Algorithmique des structures de données arborescentes
Exercise sheet 5

1 Reminders on the height of a binary tree

Exercice 1: height

1. Recall what the *minimum* height of a binary tree of n nodes is. For which trees is it reached ?
2. Recall what the *maximum* height of a binary tree of n nodes is. For which trees is it reached ?

2 AVL trees

We saw that for a binary search tree of height h, the operations of

— searching for an element,

— adding an element,

— removing an element

can be done in time $O(h)$ in the worst case.
To guarantee a logarithmic complexity for those operations, we will consider families of trees that guarantee a height of at most $\alpha \log(n)$ for trees with n nodes (where the constant $\alpha > 0$ depends on the family of trees). Such trees are called "balance ratiod". Intuitively, a tree of a family of balance ratiod trees will have a lot of nodes, exponentially many compared to its height.

A 1st family of such trees is that of AVL trees. The term AVL comes from the initials of the researchers that introduced those trees, Georgii Adelson-Velsky et Evguenii Landis.

AVL trees are binary search trees, but they enforce an additional constraint of balance ratio for each node. That constraint bounds the height of the tree.

2.1 Definition of the balance ratio of a node.

A notion of *balance ratio* is associated to each node. We recall that for every binary tree t, we denote its height by $h(t)$, and it is -1 if the tree is empty.

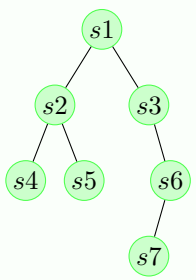
Definition : balance ratio of a node.

Let t be a binary tree. Let x be a node of t, ℓ the left subtree of x and r the right subtree of x. The *balance ratio of the node* x is the difference between the heights of its subtrees : $h(\ell) - h(r)$.

Exemple

We give the balance ratio of the nodes of the tree on the left in the following array.

Nœuds	s1	s2	s3	s4	s5	s6	s7
Équilibres	-1	0	-2	0	0	1	0



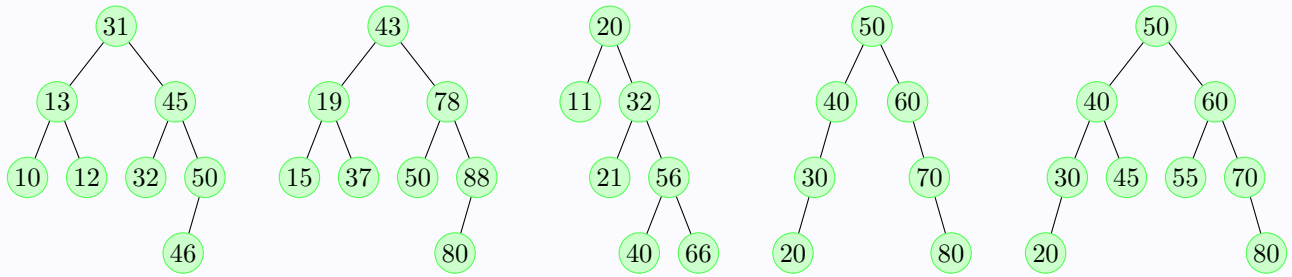
In the family of AVL trees, the balance ratio of each node is bounded.

Definition : AVL trees

An *AVL* is a binary search tree in which *every* node has balance ratio -1, 0, or 1.

Exercice 2: Exemples d'arbres AVL

Among the following binary trees, which ones are AVLs ? Justify your answer.

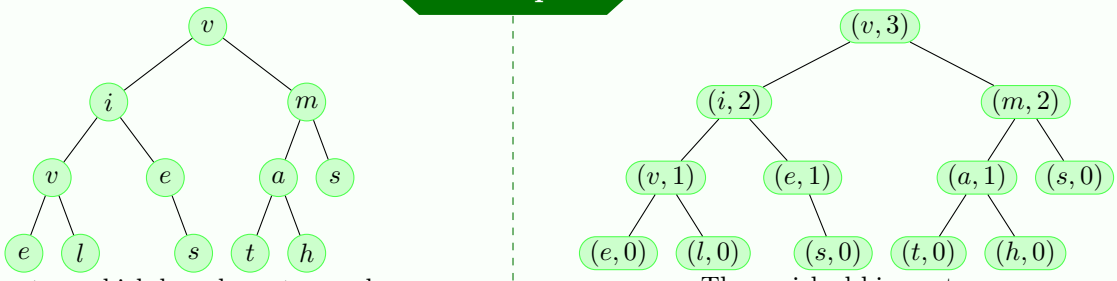


2.2 Notion of enriched tree

Computing the balance ratio of a node requires accessing the height of the subtrees. But the function that computes the height of a binary tree has linear complexity. Calling this function for each node of the tree would thus lead to a quadratic complexity (that is, $\mathcal{O}(n^2)$ of trees with n nodes). This is not compatible with the logarithmic complexity we want.

