Estruturas de Dados

aula 21: Multilistas e listas esparsas

1 Introdução

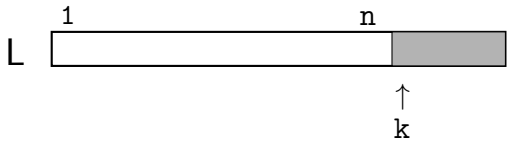
- *O que é melhor: manter a lista ordenada ou manter a lista desordenada ?*

Bom, isso depende ...

Quer dizer, a busca na lista desordenada leva tempo $O(n)$ — (*porque no pior caso a gente tem que examinar todos os elementos*)

Mas a gente viu que a busca na lista ordenada pode ser feita em tempo $O(\log n)$.

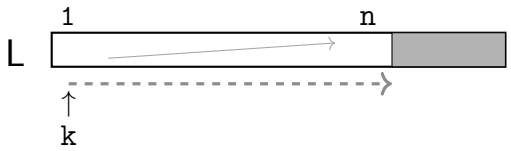
Por outro lado, a inserção na lista desordenada leva tempo $O(1)$: basta colocar o novo elemento na primeira posição vazia — (*porque a ordem não importa ...*)



Mas, a inserção na lista ordenada pode levar tempo $O(n)$.

Quer dizer, no pior caso, a inserção é feita na primeira posição.

E daí, todos os elemento vão ser deslocados para a direita



Finalmente, a remoção na lista desordenada também leva tempo $O(1)$: basta mover o último elemento da lista para a posição do elemento que está sendo removido.

Mas para fazer a remoção é preciso fazer a busca primeiro.

E daí, a coisa toda leva tempo $O(n)$.

Na lista ordenada, a busca leva tempo $O(\log n)$.

Mas a remoção pode levar tempo $O(n)$ — (*quando nós estamos removendo o primeiro elemento*)

Abaixo nós temos uma tabela que resume a situação

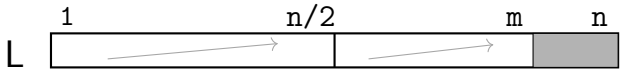
	busca	inserção	remoção
lista desordenada	$O(n)$	$O(1)$	$O(n)$
lista ordenada	$O(\log n)$	$O(n)$	$O(n)$

2 Multilistas

Mas, algumas espertezas nos permitem melhorar a situação no caso ordenado.

Quer dizer, nós vamos usar a mesma esperteza da aula passada.

A ideia é dividir a lista na metade



onde tanto a primeira parte como a segunta parte vão estar ordenadas.

Esse esquema duplica o tempo da busca.

Porque para procurar um número `k`, nós precisamos fazer a busca na primeira parte e fazer a busca na segunda parte — (*no pior caso*)

Daí que, o tempo total é

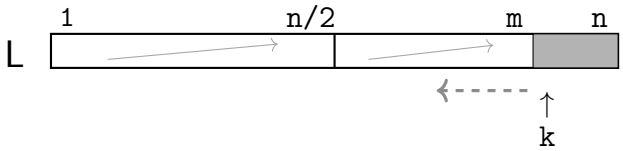
$$\begin{aligned} \log_2(n/2) \quad + \quad \log_2(n/2) \quad &= \quad 2 \cdot \log_2(n) \quad - \quad 2 \\ &= \quad 2 \cdot \log_2(n) \qquad \text{(aprox.)} \end{aligned}$$

Por outro lado, o tempo da inserção é reduzido à metade.

Quer dizer, a inserção é sempre feita na segunda parte.

E ela é realizada em duas etapas

1. colocar o novo elemento na primeira posição vazia
2. e depois deslocá-lo até a sua posição correta



É bem fácil ver que

- a primeira etapa leva tempo $O(1)$
- e a segunda etapa envolve no máximo $n/2$ trocas de posição

Daí que, a coisa toda leva tempo

$$n/2 \qquad \text{(aprox.)}$$

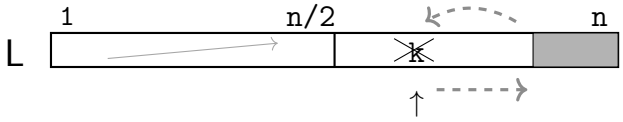
Legal.

