Estruturas de dados: listas lineares
Lista sequencial

Lista linear

- **Definição:** Uma **lista linear** é uma coleção ordenada de componentes de um mesmo tipo. Ela pode ser:
  1. ou vazia;
  2. ou ser escrita na sua forma padrão : $(a_1, a_2, \ldots, a_n)$, em que:
     - a_i : são os componentes de um mesmo conjunto S.
     - a_1 : é o primeiro elemento da lista, e a_n o último.
     - a_i precede a_{i+1}.

- **Exemplo:** listas de chamada, de compras, de pessoas, etc.

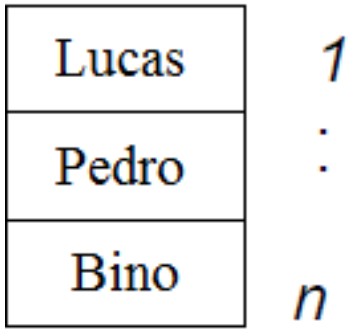
Lista linear

- **É uma estrutura homogênea:** elementos de um mesmo tipo base.
- Uma lista pode ser: **ordenada** (via um campo "chave"); ou **não-ordenada**.
- **Operações básicas:**
  1. Verificar se uma lista vazia.
  2. Inserir um elemento na lista.
  3. Remover um elemento da lista.
  4. Busca (acesso) por algum elemento, dada uma chave, ou uma posição de busca na lista.
- Outras operações:
  - ordenar, concatenar, inverter, etc ...

Lista linear – tipos de implementação

1. **Sequencial:** sucessor lógico de um elemento ocupa posição física consecutiva na memória (endereços consecutivos).

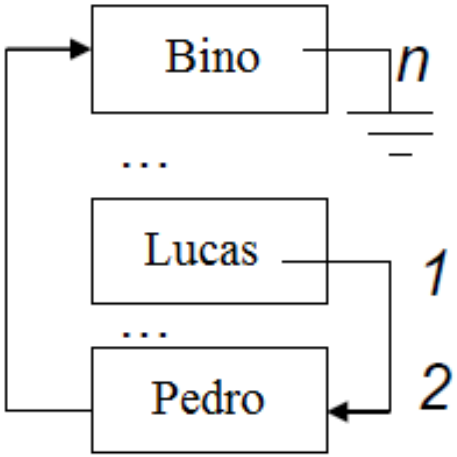
   L = (Lucas, Pedro, Bino)



Array
(estática)

2. **Encadeada:** elementos logicamente consecutivos **não implicam** em elementos (endereços) consecutivos na memória

   L = (Lucas, Pedro, Bino)



Ponteiros
(dinâmica)


Perguntas?

TAD: lista sequencial (não-ordenada e ordenada)

- **Atributos:**
  - ***tipo_elem***: pode ser uma ED composta por **vários campos**, sendo um deles o chamado "campo chave" (***tipo_chave***).
  - Um campo é dito **chave** se ele guarda **valores distintos para elementos distintos na lista**.
    - O campo chave é importante, pois garante **unicidade aos elementos**, isto é, identifica cada elemento na lista de forma única.
    - **Exemplos:** RG, CPF, Id, Cidade + Estado, etc.

TAD: lista

- **Operações sobre a lista:**
  - **Inicialização:** cria lista vazia.
  - **Inserir:** insere um elemento na lista.
  - **Busca #1:** utilizada para buscar a posição (na lista) de um elemento dada sua chave de identificação.
  - **Busca #2:** utilizada para buscar um elemento na lista a partir de uma posição *p* dada.
  - **Elimina:** remove elemento da lista.
  - **Conta:** efetua contagem do número de elementos da lista.
  - **Destrói:** destrói a lista (logicamente).
  - **Verifica (vazia):** verifica se a lista está vazia.
  - **Verifica (cheia):** verifica se a lista está cheia.

