Estruturas de Dados 1

13 – Pilhas Estáticas e Dinâmicas

Antonio Angelo de Souza Tartaglia
angelot@ifsp.edu.br

Estrutura de Dados 1

Pilha

- O conceito de Pilha é algo muito comum para as pessoas. Uma Pilha é um conjunto finito de itens de um mesmo tipo, mas diferente das Listas e Filas, os elementos de uma Pilha encontram-se dispostos uns sobre os outros.

- Assim, somente é possível a inserção de um novo elemento na Pilha se este for colocado acima dos demais. Por esse motivo, somente é possível a remoção de elementos em seu topo.

- As Pilhas são implementadas e se comportam de modo muito similar às Listas, e são muitas vezes consideradas um tipo especial de Lista, em que as inserções e remoções são realizadas em apenas na mesma extremidade.

Estrutura de Dados 1

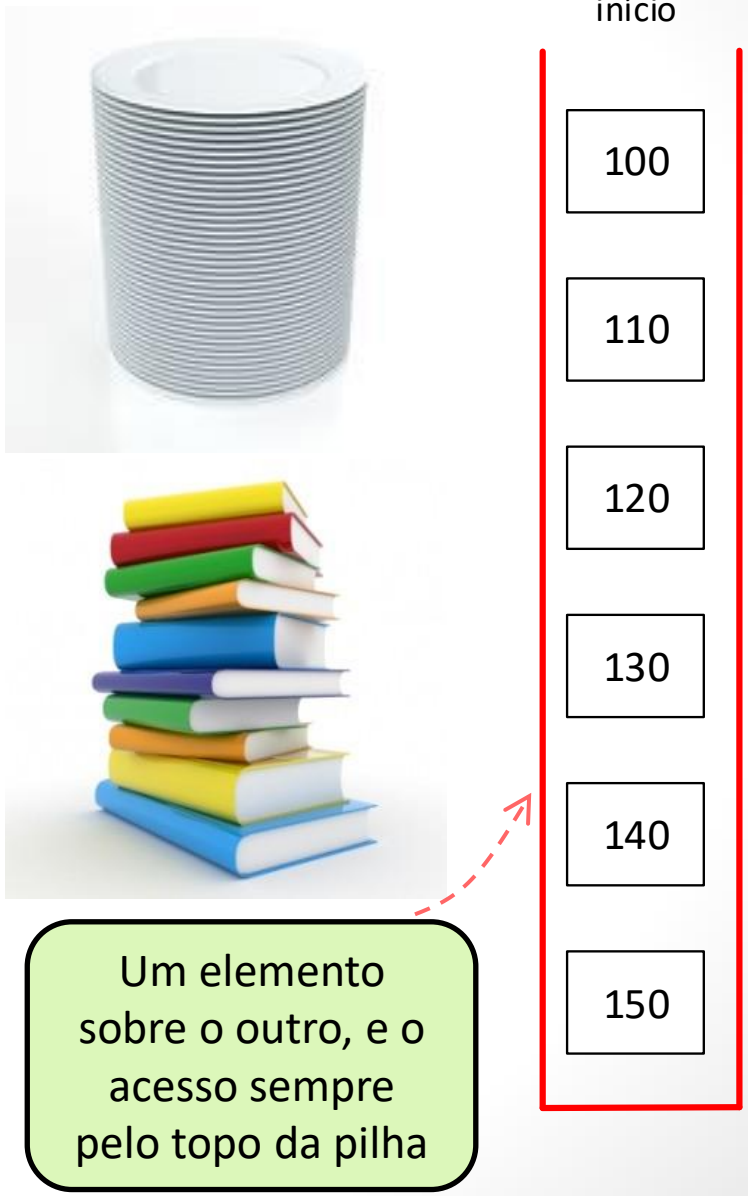
Pilha

- Desse modo, se é necessário acessar determinado elemento da Pilha, antes será preciso remover todos os elementos que estiverem sobre o elemento que se quer acessar.

- Por esse motivo, as Pilhas são conhecidas como estruturas de dados do tipo **último a entrar, primeiro a sair** ou `LIFO` (***L**ast **I**n, **F**irst **O**ut*):

  - Os elementos são removidos da Pilha na ordem inversa daquela em que foram inseridos.

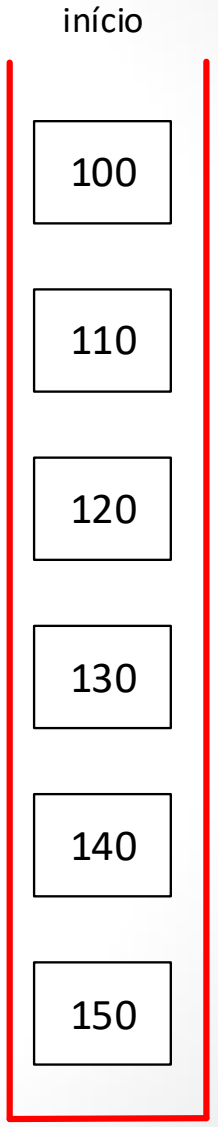
  - Uma Pilha é uma estrutura de dados linear utilizada para organizar dados em um computador, que armazena elementos do mesmo tipo.



Estrutura de Dados 1

Pilha

- Seus elementos possuem estrutura interna abstraída, ou seja, sua complexidade é arbitrária e não afeta seu funcionamento.
- Além disso, uma Pilha pode possuir elementos repetidos, dependendo da aplicação que a implemente.
- Uma Pilha pode possuir $n \ (n \geq 0)$ elementos ou itens.
- Se $n = 0$, dizemos que a pilha está vazia.



Não é possível remover um elemento no meio da pilha, sem desmanchá-la

Pilha

- Tipos de Pilhas
- Basicamente, existem dois tipos de implementações principais para uma Pilha. Essas implementações se diferenciam no tipo de alocação de memória usada, e no tipo de acesso aos seus elementos. Uma Pilha pode ser implementada utilizando alocação estática com acesso sequencial ou alocação dinâmica com acesso encadeado:

  - Alocação estática – O espaço de memória é alocado no momento da compilação do programa, ou seja, é necessário definir o número máximo de elementos que a Pilha suportará. Seus elementos são armazenados de forma consecutiva na memória, porém, o acesso é permitido apenas ao primeiro elemento da Pilha.
  - Alocação dinâmica – O espaço de memória é alocado em tempo de execução, ou seja, a Pilha cresce à medida em que novos elementos são armazenados e diminui quando são removidos. Nessa implementação, cada elemento pode estar em uma área distinta da memória, não necessariamente de forma consecutiva. Por esse motivo cada elemento deve armazenar além de sua informação, o endereço de memória onde se encontra o próximo elemento.

Pilha sequencial estática

- Uma Pilha sequencial estática, é uma estrutura de dados definida utilizando-se a alocação estática e acesso sequencial aos seus elementos.

- Trata-se do tipo de Pilha mais simples possível, sendo definida utilizando-se um *array*, e dessa forma, o sucessor de um elemento ocupa a posição física seguinte do mesmo.

- Além do *array*, este projeto Pilha será implementado utilizando um campo adicional (`qtd`) que servirá para indicar o quanto do *array* já está ocupado pelos elementos já inseridos.

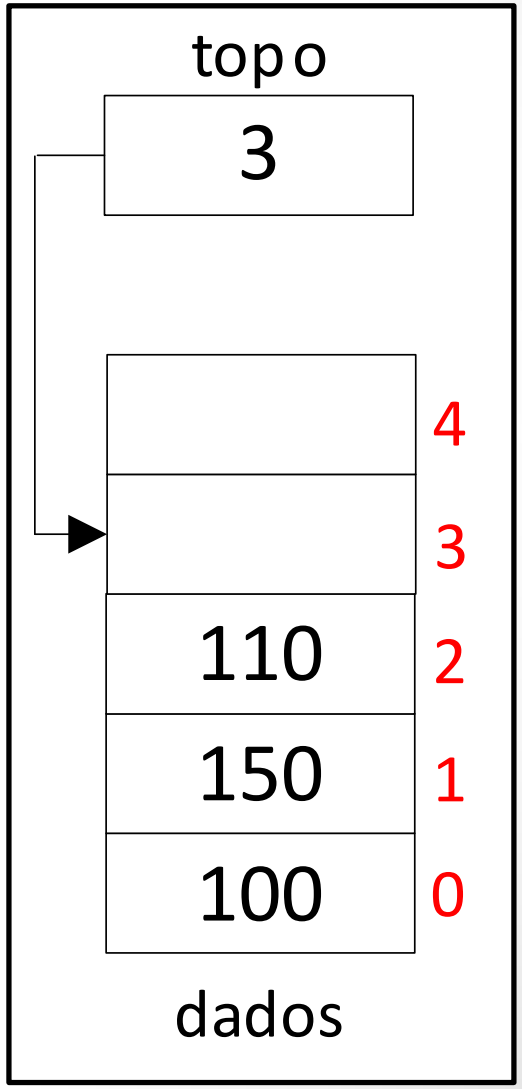
Pilha sequencial estática

- Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo PILHA (que será o tipo opaco). Isso impede ao usuário/programador saber como a Pilha foi realmente implementada, além de limitar o seu acesso à apenas as funções que manipulam o topo da Pilha.

- A principal vantagem em se utilizar um *array* na definição de uma Pilha estática, está na facilidade de se criar e destruir esta estrutura.

- Já a sua principal desvantagem é a necessidade de definir previamente o tamanho de seu *array*, e consequentemente a quantidade de elementos que a Pilha suportará.

- Considerando as vantagens e desvantagens, o ideal é utilizar este tipo de Pilha para poucos elementos, ou quando o número máximo que ela suportará é bem definido, e nunca será ultrapassando.



Pilha sequencial estática

- Para sua implementação, é necessário definir o tipo de dado que será armazenado. Uma Pilha pode armazenar qualquer tipo de informação.

- Para tanto, é necessário especificar isso em sua declaração. Como estamos trabalhando com modularização, também definiremos o tipo opaco que representará a Pilha. Este tipo será um ponteiro para a estrutura que define a Pilha.

- Além disso, também serão definidas as funções que serão visíveis no `main()` para o usuário/programador que utilizar esta biblioteca. Tudo aquilo que será visível no `main()` deve ser declarado no arquivo `pilha.h`.

- Tudo que ficará oculto do usuário/programador, será implementado no arquivo `fila.c`.

Pilha sequencial estática

- Em `pilha.h` serão declarados:
  - O tamanho máximo do *array* utilizado na Pilha, representado pela constante `MAX`;
  - O tipo de dado que será armazenado na Pilha, a estrutura `ALUNO`; composto por 4 campos, `matricula`, `n1`, `n2` e `n3`.
  - Um novo nome será definido para o tipo Pilha, por questões de padronização e compatibilidade. Por meio do comando `typedef`, o novo tipo será chamado `PILHA`, e será o tipo a ser utilizado sempre que ser for trabalhar com esta estrutura de dados.
  - Os protótipos das funções que estarão disponíveis para se trabalhar com a Pilha, e que serão implementadas no arquivo `pilha.c`.
- Em `pilha.c` serão declarados:
  - As chamadas necessárias à implementação da estrutura Pilha.
  - A definição do tipo opaco, `struct pilha`, que só será acessível através de ponteiros pelas funções disponibilizadas para isso.
  - A implementação das funções que manipularão os dados em `struct pilha`.

Pilha sequencial estática

- Variáveis para utilização no programa
  - A variável x será utilizada para o retorno de informações de execução das funções de manipulação do tipo de dado Pilha.
  - As variáveis `al1`, `al2` e `al3` serão inseridas na Fila. A variável `al_consulta` será utilizada para o retorno de dados da função consulta.

```c
//Arquivo main.c
#include <stdio.h>
#include <stdlib.h>
#include "pilha.h"

int main()
{
    int x; //para os codigos de erro
    ALUNO al_consulta, al1, al2, al3;
    al1.matricula = 100;
    al1.n1 = 8.3;
    al1.n2 = 8.4;
    al1.n3 = 8.5;

    al2.matricula = 110;
    al2.n1 = 7.3;
    al2.n2 = 7.4;
    al2.n3 = 7.5;

    al3.matricula = 120;
    al3.n1 = 6.3;
    al3.n2 = 6.4;
    al3.n3 = 6.5;
```

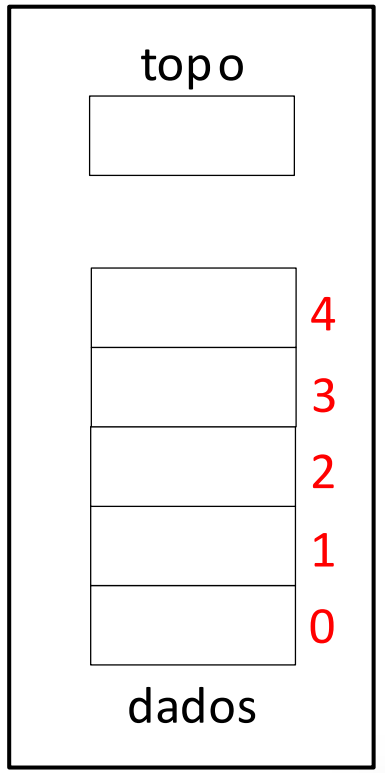
Pilha sequencial estática

- Implementando a Pilha estática:

```c
//arquivo pilha.h
#define MAX 100

typedef struct aluno{
    int matricula;
    float n1, n2, n3;
}ALUNO;

typedef struct pilha *PILHA;
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
#include <stdio.h>
#include <stdlib.h>
#include "pilha.h"

//definição do tipo PILHA
struct pilha{
    int qtd;
    ALUNO dados[MAX];
};
```

```c
//arquivo main.c
//ponteiro para o topo
PILHA *pi = NULL;
```



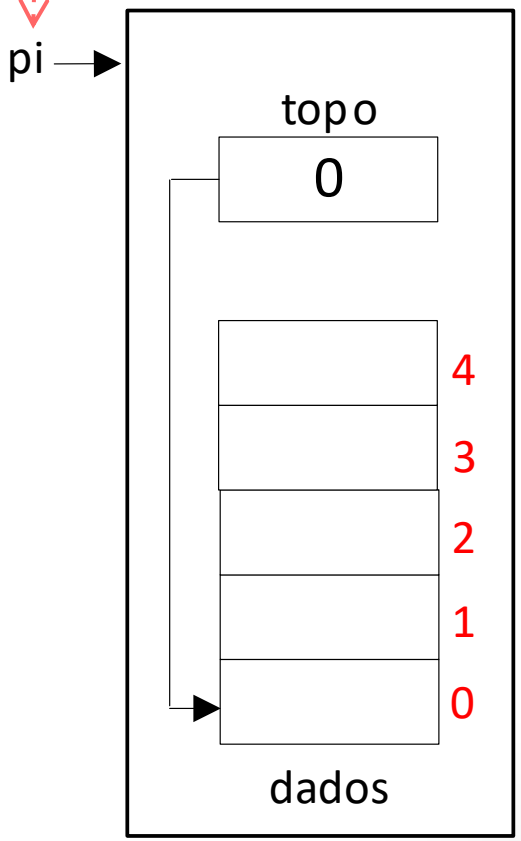
Pilha sequencial estática

- Criando a Pilha estática:

```c
//arquivo pilha.h
PILHA *criaPilha();
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
PILHA *criaPilha(){
    PILHA *pi;
    pi = (PILHA*) malloc(sizeof(PILHA));
    if(pi != NULL){
        pi->qtd = 0;
    }
    return pi;
}
```

```c
//arquivo main.c
pi = criaPilha();
```



Pilha sequencial estática

- Destruir uma Pilha estática é bastante simples, basicamente o que tem que se fazer é liberar a memória alocada para toda a estrutura que representa a Pilha. Isso pode ser feito com apenas uma chamada à função `free()`. Então por que criar uma função para destruir a fila, se podemos apenas chamar a dita função?
- Por questões de modularização. Destruir uma Pilha estática é muito simples, mas destruir uma Pilha alocada dinamicamente é uma tarefa mais complicada. Ao criar esta função, estamos escondendo a implementação dessa tarefa do usuário/programador, e ao mesmo tempo mantemos a mesma notação utilizada por uma Pilha com alocação dinâmica.
- Dessa forma poderemos trocar os módulos (estático pelo dinâmico e vice e versa) entre si, e o funcionamento da Pilha para o usuário/programador será sempre o mesmo, apesar terem as suas implementações completamente diferentes.

```c
//arquivo main.c
liberaPilha(pi);
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
void liberaPilha(PILHA *pi){
    free(pi);
}
```

```c
//arquivo pilha.h
void liberaPilha(PILHA *pi);
```

Pilha sequencial estática

- Abortando o programa em caso de erro de alocação:

```
//arquivo pilha.C - PILHA ESTÁTICA
void abortaPrograma(){
    printf("ERRO! Pilha nao foi alocada, ");
    printf("programa sera encerrado...\n\n\n");
    system("pause");
    exit(1);
}
```

- Em todas as funções que manipulam e acessam os dados encapsulados, deve-se testar antes da manipulação dos dados, se a Pilha foi realmente alocada.

- Caso não tenha sido alocada por qualquer motivo, não teremos uma Pilha válida para trabalhar. O programa então deverá ser abortado, antes porém, informando ao usuário sobre a falha na alocação, que portanto, impede o correto funcionamento do programa.

Pilha sequencial estática

- Coletando informações básicas sobre a Pilha:
- Assim como na Fila, as operações de inserção, remoção e consulta são consideradas as principais operações de uma Pilha.
- Apesar disso, para realizar estas operações, torna-se necessário ter algumas outras informações mais básicas sobre a Pilha. Por exemplo, não é possível a remoção de um elemento da Pilha se a mesma estiver vazia, ou se há uma tentativa de inserção em uma Pilha que está cheia.
- Também útil, a informação de quantos elementos estão empilhados, nos dá uma ideia de quanto espaço ainda temos antes que a Pilha fique cheia, impossibilitando o empilhamento de novos elementos.

Pilha sequencial estática

- Verificando o tamanho de uma Pilha:

- Basicamente, verificar o tamanho de uma Pilha consiste em acessar e retornar o valor de seu campo `qtd`, que armazena a quantidade de elementos que estão inseridos na Pilha.

```c
//arquivo pilha.h
int tamanhoPilha(PILHA *pi);
```

```c
//arquivo main.c
x = tamanhoPilha(pi);
printf("\nO tamanho da Pilha e: %d", x);
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
int tamanhoPilha(PILHA *pi){
    if(pi == NULL){
        abortaPrograma();
    }else{
        return pi->qtd;
    }
}
```

Pilha sequencial estática

- Verificando se a Pilha está cheia:

- De forma similar, a verificação de Pilha cheia, consiste em acessar o campo `qtd` e testar se seu valor é igual a constante MAX, que define o número máximo de elementos que a Pilha suporta:

```c
//arquivo pilha.h
int pilhaCheia(PILHA *pi);
```

```c
//arquivo main.c
x = pilhaCheia(pi);
if(x){
    printf("\nA Pilha esta cheia!");
}else{
    printf("\nA Pilha nao esta cheia.");
}
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
int pilhaCheia(PILHA *pi){
    if(pi == NULL){
        abortaPrograma();
    }
    return (pi->qtd == MAX);
}
```

Pilha sequencial estática

- Verificando se a Pilha está vazia:

- Também de forma similar, a verificação de Pilha vazia, consiste em acessar o campo `qtd` e testar se seu valor é igual a `0` (zero), indicando então que não há elementos na Pilha:

```c
//arquivo pilha.h
int pilhaVazia(PILHA *pi);
```

```c
//arquivo main.c
x = pilhaVazia(pi);
if(x){
    printf("\nA Pilha esta vazia!");
}else{
    printf("\nA Pilha nao esta vazia.");
}
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
int pilhaVazia(PILHA *pi){
    if(pi == NULL){
        abortaPrograma();
    }
    return (pi->qtd == 0);
}
```

Pilha sequencial estática

- Inserindo elementos na Pilha:

- Após a verificação de alocação da Pilha, é necessário verificar se ainda há espaço disponível para inserção. Caso não exista espaço a inserção é cancelada. Se houver espaço, basta copiar os dados a serem armazenados para dentro da posição `pi->qtd` e após, incrementar este mesmo campo:

```c
//arquivo pilha.h
int inserePilha(PILHA *pi, ALUNO al);
```

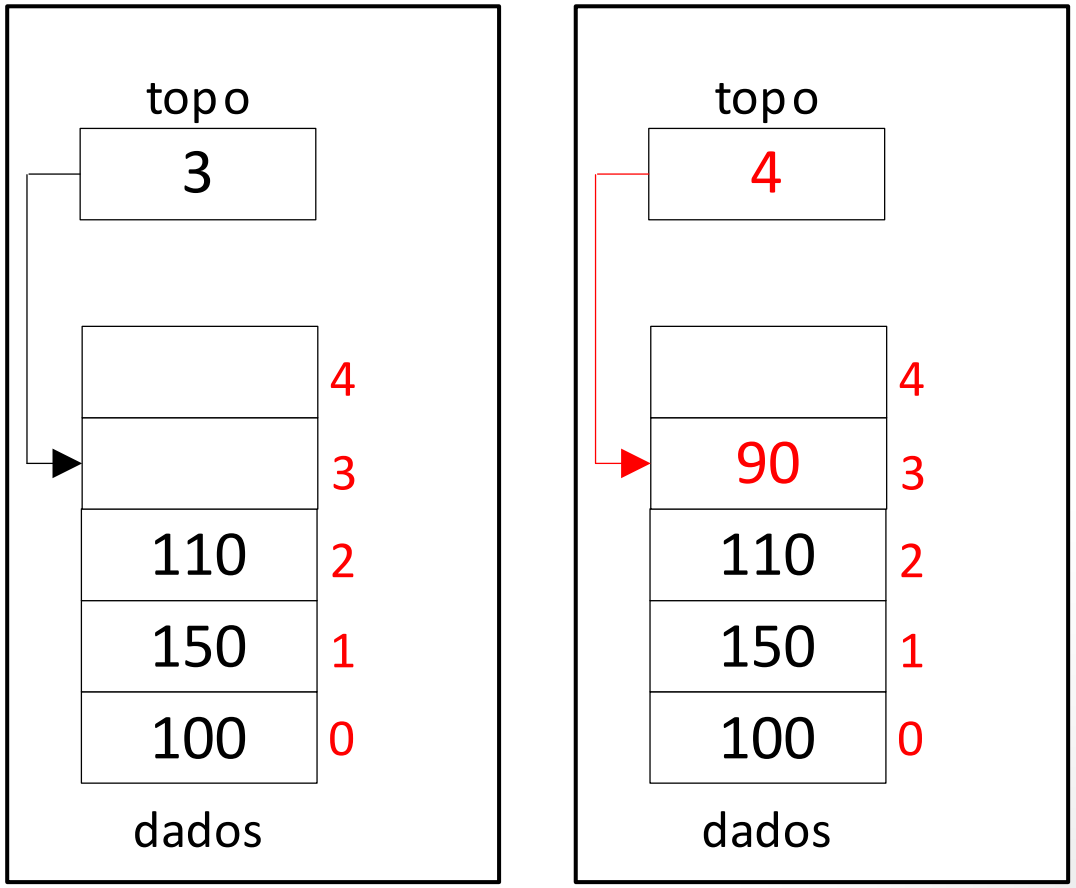
```c
//arquivo main.c
x = inserePilha(pi, al1);
if(x){
    printf("\nElemento %d inserido com sucesso!", x);
}else{
    printf("\nErro, elemento nao inserido.");
}
```

```c
//arquivo pilha.C - PILHA ESTÁTICA
int inserePilha(PILHA *pi, ALUNO al){
    if(pi == NULL){
        abortaPrograma();
    }
    if(pi->qtd == MAX){
        return 0;
    }
    pi->dados[pi->qtd] = al;
    pi->qtd++;
    return al.matricula;
}
```

Pilha sequencial estática

- Inserindo elementos na Pilha:

```
//arquivo pilha.C - PILHA ESTÁTICA
int inserePilha(PILHA *pi, ALUNO al){
    if(pi == NULL){
        abortaPrograma();
    }
    if(pi->qtd == MAX){
        return 0;
    }
    pi->dados[pi->qtd] = al;
    pi->qtd++;
    return al.matricula;
}
```



