

Estruturas de dados com alocação dinâmica de memória

Assuntos:

- Introdução
 - Alocação dinâmica de memória
- Revisão sobre ponteiros em Pascal
 - Manipulação de ponteiros
- Listas encadeadas
 - Lista encadeada simples
 - Lista encadeada dupla
 - Operações com listas
 - Aplicações de listas
 - Pilhas e filas em listas encadeadas
 - Fila circular encadeada
- Árvores
 - Conceitos e propriedades
 - Árvore binária

Estruturas de dados com alocação dinâmica de memória

Introdução:

Pergunta: Que fatores devem ser levados em conta na escolha de uma estrutura estática ou dinâmica ?

Estruturas de dados estáticas:

- Necessitam que uma certa quantidade de memória seja reservada na programação.
- A quantidade de memória reservada pode ser sub-utilizada ou insuficiente.
- Utilização de Arranjos, matrizes, vetores, possibilitam seu uso em quase todas as linguagens de programação
- Acesso direto aos dados por meio de mapeamento direto índice -> informação. Implementação simples.
- Constante necessidade de realocação de informações.

Estruturas de dados dinâmicas:

- São utilizadas quando não se pode definir inicialmente a quantidade de informação a ser armazenada – não se pode definir a quantidade de memória a ser reservada pelo programa
- A quantidade de memória é determinada em tempo de execução, não havendo sub-utilização de memória reservada e o limite de memória é dado pelos recursos computacionais.
- Utilização de Ponteiros limita seu uso a algumas linguagens (???)
- Acesso a informação não é direto. Implementação necessita de cuidados especiais do programador.
- A realocação de informações é substituída pelo redirecionamento de ponteiros.

Estruturas de dados com alocação dinâmica de memória

Uso de ponteiros em Pascal:

- Os ponteiros nos habilitam a representar estruturas de dados complexas e trabalhar com memória alocada dinamicamente. Trabalhar com memória alocada dinamicamente significa alocar memória quando necessário e liberar (ou desalocar) a memória quando não for mais utilizada.
- Ponteiros são uma forma indireta de acessar o valor de um dado item em particular – indicam uma região de memória reservada para este item.

Exemplo:

```
program pointers1;
type    int_pointer = ^integer;

var     iptr : int_pointer;

begin
    new( iptr );
    iptr^ := 10;
    writeln('the value is ', iptr^);
    dispose( iptr );
end.
```

Estruturas de dados com alocação dinâmica de memória

A linha

```
type    int_pointer = ^integer;
```

declara um novo tipo de variável chamada `int_pointer`, que é um ponteiro (denotado por `^`) para um inteiro.

A linha

```
var     iptr : int_pointer;
```

declara uma variável chamada `iptr` do tipo `int_pointer`. `iptr` não armazena um valor numérico e sim um endereço de memória

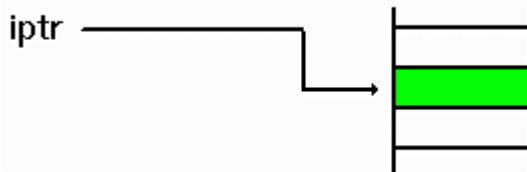


A diagram showing the variable `iptr` with a horizontal arrow pointing to the right, indicating it points to a memory location.

A linha

```
new( iptr );
```

cria uma variável dinâmica em tempo de execução. O ponteiro `iptr` aponta para uma área de memória alocada para armazenar um inteiro.

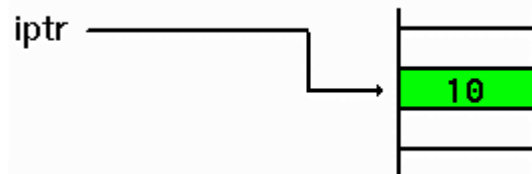


Estruturas de dados com alocação dinâmica de memória

A linha

```
iptr^ := 10;
```

significa: vá para a área de memória apontada por `iptr` e escreva o valor 10



A linha

```
writeln('the value is ', iptr^);
```

busca o valor da área de memória apontada por `iptr` e o imprime na tela.

A linha

```
dispose( iptr );
```

libera a memória associada com `iptr`.



Estruturas de dados com alocação dinâmica de memória

Quando o ponteiro não referencia qualquer região de memória, deve ser atribuído a ele o valor `nil`.

```
program pointers2;
type    int_pointer = ^integer;

var     iptr : int_pointer;
begin
  new( iptr );
  iptr^ := 10;
  writeln('the value is ', iptr^);
  dispose( iptr );
  iptr := nil;
  if iptr = nil
  then writeln('iptr does not reference any variable')
  else
  writeln('The value of the reference for iptr is ', iptr^);
end.
```

A linha

```
    iptr := nil;
```

atribui o valor **nil** para a variável ponteiro `iptr`. Isto significa que o ponteiro é válido e ainda existe, mas não aponta para nenhuma localização de memória ou variável dinâmica.

A linha

```
    if iptr = nil
```

testa `iptr` para ver se é um *ponteiro nil*, isto é, se não está apontando para nenhum ponto válido.

A atribuição **nil** e este teste é muito útil na construção de estruturas de dados mais elaboradas.

Estruturas de dados com alocação dinâmica de memória

Ponteiros do mesmo tipo podem ser igualados e ter os valores atribuídos para os dois ao mesmo tempo.

Exemplo:

```
program pointers3( output );
  type    int_pointer = ^integer;

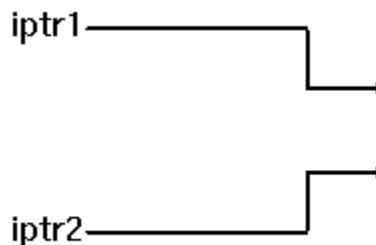
  var     iptr1, iptr2 : int_pointer;
begin
  new( iptr1 );
  new( iptr2 );
  iptr1^ := 10;
  iptr2^ := 25;
  writeln('the value of iptr1 is ', iptr1^);
  writeln('the value of iptr2 is ', iptr2^);
  dispose( iptr1 );
  iptr1 := iptr2;
  iptr1^ := 3;
  writeln('the value of iptr1 is ', iptr1^);
  writeln('the value of iptr2 is ', iptr2^);
  dispose( iptr2 );
end.
```

A linha

```
var     iptr1, iptr2 : int_pointer;
```

Cria dois ponteiros do tipo *int_pointer* (inteiro): *iptr1* e *iptr2*. Eles não estão associados com qualquer endereço de memória, o que acontece com as linhas:

```
new( iptr1 );
new( iptr2 );
```

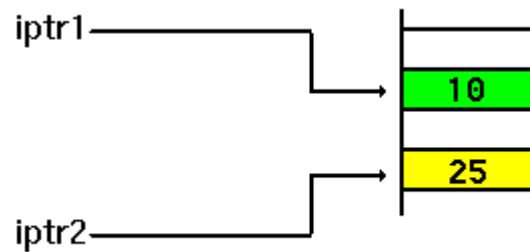


Estruturas de dados com alocação dinâmica de memória

As linhas

```
iptr1^ := 10;  
iptr2^ := 25;
```

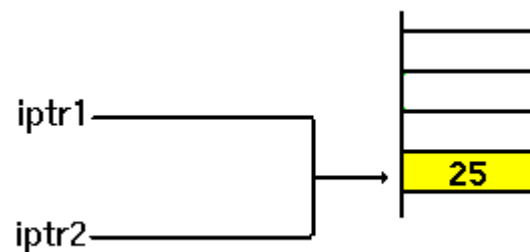
atribuem valores para os endereços alocados aos ponteiros.



