MC202 – Estruturas de Dados

Skip Lists

Emilio Francesquini – UNICAMP

13 de junho de 2017

Disclaimer

- Esses slides foram preparados pelo Prof. Orlando Lee para o curso de Estrutura de Dados ministrado na UNICAMP.
- Este material pode ser usado livremente desde que sejam mantido os créditos dos autores e da instituição.
- Muitos dos exemplos apresentados aqui foram retirados de uma apostila ou slides preparados pelo Prof. Tomasz Kowaltowski da UNICAMP. Outros exemplos foram retirados do livro "Algoritmos em Linguagem C" de Paulo Feofiloff, Editora Campus.

Referências


R. Sedgewick. Algorithms in C. Addison Wesley, 1990. Seção 13.5


http://igoro.com/archive/skip-lists-are-fascinating/.

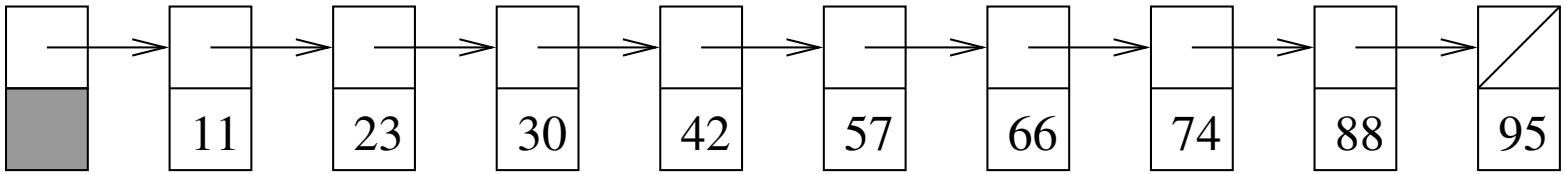

eternallyconfuzzled.com/tuts/datastructures/jsw_tut_skip.aspx
(artigo interessante e cheio de código!).


https://www.ime.usp.br/~cris/aulas/09_1_5710/slides/skiplist.pdf.

Skip lists

- Skip lists (listas com pulos?) são extensões de listas ligadas ordenadas.
- Tem desempenho semelhante a árvores de busca (balanceadas) mas são mais simples de implementar (não precisa fazer rotações).
- Usam apenas percurso em vetores ou listas ligadas.

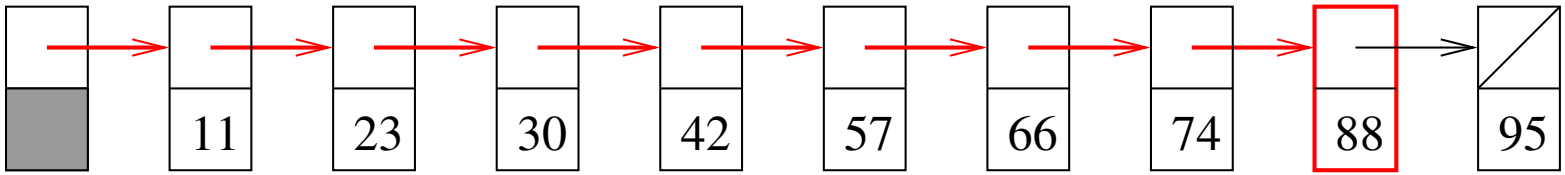
Skip list



Considere uma lista ligada ordenada com N elementos.

As operações de busca, inserção e remoção levam tempo $O(N)$.

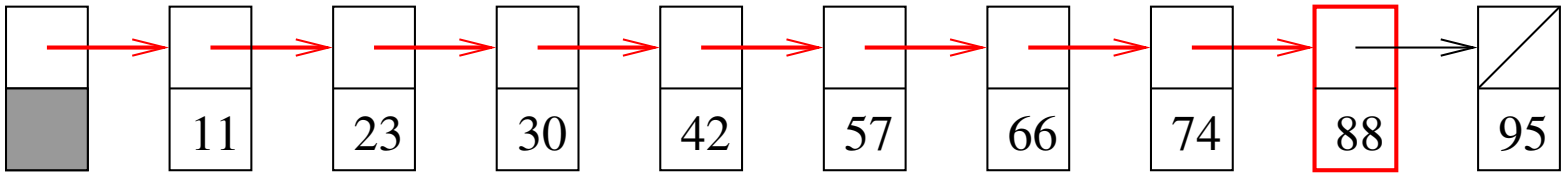
Skip list



Considere a busca por 88. É necessário examinar quase todos os N elementos da lista.

É possível acelerar este processo?

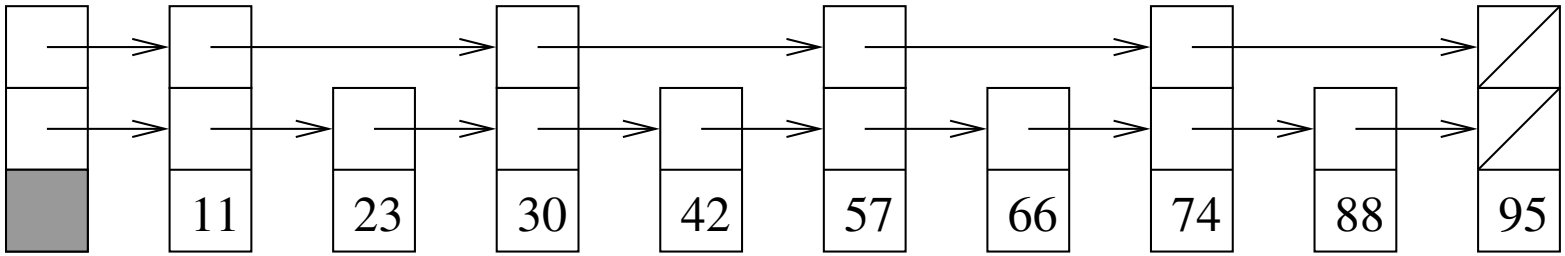
Skip list



Considere a busca por 88. É necessário examinar quase todos os N elementos da lista.

