MAC0323 Algoritmos e Estruturas de Dados II

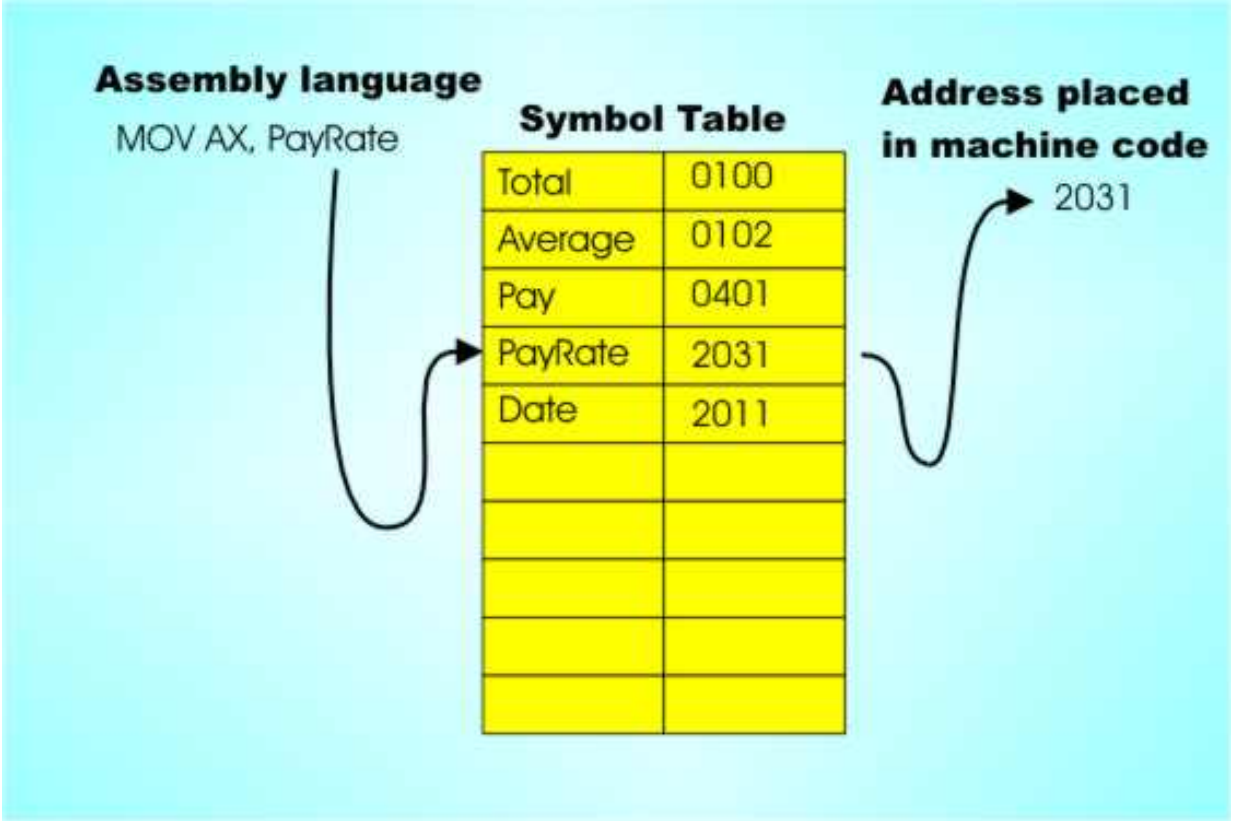
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 8

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 `STs`:

- `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`.

`STs` são também chamadas de
dictionaries, *maps* e *associative arrays*.

Interface ST.h

Arquivo ST.h		
`void`	`stInit()`	cria uma ST
`void`	`stPut(Key key, Value val)`	insere (`key`,`val`) na ST
`Value`	`stGet(Key key)`	busca o valor associado a `key`
`void`	`stDelete(Key key)`	remove (`key`,`val`) da ST
`int`	`stRank(Key key)`	no. de keys menor que `key`
`bool`	`stEmpty()`	ST está vazia?
`bool`	`stContains(Key key)`	a `key` está na ST?

Além destas, usaremos as rotinas de um iterador.

Cliente: `Index()`

`keys` = palavras e
`vals` = lista de posições onde a palavra ocorre

```
int main(int argc, char *argv) {
    int minLength = atoi(argv[1]),
        minOccurrence = atoi(argv[2]), n;
    char **words = readAllStrings(&n), *s;
    /* words[0..n-1] guarda as palavras */
    Queue q;
```

Cliente: Index()

```
stInit();
for (int i = 0; i < n; i++) {
    s = words[i];
    if (strlen(s) < minLength) continue;
    if (!stContains(s))
        stPut(s, queueInit());
    q = stGet(s);
    queuePut(q, i);
}
```

Cliente: Index()

```
    stStartIterator();
    while (stHasNext()) {
        s = stNext();
        q = stGet(s);
        if (queueSize(q) >= minOccurrence) {
            printf("%s : ", s);
            queueDump(q);
        }
    }
    stFree();
}
```

AULA 9

Consumo de tempo

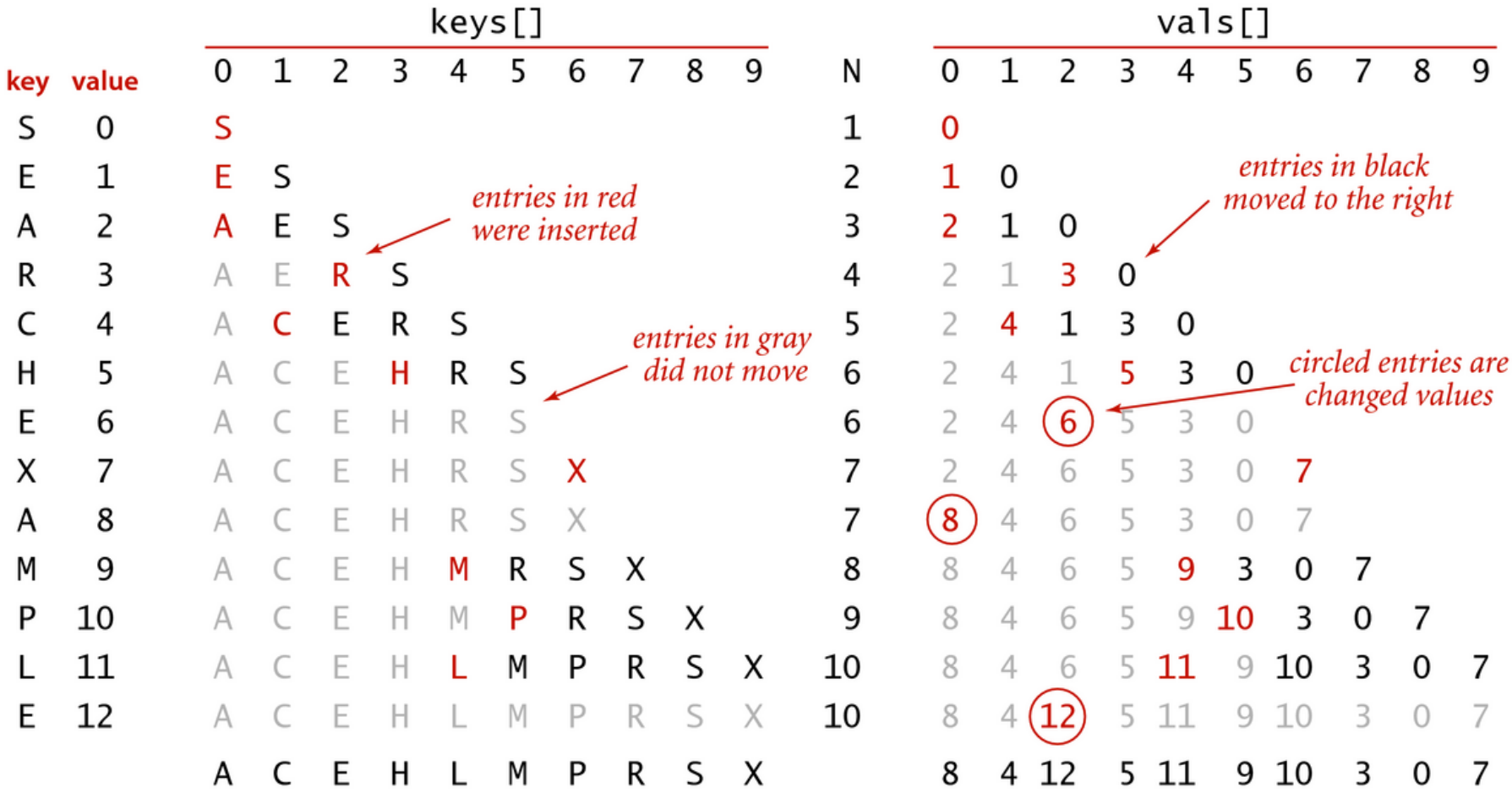
Durante a execução de `get(key)` ou `put(key,val)`, uma chave da `ST` é tocada quando comparada com `key`.
O consumo de tempo é proporcional ao **número de chaves tocadas**.

