

Introduction to SML

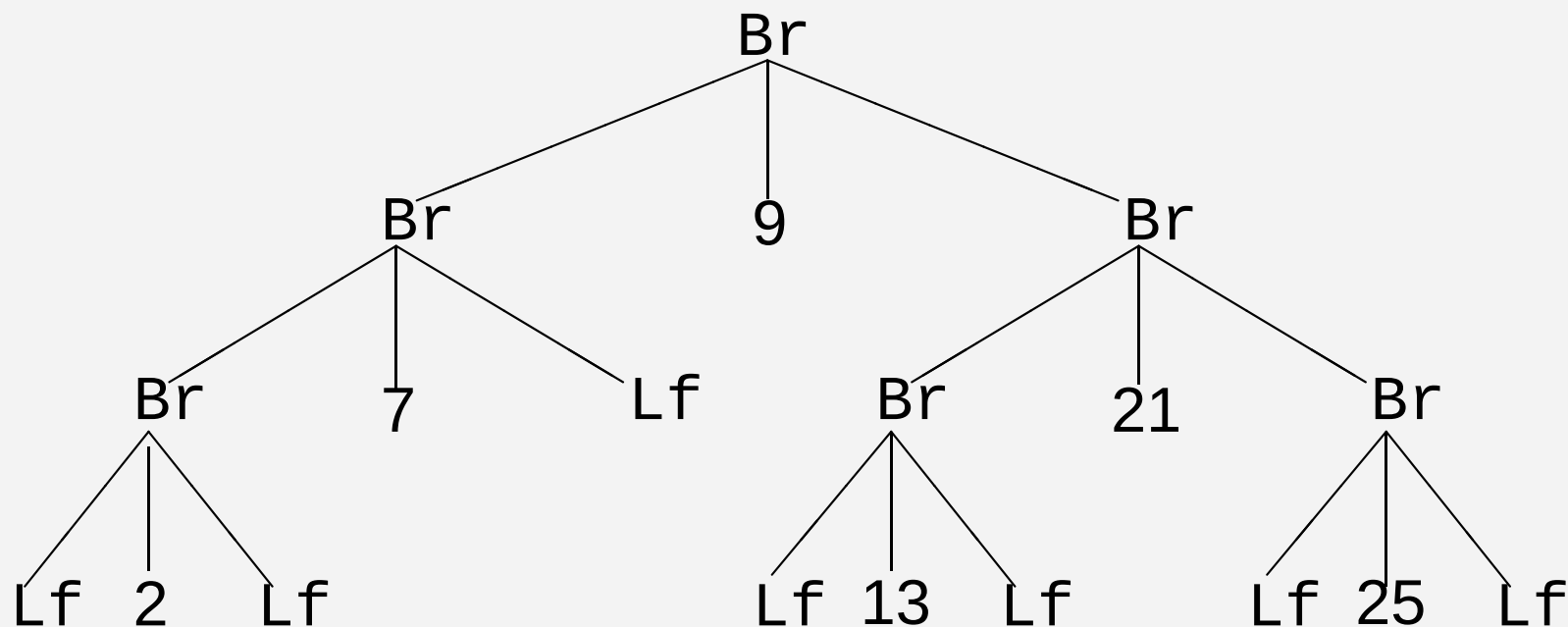
AVL Trees

Michael R. Hansen

`mrh@imm.dtu.dk`

Informatics and Mathematical Modelling
Technical University of Denmark

Binary Search Trees



Condition: for every node containing the value x : every value in the left subtree is smaller than x , and every value in the right subtree is greater than x .

SML Declaration of Binary Trees

```
datatype tree = Lf | Br of tree*int*tree;
```

The tree from the previous slide is denoted by:

```
Br(Br(Br(Lf, 2, Lf), 7, Lf),  
    9,  
    Br(Br(Lf, 13, Lf), 21, Br(Lf, 25, Lf)))
```

The function

```
fun member(i, Lf) = false  
  | member(i, Br(t1, j, t2)) =  
      case Int.compare(i, j) of  
          EQUAL => true  
        | LESS   => member(i, t1)  
        | GREATER => member(i, t2)  
> val member = fn : int * tree -> bool
```

has a linear worst-case running time

Balanced binary search trees: AVL trees

An **AVL tree** (after Adelson-Velskii and Landis) is a search tree satisfying the following **balancing condition**:

- for every branching node: the difference of the heights of the left and right subtrees should not be greater than 1.

