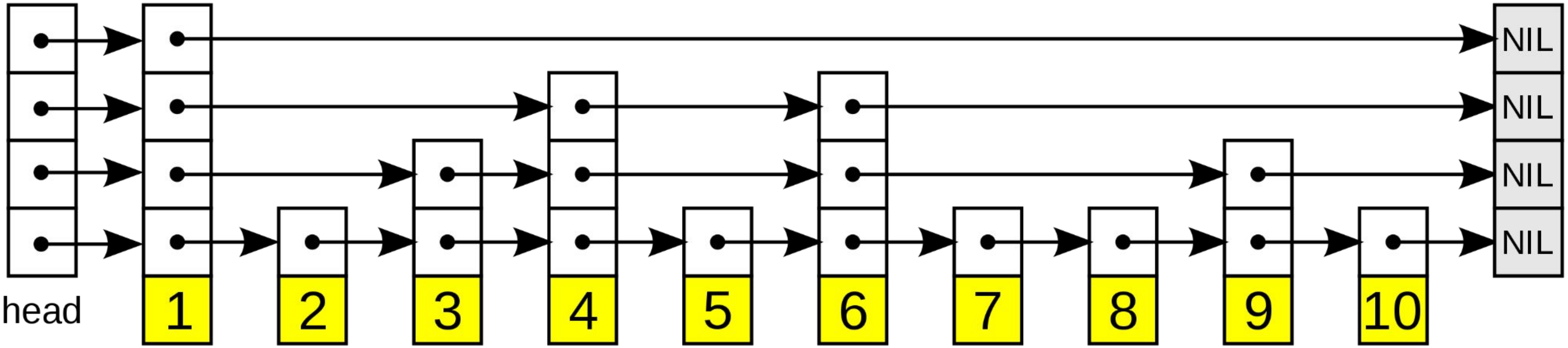
Implementation and Formal Verification of a *functional* SkipList

Lepeytre Hugo, Juillard Paul, Rocher Tanguy

CS550 - EPFL

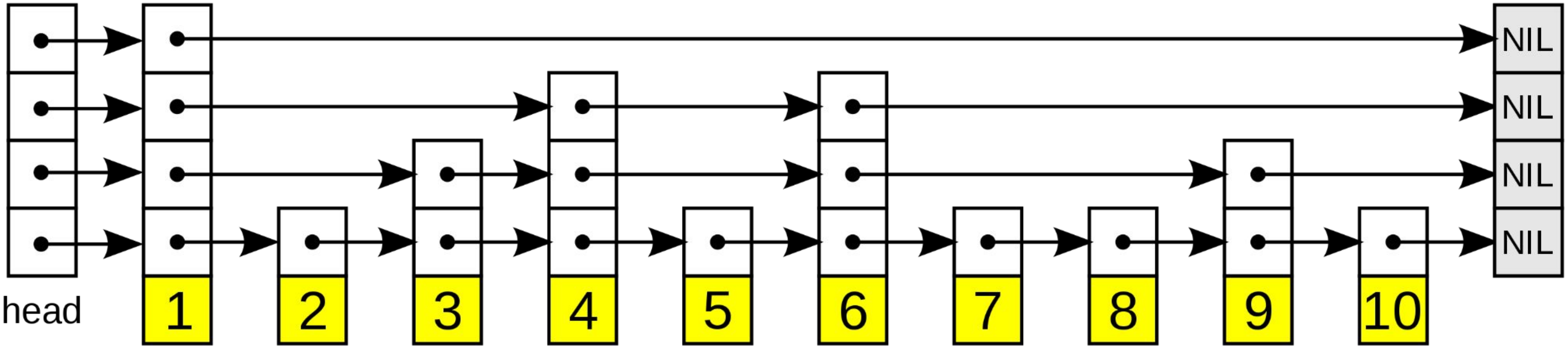
What is a SkipList

- Store an ordered sequence of n elements
- Insertion, removal is O(log n)
- Search is O(log n)



What is a SkipList

- The SkipList is organised in levels
- Each level i is itself an ordered list
- Each level i is a subset of level i-1
- Each node consists in :
    - A value
    - A pointer on the next node on level i
    - A pointer to a node with the same value on level i-1 if i>0



Abstract

SkipLists and implementations have been formally verified:

- leveraging imperative constructs and memory layouts etc.
- by constructing a complex signature space in Presburger arithmetic and verified in Coq

Our work:

- functional implementation of skiplists
- formal verification of a functional implementation
- verified with stainless, without the need to develop the formal space

