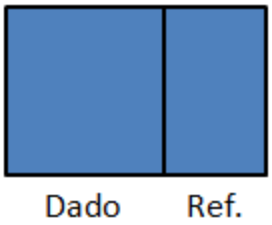
Lista Encadeada (Linked List)

As listas ou listas encadeadas são a estrutura de dados mais simples concebível excetuando-se naturalmente os arrays. Listas encadeadas nada mais são que uma seqüência de células ligadas ou encadeadas umas as outras. As células de uma lista encadeada são compostas de dois elementos cada. O primeiro elemento é o dado efetivo a ser armazenado e o segundo compraz uma referência para o próximo elemento da lista. A figura abaixo apresenta um modelo esquemático da célula de uma lista encadeada assim como a sua estrutura de dados.




```
struct celula{
    int conteudo;
    struct celula *prox;
};
```

Quando lidamos com arrays, devemos saber de antemão ou em tempo de compilação qual o tamanho ou quantidade de bytes a ser alocados para o array. Como foi visto em sala de aulas via exemplos, existem muitas situações em programação onde a quantidade exata de memória a ser alocada não é conhecida na criação do programa devendo ser definida durante sua execução. Para estes casos pode-se alocar a memória necessária via alocação dinâmica de memória. No entanto ainda existem outros casos em que mais memória deve ser alocada para armazenar os dados do programa a medida que o programa vai sendo utilizado. Este é um dos casos em que a utilização de listas ligadas é mais indicada do que a utilização de arrays estáticos ou dinâmicos. Outra vantagem das listas ligadas é que não existe nenhuma restrição indicando que as células que as compõem devem estar alocadas seqüencialmente na memória tal como ocorre com arrays ou memória alocada dinamicamente via malloc. Em verdade, cada célula pode existir em diferentes regiões a memória.

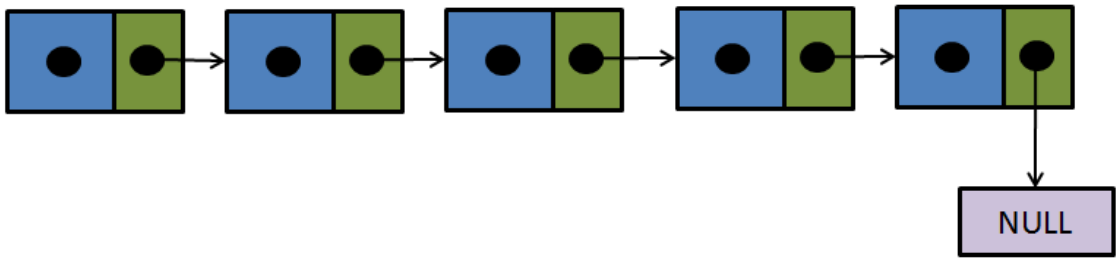


Figura 1 – representação gráfica de uma lista ligada

A lista ligada é composta, como foi dito anteriormente, por células de informação. Cada célula contém um item de dado e um ponteiro para a próxima célula da lista. O final da lista é marcado pela última célula da lista apontando para o ponteiro nulo (NULL).

Uma lista ligada vazia nada mais é que um ponteiro apontando para NULL.

Por fim, pode-se imaginar que o requisito imposto de um ponteiro apontando para o próximo elemento de cada elemento da lista seja um uso desnecessário de memória. No entanto vale lembrar que em geral as itens armazenados em uma lista encadeada são muito maiores que os tipos atômicos de dados, tornando assim a proporção entre memória de dados e ponteiros relativamente negligenciável.

Listas com e sem cabeçalho

Listas ligadas podem ser organizadas de duas maneiras distintas.

