MAC0323 Algoritmos de Estruturas de Dados II

Prova 2 – 14 de maio de 2019

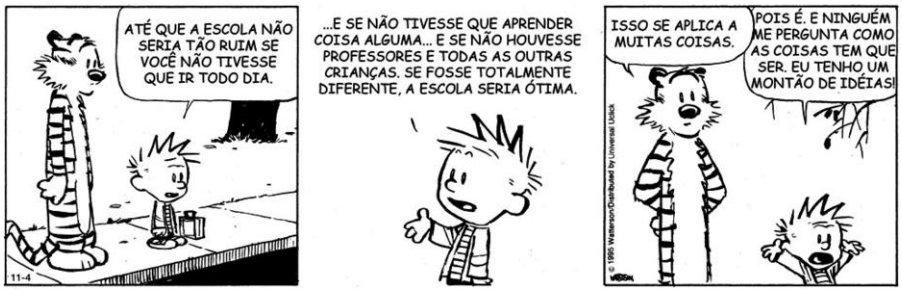
Nome: _____

Assinatura: _____

Nº USP: _____

Instruções:

1. Não destaque as folhas deste caderno. A prova pode ser feita a lápis.
2. A prova consta de 10 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

Questão	Valor	Nota
1	1,0	
2	1,0	
3	1,0	
4	1,0	
5	1,0	
6	1,0	
7	1,0	
8	1,0	
9	1,0	
10	1,0	
Total	10,0	

1. Memória e estruturas de dados (1 ponto)

Suponha que você implementou um árvore rubro-negra usando a seguinte representação.

```
public class RedBlackBST<Key extends Comparable<Key>, Value> {
   private Node root; // raiz da BST
   private int N; // numero de pares key-value na ST

   private class Node {
        private Key key; // chave
        private Value value; // valor
        private Node left; // filho(a) esquerdo(a)
        private Node right; // filho(a) direto(a)
        private boolean color; // cor do link para o pai/mãe
        private int count; // número de nós na subárvore que tem esse nó como raiz
   }
   [...]
}
```

Sabendo que tipicamente um `int` ocupa 4 bytes, um `boolean` ocupa 1 byte e variáveis de referência ocupam 8 bytes, qual é a quantidade de memória ocupada por um objeto `RedBlackBST` para armazenar `n` pares `key-value`?

O espaço gasto pelas chaves e pelos valores deve ser desconsiderado já que desconhecemos os valores.

Resposta: _____bytes

2. Codificação de BTs (1 ponto)

Um árvore binária em que todos os nós internos têm dois filhos pode ser codificada através de uma sequência de bits. Para isso, basta percorrer a árvore em **pré-ordem** e enviar para saída um bit `0` sempre que um nó interno é visitado e um bit `1` smpre que uma folha é visitada.
Desenhe a árvore binária codificada pela sequência de bits

```
0 0 0 1 0 1 1 0 0 1 1 1 0 1 1
```

3. BST (1 ponto)

Suponha que você implementou um BST usando a seguinte representação.

```
public class BST<Key extends Comparable<Key>, Value> {
   private Node root; // raiz da BST
   private int n; // numero de pares key-value na ST
   private class Node {
        private Key key; // chave
        private Value val; // valor
        private Node left; // filho(a) esquerdo(a)
        private Node right; // filho(a) direto(a)
        public Node(Key key, Value val) {
            this.key = key;
            this.val = val;
        }
   }
}
```

Escreva um construtor para essa classe BST de assinatura

```
public BST(Key[] keys, Value[] vals)
```

