The focus of this chapter includes:

- Understanding the **concept of Array** data structure and its **basic operations**;
- Introducing the built-in data structures include **list, string, and tuple** which are arrays but each come with different features and popular module as a complement;
- Understanding the concept of **the linked list**, either single linked list or the doubly linked list, and the Python implementation of each data structure.

-1.1 Array

An array is container that holds a **fixed size** of sequence of items stored at **contiguous memory locations** and each item is identified by *array index* or *key*. The Array representation is shown in Fig. 1. Since using contiguous memory locations, once we know the physical position of the first element, an offset related to data types can be used to access any other items in the array with $O(1)$. Because of these items are physically stored contiguous one after the other, it makes array the most efficient data structure to store and access the items. Specifically, array is designed ans used for fast random access of data.

Static Array VS Dynamic Array There are two types of array: static array and dynamic array. They are different in the matter of fixing size or not. In the static array, once we declared the size of the array, we are not allowed to insert or delete any item at any position of the array. This is due to the inefficiency of doing so, which can lead to $O(n)$ time and space complexity. For dynamic array, the fixed size restriction is removed but with high price to allow it to be dynamic. However, the flip side of the coin is that if the memory size of the array is beyond the memory size of your computer, it could be impossible to fit the entire array in, and then we would retrieve to other data structures that would not require the physical contiguity, such as linked list, trees, heap, and graph.

Commonly, arrays are used to implement mathematical vectors and matrices. Also, arrays are the basic units implementing other data structures, such as hashtables, heaps, queues, stacks. We will see from other remaining contents of this part that how array-based Python data structures are used to implement the other data structures. On the LeetCode, these two data structures are involved into 25% of LeetCode Problems. *To note that it is not necessarily for array data structure to have the same data type in the real implementation of responding programming language as Python.*

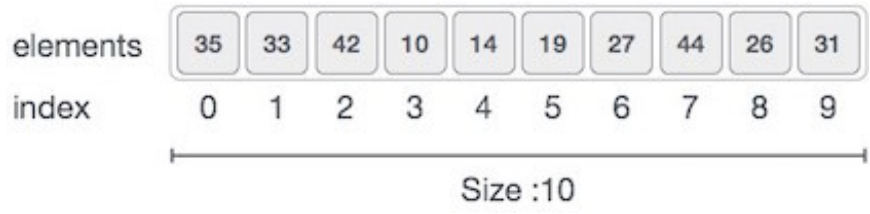

Figure 1: Array Representation

Operations Array supports the following operations:

- Access: it takes $O(1)$ time to access one item in the array given the index;
- Insertion and Deletion (for dynamic array only): it consumes Average $O(n)$ time to insert or delete an item from the array due to the fact that we need to shift the items after the modified position;
- Search and Iteration: $O(n)$ time for array to iterate all the elements in the array. Similarly to search an item by value through iteration takes $O(n)$ time too.

In Python, there is no strictly defined built-in data types that are static array. But it does have three there are built-in Array-like data structures: List, Tuple, String, and Range. These data structures are different in the sense of mutability, static or dynamic. More details of these data structures in Python will be given in the next section. **module array** which is of same data types just as the definition of Array here is also implemented the same as in C++ with its array data structure. However, **array** is not as widely used as of these three Items in list are actually not consecutive in memory because it is mutable object. Memory speaking, list is not as efficient as module Array and Strings.

-1.1.1 Python Built-in Sequence: List, Tuple, String, and Range

In Python, *sequences* are defined as ordered sets of objects indexed by non-negative integers. *Lists* and *tuples* are sequences of arbitrary objects. While *strings* are sequences of characters. Unlike List, which is mutable[1] and dynamic[2] , strings and tuples are immutable. All these sequence type data structures share the most common methods and operations shown in Table 1 and 2. To note that in Python, the indexing starts from 0.

[1] can modify item after its creation

[2]

Table 1: Common Methods for Sequence Data Type in Python

Function Method	Description
len(s)	Get the size of sequence s
min(s, [,default=obj, key=func])	The minimum value in s (alphabetically for strings)
max(s, [,default=obj, key=func])	The maximum value in s (alphabetically for strings)
sum(s, [,start=0)	The sum of elements in s(return $TypeError$ if s is not numeric)
all(s)	Return $True$ if all elements in s are True (Similar to *and*)
any(s)	Return $True$ if any element in s is True (similar to *or*)

Table 2: Common Methods for Sequence Data Type in Python

Operation	Description
s + r	Concatenates two sequences of the same type
s * n	Make n copies of s, where n is an integer
$v_1, v_2, ..., v_n = s$	Unpack n variables from s
s[i]	Indexing-returns ith element of s
s[i:j:stride]	Slicing-returns elements between i and j with optinal stride
x in s	Return $True$ if element x is in s
x not in s	Return $True$ if element x is not in s

Negative indexing and slicing For indexing and slicing, we can pass by negative integer as index and stride. The negative means backward, such as -1 refers to the last item, -2 to the second last item and so on. For the slicing

We list other characters and operations for each of these sequence data types:

List and Range

A list is similar to an array, but has a variable size which makes it more like a dynamic array, and does not necessarily need to be made up of a single continuous chunk of memory. While this does make lists more useful in general then arrays, you do incur a slight performance penalty due to the overhead needed to have those nicer characteristics. Also, list can be composed of items of any data types: including boolean, int, float, string, tuple, dictionary or even list itself. A list can be easily embedded into a list, which makes multi-dimensional data structure. A list is an ordered collection of items just like the definition of array.

A list is a *dynamic mutable* type and this means you can add and delete elements from the list at any time. **list** is optimized for fast fixed-length

operations as the definition of Array. It is dynamic, thus it supports size growth, however, it will incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

The range() type returns an immutable sequence of numbers between the given start integer to the stop integer. range() constructor has two forms of definition: range(stop) and range([start], stop[, step]): it is used to generate integers in range [start/0, stop), and the step can be positive/negative integer which determines the increment between each integer in the sequence.

Use List as Static Array Therefore, list is better used with fixed size, and no operation that incur items shifting such as pop(0) and insert(0, v), or operation that incurs size growth such as append(). Because, it pre-alloc a fixed size and once the size larger than this size, a new larger array is made and everything inside the old array is copied over, then the old array is marked for deletion. For example, we new a fixed size list, and we can do slicing, looping, indexing.

```
lst1 = [3]*5      # new a list size 5 with 3 as initialization
lst2 = [4 for i in range(5)]
for idx, v in enumerate(lst1):
    lst1[idx] += 1
```

SEARCH We use method list.index() to obtain the index of the searched element.

```
# SEARCHING
print(lst.index(4)) #find 4, and return the index
# output
# 3
```

If we print(lst.index(5)) will raise ValueError: 5 is not in list. Use the following code instead.

```
if 5 in lst:
    print(lst.index(5))
```

Use List as Dynamic Array When the input size is reasonable, list can be used dynamically. Now, Table 3 shows us the common List Methods, and they will be used as list.methodName().

Now, let us look at some examplary code.

New a List: We have multiple ways to new either empty list or with initialized data. List comprehension is an elegant and concise way to create new list from an existing list in Python.

Table 3: Common Methods of List

Method	Description
append()	Add an element to the end of the list
extend(l)	Add all elements of a list to the another list
insert(index, val)	Insert an item at the defined index s
pop(index)	Removes and returns an element at the given index
remove(val)	Removes an item from the list
clear()	Removes all items from the list
index(val)	Returns the index of the first matched item
count(val)	Returns the count of number of items passed as an argument
sort()	Sort items in a list in ascending order
reverse()	Reverse the order of items in the list (same as list[::-1])
copy()	Returns a shallow copy of the list (same as list[::])

```
# new an empty list
lst = []
lst2 = [2, 2, 2, 2] # new a list with initialization
lst3 = [3]*5      # new a list size 5 with 3 as initialization
print(lst, lst2, lst3)
# output
# [] [2, 2, 2, 2] [3, 3, 3, 3, 3]
```

INSERT and APPEND: To insert an item into the list, it actually involves one position shift to all the items that are after this position. This makes it takes $O(n)$ time complexity.

```
# INSERTION
lst.insert(0, 1) # insert an element at index 0, and since it is
    empty lst.insert(1, 1) has the same effect
print(lst)

lst2.insert(2, 3)
print(lst2)
# output
# [1]
# [2, 2, 3, 2, 2]
# APPEND
for i in range(2, 5):
    lst.append(i)
print(lst)
# output
# [1, 2, 3, 4]
```

The time complexity of different built-in method for list can be found at `wiki.python.org/moin/TimeComplexity`.

