MAC0323 Algoritmos de Estruturas de Dados II

Prova 2 – 22 de maio de 2018

Nome: _____

Assinatura: _____

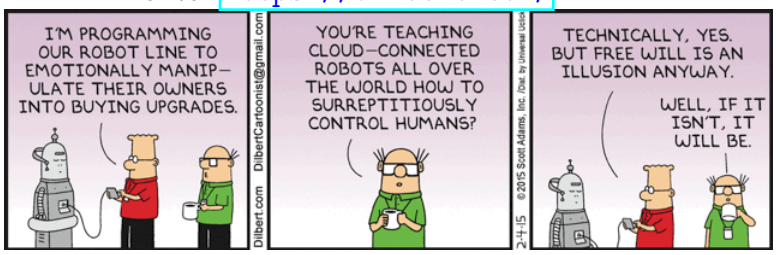
Nº USP: _____

Instruções:

1. Não destaque as folhas deste caderno. A prova pode ser feita a lápis.
2. A prova consta de 8 questões; Verifique antes de começar a prova se o seu caderno está completo.
3. Cuidado com a legibilidade.
4. Não é permitido o uso de folhas avulsas para rascunho, a consulta a livros, apontamentos, colegas ou equipamentos eletrônicos. Desligue o seu celular e qualquer equipamento que possa perturbar o andamento da prova.

DURAÇÃO DA PROVA: 100 minutos

Fonte: https://dilbert.com/



Questão	Valor	Nota
1	1,5	
2	1,0	
3	1,5	
4	1,5	
5	1,0	
6	1,5	
7	1,0	
8	1,0	
Total	10,0	

1. BST (1,5 pontos)

Considere o seguinte método para a classe `BST`:

```
public Key mystery(Key key) {
    Node best = mystery(root, key, null);
    if (best == null) return null;
    return best.key;
}

private Node mystery(Node x, Key key, Node best) {
    if (x == null) return best;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return mystery(x.left, key, x);
    else if (cmp > 0) return mystery(x.right, key, best);
    else return x;
}
```

(a) Suponha que `key` não é `null`. O que o método `mystery(key)` retorna?
Circule o rótulo da melhor resposta.

A. predecessor: a maior chave da `BST` menor que `key`;

B. chão ou piso (`floor()`): maior chave da `BST` que é menor que ou igual a `key`;

C. teto (`ceil()`): menor chave da `BST` que é maior que ou igual a `key`;

D. successor: a menor chave da `BST` maior que `key`;

E. `get()`: a chave `key` se ela está na `BST` e `null` em caso contrário;

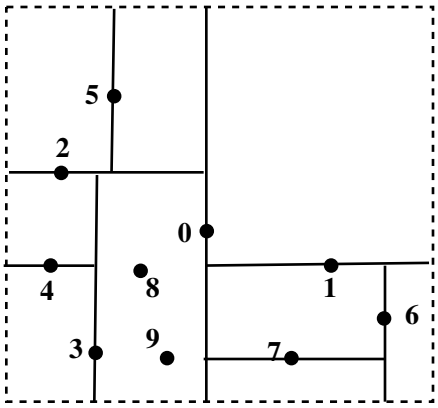
F. erro: `Null pointer exception` ou *loop infinito* para algumas entradas.

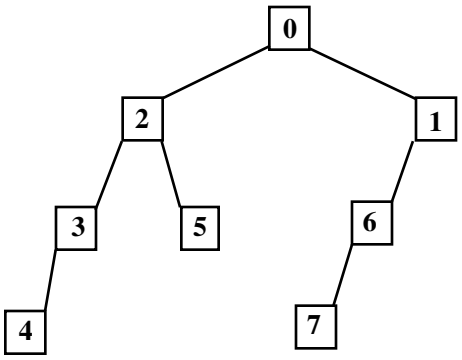
(b) Qual é o maior número de comparações feitas por `mystery()`? Suponha que a `BST` é balanceada.
Circule a melhor resposta.