É possível acelerar este processo? Sim, ao custo de mais memória e mais codificação.

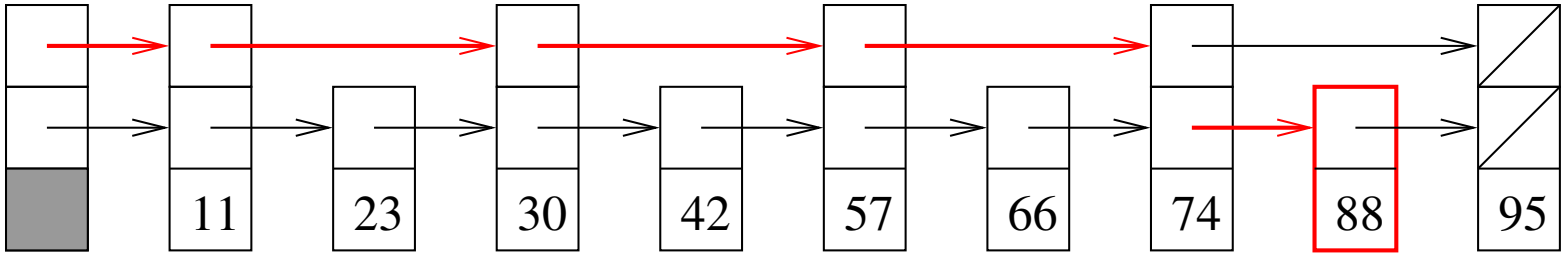
Skip list



Adicione mais um nível de apontadores como na figura.

O elemento 88 tem que estar entre dois nó de altura 2.

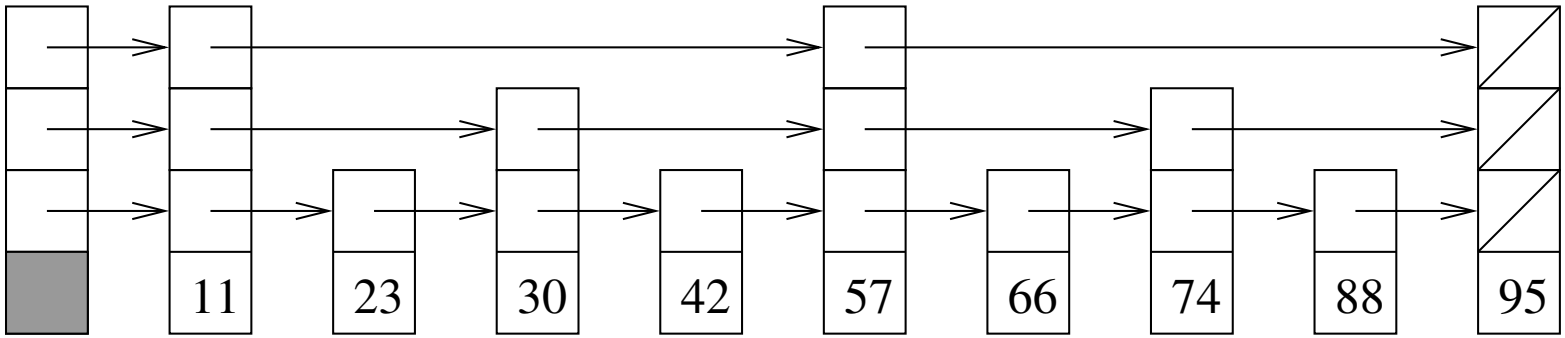
Skip list



Começando do nível 1, o algoritmo vai visitando os nós em cada nível até chegar ao nível 0 onde faz simplesmente uma busca linear em lista.

Note que só é preciso examinar $\approx N/2$ nós da lista.

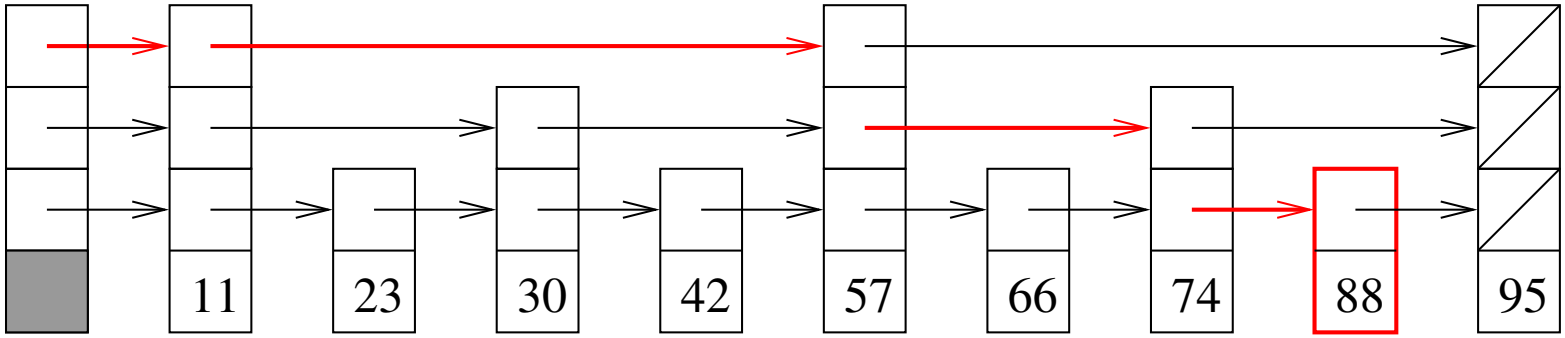
Skip list



Adicione outro nível de apontadores como na figura.

O nó 88 tem que estar entre dois nós de altura 3.

