Estruturas de Dados

aula 26: Endereçamento aberto

1 Introdução

O resultado que nós obtivemos na aula passada foi muito bom mesmo

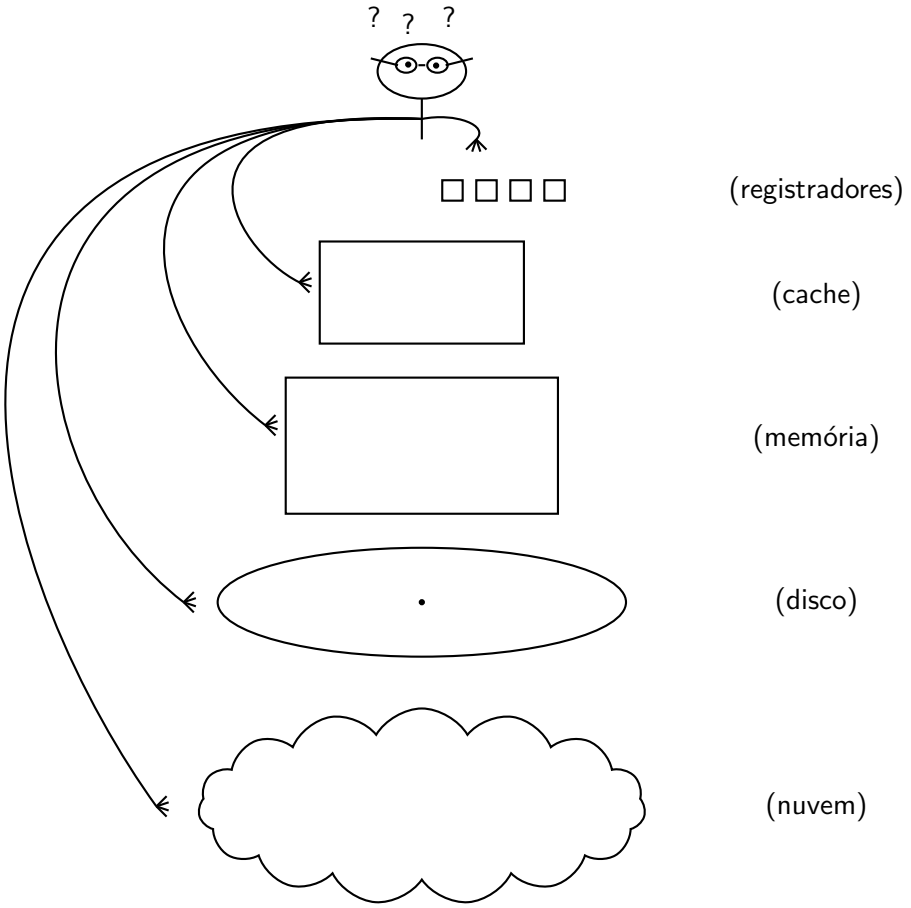
→ *A tabela de dispersão com encadeamento externo é uma estrutura de dados onde*

- *no pior caso, a busca leva tempo* $O(\ln n)$
- *mas quase todos os elementos (99,9%) podem ser encontrados em tempo* $O(1)$

Só que a gente ainda pode fazer melhor do que isso.

Quer dizer, a ideia de hoje é que um $O(1)$ nem sempre é igual a um $O(1)$.

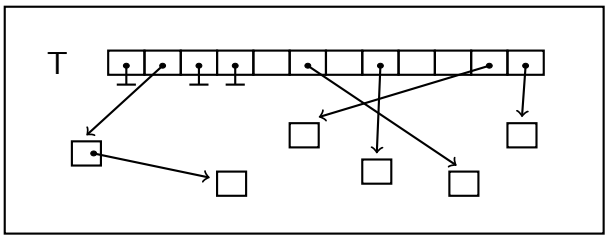
E a razão para isso tem a ver com a arquitetura dos nossos computadores



Quer dizer, uma operação $O(1)$ envolvendo dados que estão nos registradores ou no cache não é a mesma coisa que uma operação $O(1)$ envolvendo dados que estão na memória principal.

E quanto mais longe a gente vai, pior as coisas acabam ficando — (*apesar de tudo continuar levando tempo* $O(1)$)

O problema específico do encadeamento externo é que os registros que formam as listas encadeadas podem estar em qualquer lugar na memória



Daí que, no momento em que a gente percorre uma lista encadeada no procedimento de busca, a gente nunca espera encontrar o próximo elemento no cache.

