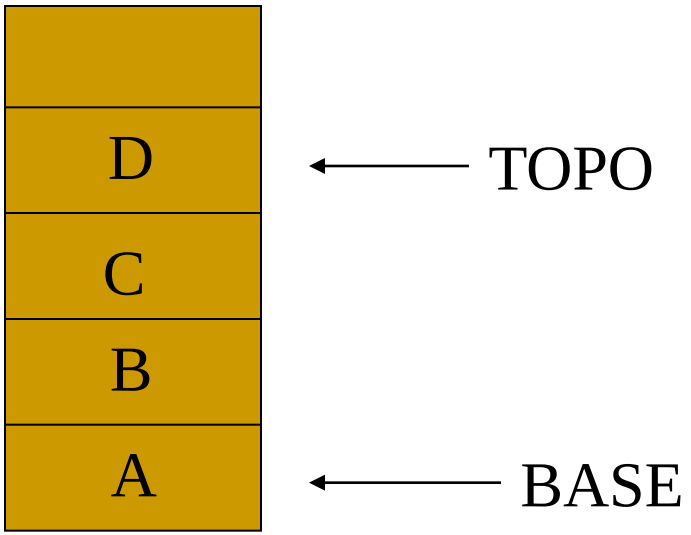
SCC 202– Algoritmos e Estruturas de Dados I

Pilhas (*Stacks*)

(implementação sequencial)

Pilhas

- Pilha: lista linear em que inserção e eliminação de elementos só ocorrem em uma das extremidades (TOPO da pilha)



Pilhas

- Dada uma Pilha P = (a_1, a_2, ..., a_n), dizemos que a_1 é o elemento na base; a_n é o elemento do topo, e a_{i+1} está acima de a_i na pilha
- São também conhecidas como listas do tipo LIFO (*Last In*, *First Out*)

Ex: Pilhas de bandejas no Bandejão

1) Bandejas inicialmente são empilhadas
2) Pega-se (remove-se) bandeja no 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 não querer pegar (remover)

Este ex. faz referência a 4 operações de pilhas:

1) Inserir = *push*
2) Remover = *pop*
3) Verificar se está vazia
4) Examinar o elemento do topo (sem removê-lo)

Exemplos

- Comportamento dos retornos de chamadas a procedimentos

- Inverter listas:

  - (a, b, c, d, e) ➔ (e, d, c, b, a)

TAD Pilha - operações

- ```
  void define (pilha *p);
  /* Cria pilha vazia. Deve ser usada antes de
  qqr outra operação */
  ```
- ```
  boolean push (tipo_info item, pilha *p);
  /* Insere item no topo da pilha. Retorna true
  se sucesso, false c.c. */
  ```
- ```
  boolean vazia (pilha *p);
  /* Retorna true se pilha vazia, false c.c. */
  ```
- ```
  void esvaziar (pilha *p);
  /* Reinicializa pilha */
  ```

TAD Pilha - operações

- `tipo_elem top (pilha *p);`

  `/* Devolve o elemento do topo sem removê-lo. Chamada apenas se pilha não vazia */`
- `void pop_up (pilha *p);`

  `/* Remove item do topo da pilha. Chamada apenas se pilha não vazia */`
- `tipo_elem pop (pilha *p);`

  `/* Remove e retorna o item do topo da pilha. Chamada apenas se pilha não vazia */`

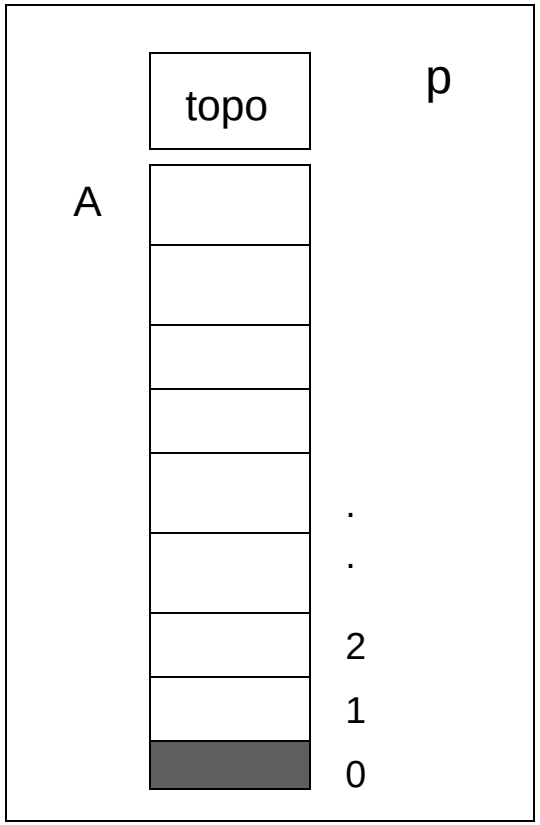
Operações – alocação sequencial

```
#define MAXP 1000

#define indice int

typedef struct{
    tipo_info info;
}tipo_elem;

typedef struct{
    tipo_elem A[MAXP+1];
    indice topo;
}pilha;
```




```
pilha p; /*exemplo de declaração*/
```

