Agda from Nothing

Order in the Types

Agda

- Agda is a **dependently typed functional programming language**.
  - It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length.
  - It also has parameterised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.
- Agda is a **proof assistant**.
  - It is an interactive system for writing and checking proofs.
  - Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf.
  - It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

Source: The Agda Wiki http://wiki.portal.chalmers.se/agda/pmwiki.php

How to Keep Your Neighbours in Order

Conor McBride
University of Strathclyde
Conor.McBride@strath.ac.uk

Abstract

I present a datatype-generic treatment of recursive container types whose elements are guaranteed to be stored in increasing order, with the ordering invariant rolled out systematically. Intervals, lists and binary search trees are instances of the generic treatment. On the journey to this treatment, I report a variety of failed experiments and the transferable learning experiences they triggered. I demonstrate that a *total* element ordering is enough to deliver insertion and flattening algorithms, and show that (with care about the formulation of the types) the implementations remain as usual. Agda's *instance arguments* and *pattern synonyms* maximize the proof search done by the typechecker and minimize the appearance of proofs in program text, often eradicating them entirely. Generalizing to indexed recursive container types, invariants such as *size* and *balance* can be expressed in addition to *ordering*. By way of example, I implement insertion and deletion for 2-3 trees, ensuring both order and balance by the discipline of type checking.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords dependent types; Agda; ordering; balancing; sorting

1. Introduction

- a precise implementation of insertion and deletion for 2-3 trees

I take the time to explore the design space, reporting a selection of the wrong turnings and false dawns I encountered on my journey to these results. I try to extrapolate transferable design principles, so that others in future may suffer less than I.

2. How to Hide the Truth

If we intend to enforce invariants, we shall need to mix a little bit of logic in with our types and a little bit of proof in with our programming. It is worth taking some trouble to set up our logical apparatus to maximize the effort we can get from the computer and to minimize the textual cost of proofs. We should prefer to encounter logic only when it is dangerously absent!

Our basic tools are the types representing falsity and truth by virtue of their number of inhabitants:

```
data 0 : Set where                          -- no constructors!
record 1 : Set where constructor ⟨⟩         -- no fields!
```

Dependent types allow us to compute sets from data. E.g., we can represent evidence for the truth of some Boolean expression which we might have tested.

```
data 2 : Set where tt ff : 2
So : 2 → Set
```

