Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 03 – Pilha (Implementação)

Agenda

- Possíveis implementações
- Pilha com vetor
- Pilha com alocação dinâmica

Possíveis Implementações

- Existem duas principais formas de implementarmos uma Pilha:
  - Usando um vetor para armazenar os valores
  - Usando alocação dinâmica de memória para armazenar os valores

- A seguir, veremos um pouco mais das duas implementações, incluindo os seus prós e contras.

Possíveis Implementações

- Importante salientar que a Pilha é um **tipo abstrato de dado**. Isso quer dizer que existe mais de uma implementação possível para a Pilha.

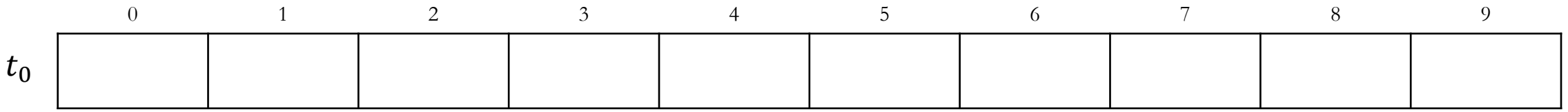
- Os conteúdos que serão abordados na aula de hoje mostram **uma possível implementação** de Pilhas, seja utilizando vetores ou seja utilizando alocação dinâmica de memória.

Pilha com Vetores

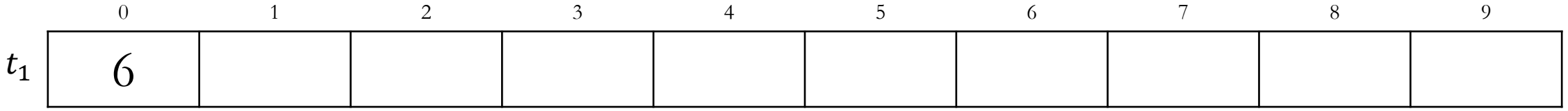
- A implementação da pilha com vetores utiliza-se da estrutura já existente das linguagens de programação (vetor/array) para armazenar seus valores.

- A implementação é simples: Basta usar um vetor (do tipo que for a pilha) e criar uma nova variável chamada **topo** que será a responsável por delimitar quem é o topo da pilha.

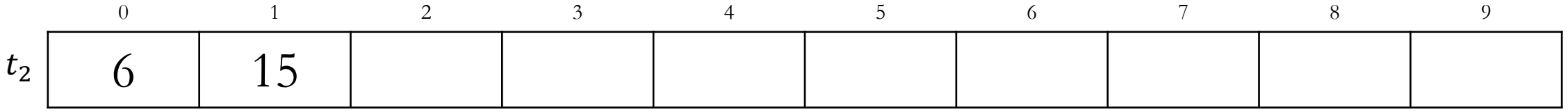
Pilhas com Vetores



Topo: -1
Ação: Empilhar o item "6"



Topo: 0
Ação: Empilhar o item "15"



Topo: 1

Pilhas com Vetores

t_3

0	1	2	3	4	5	6	7	8	9
6	15								

Topo: 1
Ação: Retornar Topo (Somente retorna o "15" sem mexer na Pilha)

t_4

0	1	2	3	4	5	6	7	8	9
6	15								

Topo: 1
Ação: Desempilhar (Retorna o "15")

t_5

0	1	2	3	4	5	6	7	8	9
6									

Topo: 0

Pilhas com Vetores

```c
#define TAM 4

int main() {
    char pilha[TAM];
    int topo = -1;
```

- Definição da Pilha
  - Defini um tamanho máximo para a pilha (neste exemplo, o tamanho é 4)
  - Criei uma Pilha de *char* (poderia ser de *int* ou qualquer outro tipo)
  - Iniciei o topo com -1 (indicando que não existem elementos na Pilha)

Pilhas com Vetores

```c
void empilhar(char *pilha, int *topo, char valor) { // push
    if (*topo >= TAM-1) {
        printf("Pilha cheia!\n");
    } else {
        *topo = *topo + 1;
        pilha[*topo] = valor;
    }
}
```

- Empilhar
  - Para empilhar, precisamos receber a pilha, topo e o valor a ser empilhado.
  - Inicialmente, verifico se a pilha está cheia. Se estiver, não consigo empilhar
  - Caso a pilha não esteja cheia, incremento o topo e adiciono a pilha no novo topo.

Pilhas com Vetores

```c
char desempilha(char *pilha, int *topo) { // pop
    if (*topo == -1) {
        printf("Pilha vazia! Impossível remover.\n");
        return '\0';
    }
    *topo = *topo - 1;
    return pilha[*topo + 1];
}
```

- Desempilhar
  - Para desempilhar, precisamos receber a pilha e o topo.
  - Inicialmente, verifico se a pilha está vazia. Se estiver, não há o que desempilhar.
  - Caso a pilha não esteja vazia, decremento o topo e retorno o item no antigo topo.

