Programação 2

Classificação por Seleção (Seleção Direta e HeapSort)

Instituto Federal do Espírito Santo - Ifes
Campus Colatina
Professor: Giovany Frossard Teixeira

Definição

- Os métodos de classificação por seleção buscam o menor (ou maior) valor do vetor e colocá-o na sua posição definitiva correta.
- O vetor classificado fica, dessa maneira, reduzido a um elemento.
- O mesmo processo é repetido para a parte restante do vetor, até que fique reduzido a um único elemento, quando então a classificação estará concluída.

Seleção Direta

- A seleção da menor chave é feita por pesquisa sequencial. Uma vez encontrado o menor elemento, ele é permutado e ocupa a posição inicial do vetor, que fica, assim, reduzido a um elemento.
- O processo de seleção é repetido no restante do vetor.

Seleção Direta

```
#include "stdafx.h"

void troca(int *a, int *b){
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
}

int indice_valor_minimo(int *vet, int inicio, int fim){
    int min = inicio;
    for(int j = inicio + 1; j < fim; j++){
        if (vet[j] < vet[min]) min = j;
    }
    return min;
}
```

Seleção Direta

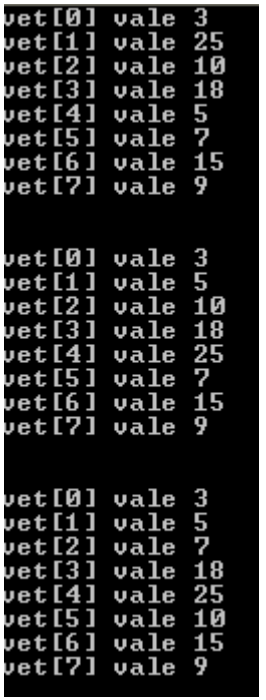
```
void selecao_direta(int *vet, int n){
    int min;
    for(int i= 0; i<n-1; i++){
        min = indice_valor_minimo(vet, i, n);
        troca(&vet[i], &vet[min]);
    }
}
```

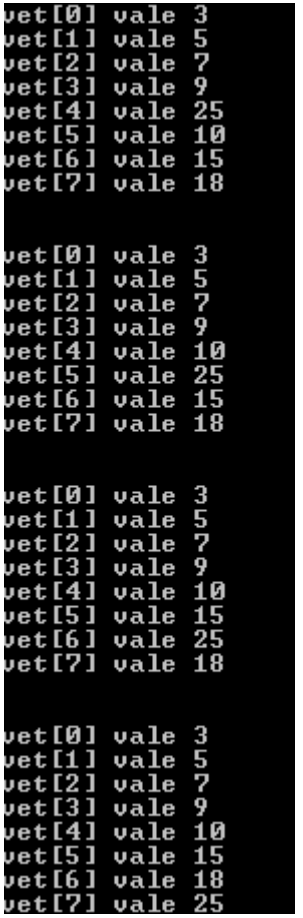
Exemplo

```
void escreveVetor(int *vet, int n){
     for(int i=0; i<n; i++)
        printf("vet[%d] vale %d \n", i, vet[i]);
     printf("\n\n");
}


int _tmain(int argc, _TCHAR* argv[]) {
    int vetor[8] = { 9, 25, 10, 18, 5, 7, 15, 3};
    selecao_direta(vetor, 8);
    escreveVetor(vetor, 8);
    return 0;
}
```

Exemplo

- **A impressão do vetor após cada troca é apresentada ao lado.**





Análise de Desempenho

- O **for** do método de seleção direta roda n-1 vezes.
- Já a função **indice_valor_minimo** possui um for que depende dos parâmetros inicio e fim. Na primeira vez n-1 vezes(inicio=0, fim=n). Lembrando que no laço i vai de inicio+1 até menor que fim, ou seja, n-1 vezes.
- Na segunda vez temos n-2 repetições do laço, em seguida n-3 e assim sucessivamente.
- $(n-1) + (n-2) + (n-3) \ldots + 3 + 2 + 1 = n * n/2 = O(n^2)$

