Algoritmos de Ordenação

Estruturas de Dados e Algoritmos

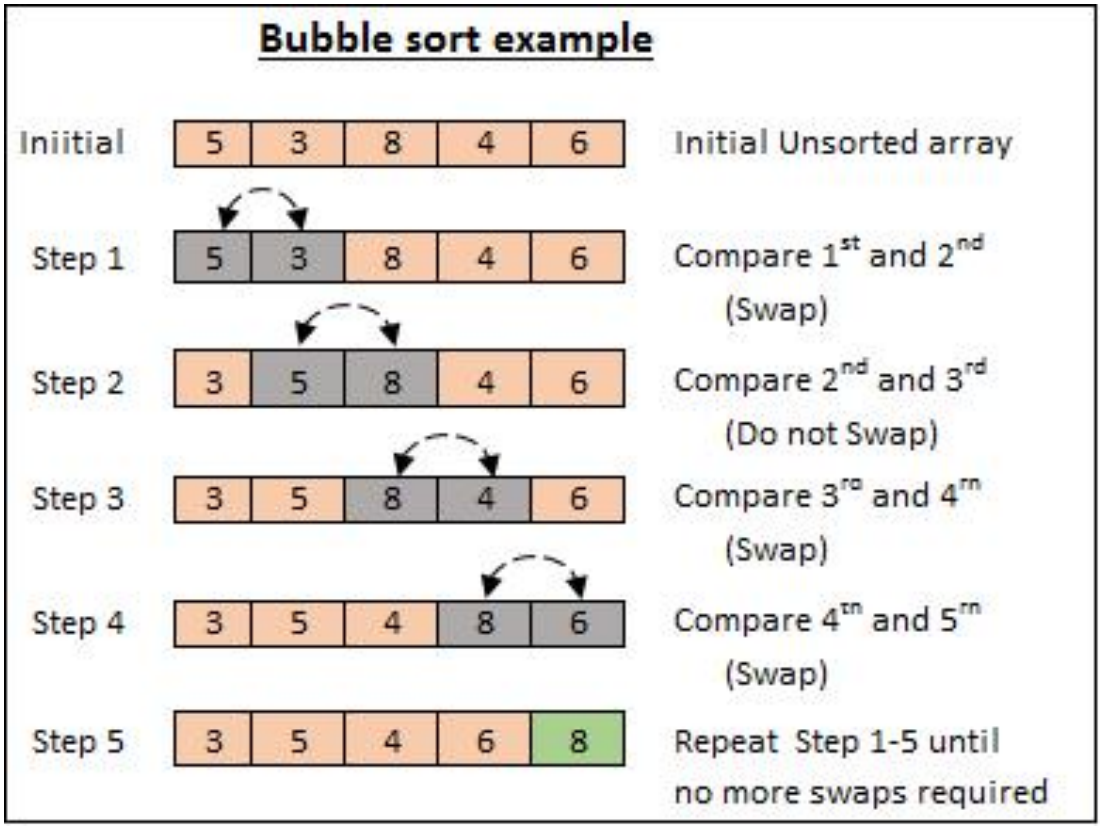
Bubble Sort

Bubblesort

- O bubblesort é um algoritmo de ordenação popular.

- Funciona permutando repetidamente elementos adjacentes que estão fora de ordem

Bubblesort


Bubble sort example
Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd
(Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd
(Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th
(Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th
(Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until
no more swaps required

Algoritmo Bubble sort

```
public static void bubble(int a[]){
    int tam = a.length
    for (int i = tam - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j+1]) {
                int aux = a[j];
                a[j] = a[j+1];
                a[j+1] = aux;
        }
    }
}
```

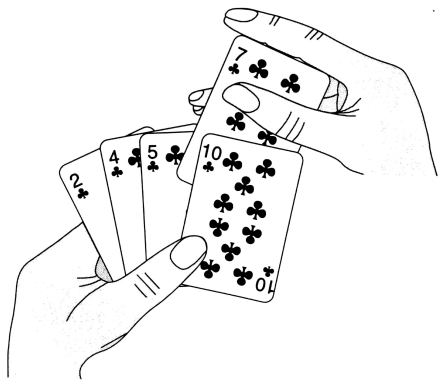
Insertion Sort

Ordenação por inserção

- Algoritmo eficiente para ordenar um número pequeno de elementos
- Funciona de maneria como muitas pessoas ordenam as cartas em um jogo de baralho
- Inicia-se com a mão esquerda vazia e as cartas viradas com a face para baixo da mesa

Ordenação por inserção

- Para encontrar a posição correta de uma carta, compara-se a mesma com cada uma das cartas que já estão na mão, da direita para a esquerda



