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.
   1. **Qualquer tentativa de “cola” resultará na anulação da prova do aluno ou dos alunos envolvidos**.
   2. Os aparelhos celulares deverão permanecer desligados durante toda a prova. **Aparelhos celulares ligados 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) Considere um banco cujas agências são identificadas por um código de 5 dígitos, onde o último dígito, chamado de *dígito verificador*, é computado a partir dos 4 primeiros e serve para detectar códigos incorretos. O código é representado como um *string* de comprimento 5.

Considere uma tabela de hash para representar as agências do banco.

a) (0.5 ponto) Suponha que o banco possua 400 agências. Qual deve ser o tamanho, *tam*, da tabela de hash? Explique sua resposta.

b) (1.0 ponto) Escreva em C uma função de dispersão (função *hash*) que receba como argumentos:

* um *string* de 5 dígitos, *cod*, representando o código de uma agência
* um inteiro *tam* que define o tamanho da tabela *hash*

e retorne um inteiro que identifica a entrada na tabela de *hash* para a qual *cod* é mapeado.

c) (0.5 ponto) Argumente porque a sua implementação está correta e representa uma função de hash adequada.

d) (0.5 ponto) Suponha que o banco expanda as suas operações e abra 100 novas agências. Qual será o efeito desta expansão e o que deve ser modificado na função de hash e na tabela de hash para acomodar esta expansão? Explique a sua resposta.

**Resposta**

a) Para armazenar 400 agências, o tamanho da tabela de hash deve ser aproximadamente 10% maior, ou seja, 440. Podemos então tomar o tamanho da tabela de hash como sendo *tam* = 443 (um número primo).

b)

1. int hash(char \*cod, int tam) {

2. int i, hash=0;

3. for( i=0 ; i<4 ; i++ ) {

4. hash = hash + (int)(cod[i]) \* 10\*\*(3-i); }

5. return hash%tam; }

c) A função de hash do item (b) está correta pois devolve um inteiro entre 0 e *tam*-1 (passo 5).

Ela é uma função de hash adequada pois desconsidera o último dígito de *cod*, que é dependente dos demais (no passo 3, a variável *i* varia de 0 a 3). Além disto, como o tamanho da tabela é um primo, o passo 5 espalhará os códigos das agências adequadamente.

d) Ao abrir 100 novas agências, a tabela de hash terá fator de carga *α* = 500/443 = 1,12, portanto maior do que 1. Uma nova tabela deverá então ser alocada com tamanho, digamos, 557 (um primo maior do que 500+50).

A função de hash não precisa ser modificada pois o tamanho da tabela já é o segundo parâmetro da função.

Porém, cada código deverá removido da tabela original e reinserido na nova tabela, usando a função de hash com segundo parâmetro igual a 557. Para tal é necessário varrer toda a tabela original, sem usar a função de hash.

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

*G1* = { *x* ∈ *G* / *x* estuda Engenharia de Computação }

*G2* = { *x* ∈ *G* / *x* é já cursou INF1010 }

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* estuda Engenharia de Computação e *x* ainda não cursou INF1010 }

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 o problema de agrupar os empregados de uma empresa pela departamento em que trabalham, assumindo que um empregado só pode trabalhar em um departamento e que um empregado pode mudar de departamento ao longo do tempo. Assuma ainda que se deseja realizar duas operações:

* Determinar eficientemente se dois empregados trabalham no mesmo departamento;
* Implementar eficientemente a fusão de 2 departamentos em um só.

a) (1.0 ponto) Qual estrutura de dados, dentre as apresentadas em sala, seria mais eficiente para resolver o problema acima? Explique sua resposta, indicando como as operações acima serão implementadas.

c) (1.5 pontos) Defina em C a estrutura de dados escolhida no item (a) e implemente uma rotina para determinar se dois empregados não trabalham no mesmo departamento da forma mais eficiente possível. A rotina não deverá chamar as operações definidas em sala para a estrutura escolhida.

**Resposta**

a) A estrutura mais apropriada seria uma partição dinâmica de conjuntos, implementada como uma floresta com ranking nos nós e com balanceamento de união e compressão de caminhos no Find (algoritmo Union-Find).