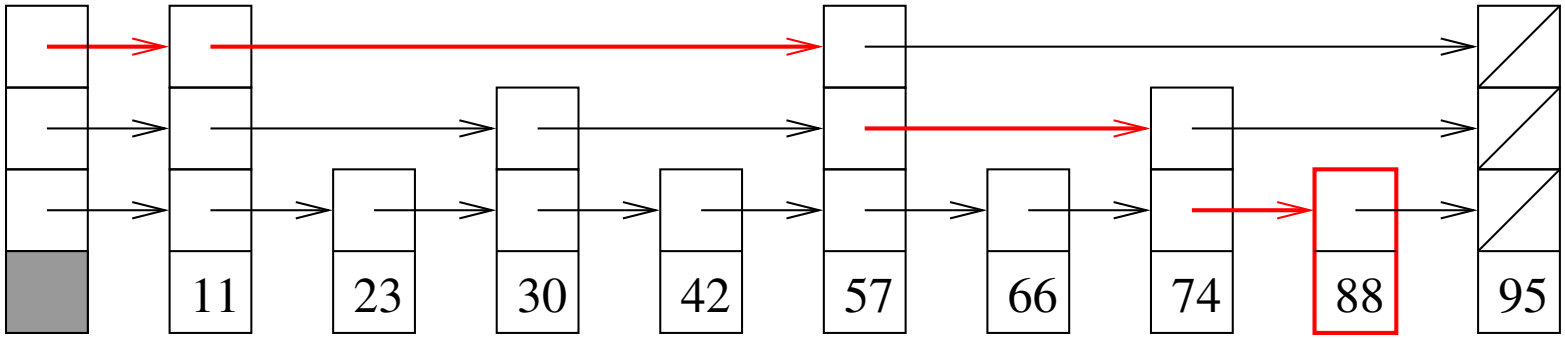
Skip list



Como antes, começamos a busca na lista do nível maior e descendo até chegar ao nível 0.

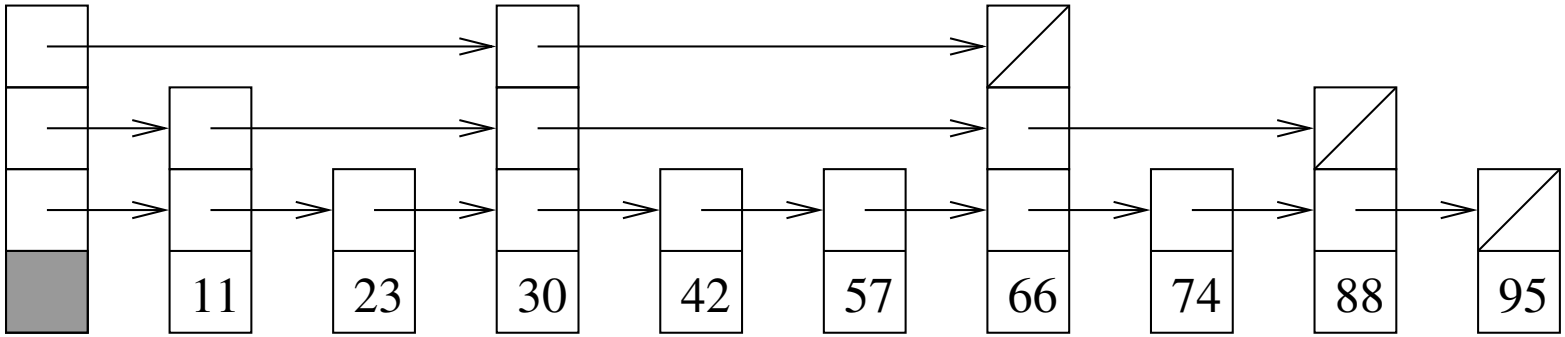
O número de nós examinado é $\approx N/4$.

Skip list



- Seja $h = \lfloor \lg N \rfloor$. Se a skip list tiver h níveis (altura h) e os apontadores estiverem (quase) igualmente espaçados, então em cada nível a busca examinha apenas um nó até chegar ao nó procurado, totalizando $O(\lg N)$ nós examinados.
- Em geral, **não** é possível garantir uma estrutura tão regular como na figura, mas em média o comportamento será bem próximo disto.

Skip list



- Cada nó tem uma altura que é gerada aleatoriamente (no momento da alocação).
- Aleatoriedade garante um comportamento próximo da configuração mais balanceada.
- O nó cabeça tem altura máxima.

Implementação de skip list

Skip list clássica

Cada nó x tem três campos:

- $x{->}chave$: chave do nó;
- $x{->}altura$: altura do nó;
- $x{->}prox[0\,..\,\mathrm{MAX}-1]$: um vetor de apontadores em que $x{->}prox[i]$ é o apontador do nível i.

Na prática, $\mathrm{MAX} = 24$ é mais que suficiente. Para economizar espaço, pode-se alocar dinamicamente o vetor $prox$. Suporemos entretanto que a alocação é estática para simplificar a discussão e o pseudo-código.

Implementação de skip list

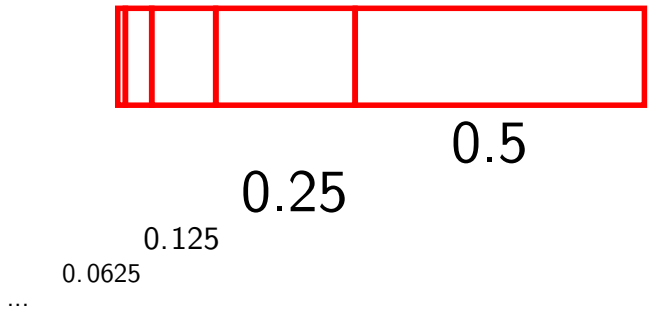
Supomos que a skip list é descrita por um registro S com dois campos:

- $S->cabeca$: um apontador para o nó cabeça da skip list;
- $S->altura$: altura do skip list.

