TIPO ABSTRATO DE DADOS - LISTA

Representação por encadeamento

Uma lista é uma seqüência de elementos. As características de uma lista são: a ordem de seus elementos, seu comprimento (número de elementos), além, é claro, do conjunto de elementos. Uma das representações de uma lista em um computador é por meio de encadeamento: Junto a cada registro da lista guardamos o endereço de memória do registro seguinte. Essa representação é denominada lista encadeada ou lista ligada.

Uma das implementações de uma lista ligada define a lista por uma estrutura que contém o endereço do primeiro registro da lista e o comprimento da lista, por exemplo. Essa estrutura está sempre no mesmo lugar da memória, mesmo que a lista seja vazia. Vamos considerar a implementação do TAD Lista, considerando um tipo pré-definido de elementos da lista denominado t_Item.

TAD Lista (versão com cabeça)

Estruturas de armazenamento de dados

```
typedef struct Celula{
        t_Item elemento;
        struct Celula *next;
} Celula;

typedef struct{
        Celula *inicio;
        int tamanho;
} Lista;
```

INTERFACE:

1) Construtor: Lista criarLista( ); // Esse construtor cria uma lista vazia.

2) Acesso:

```
int buscarElemento(Lista, char);          // Devolve a ordem do elemento na lista.
t_Item obterElemento(Lista, int);         // Devolve o elemento da lista cuja ordem é dada.
bool verificarListaVazia(Lista);          // Devolve true se a lista está vazia, false, caso contrário.
void mostrarLista(Lista);                 // Apresenta todos os elementos na seqüência da lista.
int determinarTamanho(Lista);             // Devolve o tamanho da lista.
```

3) Manipulação

```
void inserirLocal(Lista*, t_Item, int k); // Acrescenta um novo elemento na posição k .
void removerLocal(Lista*, int);           // Remove o elemento da lista cuja ordem é dada.
```

4) Destrutor: void esvaziarLista(Lista*); // Torna a lista vazia.

Em uma lista encadeada simples, a operação de inserção de menor custo é a inserção no início da lista, uma vez que o acesso nesse tipo de representação é sequencial.

Para inserir um elemento no final da lista, teríamos que percorrê-la inteiramente. Se esse tipo de inserção é necessário, acrescentamos um campo com o endereço do último registro.

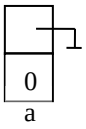
Não se aplicam, em uma lista encadeada, os algoritmos clássicos de ordenação (sorting) porque exigem acesso direto. Em vez disso, adaptamos o processo de inserção na lista encadeada para fazer a inserção em uma lista ordenada, isto é, mantemos a lista ordenada por meio da operação de inserção.

Vamos considerar na interface a seguir, as operações básicas de uma lista linear, considerando a implementação do tipo Lista com a operação de inserção em uma posição determinada, fornecida por um parâmetro inteiro.

Esquemas de ilustração

Construtor: Lista criarLista( ){.....}

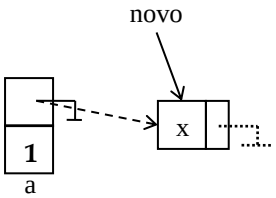
Basta colocar NULL no campo inicio e zero no campo tamanho



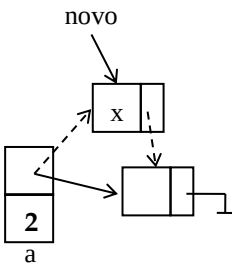
Inserção: void inserirLista(t_Item x, Lista *a, int k){.....}

Vamos considerar que o valor de k seja válido nessas ilustrações. Inicialmente criamos uma nova célula. Seja novo o endereço da mesma. No campo elemento copiamos o valor de x. Qualquer que seja o caso de inserção, o campo tamanho deverá ser atualizado.

caso (1) k = 1 e a lista é vazia.
O seguinte de novo é NULL.
O ponteiro inicio deve receber o valor de novo.



caso (2) k = 1 e a lista não é vazia.
O seguinte de novo é o primeiro da lista.
O ponteiro inicio deve receber o valor de novo.
Observe que esse procedimento serve para o caso 1.