Heap Sort

- Dado um vetor de elementos $C_1, C_2, ..., C_n$ devemos visualizar esse vetor como uma árvore binária e a seguinte condição deve ser satifeita:

$$\left.\begin{array}{l} C_i \geq C_{2i} \\ C_i \geq C_{2i+1} \end{array}\right\} \text{para } i = 1, n \text{ div } 2$$

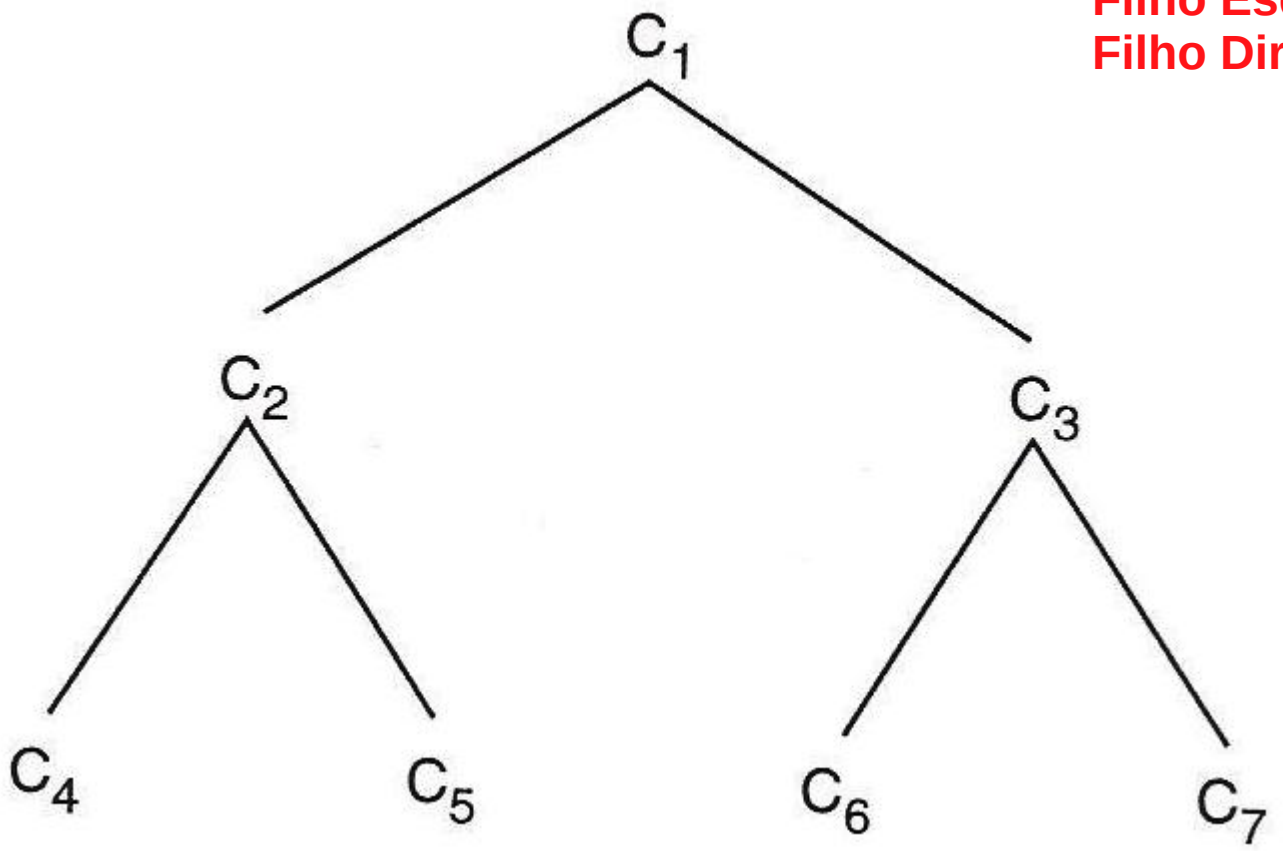
- Onde: C_1 é raiz da árvore;

$$\left.\begin{array}{l} C_{2i} = \text{subárvore da esquerda de } C_i \\ C_{2i+1} = \text{subárvore da direita de } C_i \end{array}\right\} \text{para } i = 1, n \text{ div } 2$$

