LISTAS (LIST)

Listas são um importante recurso que estamos acostumados a utilizar! Fazemos lista de compras, lista de convidados para uma festa, lista de alunos em uma disciplina, etc.

Nosso objetivo agora é compreender e aprender a implementar o Tipo Abstrato de Dados Lista

Conjuntos de elementos podem ser intuitivamente representados através de listas. Essas são estruturas lineares, flexíveis, uma vez que as operações de inclusão e remoção de dados podem ser feitas em qualquer ponto da lista, conforme o critério de ordenação que for adotado. A ordem de uma lista pode ser simplesmente a ordem de inclusão, como ocorre com filas e pilhas, mas também pode ter algum campo(s) que determina(m) a relação de ordem dos elementos da lista. Por exemplo, se tivermos uma lista com dados de pessoas, esta poderia ser ordenada pelo nome.

Assim como pilhas e filas, as listas podem conter todo tipo de informação em cada um de seus elementos. O elemento pode ser apenas um número ou caractere, mas também pode ser um conjunto heterogêneo de dados: um registro. Por exemplo um registro com os dados de pessoas (código, nome, idade, telefone, etc.).

Listas são uma generalização das pilhas e filas. Todas são estruturas lineares, onde os elementos possuem relação de ordem entre eles. Essa relação pode ser estabelecida por contiguidade física (no caso dos vetores) ou por encadeamento (alocação dinâmica). Mas pilhas e filas possuem uma lógica de funcionamento restrita, pois pilhas somente incluem e removem elementos no topo (“fim” da pilha), já filas somente incluem elementos no fim da fila e somente removem elementos no início da fila. Por outro lado, as listas permitem que a inserção de novos elementos seja feita no início, meio ou fim da lista e a remoção de elementos também pode ser feita no início, meio ou fim da lista.

TIPOS DE LISTAS

Existem diversos tipos de listas, conforme a definição da estrutura de dados e lógica de operação.

Assim temos: Listas simplesmente encadeadas; Listas duplamente encadeadas; e Listas circulares.

Nas listas, cada nodo (também chamado de nodo ou nó) possui suas informações e referência (endereço).

Nas listas simplesmente encadeadas, a referência aponta para o próximo elemento da lista, que é um ponteiro para o endereço alocado para o nodo. Neste tipo de lista a varredura é sempre realizada em um único sentido, a partir do início da lista para frente (em inglês, forward), passando pelos nodos e chegando ao fim da lista, que é indicado pelo nodo em que a referência para o próximo é nula (NULL). Observe a seguir a representação desse tipo de lista.

Fonte: autor

Já nas listas duplamente encadeadas, é adicionado ao nodo mais um ponteiro (referência), que apontará para o elemento (nodo) anterior. Assim, se possui mais flexibilidade de varredura na lista, pois tanto podemos varrer a lista a partir do início para o fim, bem como do fim para o início e também se pode chegar a qualquer extremidade a partir de qualquer nodo! Percebas que o próximo do último elemento é nulo (NULL), indicando o fim da lista e que o anterior do primeiro elemento também é nulo (NULL), indicando o início da lista. Você pode ver isso na figura a seguir:

Fonte: autor

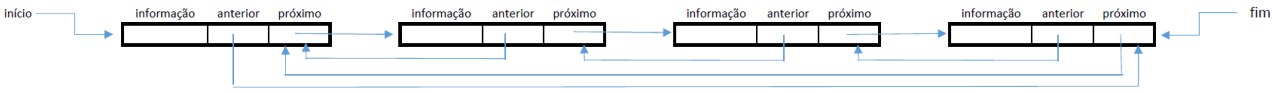
Agora vamos ver o que muda quando utilizamos listas circulares!