As linhas

```
dispose( iptr1 );  
iptr1 := iptr2;
```

liberam a memória associada a *iptr1* e a seguir faz *iptr1* apontar para o mesmo endereço que *iptr2*.

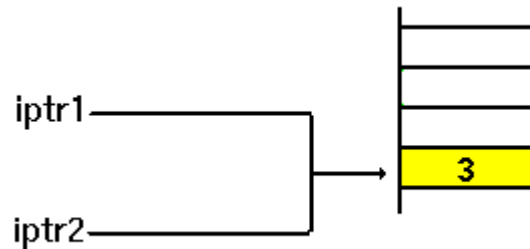


Estruturas de dados com alocação dinâmica de memória

Ao executar a linha

```
iptr1^ := 3;
```

o valor inteiro 3 é armazenado no endereço de memória apontado por *iptr1*, que é o mesmo de *iptr2*.



A saída do programa é

```
the value of iptr1 is 10  
the value of iptr2 is 25  
the value of iptr1 is 3  
the value of iptr2 is 3
```

Estruturas de dados com alocação dinâmica de memória

Resumo de ponteiros:

- Um ponteiro pode apontar para uma localização de qualquer tipo de dado, incluindo *records* (estruturas). Sua sintaxe básica é

```
type PointerType = ^datatype;  
var NameofPointerVariable : PointerType;
```

- O comando *new* aloca espaço de memória para um ponteiro usar.
- O comando *dispose* libera a memória alocada associada a um ponteiro. Deve-se tomar o cuidado de liberar todas as áreas alocadas durante o programa, caso contrário, à medida em que o programa for executado, menos área de memória poderá ser alocada, levando a problemas futuros.
- Pode-se alocar espaço de memória usando *new* ou atribuindo o espaço de memória alocado para outro ponteiro do mesmo tipo (ex: *iptr1 := iptr2*)
- Um ponteiro pode ser definido sem apontar para nenhum espaço de memória, atribuindo *nil* para ele.
- O valor do espaço de memória associado com um ponteiro pode ser lido ou alterado usando a seguinte sintaxe:

NameofPointerVariable[^]

- Um ponteiro pode referenciar um tipo que não foi criado ainda (será visto a seguir).

Estruturas de dados com alocação dinâmica de memória

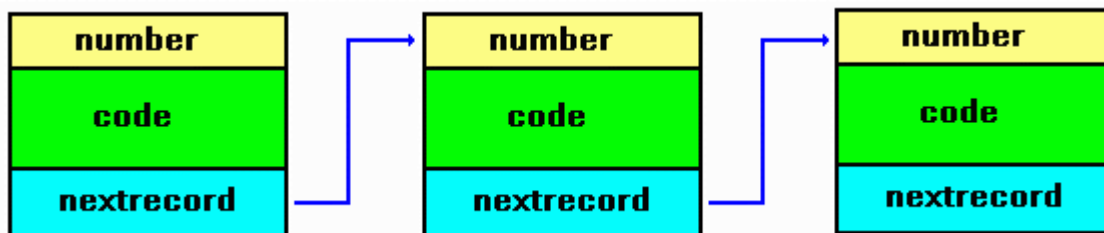
Ponteiros que referenciam dados que não existem ainda:

- Um dos usos mais frequentes de ponteiros é referenciar estruturas (*record*).
- As estruturas podem armazenar ponteiros juntamente com as demais informações.

Exemplo:

```
type rptr = ^recdata;  
recdata = record  
    number : integer;  
    code   : string;  
    nextrecord : rptr;  
end;  
  
var currentrecord : rptr;
```

- *rptr* é um ponteiro para o tipo de dados *recdata*, o qual é definido somente na linha seguinte do programa
- O campo *nextrecord* de *recdata* é uma referência para o ponteiro do tipo *rptr*.
- Usando a definição de *recdata*, é possível criar uma lista de *records* (estruturas), como mostra a figura:



- Tem-se então uma lista formada por uma série de registros iguais, ligados por ponteiros (Lista Encadeada)

Estruturas de dados com alocação dinâmica de memória

Manipulando dados em um ponteiro para estrutura:

Exemplo:

```
program PointerRecordExample( output );
```

```
type rptr = ^reodata;
```

Define o tipo *rptr* como ponteiro para *reodata*

```
reodata = record  
  number : integer;  
  code   : string;  
  nextrecord : rptr  
end;
```

Define a estrutura *reodata*

```
var startrecord : rptr;
```

Cria a variável dinâmica *startrecord* como ponteiro para *reodata*

```
begin
```

```
new( startrecord );
```

Aloca memória para *startrecord*

```
if startrecord = nil then
```

Testa se memória foi alocada

```
begin
```

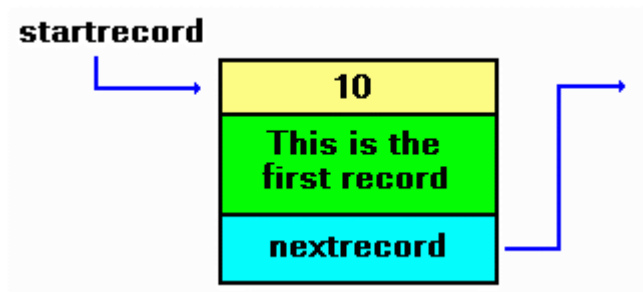
```
  writeln('1: unable to allocate storage space');
```

```
  exit
```

```
end;
```

```
startrecord^.number := 10;
```

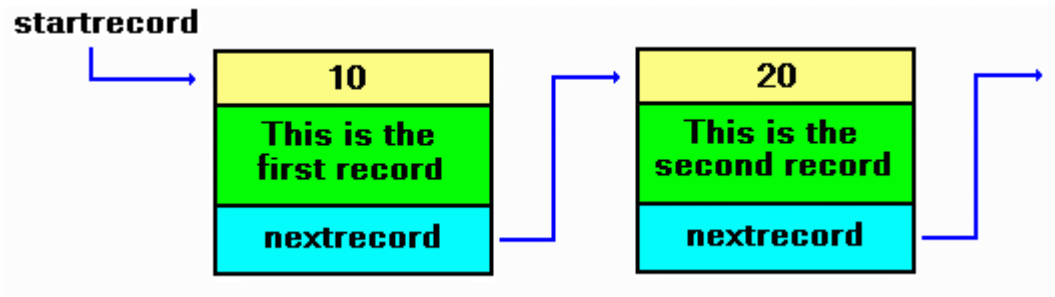
```
startrecord^.code := 'This is the first record';
```