O tempo da remoção também é reduzido à metade.

Quer dizer, a remoção também é feita em duas etapas

1. primeiro nós fazemos a busca
2. e depois nós removemos o elemento da lista — (*se ele é encontrado*)

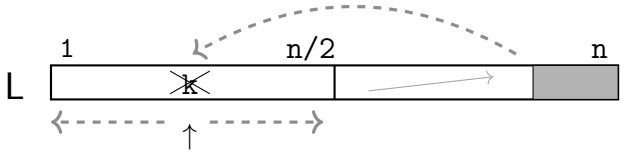
Quando o elemento é encontrado na segunda parte, nós colocamos o último elemento na sua posição, e depois fazemos o deslocamento



O tempo do deslocamento domina todos os outros — (*inclusive o tempo da busca*)

Mas o deslocamento envolve no máximo $n/2$ trocas de posição.

Quando o elemento é encontrado na primeira parte, nós colocamos o último elemento (da segunda parte) na sua posição, e depois fazemos o deslocamento



— (*note que o deslocamento pode acontecer para os dois lados ...*)

Mais uma vez, o tempo do deslocamento domina todos os outros.

Mas o deslocamento envolve no máximo $n/2$ trocas de posição.

Daí que, em ambos os casos a remoção leva tempo

$$n/2 \qquad \text{(aprox.)}$$

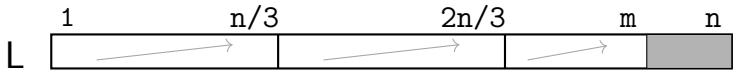
— (*no pior caso*)

2.1 Repetindo a esperteza

Certo.

Mas agora que a gente entendeu, a gente pode levar a coisa mais longe.

Quer dizer, a gente pode pensar em dividir a lista em 3 partes



Daí,

- a busca vai levar tempo $3 \log n$ (aprox.)
- e a inserção e a remoção vão levar tempo $n/3$ (aprox.)

Quer dizer, o tempo da busca aumentou outra vez.

Mas agora, os tempos da inserção e da remoção foram reduzidos a 1/3 do tempo original.

Já dá para advinhar que isso vai continuar acontecendo quando a gente dividir a lista em mais partes.

E daí, a gente pode perguntar

- *Qual é o número de partes ótimo ?*

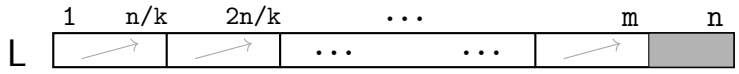
Só que dessa vez a pergunta não tem resposta ...

Quer dizer, tudo depende do contexto.

Se a gente tem uma situação onde acontecem muitas buscas e quase nenhuma inserção e remoção, então a gente deve manter o tempo da busca o menor possível.

Por outro lado, se ocorrem muitas inserções e remoções, então a gente vai querer manter os tempos da inserção e remoção pequenos.

Para entender melhor o que está acontecendo, imagine que a lista é dividida em k partes



Daí, no pior caso, a busca precisa examinar todas as partes.

E isso leva tempo

$$k \cdot \log(n/k) \qquad \text{(aprox.)}$$

Por outro lado, a inserção sempre ocorre na última parte.

E como essa parte tem tamanho no máximo n/k, a coisa leva tempo

$$n/k \qquad \text{(aprox.)}$$

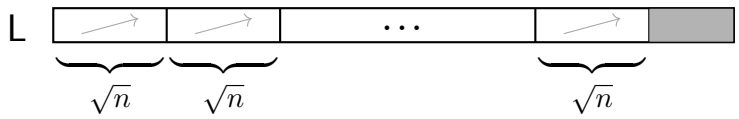
Finalmente, a remoção envolve uma busca seguida de um deslocamento.

E isso leva tempo

$$k \cdot \log(n/k) \quad + \quad n/k \qquad \text{(aprox.)}$$

Legal.

A estrutura de dados *multilista* corresponde à situação em que a gente divide a lista em blocos de tamanho $\sqrt{n}$



E nesse caso, os tempos de execução das operações de busca, inserção e remoção ficam assim

	busca	inserção	remoção
multilista	$O(\sqrt{n} \cdot \log n)$	$O(\sqrt{n})$	$O(\sqrt{n} \cdot \log n)$

2.2 Algoritmos para multilistas

Como já vimos, a busca na multilista consiste em fazer a busca binária em cada uma das suas partes

A coisa fica assim

```
função  busca-Multista ( k, L[1..n], m)
 {
     Para  j <-- 0  Até  piso ( m / raiz(n) )
      {
           i  <--  j * raiz(n) + 1                // início da (j+1)-ésima sublista

           f  <--  Min { m, (j+1) * raiz(n) }     // fim da (j+1)-ésima sublista

        resp  <--  busca-Binária ( k, L[i,f] )

        Se ( resp = SIM )  Retorna (SIM)
      }
     Retorna (NÃO)
 }
```

Como já vimos, a inserção sempre ocorre na última parte da lista.

E a coisa fica assim

```
função  inserção-Multista ( k, L[1..n], m)
 {
    i  <--  piso ( m / raiz(n) ) * raiz(n)  +  1    // início da última sublista

    m ++;   L[m]  <--  k                            // inserção no fim

    j  <--  m
    Enquanto ( j > i )                              // deslocamento
     {
        Se ( L[j] < L[j-1] )
         {
               aux  <--  L[j]
             L[j]  <--  L[j-1]
          L[j-1]  <--  aux
         }
        Senão  Break
 }   }
```

Finalmente, a remoção envolve primeiro uma busca pelo elemento a ser removido.

Daí, nós movemos o último elemento da última sublista para a sua posição, e depois fazemos o deslocamento.

A coisa fica assim

```
função  remoção-Multista ( k, L[1..n], m)
 {
    pos  <--  busca-Multilista ( k, L[1..n], m )

    Se ( pos = 0 )  Retorna                         // o elemento não está lá
```

```
    L[pos]  <--  L[m];    m --

    Se ( L[m] < k )                                        // deslocamento para trás
     {
        i  <--  piso ( pos / raiz(n) ) * raiz(n) + 1       // início da sublista de k

        j  <--  pos

        Enquanto ( j > i )
         {
            Se ( L[j] < L[j-1] )
             {
                   aux  <--  L[j]
                 L[j]  <--  L[j-1]
              L[j-1]  <--  aux
             }
            Senão  Break
     }   }

    Se ( L[m] > k )                                        // deslocamento para frente
     {
        f  <--  piso ( pos / raiz(n) + 1 ) * raiz(n)       // fim da sublista de k

        j  <--  pos

        Enquanto ( j < f )
         {
            Se ( L[j] < L[j+1] )
             {
                   aux  <--  L[j]
                 L[j]  <--  L[j+1]
              L[j+1]  <--  aux
             }
            Senão  Break
     }   }
}
```

3 Listas esparsas

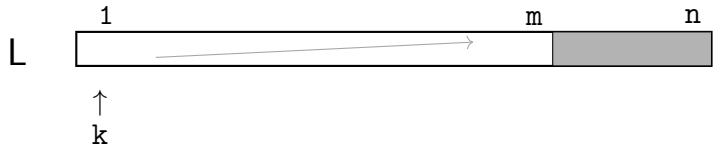
Legal.

Mas essa não é a única maneira de fazer as coisas.

Quer dizer, aqui nós vamos usar uma esperteza diferente.

E para chegar lá, nós vamos examinar o problema da inserção na lista ordenada outra vez.

Quer dizer, o pior caso acontece quando nós queremos inserir um elemento na primeira posição da lista

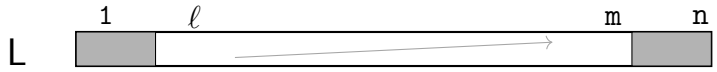


Mas porque esse é o pior caso?

