Heaps.
pág 221 até 233.

Capítulo 7.
Heaps e Filas de Prioridade:

Uma estrutura de dados de heap é uma estrutura de dados baseada em árvore na qual cada nó da árvore tem uma relação específica com outros nós, e eles são armazenados em uma ordem específica. Dependendo da ordem específica dos nós na árvore, os heaps podem ser de diferentes tipos, como um heap mínimo e um heap máximo.

Uma fila de prioridade é uma estrutura de dados importante que é semelhante às estruturas de dados de fila e pilha que armazenam dados juntamente com a prioridade associada a eles. Nisso, os dados são servidos de acordo com a prioridade. As filas de prioridade podem ser implementadas usando um array, lista encadeada e árvores; no entanto, elas são frequentemente implementadas usando um heap, pois é muito eficiente.

Neste capítulo, aprenderemos o seguinte:

- O conceito da estrutura de dados de heap e diferentes operações sobre ela
- Compreender o conceito de fila de prioridade e sua implementação usando Python

Heaps:

Uma estrutura de dados de heap é uma especialização de uma árvore na qual os nós são ordenados de uma maneira específica. Um heap é uma estrutura de dados na qual cada elemento de dados satisfaz uma propriedade de heap, e a propriedade de heap afirma que deve haver uma determinada relação entre um nó pai e seus nós filhos. De acordo com essa determinada relação na árvore, os heaps podem ser de dois tipos, em outras palavras, heaps máximos e heaps mínimos. Em um

heap máximo, o valor de cada nó pai deve ser sempre maior ou igual a todos os seus filhos. Nesse tipo de árvore, o nó raiz deve ser o maior valor na árvore. Por exemplo, veja a Figura 7.1 mostrando o heap máximo no qual todos os nós têm valores maiores em comparação com seus filhos:

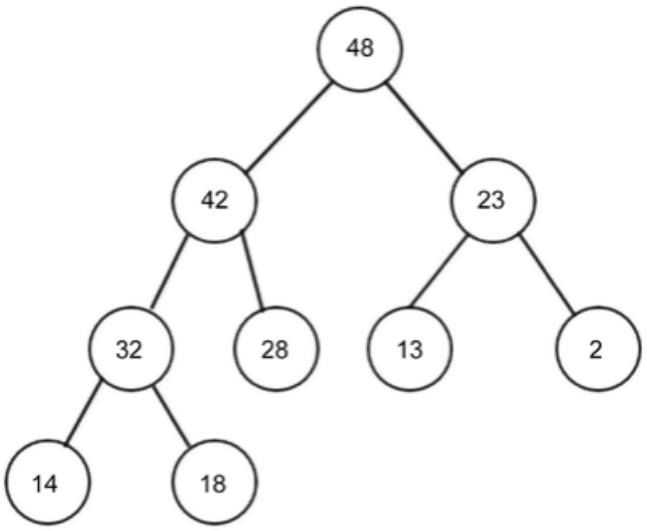

Figure 7.1: An example of a max heap

Em um heap mínimo, a relação entre o nó pai e os filhos é que o valor do nó pai deve sempre ser menor ou igual ao de seus filhos. Essa regra deve ser seguida por todos os nós na árvore. No heap mínimo, o nó raiz contém o valor mais baixo. Por exemplo, veja a Figura 7.2 mostrando o heap mínimo em que todos os nós têm valores menores em comparação com seus filhos:

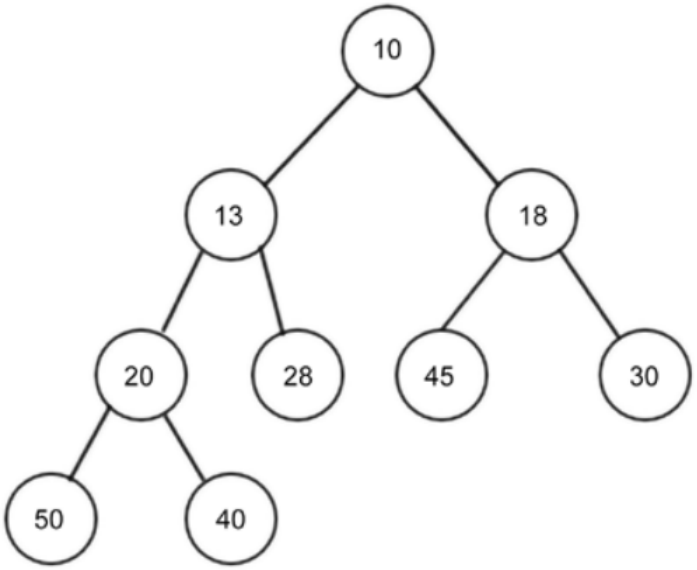

Figure 7.2: An example of a min heap

O heap é uma estrutura de dados importante devido a suas diversas aplicações e usos na implementação de algoritmos de ordenação de heap e filas de prioridade. Vamos discutir esses detalhadamente mais tarde no capítulo. O heap pode ser qualquer tipo de árvore; no entanto, o tipo mais comum de heap é um heap binário, em que cada nó tem no máximo dois filhos.

Se o heap binário for uma árvore binária completa com n nós, então terá uma altura mínima de log2n.

Uma árvore binária completa é aquela em que cada linha deve ser preenchida completamente antes de começar a preencher a próxima linha, como mostrado na seguinte Figura 7.3:

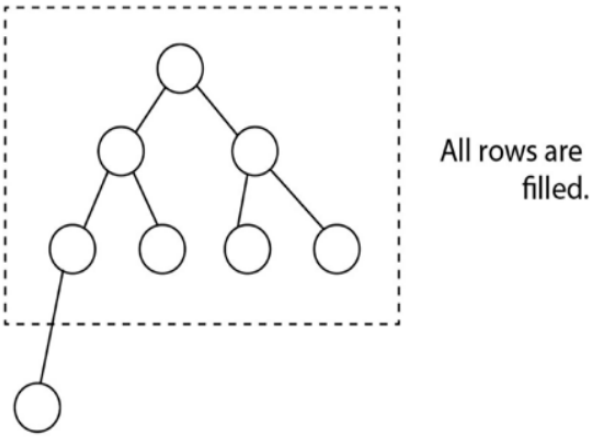

Figure 7.3: An example of a complete binary tree

Para implementar o heap, podemos derivar uma relação entre os nós pai e filho nos valores de índice. A relação é que os filhos de qualquer nó no índice n podem ser recuperados facilmente, em outras palavras, o filho esquerdo estará localizado em 2n e o filho direito estará localizado em 2n+1. Por exemplo, o nó C estaria no índice 3, já que o nó C é um filho direito do nó A, que está no índice 1, então ele se torna 2n+1 = 2*1 + 1 = 3. Essa relação sempre é verdadeira. Digamos que temos uma lista de elementos {A, B, C, D, E} como mostrado na Figura 7.4. Se armazenarmos qualquer elemento em um índice i, seu pai pode ser armazenado no índice i/2, por exemplo, se o índice do nó D for 4, seu pai estará no 4/2 = 2, índice 2. O índice da raiz deve começar a partir de 1 no array. Veja a Figura 7.4 para entender o conceito:

