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

|  |  |  |
| --- | --- | --- |
| 1a) | 2.0 |  |
| 2a) | 2.0 |  |
| 3a) | 2.0 |  |
| 4a) | 2.0 |  |
| 5a) | 2.0 |  |
|  |  |  |

* A prova é individual e sem consulta.
* A interpretação faz parte da questão.
* O tempo de prova é 1:45 h.
* As respostas devem seguir as questões.
* Caso precise de rascunho use o verso da folha.
* Caso parte da resposta esteja no verso, indique claramente este fato.

1. (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)

int funcao\_C(int n) {

int i, j, sum = 0;

for (i=0; i<n; i++) {

sum+=i;

}

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

sum+=2\*j;

}

return sum;

}

Resp.:

a) 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(∞)*.

Se o enunciado da segunda linha fosse:

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

A resposta seria semelhante a outros exemplos dos exercícios. O laço mais externo é executado *log2(n)* vezes, visto que *i* dobra a cada passagem. O laço do meio e o último laço são executados *n/2* vezes. Como os laços estão aninhados, o comando “sum += (-j \* k) << i/2” é executado *(n/2)2log2(n)* vezes. Logo, a complexidade desse algoritmo é *O(n2(log n))*.

b) Os dois laços são executados *n* vezes; como estão em paralelo, a complexidade deve ser somada, o que resulta em *O(n)*.

1. (2.0 pontos) Uma tabela de dispersão de tamanho 11 é implementada com encadeamento externo através da seguinte função de dispersão:
   1. (1.0 ponto) Desenhe a estrutura de dados após a inserção das chaves (nesta ordem):

7,10,15,14,17,3,21,25

* 1. (1.0 ponto) Explique, com base no exemplo anterior, como remover sucessivamente cada uma das seguintes chaves:

5, 7, 10, 3

Resp:

a)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *x* | *h(x)* |  |  |  |  |
| 7 | 7 |
| 10 | 10 |
| 15 | 4 |
| 14 | 3 |
| 17 | 6 |
| 3 | 3 |
| 21 | 10 |
| 25 | 3 |

|  |  |
| --- | --- |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |  | 14 |  | 3 |  | 25 |
| 4 |  |  | 15 |
| 5 |  |  |  |
| 6 |  |  | 17 |
| 7 |  |  | 7 |
| 8 |  |
| 9 |  |
| 10 |  |  | 10 |  | 21 |

1. Remoção de 5: como 5 não está na estrutura, devolva NULL.

Remoção de 7: como 7 é o único elemento da lista apontada pela entrada 7 da tabela, coloque NULL na posição 7 da tabela.

Remoção de 10: como 10 é o primeiro elemento da lista apontada pela entrada 10 da tabela, remova 10 e faça a entrada 10 da tabela apontar para o elemento que contém 21.

Remoção de 3: remova 3 da lista apontada pela entrada 3 da tabela, fazendo com que o elemento que contém 14 aponte para o elemento que contém 25.

1. (2.0 pontos) Considere a seguinte sequencia de inteiros: 90, 60, 30, 15.
   1. (0.5 ponto) Mostre, passo a passo, como o vetor armazenando um *heap* mínimo é construído pela inserção sucessiva destes 4 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 terceiro menor elemento do heap construído no Item (a). Comente brevemente cada passo do algoritmo de remoção. Cada passo deve corresponder a uma operação de inserção ou remoção do *heap* e usar apenas o vetor armazenando o *heap*, uma variável auxiliar *T* para trocas e uma variável *R* para devolver o resultado.

Resp:

a)

|  |  |  |  |
| --- | --- | --- | --- |
| 90 |  |  |  |
| 90 | 60 |  |  | Acrescente 60 depois do final do heap | | |
| 60 | 90 |  |  | Compare com o pai e troque | | |
| 60 | 90 | 30 |  | Acrescente 30 depois do final do heap | |
| 30 | 90 | 60 |  | Compare com o pai e troque | |
| 30 | 90 | 60 | 15 | Acrescente 15 depois do final do heap |
| 30 | 15 | 60 | 90 | Compare com o pai e troque |
| 15 | 30 | 60 | 90 | Compare com o pai e troque |

