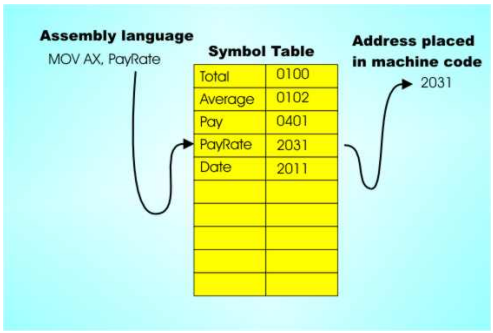
AULA 7

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	
	`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

Consumo de tempo

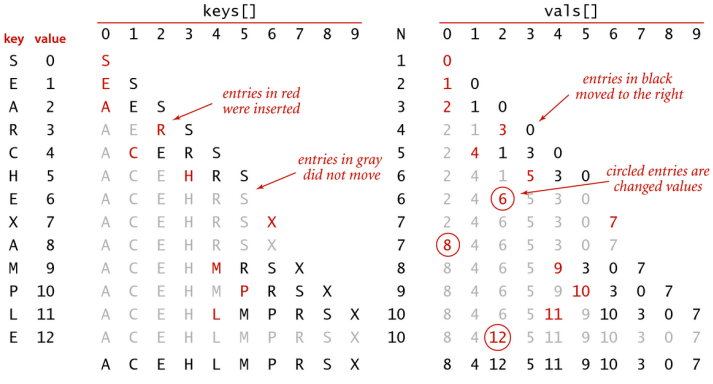
Durante a execução de `get(key)` ou `put(key,val)`, uma chave da TS é tocada quando comparada com `key`.
O consumo de tempo é proporcional ao **número de chaves tocadas**.

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.



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;
   privateint 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: $\mathrm{O}(\lg \mathrm{n})$.

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: $\mathrm{O}(\lg \mathrm{n})$.

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

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

Conclusão

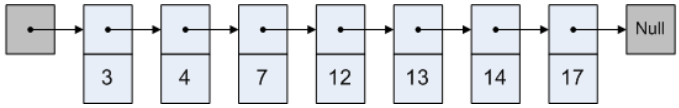
O consumo de tempo da função get() no pior caso é proporcional a $\lg \mathrm{n}$.

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

O consumo de tempo para criar uma STé no pior caso $\mathrm{O}(\mathrm{n}^2)$.

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++;
}
```

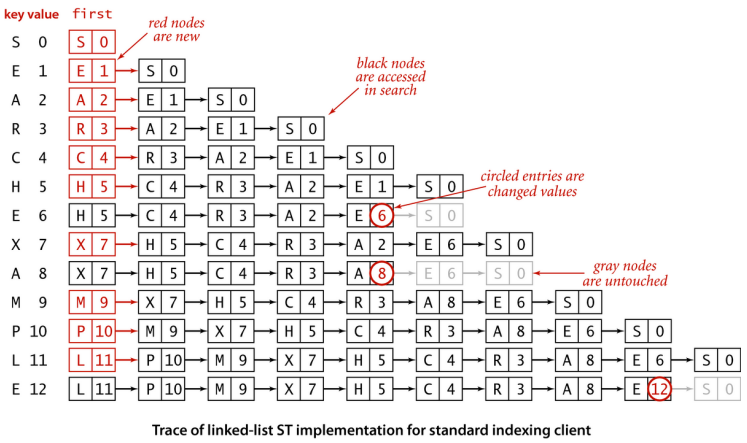
Operação básica

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 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. . .

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.

estrutura	tempo
vetor	59.561
vetor ordenado	1.532
lista ligada	147.1
lista ligada ordenada	115.227

Tempos em segundos obtidos com `StopWatch`.