UPPSALA UNIVERSITET

MA3: Binary trees

Tom Smedsaas

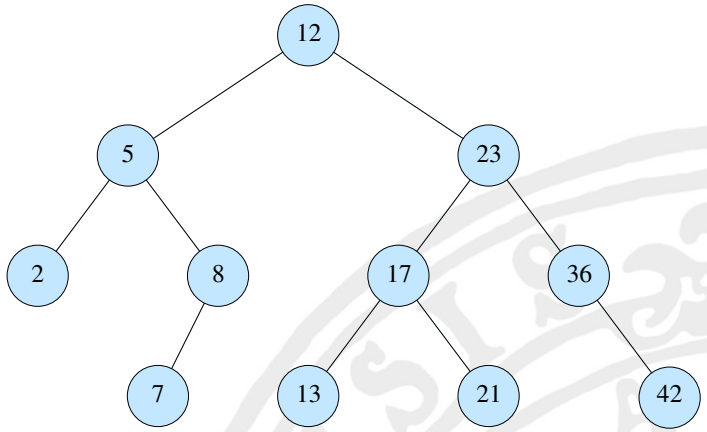
UPPSALA UNIVERSITET

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

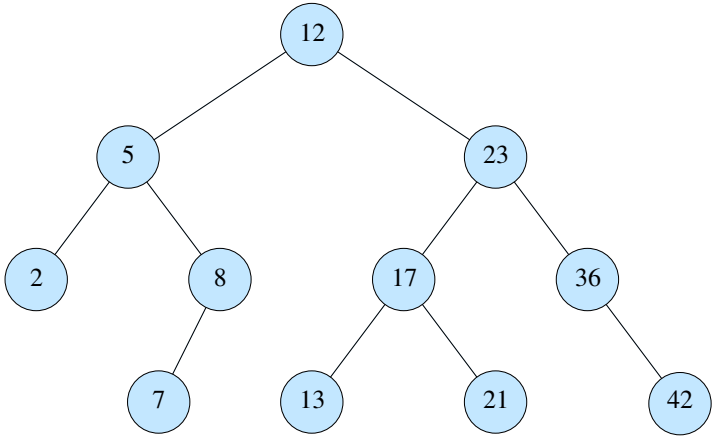


Inorder traversal: 2, 5, 7, 8, 12, 13,17, 21, 23, 36, 42

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.

