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

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

* 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.

**Questão 1** (2.0 pontos): Os algoritmos abaixo são usados para resolver problemas de tamanho *n*. Determine a complexidade de cada algoritmo, no pior caso (na notação O(.)). 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;

}

}

}

**Resposta**

O tempo de execução do laço mais externo é proporcional a log2, visto que i dobra a cada passagem.

O tempo de execução do laço do meio é semelhante, exceto que j é dividido pela metade, começando em n.

O tempo de execução do último laço é proporcional a n/2.

Como os laços estão aninhados, devem-se multiplicar todos eles. Logo, a complexidade desse algoritmo é O(n(log n)2).

b) (1.0 ponto):

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

for( int j = 0; j < i; j++ ) {

for( int k = 0; k < n; k += 2 ) {

sum \*= k;

}

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

sum /= k

}

}

}

**Resposta**

Os tempos de execução dos dois laços mais internos são proporcionais a n e log n, respectivamente. Como estão em paralelo, a complexidade deve ser somada, ou seja, o tempo de execução dos dois laços, quando combinados, é dominado pelo custo do primeiro, ou seja, proporcional a n.

O tempo de execução do laço mais externo é proporcional a log n, visto que os passos dobram a cada passagem.

Já o laço do meio é sequencial, ou seja, o tempo de execução é proporcional a n.

Assim a complexidade combinada dos laços é O(n2 log n).

**Questão 2** (2.0 pontos): Uma tabela de dispersão de tamanho 11 é implementada com encadeamento interior, utilizando todo o espaço de endereçamento para tratar colisões (ou seja, sem área de overflow). A função de dispersão é a seguinte:

* 1. (0.5 ponto) Desenhe a estrutura de dados após a inserção das chaves (nesta ordem):

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

Explique como cada colisão foi tratada.

* 1. (1.0 ponto) Explique como deve ser executada uma operação de remoção para que as consultas e inserções feitas em seguida executem corretamente. Mostre como ficará a estrutura após a remoção da chave 3.
  2. (0.5 pontos) Na estrutura resultante do item (b), insira a chave 4.

**Resposta**

a)

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

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 21 | *7* | Houve colisão na posição 10; a chave 21 é inserida na primeira posição vazia |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 | 14 | *4* |  |
| 4 | 15 | *3* |  |
| 5 | 3 | *6* | Houve colisão na posição 3; a chave 3 é inserida na primeira posição vazia |
| 6 | 17 | *5* |  |
| 7 | 7 | *1* |  |
| 8 | 25 | *8* | Houve colisão na posição 3; a chave 25 é inserida na primeira posição vazia |
| 9 |  |  |  |
| 10 | 10 | *2* |  |

b) As posições nunca ocupadas são marcadas com BRANCO e as previamente ocupadas por chaves removidas são marcadas com -1.

A remoção de 3: como 3 não está na posição 3, pesquise por 3 (com “loop back” para 0 quando chegar a 10) até achar uma posição que nunca foi ocupada ou até achar 3. Remova 3 então da posição 5, marcando-a com -1.

c) Inserção de 4: como a posição 4 está ocupada com a chave 15, procure a primeira posição seguinte (com “loop back” para 0 quando chegar a 10) marcada com BRANCO ou com -1; neste caso será a posição 5. Insira 4 nesta posição.

**Questão 3** (3.0 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. (1.0 ponto) Mostre, passo a passo, como o vetor armazenando um *heap* mínimo é construído de forma otimizada com estes 4 elementos. Comente brevemente cada passo do algoritmo otimizado de construção.
  3. (0.5 ponto) Considere o vetor construído no item (a). Mostre, passo a passo, como fica o vetor após a remoção do menor elemento e do segundo menor elemento do heap, usando o algoritmo de remoção apresentado em sala. Comente brevemente cada passo destas remoções.
  4. (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*.

**Resposta**

a)

|  |  |  |  |
| --- | --- | --- | --- |
| *0* | *1* | *2* | *3* |
| 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)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 90 | 60 | Acrescente metade dos elementos ao final o vetor. | | |
|  | 30 | 90 | 60 | Acrescente 30 na primeira posição livre. | | |  | | |
|  | 30 | 90 | 60 | Compare 30 com os filhos nas posições 2 e 3. | | |  | | |
|  |  |  |  | Como 30>90 e 30>60, não há trocas. | | |  | |
| 15 | 30 | 90 | 60 | Acrescente 15 na primeira posição livre. | | |  | |
|  |  |  |  | Compare 15 com os filhos nas posições 1 e 2. | | |  | |
|  |  |  |  | Como 15>30 e 15>90, não há trocas. | | |  | |
|  |  |  |  | O processo para. | | |  |
|  | | | | |  |

c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

d) 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.

**Questão 4** (3.0 pontos): Considere o problema de representar as turmas de uma disciplina em um dado semestre. Assuma que cada aluno está matriculado em uma única turma. Considere duas operações: (1) determinar se dois alunos estão na mesma turma; (2) juntar duas turmas em uma 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.

c) (2,0 pontos) Defina em C a estrutura de dados escolhida no item (a) e implemente uma rotina para determinar se dois alunos estão na mesma turma. A rotina 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, com balanceamento de união e compressão de caminhos no Find, como no algoritmo Union-Find. Para determinar se dois alunos estão na mesma turma, basta realizar o Find para cada uma delas e verificar se os Finds retornam raízes iguais.

b)

forest\_node\* Find\_same(forest\_node\* node1, node2) {

forest\_node\* temp;

forest\_node\* root1 = node1;

forest\_node\* root2 = node2;

while (root1-> parent != NULL) {

root1 = root1-> parent;

}

while (node1->parent != root1) {

temp = node1->parent;

node1->parent = root1;

node1 = temp;

}

while (root2-> parent != NULL) {

root2 = root2-> parent;

}

while (node2->parent != root2) {

temp = node2->parent;

node2->parent = root2;

node2 = temp;

}

return root1 == root2;

}