Para determinar se dois empregados não trabalham no mesmo departamento da forma mais eficiente possível, basta realizar o Find para cada um deles e verificar se os Finds retornam raízes diferentes. Durante o find será realizada compressão de caminho.

Para combinar dois departamentos em um só, basta realizar o union com balanceamento.

b)

forest\_node\* Find\_same(forest\_node\* node1, forest\_node\* node2)

{

forest\_node\* temp;

forest\_node\* root1 = node1;

forest\_node\* root2 = node2;

while (root1-> parent != NULL) {

root1 = root1-> parent; /\*determinação da raiz \*/

}

while (node1->parent != root1) {

temp = node1->parent; /\* compressão de caminho \*/

node1->parent = root1;

node1 = temp;

}

while (root2-> parent != NULL) {

root2 = root2-> parent; /\*determinação da raiz \*/

}

while (node2->parent != root2) {

temp = node2->parent; /\* compressão de caminho \*/

node2->parent = root2;

node2 = temp;

}

return root1 != root2;

}

**Questão 4** **Questão 4** (2.5 pontos) Considere o problema de organizar uma fila de atendimento em que as pessoas mais velhas tenham prioridade sobre as mais novas. Caso duas pessoas tenham a mesma idade, a ordem é indistinta. Assuma ainda que se deseja realizar as seguintes operações:

* Organizar inicialmente um conjunto de pessoas, segundo a prioridade por idade;
* Atender, ou seja, tirar da fila de atendimento, a pessoa de maior idade;
* Incluir uma nova pessoa na fila de atendimento, segundo a prioridade por idade.

a) (0.5 ponto) Porque um *heap* seria a estrutura mais eficiente para resolver o problema acima? Explique sua resposta.

b) (0.5 ponto) Considere inicialmente um conjunto de pessoas com as seguintes idades: 15, 60, 30, 45, 90. Mostre como construir o *heap*, utilizando um vetor, da forma mais eficiente possível, para organizar este conjunto de pessoas na fila de atendimento.

c) (1.5 ponto) Escreva em C uma função que recebe como argumento um *heap*, organizado como nos itens anteriores, e um inteiro *i* e remove o elemento do *heap* que ocupa a *i*-ésima posição do vetor. A rotina deve retornar 1 se a alteração ocorreu corretamente e 0 em caso contrário. Assuma que o *heap* seja definida como abaixo:

struct \_heap {

int max; /\* tamanho máximo do heap \*/

int pos; /\* próxima posicao disponível no vetor \*/

float\* prioridade; /\* vetor das prioridades \*/

};

typedef struct \_heap Heap;

Assuma ainda que a assinatura da rotina seja:

int heap\_altera(Heap\* heap, int i)

**Resposta**

a) A estrutura mais eficiente será um *heap*, **já que não é necessário manter uma ordenação completa das pessoas na fila de atendimento**.

b) A construção mais eficiente possível aloca um vetor com 5 posições. Em seguida aplica-se o algoritmo eficiente para construção de *heap*. Tomando-se os 3 últimos elementos do conjunto inicial, temos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **Explicação** |
|  |  | 30 | 45 | 90 | Aloque os 3 últimos elementos do conjunto inicial, reservando 2 posições para os 2 primeiros elementos. | |
|  | 15 | 30 | 45 | 90 | Acrescente 15 na primeira posição vazia. | |
|  | 15 | 30 | 45 | 90 | Compare 15, na posição 1, com os filhos, nas posições (2\*1+1) e (2\*1+2). | |
|  | 90 | 30 | 45 | 15 | Troque com o mais velho. | |
| 60 | 90 | 30 | 45 | 15 | Acrescente 60 na primeira posição vazia. | |
| 60 | 90 | 30 | 45 | 15 | Compare 60, na posição 0, com os filhos, nas posições (2\*0+1) e (2\*0+2). | |
| 90 | 60 | 30 | 45 | 15 | Troque com o mais velho. | |
| 90 | 60 | 30 | 45 | 15 | Compare 60, na posição 1, com os filhos (2\*1+1) e (2\*1+2). | |
| 90 | 60 | 30 | 45 | 15 | Nenhuma troca é necessária. | |

