UPPSALA UNIVERSITET

MA2 introduction

Courtesy of Tom Smedsaas

Senior Lecturer/Associate Professor at Department of Information Technology

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

We will also discuss some new features:

- Operator overloading
- Iterators
- Generators

UPPSALA UNIVERSITET

Sorted lists

- In the previous module, we discussed that searching in a list is faster if the list is sorted
- We also discussed sorting algorithms and their complexity: merge sorting is more efficient than insertion sorting

UPPSALA UNIVERSITET

Sorted lists

- In the previous module, we discussed that searching in a list is faster if the list is sorted
- We also discussed sorting algorithms and their complexity: merge sorting is more efficient than insertion sorting
- This lecture considers new data structures different from a list of elements
- We will see how insert, delete and search elements
- And how efficiently it can be done

UPPSALA UNIVERSITET

Everything in Python is an object

Examples:

- Numbers
- Strings
- Lists
- Dictionaries
- Functions
- Modules

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

UPPSALA UNIVERSITET

Everything in Python is an object

Examples:

- Numbers
- Strings
- Lists
- Dictionaries
- Functions
- Modules

Let’s check google colab

UPPSALA UNIVERSITET

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`.

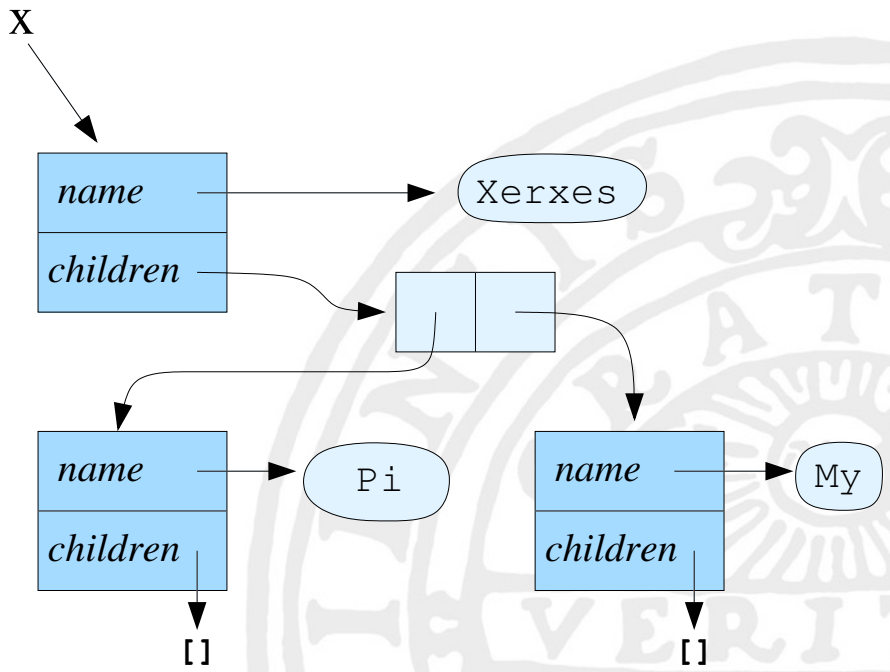
We can also see methods defining some list operators e.g. `__add__` for +, `__eq__` for == and `__le__` for <=.

