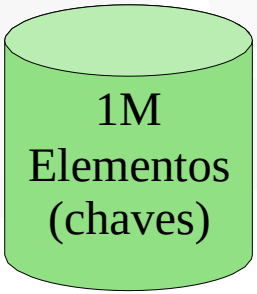
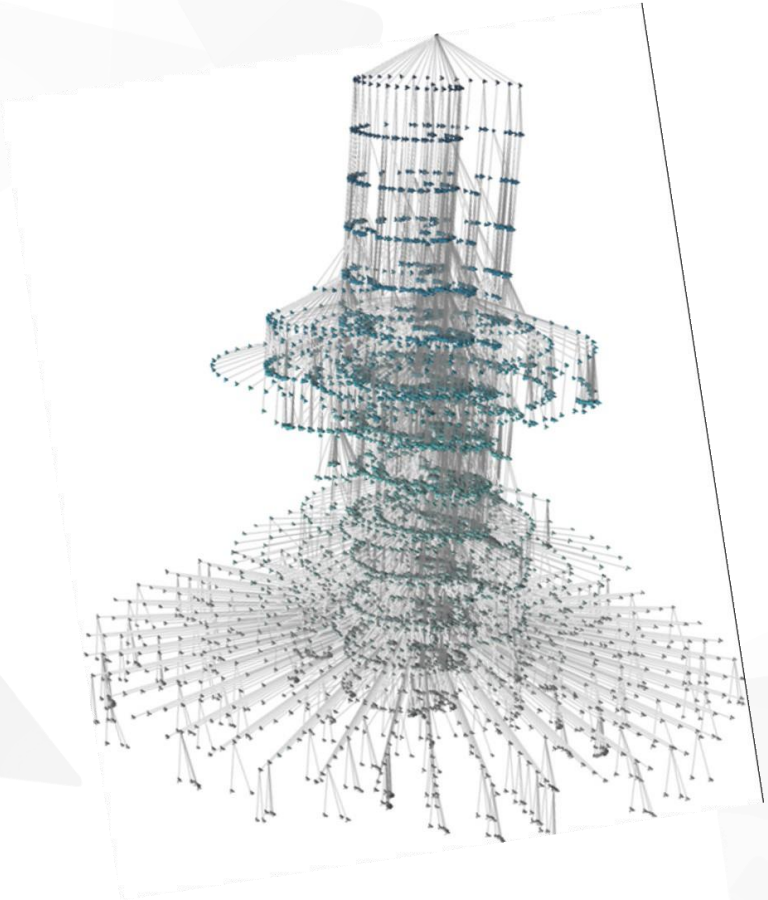
AED2
Algoritmos e Estruturas de Dados II

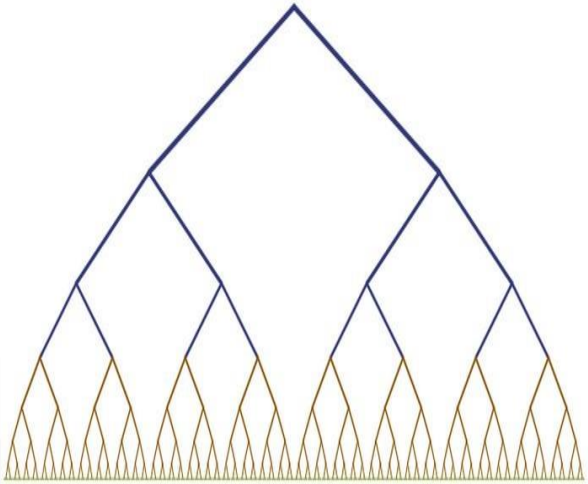
Aula 09 – Árvores Binárias de Busca
Árvores de Huffman

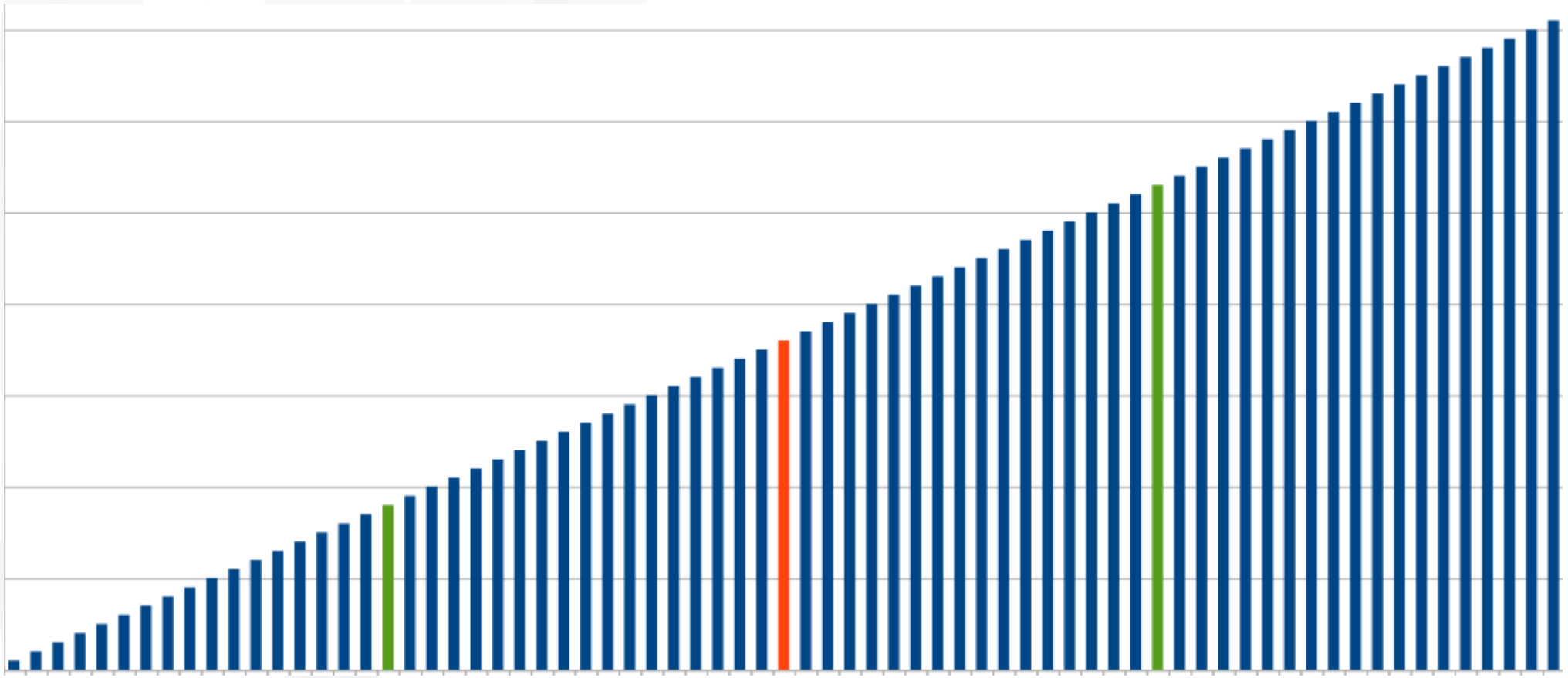
Prof. Aléssio Miranda Júnior
alessio@timoteo.cefetmg.br
2Q-2016