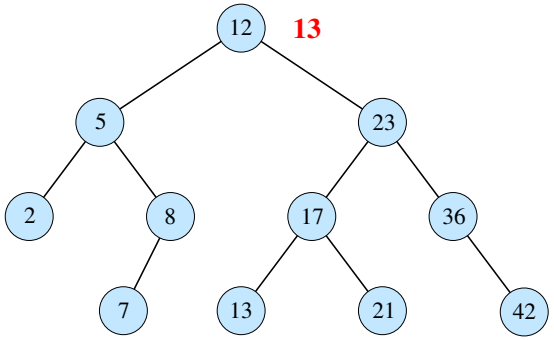
UPPSALA UNIVERSITET

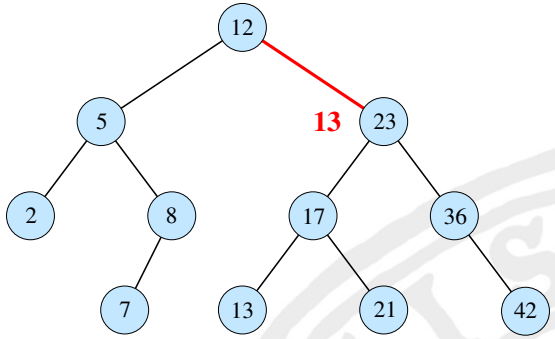
A class for binary search trees

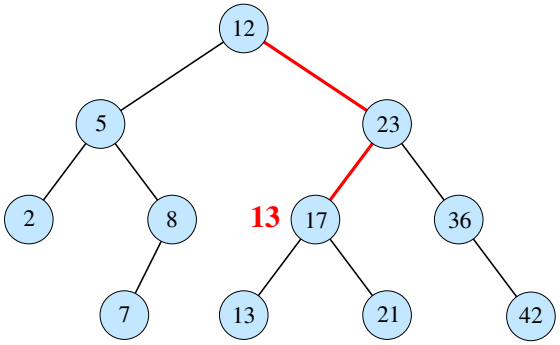
```
class BST:
    class Node:
        def __init__(self, key,
                     left = None,
                     right = None):
            self.key = key
            self.left = left
            self.right = right

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

     . . . .
```

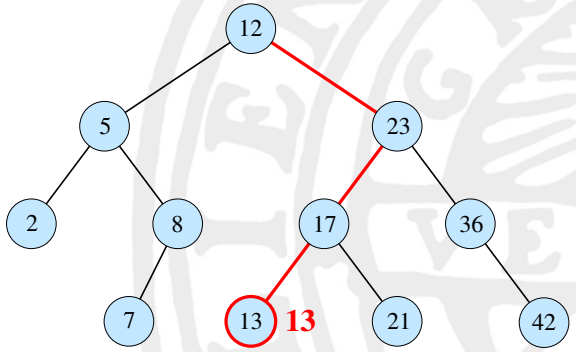
Searching

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

Traversal: Counting nodes

Use a recursive help method:

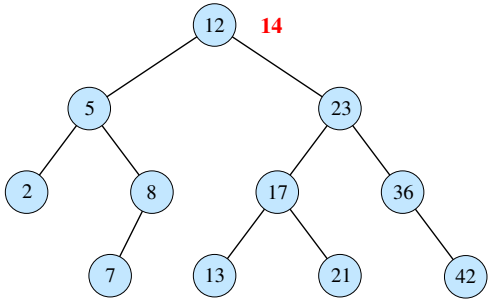
```
def size(self):
    return self._size(self.root)

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

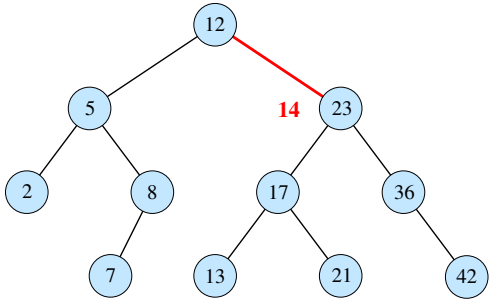
Note that `r == None` is a much better base case than `r.left == None` and `r.right == None`.

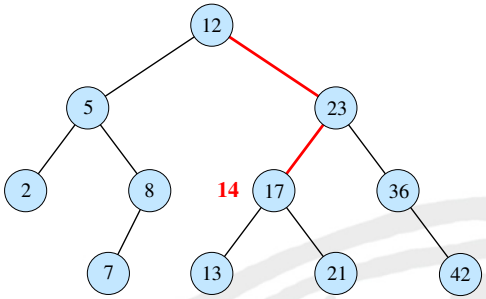
We will later see how this could be done using a *generator*.

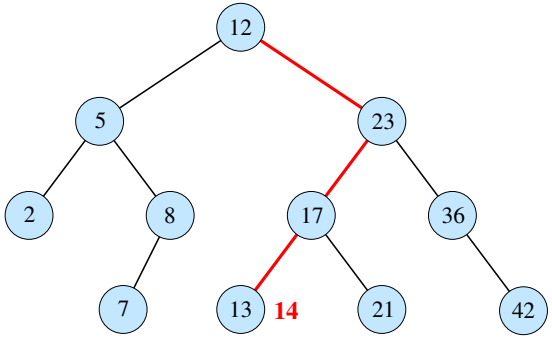
UPPSALA UNIVERSITET

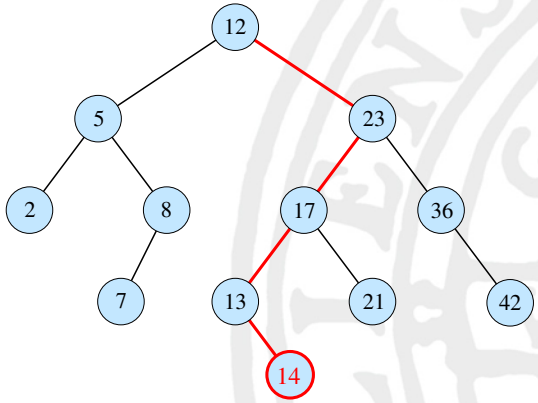
Inserting in binary search tree











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.

UPPSALA UNIVERSITET

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
```

