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

|  |  |  |
| --- | --- | --- |
| 1a) | 2.5 |  |
| 2a) | 2.5 |  |
| 3a) | 2.5 |  |
| 4a) | 2.5 |  |
|  | 10.0 |  |

1. A prova é individual e sem consulta. Qualquer tentativa de “cola” resultará na anulação da prova de ambos os alunos envolvidos.
2. A interpretação faz parte da questão. Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
3. O tempo de prova é 1:45 h.
4. As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
5. A prova pode ser feita a lápis.

1) (2.5 pontos) Escreve 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

h(K, i) = ((K % N) + i) % N

onde

K = chave a ser inserida

N = tamanho da tabela de dispersão

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

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

#define N 1007; /\* tamanho da tabela de dispersão \*/

#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

Resposta:

int insere(int K)

{

int p, i;

p = (K % N);

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

{

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

{

tabela[p] = K;

return 1;

}

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

return 0;

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

}

return (-1);

}

2) (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) abaixo:

100

53 77 123 200

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

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

1. (0.5 ponto) Inserção de 243.
2. (1.0 ponto) Remoção de 40.
3. (1.0 ponto) Remoção de 100.

Resposta:

a) Inserção de 243.

Pesquise o nó onde a chave 243 deve ser inserida.

100

53 77 123 200

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

***243***

Divida o nó e insira a chave do meio (242) no pai.

100

53 77 123 200

10 40 60 70 80 90 110 113 130 138 140 170 230 236 ***242*** 243 245

100

53 77 123 200 242

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

b) Remoção de 40.

A chave 40 ocorre em uma folha e portanto pode ser removida.

100

53 77 123 200 242

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

100

***53*** 77 123 200 242

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

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

***100***

77 123 200 242

10 53 60 70 80 90 110 113 130 138 140 170 230 236 243 245

Não pode haver concatenação dos dois nós no segundo nível pois o total de chaves dos dois nós já está no limite (4 chaves). Redistribua então chaves entre os nós do segundo nível, com transferência de chave do pai:

123

77 100 200 242

10 53 60 70 80 90 110 113 130 138 140 170 230 236 243 245

c) Remoção de 100:

A chave 100 não ocorre em uma folha. Troque 100 com a sua sucessora 110 (poderia ser a predecessora).

123

77 110 200 242

10 53 60 70 80 90 100 113 130 138 140 170 230 236 243 245

Remova 100.

123

77 110 200 242

10 53 60 70 80 90 113 130 138 140 170 230 236 243 245

Redistribua as chaves com o vizinho à esquerda e o pai.

123

77 90 200 242

10 53 60 70 80 110 113 130 138 140 170 230 236 243 245

Novamente redistribua as chaves com o vizinho à esquerda e o pai.

123

70 90 200 242

10 53 60 77 80 110 113 130 138 140 170 230 236 243 245

3) (2.5 pontos) Um nó *m é vizinho* de um nó *n* sse há uma aresta entre *m* e *n* no grafo. O *clustering coefficient* de um nó *n* é definido como o número de arestas existentes entre os vizinhos de *n* dividido pelo número de todas as possíveis arestas entre os vizinhos de *n*; se *n* tem *0* ou *1* vizinho, então o *clustering coeficiente* é *0.*

Implemente uma função em C que receba como entrada um grafo não dirigido e um nó do grafo e retorne o *clustering coefficient* do nó no grafo. A função deve ter o seguinte protótipo:

int cc(Graph\* G, int n);

Considere que o grafo está representado por listas de adjacências:

typedef struct graph Graph;

typedef struct listNode ListNode;

struct listNode {

int node;

ListNode\* link;

};

struct graph {

int nv; /\* número de nós no grafo \*/

ListNode\*\* vv; /\* vv[i] aponta para os vizinhos do nó i \*/

};

Resposta:

int cc(Graph\* g, n) {

int a=0; /\* número de arestas existentes entre vizinhos de n \*/

int v=0; /\* número de vizinhos do nó n \*/

int m1, m2;

listNode\* w1, w2, w3;

/\*

Se n não tem vizinhos ou tem um só vizinho, então cc = 0

\*/

if (vv[n] == NULL || vv[n]->link == NULL) return 0;

/\*

Percorre os vizinhos de n até o penúltimo (w1->link != NULL)

\*/

for(w1 = vv[n]; w1->link != NULL; w1 = w1->link)

{ v = v + 1;

m1 = w1->node;

/\*

Percorre os vizinhos de n que ocorrem depois de m1 na lista

de vizinhos de n (evitando assim contar a mesma aresta duas vezes)

\*/

for(w2 = w1->link; w2 != NULL; w2 = w2->link)

{

m2 = w2->vertex /\* m2 é vizinho de n que ocorre depois de m1

na lista de vizinhos de n \*/

/\*

Verifica se m2 é vizinho de m1

\*/

for(w3 = vv[m1]; w3 != NULL && w3->node != m2; w3 = w3->link);

if (w3->node == m2) a = a + 1; /\* m2 é vizinho de m1 \*/

}

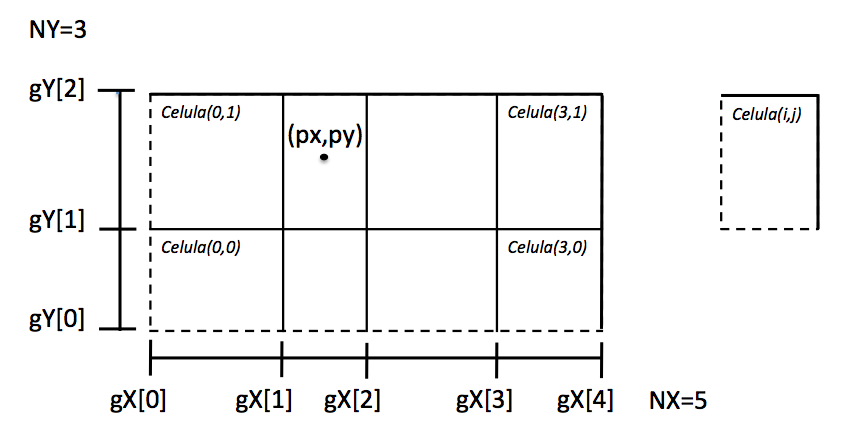
}

v = v + 1; /\* … pois o “for” vai até o penúltimo vizinho de n \*/

return a/(v\*(v-1)/2); /\* o número possível de arestas entre v nós \*/

} /\* é dado por v\*(v-1)/2 \*/

4) (2.5 pontos) Considere uma grade não regular bidimensional tal que (ver figura abaixo): as bordas lateral esquerda e inferior não pertencem à grade; para cada célula, as bordas lateral esquerda e inferior 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; /\* 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 \*/

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 \*/

Lista\* prim[NX][NY]; /\* 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)*, como na figura, e insira o ponto na grade, retornando 1 se a inserção foi bem sucedida, e 0, em caso contrário. A função deve seguir o seguinte protótipo:

int insere(Grade\* g, float px, float py);

e deve retornar:

0 se o ponto não puder ser inserido

1 se o ponto for inserido corretamente

Resposta:

int insere(Grade\* g, float x, float y)

{ int i, j;

Lista\* w;

if (x <= gX[0] || x > gX[NX-1] || y <= gY[0] || y > gX[NY-1]) return 0;

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

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

for(w = g->prim[i-1][j-1]; w != NULL; w = w->prox) /\* procura ponto na \*/

if (w->x = x && w->y = y) return 1; /\* lista da célula \*/

return 0

}