Lista 1 - Algo ED I - Respondida

Disciplina: Algoritmos e Estruturas de Dados I
Professor: Eduardo de Lucena Falcão
Aluno: Renan de Aquino Pereira
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.

O TAD define o que aquela estrutura faz, quais operações suporta e quais dados são trabalhados, mas não são definidos detalhes de implementação. Um exemplo disso seria a pilha que tem funções como Push que coloca no topo da pilha, Pop que remove do início da pilha e Peek que olha para o que está no topo da pilha.
Já a ED define a implementação, explicando como os dados serão manipulados, um exemplo disso pode ser visto abaixo:

```
// Push adiciona no topo
func (s *StackWithIndex) Push(e int) {
    if s.top == len(s.items) {
        // Por enquanto, vou ignorar a inserção se estiver cheio
        // Em um caso real, tem que ter um tratamento adequado para o aumento de
capacidade (caso tenha)
        return
    }
    // Adiciona no topo
    s.items[s.top] = e
    // Atualiza o indíce de topo
    s.top++
}

// Pop remove do topo
func (s *StackWithIndex) Pop() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    // Decrementa do topo e aponta para o último elemento válido
    s.top--
    // Retorna o elemento que estava no topo
    item := s.items[s.top]
    return item, nil
}

// Peek "espia" o elemento na posição "top - 1"
func (s *StackWithIndex) Peek() (int, error) {
```

```
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    return s.items[s.top-1], nil
}
```

2.A ordem de classes que representam as implementações de estruturas de dados disponíveis da linguagem JAVA

É a **opção b)**.

3.Qual a estrutura de dados mais adequada para o registro de recordes?

Seria a **opção c) Pilha**, por justamente ter a dinâmica de podermos olhar os pontos mais recentes por ser uma LIFO.

4.Estrutura de dados do Texto Estruturado

Olhando para as funções:

- `insereEvento` adiciona um novo evento no final da estrutura, controlado pelo índice `fim`;
- `processaEvento` recupera um evento do início da estrutura, controlado pelo índice `inicio`;
  Então a dinâmica representa uma fila, pois como os elementos são processados na ordem em que foram inseridos, os mais antigos são processados primeiro.
  Sendo assim a **resposta é letra a)**

Listas

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

Implementação da ArrayList

```
// ==== Implementação de ArrayList ====

type ArrayList struct {
    v        []int
    inserted int
}

// Init cria e retorna uma nova instância de ArrayList.
// Complexidade: O(1) no pior caso, pois o tempo não depende da entrada
func (l *ArrayList) Init(size int) {
    l.v = make([]int, size)
}

// Size retorna o tamanho do Array
// Complexidade: theta(1) no melhor e pior caso, pois o tempo não depende da entrada
func (list *ArrayList) Size() int {
    return list.inserted
}

// Get retorna o elemento da posição index
```

```go
// Complexidade: theta(1) no pior caso, pois o tempo não depende da entrada
func (list *ArrayList) Get(index int) (int, error) {
    if index >= 0 && index < list.inserted {
        return list.v[index], nil
    } else {
        return -1, errors.New(fmt.Sprintf("Index invalido: %dA lista tem %d elementos.",
index, list.inserted))
    }
}

// Set altera o valor do elemento na posição index
// Complexidade: theta(1) no melhor e pior caso, pois o tempo não depende da entrada
func (list *ArrayList) Set(e int, index int) error {
    if index < 0 || index >= list.inserted {
        return errors.New(fmt.Sprintf("Índice inválido: %d. A lista tem %d elementos.",
index, list.inserted))
    }
    list.v[index] = e
    return nil
}

// doubleV duplica a capacidade do Array
// Complexidade: theta(n) no melhor e pior caso, pois o tempo depende de uma multiplicação
func (list *ArrayList) doubleV() { //Theta(n)
    newV := make([]int, list.inserted*2)
    for i := 0; i < len(list.v); i++ {
        newV[i] = list.v[i]
    }
    list.v = newV
}

// Add adiciona um novo elemento no final
// Complexidade: O(n) no pior caso, pois o tempo depende de uma multiplicação da doubleV
func (list *ArrayList) Add(val int) { //O(n), Ômega(1)
    if list.inserted == len(list.v) {
        list.doubleV()
    }
    list.v[list.inserted] = val
    list.inserted++
}

// AddOnIndex adiciona um novo valor em um local específico
// Complexidade: O(n) no pior caso, pois o tempo depende de uma multiplicação da doubleV
func (list *ArrayList) AddOnIndex(e int, index int) error {
    if index < 0 || index > list.inserted {
        return errors.New(fmt.Sprintf("Índice inválido: %d. O índice deve estar entre 0 e
%d.", index, list.inserted))
    }

    if list.inserted == len(list.v) {
        list.doubleV()
    }
    //Desloca os elementos para a direita
```

```
    for i := list.inserted; i > index; i-- {
        list.v[i] = list.v[i-1]
    }

    // Insere o novo elemento na posição correta
    list.v[index] = e

    // Incrementa o contador de elementos inseridos
    list.inserted++

    return nil
}

// Remove remove o elemento de um índice específico.
// Complexidade: O(n) no pior caso, pois pode precisar deslocar todos os elementos.
func (list *ArrayList) Remove(index int) error {
    if index < 0 || index >= list.inserted {
        return errors.New(fmt.Sprintf("Índice inválido: %d. A lista tem %d elementos.",
index, list.inserted))
    }
    // Desloca os elementos
    for i := index; i < list.inserted-1; i++ {
        list.v[i] = list.v[i+1]
    }
    list.inserted--

    list.v[list.inserted] = 0 // Define o valor zero para o tipo int

    return nil
}

// Print imprime todos os elementos do ArrayList
// Complexidade: O(n) no pior caso, pois o tempo depende do número de elementos
func (list *ArrayList) Print() {
    for i := 0; i < list.inserted; i++ {
        fmt.Print(list.v[i], " ")
    }
    fmt.Println()
}
```