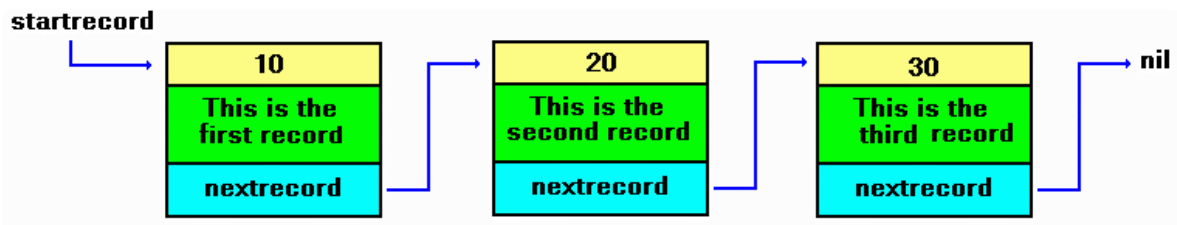
Estruturas de dados com alocação dinâmica de memória

Manipulando dados em um ponteiro para estrutura (continuação):

```
new( startrecord^.nextrecord );  
if startrecord^.nextrecord = nil then  
begin  
    writeln('2: unable to allocate storage space');  
    exit  
end;  
startrecord^.nextrecord^.number := 20;  
startrecord^.nextrecord^.code := 'This is the second record';
```



```
new( startrecord^.nextrecord^.nextrecord );  
if startrecord^.nextrecord^.nextrecord = nil then  
begin  
    writeln('3: unable to allocate storage space');  
    exit  
end;  
startrecord^.nextrecord^.nextrecord^.number := 30;  
startrecord^.nextrecord^.nextrecord^.code := 'This is the third record';  
startrecord^.nextrecord^.nextrecord^.nextrecord := nil;
```



Estruturas de dados com alocação dinâmica de memória

Lista Encadeada:

Para evitar longas notações do tipo:

```
startrecord^.nextrecord^.nextrecord^.number := 30;
```

poderíamos utilizar um ponteiro que referencie o último elemento da lista.

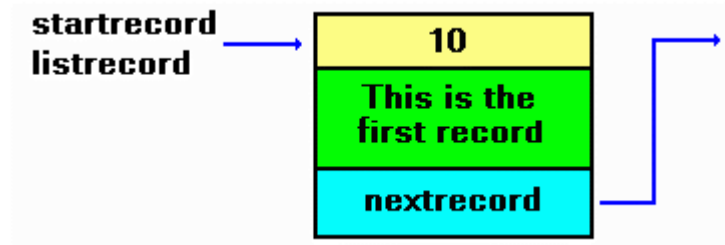
Definição da estrutura de dados:

```
type rptr = ^reodata;  
reodata = record  
    number : integer;  
    code   : string;  
    nextrecord : rptr;  
end;  
var startrecord, listrecord : rptr;
```

Ponteiro para último elemento da lista

Inicializando a estrutura:

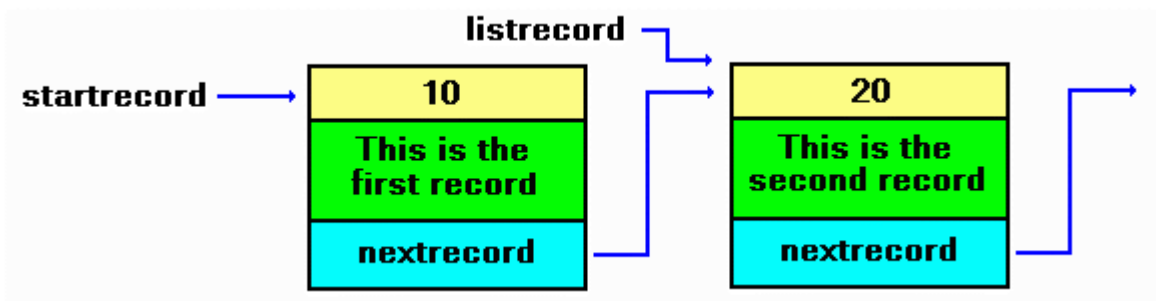
```
new( listrecord );  
if listrecord = nil then  
begin  
    writeln('1: unable to allocate storage space');  
    exit  
end;  
startrecord := listrecord;  
listrecord^.number := 10;  
listrecord^.code := 'This is the first record';
```



Estruturas de dados com alocação dinâmica de memória

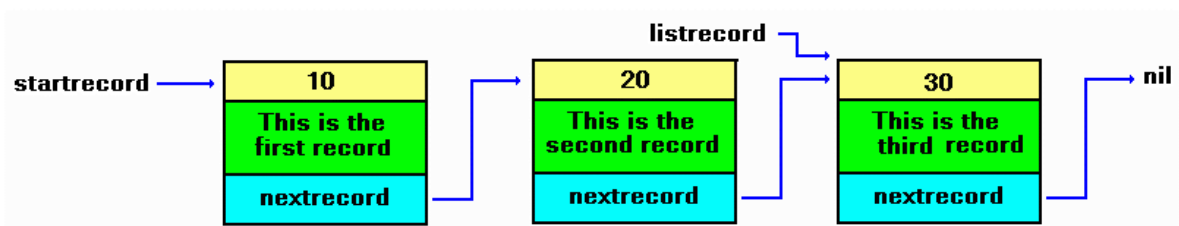
Incluindo novos elementos na lista:

```
new( listrecord^.nextrecord );
if listrecord^.nextrecord = nil then
begin
    writeln('2: unable to allocate storage space');
    exit
end;
listrecord := listrecord^.nextrecord;
listrecord^.number := 20;
listrecord^.code := 'This is the second record';
```



```
new( listrecord^.nextrecord );
if listrecord^.nextrecord = nil then
begin
    writeln('3: unable to allocate storage space');
    exit
end;
listrecord := listrecord^.nextrecord;
listrecord^.number := 30;
listrecord^.code := 'This is the third record';
listrecord^.nextrecord := nil;
```

