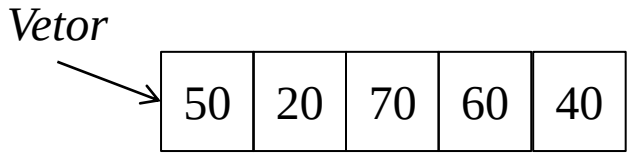
Programação 2

Listas

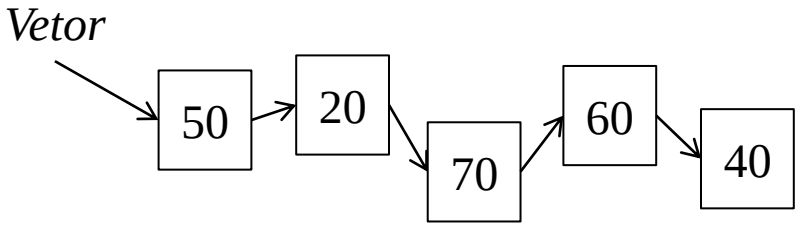
Rivera

Motivação

- Vetor
  - Ocupa um espaço contíguo de memória
  - Permite acesso randômico aos elementos
  - Deve ser dimensionado com um número máximo de elementos

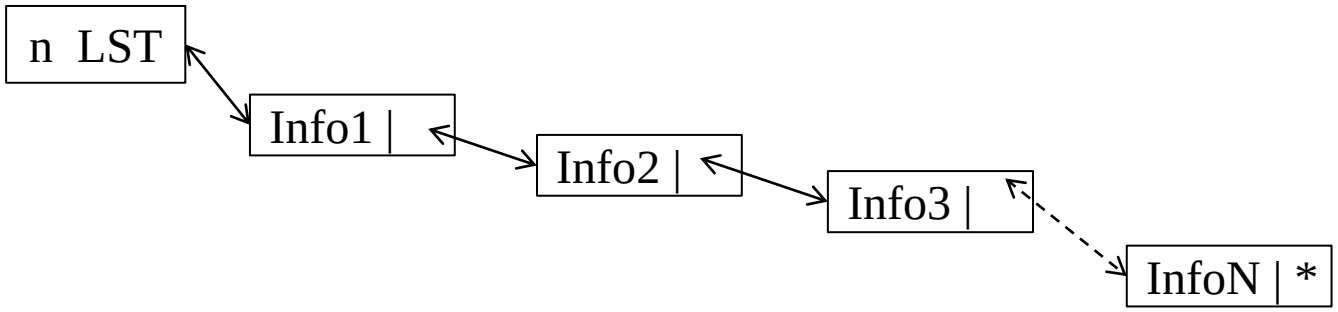


- Vetor como lista?



Motivação

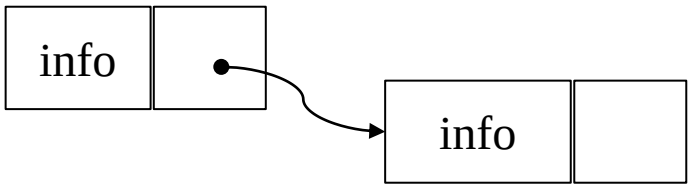
- Estrutura de dados dinâmicas
  - Crescem (ou decrescem) à medida que elementos são inseridos (ou removidos)
  - Ex.
    - Listas encadeadas
      - Amplamente usadas para implementar outras estruturas de dados