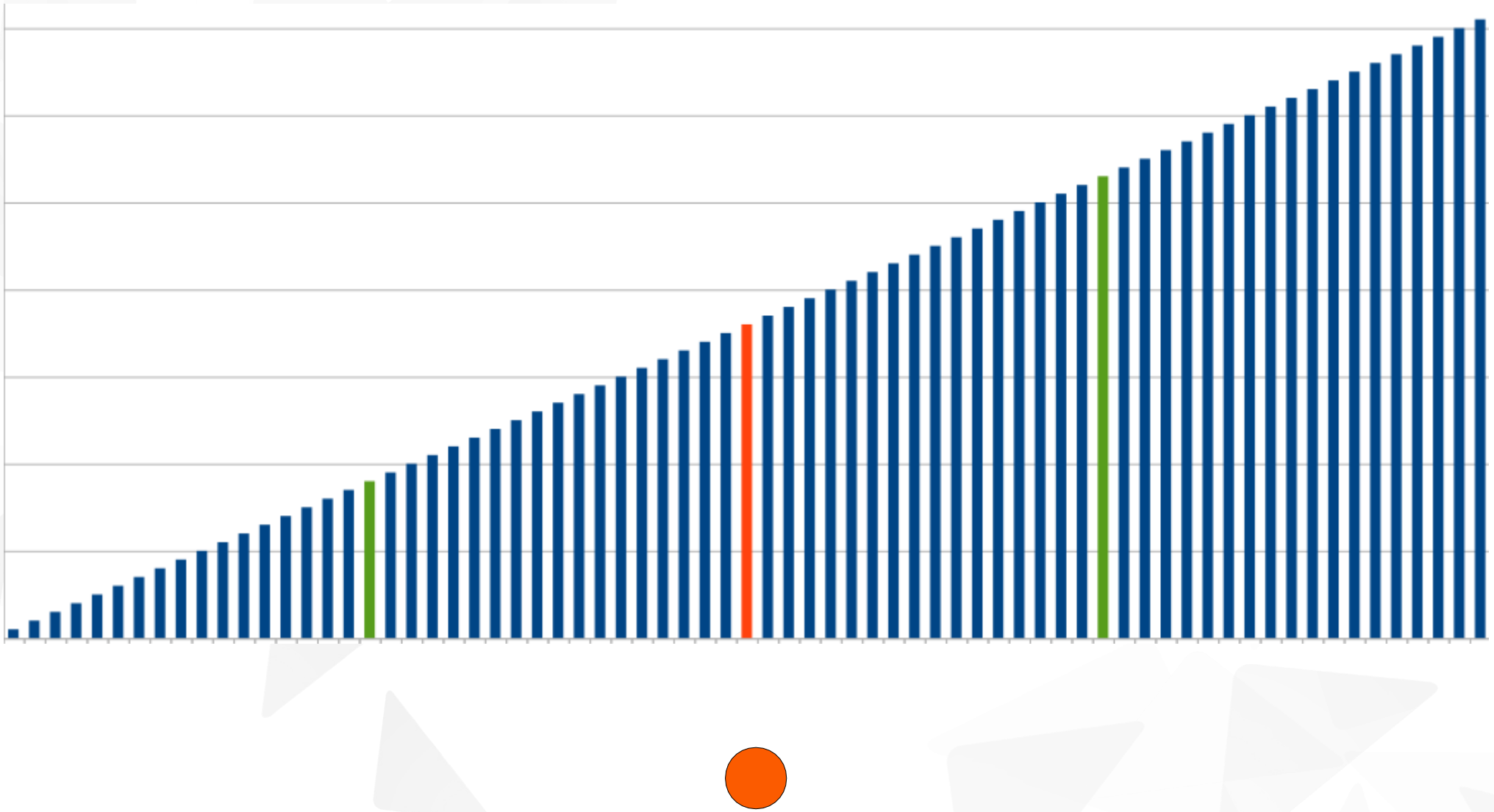
Construindo a melhor árvore binária de busca?