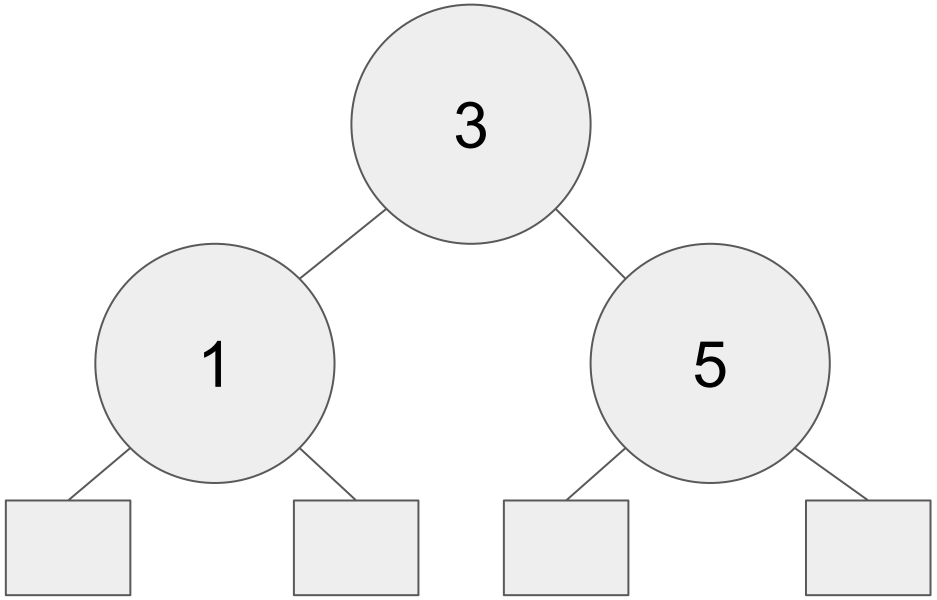
In Partes Tres

1. Basics, working toward BST
2. Bounded BSTs
3. 2-3 Trees

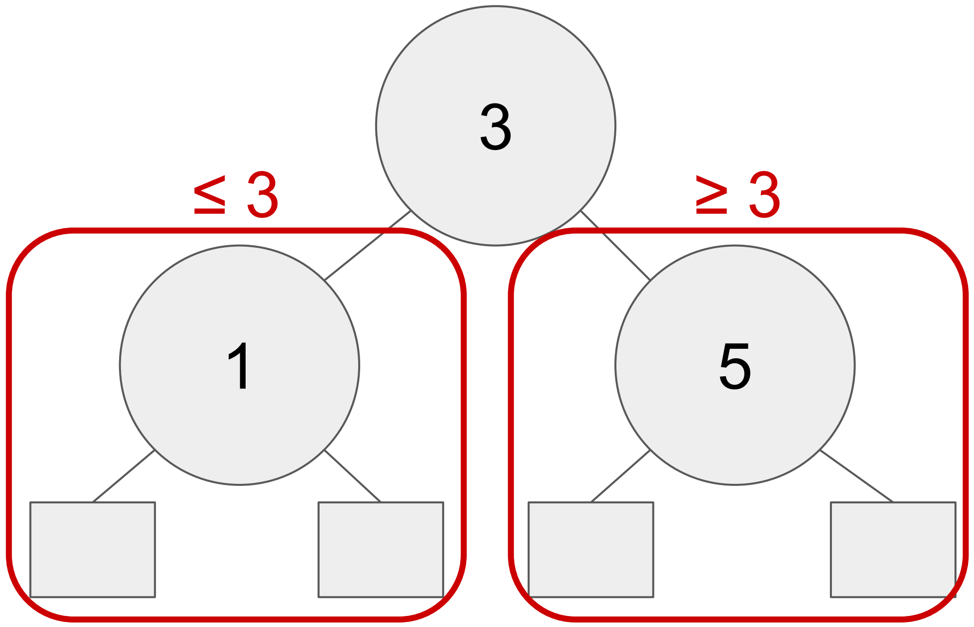
Part 1

Binary Search Trees

Binary Search Trees



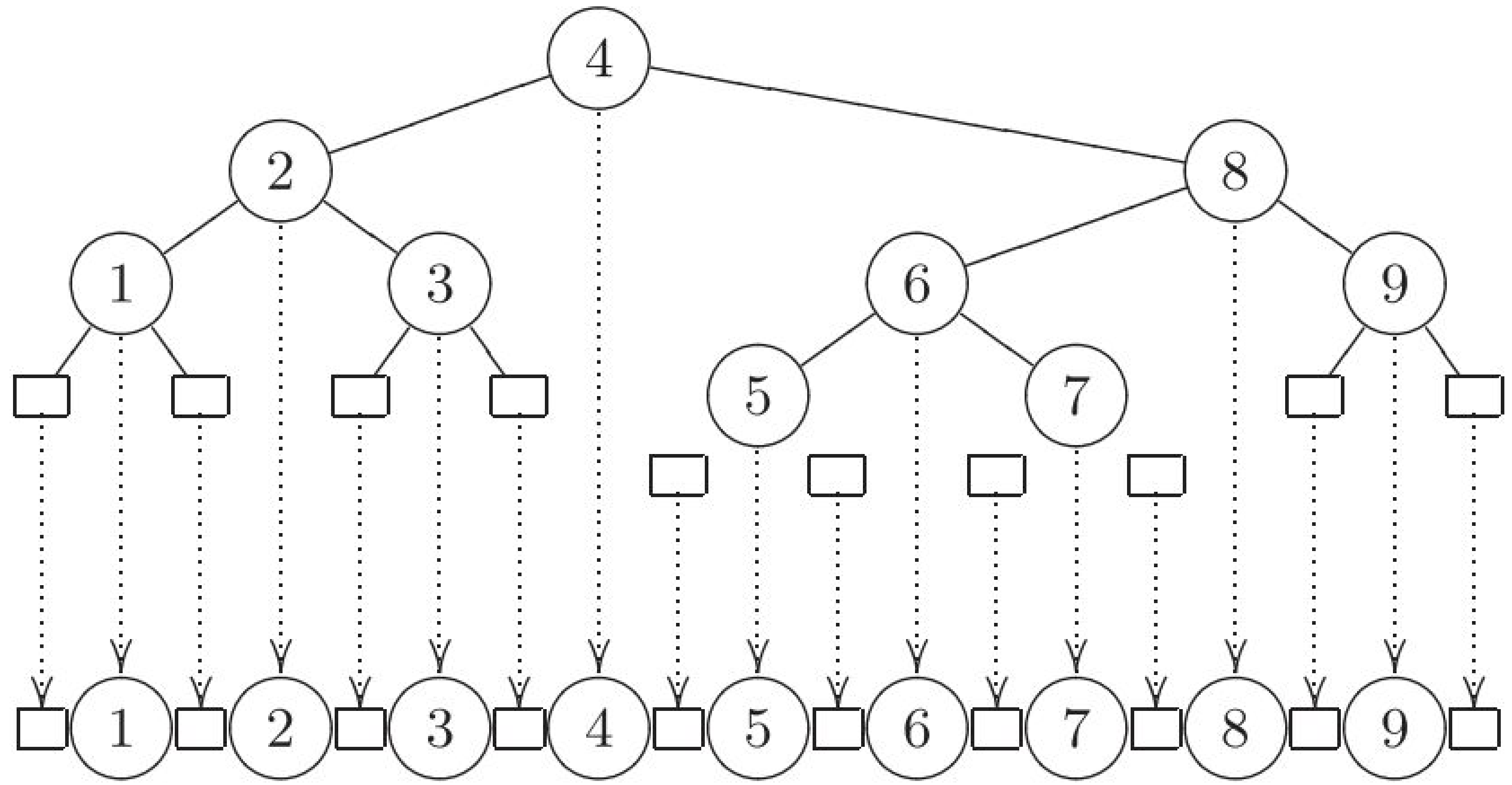
