Aluno(a):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Matrícula:\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| 1a) | 2,5 |  |
| 2a) | 2,5 |  |
| 3a) | 2,5 |  |
| 4a) | 2,5 |  |
|  | 10,0 |  |

**LEIA COM CUIDADO**

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

**Questão 1 (2,5 pontos)**.

**(a) (1,5 pontos)** Escreva uma rotina em C para inserção de chaves, definidas como números inteiros, em uma tabela de dispersão com endereçamento aberto e função de dispersão *h* definida como

onde

*K* = chave a ser inserida

*i* = 0, 1, 2, ... = número da tentativa de inserção da chave na tabela

*P* = constante para redispersão, em caso de colisão

*N* = tamanho da tabela de dispersão

Assuma que as posições são preenchidas, inicialmente, com o valor -1 e que a tabela seja declarada como:

#define N 24679; /\* tamanho da tabela de dispersão \*/

#define P 23; /\* coeficiente para rehash \*/

#define VAZIO (-1); /\* indica que a posição está desocupada \*/

int tabela[N]; /\* tabela de dispersão \*/

A função deve ter o seguinte protótipo:

int insere(int K)

e deve retornar:

-1 se a chave não puder ser inserida

0 se a chave já existe

1 se a chave for inserida corretamente

**(b) (0,5 ponto)** A escolha de *N* = 24.679 é uma boa escolha, supondo-se que o número de chaves não ultrapasse 22.000 chaves? Explique a sua resposta cuidadosamente.

**(c) (0,5 ponto)** A função de dispersão adotada é adequada para as escolhas de *N*=24.679 e *P*=23? Explique a sua resposta cuidadosamente.

***Resposta***

int insere(int K)

{

int i, j;

for (i = 0; i < N; i++)

{

j = (j + P\*i) % N; /\* Próxima tentativa \*/

if (tabela[j] == VAZIO) /\* Insere a chave \*/

{

tabela[j] = K;

return 1;

}

if (tabela[j] == K) /\* Chave já existe \*/

return 0;

}

return (-1);

}

**(b)** Note que *N* = 23 \* 29 \* 37 = 24.679. Portanto, *N* é um bom tamanho para uma tabela de dispersão com até cerca de 22.000 chaves pois 22.000 \* 1,1 < 24.679.

**(c)** Porém, as escolhas de *N*=24.679 e *P*=23 não são adequadas para a função de dispersão

pois *P* é um fator primo de *N*, o que pode forçar a inserção a falhar, mesmo que haja alguma posição vazia na tabela de dispersão.

**Questão 2** **(2.5 pontos)** Considere o problema de representar 2 subconjuntos de um grupo *G* de pessoas, definidos como:

*G1* = { *x* ∈ *G* / *x* é gestante }

*G2* = { *x* ∈ *G* / *x* vive em área infestada de “Aedes aegypti” }

Suponha que os subconjuntos estão representados utilizando vetores de bits:

struct \_bitvector {

int max;

int \*vector;

};

typedef struct \_bitvector BitVector;

**(a)** **(1,5 pontos)** Implemente em C uma rotina que receba como entrada *G1* e *G2* e devolva o subconjunto *H* de *G* definido como:

*H* = { *x* ∈ *G* / *x* é gestante e *x* não vive em área infestada de “Aedes aegypti” }

A rotina não deverá chamar as operações definidas em sala para a estrutura escolhida.

**(b) (0,5 ponto)** Qual a complexidade, no pior caso, da rotina implementada no item (a), medida em termos do número de elementos de cada subconjunto. Explique sua resposta.

**(c)** **(0,5 ponto)** Baseando-se na resposta do item (b), argumente porque a sua implementação é mais eficiente do que uma implementação da rotina do item (a) que utilize listas encadeadas para representar os conjuntos.

***Resposta***

(a)

BitVector\* bvH(BitVector\* G1, BitVector\* G2)

{

int i;

int min=(G1->max < G2->max)? G1->max : G2->max;

int num=(min-1)/sizeof(int)+1; /\* tamanho do menor vetor \*/

int tam=(G1->max-1)/sizeof(int)+1; /\* tamanho do vetor de H é igual

ao tamanho do vetor de G1 \*/

BitVector\* H = (BitVector\*)malloc(sizeof(BitVector));

H->max = G1->max;

H->vector = (int\*)malloc(tam\*sizeof(int));

for (i=0; i<num; i++) /\* computa H \*/

H->vector[i] = (G1->vector[i]) & ~(G2->vector[i]);

for (i=num; i<tam; i++) /\* copia o resto de G1 para H, \*/

H->vector[i] = G1->vector[i]; /\* se G1 for maior do que G2 \*/

return H;

}

(b) Em função dos for-loops, a rotina possui complexidade *O(n1)*, onde *n1* é o número de elementos do conjunto *G1*. Note que as operações bit-a-bit não influenciam a complexidade do pior caso.

(c) Se os conjuntos fossem implementados como listas encadeadas, a rotina do item (a) teria que percorrer a lista correspondente a *G2*, para cada elemento de *G1*. Esta rotina teria complexidade *O(n1*.*n2)*. Logo, a implementação utilizando vetores de bits é mais eficiente do que uma implementação utilizando listas encadeadas.