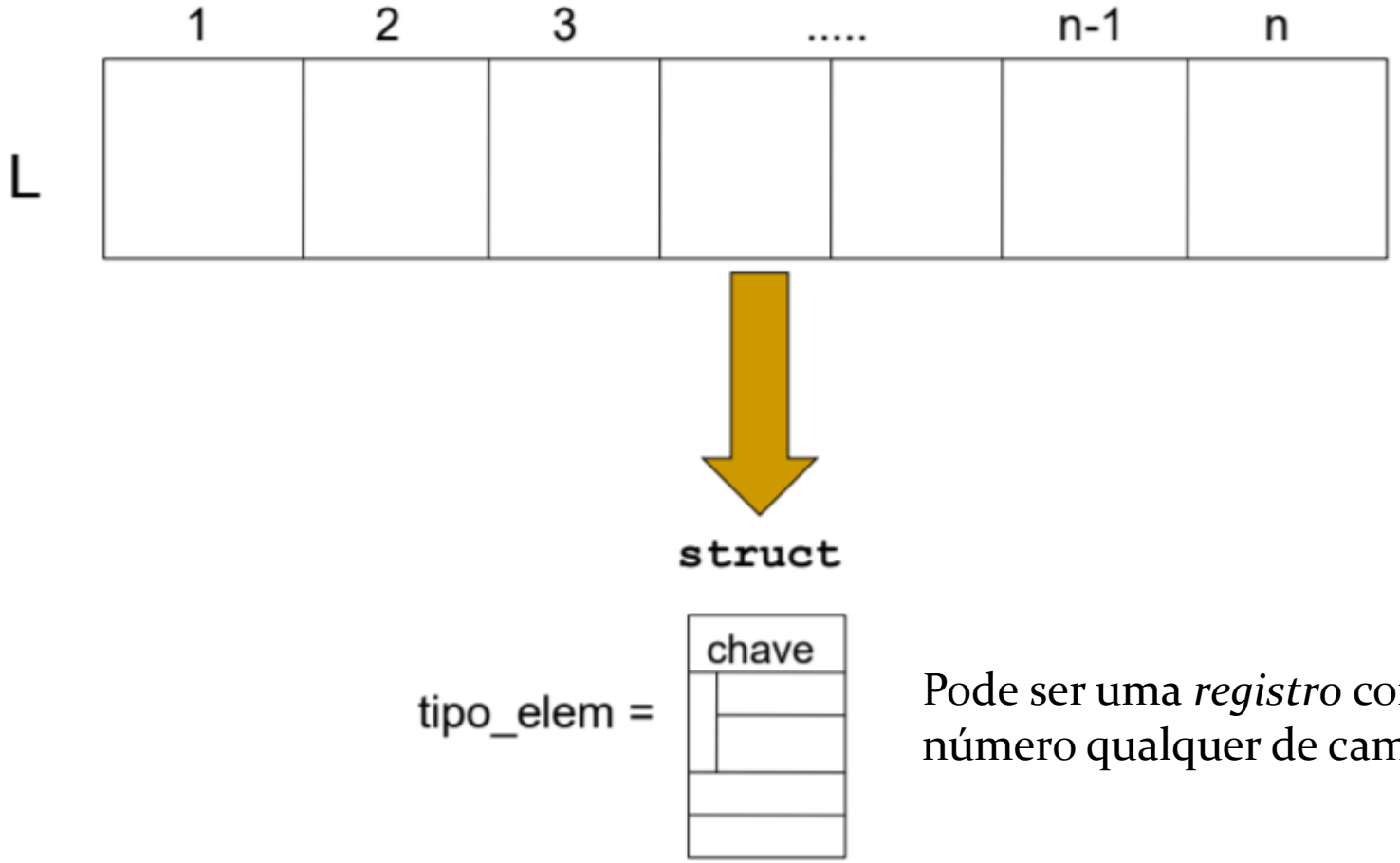
TAD: implementação **lista sequencial**

- **Implementação**

  - As escolhas dos tipos de dados a seguir valem tanto para uma lista ordenada como não-ordenada.
    - Diferença no caso da lista não-ordenada: funções de **inserção**, **busca** e **remoção**.

  - **Observação**: na implementação **sequencial**, utiliza-se **ARRAY** para armazenar e manipular os elementos da lista.

