ED2 - Exercícios.pdf - Rafael Manfrim

Implemente em C/C++ as operações de inserção e remoção em uma lista singularmente ligada.

```cpp
#include <iostream>

using namespace std;

struct No {
    int valor;
    No* prox;
};

void inserir_inicio(No** cabeca, int novo_valor) {
    No* novo_no = new No;
    novo_no->valor = novo_valor;
    novo_no->prox = *cabeca;
    *cabeca = novo_no;
}

void inserir_fim(No** cabeca, int novo_valor) {
    No* novo_no = new No;
    novo_no->valor = novo_valor;
    novo_no->prox = nullptr;

    if (*cabeca == nullptr) {
        *cabeca = novo_no;
        return;
    }
    No* no_selecionado = *cabeca;
    while(no_selecionado->prox != nullptr) {
        no_selecionado = no_selecionado->prox;
    }
    no_selecionado->prox = novo_no;
}

void inserir_posicao(No** cabeca, int novo_valor, int posicao) {
    // Assume-se que as posições começarão à partir de 1
    if (posicao <= 0) {
        return;
    }
    if (posicao == 1) {
        inserir_inicio(cabeca, novo_valor);
        return;
    }
    No* no_selecionado = *cabeca;
    int i = 1;
    while (i < posicao - 1) {
        if(no_selecionado->prox == nullptr) {
            cout << "Posição inválida!" << endl;
            return;
        }
        no_selecionado = no_selecionado->prox;
        i++;
    }
    No* novo_no = new No;
    novo_no->valor = novo_valor;
```

```
    novo_no->prox = no_selecionado->prox;
    no_selecionado->prox = novo_no;
}

int remover_inicio(No** cabeca) {
    if (*cabeca == nullptr) {
        cout << "A lista está vazia, não há nada para remover." << endl;
        exit(1);
    }
    No* no_a_remover = *cabeca;
    *cabeca = (*cabeca)->prox;
    int valor_removido = no_a_remover->valor;
    delete no_a_remover;
    return valor_removido;
}

int remover_fim(No** cabeca) {
    if (*cabeca == nullptr) {
        cout << "A lista está vazia, não há nada para remover." << endl;
        exit(1);
    }
    No* no_selecionado = *cabeca;

    if (no_selecionado->prox == nullptr) {
        int valor_removido = no_selecionado->valor;
        delete no_selecionado;
        *cabeca = nullptr;
        return valor_removido;
    }
    while(no_selecionado->prox->prox != nullptr) {
        no_selecionado = no_selecionado->prox;
    }
    int valor_removido = no_selecionado->prox->valor;
    delete no_selecionado->prox;
    no_selecionado->prox = nullptr;
    return valor_removido;
}

int remover_posicao(No** cabeca, int posicao) {
    // Assume-se que as posições começarão à partir de 1
    if (posicao <= 0) {
        cout << "Posição Inválida!";
        exit(1);
    }
    if (posicao == 1) {
        return remover_inicio(cabeca);
    }
    No* no_selecionado = *cabeca;
    int i = 1;
    while (i < posicao - 1) {
        if(no_selecionado->prox == nullptr) {
            cout << "Posição inválida!" << endl;
            exit(1);
        }
        no_selecionado = no_selecionado->prox;
        i++;
    }
    No* no_a_remover = no_selecionado->prox;
    no_selecionado->prox = no_a_remover->prox;
    int valor_removido = no_a_remover->valor;
    delete no_a_remover;
    return valor_removido;
```

```
}

void imprime_lista(No* no) {
    cout << "Lista: ";

    while (no != nullptr) {
        cout << no->valor << " -> ";
        no = no->prox;
    }
    cout << "nullptr" << endl;
}

void excluir_lista(No** cabeca) {
    No* no_selecionado = *cabeca;
    No* prox = nullptr;

    while (no_selecionado != nullptr) {
        prox = no_selecionado->prox;
        delete no_selecionado;
        no_selecionado = prox;
    }
    *cabeca = nullptr;
}

int main() {
    No* cabeca = nullptr;

    cout << "Inserções no início: " << endl;

    inserir_inicio(&cabeca, 1);
    imprime_lista(cabeca);

    inserir_inicio(&cabeca, 3);
    inserir_inicio(&cabeca, 5);
    imprime_lista(cabeca);

    inserir_inicio(&cabeca, 4);
    imprime_lista(cabeca);

    cout << "Inserções no fim: " << endl;

    inserir_fim(&cabeca, 9);
    imprime_lista(cabeca);

    inserir_fim(&cabeca, 7);
    imprime_lista(cabeca);

    cout << "Inserções no meio: " << endl;

    inserir_posicao(&cabeca, 8, 2);
    imprime_lista(cabeca);

    inserir_posicao(&cabeca, 2, 7);
    imprime_lista(cabeca);

    inserir_posicao(&cabeca, 0, 1);
    imprime_lista(cabeca);

    cout << "Tentando inserir em uma posição não existente: " << endl;

    inserir_posicao(&cabeca, 15, 18);
```

```
    cout << "Excluindo do começo: " << endl;

    remover_inicio(&cabeca);
    imprime_lista(cabeca);

    remover_inicio(&cabeca);
    imprime_lista(cabeca);

    cout << "Excluindo do fim: " << endl;

    remover_fim(&cabeca);
    imprime_lista(cabeca);

    remover_fim(&cabeca);
    imprime_lista(cabeca);

    cout << "Excluindo no meio: " << endl;

    remover_posicao(&cabeca, 4);
    imprime_lista(cabeca);

    remover_posicao(&cabeca, 1);
    imprime_lista(cabeca);

    cout << "Tentando remover uma posição não existente: " << endl;

    remover_posicao(&cabeca, 10);

    excluir_lista(&cabeca);
    return 0;
}
```