Heap Sort

Se começar de ZERO:

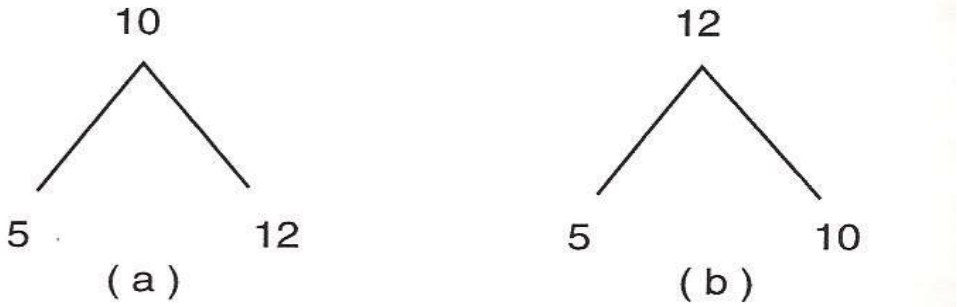
Raiz: Ci
Filho Esquerda: 2i + 1
Filho Direita: 2i + 2



Instituto Federal do Espírito Santo - Ifes
Campus Colatina
Professor: Giovany Frossard Teixeira

Heap Sort

- Quando todas as raízes das subárvores satisfazem essas condições, dizemos que a árvore forma um ***heap***, daí a denominação do método.



Transformação de uma subárvore em heap.

Heap Sort

```
void troca( int *a, int *b) {
   int aux;
   aux= *a;
   *a = *b;
   *b = aux;
}

int FilhoEsquerda(int raiz){
    return 2 * raiz + 1;
}

int FilhoDireita(int raiz){
    return 2 * raiz + 2;

}
```

```
void heapify (int *v,  int n, int indice_raiz ) {
    int esquerda = FilhoEsquerda(indice_raiz);
    int direita  = FilhoDireita(indice_raiz);
    int max;
    if(esquerda>n-1){
        return;
    }
    else if (direita>n-1){
        max = esquerda;
    }
    else {
        if(v[esquerda] > v[direita]){
            max=esquerda;
        }
        else{
            max=direita;
        }
    }
    if (v[max] > v[indice_raiz]) {
        troca(&v[max], &v[indice_raiz]);
        heapify(v, n, max);
    }

}
```

Heap Sort

```
void construirHeap(int *v, int n){
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(v, n, i);
    }
}

void heapSort(int *v, int n){
    construirHeap(v, n);
    for(int i = n-1; i > 0; i--){
        troca(&v[i], &v[0]);
        heapify(v, i, 0);
    }
}
```

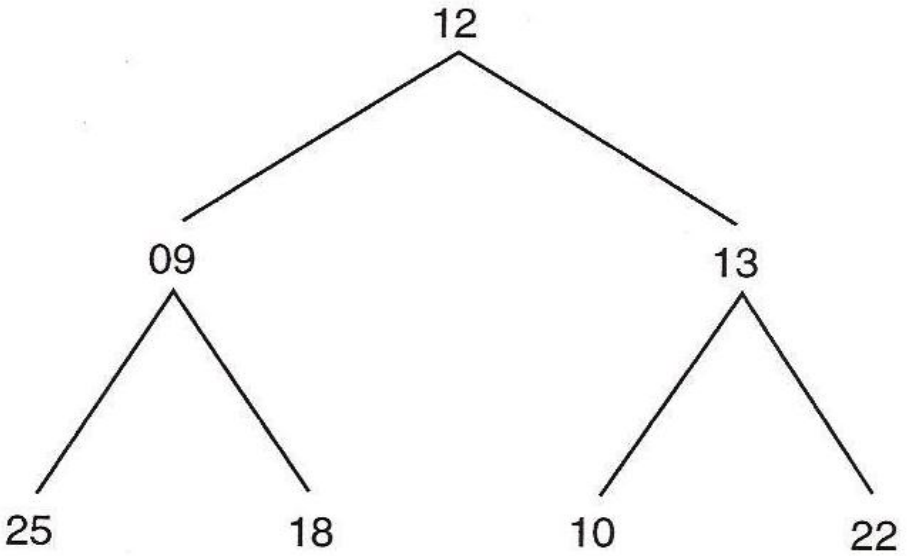
Exemplo

```
void escreveVetor(int *vetor, int n){
    for(int i=0; i<n; i++)
        printf("vet[%d] vale %d \n", i, vetor[i]);
    printf("\n\n");
}


int _tmain(int argc, _TCHAR* argv[])
{
    int vet[7] = {12, 9, 13, 25, 18, 10, 22};
    heapSort(vet, 7);
    escreveVetor(vet, 7);
    return 0;
}
```

