MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 9

Skip lists

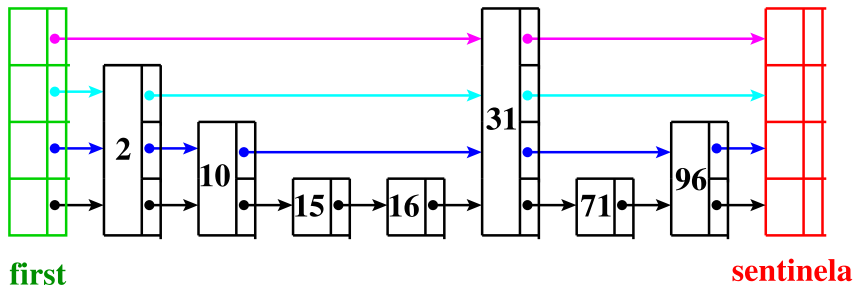
A Probabilistic Alternative to Balanced Trees

William Pugh

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

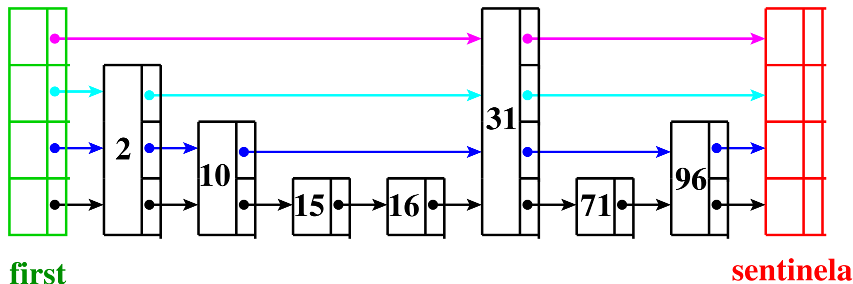
Referências: CMSC 420; Skip Lists: Done Right; Open Data Structures; ConcurrentSkipListMap (Java Platform SE 8); Randomization: Skip Lists (YouTube)

Skip list



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

Skip list



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

Chamada **skip list** pois listas de mais altos níveis permitem *skip* vários itens.

Os tipos struct node e Link

```
typedef struct node *Link;

struct node {
    Key key;
    Value val;
    Link *next;
}

Link newNode(Key key, Value val, int levels) {
    Link p = mallocSafe(sizeof(*p));  int k;
    p->key = key;
    p->val = val;
    p->next = mallocSafe(levels * sizeof(Link));
    for (k = 0; k < levels; k++)
        p->next[k] = NULL;        /* ou sentinela */
    return p;
}
```

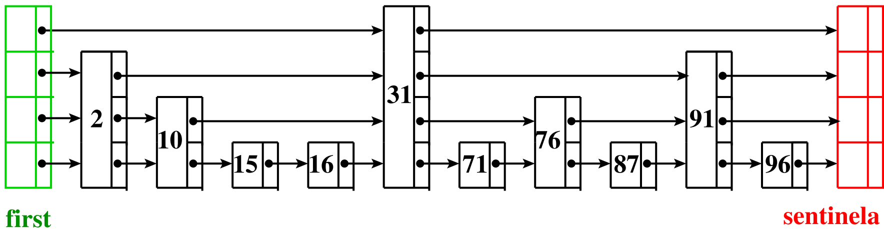
AULA 10

Arquivo SkipListST.c: esqueleto

```
#include "st.h"

static struct node first;                    /* nó cabeça */
static int n;              /* número de elementos na ST */
static int MAXLEVELS;        /* número máximo de níveis */
static int lgN;

void  stInit(cap) {...}
Value stGet(Key key) {...}
void  stPut(Key key, Value val) {...}
void  stDelete(Key key) {...}

Link  rank(Key key, Link p, int level) {...}
```