BST Invariants



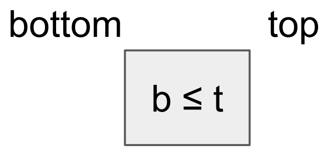
Part 2

Bounded BSTs

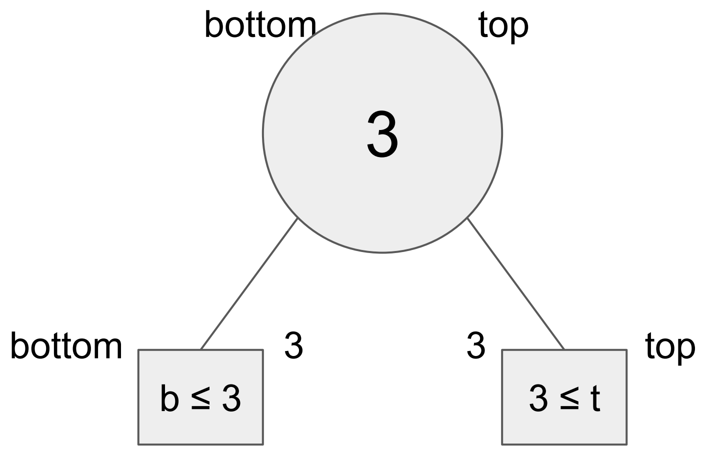
Leaf-Node-Leaf-Node-Leaf



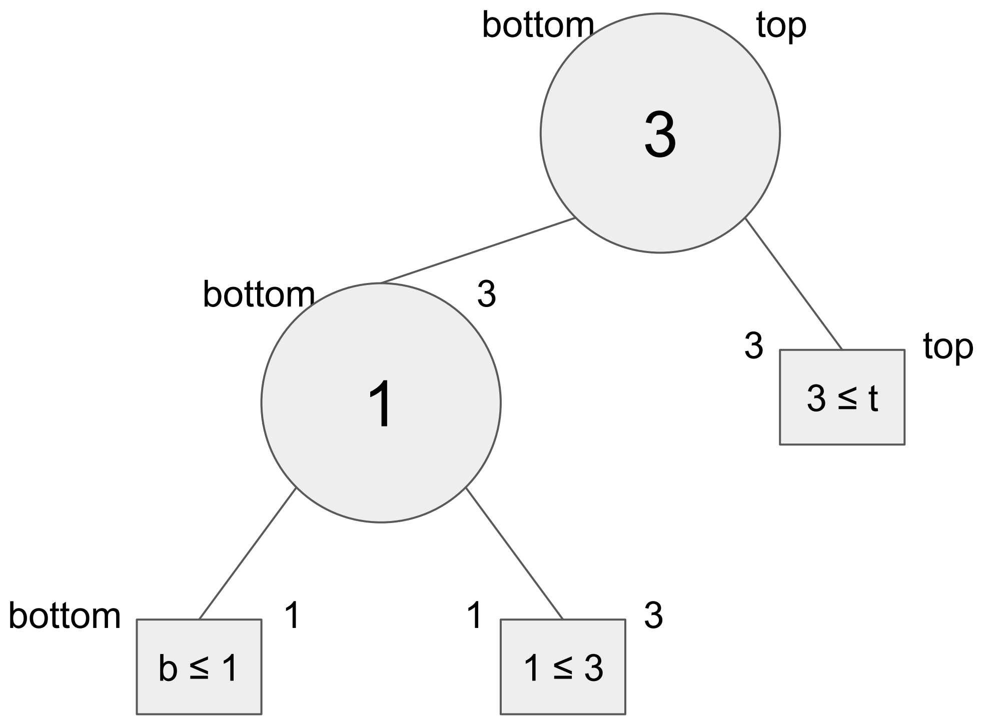
Bounded BST (Empty)



