DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Heap binário

Professores: Anisio Lacerda
Lucas Ferreira
Wagner Meira Jr.
Washington Cunha

Heap binário

Durante nossa aula sobre árvores mencionamos um outro tipo de política de inserção de nós, que chamamos de **Heap**. Um Heap binário é uma árvore binária que satisfaz a seguinte propriedade:

O valor gravado em um nó é sempre maior ou igual ao valor gravado em seus sucessores.

Heap binário

Note que não necessariamente essa estrutura armazena apenas inteiros, este valor pode ser apenas uma **chave** ou **id**, associado a uma estrutura de dados mais complexa.

Por fins de simplicidade nos nossos exemplos utilizaremos apenas inteiros.

Heap binário - operações

Para esta estrutura estamos interessados nas seguintes operações:

- Inserir um elemento
- Remover um elemento

Note que a inserção e remoção, independentemente da política utilizada, deve manter a propriedade do Heap ao longo dos nós. Por isso surgirá a necessidade de "consertar" a árvore, que chamamos de *Heapify*.

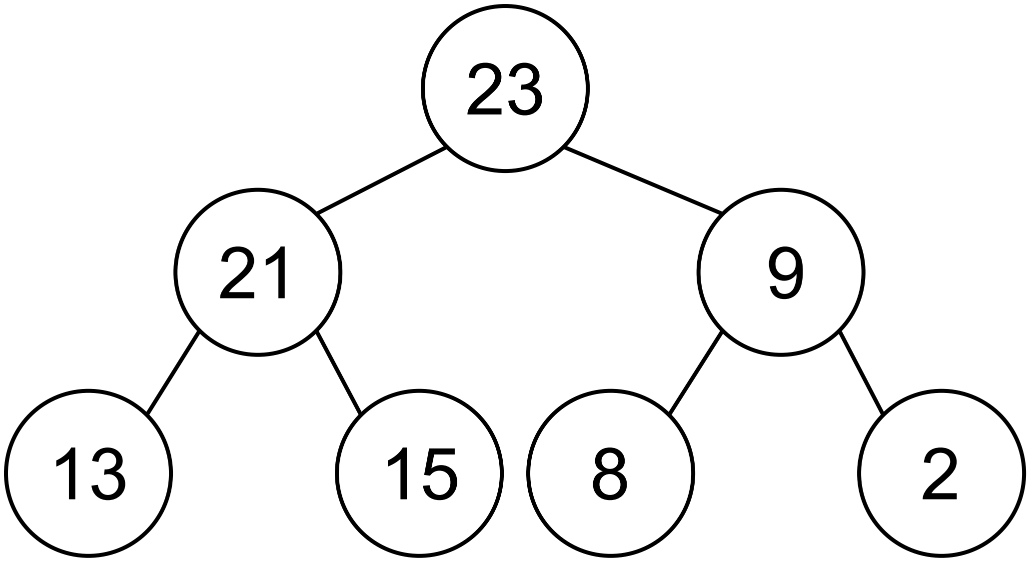
Heap binário - representação

Mas antes de entendermos as operações, precisamos definir como iremos representar nossa árvore.

A escolha mais intuitiva seria utilizarmos ponteiros assim como fizemos com a árvore binária de busca. No entanto para não aumentar muito a complexidade de tempo das operações, iremos adotar uma estratégia diferente.

A estratégia será **simular** uma árvore em um vetor!

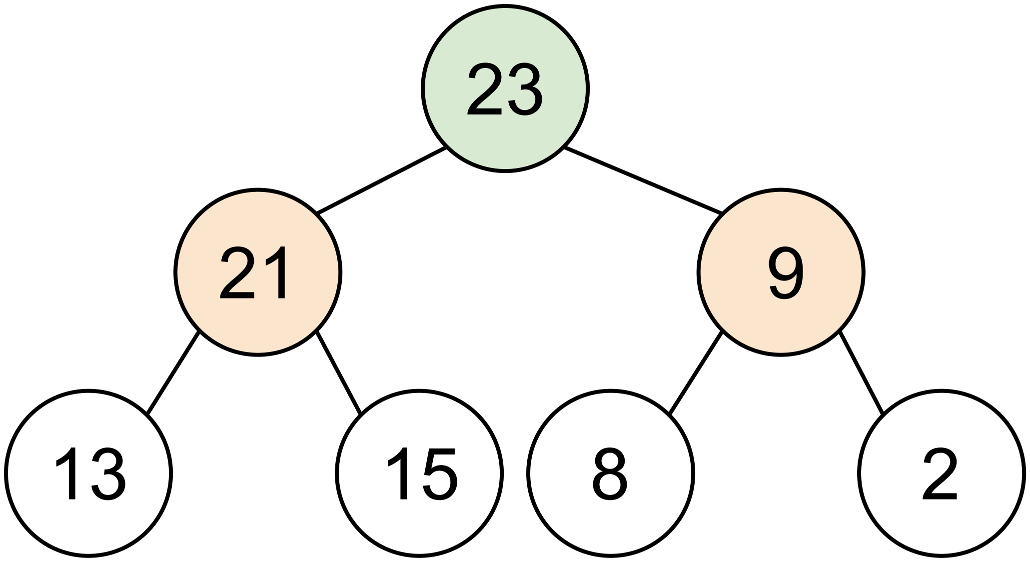
Heap binário - representação



0	1	2	3	4	5	6
23	21	9	13	15	8	2

Este é um exemplo de uma árvore e a representação que utilizaremos com um vetor ao lado. A ordem em que os nós aparecem no vetor é a resultante de um caminhamento por nível.

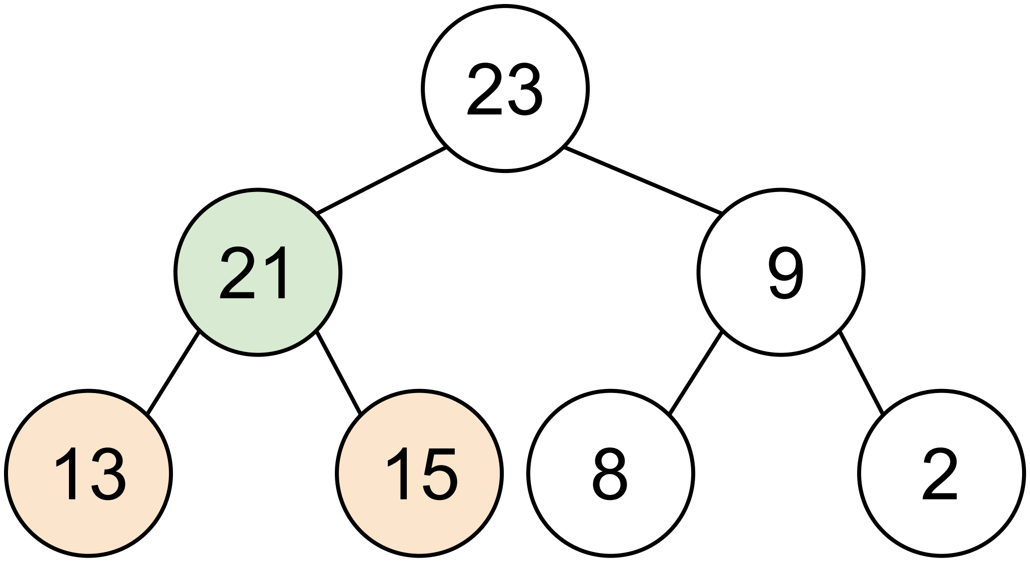
Heap binário - representação



0	1	2	3	4	5	6
23	21	9	13	15	8	2

Note que com essa representação se um nó aparece na posição **i** do vetor, então seus sucessores da esquerda e da direita aparecerão, respectivamente nas posições **2*i+1** e **2*i+2**.

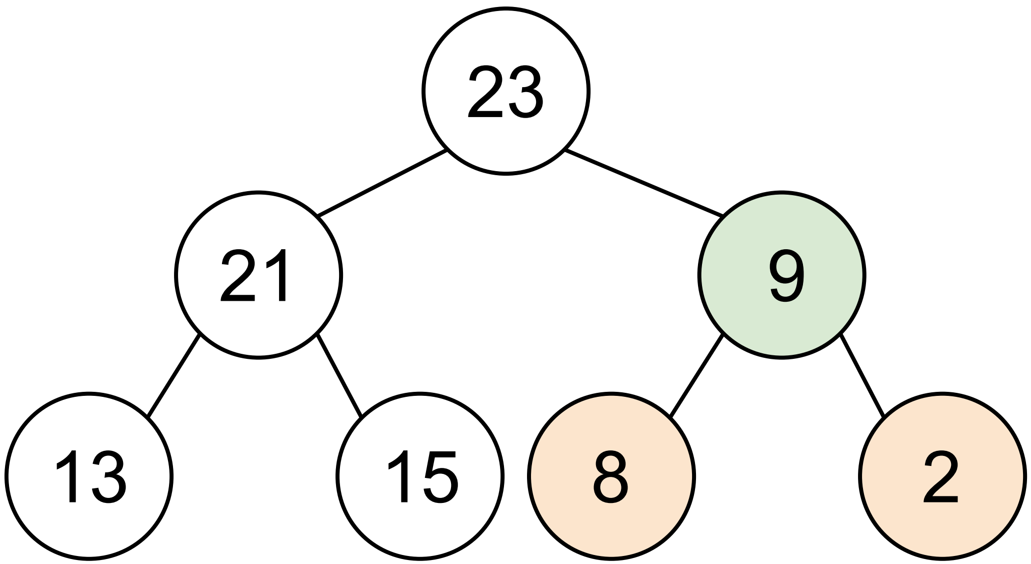
Heap binário - representação



0	1	2	3	4	5	6
23	21	9	13	15	8	2

Note que com essa representação se um nó aparece na posição **i** do vetor, então seus sucessores da esquerda e da direita aparecerão, respectivamente nas posições **2*i+1** e **2*i+2**.

Heap binário - representação



0	1	2	3	4	5	6
23	21	9	13	15	8	2

Note que com essa representação se um nó aparece na posição **i** do vetor, então seus sucessores da esquerda e da direita aparecerão, respectivamente nas posições **2*i+1** e **2*i+2**.

Heap binário - representação

Utilizando essa representação podemos construir essas três funções. Tendo elas em mãos podemos navegar pela árvore.

```
int GetAncestral(int i){
    return (i-1)/2;
}
int GetSucessorEsq(int i){
    return 2 * i + 1;
}
int GetSucessorDir(int i){
    return 2 * i + 2;
}
```

Heap binário - Inserção

A inserção será dividida em duas partes:

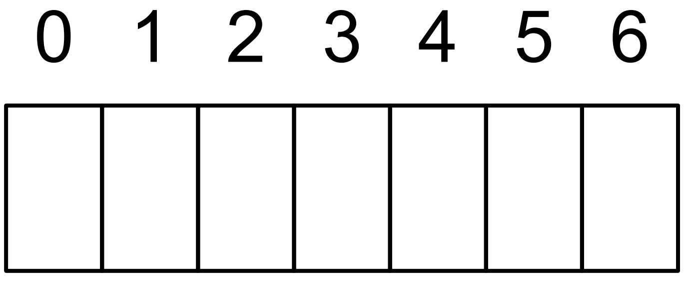
- O novo elemento será inserido como uma folha da árvore.
- Isso pode quebrar nossa propriedade, então precisamos verificar se é necessário “consertar” a árvore.

Na nossa representação o que faremos é inserir o novo elemento no final do vetor.

Heap binário - Exemplo de inserção

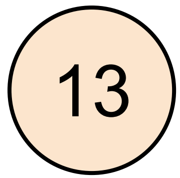
$\emptyset$

Tamanho = 0



Assim como nas outras implementações que fizemos utilizando vetores, nossa árvore terá um tamanho máximo. Neste exemplo será 7. Então temos nosso vetor reservado e por hora, a árvore está vazia.

Heap binário - Exemplo de inserção

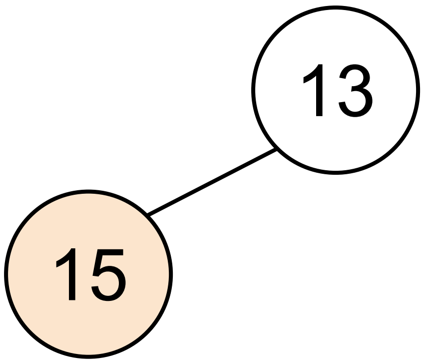


Tamanho = 1

0	1	2	3	4	5	6
13						

O primeiro elemento inserido foi 13. A árvore estava vazia então após a inserção ela continua sendo um Heap.

Heap binário - Exemplo de inserção

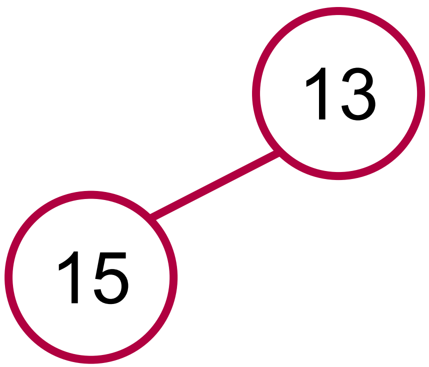


Tamanho = 2

0	1	2	3	4	5	6
13	15					

Agora foi inserido o valor 15. Note que após a inserção a árvore deixou de ser um heap, pois o 15 é sucessor do 13 mas é maior que ele. Precisamos então “consertar” a árvore.

Heap binário - Exemplo de inserção

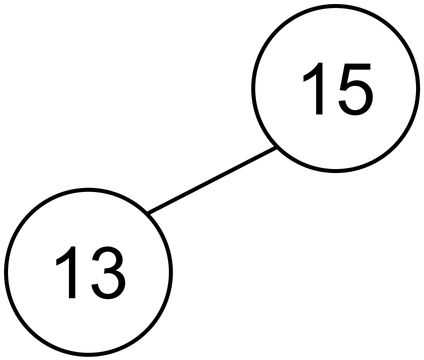


Tamanho = 2

0	1	2	3	4	5	6
13	15					

Como a árvore até então era um heap até a inserção ser efetuada, basta verificar se o nó recém inserido é maior que seu ancestral. Em caso afirmativo trocamos os dois de lugar.

