# AULA 8

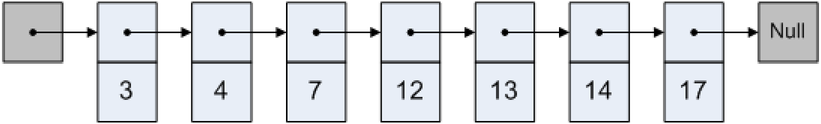
# Skip lists

## A Probabilistic Alternative to Balanced Trees
## William Pugh

Skip lists é uma estrutura de dados probabilística baseada em uma generalização de listas ligadas: utilizam balanceamento probabilístico em vez de forçar balanceamento.

Referências: CMSC 420; Skip Lists: Done Right; Open Data Structures; ConcurrentSkipListMap (Java Platform SE 8); Randomization: Skip Lists (YouTube)

# Lista (simplemente) ligada

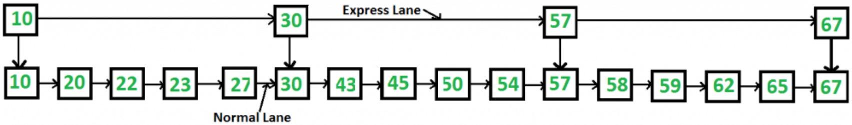



Fonte: Skip lists are fascinating!

Cada nó `x` tem três campos:

1. `key`: chave do item;
2. `val`: valor associado a chave;
3. `next`: próximo nó na lista

# 2 níveis de listas ligadas




Fonte: GeeksforGeeks

Cada nó x tem quatro campos:

1. key: chave do item;
2. val: valor associado a chave;
3. next[0]: próximo nó na lista no níveis 0
4. next[1]: próximo nó na lista no níveis 1

# Consumo de tempo de get()

$L_0$ = lista ligada do nível 0 (=térreo)

$L_1$ = lista ligada do nível 1 (= 1o. andar)

$n$ = número de itens na ST = número de nós em $L_0$

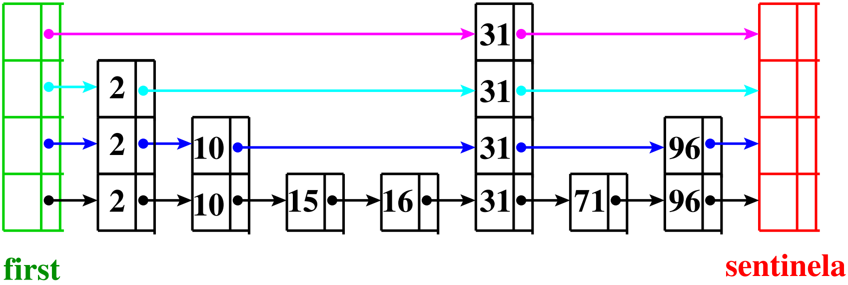
Consumo de tempo de get() é pelo menos

$$|L_1| + n/|L_1|$$

Valor minimizado quando $|L_1| = \sqrt{n}$.

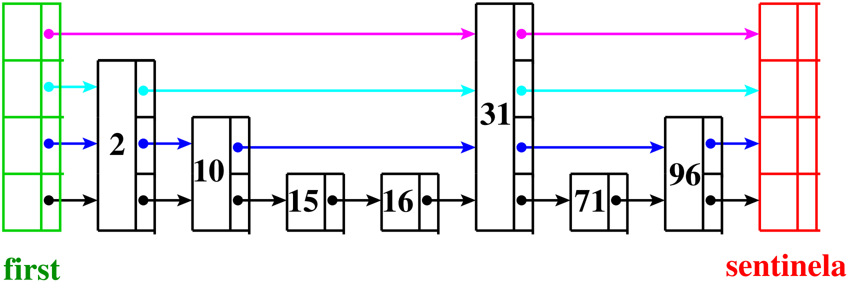
De fato, $\sqrt{n}$ é ponto de mínimo de $x + n/x$

