Universidade Federal do Ceará
– UFC Centro de Ciências –
CC
Mestrado e Doutorado em Ciências da Computação - MDCC
Estruturas de Dados Avançadas

Exercício: Listas, Complexidade de Algoritmos, Tabelas de Dispersão, Hashing Dinâmico, Heap, Conjuntos e Partições
Objetivos: Exercitar os conceitos de Listas, Complexidade de Algoritmos, Tabelas de Dispersão, Hashing Dinâmico, Heap, Conjuntos e Partições
Data da Entrega: 30/04/2017

OBS 1: Exercício Individual.
OBS 2: A entrega desta lista deverá ser executada via repositório de código..

NOME: Renan Pereira de Figueiredo MATRÍCULA: 396921

1) Escreva uma função de dispersão (*hash*) que tenha como chave um *string* com os nomes de países, de até 32 caracteres. A tabela *hash* deve ter 513 elementos.

```
hashFunction(k: String): Int
    while(i < k.length)
        sum += ASCII(k(i))
    return sum % 513
```

2) No encadeamento interior, em que as chaves são armazenadas na própria tabela de *hash*, forneça um exemplo em que duas chaves distintas, x e y, com $h(x) \neq h(y)$ podem apresentar colisões. Exemplifique com uma tabela *hash* de 11 elementos. Defina uma função de *hash* para tornar sua explicação mais concreta.

Sendo a função hash h(x) = x%11 para x = {11, 5, 8, 16, 17}, teremos:

h(11) = 0, h(5) = 5, h(8) = 8, h(16) = 5, h(17) = 6

11					5	16	17	8		
0	1	2	3	4	5	6	7	8	9	10

Como pode ser observado, temos um exemplo de colisão secundária na tabela acima, pois 16 e 17 com h(16) != h(17), apresentaram colisão. Isso ocorre porque o h(16) = 5 mas como a posição 5 do vetor já estava ocupada com o elemento 5 que tem o mesmo hash de 16, então 16 passou a ocupar a próxima posição livre do vetor, nesse caso, a posição 6. Contudo, ao inserir o elemento 17, cujo hash é 6, temos uma colisão pois a posição 6 já está ocupada com o element 16 que não tem o mesmo hash que 17.

3) Explique com base no seu exemplo anterior porque ao remover uma chave de uma tabela *hash*
você não pode simplesmente apagar a entrada da tabela.

Para remover uma chave da tabela não se deve apagar a sua entrada da tabela porque isso tornaria impossível recuperar todos os elementos que ao inserir

tinham sofrido uma colisão com essa entrada, devido ela já estar ocupada.

4) Mostre, através de um desenho, como ficaria a tabela *hash* de 7 elementos que recebesse as seguintes chaves de busca 7,10,15,14,17,16 (nesta ordem).

   a) Se ela fosse com encadeamento exterior com a função de dispersão: $h(x) = (13 * x)\ \%\ 7$

   h(7) = 91 % 7 = 0; h(10) = 130 % 7 = 4; h(15) = 195 % 7 = 6; h(14) = 182 % 7 = 0; h(17) = 221 % 7 = 4; h(16) = 208 % 7 = 5.

14	**=> 7**
17	**=> 10**
16	
15	

   b) Se ela fosse com encadeamento aberto e com segunda função de dispersão:

   $h(x, j) = (13x + 5j)\ \%7, j = 0,1,2, \ldots$

   h(7,0) = (91 + 0) % 7 = 0;

   h(10,0) = (130 + 0) % 7 = 130 % 7 = 4

   h(15,0) = (195 + 0) % 7 = 195 % 7 = 6

   h(14, 0) = (182 + 0) % 7 = 182 % 7 = 0 => h(14,1) = (182 + 5) % 7 = 187 % 7 = 5

   h(17, 0) = (221 + 0) % 7 = 221 % 7 = 4 => h(17,1) = (221 + 5) % 7 = 226 % 7 = 2

   h(16, 0) = (208 + 0) % 7 = 208 % 7 = 5 => h(16,1) = (208 + 5) % 7 = 213 % 7 = 3

7
17
16
10
14
15

5) A seguinte função implementa a inserção de chaves inteiras positivas em uma tabela de dispersão com desempate interno de colisões:

```
#define MAX      8
#define VAZIO   (-1)
int hash[MAX];

int insere (int x)
{
   int pos, k;

   pos = x;
   for (k = 0; k < MAX; k++)
   {
      pos = (pos + k) % MAX;

      if (hash[pos] == x)        /* Chave duplicada - não insere */
         return (pos);

      if (hash[pos] == VAZIO)    /* Posição livre */
      {
         hash[pos] = x;
         return (pos);
      }
   }
   return (-1);
}
```

Supondo a tabela inicialmente preenchida com o valor VAZIO em todas as posições, responda:

a) Insira as chaves 4, 12, 20, 28, 36, 44, 52 e 60 na tabela.

