Universidade de Brasília

Ordenação

Algoritmos e Estruturas de Dados

Material gentilmente cedido pelo Prof. Bruno Machiavello

Departamento de Engenharia Elétrica (ENE), Faculdade de Tecnologia (FT)

Roteiro

- Ordenação
  - Bubblesort
  - SelectionSort
  - InsertionSort
  - QuickSort
  - MergeSort
  - HeapSort
- Busca Binária

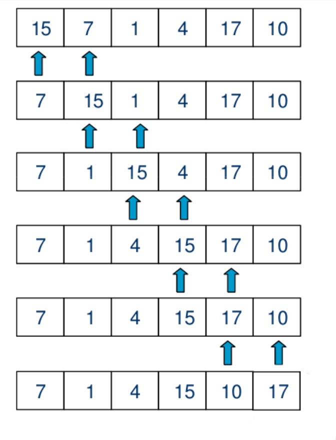
Leitura sugerida: capítulo 9 do livro-texto (Drozdek).

Atenção: embora aqui veremos implementações com vetores, todos os algoritmos podem ser implementados para ordenar qualquer conjunto linear de objetos, e.g. listas ligadas.

Ordenação

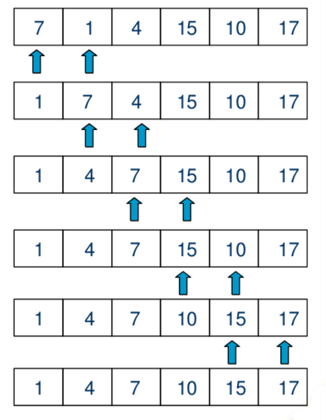
Bubblesort

- Este método consiste em ler todo o vetor, comparando os elementos vizinhos entre si.
- Caso estejam fora da ordem (determinada pela classificação em questão), os mesmos trocam de posição entre si.



Bubblesort

- O vetor está agora mais próximo de uma ordenação, mas ainda não da ordenação desejada.
- Isso indica que devemos repetir o processo mais vezes até que o vetor esteja ordenado.
- Executando mais uma vez o trecho de algoritmo...



```
void bolha (int* v, int n){
    int i,j, temp;
    /* Pior caso: repetir n-1 vezes, de n-1 a 1 */
    for (i=n-1; i>0; i--){
        for (j=0; j<i; j++){
            if (v[j]>v[j+1]) { /* troca */
                temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
            }
        }
    }
}
```

Código 1: Implementação em C/C^{++}

Bubblesort

- O número máximo de execuções do trecho do algoritmo para que o vetor fique ordenado é $N - 1$ vezes, onde N é o número de elementos do vetor.
- É sempre necessário repetir $N - 1$ vezes?
  - No exemplo apresentado em apenas duas execuções do algoritmo o vetor já estava ordenado!
- Como controlar o número de vezes?
  - Se o vetor já estiver ordenado, não precisa repetir o passo mais uma vez.
  - Se não houve trocas entre os elementos do vetor ao executar o trecho do algoritmo, então ele está ordenado.

```c
void bolha2 (int* v, int n){

    int i, j, temp, troca = 1;

    for(i = n-1; troca && i > 0; i--){
        troca = 0;
        for (j=0; j<i; j++){
            if (v[j]>v[j+1]){ /* troca */
                temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        }
    }
}
```

Código 2: Implementação em C/C^{++}

```c
void bolha_rec (int* v, int n){
    int j, temp, troca = 0;

    for (j=0; j<n-1; j++)
        if (v[j]>v[j+1]) { /* troca */
            temp = v[j];
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
        }
    }
    if (troca != 0){ /* houve troca */
        bolha_rec(v, n-1);
    }
}
```

Código 3: Implementação Recursiva em C/C^{++}

Pior Caso
Eficiência do Bubblesort (pior caso):

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Bubblesort

- Eficiência do *BubbleSort* (caso médio):

  - Interrupção após a **1ª passada**:
  - **(n-1)** comparações

  - Interrupção após a **2ª passada**:
  - **(n-1) +(n-2)** comparações

  - Interrupção após a **3ª passada**:
  - **(n-1) +(n-2) +(n-3)** comparações
    - .
    - .
    - .
  - Interrupção após a **última passada possível**::
  - **(n-1) +(n-2) +(n-3) +...+2 + 1** comparações

n-1
eventos
equiprováveis

Bubblesort

- Eficiência do *BubbleSort* (caso médio):

  - Interrupção após a **1ª passada**:
  - **(n-1)** comparações

  - Interrupção após a **2ª passada**:
  - **(n-1) + (n-2)** comparações

  - Interrupção após a **3ª passada**:
  - **(n-1) + (n-2) + (n-3)** comparações
    - .
    - .
    - .
  - Interrupção após a **última passada possível**::
  - **(n-1) + (n-2) + (n-3) +...+ 2 + 1** comparações

n-1 vezes | n-2 vezes | n-3 vezes | 2 vezes | 1 vez

n-1 eventos equiprováveis

Caso Médio
A eficiência do Bubblesort no caso médio é dada por:

$$T(n) = \frac{(n-1)(n-1) + (n-2)(n-2) + \ldots + 2 \cdot 2 + 1 \quad \text{Total de comparações}}{n - 1 \quad \text{Total de eventos possíveis}}$$

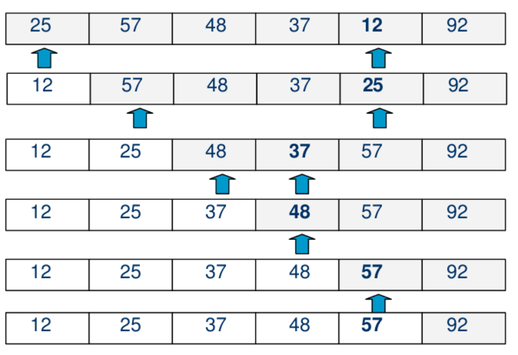
Ou seja:

$$T(n) = \frac{1}{n-1}\sum_{i=1}^{n-1} i^2 = \frac{n}{6}(2n-1) = \frac{n^2}{3} - \frac{n}{6} = O(n^2)$$

SelectionSort

- Identificamos o menor (ou maior) elemento no segmento do vetor que contém os elementos ainda não selecionados.
- Trocamos o elemento identificado com o primeiro elemento do segmento.
- Atualizamos o tamanho do segmento (diminuímos uma posição).
- Interrompemos o processo quando o segmento contiver apenas um elemento.

SelectionSort



Algoritmo 1 SelectionSort em pseudo-código

for $k \leftarrow 0$ to $N-2$ **do**
  $posMenor \leftarrow k$
  for $i \leftarrow k+1$ to $N-1$ **do** {Percorre todo o vetor}
    if $numero[i] < numero[posMenor]$ **then**
      $posMenor \leftarrow i$
    end if
  end for
  if $posMenor \neq k$ **then**
    $aux \leftarrow numero[posMenor]$
    $numero[posMenor] \leftarrow numero[k]$
    $numero[k] \leftarrow aux$
  end if
end for

Selection Sort

- Eficiência:
  - Pior caso $O(n^2)$.
  - Caso médio $O(n^2)$.
- Na prática, para n não muito grande, o Selection sort normalmente é melhor que o Bubblesort, porém é normalmente pior que o Insertion sort.

- A ordenação por inserção funciona da maneira parecida como muitas pessoas ordenam as cartas em um jogo

Insertion Sort

- Iniciaremos com a mão esquerda vazia e as cartas viradas com a face para baixo na mesa.
- Em seguida, removeremos uma carta de cada vez da mesa.
- Vamos compará-la a cada uma das cartas que já estão na mão, da direita para a esquerda, inserindo-a na posição correta na mão esquerda.
- Ou seja, percorremos um vetor da esquerda para a direita e à medida que avançamos, deixamos os elementos mais à esquerda ordenados.
- Eficiência:
    - Pior caso: $O(n^2)$.
    - Caso médio: $O(n^2)$.

