```
        while  j >= gap and arr[j-gap] >temp:
          print("Inside While:",arr,"\n")
          arr[j] = arr[j-gap]
          j = j-gap
        arr[j] = temp
        print("Outside While:",arr,"\n")
    gap //= 2
arr=[12,34,54,2,3]
shellSort(arr)
print(arr)
```

Output:
[2, 3, 12, 34, 54]

➢ **Hashtables:**

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables.

The Keys in the dictionary satisfy the following requirements.

- **The keys of the dictionary are hashable i.e. that is generated by hashing function which generates unique result for each unique value supplied to the hash function.**
- **The order of data elements in a dictionary is not fixed.**

Compiled by: Prashant Dave
HOD

So we can see the implementation of hash table by using the dictionary data types as below.

Accessing Values in Dictionary
To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example:
```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

# Accessing the dictionary with its key
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

Output:
```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary
You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

Example:
```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Output:
```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements
You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement. –

Example:
```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary
```

Compiled by: Prashant Dave
HOD

```
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Output:

```
dict['Age']:

Traceback (most recent call last):
  File "test.py", line 8, in
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Creating Own Hash Table:

Hash Table is a data structure where data are stored in an associative manner (in key, value format). The key/index is unique. This kind of storage makes it easier to find the data later on.

Hash Table stores data into an array format. It uses a hashing function that generates a slot or an index to store/insert any element or value. Following is the way for creating own hashing function and hash table.

Hash Function / Hashing

This deals with generating slot or index to any "key" value. Perfect hashing or perfect hash function is the one which assigns a unique slot for every key value. Sometimes, there can be cases where the hash function generates the same index for multiple key values. The size of the hash table can be increased to improve the perfection of the hash function.

Following is creating a hash table of size 10 with empty data.

Example:

```
hash_table = [None] * 10
print (hash_table)
```

Output:

```
[None, None, None, None, None, None, None, None, None, None]
```

Below is a hash function that returns the modulus of the length of the hash table. Here the length of the hash table is 10.

Modulo operator (%) is used in the hashing function. The % (modulo) operator yields the remainder from the division of the first argument by the second.

Compiled by: Prashant Dave
HOD

Example:

```
def hashing_func(key):
    return key % len(hash_table)

print (hashing_func(10)) # Output: 0
print (hashing_func(20)) # Output: 0
print (hashing_func(25)) # Output: 5
```

Inserting Data into Hash Table

Here's a simple implementation of inserting data/values into the hash table. We first use the hashing function to generate a slot/index and store the given value into that slot.

Example:

```
def insert(hash_table, key, value):
    hash_key = hashing_func(key)
    hash_table[hash_key] = value

insert(hash_table, 10, 'Nepal')
print (hash_table)
```

Output:

['Nepal', None, None, None, None, None, None, None, None, None]

```
insert(hash_table, 25, 'USA')
print (hash_table)
```

Output:

['Nepal', None, None, None, None, 'USA', None, None, None, None]

Collision

A collision occurs when two items/values get the same slot/index, i.e. the hashing function generates same slot number for multiple items. If proper collision resolution steps are not taken then the previous item in the slot will be replaced by the new item whenever the collision occurs.

Example:

In the example code above, we have inserted items Nepal and USA with key 10 and 25 respectively. If we try to insert a new item with key 20 then the collision occurs because our hashing function will generate slot 0 for key 20. But, slot 0 in the hash table has already been assigned to item 'Nepal'.

```
insert(hash_table, 20, 'India')
print (hash_table)
```

Output:

['India', None, None, None, None, 'USA', None, None, None, None]

Compiled by: Prashant Dave
HOD

Collision Resolution

There are generally two ways to resolve a collision:

1. Linear Probing
2. Chaining

1. Linear Probing

One way to resolve collision is to find another open slot whenever there is a collision and store the item in that open slot. The search for open slot starts from the slot where the collision happened. It moves sequentially through the slots until an empty slot is encountered. The movement is in a circular fashion. It can move to the first slot while searching for an empty slot. Hence, covering the entire hash table. This kind of sequential search is called Linear Probing.

2. Chaining

The other way to resolve collision is Chaining. This allows multiple items exist in the same slot/index. This can create a chain/collection of items in a single slot. When the collision happens, the item is stored in the same slot using chaining mechanism.

While implementing Chaining in Python, we first create the hash table as a nested list (lists inside a list).

Example:

```
hash_table = [[] for _ in range(10)]
print (hash_table)
```

Output:

```
[[], [], [], [], [], [], [], [], [], []]
```

The hashing function will be the same as we have done in above example.

```
def hashing_func(key):
  return key % len(hash_table)

print (hashing_func(10)) # Output: 0
print (hashing_func(20)) # Output: 0
print (hashing_func(25)) # Output: 5
```

Change the insert function. Use **append()** function to insert key-value pairs in the hash table.

Example:

```
def insert(hash_table, key, value):
    hash_key = hashing_func(key)
    hash_table[hash_key].append(value)

insert(hash_table, 10, 'Nepal')
print (hash_table)
```

Output:

```
[['Nepal'], [], [], [], [], [], [], [], [], []]
```

```
insert(hash_table, 25, 'USA')
print (hash_table)
```

Output:

```
[['Nepal'], [], [], [], [], ['USA'], [], [], [], []]
```

```
insert(hash_table, 20, 'India')
print (hash_table)
```

Output:

```
[['Nepal', 'India'], [], [], [], [], ['USA'], [], [], [], []]
```

For more detail refer to following link:
http://blog.chapagain.com.np/hash-table-implementation-in-python-data-structures-algorithms/

References:

http://interactivepython.org/courselib/static/pythonds/SortSearch/searching.html
https://www.geeksforgeeks.org/linear-search/
http://www.discoversdk.com/blog/searching-algorithms-in-python
https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html#searching-algorithms

Compiled by: Prashant Dave
HOD