Variantes de Lista Encadeada

1. Introdução

A lista encadeada que estudamos até aqui possui alocação dinâmica e encadeamento simples, sendo útil em muitas situações. O código aqui trabalhado considera a inserção no início da lista, que é o código mais simples.

Entretanto, o conceito de lista permite diversas adaptações, conforme as características da aplicação. Neste texto veremos as seguintes variantes: lista encadeada ordenada; lista encadeada com nodo cabeça; lista duplamente encadeada; lista encadeada circular. Obviamente podem ser feitas combinações dessas variantes, a exemplo da lista duplamente encadeada ordenada.

Cada variante altera algum detalhe no código, e no seu desenho.

Importante: Se você entender o funcionamento da estrutura no desenho, fica fácil implementar os códigos de cada uma das operações sobre a lista.

Para todas as variantes, vamos considerar o conteúdo dos dados o mais simples possível, ou seja, um valor inteiro. A título de comparação, considere a inserção dos quatro valores inteiros, 4, 9, 5 e 3, nesta ordem. Para a primeira lista já implementada, com inserção no início, o resultado da inserção é a lista representada abaixo.

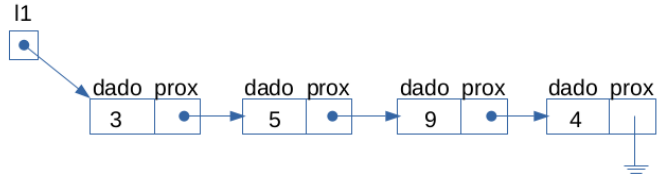


Figura 1. Lista simplesmente encadeada, com iserção no início.

2. Lista Encadeada Ordenada.

No caso da lista ordenada, o desenho da estrutura do nodo, e do apontador para o início da lista são iguais aos da primeira lista. O que muda é somente a rotina de inserção de um elemento na lista. A função `Lista insereLista(Lista l, int e)` precisa inserir um elemento `e` na lista `l` na ordem definida (crescente ou decrescente). Para esse exemplo, vamos usar uma lista ordenada em ordem crescente.

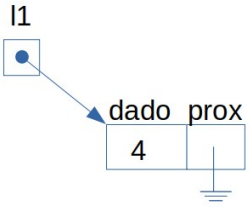
Antes da inserção do primeiro elemento, é preciso inicializar a lista, (`l1 = criaLista();`) que nesse caso resulta em uma variável do tipo `Lista` (que é um apontador para uma estrutura do tipo `struct elemento`), apontando para `NULL`.

Isso indica que a lista está vazia.

Após inserir o primeiro elemento, com a chamada

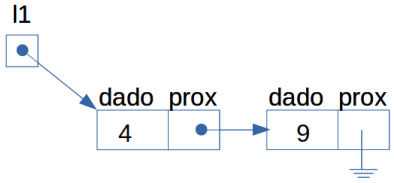
```
l1 = insereLista(l1,4);
```

o resultado será:



Observe que agora o `l1` não contém mais o valor `NULL`, mas sim o endereço do primeiro elemento da lista (valor 4), que também é o último elemento (campo `prox` contém o valor `NULL`). Até aqui sem novidades, o funcionamento é o mesmo da lista vista anteriormente.

No entanto, quando inserimos o valor 9, ele não deve ser inserido no início da lista, mas depois do elemento 4, porque o 9 é maior que o 4. Então, o algoritmo de inserção, além de testar se o início da lista é nulo (se for nulo precisa inserir no início mesmo), caso não seja nulo, precisa testar se a inserção deve ser antes ou depois do elemento corrente. Nesse caso, depois. E o desenho ficará assim:

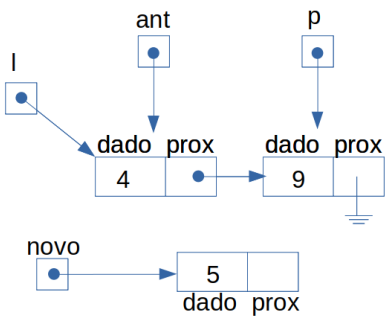


Na codificação deste ponto, é importante saber que antes da inserção, depois do 4 não há outros elementos, e portanto, chegou ao final da lista. Quando se chega no final da lista, é importante saber qual o último elemento testado, pois o novo elemento (nesse caso, o 9) precisa ser indicado como o próximo elemento depois do anterior (nesse caso, o 4). Isso é necessário para manter a conexão entre os elementos. Mas para isso é necessário armazenar o endereço do elemento 4. Uma possibilidade para isso é criar uma variável `ant`, além da variável `p`, que aponta para o elemento corrente (aquele que eu estou testando para ver se o novo elemento é maior ou menor).