Criação de nós com altura aleatória

CriaNo(k)

1: $x \leftarrow \text{malloc}(sizeof(No))$
2: $x{->}chave \leftarrow k$
3: $x{->}altura = \text{random}(\text{MAX})$
4: **return** x

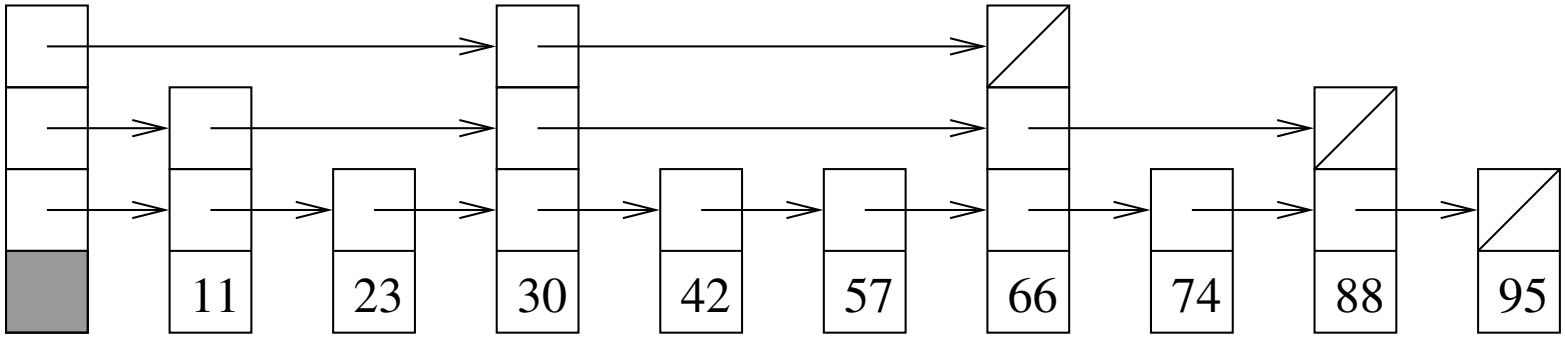


A função random(N) devolve um inteiro pseudo-aleatório entre 1 e N.

Uma função deste tipo não é trivial de implementar. A biblioteca do C não tem uma função que faz **exatamente** isto. Consulte as referências se você quiser saber mais sobre isto.

A simplificação (marcada em vermelho acima) quebra algumas propriedades importantes da Skip List. Veja a lista de exercícios.

Skip list



Note que os apontadores do nível 0 formam uma lista ligada! Isto permite visitar/processar eficientemente os elementos das listas em ordem crescente.

PercorreSkipList(S)

```
1: x ← S->cabeca->prox[0]
2: while x ≠ NULL do
3:     Visite(x)
4:     x ← x->prox[0]
```

Busca em skip list

Entrada: uma skip list S e uma chave k
Saída: o nó que contém k, ou NULL, se tal nó não existir.

BuscaSkipList(S, k)

```
1: x ← S->cabeca
2: h ← S->altura
3: for i ← h − 1 downto 0 do
4:     while x->prox[i] ≠ NULL and k > x->prox[i]->chave do
5:         x ← x->prox[i]
6: if k = x->chave then
7:     return x
8: else
9:     return NULL
```

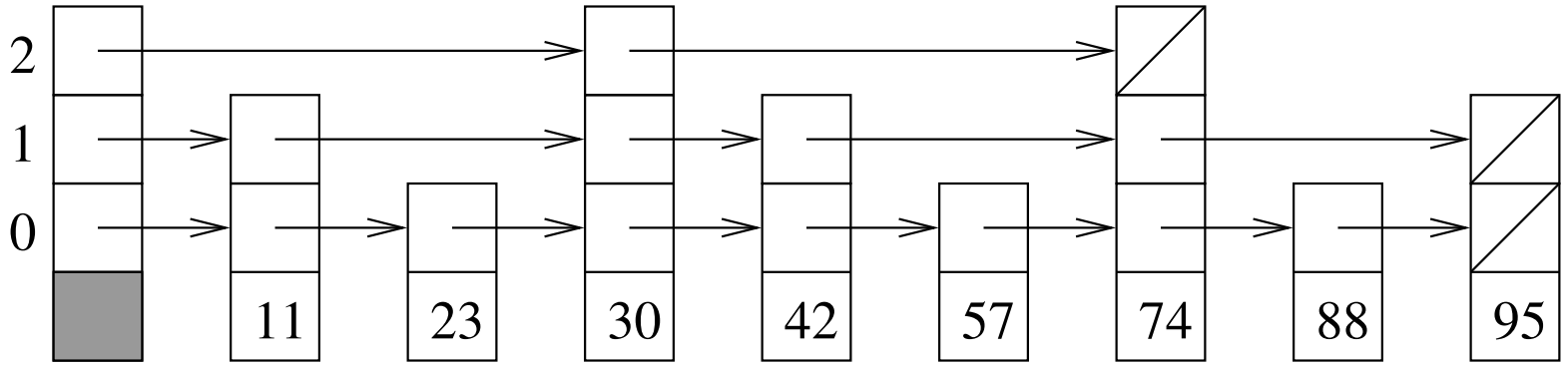
Busca em skip list – versão recursiva

Entrada: um nó skip x da skip list, uma chave k e um nível i de x
Saída: o nó que contém k, ou NULL, se tal nó não existir.

```
BuscaNivel(x, k, i)
 1: if x = NULL then
 2:     return NULL
 3: if k = x->chave then
 4:     return x
 5: if k < x->prox[i]->chave then
 6:     if k = 0 then
 7:         return NULL
 8:     else
 9:         return BuscaNivel(x, k, i − 1)
10: else
11:     return BuscaNivel(x->prox[i], k, i)
```

