Respostas Lista de Exercícios 02 sobre
Filas

Sistemas de Informação - UFG
Aluno: Marcos Vinicius Vieira da Silva
Matrícula: 147255

A maioria dos exercícios foram criados utilizando a IDE CLion e para cada um foi criado um projeto (pasta) com os arquivos .c que será enviada junto com esse PDF.

1 - **R:** pasta: exercício-1

2 - **R:** pasta: exercício-2

3 - **R:** pasta: exercício-3

4 - **R:** pasta: exercício-4

5 - **R:**
Para que uma sequência de operações Empilha e Desempilha em uma pilha inicialmente vazia termine com a pilha vazia, sem causar underflow, a condição necessária e suficiente é que o número de operações Desempilha seja igual ao número de operações Empilha.
E para que uma sequência de operações Empilha e Desempilha em uma pilha inicialmente não vazia a deixe inalterada, a condição necessária e suficiente é que para cada operação Desempilha, exista uma operação Empilha anterior com o mesmo elemento, e que a ordem das operações Desempilha seja a inversa da ordem das operações Empilha correspondentes.

```C/C++

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef int TipoItem;

typedef struct {
    TipoItem *itens;
    int topo;
    int max;
} Pilha;

Pilha* criaPilha(int max) {
    Pilha *p = (Pilha*) malloc(sizeof(Pilha));
    p->itens = (TipoItem*) malloc(max * sizeof(TipoItem));
    p->topo = -1;
    p->max = max;
```

```c
    return p;
}

bool pilhaVazia(Pilha *p) {
    return p->topo == -1;
}

bool pilhaCheia(Pilha *p) {
    return p->topo == p->max - 1;
}

bool empilha(Pilha *p, TipoItem item) {
    if (pilhaCheia(p)) {
        printf("Erro: pilha cheia!\n");
        return false;
    }
    p->itens[++p->topo] = item;
    return true;
}

bool desempilha(Pilha *p, TipoItem *item) {
    if (pilhaVazia(p)) {
        printf("Erro: pilha vazia!\n");
        return false;
    }
    *item = p->itens[p->topo--];
    return true;
}

void imprimePilha(Pilha *p) {
    if (pilhaVazia(p)) {
        printf("Pilha vazia.\n");
        return;
    }
    printf("Conteudo da pilha (base -> topo): ");
    for (int i = 0; i <= p->topo; i++) {
        printf("%d ", p->itens[i]);
    }
    printf("\n");
}

void liberaPilha(Pilha *p) {
    free(p->itens);
    free(p);
}


int main() {
    Pilha *p = criaPilha(10);
    TipoItem item;

    empilha(p, 1);
    empilha(p, 2);
    empilha(p, 3);
```

```
    imprimePilha(p);

    if (desempilha(p, &item)) {
        printf("Elemento desempilhado: %d\n", item);
    }
    if (desempilha(p, &item)) {
        printf("Elemento desempilhado: %d\n", item);
    }

    imprimePilha(p);

    if (!desempilha(p, &item)) {
        printf("Underflow detectado.\n");
    }

    liberaPilha(p);
    return 0;
}
```

6 - **R:**
A estrutura de dados mais adequada é a **pilha**. A pilha opera no princípio LIFO (último a entrar, primeiro a sair). Ao encontrarmos os fatores primos, podemos empilhá-los. Ao final do processo, desempilhamos os elementos para imprimi-los. Como o último elemento a entrar na pilha será o primeiro a sair, a ordem de impressão será a inversa da ordem em que os fatores foram encontrados, resultando na ordem decrescente desejada.

```C/C++

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

// Definição da estrutura da pilha
typedef struct {
    int *itens;
    int topo;
    int max;
} Pilha;

// Funções da pilha
Pilha* criaPilha(int max) {
    Pilha *p = (Pilha*) malloc(sizeof(Pilha));
    p->itens = (int*) malloc(max * sizeof(int));
    p->topo = -1;
    p->max = max;
    return p;
}
```

```c
int pilhaVazia(Pilha *p) {
    return p->topo == -1;
}

int pilhaCheia(Pilha *p) {
    return p->topo == p->max - 1;
}

int empilha(Pilha *p, int item) {
    if (pilhaCheia(p)) {
        return 0;  // Retorna erro se a pilha estiver cheia
    }
    p->topo++;
    p->itens[p->topo] = item;
    return 1;
}

int desempilha(Pilha *p, int *item) {
    if (pilhaVazia(p)) {
        return 0;  // Retorna erro se a pilha estiver vazia
    }
    *item = p->itens[p->topo];
    p->topo--;
    return 1;
}

void liberaPilha(Pilha *p) {
    free(p->itens);
    free(p);
}

// Função para calcular a fatoração prima
void fatoracaoPrima(int n) {
    Pilha *p = criaPilha(100); // Cria uma pilha com capacidade suficiente

    for (int d = 2; n > 1; d++) {
        while (n % d == 0) {
            if (!empilha(p, d)) {
                printf("Erro: pilha cheia!\n");
                liberaPilha(p);
                return;
            }
            n /= d;
        }
    }

    // Imprime os fatores primos em ordem decrescente
    printf("Fatores primos em ordem decrescente: ");
    int fator;
    while (desempilha(p, &fator)) {
        printf("%d ", fator);
        if (!pilhaVazia(p)) {
            printf("* ");
        }
```

```
    }
    printf("\n");

    liberaPilha(p);
}

int main() {
    int n;

    printf("Digite um numero inteiro maior que 1: ");
    scanf("%d", &n);

    if (n <= 1) {
        printf("Numero invalido. Deve ser maior que 1.\n");
        return 1;
    }

    fatoracaoPrima(n);
    return 0;
}
```

