Estruturas de Dados – Listas Lineares

Vamos estudar representações para listas em geral, e para listas com critérios de acesso restritos: pilhas, filas e deques.

Listas:

Definição: seqüência finita de itens de dados (elementos). Na lista, os elementos podem estar ordenados ou não.

Cada elemento tem uma posição: há um 1º, 2º, ... (pode haver também uma ordem relativa aos valores dos elementos, por exemplo, podem estar ordenados alfabeticamente, ou por ordem crescente).

Em implementações simples de listas todos os elementos possuem o mesmo tipo de dado, ou seja, são listas homogêneas.
Ex:

- lista de inteiros
- lista de caracteres

Não há no entanto qualquer objeção conceitual para listas heterogêneas (cujos elementos podem ser de tipos diferentes).

Alguns termos que iremos utilizar:

- Lista vazia: não contém elementos.
- comprimento (*length*): número de elementos na lista.
- Cabeça ou início (*head*): início da lista
- fim (*tail*): final da lista
- Listas ordenadas: elementos posicionados em ordem (ascendente/descendente de valor).
- Listas não ordenadas: não há relações entre valores e posições.

- Exemplo de uma notação para representar listas:

L = (e_0, e_1, e_2, e_3, ... , e_{n-1})
n = comprimento da lista
para todo elemento **e** da lista e i>0, $\mathbf{e_i}$ precede $\mathbf{e_{i+1}}$ e segue $\mathbf{e_{i-1}}$

Lista vazia: ( )

Operações básicas que uma implementação deve suportar:

- Listas devem crescer e diminuir: é possível inserir e retirar elementos
- Devemos ter acesso a qualquer elemento para efetuar operações: ler e modificar.
- Deve ser possível criar e destruir (reinicializar) listas.

Um T.A.D. para listas

Definindo o tipo lista:

```
TAD Lista {
        elemento;
Operações:      novaLista()
                reinicializa()
                inserir(E: elemento, posição)
                retirar(E: posição)
                consultar(E: elemento; S: posição)
        ...
}
```

Essa definição e uma descrição do comportamento das listas é tudo o que um programador precisa para usar uma implementação em particular de listas, ou para construir uma nova implementação.

elemento - pode ser de qualquer tipo

```
Ex:     lst Lista;
        lst = novaLista();
        lst.inserir(10, 1);
```

Outras operações sobre a lista:

```
lst.primeiro(S: elemento)
lst.comprimento(S: comprimento)
lst.estaVazia()
lst.elemento(E: posição S: elemento)
...
```

Esse T.A.D. representa uma dentre muitas interpretações para listas: não contém portanto detalhes de implementação.

Duas abordagens básicas de implementação

- através de vetores (ou arranjos, ou sequencial)
- listas encadeadas (com referências ou apontadores)

Implementação através de vetores:

Características:

- Explora a sequencialidade da memória (contiguidade física)
- Os elementos são armazenados em um vetor (array)
- Normalmente com tamanho fixo (o tamanho precisa ser conhecido quando a lista é criada).

Criação da lista: A inicialização da lista ("lst" é uma variável cujo tipo é "Lista") poderia ser simplesmente:

lst.novaLista()

Poderíamos também considerar a situação em que o número máximo de elementos da lista é indicado na inicialização:

lst.novaLista (tamanho)

onde, tamanho = número máximo de elementos na lista.

Em uma implementação precisamos manter:

- tamanho máximo
- número atual de elementos (ou posição do último elemento da lista)

Ex: para a lista, A = (55,13,42,37)

Tamanho máximo = 7

Número atual de elementos = 4

A =

1	*2*	*3*	*4*	*5*	*6*	*7*
55	13	42	37			

Alteração da lista: Nesse tipo de implementação o **acesso aleatório a elementos** é simples, assim como os processos de inserir e remover elementos no final da lista. No entanto, se quisermos **inserir em uma posição qualquer** i, se há n elementos (início = 1) teremos que mover (n-i)+1 elementos para criar um espaço para inserção.