A ideia das listas circulares é criar um anel, como se a lista não tivesse nem início e nem fim. Isso torna mais flexível a varredura da lista. Assim, nas listas circulares

simplesmente encadeadas, a referência próximo do último nodo passa a apontar para o primeiro nodo, criando um círculo de encadeamento. Já no caso das listas circulares duplamente encadeadas, o ponteiro para o elemento anterior do primeiro nodo passa a apontar para o último. Assim, nesse tipo de lista, não temos delimitadores nos nodos (elementos) que indicam o início ou fim da lista (ponteiros próximo ou anterior com valores NULL), apenas as informações do cabeçalho da lista indicarão os limites da lista. A figura a seguir apresenta a estrutura de uma lista circular simplesmente encadeada, percebas que o ponteiro próximo do último nodo da lista aponta (referencia) para o primeiro nodo da lista.

Fonte: autor

E a seguir você pode visualizar uma lista circular duplamente encadeada:



A escolha do tipo de lista é uma decisão de quem está implementando o TAD (Tipo Abstrato de Dados) LISTA, pois não interfere na interface do TAD, visto que não interfere na interface de operações primitivas, mas sim define alternativas para a estrutura de dados do TAD e consequentemente terá impacto na implementação, que em alguns casos poderá facilitar a realização de algumas operações e pode tornar outras mais complexas.

IMPLEMENTAÇÃO DO TAD LISTA COM ENCADEAMENTO

Para trabalharmos com uma lista encadeada, temos que ter em mente que para cada novo elemento que é inserido na estrutura, devemos alocar dinamicamente um espaço de memória para armazená-lo. Desta forma, o espaço de armazenamento na memória é proporcional a quantidade de elementos. No entanto, a alocação dinâmica não ocupa espaços contíguos de memória, portanto não é possível ter acesso direto aos elementos

da lista. Assim, para constituir a lista precisamos encadear os elementos, ou seja, cada elemento precisa referenciar o próximo elemento da estrutura (e o anterior, no caso de listas duplamente encadeadas) e essa referência aponta para o endereço alocado para cada elemento, conforme a ordem pré-definida para a lista. Desta forma, cada elemento possui sua informação (número, caractere, estrutura de informações, etc.) mais a informação de referência que é ponteiro para o próximo elemento e ponteiro para o anterior, conforme o tipo de lista.

Vamos considerar a implementação de uma Lista Simplesmente Encadeada!

Precisamos definir, usando a linguagem de programação, a estrutura de dados da LISTA, seu cabeçalho e as operações primitivas do TAD que constituem sua interface básica. Vamos considerar em nossos exemplos um registro simples de dados de pessoas, com as seguintes informações: código (que identifica cada pessoa); nome; idade. Você pode facilmente incluir mais informações nesse registro. Esses dados foram escolhidos pensando em diferentes possibilidades de ordenação da lista. Assim, a informação do nodo será uma estrutura conforme segue:

```
struct dado {
 int codigo;
 char nome[40];
 int idade;
};
```

Fonte: autor

A partir disso, podemos definir a estrutura do nodo:

```
struct nodo {
 dado informacao;
 struct nodo* proximo;
};
```

Fonte: autor

Agora, faremos a definição do cabeçalho da lista - nós já utilizamos esse recurso na implementação das FILAS e tem por objetivo ter as informação da estrutura da lista. As

informações básicas do cabeçalho são: início (ponteiro para o primeiro nodo da lista); fim (ponteiro para o último nodo da lista); tamanho (que indica quantos elementos constituem a lista, economizando o processo de varrer a lista para obter esse dado). Assim, podemos definir esta estrutura:

```
struct lista {
 int tamanho;
 struct nodo* inicio, fim;
};

struct lista* LISTA;
```

Fonte: autor

A partir da definição da estrutura de armazenamento do TAD LISTA, podemos definir sua interface de funções, ou seja, as operações primitivas do TAD que manipulam sua estrutura.

```
//função para criação da lista
LISTA* cria ();

//função que testa se a lista está vazia
int vazia (LISTA * l);

//função que libera a memória
void libera (LISTA * l);

//função que retorna o tamanho da lista
int elementos(LISTA * l);

//função para incluir os dados v na lista l
void insere (LISTA * l, dado v);

//função que retorna o valor do início da lista l e o exclui
int remove (LISTA * l, dado* v);
```

Fonte: autor

Na sequência você poderá analisar a implementação de cada uma dessas funções.

