CES-11

Algoritmos e Estruturas de Dados

Listas encadeadas lineares

Carlos Forster
Carlos Alonso
Juliana Bezerra

CES-11

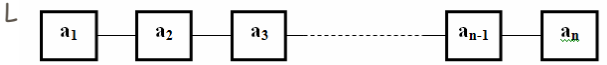
- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Definição

- *Lista linear* é uma sequência de zero ou mais elementos de um mesmo tipo.
- Simbolicamente, L = $a_1, a_2, \ldots, a_n$, onde $n \geq 0$.



- Suas posições obedecem uma ordem: a_i precede a_{i+1}, onde $0 < i \leq n-1$.
- Seus elementos podem ser consultados, inseridos ou eliminados em qualquer posição.
- Listas lineares distintas podem ser concatenadas ou repartidas.

Definição Recursiva

- L é uma lista encadeada se
  - L é vazia (NIL ou NULL), ou
  - L contém o primeiro elemento first(L) e uma lista com os demais elementos rest(L)
- Esta definição é útil para percorrer as listas recursivamente
  - Para percorrer a lista L, não vazia:
    - Visita-se o elemento first(L)
    - Percorre-se a sub-lista rest(L)
  - Para percorrer inversamente L não vazia:
    - Percorre-se inversamente a sub-lista rest(L)
    - Visita-se o elemento first(L)

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Linearização de Estruturas

- Considere a representação de uma matriz numérica na forma de um vetor linear

$$\begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

Exercício:

Definir get_matriz

[11 12 13 14 21 22 23 24 31 32 33 34]

```
void set_matriz(int m[],int i,int j,int valor)
{assert(i>=1 && i<=altura && j>=1 && j<=largura);
     m[(i-1)*largura+(j-1)]=valor;
}
```

Linearização de Estruturas

- Como determinar a posição do vetor linear correspondente a um elemento da estrutura?
  - Utiliza-se o tamanho da estrutura que contém os elementos anteriores.
  - Exemplo: matriz triangular inferior (notação compacta)

$$\begin{bmatrix} 11 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 \\ 31 & 32 & 33 & 0 \\ 41 & 42 & 43 & 44 \end{bmatrix}$$

[11 21 22 31 32 33 41 42 43 44]

  - Quantos elementos há na linha N? Resp: N
  - Quantos elementos há antes da linha N? Resp: N(N-1)/2

Linearização de Estruturas

- Definindo as funções de acesso para a matriz triangular inferior:

```
void set_matriz(int m[],int i,int j,int valor)
{
     assert(i>=1 && i<=altura
            && j>=1 && j<=i);
     m[(i*(i-1))/2+(j-1)]=valor;
}
```

Este é o número de elementos antes da linha i

i é o tamanho da linha

Exercícios: Definir get_matriz e modificar para aceitar acesso no caso j>i também (retorna zero).

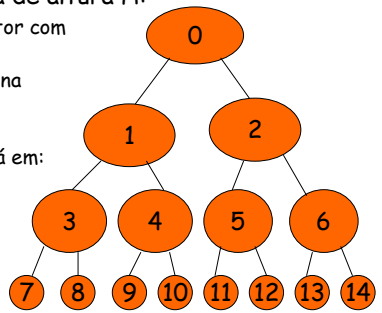
A altura pode vir definida em um struct junto ao apontador para o vetor dos dados dinamicamente alocado.

Linearização de Estruturas

- Exercício: Definir uma estrutura linear (estrutura de dados e funções de acesso) para as seguintes matrizes:
  - Matriz faixa
  - Matriz tri-diagonal
  - Matriz simétrica
  - Matriz anti-simétrica

Linearização de Estruturas

- Árvore binária completa de altura H:
  - Representa-se em um vetor com 2^H-1 espaços
  - O filho à esquerda do nó na posição i é: 2i+1
  - O filho à direita de i está em: 2i+2
  - O nó pai do nó n posição i está em: $\lfloor (i-1)/2 \rfloor$

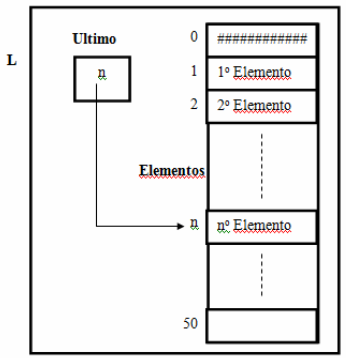


CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Implementação com vetor

- Um exemplo: lista linear de caracteres.