Da mesma forma para **remover elementos** será necessário mover outros elementos para preencher o espaço vazio, já que estamos mantendo os elementos da lista em bloco na parte esquerda do vetor. Remover um elemento da posição i implica em deslocar n - i elementos:

*-> é preciso fazer **relocação** de elementos*

Implementações (exemplo):

Tipo Lista:

```
pública      novaLista ( )
             inserir (elemento, posição)
             remover(posição) : elemento
             comprimento() : inteiro

privada      inteiro vetor [1 .. 100]
             inteiro tamanhoMaximo
             inteiro númeroDeElementos

     novaLista( )
          tamanhoMaximo = 100
          númeroDeElementos = 0
     fim

     inserir(elemento, posição)
          Se (numeroDeElementos < tamanhoMaximo) entao
               i = numeroDeElementos + 1
               Enquanto i >= posição faça
                    vetor [i] = vetor [i-1]
                    i =  i-1
               Fim Enquanto
               vetor[posição] = elemento
               numeroDeElementos = numeroDeElementos+1
          Fim Se
     Fim

Fim da definição
```

Exercício 1: Implemente os seguintes procedimentos para a lista acima.

remover(posição) : elemento
encontre(elemento) : posição

Implementação através de listas encadeadas:

Características:

- Usamos referências (ou apontadores) para conectar nodos que compõem a lista.

- Diz-se que a lista é "dinâmica": alocamos espaço em memória conforme necessário.

A lista é formada por uma série de objetos normalmente denominados ***nodos***. Cada nodo irá conter um elemento da lista. Precisamos determinar a estrutura dos nodos que compõem a implementação da lista.

Uma implementação inicial: lista com encadeamento simples.

```
Tipo Lista:
publica        ... o mesmo ...
privada
               tipo Nodo
                     elemento: elem
                     próximo: referência para Nodo
               fim

               númeroDeElementos
               início : referência para Nodo
Fim Lista
```

Exemplo de uma classe em Java e de uma struct em C que definem a estrutura e criam um nodo de uma lista de encadeamento simples:

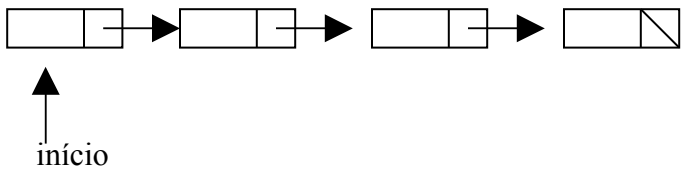
```
/* Java */
class Nodo {
      int elemento;
      Nodo prox;

      Nodo (int valor) {
            elemento = valor;
            prox = null;
}
...
Nodo n = new Nodo (5);
```

```
/* C */
struct nodo {
        int elemento;
        struct nodo *prox;
} *n;

typedef struct nodo Nodo;
…
N= (*Nodo) malloc (sizeof (Nodo));
n->elemento = 5;
n->prox = NULL;
```

Uma representação gráfica para listas com encadeamento simples.



É preciso manter uma variável na classe Lista que armazene o nodo inicial da lista, no exemplo anterior ela é representada pela variável **início**. Em Java, a variável início (assim como uma variavel **fim**, que se fosse o caso armazenaria o nodo final da

lista) seria declarada como do tipo Nodo, e em C ambas seriam do tipo Nodo (ou struct nodo):

```
/* em Java */
class Lista {
      Nodo inicio, fim;

      Lista () {
            inicio = fim = null;
      }
      Lista (Nodo n) {
            inicio = n;
            fim = n;
      }
}
```

```
/* em C */
Nodo *inicio, *fim, *n;
…
n= (Nodo) malloc (sizeof (Nodo));
n->elemento = 5;
n->prox = NULL;

/* se n for o nodo inicial da lista: */
if (inicio!=NULL)
   inicio = fim = n;
...
```

Listas implementadas com encadeamento evitam o problema de limitar o número de elementos que podem ser inseridos na lista (na verdade o limite é a capacidade de memória do computador). No entanto, quando consideramos as operações de inserção e remoção de elementos em uma lista com encadeamento simples, percebemos que, para efetuar a operação, é preciso percorrer todos os elementos que estão antes da posição desejada. Esse processo pode tornar a implementação dessas operações ineficiente.

