IMD0029 - Estrutura de Dados Básicas 1 – 2017.2 – Prova 04
Prof. Eiji Adachi M. Barbosa

Nome:_____
Matrícula: _____

ANTES DE COMEÇAR A PROVA, leia atentamente as seguintes instruções:

- Esta é uma prova escrita de caráter individual e sem consultas a pessoas ou material (impresso ou eletrônico).
- A prova vale 10, pontos e o valor de cada questão é informado no seu enunciado.
- **Preze por respostas legíveis, bem organizadas e simples.**
- As respostas devem ser fornecidas preferencialmente em caneta. Respostas fornecidas a lápis serão aceitas, mas eventuais questionamentos sobre a correção não serão aceitos.
- Celulares e outros dispositivos eletrônicos devem permanecer desligados durante toda a prova.
- Desvios éticos ou de honestidade levarão à anulação da prova do candidato (nota igual a zero).

Questão 1 (2,5 pontos): Considere que um array inicialmente ordenado foi deslocado a direita sem querer e não se sabe quantas vezes ele foi deslocado a direita. Um deslocamento a direita de um array faz com que o último elemento do array torne-se o primeiro. Por exemplo: dado o array de entrada {1,2,3,4,5,6,7,8,9,10}, após 3 deslocamentos a direita, o array estará da seguinte forma: {8,9,10,1,2,3,4,5,6,7}. Neste contexto, implemente uma função capaz de descobrir quantas vezes um array A foi deslocado. Em outras palavras, implemente uma função que recebe como entrada um array A que foi deslocado algumas vezes a direita, e retorna um inteiro X indicando quantas vezes o array de entrada foi deslocado a direita. Sua solução deverá ter complexidade Θ (lg(n)).

Questão 2 (3,0 pontos): Suponha que você tem à sua disposição uma Pilha genérica (ou parametrizada) já implementada corretamente, provendo as seguintes operações:

- `void Stack<Element>::Pop()`
- `Element Stack<Element>::Top()`
- `Bool Stack<Element>::Push(Element)`
- `Int Stack<Element>::Size()`

Suponha também que você possui uma lista simplesmente encadeada implementada seguindo a estrutura abaixo:

```
List{
   Node* first;
}
```

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

Neste contexto, use a Pilha para inverter uma instância desta lista simplesmente encadeada. Sua solução deverá seguir a assinatura `void List::Revert()`.

Obs.: A solução deverá única e exclusivamente operar sobre a lista usando uma pilha auxiliar. A solução não poderá criar outras listas auxiliares, nem qualquer outra estrutura auxiliar que não seja a pilha (ex.: array).

Questão 3 (2,5 pontos): Em tabelas de dispersão implementadas seguindo o endereçamento aberto, todos os elementos estão armazenados no array interno da tabela hash. Ou seja, cada posição do array interno da tabela contém um elemento (um par (chave,valor)), ou NULL. Um possível problema com esta representação ocorre durante a remoção de um elemento da tabela. Ao removermos elementos de uma tabela de dispersão com endereçamento aberto, não podemos atribuir NULL a esta posição, mas sim marcá-la com DELETED. Desta forma, podemos diferenciar as posições que estão vazias desde o momento da criação da tabela (posições NULL), daquelas posições que já foram ocupadas, mas tiveram seus elementos removidos posteriormente (posições DELETED). No entanto, esta solução para a remoção traz uma complicação para o método put da tabela de dispersão. Em sala de aula, implementamos o método put usando uma estratégia "olhar adiante" (do inglês, *look ahead*). Nesta estratégia, nós inserimos novos elementos em posições DELETED. No entanto, antes de realmente inserirmos um elemento numa posição DELETED, nós precisamos "olhar adiante" para ver se a chave deste elemento já não se encontra na tabela, caso em que nós devemos apenas atualizar o valor do elemento. Esta solução baseada na estratégia "olhar adiante" causou bastante confusão no último mini teste, em que poucos alunos conseguiram implementar corretamente o método put com a estratégia "olhar adiante".

Uma estratégia alternativa para implementarmos o método put, e que é bem mais simples, é chamada de estratégia "preguiçosa" (do inglês, *lazy*). Nesta estratégia, as posições DELETED são perdidas, isto é, nós não tentamos inserir em posições DELETED; nós inserimos apenas em posições NULL. Já a parte de atualização de valores na tabela, que também é feita pelo método put, continua do mesmo jeito que é realizado na estratégia "olhar adiante". Quando a tabela ficar muito ocupada, isto é, com muitas posições com elementos válidos ou DELETED, nós aplicamos um

redimensionamento e com isso "limpamos" as posições DELETED da tabela, liberando espaços para novos elementos. Neste contexto, implemente o método put de acordo com a estratégia preguiçosa descrita acima e tratando colisões por sondagem linear. Sua implementação deve seguir a assinatura `void HashTable::Put(K key, V value)` e operar sobre a estrutura abaixo:

```
HashTable{
   HashEntry** data;
   Int size;
   Int quantity;
}
```

```
HashEntry {
   K key;
   V value;
}
```

Obs.: Nesta questão, você pode assumir que a função hash já está implementada e pode ser reusada. Não é necessário se preocupar com o redimensionamento dinâmico da tabela.

Questão 5 (2,0 pontos): Para cada uma das afirmações a seguir, marque V (verdadeiro) ou F (falso), **justificando sucintamente** sua resposta (mesmo as verdadeiras devem ser justificadas). Marcações de V ou F **sem justificativas não serão aceitas.**

1. ( ) O algoritmo *Merge Sort* possui complexidade assintótica $\Theta(n*\log_2(n))$ para o melhor caso e $\Theta(n^2)$ para o pior caso.
2. ( ) Os algoritmos *Insertion Sort*, *Bubble Sort* e *Selection Sort* possuem a mesma complexidade assintótica para o melhor caso.
3. ( ) O pior caso do *Quick Sort* é quando nas sucessivas chamadas recursivas se divide um problema de tamanho N em dois sub-problemas de tamanhos bastante similares.
4. ( ) Os algoritmos de busca sequencial e de binária podem ser empregados nos mesmos tipos de array.
5. ( ) As operações de inserir e remover elementos de uma Pilha só podem ser realizadas em tempo $\Theta(1)$ quando implementamos a Pilha usando listas encadeadas, não sendo possível realiza-las em tempo $\Theta(1)$ usando uma implementação baseada em arrays.
6. ( ) As operações de inserir e remover elementos de uma Fila só podem ser realizadas em tempo $\Theta(1)$ quando implementamos a Fila usando listas encadeadas, não sendo possível realiza-las em tempo $\Theta(1)$ usando uma implementação baseada em arrays.
7. ( ) Se tivermos um Deque à nossa disposição, é possível usá-lo como uma Pilha, mas não é possível usá-lo como uma Fila.
8. ( ) De posse de uma lista simplesmente encadeada com ponteiros para início e fim, é possível criarmos uma estrutura que segue a estratégia F.I.L.O. (first-in last-out) da seguinte forma: ao realizarmos uma *inserção* nós inserimos no início da lista e ao realizarmos uma *remoção* nós removemos do fim da lista.
9. ( ) De posse de uma lista simplesmente encadeada com ponteiros para início e fim, é possível criarmos uma estrutura que segue a estratégia F.I.F.O. (first-in first-out) da seguinte forma: ao realizarmos uma *inserção* nós inserimos no início da lista e ao realizarmos uma *remoção* nós removemos do fim da lista.
10. ( ) As operações de inserir no início e no fim são realizadas em tempo constante apenas em listas duplamente encadeada com sentinelas cabeça e cauda, não sendo possível realiza-las em tempo constante em listas simplesmente encadeadas.