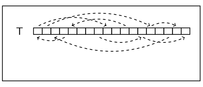
Endereçamento aberto

A ideia que nós vamos ver na aula de hoje consiste em armazenar os elementos dentro da própria tabela.

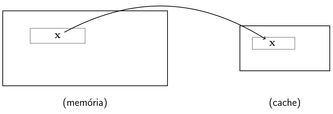
E resolver as colisões dentro da própria tabela também



Esse esquema é conhecido como *endereçamento aberto*, e o seu objetivo é tirar vantagem do modo de operação da memória cache.

Quer dizer, quando a gente faz um acesso à memória principal do computador, o sistema operacional não traz apenas a informação que a gente pediu.

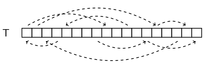
Ele também traz todo um monte de coisas que estão ali por perto, e armazena essas informações no cache



Isso faz sentido, porque os programas de computador costumam ficar perambulando em pequenas regiões de memória, antes de solicitar acesso a um local mais distante.

Daí, uma vez que a gente sabe que a memória cache funciona assim, a gente pode escrever programas que tiram vantagem disso.

No caso do esquema de enderaçamento aberto



- a medida que os elementos vão sendo acessados,
  porções inteiras da tabela vão parar dentro do cache
- daí, ao seguir uma sequência de encadeamentos no processo de busca,
  nós podemos encontrar os elementos no cache
- e quando isso acontece, tudo é muito mais rápido ...

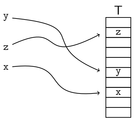
Legal.

Agora vamos ver como isso funciona.

Tudo começa como no mecanismo de encadeamento externo.

Quer dizer, quando chega um elemento, nós aplicamos a função de dispersão sobre ele para determinar a sua posição na tabela.

E se essa posição está vazia, o elemento é armazenado ali



Daí, mais cedo ou mais tarde, acaba acontecendo uma colisão.

E agora, nós precisamos encontrar uma outra posição para o novo elemento.

Como é que a gente faz isso?

Bom, a solução mais simples consiste em utilizar uma coleção de funções de dispersão

$$h_1(.), \quad h_2(.), \quad h_3(.), \quad h_4(.), \quad \ldots$$

E daí, no momento que ocorre a colisão, a gente pode ir aplicando as funções uma a uma, até encontrar uma posição vazia.

Na prática, a gente pode usar uma função de dispersão com 2 parâmetros, para representar a coleção de funções

$$H(i,x) \quad = \quad h_i(x)$$

Abaixo, nós temos o código da função de inserção na tabela de dispersão com endereçamento aberto

```
função Inserção-TDea (x, T)
 {
    i  <--  1
    Enquanto ( i <= n )
     {
        k  <--  H(i,x)
        Se ( T[k] está vazia )
         {
            T[k]  <--  x;    Retorna(k)
         }
        i++
     }
    Retorna (-1)           // a inserção falhou
 }
```

Certo.

O procedimento de busca é muito parecido.

Quer dizer, se alguém nos pede para localizar o elemento `x` na tabela, nós examinamos a seguinte sequência de posições uma a uma

$$h_1(x), \quad h_2(x), \quad h_3(x), \quad h_4(x), \quad \ldots$$

E se em algum momento nós encontramos uma posição vazia, nós sabemos que o `x` não está na tabela — (*porque?*)

A coisa fica assim

```
função Busca-TDea (x, T)
 {
    i  <--  1
    Enquanto ( i <= n )
     {
        k  <--  H(i,x)
        Se ( T[k] está vazia )    Retorna(-1)
        Se ( T[k] = x )           Retorna(k)
        i++
 }   }
```

Legal.

Mas aqui nós já temos um probleminha ...

Quer dizer, a busca pára quando ela encontra uma posição vazia — (*porque se o* `x` *estivesse na tabela, ele deveria estar ali — (porque?)*)

Mas isso significa que nós não podemos remover nenhum elemento da tabela.

Porque se essa posição estiver no caminho que leva até um outro elemento, esse outro elemento não será mais encontrado.

A solução para o problema consiste em marcar a posição como "`apagada`".

Quer dizer, agora as posições da tabela podem estar em 3 situações diferentes

- vazia
- ocupada
- apagada

E a ideia é que

- para o procedimento de inserção, as posições apagadas se comportam como posições vazias
  — (*i.e., o novo elemento pode ser colocado ali*)
- mas para o procedimento de busca, as posições apagadas se comportam como posições ocupadas
  — (*i.e., a busca continua quando ela chega ali*)

Utilizando essa ideia, a função de remoção fica assim

```
função  Remoção-TDea (x, T)
 {
    k  <--  Busca-TDea (x,T)

    Se ( k != -1 )   T[k]  <--  apagada
 }
```