If you have a lot of numeric arrays you want to work with then it is worth

using the **NumPy** library which is an extensive array handling library often used by software engineers to do linear algebra related tasks.

String

String are similar to static array and it follows the restriction that it only stores one type of data: characters represented using ASCII or Unicode [3]. String is more compact compared with storing the characters in *list*. In all, string is immutable and static, meaning we can not modify its elements or extend its size once its created.

String is one of the most fundalmental built-in data types, this makes managing its common methods shown in Table 4 and 5 necessary. Use boolean methods to check whether characters are lower case, upper case, or title case, can help us to sort our data appropriately, as well as provide us with the opportunity to standardize data we collect by checking and then modifying strings as needed.

Following this, we give some examples showing how to use these functions.

join(), split(), and replace() The str.join(), str.split(), and str.replace() methods are a few additional ways to manipulate strings in Python.

The str.join() method will concatenate two strings, but in a way that passes one string through another. For example, we can use the str.join() method to add whitespace to that string, which we can do like so:

```
balloon = "Sammy has a balloon."
print(" ".join(balloon))
#Ouput
S a m m y   h a s   a   b a l l o o n .
```

The str.join() method is also useful to combine a list of strings into a new single string.

```
print(",".join(["a", "b", "c"]))
#Ouput
abc
```

Just as we can join strings together, we can also split strings up using the str.split() method. This method separates the string by whitespace if no other parameter is given.

```
print(balloon.split())
#Ouput
['Sammy', 'has', 'a', 'balloon.']
```

We can also use str.split() to remove certain parts of an original string. For example, let's remove the letter 'a' from the string:

[3] In Python 3, all strings are represented in Unicode. In Python 2 are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. It is no longer necessary now.

Table 4: Common Methods of String

Method	Description
count(substr, [start, end])	Counts the occurrences of a substring with optional start and end position
find(substr, [start, end])	Returns the index of the first occurrence of a substring or returns -1 if the substring is not found
join(t)	Joins the strings in sequence t with current string between each item
lower()/upper()	Converts the string to all lowercase or uppercase
replace(old, new)	Replaces old substring with new substring
strip([characters])	Removes withspace or optional characters
split([characters], [maxsplit])	Splits a string separated by whitespace or an optional separator. Returns a list
expandtabs([tabsize])	Replaces tabs with spaces.

Table 5: Common Boolean Methods of String

Boolean Method	Description
isalnum()	String consists of only alphanumeric characters (no symbols)
isalpha()	String consists of only alphabetic characters (no symbols)
islower()	String's alphabetic characters are all lower case
isnumeric()	String consists of only numeric characters
isspace()	String consists of only whitespace characters
istitle()	String is in title case
isupper()	String's alphabetic characters are all upper case

```
print(balloon.split("a"))
#Ouput
['S', 'mmy h', 's ', ' b', 'lloon.']
```

Now the letter a has been removed and the strings have been separated where each instance of the letter a had been, with whitespace retained.

The str.replace() method can take an original string and return an updated string with some replacement.

Let's say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we will change the substring "has" from the original string balloon to "had" in a new string:

```
print(balloon.replace("has","had"))
#Ouput
Sammy had a balloon.
```

We can use the replace method to delete a substring:

```
ballon.replace("has", '')
```

Using the string methods str.join(), str.split(), and str.replace() will provide you with greater control to manipulate strings in Python.

Related Useful Functions Function ord() would get the int value (ASCII) of the char. And in case you want to convert back after playing with the number, function chr() does the trick.

```
print(ord('A'))# Given a string of length one, return an integer
    representing the Unicode  code point of the character when
   the argument is a unicode object,
print(chr(65))
```

Tuple

A tuple is a sequence of immutable Python objects, which is to say the values of tuples can not be changed once its assigned. Also, as an immutable objects, they are hashable, and thus be used as keys to dictionaries. Like string and lists, tuple indices start at 0, and they can be indexed, sliced, concatenated and so on. Tuples only offer two additional methods shown in Table 6.

Table 6: Methods of Tuple

Method	Description
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

Since, tuples are quite similiar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

New and Initialize Tuple Tuple can be created in two different syntax: (1) putting different comma-separated values in a pair of parentheses; (2) creating a tuple using built-in function *tuple*(), if the argument to tuple() is a sequence then this creates a tuple of elements of that sequences. See the Python snippet:

```
''' new a tuple'''

# creat with ()
tup = () # creates an empty tuple
tup1 = ('crack', 'leetcode', 2018, 2019)
tup2 = ('crack', ) # when only has one element, put comma behind, so that it wont be translated as string

# creat with tuple()
tup3 = tuple() # new an empty tuple
tup4 = tuple("leetcode") # the sequence is passed as a tuple of elements
tup5 = tuple(['crack', 'leetcode', 2018, 2019]) # same as tuple1
print('tup1: ', tup1, '\ntup2: ', tup2, '\ntup3: ', tup3 , '\ntup4: ', tup4, '\ntup5: ', tup5)
```

The out put is:

```
tup1:  ('crack', 'leetcode', 2018, 2019)
tup2:  crack
tup3:  ()
tup4:  ('l', 'e', 'e', 't', 'c', 'o', 'd', 'e')
tup5:  ('crack', 'leetcode', 2018, 2019)
```

Changing a Tuple A tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
'''change a tuple'''
tup = ('a', 'b', [1, 2, 3])
#tup[0] = 'c' #TypeError: 'tuple' object does not support item assignment
tup[-1][0] = 4
print(tup)
tup = ('c', 'd')
print(tup)
```

The output is:

```
('a', 'b', [4, 2, 3])
('c', 'd')
```

Deleting a Tuple As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple. But deleting a tuple entirely is possible using the keyword del.

```
del tup
print(tup)
```

After del, when try to use tup again it returns NameError.

```
NameError: name 'tup' is not defined
```

-1.1.2 Bonus

Circular Array The corresponding problems include:

1. 503. Next Greater Element II

-1.1.3 Exercises

1. 985. Sum of Even Numbers After Queries (easy)
2. 937. Reorder Log Files

   You have an array of logs. Each log is a space delimited string of words.

   For each log, the first word in each log is an alphanumeric identifier. Then, either:

   Each word after the identifier will consist only of lowercase letters, or; Each word after the identifier will consist only of digits.

   We will call these two varieties of logs letter-logs and digit-logs. It is guaranteed that each log has at least one word after its identifier.

   Reorder the logs so that all of the letter-logs come before any digit-log. The letter-logs are ordered lexicographically ignoring identifier, with the identifier used in case of ties. The digit-logs should be put in their original order.

   Return the final order of the logs.

```
Example 1:

Input: ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key
    dog","a8 act zoo"]
Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9
    2 3 1","zo4 4 7"]



Note:

    0 <= logs.length <= 100
    3 <= logs[i].length <= 100
    logs[i] is guaranteed to have an identifier, and a word
     after the identifier.
```

```python
def reorderLogFiles(self, logs):
    letters = []
    digits = []
    for idx, log in enumerate(logs):
        splited = log.split(' ')
        id = splited[0]
        type = splited[1]

        if type.isnumeric():
            digits.append(log)
        else:
            letters.append((' '.join(splited[1:]), id))
    letters.sort() #default sorting by the first element
    and then the second in the tuple

    return [id + ' ' + other for other, id in letters] +
    digits
```

```python
def reorderLogFiles(logs):
    digit = []
    letters = []
    info = {}
    for log in logs:
        if '0' <= log[-1] <= '9':
            digit.append(log)
        else:
            letters.append(log)
            index = log.index(' ')
            info[log] = log[index+1:]

    letters.sort(key= lambda x: info[x])
    return letters + digit
```

-1.2 Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers. The benefits of linked lists include: first, they do not require sequential spaces; second, they can start small and grow arbitrarily as we add more items to the data structures. Linked list is designed to offer flexible change of size and constant time complexity for inserting a new element which can not be obtained from array.

Even in Python, lists are actually dynamic arrays, However, it still requires growing by copy and paste periodically. However, linked list suffers from its own demerits:

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.

2. Extra memory space for a pointer is required with each element of the list.

The composing unit of linked list is called **nodes**. There are two types of linked lists based on its ability to iterate items in different directions: Singly Linked List wherein a node has only one pointer to link the successive node, and Doubly Linked List wherein a node has one extra pointer to link back to its predecessor.

We will detail these two sub data structures of linked list in the following sections.

-1.2.1 Singly Linked List

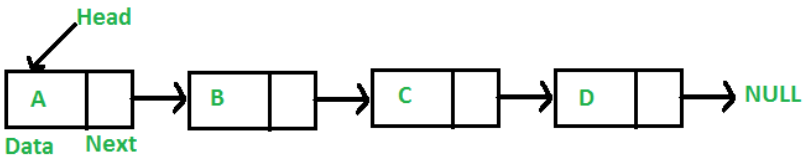

Figure 2: Linked List Structure

Fig. 2 shows the structure of a singly linked list. As we can see, a singly linked is a linear data structure with only one pointer between two successive nodes, and can only be traversed in a single direction, that is, we can go from the first node to the last node, but can not do it in backforward direction.

Node To implement a singly linked list, we need to first implement a Node which has two members: **val** which is used to save contents and **next** which is a pointer to the successive node. The Node class is given as:

```
class SinglyListNode(object):
    def __init__(self, val = None):
        self.val = val
        self.next = None
```

Implementation Here, we define a class as **SinglyLinkedList** which implements the operations needed for a singly linked list data structure and hide the concept of Node to users. The **SinglyLinkedList** usually needs a head that points to the first node in the list, and we need to make sure the last element will be linked to a **None** node. The necessary operations of a linked list include: insertion/append, delete, search, clear. Some linked list can only allow insert node at the tail which is Append, some others might allow insertion at any location. To get the length of the linked list easily in O(1), we need a variable to track the size

```
class SinglyLinkeList:
    def __init__(self):
        # with only head
        self.head = None
        self.size = 0
    def len(self):
        return self.size
```

Append: it is a common scenerio that we build up a linked list from a list, which requires append operations. Because in our implementation, the head pointer always points to the first node, it requires us to traverse all the nodes to implement the Append, which gives $O(n)$ as the time complexity for append.

```
#...
    def append(self, val):
        node = SinglyLinkedNode(val)
        if self.head:
            # traverse to the end
            current = self.head
            while current:
                current = current.next
            current.next = node
        else:
            self.head = node
        self.size += 1
```

Deletion: sometimes we need to delete a node by value in the linked list, this requires us to rewire the pointers between the precessor and successor of the deleting node. This requires us to find iterate the list to locate the node to be deleted and track the previous node for rewiring. We can possibly have two cases:

1. if the node is head, directly repoint the head to the next node
2. otherwise, we need to connect the previous node to current node's next node, and the head pointer remains untouched.

```
#...
    def delete(self, val):
        current = self.head
        prev = self.head
        while current:
            if current.val == val:
                # if the node is head
                if current == self.head:
                    self.head = current.next
                # rewire
                else:
                    prev.next = current.next
                self.size -= 1
            prev = current
            current = current.next
```

Sometimes, we will be asked to delete a List node, this deleting process does not need the head and the traversal to find the value. We simply need to change the value of this node to the value of the next node, and connect this node to the next node's node

```
def deleteByNode(self, node):
    node.val = node.next.val
    node.next = node.next.next
```

Search and iteration: in order to traverse the list and not to expose the users to the node class by usig node.val to get the contents of the node, we need to implement a method **iter()** that returns a generator gives out the contents of the list.

```
# ...
    def iter(self):
        current = self.head
        while current:
          val = current.val
          current = current.next
          yield val
```

Now, the linked list iteration looks just like a normal like iteration. Search operation can now built upon the iteration and the process is the same as linear search:

```
# ...
    def search(self, val):
        for value in self.iter():
            if value == val:
                return True
        return False
```

Clear: in some cases, we need to clear all the nodes of the list, this is a quite simple process. All we need to do to set the head to None

```
    def clear(self):
        self.head = None
        self.size = 0
```

-1.2.2 Doubly Linked List

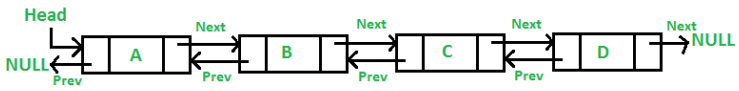

Figure 3: Doubly Linked List

On the basis of Singly linked list, doubly linked list (dll) contains an extra pointer in the node structure which is typically called **prev** (short

for previous) and points back to its predecessor in the list. Because of the prev pointer, a DLL can traverse in both forward and backward direction. Also, compared with SLL, some operations such as deletion is more efficient because we do not need to track the previous node in the traversal process.

```
# Node of a doubly linked list
class Node:
    def __init__(self, val, prev = None, next = None):
        self.val = val
        self.prev = prev # reference to previous node in DLL
        self.next = next # reference to next node in DLL
```

We define our class as DoublyLinkedList. Same as class SinglyLinkedList, have one pointer called **head** and another variable to track the size. Therefore we skip the definition of the class and its init function.

Append: The only difference is to link the nodes when adding and relinking when deleting.

```
# linking
# replace line 3, line 9
node = Node(val)
current.next, node.prev = node, current
```

Deletion: compared with sll, we do not need to track the previous node.

```
#...
    def delete(self, val):
        current = self.head
        #prev = self.head
        while current:
            if current.val == val:
                # if the node is head
                if current == self.head:
                    current.prev = None #set the prev
                    self.head = current.next
                # rewire
                else:
                    #prev.next = current.next
                    current.prev.next, current.next.prev =
    current.next, current.prev

                self.size -= 1
            #prev = current
            current = current.next
```

All the remaining operations such as Search, Iteration, and Clear are exactly the same as in sll.

-1.2.3 Bonus

Tail Pointer For both singly and doubly linked list, if we add another **tail** pointer to its class, which points at the last node in the list, can simplify some operations of the linked list from $O(n)$ to $O(1)$.

Circular Linked List A circular linked list is a variation of linked list in which the first node connects to last node. To make a circular linked list from a normal linked list: in singly linked list, we simply set the last node's next pointer to the first node; in doubly linked list, other than setting the last node's next pointer, we set the prev pointer of the first node to the last node making the circular in both directions.

Compared with a normal linked list, circular linked list saves time for that to go to the first node from the last (both sll and dll) or go to the last node from the first node (in dll) by doing it in a single step through the extra connection. This has the same usage of adding the tail pointer mentioned before. While, the other side of the flip coin is when we are iterating the nodes, we need to compare current visiting node with the head node and when we make sure we end the iteration after visiting the tail node when the next points to the head pointer.

And for circular linked list, to iterate all items, we need to set up the end condition for the while loop. If we let the current node start from the head node (the head is not None), the loop will terminate if the next node is head node again.

```
def iterateCircularList(head):
    if not head:
        return
    cur = head
    while cur.next != head:
        cur = cur.next
    return
```

Dummy Node Dummy node is a node that does not hold any value – an empty Node use None as value, but is in the list to provide an extra node at the front and/or read of the list. It is used as a way to reduce/remove special cases in coding so that we can simply the coding complexity.

Divide and Conquer + Recursion With the coding simplicity of recursion and the ability to iterate in a backward (or bottom-up) direction, we can use divide and conquer to solve problems from the smallest problem. The practical experience say that this method can be very helpful in solving linked list problems.

Let's look at a very simple example on LeetCode which demonstrates both the usage of dummy node and recursion.

-1.1 **83. Remove Duplicates from Sorted List (easy)**. Given a sorted linked list, delete all duplicates such that each element appear only once.

```
Example 1:

Input: 1->1->2
Output: 1->2

Example 2:

Input: 1->1->2->3->3
Output: 1->2->3
```

Analysis: This is a linear complexity problem, the most straightforward way is to traverse the list and compare the current node's value with the next's to check its equivalency: (1) if YES: delete the next code and not move the current node; (2) if NO: we can move to the next node. We can also solve it in recursion way. We recursivly call the node.next and the end case is when we meet the last node, we return that node directly. Then in the bottom-up process, we compare the current node and the returning node (functioning as a head for the subproblem).

Solution 1: Iteration. The code is given:

```
def deleteDuplicates(self, head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if not head:
        return None

   def iterative(head):
        current = head
        while current:
            # current pointer wont move unless the next has different value
            if current.next and current.val == current.next.val:
                # delete next
                current.next = current.next.next
            else:
                current = current.next
        return head

    return iterative(head)
```

We can see each time we need to check if current.next has value or not, this process can be avoid using a dummy node before the head.

