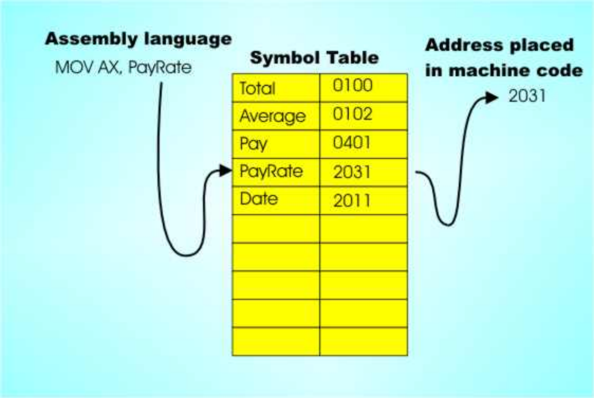
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 7

Tabelas de Símbolos



Fonte: http://www.i-programmer.info/

Tabelas de símbolos (PF)
Elementary Symbol Tables (S&W)

Tabelas de símbolos

Uma **tabela de símbolos** (`ST` = *symbol table*) é um ADT que consiste em um conjunto de itens, sendo cada item um par `chave`-`valor` ou `key`-`value`, munido de duas operações fundamentais:

- `put()`, que insere um novo item na `ST`, e
- `get()`, que busca o valor associado a uma dada chave.

Tabelas de símbolos

Convenções sobre `ST`s:

- não há chaves repetidas (as chaves são duas a duas distintas),
- `null` nunca é usado como `key`,
- `null` nunca é usado como `value` associado a uma `key`.

`ST`s são também chamadas de *dictionarys*, *maps* e *associative arrays*.

API ST

`public class ST<Key,Value>`

	public class ST	
	`ST()`	cria uma ST
`void`	`put(Key key, Value val)`	insere (`key`,`val`)
`Value`	`get(Key key)`	busca o valor associado a `key`
`void`	`delete(Key key)`	remove (`key`,`val`
`int`	`rank(Key key)`	no. de keys menor que `key`
`boolean`	`isEmpty()`	ST está vazia?
`boolean`	`contains(Key key)`	a `key` está na ST?
`Iterable<Key>`	`keys()`	lista todas as chaves na ST

ST em vetor ordenado

Implementação usa dois vetores paralelos: um para as chaves, outro para os valores associados.

key	value	keys[] 0	1	2	3	4	5	6	7	8	9	N	vals[] 0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

entries in red were inserted

entries in gray did not move

entries in black moved to the right

circled entries are changed values

Trace of ordered-array ST implementation for standard indexing client

`BinarySearchST`: Conclusões

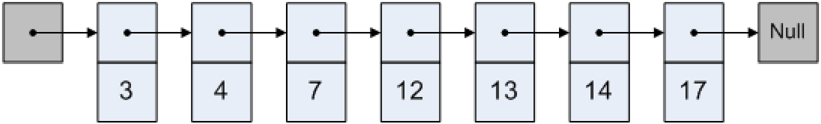
O consumo de tempo da função `get()` no pior caso é proporcional a $\lg \mathtt{n}$.

O consumo de tempo da função `put()` no pior caso é proporcional a $\mathtt{n}$.

O consumo de tempo para criar uma `ST` é no pior caso $O(\mathtt{n}^2)$.

ST em lista ligada ordenada

Implementação usa uma lista ligada ordenada.



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

`LinkedListST`: Conclusões

O consumo de tempo da função `get()` no pior caso é proporcional a `n`.

O consumo de tempo da função `put()` no pior caso é proporcional a `n`.

O consumo de tempo para criar uma `ST` é no pior caso $O(n^2)$.