Implemente em C/C++ uma fila utilizando duas pilhas. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```
#include <iostream>
#define MAX_ITEMS 5

using namespace std;

struct Stack {
    int top;
    int items[MAX_ITEMS];
};

struct Queue {
    Stack stack1;
    Stack stack2;
};

bool stack_empty(Stack &stack) {
    return stack.top == -1;
}

bool stack_full(Stack &stack) {
    return stack.top == MAX_ITEMS - 1;
}

void stack_push(Stack &stack, int value) {
    if (stack_full(stack)) {
        cout << "Houve um erro na fila" << endl;
```

```
    } else {
        stack.items[++stack.top] = value;
    }
}

int stack_pop(Stack &stack) {
    if (stack_empty(stack)) {
        cout << "Houve um erro na fila" << endl;
        exit(1);
    } else {
        return stack.items[stack.top--];
    }
}

void initialize_queue(Queue &queue) {
    queue.stack1.top = -1;
    queue.stack2.top = -1;
}

void queue_push(Queue &queue, int value) {
    if(!stack_full(queue.stack1)) {
        stack_push(queue.stack1, value);
        return;
    }
    if(stack_empty(queue.stack2)) {
        for(int i = 0; i < MAX_ITEMS; i++) {
            int transited_value = stack_pop(queue.stack1);
            stack_push(queue.stack2, transited_value);
        }
        stack_push(queue.stack1, value);
        return;
    }
    cout << "A fila está cheia!" << endl << endl;
}

int queue_pop(Queue &queue) {
    if(!stack_empty(queue.stack2)) {
        return stack_pop(queue.stack2);
    }
    if(!stack_empty(queue.stack1)) {
        while(queue.stack1.top >= 0) {
            int transited_value = stack_pop(queue.stack1);
            stack_push(queue.stack2, transited_value);
        }
        return stack_pop(queue.stack2);
    }
    cout << "A fila está vazia!" << endl;
    exit(1);
}

void queue_print(Queue queue) {
    cout << "Fila: " << endl;
    int i = 0;

    cout << "Entrada: ";
    while(i <= queue.stack1.top) {
        cout << " <- " << queue.stack1.items[i];
        i++;
    }
    cout << endl << "Saída: ";
    i = 0;
    while(i <= queue.stack2.top) {
```