Permite a posição 0 desocupada

```
const int max = 50;

typedef char vetor[max+1];
typedef struct lista lista;
struct lista {
       vetor Elementos;
       int Ultimo;
};

lista L1, L2, L3;
```

Implementação com vetor - Criação

```
lista entrarLista () {
   lista L; int i;
   write ("Digite o numero de elementos: ");
   read (L.Ultimo);
   if (L.Ultimo > max) {
        Erro ("Numero excede tamanho maximo para listas");
        L.Ultimo = 0;
   }
   else if (L.Ultimo > 0) {
        write ("Digite ", L.Ultimo, " elemento(s): ");
        for (i = 1; i <= L.Ultimo; i++)
              read (L.Elementos[i]);
   }
   return L;
}
```

```
void main () {
    lista L1;
    L1 = entrarLista ();
}
```

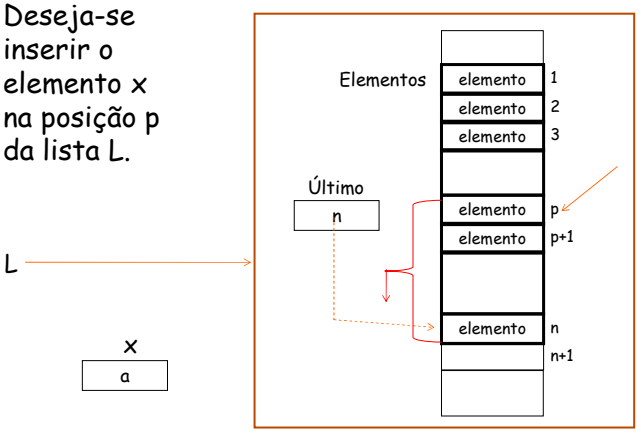
Implementação com vetor - Impressão

```
void escreverLista (lista L) {
   int i;
   if (L.Ultimo < 1) write ("Lista vazia");
   else
        for (i = 1; i <= L.Ultimo; i++)
              write (L.Elementos[i]);
}
```

```
void main () {
       lista L1;
       - - - - -
       escreverLista (L1);
}
```

Implementação com vetor - Inserção

- Deseja-se inserir o elemento x na posição p da lista L.



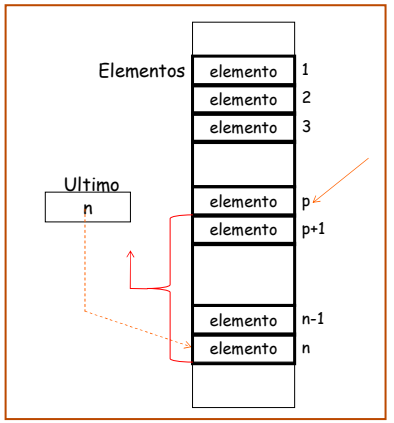
Implementação com vetor - Inserção

```
void inserirElemento (char x, int p, lista *L) {
   int q;
   if (L->Ultimo >= max)
        Erro ("Lista cheia: insercao impossivel");
   else if (p < 1 || p > L->Ultimo + 1)
        Erro ("A posicao de insercao nao existe");
        else {
               L->Ultimo++;
               for (q = L->Ultimo - 1; q >= p; q--)
                     L->Elementos[q+1] = L->Elementos[q];
               L->Elementos[p] = x;
               }
}
```

No pior caso, tempo gasto será proporcional ao tamanho da lista

Implementação com vetor - Eliminação

- Deseja-se eliminar o elemento da posição p da lista L.



