Aluno(a):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ matrícula:\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| 1a) | 2,0 |  |
| 2a) | 3,0 |  |
| 3a) | 3,0 |  |
| 4a) | 2,0 |  |
|  |  |  |

* A prova é individual e sem consulta.
  + **Qualquer tentativa de “cola” resultará na anulação da prova do aluno ou dos alunos envolvidos**.
  + Os aparelhos celulares deverão permanecer desligados durante toda a prova. **Aparelhos celulares ligados serão tratados como tentativa de “cola”**.
* A interpretação faz parte da questão.
  + **Não há perguntas durante a prova.**
  + Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
* O tempo de prova é 1:45 h.
* **Após o início da prova, não será possível sair e depois voltar à sala.**
* As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
* A prova pode ser feita a lápis.

**Questão 1** (2,0 pontos) Considere uma árvore *A* com *n* chaves.

a) (1,0 ponto) Suponha que *A* seja uma árvore AVL. Qual a complexidade no pior caso, expressa em função de *n*, do problema de determinar a altura de *A*? Explique a sua resposta.

b) (1,0 ponto) Suponha que *A* seja uma árvore-B de grau *m*. Qual a altura máxima *h* de *A*, expressa em função de *n* e *m*? Explique a sua resposta.

**Resposta**

(a) A altura de uma árvore AVL com *n* chaves (ou nós) é dada por *h* ≈ 1.44*log*(*n*) (ver notas de aula). Assim, a complexidade, no pior caso, do problema de determinar a altura de uma árvore AVL será *O*(*log*(*n*)).

(b) O número mínimo *k* de chaves de uma árvore-B de ordem *m* é *k* = 2 × ⎡*m*/2⎤*h* − 1. Assim, a altura máxima *h* de uma árvore-B de ordem *m* com *n* chaves será:

*n* = 2 × ⎡*m*/2⎤*h* − 1

*n* + 1 = 2 × ⎡*m*/2⎤*h*

**Questão 2** (3,0 pontos)

a) (1,0 ponto) Implemente em C uma função que calcula a altura de uma árvore AVL de forma otimizada (respostas não otimizadas não serão consideradas). A função deverá ter o seguinte protótipo:

int avl\_alt(AvlNode\* node)**;**

utilizar a estrutura:

typedef struct \_avl\_node AvlNode;

struct \_avl\_node {

void\* info;

int bf; /\* balance factor \*/

AvlNode\* parent;

AvlNode\* left;

AvlNode\* right;

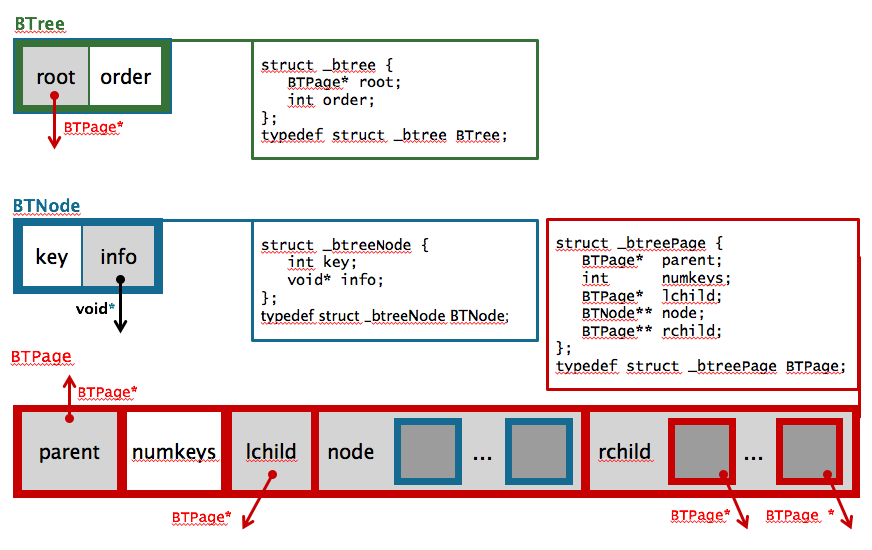
};

e retornar -1, se a árvore for vazia.

