Algoritmos e Estrutura de Dados I - Listas de Exercícios

Jorge Augusto Salgado Salhani

Agosto, 2022

1 Lista 2

1.1 O que é e para que serve uma pilha?

Podemos definir uma pilha em dois níveis. No nível lógico, consiste em um tipo abstrato de dado (TAD) capaz de armazenar itens quaisquer, cujo acesso prioriza os itens mais recentes. Dessa forma, para resgatar e/ou adicionar novos elementos, o acesso é feito apenas no topo da pilha (LIFO: *Last in, First Out*).

Já no nível de operações, podemos definir os seguintes conjuntos de especificações:

Estrutura

- **Protocolo**

  Elementos adicionados e removidos do topo (LIFO)

- **Definição**

  MAX_ITENS: Número máximo de itens da pilha;

  TipoItem: Tipo de dado de cada elemento armazenado na pilha;

Operações

- **pilha_criar**

  Função: Criar uma estrutura de pilha

  Pré-condições: Nenhuma

  Pós-condições: Pilha está vazia

- **pilha_apagar**

  Função: Liberar memória alocada para a estrutura e seus elementos remanescentes

  Pré-condições: Pilha existe

  Pós-condições: Pilha e itens liberados da memória

- **pilha_vazia**

  Função: Verificar se a pilha está vazia

  Pré-condições: Pilha existe

  Pós-condições: Número de elementos da pilha é maior que zero (sim/não)

- **pilha_cheia**

  Função: Verificar se a pilha está cheia

  Pré-condições: Pilha existe

  Pós-condições: Número de elementos da pilha é igual ao número máximo de elementos pré-definidos (sim/não)

- **pilha_topo**

  Função: Retorna uma cópia do elemento do topo da pilha, sem removê-lo

  Pré-condições: Pilha existe e não está vazia

  Pós-condições: Pilha inalterada

- **pilha_empilhar**

  Função: Insere um novo elemento à pilha

  Pré-condições: Pilha existe e não está cheia

  Pós-condições: Pilha com um novo item adicionado

- **pilha_desempilhar**

  Função: Remove um elemento da pilha, retornando-o ao usuário

  Pré-condições: Pilha existe e não está vazia

  Pós-condições: Pilha com um elemento a menos

Pilhas são úteis em operações de desfazer alterações recentes (*undo* em editores de texto, e.g.) ou em operações encadeadas ou recursivas, onde o retorno da última chamada é utilizado na anterior (processo de desempilhar após todas as chamadas). Retornos de erros seguem esse padrão recursivo e levam o nome de *stacktrace*, pela utilização de estruturas em pilha.

1.2 O que significa alocação sequencial de memória para um conjunto de elementos?

Alocação sequencial de memória refere-se à forma de armazenamento de um conjunto de elementos em blocos contínuos de memória. Por exemplo, caso seja solicitado ao sistema armazenar um vetor (conjunto) de 10 elementos do tipo *int*, mesmo que a memória seja alocada na *stack* ou na *heap*, um conjunto de 10 posições de memória contíguas são separadas, podendo armazenar valores de tamanho *sizeof(int)* (em geral = 8 bytes).

1.3 O que ignifica alocação estática de memória para um conjunto de elementos?

Alocação estática de memória está relacionado ao modo como um conjunto de elementos será alocado na memória, análogo ao item anterior. A memória estática costuma ser realizada em estruturas de pilha (*stack*), em contraste com a memória dinâmica, realizada em estruturas de montes (*heap*).

Dentre as principais características da memória estática temos a separação da memória em tempo de compilação e o fato de que, uma vez alocada, um "bloco" da memória não poderá ser reutilizado por outro tipo de dado definido no momento da compilação.