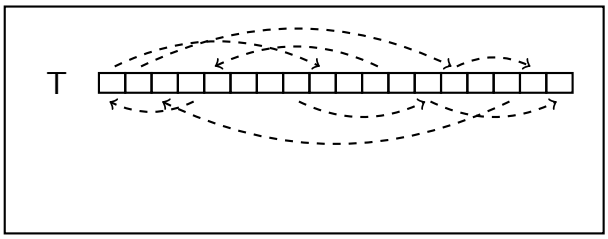
Cada elemento inspecionado na busca envolve um acesso à memória principal.

E isso leva tempo pra caramba — (*apesar de ainda ser* $O(1)$ ...)

2 Endereçamento aberto

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

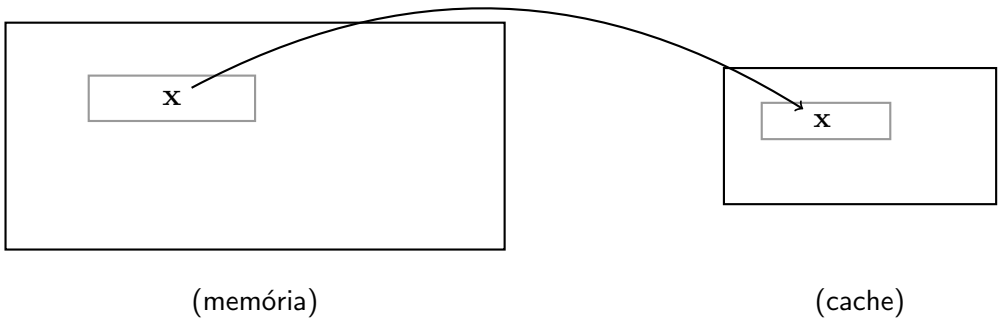
E resolver as colisões dentro da própria tabela também



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.

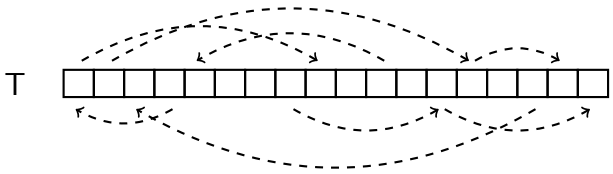
Ele também traz todo um monte de coisas que estão ali por perto, e armazena essas informações no cache



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,
  porcões inteiras da tabela vão parar dentro do cache
- dai, 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 ...

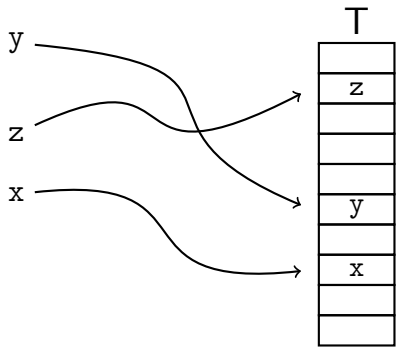
Legal.

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

$$\mathsf{h}_1(.), \quad \mathsf{h}_2(.), \quad \mathsf{h}_3(.), \quad \mathsf{h}_4(.), \quad \ldots$$

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

$$\mathsf{H}(\mathsf{i},\mathsf{x}) \quad = \quad \mathsf{h}_\mathsf{i}(\mathsf{x})$$

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

$$\mathsf{h}_1(\mathsf{x}), \quad \mathsf{h}_2(\mathsf{x}), \quad \mathsf{h}_3(\mathsf{x}), \quad \mathsf{h}_4(\mathsf{x}), \quad \ldots$$