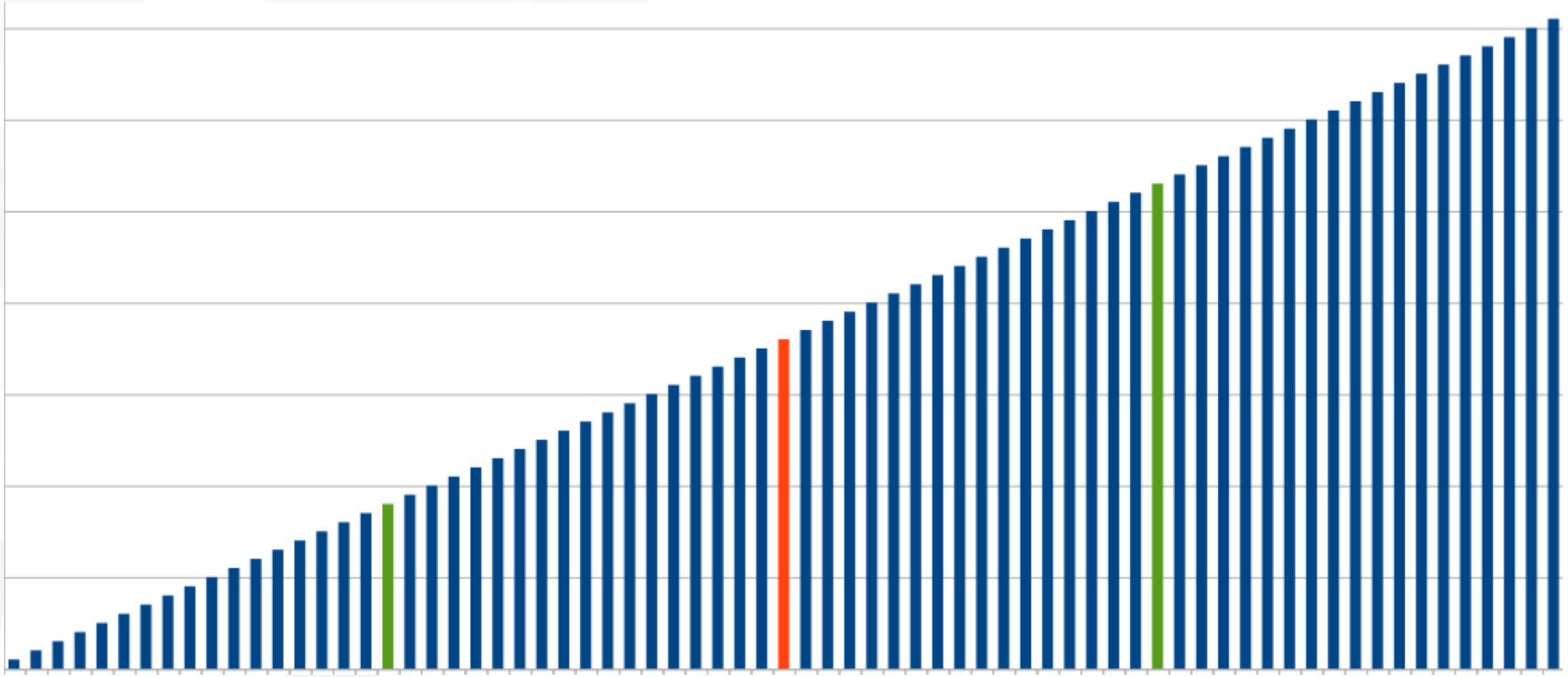

1M
Elementos
(chaves)