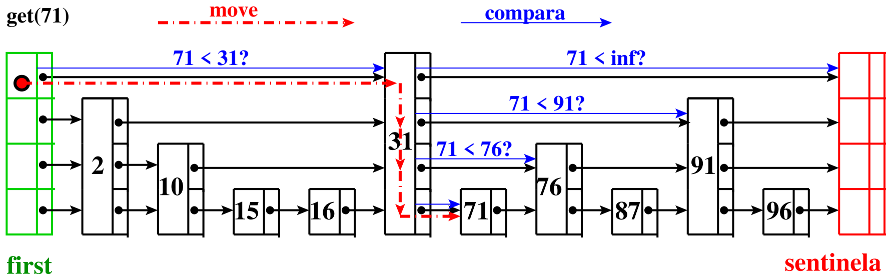
Note os parâmetros diferentes no `rank`.

get(k)



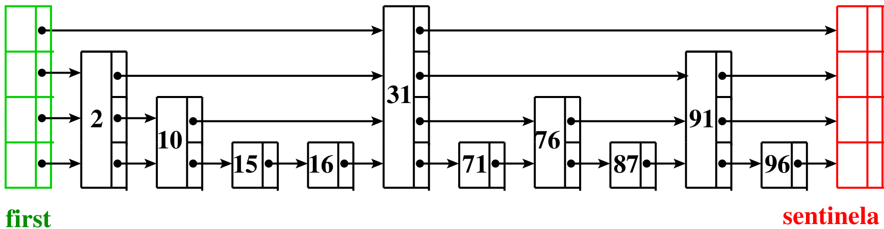

```
if k == key, achou
if k <  next->key, vá para nível inferior
if k >= next->key, vá para direita
```

get(k)



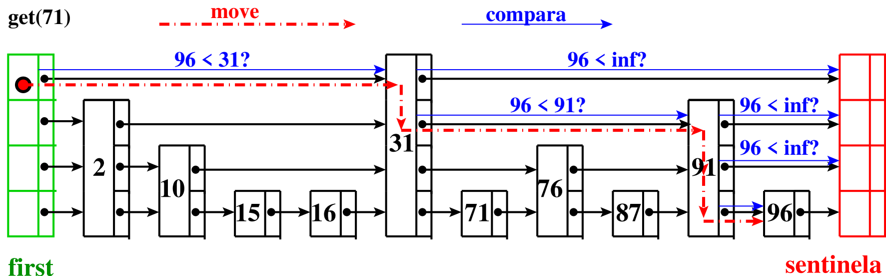

```
if k == key, achou
if k <  next->key, vá para nível inferior
if k >= next->key, vá para direita
```

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 ordenada

```
Value stGet(Key key) {
   Link p, q;
   p = rank(key);
   q = p->next;
   if (q != NULL                  /* key está na ST? */
         && compare(key, q->key) == 0))
      return q->val;
   return NULL;
}
```

get() para skip list

```
Value stGet(Key key) {
   Link p = &first, q;
   int k;
   for (k = lgN-1; k >= 0; k--) {
      p = rank(key, p, k);
      q = p->next[k];
      if (q != NULL          /* key está na ST? */
            && compare(key, q->key) == 0))
         return q->val;
      }
   return NULL;
}
```

Operação básica para lista ligada ordenada

Aqui usamos a ordem nas chaves.

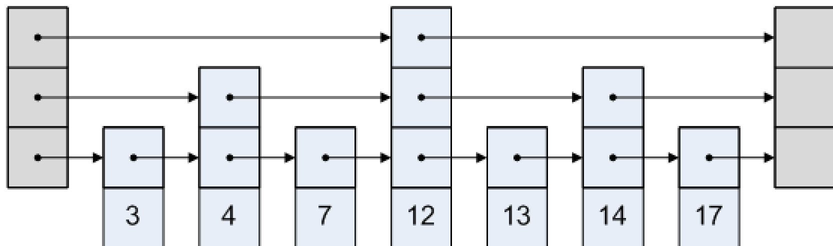
```
static Link rank(Key key) {
   Link p = &first, q = p->next;
   while (q != NULL
         && compare(key, q->key) < 0) {
      p = q;
      q = q->next;
   }
   return p;
}
```

Operação básica para skip list

Aqui usamos a ordem nas chaves.

```
static
Link rank(Key key, Link start, int k) {
   Link p = start, q = start->next[k];
   while (q != NULL
         && compare(key, q->key) < 0) {
      p = q;
      q = q->next[k];
   }
   return p;
}
```

Skip list “perfeita”

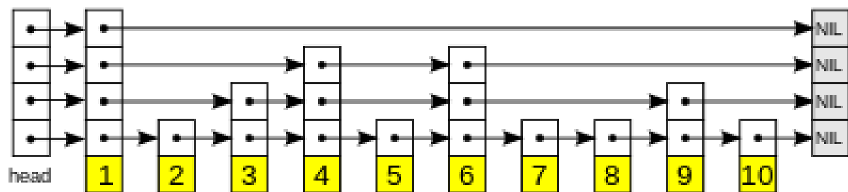


Fonte: Skip lists are fascinating!

