MAC0323 Algoritmos de Estruturas de Dados II

Primeira Prova – 02 de abril de 2019

Nome: _____

Assinatura: _____

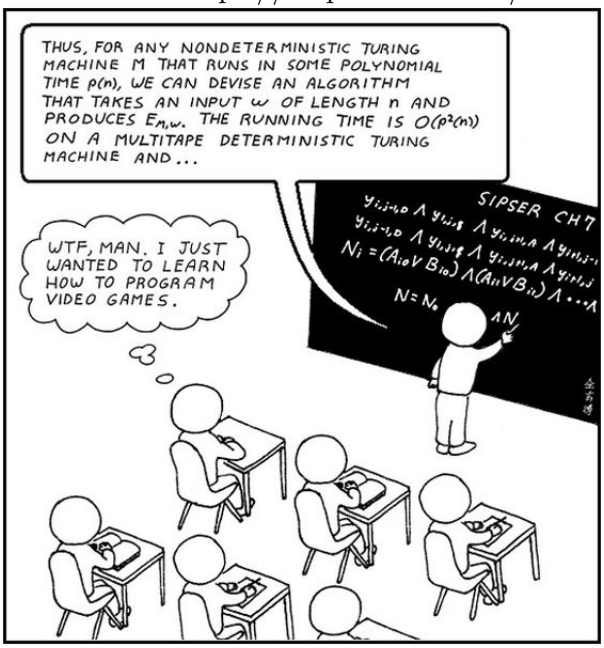
Nº USP: _____

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

Fonte: https://br.pinterest.com/



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. Estruturas de dados (vale 1,0 ponto)

A classe `StringBuilder` do Java representa uma sequência de caracteres mutáveis. Suponha que a implementação dessa classe utilize um **vetor redimensionável** (dobra o vetor quando esta cheio e diminui pela metade quando menos de um quarto está ocupado). A classe mantém uma variável de intância `n` para contar o número de caracteres na string e uma outra variável de instância `a[]` para armazenar a sequência de caracteres.

```
public class StringBuilder {
   private int n; // número de caracteres na sequência
   private char[] a; {// caracteres da sequência a[0], a[1],...
   [...]
}
```

(a) Sabendo que tipicamente um `int` ocupa 4 bytes e um `char` ocupa 2 bytes, qual é a quantidade de memória ocupada por um objeto `StringBuilder` para armazenar uma sequência de `n` caracteres?

melhor caso: $2n + 4$ (quando `a[]` está cheio) bytes

pior caso: $8n + 4$ (quando `a[]` está 1/4 cheio) bytes

(b) Qual o consumo de tempo **amortizado** de cada uma das operação a seguir?
Escreva a melhor resposta no espaço fornecido, usando uma das seguintes possibilidades.

$1 \qquad \lg n \qquad \sqrt{n} \qquad n \qquad n \log n \qquad n^2$

`charAt(int i)`	retorna o `i`-ésimo caractere na sequência	1
`deleteCharAt(int i)`	remove o `i`-ésimo caractere da sequência	n
`append(char c)`	coloca `c` no final da sequência	1
`set(int i, char c)`	troca o `i`-ésimo caractere por `c`	1

2. Union find (vale 1,0 ponto)

Circule as letras correspondente a configuração de vetores `id[]` (=`pai[]`) que **não podem** ocorrer durante a execução de uma sequência de operações `find()` e `union()` que utilizam o algoritmo *weighted quick union*.

	i:	0	1	2	3	4	5	6	7	8	9
A.	id[i]:	3	3	6	9	3	6	3	4	1	9
B.	id[i]:	9	0	0	0	0	0	9	9	9	9
C.	id[i]:	2	1	1	1	1	1	1	2	1	7
D.	id[i]:	8	0	4	0	0	4	0	4	2	0
E.	id[i]:	4	1	8	2	1	5	1	1	4	5

Resposta: A, B, D, e E

A. o número de nós na árvore com raiz 9 (pai de 3) é menor que duas vezes o número de nós na subárvore com raiz 3.

B. os nós 1, 2, 3, 4, e 5 só podem ter se ligado a 0 quando este era raiz de uma árvore; portanto, 0 nunca poderia se ligar a 9 pois é raiz de uma árvore com mais nós.