Pilha sequencial estática

- Removendo elementos na Pilha:

- Na função de remoção, após a verificação de alocação que garante que temos uma Pilha válida para trabalho, testamos se a Pilha está vazia, condição em que não há possibilidade de remoções. Como a remoção é realizada no topo da Pilha, basta decrementar em uma unidade o valor do campo `qtd`, indicando assim, que aquela posição apontada por `qtd`, está agora disponível para uma inserção:

```c
//arquivo pilha.h
int removePilha(PILHA *pi);
```

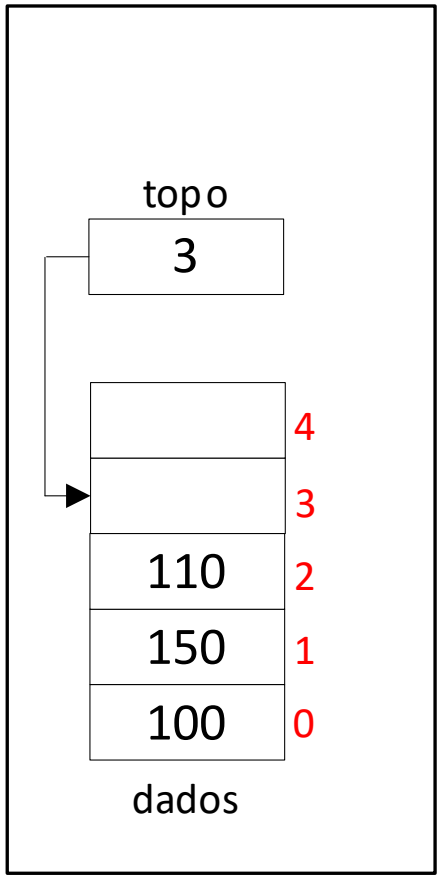
```c
//arquivo main.c
x = removePilha(pi);
if(x){
    printf("\nElemento %d removido com sucesso!", x);
}else{
    printf("\nErro, elemento nao removido.");
}
```

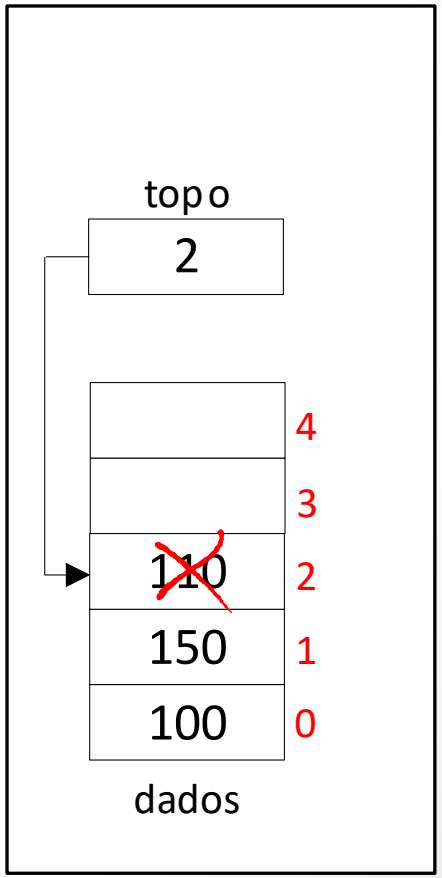
```c
//arquivo pilha.C - PILHA ESTÁTICA
int removePilha(PILHA *pi){
    int matricula;
    if(pi == NULL){
        abortaPrograma();
    }
    matricula = pi->dados[pi->qtd - 1].matricula;
    pi->qtd--;
    return matricula;
}
```

Pilha sequencial estática

- Removendo elementos na Pilha:

```
//arquivo pilha.C - PILHA ESTÁTICA
int removePilha(PILHA *pi){
    int matricula;
    if(pi == NULL){
        abortaPrograma();
    }
    matricula = pi->dados[pi->qtd - 1].matricula;
    pi->qtd--;
    return matricula;
}
```





Pilha sequencial estática

- Consultando elementos na Pilha:

- Assim como na estrutura Fila, também nas Pilhas somente é possível a consulta do elemento que está no seu topo. Caso houvesse consulta aos outros elementos empilhados, esta estrutura deixaria de ser uma Pilha...

```
//arquivo pilha.h
int acessaTopoPilha(PILHA *pi, ALUNO *al);
```

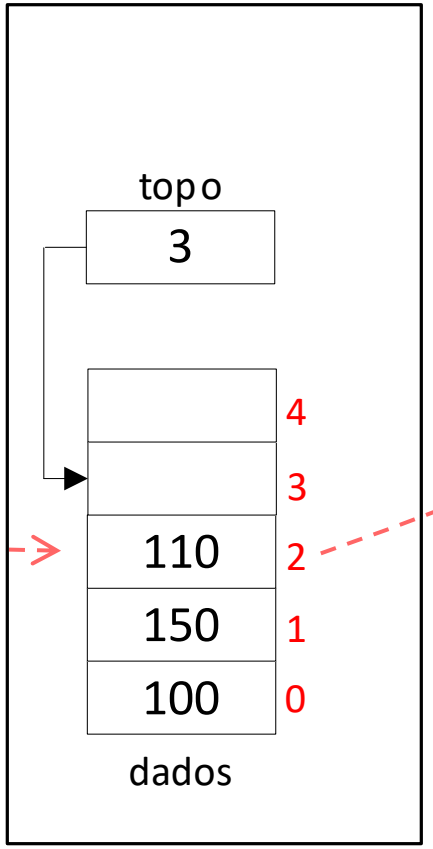
```
//arquivo main.c
x = acessaTopoPilha(pi, &al_consulta);
if(x){
    printf("\nConsulta realizada com sucesso:");
    printf("\nMatricula: %d", al_consulta.matricula);
    printf("\nNota 1:    %.2f", al_consulta.n1);
    printf("\nNota 2:    %.2f", al_consulta.n2);
    printf("\nNota 3:    %.2f", al_consulta.n3);
}else{
    printf("\nErro, consulta nao realizada.");
}
```

```
//arquivo pilha.C - PILHA ESTÁTICA
int acessaTopoPilha(PILHA *pi, ALUNO *al){
    if(pi == NULL){
        abortaPrograma();
    }
    if(pi->qtd == 0){
        return 0;
    }
    *al = pi->dados[pi->qtd - 1];
    return 1;
}
```

Pilha sequencial estática

- Consultando elementos na Pilha:

```
//arquivo pilha.C - PILHA ESTÁTICA
int acessaTopoPilha(PILHA *pi, ALUNO *al){
    if(pi == NULL){
        abortaPrograma();
    }
    if(pi->qtd == 0){
        return 0;
    }
    *al = pi->dados[pi->qtd - 1];
    return 1;
}
```