O número de chaves tocadas durante uma operação é o custo da operação.

O **custo médio** de uma busca bem-sucedida, é o quociente `c`/`n`, onde `c` é a soma dos custos das buscas de todas as chaves na tabela e `n` é o número total de chaves na tabela.

ST em vetor ordenado

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



Trace of ordered-array ST implementation for standard indexing client

Arquivo stSortedArray.c: esqueleto

```
#include "st.h"

static Key   *keys;     /* Vetor para as chaves */
static Value *vals;     /* Vetor para os valores */
static int n;           /* número de elementos na ST */
static int s;  /* tamanho dos vetores redimension. */

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

Key   stMin() {...}
Key   stMax() {...}
int   rank(Key key) {...} ♡
```

stSortedArray.c: init, isEmpty e size

```
void stInit(int cap) {
    keys = mallocSafe(cap * sizeof(Key));
    vals = mallocSafe(cap * sizeof(Value));
    n = 0;
    s = cap;
}
```

Os outros dois, stEmpty() e stSize(), são os de sempre.

Operação básica `rank()`

Retorna o posto ou `rank` de `key`,
número de chaves menores que `key`.

```
int rank(Key key) {
   int lo = 0, hi = n-1, mid, cmp;
   while (lo <= hi) {
      mid = lo + (hi - lo) / 2;
      cmp = compare(key, keys[mid]);
      if (cmp < 0) hi = mid - 1;
      else if (cmp > 0) lo = mid + 1;
      else return mid;
   }
   return lo;
}
```

Consumo de tempo: $\mathrm{O}(\lg \mathtt{n})$.

stSortedArray.c: get()

```
Value stGet(Key key) {
   int i = rank(key);
   if (i < n && compare(key, keys[i]) == 0)
      return vals[i];
   return NULL;
}
```

Consumo de tempo: $\mathrm{O}(\lg \mathtt{n})$.

stSortedArray.c: put()

```
void stPut(Key key, Value val) {
   int i = rank(key), j;
   if (i < n && compare(key,keys[i]) == 0)
      vals[i] = val;              /* atualiza o valor */
   else {
      if (n == s) resize(2*s);
      for (j = n; j > i; j--){
         keys[j] = keys[j-1];
         vals[j] = vals[j-1];
      }
      keys[i] = key; vals[i] = val;
      n++;
   }
}
```

stSortedArray.c: delete()

```
void stDelete(Key key) {
   int i = rank(key), j;
   if (i == n || compare(key,keys[i]) != 0)
      return;
   for (j = i; j < n-1; j++) {
      keys[j] = keys[j+1];
      vals[j] = vals[j+1];
   }
   n--;
   keys[n] = NULL;
   vals[n] = NULL;
   if(n > 0 && n == s/4) resize(s/2);
}
```

Consumo de tempo para criar um ST

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

Esse consumo de tempo é
devido aos deslocamentos.

Portanto, o consumo de tempo para se
criar uma lista como `n` itens é proporcional

$$1 + 2 + \cdots + \mathtt{n} - 1 \approx \mathtt{n}^2/2 = \mathrm{O}(\mathtt{n}^2).$$

stSortedArray: Conclusões

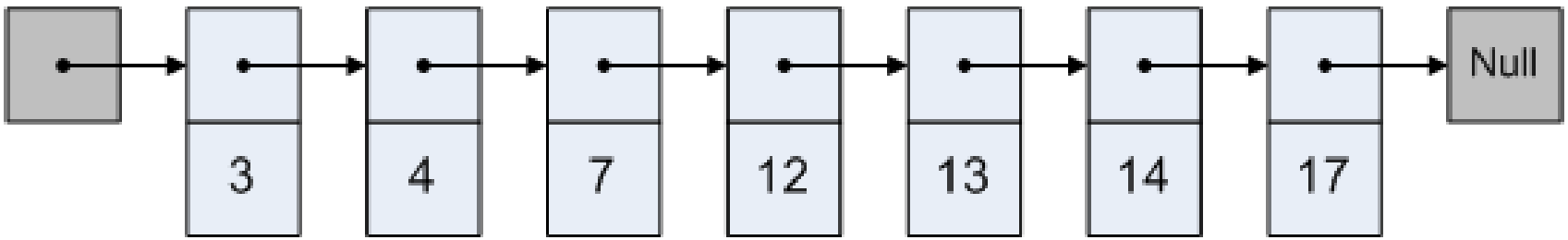
O consumo de tempo da função get()
no pior caso é proporcional a $\lg n$.

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 $O(n^2)$.

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 à chave;
3. `next`: próximo nó na lista

Qual o custo de cada função da biblioteca?

