# Algoritmo HeapSort

## Estrutura de Dados — QXD0010

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ


Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará


1º semestre/2021

# Introdução

# Introdução

- Algoritmo criado por John Williams em 1964.

# Introdução

- Algoritmo criado por John Williams em 1964.
- Complexidade $O(n \log n)$ no pior caso.

# Introdução

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Algoritmo criado por John Williams em 1964.
- Complexidade $O(n \log n)$ no pior caso.
- Ao contrário do Mergesort, não requer um vetor auxiliar (é in loco)

# Introdução

- Algoritmo criado por John Williams em 1964.
- Complexidade $O(n \log n)$ no pior caso.
- Ao contrário do Mergesort, não requer um vetor auxiliar (é in loco)
- Utiliza abordagem similar a do selectionSort:

# Introdução

Universidade Federal do Ceará
Campus Quixadá

- Algoritmo criado por John Williams em 1964.
- Complexidade $O(n \log n)$ no pior caso.
- Ao contrário do Mergesort, não requer um vetor auxiliar (é in loco)
- Utiliza abordagem similar a do selectionSort:
  - SelectionSort seleciona o $i$-ésimo menor elemento e o coloca na $(i-1)$-ésima posição do vetor.

# Introdução

- Algoritmo criado por John Williams em 1964.
- Complexidade $O(n \log n)$ no pior caso.
- Ao contrário do Mergesort, não requer um vetor auxiliar (é in loco)
- Utiliza abordagem similar a do selectionSort:
  - SelectionSort seleciona o $i$-ésimo menor elemento e o coloca na $(i-1)$-ésima posição do vetor.
  - Para ordenar em ordem crescente, o Heapsort põe o maior elemento no final do vetor e o segundo maior antes dele, e assim por diante.

# Estrutura de dados Heap Binário

- Existem dois tipos de heap: heap máximo e heap mínimo.
- Nesta aula analisamos o heap binário máximo.
- Chamaremos apenas de heap.

# Estrutura de dados Heap Binário



- Existem dois tipos de heap: heap máximo e heap mínimo.
- Nesta aula analisamos o heap binário máximo.
- Chamaremos apenas de heap.
- Os heaps têm uma propriedade estrutural e uma propriedade de ordem.
    - Uma operação em um heap pode destruir uma das propriedades e não deve terminar até que todas as propriedades do heap estejam satisfeitas.

# Heap — Propriedade de ordem

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Um heap é um vetor $A[1 \dots n]$ tal que

$$A[\lfloor i/2 \rfloor] \geq A[i], \text{ para } 2 \leq i \leq n.$$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 999 | 888 | 777 | 555 | 666 | 777 | 555 | 222 | 333 | 444 | 111 | 333 | 666 | 333 |

# Heap — Propriedade de ordem

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Um heap é um vetor $A[1 \ldots n]$ tal que

$$A[\lfloor i/2 \rfloor] \geq A[i], \text{ para } 2 \leq i \leq n.$$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 999 | 888 | 777 | 555 | 666 | 777 | 555 | 222 | 333 | 444 | 111 | 333 | 666 | 333 |

- **Observação I:** A fim de simplificar a descrição do algoritmo, supomos que os índices do vetor são $1 \ldots n$ e não $0 \ldots n-1$.

# Heap — Propriedade de ordem

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Um heap é um vetor $A[1 \dots n]$ tal que

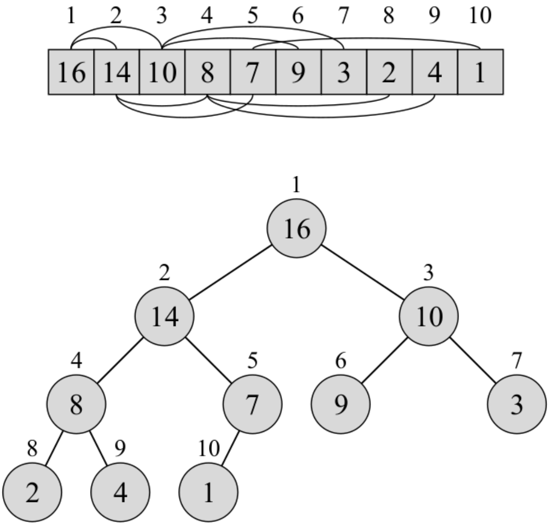
$$A[\lfloor i/2 \rfloor] \geq A[i], \text{ para } 2 \leq i \leq n.$$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 999 | 888 | 777 | 555 | 666 | 777 | 555 | 222 | 333 | 444 | 111 | 333 | 666 | 333 |

- **Observação I:** A fim de simplificar a descrição do algoritmo, supomos que os índices do vetor são $1 \dots n$ e não $0 \dots n-1$.

- **Observação II:** Segue imediatamente da definição que $A[1]$ é um elemento máximo do heap.

# Heap — Propriedade estrutural



# Árvores Binárias Completas



- Uma árvore binária é dita completa se todos os níveis estão cheios, com exceção possivelmente do último, que pode ou não estar cheio.

# Árvores Binárias Completas



- Uma árvore binária é dita completa se todos os níveis estão cheios, com exceção possivelmente do último, que pode ou não estar cheio.

No heap, os nós do último nível estão o mais a esquerda possível.

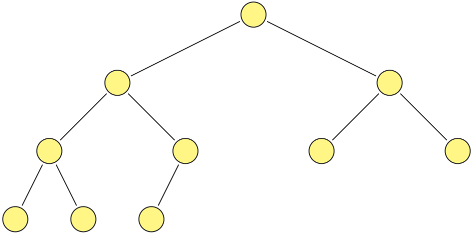
# Árvores Binárias Completas



- Uma árvore binária é dita completa se todos os níveis estão cheios, com exceção possivelmente do último, que pode ou não estar cheio.

No heap, os nós do último nível estão o mais a esquerda possível.

Exemplo:

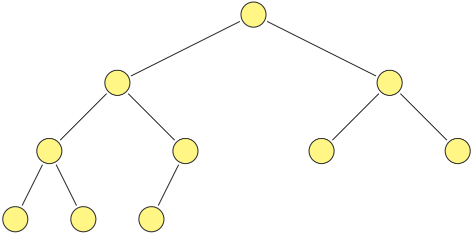


# Árvores Binárias Completas

- Uma árvore binária é dita completa se todos os níveis estão cheios, com exceção possivelmente do último, que pode ou não estar cheio.

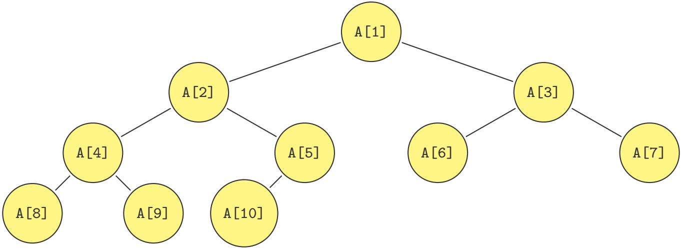
No heap, os nós do último nível estão o mais a esquerda possível.

Exemplo:

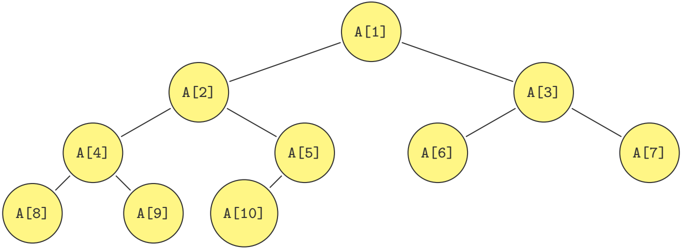


Uma árvore binária completa com $n$ nós tem $\lfloor \log n \rfloor + 1$ níveis.

# Árvores Binárias Completas e Vetores



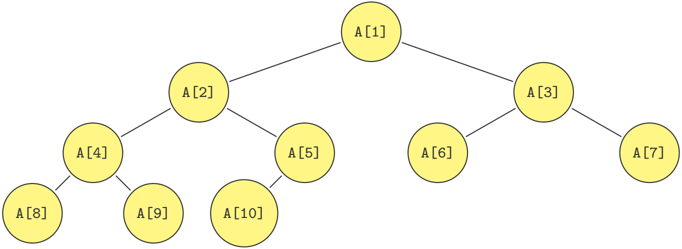
# Árvores Binárias Completas e Vetores




- O nó `A[1]` é a raiz da árvore.

# Árvores Binárias Completas e Vetores

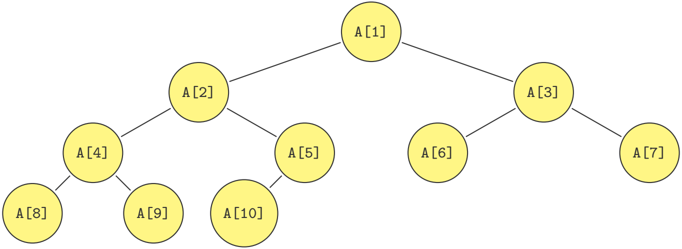





- O nó `A[1]` é a raiz da árvore.

Em relação a `A[i]`:

# Árvores Binárias Completas e Vetores

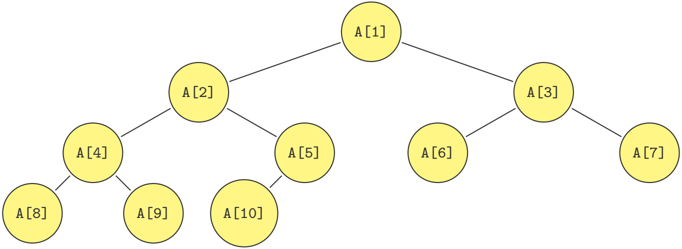



- O nó `A[1]` é a raiz da árvore.

Em relação a `A[i]`:

- o filho esquerdo é `A[2i]` e o filho direito é `A[2i+1]`

# Árvores Binárias Completas e Vetores

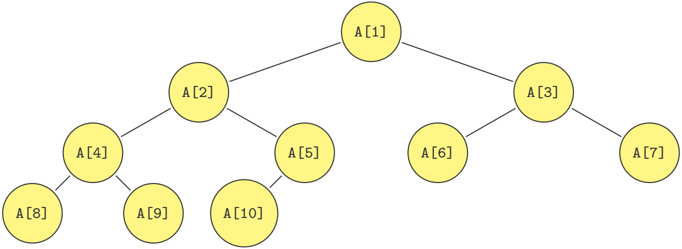



- O nó `A[1]` é a raiz da árvore.

Em relação a `A[i]`:

- o filho esquerdo é `A[2i]` e o filho direito é `A[2i+1]`
- o pai é `A[i/2]`

# Árvores Binárias Completas e Vetores



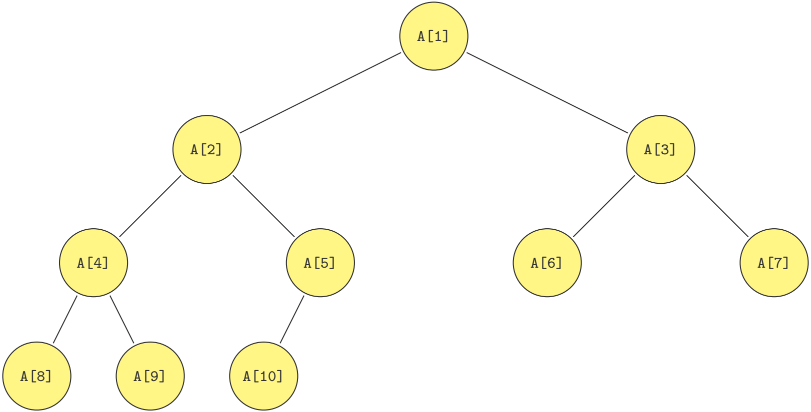

- O nó `A[1]` é a raiz da árvore.

Em relação a `A[i]`:

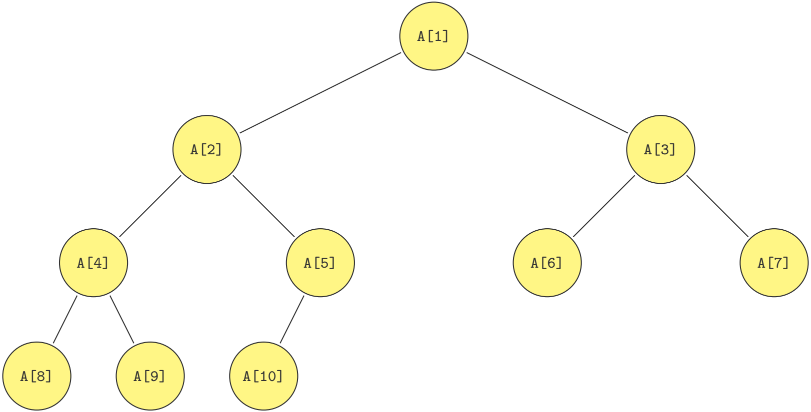
- o filho esquerdo é `A[2i]` e o filho direito é `A[2i+1]`
- o pai é `A[i/2]`

**Atenção:** A[1] não tem pai, o filho esquerdo de $A[i]$ só existe se $2i \leq n$ e o filho direito de $A[i]$ só existe se $2i + 1 \leq n$.

# Heap

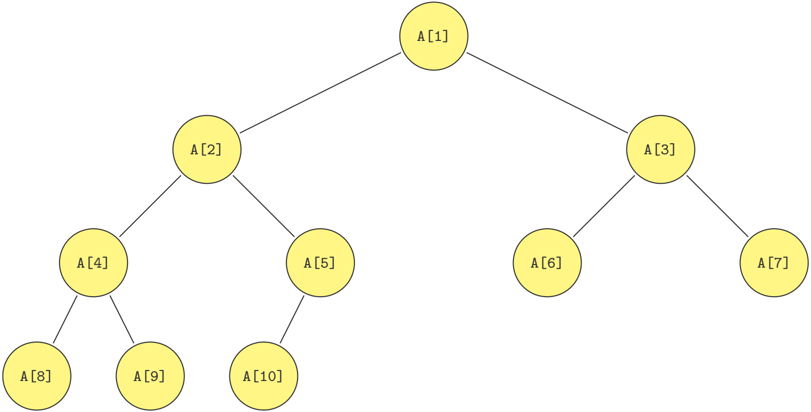


# Heap




Em um Heap:

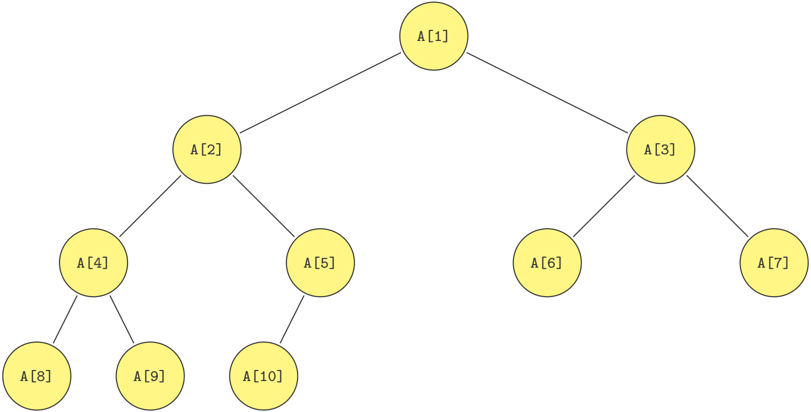
# Heap




Em um Heap:

- Os filhos são menores ou iguais ao pai

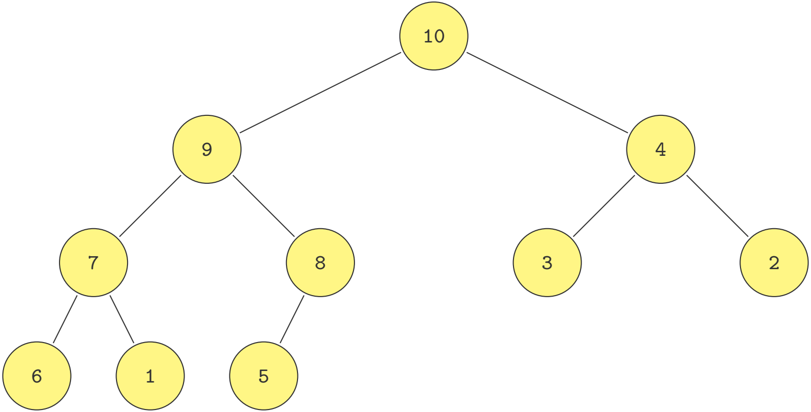
# Heap




Em um Heap:

- Os filhos são menores ou iguais ao pai
- Ou seja, a raiz é o máximo

# Heap




Em um Heap:

- Os filhos são menores ou iguais ao pai
- Ou seja, a raiz é o máximo

# Construção de um heap

- O algoritmo Heapsort recebe um vetor $A[1..n]$ como entrada e, para que possa ordená-lo com sucesso, ele precisa, antes de tudo, transformar o vetor $A$ em um heap.

# Construção de um heap

- O algoritmo Heapsort recebe um vetor $A[1..n]$ como entrada e, para que possa ordená-lo com sucesso, ele precisa, antes de tudo, transformar o vetor $A$ em um heap.

- Para entender o Heapsort, será preciso tratar, às vezes, de heaps com defeitos.

# Construção de um heap



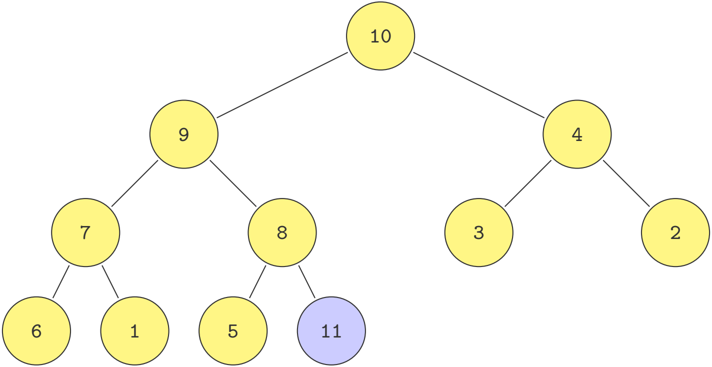
- O algoritmo Heapsort recebe um vetor $A[1..n]$ como entrada e, para que possa ordená-lo com sucesso, ele precisa, antes de tudo, transformar o vetor $A$ em um heap.

- Para entender o Heapsort, será preciso tratar, às vezes, de heaps com defeitos.

- Um vetor $A[1..n]$ é um quase-max-heap se existe um único índice $k$ tal que a desigualdade $A[i/2] \geq A[i]$ vale para todo $i$ diferente de $k$.

# Exemplo: Vetor que é quase um Heap

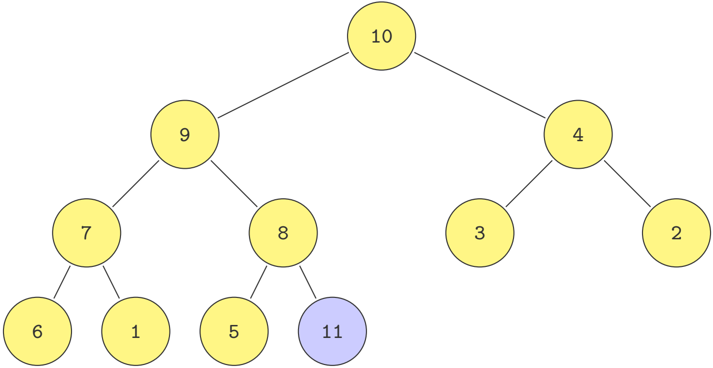
Como consertar?



# Exemplo: Vetor que é quase um Heap



## Como consertar?



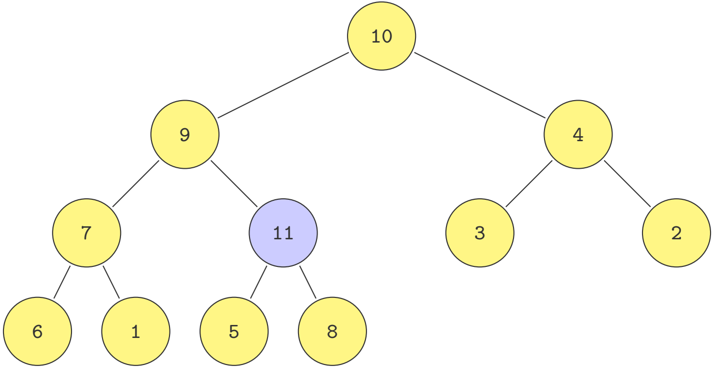

Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Exemplo: Vetor que é quase um Heap

Universidade Federal do Ceará
Campus Quixadá

## Como consertar?



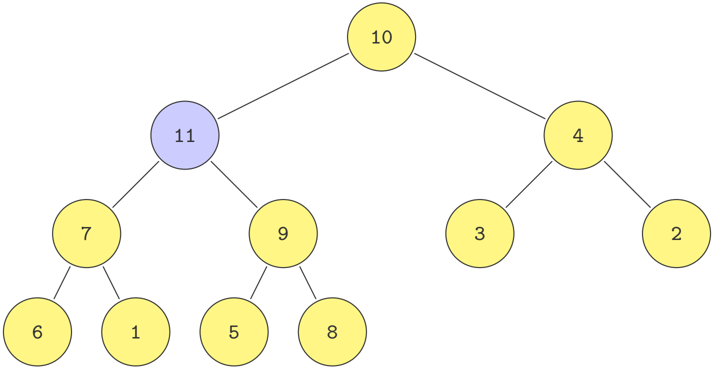

Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Exemplo: Vetor que é quase um Heap



Como consertar?



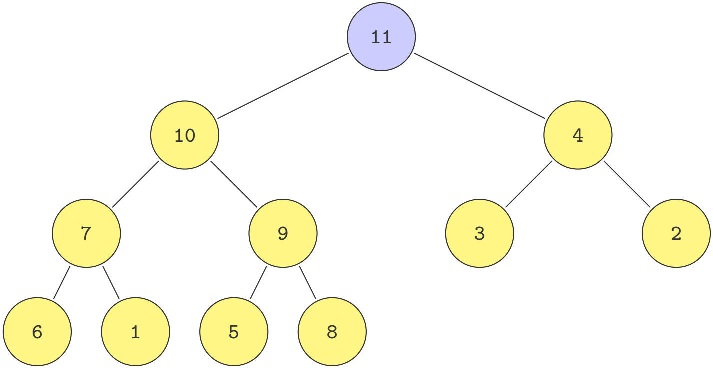

Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Exemplo: Vetor que é quase um Heap



## Como consertar?




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Construção de um heap

- É fácil rearranjar um vetor $A[1..n]$ que é um quase-max-heap para que ele se torne um heap.

# Construção de um heap



- É fácil rearranjar um vetor $A[1..n]$ que é um quase-max-heap para que ele se torne um heap.
- A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai:

# Construção de um heap

- É fácil rearranjar um vetor $A[1..n]$ que é um quase-max-heap para que ele se torne um heap.
- A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai:
  - troque os valores de pai e filho e suba um passo em direção à raiz.

# Construção de um heap



- É fácil rearranjar um vetor $A[1..n]$ que é um quase-max-heap para que ele se torne um heap.
- A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai:
    - troque os valores de pai e filho e suba um passo em direção à raiz.
- Mais precisamente, se $A[i/2] < A[i]$, então troque o valor de $A[i]$ com o valor de $A[i/2]$ e, em seguida, faça $i = i/2$, repetindo o processo para o novo valor de $i$.

# Construção de um heap

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- É fácil rearranjar um vetor $A[1..n]$ que é um quase-max-heap para que ele se torne um heap.
- A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai:
    - troque os valores de pai e filho e suba um passo em direção à raiz.
- Mais precisamente, se $A[i/2] < A[i]$, então troque o valor de $A[i]$ com o valor de $A[i/2]$ e, em seguida, faça $i = i/2$, repetindo o processo para o novo valor de $i$.
- Como usar isso para transformar um vetor qualquer em um heap?

# Construção de um heap



```
/**
 * Recebe um vetor A[1..n] e o transforma em
 * um max-heap. Note que n eh o indice do
 * ultimo elemento.
 */
void constroiHeap (int A[], int n) {
    for (int k = 1; k < n; k++) {
        // v[1..k] eh um heap
        int f = k + 1;
        while (f > 1 && A[f/2] < A[f]) {
            int aux = A[f/2];
            A[f/2] = A[f];
            A[f] = aux;
            f = f/2;
        }
    }
}
```

# Tempo de execução de constroiHeap()

```
void constroiHeap (int A[], int n) {
    for (int k = 1; k < n; k++) {
        // v[1..k] eh um heap
        int f = k + 1;
        while (f > 1 && A[f/2] < A[f]) {
            int aux = A[f/2];
            A[f/2] = A[f];
            A[f] = aux;
            f = f/2;
        }
    }
}
```

# Tempo de execução de constroiHeap()

```
void constroiHeap (int A[], int n) {
    for (int k = 1; k < n; k++) {
        // v[1..k] eh um heap
        int f = k + 1;
        while (f > 1 && A[f/2] < A[f]) {
            int aux = A[f/2];
            A[f/2] = A[f];
            A[f] = aux;
            f = f/2;
        }
    }
}
```

- Em cada iteração do laço **while**, o índice $f$ pula de um nível da "árvore" para o nível anterior.

# Tempo de execução de `constroiHeap()`

Universidade Federal do Ceará
Campus Quixadá

```
void constroiHeap (int A[], int n) {
    for (int k = 1; k < n; k++) {
        // v[1..k] eh um heap
        int f = k + 1;
        while (f > 1 && A[f/2] < A[f]) {
            int aux = A[f/2];
            A[f/2] = A[f];
            A[f] = aux;
            f = f/2;
        }
    }
}
```

- Em cada iteração do laço **while**, o índice $f$ pula de um nível da “árvore” para o nível anterior.
- Portanto, esse bloco de linhas é repetido no máximo $\log k$ vezes para cada $k$ fixo.

# Tempo de execução de `constroiHeap()`

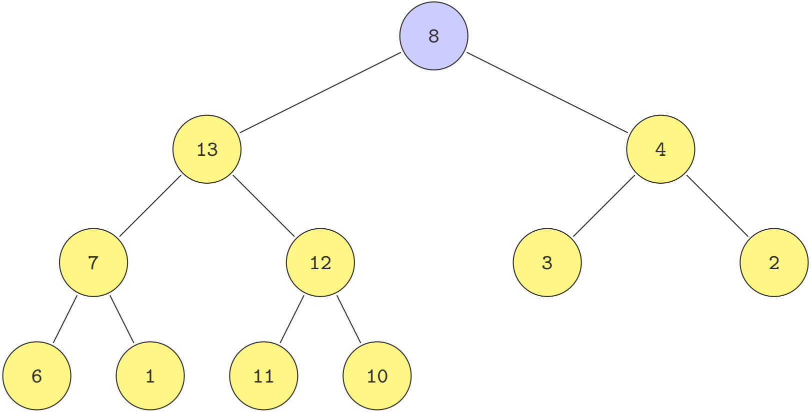
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  void constroiHeap (int A[], int n) {
2      for (int k = 1; k < n; k++) {
3          // v[1..k] eh um heap
4          int f = k + 1;
5          while (f > 1 && A[f/2] < A[f]) {
6              int aux = A[f/2];
7              A[f/2] = A[f];
8              A[f] = aux;
9              f = f/2;
10         }
11     }
12 }
```

- Em cada iteração do laço **while**, o índice $f$ pula de um nível da “árvore” para o nível anterior.
- Portanto, esse bloco de linhas é repetido no máximo $\log k$ vezes para cada $k$ fixo.
- Logo, o número total de comparações entre elementos do vetor (todas acontecem na linha 5) não passa de $n \log n$.

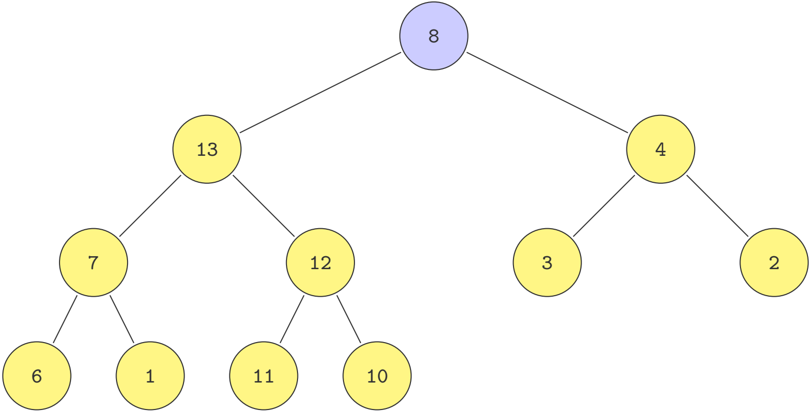
# Exemplo: Outro vetor que é quase um Heap

Como consertar?



# Exemplo: Outro vetor que é quase um Heap

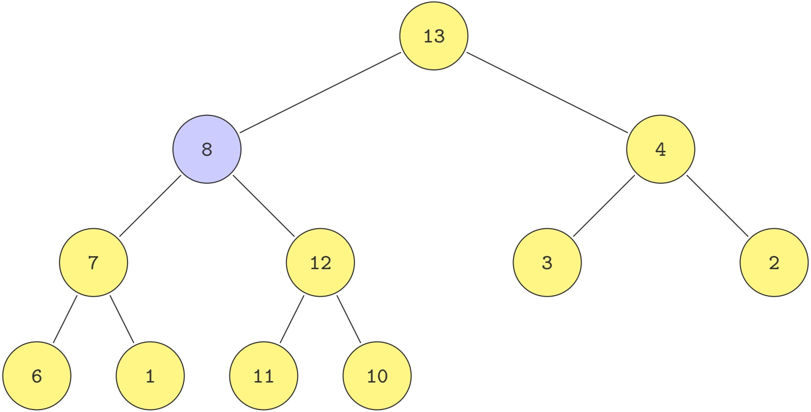
Como consertar?




Agora, vamos descer no heap, trocando o pai com o seu maior filho, caso esse filho seja maior que o pai.

# Exemplo: Outro vetor que é quase um Heap

Como consertar?

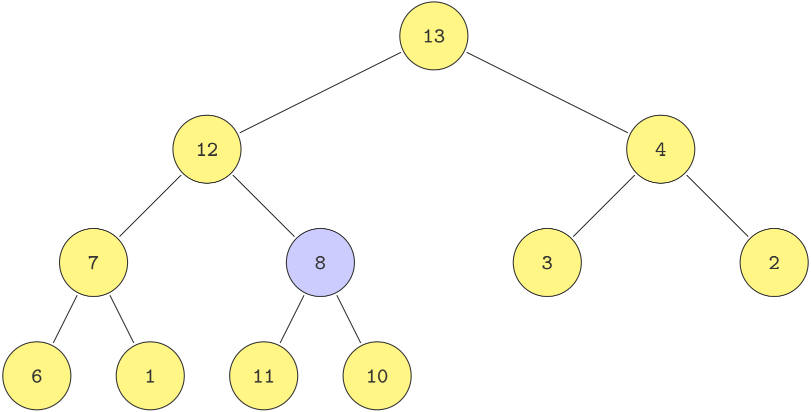



Agora, vamos descer no heap, trocando o pai com o seu maior filho, caso esse filho seja maior que o pai.

# Exemplo: Outro vetor que é quase um Heap



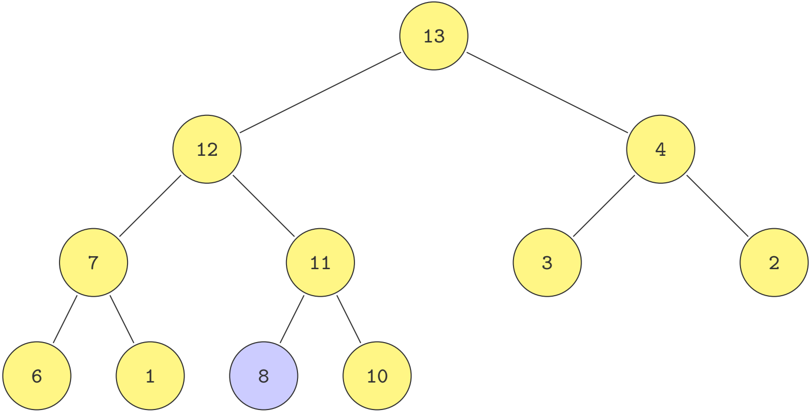
Como consertar?




Agora, vamos descer no heap, trocando o pai com o seu maior filho, caso esse filho seja maior que o pai.

# Exemplo: Outro vetor que é quase um Heap

Como consertar?




Agora, vamos descer no heap, trocando o pai com o seu maior filho, caso esse filho seja maior que o pai.

# A função `sacodeHeap`

- O coração de muitos algoritmos que manipulam heaps é uma função que, ao contrário de `constroiHeap`, desce em direção ao último nível da árvore.

# A função `sacodeHeap`



- O coração de muitos algoritmos que manipulam heaps é uma função que, ao contrário de `constroiHeap`, desce em direção ao último nível da árvore.

- Essa função, que chamaremos sacodeHeap, recebe um vetor qualquer $A[1..n]$ e faz $A[1]$ descer até sua posição correta, pulando de um nível para o seguinte da "árvore".

# A função `sacodeHeap`

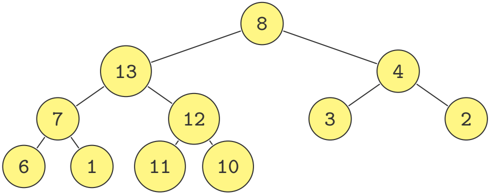
- O coração de muitos algoritmos que manipulam heaps é uma função que, ao contrário de `constroiHeap`, desce em direção ao último nível da árvore.

- Essa função, que chamaremos sacodeHeap, recebe um vetor qualquer $A[1..n]$ e faz $A[1]$ descer até sua posição correta, pulando de um nível para o seguinte da "árvore".

- Como isso é feito?
  - Se $A[1] \geqslant A[2]$ e $A[1] \geqslant A[3]$, não é preciso fazer nada.
  - Se $A[1] < A[2]$ e $A[2] \geqslant A[3]$, basta trocar $A[1]$ com $A[2]$ e depois fazer $A[2]$ descer para sua posição correta.
  - Os outros casos são semelhantes.

# A função sacodeHeap

```
/** Rearranja um quase-max-heap A[1..n] de
 * modo a transforma-lo em um max-heap. */
void sacodeHeap (int A[], int n) {
    int f = 2;
    while (f <= n) {
        if (f < n && A[f] < A[f+1])
            f++;
        // f eh o maior filho de f/2
        if (A[f/2] >= A[f])
            break;
        int aux = A[f/2];
        A[f/2] = A[f];
        A[f] = aux;
        f = f*2;
    }
}
```



# Análise de Complexidade

## `sacodeHeap`

- A função `sacodeHeap` faz no máximo $\log n$ iterações, uma vez que a árvore completa gerada pelo vetor tem $1 + \lfloor \log n \rfloor$ níveis.

# Análise de Complexidade

## `sacodeHeap`

- A função `sacodeHeap` faz no máximo $\log n$ iterações, uma vez que a árvore completa gerada pelo vetor tem $1 + \lfloor \log n \rfloor$ níveis.

- Cada iteração envolve duas comparações entre elementos do vetor e portanto o número total de comparações não passa de $2 \log n$.

# Análise de Complexidade

## `sacodeHeap`

- A função `sacodeHeap` faz no máximo $\log n$ iterações, uma vez que a árvore completa gerada pelo vetor tem $1 + \lfloor \log n \rfloor$ níveis.
- Cada iteração envolve duas comparações entre elementos do vetor e portanto o número total de comparações não passa de $2 \log n$.
- O consumo de tempo é proporcional ao número de comparações e, portanto, proporcional a $\log n$ no pior caso.

# Heapsort

# O algoritmo Heapsort



- Não é difícil reunir os dois algoritmos que vimos para obter um algoritmo que rearranja um vetor $A[1..n]$ em ordem crescente.

- O algoritmo tem duas fases:

# O algoritmo Heapsort

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Não é difícil reunir os dois algoritmos que vimos para obter um algoritmo que rearranja um vetor $A[1..n]$ em ordem crescente.

- O algoritmo tem duas fases:
  - a primeira transforma o vetor em heap

# O algoritmo Heapsort



- Não é difícil reunir os dois algoritmos que vimos para obter um algoritmo que rearranja um vetor $A[1..n]$ em ordem crescente.

- O algoritmo tem duas fases:
  - a primeira transforma o vetor em heap
  - a segunda rearranja o heap em ordem crescente.

# O algoritmo Heapsort



- Não é difícil reunir os dois algoritmos que vimos para obter um algoritmo que rearranja um vetor $A[1..n]$ em ordem crescente.

- O algoritmo tem duas fases:
  - a primeira transforma o vetor em heap
  - a segunda rearranja o heap em ordem crescente.

# O algoritmo Heapsort

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
// Rearranja vetor A[1..n] de modo que ele fique crescente
void heapsort (int A[], int n) {
    constroiHeap(A, n);

    // ordena o vetor
    for (int k = n; k >= 2; k--) {
        int aux = A[1];
        A[1] = A[k];
        A[k] = aux;
        sacodeHeap(A, k-1);
    }
}
```

# Análise de Complexidade do Heapsort

Universidade Federal do Ceará
Campus Quixadá

- Quantas comparações entre elementos do vetor a função `heapsort` executa?

# Análise de Complexidade do Heapsort



- Quantas comparações entre elementos do vetor a função `heapsort` executa?
- A função auxiliar `constroiHeap` faz $n \log n$ comparações no máximo.

# Análise de Complexidade do Heapsort

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Quantas comparações entre elementos do vetor a função `heapsort` executa?
- A função auxiliar `constroiHeap` faz $n \log n$ comparações no máximo.
- Em seguida, a função `peneira` é chamada cerca de $n$ vezes e cada uma dessas chamadas faz $2 \log n$ comparações, no máximo.

# Análise de Complexidade do Heapsort

Universidade Federal do Ceará
Campus Quixadá

- Quantas comparações entre elementos do vetor a função `heapsort` executa?
- A função auxiliar `constroiHeap` faz $n \log n$ comparações no máximo.
- Em seguida, a função `peneira` é chamada cerca de $n$ vezes e cada uma dessas chamadas faz $2 \log n$ comparações, no máximo.
- Logo, o número total de comparações não passa de $3n \log n$.

# Análise de Complexidade do Heapsort

- Quantas comparações entre elementos do vetor a função `heapsort` executa?

- A função auxiliar `constroiHeap` faz $n \log n$ comparações no máximo.

- Em seguida, a função `peneira` é chamada cerca de $n$ vezes e cada uma dessas chamadas faz $2 \log n$ comparações, no máximo.

- Logo, o número total de comparações não passa de $3n \log n$.

- Quanto ao consumo de tempo do `heapsort`, ele é proporcional ao número de comparações entre elementos do vetor e, portanto, proporcional a $n \log n$ no pior caso.

# FIM