Fundamentos de programação

aula 06: Manipulação de dados

1 Introdução

Na primeira parte do curso, nós escrevemos programas que inspecionavam vários tipos de dados: números, listas, palavras, etc.

Agora o foco vai ser a *manipulação*: transformar as coisas umas nas outras.

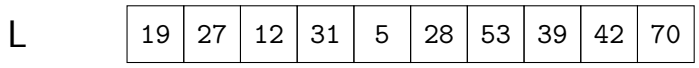
Na aula de hoje, nós vamos realizar manipulações sobre listas: inserir elementos, remover elementos, e procurar elementos também.

E nós vamos fazer essas coisas em dois contextos diferentes:

- listas ordenadas
- listas onde a ordem dos elementos não é importante

2 Manipulação de listas

Considere uma lista com `N` números, onde os elementos aparecem em uma ordem qualquer

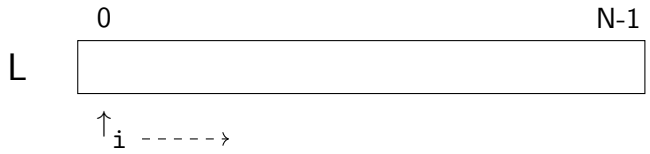


Imagine que nós queremos saber se o número `x` se encontra na lista ou não.

Nesse ponto, isso já é uma tarefa bem fácil.

Quer dizer, basta percorrer a lista da esquerda para a direita e, para cada elemento, verificar se ele é igual a `x`.

E isso pode ser feito com o auxílio de uma variável `i`



O programa abaixo implementa essa ideia

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70};
int x = 53;
int i;
```

```
int main()
 {
    for (i=0; i<N; i++)
     {
        if ( L[i] == x) { printf("O número x foi encontrado na posição %d", i);
                          return(0);
     }                  }
    printf ("O número x não está na lista");
 }
```

Esse programa tem duas pequenas novidades:

- A primeira dela é a linha

  ```
  #define   N   10
  ```

  Nós podemos pensar nessa instrução como a declaração de uma variável `N` que não vai mudar de valor durante toda a execução do programa[1]

  E essa variável pode ser utilizada na declaração de outras variáveis

  ```
  int L[N];
  ```

  Esse tipo de coisa é muito conveniente porque, se mais tarde nós resolvemos mudar o tamanho da lista, só vai ser preciso modificar um programa em um único lugar.

- A segunda novidade é o comando

  ```
  return(0);
  ```

  Esse comando é semelhante ao `break`, só que ao invés de interromper o comando `for`, ele interrompe o programa inteiro.

  Isso faz sentido nesse caso, porque depois que nós encontramos o número `x` na lista, não há realmente mais nada a fazer.

Certo.

Mas, esse programa ainda tem algo de estranho.

Quer dizer, o comando

```
for (i=0; i<N; i++)
```

pode ser interpretado como

- Faça o índice `i` percorrer a lista inteira

E daí, quando a gente encontra o `x`, a gente força uma interrupção.

Pensando um pouquinho sobre o assunto, seria mais natural escrever algo como

- Enquanto o `x` ainda não foi encontrado, continue procurando

[1] Na prática, o compilador simplesmente substitui toda ocorrência do termo `N` por 10, antes de iniciar a compilação do programa.

Essa é exatamente a lógica do comando `while`.

Utilizando esse comando, o miolo do nosso programa ficaria assim:

```
 . . .
i = 0;
while ( L[i] != x )   i++;
 . . .
```

A primeira observação aqui é que o comando `while` não atualiza o índice `i` automaticamente.

Quer dizer, antes de começar nós precisamos dar um valor inicial para `i` por meio de uma instrução separada: `i = 0`.

E nós também precisamos atualizar o índice `i` explicitamente a cada passo, por meio de uma instrução separada: `i++`.

Mas agora, o nosso programa tem uma interpretação bem natural