Lista *com* cabeçalho: O conteúdo da primeira célula não importa, marca apenas o início da lista. A primeira célula está sempre no mesmo lugar na memória, mesmo que a lista fique vazia.Digamos que LISTA é o endereço da primeira célula. Então LISTA->prox == NULL se e somente se a lista está vazia. Para criar uma lista vazia, basta dizer

```
celula c,*LISTA;
c.prox = NULL;
LISTA = &c;
```

ou

```
celula *LISTA;
LISTA = malloc (sizeof (celula));
LISTA->prox = NULL;
```

Lista *sem* cabeçalho: O conteúdo da primeira célula é tão relevante quanto o das demais. Nesse caso, a lista está vazia se o endereço de sua primeira célula é NULL. Para criar uma lista vazia basta fazer

- `celula *LISTA;`
- `LISTA = NULL;`

Itens podem ser inseridos em qualquer ponto da lista ligada:

- Início;
- Meio;
- Fim;

Inserção no Início da lista

Na inserção de elementos no início da lista deve-se observar se a mesma possui ou não um cabeçalho. Para o caso em que a lista possua cabeçalho, deve-se proceder da seguinte forma:

1. Criar a nova célula a ser inserida;
2. Apontar o ponteiro “prox” da célula sendo inserida para a primeira célula da lista original (NEW_CEL->prox = HEAD->prox);
3. Apontar o ponteiro “prox” do cabeçalho para a célula a ser inserida HEAD->prox = NEW_CEL->prox.

Exercício: Como deve se proceder a inserção de células no início de listas sem cabeçalho?

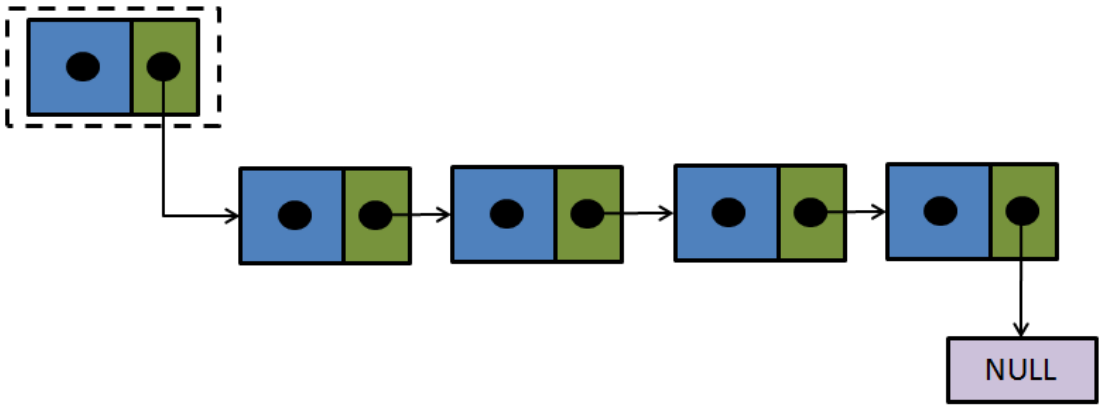


Figura 2 Inserção no início de uma lista sem cabeçalho

```
void insereInicio (celula *head, celula *cel)
{
  cel->prox  = head->prox;
  head->prox = cel->prox;
}
```

Inserção no Méio da lista

A inserção de um elemento no meio da lista ou em uma posição arbitrária independe do fato da lista possuir ou não cabeçalho. A idéia geral é encontrar a posição na qual a célula corrente deve ser inserida e então identificar as células anterior e posterior. Uma vez identificadas tais células basta que os ponteiros sejam ajustados.

