Estrutura de Dados

Listas Lineares Encadeadas

Alocação Dinâmica

Lista Encadeada Dinâmica

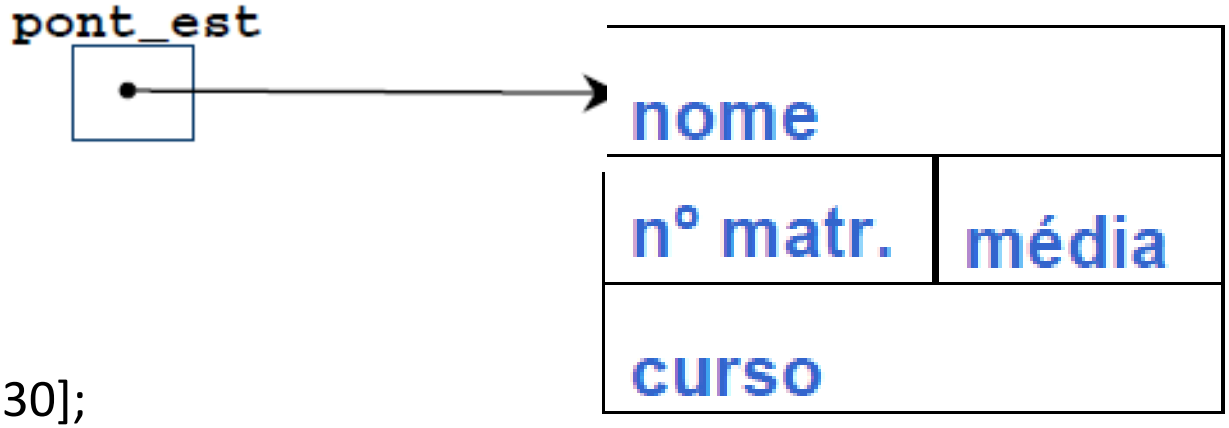
- Utiliza alocação dinâmica de memória ao invés de vetores pré-alocados.
- Inserção de elementos na lista: toda memória disponível para o programa durante a execução (*heap)*
- Espalhados, cada elemento deve conter o endereço do seu sucessor na lista: campo de ligação contém endereços reais da memória principal
- Alocação/liberação desses endereços gerenciada pelo S.Op., por meio de comandos da linguagem de programação
- Linguagem C: malloc e free

Variável Dinâmica

- Uma variável dinâmica é uma variável criada (e destruída) explicitamente durante a execução do programa
- Objetivo: Otimizar o uso da Memória Principal
- Variáveis dinâmicas não são declaradas, pois inexistem antes da execução do programa
- Ela é referenciada por uma variável ponteiro, que contém o endereço da variável dinâmica
- A variável ponteiro deve ser declarada
- Ao declarar uma variável ponteiro, deve-se declarar o tipo de valor que ela referencia

Variável dinâmica: exemplo

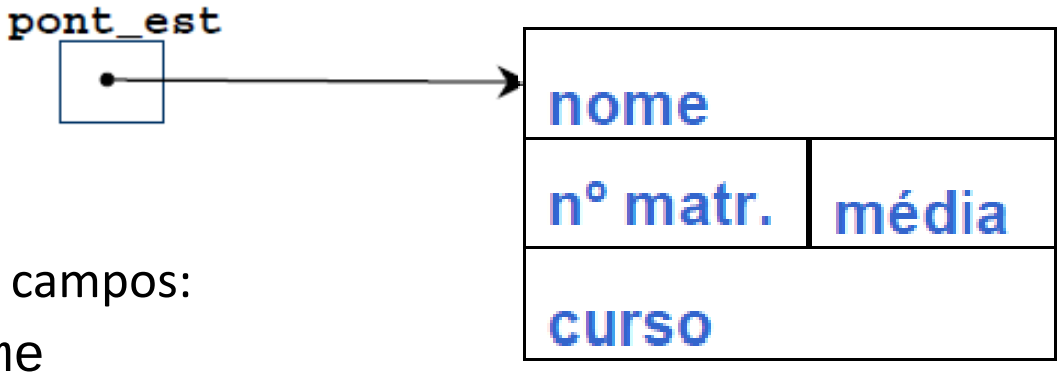
- Suponha que uma variável dinâmica deverá conter os dados de um estudante (***nome, nº matrícula, curso, média***)




```
typedef  struct{
        char nome[30];
        int n_matr;
        float media
        char curso[30];
}treg;

treg  *pont_est;
```

Criação da variável dinâmica

- `pont_est = (treg *) malloc (sizeof(treg));`
- Aloca memória para um dado do tipo treg e estabelece a ligação:



- Para acessar os campos:

```
pont_est->nome
pont_est->n_matr
pont_est->media
pont_est->curso
```

Destruindo uma variável dinâmica

- Para liberar o espaço de memória que a variável dinâmica ocupa:
  `free(pont_est)`

- A constante pré-definida **NULL** indica ligação nula.