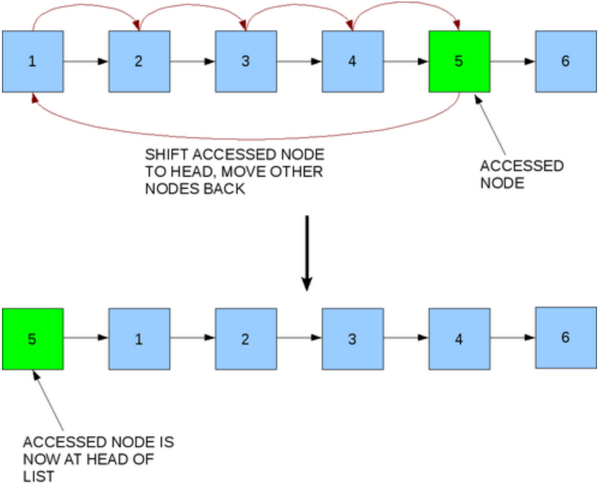
Self-organizing lists

Um busca é **auto-organizada** (*self-organizing*) se rearranja os `itens` da tabela de modo que aqueles mais frequentemente usados sejam mais fáceis de encontrar.

Como as probabilidades de acesso dos elementos geralmente não são conhecidas antecipadamente, foram desenvolvidas várias heurísticas para aproximar o comportamento ideal.

Método *mova para frente*

Assim que uma chave é consultada ela é movida para o início da lista (*Move to Front Method*).



Análise competitiva

J.L. Bentley, C.C. McGeoch, D.D. Sleator e R.E. Tarjan demonstraram que *move to front* nunca faz mais que quatro vezes o número de acessos a memória feito por qualquer outro algoritmo em listas lineares, dada qualquer sequência de consultas — mesmo que o outro algoritmo tenha conhecimento do futuro.

Com essa demonstração parece que nasceu a chamada **Análise Competitiva** de algoritmos online: comparamos o desempenho de um algoritmo com o desempenho de um algoritmo que sabe o futuro.

Análise competitiva

Um algoritmo ALG *online*/dinâmico é α-**competitivo** se existe uma constante k tal que para qualquer sequência de operações vale que

> *consumo de tempo de ALG* $\leq \alpha$ *consumo de tempo OPT* $+k$

Aqui, OPT é um algoritmo *offline*/estático para o mesmo problema. O ponto é que OPT pode pré-processar a sequência!.

(Sleator e Tarjan) MTF para listas é 4-competitivo para vetores e 2-competitivo para listas ligadas.

Experimentos

Consumo de tempo para se criar um ST em que a 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

Tempos em segundos obtidos com `StopWatch`.

Fique atento!

Falaremos sobre *Move to Front* pelo menos mais duas ou três vezes em MAC0323!

De maneira semelhante, redimensionamento de vetores nos acompanhará até o final do semestre.

Veja também Cache replacement policies.

AULA 8

Skip lists

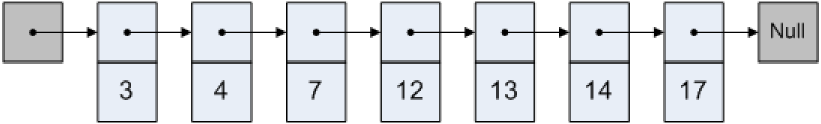
A Probabilistic Alternative to Balanced Trees