4 => pos = (4 +0) % 8 = 4

12 => pos = (12 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

20 => pos = (20 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

28 => pos = (28 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

pos = (7 + 3) % 8 = 2

36 => pos = (36 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

pos = (7 + 3) % 8 = 2

pos = (2 + 4) % 8 = 6

44 => pos = (44 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

pos = (7 + 3) % 8 = 2

pos = (2 + 4) % 8 = 6

pos = (6 + 5) % 8 = 3

52 => pos = (52 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

pos = (7 + 3) % 8 = 2

pos = (2 + 4) % 8 = 6

pos = (6 + 5) % 8 = 3

pos = (3 + 6) % 8 = 1

60 => pos = (60 + 0) % 8 = 4

pos = (4 + 1) % 8 = 5

pos = (5 + 2) % 8 = 7

pos = (7 + 3) % 8 = 2

pos = (2 + 4) % 8 = 6

pos = (6 + 5) % 8 = 3

pos = (3 + 6) % 8 = 1

pos = (1 + 7) % 8 =

60	**52**	**28**	**44**	**4**	**12**	**36**	**20**

b) Qual a função de dispersão H(x, k) implicitamente utilizada no algoritmo?
H(x,k) = (x + k) % 8; k = 0,1,...,7

c) Trata-se de uma função linear ou quadrática em k?
Trata-se de um função linear em k, pois k aumenta sempre mais 1 unidade a cada interação.

d) Em que situações a função retornará um valor menor que 0? Por quê?
Se sair do loop do for sem retornar nada. Isso porque não existe mais espaço livre no vetor.

e) Para quais valores de MAX é possível assegurar a varredura integral da tabela?

6) Quantas colisões existem ao se inserir dados dos n primeiros números naturais em uma tabela de dimensão m, usando as seguintes funções hash:

a) n mod 5
n-5 colisões

b) n mod 7
n-7 colisões

c) n mod 35
n-35 colisões

d) n mod (m/5)
n – (m/5) colisões

e) n mod m
n - m colisões

7) Em uma tabela hash de tamanho m=12, assumindo uma função de hash h'(x) = x mod m:

a) Insira as seguintes chaves: 4, 10, 24, 36, 25, 14, 23, 100, 2, 3, 12, 13, Caso haja colisão utilize a técnica de armazenamento interno, utilizando para isso uma função de incremento linear, com passo 2.

24	**36**	**25**	**14**	**4**	**100**	**2**	**3**	**12**	**13**	**10**	**23**

Função de Incremento Linear: hi(x,j) = (x + j) mod 12, j = 1...11 (Usado após colisão no uso da primeira função)

h(4) = 4 mod 12 = 4

h(10) = 10 mod 12 = 10

h(24) = 24 mod 12 = 0

h(36) = 36 mod 12 = 0 => hi(36,1) = 37 mod 12 = 1

h(25) = 25 mod 12 = 1 => hi(25,1) = 26 mod 12 = 2

h(14) = 14 mod 12 = 2 => hi(14,1) = 15 mod 12 = 3

h(23) = 23 mod 12 = 11

h(100) = 100 mod 12 = 4 => hi(100,1) = 101 mod 12 = 5

h(2) = 2 mod 12 = 2 => hi(2,1) = 3 mod 12 = 3 => hi(2,2) = 4 => hi(2,3) = 5 => hi(2,4) = 6

h(3) = 3 mod 12 = 3 => hi(3,1) = 4 mod 12 = 4 => hi(3,2) = 5 => hi(3,3) = 6 => hi(3,4) = 7

h(12) = 12 mod 12 = 0 => hi(12,1) = 13 mod 12 = 1 => hi(12,2) = 2 =>...=> hi(12,8) = 8

h(13) = 13 mod 12 = 1 => hi(13,1) = 14 mod 12 = 2 => hi(13,2) = 3 =>...=> hi(13,8) = 9

b) Escreva a expressão matemática correspondente.

8) Esta é uma boa função de Hash? Justifique detalhadamente a sua resposta.
Não, pois a maioria das chaves são pares e como m também é par então a maioria dos índices produzidos pela função modular também será par, logo teremos uma grande possibilidade de colisão entre os números pares.

9) Considere um TAD de hash com a interface:

```
typedef struct hash Hash;

Hash* hsh_cria(int n, int hash_func(char*), void free_func(void*));
int   hsh_insere(Hash* tabela, void* info, char* chave);
int   hsh_remove(Hash* tabela, char* chave);
void* hsh_busca(Hash* tabela, void* chave);
Hash* hsh_libera(Hash* tabela);
void  hsh_percorre(Hash* tabela, void callback_func(void*));
```

Considere ainda que, depois de ler uma lista de dados de países, ele tenha armazenado numa tabela hash, `tabela_pais`, os ponteiros que contem os endereços da estrutura `Pais` de cada um dos países lidos.

```
typedef struct pais Pais;

struct pais {
       char nome[81];
       char continente[81];
       int  exercitos;
       char dono[81];
};
```

Escreva, como um módulo cliente do TAD que conhece a estrutura `Pais`, uma função que troque o nome do dono de um país e tenha o seguite protótipo:

```
int dono_do_pais(char* novo_dono, Hash* tabela_pais, char* nome_do_pais);
```

o valor de retorno é 1 se teve sucesso e 0, caso contrario. Pode utilizar as funções das bibliotecas padrão do C, tipo stdio.h string.h, etc...

```
int dono_do_pais(char* novo_dono, Hash* tabela_pais, char* nome_do_pais){
   Pais* pais = hsh_busca(tabela_pais, nome_do_pais);
      If(pais == null){
         Return 0;
      }
      else{
         strcpy( pais.dono, novo_dono);
         return 1;
      }
   }
```