To solve this problem, we will use *enriched binary search trees*, that is binary search trees where the nodes contain not only the keys, but also additional information.
In the present case, the information we need is the height : if we store in each node, in addition to its key, the height of the subtree it is a root of, then the balance ratio of a node can be computed in constant time.
We will thus work with enriched binary trees, where the nodes are labeled by pairs : the first element will be its key, and the second one the height of the subtree. Thus, in OCaml, the following tree t
Node((5,−1), Empty, Node((10,0), Empty, Empty)) is an enriched tree of height 1 with key 5 in its root.

Exemple



Binary tree which has characters as keys

The enriched binary tree.

Exercice 3: Enriched trees

The following functions will enable us to manipulate enriched trees.

1. Write a function get_height of type ('a ∗ int) btree −> int that takes an enriched binary tree t as an input and returns its height (with complexity $\mathcal{O}(1)$).
2. Write a function tag_node of type 'a −> ('a ∗ int) btree −> ('a ∗ int) btree −> ('a ∗ int) btree that takes a key c and two enriched binary trees ℓ and r as input. It must return the enriched binary tree with key c and left and right subtrees ℓ and r respectively (with complexity $\mathcal{O}(1)$).
3. Using the two previous functions, write a function tag_btree of type 'a btree −> ('a ∗ int) btree that takes any binary tree t as an input and returns the corresponding enriched tree(with complexity$\mathcal{O}(n)$).

To test if an enriched tree is an AVL, we need to test if its a binary search tree (we can adapt the function is_bst from the previous exercise sheet to enriched trees). In the following exercise, we will assume that test is already done.

Exercice 4: Belonging to the AVL family

Write a function is_avl of type ('a * int) btree -> bool that takes an enriched bst and tests if it is AVL (with complexity $\mathcal{O}(n)$)

3 Height of an AVL tree

We will now show that the height of AVL trees is always of the order of the *logarithm* of its height. More precisely, we will show the following : For every AVL tree t of height $h(t)$ and of size $s(t)$, there exists a constant $\alpha > 0$ such that :

$$h(t) \leqslant \alpha \times log_2(s(t) + 1). \tag{5.1}$$

Therefore the search algorithm in a BST applied to an AVL tree t, will have complexity $\mathcal{O}(log_2(s(t)))$.

Exercice 5: Height of an AVL tree

For all integer $h \geqslant 0$, we denote by $S_{min}(h) \in \mathbb{N}$ the minimum size of an AVL tree of height h.

1. What is the valued of $S_{min}(0)$, $S_{min}(1)$?
   What is the recursive relation verified by $S_{min}(h)$ for $h \geqslant 2$?
2. Let $u(h) = S_{min}(h) + 1$ for all $h \geqslant 0$. What is the recursive relation verified by the sequence u ?
3. We denote by φ the golden ratio, that is the unique positive solution to the relation $x^2 = x + 1$ (on a $\varphi = \frac{1+\sqrt{5}}{2}$). Show that for all $h \geqslant 0$, we have,
   $$\varphi^h \leqslant u(h).$$
4. We will admit that $\frac{1}{log_2(\varphi)} \leqslant 1.45$. Use the previous question to show that (5.1) holds.

4 Insertion in an AVL tree

Adding a new key in an AVL can disrupt the balance ratio of the tree. However, we will show that we can modify the tree after the insertion to make it an AVL, without additional complexity cost. We start by changing the AVL by an AVL enriched with the heights.
The insertion algorithm in AVL trees will have two steps :

— apply the normal insertion algorithm on binary search trees, by adding a new leaf.

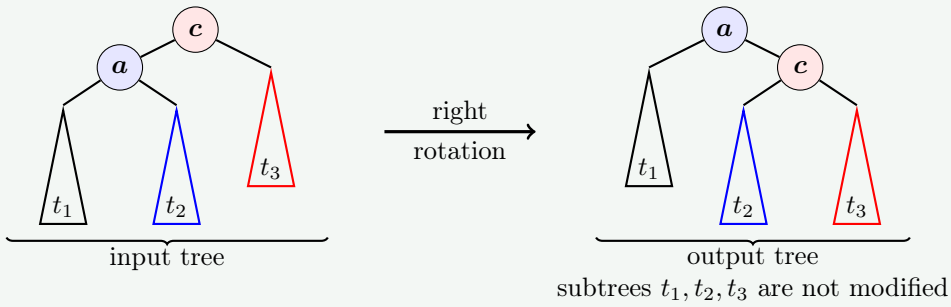
— if the AVL property was destroyed by the insertion, we rebalance ratio the tree by reorganizing it.

The rebalance ratio of a tree is based on the *rotation* notion, that we will now introduce.