Na sequência, vamos inserir o elemento 5, pela chamada da função:

```
l1 = insereLista(l1,5);
```

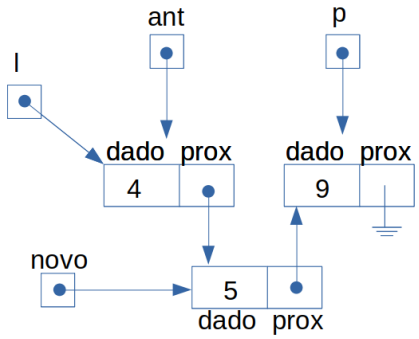
Observe que o valor 5 deve ser inserido no meio da lista, pois é maior que o 4 e menor que o 9. Sempre que temos uma inserção no meio da lista, uma vez encontrado o elemento apontado pelo ponteiro `p`, (nesse caso, 9) que é maior que o novo elemento `e` (nesse caso, 5), e o elemento apontado por `ant`, que é o elemento anterior da lista (nesse caso, 4), que é, portanto, menor que `e`, podemos dizer que o próximo elemento do anterior será o novo, e que o próximo elemento do novo será o elemento corrente (apontado por `p`), nesse caso, 9.



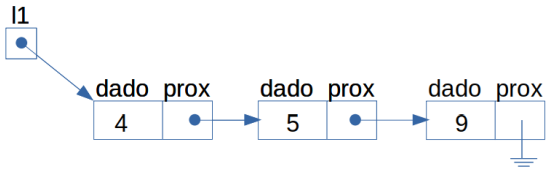
Uma sequência possível de comandos para fazer essa conexão seria:

```
ant -> prox = novo;
novo -> prox = p;
```

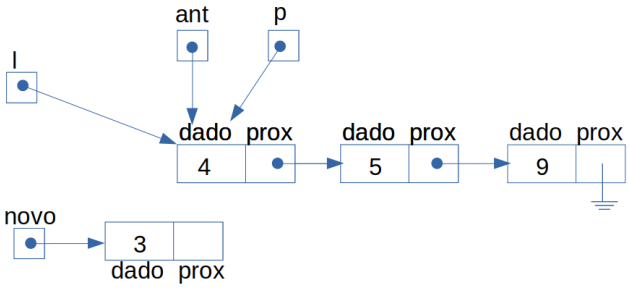
Após esses dois comandos, a conexão seria:



Ao sair da função, as variáveis internas (p, ant e novo) são desativadas, e a lista será



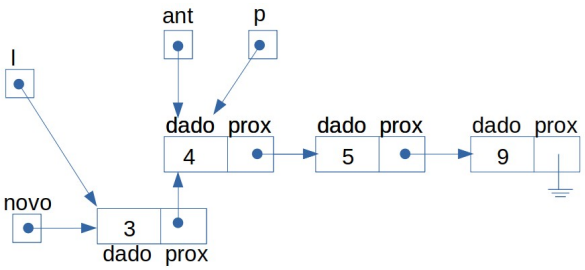
Ao inserir o último número (3), a inserção se dará no início da lista, alterando o endereço do primeiro elemento. Como essa rotina de inserção tem dois ponteiros para percorrer a lista (`ant` e `p`), uma forma de identificar se a inserção foi antes do primeiro elemento é verificar se `ant` é igual a `p`. Se ambos forem iguais, então a inserção é no início da lista, e o início da lista precisa ser atualizado. Ao sair do laço que busca a posição de inserção, a rotina insereLista() estará em um contexto similar ao da figura abaixo.