Signature	Interpretation
Σ_{level}	• $0^{\mathcal{A}} = 0$ • $s^{\mathcal{A}}(l) = s(l)$, for each $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{ord}	• $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} x \rightarrow x = y$ • $x \preceq^{\mathcal{A}} y \vee y \preceq^{\mathcal{A}} x$ • $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} z \rightarrow x \preceq^{\mathcal{A}} z$ • $-\infty^{\mathcal{A}} \preceq^{\mathcal{A}} x \wedge x \preceq^{\mathcal{A}} +\infty^{\mathcal{A}}$ for any $x, y, z \in \mathcal{A}_{\mathsf{ord}}$
Σ_{array}	• $A[l]^{\mathcal{A}} = A(l)$ • $A\{l \leftarrow a\}^{\mathcal{A}} = B$, where $B(l) = a$ and $B(i) = A(i)$ for $i \neq l$ for each $A, B \in \mathcal{A}_{\mathsf{array}}$, $l \in \mathcal{A}_{\mathsf{level}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$
Σ_{cell}	• $mkcell^{\mathcal{A}}(e, k, A, l) = \langle e, k, A, l \rangle$ • $error^{\mathcal{A}}.arr^{\mathcal{A}}(l) = null^{\mathcal{A}}$ • $\langle e, k, A, l \rangle.data^{\mathcal{A}} = e$ • $\langle e, k, A, l \rangle.key^{\mathcal{A}} = k$ • $\langle e, k, A, l \rangle.arr^{\mathcal{A}} = A$ • $\langle e, k, A, l \rangle.max^{\mathcal{A}} = l$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $k \in \mathcal{A}_{\mathsf{ord}}$, $A \in \mathcal{A}_{\mathsf{array}}$, and $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{mem}	• $rd(m, a)^{\mathcal{A}} = m(a)$ • $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$ • $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$
Σ_{reach}	• $\epsilon^{\mathcal{A}}$ is the empty sequence • $[a]^{\mathcal{A}}$ is the sequence containing $a \in \mathcal{A}_{\mathsf{addr}}$ as the only element • $([a_1 \,..\, a_n], [b_1 \,..\, b_m], [a_1 \,..\, a_n, b_1 \,..\, b_m]) \in append^{\mathcal{A}}$ iff $a_k \neq b_l$. • $(m, a_{init}, a_{end}, l, p) \in reach^{\mathcal{A}}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [a_1 \,..\, a_n]$ (c) $m(a_r).arr^{\mathcal{A}}(l) = a_{r+1}$, for $r < n$ (b) $a_1 = a_{init}$ (d) $m(a_n).arr^{\mathcal{A}}(l) = a_{end}$
Σ_{bridge}	for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level}}$, $a_i, a_e \in \mathcal{A}_{\mathsf{addr}}$, $r \in \mathcal{A}_{\mathsf{set}}$ • $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$ • $addr2set^{\mathcal{A}}(m, a, l) = \{a' \mid \exists p \in \mathcal{A}_{\mathsf{path}} \,.\, (m, a, a', l, p) \in reach^{\mathcal{A}}\}$ • $getp^{\mathcal{A}}(m, a_i, a_e, l) = p$ if $(m, a_i, a_e, l, p) \in reach^{\mathcal{A}}$, and ϵ otherwise • $ordList^{\mathcal{A}}(m, p)$ iff $p = \epsilon$ or $p = [a]$, or $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and $m(a_j).key^{\mathcal{A}} \preceq m(a_{j+1}).key^{\mathcal{A}}$ for all $1 \leq j < n$, for any $m \in \mathcal{A}_{\mathsf{mem}}$ • $skiplist^{\mathcal{A}}(m, r, l, a_i, a_e)$ iff $\left[\begin{array}{c} ordList^{\mathcal{A}}(m, getp^{\mathcal{A}}(m, a_i, a_e, 0)) \quad \wedge \\ r = addr2set^{\mathcal{A}}(m, a_i, 0) \quad \wedge \\ 0 \leq l \wedge \forall a \in r \,.\, m(a).max^{\mathcal{A}} \leq l \quad \wedge \\ m(a_e).arr^{\mathcal{A}}(l) = null^{\mathcal{A}} \quad \wedge \\ (0 = l) \vee \\ (\exists l_p \,.\, s^{\mathcal{A}}(l_p) = l \wedge \forall i \in 0, \ldots, l_p \,. \\ m(a_e).arr^{\mathcal{A}}(i) = null^{\mathcal{A}} \wedge \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a_i, a_e, s^{\mathcal{A}}(i))) \subseteq \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a_i, a_e, i))) \end{array}\right]$

