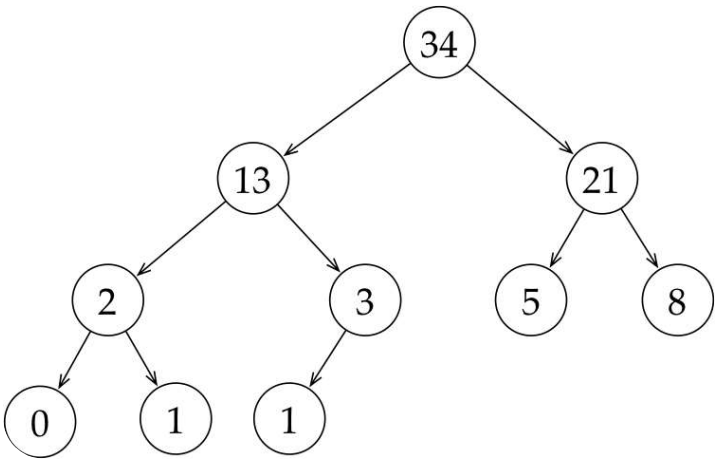
Heaps

Nesta seção aprenderemos o funcionamento de *heaps*, uma estrutura de dados extremamente útil em Ciência da Computação. Primeiro, veremos como heaps são visualizados e representados. Depois, aprenderemos como inserir ou remover um elemento de um heap e entenderemos como heaps fazem essas duas operações de forma extremamente eficiente.

Representação

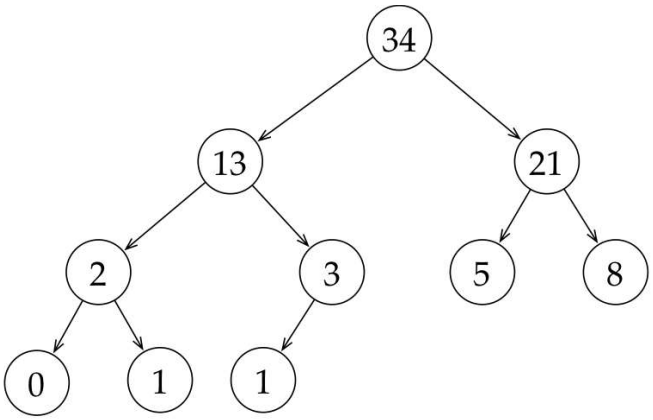
Um heap é uma estrutura de dados usada para representar uma coleção de elementos organizados de uma maneira específica. Podemos visualizar um heap como sendo uma árvore binária balanceada—uma árvore em que os elementos folha estão a no máximo um nível de distância uns dos outros—que obedece a uma importante propriedade, chamada aqui de *propriedade de heap*. Essa propriedade nos diz que o valor de qualquer nodo em um heap não é maior que o valor de seus filhos. A figura abaixo mostra um exemplo de heap.



Exemplo de heap. É útil visualizar um heap como uma árvore.

Apesar de heaps serem visualizados como árvore binárias, eles são implementados de modo diferente. Em geral, árvores binárias são implementadas usando uma estrutura encadeada, na qual um nodo possui apontadores (referências) para seus dois filhos (filho da direita e filho da esquerda). Heaps, por outro lado, são implementados usando uma estruturaa contígua (mais especificamente, um arranjo), no qual os filhos de um nodo estão em posições específicas do arranjo. Se um nodo estiver na posição i do arranjo, seus filhos da esquerda e direita estarão nas posições $2i$ e $2i + 1$, respectivamente.

O pai do nodo na posição i do arranjo estará na posição $i/2$, em que o símbolo $/$ representa a divisão inteira de um valor pelo outro, ou seja, $6/2$ e $7/2$ dão 3 como resultado. Em Python 3.x, o operador `//` é responsável por realizar a divisão inteira de dois números. A figura abaixo mostra o heap mostrado anteriormente bem como sua representação usando um arranjo.



	34	13	21	2	3	5	8	0	1	1
0	1	2	3	4	5	6	7	8	9	10

Representação de um heap.