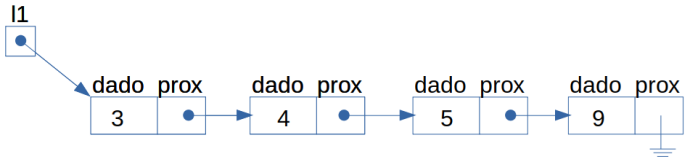
Observe que nesse caso, os dois comandos citados no exemplo anterior não funcionam, pois o novo não será o próximo do ant. Isso só acontece se `ant == p`, ou seja, o primeiro elemento da lista já é maior que o novo elemento. Nesse caso, o novo será inserido antes do primeiro da lista. Ou seja, o próximo do novo será o elemento apontado por `p` (ou `l`, ou `ant`). E a lista passa a iniciar pelo novo elemento. E isso pode ser feito com os seguintes comandos

```
novo -> prox = p;
l = novo;
```

O que deixa a lista da seguinte forma:



E, ao sair da função, a lista `l1` ficará da seguinte forma:



E a lista permanece ordenada em todos os momentos.

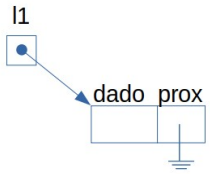
Lembre que para testar se o algoritmo de inserção ordenada funciona, é preciso testar todas as situações possíveis. Nesse caso, testar a inclusão: (i) em uma lista vazia; (ii) ao final da lista; (iii) no meio da lista; e (iv) no início de uma lista não vazia; Essa sequência testou as quatro situações.

Segue um exemplo de implementação da inserção em lista ordenada.

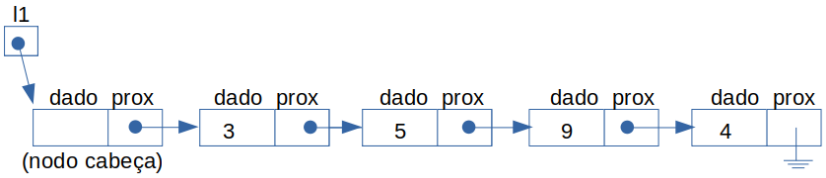
```
Lista insereLista(Lista l, int e)
{
	Lista p, ant, novo;

	// Aloca o espaço e faz as atribuições de valores
	novo = malloc(sizeof(struct elemento));
	novo -> dado = e;
	// Procura o ponto de inserção na lista
	p=l;
	ant=p;
	while ((p != NULL) && (p->dado < e)) {
		ant = p;
		p = p->prox;
	}
	if (p != ant) { // Não vai inserir antes do primeiro, insere entre ant e p
		ant->prox = novo;
	}
	else {  // Lista vazia ou inserindo antes do primeiro elemento
		l = novo;
	}
	novo->prox = p;
	return l;
}
```

3. Lista Encadeada com Nodo Cabeça.

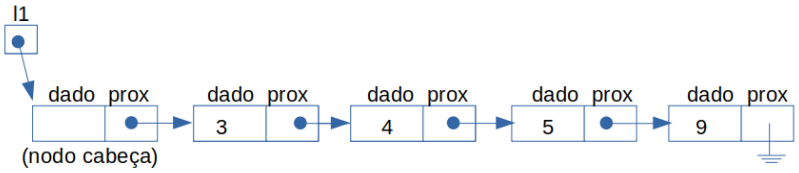
Um modo de simplificar alguns dos algoritmos das operações sobre uma lista encadeada é criar um nodo que servirá somente para indicar o início da lista, e seu conteúdo será ignorado. Assim, uma lista vazia `l1` não será simplesmente um ponteiro para NULL, mas um ponteiro para um único elemento, cujo conteúdo será ignorado, e o próximo elemento dele é NULL, como na figura abaixo.



Ao final da inserção dos mesmos 4 valores (4, 9, 5 e 3), com inserção no início da lista, o resultado seria



Para uma lista encadeada ordenada, com nodo cabeça, o resultado da inserção da mesma sequência seria



As operações sobre a lista encadeada com nodo cabeça são as mesmas, mas a sua implementação muda.