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 anteriores
- 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 pode produzir estruturas diferentes dependendo do gerador de números aleatórios.

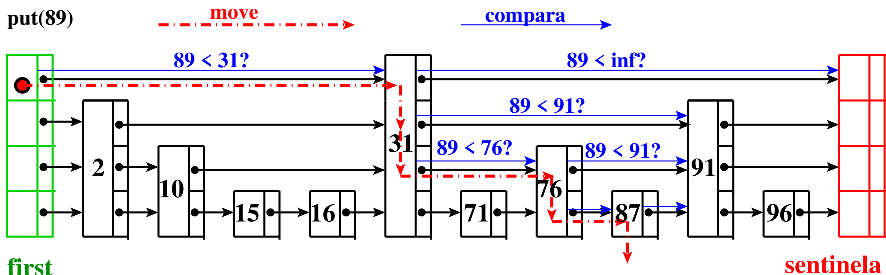
Aleatorização

- permite imperfeição
- comportamento **esperado** é
  o mesmo que de skip lists perfeitas
- Ideia: cada nó é promovido para
  o nível superior 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.

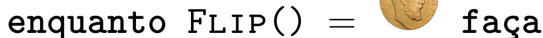
put(key,val)



procure key

insira item key,val no nível 0

i ← 1

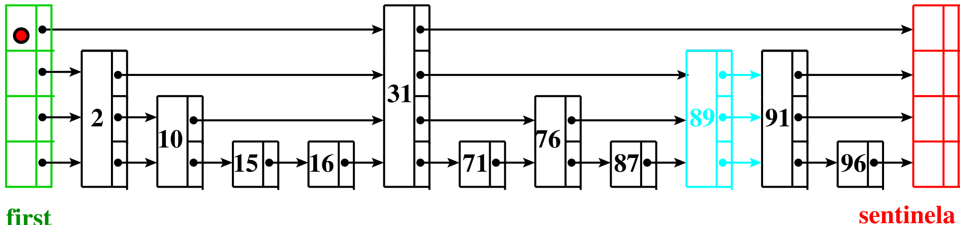


enquanto FLIP() = [coin] faça

insira item key,val no nível i

i ← i + 1

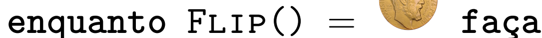
put(key,val)



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() para lista ligada ordenada

```
void stPut(Key key, Value val) {
    Link p = rank(key), q = p->next;
    if (q != NULL && compare(key, q->key) == 0)
        q->val = val;              /* key está na ST */
    else {                     /* key não está na ST */
        q = newNode(key, val, p->next);
        p->next = q;
        n++;
    }
}
```

put() para skip list

```
void stPut(Key key, Value val) {
   Link *s, p, q; int k, levels;
   s = mallocSafe(MAXLEVELS * sizeof(Link));
   p = &first;
   for (k = lgN-1; k >= 0; k--) {
      p = rank(key, p, k);
      q = p->next[k];
      if (q != NULL
            && compare(key, q->key) == 0) {
         q->val = val;
         return;
      }
      s[k] = p;
   }
```