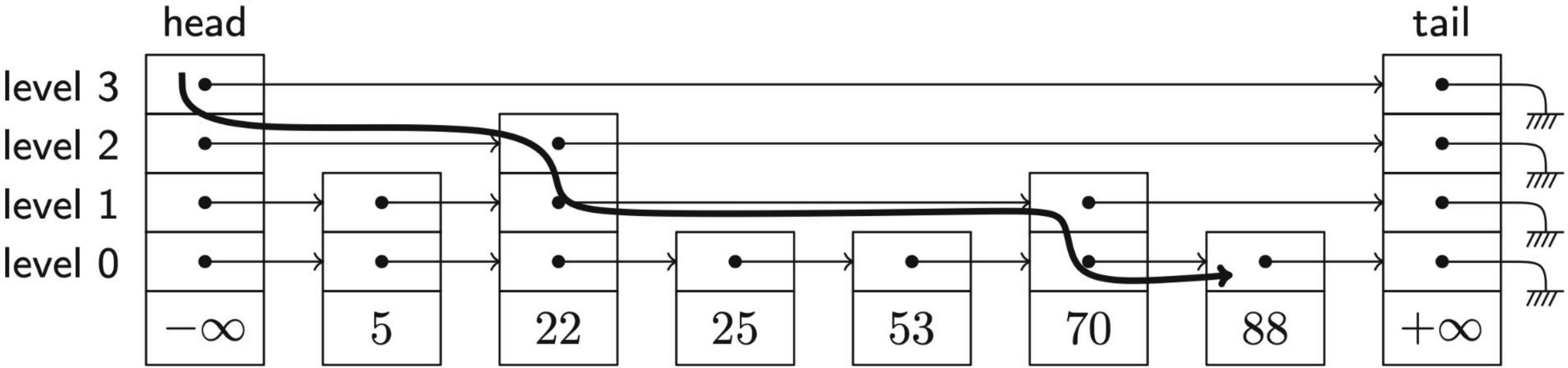
Fig. 4. Characterization of a TSL-interpretation $\mathcal{A}$

```
ensuring (_ => levelLeftmost(top, level).hasValue(Int.MinValue))
```

Operations

Search :

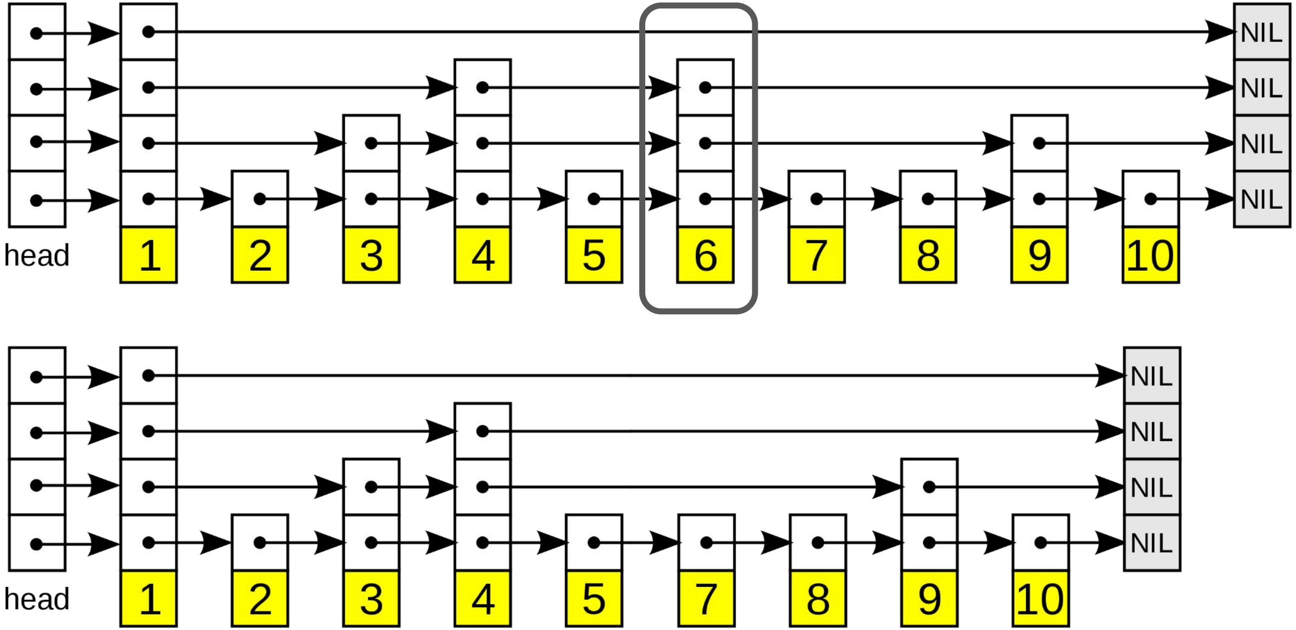
- Start at the top level
- either further right or lower



example: search(88)