```
# use of dummy node
```

```
def iterative(head):
    dummy = ListNode(None)
    dummy.next = head
    current = dummy
    while current.next:
        # current pointer wont move unless the next has
    different value
        if current.val == current.next.val:
            # delete next
            current.next = current.next.next
        else:
            current = current.next
    return head
```

Solution 2: Recursion.

```
def recursive(node):
        if node.next is None:
            return node

        next = recursive(node.next)
        if next.val == node.val:
            # delete next
            node.next = node.next.next
        return node
```

-1.2.4 Exercises

Basic operations:

1. 237. Delete Node in a Linked List (easy, delete only given current node)
2. 2. Add Two Numbers (medium)
3. 92. Reverse Linked List II (medium, reverse in one pass)
4. 83. Remove Duplicates from Sorted List (easy)
5. 82. Remove Duplicates from Sorted List II (medium)
6. Sort List
7. Reorder List

Fast-slow pointers:

1. 876. Middle of the Linked List (easy)
2. Two Pointers in Linked List
3. Merge K Sorted Lists

Recursive and linked list:

1. 369. Plus One Linked List (medium)

-1.3 Stack and Queue

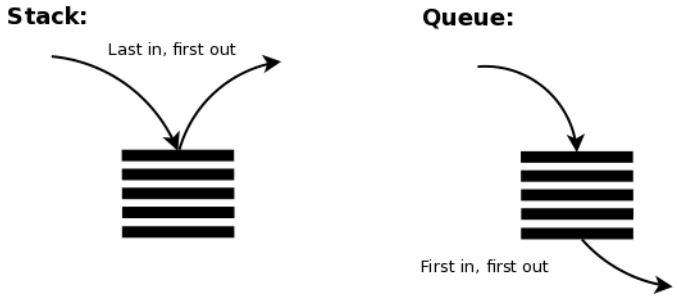

Figure 4: Stack VS Queue

Stacks and queue are dynamic arrays with restrictions on deleting elements. Stack data structure can be visualized a stack of plates, we would always put a plate on top of the pile, and get one from the top of it too. This is stated as **Last in, first out (LIFO)**. Queue data structures are like real-life queue in the cashier out line, it follows the rule 'first come, first served', which can be officiallized as **first in, first out (FIFO)**.

Therefore, given a dynamic array, we always add element by appending at the end, a stack and a queue can be implemented with prespecified deleting operation: for stack, we delete from the rear; for a queue, we delete from the front of the array.

Stack data structures fits well for tasks that require us to check the previous states from cloest level to furtherest level. Here are some examplary applications: (1) reverse an array, (2) implement DFS iteratively as we will see in Chapter **??**, (3) keep track of the return address during function calls, (4) recording the previous states for backtracking algorithms.

Queue data structures can be used: (1) implement BFS shown in Chapter **??**, (2) implement queue buffer.

In the remaining section, we will discuss the implement with the built-in data types or using built-in modules. After this, we will learn more advanced queue and stack: the priority queue and the monotone queue which can be used to solve medium to hard problems on LeetCode.

-1.3.1 Basic Implementation

For Queue and Stack data structures, the essential operations are two that adds and removes item. In Stack, they are usually called **PUSH** and **POP**. PUSH will add one item, and POP will remove one item and return its value. These two operations should only take $O(1)$ time. Sometimes, we need another operation called PEEK which just return the element that can be accessed in the queue or stack without removing it. While in Queue, they are named as **Enqueue** and **Dequeue**.

The simplest implementation is to use Python List by function *insert*() (insert an item at appointed position), *pop*() (removes the element at the given index, updates the list , and return the value. The default is to remove the last item), and *append*(). However, the list data structure can not meet the time complexity requirement as these operations can potentially take $O(n)$. We feel its necessary because the code is simple thus saves you from using the specific module or implementing a more complex one.

Stack The implementation for stack is simplily adding and deleting element from the end.

```
# stack
s = []
s.append(3)
s.append(4)
s.append(5)
s.pop()
```

Queue For queue, we can append at the last, and pop from the first index always. Or we can insert at the first index, and use pop the last element.

```
# queue
# 1: use append and pop
q = []
q.append(3)
q.append(4)
q.append(5)
q.pop(0)
```

Running the above code will give us the following output:

```
print('stack:', s, ' queue:', q)
stack: [3, 4]  queue: [4, 5]
```

The other way to implement it is to write class and implement them using concept of node which shares the same definition as the linked list node. Such implementation can satisfy the $O(1)$ time restriction. For both the stack and queue, we utilize the singly linked list data structure.

Stack and Singly Linked List with top pointer Because in stack, we only need to add or delete item from the rear, using one pointer pointing at the rear item, and the linked list's next is connected to the second toppest item, in a direction from the top to the bottom.

```
# stack with linked list
'''a<-b<-c<-top'''
class Stack:
    def __init__(self):
        self.top = None
        self.size = 0

```

```
    # push
    def push(self, val):
        node = Node(val)
        if self.top: # connect top and node
            node.next = self.top
        # reset the top pointer
        self.top = node
        self.size += 1

    def pop(self):
        if self.top:
            val = self.top.val
            if self.top.next:
                self.top = self.top.next # reset top
            else:
                self.top = None
            self.size -= 1
            return val

        else: # no element to pop
            return None
```

Queue and Singly Linked List with Two Pointers For queue, we need to access the item from each side, therefore we use two pointers pointing at the head and the tail of the singly linked list. And the linking direction is from the head to the tail.

```
# queue with linked list
'''head->a->b->tail'''
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    # push
    def enqueue(self, val):
        node = Node(val)
        if self.head and self.tail: # connect top and node
            self.tail.next = node
            self.tail = node
        else:
            self.head = self.tail = node

        self.size += 1

    def dequeue(self):
        if self.head:
            val = self.head.val
            if self.head.next:
                self.head = self.head.next # reset top
            else:
                self.head = None
```

```
27                  self.tail = None
28              self.size -= 1
29              return val
30
31          else: # no element to pop
32              return None
```

Also, Python provide two built-in modules: **Deque** and **Queue** for such purpose. We will detail them in the next section.

-1.3.2 Deque: Double-Ended Queue

Deque object is a supplementary container data type from Python **collections** module. It is a generalization of stacks and queues, and the name is short for "double-ended queue". Deque is optimized for adding/popping items from both ends of the container in $O(1)$. Thus it is preferred over **list** in some cases. To new a deque object, we use **deque([iterable[, maxlen]])**. This returns us a new deque object initialized left-ro-right with data from iterable. If maxlen is not specified or is set to None, deque may grow to an arbitray length. Before implementing it, we learn the functions for **deque class** first in Table 7.

Table 7: Common Methods of Deque

Method	Description
append(x)	Add x to the right side of the deque.
appendleft(x)	Add x to the left side of the deque.
pop()	Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.
popleft()	Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.
maxlen	Deque objects also provide one read-only attribute:Maximum size of a deque or None if unbounded.
count(x)	Count the number of deque elements equal to x.
extend(iterable)	Extend the right side of the deque by appending elements from the iterable argument.
extendleft(iterable)	Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.
remove(value)	emove the first occurrence of value. If not found, raises a ValueError.
reverse()	Reverse the elements of the deque in-place and then return None.
rotate(n=1)	Rotate the deque n steps to the right. If n is negative, rotate to the left.

In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in

operator, and subscript references such as d[-1].

Now, we use deque to implement a basic stack and queue,the main methods we need are: append(), appendleft(), pop(), popleft().

```
'''Use deque from collections'''
from collections import deque
q = deque([3, 4])
q.append(5)
q.popleft()

s = deque([3, 4])
s.append(5)
s.pop()
```

Printing out the q and s:

```
print('stack:', s, ' queue:', q)
stack: deque([3, 4])  queue: deque([4, 5])
```

Deque and Ring Buffer Ring Buffer or Circular Queue is defined as a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. This normally requires us to predefine the maximum size of the queue. To implement a ring buffer, we can use deque as a queue as demonstrated above, and when we initialize the object, set the maxLen. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end.

-1.3.3 Python built-in Module: Queue

The **queue module** provides thread-safe implementation of Stack and Queue like data structures. It encompasses three types of queue as shown in Table 8. *In python 3, we use lower case queue, but in Python 2.x it uses Queue, in our book, we learn Python 3.*

Table 8: Datatypes in Queue Module, maxsize is an integer that sets the upperbound limit on the number of items that can be places in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

Class	Data Structure
class queue.Queue(maxsize=0)	Constructor for a FIFO queue.
class queue.LifoQueue(maxsize=0)	Constructor for a LIFO queue.
class queue.PriorityQueue(maxsize=0)	Constructor for a priority queue.

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below in Table 9.

Now, using Queue() and LifoQueue() to implement queue and stack respectively is straightforward:

Table 9: Methods for Queue's three classes, here we focus on single-thread background.

Class	Data Structure
Queue.put(item[, block[, timeout]])	Put item into the queue.
Queue.get([block[, timeout]])	Remove and return an item from the queue.
Queue.qsize()	Return the approximate size of the queue.
Queue.empty()	Return True if the queue is empty, False otherwise.
Queue.full()	Return True if the queue is full, False otherwise.

```
# python 3
import queue
# implementing queue
q = queue.Queue()
for i in range(3, 6):
    q.put(i)
```

```
import queue
# implementing stack
s = queue.LifoQueue()

for i in range(3, 6):
    s.put(i)
```

Now, using the following printing:

```
print('stack:', s, ' queue:', q)
stack: <queue.LifoQueue object at 0x000001A4062824A8>  queue: <
    queue.Queue object at 0x000001A4062822E8>
```

Instead we print with:

```
print('stack: ')
while not s.empty():
    print(s.get(), end=' ')
print('\nqueue: ')
while not q.empty():
    print(q.get(), end = ' ')
stack:
5 4 3
queue:
3 4 5
```

-1.3.4 Monotone Stack

A monotone Stack is a data structure the elements from the front to the end is strictly either increasing or decreasing. For example, there is an

line at the hair salo, and you would naturally start from the end of the line. However, if you are allowed to kick out any person that you can win at a fight, if every one follows the rule, then the line would start with the most powerful man and end up with the weakest one. This is an example of monotonic decreasing stack.

- Monotonically Increasing Stack: to push an element e, starts from the rear element, we pop out element $r >= e$ (violation);
- Monotonically Decreasing Stack: we pop out element $r <= e$ (violation). T

The process of the monotone decresing stack is shown in Fig. 5. *Sometimes,*

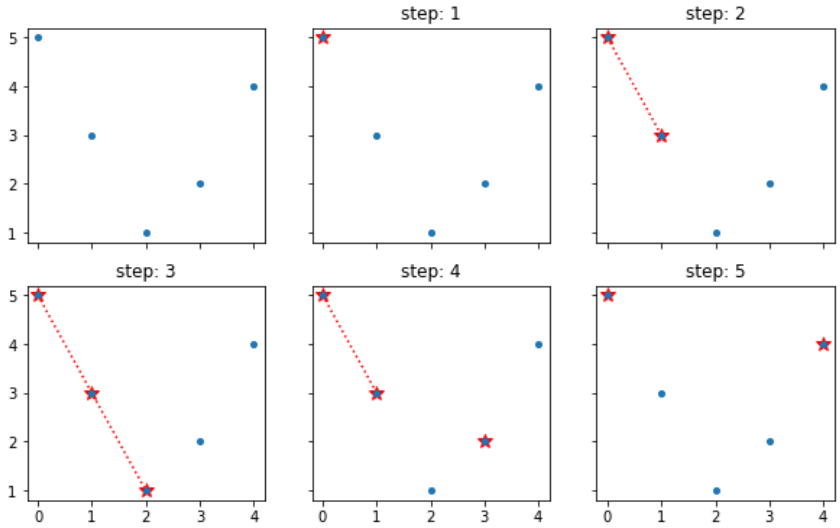

Figure 5: The process of decreasing monotone stack

we can relax the strict monotonic condition, and can allow the stack or queue have repeat value.

To get the feature of the monotonic queue, with $[5, 3, 1, 2, 4]$ as example, if it is increasing:

index	v	Increasing stack	Decreasing stack
1	5	[5]	[5]
2	3	[3] 3 kick out 5	[5, 3] #3–>5
3	1	[1] 1 kick out 3	[5, 3, 1] #1–>3
4	2	[1, 2] #2–>1	[5, 3, 2] 2 kick out 1
5	4	[1, 2, 4] #4–>2	[5,4] 4 kick out 2, 3

By observing the above process, what features we can get?

- Pushing in to get smaller/larger item to the left: When we push an element in, if there exists one element right in front of it, 1) for increasing stack, we find the **nearest smaller item to the left** of current

item, 2) for decreasing stack, we find the **nearest larger item** to the left instead. In this case, we get [-1, -1, -1, 1, 2], and [-1, 5, 3, 3, 5] respectively.

- Popping out to get smaller/larger item to the right: when we pop one element out, for the kicked out item, such as in step of 2, increasing stack, 3 forced 5 to be popped out, for 5, 3 is the first smaller item to the right. Therefore, if one item is popped out, for this item, the current item that is about to be push in is 1) for increasing stack, **the nearest smaller item to its right**, 2) for decreasing stack, **the nearest larger item to its right**. In this case, we get [3,1, -1, -1, -1], and [-1, 4, 2, 4, -1] respectively.

The conclusion is with monotone stack, we can search for smaller/larger items of current item either to its left/right.

Basic Implementation This monotonic queue is actually a data structure that needed to add/remove element from the end. In some application we might further need to remove element from the front. Thus Deque from collections fits well to implement this data structure. Now, we set up the example data:

```
A = [5, 3, 1, 2, 4]
import collections
```

Increasing Stack We can find first smaller item to left/right.

```
def increasingStack(A):
    stack = collections.deque()
    firstSmallerToLeft = [-1]*len(A)
    firstSmallerToRight = [-1]*len(A)
    for i,v in enumerate(A):
        while stack and A[stack[-1]] >= v: # right is from the
    popping out
            firstSmallerToRight[stack.pop()] = v  # A[stack[-1]]
     >= v
        if stack:  #left is from the pushing in, A[stack[-1]] <
    v
            firstSmallerToLeft[i] = A[stack[-1]]
        stack.append(i)
    return firstSmallerToLeft, firstSmallerToRight, stack
```

Now, run the above example with code:

```
firstSmallerToLeft, firstSmallerToRight, stack = increasingQueue
    (A)
for i in stack:
    print(A[i], end = ' ')
print('\n')
print(firstSmallerToLeft)
print(firstSmallerToRight)
```

The output is:

```
1 2 4

[-1, -1, -1, 1, 2]
[3, 1, -1, -1, -1]
```

Decreasing Stack We can find first larger item to left/right.

```
def decreasingStack(A):
    stack = collections.deque()
    firstLargerToLeft = [-1]*len(A)
    firstLargerToRight = [-1]*len(A)
    for i,v in enumerate(A):
        while stack and A[stack[-1]] <= v:
            firstLargerToRight[stack.pop()] = v

        if stack:
            firstLargerToLeft[i] = A[stack[-1]]
        stack.append(i)
    return firstLargerToLeft, firstLargerToRight, stack
```

Similarily, the output is:

```
5 4

[-1, 5, 3, 3, 5]
[-1, 4, 2, 4, -1]
```

For the above problem, If we do it with brute force, then use one for loop to point at the current element, and another embedding for loop to look for the first element that is larger than current, which gives us $O(n^2)$ time complexity. If we think about the BCR, and try to trade space for efficiency, and use monotonic queue instead, we gain $O(n)$ linear time and $O(n)$ space complexity.

Monotone stack is especially useful in the problem of subarray where we need to find smaller/larger item to left/right side of an item in the array. To better understand the features and applications of monotone stack, let us look at some examples. First, we recommend the audience to practice on these obvious applications shown in LeetCode Problem Section before moving to the examples:

There is one problem that is pretty interesting:

Sliding Window Maximum/Minimum Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window. (LeetCode Probelm: 239. Sliding Window Maximum (hard))

```
Example:
```

```
Input: nums = [1,3,-1,-3,5,3,6,7], and k = 3
Output: [3,3,5,5,6,7]
Explanation:

Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Analysis: In the process of moving the window, any item that is smaller than its predecessor will not affect the max result anymore, therefore, we can use decrese stack to remove any trough. If the window size is the same as of the array, then the maximum value is the first element in the stack (bottom). With the sliding window, we record the max each iteration when the window size is the same as k. At each iteration, if need to remove the out of window item from the stack. For example of [5, 3, 1, 2, 4] with k = 3, we get [5, 3, 4]. At step 3, we get 5, at step 4, we remove 5 friom the stack, and we get 3. At step 5, we remove 3 if it is in the stack, and we get 4. With the monotone stack, we decrease the time complexity from $O(kn)$ to $O(n)$.

```
1  import collections
2
3  def maxSlidingWindow(self, nums, k):
4      ds = collections.deque()
5      ans = []
6      for i in range(len(nums)):
7          while ds and nums[i] >= nums[ds[-1]]: indices.pop()
8          ds.append(i)
9          if i >= k - 1: ans.append(nums[ds[0]]) #append the
       current maximum
10         if i - k + 1 == ds[0]: ds.popleft() #if the first also
       the maximum number is out of window, pop it out
11     return ans
```

-1.2 **907. Sum of Subarray Minimums (medium).** Given an array of integers A, find the sum of min(B), where B ranges over every (contiguous) subarray of A. Since the answer may be large, return the answer modulo $10^9 + 7$. *Note: 1 <= A.length <= 30000, 1 <= A[i] <= 30000.*

```
Example 1:

Input: [3,1,2,4]
Output: 17
Explanation: Subarrays are [3], [1], [2], [4], [3,1],
    [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.  Sum is 17.
```

Analysis: For this problem, using naive solution to enumerate all possible subarries, we end up with n^2 subarray and the time complexity would be $O(n^2)$, and we will receive LTE. For this problem, we just need to sum over the minimum in each subarray. Try to consider the problem from another angel, what if we can figure out how many times each item is used as minimum value corresponding subarry? Then res = sum(A[i]*f(i)). If there is no duplicate in the array, then To get f(i), we need to find out:

- left[i], the length of strict bigger numbers on the left of A[i],
- right[i], the length of strict bigger numbers on the right of A[i].

For the given examples, if A[i] = 1, then the left item is 3, and the right item is 4, we add 1*(left_len*right_len) to the result. However, if there is duplicate such as [3, 1, 4, 1], for the first 1, we need [3,1], [1], [1,4], [1, 4,1] with subarries, and for the second 1, we need [4,1], [1] instead. Therefore, we set the right length to find the >= item. Now, the problem in converted to the first smaller item on the left side and the first smaller or equal item on the right side. From the feature we draw above, we need to use increasing stack, as we know, from the pushing in, we find the first smaller item, and from the popping out, for the popped out item, the current item is the first smaller item on the right side. The code is as:

```
def sumSubarrayMins(self, A):
    n, mod = len(A), 10**9 + 7
    left, s1 = [1] * n, []
    right = [n-i for i in range(n)]
    for i in range(n): # find first smaller to the left
    from pushing in
        while s1 and A[s1[-1]] > A[i]: # can be equal
            index = s1.pop()
            right[index] = i-index # kicked out
        if s1:
            left[i] = i-s1[-1]
        else:
            left[i] = i+1
        s1.append(i)
    return sum(a * l * r for a, l, r in zip(A, left, right)
    ) % mod
```

The above code, we can do a simple improvement, by adding 0 to each side of the array. Then eventually there will only have [0, 0] in the stack. All of the items originally in the array they will be popped out, each popping, we can sum up the result directely:

```
def sumSubarrayMins(self, A):
    res = 0
    s = []
```

```
    A = [0] + A + [0]
    for i, x in enumerate(A):
        while s and A[s[-1]] > x:
            j = s.pop()
            k = s[-1]
            res += A[j] * (i - j) * (j - k)
        s.append(i)
    return res % (10**9 + 7)
```

-1.3.5 Bonus

Circular Linked List and Circular Queue The circular queue is a linear data structure in which the operation are performed based on FIFO principle and the last position is connected back to the the first position to make a circle. It is also called "Ring Buffer". Circular Queue can be either implemented with a list or a circular linked list. If we use a list, we initialize our queue with a fixed size with None as value. To find the position of the enqueue(), we use $rear = (rear + 1)\%size$. Similarily, for dequeue(), we use $front = (front + 1)\%size$ to find the next front position.

-1.3.6 Exercises

Queue and Stack

1. 225. Implement Stack using Queues (easy)
2. 232. Implement Queue using Stacks (easy)
3. 933. Number of Recent Calls (easy)

Queue fits well for buffering problem.

1. 933. Number of Recent Calls (easy)
2. 622. Design Circular Queue (medium)

```
Write a class RecentCounter to count recent requests.

It has only one method: ping(int t), where t represents some
    time in milliseconds.

Return the number of pings that have been made from 3000
    milliseconds ago until now.

Any ping with time in [t - 3000, t] will count, including the
    current ping.

It is guaranteed that every call to ping uses a strictly larger
    value of t than before.

```

```
11
12
13 Example 1:
14
15 Input: inputs = ["RecentCounter","ping","ping","ping","ping"],
       inputs = [[],[1],[100],[3001],[3002]]
16 Output: [null,1,2,3,3]
```

Analysis: This is a typical buffer problem. If the size is larger than the buffer, then we squeeze out the easilest data. Thus, a queue can be used to save the t and each time, squeeze any time not in the range of [t-3000, t]:

```
1  class RecentCounter:
2
3      def __init__(self):
4          self.ans = collections.deque()
5
6      def ping(self, t):
7          """
8          :type t: int
9          :rtype: int
10         """
11         self.ans.append(t)
12         while self.ans[0] < t-3000:
13             self.ans.popleft()
14         return len(self.ans)
```

Monotone Queue

1. 84. Largest Rectangle in Histogram
2. 85. Maximal Rectangle
3. 122. Best Time to Buy and Sell Stock II
4. 654. Maximum Binary Tree

Obvious applications:

1. 496. Next Greater Element I
2. 503. Next Greater Element I
3. 121. Best Time to Buy and Sell Stock

1. 84. Largest Rectangle in Histogram
2. 85. Maximal Rectangle
3. 122. Best Time to Buy and Sell Stock II
4. 654. Maximum Binary Tree

5. 42 Trapping Rain Water
6. 739. Daily Temperatures
7. 321. Create Maximum Number

-1.4 Hash Table

A hash map (or hash table) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function $h(key)$ to compute an index into an array of buckets or slots, from which the desired value will be stored and found. A well-designed hashing should gives us constant average time to insert and search for an element in a hash map as $O(1)$. In this section, we will examine the hashing design and analysis mechanism.

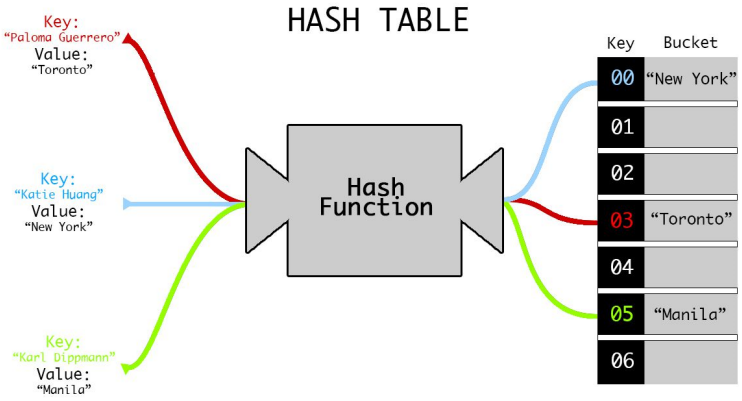

Figure 6: Example of Hashing Table

First, let us frame the hashing problem: given a universe U of keys (or items) with size n, with a hash table denoted by $T[0...m-1]$, in which each position, or slot, corresponds to a key in the hash table. Fig. 6 shows an example of hash table. When two keys have the same hash value produced by hash function, it is called **collision**. Because $\alpha > 1$, there must be at least two keys that have the same hash value; avoiding collisions altogether seems impossible. Therefore, in reality, a well-designed hashing mechanism should include: (1) a hash function which minimizes the number of collisions and (2) a efficient collision resolution if it occurs.