Lista encadeada dinâmica

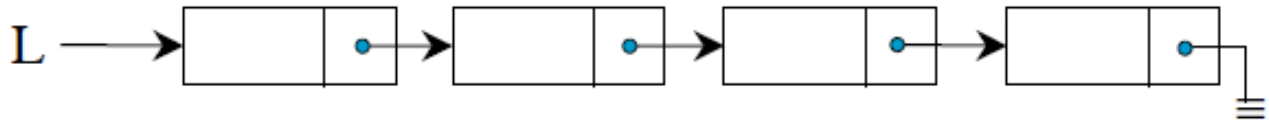
- Os registros disponíveis para **inserção** correspondem a toda memória disponível para o programa durante a execução.

- O campo de ligação dos registros não contem mais índices de vetores, mas endereços reais da memória principal.

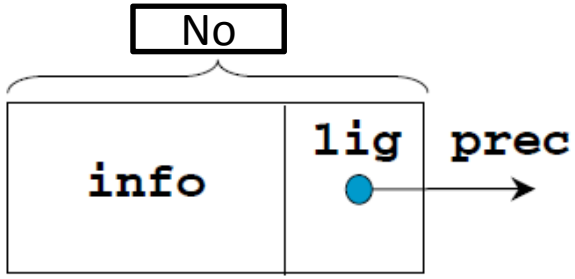
- Quem controla esses endereços é o SO através de comandos da linguagem de programação.

- **C** manipula registro da memória principal dinamicamente (em tempo de execução) através do tipo de dados **ponteiro->**

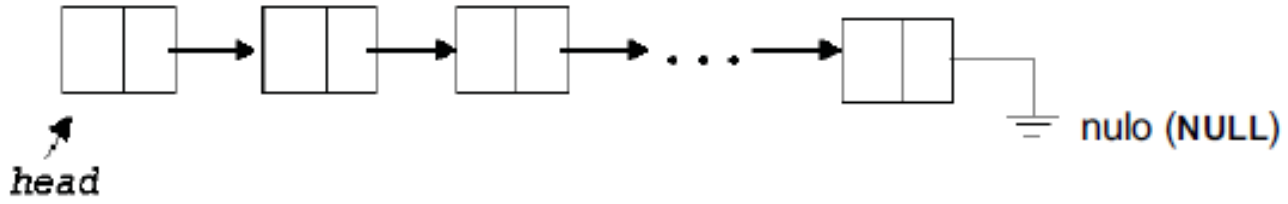
Lista dinâmica simplesmente encadeada




```
typedef struct no_lista{
        tipo_elem info;
        struct no_lista *lig;
}No;
```



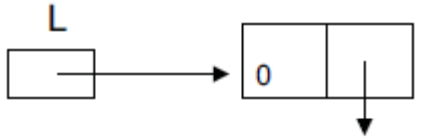

```
typedef struct{
        int nelem;
        No *head;
}Lista;
```