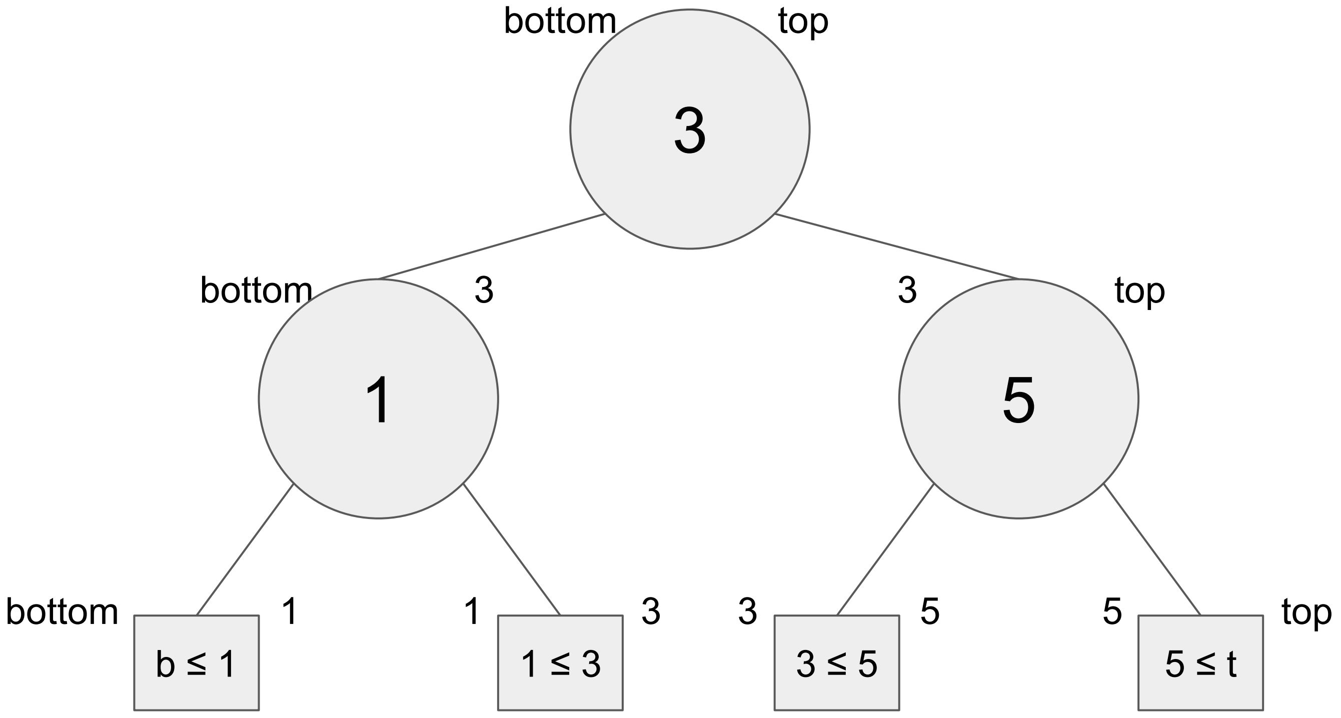
Bounded BST



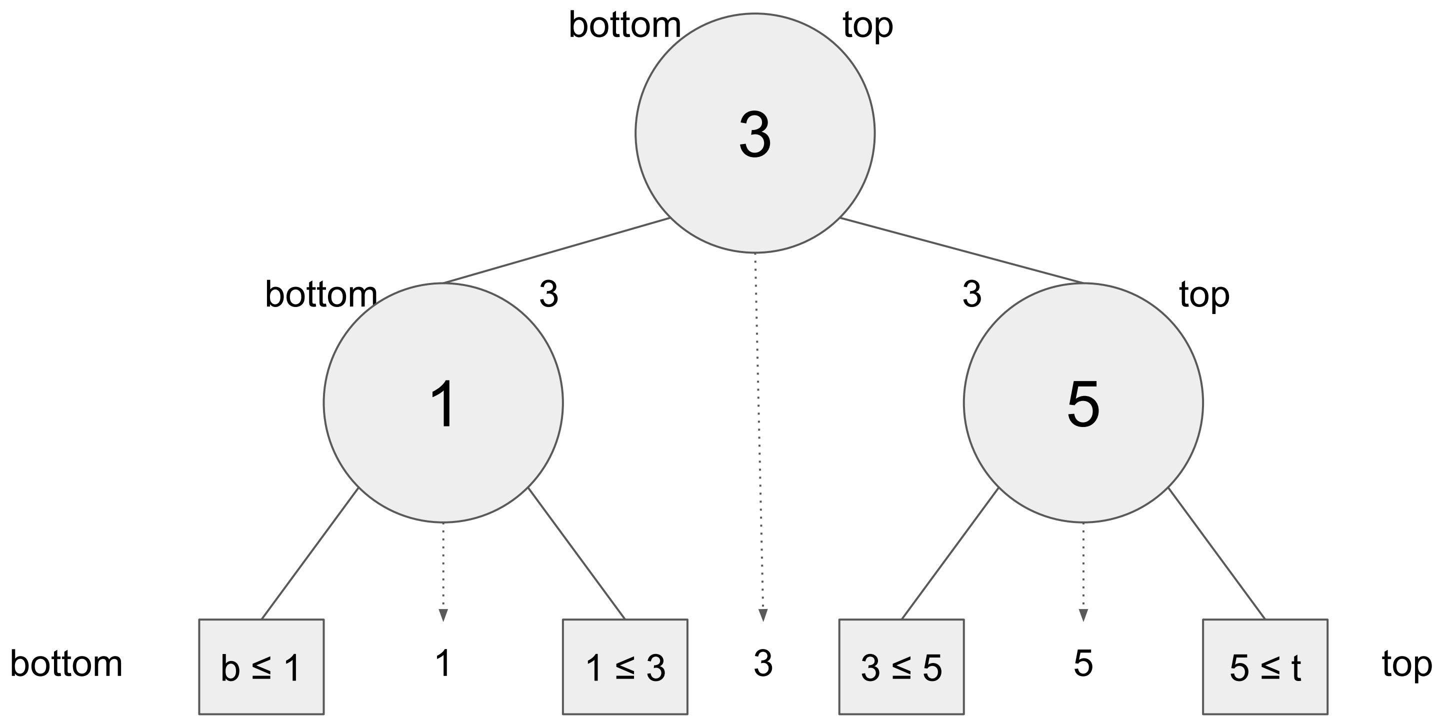
