Listas, pilhas e filas

Profa. Vânia de Oliveira Neves
Slides baseados no material da Profa. Graça Nunes

Listas

Lista

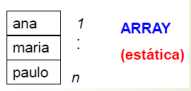
- Uma Lista Linear é uma coleção ordenada de componentes de um mesmo tipo.
- Ela é
  - ou vazia
  - ou pode ser escrita como
    $(a_1, a_2, ..., a_n)$.
  - onde
    - a_i são átomos de um mesmo conjunto S;
    - a_1 é o primeiro elemento;
    - a_i precede a_{i+1};
    - a_n é o último elemento da lista
- Ex. listas de nomes, de peças, de valores, de pessoas, de compras, etc.

Lista

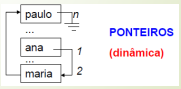
- Estrutura Homogênea: elementos de um mesmo tipo base
- Uma lista pode ser ordenada (campo "chave") ou não-ordenada
  - Em geral, é ordenada segundo algum critério...
  - Operações básicas:
    - Verificar se lista vazia;
    - Inserção de elemento na lista;
    - Eliminação de elemento da lista;
    - Busca (Acesso)por um elemento, dada uma chave ou uma posição
  - Outras operações:
    - ordenar, concatenar, inverter, etc.

Lista - Representação

- **Seqüencial:** sucessor lógico de um elemento ocupa posição física consecutiva na memória (endereços consecutivos)
- L = (ana, maria, paulo)

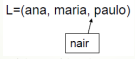


- **Encadeada:**elementos logicamente consecutivos não implicam elementos (endereços) consecutivos na memória
- L = (ana, maria, paulo)

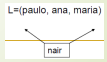


Lista - Ordenação

- Qualquer que seja o tipo de representação, a lista pode diferir quanto à ordenação:
  - Ordenada: elementos ordenados segundo valores do campo chave e, eventualmente, de outros campos
    - inserção é feita em local definido pela ordenação



    - Se sequencial, os registros devem ser deslocados → maior tempo
  - Não ordenada
    - inserção ocorre nas extremidades (mais barato)



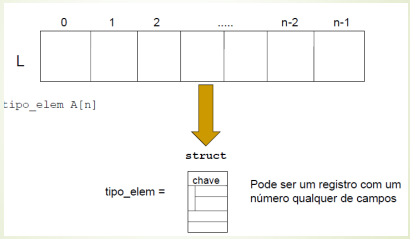
Lista linear

- **Escolher** entre uma ou outra representação (sequencial ou encadeada) vai depender do comportamento da lista na aplicação (tamanho, operações mais frequentes, etc.).
- A **eficiência** das operações depende também da representação usada e de outros fatores: se a lista está ordenada, se é grande ou pequena, etc.

Lista – Implementação Sequencial Estática

- As escolhas dos tipos de dados a seguir valem tanto para lista **ordenada** como **não ordenada**. O que vai diferir no caso de não ordenada são as funções de inserção, busca e remoção.
- No caso da combinação Sequencial Estática, usa-se o **ARRAY** para armazenar os elementos da lista.

Lista – Tipo de dado



Lista - Implementação

```
#define MAX 100  /*estimativa do tamanho máximo da lista*/
#define TRUE 1   /*define tipo boleano - não existe na linguagem C*/
#define FALSE 0
#define boolean int

typedef int tipo_chave;  /*tipo da chave - deve admitir comparações*/

typedef struct{  /*tipo registro*/
    char nome[30];
    // ...
}tipo_info;

typedef struct{  /*tipo elemento*/
    tipo_chave chave;
    tipo_info info;
}tipo_elem;

typedef struct{
    int nelem;  /*número atual de elementos*/
    tipo_elem A[MAX];
}Lista;

Lista L;  /*exemplo de declaração*/
```