C. a sequência de `union()` produzirá `id[]`: `2-0 1-8 7-9 0-9 8-5 4-1 1-9 3-8 5-6`.

D. o vetor `id[]` contém um ciclo: $8 \to 2 \to 4 \to 0 \to 8$.

E. a altura da árvore com raiz 1 é 3 e contém $7 < 2^3$ nós.

3. Análise de algoritmos (vale 1,0 ponto)

Para cada função abaixo à esquerda, forneça a opção que melhor corresponde ao consumo de tempo da função em notação assintótica, segundo a lista fornecida à direita.

[B]
```
public static int f1(int N) {
    int x = 0;
    for (int i; i < N; i++)
        x += 1;
    return x;
}
```

[G]
```
public static int f2(int N, int R) {
    int x = 0;
    for (int i = 0; i < R; i++)
        x += f1(i);
    return x;
}
```

[L]
```
public static int f3(int N, int R) {
    int x = 0;
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < N; j++)
            x += f1(j);
    }
    return x;
}
```

[D]
```
public static int f4(int N, int R) {
    int x = 0;
    for (int i = 0; i < N; i++) {
        int j = 1;
        while (j <= R) {
            x += 1;
            j += j;
        }
    }
    return x;
}
```

[F]
```
public static int f5 (int N, int R) {
    int x = 0;
    for (int i = 0; i < N; i++) {
        int j = 1;
        while (j <= R) {
            x += f1(j);
            j += j;
        }
    }
    return x;
}
```

A. O(R)

B. O(N)

C. O(N + R)

D. O(N log R)

E. O(R log N)

F. O(N R)

G. $O(R^2)$

H. $O(N^2)$

I. O(N R log N)

J. O(N R log R)

K. $O(N R^2)$

L. $O(R N^2)$

M. $O(R^3)$

N. $O(N^3)$

4. Construção de heaps (vale 1,0 ponto)

(a) No preprocessamento do `Heapsort` uma lista dada é transformada em um **max-heap**. Considere a seguinte lista em que a parte relevante é a partir do índice 1.

—	-1	77	99	33	66	88	55	22	11	44
0	1	2	3	4	5	6	7	8	9	10

Mostre a seguir o resultado desse preprocessamento aplicado na lista acima.

—	99	77	88	33	66	-1	55	22	11	44
0	1	2	3	4	5	6	7	8	9	10

(b) Qual o consumo de tempo desse preprocessamento para construir um max-heap com `n` itens? Na sua resposta utilize a notação assintótica O(...).

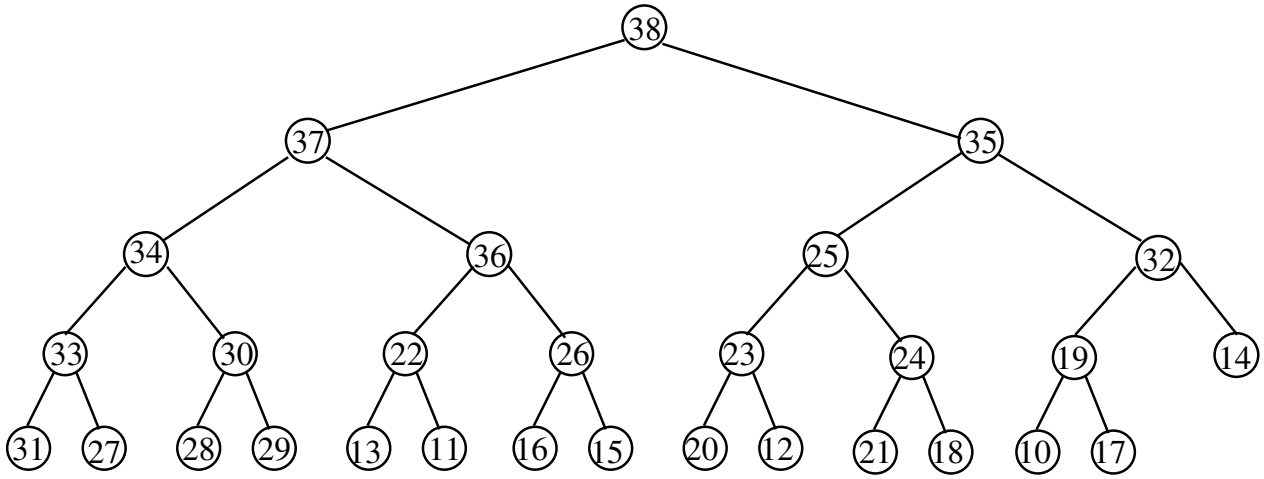
Resposta: O(`n`)