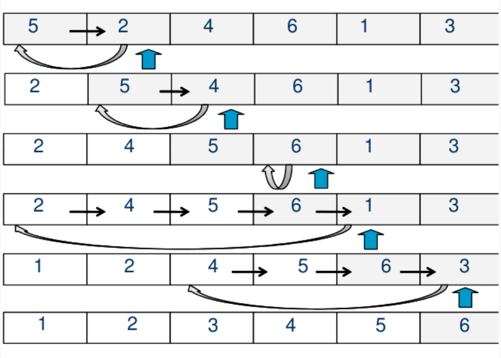


Figura 1: Passo a passo do Insertion Sort

Quicksort

- Este método parte do princípio de que é mais rápido ordenar dois vetores com $\frac{n}{2}$ elementos cada um, do que um com n elementos
- Este é o princípio de projeto de algoritmos chamado de “dividir para conquistar” ou “divisão e conquista”.
  - O primeiro passo é dividir o vetor original.
  - Esse procedimento é denominado particionamento.
  - Deve-se escolher umas das posições do vetor a qual é denominada de pivô:

$$V[i]$$

Quicksort
Uma vez escolhido o pivô, os elementos do vetor são movimentados de forma que:

- O subvetor à esquerda do pivô contenha somente os elementos cujos valores são menores que o pivô.
- O subvetor da direita contenha valores maiores que o valor do pivô.

$$V[0], \ldots, V[i-1] \qquad V[i] \qquad V[i+1], \ldots, V[n-1]$$

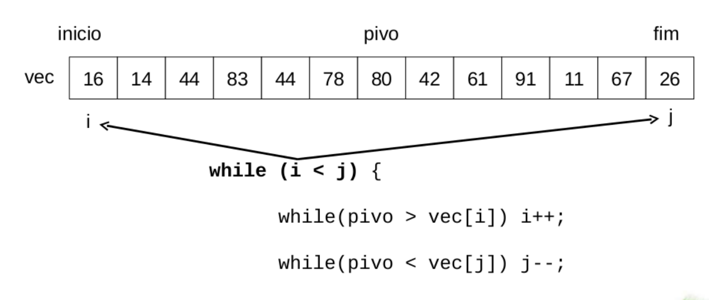
- O procedimento é repetido até que o vetor esteja ordenado.
- Existem várias formas de se escolher o pivô. Vamos escolher o valor do meio do (sub)vetor.

Quicksort: Algoritmo

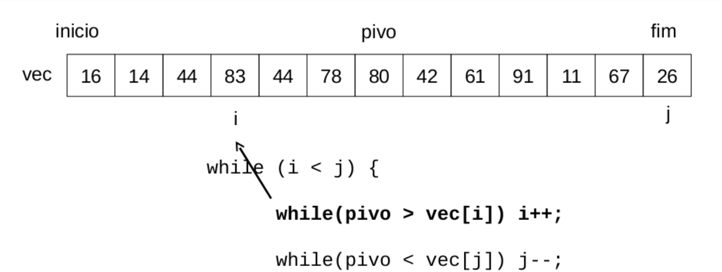
1) Pivô é escolhido no meio do vetor. O elemento é colocado numa variável auxiliar pivo;
2) São iniciadas duas variáveis auxiliares $i = inicio$ e $j = fim$;
3) O vetor é percorrido do *inicio* até que se encontre um $V[i] \geq pivo$ (i é incrementado no processo).
4) O vetor é percorrido a partir do *fim* até que se encontre um $V[j] \leq pivo$ (j é decrementado no processo).
5) $V[i]$ e $V[j]$ são trocados; i é incrementado de 1 e j é decrementado de 1.
6) O processo é repetido até que i e j se cruzem em algum ponto do vetor ($i > j$).
7) Quando são obtidos os dois segmentos do vetor por meio do processo de partição, realiza-se a ordenação de cada um deles de forma recursiva.

- Eficiência no pior caso: $O(n^2)$
- Eficiência no caso médio: $O(n \log_2 n)$

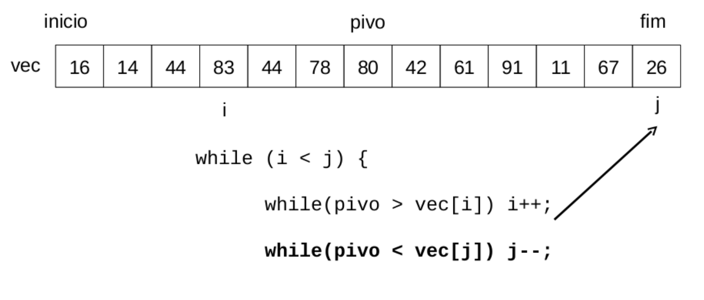
Quicksort: Exemplo



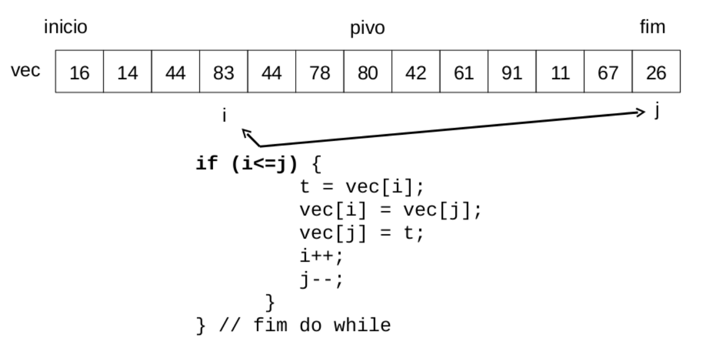
Quicksort: Exemplo



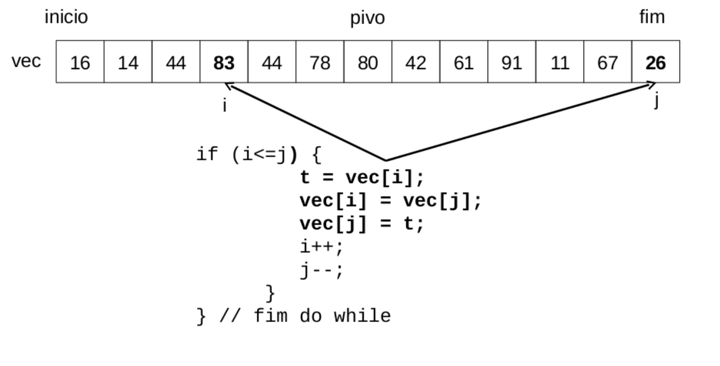
Quicksort: Exemplo

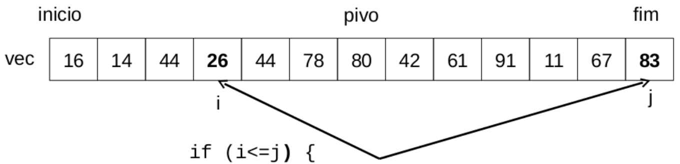


Quicksort: Exemplo



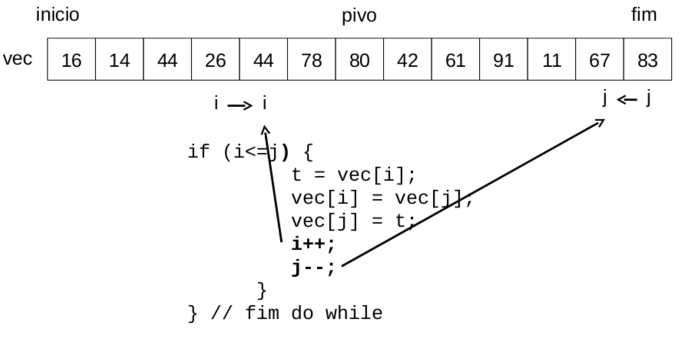
Quicksort: Exemplo



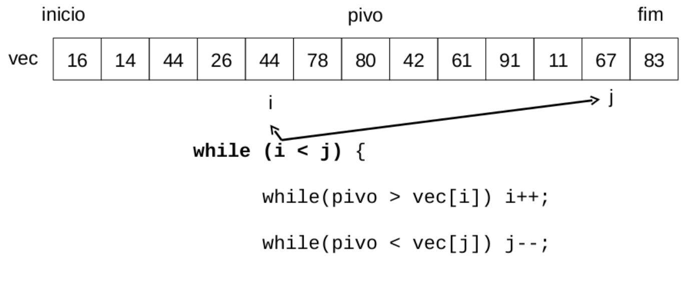



```
if (i<=j) {
        t = vec[i];
        vec[i] = vec[j];
        vec[j] = t;
        i++;
        j--;
     }
} // fim do while
```

