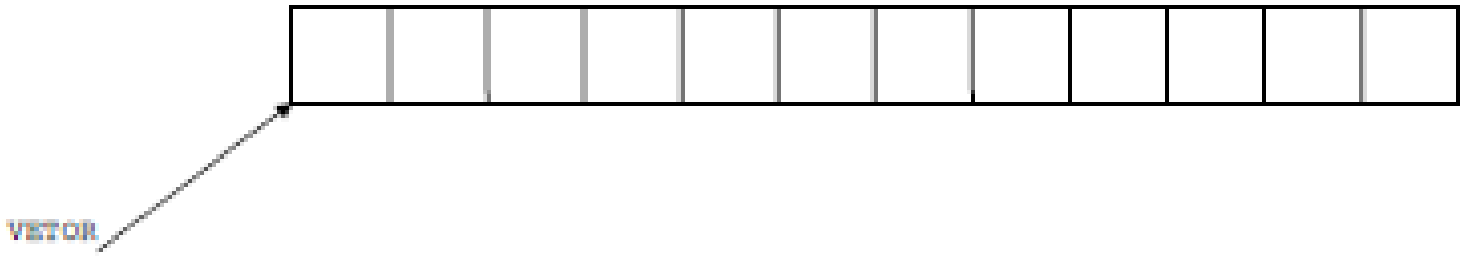
Programação 2

Listas

Rivera

Motivação

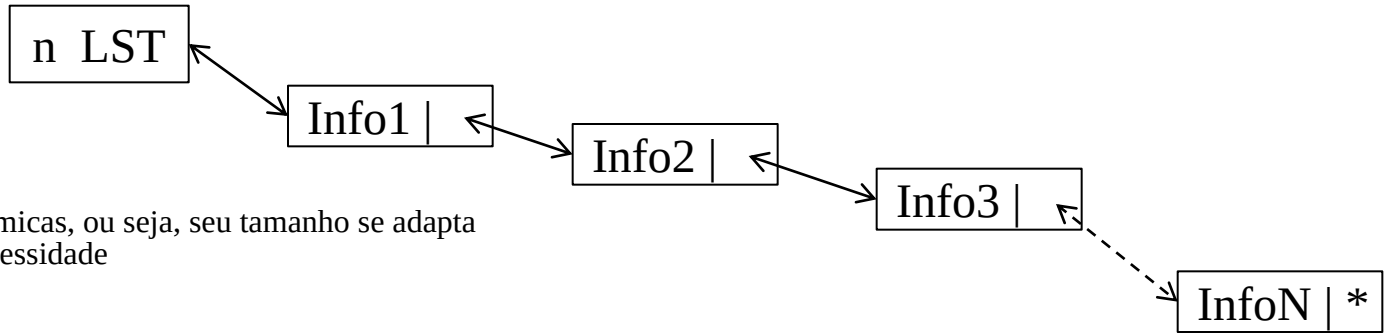
- Vetor
  - Ocupa um espaço contíguo de memória
  - Permite acesso randômico aos elementos
  - Deve ser dimensionado com um número máximo de elementos



um vetor é definido previamente, com tamanho máximo restrito

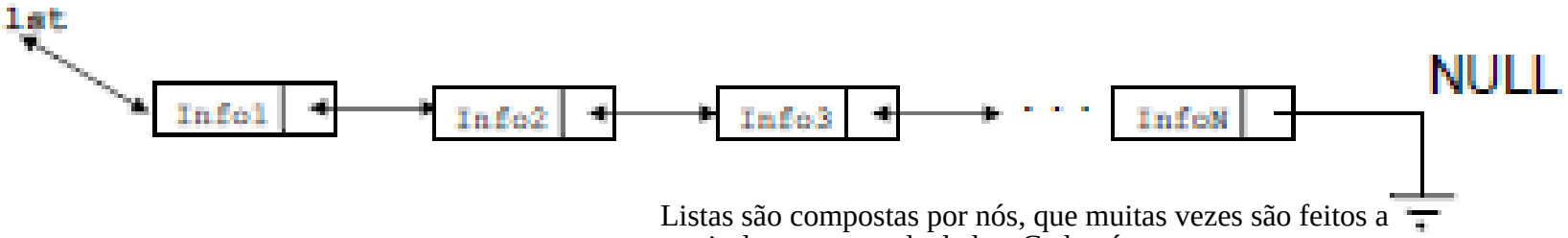
Motivação

- 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



Listas são dinâmicas, ou seja, seu tamanho se adapta conforme a necessidade

Listas

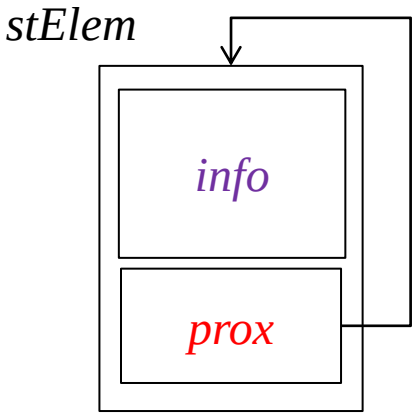


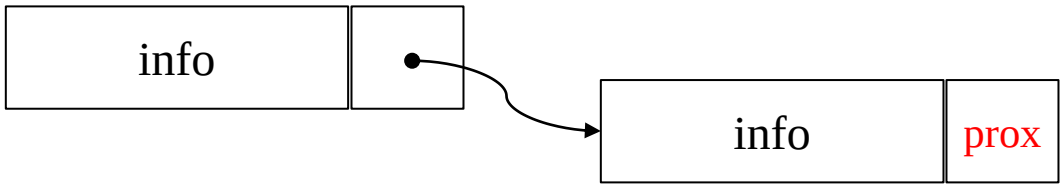
Listas são compostas por nós, que muitas vezes são feitos a partir de estruturas de dados. Cada nó carrega sua respectiva informação, e aponta para o próximo elemento da lista. O último elemento da lista aponta para NULL

- Lista encadeada
  - Sequencia encadeada de elementos
    - Chamados de NÓS da lista
  - NÓ da lista é representado por 2 campos
    - A informação armazenada
    - O ponteiro para o próximo elemento da lista
  - A lista é representada por um ponteiro para o primeiro NÓ
  - O ponteiro do último elemento é NULL

Estrutura

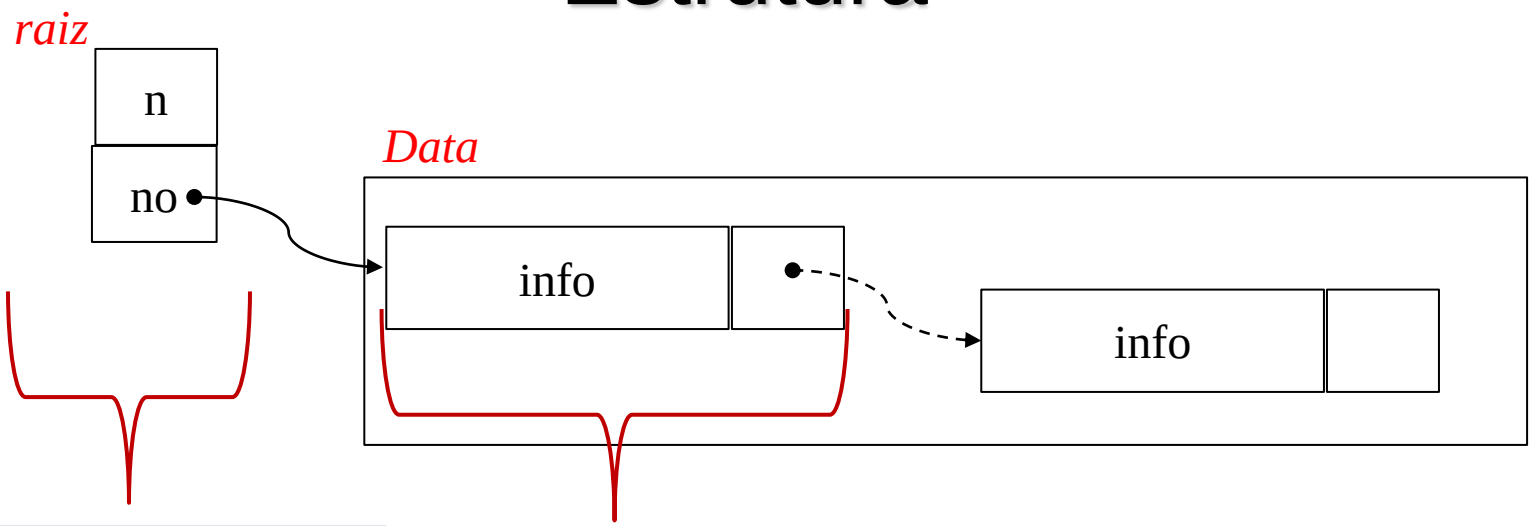
```
Tipo
    estrutura Elem
    {
        tipoData  info;
        estrutura Elem* prox;
    }
stElem;
```





uma estrutura pode apontar pra ela mesma, como no exemplo acima, onde a informação de uma estrutura aponta para o próximo elemento, que é do mesmo tipo de estrututa.

Estrutura

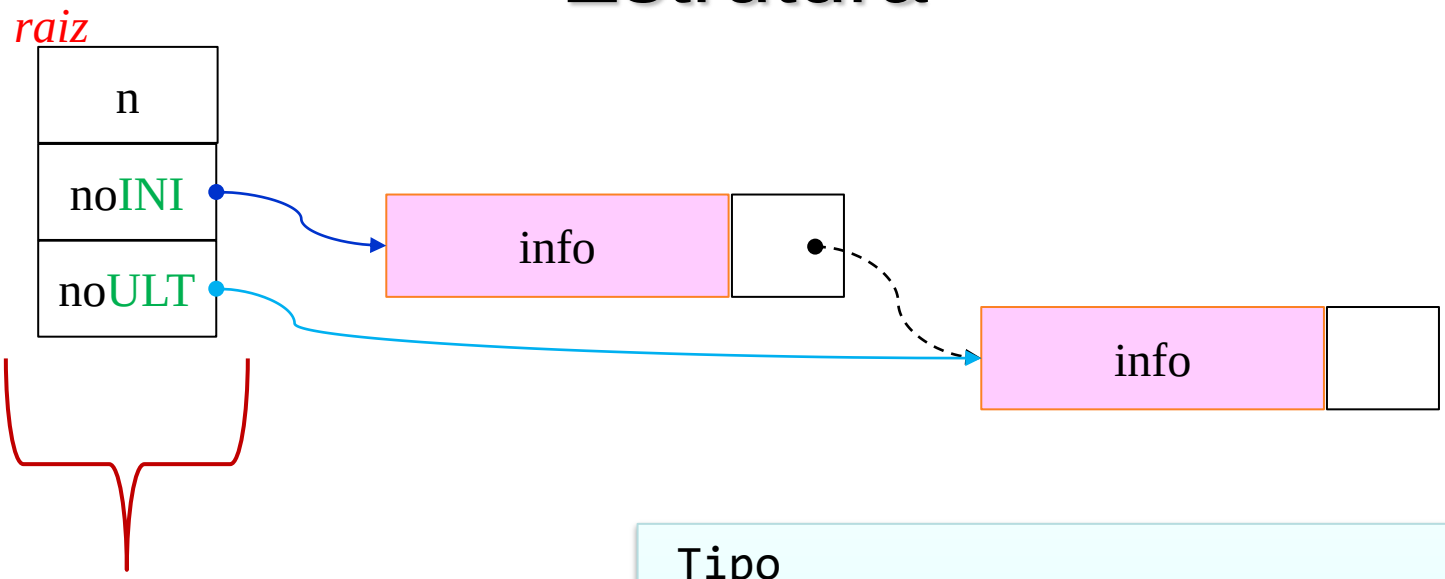



```
Estrutura lista
{
   int  n;
   estruturaElem *no;
}
```

```
Tipo
   estrutura Elem
   {

      tipoData  info;
      estrutura Elem* prox;

   }
stElem;
```

A raiz define o início, o nó inicial, e a estrutura da lista.

Estrutura




```
Estrutura lista
{
    int  n;
    estruturaElem *noINI;
    estruturaElem *noULT;
}
```

```
Tipo
    estrutura Elem
    {

        tipoData  info;
        estrutura Elem* prox;

    }
stElem;
```

nesse caso, a raiz aponta tanto para o nó inicial, ou seja, a primeira informação, tanto para o último nó, ou seja, a última informação dentro da lista.

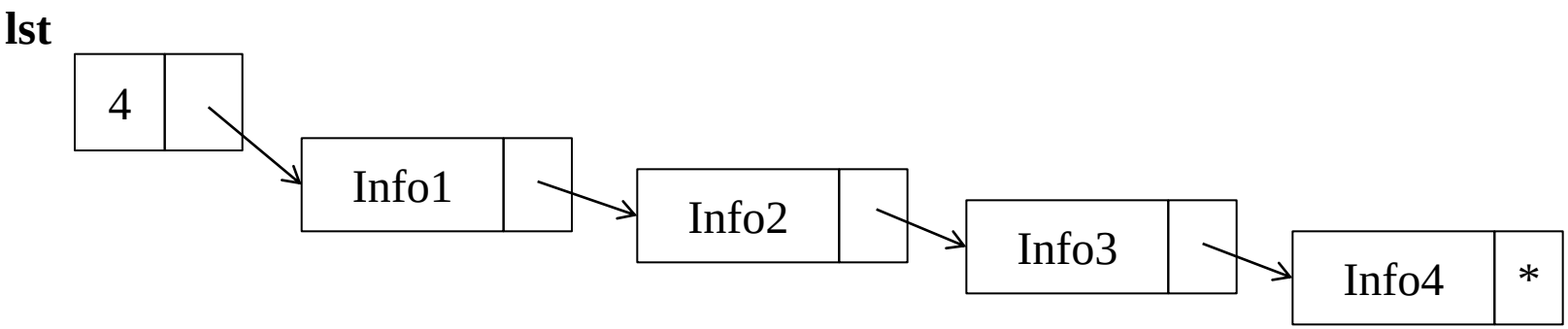
Em C ?

Exemplo: « Criar uma lista para armazenar n números aleatoriamente »

Listas Encadeadas: impressão

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

void listImprime (        )
{



}
```

A função de impressão recebe um ponteiro, que aponta para o primeiro nó da lista, e imprime sucessivamente todos os dados, um a um, até que se chega no último nó, contendo NULL.



Listas Encadeadas: busca

- Recebe a informação referente ao elemento a pesquisar
- Buscar um elemento (int v) na lista (Elem list)

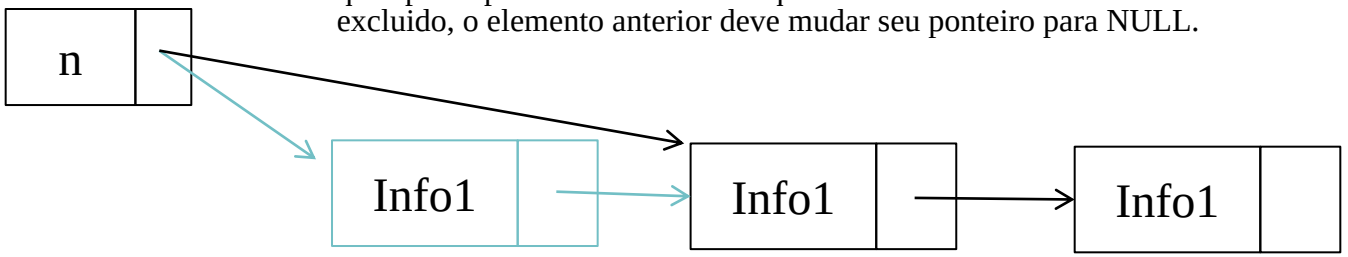
```
/* função busca: busca um elemento na lista */

Elemento* busca (Elem* lst, int v)
{


}
```

a função de busca recebe um ponteiro inicial, que aponta para o primeiro nó, e percorre cada nó da lista, analisando seus elementos, até que se ache o resultado, ou encontre NULL.

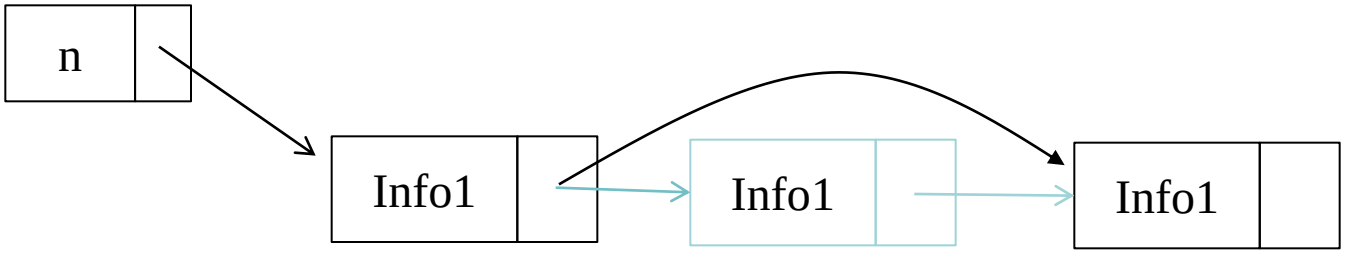
Listas Encadeadas: remover um elemento

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

para remover um elemento, recebe-se a lista, o elemento a ser retirado, e muda o endereço apontando pelo elemento anterior (caso o elemento a ser removido não seja o primeiro) para que aponte para o elemento subsequente do nó removido. Caso o último elemento seja excluido, o elemento anterior deve mudar seu ponteiro para NULL.



- Caso contrário, apenas remove o elemento da lista



Listas Encadeadas: Libera a lista

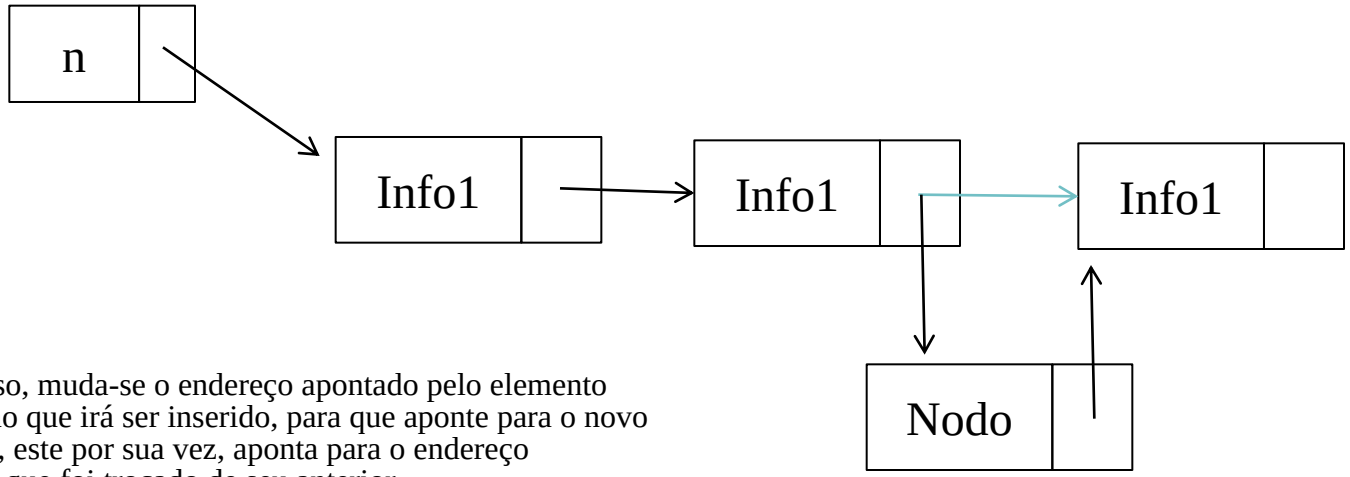
- Destrói a lista, liberando todos os elementos alocados

```
void lst_libera (Elem* lst)
{



}
```

A liberação da lista ocorre do início para o final, com um ponteiro auxiliar, que guarda o endereço do próximo nó, e exclui o atual, tornando o próximo o primeiro, e assim sucessivamente.

Listas Encadeadas Ordenadas

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



Nesse caso, muda-se o endereço apontado pelo elemento anterior ao que irá ser inserido, para que aponte para o novo elemento, este por sua vez, aponta para o endereço apontado que foi trocado de seu anterior.

Exemplo em C.

Função imprime recursiva

- De inicio para fim

```
void lst_imprime_rec (Elemento* lst)
{
   if ( ! lst_vazia(lst)) {



      }
}
```

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

- De fim para inicio

```
void lst_imprime_rec (Elemento* lst)
{
   if ( ! lst_vazia(lst)) {


   }
}
```

a lista pode ser impressa de maneira recursiva, sempre imprimindo o elemento atual e apontando para o próximo, chamando a função de impressão novamente, com o próximo nó como parâmetro

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;
```

cada lista pode também utilizar de tipos abstratos em sua composição, que são definidos anteriormente no código, priorizando a funcionalidade, sem se importar com a implementação.

Lista de Tipos Estructurados

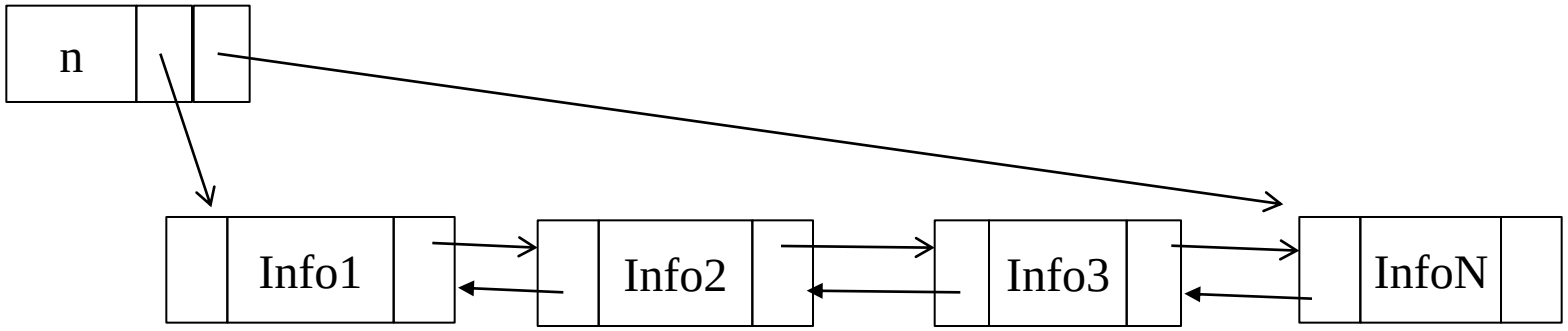
- Exemplo:

```
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;
}
```

estrutura composta, onde as informações dentro dela são também estruturas

Complementares

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



listas não possuem necessariamente apenas 1 nó, podendo indicar seu elemento posterior e anterior, caso seja necessário.

Exemplo: Listas Duplamente Encadeadas

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

insere um novo elemento na lista, verificando se a posição desejada não é a ultima, e caso não seja, o elemento posterior tem o novo elemento declarado como seu anterior.

Trabalho:

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

Criar uma lista simples, em modo FIFO, 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 .

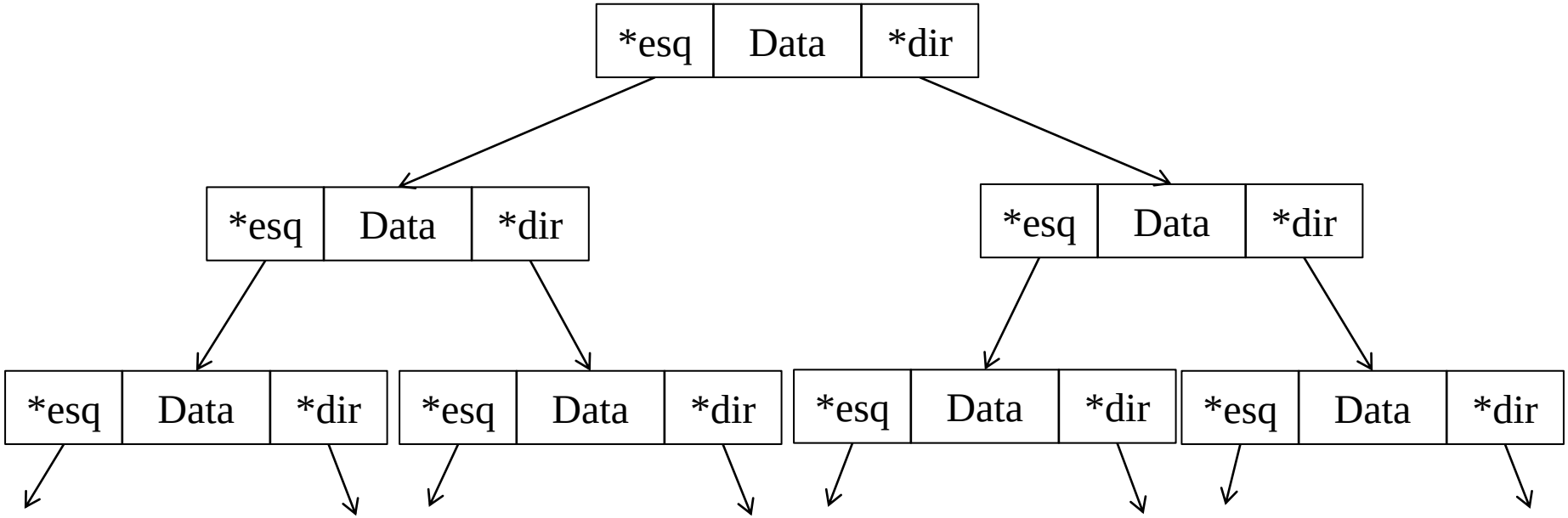
Cada letra vai ser inserida (em node) de forma ordenada e crescente

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;
```

Árvore Binária



cada informação possui ponteiros apontando para a informação a esquerda ou a direita, ramificando a árvore binária.

Árvores Binárias

Tipos

- Balanceada
- Não balanceada

Árvores binárias balanceadas são as que todos os elementos possuem o mesmo número de ramificações, enquanto as não balanceadas não compartilham desta característica