(c) Considere o algoritmo *online* para construir um max-heap, ou seja, os itens são fornecidos ao algoritmo um após o outro e a operação básica par construir-se o max-heap é `insert()`. Qual o consumo de tempo desse algoritmo para construir um max-heap com `n` itens? Na sua resposta utilize a notação assintótica O(...).

Resposta: O(`n` lg `n`)

5. Heaps binários (vale 1,0 ponto)

Considere o seguinte max-heap



(a) Suponha que a última operação executada no max-heap acima foi `insert(x)`.
Circule todos os possíveis valores para `x`.

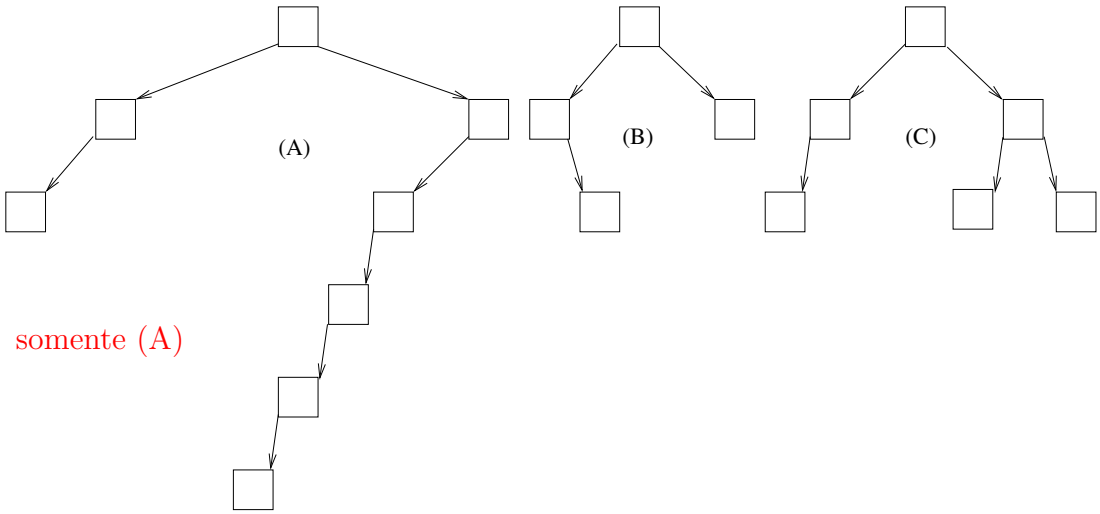
10	11	12	13	14	15	16	_17_	18	_19_
20	21	22	23	24	25	26	27	28	29
30	31	_32_	33	34	_35_	36	37	_38_	39

(b) Considere a execução da operação `delMax()` no max-heap acima.
Circule todos os valores envolvidos em alguma comparação.