Removing :

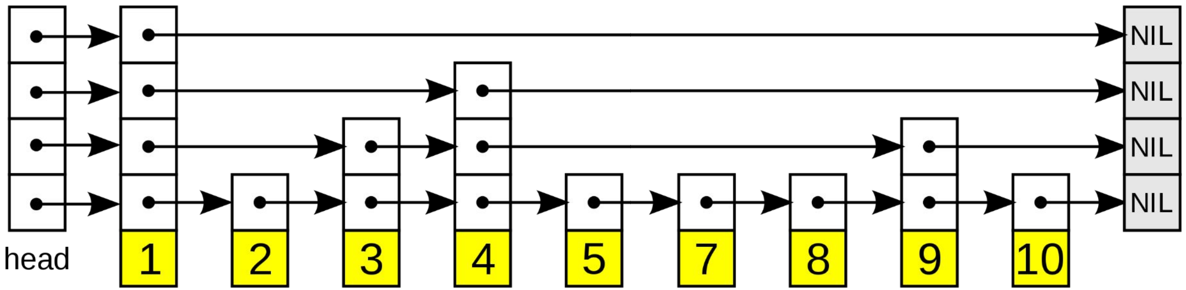
- Removing a value consists in removing all nodes containing this value
- Pointers have to be reassigned to skip the removed value

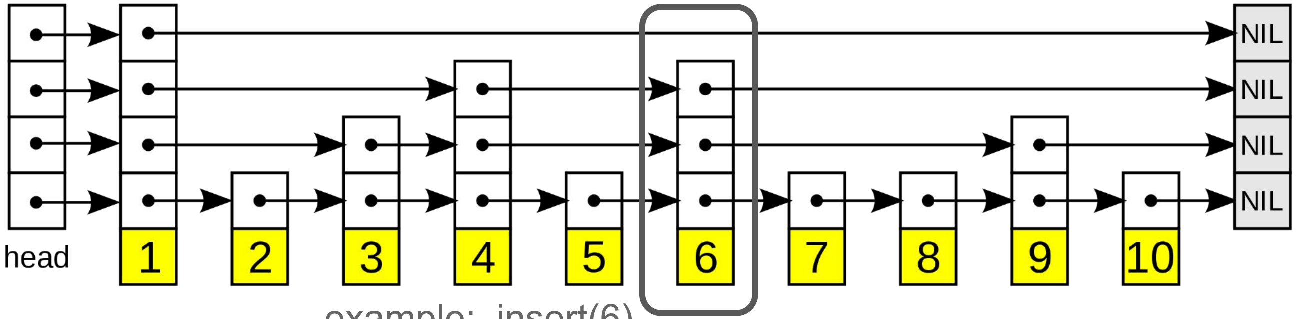


example: remove(6)

Inserting:

- To insert, we first need to pick at random a level i (which can be higher than the actual max level)
- And then insert a that value in each level smaller than or equal to i





example: insert(6)

Our implementation:
structure and properties

Our Implementation of a SkipList :