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



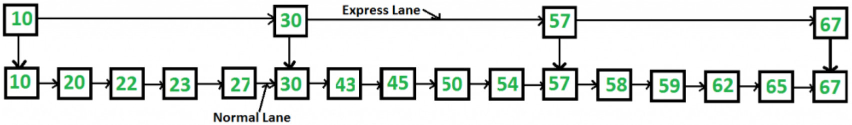
Fonte

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

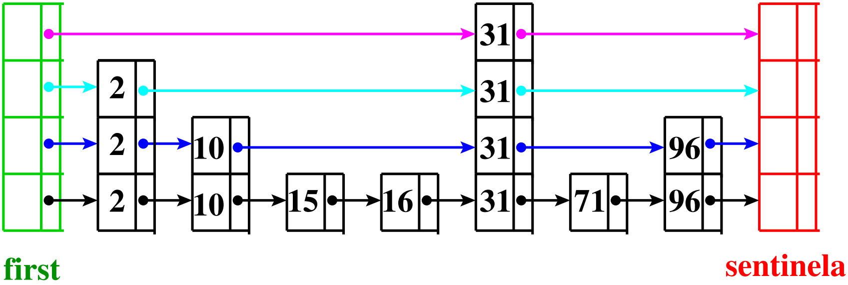
Consumo de tempo de get() é no máximo

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

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

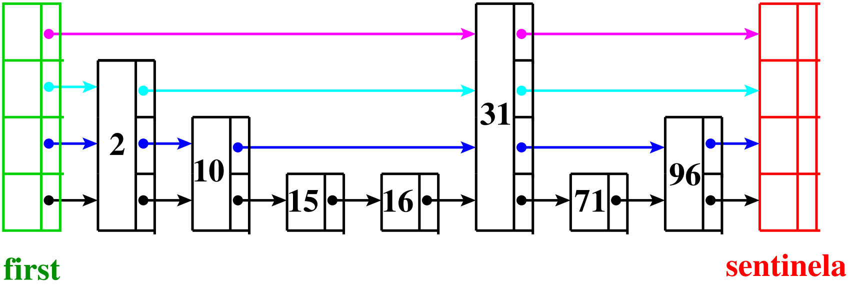
De fato, $\sqrt{n}$ é ponto de mínimo de $x + n/x$

Multiplas listas



- keys ordenadas
- first e setinela em lista

Skip list

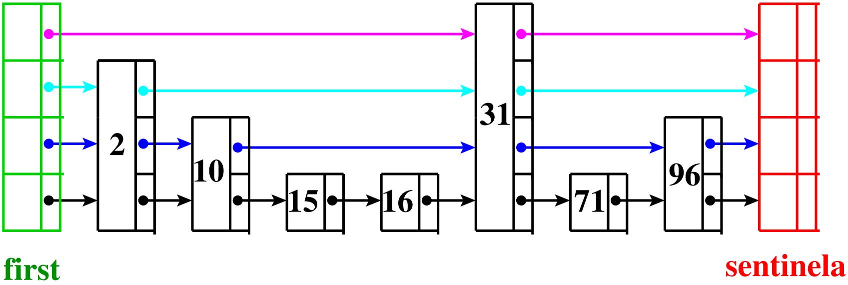


- keys ordenadas
- first e setinela em cada nível
- next[] de tamanho variado

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

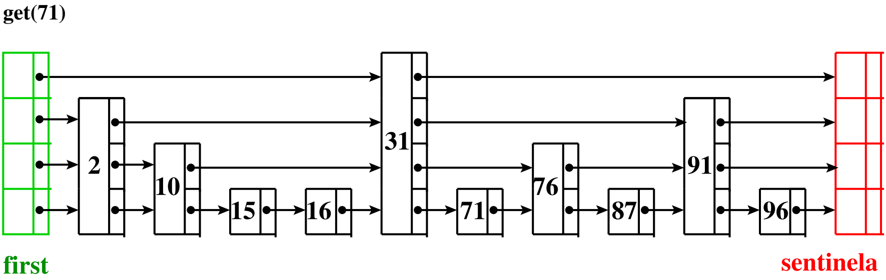


Chamada **skip list** pois listas de mais alto níveis permite *skip* vários itens.

SkipListST

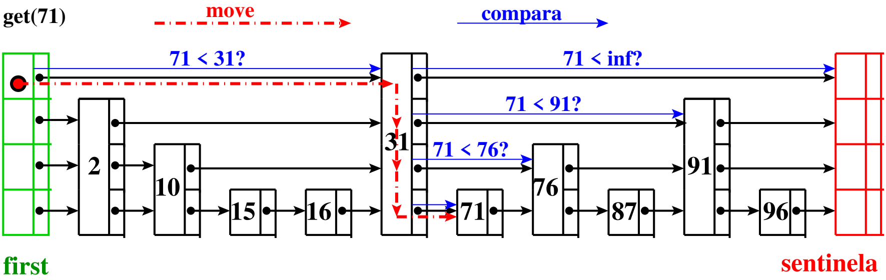
```
public class SkipListST{
  // temos no máximo 31 listas
  private int MAXLEVELS = 31;
  // número de níveis 0,1,...,lgN-1
  private int lgN;
  private Node first; nó cabeça
  // número de itens na ST
  private int n = 0;

  public SkipListST() {
     first = new Node(null, null,
                      MAXLEVELS);
  }
```

get(k)