Lista - Implementação

```
boolean Vazia(Lista *L){
   /*Retorna true (1) se lista vazia, false (0) caso contrário*/
   return (L->nelem == 0);
}

boolean Cheia(Lista *L){
   /*Retorna true (1) se lista cheia, false (0) caso contrário*/
   return (L->nelem == MAX);
}

void Definir(Lista *L){
   /*Cria uma lista vazia. Este procedimento deve ser chamado para
   cada nova lista antes de qualquer outra operação.*/
   L->nelem = 0;
}

void Apagar(Lista *L){
   /*Apaga logicamente uma lista*/
   L->nelem = 0;
}
```

Lista - Implementação

```
boolean Inserir_posic(tipo_elem x, int p, Lista *L){
   /* Insere novo elemento, x, na posição p da Lista.
   Se L = a1, a2,... an então temos a1, a2,...ap-1 x ap+1 ... an.
   Devolve true se sucesso, false c.c. (L não tem nenhuma posição p
   ou Lista cheia). Operação para LISTA NÃO ORDENADA! */

   int q;

   if (Cheia(L) || p > L->nelem+1 || p < 1){
        /* lista cheia ou posição não existe */
        return FALSE;

   } else {
        for(q=L->nelem; q>=p;q--)
                L->A[q] = L->A[q-1];

        L->A[p-1] = x; /* o p-ésimo elemento ocupa posição p-1 */
        L->nelem++;
        return TRUE; /* inserção com sucesso */
   }
}
```

Lista - Implementação

```
boolean Inserir_ord(tipo_elem x, Lista *L){
   /*Insere novo elemento de forma a manter a Lista ordenada
   (crescente). Devolve true se sucesso, false caso contrário
    Supõe a inserção de registros com chave repetida*/

   if (Cheia(L) return FALSE;

   if (Vazia(L)) /* insere 1°. elemento na posição 1 */
        return Inserir_posic(x, 1, L);

   else {    /*acha posição de inserção*/
        int i = 0;
        while (i <= L->nelem-1)
          if (x.chave <= L->A[i].chave)
                return Inserir_posic(x,i+1,L);
          else  i++;

        return Inserir_posic(x,i+1,L); /*insere na última posição*/
   }
}
```

Lista - Implementação

```
boolean Inserir_ord(tipo_elem x, Lista *L){
   /*Insere novo elemento de forma a manter a Lista ordenada
   (crescente). Devolve true se sucesso, false caso contrário
    Supõe ERRO ao inserir registros com chave repetida*/

   if (Cheia(L) return FALSE;

   if (Vazia(L)) /* insere 1°. elemento na posição 1 */
        return Inserir_posic(x, 1, L);

   else {    /*acha posição de inserção*/
        int i = 0;
        while (i <= L->nelem-1)
          if (x.chave < L->A[i].chave)
                return Inserir_posic(x,i+1,L);
          else  if (x.chave == L->A[i].chave)
                      return FALSE;   /* já está na lista */
                else i++;

        return Inserir_posic(x,i+1,L); /*insere na última posição*/
   }
}
```

Lista - Implementação

```
boolean Buscar(tipo_chave x, Lista *L, int *p){
   /*Retorna true se x ocorre na posição p. Se x ocorre mais de
   uma vez, retorna a posição da primeira ocorrência. Se x não
   ocorre, retorna false. Para Listas NÃO ORDENADAS*/
   /* Primeira implementação com busca linear simples*/
   if (!Vazia(L)){
        int i = 0;
        while (i <= L->nelem-1)
                if (L->A[i].chave == x){
                        *p = i;
                        return TRUE;
                } else
                        i++;
   }
   return FALSE; /*não achou*/
}
```

Lista - Implementação

```
boolean Buscar_ord(tipo_chave x, Lista *L, int *p){
   /*Retorna true se x ocorre na posição p. Se x ocorre mais de
   uma vez, retorna a posição da primeira ocorrência. Se x não
   ocorre, retorna false. Para Listas ORDENADAS*/    /*
   Implementação com busca linear simples*/
   if (!Vazia(L)){
        int i = 0;
        while (i <= L->nelem-1)
                if (L->A[i].chave >= x)
                        if (L->A[i].chave == x){
                                *p = i;
                                return TRUE;
                        } else
                                return FALSE;
                                /*achou maior, pode parar*/
                else
                        i++;
   }
   return FALSE; /*não achou*/
}
```

