Pós-Graduação em Computação – UFF
Estruturas de Dados e Algoritmos – Profº Ígor Machado Coelho
Lista 2 – Estruturas Lineares
Aluno: Ítalo Leite Ferreira Portinho

Questão 1(a):

Primeiramente vamos definir o tipo do nó a ser usado no Deque. Inicialmente implementei sem o ponteiro ant , porém seria necessário percorrer o deque todo para remover um elemento do fim do deque (para saber o elemento anterior a ele e poder atualizar o fim):

```
class NoDeque {
 public:
    char dado;
    NoDeque* prox;
    NoDeque* ant;
};

class Deque {
 public:
    NoDeque* inicio;
    NoDeque* fim;
    int N;

    // Função para inicializar o deque
    // COMPLEXIDADE: O(1)
    void cria() {
        this->N = 0;
        this->inicio = 0;
        this->fim = 0;
    }

    // Função para retornar o elemento inicio do Deque
    // COMPLEXIDADE: O(1)
    char retornaInicio() {
        return this->inicio->dado;
    }

    // Função para retornar o elemento final do Deque
    // COMPLEXIDADE: O(1)
    char retornaFim() {
        return this->fim->dado;
    }
```

```
// Função para inserir elemento no inicio do Deque
    // Cria-se um nó com o ponteiro prox apontando para o inicio
do deque.
    //  e ponteiro ant para 0.
    // SE for o primeiro nó a ser inserido, o fim tb é o inicio.
    // SENAO for o primeiro no a ser inserido, o ant do antigo
inicio tem que
    //  apontar para o novo no (sera promovido à inicio).
    // Após, atualizamos o inicio para ser esse nó recém criado
    // Atualizamos também o tamanho N do deque.
    // COMPLEXIDADE: O(1)
    void insereInicio(char novo) {
        NoDeque* no = new NoDeque {
                .dado = novo
                , .prox = this->inicio
                , .ant = 0
        };
        if (N == 0) {
            this->fim = no;
        } else {
            this->inicio->ant = no;
        }
        this->inicio = no;
        this->N++;
    }

    // Função para inserir elemento no fim do Deque
    // Cria-se um nó com o ponteiro prox apontando para 0.
    //  e ponteiro ant(anterior) apontando para o fim.
    // Após, atualizamos o ponteiro prox do ultimo elemento do
deque(this->fim)
    //  para este nó recém criado.
    // Atualizamos o fim do deque para este nó recém criado.
    // Atualizamos também o tamanho N do deque.
    // COMPLEXIDADE: O(1)
    void insereFim(char novo) {
        NoDeque* no = new NoDeque { .dado = novo, .prox = 0, .ant
= this->fim };
        if (N > 0) {
            this->fim->prox = no;
        } else {
            no->ant = 0;
            this->inicio = no;
        }
        this->fim = no;
        this->N++;
    }
```

```
// Função para remover elemento do início do Deque
    // Guardamos o ponteiro prox para depois atualizar o inicio
    // Guardamos o dado do elemento inicio, para retornar ao
final.
    // Deletamos o nó inicio antigo, e atualizamos o inicio para o
nó guardado.
    // COMPLEXIDADE: O(1)
    char removeInicio() {
        NoDeque* no = this->inicio->prox;
        char dado = this->inicio->dado;
        delete this->inicio;
        this->N--;
        this->inicio = no;
        return dado;
    }

    // Função para remover elemento do fim do Deque.
    // Obs.:
    // Foi adotada um implementação duplamente encadeada para esta
função ser executada em tempo constante O(1), do contrário seria
necessário percorrer todo o deck (até N - 1) para poder remover o
último elemento e atualizar o penultimo para último, com
complexidade O(N) .

    // Guardamos o ponteiro anterior do final (para depois
promove-lo ao fim).
    // Este nó terá seu prox para 0, pois vamos deletar para onde
ele aponta.
    // Guardamos o dado do fim para poder retornar.
    // Deletemos o fim e atualizamos this->fim para o nó guardado.
    // Retornamos o dado.
    // COMPLEXIDADE: O(1)
    char removeFim() {
        NoDeque* no = this->fim->ant;
        no->prox = 0;
        char dado = this->fim->dado;
        delete this->fim;
        this->N--;
        this->fim = no;
        return dado;
    }
    // Função auxiliar para mostrar o deque para fins de teste.
    void mostraDeque() {
        NoDeque* head = this->inicio;
        printf("\n");
        for ( ; head; head = head->prox ) {
            printf("%c \n", head->dado);
        }
        printf("\n");
    }
```

```
};
```

