Listas esparsas

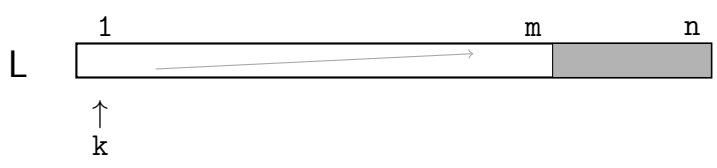
Legal.

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

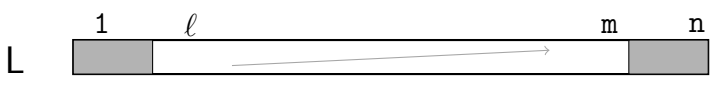


Mas porque esse é o pior caso?

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*[a])

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.

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