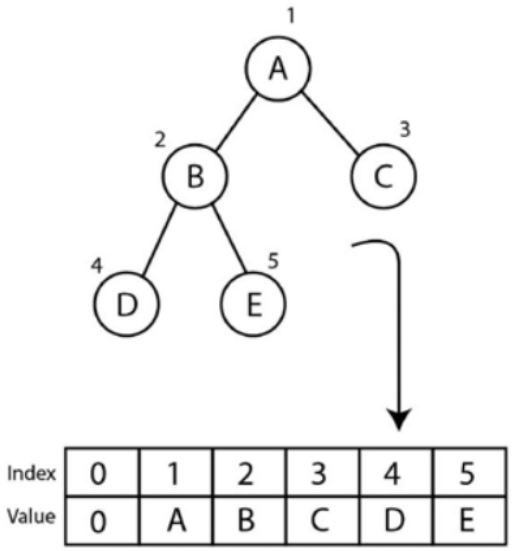

Figure 7.4: Binary tree and index positions of all the nodes

Essa relação entre pai e filho é uma árvore binária completa. Em relação aos valores de indexação, é muito importante para recuperar, pesquisar e armazenar eficientemente os elementos de dados no heap. Devido a essa propriedade, é muito fácil implementar o heap. A única restrição é que devemos ter a indexação começando em 1 e, se implementarmos o heap usando um array, teremos que adicionar um elemento fictício no índice 0 no array. A seguir, vamos entender a implementação do heap. É importante notar que discutiremos todos os conceitos em relação ao min heap e a implementação para o max heap será muito semelhante a ele, com a única diferença sendo a propriedade do heap.

Vamos discutir a implementação do min heap usando o Python. Começamos com a classe heap, como segue:

```
class MinHeap:
    def __init__(self):
        self.heap = [0]
        self.size = 0
```

Inicializamos a lista heap com um zero para representar o primeiro elemento fictício e adicionamos um elemento fictício apenas para iniciar a indexação dos itens de dados a partir de 1, uma vez que, se começarmos a indexação a partir de 1, o acesso aos elementos se torna muito fácil devido à relação pai-filho. Também criamos uma variável para armazenar o tamanho do heap. Discutiremos mais adiante diferentes operações, como inserir, excluir e excluir em uma localização específica no heap. Vamos começar com a operação de inserção no heap.

Operação de inserção

A inserção de um item em um min heap funciona em duas etapas. Primeiro, adicionamos o novo elemento ao final da lista (que entendemos ser o fundo da árvore) e incrementamos o tamanho do heap em um. Em segundo lugar, após cada operação de inserção, precisamos organizar o novo elemento na árvore heap, para organizar todos os nós de tal forma que satisfaça a propriedade do heap, que, neste caso, é que cada nó deve ser maior que seu pai. Em outras palavras, o valor do nó pai deve sempre ser menor ou igual ao de seus filhos, e o elemento mais baixo no min-heap precisa ser o elemento raiz. Portanto, primeiro inserimos um elemento no último heap da árvore; no entanto, após inserir um elemento no heap, é possível que a propriedade do heap seja violada. Nesse caso, os nós devem ser reorganizados para que todos os nós satisfaçam a propriedade do heap. Esse processo é chamado de heapifying. Para heapify o min heap, precisamos encontrar o mínimo de seus filhos e trocá-lo pelo elemento atual, e esse processo deve ser repetido até que a propriedade do heap seja satisfeita para todos os nós.

Vamos considerar um exemplo de adicionar um elemento no min heap, como inserir um novo nó com um valor de 2 na Figura 7.5:

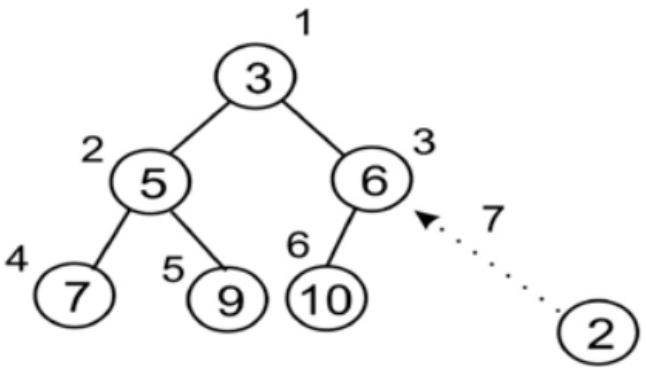

Figure 7.5: Insertion of a new node 2 in the existing heap

O novo elemento será adicionado à última posição na terceira linha ou nível. Seu valor de índice é 7. Comparamos esse valor com o de seu pai. O pai está no índice 7/2 = 3 (divisão inteira). O nó pai contém o valor 6, que é maior do que o valor do novo nó (em outras palavras, 2), então, de acordo com a propriedade do heap mínimo, trocamos esses valores, como mostrado na Figura 7.6:

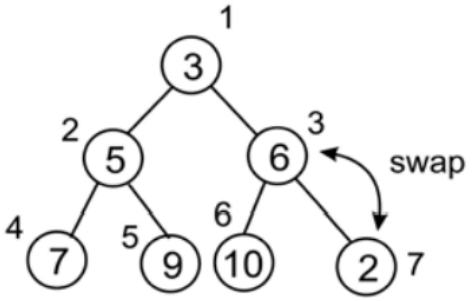

Figure 7.6: Swapping nodes 2 and 6 to maintain the heap property

O novo elemento de dados foi trocado e movido para o índice 3. Como temos que verificar todos os nós até a raiz, verificamos o índice do nó pai, que é 3/2 = 1 (divisão inteira), então continuamos o processo para heapify.

Então, comparamos ambos os elementos e trocamos novamente, como mostrado na Figura 7.7:

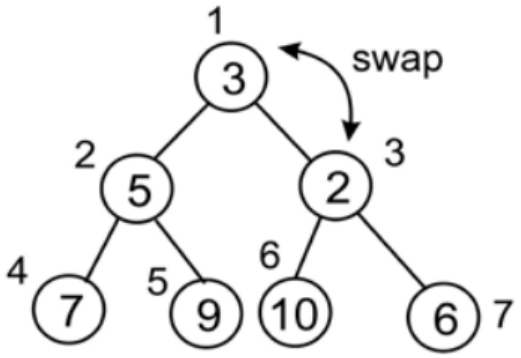

Figure 7.7: Swapping nodes 2 and 3 to maintain the heap property

Após a troca final, chegamos à raiz. Aqui, podemos perceber que esse heap segue a definição do heap mínimo, como mostrado na Figura 7.8:

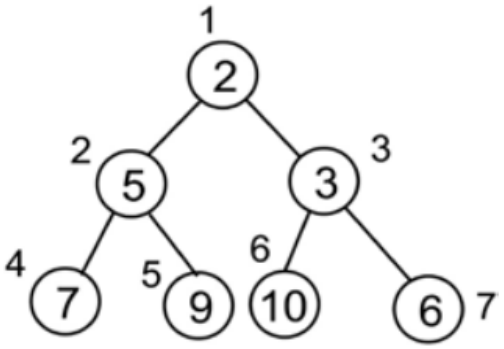

Figure 7.8: Final heap after insertion of a new node 2

Agora, vamos pegar outro exemplo para ver como criar e inserir elementos em um heap. Começamos com a construção de um heap inserindo 10 elementos, um por um. Os elementos são {4, 8, 7, 2, 9, 10, 5, 1, 3, 6}. Podemos ver um processo passo a passo para inserir elementos no heap na Figura 7.9:

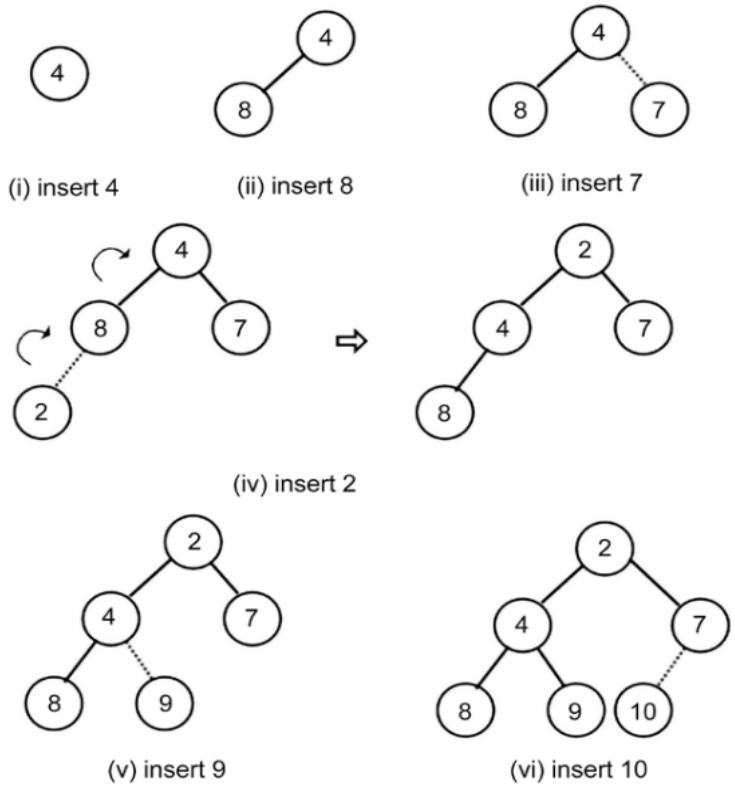

Figure 7.9: The step-by-step procedure to create a heap

Podemos ver, no diagrama anterior, um processo passo a passo para inserir elementos no heap. Aqui, continuamos adicionando elementos, como mostrado na Figura 7.10:

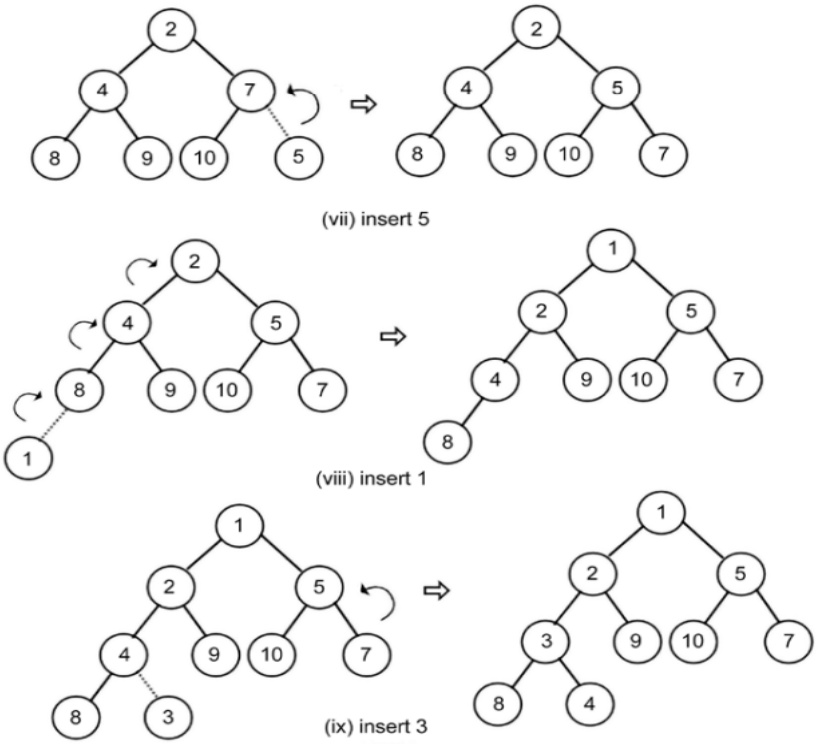

Figure 7.10: Steps 7 to 9 in creating the heap

Finalmente, inserimos um elemento, 6, no heap, como mostrado na Figura 7.11:

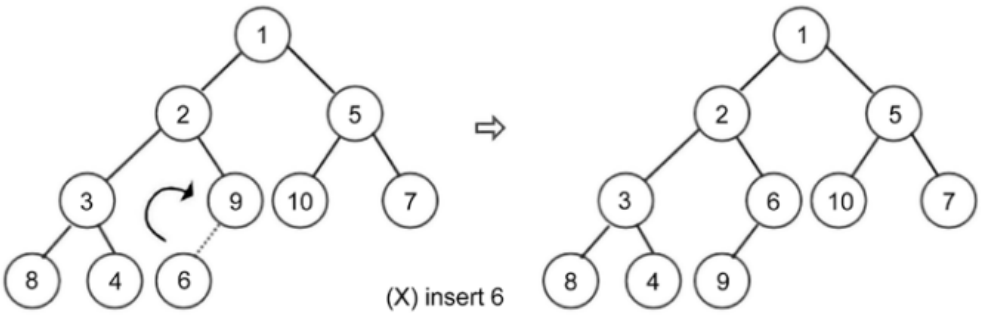

Figure 7.11: Last step and construction of the final heap

A implementação da operação de inserção no heap é discutida da seguinte forma. Primeiramente, criamos um método auxiliar, chamado arrange, que cuida dos arranjos de todos os nós após a inserção de um novo nó. Aqui está a implementação do método arrange(), que deve ser definido na classe MinHeap:

```
def arrange(self, k):
    while k // 2 > 0:
        if self.heap[k] < self.heap[k//2]:
            self.heap[k], self.heap[k//2] = self.heap[k//2],
self.heap[k]

        k //= 2
```

Executamos o loop até chegarmos ao nó raiz; até lá, podemos continuar arranjando o elemento. Aqui, estamos usando a divisão inteira. O loop sairá após a seguinte condição:

```
while k // 2 > 0:
```

Depois disso, comparamos os valores entre o nó pai e o nó filho. Se o pai for maior que o filho, trocamos os dois valores:

```
if self.heap[k] < self.heap[k//2]:
            self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap[k]
```

Finalmente, após cada iteração, subimos na árvore:

```
        k //= 2
```

Este método garante que os elementos sejam ordenados corretamente. Agora, para adicionar novos elementos no heap, precisamos usar o seguinte método de inserção, que deve ser definido na classe MinHeap:

```
def insert(self, item):
    self.heap.append(item)
    self.size += 1
    self.arrange(self.size)
```

No código acima, podemos inserir um elemento usando o método append; em seguida, aumentamos o tamanho do heap. Então, na última linha do método insert, chamamos o método arrange() para reorganizar o heap (heapify-lo) para garantir que todos os nós no heap satisfaçam a propriedade do heap.

Agora, vamos criar o heap e inserir os dados {4, 8, 7, 2, 9, 10, 5, 1, 3, 6} usando o método insert(), que está definido na classe MinHeap, como mostrado no código a seguir:

```
h = MinHeap()
for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
    h.insert(i)
```

Podemos imprimir a lista do heap apenas para inspecionar como os elementos estão ordenados. Se você redesenhar isso como uma estrutura de árvore, perceberá que atende às propriedades necessárias de um heap, similar ao que criamos manualmente:

```
print(h.heap)
```

A saída do código acima é a seguinte:

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
```

Podemos ver na saída que todos os itens de dados do heap na matriz estão na posição de índice conforme a Figura 7.11. Em seguida, discutiremos a operação de exclusão no heap.

Operação de exclusão:

A operação de exclusão remove um elemento do heap. Para excluir qualquer elemento do heap, vamos primeiro discutir como podemos excluir o elemento raiz, pois é frequentemente usado para vários casos de uso, como encontrar o elemento mínimo ou máximo em um heap. Lembre-se de que, em um min-heap, o elemento raiz denota o valor mínimo da lista, e a raiz do max-heap fornece o valor máximo da lista de elementos.

Depois de excluir o elemento raiz do heap, fazemos o último elemento do heap o novo elemento raiz do heap. Nesse caso, a propriedade do heap não será satisfeita pela árvore. Portanto, temos que reorganizar os nós da árvore de modo que todos os nós da árvore satisfaçam a propriedade do heap. A operação de exclusão em min-heap funciona da seguinte maneira:

1. Depois de excluir o nó raiz, precisamos de um novo nó raiz. Para isso, pegamos o último item da lista e o tornamos o novo nó raiz.
2. Como o último nó selecionado pode não ser o menor elemento do heap, temos que reorganizar os nós do heap.
3. Reorganizamos os nós do nó raiz ao último nó (que é transformado em um novo nó raiz); esse processo é chamado de heapify. Como nos movemos de cima para baixo (ou seja, do nó raiz até o último elemento) do heap, esse processo é chamado de percolate down.

Vamos considerar um exemplo para nos ajudar a entender esse conceito no heap a seguir. Primeiro, excluímos o nó raiz que tem valor 2, como mostrado na Figura 7.12:

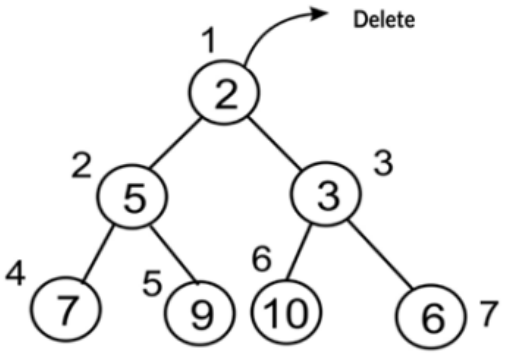

Figure 7.12: Deletion of a node with value 2 at the root in the existing heap

Uma vez que excluímos a raiz, precisamos escolher um nó que possa ser a nova raiz; comumente, escolhemos o último nó, ou seja, o nó 6 no índice 7. Portanto, o último elemento, 6, é colocado na posição da raiz, conforme mostrado na Figura 7.13:

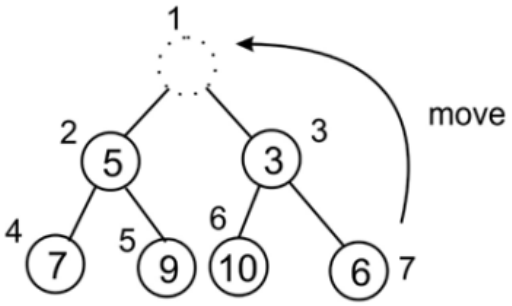

Figure 7.13: Moving the last element, in other words, node 6 to the root position

Depois de mover o último elemento para a nova raiz, fica claro que esta árvore agora não está satisfazendo a propriedade de heap mínimo. Portanto, temos que reorganizar os nós do heap, assim movemos para baixo da raiz aos nós no heap, isto é, heapify na árvore. Então, comparamos o valor do nó recém substituído com todos os seus nós filhos na árvore. Neste exemplo, comparamos os dois filhos da raiz, ou seja, 5 e 3. Como o filho direito é menor, seu índice é 3, o que é representado como (índice da raiz * 2 + 1). Vamos em frente com este nó e comparar o novo nó raiz com o valor neste índice, conforme mostrado na Figura 7.14:

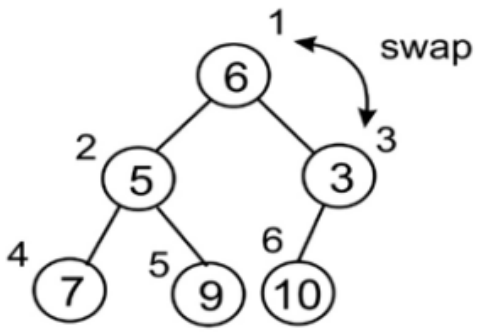

Figure 7.14: Swapping of the root node with the node 3

Agora, o nó com valor 6 deve ser movido para o índice 3 de acordo com a propriedade de heap mínimo. Em seguida, precisamos compará-lo com seus filhos até o heap. Aqui, só temos um filho, então não precisamos nos preocupar com qual filho compará-lo (para um heap mínimo, é sempre o filho menor), conforme mostrado na Figura 7.15:

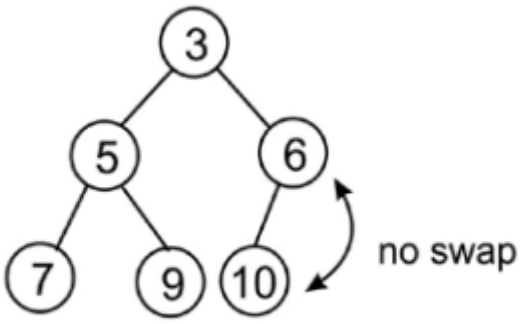

Figure 7.15: Swapping of node 6 and node 10

Não há necessidade de trocar aqui, pois está seguindo a propriedade de heap mínimo. Depois de chegar ao último, o heap final adere à propriedade de heap mínimo.

Para implementar a exclusão do nó raiz do heap usando Python, primeiro, implementamos o processo de percolate-down, em outras palavras, o método sink(). Antes de implementar o método sink(), implementamos um método auxiliar para descobrir qual dos filhos comparar com o nó pai. Esse método auxiliar é minchild(), que deve ser definido na classe MinHeap:

```
def minchild(self, k):
    if k * 2 + 1 > self.size:
        return k * 2
```

```
    elif self.heap[k*2] < self.heap[k*2+1]:
        return k * 2
    else:
        return k * 2 + 1
```

Neste método, primeiramente, verificamos se ultrapassamos o final da lista - se o fizermos, então retornamos o índice do filho esquerdo:

```
if k * 2 + 1 > self.size:
        return k * 2
```

Caso contrário, simplesmente retornamos o índice do menor dos dois filhos:

```
elif self.heap[k*2] < self.heap[k*2+1]:
        return k * 2
    else:
        return k * 2 + 1
```

Agora podemos criar o método sink(). O método sink() deve ser definido na classe MinHeap:

```
def sink(self, k):
    while k * 2 <= self.size:
        mc = self.minchild(k)
        if self.heap[k] > self.heap[mc]:
                self.heap[k], self.heap[mc] = self.heap[mc],
self.heap[k]
        k = mc
```

No código acima, primeiro executamos o loop até o final da árvore para que possamos afundar (mover para baixo) nosso elemento o quanto for necessário; isso é mostrado no seguinte trecho de código:

```
def sink(self, k):
    while k * 2 <= self.size:
```

Em seguida, precisamos saber qual dos filhos esquerdo ou direito comparar. É aqui que usamos a função minindex(), como mostrado no seguinte trecho de código:

```
mc = self.minchild(k)
```

Em seguida, comparamos o pai e o filho para ver se precisamos fazer a troca, como fizemos no método arrange() durante a operação de inserção:

```
if self.heap[k] > self.heap[mc]:
            self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
```

Finalmente, precisamos garantir que movemos para baixo na árvore em cada iteração para não ficarmos presos em um loop, como segue:

```
k = mc
```

Agora podemos implementar o método principal delete_at_root() em si, que deve ser definido na classe MinHeap:

```
def delete_at_root(self):
    item = self.heap[1]
    self.heap[1] = self.heap[self.size]
    self.size -= 1
    self.heap.pop()
    self.sink(1)
    return item
```

No código acima para exclusão do nó raiz, primeiro copiamos o elemento raiz em uma variável item, e então o último elemento é movido para o nó raiz na seguinte instrução:

```
 self.heap[1] = self.heap[self.size]
```

Além disso, reduzimos o tamanho do heap e removemos o elemento do heap, e então usamos o método sink() para reorganizar os elementos

do heap para que todos os elementos do heap sigam a propriedade de heap.

Agora podemos usar o seguinte código para excluir o nó raiz do heap. Primeiro, inserimos alguns itens de dados {2, 3, 5, 7, 9, 10, 6} no heap e depois removemos o nó raiz:

```
h = MinHeap()
for i in (2, 3, 5, 7, 9, 10, 6):
    h.insert(i)
print(h.heap)
n = h.delete_at_root()
print(n)
print(h.heap)
```

A saída do código acima é a seguinte:

```
[0, 2, 3, 5, 7, 9, 10, 6]
2
[0, 3, 6, 5, 7, 9, 10]
```

Podemos ver na saída que o elemento raiz 2 é retornado no novo heap e que os elementos de dados são rearranjados para que todos os nós do heap sigam a propriedade de heap (os índices dos nós podem ser verificados como mostrado na Figura 7.16).
A seguir, discutiremos se quisermos excluir algum nó com a posição de índice fornecida.