Exemplo

0	1	2	3	4	5	6
12	09	13	25	18	10	22

Cuja interpretação sob a forma de árvore é:



- Esse algoritmo começa pela subárvore cuja raiz é 13, já que seu índice no vetor é n/2 – 1.

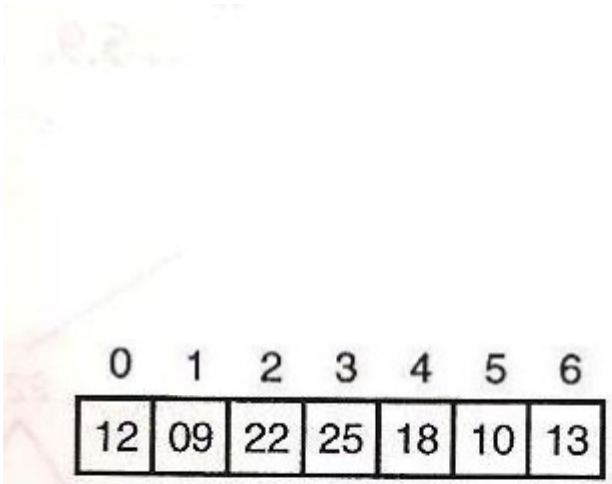
```
void construirHeap(int *v, int n){
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(v, n, i);
    }
}
```

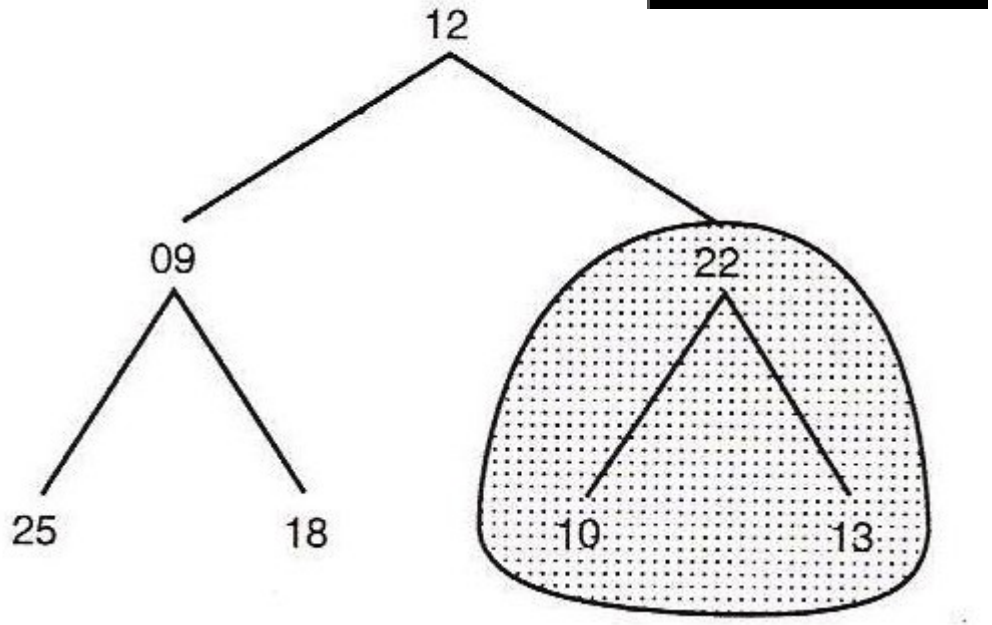
Exemplo

```
void construirHeap(int *v, int n){
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(v, n, i);
        escreveVetor(v, n);
    }
}
```

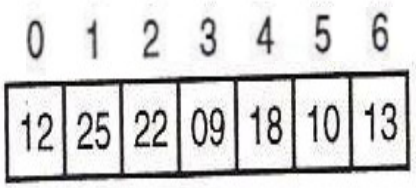
- Estamos construindo a Heap através do método construirHeap.
- Sua primeira chamada gera o resultado apresentado pelas imagens abaixo e ao lado.