Implementação da LinkedList

```
// ==== Implementação de LinkedList ====

type Node struct {
    val  int
    next *Node
}

type LinkedList struct {
    head     *Node
    inserted int
}
```

```go
func (list *LinkedList) Size() int { //Theta(1)
    return list.inserted
}

func (list *LinkedList) Get(index int) (int, error) { //O(n), Ômega(1)
    if index >= 0 && index < list.inserted {
        aux := list.head
        for i := 0; i < index; i++ {
            aux = aux.next
        }
        return aux.val, nil
    } else {
        return -1, errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *LinkedList) Set(e int, index int) error {
    if index < 0 || index >= list.inserted {
        return errors.New(fmt.Sprintf("Índice inválido: %d. A lista tem %d elementos.",
index, list.inserted))
    }
    aux := list.head
    for i := 0; i < index; i++ {
        aux = aux.next
    }
    aux.val = e

    return nil
}

func (list *LinkedList) Add(val int) { //O(n), Ômega(1)
    newNode := &Node{val: val}
    if list.head == nil {
        list.head = newNode
    } else {
        aux := list.head
        for aux.next != nil {
            aux = aux.next
        }
        aux.next = newNode
    }
    list.inserted++
}

func (list *LinkedList) AddOnIndex(val int, index int) error { //O(n), Ômega(1)
    if index >= 0 && index <= list.inserted {
        newNode := &Node{val: val}
        if index == 0 {
            newNode.next = list.head
            list.head = newNode
        } else {
            aux := list.head
```

```
            for i := 0; i < index-1; i++ {
                aux = aux.next
            }
            newNode.next = aux.next
            aux.next = newNode
        }
        list.inserted++
        return nil
    } else {
        return errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *LinkedList) Remove(index int) error { //Ômega(1), O(n)
    if index >= 0 && index < list.inserted {
        if index == 0 {
            list.head = list.head.next
        } else {
            aux := list.head
            for i := 0; i < index-1; i++ {
                aux = aux.next
            }
            aux.next = aux.next.next
        }
        list.inserted--
        return nil
    } else {
        return errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *LinkedList) Print() {
    aux := list.head
    for aux != nil {
        fmt.Print(aux.val, " ")
        aux = aux.next
    }
    fmt.Println()
}
```

