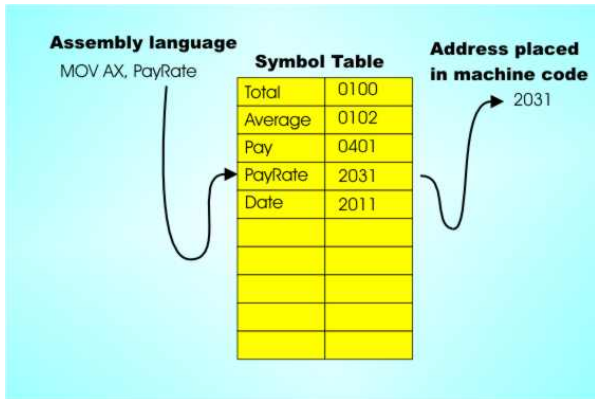


# AULA 7

# 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** (= *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 TS, e
- ▶ **get** (), que **busca** o valor associado a uma dada chave.

Convenções sobre TSs:

- ▶ não há chaves repetidas (as chaves são duas a duas distintas),
- ▶ **null** nunca é usado como chave,
- ▶ **null** nunca é usado como valor associado a uma chave.

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

---

## Consumo de tempo

Durante a execução de `get(key)` ou `put(key, val)`, uma chave da TS 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 busca 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.

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

*entries in red were inserted*  
*entries in black moved to the right*  
*entries in gray did not move*  
*circled entries are changed values*

Trace of ordered-array ST implementation for standard indexing client

## BinarySearchST

```
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];
    }
}
```

## Operação básica rank()

Retorna o **posto** (número de itens menores) de **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:**  $O(\lg n)$ .



## 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:  $O(\lg n)$ .

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;  
    }  
    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

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

## Conclusão

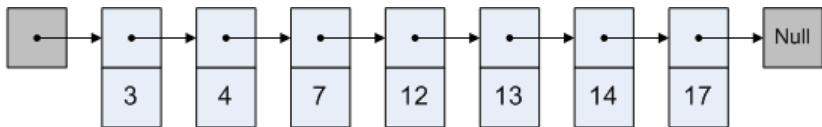
O consumo de tempo da função `get()` no pior caso é proporcional a  $\lg n$ .

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

## 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

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

## get()

```
public Value get(Key key) {  
    Node p = prev(key);  
    // key está na ST?  
    Node q = p.next;  
    if (q != null && q.key.equals(key))  
        return q.val;  
    return null;  
}
```

Consumo de tempo:  $O(n)$ .



## put()

```
public void put(Key key, Value val) {  
    if (val == null) {  
        delete(key); return;  
    }  
    Node p = prev(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++;  
}
```

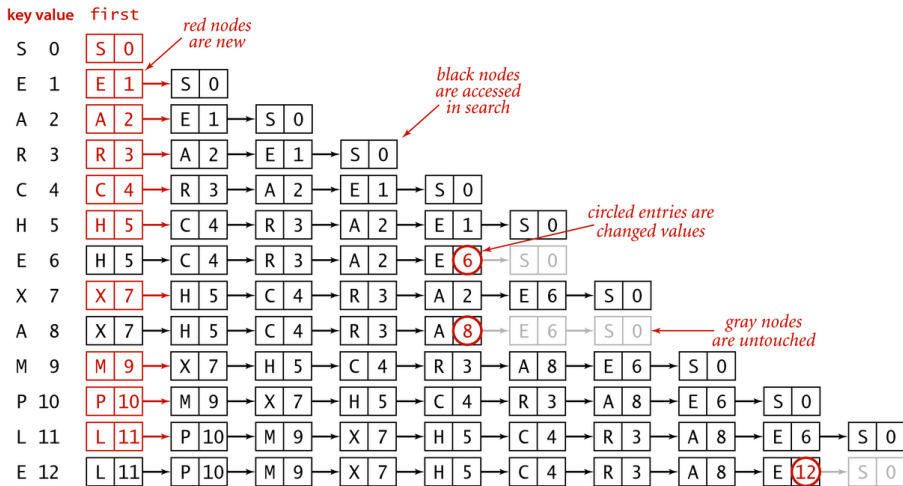
## Operação básica

Aqui usamos a ordenação (`compareTo()`)

```
private Node prev(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:  $O(n)$ .

# Simulação



Trace of linked-list ST implementation for standard indexing client

## 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 `get()`.

Portanto, o consumo de tempo para se criar uma lista com  $n$  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...

# Experimento

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.

<b>estrutura</b>	<b>tempo</b>
vetor	59.561
vetor <b>ordenado</b>	1.532
lista ligada	147.1
lista ligada <b>ordenada</b>	115.227

Tempos em segundos obtidos com **StopWatch**.