10	11	12	13	14	_15_	_16_	_17_	18	19
20	21	_22_	23	24	25	_26_	27	28	29
30	31	32	33	_34_	_35_	_36_	_37_	38	39

6. Leftist heaps (vale 1,0 ponto)

(a) Indique quais das árvores abaixo são esquerdistas (*leftist*)? Justifique a sua resposta. Você pode rabiscar sobre as figura para justificar.

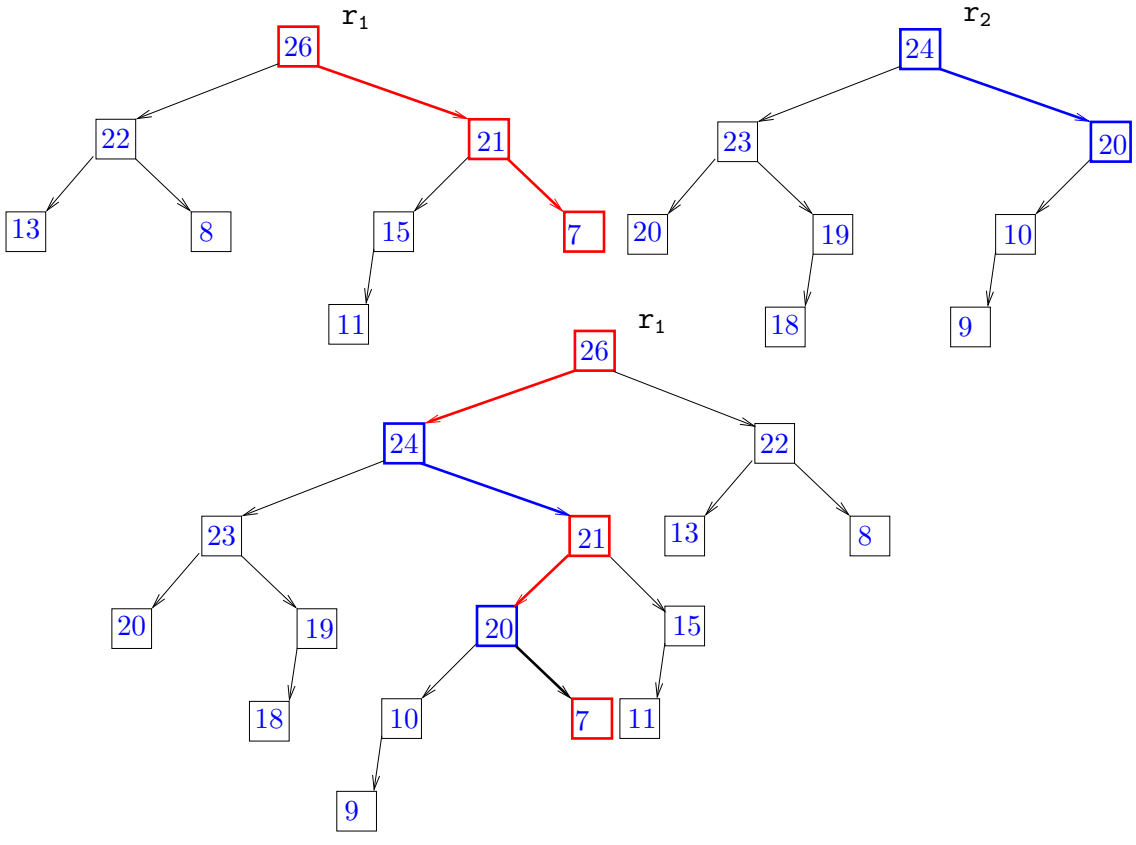


Resposta: somente (A)

(b) Em um **max-heap esquerdista** com `n` itens, qual é o consumo de tempo de `delMax()`? E o consumo de tempo de `insert()`? Nas respostas utilize notação assintótica.