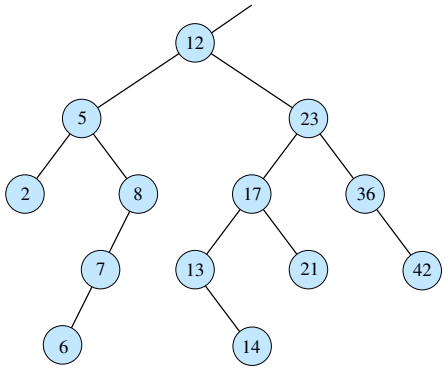
UPPSALA UNIVERSITET

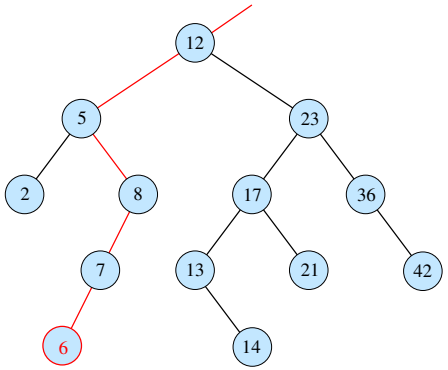
Modified insertion code

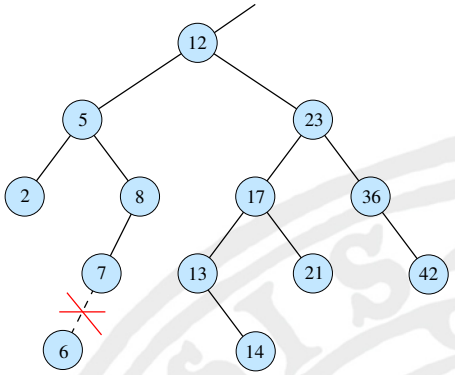
```
def insert(self, key):
        self.root, result = self._insert(self.root, key)
        return result

    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

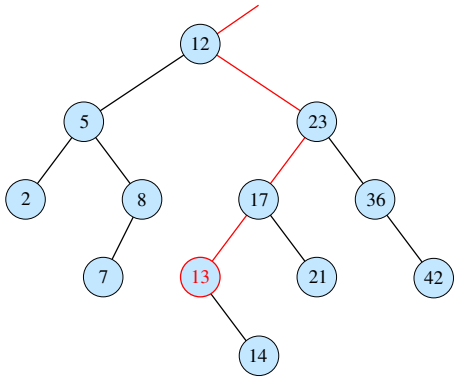
If the node to be removed has no children. Example: remove 6.

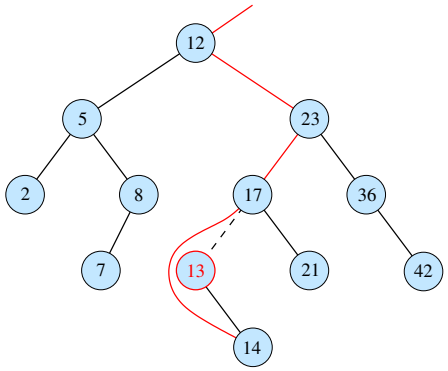


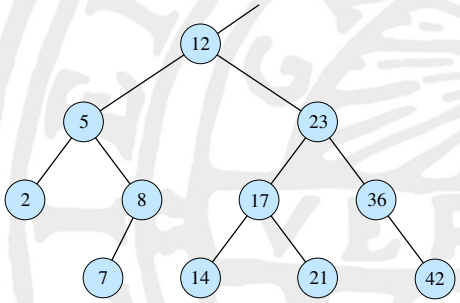




If the node to be removed has one child. Example: remove 13.



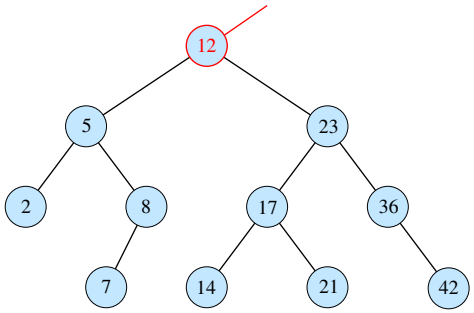