Representação de uma **lista sequencial**

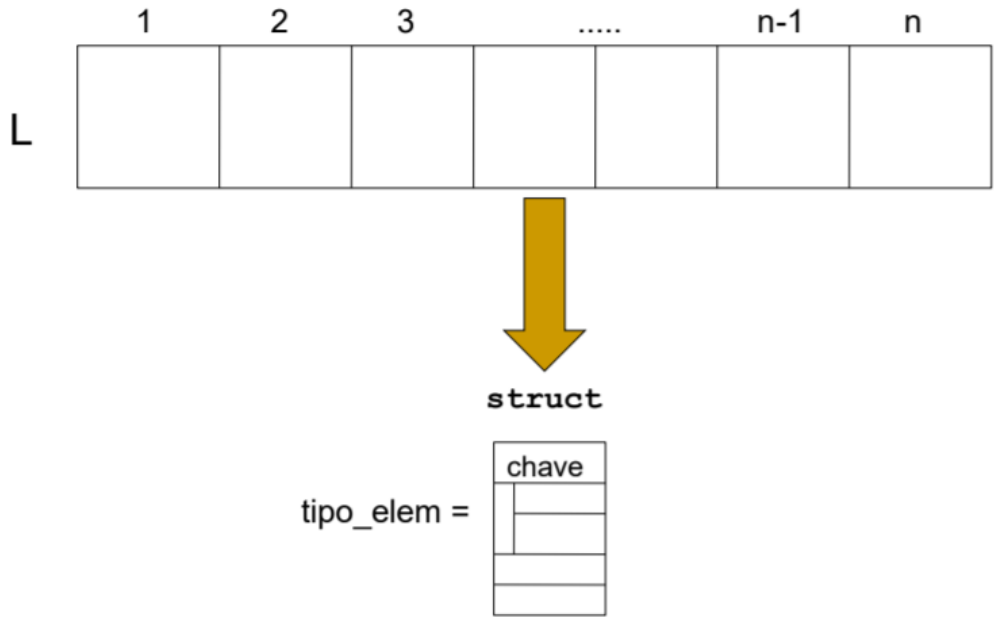


Pode ser uma *registro* com um número qualquer de campos

Exemplo do código de implementação de uma lista sequencial: **lista.c**

```c
#include <stdlib.h> //Para usar malloc, free, exit ...
#include <stdio.h>  //Para usar printf ,...
#include "lista.h" //Carrega o arquivo .h criado

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

//Implementação: lista seq. encadeada

//Estruturas e tipos empregados
//--------------------------------------
```

```c
//Tipo chave
typedef int tipo_chave;

//Tipo registro
typedef struct
{
   char nome[30];
   //... (caso tenha mais campos)
} tipo_dado;

//Tipo elemento (registro + chave)
typedef struct
{
   tipo_chave chave;
   tipo_dado info;
} tipo_elem;

//Tipo lista (seq. encadeada)
typedef struct
{
   int nelem; //nro de elementos
   tipo_elem A[MAX+1];
} lista;
```



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

boolean Cheia(lista *L)
{
   //Retorna true (1): se 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;
   L->A[0].chave = 0;
}
```

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

boolean Inserir_posic(tipo_elem x, int p, lista *L)
{
   /*Insere x, que é um novo elemento na posição p da lista
   Se L = a_1, a_2,... a_n então temos a_1, a_2, ...
   a_{p-1}, x, a_{p+1}, ... an.
   Devolve true se sucesso, false c.c. (isto é: L não tem nenhuma
   posição p ou lista cheia). Obs: 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--) //Copia vizinho p/ direita
          L->A[q+1] = L->A[q];

       L->A[p] = x;
       L->nelem++;

       return TRUE; //Inserção feita com sucesso
    }
}
```

```
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*/

   if (!Vazia(L))
   {
      int i = 1;
      while (i <= L->nelem)
         if (L->A[i].chave == x)
         {
            *p = i;
            return TRUE;
         }
         else
            i++;
   }
   return FALSE; //Retorna false se não encontrou
}
```

```
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; i++)
      L->A[i-1] = L->A[i];

   L->nelem--;
}
```

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

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

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

Estamos quase la...

```
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 = 1;
   int sup = L->nelem;
   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;
}
```

PARA LISTAS ORDENADAS APENAS!!!

```
boolean Remover_ch(tipo_chave x, lista *L)
{
   /*Remoção dada a chave. Retorna true, se removeu, ou
   false, c.c.*/

   int *p;
   boolean removeu = FALSE;

   if (Busca_bin(x, L, p)) //Procura via busca binária
   {
       Remover_posic(p, L);
       removeu = TRUE;
   }

   return removeu;
}
```

PARA LISTAS ORDENADAS APENAS!!!

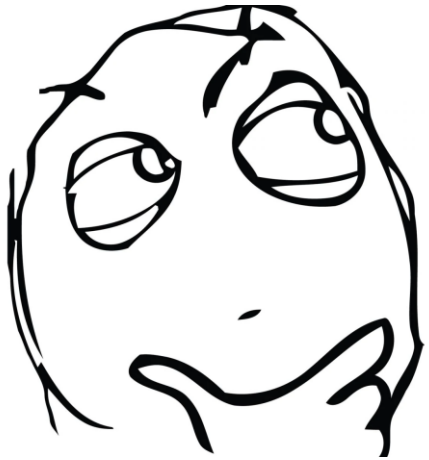

Perguntas?

Lista linear sequencial: sumário

- **Vantagens**
  - Acesso direto a cada elemento:
    - lista.A[i] e (lista->A[i])
  - Tempo Constante (devido ao uso do array).
- **Desvantagens**
  - Necessidade de deslocar os dados na **Inserção** e **Eliminação**, se a lista é **ordenada**.
  - Tamanho máximo da lista é delimitado no início (em decorrência do uso do array).
    - Risco de overflow.