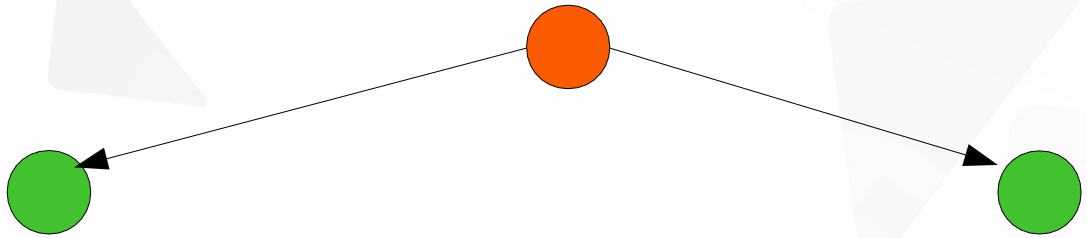


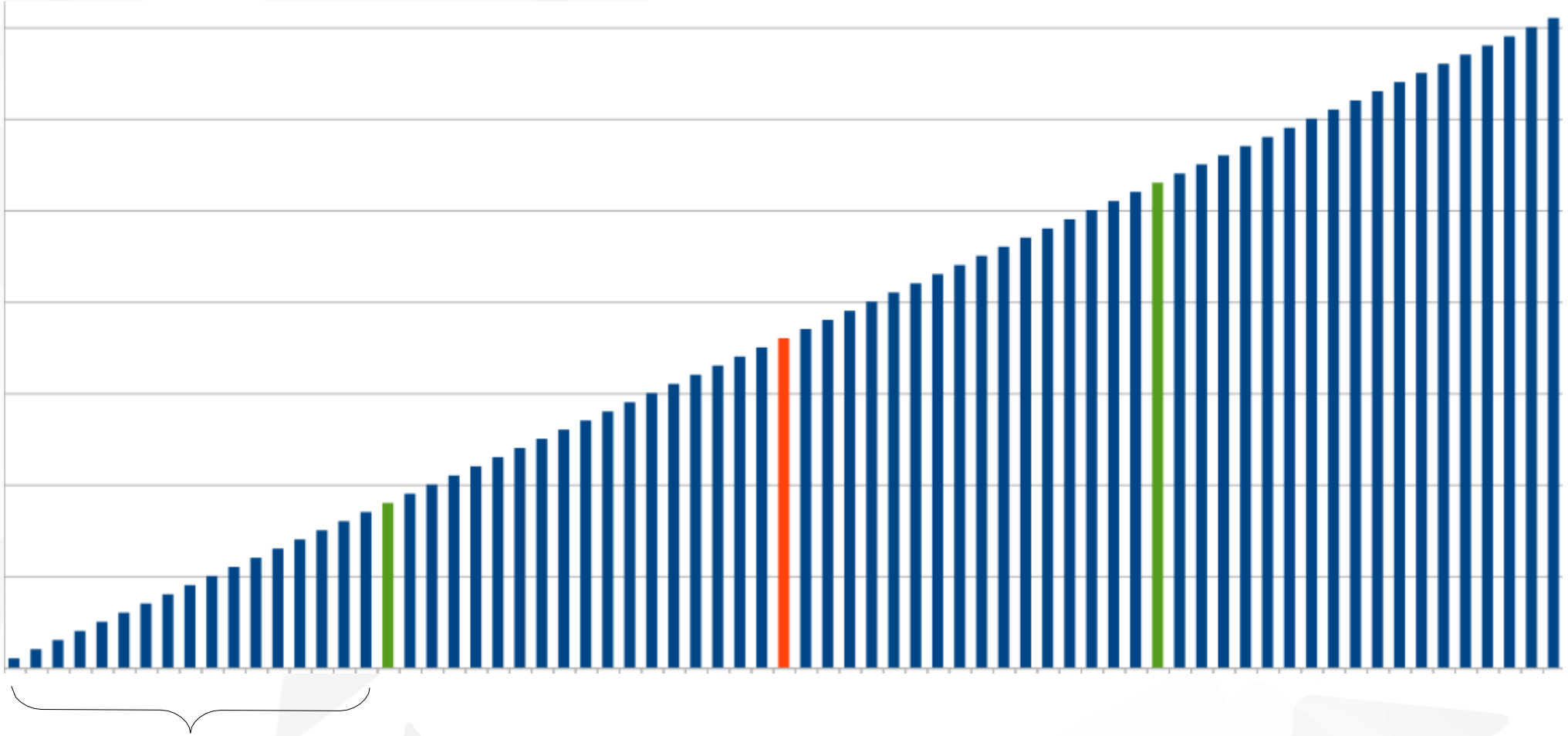


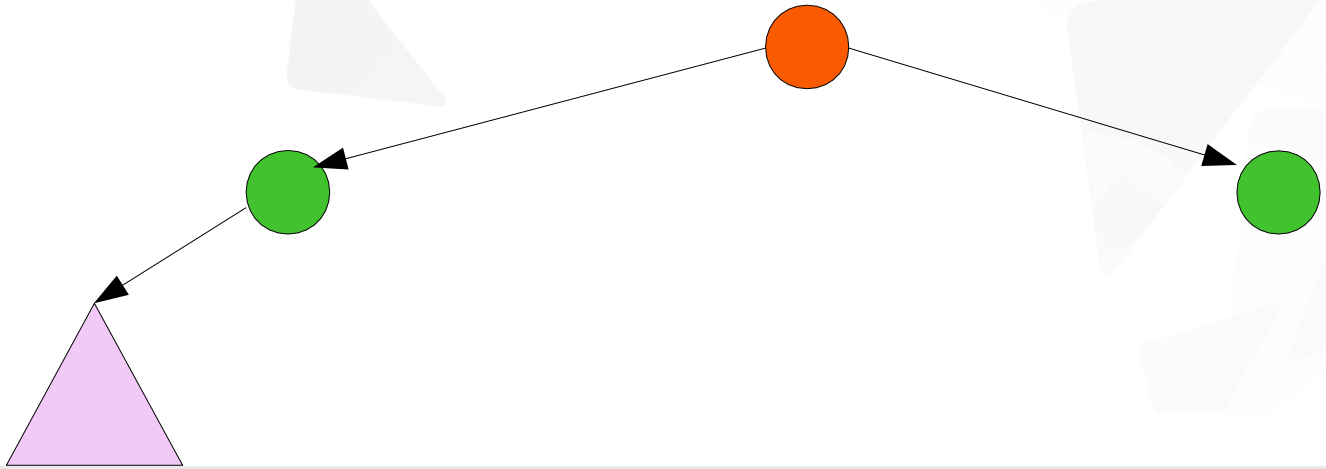


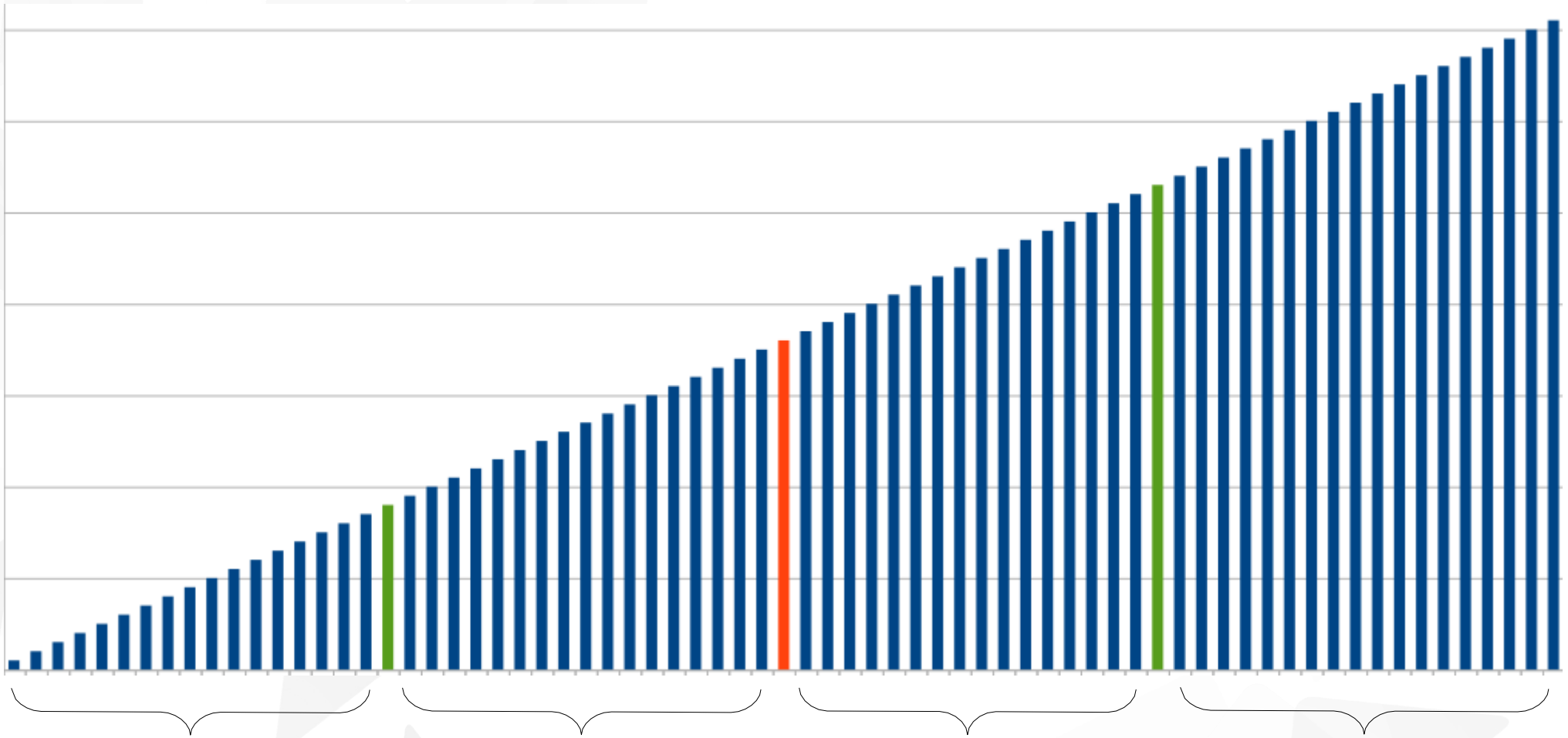


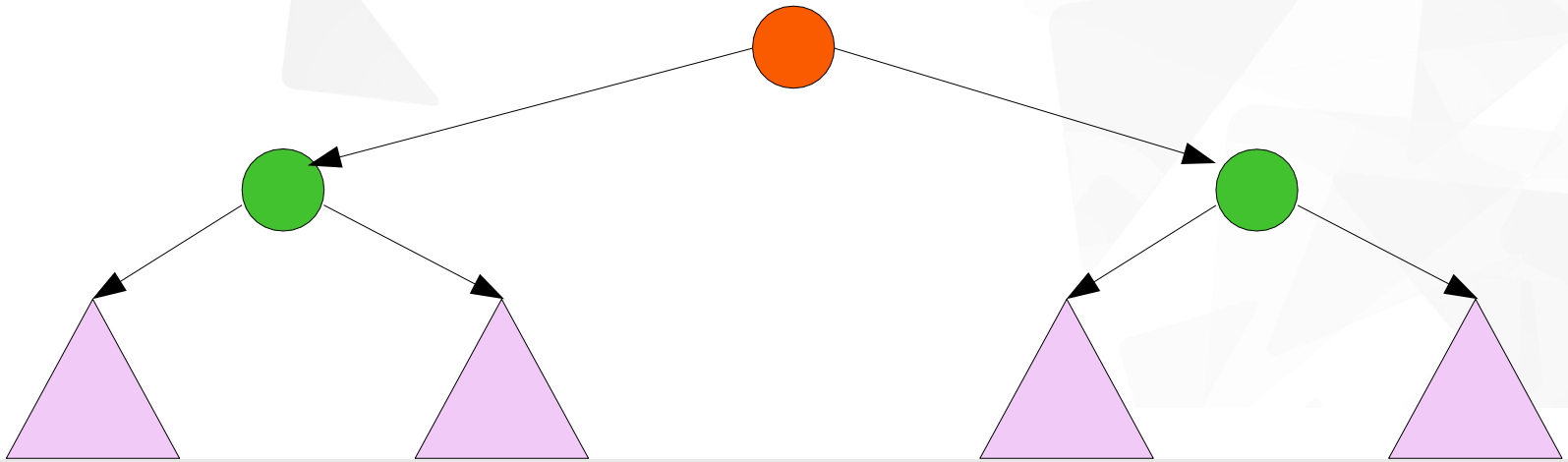


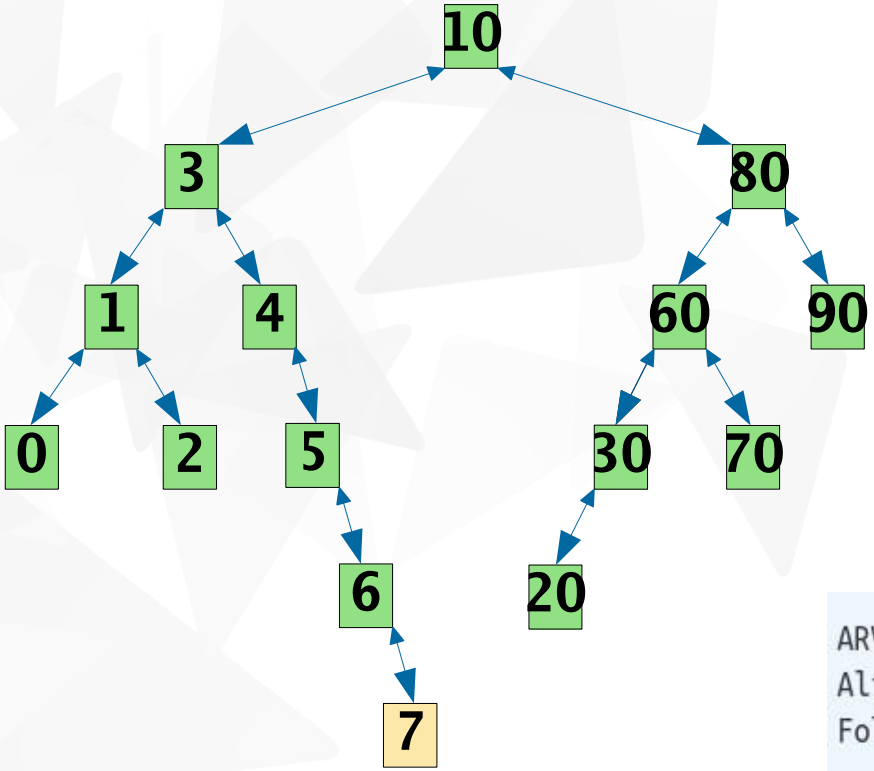






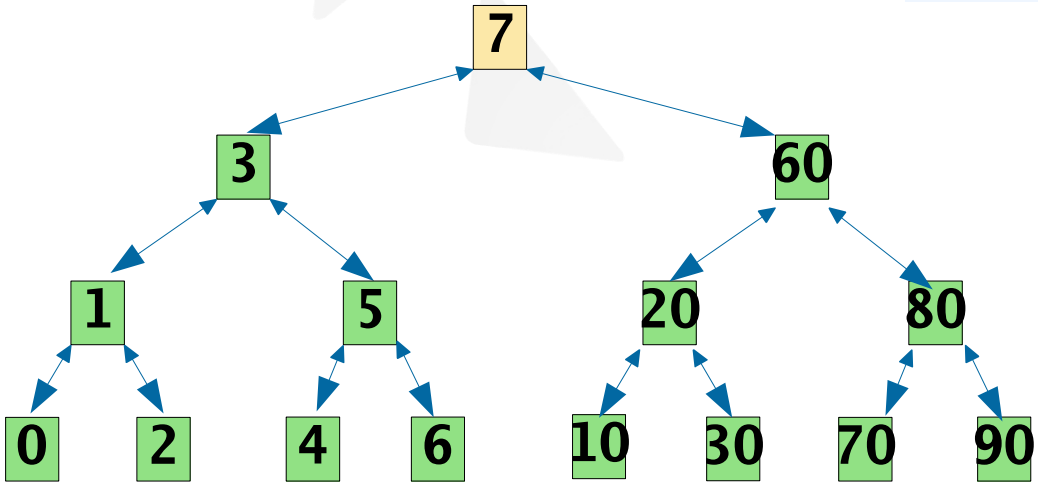







```
ARVORE 1
Altura: 5
Folhas: 0       2       7       20      70      90

ARVORE 2
Altura: 3
Folhas: 0       2       4       6       10      30      70      90
```



Teste01.java

```
no* criarArvoreBalanceada(no *r, int v[], int n) {
    if (n>0) {
        int q = n/2;
        r = inserirNoNaArvore(r, v[q]);
        r = criarArvoreBalanceada(r, v, q);
        r = criarArvoreBalanceada(r, &v[q+1], n-q-1);
    }
    return r;
}
```

```
int main(int argc, char *argv[])
{
    int i;
    int v[] = {10,3,80,1,4,60,90,0,2,5,30,70,6,20,7};
    int n = sizeof(v)/sizeof(v[0]);

    printf("\nARVORE 1");
    no *arvore1 = NULL;
    for (i=0; i<n; i++)
        arvore1 = inserirNoNaArvore(arvore1, v[i]);
    printf("\nAltura: %d", altura(arvore1));
    printf("\nFolhas: "); imprimirFolhas(arvore1);

    printf("\n\nARVORE 2");
    QuickSort(v, 0, n-1);
    no *arvore2 = NULL;
    arvore2 = criarArvoreBalanceada(arvore2, v, n);
    printf("\nAltura: %d", altura(arvore2));
    printf("\nFolhas: "); imprimirFolhas(arvore2);
}
```

