HASHMAPS implementation

Separate Chaining

Whenever there is a collision, for each index in our array, we have node objects. We add items to our node objects. *So basically, we handle collision by having list of linked lists.*

```
Class Node:
    Def __init__(self,key,val):
        self.key=key
        self.val=val
        self.next=None
Class SeparateChaining:
# runtime and space - O(capacity)
    Def __init__(self, capacity): → how many indices we have in our array
        self.map=[Node('Dummy','dummy') for _ in range(capacity) ]--> list comprehension


    Def hashfunction(self,key):
        Return key% len(self.map)

    Def put(self,key,val):
    #O(N)/capacity- RT
    #O(1)- space
        idx= self.hashfunction(key)
        cur= self.map[idx]
        While cur.next:
            If cur.next.key==key:
                cur.net.val=val
            cur=cur.next
        cur.next=Node(key,val)


    Def get(self,key):
        #O(N)/capacity- RT
        #O(1)- space
        idx=self.hashfunction(key)
        cur=self.map[idx]
        While cur.next:
            If cur.next.key==key:
                Return cur.next.val
```

```
                cur=cur.next


        Def delete(self,key):
                #we have to keep track of previous
                idx= self.hashfunction(key)
                prev=self.map[idx]
                cur=prev.next
                While cur:
                        If cur.key==key:
                                prev.next=cur.next
                        prev=cur
                        cur=cur.next

        Def __str__(self):--> output everything in our map
#O(N)-runtime
# O(N)- space
                out=""
                For idx in range(len(self.map)):
                        cur=self.map[idx].next→ we don't care about dummy node
                        While cur:
                                out+= str(cur.val) + " "
                                cur=cur.next
                        out+="/n"
                Return out



If __name__=='__main__':
map=SeparateChaining(3)
For i range(10):
                map.put(i,i*2)
Print(map)
print(map.get(4))
```

HEAPS

```
Class Maxheap:
        Def __init__(self):
                self.arr=[None]
```

```
Def insert(self,val):
        self.arr.append(val)
        self.swim()

Def swim(self):
        idx=len(arr)-1
        while  (idx//2>0 and self.arr[idx//2]<self.arr[idx]):
                self.arr[idx//2],self.arr[idx]=self.arr[idx],self.arr[idx//2]
                idx=idx//2

Def get_max(self):
        Assert not self.is_empty()
        Return self.arr[1]

Def del_max(self):
        Assert not self.is_empty()
        Self.arr[1], self.arr[len(self.arr)-1] = self.arr[len(self.arr)-1], self.arr[1]
        val= self.arr.pop()
        self.sink()
        Return val

Def sink(self):
        idx=1
        While (idx*2< len(self.arr)):
                cur=self.arr[idx]
                left=self.arr[idx*2]
                right= float("-inf")
                If ((idx*2)+1 < len(self.arr)):
                        right=self.arr[(idx*2)+1]
                If cur>=left and cur>=right:
                        Return

                If left>cur:
                        self.arr[idx],self.arr[idx*2]=self.arr[idx], self.arr[idx*2]
                        idx=idx*2
                Else:
                        self.arr[idx],self.arr[(idx*2) +1]=self.arr[idx], self.arr[(idx*2)+1]
                        idx=(idx*2)+1

Def is_empty(self):
        Return len(self.arr)==1

Def heapsort(arr):-----> sort arrays using maxheap
```

```
’       pq=Maxheap()
        For item in arr:
                pq.insert(item)

        idx=len(arr)-1

        While pq.is_empty is False:
                        arr[idx]=self.del_max()
                        idx-=1
```