1 $\quad$ $\lg n$ $\quad$ n $\quad$ n^2 $\quad$ 2^n $\quad$ ∞

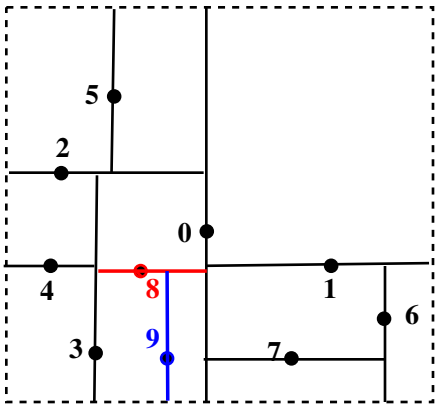
2. 2d-tree (1 ponto)

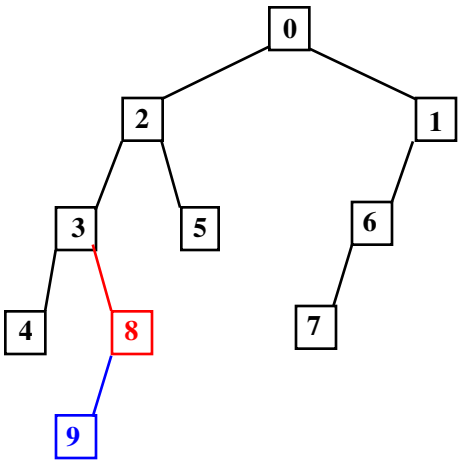
Abaixo está o resultado da inserção dos pontos 0 a 7, nessa ordem, em uma árvore `2d-tree` (`Kd-Trees`).





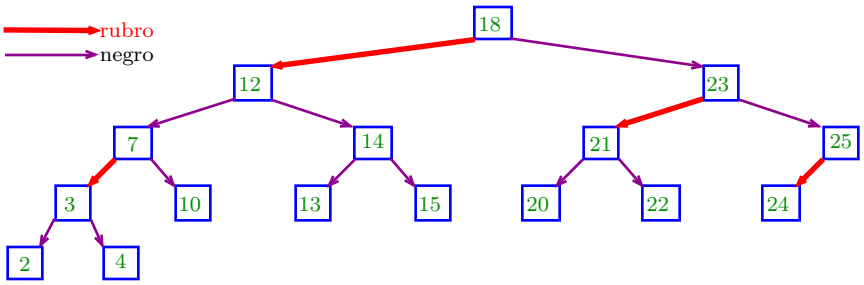
Insira os pontos 8 e 9, nessa ordem, na `2d-tree` e desenhe a árvore resultante.