b) (0,5 ponto) Argumente porque a sua implementação está correta e é otimizada.

c) (0,5 ponto) Descreva em pseudo-código (ou seja, sem escrever código em C) uma função que calcula a altura de uma árvore-B de grau *m* de forma otimizada (respostas não otimizadas não serão consideradas). A função deverá usar as estruturas de dados da Figura 1 para representar uma árvore B, e retornar -1, se a árvore for vazia

d) (1,0 ponto) Argumente porque a sua implementação está correta e é otimizada.



**Figura 1.** Estruturas para representar uma árvore-B.

**Resposta**

a)

1. int avl\_alt(AvlNode\* node) {
2. int h=-1;
3. while(node!=NULL) {
4. node = (node->bf>0) ? node->right : node->left;
5. h++;
6. }
7. return h;
8. }

b) O teste da linha 4 força o loop a continuar descendo pelo filho de node cuja subárvore tem o maior ramo: se node->bf>0 será o filho à direita; se node->bf<0 será o filho à direita; se node->bf=0 será indiferente pois as duas subárvores tem o maior ramo de igual comprimento. Portanto, se a árvore for AVL, o algoritmo corretamente computará o número de arestas do maior ramo da árvore, que é a sua altura. O algoritmo é otimizado pois caminha apenas pelo maior ramo, ou seja, não visita nenhum nó desnecessariamente.

c)

1. ab\_altura:
2. **entrada**: um apontador para BTree
3. **saída**: a altura da árvore-B apontada por BTree
4. **inicio**
5. h = 0;
6. Page = BTree -> root;
7. **enquanto** Page não for nulo **faça**
8. **inicio**
9. Page = Page -> lchild;
10. h = h +1;
11. **fim**
12. **retorne** h
13. **fim**

d) O loop da linha 7 continua descendo pelo apontador do filho mais à esquerda do nó até chegar a uma folha, contando o número de passos. Ou seja, o algoritmo retorna o comprimento do ramo mais à esquerda da árvore. Como a árvore-B é balanceada, todos os ramos têm o mesmo comprimento, que é a altura da árvore. O algoritmo é otimizado pois caminha apenas por um ramo, ou seja, não visita nenhum nó desnecessariamente.

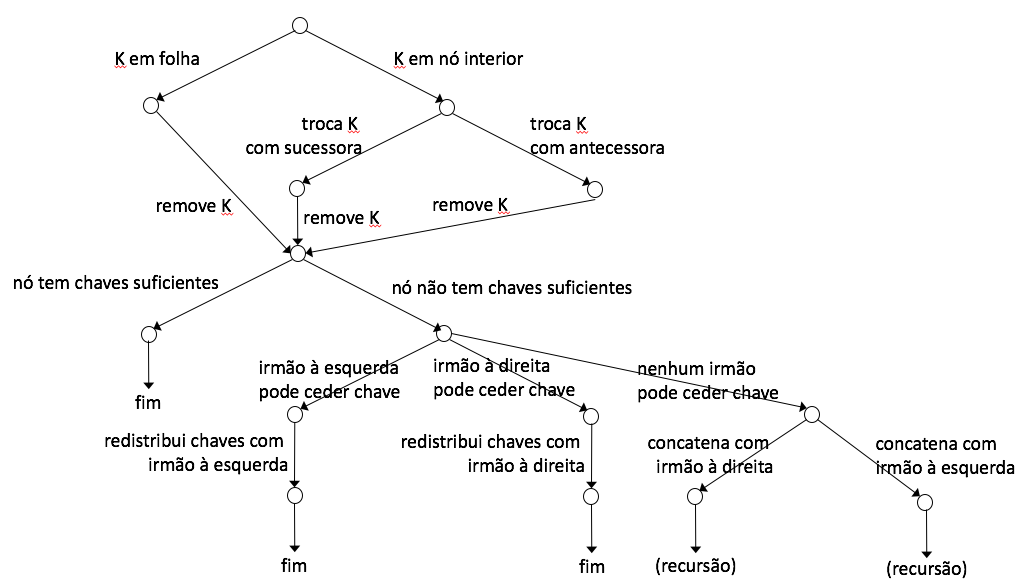
**Questão 3** (3,0 pontos) Seja *A* uma árvore-B de grau *m*.