Implementação da DoublyLinkedList

```
// ==== Implementação de DoublyLinkedList ====

type DoublyNode struct {
    val  int
    next *DoublyNode
    prev *DoublyNode
}

type DoublyLinkedList struct {
    head     *DoublyNode
    tail     *DoublyNode
```

```
    inserted int
}

func (list *DoublyLinkedList) Size() int { //Theta(1)
    return list.inserted
}

func (list *DoublyLinkedList) Get(index int) (int, error) { //O(n), Ômega(1)
    if index >= 0 && index < list.inserted {
        aux := list.head
        for i := 0; i < index; i++ {
            aux = aux.next
        }
        return aux.val, nil
    } else {
        return -1, errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *DoublyLinkedList) Add(val int) { //O(n), Ômega(1)
    newNode := &DoublyNode{val: val}
    if list.head == nil {
        list.head = newNode
        list.tail = newNode
    } else {
        newNode.prev = list.tail
        list.tail.next = newNode
        list.tail = newNode
    }
    list.inserted++
}

func (list *DoublyLinkedList) AddOnIndex(val int, index int) error { //O(n), Ômega(1)
    if index >= 0 && index <= list.inserted {
        newNode := &DoublyNode{val: val}
        if index == 0 {
            newNode.next = list.head
            if list.head != nil {
                list.head.prev = newNode
            }
            list.head = newNode
            if list.tail == nil {
                list.tail = newNode
            }
        } else if index == list.inserted {
            newNode.prev = list.tail
            if list.tail != nil {
                list.tail.next = newNode
            }
            list.tail = newNode
        } else {
            aux := list.head
            for i := 0; i < index-1; i++ {
```

```
                aux = aux.next
            }
            newNode.next = aux.next
            newNode.prev = aux
            if aux.next != nil {
                aux.next.prev = newNode
            }
            aux.next = newNode
        }
        list.inserted++
        return nil
    } else {
        return errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *DoublyLinkedList) Remove(index int) error { //Ômega(1), O(n)
    if index >= 0 && index < list.inserted {
        if index == 0 {
            list.head = list.head.next
            if list.head != nil {
                list.head.prev = nil
            } else {
                list.tail = nil
            }
        } else if index == list.inserted-1 {
            list.tail = list.tail.prev
            if list.tail != nil {
                list.tail.next = nil
            } else {
                list.head = nil
            }
        } else {
            aux := list.head
            for i := 0; i < index; i++ {
                aux = aux.next
            }
            aux.prev.next = aux.next
            if aux.next != nil {
                aux.next.prev = aux.prev
            }
        }
        list.inserted--
        return nil
    } else {
        return errors.New(fmt.Sprintf("Index inválido: %d", index))
    }
}

func (list *DoublyLinkedList) Print() {
    aux := list.head
    for aux != nil {
        fmt.Print(aux.val, " ")
```

```
        aux = aux.next
    }
    fmt.Println()
}
```

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

Exemplo de vantagem seria o acesso a elemento por índice (usando `Get`) por causa da memória contínua, já uma desvantagem seria remover elementos do início ou do meio da lista, pois a complexidade é `O(n)`, pois vai ser necessário deslocar os elementos subsequentes.

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

Um exemplo de vantagem seria a possibilidade de percorrer nos dois sentidos (usando os ponteiros `head` e `tail`), onde ações como adicionar e remover no final da lista `O(1)`, sendo uma grande melhoria. E um exemplo de desvantagem seria a parte de maior consumo de memória, pois cada nó vai precisar armazenar um ponteiro extra (`prev`).

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

```
func (list *ArrayList) Reverse() {
    // Usa dois ponteiros, um no início e outro no fim
    for i, j := 0, list.inserted-1; i < j; i, j = i+1, j-1 {
        // Troca os elementos de posição
        list.v[i], list.v[j] = list.v[j], list.v[i]
        // E vão indo em direção ao centro do laço
    }
}
```

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

```
func (list *LinkedList) Reverse() {
    var prev *Node
    current := list.head
    var next *Node

    for current != nil {
        next = current.next // Guarda o próximo
        current.next = prev  // Inverte o ponteiro do nó atual
        prev = current       // Move o 'prev' para a frente
        current = next       // Move o 'current' para a frente
    }
    list.head = prev // O novo head é o antigo último elemento
}
```

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

```
func (list *DoublyLinkedList) Reverse() {
    current := list.head
    var temp *DoublyNode
```

```
    // Troca o head e o tail
    list.head, list.tail = list.tail, list.head

    // Percorre a lista trocando os ponteiros prev e next de cada nó
    for current != nil {
        temp = current.prev
        current.prev = current.next
        current.next = temp
        // Move para o próximo nó na lista invertida (que era o anterior)
        current = current.prev
    }
}
```

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

Caso seja adicionado um ponteiro `tail` em uma `LinkedList` a adição no final se torna uma operação `O(1)`, o que é muito bom. Porém, a operação de remoção do último elemento continua sendo `O(n)`. Isso acontece porque, para remover o último, você precisa do penúltimo elemento para que o `next` dele aponte para `nil` e em uma lista ligada de forma simples, não há como chegar ao penúltimo elemento a partir do `tail`, ou seja ainda precisaria percorrer tudo desde o `head`, dito isso o benefício acaba sendo bem limitado.