Lista linear sequencial: sumário

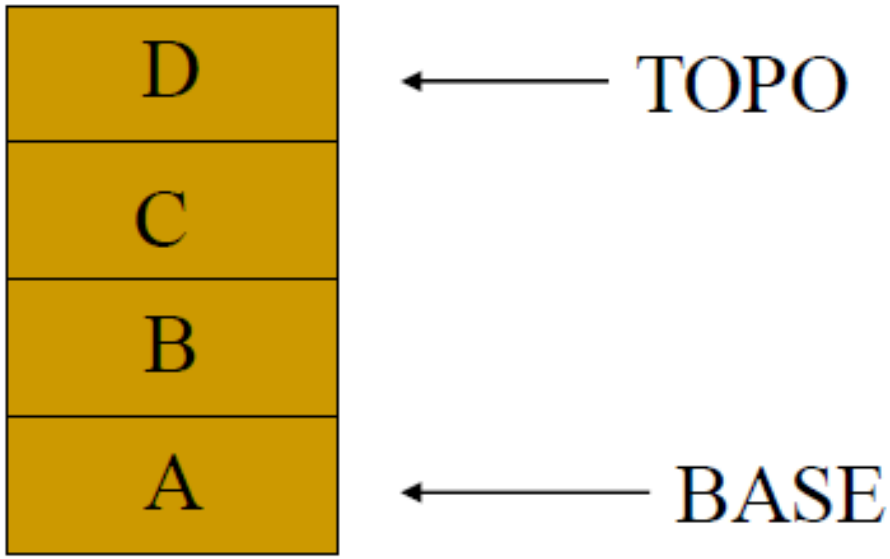
- **Quando devemos optar por ela?**
  - Listas pequenas ➔ custo insignificante.
  - Conhecimento prévio do comportamento da lista:
    - Poucas inserções/remoções a serem feita!



Pilha

Pilhas

- **Definição:** Uma **pilha** (*stack*) é uma lista linear cujas operações de **inserção** e **eliminação** só ocorrem no **TOPO** da pilha (extremidade onde o último elemento foi inserido).



Pilhas

- Dada uma pilha P = $(a_1, a_2, \ldots, a_n)$ escrita em sua forma padrão. Dizemos que:
  - a_1 : é o elemento da **BASE** da pilha.
  - a_n : é o elemento do **TOPO** da pilha.
  - a_{i+1} está **ACIMA** de a_i na pilha.

- Os elementos são colocados uns sobre os outros. O elemento inserido mais recentemente está no **TOPO**, e o inserido menos recentemente na **BASE**.
- **Propriedade:** o **último elemento inserido é o primeiro que poderá ser retirado**. São chamadas listas LIFO ("Last-in, First-out" – "último que entrou é o primeiro a sair").

Pilhas – exemplo

- **Exemplo:** Pilhas de bandejas em um "Bandejão" ou RU.
  1. Bandejas são empilhadas uma a uma.
  2. Pega-se (remove-se) sempre a bandeja do topo.
  3. Se não há mais bandejas, a pilha está vazia e não podemos remover mais.
  4. Uma vez que podemos ver a bandeja no topo, se ela estiver suja, podemos então não querer pegar (remover) ela.
- As situações acima fazem referência à 4 operações de pilhas:
  1. Inserir = *push*.
  2. Remover = *pop*.
  3. Verificar se está vazia.
  4. Examinar o elemento do topo (mas sem removê-lo).

TAD: pilha sequencial

```c
#define MAX 1000    //estimativa para tamanho máximo
#define TRUE 1      //define tipo boleano
#define FALSE 0
#define boolean int
#define indice int

//Estruturas e tipos empregados
//-----------------------------------------
//Tipo chave
typedef int tipo_chave;

//Tipo registro
typedef struct
{
   char nome[30];
   //... (caso tenha mais campos)
} tipo_dado;
```

pilha.h

```c
//Tipo elemento (registro + chave)
typedef struct
{
   //tipo_chave chave;
   tipo_dado info;
} tipo_elem;

//Tipo lista (seq. encadeada)
typedef struct
{
   tipo_elem A[MAX+1];
   indice topo;
} pilha;
//----------------------------------------
```

pilha.h

```c
//Declarações de funções/operações
//-----------------------------------------------------
//Cria pilha vazia (deve ser usada antes de qualquer outra operação)
void Define(pilha *P);

//Insere item no topo da pilha. Retorna true se sucesso, false c.c.
boolean Push(tipo_dado elem, pilha *P);

//Retorna true se pilha é vazia, false c.c.
boolean Vazia(pilha *P);

//Reinicializa pilha
void Desvaziar(pilha *P);

//Devolve o elemento do topo sem remove-lo. Chamada apenas se pilha é não vazia
tipo_elem top(pilha *P);

//Remove item do topo da pilha. Chamada apenas se pilha é não vazia
void Pop_up(pilha *P);

//Remove e retorna o item do topo da pilha. Chamada apenas se pilha nao vazia
tipo_elem pop(pilha *P);
//-----------------------------------------------------
```

pilha.h

TAD: implementação das operações

```
//Operações
//-----------------------------------------
//Define (P): cria uma pilha P vazia
void Define(pilha *P)
{
   P->topo = 0;
}

//Insere x no topo de P (empilha): Push (x, P)
boolean Push(tipo_dado x, pilha *P)
{
   if(P->topo == MAX)
       return FALSE;    //Pilha cheia

   P->topo ++;
   P->A[P->topo].info = x;

   return TRUE;
}
```

