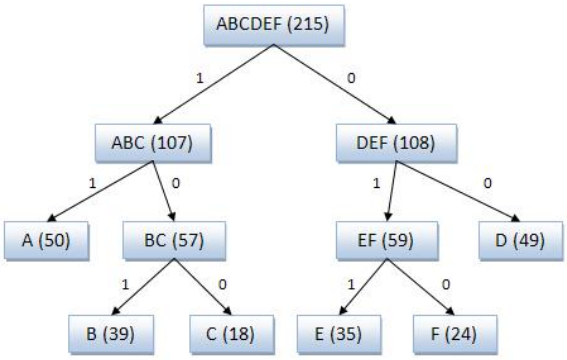
Tutorial #9. Binary search trees

Theoretical part

Binary trees in common

One of the old and very efficient methods of compression for natural language is **prefix code**. This is a code of variable length, in which code for a char will never start with the code for other char. E.g. [0, 11, 101] alphabet is ok, [0, 1, 10] is not ok. **Code trees –** graphical representation of such code. Assign 0/1 to left edge and 1/0 to right. Chars – are always leaves.

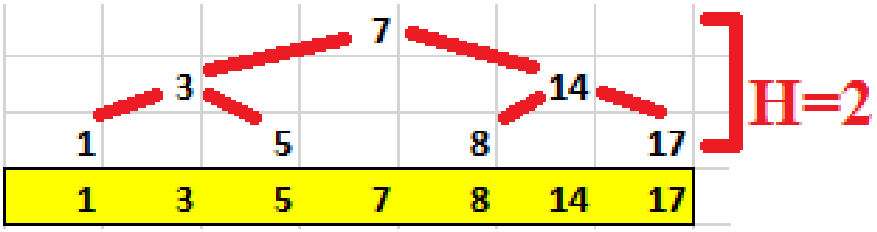


Shannon-Fano coding

Building such a tree is a purpose of algorithms, that base on probability. Most popular codes used even now are Shannon-Fano code and Huffman code.

Binary search tree

BST is a binary tree with a constraint on order: everything on the left is smaller; everything on the right is greater. You can consider BST as pre-calculated binary search middle elements. Just imagine you run binary search algorithm for all possible values and record all middle values in a tree.



Generally BST **depends on the order of element insertion**. Remember **degenerate** example from lecture. "Balanced" property is very important to preserve tree CRUD operations fast. How to do it we will study on future lectures with example of AVL and red-black trees.

Practical part

1) Implement linked binary search tree. Interface includes at least **find**(), **add**(), **delete**(), **size**() methods.
2) Write a method that calculated tree height. Can you do it for O(n)?
3) Create
    a. One BST and build it from your own binary search implementation (put mid-element values in a tree).
    b. Second BST and build it from ordered array.
4) Calculate both tree's height.
5) ***Try to visualize tree somehow:***

```
        5
    3       7
  1   4   6   9
```