Lista – Implementação

```
boolean Busca_bin(tipo_chave x, Lista *L, int *p){
   /*Retorna em p a posição de x na Lista ORDENADA, e true. Se x não
   ocorre retorna false*/
   /* Implementação de Busca Binária */
   int inf = 0;
   int sup = L->nelem-1;
   int meio;
   while (!(sup < inf)){
        meio = (inf + sup)/2;
        if (L->A[meio].chave == x) {
                *p = meio; /*sai da busca*/
                return TRUE;
        } else {
                if (L->A[meio].chave < x)
                        inf = meio+1;
                else
                        sup = meio-1;
                }
   }
   return FALSE;
}
```

Lista - Implementação

Versão Recursiva da Busca Binária (booleana)

```
boolean Busca_bin_rec(tipo_chave x, Lista *L, int inf, int sup, int *p){
   /*Se encontrar x, retorna em p sua posição e true; false, c.c.*/
   if (inf > sup)
        return FALSE;
   else {
        int meio = (inf + sup)/2;
        if (L->A[meio].chave > x)
                return Busca_bin_rec(x, L, inf, meio-1, p);
        else if (L->A[meio].chave < x)
                return Busca_bin_rec(x, L, meio+1, sup, p);
        else { /*achou x na posição meio*/
                *p = meio;
                return TRUE;
        }
   }
}
```

Nro Max de Comparações: $\log_2(nelem)$

Função externa para o usuário (booleana)

```
boolean Busca_bin(tipo_chave x, Lista *L, int *p){
   return Busca_bin_rec(x, L, 0, L->nelem-1, p);
}
```

Lista - Implementação

```
void Remover_ch(tipo_chave x, Lista *L, boolean *removeu){
   /* Remover dada a chave. Retorna true, se removeu, ou
   false, c.c.*/
   int *p = malloc(sizeof(int));
   *removeu = FALSE;
   if (Busca_bin(x, L, p)){  /*procura por busca binária*/
        Remover_posic(p, L);
        *removeu = TRUE;
   }
}
```

Lista - Implementação

```
void Remover_posic(int *p, Lista *L){
   /* Só é ativada após a busca ter retornado a posição p
   do elemento a ser removido - Nro de Mov = (nelem – p)*/
   int i;
   for (i = *p+1;  i < L->nelem-1; i++)
        L->A[i-1] = L->A[i];

   L->nelem--;
}
```

No programa, se quero eliminar o registro com chave x

```
boolean sucesso
...
Remover_ch(x, L, &sucesso);
if  (!sucesso) ... /*x não está na lista*/
else ... /*removeu*/
```

Lista - Implementação

```
void Imprimir(Lista *L){
  /*Imprime os elementos na sua ordem de precedência*/
  int i;
  if (!Vazia(L))
    for (i = 0; i< L->nelem-1; i++)
        Impr_elem(L->A[i]);
}

void Impr_elem(tipo_elem t){
   printf("chave: %d", t.chave);
   printf("info: %s", t.info.nome);
   //...
}

int Tamanho(Lista *L){
  /* Retorna o tamanho da Lista. Se L é vazia retorna 0 */
  return L->nelem;
}
```

Lista linear sequencial estática Resumo

- Vantagens:
  - Acesso Direto a cada elemento:
    - Lista.A[i] e(Lista->A[i])
  - Tempo Constante (decorrência do array)
    - independente do valor de i
  - Busca pode ser Binária, se Lista Ordenada (decorrência do array) – **O(log2(nelem))**
- Desvantagens:
  - Movimento de dados na Inserção e Eliminação, se Lista Ordenada
  - Tamanho máximo da lista é delimitado a priori (decorrência do array)
    - risco de overflow

Lista linear sequencial estática Resumo

- Quando optar por ela?
  - listas pequenas → custo insignificante
  - conhecimento prévio do comportamento da lista:
    - poucas inserções/eliminações (ou "comportadas")