Bounded BST



Bounded BST

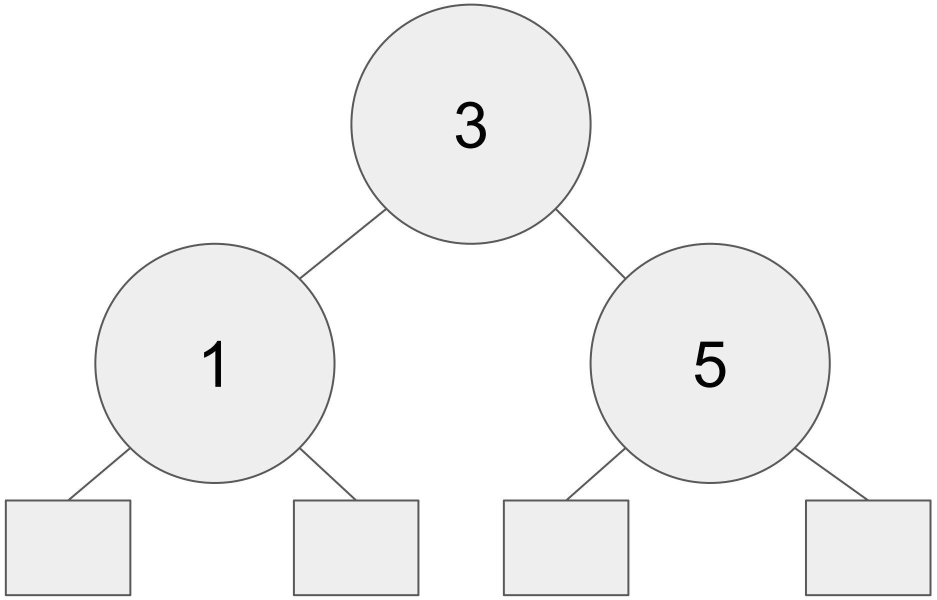


Bounded BST

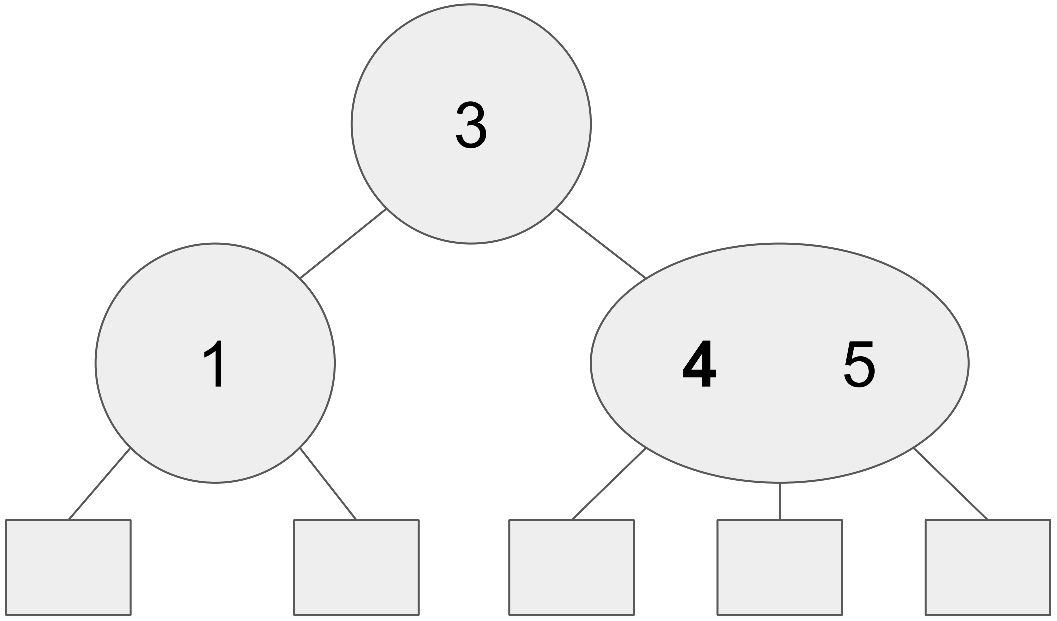


Part 3
2-3 Trees

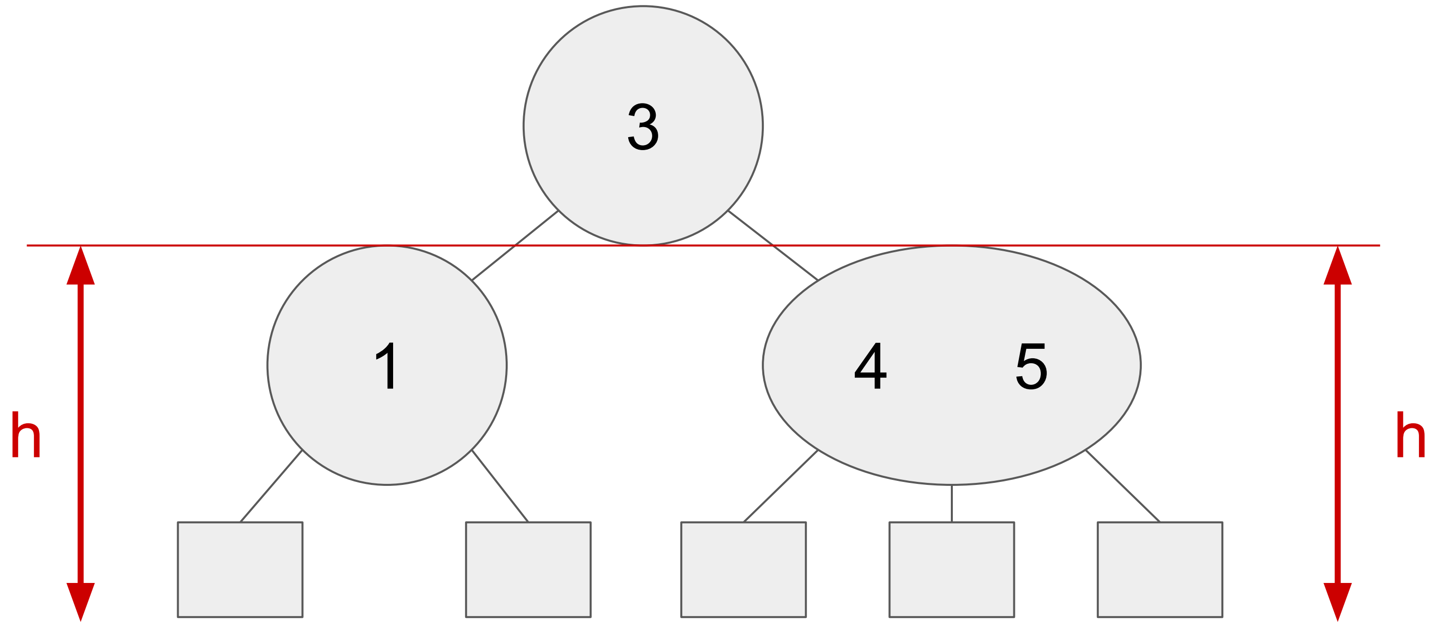
2-3 Tree



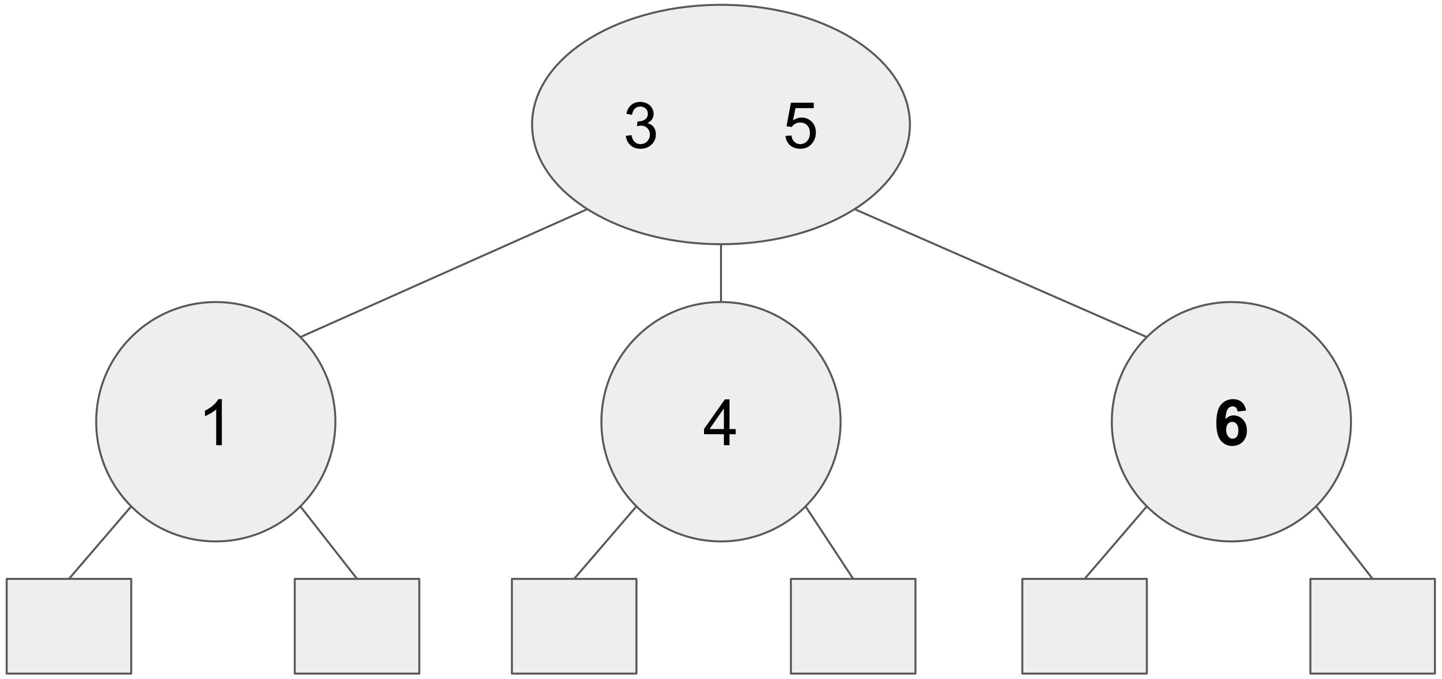
2-3 Tree



