Binary Search Trees (Arbres binaires de Recherche)

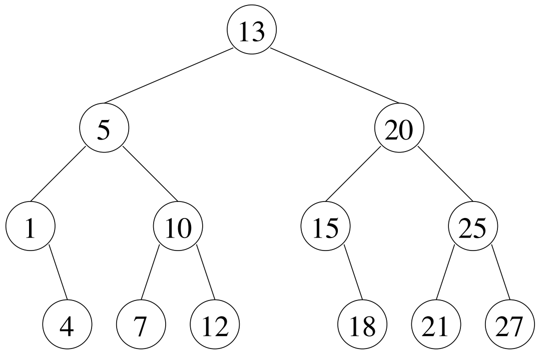

Figure 1: Binary Search Tree (BST)

1 Tools

Exercise 1.1 (List ↔ BST)

1. Write a function that builds a sorted list with the elements of a binary search tree.
2. Write a function that builds a *balanced* binary search tree using a sorted list.

Exercise 1.2 (Test)

Write a function that tests whether a binary tree is a search tree or not.

---

2 Classics

Exercise 2.1 (Researches)

1. (a) Where are the maximum and the minimum values in a non-empty binary search tree?
   (b) Write the two functions `minBST(`B`)` and `maxBST(`B`)`, where B is a non empty BST.
2. Write a function that searches a value x in a binary search tree. It returns the tree whose root contains x if found, the value `None` otherwise.

Two versions for each function: recursive and iterative!

Exercise 2.2 (Insertion at the leaf)

1. Use the insertion at the leaf principle to create the binary search tree obtained, from an empty tree, by successive insertions of the following values (in that order):

$$13, 20, 5, 1, 15, 10, 18, 25, 4, 21, 27, 7, 12$$

2. Write a function that adds an element to a binary search tree.

Exercise 2.3 (Deletion)

Write a recursive function that deletes an element from a binary search tree.

Exercise 2.4 (Root insertion)

1. Use the root insertion principle to create the binary search tree obtained, from an empty tree, by successive insertions of the following values (in that order):

$$13, 20, 5, 1, 15, 10, 18, 25, 4, 21, 27, 7, 12$$

2. Write the function that inserts an element in a binary search tree as a new root.

---

3 Final (partiel 2016)

Exercise 3.1 (BST and mystery)

```python
def bstMystery(x, B):

 # first part
    P = None
    while B != None and x != B.key:
        if x < B.key:
            P = B
            B = B.left
        else:
            B = B.right
    if B == None:
        return None

 # second part
    if B.right == None:
        return P
    else:
        B = B.right
        while B.left != None:
            B = B.left
        return B
```

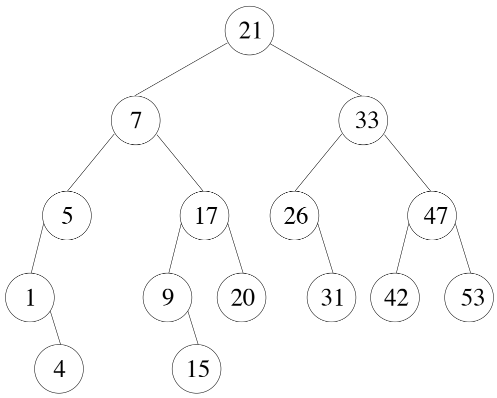

FIGURE 2 – tree B_1

```python
def call(x, B):
    p = bstMystery(x, B)
    if p == None:
        return None
    else:
        return p.key
```

1. Let B_1 be the tree given above. What are the results of each of the following calls?
   (a) `call(25,` B_1`)`
   (b) `call(21,` B_1`)`
   (c) `call(20,` B_1`)`
   (d) `call(9,` B_1`)`
   (e) `call(53,` B_1`)`

2. `bstMystery(x, B)` is called with `B` any binary search tree, where all elements are different. During execution, at the end of part 1:
   (a) What does `B` represent?
   (b) What does `P` represent?

3. What does the fonction `call(x, B)` do?

In the two following exercises, we use a new implementation of binary trees where each node contains the size of which it is the root of.

```python
class BinTreeSize:
    def __init__(self, key, size, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right
        self.size = size
```

Exercise 3.2 (Add the size)

Write the function `copyWithSize(`B`)` that takes a "classic" binary tree B (`BinTree` without the size) as parameter and returns an equivalent tree (containing the same values at the same places) but with the size specified in each node (`BinTreeSize`).

Exercise 3.3 (Median)

We will study the research of the node that contains the median value in a binary search tree, that is, the value at the rank $size(B) + 1 \ div\ 2$ in the list of elements in increasing order.

For this, we want to write the function `nthBST(`B`, `k`)` that returns the node that contains the k^{th} element of the tree B. For example, the call `nthBST(`B_1`, 3)` with B_1 the tree in exercise 3.1 will return the node that contains the value 5.

1. **Abstract study:**

   The *size* operation, defined as follows, is added to the abstract definition of binary trees (given in appendix):

   OPERATIONS
   $size$: BinaryTree → Integer

   AXIOMS
   $size\ (\text{emptytree}) = 0$
   $size\ (<\text{o}, \text{L}, \text{R}>) = 1 + size\ (\text{L}) + size\ (\text{R})$

   Give an abstract definition of the operation *nth* (that has to use the operation *size*).

2. **Implementation:**

   The functions you have to write use binary trees with the size in each node (`BinTreeSize`).

   - Write the function `nthBST(`B`, `k`)` that returns the tree with the k^{th} element as root. We suppose that this element always exists: $1 \leq k \leq size(B)$.
   - Write the function `median(`B`)` that returns the median value of the binary search tree B if non empty.