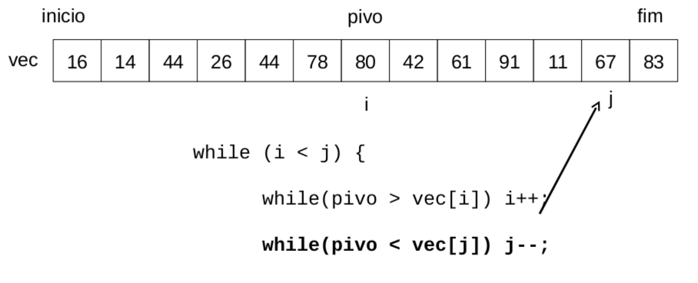
Quicksort: Exemplo



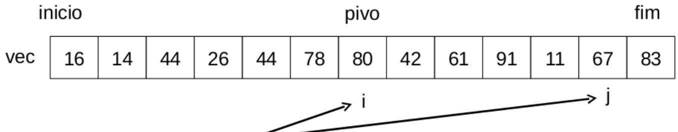
Quicksort: Exemplo



Quicksort: Exemplo

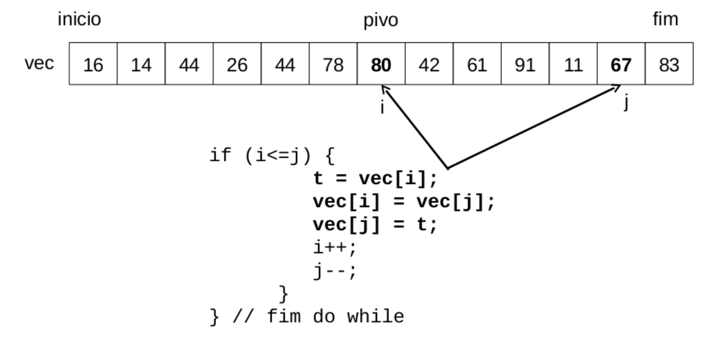


Quicksort: Exemplo

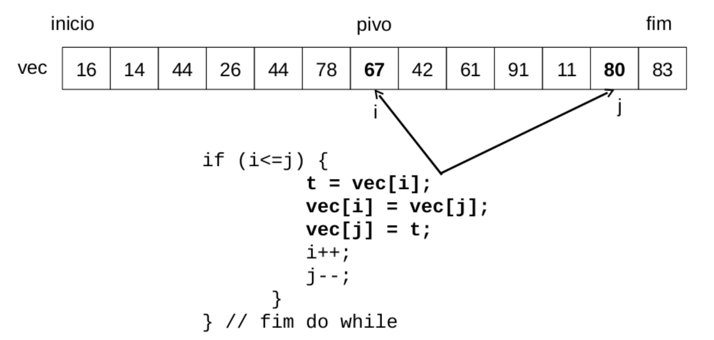



```
if (i<=j) {
        t = vec[i];
        vec[i] = vec[j];
        vec[j] = t;
        i++;
        j--;
    }
} // fim do while
```

Quicksort: Exemplo



Quicksort: Exemplo

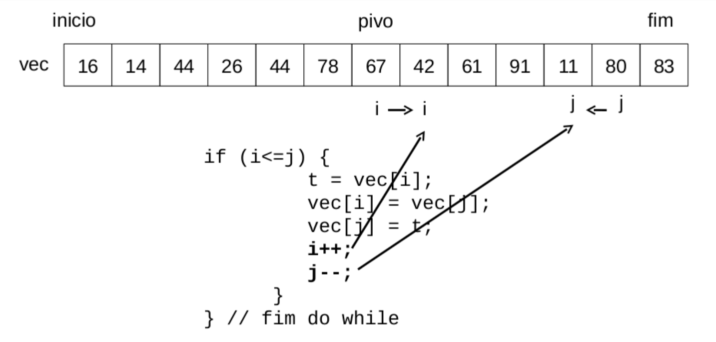


Quicksort: Exemplo

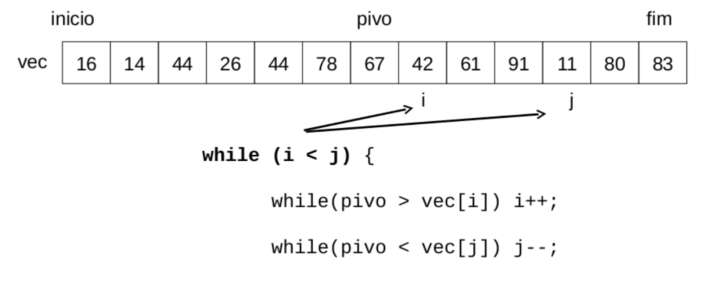
	inicio						pivo						fim
vec	16	14	44	26	44	78	67	42	61	91	11	80	83
							i →	i			j	← j	

```
if (i<=j) {
        t = vec[i];
        vec[i] = vec[j];
        vec[j] = t;
        i++;
        j--;
    }
} // fim do while
```

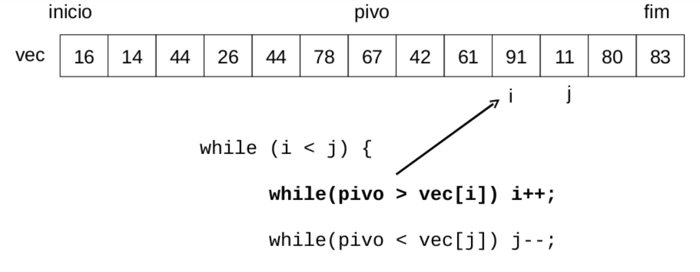
Quicksort: Exemplo



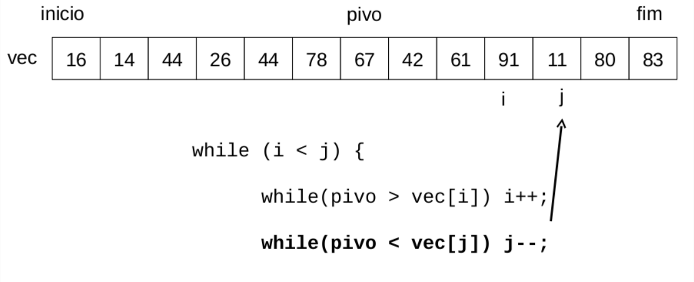
Quicksort: Exemplo



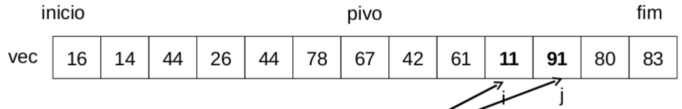
Quicksort: Exemplo



Quicksort: Exemplo

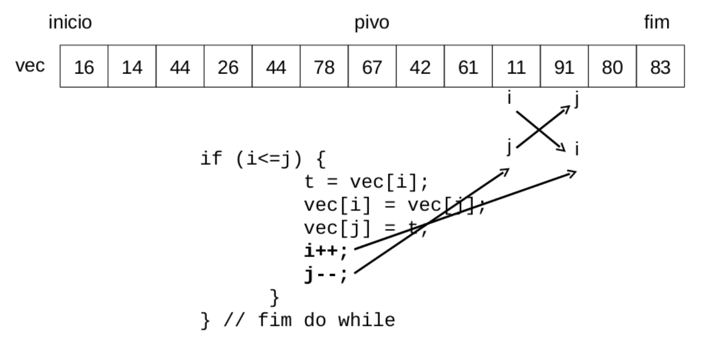


Quicksort: Exemplo

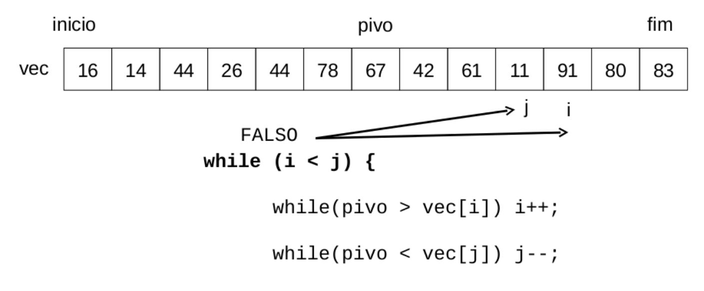



```
if (i<=j) {
        t = vec[i];
        vec[i] = vec[j];
        vec[j] = t;
        i++;
        j--;
    }
} // fim do while
```