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

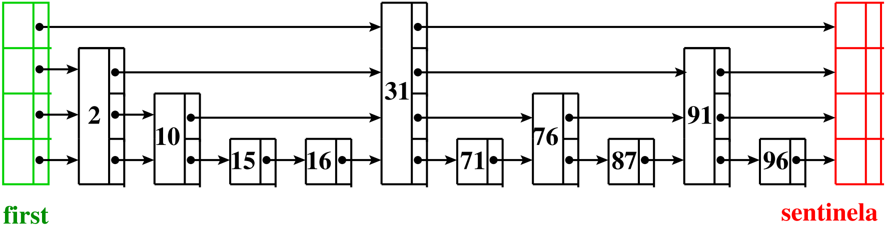
get(k)




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

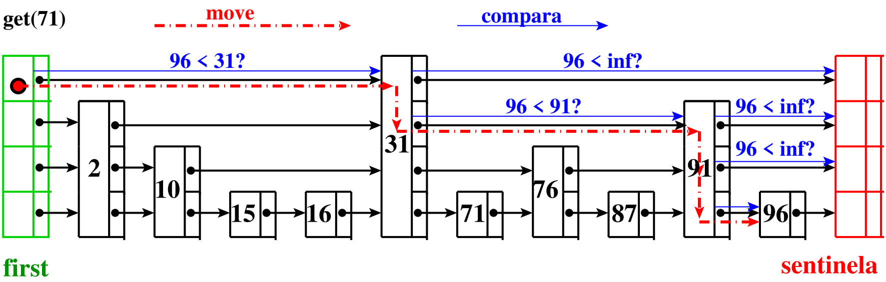
get(k)

get(96)




```
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 = rank(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 = rank(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()`)

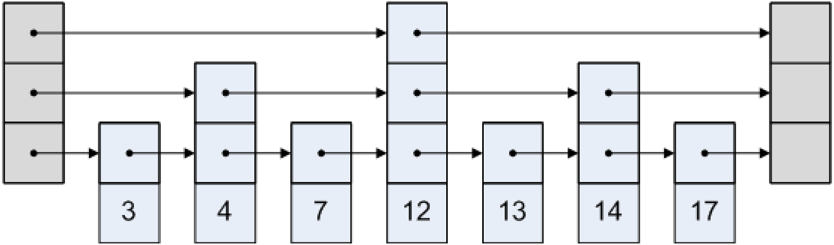
```
private Node rank(Key key) {
   Node p = first;
   Node q = first.next;
   while (q != null
          && q.key.compareTo(key) < 0) {
      p = q;
      q = q.next;
   }
   return p;
}
```

Operação básica para skip list

Aqui usamos a ordenação (`compareTo()`)

```
private Node rank(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"



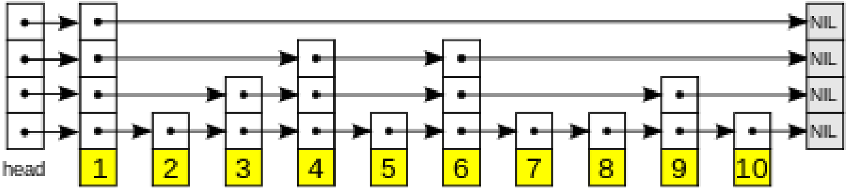
Font

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.

