

**Algoritmos e Estruturas de Dados**

## Mini-Teste 2: Algoritmos e Estruturas de Dados

**GRUPO 1 (5.0 Val.)**

1. Considere um jogo onde existem 2 roletas com  $n$  elementos cada. Os elementos das roletas são frutos identificados por uma string (“maçã”, “morango”, ...). Considere a classe **Jogo** apresentada:

```
class Jogo {
    list<string> roleta1;
    list<string> roleta2;
    stack<int> resultado;
    int valorJogada();
public:
    Jogo(int n) { ... } // coloca n valores aleatórios nas 2 roletas
    void jogada() { ... } // realiza uma jogada, rodando as roletas
    int valorJogo(int njog);
};
```

Uma jogada consiste em rodar as roletas de um valor aleatório, o que é implementado pelo método *jogada()*. Considere que este método já está implementado. Os resultados obtidos em cada jogada vão sendo guardados na pilha *resultado*.

- a. Implemente o método que determina o valor da última jogada. (2.5 Val.)

```
int Jogo::valorJogada()
```

Este método percorre as duas roletas verificando se existem frutos iguais na mesma posição, e retorna o número de pares encontrados.

- b. Implemente o método que determina o valor de um jogo após um número de jogadas especificado como argumento. (2.5 Val.)

```
int Jogo::valorJogo(int njog)
```

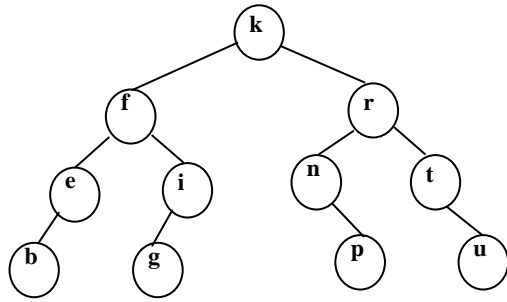
Este método realiza  $njog$  jogadas, colocando após cada jogada o respectivo valor na pilha *resultado* (se diferente de zero). O valor do jogo é calculado por leitura da pilha, como sendo a soma dos números de pares encontrados em todas as jogadas, pesados de um factor  $f$ , sendo que:

- $f = 1.0$  para a última jogada com pontuação;
- $f = 0.9$  se o número de pares obtido nesta jogada é superior ao obtido na jogada posterior;
- $f = 1.2$  se o número de pares obtido nesta jogada é inferior ao obtido na jogada posterior;
- $f = 1.0$  se o número de pares obtido nesta jogada é igual ao obtido na jogada posterior.

**GRUPO 2 (4.0 Val.)**

2. Considere a implementação de árvores binárias de pesquisa estudada nas aulas (classe *BST*).

- a. A figura seguinte representa uma árvore binária de pesquisa. Desenhe a estrutura da árvore após a remoção do elemento ' $k$ '. Explique detalhadamente o seu raciocínio. (2.0 Val.)



- b. Uma árvore diz-se completa quando todos os seus níveis estão completamente preenchidos, com possível exceção do último de um certo ponto para a direita. Escreva uma função em C++ que, dada uma árvore binária de pesquisa passada como argumento, determina se esta é completa ou não. **(2.0 Val.)**

```
template <class T> boolean eCompleta(BST<T> &arv)
```

Sugestão: use iterador por nível e uma fila auxiliar.

Nota: No caso de não conseguir implementar esta função, e só neste caso, pode implementar um novo método na classe **BST** estudada (esta solução deve ser vista como alternativa, e acarreta uma penalização na cotação da questão).

```
template <class T> boolean BST<T> veSeCompleta()
```

### GRUPO 3 (2.0 Val.)

3. Considere que para implementar uma tabela de dispersão (com dispersão aberta) utiliza um vector de tamanho 10 e função de dispersão  $h(x) = x \% 10$ .

