SCC 202 – Algoritmos e Estruturas de Dados I

Listas Lineares Encadeadas

Alocação dinâmica

Lista Encadeada Dinâmica

- Utiliza alocação dinâmica de memória ao invés de arranjos (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`

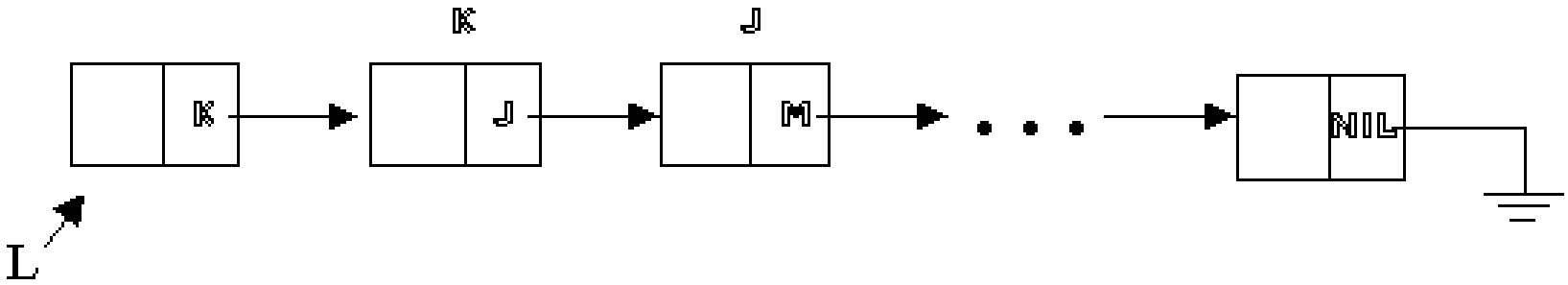
Variável Dinâmica

- Uma variável 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

Lista Dinâmica

Visualização de uma lista encadeada

k, j e m são posições de memória não consecutivas e L é o ponteiro para o início da lista

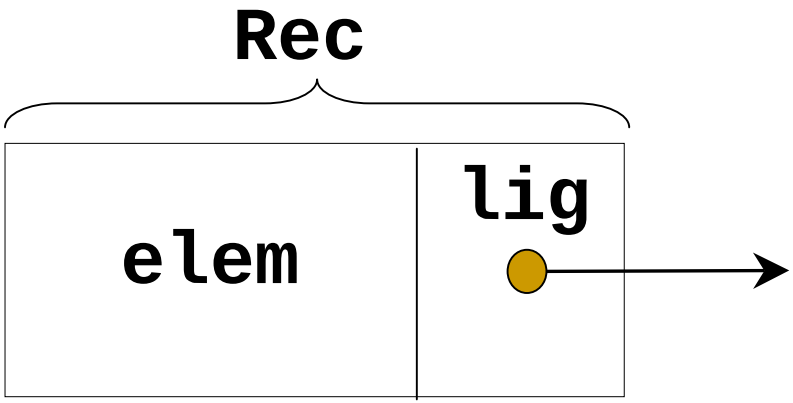


Endereço nulo (terra): null

L: ponteiro para o primeiro elemento (ou null)

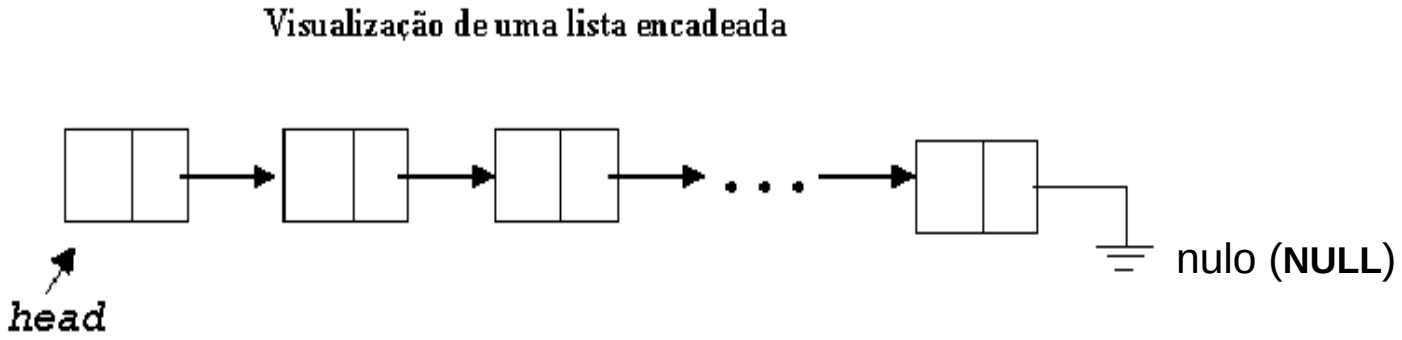
Lista Dinâmica

- **Definição da ED**



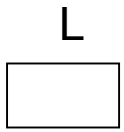

```
struct list_rec {
  tipo_elem elem;
  struct list_rec *lig;
};

typedef struct list_rec  Rec;
```