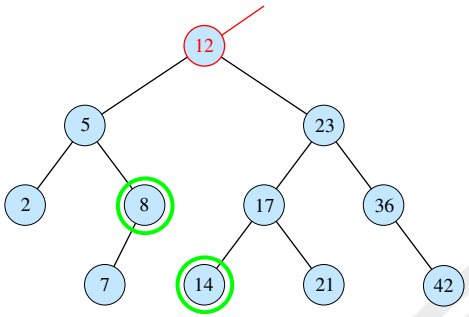


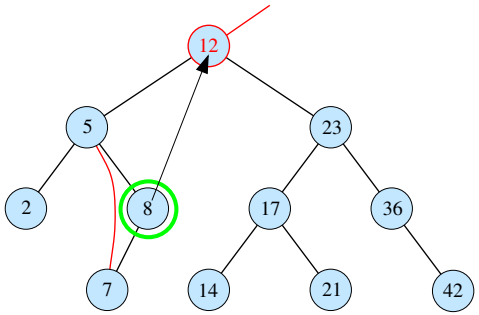
UPPSALA UNIVERSITET

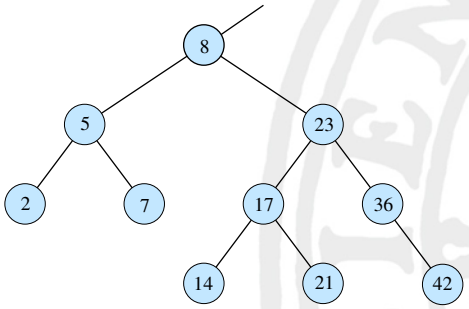
Remove – the harder case

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

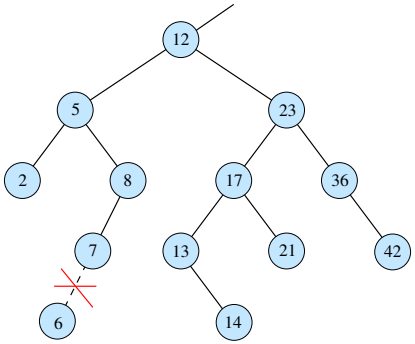


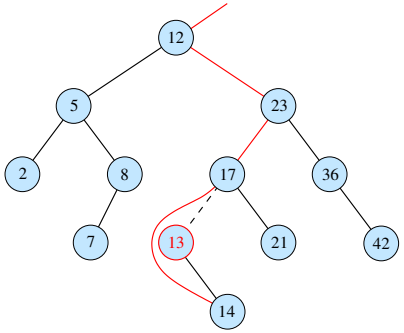


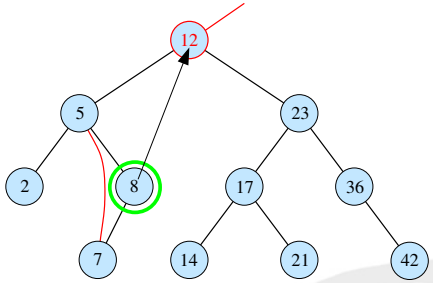




Coding hints for remove








```
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.

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

Find the largest and then then call `_remove` recursively

```
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