Pilhas

Pilhas

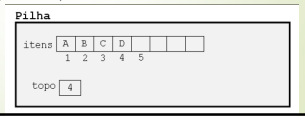
- *Estrutura para armazenar um conjunto de elementos que funciona da seguinte forma*
  - Novos elementos sempre entram no "topo" da pilha
  - O único elemento que se pode retirar da pilha em um dado momento é o elemento do topo
- Para que serve
  - Para modelar situações em que é preciso "guardar para mais tarde" vários elementos e "lembrar" sempre do último elemento armazenado
- L.I.F.O.
  - *Last In, First Out*

Pilha

- Operações usuais
  - Push(P,X): empilha o valor da variável X na pilha P
  - Pop(P,X): desempilha P e retorna em X o valor do elemento que estava no topo de P
  - X=top(P): acessa o valor do elemento do topo de P, sem desempilhar
  - Create(P): cria uma pilha vazia P
  - IsEmpty(P): é *true* se a pilha estiver vazia; *false* caso contrário
  - Empty(P): esvazia logicamente uma pilha P

Implementação

- Alocação sequencial
  - Os elementos da pilha ficam, necessariamente, em sequência (um ao lado do outro) na memória
- Alocação estática
  - Todo o espaço de memória a ser utilizado pela pilha é reservado (alocado) em tempo de compilação
  - Todo o espaço reservado permanece reservado durante todo o tempo de execução do programa, independentemente de estar sendo efetivamente usado ou não



Declaração em C

```
#define TamPilha 100
typedef int elem;

typedef struct {
    int topo;
    elem A[TamPilha];
} Pilha;

Pilha P;
```

Implementação

```
#define TRUE 1 /*define tipo booleano*/
#define FALSE 0
#define boolean int

void Create(Pilha *P) { /*cria uma pilha vazia P */
      P->topo=-1;
      return; }
void Empty(Pilha *P) { /* esvazia logicamente a pilha P */
      P->topo=-1;
      return; }
boolean IsEmpty(Pilha *P) {/*retorna true se a pilha estiver vazia; false caso contrário */
      return (P->topo==-1);
}
```

Implementação

```
boolean IsFull(Pilha *P) { /* verifica se P cresceu até o final do array */
      return (P->topo==TamPilha-1);
}
boolean Push(Pilha *P, elem *X) { /*empilha o valor da variável X na pilha P */
      if (!IsFull(P)) {
            P->topo++;
            P->A[P->topo]=*X;
            return TRUE;
      }
      return FALSE;
}
```

Implementação

```
boolean Pop(Pilha *P, elem *X) { /*desempilha P e retorna em X o valor do elemento
que estava no topo de P; retorna TRUE se sucesso; FALSE, c.c. */
      if (!IsEmpty(P)) {
            *X=P->A[P->topo];
            P->topo--;
            return TRUE;
      }
      return FALSE;
}
elem Top(Pilha *P) { /* retorna o valor do elemento do topo de P, sem desempilhar –
chamada apenas se Pilha P não estiver vazia!!! */
      return P->A[P->topo];
}
```

Filas

Filas

- *É uma estrutura para armazenar um conjunto de elementos, que funciona da seguinte forma*
  - Novos elementos sempre entram no fim da fila
  - O único elemento que se pode retirar da fila em um dado momento é seu primeiro elemento
- Para que serve?
  - Modelar situações em que é preciso armazenar um conjunto ordenado de elementos, no qual o primeiro elemento a entrar no conjunto será também o primeiro elemento a sair do conjunto, e assim por diante
- F.I.F.O
- *First In, First Out*

Filas

- Operações
  - Cria(F): cria uma fila F vazia
  - Entra(F,X): X entra no fim da fila F
  - Sai(F,X): o primeiro elemento da fila F é retirado da fila e atribuído a X
  - IsEmpty(F): verdade se a fila estivar vazia; caso contrário, falso
  - IsFull(F): verdade se a fila estiver cheia; caso contrário, falso