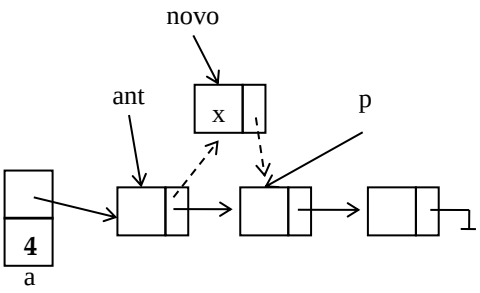


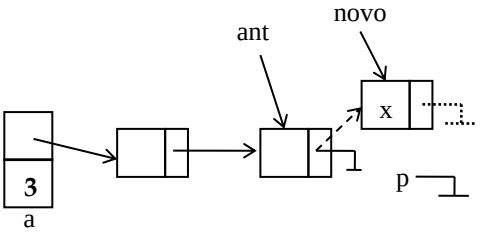
caso (3) k ≠ 1.

Neste caso inicialmente é preciso localizar o registro de ordem k. Uma vez que devemos alterar o campo next do registro anterior, é preciso obter também o endereço do anterior.

Nos desenhos, p e ant são dois pointers, p contém o endereço do registro de ordem k e ant contém o endereço do registro anterior.

Observe que se k = n+1 o novo registro deve ser inserido no final da lista, após o último. Neste caso, ant deve apontar para o último e p deve ter NULL.





O seguinte de novo é o registro apontado por p. O campo next do anterior deve receber novo.

Esboço do algoritmo de inserção para inserção no início:

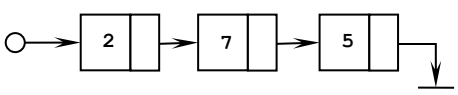
```
Alocar espaço para uma célula
Copiar valor no campo item
Definir valor no campo next
Ligar a célula à lista existente (no início)
Atualizar tamanho da lista
```

Esboço do algoritmo de inserção para inserção na posição k:

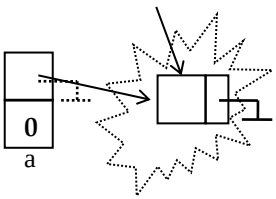
```
Alocar espaço para uma célula
Copiar valor no campo item
Se k = 1   então   Definir valor no campo next
                   Ligar a célula à lista existente (no início)
           senão   Localizar o registro de ordem k (o novo registro deve entrar antes dele)
                   Ligar a célula à lista existente
                      Definir valor no campo next do novo
                      Definir valor no campo next do anterior
Atualizar tamanho da lista
```

Remoção: void removerLista(Lista *a, int k){.....}

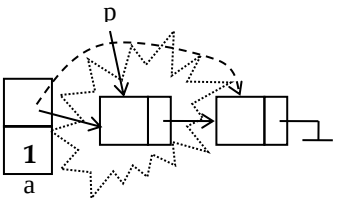
Vamos considerar que o valor de k seja válido nessas ilustrações. Temos que localizar a célula que contém o registro de ordem k pois a mesma deve ser liberada para que o espaço de memória possa ser reutilizado. Seja p o endereço da mesma. No final da remoção, devemos liberar a célula apontada por p. Qualquer que seja o caso de inserção, o campo tamanho deverá ser atualizado.



caso (1) k = 1 e a lista contém um único registro.
O ponteiro p recebe o endereço da primeira célula.
O ponteiro inicio recebe NULL.

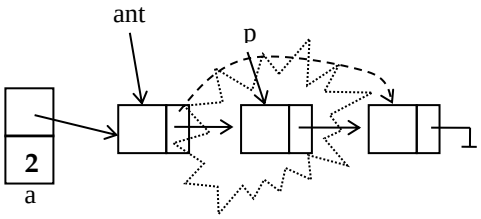


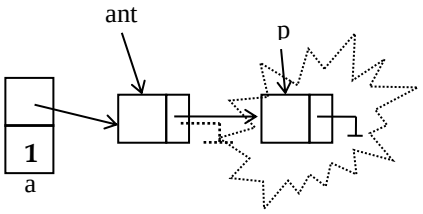
caso (2) k = 1 e a lista contém mais do que um registro.
O ponteiro p recebe o endereço da primeira célula.
O ponteiro inicio recebe o endereço do seguinte de p.
Observe que é o mesmo procedimento do caso 1.



caso (3) k ≠ 1.

