Section A4 Heaps&Hash Maps

Tree Terminology

- **Node:** Nodes represent any value or object stored in the tree.
- **Edge:** The link that connects any two nodes is called an edge. If a tree has n number of nodes, there are exactly (n-1) edges.
- **Root:** The root is the base node of the tree. This node is usually drawn at the top of a graphic representation of a tree. In any tree, there must be only one root node.
- **Parent:** A parent node is any node that contains 1 to n child nodes. In other words, the parent node has a branch from it to any other node.
- **Child:** The node which is a descendant of some node is called a child node. Every node in a tree, except its root node, are child nodes.
- **Siblings:** Nodes that belong to the same parent are called siblings.
- **Internal Node:** Nodes that have at least one child is called internal nodes. They are also known as non-terminal nodes.
- **Leaf:** Nodes that do not have any child is called leaf nodes. They are also known as external nodes or terminal nodes.
- **Level:** In a tree, each step from top to bottom is called its level. The level starts with 0 and increments by 1 at each step.
- **Subtree:** In a tree, each child forms a subtree on its parent node.

Introduction to Heaps

Heap is a ***complete binary tree***. This means that all levels of the tree have both left and right child nodes (with the exception of the last-level leaves), and the last level has all nodes orientated to the left.

A *full binary tree* is a tree in which every node other than the leaf nodes has two children. Therefore, a complete binary tree is just like a full binary tree, except for the following two characteristics.

1. All the leaf nodes must lean towards the left side.
2. The last leaf node may not have the right sibling. In other words, a complete binary tree does not need to be a full binary tree.

Heap Order Property

Heaps are **ordered** in either ascending order (in a min heap), or descending order (in a max heap). Therefore, a heap is either a ***min heap*** or a ***max heap***.

- Min Heap: In a min-heap, the parent node is lesser in value than the children nodes.
- Max Heap: In a max-heap, the parent node is greater than the children nodes.

The min heap lets us quickly extract the minimum value of a tree, whereas the max heap lets us quickly extract the maximum value of a tree.

Insertion of a Max Heap

When inserting a new node to a max heap, the below-mentioned steps needs to be followed.

- Step 1: Create a new node at the end of the max heap.
- Step 2: Assign the new value to the node.
- Step 3: Compare the value of this child node with its parent.
- Step 4: If value of parent is less than child, then swap them.
- Step 5: Repeat steps 3 and 4 until heap properties hold.

When inserting a new node to min heap, same steps are followed, expect the value of the parent node to be less than that of the child node.

Deletion of a Max Heap

Deletion in max (or min) heap always happens at the root, where we remove the maximum (or minimum) value. The following are the steps that needs to be followed when deleting the root f a max heap.

- Step 1: Remove root node.
- Step 2: Move the last element of last level to root.
- Step 3: Compare the value of this node with its child nodes.
- Step 4: If value of parent is less than child, then swap them.
- Step 5: Repeat steps 3 and 4 until heap properties hold.

When deleting the root node of a min heap, same steps are followed, expect the value of the parent node to be less than that of the child node.

Building a Heap

Since heaps are complete binary trees, they may be stored using arrays.
The children of any element of the array can be calculated from the index of the parent using the following two formulae.

```
leftChildIndex = 2 * parentIndex + 1

rightChildIndex = 2 * parentIndex + 2
```

It is also possible to discover where the parent is located, given a child's index using the following formula.

```
parentIndex = (childIndex - 1)//2
```

Note that // represents integer division.

Heap Operations

We use Python's inbuilt library named **heapq** to build heaps. Note that the heapq library provides the *min heap implementation*.

- ***heapify(iterable)***: This function is used to convert a list into a heap data structure.
- ***heappush(heap, element)***: This function is used to insert an element into a heap. The order of the heap is adjusted, thus, the heap properties are maintained.
- ***heappop(heap)***: This function is used to remove and return the smallest element from a heap. The order of the heap is adjusted, thus, the heap properties are maintained.

- ***heappushpop(heap, element)***: This function combines both push and pop operations to increase efficiency. The order of the heap is adjusted, thus, the heap properties are maintained.
- ***heapreplace(heap, element)***: This function also does both push and pop operations in one statement. However, the only difference of heapreplace() (compared to heappushpop()) is that the element is first popped, then the element is pushed. Therefore, heapreplace() function returns the original smallest value in the heap regardless of the pushed element as opposed to the heappushpop() function.
- ***nlargest(k, iterable, key = fun)***: This function is used to return the k largest elements from the iterable specified and satisfying the key if mentioned.
- ***nsmallest(k, iterable, key = fun)***: This function is used to return the k smallest elements from the iterable specified and satisfying the key if mentioned.