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
 }
```

2.1 Funções de dispersão

Uma outra consideração prática são as funções de dispersão.

Quer dizer,

- *Aonde é que a gente vai encontrar todo um monte de funções de dispersão, para formar a coleção*

$$\mathsf{h}_1(.), \quad \mathsf{h}_2(.), \quad \mathsf{h}_3(.), \quad \mathsf{h}_4(.), \quad \ldots \quad ?$$

Bom, na prática a gente só precisa de uma função.

Porque as demais podem ser definidas a partir dela.

Por exemplo, imagine que o tamanho n da tabela T é um número primo.

Imagine que p é um outro número primo

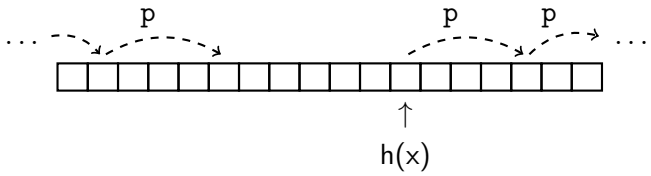
E imagine que h(.) é uma função de dispersão qualquer.

Daí, a gente define a nossa coleção de funções de dispersão assim

$$h_i(x) \quad = \quad h(x) \quad + \quad (i-1) * p \mod n$$

Quer dizer,

- a gente começa a busca pelo elemento x na posição h(x)
- e daí, se ele não estiver lá, a gente vai dando pulos de tamanho p até encontrá-lo
  — (*ou até encontrar uma posição vazia*)



E a boa notícia aqui é a seguinte propriedade dos números primos

→ *Dando pulos de tamanho* p*,*
a gente passa por todas as posições da tabela (sem repetir nenhuma)
antes de retornar ao lugar onde a gente começou

Legal, não é?

Em particular, isso significa que a inserção sempre tem sucesso, quando existe ao menos uma posição vazia na tabela.

3 Análise

Até aqui tudo bem.

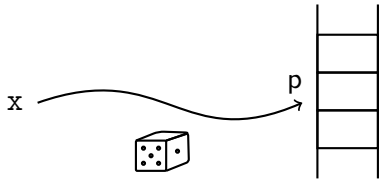
Mas será que esse negócio funciona bem na prática?

Bom, vamos ver.

Quer dizer, vamos fazer a análise.

E para fazer a análise, a gente faz a nossa suposição usual

→ *Imagine que as funções de dispersão* $h_i(.)$ *tem comportamento aleatório*



E daí, a gente faz uma observação bem fácil

→ *Quando a tabela está completamente cheia, uma busca sem sucesso sempre leva tempo* $O(n)$

Porque a busca só pára quando ela encontra uma posição vazia.

Mas uma tabela completamente vazia não tem posições vazias.

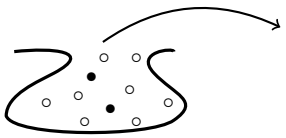
Certo.

Então para a coisa funcionar direito, a tabela não pode estar completamente cheia.

Daí que, nós vamos assumir que 10% das posições da tabela estão vazias.

E que as nossas funções de dispersão tem comportamento aleatório.

Nesse caso, aplicar uma função de dispersão a um elemento `x` é como selecionar uma bolinha de um saco onde 90% das bolinhas são brancas, e 10% das bolinhas são pretas



Quer dizer,

- a probabilidade de obter uma posição ocupada é igual a 9/10
- a probabilidade de obter uma posição vazia é igual a 1/10

E o tempo médio da busca é dado pela resposta da seguinte pergunta

- *Quantas bolinhas precisam ser selecionadas do saco (em média) até obter uma bolinha preta ?*

Bom, isso também é a mesma coisa que lançar uma moeda que dá `Cara` com probabilidade 1/10 até obter `Cara` pela primeira vez.

E a resposta nesse caso é

10 lançamentos (em média)

— (*porque?*)

Quer dizer, uma busca sem sucesso examina (em média) 10 posições na tabela.

E isso leva tempo $O(1)$.

Mas a coisa é ainda melhor para as buscas com sucesso.

Quer dizer, no momento em que a tabela ainda está vazia e os elementos começam a ser inseridos, a função de dispersão localiza uma posição vazia na primeira ou na segunda tentativa.

Daí, mais tarde, quando nós fizermos uma busca por esses elementos, eles também serão encontrados na primeira ou na segunda tentativa — (*mesmo que a tabela esteja cheia*)

— (*porque?*)

Quer dizer, o tempo que é gasto na busca por um elemento que está na tabela é o mesmo tempo que foi gasto no momento da sua inserção.

Mas a maioria dos elementos foi inserida quando a tabela ainda estava relativamente vazia.

Logo, o tempo de busca da maioria dos elementos da tabela é bem pequeno.

Certo.

Mas aqui nós não estamos levando em conta as remoções.

Quer dizer, quando as remoções acontecem, elas marcam as posições que ficaram vazias como apagadas.

E daí, se 90% das posições da tabela estão ocupadas, então os outros 10% vão ser divididos entre posições vazias e posições apagadas.

Mas, o desempenho da busca depende apenas do número de posições vazias.

Quer dizer, a medida que os elementos vão sendo inseridos e removidos, começam a surgir posições apagadas.

E nesse momento, o tempo de execução da busca começa a se degradar.

Daí, quando esse problema se torna muito grave, a solução é reconstruir a tabela.

Quer dizer,

- todos os elementos são removidos
- e depois eles são inseridos em uma tabela completamente vazia

Desa maneira, as posições apagadas são eliminadas.

E a busca volta a ter o seu desempenho ótimo.

3.1 Linear probing

(. . .)