9. Hash tables. Open addressing

1. Recap. Separate-chaining symbol table implementation

```python
class SeparateChainingHashST:
    class Node:
        def __init__(self, key, value, next):
            self.key = key
            self.value = value
            self.next = next

    def __init__(self, bucket_count):
        self.arr = np.empty(bucket_count, dtype=self.Node)
        self.bucket_count = bucket_count
```

```python
def find_index(self, key):
    return key.__hash__() % self.bucket_count

def __setitem__(self, key, value):
    index = self.find_index(key)

    node = self.arr[index]
    while node is not None:
        if key == node.key:
            node.value = value
            return
        node = node.next

    self.arr[index] = self.Node(key, value, next=self.arr[index])
```

2. Recap. Separate-chaining symbol table implementation

```python
def __getitem__(self, key):
    index = self.find_index(key)

    node = self.arr[index]
    while node is not None:
        if key == node.key:
            return node.value
        node = node.next

    raise KeyError(key)
```

```python
def delete(self, key):
    index = self.find_index(key)

    node = self.arr[index]
    previous_node = None
    while node is not None:
        if key == node.key:
            if previous_node:
                previous_node.next = node.next
            else:
                self.arr[index] = None
            return node.value
        previous_node = node
        node = node.next

    raise KeyError(key)
```

3. Recap. Separate-chaining symbol table implementation

- Is this hash table bounded (meaning it can only contain some specific number of elements, at most)?

  No. It's not limited to '`bucket_count`' elements, we can put any number of key-value pairs (as long as we don't run out of memory)

- Is there a performance issue related to the internal structure of this hash table?

  Yes. If we keep adding elements to this hash table, we'll eventually have very long chains (because we don't change the number of buckets)

Resizing

- The typical approach is to keep the number of buckets (M) below (N / L), where N – number of key-value pairs, L – load factor (usually L = 0.75)
- The capacity is the number of buckets in the hash table
- The load factor is a measure of how full the hash table is allowed to get before its capacity is increased
- Example:
  - Capacity is 16, and we have 10 elements. Therefore, load factor is 62.5%
  - Once we have 13 elements, the load factor becomes 81.25%, higher than the desired 75%, and we should increase the capacity
- The usual approach is to double the number of buckets when the number of entries in the hash table exceeds the product of the load factor and the current capacity

Resizing

- Resizing is a complex operation: besides replacing the existing array with the new one that has 2x the size, we also need to **rehash** all of the keys
- Rehashing:
  - We need to recompute modular hash (`find_index(key)`) because the divisor (`bucket_count`) value has changed
  - Key's hash (before computing modulo) remains the same, and can therefore be cached for efficiency
  - Complexity of rehashing operation is O(n) because we need to iterate over all key-value pairs, recompute the indexes, and insert these elements
  - With resizing, amortized insertion complexity is still O(1) (under uniform hashing assumption) because we double the number of buckets, so rehashing doesn't happen every time we insert

Resizing implementation

```
def resize(self, new_capacity):
   new_arr = np.empty(new_capacity, dtype=object)
   for bucket in self.arr:
       node = bucket
       while node is not None:
           key = node.key
           index = key.__hash__() % new_capacity

SeparateChainingHashST.insert_at_index(new_arr, key,
node.value, index)
           node = node.next
   self.arr = new_arr
   self.capacity = new_capacity

@staticmethod
def insert_at_index(arr, key, value, index):
   node = arr[index]
   while node is not None:
       node = node.next
   arr[index] = Node(key, value, next=arr[index])
```

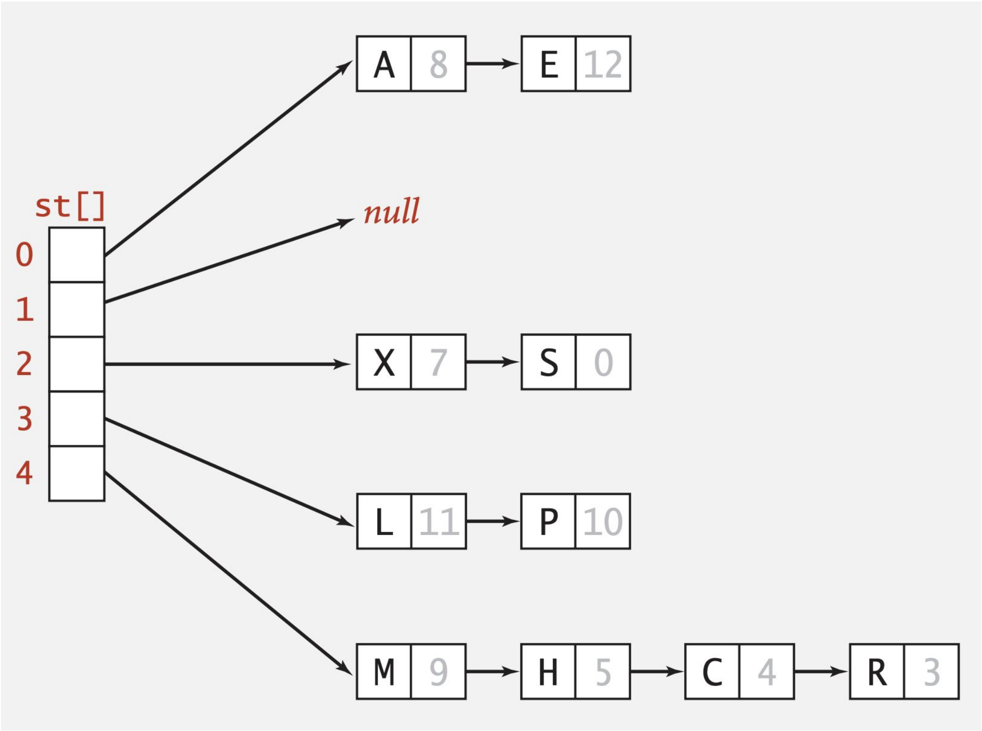
```
def __setitem__(self, key, value):
   current_load_factor =
(self.__len__() + 1) / self.capacity
   if current_load_factor >
self.load_factor:
       self.resize(self.capacity * 2)

   index = self.find_index(key)

   node = self.arr[index]
   while node is not None:
       if key == node.key:
           node.value = value
           return
       node = node.next

   self.arr[index] = Node(key, value,
next=self.arr[index])
   self.size += 1
```