Min Heap Implementation

Let's create a min heap using heapq and demonstrate the following operations.

- ***heapify(iterable)***
- ***heappush(heap, element)***
- ***heappop(heap)***

```
#functions: heapify(), heappush() and heappop()

#importing "heapq" to implement heap
import heapq

#initialising list
li = [5, 7, 9, 1, 3]

#using heapify to convert list into heap
heapq.heapify(li)

#printing created heap
print ("The created heap is : ",end="")
print (list(li))

#using heappush() to push elements into heap
#pushes 4
heapq.heappush(li,4)

#printing modified heap
print ("The modified heap after push is : ",end="")
print (list(li))

#using heappop() to pop smallest element
print ("The popped and smallest element is : ",end="")
print (heapq.heappop(li))
```

we have a class with the following attributes, ‘name‘, ‘designation‘, ‘yos‘ (years of service), ‘salary‘. The objects of this class have to be maintained in min-heap based on ‘yos‘ (years of service). In this instance, we need to override the '<' operator such that it compares the 'yos' of each employee and returns true or false. The heapq module arranges the objects in min-heap order based on this returned boolean value.

```
import heapq as hq

class employee:

    def __init__(self, n, d, yos, s):
        self.name = n
        self.des = d
        self.yos = yos
        self.sal = s

    def print_me(self):
        print("Name :", self.name)
        print("Designation :", self.des)
        print("Years of service :", str(self.yos))
        print("salary :", str(self.sal))

   #override the comparison operator
    def __lt__(self, nxt):
        return self.yos < nxt.yos

#creating objects
e1 = employee('John', 'manager', 3, 24000)
e2 = employee('Danise', 'programmer', 2, 15000)
e3 = employee('Kelly', 'Analyst', 5, 30000)
e4 = employee('Smith', 'programmer', 1, 10000)

#list of employee objects
emp = [e1, e2, e3, e4]

#converting to min-heap based on yos
hq.heapify(emp)

#printing the results
for i in range(0, len(emp)):
    emp[i].print_me()
    print()
```

Max Heap Implementation

We could simply implement a max heap using **heapq** by multiplying each value by -1

```python
import heapq

class MaxHeap:

    #initialize the max heap
    def __init__(self, data=None):
        if data is None:
            self.data = []
        else:
            self.data = [-i for i in data]
            heapq.heapify(self.data)

    #push item onto max heap, maintaining the heap invariant
    def push(self, item):
        heapq.heappush(self.data, -item)

    #pop the largest item off the max heap, maintaining the heap invariant
    def pop(self):
        return -heapq.heappop(self.data)

    #pop and return the current largest value, and add the new item
    def replace(self, item):
        return heapq.heapreplace(self.data, -item)

    #return the current largest value in the max heap
    def top(self):
        return -self.data[0]

list1 = [7, 4, 6, 3, 9, 1]

#build a max heap from all elements in the list
pq = MaxHeap(list1)
```

For objects, you can directly use the heapq module to obtain a max heap. The idea is to override the magic method **__lt__** to make our class work with max heaps.

```
import heapq

class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    #override the less-than operator __lt__ to make Pair class work with max heap
    def __lt__(self, other):
        return self.x > other.x

    def __repr__(self):
        return f'({self.x}, {self.y})'

#build a max heap of Pair
pq = [Pair(7, 0), Pair(3, 3), Pair(9, 4), Pair(4, 1), Pair(6, 2), Pair(1, 5)]
heapq.heapify(pq)

# run until max heap is empty
while pq:
    # print the next maximum element from the heap
    print(heapq.heappop(pq))
```

```
(9, 4)
(7, 0)
(6, 2)
(4, 1)
(3, 3)
(1, 5)
```

Priority Queues

There are different ways of implementing priority queues in Python. Let's explore the following three methods in this section.

