JULIO CESAR GARCIA RIBEIRO **RA: 1994484**

Resolução P1 – Estrutura de Dados (EDCO3A)

Exercício 1.

```
1 – FILA = {10}
2 – FILA = {10, 42}
3 – FILA = {10, 42, -4}
4 – FILA = {10, 42, -4, 25}
5 – FILA = {10, 42, -4, 25, 17}
6 – Erro ao inserir -> fila cheia. FILA = {10, 42, -4, 25, 17}
7 – FILA = {42, -4, 25, 17}
8 – Primeiro elemento = 42
9 – FILA = {-4, 25, 17}
10 – FILA = {25, 17}
11 – FILA = {25, 17, 36}
12 – FILA = {25, 17, 36, 67}
13 – FILA = {17, 36, 67}
14 – FILA = {36, 67}
15 – FILA = {36, 67, 90}
```

Exercício 2.

Pilhas são estrutura de dados lineares em que os elementos são organizados um em cima do outro, como em uma pilha de pratos. As operações de inserção e de exclusão são realizadas da mesma extremidade, o que faz com que esta obedeça a ordem FILO (First In Last Out), garantindo que o primeiro elemento inserido é o último a ser removido.

Pilhas podem ser implementadas de maneira estática ou dinâmica. Em uma aplicação estática o tamanho da pilha é pré-definido pelo programador, e este não pode ser alterado durante a execução do programa, isso acontece pois é utilizado um vetor com tamanho definido para armazenar os elementos da pilha. Já a implementação dinâmica possibilita aumentar e diminuir o tamanho da estrutura por meio de alocação e liberação de espaços de memória. Os elementos de uma pilha dinâmica são encadeados por meio de ponteiros que são organizados de acordo com a inserção ou remoção de elementos na pilha. Cabe ao programador definir qual a melhor implementação a se usar, analisando o problema que deve ser resolvido e a quantidade de espaços na memória que este pode ocupar.

Pilhas podem ser utilizadas em aplicações como: Gerenciamento de memória (a memória RAM é organizada um elemento em cima do outros); Avaliação de expressões matemáticas; Conversão de tipos (decimal para binário como exemplo).

As operações de inserção e remoção da pilha são de complexidade O(1) (são realizadas apenas no elemento do topo) e operações de acesso e pesquisa são O(n) (se deve percorrer os elementos da pilha para encontrar o desejado). Se as pesquisas na pilha só forem feitas analisando apenas o elemento do topo, e o tamanho da estrutura for guardado em uma variável então operações de pesquisa e contagem de elementos serão O(1).

Filas são estruturas lineares semelhantes às pilhas, mas a sua ordem é FIFO, ou seja, o primeiro elemento a entrar é o primeiro a sair. Esta estrutura possui duas extremidades para realizar as operações de inserção e remoção, o que faz com que os elementos inseridos sempre entrem no final da fila e os elementos removidos sempre saiam do começo da fila.

As implementações são as mesmas das pilhas (estáticas ou dinâmicas).

Algumas aplicações de filas são: Representação de filas da vida real (a ordem de chegada importa); Requisição de aplicações em processos single-threaded (o primeiro a chegar é o primeiro a ser atendido).

A complexidade de operações são as mesmas das pilhas (inserção e remoção O(1), acesso e pesquisa O(n), ou O(1) se for avaliado apenas o primeiro e último elemento da fila).

Listas são estruturas de dados em que existe uma ordenação de elementos (por ordem crescente ou decrescente).

A implementação dessa estrutura é feita de maneira dinâmica, podendo ser singly-linked (apenas uma relação entre os elementos, apontando para o próximo elemento), doubly-linked (com duas ligações entre os elementos, apontando para o próximo e para o anterior). Também existe uma implementação utilizando um nó sentinela, que aponta para o primeiro e para o último elemento da lista, o que garante um comportamento circular para a lista.

Os elementos novos são inseridos em uma posição que gere uma ordem para a lista. Se formos inserir um elemento menor que o primeiro a complexidade é O(1), e O(n) para demais inserções (é necessário percorrer a lista para encontrar a posição de inserção). Se for uma implementação com nó sentinela, a complexidade para inserções de elementos maiores que o último é O(1) também, pois existe uma referência para o último elemento. A remoção segue a mesma lógica, se a chave fornecida para remoçar estiver na primeira posição será O(1), se a lista tiver um nó sentinela e a posição for a mesma do último elemento é O(1) e O(n) para as demais. Operações de acesso e pesquisa são O(n).