Implementação das operações

1. Criação da Lista Vazia

```
void CriarLista(Lista *L){
        L->nelem = 0;
        L->head = NULL;
}
```



/* a constante NULL é parte da biblioteca <stdlib.h> */

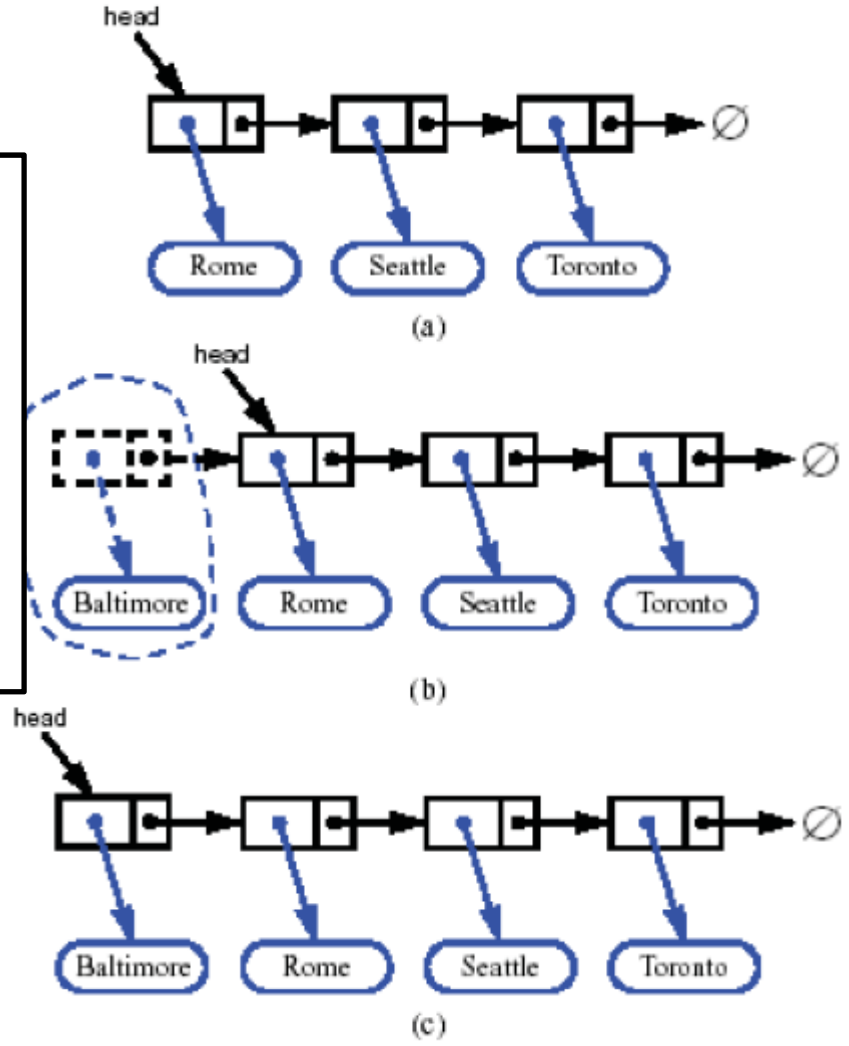
2. Inserção do primeiro elemento

```
void Insere_Prim(Lista *L, Tipo_elem elem){
        No_Lista *p;
        p = (No *)malloc(sizeof(No));
        p->info = elem;
        p->lig = NULL;
        L->head = p;
        L->nelem++;
}
```

Implementação das operações

3. Inserção no início de uma lista

```
void Insere_Inicio(Lista *L, tipo_elem  elem){
        No *p;
        p = malloc(sizeof(No));
        p->info = elem;
        p->lig = L->head;
        L->head = p;
        L->nelem++;
}
```



Implementação das operações

4. Acesso ao primeiro elemento da lista

```
tipo_elem Primeiro(Lista *L){
        return L->head->info;
}
```

5. Quantos elementos tem na lista?

```
int Nelem(Lista *L){
        return L->nelem;
}
/* se nelem tiver atualizado */
```

```
int Nelem(Lista *L){
        No *p = L->head;
        int count = 0;
        while (p != NULL){
                count ++;
                p = p->lig;
        }
        return count;
}
```

Implementação das operações

Versão recursiva

```
int Nelem_rec(No *p){
        if (p == NULL)
                return 0;
        else
                return (1 + Nelem_rec(p->lig));
}
```

```
int Nelem_rec_init(Lista *L){
        return Nelem_rec(L->head);
}
```

Implementação das operações

5-a) Buscar registro de chave x em lista ordenada – versão iterativa

```
/*Busca por x e retorna TRUE e o endereço (p) de x numa Lista Ordenada, se achar; senão,retorna FALSE */
Boolean Buscar_ord (Lista *L, Tipo_chave x, No *p){
      if(L->nelem == 0) /*Lista vazia, retorna NULL*/
            return FALSE;
      else{
            p = L->head;
            while (p != NULL){ /* enquanto não achar o final */
                  if (p->info.chave >= x){
                        if (p->info.chave == x)/* achou o registro*/
                              return TRUE;
                        else  /* achou um registro com chave maior*/
                              return FALSE;
                  }else p = p->lig;
            }
            return FALSE; /* achou final da lista*/
            }
}
```