Teste02.java

```c
int main(int argc, char *argv[])
{
    int i;
    int n = atoi(argv[1]);
    int v[n];

    for (i=0; i<n; i++)
        scanf("%d", &v[i]);


    printf("\nARVORE 1");
    no *arvore1 = NULL;
    for (i=0; i<n; i++)
        arvore1 = inserirNoNaArvore(arvore1, v[i]);
    printf("\nAltura: %d", altura(arvore1));


    printf("\n\nARVORE 2");
    QuickSort(v, 0, n-1);
    no *arvore2 = NULL;
    arvore2 = criarArvoreBalanceada(arvore2, v, n);
    printf("\nAltura: %d", altura(arvore2));
}
```

```
ARVORE 1
Altura: 37

ARVORE 2
Altura: 15
```

Log2(40000) = 15,288

n	lg(n)
2	1
32	5
512	9
8192	13
131072	17
2097152	21
33554432	25
536870912	29
8589934592	33
137438953472	37
2199023255552	41
35184372088832	45
562949953421312	49
9007199254740990	53
144115188075856000	57
2305843009213690000	61
36893488147419100000	65

```
$ gcc  teste02.c  o teste02.exe
$ ./teste02.exe 40000 < vetor4.dat
```

Teste02. java

```
no* criarArvoreBalanceada(no *r, int v[], int n) {
    if (n>0) {
        int q = n/2;
        r = inserirNoNaArvore(r, v[q]);
        r = criarArvoreBalanceada(r, v, q);
        r = criarArvoreBalanceada(r, &v[q+1], n-q-1);
    }
    return r;
}
```

$$T(n) = \begin{cases} 0 & , \text{se } n = 0 \\ \lg(n) + 2T(n/2) & , \text{se } n > 0 \end{cases}$$

Teste02.java

```
no* criarArvoreBalanceada(no *r, int v[], int n) {
    if (n>0) {
        int q = n/2;
        r = inserirNoNaArvore(r, v[q]);
        r = criarArvoreBalanceada(r, v, q);
        r = criarArvoreBalanceada(r, &v[q+1], n-q-1);
    }
    return r;
}
```

$$T(n) = \begin{cases} 0 & , \text{se } n = 0 \\ \lg(n) + 2T(n/2) & , \text{se } n > 0 \end{cases}$$

$$T(n) = \lg(n) - 2$$

Número de comparações realizadas na criação da árvore

Árvores de Huffman
(Huffman trees)

Alfabeto & codificação

- Considere um **alfabeto** contendo n símbolos e uma mensagem composta por símbolos deste alfabeto.

- Deseja-se **codificar** a mensagem como um conjunto de bits (**0 ou 1**), atribuindo um código a cada símbolo, e concatenando-os para produzir uma mensagem codificada.

- Exemplo:
  - Alfabeto = {A,B,C,D}

Símbolo	Código
A	010
B	100
C	000
D	111

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	010
B	100
C	000
D	111

A mensagem ABACCDA pode ser codificada como:
01010001000000111010

Aqui serão necessários 21 bits → Codificação ineficiente!

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	00
B	01
C	10
D	11

A mensagem ABACCDA pode ser codificada como:
00010010101100

Aqui serão necessários 14 bits → Codificação eficiente!

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	00
B	01
C	10
D	11

A mensagem ABACCDA pode ser codificada como:
00010010101100

Aqui serão necessários **14** bits → Codificação eficiente!

Podemos fazer algo melhor (menos bits)?

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	0
B	110
C	10
D	111

Códigos de tamanho variável!

A mensagem ABACCDA pode ser codificada como:
0110010101110

Aqui serão necessários **13** bits → Codificação eficiente!

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	0
B	110
C	10
D	111

Códigos de tamanho variável!

A mensagem ABACCDA pode ser codificada como:
0110010101110

Aqui serão necessários **13** bits → Codificação eficiente!

Em mensagens de milhares de palavras, os símbolos mais frequentes Devem ter uma "melhor" (menor) codificação

Alfabeto & codificação

Alfabeto = {A,B,C,D}

Símbolo	Código
A	0
B	110
C	10
D	111

O código de um símbolo não deve ser o prefixo de outro!

ABACCDA → 0110010101110

0110010101110 → ABACCDA

Algoritmo de Huffman

- O árvore de Huffman pode ser utilizado para criar este tipo de codificação.
- O algoritmo de Huffman usa a árvore para **compressão de dados.**
- Reduções dos tamanhos dos arquivos dependem das características dos dados neles contidos (valores oscilam entre 20-90%)

Símbolo	Código
A	0
B	110
C	10
D	111

Algoritmo de Huffman

- Exemplo: Considere um alfabeto = {a,b,c,d,e,f}

	a	*b*	*c*	*d*	*e*	*f*
Freqüência (em milhares)	45	13	12	16	9	5
Código de tamanho fixo	000	001	010	011	100	101
Código de tamanho variável	0	101	100	111	1101	1100

Códigos de tamanho fixo: $3 \times 100.000 = 300.000$

Códigos de tamanho variável:

$$(\underbrace{45 \times 1}_{a} + \underbrace{13 \times 3}_{b} + \underbrace{12 \times 3}_{c} + \underbrace{16 \times 3}_{d} + \underbrace{9 \times 4}_{e} + \underbrace{5 \times 4}_{f}) \times 1.000 = 224.000$$

Ganho de $\approx$ 25% em relação à solução anterior.

Fonte: Material de aula de C. de Souza et al. (MAC448/Unicamp)

