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

|  |  |  |
| --- | --- | --- |
| 1a) | 1,5 |  |
| 2a) | 1,0 |  |
| 3a) | 2,5 |  |
| 4a) | 2,5 |  |
| 2a) | 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 (1,5 pontos)** Os trechos de código abaixo são usados para resolver problemas de tamanho *n*. Determine a complexidade, no pior caso, de cada trecho de código. Explique sua resposta.

a) **(0,5 ponto)**

for ( i=1; i < n; i \*= 2 ) {

for ( j = n; j > 0; j /= 2 ) {

for ( k = j; k < n; k += 2 ) {

sum += (-j \* k) << i/2;

}

}

}

b) **(0,5 ponto)**

for( int i = 1; i <= n; i \*= 2 ) {

for( int j = 0; j < i; j++ ) {

for( int k = 0; k < n; k += 2 ) {

sum \*= k;

}

for( int k = 1; k < n; k \*= 2 ) {

sum /= k

}

}

}

c) **(0,5 ponto)**

for ( i=1; i < n; i \*= 2 ) {

for ( j = n; j > 0; j += 2 ) {

for ( k = j; k < n; k += 2 ) {

sum += (-j \* k) << i/2;

}

}

}

***Resposta***

(a) O tempo de execução do laço mais externo é proporcional a *log2(n)*, visto que *i* dobra a cada passagem. O tempo de execução do laço do meio é semelhante, exceto que *j* é dividido pela metade, começando em *n*. O tempo de execução do último laço é proporcional a *n*. Como os laços estão aninhados, devem-se multiplicar todos eles. Logo, a complexidade desse algoritmo é *O(n(log n)2).*

(b) Os dois laços mais internos são respectivamente proporcionais a *n* e *log n*; como estão em paralelo, a complexidade deve ser somada, o que resulta em serem proporcionais a *n*. O laço mais externo é proporcional a *log2(n)*, visto que os passos dobram a cada passagem. Já o laço do meio é sequencial, ou seja, proporcional a *n*. Assim a complexidade combinada dos laços é *O(n2 log n)*

(c) O segundo laço não para, já que *j* começa em *n* (um inteiro não negativo), é incrementado de 2 em 2 e, portanto, nunca chega a 0. Logo, o programa tem complexidade no pior caso de O(∞).

**Questão 2 (1,0 ponto)**

a) **(0,5 ponto)** Seja *T* uma tabela de dispersão (*hash table*) implementada com *encadeamento externo*. Assuma que *T* possui tamanho *n* e que a função de dispersão seja dada por:

Assuma que o total de elementos a serem inseridos seja de aproximadamente 600. Qual valor de *n* você escolheria? Explique sua resposta.

b) **(0,5 ponto)** Seja *T* uma tabela de dispersão (*hash table*) implementada com *encadeamento interior*. Assuma que *T* possui tamanho *n*. Defina uma função de dispersão *h(x,j)* que retorna a posição onde a chave *x* deve ser inserida, se possível, na *j-ésima* tentativa de inserir *x*. Não é necessário escrever a função em C; basta defini-la matematicamente. Explique porque a sua função de dispersão é adequada.

***Resposta***

a) *n* = 23 × 29 = 667 seria um valor adequado pois é aproximadamente 10% a mais do que 600 e possui fatores primos grandes, maiores do que 20.

b) Uma boa função de dispersão seria

De fato, *h(x,j)* retorna uma posição entre 0 e *n* onde a chave *x* deve ser inserida, se possível, na *j-ésima* tentativa de inserir *x*. Neste caso, *p* não deve ser um submúltiplo de *n*.

**Questão 3 (2,5 pontos)**

a) **(1,5 pontos)** Defina em pseudo-código (ou seja, não é necessário apresentar o código em C) um algoritmo não recursivo que recebe como entrada um ponteiro *r* para a raiz de uma árvore binária de busca e uma chave *K* e retorna um ponteiro *n* para o nó *N* da árvore tal que a chave armazenada em *N* seja a *menor chave armazenada na árvore que é maior ou igual* a *K*. Se *N* não existir, o algoritmo deve retornar NULL. Por simplicidade, assuma que os valores de chave são inteiros.

Assuma a seguinte estrutura para os nós de uma árvore binária de busca:

typedef struct \_abb Abb;

struct \_abb {

int chave;

Abb\* pai;

Abb\* esq;

Abb\* dir;

};

b) **(1,0 ponto)** Argumente porque a sua implementação está correta.

