Lista Encadeada

1. Introdução

A lista simples, que consideramos no primeiro exemplo, é útil para para armazenar informações, mas quando a quantidade de informações cresce, algumas operações podem começar a se tornar bastante lentas.

No código de exemplo, a operação para retirar um elemento da lista precisa deslocar todos os elementos a partir do elemento a ser retirado, uma unidade para cima. Caso o número de elementos da lista seja razoavelmente pequeno (por exemplo, até uma centena de itens), a operação é rápida, em especial se realizada esporadicamente, como por exemplo, se solicitada por um usuário humano. Entretanto, se a quantidade de elementos da lista for grande, na casa dos dezenas ou centenas de milhares, a operação pode passar a demorar um tempo significativo, e também consumir recursos computacionais de modo mais significativo. Se considerarmos o uso dessa estrutura em um sistema que está atendendo a centenas ou milhares de usuários, este algoritmo provavelmente se tornará inviável.

Uma outra alternativa para resolver esse problema é o conceito de lista encadeada, em que os elementos consecutivos na lista não estão necessariamente na posição de memória contígua.

Isso é implementado incluindo em cada elemento um campo que indica a posição do próximo elemento na lista. Assim, além do conteúdo (que no exemplo anterior era um string com o item a ser comprado, mas pode ser uma estrutura contendo vários dados de um aluno, por exemplo), cada elemento conterá também um campo que indica a posição (ou endereço) do próximo elemento. Vamos chamar esse campo de `prox`.

Os elementos da lista são também frequentemente chamados de células ou nodos, e a figura 1 ilustra um exemplo de um nodo com um campo conteúdo sendo um inteiro e o campo prox.

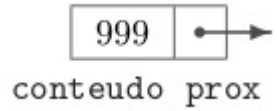


Figura 1. Exemplo de nodo de lista encadeada

É possível implementar uma lista encadeada com alocação estática, implementada como um vetor de elementos, usando, por exemplo, uma estrutura para cada elemento, incluindo o conteúdo a ser armazenado e o campo prox, que é o índice do vetor em que está o próximo elemento. A lista completa será uma estrutura com o vetor de elementos e um inteiro que indica a posição do primeiro elemento.

Para simplificar, segue um exemplo de como seria a definição da estrutura, considerando que o conteúdo a ser armazenado em cada elemento da lista seja simplesmente um número inteiro.

```
#define MaxItens 10

struct elemento {
   int conteudo;  // Conteúdo inteiro de cada elemento
   int prox;      // Posição do próximo elemento da lista
}

struct tpLista {
   struct elemento e[MaxItens]; // vetor com os elementos
   int inicio; // indica a posição do primeiro elemento
}
```

A criação do código para essa implementação é deixada como exercício.

2. Alocação dinâmica de memória.

A implementação de estruturas de dados com alocação estática possui uma limitação de capacidade de armazenamento de elementos definida no momento da programação, que pode não ser suficiente para determinada utilização, ou, em outros casos, alocar previamente uma quantidade muito grande de memória RAM sem utilização.

Em contrapartida, pode-se utilizar a alocação dinâmica de memória para esse tipo de estruturas de dados. Desta forma, a cada novo elemento inserido na estrutura é realizada a alocação da memória necessária para armazenar um nodo, o que otimiza o aproveitamento da memória RAM. Com alocação dinâmica, a quantidade de elementos que uma lista pode comportar está limitada somente à quantidade de memória disponível para a aplicação utilizar, que em última instância é limitada pela quantidade de memória disponível no computador.

A partir deste ponto, utilizaremos prioritariamente a alocação dinâmica para as próximas estruturas de dados.

A definição da estrutura de cada nodo (célula ou elemento) da lista pode seguir o exemplo a seguir, considerando que queremos armazenar números inteiros na lista.

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

É interessante tratar os nodos como um novo tipo de dados, e atribuir um novo nome a esse tipo.

```
typedef struct elemento Nodo;
```

Um nodo `n` e um ponteiro `p` para um nodo podem ser declarados assim:

```
Nodo  n;
Nodo *p;
```

Nesse exemplo, `n` é um nodo, `n.dado` é o conteúdo do nodo (o valor armazenado no nodo) e `n.prox` é o dendereço do próximo nodo. Se `p` é o endereço de um nodo, então `p->dado` é o conteúdo do nodo e `p->prox` é o endereço do próximo nodo. Se `p` é o endereço do último nodo da lista, então `p->prox` vale `NULL`.