Algoritmo de Huffman

Alfabeto = {A,B,C,D}

ABACCDA → 0110010101110

Símbolo	Código	Freq
A	0	3
B	110	1
C	10	2
D	111	1

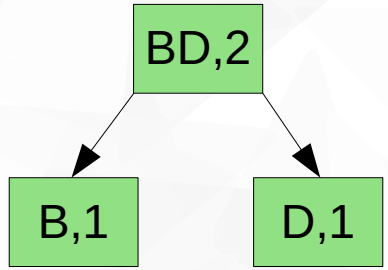
B,1 D,1

Algoritmo de Huffman

Alfabeto = {A,B,C,D}

ABACCDA → 0110010101110

Símbolo	Código	Freq
A	0	3
B	110	1
C	10	2
D	111	1

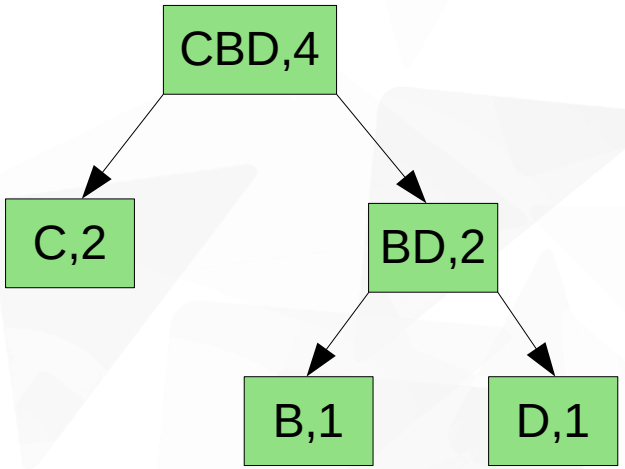


Algoritmo de Huffman

Alfabeto = {A,B,C,D}

ABACCDA → 0110010101110

Símbolo	Código	Freq
A	0	3
B	110	1
C	10	2
D	111	1

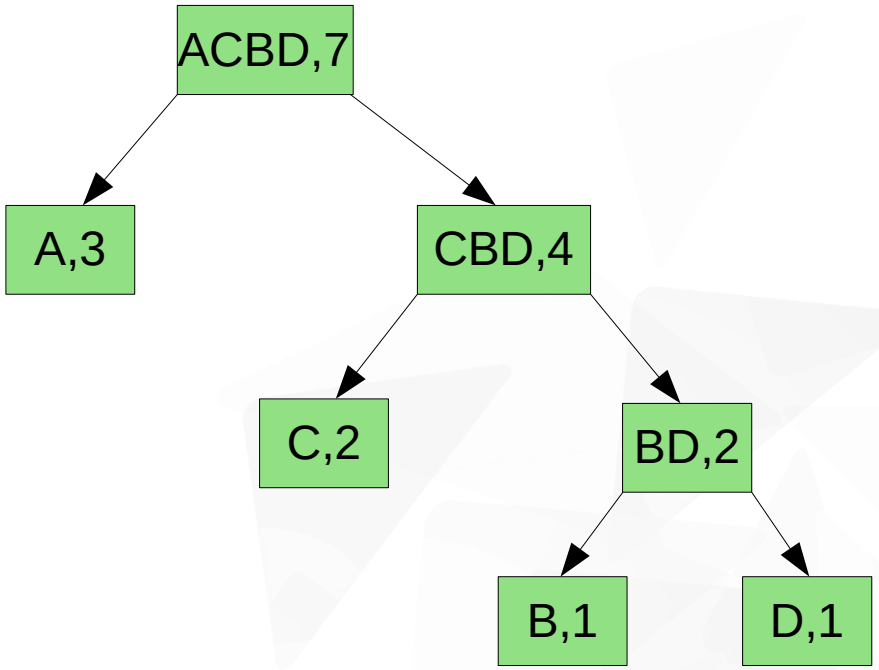


Algoritmo de Huffman

Alfabeto = {A,B,C,D}

ABACCDA → 0110010101110

Símbolo	Código	Freq
A	0	3
B	110	1
C	10	2
D	111	1

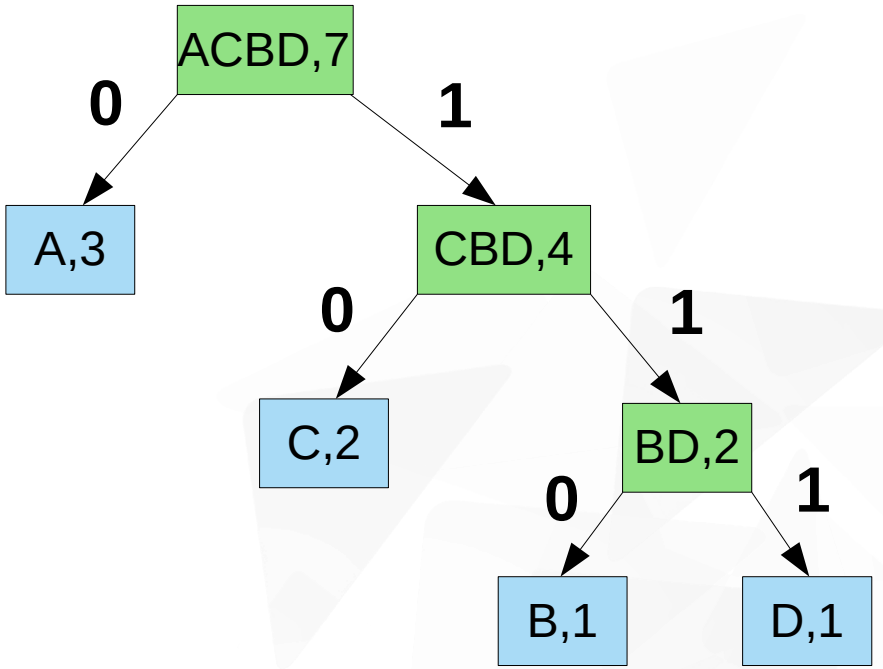


Algoritmo de Huffman

Alfabeto = {A,B,C,D}

ABACCDA → 0110010101110

Símbolo	Código	Freq
A	0	3
B	110	1
C	10	2
D	111	1



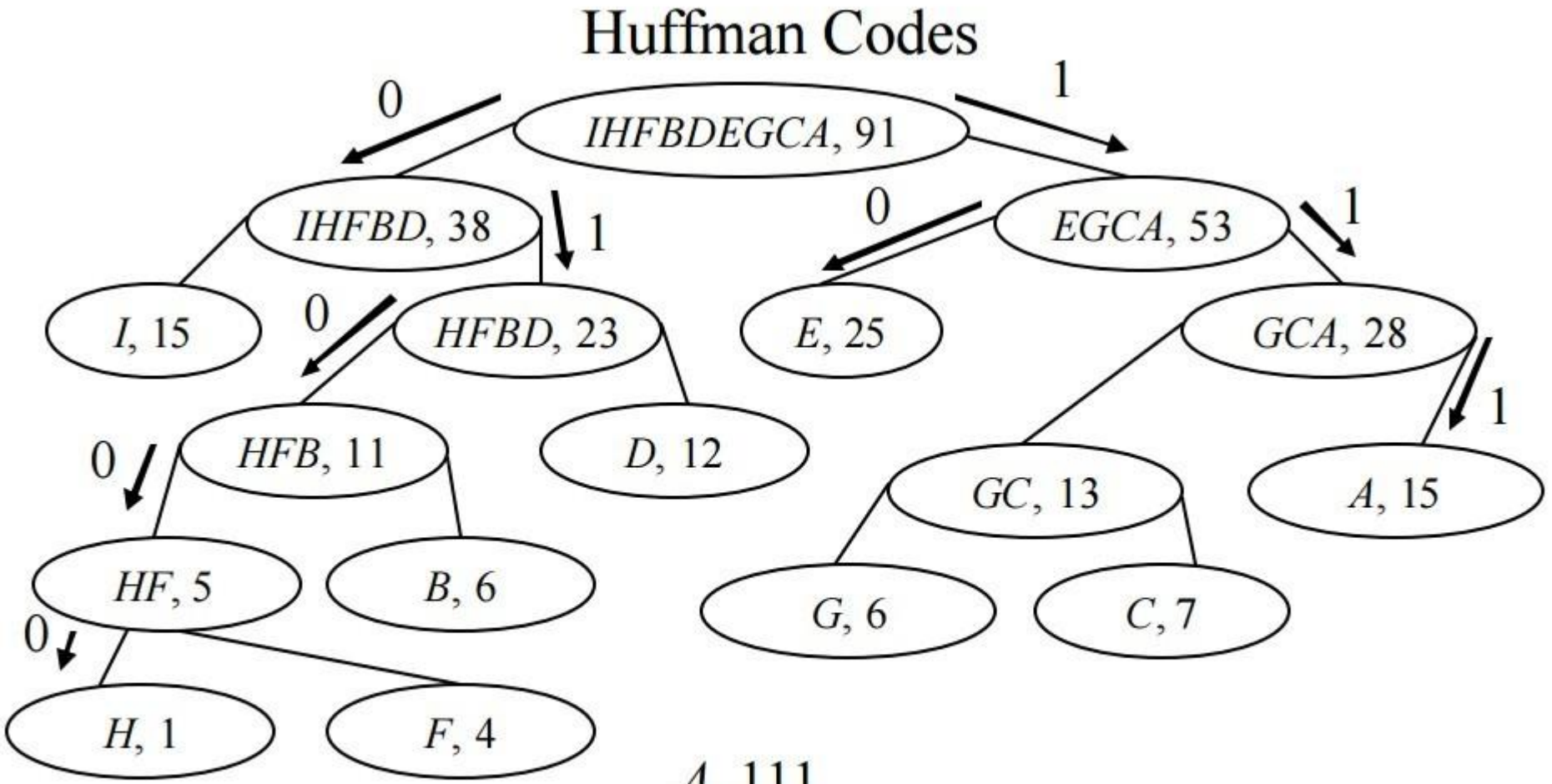
Algoritmo de Huffman

- Considere o alfabeto {E,I,A,D,C,G,B,F,H} e a seguinte frequência.

E, 25
I, 15
A, 15
D, 12
C, 7
G, 6
B, 6
F, 4
H, 1

- Contrua uma árvore de Huffman

Algoritmo de Huffman



Huffman.java

```c
struct cel {
    char conteudo;
    int frequencia;
    struct cel *esq;
    struct cel *dir;
};
typedef struct cel no;
```

```c
int contarCaractere(no *vAlfabeto[], int tAlfabeto, char c) {
    int i;
    for (i=0; i<tAlfabeto; i++) {
        if (vAlfabeto[i]->conteudo==c) {
            vAlfabeto[i]->frequencia+=1;
            return tAlfabeto;
        }
    }
    no* novoElemento = (no*)malloc(sizeof(no));
    novoElemento->conteudo = c;
    novoElemento->frequencia = 1;
    vAlfabeto[tAlfabeto] = novoElemento;
    return tAlfabeto+1;
}
```

```c
int i, tAlfabeto=0;
char c;
int n = atoi(argv[1]); // numero de carateres
no *vAlfabeto[256] = {NULL};

// Leitura dos caracteres
for (i=0; i<n; i++) {
    scanf(" %c", &c);
    tAlfabeto = contarCaractere(vAlfabeto, tAlfabeto, c);
}

// Informacoes do alfabeto
printf("\nTamanho do alfabeto: %d", tAlfabeto);
for (i=0; i<tAlfabeto; i++)
    printf("\n%c : %d", vAlfabeto[i]->conteudo, vAlfabeto[i]->frequencia);
```

Huffman. java

```
$ gcc huffman.c o huffman.exe

$ ./huffman.exe 7 < exemplo1.txt
Tamanho do alfabeto: 4
A : 3
B : 1
C : 2
D : 1

$ ./huffman.exe 90 < exemplo2.txt
Tamanho do alfabeto: 9
A : 15
B : 6
C : 7
D : 12
E : 25
F : 4
G : 6
H : 1
I : 14
```

```
no* contruirArvoreDeHuffman (no *vAlfabeto[], int n) {
    while (n>1) {
        InsertionSort(vAlfabeto, n);

        no* filhoEsq = vAlfabeto[0];
        no* filhoDir = vAlfabeto[1];

        no* novoElemento = (no*)malloc(sizeof(no));
        novoElemento->frequencia = filhoEsq->frequencia+filhoDir->frequencia;
        novoElemento->conteudo = ' ';
        novoElemento->esq = filhoEsq;
        novoElemento->dir = filhoDir;

        vAlfabeto[1] = novoElemento;
        n--;
        vAlfabeto = &vAlfabeto[1];
    }
    return vAlfabeto[0];
}
```

```
$ wc historiasdameianoite.txt
 4168   41768   238961   historiasdameianoite.txt

$ ./huffman.exe 230000 < historiasdameianoite.txt
```

Histórias da Meia-Noite

Texto-fonte:
Obra Completa, de Machado de Assis, vol. II,
Nova Aguilar, Rio de Janeiro, 1994.

Publicado originalmente por Editora Garnier, Rio de Janeiro, 1873

```
$ ./java huffman < historiasdameianoite.txt
Tamanho do alfabeto: 51
h : 2402
i : 12038
s : 14401
t : 7974
o : 20628
r : 12188
a : 26808
d : 9243
m : 41525
e : 24705
  : 1490
n : 9184
x : 483
f : 1708
: : 129
b : 1481
c : 7024
p : 4911
l : 5876
, : 2822
v : 3325
. : 2840
g : 2122
u : 9051
```

```
…
Numero de nos na arvore de Huffman: 101

Codigos de Huffman:
 o: 20628 : 000
 ,: 2822 : 001000
 .: 2840 : 001001
 l: 5876 : 00101
 i: 12038 : 0011
 e: 24705 : 010
 r: 12188 : 0110
 b: 1481 : 0111000
 :  1490 : 0111001
 j: 640 : 01110100
 9: 2 : 0111010100000000
 y: 2 : 0111010100000001
 4: 5 : 011101010000001
 5: 11 : 01110101000001
 +: 1 : 01110101000010000
 k: 1 : 0111010100000100010
 $: 1 : 0111010100000100011
 w: 3 : 0111010100001001
 3: 6 : 011101010000101
```