Métodos eficientes de ordenação

Heapsort

Alessandro Jean
Igor Neres Trindade

25 de Maio de 2018

Universidade Federal do ABC

Sumário

Introdução

Introdução

- Inventado por J. W. J. Williams em 1964;
- No mesmo ano, Robert W. Floyd publicou uma versão aprimorada que ordena um vetor in-place;
- Uso de uma estrutura de dados chamada *heap*;
- Parecido com o *selection sort*;
- Não é considerado um algoritmo estável.
- Tem complexidade $O(n \log(n))$ no pior caso.

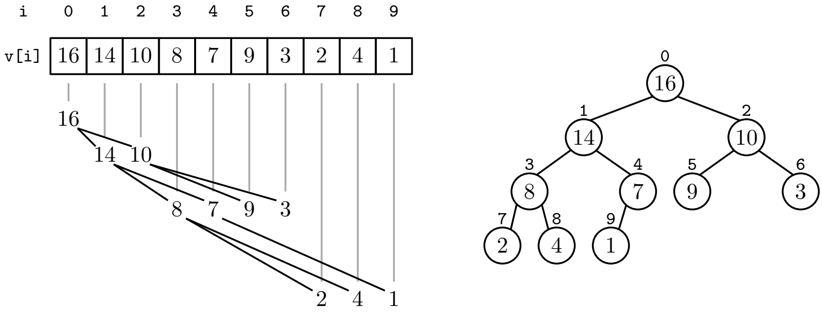
Heap

Definição

- Estrutura que utiliza como base um vetor;
- Pode ser visto como uma árvore binária;
- Cada elemento no vetor corresponde a um nó da árvore;
- Alguns livros implementam uma versão para vetores `[1..n]`. Neste caso, será adotada a versão `[0..n - 1]`.

Definição

- Estrutura que utiliza como base um vetor;
- Pode ser visto como uma árvore binária;
- Cada elemento no vetor corresponde a um nó da árvore;
- Alguns livros implementam uma versão para vetores `[1..n]`. Neste caso, será adotada a versão `[0..n - 1]`.

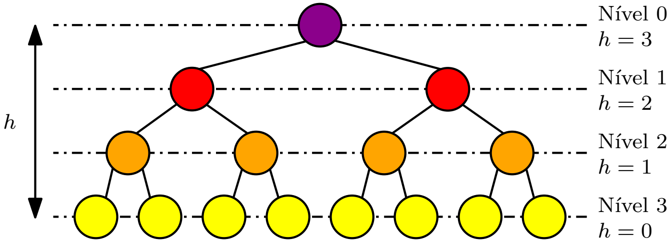


Heap máximo representado por um vetor e uma árvore binária completa.

Definição

Dado uma árvore binária de altura h:

- Uma árvore binária **cheia** é aquela em que cada nó até o nível $h-1$ possui exatamente dois filhos, e os nós no nível h não possuem filhos, ou seja, são as folhas.

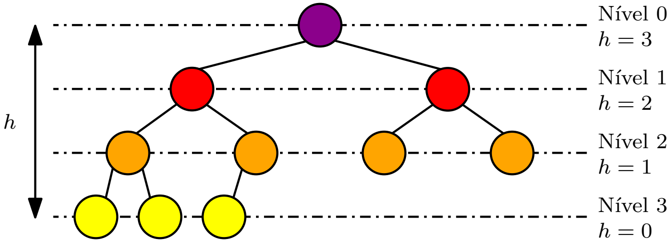


Árvore binária cheia.

Definição

Dado uma árvore binária de altura h:

- Uma árvore binária **cheia** é aquela em que cada nó até o nível $h-1$ possui exatamente dois filhos, e os nós no nível h não possuem filhos, ou seja, são as folhas.
- Uma árvore binária **completa** é aquela que possui todos os níveis preenchidos, exceto possivelmente o último, que é preenchido da esquerda para a direita.



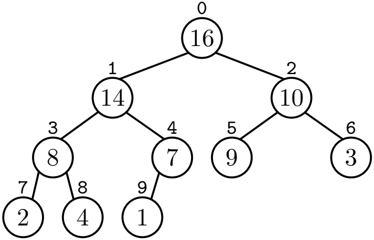
Árvore binária completa.

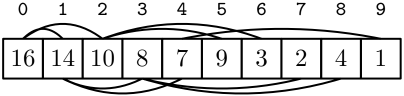
Heap

Relações

Relações

Seja uma árvore binária completa representada por um vetor `v[0..n - 1]`.

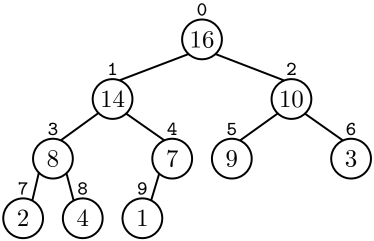


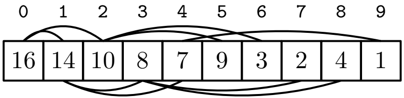


- Para qualquer índice ou nó i:
  - O pai de i é $\left\lfloor \frac{i-1}{2} \right\rfloor$.
  - O filho esquerdo de i é $2i + 1$.
  - O filho direito de i é $2i + 2$.
- O nó `v[0]` não tem pai, sendo o nó raiz;

Relações

Seja uma árvore binária completa representada por um vetor `v[0..n - 1]`.





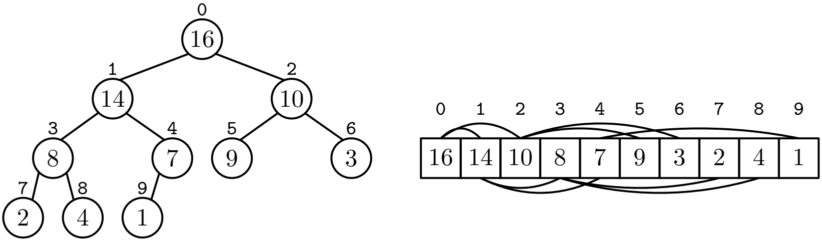

```
int left (int i) { return 2 * i + 1; }

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

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

Relações

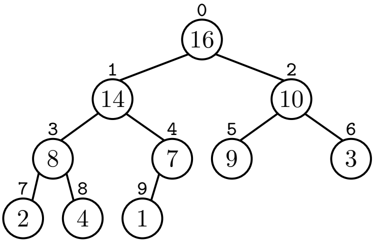
Seja uma árvore binária completa representada por um vetor `v[0..n - 1]`.

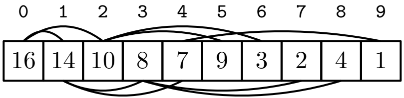


- Um nó i tem filho esquerdo se $2i + 1 < n$;
- Um nó i tem filho direito se $2i + 2 < n$;
- Um nó i é um nó folha se $2i + 1 \geq n$.

Relações

Seja uma árvore binária completa representada por um vetor `v[0..n - 1]`.






