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

|  |  |  |
| --- | --- | --- |
| 1a) | 1.5 |  |
| 2a) | 1.5 |  |
| 3a) | 2.0 |  |
| 4a) | 2.5 |  |
| 5a) | 2.5 |  |
|  | 10.0 |  |

**LEIA COM CUIDADO**

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 e guardados fora do alcance durante toda a prova. **Aparelhos celulares ligados ou de alguma forma visíveis 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 (1.5 pontos)**

a) **(1.0 ponto)** Considere a tabela de dispersão estendida, *T*, parcialmente representada abaixo. Quantos apontadores em *T* apontam para o bloco *B1* ? Caso haja mais de um apontador, além do ilustrado na figura, em quais posições da tabela *T* eles estarão armazenados? Explique sua resposta.

*T*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =3* |  |  |  |
| 0 |  |  |  | *B1* |
| 1 |  |  |  | *j1 =2* | |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| 6 |  |  |  |  |
| 7 |  |  |  |  |

b) **(0.5 pontos)** Considere a tabela de dispersão estendida, *T*, representada na figura abaixo. Suponha que cada bloco possa conter até 3 chaves. Mostre como a estrutura ficará após a remoção de *K31*, assumindo que a estrutura deve economizar espaço sempre que possível. Explique sua resposta.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =2* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  | *K12* | *j1 =2* | |
| 2 |  |  |  |  | |
| 3 |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  | *K22* | *j2 =2* | |
|  |  |  |  |  | |
|  |  |  |  |  |
|  |  |  | *K31* | *B3* |
|  |  |  | *K32* | *j3 =2* | | |
|  |  |  |  |  | | |
|  |  |  |  |  |
|  |  |  | *K41* | *B4* |
|  |  |  | *K42* | *j4 =2* | |
|  |  |  |  |  | |

***Resposta***

a) Com *i = 3* e *j1 = 2*, há *p = 2(3-2)= 2* apontadores para *B1* . O segundo apontador estará em *T[4]* pois as chaves mapeadas em *T[0]* e *T[4]* possuem os 2 últimos bits iguais.

b) Após a remoção de *K31*, a chave *K32* deve ser movida para *B1* e o bloco *B3* descartado. De fato, *B1* tem 1 posição livre e os blocos *B1* e *B3* são “companheiros”, no sentido de que as chaves mapeadas em *T[0]* e *T[2]* possuem o último bit igual.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *i =2* |  |  |  |
| 0 |  |  | *K11* | *B1* |
| 1 |  |  | *K12* | *j1 =1* | |
| 2 |  |  | *K32* |  | |
| 3 |  |  |  |  |
|  |  |  | *K21* | *B2* |
|  |  |  | *K22* | *j2 =2* | |
|  |  |  |  |  | |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  | | |
|  |  |  |  |  | | |
|  |  |  |  |  |
|  |  |  | *K41* | *B4* |
|  |  |  | *K42* | *j4 =2* | |
|  |  |  |  |  | |

**Questão 2 (1.5 pontos)**

1. **(0.5 pontos)** Crie uma árvore 2-3 através da inserção, na ordem, das chaves: 1, 12, 8, 2, 6. Mostre todos os passos.
2. **(1.0 ponto)** Remova a chave 3 da árvore 2-3 abaixo, de 3 formas diferentes. Mostre e explique todos os passos.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 3 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 |  | 6 |  |  | 32 |  |

***Resposta***

(a)

|  |  |
| --- | --- |
| 1 |  |

|  |  |
| --- | --- |
| 1 | 12 |

|  |  |  |
| --- | --- | --- |
| 1 | 8 | 12 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 8 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  |  | 12 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 8 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 |  | 12 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 8 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 6 |  | 12 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 2 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  | 6 |  |  | 12 |  |

(b) Remoção de 3:

Árvore inicial:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 3 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 |  | 6 |  |  | 32 |  |

1a) Troque 3 com a sua sucessora, 6, e remova 3:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 6 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 |  |  |  |  | 32 |  |

2a) Elimine o nó vazio, movendo a maior chave do pai para o filho à direita.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 6 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 |  | 8 | 32 |

1b) Alternativamente, no passo (1a), mova a chave 6 para o nó vazio e a chave 2 para o pai:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 2 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  | 6 |  |  | 32 |  |

1c) Alternativamente, no passo inicial, troque 3 com a sua antecessora, 2, e remove 3:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 2 | 8 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  | 6 |  |  | 32 |  |

**Questão 3 (2.0 pontos)** Escreva uma rotina em C que receba como entrada um *min heap* e um float *p* maior do que 1.0, modifique a prioridade *x* do menor elemento do *min heap* para (*p* × *x*) e reorganize a estrutura para que mantenha as propriedades de um *min heap*. A rotina deve retornar 1, se a alteração ocorreu corretamente, e 0, em caso contrário.

A sua implementação deve fazer o menor número possível de comparações e movimentações. A implementação não deve usar as operações sobre *min heaps* pré-definidas, mas poderá usar a função troca definida abaixo.