3.1 Criação de uma lista vazia

A criação de uma lista com nodo cabeça vazia precisa alocar o nodo cabeça, e fazer os apondamentos corretos. Isso pode ser implementado da seguinte forma.

```
Lista criaLista()
{
     Lista novo;

     novo = malloc(sizeof(struct elemento));
     novo->prox = NULL
     return(novo);
}
```

3.2. Inserir um elemento na lista encadeada com nodo cabeça.

Inserir um elemento na lista dinamicamente encadeada implica em criar um novo nodo, colocar o conteúdo desejado nesse novo nodo e depois conectar esse novo nodo na lista. No caso de uma lista com nodo cabeça, o início da lista nunca será alterado na inserção ou retirada. Então, não é necessário retornar o ponteiro para o início da lista. Um exemplo de código para inserir um elemento no início da lista é mostrado abaixo.

```
// Insere um elemento e no início da lista l
// O primeiro elemento da lista é indicado por l->prox
void insereLista(Lista l, int e)
{
     Lista novo;

     // Aloca o espaço e faz as atribuições de valores
     novo = malloc(sizeof(struct elemento));
     novo->dado = e;

     // O próximo do novo é o início da lista
     novo->prox = l->prox;

     // O primeiro elemento é o novo elemento
     l->prox = novo;
}
```

3.3. Contar o número de elementos de uma lista encadeada com nodo cabeça.

Para contar o número de elementos (ou nodos) de uma lista é necessário percorrer toda a lista, sequencialmente, do primeiro elemento até o final da lista, incrementando um contador. Ao final, retornar o contador de elementos. É preciso lembrar que o primeiro nodo não é um elemento válido. Isso pode ser implementado pelo código abaixo.

```
// Conta o número de nodos da lista l
int contaLista(Lista l)
{
     Lista p;
     int cont = 0;       // inicia o contador com 0

     p = l->prox;        // aponta p para o início da lista
     while (p != NULL)   // enquando não acabou a lista
     {
          cont++;        // incrementa o contador
          p = p->prox;   // passa para o próximo elemento
     }
     return cont;        // retorna o contador
}
```

Os códigos das demais operações sobre a lista com nodo cabeça são deixados como exercício.

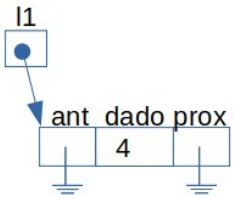
4. Lista Duplamente Encadeada

Uma característica de uma lista simplesmente encadeada é só ser possível percorrer a lista em um sentido (no sentido dos ponteiros). Ela não é projetada para permitir o percurso no sentido inverso. Mas isso pode ser feito se for implementado um encadeamento duplo (nos dois sentidos).

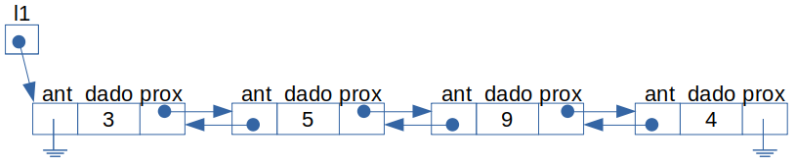
Para isso é necessário alterar a estrutura do elemento, para comportar um segundo ponteiro, para o elemento anterior na sequência da lista.

```
struct elemento {
   int dado;                  // Conteúdo (inteiro)
   struct elemento *prox;     // Ponteiro para o próximo registro
   struct elemento *ant;      // Ponteiro para o registro anterior
};
```

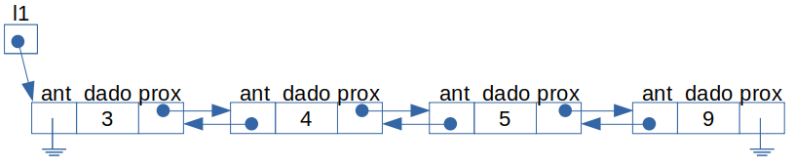
Se considerarmos a versão de lista sem o nodo cabeça, a lista vazia continua sendo um ponteiro para NULL. E após a inserção do primeiro elemento (valor 4), o desenho da lista ficaria.



E, depois de inserir os elementos 4, 9, 5 e 3 na lista duplamente encadeada, com inclusão no início da lista, o desenho ficaria



Já no caso de usarmos uma versão de lista duplamente encadeada ordenada, o resultado da inclusão da mesma sequência de valores seria



Observe que se eu estiver com um ponteiro p apontando para o elemento 5, eu consigo facilmente descobrir o endereço do próximo (`p->prox`) e do anterior (`p->ant`).

Desta forma, a rotina de exclusão de um elemento fica mais simples, pois não é necessário manter um outro ponteiro para guardar o endereço do anterior, e pode ser implementada como no exemplo abaixo. Compare o código com o retiraLista() da primeira lista encaeada.

```
// Retira o elemento e da lista l, se ele existir.
// Retorna o endereço para o início da lista.
Lista retiraLista (Lista l, int e)
{
     Lista p,    // Ponteiro p para o elemento atual

     p=l;
     // Procura o elemento e até o fim da lista ou encontrá-lo
```

```
      while ((p!=NULL) && (p->dado != e))
      {   // Procura o elemento e
           p = p->prox;
      }
      if (p!=NULL)
      {   // Encontrou o elemento e. Remove
           if (p->ant == NULL) // Removendo primeiro elemento
           {    // Atualiza o início da lista
                l = p->prox;
                free (p);
           }
           else // Não é o primeiro elemento da lista
           {
                // O próximo do anterior será o próximo do e.
                p->ant->prox = p->prox;
                if (p->prox != NULL)
                { // Não é o último elemento da lista
                     // O anterior do próximo será o anterior do e.
                     p->prox->ant = p->ant;
                }
                // Libera o elemento encontrado
                free(p);
           }
      }
      return(l);
}
```

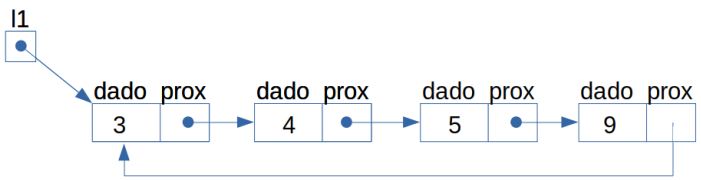
As demais operações se asemelham às operações sobre a lista com encadeamento simples, embora sempre precisando atualizar os ponteiros `prox` e `ant`, do elemento.

5. Lista encadeada circular.

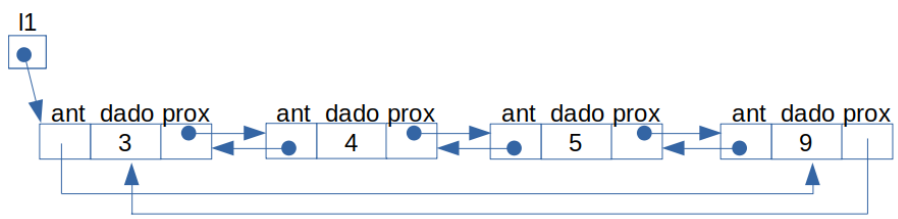
Em certas aplicações, precisamos processar os elementos de uma lista, e ao chegar ao último elemento, precisamos reiniciar o processamento a partir do primeiro elemento da lista. Uma forma de estrutura de dados que pode se prestar a esse tipo de aplicação é a lista encadeada circular.

O que muda em uma lista encadeada circular é que o campo `prox` do último elemento aponta para o primeiro elemento da lista, no lugar de apontar para `NULL`.

Um exemplo de desenho de uma lista com encadeamento simple, circular, com inserção ordenada, após a inclusão dos números 4, 9, 5 e 3, é:



E se a lista for duplament encadeada, o resultado é:



Nesse caso, as operações em que é necessário percorrer a lista (imprimir, buscar, retirar ou mesmo inserir ordenado) precisam avaliar se já foi percorrida toda a lista de um modo diferente, e não testando se o ponteiro `p` é igual a `NULL`. O teste precisa verificar se já voltou ao endereço do início da lista (`l`).

Considere o exemplo abaixo, para a operação imprimeLista().

```
// Imprime todos os nodos da lista l
void imprimeLista(Lista l)
{
     Lista p;       // Ponteiro auxiliar, para percorrer a lista

     printf("\nItens da lista\n");
     p = l;         // Fazer p apontar para o início da lista
     if (p != NULL) // Se a lista não é vazia
     {
          do {
              printf("%d - ",p->dado);  // Imprime o elemento
              p = p->prox;    // Avança para o próximo nodo
          } while (p != l) // Enquanto não voltar ao início
     }
     printf("\n");  // Avança para a próxima linha
}
```