Heap binário - Exemplo de inserção

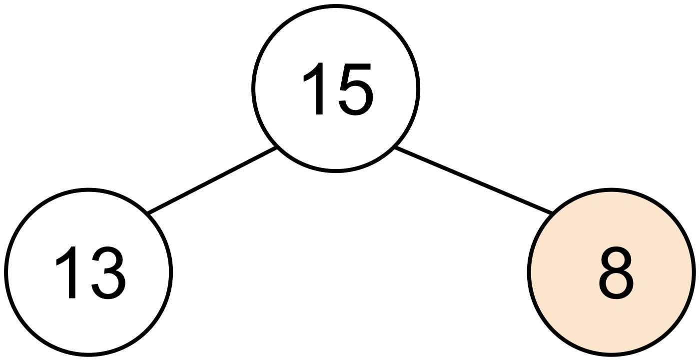


Tamanho = 2

0	1	2	3	4	5	6
15	13					

Neste caso uma troca foi o suficiente para que a árvore voltasse a ser um Heap. Mas isso nem sempre será verdade, devemos continuar verificando os ancestrais dos nós até que a propriedade seja satisfeita.

Heap binário - Exemplo de inserção

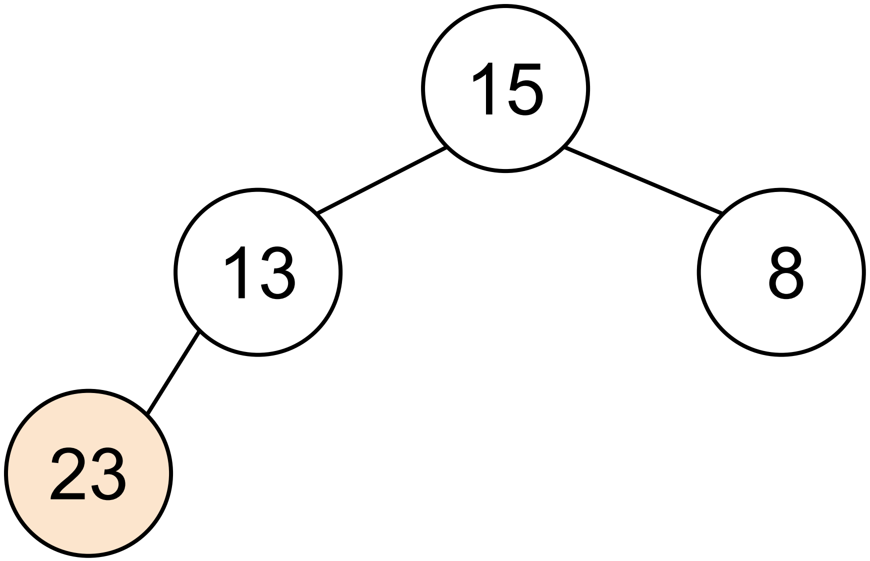


Tamanho = 3

0	1	2	3	4	5	6
15	13	8				

Agora foi inserido o valor 8. Note que após a inserção a árvore continua sendo um heap, então não precisamos consertar nada.

Heap binário - Exemplo de inserção

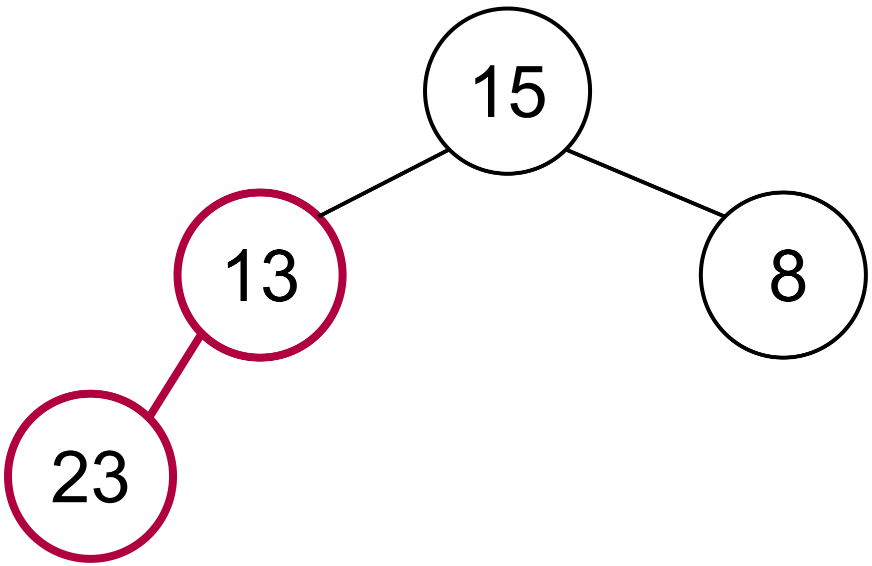


Tamanho = 4

0	1	2	3	4	5	6
15	13	8	23			

Agora foi inserido o valor 23. Note que após a inserção a árvore deixou de ser um heap, pois o 23 é sucessor do 13 mas é maior que ele. Mais uma vez precisaremos "consertar" a árvore.

Heap binário - Exemplo de inserção

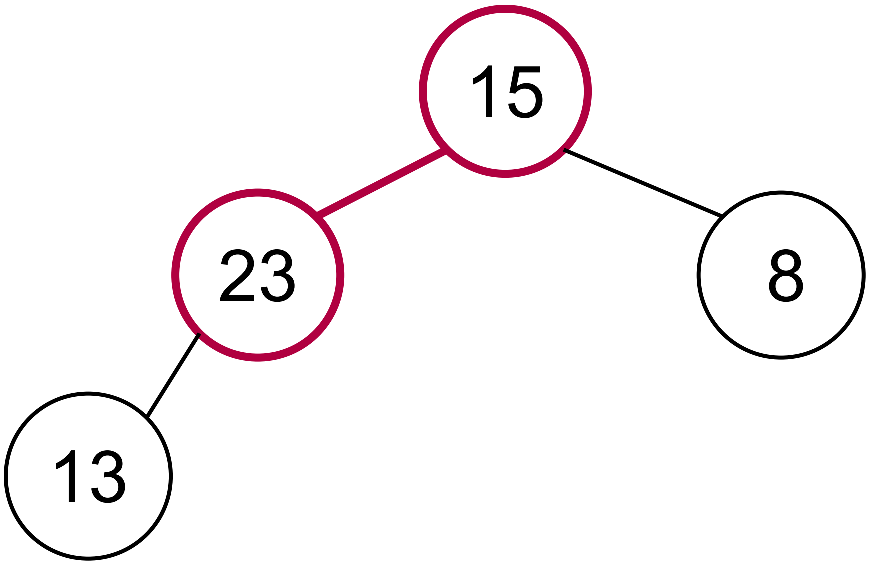


Tamanho = 4

0	1	2	3	4	5	6
15	13	8	23			

Agora foi inserido o valor 23. Note que após a inserção a árvore deixou de ser um heap, pois o 23 é sucessor do 13 mas é maior que ele. Mais uma vez precisaremos "consertar" a árvore.

Heap binário - Exemplo de inserção

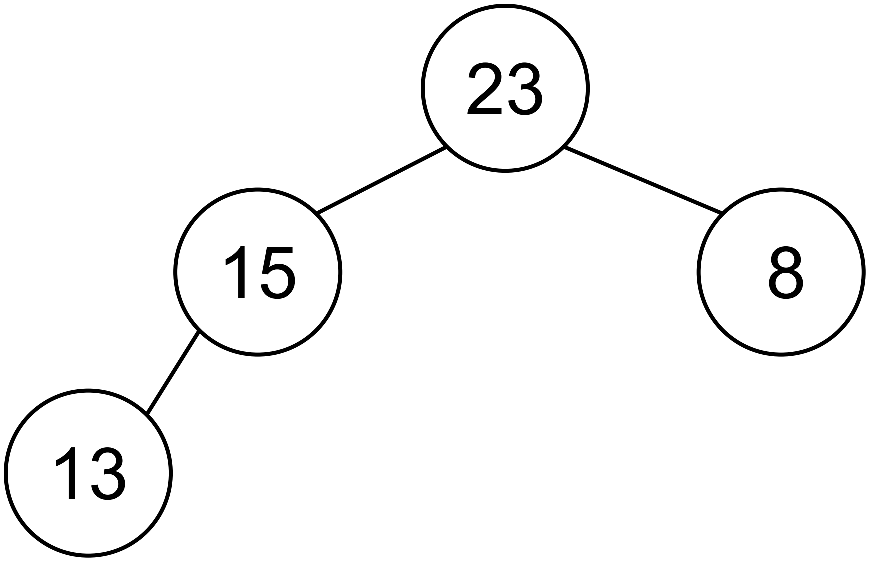


Tamanho = 4

0	1	2	3	4	5	6
15	23	8	13			

A troca foi efetuada mas desta vez não foi o suficiente. A árvore resultante ainda não é um Heap, pois o 23 é sucessor do 15 e é maior que ele. Então devemos mais uma vez “consertar” a árvore.

Heap binário - Exemplo de inserção

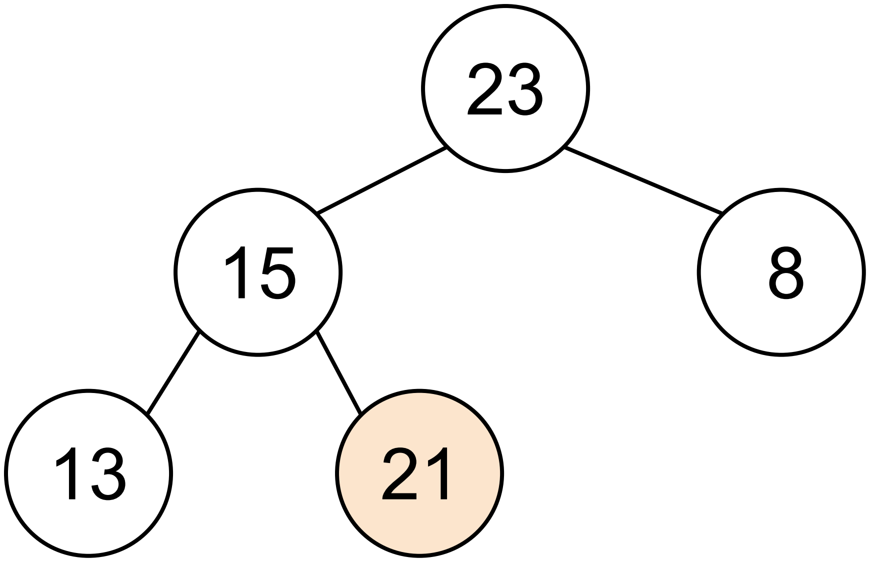


Tamanho = 4

0	1	2	3	4	5	6
23	15	8	13			

O processo de consertar a árvore após a inserção deve continuar subindo a árvore enquanto for necessário, o que assim como nesse exemplo, pode nos levar a efetuar trocas da folha até a raiz.

Heap binário - Exemplo de inserção

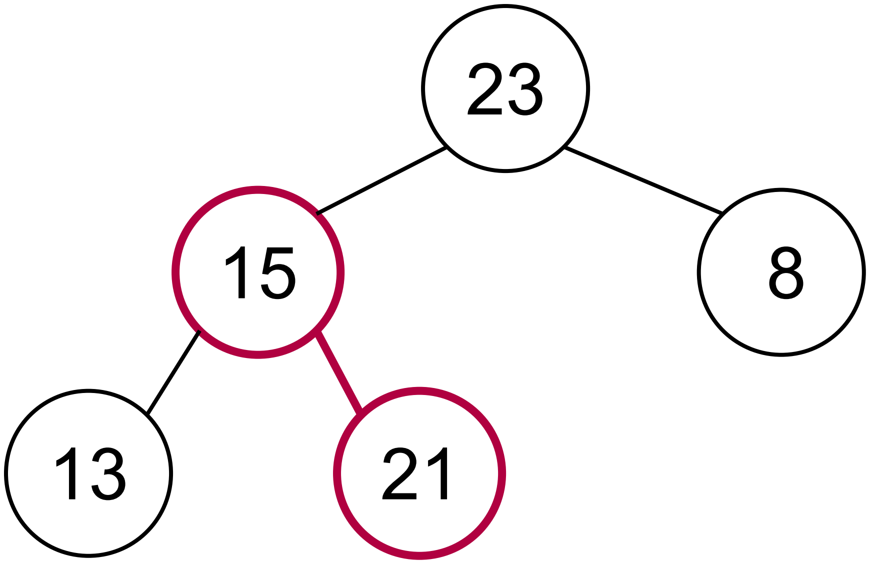


Tamanho = 5

0	1	2	3	4	5	6
23	15	8	13	21		

Mas as trocas também podem parar “no meio do caminho”. Agora o valor inserido é 21. Note que mais uma vez a propriedade foi violada.

Heap binário - Exemplo de inserção

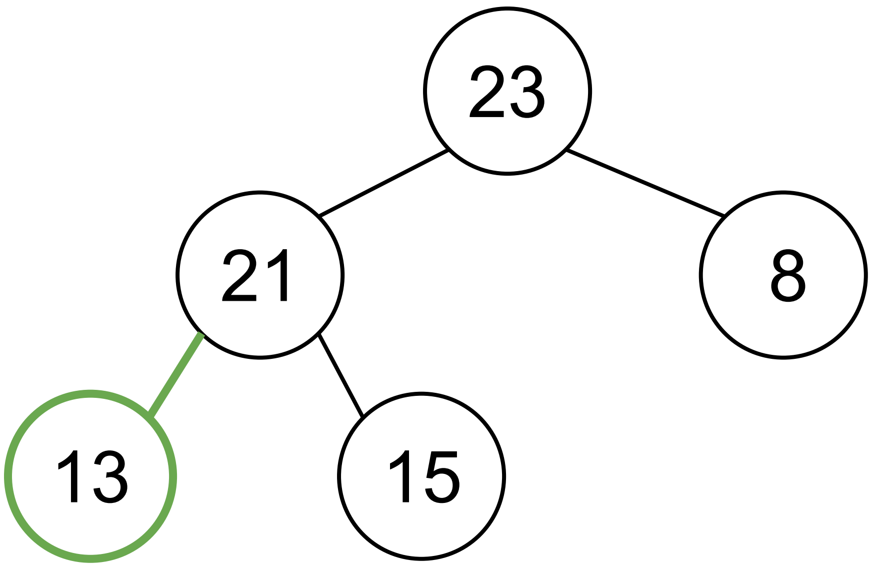


Tamanho = 5

0	1	2	3	4	5	6
23	15	8	13	21		

Mais uma vez iniciamos o processo de troca dos elementos, trocando o 21 por seu ancestral, 15.