Pilhas com Vetores

- **Vantagens:**
  i. Acesso rápido aos elementos
     - Como um vetor é uma estrutura de dados contígua na memória, o acesso aos elementos é muito rápido, com complexidade **O(1)**, tanto para inserção quanto remoção (no topo da pilha).
  ii. Simplicidade de Implementação
     - A pilha pode ser facilmente implementada usando um vetor, o que torna o código simples e direto.

Pilhas com Vetores

- **Desvantagens:**
  i. Tamanho fixo (em vetores estáticos)
     - Se o vetor for alocado com um tamanho fixo, ele pode levar a um desperdício de memória se o vetor não for completamente utilizado ou a necessidade de realocar o vetor quando ele se lota, o que pode ser custoso (complexidade **O(n)** na realocação e cópia).

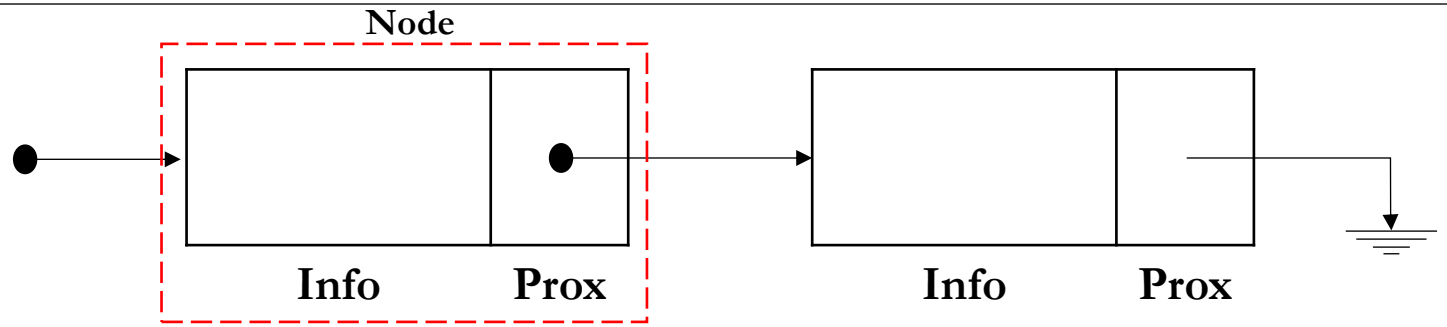
  ii. Realocação Cara
     - Em alguns casos, quando a pilha atinge o tamanho máximo do vetor, é necessário realocar o vetor para um novo espaço de memória maior. A realocação pode ser lenta e levar a uma cópia dos dados existentes, o que tem um custo de tempo **O(n)**.

Pilhas com Alocação Dinâmica

- Como vimos, uma das desvantagens da pilha com vetores é o tamanho fixo. A solução para esta desvantagem veio através da alocação dinâmica de memória.

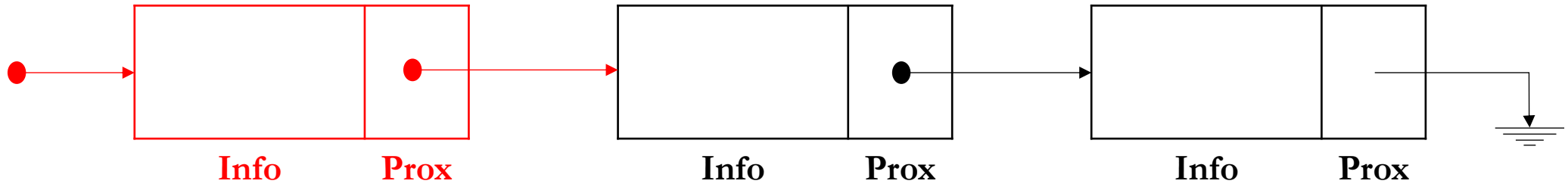
- Para isso, é preciso definirmos uma nova estrutura que será utilizada para armazenar o conteúdo de nossa Pilha.

Pilhas com Alocação Dinâmica



- Definimos uma estrutura chamada Node, que contém dois campos:
  - **Info**, que armazena o conteúdo de cada item da nossa pilha.
  - **Prox**, que armazena o endereço de memória do próximo item da nossa pilha.