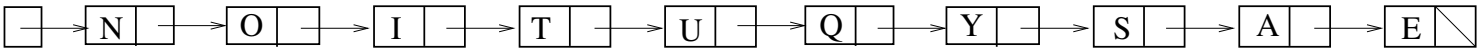
`delmax()`: $O(\lg n)$ no pior caso `insert()`: $O(\lg n)$ no pior caso

(c) Desenhe o heap esquerdista resultante da **união** (=`merge()`) do dois heaps esquerdistas abaixo. Os valores nos nós representam as chaves do itens.



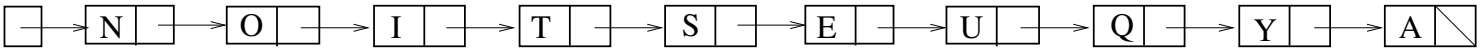
7. Tabelas de símbolos (vale 1,0 ponto)

(a) Simule o processo de inserir as chaves `E A S Y Q U E S T I O N` em uma tabela se símbolos implementada através de um **lista simplesmente ligada não ordenada**. Desenhe o estado final da lista ligada. Quantas comparações foram feitas?



Comparações: 55 comparações entre chaves

(b) Repita o exercício anterior usando a heurística *move to front*.



Comparações: 56 comparações entre chaves

(c) Qual das implementações de tabela de símbolos vistas até agora você utilizaria para uma aplicação que executa 10^3 operações `put()` e 10^6 operações `get()`, misturadas aleatoriamente? Justifique sua resposta.

Tinhamos visto até agora busca sequencial em vetores e listas, ordenadas e não ordenadas. Tinhamos também visto *skip list*, que tem consumo de tempo esperado O(log `n`) para `\put()` e `get()` e número esperado de links $n \lg n$.

Utilizaria ST em vetor ordenado: classe `BinarySearchST`.

O tempo gasto por `BinarySearchST` em cada operação `put()` é **no pior caso** proporcional ao número de pares `key-val` na ST (como nos exemplos acima). Assim, o tempo para construir a ST com `n` pares `key-val` é O($\mathtt{n}^2$) **no pior caso**.

O tempo gasto por `BinarySearchST` em cada operação `get()` é **no pior caso** proporcional ao logaritmo do número de pares `key-val` na tabela. Se temos `n` pares `key-val`, o tempos gasto por `get()` é O(lg `n`).

Em uma sequência de `n` operações `put()` e `m` operações `get()` o consumo de tempo **no pior caso** é O($\mathtt{n}^2 + \mathtt{m} \lg \mathtt{n}$. No caso em questão tempos apenas `n = 1` mil `put()` e `m = 1` milhão de `get()` o tempo da computação será dominado pelo tempo de `get()` e `BinarySearchST` se comportará bem nesse caso.

Se temos `n` pares `key-val` na tabela. O espaço gasto por `BinarySearchST` para cada vetor (redimensionável) é algo entre `n` e `2n` variáveis de referência (fora o espaço de *overhead* que é pouco). `BinarySearchST` possui um vetor `keys` e um vetor `vals` e cada variável de referência gasta 8 bytes, logo, o espaço gasto será algo entre `16000` e `32000` bytes.

Em termos de espaço, `BinarySearchST` de compara positivamente em relação as demais ST vistas.