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

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']
```

.

UPPSALA UNIVERSITET

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__`, ...)

UPPSALA UNIVERSITET

Python lists

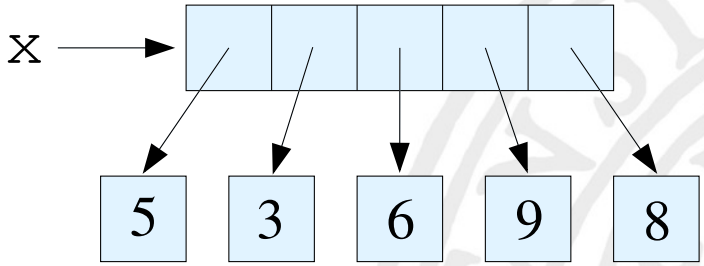
Lists are the most fundamental structure in Python.

We can, for example, write

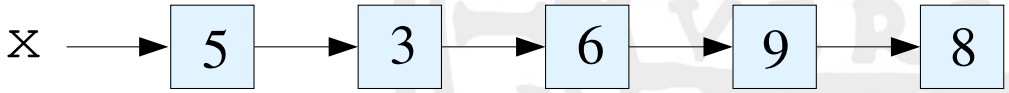
```
x = [5, 3, 6, 9, 8]
```

which can be illustrated:

Remember that it really is more like this:



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

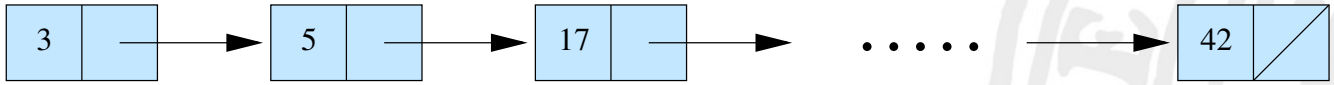


UPPSALA UNIVERSITET

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.

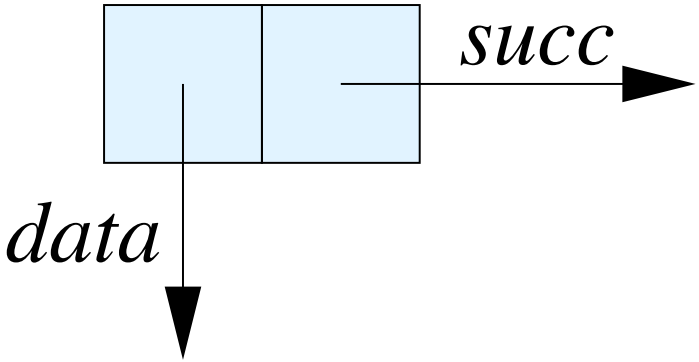
A linked list can be illustrated like this:



Each element has a value AND a reference to its follower.

UPPSALA UNIVERSITET

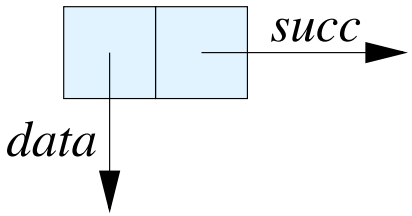
Python representation of nodes



UPPSALA UNIVERSITET

Python representation

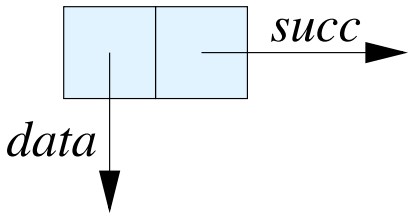
We define a class for the nodes in the list:




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

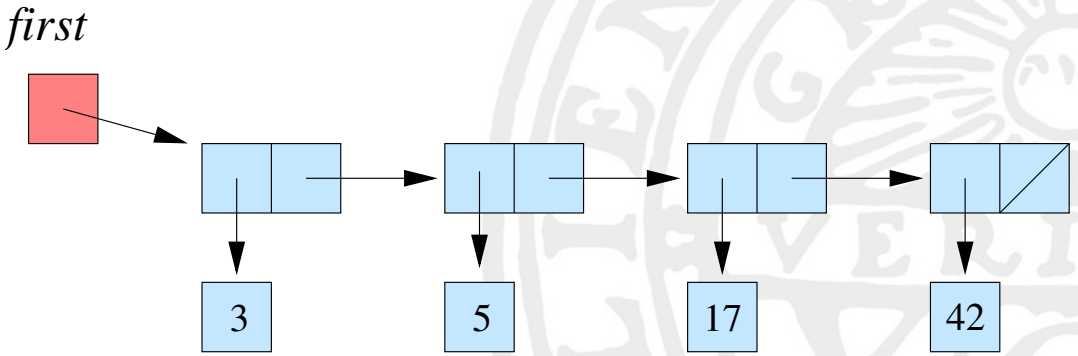
Python representation

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:



UPPSALA UNIVERSITET

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)
```

UPPSALA UNIVERSITET

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
```

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

The LinkedList class

```
class LinkedList:

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


    def __init__(self):
        self.first = None

    . . .
```

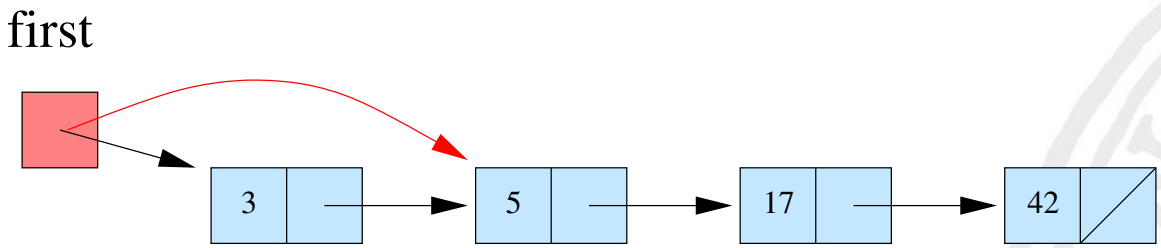
The `Node` class is inside the `LinkedList` class

Creates an empty list

UPPSALA UNIVERSITET

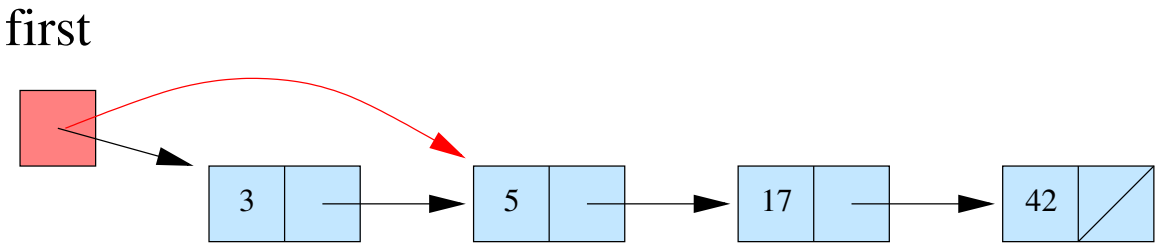
The `remove_first` method

A function removes the first element and returns its value



UPPSALA UNIVERSITET

The `remove_first` method




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

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

UPPSALA UNIVERSITET

Better `remove_first`

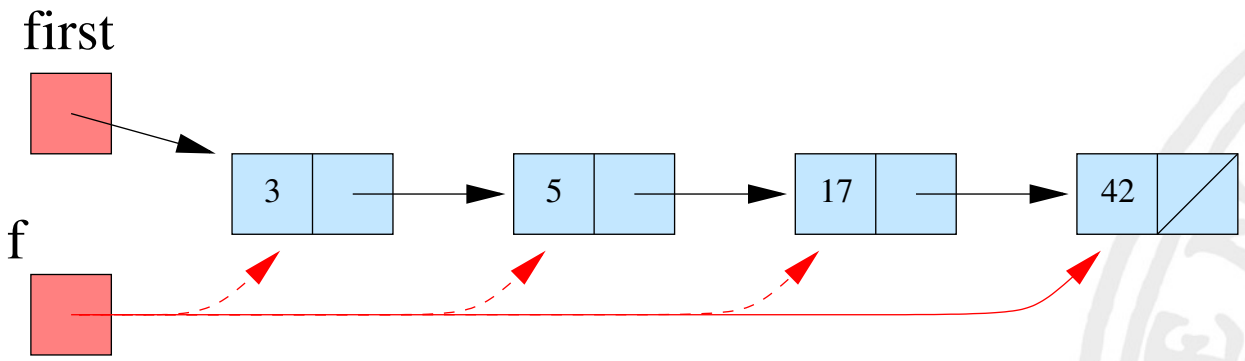
```
def remove_first(self):
    '''Removes the first element and returns its value'''
    if self.first == None:
        return None                 # Or raise an exception
    result = self.first.data
    self.first = self.first.succ
    return result
```

To check all the possible cases is one of the keys to MA2.

UPPSALA UNIVERSITET

The `get_last` method

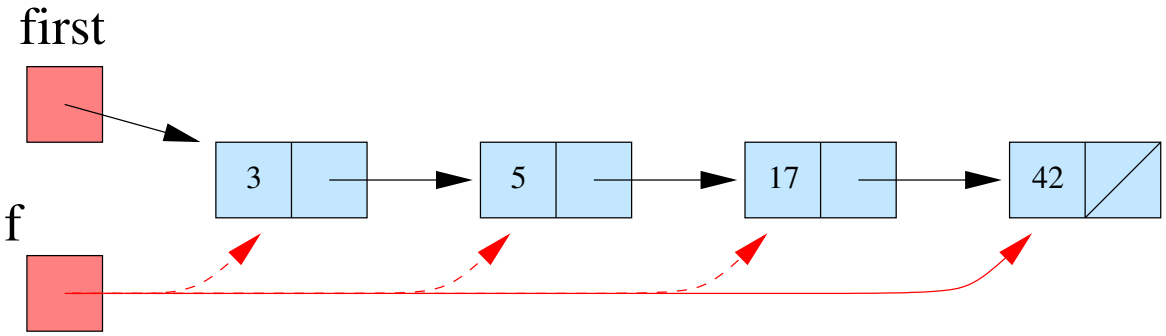
A method `returns the last stored data item`



Question: Are there any cases we need to think about?

UPPSALA UNIVERSITET

The `get_last` method