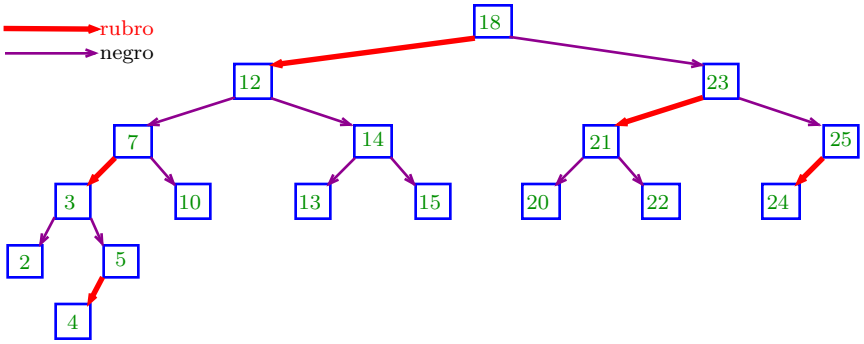


3. Inserção em ST rubro-negra (1,5 pontos)

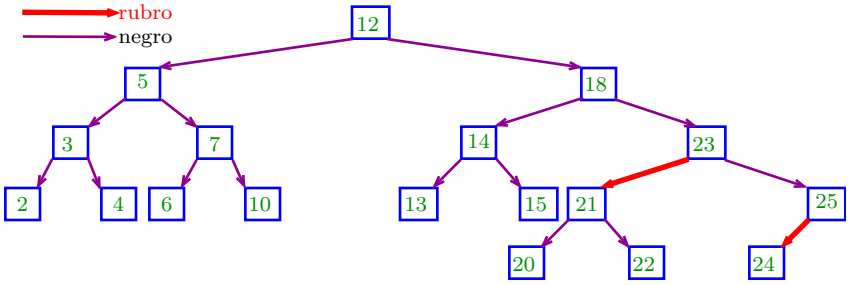
A figura abaixo mostra um `ST` (*left-leaning*) rubro-**negra**.



(a) Desenhe a `ST` rubro-**negra** resultante após a inserção da chave `5`.



(b) Desenhe a `ST` rubro-**negra** após a inserção da chave `6` na `ST` rubro-**negra** resultante do item (a).



4. Hashing (1,5 pontos)

(a) Suponha que as chaves `A`, `B`, `C`, `D`, `E`, `F`, `G` foram inseridas em uma `ST` implementa com hashing com sondagem linear (=*linear probing*). A `ST` implementada tem tamanho 7 e utiliza a seguinte função de hashing:

key	A	B	C	D	E	F	G
hash()	3	1	4	1	5	2	5

Circule o rótulo das tabelas a seguir que podem ser o resultado das inserções?

I.

0	1	2	3	4	5	6
G	B	D	F	A	C	E

II.

0	1	2	3	4	5	6
B	G	D	F	A	C	E

III.

0	1	2	3	4	5	6
E	G	F	A	B	C	D

(b) A seguir estão algumas afirmações envolvendo `ST`s implementadas com tabelas de hash. Nas afirmações `m` é o número de posições na tabela, `n` é o número de chaves na tabela e α é o fator de carga da tabela. Circule o rótulo das afirmações verdadeiras.

a. Podemos ter $\alpha > 1$ em uma tabela de hash com sondagem linear.

b. Podemos ter $\alpha < 1$ em uma tabela de hash com encadeamento (=*separate chaining*).

c. Sob a *hipótese do hashing uniforme* o comprimento médio das listas em hashing com encadeamento é `n/m`.

d. Tabelas de hash são convenientes para realizarmos eficientemente operações que envolvam ordem entre as chaves como `min()`, `max()`, `floor()`, `ceil()`, ...

e. Sob a *hipótese do hashing uniforme*, se $\alpha <$ `c` para alguma constante c, então o consumo de tempo médio de `get()` e `put()` é constante para hashing com encadeamento.

f. Em uma `ST` implementada com tabela de hash o consumo de tempo de `get()` e `put()` pode ser proporcional a `n`.

5. ST para strings (1 ponto)

A seguir estão descritas possíveis característica de uma `ST` para strings. Suponha que as strings são sobre um alfabeto com `R` símbolos. Nas características, `n` indica o número de strings e `w` o maior comprimento de uma string na `ST`.

a. A forma da `ST` depende da ordem em que as chaves são inseridas.

b. No pior caso, o número de chaves comparadas por `get()` é proporcional a `w`.

c. No pior caso, o número de chaves comparadas por `get()` é proporcional a `n`.

d. No pior caso, o número de chaves comparadas por `get()` é proporcional a `lg n`.

e. No pior caso, o número de chaves comparadas por `get()` é proporcional a `wn`.