Pilhas

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

Implementação da ArrayStack

```
// ==== Implementação de Pilha usando ArrayList ====

type StackArrayList struct {
    list *ArrayList
}

// NewStackArrayList cria uma nova pilha.
func NewStackArrayList() *StackArrayList {
    newList := &ArrayList{}
    newList.Init(10)

    return &StackArrayList{list: newList}
}

// Size retorna o número de elementos na pilha.
func (s *StackArrayList) Size() int {
    return s.list.Size()
}

// IsEmpty verifica se a pilha está vazia.     '-'
func (s *StackArrayList) IsEmpty() bool {
    return s.list.Size() == 0
```

```
}

// Push adiciona um elemento no topo da pilha.
func (s *StackArrayList) Push(e int) {
    s.list.Add(e)
}

// Peek (ou topo) retorna o elemento do topo sem removê-lo.
func (s *StackArrayList) Peek() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    return s.list.Get(s.list.Size() - 1)
}

// Pop remove e retorna o elemento do topo da pilha.
func (s *StackArrayList) Pop() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    val, err := s.list.Get(s.list.Size() - 1)
    if err != nil {
        return 0, err
    }
    s.list.Remove(s.list.Size() - 1)
    return val, nil
}
```

Implementação da LinkedListStack

```
// ==== Implementação de Pilha usando LinkedList ====
type StackLinkedList struct {
    list *LinkedList
}

// NewStackLinkedList cria uma nova pilha.
func NewStackLinkedList() *StackLinkedList {
    return &StackLinkedList{list: &LinkedList{}}
}

// Size retorna o número de elementos na pilha.
func (s *StackLinkedList) Size() int {
    return s.list.Size()
}

// IsEmpty verifica se a pilha está vazia.
func (s *StackLinkedList) IsEmpty() bool {
    return s.list.Size() == 0
}

// Push adiciona um elemento no topo da pilha.
func (s *StackLinkedList) Push(e int) {
    s.list.AddOnIndex(e, 0)
```

```go
}

// Peek (ou topo) retorna o elemento do topo sem removê-lo.
func (s *StackLinkedList) Peek() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    return s.list.Get(0)
}

// Pop remove e retorna o elemento do topo da pilha.
func (s *StackLinkedList) Pop() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    val, _ := s.list.Get(0)
    s.list.Remove(0)
    return val, nil
}

type StackRaw struct {
    data []int
    top  int
}

// NewStackRaw cria uma nova pilha.
func NewStackRaw() *StackRaw {
    return &StackRaw{data: make([]int, 10), top: -1}
}

// Size retorna o número de elementos na pilha.
func (s *StackRaw) Size() int {
    return s.top + 1
}

// IsEmpty verifica se a pilha está vazia.
func (s *StackRaw) IsEmpty() bool {
    return s.top == -1
}

// Push adiciona um elemento no topo da pilha.
func (s *StackRaw) Push(e int) {
    if s.top+1 == len(s.data) {
        newData := make([]int, len(s.data)*2)
        copy(newData, s.data)
        s.data = newData
    }
    s.top++
    s.data[s.top] = e
}

// Peek (ou topo) retorna o elemento do topo sem removê-lo.
func (s *StackRaw) Peek() (int, error) {
```

```
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    return s.data[s.top], nil
}

// Pop remove e retorna o elemento do topo da pilha.
func (s *StackRaw) Pop() (int, error) {
    if s.IsEmpty() {
        return 0, errors.New("a pilha está vazia")
    }
    val := s.data[s.top]
    s.top--
    return val, nil
}
```