```
 . . .
i = 0;                    //  comece no início da lista
while ( L[i] != x )       //  e enquanto o seu elemento é diferente de x
   i++;                   //  continue andando para frente
 . . .
```

Não é legal?

Sim é legal, mas o programa ainda tem um pequeno problema.

Quer dizer, *e se o número* `x` *não estiver na lista?*

Bom, daí o índice `i` vai ultrapassar o limite `N` da lista, e o nosso programa vai dar um erro de execução.

Mas a solução para isso é bem fácil

```
while ( i < N  &&  L[i] != x )
```

onde && é o operador lógico E na linguagem C.

Certo.

Mas agora, quando o comando `while` termina, a gente não sabe se o número `x` foi encontrado ou não.

E nós vamos precisar de um teste adicional no final do programa para verificar isso.

A coisa fica assim

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70};
int x = 53;
```

```
int i;

int main()
 {
    i = 0;
    while ( i < N  &&  L[i] != x )   i++;

    if (i == N)    printf ("O número x não está na lista");
    else           printf("O número x foi encontrado na posição %d", i);
 }
```

2.1 Procurando o x em uma lista ordenada

Agora suponha que os números aparecem em ordem crescente na lista L.

Por exemplo

L	5	12	19	27	28	31	39	42	53	70

Então, se nós sabemos disso, nós podemos fazer as coisas de maneira mais eficiente.

Quer dizer, em certos casos, nós vamos poder concluir que o número x não está na lista sem ter que olhar a lista inteira.

Por exemplo, se nós estamos procurando pelo número 30 na lista acima, então quando nós chegamos no 31 nós já podemos parar.

O trecho de programa abaixo implementa essa ideia

```
i = 0;
while ( i < N  &&  L[i] < x )   i++;
```

Mas agora, o comando `while` pode terminar de 3 maneiras diferentes

- a lista chegou ao fim
- nós encontramos um elemento maior que x
- o número x foi encontrado

E daí, nós vamos precisar de um teste um pouquinho mais complicado no final do programa

```
if ( i == N  ||  L[i] > x )
       printf ("O número x não está na lista");
else   printf("O número x foi encontrado na posição %d", i);
```

onde o termo || é o operador lógico OU da linguagem C.

2.2 Uma esperteza bem legal (Opcional)

Mas, nós podemos fazer as coisas de maneira ainda mais eficiente.

A observação chave é a seguinte

- Quando a lista está ordenada

  o início não é o melhor lugar para começar a busca

Quer dizer, nós podemos começar pelo elemento do meio: `L[N/2]`.

E daí, nós temos 3 possibilidades outra vez

- ou nós damos a sorte de que `L[N/2]` = `x` — (o que é pouco provável)
- ou `L[N/2] > x`, e daí o `x` só pode estar do lado esquerdo
- ou `L[N/2] < x`, e daí o `x` só pode estar do lado direito

E nos dois últimos casos nós precisamos continuar procurando.

O programa abaixo implementa essa ideia

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 5, 12, 19, 27, 28, 31, 39, 42, 53, 70};
int x = 53;
int i;

int main()
 {
    i = N/2;

    if ( L[i] == x )  {  printf("O número x foi encontrado na posição %d", i);
                         return(0);
                      }

    if ( L[i] > x )
     {
        while ( i >= 0  &&  L[i] != x )   i--;

        if (i < 0)    printf ("O número x não está na lista");
        else          printf("O número x foi encontrado na posição %d", i);
     }

    if ( L[i] < x )
     {
        while ( i < N  &&  L[i] != x )   i++;