Podemos implementar a função de criação da lista de forma bem simples, pois consiste apenas em alocar o cabeçalho, inicializar o tamanho e atribuir NULL para o início e fim da lista, pois estamos criando uma lista vazia (que ainda não possui elementos).

```
//função para criação da lista
LISTA * cria (){
 LISTA* p;
 p = (LISTA) malloc (sizeof (LISTA));
 p->tamanho=0;
 p->inicio=NULL;
 p->fim=NULL;
 return(p);
}
```

Fonte: autor

Para verificar se uma lista está vazia, podemos testar se o tamanho é 0 (zero) ou se o início ou fim são nulos (NULL). Optaremos por testar o início da lista, se este for igual NULL indica que não há elementos na lista e a função retornará o valor 1, caso contrário retornará 0. Veja a seguir a implementação desta função:

```
//função que testa se a lista está vazia
int vazia (LISTA * l){
 if (l->inicio == NULL)
    return(1);
 else
    return(0);
}
```

Fonte: autor

A função que libera memória precisa liberar o espaço alocado para o cabeçalho. Em princípio, consideraremos que a lista está vazia e portanto não precisam ser liberados os nodos ocupados. Seria possível implementar um função que esvazia uma lista, mais adiante faremos isso. A seguir vejas a implementação da função libera:

```
//função que libera a memória
void libera (LISTA * l){
 free(l);
}
```

Fonte: autor

Para consultar a quantidade de elementos da lista, basta consultar a informação de tamanho do cabeçalho da lista, conforme segue:

```
//função que retorna o tamanho da lista
int elementos(LISTA * l){
 return(l->tamanho);
}
```

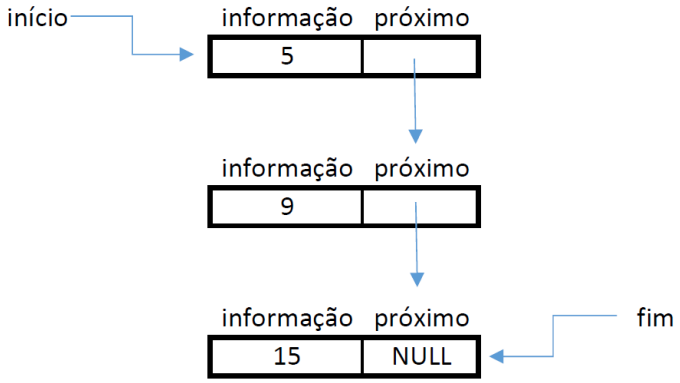
Fonte: autor

Agora vamos analisar a implementação da função de inclusão de novos elementos.

Um novo elemento pode ser incluído no fim da lista, neste caso a lista passa a ter um comportamento semelhante ao de uma fila, onde o critério de ordenação é a ordem de inclusão. Por outro lado, a lista pode ter outro critério de ordenação de seus elementos, que pode ser a ordem crescente dos códigos de identificação das pessoas. Nesse caso, o novo elemento poderá ser incluído no início da lista, no meio da lista ou no fim da lista.

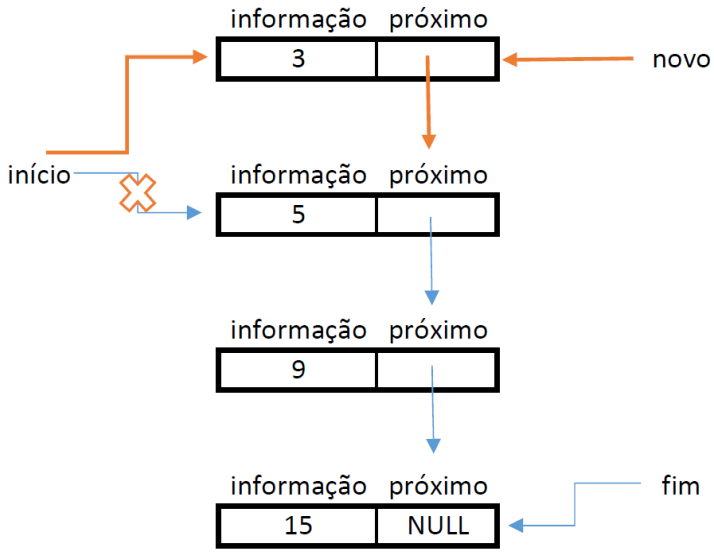
Vamos analisar essa situação!

