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

|  |  |  |
| --- | --- | --- |
| 1a) | 2.0 |  |
| 2a) | 2.0 |  |
| 3a) | 2.0 |  |
| 4a) | 2.0 |  |
| 5a) | 2.0 |  |
|  | 10.0 |  |

1. A prova é individual e sem consulta.
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.
6. (2.0 pontos) Uma tabela de dispersão (*hash table*) de tamanho 11 é implementada com *encadeamento externo* através da seguinte função de dispersão:   
   Nela são inseridas 8 dados que possuem as seguintes chaves de busca (nesta ordem):

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

1. (1.0 ponto) Desenhe a estrutura de dados após a inserção destas chaves.
2. (1.0 ponto) Explique, com base no exemplo anterior, quais são os casos que devem ser considerados para implementar a operação de remoção de uma chave x:

remocao(x)

Entrada: um valor x de chave

Saída: NULL, se x não é encontrada

p, ponteiro para o elemento que contém x

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. Há 4 casos a considerar:

Caso 1: x não está na estrutura. Devolva NULL.

Caso 2: x está no único elemento de uma lista. Remova x e coloque NULL na entrada da tabela.

Caso 3: x está no primeiro elemento p de uma lista. Remova x e faça a tabela apontar para o elemento seguinte a p na lista.

Caso 4: Nenhum dos casos acima. Remova x da lista a que pertencia, como usual.

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. (0.5 ponto) Mostre, passo a passo, como fica o vetor após a remoção do menor elemento e do segundo menor elemento do heap construído no Item (a). Comente brevemente cada passo do algoritmo de remoção.
   3. (1.0 ponto) Descreva como você implementaria a operação

remove(n, Heap\* h)

que remove e devolve os n menores elementos de um heap mínimo h, *sem usar um vetor auxiliar de* n *posições*.

Resp:

a)

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

b)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 15 | 30 | 60 | 90 | Devolva a raiz |
| 90 | 30 | 60 |  | Coloque o último na raiz |
| 30 | 90 | 60 |  | Troque com o menor filho |
| 30 | 90 | 60 |  | Devolva a raiz |
| 60 | 90 |  |  | Coloque o último na raiz |
| 60 | 90 |  |  | Pare: em ordem |

c) Use a mesma ideia do heapsort:

1) Troque o menor elemento, que está na raiz, com o último elemento no vetor que representa o heap.

2) Reorganize o heap, como no algoritmo de remoção.

3) Repita os passos (1) e (2) até retirar os n menores elementos do heap.

4) Os n menores elementos estarão nas n últimas posições do heap.

1. (2.0 pontos) Implemente as seguintes funções de um TAD de Conjuntos (*Set*) pequenos (≤32) cuja interface é dada:

typedef unsigned int Set;

1. (1.0 ponto) set\_remove

/\* remove o i-ésimo elemento do conjunto s \*/

Set set\_remove(Set s, int i);

1. (1.0 ponto) set\_diferenca

/\* entrada: conjuntos r e s

saída: o conjunto t formado pelos elementos de r que não estão em s

\*/

Set set\_diferenca(Set r, Set s);

*A sua implementação deverá ser a mais simples possível.*

Dicas:

1. O operador k<< i, desloca os bits de k de i posições para a esquerda.
2. O operador ~b inverte os bits de b.

Resp:

a)

Set set\_remove(Set s, unsigned int i){

int mask = 1 << i;

mask = ~mask;

s = (s & mask);

return s;

}

b)

Set set\_diferenca(Set r, Set s){

return ((r & ~s));

1. (2.0 pontos) Considere a representação de partições dinâmicas por florestas, com a implementação de UNION por altura e FIND com compressão de caminhos (como explicado em sala).
2. (1.0 ponto) Desenhe a floresta resultante das operações de UNION, executadas na seguinte ordem:

UNION(0,2), UNION(1,5), UNION(3,8), UNION(4,6),

UNION(0,3), UNION(1,4), UNION(0,1)

1. (1.0 ponto) Partindo da floresta obtida no item (a), desenhe a floresta resultante das operações de FIND, executadas na seguinte ordem:

FIND(6), FIND(5)

Resp:

a)

0

2

3

8

1

5

4

6

0

2

1

5

3

8

4

6

0

2

3

8

1

5

4

6

b)

0

2

3

8

1

5

4

6

FIND(6)

0

2

3

8

1

5

4

6

FIND(5)

1. (2.0 pontos) Qual a ordem da complexidade dos algoritmos A e B abaixo? Explique como você chegou a tal resultado.
2. (1.0 ponto)

int funcao\_A(int n) {

int i, j, sum = 0;

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

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

sum+=i+2\*j;

}

}

return sum;

}

1. (1.0 ponto)

void B (int\* vet, int n)

{

int i, j, pos;

for (i=0; i<n-1; i++)

{

pos = i;

for (j=i+1; j < n; j++)

if (vet[pos] > vet[j])

pos = j;

if (pos != i) {

int tmp = vet[pos];

vet[pos]=vet[i];

vet[i] = tmp;

}

}

}

Resp:

a) O loop mais externo é executado log3(n) vezes e o mais interno log2(n) vezes. Logo a função A tem complexidade de tempo O(log2(n)).

(Este programa não implementa nada).

b) O loop mais externo é executado n vezes e o mais interno n vezes. Logo a função B tem complexidade de tempo O(n2).

(Este programa implementa o selection sort).