Implementação com vetor - Eliminação

```
void eliminarElemento (int p, lista *L) {
   int q;
   if (p < 1 || p > L->Ultimo)
        Erro ("A posicao de eliminacao nao existe");
   else {
        L->Ultimo--;
        for (q = p; q <= L->Ultimo; q++)
               L->Elementos[q] = L->Elementos[q+1];
   }
}
```

No pior caso, tempo gasto será proporcional ao tamanho da lista

Implementação com vetor - Busca

- Deseja-se encontrar a posição do elemento x na lista L.

```
int buscarElemento (char x, lista L) {
   int posic = -1, q = 1;
   while (q <= L.Ultimo) {
      if(L.Elementos[q] != x)
         q++;
       else {
         posic = q;
          break;
       }
    }
    return posic;
}
```

No pior caso, tempo gasto será proporcional ao tamanho da lista

Avaliação em curto-circuito

```
int buscarElemento (char x, lista L) {
   int q = 1;
   while (q <= L.Ultimo && L.Elementos[q] != x) q++;
   if (q > L.Ultimo) return -1;
   else return q;
}
```

Reservar espaço extra

```
int buscarSentinela (char x, lista L) {
   int q = 1;
   L.Elementos[L.Ultimo+1] = x;
   while (L.Elementos[q] != x) q++;
   if (q > L.Ultimo) return -1;
   else return q;
}
```

Implementação com vetor

- Outras operações:
  - Encontrar em L o p-ésimo elemento
    - `L->Elementos[p]`
  - Encontrar em L o elemento sucessor de p
    - `L->Elementos[p+1]`
  - Encontrar em L o elemento anterior a p
    - `L->Elementos[p-1]`
  - Esvaziar a lista L
    - `L->Ultimo = 0`

Todas essas operações podem ser realizadas em *tempo constante*

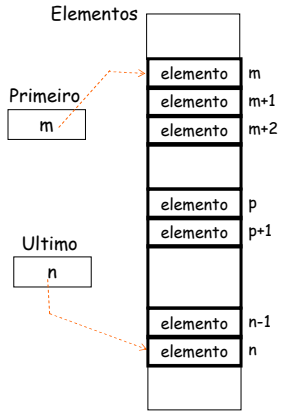
Implementação com vetor

- Uma implementação alternativa

```
const int max = 50;

typedef char vetor[max+1];

struct lista {
       vetor Elementos;
       int Primeiro, Ultimo;
};
```

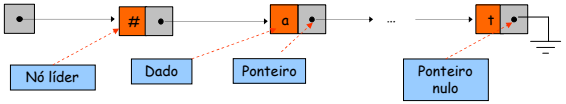


CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Implementação com nós encadeados

- Os elementos da lista são armazenados num encadeamento de estruturas com ponteiros:



- Cada uma dessas estruturas:
  - recebe o nome de nó;
  - armazena um elemento e um ponteiro para o próximo nó.
- Para simplificar os códigos, o primeiro nó *costuma* ser definido como líder, e não armazena nenhum elemento.
- Esta estrutura de dados é chamada de lista encadeada ou lista ligada (*linked list*).

Implementação com nós encadeados

- Mesmo exemplo: lista linear de caracteres.

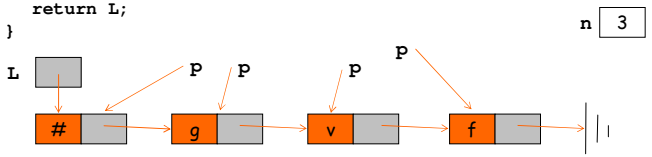

```
typedef struct node node;
struct node {
      char elem;
      node *prox;
};
typedef node *lista;
typedef node *posicao;

lista L1, L2, L3;
L1 = NULL;
posicao p, q, r;
```

Agora lista é um ponteiro!

Implementação com nós - Criação

```
lista entrarLista () {
  int i, n; lista L; posicao p;
  L = (node *) malloc (sizeof (node));
  write ("Digite o numero de elementos: "); read (n);
  if (n > 0) {
      write ("Digite ", n, " elemento(s): ");
      for (p = L, i = 1; i <= n; i++) {
            p->prox = (node *) malloc (sizeof (node));
            p = p->prox; read (p->elem);
      }
  }
  p->prox = NULL;
  return L;
}
```



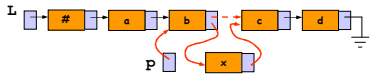
Implementação com nós - Impressão

```
void escreverLista (lista L) {
  posicao p;
  if (L->prox == NULL) write ("Lista vazia");
  else
      for (p = L->prox; p != NULL; p = p->prox)
            write (p->elem);
}
```

```
void main () {
      lista L1;
      - - - - -
      escreverLista (L1);
}
```

Implementação com nós - Inserção

- Deseja-se inserir o elemento x após o nó da lista L apontado por p.