Filas - Exemplo

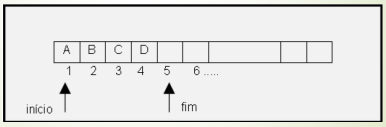
operação	fila	resultado
cria(F)	1º DA FILA -->	
entra(F, a)	1º DA FILA --> a	
entra(F, b)	1º DA FILA --> a, b	
entra(F, c)	1º DA FILA --> a, b, c	
sai(F, X)	1º DA FILA --> b, c	X = a
entra(F, d)	1º DA FILA --> b, c, d	
sai(F, X)	1º DA FILA --> c, d	X = b

Implementação da fila

- **Alocação sequencial**
  - Os elementos da fila ficam, necessariamente, em sequência (um ao lado do outro) na memória
- **Alocação estática**
  - Todo o espaço de memória a ser utilizado pela fila é reservado (alocado) em tempo de compilação
  - Todo o espaço reservado permanece reservado durante todo o tempo de execução do programa, independentemente de estar sendo efetivamente usado ou não

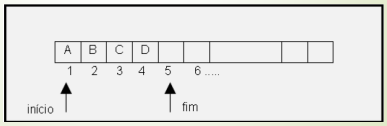
Implementação da fila

- **Início** aponta para/indica o primeiro da fila, ou seja, o primeiro elemento a sair
- **Fim** aponta para/indica o fim da fila, ou seja, onde o próximo elemento entrará



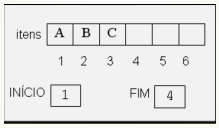
Implementação da fila

- Qual a condição inicial, quando a fila é criada?
  - Início=0, fim=1
- Qual a condição para fila vazia?
  - Início=0, fim=1 ?
- Qual a condição para fila cheia?
  - fim=tamanho do array+1 ?



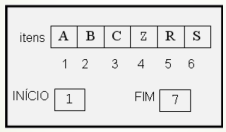
Criação da fila

entra(F,A), entra(F,B), entra(F,C)

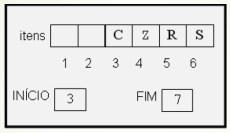


entra(F,Z), entra(F,R), entra(F,S)

- IsFull=TRUE



sai(F,X), sai(F,X)

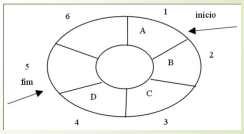


- Como inserir mais elementos?
- Qual o problema com a fila?

Fila

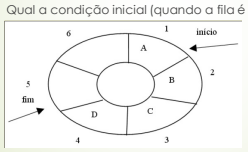
- Como reutilizar os espaços do início da fila?
  - Outra forma de implementação
  - Melhor aproveitamento da representação utilizada

Fila vista como um anel!



Fila como um anel

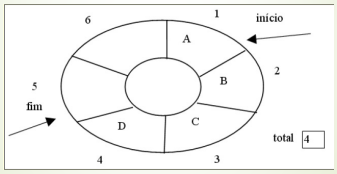
- Qual a condição para fila vazia?
- Qual a condição para fila cheia?
- Qual a condição inicial (quando a fila é criada)?



Difícil! Perde-se um pouco do sentido com essa representação

Fila como um anel

- **Solução**: campo extra para guardar número de elementos

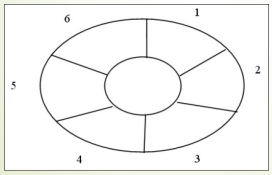


Fila como um anel

- Qual a condição para fila vazia?
  - Total=0
- Qual a condição para fila cheia?
  - Total=tamanho da fila
- Qual a condição inicial (quando a fila é criada)?
  - Total=0, início=1, fim=1