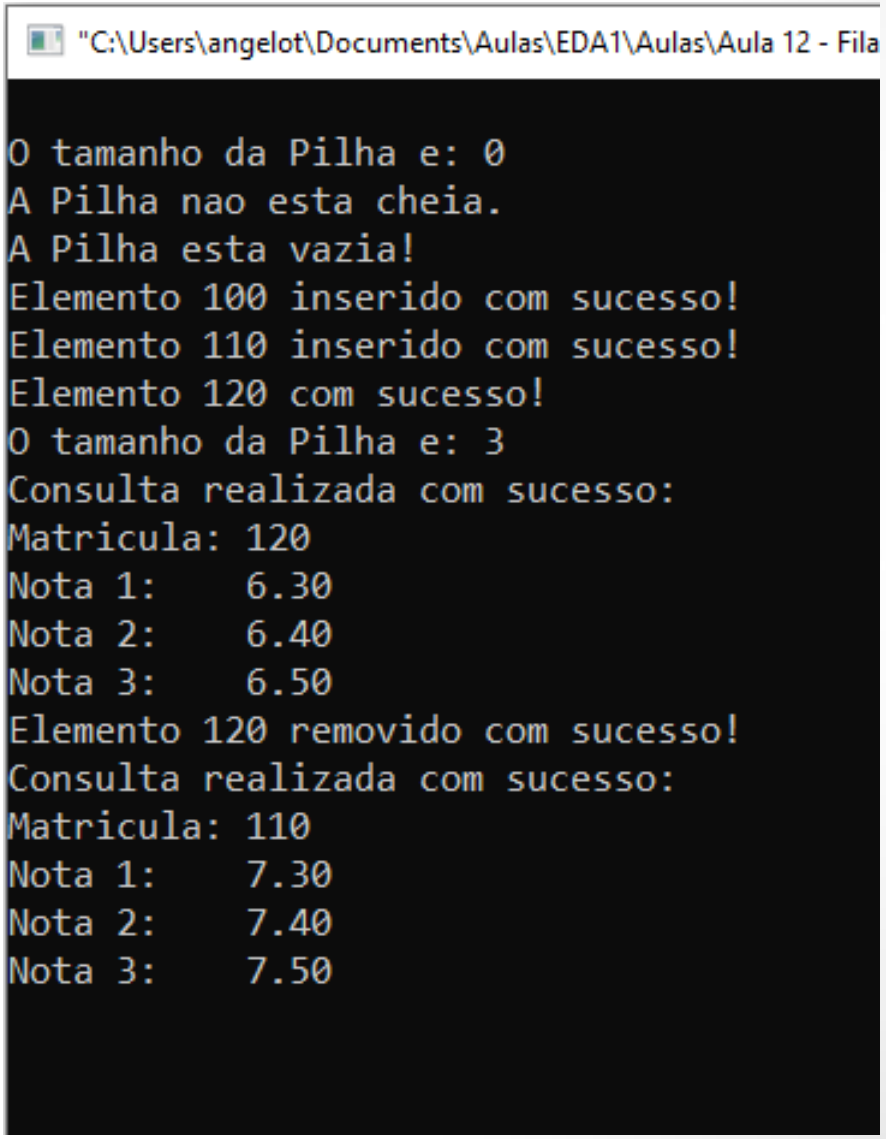



```
Consulta realizada com sucesso:
Matricula: 110
Nota 1:    7.30
Nota 2:    7.40
Nota 3:    7.50
```

Atividade 1

- Atividade 1 – Pilha estática:

- Monte o programa Pilha estática, posicionando as chamadas das funções no `main()`, para manipulação da Pilha, de forma que apresentem o funcionamento e as mensagens iguais ao exemplo ao lado. Entregue no Moodle todos os arquivos zipados.



Pilha sequencial dinâmica

- Em uma Pilha sequencial dinâmica o acesso aos seus elementos é realizado de forma encadeada, e seus elementos são alocados em tempo de execução. Por esse motivo, o tamanho da Pilha cresce a medida que novos elementos são inseridos e diminui quando estes elementos são removidos.

- Neste tipo de Pilha, cada elemento pode estar em uma área distinta da memória e de forma não necessariamente consecutiva. Uma vez que seus blocos de memória alocada são pequenos, podem assim dispersarem-se, ocupando pouco, ou quase nenhum espaço consecutivo em memória.

- Portanto, estes blocos pequenos de memória alocada, devem armazenar além de sua informação, o endereço de memória onde se encontra o bloco alocado que é o seu sucessor na Pilha

Pilha sequencial dinâmica

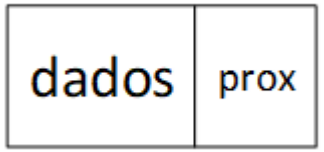
- Assim como na estrutura Fila, o módulo dinâmico deste programa foi construído para ser compatível função a função com a Pilha estática, bastando para isso, substituir o módulo `pilha.c` (estático) pelo módulo que veremos agora, o `pilha.c` (dinâmico), trocando um arquivo pelo outro.

- Observe que por ter sua implementação em módulos, o usuário/programador que utilizar esta biblioteca é impedido de acessar o módulo encapsulado, não tendo assim o conhecimento do código interno e qual tipo de Pilha está usando (estática ou dinâmica), uma vez que em uma distribuição real, os módulos `pilha.c` são distribuídos pré-compilados, não permitindo o acesso e leitura de seu conteúdo.

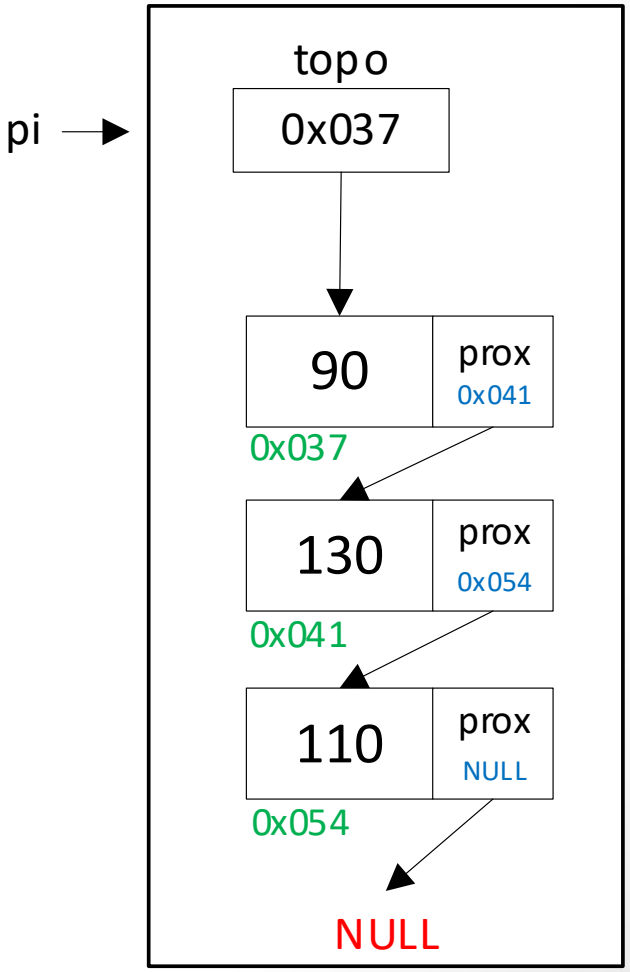
- Essa é a grande vantagem da modularização e da utilização dos tipos opacos e encapsulamento: mudar a maneira como a Pilha foi implementada não altera nem interfere no funcionamento do programa que a utiliza.

Pilha sequencial dinâmica

- O topo da Pilha, nada mais é do que um ponteiro para uma estrutura chamada `struct pilha` que contém dois campos de informação:
  - Um campo para o dado, utilizado para armazenar a informação que será inserida na pilha, e;
  - Um campo chamado de `prox`, que é um ponteiro que indica o próximo elemento da Pilha.



- Além dessa estrutura que define seus elementos, esse tipo de Pilha utiliza ainda um ponteiro para ponteiro para guardar a localização do primeiro elemento, ou topo, da Pilha.



Pilha sequencial dinâmica

- Todos os elementos de uma estrutura do tipo Pilha dinâmica, são ponteiros, alocados dinamicamente em tempo de execução. Para inserir um elemento no topo, ou início da Pilha, é necessário utilizar um campo que seja fixo, mas que ao mesmo tempo seja capaz de apontar para um novo elemento.

- É necessário então, o uso de um ponteiro para ponteiro para guardar o endereço de outro ponteiro,e dessa forma, com um ponteiro para ponteiro representando o topo da pilha, fica fácil mudar quem é o primeiro elemento da pilha, apenas mudando o conteúdo do topo.

- Após o último elemento não existe nenhum novo elemento. Assim, o último elemento da pilha aponta para `NULL`.

- O usuário terá acesso apenas a um ponteiro do tipo `PILHA`. Isso o impede de saber como foi realmente implementada a Pilha, e limita o seu acesso apenas às funções que a manipulam.

Pilha sequencial dinâmica

- A principal vantagem de se utilizar uma abordagem dinâmica e encadeada na definição da Pilha, é a melhor utilização dos recursos de memória, uma vez que não é mais necessário definir previamente o seu tamanho.

- Sua principal desvantagem é a necessidade de percorrer toda a pilha para destruí-la.