```
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
```

UPPSALA UNIVERSITET

The `__str__` method

Does a method for string representation of a linked list have a common structure with the last method (get_last)?

Let's make a pseudocode

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

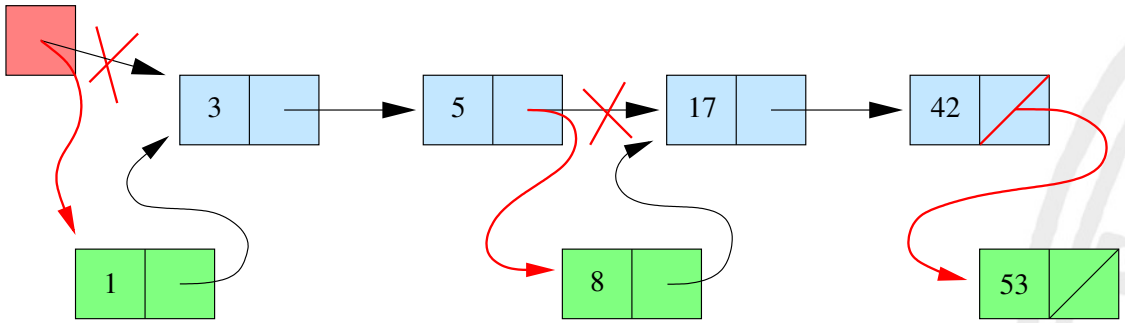
An *iterative* `insert` method

For pseudocode: what are special cases to consider when we want to insert in a linked list?

UPPSALA UNIVERSITET

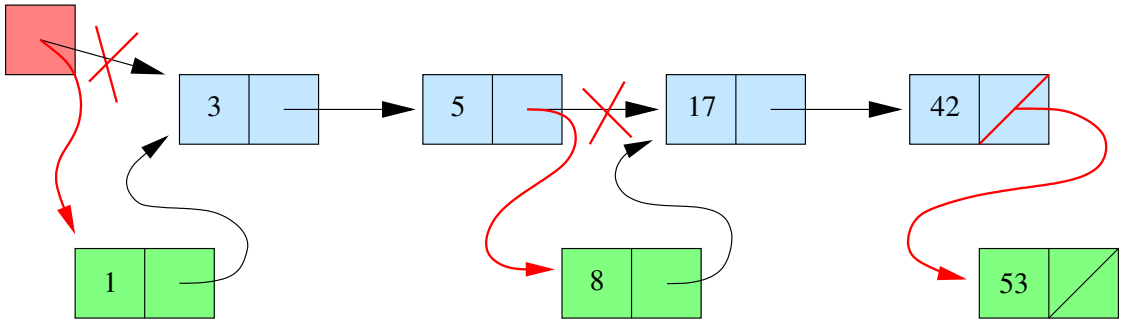
An *iterative* `insert` method

For pseudocode: what are special cases to consider when we want to insert in a linked list?



UPPSALA UNIVERSITET

An *iterative* `insert` method




```
    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)
```

Note: codes are not so long for each of the method

UPPSALA UNIVERSITET

A *recursive* insert: method 1.

`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)
```

The same as before

Insert function called for the first node

UPPSALA UNIVERSITET

A *recursive* insert: method 1.