Observe que os espaços de memória alocados dinamicamente não são acessados por meio do nome de uma variável, da mesma forma que as variáveis declaradas no código. Então para acessar os nodos de uma lista encadeada, é preciso ter uma variável declarada, com nome, para indicar a posição do primeiro elemento. Nesse exemplo poderia ser uma variável ponteiro para nodo.

```
Nodo *l1;
```

O exemplo da figura 2 representa uma lista encadeada contendo os valores 3, 5, 9 e 4.

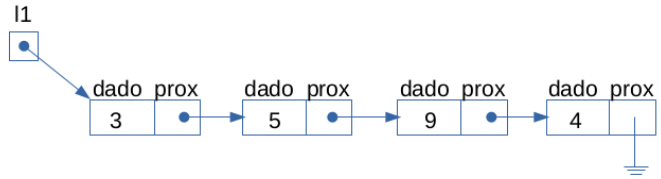


Figura 2. Lista encadeada com 4 elementos inteiros.

Observe que o campo `prox` do último nodo (com valor 4) está apontando para `NULL`, indicando o final da lista. A figura 2 pode dar a falsa impressão de que os nodos ocupam posições consecutivas na memória, mas na realidade geralmente estão espalhados de maneira totalmente imprevisível pela memória.

Neste exemplo, o endereço da lista é o endereço do primeiro nodo, indicado pelo ponteiro `l1`. A lista está vazia (ou seja, não contém nenhum nodo), se e somente se `l1 == NULL`.

Também podemos definir um tipo Lista como um ponteiro para nodo.

```
typedef Nodo *Lista;
```

ou

```
typedef struct elemento *Lista;
```

A partir dessa definição, podemos declarar as variáveis `l1, l2` e `l3` do tipo `Lista`. Isso também permite definir mais claramente que uma função retorna uma variável do tipo `Lista`.

```
Lista l1, l2, l3;
```

O comando em linguagem C para alocar espaço de memória é o `malloc`, e tem a seguinte sintaxe:

```
p = malloc(<tam>);
```

A função `malloc(<tam>)` reserva um espaço de memória com tamanho `<tam>` e retorna o endereço do espaço alocado. Esse endereço deve ser armazenado em um ponteiro, nesse exemplo, `p`.

Para liberar o espaço de memória alocado por um ponteiro é usada a função `free(<p>)`, onde `<p>` é o ponteiro que contém o endereço da área a ser liberada.

3. Operações com a lista encadeada.

A lista encadeada com alocação dinâmica contém as mesmas operações definidas sobre uma lista com alocação estática, apenas mudando a forma como cada operação é implementada.

- Criar uma lista vazia;
- Inserir um elemento na lista;
- Excluir um elemento da lista;

- Contar o número de elementos na lista;
- Exibir todos os elementos da lista;
- Buscar um elemento na lista.

3.1. Criação de uma lista.

A criação de uma lista precisa produzir uma lista vazia, o que neste caso significa um ponteiro do tipo `Lista`, apontando para `NULL`. Isso pode ser implementado por uma função que retorna um ponteiro nulo.

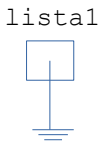
```
Lista criaLista()
{
      return NULL;
}
```

Para chamar essa função, o retorno da função `criaLista()` deve ser armazenado em uma variável do tipo `Lista`, a exemplo das linhas a seguir.

```
Lista lista1;                 // Declaração da variável lista1

lista1 = criaLista();         // Inicialização da lista1 (lista vazia)
```

Observe que após a execução dessas linhas a lista `lista1` contém uma lista vazia, ou seja, `lista1` contém o valor `NULL`. A representação gráfica é:



3.2. Exibição do conteúdo de uma lista.

Para imprimir o conteúdo de uma lista é necessário percorrer toda a lista, sequencialmente, do primeiro elemento até o final da lista. Isso pode ser implementado pelo código abaixo.

```
// 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
      while (p != NULL) // Enquanto não chegar ao final da lista
      {
            printf("%d - ",p->dado);      // Imprime o elemento
            p = p->prox;      // Avança o ponteiro para o próximo nodo
      }
      printf("\n");     // Avança para a próxima linha
}
```

3.3. Contar o número de elementos de uma lista.

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. 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;		// 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
}
```

3.4. Buscar um elemento na lista.

Uma possível necessidade é descobrir se um elemento está na lista. Caso o elemento esteja na lista, uma opção é retornar o endereço do nodo, e caso o elemento não exista na lista, o retorno é NULL.