10) Escreva uma função de *hash* que possa ser utilizada para armazenar no TAD descrito na Questão 8 os nomes dos países. Considere o protótipo da interface do TAD acima e uma tabela *hash* com 307 elementos.

```
 int hash_function(char* nome_do_pais){
      int i = 0;
      int k = 0;
      for(i = 0; nome_do_pais[i] != '\0'; i++){
         k += nome_do_pais;
      }
      Return k%307;
 }
```

11) Mostrar o resultado da inserção das chaves 19, 38, 64, 100, 81, 47, 27, 31 em uma tabela de dispersão de tamanho m = 17, sendo que as colisões são tratadas por endereçamento aberto. A sequência de tentativas é dada por:

a) h' (x) = x % m

b) h(x, k) = (h'(x) + k) % m, 0<k<=m-1

31		19		38						27			64	81	100	47
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**

h'(19) = 19 % 17 = 2

h'(38) = 38 % 17 = 4

h'(64) = 64 % 17 = 13

h'(100) = 100 % 17 = 15

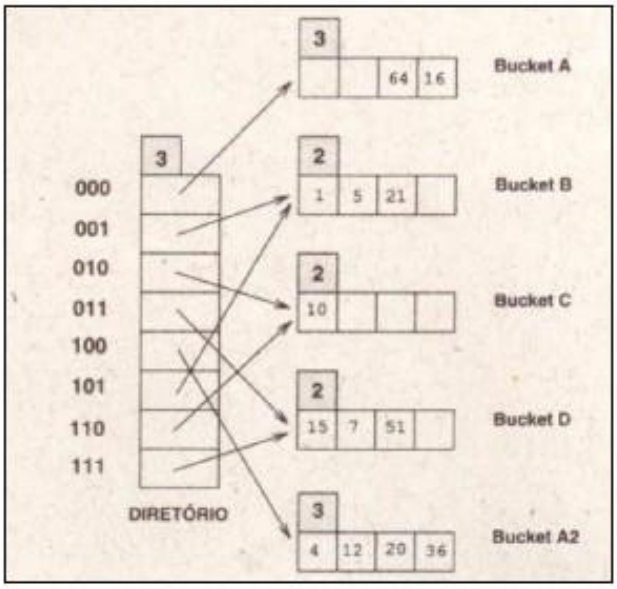
h'(81) = 81 % 17 = 13 => h(81,1) = (13+1) % 17 = 14

h'(47) = 47 % 17 = 13 => h(47,1) = (13+1) % 17 = 14 => h(47,2) = 15 => h(47,3) = 16

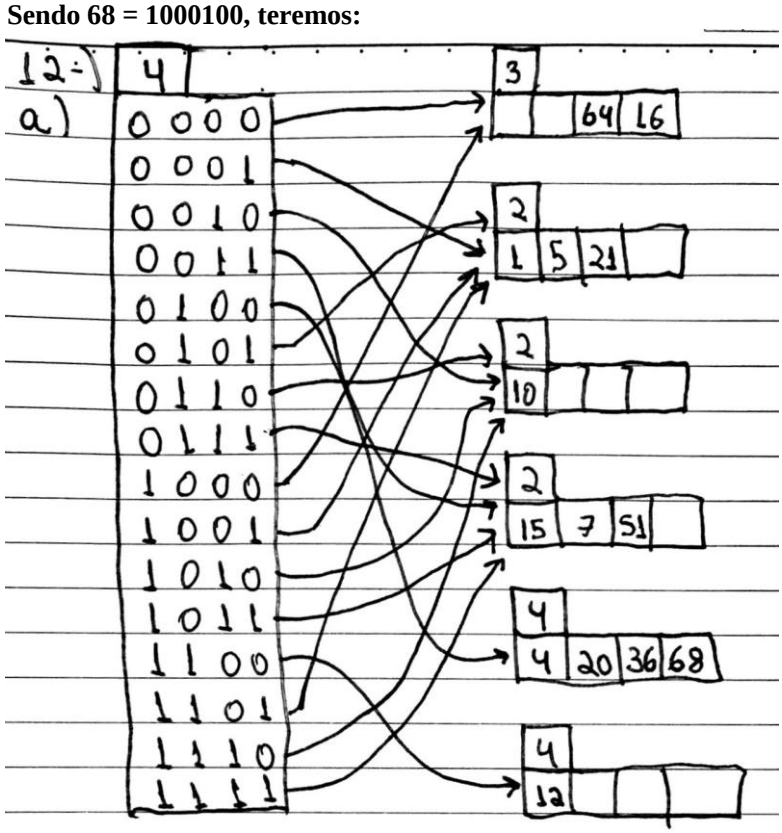
h'(27) = 27 % 17 = 10

h'(31) = 31 % 17 = 14 => h(31,1) = (14+1) % 17 = 15 => h(31,2) = 16 => h(31,3) = 0

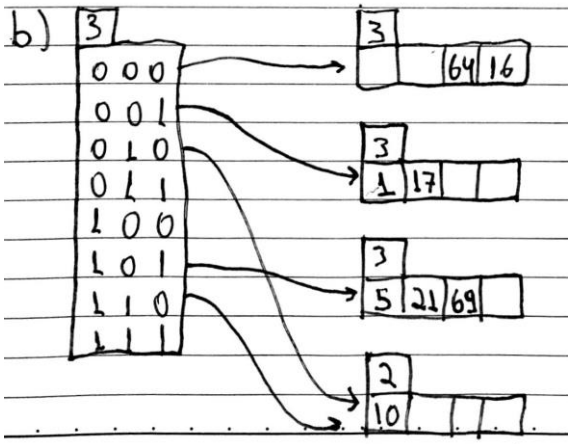
12) Considere o índice hash extensível mostrado na figura abaixo.