In the `Node` class: #recursion here

```
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: # need to call insert from the first node

```
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)
```

UPPSALA UNIVERSITET

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 help method gets a reference to a node and places the new node in the **sublist** that starts in that node AND returns a reference to the first node in the modified list.
The return node may be the new node or it may be the same node that it got as input.

UPPSALA UNIVERSITET

Use a *local function* instead.

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

    self.first = _insert(x, self.first)
```

Since it is now a local function we do not use self as a parameter.

Note: helper (local) functions are useful in MA2

UPPSALA UNIVERSITET

Google colab time

This lecture discusses

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

applied to the `LinkedList` class

UPPSALA UNIVERSITET

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

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

UPPSALA UNIVERSITET

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}')
```

Question: are our class of linked list ready to make such kind of for loop?

UPPSALA UNIVERSITET

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!

What is `in` and what is `for`? How does addition work?

UPPSALA UNIVERSITET

Make the list *iterable*!

An iterable is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop

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

When provided to the **iter()** method, it generates an Iterator.

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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
```

- The `yield` statement.. It works as a return statement BUT it remembers its state (what current is) so the next call will continue from that point

- No `__next__` method. We do not have to create a next function

- `current` is a local variable. it was added as an instance variable in the former implementation

UPPSALA UNIVERSITET

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 list:
    print(”Here”)
else:
    print (”No”)
```

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

UPPSALA UNIVERSITET

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])
```

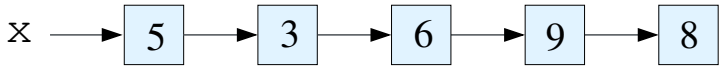
but **not**:

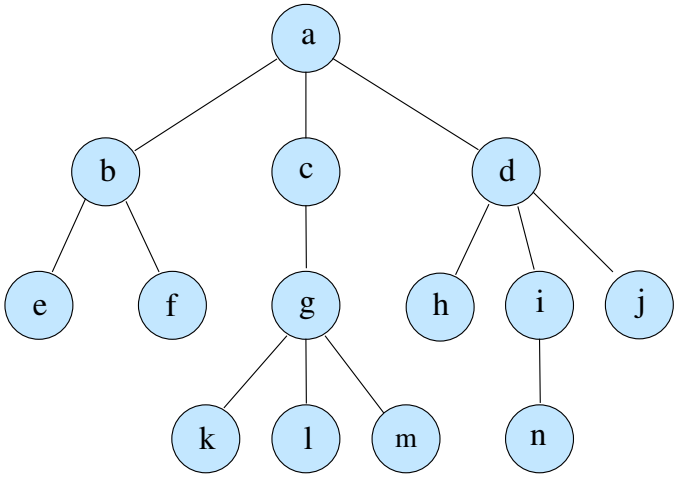
```
ll[3] = ll[0] + ll[2]
```

#__setitem__

Tree structures

- We move from lists to trees





Terminology:

- Nodes
- Root
- Leefs
- Children
- Siblings
- Height (4 or 3)
- Size (14)

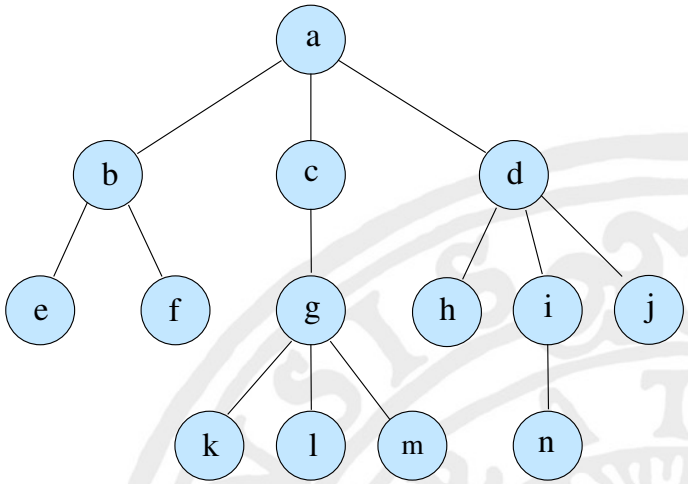
UPPSALA UNIVERSITET

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



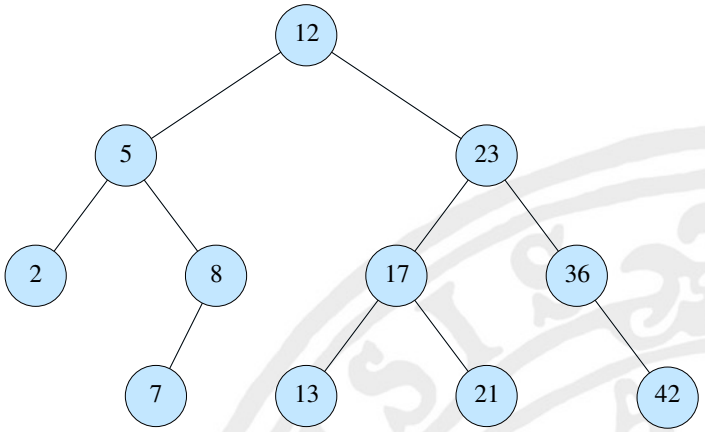
UPPSALA UNIVERSITET

Binary trees

Definition:

A set of nodes that is either empty or consists of 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