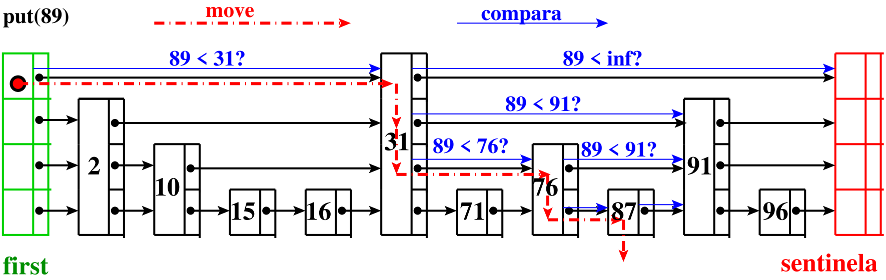
Aleatorização

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

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

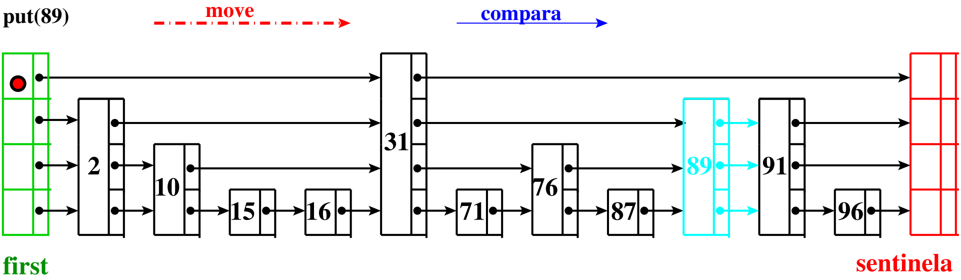
$i \leftarrow 1$

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 $\leftarrow$ 1

enquanto FLIP() =  faça

  insira item key,val no nível i

  i $\leftarrow$ i + 1

put() para lista ligada

```
public void put(Key key, Value val) {
   if (val == null) {
      delete(key); return;
   }
   Node p = rank(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 = rank(key, p, k);
      Node q = p.next[k];
      if (q != null || q.key.equals(key)){
         q.val = val; return;
      }
      s[k] = p;
   }
```

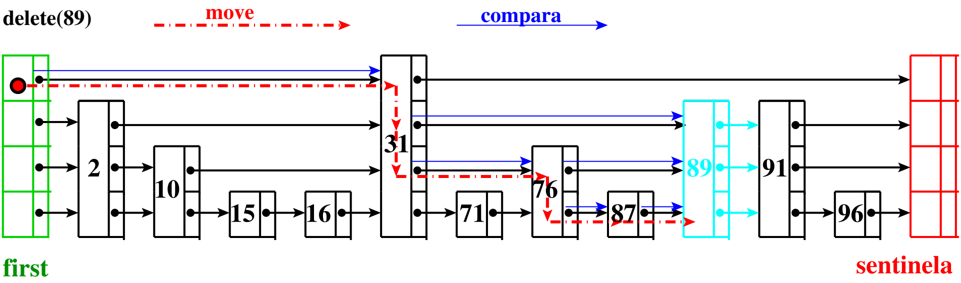
put() para skip list

```
// 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()

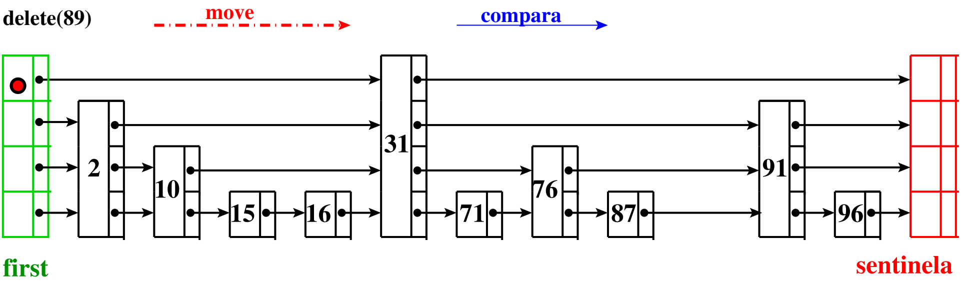
```
private int randLevel() {
   int level= 0;
   int r=StdRandom.uniform((1<<(MAXL-1)));
   while ((r & 1) == 1) {
      if (level == lgN) {
         if(lgN == MAXL) return MAXL;
         else return lgN + 1;
      }
      level++;
      r >>= 1;
   }
   return level+1;
}
```

delete(k)



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

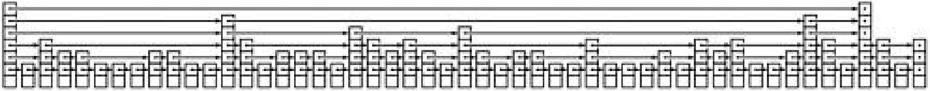
delete(k)



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 $O(\lg n)$.

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 $O(\lg n)$.

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,

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

Conclusão

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

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.

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 um ST em que a 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 ♡	não-ordenada	1.1

Tempos em segundos obtidos com StopWatch.