Indica fim da lista



Estruturas de dados com alocação dinâmica de memória

Listagens completas dos programas exemplos:

```
program PointerRecordExample( output );

type  rptr = ^recdata;
recdata = record
    number : integer;
    code   : string;
    nextrecord : rptr;
end;

var  startrecord : rptr;

begin
new( startrecord );
if startrecord = nil then
begin
    writeln('1: unable to allocate storage space');
    exit
end;
startrecord^.number := 10;
startrecord^.code := 'This is the first record';
new( startrecord^.nextrecord );
if startrecord^.nextrecord = nil then
begin
    writeln('2: unable to allocate storage space');
    exit
end;
startrecord^.nextrecord^.number := 20;
startrecord^.nextrecord^.code := 'This is the second record';
new( startrecord^.nextrecord^.nextrecord );
if startrecord^.nextrecord^.nextrecord = nil then
begin
    writeln('3: unable to allocate storage space');
    exit
end;
startrecord^.nextrecord^.nextrecord^.number := 30;
startrecord^.nextrecord^.nextrecord^.code := 'This is the third record';
startrecord^.nextrecord^.nextrecord^.nextrecord := nil;
writeln( startrecord^.number );
writeln( startrecord^.code );
writeln( startrecord^.nextrecord^.number );
writeln( startrecord^.nextrecord^.code );
writeln( startrecord^.nextrecord^.nextrecord^.number );
writeln( startrecord^.nextrecord^.nextrecord^.code );
dispose( startrecord^.nextrecord^.nextrecord );
dispose( startrecord^.nextrecord );
dispose( startrecord );
end.
```

Estruturas de dados com alocação dinâmica de memória

```
program PointerRecordExample2( output );

    type  rptr = ^recdata;
           recdata = record
               number : integer;
               code   : string;
               nextrecord : rptr
           end;

    var  startrecord, listrecord : rptr;

begin
new( listrecord );
if listrecord = nil then
begin
    writeln('1: unable to allocate storage space');
    exit
end;
startrecord := listrecord;
listrecord^.number := 10;
listrecord^.code := 'This is the first record';
new( listrecord^.nextrecord );
if listrecord^.nextrecord = nil then
begin
    writeln('2: unable to allocate storage space');
    exit
end;
listrecord := listrecord^.nextrecord;
listrecord^.number := 20;
listrecord^.code := 'This is the second record';
new( listrecord^.nextrecord );
if listrecord^.nextrecord = nil then
begin
    writeln('3: unable to allocate storage space');
    exit
end;
listrecord := listrecord^.nextrecord;
listrecord^.number := 30;
listrecord^.code := 'This is the third record';
listrecord^.nextrecord := nil;
while startrecord <> nil do
begin
    listrecord := startrecord;
    writeln( startrecord^.number );
    writeln( startrecord^.code );
    startrecord := startrecord^.nextrecord;
    dispose( listrecord )
end
end.
```

Estruturas de dados com alocação dinâmica de memória

```
program PointerRecordExample3( input, output );

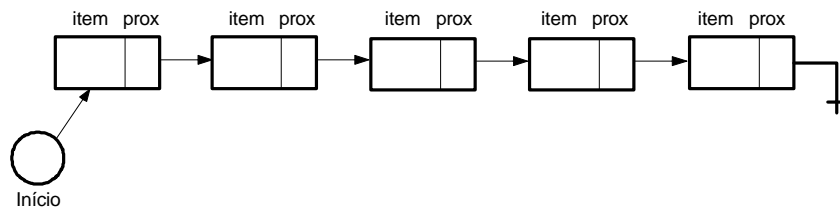
type  rptr = ^recdata;
recdata = record
    number : integer;
    code : string;
    nextrecord : rptr
end;

var    startrecord, listrecord, insertptr : rptr;
        digitcode : integer;
        textstring : string;
        exitflag, first : boolean;

begin
    exitflag := false;
    first := true;
    while exitflag = false do
        begin
            writeln('Enter in a digit [-1 to end]');
            readln( digitcode );
            if digitcode = -1 then
                exitflag := true
            else
                begin
                    writeln('Enter in a small text string');
                    readln( textstring );
                    new( insertptr );
                    if insertptr = nil then
                        begin
                            writeln('1: unable to allocate storage space');
                            exit
                        end;
                    if first = true then begin
                        startrecord := insertptr;
                        listrecord := insertptr;
                        first := false
                    end
                    else begin
                        listrecord^.nextrecord := insertptr;
                        listrecord := insertptr
                    end;
                    insertptr^.number := digitcode;
                    insertptr^.code := textstring;
                    insertptr^.nextrecord := nil
                end
            end;
        end;
    while startrecord <> nil do
        begin
            listrecord := startrecord;
            writeln( startrecord^.number );
            writeln( startrecord^.code );
            startrecord := startrecord^.nextrecord;
            dispose( listrecord )
        end
    end.
end.
```

Estruturas de dados com alocação dinâmica de memória

Listas Encadeadas Símples:



- Em uma implementação através de ponteiros, cada item da lista é encadeado com o seguinte através de uma variável do tipo apontador
- Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e retirar elementos sem haver necessidade de deslocar os itens seguintes da lista
- Uma lista é constituída de células. Cada célula contém um ítem da lista e um apontador para a célula vizinha
- Utiliza-se um ponteiro para referenciar o início da lista.
- Pode-se utilizar dois ponteiros adicionais: um ponteiro que referencie o final da lista e um ponteiro para referenciar alguma célula da lista.

Estruturas de dados com alocação dinâmica de memória

Projeto: “Exame vestibular de uma faculdade” (Furtado, A. L., 1984)

- Uma faculdade necessita de um sistema que permita cadastrar os resultados de um vestibular e faça a distribuição dos aprovados nos cursos, de acordo com suas opções.
- Cada candidato tem direito a 3 opções para tentar uma vaga em um dos 7 cursos.
- Registro para cada candidato:
 - Chave: 1..999
 - NotaFinal: 0..10
 - Opção: array[1..3] of 1..7

Chave: número de inscrição do candidato

Nota Final: média das notas do candidato

Opção: Vetor contendo a primeira, Segunda e terceira opções do candidato

PROBLEMA:

- Distribuir os candidatos entre os cursos, de acordo com a nota final e as opções apresentadas por cada candidato.
- No caso de empate: atende primeiro os candidatos com nota igual de acordo com a ordem da inscrição para os exames.

Estruturas de dados com alocação dinâmica de memória

Possível Solução:

1. Ordenar os registros pelo campo NotaFinal, respeitando-se a ordem de inscrição dos candidatos.
2. Percorrer cada conjunto de registro com mesma NotaFinal, iniciando-se pelo conjunto com NotaFinal 10, seguido do conjunto com NotaFinal 9 e assim por diante.