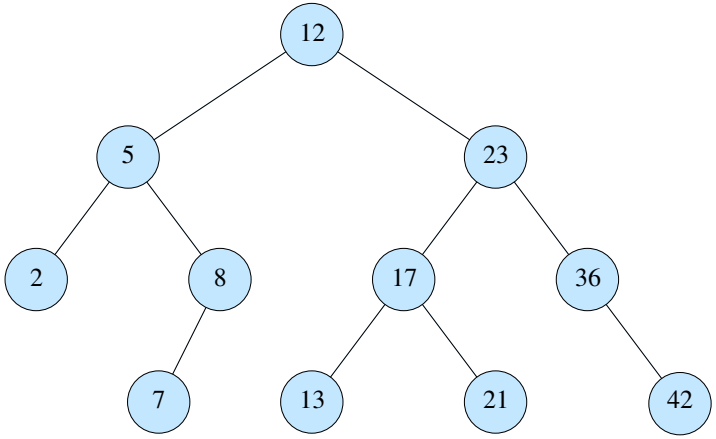


In-order traversal: 2, 5, 7, 8, 12, 13,17, 21, 23, 36, 42

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

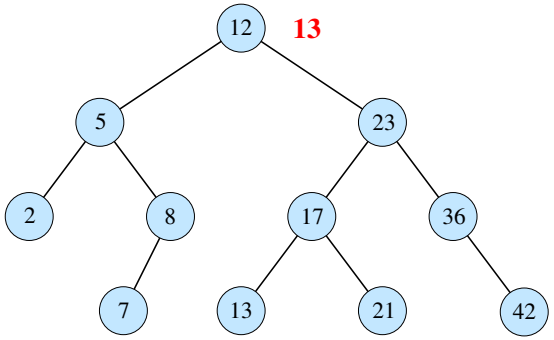
A class for binary search trees

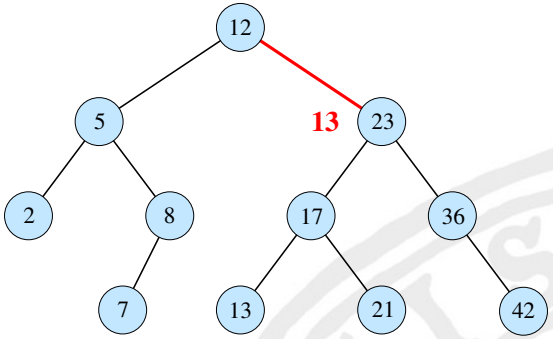
```
class BST:
    class Node:
        def __init__(self, key,
                     left = None,
                     right = None):
            self.key = key
            self.left = left
            self.right = right

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

     . . . .
```

