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

|  |  |  |
| --- | --- | --- |
| 1a) | 3.0 |  |
| 2a) | 2.0 |  |
| 3a) | 2.0 |  |
| 4a) | 3.0 |  |
|  | 10.0 |  |

1. A prova é individual e sem consulta. Qualquer tentativa de “cola” resultará na anulação da prova de ambos os alunos envolvidos.
2. A interpretação faz parte da questão. Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
3. O tempo de prova é 1:45 h.
4. As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
5. A prova pode ser feita a lápis.

1) Considere uma rotina para inserção de chaves, definidas como números inteiros, em uma tabela de dispersão com endereçamento aberto e função de dispersão h definida como

h(K, i) = ((K % N) + i) % N

onde

K = chave a ser inserida

N = tamanho da tabela de dispersão

i = 0, 1, 2, ... = número da tentativa de inserção da chave na tabela

(a) (1.0 ponto) Assuma que a tabela está originalmente vazia e que N=1007. Mostre a computação da função de dispersão e indique em que posição da tabela cada chave será inserida, nesta ordem: 17, 1024, 18, 1031.

(b) (2.0 pontos) Escreve uma rotina em C para inserção de chaves, definidas como números inteiros, em uma tabela de dispersão com endereçamento aberto com a função de dispersão definida acima. Assuma que as posições são preenchidas, inicialmente, com o valor -1 e que a tabela seja declarada como:

#define N 1007; /\* tamanho da tabela de dispersão \*/

#define VAZIO (-1); /\* indica que a posição está desocupada \*/

int tabela[N]; /\* tabela de dispersão \*/

A função deve ter o seguinte protótipo:

int insere(int K)

e deve retornar:

-1 se a chave não puder ser inserida

0 se a chave já existe

1 se a chave for inserida corretamente

Resposta:

(a) h(17,0) = ((17%1007)+0)%1007 = 17 --- inserida na posição 17

h(1024,0) = ((1024%1007)+0)%1007 = ((17)+0)%1007 = 17 --- colisão

h(1024,1) = ((1024%1007)+1)%1007 = ((17)+1)%1007 = 18 --- inserida na posição 18

h(18,0) = ((18%1007)+0)%1007 = ((18)+0)%1007 = 18 --- colisão

h(18,1) = ((18%1007)+1)%1007 = ((18)+1)%1007 = 19 --- inserida na posição 19

h(2031,0) = ((2031%1007)+0)%1007 = ((24)+0)%1007 = 24 --- inserida na posição 24

(b)

int insere(int K)

{

int p, i;

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

{

p = ((K % N) + i) % N;

if (tabela[p] == VAZIO) /\* Insere a chave \*/

{

tabela[p] = K;

return 1;

}

if (tabela[p] == K) /\* Chave já existe \*/

return 0;

}

return (-1);

}

2) (2.0 pontos) Considere a seguinte sequencia de inteiros: 90, 60, 30, 15, 45.

* 1. (0.5 ponto) Mostre, passo a passo, como o vetor armazenando um *heap* mínimo é construído pela inserção sucessiva destes 5 elementos, na ordem dada. Comente brevemente cada passo do algoritmo de inserção.
  2. (1.5 ponto) Mostre, passo a passo, como fica o vetor após a remoção do segundo menor elemento do heap construído no Item (a). Cada passo deve corresponder a uma operação de inserção ou remoção do *heap* e deve usar apenas o vetor armazenando o *heap*, uma variável auxiliar *T* para trocas e uma variável *R* para devolver o resultado.

Resp:

a)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **Explicação** |
| 90 |  |  |  |  | Acrescente 90 depois do final do heap (que está vazio). | |
| 90 | 60 |  |  |  | Acrescente 60 depois do final do heap. | |
| 60 | 90 |  |  |  | Compare com o pai ((1-1)/2=0) e troque. | |
| 60 | 90 | 30 |  |  | Acrescente 30 depois do final do heap. | |
| 30 | 90 | 60 |  |  | Compare com o pai ((2-1)/2=0) e troque. | |
| 30 | 90 | 60 | 15 |  | Acrescente 15 depois do final do heap. | |
| 30 | 15 | 60 | 90 |  | Compare com o pai ((3-1)/2=1) e troque. | |
| 15 | 30 | 60 | 90 |  | Compare com o pai ((1-1)/2=0) e troque. | |
| 15 | 30 | 60 | 90 | 45 | Acrescente 45 depois do final do heap. | |
| 15 | 30 | 60 | 90 | 45 | Compare com o pai ((4-1)/2=1) e não troque. Pare. | |

b)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |  | T | R | | **Explicação** |
| 15 | 30 | 60 | 90 | 45 |  |  | |  |  |
|  |  |  |  |  |  | 45 | |  | Remova a raiz, 15, trocando-a com o último, 45, usando a variável T. |
| 45 | 30 | 60 | 90 | 15 |  |  | |  | O último passa a estar na posição 3; o elemento salvo na posição 4. |
|  |  |  |  |  |  | 30 | |  | O heap precisa ser rearrumado. Troque 45 com o menor filho, 30, usando a variável T. |
| 30 | 45 | 60 | 90 | 15 |  |  | |  |  |
|  |  |  |  |  |  |  | | 30 | O segundo menor elemento é a nova raiz, 30. Remova 30, devolvendo-a na variável R. Para remover 30, troque-a pelo último, 90. |
| 90 | 45 | 60 |  | 15 |  |  | |  |  |
|  |  |  |  |  |  | 45 | |  | O heap precisa ser rearrumado. Troque 90 com o menor filho, 45, usando a variável T. |
| 45 | 90 | 60 |  | 15 |  |  | |  |  |
| 45 | 90 | 60 | 15 |  |  |  | |  | O último elemento salvo, 15, continua na posição 4. Reinsira 15 no heap, colocando-o depois do final do heap. |
|  |  |  |  |  |  | 90 | |  | O heap precisa ser rearrumado; troque 15 com o seu pai, 90, usando a variável T. |
| 45 | 15 | 60 | 90 |  |  |  | |  |  |
|  |  |  |  |  |  | 15 | |  | O heap precisa ser rearrumado; troque 15 com o seu pai, 45, usando a variável T. |
| 15 | 45 | 60 | 90 |  |  |  | |  | O heap não precisa mais ser rearrumado. Pare. |

(3) (2.0 pontos) Os algoritmos abaixo são usados para resolver problemas de tamanho *n*. Determine a complexidade, no pior caso, de cada algoritmo. Explique sua resposta.

a) (1.0 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) (1.0 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 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(∞)*.

b) O laço mais externo é executado *log2(n)* vezes, visto que *i* dobra a cada passagem. O laço do meio é executado *n/2* vezes, já que *j* é decrementado de 2 em 2. O último laço é executado no máximo *n/2* vezes, para cada interação do laço do meio. Como os laços estão aninhados, o comando

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

é executado *(n/2)2log2(n)* vezes. Logo, a complexidade do algoritmo é *O(n2(log n))*.

(4) (3.0 pontos) Considere um TAD de conjuntos, com a interface apresentada em sala:

typedef struct \_bitvector BitVector;

e a implementação:

struct \_bitvector {

int max;

int \*vector;

};

Implemente em C uma função que computa a união de dois conjuntos. A interface da função é a seguinte:

BitVector\* bvUnion(BitVector\* a, BitVector\* b);

Nota: assuma que os conjuntos a e b foram originalmente inicializados pela função bvInit.

Resposta:

BitVector\* bvUnion(BitVector\* a, BitVector\* b)

{

int i;

int minab = (a->max < b->max) ? a->max : b->max;

int maxab = (a->max > b->max) ? a->max : b->max; /\* c tem o tamanho do maior\*/

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

int maxnum = (maxab-1) /sizeof(int)+1; /\* número de inteiros do maior \*/

BitVector\* t;

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

c->max = maxab;

c->vector = (int\*)malloc(maxnum\*sizeof(int));

for (i=0; i < minnum; i++) /\* computa a união de a e b, até o menor acabar \*/

c->vector[i] = (a->vector[i]) | b->vector[i]);

t = (a->max > b->max)? a : b;

for (i= minnum; i < maxnum; i++) /\* copia o resto de a ou b para c \*/

c->vector[i] = t->vector[i];

return c;

}