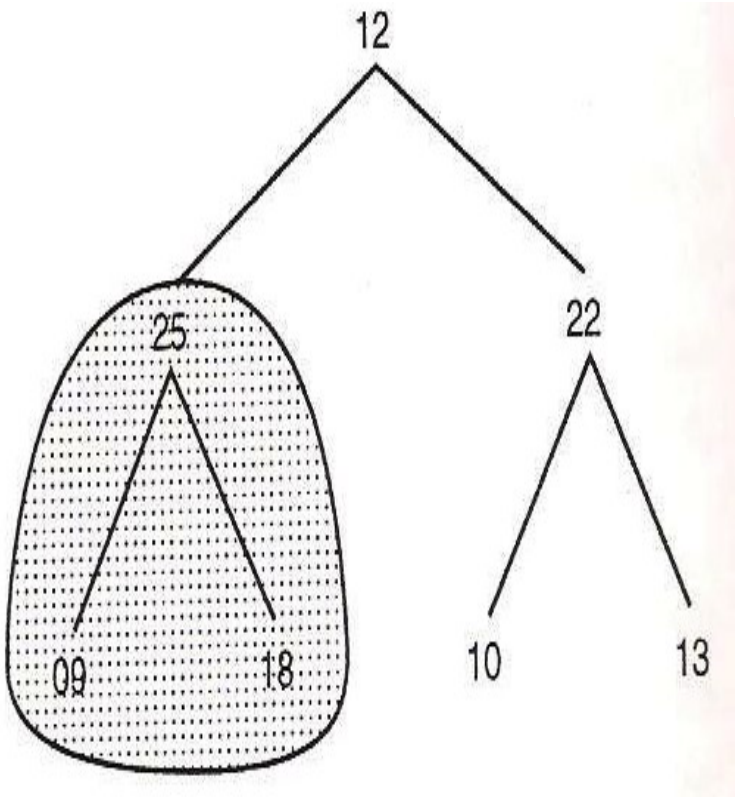
```
vet[0] vale 12
vet[1] vale 9
vet[2] vale 22
vet[3] vale 25
vet[4] vale 18
vet[5] vale 10
vet[6] vale 13
```





Exemplo






```
void construirHeap(int *v, int n){
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(v, n, i);
        escreveVetor(v, n);
    }
}
```

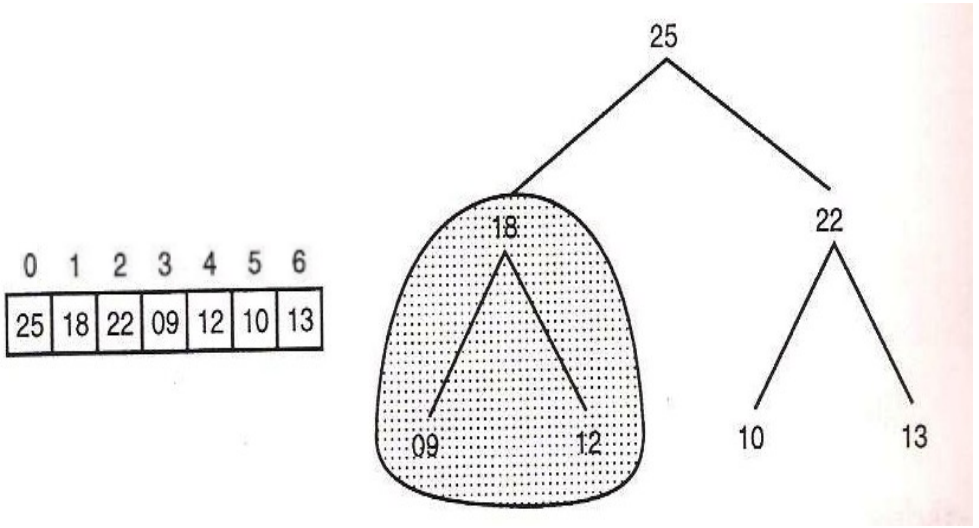
```
vet[0] vale 12
vet[1] vale 9
vet[2] vale 22
vet[3] vale 25
vet[4] vale 18
vet[5] vale 10
vet[6] vale 13


vet[0] vale 12
vet[1] vale 25
vet[2] vale 22
vet[3] vale 9
vet[4] vale 18
vet[5] vale 10
vet[6] vale 13
```

- Passo seguinte

Exemplo

```
void construirHeap(int *v, int n){
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(v, n, i);
        escreveVetor(v, n);
    }
}
```

```
vet[0] vale 12
vet[1] vale 9
vet[2] vale 22
vet[3] vale 25
vet[4] vale 18
vet[5] vale 10
vet[6] vale 13

vet[0] vale 12
vet[1] vale 25
vet[2] vale 22
vet[3] vale 9
vet[4] vale 18
vet[5] vale 10
vet[6] vale 13

vet[0] vale 25
vet[1] vale 18
vet[2] vale 22
vet[3] vale 9
vet[4] vale 12
vet[5] vale 10
vet[6] vale 13
```



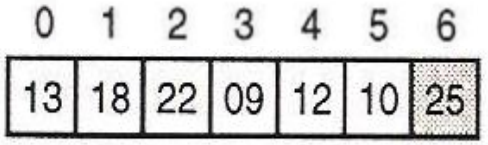
- Agora temos uma Heap !!!

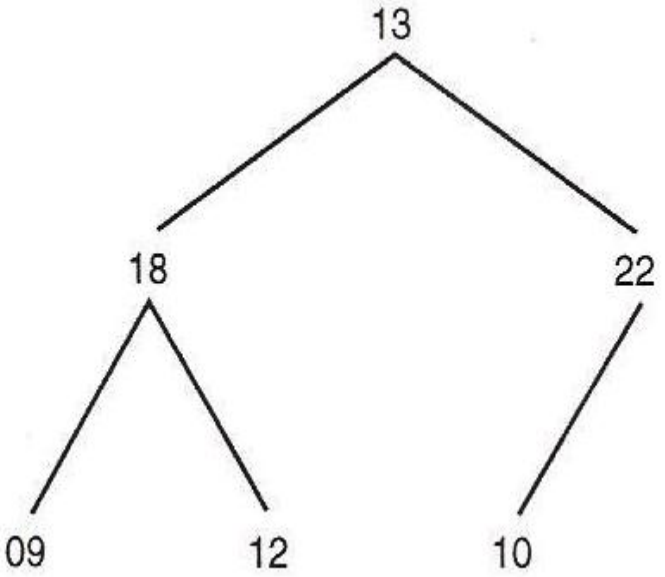
Exemplo

- Se a chave na raíz é a maior de todas, então sua posição definitiva correta na ordem crescente é na última posição do vetor, onde ela deverá ser colocada, por troca pelo elemento que está ocupando aquela posição.
- Com o maior elemento já ocupando a posição definitiva podemos considerar o vetor como tendo uma posição a menos (para fins de ordenação). Mas deveremos chamar a função heapify para transformar novamente nosso vetor numa heap.

```
void heapSort(int *v, int n){
    construirHeap(v, n);
    for(int i = n-1; i > 0; i--){
        troca(&v[i], &v[0]);
        heapify(v, i, 0);
    }
}
```

Exemplo



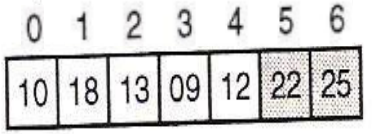


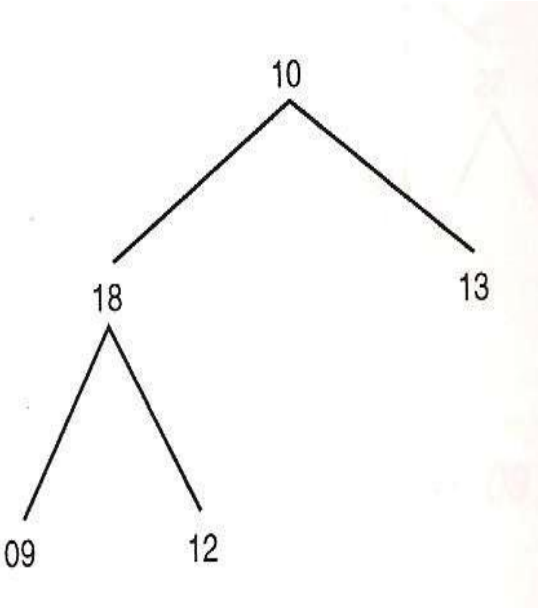

```
vet[0] vale 13
vet[1] vale 18
vet[2] vale 22
vet[3] vale 9
vet[4] vale 12
vet[5] vale 10
vet[6] vale 25
```

```
void heapSort(int *v, int n){
    construirHeap(v, n);
    for(int i = n-1; i > 0; i--){
        troca(&v[i], &v[0]);
        escreveVetor(v, n);
        heapify(v, i, 0);
    }
}
```

Exemplo

```
vet[0] vale 13
vet[1] vale 18
vet[2] vale 22
vet[3] vale 9
vet[4] vale 12
vet[5] vale 10
vet[6] vale 25


vet[0] vale 10
vet[1] vale 18
vet[2] vale 13
vet[3] vale 9
vet[4] vale 12
vet[5] vale 22
vet[6] vale 25
```



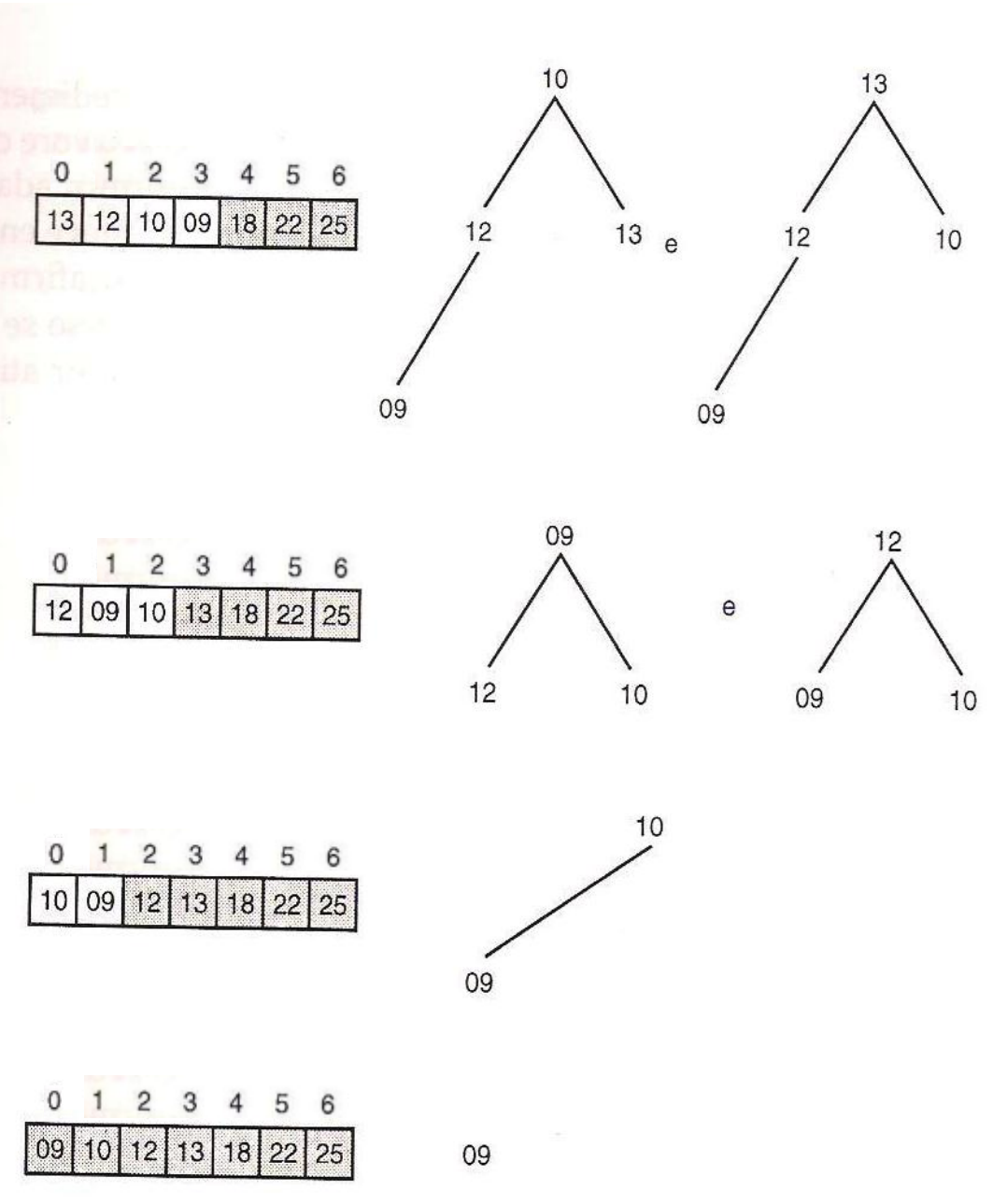



```
void heapSort(int *v, int n){
    construirHeap(v, n);
    for(int i = n-1; i > 0; i--){
        troca(&v[i], &v[0]);
        escreveVetor(v, n);
        heapify(v, i, 0);
    }
}
```

Exemplo

```
void heapSort(int *v, int n){
    construirHeap(v, n);
    for(int i = n-1; i > 0; i--){
        troca(&v[i], &v[0]);
        escreveVetor(v, n);
        heapify(v, i, 0);
    }
}
```

```
vet[0] vale 13
vet[1] vale 18
vet[2] vale 22
vet[3] vale 9
vet[4] vale 12
vet[5] vale 10
vet[6] vale 25

vet[0] vale 10
vet[1] vale 18
vet[2] vale 13
vet[3] vale 9
vet[4] vale 12
vet[5] vale 22
vet[6] vale 25

vet[0] vale 10
vet[1] vale 12
vet[2] vale 13
vet[3] vale 9
vet[4] vale 18
vet[5] vale 22
vet[6] vale 25

vet[0] vale 9
vet[1] vale 12
vet[2] vale 10
vet[3] vale 13
vet[4] vale 18
vet[5] vale 22
vet[6] vale 25

vet[0] vale 10
vet[1] vale 9
vet[2] vale 12
vet[3] vale 13
vet[4] vale 18
vet[5] vale 22
vet[6] vale 25

vet[0] vale 9
vet[1] vale 10
vet[2] vale 12
vet[3] vale 13
vet[4] vale 18
vet[5] vale 22
vet[6] vale 25
```



Instituto Federal do Espírito Santo - Ifes
Campus Colatina
Professor: Giovany Frossard Teixeira

Análise de desempenho

- *Função heapify: possui complexidade de O($log_2 n$)* pois cada troca e comparação tem custo O(1) e são feitas no máximo *$log_2 n$* trocas, pois, cada vez que avançamos na árvore descemos um nível, como a árvore tem log n níveis (altura de uma árvore) o algoritmos é *O($log_2 n$)*.
- Função construirHeap: possui um for executado n/2 – 1 vezes, assim, esse algoritmo é O(n *$log_2 n$).*
- *Função heapSort: executa n-1 fezes a função heapify, como heapify é O($log_2 n$)* essa parte do algoritmo é O(n *$log_2 n$). Como construirHeap é* O(n *$log_2 n$) pela regra da soma nosso algoritmo é* O(n *$log_2 n$) para todos os casos.*

Bibliografia

- AZEREDO, Paulo A. **Métodos de Classificação de Dados**. Rio de Janeiro: Campus,1996.
- SANTOS, Henrique José. **Curso de Linguagem C da UFMG**, apostila.
- FORBELLONE, André Luiz. **Lógica de Programação – A Construção de Algoritmos e Estruturas de Dados.** São Paulo: MAKRON, 1993.