- Em cada instante, as cartas seguradas na mão esquerda já estão ordenadas

Ordenação por inserção

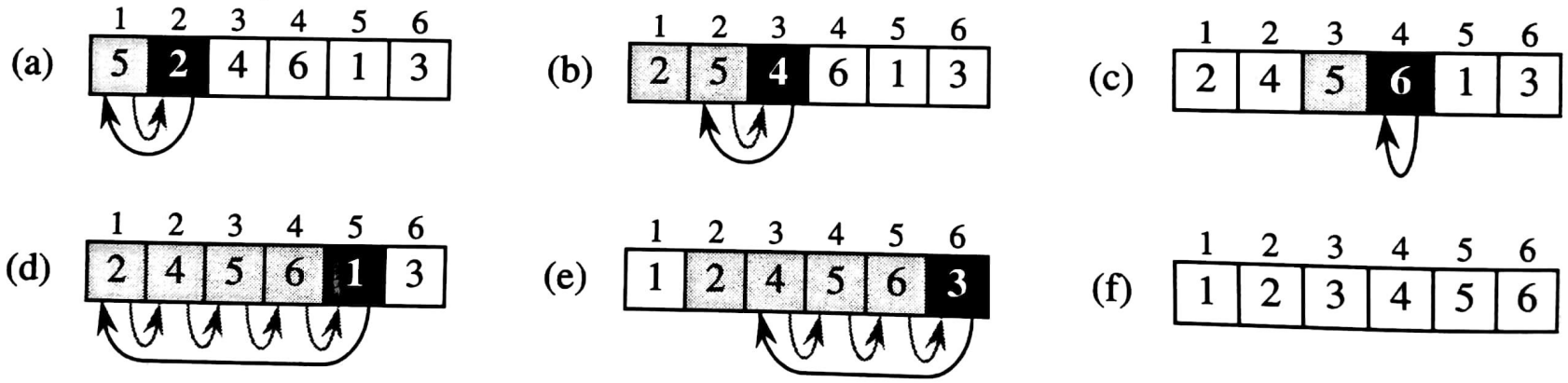


Figura extraída de Cormen et. al (2002)

Ordenação por inserção

```
public static void insertion(int a[]){
    int i, j, chave;
    for (j = 1; j < a.length; j++) {
        chave = a[j];
        i = j - 1;
        while (i >= 0 && a[i] > chave) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = chave;
    }
}
```

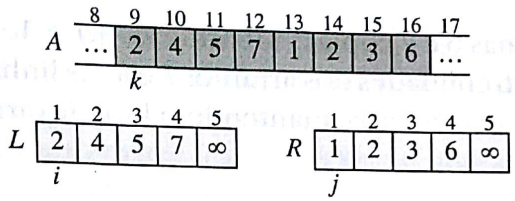
Merge Sort

Ordenação por Intercalação

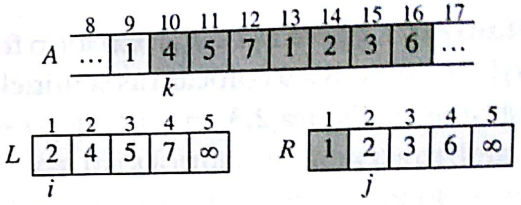
- Também conhecida como Merge sort
- Ordenação que obedece ao paradigma de dividir e conquistar

Ordenação por Intercalação

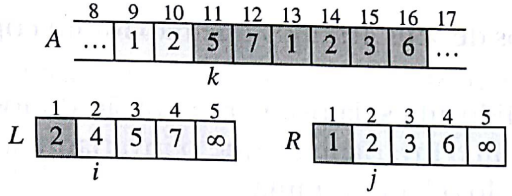
- **Dividir:** Divide uma sequência de *n* elementos a serem ordenados em duas subsequências de n/2 elementos cada uma
- **Conquistar:** Classifica as duas subsequencias recursivamente, utilizando a ordenação por intercalação
- **Combinar:** Faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada


A
8 9 10 11 12 13 14 15 16 17
… 2 4 5 7 1 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

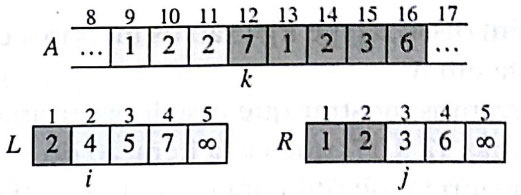
(a)


A
8 9 10 11 12 13 14 15 16 17
… 1 4 5 7 1 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

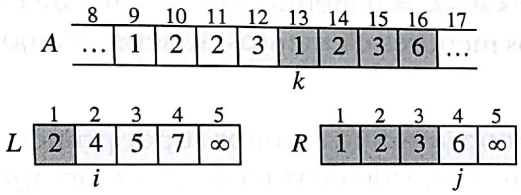
(b)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 5 7 1 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

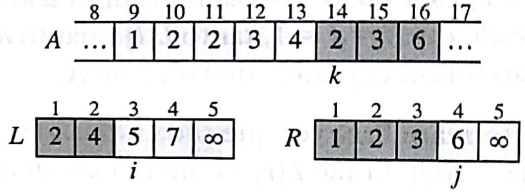
(c)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 7 1 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

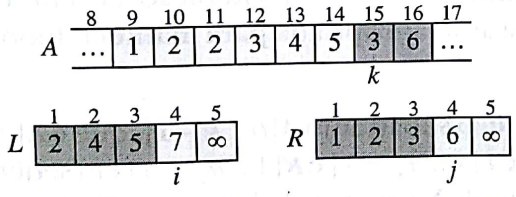
(d)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 3 1 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

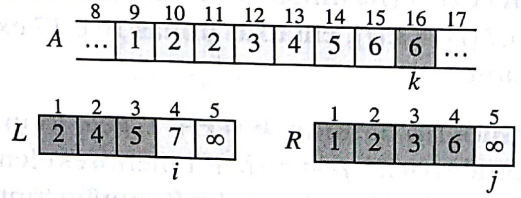
(e)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 3 4 2 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

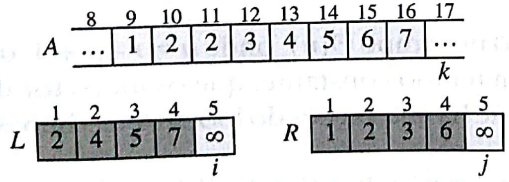
(f)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 3 4 5 3 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(g)