7 - R:

C/C++

```
// Função para remover um item com chave c de uma posição qualquer da pilha
void removeItem(Pilha *p, int c) {
    Pilha *aux = criaPilha(p->max);
    int item;
    int encontrado = 0;

    while (!pilhaVazia(p)) {
        desempilha(p, &item);
        if (item == c && !encontrado) {
            encontrado = 1;  // Marca o item como encontrado
        } else {
            empilha(aux, item);
        }
    }

    // Restaura os itens para a pilha original
    while (!pilhaVazia(aux)) {
        desempilha(aux, &item);
        empilha(p, item);
    }

    liberaPilha(aux);

    if (!encontrado) {
        printf("Item com chave %d nao encontrado na pilha.\n", c);
    }
}
```

8 - **R:**

C/C++

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

#define MAX 100

typedef int TipoItem;

typedef struct {
    TipoItem itens[MAX];
    int topo1;
    int topo2;
} PilhaDupla;

void inicializaPilhaDupla(PilhaDupla *p) {
    p->topo1 = -1;
    p->topo2 = MAX;
}

int pilhaDuplaVazia(PilhaDupla *p, int pilha) {
    if (pilha == 1)
        return p->topo1 == -1;
    else
        return p->topo2 == MAX;
}

int pilhaDuplaCheia(PilhaDupla *p) {
    return p->topo1 + 1 == p->topo2;
}

void empilha(PilhaDupla *p, TipoItem item, int pilha) {
    if (pilhaDuplaCheia(p)) {
        printf("Erro: pilha cheia!\n");
        exit(1);
    }
    if (pilha == 1)
        p->itens[++p->topo1] = item;
    else
        p->itens[--p->topo2] = item;
}

TipoItem desempilha(PilhaDupla *p, int pilha) {
    if (pilhaDuplaVazia(p, pilha)) {
        printf("Erro: pilha vazia!\n");
        exit(1);
    }
    if (pilha == 1)
        return p->itens[p->topo1--];
    else
        return p->itens[p->topo2++];
}
```

```c
int main() {
    PilhaDupla p;
    inicializaPilhaDupla(&p);

    empilha(&p, 1, 1);
    empilha(&p, 2, 1);
    empilha(&p, 3, 2);
    empilha(&p, 4, 2);

    printf("Desempilhando da pilha 1: %d\n", desempilha(&p, 1));
    printf("Desempilhando da pilha 2: %d\n", desempilha(&p, 2));

    return 0;
}
```

9 - **R:**

C/C++

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

#define MAX 10 // Capacidade máxima do estacionamento

typedef struct {
    char placa[8];
} Carro;

typedef struct {
    Carro carros[MAX];
    int topo;
} Estacionamento;

// Inicializa o estacionamento
void inicializaEstacionamento(Estacionamento *e) {
    e->topo = -1;
}

// Verifica se o estacionamento está cheio
int estacionamentoCheio(Estacionamento *e) {
    return e->topo == MAX - 1;
}

// Verifica se o estacionamento está vazio
int estacionamentoVazio(Estacionamento *e) {
    return e->topo == -1;
}

// Adiciona um carro ao estacionamento
int entraCarro(Estacionamento *e, char placa[]) {
```

```c
    if (estacionamentoCheio(e)) {
        printf("Estacionamento cheio. Carro %s não pode entrar.\n", placa);
        return 0;
    }
    e->topo++;
    strcpy(e->carros[e->topo].placa, placa);
    printf("Carro %s entrou no estacionamento.\n", placa);
    return 1;
}

// Remove um carro do estacionamento
int saiCarro(Estacionamento *e, char placa[]) {
    if (estacionamentoVazio(e)) {
        printf("Estacionamento vazio. Carro %s não está presente.\n", placa);
        return 0;
    }

    Estacionamento temporario;
    inicializaEstacionamento(&temporario);
    int manobras = 0;
    int encontrado = 0;

    // Remove carros até encontrar o carro desejado
    while (!estacionamentoVazio(e)) {
        Carro carroAtual = e->carros[e->topo];
        e->topo--;

        if (strcmp(carroAtual.placa, placa) == 0) {
            encontrado = 1;
            break;
        } else {
            entraCarro(&temporario, carroAtual.placa);
            manobras++;
        }
    }

    // Reinsere os carros de volta no estacionamento
    while (!estacionamentoVazio(&temporario)) {
        Carro carroAtual = temporario.carros[temporario.topo];
        temporario.topo--;
        entraCarro(e, carroAtual.placa);
    }

    if (encontrado) {
        printf("Carro %s saiu do estacionamento. Foram feitas %d manobras.\n", placa,
manobras);
        return 1;
    } else {
        printf("Carro %s não encontrado no estacionamento.\n", placa);
        return 0;
    }
}

int main() {
```

```
    Estacionamento estacionamento;
    inicializaEstacionamento(&estacionamento);

    char operacao;
    char placa[8];

    printf("Digite as operações (E/S para entrada/saida e placa do carro, ou F para
finalizar):\n");

    while (1) {
        scanf(" %c", &operacao);

        if (operacao == 'F') {
            break;
        }

        scanf("%s", placa);

        if (operacao == 'E') {
            entraCarro(&estacionamento, placa);
        } else if (operacao == 'S') {
            saiCarro(&estacionamento, placa);
        } else {
            printf("Operação inválida. Use 'E' para entrada, 'S' para saída ou 'F'
para finalizar.\n");
        }
    }

    printf("Programa finalizado.\n");
    return 0;
}
```