pilha.c

```c
//Testa se P está vazia
boolean Vazia (pilha *P)
{
   return(P->topo == 0);
}

//Acessa o elemento do topo da pilha (sem remover)
//Obs: testar antes se a pilha não está vazia
tipo_elem Top (pilha *P)
{
   return P->A[P->topo];
}

//Remove o elemento no topo de P sem retornar valor (desempilha, v. 1)
//Obs: testar antes se pilha não está vazia.
void Pop_up (pilha *P)
{
   P->topo --;
}
```

pilha.c

```c
//Remove e retorna o elemento (todo o registro) eliminado (desempilha, v. 2)
//Obs: testar antes se pilha não está vazia.
tipo_elem Pop (pilha *P)
{
   tipo_elem x = P->A[P->topo];
   P->topo --;
   return x;
}
//-----------------------------------------
```

pilha.c

Pilhas – vantagens e desvantagens

- **Em listas em geral**, há a necessidade de se movimentar os elementos nas operações de inserções e remoções.

- **No caso das pilhas:** essas movimentações não ocorrem!

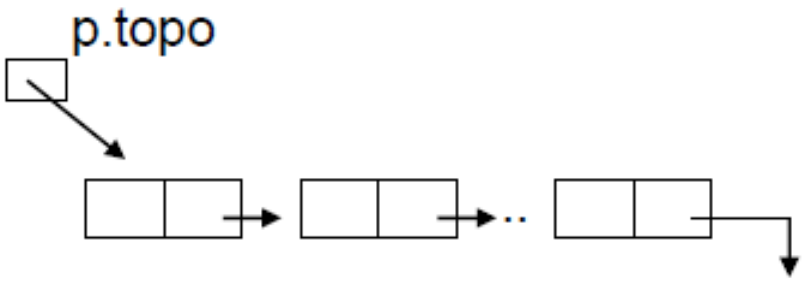
- A alocação sequencial é vantajosa a menos de quando não sabemos ao certo qual será o tamanho máximo da pilha.


Perguntas?

Pilha dinâmica

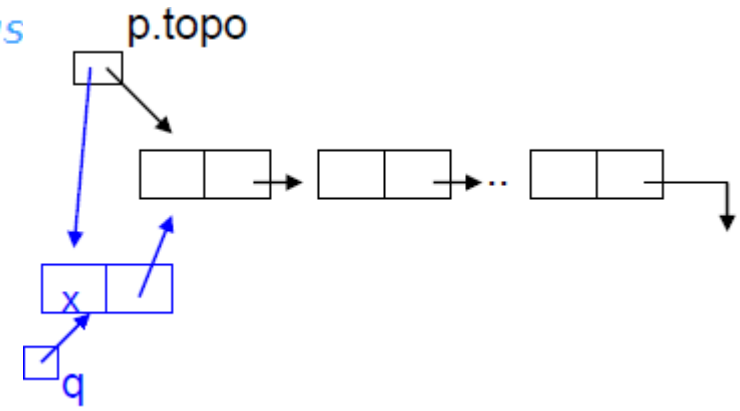
TAD: pilha dinâmica - implementação

```c
//Tipo registro
typedef struct
{
   char nome[30];
   //... (caso tenha mais campos)
} tipo_dado;

//Tipo elemento (unidade dinamica)
typedef struct elem
{
  tipo_dado info;
  struct elem *lig;
} tipo_elem;

//Tipo pilha
typedef struct
{
  tipo_elem *topo;
} pilha;
```



Obs: p.topo aponta para o endereço do elemento do topo.

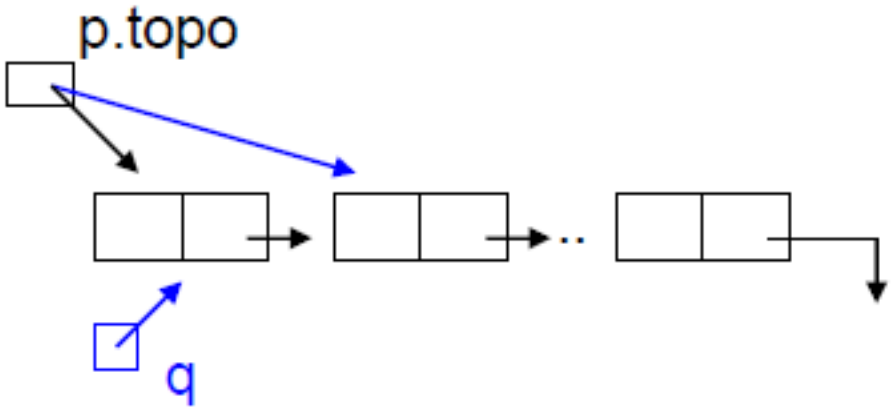
```c
//Cria uma pilha p vazia
void Define(pilha *p)
{
   p->topo = NULL;
}

//Insere x no topo da pilha p (empilha): Push(x, p)
boolean Push(tipo_dado x, pilha *p)
{
   tipo_elem *q = malloc(sizeof(tipo_elem));

   if (q == NULL)  //Nao possui memória disponível (esta cheio)
      return FALSE;

   //Insere x e faz as ligações necessárias
   q->info = x;
   q->lig = p->topo;
   p->topo = q;

   return TRUE;
}
```



```c
//Testa se a pilha p está vazia
boolean Vazia (pilha *p)
{
   return (p->topo == NULL);
}

//Acessa o elemento do topo da pilha (mas sem remove-lo)
//Obs: testar antes da chamada se a pilha não está vazia
tipo_elem *Topo(pilha *p)
{
   return p->topo;
}
```

```c
//Remove o elemento no topo de p sem retornar valor (desempilha v.1)
//Obs: testar antes se pilha não está vazia.
void Pop_up(pilha *p)
{
    tipo_elem *q = p->topo;
    p->topo = p->topo->lig;
    free(q);
}
```

```c
//Remove e retorna o elemento (todo o registro) eliminado (desempilha v.2)
//Obs: testar antes se pilha não está vazia
tipo_elem *Pop(pilha *p)
{
    tipo_elem *q = p->topo;
    p->topo = p->topo->lig;
    return q;
}
```