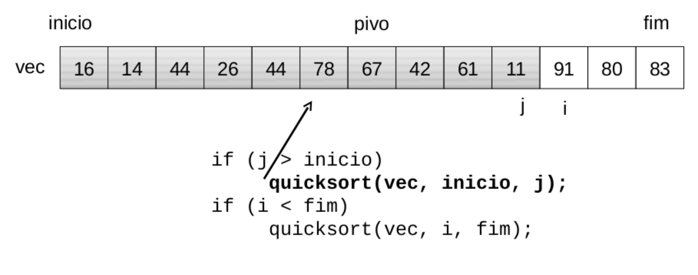
Quicksort: Exemplo



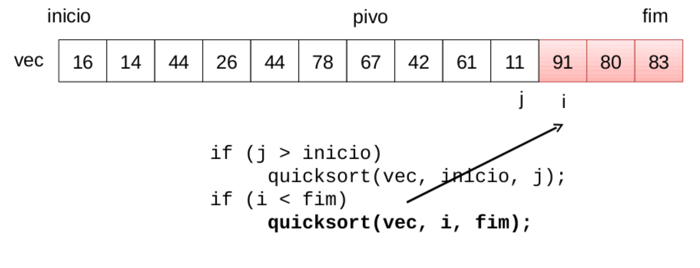
Quicksort: Exemplo



Quicksort: Exemplo



Quicksort: Exemplo



Quicksort: Implementação em C/C^{++}

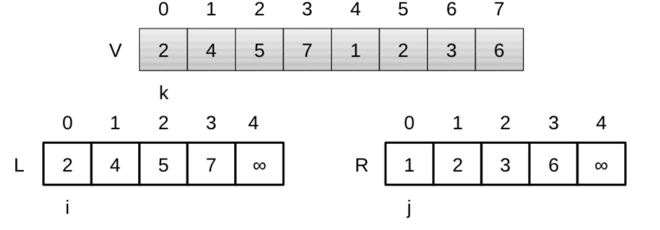
```c
void quicksort(int vec[], int inicio, int fim) {
    int pivo = vec[ (int)(inicio+fim)/2 ];
    int i = inicio, j = fim, temp;
    while (i < j) {
        while(pivo > vec[i]) i++;
        while(pivo < vec[j]) j--;
        if (i<=j) {
            temp = vec[i];
            vec[i] = vec[j];
            vec[j] = temp;
            i++;
            j--;
        }
    }
    if (j > inicio)
      quicksort(vec, inicio, j);
    if (i < fim)
      quicksort(vec, i, fim);
}
```

Mergesort

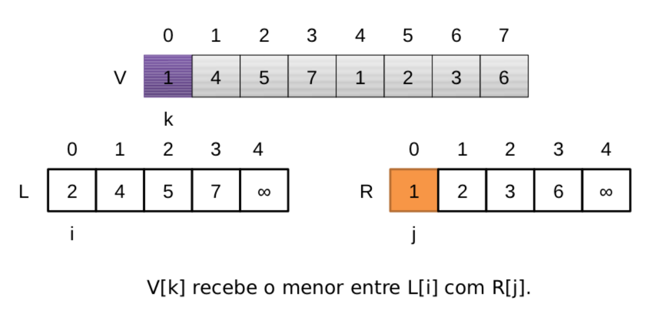
- Também baseado no princípio de dividir para conquistar.
- A idéia básica é criar uma sequência ordenada a partir da mescla de duas outras também ordenadas.
- Usamos um procedimento auxiliar $MERGE(V, i, j, k)$ onde V é um vetor e i,j, e k são índices de elementos do vetor, tais que $i \leq j \leq k$.
- O procedimento pressupõe que os sub-arranjos $V[i..j]$ e $V[j + 1..k]$ estão ordenados.
- Eficência
    - Pior caso: $O(n \log_2 n)$.
    - Caso médio: $O(n \log_2 n)$

.

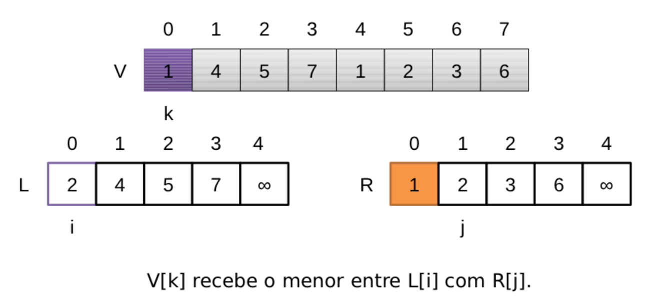
Mergesort: Exemplo do Procedimento *MERGE*



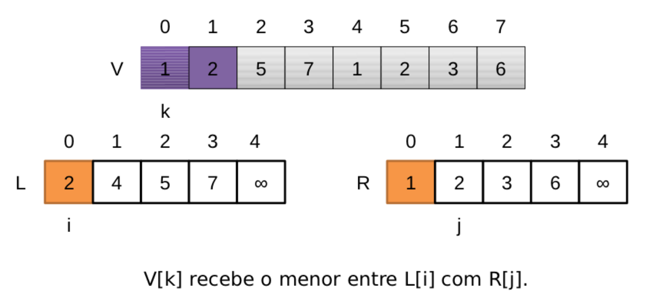
Mergesort: Exemplo do Procedimento *MERGE*



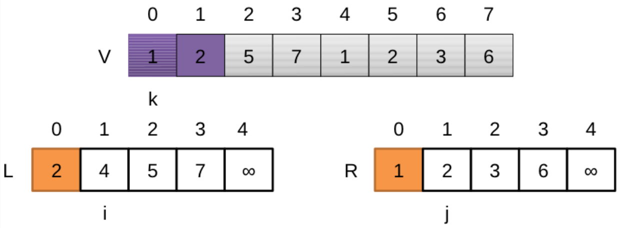
Mergesort: Exemplo do Procedimento *MERGE*



Mergesort: Exemplo do Procedimento *MERGE*

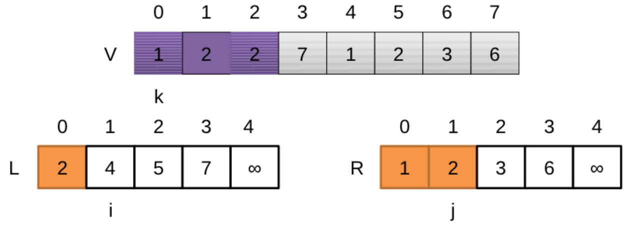


Mergesort: Exemplo do Procedimento *MERGE*



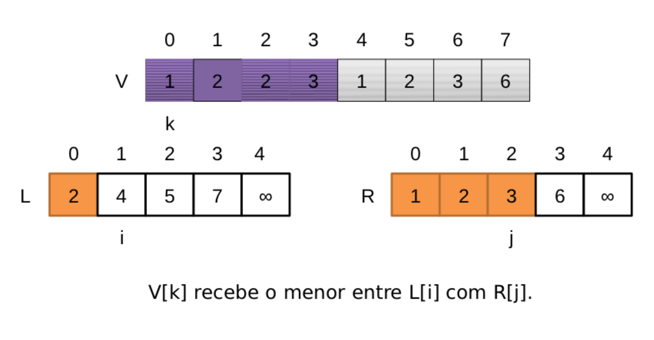
V[k] recebe o menor entre L[i] com R[j].

Mergesort: Exemplo do Procedimento *MERGE*

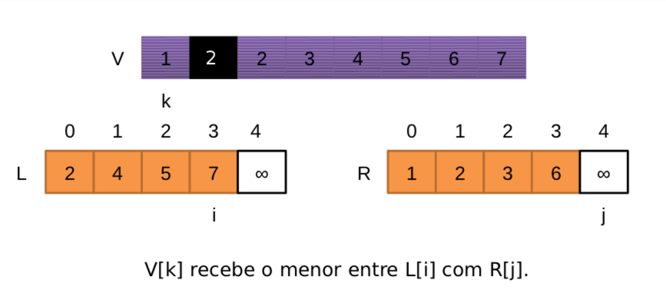


V[k] recebe o menor entre L[i] com R[j].