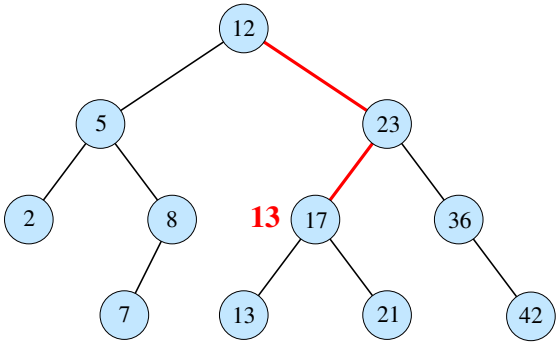
UPPSALA UNIVERSITET

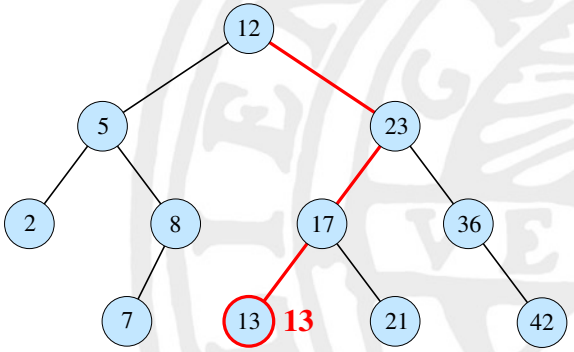
Searching

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









UPPSALA UNIVERSITET

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
```

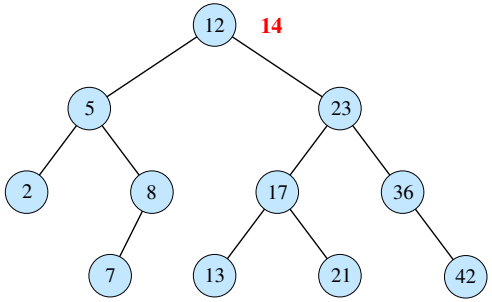
UPPSALA UNIVERSITET

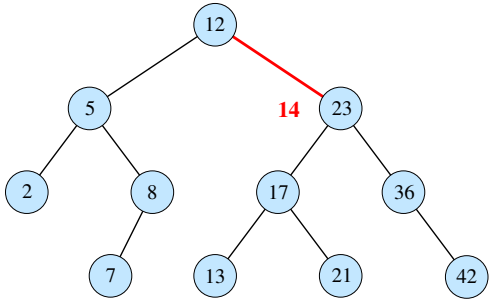
Inserting in binary search tree

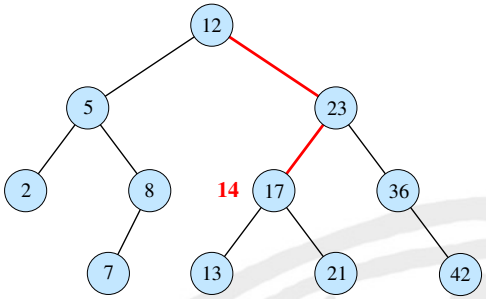
Question: Is inserting an element has something in similar with searching an element?

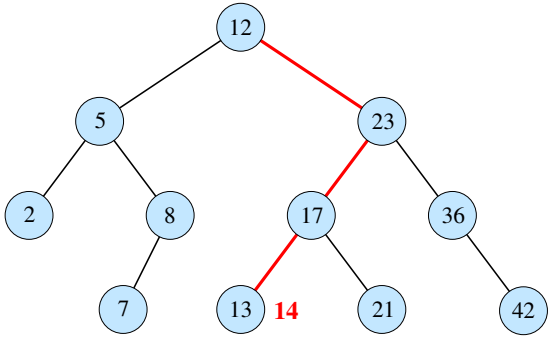
UPPSALA UNIVERSITET

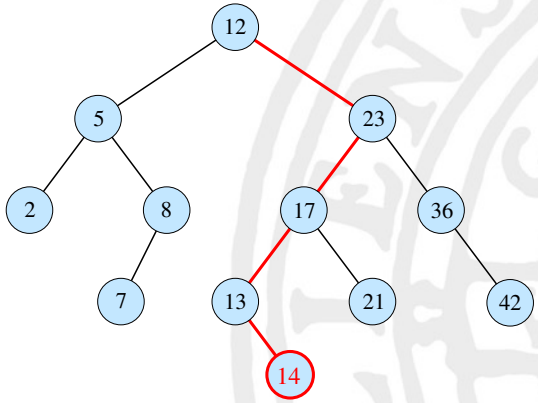
Inserting in binary search tree











UPPSALA UNIVERSITET

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 has to return the root of the modified subtree regardless if it is the new node or not.

UPPSALA UNIVERSITET

Modified insertion code

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

```
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
```

UPPSALA UNIVERSITET

Traversal: Counting nodes

```
def size(self):

    def _size(r):
        if r is None:
            pass
        else:
            pass

    return _size(self.root)
```

Question: What to return instead pass lines?

UPPSALA UNIVERSITET

Traversal: Counting nodes

```
def size(self):

    def _size(r):
        if r is None:
            return 0
        else:
            return 1 + _size(r.left) + _size(r.right)

