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

|  |  |  |
| --- | --- | --- |
| 1a) | 2,0 |  |
| 2a) | 3,0 |  |
| 3a) | 2,0 |  |
| 4a) | 3,0 |  |
|  | 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** (2,0 pontos) Considere um município onde os veículos são identificados por placas. Cada placa é uma cadeia de símbolos de comprimento 8, onde cada símbolo ou é uma letra do alfabeto ou é um dos 10 dígitos. Por exemplo, “X9Y0BCD9” é uma possível placa. Porém, nem todas as possíveis combinações de letras e números são de fato utilizadas.

Considere uma tabela de hash para representar as placas de veículos que de fato estão em uso.

a) (0,5 ponto) Suponha que o município possua hoje 100.000 veículos. 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 comprimento 8, *cod*, representando a placa de um veículo
* 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 mapeada.

Argumente porque a sua implementação está correta e representa uma função de hash adequada.

c) (0,5 ponto) Suponha que se espera que o número de veículos cresça 20% nos próximos 5 anos. Diante desta nova informação, o que deve ser modificado nas letras (a) e (b) para que não seja necessário reconstruir a estrutura de dados nos próximos 5 anos?

**Resposta**

a) Para armazenar 100.000 de veículos, o tamanho da tabela de hash deve ser aproximadamente 10% maior, ou seja, 110.000. Podemos então tomar o tamanho da tabela de hash como sendo *tam* = 37\*47\*67=116.513 (um número com fatores primos maiores do que 20).

b)

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

2. int i, hash=0;

3. for( i=0 ; i<8 ; i++ ) {

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

5. return hash%tam; }

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 espalhará as placas adequadamente.

c) Ao crescer 20% a frota de veículos nos próximos 5 anos, precisaríamos de uma tabela de hash para 120.000 chaves. A tabela de hash deveria ter então tamanho perto de 132.000.

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

Porém, deveríamos já alocar a tabela para este futuro número de veículos, ou seja, com tamanho perto de 132.000. Um tamanho adequado seria tam = 37\*53\*67 = 131.387 (novamente, um número com fatores primos maiores do que 20).

**Questão 2** (3,0 pontos)

Considere a representação de conjuntos utilizando vetores de bits, definidos como:

struct \_bitvector {

int max; /\* número máximo de elementos do conjunto \*/

int \*vector;

};

typedef struct \_bitvector BitVector;

a) (1,5 ponto) Implemente em C uma rotina que recebe como entrada dois conjuntos *A* e *B* e devolve o conjunto *D* = *A* – *B*. A rotina não deverá chamar as operações definidas em sala para vetores de bits.

Nota: *A* – *B*, a *diferença* de *A* e *B*, é o conjunto de todos os elementos de *A* que não estão em *B*.

b) (0,5 ponto) Qual a complexidade, no pior caso, da rotina implementada no item (a), medida em termos do tamanho dos vetores de bits. Explique sua resposta.

c) (0,5 ponto) Assuma que os conjuntos *A* e *B* estão representados como tabelas de hash. Explique como seria implementada a operação de diferença. Esta implementação seria mais ou menos eficiente do que a descrita no item (a)?

d) (0,5 ponto) Qual seria a complexidade, no pior caso, desta segunda implementação da operação de diferença, medida em termos do número de elementos dos conjuntos. Explique sua resposta.

**Resposta**

**a)** BitVector\* bvN(BitVector\* A, BitVector\* B)

{

int i;

int minab = (A->max < B->max) ? A->max : B->max;

int minnum = (minab-1) /sizeof(int)+1; /\* número de int. do menor \*/

int num=((A->max-1)/sizeof(int))+1; /\* tamanho do vetor D \*/

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

D->max = A->max; /\* D terá no máximo o número de elementos de A \*/

D->vector = (int\*)malloc(num\*sizeof(int));

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

D->vector[i] = A->vector[i] & ~(B->vector[i]);

for (i=minnum; i<num; i++) /\* copia o resto de A para D, se preciso \*/

D->vector[i] = A->vector[i];

return DIFF;

}

(b) Em função dos for-loops, a rotina possui complexidade *O(n)*, onde *n* é o tamanho do vetor de bits do conjunto *A*. Note que as operações bit-a-bit não influenciam a complexidade do pior caso.

(c) Se os conjuntos fossem representados por tabelas de hash, a implementação da diferença seria:

1. Percorra sequencialmente toda a tabela que representa o conjunto *A*.
2. Para cada elemento *a* encontrado na tabela representando *A*,

Procure se *a* existe na tabela que representa *B*, usando a função de hash.

Se não existir, insira *a* na tabela representando a diferença.

A implementação seria viável, mas não tão eficiente quanto a proposta em (a) – ver item (d).

(d) A complexidade seria *O*(*n.m*), onde *m* e *n* são as cardinalidades de *A* e *B*, pois, para cada elemento de *A*, é necessário pesquisa-lo em *B*, que no pior caso da tabela de hash é proporcional à cardinalidade de *B*.

**Questão 3** (2,0 pontos) Considere a representação de partições por florestas, com as otimizações da operação de union através do balanceamento por altura e da operação de find por compressão de caminhos. Caso haja empate na operação de union, escolha o menor inteiro para raíz.