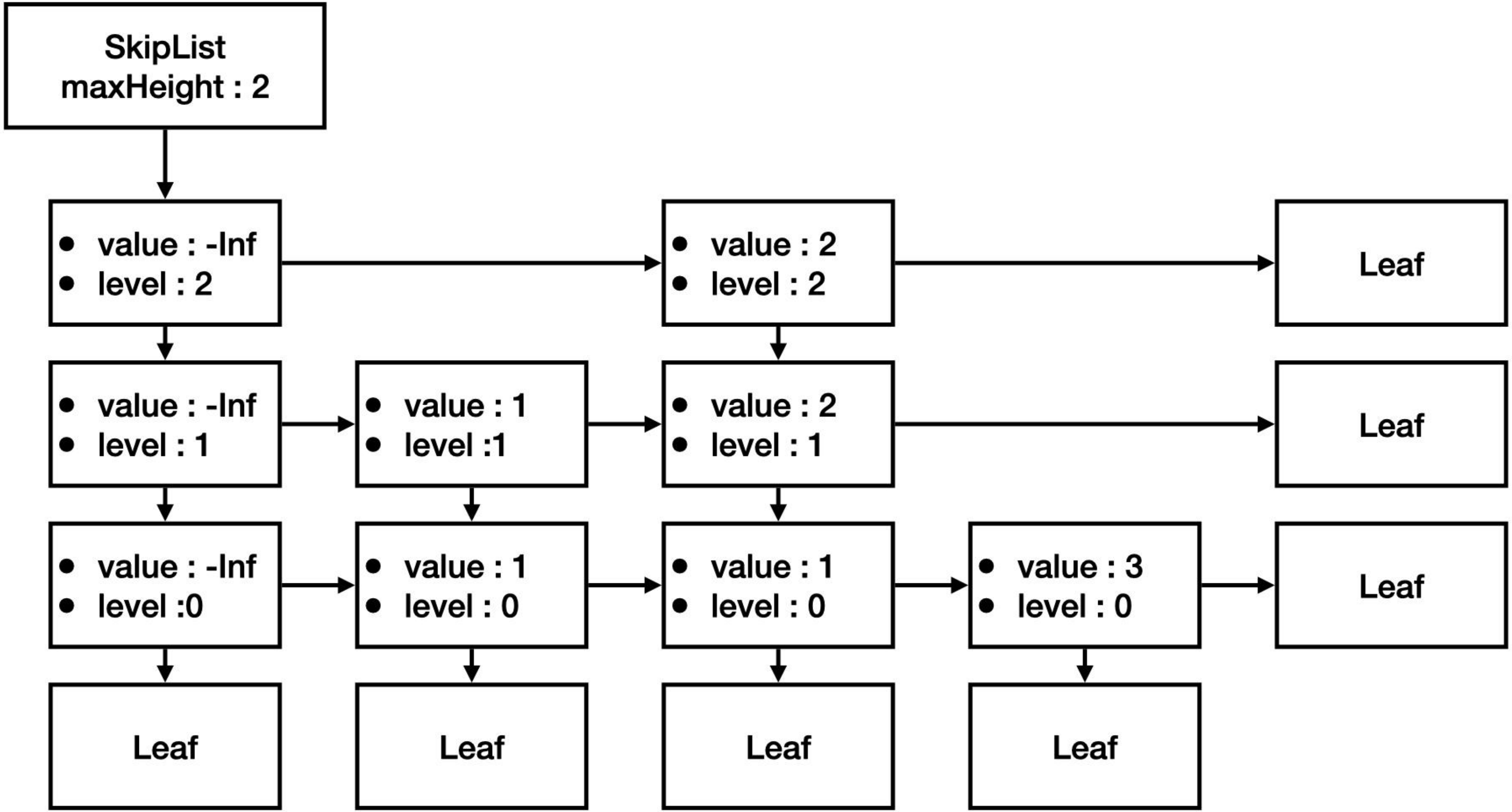
- A Skiplist is composed of :
  - *maxHeight*, the current maximum level of the SkipList
  - A *Node*, the first element at level *maxHeight* of the Skiplist
- A Node can be either :
  - A *SkipNode* itself composed of these elements
    - A value *v*
    - A level i
    - A node right
    - A node down
  - A *Leaf*, used as boundary for the bottom and the right of the Skiplist

```scala
case class SkipList(head: Node, maxHeight: BigInt)

abstract class Node
case class SkipNode(value: Int, down: Node, right: Node, height: BigInt) extends Node
case object Leaf extends Node
```

Properties of a valid SkipList :

- A SkipList is valid if:
  - maxHeight >= 0
  - it holds a valid SkipNode with value -Infinity and with level i = maxHeight
- A SkipNode n is valid if it is a Leaf, or if :
  - n.i >= 0
  - n.right and n.down are valid
  - If n.right is not a leaf then:
    - n.right.value > n.value
    - n.right.i = n.i
  - If n.i = 0:
    - n.down is a Leaf
  - If n.i > 0:
    - n.down.v = n.v
    - n.down.i = n.i-1
  - If n.i > 0:
    - n.right.down is found somewhere to the right of n.down