Questão 1(b):

Para implementar uma pilha usando o deque, inserimos e removemos no fim do deque.

COMPLEXIDADE:
empilha: a mesma complexidade de insereFim, O(1).
desempilha: a mesma complexidade de removeFim, O(1).
topo: O(1).
tamanho: O(1).

```
class PilhaDeque {
 public:
    Deque d;
    char topo() {
        return d.fim->dado;
    }
    void empilha(char novo) {
        d.insereFim(novo);
    }
    char desempilha() {
        return d.removeFim();
    }
    int tamanho() {
        return d.N;
    }
};
```

Questão 1(c):

Para implementar um fila usando o deque, devemos inserir e remover em extremos opostos do deque. Aqui vou inserir no fim e remover do inicio.

COMPLEXIDADE:
enfileira: a mesma complexidade de insereFim, O(1).
desenfileira: a mesma complexidade de removeInicio, O(1).
frente: O(1).
fim: O(1)
tamanho: O(1).

```
class FilaDeque {
 public:
    Deque d;
    char frente() {
        return d.inicio->dado;
    }
    char fim() {
        return d.fim->dado;
```

```
    }
    void enfileira(char novo) {
        d.insereFim(novo);
    }
    char desenfileira() {
        return d.removeInicio();
    }
    int tamanho() {
        return d.N;
    }
};
```

Questão 2:

Implementação de uma pilha usando duas filas f1 e f2.

Para a função empilhar, basta apenas inserir na fila. Para o topo usamos a função back da implementação nativa de fila da linguagem C++. Para a função desempilhar transferimos N - 1 elementos para a fila f2. Resta na fila f1 o topo da pilha, e retornamos ele. Depois, transferimos de volta todos os elementos de f2 para f1.

COMPLEXIDADE:
topo: O(1) - assumindo que std::queue ter ponteiro para o final da fila.
empilha: O(1) - assumindo complexidade constante de push (como no deque).
desempilha: O(N) - é preciso percorrer a fila toda.

```
class Pilha2F {
 public:
    std::queue<char> f1;
    std::queue<char> f2;
    int N;
    void cria() {
        N = 0;
    }
    char topo() {
        return f1.back();
    }
    void empilha(char dado) {
        f1.push(dado);
        N++;
    }
    char desempilha() {
        int i = N;
        char valor;
        // Todos os elementos menos o primeiro são transferidos
para a fila 2.
        // Agora na fila 1 resta somente o topo da pilha.
//Guardamos esse valor para retornar depois, damos pop, e
//atualizamos N1.
        while (i < N - 1) {
```

```
            f2.push(f1.front());
            f1.pop();
            i++;
        }
        valor = f1.front();
        f1.pop();
        N--;

        // Agora retornamos todos os elementos da fila 2 para a
fila 1.
        i = N;
        while (i < N) {
            f1.push(f2.front());
            f2.pop();
        }
        return valor;
    }

    // Função auxiliar para mostrar o tamanho da pilha.
    int tamanho() {
        return this->N;
    }
};
```

Questão 3:

Implementação de uma fila usando duas pilhas.

```
class Fila2P {
 public:
    std::stack<char> p1;
    std::stack<char> p2;
    int N;
    char _back;
    void cria() {
        N = 0;
        _back = ' ';
    }
```

Enfileira: Para enfileirar, passaremos todos os elementos para p2. Colocamos o novo elemento em p1, em seguida passamos todo mundo de p2 de volta para p1. Dessa forma, o método top() da pilha vai poder se usado como o método front() da fila. Para saber quem está no final da fila, método back(), sempre atualizamos esse valor ao enfileirar alguém.

