IMD0029 - Estrutura de Dados Básicas 1 – 2018.1 – 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 5,0 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 (1,5 ponto): Em tabelas de dispersão implementadas seguindo o endereçamento aberto, todos os elementos estão armazenados no array interno da tabela de dispersão. 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, daquelas posições que já foram ocupadas, mas tiveram seus elementos removidos posteriormente. 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.

Uma estratégia alternativa para implementarmos o método *put* em tabelas de dispersão com endereçamento aberto, 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".

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. Sua solução deve partir do código acima e corrigi-lo, e não reescrevê-lo totalmente diferente a partir do zero. 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 (1,0 ponto): 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 arquivo texto de entrada, monte um dicionário que registre o número das linhas em que cada palavra ocorre neste arquivo de entrada.

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 arquivo texto de entrada possui apenas caracteres alfanuméricos e que todas as letras das strings estão em caixa alta (i.e., estão maiúsculas). Além disso, pode assumir que existem as seguintes funções auxiliares:

- *Arquivo AbreArquivo(String s) – Recebe uma string com o nome do arquivo e retorna um objeto representando o respectivo arquivo texto.*
- *String LerLinha(Arquivo a) – Recebe um objeto Arquivo e retorna uma string correspondente à linha do texto atual. Quando o arquivo chegar ao fim, esta função retornará NULL.*
- *String[] QuebrarLinha(String l) – Recebe uma string l representando uma linha de texto e retorna um array de strings contendo cada palavra desta linha.*

Questão 3 (1,0 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 problema 2-sum em tempo $\Theta(n)$ (use as mesmas assinaturas dos métodos definidos para o dicionário da questão anterior).

Questão 4 (1,5 ponto): Em sala de aula, vimos o conceito de tabela de dispersão com redimensionamento dinâmico. Neste contexto, responda:

A) (0,5 ponto) Por que é interessante implementarmos a tabela de dispersão com redimensionamento dinâmico?

B) (1,0 ponto) Implemente a função *resize* abaixo, a qual é responsável por realizar o redimensionamento dinâmico da tabela de dispersão:

```
void HashTable::resize(int newSize)
```

Obs.: Assuma nesta questão que a estrutura da Tabela de Dispersão é a seguinte:

```
class HashTable {
	unsigned long size;
	unsigned long quantity;
	unsigned long hash(const std::string);
	HashEntry<std::string, std::string> **entries;

	unsigned long getSize();
	unsigned long getQuantity();
	bool isEmpty();
	bool isFull();
};
```