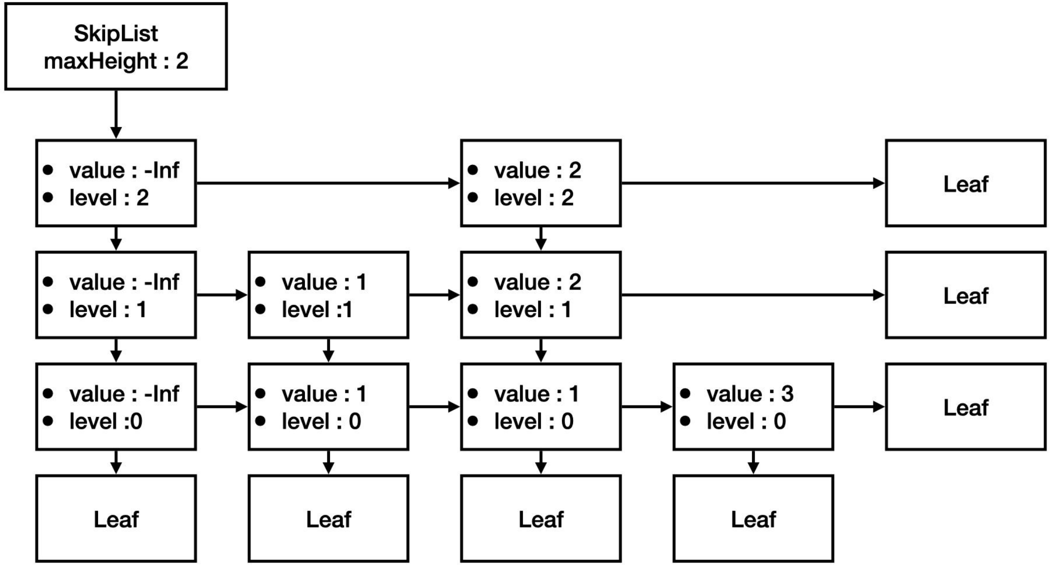


SkipList
maxHeight : 2
value : -Inf
level : 2
value : 2
level : 2
Leaf
value : -Inf
level : 1
value : 1
level :1
value : 2
level : 1
Leaf
value : -Inf
level :0
value : 1
level : 0
value : 1
level : 0
value : 3
level : 0
Leaf
Leaf
Leaf
Leaf
Leaf

The properties we want to prove to have valid operations:

Assuming all subsequent nodes are valid :

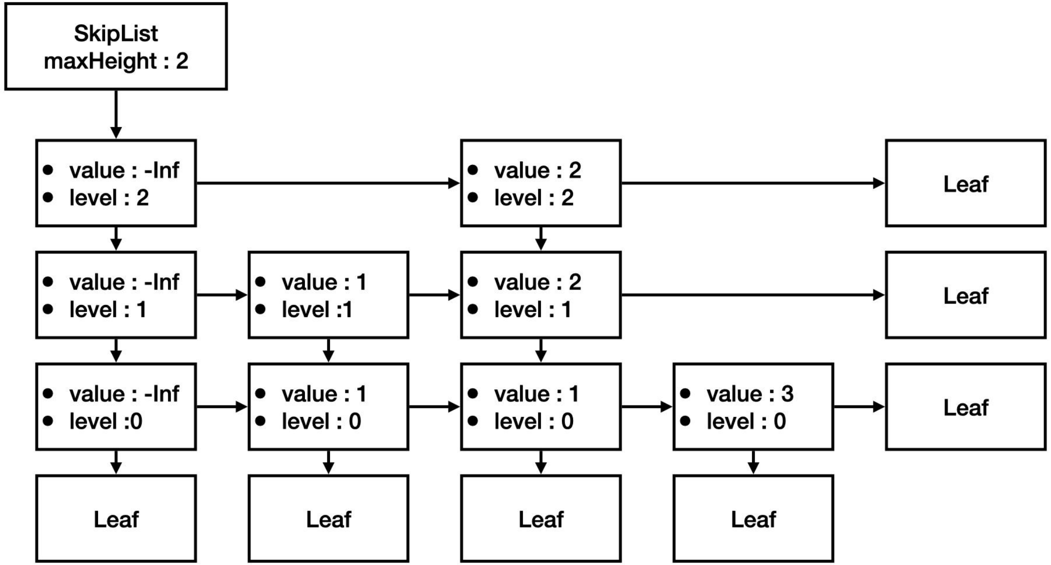
- If a node v2 is somewhere to the right of a node v1 then v2.down is somewhere to the right of v1.down



The properties we want to prove to have valid operations:

Stability properties :