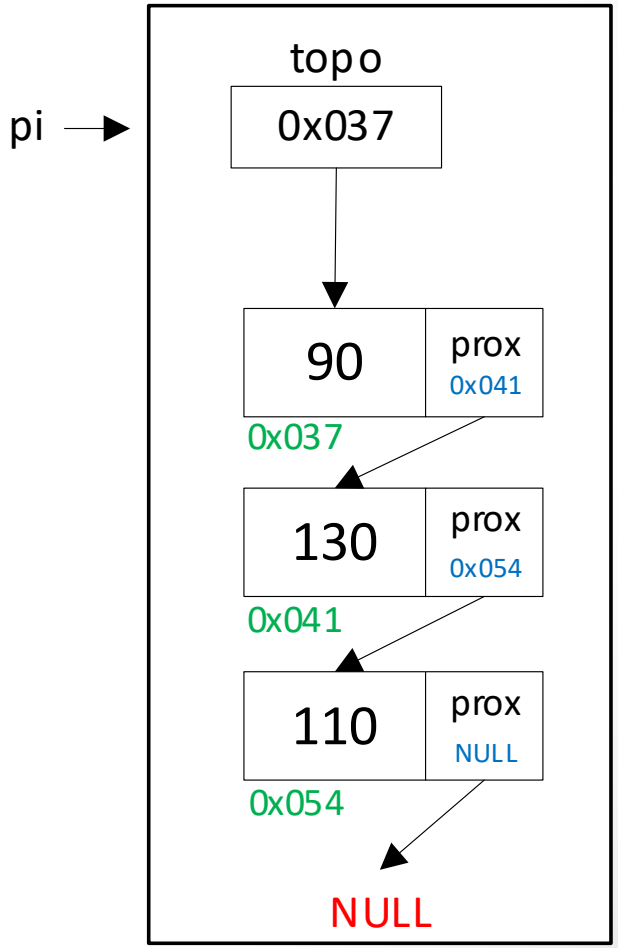
- Considerando as vantagens e desvantagem, o cenário ideal para a utilização para este tipo de Pilha, é quando não há a necessidade de garantir um espaço mínimo para a execução da aplicação, ou quando o tamanho máximo que a Pilha deverá suportar, não é bem definido.

Pilha sequencial dinâmica

- Implementação:

```c
//Arquivo pilhaDinamica.c
#include <stdio.h>
#include <stdlib.h>
#include "pilha.h"

struct pilha{
    ALUNO dados;
    struct pilha *prox;
};

typedef struct pilha ELEM;
```

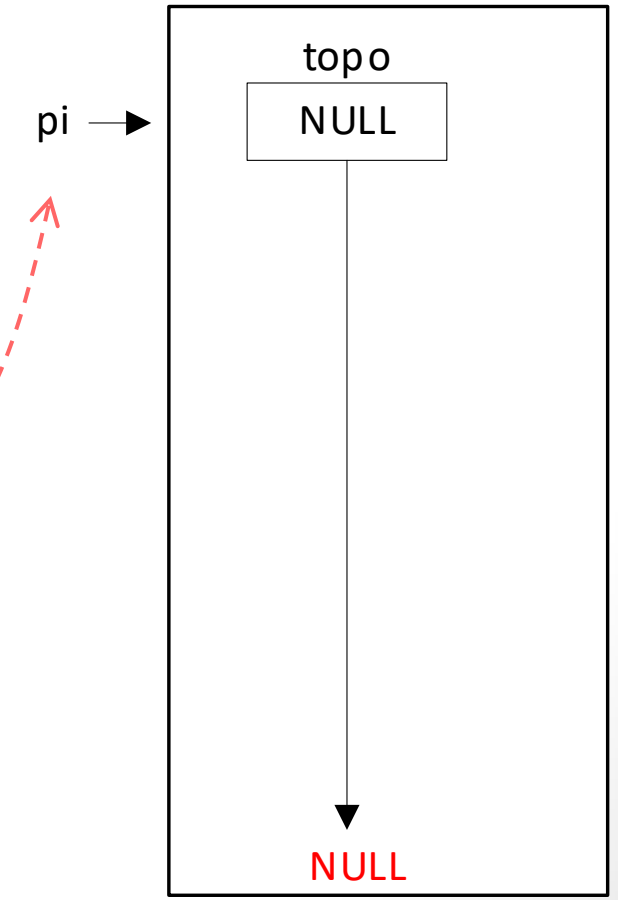
Apenas para não digitar muito a todo instante...



Pilha sequencial dinâmica

- O ato de criação da pilha, se resume a alocar espaço para o tipo PILHA, e, se a alocação for bem sucedida, fazer com o topo aponte para NULL indicando que a Pilha esta vazia, sem elementos:

```c
//Arquivo pilhaDinamica.c
PILHA *criaPilha(){
    PILHA *pi = (PILHA*) malloc(sizeof(PILHA));
    if(pi != NULL){
        *pi = NULL;
    }
    return pi;
}
```



Pilha sequencial dinâmica

- Destruir uma Pilha dinâmica não é uma tarefa tão simples quanto destruir uma Pilha estática. Para liberar a memória utilizada pelos elementos da Pilha, é necessário percorre-la por inteiro, em todos os seus elementos, liberando a memória de um por um:

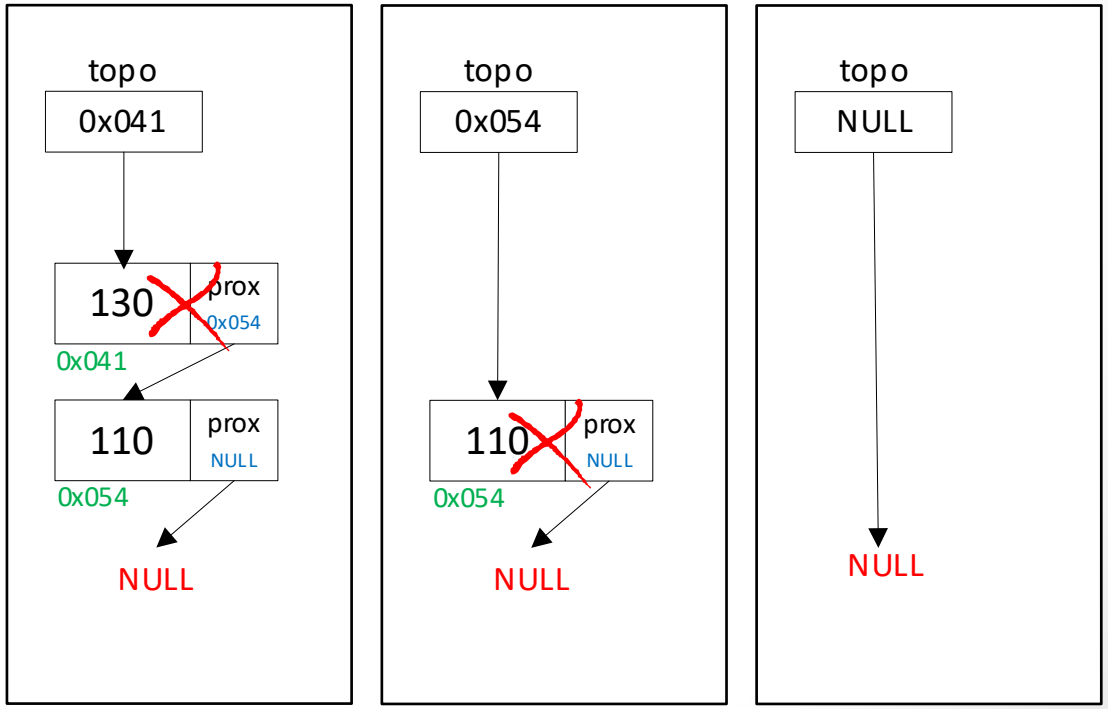
```
//Arquivo pilhaDinamica.c
void liberaPilha(PILHA *pi){
    if(pi != NULL){
        ELEM *no;
        while ((*pi) != NULL){
            no = *pi;
            *pi = (*pi)->prox;
            free(no);
        }
        free(pi);
    }
}
```

Recebe o 1º elemento da Pilha.

Libera o antigo topo da Pilha.

O topo recebe o próximo elemento da Pilha.

Remove o topo.