Listas Simplesmente Encadeadas



- **Lista:**

```
typedef struct {
        int nelem;
        Rec *head;
} Lista;
```

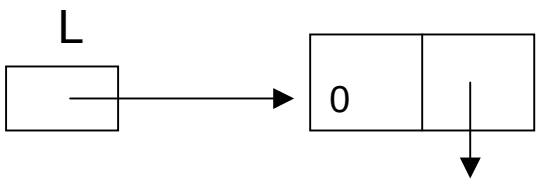



```
Lista *L;   /* Exemplo de Declaração */
```

Implementação das Operações

1) **Criação da lista vazia**

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




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

Implementação das Operações

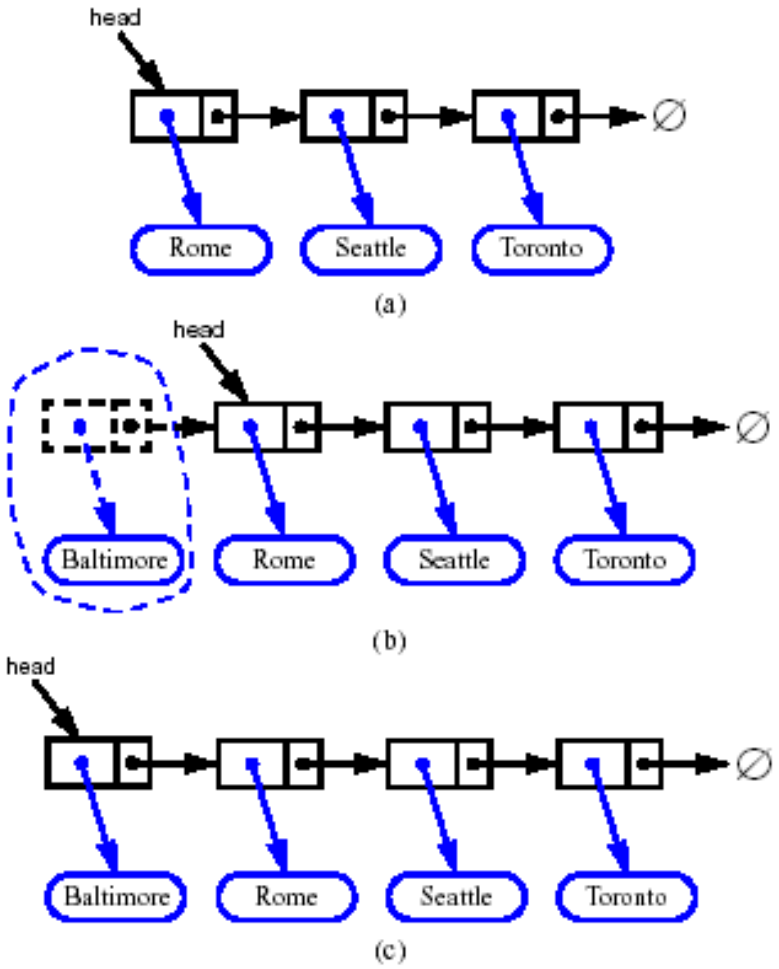
2) **Inserção do primeiro elemento**

```
void Insere_Prim(Lista *L, Tipo_elem elem){

  Rec *p;
  p = malloc(sizeof(Rec));
  p->elem = 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){
     Rec *p;
     p = malloc(sizeof(Rec));
     p->elem = 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->elem;
}
```

Implementação das Operações

Quantos elementos tem a lista ?

```
int Nelem(Lista *L){
    return L->nelem;
}

/* se nelem tiver atualizado */
```

```
int Nelem(Lista *L){

    Rec *p = L->head;
    int count = 0;

    while (p != NULL){
        count ++;
        p = p->lig;
    }

    return count;
}
```