Balanced binary search trees: AVL trees

An **AVL tree** (after Adelson-Velskii and Landis) is a search tree satisfying the following **balancing condition**:

- for every branching node: the difference of the heights of the left and right subtrees should not be greater than 1.

AVL trees are generated from the empty tree L_f using insertion and deletion operations.

Balanced binary search trees: AVL trees

An **AVL tree** (after Adelson-Velskii and Landis) is a search tree satisfying the following **balancing condition**:

- for every branching node: the difference of the heights of the left and right subtrees should not be greater than 1.

AVL trees are generated from the empty tree L_f using insertion and deletion operations.

Insertion and deletion must **preserve** balancing condition

Auxiliary functions

The **height** of a tree

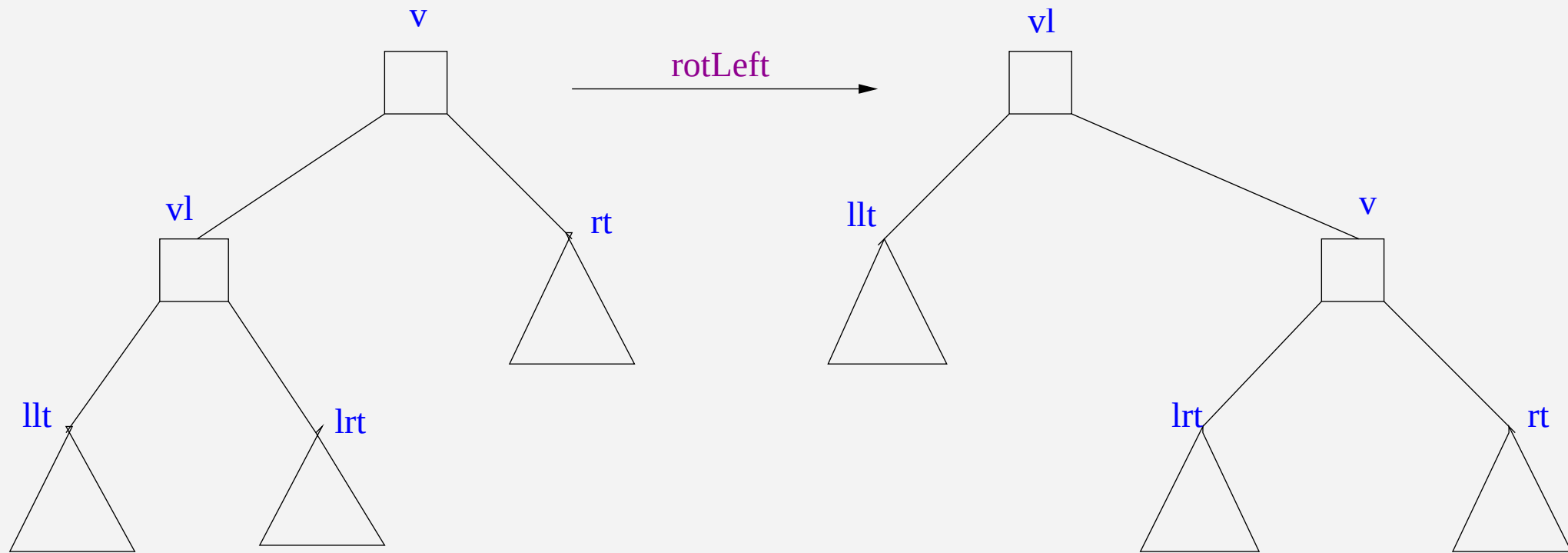
```
fun height Lf                = 0
  | height(Br(lt, _, rt)) =
      1 + Int.max(height lt, height rt);
```

is computed in linear time.