Applications If average lookup times in an algorithm of each item is n and each time, using linear search will take $O(n)$, this makes the total time complexity to $O(n^2)$. If we spend $O(n)$ to save them in the hash table

and each search will be $O(1)$, thus decrease the total time complexity to $O(n)$. For example, string process such as Rabin-Karp algorithm for pattern matching in a string in $O(n)$ time. Also, there are two data structures offered by all programming languages uses hashing table to implement: hash set and hash map, in Python, there are *set* and *dict* (dictionary) built-in types.

- Hash Map: hash map is a data structures that stores items as (key, value) pair. And "key" are hashed using hash table into an index to access the value.
- Hash Set: different to hash map, in a hash set, only keys are stored and it has no duplicate keys. Set usually represents the mathematical notion of a set, which is used to test membership, computing standard operations on such as intersection, union, difference, and symmetric difference.

-1.4.1 Hash Function Design

Hash function maps the universe U of keys into the slots of a hashtable $T[0..m-1]$. With hash function the element is stored in $f(k,m)$. $h : U- > \{0, 1, ..., m-1\}$. The hash function design includes two steps: interpreting keys as nature numbers and design hashing functions.

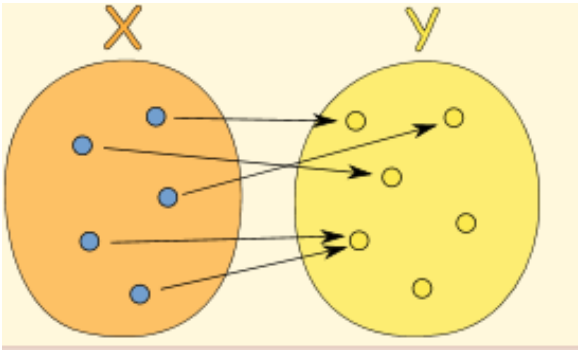

Figure 7: The Mapping Relation of Hash Function

Interpreting Keys For a given key, if the keys are not natural numbers, such as any string, or tuple, they need to be firstly interpreted as natural integers $N = \{0, 1, 2, ...\}$. This interpretation relation(function) needs to be one to one; given two distinct keys, they should never be interpreted as the same natural number. And we denote it as a interpret function $k = f(key)$, where key is a input key and k is the interpreted natural number. For string or character, one possible way is to express them in suitable radix notation. we might translate "pt" as the pair of decimal integers $(112, 116)$ with their ASCII character; then we express it as a radix128 integer, then the number

we get is $(112 \times 128) + 116 = 14452$. This is usually called **polynomial rolling hashing**, to generalize, $k = s[0] + s[1] * p + s[2] * p^2 + ... + s[n-1] * p^{n-1}$, where the string has length n, and p is a chosen prime number which is roughly equal to the number of characters in the input alphabet. For instance, if the input is composed of only lowercase letters of English alphabet, p=31 is a good choice. If the input may contain both uppercase and lowercase letters, then p=53 is a possible choice.

Hash Function Design As the definition of function states: a function relates **each element** of a set with **exactly one element** of another set (possibly the same set). The hash function is denoted as $index = f(k, m)$. One essential rule for hashing is if two keys are equal, then a hash function should produce the same key value ($f(s, m) = f(t, m)$, if $s = t$). And, we try our best to minimize the collision to make it unlikely for two distinct keys to have the same value. Therefore our expectation for average collision times for the same slot will be $\alpha = \frac{n}{m}$, which is called **loading factor** and is a critical statistics for design hashing and analyze its performance. The relation is denoted in Fig. 7. Besides, a good hash function satisfied the condition of simple uniform hashing: each key is equally likely to be mapped to any of the m slots. There are generally four methods:

1. **The Direct addressing method**, $f(k, m) = k$, and $m = n$. Direct addressing can be impractical when n is beyond the memory size of a computer. Also, it is just a waste of spaces when $m << n$.

2. **The division method**, $f(k, m) = k\%m$, where % is the module operation in Python, it is the reminder of k divided by m. A large prime number not too close to an exact power of 2 is often a good choice of m. The usage of prime number is to minimize collisions when the data exhibits some particular patterns. For example, in the following cases, when $m = 4$, and $m = 7$, keys = [10, 20, 30, 40, 50]

	m = 4	m = 7
10	10=4*2+2	10=7*1+3
20	20=4*5+0	20=7*2+6
30	30=4*7+2	30=7*4+2
40	40=4*10+0	40=7*5+5
50	50=4*12+2	50=7*7+1

   Because the keys share a common factor $c = 2$ with the bucket number 4, then when we apply the division, it became $(key/c)/(m/c)$; both the quotient(also a multiple of the bucket size) and the remainder(modulo or bucket number) can be written as multiple of the common factor. So, the range of the slot index will be decrease to m/c. The real loading factor increase to $c\alpha$. Using a prime number is a easy way to avoid this since a prime number has no factors other than 1 and itself.

3. **The multiplication method**, $f(k, m) = \lfloor m(kA\%1) \rfloor$. $A \in (0, 1)$ is a chosen constant and a suggestion to it is $A = (\sqrt{5} - 1)/2$. $kA\%1$ means the fractional part of kA and equals to $kA - \lfloor kA \rfloor$. It is also shorten as $\{kA\}$. E.g. for 45.2 the fractional part of it is .2.

4. **Universal hashing method**: because any fixed hash function is vulnerable to the worst-case behavior when all n keys are hashed to the same index, an effective way is to ch

-1.4.2 Collision Resolution

Collision is unavoidable given that $m < n$ and the data can be adversary.

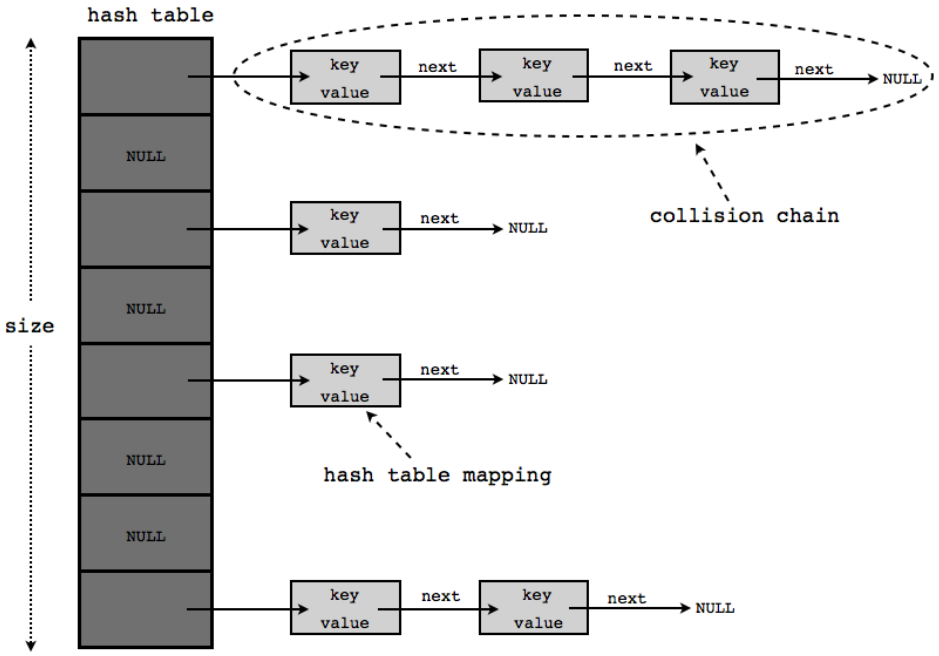

Figure 8: Hashtable chaining to resolve the collision

Resolving by Chaining An easy way to think of is by chaining the keys that have the same hashing value using a linked list (either singly or doubly). For example, when $f = k\%4$, and keys = [10,20,30,40,50]. For key as 10, 30, 50, they are mapped to the same slot 2. Therefore, we chain them up at index 2 using a single linked list shown in Fir. 8.

The average-case time for searching and insertion is $O(\alpha)$ under the assumption of simple uniform hashing. However, the worst-case for operation can be $O(n)$ when all keys are mapped to the same slot.

The advantage of chaining is that hash table never fills up, and we can always add more elements by chaining behind. It is mostly used when the number of keys and the frequency of operations.

However, because using linked list, (1) the cache performance is poor; and (2) the use of pointers takes extra space that could be used to save data.