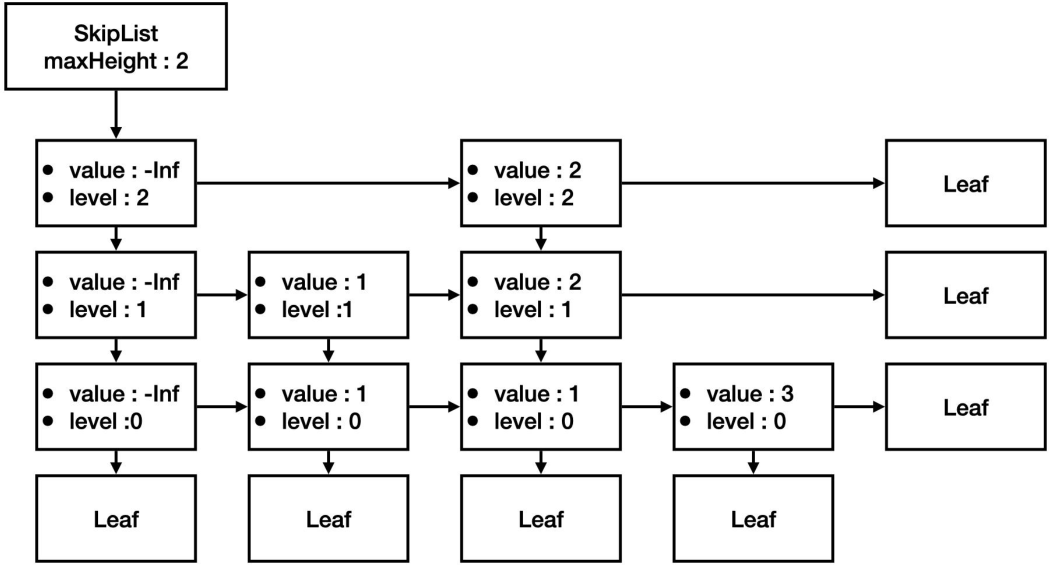
- Insert(sl,v) is also a Skiplist
- Remove(sl,v) is also a Skiplist
- If sl contains v1 and v1!=v2 insert(sl,v2) contains v1
- If sl contains v1 and v1!=v2, remove(sl, v2) contains v1



The properties we want to prove to have valid operations:

Correctness properties :

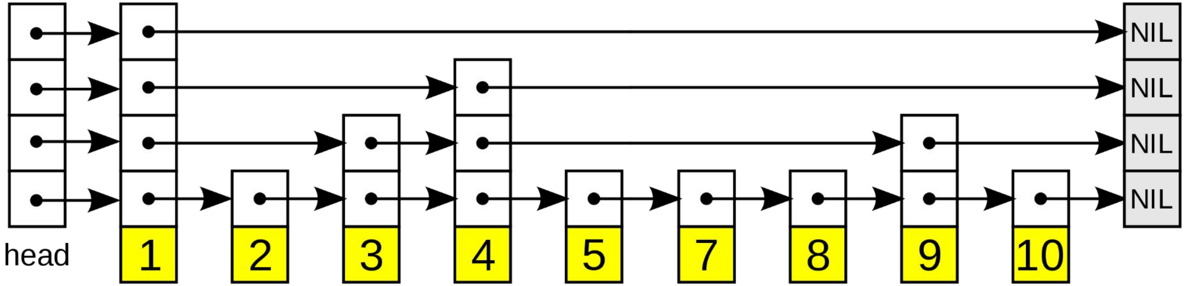
- Insert(sl,v) contains v
- Remove(sl,v) does not contains v
- If sl contains v, search(sl,v) returns v
- If sl does not contain v, search(sl,v) returns None