Pilhas com Alocação Dinâmica



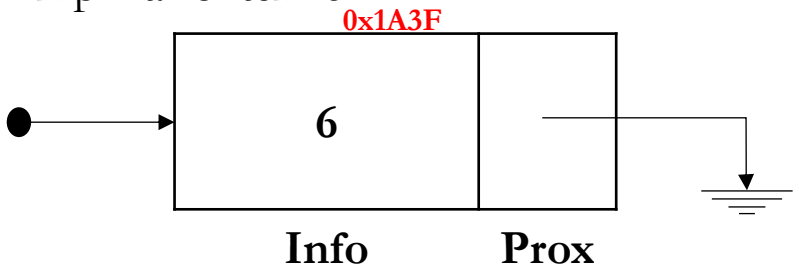
- Utilizando esta estrutura, a cada novo item a ser inserido (empilhado), basta criar outro Node e realizar as atribuições para que este node seja o novo topo da Pilha.

Pilhas com Alocação Dinâmica

t_0 **NULL**

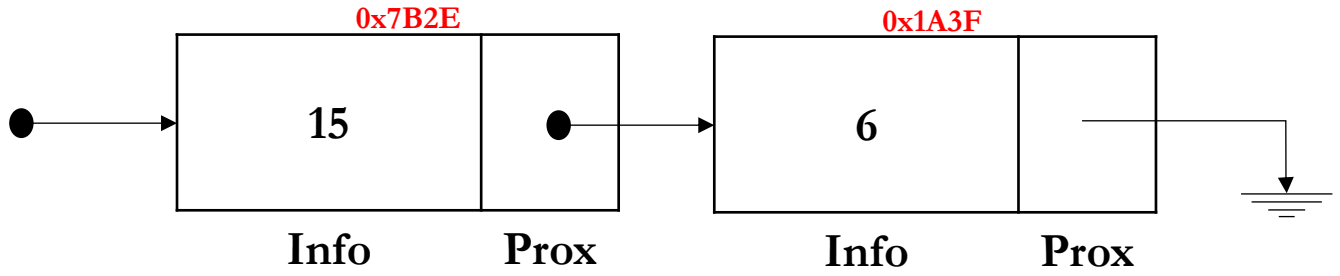
Topo: NULL
Ação: Empilhar o item 6

t_1



Topo: 0x1A3F
Ação: Empilhar o item 15

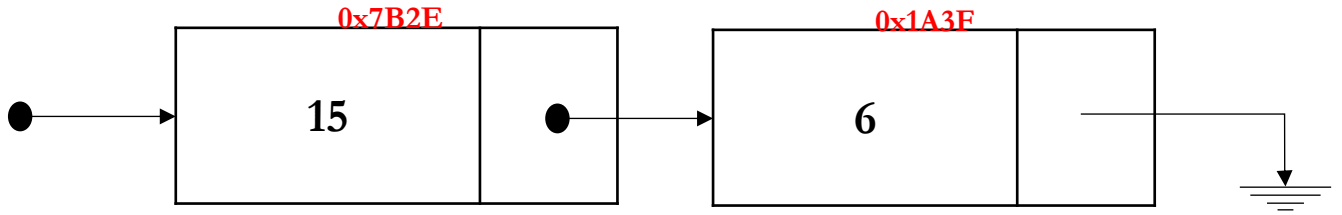
t_2



Topo: 0x7B2E

Pilhas com Alocação Dinâmica

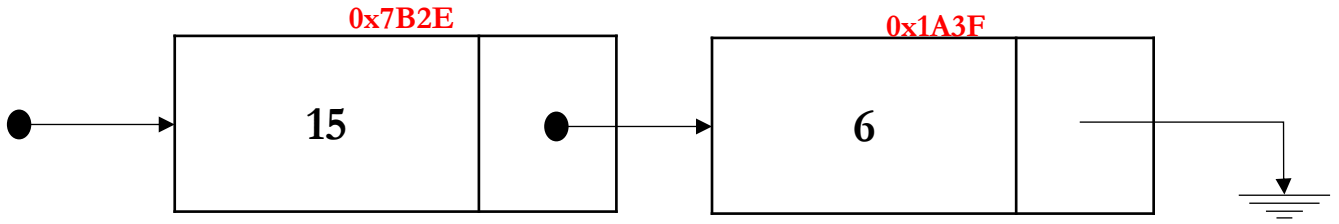
t_3



Topo: 0x7B2E

Ação: Retornar Topo (Somente retorna o Node 0x7B2E sem mexer na Pilha)

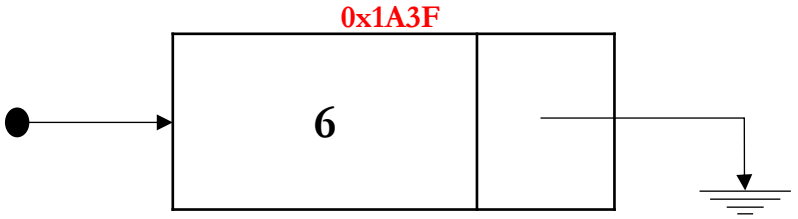
t_4



Topo: 0x7B2E

Ação: Desempilhar (Retorna 0x7B2E)

t_5



Topo: 0x1A3F

Pilhas com Alocação Dinâmica

```c
typedef struct no {
    int info;
    struct no *prox;
} Node;

typedef struct {
    struct no *topo;
} Pilha;
```

- Estrutura
  - Para criar a estrutura, precisamos criar dois tipos novos: Node e Pilha.
  - O Node possui dois atributos: info (do tipo dos elementos da pilha) e o prox (do tipo no).
  - A pilha possui um no inicial, chamado topo.