Pilha sequencial dinâmica

- Abortando o programa em caso de erro de alocação:

```
//arquivo pilha.C - PILHA ESTÁTICA
void abortaPrograma(){
    printf("ERRO! Pilha nao foi alocada, ");
    printf("programa sera encerrado...\n\n\n");
    system("pause");
    exit(1);
}
```

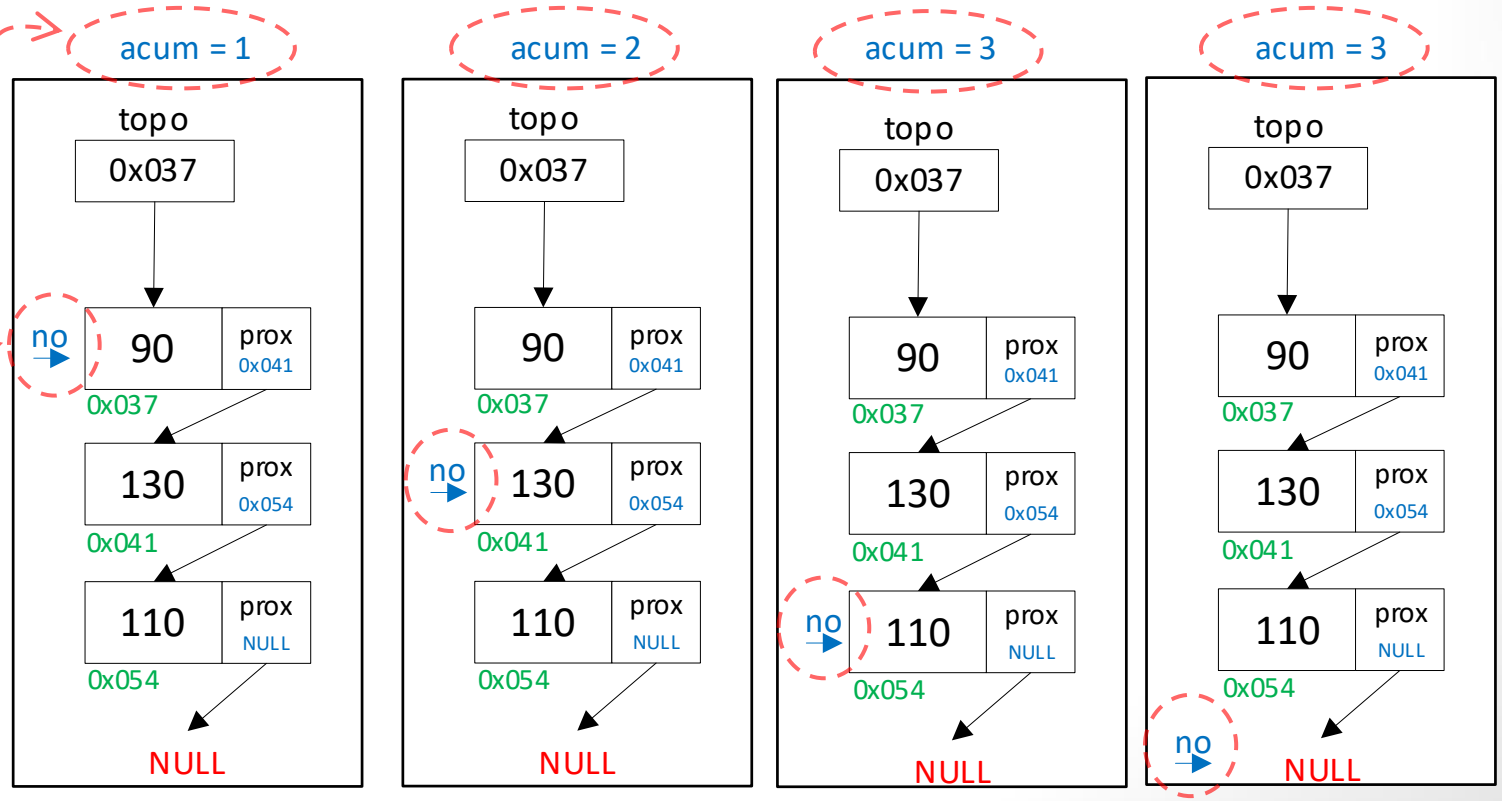
- Em todas as funções que manipulam e acessam os dados encapsulados, deve-se testar antes da manipulação dos dados, se a Pilha foi realmente alocada.

- Caso não tenha sido alocada por qualquer motivo, não teremos uma Pilha válida para trabalhar. O programa então deverá ser abortado, antes porém, informando ao usuário sobre a falha na alocação, que portanto, impede o correto funcionamento do programa.

Pilha sequencial dinâmica

- Informações básicas sobre a Pilha, retornando o tamanho:

```
//Arquivo pilhaDinamica.c
int tamanhoPilha(PILHA *pi){
    if(pi == NULL){
        abortaPrograma();
    }
    int acum = 0;
    ELEM *no = *pi;
    while(no != NULL){
        acum++;
        no = no->prox;
    }
    return acum;
}
```



Pilha sequencial dinâmica

- Informações básicas sobre a Pilha, retornando se cheia:

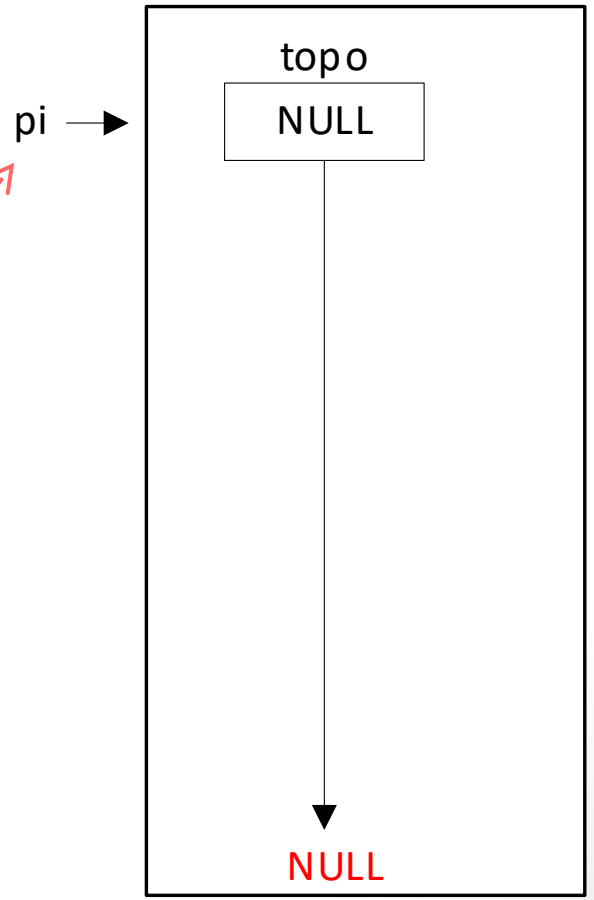
```
//Arquivo pilhaDinamica.c
int pilhaCheia(PILHA *pi){
    return 0;
}
```

- Em estruturas alocadas dinamicamente não faz sentido o conceito de “estruturas cheias”. Mantêm-se a função por uma questão de padronização e compatibilidade com as Pilhas estáticas.