# Multiplas listas




- keys ordenadas
- `first` e `setinela` em lista

# Skip list

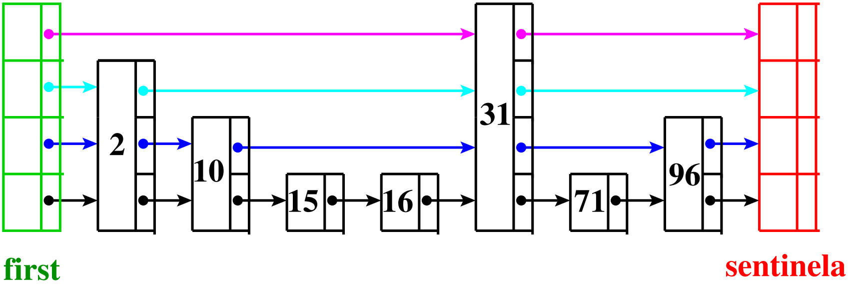



- `keys` ordenadas
- `first` e `setinela` em cada nível
- `next[]` de tamanho variado

# subclasse Node

```
private class Node {
   private String key;
   private Integer val;
   private Node[] next;
   public Node(String key, Integer val,
                    int levels) {
       this.key = key;
       this.val = val;
       this.next = new Node[levels];
   }
}
```

# Skip list



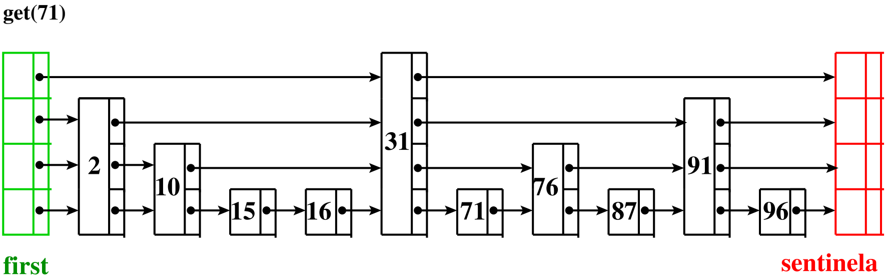

Chamada **skip list** pois listas de mais alto níveis permite *skip* vários itens.

# SkipListST

```
public class SkipListST{
   // temos no máximo 31 listas
   private int MAXLEVELS = 31;
   // número de níveis 0,1,...,lgN-1
   private int lgN;
   private Node first; nó cabeça
   // número de itens na ST
   private int n = 0;

   public SkipListST() {
       first = newNode(MAXLEVELS);
   }
```

# get(k)

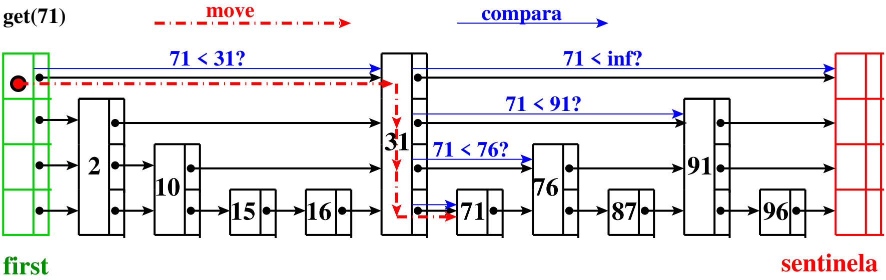



if k == key, achou

if k < next.key, vá para nível inferior

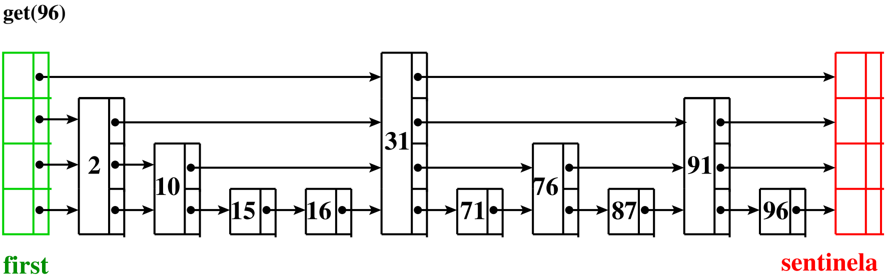
if k >= next.key, vá para direita

# get(k)



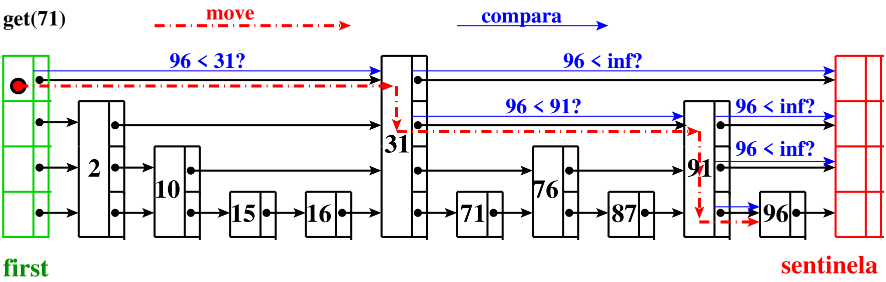

```
if k == key, achou
if k < next.key, vá para nível inferior
if k >= next.key, vá para direita
```

