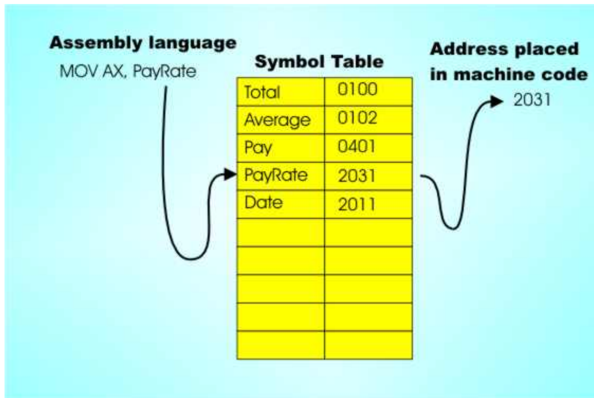
AULA 7

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

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);
}
```

Cliente: Index()

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

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 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[] 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

Classe BinarySearchST: esqueleto

```
public class BinarySearchST<Key extends
            Comparable<Key>, Value> {
   private Key[] keys;
   private Value[] vals;
   privateint 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) {...} ♡
}
```

BinarySearchST: construtor

```
public class BinarySearchST<Key extends
         Comparable<Key>, Value> {
  private Key[] keys;
  private Value[] vals;
  privateint n = 0;

  public BinarySearchST(cap) {
      keys = (Key[]) new Comparable[cap];
      vals= (Value[]) new Object[cap];
  }
```

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 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 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++;
}
```

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);
}
```

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

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

Conclusão

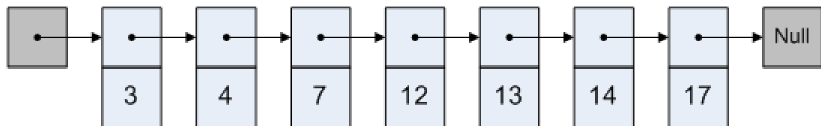
O consumo de tempo da função get() no pior caso é proporcional a $\lg n$.

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;
   }
}
```

Classe LinkedListST: esqueleto

```
public class LinkedListST<Key extends
            Comparable<Key>, Value> {
  private Node first; // nó cabeça
  private int n;
  private class Node {...}
  public LikedListST() {...}
  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: 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 LikedListST() {
       first = new Node(null, null, null);
   }
```

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: $O(n)$.

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: $O(n)$.

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: $O(n)$.

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

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

Listas ligadas gastam $\mathrm{O}(\mathtt{n})$ espaço extra com referências de .

Em listas ligadas não temos busca binária...

Frequência de acessos

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

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

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

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

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

Exemplos

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

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

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

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

Mais exemplos

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

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

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

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

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,

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

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

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

.

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

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

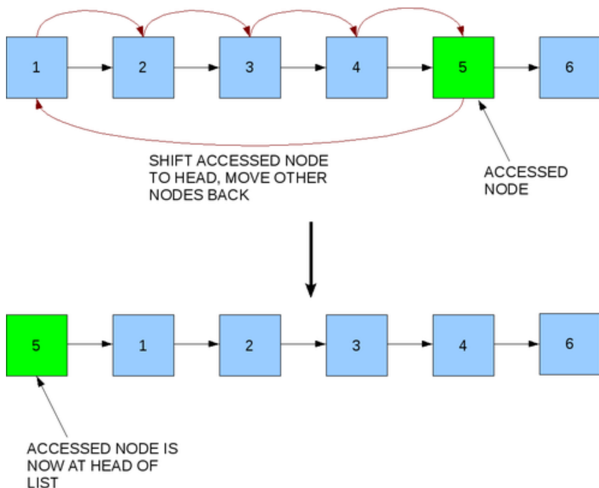
Self-organizing lists

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*).

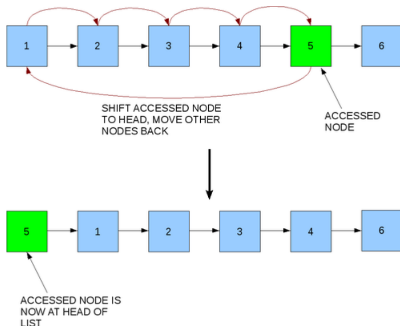


Método *mova para frente*

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

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

Método *mova para frente*



Fonte: Wikipedia

Vantagens:

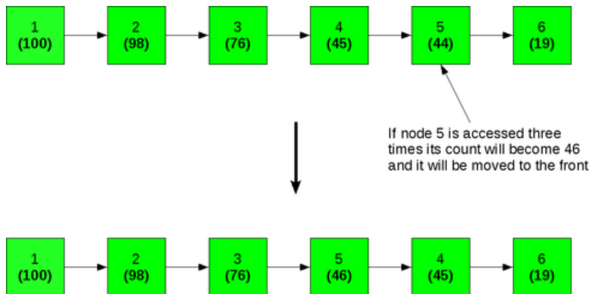
- fácil de se implementar;
- não utiliza espaço extra;
- se adapta rapidamente à sequência de acessos.