- Using lists
- Using the heapq module
- Using the queue module

Using Lists

```
customers = []
customers.append((2, "Harry")) #no sort needed here because 1 item.
customers.append((3, "Charles"))
customers.sort()
#need to sort to maintain order
customers.append((1, "Riya"))
customers.sort()
#need to sort to maintain order
customers.append((4, "Stacy"))
customers.sort()
while customers:
     print(customers.pop(0))
```

Using the heapq Module

Heaps are commonly used to implement priority queues

```
import heapq
customers = []
heapq.heappush(customers, (2, "Harry"))
heapq.heappush(customers, (3, "Charles"))
heapq.heappush(customers, (1, "Riya"))
heapq.heappush(customers, (4, "Stacy"))
while customers:
    print(heapq.heappop(customers))
```

Using the queue Module

let's see how to use the **PriorityQueue** from the queue module. It is different from the heapq module in two key ways. Firstly, it is synchronised, thus, supporting concurrent processes. Secondly, it is a class interface, instead of the function based interface of heapq . Therefore, PriorityQueue is the classic OOP style of implementing priority queues.

```
from queue import PriorityQueue
customers = PriorityQueue()
customers.put((2, "Harry"))
customers.put((3, "Charles"))
customers.put((1, "Riya"))
customers.put((4, "Stacy"))
while customers:
     print(customers.get())
```

Hash Maps

Hash Functions

The purpose of a hash function is to create a key that will be used to access the stored value.
The following is an example of a simple hash function where

- h(k) is the hash function. It accepts a parameter k.
- The parameter k is the value that you pass in to calculate the key.
- k1 % m is the algorithm of our hash function (e.g., k1 is the value you want to store, and m is the list size). We use the modulus (i.e., %) operator to calculate the key.

```
h(k) = k1 % m
```

Hash Collision can be solved by using *chaining* or *probing*.
In probing, if a collision occurs, the element will be stored in an available slot in the hash map. We use the following methods to facilitate probing.

- Linear probing: This method searches for empty slots linearly starting from the position where the collision occurred and moving forward. If the end of the list is reached without finding an empty slot, the search will wrap around to the beginning of the hash map and continue from

there. If an empty slot is not found before reaching the point of collision, this means that the hash map is full.
- Quadratic probing: This method uses quadratic polynomial expressions to find the next available slot in the hash map.
- Double Hashing: This method uses a secondary hash function algorithm to find the next available slot in the hash map.

Hash Operations

- Insertion: This is used to store values in the hash map. When a new value is stored in the hash map, it is assigned an index number (or hash key), which is generated using a hash function. This hash function resolves any collisions that occur when calculating the hash key.
- Search: This operation is used to look up values in the hash map using an index number. The search operation returns the value that is associated with the search index number.
- Deletion: This operation is used to remove a value from a hash map using an index number. Once a value has been deleted, the slot of the given index number will be free and will be used to store other values during the insertion operation.

***Hash* Map Implementation**

```
class HashMap:

    #create empty bucket list of given size
    def __init__(self, size):
        self.size = size
        self.hash_map = self.create_buckets()

    def create_buckets(self):
        return [[] for _ in range(self.size)]

    #insert values into hash map
    def set_val(self, key, val):

        #get the index from the key using hash function
        hashed_key = hash(key) % self.size

        #get the bucket corresponding to index
        bucket = self.hash_map[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            #check if the bucket has same key as the key to be inserted
            if record_key == key:
                found_key = True
                break

        #if the bucket has same key as the key to be inserted, update the key value.
cket
        if found_key:
            bucket[index] = (key, val)
        else:
            bucket.append((key, val))
```

```
    #return searched value with specific key
    def get_val(self, key):

        #get the index from the key using hash function
        hashed_key = hash(key) % self.size

        #get the bucket corresponding to index
        bucket = self.hash_map[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            #check if the bucket has same key as the key being searched
            if record_key == key:
                found_key = True
                break

        #if the bucket has same key as the key being searched, return the value found.
        if found_key:
            return record_val
        else:
            return "No record found"

    #remove a value with specific key
    def delete_val(self, key):

        #get the index from the key using hash function
        hashed_key = hash(key) % self.size

        #get the bucket corresponding to index
        bucket = self.hash_map[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            #check if the bucket has same key as the key to be deleted
            if record_key == key:
                found_key = True
                break
        if found_key:
            bucket.pop(index)
        return

    #to print the items of hash map
    def __str__(self):
        return "".join(str(item) for item in self.hash_map)


hash_map = HashMap(10)
```