A figura acima nos chama a atenção para um ponto fundamental. O fato de os filhos de um nodo estaem nas posições $2i$ e $2i + 1$ nos força a tomar uma decisão importante sobre onde colocar o primeiro elemento do heap (raiz da árvore). Normalmente, o primeiro elemento de um arranjo A fica em $A[0]$. Entretanto, se colocássemos o primeiro elemento do heap na posição 0 do arranjo, seus filhos ficariam nas posições $2 \times 0 = 0$ e $2 \times 0 + 1 = 1$. Mas neste caso teríamos um elemento e seu filho na mesma posição do arranjo (posição 0). Isso não pode acontecer. Por causa disso, a convenção adotada é que o primeiro elemento do heap fique na posição 1 do arranjo. A posição 0 do arranjo fica vazia (inutilizada).

O primeiro elemento do heap é um elemento particularmente importante. O fato de o valor de um nodo ser sempre maior que o valor de seu pai nos diz que o menor elemento do heap é o elemento mais acima (raiz da árvore). E o fato de os filhos de um elemento estarem em posições à direita dele no arranjo nos diz que o menor elemento do heap será o primeiro elemento do arranjo (elemento na posição 1).

Resumindo o que vimos até agora sobre a organização dos elementos em um heap, temos:

- O primeiro elemento do heap (raiz da árvore) é o menor elemento do heap e está na posição 1 do arranjo (e não na posição 0).
- Para um elemento na posição i do arranjo A que representa o heap:
    - O filho da esquerda está em $A[2i]$.
    - O filho da direita está em $A[2i + 1]$.
    - O pai está em $A[i/2]$.

Podemos traduzir essas coisas para código da seguinte forma:

```
def menor_elemento(A):
    return A[1]

def filho_esquerda(A, i):
    return A[2 * i]

def filho_direita(A, i):
    return A[2 * i + 1]

def pai(A, i):
    return A[i // 2]
```

Juntando o que vimos até agora, podemos dizer que um dado arranjo representa um heap se cada elemento possuir um valor maior que o valor de seu pai, ou analogamente, que cada elemento possui um valor menor que o valor de seus filhos. Com isso, podemos escrever uma função que verifica se um dado arranjo é um heap.

```
def eh_heap(A):
    for i in range(2, len(A)):
        if A[i] > A[i // 2]:
            return False
    return True
```

Construindo um Heap

Dado um arranjo de elementos, vimos que é fácil verificar se ele é ou não um heap. Mas existe uma pergunta importante que não respondemos até agora: *Como fazer com que um arranjo de elementos se torne um heap?* O objetivo desta seção é responder a essa pergunta.

Inicialmente, aprenderemos como transformar em heap um arranjo que teve a propriedade de heap violada em uma das pontas. Depois, usaremos esse conhecimento para transformar um arranjo qualquer de elementos em um heap. Nossos exemplos usarão arranjos de números inteiros, mas é possível

implementar as mesmas coisas com arranjos de elementos de qualquer tipo comparável.

Heaps nos permitem realizar duas operações de modo eficiente: remoção do menor elemento e inserção de um novo elemento. Cada uma dessas operações acontece em uma das pontas do arranjo. Remoções ocorrem à esquerda (no início do arranjo) e inserções ocorrem à direita (no final do arranjo). Sempre que removemos ou inserimos um elemento em um heap, a propriedade de heap é violada, ou seja, os elementos restantes não formam mais um heap. Precisamos de meios de "consertar" o heap sempre que isso acontecer.