Considere uma lista em que já foram incluídos 3 elementos (na figura, apenas estamos representando os códigos da informação). Se formos adicionar um novo nodo em que o código da pessoa for 3, esse deverá ser incluído antes do atual início da lista, portando passará a ser o novo início. Agora, se formos adicionar o código 11, este novo elemento deverá ser encadeado entre os nodos de código 9 e 15. Por outro lado, se formos adicionar o nodo com o código 20, este deverá ser encadeado no fim da lista.



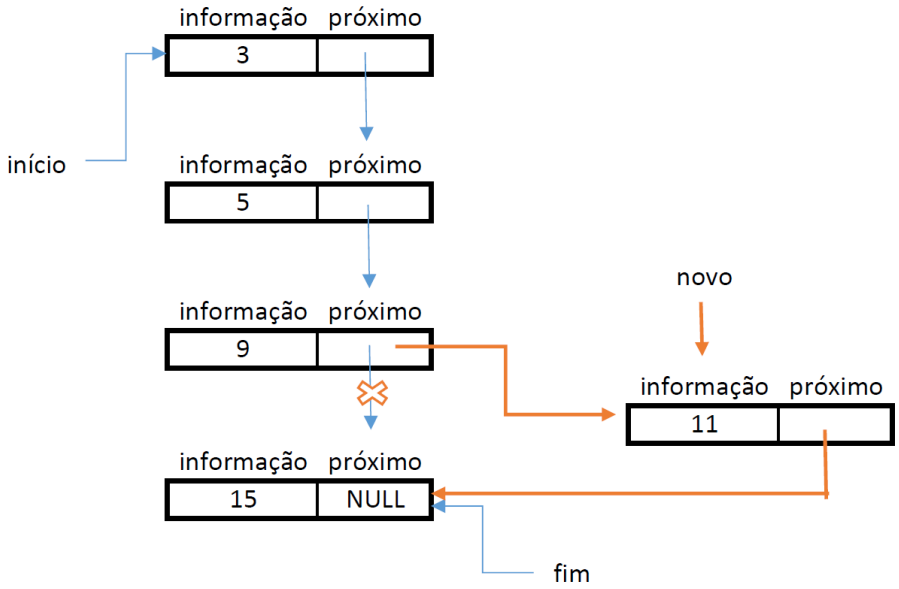
Fonte: autor

Verifiques na figura a seguir a inclusão de um novo elemento antes do início da lista, onde o novo elemento com código 3 tem como próximo o atual início da lista e o início passa a apontar para o novo elemento.