b)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |  | T |  |
| 15 | 30 | 60 | 90 |  |  |  |
|  |  |  |  |  | 90 | Remova a raiz, colocando o último na raiz, mas salvando a raiz na última posição do heap. Ou seja, troque a raiz com o último (usa a variável T). |
| 90 | 30 | 60 | 15 |  |  | O último passa a ser a posição 2 do vetor; os elementos salvos estarão a partir da posição 3. |
|  |  |  |  |  | 30 | O heap precisa ser rearrumado: troque 90 com o menor filho (usa a variável T). |
| 30 | 90 | 60 | 15 |  |  |  |
|  |  |  |  |  | 60 | Remova a raiz, colocando o último na raiz, mas salvando a raiz na última posição do heap. Ou seja, troque a raiz com o último (usa a variável T). |
| 60 | 90 | 30 | 15 |  |  | O último passa a ser a posição 1 do vetor; os elementos salvos estão a partir da posição 2. |
|  |  |  |  |  |  | O heap não precisa ser rearrumado.  Remova a raiz, que é o terceiro menor elemento, colocando o último na raiz (usa a variável R). |
| 90 |  | 30 | 15 |  |  | O último passa a ser a posição 0 do vetor. |
|  |  |  |  |  |  | O heap não precisa ser rearrumado. O último elemento salvo, 30, continua na posição 2. Insira 30 no heap, colocando-o depois do final do heap. |
| 90 | 30 |  | 15 |  |  | O último passa a ser a posição 1 do vetor; os elementos salvos estão a partir da posição 3. |
|  |  |  |  |  | 90 | O heap precisa ser rearrumado: troque 90 com o menor filho (usa a variável T). |
| 30 | 90 |  | 15 |  |  |  |
|  |  |  |  |  |  | O penúltimo elemento salvo, 15, continua na posição 3. Insira 15 no heap, colocando-o depois do final do heap. |
| 30 | 90 | 15 |  |  |  | O último passa a ser a posição 2 do vetor; não há mais elementos salvos. |
|  |  |  |  |  | 30 | O heap precisa ser rearrumado; troque 30 com o menor filho (usa a variável T). |
| 15 | 90 | 30 |  |  |  |  |

1. (2.0 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 diferença de dois conjuntos. A função recebe como entrada dois conjuntos, *a* e *b*, e retorna o conjunto *c* formado pelos elementos de *a* que não estão em *b*. A interface da função é a seguinte:

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

Resp:

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

{

int i;

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

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

int tam = (a->max-1)/sizeof(int)+1; /\* c->vetor tem o tamanho de a->vetor \*/

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

c->max = a->max;

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

for (i=0; i < num; i++) /\* computa a diferença (a-b) \*/

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

for (i=num; i < tam; i++) /\* copia o resto de a para c, \*/

c->vector[i] = a->vector[i]; /\* se a for maior do que b \*/

return c;

}

Nota: esta implementação assume que o vetor b foi inicializado por bvInit. Ou seja, os bits de um int não utilizados para representar o conjunto b estão zerados.

1. (2.0 pontos) Considere uma partição dinâmica do conjunto {0,1,2,...8}, representada em uma floresta otimizada com UNION por altura (em caso de empate na altura, a raiz com o maior rótulo deverá ser colocada como filha da outra raiz) e FIND com compressão de trajetória.

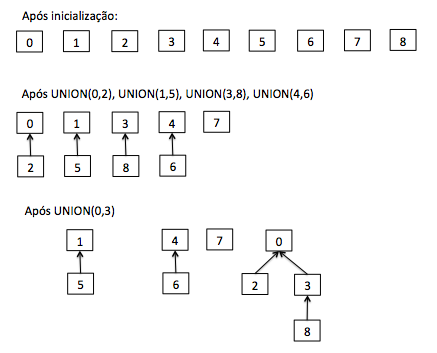
Assuma que a floresta inicialmente resulta da aplicação das operações *MAKE\_SET(i)*, para *i=0,...,8.*

Mostre a floresta após aplicar sucessivamente cada uma das sequencias de operações abaixo:

* 1. (1.0 ponto) UNION(0,2), UNION(1,5), UNION(3,8), UNION(4,6), UNION(0,3)
  2. (1.0 ponto) UNION(1,4), UNION(0,1), FIND(6)

Resp:

(a)



(b)

