3. Hashing Motivation and Terminology:

a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.
If *n* is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?

average case *O*(n)

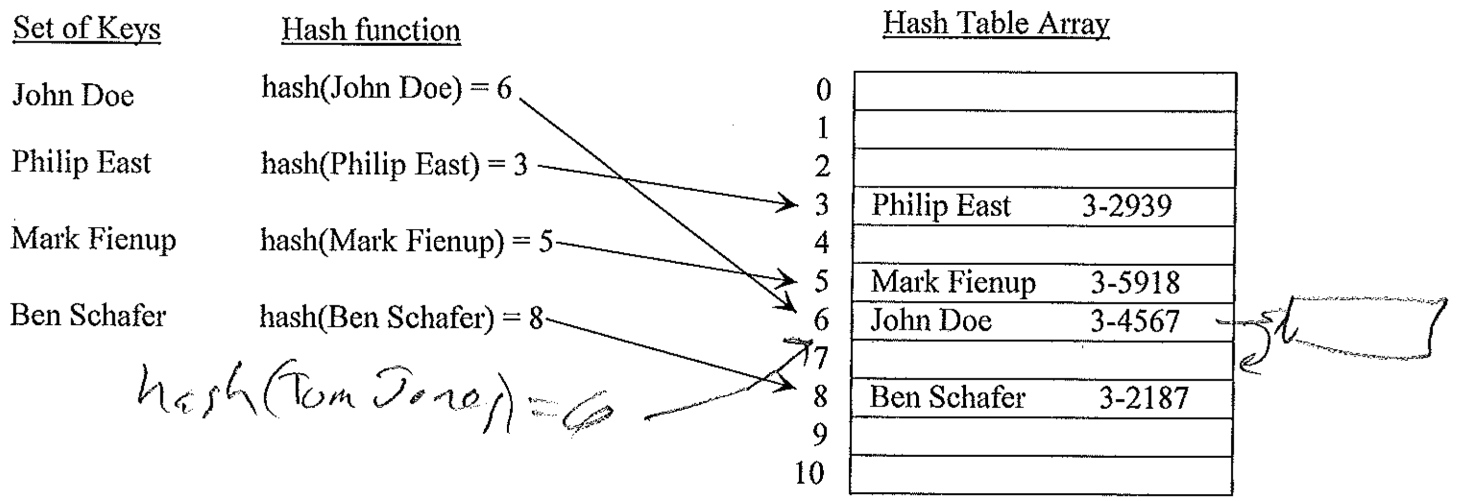
worst case *O*(n)

b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.
If *n* is the number of items being searched, what is the average and worst case big-oh notation for a search?

average case *O*($\log_2 n$)

worst case *O*($\log_2 n$)

Hashing tries to achieve average constant time (i.e., *O*(1)) searching by using the target's value to calculate where in the array/Python list (called the *hash table*) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target's *home address*) is the job of the *hash function.* A *perfect hash function* would take your set of target values and map each to a unique array index.



a) If *n* is the number of items being searched and we had a perfect hash function, what is the average and worst case big-oh notation for a search?

average case *O*(1)