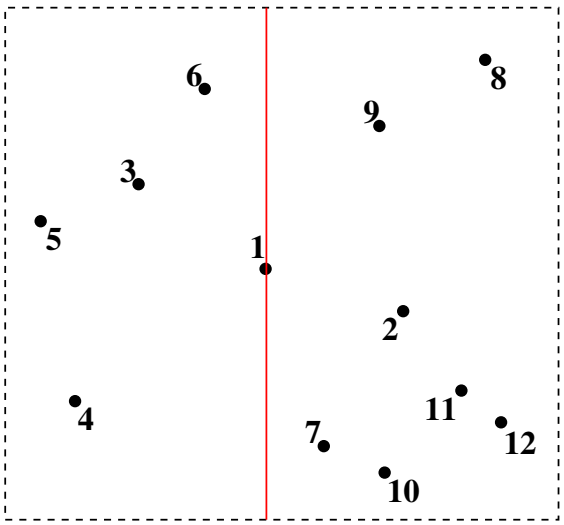
que recebe um vetor de chaves `keys` **ordenado** e constrói uma BST que seja equivalente à busca binária em vetor ordenado no seguinte sentido: a sequência de comparações produzida por um `get()` na BST é igual à sequência de comparações que seria feita pela busca binária no vetor `keys`.

(b) Qual é o consumo de tempo do seu construtor em função do número `n` de pares `key-val`?

Resposta: _____

4. 2d-tree (1 ponto)

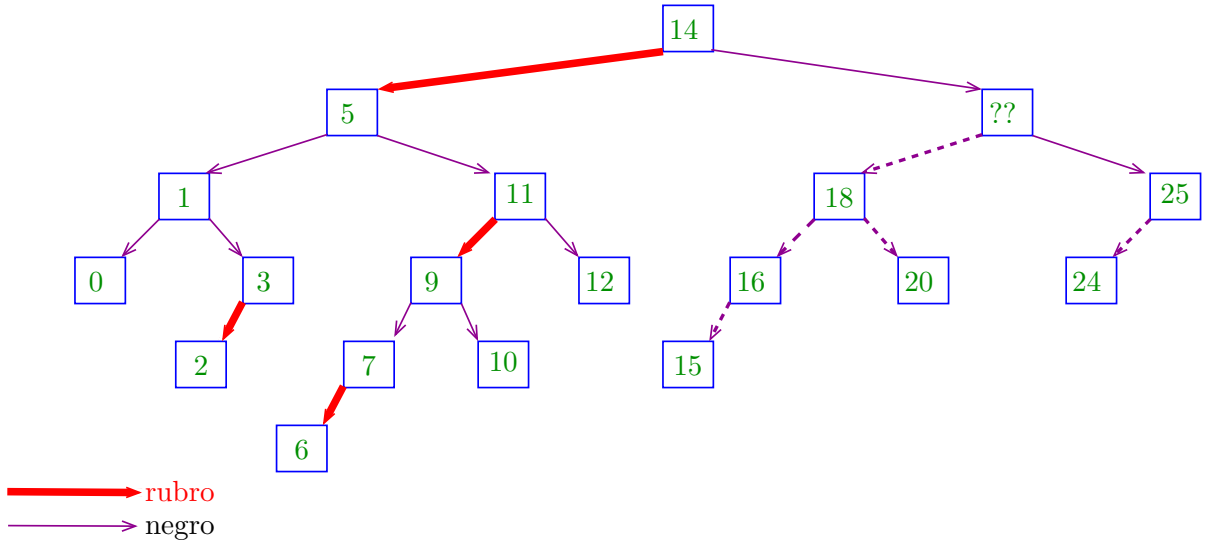
Na figura abaixo os números próximos aos pontos indicam a ordem em que foram inseridos em uma `2d-tree` (`Kd-Trees`).



Desenhe a árvore `2d-tree` resultante dessas inserções. No desenho, para cada nó da árvore, indique se ele está sobre uma linha horizontal ou vertical.

5. BST rubro-negra (1 ponto)

Considere a seguinte BST rubro-**negra** (*left-leanning*) em que as cores de alguns links e o valor de uma das chaves estão omitidos.



(a) Quais dos números inteiros a seguir podem ser a chave `'??'`?

