FPV Tutorübung

Woche 6

Ocaml: List-Module, Binary Search Trees

Manuel Lerchner

31.05.2023

https://github.com/ManuelLerchner/fpv-tutorial-SS23

T01: Explicit Type Annotation

In OCaml, types are inferred automatically, so there is no need to write them down explicitly. However, types can be annotated by the programmer. Discuss:

1. In the following expression, annotate the types of all subexpressions:

```
let f = fun x y -> x, [y]
```

2. When can explicitly annotated types be helpful?

T02: The List Module

Check the documentation of the OCaml `List` module here and find out what the following functions do. Then implement them yourself. Make sure your implementations have the same type. In cases where the standard functions throw exceptions, you may just `failwith "invalid"`.

1. ⊗ **hd** 0 of 1 tests passing
   Implement the function `hd`
2. ⊗ **tl** 0 of 1 tests passing
   Implement the function `tl`
3. ⊗ **length** 0 of 1 tests passing
   Implement the function `length`
4. ⊗ **append** 0 of 1 tests passing
   Implement the function `append`
5. ⊗ **rev** 0 of 1 tests passing
   Implement the function `rev`
6. ⊗ **nth** 0 of 1 tests passing
   Implement the function `nth`

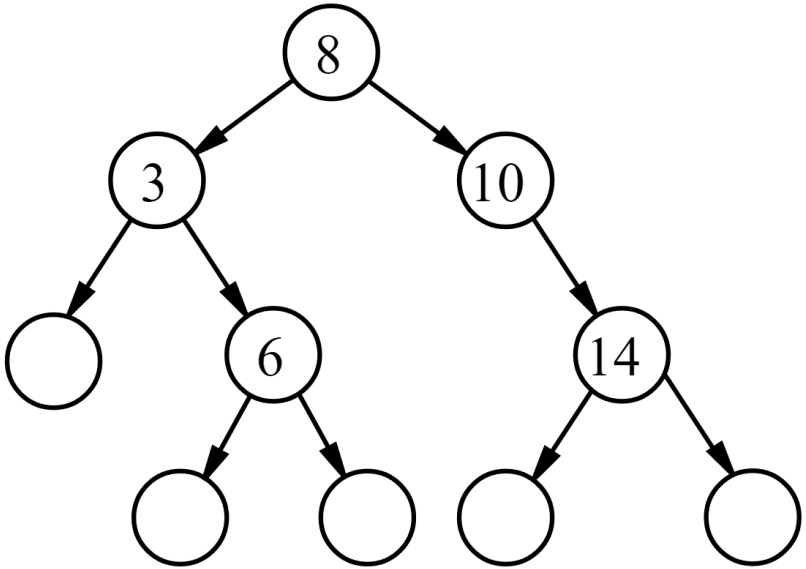
T03: Binary Search Tree 1

In this assignment, a collection to organize integers shall be implemented via binary search trees.

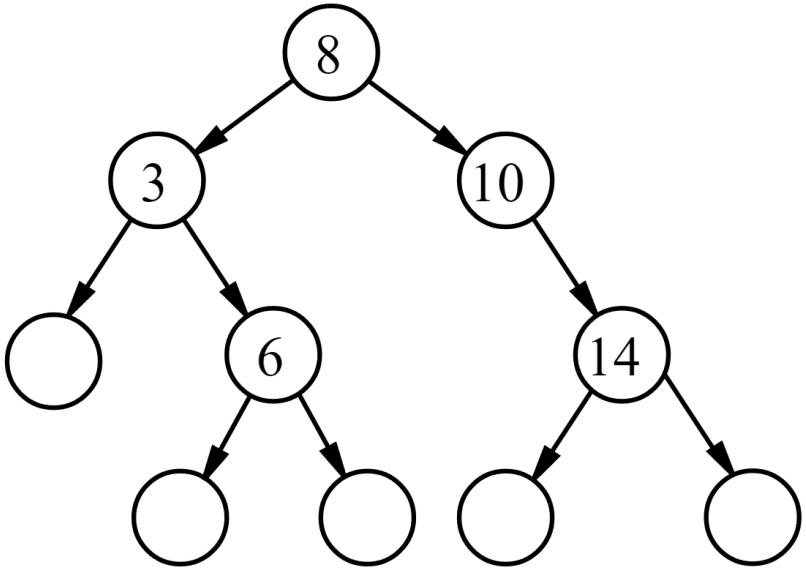
1. Define a suitable data type `tree` for binary trees that store integers. Each node in the binary tree should either be
   - an inner node which stores a value of type `int` and has a left and a right child of type `tree`, or
   - a leaf node and contain no value.