```c
int hasLeftChild (int i, int n) { return left(i) < n; }

int hasRightChild (int i, int n) { return right(i) < n; }

int isLeaf (int i, int n) { return left(i) >= n; }
```

Heap

Tipos

Tipos

- Existem dois tipos de *heaps*;
- Cada um satisfaz uma **propriedade de *heap***;

Tipos

- Existem dois tipos de *heaps*;
- Cada um satisfaz uma **propriedade de *heap***;
- ***Heap* máximo:** o maior elemento é a raiz e as sub-árvores possuem valores menores ou iguais que o pai, ou seja, `v[parent(i)] >= v[i]`;

Tipos

- Existem dois tipos de *heaps*;
- Cada um satisfaz uma **propriedade de *heap***;
- ***Heap* máximo:** o maior elemento é a raiz e as sub-árvores possuem valores menores ou iguais que o pai, ou seja, `v[parent(i)] >= v[i]`;
- ***Heap* mínimo:** o menor elemento é a raiz e as sub-árvores possuem valores maiores ou iguais que o pai, ou seja, `v[parent(i)] <= v[i]`;

Heap

Manutenção

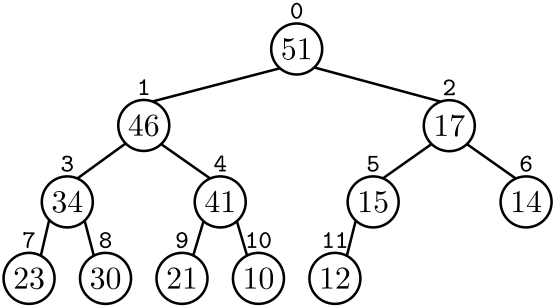
Manutenção

Muitas vezes, um *heap* não vai estar cumprindo sua propriedade de seu tipo, portanto é preciso realizar a manutenção.

$\textsc{Max-Heapify}(v, n, i)$

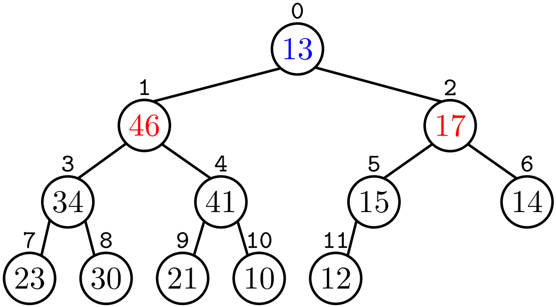
$e \leftarrow$ LEFT(i)
$d \leftarrow$ RIGHT(i)
$maior \leftarrow i$
se *Has-Left-Child*(i, n) **e** $v[maior] > v[i]$ **então**
    $maior \leftarrow e$
se *Has-Right-Child*(i, n) **e** $v[maior] > v[i]$ **então**
    $maior \leftarrow d$
se $maior \neq i$ **então**
    $v[i] \leftrightarrow v[maior]$
    $\textsc{Max-Heapify}(v, n, maior)$

Manutenção do *heap* - Exemplo



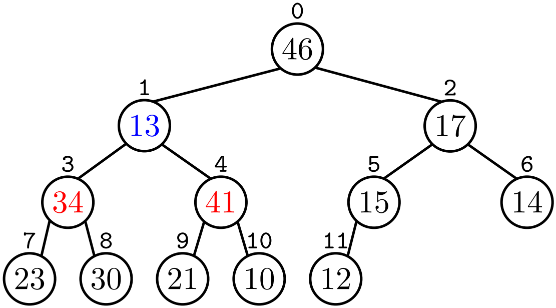
0	1	2	3	4	5	6	7	8	9	10	11
51	46	17	34	41	15	14	23	30	21	10	12

Manutenção do *heap* - Exemplo



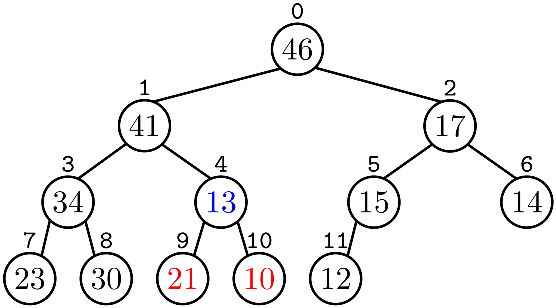
0	1	2	3	4	5	6	7	8	9	10	11
13	46	17	34	41	15	14	23	30	21	10	12

Manutenção do *heap* - Exemplo



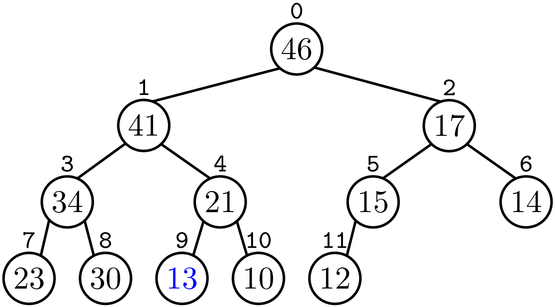
0	1	2	3	4	5	6	7	8	9	10	11
46	13	17	34	41	15	14	23	30	21	10	12

Manutenção do *heap* - Exemplo



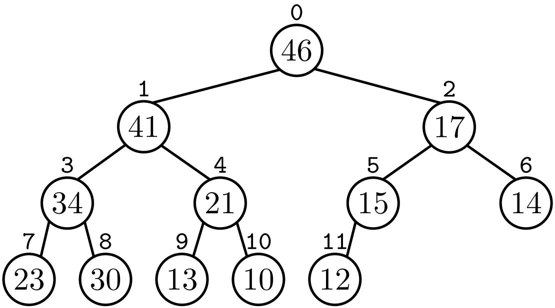
0	1	2	3	4	5	6	7	8	9	10	11
46	41	17	34	13	15	14	23	30	21	10	12

Manutenção do *heap* - Exemplo



0	1	2	3	4	5	6	7	8	9	10	11
46	41	17	34	21	15	14	23	30	13	10	12

Manutenção do *heap* - Exemplo



0	1	2	3	4	5	6	7	8	9	10	11
46	41	17	34	21	15	14	23	30	13	10	12

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja `v` de modo que a raiz `i` seja um *heap* máximo.