1. (1,5 ponto) Considere o conjunto {1,2,3,4,5}. Começando com a floresta representando a partição {{1},{2},{3},{4},{5}}, desenhe a sequência de florestas obtidas sucessivamente pelas operações:

union(1,2), union(3,4), union(3,5), find(4), union(1,3), find(4), find(4)

(0,5 ponto) Quantas comparações são realizadas, ao todo, pelas 3 execuções da operação find(4) da sequencia acima? Explique a sua resposta.

**Resposta**

(a)

4

5

find(4)

(terceira vez)

1

2

union(1,2)

3

4

union(3,4)

union(3,5)

3

4

5

1

2

union(1,3)

3

4

5

1

2

find(4)

(segunda vez)

3

4

5

3

2

1

find(4)

(primeira vez)

3

4

5

5

(b) find(4) – 1a vez: 1 comparação

find(4) – 2a vez: 2 comparações

find(4) – 3a vez: 1 comparação

Total de comparações: 4

**Questão 4** (3,0 pontos)

a) (1,0 ponto) Considere o conjunto de prioridades *P* = { 6.2, 7.2, 7.1, 6.0, 7.1 }. Mostre como construir um *max heap*, utilizando um vetor, da forma mais eficiente possível, para organizar este conjunto de prioridades.

b) (0,5 ponto) Suponha que o elemento com a segunda maior prioridade do *heap* esteja na segunda posição do vetor. É possível remover do *heap* o elemento com a segunda maior prioridade copiando a terceira posição do vetor para a segunda, a quarta para a terceira e assim por diante? Explique a sua resposta.

c) (1,5 ponto) Escreva em C uma função que recebe como argumento um *heap*, organizado como no item (a), e remove do *heap* o elemento com a segunda maior prioridade, usando o menor espaço e o menor tempo possível. A rotina deve retornar 1 se a alteração ocorreu corretamente e 0 em caso contrário. Assuma que o *heap* seja definido 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)

A rotina não deve chamar as operações de inserção ou remoção de um heap apresentadas em sala, mas poderá usar uma das operações auxiliares apresentadas em sala.

**Resposta**

a) 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 primeiros elementos do conjunto inicial, temos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **Explicação** |
|  |  | 6.2 | 7.2 | 7.1 | Aloque os 3 primeiros elementos do conjunto inicial, reservando 2 posições para os 2 primeiros elementos. | |
|  | 6.0 | 6.2 | 7.2 | 7.1 | Acrescente 6.0 na primeira posição vazia. | |
|  | 6.0 | 6.2 | 7.2 | 7.1 | Compare 6.0, na posição 1, com os filhos, nas posições (2\*1+1) e (2\*1+2). | |
|  | 7.2 | 6.2 | 6.0 | 7.1 | Troque com o de CR mais alto. | |
| 7.1 | 7.2 | 6.2 | 6.0 | 7.1 | Acrescente 7.1 na primeira posição vazia. | |
| 7.1 | 7.2 | 6.2 | 6.0 | 7.1 | Compare 7.1, na posição 0, com os filhos, nas posições (2\*0+1) e (2\*0+2). | |
| 7.2 | 7.1 | 6.2 | 6.0 | 7.1 | Troque com o de CR mais alto. | |
| 7.2 | 7.1 | 6.2 | 6.0 | 7.1 | Compare 7.1, na posição 1, com os filhos (2\*1+1) e (2\*1+2). | |
| 7.2 | 7.1 | 6.2 | 6.0 | 7.1 | Nenhuma troca é necessária. | |

b) Não é possível remover a segunda posição do vetor simplesmente através de um “shift-left” a partir da segunda 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 só contiver 1 elemento (pos <= 1), retorne 0.
2. Se o heap só contiver 2 elementos (ou seja, pos = 2), diminua o tamanho de *heap* de 1 e retorne 1.
3. Guarde o primeiro elemento do heap, ou seja, o elemento de maior prioridade, em uma variável temporária f.
4. Coloque o elemento da última posição do *heap* na primeira posição.
5. Diminua o tamanho de *heap* de 1.
6. Execute corrige\_abaixo(heap). O primeiro elemento do novo heap é o de segunda maior prioridade do heap original.
7. Coloque o valor de f na primeira posição do heap. Ou seja, remova o primeiro elemento do heap (que é o segundo do heap original) e coloque no lugar o primeiro elemento do heap original. Não é necessário reorganizar o heap pois o elemento na primeira posição é maior do que todos os outros (pois era o maior elemento do heap original).
8. Retorne 1;

c)

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

{ float f;

/\* o heap é vazio ou só tem 1 elemento \*/

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

/\* O segundo é o último elemento do heap \*/

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

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

/\* O segundo não é o último elemento do heap \*/

f = heap->prioridade[0]; /\* salva em f o maior elemento

da estrutura original

\*/

heap->prioridade[0] =

heap->prioridade[heap->pos-1]; /\* remove o maior \*/

heap->pos--;

corrige\_abaixo(heap); /\* o segundo maior agora está na raiz,

ou seja, é o primeiro elemento do vetor

\*/

heap->prioridade[0] = f; /\* recoloca o maior original na 1a posição,

o que efetivamente remove o segundo maior \*/

return 1;

}