Quando estamos procurando definir listas com uma interface útil para seus usuários e buscando uma implementação eficiente devemos considerar possíveis alternativas. Abaixo fazemos uma análise de problemas e possíveis soluções que podem melhorar a definição e a implementação das listas: *em que posição da lista realizar a inserção e a remoção, e listas com definição do elemento corrente*.

1. Onde realizar a inserção?

Pode-se considerar diferentes possibilidades, a serem implementadas ou não em função das necessidades da aplicação em questão:

- No início da lista
- No meio da lista
- No final da lista

Se a lista não é ordenada, pode-se optar por inserir sempre no início ou no final da mesma. Caso a lista seja ordenada, é preciso considerar as três possibilidades para atender a todas as necessidades de inserção.

É necessário também considerar em separado a inserção do primeiro elemento da lista quando a lista está vazia.

Exemplo de inserção em uma lista de encadeamneto simples, na qual o nodo é sempre inserido no final da lista (exemplo em Java):

```
class ListaSimples {
      Nodo inicio, fim;

      ListaSimples ( ) {
```

```
            inicio = fim = null;
}

void insere_fim (Nodo novonodo) {
      novonodo.prox = null;
      if (inicio == null)
            inicio = novonodo;
      if (fim != null)
            fim.prox = novonodo;
      fim = novonodo;
}
}
```

```
/* em C */
Nodo *inicio, *fim, *n;
void insere_fim (int v){
    n= (*Nodo) malloc (sizeof (Nodo));
    n->elemento = v;
    n->prox = NULL;
    if (inicio==NULL)
       inicio = n;
    if (fim!=NULL)
       fim->prox = n;
   fim = n;
}
```

2. Onde realizar a remoção?

Pode-se considerar diferentes possibilidades, a serem implementadas ou não em função das necessidades da aplicação em questão:

- Do início da lista (inserção do primeiro e único; inserção na primeira posição)
- Do meio da lista
- Do final da lista

Para remoção de elementos específicos (por posição ou por um valor determinado) é preciso considerar as três possibilidades para atender a todas as necessidades de inserção.

É necessário também considerar em separado a remoção do último elemento da lista, deixando-a vazia.

3. Listas com definição de elemento corrente:

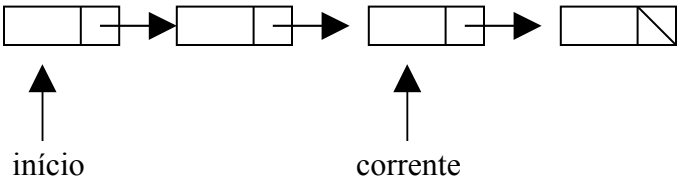
Em uma definição genérica de listas temos sempre que indicar uma posição específica onde uma inserção ou retirada vai acontecer. No entanto é bastante comum que uma série de operações sejam realizadas na mesma "região" da lista (várias posições próximas umas das outras).

Para simplificar a execução de um conjunto de operações em uma região da lista poderíamos modificar nossa definição de lista. Uma alternativa seria manter uma indicação de uma posição "corrente" onde todas as operações acontecem.

