Institut für Informatik **Ludwig-Maximilians-Universität München**
Prof. Dr. Francois Bry

Higher level languages: Rust, WS 18/19 **Tutorial 5**

November 14, 2018

Exercise 5-1 Huffman encoding trees

A Huffman code is a type of prefix code that can be used for lossless data compression. Huffman codes are based on variable-length code tables where frequent source symbols are assigned shorter codes and less frequent source symbols are assigned longer codes. In the following we describe binary Huffman codes.

Generating a Huffman encoded string from an input string requires these steps:

- Determine the frequency of the individual characters in the input string.
- Create a single-node tree for every unique character and its frequency in the string.
- Combine two trees which have the least frequency. The frequency of the new tree is the added frequency of its two subtrees.
- Continue combining two trees until there is only one tree left („Huffman tree“).
- The codes 0 and 1 are assigned to the left and right edges in the tree. Traversing the tree from the root node to the leaves yields the encoding for each character. This called the „code book“.
- Encoding a string corresponds to mapping each source character to its code with the help of the code book.

As an example, assume we have to encode the string „CODEBOOK“. This string consists of 6 different characters, so in a standard encoding scheme each character would require 3 bits and the string would take up $3 \cdot 8 = 24$ bits of storage. Given the Huffman code book

```
B: 000, C: 001, D: 010, E: 011, K: 10, O: 11
```

the input string corresponds to the Huffman encoded string `00111010011000111110` and takes up 20 bits of storage.

a) Write a generic function `fn frequency<T>(vec: &Vec<T>) -> BTreeMap<T, u32>` that transforms a vector of elements into a `BTreeMap` containing these elements and their frequency. Determine what traits the type variable `T` must satisfy and specify these.

b) In file `huffman_gap.rs` a type definition `HuffTree` has been given. Complete its implementation by writing the method `fn lettercount(&self) -> u32` which returns the total number of letters a given (partial) Huffman tree represents in the source string. Either derive or implement the traits `Display`, `PartialOrd`, `Ord`, `PartialEq`, and `Eq`.

   The ordering of two (partial) Huffman trees t_1, t_2 should be defined as follows: $t_1 < t_2$ means that the (partial) code for t_1 will be shorter than the (partial) code for t_2. (The characters in t_1 occur more often than the characters in t_2.)

c) Write a function `fn huffman(occ: BTreeMap<char, u32>) -> Option<HuffTree>` that takes a set of characters and their frequency (as implemented in sub-task a) and turns it into a Huffman tree. This task is a bit tricky, so here are a few hints:

- Use the functions `into_iter`, `map`, and `collect` to convert the `BTreeMap` into a `BinaryHeap`. `BinaryHeap` is used as an intermediate data structure which helps in assembling the Huffman tree from its parts.
- The function `BinaryHeap::pop` always removes and returns the maximum element in the `BinaryHeap`. Pushing a new element onto the heap can be done with `BinaryHeap::push`.

d) The function `fn encode(m: &str) -> Option<(Codebook, BitVec)>` generates the Huffman encoded message from the input string and returns the code book and the encoded message as a tuple to the caller. Implement this function.