Os tipos struct node e Link

```
typedef struct node *Link;

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

Arquivo stLinkedList.c: esqueleto

```
#include "st.h"

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

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

Link  rank(Key key) {...}        /* anterior */
```

Quanto consome cada rotina em função do `n`?

stLinkedList: 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 $O(n^2)$.

Frequência de acessos

Suponha que cada chave `keys[i]` é argumento de `get(key)` com probalidade `Pr[i]`.

O custo médio `T(n)` de uma busca *bem-sucedida* é proporcional ao número

$$\texttt{Pr}[0] + 2\texttt{Pr}[1] + 3\texttt{Pr}[2] + \cdots + \texttt{n}\texttt{Pr}[\texttt{n}-1].$$

Se pudéssemos colocar as chaves na lista em qualquer ordem, para minimizar `T(n)`, deveríamos por as chaves mais frequentes no início da lista. Ou seja, deveríamos ter que

$$\texttt{Pr}[0] \geq \texttt{Pr}[1] \geq \texttt{Pr}[2] \geq \cdots \geq \texttt{Pr}[\texttt{n}-1].$$

Exemplos

Para $\Pr[0] = \Pr[1] = 0.1$, $\Pr[2] = 0.3$,
$\Pr[3] = 0.1$, $\Pr[4] = 0.4$, temos que

$$\mathtt{T(n)} = 1{\times}0.1{+}2{\times}0.1{+}3{\times}0.3{+}4{\times}0.1{+}5{\times}0.4 = 3.6$$

Se as chaves são rearranjadas em