Heap binário - Exemplo de inserção

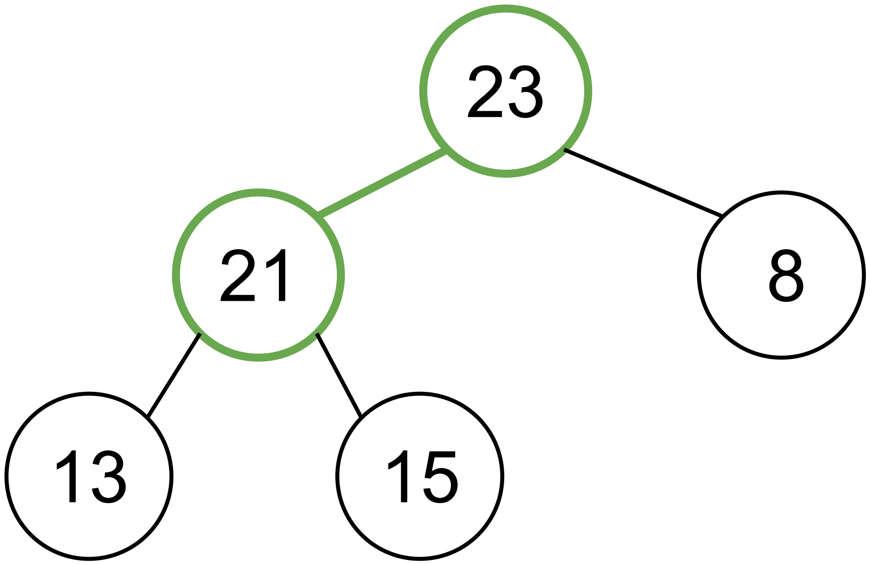


Tamanho = 5

0	1	2	3	4	5	6
23	21	8	13	15		

Note que não existe a necessidade de se preocupar com a subárvore da esquerda, afinal se o 15 nesta posição já satisfazia a propriedade, substituí-lo por um valor ainda maior não pode fazer com que a árvore deixe de ser um heap.

Heap binário - Exemplo de inserção

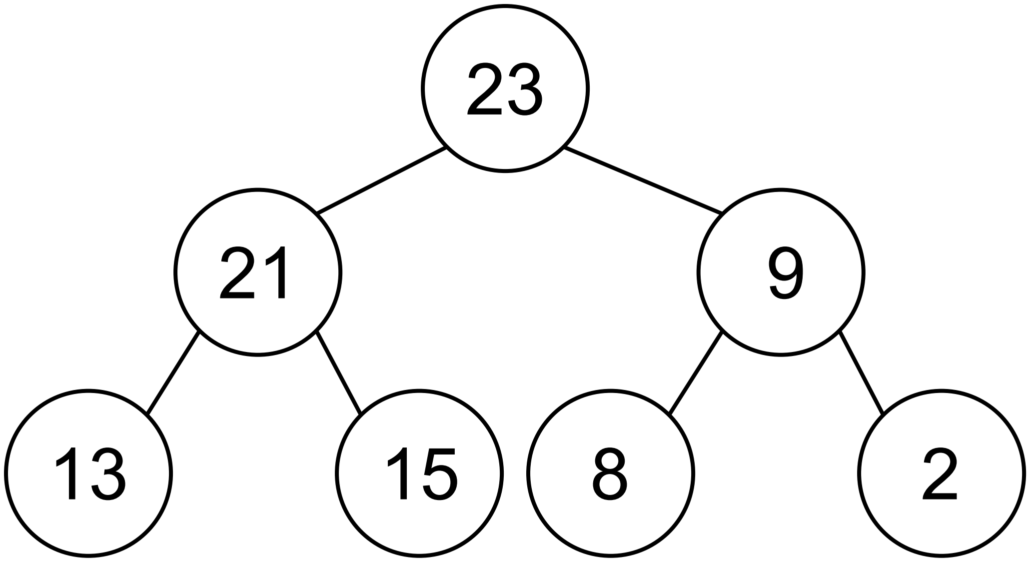


Tamanho = 5

0	1	2	3	4	5	6
23	21	8	13	15		

Continuando as comparações com os nós ancestrais, agora o 21 é menor que seu ancestral, então isso significa que não precisamos consertar mais nada na árvore, e agora ela é um Heap.

Heap binário - Exemplo de inserção



Tamanho = 7

0	1	2	3	4	5	6
23	21	9	13	15	8	2

Execute mentalmente a inserção dos valores 9 e 2 (nesta ordem) e se convença de que este é o resultado final!

Heap binário - Inserção

Seja **v** o vetor onde estamos guardando os valores, **t** um inteiro contendo quantos elementos já existem no Heap, o processo de inserção então consiste nos seguintes passos:

```
Insere(x)
   v[t] ← x
   i ← t
   p ← (i - 1)/2
   Enquanto v[i] > v[p] faça:
       Troque os valores de v[i] e v[p]
       i ← p
       p ← (i - 1)/2
   t ← t + 1
```

Exercício: Como seria a versão recursiva do algoritmo?

Heap binário - Inserção

Considerando um Heap com *n* elementos, qual a complexidade de tempo dessa operação em função de *n*?

```
Insere(x)
   v[t] ← x
   i ← t
   p ← (i - 1)/2
   Enquanto v[i] > v[p] faça:
       Troque os valores de v[i] e v[p]
       i ← p
       p ← (i - 1)/2
   t ← t + 1
```

Heap binário - Inserção

Considerando um Heap com *n* elementos, qual a complexidade de tempo dessa operação em função de *n*?

O(log *n*)

```
Insere(x)
   v[t] ← x
   i ← t
   p ← (i - 1)/2
   Enquanto v[i] > v[p] faça:
      Troque os valores de v[i] e v[p]
      i ← p
      p ← (i - 1)/2
   t ← t + 1
```

Heap binário - Inserção

Note que a escolha de simular a árvore com um vetor não foi aleatória. A forma como inserimos os nós nos garante que a profundidade das folhas irá diferir em no máximo 1. Em outras palavras a árvore estará sempre balanceada, fato que é crucial para garantirmos a complexidade de tempo O(log *n*).

Heap binário - Remoção

Nossa política de remoção será bem simples. Iremos sempre remover a raiz da árvore. Em outras palavras isso nos retornará o maior elemento presente no Heap. Mas após a remoção precisamos eleger uma nova raiz e "consertar" a árvore caso seja necessário.

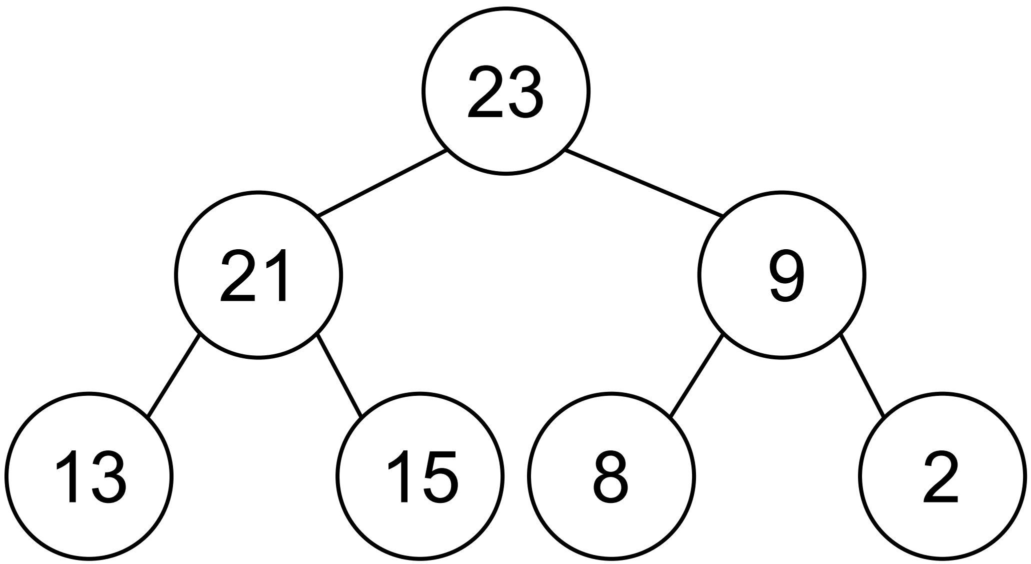
Nosso candidato a nova raiz será sempre o último elemento escrito no vetor.

Heap binário - Remoção

Dessa forma a remoção consiste em:

- Criar uma variável para receber uma cópia da raiz, que iremos retornar
- A nova raiz será o último elemento do vetor
- Isso pode quebrar nossa propriedade, então precisamos verificar se é necessário "consertar" a árvore.

Heap binário - Exemplo de remoção

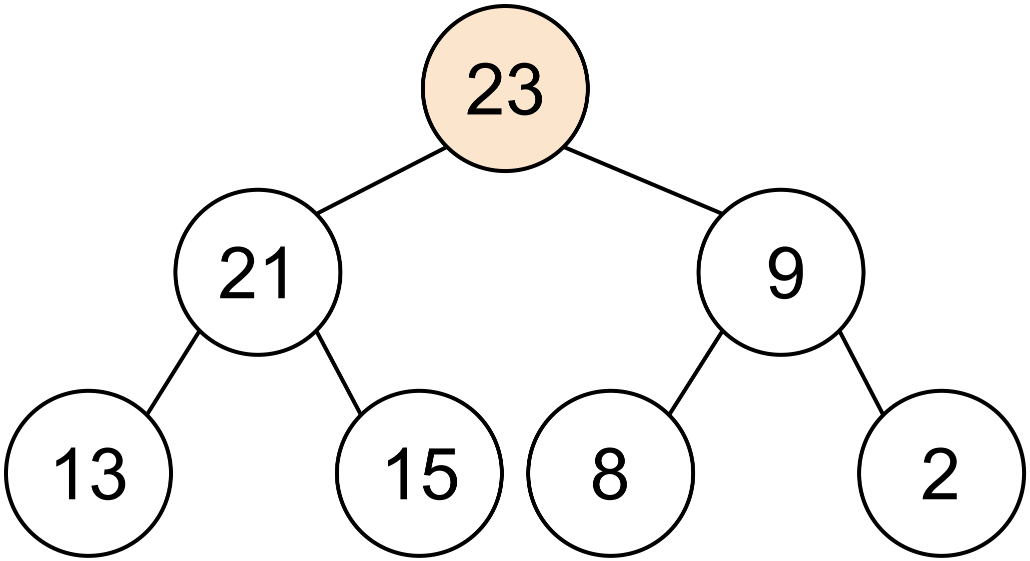


Retorno =
Tamanho = 7

0	1	2	3	4	5	6
23	21	9	13	15	8	2

Vamos executar o processo de remoção no nosso Heap de exemplo.

Heap binário - Exemplo de remoção



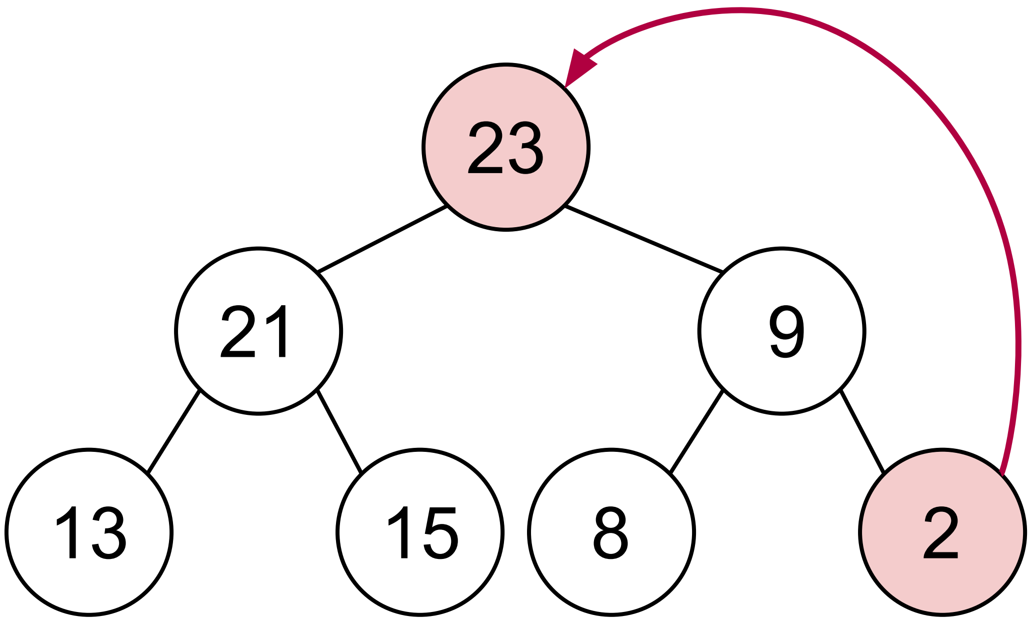
Retorno = 23

Tamanho = 7

0	1	2	3	4	5	6
23	21	9	13	15	8	2

Primeiramente criamos uma cópia de retorno da raiz.

Heap binário - Exemplo de remoção



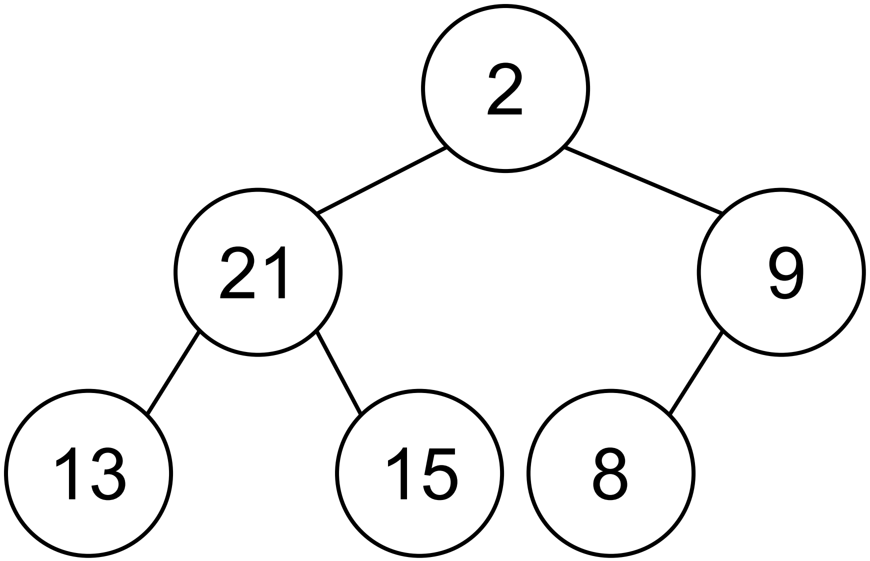
Retorno = 23

Tamanho = 7

0	1	2	3	4	5	6
23	21	9	13	15	8	2

Agora copiamos o último elemento para a raiz.
Também reduzimos o tamanho do Heap em um

Heap binário - Exemplo de remoção

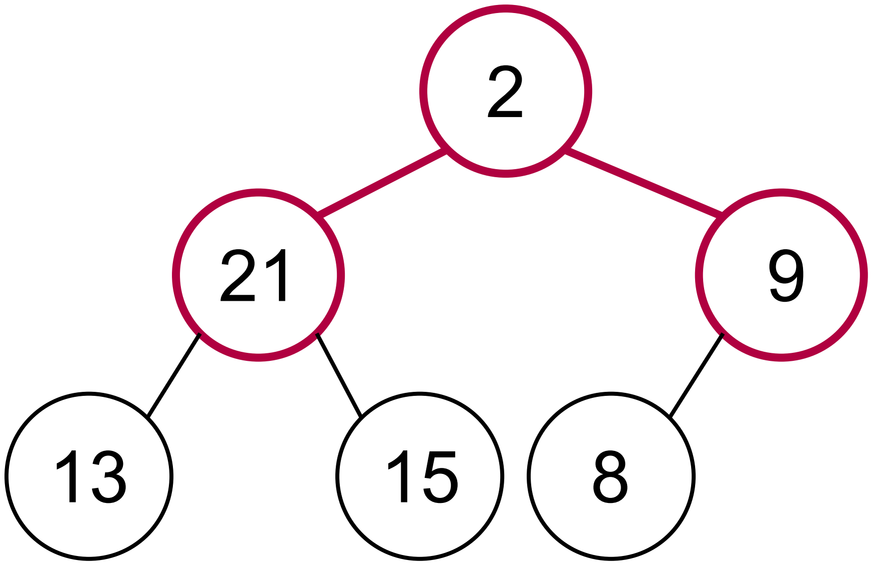


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
2	21	9	13	15	8	2

Mas agora nossa árvore não é mais um Heap. Vamos agora iniciar o processo de “consertar” a árvore.