Assuma que o *min heap* seja definida como abaixo:

struct \_heap {

int max; /\* tamanho maximo do heap \*/

int pos; /\* proxima posicao disponivel no vetor \*/

float\* prioridade; /\* vetor das prioridades \*/

};

typedef struct \_heap Heap;

Assuma ainda que a assinatura da rotina seja:

int heap\_altera(Heap\* heap, float p)

A função troca é definida como:

static void troca(int a, int b, float\* v) {

float f = v[a];

v[a] = v[b];

v[b] = f;

}

***Resposta***

A rotina apenas aumenta a prioridade do elemento de menor prioridade (o primeiro elemento) e corrige o *min heap*, chamando a rotina corrige\_abaixo.

int heap\_altera(Heap\* heap, float p)

{

if (heap->pos>0) {

heap->prioridade[0]=heap->prioridade[0]\*p;

corrige\_abaixo(heap);

return 1;

}

else return 0;

}

Como apresentado em sala, a rotina corrige\_abaixo para *min heaps* é:

static void corrige\_abaixo(Heap\* heap){

int pai=0;

while (2\*pai+1 < heap->pos){

int filho\_esq = 2\*pai+1;

int filho\_dir = 2\*pai+2;

int filho;

if (filho\_dir >= heap->pos) filho\_dir = filho\_esq;

if (heap->prioridade[filho\_esq] < heap->prioridade[filho\_dir])

filho = filho\_esq;

else filho = filho\_dir;

if (heap->prioridade[pai] > heap->prioridade[filho])

troca(pai,filho,heap->prioridade);

else break;

pai = filho;

}

}

**Questão 4** **(2.5 pontos)** Considere o problema de representar a que partido político cada pessoa de um conjunto de pessoas pertence. Assuma que: (1) uma pessoa só pertence a um partido em um dado instante; (2) dois partidos *A* e *B* podem se fundir (ou seja, para efeitos desta questão, as pessoas que pertencem a *A* ou *B* passam a pertencer ao mesmo partido).

a) **(0.5 pontos)** Qual estrutura de dados, dentre as apresentadas na disciplina, seria mais eficiente para representar a situação acima e implementar as rotinas dos itens (b) e (c)? Explique sua resposta. Defina em C a estrutura de dados escolhida, ou seja, o typedef, struct, etc... necessários às rotinas dos itens (b) e (c).

b) **(1.0 pontos)** Usando a estrutura de dados escolhida no item (a), implemente em C uma rotina para determinar se duas pessoas são do mesmo partido. A rotina não deverá chamar as operações definidas em sala para a estrutura escolhida.

c) **(1.0 pontos)** Usando a estrutura de dados escolhida no item (a), implemente em C uma rotina para fundir dois partidos, como definido acima. 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 e com balanceamento de união e compressão de caminhos no Find (algoritmo Union-Find). Para determinar se duas pessoas são do mesmo partido, basta realizar o Find para cada uma delas e verificar se os Finds retornam a mesma raiz. Para fundir 2 partidos, basta executar a operação de Union.

As definições em C seriam:

typedef struct forest\_node\_t {

void\* value;

struct forest\_node\_t\* parent;

int rank;

} forest\_node;

b) As pessoas a serem pesquisadas são representadas por node1 e node2.

forest\_node\* Find\_same(forest\_node\* node1, forest\_node\* 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;

}

1. Os partidos representados por node1 e node2 são fundidos.

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)** Implemente uma função em C que receba como entrada um grafo não dirigido *G=(V,E)*, com pelo menos 1 nó, e retorne o diâmetro do grafo. A função deve ter o seguinte protótipo:

int diametro(Graph\* G)

Considere que o grafo está representado pela sua matriz de adjacências:

typedef struct graph Graph;

struct graph {

int nv; /\* número de nós no grafo \*/

int\*\* adj; /\* matriz de adjacências do grafo \*/

};

A função deverá se basear no algoritmo de grafos apresentado em sala que melhor se adequa para a tarefa, supondo que o grafo é esparso, ou seja, *G=(V,E)* é tal que |*E*| é muito menor do que |*V*|2.

Não é necessário incluir o algoritmo na solução, mas indique o protótipo do algoritmo adotado e qualquer outra estrutura de dados necessária para a sua implementação. A sua implementação não deverá modificar a estrutura de dados passada como entrada.

Argumente porque a sua implementação da função diametro está correta, prove qual é o custo da sua implementação e argumente porque o algoritmo escolhido é o que melhor se adequa para a tarefa.

***Resposta***

* Dijkstra:

int diametro(Graph\* g)

{

int i, k, n;

int dist[g->nv];

diam = 0;

for (i=0; i < g->nv; i=i+1)

{

Dijkstra(g,i,dist);

for (i=0; i < g->nv; i=i+1)

if (dist[i] > diam) diam = dist[i];

}

return diam;

}

A função usa o algoritmo de Dijkstra como subrotina, com o seguinte protótipo:

void Dijkstra(Graph\* g, int n, int\* dist);

Dijkstra(g,i,dist) retorna em dist o menor caminho entre i e cada um dos outros nós do grafo. Portanto, a função acima retorna o maior caminho mínimo entre dois nós do grafo, ou seja, calcula o diâmetro do grafo.

O custo do algoritmo de Dijkstra é *O(|V|2)* e, portanto, o custo de diametro será *O(|V|3)*.

Como o grafo é esparso, Dijkstra é a melhor opção (vide notas de aula).

* Warshall não é a melhor opção pois o grafo é esparso.