        if (i == N)    printf ("O número x não está na lista");
        else           printf("O número x foi encontrado na posição %d", i);
 }   }
```

Não é legal?

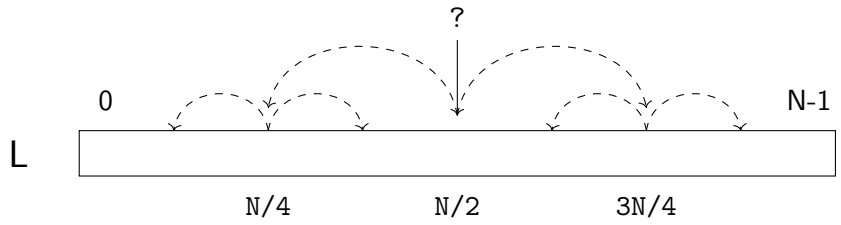
Sim é legal, mas vai ficar ainda mais legal.

Imagine que `L[N/2] > x` e que agora você tem que procurar o `x` no lado esquerdo.

Então, a observação chave diz que o melhor lugar para começar não é o elemento `L[`$\frac{N}{2}$`-1]`, mas é o elemento do meio do lado esquerdo.

Faz sentido usar a mesma ideia outra vez, não é?

Daí que, a busca evolui sempre pulando para o meio do pedaço onde o número `x` ainda pode estar

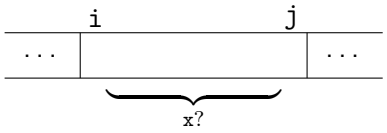


Mas, como é que a gente escreve um programa que faz uma coisa dessas?

Bom, na verdade é mais fácil do que parece.

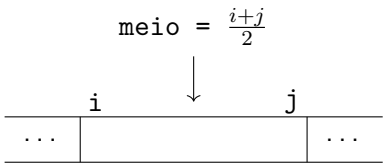
Quer dizer,

- Nós sempre estamos procurando o número `x` em um certo pedaço da lista



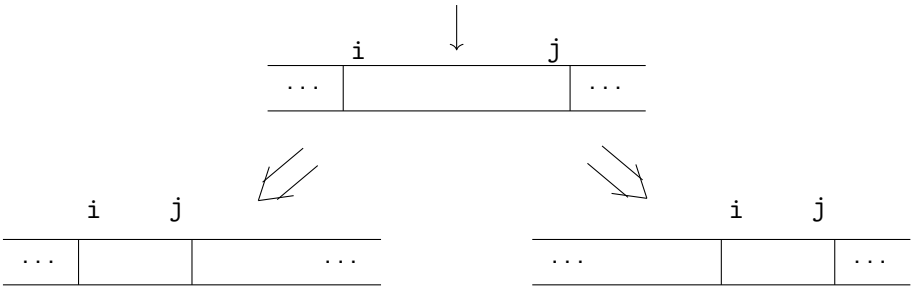
  (no início, esse pedaço é a lista inteira)

- E para procurar o `x` ali, a gente sempre olha na posição do meio



- Daí, se a gente acha o `x`, então acabou.

  E se a gente não acha, existem basicamente 2 casos



  e cada um desses casos leva a gente outra vez para a situação inicial.

Quer dizer, a gente fica fazendo isso enquanto ainda existe um pedaço da lista para procurar.

Daí que, a gente pode implementar essa ideia usando o comando `while`.

O programa abaixo faz exatamente isso.

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 5, 12, 19, 27, 28, 31, 39, 42, 53, 70};
int x = 53;
int i,j,meio;

int main()
 {
    i = 0;   j = N-1;

    while ( i <= j )
     {
        meio = (i+j)/2;

        if ( L[meio] == x )
           printf("O número x foi encontrado na posição %d", i);

        if ( L[meio] > x )  j = meio - 1;

        if ( L[meio] < x )  i = meio + 1;
     }
    printf ("O número x não está na lista");
 }
```

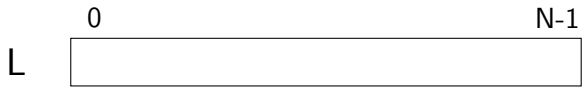
Pronto, é só isso!

Esse método de busca é tão eficiente, que ele é capaz de encontrar o `x` em uma lista de tamanho 1.000.000 (ou descobrir que ele não está lá) fazendo aproximadamente 20 comparações.

Isso é realmente legal!