Heap binário - Exemplo de remoção

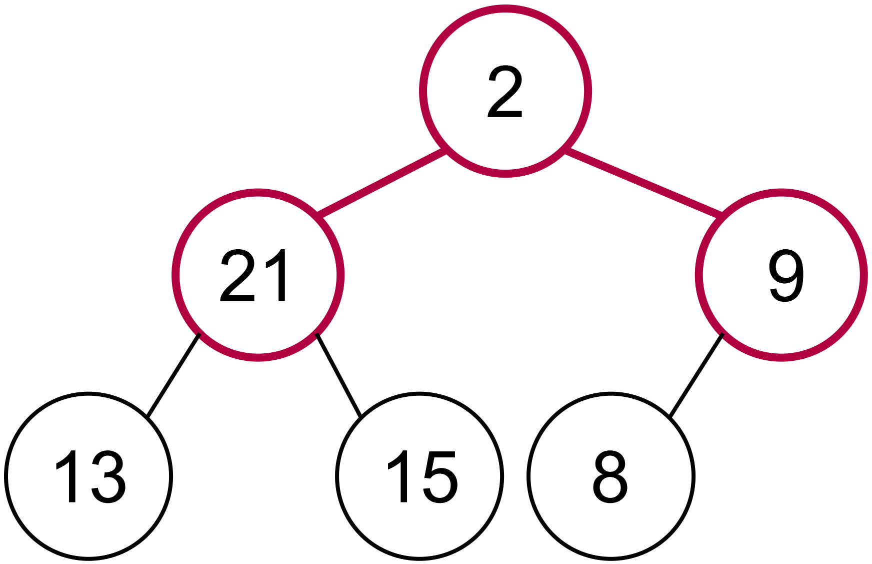


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
2	21	9	13	15	8	2

Agora o processo começa da raiz. Mas temos dois filhos, qual dos dois devemos escolher para efetuar uma troca?

Heap binário - Exemplo de remoção

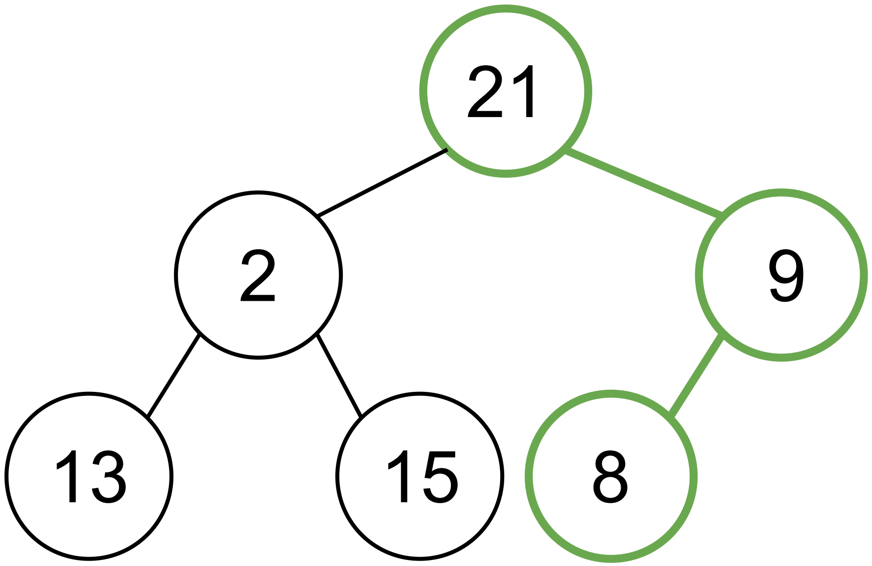


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
2	21	9	13	15	8	2

A escolha será o **maior** dos sucessores, pois dessa forma resolvemos o problema para uma das subárvores. Nesse caso trocaremos o 2 com o 21.

Heap binário - Exemplo de remoção

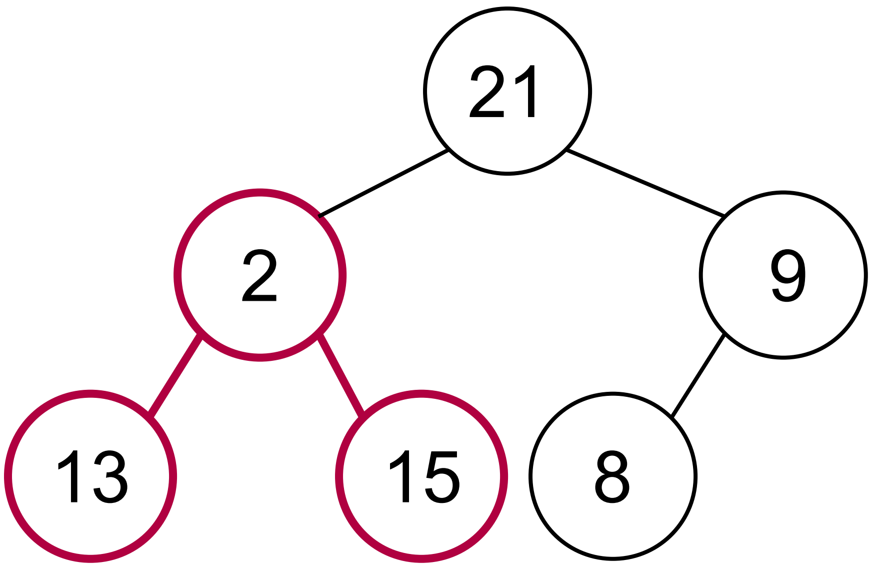


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
21	2	9	13	15	8	2

Note que agora a raiz com toda a subárvore da direita satisfazem a propriedade da Heap.

Heap binário - Exemplo de remoção

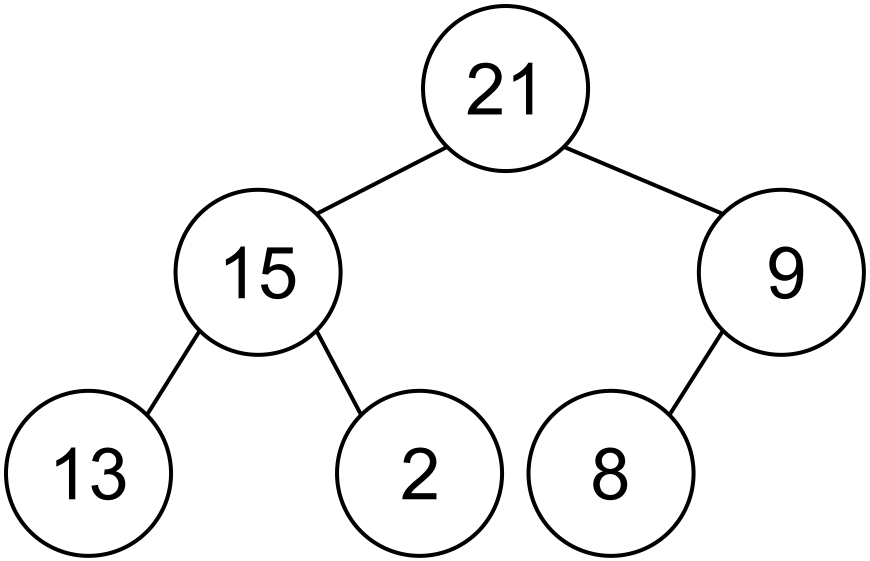


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
21	2	9	13	15	8	2

Agora basta executar o mesmo procedimento para a subárvore da esquerda.

Heap binário - Exemplo de remoção

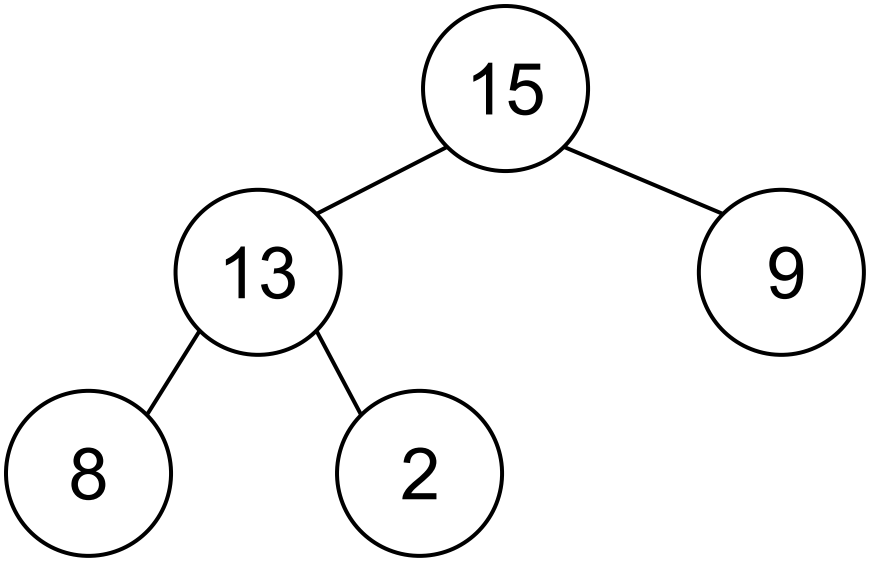


Retorno = 23
Tamanho = 6

0	1	2	3	4	5	6
21	15	9	13	2	8	2

Por fim nossa árvore é novamente um Heap. Os próximos slides contém o resultado final do processo de remoção até que o heap fique vazio.

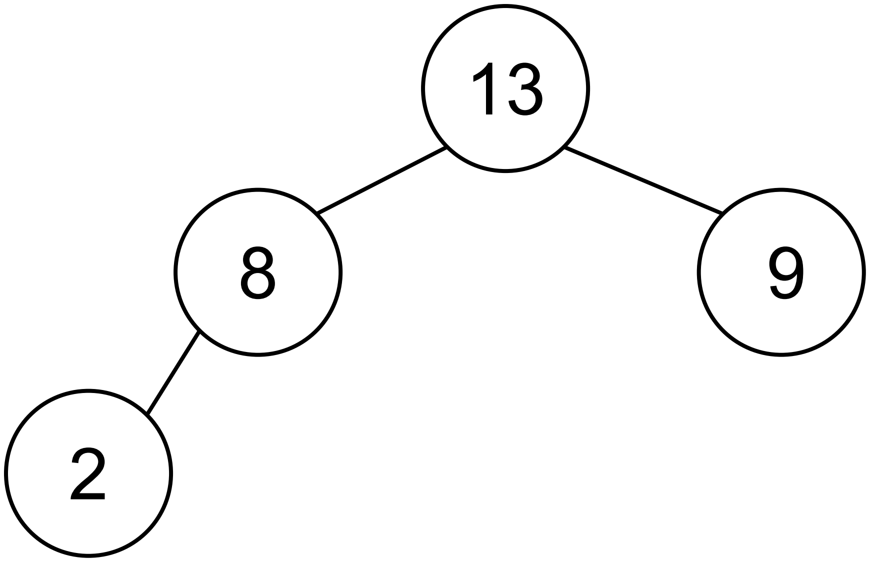
Heap binário - Exemplo de remoção



Retorno = 21
Tamanho = 5

0	1	2	3	4	5	6
15	13	9	8	2	8	2

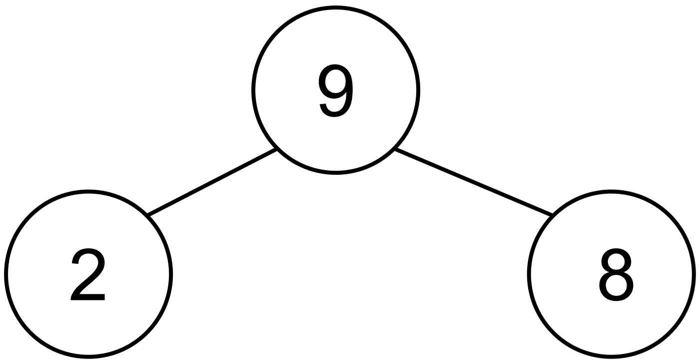
Heap binário - Exemplo de remoção



Retorno = 15
Tamanho = 4

0	1	2	3	4	5	6
13	8	9	2	2	8	2

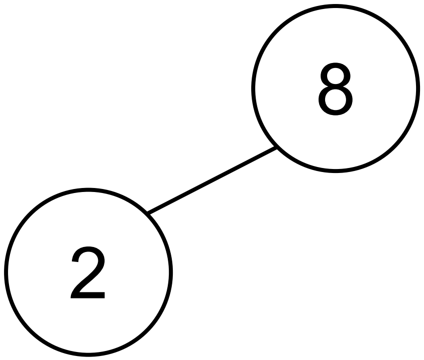
Heap binário - Exemplo de remoção



Retorno = 13
Tamanho = 3

0	1	2	3	4	5	6
9	2	8	2	2	8	2

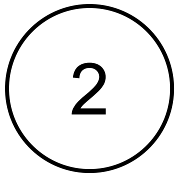
Heap binário - Exemplo de remoção



Retorno = 9
Tamanho = 2

0	1	2	3	4	5	6
8	2	8	2	2	8	2

Heap binário - Exemplo de remoção



Retorno = 8
Tamanho = 1

0	1	2	3	4	5	6
2	2	8	2	2	8	2

Heap binário - Exemplo de remoção

$\emptyset$

Retorno = 2
Tamanho = 0

0	1	2	3	4	5	6
2	2	8	2	2	8	2

Heap binário - Remoção

Seja **v** o vetor onde estamos guardando os valores, **t** um inteiro contendo quantos elementos já existem na Heap, o processo de remoção consiste nos seguintes passos:

```
Remove()
    x ← v[0]
    v[0] ← v[t-1]
    t ← t - 1
    i ← 0
    s ← índice do maior sucessor de i
    Enquanto v[i] < v[s] faça:
        Troque os valores de v[i] e v[s]
        i ← s
        s ← índice do maior sucessor de i
    Retorne x
```

Exercício: Como seria a versão recursiva do algoritmo?

