UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Algoritmos e Estruturas de Dados I (EDI)
Professor: Eduardo de Lucena Falcão
Exercício sobre Listas, Pilhas, Filas, Filas de Duas Pontas, e Algoritmos de Busca

Gerais

1. Explique a diferença entre um TAD e uma ED. Exemplifique.

2. A biblioteca de coleções da linguagem Java disponibiliza implementações de propósito geral para estruturas de dados elementares, como listas, filas e pilhas. Considere as seguintes definições de classes que representam implementações de estruturas de dados disponíveis na biblioteca da linguagem:

   • Classe A: os objetos são organizados em uma ordem linear e podem ser inseridos somente no início ou no final dessa sequência;

   • Classe B: os objetos são organizados em uma ordem linear determinada por uma referência ao próximo objeto;

   • Classe C: os objetos são removidos na ordem oposta em que foram inseridos;

   • Classe D: os objetos são inseridos e removidos respeitando a seguinte regra: o elemento a ser removido é sempre aquele que foi inserido primeiro.

   Nesse contexto, assinale a alternativa que representa, respectivamente, as estruturas de dados implementadas pelas classes A, B, C e D.

   a) Lista circular, lista simplesmente ligada, pilha e fila.
   b) Deque, lista simplesmente ligada, pilha e fila.
   c) Lista duplamente ligada, lista simplesmente ligada, fila e pilha.
   d) Pilha, fila, deque e lista simplesmente encadeada.
   e) Deque, pilha, lista ligada e fila.

Listas

1. Na linguagem GoLang, use a interface **`IList`** definida abaixo e programe as seguintes estruturas de dados: **`ArrayList`**, **`LinkedList`**, **`DoublyLinkedList`**. (replit)

```
type IList interface {
      Add(value int)
      AddOnIndex(value int, index int) error
      RemoveOnIndex(index int) error
      Get(index int) (int, error)
      Set(value int, index int) error
      Size() int
}
```

2. Considere as EDs apresentadas na tabela a seguir e responda o desempenho de tempo de pior caso e melhor caso para cada operação listada.

	`ArrayList`		**`LinkedList`**		**`DoublyLinkedList`**	
Operação	**Pior Caso**	**Melhor Caso**	**Pior Caso**	**Melhor Caso**	**Pior Caso**	**Melhor Caso**
`Add(value int)`	**O(n)** duplicar array		**O(n)** navegar até o final		**O(1)** usamos o tail	**Ômega(1)**
`AddOnIndex(value int, index int)`	**O(n)** deslocar a direita, e duplicar array		**O(n)** navegar até o final		**O(n)** no meio requer navegação (n/2)	
`RemoveOnIndex(index int)`	**O(n)** deslocar a esquerda	**Ômega(1)** final	**O(n)** navegar até o final	**Ômega(1)** início	**O(n)** no meio requer navegação (n/2)	**Ômega(1)** início e final
`Get(index int)`	**O(1)** end do vetor + index*xBytes	**Ômega(1)**	**O(n)** navegar até o final		**O(n)** no meio requer navegação (n/2)	
`Set(value int, index int)`	**O(1)** end do vetor + index*xBytes	**Ômega(1)**	**O(n)** navegar até o final		**O(n)** no meio requer navegação (n/2)	
`Size()`	**O(1)** variável p/		**O(1)** variável		**O(1)** variável p/	

	controlar		p/ controlar		controlar	

3. Cite uma vantagem e uma desvantagem do array list em relação à lista ligada.

4. Cite uma vantagem e uma desvantagem da lista duplamente ligada em relação à lista ligada.

5. Escreva uma função **in-place** para inverter a ordem de um ArrayList.

```
func (list *ArrayList) Reverse()
```

```
type ArrayList struct {
      values   []int
      inserted int
}
```

6. Escreva uma função **in-place** para inverter a ordem de uma LinkedList.

```
func (list *LinkedList) Reverse()
```

```
type LinkedList struct {
      head *Node
      size int
}
type Node struct {
      value int
      next  *Node
}
```

7. Escreva uma função **in-place** para inverter a ordem de uma DoublyLinkedList.

```
func (list *DoublyLinkedList) Reverse()
```

```
type DoublyLinkedList struct {
      head *Node2P
      tail *Node2P
      size int
}
type Node2P struct {
      prev  *Node
      value int
      next  *Node
}
```

8. Por que não faz sentido adicionarmos uma cauda (tail) em LinkedLists?

Pilhas

1. Na linguagem GoLang, use a interface **`IStack`** definida abaixo e programe as seguintes estruturas de dados: **`ArrayStack, LinkedListStack`**. (replit)

```
type IStack interface {
      Push(value int)
      Pop() (int, error)
      Peek() (int, error)
      IsEmpty() bool
      Size() int
}
```

2. Considere as EDs apresentadas na tabela a seguir e responda o desempenho de tempo de pior caso e melhor caso para cada operação listada.