# get(k)




```
if k == key, achou
if k < next.key, vá para nível inferior
if k >= next.key, vá para direita
```

# get(k)




```
if k == key, achou
if k < next.key, vá para nível inferior
if k >= next.key, vá para direita
```

# get() para lista ligada

```
public Value get(Key key) {
   Node p = prev(key);
   // key está na ST?
   Node q = p.next;
   if (q != null && q.key.equals(key))
       return q.val;
   return null;
}
```

# get() para skip list

```
public Value get(Key key) {
   Node p = first;
   for (int k = lgN-1; k >= 0; k--) {
      Node p = prev(key, p, k);
      // key está na ST?
      Node q = p.next[k];
      if (q != null && q.key.equals(key))
         return q.val;
   }
   return null;
}
```

# Operação básica para lista ligada

Aqui usamos a ordenação (`compareTo()`)

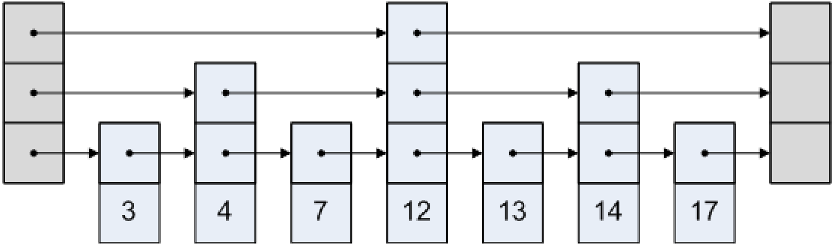
```
private Node prev(Key key) {
   Node p = first;
   Node q = first.next;
   while (q != null
          && q.key.compareTo(key) < 0) {
      p = q;
      q = q.next;
   }
   return p;
}
```

# Operação básica para skip list

Aqui usamos a ordenação (compareTo())

```
private Node prev(Key key, Node start,
                          int k) {
   Node p = start;
   Node q = start.next[k];
   while (q != null
         && q.key.compareTo(key) < 0) {
      p = q;
      q = q.next[k];
   }
   return p;
}
```

# Skip list “perfeita”

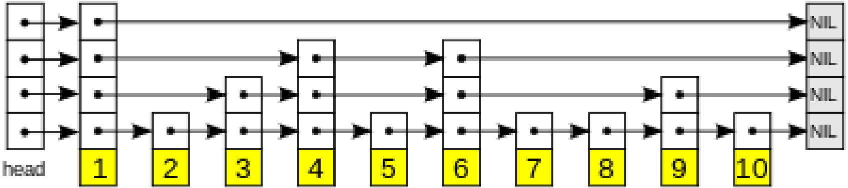



Fonte: Skip lists are fascinating!

Exemplo: **perfeita**

Cada link em um nível “pula” dois links do nível inferior.

# Skip list “perfeita”




Fonte: https://www.geeksforgeeks.org/skip-list/

Exemplo: **não-perfeita**

Cada link em um nível “pula” dois links do nível inferior.

# Consumo de tempo de `get()`

Supondo a skip list "perfeita": usando links de um nível superior pulamos um nó do seu nível inferior.

**Fato.** O número de níveis é proporcional $\leq \lg \mathtt{n}$.

**Fato.** Em uma busca visitamos no máximo 2 nós por nível, caso contrário usaríamos o nível superior.

**Conclusão.** Número de comparações é $\leq 2 \lg \mathtt{n}$.

# Inserções e remoções

Inserções e remoções podem destruir *perfeição*

Exigência de perfeição pode custar **muito caro**.

**Ideia.**

- relaxar a exigência de que cada nível tenha metade dos links do anteriors
- estrutura que **esperamos** que cada nível tenha metade dos links do nível anterior bem distribuídos

Skip list é uma estrutura de dados **aleatorizada** (*randomized*): a mesma sequência de inserções e remoções podem produzir estruturas diferentes dependendo de um gerador de números aleatórios.