Heap binário - Remoção

Considerando um Heap com *n* elementos, qual a complexidade de tempo dessa operação em função de *n*?

```
Remove()
   x ← v[0]
   v[0] ← v[t-1]
   t ← t - 1
   i ← 0
   s ← índice do maior sucessor de i
   Enquanto v[i] < v[s] faça:
       Troque os valores de v[i] e v[s]
       i ← s
       s ← índice do maior sucessor de i
   Retorne x
```

Heap binário - Remoção

Considerando um Heap com *n* elementos, qual a complexidade de tempo dessa operação em função de *n*?

O(log *n*)

```
Remove()
   x ← v[0]
   v[0] ← v[t-1]
   t ← t - 1
   i ← 0
   s ← índice do maior sucessor de i
   Enquanto v[i] < v[s] faça:
      Troque os valores de v[i] e v[s]
      i ← s
      s ← índice do maior sucessor de i
   Retorne x
```

Heap binário - Min/Max Heap

Quando a propriedade do Heap é a de que dado um nó seu valor é sempre maior ou igual que o de seus sucessores chamamos essa estrutura de **Max Heap**, pois o elemento contido na raiz é o **máximo** dos elementos no Heap.

Exercício: desenvolver um **Min Heap**, que é extremamente parecido com o que vimos, mas agora a propriedade que temos que satisfazer é:

O valor gravado em um nó é sempre **menor** ou igual ao valor gravado em seus sucessores.

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Heapsort

Professores: Anisio Lacerda
Lucas Ferreira
Wagner Meira Jr.
Washington Cunha

Heapsort

Vamos utilizar a ideia para ordenar um vetor. Se quiséssemos ordena-lo de forma crescente poderíamos apenas inserir todos os elementos em um min-heap e depois removê-los.

Mas vamos ver uma forma de fazer isso no próprio vetor, sem a necessidade de uma estrutura adicional.

Heapsort - Exemplo

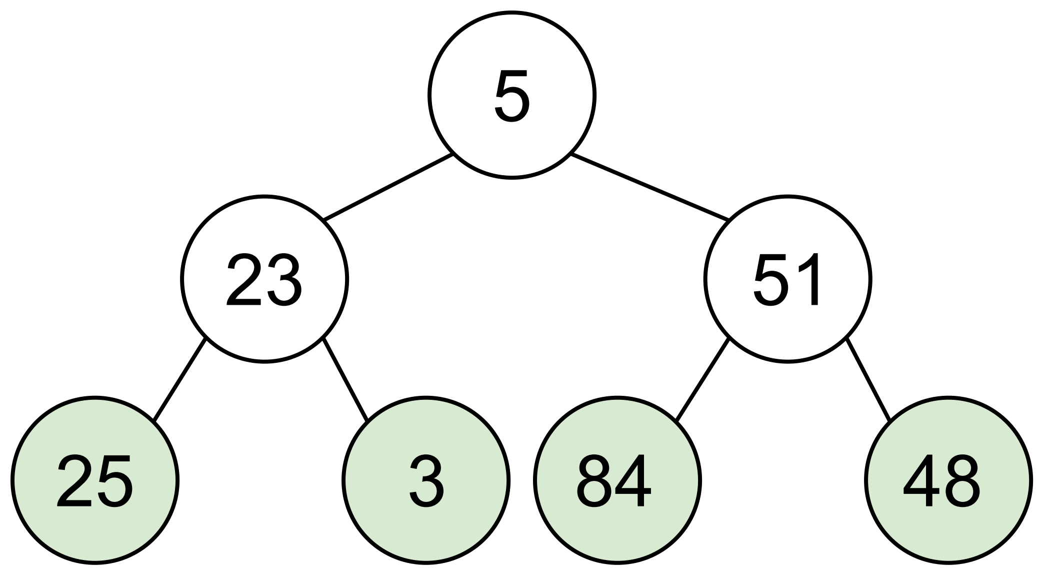
Nosso primeiro objetivo será garantir que o vetor satisfaça as propriedades de um *maxheap*. Considere o seguinte vetor:

0	1	2	3	4	5	6
5	23	51	25	3	84	48

Heapsort - Exemplo

Note que ao representarmos como uma árvore, segunda metade dos elementos são exatamente as folhas, e por construção elas já satisfazem a propriedade do *maxheap*.

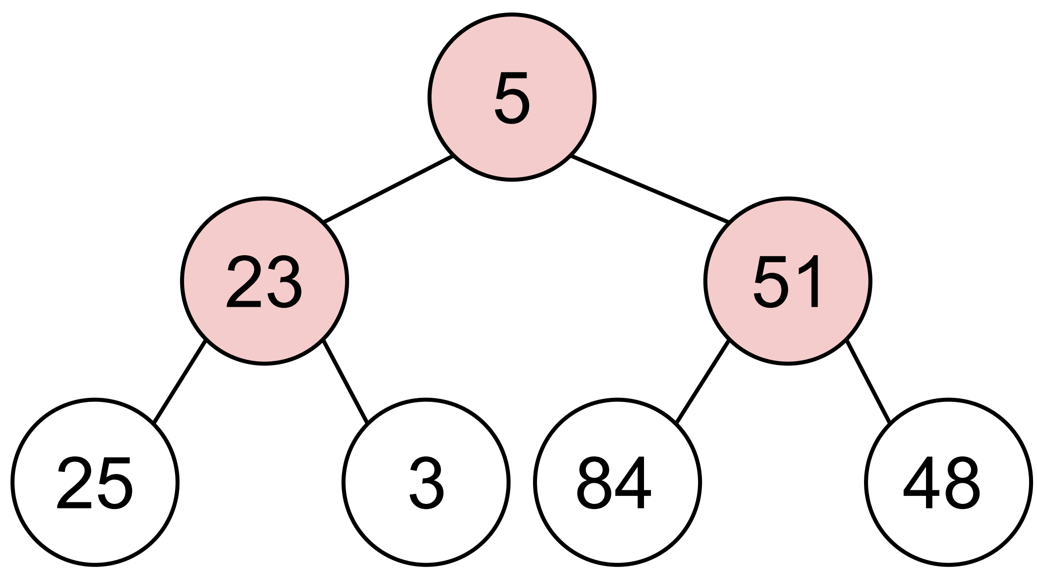
0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

Logo se algum elemento quebra a propriedade, então ele está na primeira metade.

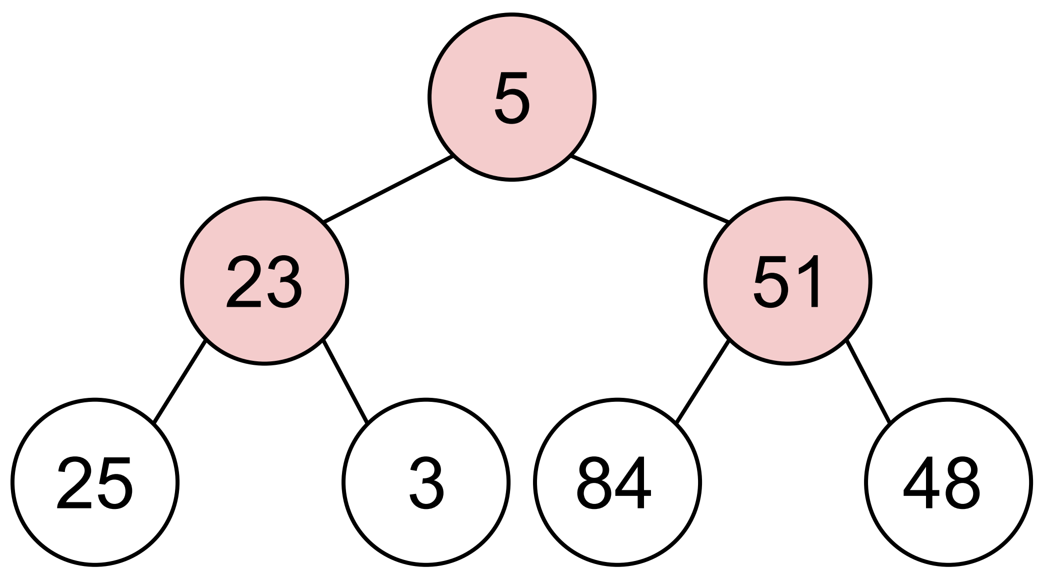
0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

Portanto, iremos verificar as subárvores enraizadas por cada um dos elementos da primeira metade, começando do mais a direita.

0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

Para “consertar” uma subárvore vamos fazer um procedimento semelhante ao quando removemos um elemento do *heap*. Seja **v** o vetor onde estamos guardando os valores, **n** um inteiro indicando o tamanho do vetor e **raiz** o índice da raiz da subárvore.

```
VerificaSubarvore(v, n, raiz)
   i ← raiz
   s ← índice do maior sucessor de i
   Enquanto v[i] < v[s] e s < n faça:
       Troque os valores de v[i] e v[s]
       i ← s
       s ← índice do maior sucessor de i
```

Heapsort - Exemplo

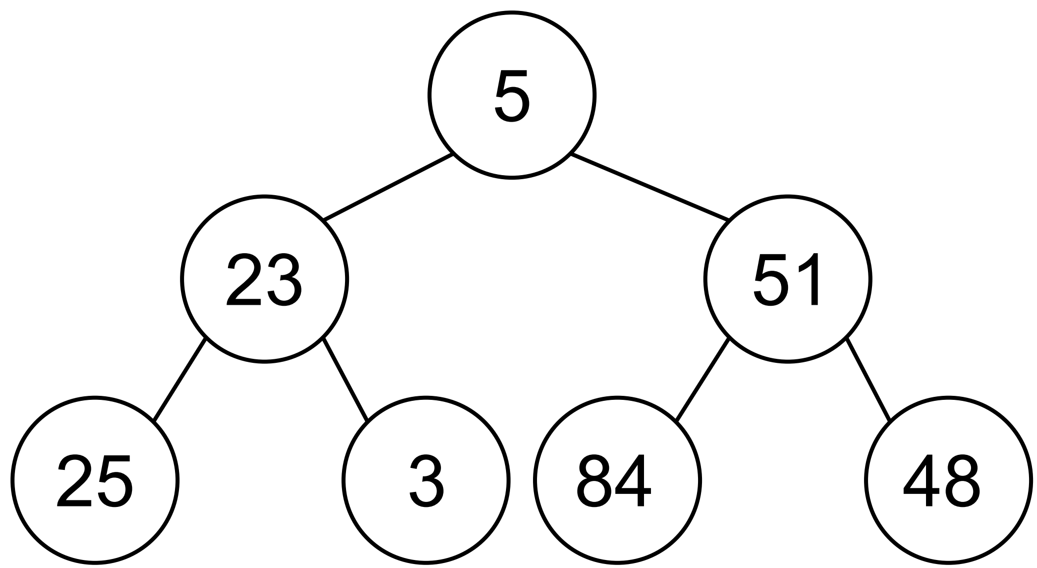
Por fim como dito anteriormente, temos que verificar todas as subárvores enraizadas nos elementos da primeira metade do vetor.

```
ConstroiHeap(v, n)
   i ← n / 2
   Enquanto i > 0 faça:
       i ← i - 1
       VerificaSubarvore(v, n, i)
```

Heapsort - Exemplo

Vamos executar o algoritmo no nosso exemplo.

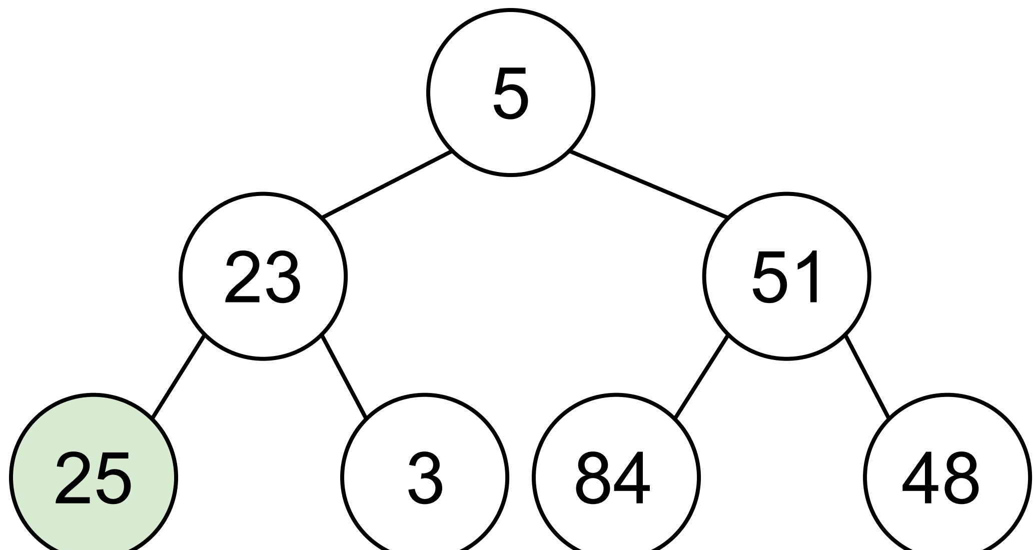
0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

O elemento na posição n/2 é a “primeira” das folhas, portanto queremos um elemento antes dele.