A function to test whether a search tree is an AVL tree:

```
fun isAVL Lf                = true
  | isAVL(Br(tl, v, tr)) =
      isAVL tl andalso isAVL tr
      andalso abs(height tl - height tr) <= 1;
```

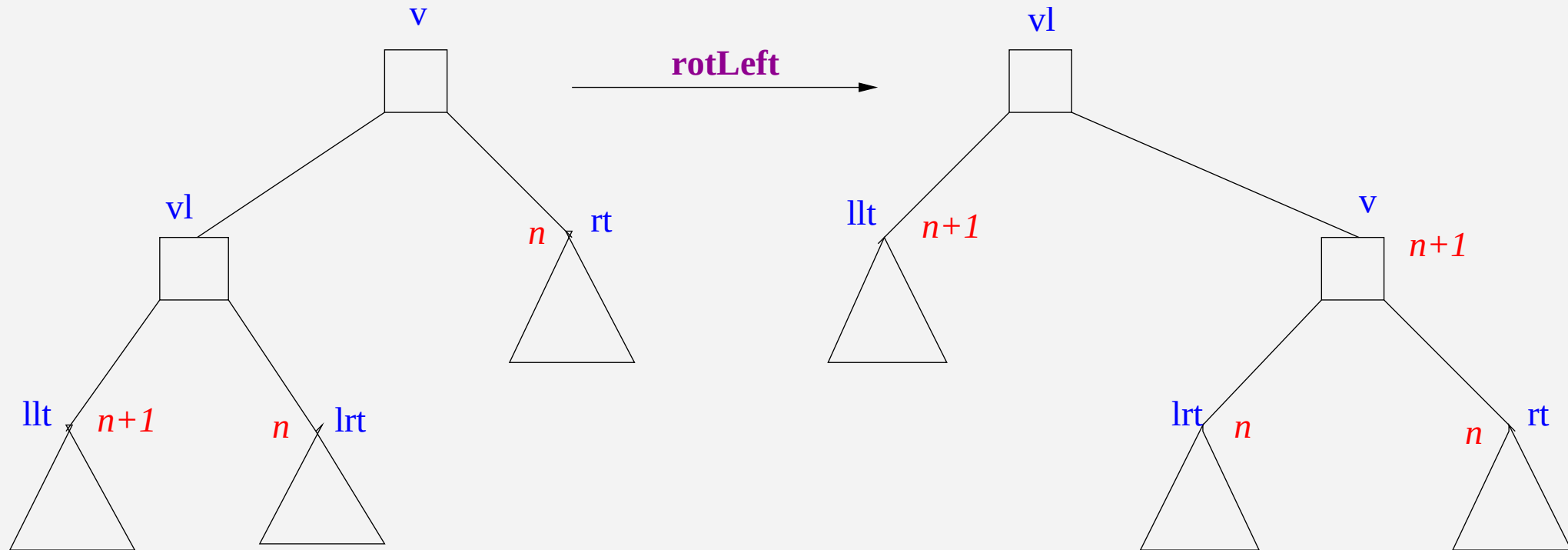
Rotate from Left to Right: 1



Preserves the search tree condition

Rotate from Left to Right: 2

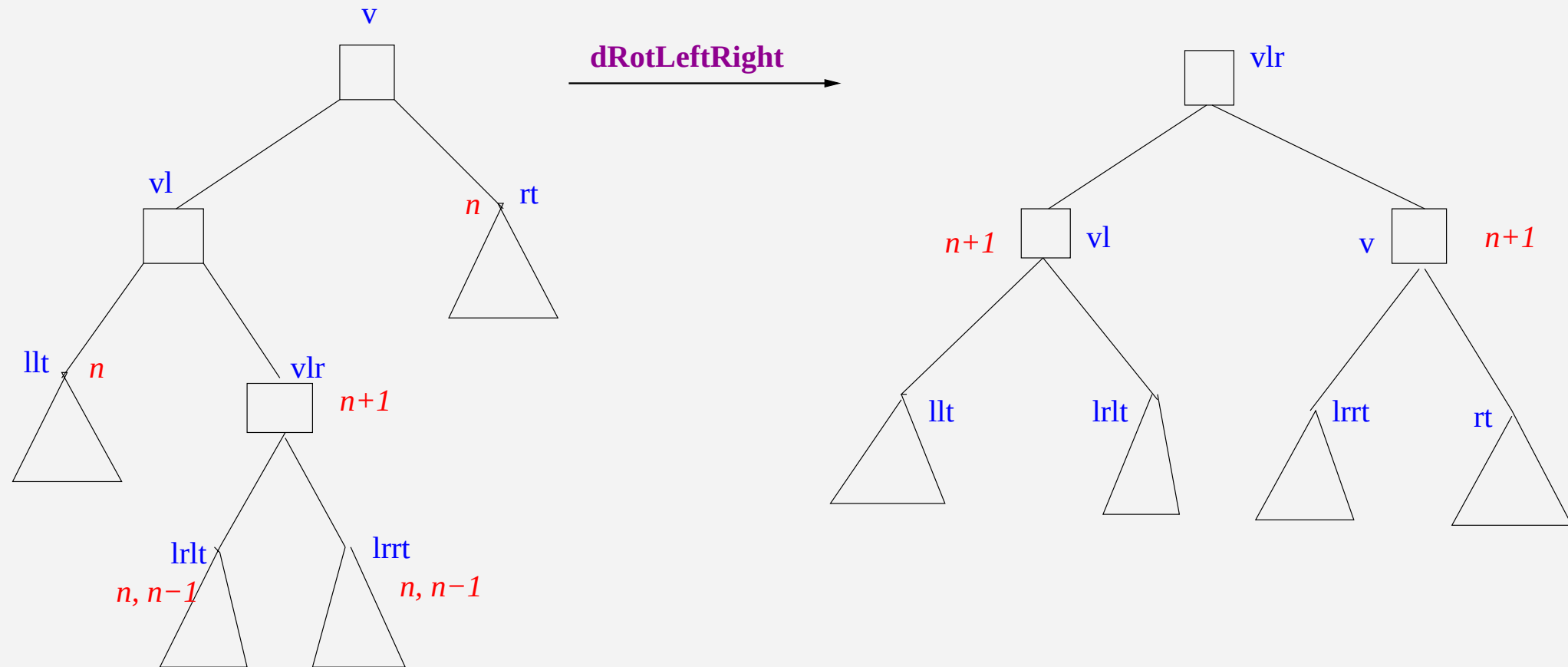
Taking **heights** into account:



Establishes AVL condition in some cases

```
fun rotLeft(Br(Br(llt, vl, lrt), v, rt))  
            = Br(llt, vl, Br(lrt, v, rt));
```

Double Rotation from Left to Right



In SML

```
fun dRotLeftRight(Br(Br(llt,vl,Br(lrlt,vlr,lrrt)),v,rt)
  = Br(Br(llt,vl,lrlt),vlr,Br(lrrt,v,rt))
```

Insertion in AVL Trees

```
fun insert(i, Lf) = Br(Lf, i, Lf)
  | insert(i, t as Br(lt, v, rt)) =
    case Int.compare(i, v) of
      LESS =>
        let val nlt as Br(_, nv1, _) = insert(i, lt)
            val nt = Br(nlt, v, rt)
        in if height nlt - height rt = 2
            then if i < nv1 then rotLeft nt
                  else dRotLeftRight nt
            else nt
        end
      | EQUAL => t
      | GREATER => ... SIMILAR ...
```

Property: If t is an AVL tree, then so is $\text{insert}(i, t)$

Worst-case running time?

Height information in nodes

AVL trees with **height** information in nodes

```
datatype tree = Lf   | Br of tree * int * int * tree;
```

Computation of height in **constant time**

```
fun height Lf           = 0
    | height(Br(_, _, h, _)) = h;
```

Rotations must maintain height information, e.g.

```
fun rotLeft(Br(Br(l1t, v1, h, lrt), v, _, rt)) =
    Br(l1t, v1, h, Br(lrt, v, h-1, rt));
```

Insertion and deletions with logarithmic worst-case running time.

Programs on the course homepage.