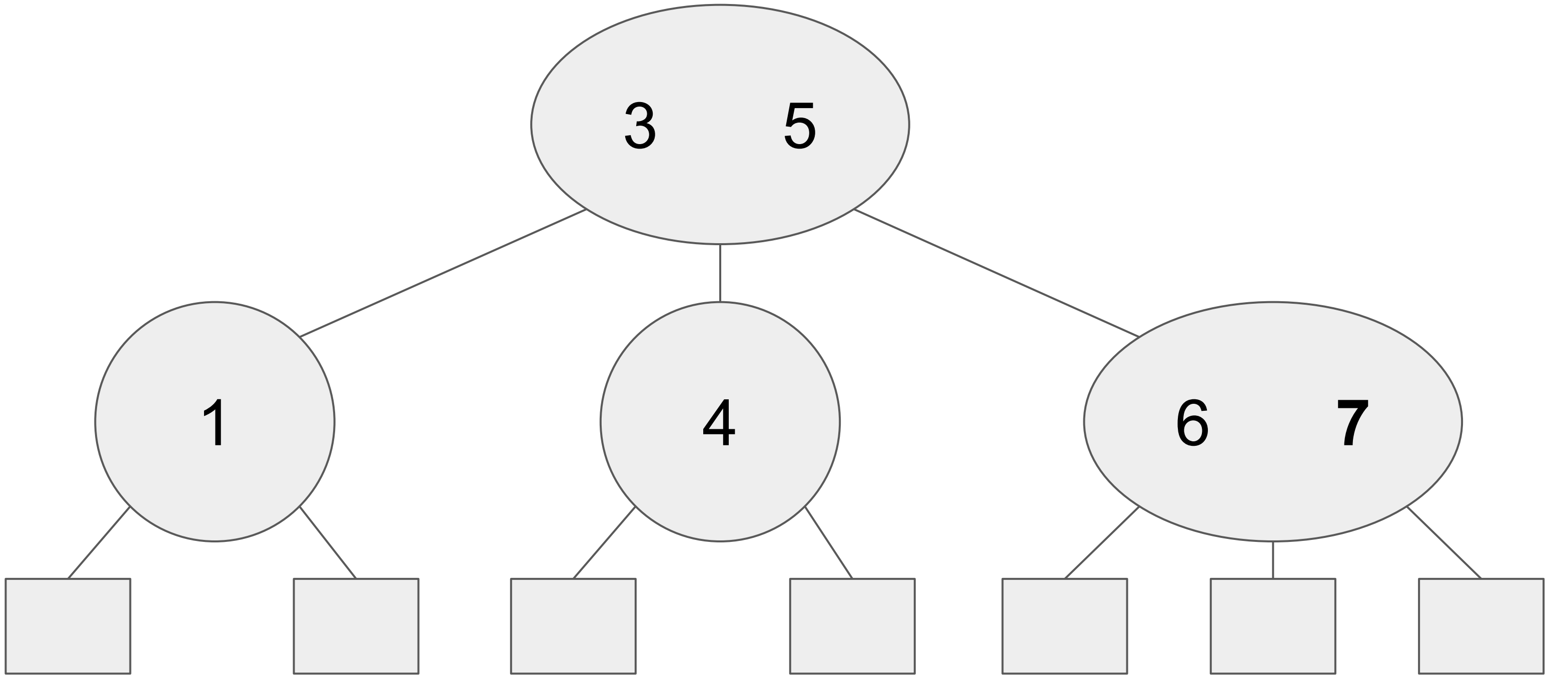
2-3 Tree Invariant



2-3 Tree



2-3 Tree



2-3 Tree