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

|  |  |  |
| --- | --- | --- |
| 1a) | 2.0 |  |
| 2a) | 1.0 |  |
| 3a) | 1.0 |  |
| 4a) | 3.0 |  |
| 5a) | 3.0 |  |
|  |  |  |

1. A prova é individual e sem consulta.
2. A interpretação faz parte da questão.
3. O tempo de prova é 1:30 h.
4. As respostas devem seguir as questões.
5. Caso precise de rascunho use o verso da folha.
6. Caso parte da resposta esteja no verso, indique claramente este fato.
7. **(2.5 pontos)**
   1. **(2.0 ponto)** Complete o código da função parcialmente abaixo para que realize inserção de chaves em uma tabela de dispersão, utilizando desempate interno de colisões:

**#define MAX 8 /\* tamanho da tabela de hash \*/**

**#define DEC 2**

**#define VAZIO (-1)**

**int hash[MAX];**

**int insere (int x)**

**{**

**int k, pos;**

**pos = x % MAX;**

**(COMPLETAR)**

**return (-1); /\* não inseriu \*/**

**}**

c) **(0.5 ponto)** Escreva a expressão matemática da função de dispersão H(x,k) implicitamente utilizada na rotina acima. Por que H(x,k) não é uma boa função de dispersão? Justifique sua resposta.

Resp.:

(a)

#define MAX 8 /\* tamanho da tabela de hash \*/

#define DEC 2

#define VAZIO (-1)

int hash[MAX];

int insere (int x)

{

int k, pos;

pos = x % MAX;

for (k = 0; k < MAX; k++)

{

if (hash[pos] == VAZIO) /\* entrada vazia \*/

{

hash[pos] = x; /\* insere a chave \*/

return (pos);

}

if (hash[pos] == x) /\* chave duplicada \*/

return (pos);

pos = pos - DEC; /\* colisão \*/

if (pos < 0)

pos = pos + MAX;

}

return (-1); /\* não inseriu \*/

}

(b) A função de hash H(x,k) = (x + k\*DEC) % MAX não é adequada pois MAX deveria ser um número primo não próximo a uma potência de 2, ou um número sem divisores primos menores que 20, para evitar colisões. O valor do decremento, DEC, também não deveria ser um submúltiplo do tamanho da tabela de hash, MAX.

**2) (2.5 pontos)** Considere uma tabela de dispersão estendida, parcialmente representada na figura abaixo, onde

i = número de bits de relevância

h = função de hash

H = tabela de hash

B = um bucket arbitrário

jB = informação adicional sobre B

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

i

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

jB

H

B

* 1. **(2.0 pontos)** Suponha que o bucket *B* está completo. Explique todos os casos de uma inserção de uma nova chave *K* em *B*, supondo que possam haver chaves duplicadas
  2. **(0.5 ponto)** Suponha que cada bucket contenha *b* chaves. Qual o número mínimo, *nB*, de buckets e o tamanho mínimo, *hmin*, da tabela *H* para acomodar *N* chaves, supondo que não hajam chaves duplicadas?

Resp.:

a) Quando o bucket *B* está completo e *i-jB* > 0, o número de bits de relevância incrementado deve ser incrementado, um novo bucket *B’* deve ser criado e chaves redistribuídas entre *B* e *B’*. Este processo deve ser repetido até seja possível armazenar *K* ou *i-jB* = 0

Quando o bucket *B* está completo e *i-jB* = 0, a tabela de hash *H* deve ser duplicada, e o processo anterior repetido.

Quando dois registros podem ter a mesma chave (passada como entrada para a função de hash), então o processo descrito anteriormente pode não ser suficiente para resolver o problema de overflow. Uma estratégia de encadeamento de buckets deverá ser então implementada.

b) O número mínimo, *nB*, de buckets é dado simplesmente por: *nB* = ⎡N/b⎤

Supondo-se que não há chaves duplicadas, o tamanho mínimo da tabela *H*, *hmin*, será a menor potência de 2 maior do que *nB*.

**3)** **(2.5 pontos)**

a) **(0.5 ponto)** Insira a chave 31 na árvore B de ordem 4 abaixo:

50

30 70 90

10 20 39 40 60 80 100

Descreva, passo a passo, todas as modificações sofridas, redesenhando apenas a parte da árvore modificada a cada passo. Justifique porque não há mais modificações a fazer.

b) **(0.5 ponto)** Insira a chave 31 na árvore 2-3 abaixo:

50

30 70 90

10 20 39 40 60 80 100

Descreva, passo a passo, todas as modificações sofridas, redesenhando apenas a parte da árvore modificada a cada passo. Justifique porque não há mais modificações a fazer.

c) **(0.5 ponto)** Remova a chave 70 da árvore B de ordem 4 abaixo:.

50

30 33 70 90

10 20 31 32 40 41 55 60 81 80 94 100

Descreva, passo a passo, todas as modificações sofridas, redesenhando apenas a parte da árvore modificada a cada passo. Justifique porque não há mais modificações a fazer.

d) **(1.0 ponto)** Remova a chave 70 da árvore B de ordem 3 abaixo:.