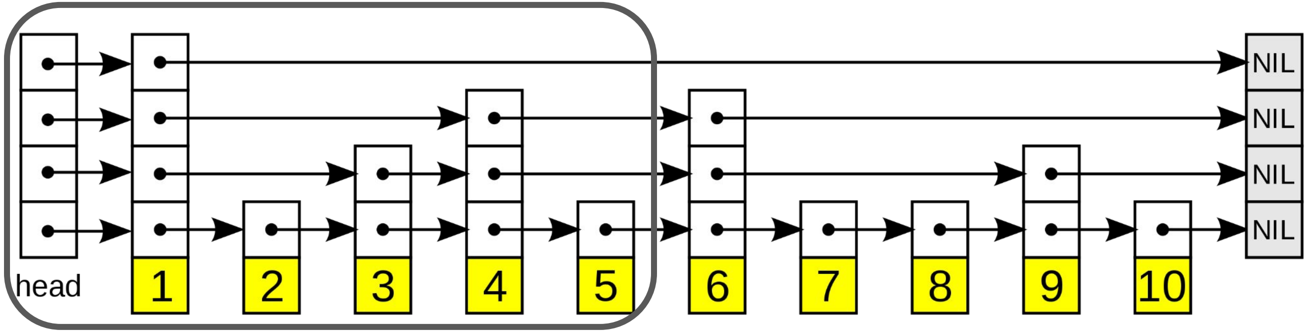


Issues

First problem : functional programming for dynamic links

A SkipNode does not contain **pointers to nodes but the nodes themselves**.
So inserting or removing an element k means we have to **recreate every node** with value < k.
So actually insert and remove are O(n), and only search is O(log n).

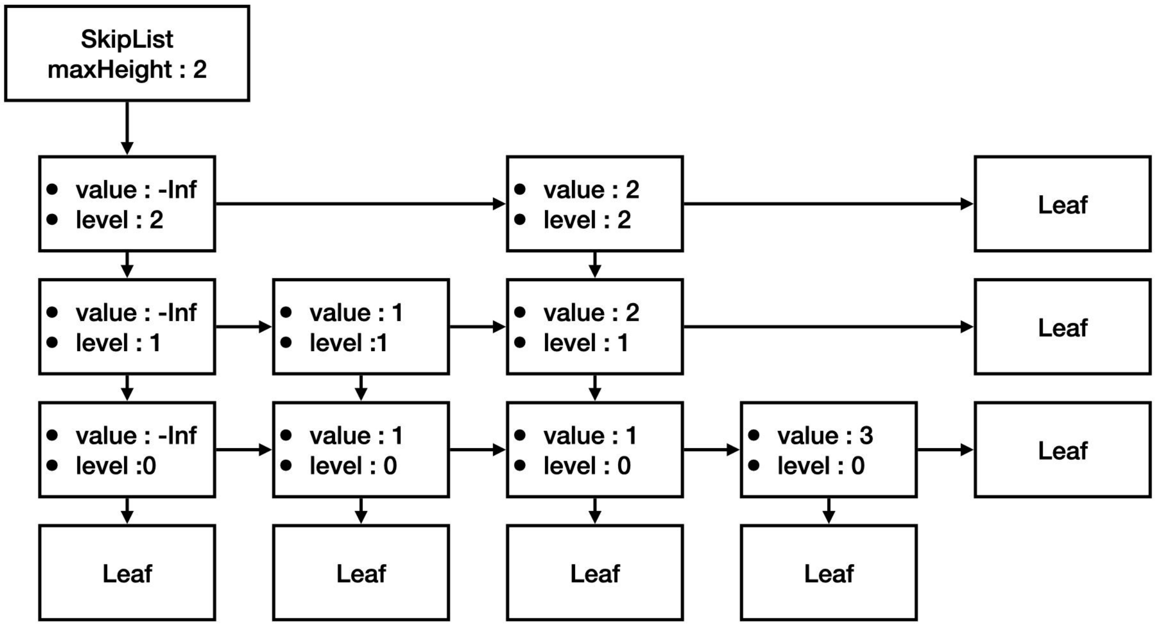




example: insert(6)

Second problem : every SkipNode has 2 incoming links

We are not working with a tree, so for a node n, to update its subtree, it is not possible to have a simple recursion on n.down and n.right as it would create duplicates of some nodes.



Third problem : Decreasing measures in mutual recursion

```scala
sealed abstract class Node
case class SkipNode(right: Node) extends Node
case object Leaf extends Node
def insert(n: Node): Node = {
    decreases(size(n))
    n match {
        case SkipNode(r) => {
            insertIsSkipList(r)
            insert(r)
        }
        case Leaf => Leaf
    }
}

def insertIsSkipList(n: Node): Unit = {
    decreases(size(n))
    assert(isLeaf(insert(n)))
} ensuring (_ => isLeaf(insert(n)))
```

⇒ Means we had to rewrite our insertion and removal method to be tail-recursive

Fourth problem : It becomes very long, very fast

Every non-trivial lemma usually requires multiple sub-lemmas

Even with the stainless cache, the running times are now ~ 1min

```
total: 1016 valid: 1011 (526 from cache) invalid: 0    unknown: 5    time:    56.8
```

Where are we at ?

Not quite done with termination proof of insert. Remove remaining.

Not started proving properties, but we now have lots of lemmas, so it might be faster.

Questions ?