Ora, porque as posições vazias estão todas no final da lista.

Quer dizer, se as posições estivessem no início, então o pior caso seria inserir um elemento na última posição da lisa.

Por outro lado, se as posições vazias estão no início e no fim, então nenhum dos dois casos é mais o pior caso



Quer dizer, agora o pior caso é inserir um elemento no meio da lista.

E nesse caso, a inserção (e a remoção) envolvem o deslocamento de no máximo

$$\frac{n}{2} \quad \text{elementos}$$

Porque os deslocamentos sempre acontecem na direção da posição vazia mais próxima.

Pronto.

Essa esperteza permite reduzir o tempo das operações de inserção e da remoção à metade
— (*sem afetar o tempo da busca*)

E agora, para continuar reduzindo esse tempo ainda mais, nós podemos colocar posições vazias no meio da lista

Nesse caso, a inserção (ou remoção) de um elemento envolve o deslocamento de no máximo

$$\frac{n}{4} \quad \text{elementos}$$

Certo.

Para generalizar essa ideia, nós vamos espalhar as posições vazias por toda a lista.

E para implementar essa ideia, nós vamos imaginar que cada posição da lista tem um campo `L[j].vazia`, que indica se aquela posição está vazia ou não.

Quer dizer, agora não existe mais um lugar pré-determinado para as posições vazias.

Elas podem estar em qualquer lugar.

E a primeira boa notícia, é que nessa situação a operação de remoção leva tempo $O(1)$
— (*sem levar em conta o tempo da busca*)

Porque para remover um elemento, basta marcar a sua posição como vazia

```
L[j].vazia  =  Sim
```

Por outro lado, a operação de inserção ainda pode envolver o deslocamento de alguns elementos.

E para simplificar as coisas, nós vamos assumir que o deslocamente sempre é feito para frente
— (*e que sempre existe uma posição vazia mais à frente*[1])

[1] Na prática, é preciso procurar por posições vazias para frente e para trás.

Daí, a operação de inserção é realizada em 4 etapas

1. busca binária para localizar a posição de inserção
2. localizar a posição vazia mais próxima
3. deslocar os elementos até lá, para liberar a posição de inserção
4. inserir o elemento na lista

E a coisa fica assim

```
func  Inserção-LE ( k, L )
 {
     pos  <--  busca_Binária-LE ( k, L )

     j = pos                                    // localiza posição vazia mais próxima
     Enquanto ( L[j].vazia !=  Sim )
         j++
     L[j].vazia  <--  Não

     Enquanto ( j > pos )                       // deslocamento dos elementos
       {
          L[j-1]  =  L[j];    j--
       }

     L[pos]  <--  k                             // insere o novo elemento
 }
```

Certo.

Agora nós podemos perguntar: *Quanto tempo leva isso?*

Bom, a resposta é

- o tempo da busca binária
- mais o tempo da procura pela posição mais próxima
- mais o tempo do deslocamento

O tempo da busca binária nós vamos analisar mais adiante.

Agora, os outros dois componentes do tempo dependem da distribuição das posições vazias na lista.

Quer dizer, se as posições vazias estão todas no fim da lista e nós estamos fazendo a inserção no início, então a coisa leva tempo $O(n)$.

Mas a ideia chave das listas esparsas consiste justamente em manter as posições vazias bem distribuídas na lista.

Daí que, nós vamos fazer as seguintes suposições para a análise

i. Imagine que existem 90% de posições ocupadas, e 10% de posições vazias na lista

ii. Imagine que as posições vazias estão bem distribuídas ao longo da lista

Quer dizer, em média nós encontramos uma posição vazia a cada 10 posições.

Nessas condições, a chamada à função `Vazia_+_próxima()` leva tempo $O(1)$.

E o deslocamento também leva tempo $O(1)$.

Daí que, o tempo da inserção é basicamente o tempo da busca binária.

3.1 Busca binária na lista esparsa

(. . .)