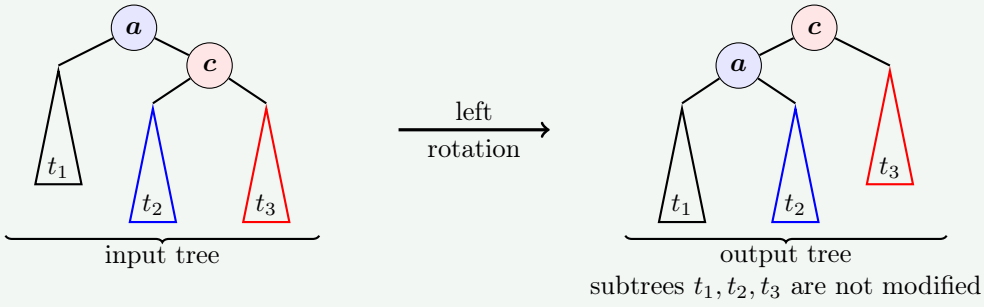
4.1 Rotations

We will define four transformations that can be applied on a binary tree and can be applied in constant time. They output a new tree obtained by reorganizing the trees that are close to the root. We represent them graphically for definition.

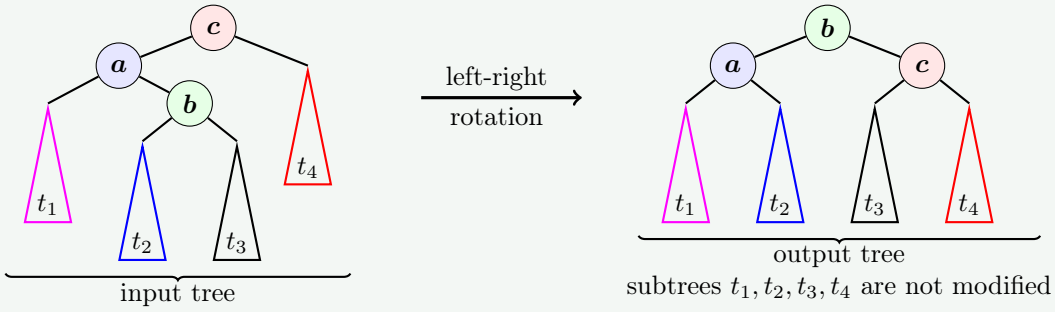
Definition : right rotation



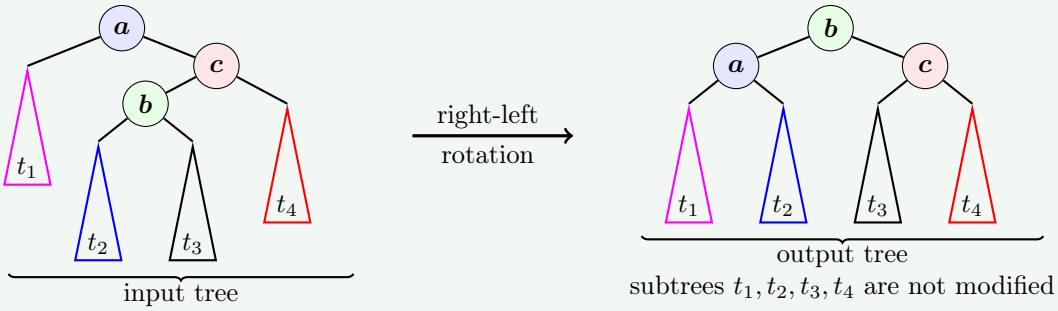
Definition : left rotation



Definition : left-right rotation



Definition : right-left rotation



Each rotation applied to a binary search tree outputs a new tree that remains a binary search tree.

Exercice 6: Property of a tree post rotation

Let t be a BST and t' a second tree, built from t by applying one of the four rotations. Show that t' is a BST.

Exercice 7: Rotations

Write four functions l_rotate, r_rotate,
lr_rotate, and rl_rotate, each of type ('a * int) btree -> ('a * int) btree, that implement the four rotations for *enriched* binary trees (their time complexity must be $\mathcal{O}(1)$).

Beware : Since we work with *enriched* binary trees, do not forget to modify the heights of the nodes moved in the rotation.

4.2 Bebalancing

The rebalancing algorithm is a "bottom-up" algorithm, climbing from the leaf that contains the new key up until the root.

Exercice 8: Example

Consider the following key sequence $10, 20, 30, 40, 50, 60, 70, 45, 47$. Draw the steps of successively inserting those keys in an originally empty AVL. We will take care of writing the balance ratio of the nodes for each intermediate tree.

We can now sum up all of the steps of inserting a new node in an AVL.

Exercice 9: Insertion

1. Suggest an insertion algorithm that takes an enriched AVL t and a key c as input. It must output an new enriched AVL t' whose set of keys is that of t, plus the additional key c, and satisfies $h(t) \leqslant h(t') \leqslant h(t) + 1$. The complexity of your algorithm must be $\mathcal{O}(h(t))$. Justify that your algorithm is correct.
2. Write a function avl_insert of type ('a * int) btree -> ('a * int) btree that implements the algorithm from the previous question.