**Questão 3 (2,5 pontos)**. Considere a árvore B de ordem 5 (todo nó, exceto a raiz, tem nó mínimo 2 chaves e no máximo 4 chaves) da Figura 1.

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 140 170 230 236 242 245

**Figura 1**. Árvore-B da Questão 2.

Realize as seguintes operações, **utilizando sempre a árvore da Figura 1 como árvore inicial**. Indique os nós que sofrem modificações após cada operação, bem como a ocorrência de divisão, redistribuição ou concatenação:

1. **(0,5 ponto)** Inserção de 142.
2. **(1,0 ponto)** Remoção de 60.
3. **(1,0 ponto)** Remoção de 123 de duas maneiras diferentes.

***Resposta***

a) Inserção de 142.

Pesquise o nó onde a chave 142 deve ser inserida.

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 140 170 230 236 242 245

***142***

Divida o nó e insira a chave do meio (140) no pai.

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 **140** 142 170 230 236 243 245

100

53 77 123 140 200

10 40 60 70 80 90 110 113 130 138 142 170 230 236 243 245

b) Remoção de 60.

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 140 170 230 236 242 245

A chave 60 ocorre em uma folha e, portanto, pode ser removida.

100

53 77 123 200

10 40 70 80 90 110 113 130 138 140 170 230 236 242 245

Concatene nós, com transferência de chave do pai para o nó à esquerda:

100

77 123 200

10 40 53 70 80 90 110 113 130 138 140 170 230 236 242 245

Não pode haver redistribuição entre os dois nós no segundo nível pois o total de chaves do nó da direita já está no limite (2 chaves). Concatene então os nós do segundo nível com a raiz:

77 100 123 200

10 40 53 70 80 90 110 113 130 138 140 170 230 236 242 245

c) Remoção de 123:

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 140 170 230 236 242 245

A chave 123 não ocorre em uma folha.

* Alternativa 1: Troque 123 com a sua sucessora 130

100

53 77 130 200

10 40 60 70 80 90 110 113 123 138 140 170 230 236 242 245

Remova 123.

100

53 77 130 200

10 40 60 70 80 90 110 113 138 140 170 230 236 242 245

* Alternativa 2: Troque 123 com a antecessora:

100

53 77 113 200

10 40 60 70 80 90 110 123 130 138 140 170 230 236 242 245

Remova 123.

100

53 77 113 200

10 40 60 70 80 90 110 130 138 140 170 230 236 242 245

Redistribua as chaves com o vizinho à direita.

100

53 77 130 200

10 40 60 70 80 90 110 113 138 140 170 230 236 242 245

**Questão 4 (2,5 pontos)**. Considere uma grade não regular tridimensional tal que as fronteiras lateral esquerda, inferior e frontal não pertencem à grade; para cada célula, as fronteiras lateral esquerda, inferior e frontal não pertencem à célula. Cada célula da grade armazena uma lista encadeada de pontos.

Considere os seguintes tipos que representam a grade:

typedef struct lista Lista;

struct lista {

float x, y, z; /\* ponto na lista \*/

Lista\* prox; /\* ponteiro para próximo elemento da lista \*/

};

typedef struct grade Grade;

#define NX 200 /\* número de elementos do vetor gX \*/

#define NY 135 /\* número de elementos do vetor gY \*/

#define NZ 400 /\* número de elementos do vetor gZ \*/

struct grade {

float gX[NX]; /\* vetor que define as divisões no eixo X da grade \*/

float gY[NY]; /\* vetor que define as divisões no eixo Y da grade \*/

float gZ[NZ]; /\* vetor que define as divisões no eixo Z da grade \*/

Lista\* prim[NX][NY][NZ]; /\* lista por célula (inicializada com NULL) \*/

};

Implemente uma função que receba como entrada uma grade *g*, com a definição acima, e um ponto *(px, py, pz)*, remove o ponto da grade, se existir, e retorne

0 se o ponto não existia na grade

1 se o ponto existia e foi removido da grade

-1 se o ponto está fora da grade

A função deve seguir o seguinte protótipo:

int remove(Grade\* g, float x, float y, float z);

***Resposta***

int ocorre(Grade\* g, float x, float y, float z)

{ int i, j, k;

Lista\* ant, w;

if (x <= g->gX[0] || x > g->gX[NX-1] ||

y <= g->gY[0] || y > g->gY[NY-1] ||

z <= g->gZ[0] || z > g->gZ[NZ-1]) return -1;

for(i = 1; i < NX && x > g->gX[i]; i++) /\* calcula posição x da célula \*/

for(j = 1; j < NY && y > g->gY[j]; j++) /\* calcula posição y da célula \*/

for(k = 1; k < NZ && z > g->gZ[k]; k++) /\* calcula posição z da célula \*/

ant = NULL;

w = g->prim[i-1][j-1][k-1];

while(w != NULL && (w->x != x || w->y != y || w->z != z) /\* procura pt \*/

{ant = w; w = w->prox:} /\* na célula \*/

if (w == NULL) /\* não achou ponto \*/

return 0;

if (ant == NULL) /\* retira ponto do início da lista \*/

g->prim[i-1][j-1][k-1] = w->prox;

else p->prox = w->prox; /\* retira ponto do meio da lista \*/

free(w);

return 1;

}