Implementação das operações

5-b) Buscar registro de chave x em lista ordenada – versão recursiva

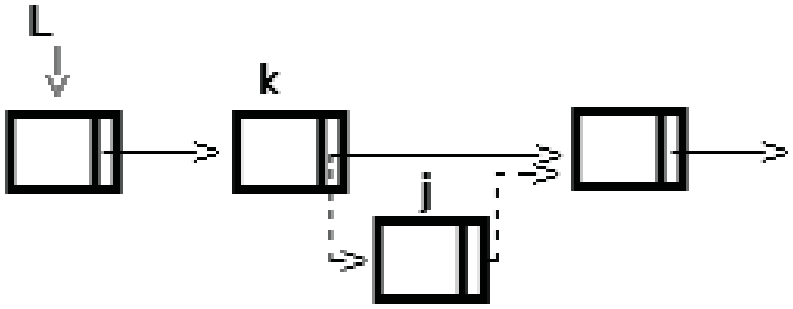
```
/*Busca por x e retorna TRUE e o endereço (p) de x numa
Lista Ordenada, se achar; senão,retorna FALSE */

Boolean Busca_ord_rec_init(Lista *L, Tipo_chave x, No* p){
      if(L->nelem == 0) /*Lista vazia, não achou*/
            return FALSE;

      p = L->head;

      return (Busca_ord_rec(p, &x));
}
```

```
Boolean Busca_ord_rec(No *q, Tipo_chave *x){
      if (q == NULL)
            /* chegou no final da lista, sem achar*/
            return FALSE;
      else
            if (q->info.chave >= *x){
                  if (q->elem.chave == *x)  /* achou o registro*/
                        return TRUE;
                  else  /* achou um registro com chave maior*/
                        return FALSE;
            }else
                  return Busca_ord_rec(q->lig, x);
}
```

Implementação das operações

6a) Inserção de elemento *v* como sucessor do elemento no endereço k

```
void Insere_Depois(Lista *L,Tipo_elem v, No *k){
        /*k não pode ser null*/
        No* j = malloc(sizeof(No));
        j->info = v;
        j->lig = k->lig;
        k->lig = j;
        L->nelem++
}
```



OBS.: FUNCIONA PARA INSERÇÃO APÓS ÚLTIMO ELEMENTO?

6b) Inserção do elemento v na lista ordenada L

```
/*Insere item de forma a manter a lista ordenada.
Retorna true se inseriu; false, se não foi possível inserir*/

boolean Insere(Lista *L, Tipo_elem v){
    if (L->nelem == 0){ /*insere como primeiro elemento*/
        insere_Prim(L, v);
        return TRUE;
    }
    No *p = L->head;
    No *pa = NULL;

    /*****/
```

```
        while (p != NULL){
            if (p->info.chave >= v.chave){
                if (p->info.chave == v.chave) /* v já existe na lista*/
                    return FALSE;
                else{
                    if (pa == NULL)    /*insere no inicio */
                        Insere_Inicio(L, v);
                    else{     /*insere no meio*/
                        Insere_Depois(L, v, pa);
                    }
                    return TRUE;
                }
            }else{
                pa = p;
                p = p->lig;
            }
        }
        /*insere no final*/
        Insere_Depois(L, v, pa);
        return TRUE;
} // fim
```

(c) Inserção do elemento *v* na lista ordenada L (Recursivo)

```
/*Insere item de forma a manter a lista ordenada.
Retorna true se inseriu; false, se não foi possível inserir*/
boolean Insere_rec_init(Lista *L, Tipo_elem v){
        if (L->nelem == 0){
        /*insere como primeiro elemento*/
        insere_Prim(L, v);
                return TRUE;
        }
        No *p = L->head;
        No *pa = NULL;
        return Insere_rec(L, p, pa, &v);
}
```

```
boolean Insere_rec(Lista *L, No *p, No *pa, tipo_elem *v){
    if (p == NULL) {
        /*insere no final */
        Insere_Depois(L, *v, pa);
        return TRUE;
    }
    if (p->info.chave == v->chave)
        /* v já existe na lista*/
        return FALSE;
    if (p->info.chave > v->chave){
        if (pa == NULL)
            /*insere no inicio */
            Insere_Inicio(L, *v);
        else{
            /*insere entre pa e p*/
            Insere_Depois(L, *v, pa);
        }
        return TRUE;
    }
    return Insere_rec(L, p->lig, p, v);
}
```