ordem decrescente de probabilidades,
temos que

$$\mathtt{T(n)} = 1{\times}0.4{+}2{\times}0.3{+}3{\times}0.1{+}4{\times}0.1{+}5{\times}0.1 = 2.2$$

Mais exemplos

Se $\mathsf{Pr}[0] = \mathsf{Pr}[1] = \cdots = \mathsf{Pr}[\mathtt{n}-1] = 1/\mathtt{n}$,
então

$$\mathtt{T}(\mathtt{n}) = (\mathtt{n}+1)/2.$$

Se $\mathsf{Pr}[0] = 1/2, \mathsf{Pr}[1] = 1/2^2, \ldots,$
$\mathsf{Pr}[\mathtt{n}-2] = 1/2^{\mathtt{n}-1}, \mathsf{Pr}[\mathtt{n}-1] = 1/2^{\mathtt{n}}$,
então

$$\mathtt{T}(\mathtt{n}) = 2 - \frac{1}{2^{\mathtt{n}}} < 2.$$

Algumas distribuições de probabilidade

G.K.Zipf observou que a $\mathtt{i}$-ésima palavra mais frequente em um texto em liguagem natural ocorre com frequência aproximada $1/\mathtt{i}$. Nesse caso,

$$\mathtt{Pr}[0] = \mathtt{c}, \mathtt{Pr}[1] = \mathtt{c}/2, \ldots, \mathtt{Pr}[\mathtt{n}-1] = \mathtt{c}/\mathtt{n},$$

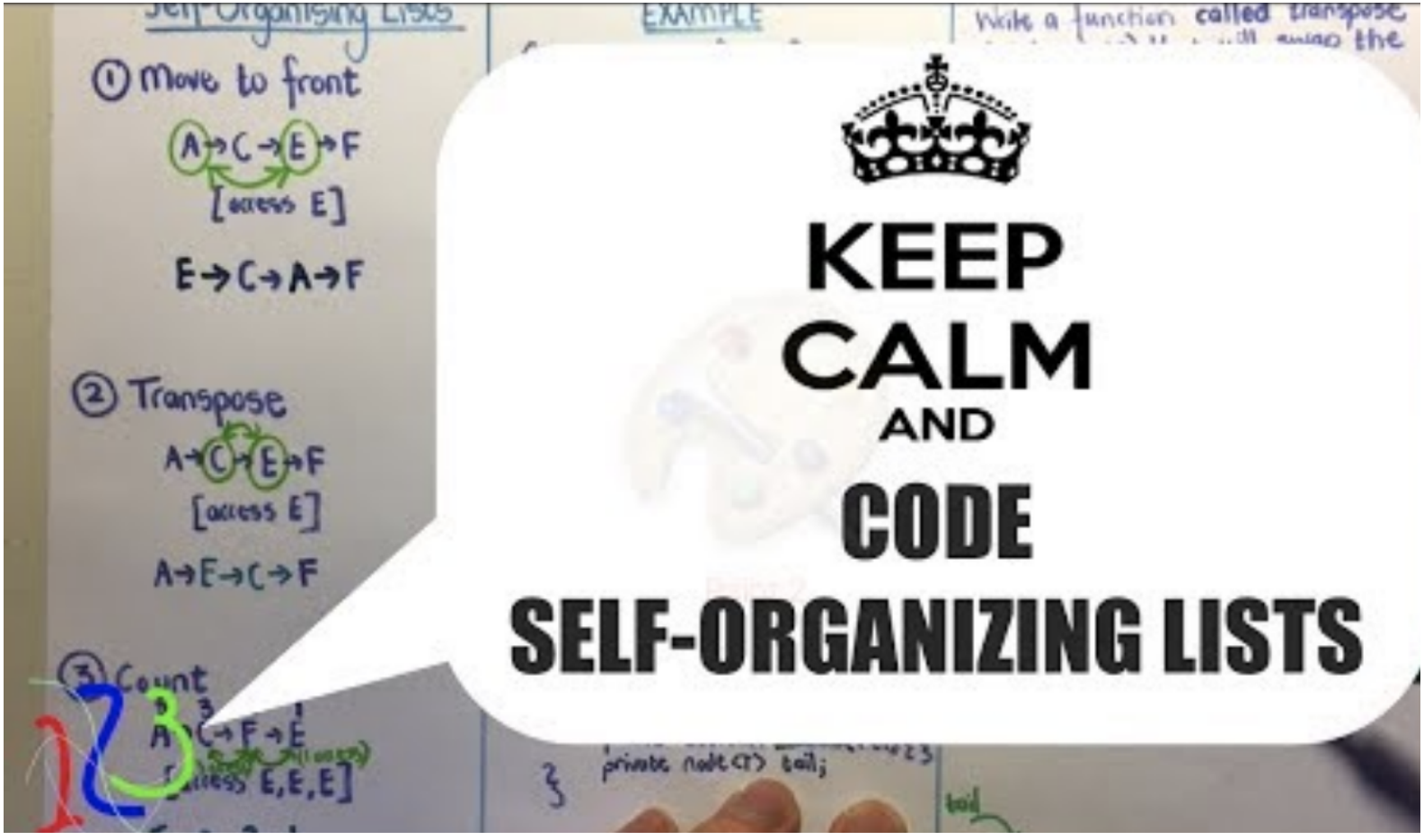
onde $\mathtt{c} = 1/\mathtt{H_n}$ e $\mathtt{H_n} = 1 + 1/2 + 1/3 + \cdots + 1/\mathtt{n}$, logo

$$\mathtt{T(n)} = \frac{\mathtt{n}}{\mathtt{H_n}}.$$

Outra distribuição que aproxima a realidade diz que 80% das consultas recaem sobre 20% das chaves. Nesse caso,

$$\mathtt{T(n)} \approx 0.122\,\mathtt{n}.$$

Self-organizing lists