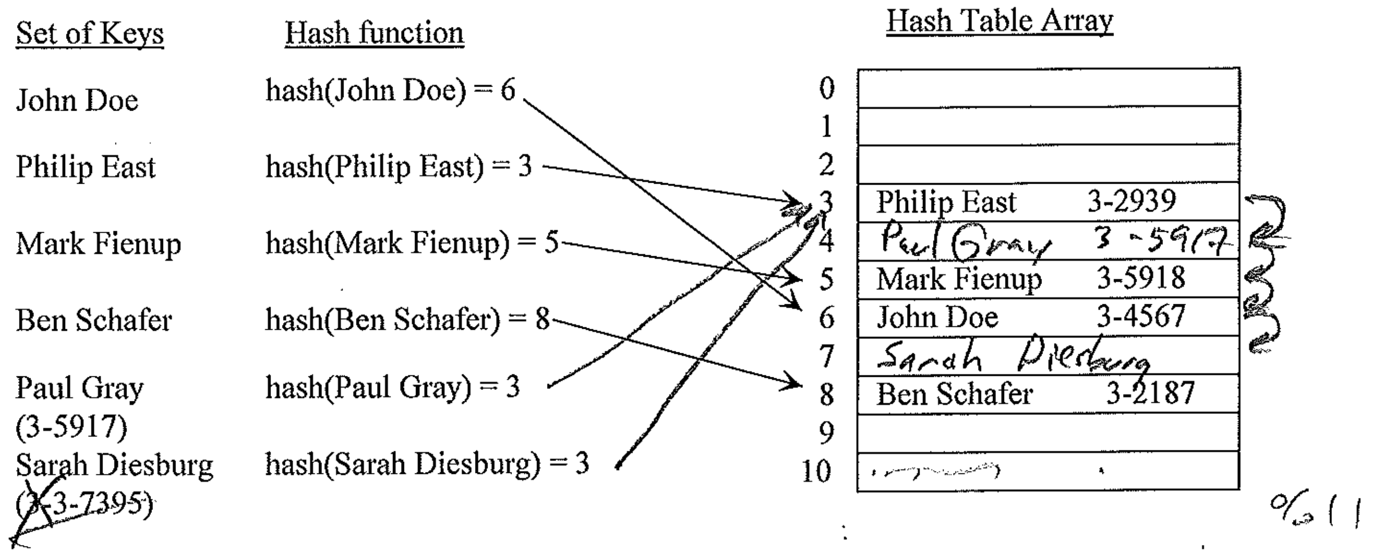
worst case *O*(1)

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same hash-table index, called a *collision.*

Collisions are handled by two approaches:

- *open-address* with some *rehashing* strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.
- *chaining, closed-address*, or *external chaining*: all target values hashed to the same home address are stored in a data structure (called a *bucket*) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)

5. Consider the following examples using *open-address* approach with a simple rehashing strategy of *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.

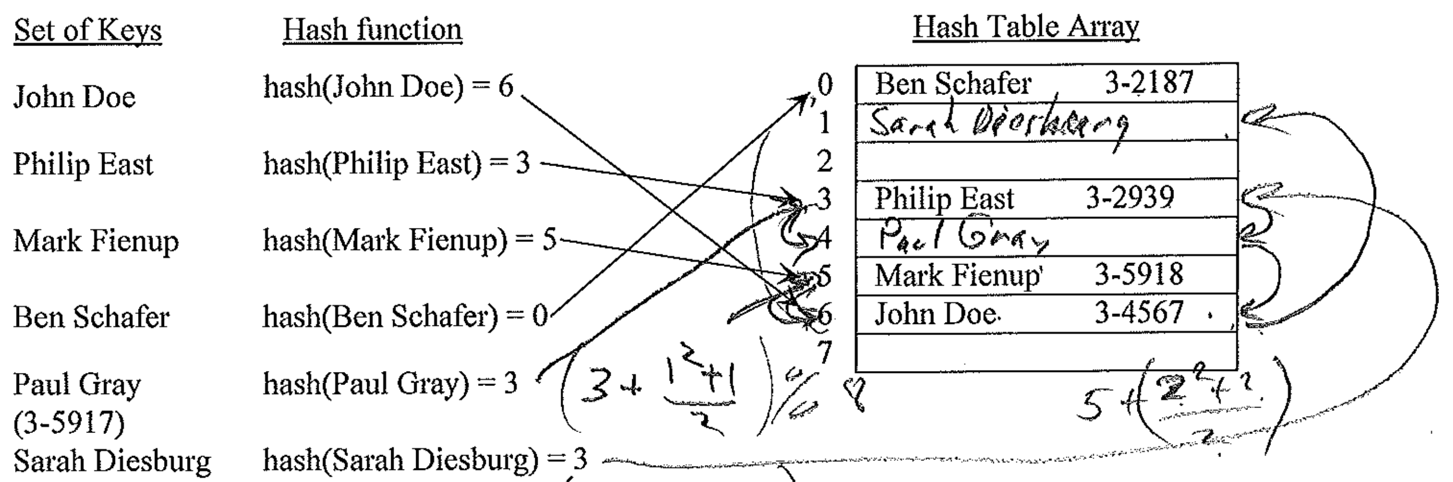


a) Assuming open-address with linear probing where would Paul Gray and then Sarah Diesburg be placed?