1.4 Faça o esquema de uma implementação sequencial e estática de uma pilha e descreva seu funcionamento.

1.5 Desenvolva uma rotina para inverter a posição dos elementos de uma pilha P.

```
// Arquivo pilha.h
#ifndef PILHA_H
    #define PILHA_H
    #include "item.h"
    typedef struct pilha_ PILHA;

    // Funcoes padrao de interface TAD pilha utilizadas//
    bool pilha_vazia(PILHA* pilha);
    bool pilha_cheia(PILHA* pilha);
    bool pilha_empilhar(PILHA* pilha, ITEM* item);
    ITEM* pilha_desempilhar(PILHA* pilha);
    // =======
    void pilha_inverter(PILHA* pilha);
#endif
```

```c
// Arquivo pilha.c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
void pilha_inverter_suporte(PILHA* pilha, PILHA* pilha_invertida) {
    if (pilha == NULL || pilha_invertida == NULL) return;
    if (pilha_vazia(pilha) && !pilha_cheia(pilha_invertida)) return;
    pilha_empilhar(pilha_invertida, pilha_desempilhar(pilha));
    pilha_inverter_suporte(pilha, pilha_invertida);
}

void pilha_inverter(PILHA* pilha) {
    if (pilha == NULL) return;
    PILHA* pilha_invertida = (PILHA*) malloc(sizeof(PILHA));
    pilha_inverter_suporte(pilha, pilha_invertida);
    return;
}
```

1.6 Desenvolva uma função para testar se uma pilha P1 tem mais elementos que uma pilha P2.

```c
\\ Arquivo pilha.h
#ifndef PILHA_H
    #define PILHA_H
    #include <stdbool.h>
    typedef struct pilha_ PILHA;

    // Funcoes padrao TAD pilha utilizadas
    int pilha_tamanho(PILHA* pilha);
    // ======
    bool pilha_comparar_tamanho(PILHA* P1, PILHA* P2);
#endif

// Arquivo pilha.c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool pilha_comparar_tamanho(PILHA* P1, PILHA* P2) {
    if (P1 == NULL || P2 == NULL) return false;
```

```
    return pilha_tamanho(P1) >= pilha_tamanho(P2) ? true : false;
}
```

1.7 Desenvolva uma função para testar se duas pilhas P1 e P2 são iguais.

```
// Arquivo item.h
#ifndef ITEM_H
    #define ITEM_H
    typedef struct item_ ITEM;

    // Funcoes padrao TAD item utilizadas
    int item_get_chave(ITEM* item);
    // =====
#endif

// Arquivo pilha.h
#ifndef PILHA_H
    #define PILHA_H
    #include <stdbool.h>
    #include "item.h"

    // Funcoes padrao TAD pilha utilizadas
    ITEM* pilha_desempilhar(PILHA* pilha);
    bool pilha_empilhar(PILHA* pilha, ITEM* item);
    bool pilha_vazia(PILHA* pilha);
    bool pilha_cheia(PILHA* pilha);
    bool pilha_apagar(PILHA* pilha);
    int pilha_tamanho(PILHA* pilha);
    // ======

    bool pilha_comparar(PILHA* P1, PILHA* P2);
#endif

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

void pilha_reempilhar(PILHA* P, PILHA* p_aux) {
    if (pilha_vazia(p_aux) || pilha_cheia(P)) return;
```

```
    pilha_empilhar(P, pilha_desempilhar(p_aux));
    pilha_reempilhar(P, p_aux);
}

bool pilha_desempilhar_e_comparar(PILHA* P1, PILHA* P2, PILHA* p1_aux, PILHA*
    p2_aux) {
    if (p1_aux == NULL || p2_aux == NULL) return false;
    int chave_p1, chave_p2;
    ITEM* item1;
    ITEM* item2;
    if (!pilha_vazia(P1) && !pilhA_vazia(P2)) {
        item1 = pilha_desempilhar(P1);
        item2 = pilha_desempilhar(P2);
        chave_p1 = item_get_chave(item1);
        chave_p2 = item_get_chave(item2);

        if (chave_p1 == chave_p2) {
            pilha_empilhar(p1_aux, item1);
            pilha_empilhar(p2_aux, item2);
        } else {
            pilha_reempilhar(P1, p1_aux);
            pilha_reempilhar(P2, p2_aux);
            return false;
        }
    } else {
        return true;
    }
    pilha_desempilhar_e_comparar(P1, P2, p1_aux, p2_aux);
}

bool pilha_comparar(PILHA* P1, PILHA* P2) {
    if (P1 == NULL || P2 == NULL) return false;
    PILHA* p1_aux = (PILHA*) malloc(sizeof(PILHA));
    PILHA* p2_aux = (PILHA*) malloc(sizeof(PILHA));
    if (pilha_tamanho(P1) != pilha_tamanho(P2)) return false;
    pilha_desempilhar_e_comparar(P1, P2, p1_aux, p2_aux);
}
```