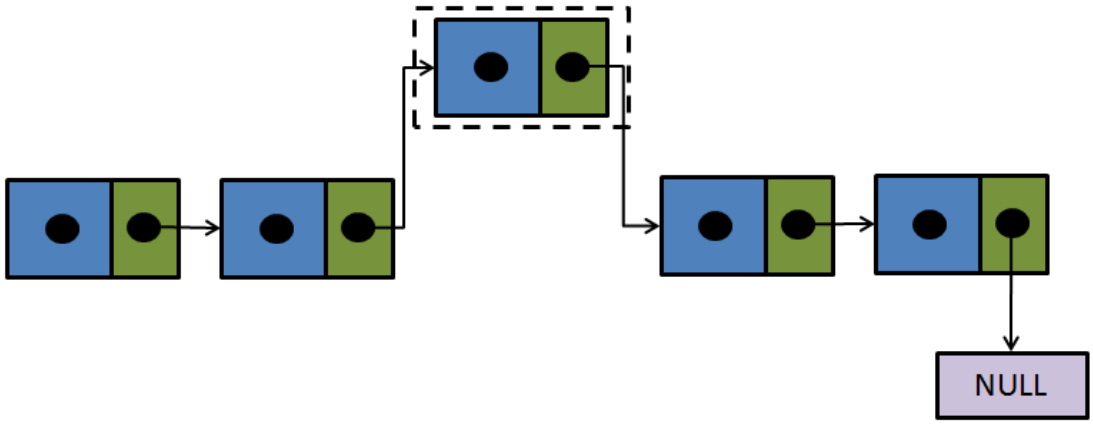


Figura 3 Inserção no meio de uma lista sem cabeçalho

```
void inserePosicao (celula *head, celula *cel, int pos)
{
  //itera até encontrar a posição certa
  //testa se a posição indicada é valida
  //EXERCICIO: escrever a atribuição dos ponteiros prox da célula
  //  anterior e próxima a posição em que a célula deve ser inserida

}
```

Inserção no Fim da lista

Tal inserção ocorre de maneira muito similar ao caso anterior. A única diferença é que célula->prox deve apontar para NULL.

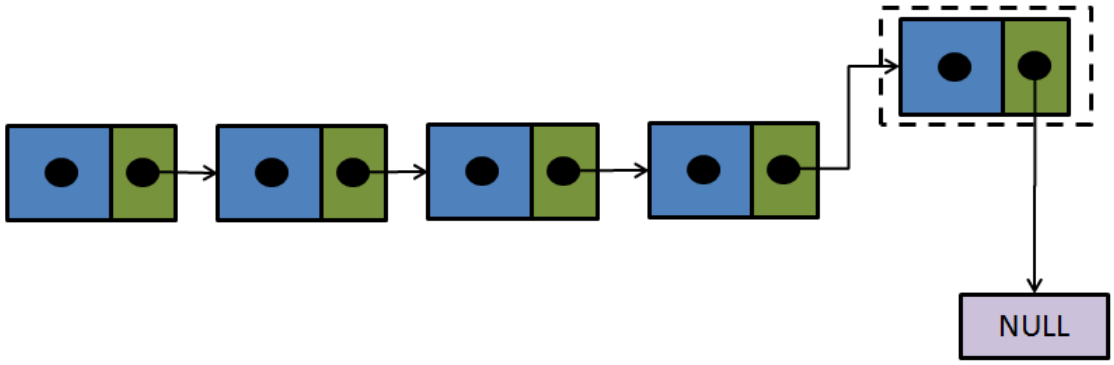


Figura 4 Inserção no final de uma lista sem cabeçalho

Remoção de Elementos de uma Lista

Os elementos de uma lista podem ser removidos, a semelhança das regras de inserção, de três maneiras distintas:

a) Remoção do primeiro elemento da lista;
b) Remoção do último elemento da lista;
c) Remoção de um elemento em uma posição arbitrária.

As funções de remoção são muito similares as de inserção e, portanto ficam como exercício.

Compare a remoção de um item de uma lista encadeada com a remoção de um item de um array!

Busca em Listas Encadeadas

Embora existam vantagens obvias relativas à utilização de listas encadeadas para gerência de memória alocada e inserção e remoção de itens, listas encadeadas não podem ser indexadas tal como um array. Existem todavia diversas formas de acessar os elementos de uma lista encadeada. As duas mais relevantes são:

a) Busca por um dado elemento – em que um elemento de dado "elem" é especificado. A lista deve ser percorrida iterativamente até que o elemento seja encontrado;
b) Recuperação do elemento N a partir do início da lista – A lista é percorrida a partir do início até que o elemento N é encontrado.

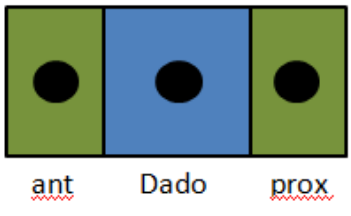
```
celula *buscaElemento (int elem, celula *HEAD)
{
   celula *p;
   p = HEAD->prox;
   while (p != NULL && p->conteudo != elem)
      p = p->prox;
   return p; }
```

EXERCÍCIO: implemente a recuperação do elemento N a partir do início da lista.