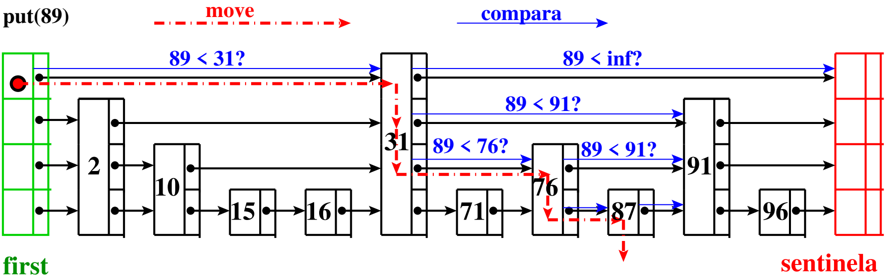
# Aleatorização

- permite imperfeição
- comportamento **esperado** é o mesmo que de skip lists perfeitas
- Ideia: cada nó é promovido para o nível superio com probabilidade $1/2$
    - número de nós esperados no nível 1 é $\mathtt{n}/2$ dos nós
    - número de nós esperados no nível 1 é $\mathtt{n}/2^2$ dos nós
    - $\ldots$

Número de nós esperados em cada nível é o mesmo de uma skip list perfeita

É esperado que os nós promovidos sejam bem distribuídos.

# put(key,val)




procure key

insira item key,val no nível 0

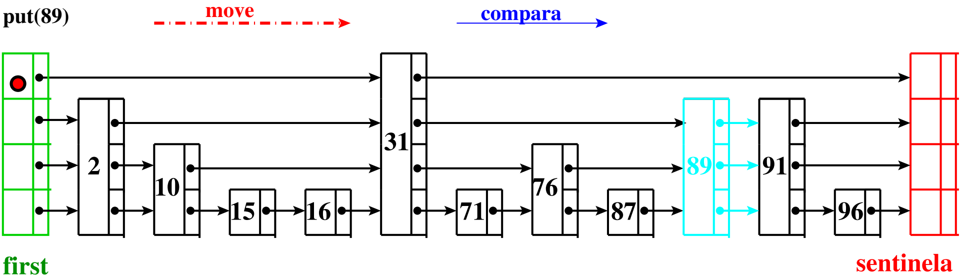
i $\leftarrow 1$

enquanto FLIP() $=$  faça

  insira item key,val no nível i

  i $\leftarrow$ i $+ 1$

# put(key,val)




procure key

insira item key,val no nível 0

i ← 1

enquanto FLIP() =  faça

  insira item key,val no nível i

  i ← i + 1

# put() para lista ligada

```
public void put(Key key, Value val) {
   if (val == null) {
      delete(key); return;
   }
   Node p = prev(key);
   Node q = p.next;
   // key está na ST?
   if (q != null || q.key.equals(key)) {
      q.val = val; return;
   }
   // key não está na ST
   p.next = new Node(key, val, q);
   n++;
}
```

# put() para skip list

```
public void put(Key key, Value val) {
   if (val == null) {
      delete(key); return;
   }
   Node[] s = new Node[MAXLEVELS];
   Node p = first;
   for (int k = lgN-1; k >= 0; k--) {
      Node p = prev(key, p, k);
      Node q = p.next[k];
      if (q != null || q.key.equals(key)){
         q.val = val; return;
      }
      s[k] = p;
   }
```

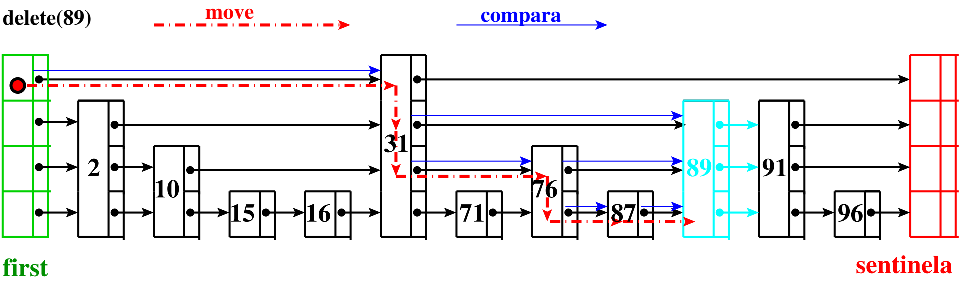
# put() para skip list

```
// key não está na ST
int levels = randLevel();
Node novo = new Node(key,val,levels);
if (levels == lgN+1) {
    s[lgN] = first;
    lgN++; // atualiza o no.  níveis
}
for (int k = levels-1; k >= 0; k--) {
    Node t = s[k].next[k];
    s[k].next[k] = novo;
    novo.next[k] = t;
}
n++;
}
```