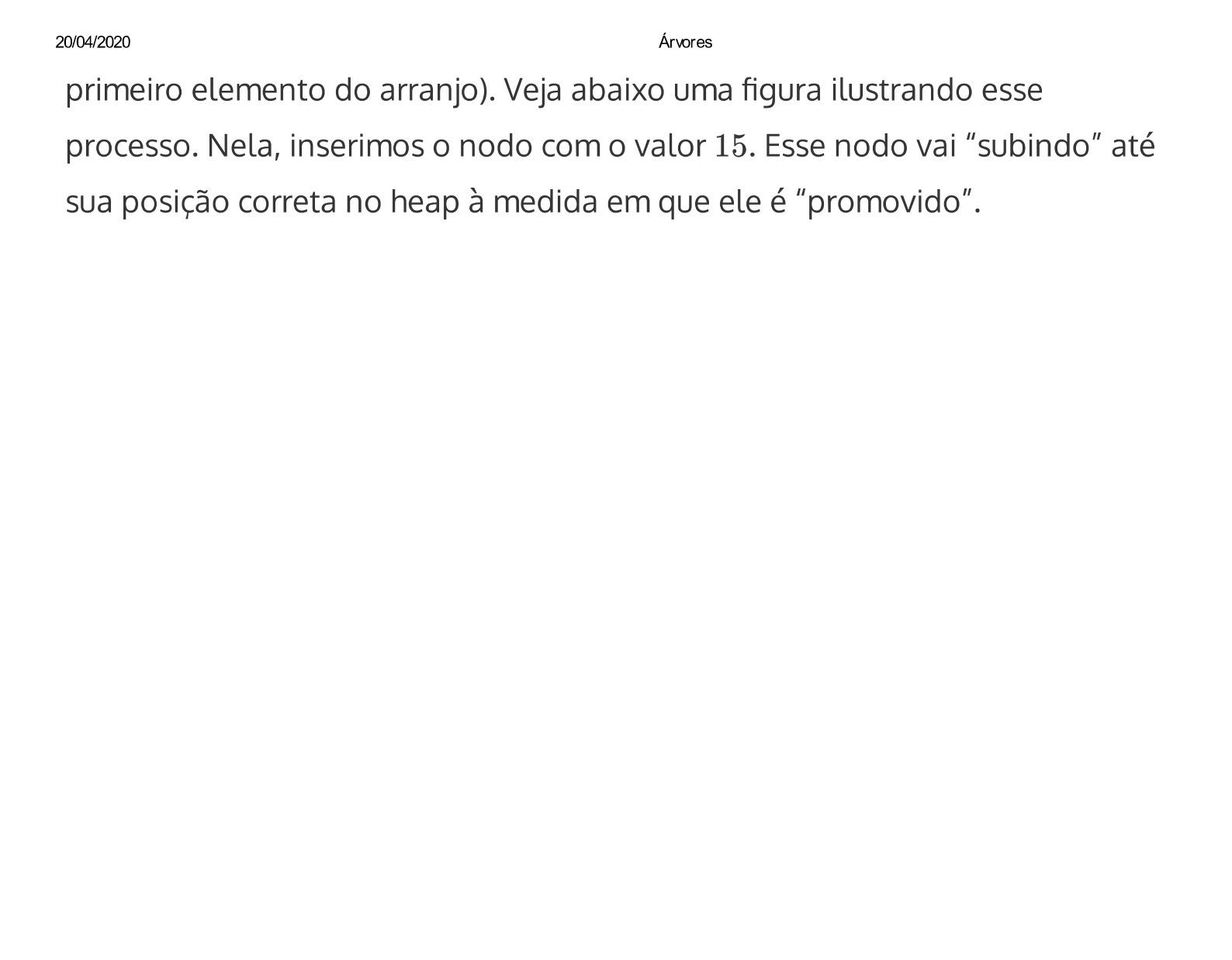
Vamos começar aprendendo como consertar um heap quando inserimos um novo elemento no arranjo.

Inserção

Inserir um elemento em um heap significa colocar o novo elemento no final do arranjo e reconstruir o heap (fazer com que ele volte a obedecer a *propriedade de heap*). De forma mais precisa, temos um arranjo A em que os elementos $A[1]$ até $A[n-1]$ constituem um heap. Então adicionamos um novo elemento ao final do arranjo, na posição $A[n]$. Este novo arranjo pode não ser um heap. Para reconstruir o heap, precisamos "promover" o novo elemento, ou seja, fazer com que ele "suba" até o lugar em que ele deve ficar no heap.

Para fazer com que o elemento recentemente inserido fique na posição correta, usaremos a função `promove`. Essa função faz com que o elemento suba até a posição correta na árvore, o que fazemos trocamos ele de posição com o pai dele (na prática, o elemento está sendo movido para posições mais à esquerda no arranjo). Este processo continua até que o novo elemento seja maior ou igual ao pai dele, ou até que ele chegue na raiz da árvore (seja o

primeiro elemento do arranjo). Veja abaixo uma figura ilustrando esse processo. Nela, inserimos o nodo com o valor 15. Esse nodo vai "subindo" até sua posição correta no heap à medida em que ele é "promovido".



Promoção de um elemento em um heap.

Em alto nível, podemos definir o funcionamento da função `promove` como:

- Enquanto o novo elemento for maior que seu pai ou ainda não for a raiz da árvore, troque o elemento de posição com seu pai.

A implementação da ideia acima pode parecer complicada, mas se usarmos o que vimos até agora ela se tornará simples. Vejamos.

```
def promove(A, n):
    i = n
    while True:
        # Elemento chegou na raiz da árvore.
        if i == 1:
            break

        # Elemento chegou na posição correta.
```

```
        p = i // 2
        if A[p] >= A[i]:
            break

        # Troca elemento de lugar com o pai.
        A[p], A[i] = A[i], A[p]
        i = p
```

Remoção

Enquanto a inserção adiciona um elemento no último nível da árvore (final do arranjo) e vai subindo o elemento até sua posição correta (posições mais à esquerda no arranjo), a remoção retira a raiz da árvore (primeiro elemento do arranjo).

Quando removemos o primeiro elemento do arranjo precisamos colocar outro em seu lugar, pois, da forma como heaps são organizados, não podem existir "buracos" no arranjo. Para tapar o buraco deixado pela remoção do primeiro elemento do arranjo, podemos, por exemplo, colocar o último elemento do arranjo nesse buraco. Ao fazer isso, a propriedade de heap pode não ser mantida para o arranjo como um todo. Logo, precisamos reconstruir o heap. Para reconstruir o heap, precisamos "demover" o novo elemento, ou seja, fazer com que ele "desça" até o lugar em que ele deve ficar no heap.

A demoção de um elemento no heap é feita por meio de sucessivas trocas de posição do elemento com seus filhos. Essas trocas acontecem até que o elemento não tenha filhos com quem ser trocado, ou até que ele seja menor ou igual a seus filhos.

O processo de demoção é um pouco mais caótico que o processo de promoção. Na promoção de um elemento, ele sempre sobe em linha reta em direção à raiz da árvore. Durante a demoção, na descida do elemento em direção aos níveis mais baixos da árvore, podem ocorrer uns zigue-zagues,

pois o elemento sempre troca de lugar com o maior de seus filhos. A figura abaixo ilustra o processo de demoção de um elemento.

Demoção de um elemento em um heap.

Em alto nível, podemos definir o procedimento de demoção como:

- Enquanto o novo elemento send demovido for maior que seu pai ou ainda tiver filhos, troque o elemento de posição com o maior de seus filhos.

Antes de partir para uma implementação, vamos detalhar um pouco mais a ideia acima:

- Se o elemento não possuir filhos, podemos terminar o procedimento. Caso contrário, precisamos encontrar o menor dos filhos.

- Se o elemento for menor que o menor de seus filhos, podemos terminar o procedimento. Caso contrário, trocamos o elemento de lugar com o maior de seus filhos e repetimos o processo.

Vejamos uma implementação do que acabamos de discutir.

```
def demove(A, n):
    i = 1
    while True:
        c = 2 * i

        # Elemento não tem mais filhos.
        if c > n:
            break

        # Encontra o índice do maior dos filhos.
        if c + 1 <= n:
            if A[c + 1] > A[c]:
                c += 1

        # O elemento é menor que seu maior filho.
        if A[i] <= A[c]:
            break

        # Troca elemento de lugar com o maior filho.
        A[c], A[i] = A[i], A[c]
        i = c
```

Com as operações vistas até aqui, somos capazes de implementar tanto a inserção de um novo elemento em um heap (que usará o procedimento `promove`) quando a remoção do maior elemento do heap (que usará o procedimento `demove`) em complexidade logarítmica no tamanho do heap. Por causa da eficiência das operações em heaps, eles são usados em diversos contextos em Ciência da Computação. Por exemplo, heaps são a base de filas de prioridade e do algoritmo de ordenação *Heapsort*.

← Anterior Início Próximo →