	`ArrayStack`		**`LinkedListStack`**	
Operação	**Pior Caso**	**Melhor Caso**	**Pior Caso**	**Melhor Caso**
`Push(value int)`	**O(n)** duplicar		**O(1)** topo está na cabeça da lista	
`Pop() (int, error)`	**O(1)**		**O(1)** basta apontar a cabeça da lista p/ prox	
`Peek() (int, error)`	**O(1)**		**O(1)** topo está na cabeça da lista	
`IsEmpty()`	**O(1)**		**O(1)**	
`Size()`	**O(1)**		**O(1)**	

3. Escreva uma função que detecta se uma certa combinação de parênteses está balanceada. Dica 1: usar uma pilha. Dica 2: pensar nos casos de sucesso e casos de falha antes da implementação

```
func balparenteses(par string) bool
```

Filas

1. Mencione algumas aplicações de Filas.

2. Na linguagem GoLang, use a interface **`IQueue`** definida abaixo e programe as seguintes estruturas de dados: **`ArrayQueue, LinkedListQueue`**. (replit)

```
type IQueue interface {
      Enqueue(value int)
      Dequeue() (int, error)
      Front() (int, error)
      IsEmpty() bool
      Size() int
}
```

3. Considere as EDs apresentadas na tabela a seguir e responda o desempenho de tempo de pior caso e melhor caso para cada operação listada.

	ArrayQueue		**LinkedListQueue**	
Operação	**Pior Caso**	**Melhor Caso**	**Pior Caso**	**Melhor Caso**
`Enqueue(value int)`	**O(n) aumentar o array**		**O(1) uso o rear**	
`Dequeue() (int, error)`	**O(1)**		**O(1) uso o front**	
`Front() (int, error)`	**O(1)**		**O(1) uso o rear**	
`IsEmpty()`	**O(1)**		**O(1)**	
`Size()`	**O(1)**		**O(1)**	

4. Escreva uma função que retorne a quantidade de elementos inseridos em uma Fila implementada com vetor. Escreva a função **Size( )** considerando que o struct ArrayQueue **não contém a variável size**, como apresentado na tabela a seguir. Lembre-se que os índices de **front e rear** inicialmente assumem o valor -1, e que o ArrayQueue tem um caráter circular.

```
func (queue *ArrayQueue) Size()
```

```
type ArrayQueue struct {
```

```
    values []int
    front  int
    rear   int
}
```

Filas de Duas Pontas (Deque)

1. Mencione algumas aplicações de Deques.

2. Na linguagem GoLang, use a interface **`IQueue`** definida abaixo e programe as seguintes estruturas de dados: **`ArrayQueue, LinkedListQueue`**. (replit)

```
type IDeque interface {
      EnqueueFront(value int)
      EnqueueRear(value int)
      DequeueFront() (int, error)
      DequeueRear() (int, error)
      Front() (int, error)
      Rear() (int, error)
      IsEmpty() bool
      Size() int
}
```

3. Considere as EDs apresentadas na tabela a seguir e responda o desempenho de tempo de pior caso e melhor caso para cada operação listada.

	`ArrayDeque`		**`DoublyLinkedListDeque`**	
Operação	**Pior Caso**	**Melhor Caso**	**Pior Caso**	**Melhor Caso**
`EnqueueFront(value int)`	**O(n)** duplicar array		**O(1)**	
`EnqueueRear(value int)`	**O(n)** duplicar array		**O(1)**	
`DequeueFront() (int, error)`	**O(n)** duplicar array		**O(1)**	
`DequeueRear() (int, error)`	**O(n)** duplicar array		**O(1)**	
`Front() (int, error)`	**O(1)**		**O(1)**	
`Rear() (int, error)`	**O(1)**		**O(1)**	
`IsEmpty()`				