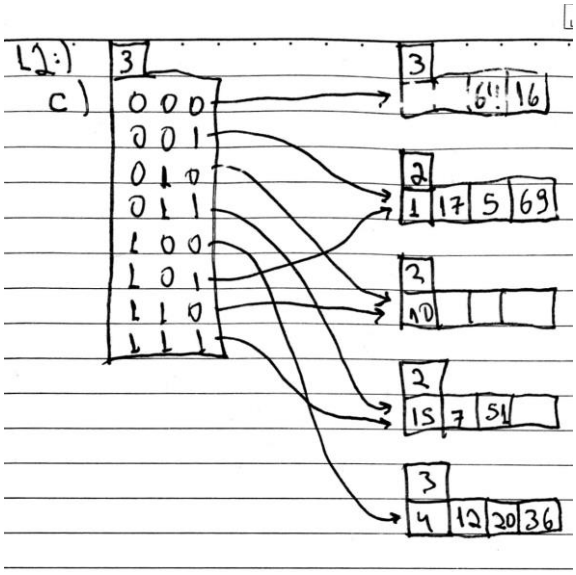
a) Mostre o índice após a inserção de uma chave 68*
Sendo 68 = 1000100, teremos:



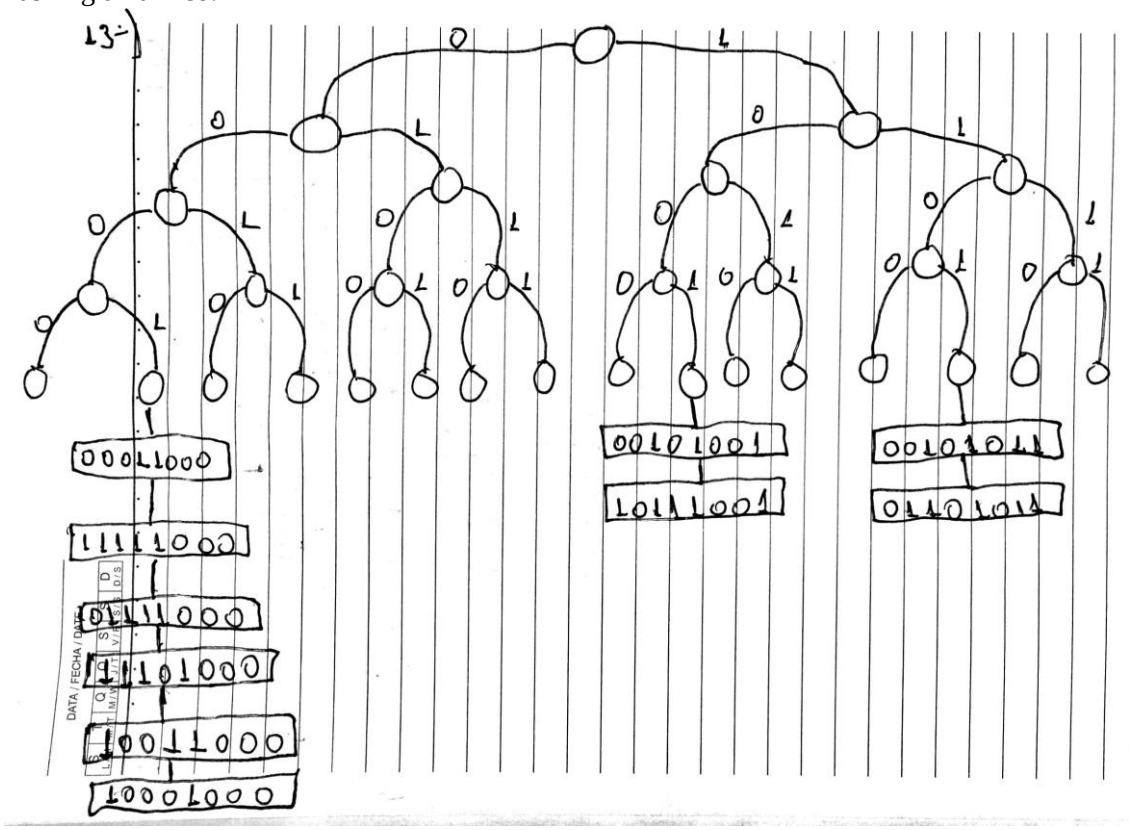
b) Mostre o índice original após a inserção das entradas 17* e 69*.
Sendo 17 = 10001 e 69 = 1000101



c) Mostre o índice após a exclusão da entrada com valor 21*
Sendo 21 = 10101



13) O DBA da empresa X utilizou como estratégia para indexar tuplas da tabela de clientes o mecanismo de hashing dinâmico. Para tanto, foi especificado como chave de busca o CPF dos clientes. Considere que cada da bucket tem capacidade de armazenar duas entradas (chave de busca e ponteiro). Suponha que os resultados de uma função hash aplicada aos valores de chave de busca existentes em sua representação binária são os seguintes: 00011000, 11111000, 01111000, 00101001, 10111001, 11101000, 10011000, 10001000, 00101011, 01101011. Construa a estrutura de diretório (árvore binária) para esta tabela com hashing dinâmico.