```c
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja `v` de modo que a raiz `i` seja um *heap* máximo.

```c
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja v de modo que a raiz i seja um *heap* máximo.

```c
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja v de modo que a raiz `i` seja um *heap* máximo.

```c
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja `v` de modo que a raiz `i` seja um *heap* máximo.

```
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify

Recebe `v[0..n - 1]` e `i >= 0` tal que as sub-árvores da esquerda e direita são *heaps* máximos e rearranja v de modo que a raiz `i` seja um *heap* máximo.

```
int maxHeapify (int * v, int n, int i) {
  int l = left(i), r = right(i), max = i, comp = 1;

  if (hasLeftChild(i, n) && v[l] > v[max])
    max = l;

  if (hasRightChild(i, n) && v[r] > v[max])
    max = r;

  if (max != i) {
    int aux = v[i];
    v[i] = v[max];
    v[max] = aux;

    comp += maxHeapify(v, n, max);
  }

  return comp;
}
```

Max-Heapify - Análise de custo

Dado uma árvore com n nós, e um dado nó i:

- Para corrigir os relacionamentos entre `v[i]`, `v[left(i)]` e `v[right(i)]`, tem-se custo $\Theta(1)$ mais o tempo de execução de `maxHeapify` em uma das sub-árvores de i;
- As sub-árvores têm tamanho máximo igual a $\frac{2n}{3}$ [1];
- O caso pior ocorre quando o último nível da árvore está exatamente metade cheio [2].

Pode-se, então, descrever a recorrência de `maxHeapify` como:

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

[1]Mais detalhes: https://bit.ly/2HnTJg9
[2]Mais detalhes: https://bit.ly/2HNJnp8

Max-Heapify - Caso pior

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

Relembrando o Teorema Mestre: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

1. $f(n) = O(n^{\log_b(a)-\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b(a)})$
2. $f(n) = \Theta(n^{\log_b(a)}) \Rightarrow T(n) = \Theta(n^{\log_b(a)} \log(n))$
3. $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ e existe $c < 1$ tal que para todo n suficientemente grande $af\left(\frac{n}{b}\right) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$

Max-Heapify - Caso pior

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

Relembrando o Teorema Mestre: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

1. $f(n) = O(n^{\log_b(a)-\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b(a)})$
2. $f(n) = \Theta(n^{\log_b(a)}) \Rightarrow T(n) = \Theta(n^{\log_b(a)} \log(n))$
3. $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ e existe $c < 1$ tal que para todo n suficientemente grande $af\left(\frac{n}{b}\right) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$

Neste caso, $a = 1$, $b = \frac{3}{2}$, $n^{\log_b(a)} = n^0 = 1$, $f(n) = \Theta(1)$.

Logo, pelo caso 2, $T(n) = \Theta(\log(n))$.

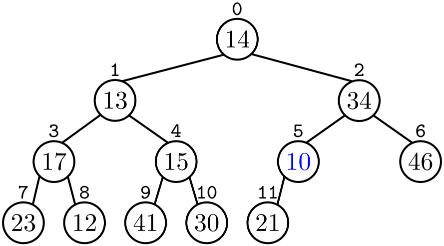
Heap

Construção

Último pai

Dado um *heap* representado por um vetor `v[0..n - 1]`, o último pai, ou seja, o último que não é folha, de v é $\left\lfloor \frac{n}{2} \right\rfloor - 1$.

```
int lastParent (int n) {
  return (n / 2) - 1;
}
```



0	1	2	3	4	5	6	7	8	9	10	11
14	13	34	17	15	10	46	23	12	41	30	21

Neste exemplo, $n = 12$, portanto: $\left\lfloor \frac{12}{2} \right\rfloor - 1 = 6 - 1 = 5$.

Construção de um *heap* máximo

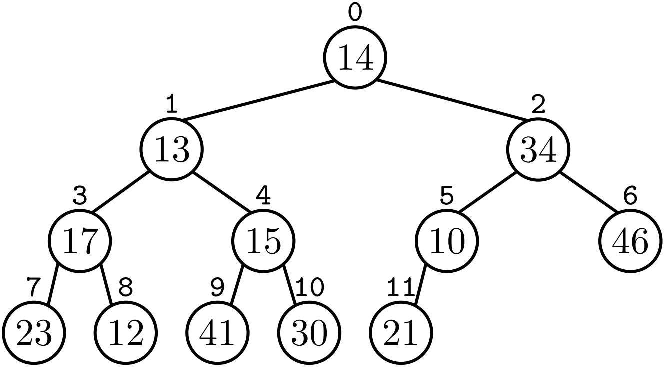
```
BUILD-MAX-HEAP(v, n)
lp ← LAST-PARENT(n)
para i ← lp decrescendo até 0 faça
    MAX-HEAPIFY(v, n, i)
```

É importante começar do último pai até o primeiro, para fazer com que o *heap* fique correto e respeite a propriedade de seu tipo. [1]

Caso isso não seja respeitado, pode ocorrer de certas sub-árvores não serem checadas e não estejam respeitando a propriedade, fazendo com que o primeiro nó não necessariamente seja o maior (ou menor) do *heap*.

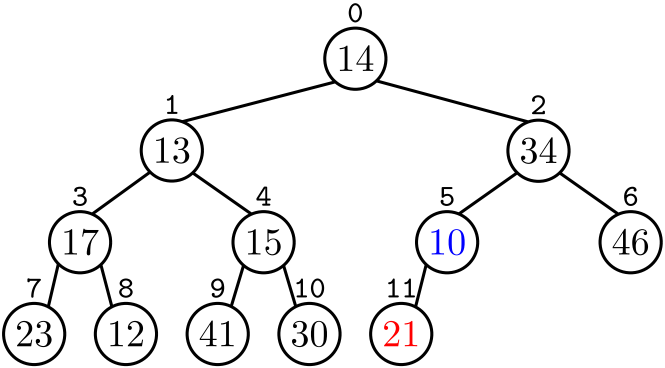
[1]Mais detalhes: https://bit.ly/2qMGbzE