* 1. (1,0 ponto) O método para inserção de uma nova chave *K* em *A*, apresentado nas notas de aula, é determinístico, ou seja, processa uma inserção sempre da mesma forma. Em que situação o algoritmo poderia ser modificado para operar de 2 formas diferentes, sem violar as regras de formação da árvore-B? Explique a sua resposta.
  2. (2,0 pontos) Descreva todos os casos do método para remoção de uma chave *K* em *A* apresentado nas notas de aula.

**Resposta**

a) Caso *m* seja impar, ao tentar inserir uma nova chave *K* em um nó *N* completamente cheio, o número de chaves será (*m*+1) e, portanto, par. Assim, não há claramente uma chave do meio. Podemos mover para o pai de *N* a chave na posição ((m+1)/2) ou ((m+1)/2+1) (começando a contar de 1).

b) (Ver notas de aula).



**Questão 4** (2,0 pontos) Considere uma tabela de dispersão estendida *T* tal que:

* Após aplicar a função de hash a uma chave, os bits são considerados da direita para a esquerda.
* Cada bloco pode conter 1 ou 2 chaves.
* Blocos vazios devem ser descartados.

a) (1,0 ponto) Assuma que *T* possui *n* chaves armazenadas. Qual o custo, no pior caso, para pesquisar uma chave *K* em *T* ? Explique sua resposta.

b) (1,0 ponto) Assuma que *T* está no estado descrito pela Figura 2. Mostre como a estrutura ficará após as remoções, sucessivamente, de *K41* e *K31*. Explique sua resposta, incluindo desenhos dos vários estados da estrutura.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =2* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  |  | *j1 =2* | |
| 2 |  |  |  |  | |
| 3 |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  |  | *j2 =2* | |
|  |  |  |  |  | |
|  |  |  |  |  |
|  |  |  | *K31* | *B3* |
|  |  |  |  | *j3 =2* | | |
|  |  |  |  |  | | |
|  |  |  |  |  |
|  |  |  | *K41* | *B4* |
|  |  |  |  | *j4 =2* | |
|  |  |  |  |  | |

**Figura 2.** Estado da tabela *T*.

**Resposta**

a) No pior caso, todas as *n* chaves possuem os mesmo bits finais, depois de aplicada a função de hash. Ou seja, *T* degenera em uma lista encadeada de buckets, cada um com 2 chaves. O custo de pesquisar uma chave será então proporcional a *n/2* (pois cada bucket contém 2 chaves no máximo). Ou seja, o custo, no pior caso, para pesquisar uma chave *K* em *T* será *O*(*n*).

b) Estado de *T* após a remoção de *K41*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =2* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  |  | *j1 =2* | |
| 2 |  |  |  |  | |
| 3 |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  |  | *j2 =1* | |
|  |  |  |  |  | |
|  |  |  |  |  |
|  |  |  | *K31* | *B3* |
|  |  |  |  | *j3 =2* | | |
|  |  |  |  |  | |

Estado de *T* após a remoção de *K31*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =2* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  |  | *j1 =1* | |
| 2 |  |  |  |  | |
| 3 |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  |  | *j2 =1* | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =1* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  |  | *j1 =1* | |
|  |  |  |  |  | |
|  |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  |  | *j2 =1* | |