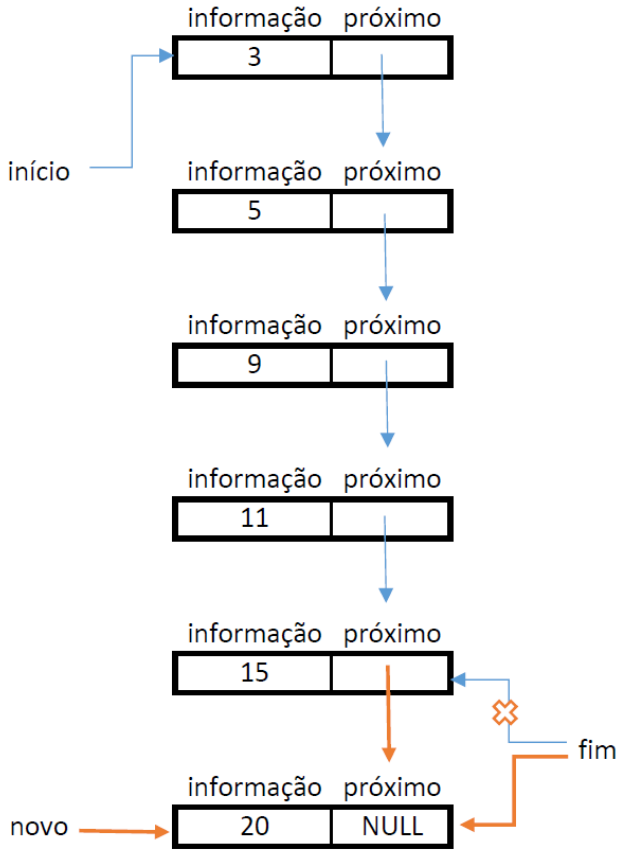
Fonte: autor

Agora, o diagrama demonstra a inclusão de um novo elemento no meio da lista, ajustando os ponteiros para manter o encadeamento conforme o critério de ordenação.



Fonte: autor

Por fim, vejas a representação da inclusão de um novo nodo no fim da lista:



Fonte: autor

Como definir onde o novo nodo deve ser incluído?

Para definir onde o novo nodo deve ser incluído, teremos que fazer uma varredura da lista, ou seja, a partir do início da lista comparar a chave de ordenação do novo nodo com a do nodo corrente, se a do novo elemento for menor, significa que ele será inserido antes do atual. Mas se chegarmos ao final da lista e não encontrarmos um elemento maior que o novo, ele se tornará o novo final da lista. Em todas as situações é necessário preservar o encadeamento.

A implementação da função de inclusão de um novo nodo deve considerar todas essas situações e inclusive o fato de a lista estar vazia. Então, vamos ver essa implementação:

```
//função para incluir os dados v na lista l
void insere (LISTA * l, dado v){
 nodo* novo;
 nodo* anterior, corrente;
 novo = (nodo) malloc(sizeof(nodo));//alocação dinâmica do novo nodo
//a seguir passamos as informações do registro v para o nodo
 nodo->codigo = v.codigo;
 strcpy(nodo->nome,v.nome);
 nodo->idade = v.idade;
 nodo->próximo=NULL;
 if (vazia(l)){ //o novo elemento será o início e fim da lista
  l->inicio = novo;
  l->fim = novo;
 }
 else{
 //agora precisamos determinar a posição de inclusão do novo nodo
   corrente = l->inicio;
   if (novo->codigo < corrente->codigo){//inclusão no início
     novo->proximo = l->inicio;
     l->inicio = novo;
   }
   else {//varredura da lista
   while((corrente != NULL)&&(novo->código >= corrente->codigo)){
     anterior = corrente;
     corrente = anterior->proximo;
   }
   anterior->proximo = novo;
   novo->proximo = corrente;
   if (corrente == NULL)
      l->fim = novo;
   }
 }
  l->tamanho = l->tamanho + 1; //incrementa tamanho
}
```

Fonte: autor

Por fim, vamos implementar a função que remove um elemento da lista e retorna as suas informações! Em primeiro lugar, precisamos determinar qual será o nodo que será excluído! Optamos por receber o código do elemento a ser excluído, localizá-lo, consultar seus dados e então excluí-lo. Podemos identificar 4 situações a partir desta busca: 1ª) O código não existe, portanto nada será excluído ou retornado; 2ª) O