Pilha dinâmica – vantagens e desvantagens

- **Quando devo usar?**

  - Alocação dinâmica é interessante para pilhas cujo tamanho:
    1. Não pode ser antecipado, ou
    2. É muito variável.

Acabou?

Aplicação de pilhas: sequência de parênteses e colchetes

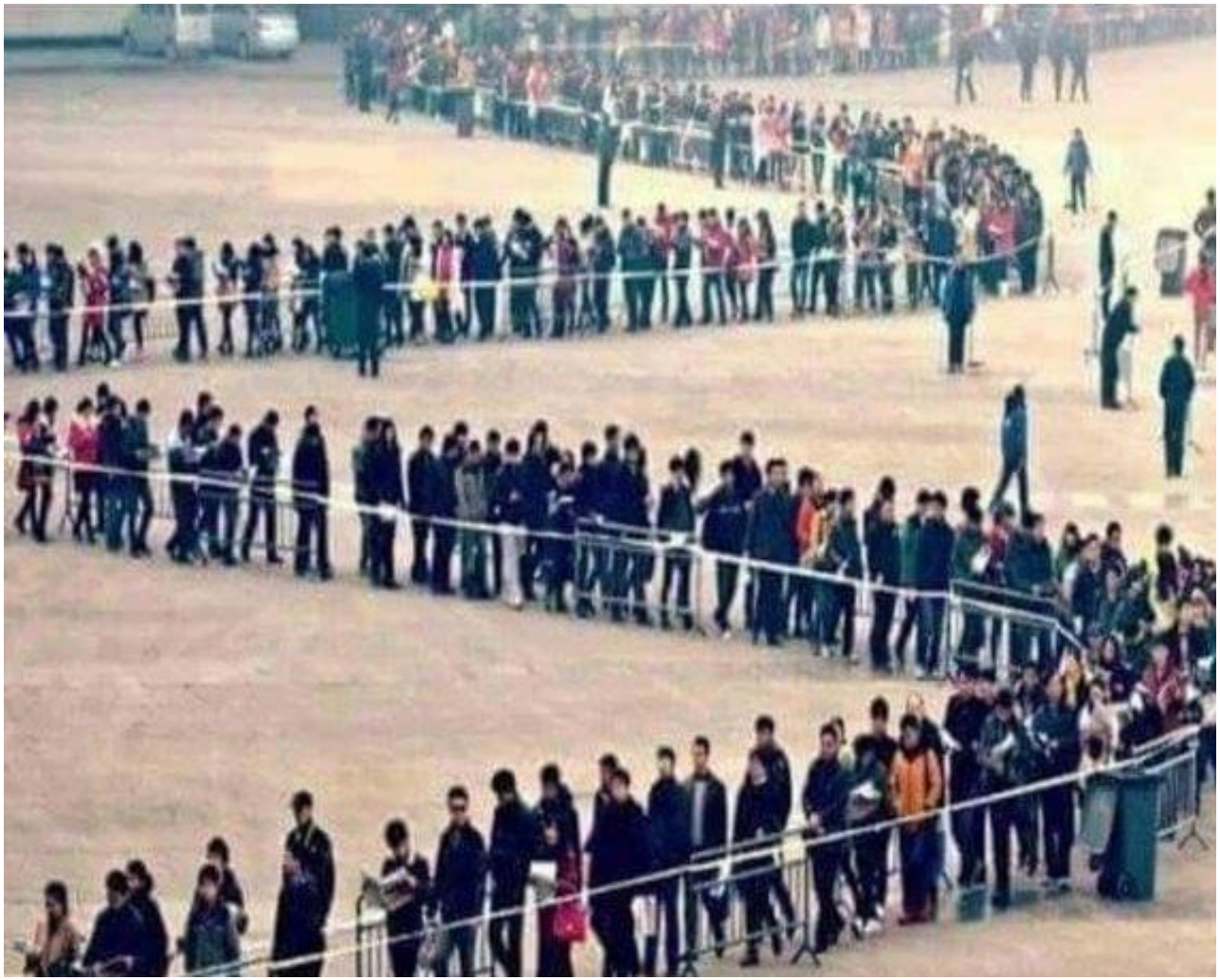
```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define boolean int
#define MAX 101

char pilha[MAX]; //Pilha = conjunto de caracteres (uma string)
int topo;        //Índice do topo

//Funções simples para manusear uma pilha
//---------------------------------------
void Define(void)
{
   topo = -1;
}

void Push(char x)
{
   topo++;
   pilha[topo] = x;
}
```

```
char Pop(void)
{
    char c = pilha[topo];
    topo--;
    return c;
}

boolean Vazia(void)
{
    return (topo == -1);
}
//---------------------------------------
```

```
/* TRUE se a string contém uma seq. válida (parênteses + colchetes), FALSE, c.c.*/
boolean SequenciaValida(char s[]) {
   int i;
   Define();
   for (i = 0; s[i] != '\0'; ++i)
   {
      char c;
      switch (s[i])
      {
         case ')': if (Vazia()) return FALSE;
                   c = Pop(); //Desempilha se encontrar parênteses à direita
                   if (c != '(') return FALSE;
                   break;     //Novo parênteses entrando, então não faz nada
         case ']': if (Vazia()) return FALSE;
                   c = Pop(); //Desempilha se encontrar colchetes à direita
                   if (c != '[') return FALSE;
                   break;     //Novo colchetes entrando, então não faz nada
         default:  Push(s[i]);
      }
   }
   return Vazia();
}
```

