 = 18$
- External path length (e or epl): $4 + 4 + 3 + 5 + 5 + 4 + 4 + 4 = 33$



The two path length properties are a measurements of how well balanced the tree is. They are also linked by the relation $e = i + 2n + 1$

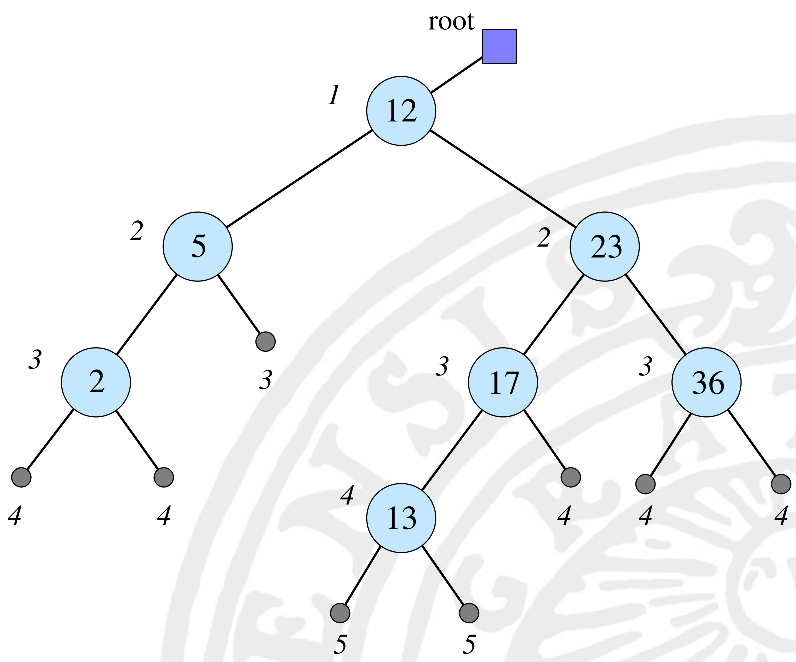
What does these numbers mean?

Searching, *removing* and *inserting* are all O(h) operations.

What about the average?

A *successful* search in a given tree will, *on the average*, require: $\frac{i}{n}$ tries which, in this case, is $\frac{18}{7} = 2.57$.

An *unsuccessful* search in a given tree will, *on the average*, require: $\frac{e}{n+1}$ tries which, in this case, is $\frac{33}{7+1} = 4.12$.



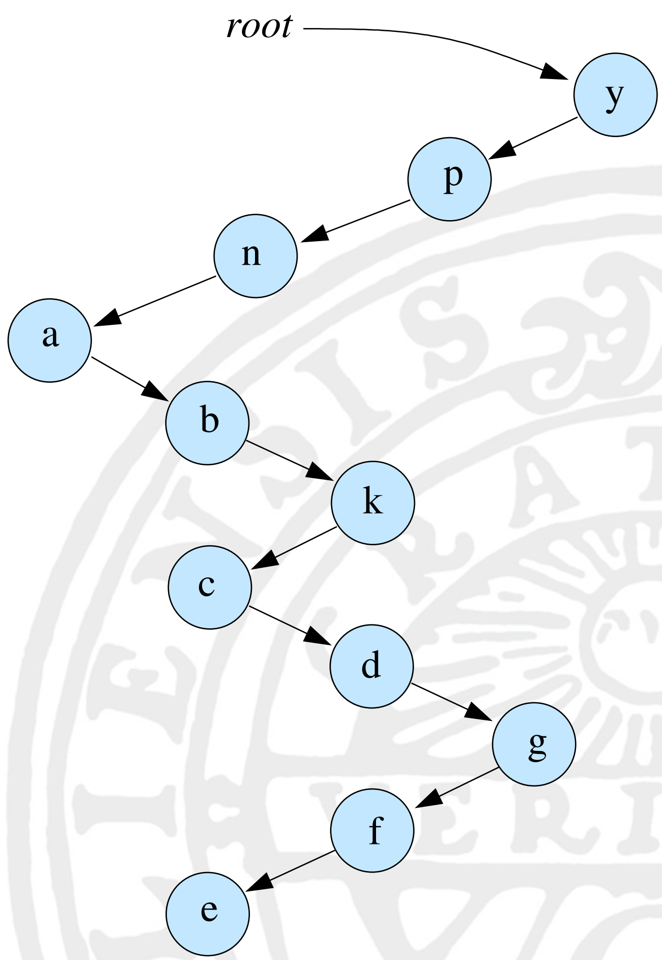
Maximal values?

This tree will give maximal values:

the height is n

$$i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

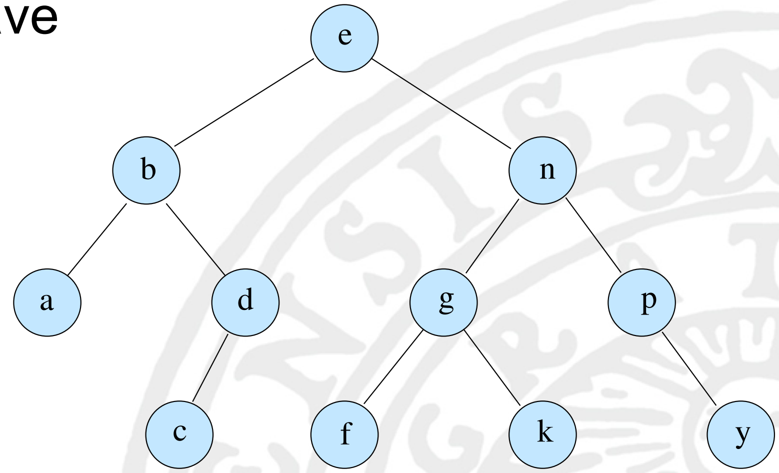
which makes the operations $\Theta(n)$ on the average.



Minimal values

As well branched as possible.

If we have h completely filled levels we will have
$n = 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$
giving that h = $\log_2(n+1)$



Thus, searching, inserting and and removing keys are all done $O(\log(n))$

Average search tree

What is an average tree?

Suppose we have *n* different keys. There are *n*! possible permutations these keys. If we use each of these permutations to build a tree we can see what the average values of the height and the path lengths are.

It can be shown the the average internal path length over all these trees is

$$1.39 \cdot n \log_2 n + O(n)$$

Thus, for example, searching for a key in a tree with 1000000 keys require, on the average on the average tree $1.39 \cdot \log_2 10^6 \approx 28$ tries.

Thus

The search, insert and remove operations of a key in the ”average” binary search tree with n keys require

$$\frac{1.39n\log_2 n + O(n)}{n} = 1.39\log_2 n + O(1)$$

node visits on the average.

The average case in the worst tree is n node visits. However, there are several insertion algorithms that keep the tree well balanced (AVL-trees, RB-trees …)

The end