3 Inserindo um novo elemento na lista

Considere outra vez a situação da lista desordenada

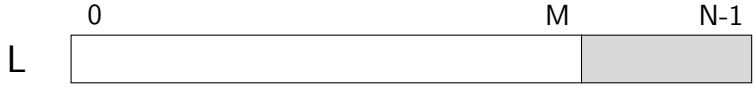


E imagine que nós queremos inserir o número `58` nessa lista.

Bom, como a lista está desordenada, nós podemos inserir o `58` em qualquer lugar.

Mas para isso é precisso que haja espaço.

Quer dizer, agora nós vamos trabalhar com a ideia de que a lista não ocupa todo o espaço reservado para ela na memória



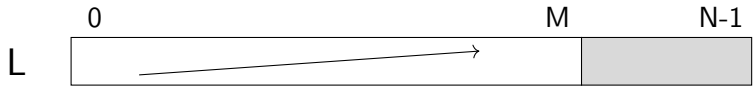
Daí que, se $\texttt{M} < \texttt{N} - 1$ ainda espaço para colocar mais um elemento.

O programa abaixo implementa essa ideia.

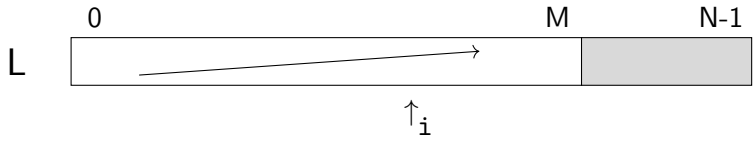
```
#include <stdlib.h>
#include <stdio.h>

#define  N  20

int L[N] = { 5, 12, 19, 27, 28, 31, 39, 42, 53, 70}, M=10;
int x = 53;

int main()
 {
    if ( M == N-1 )  {  printf("Não há espaço para um novo elemento");
                        return(0);
                     }

    M++;           // aumenta o tamanho da lista

    L[M] = x;      // coloca o novo elemento na última posição
 }
```

3.1 Inserção na lista ordenada

Agora imagine que a lista está ordenada

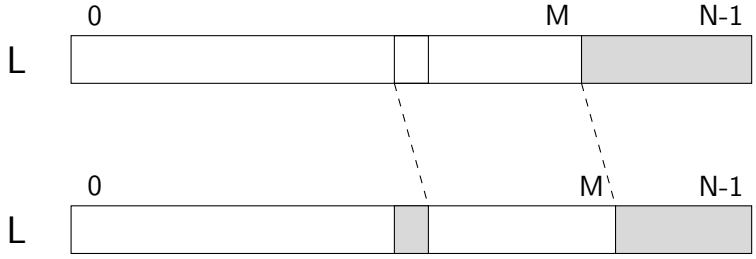


Nesse caso, existe um lugar exato para colocar o `x` (de modo a preservar a ordem)



Mas, já tem alguém ali ...

A solução, então, é deslocar todo mundo a partir dali uma posição para a direita



E daí, basta colocar o `x` no seu lugar.

A ideia é implementar esse procedimento em 3 etapas

1. encontrar o lugar do `x`
2. deslocar os elementos
3. inserir o `x` na lista

A primeira etapa é só uma variante da busca, e nós já sabemos como fazer isso.

E a etapa 3 é bem simples.

Vejamos então como a etapa 2 pode ser realizada.

A ideia é começar pelo final.

Quer dizer, começando pela posição `M`, nós vamos puxando os elementos 1 posição para a direita, até chegar na posição `i`.

A coisa fica assim

```
j = M;
while ( j <= i )  {  L[j+1] = L[j];   j--;  }
M++;
```

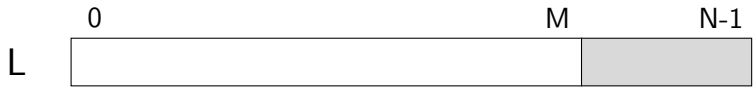
É só isso.