Common rehashing strategies include the following.

Rehash Strategy	**Description**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)
quadratic probing	Check the square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))//2) % (hash table size), where the hash table size is a power of 2
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size

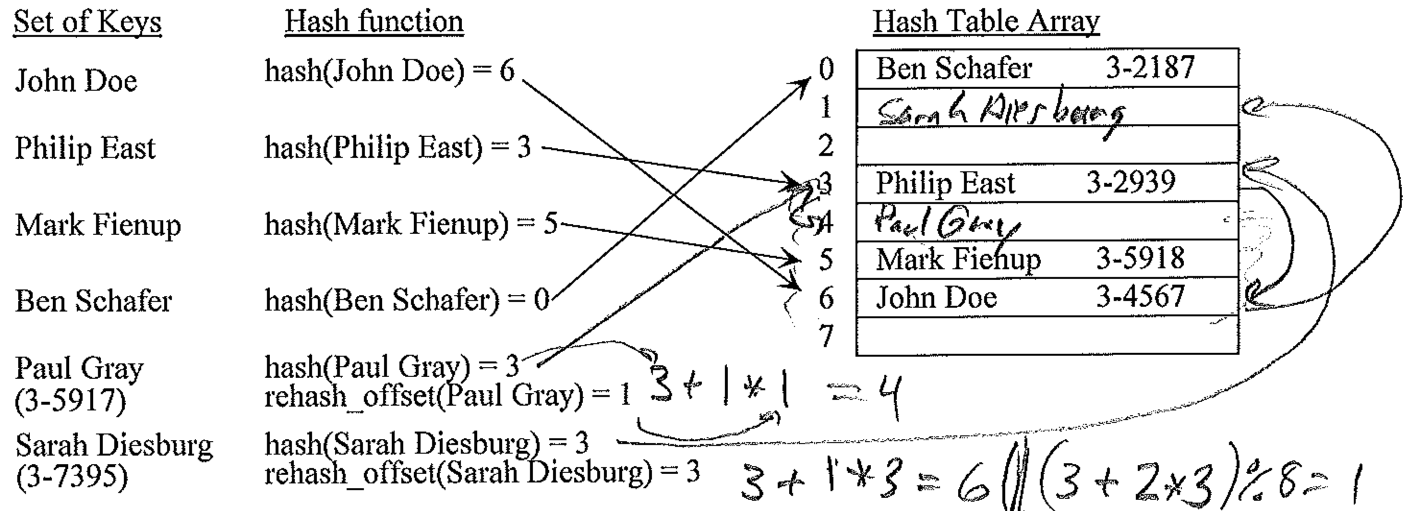
b) Assume quadratic probing, insert "Paul Gray" and "Sarah Diesburg" into the hash table.



$(3 + \frac{2^2+2}{2}) \% 8 = (3+3) \% 8 = 6$

$(3 + \frac{3^2+3}{2}) \% 8 = (3+6) \% 8 = 9 \% 8 = 1$

c) Assume double hashing, insert "Paul Gray" and "Sarah Diesburg" into the hash table.



Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 0
Paul Gray (3-5917)	hash(Paul Gray) = 3 rehash_offset(Paul Gray) = 1
Sarah Diesburg (3-7395)	hash(Sarah Diesburg) = 3 rehash_offset(Sarah Diesburg) = 3

Hash Table Array

0	Ben Schafer	3-2187
1	Sarah Diesburg	
2		
3	Philip East	3-2939
4	Paul Gray	
5	Mark Fienup	3-5918
6	John Doe	3-4567
7		

$3 + 1 * 1 = 4$

$3 + 1 * 3 = 6 \,||\, (3 + 2 * 3) \% 8 = 1$

d) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Sarah Diesburg if the table was full? For the above example, (home address + (rehash attempt #) * offset) % (hash table size) would be: (3 + (rehash attempt #) * 3) % 8

Rehash Attempt #	0	1	2	3	4	5	6	7	8	9	10
Address	3	6	1	4	7	2	5	0	3		

e) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- *primary clustering* - keys mapped to a home address follow the same rehash pattern
- *secondary clustering* - rehash patterns from initially different home addresses merge together

Rehash Strategy	**Description**	**Suffers from:** primary clustering	**Suffers from:** secondary clustering
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)	Yes	Yes
quadratic probing	Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2	Yes	No
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size	No	No (?)