```
// Procura o elemento e na lista l, retornando o endereço
// do elemento se ele estiver na lista, ou NULL em caso contrário
Lista buscaLista(Lista l, int e)
{
	Lista p;	// Ponteiro para percorrer a lista

	p = l;	// Faz o ponteiro apontar para o início da lista
	// Percorre a lista enquanto não chegar ao final da lista
	// e não encontrar o elemento
	while ((p != NULL) && (p->dado != e))
	{
		p = p->prox;
	}
	return (p); // Retorna o endereço do nodo encontrado
		    // ou NULL caso tenha chegado ao final sem
		    // encontrar o elemento e na lista.
}
```

3.5. Inserir um elemento na lista.

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. Existem três maneiras clássicas de escolher o ponto da lista onde o novo nodo deve ser conectado. No início da lista, ao final da lista ou respeitando algum critério de ordem na lista (por exemplo, ordem crescente ou decrescente), para algum campo do conteúdo usado para classificação.

O modo mais simples é a **inserção no início da lista**.

```
// Insere um elemento e no início da lista l
Lista insereLista(Lista l, int e)
{
      Lista novo;

      // Aloca o espaço e faz as atribuições de valores
      novo = malloc(sizeof(struct elemento));
      novo -> dado = e;
      novo -> prox = l;  // O próximo do novo é o início da lista
      return (novo);     // Retorna o endereço do novo elemento
}
```

Essa função sempre inclui um novo elemento no início da lista, e portanto, altera o início da lista. O endereço do novo elemento é retornado, e o início da lista precisa ser atualizado com o novo endereço. Nesse caso, considere que no programa principal tenhamos uma lista `l1` e queremos inserir um dado que está na variável inteira `valor`. A chamada da função pode ser:

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

Para **inserir um elemento ao final da lista**, antes de conectar o novo nodo à lista, é preciso procurar a posição de inserção (no final da lista). Para isso uma forma é fazer um laço até encontrar um ponteiro para NULL, lembrando de manter o endereço do último elemento para conseguir fazer a conexão.

Já a função de **inserção de um elemento em uma lista ordenada** precisa buscar a posição de inserção considerando o critério de ordem definido (por exemplo, ordem crescente ou decrescente), comparando o identificador do elemento a ser inserido com os identificadores de cada elemento até encontrar a posição de inserção. Obviamente também é preciso testar no laço, se ainda não chegou ao final da lista.

Observe que o único ponto que precisa ser alterado para obter uma lista ordenada é a função que insere os elementos na lista. Se todos os elementos forem inseridos em ordem, automaticamente a lista permanecerá sempre ordenada. Entretanto, o modo de inserção em uma lista deve ser sempre o mesmo em um programa, de modo a garantir que os elementos sempre permaneçam na ordem definida. Assim, ao implementar uma lista encadeada ordenada, a única função de inserção deve ser a que insere o elemento na ordem definida.

A implementação da lista encadeada com iserção ao final e de lista encadeada ordenada são deixadas como exercícios.

3.6. Retirar um elemento na lista.

Para retirar um elemento na lista dinamicamente encadeada é necessário: 1) localizar o endereço do elemento na lista, e, se encontrado: 2) conectar elemento anterior ao próximo e 3) liberar o espaço de memória ocupado pelo elemento retirado. O comando `free(x)` é usado para liberar o espaço de memória apontado pelo ponteiro `x`

```
// 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
           ant;  // Ponteiro ant para o elemento anterior

      p=l;
      ant=l;
      // Procura o elemento e até o fim da lista ou encontrá-lo
      while ((p!=NULL) && (p->dado != e))
      {   // Procura o elemento e
            ant = p;
            p = p->prox;
      }
      if (p!=NULL)
      {   // Encontrou o elemento e. Remove
            if (p == ant) // Removendo primeiro elemento
            {
                l = p->prox;
                free (p);
            }
            else // Não é o primeiro elemento da lista
            {
                  ant->prox = p->prox;
                free(p);
            }
      }
      return(l);
}
```