Aluno(a):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ matrícula:\_\_\_\_\_\_\_\_\_\_\_\_

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

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

1. **Questão 1** (2,5 pontos): A rotina teste abaixo é usada para resolver problemas de tamanho n. Qual a sua complexidade no pior caso (O maiúsculo/Ômicron) da rotina teste. Explique a sua resposta detalhadamente.

int teste (int n) {

int i, j, k, sum = 0;

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;

}

}

}

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

for (int k = n; k > 0; k -= 2) {

sum \*= k;

}

}

return sum;

}

**Resposta**

Primeiro ninho de laços:

O número de execuções do laço mais externo é log2(n) visto que i dobra a cada passagem. O número de execuções do laço do meio é semelhante, exceto que j é dividido pela metade, começando em n. O número de execuções do laço mais interno é limitado por n/2. Como os laços estão aninhados, devem-se multiplicar todos eles. Assim, a complexidade combinada dos laços é O(n(log n)2).

Segundo ninho de laços:

O número de execuções do laço externo é n/2. O número de execuções do laço interno é limitado por n/2. Assim, a complexidade combinada dos laços é O(n2).

Custo final:

Como o custo do segundo ninho domina o custo do primeiro ninho, o custo da rotina teste é   
O(n2).

**Questão 2** (2,5 pontos): Considere uma empresa cujos funcionários são identificados por um código de 6 dígitos, onde o último dígito, chamado de *dígito verificador*, é computado a partir dos 5 primeiros e serve para detectar códigos incorretos. O código é representado como uma string de comprimento 6.

Considere uma tabela de hash para representar os funcionários da empresa.

a) (0,5 ponto) Suponha que a empresa possua 600 funcionários. Qual deve ser o tamanho, *tam*, da tabela de hash? Explique sua resposta.

b) (0,5 ponto) Escreva em C uma função de dispersão (função *hash*) que receba como argumentos:

* uma *string* de 6 dígitos, *cod*, representando o código de um funcionário
* 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) (1,0 ponto) Suponha que a empresa expanda as suas operações e admita 200 novos funcionários. 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 600 funcionários, o tamanho da tabela de hash deve ser aproximadamente 10% maior, ou seja, 660. Podemos então tomar o tamanho da tabela de hash como sendo *tam* = 667 = 29\*23 (fatores primos maiores do que 20 portanto).

b)

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

2. int i, hash=0;

3. for( i=0 ; i<5 ; 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 admitir 200 novos funcionários, a tabela de hash terá fator de carga *α* = 800/667 = 1,19, portanto maior do que 1. Uma nova tabela deverá então ser alocada com tamanho, digamos, 889 (novamente fatores primos maiores do que 20 e 10% de folga).

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 889. Para tal é necessário varrer toda a tabela original, sem usar a função de hash.

**Questão 3** (2,5 pontos): Considere a implementação de conjuntos como vetores de bits.

Implemente em C uma função que computa a *cardinalidade do complemento* de um conjunto. A função recebe como entrada um conjunto *a* e retorna a cardinalidade do complemento de *a*. A interface da função é a seguinte:

int bvCardCompl(BitVector\* a);

*A sua implementação não deverá usar as operações do TAD de conjuntos visto em sala.*

Assuma que os bitvectors estejam representados por:

struct \_bitvector {

  int max;

  int \*vector;

};

typedef struct \_bitvector BitVector;

Assuma ainda que o universo (o conjunto de todos os elementos) e todos os conjuntos considerados tenham o mesmo tamanho, indicado por max no struct acima.

**Nota:**

O *complemento* de um conjunto *A*, denotado , é o conjunto de elementos do universo *U* que não estão em *A*. Por exemplo, se *U*={*a,b,c,d,e*} e *A=*{*b,e*}, o complemento de *A* será ={*a,c,d*}.

A *cardinalidade* de um conjunto é o número de elementos do conjunto.

**Resposta**

int bvCardCompl(BitVector\* a)

{

int i, j, bitmap, card = 0;

int size= sizeof(int);

int num =((a->max-1)/size)+1; /\* numero de ints para max bits \*/

for (i=0; i < num-1; i++) { /\* conta elementos do conjunto no vetor \*/

bitmap = a->vector[i]; /\* exceto no último elemento do vetor \*/

for (j=0; j < size; j++) {

card = card + (1-(bitmap%2)); /\* incrementa card se o último bit for 0 \*/

bitmap = bitmap / 2; /\* remove o último bit \*/

}

}

bitmap = a->vector[num-1]; /\* tratamento do último elemento do vetor \*/

for (j=0; j < size%max; j++) { /\* que pode estar incompleto \*/

card = card + (1-(bitmap%2)); /\* incrementa card se o último bit for 0 \*/

bitmap = bitmap / 2; /\* remove o último bit \*/

}

}

return card;

}

**Questão 4** (2,5 pontos): Escreva uma rotina em C que receba como entrada um *min heap* e uma percentagem *p*, onde p > 1.0, modifique a prioridade *x* do menor elemento do heap para *p.x* e reorganize a estrutura para que mantenha as propriedades de um *min heap*. A rotina deve retornar 1 se a alteração ocorreu corretamente e 0 em caso contrário.

A sua implementação deve fazer o menor número possível de comparações e movimentações e não deve usar as operações sobre heaps pré-definidas.

Assuma que o *min heap* seja definida como abaixo:

struct \_heap {

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

int pos; /\* proxima posicao disponivel no vetor \*/

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

};

typedef struct \_heap Heap;

Assuma ainda que a assinatura da rotina seja:

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

**Resposta**

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

{

if (heap->pos>0) {

heap->prioridade[0]=heap->prioridade[0]\*p;

corrige\_abaixo(heap);

return 1;

}

else {

printf("Heap VAZIO!");

return 0;

}

}

static void corrige\_abaixo(Heap\* heap){

int pai=0;

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);

else

break;

pai=filho;

}

}