50

30 70 90

10 20 40 60 80 100

Descreva, passo a passo, todas as modificações sofridas, redesenhando apenas a parte da árvore modificada a cada passo. Justifique porque não há mais modificações a fazer.

Resp.:

a) Insira a chave 31 na árvore B de ordem 4 abaixo:

50

30 70 90

10 20 31 39 40 60 80 100

A inserção pode ser feita sem modificações nos nós pois uma árvore B de ordem 4 admite

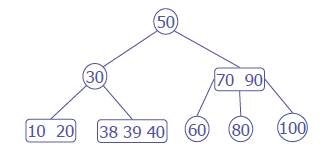
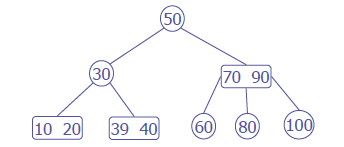
no máximo

*kmax* = (4-1)=3

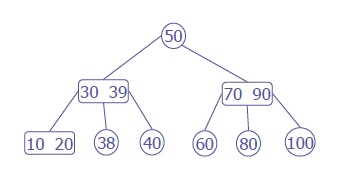
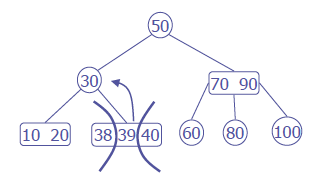
chaves por nó.

b) Insira a chave 31 na árvore 2-3 abaixo:

(Ver notas de aula sobre árvores 2-3, página 11, trocando 38 por 31)



31



39

39

31

c) Remova a chave 70 da árvore B de ordem 4 abaixo:.

50

30 33 70 90

10 20 31 32 40 41 55 60 81 80 94 100

Troque 70 a sua sucessora 81.

50

30 33 80 90

10 20 31 32 40 41 55 60 70 80 94 100

Remova 70.

50

30 33 80 90

10 20 31 32 40 41 55 60 80 94 100

Não é necessário fazer qualquer outra alteração pois uma árvore B de ordem 4 admite no mínimo

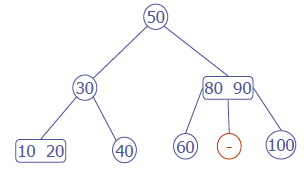
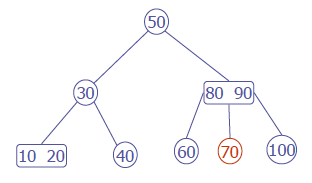
*kmin* = (4/2-1)=1

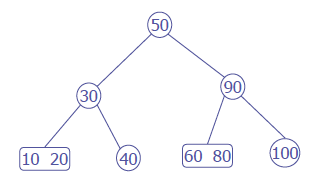
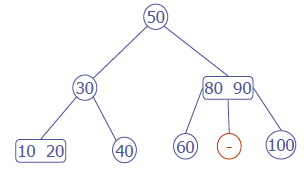
chaves por nó.

d) Remova a chave 70 da árvore B de ordem 3 abaixo:.

(Ver notas de aula sobre árvores 2-3, página 30)

Troque 70 com 80 inicialmente. Em seguida, prossiga da seguinte forma:

****



**4) (2.5 pontos)**

* 1. **(0.5 ponto)** Explique como determinar quantas componentes conexas um grafo não dirigido possui. Use a figura abaixo para ilustrar a sua solução.

0

1

2

3

4

5

6

7

* 1. **(2.0 pontos)** Implemente uma função em C que determine o número de componentes conexas de um grafo não dirigido. A função deve ter o seguinte protótipo:

int nconnected(Graph\* G);

Considere que o grafo está representado como listas de adjacências:

typedef struct graph Graph;

typedef struct listNode ListNode;

struct listNode {

int vertex;

ListNode\* link;

};

struct graph {

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

int\* vis; /\* vis[i]=1 sse vertice i foi visitado \*/

ListNode\*\* vv; /\* vv[i] aponta p/ lista de vértices adjacentes a i \*/

};

Resp.:

1. O procedimento será:
   1. Inicie um percurso em profundidade (ou em amplitude) a partir de qualquer nó. A primeira componente conexa do grafo será composta de todos os nós visitados.
   2. Se todos os nós não foram visitados, escolha um nó não visitado e inicie novamente o procedimento em (1) para determinar a segunda componente conexa, e assim por diante.
   3. Pare quando todos os nós tenham sido visitados. O número de componentes conexas será o número de vezes que o procedimento em (1) foi executado.
2. O programa utiliza a rotina dfs apresentada em sala:

(g->vis[n] = j indica que o nó n pertence à j-ésima componente conexa do grafo)

void dfs(Graph\* g, int N, int j){

ListNode\* w;

g->vis[N] = j;

for (w = g->vv[N]; w != NULL; w = w->link)

if (g->vis[w->vertex] == 0) dfs(g, w->vertex, j);

}

int nconectado(Graph\* g){

int i, cont=0;

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

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

if (g->vis[i] == 0){

dfs(g,i,cont);

cont++;}

return cont;

}