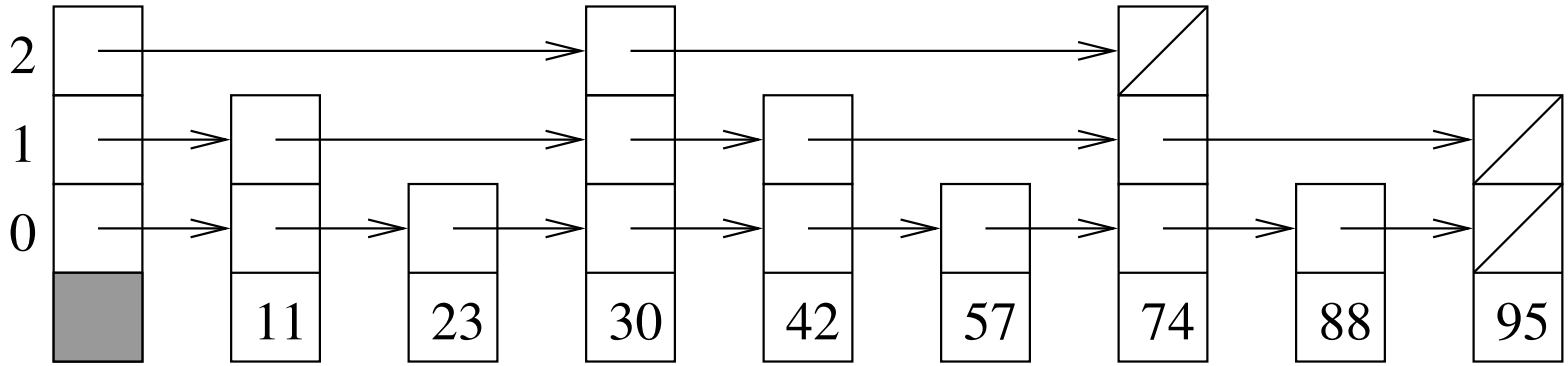
```
BuscaSkipList(S, k)
 1: return BuscaNivel(S->cabeca, k, S->altura − 1)
```

Inserção em uma skip list



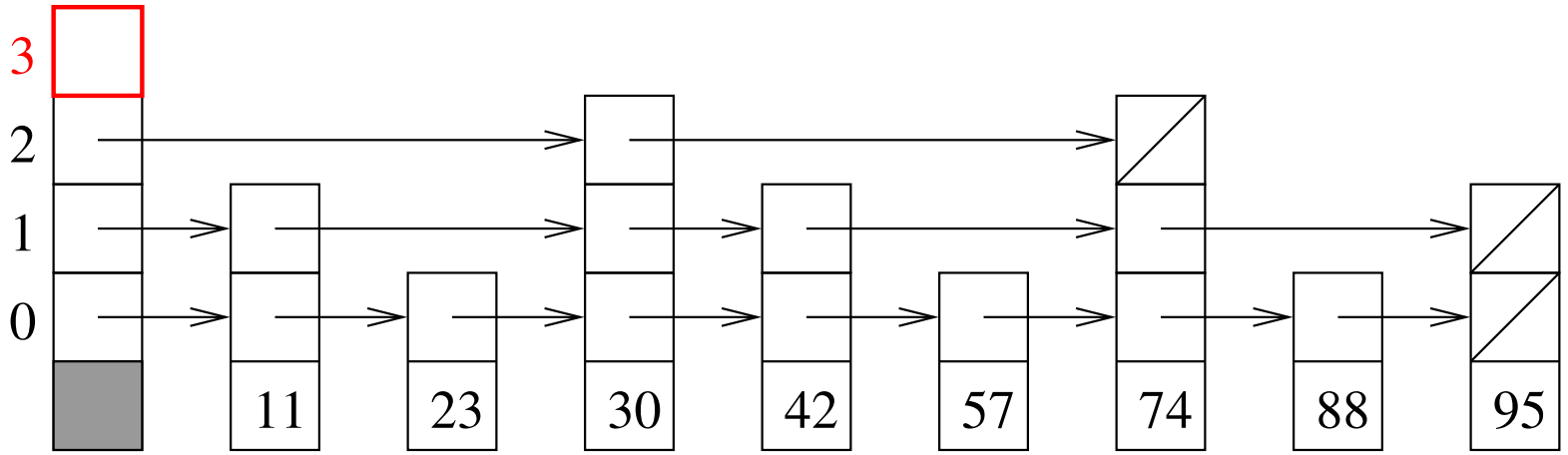
- Inicialmente cria-se um nó com a nova chave e altura gerada aleatoriamente. Digamos chave $= 66$ e altura $= 4$ na figura.

Inserção em uma skip list



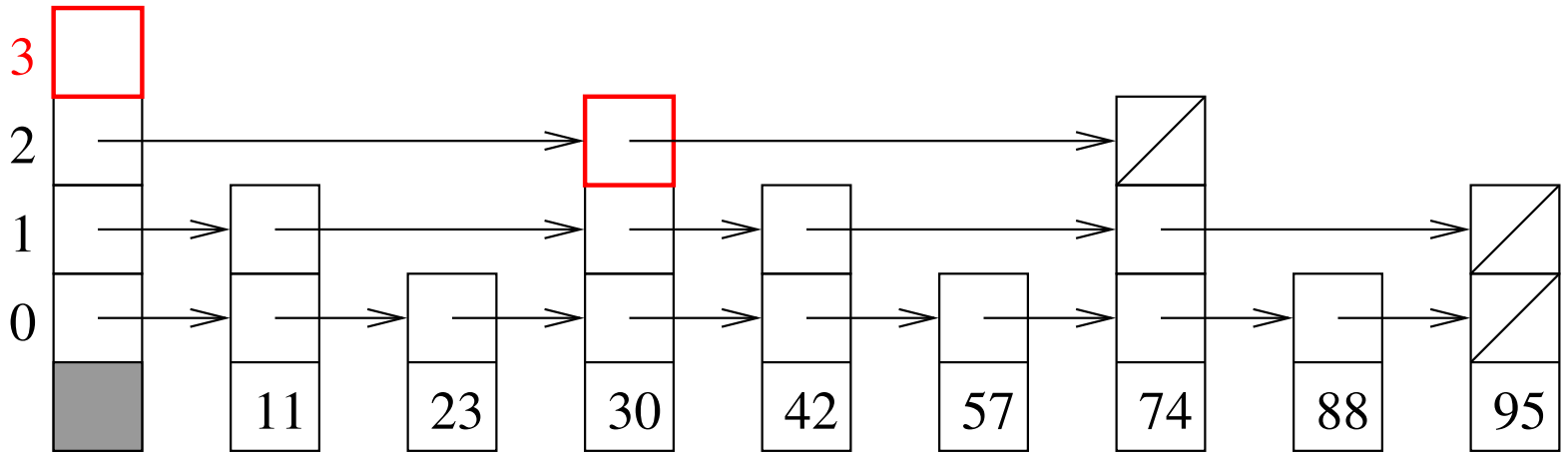
- Inicialmente cria-se um nó com a nova chave e altura gerada aleatoriamente. Digamos chave $= 66$ e altura $= 4$ na figura.
- A ideia é fazer o mesmo percurso feito na busca mas guardar os nós cujos apontadores precisam ser atualizados.