```
        cout << queue.stack2.items[i] << " -> ";
        i++;
    }
    cout << endl << endl;
}

int main() {
    Queue queue;
    initialize_queue(queue);

    queue_push(queue, 3);
    queue_push(queue, 7);
    queue_push(queue, 2);
    queue_print(queue);

    queue_push(queue, 5);
    queue_push(queue, 8);
    queue_print(queue);

    queue_push(queue, 0);
    queue_push(queue, 9);
    queue_push(queue, 4);
    queue_push(queue, 1);
    queue_push(queue, 6);
    queue_print(queue);

    cout << "Testando push na fila cheia: ";
    queue_push(queue, 10); // Testar se enche a fila

    int popped_value = queue_pop(queue);
    cout << "Removido valor: " << popped_value << endl << endl;

    queue_print(queue);

    popped_value = queue_pop(queue);
    cout << "Removido valor: " << popped_value << endl << endl;

    queue_print(queue);

    cout << "Testando push na pilha 1 cheia: ";
    queue_push(queue, -1);

    cout << "Removendo 3 valores: " << endl << endl;
    queue_pop(queue);
    queue_pop(queue);
    queue_pop(queue);
    queue_print(queue);

    cout << "Removendo mais um valor: " << endl << endl;
    queue_pop(queue);
    queue_print(queue);

    cout << "Removendo 4 valores: " << endl << endl;
    queue_pop(queue);
    queue_pop(queue);
    queue_pop(queue);
    queue_pop(queue);
    queue_print(queue);

    cout << "Testando remoção com a fila totalmente vazia: ";
    queue_pop(queue);
```

```
    return 0;
}
```

Implemente em C/C++ uma pilha utilizando duas filas. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```
#include <iostream>
#define MAX_ITEMS 5

using namespace std;

struct Queue {
    int front;
    int back;
    int items[MAX_ITEMS];
};

struct Stack {
    Queue queue1;
    Queue queue2;
    Queue* main_queue;
};

void initialize_queue(Queue &queue) {
    queue.front = 0;
    queue.back = -1;
}

void initialize_stack(Stack &stack) {
    initialize_queue(stack.queue1);
    initialize_queue(stack.queue2);
    stack.main_queue = &stack.queue1;
}

bool queue_empty(Queue &queue) {
    return queue.front > queue.back;
}

bool queue_full(Queue &queue) {
    return queue.back == MAX_ITEMS - 1;
}

void queue_push(Queue &queue, int value) {
    if (queue_full(queue)) {
        cout << "Pilha cheia!" << endl;
        return;
    }
    queue.items[++queue.back] = value;
}

int queue_pop(Queue &queue) {
    if (queue_empty(queue)) {
        cout << "Pilha vazia!" << endl;
        exit(1);
    }
    int removed = queue.items[queue.front];

    for (int i = queue.front; i < queue.back; i++) {
        queue.items[i] = queue.items[i + 1];
    }
    queue.back--;
```

```
    return removed;
}

void stack_push(Stack &stack, int value) {
    queue_push(*stack.main_queue, value);
}

int stack_pop(Stack &stack) {
    Queue* aux_queue;

    if(stack.main_queue == &stack.queue1) {
        aux_queue = &stack.queue2;
    } else {
        aux_queue = &stack.queue1;
    }
    while(stack.main_queue->back >= 1) {
        int popped_value = queue_pop(*stack.main_queue);
        queue_push(*aux_queue, popped_value);
    }
    int stack_popped_value = queue_pop(*stack.main_queue);

    stack.main_queue = aux_queue;

    return stack_popped_value;
}

void stack_print(Stack stack) {
    cout << "Pilha: ";
    for(int i = 0; i <= stack.main_queue->back; i++) {
        cout << stack.main_queue->items[i] << " -> ";
    }
    cout << "Saída" << endl << endl;
}

int main() {
    Stack stack;
    initialize_stack(stack);

    stack_print(stack);

    stack_push(stack, 3);
    stack_push(stack, 7);
    stack_print(stack);

    stack_push(stack, 5);
    stack_push(stack, 4);
    stack_print(stack);

    stack_push(stack, 1);
    stack_print(stack);

//    stack_push(stack, 1); // Testando limite de items

    int popped = stack_pop(stack);
    cout << "Removido: " << popped << endl;
    stack_print(stack);

    popped = stack_pop(stack);
    cout << "Removido: " << popped << endl;
    stack_print(stack);
```

```
    stack_push(stack, 9);
    stack_push(stack, 8);
    stack_print(stack);

    popped = stack_pop(stack);
    cout << "Removido: " << popped << endl;
    stack_print(stack);

    return 0;
}
```

Implemente em C/C++ uma pilha usando lista singularmente ligada. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```
#include <iostream>

using namespace std;

struct Node {
    int value;
    Node* next;
};

struct Stack {
    Node* head;
};

bool stack_is_empty(Stack* stack) {
    return stack->head == nullptr;
}

void push(Stack* stack, int new_value) {
    Node* new_node = new Node;
    new_node->value = new_value;
    new_node->next = stack->head;
    stack->head = new_node;
}

int pop(Stack* stack) {
    if (stack_is_empty(stack)) {
        cout << "A pilha está vazia, não há nada para remover." << endl;
        exit(1);
    }
    Node* node = stack->head;
    stack->head = stack->head->next;
    int removed_value = node->value;
    delete node;
    return removed_value;
}

void print(Stack stack) {
    Node* node = stack.head;

    cout << "Pilha: ";

    while (node != nullptr) {
        cout << node->value << " -> ";
        node = node->next;
    }
    cout << "nullptr" << endl;
}
```

```
int main() {
    Stack stack;
    stack.head = nullptr;

    print(stack);

    push(&stack, 3);
    print(stack);

    push(&stack, 7);
    push(&stack, 8);
    print(stack);

    int popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

    popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

    popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

    // pop(&stack); Usar para testar a validação de erro

    return 0;
}
```

Implemente em C/C++ uma pilha usando lista duplamente ligada. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```
#include <iostream>

using namespace std;

struct Node {
    int value;
    Node* next;
    Node* prev;
};

struct Stack {
    Node* head;
};

bool stack_is_empty(Stack* stack) {
    return stack->head == nullptr;
}

void push(Stack* stack, int new_value) {
    Node* new_node = new Node;
    new_node->value = new_value;
    new_node->next = stack->head;
    new_node->prev = nullptr;

    if (stack->head != nullptr) {
        stack->head->prev = new_node;
    }
```

```
    stack->head = new_node;
}

int pop(Stack* stack) {
    if (stack_is_empty(stack)) {
        cout << "A pilha está vazia, não há nada para remover." << endl;
        exit(1);
    }
    Node* node = stack->head;
    stack->head = stack->head->next;

    if (stack->head != nullptr) {
        stack->head->prev = nullptr;
    }
    int removed_value = node->value;
    delete node;
    return removed_value;
}

void print(Stack stack) {
    Node* node = stack.head;

    cout << "Pilha: ";

    while (node != nullptr) {
        cout << node->value << " -> ";
        node = node->next;
    }
    cout << "nullptr" << endl;
}

int main() {
    Stack stack;
    stack.head = nullptr;

    print(stack);

    push(&stack, 3);
    print(stack);

    push(&stack, 7);
    push(&stack, 8);
    print(stack);

    int popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

    popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

    popped_value = pop(&stack);
    cout << "Removido: " << popped_value << endl;
    print(stack);

     //pop(&stack); //Usar para testar a validação de erro

    return 0;
}
```