Implementação das Operações

versão recursiva

```
int Nelem_rec(Rec *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

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

    if(L->nelem == 0) /*Lista vazia, retorna NULL*/

        return FALSE;

    else{

        p = L->head;

   /*...*/
```

```
    while (p != NULL){ /* enquanto não achar o final */

            if (p->elem.chave >= x){
                if (p->elem.chave == x)/* achou o registro*/
                    return TRUE;
                else
                    /* achou um registro com chave maior*/
                    return FALSE;
            }else{
                p = p->lig;
            }
        }

        /* achou final da lista*/
        return FALSE;
    }
}
```

Implementação das Operações

5 (b) Buscar registro de chave x em lista ordenada (Versão Recursiva)

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

    if(L->nelem == 0) /*Lista vazia, não achou*/
        return FALSE;

     p = L->head;
    return Busca_ord_rec(p, &x);

}
```

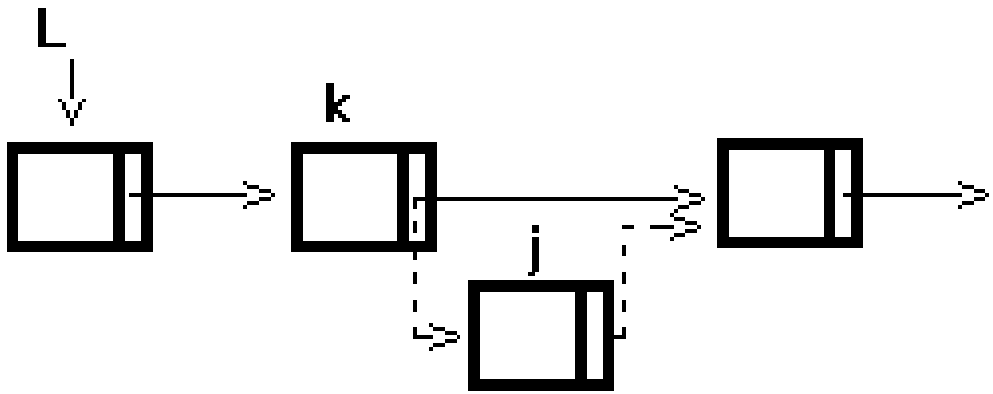
Passagem por endereço, mas poderia ser por valor (economiza espaço)

Exercício: Implementar uma versão para Lista não ordenada!

```
Boolean Busca_ord_rec(Rec *q, Tipo_chave *x){

        if (q == NULL)
            /* chegou no final da lista, sem achar*/
            return FALSE;
        else
            if (q->elem.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

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



Implementação das Operações

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

```
void Insere_Depois(Lista *L,Tipo_elem v, Rec *k){
    /*k não pode ser null*/
    Rec *j = malloc(sizeof(Rec));
    j->elem = v;
    j->lig = k->lig;
    k->lig = j;
    L->nelem++
}
```

Obs.: funciona para inserção após último elemento?

Implementação das Operações

(b) Inserção do elemento *v* na lista ordenada L

```
boolean Insere(Lista *L, Tipo_elem v){
/*Insere item de forma a manter a lista ordenada.
Retorna true se inseriu; false, se não foi possível inserir*/

    if (L->nelem == 0){
        /*insere como primeiro elemento*/
        insere_Prim(L, v);
        return TRUE;
    }

    Rec *p = L->head;
    Rec *pa = NULL;

     /*...*/
```

```
while (p != NULL){

    if (p->elem.chave >= v.chave){
        if (p->elem.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;
}
```