Fonte: https://www.youtube.com/watch?v=cryoqB8TPRA

Self-organizing lists

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

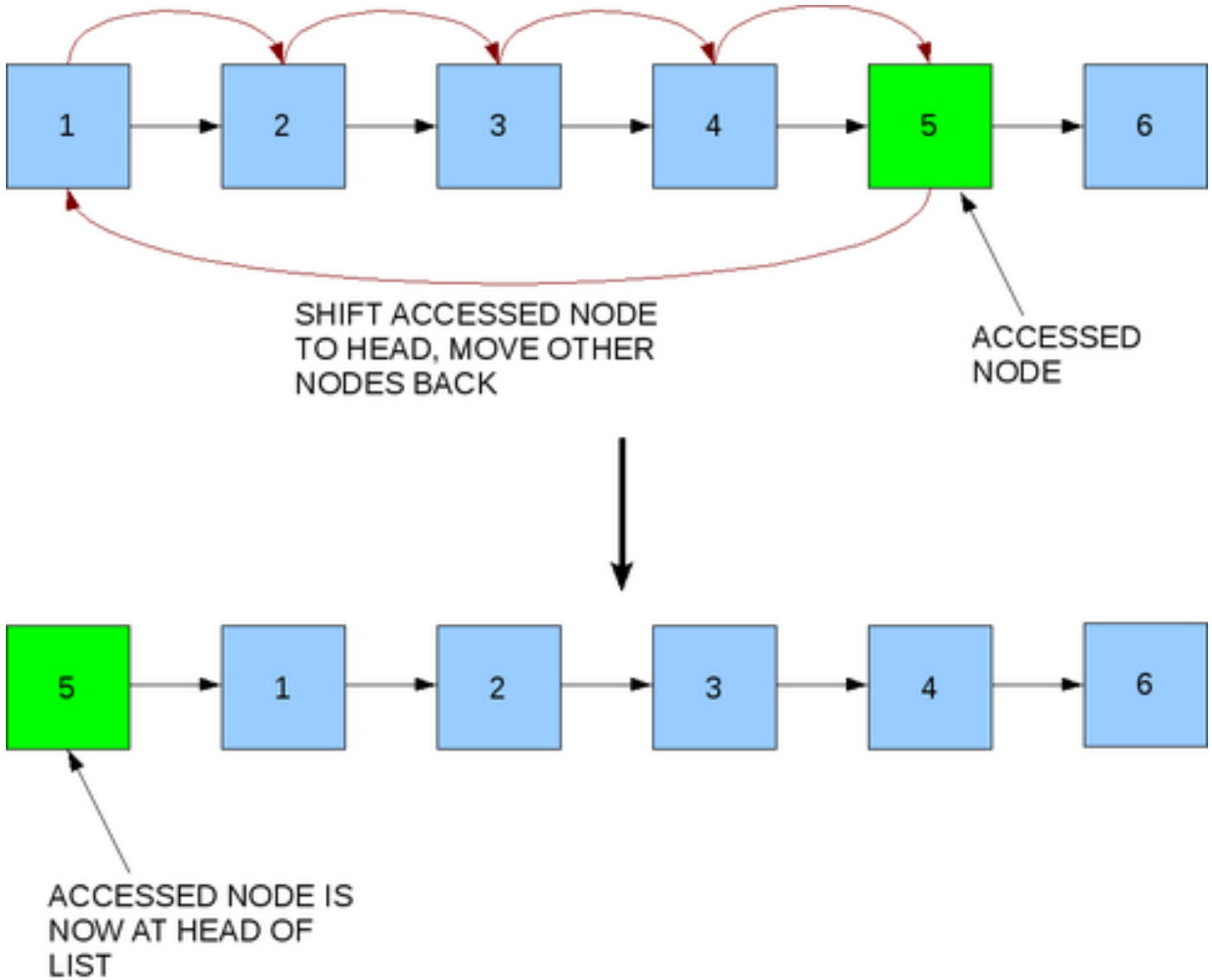
Self-organizing lists

Uma 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*).



Método *mova para frente*

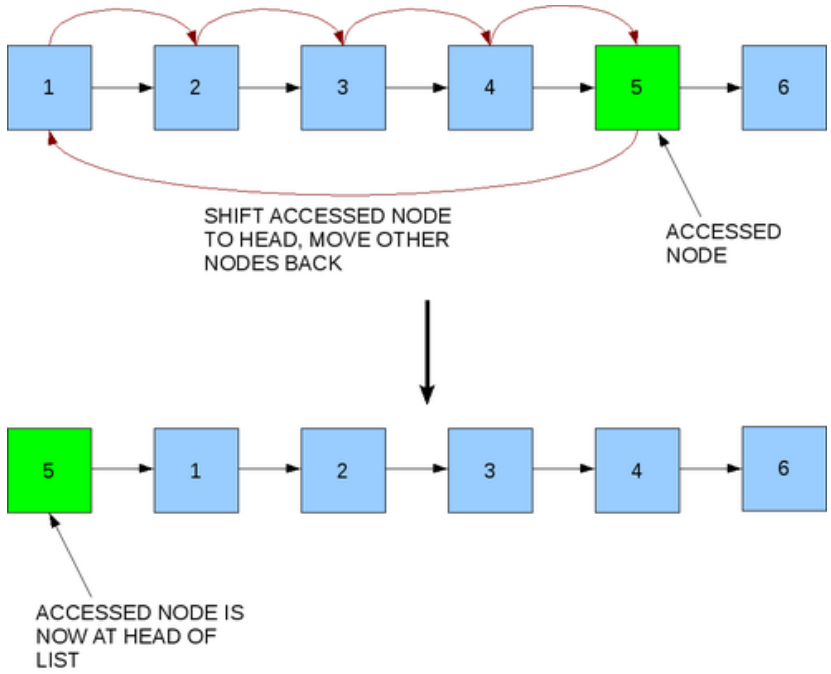
Pode-se demonstrar que o número médio de comparações para encontrar uma chave usando *move to front* tende a

$$\mathtt{T(n)} = \frac{1}{2} + \sum_{\mathtt{i},\mathtt{j}} \frac{\mathtt{Pr[i]Pr[j]}}{\mathtt{Pr[i]} + \mathtt{Pr[j]}}$$

Método *mova para frente*

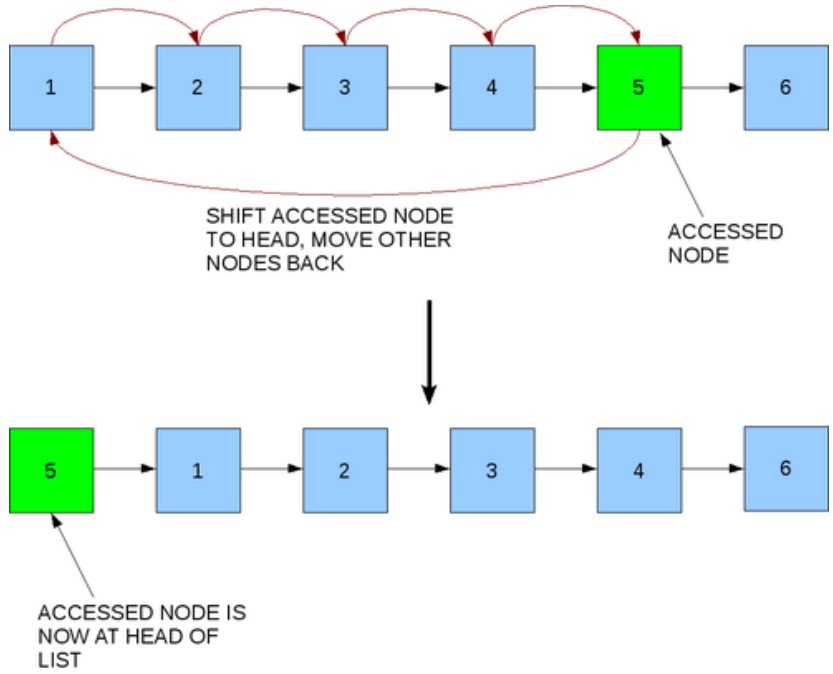
Vantagens:

- fácil de se implementar;
- não utiliza espaço extra;
- se adapta rapidamente à sequência de acessos.



Fonte: Wikipedia

Método *mova para frente*



Fonte: Wikipedia

Vantagens:

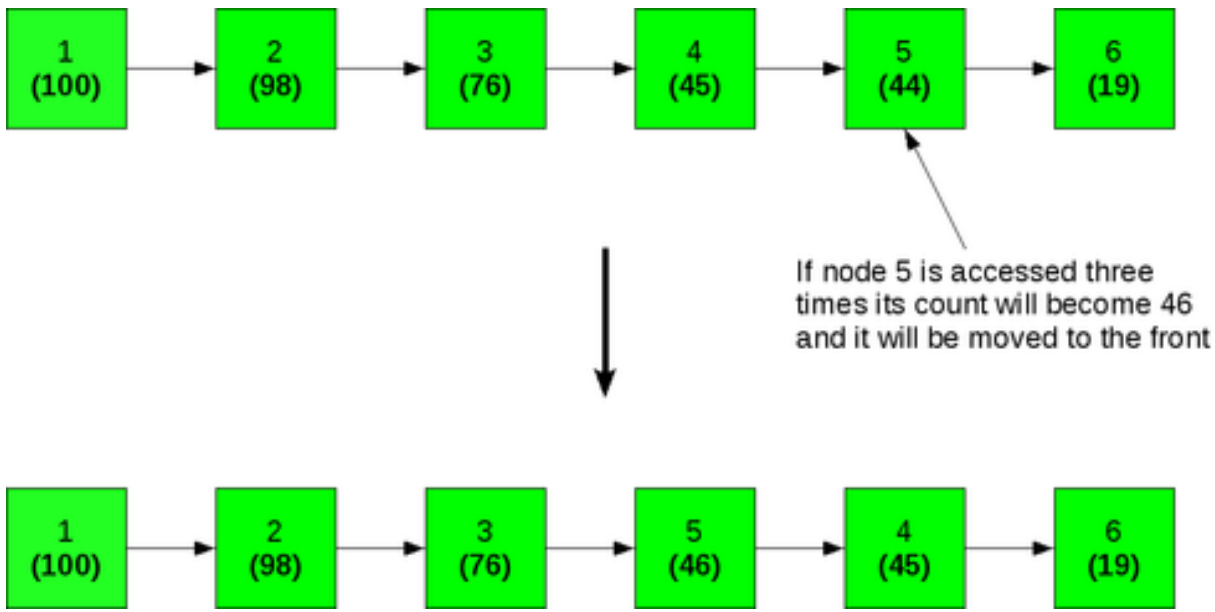
- fácil de se implementar;
- não utiliza espaço extra;
- se adapta rapidamente à sequência de acessos.

Desvantagens:

- pode sobrevalorizar chaves não acessadas de maneira frequente;
- a *memória* dos últimos itens acessados é relativamente curta.

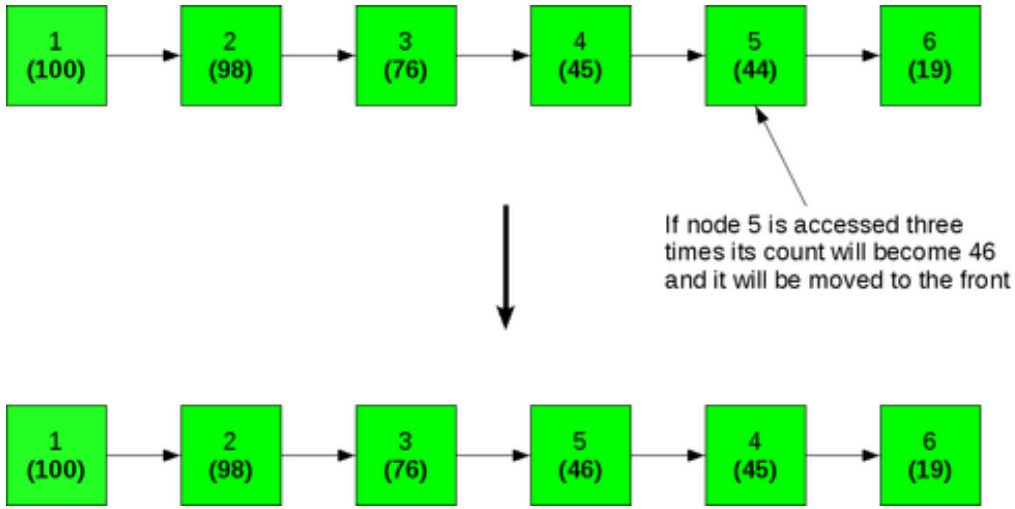
Método *do contador*

Cada chave possui um contador de consultas.