COMPLEXIDADE: O(N)

```
    void enfileira(char dado) {
        int i = 0;
        while (i < N) {
            p2.push(p1.top());
```

```
            p1.pop();
            i++;
        }

        p1.push(dado);

        i = 0;
        while (i < N) {
            p1.push(p2.top());
            p2.pop();
            i++;
        }
        N++;
        _back = dado;
    }
```

Desenfileira: Com nossa implementação o topo da pilha coincide com a frente da fila. Guardamos o topo para retornar. Esse elemento sai da pilha e atualizamos nosso N.

COMPLEXIDADE: O(1)

```
    char desenfileira() {
        char valor = p1.top();
        p1.pop();
        N--;
        return valor;
    }
```

Frente: Com nossa implementação o topo da pilha coincide com a frente da fila.

COMPLEXIDADE: O(1)

```
    char front() {
        return p1.top();
    }
```

Back: Basta retornar o valor que guardamos sempre alguém entra na fila.

COMPLEXIDADE: O(1)

```
    char back() {
        return this->_back;
    }
```

Questão 4(a):

Inverter uma pilha usando uma fila: A estratégia aqui é transferir todos os elementos para a fila, e em seguida transferi-los de volta para a pilha.

COMPLEXIDADE: O(N) - percorre a pilha e a fila inteiras uma vez cada.

```
void inverte_A(std::stack<char>* p) {
    std::queue<char> fila;
    while (p->size() > 0) {
        fila.push(p->top());
        p->pop();
    }
    while (fila.size() > 0) {
        p->push(fila.front());
        fila.pop();
    }
}
```

Questão 4(b):

Inverter uma pilha usando duas pilhas: A estratatégia aqui é transferir todos os elementos da pilha original para a pilha auxiliar p1. Em seguida transferir N – 1 elementos de p1 para p2. O primeiro elemento da pilha original está em p1, e deve voltar pra pilha orinal agora. O resto dos elementos estão invertidos em p2 e basta transferi-los de volta para a pilha original.

COMPLEXIDADE: O(N) - é preciso percorrer cada pilha uma vez.

```
void inverte_B(std::stack<char>* p) {
    std::stack<char> p1;
    std::stack<char> p2;

    // transferindo da original pra p1 auxiliar.
    while (p->size() > 0) {
        p1.push(p->top());
        p->pop();
    }

    // transferindo N - 1 elementos de p1 pra p2
    while (p1.size() > 1) {
        p2.push(p1.top());
        p1.pop();
    }

    // O elemento restante em p1, volta pra pilha original.
    p->push(p1.top());
    p1.pop();

    // Todo mundo da pilha auxiliar p2 volta pra original.
```

```
    while (p2.size() > 0) {
        p->push(p2.top());
        p2.pop();
    }
}
```

Questão 4(c):

Inverter uma pilha usando somente outra pilha: A estratégia aqui é guardar o topo e transferir N - i elementos para a pilha auxiliar, com i = 0 inicialmente. Em seguida, topo volta para a pilha original, e todos os elementos da pilha auxiliar também voltam pra pilha original. Ao final desse ciclo i é incrementado e seguimos repetindo a operação até que i seja igual a N.

