Heaps(cont) Heap Sort
e Filas de Prioridade.
Pág 234 a 245

Excluindo um elemento de uma posição específica de um heap:

Geralmente, excluímos um elemento na raiz, porém, um elemento pode ser excluído de uma posição específica do heap. Vamos entender com um exemplo. Dado o seguinte heap, vamos assumir que queremos excluir um nó com valor 3 no índice 2. Após excluir o nó com valor 3, movemos o último nó para o nó excluído, em outras palavras, o nó com valor 15, como mostrado na Figura 7.16:

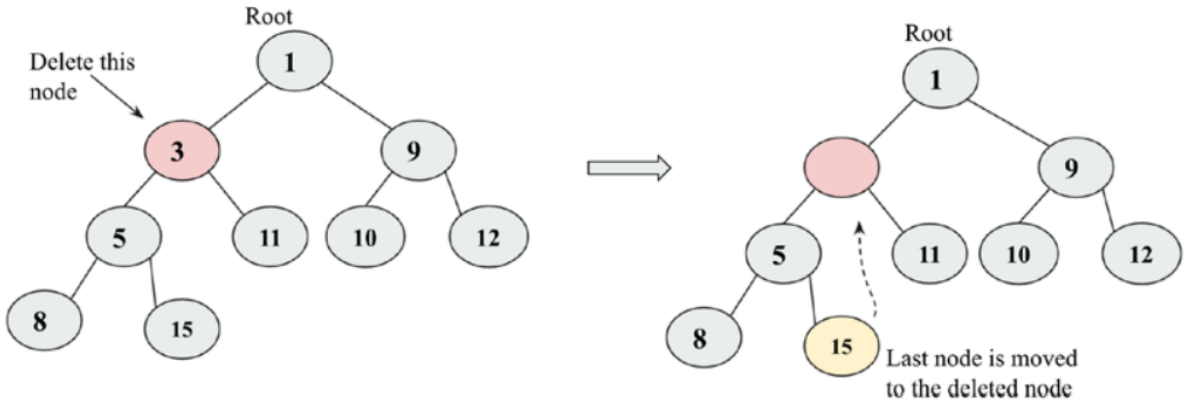

Figure 7.16: The deletion of node 3 from the heap

Depois de mover o último elemento para o nó excluído, comparamos isso com o elemento da raiz, como já é maior que o elemento da raiz, não trocamos. Em seguida, comparamos este elemento com todos os seus filhos e, como o filho da esquerda é menor, ele é trocado com o filho da esquerda, como mostrado na Figura 7.17:

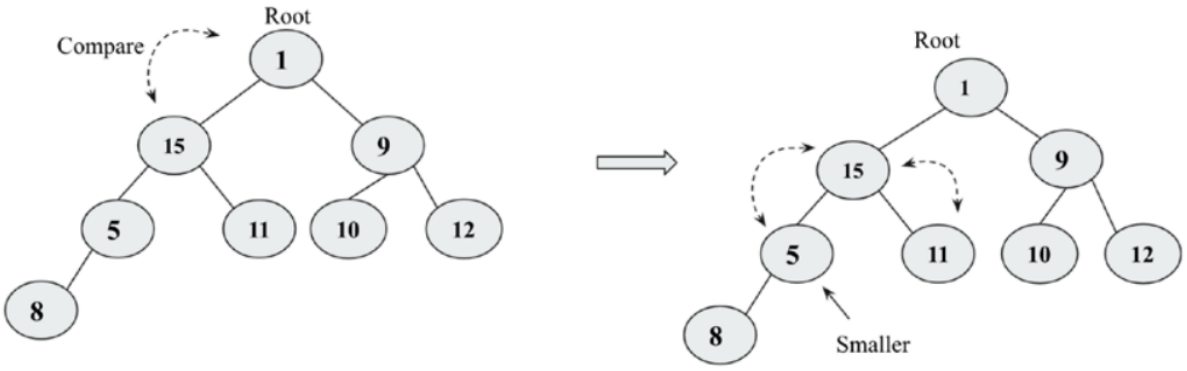

Figure 7.17: A comparison of node 15 with 5 and 11, and swapping node 15 and node 5

Depois de trocar o nó 15 com o nó 5, descemos no heap. Em seguida, comparamos o nó 15 com seu filho, o nó 8. Finalmente, o nó 8 e o nó 15 são trocados. Agora, a árvore final segue a propriedade do heap, como mostrado na Figura 7.18:

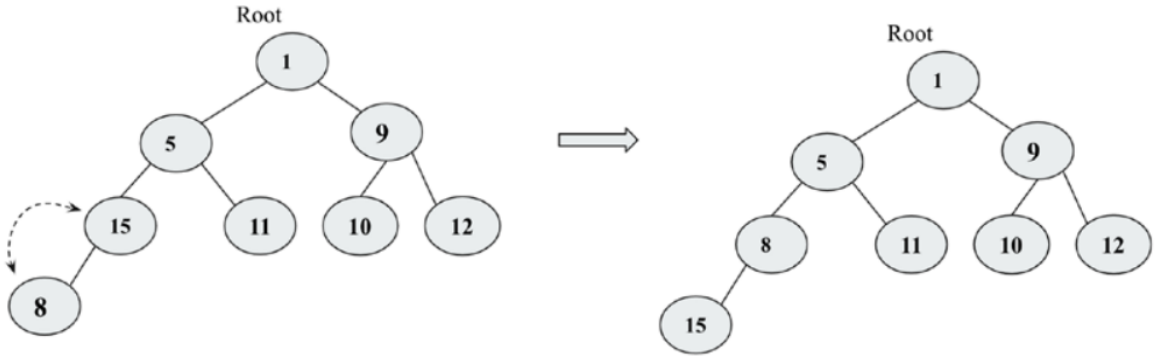

Figure 7.18: The final heap after swapping node 8 and node 15

A implementação da operação de exclusão para remover um item de dados em qualquer índice específico é dada abaixo, que deve ser definida na classe MinHeap:

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

Essa implementação é muito semelhante ao que vimos na seção anterior para excluir o elemento da raiz. A única diferença é que, neste código, especificamos o local do índice que deve ser excluído. O seguinte trecho de código demonstra a exclusão de um nó em uma localização específica 2 do heap criado a partir dos elementos de dados {4, 8, 7, 2, 9, 10, 5, 1, 3, 6}:

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

A saída do código anterior é a seguinte:

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

Na saída acima, vemos que, antes e depois, os nós da heap são colocados de acordo com suas posições de índice. Discutimos os conceitos e implementação usando exemplos de minheap; todas essas operações e conceitos podem ser facilmente implementados para um max-heap simplesmente revertendo a lógica nas condições em que garantimos que o nó pai deve ter valores menores em comparação com os filhos em min-heap. Agora, no caso de um max-heap, temos que fazer o valor maior no pai. Heaps são usados em várias aplicações, como implementar heap sort e filas de prioridade, que discutiremos nas seções subsequentes.

Heap sort

Heap é uma importante estrutura de dados para ordenar uma lista de elementos, já que é muito adequada para um grande número de elementos. Se quisermos ordenar uma lista de elementos, digamos em ordem crescente, podemos usar min-heap para este propósito; primeiro,

criamos um min-heap de todos os elementos de dados fornecidos, e de acordo com a propriedade da heap, o menor valor de dados será armazenado na raiz da heap. Com a ajuda da propriedade da heap, é fácil ordenar os elementos. O processo é o seguinte:

1. Crie um min-heap usando todos os elementos de dados fornecidos.

2. Leia e exclua o elemento raiz, que é o valor mínimo. Depois disso, copie o último elemento da árvore para a nova raiz e reorganize a árvore para manter a propriedade da heap.

3. Agora, repetimos o passo 2 até obtermos todos os elementos.

4. Finalmente, obtemos a lista ordenada de elementos.

Os elementos de dados são armazenados na heap aderindo à propriedade da heap; sempre que um novo elemento é adicionado ou excluído, a propriedade da heap é mantida usando os métodos auxiliares arrange() e sink(), respectivamente, conforme discutido nas seções anteriores.

Para implementar o heap sort usando a estrutura de dados heap, primeiro criamos uma heap com os itens de dados {4, 8, 7, 2, 9, 10, 5, 1, 3, 6} usando o código abaixo (detalhes da criação da heap são dados nas seções anteriores):

```
h = MinHeap()
unsorted_list = [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
for i in unsorted_list:
    h.insert(i)
print("Unsorted list: {}".format(unsorted_list))
```

No código acima, é criado um min-heap, h, e os elementos na lista não ordenada são inseridos. Após cada chamada do método insert(), a propriedade de ordem do heap é restaurada pela chamada subsequente do método sink.

Após a criação do heap, em seguida, lemos e excluímos o elemento raiz. Em cada iteração, obtemos o valor mínimo e, portanto, os itens de dados em ordem crescente. A implementação do método heap_sort() deve ser definida na classe minHeap (ela usa o método delete_at_root() discutido em seções anteriores):

```
def heap_sort(self):
    sorted_list = []
    for node in range(self.size):
        n = self.delete_at_root()
        sorted_list.append(n)
    return sorted_list
```

No código acima, criamos uma matriz vazia, sorted_list, que armazena todos os elementos de dados em ordem ordenada. Em seguida, executamos o loop pelo número de itens na lista. Em cada iteração, chamamos o método delete_at_root() para obter o valor mínimo, que é adicionado a sorted_list.

Agora podemos usar o algoritmo de ordenação por heap usando o seguinte código:

```
print("Unsorted list: {}".format(unsorted_list))
print("Sorted list: {}".format(h.heap_sort()))
```

A saída do código acima é a seguinte:

```
Unsorted list: [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
Sorted list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

A complexidade de tempo para construir o heap usando o método insert leva O(n) vezes. Além disso, para reorganizar a árvore após excluir o elemento raiz leva O(log n) vezes, já que percorremos de cima para baixo na árvore do heap e a altura do heap é log2(n), portanto, a complexidade de rearranjar a árvore é O(log n). Então, em geral, a complexidade de tempo pior caso do heapsort é O(n logn). Heapsort é

muito eficiente em geral, dando uma complexidade de pior caso, caso médio e melhor caso de O(nlogn).

Fila de prioridade:

Uma fila de prioridade é uma estrutura de dados semelhante a uma fila em que os dados são recuperados com base na política Primeiro a entrar, Primeiro a sair (FIFO), mas na fila de prioridade, a prioridade é anexada aos dados. Na fila de prioridade, os dados são recuperados com base na prioridade associada aos elementos de dados, sendo que os elementos de dados com a maior prioridade são recuperados antes dos elementos de dados de menor prioridade, e se dois elementos de dados tiverem a mesma prioridade, eles serão recuperados de acordo com a política FIFO.

Podemos atribuir a prioridade dos dados dependendo da aplicação. É usado em muitas aplicações, como escalonamento de CPU, e muitos algoritmos também dependem de filas de prioridade, como o menor caminho de Dijkstra, a busca A* e os códigos Huffman para compressão de dados.

Portanto, na fila de prioridade, o item com a maior prioridade é servido primeiro. A fila de prioridade armazena os dados de acordo com a prioridade associada aos dados, portanto, a inserção de um elemento será em uma posição específica na fila de prioridade. As filas de prioridade podem ser consideradas como filas modificadas que retornam os itens na ordem de maior prioridade em vez de retornar os itens na ordem FIFO. Uma fila de prioridade pode ser implementada modificando uma posição de enfileiramento inserindo o item de acordo com a prioridade. Isso é demonstrado na Figura 7.19, na qual, dada a fila, um novo item 5 é adicionado à fila em um índice específico (assumindo aqui que os itens de dados com valores mais altos têm prioridade mais alta):

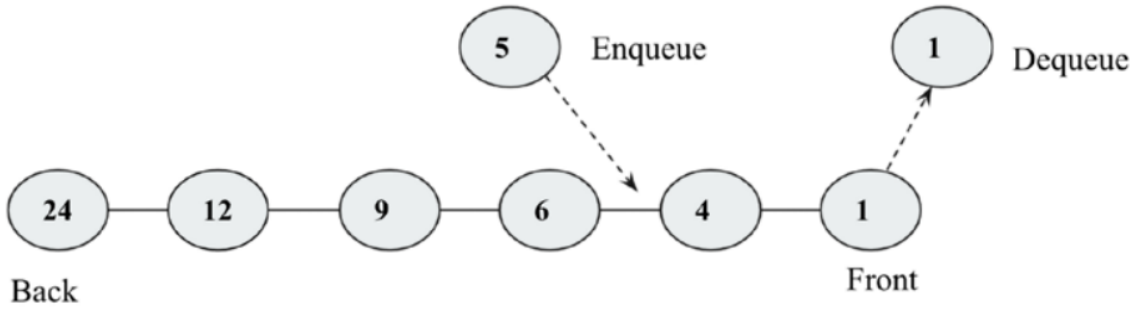

Figure 7.19: A demonstration of a priority queue

Vamos entender a fila de prioridade com um exemplo. Quando recebemos elementos de dados em uma ordem, os elementos são enfileirados na fila de prioridade na ordem de prioridade (assumindo que o valor de dados mais alto é de maior importância). Inicialmente, a fila de prioridade está vazia, então 3 é adicionado inicialmente na fila; o próximo elemento de dados é 8, que será enfileirado no início, pois é maior que 3. Em seguida, o item de dados é 2, depois 6 e, finalmente, 10, que são enfileirados na fila de prioridade de acordo com sua prioridade, e quando a operação de desenfileiramento é aplicada, o item de alta prioridade será desenfileirado primeiro. Todos os passos são representados na Figura 7.20:

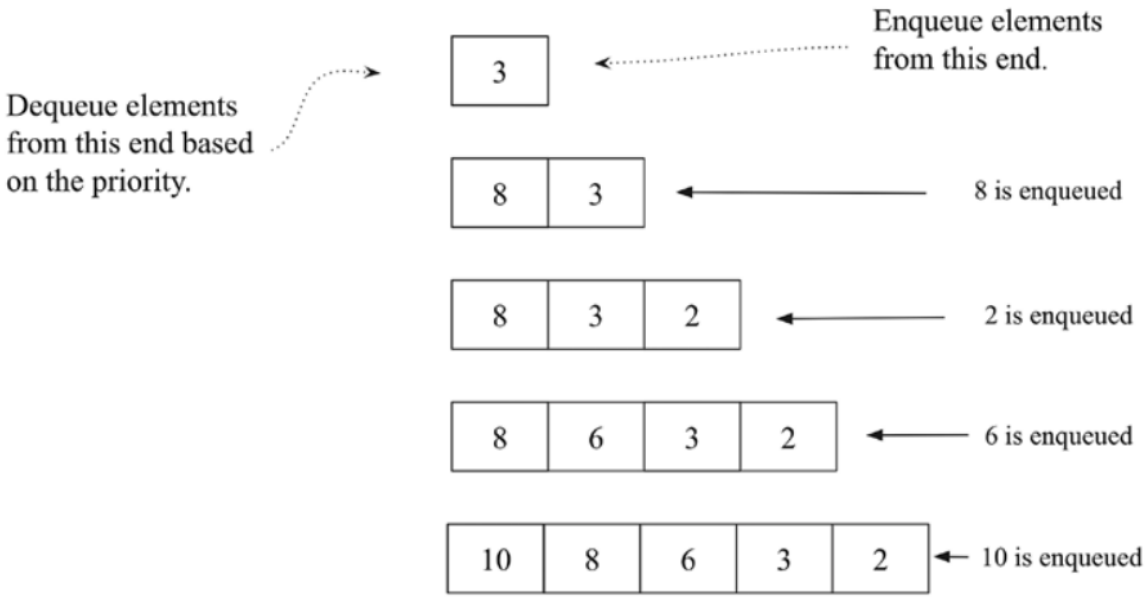

Figure 7.20: A step-by-step procedure to create a priority queue

Vamos discutir a implementação de uma fila de prioridade em Python. Primeiro, definimos a classe de nó. Uma classe de nó terá os elementos de dados juntamente com a prioridade associada aos dados na fila de prioridade:

```
# class for Node with data and priority
class Node:
    def __init__(self, info, priority):
        self.info = info
        self.priority = priority
```

Em seguida, definimos a classe PriorityQueue e inicializamos a fila:

```
# class for Priority queue
class PriorityQueue:
    def __init__(self):
        self.queue = []
```

A seguir, vamos discutir a implementação da operação de inserção para adicionar um novo elemento de dados à fila de prioridade. Na implementação, assumimos que o elemento de dados tem alta prioridade se tiver um valor de prioridade menor (por exemplo, um elemento de dados com o valor de prioridade 1 tem uma prioridade maior em comparação com o elemento de dados que tem um valor de prioridade 4). As seguintes são os casos de inserção de elementos em uma fila de prioridade:

1. Inserção de um elemento de dados na fila de prioridade quando a fila está inicialmente vazia.

2. Se a fila não estiver vazia, realizamos a travessia da fila e chegamos à posição de índice apropriada na fila de acordo com as prioridades associadas comparando as prioridades do nó existente com o novo nó. Adicionamos o novo nó antes do nó que possui uma prioridade maior que a do novo nó.

3. Se o novo nó tiver uma prioridade mais baixa que o valor de alta prioridade, então o nó será adicionado ao início da fila.

A implementação do método insert() é a seguinte, que deve ser definido na classe PriorityQueue:

```
def insert(self, node):
    if len(self.queue) == 0:
    # add the new node
        self.queue.append(node)
    else:
        # traverse the queue to find the right place for new
node
        for x in range(0, len(self.queue)):
            # if the priority of new node is greater
            if node.priority >= self.queue[x].priority:
            # if we have traversed the complete queue
                if x == (len(self.queue)-1):
                    # add new node at the end
                    self.queue.insert(x+1, node)
                else:
                    continue
            else:
                self.queue.insert(x, node)
                return True
```

A seguir, quando aplicamos a operação de exclusão na fila de prioridade, o elemento de dados de maior prioridade é retornado e removido da fila. Ele deve ser definido na classe PriorityQueue da seguinte forma:

```
def delete(self):
    # remove the first node from the queue
    x = self.queue.pop(0)
     print("Deleted data with the given priority-", x.info,
x.priority)
    return x
```

No código anterior, obtemos o elemento superior com o valor de prioridade mais alto. Além disso, a implementação do método show() que imprime todos os elementos de dados da fila de prioridade na ordem das prioridades deve ser definida na classe PriorityQueue:

```
def show(self):
    for x in self.queue:
        print(str(x.info)+ " - "+ str(x.priority))
```

Agora, vamos considerar um exemplo para ver como usar a fila de prioridade em que primeiro adicionamos elementos de dados ("Cat", "Bat", "Rat", "Ant" e "Lion") com prioridades associadas 13, 2, 1, 26 e 25, respectivamente:

```
p = PriorityQueue()
p.insert(Node("Cat", 13))
p.insert(Node("Bat", 2))
p.insert(Node("Rat", 1))
p.insert(Node("Ant", 26))
p.insert(Node("Lion", 25))
p.show()
p.delete()
```

A saída do código acima é a seguinte:

```
Rat - 1
Bat - 2
Cat - 13
Lion - 25
Ant - 26
Deleted data with the given priority- Rat 1
```

As filas de prioridade podem ser implementadas usando várias estruturas de dados. No exemplo acima, vimos sua implementação usando uma lista de tuplas, onde a tupla contém a prioridade como primeiro elemento e o item de dados de valor como próximo elemento. No entanto, as filas de prioridade são principalmente implementadas usando um heap, já que é eficiente com a complexidade de tempo de pior caso de O(log n) nas operações de inserção e remoção.

A implementação da fila de prioridade usando heap é muito semelhante ao que discutimos na implementação de min-heap. A única diferença é que agora armazenamos as prioridades associadas aos elementos de dados e criamos uma árvore de min-heap considerando os valores de prioridade usando uma lista de tuplas em Python. Para completude, o código para a fila de prioridade usando heaps é o seguinte:

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

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

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

    def sink(self, k):
        while k * 2 <= self.size:
        mc = self.minchild(k)
        if self.heap[k][0] > self.heap[mc][0]:
            self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
        k = mc

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

```
            return k * 2
        elif self.heap[k*2][0] < self.heap[k*2+1][0]:
            return k * 2
        else:
            return k * 2 + 1

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

Usamos o código abaixo para criar uma fila de prioridade com elementos de dados "Bat", "Cat", "Rat", "Ant", "Lion" e "Bear" com os valores de prioridade associados 2, 13, 18, 26, 3 e 4, respectivamente:

```
h = PriorityQueueHeap()
h.insert(2, "Bat")
h.insert(13,"Cat")
h.insert(18, "Rat")
h.insert(26, "Ant")
h.insert(3, "Lion")
h.insert(4, "Bear")
h.heap
```

A saída do código acima é a seguinte:

```
[(), (2, 'Bat'), (3, 'Lion'), (4, 'Bear'), (26, 'Ant'), (13, 'Cat'),
(18, 'Rat')]
```

Na saída acima, podemos ver que ela mostra uma árvore de min-heap que adere à propriedade de min-heap. Agora podemos usar o código abaixo para remover os elementos de dados:

```
for i in range(h.size):
```

```
    n = h.delete_at_root()
    print(n)
    print(h.heap)
```

A saída do código anterior é a seguinte:

```
'Bat
[(), (3, 'Lion'), (13, 'Cat'), (4, 'Bear'), (26, 'Ant'), (18, 'Rat')]
Lion
[(), (4, 'Bear'), (13, 'Cat'), (18, 'Rat'), (26, 'Ant')]
Bear
[(), (13, 'Cat'), (26, 'Ant'), (18, 'Rat')]
Cat
[(), (18, 'Rat'), (26, 'Ant')]
Rat
[(), (26, 'Ant')]
Ant
[()]
```

Na saída acima, podemos ver que os itens de dados são produzidos de acordo com as prioridades associadas aos elementos de dados.

Resumo:

Neste capítulo, discutimos uma importante estrutura de dados, ou seja, a estrutura de dados de heap. Também discutimos as propriedades de heap para min-heap e max-heap. Vimos a implementação de várias operações que podem ser aplicadas à estrutura de dados de heap, como heapifying e a inserção e remoção de um elemento de dados do heap. Também discutimos duas das aplicações importantes do heap - heap sort e uma fila de prioridade. O heap é uma estrutura de dados importante, pois tem muitas aplicações, como ordenação, seleção de valores mínimo e máximo em uma lista, algoritmos de gráficos e filas de prioridade.

Além disso, o heap também pode ser útil quando temos que remover repetidamente um objeto de dados com os valores de prioridade mais altos ou mais baixos.

No próximo capítulo, discutiremos os conceitos de Hashing e Tabelas de Símbolos.

Exercícios:

1. Qual será a complexidade de tempo para excluir um elemento arbitrário do min-heap?
2. Qual será a complexidade de tempo para encontrar o k-ésimo menor elemento do min-heap?
3. Qual será a complexidade de tempo no pior caso para determinar o menor elemento de um binary max-heap e binary min-heap?
4. Qual será a complexidade de tempo para criar um max-heap que combina dois max-heap cada um de tamanho n?
5. A travessia de ordem de nível do max-heap é 12, 9, 7, 4 e 2. Após inserir os novos elementos 1 e 8, qual será o max-heap final e a travessia de ordem de nível do max-heap final?
6. Qual dos seguintes é um binary max-heap?

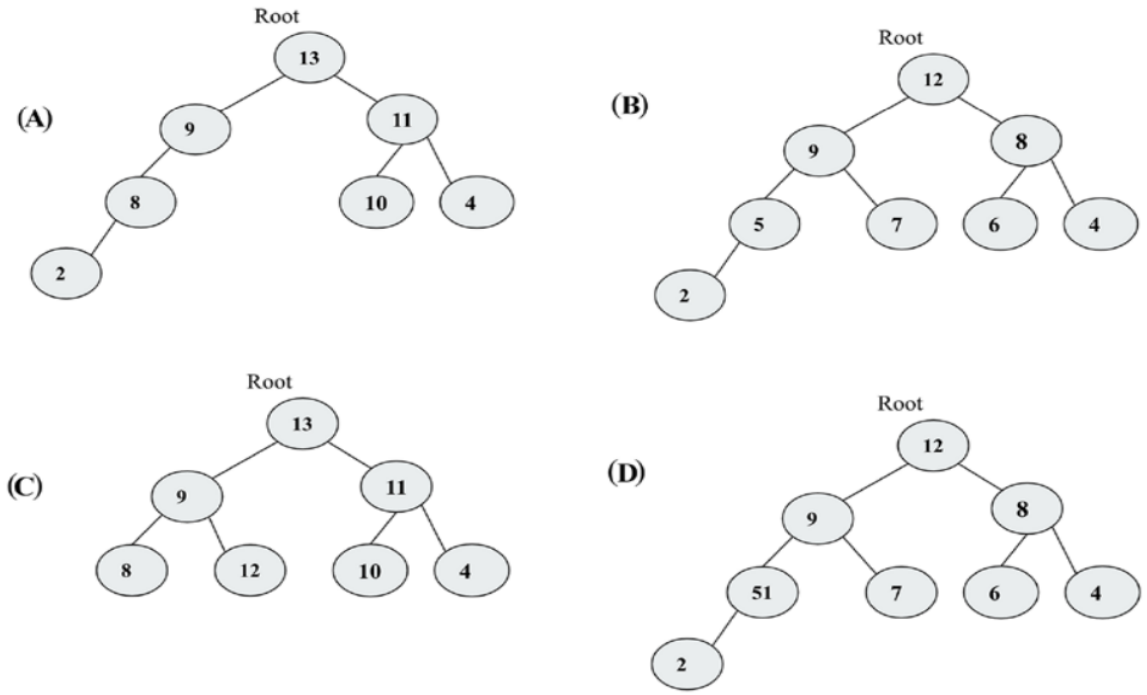

Figure 7.21: Example trees