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

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

a) (0,5 ponto)

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

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

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

}

}

}

for ( i=1; i < n; i \*= 5 ) {

for ( j = n; j > 0; j /= 3 ) {

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

}

}

}

b) (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) *Primeiro ninho de “for”:* O laço mais externo é executado *log2(n)* vezes, visto que *i* dobra a cada passagem. O laço interno é executado *n/2* vezes, já que *j* é decrementado de 2 em 2. Como os laços estão aninhados, o comando

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

é executado *log2(n)\*(n/2)* = *(n* \* *log2(n))/2* vezes.

*Segundo ninho de “for”:* O laço mais externo é executado *log5(n)* vezes, visto que *i* é multiplicado por 5 a cada passagem. O laço interno é executado *log3(n)* vezes, já que *j* é dividido por 3 a cada passagem. Como os laços estão aninhados, o comando

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

é executado *log5(n)\*log3(n)* vezes.

Logo, a complexidade do algoritmo é *O(n* \* *log(n))*.

b) O enunciado da segunda linha é:

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

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** [1,5 pontos]:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  | 30 |  |  | sp | ! | " | # | $ | % | & | ' |
|  | 40 | ( | ) | \* | + | , | - | . | / | 0 | 1 |
|  | 50 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
|  | 60 | < | = | > | ? | @ | A | B | C | D | E |
|  | 70 | F | G | H | I | J | K | L | M | N | O |
|  | 80 | P | Q | R | S | T | U | V | W | X | Y |
|  | 90 | Z | [ | \ | ] | ^ | \_ | ` | a | b | c |
|  | 100 | d | e | f | g | h | i | j | k | l | m |
|  | 110 | n | o | p | q | r | s | t | u | v | w |
|  | 120 | x | y | z | { | | | } | ~ |  |  |  |

Figura 1: Códigos ASCII de alguns caracteres que podem ser impressos (sp representa espaço).

A Figura 1 apresenta os códigos ASCII de alguns caracteres. Por exemplo, os códigos das letras “d” e “e” são respectivamente “100” e “101”. Considere as seguintes funções de hash, onde s é uma cadeia de caracteres ASCII formada apenas por letras minúsculas do alfabeto, representando a chave, e M é o tamanho da tabela de hash:

1. int hash1(string s, int M)
2. {
3. int i, h=0;
4. for (i=0; s[i]!='\0'; i++)
5. h = s[i] + (h << 6) + (h << 16) - h;
6. h = abs(h) % M;
7. return h;
8. }
9. int hash2(string s, int M)
10. {
11. int i, h=0;
12. for (i=0; s[i]!='\0'; i++)
13. h+=s[i];
14. h = abs(h) % M;
15. return h;
16. }

a) [0,5 ponto] Estas funções estão corretas? Explique a sua resposta.

b) [1,0 ponto] Estas funções são boas funções de hash? Explique a sua resposta.

**Resposta**

a) As funções estão corretas pois retornam um inteiro entre 1 e M-1, ou seja, o resto da divisão do valor absoluto de h por M (linhas 6 e 14).

b) hash1 é uma boa função de hash pois é computável eficientemente, espalha uniformemente as chaves, levando em consideração todos os caracteres da cadeia e a própria ordem dos caracteres na cadeia, minimizando assim as colisões.

Já a função hash2 não é uma função de hash inteiramente adequada pois considera os 4 bits mais à esquerda de cada caracter, que são iguais para as letras “a”- “c”, “d”-“m”, “n”-“w” e “x”-“z”, e não leva em consideração a ordem dos caracteres na cadeia. Portanto, a primeira função é uma melhor opção.

Nota: hash1 computa a função h(n), definida pela equação de recorrência abaixo:

h(0) = s0;

h(i) = h(i - 1) \* 65599 + si;

Por uma questão de eficiência, hash1 não utiliza diretamente multiplicação, mas o *bit left shift operator* “<<”. De fato, observe que 65599 = 64 + 65536 – 1 = 2\*\*6 + 2\*\*16 – 1.

**Questão 3** (2,5 pontos) Considere o problema de representar 2 subconjuntos de um universo *A* de alunos, definidos como:

*E* = { *x* ∈ *A* / *x* estuda Engenharia de Computação }

*I* = { *x* ∈ *A* / *x* cursou INF1010 }

Suponha que os subconjuntos são representados utilizando vetores de bits, definidos como:

struct \_bitvector {

int max;

int \*vector;

};

typedef struct \_bitvector BitVector;

a) (1,0 ponto) Suponha que:

*A* = {A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, A18}

*E* = {A1, A3, A4, A10, A18}

*I* = {A2, A3, A4}

Considere que um número inteiro ocupe apenas 16 bits e que os bits são ordenados da direita para a esquerda. Assuma que A1 corresponda ao bit mais à direita, A2 corresponda ao segundo bit mais à direita, e assim por diante.

Mostre os vetores de bits que representam *E* e *I*.

b) (1,5 pontos) Implemente em C uma rotina que recebe como entrada *E* e *I* e devolva o subconjunto *N* de *A* definido como:

*N* = { *x* ∈ *A* / *x* não estuda Engenharia de Computação e *x* cursou INF1010 }

A rotina não deverá chamar as operações definidas em sala para a estrutura escolhida.

**Resposta**

(b)

BitVector\* bvN(BitVector\* E, BitVector\* I)

{

int i;

int min=(E->max < I->max)? E->max : I->max;

int num=((min-1)/sizeof(int))+1; /\* tamanho do menor vetor \*/

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

N->max = min;

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

for (i=0; i<num; i++) /\* computa H \*/

N->vector[i] = ~(E->vector[i]) & (I->vector[i]);

return N;

}

**Questão 4** (2,5 pontos) Considere o problema de agrupar os alunos de uma universidade pelo curso em que estão matriculados, assumindo que um aluno só pode estar matriculado em um curso, mas que ele pode mudar de curso ao longo do tempo. Assuma ainda que se deseja realizar duas operações:

* Determinar eficientemente se dois alunos não estão no mesmo curso;
* Implementar eficientemente a união de 2 cursos 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.

b) (0,5 pontos) Defina um typedef que corresponda à estrutura escolhida no item (a).

c) (1,0 ponto) Implemente em C uma rotina para unir 2 cursos em um só. A rotina deverá usar o typedef definido no item (b), mas 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 alunos não estão no mesmo curso 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 unir dois cursos em um só, basta realizar o union com balanceamento.

b)

typedef struct forest\_node\_t {

void\* value;

struct forest\_node\_t\* parent;

int rank;

} forest\_node;

c)

forest\_node\* Union(forest\_node\* node1, forest\_node\* node2) {

if (node2-> rank > node1-> rank) {

node1->parent = node2;

return node2;

} else if (node1-> rank > node2->rank) {

node2->parent = node1;

} else { /\* iguais \*/

node2->parent = node1;

node1-> rank++;

}

return node1;

}

**Questão 5** (2,5 pontos) Considere o problema de organizar um conjunto de alunos de tal forma que os alunos com maior CR tenham prioridade sobre os alunos que tem menor CR. Caso dois alunos tenham o mesmo CR, a ordem é indistinta. Assuma ainda que se deseja realizar as seguintes operações:

* Organizar inicialmente o conjunto de alunos, segundo a prioridade por CR;
* Remover o aluno com maior CR;
* Incluir um novo aluno, segundo a prioridade por CR.

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 alunos com os seguintes CRs: 6.2, 7.2, 7.1, 6.0, 7.1. Mostre como construir o *heap*, utilizando um vetor, da forma mais eficiente possível, para organizar este conjunto de alunos com prioridades dadas pelo CR.

c) (1,5 ponto) Escreva em C uma função que recebe como argumento um *heap*, organizado como nos itens anteriores, e remove do *heap* o aluno com o *segundo* maior CR, 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 estrutura mais eficiente será um *heap*, **já que não é necessário manter uma ordenação completa do alunos**.

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 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. | |

c) 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;

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];

heap->prioridade[0] = heap->prioridade[heap->pos-1];

heap->pos--;

corrige\_abaixo(heap);

heap->prioridade[0] = f;

return 1;

}