Construção de um *heap* máximo - Exemplo



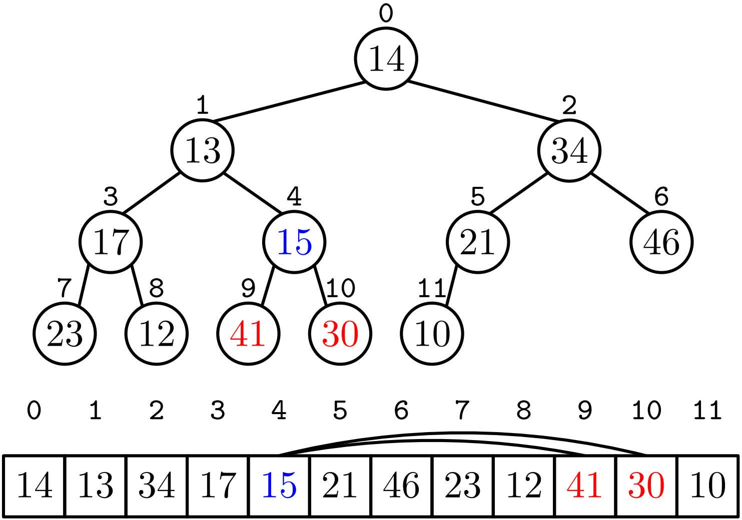
0	1	2	3	4	5	6	7	8	9	10	11
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um *heap* máximo - Exemplo

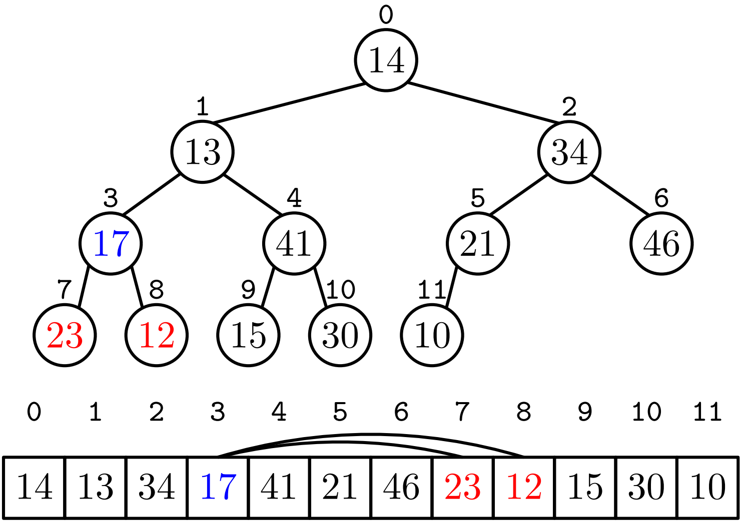


0	1	2	3	4	5	6	7	8	9	10	11
14	13	34	17	15	10	46	23	12	41	30	21

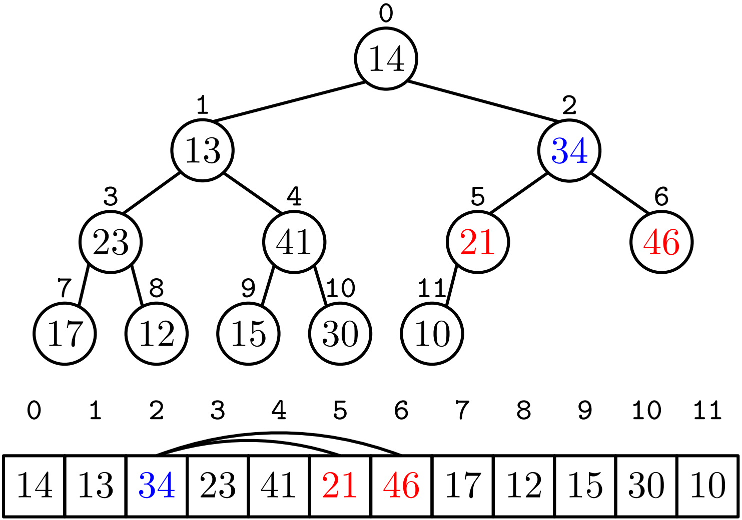
Construção de um *heap* máximo - Exemplo



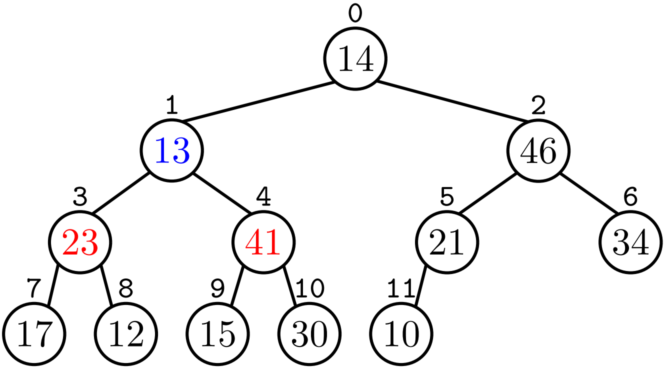
Construção de um *heap* máximo - Exemplo