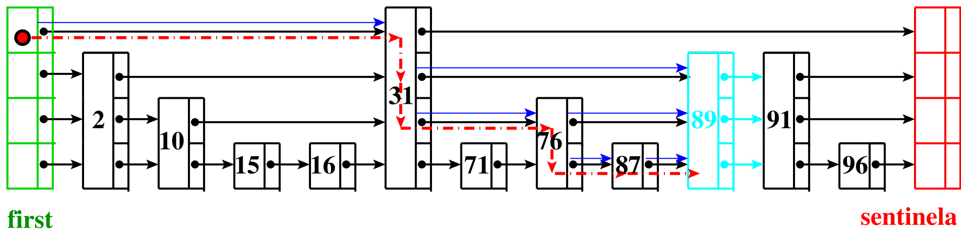
put() para skip list

```
    /* key não está na ST */
    levels = randLevel();
    p = newNode(key, val, levels);
    /* atualizar o no. de níveis?  */
    if (levels == lgN + 1) {
        s[lgN] = &first;
        lgN++;            /* atualiza o no. níveis */
    }
    for (k = levels-1; k >= 0; k--) {
        q = s[k]->next[k];
        s[k]->next[k] = p;
        p->next[k] = q;
    }
    n++;
}
```

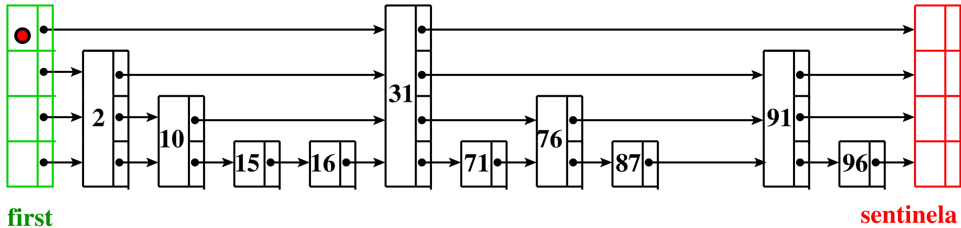
randLevel()

```
static int randLevel() {
    int level = 0;
    int r = rand() % (1<<(MAXLEVELS-1));
    while ((r & 1) == 1) {
        if (level == lgN) {
            if (lgN == MAXLEVELS)
                return MAXLEVELS;
            else
                return lgN + 1;
        }
        level++;
        r >>= 1;
    }
    return level + 1;
}
```

delete(k)

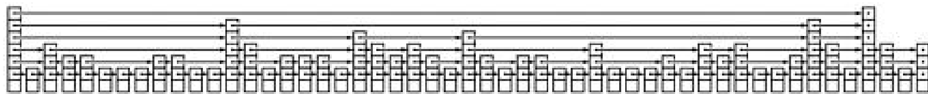


delete(k)



Skip list

Estrutura aleatorizada (*randomized*)



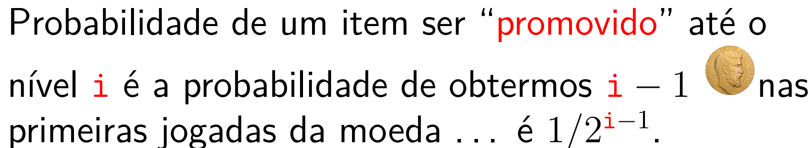
Fonte: 13.5 Skip Lists

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

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 $O(\lg \mathtt{n})$.

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 uma skip list com $\mathtt{n}$ itens.

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

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

Conclusão

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

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

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

Prós

Skip lists são:

- fáceis de serem implementadas;
- mantém `n` pares `key`-`value` e consomem tempo esperado $\mathrm{O}(\lg \mathtt{n})$ por operação com alta probabilidade; e
- são *concurrency-friendly* já que atualizações são feitas apenas localmente.

Prós

Veja também

- *Choose Concurrency-Friendly Data Structures*
- `class` ConcurrentSkipListMap<K,V>: This class implements a concurrent variant of SkipLists providing expected average $\lg n$ time cost for the `containsKey`, `get`, `put` and `remove` operations and their variants. Insertion, removal, update, and access operations safely execute concurrently by multiple threads.
- `class ConcurrentSkipListSet<E>`: This implementation provides expected average $\lg n$ time cost for the contains, `add`, and `remove` operations and their variants. ...

Experimentos

Consumo de tempo para se criar uma `ST` em que as chaves são as palavras em `les_miserables.txt` e os valores o número de ocorrências.

estrutura	ST	tempo
vetor	não-ordenada	59.5
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada	não-ordenada	147.1
lista ligada MTF	não-ordenada	15.3
lista ligada	ordenada	115.2
skiplist ♡	ordenada	1.1

Tempos em segundos obtidos com `StopWatch`.