IMD0029 - Estrutura de Dados Básicas 1 – 2017.2 – Prova 03
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 6,5 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 (2,0 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, considere o código a seguir que implementa o método put seguindo a estratégia preguiçosa:

```
bool put(const K _key, const V _value){
	unsigned long index = hash(_key);
	while(this->data[index] != nullptr) {
		if( _key == this->data[index]->getKey() ) {
			this->data[index]->setValue(_value);
			return true;
		}
		++index;
	}
	this->data[index] = new HashEntry(_key, _value);
	return true;
}
```

O código apresentado para o método put apresenta alguns defeitos em sua implementação. Nesta questão, liste os defeitos que você encontrou no código acima e reescreva o método put corrigindo tais defeitos. Não precisa se preocupar com defeitos de tipos e nomes de variáveis, caso existam. Preocupe-se apenas com erros na lógica do método put seguindo a estratégia "preguiçosa", o que inclui instruções que estão no código, mas estão erradas, e instruções que deveriam estar no código, mas não estão.

Obs.: Nesta questão, não é necessário se preocupar com o redimensionamento dinâmico da tabela.

Questão 2 (2,0 pontos): Dicionários são tipos abstratos de dados que modelam coleções de elementos, em que cada elemento corresponde a um par (chave, valor), ou, em inglês, (Key, Value). Suponha que você tem à sua disposição um dicionário já implementado corretamente e que provê as seguintes operações:

```
Dictionary:: Dictionary ( ); // Construtor
bool Dictionary::Put( K key, V value ); // Inserir-Atualizar
V Dictionary::Remove( K key ); // Remover
V Dictionary::Get( K key ); // Buscar
```

Nesta questão, reuse este dicionário para resolver o seguinte problema: dado um array de strings de entrada, contendo algumas strings repetidas, monte um dicionário que registre as posições em que cada palavra ocorre no array de entrada. Sua função deverá ter a seguinte assinatura (input é o array de entrada e inputSize é a quantidade de elementos deste array):

```
Dictionary<string, list<int>> countWords(string[] input, int inputSize)
```

Obs.: Nesta questão, você deverá apenas reusar o Dicionário; você não deverá tentar de alguma forma modificar a sua implementação. Assuma que cada string no array de entrada possui apenas caracteres alfanuméricos e que todas as letras das strings estão em caixa alta (i.e., estão maiúsculas).

Questão 3 (1,5 ponto): O problema 2-Sum é definido da seguinte forma. Dado um array A de entrada e um valor inteiro K, retorne verdadeiro se existem dois elementos x e y em A tais que x + y = K, ou falso, caso contrário. Suponha que você possui a sua disposição a implementação de um dicionário que garante, sob qualquer condição, que as operações put, remove e delete são feitas em tempo $\Theta(1)$. Neste contexto, use este dicionário para resolver o problem 2-sum em tempo $\Theta(n)$ (use as mesmas assinaturas dos métodos definidos para o dicionário da questão anterior).

Obs.: Nesta questão, você deverá apenas reusar o Dicionário; você não deverá tentar de alguma forma modificar a sua implementação. Assuma que cada string no array de entrada possui apenas caracteres alfanuméricos e que todas as letras das strings estão em caixa alta (i.e., estão maiúsculas).

Questão 4 (1,0 ponto): Suponha que alguém implementou uma tabela de dispersão com redimensionamento dinâmico. Nesta implementação, todas as vezes que o fator de carga fica maior ou igual do que 0.5, a tabela é expandida. Similarmente, todas as vezes que o fator de carga fica menor ou igual do que 0.25, a tabela é reduzida. Após uma bateria de testes, verificou-se que esta implementação não é muito eficiente. Em particular, a tabela de dispersão foi testada nas seguintes condições: primeiro, foram inseridos elementos na tabela até que ela estivesse com uma quantidade de itens exatamente menor do que o limite superior do fator de carga (ex: tabela com tamanho 17 e com 8 elementos); em seguida, foram feitas sucessivamente operações de inserir, remover, inserir, remover, inserir, remover... Neste contexto, (a) explique e justifique por que esta implementação não se mostrou eficiente nos testes e (b) sugira uma melhoria simples para o problema apresentado por esta tabela de dispersão.