6. Let λ be the *load factor* (# item/hash table size). The average probes with **linear probing** for insertion or unsuccessful search is: $\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\lambda)^2}\right)\right)$. The average for successful search is: $\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\lambda)}\right)\right)$.

a) Why is an unsuccessful search worse than a successful search?

$\lambda = 0.5$

$\left(\frac{1}{2}\right)\left(1+\frac{1}{(1-0.5)^2}\right) = \frac{5}{2} = 2.5$

$\left(\frac{1}{2}\right)\left(1+\frac{1}{.5}\right) = \frac{3}{2} = 1.5$

Because for successful searches we stop when we find item, but unsuccessful must search until empty spot found.

The average probes with **quadratic probing** for insertion or unsuccessful search is: $\left(\frac{1}{1-\lambda}\right) - \lambda - \log_e(1-\lambda)$

The average probes with quadratic probing for successful search is: $1 - \left(\frac{\lambda}{2}\right) - \log_e(1-\lambda)$

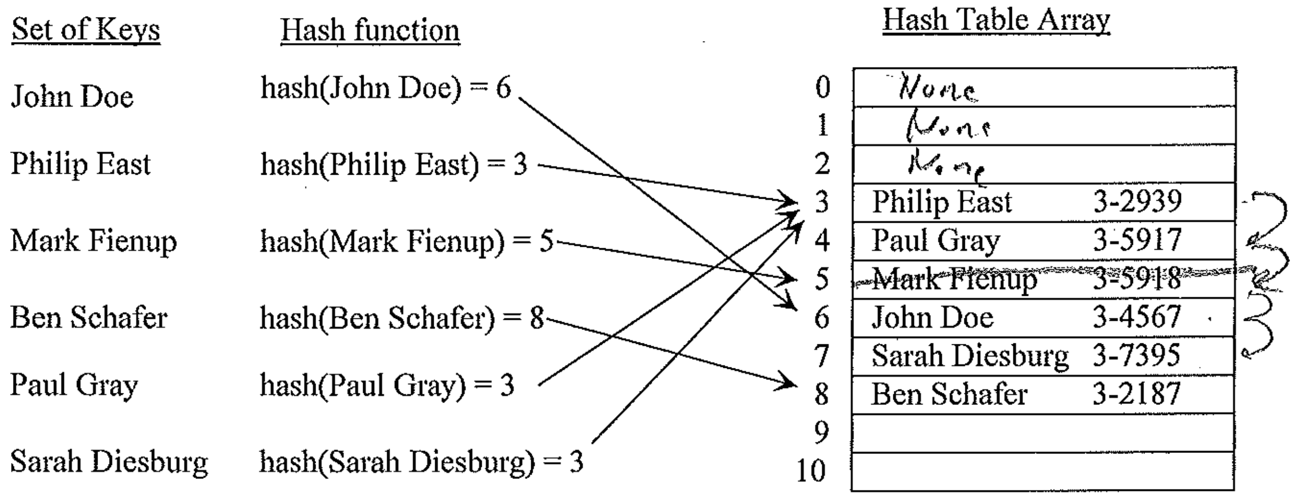
Consider the following table containing the average number probes for various load factors:

Probing Type	**Search outcome**	**Load Factor, λ**				
		0.25	**0.5**	**0.67**	**0.8**	**0.99**
Linear Probing	unsuccessful	1.39	2.50	5.09	13.00	5000.50
	successful	1.17	1.50	2.02	3.00	50.50
Quadratic Probing	unsuccessful	1.37	2.19	3.47	5.81	103.62
	successful	1.16	1.44	1.77	2.21	5.11

b) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

If load factor < 0.5, then we are wasting memory space. If load factor > 0.67, each search takes more probes/time.

7. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following



Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 8
Paul Gray	hash(Paul Gray) = 3
Sarah Diesburg	hash(Sarah Diesburg) = 3

Hash Table Array

0	None	
1	None	
2	None	
3	Philip East	3-2939
4	Paul Gray	3-5917
5	~~Mark Fienup~~	~~3-5918~~
6	John Doe	3-4567
7	Sarah Diesburg	3-7395
8	Ben Schafer	3-2187
9		
10		

a) If "Mark Fienup" is deleted, how will we find Sarah Diesburg? Cannot stop when searching for "Sarah Diesburg" when run across deleted entry. Use a special "deleted" value, say True.

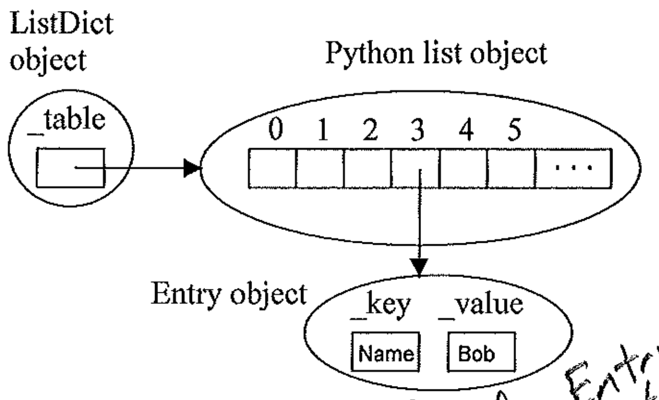
b) How might we fix this problem? "Deleted" special value

= True

"Empty" = None never had a value.

1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

Method call	Class Name	Description
`d = ListDict()`	`__init__(self)`	Constructs an empty dictionary
`d["Name"] = "Bob"`	`__setitem__(self,key,value)`	Inserts a key-value entry if `key` does not exist or replaces the old value with `value` if `key` exists.
`temp = d["Name"]`	`__getitem__(self,key)`	Given a `key` return it value or `None` if `key` is not in the dictionary
`del d["Name"]`	`__delitem__(self,key)`	Removes the entry associated with `key`
`if "Name" in d:` [illegible]	`__contains__(self,key)`	Return `True` if `key` is in the dictionary; return `False` otherwise
`for k in d:`	`__iter__(self)`	Iterates over the keys in the dictionary
`len(d)`	`__len__(self)`	Returns the number of items in the dictionary
`str(d)`	`__str__(self)`	Returns a string representation of the dictionary



Note: need an Entry object with target key to search Python list with index method calls --eq--

```
from entry import Entry

class ListDict(object):
    """Dictionary implemented with a Python list."""

    def __init__(self):
        self._table = []

    def __getitem__(self, key):
        """Returns the value associated with key or
        returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table[index].getValue()
        except:
            return None

    def __delitem__(self, key):
        """Removes the entry associated with key."""
        entry = Entry(key, None)
        try:    # NOTE: Python list index method
                # errors on unsuccessful search
            index = self._table.index(entry)
            self._table.pop(index)
        except:
            return

    def __str__(self):
        """Returns string repr. of the dictionary"""
        resultStr = "{"
        for item in self._table:
            resultStr = resultStr + " " + str(item)
        return resultStr + " }"

    def __iter__(self):
        """Iterates over keys of the dictionary"""
        for item in self._table:
            yield item.getKey()
        raise StopIteration
```

```
class Entry(object):
    """A key/value pair."""

    def __init__(self, key, value):
        self._key = key
        self._value = value

    def getKey(self):
        return self._key

    def getValue(self):
        return self._value

    def setValue(self, newValue):
        self._value = newValue

    def __eq__(self, other):
        if not isinstance(other, Entry):
            return False
        return self._key == other._key

    def __str__(self):
        return str(self._key) + ":" + str(self._value)
```

a) Complete the code for the `__contains__` method.

```
def __contains__(self, key):
```

b) Complete the code for the `__setitem__` method.

```
def __setitem__(self, key, value):
```