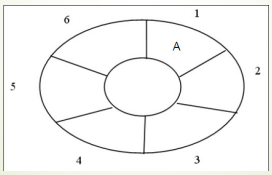
Exemplo

- Fila criada
  - início=1, fim=1, total=0



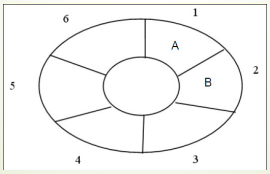
Exemplo

- Entra A
  - início=1, fim= fim+1=2, total=1



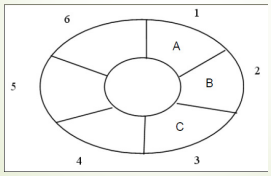
Exemplo

- Entra B
  - início=1, fim=fim+1=3, total=2



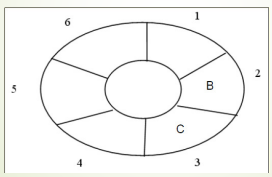
Exemplo

- Entra C
  - início=1, fim=fim+1=4, total=3



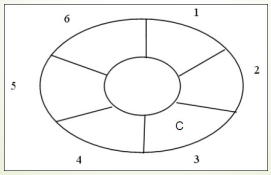
Exemplo

- Sai primeiro
  - início= início+1=2, fim=4, total=2



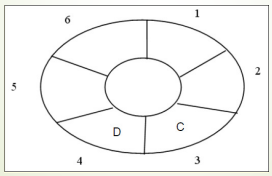
Exemplo

- Sai primeiro
  - início=início+1=3, fim=4, total=1



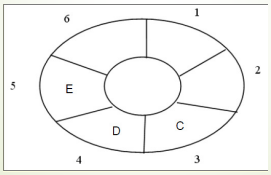
Exemplo

- Entra D
  - início=3, fim=fim+1=5, total=2



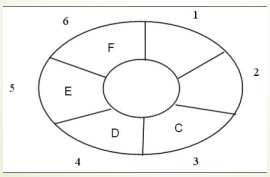
Exemplo

- Entra E
  - início=3, fim=fim+1=6, total=3



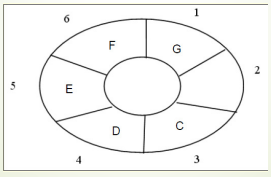
Exemplo

- Entra F
  - início=3, fim= (fim+1) % 6 =1, total=4



Exemplo

- Entra G
  - Início=3, fim=fim+1=(fim+1) % 6 =2, total=5



Passo a passo para *Entra* e *Sai*

- Entra elemento no fim da fila
  - Se não estiver cheia (Total = tamanho array):
  - vetor[fim]=elemento
  - avança fim ("módulo tamanho do array" para "fazer a curva", se preciso)
  - atualiza total
- Sai primeiro elemento
  - Se não estiver vazia (Total ≠ 0):
  - elemento=vetor[início]
  - avança início ("módulo tamanho do array" para "fazer a curva", se preciso)
  - atualiza total

Implementação em C

```
#define TamFila 100
typedef int elem;
typedef struct {
    int inicio, fim, total;
    elem itens[TamFila];
} Fila;
Fila F;
```

Implementação – Fila como anel

```
#define TRUE 1 /*define tipo boleano*/
#define FALSE 0
#define boolean int
void Create(Fila *F) {/* inicializa fila F como vazia */
    F->inicio=0;
    F->fim=0;
    F->total=0;
    return;
}
void Empty(Fila *F) { /* esvazia logicamente a fila F */
    F->inicio=0;
    F->fim=0;
    F->total=0;
    return;
}
```

Implementação – Fila como anel

```
boolean IsEmpty(Fila *F) { /*verifica se F está vazia */
    return (F->total==0);
}
boolean IsFull(Fila *F) {/*verifica se array está cheio */
    return (F->total==TamFila-1);
}
boolean Entra(Fila *F, elem *X) { /*insere X no início da Fila F, se não estiver cheia*/
    if (!IsFull(F)) {
        F->total++;
        F->A[F->fim]=*X;
        F->fim = (F->fim +1) % TamFila;
    return TRUE;
    }
    return FALSE;
}
```

Implementação – Fila como anel

```
boolean Sai(Fila *F, elem *X) { /*elimina do início da fila e copia valor eliminado em X */
    if (!IsEmpty(F)) {
        F->total--;
        *X=F->A[F->inicio];
        F->inicio = (F->inicio +1) % TamFila;
        return TRUE;
    }
    return FALSE;
}
```