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 verificarBalanceamentoComArrayList(expressao string) bool {
    pilha := datastructures.NewStackArrayList()
    for _, char := range expressao {
        if char == '(' {
            pilha.Push(1)
        } else if char == ')' {
            if pilha.IsEmpty() {
                return false
            }
            pilha.Pop()
        }
    }
    return pilha.IsEmpty()
}
```

4.Sobre o texto apresentado, analise as afirmativas e escolha a opção correta

- Afirmação I: Correta
- Afirmação II: Correta
- Afirmação III: É obrigatório usar `free(p)` para liberar a memória da struct, mesmo que o array interno seja estático, logo a afirmação de que é opcional está incorreta.
  Então a reposta correta é a **letra c) I e II**

Filas

1.Mencione algumas aplicações de Filas

- Gerenciamento de Recursos Compartilhados: usando um exemplo de FreeRTOS em aplicações aeroespaciais, podemos descrever o comportamento da interpretação de dados de sensores, onde uma tarefa (`task_sensorRead`) produz (adquire) esses dados, os coloca em uma fila

(`sensor_data_queue`) e uma outra tarefa (`task_FSM`) consome esses dados (em ordem) e realiza os processamentos deles.

- Outro exemplo seguindo a mesma linha pode ser a de telecomandos vindo de uma estação de telemetria para a aviônica ou nanossatélite, pois podemos encadear uma sequência de comandos a serem executados e eles serem agrupados numa fila e uma tarefa, depois, processaria tais comandos;

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

Implementação em ArrayQueue

```
// ==== Implementação com ArrayList ====
// ineficiente para Dequeue

type QueueArrayList struct {
    list *ArrayList
}

func NewQueueArrayList() *QueueArrayList {
    list := &ArrayList{}
    list.Init(10) // Inicializa com capacidade 10
    return &QueueArrayList{list: list}
}

func (q *QueueArrayList) Size() int {
    return q.list.Size()
}

func (q *QueueArrayList) IsEmpty() bool {
    return q.list.Size() == 0
}

// Enqueue adiciona no fim do array - O(1) amortizado
func (q *QueueArrayList) Enqueue(e int) {
    q.list.Add(e)
}

// Dequeue remove do início do array - processo lento (O(n))
func (q *QueueArrayList) Dequeue() (int, error) {
    if q.IsEmpty() {
        return 0, errors.New("a fila está vazia")
    }
    val, _ := q.list.Get(0)
    q.list.Remove(0) // Esta operação desloca todos os elementos
    return val, nil
}

// Peek espia o início do array - Rápido (O(1))
func (q *QueueArrayList) Peek() (int, error) {
    if q.IsEmpty() {
        return 0, errors.New("a fila está vazia")
    }
```

```
    return q.list.Get(0)
}
```

Implementação em LinkedListQueue

```
// ==== Implementação feita a partir de DoublyLinkedList ====
type QueueLinkedList struct {
    list *DoublyLinkedList
}

func NewQueueLinkedList() *QueueLinkedList {
    return &QueueLinkedList{list: &DoublyLinkedList{}}
}

func (q *QueueLinkedList) Size() int {
    return q.list.Size()
}

func (q *QueueLinkedList) IsEmpty() bool {
    return q.list.Size() == 0
}

// Enqueue adiciona no final da lista (tail) - O(1)
func (q *QueueLinkedList) Enqueue(e int) {
    q.list.Add(e)
}

// Dequeue remove do início da lista (head) - O(1)
func (q *QueueLinkedList) Dequeue() (int, error) {
    if q.IsEmpty() {
        return 0, errors.New("a fila está vazia")
    }
    val, _ := q.list.Get(0)
    q.list.Remove(0)
    return val, nil
}

// Peek espia o início da lista (head) - O(1)
func (q *QueueLinkedList) Peek() (int, error) {
    if q.IsEmpty() {
        return 0, errors.New("a fila está vazia")
    }
    return q.list.Get(0)
}
```

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

---

```
type ArrayQueue struct {
    values []int
```

```
    front  int
    rear   int
}
func (queue *ArrayQueue) Size() int {
    // Se a fila está vazia, front e rear são -1.
    if queue.front == -1 {
        return 0
    }

    capacity := len(queue.values)
    // Se o ponteiro 'rear' ainda não deu a volta no array:
    if queue.rear >= queue.front {
        return queue.rear - queue.front + 1
    }