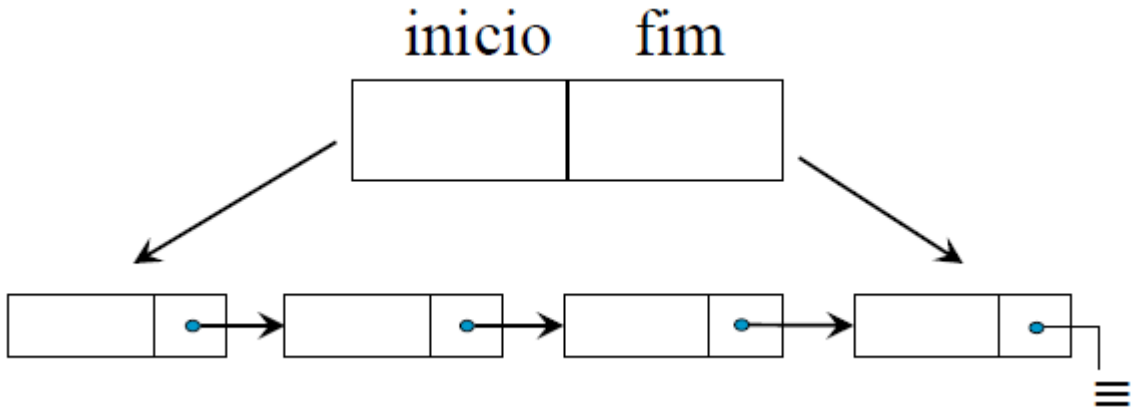
```c
//Principal
int main()
{
    char sequencia[MAX];

    printf("Digite uma sequencia de parênteses e colchetes: ");
    //scanf("%s", &sequencia);
    scanf("%s",  sequencia);

    if(SequenciaValida(sequencia))
      printf("Sequencia valida!");
    else
      printf("Sequencia invalida");

    return 0;
}
```

Fila dinâmica



TAD: fila dinâmica - implementação

```c
//Estruturas e tipos
//-----------------------------------------
//Tipo registro
typedef struct
{
   char nome[30];
   //... (caso tenha mais campos)
} tipo_dado;

//Tipo elemento (unidade de elemento p/ impl. dinâmica)
typedef struct elem
{
   tipo_dado info;
   struct elem *lig;
} tipo_elem;

//Tipo fila
typedef struct
{
   tipo_elem *inicio;
   tipo_elem *fim;
} fila;
//-----------------------------------------------------
```