# randLevel()

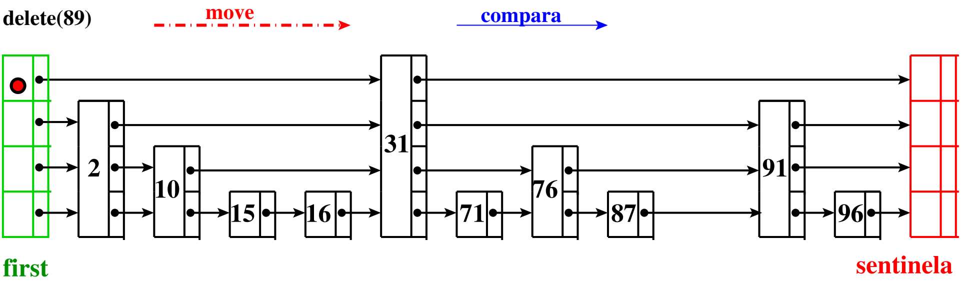
```
private int randLevel() {
   int level= 0;
   int r=StdRandom.uniform((1<<(MAXL-1)));
   while ((r & 1) == 1) {
      if (level== lgN) {
         if(lgN == MAXL) return MAXL;
         else return lgN + 1;
      }
      level++;
      r >>= 1;
   }
   return level+1;
}
```

# delete(k)



delete(89)
move
compara
2
10
15
16
31
71
76
87
89
91
96
first
sentinela

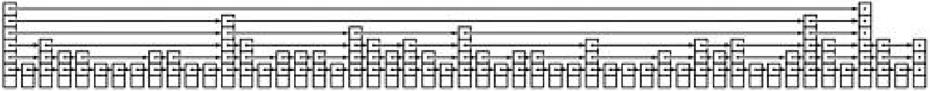
# `delete(k)`



delete(89)
move
compara
31
2
10
15
16
71
76
87
91
96
first
sentinela

# Skip list

Estrutura aleatorizada (*randomized*)



Fonte: 13.5 Skip Lists

**Fato.** O número esperado de níveis é $\mathrm{O}(\lg \mathtt{n})$.

**Fato.** Em uma busca o número esperado de nós visitados por nível é 2.

**Conclusão.** O consumo de tempo esperado de `get()`, `put()`, `delete()` é $\mathrm{O}(\lg \mathtt{n})$.

# Rascunho de uma prova ...

Probabilidade de um item ser "promovido" até o nível $\mathtt{i}$ é a probabilidade de obtermos $\mathtt{i}-1$  nas primeiras jogadas da moeda ... é $1/2^{\mathtt{i}-1}$.

Seja $\mathtt{H}$ o número máximo de níveis de um skip list com $\mathtt{n}$ itens.

Temos que $\Pr[\mathtt{H} \geq \mathtt{i}] \leq \mathtt{n}/2^{\mathtt{i}-1}$. De fato,

$$\begin{aligned}\Pr[\mathtt{H} \geq i] &= \Pr[\text{nível } \mathtt{i} \text{ conter algum item}] \\ &\leq \sum_x \Pr[\text{item } \mathtt{x} \text{ está no nível } \mathtt{i}] \\ &= \mathtt{n}/2^{\mathtt{i}-1}\end{aligned}$$

# Conclusão

$$\Pr[\mathtt{H} \geq \mathtt{c} \lg \mathtt{n}] \leq \mathtt{n}/2^{\mathtt{c} \lg \mathtt{n} - 1} < \frac{\mathtt{n}}{2^{\mathtt{c} \lg \mathtt{n}}} = \frac{\mathtt{n}}{\mathtt{n}^{\mathtt{c}}} = \frac{1}{\mathtt{n}^{\mathtt{c}-1}}$$

Em palavras, $\mathtt{H}$ é $O(\lg \mathtt{n})$ com alta probabilidade.

Se $\mathtt{n} = 1000$ e $\mathtt{c} = 3$ então a probabilidade de $\mathtt{H}$ ser maior que $3 \lg 1000 < 30$ é menor que 1 em um milhão.