***Resposta***

a)

Entrada: *r* – um ponteiro para a raiz de uma árvore binária de busca

*K* – um inteiro positivo

Saída: *n* – um ponteiro para o nó *N*, se houver, definido como acima, ou o ponteiro nulo

1. Se a árvore for vazia, pare e retorne NULL;
2. Pesquise se a chave *K* existe na árvore, mantendo também um ponteiro *p* para o pai *P* do nó sendo visitado;
3. Se existir um nó *N* na árvore cuja chave é *K*, retorne um ponteiro para *N* e pare;
4. Se a chave de *P* for maior do que *K*, retorne *p* e pare.
5. Suba na árvore a partir de *p* até encontrar o primeiro ancestral *Q* cuja chave seja maior que do que a chave do seu filho ou chegar à raiz.
6. Retorne um ponteiro para *Q*, ou NULL, se o loop de (5) tiver chegado à raiz.

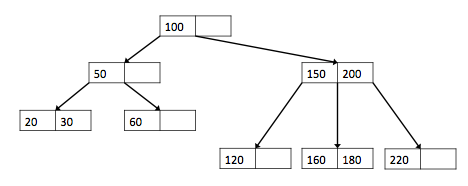
b)

* O algoritmo do item (a) pesquisa se existe um nó *N* na árvore com a chave *K*. Se existir, um ponteiro para *N* será a resposta correta.
* Caso não exista um nó *N* na árvore com a chave *K*, o algoritmo simula a inserção de um nó *M* com a chave *K* (como filho do nó *P*) e pesquisa o sucessor deste nó.
* Como o nó fictício *M* seria uma folha, não tem uma subárvore à direita.
* Logo, o sucessor do nó fictício *M* é computado subindo na árvore a partir do pai *P* de *M* e procurando o primeiro ancestral *Q* cuja chave seja maior que do que a chave do seu filho. Este ancestral *Q* conterá a chave que seria a sucessora de *K* na árvore, ou seja, a menor chave maior do que *K*.
* O algoritmo retorna um ponteiro para *Q*. Se a busca chegar na raiz (ancestral NULL), o algoritmo retorna NULL. Neste caso, a chave *K* é maior do que qualquer chave armazenada na árvore.

**Questão 4 (2,5 pontos)**

a) **(1.0 ponto)** Qual é o número máximo de chaves que uma árvore 2-3 de altura *h* pode armazenar (uma árvore que só possui a raiz tem altura 0)? Explique sua resposta.

b) **(0,5 ponto)** Qual é a menor altura de uma árvore 2-3 que armazena 80 chaves? Explique sua resposta.

c) **(1,0 ponto)** Remova sucessivamente as chaves 150 e 160 da árvore 2-3 abaixo. Mostre e comente todos os passos.

**\**

**Resposta**

a) Uma árvore 2-3 pode ter no máximo *N* chaves, onde

*N = (m-1)\*(1 + m + m2 + m3+ ... + mh) = (m-1) = mh+1 - 1*  onde *m*=3

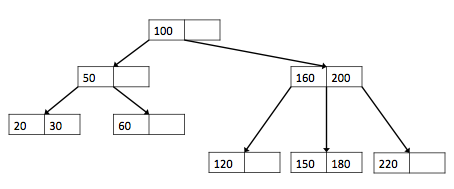
b) A menor altura *h* de uma árvore 2-3 com 80 chaves é o menor inteiro tal que:

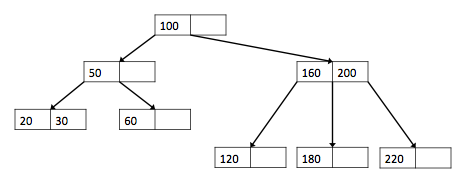
80 ≤ 3*h+1 – 1*

81 ≤ 3*h+1*

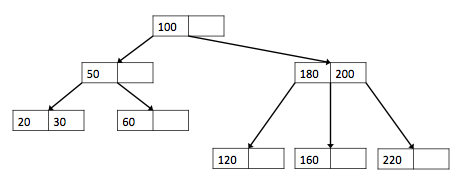
Logo, *h =3.*

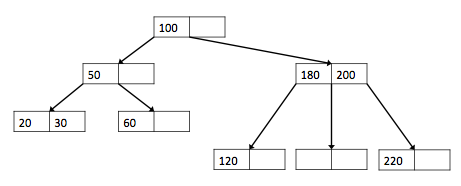
c) *Remoção de 150*

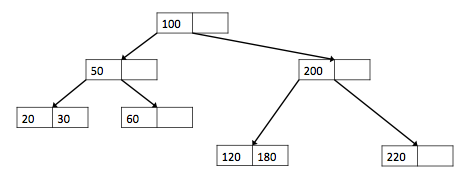
1) Troque 150 com a sua sucessora, 160

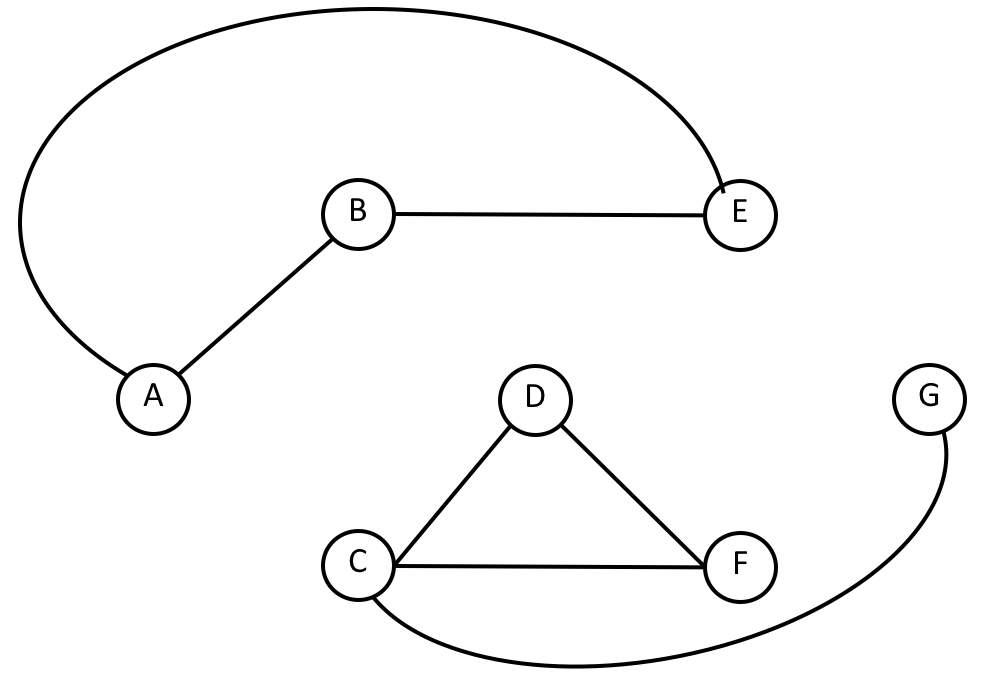
2) Remova 150

*Remoção de 160*

1) Troque com a sua sucessora, 180

2) Remova 160

Combine nós, rearrumando as chaves

**Questão 5 (2,5 pontos)**. O *giant coefficient* de um grafo não dirigido é definido como o número de vértices da maior componente conexa do grafo dividido pelo número total de vértices do grafo. Se o grafo for vazio, o *giant coefficient* é 0 por definição.

a) **(0,5 ponto)** Compute o *giant coefficient* do grafo ao lado.

Mostre os passos da computação.

b) **(2,0 pontos)** Considere que os grafos estão representados por listas de adjacências:

typedef struct graph Graph;

typedef struct listNode ListNode;

struct listNode {

int vertex;

ListNode\* link;

};

struct graph {

int nv; /\* número de vértices do grafo \*/

int\* vis; /\* vis[i]=0 sse o vertice i não foi visitado \*/

ListNode\*\* vv; /\* vv[i] aponta p/ lista de vértices adjacentes a i \*/

};

Considere a função para pesquisa em profundida apresentada em sala, mas modificada para marcar os vértices visitados com um inteiro *k*:

void dfs(Graph\* g, int N, int k)

onde g é um grafo representado por listas de adjacências como acima, N é um vértice do grafo, e k é um inteiro para marcar (no vetor vis do grafo g) todos os vértices visitados na chamada da função.

Define em pseudo-código uma função (não precisa implementar em C) que recebe como entrada um grafo não dirigido representado por listas de adjacências como acima e retorne o *giant coefficient* do grafo:

float giantCoefficient(Graph\* g)

A função deve chamar a pesquisa em profundida modificada descrita acima como subrotina.

***Resposta***

a)

1. Compute inicialmente as componentes conexas do grafo, contando quantos vértices há em cada uma. O grafo possui 2 componentes conexas, com 3 e 4 nós.
2. O *giant coefficient* será então *gc* = 4/7=0,5714

b)giantCoefficient(g)

1. Se g->nv = 0, retorne 0;
2. Inicialize gc = 0, N = 0, k = 0;
3. Inicialize vis(i)=0, para i=0 até g->nv-1.
4. Enquanto houver um vértice N não visitado, ou seja, com vis(N)=0 faça:
   1. Incremente k;
   2. Chame dfs(g,N,k);
   3. Conte quantos vértices em vis foram marcados com k;
   4. Se a contagem for maior do que gc, guarde a contagem em gc;
5. Retorne gc/g->nv;