`Size()`				

Algoritmos de Busca

1. Explique a diferença e aplicabilidade entre uma busca linear e uma busca binária.
2. Qual a complexidade de tempo da busca linear e da busca binária? Apresente um gráfico com as duas funções, sendo o eixo horizontal referente ao espaço de busca (tamanho do vetor), e o eixo vertical referente à complexidade de tempo.
3. Implemente um algoritmo de busca binária que opere em vetores ordenados de modo crescente.

```
// versao recursiva
func bin_search(val int, list []int, start int, end int) int
// ou versao iterativa
func bin_search(val int, list []int) int
```

4. Implemente um algoritmo de busca binária que opere em vetores ordenados de modo decrescente.

```
// versao recursiva
func rev_bin_search(val int, list []int, start int, end int) int
// ou versao iterativa
func rev_bin_search(val int, list []int) int
```

5. Suponha que você queira criar uma nova implementação do TAD List que sempre se mantém ordenada: **`OrderedList`**. Uma forma de fazer isso seria anulando a função que permite adicionar em uma posição arbitrária, **AddOnIndex**, e ajustar a implementação de **Add(value int)** para que ela sempre adicionasse **value** na posição correta da lista. Proveja a implementação das funções de OrderedList, apresentadas na tabela a seguir.

```
type IList interface {
     Add(value int)
     AddOnIndex(value int, index int) error
     RemoveOnIndex(index int) error
     Get(index int) (int, error)
     Set(value int, index int) error
     Size() int
}
```

```
type OrderedList struct {
     //…
}

func (list *OrderedList) Add(val int) {}
func (list *OrderedList) AddOnIndex(value int, index int) error
{}
func (list *OrderedList) RemoveOnIndex(index int) error {}
func (list *OrderedList) Get(index int) (int, error) {}
```

```
func (list *OrderedList) Set(value, index int) error {}
func (list *OrderedList) Size() int {}
```

6. Qual estratégia você usou para encontrar a posição correta a ser adicionada o novo valor? Justifique sua escolha.

7. Faz sentido executar algoritmos de busca sobre quaisquer implementação de listas? Justifique sua resposta.

8. A linguagem Python não permite alguns tipos de otimização como, por exemplo, a recursão em cauda e, devido à sua natureza dinâmica, é impossível realizar esse tipo de otimização em tempo de compilação tal como em linguagens funcionais como Haskell ou ML.

   Disponível em: http://www.python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html
   Acesso: em 15 jun. 2019 (adaptado).

   O trecho de código a seguir, escrito em Python, realiza a busca binária de um elemento `x` em uma lista `lst` e a função `binary_search` tem código recursivo em cauda.

```
1 def binary_search(x, lst, low=None, high=None):
2     if low == None : low = 0
3     if high == None : high = len(lst)-1
4     mid = (high + low) // 2
5     if low > high :
6         return None
7     elif lst[mid] == x :
8         return mid
9     elif lst[mid] > x :
10        return binary_search(x, lst, low, mid-1)
11    else :
12        return binary_search(x, lst, mid+1, high)
```

   Acesso em: 15 jun. 2019 (adaptado).

   Considerando esse trecho de código, avalie as afirmações a seguir.

   I. Substituindo-se o conteúdo da linha 10 por `high = mid - 1` e substituindo-se o conteúdo da linha 12 por `low = mid + 1`, não se altera o resultado de uma busca.

   II. Envolvendo-se o código das linhas 4 a 12 em um laço `while True`, substituindo-se o conteúdo da linha 10 por `high = mid - 1` e substituindo-se o conteúdo da linha 12 por `low = mid + 1` remove-se a recursão de cauda e o resultado da busca não é alterado.

   III. Substituindo-se o código da linha 10 por:

```
newhigh = mid-1
return binary_search(x, lst, low, newhigh)
```

e substituindo-se o código da linha 12 por:

```
newlow = mid+1
return binary_search(x, lst, newlow, high)
```

remove-se a recursão de cauda.

IV. Substituindo-se o conteúdo das linhas 9 a 12 por:

```
if lst[mid] > x :
    newlow = low
    newhigh = mid-1
else:
    newlow = mid+1
    newhigh = high
return binary_search(x, lst, newlow, newhigh)
```

mantém-se o resultado da busca.

É correto o que se afirma em:

A. I, apenas.
B. II e III, apenas.
C. II e IV, apenas.
D. I, III e IV, apenas.
E. I, II, III e IV.