14) No vetor mostrado abaixo, mostre os passos do algoritmo O(n) para construção de um heap de mínimo (min Heap). Quantas trocas você precisou fazer?

95,60,78,39,28,66,70,33

14:) Temos 8 valores no vetor sendo 4 nós que não tem filhos como podemos observar no vetor:

95	60	78	39	28	66	70	33
1	2	3	4	5	6	7	8

nós folhas

Dessa forma os elementos a serem ajustados são 39, 78, 60, 95.

OBS: p/ nó na posição i temos que os nós na posição i×2 e i×2+1 são seus filhos.

→ Considerando apenas os nós folhas vamos inserindo os nos não folhas em sua posição e fazendo a troca:

para —, —, —, —, 28, 66, 70, 33, insiro o 39
—, —, —, 39, 28, 66, 70, 33, troco com 33 que é seu filho e é menor que ele
—, —, —, 33, 28, 66, 70, 39

insiro 78
—, —, 78, 33, 28, 66, 70, 39

troco com 66
—, —, 66, 33, 28, 78, 70, 39

insiro 60
—, 60, 66, 33, 28, 78, 70, 39

troco 28
—, 28, 66, 33, 66, 78, 70, 39

insiro 95
95, 28, 66, 33, 60, 78, 70, 39

troco com 28
28, 95, 66, 33, 60, 78, 70, 39

troco com 33
28, 33, 66, 95, 60, 78, 70, 39

troco com 39
28, 33, 66, 39, 60, 78, 70, 95

Ao todo foram 6 trocas.

15) Transforme o vetor 30,15,28,60,45,90,10,23 num *heap* de máximo fazendo o mínimo de trocas possível (mostre cada uma das trocas).

15: Temos 8 valores, tendo então 4 nós folhas.

30, 15, 28, 60, 45, 90, 10, 23

nós filhos

Elementos a serem ajustados: 60, 28, 15, 30

- para -, -, -, -, 45, 90, 10, 23 insiro o 60
  -, -, -, 60, 45, 90, 10, 23, como ele é maior que seu filho 23 então não faço a troca.
- insiro 28: -, -, 28, 60, 45, 90, 10, 23
  troco com 90: -, -, 90, 60, 45, 28, 10, 23
- insiro 15: -, 15, 90, 60, 45, 28, 10, 23
  troco com 60: -, 60, 90, 15, 45, 28, 10, 23
- insiro 30: 30, 60, 90, 15, 45, 28, 10, 23
  troco com 90: 90, 60, 30, 15, 45, 28, 10, 23

90, 60, 30, 15, 45, 28, 10, 23

Ao todo foram realizados 3 trocas

16) Supondo que você crie um *heap* vazio mostre como ele vai sendo construído caso receba os elementos do vetor da questão anterior, um de cada vez. O *heap* final é o mesmo?

16 -) 30, 15, 28, 60, 45, 90, 10, 23

para _, _, _, _, _, _, _, _

- insere 30: 30, _, _, _, _, _, _, _
- insere 15: 30, 15, _, _, _, _, _, _
- insere 28: 30, 15, 28, _, _, _, _, _
- insere 60: 30, 15, 28, 60, _, _, _, _
  troca com 15: 30, 60, 28, 15, _, _, _, _
  troca com 30: 60, 30, 28, 15, _, _, _, _
- insere 45: 60, 30, 28, 15, 45, _, _, _
  troca com 30: 60, 45, 28, 15, 30, _, _, _
- insere 90: 60, 45, 28, 15, 30, 90, _, _
  troca com 28: 60, 45, 90, 15, 30, 28, _, _
  troca com 60: 90, 45, 60, 15, 30, 28, _, _
- insere 10: 90, 45, 60, 15, 30, 28, 10, _
- insere 23: 90, 45, 60, 15, 30, 28, 10, 23
  troca com 15: 90, 45, 60, 23, 30, 28, 10, 15