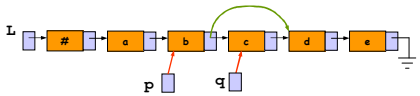

```
void inserirElemento (char x, posicao p, lista L) {
  posicao q;
  if (p == NULL) Erro ("Posicao nao existe");
  else {
       q = p->prox;
       p->prox = (node *) malloc (sizeof (node));
       p->prox->elem = x;
       p->prox->prox = q;
  }
}
```

Desnecessário, pois p é ponteiro

Inserção pode ser realizada em *tempo constante*

Implementação com nós - Eliminação

- Deseja-se eliminar o elemento da lista L que está após o nó apontado por p.




```
void eliminarElemento (posicao p, lista L){
  posicao q;
  if (p == NULL || p->prox == NULL)
       Erro ("Posicao nao existe");
  else {
       q = p->prox;
       p->prox = q->prox;
       free (q);
  }
}
```

Desnecessário, pois p é ponteiro

Eliminação também pode ser realizada em *tempo constante*

Implementação com nós - Esvaziamento

```
void esvaziarLista (lista L) {
  posicao p;
  if (L == NULL)
       Erro ("A lista nao foi inicializada");
  else {
       while (L->prox != NULL) {
            p = L->prox;
            L->prox = L->prox->prox;
            free (p);
       }
  }
}
```

Vai restar apenas o nó lider

```
void main () {
      lista L1;
      - - - - -
      esvaziarLista (L1);
}
```

Implementação com nós encadeados

- Outras operações:
  - Encontrar em L o elemento sucessor do nó apontado por p
    - `p->prox->elem`
  - Encontrar em L o p-ésimo elemento
    - De modo geral, será preciso percorrer a lista
  - Encontrar em L o elemento anterior a p
    - Idem: também será preciso percorrer a lista

As duas últimas operações acima, no pior caso, gastam *tempo linear* em relação ao tamanho da lista

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Comparação: tempo

- Podemos comparar as duas implementações de listas lineares de n elementos em termos de *tempo de pior caso* na execução das suas principais operações:

Operações	*Vetor*	*Nós encadeados*
Criação	O(n)	O(n)
Impressão	O(n)	O(n)
Inserção	O(n)	O(1)
Eliminação	O(n)	O(1)
Busca	O(n)	O(n)
Esvaziamento	O(1)	O(n)
p-ésimo	O(1)	O(n)
Sucessor	O(1)	O(1)
Anterior	O(1)	O(n) ?

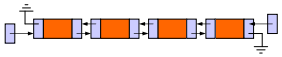
Comparação: espaço

- Em relação ao gasto de memória:
  - Vetores utilizam espaço constante, o que pode ser desvantajoso para listas pequenas
  - Listas encadeadas exigem espaço extra para o armazenamento de ponteiros
- No caso das listas encadeadas, com um gasto extra de memória (um ponteiro para o nó anterior), é possível tornar constante o tempo de acesso ao elemento anterior.
  - Esta estrutura é chamada de *lista duplamente encadeada*

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Listas duplamente encadeadas



```
struct node {
      char elem;
      node *prox, *prev;
};

typedef node *lista;
typedef node *posicao;
```

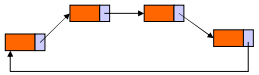
- Também será mantido um ponteiro para o final da lista.
- Vantagem: acesso ao anterior em tempo constante.
- Desvantagens: maior consumo de memória, código um pouco mais complicado (atualização de mais ponteiros).

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Listas circulares

- Determinadas aplicações exigem outra variante: uma *lista encadeada circular*.