A lista é mantida em ordem decrescente desse contador (*Count Method*).



Fonte: Wikipedia

Método *do contador*



Fonte: Wikipedia

Vantagens:

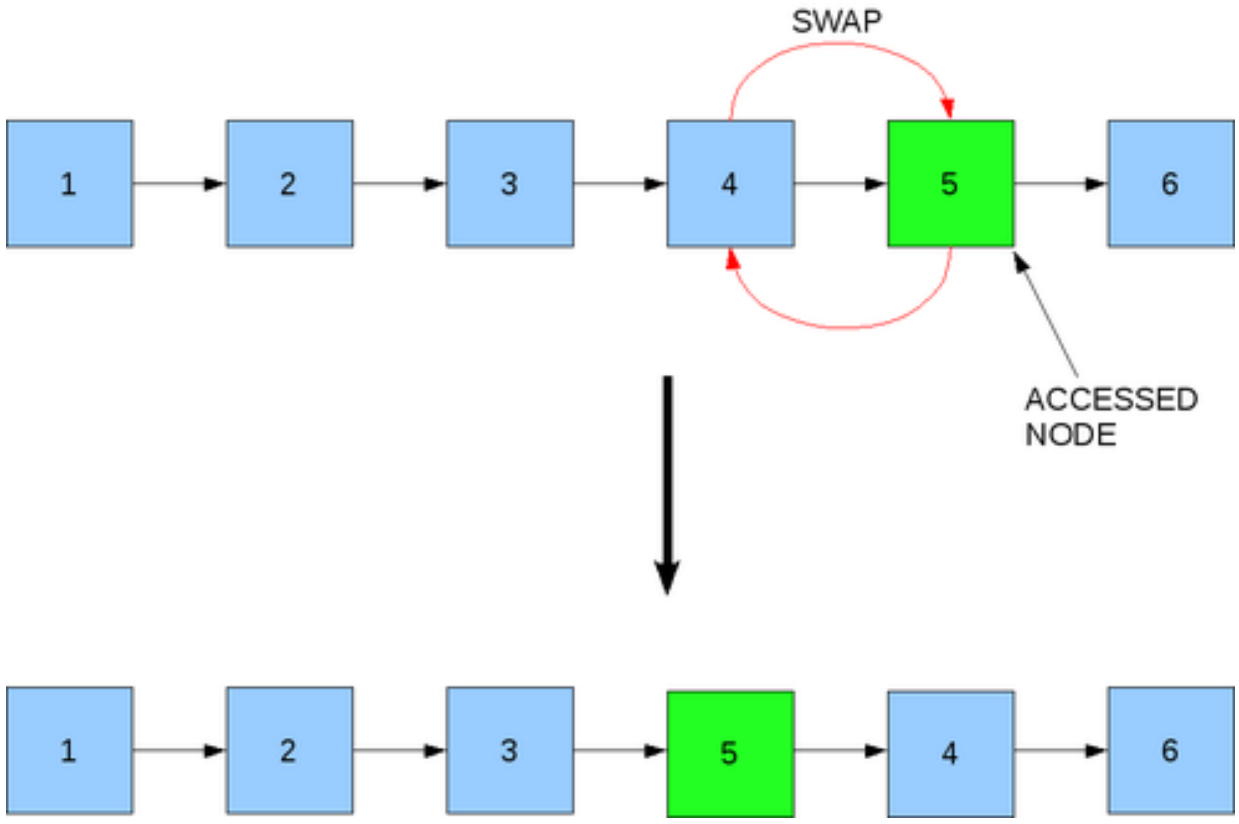
- reflete o padrão de acessos;

Desvantagens:

- deve manter um contador para cada `key-val`;
- não se adapta rapidamente a mudanças no padrão de acessos;

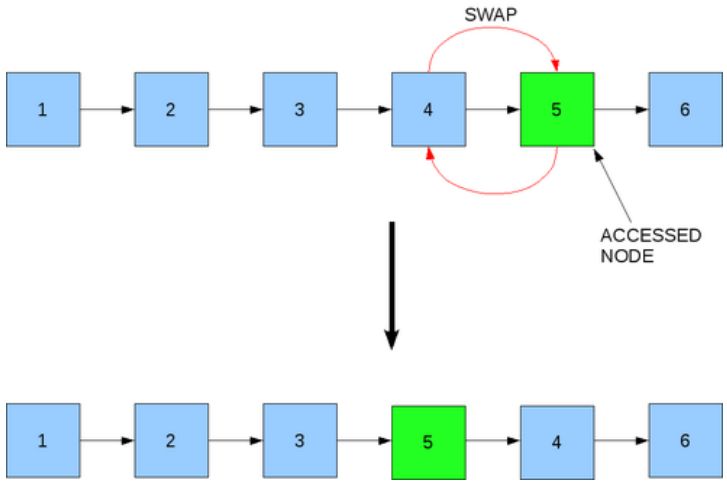
Método *da transposição*

Toda chave consultada é trocada de posição com seu predecessor (*Transpose Method*).



Fonte: Wikipedia

Método *da transposição*



Fonte: Wikipedia

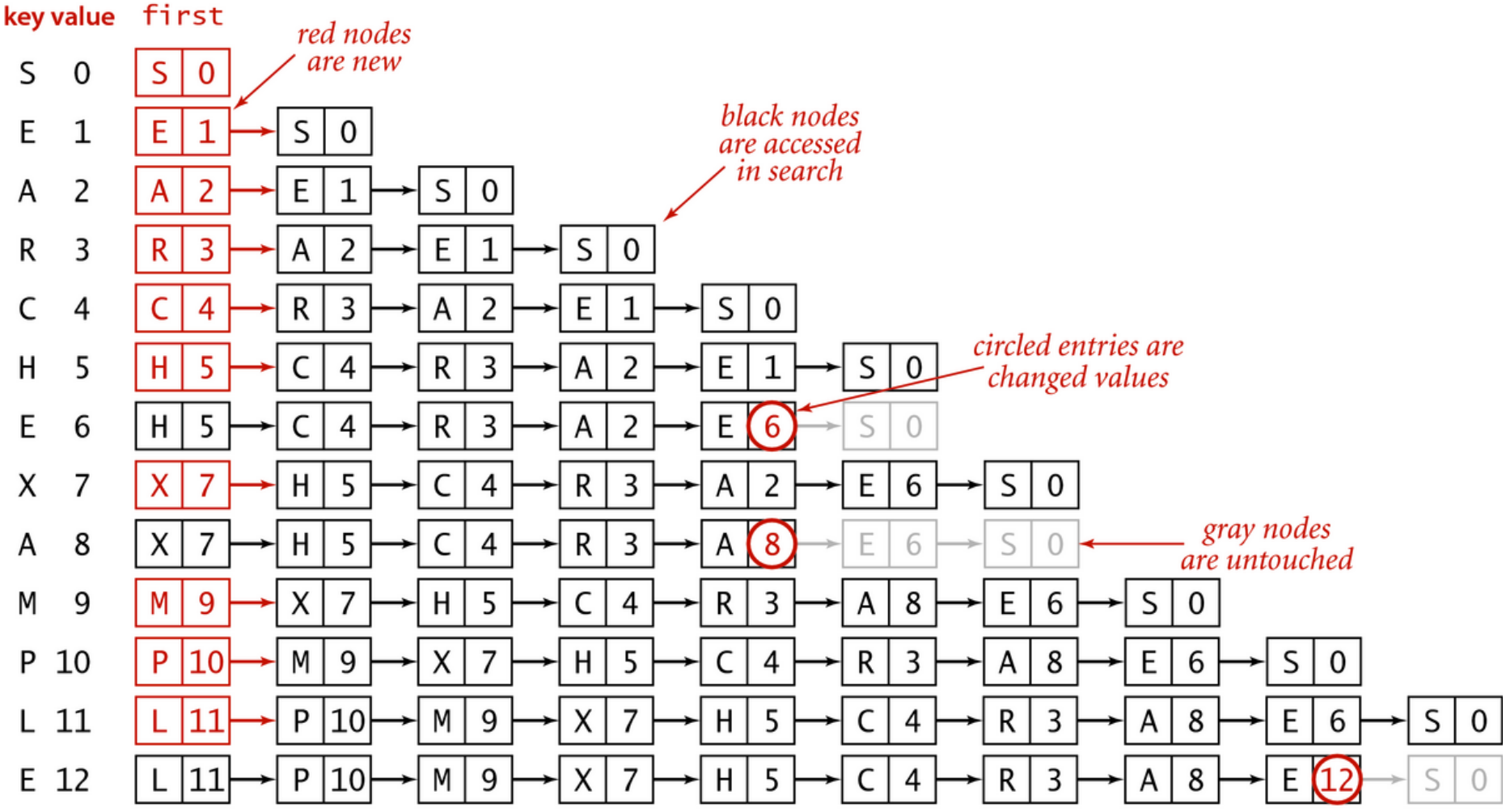
Vantagens:

- fácil de se implementar;
- não utiliza espaço extra;
- pares `key`-`val` frequentemente acessados estão provavelmente perto do início.

Desvantagens:

- mais conservador que *move to front*: gasta mais acessos até mover um par `key`-`val` para o início.

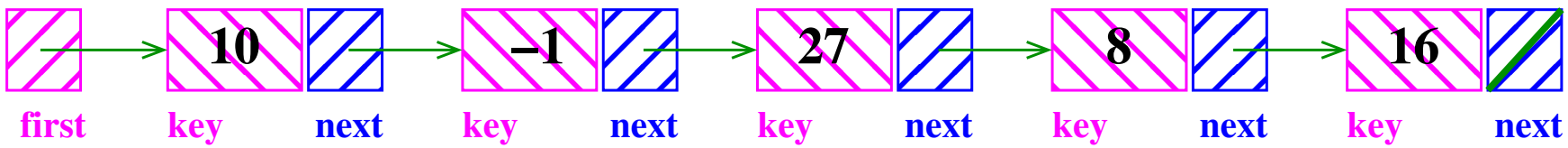
Simulação de lista não ordenada



Trace of linked-list ST implementation for standard indexing client

ST em lista ligada com MTF

Implementação usa uma lista ligada
não ordenada com a heurística *move to front*.



Mesmo nó da implementação anterior,
com três campos:

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

Arquivo stMTF.c: esqueleto

