Algorithmic Design / Algorithmic Data Mining

Data Mining - Lecture 1

Tommaso Padoan <tommaso.padoan@units.it>

About the course

Textbooks:

1. Cormen, Thomas H., et al.: *Introduction to algorithms.* MIT press.
2. Gusfield, Dan: *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press.
3. Aggarwal, Charu C.: *Data Mining: The Textbook.* Springer.

Recordings and other material on the Team:

- CD2025 298SM ALGORITHMIC DESIGN
  access code: **i5yqgwo**

Class schedule:

- Monday: 10:15 - 12:45, room 3B [building H3]
- Tuesday: 11:00 - 13:30, room 4D [building H2bis]

Exam:

One written exam on the corresponding two modules. The part for the Data Mining module will consist of theory questions about the topics explained during lectures, and/or require to apply what you learned to (simple) examples or problems.

Tutor:

The courses of Algorithmic Design and Algorithmic Data Mining have a tutor:

- *Roberta Lamberti* <roberta.lamberti@studenti.units.it>

Don't hesitate to ask questions to her by email or on Teams about any topic of the course. (Or just ask me in class.)

Main topics:

- Advanced data structures
- Probabilistic data structures
- Pattern matching

Asymptotic notation

Let f and g be two functions with domain the natural numbers (or reals).

- **O-notation:**
  $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$.
- **Ω-notation:**
  $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $c\, g(n) \leq f(n)$ for all $n \geq n_0$.
- **Θ-notation:**
  $f(n) = \Theta(g(n))$ if there exist positive constants c, c' and n_0 such that $c\, g(n) \leq f(n) \leq c'\, g(n)$ for all $n \geq n_0$.

Red-Black Trees

A *red-black tree* (RBT) is a binary search tree (BST) with some extra balancing properties. So, first of all, by the BST property: the key in every node is larger or equal to those in its left subtree and smaller or equal to those in its right subtree. The additional properties that a RBT must satisfy are meant to enforce some balance on the length of the paths in the tree, hence its height. These properties involve a coloring of the nodes and are called the **red-black properties**:

1. Every node is either red or black.
2. The root is black.
3. Every *NIL* node is a black leaf.
4. Children of red nodes must be black.
5. For every node, all downwards paths from the node to a descendant leaf must contain the same number of black nodes.

Property (3) means that the leaves of a RBT are not actually real nodes, although they are still colored and treated as such. On the other hand, nodes bearing a key are always regarded as inner nodes. This also means that inner nodes have always exactly two children. Properties (4) and (5) focus on providing the guarantees on the tree height.

To properly employ RBTs, we will assume that nodes have the following attributes:

- *left:* a pointer to the left child
- *right:* a pointer to the right child
- *p:* a pointer to the parent node
- *key:* the key contained in the node
- *color:* the color of the node, which can be red or black.

Moreover, *NIL* leaves will be actual node objects, since they have a color, and it will also be useful for implementing operations on RBTs.

Since we are just interested in the inner nodes of the tree, the only ones actually containing data, we consider the *size* of the tree to be the number of its inner nodes. We define the *black-height* of a node to be the number of black nodes along a path downwards from the node (itself excluded) to a leaf. The properties above ensure that any RBT is balanced enough to have height logarithmic in its size. Intuitively, this is because rule (5) requires that every full path descending from the root contains the same amount of black nodes, corresponding to its black-height, while rule (4) ensures that the total height of the tree is within a constant factor of such black-height. Formally, this is stated by the following lemma.

Lemma *(red-black tree heigth).* A red-black tree with n inner nodes has height $h \leq 2\log_2(n+1)$.

The lemma can be proved by induction on the height of the tree, by showing that every subtree rooted in a node x contains at least $2^{bh(x)} - 1$ inner nodes, where $bh(x)$ is the black-height of the node x.

- *Base case:* when the height of the subtree rooted in x is 0, then x must be a leaf (*NIL*), and so the subtree has no inner node, indeed $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
- *Inductive step:* a node x, whose subtree has positive height, must be an inner node and thus have two children. Each child must have black-height $bh(x)$ or $bh(x) - 1$ depending on whether it is red or black, respectively. Since the height of the subtree rooted in each child is strictly less (at least 1 less) than that of x, by inductive hypothesis, each must have at least $2^{bh(x)-1} - 1$ inner nodes. Then, by adding those inner nodes together with x itself, we get that the subtree rooted in x must have at least $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 2 + 1 = 2^{bh(x)} - 1$ inner nodes.

To conclude the proof we just observe that by rule (4) we know that the black-height of the root must be at least half the height h of the tree, since the number of red nodes along a maximal path cannot be more than the number of black ones. Therefore, we have that $n \geq 2^{h/2} - 1$ and so, after some basic manipulations, $2\log_2(n+1) \geq h$.