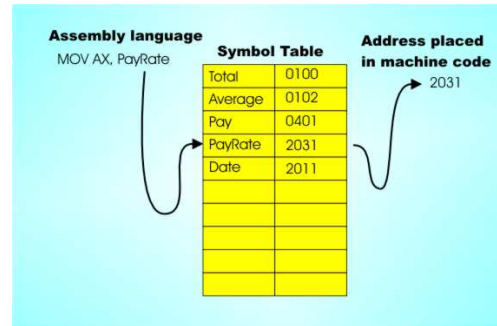


# AULA 7

## 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 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 *dictionary*, *maps* e *associative arrays*.

## API ST

```
public class ST<Key, Value>
```

```
public class ST
```

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

## Cliente: Index()

**keys** = palavras e

**vals** = lista de posições onde a palavra ocorre

```
public static void main(String[] args) {  
    int minLength =  
        Integer.parseInt(args[0]);  
    int minOccurrence =  
        Integer.parseInt(args[1]);  
    String[] words =  
        StdIn.readAllStrings();  
    // build ST of words and locations  
    ST<String, Queue<Integer>> st =  
        new ST<String, Queue<Integer>>();  
}
```

## Cliente: Index()

```
for (int i = 0; i < words.length; i++){
    Strings = words[i];
    if (s.length() < minLength)
        continue;
    if (!st.contains(s)) {
        st.put(s, new Queue<Integer>());
    }
    Queue<Integer> q = st.get(s);
    q.enqueue(i);
}
```

Navigation icons

## 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.

Navigation icons

## Classe BinarySearchST: esqueleto

```
public class BinarySearchST<Key extends
    Comparable<Key>, Value> {
    private Key[] keys;
    private Value[] vals;
    private int n = 0;
    public BinarySearchST(cap) {...}
    public Value get(Key key) {...}
    public void put(Key key, Value val){}
    public void delete(Key key) {...}
    public Key minM() {...}
    public Key max() {...}
    public int rank(Key key) {...}
}
```

Navigation icons

## Cliente: Index()

```
for (Strings: st.keys()) {
    Queue<Integer> q= st.get(s);
    if (q.size() >= minOccurrence) {
        StdOut.println(s + : + q);
    }
}
```

Navigation icons

## ST em vetor ordenado

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

	keys[]											vals[]										
key	value	0	1	2	3	4	5	6	7	8	9	N	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

Trace of ordered-array ST implementation for standard indexing client

Navigation icons

## BinarySearchST: construtor

```
public class BinarySearchST<Key extends
    Comparable<Key>, Value> {
    private Key[] keys;
    private Value[] vals;
    private int n = 0;
    public BinarySearchST(cap) {
        keys = (Key[]) new Comparable[cap];
        vals = (Value[]) new Object[cap];
    }
}
```

Navigation icons

## Operação básica rank()

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

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

Consumo de tempo:  $O(\lg n)$ .

## BinarySearchST put()

```
public void put(Key key, Value val) {
    if (val == null)
        delete(key); return;
    int i = rank(key);
    if (i < n && key.equals(key)){
        vals[i] = val; return; }
    if (n == keys.length)
        resize(2*keys.length);
    for (int j = n; j > i; j--){
        keys[j] = keys[j-1];
        vals[j] = vals[j-1];
    }
    keys[i] = key; vals[i] = val; n++;
}
```

## 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 a

$$1 + 2 + \dots + n - 1 \approx n^2/2 = O(n^2).$$

## BinarySearchST get()

```
public Value get(Key key) {
    int i = rank(key);
    if (i < n && key.equals(keys[i]))
        return vals[i];
    return null;
}
```

Consumo de tempo:  $O(\lg n)$ .

## BinarySearchST delete()

```
public void delete(Key key) {
    if (isEmpty()) return;
    int i = rank(key);
    if (i == n || !key.equals(keys[i]))
        return;
    for (int 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 == keys.length/4)
        resize(keys.length/2);
}
```

## Conclusão

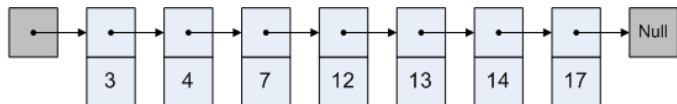
O consumo de tempo da função `get()` no pior caso é proporcional a  $\lg n$ .