```
#include "st.h"

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

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

Link  rank(Key key) {...} ♡
```

O esqueleto é igual ao anterior.

stMTF: stInit, stEmpty e stSize

```
void stInit(int cap) {
   first.next = NULL;
   n = 0;
}
```

Os outros dois, stEmpty() e stSize(),
são os de sempre.

Operação básica `rank()`

Retorna o apontador para a célula anterior a que contém `key`, ou para o último se `key` não está presente.

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

Consumo de tempo: $O(n)$.

stMTF: get()

```
Value stGet(Key key) {
   Link p = rank(key);
   if (p->next != NULL) {
      moveToFront(p);
      return first.next->val;
   }
   return NULL;
}
```

Consumo de tempo: $O(n)$.

stMTF: moveToFront()

Move a célula seguinte a p para o início da lista.

```
void moveToFront(Link p) {
   Link q = p->next;
   p->next = q->next;
   q->next = first.next;
   first.next = q;
}
```

stMTF: put()

```
void stPut(Key key, Value val) {
   Link p = rank(key);
   if (p->next != NULL) {
      p->next->val = val;
      moveToFront(p);
   }
   else {
      p = newNode(key, val, first.next);
      first.next = p;
      n++;
   }
}
```

Consumo de tempo: $O(n)$.

stMTF: delete()