Insira a seguinte sequência de valores: 3, 23, 5, 14, 4, 25, 15

Mostre o conteúdo do vector após cada inserção, usando resolução de colisões por sondagem quadrática (considere que **não** se efectua *rehash*). Explique detalhadamente o seu raciocínio. **(2.0 Val.)**

### GRUPO 4 (3.0 Val.)

4. Num sistema de uma escola para armazenamento e consulta de informação sobre alunos, a pesquisa de informação é realizada sobre o código do aluno (obter nome e email). A classe **Aluno**, apresentada a seguir de forma incompleta, contém a informação sobre um aluno:

```
class Aluno {
    string nome, email;
    int codigo;
public:
    Aluno (string &n, string &e, int c): nome(n), email(e), codigo(c)
    {};
};
```

A informação relativa aos alunos é guardada numa tabela de dispersão. Use a classe **hash\_set**.

```
typedef hash_set<Aluno, alunoFD, alunoIg> HashTaluno;
```

- a. Implemente a função de dispersão e a função de igualdade a ser usada pela tabela de dispersão. **(2.0 Val.)**
- b. Implemente a função *criaTabela*, que, dado um vector de alunos, cria e retorna uma tabela de dispersão com esses assinantes. **(1.0 Val.)**

```
HashTaluno criaTabela(const vector<Aluno> &alus)
```

Nome: \_\_\_\_\_

Turma: \_\_\_\_\_

**GRUPO 5 (6.0 Val.)**

Responda às seguintes questões, preenchendo a seguinte tabela com a opção correcta (em maiúsculas):

5.1	5.2	5.3	5.4	5.5	5.6

- 5.1) A operação de remoção de um elemento numa **lista (1.0/-0.4 Val.)**
- A) Possui complexidade temporal linear se a implementação da lista é baseada em vectores ou em listas ligadas;
  - B) Possui complexidade temporal linear se a implementação da lista é baseada em vectores, e constante se baseada em listas ligadas;
  - C) Possui complexidade temporal constante se a implementação da lista é baseada em vectores ou em listas ligadas;
  - D) Nenhuma das anteriores.
- 5.2) A operação de pesquisa de um elemento numa **lista: (1.0/-0.4 Val.)**
- A) Possui complexidade temporal linear se a implementação da lista é baseada em vectores, e constante se baseada em listas ligadas;
  - B) Possui complexidade temporal linear se a implementação da lista é baseada em vectores ou em listas ligadas;
  - C) Possui complexidade temporal constante se a implementação da lista é baseada em vectores ou em listas ligadas;
  - D) Nenhuma das anteriores.
- 5.3) A operação de inserção de um elemento numa **pilha: (1.0/-0.4 Val.)**
- A) Possui complexidade temporal constante se a implementação da pilha é baseada em vectores ou em listas ligadas;
  - B) Possui complexidade temporal linear se a implementação da pilha é baseada em vectores e constante se baseada em listas ligadas;
  - C) Possui complexidade temporal linear se a implementação da pilha é baseada em vectores ou em listas ligadas;
  - D) Nenhuma das anteriores.
- 5.4) Numa árvore binária: **(1.0/-0.4 Val.)**
- A) Qualquer nó tem tamanho superior ao do seu pai;
  - B) Qualquer nó tem tamanho superior a qualquer dos seus filhos;
  - C) Um nó não folha tem pelo menos um filho de tamanho superior;
  - D) Nenhuma das anteriores.
- 5.5) Conhecendo apenas a sequência de valores que representa a visita de uma qualquer árvore binária é possível reconstruir essa mesma árvore: **(1.0/-0.4 Val.)**
- A) Se a sequência de valores representa a visita em pré-ordem da árvore binária;
  - B) Se a sequência de valores representa a visita em pós-ordem da árvore binária;
  - C) Não é possível reconstruir a mesma árvore;
  - D) Nenhuma das anteriores.
- 5.6) Numa árvore binária de pesquisa, a ordem de inserção dos elementos: **(1.0/-0.4 Val.)**
- A) Não influencia a visita pós-ordem da árvore;
  - B) Não influencia a visita pré-ordem da árvore;
  - C) Não influencia a visita em-ordem da árvore;
  - D) Nenhuma das anteriores.