    return _size(self.root)
```

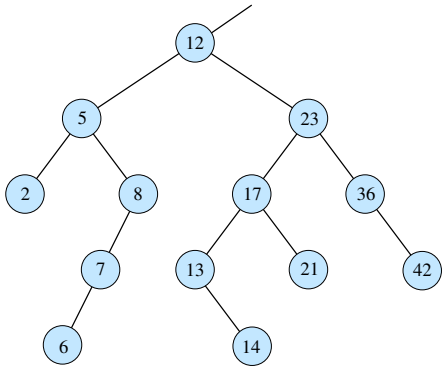
Note that `r is None` is a much better base case than `r.left is None` and `r.right is None`.

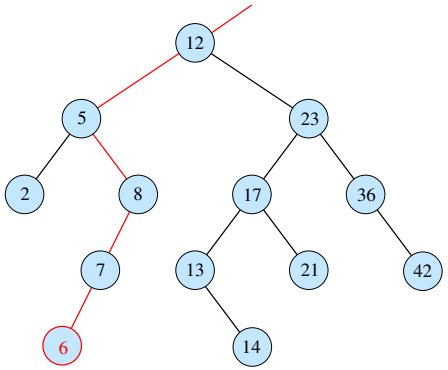
We will later see how this could be done using a *generator*.

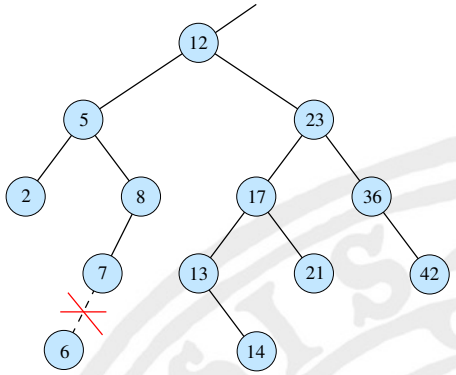
UPPSALA UNIVERSITET

Remove

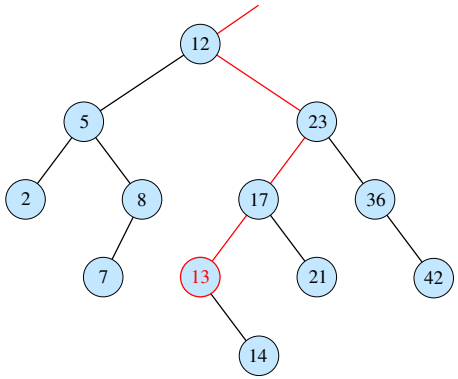
If the node to be removed has no children. Example: remove 6.

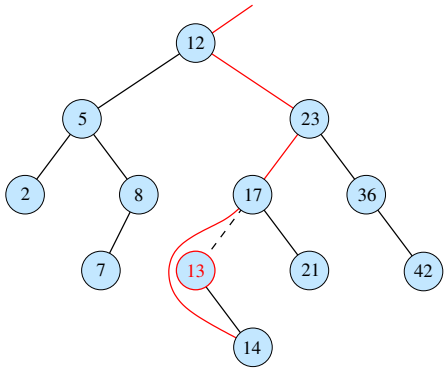


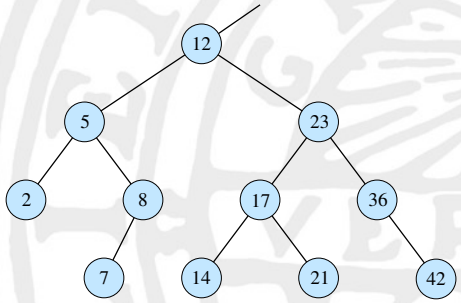




If the node to be removed has one child. Example: remove 13.



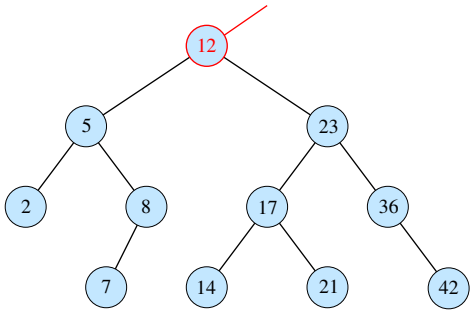


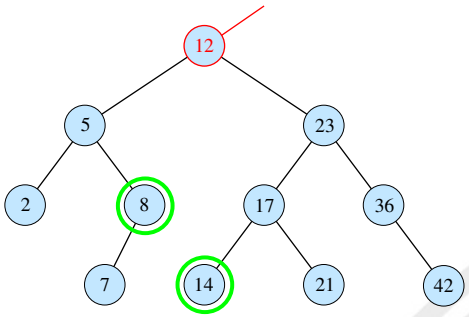


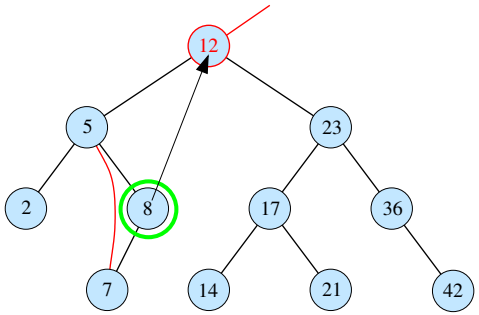
UPPSALA UNIVERSITET

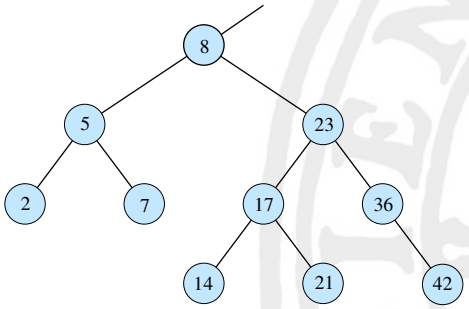
Remove – the harder case

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



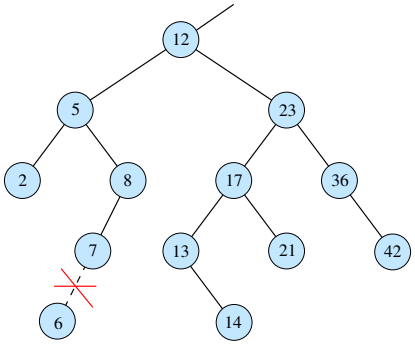


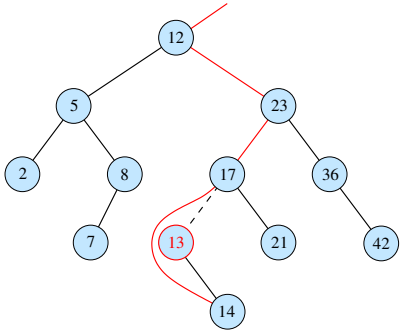


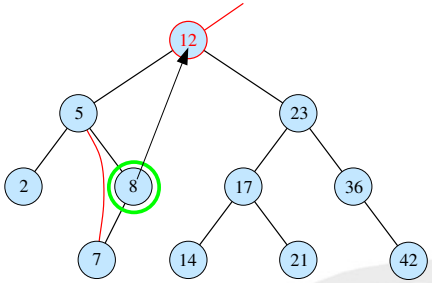


UPPSALA UNIVERSITET

useful recursive help function








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

- It starts with searching the node with requested key.
- If it has no left (right) subtree we return the right (left) subtree
- If it has both a left and right subtree we find the largest (smallest) value in the left (right) subtree in node X
- We store that key in the current node and we then `_remove` the node

UPPSALA UNIVERSITET

The LinkedList generator

The easiest way to to write the __iter__ function in the LinkedList class is 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.

UPPSALA UNIVERSITET

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.

UPPSALA UNIVERSITET

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
```