Desvantagens:

- pode sobrevalorizar chaves não acessadas de maneira frequente;

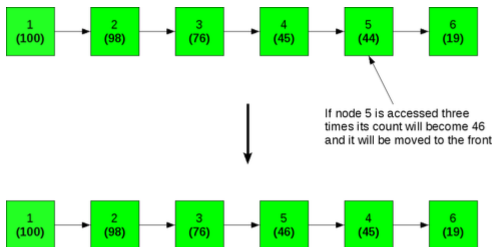
Método *do contador*

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



Fonte: Wikipedia

Método *do contador*



Fonte: Wikipedia

Vantagens:

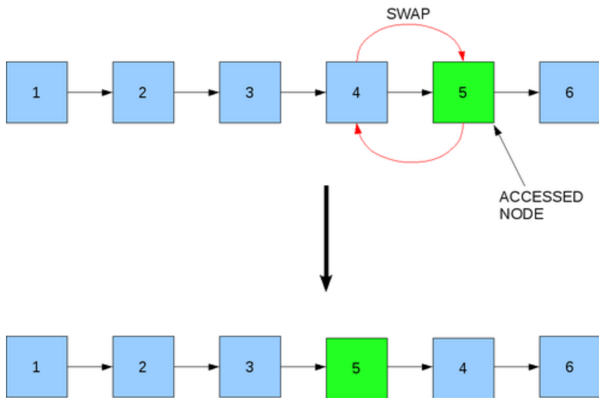
- reflete o padrão de acessos;

Desvantagens:

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

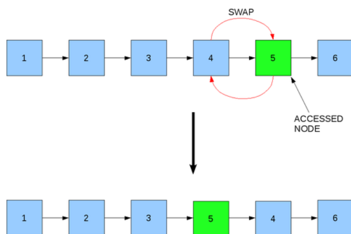
Método *da transposição*

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



Fonte: Wikipedia

Método *da transposição*



Fonte: Wikipedia

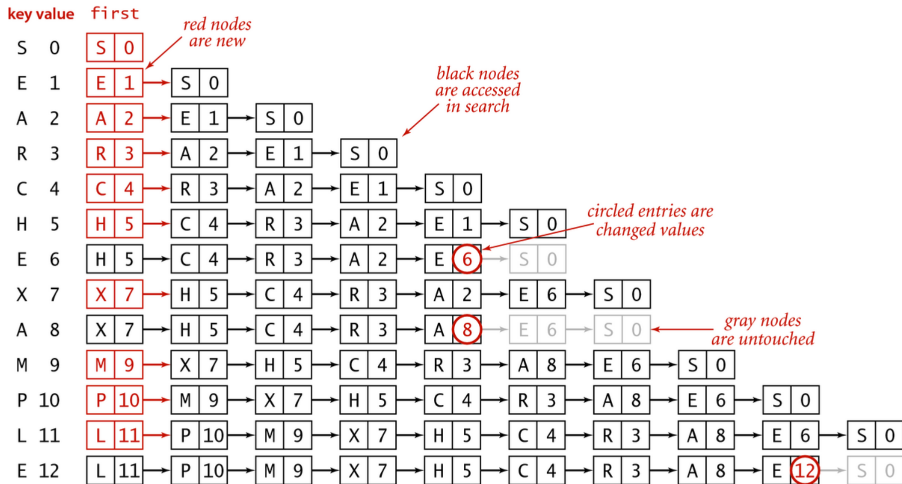
Vantagens:

- 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.

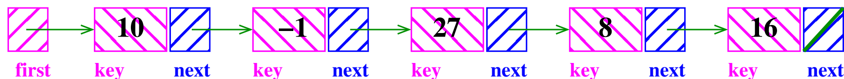
Simulação de lista não ordenada



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*.
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) {...} ♡
}
```

SequentialMTFST: construtor

Implementação em uma lista ligada com nó cabeça.

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

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

SequentialMTFST: get()

```
public Value get(Key key) {
   Node p = rank(key);
   // key está na ST?
   Node q = p.next;
   if (q == null) return ;
   // move to front
   p.next = q.next;
   q.next = first.next;
   first.next = q;
   return q.val;
}
```

Consumo de tempo: $O(n)$.

SequentialMTFST: put()

```
public void put(Key key, Value val) {
   if (val == null)
       { delete(key); return; }
   Node p = rank(key); Node q = p.next;
   if (q == null) { // key não está na ST
       first.next=
           new Node(key, val, first.next);
       n++; return; }
   q.val = val; // key está na ST
   p.next = q.next; // move to front
   q.next = first.next;
   first.next = q;
}
```

Consumo de tempo: $O(n)$.

SequentialMTFST: delete()

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

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

Operação básica

Não usamod ordenação apenas equals()

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

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.

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 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.