Neste caso inicialmente é preciso localizar o registro de ordem k. Uma vez que devemos alterar o campo next do registro anterior, é preciso obter também o endereço do anterior. Nos desenhos, p e ant são dois pointers, p contém o endereço do registro de ordem k e ant contém o endereço do registro anterior. Observe que se k = n o último registro deve ser removido. Neste caso, o seguinte de ant deve ser NULL (que é o seguinte de p).

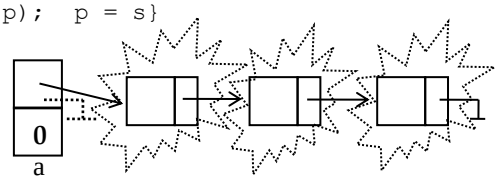




O campo next do anterior recebe o endereço do seguinte de p.

DESTRUTOR

```
void esvaziarLista(Lista *a){
   Célula *p, *s;
   p = a->inicio;
   while (p !=NULL) {s = p->next; free(p);  p = s}
   a->inicio = NULL;
   a->tamanho = 0;
}
```



TAD Lista (versão sem cabeça)

Outra implementação de uma lista ligada define a lista sem a estrutura de inicio, isto é, a lista é somente uma referência para a primeira célula, que contém o primeiro registro da lista.

Estruturas de armazenamento de dados

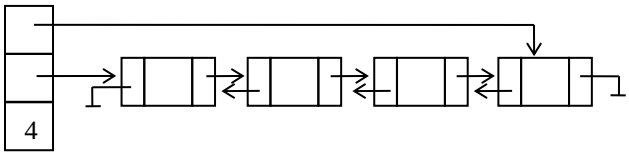
```
typedef struct Celula{
    int elemento;
    Celula *next;
} Celula;

typedef Celula* Lista;
```

Na ilustração, L1 = (2,7,5) e tamanho da lista = 3

TAD Lista - LISTA DUPLAMENTE LIGADA

Uma lista duplamente ligada é uma lista representada por duplo encadeamento: cada célula contém dois ponteiros, um para a célula seguinte e outro para a célula anterior.




```
typedef struct Celula{
    int elemento;
    struct Celula *suc;
    struct Celula *pre;
} Celula;

typedef struct{
    Celula *inicio;
    Celula *fim;
    int tamanho;
} Lista;
```

EXERCÍCIOS

1) Implemente o TAD Lista na versão "com cabeça" para uma lista de números inteiros.

Lista ligada com cabeça

```
typedef struct Celula{
        int elemento;
        struct Celula * next;
}Celula;

typedef struct{
  int tamanho;
  Celula * inicio;
}Lista;

//INTERFACE
Lista criarLista();  // cria uma lista vazia
int determinarTamanho(Lista);        // devolve o tamanho da lista
bool verificarListaVazia(Lista);     // devolve TRUE se a lista é vazia
void mostrarLista(Lista);            // mostra na tela os elementos da lista
int buscarElemento(Lista, char);     // devolve a posição na lista
t_Item obterElemento(Lista, int);    // devolve o registro de ordem k na lista
void inserirLista(Lista *, t_Item);  // acrescenta o novo no inicio
void removerLocal(Lista *, int);     // remove o registro de ordem k
void esvaziarLista(Lista *);         // torna a lista vazia
```

2) Para o tipo Lista do exercício anterior, escreva a implementação da função

```
void inserirOrdenada(Lista *, int);
```

que insere cada registro mantendo a lista em ordem crescente (não decrescente).

3) Para o tipo Lista do exercício anterior, escreva a implementação da função

```
void buscaInsere(Lista *, int);
```

que insere um registro somente se não é duplicata de qualquer registro existente na lista.

4) Implemente o TAD Lista na versão "sem cabeça" para uma lista de números inteiros.

```
typedef struct Celula{
        int elemento;
        struct Celula * next;
}Celula;

typedef Celula* Lista;

//INTERFACE
Lista newLista();                  // cria uma lista vazia
int determinarTamanho(Lista);      // devolve o tamanho da lista
bool verificarListaVazia(Lista);   // devolve TRUE se a lista é vazia
void mostrarLista(Lista);          // mostra na tela os elementos da lista
int buscarElemento(Lista, char);   // devolve a posição do registro na lista
t_Item obterElemento(Lista, int); // devolve o registro de ordem k na lista
void inserirLista(Lista, t_Item); // acrescenta o novo no inicio
void removerLocal(Lista, int);     // remove o registro de ordem k
void esvaziarLista(Lista);         // torna a lista vazia
```