COMPLEXIDADE: O(N^2) – Percorremos a pilha N vezes, depois N – 1, N – 2, N – 3…. Totalizando (N^2 + N)/2

```
void inverte_C(std::stack<char>* p) {
    std::stack<char> p1;
    char old_top;
    int i = 0;
    int N = p->size();
    while (i < N) {
        old_top = p->top();
        p->pop();
        while (p->size() > i) {
            p1.push(p->top());
            p->pop();
        }
        p->push(old_top);
        i++;
        while (p1.size() > 0) {
            p->push(p1.top());
            p1.pop();
        }
    }
}
```

Questão 5(a):

Inverter uma fila usando somente uma pilha: A estratégia aqui é todos os transferir os elementos da fila para a pilha. Em seguida transferir de volta para a fila e, os elementos estarão com a ordem invertida. A estratégia funciona porque na fila inserimos no final da estrutura, e na pilha inserimos no início, antes do primeiro elemento.

COMPLEXIDADE: O(N) - percorremos a fila/pilha inteira uma vez cada.

```
void inverte_A(std::queue<char>* f) {
    std::stack<char> p1;
    while (f->size() > 0) {
        p1.push(f->front());
        f->pop();
    }
    while (p1.size() > 0) {
        f->push(p1.top());
        p1.pop();
    }
}
```

Questão 5(b):

Inverter uma fila usando somente duas filas: A estratégia aqui é passar N - 1 elementos para a fila auxiliar f1, e o elemento restante para a fila f2. Quem foi para f1 retorna para a fila original e repetimos o ciclo até que todos os elementos estejam em f2, já invertidos. Depois é só transferir os elementos de volta de f2 para a fila original.

COMPLEXIDADE: $O(N^2)$ - cada loop interno percorre a fila inteira uma vez. Esse ciclo é repetido N vezes (loop externo).

```
void inverte_B(std::queue<char>* f) {
    std::queue<char> f1;
    std::queue<char> f2;
    int i = 0;
    int N = f->size();

    while (i < N) {
        // Passamos todos os elementos, menos o final da original,
para f1.
        while (f->size() > 1) {
            f1.push(f->front());
            f->pop();
        }

        // O final da fila original vai para f2.
        f2.push(f->front());
        f->pop();

        // Quem estava em f1 retorna para a fila original.
```

```
        while (f1.size() > 0) {
            f->push(f1.front());
            f1.pop();
        }
        i++;
    }

    // Devolvemos os elementos para a fila original, agora invertidos.
    while (f2.size() > 0) {
        f->push(f2.front());
        f2.pop();
    }
}
```

Questão 6:

Obter o menor elemento de uma pilha em tempo constante: Vamos manter uma variável com o valor mínimo e na hora de empilhar um elemento, fazemos uma comparação para atualizar o mínimo, se for necessário. Essa estratégia funciona, porém surge um problema quando desempilhamos o elemento mínimo. Como encontrar o novo valor mínimo? A estratégia é criar um tipo NoMin para o valor mínimo. Assim podemos ter um ponteiro para o mínimo anterior ao empilhar um novo mínimo. Ao desempilhar, basta atualizar o mínimo para esse valor.

COMPLEXIDADE:
topo: O(1)
empilha: O(1)
desempilha: O(1)
obterMinimo: O(1)

```
class NoPilhaMin {
 public:
    int dado;
    NoPilhaMin* prox;
};

class NoMin {
 public:
    int dado;
    NoMin* previous_min;
};

class PilhaMin {
 public:
    int N;
    NoPilhaMin* inicio;
    NoMin* min;

    void cria() {
```

```
        this->N = 0;
        this->inicio = 0;
    }
    int topo() {
        return this->inicio->dado;
    }
    void empilha(int novo) {
        // Atualiza o minimo
        if (this->N == 0) {
            NoMin* new_min = new NoMin { .dado =
novo, .previous_min = 0 };
            min = new_min;
        } else {
            if (novo < this->min->dado) {
                NoMin* new_min = new NoMin { .dado =
novo, .previous_min = this->min };
                this->min = new_min;
            }
        }
        // Atualiza o topo
        NoPilhaMin* no = new NoPilhaMin { .dado = novo, .prox =
this->inicio };
        this->inicio = no;
        this->N++;
    }

    //      Ao desempilhar temos que testar se o mínimo precisa ser
atualizado.
    // Basta apontar para o mínimo anterior.
    int desempilha() {
        if (this->inicio->dado == this->min->dado) {
            NoMin* new_min = this->min->previous_min;
            delete this->min;
            this->min = new_min;
        }
        NoPilhaMin* no = this->inicio->prox;
        int valor = this->inicio->dado;
        delete inicio;
        this->inicio = no;
        this->N--;
        return valor;
    }

    int obterMinimo() {
        return this->min->dado;
    }
};
```