Pilha sequencial dinâmica

- Informações básicas sobre a Pilha, retornando se vazia:

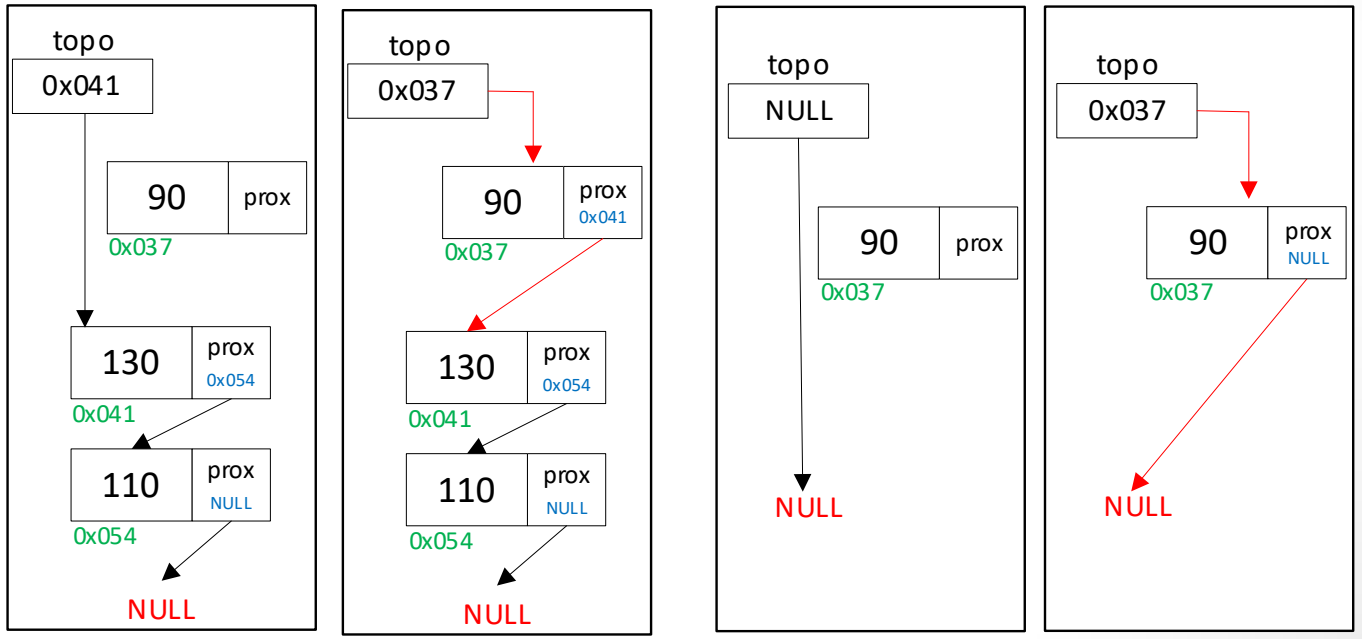
```
//Arquivo pilhaDinamica.c
int pilhaVazia(PILHA *pi){
    if(pi == NULL){//não alocada
        abortaPrograma();
    }
    if(*pi == NULL){//pilha existe, mas
        return 1;    //sem elementos
    }
    return 0;
}
```



Pilha sequencial dinâmica

- Em uma Pilha, a inserção se dá sempre em seu início (topo), e temos também o caso o caso onde a inserção é feita em uma Pilha que está vazia.

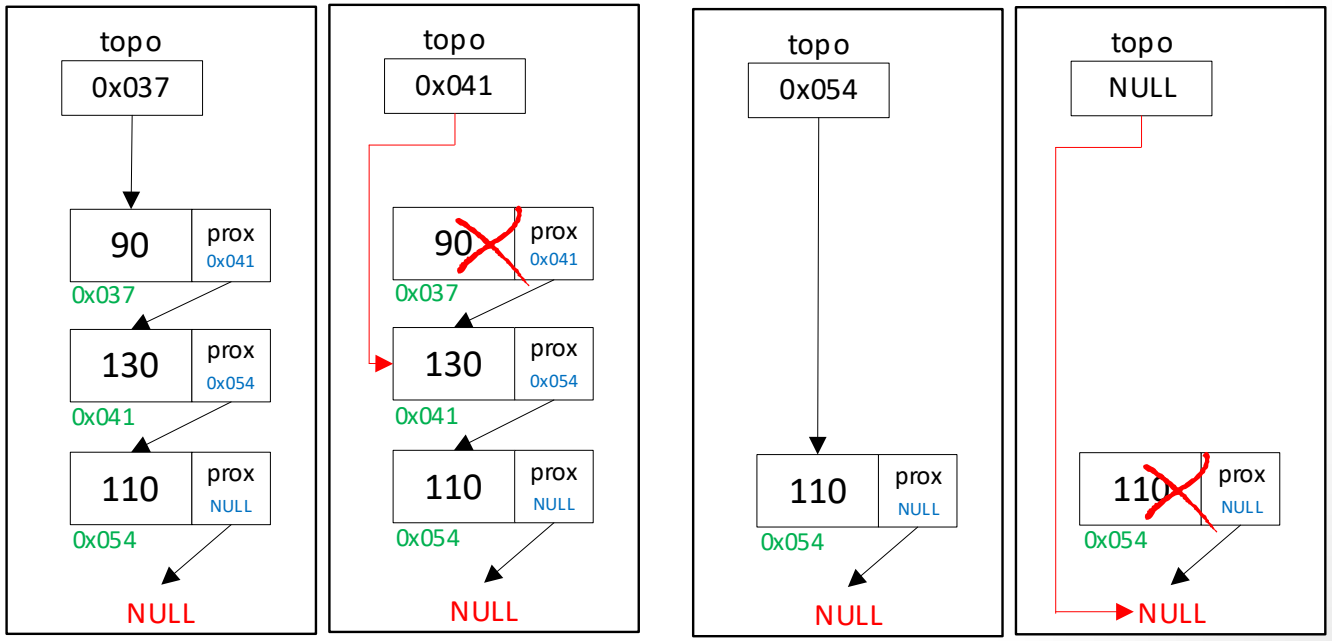
```
//Arquivo pilhaDinamica.c
int inserePilha(PILHA *pi, ALUNO al){
    if(pi == NULL){
        abortaPrograma();
    }
    ELEM *no = (ELEM*) malloc(sizeof(ELEM));
    if (no == NULL){
        return 0;
    }
    no->dados = al;
    no->prox = (*pi);
    *pi = no;
    return al.matricula;
}
```



Pilha sequencial dinâmica

- Em uma Pilha a remoção se dá sempre em seu início (topo), e temos que ter o cuidado de não se tentar remover de uma Pilha que está vazia.

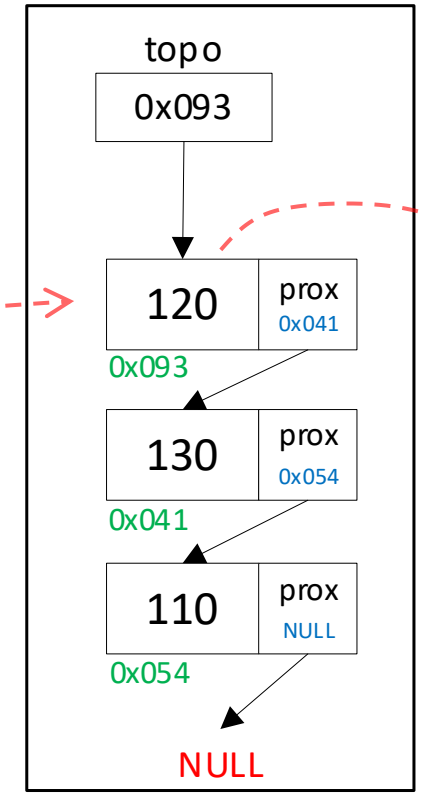
```
//Arquivo pilhaDinamica.c
int removePilha(PILHA *pi){
    int matricula;
    if(pi == NULL){//se pilha
        abortaPrograma();  //não alocada
    }
    if((*pi) == NULL){//se pilha vazia
        return 0;
    }
    ELEM *no = *pi;
    *pi = no->prox;
    matricula = (*no).dados.matricula;
    free(no);
    return matricula;
}
```



Pilha sequencial dinâmica

- A consulta em uma Pilha dinâmica também se dá apenas em seu topo, ou início, pois é o único elemento em que é permitido o acesso:

```
//Arquivo pilhaDinamica.c
int acessaTopoPilha(PILHA *pi, ALUNO *al){
    if(pi == NULL){
        abortaPrograma();
    }
    if((*pi) == NULL){
        return 0;
    }
    *al = (*pi)->dados;
    return 1;
}
```




```
Consulta realizada com sucesso:
Matricula: 120
Nota 1:    6.30
Nota 2:    6.40
Nota 3:    6.50
```

Atividade 2

- Atividade 2 – Pilha dinâmica:

- Monte o programa Pilha dinâmica, posicionando as chamadas das funções no `main()`, para manipulação da Pilha, de forma que apresentem o funcionamento e as mensagens iguais ao exemplo ao lado. Entregue no Moodle todos os arquivos zipados.