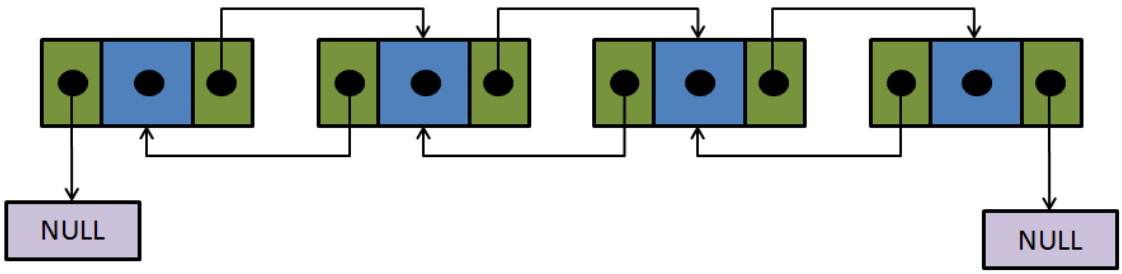
A partir da idéia geral de listas encadeadas, estruturas de dados variantes podem ser concebidas. Em especial duas estruturas são de especial interesse. As listas duplamente encadeadas e as listas circulares.

Listas Duplamente Encadeadas

Uma lista duplamente encadeada é assim denominada pela forma como as células que compõem a lista são ligadas. Ao contrário das listas simples, existem dois caminhos possíveis para se percorrer a lista. Pode-se pesquisar a lista duplamente encadeada em sentido normal e em sentido inverso. Isso se da pelo fato de que os elementos da lista possuem dois ponteiros, um que aponta para a próxima célula da posta e outro que aponta para a célula anterior. Adicionalmente, a primeira célula terá seu ponteiro "ant" apontando para NULL e a última célula terá seu ponteiro "prox" apontando igualmente para NULL.

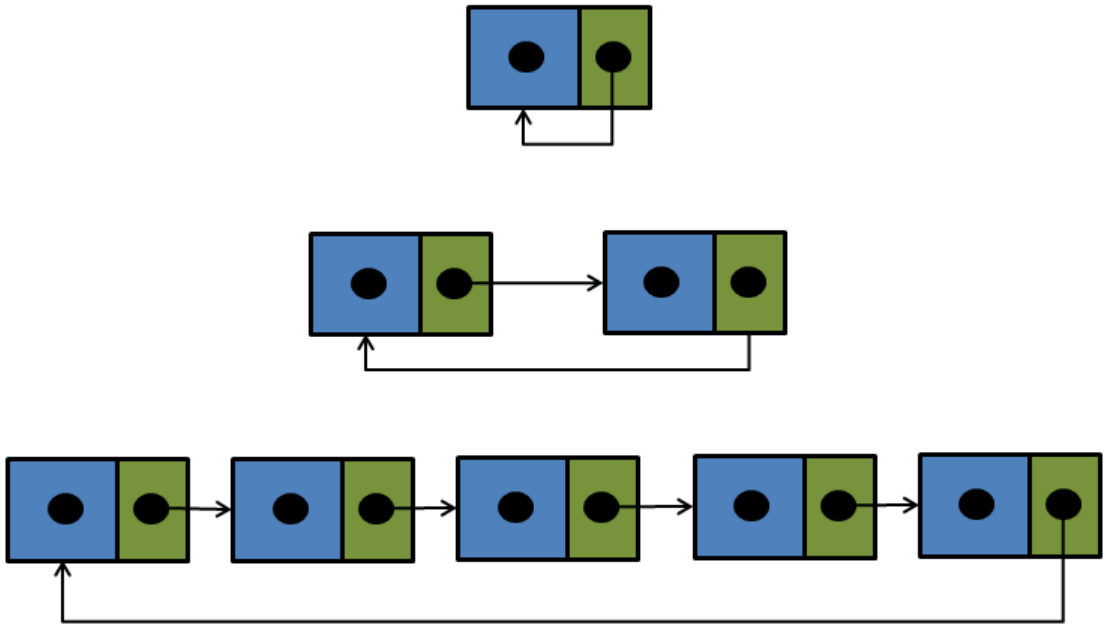



```
struct celula{
    struct celula *ant;
    int conteudo;
    struct celula *prox;
};
```



Listas Circulares

Listas circulares são listas encadeadas simples com a diferença de que elas não possuem um começo e um fim. Um ponteiro externo aponta para uma célula qualquer da lista. Itens são inseridos ou removidos da lista a partir da posição atual do ponteiro externo.