- Nesses casos, deixa de haver a noção de primeiro da lista, dispensando-se o uso de um nó líder.
- Basta manter um ponteiro L para algum dos elementos da lista.
- As listas circulares também podem ser duplamente encadeadas.

Diferenças

Exercício: modificar para eliminar todas as ocorrências de elt

- Busca e remoção do primeiro elemento encontrado: lista ligada simples sem cabeça

```
void search_and_destroy(char elt,lista* L)
{     posicao p,q;
      if(*L==NULL) return; /*lista vazia*/
      if((*L)->elem==elt)
      {     p=*L;
            *L=(*L)->prox;
            free(p); return; /*remocao do primeiro*/
      }
      /* a partir daqui não precisa modificar (*L) */
      p=*L; q=p->prox;
      while(q!=NULL && q->elem!=elt) {p=q; q=q->prox;}
      if (q==NULL) return; /*não encontrou*/
      p->prox=q->prox;
      free(q);
}
```

Diferenças

Exercício: modificar para eliminar todas as ocorrências de elt

- Busca e remoção do primeiro elemento encontrado: lista ligada simples com cabeça

```
void search_and_destroy(char elt,lista L)
{     posicao p,q;

      p=L; q=L->prox; /*pula a cabeça*/
      while(q!=NULL && q->elem!=elt) {p=q; q=q->prox;}
      if (q==NULL) return; /*não encontrou*/
      p->prox=q->prox;
      free(q);
}
```

Não precisa passar lista por referência

Diferenças

Exercício: modificar para eliminar todas as ocorrências de elt

- Busca e remoção do primeiro elemento encontrado: lista duplamente ligada com cabeça

```
void search_and_destroy(char elt,lista L)
{     posicao p,q;

      p=L; q=L->prox;
      while(q!=NULL && q->elem!=elt) {p=q; q=q->prox;}
      if (q==NULL) return; /*não encontrou*/
      p->prox=q->prox;
      q->prox->prev=p;
      free(q);
}
```

Não precisa passar lista por referência

Diferenças

Exercício: modificar para eliminar todas as ocorrências de elt

- Busca e remoção do primeiro elemento encontrado: lista circular com cabeça

```
void search_and_destroy(char elt,lista L)
{     posicao p,q;

      assert(L->elem=='#');
      p=L; q=L->prox; /*pula a cabeça*/
      while(q->elem!=elt && q->elem!='#')
            {p=q; q=q->prox;}
      if (q->elem=='#') return; /*não encontrou*/
      p->prox=q->prox;
      free(q);
}
```

Não precisa passar lista por referência

CES-11

- Listas lineares
  - Definição
  - Linearização de Estruturas
  - Implementação com vetor
  - Implementação com nós encadeados
  - Comparação
  - Outras implementações
    - Listas duplamente encadeadas
    - Listas circulares
  - Exercícios

Exercícios

- Dadas duas listas lineares, deseja-se implementar a operação de *concatenação*, isto é, acrescentar a segunda lista no final da primeira. Qual seria a melhor implementação para realizar essa operação?
- Escreva o código das principais operações (criação, impressão, inserção, eliminação e esvaziamento) em uma:
  - lista encadeada sem nó líder;
  - lista duplamente encadeada;
  - lista encadeada circular (sem nó líder).

Exercícios

Dica: as funções recursivas devem ser aplicadas à lista simples sem cabeça

- Definir funções recursivas para:
  - Retornar uma lista dado o primeiro elemento e uma lista com os demais elementos;
  - Criar uma cópia de uma lista ligada;
  - Buscar um elemento de uma lista ligada (retorna o apontador para o nó com o elemento ou NULL);
  - Imprimir a lista de trás para frente;
  - Inserir elemento no fim da lista;
  - Obter a lista em ordem inversa;
  - Dividir a lista em uma com elementos menores que o valor dado e outra com os maiores ou iguais;
  - Combinar duas listas ordenadas numa só (ordenada);
  - Ordenar inserindo um elemento de cada vez.

Exercícios

- Definir funções iterativas para:
  - Inverter a lista (com inversão de apontadores)
  - Ordenar a lista selecionando o maior elemento e removendo da lista, até esgotar a lista.

Inversão da Lista

- Técnica da inversão de apontadores