Since you are free to define your `tree` type however you wish (the type `tree` is said to be *abstract*), we need to define functions for creating trees.

2. Define functions `node` and `leaf`, which should create inner nodes and leaves, respectively:
   - `node v l r` should create an inner node with the value `v`, left child `l` and right child `r`,
   - `leaf ()` should create an leaf node of your `tree` type.



T03: Binary Search Tree 2



Similarly, we need a function that allows us to inspect the structure of your tree, specifically to access the children of a node. The OCaml `Option` type (API documentation for Option) allows us to cleanly distinguish between the presence of absence of a value. Here, we will use it to distinguish between inner nodes, which have children, and leaf nodes, which do not. Using the `Option` type and returning `None` instead of raising an exception is (often) good functional programming style!

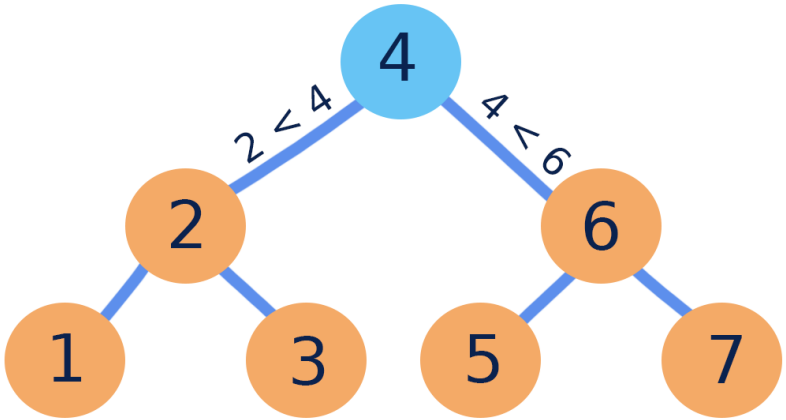
3. Define the function `inspect`, which allows us to access the children of a node:
   - `inspect n`, where `n` is an inner node of your `tree` type with the value `v` and children `l` and `r`, should return `Some (v, l, r)`. Here, `Some` is used to indicate the *presence* of a value and children.
   - `inspect l`, where `l` is a leaf of your `tree` type (with no children), should return `None`. Here, `None` is used to indicate the *absence* of a value and children.

T03: Binary Search Tree 3

4. Define a binary tree `t1` which contains the values $8, 12, 42, 1, 6, 9, 8$. To construct the tree, start with an empty tree, then insert the given values in order.

T03: Binary Search Tree 4

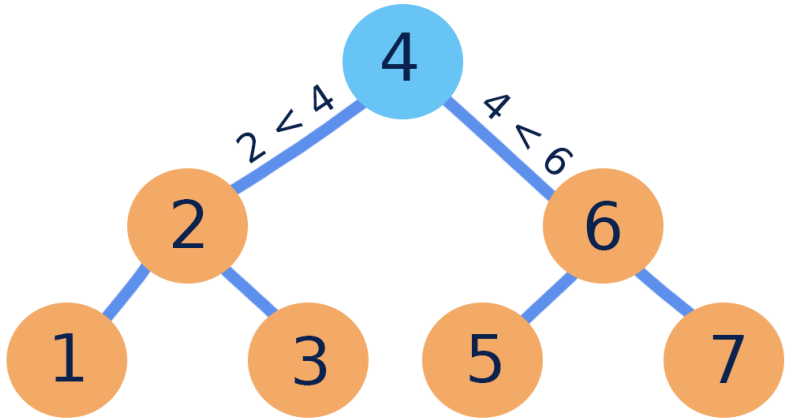
5. Implement a function `to_list : tree -> int list` that returns an ordered list of all values in the tree.



In Order Traversal: [1 2 3 4 5 6 7]

T03: Binary Search Tree 5

6. Implement a function `insert : int -> tree -> tree` which inserts a value into the tree. If the value exists already, the tree is not modified.



T03: Binary Search Tree 6

7. Implement a function `remove : int -> tree -> tree` to remove a value (if it exists) from the tree. Upon removing a value from an inner node:
    - if either child of the inner node is empty, replace the entire inner node with the other child, otherwise
    - if neither child of the inner node is empty, the value should be replaced with the largest value from the *left* subtree.