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 a chave;
3. **next**: próximo nó na lista

◀ ▶ ⏪ ⏩ 🔍

◀ ▶ ⏪ ⏩ 🔍

## Classe LinkedListST: esqueleto

```
public class LinkedListST<Key extends
    Comparable<Key>, Value> {
    private Node first; // nó cabeça
    private int n;
    private class Node {...}
    public LinkedListST() {...}
    public Value get(Key key) {...}
    public void put(Key key, Value val){}
    public void delete(Key key) {...}
    public Key minM() {...}
    public Key max() {...}
    public Node rank(Key key) {...} ♥
}
```

◀ ▶ ⏪ ⏩ 🔍

◀ ▶ ⏪ ⏩ 🔍

## LinkedListST: get()

```
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;
}
```

Consumo de tempo:  $O(n)$ .

◀ ▶ ⏪ ⏩ 🔍

## subclasse Node

```
private class Node {
    private Key key;
    private Value val;
    private Node next;

    public Node(Key key, Value val,
        Node next) {
        this.key = key;
        this.val = val;
        this.next = next;
    }
}
```

## LinkedListST: construtor

Implementação em uma lista ligada com **nó cabeça**.

```
public class LinkedListST{
    private Node first; nó cabeça
    // número de itens na ST
    private int n = 0;

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

◀ ▶ ⏪ ⏩ 🔍

## LinkedListST: put()

```
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++;
}
```

◀ ▶ ⏪ ⏩ 🔍

## LinkedListST: delete()

```
public void delete(Key key) {
    Node p = rank(key);
    Node q = p.next;
    if (q == null) return null;
    if (!key.equals(q.key))
        return;
    p.next = q.next;
    n--;
}
```

Consumo de tempo:  $O(n)$ .

## 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 a `rank()`.

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

$$1 + 2 + \dots + n - 1 \approx n^2/2 = O(n^2).$$

Listas ligadas gastam  $O(n)$  espaço extra com referências de .

Em listas ligadas não temos busca binária...

## Exemplos

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

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

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

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

## Operação básica

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;
}
```

Consumo de tempo:  $O(n)$ .

## Frequência de acessos

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

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

$$Pr[0] + 2Pr[1] + 3Pr[2] + \dots + nPr[n-1].$$

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

$$Pr[0] \geq Pr[1] \geq Pr[2] \geq \dots \geq Pr[n-1].$$

## Mais exemplos

Se  $Pr[0] = Pr[1] = \dots = Pr[n-1] = 1/n$ , então

$$T(n) = (n + 1)/2.$$

Se

$Pr[0] = 1/2$ ,  $Pr[1] = 1/2^2$ , ...,  $Pr[n-2] = 1/2^{n-1}$ ,  $Pr[n-1] = 1/2^{n-1}$ , então

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

## Algumas distribuições de probabilidade

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

$$\Pr[0] = c, \Pr[1] = c/2, \dots, \Pr[n-1] = c/n,$$

onde  $c = H_n = 1 + 1/2 + 1/3 + \dots + 1/n$ , e portanto

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

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

Nesse caso

$$T(n) \approx 0.122n$$

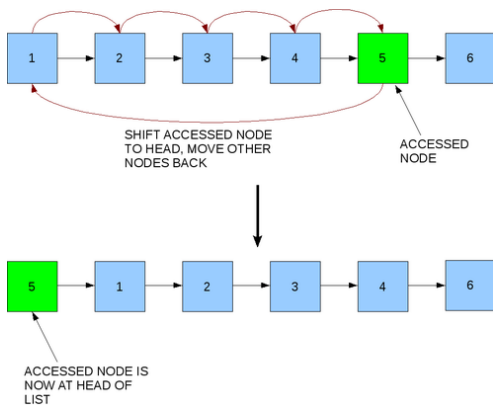
## Self-organizing lists

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*).