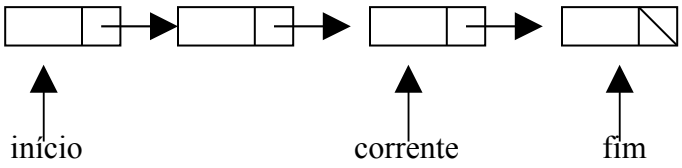
Assim as operações de inserção e retirada de elementos poderiam ser definidas dessa forma:

insere(elemento)
retira()

Como nessas operações não há indicação explícita de posição, na inserção o "elemento" será inserido na posição "corrente" e na retirada será removido da posição "corrente". Poderíamos portanto representar uma lista desse tipo como na figura abaixo:

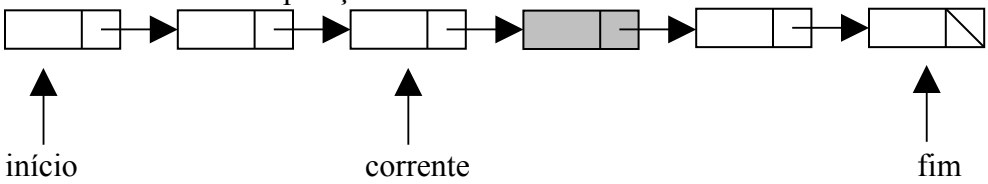


Pode ser útil também incluir na nossa representação de lista uma indicação de fim da lista. Assim podemos tratar de inserções, remoções e consultas no início e no fim da lista usando os indicadores "início" e "fim". Quando essas operações são realizadas em uma posição intermediária usamos o indicador "corrente".



Alguns problemas com essa organização: ainda é preciso percorrer toda a lista para atualizar o apontador do nodo anterior ao nodo corrente quando executamos remoções e inserções. Note que não temos uma referência direta para o nodo anterior.

Uma solução: o indicador de nodo corrente pode apontar na verdade para o nodo imediatamente anterior à posição corrente



Na representação acima o nodo corrente é o nodo preenchido com cinza. O indicador "corrente" aponta para o nodo imediatamente anterior a ele. Dessa forma, caso a operação "remover()" seja executada, removeremos o nodo preenchido com cinza, nessa situação todas as referências necessárias estarão disponíveis.

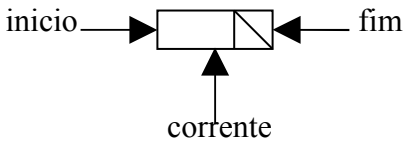
Problema: listas com 1 ou 0 elementos devem ser tratadas como casos especiais. Requerem código adicional na implementação.

Outra solução: nodo cabeça (*header*): igual aos outros nodos mas ignorado como elemento real.
Elimina os casos especiais de:

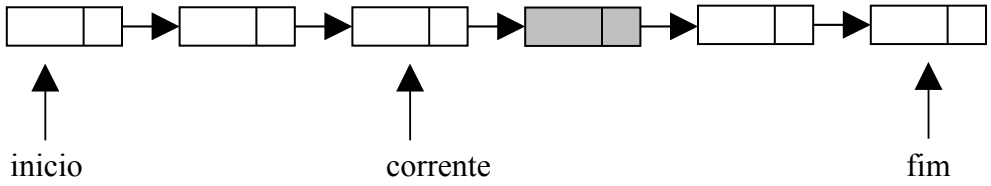
- lista vazia
- lista com 1 nodo
- corrente = 1º nodo

Custo adicional: espaço ocupado pelo *header*.

Estado inicial (lista vazia):



Estado intermediário (com nodos):



4. Implementações com listas de espaço livre:

Problema: manipulação de áreas de armazenamento livre pode ser ineficiente.

Uma solução: manter uma lista de nodos livres
- Quando um nodo é removido ele é inserido na lista de nodos livres.
- Antes de criar um novo nodo verificar se há nodos livres disponíveis (30% ganho no tempo de manipulação).

_____

Exercício: escreva a implementação para a operação inverterLista()

_____

Comparando implementações: vetores ou encadeamento?