8. Skip lists (vale 1,0 ponto)

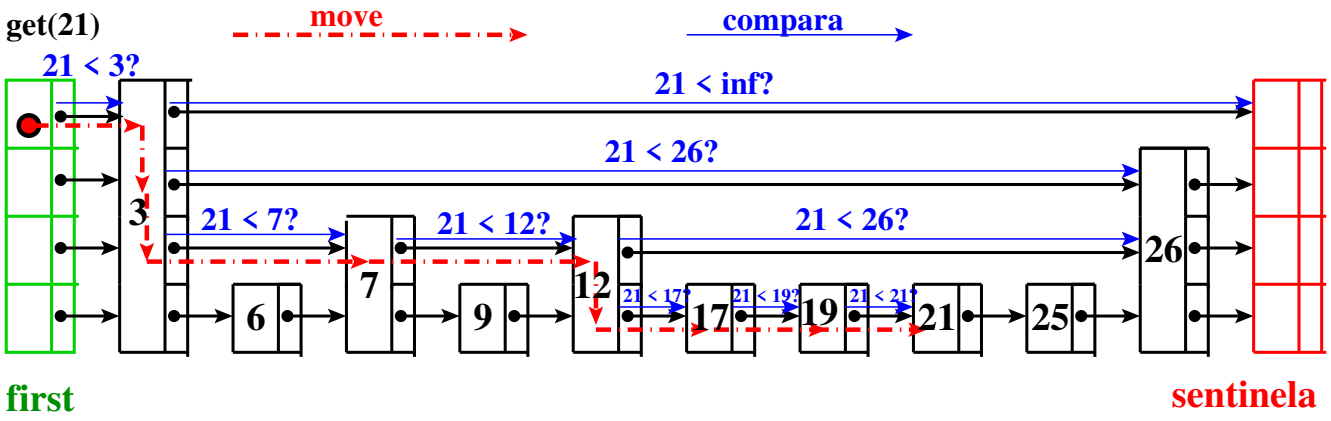
Desenhe a skip list resultante resultante da inclusão das chaves

```
9   3   6   7   19   12   17   21   26   25
```

em uma skip list inicialmente vazia. Suponha que a sequência de níveis devolvidos pelas chamadas sucessivas do método `randLevel()` seja

```
1   4   1   2   1   2   1   1   3   1
```

Na sua ilustração destaque com **linhas grossas** o caminho feito pela operação `get(21)`.



9. Árvores binárias (vale 1,0 ponto)

A classe BT a seguir que representa uma **árvore binária**.

```
public class BT<Item> {
   private Node root; // raiz da árvore
   private int n; // número de nós na árvore
   private class Node {
        private Item item; // o item
        private Node left; // subárvore esquerda
        private Node right; // subárvore direita
        public class Node(Item item) {
            this.item = item;
        }
   }
   [...]
}
```

(a) Escreva um método `altura()` que calcule a altura de uma BT.

```
public int altura() {
    return altura(root);
}
private int altura(Node x) {
    if (x == null) return -1;
    return 1 + Math.max(altura(x.left), altura(x.right));
}
```

(b) Escreva um método `posOrdem()` que imprima (`StdOut.println(item)`) os itens de uma BT em pós-ordem.

```
public void posOrdem() {
    posOrdem(root);
}
private void posOrdem(Node x) {
    if (x == null) return;
    posOrdem(x.left);
    posOrdem(x.right);
    StdOut.printl(x.item);
}
```

10. Análise de desempenho (vale 1,0 ponto)

Suponha que experimentalmente você observou os seguintes tempos de execução de três funções usando entradas com `n` itens.

função `a()`

`n`	tempo
1024	0.03 segundos
2048	0.09 segundos
4096	0.37 segundos
8192	1.70 segundos
16384	7.08 segundos
32768	28.54 segundos
65536	113.55 segundos
131072	493.08 segundos

função `b()`

`n`	tempo
1000000	0.06 segundos
2000000	0.11 segundos
3000000	0.16 segundos
6000000	0.32 segundos
8000000	0.43 segundos
10000000	0.55 segundos
15000000	0.82 segundos
20000000	1.08 segundos
25000000	1.37 segundos
30000000	1.70 segundos

função `c()`

`n`	tempo
10.000	1,2 segundos
40.000	2,1 segundos
160.000	3,9 segundos
640.000	7,9 segundos
2.560.000	16,0 segundos

Qual o consumo de tempo das funções (em segundos) em função de `n`?
Nas respostas a seguir utilize a notação assintótica O(...).

Qual o consumo de tempo da função `a()`? Resposta: $O(n^2)$

Qual o consumo de tempo da função `b()`? Resposta: $O(n)$

Qual o consumo de tempo da função `c()`? Resposta: $O(\sqrt{n})$