COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 12: Mar 19

Lecturer: Dr. Andrew Hines *Scribes: Patrick Malatesta, Conor Lonergan*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*
Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

12.1 Outline

Nowadays, thanks to internet and modern computing, a vast amount of information has been made accessible. In order to efficiently process all these information it's fundamental the ability to efficiently search through it. Searching algorithms have been and still are fundamental for the development of the computational infrastructure that we enjoy today. This session introduces **Hash Tables** and, in order to help understanding this data structure, we'll first touch on the concept of *Symbol Tables*.

12.1.1 Symbol Tables

A **symbol table** is an abstract mechanism where we save information (a *value*) that we can search and retrieve using a *key*. The implementation of an efficient symbol table is a challenge that depends upon the applications of this data structure. The primary purpose of a symbol table is to associate a *value* with a *key*. The client can insert key-value pairs into the symbol table with the expectation of later being able to search for the value associated with a given key, from among all of the key-value pairs that have been put into the table. Symbol tables are also referred to as *dictionaries* or *indices*. The table below gives an example of typical symbol table application.

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID

Figure 12.1: symbol-table applications

In general, an **unordered symbol table** data structure, supports the following operation:

- `put(key, value)` $\rightarrow$ put key-value pair into the table
- `get(key)` $\rightarrow$ value paired with key (null if key is absent)
- `delete(key)` $\rightarrow$ remove key from table and value paired with key
- `contains(key)` $\rightarrow$ is there a value paired with key?
- `isEmpty()` $\rightarrow$ is the table empty?

- `size()` $\rightarrow$ number of key-value pairs in the table
- `keys()` v all the keys in the table

Note that the implementation of an **ordered symbol table** data structure requires that we keep the symbols ordered. Therefore, we need to keep information about their rank and a number of other operations are required.

12.2 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations *insert*, *search*, and *delete*. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.

A **hash table** data structure is one effective way of implementing a symbol table. Its main feature is to save items in a key-indexed table where the *index* is a function of the *key*. The figure below shows an example of a hash table that stores the *value* "it" at the *index* 3.

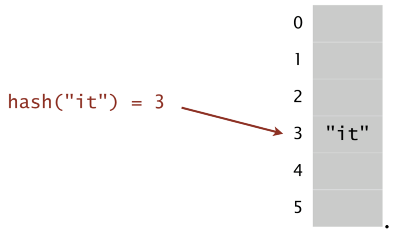


Figure 12.2: hash table

Theoretically, searching for an element in a hash table can take as long as searching for an element in a linked list: $\mathcal{O}(n)$ time in the *worst case*. However, in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $\mathcal{O}(1)$.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $\mathcal{O}(1)$ time. We can take advantage of direct addressing when we can afford to allocate an array that has one position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table. Specifically, we can reduce the storage requirement to $\mathcal{O}(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $\mathcal{O}(1)$ time. The main advantage is that this bound is for the *average-case time*, whereas for direct addressing it holds for the *worst-case time*.

12.3 Hash Functions

A **hash function** defines the method for computing the array *index* from a *key*.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot *h(k)*. We use a **hash function** h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a **hash table** T [0 .. m-1]:

$h : U \to \{0, 1, ..., m - 1\}$,

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k **hashes** to slot *h(k)*; we also say that *h(k)* is the **hash value** of key k. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m.

The figure below illustrates this idea and introduces the concept of **collision**, which will be discussed in the next section.

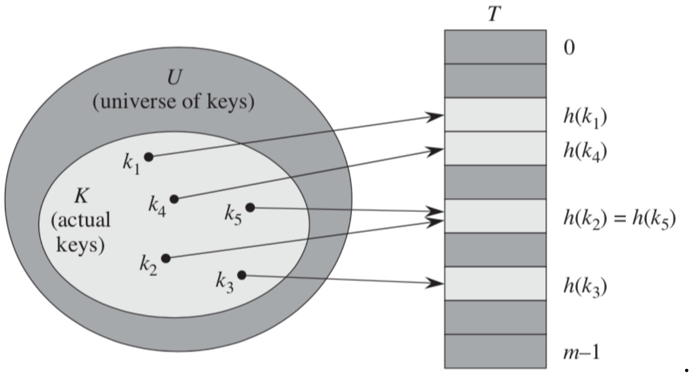


Figure 12.3: Using a hash function h to map keys to hash-table slots. The keys *k2* and *k5* map to the same slot and collide

When implementing a new hash function for a given data type, there are three main requirements to take into consideration:

- *consistent*. Equal keys must produce the same hash value
- *efficient to compute*
- *uniformly distribute the keys*

A bad hash function is a classic example of a *performance bug*: everything will work properly, but much more slowly than expected.

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

In practice, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the data type and the distribution of keys may be useful in this design process.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data. For example, the **division method** computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.

Most hash functions assume that the keys are represented by natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers In the **division method** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$h(k) = k \bmod m$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast. When using the division method, we usually avoid certain values of m, in particular m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.

Another method that can be used for creating hash functions is the **multiplication method** and it operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 0$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. An advantage of the multiplication method is that the value of m is not critical.

12.4 Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash function is perfect, collisions will never occur. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well- designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

12.4.1 Open Addressing

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table.

12.4.1.1 Linear Probing

In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1

Pseudocode - Linear Probing

```
Let hash(x) be the slot index computed using hash function and S be the table size
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
..................................................
..................................................
```

Python - Linear Probing

```
In [75]: class HashTable(object):

             def __init__(self):
                 self.max_length = 8
                 self.max_load_factor = 0.75
                 self.length = 0
                 self.table = [None] * self.max_length

             def __len__(self):
                 return self.length

             def __setitem__(self, key, value):
                 self.length += 1
                 hashed_key = self._hash(key)
                 while self.table[hashed_key] is not None:
                     if self.table[hashed_key][0] == key:
                         self.length -= 1
                         break
                     hashed_key = self._increment_key(hashed_key)
                 tuple = (key, value)
                 self.table[hashed_key] = tuple
                 if self.length / float(self.max_length) >= self.max_load_factor:
                     self._resize()

             def __getitem__(self, key):
                 index = self._find_item(key)
                 return self.table[index][1]

             def __delitem__(self, key):
                 index = self._find_item(key)
```

```
                self.table[index] = None

        def _hash(self, key):
            # TODO more robust
            return hash(key) % self.max_length

        def _increment_key(self, key):
            return (key + 1) % self.max_length

        def _find_item(self, key):
            hashed_key = self._hash(key)
            if self.table[hashed_key] is None:
                raise KeyError
            if self.table[hashed_key][0] != key:
                original_key = hashed_key
                while self.table[hashed_key][0] != key:
                    hashed_key = self._increment_key(hashed_key)
                    if self.table[hashed_key] is None:
                        raise KeyError
                    if hashed_key == original_key:
                        raise KeyError
            return hashed_key

        def _resize(self):
            self.max_length *= 2
            self.length = 0
            old_table = self.table
            self.table = [None] * self.max_length
            for tuple in old_table:
                if tuple is not None:
                    self[tuple[0]] = tuple[1]
```

12.4.1.2 Rehashing / Plus 3 Rehash

One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs and is known as rehashing.

One common approach to rehashing is done with a ”plus 3” probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

Pseudocode - Plus 3 Probing

```
rehash(pos) = (pos + 3) % sizeoftable
or
rehash(pos) = (pos + skip) % sizeoftable
```

let `hash(x)` be the slot index computed using hash function
If slot `hash(x) % S` is full, then we try `(hash(x) + skip) % S`
If slot `(hash(x) + 3) % S` is full, then we try `(hash(x) + 2 * 3) % S`
If slot `(hash(x) + 2 * 3) % S` is full, then we try `(hash(x) + 3 * 3) % S`

It is important to note that the size of the "skip" must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size be a prime number.

Python - Plus 3 Probing

```
In [76]: class HashTable:
             def __init__(self):
                 self.size = 11
                 self.slots = [None] * self.size
                 self.data = [None] * self.size
                 self.skip = 3 #Assign skip value to 3 for "plus 3" probe

             def put(self,key,data):
                 hashvalue = self.hashfunction(key,len(self.slots))

                 if self.slots[hashvalue] == None:
                     self.slots[hashvalue] = key
                     self.data[hashvalue] = data
                 else:
                     if self.slots[hashvalue] == key:
                         self.data[hashvalue] = data  #replace
                     else:
                         nextslot = self.rehash(hashvalue,len(self.slots))
                         while self.slots[nextslot] != None and self.slots[nextslot] != key:
                             nextslot = self.rehash(nextslot,len(self.slots), self.skip) #This

                         if self.slots[nextslot] == None:
                             self.slots[nextslot]=key
                             self.data[nextslot]=data
                         else:
                             self.data[nextslot] = data #replace

             def hashfunction(self,key,size):
                  return key%size

             #Rehash function that takes current position, size of array and skip value parame
             def rehash(self,oldhash,size, skip):
                 return (oldhash + skip) % size
```

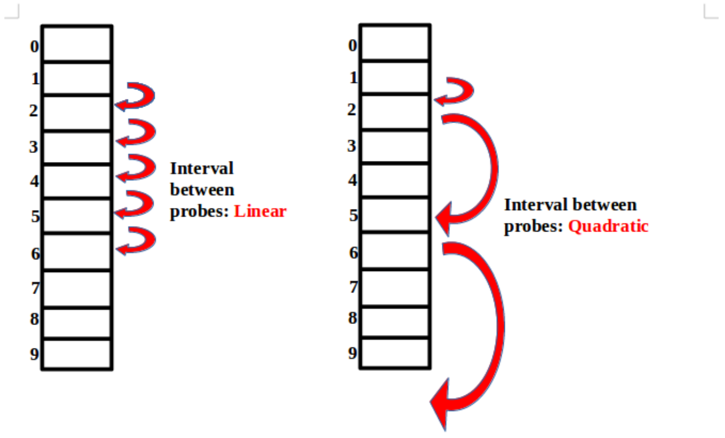

0
1
2
3
4
5
6
7
8
9
Interval between probes: Linear
Interval between probes: Quadratic

Figure 12.4: Linear Probing Vs. Quadratic Probing

12.4.1.3 Quadratic Probing

A variation of the linear probing idea is called quadratic probing. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on. This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

Pseudocode - Quadratic Probing

```
Let hash(x) be the slot index computed using hash function
If slot hash(x) % S is full, then we try (hash(x) + 1 * 1) % S
If (hash(x) + 1 * 1) % S is also full, then we try (hash(x) + 2 * 2) % S
If (hash(x) + 2 * 2) % S is also full, then we try (hash(x) + 3 * 3) % S
..................................................
..................................................
```

Python - Quadratic Probing

```
In [77]: class HashTable:
             def __init__(self):
                 self.size = 11
                 self.slots = [None] * self.size
                 self.data = [None] * self.size
                 self.collisionCount = 0
                 # Keeps track of the number of collisions for Quadratic calculation

             def put(self,key,data):
                 hashvalue = self.hashfunction(key,len(self.slots))
```

```
                if self.slots[hashvalue] == None:
                    self.slots[hashvalue] = key
                    self.data[hashvalue] = data
                else:
                    if self.slots[hashvalue] == key:
                        self.data[hashvalue] = data  #replace
                    else:
                        nextslot = self.rehash(hashvalue,len(self.slots))
                        while self.slots[nextslot] != None and self.slots[nextslot] != key:
                            # Incrememnt the number of collisions by 1
                            self.collisionCount += 1
                            nextslot = self.rehash(nextslot,len(self.slots),
                            self.collisionCount) #This is where we rehash

                        # Reset the number of collisions to 0 as slot has been found
                        self.collisionCount = 0
                        if self.slots[nextslot] == None:
                            self.slots[nextslot]=key
                            self.data[nextslot]=data
                        else:
                            self.data[nextslot] = data #replace

            def hashfunction(self,key,size):
                 return key%size

            #Rehash function that takes current position, size of array and the number
            # of collision that have occurred so far
            def rehash(self,oldhash,size, collisions):
                return (oldhash + collisions**2) % size
```

12.4.1.4 Double Hashing

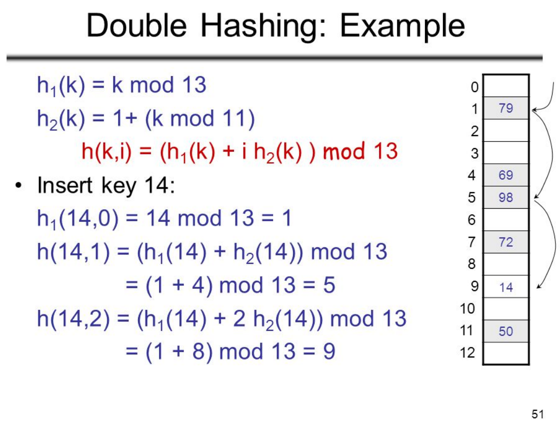
Double hashing uses the idea of applying a second hash function to key when a collision occurs.

First hash function is typically hash1(key) = key
A popular second hash function is : hash2(key) = PRIME – (key
A good second Hash function is:

- It must never evaluate to zero
- Must make sure that all cells can be probed

Pseudocode - Double Hashing

Let `hash(x)` be the slot index computed using hash function
If slot `hash(x) % S` is full, then we try `(hash(x) + 1 * hash2(x))` % S
If `(hash(x) + 1 * hash2(x)) % S` is also full, then we try `(hash(x) + 2 * hash2(x)) % S`
If `(hash(x) + 2 * hash2(x)) % S` is also full, then we try `(hash(x) + 3 * hash2(x)) % S`


Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$
$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$
$= (1 + 8) \bmod 13 = 9$

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

51

Figure 12.5: Double Hashing Example

....
....

Python - Double Hashing

```
In [85]: class HashTable:
             def __init__(self):
                 self.size = 11
                 self.slots = [None] * self.size
                 self.data = [None] * self.size
                 0 # Keeps track of the number of collisions for Quadratic calculation
                 self.collisionCount =

             def put(self,key,data):
                 hashvalue = self.hashfunction(key,len(self.slots))

                 if self.slots[hashvalue] == None:
                     self.slots[hashvalue] = key
                     self.data[hashvalue] = data
                 else:
                     if self.slots[hashvalue] == key:
                         self.data[hashvalue] = data  #replace
                     else:
                         nextslot = self.rehash(hashvalue,len(self.slots))
                         while self.slots[nextslot] != None and self.slots[nextslot] != key:
                             self.collisionCount += 1 # Incrememnt the number of collisions by
                             nextslot = self.rehash(nextslot,len(self.slots),self.collisionCou

                         self.collisionCount = 0 # Reset the number of collisions to 0 as slot
                         if self.slots[nextslot] == None:
```

```
                        self.slots[nextslot]=key
                        self.data[nextslot]=data
                    else:
                        self.data[nextslot] = data #replace

        def hashfunction(self,key,size):
             return key%size

        #Rehash function that takes current position, size of array and the number of col
        def rehash(self,oldhash,size, collisions, key):
            return (oldhash + collisions * hashTwo(key)) % size

        # A second independant hash function. We don't need to worry about size of array 
        # As returned value will be constrained in rehash function
        def hashTwo(self, key):
            return 1 + (key % 11)
```

12.4.2 Performance of Open Addressing

m = Number of slots in the hash table n = Number of keys to be inserted in the hash table
Load factor α = n/m (< 1)
Expected time to search/insert/delete < 1 / (1 - α)
So Search, Insert and Delete take (1 / (1 - α)) time

12.4.3 Closed Addressing Techniques

12.4.3.1 Linked List Chaining

A straightforward and general approach to collision resolution is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. This method is known as separate chaining because items that collide are chained together in separate linked lists. The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process: hash to find the list that could contain the key, then sequentially search through that list for the key.

Since we have M lists and N keys, the average length of the lists is always N/M, no matter how the keys are distributed among the lists. For example, suppose that all the items fall onto the first list—the average length of the lists is (N + 0 + 0 + 0 +. . . + 0)/M = N/M. However the keys are distributed on the lists, the sum of the list lengths is N and the average is N/M. Separate chaining is useful in practice because each list is extremely likely to have about N/M key-value pairs.
Table size. In a separate-chaining implementation, our goal is to choose the table size M to be

sufficiently small that we do not waste a huge area of contiguous memory with empty chains but sufficiently large that we do not waste time searching through long chains. One of the virtues of separate chaining is that this decision is not critical: if more keys arrive than expected, then searches will take a little longer than if we had chosen a bigger table size ahead of time; if fewer keys are in the table, then we have extra-fast search with some wasted space. When space is not a critical resource, M can be chosen sufficiently large that search time is constant; when space is a

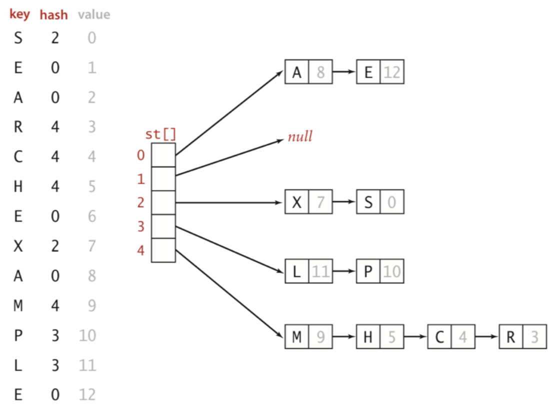


Figure 12.6: Chaining Using Linked Lists

critical resource, we still can get a factor of M improvement in performance by choosing M to be as large as we can afford.
Resizing. Another option is to use array resizing to keep the lists short. The load factor, N/M,

which is the ratio of the number of items in a hash table, M, and the capacity of the table, N, has a big impact on the performance of a hash table. In order to keep the load factor below the specified constant, we need to increase the size of our bucket array, A, and change our hash function to match this new size. Moreover, we must then insert all the existing hash-table elements into the new bucket array using the

Pseudocode - Chaining

Let `hash(x)` be the slot index computed using hash function, `L` a Linked-list at slot `hash(x)`, `e` an element, `tail` a pointer to the last node

```
newest ← e
newest.next ←None
L.tail.next ← newest
L.tail ← newest
```

Python - Chaining

```
In [86]: class HashTable:

             ratio_expand = .8
             ratio_shrink = .2
             min_size = 11


             def __init__(self, size=None):
                 self._size = size or self.min_size
                 self._buckets = [None] * self._size
```

```
        self._list = None
        self._count = 0


    def _entry(self, key):
        # get hash and index
        idx = hash(key) % self._size
        # find entry by key
        p, q = self._buckets[idx], None
        while p and p.key != key:
            p, q = p.next, p
        # index, entry, previous entry
        return idx, p, q


    def _ensure_capacity(self):
        fill = self._count / self._size

        # expand or shrink?
        if fill > self.ratio_expand:
            self._size = self._size * 2 + 1
        elif fill < self.ratio_shrink and \
                self._size > self.min_size:
            self._size = (self._size - 1) // 2
        else:
            return
        # reallocate buckets
        self._buckets = [None] * self._size
        # store entries into new buckets
        p = self._list
        while p:
            idx = hash(p.key) % self._size
            p.next = self._buckets[idx]
            self._buckets[idx] = p
            p = p.entry_next


    def __len__(self):
        return self._count


    def __contains__(self, key):
        _, p, _ = self._entry(key)
        return bool(p)


    def __getitem__(self, key):
        _, p, _ = self._entry(key)
        return p and p.value


    def __setitem__(self, key, value):
        idx, p, _ = self._entry(key)
        # set entry if key was found
        if p:
            p.value = value
            return
        # create new entry
        p = SimpleNamespace(
            key=key,
            value=value,
            next=self._buckets[idx],
            entry_next=self._list,
            entry_prev=None
        )
        # store to bucket
        self._buckets[idx] = p
        # store to list
        if self._list:
            self._list.entry_prev = p
        self._list = p
        # expand
        self._count += 1
        self._ensure_capacity()


    def __delitem__(self, key):
        idx, p, q = self._entry(key)
        # key not found
        if not p:
            return
        # remove from bucket
        if q:
            q.next = p.next
        else:
            self._buckets[idx] = p.next
        # remove from list
        if p.entry_next:
            p.entry_next.entry_prev = p.entry_prev
        if p.entry_prev:
            p.entry_prev.entry_next = p.entry_next
        else:
            self._list = p.entry_next
        # shrink
        self._count -= 1
        self._ensure_capacity()
    def __iter__(self):
        p = self._list
        while p:
            yield p.key
            p = p.entry_next


    def slots(self):
        return ''.join(p and 'x' or '-' for p in self._buckets)
```

12.4.4 Performance of Closed Addressing (Chaining)

m = Number of slots in hash table n = Number of keys to be inserted in hash table
Load factor $\alpha = n/m$
Expected time to search = $O(1 + \alpha)$
Expected time to insert/delete = $O(1 + \alpha)$
Time complexity of search insert and delete is $O(1)$ if α is $O(1)$

12.4.5 Closed Addressing Vs. Open Addressing

Closed Addressing	Open Addressing
Chaining is Simpler to implement	Open Addressing requires more computation
In chaining, Hash table never fills up, we can always add more elements to chain	In open addressing, table may become full
Chaining is Less sensitive to the hash function or load factors	Open addressing requires extra care for to avoid clustering and load facto
Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted	Open addressing is used when the frequency and number of keys is known

Closed Addressing	Open Addressing
Cache performance of chaining is not good as keys are stored using linked list	Open addressing provides better cache performance as everything is stored in the same table
Wastage of Space (Some Parts of hash table in chaining are never used)	In Open addressing, a slot can be used even if an input doesn't map to it.
Chaining uses extra space for links	No links in Open addressing

12.4.6 Example: Implementing the Map Abstract Data Type

One of the most useful Python collections is the dictionary. Recall that a dictionary is an associative data type where you can store key–data pairs. The key is used to look up the associated data value. We often refer to this idea as a map.

The map abstract data type is defined as follows. The structure is an unordered collection of associations between a key and a data value. The keys in a map are all unique so that there is a one-to-one relationship between a key and a value. The operations are given below.

- `Map()` Create a new, empty map. It returns an empty map collection.
- `put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- `get(key)` Given a key, return the value stored in the map or `None` otherwise.
- `del` Delete the key-value pair from the map using a statement of the form `del map[key]`.
- len() Return the number of key-value pairs stored in the map.
- `in` Return `True` for a statement of the form `key in map`, if the given key - is in the map, `False` otherwise.

One of the great benefits of a dictionary is the fact that given a key, we can look up the associated data value very quickly. In order to provide this fast look up capability, we need an implementation that supports an efficient search. We could use a list with sequential or binary search but it would be even better to use a hash table as described above since looking up an item in a hash table can approach $O(1)$ performance.

In the code cell below we use two lists to create a `HashTable` class that implements the Map abstract data type. One list, called `slots`, will hold the key items and a parallel list, called `data`, will hold the data values. When we look up a key, the corresponding position in the data list will hold the associated data value. We will treat the key list as a hash table using the ideas presented earlier. Note that the initial size for the hash table has been chosen to be 11. Although this is arbitrary, it is important that the size be a prime number so that the collision resolution algorithm can be as efficient as possible.

```
In [51]: class HashTable:
             def __init__(self):
                 self.size = 11
                 self.slots = [None] * self.size
                 self.data = [None] * self.size
```

hashfunction implements the simple remainder method. The collision resolution technique is linear probing with a “plus 1” rehash function. The 'put' function below assumes that there will eventually be an empty slot unless the key is already present in the `self.slots`. It computes the original hash value and if that slot is not empty, iterates the `rehash` function until an empty slot occurs. If a nonempty slot already contains the key, the old data value is replaced with the new data value. Dealing with the situation where there are no empty slots left is an exercise.

```
In [52]: def put(self,key,data):
           hashvalue = self.hashfunction(key,len(self.slots))

           if self.slots[hashvalue] == None:
             self.slots[hashvalue] = key
             self.data[hashvalue] = data
           else:
             if self.slots[hashvalue] == key:
               self.data[hashvalue] = data  #replace
             else:
               nextslot = self.rehash(hashvalue,len(self.slots))
               while self.slots[nextslot] != None and \
                               self.slots[nextslot] != key:
                 nextslot = self.rehash(nextslot,len(self.slots))

               if self.slots[nextslot] == None:
                 self.slots[nextslot]=key
                 self.data[nextslot]=data
               else:
                 self.data[nextslot] = data #replace

         def hashfunction(self,key,size):
              return key%size

         def rehash(self,oldhash,size):
             return (oldhash+1)%size
```

Likewise, the `get` function below begins by computing the initial hash value. If the value is not in the initial slot, `rehash` is used to locate the next possible position. Notice that line 15 guarantees that the search will terminate by checking to make sure that we have not returned to the initial slot. If that happens, we have exhausted all possible slots and the item must not be present.

The final methods of the `HashTable` class provide additional dictionary functionality. We overload the `__getitem__` and `__setitem__` methods to allow access using `[]`. This means that once a `HashTable` has been created, the familiar index operator will be available. We leave the remaining methods as exercises.

```
In [53]: def get(self,key):
           startslot = self.hashfunction(key,len(self.slots))
```

```
            data = None
            stop = False
            found = False
            position = startslot
            while self.slots[position] != None and  \
                                 not found and not stop:
               if self.slots[position] == key:
                 found = True
                 data = self.data[position]
               else:
                 position=self.rehash(position,len(self.slots))
                 if position == startslot:
                     stop = True
            return data

        def __getitem__(self,key):
            return self.get(key)

        def __setitem__(self,key,data):
            self.put(key,data)
```

The following session shows the full implementation of HashTable class in action. First we will create a hash table and store some items with integer keys and string data values.

```
In [62]: class HashTable:
             def __init__(self):
                 self.size = 11
                 self.slots = [None] * self.size
                 self.data = [None] * self.size

             def put(self,key,data):
               hashvalue = self.hashfunction(key,len(self.slots))

               if self.slots[hashvalue] == None:
                 self.slots[hashvalue] = key
                 self.data[hashvalue] = data
               else:
                 if self.slots[hashvalue] == key:
                   self.data[hashvalue] = data  #replace
                 else:
                   nextslot = self.rehash(hashvalue,len(self.slots))
                   while self.slots[nextslot] != None and \
                                   self.slots[nextslot] != key:
                     nextslot = self.rehash(nextslot,len(self.slots))

                   if self.slots[nextslot] == None:
                     self.slots[nextslot]=key
```

```
                self.data[nextslot]=data
              else:
                self.data[nextslot] = data #replace

        def hashfunction(self,key,size):
             return key%size

        def rehash(self,oldhash,size):
            return (oldhash+1)%size

        def get(self,key):
          startslot = self.hashfunction(key,len(self.slots))

          data = None
          stop = False
          found = False
          position = startslot
          while self.slots[position] != None and  \
                                 not found and not stop:
             if self.slots[position] == key:
               found = True
               data = self.data[position]
             else:
               position=self.rehash(position,len(self.slots))
               if position == startslot:
                   stop = True
          return data

        def __getitem__(self,key):
            return self.get(key)

        def __setitem__(self,key,data):
            self.put(key,data)
```

```
In [63]: H=HashTable()
         H[54]="cat"
         H[26]="dog"
         H[93]="lion"
         H[17]="tiger"
         H[77]="bird"
         H[31]="cow"
         H[44]="goat"
         H[55]="pig"
         H[20]="chicken"

In [64]: H.slots

Out[64]: [77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
```

```
In [65]: H.data

Out[65]: ['bird',
          'goat',
          'pig',
          'chicken',
          'dog',
          'lion',
          'tiger',
          None,
          None,
          'cow',
          'cat']
```

Next we will access and modify some items in the hash table. Note that the value for the key 20 is being replaced.

```
In [66]: H[20]

Out[66]: 'chicken'

In [67]: H[17]

Out[67]: 'tiger'

In [68]: H[20]='duck'

In [69]: H[20]

Out[69]: 'duck'

In [70]: H.data

Out[70]: ['bird',
          'goat',
          'pig',
          'duck',
          'dog',
          'lion',
          'tiger',
          None,
          None,
          'cow',
          'cat']
```

References

[1] Cormen H. T., Leiserson E. C., Rivest L. R., Stein C., (2009), *Introduction to Algorithms*, Third Edition, The MIT Press Cambridge, Massachusetts, pp. 254-278.

[2] Goodrich M. T., Tamassia R., (2015), *Algorithm Design and Applications*, Wiley, pp. 187-205.

[3] Sedgewick R., Wayne K., (2011), *Algorithms*, Fourth Edition, Princeton University, Addison-Wesley, pp. 61-477.

[4] http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/intro.html. Last accessed 11 Mar 2019

[5] https://www.geeksforgeeks.org/hashing-set-3-open-addressing/. Last accessed 12 Mar 2019

[6] https://www.youtube.com/watch?v=KyUTuwz_b7Q. Last accessed 12 Mar 2019

[7] https://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html. Last accessed 12 Mar 2019

[8] https://gist.github.com/scwood/b9a106b1450bfa67b1d5860b91a64c96. Last accessed 12 Mar 2019

[9] https://medium.com/100-days-of-algorithms/day-71-hash-table-chaining-c417b7cf4de6. Last accessed 12 Mar 2019