Inserção em uma skip list



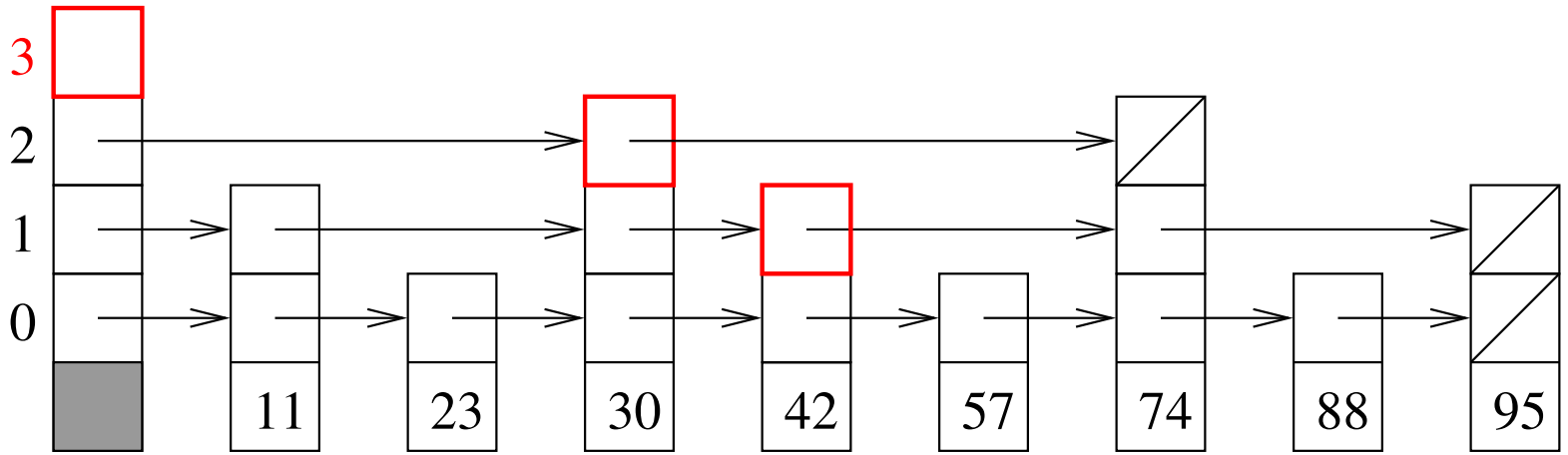
- Novo nó tem chave $= 66$ e altura $= 4$ na figura.
- fix[3]
- Como a altura do novo nó é maior que a altura da skip list, todos os apontadores do nó cabeça com nível maior que 3 são atualizados.

Inserção em uma skip list



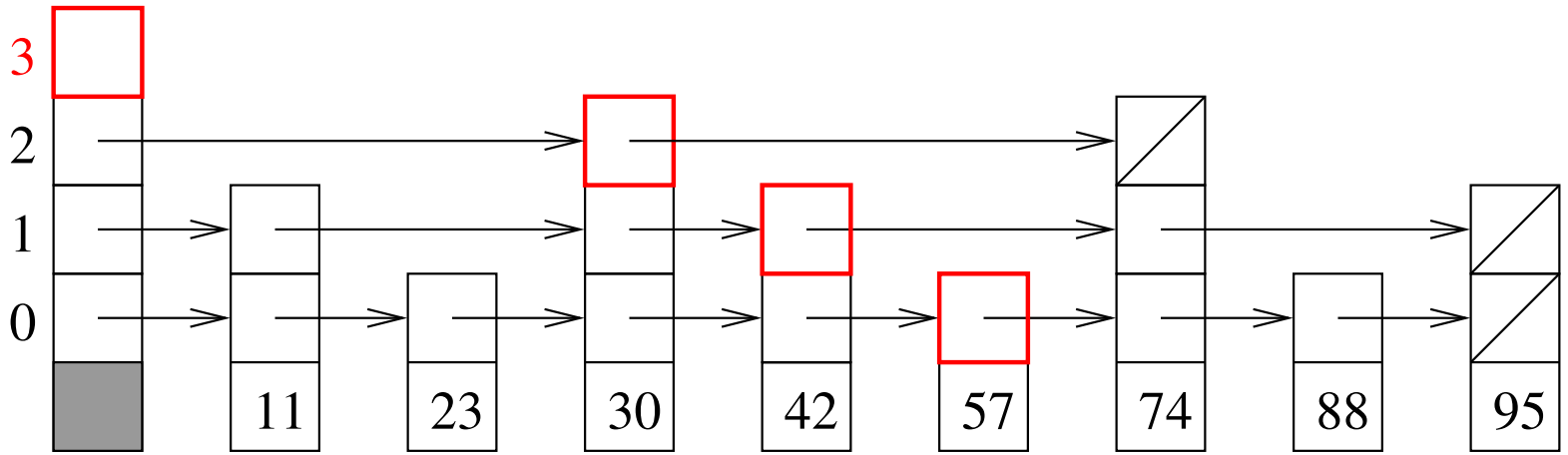
- Novo nó tem chave $= 66$ e altura $= 4$ na figura.
- fix[3], fix[2]
- Como $30 < 66 < 74$, o apontador de altura 2 de nó que contém 30 deve ser atualizado.