Exercícios

1. Crie uma função para imprimir todos os elementos de uma lista encadeada com e sem cabeçalho;
2. Escreva uma função que receba uma lista encadeada e devolva o endereço de um nó que esteja o mais próximo possível do meio da lista. Faça isso sem contar explicitamente o número de nós da lista;
3. Compile e execute o seguinte programa:

```
typedef struc cel celula;
struct cel {
   int     conteudo;
   celula *prox;
};
int main (void) {
   printf ("sizeof (celula) = %d\n", sizeof (celula));
   return 0;
}
```

4. Por que a seguinte versão de `insere` não funciona?

```
void insere (int x, celula *p) {
   celula nova;
   nova.conteudo = x;
   nova.prox = p->prox;
   p->prox = &nova; }
```

5. Escreva uma função que insira um novo elemento em uma lista encadeada *sem* cabeça. Será preciso tomar algumas decisões de projeto antes de começar a programar.
6. Critique a seguinte versão da função `remove`:

```
void remove (celula *p, celula *ini) {
   celula *morta;
   morta = p->prox;
   if (morta->prox == NULL)  p->prox = NULL;
   else  p->prox = morta->prox;
   free (morta); }
```

7. Invente um jeito de remover uma célula de uma lista encadeada *sem* cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)
8. Escreva uma função que copie um vetor para uma lista encadeada. Faça duas versões: uma iterativa e uma recursiva.
9. Escreva uma função que copie uma lista encadeada para um vetor. Faça duas versões: uma iterativa e uma recursiva.
10. Escreva uma função que faça uma *cópia* de uma lista dada.
11. Escreva uma função que *concatena* duas listas encadeadas (isto é, "amarra" a segunda no fim da primeira).
12. Escreva uma função que *conta* o número de células de uma lista encadeada.
13. Escreva uma função que remove a `k`-ésima célula de uma lista encadeada sem cabeça. Escreva uma função que insere na lista uma nova célula com conteúdo `x` entre a `k`-ésima e a `k+1`-ésima células.

14. Escreva uma função que verifica se duas listas dadas são *iguais*, ou melhor, se têm o mesmo conteúdo. Faça duas versões: uma iterativa e uma recursiva.
15. Escreva uma função que *desaloca* (função `free`) todos os nós de uma lista encadeada. Estamos supondo, é claro, que cada nó da lista foi originalmente alocado por `malloc`.
16. Escreva uma função que *inverte* a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem usar espaço auxiliar; apenas altere os ponteiros. Dê duas soluções: uma iterativa e uma recursiva.
17. Projeto de Programação. Digamos que um *texto* é um vetor de caracteres contendo apenas letras, espaços e sinais de pontuação. Digamos que uma *palavra* é um segmento maximal de texto que consiste apenas de letras. Escreva uma função que recebe um texto e imprime uma relação de todas as palavras que ocorrem no texto juntamente com o número de ocorrências de cada palavra.
18. Descreva, em linguagem C, a estrutura de uma das célula de uma lista duplamente encadeada.
19. Escreva uma função que remove de uma lista duplamente encadeada a célula apontada por `p`. (Que dados sua função recebe? Que coisa devolve?)
20. Escreva uma função que insira em uma lista duplamente encadeada, logo após a célula apontada por `p`, uma nova célula com conteúdo `y`. (Que dados sua função recebe? Que coisa devolve?)
21. Problema de Josephus. Imagine que temos *n* pessoas dispostas em círculo. Suponha que as pessoas estão numeradas `1` a *n* no sentido horário. Começando com a pessoa de número `1`, percorra o círculo no sentido horário e elimine cada `m`-ésima pessoa enquanto o círculo tiver duas ou mais pessoas. Qual o número do sobrevivente?
22.