`0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 11 16 17 18 19 20 21 22 23 24 25`

(b) Circule os links que são **necessariamente** rubros?

`??->18` `18->16` `18->20` `16->15` `25->24`

(c) Quantas operações `rotateLeft()`, `rotateRight()` e `flipColors()` serão realizadas para inserir na árvore acima a chave `-1` (operação `put(-1, val)`)?
E para inserir na árvore acima (**original**) a chave **4**?
E para inserir na árvore acima (**original**) a chave **8**?

operações	`-1`	`4`	`8`
`rotateLeft()`			
`rotateRight()`			
`flipColors()`			

6. Hashing com sondagem linear (1 ponto)

Considere a seguinte tabela com os valores da função de hashing para as chaves `A`, `B`, `C`, `D`, `E`, `F`, `G`.

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

(a) Suponha que as chaves foram inseridas na ordem `A`, `B`, `C`, `D`, `E`, `F`, `G` em uma tabela de hashing **com sondagem linear** (=*linear probing*) de tamanho 7, inicialmente vazia e sem redimensionamento. Qual o conteúdo da tabela de hashing?

0	1	2	3	4	5	6

(b) Circule o rótulo das tabelas a seguir que podem resultar das inserções das chaves, **em alguma ordem**, em uma tabela de hashing **com sondagem linear** de tamanho 7, inicialmente vazia e sem redimensionamento.

I.

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

II.

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

III.

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

(c) A seguir estão algumas afirmações envolvendo `ST`s implementadas com hashing **com sondagem linear**. 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 $\alpha > 1$.

b. Podemos ter $\alpha < 1$.

c. São convenientes para realizarmos eficientemente operações que envolvam ordem entre as chaves como `min()`, `max()`, `floor()`, `ceil()`, ...

d. 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.

e. O consumo de tempo de `get()` e `put()` pode ser proporcional a `n`.

7. Hashing com encadeamento (1 ponto)

Considere uma `ST` implementada por tabela de hash **com encadeamento** (=*separate chaining*). Suponha que a tabela tem `m` = 7 posições e que utiliza a função de hashing `hash(key) = key % m`.

(a) Desenhe o estado da `ST` após inserção, **nessa ordem**, das chaves `22`, `1`, `13`, `11`, `24`, `33`, `18`, `42` e `31`.

(b) Qual o fator de carga da tabela que você obteve?

Resposta: _____

(c) A seguir estão algumas afirmações envolvendo `ST`s implementadas com hashing **com encadeamento**. 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 $\alpha > 1$.

b. Podemos ter $\alpha < 1$.

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

d. 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.

f. O consumo de tempo de `get()` e `put()` pode ser proporcional a `n`.

8. 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. `ST` é uma árvore de busca.

b. A estrutura da `ST` não depende da ordem em que as chaves são inseridas.

c. A operação `get(key)` examina um a um os símbolos de `key`.

d. A `ST` é utilizada para encontrar todas as chaves que têm um dado prefixo.

e. A operação `get(key)` visita no máximo `1 + key.length()` nós.

f. A operação `get(key)` pode visitar até `n` nós.

g. A operação `get(key)` visita no máximo `2 lg n` nós.

h. A operação `get(key)` pode visitar até `Wn` nós.

i. O espaço gasto pelos *links* (=variáveis de referência) da `ST` depende de `n`.

j. O espaço gasto pelos *links* da `ST` depende de `w`.