Inserção em uma skip list



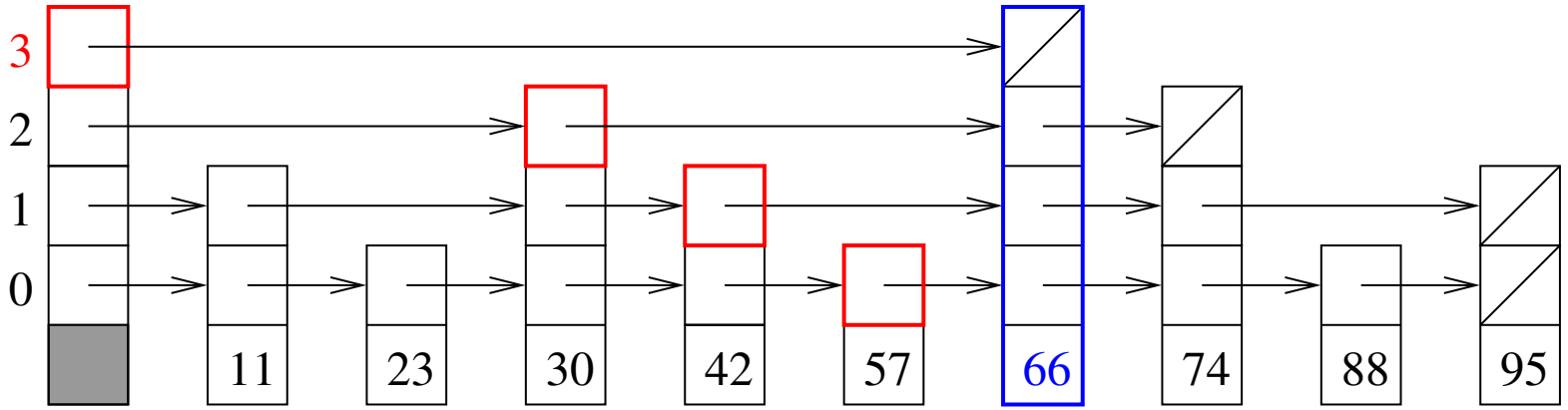
- Novo nó tem chave $= 66$ e altura $= 4$ na figura.
- fix[3], fix[2], fix[1]
- Como $42 < 66 < 74$, o apontador de altura 1 de nó que contém 42 deve ser atualizado.

Inserção em uma skip list



- Novo nó tem chave $= 66$ e altura $= 4$ na figura.
- fix[3], fix[2], fix[1], fix[0]
- Como $57 < 66 < 74$, o apontador de altura 0 de nó que contém 57 deve ser atualizado.

Inserção em uma skip list



- Novo nó tem chave $= 66$ e altura $= 4$ na figura.
- fix[3], fix[2], fix[1], fix[0]
- Chegou ao nível 0. O novo nó é inserido e os apontadores fix[ ] são atualizados.

Inserção em uma skip list

Entrada: uma skip list S e uma chave k.
Saída: skip list com um novo nó com chave k inserido.

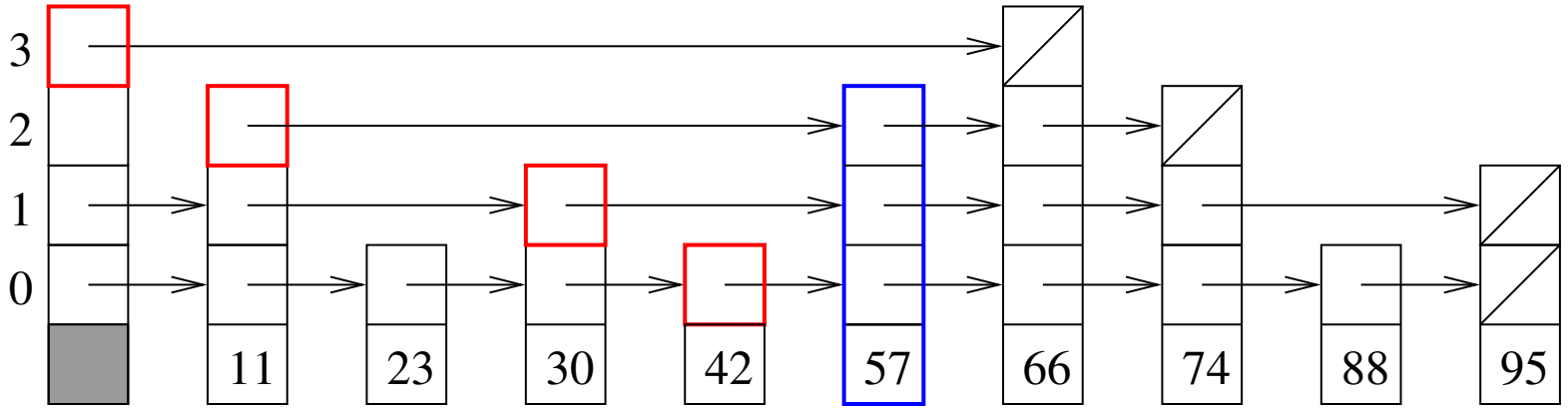
```
InsereSkipList(S, k)
 1: novo ← CriaNo(k)
 2: x = S->cabeca
 3: for i ← S->altura − 1 downto 0 do
 4:     while x->prox[i] ≠ NULL and k > x->prox[i]->chave do
 5:         x = x->prox[i]
 6:     fix[i] = x
 7: if x->prox[0] ≠ NULL and x->prox[0]->chave = k then
 8:     return 0
 9: else
10:     continua...
```

Inserção em uma skip list

```
 7: if x->prox[0] ≠ NULL and x->prox[0]->chave = k then
 8:     return 0
 9: else
10:     while novo->altura > S->altura do
11:         fix[S->altura] ← S->cabeca
12:         S->altura ← S->altura + 1
13:     i ← novo->altura − 1
14:     while i ≥ 0 do
15:         novo->prox[i] ← fix[i]->prox[i]
16:         fix[i]->prox[i] ← novo
17:         i ← i − 1
18: return 1
```

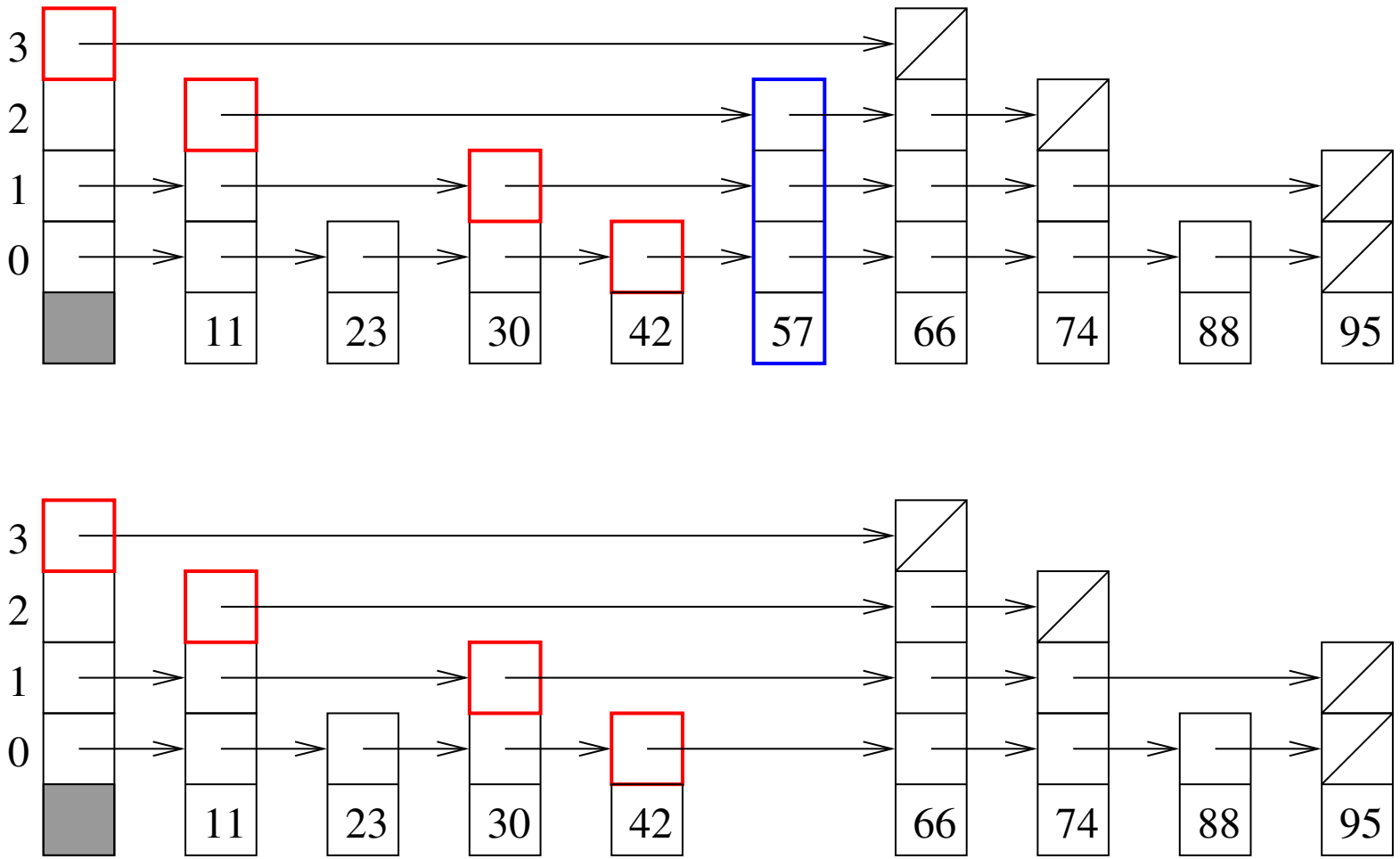
É possível escrever uma versão que não usa o vetor fix. Para isto, basta apenas atualizar os apontadores à medida que descemos nos níveis. É importante testar se k está na skip list antes de executar este processo.

Remoção em uma skip list



- A remoção é essencialmente o processo inverso da inserção.
- À medida que descemos nos níveis procurando o elemento a ser removido, guardamos os apontadores que devem ser atualizados.
- Então em cada nível executamos uma simples remoção em lista ligada.

Remoção em uma skip list



Remoção em uma skip list

```
RemoveSkipList(S, k)
1: x = S->cabeca
2: for i ← S->altura − 1 downto 0 do
3:     while x->prox[i] ≠ NULL and k > x->prox[i]->chave do
4:         x = x->prox[i]
5:     fix[i] = x
6: if x->prox[0] = NULL or x->prox[0]->chave ≠ k then
7:     return 0
8: else
9:     continua...
```

Remoção em uma skip list

```
 6: if x->prox[0] = NULL and x->prox[0]->chave ≠ k then
 7:     return 0
 8: else
 9:     fora = fix[0]->prox[0]
10:     for i = 0 to S->altura − 1 do
11:         if fix[i]->prox[i] ≠ NULL then
12:             fix[i]->prox[i] ← fix[i]->prox[i]->prox[i]
13:     while S->altura > 0 do
14:         if S->cabeca->prox[S->altura − 1] ≠ NULL then
15:             break
16:         S->altura ← S->altura − 1
17:         S->cabeca->prox[S->altura − 1] ← NULL
18:     free(fora)
19:     return 1
```

Descreva o que ocorre se tentarmos retirar a chave 30 em vez da chave 57 no exemplo acima. O código se comporta como esperado?

É possível escrever uma versão que não usa o vetor fix. Para isto, basta apenas atualizar os apontadores à medida que descemos nos níveis. É importante testar se k está na skip list antes de executar este processo.