Questão 7:

Transformar uma expressão matemática para RPN, notação polonesa reversa: A expressão é dividida em tokens e armazenada em um array de char. Os tokens são lidos um a um, ações são tomadas após classificar os tokens em três grupos:

- Se for um "(" ou operador "+,-,*,/", apenas colocamos na pilha.
- Se for um operando, armazenamos em uma pilha auxiliar. Se essa pilha já tiver um elemento, transferimos este para o buffer de saída e empilhamos o novo elemento. Essa pilha deve ter no máximo um elemento.
- Se for um ")", havendo alguém na pilha de operandos, transferimos para o buffer de saída. Em seguida desempilhamos operadores da pilha principal até encontrarmos o "(" correspondente.

COMPLEXIDADE:
Cada token é lido , empilhado ou desempilhado somente uma vez. Portanto a complexidade é O(N), o que é condizendo com o fato da solução apresentada ser uma implementação do algoritmo Shunting yard, que tem complexidade linear.

```
int main() {
    char const ABRE_PARENTESES = '(';
    char const FECHA_PARENTESES = ')';
    char const OPERADOR_SOMA = '+';
    char const OPERADOR_SUBTRACAO = '-';
    char const OPERADOR_MULTIPLICACAO = '*';
    char const OPERADOR_DIVISAO = '/';
    char token;
    std::stack<char> pilha_1;  // parenteses e operadores.
    std::stack<char> pilha_2;  // operandos.
    std::queue<char> saida_rpn;  // saida na notação polonesa
reversa (rpn)
    std::string expressao = "((A+B)*(C-(F/D)))";
    // "((A+B)*C)";

    char* char_array = new char[expressao.length() + 1];

    // copying the contents of the
    // string to char array
    strcpy(char_array, expressao.c_str());
    for (int i = 0; i < expressao.length(); i++) {
        printf("%c", char_array[i]);
    }
    printf("\n", ' ');
    bool continuar = true;
    for (int i = 0; i < expressao.length(); i++) {
        token = char_array[i];
        switch (token) {
            case ABRE_PARENTESES:
            case OPERADOR_SOMA:
            case OPERADOR_SUBTRACAO:
            case OPERADOR_MULTIPLICACAO:
            case OPERADOR_DIVISAO:
```

```
                printf("E operador ou abre parenteses: %c \n",
token);
                pilha_1.push(token);
                break;
            case FECHA_PARENTESES:
                // desempilhar até encontrar um parenteses esquerdo
                //  e colocar na fila
                printf("E fecha parenteses: %c \n", token);
                if (pilha_2.size() > 0) {
                    saida_rpn.push(pilha_2.top());
                    pilha_2.pop();
                }
                continuar = true;
                while (continuar) {
                    char topo = pilha_1.top();
                    if (!(topo == ABRE_PARENTESES)) {
                        saida_rpn.push(topo);
                    } else {
                        continuar = false;
                    }
                    pilha_1.pop();
                    printf("Saiu da pilha_1: %c \n", topo);
                }
                break;
            default:
                // É um operando
                printf("E operando: %c \n", token);
                if (pilha_2.size() > 0) {
                    saida_rpn.push(pilha_2.top());
                    pilha_2.pop();
                }
                pilha_2.push(token);
                break;
        }
    }

    while (saida_rpn.size() > 0) {
        printf("%c", saida_rpn.front());
        saida_rpn.pop();
    }
}
```