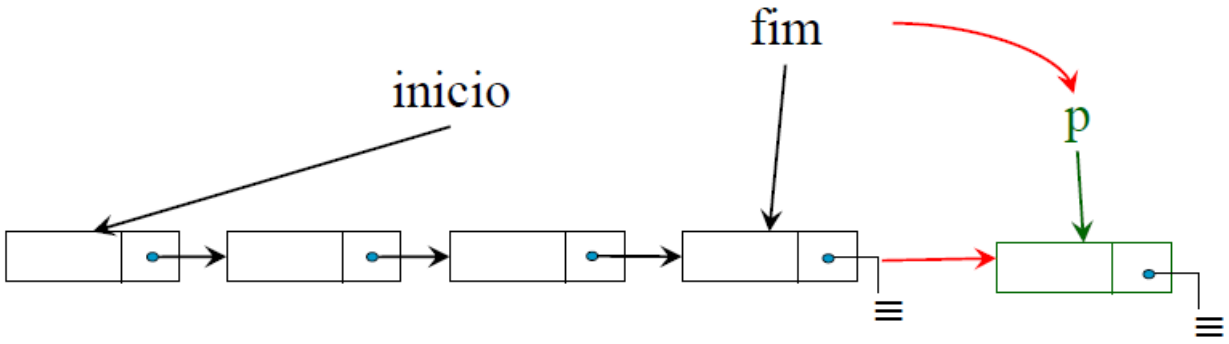


```c
//Cria uma fila vazia (em geral, usado antes de qualquer operacao)
void Definir(fila *q)
{
    q->inicio = NULL;
    q->fim = NULL;
}
```

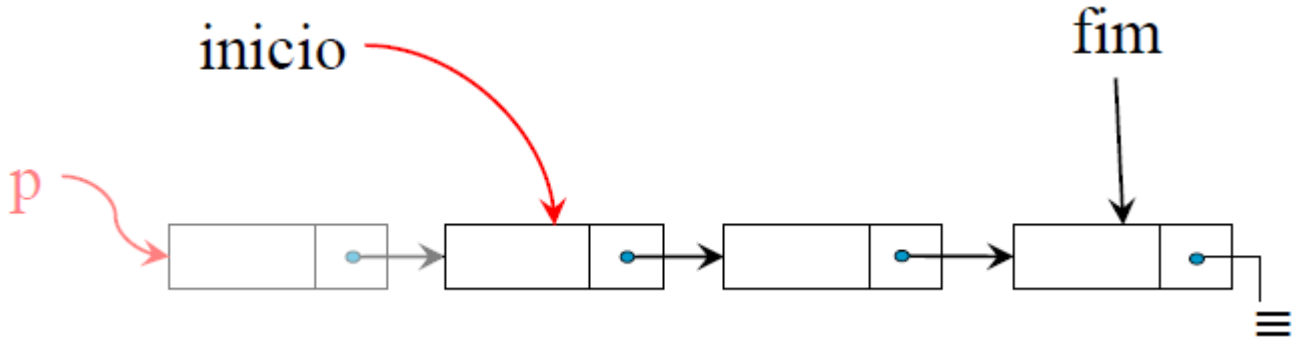
```c
//Verif. lista vazia (retorna true se fila vazia, false c.c.)
boolean Vazia(fila *q)
{
    return (q->inicio == NULL);
}
```

```c
/*Reinicializa uma fila existente q como uma fila vazia
removendo todos os seus elementos.*/
void Tornar_vazia(fila *q)
{
    tipo_elem *ndel, *nextno;

    if(!Vazia(q))
    {
        nextno = q->inicio;
        while (nextno != NULL)
        {
            ndel = nextno;
            nextno = nextno->lig;
            free(ndel);
        }
    }

    Definir(q);
}
```

```
/*Adiciona um elemento no fim da fila q. (retorna true se
operação realizada com sucesso, false caso contrário)*/
boolean Inserir(fila *q, tipo_dado info)
{
   tipo_elem *p;

   p = malloc(sizeof(tipo_elem));
   if (p == NULL)
      return FALSE;

   p->info = info;
   p->lig = NULL;

   if (Vazia(q))
      q->inicio = p;
   else
      q->fim->lig = p;

   q->fim = p;
   return TRUE;
}
```



```
/*Remove um elemento do início da fila q (retorna true se
operação realizada com sucesso, false caso contrário)*/
boolean Remover(fila *q, tipo_dado *info)
{
   tipo_elem *p;

   if (Vazia(q))
      return FALSE;

   p = q->inicio;
   *info = p->info;
   q->inicio = p->lig;

   if (q->inicio == NULL)
      q->fim = NULL;

   free(p);
   return TRUE;
}
```



```c
//Retorna o tamanho da fila
int Tamanho(fila *q)
{
   tipo_elem *p;
   int cont = 0;

   p = q->inicio;
   while(p != NULL)
   {
      cont ++;
      p = p->lig;
   }

   return cont;
}

/*Mostra o começo da fila sem remover o elemento (retorna true
se operação realizada com sucesso, false caso contrário)*/
boolean Inicio_fila(fila *q, tipo_dado *elem)
{
   if (Vazia(q))
      return FALSE;

   *elem = q->inicio->info;
   return TRUE;
}
```

Perguntas?

Fila dinâmica – vantagens e desvantagens

- **Vantagens da Fila Dinâmica:**
  - Ocupa espaço estritamente necessário.
- **Desvantagens da Fila Estática:**
  - Custos usuais da alocação dinâmica (tempo de alocação, e campos para ligações).

Hoje a aula foi tranquila....