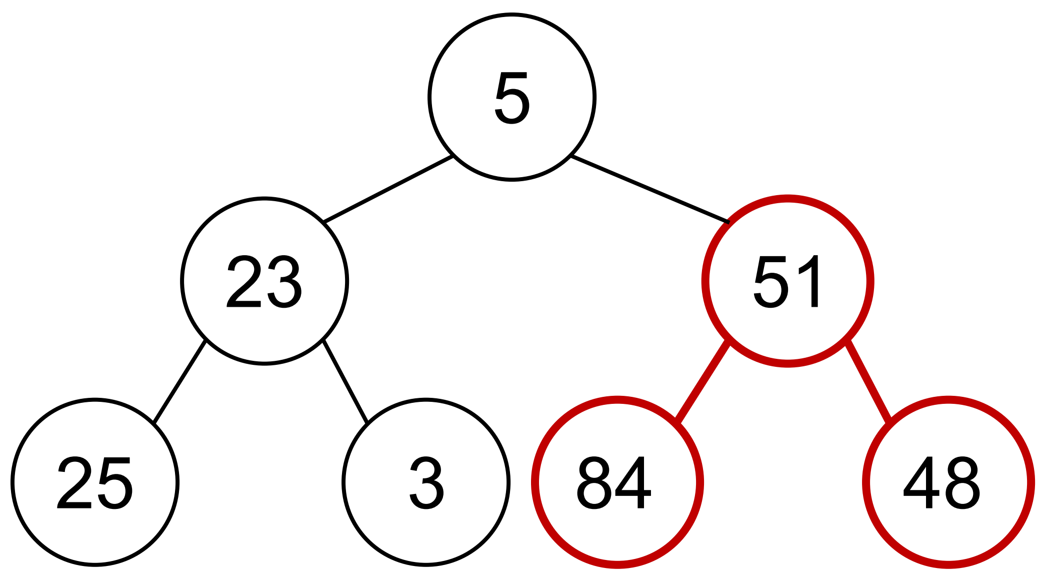
0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

Esta é a primeira árvore que iremos verificar. Para isso comparamos a raiz com o maior dos filhos.

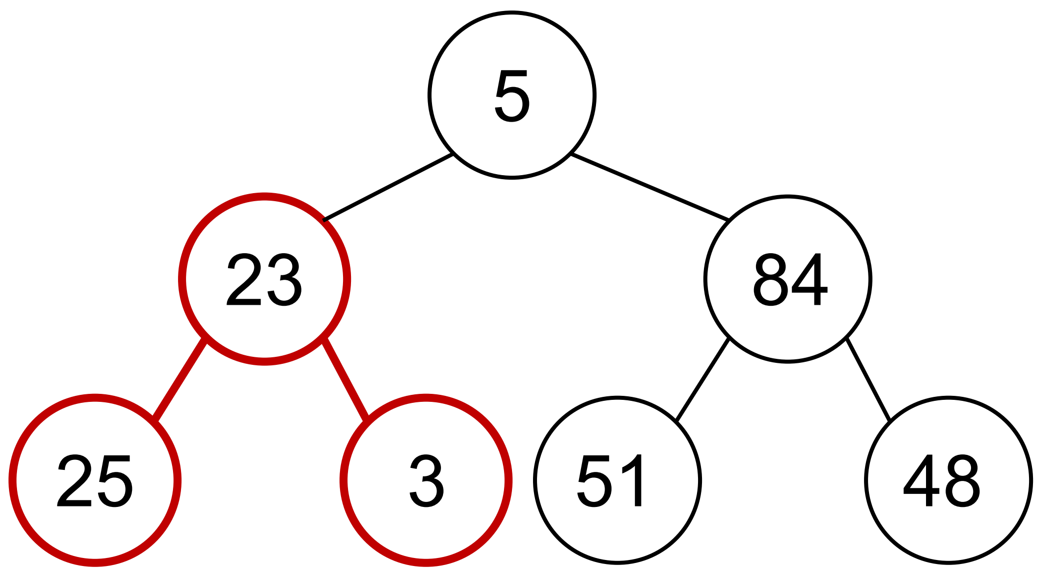
0	1	2	3	4	5	6
5	23	51	25	3	84	48



Heapsort - Exemplo

Seguindo com o algoritmo vamos verificar a subárvore enraizada pelo elemento na posição 1.

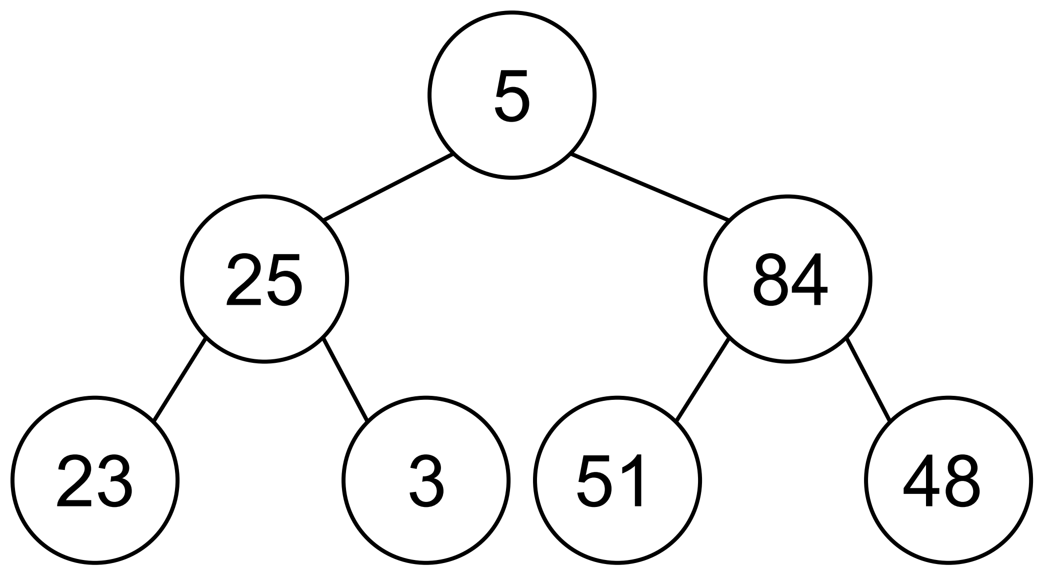
0	1	2	3	4	5	6
5	23	84	25	3	51	48



Heapsort - Exemplo

Por fim, vamos verificar a árvore toda, e após essa verificação ela irá satisfazer a propriedade de um *maxheap*.

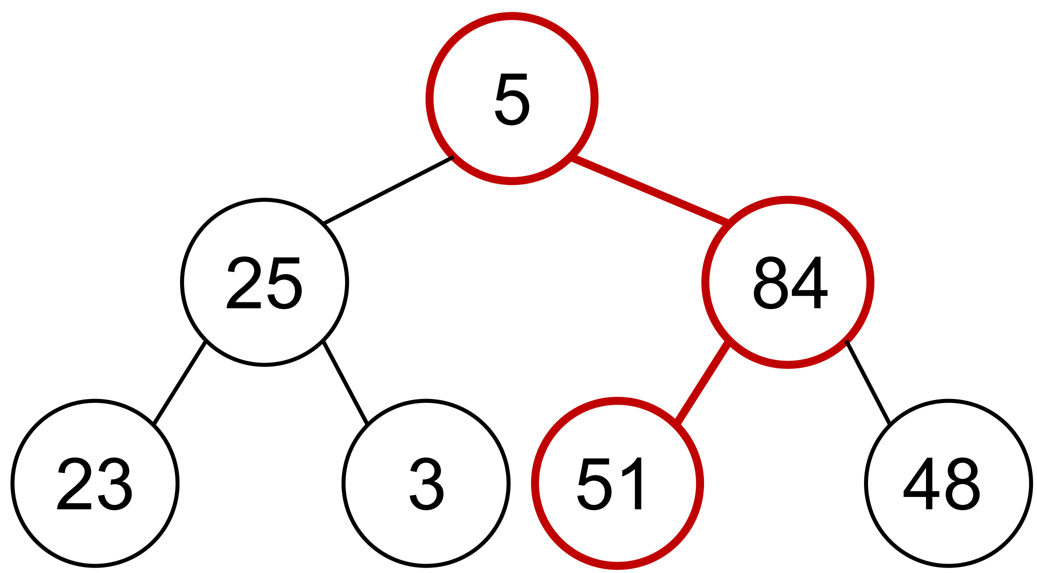
0	1	2	3	4	5	6
5	25	84	23	3	51	48



Heapsort - Exemplo

As trocas realizadas nesse passo estão destacadas no caminho em vermelho.

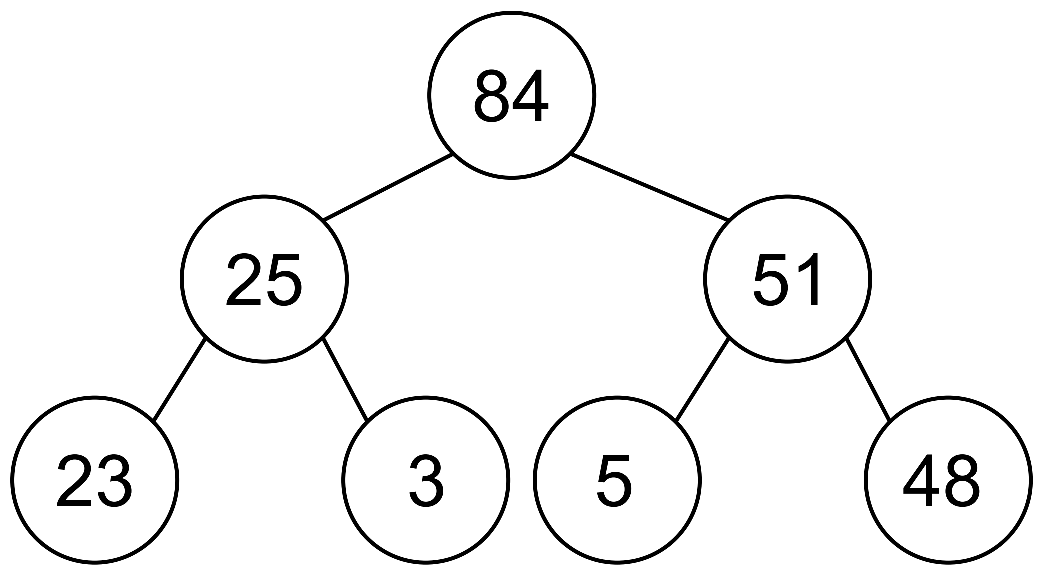
0	1	2	3	4	5	6
5	25	84	23	3	51	48



Heapsort - Exemplo

Ao final deste processo temos um *maxheap*. Mas como ele pode nos ajudar a ordenar o vetor de forma crescente?

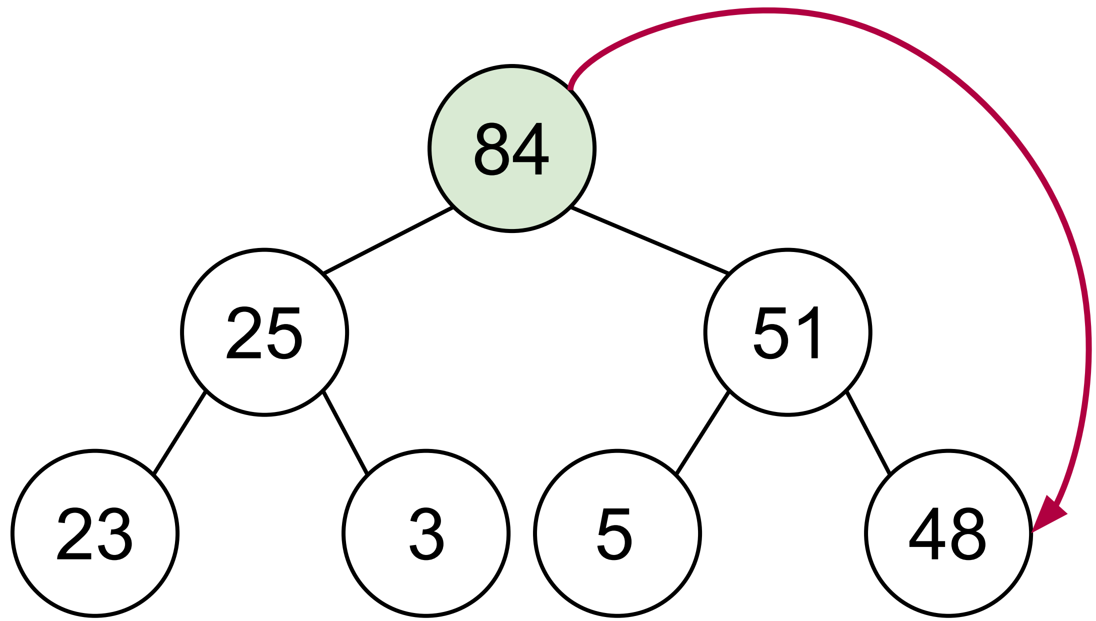
0	1	2	3	4	5	6
84	25	51	23	3	5	48



Heapsort - Exemplo

Sabemos que o elemento da raiz é o maior de todos, e que portanto ele deverá estar no final do vetor ordenado. Vamos então trocar ele de posição com o último elemento.

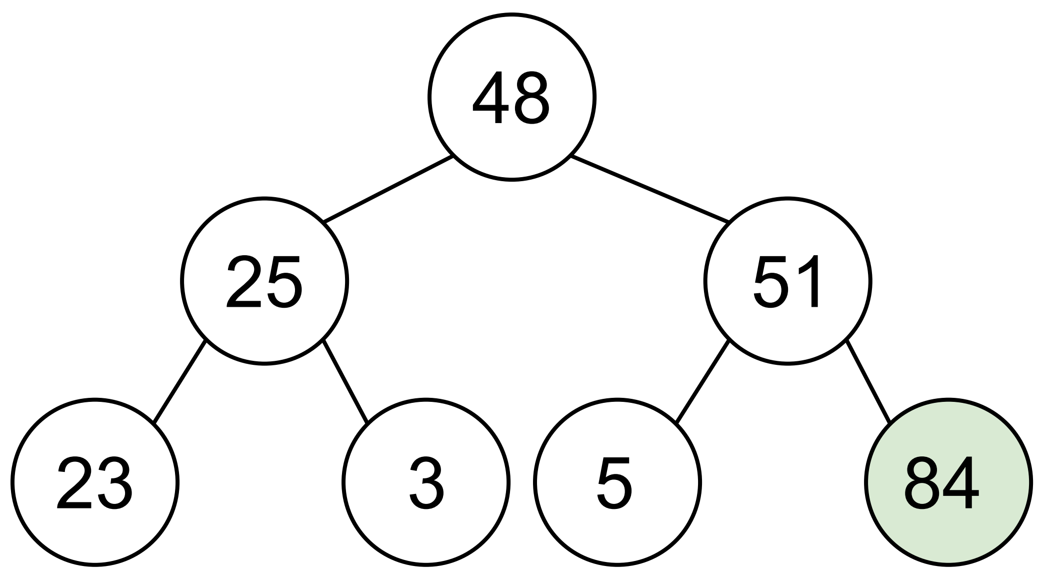
0	1	2	3	4	5	6
84	25	51	23	3	5	48



Heapsort - Exemplo

Agora o elemento 84 está na posição correta, mas nossa estrutura não é mais um *maxheap*.