    // Se o ponteiro 'rear' já deu a volta:
    // A conta então fica: (tamanho total) - (espaço vazio no meio)
    return capacity - queue.front + queue.rear + 1
}
```

Filas de duas Pontas (Deque)

1.Mencione algumas aplicações de Deques.

- Gerenciador de históricos: Quando um usuário realiza uma certa ação ela é adicionada no final do deque, mas caso queira "desfazer" a última ação ela então é removida no final do deque, assim como caso queira "refazer" um "desfazer", essa ação desfeita é colocada de novo no final do deque.
- Escalonadores de Tarefas: Em algoritmos de escalonamento, cada processador tem seu próprio deque de tarefas com ações pra fazer. A ideia é cada um ir completando o seu próprio (olhando pro início do deque), porém quando o deque de um ficar vazio ele pode fazer uma tarefa no final do deque do outro processador, ajudando assim a terminar mais rápido e evitando um conflito entre suas ações.

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

Implementação de ArrayQueue

```
// ==== Implementação com Array Circular (ArrayDeque) ====

// ArrayDeque implementa um Deque usando um array circular de capacidade fixa.
type ArrayDeque struct {
    items    []int
    front    int
    rear     int
    size     int
    capacity int
}

// NewArrayDeque cria um novo Deque com capacidade fixa.
```

```go
func NewArrayDeque(capacity int) *ArrayDeque {
    return &ArrayDeque{
        items:    make([]int, capacity),
        front:    0,
        rear:     0,
        size:     0,
        capacity: capacity,
    }
}

func (d *ArrayDeque) IsEmpty() bool {
    return d.size == 0
}

func (d *ArrayDeque) isFull() bool {
    return d.size == d.capacity
}

func (d *ArrayDeque) Size() int {
    return d.size
}

// EnqueueRear adiciona um elemento no final (na 'rear' da fila).
func (d *ArrayDeque) EnqueueRear(value int) error {
    if d.isFull() {
        return errors.New("deque está cheio")
    }
    d.items[d.rear] = value
    d.rear = (d.rear + 1) % d.capacity
    d.size++
    return nil
}

// EnqueueFront adiciona um elemento no início (na 'front' da fila).
func (d *ArrayDeque) EnqueueFront(value int) error {
    if d.isFull() {
        return errors.New("deque está cheio")
    }
    d.front = (d.front - 1 + d.capacity) % d.capacity
    d.items[d.front] = value
    d.size++
    return nil
}

// DequeueFront remove um elemento do início.
func (d *ArrayDeque) DequeueFront() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    item := d.items[d.front]
    d.front = (d.front + 1) % d.capacity
    d.size--
    return item, nil
```

```
}

// DequeueRear remove um elemento do final.
func (d *ArrayDeque) DequeueRear() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    d.rear = (d.rear - 1 + d.capacity) % d.capacity
    item := d.items[d.rear]
    d.size--
    return item, nil
}

// Front espia o primeiro elemento.
func (d *ArrayDeque) Front() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    return d.items[d.front], nil
}

// Rear espia o último elemento.
func (d *ArrayDeque) Rear() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    rearIndex := (d.rear - 1 + d.capacity) % d.capacity
    return d.items[rearIndex], nil
}
```

Implementação de DoublyLinkedListQueue

```
// ==== Implementação com Lista Duplamente Ligada (DoublyLinkedListDeque) ====

// DoublyLinkedListDeque implementa um Deque usando a DoublyLinkedList existente.
type DoublyLinkedListDeque struct {
    list *DoublyLinkedList
}

// NewDoublyLinkedListDeque cria um novo Deque dinâmico.
func NewDoublyLinkedListDeque() *DoublyLinkedListDeque {
    return &DoublyLinkedListDeque{list: &DoublyLinkedList{}}
}

func (d *DoublyLinkedListDeque) IsEmpty() bool {
    return d.list.Size() == 0
}

func (d *DoublyLinkedListDeque) Size() int {
    return d.list.Size() // Usa o Size() da lista interna.
}

// EnqueueFront adiciona no início da lista (no 'head').
```

```go
func (d *DoublyLinkedListDeque) EnqueueFront(value int) error {
    return d.list.AddOnIndex(value, 0)
}

// EnqueueRear adiciona no final da lista (no 'tail').
func (d *DoublyLinkedListDeque) EnqueueRear(value int) error {
    d.list.Add(value)
    return nil
}

// DequeueFront remove do início da lista (o 'head').
func (d *DoublyLinkedListDeque) DequeueFront() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    val, _ := d.list.Get(0)
    d.list.Remove(0)
    return val, nil
}

// DequeueRear remove do final da lista (o 'tail').
func (d *DoublyLinkedListDeque) DequeueRear() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    lastIndex := d.list.Size() - 1
    val, _ := d.list.Get(lastIndex)
    d.list.Remove(lastIndex)
    return val, nil
}