Para um mesmo conjunto de NotaFinal, tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções que houver vaga (se houver)

Objetivos:

1. Desenvolver o projeto do sistema, de acordo com especificação exemplo.
2. Desenvolver o programa em pascal
3. Utilizar lista encadeada (ponteiros)

Estruturas de dados com alocação dinâmica de memória

Modelo de Documento - AnteProjeto

P R O J E T O

< Controle para classificação do vestibular >

< C.C.V >

<Unitau> - <Departamento de Informática>

<Taubaté>

<09/2000>

<2º Tecnólogo em Informática>

Estruturas de dados com alocação dinâmica de memória

SUMÁRIO

	Página
1 - INTRODUÇÃO.....	23
2 - DEFINIÇÃO DOS OBJETIVOS.....	23
3 - DEFINIÇÃO DA ABRANGÊNCIA.....	23
4 - DESCRIÇÃO LÓGICA DA SOLUÇÃO DO PROBLEMA.....	24
5 - ANÁLISE DE DADOS.....	24
6 - ANÁLISE FUNCIONAL.....	25
7 - CÓDIGO COMPUTACIONAL DOCUMENTADO.....	26
8 - RESULTADOS EM TESTES DE VALIDAÇÃO.....	26
9 - CONCLUSÕES.....	27

Estruturas de dados com alocação dinâmica de memória

1 - Introdução

Deverá conter o objetivo e a estrutura do documento.

2 - Definição dos Objetivos

Deverá conter o objetivo global e os específicos do software, conforme a atividade “Identificação do Objetivos”.

3 - Análise de Dados

Deverá conter:

- Descrição e objetivos de toda a estrutura de dados utilizada;

4 - Análise Funcional

Deverá Conter:

- Descrição das etapas ou procedimentos necessários para resolver o problema e como estes procedimentos atuam sobre a estrutura de dados

5 – Código Computacional Documentado

Deverá Conter:

- Descrição da estrutura de dados, destacando a construção e utilização dos dados, conforme item 3
- Descrição do funcionamento do programa, conforme item 4.

4 – Resultado em teste de validação

Deverá Conter:

- Amostra de funcionamento do programa – impressão de telas de inicialização, entrada de dados, relatórios

4 - Conclusão

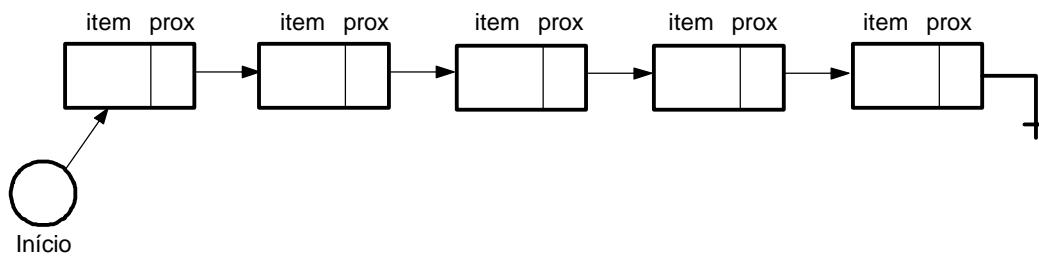
Deverá Conter:

- Analisar a importância da utilização do seu sistema e as vantagens em utilizá-lo

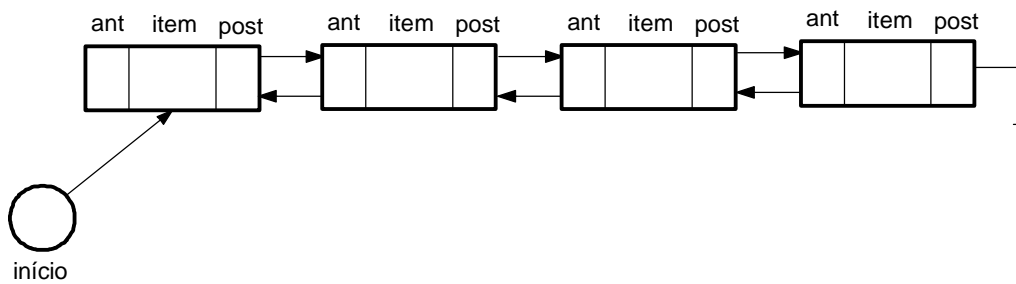
Estruturas de dados com alocação dinâmica de memória

LISTAS LINEARES LIGADAS (OU ENCADEADAS)

Representação Genérica:



Lista Linear Simplesmente Ligada



Lista Linear Duplamente Ligada

Estruturas de dados com alocação dinâmica de memória

Características das listas encadeadas:

- são implementadas através de variáveis dinâmicas
- os nós que compõem a lista devem ser agregados do tipo registro contendo, pelo menos, dois campos: um campo de tipo simples ou construído para abrigar as informações armazenadas na lista e um campo do tipo ponteiro para abrigar o endereço do nó subsequente da lista.
- o acesso aos nós componentes da lista é seqüencial (para se acessar o 4º elemento, é necessário passar antes pelo 3º, para se acessar o 3º elemento, é necessário passar antes pelo 2º e assim sucessivamente)
- as estruturas de representação de Listas Ligadas devem obrigatoriamente suportar conceitos como ponteiros e alocação dinâmica
- deve existir pelo menos um ponteiro externo a lista, que aponte para ela, permitindo assim acessá-la. Nas representações genéricas acima este ponteiro é denominado início.
- as Listas Ligadas podem ser simplesmente encadeadas (um único ponteiro por nó, referenciando nó posterior) ou duplamente encadeadas (dois ponteiros por nó, referenciando nós anterior e posterior).

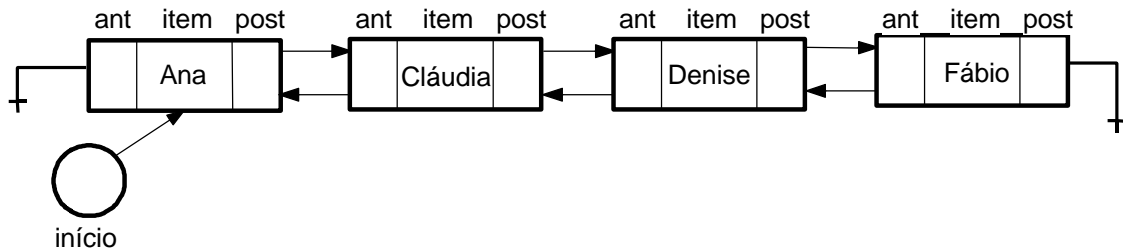
Estruturas de dados com alocação dinâmica de memória

Vantagens e Desvantagens das listas encadeadas: são inerentes à utilização de variáveis dinâmicas, como por exemplo:

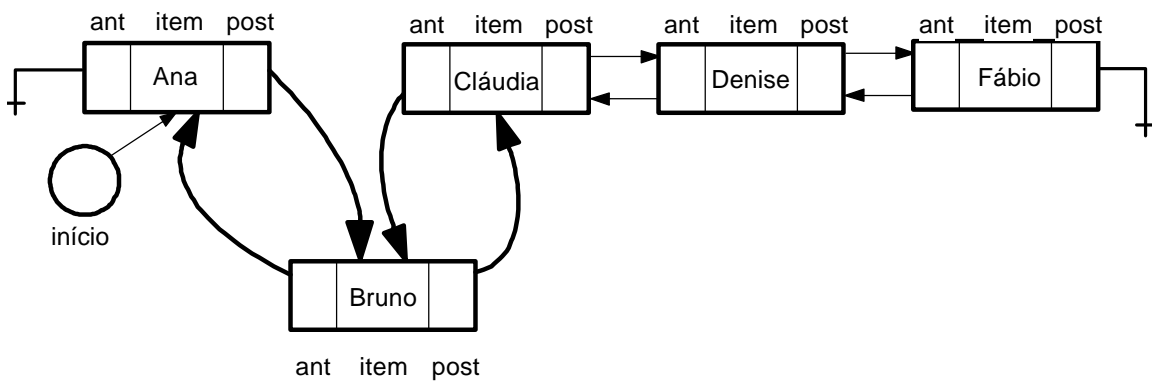
- maior complexidade inerente à manipulação de ponteiros (desvantagem)
- o fato de nem todas as linguagens de programação permitirem a construção de estruturas para a representação de Listas Ligadas (desvantagem)
- a possibilidade de se trabalhar com listas de tamanhos indefinidos, que podem crescer e decrescer conforme a necessidade (vantagem)
- maior facilidade para a realização de operações de inserção e remoção, que consistem basicamente no rearranjo de alguns ponteiros (vantagem).

Estruturas de dados com alocação dinâmica de memória

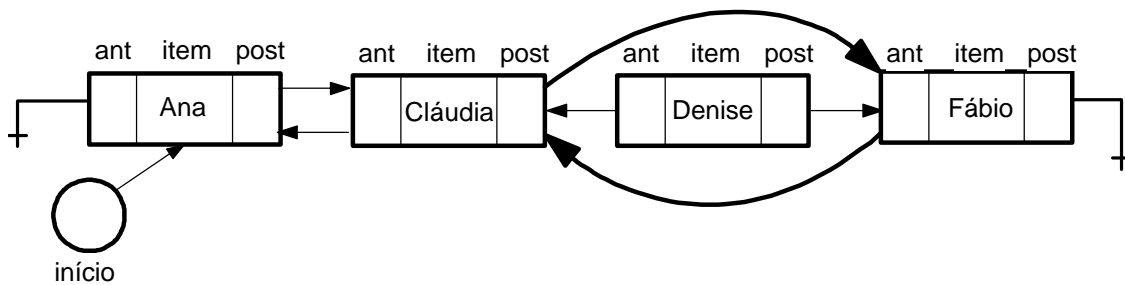
Exemplo: suponha a lista abaixo:



Inserção de um item: Uma área de memória é alocada e ponteiros são redirecionados.



Remoção de um item: Redireciona-se os ponteiros e desaloca-se a memória do item removido.

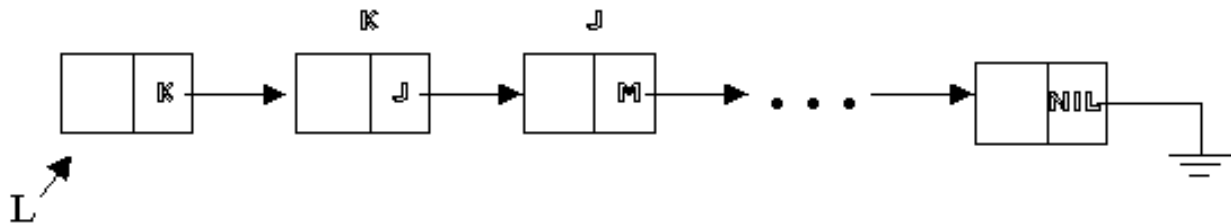


Estruturas de dados com alocação dinâmica de memória

Operações em Listas Encadeadas:

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



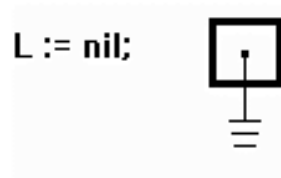
Definição da ED

```
Type  prec = ^rec;
      Lista = prec;
      rec = record
          info: T;      {seu tipo preferido}
          lig:  Lista
      End;
```

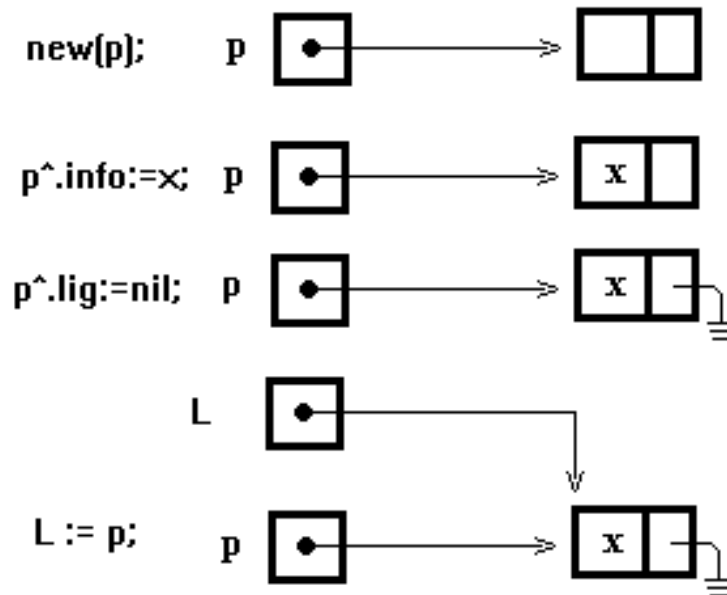
```
Var p: Lista; {ponteiro para qualquer elemento
              da lista}
    L: Lista; {ponteiro para o primeiro elemento
              da lista }
```

Estruturas de dados com alocação dinâmica de memória

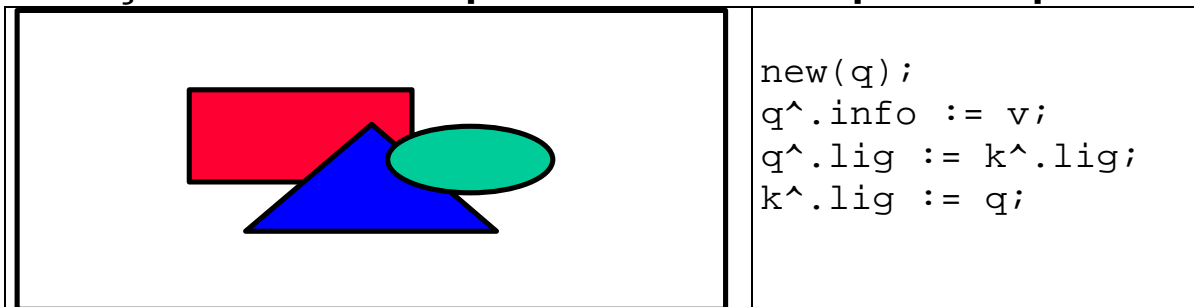
Criação de lista vazia:



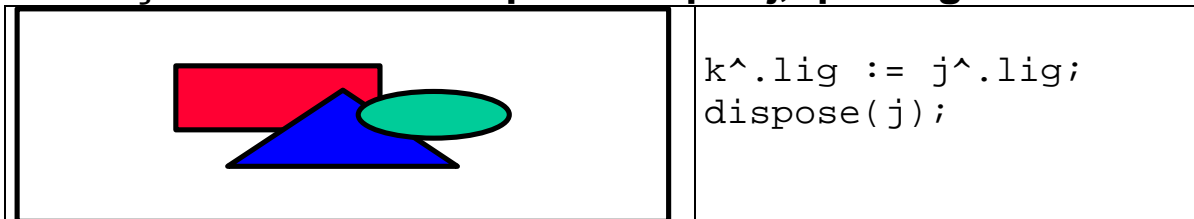
Inserção do primeiro elemento:



Inserção do valor v depois do elemento apontado por k



Remoção do elemento apontado por j, que segue k



Estruturas de dados com alocação dinâmica de memória

Listas Duplamente Encadeadas:

Características

- Listas encadeadas simples são percorridas do início ao final.
- Ponteiro "anterior" necessário para muitas operações.
- Em alguns casos pode-se desejar percorrer uma lista nas duas direções indiferentemente.
- Nestes casos, o gasto de memória imposto por um novo campo de ponteiro pode ser justificado pela economia em não reprocessar a lista toda.

Definição da ED

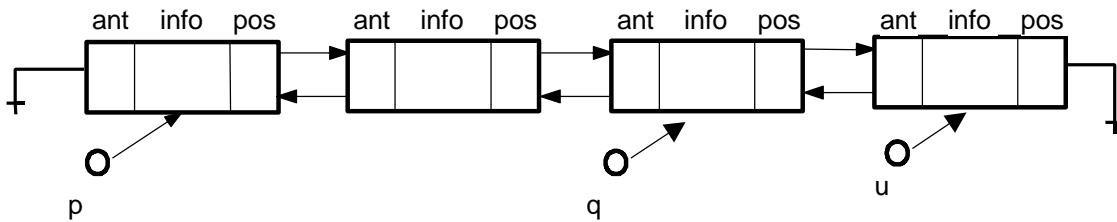
```
Type Lista = ^rec;
```

```
    rec = record
        info: T;      {seu tipo preferido}
        ant:  Lista;
        pos:  Lista;
    End;
```

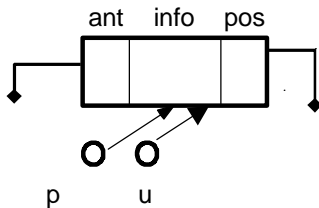
```
Var q,r: Lista; {ponteiros para qualquer elemento
                da lista}
    p  : Lista; {ponteiro para o primeiro elemento
                da lista }
    u  : Lista; {ponteiro para o último elemento
                da lista }
```

Estruturas de dados com alocação dinâmica de memória

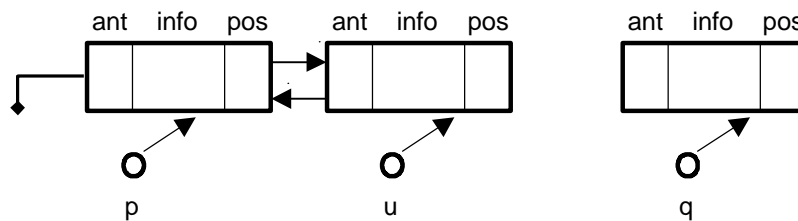
Representação gráfica de uma lista duplamente encadeada:



Inicializando a lista:



```
new(p);    u = p;  
p^.info := " ";  
p^.pos := nil;  
p^.ant := nil;
```



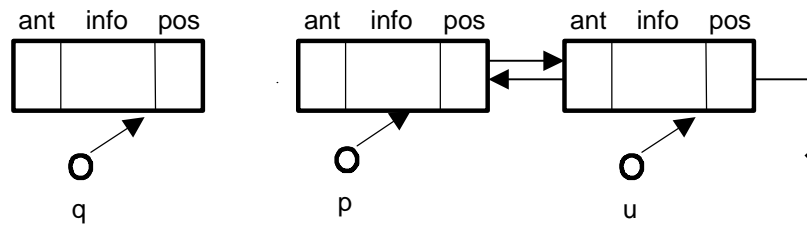
Inserindo um novo elemento no fim da lista:

```
new(q);  
q^.info := " ";  
q^.pos := nil;  
q^.ant := u;  
u^.pos := q;  
u := q;
```

Retirando um elemento do fim da lista:

```
q := u^.ant;  
q^.pos := nil;  
dispose(u);  
u := q;
```

Estruturas de dados com alocação dinâmica de memória



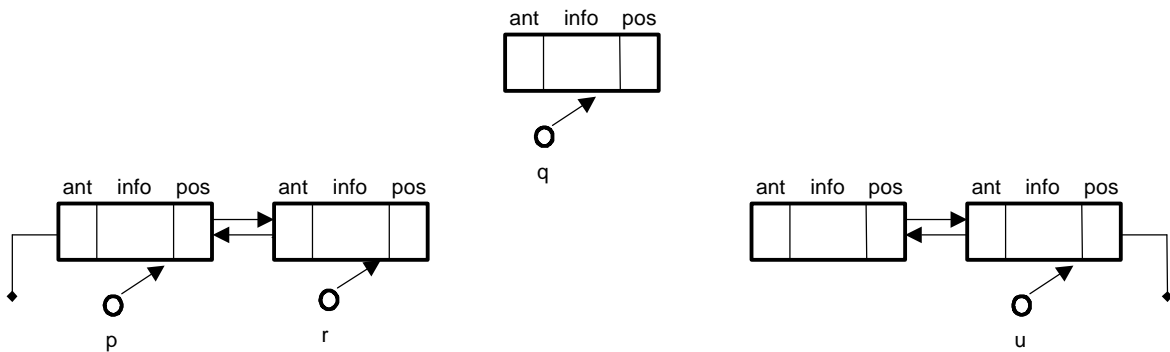
Inserindo um novo elemento no início da lista:

```
new(q);  
q^.info := "  ";  
q^.pos := p;  
q^.ant := nil;  
p^.ant := q;  
u := q;
```

Retirando um elemento do início da lista:

```
q := p^.pos;  
dispose(p);  
p := q;
```

Estruturas de dados com alocação dinâmica de memória



Inserindo um novo elemento no meio da lista:

```
new(q);  
q^.info := "  ";  
q^.ant := r;  
q^.pos := r^.pos;  
r^.pos^.ant := q;  
r^.pos = q;  
u := q;
```

Removendo um elemento do meio da lista (q):

```
r := q^.ant;  
q^.pos^.ant := r;  
r^.pos = q^.pos;  
dispose(q);
```