1. define (P) - cria uma pilha P vazia

```
void define (pilha *p){
    p->topo = 0;
}
```

2. insere x no topo de P (empilha): push (x, P)

```
boolean push (tipo_info x, pilha *p){

      if (p->topo == MAXP)
            /* pilha cheia */
            return FALSE;

      p->topo ++;
      p->A[p->topo].info = x;
      return TRUE;
}
```

3. testa se P está vazia

```
boolean vazia (pilha *p){
    return (p->topo == 0);
}
```

4. acessa o elemento do topo da pilha (sem remover) - testar antes se a pilha não está vazia!!!

```
tipo_elem top (pilha *p){
    return p->A[p->topo];
}
```

5. remove o elemento no topo de P sem retornar valor (desempilha, v. 1) – testar antes se pilha não está vazia!!!

```
void pop_up (pilha *p){
    p->topo --;
}
```

6. Remove e retorna o elemento (todo o registro) eliminado (desempilha, v. 2) – testar antes se pilha não está vazia!!!

```
tipo_elem pop (pilha *p){
    tipo_elem x = p->A[p->topo];
    p->topo --;
    return x;
}
```

Problemas da implementação sequencial

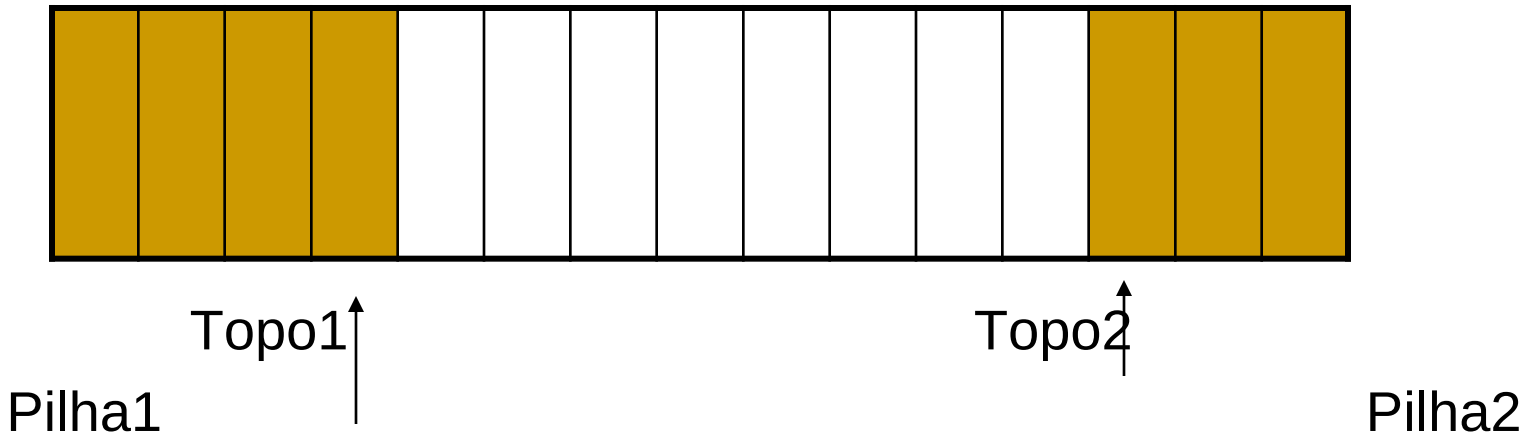
- Nas listas gerais: Necessidade de movimentar itens em inserções e remoções
- **No caso das pilhas**, tais movimentações não ocorrem!
- Alocação sequencial vantajosa a menos de quando não sabemos prever o tamanho máximo da pilha.

Alocação Múltipla de Pilhas

2 pilhas com elementos do mesmo tipo

P1[1..m1] P2[1..m2]

1 único *array*: a[1..M]; M > m1+m2



Conseqüências

- *Overflow* só ocorre se o no. total de elementos de ambas exceder M
- Bases fixas

    início: P.Topo1 = 0 cresce à direita

    P.Topo2 = M+1 cresce à esq.

Inserção na Pilha 1

```
if P. topo1  < P. topo2 – 1 then
   begin
         P.topo1:= P.topo1+1;
         P.A[topo1]:= x
   end
else
   “overflow”
```

Inserção na Pilha 2

```
if P.topo2  > P.topo1 + 1 faça
   begin
        P.topo2:= P.topo2-1;
        P.A[topo2]:= x
   end
else
   “overflow”
```

N Pilhas

- Bases não podem ser fixas para garantir o melhor aproveitamento do espaço