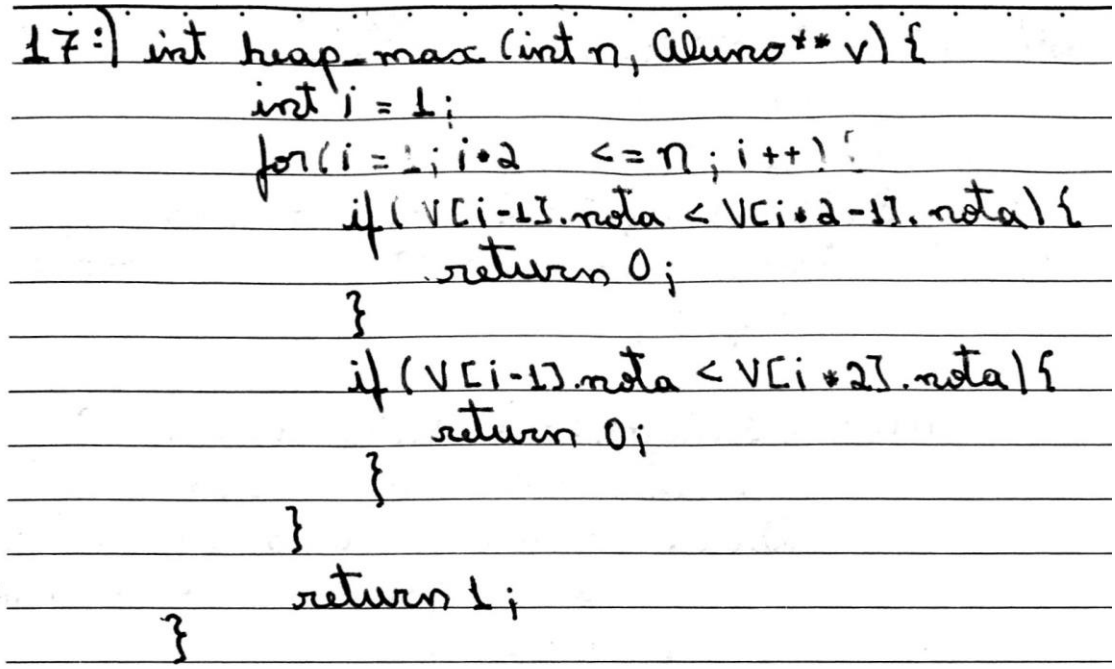
90, 45, 60, 23, 30, 28, 10, 15

Como pode perceber o heap final não é o mesmo da questão anterior.

17) Implemente uma função que verifica se a ordem dos elementos de um vetor de ponteiro para estruturas com dados de alunos representa uma fila de prioridade (*heap*), onde a raiz armazena o aluno com a maior nota. A função recebe como parâmetros o número de elementos no vetor e o vetor de ponteiro para o tipo que representa o aluno. A função deve retornar 1 se a ordem dos elementos representa um *heap*; caso contrário, deve retornar zero.

```
struct aluno {
   char nome[64];
   float nota;
};
typedef struct aluno Aluno;


int heap_max (int n, Aluno** v);
```

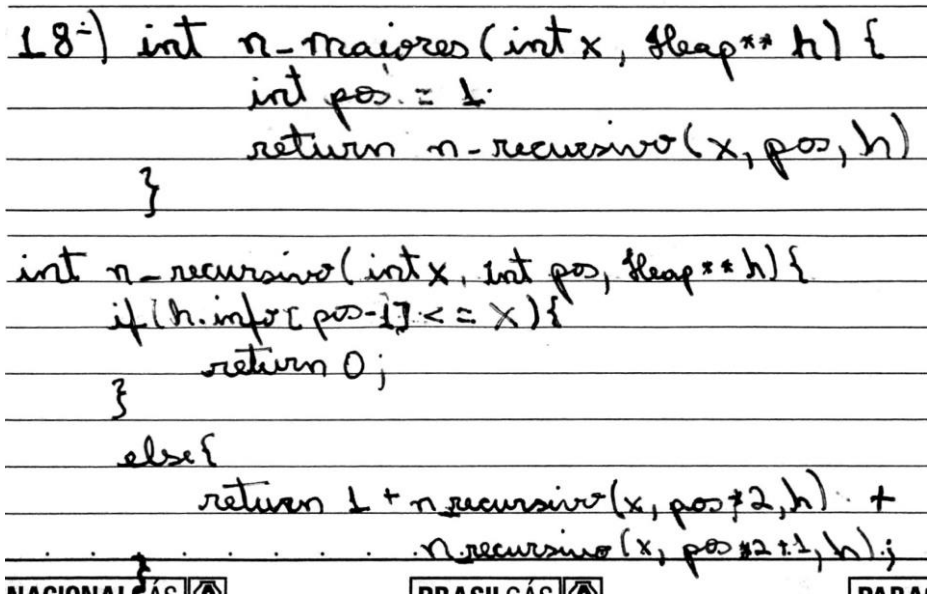


```
17:) int heap_max (int n, Aluno** v) {
        int i = 1;
        for(i = 1; i*2 <= n; i++) {
            if( V[i-1].nota < V[i*2-1].nota) {
                return 0;
            }
            if( V[i-1].nota < V[i*2].nota) {
                return 0;
            }
        }
        return 1;
    }
```

Para as questões a seguir, considere o heap com a seguinte estrutura:

```
typedef struct _heap Heap;
struct _heap {
    int max;   /* tamanho maximo do heap */
    int pos;   /* proxima posicao disponivel no vetor */
    int* info; /* vetor dos elementos do heap */
};
```

18) Escreva uma função em C que retorne o número de elementos maiores que um elemento de valor *x* de um *heap max*, sendo *heap* e *x* enviados como parâmetros da função. A função deve examinar o menor número possível de elementos.