O programa abaixo implementa a inserção de um elemento na lista ordenada, utilizando a versão mais simples do procedimento de busca.

```
#include <stdlib.h>
#include <stdio.h>

#define  N  20

int L[N] = { 5, 12, 19, 27, 28, 31, 39, 42, 53, 70}, M=10;
int x = 53;
int i,j;

int main()
 {
    // ETAPA 1:  busca
    i = 0;
    while ( i <= M  &&  L[i] < x )  i++;

    if (L[i] == x )  {  printf("O número x já está na lista");
                        return(0);
                     }

    // ETAPA 2:  deslocamento (p/ a direita)
    j = M;
    while ( j <= i )  {  L[j+1] = L[j];   j--;  }
    M++;

    // ETAPA 3:  inserção
    L[i] = L[M];
 }
```

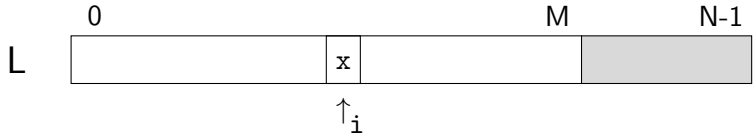
4 Remoção de um elemento da lista

Considere mais uma vez o caso da lista desordenada

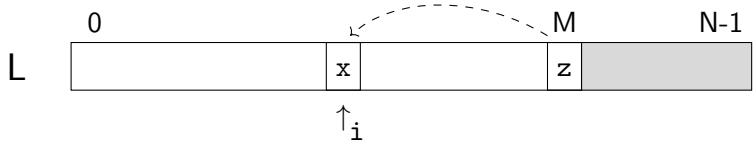


E imagine que nós queremos remover o número `x` da lista.

O primeiro passo, claro, é descobrir onde o `x` está (se é que ele está lá)



E daí, como a lista está desordenada, basta colocar o último elemento no seu lugar para que a remoção seja feita

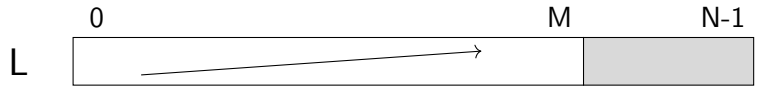


O programa abaixo implementa essa ideia

```
#include <stdlib.h>
#include <stdio.h>

#define  N  20

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70}, M=10;
int x = 53;
int i,j;

int main()
 {
    // ETAPA 1:  busca
    i = 0;
    while ( i < N  &&  L[i] != x )  i++;
    if ( L[i] == N )  {  printf("O número x não está na lista");
                         return(0);
                      }

    // ETAPA 2:  remoção
    L[i] = x;
 }
```

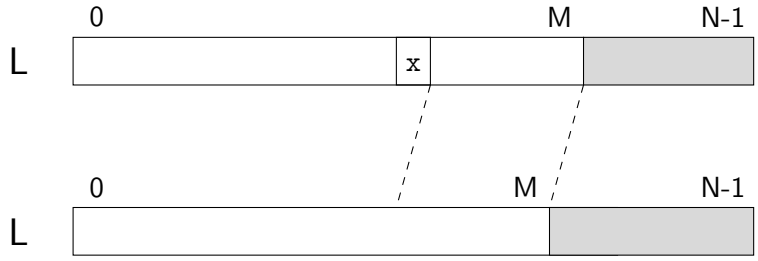
4.1 Remoção na lista ordenada

Agora imagine que a lista está ordenada



Como no caso desordenado, o primeiro passo é localizar o número `x`.

E daí, para fazer a remoção, basta deslocar todos os elementos à direita de `x` uma posição para a esquerda



A coisa fica assim

```
#include <stdlib.h>
#include <stdio.h>

#define  N  20

int L[N] = { 5, 12, 19, 27, 28, 31, 39, 42, 53, 70}, M=10;
int x = 53;
int i,j;

int main()
 {
    // ETAPA 1:  busca
    i = 0;
    while ( i <= M  &&  L[i] < x )  i++;

    if ( i > M  ||  L[i] != x )
     {
        printf("O número x não está na lista");
        return(0);
     }

    // ETAPA 2:  deslocamento (p/ a esquerda)
    j = i;
    while ( j < M )  {  L[j+1] = L[j];   j++;  }
    M--;
 }
```