Implementação das Operações

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

```
boolean Insere_rec_init(Lista *L, Tipo_elem v){
/*Insere item de forma a manter a lista ordenada.
Retorna true se inseriu; false, se não foi possível inserir*/

    if (L->nelem == 0){
        /*insere como primeiro elemento*/
        insere_Prim(L, v);
        return TRUE;
    }

    Rec *p = L->head;
    Rec *pa = NULL;

    return Insere_rec(L, p, pa, &v);
}
```

```
boolean Insere_rec(Lista *L, Rec *p, Rec *pa, Tipo_elem *v){

   if (p == NULL) {
/*insere no final */
   Insere_Depois(L, *v, pa);
    return TRUE;}

   if (p->elem.chave == v->chave)
        /* v já existe na lista*/
        return FALSE;

   if (p->elem.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

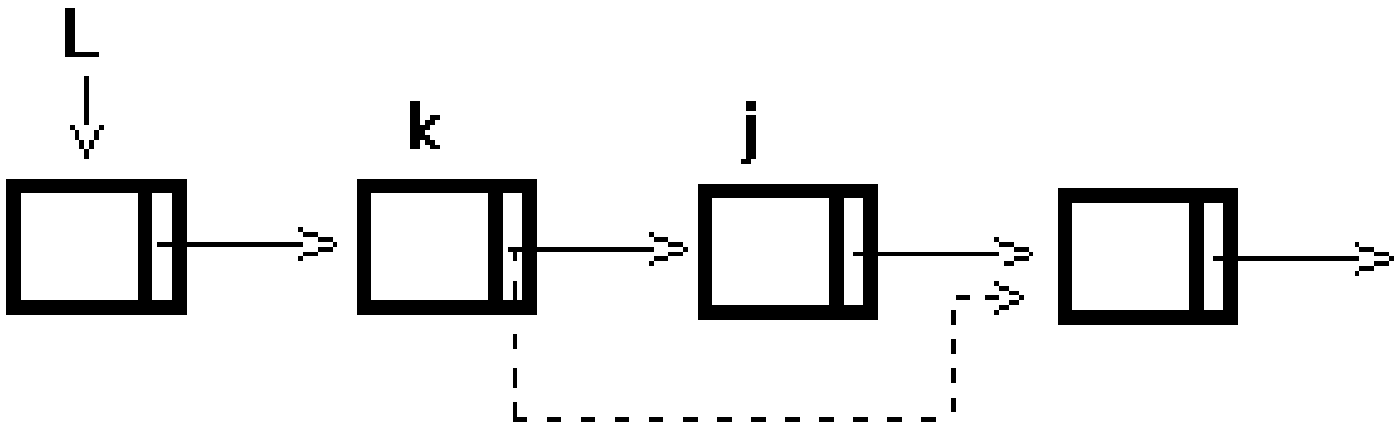
7) **Remoção do primeiro elemento**

```
void Remove_Prim(Lista *L){
   /* supõe que a Lista não está vazia */
     Rec *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**



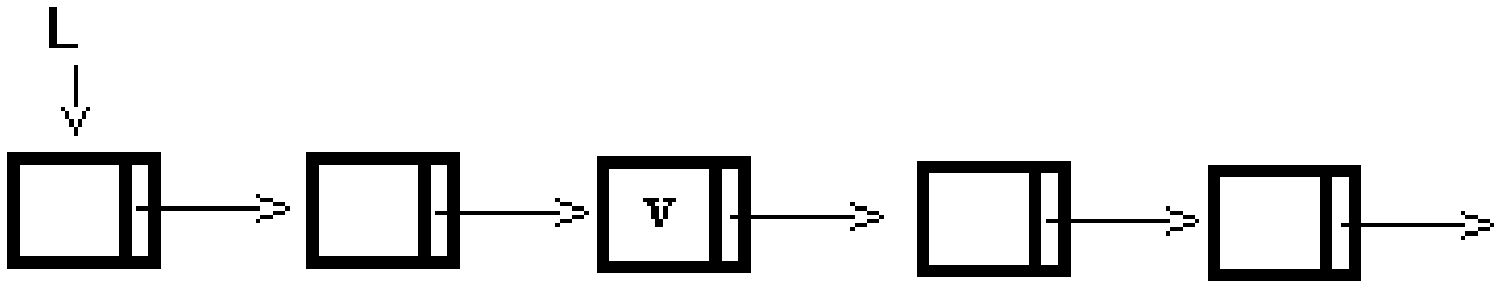
Implementação das Operações

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

```
void Elimina_Depois(Lista *L, Rec *k){

    Rec *j = k->lig;

    k->lig = j->lig;
    free(j);
     L->nelem--;
}
```

Implementação das Operações

9) **Eliminar elemento *v* de uma lista ordenada L**



Implementação das Operações

9) **Eliminar elemento *v* de uma lista ordenada L**

```
boolean Remove(Tipo_elem v, Lista*L){

    Rec *p = L->head;
    Rec *pa = NULL;

    while (p != NULL){

        if (p->elem.chave < v.chave){
            pa = p;
            p = p->lig;

        } else {

            if (p->elem.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){
    Rec *p;
    p = L->head;

    while (p != NULL){
        impr_elem(p->elem);
        p = p->lig;
    }
}

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

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;
```

Implementação das Operações

- 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:
  - Verifica se L1 está ordenada ou não (a ordem pode ser crescente ou decrescente)
  - Faça uma cópia da lista L1 em uma outra lista L2
  - Faça uma cópia da Lista L1 em L2, eliminando elementos repetidos
  - inverta L1 colocando o resultado em L2
  - inverta L1 colocando o resultado na própria L1
  - 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)