// Front espia o primeiro elemento (o 'head' da lista).
func (d *DoublyLinkedListDeque) Front() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    return d.list.Get(0)
}

// Rear espia o último elemento (o 'tail' da lista).
func (d *DoublyLinkedListDeque) Rear() (int, error) {
    if d.IsEmpty() {
        return 0, errors.New("deque está vazio")
    }
    return d.list.Get(d.list.Size() - 1)
}
```

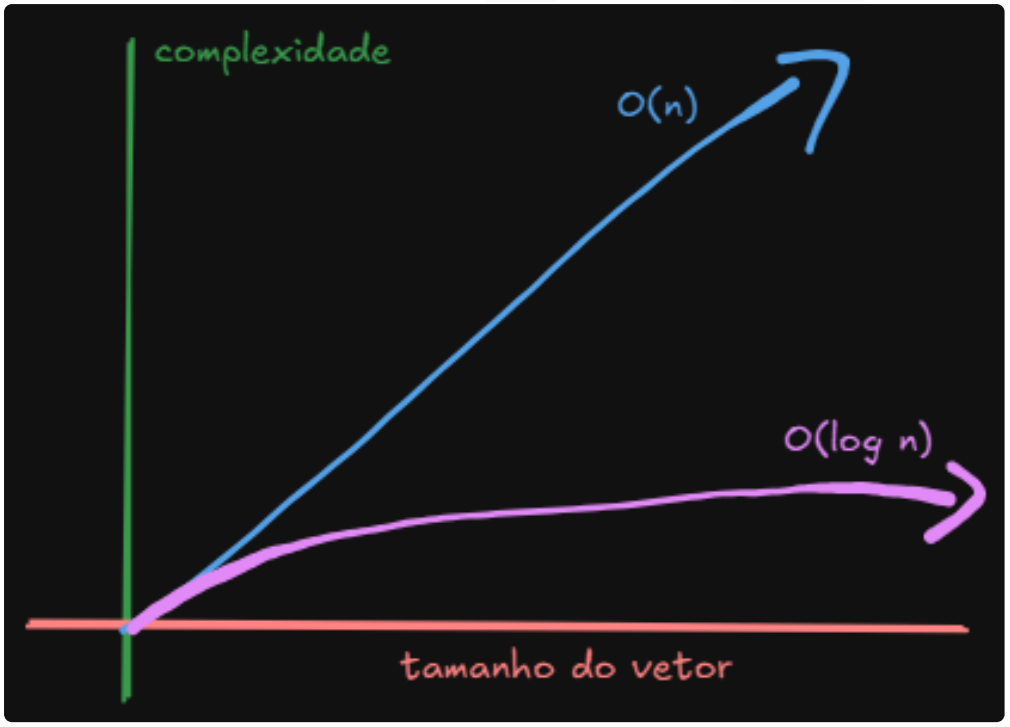
Algoritmos de Busca

1.Explique a diferença e aplicabilidade entre uma busca linear e uma busca binária.

- **Busca Linear:** É como procurar um livro em uma pilha desorganizada na mesa. Você pega um por um. É garantido que funciona, mas vai ser lento.
- **Busca Binária:** É como procurar um livro em uma estante perfeitamente organizada por ordem alfabética, podendo já ir direto para a seção correta e rapidamente afunila a busca. Mas, para isso, teve o trabalho de organizar a estante primeiro.

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

A busca linear tem complexidade `O(n)` e a binária `O(log n)`.



3.Implemente um algoritmo de busca binária que opere em vetores ordenados de modo crescente

Implementação da forma interativa

```go
// BinarySearch busca o alvo em uma lista ordenada e crescente
// Retorna o índice se encontrar, ou -1 se não encontrar
func BinarySearch(sortedData []int, target int) int {
    low := 0
    high := len(sortedData) - 1

    for low <= high {
        mid := low + (high-low)/2

        if sortedData[mid] == target {
            return mid //Encontrado
        } else if sortedData[mid] < target {
            low = mid + 1 //Olha pra metade direita
        } else {
            high = mid - 1 //Olha pra metade esquerda
        }
    }
```