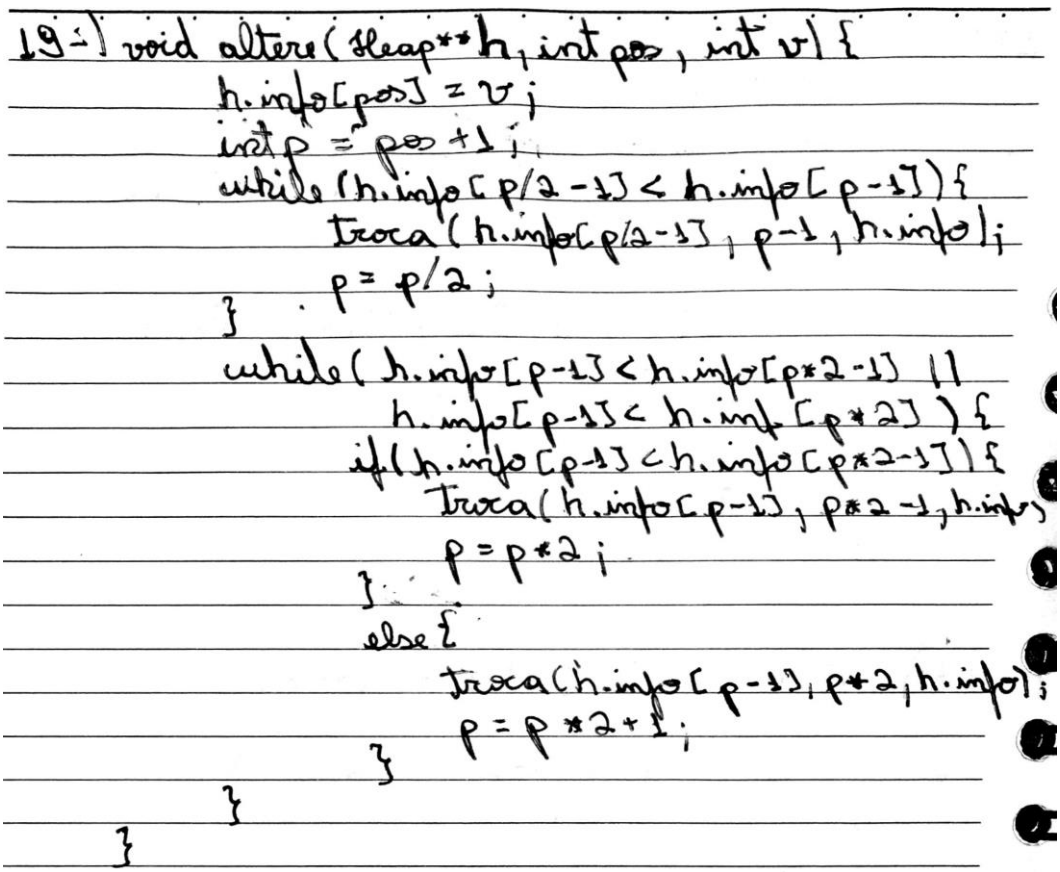


```
18-) int n-maiores(int x, Heap** h) {
        int pos = 1;
        return n-recursivo(x, pos, h)
    }

int n-recursivo(int x, int pos, Heap** h){
    if(h.info[pos-1] <= x){
        return 0;
    }
    else{
        return 1 + n-recursivo(x, pos*2, h) +
                   n-recursivo(x, pos*2+1, h);
    }
```

19) Escreva uma função em C que receba como parâmetros um *heap max*, a posição de um elemento do *heap* (no vetor que o representa) e um valor inteiro, e altere a prioridade do elemento para o novo valor, reposicionando-o, caso necessário. Assuma que já foi definida a função

```
void troca (int pai, int pos, int *info).
```



```
19-) void altere(Heap** h, int pos, int v){
        h.info[pos] = v;
        int p = pos+1;
        while(h.info[p/2-1] < h.info[p-1]){
            troca(h.info[p/2-1], p-1, h.info);
            p = p/2;
        }
        while(h.info[p-1] < h.info[p*2-1] ||
              h.info[p-1] < h.info[p*2]){
            if(h.info[p-1] < h.info[p*2-1]){
                troca(h.info[p-1], p*2-1, h.info);
                p = p*2;
            }
            else{
                troca(h.info[p-1], p*2, h.info);
                p = p*2+1;
            }
        }
    }
```

20)Partição dinâmica. Considere o conjunto {1,2,3,4,5,6}. Como ficaria a representação por vetor deste conjunto após a operação de criar_particao_dinamica? Como ficaria este vetor após as operações: união(1,3), união(2,3), união(2,5), união(3,4), união(1,6) se: (a) a operação de união for feita sem critério de tamanho, e (b) com critérios de tamanho.

20:) {1,2,3,4,5,6}

• após criar_partição:

elemento:	1	2	3	4	5	6
ponteiro:	-1	-1	-1	-1	-1	-1

a) Depois das operações de União

1	2	3	4	5	6
2	-1	1	2	2	2

b) Depois das operações de União

1	2	3	4	5	6
-1	1	1	1	1	1

21)Considerando a partição (a) do problema anterior, como ficaria a partição depois de uma busca(5) seguido de uma busca(3) com a estratégia de compressão de caminho?

21:) Uma vez que a busca com a estratégia de compressão de caminho faz todos os nós por onde passa apontar para a raiz teremos que:

busca(5) =

1	2	3	4	5	6
2	-1	1	2	2	2

busca(3) =

1	2	3	4	5	6
2	-1	2	2	2	2

↳ o 3 passaria a apontar para 2.

