- 1 - Algorithms analysis

This section will focus on the analysis of algorithms in terms of complexity. To be more precise, even though we should also focus on the space complexity, the following discussion will just cover the time complexity. Also, we will make a focus on the *amortized time complexity*.

To start, what do we mean by time complexity? Basically we refer to how well (or bad) our algorithm does in terms of time, or put in other words, how much time it needs to complete its execution.

We can distinguish three cases, respectively:

1. *the best case:* the time required by the best input configuration;
2. *the average case:* the time required by the average input configuration;
3. *the worst case:* the time required by the worst possible input configuration.

Remark: From now on we will refer to each case using the associated notation. Briefly, we will use Ω, Θ, $\mathcal{O}$ to refer to the best, average and worst case respectively[1].

- 1.1 - Amortized analysis

Until now we have focus our attention to the single operation done by the algorithm; let's now suppose to have some data structure, and to operate on its data. More precisely we'll considere a sequece S of operation done onto such structure.

Just as a reference let's consider the following: Let's assume to run

```
procedure increaseCounter(array v, int n)
    for i = 0 to n − 1 do
        v[i] = not v[i]
        if ( v[i] == 1 ) then break
```

Figure 1: Algorithm for a n bit counter.

such algorithm n times: one would expect that $\mathcal{O}(n^2)$ is required in the worst case, but in reality the time required is just $\mathcal{O}(n)$ due to the way data may be arranged.

Definition: Let $T(n,k)$ be the time spent in the worst case to compute the k operations, then

$$T_a(n) = \frac{T(n,k)}{k}$$

represents the amortized time of the algorithm.

- 2 - Search trees

The *search tree* is a data structure that supports operation such SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE, each of which must take time proportional to the height of the tree.

In the following we'll focus on the binary search tree and some versions of it.

- 2.1 - Binary seach tree and its variants

As the name suggests, a binary search tree is a search tree organized as a binary tree. As an example, let's consider the binary tree shown in *Figure 2.a* whose representation as binary search tree is shown in *Figure 2.b*. The latter, in addition to the *key* field has also three additional fields

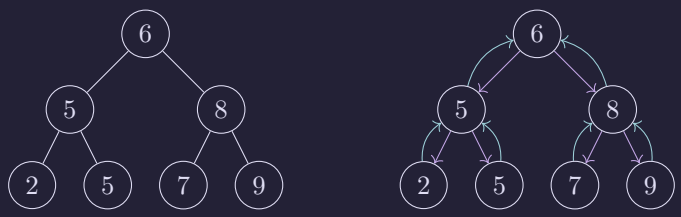

Figure 2: A simple binary tree, alongside its binary tree representation. Here with $\rightarrow$ we indicate the pointers to the left and right child, while with the $\rightarrow$ we indicate the pointer to the parent.

used to indicate the parent node, the left and right subtree roots.

Remark: Given T a binary tree, we call it binary search tree if the following holds: for every node x in T, if y is a node in the left subtree of x, then $y.key \leq x.key$, if y is a node in the right subtree of x, then $y.key \geq x.key$.

The above remark tells us that, if visited in order, the elements of the tree appear sorted.

- 2.1.1 - Red black trees

[1] We assume the reader knows what $\Omega(f(x))$, $\Theta(f(x))$ and $\mathcal{O}(f(x))$ represents.