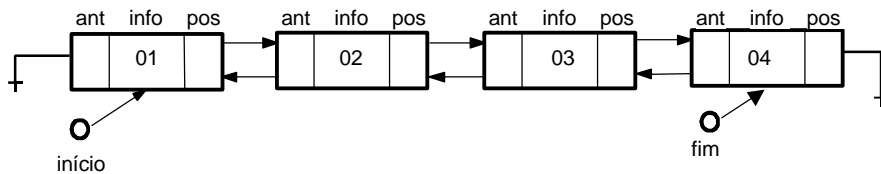
Estruturas de dados com alocação dinâmica de memória

Filas e pilhas em listas encadeadas duplas:

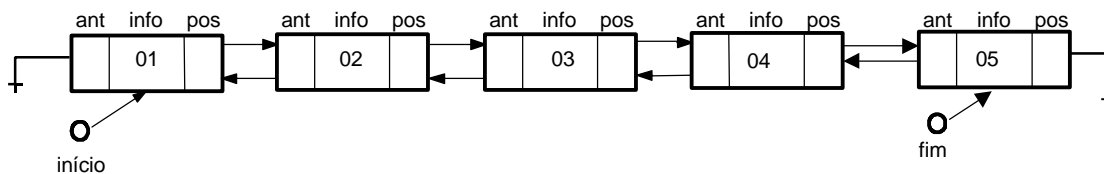
Relembrando:

- Filas são listas em que a inclusão de elementos é feita no fim da lista e a retirada do início da lista -> Primeiro que entra é o primeiro que sai;
- Pilhas são listas em que a inclusão e remoção de elementos é feita no fim da lista -> primeiro que entra é o último que sai;

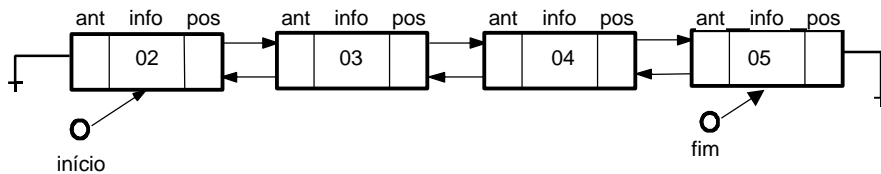
Operações em fila:



Inserção: novo elemento no fim da fila

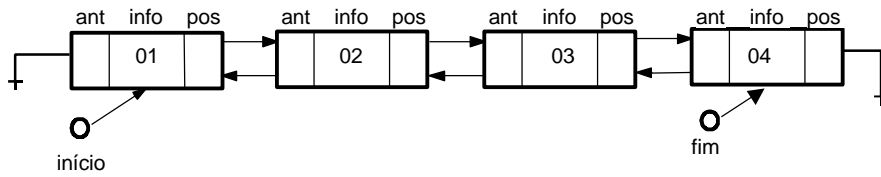


Remoção: elemento retirado do início da fila

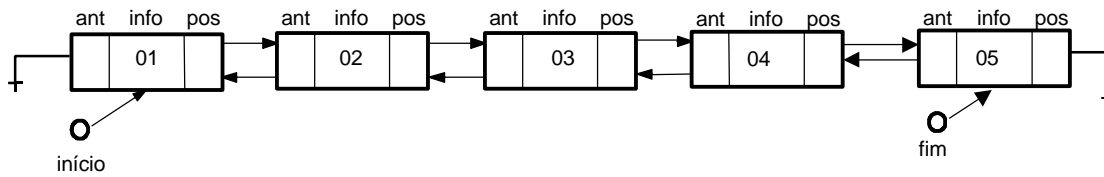


Estruturas de dados com alocação dinâmica de memória

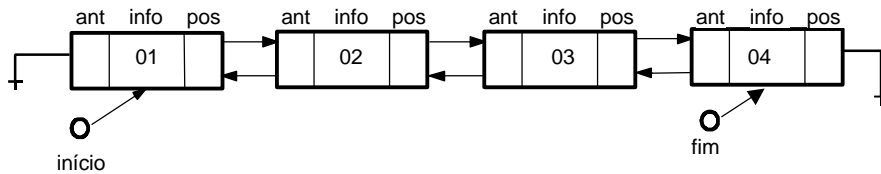
Operações em pilha:



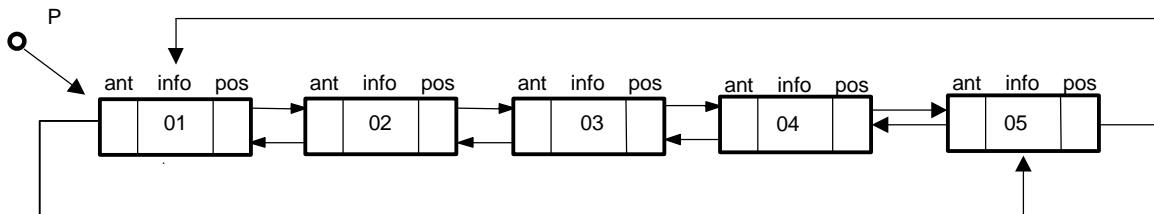
Inserção: novo elemento no fim da fila



Remoção: elemento retirado do fim da fila

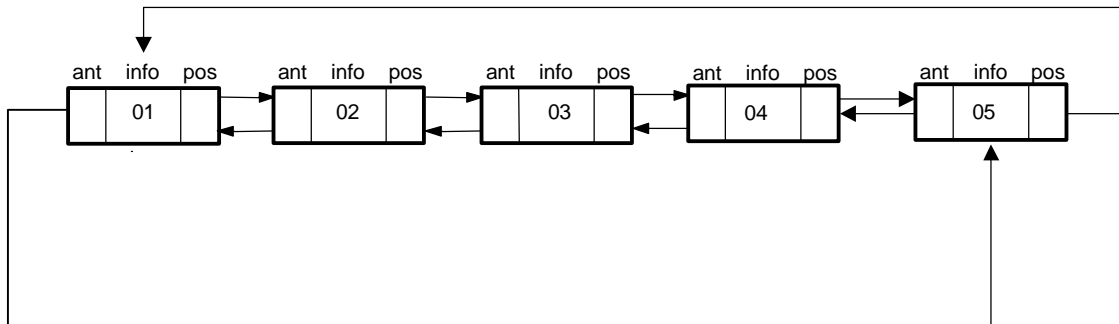


Lista circular:



Estruturas de dados com alocação dinâmica de memória

Fila com Lista circular:



Definição da ED:

```
Type Lista = ^rec;
```

```
rec = record
    info: T;      {seu tipo preferido}
    ant: Lista;
    pos: Lista;
End;
```

```
Var p : Lista; {ponteiro para o primeiro elemento
                da lista }
    u : Lista; {ponteiro para o último elemento
                da lista }
```

Operações básicas:

- Inserção no fim da fila;
- Remoção no início da fila;
- Verificação de fila cheia (*overflow*);
- Verificação de fila vazia (*underflow*);