Espaço:
Baseada em vetores:
- tamanho pré-determinado
- não crescem além desse tamanho
- uma lista com poucos elementos pode estar ocupando espaço inutilmente.

Encadeadas:
- ocupam somente o espaço necessário para os elementos que estão na lista
- limite de crescimento = memória disponível.

Vantagens de vetores:
- não há desperdício de espaço para cada elemento (as encadeadas precisam de um apontador)
- quando o vetor está completamente ocupado não há desperdício algum.

Em geral listas encadeadas requerem menos espaço quando há poucos elementos na lista.

Considerando espaço de armazenamento:

- listas encadeadas são melhores quando o número de elementos varia muito ou é desconhecido

- vetores são melhores quando sabemos aproximadamente o tamanho que a lista vai ter.

Tempo de Processamento:

Acesso:
- vetores:
    - mais rápido para acesso aleatório por posição;
    - operações como “próximo” e “anterior” são muito simples
- listas com encadeamento simples:
    - acesso a uma posição pode requerer um “caminhamento” na lista;
    - não há acesso explícito ao elemento anterior (operação “anterior”).

Inserção e Remoção:
- simples para listas encadeadas
- requer movimentação de elementos em vetores.

Em muitas aplicações “inserção” e “remoção” são mais importantes. Por essa razão muitas vezes escolhemos encadeamento.

Variações sobre a implementação

Elementos ou referências para elementos?

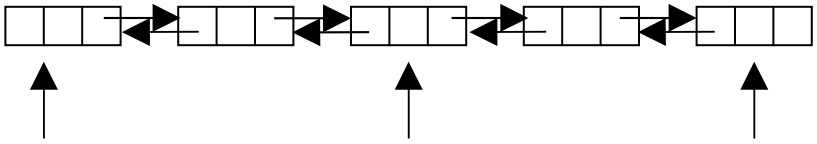
Quando utilizar referências:

- elementos ocupam muito espaço e há duplicação.

Listas Duplamente Encadeadas:

Facilitam o percurso na lista, uma vez que pode-se facilmente atingir o próximo nodo ou o nodo anterior.

Em cada nodo há acesso ao próximo nodo e ao nodo anterior:



inicio corrente fim

TAD NodoDuplo{
elemento: ELEM
próximo: referencia para nodo
anterior: referencia para nodo
Operações: ...
}

Exemplo da definição de uma classe em Java e struct em C:

```
/* Java */
class NodoDuplo {
      int elemento;
      NodoDuplo prox, ant;

      NodoDuplo (int valor) {
            elemento = valor;
            prox = ant = null;
}
```

```
/* C */
struct nododuplo {
        int elemento;
        struct nodo *prox, *ant;
} n;

typedef struct nododuplo NodoD;
…
n=malloc (sizeof (NodoD));
n->elemento = 5;
n->prox = n-> ant = NULL;
```

Mais fácil de implementar

- corrente: pode referenciar diretamente o nodo corrente (“anterior” indicará o anterior ao nodo corrente)

Mas:

- teríamos um caso especial para inserção em lista vazia
- insere não poderia ser usado para inserir no fim da lista

Portanto pode ser melhor manter o indicador “corrente” referenciando o nodo anterior ao nodo que é explicitamente o corrente.

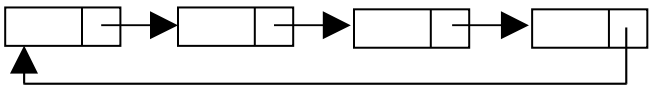
Inserção:

```
inserir(elemento)
        temp =  novoNodo()
        temp.anterior = corrente
        temp.próximo = corrente.próximo
        corrente.próximo = temp
        temp.próximo.anterior = temp
Fim.
```

Listas Circulares:

O último elemento possui uma referência para o primeiro. Em caso de implementação através de uma lista duplamente encadeada, o primeiro elemento também possui uma referência para o último.

Exs: com encadeamento simples



com duplo encadeamento: