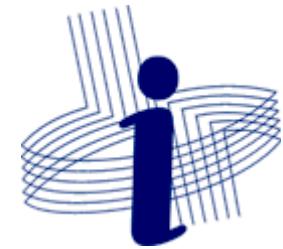


Universidade Federal de Viçosa
Departamento de Informática
Centro de Ciências Exatas e Tecnológicas



INF 112
Programação 2
Aula “9”
Ordenação - 3

QuickSort

- Um dos algoritmos mais famosos de ordenação é o QuickSort. Ele (e algumas variações dele) é muito utilizado na prática, sendo, em geral, bastante eficiente.
- Foi proposto por Hoare, em 1960.
- Assim como o MergeSort, ele também é baseado na ideia de “dividir para conquistar”.
- A divisão é feita com base no valor dos elementos do array.



QuickSort

- A divisão é feita com base no valor dos elementos do array.
 - Para dividir o array, escolhe-se um elemento, denominado pivô , e rearanja o arranjo da seguinte forma que o pivô fique na posição correta $A[s]$:
 - $A[0] \dots A[s-1]$, $A[s]$, $A[s+1] \dots A[n-1]$
 $\leq A[s]$ $\geq A[s]$
 - Após realizar a reorganização, os dois subarranjos restantes são ordenados de forma recursiva.



QuickSort

- Ideia em alto nível do algoritmo:
 - Escolha um elemento K para ser o pivô.
 - Pegue todos os elementos menores ou iguais a K e coloque do lado esquerdo do arranjo. Pegue os maiores ou iguais e coloque do lado direito. Coloque K na posição entre os menores ou iguais e os maiores ou iguais.
 - Ordene os dois subarranjos de forma recursiva.
- A seguir, temos um exemplo de implementação do algoritmo...



QUICKSORT(A, p, r)

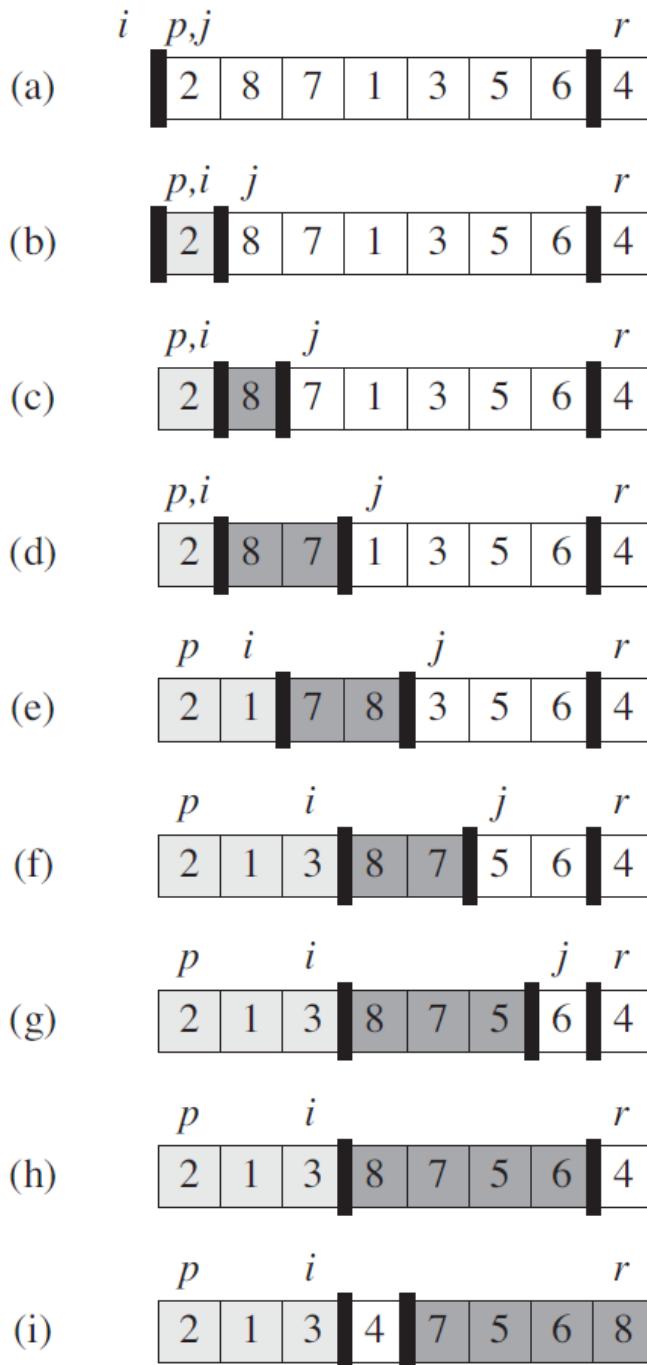
```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 
```

PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

Adaptado do livro do Cormen, terceira edição.