Construção de um *heap* máximo - Exemplo

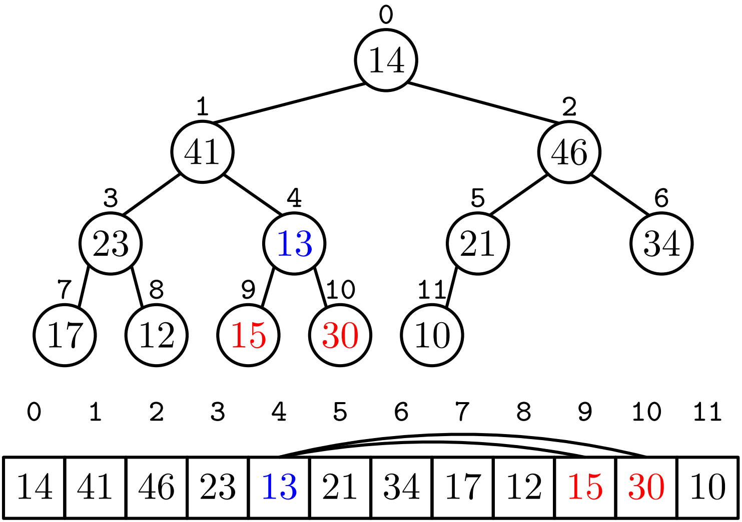


Construção de um *heap* máximo - Exemplo

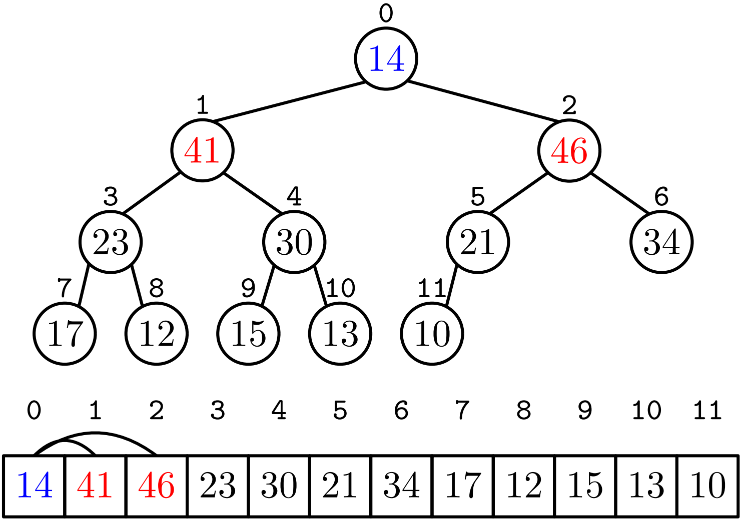


0	1	2	3	4	5	6	7	8	9	10	11
14	13	46	23	41	21	34	17	12	15	30	10

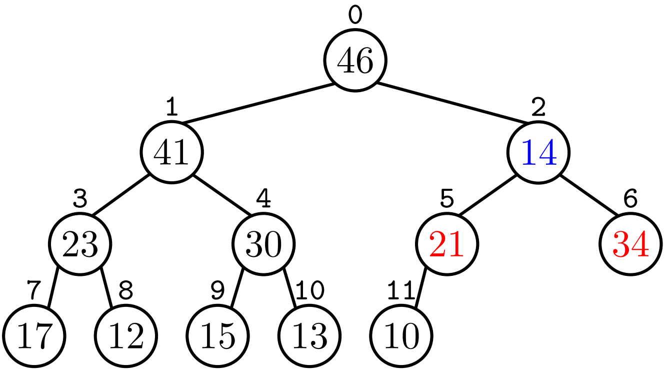
Construção de um *heap* máximo - Exemplo



Construção de um *heap* máximo - Exemplo

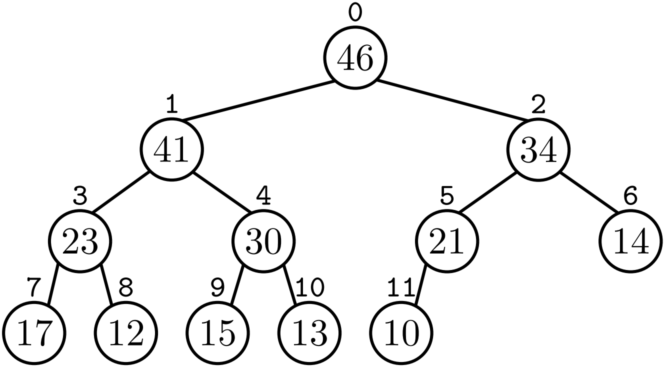


Construção de um *heap* máximo - Exemplo



0	1	2	3	4	5	6	7	8	9	10	11
46	41	14	23	30	21	34	17	12	15	13	10

Construção de um *heap* máximo - Exemplo



0	1	2	3	4	5	6	7	8	9	10	11
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um *heap* máximo - Implementação

```
void buildMaxHeap (int * v, int n) {
  int lp = lastParent(n), comp = 0;

  for (int i = lp; i >= 0; i--)
    comp += maxHeapify(v, n, i);

  return comp;
}
```

Construção de um *heap* máximo - Implementação

```
1 void buildMaxHeap (int * v, int n) {
2   int lp = lastParent(n), comp = 0;
3 
4   for (int i = lp; i >= 0; i--)
5     comp += maxHeapify(v, n, i);
6 
7   return comp;
8 }
```

Análise grosseira do custo:

- Linha 2 tem custo $O(1)$;

Construção de um *heap* máximo - Implementação

```
1 void buildMaxHeap (int * v, int n) {
2   int lp = lastParent(n), comp = 0;
3
4   for (int i = lp; i >= 0; i--)
5     comp += maxHeapify(v, n, i);
6
7   return comp;
8 }
```

Análise grosseira do custo:

- Linha 2 tem custo $O(1)$;
- Linha 4 tem custo $O\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) = O(n)$;

Construção de um *heap* máximo - Implementação

```
1 void buildMaxHeap (int * v, int n) {
2   int lp = lastParent(n), comp = 0;
3 
4   for (int i = lp; i >= 0; i--)
5     comp += maxHeapify(v, n, i);
6 
7   return comp;
8 }
```

Análise grosseira do custo:

- Linha 2 tem custo $O(1)$;
- Linha 4 tem custo $O\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) = O(n)$;
- Linha 5 tem custo $O(\log(n))$ como descrito anteriormente.

Construção de um *heap* máximo - Implementação

```
1 void buildMaxHeap (int * v, int n) {
2   int lp = lastParent(n), comp = 0;
3 
4   for (int i = lp; i >= 0; i--)
5     comp += maxHeapify(v, n, i);
6 
7   return comp;
8 }
```

Análise grosseira do custo:

- Linha 2 tem custo $O(1)$;
- Linha 4 tem custo $O\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) = O(n)$;
- Linha 5 tem custo $O(\log(n))$ como descrito anteriormente.

Isso totaliza, então, $O(n \log(n))$.

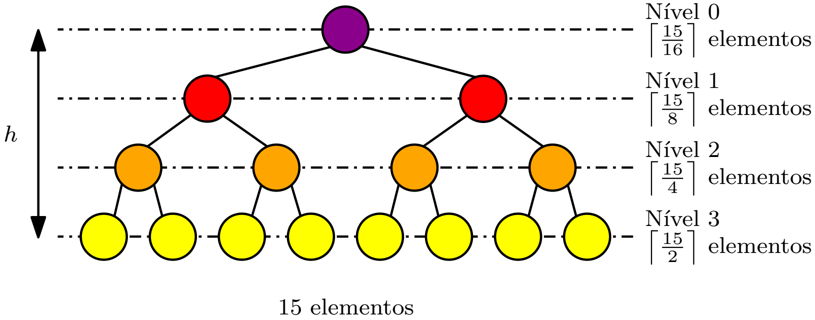
No entanto, pode-se fazer uma análise mais refinada. **Como?**

Construção de um *heap* máximo - Análise refinada

Dado um *heap* `v[0..n - 1]`:

- Sua altura h é $\lfloor \log(n) \rfloor$;
- O máximo de nós em uma altura h é $\lceil \frac{n}{2^{h+1}} \rceil$.

Então pode-se expressar o custo de `maxHeapify` por $O(h)$.



Construção de um *heap* máximo - Análise refinada

O total de passos N para construir um *heap* de tamanho n pode ser escrito matematicamente.

Na altura h, existem $\frac{n}{2^{h+1}}$ elementos em que o `maxHeapify` precisa ser chamado, e sabemos que o `maxHeapify` em uma altura h tem custo no caso pior de $O(h)$. Então:

$$
\begin{aligned}
N &= \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{n}{2^{h+1}} \cdot h \\
&= \frac{n}{2} \left(\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h} \right) \\
&\leq \frac{n}{2} \left(\sum_{h=0}^{\infty} \frac{h}{2^h} \right)
\end{aligned}
$$

A solução do último somatório pode ser achada derivando e multiplicando por x ambos os lados da série geométrica [1]:

$$\frac{\partial}{\partial x}\left(\sum_{k=0}^{\infty} x^k\right) = \frac{\partial}{\partial x}\left(\frac{1}{1-x}\right)$$

$$\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$$

$$\sum_{k=0}^{\infty} k x^{k} = \frac{x}{(1-x)^2}$$

[1] Para mais detalhes sobre, consultar o Cormen, apêndice A.8.

Construção de um *heap* máximo - Análise refinada

Então, substituindo $x = \frac{1}{2}$ na equação superior:

$$\sum_{k=0}^{\infty} k \cdot \left(\frac{1}{2}\right)^k = \frac{(1/2)}{(1-1/2)^2} = 2$$

Substituindo a solução do somatório de volta:

$$N \leq \frac{\cancel{2}n}{\cancel{2}} = n$$

Assim, o custo de `buildMaxHeap` no pior caso é $O(n)$ [1].

[1]Tópico no StackOverflow: `https://bit.ly/2HW0yCJ`

Heapsort

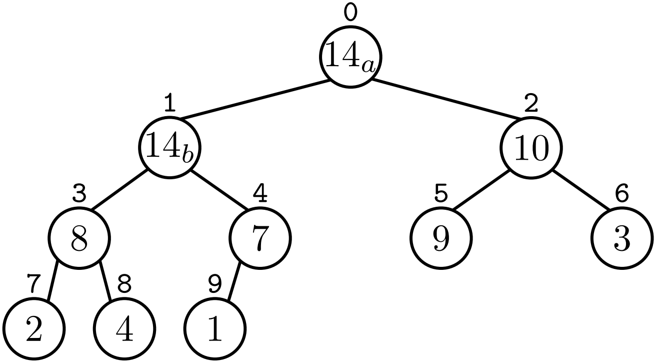
Heap Sort

- Permite ordenar um vetor em ordem crescente ou decrescente;
- Realiza muitas trocas entre os elementos;
- Não é necessário um vetor auxiliar como no Merge Sort;
- Não precisa escolher um ótimo pivô como no Quick Sort.

Heap Sort

```
HEAPSORT(v, n)
BUILD-MAX-HEAP(v, n)
para i ← n − 1 decrescendo até 1 faça
    v[0] ↔ v[i]
    MAX-HEAPIFY(v, i, 0)
```

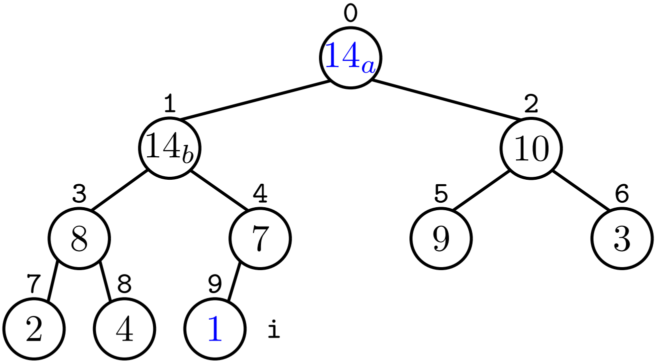
Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
14_a	14_b	10	8	7	9	3	2	4	1

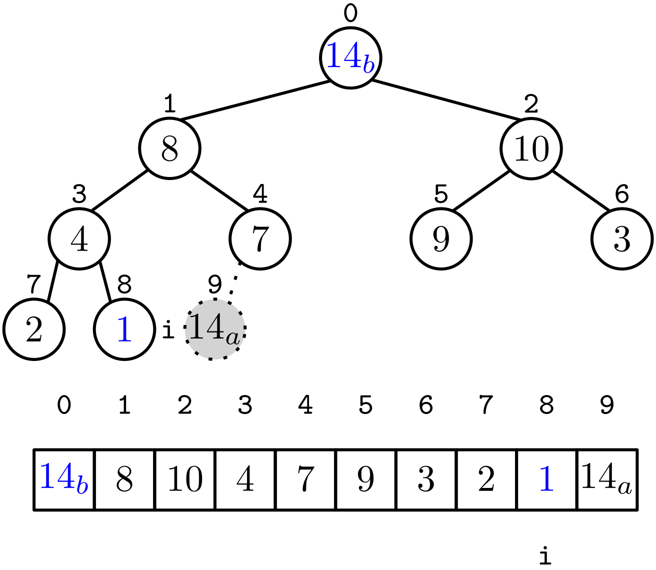
0 n - 1

Heap Sort - Exemplo

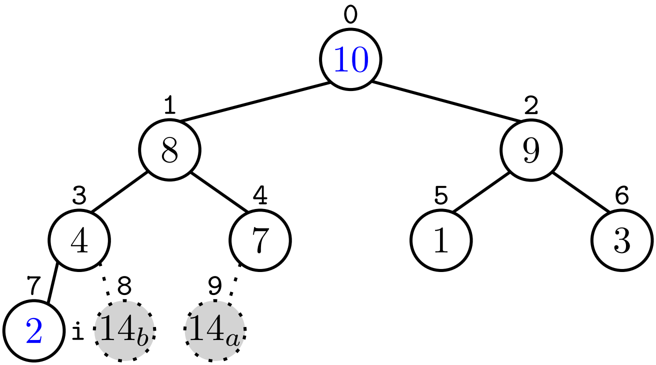


0	1	2	3	4	5	6	7	8	9
14_a	14_b	10	8	7	9	3	2	4	1
									i

Heap Sort - Exemplo

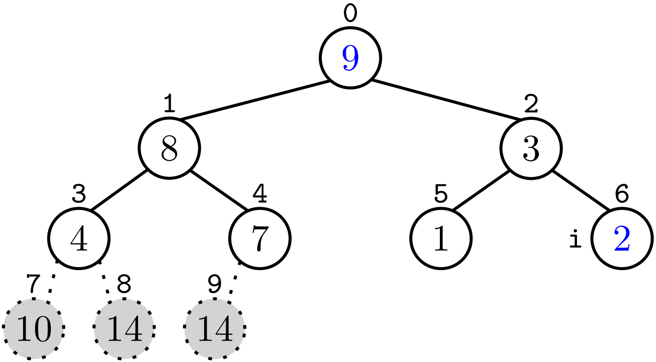


Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
10	8	9	4	7	1	3	2	14_b	14_a
							i		

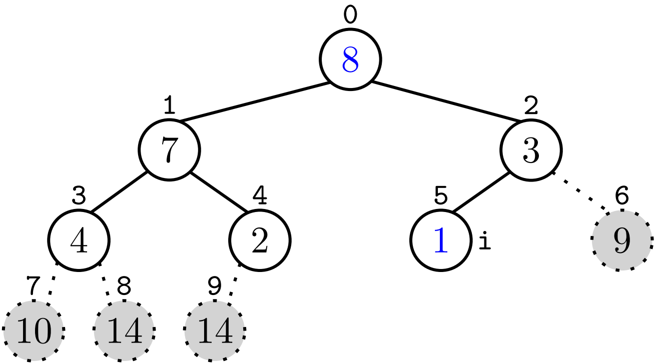
Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
9	8	3	4	7	1	2	10	14	14

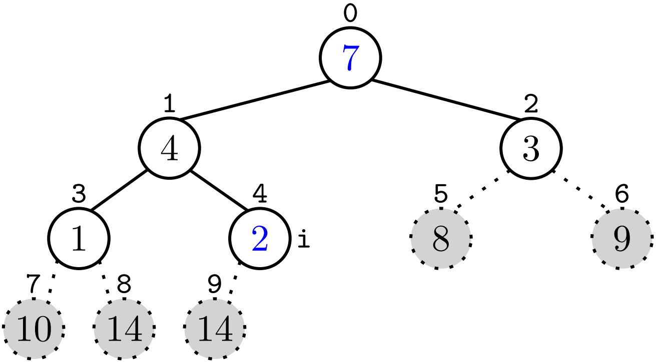
i

Heap Sort - Exemplo



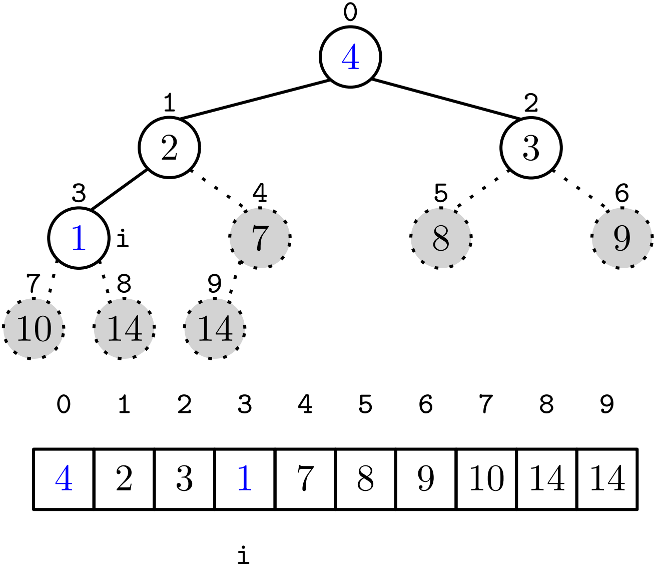
0	1	2	3	4	5	6	7	8	9
8	7	3	4	2	1	9	10	14	14

i

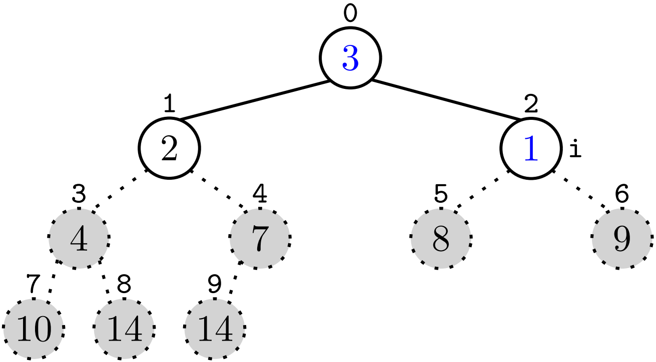


0	1	2	3	4	5	6	7	8	9
7	4	3	1	2	8	9	10	14	14
				i					

Heap Sort - Exemplo



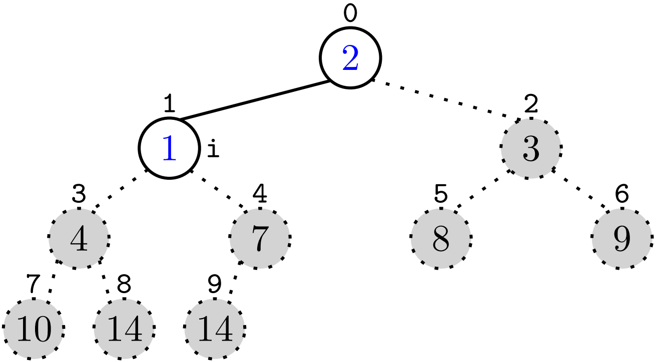
Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
3	2	1	4	7	8	9	10	14	14

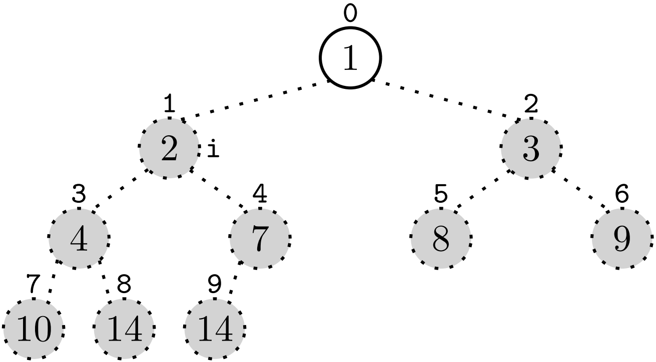
i

Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
2	1	3	4	7	8	9	10	14	14
	i								

Heap Sort - Exemplo



0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	14

i

Heapsort

Implementação

Heap Sort - Implementação

```c
int heapSort (int * v, int n) {
  int comp = 0;

  comp = buildMaxHeap(v, n);

  for (int i = n - 1; i >= 0; i--) {
    int aux = v[0];
    v[0] = v[i];
    v[i] = aux;

    comp += maxHeapify(v, i, 0);
  }

  return comp;
}
```

Heap Sort - Implementação

```c
int heapSort (int * v, int n) {
  int comp = 0;

  comp = buildMaxHeap(v, n);

  for (int i = n - 1; i >= 0; i--) {
    int aux = v[0];
    v[0] = v[i];
    v[i] = aux;

    comp += maxHeapify(v, i, 0);
  }

  return comp;
}
```

1. Construir um *heap* máximo;

Heap Sort - Implementação

```
int heapSort (int * v, int n) {
  int comp = 0;

  comp = buildMaxHeap(v, n);

  for (int i = n - 1; i >= 0; i--) {
    int aux = v[0];
    v[0] = v[i];
    v[i] = aux;

    comp += maxHeapify(v, i, 0);
  }

  return comp;
}
```

1. Construir um *heap* máximo;
2. Percorrer o heap de trás pra frente, de $i = n - 1$ até 1:

Heap Sort - Implementação

```
int heapSort (int * v, int n) {
  int comp = 0;

  comp = buildMaxHeap(v, n);

  for (int i = n - 1; i >= 0; i--) {
    int aux = v[0];
    v[0] = v[i];
    v[i] = aux;

    comp += maxHeapify(v, i, 0);
  }

  return comp;
}
```

1. Construir um *heap* máximo;
2. Percorrer o heap de trás pra frente, de $i = n - 1$ até 1:
   2.1 Trocar o elemento `v[0]` com o último, ou seja, `v[i]`.

Heap Sort - Implementação

```c
int heapSort (int * v, int n) {
  int comp = 0;

  comp = buildMaxHeap(v, n);

  for (int i = n - 1; i >= 0; i--) {
    int aux = v[0];
    v[0] = v[i];
    v[i] = aux;

    comp += maxHeapify(v, i, 0);
  }

  return comp;
}
```

1. Construir um *heap* máximo;
2. Percorrer o heap de trás pra frente, de $i = n - 1$ até 1:
   2.1 Trocar o elemento `v[0]` com o último, ou seja, `v[i]`.
   2.2 Chamar `maxHeapify` para refazer o *heap* máximo.

https://repl.it/@alessandrojean/HeapSortComparison

Heapsort

Análise do custo

Heap Sort - Análise do custo

- A chamada ao `buildMaxHeap` tem custo $O(n)$;
- As $n-1$ chamadas a `maxHeapify` tem custo $O(\log(n))$.

Juntando, temos:

$$\begin{aligned} T(n) &= O(n) + O((n-1)\cdot\log(n)) \\ &= O(n) + O(n\cdot\log(n)) \\ &= O(n\log(n)) \end{aligned}$$

Conclusões

Conclusões

- Algoritmo instável, com custo $O(n \log(n))$ no pior caso;
- Algoritmo in-place, não requer memória auxiliar;
- Efetua muitas comparações;
- É mais custoso fazer a manutenção do *heap* do que particionar, como o Quick Sort, por exemplo.
- Pode ordenar um vetor em ordem decrescente, utilizando funções para *heaps* mínimos (**Estudo independente**);
- Pode ser implementado de baixo para cima, gerando o Bottom-up heapsort com custo 0.25 vezes melhor do que o original (**Estudo independente**);
- Pode ser combinado com o Quick Sort, gerando o Intro Sort, que obtém as vantagens de ambos;

Recursos computacionais

- Visualização interativa do algoritmo:
  https://visualgo.net/en/heap
- Comparação com outros algoritmos:
  http://sorting.at
- Comparação entre diversas entradas:
  http://sorting-algorithms.com
- Funcionamento em vídeo *(em inglês)*:
  https://youtu.be/MtQL_ll5KhQ
- Animação comparando com o Merge sort *(em inglês)*:
  https://youtu.be/H5kAcmGOn4Q
- Heap sort em 4 minutos *(em inglês)*:
  https://youtu.be/2DmK_H7IdTo
- Aula da UNIVESP sobre Heap sort:
  https://youtu.be/KtlWBhaygH8
- Aula do MIT sobre Heap sort *(em inglês)*:
  https://youtu.be/B7hVxCmfPtM

Referências Bibliográficas

- SEDGEWICK, R. Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley, 1997.
- CORMEN, T. H. et al. Algoritmos: Teoria e Prática. Editora Campus, 2002.
- RAVULA, R. Algorithms lecture 13 - Build max heap algorithm and analysis. Youtube, Julho de 2014. Disponível em: https://youtu.be/HI97KDV23Ig
- MOTA, G. O. Análise de Algoritmos e Estruturas de Dados: Notas de aula. 2018.

Referências Bibliográficas

- MENA-CHALCO, J. P. Heap Sort. 2017. 65 slides. [1]
  http://professor.ufabc.edu.br/~jesus.mena/courses/mcta001-1q-2017/AED1-10.pdf
- RIBEIRO, M. P. Ordenação Heaps binários. 2018. 78 slides. [1]
  https://drive.google.com/file/d/1kA4ijg20vshT-b_RUFEV3mIBZZ2_aVGW/view
- BUENO, L. R. Heapsort. 2013. 62 slides. [1]
  http://professor.ufabc.edu.br/~leticia.bueno/classes/aa/materiais/heapsort.pdf
- SOUZA, J. F. de. HeapSort. 2009. 46 slides. [2]
  http://www.ufjf.br/jairo_souza/files/2009/12/2-Ordenação-HeapSort.pdf
- SANTI, J. Heap e Heap-Sort. 2017. 127 slides. [3]

---

[1] Material apresentado para a disciplina de Algoritmos e Estruturas de Dados I no curso de Ciência da Computação da UFABC.
[2] Material apresentado para a disciplina de Estrutura de Dados II no curso de Ciência da Computação da UFJF.
[3] Material apresentado para a disciplina de Estrutura de Dados II no curso de Sistemas de Informação da UTFPR.

Material disponível no GitHub.

https://github.com/alessandrojean/AED-I-2018.1

https://repl.it/@alessandrojean/HeapSort