```
void stDelete(Key key) {
   Link p = rank(key), q;
   if (p->next == NULL) return;
   q = p->next;
   p->next = q->next;
   freeNode(q);
   n--;
}
```

Consumo de tempo: $O(n)$.

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.

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.

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	tempo
vetor	59.5
vetor mtf	7.6
vetor ordenado	1.5
lista ligada	147.1
lista ligada mtf	15.3
lista ligada ordenada	115.227

Tempos em segundos obtidos com `StopWatch`.

Skip lists

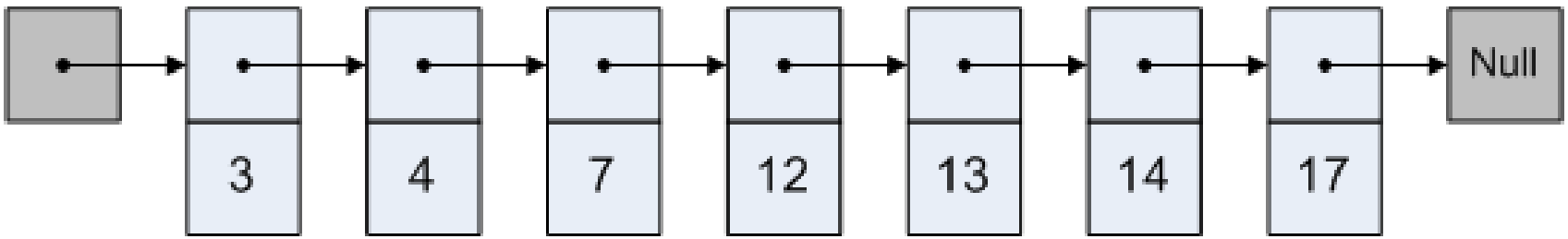
A Probabilistic Alternative to Balanced Trees

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)

Lista (simplemente) ligada

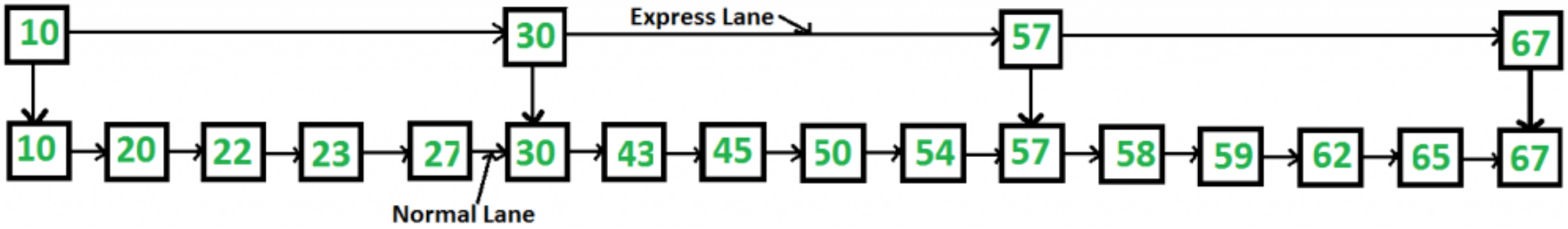


Fonte: Skip lists are fascinating!

Cada nó `x` tem três campos:

1. `key`: chave do item;
2. `val`: valor associado à 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 à chave;
3. `next`$[0]$: próximo nó na lista no nível 0
4. `next`$[1]$: próximo nó na lista no nível 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

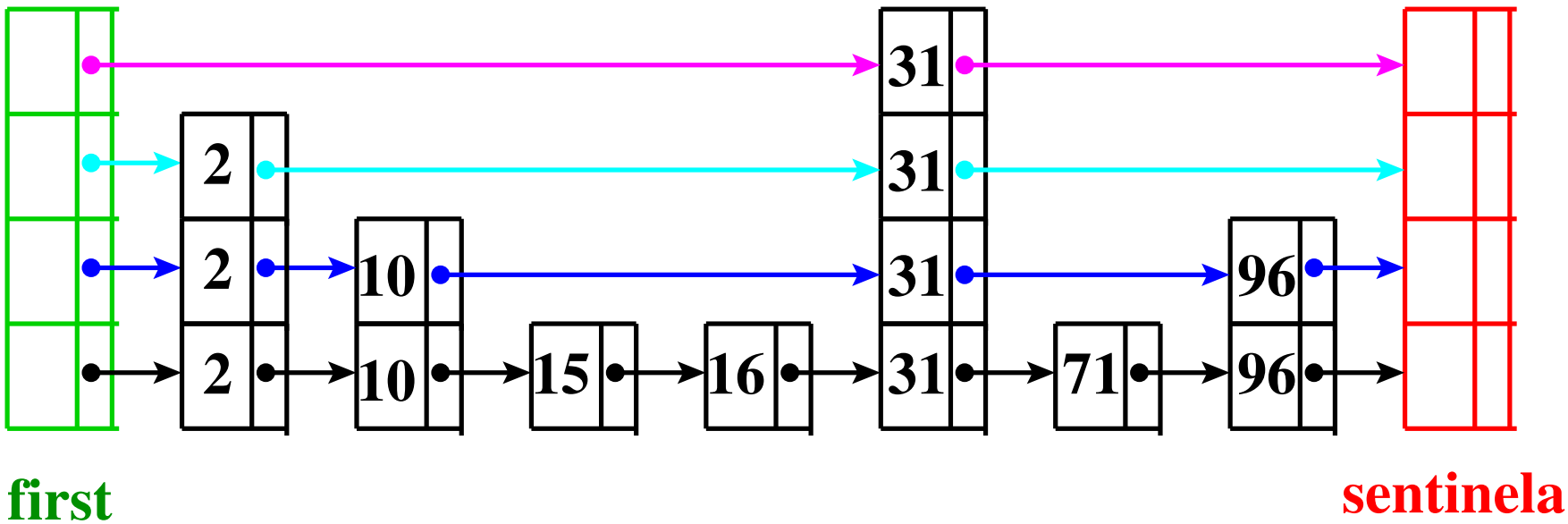
Consumo de tempo de `get()` é no máximo

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

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

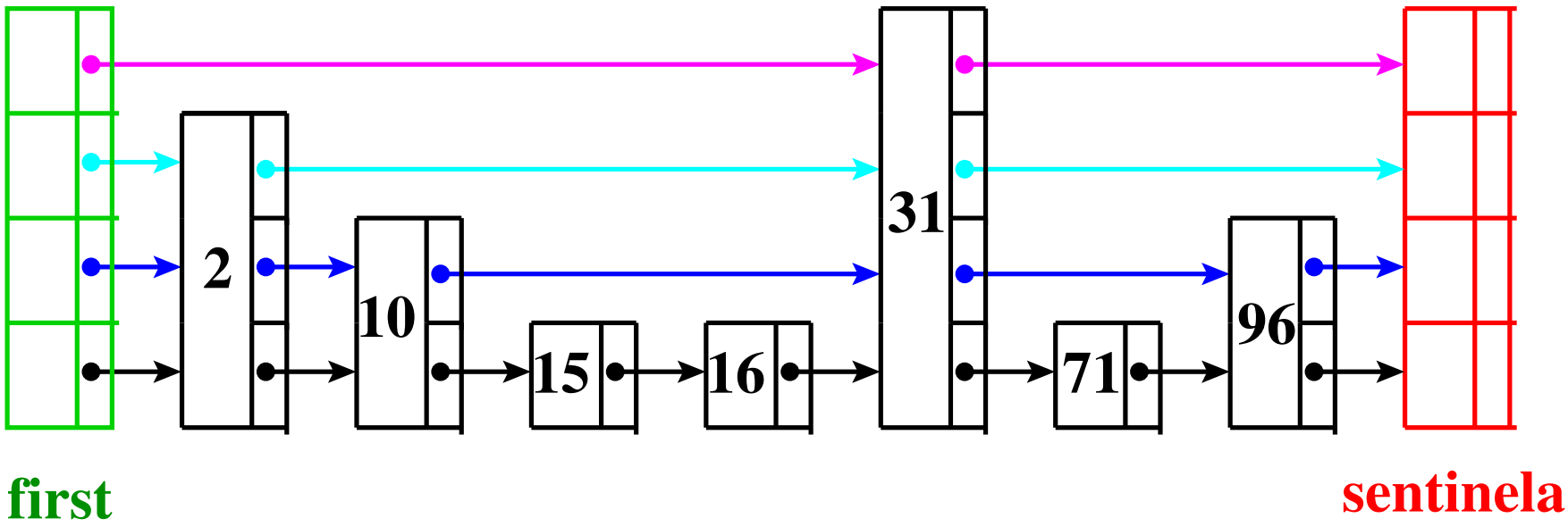
De fato, $\sqrt{n}$ é ponto de mínimo de $x + n/x$.

Multiplas listas



- `key`s ordenadas
- `first` e `setinela` em lista

Skip list



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

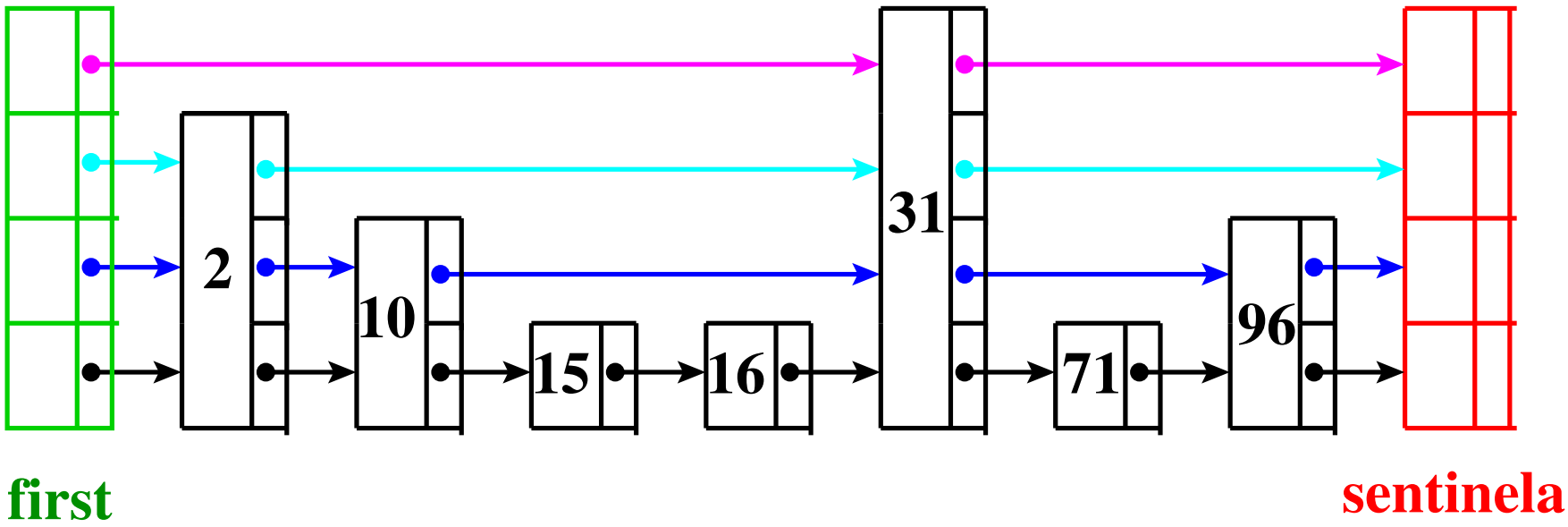
Os tipos struct node e Link

```
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;
}
```

Skip list



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