1.8 O que é e como funciona uma estrutura do tipo Fila?

Assim como fizemos para a estrutura do tipo Pilha, podemos separar em dois tipos de definição: lógica e operacional. Para o primeiro, uma fila consiste em um tipo abstrato de dado (TAD) responsável por armazenar elementos priorizados conforme sua ordem de inserção, ou seja, prioriza os itens mais antigos. Temos portanto duas extremidades: inicial e final. Novos elementos são adicionados na extremidade inicial, e elementos são lidos/resgatados na extremidade final (FIFO: *First in, First out*).

Em nível de operações, temos as seguintes definições

Estrutura

- **Protocolo**

  Elementos adicionados à extremidade inicial. Elementos removidos da extremidade final (oposta).

- **Definição**

  MAX_ITENS: Número máximo de itens na fila;

  TipoItem: Tipo de dado de cada elemento armazenado na fila

Operações

- **fila_criar**

  Função: Criar uma estrutura de fila

  Pré-condição: Nenhuma

  Pós-condição: Fila existe e está vazia

- **fila_apagar**

  Função: Limpar memória alocada para a fila e para os elementos nela remanescentes

  Pré-condições: Fila existe

  Pós-condições: Fila e itens liberados, com conteúdo nulo

- **fila_frente**

  Função: Retornar cópia do primeiro item da fila

  Pré-condições: Fila existe e fila não está vazia

  Pós-condições: Fila inalterada, item retornado para usuário

- **fila_inserir**

  Função: Adiciona novo elemento ao fim da fila

  Pré-condições: Fila existe e fila não está cheia

  Pós-condições: Fila com um item a mais em sua extremidade final

- **fila_remover**

  Função: Remover e retornar o primeiro elemento da fila

  Pré-condições: Fila existe e fila não está vazia

  Pós-condições: Item retornado ao usuário e fila com um item a menos em sua extremidade inicial

- **fila_tamanho**

  Função: Retornar o número de elementos presentes na fila

  Pré-condições: Fila existe

  Pós-condições: Fila inalterada

- **fila_cheia**

  Função: Verificar se a fila está cheia

  Pré-condições: Fila existe

  Pós-condições: Fila inalterada, e resposta se o número de itens é igual ao número máximo de itens pré-definidos (sim/não)

- **fila_vazia**

  Função: Verificar se a fila está vazia

  Pré-condições: Fila existe

  Pós-condições: Fila inalterada e resposta se o número de itens na fila é maior que zero (sim/não)

1.9 Em que situações uma fila pode ser utilizada?

A estrutura de fila é interessante para contextos onde processos ou itens são executados ou resgatados em tempo inversamente proporcional ao tempo de espera desde o instante em que foi criado.

Por exemplo, quando múltiplos processos ou comandos simultâneos são requeridos ao sistema, deve existir um tempo de espera entre execuções (ou *buffers*). Estruturas de fila fornece um bom suporte a este cenário, pois

gerencia um histórico de solicitações em memória (*buffers*) e executa-os na ordem original da solicitação, dando o devido tempo entre processos. Sistemas de input/output padrão na digitação de texto utiliza de filas para que a ordem de teclas digitadas seja inserida corretamente no editor de texto.

1.10 Faça um esquema da implementação estática e sequencial de uma fila e explique resumidamente o seu funcionamento.

Código

1.11 Implemente um TAD fila, e faça um programa para teste.

Código

1.12 Desenvolva uma função para testar se uma fila F1 tem mais elementos do que uma fila F2.

Código

1.13 Implemente uma fila em um vetor circular, sem armazenar o número total de elementos (sugestão: nunca deixe que o indicador "fim" alcance o indicador "início", ainda que seja necessário perder uma posição do vetor).

Código

1.14 Implemente a funcionalidade de uma fila a partir de uma ou mais pilhas (sugestão: use 2 pilhas).

Código