Estruturas de Dados

aula 25: Tabelas de dispersão

1 Introdução

A disciplina de Estruturas de Dados só tem 2 ideias.

A primeira delas é a *busca binária*.

Quer dizer, tudo o que foi feito até aqui tinha o objetivo de fazer a busca binária funcionar sem pagar a penalidade $O(n)$ associada à inserção e remoção na lista ordenada.

Foi assim que nós passamos pelas

- listas encadeadas — (*inserção e remoção em tempo* $O(1)$*, desconsiderando o tempo da busca*)
- árvores binárias de busca — (*a lógica da busca binária sem o acesso indexado*)
- árvores balanceadas — (*controlando o pior caso*)

Certo.

Agora está na hora de ver a segunda ideia.

E para chegar lá, nós começamos perguntando

- *O que é mais rápido do que a busca binária ?*

Quer dizer, nem sempre a gente faz a busca binária.

Por exemplo, imagine que alguém diz pra você: `pinguim`.

Daí, será que você consegue perceber a sua cabeça pensando em

- `abacaxi, zumbi, linguiça, siriguela, orelha, quindim, panela, pinico, ...`

Quer dizer, será que você percebe a sua cabeça fazendo a busca binária?

Não parece que é isso o que acontece, não é?

Quer dizer, o cérebro já sabe onde o `pinguim` está armazenado na nossa memória.

E assim que a gente ouve o seu nome, a gente já pode começar a pensar sobre ele.

— *O seu quarto também é assim, não é?*

Quer dizer, apesar dele estar todo bagunçado, você sabe onde está cada coisa — (ou não?)

Como é que acontece isso?

2 Funções de dispersão

Hmm ...

Para responder essa pergunta, nós fazemos outra pergunta

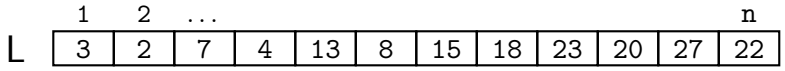
- *Como é que a gente acelera a busca binária ?*

Bom, a gente já viu essa ideia nas primeiras aulas da disciplina.

Quer dizer, a ideia é você ter uma lista onde

- os números pares estão nas posições pares
- os números ímpares estão nas posições ímpares
- e tanto os pares como os ímpares estão em ordem crescente
  — (*mas a lista inteira não precisa estar ordenada*)

Por exemplo,



Na prática, o que nós temos aqui são duas listas ordenadas com tamanho $n/2$.

Daí que, na prática nós fazemos a busca binária em uma lista de tamanho $n/2$.

E isso leva tempo

$$\log_2 (n/2) \quad = \quad \log_2 n \; - \; \log_2 2 \quad = \quad \log_2 n \; - \; 1$$

Mas ainda tem um probleminha aqui.

Quer dizer, para saber em qual das listas a busca vai ser feita, é preciso verificar se o número é par ou ímpar.

E isso leva tempo $O(1)$.

Daí que o tempo total do processo é

$$\log_2 n \; - \; 1 \; + \; O(1) \quad = \quad \log_2 n$$

E nós acabamos não ganhando nada ...

E agora?

Bom, agora nós podemos tentar levar a nossa esperteza mais adiante.

Quer dizer, ao invés de trabalhar apenas com números pares e ímpares, nós podemos trabalhar com o resto da divisão por 4.

A ideia é que

- os números que deixam resto 1 são armazenados nas posições 1, 5, 9, 13, 17, ...
- os números que deixam resto 2 são armazenados nas posições 2, 6, 10, 14, 18, ...
- os números que deixam resto 3 são armazenados nas posições 3, 7, 11, 15, 19, ...
- os números que deixam resto 0 são armazenados nas posições 4, 8, 12, 16, 20, ...

E em cada um desses grupos os números estão organizados em ordem crescente.

Na prática, nós temos 4 listas ordenadas com tamanho $n/4$.

Daí que a busca binária leva tempo

$$\log_2 n \quad - \quad 2$$

E levando em conta o tempo $O(1)$ para calcular o resto da divisão por 4, isso nos dá

$$\log_2 n \quad - \quad 2 \quad + \quad O(1) \qquad = \qquad \log_2 n \quad - \quad 1$$

Legal, a coisa deu certo!