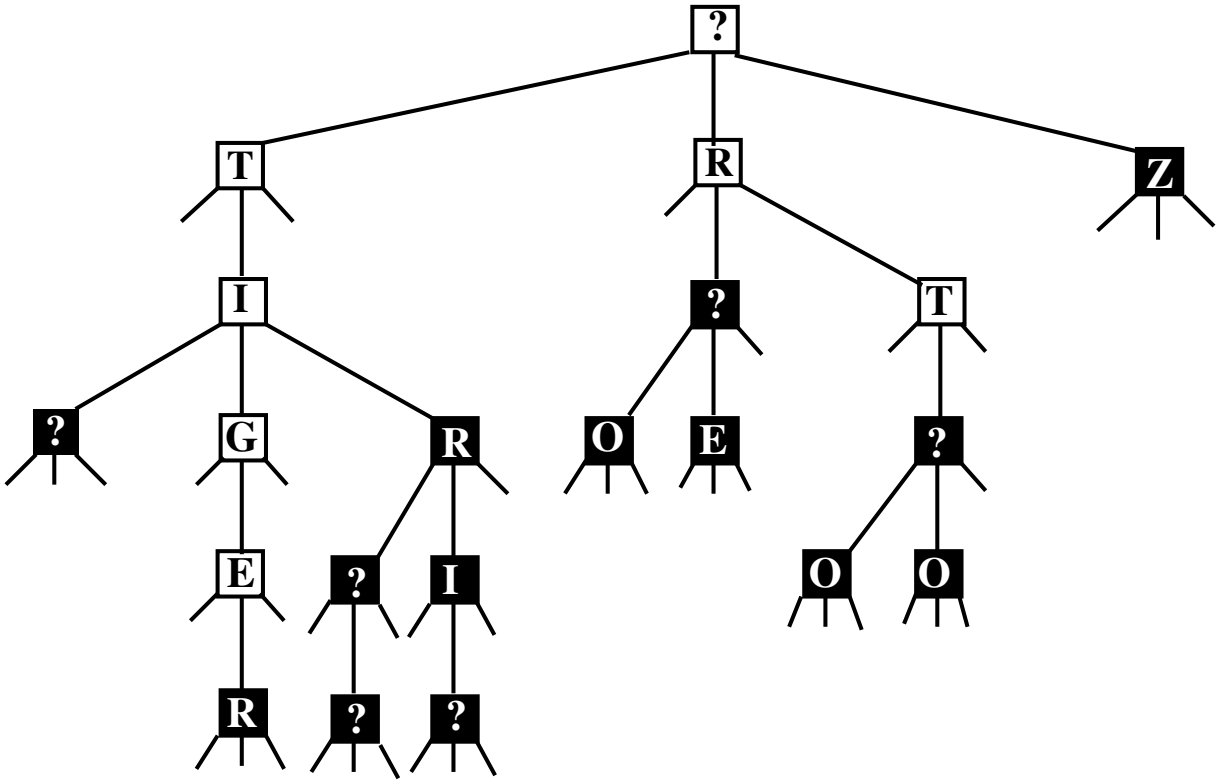
k. O espaço gasto pelos *links* da `ST` depende de `R`.

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`	
`ST` rubro-`negra`	
(`R`-way) Trie	
`TST`	

9. Tries ternárias (1 ponto)

Considere a `TST` abaixo sobre o alfabeto `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`. Os nós em preto correspondem às strings na `TST` (`x.val != null`). Nessa `TST` cada `'?'` representa um **símbolo qualquer** do alfabeto que **respeite as condições** da árvore ser uma `TST`.



Circule as strings abaixo que **são ou podem ser** chaves dessa `TST`.

`TIGER` `TILE` `TO` `TOO` `TREE` `TRIE` `TRUE` `TWO` `URGE`

10. Algoritmo de Huffman (1 ponto)

Em uma string formada por 36 caracteres sobre o alfabeto `"ABCDEF"` a frequência de cada símbolo é dada pela tabela:

A	B	C	D	E	F
3	4	8	5	6	10

(a) Desenhe uma correspondente trie de códigos do algoritmo de Huffman.

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

Resposta: _____bits

(c) Quais dos códigos a seguir é **livre de prefixos**?

símbolo	código 1	código 2	código 3	código 4	código 5
A	110	011	011	1110	100
B	111	010	010	1111	101
C	10	00	00	00	01
D	010	110	101	110	110
E	011	001	100	11	111
F	00	10	11	01	00

Resposta: