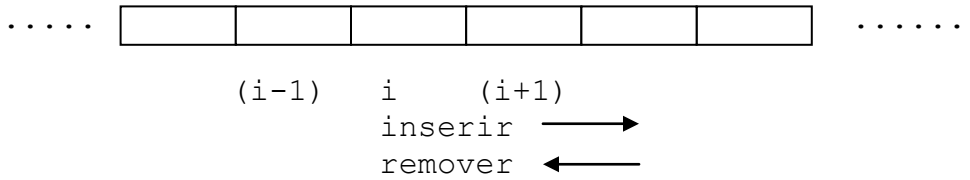
Alocação Dinâmica – Listas Ligadas

1. Listas sequenciais versus listas ligadas

Lista sequencial

Uma lista sequencial é um conjunto de elementos contíguos na memória. Um vetor é o melhor exemplo de lista sequencial. O elemento `i` é precedido pelo elemento `i-1` e seguido pelo elemento `i+1`. Como os elementos são contíguos, para se acessar um elemento só é necessário conhecer o seu índice, pois o índice indica também o deslocamento a partir do início. Claro que para isso, os elementos devem ter o mesmo tamanho.

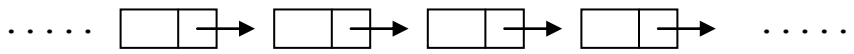
A vantagem é o acesso direto a qualquer dos elementos da lista. A desvantagem é que para inserir ou remover elementos, temos que deslocar muitos outros elementos.



Listas Ligada

Uma lista ligada é um conjunto de elementos onde cada elemento indica qual o próximo. Não existe contiguidade entre os elementos. Elementos vizinhos podem estar em posições físicas de memória separadas.

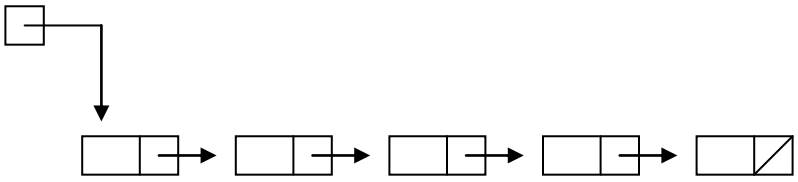
A vantagem é a flexibilidade na inserção e remoção de elementos. A desvantagem é que não temos acesso direto aos elementos. Não existe algo equivalente ao índice para se acessar diretamente o elemento.



2. Listas ligadas - definições

É um conjunto de itens, onde cada elemento contém uma informação e um ponteiro para o próximo item.
Possui também um ponteiro para o seu início e o ponteiro do último elemento tem um valor especial `(NULL)`.

P (inicio)



O campo de informação pode conter qualquer tipo ou conjunto de tipos de elementos. O ponteiro para o próximo item é um ponteiro para um item do mesmo tipo.

Exemplos:

a) campo de informação = `int`:

```
struct item {
  int info;
  struct item *prox;
}
```

b) campo de informação = `2 ints e 1 double`:

```
struct novoitem {
  int i, j;
  double a;
  struct novoitem *prox;
}
```

c) campo de informação = `vetor de char`:

```
struct no {
   char s[30];
   struct no *prox
}
```

d) campo de informação = `struct com 2 ints e 1 double`:

```
struct info {
   int i, j;
   double a;
}

struct novono {
  struct info x;
  struct novono *prox;
}
```

Observe que a definição do tipo de um elemento de uma lista é recursiva. Veja no primeiro exemplo: usamos na definição de `struct int` o próprio `struct item`. O compilador C permite.

Considere ainda o primeiro exemplo. Outra maneira de fazer a mesma definição é usar o `typedef` que é apenas um modo de atribuirmos um sinônimo (em geral um nome mais simples) a um tipo (em geral mais complexo) na linguagem C. Fica um pouco mais elegante.

```
/* cria sinomimo para struct item * */
typedef struct item *link;

/* declara a struct item */
struct item {
  int info;
  link prox;
}

/* declara um ponteiro para um elemento */
link p;
```

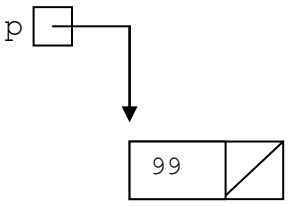
Refaça os exemplos acima usando typedef.

Nos exemplos a seguir usaremos sempre esta última definição de `struct item`, com o campo `info` do tipo `int`. Porém, fica claro que `info` pode conter qualquer tipo de informação.

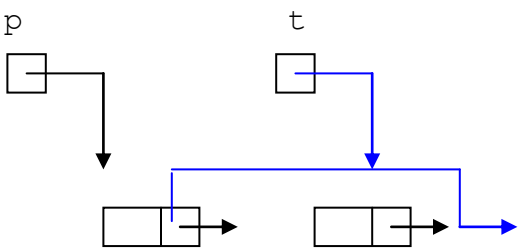
3. Operações básicas com listas ligadas

Vejamos agora, algumas operações com listas ligadas, usando a declaração acima:

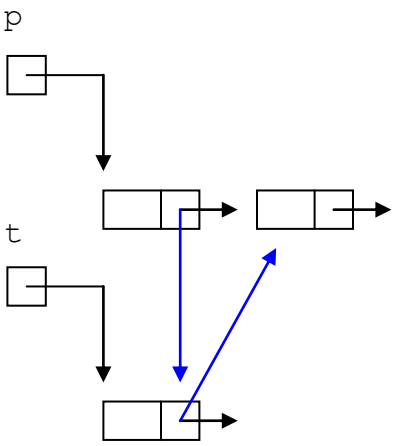
```
/* criar um lista apontada por p com o primeiro elemento */
p = (link) malloc (sizeof(struct item));
p->info = 99;
p->prox = NULL;
```




```
/* remover o nó seguinte ao apontado por p */
t = p->prox;
p->prox = t->prox;
free(t);
```




```
/*inserir um novo nó(apontado por t)após nó apontado por p*/
t->prox = p->prox;
p->prox = t;
```




```
/* função que procura elemento com info=x numa lista ligada
   devolve ponteiro para elemento encontrado ou NULL */
link busca(link t, int x) {
   link q;
   q = t;
   while (q != NULL) {
       if (q->info == x) return q;
       q = q->prox;
   }
   return NULL; /* não encontrou */
}

/* Dado um um trecho de um vetor a[l..r] de inteiros, criar
   lista ligada onde a[l] é o primeiro elemento, a[l+1] o
   segundo, etc... até a[r] o último elemento.
   Devolve ponteiro para o início da lista */
link crialista(int a[], int l, int r) {
  link q=NULL; int i; link t;
  /* varre o vetor no sentido inverso */
  for (i=r; i >=l; i--) {
```

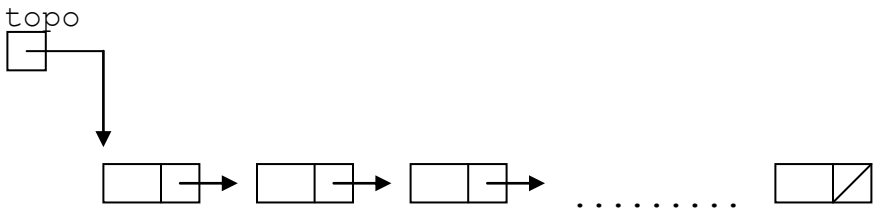
```
      t = (link) malloc (sizeof(struct item));
      t->info = a[i];
      t->prox = q; /* aponta para o anterior */
      q = t;  /* novo anterior */
   }
   return q;
 }
```

Na solução acima a lista é varrida do fim para o começo.
Exercício – fazer o mesmo, varrendo a lista do início para o fim.

4. Pilhas e filas – Alocação Dinâmica

Pilhas e filas são estruturas de dados básicas que podem ser construidas como listas ligadas.

Pilha (LIFO - Last In First Out)



Empilhar - inserir elemento com conteúdo **x** antes do primeiro (apontado por `topo`).

```
link topo; /* variável global */

void EmPilha(int x) {
   link t = (link) malloc (sizeof(struct item));
   t->info = x;
   t->prox = topo;
   topo = t;
}
```

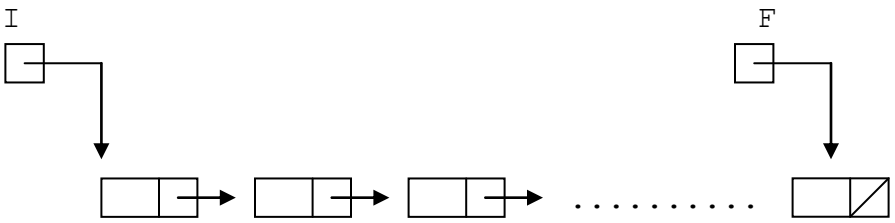
Desempilhar - retirar elemento apontado por `topo`. Retorna -1 se a pilha vazia ou 0 se a remoção for normal.

```
int DesEmPilha(int * x) {
   link t;
   if (topo == NULL) return -1; /* pilha vazia */
   else {*x = topo->info;
         t = topo;
         topo = topo -> prox; /* novo topo */
         free(t); /* libera elemento removido */
         return 0;
        }
```

```
}
```

Fila (FIFO - First In First Out)

Tem 2 apontadores (`I`- início e `F`- fim).



Enfilar - Inserir elemento após o último (apontado por `F`). Se a fila estiver vazia, I e F são NULL.

```
link I,F; /* globais */

void EnFila(int x) {
   link t = (link) malloc (sizeof(struct item));
   t->prox = NULL;
   t->info = x;
   if (F == NULL) I = F = t; /* a fila estava vazia */
   else {F->prox = t; F = t;}
}
```

Desenfilar - Remover o primeiro elemento (apontado por `I`). Devolver -1 se a fila estava vazia, senão devolve 0.

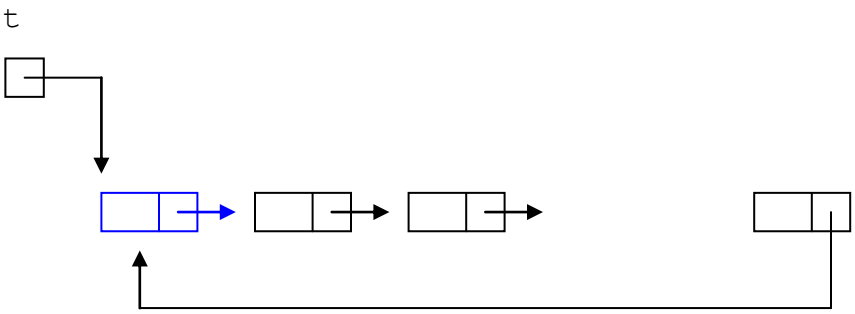
```
int DesEnFila(int *x) {
   if (I == NULL) return -1; /* a fila está vazia */
   else {*x = I->info;
         t = I;
         I = I->prox;
         if (I == NULL) F = NULL; /* a fila ficou vazia */
         free(t); /* libera elemento removido */
         return 0;
        }
}
```

5. Listas Circulares e duplamente ligadas

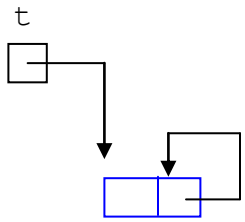
Circulares

O último item aponta para o primeiro. É conveniente o uso de nó especial (sentinela) para determinar o início e fim.

Numa lista circular, pode-se acessar todos os elementos a partir de qualquer elemento.



Quando a lista está vazia:

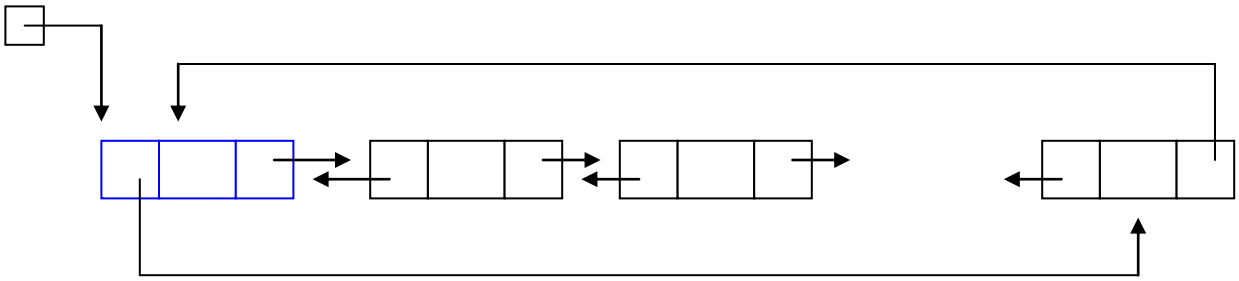



```
/* busca elemento de info=x - lista circular com sentinela */
link busca(link t, int x) {
  link p;
  p = t->prox;
  t->info = x; /* o primeiro item é a sentinela */
  while (p->info != x) p = p->prox;
  /* verifica se achou */
  if (p == t) return NULL; /*sentinela - então não achou */
  return p; /* achou */
}


/* idem mas devolvendo ponteiro ao elemento anterior */
link busca(linkt, int x) {
  link p, pant;
  pant = t; p = t->prox;
  t->info = x; /* o primeiro item é a sentinela */
  while (p->info != x) {
     pant = p; p = p->prox;
  }
  /* verifica se achou */
  if (p == t) return NULL;/*achou sentinela então não achou*/
  return pant; /* achou */
}
```

Duplamente ligadas

Gastam mais espaço, mas a mobilidade é maior (é possível percorrer a lista nos dois sentidos). O nó sentinela também é bastante útil.

```
typedef struct item *link;
struct item {int info; link ant, prox;}

t
```



Para se remover um elemento de uma lista ligada simples, é necessário o acesso ao elemento anterior. Isso não acontece numa lista duplamente ligada. Com um ponteiro para um dos elementos pode-se:

a) Remover este elemento
b) Inserir elemento anterior ou posterior a este.

6. Listas ligadas e recursão

Listas ligadas simples (não circulares e sem sentinela) são estruturas claramente recursivas. Por isso, admitem também algoritmos recursivos. Cada nó de uma lista ligada pode ser entendido como um ponteiro para uma nova lista ligada. Vejamos uma definição informal:

Definição:

Lista Ligada = {
NULL ou
Elemento contendo um campo de informação e um ponteiro para uma **Lista Ligada**
}

Algumas funções recursivas com lista ligada

```
/* função conta(x) que conta o número de elementos de uma
   lista apontada por x */
int conta(link x) {
  if (x == NULL) return 0;
  return 1 + conta(x->prox)
}

/* função procura(x,v) que procura elemento com info = v
```

```
   na lista apontada por x */
link procura(link x, int v) {
  if (x == NULL) return NULL;
  if (x->info == v) return x;
  return procura(x->prox, v);
}

/* Função compara(x, y) que compara o conteúdo (campo info)
   de duas listas ligadas apontas por x e y respectivamente.
   Devolve 0 se forem iguais, 1 se x>y e -1 se x<y. */
int compara(link x, link y) {
  if (x == NULL && y == NULL) return 0;
  if (x->info == y->info) return compara(x->prox, y->prox);
  if (x->info > y->info) return 1;
  return -1;
}

/* função remove(x,v) que remove item com info = v da lista
   apontada por x. Solução um pouco artificial */
link remove(link x, int v) {
  if (x == NULL) return NULL;
  if (x->info == v) {
      link t;
      /* retorna o próximo elemento da lista */
      t = x->prox;
      free(x);
      return t;
  }
  x->prox = remove(x->prox, v);
  return x;
 }
```

7. A vantagem das listas ligadas sobre as listas sequenciais

Não resta dúvida que mexer com uma lista ligada é mais complexo que mexer com lista sequencial.
Qual a vantagem afinal?
Numa lista ligada estamos usando exatamente a quantidade necessária de elementos para a lista num determinado instante. O número de elementos é em geral só conhecido durante a execução do programa. A cada elemento adicionado à lista, cria-se apenas mais um elemento com o `malloc`.
Numa lista sequencial, o vetor tem que ser declarado ou alocado (**`malloc`**), com um número máximo de elementos. Isso nos leva sempre a dimensioná-lo para o pior caso. Mesmo quando sabemos o número exato de elementos e o dimensionamos assim, o problema de adicionar ou eliminar elementos não fica resolvido.

Exercícios:

Lista ligada simples

`1) int conta(link p)`
Devolve a quantidade de elementos da lista apontada por p.

`2) int conta (link p, info x)`
Devolve a quantidade de elementos com o campo de informação igual a x da lista apontada por p.

`3) link remove (link p, info x)`
Remove da lista apontada por p, o primeiro elemento com o campo de informação igual a x. Devolve ponteiro para o início da lista. Note que se for o primeiro elemento a ser removido, o início da lista muda.
Obs: neste caso, ao procurar elemento com informação igual a x, é necessário sempre manter um ponteiro para o anterior a ele. Cuidado com os casos particulares: lista vazia, lista com um só elemento ou remoção do primeiro elemento.

`4) link removetodos (link p, info x)`
Remove da lista apontada por p, todos os elementos com o campo de informação igual a x.
Obs: mesmas observações anteriores.

`5) void remove (link * p, info x)`
Idem à `remove` acima. Se for removido o primeiro elemento altera o ponteiro para o início da lista. Por isso, o parâmetro p é ponteiro de ponteiro.

`6) void removetodos (link * p, info x)`
Idem à `removetodos` acima. Se for removido o primeiro elemento altera o ponteiro para o início da lista. Por isso, o parâmetro p é ponteiro de ponteiro.

`7) void troca (link p, link q)`
Troca o conteúdo dos nós apontados por p e q numa lista ligada.

Listas ligadas circulares (com sentinela t)

`8) int procura (link p)`
Verifica se p pertence a lista, ou seja, se é um dos elementos da lista.

`9) void transforma(link p)`
Transforma a lista numa lista ligada não circular

Listas duplamente ligadas (com sentinela t)

`10)void remove (link p)`
Remove elemento apontado por p da lista

11)`void insere (link p, info x)`
Insere elemento apontado por p à frente do elemento cujo campo de informação é igual a x