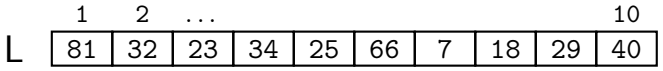
Mas não há qualquer razão para parar aqui.

Quer dizer, trabalhando com os restos da divisão por 8, o tempo da busca é reduzido para

$$\log_2 n \quad - \quad 2$$

E levando a coisa até o limite, nós podemos trabalhar com os restos da divisão por n.

Abaixo nós temos um exemplo com `n = 10`



Quer dizer, à primeira vista a lista está toda bagunçada.

Mas basta olhar para o último dígito dos número para saber a sua posição na lista.

E isso reduz o tempo da busca para $O(1)$.

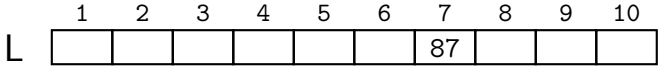
— (*é assim que você encontra as coisas no seu quarto?*)

Mas, não é difícil ver os problemas dessa solução.

Quer dizer, imagine que os 10 números que nós temos para armazenar são

`13, 83, 53, 123, 133, 673, 543, 393, 703, 463`

Ou então imagine que nós temos uma lista com apenas uma posição ocupada



E daí, alguém nos pede para armazenar o número `27`.

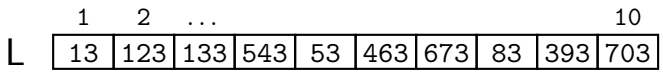
Nesse caso, apesar da lista estar quase toda vazia, nós vamos dizer que não há espaço para ele.

E agora?

Bom, agora é a hora para uma pequena esperteza.

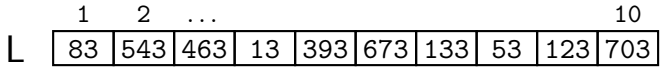
Quer dizer, e se ao invés de olhar para o último dígito do número, a gente olha para o penúltimo dígito?

Daí, os números acima seriam armazenados assim



Ou então, a gente também pode somar todos os dígitos do número, e depois calcular o resto da divisão por 10.

Fazendo isso, nós obtemos o seguinte resultado



Ou então, a gente pode pensar em alguma outra função qualquer.

Quer dizer, o truque é que a nossa função deve distribuir ou espalhar os números por todas as posições da lista.

Essas funções são conhecidas como *funções de dispersão* — (*ou hash functions, em inglês*)

Abaixo nós temos algumas funções de dispersão populares

- função quadrática: $h(x) = x^2 \bmod n$
- função multiplicativa:

  Suponha que A é um número entre 0 e 1 — (*de preferência um número irracional*)

  Daí o procedimento consiste em

  - multiplicar o valor de x por A
  - extrair a parte decimal do resultado
  - multiplicar a parte decimal por n, e arredondar para baixo

  Por exemplo, para $\mathtt{x} = 1235$, $\mathtt{A} = \sqrt{2}/2$, e $\mathtt{n} = 15$

  => $\mathtt{x} \cdot \mathtt{A} = 873,2768...$

  => $0,2768... \cdot 15 = 4$

E por aí vai ...

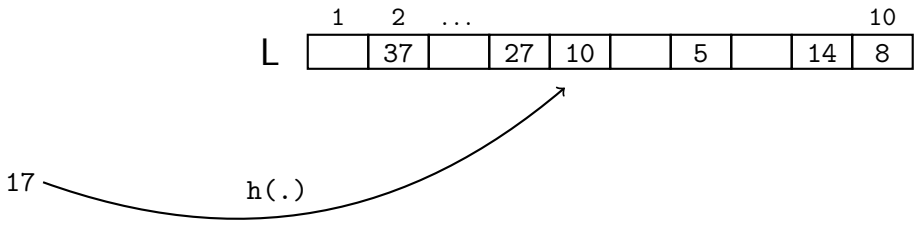
3 Tabelas de dispersão

Na prática, não existem funções de dispersão perfeitas — (*apesar de que esse é o tema da aula 28*)

E um procedimento comum consiste em utilizar uma lista maior do que a quantidade de elementos que serão armazenados — (*para facilitar o trabalho da função de dispersão*)

Mas isso também não resolve o problema, principalmente quando aconteçem inserções e remoções.