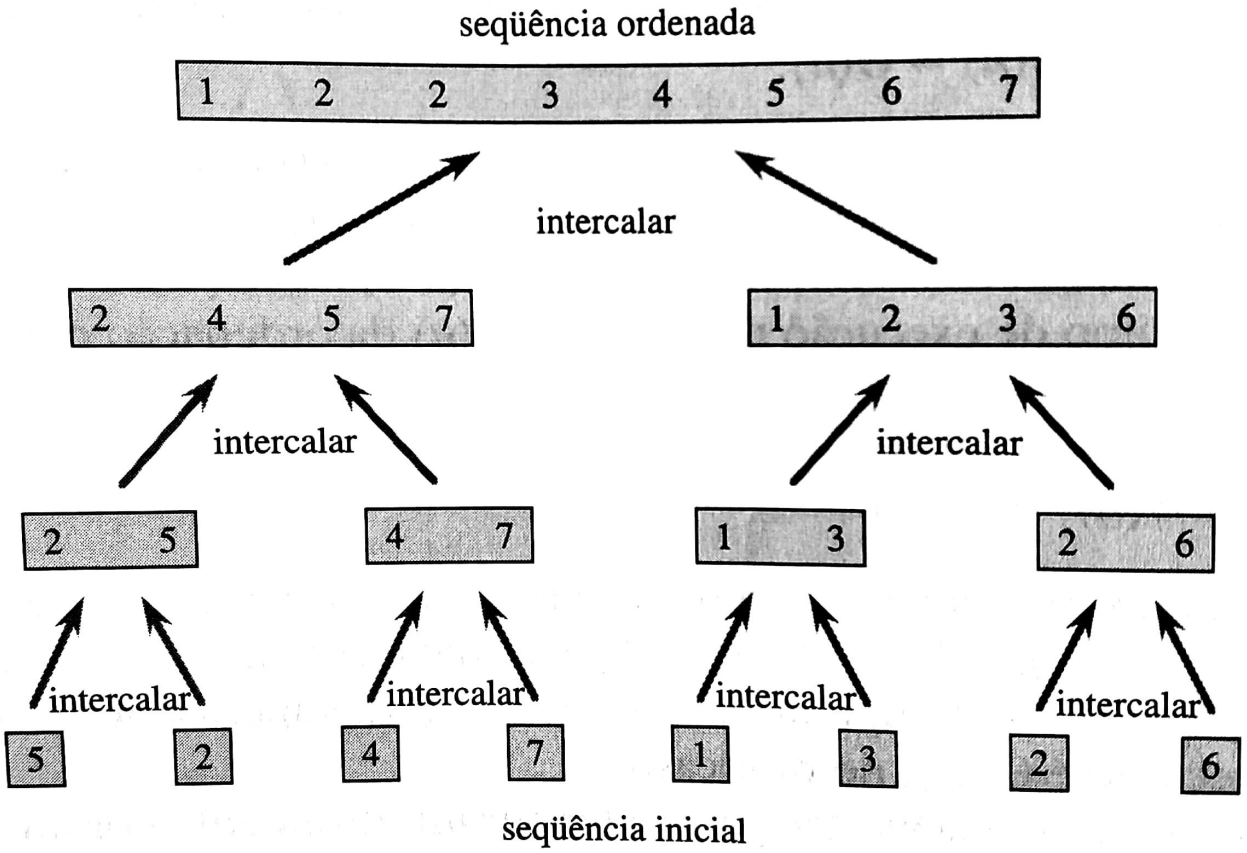

A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 3 4 5 6 6 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(h)


A
8 9 10 11 12 13 14 15 16 17
… 1 2 2 3 4 5 6 7 …
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(i)

Ordenação por Intercalação



Array inicial = [5, 2, 4, 7, 1, 3, 2, 6]

Método merge()

- O algoritmo Merge é responsável por intercalar dois vetores ordenados em um único vetor também ordenado

- Os parâmetros enviados são o vetor, além das posições de início, meio e fim

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- Esse vetor está logicamente dividido em duas partes já ordenadas.
- Cabe ressaltar que as posições de início, meio e fim são necessárias pois o método é executado em diferentes partes de um vetor inicial que está sendo ordenado.

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- A primeira parte do algoritmo é responsável por separar fisicamente o vetor enviado em outros dois vetores, sendo eles o vetor da esquerda e o da direita

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- Isso implica em descobrir o tamanho de cada um desses dois vetores

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- Isso implica em descobrir o tamanho de cada um desses dois vetores
- Alocar o espaço para eles na memória

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- Isso implica em descobrir o tamanho de cada um desses dois vetores
- Alocar o espaço para eles na memória
- E armazenar os valores do vetor inicial nos vetores alocados

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- A segunda parte do algoritmo é responsável por intercalar os elementos dos vetores da esquerda e da direita no vetor inicialmente enviado como parâmetro

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- *j* é o contador do vetor *a[ ]*
  - O início do vetor *a[ ]* está na posição *p* que foi enviada como parâmetro
- *e* é o contador do vetor da esquerda
- *d* é o contador do vetor da direita
  - Ambos vetores da esquerda e direita foram criados dentro do método e iniciam na posição 0 (zero).

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- É realizada uma comparação entre os elementos do vetor da esquerda e da direita

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- É realizada uma comparação entre os elementos do vetor da esquerda e da direita
- Sempre que se descobre qual o valor menor, o mesmo é armazenado no vetor *a[ ]*

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- É realizada uma comparação entre os elementos do vetor da esquerda e da direita
- Sempre que se descobre qual o valor menor, o mesmo é armazenado no vetor *a[ ]*
- Isso implica em incrementar o contador do vetor que teve seu elemento armazenado, além de incrementar o contador do vetor *a[ ]*

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- Esse percorrimento é feito até encontrarmos o final do vetor da esquerda ou o final do vetor da direita

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int i = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método merge()

- A execução da parte anterior termina quando se chega ao final de um dos vetores (da esquerda ou da direita)
- Isso implica que ainda existe elementos sobrando em um dos vetores
- Nessa última parte do algoritmo, se detecta qual dos vetores possui elementos sobrando e se copia esses elementos para o vetor *a[ ]*

```
public static void merge(int a[], int p,
                         int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] esquerda = new int[n1];
    int[] direita = new int[n2];

    for (int i = 0; i < n1; i++) {
        esquerda[i] = a[p+i];
    }
    for (int i = 0; i < n2; i++) {
        direita[i] = a[q+i+1];
    }
    int j = p; int e = 0; int d = 0;
    do {
        if (esquerda[e] <= direita[d]){
            a[j] = esquerda[e];
            e++;
        }
        else {
            a[j] = direita[d];
            d++;
        }
        j++;
    } while ((e < n1) && (d < n2));

    for (int i = e; i < n1; i++) {
        a[j] = esquerda[i];
        j++;
    }
    for (int i = d; i < n2; i++) {
        a[j] = direita[i];
        j++;
    }
}
```