Listas são estruturas muito usadas, e aparecem de forma simples como estruturas contendo dados organizados e até em aplicações como: Visualizador de imagens (imagens linkadas com próxima imagem e imagem anterior); player de músicas (próxima música ou anterior), entre outras aplicações que usam elementos ordenados e ligados entre si.

Exercício 3.

Estado inicial: `A = {5, 4, 3, 2, 1}; B = {}; C = {}`

1 – `push(B, pop(A))` `A = {5, 4, 3, 2}; B = {1}; C = {}`
2 – `push(C, pop(A))` `A = {5, 4, 3}; B = {1}; C = {2}`
3 – `push(C, pop(B))` `A = {5, 4, 3}; B = {}; C = {2, 1}`
4 – `push(B, pop(A))` `A = {5, 4}; B = {3}; C = {2, 1}`
5 – `push(A, pop(C))` `A = {5, 4, 1}; B = {3}; C = {2}`
6 – `push(B, pop(C))` `A = {5, 4, 1}; B = {3, 2}; C = {}`
7 – `push(B, pop(A))` `A = {5, 4}; B = {3, 2, 1}; C = {}`
8 – `push(C, pop(A))` `A = {5}; B = {3, 2, 1}; C = {4}`
9 – `push(C, pop(B))` `A = {5}; B = {3, 2}; C = {4, 1}`
10 – `push(A, pop(B))` `A = {5, 2}; B = {3}; C = {4, 1}`
11 – `push(A, pop(C))` `A = {5, 2, 1}; B = {3}; C = {4}`
12 – `push(C, pop(B))` `A = {5, 2, 1}; B = {}; C = {4, 3}`
13 – `push(B, pop(A))` `A = {5, 2}; B = {1}; C = {4, 3}`

14 – push(C, pop(A)) A = {5}; B = {1}; C = {4, 3, 2}
15 – push(C, pop(B)) A = {5}; B = {}; C = {4, 3, 2, 1}
16 – push(B, pop(A)) A = {}; B = {5}; C = {4, 3, 2, 1}
17 – push(A, pop(C)) A = {1}; B = {5}; C = {4, 3, 2}
18 – push(B, pop(C)) A = {1}; B = {5, 2}; C = {4, 3}
19 – push(B, pop(A)) A = {}; B = {5, 2, 1}; C = {4, 3}
20 – push(A, pop(C)) A = {3}; B = {5, 2, 1}; C = {4}
21 – push(C, pop(B)) A = {3}; B = {5, 2}; C = {4, 1}
22 – push(A, pop(B)) A = {3, 2}; B = {5}; C = {4, 1}
23 – push(A, pop(C)) A = {3, 2, 1}; B = {5}; C = {4}
24 – push(B, pop(C)) A = {3, 2, 1}; B = {5, 4}; C = {}
25 – push(B, pop(A)) A = {3, 2}; B = {5, 4, 1}; C = {}
26 – push(C, pop(A)) A = {3}; B = {5, 4, 1}; C = {2}
27 – push(C, pop(B)) A = {3}; B = {5, 4}; C = {2, 1}
28 – push(B, pop(A)) A = {}; B = {5, 4, 3}; C = {2, 1}
29 – push(A, pop(C)) A = {1}; B = {5, 4, 3}; C = {2}
30 – push(B, pop(C)) A = {1}; B = {5, 4, 3, 2}; C = {}
31 – push(B, pop(A)) A = {}; B = {5, 4, 3, 2, 1}; C = {}

Exercício 4.

Segue print da função, da main() e da saída obtida.

```
ListNodePtr split(List *list, int n) {
    if(isEmpty(list)) {
        printf("Não foi possivel separar -> Lista vazia\n");
        return NULL;
    }

    ListNodePtr aux;
    for(aux = list->start; aux != NULL; aux = aux->next) {
        if(aux->x == n)
            break;
    }

    if(aux == NULL) {
        printf("Não foi possivel separar -> Lista fornecida não tem uma chave igual a n\n");
        return NULL;
    }

    if(aux->next == NULL) {
        printf("Não foi possivel separar -> n fornecido é o último elemento\n");
        return NULL;
    }

    ListNodePtr separada;
    separada = aux->next;
    separada->previous = NULL;
    aux->next = NULL;
    return separada;
}
```

```c
int main() {
    List list;
    init(&list);
    insert(&list, 10);
    insert(&list, 3);
    insert(&list, 7);
    insert(&list, 9);
    insert(&list, 6);
    insert(&list, 4);
    insert(&list, 8);
    printListIncreasing(&list);

    ListNodePtr ret;
    ret = split(&list, 6);
    if(ret != NULL) {
        List splitted;
        init(&splitted);
        splitted.start = ret;
        printListIncreasing(&list);
        printListDecreasing(&list);
        printListIncreasing(&splitted);
        printListDecreasing(&splitted);
    }

    return 0;
}
```

```
Lista crescente = { 3 4 6 7 8 9 10 }
Lista crescente = { 3 4 6 }
Lista decrescente = { 6 4 3 }
Lista crescente = { 7 8 9 10 }
Lista decrescente = { 10 9 8 7 }
[1] + Done                       "/us
```

Exercício 5.

Segue print da função, da main() e da saída obtida.

```c
List* mergeLists(List *list1, List *list2) {
    if(isEmpty(list1) || isEmpty(list2)) {
        printf("Erro ao unir as listas -> uma das listas é vazia");
        return NULL;
    }

    static List merged;
    init(&merged);
    ListNodePtr aux = list1->start;
    while(aux != NULL) {
        if(!searchFast(&merged, aux->x))
            insert(&merged, aux->x);
        aux = aux->next;
    }
    aux = list2->start;
    while(aux != NULL) {
        if(!searchFast(&merged, aux->x))
            insert(&merged, aux->x);
        aux = aux->next;
    }

    return &merged;
}
```

```c
int main() {
    List list1;
    init(&list1);
    insert(&list1, 10);
    insert(&list1, 3);
    insert(&list1, 7);
    printListIncreasing(&list1);
    List list2;
    init(&list2);
    insert(&list2, 8);
    insert(&list2, 45);
    insert(&list2, 7);
    insert(&list2, 32);
    printListIncreasing(&list2);

    List *mergedList;
    mergedList = mergeLists(&list1, &list2);
    printListIncreasing(mergedList);
    printListDecreasing(mergedList);

    printListIncreasing(&list1);
    printListIncreasing(&list2);

    return 0;
}
```

```
Lista crescente = { 3 7 10 }
Lista crescente = { 7 8 32 45 }
Lista crescente = { 3 7 8 10 32 45 }
Lista decrescente = { 45 32 10 8 7 3 }
Lista crescente = { 3 7 10 }
Lista crescente = { 7 8 32 45 }
[1] + Done                       "/usr/
```

Exercício 6.

a.

É necessário um tipo caixa contendo variável para o peso e o nome da caixa, um tipo pilha contendo um ponteiro para o topo da pilha e uma variável para contar quantos elementos existem na mesma, um tipo nó de pilha que contêm um ponteiro para o próximo elemento da pilha e uma variável do tipo caixa e um tipo ponteiro nó de pilha que é um ponteiro para cada elemento da pilha. Se usa também um vetor de pilhas, que foi definido na main().

```c
typedef struct {
    float weight;
    char name[100];
} Box;

typedef struct StackNode *StackNodePtr;

typedef struct StackNode {
    Box box;
    StackNodePtr next;
} StackNode;

typedef struct {
    StackNodePtr top;
    int size;
} DynamicStack;
```

b.

Segue print da função, da main() e da saída obtida.

```c
int push(Stack *stack, int size, Box *box) {
    StackNodePtr aux;
    aux = (StackNodePtr)malloc(sizeof(StackNode));
    aux->box = *box;
    for(int i = 0; i < size; i++) {
        if((stack+i)->size == 0 || ((stack+i)->size < 8 && (stack+i)->top->box.weight > box->weight)) {
            aux->next = (stack+i)->top;
            (stack+i)->top = aux;
            (stack+i)->size++;
            return i;
        }
    }

    printf("Nao foi possivel empilhar a caixa\n");
    return -1;
}
```

```c
int main() {
    int size = 5;
    Stack stacks[size];
    for(int i = 0; i < size; i++)
        init(&stacks[i]);

    Box box = {125.8, "caixa"};
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 100;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 150;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 200;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 164;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 148;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 32;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 45;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 21;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));
    box.weight = 10;
    printf("Caixa empilhada na pilha %d\n", push(stacks, size, &box));

    return 0;
}
```

```
Caixa empilhada na pilha 0
Caixa empilhada na pilha 0
Caixa empilhada na pilha 1
Caixa empilhada na pilha 2
Caixa empilhada na pilha 2
Caixa empilhada na pilha 1
Caixa empilhada na pilha 0
Caixa empilhada na pilha 1
Caixa empilhada na pilha 0
Caixa empilhada na pilha 0
[1] + Done
```