Quer dizer, mais cedo ou mais tarde, sempre vai aparecer um elemento que é mapeado para uma posição ocupada da lista

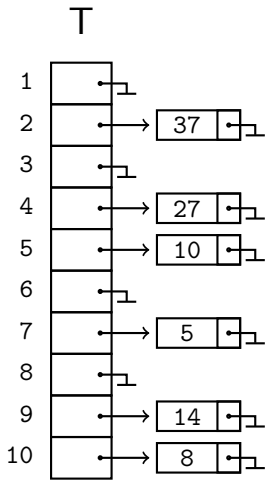


Quando isso acontece, nós dizemos que houve uma *colisão*.

E como isso sempre acaba acontecendo, mais cedo ou mais tarde, nós precisamos de algum procedimento de *resolução de colisões*.

Bom, a ideia mais simples de todas é armazenar os elementos que colidem todos na mesma posição.

Na prática, isso corresponde a ter uma lista (encadeada) de elementos associada a cada posição



Esse arranjo é conhecido como a *tabela de dispersão com encadeamento externo*.

Daí, quando um novo elemento `k` precisa ser inserido na tabela

- nós calculamos a sua posição `p` utilizando a função de dispersão `h`(.)
- e fazemos a sua inserção no início da lista associada à posição `p`

A busca é feita de maneira semelhante.

E a remoção também.

3.1 Algoritmos para a tabela com encadeamento externo

Abaixo nós temos o algoritmo que realiza a busca na tabela

```
função  Busca-TDee ( k, T )
 {
     p  <--  Hash(k)

     if ( T[p] = Nulo )   Retorna (NÃO)
```

```
    q  <--  T[p]

    while ( q != Nulo )
     {
        if ( q->val  =  k )    Retorna (SIM)

        q  <--  q->prox
     }
    Retorna (SIM)
}
```

Abaixo nós temos o algoritmo de inserção

```
função  Inserção-TDee ( k, T )
 {
      p  <--  Hash(k)

      q  <--  Novo ( RegInt )

   q.val  <--  k

  q.prox  <--  T[p]

    T[p]  <--  q
 }
```

E abaixo nós temos a função de remoção

```
função  Remoção-TDee ( k, T )
 {
    p  <--  Hash(k)

    q  <--  T[p];      qant  <--  Nulo

    while ( q != Nulo )
     {
        if ( q->val  =  k )    break

        q  <--  q->prox
     }

    if ( q = Nulo )  Retorna       // k não está na tabela

    if ( qant = Nulo )
     {
        T[p]  <--  q->prox
     }
    else
     {
        qant->prox  <--  q->prox
     }
 }
```

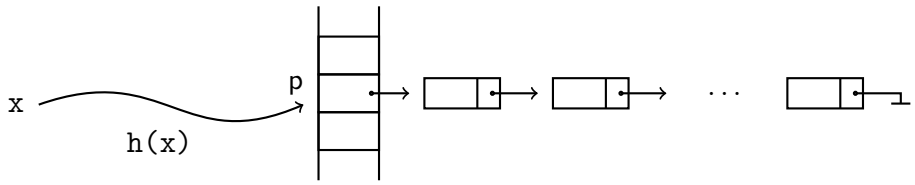
3.2 Análise

É fácil ver que o procedimento de *inserção* executa em tempo $O(1)$

— (*ele se comporta como a inserção em uma lista desordenada*)

A *busca* e a *remoção* na tabela de dispersão também se comportam como na lista desordenada.

Portanto, o seu tempo de execução é proporcional ao tamanho da lista associada à posição `p` da tabela



Considerando o cenário de pior caso, a função de dispersão `h`(.) mapeia todos os elementos da tabela para a posição `p`

E nesse caso, tanto a busca como a remoção levam tempo $O(n)$.