Alternativamente, tomando-se os 3 primeiros elementos do conjunto inicial, temos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **Explicação** |
|  |  | 15 | 60 | 30 | Aloque os 3 primeiros elementos do conjunto inicial, reservando 2 posições para os 2 primeiros elementos. | |
|  | 45 | 15 | 60 | 30 | Acrescente 45 na primeira posição vazia. | |
|  | 45 | 15 | 60 | 30 | Compare 45, na posição 1, com os filhos, nas posições (2\*1+1) e (2\*1+2). | |
|  | 60 | 15 | 45 | 30 | Troque com o mais velho. | |
| 90 | 60 | 15 | 45 | 30 | Acrescente 90 na primeira posição vazia. | |
| 90 | 60 | 15 | 45 | 30 | Compare 90, na posição 0, com os filhos, nas posições (2\*0+1) e (2\*0+2). | |
| 90 | 60 | 15 | 45 | 30 | Nenhuma troca é necessária. | |

c) Não é possível remover a i-ésima posição do vetor simplesmente através de um “shift-left” a partir da i-ésima posição em diante pois não se trata de um vetor ordenado, mas de um vetor que representa um *heap*. Deve-se implementar uma rotina específica, como abaixo:

1. Se o *heap* estiver vazio (ou seja, pos=0) ou i estiver fora do heap (ou seja, i < 0 ou i > pos -1), retorne 0.
2. Se i for a última posição do heap (ou seja, i = pos -1), diminua o tamanho de *heap* de 1 e retorne 1.
3. Coloque a prioridade da última posição do *heap* na i-ésima posição.
4. Diminua o tamanho de *heap* de 1.
5. Se a prioridade de i for menor do que a prioridade de algum filho de i, execute corrige\_abaixo\_i(heap), semelhante ao apresentado em sala.
6. Se i não for a primeira posição do heap (ou seja, i=0) e a prioridade de i for maior do que a prioridade do pai de i, execute corrige\_acima\_i(heap), semelhante ao apresentado em sala.
7. Retorne 1;

Notas:

* **Os passos (5) e (6) são necessários pois, diferentemente das rotinas para remover o primeiro elemento e inserir um novo elemento no *heap*, pode ser necessário corrigir abaixo ou corrigir acima.**
* Os passos (5) e (6) podem ser simplificados se os testes forem incorporados às rotinas corrige\_abaixo\_i(heap) e corrige\_acima\_i(heap), com na implementação em C abaixo.

float heap\_altera(Heap\* heap, int i)

{ int filho\_esq, filho\_dir, filho;

/\* heap vazio ou i errado \*/

if (heap->pos = 0 || i < 0 || i > heap->pos-1) return 0;

/\* i é o último elemento do heap \*/

if (i == heap->pos - 1)

{ heap->pos--; return 1; }

/\* i não é o último elemento do heap \*/

heap->prioridade[i] = heap->prioridade[heap->pos-1];

heap->pos--;

if (corrige\_abaixo\_i(heap,i)) /\* se não houve troca com um filho, \*/

corrige\_acima\_i(heap,i); /\* então tente uma troca com o pai \*/

return 1;

}

As rotinas abaixo são quase idênticas às apresentadas em sala:

int corrige\_abaixo\_i(Heap\* heap, int i) {

int pai = i; /\* (na rotina original era pai=0) \*/

int trc = 1; /\* trc=1, não houve troca; trc = 0 houve troca \*/

while (2\*pai+1 < heap->pos){

int filho\_esq=2\*pai+1;

int filho\_dir=2\*pai+2;

int filho;

if (filho\_dir >= heap->pos) filho\_dir=filho\_esq;

if (heap->prioridade[filho\_esq]>heap->prioridade[filho\_dir])

filho=filho\_esq;

else

filho=filho\_dir;

if (heap->prioridade[pai] < heap->prioridade[filho]){

troca(pai,filho,heap->prioridade);

trc = 0;}

else

break;

pai=filho;

}

return trc; /\* trc permite uma pequena otimização na rotina principal \*/

}

void corrige\_acima\_i(Heap\* heap, int pos) { /\* (a rotina é idêntica) \*/

while (pos > 0) {

int pai = (pos-1)/2;

if (heap->prioridade[pai] < heap->prioridade[pos])

troca(i,pai,heap->prioridade);

else

break;

pos=pai;

}

}

void troca(int a, int b, int\* v) {

int f = v[a];

v[a] = v[b];

v[b] = f;

}