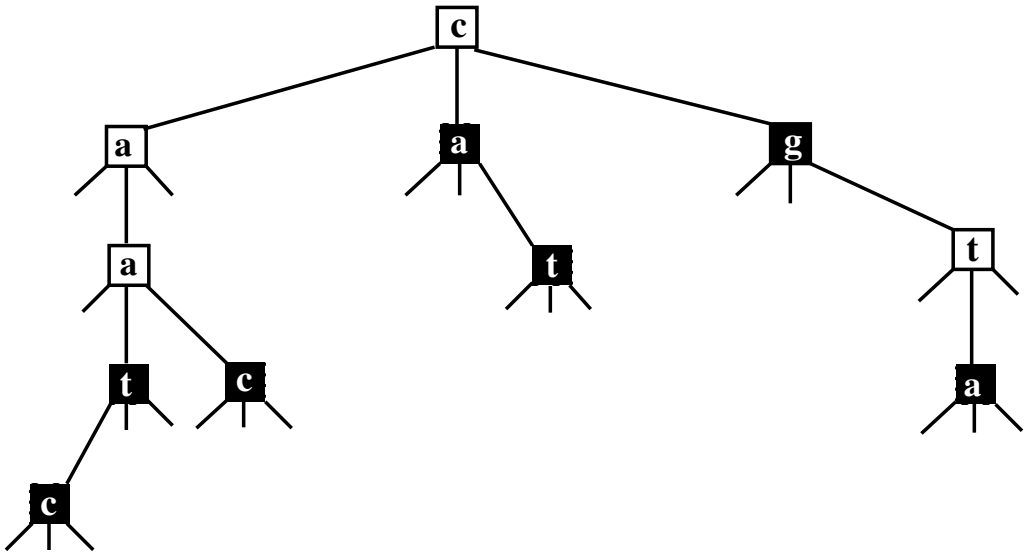
f. O espaço gasto, sem levar em consideração o espaço para as próprias strings, depende de `w`.

Para cada uma das `ST` abaixo associe a maior quantidade de características possível. Coloque os rótulos (letras) correspondente às características nas linhas das `ST`s.

`ST`	características
`BST`	`a c`
`ST` rubro-`negra`	`a d`
(`R`-way) trie	`b f`
`TST`	`a f`

6. Tries ternárias (1,5 pontos)

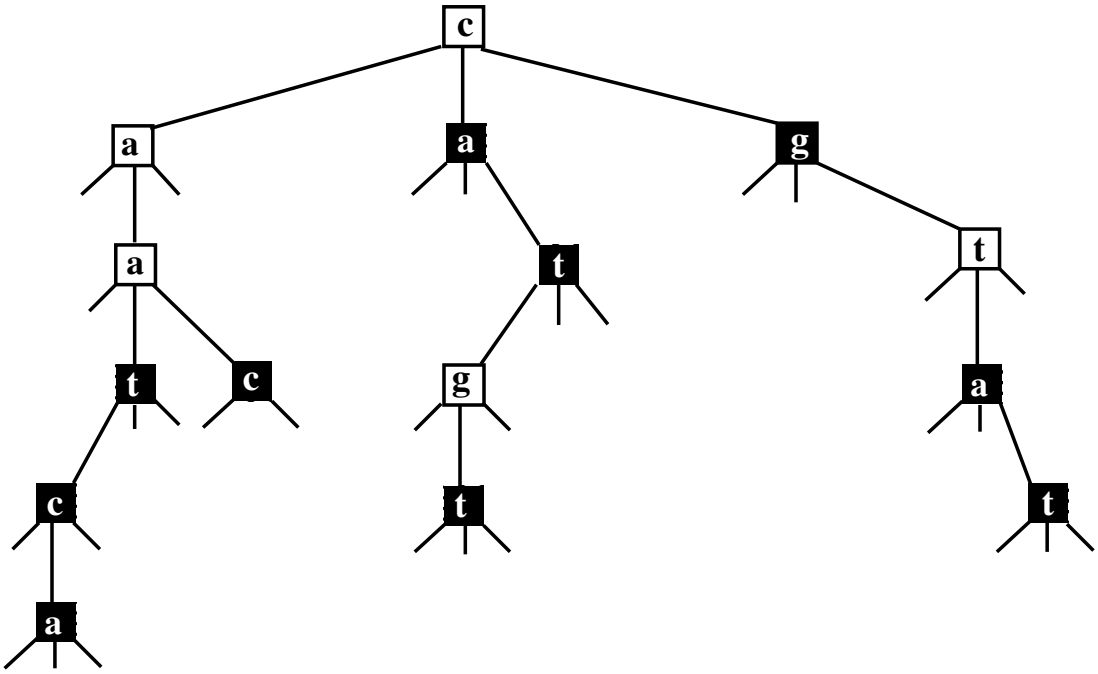
Considere a `TST` abaixo onde os nós em preto correspondem às strings na `TST` (`x.val != null`).