```
    return -1 //Não encontrado
}
```

Implementação da forma recursiva

```
func BinarySearchRecursive(list []int, val int, start int, end int) int {
    // se a janela for inválida, o elemento não existe.
    if start > end {
        return -1
    }

    mid := start + (end-start)/2

    if list[mid] == val {
        return mid // Encontrado
    } else if list[mid] < val {
        // Chama a função com a nova janela de busca
        return BinarySearchRecursive(list, val, mid+1, end)
    } else {
        return BinarySearchRecursive(list, val, start, mid-1)
    }
}
```

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

Implementação da forma interativa

```
// BinarySearch busca o alvo em uma lista ordenada e decrescente
// Retorna o índice se encontrar, ou -1 se não encontrar.
func BinarySearch(sortedData []int, target int) int {
    low := 0
    high := len(sortedData) - 1

    for low <= high {
        mid := low + (high-low)/2

        if sortedData[mid] == target {
            return mid //Encontrado
        } else if sortedData[mid] < target {
            high = mid - 1 //Olha pra metade esquerda
        } else {
            low = mid + 1 //Olha pra metade direita
        }
    }
    return -1 //Não encontrado
}
```

Implementação da forma recursiva

```
// BinarySearchDescRecursive é uma única função que busca o alvo em uma lista
// ordenado e decrescente de forma recursiva
```

```go
func BinarySearchDescRecursive(list []int, val int, start int, end int) int {
    // para se a janela de busca for inválida
    if start > end {
        return -1
    }

    mid := start + (end-start)/2

    if list[mid] == val {
        return mid // Encontrado
    } else if list[mid] < val {
        // Se o elemento do meio é menor que o alvo, busca na metade esquerda
        return BinarySearchDescRecursive(list, val, start, mid-1)
    } else {
        // Se o elemento do meio é maior que o alvo, busca na metade direita
        return BinarySearchDescRecursive(list, val, mid+1, end)
    }
}
```

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

A Busca Linear sim, pois seu único "requisito" é a capacidade da lista ser percorrida, o que é natural. Porém a Busca Binária também requere que a lista esteja ordenada, pois caso não a chance de não encontrar o alvo ou fornecer um falso positivo é alta, devido a natureza de busca dela. Ou seja a Busca Binária num `ArrayList` ordenados é tranquilo, pois seus valores além de estarem sequenciais, ainda permitem o acesso aleatório, o que não é possível em `LinkedList` e `DoublyLinkedList`, pois é necessário percorrer a lista a partir do `head` para chegar num ponto específico. Em suma:

Método	`ArrayList`	`LinkedList` / `DoublyLinkedList`
Linear	Sim Custo `O(n)`	Custo `O(n)`
Binária	Sim Custo `O(log n)`	Custo `O(n log n)` Extremamente ineficiente

6.A partir do trecho de código em Python, avalie as afirmações a seguir:

- Afirmação I: Esta afirmação é incorreta, pois se a chamada de `return binary_search` for removida, o fluxo do programa não continuará a busca, pois a função simplesmente atribuiria um novo valor a `high` ou `low` e terminaria, sem entrar em um novo ciclo de busca. O resultado da busca seria alterado.
- Afirmação II: Esta afirmação está correta, pois transforma de forma correta uma função com recursão de cauda em uma versão iterativa com loop.
- Afirmação III: Esta afirmação é incorreta. O código sugerido apenas adiciona variáveis intermediárias antes de fazer exatamente a mesma chamada recursiva `return binary_search`, ou seja, só deixa o código menos otimizado.
- Afirmação IV: Essa sugestão usa um único `if/else` que define as variáveis `newlow` e `newhigh` e em seguida, faz uma única chamada `return binary_searc` usando essas variáveis. Olhando para os casos:
    - Se `lst[mid] > x` (o alvo está na metade esquerda), ele define os novos limites como `low` e `mid-1`. O que está certo;

- Senão (o alvo está na metade direita), ele define os novos limites como `mid+1` e `high`. O que também está certo;
  Então a lógica do algoritmo se mantém, apenas reescrita de uma forma diferente, então a afirmação está correta.

Dito isso a alternativa é a **letra c) II e IV**.

7.Observe o código abaixo escrito na linguagem C e a respeito das funções implementadas, avalie as afirmações a seguir.

- Afirmação I: Está correto.
- Afirmação II: Está incorreta, pois nesse cenário a função de busca binária será bem mais rápida que a busca linear, pois será um `O(log n)` contra um `O(n)`.
- Afirmação III: Está incorreta, pois a busca binária trabalha com recursividade e não iterativa;

Então a alternativa é a **letra a) I, apenas**.