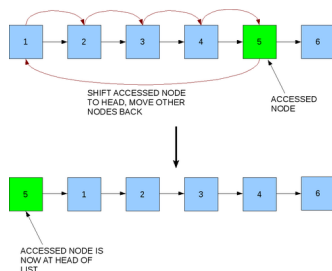


### Método *mova para frente*

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

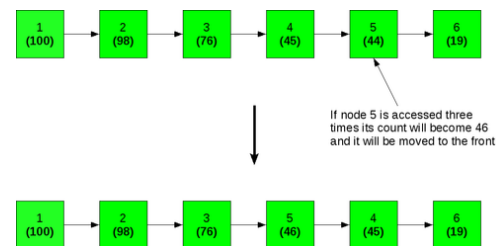
$$T(n) = \frac{1}{2} + \sum_{i,j} \frac{\Pr[i]\Pr[j]}{\Pr[i] + \Pr[j]}$$

### Método *mova para frente*



### Método *do contador*

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



### Vantagens:

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

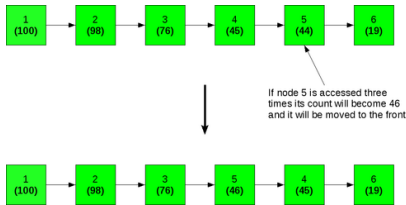
Fonte: [Wikipedia](#)

### Desvantagens:

- ▶ pode **sobrevalorizar** chaves não acessadas de maneira frequente;

Fonte: [Wikipedia](#)

## Método do contador



Fonte: Wikipedia

### Vantagens:

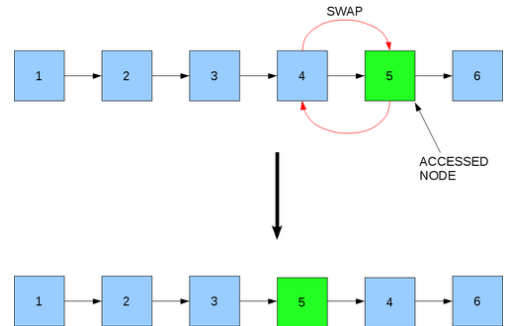
- ▶ reflete o padrão de acessos;

### Desvantagens:

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

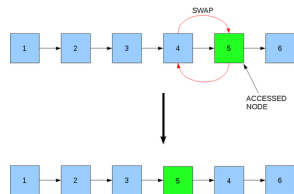
## Método da transposição

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



Fonte: Wikipedia

## Método da transposição



Fonte: Wikipedia

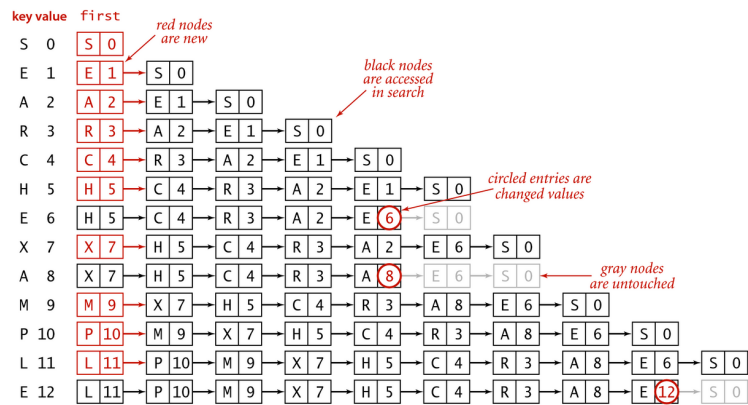
### Vantagens:

- ▶ 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 para key-val para o início.

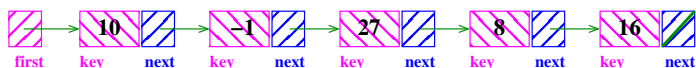
## Simulação de lista não ordenada



## ST em lista ligada com MTF

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

A implementação se apoia em `equals()` e não `compareTo()`



Cada nó *x* tem três campos:

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

## Classe SequentialMTFST: esqueleto

```
public class SequentialMTFST<Key, Value>{
    private Node first; // nó cabeça
    private int n;
    private class Node {...}
    public SequentialMTFST() {...}
    public Value get(Key key) {...}
    public void put(Key key, Value val){}
    public void delete(Key key) {...}
    public Key minM() {...}
    public Key max() {...}
    public Node rank(Key key) {...}
}
```





## 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	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**.