(a) Liste as strings na `TST` em ordem alfabética.

`aac  aat  ac  ca  ct  g  ta`

(b) Desenhe o resultado da inserção das strings `cgt`, `aaca` e `tt`, nessa ordem, na `TST`.



7. Algoritmo de Huffman (1 ponto)

Considere a string

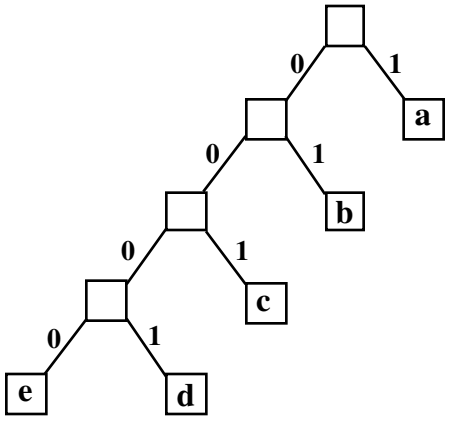
`a b a a b a c a b a a b a c d a b a a b a c a b a a b a c d e`

Com quantos bits o algoritmo de Huffman codifica essa string? Desconsidere os bits gastos para representar a tabela de codificação.

Na string a frequência de cada símbolo é dada pela tabela:

a	b	c	d	e
16	8	4	2	1

Árvore de Huffman



Código de Huffman

caractere	código
a	1
b	01
c	001
d	0001
e	0000

Número de bits para codificar a string:

$$1 \times 16 + 2 \times 8 + 3 \times 4 + 4 \times 2 + 4 \times 1 = 56 \ .$$

8. Análise de algoritmos (1 ponto)

Considere que temos três STs: `A`, `B` e `C`. A seguir `n` representa o número de itens na tabela. Suponha que na ST:

- `A` o número de comparações entre chaves feitas por `get()` e `put()` é **no pior caso** $2 \lg n$;
- `B`, começando com a ST vazia, o número **amortizado** de comparações entre chaves feitas por `get()` e `put()` é $2 \lg n$;
- `C` o número **médio** de comparações entre chaves feitas por `get()` e `put()` é $2 \lg n$.

Circule o rótulo das afirmações verdadeiras. Nas afirmações `N` representa o número de itens na tabela **depois das operações**.

a. Na ST `A`, é possível que alguma operação `get()` ou `put()` faça **mais que** $2 \lg N$ comparações entre chaves.

b. Na ST `A`, é possível que alguma sequência de `m` operações `get()` ou `put()` faça **menos que** $2m \lg N$ comparações entre chaves.

c. Na ST `A`, é possível que alguma sequência de `m` operações `get()` ou `put()` faça **mais que** $2m \lg N$ comparações entre chaves.

d. Na ST `B`, é possível que alguma operação `get()` ou `put()` faça **mais que** $2 \lg N$ comparações entre chaves.

e. Na ST `B`, é possível que, começando com a ST vazia, alguma sequência de `m` operações `get()` ou `put()` faça **menos que** $2m \lg N$ comparações entre chaves.

f. Na ST `B`, é possível que, começando com a ST vazia, alguma sequência de `m` operações `get()` ou `put()` faça **mais que** $2m \lg N$ comparações entre chaves.

g. Na ST `C`, é possível que alguma operação `get()` ou `put()` faça **mais que** $2 \lg N$ comparações entre chaves.

h. Na ST `C`, é possível que alguma sequência de `m` operações `get()` ou `put()` faça **menos que** $2m \lg N$ comparações entre chaves.

i. Na ST `C`, é possível que alguma sequência de `m` operações `get()` ou `put()` faça **mais que** $2m \lg N$ comparações entre chaves.