- $A[r]$ é o pivot;
- Os elementos menores que o pivot estão em cinza claro;
- Os elementos maiores que o pivot estão em cinza escuro;
- Os elementos em branco ainda não estão em nenhuma das partições;

Adaptado do livro do Cormen, terceira edição.



```
int partition (int *arr, int low, int high){  
    int pivot = arr[high]; // pivot  
    cout << "\npivot = " << pivot << "\n";  
    int i = (low - 1); // Index of smaller element  
  
    for (int j = low; j <= high- 1; j++) {  
        if (arr[j] <= pivot){  
            i++; // increment index of smaller element  
            swap(arr[i], arr[j]);  
        }  
    }  
    swap(arr[i + 1], arr[high]);  
    return (i + 1);  
}
```

```
void quickSort(int *arr, int low, int high){  
    if (low < high){  
        int pi = partition(arr, low, high);  
        quickSort(arr, low, pi - 1);  
        quickSort(arr, pi + 1, high);  
    }  
}
```

QuickSort

- Exemplo de execução: rastreie para o arranjo [7,6,9,3,1,10].
- Qual elemento essa versão do QuickSort utiliza como pivô?
- Isso poderia gerar algum problema?



QuickSort

- Pode-se mostrar que, no melhor caso, o QuickSort consegue dividir o vetor em dois pedaços aproximadamente iguais e, com isso, ter complexidade $O(n \log(n))$.
- No caso médio, tal complexidade também é atingida.
- Porém, no pior caso, a complexidade dele seria $O(n^2)$. Por que?



QuickSort

- Pode-se mostrar que, no melhor caso, o QuickSort consegue dividir o vetor em dois pedaços aproximadamente iguais e, com isso, ter complexidade $O(n \log(n))$.
- No caso médio, tal complexidade também é atingida.
- Porém, no pior caso, a complexidade dele seria $O(n^2)$. Por que? R: imagine um arranjo já ordenado seja dado ao nosso algoritmo. Em cada chamada recursiva o QuickSort iria dividir o arranjo em um subarranjo com 0 elementos e outro com um elemento a menos.
- Como resolver tal problema?
 - Ideia 1: utilizar a mediana como pivô.



QuickSort

- Pode-se mostrar que, no melhor caso, o QuickSort consegue dividir o vetor em dois pedaços aproximadamente iguais e, com isso, ter complexidade $O(n \log(n))$.
- No caso médio, tal complexidade também é atingida.
- Porém, no pior caso, a complexidade dele seria $O(n^2)$. Por que? R: imagine um arranjo já ordenado seja dado ao nosso algoritmo.. em cada chamada recursiva o QuickSort iria dividir o arranjo em um subarranjo com 0 elementos e outro com um elemento a menos.
- Como resolver tal problema?
 - Ideia 1: utilizar a mediana como pivô: seria perfeito!, mas como calcularíamos a mediana?



QuickSort

- Pode-se mostrar que, no melhor caso, o QuickSort consegue dividir o vetor em dois pedaços aproximadamente iguais e, com isso, ter complexidade $O(n \log(n))$.
- No caso médio, tal complexidade também é atingida. Pode-se provar matematicamente que, no caso médio (arranjos com números aleatórios), o algoritmo realiza apenas 38% mais comparações do que melhor caso.
- Assim, em média, o algoritmo é um dos métodos mais rápidos existentes (justificando o nome).



QuickSort

- Porém, no pior caso, a complexidade dele seria $O(n^2)$. Por que? R: imagine um arranjo já ordenado seja dado ao nosso algoritmo.. em cada chamada recursiva o QuickSort iria dividir o arranjo em um subarranjo com 0 elementos e outro com um elemento a menos.
- Como resolver tal problema?
 - Ideia 1: utilizar a mediana como pivô: seria perfeito!, mas como calcularíamos a mediana?
 - Ideia 2: pegar o elemento do meio do arranjo como pivô? funcionaria bem para arranjos já ordenados. Teria algum problema?
 - Ideia 3: sortear 1 elemento e usar como pivô?
 - Ideia 4: sortear 3 elementos, tirar a mediana deles e usar como pivô?



QuickSort

- Ideia 3: sortear 1 elemento e usar como pivô.
- Ideia 4: sortear 3 elementos, tirar a mediana deles e usar como pivô.
 - ideias muito utilizada na prática! no pior caso, o algoritmo continua $O(n^2)$, mas o pior caso é MUITO difícil de acontecer! na prática o algoritmo funciona de forma excelente!
- Existem várias variações do QuickSort... algumas eliminam o uso de recursividade, outras tentam ser mais eficientes na escolha do pivô, outras implementam algoritmos híbridos (ex: Introsort = QuickSort + HeapSort).



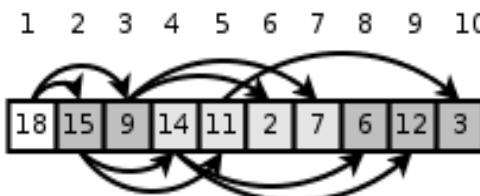
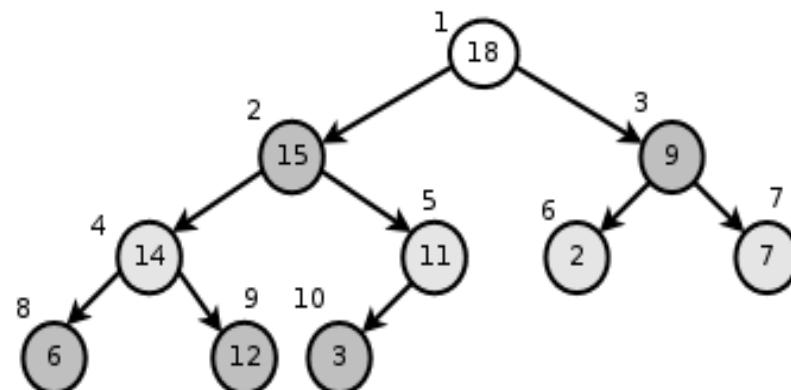
HeapSort

- Um heap é uma estrutura de dados parcialmente ordenada e muito utilizada na implementação de filas de prioridade.
- Mais especificamente, um heap armazena elementos contendo “prioridades” e provê as seguintes funcionalidades:
 - Retornar o elemento com maior prioridade.
 - Deletar o elemento com maior prioridade.
 - Adicionar um novo elemento ao heap.



HeapSort

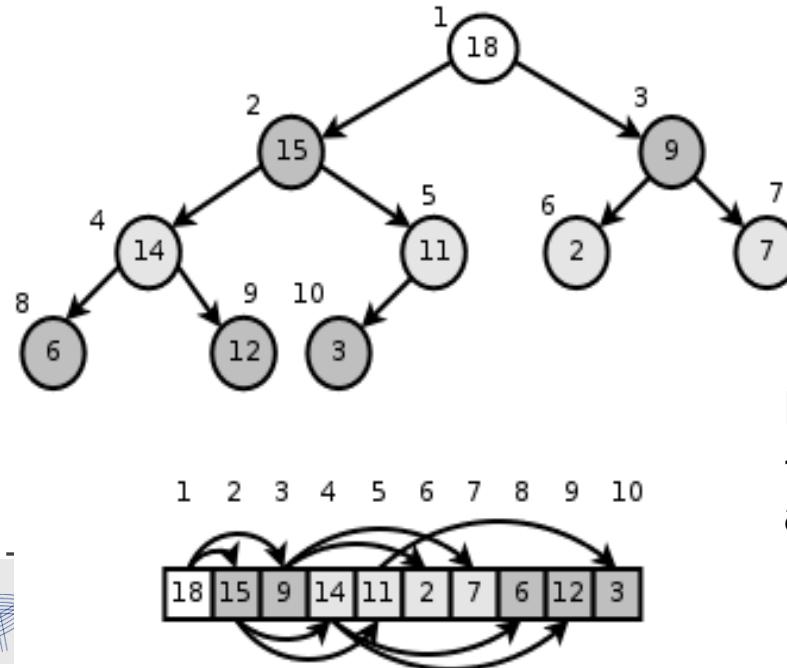
- Um heap pode ser definido de duas formas:
- Árvore binária (estrutura de dados muito usada na computação – será vista em outras disciplinas):
 - Contendo todos os níveis completos, com exceção do último que pode ter as folhas mais à direita faltando.
 - O nodo de um nível é maior do que todos os seus filhos.



Fonte: http://cs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/

HeapSort

- Um heap pode ser definido de duas formas:
- **Array** (vamos supor que o primeiro índice é o 1):
 - Array $H[1..n]$ onde os filhos do elemento i são os elementos $2i$ e $2i+1$ (se existirem). Assim, $H[i] \geq H[2i]$ e $H[2i+1]$.
 - Vamos focar nessa representação.

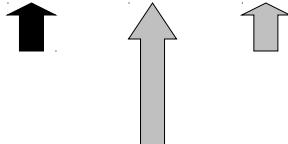


Fonte: http://cs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/

HeapSort

- Vamos trabalhar com heaps representados por array.
- Uma função importante para trabalhar com heap é a função “peneira”. A ideia dela é receber um vetor $v[1..m]$, um índice p e, então, fazer $v[p]$ “descer” para a posição correta no heap v .
- Exemplo com $p = 1$: (suponha que apenas o elemento 1 esteja fora da posição no heap – note que os filhos já são sub-heaps!)
- Ideia:
 - se o $v[p] \geq v[2p]$ e $v[2p+1]$, então não precisa fazer nada...
 - senão, trocamos $v[p]$ pelo maior dos filhos e repetimos o processo da nova posição de p .

1	2	3	4	5	6	7	8	9	10	11	12	13	14
85	99	98	97	96	95	94	93	92	91	90	89	87	86



HeapSort

- Vamos trabalhar com heaps representados por array.
- Uma função importante para trabalhar com heap é a função “peneira”. A ideia dela é receber um vetor $v[1..m]$, um índice p e, então, fazer $v[p]$ “descer” para a posição correta no heap v .
- Exemplo com $p = 1$: (suponha que apenas o elemento 1 esteja fora da posição no heap – note que os filhos já são sub-heaps!)
- Ideia:
 - se o $v[p] \geq v[2p]$ e $v[2p+1]$, então não precisa fazer nada...
 - senão, trocamos $v[p]$ pelo maior dos filhos e repetimos o processo da nova posição de p .

1	2	3	4	5	6	7	8	9	10	11	12	13	14
99	85	98	97	96	95	94	93	92	91	90	89	87	86

↑ ↑ ↑

HeapSort

- Vamos trabalhar com heaps representados por array.
- Uma função importante para trabalhar com heap é a função “peneira”. A ideia dela é receber um vetor $v[1..m]$, um índice p e, então, fazer $v[p]$ “descer” para a posição correta no heap v .
- Exemplo com $p = 1$: (suponha que apenas o elemento 1 esteja fora da posição no heap – note que os filhos já são sub-heaps!)
- Ideia:
 - se o $v[p] \geq v[2p]$ e $v[2p+1]$, então não precisa fazer nada...
 - senão, trocamos $v[p]$ pelo maior dos filhos e repetimos o processo da nova posição de p .

1	2	3	4	5	6	7	8	9	10	11	12	13	14
99	97	98	85	96	95	94	93	92	91	90	89	87	86



HeapSort

- Vamos trabalhar com heaps representados por array.
- Uma função importante para trabalhar com heap é a função “peneira”. A ideia dela é receber um vetor $v[1..m]$, um índice p e, então, fazer $v[p]$ “descer” para a posição correta no heap v .
- Exemplo com $p = 1$: (suponha que apenas o elemento 1 esteja fora da posição no heap – note que os filhos já são sub-heaps!)
- Ideia:
 - se o $v[p] \geq v[2p]$ e $v[2p+1]$, então não precisa fazer nada...
 - senão, trocamos $v[p]$ pelo maior dos filhos e repetimos o processo da nova posição de p .

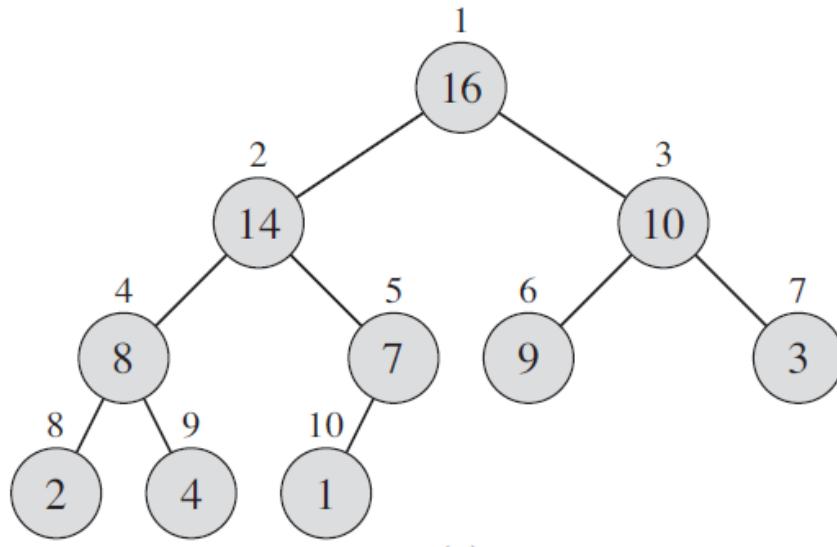
1	2	3	4	5	6	7	8	9	10	11	12	13	14
99	97	98	93	96	95	94	85	92	91	90	89	87	86



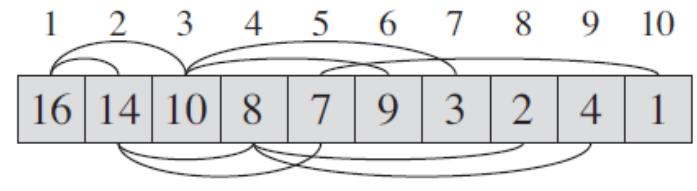
Podemos parar, pois já passamos do meio do heap! (por que podemos fazer isso?)



HeapSort



(a)



(b)

Adaptado do livro do Cormen, terceira edição.

Por que a partir da posição $(m/2)+1$, nesse caso a posição 6, temos apenas folhas!



HeapSort

Pseudocódigo da função **peneira**, também chamada de **heapify**.

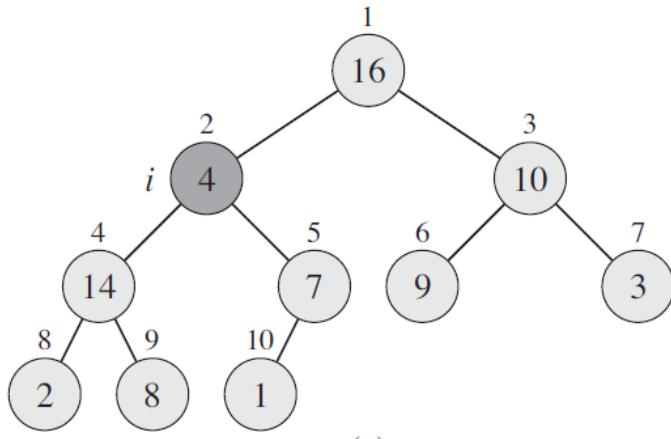
MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

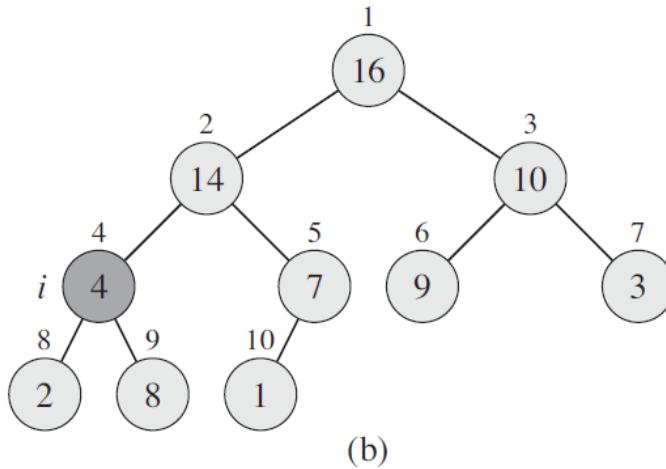
Adaptado do livro do Cormen, terceira edição.



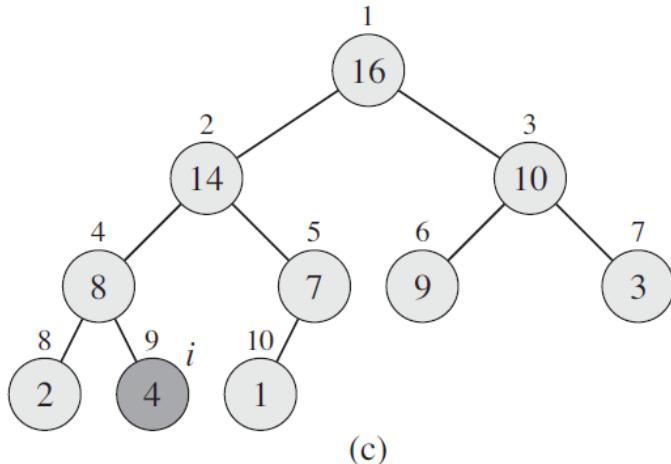
HeapSort



(a)



(b)



(c)

- A cada iteração o maior entre $A[i]$, $A[\text{left}(i)]$ e $A[\text{right}(i)]$ é determinado.
- Se $A[i]$ é o maior, não precisa fazer nada.
- Senão, troca $A[i]$ pelo maior.
- A subárvore cuja raiz é o maior, pode precisar ser rearranjada.

Adaptado do livro do Cormen, terceira edição.



```
/*
Rearranja o vetor v[1...m] de modo que
o subvetor cuja raiz eh p seja um heap.
Supõe que os filhos de p no vetor
já satisfaçam as condições de heap
*/
void peneira(int *v, int p, int m) {
    int maiorFilho = 2*p;
    while(maiorFilho <= m) {
        //Se o heap tem outro filho (ou seja, se p
        //não for o "fim" do heap) e tal filho
        //tem maior prioridade, escolhemos os outro
        //como maior filho
        if (maiorFilho < m && v[maiorFilho+1] > v[maiorFilho])
            maiorFilho++;
        //Ja temos um heap!
        if (v[p] > v[maiorFilho])
            return;
        //Vamos trocar p por seu maior filho e continuar o algoritmo
        swap(v[p],v[maiorFilho]);
        p = maiorFilho;
        maiorFilho *= 2;
    }
}
```

HeapSort

- Entenda o funcionamento da função definida no slide anterior.
- Rastreie a execução para o exemplo (com $p = 1$):

1	2	3	4	5	6	7	8	9	10	11	12	13	14
85	99	98	97	96	95	94	93	92	91	90	89	87	86

- A função peneira é muito eficiente. Ela pode ser executada em tempo $O(\log(m))$.
- Como poderíamos utilizar a função peneira para construir um heap a partir de um vetor qualquer?



HeapSort

- Entenda o funcionamento da função definida no slide anterior.
- Rastreie a execução para o exemplo (com $p = 1$):

1	2	3	4	5	6	7	8	9	10	11	12	13	14
85	99	98	97	96	95	94	93	92	91	90	89	87	86

- A função peneira é muito eficiente. Ela pode ser executada em tempo $O(\log(m))$.
- Como poderíamos utilizar a função peneira para construir um heap a partir de um vetor qualquer?
- R: Note que um vetor qualquer respeita as condições de heap depois do meio do vetor. Poderíamos, então, fazer um laço que, a cada passo, insere um novo elemento no heap redefinindo seus limites e utilizando a função peneira.



HeapSort

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

Adaptado do livro do Cormen, terceira edição.

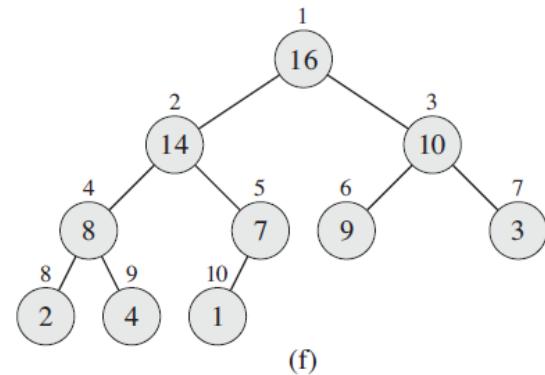
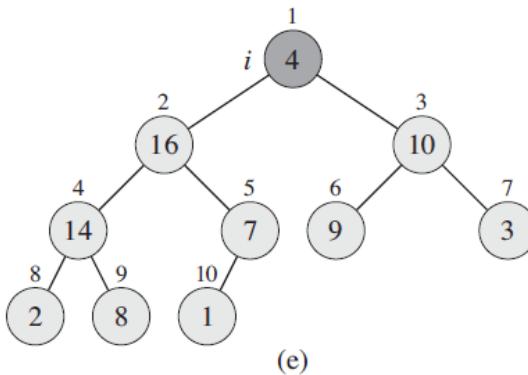
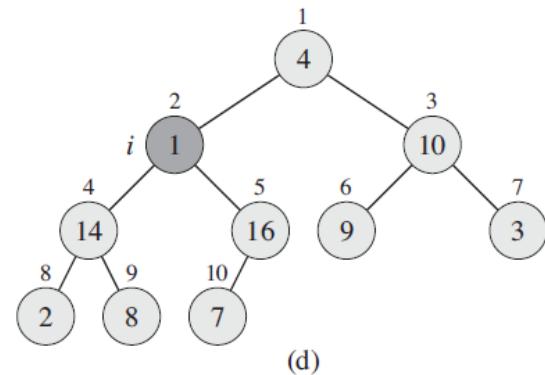
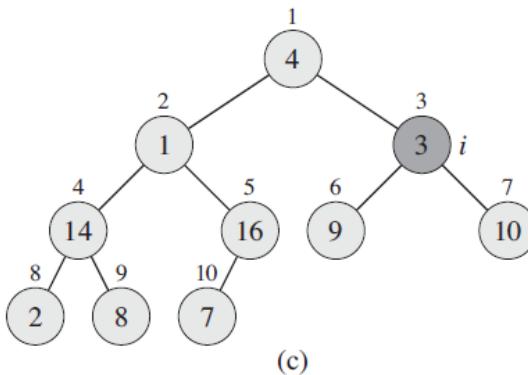
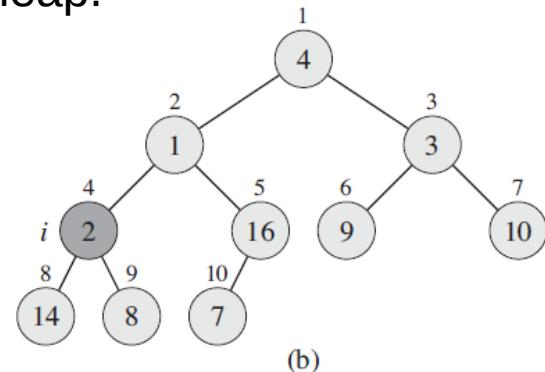
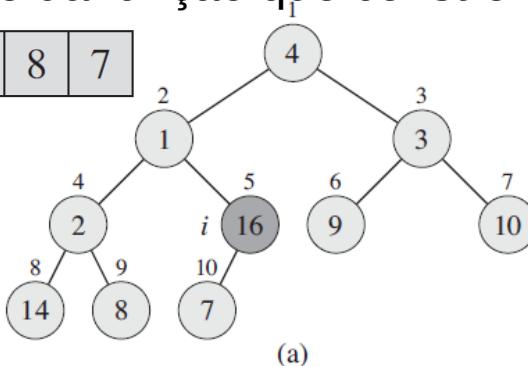


Exemplo do funcionamento da função que constrói o heap.

A	4	1	3	2	16	9	10	14	8	7
---	---	---	---	---	----	---	----	----	---	---



Começa do meio do vetor.



Adaptado do livro do Cormen,
terceira edição.



HeapSort

- Qual a relação de tudo isso com ordenação?

HeapSort

- Qual a relação de tudo isso com ordenação?
- Podemos utilizar a função “peneira” para criar um algoritmo muito eficiente.
- Ideia: dado um arranjo $v[1..n]$
 1. Utilize a função peneira para construir um heap em $v[1..n]$
 2. O maior elemento do vetor estará, necessariamente, na primeira posição. Podemos trocar o elemento da posição 1 pelo elemento da posição n .
 3. Podemos, agora, diminuir o tamanho do arranjo (conceitualmente) para $n-1$, reestabelecer o heap e continuar a execução a partir do passo 2 até que o heap fique vazio.

HeapSort

- Implementação:

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i = A.length$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

- Observações:

- A complexidade é $O(n \log n)$ (mesmo no pior caso) – similar ao mergeSort.
- Não é um algoritmo estável.
- É *in-place*?
- Assintoticamente, é melhor do que o QuickSort... Porém, em instâncias práticas o QuickSort, normalmente, ganha em desempenho.



HeapSort

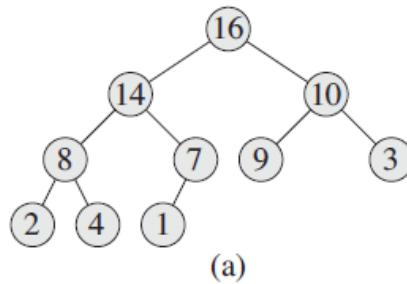
- Implementação:

```
//Cria um heap em v  
for(int p= n/2;p>=1;p--)  
    peneira(v,p,n);  
  
//Ordena v..  
for(int m = n;m>=2;m--) {  
    swap(v[1],v[m]);  
    peneira(v,1,m-1);  
}
```

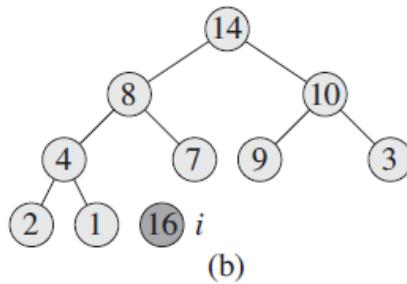
- Observações:

- A complexidade é $O(n \log n)$ (mesmo no pior caso) – similar ao mergeSort.
- Não é um algoritmo estável.
- É *in-place*?
- Assintoticamente, é melhor do que o QuickSort... Porém, em instâncias práticas o QuickSort, normalmente, ganha em performance.

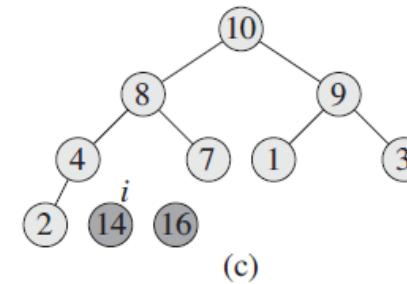




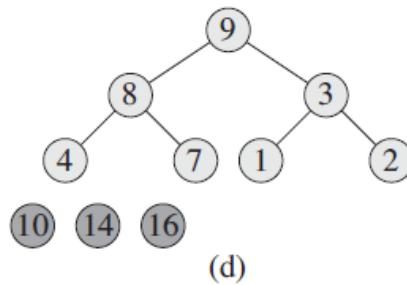
(a)



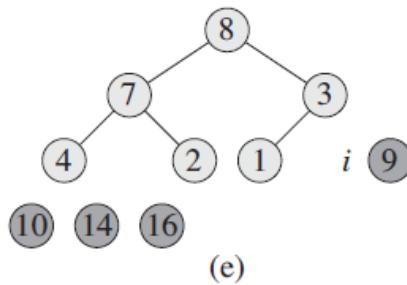
(b)



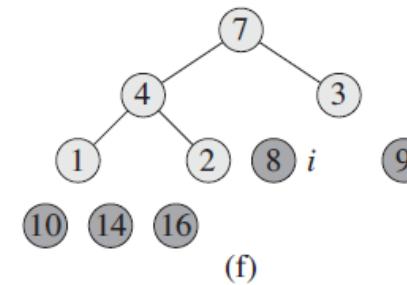
(c)



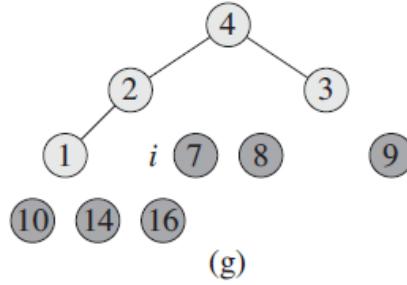
(d)



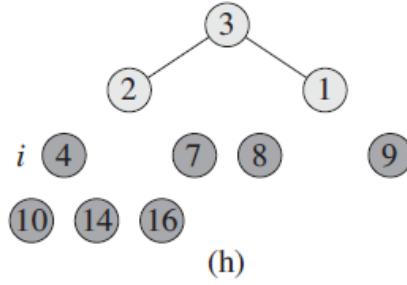
(e)



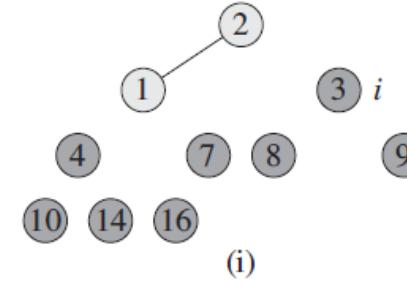
(f)



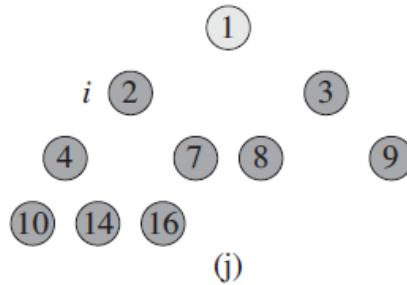
(g)



(h)



(i)



(j)

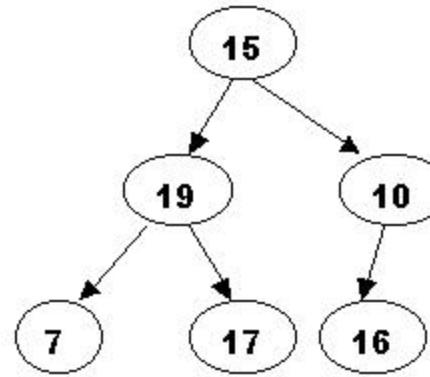
A	1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	---	----	----	----

(k)

HeapSort - Exercício

Esse vetor é um heap?

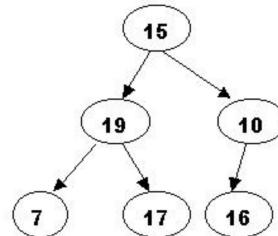
15	19	10	7	17	16
----	----	----	---	----	----



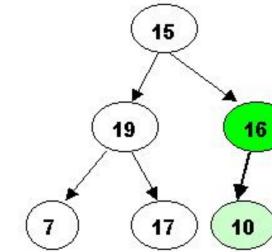
HeapSort

Lembrando que começamos do meio do heap e chamamos a função HEAPFY.

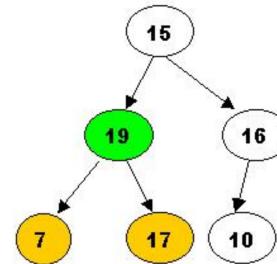
15	19	10	7	17	16
----	----	----	---	----	----



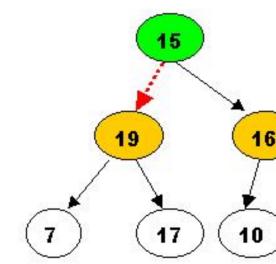
15	19	16	7	17	10
----	----	----	---	----	----



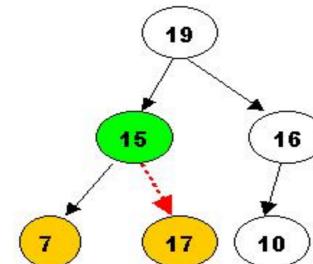
15	19	16	7	17	10
----	----	----	---	----	----



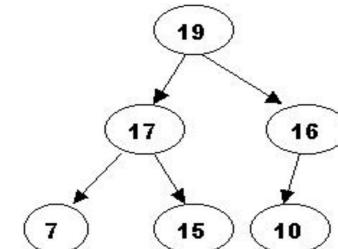
15	19	16	7	17	10
----	----	----	---	----	----



19	15	16	7	17	10
----	----	----	---	----	----



19	17	16	7	15	10
----	----	----	---	----	----



Ordenação com o *qsort*

- Como a ordenação é um problema muito importante, a maioria das linguagens de programação oferece métodos prontos para ordenar dados.
- Na linguagem C (e em C++), a biblioteca *cstdlib* possui a função de ordenação *qsort*.
- O padrão da linguagem não especifica qual algoritmo de ordenação é utilizado para implementar o *qsort* mas, em geral, as implementações dessa função são bastante eficientes.



Ordenação com o *qsort*

- Assinatura da função *qsort*.

```
void qsort ( void * v, size_t num, size_t size, int ( * compar ) ( const void *, const void * ));
```

- *v*: ponteiro para a primeira posição do vetor a ser ordenado.
- *num*: número de elementos a serem ordenados em *v*.
- *size*: número de bytes que cada elemento de *v* ocupa (pode ser obtido utilizando o operador `sizeof`).
- *compar*: ponteiro para função de comparação de dois elementos (do tipo a ser ordenado).
- Note que, como a função de ordenação não sabe como os dados deverão ser comparados (para fazer a ordenação), é necessário passar uma função de comparação como parâmetro para o método.



Ordenação com o *qsort*

- A função *compar* deve receber como argumento dois apontadores para *void* (apontadores genéricos, que podem apontar para qualquer tipo de dado) e, então, retornar:
 - 0 se os elementos apontados forem iguais
 - Um número negativo se o elemento apontado pelo primeiro apontador for menor do que o apontado pelo segundo.
 - Um valor positivo caso contrário.
- Note que, como a função recebe apontadores para *void*, você deverá fazer um *cast* para converte-los para apontadores do tipo correto (ou seja, para o tipo de dados do array que você deseja ordenar).



Ordenação com o *qsort*

- Exemplo de uso da função *qsort*.

```
#include <cstdlib>
#include <ctime>
//Código adaptado de : http://www.cplusplus.com/reference/cstdlib/qsort/

int compare (const void * a, const void * b) {
    double v1 = *((double *)a); //pega o número double apontado por a ..
    double v2 = *((double *)b); //pega o número double apontado por b ..
    return v1-v2;
}

int main () {
    double valores[] = { 40, 10.85 , 100, 90, 20, 25 };
    int n;

    qsort (valores, 6, sizeof(double), compare); // ordena o vetor valores, que contém 6 números do tipo double

    ...
}
```