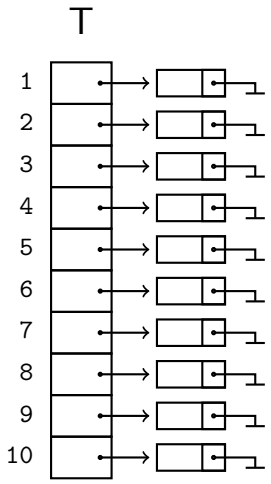
Mas dessa vez, a suposição de pior caso não é razoável.

Quer dizer,

→ *Nenhuma função de dispersão digna desse nome mapearia todos os elementos para a mesma posição da tabela*

— *(uma função assim deveria se chamar função de concentração ...)*

Outra possibilidade é considerar a função de dispersão perfeita, que mapeia todos os elementos para posições diferentes.

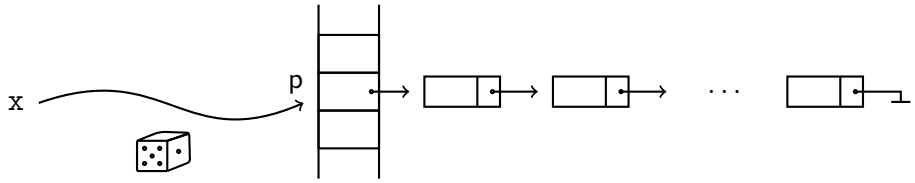


Mas essa suposição tampouco é razoável, na prática.

Na prática, o mais razoável é imaginar que uma boa função de dispersão se comporta como uma *função aleatória*.

Mas o que é isso?

Bom, a ideia é que a função aleatória lança um dado com n faces para determinar a posição de cada elemento na tabela



— (*i.e., todas as posições tem a mesma probabilidade de serem obtidas no lançamento do dado*)

Sim, mas o que significa isso?

Bom, isso significa que ainda vão haver colisões.

E por causa disso, vão haver posições da tabela com mais de um elemento.

E também vão haver posições da tabela sem elemento nenhum — (*assumindo que nós temos uma tabela de tamanho n que armazena n elementos*)

Algumas posições da tabela vão conter apenas 1 elemento.

Outras posições vão conter 2 elementos.

E por aí vai ...

Certo.

Mas, e daí?

Bom, daí que assumindo que a função de dispersão tem comportamento aleatório, nós podemos fazer várias observações interessantes — (*que hoje vão ficar sem prova ...*)

1. 63,2% dos elementos estão na primeira posição das suas listas,
   e são encontrados imediatamente pela busca — (*i.e., 1 comparação*)
2. quase 90% dos elementos estão na primeira ou segunda posição das suas listas,
   e são encontrados com até 2 comparações
3. quase 98% dos elementos estão na primeira, segunda ou terceira posição das suas listas,
   e são encontrados com até 3 comparações
4. 99,9% dos elementos estão na primeira, segunda, terceira ou quarta posição das suas listas,
   e são encontrados com até 4 comparações
5. a maior lista de todas contém (aproximadamente) $\ln n$ elementos

   daí que, no pior caso, a busca leva tempo $O(\ln n)$

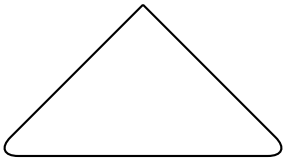
   — (*assumindo que a função de dispersão tem comportamento aleatório*)

Quer dizer, a busca quase sempre executa em tempo $O(1)$.

E nas poucas vezes em que ela apresenta o pior caso, a busca leva tempo $O(\ln n)$.

Aqui vale a pena fazer o contraste com a árvore binária de busca.

Quer dizer, mesmo na situação ideal em que nós temos uma árvore completa



- a busca quase sempre executa em tempo $O(\log n)$
  — (*porque a vasta maioria dos elementos está na parte de baixo da árvore*)
- e nas poucas vezes em que o elemento é encontrado próximo da raiz, a busca executa em tempo $O(1)$