UPPSALA UNIVERSITET

Using `yield from`

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

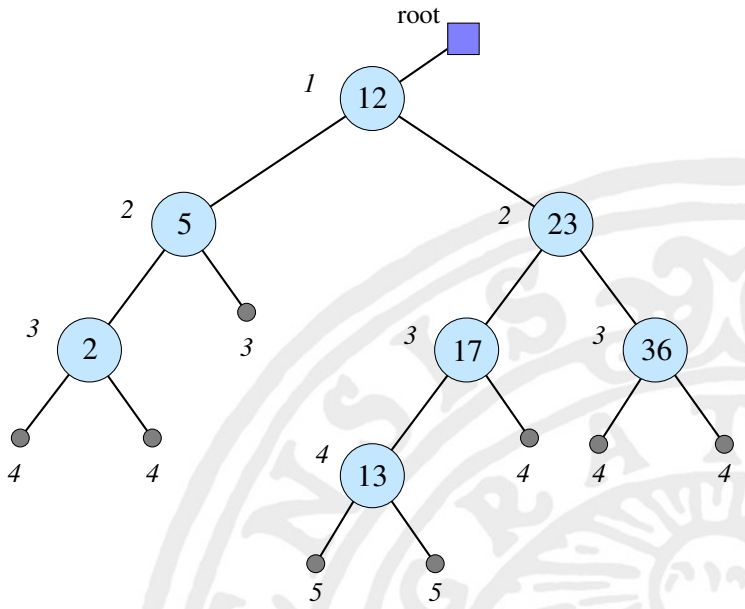
```
        def __iter__(self):              # In the Node class
            if self.left:
                yield from self.left:
            yield self.key
            if self.right:
                yield from self.right


    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!

UPPSALA UNIVERSITET

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 measurements of how well balanced the tree is. They are also linked by the relation $e = i + 2n + 1$

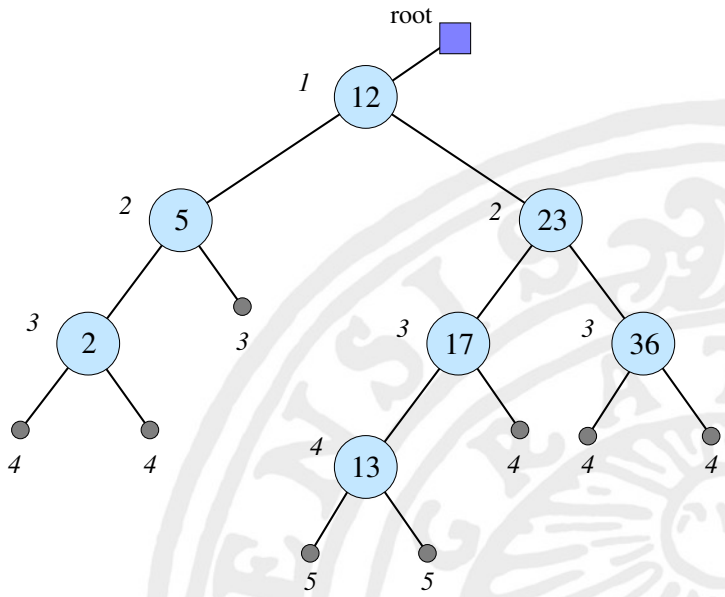
What does these numbers mean?

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 $\frac{33}{7+1} = 4.12$.



UPPSALA UNIVERSITET

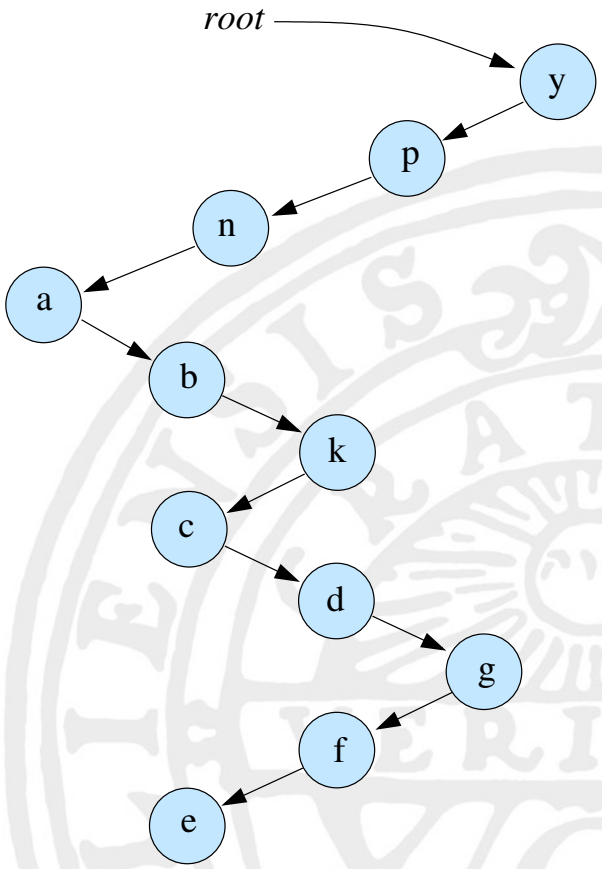
Maximal values?

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.

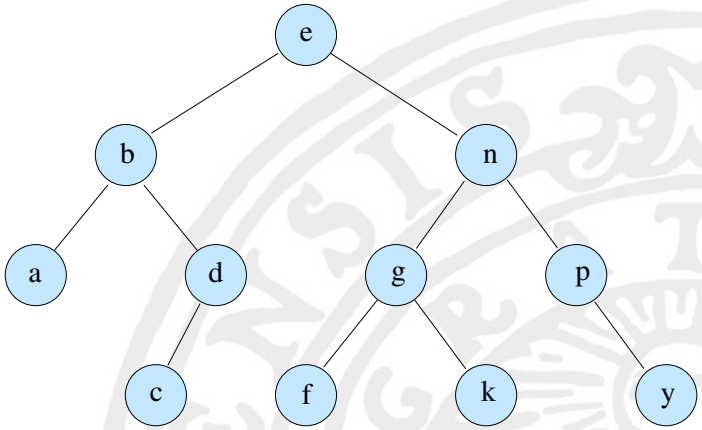


UPPSALA UNIVERSITET

Minimal values

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 $O(\log(n))$



UPPSALA UNIVERSITET

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 1,000,000 keys require, on the average on the average tree $1.39 \cdot \log_2 10^6 \approx 28$ tries.

UPPSALA UNIVERSITET

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 $(n+1)/2$ node visits. However, there are several insertion algorithms that keep the tree well balanced (AVL-trees, RB-trees …)