Implemente em C/C++ uma fila usando lista singularmente ligada. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```cpp
#include <iostream>

using namespace std;

struct Node {
    int value;
    Node* next;
};

struct Queue {
    Node* head;
};

bool queue_is_empty(Queue* queue) {
    return queue->head == nullptr;
}

void push(Queue* queue, int new_value) {
    Node* new_node = new Node;
    new_node->value = new_value;
    new_node->next = queue->head;
    queue->head = new_node;
}

int pop(Queue* queue) {
    if (queue_is_empty(queue)) {
        cout << "A fila está vazia, não há nada para remover." << endl;
        exit(1);
    }
    Node* selected_node = queue->head;

    if (selected_node->next == nullptr) {
        int removed_value = selected_node->value;
        delete selected_node;
        queue->head = nullptr;
        return removed_value;
    }
    while(selected_node->next->next != nullptr) {
        selected_node = selected_node->next;
    }
    int removed_value = selected_node->next->value;
    delete selected_node->next;
    selected_node->next = nullptr;
    return removed_value;
}

void print(Queue queue) {
    Node* node = queue.head;

    cout << "Fila: ";

    while (node != nullptr) {
        cout << node->value << " -> ";
        node = node->next;
    }
    cout << "nullptr (saída)" << endl;
}
```

```
int main() {
    Queue queue;
    queue.head = nullptr;

    print(queue);

    push(&queue, 2);
    print(queue);

    push(&queue, 5);
    push(&queue, 6);
    print(queue);

    push(&queue, 1);
    push(&queue, 4);
    print(queue);

    int popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

//     pop(&queue); // Usar para testar a validação de erro

    return 0;
}
```

Implemente em C/C++ uma fila usando lista duplamente ligada. Analise a complexidade computacional das operações de inserção e remoção de seu código.

```
#include <iostream>

using namespace std;

struct Node {
    int value;
    Node* next;
    Node* prev;
};

struct Queue {
    Node* head;
    Node* tail;
};

bool queue_is_empty(Queue* queue) {
    return queue->head == nullptr;
}
```

```
void push(Queue* queue, int new_value) {
    Node* new_node = new Node;
    new_node->value = new_value;
    new_node->next = nullptr;
    new_node->prev = queue->tail;

    if (queue->tail != nullptr) {
        queue->tail->next = new_node;
    }
    queue->tail = new_node;

    if (queue->head == nullptr) {
        queue->head = new_node;
    }
}

int pop(Queue* queue) {
    if (queue_is_empty(queue)) {
        cout << "A fila está vazia, não há nada para remover." << endl;
        exit(1);
    }
    Node* node_to_remove = queue->head;
    int removed_value = node_to_remove->value;

    queue->head = node_to_remove->next;

    if (queue->head != nullptr) {
        queue->head->prev = nullptr;
    } else {
        queue->tail = nullptr;
    }
    delete node_to_remove;
    return removed_value;
}

void print(Queue queue) {
    Node* node = queue.head;

    cout << "Fila: ";

    while (node != nullptr) {
        cout << node->value << " -> ";
        node = node->next;
    }
    cout << "nullptr (saída)" << endl;
}

int main() {
    Queue queue;
    queue.head = nullptr;
    queue.tail = nullptr;

    print(queue);

    push(&queue, 2);
    print(queue);

    push(&queue, 5);
    push(&queue, 6);
    print(queue);
```

```
    push(&queue, 1);
    push(&queue, 4);
    print(queue);

    int popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    popped_value = pop(&queue);
    cout << "Removido: " << popped_value << endl;
    print(queue);

//     pop(&queue); // Usar para testar a validação de erro

    return 0;
}
```