Mergesort: Exemplo do Procedimento *MERGE*



Mergesort: Exemplo do Procedimento *MERGE*



Mergesort: Implementação

```
void MERGESORT(int V[], int i, int k){
    int j;
    if (i < k){
        j = (i+k)/2
        MERGESORT(V, i, j)
        MERGESORT(V, j + 1, k)
        MERGE(V, i, j, k)
    }
}
```

Código 5: Visão alto nível do Mergesort em C/C^{++}

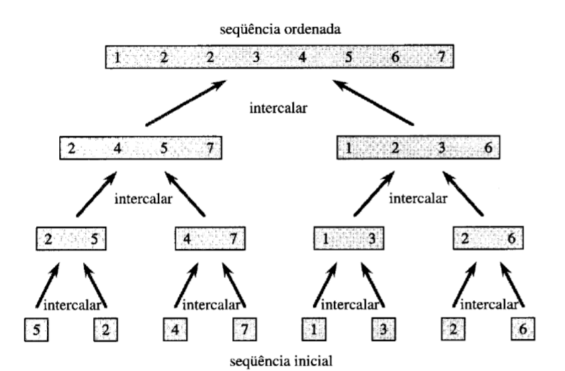


Figura 2: Visão geral do Mergesort

Heapsort

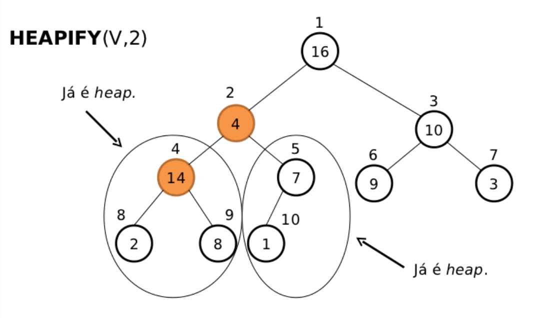
- Constrói-se uma árvore binária quase completa com todos os elementos do vetor.
- Transforma-se a árvore em uma heap: os pais são maiores ou iguais a seus filhos.
- Ordena-se a heap:
- Troca-se o valor da raiz com o valor da posição de maior índice na árvore.
- A posição de maior índice é retirada da árvore.
- Se a propriedade heap não foi mantida, "heapifica-se" a árvore novamente e o processo é repetido até restar apenas um único nó.
- Eficiência:
    - Pior caso: $O(n \log_2 n)$
    - Caso médio: $O(n \log_2 n)$

Heapsort

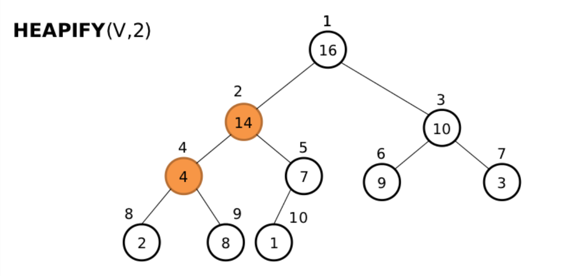
- A função *HEAPIFY*(V, p): suas entradas são um vetor V e um índice p para o vetor.
- Quando é chamada, supomos que as árvores binárias com raízes em $left(p)$ e $right(p)$ são heaps, mas que $V[p]$ pode ser menor que seus filhos, violando assim a propriedade de heap.
- A função de *HEAPIFY* é deixar que o valor em $V[p]$ ”flutue para baixo”, de tal forma que a subárvore com raiz no índice p se torne um heap.

```
HEAPIFY(int V[], int p){
    int L = left(p);
    int R = right(p);
    if (L != NULL e V[L] > V[p]){
        maior = L;
    }
    else{
        maior = p;
    }
    if (R != NULL e V[R] > V[maior]){
        maior = R;
    }
    if (maior != p) {
        troca(V[p], V[maior])
        HEAPIFY(V, maior)
    }
}
```

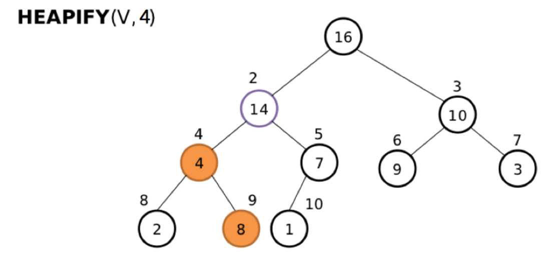
Código 6: Visão alto nível da função *HEAPIFY*


HEAPIFY(V,2)
Já é *heap.*
Já é *heap.*
1
16
2
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

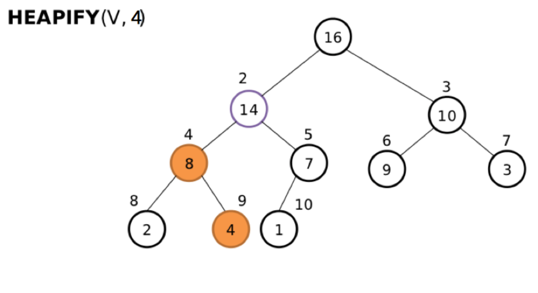
Heapsort: *HEAPIFY*



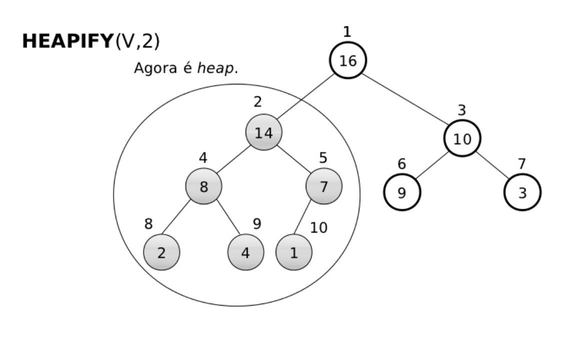
Heapsort: *HEAPIFY*



Heapsort: *HEAPIFY*



Heapsort: *HEAPIFY*

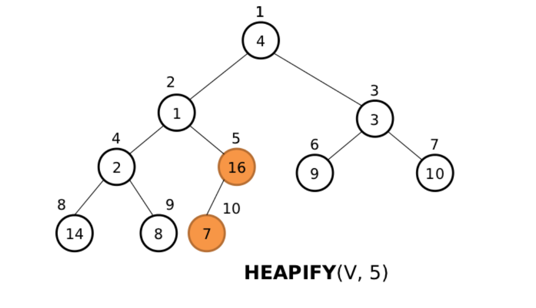


- A função *BUILD_HEAP*(*V*) é usada de baixo para cima, afim de converter um vetor V em um heap.

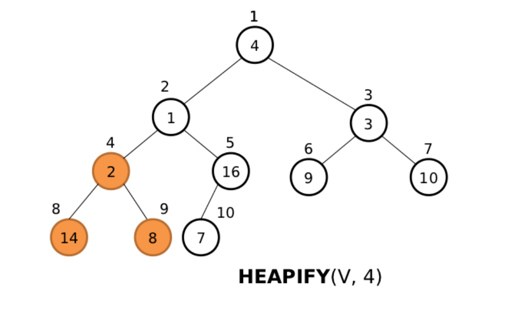
```
void BUILD_HEAP (V){
    for (p = floor(length(V)/2);p>=1; p--){
        HEAPIFY(V, p)
    }
}
```

Código 7: Visão geral do procedimento *BUILD_HEAP*

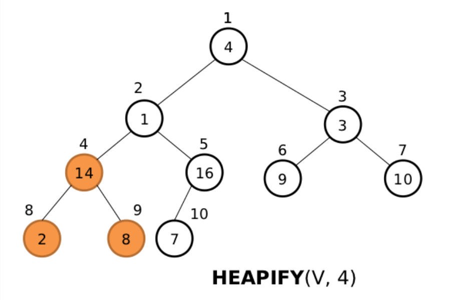
Heapsort: *BUILD_HEAP*



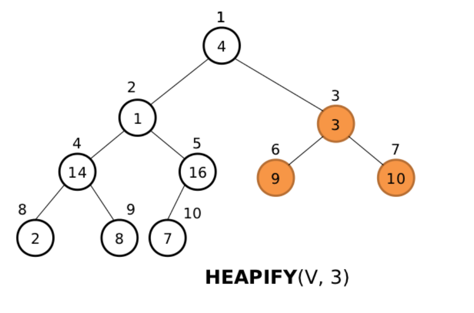
Heapsort: *BUILD_HEAP*



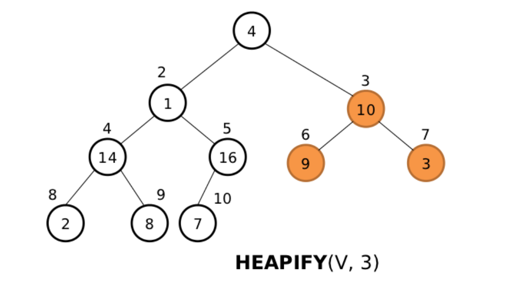
Heapsort: *BUILD_HEAP*



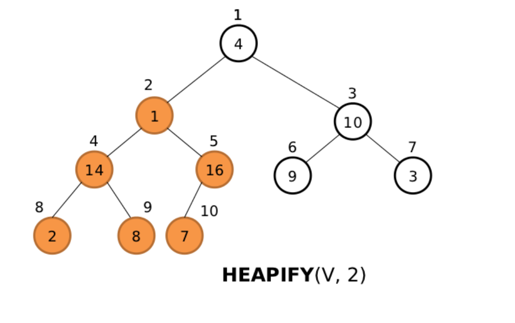
Heapsort: *BUILD_HEAP*



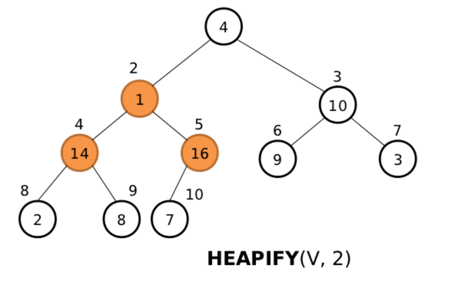
Heapsort: *BUILD_HEAP*



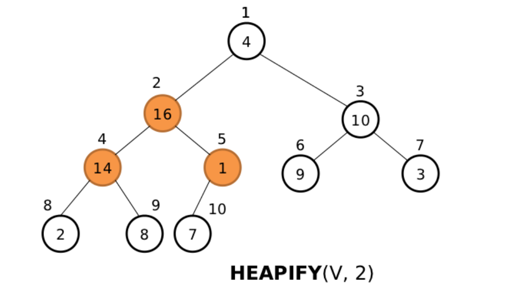
Heapsort: *BUILD_HEAP*



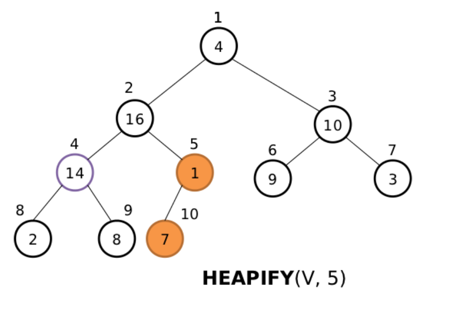
Heapsort: *BUILD_HEAP*



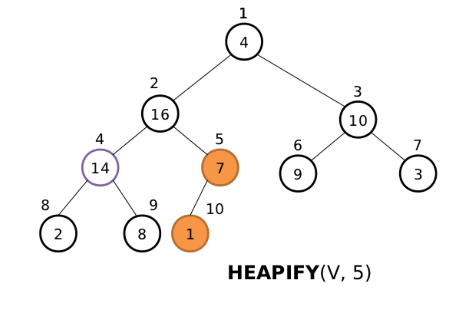
Heapsort: *BUILD_HEAP*



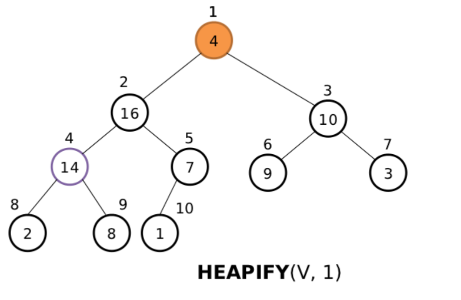
Heapsort: *BUILD_HEAP*



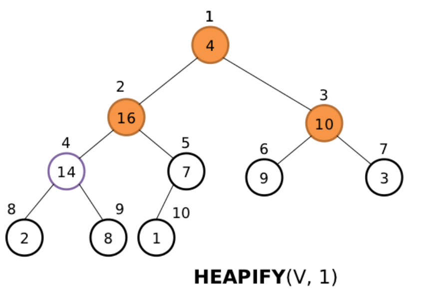
Heapsort: *BUILD_HEAP*



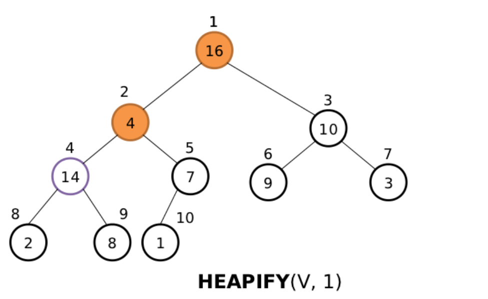
Heapsort: *BUILD_HEAP*



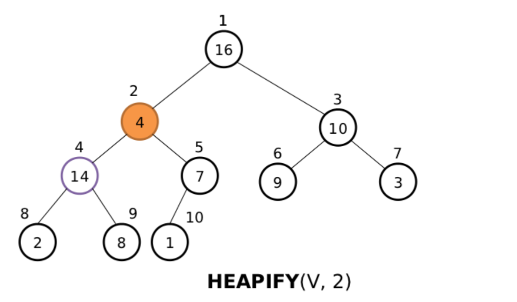
Heapsort: *BUILD_HEAP*



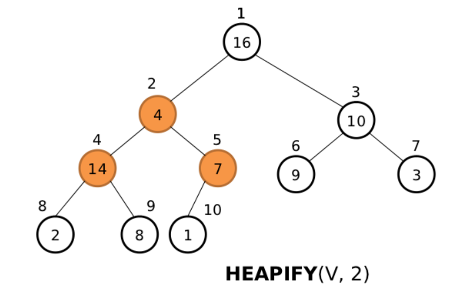
Heapsort: *BUILD_HEAP*



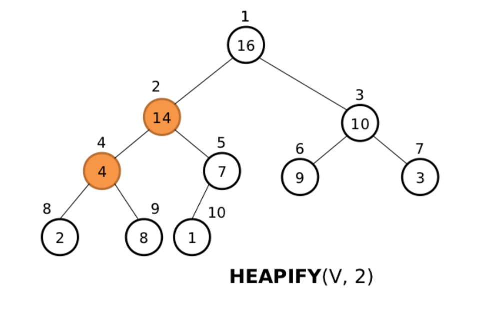
Heapsort: *BUILD_HEAP*



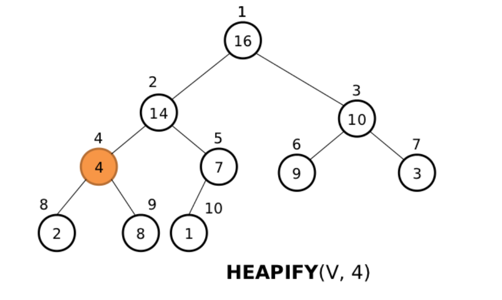
Heapsort: *BUILD_HEAP*

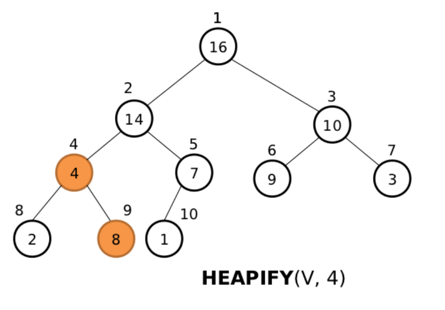


Heapsort: *BUILD_HEAP*

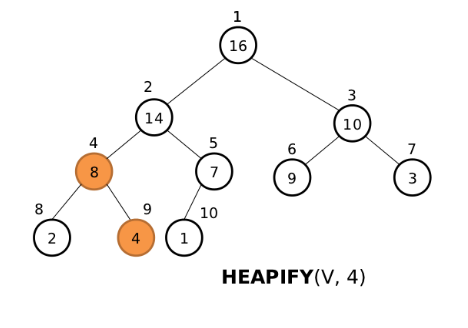


Heapsort: *BUILD_HEAP*

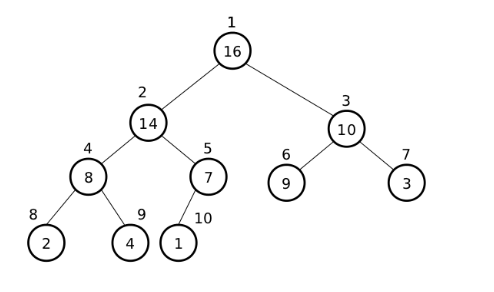



1
16
2
14
3
10
4
4
5
7
6
9
7
3
8
2
9
8
10
1
HEAPIFY(V, 4)

Heapsort: *BUILD_HEAP*



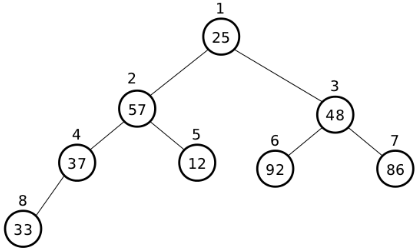
Heapsort: *BUILD_HEAP*



```
void HEAPSORT(V){
    BUILD_HEAP(V);
    for (p = length(V); p>=2; p--){
        troca (V[1], V[p]);
        tamanho_de_V = tamanho_de_V - 1;
        HEAPIFY(V, 1);
    }
}
```

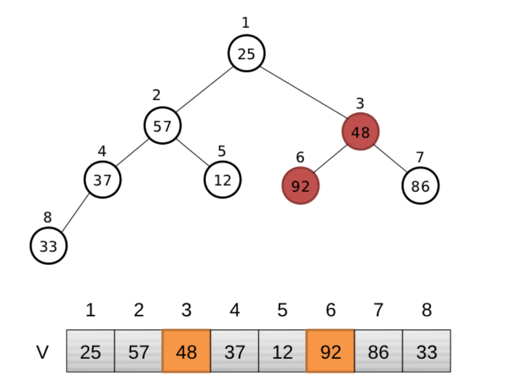
Código 8: Visão geral do Heapsort

Heapsort: Exemplo

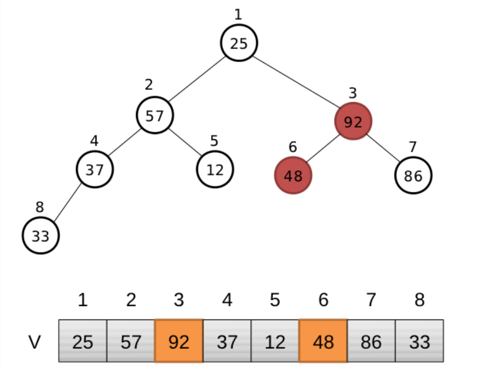


	1	2	3	4	5	6	7	8
V	25	57	48	37	12	92	86	33

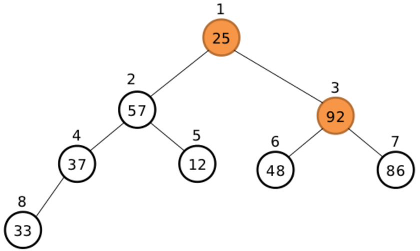
Heapsort: Exemplo



Heapsort: Exemplo

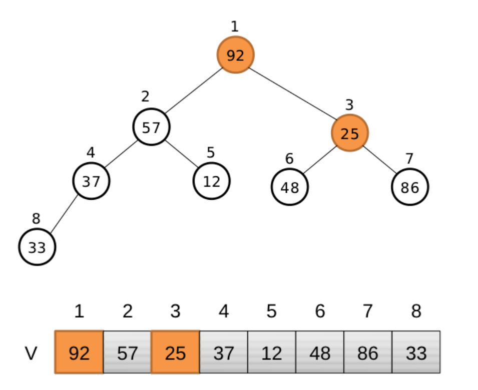


Heapsort: Exemplo

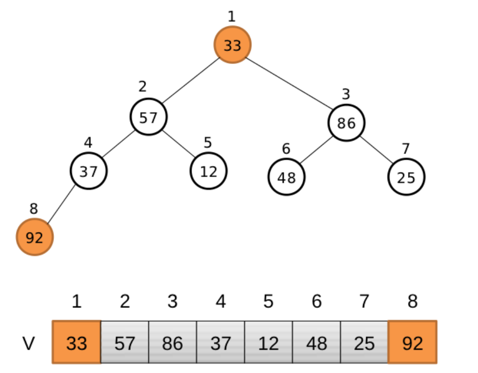


	1	2	3	4	5	6	7	8
V	25	57	92	37	12	48	86	33

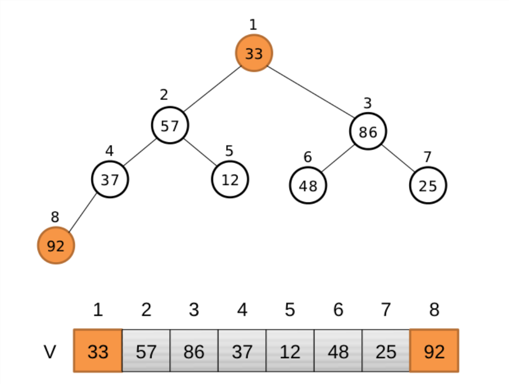
Heapsort: Exemplo



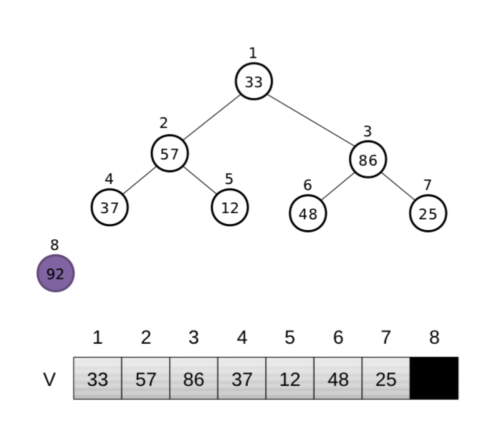
Heapsort: Exemplo



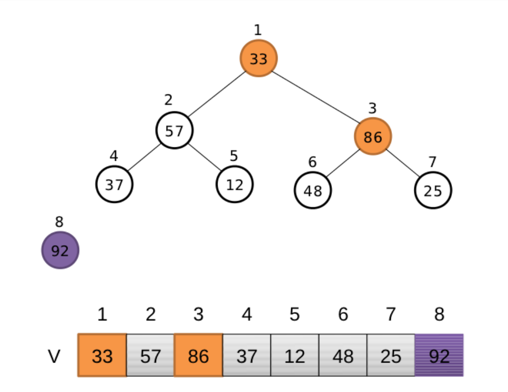
Heapsort: Exemplo



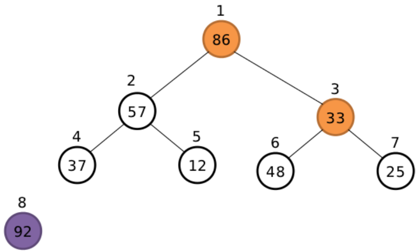
Heapsort: Exemplo



Heapsort: Exemplo

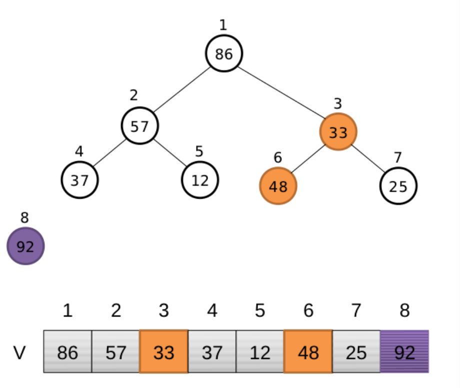


Heapsort: Exemplo

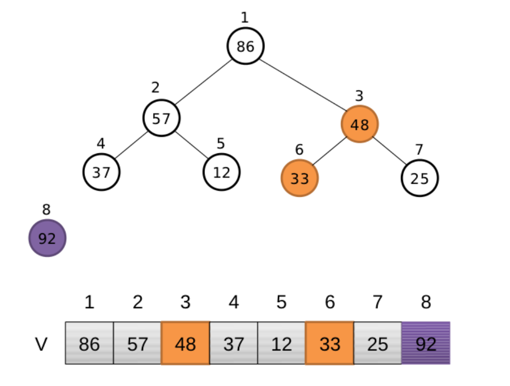


1	2	3	4	5	6	7	8
86	57	33	37	12	48	25	92

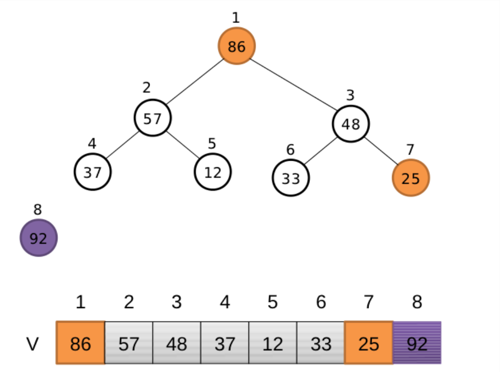
Heapsort: Exemplo



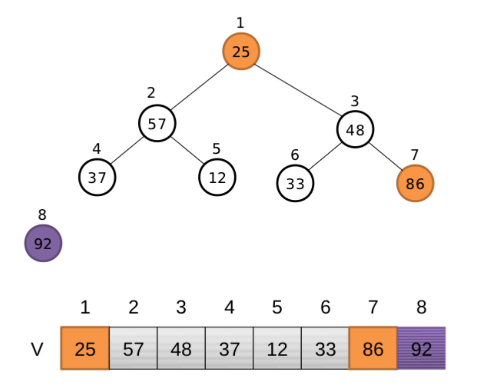
Heapsort: Exemplo



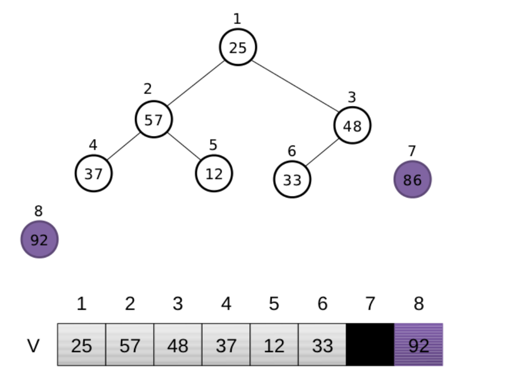
Heapsort: Exemplo



Heapsort: Exemplo



Heapsort: Exemplo

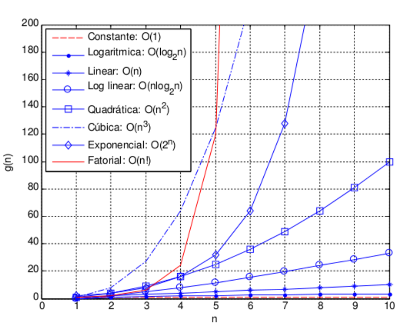


Complexidade dos Algoritmos

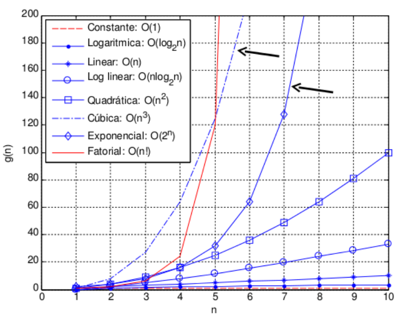
Algoritmo	Pior caso	Caso médio
Bubblesort	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n^2)$	$O(n^2)$
QuickSort	$O(n^2)$	$O(n \log_2 n)$
MergeSort	$O(n \log_2 n)$	$O(n \log_2 n)$
HeapSort	$O(n \log_2 n)$	$O(n \log_2 n)$

Tabela 1: Complexidade dos algoritmos de ordenação

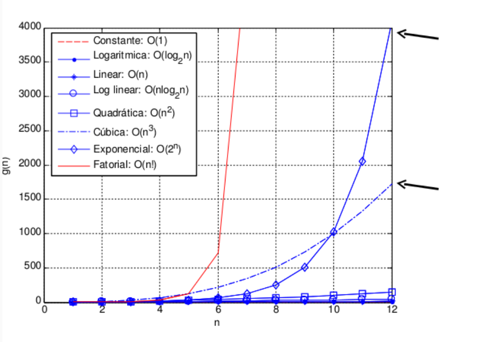
Complexidade dos Algoritmos



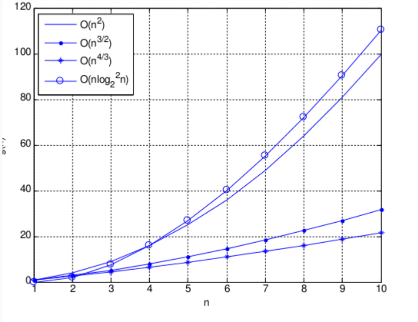
Complexidade dos Algoritmos



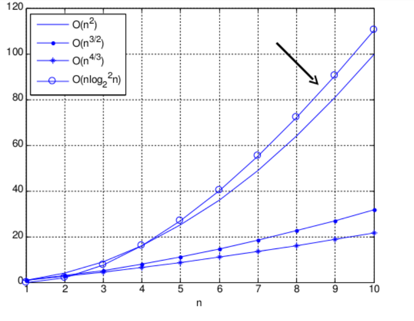
Complexidade dos Algoritmos



Complexidade dos Algoritmos



Complexidade dos Algoritmos



Busca Binária

Busca em estruturas lineares

- Um problema bem conhecido quando se manipula listas/ arrays / vetores é encontrar um elemento com um determinado valor.
- A forma trivial é percorrer da posição inicial até a final de todos os elementos do conjunto, até achar o valor desejado $\longrightarrow$ Busca Linear ou Sequencial.

Busca sequencial por um valor inteiro

```c
//Procura value em um array de inteiros - busca linear.
int buscalinear(int *v, int n, int value)
{
  int i;
  for(i = 0; i < n; i++){
    if (v[i] == value)
      return i;
  }
  return -1; //nao achou
}
```

Busca sequencial

Quanto tempo a busca sequencial demora para executar? Em outras palavras, quantas vezes a comparação `v[i] == value` é executada?

- Caso valor não esteja presente no vetor, n vezes.
- Caso valor esteja presente no vetor:
    - 1 vez no melhor caso (valor está na primeira posição).
    - n vezes no pior caso (valor está na última posição).
    - $\frac{n}{2}$ vezes no caso médio.

Em suma, a busca linear tem complexidade $O(n)$, no pior caso e no caso médio.

Busca Binária

Supondo agora que o conjunto está ordenado. Será que é possível resolver o problema de modo mais eficiente?

- A Busca Binária permite reduzir o número de comparações, no pior caso, de n para $\log_2(n)$, onde n é o número de elementos, desde que estes estejam ordenados.
- Princípio básico: a cada iteração comparamos o valor com o elemento do meio da sequência, dependendo do resultado descartamos uma das metades e a busca continua na metade restante.
  - Se em um determinado momento o conjunto, após sucessivas divisões, tiver tamanho zero, então o elemento não está presente.

Busca binária por um valor inteiro

```
int buscabinaria(int* v, int n, int value)
{
  int inicio, fim, pos;

  inicio = 0;
  fim = n-1;

  while (inicio <= fim)
  {
        pos = (inicio+fim)/2;
        if (value < v[pos])
           fim = pos-1; //value esta na 1a metade
        else
            if (value > v[pos])
               inicio = pos + 1; //value esta na 2a metade
            else
               return pos; //achou: value == v[pos]
  }
  return -1; //nao achou
}
```

Qual dos dois algoritmos é melhor?

- Para $n = 1000$, o algoritmo de busca sequencial irá executar 1000 comparações no pior caso, 500 operações no caso médio.
- Por sua vez, o algoritmo de busca binária irá executar 10 comparações (aproximadamente) no pior caso, para o mesmo n.
- O algoritmo de busca binária supõe que o conjunto está ordenado, o que também tem um custo, como já vimos anteriormente.
- Se pretendermos fazer muitas buscas em um dado conjunto linear, considerar uma pré-ordenação poderá valer a pena.