Pilhas com Alocação Dinâmica

```c
void empilha(Pilha *pilha, int info) { // push
    Node *newElement = (Node*) malloc(sizeof(Node));
    if (!newElement) {
        printf("ERRO: Não foi possível alocar memória! \n");
    } else {
        newElement->info = info;
        newElement->prox = pilha->topo;
        pilha->topo = newElement;
    }
}
```

- Empilhar
  - Para empilhar, inicialmente criamos um novo elemento usando alocação dinâmica de memória. Caso o elemento criado seja nulo, quer dizer que não temos espaço suficiente na memória para criar outro nó e, portanto, não é possível empilhar.
  - Caso haja espaço, adicionamos o item no novo nó e fazemos com que ele passe a apontar para o antigo topo da pilha, sendo assim, o próximo topo.

Pilhas com Alocação Dinâmica

- Desempilhar

```c
int desempilha(Pilha *pilha) { // pop
    if (pilha->topo == NULL) {
        printf("Pilha Vazia! Impossível desempilhar.\n");
        return -1;
    }
    Node *aux = pilha->topo;
    pilha->topo = aux->prox;
    int result = aux->info;
    free(aux);
    return result;
}
```

  - Para desempilhar, verificamos se o topo é nulo (se for, a nossa pilha está vazia e, assim, não há nada para desempilhar).
  - Posteriormente, criamos um novo Node que recebe o antigo topo, fazemos com que o próximo elemento seja o topo, liberamos o espaço da memória do elemento desempilhado e, por fim, retornamos o conteúdo do info.

Pilhas com Alocação Dinâmica

- **Vantagens:**
  i. Flexibilidade de tamanho
     - A pilha pode crescer e encolher conforme necessário, sem se preocupar com a capacidade inicial ou a realocação constante. Isso elimina o risco de ter um tamanho fixo e proporciona um uso mais eficiente da memória. Cada novo elemento é alocado de forma dinâmica conforme a necessidade.

  ii. Eficiência de uso de memória
     - Ao usar alocação dinâmica de memória (por exemplo, com listas encadeadas), a memória é alocada exatamente para o número de elementos armazenados, sem desperdício. Não é necessário reservar espaço extra como em vetores dinâmicos ou arrays de tamanho fixo.

Pilhas com Alocação Dinâmica

- **Vantagens** (cont.):
  iii. Sem necessidade de realocação ou cópia
    - Em uma lista encadeada, por exemplo, não há necessidade de realocar e copiar dados para um novo local de memória quando o número de elementos cresce, já que cada elemento é alocado separadamente. Isso evita os custos de realocação encontrados com vetores.

  iv. Desempenho estável
    - Em uma pilha com alocação dinâmica, o tempo de inserção (push) e remoção (pop) de elementos é constante **O(1)**, já que cada operação ocorre diretamente no topo da pilha sem afetar os outros elementos.

Pilhas com Alocação Dinâmica

- **Desvantagens:**
  i. <u>Sobrecarga de memória adicional</u>
     - Cada elemento em uma lista encadeada ou estrutura dinâmica requer alocação de memória para o próprio dado e para um ponteiro (ou referência) que aponta para o próximo elemento. Isso implica um desperdício de memória adicional para armazenar os ponteiros, o que pode ser ineficiente se o dado armazenado for pequeno em comparação com o custo do ponteiro.

  ii. <u>Fragmentação de memória</u>
     - A memória pode ser fragmentada ao longo do tempo, o que pode tornar o uso ineficiente. Isso é especialmente importante em sistemas com recursos limitados, como sistemas embarcados, onde a fragmentação pode levar a um uso não ideal da memória.

Pilhas com Alocação Dinâmica

- **Desvantagens** (cont.):
  - iii. Gerenciamento de memória mais complexo
    - É necessário garantir que a memória seja liberada corretamente após o uso para evitar vazamentos de memória. A alocação e desalocação dinâmica de memória podem ser mais lentas do que simplesmente manipular um vetor estático.
  - iv. Desempenho variável
    - A alocação/desalocação de memória podem ter custos de desempenho que variam dependendo do sistema e do número alocação realizadas. Sistemas com muitos elementos sendo inseridos e removidos pode resultar em penalidades de desempenho.

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 03 – Pilha (Implementação)