[Full code]

Open addressing

- When a new key collides, find next empty slot, and put it there
- **Hash.** Map key to integer i between 0 and M-1
- **Insert.** Put at table index i if free; if not try i+1, i+2, etc.
- **Search.** Search table index i; if occupied but no match, try i+1, i+2, etc.
- **Note.** Array size M must be greater than number of key-value pairs N
- Searching through alternate locations in the array is called **probing**
- Described method is called **linear probing** (interval between probes is fixed — in our case set to 1)
- Another variation is **quadratic probing**: i+1, i+4, i+9, i+16, …, i+k^2

Separate chaining vs. linear probing



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

1. Open addressing (linear probing) hash table implementation

```
class LinearProbingHashST:
   def __init__(self, initial_capacity=16):
       self.keys = np.empty(initial_capacity, dtype=object)
       self.values = np.empty(initial_capacity, dtype=object)
       self.capacity = initial_capacity

   def find_index(self, key):
       return key.__hash__() % self.capacity
```

2. Open addressing (linear probing) hash table implementation

```python
def __setitem__(self, key, value):
    i = self.find_index(key)

    while self.keys[i] is not None:
        if self.keys[i] == key:
            break
        i = (i + 1) % self.capacity

    self.keys[i] = key
    self.values[i] = value
```

3. Open addressing (linear probing) hash table implementation

```python
def __getitem__(self, key):
    i = self.find_index(key)

    while self.keys[i] is not None:
        if self.keys[i] == key:
            return self.values[i]
        i = (i + 1) % self.capacity

    raise KeyError(key)
```

Resizing in a linear-probing hash table

- Goal:
    - Average length of list $N / M \leq 1/2$
    - Double size of array M when $N / M \geq 1/2$
    - Halve size of array M when $N / M \leq 1/8$
    - Need to rehash all keys when resizing

Deletion in a linear-probing hash table

- Deletion from an open-address hash table is difficult. When we delete a key from slot **i**, we cannot simply mark that slot as empty by storing **None** in it. Doing so might make it impossible to retrieve any key **k** during whose insertion we had probed slot **i** and found it occupied
- Assume **hash(x) = hash(y) = hash(z) = i**. And assume **x** was inserted first, then **y** and then **z**
- In open addressing: **table[i] = x**, **table[i+1] = y**, **table[i+2] = z**
- Now, assume you want to delete **x**, and set it back to **None**
- When later you will search for **z**, you will find that **hash(z) = i** and **table[i] = None**, and you will return a wrong answer: **z** is not in the table
- To overcome this, you need to set **table[i]** with a special marker indicating to the search function to keep looking at index **i+1**, because there might be element there which hash is also **i**

Complexity. Summary

ST implementation	Average case		Worst case		Key requirement
	Search	Insert	Search	Insert	
Linked list or array	`N / 2`	`N`	`N`	`N`	`__eq__`
Ordered array	`log N`	`N / 2`	`log N`	`N`	`__lt__`
Separate chaining hash table	`constant (*)`	`constant (*)`	`N`	`N`	`__eq__ + __hash__`
Linear probing hash table	`constant (*)`	`constant (*)`	`N`	`N`	`__eq__ + __hash__`

(*) under uniform hashing assumption

Separate chaining vs. linear probing

Clustering:

- Cluster. A contiguous block of items.
- Observation. New keys likely to hash into middle of big clusters

Separate chaining:

- Performance degrades gracefully
- Clustering less sensitive to poorly-designed hash function

Linear probing:

- Less wasted space
- Better processor cache performance

Set data structure

- Set is an abstract data type that can store unique values, without any particular order
- It is a computer implementation of the mathematical concept of a finite set
- Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set

Set in Python

```python
s = set()
s.add('element1')
s.add('element2')
s.add('element3')

test_value = 'element2'
if test_value in s:
   print(f'{test_value} is present in the set')

test_value = 'element5'
if test_value not in s:
   print(f'{test_value} is NOT present in the set')
```

Hash-table based set implementation

```python
class HTBackedSet:
    def __init__(self, hash_table):
        self.hash_table = hash_table
        self.present = object()  # dummy object

    def add(self, value):
        self.hash_table[value] = self.present

    def __contains__(self, value):
        try:
            _ = self.hash_table[value]
            return True
        except KeyError:
            return False
```

Hash-table based set implementation. Usage

```python
s = HTBackedSet(LinearProbingHashST())
s.add('element1')
s.add('element2')
s.add('element3')

test_value = 'element2'
if test_value in s:
   print(f'{test_value} is present in the set')

test_value = 'element5'
if test_value not in s:
   print(f'{test_value} is NOT present in the set')
```

Algorithmic complexity attack

- Denial-of-service attack is possible
- Malicious adversary learns your hash function (e.g., by reading your code / library API) and causes a big pile-up in single slot that grinds performance to a halt

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!