Método MergeSort( )

- O merge( ) é utilizado dentro do método recursivo mergeSort( ).

```
public static void mergeSort(int a[], int p, int q)
    if (p < q) {
        int meio = (p + q)/2;
        mergeSort(a, p, meio);
        mergeSort(a, meio+1, q);
        merge(a, p, meio, q);
     }
}
```

Método MergeSort( )

- O mergeSort() é responsável por calcular a posição do meio do vetor que deverá ser ordenado.
- O método então chama a si mesmo enviando como até o momento em que o vetor esteja reduzido a 1 único elemento (situação em que o mesmo já é ordenado)
- Nesse momento é chamado o merge( ) que começa a intercalação de "baixo para cima"

```
public static void mergeSort(int a[], int p, int q)
    if (p < q) {
        int meio = (p + q)/2;
        mergeSort(a, p, meio);
        mergeSort(a, meio+1, q);
        merge(a, p, meio, q);
     }
}
```

Heap Sort

HeapSort

- Introduz uma estrutura de dados no processo de ordenação chamada de *heap* (monte)
- A estrutura de dados *heap binário* é um objeto de arranjo que pode ser visto como uma árvore binária praticamente completa
- A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda até um certo ponto

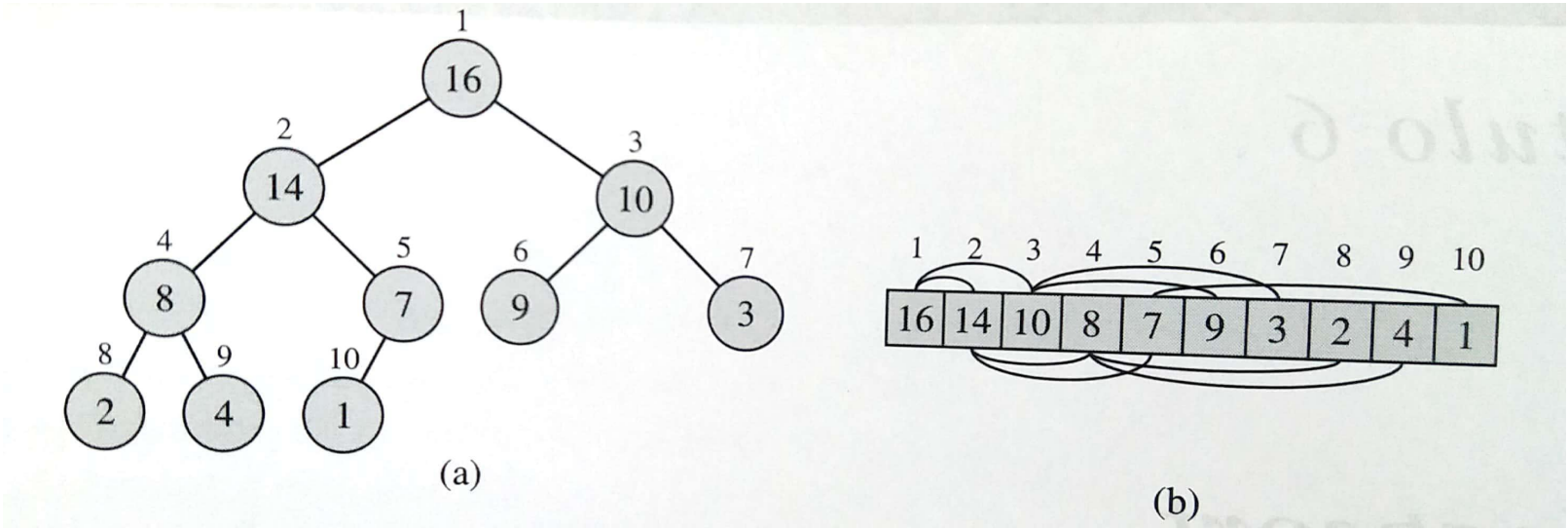
HeapSort

- Um vetor A que representa o heap é um objeto com dois atributos:
  - comprimento: número de elementos contidos no vetor
  - tamanho_do_heap: número de elementos no heap armazanedo dentro do arranjo A.

- Ou seja, embora A[0.. comprimento] possa conter números válidos, nenhum elemento além de A[tamanho_do_heap] é um elemento do heap
  - tamanho_do_heap <= comprimento

HeapSort

- A raiz da árvore é A[0], e a partir do índice de um nó, os índices de seu pai, do filho da esquerda e do filho da direita podem ser calculados de modo simples.



HeapSort

- Cálculos das posições dos nodos filho da esquerda, filho da direita, e pai a partir de um índice do vetor.

```
int left(int i) {
    return 2*i + 1;
}

int right(int i) {
    return 2*i + 2;
}

int father(int i) {
    return (i-1)/2;
}
```

HeapSort

- Existem dois tipos de heaps binários
  - Heaps máximos
  - Heaps mínimos
- Em ambos os casos, os heaps devem satisfazer a uma **propriedade de heap**
  - Em um heap máximo, todo nó é no máximo o valor de seu pai.
  - Dessa forma, em um heap máximo, o maior elemento da árvore é armazenado na raiz.
  - O heap mínimo é organizado de maneira oposta. Ou seja, todo nó é no mínimo o valor de seu pai.

HeapSort

- O algoritmo de HeapSort utiliza heaps máximos

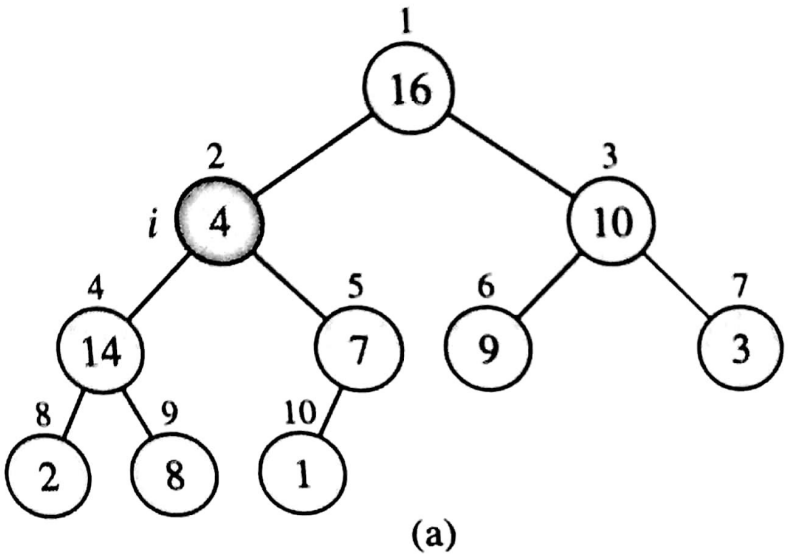
- Os principais procedimentos envolvidos na ordenação por heapSort são:
  - max_heapfy: responsável por manter a propriedade de heap máximo
  - build_max_heap: produz um heap máximo a partir de um vetor de dados não ordenado
  - HeapSort: ordena um vetor localmente

Método max_heapfy

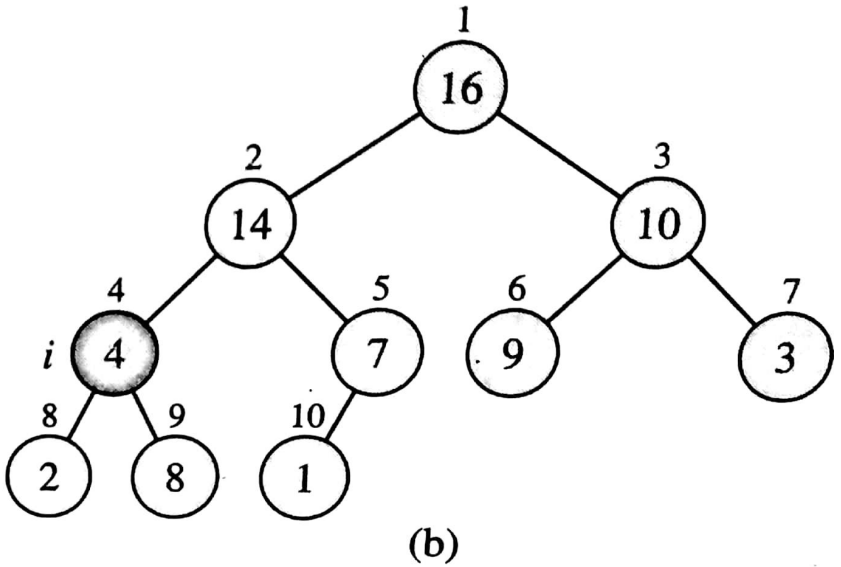
- O Max_heapfy é um método importante para a manipulação de heaps máximos
- As entradas utilizadas para o método são:
  - O vetor A
  - O índice para o vetor

- Quando max_heapfy é chamdo, supomos que as árvores binárias com raízes na esquerda e na direita são heaps máximos, mas que A[i] pode ser menor que seus filhos, violando a propriedade de heap máximo

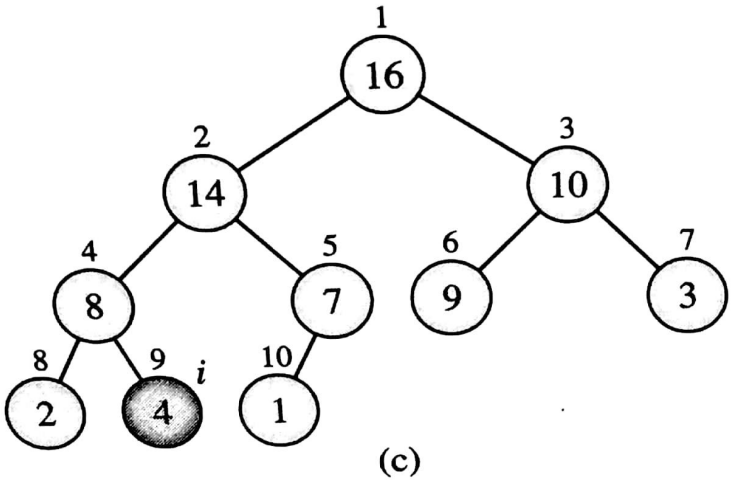
Método max_heapfy



(a)



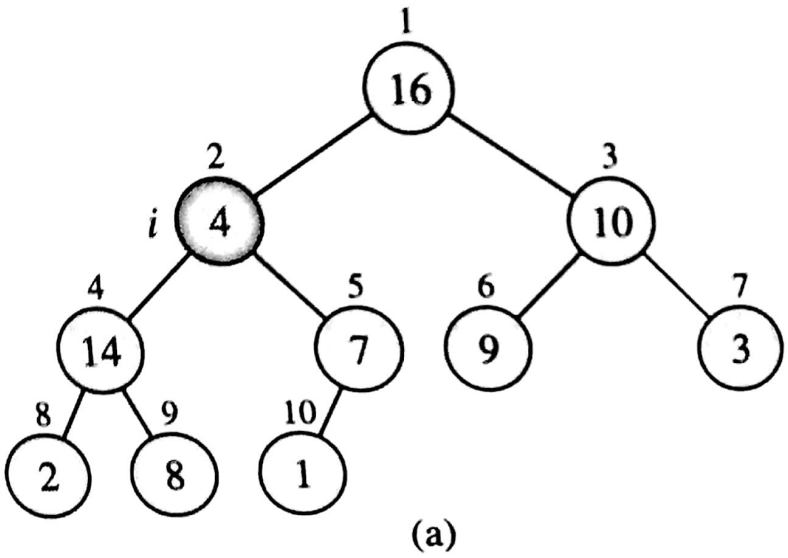
(b)



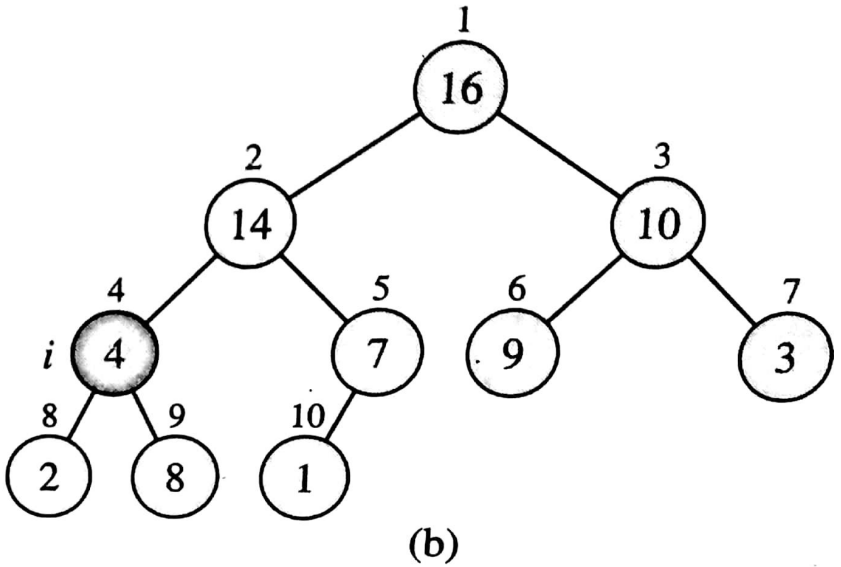
(c)

A função de max_heapfy é deixar que o valor A[i] "flutue para baixo" no heap máximo, de tal forma que a subárvore com raiz no índice i se torne um heap máximo

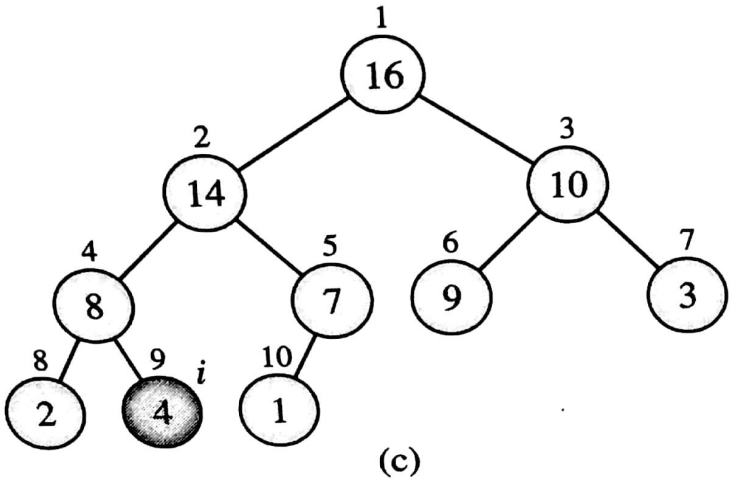
Método max_heapfy



(a)



(b)



(c)

Na figura, para a execução do método no índice 2, o valor de 4 vai descendo e o maior valor de seus filhos sempre subindo.

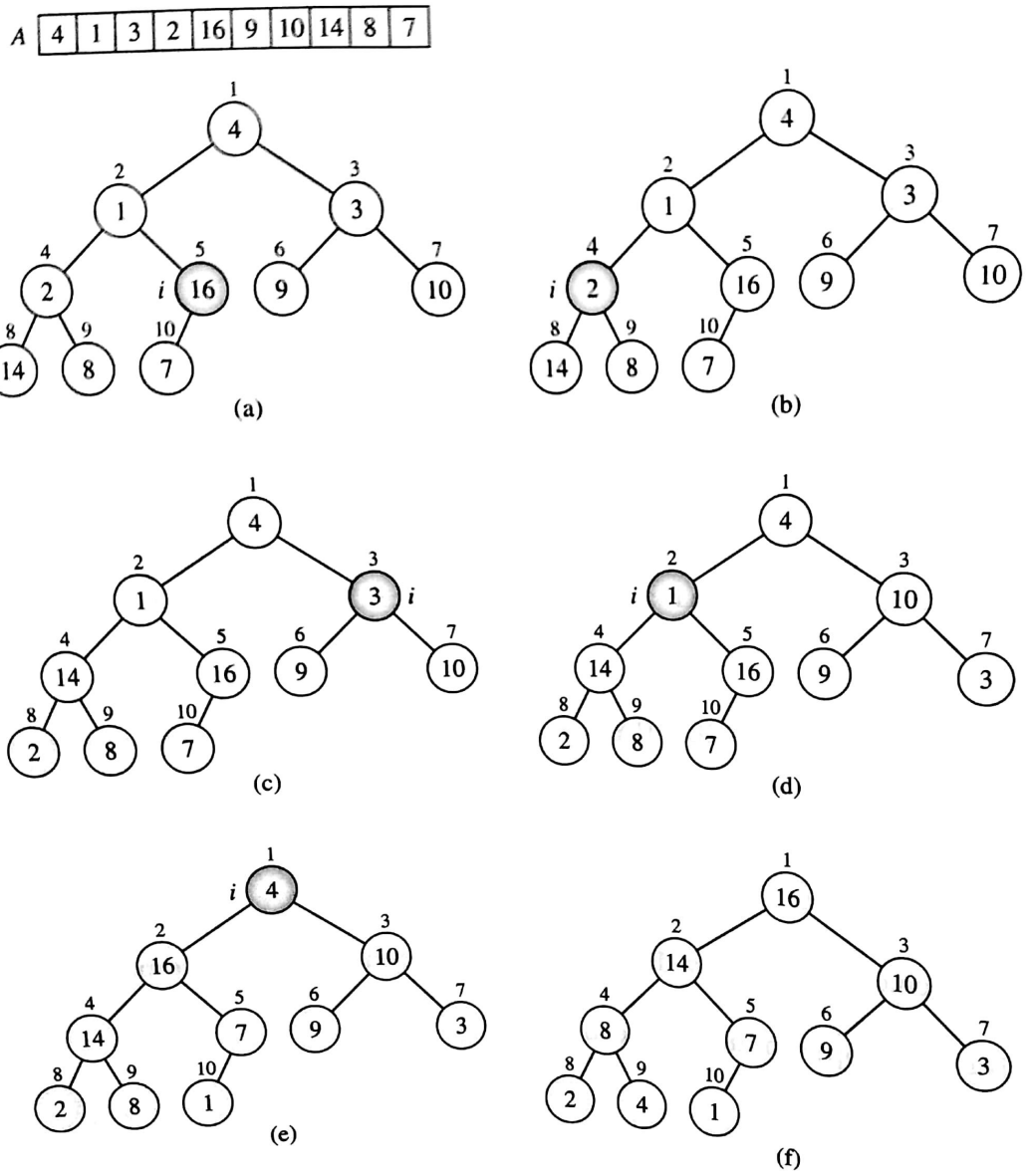
Método max_heapfy

```
27  void maxHeapfy(int a[], int i, int n) {
28      int maior;
29      int esq = left(i);
30      int dir = right(i);
31
32      if ((esq < n) && (a[esq] > a[i])){
33          maior = esq;
34      }
35      else maior = i;
36
37      if ((dir < n) && (a[dir] > a[maior])){
38          maior = dir;
39      }
40
41      if (maior != i) {
42          int aux = a[maior];
43          a[maior] = a[i];
44          a[i] = aux;
45
46          maxHeapfy(a, maior, n);
47      }
48  }
```

Método build_max_heap

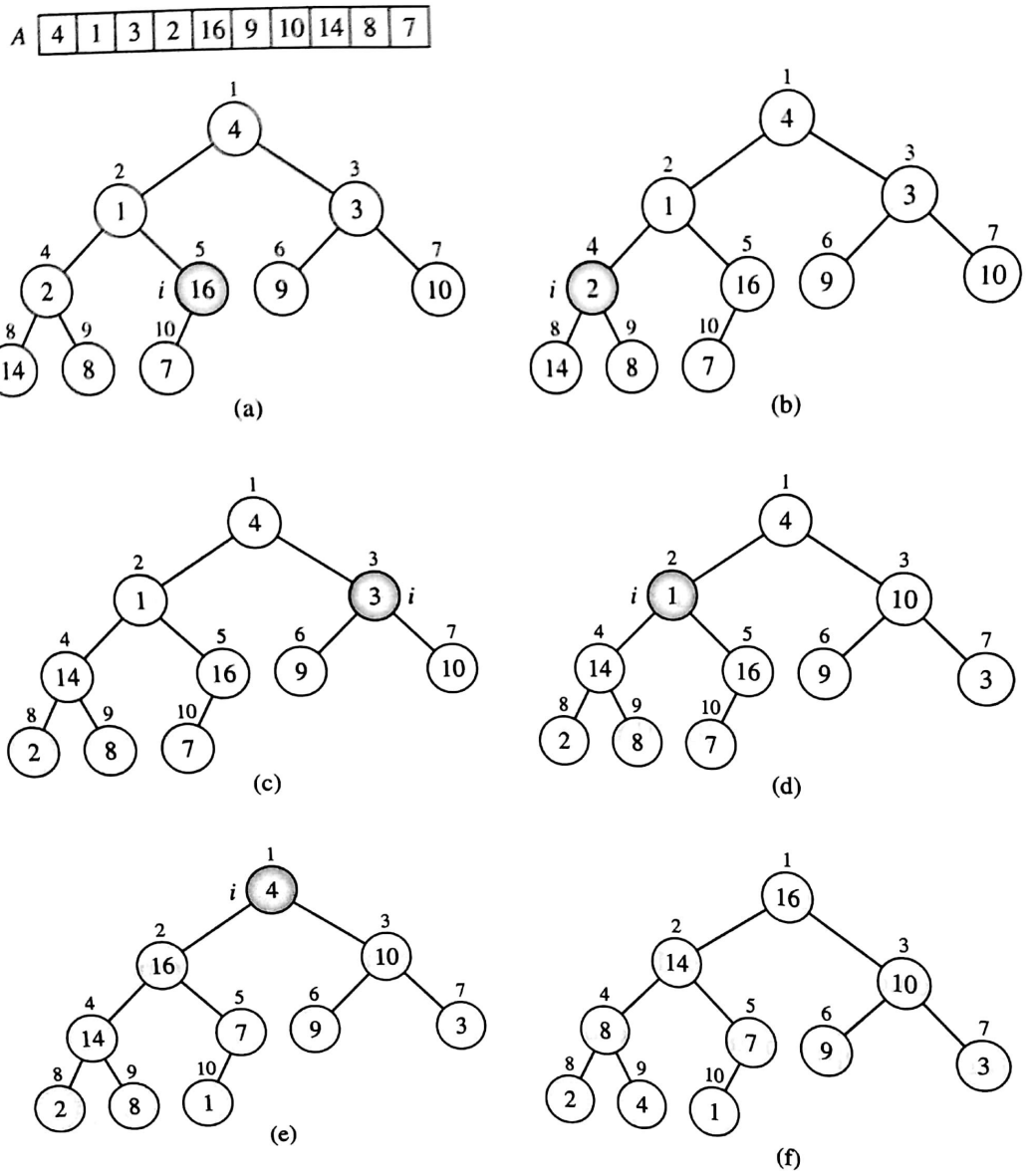
- É possível utilizar o procedimento max_heapfy de baixo para cima, a fim de converter um vetor A[1..n], em um heap máximo
- Todos os elementos folha são heaps de um único elemento e podem ser considerados heaps máximos
- O procedimento build_max_heap percorre os nós restantes da árvore (de baixo para cima) executando o método max_heapfy sobre cada um deles

Método build_max_heap



Na figura, a execução começa pela construção do heap máximo na posição 5 (último elemento do vetor que possui filhos)

Método build_max_heap

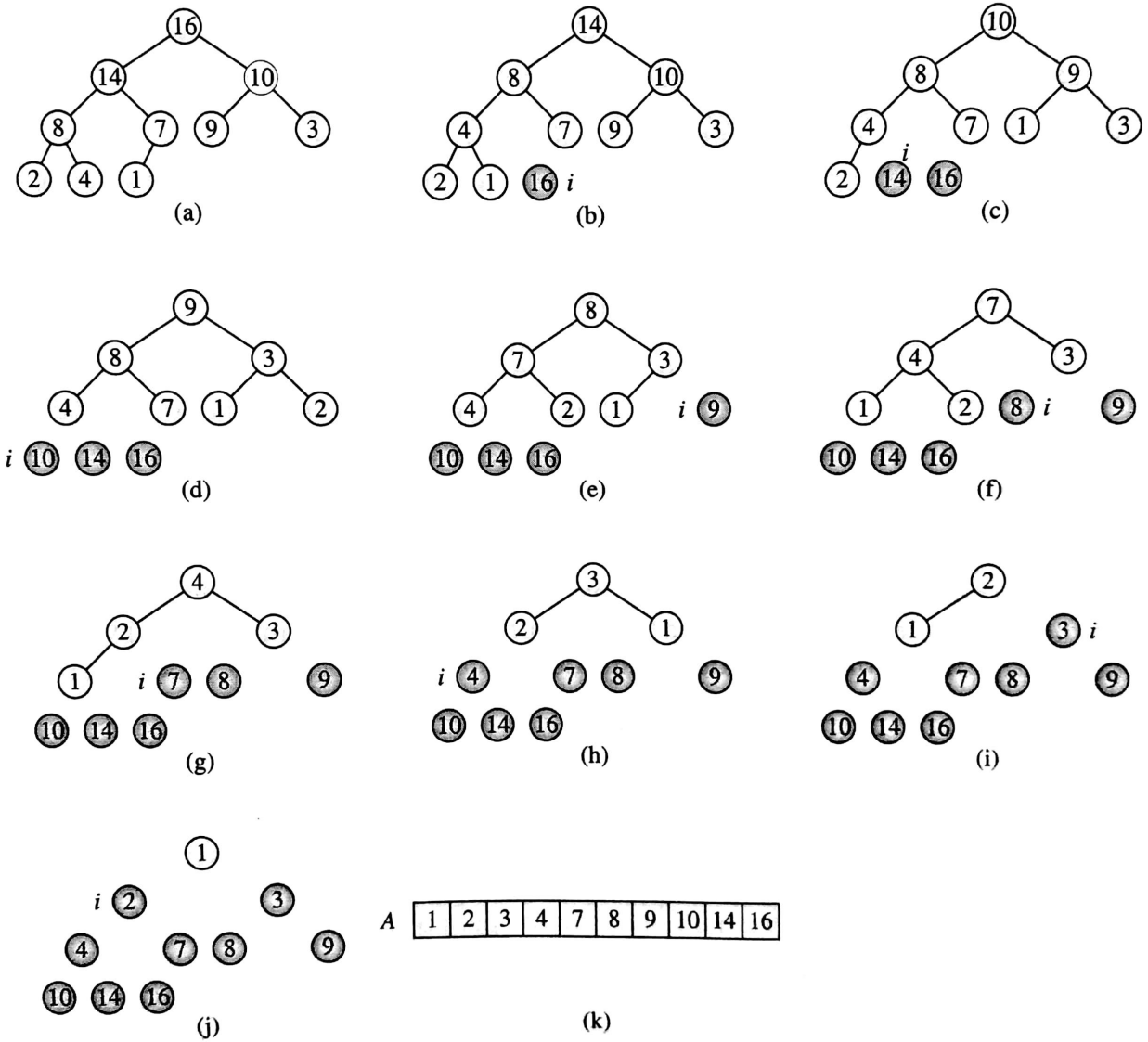


O método segue construindo o heap máximo de maneira decrescente, passando pelos elementos das posições 4, 3, 2 e 1.

Método build_max_heap

```
void buildMaxHeap(int a[], int n) {
    int i;
    for (i = (n/2) -1; i>=0; i--) {
        maxHeapfy(a, i, n);
    }
}
```

Método heapSort



Método heapSort

```
void heapSort(int a[], int n) {

 int i;
 buildMaxHeap(a, n);
 for (i = n-1; i >= 1; i--){
     int aux = a[0];
     a[0] = a[i];
     a[i] = aux;
     n--;
     maxHeapfy(a, 0, n);
 }

}
```

Referências utilizadas

- Textos e figuras extraídos de
  - Cormen, Leiserson, Rivest e Stein. Algoritmos: Tradução da 2a Edição Americana. Teoria e Prática. Editora Campus 2002.