22)Uma das muitas aplicações de estruturas de dados de conjuntos disjuntos surge na determinação dos componentes conectados de um grafo não orientado. Neste contexto, considere os algoritmos a seguir:

Recebe um grafo G e contrói uma representação dos componentes conexos.

```
CONNECTED-COMPONENTS (G)
1  para cada vértice v de G faça
2     MAKESET (v)
3  para cada aresta (u, v) de G faça
4     se FINDSET (u) ≠ FINDSET (v)
5        então UNION (u, v)
```

Decide se u e v estão no mesmo componente:

```
SAME-COMPONENT (u, v)
1  se FINDSET (u) = FINDSET (v)
2     então devolva SIM
3     senão devolva NÃO
```

Quando CONNECTED-COMPONENTS é aplicado a um grafo $G = (V, E)$ com k componentes, quantas vezes FINDSET é chamado? Quantas vezes UNION é chamado? De respostas em termos de k, $|V|$ e $|E|$.

23) Faça uma figura da floresta produzida pela seguinte sequência de operações:

```
01  para i ← 1 até16
02      faça MAKESET (x_i)
03  para i ← 1 até15 em passos de 2
04      faça UNION (x_i, x_i+1)
05  para i ← 1 até 13 em passos de 4
06      faça UNION (x_i, x_i+2)
07  UNION (x_1, x_5)
08  UNION (x_11, x_13)
09  UNION (x_1, x_10)
10  FINDSET (x_2)
11  FINDSET (x_9)
```

23:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- após makeset ↑

- após 2ª interação (Union)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	1	-1	3	-1	5	-1	7	-1	9	-1	11	-1	13	-1	15

- após 3ª interação (Union)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	1	1	3	-1	5	5	7	-1	9	9	11	-1	13	13	15

- após Union(1,5)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	1	1	3	1	5	5	7	-1	9	9	11	-1	13	13	15

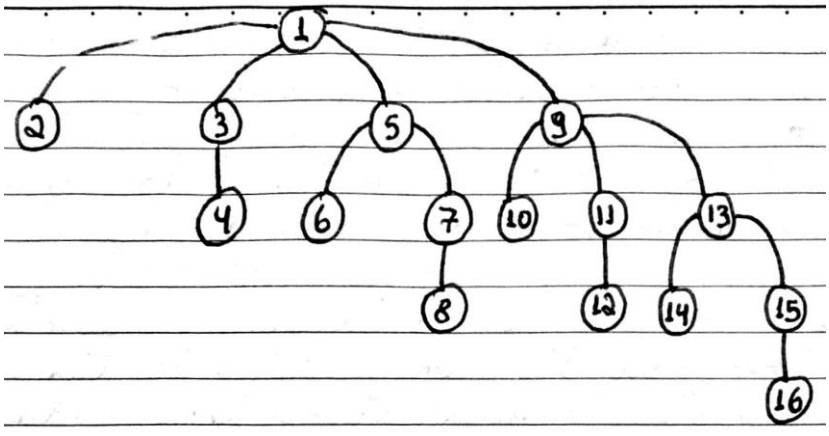
- Union(11,13):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	1	1	3	1	5	5	7	-1	9	9	11	9	13	13	15

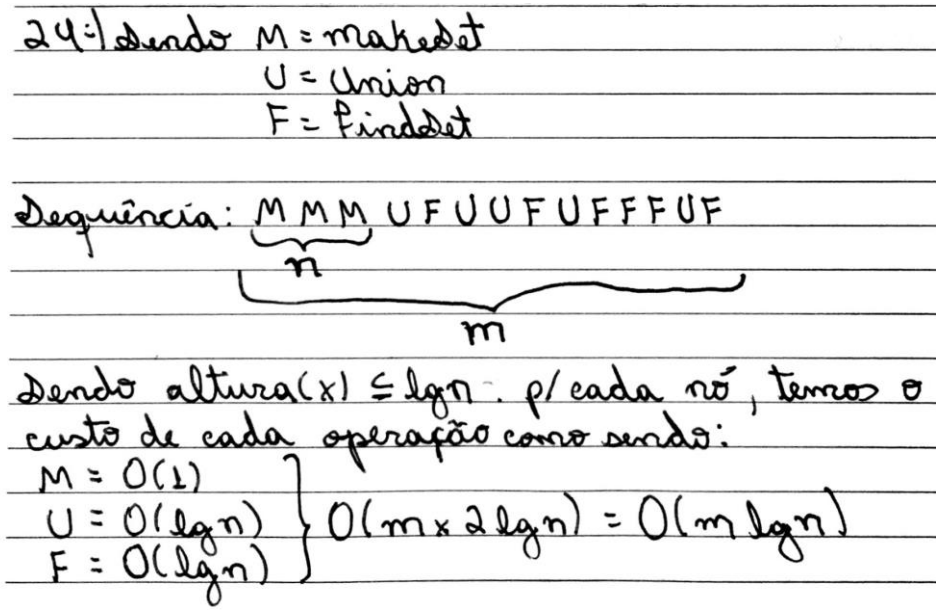
- Union(1,10):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	1	1	3	1	5	5	7	1	9	9	11	9	13	13	15

- FindSet(2) e FindSet(9) não mudam o vetor pois tanto o 2 quanto o 9 estão ligados direto na raiz.



24) Dê uma sequência de m MAKESET, UNION e FINDSET, onde n das quais são operações MAKESET, que consome $\Omega(m \lg n)$.



25) Digamos que h[x] é a altura do nó x (= comprimento do mais longo caminho que vai de x até uma folha) na estrutura disjoint-set forest. Mostre que rank[x] ≥ h[x]. Mostre que UNION (x, y) nem sempre pendura a árvore mais baixa na mais alta.