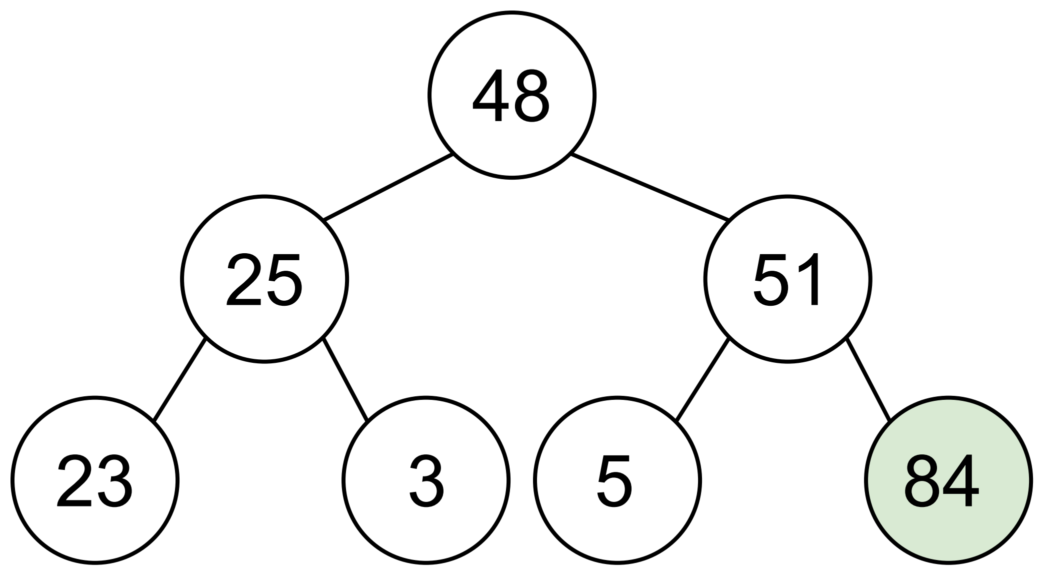
0	1	2	3	4	5	6
48	25	51	23	3	5	84



Heapsort - Exemplo

Vamos simplesmente considerar que o 84 não faz mais parte do *heap* e verificar a propriedade a partir da raiz, mas para um *heap* de tamanho *n-1*.

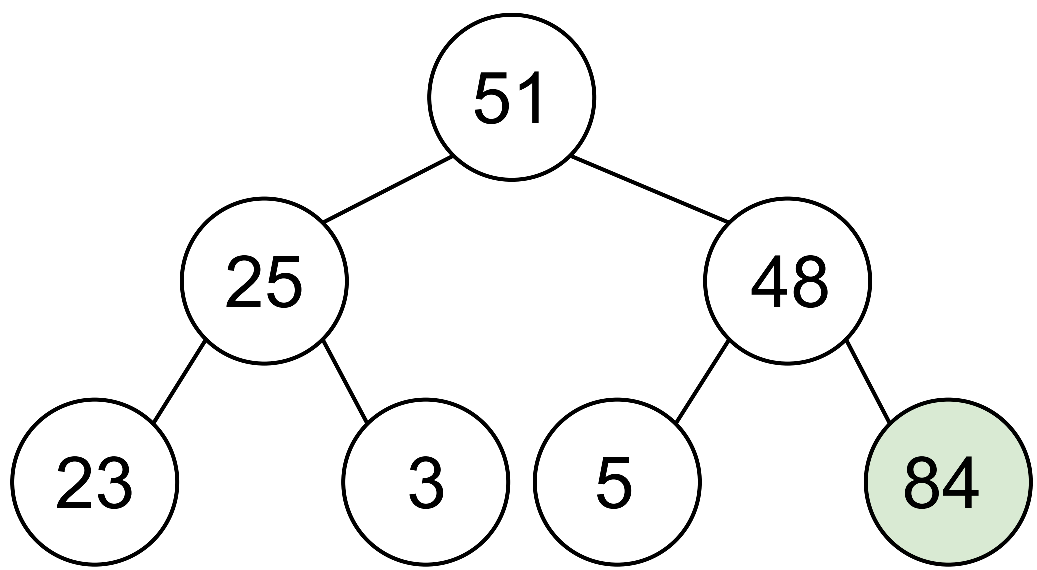
0	1	2	3	4	5	6
48	25	51	23	3	5	84



Heapsort - Exemplo

Temos então um *maxheap* de tamanho *n-1*.Basta repetir o processo até que o *heap* fique vazio.

0	1	2	3	4	5	6
51	25	48	23	3	5	84



Heapsort - Exemplo

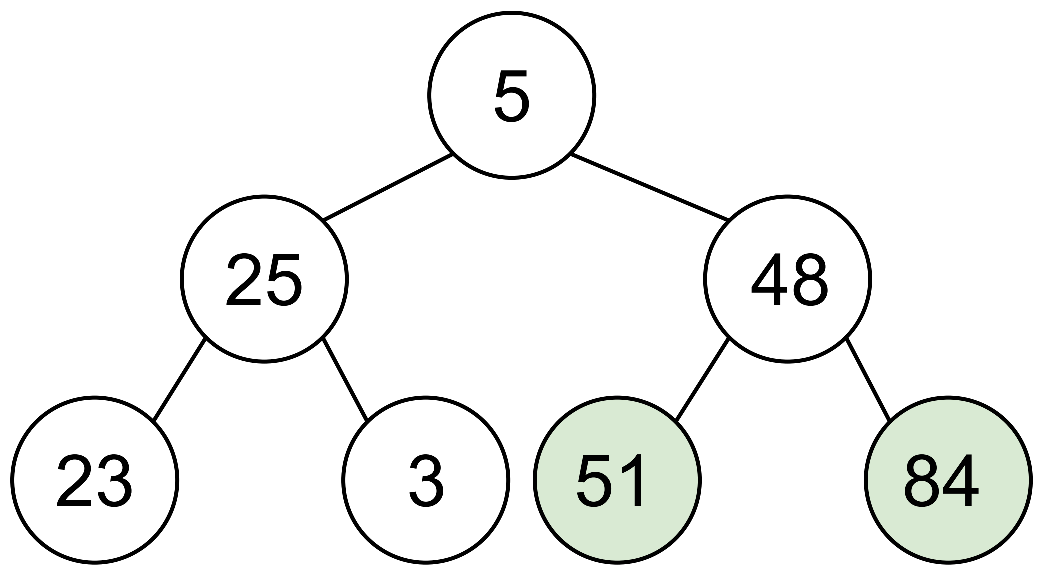
Então nosso algoritmo do heapsort segue:

```
HeapSort(v, n)
    ConstroiHeap(v,n)
    t ← n - 1
    Enquanto t > 0 faça:
        aux ← v[t]
        v[t] ← v[0]
        v[0] ← aux
        t ← t - 1
        VerificaSubarvore(v, t, 0)
```

Heapsort - Exemplo

Executando o algoritmo no exemplo:

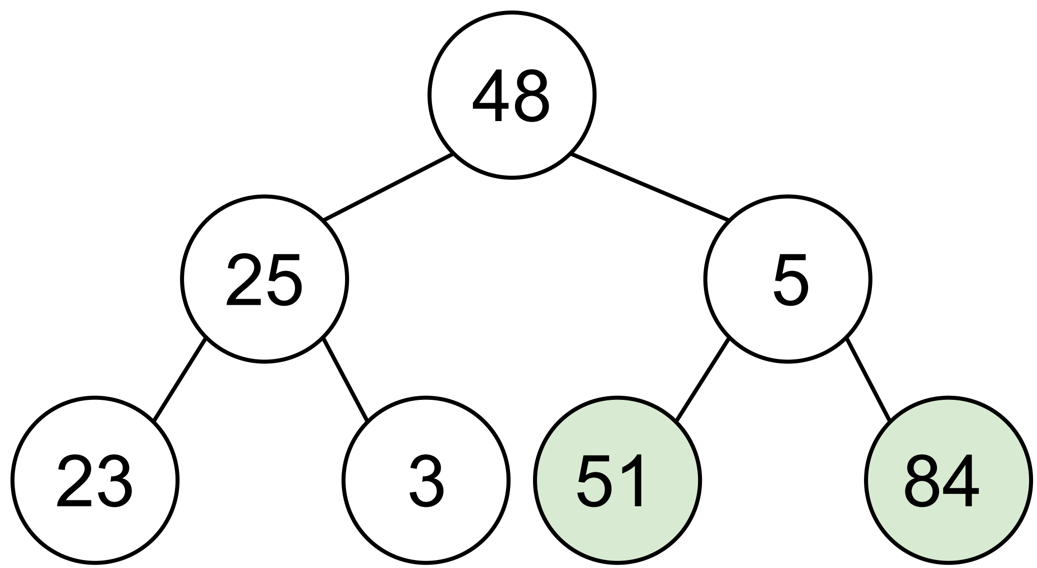
0	1	2	3	4	5	6
5	25	48	23	3	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

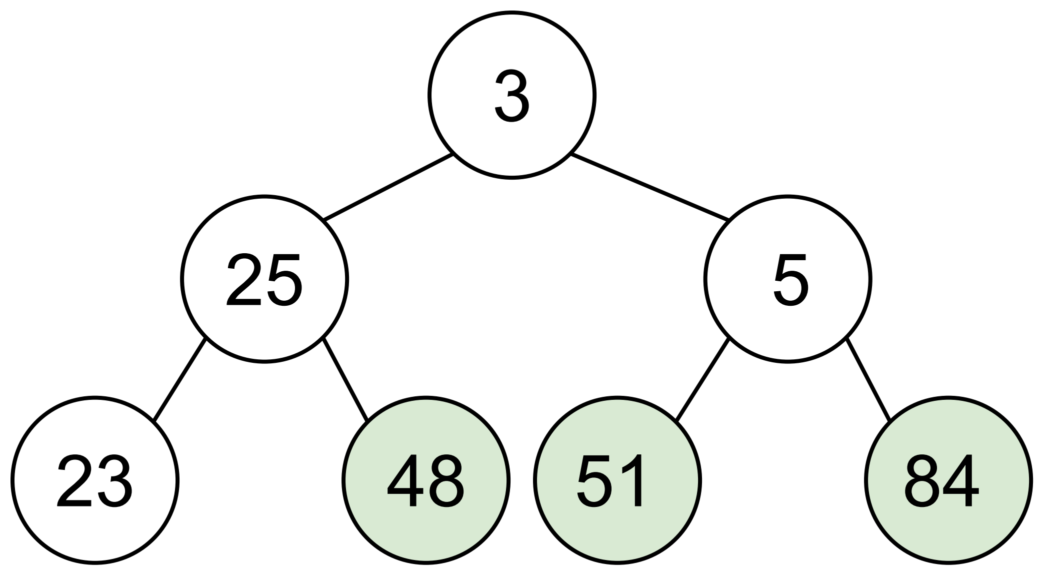
0	1	2	3	4	5	6
48	25	5	23	3	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

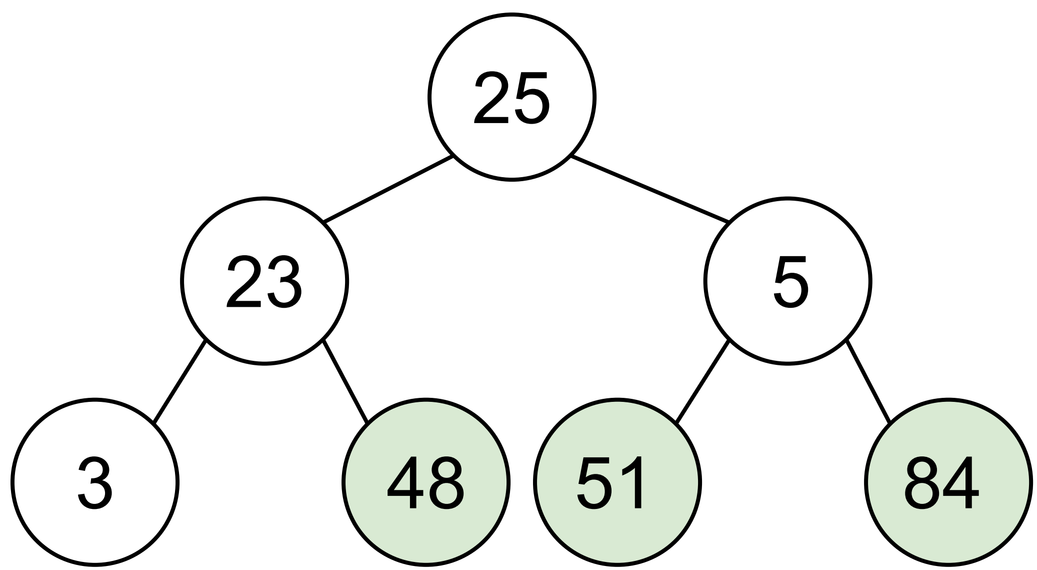
0	1	2	3	4	5	6
3	25	5	23	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

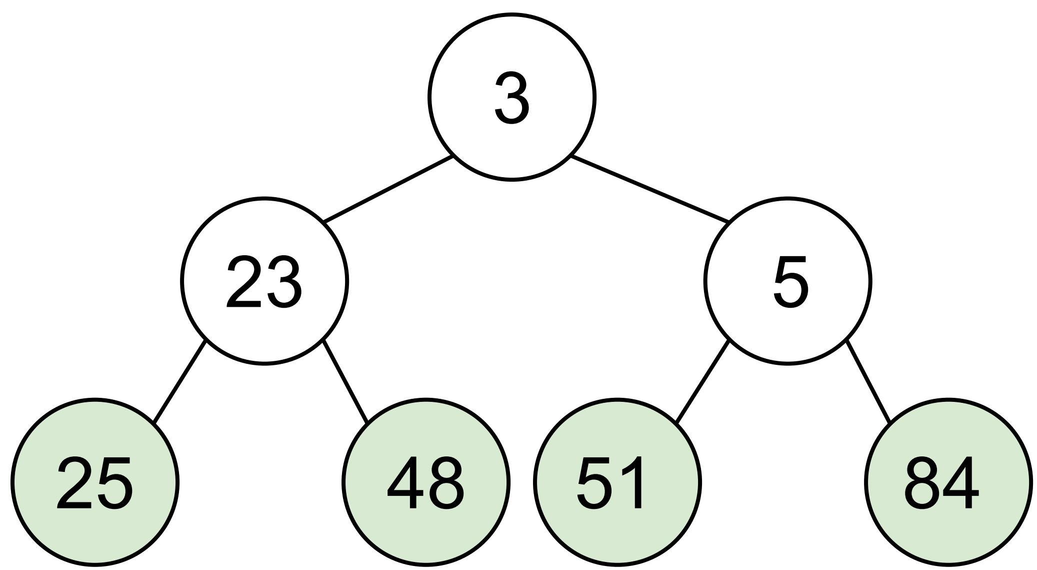
0	1	2	3	4	5	6
25	23	5	3	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

