برای ساختن جدول درهم سازی بسته در python می تواین از class استفاده کنیم. در زیر نحوه ساخت این کلاس آمده است.

```
class HashedTable:
    def __init__(self, size: int) -> None:
        self.size = size
        self.table = [None] * size
        self.filled_cells = 0
        self.insertion_counter = 0
        self.search_counter = 0
```

در تابع innit که در زمان ساخت یک شی از این کلاس صدا زده خواهد شد مقادیر اولیه این جدول را مانند اندازه آن، یک آرایه برای ذخیره اطلاعات و تعداد خانه‌های پر را ذخیره می‌کنیم.

```
class HashedTable:
    ...
    def add(self, value: str):
        if self.filled_cells >= self.size:
        raise Exception("HashTable is full")
        self.table[self.get_insertion_index(value)] = value
        self.filled_cells += 1
```

تابع بعدی، اضافه کردن یک مقدار به این جدول را انجام می‌دهد البته پس از بررسی پر نبودن جدول. تابع `get_insertion_index` پس از این تابع تعریف خواهد شد و اندیس خانه‌ای که می‌توانیم مقدار جدید را در آن ذخیره کنیم به ما خواهد داد.

```
class HashedTable:
    ...
    def get_insertion_index(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
            if self.table[i] is None or self.table[i] is False:
                self.insertion_counter += 2 * abs(start_index - i)
                return i
```

این تابع همانطور که گفته شد، اندیس خانه‌ای که این کلید می‌تواند در آن ذخیره شود را به ما خواهد داد. این کار با یک حلقه و بررسی خانه‌ها، به ترتیب و از خانه‌ای شروع خواهد شد که تابع hash به ما داده است، زیرا این اولین گزینه ما برای ذخیره این کلید است و اگر پر باشد، خانه‌های پس از آن بررسی خواهند شد. خانه‌هایی که از جدول حذف شده‌اند با $False$ نشانه گذاری شده‌اند پس ما آن‌ها را نیز خالی در نظر می‌گیریم زیرا این نشانه‌گذاری برای جستجو است، نه اضافه کردن.

```
class HashedTable:
    ...
    def hash(self, value: int):
        return value % self.size
```

برای hash کردن یا درهم سازی نیز از باقیمانده تقسیم عدد بر اندازه جدول استفاده می‌کنیم.

```
class HashedTable:
    ...
    def search(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
```

```
        for i in range(start_index, hash_index):
            if self.table[i] is None:
                self.search_counter += abs(start_index - i) + 1
                return False
            elif self.table[i] == value:
                if i >= 0:
                    return i
                else:
                    return len(self.table) + i
        self.search_counter += len(self.table)
        return False
```

برای جستجو در این جدول از گزینه اولمان که از تابع hash گرفته‌ایم شروع کرده و خانه‌ها را به ترتیب بررسی می‌کنیم. اگر مقدار خانه None باشد یعنی در این خانه هرگز مقداری ذخیره نشده‌است و نتیجه جستجو منفی خواهد بود. اگر مقدار خانه برابر False باشد یعنی مقدار این خانه قبلا حذف شده است و به جستجو ادامه می‌دهیم و اگر مقدار خانه برابر کلید جستجو باشد، جستجو موفقیت آمیز بوده‌است و اندیس خانه به عنوان نتیجه جستجو بازگردانده خواهد شد. و اگر جستجو بی‌نتیجه باشد False بازگردانده خواهد شد.

```
class HashedTable:
    ...
    def delete(self, value: str):
        index = self.search(value)
        if index:
            self.table[index] = False
            self.filled_cells -= 1
```

و به عنوان آخرین تابع این کلاس، برای حذف کردن مقادیر ابتدا آن را جستجو کرده و اگر این جستجو موفقیت آمیز بود، آن را با قراردادن False به جای مقدار اصلی حذف کرده و یک واحد از خانه‌های اشغال شده کم می‌کنیم.

حال با اضافه کردن همه‌ی توابع کلاس HashedTable به صورت زیر خواهد بود:

Listing 1: HashedTable Implementation

```
class HashedTable:
    def __init__(self, size: int) -> None:
        self.size = size
        self.table = [None] * size
        self.filled_cells = 0
        self.insertion_counter = 0
        self.search_counter = 0

    def add(self, value: int):
        if self.filled_cells >= self.size:
            raise Exception("HashTable is full")
        self.table[self.get_insertion_index(value)] = value
        self.filled_cells += 1

    def get_insertion_index(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
```

```
            if self.table[i] is None or self.table[i] is False:
                self.insertion_counter += 2 * abs(start_index - i)
                return i

    def hash(self, value: int):
        return value % self.size

    def search(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
            if self.table[i] is None:
                self.search_counter += abs(start_index - i) + 1
                return False
            elif self.table[i] == value:
                if i >= 0:
                    return i
                else:
                    return len(self.table) + i
        self.search_counter += len(self.table)
        return False

    def delete(self, value: str):
        index = self.search(value)
        if index:
            self.table[index] = False
            self.filled_cells -= 1
```

با بررسی تعداد مقایسه‌های لازم برای جستجوی ناموفق و درج m عنصر در جدول درهم سازی به نتایج زیر دست می‌یابیم که نشان می‌دهد نتایج تجربی، نتایج نظری بخش‌های الف و ب را تایید می‌کند:

```
Testing unseccussfull search comparisons ...

>> n is 10^1 and alpha is 0.5:
average uncessufull search comparisons is: 1.6
in theory the average for search should be: 2.5
>> n is 10^2 and alpha is 0.5:
average uncessufull search comparisons is: 2.12
in theory the average for search should be: 2.5
>> n is 10^3 and alpha is 0.5:
average uncessufull search comparisons is: 2.405
in theory the average for search should be: 2.5
>> n is 10^4 and alpha is 0.5:
average uncessufull search comparisons is: 2.4926
in theory the average for search should be: 2.5
>> n is 10^5 and alpha is 0.5:
average uncessufull search comparisons is: 2.48968
in theory the average for search should be: 2.5
>> n is 10^6 and alpha is 0.5:
```

```
average uncessufull search comparisons is: 2.497765
in theory the average for search should be: 2.5


Testing unseccussfull add comparisons ...

>> n is 10^1:
uncessufull add comparisons are 12
in theory it should be: 39.633272976060105
>> n is 10^2:
uncessufull add comparisons are 820
in theory it should be: 1253.3141373155001
>> n is 10^3:
uncessufull add comparisons are 52682
in theory it should be: 39633.27297606011
>> n is 10^4:
uncessufull add comparisons are 2145782
in theory it should be: 1253314.1373155
>> n is 10^5:
uncessufull add comparisons are 35076828
in theory it should be: 39633272.97606011
>> n is 10^6:
uncessufull add comparisons are 1190299018
in theory it should be: 1253314137.3155
```