elemento a ser excluído é o primeiro da lista, portanto o seu início precisa ser ajustado; 3ª) O elemento a ser excluído está no meio da lista, assim precisamos ajustar os ponteiros; 4ª) O elemento é o último elemento da lista, portanto precisamos ajustar o seu fim. Sempre que um elemento for removido, precisamos atualizar o tamanho da lista, decrementando. Além disso, a função retorna 1 se teve êxito e zero, caso contrário.

```
//função que retorna um elemento da lista e o exclui
int remove (LISTA * l, dado* v){
 int código = *v.codigo;
 nodo* anterior, corrente;
 if (vazia(l)){ return(0);}
 else{
   corrente = l->inicio;
   if (codigo == corrente->codigo){//achou no início!
     strcpy(*v.nome,corrente->nome);
     *v.idade=corrente.idade;
     l->inicio = corrente->proximo;
     free(corrente);
     l->tamanho = l->tamanho – 1;
     return(1);
   }
   else{
     while((corrente != NULL) &&
           (codigo != corrente->codigo)&&
           (codigo > corrente->codigo)){
       anterior = corrente;
       corrente = anterior->próximo;
     }
     if (corrente == NULL){//não encontrou
      return(0);
     }
     else {
       strcpy(*v.nome,corrente->nome);
       *v.idade=corrente.idade;
       anterior->próximo = corrente->próximo;
       if (l->fim == corrente){//é o último elemento
         l->fim = anterior;
       }
       l->tamanho = l->tamanho – 1;
       free(corrente);
       return(1);
     }
   }
 }
}
```

Fonte: autor

Podemos adicionar algumas operações a interface desse TAD, como por exemplo: procurar um elemento na lista e se encontrar retorna os dados; e esvaziar a lista. Vamos implementar essas duas funcionalidades!

```
//função que esvazia a lista
void esvazia(LISTA * l){
 nodo* corrente, libera;
 if (!vazia(l)){
   corrente=l->inicio;//ponteiro auxiliar para navegar na lista
   while(corrente != NULL){
     libera = corrente;
     corrente = libera->próximo;
     free(libera);
    }
//atualizar o cabeçalho da lista que agora está vazia
  l->tamanho = 0;
  l->inicio = NULL;
  l->fim = NULL;
  }
}
```

Fonte: autor

Pronto, agora já temos o ferramental básico para gerenciar o TAD LISTA! Sua flexibilidade para as operações de inclusão e remoção de elementos o torna um pouco mais complexo que PILHAS e FILAS, mas temos certeza que você conseguiu entender como funciona.

Desafio!

Altere as funções insere, remove e procura, considerando como critério de ordenação da lista o nome de uma pessoa! Portanto a lista será em ordem crescente de nome! Temos certeza que você consegue!

Amplie seus conhecimentos!

Pesquise sobre a implementação de listas duplamente encadeadas! O que muda nas funções para inserir e remover elementos? Quais as vantagens do encadeamento duplo!

FAÇA O AUTOESTUDO

Vamos lá! Verifiques o que aprendeu nesta lição?

1. Construa uma tabela comparativa com os TAD PILHA, FILA e LISTA. Considere os seguintes aspectos de cada TAD: estrutura do nodo, lógica de funcionamento, operações primitivas, inclusão de novos elementos, exclusão de elementos. Ao final, analise a tabela que construiu e identifique as principais semelhanças e diferenças entre estas estruturas lineares.
2. Liste os tipos de listas encadeadas e os descreva brevemente, diferenciando-os.
3. O que é o cabeçalho de uma lista e quais informações contém?
4. Descreva a operação de inserir um novo elemento em uma lista?
5. Descreva a operação de remover um elemento de uma lista?
6. Considere a seguinte estrutura de informações de uma lista que gerencia coordenadas cartesianas (x,y).

```
struct dado {
 float x, y;
};
```

Além disso, os elementos (nodos) são mantidos em ordem de inclusão, ou seja, novos nodos são sempre incluídos no fim da lista. Para implementar o TAD lista para esta estrutura de dados, quais funções precisam ser alteradas e quais não necessitam de ajustes?

7. Temos um conjunto de dados E, que define a ordem de códigos (separados por vírgulas) que serão progressivamente incluídos em uma lista. E={7, 20, 11, 30, 9, 5}.

Você deve fazer a representação gráfica (desenho) da lista, demonstrando sua situação após cada operação de inclusão de dados. Identificando início e fim

da lista, tamanho e encadeamento entre os elementos. Tome por base a estrutura de funcionamento de uma lista simplesmente encadeada.