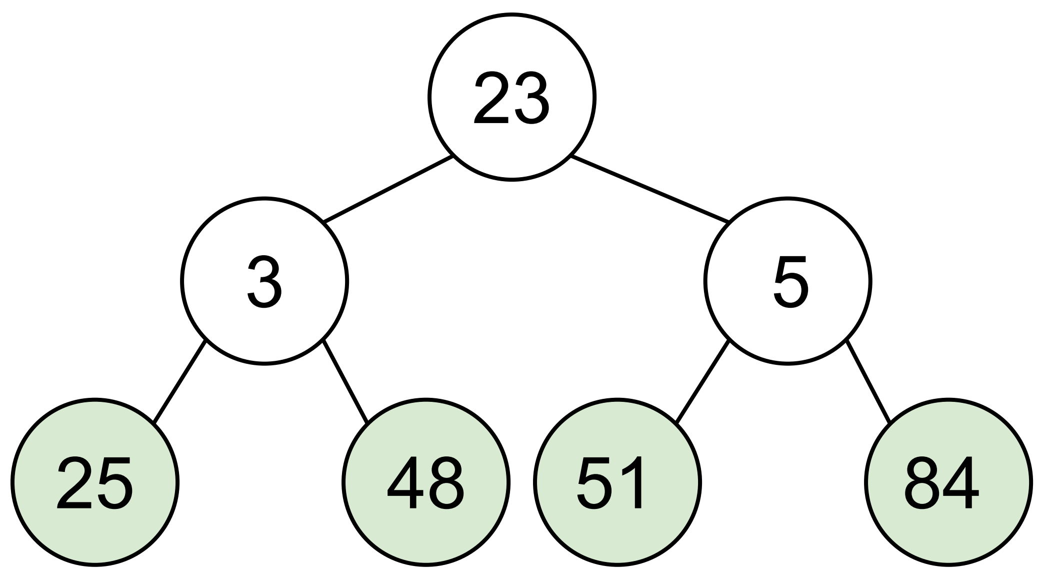
0	1	2	3	4	5	6
3	23	5	25	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

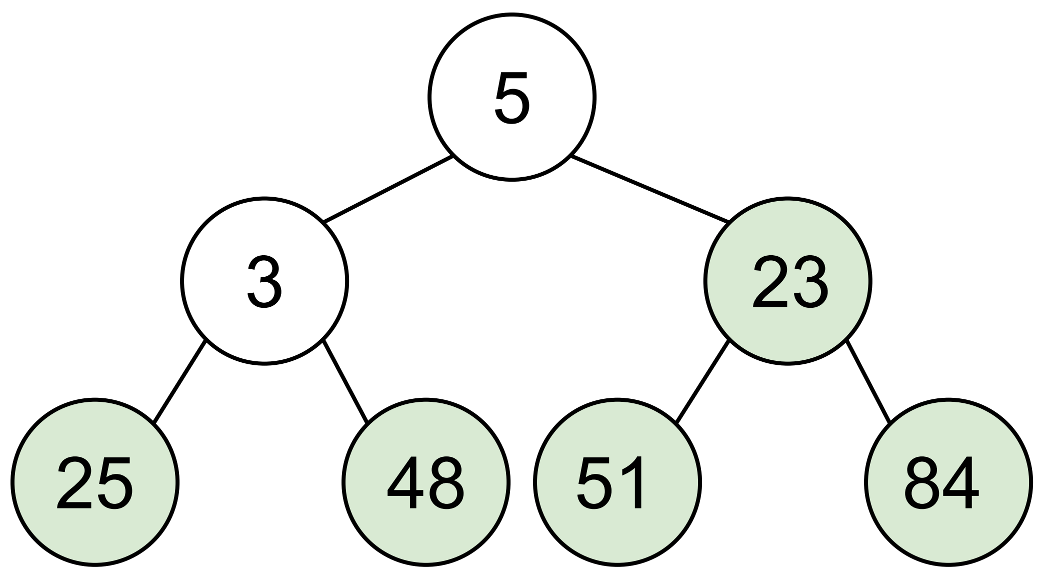
0	1	2	3	4	5	6
23	3	5	25	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

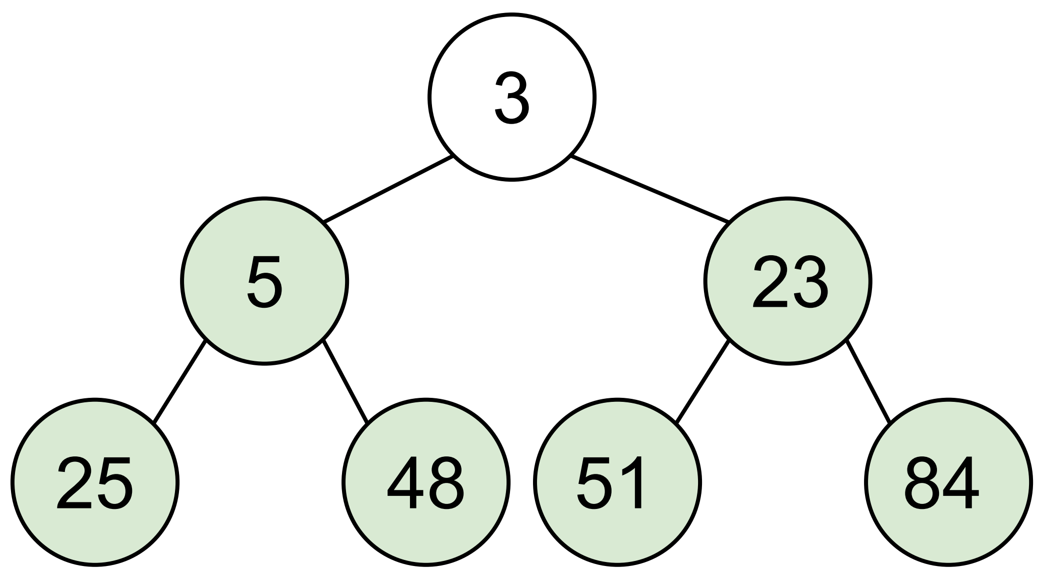
0	1	2	3	4	5	6
5	3	23	25	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

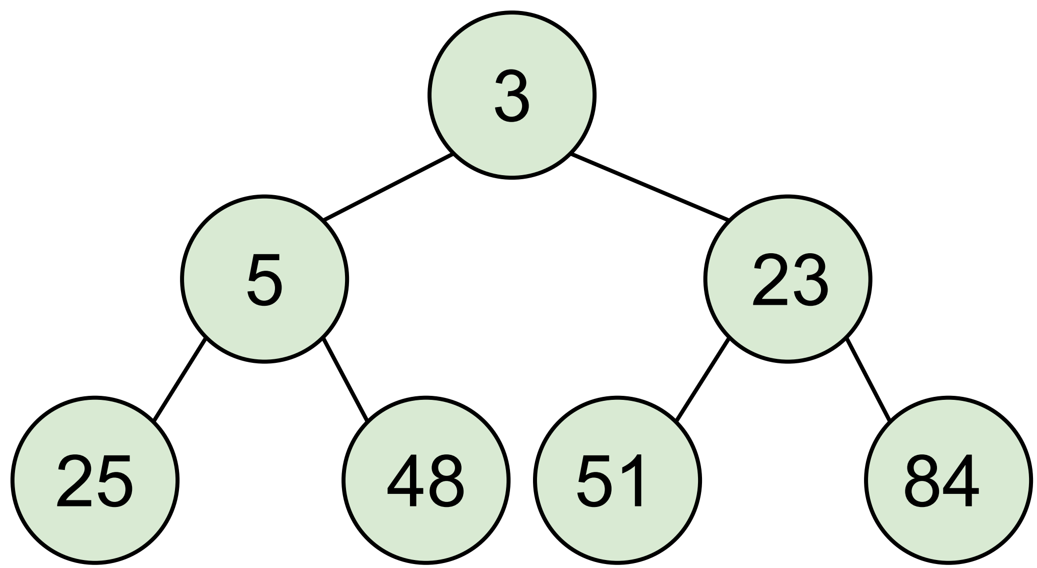
0	1	2	3	4	5	6
3	5	23	25	48	51	84



Heapsort - Exemplo

Executando o algoritmo no exemplo:

0	1	2	3	4	5	6
3	5	23	25	48	51	84



DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

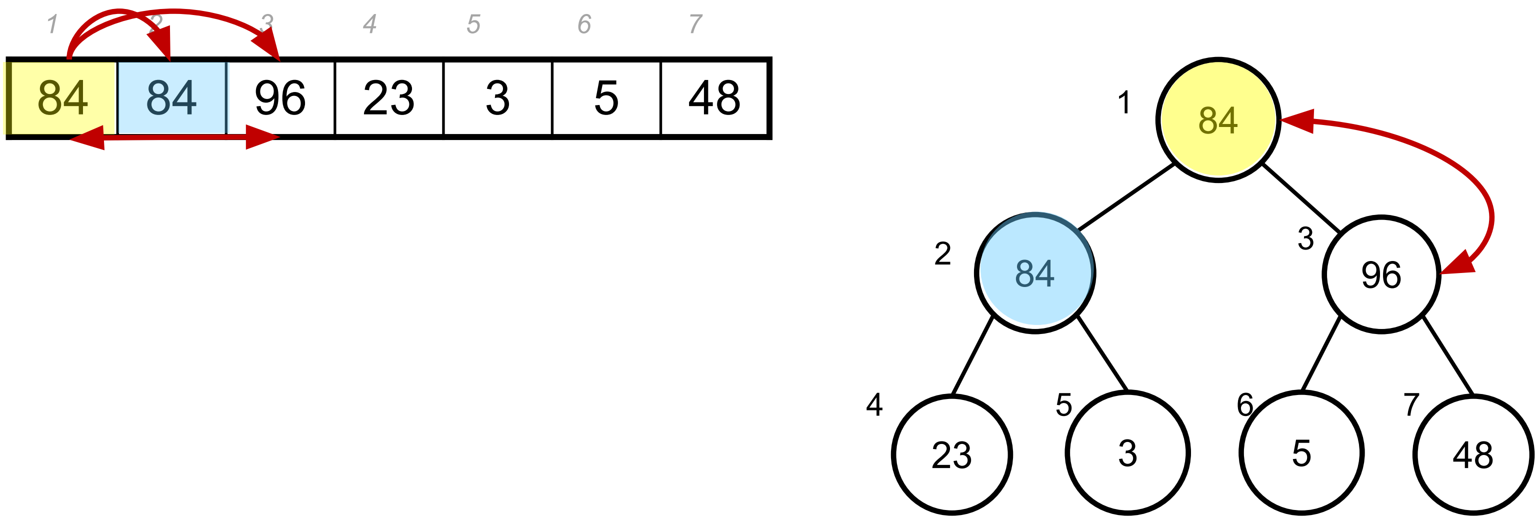
UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Heapsort - Análise

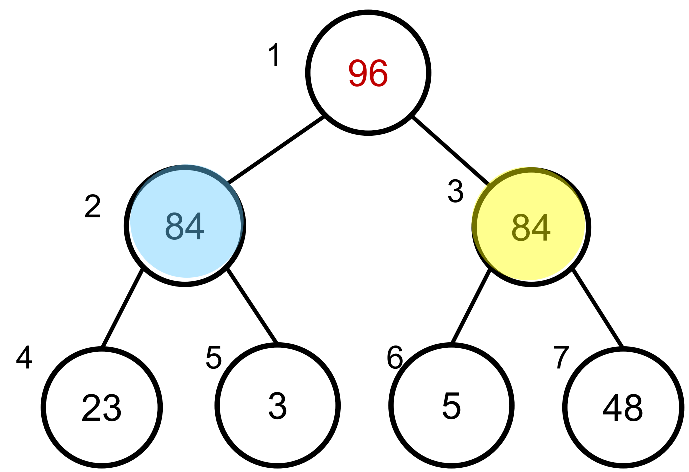
Professores: Anisio Lacerda
Lucas Ferreira
Wagner Meira Jr.
Washington Cunha

O método é estável?



O método é estável?

1	2	3	4	5	6	7
96	84	84	23	3	5	48



Inverteu a posição dos '84's

NÃO É ESTÁVEL!

Heapsort – Análise de Complexidade

- VerificaSubarvore:
  - No pior caso, percorre todo um galho da árvore binária, ou seja, executa log n operações ➔ ***C(n) = O(log n)***
- ConstroiHeap:
  - Para os nós internos (n/2 elementos), chama refaz, logo executa: n/2 log n ➔ ***C(n) = O(n log n)***
- Heapsort
  - Chama Constroi – uma vez
  - Chama VerificaSubarvore n-1 vezes

➔ ***C(n) = O(n log n)***

Heapsort

- Vantagens:
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- Desvantagens:
  - O Heapsort não é **estável**.
- Recomendado:
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap.*

Problema 1

Você faz parte do setor de admissões de uma universidade e precisa acompanhar, em tempo real, a k-ésima maior nota de teste entre os candidatos. Isso ajuda a determinar, de forma dinâmica, as notas de corte para entrevistas e admissões conforme novos candidatos enviam suas pontuações.

Sua tarefa é implementar uma classe que, dado um inteiro `k`, mantenha um fluxo de notas de teste e retorne continuamente a k-ésima maior nota sempre que uma nova nota for submetida. Mais especificamente, estamos interessados na k-ésima maior nota na lista ordenada de todas as notas recebidas até o momento.

Implemente a classe `KthLargest`:

- `KthLargest(int k, vector<int>& nums)` Inicializa o objeto com o inteiro `k` e o fluxo de notas de teste `nums`.
- `int add(int val)`: Adiciona uma nova nota `val` ao fluxo e retorna o elemento que representa a k-ésima maior nota no conjunto de notas até o momento.

Proponha uma solução usando vetores

Problema 1: vetores

Qual a complexidade?

Tempo: O(m * n log n)

Espaço: O(m)

m: # chamadas a função *add()*

DCC

Problema 1: solução 2

Melhore a complexidade para:

Tempo: O(m log k)

Espaço: O(k)

Problema 1: minHeap tamanho k

```
class KthLargest {
private:
    priority_queue<int, vector<int>, greater<int>> minHeap;
    int k;

public:
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        for (int num : nums) {
            minHeap.push(num);
            if (minHeap.size() > k) {
                minHeap.pop();
            }
        }
    }

    int add(int val) {
        minHeap.push(val);
        if (minHeap.size() > k) {
            minHeap.pop();
        }
        return minHeap.top();
    }
};
```