Implementação das operações

7) Remoção do primeiro elemento

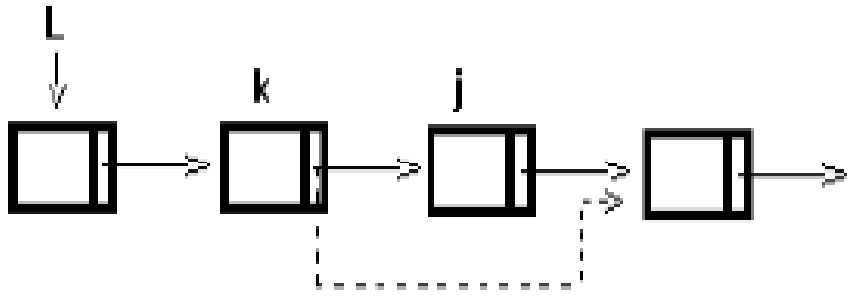
```
void Remove_Prim(Lista *L){
        /* supõe que a Lista não está vazia */
        No *p = L->head;
        L->head = p->lig;
        free(p);
        L->nelem--;
}
```

Obs: funciona no caso de remoção em lista com um único elemento?

Implementação das operações

8) Remoção do elemento apontado por j, sucessor do elemento no endereço k

```
void Elimina_Depois(Lista *L, No *k){
        No *j = k->lig;
        k->lig = j->lig;
        free(j);
        L->nelem--;
}
```



Implementação das operações

9) Eliminar elemento v de uma lista ordenada L

```
boolean Remove(Tipo_elem v, Lista*L){
        No *p = L->head;
        No *pa = NULL;
        while (p != NULL){
                if (p->info.chave < v.chave){
                        pa = p;
                        p = p->lig;
                } else {
                        if (p->info.chave > v.chave)  /* encontrou elemento com chave maior*/
                                return FALSE;
                        else {    /*encontrou o elemento*/
                                if (pa == NULL)      /*remove no inicio*/
                                        Remove_Prim(L);
                                else{      /*remove elemento p*/
                                        Elimina_Depois(L,pa);
                                }
                                return TRUE;
                        }
                }
        } /*não encontrou o elemento na lista*/
        return FALSE;}
```

Exercício: Fazer a eliminação de v de lista não ordenada

Implementação das operações

10) Impressão da lista

```
void imprime(Lista *L){
        No *p;
        p = L->head;
        while (p != NULL){
                impr_elem(p->info);
                p = p->lig;
        }
}
```

```
void impr_elem(Tipo_elem t){
        printf("chave: %d", t.chave);
        printf("info: %s", t.info.valor);
}
```

EXERCÍCIOS

Exercícios

- Explique o que acontece nas atribuições abaixo (dica: use desenhos)

a) p->lig = q;

b) p->lig = q->lig;

c) p->info = q->info;

d) p = q;

e) p->lig = nil;

f) *p = *q;

g) p = p->lig;

h) p = (p->lig)->lig;

Exercícios

Elaborar os seguintes TADs, usando alocação dinâmica. Implementar esse TAD na linguagem C usando estrutura modular.

- Lista Encadeada Ordenada
- Lista Encadeada Não-ordenada

Exercícios

Dada uma lista ordenada L1 encadeada alocada dinamicamente (i.e., implementada utilizando pointer), escreva as operações:

a) Verifica se L1 está ordenada ou não (a ordem pode ser crescente ou decrescente)

b) Faça uma cópia da lista L1 em uma outra lista L2

c) Faça uma cópia da Lista L1 em L2, eliminando elementos repetidos

d) inverta L1 colocando o resultado em L2

e) inverta L1 colocando o resultado na própria L1

f) intercale L1 com a lista L2, gerando a lista L3 (L1, L2 e L3 ordenadas)

Exercícios

•Escreva um programa que gera uma lista L2, a partir de uma lista L1 dada, em que cada registro de L2 contém dois campos de informação

-elem contém um elemento de L1, e count contém o número de ocorrências deste elemento em L1

• Escreva um programa que elimine de uma lista L dada todas as ocorrências de um determinado elemento (L ordenada)

• Assumindo que os elementos de uma lista L são inteiros positivos, escreva um programa que informe os elementos que ocorrem mais e menos em L (forneça os elementos e o número de ocorrências correspondente)

Este material didático foi adaptado do material da profa. Graça Nunes, do ICMC-USP São Carlos