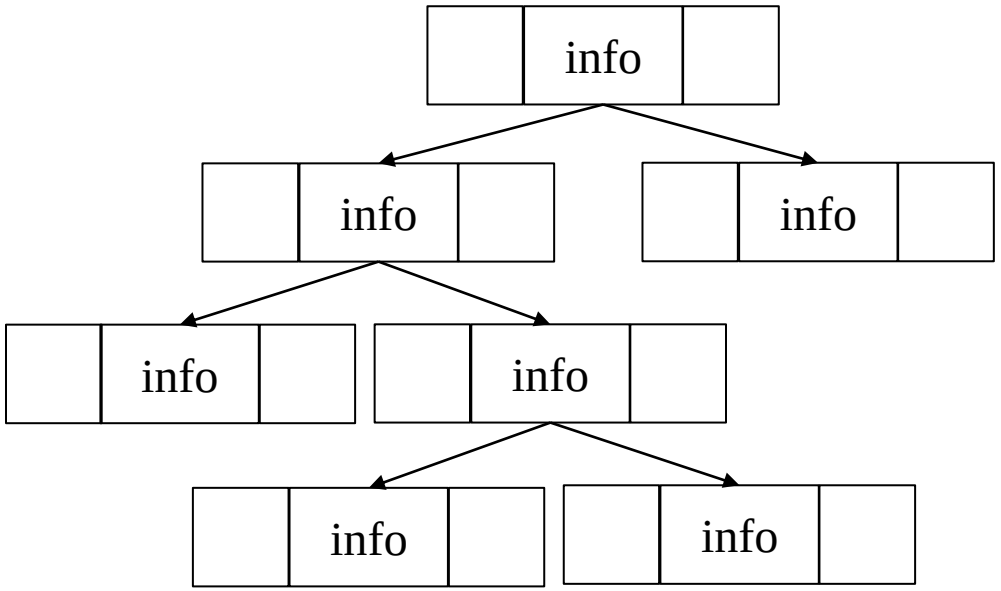


Estrutura

```
typedef struct stElem
{
   int  info;
   struct stElem* prox;
} Elem;
```

```
typedef struct stElem
{
   int  info;
   struct stElem* Esq;
   struct stElem* Dir;
} Elem;
```

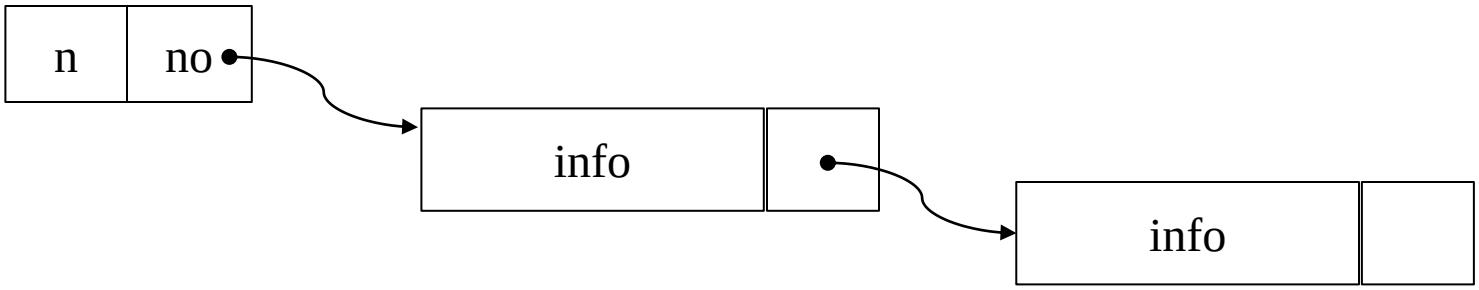




Estrutura (Raiz e Nós)

```
typedef struct stElem
{
   int  info;
   struct stElem* prox;
} Elem;
```

```
typedef struct stRaizLst
{
   int  n;
   Elem* no;
} RaizLst;
```



Listas Encadeadas: exemplo

```
typedef struct stElem
{
   int  info;
   struct stElem* prox;
} Elem;
```

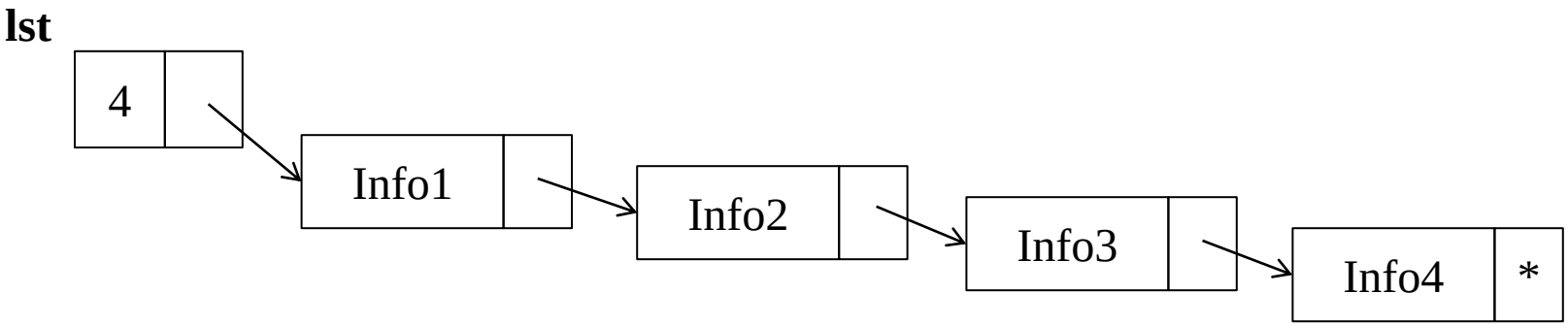
```
typedef struct stLst
{
   int  n;
   Elem* no;
} Lst;
```

```
Elemento* lst_insere (Elem* lst, int val)
{
    Elem* novo = (Elem*) malloc(sizeof(Elem));
    novo->info = val;
    novo->prox = lst->no;
    lst->n +=1;
    return novo;
}
```

```
int main (void)
{
    Lst* lst;             // declara uma lista não inicializada
    lst = lst_cria();     // cria e inicializa lista como vazia
    lst->no= lst_insere(lst, 23); // insere 23 na lista
    lst->no = lst_insere(lst, 45); // insere 5 na lista
    ...
    return 0;
}
```

Listas Encadeadas: impressão

```
/* função imprime: imprime valores dos elementos */

void lst_imprime (Elem* lst)
{
    Elem* p;
    for (p = lst; p != NULL; p = p->prox)
       printf("info = %d\n", p->info);
}
```



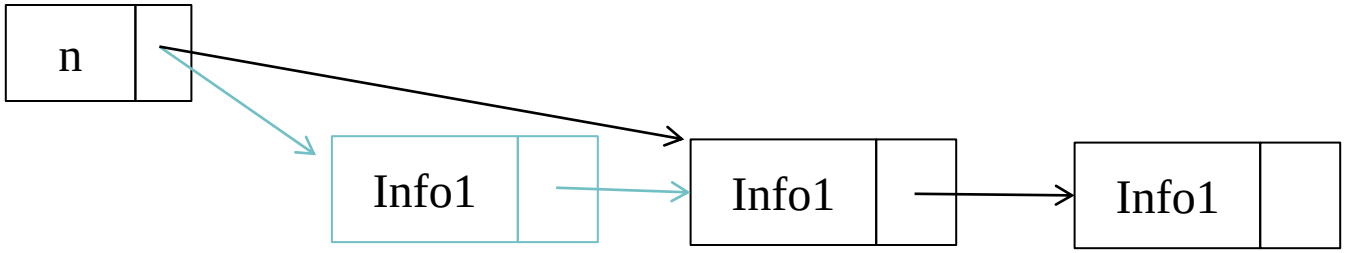
Listas Encadeadas: busca

- Recebe a informação referente ao elemento a pesquisar
- Retorna o ponteiro do nó da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista

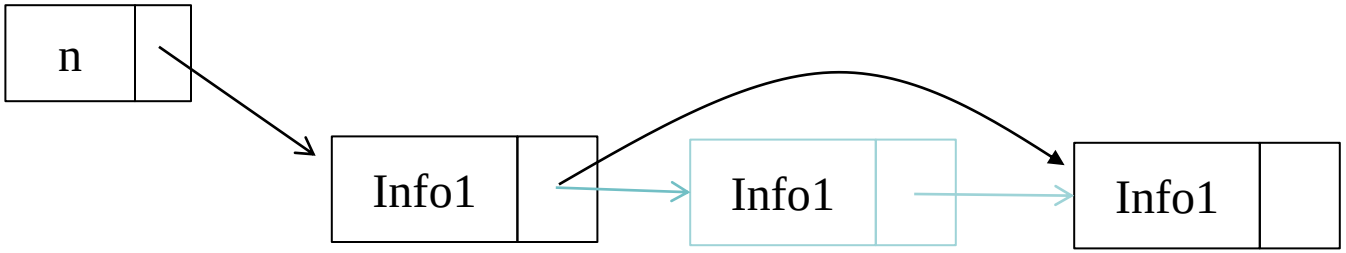
```
/* função busca: busca um elemento na lista */

Elemento* busca (Elem* lst, int v)
{
    Elem* p;
    for (p=lst; p!=NULL; p = p->prox) {
       if (p->info == v)
          return p;
    }
    return NULL;    /* não achou o elemento */
}
```

Listas Encadeadas: remover um elemento

- Recebe como entrada a lista e o valor do elemento a retirar
- Atualiza o valor da lista, se o elemento removido for o primeiro



- Caso contrário, apenas remove o elemento da lista



```c
/* função retira: retira elemento da lista */
Elem* lst_retira (Elem* lst, int val)
{
    Elem* ant = NULL; /* ponteiro para elemento anterior */
    Elemo* p = lst; /* ponteiro para percorrer a lista */
    /* procura elemento na lista, guardando anterior */
    while (p != NULL && p->info != val) {
       ant = p;
       p = p->prox;
    }
    /* verifica se achou elemento */
    if (p == NULL)
       return lst; /* não achou: retorna lista original */
    /* retira elemento */
    if (ant == NULL)
    {        /* retira elemento do inicio */
       lst = p->prox; }
    else {   /* retira elemento do meio da lista */
       ant->prox = p->prox; }
    free(p);
    return lst;
}
```

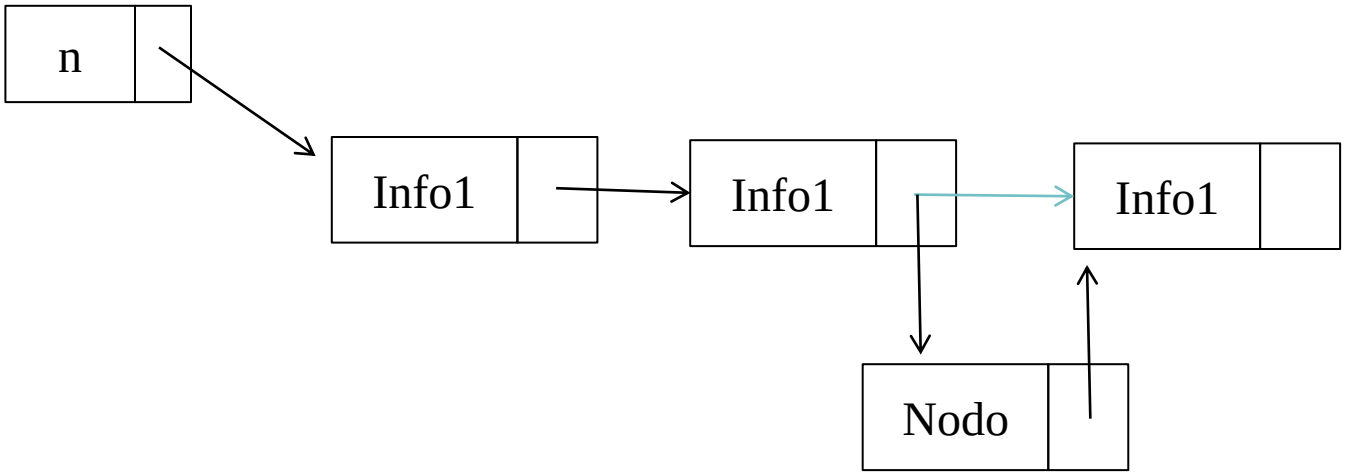
Listas Encadeadas: Libera a lista

- Destrói a lista, liberando todos os elementos alocados

```
void lst_libera (Elem* lst)
{
    Elem *p = lst, *t;
    while (p != NULL) {
       t = p->prox;    // guarda referência p/ próx. elemento
       free(p);        // libera a memória apontada por p
       p = t;          // faz p apontar para o próximo
    }
}
```

Listas Encadeadas Ordenadas

- Lista ordenada
  - Função de inserção percorre os elementos da lista até encontrar a posição correta para a inserção do nove elemento



```
Elemento* lst_insere_ordenado (Elemento* lst, int val)
{
   Elemento* novo;
   Elemento* ant = NULL;    // ponteiro para elemento anterior
   Elemento* p = lst;       // ponteiro para percorrer a lista
      // procura posição de inserção
   while (p != NULL && p->info < val)  {
       ant = p; p = p->prox; }
      // cria novo elemento
   novo = (Elemento*) malloc(sizeof(Elemento));
   novo->info = val;
      // encadeia elemento
   if (ant == NULL)
   {          // insere elemento no início
      novo->prox = lst; lst = novo; }
   else {     // insere elemento no meio da lista
      novo->prox = ant->prox;
      ant->prox = novo;
   }
   return lst;
}
```

Função imprime recursiva

- De inicio para fim

```
void função (Lista* lst)
{
   ---
   lst_imprime_rec(lst->lst);
   ---
}
```

```
void lst_imprime_rec (Elemento* lst)
{
   if ( ! lst_vazia(lst)) {
          /* imprime primeiro elemento */
      printf("info: %d\n",lst->info);
          /* imprime sub-lista */
      lst_imprime_rec(lst->prox);
   }
}
```

- De fim para inicio

```
void lst_imprime_rec (Elemento* lst)
{
   if ( ! lst_vazia(lst)) {
          /* imprime sub-lista */
      lst_imprime_rec(lst->prox);
          /* imprime ultimo elemento */
      printf("info: %d\n",lst->info);
   }
}
```

Igualdade de listas

- Elementos iguais: recursiva

```
int lst_igual (Elemento* lst1, Elemento* lst2)
{
   if(lst1 == NULL && lst2 == NULL)
       return 1;
   else
       if (lst == NULL || lst2 == NULL )
          return 0;
       else
          return (lst1->info == lst2->info) &&
                  lst_igual (lst1->prox, lst2->prox);
}
```

Lista de Tipos Estruturados

- Informação associada a cada no pode ser composta
  - Tipo de dados abstratos
  - Ponteiros de outras informações complexas

```
typedef struct freqSimb {
   char simb;
   int freq;
} tipFreqSimb;

typedef struct lista {
   tipFreqSimb *info;
   struct lista *prox;
} tipLista;
```

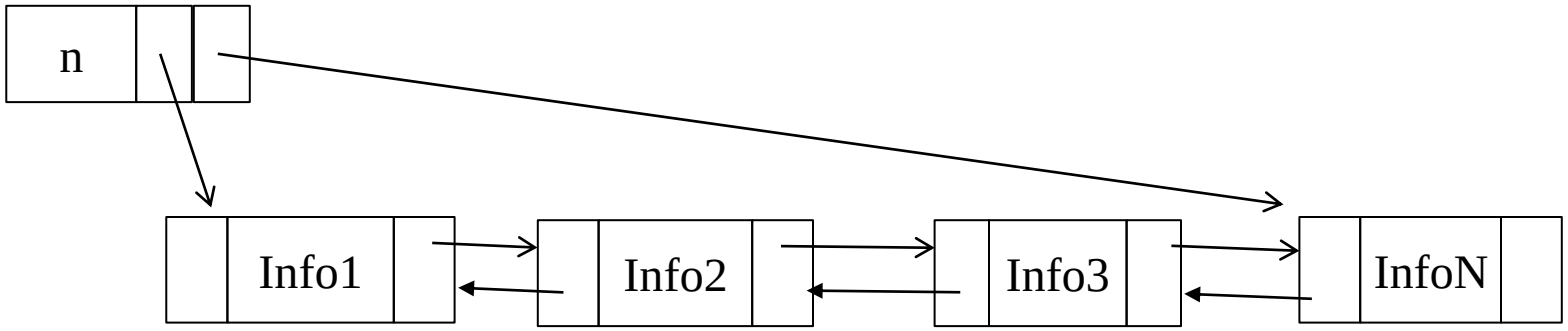
Lista de Tipos Estructurados

- Exemplo:

```
static tipLista* aloca (char c, int f)
{
   tipFreqSimb* r = (tipFreqSimb*) malloc(sizeof(tipFreqSimb));

   tipLista* p = (tipLista*) malloc(sizeof(tipLista));

   r->simb = c;
   r->freq = f;
   p->info = r;
   p->prox = NULL;

   return p;
}
```

Complementares

- Listas duplamente encadeadas
  - Cada elemento tem dois ponteiro
    - Próximo (prox) e Anterior (ant)



Exemplo: Listas Duplamente Encadeadas

```
Typedef struct lista2 {
   struct lista2* ant;
   int info;
   struct lista2* prox;
} tipLista2;

tipLista2* lst2_insere (tipLista2* lst, int val)
{
   tipLista2* novo = (tipLista2*) malloc(sizeof(tipLista2));
   novo->info = val;
   novo->prox = lst;
   novo->ant = NULL;
      /* verifica se lista não estava vazia */
   if (lst != NULL)
      lst->ant = novo;
   return novo;
}
```

Trabalho:

Dado um arquivo de texto (“arquivo.txt”) contendo um texto de algum tema (umas 100 palavras).

Criar uma lista dupla para estrutura de árvore, para registrar, em forma ordenada, cada LETRA do arquivo. Na lista deve aparecer uma única vez cada letra do documento, registrando número de vezes (frequencia) que aparece no documento.

O algoritmo deve ser:

Para cada letra lida,
Verificar se existe na lista em processo de geração.
Se existir, acumular +1 na frequencia.
Caso não existir, inserir na lista o node da letra e frequencia 1.

Cada letra vai ser inserida (em node) de forma ordenada e crescente

Imprimir o conteúdo da lista.

Estrutura de dados (recomendada):

```
typedef  struct  stNode {
      char letra;               // campo para letra ou caractere
       int   freq;              // contador de frequencia
       struct stNode  *next;    // apontador para proximo node
 } Node;

typedef  struct  stLista {
       int   n;              // acumulador de número de tipo de caracteres
       Node *first;          // apontador para o primeiro node da lista
 } Lista;
```