Resolving by Open Addressing Open addressing takes a different approach to handle the collisions, where all items are stored in the hash table. Therefore, open addressing requires the size of the hash table to be larger or equal to the size of keys ($m \geq n$). In open addressing, it computes a probe sequence as of [h(k, 0), h(k, 1),...,h(k, m-1)] which is a permutation of [0, 1, 2, ..., m-1]. We successively probe each slot until an empty slot is found.

Insertion and searching is easy to implement and are quite similar. However, when we are deleting a key, we can not simply delete the key and value and mark it as empty. If we did, we might be unable to retrieve any key during whose insertion we had probed slot i because we stop probing whenever empty slot is found.

Let us see with an example: 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 NULL. When later you will search for z, you will find that hash(z) = i and table[i] = NULL, 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 its hash is also i

Perfect Hashing

-1.4.3 Implementation

In this section, we practice on the learned concepts and methods by implementing hash set and hash map.

Hash Set Design a HashSet without using any built-in hash table libraries. To be specific, your design should include these functions: (705. Design HashSet)

```
add(value): Insert a value into the HashSet.
contains(value) : Return whether the value exists in the HashSet
    or not.
remove(value): Remove a value in the HashSet. If the value does
   not exist in the HashSet, do nothing.
```

For example:

```
MyHashSet hashSet = new MyHashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.contains(1);    // returns true
hashSet.contains(3);    // returns false (not found)
hashSet.add(2);
hashSet.contains(2);    // returns true
hashSet.remove(2);
hashSet.contains(2);    // returns false (already removed)
```

Note: Note: (1) All values will be in the range of [0, 1000000]. (2) The number of operations will be in the range of [1, 10000].

```
class MyHashSet:

    def _h(self, k, i):
        return (k+i) % 10001

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.slots = [None]*10001
        self.size = 10001

    def add(self, key: 'int') -> 'None':
        i = 0
        while i < self.size:
            k = self._h(key, i)
            if self.slots[k] == key:
                return
            elif not self.slots[k] or self.slots[k] == -1:
                self.slots[k] = key
                return
            i += 1
        # double size
        self.slots = self.slots + [None]*self.size
        self.size *= 2
        return self.add(key)


    def remove(self, key: 'int') -> 'None':
        i = 0
        while i < self.size:
            k = self._h(key, i)
            if self.slots[k] == key:
                self.slots[k] = -1
                return
            elif self.slots[k] == None:
                return
            i += 1
        return

    def contains(self, key: 'int') -> 'bool':
```

```
        """
        Returns true if this set contains the specified element
        """
        i = 0
        while i < self.size:
            k = self._h(key, i)
            if self.slots[k] == key:
                return True
            elif self.slots[k] == None:
                return False
            i += 1
        return False
```

Hash Map Design a HashMap without using any built-in hash table libraries. To be specific, your design should include these functions: (706. Design HashMap (easy))

- put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.
- get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key. remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

Example:

```
hashMap = MyHashMap();
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);            // returns 1
hashMap.get(3);            // returns -1 (not found)
hashMap.put(2, 1);          // update the existing value
hashMap.get(2);            // returns 1
hashMap.remove(2);          // remove the mapping for 2
hashMap.get(2);            // returns -1 (not found)
```

```
class MyHashMap:
    def _h(self, k, i):
        return (k+i) % 10001 # [0, 10001]
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.size = 10002
        self.slots = [None] * self.size


    def put(self, key: 'int', value: 'int') -> 'None':
        """
        value will always be non-negative.
        """
```

```
16          i = 0
17          while i < self.size:
18              k = self._h(key, i)
19              if not self.slots[k] or self.slots[k][0] in [key,
    -1]:
20                  self.slots[k] = (key, value)
21                  return
22              i += 1
23          # double size and try again
24          self.slots = self.slots + [None]* self.size
25          self.size *= 2
26          return self.put(key, value)
27
28
29      def get(self, key: 'int') -> 'int':
30          """
31          Returns the value to which the specified key is mapped,
    or -1 if this map contains no mapping for the key
32          """
33          i = 0
34          while i < self.size:
35              k = self._h(key, i)
36              if not self.slots[k]:
37                  return -1
38              elif self.slots[k][0] == key:
39                  return self.slots[k][1]
40              else: # if its deleted keep probing
41                  i += 1
42          return -1
43
44
45      def remove(self, key: 'int') -> 'None':
46          """
47          Removes the mapping of the specified value key if this
    map contains a mapping for the key
48          """
49          i = 0
50          while i < self.size:
51              k = self._h(key, i)
52              if not self.slots[k]:
53                  return
54              elif self.slots[k][0] == key:
55                  self.slots[k] = (-1, None)
56                  return
57              else: # if its deleted keep probing
58                  i += 1
59          return
```

-1.4.4 Python Built-in Data Structures

SET and Dictionary

In Python, we have the standard build-in data structure *dictionary* and *set* using hashtable. For the set classes, they are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the object defines both ___*eq*___() and ___*hash*___() methods. A Python built-in function *hash*(*object* =) is implementing the hashing function and returns an integer value as of the hash value if the object has defined ___*eq*___() and ___*hash*___() methods. As a result of the fact that *hash*() can only take immutable objects as input key in order to be hashable meaning it must be immutable and comparable (has an ___eq___() or ___cmp___() method).

Python 2.X VS Python 3.X In Python 2X, we can use slice to access keys() or items() of the dictionary. However, in Python 3.X, the same syntax will give us TypeError: 'dict_keys' object does not support indexing. Instead, we need to use function list() to convert it to list and then slice it. For example:

```
# Python 2.x
dict.keys()[0]

# Python 3.x
list(dict.keys())[0]
```

set Data Type

dict Data Type

If we want to put string in set, it should be like this:

```
>>> a = set('aardvark')
>>>
{'d', 'v', 'a', 'r', 'k'}
>>> b = {'aardvark'}# or set(['aardvark']), convert a list of
    strings to set
>>> b
{'aardvark'}
#or put a tuple in the set
a =set([tuple]) or {(tuple)}
```

Compare also the difference between and set() with a single word argument.

Collection Module

OrderedDict Standard dictionaries are unordered, which means that any time you loop through a dictionary, you will go through every key, but you are not guaranteed to get them in any particular order. The OrderedDict from the collections module is a special type of dictionary that keeps track

of the order in which its keys were inserted. Iterating the keys of an ordered-Dict has predictable behavior. This can simplify testing and debugging by making all the code deterministic.

defaultdict Dictionaries are useful for bookkeeping and tracking statistics. One problem is that when we try to add an element, we have no idea if the key is present or not, which requires us to check such condition every time.

```
dict = {}
key = "counter"
if key not in dict:
    dict[key]=0
dict[key] += 1
```

The defaultdict class from the collections module simplifies this process by pre-assigning a default value when a key does not present. For different value type it has different default value, for example, for int, it is 0 as the default value. A defaultdict works exactly like a normal dict, but it is initialized with a function ("default factory") that takes no arguments and provides the default value for a nonexistent key. Therefore, a defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory. For example, the following code use a lambda function and provide 'Vanilla' as the default value when a key is not assigned and the second code snippet function as a counter.

```
from collections import defaultdict
ice_cream = defaultdict(lambda: 'Vanilla')
ice_cream['Sarah'] = 'Chunky Monkey'
ice_cream['Abdul'] = 'Butter Pecan'
print ice_cream['Sarah']
# Chunky Monkey
print ice_cream['Joe']
# Vanilla
```

```
from collections import defaultdict
dict = defaultdict(int) # default value for int is 0
dict['counter'] += 1
```

There include: Time Complexity for Operations Search, Insert, Delete: $O(1)$.

Counter

-1.4.5 Exercises

1. 349. Intersection of Two Arrays (easy)
2. 350. Intersection of Two Arrays II (easy)

929. Unique Email Addresses

```
Every email consists of a local name and a domain name, separated by the @ sign.

For example, in alice@leetcode.com, alice is the local name, and leetcode.com is the domain name.

Besides lowercase letters, these emails may contain '.'s or '+'s.

If you add periods ('.') between some characters in the local name part of an email address, mail sent there will be forwarded to the same address without dots in the local name. For example, "alice.z@leetcode.com" and "alicez@leetcode.com" forward to the same email address. (Note that this rule does not apply for domain names.)

If you add a plus ('+') in the local name, everything after the first plus sign will be ignored. This allows certain emails to be filtered, for example m.y+name@email.com will be forwarded to my@email.com. (Again, this rule does not apply for domain names.)

It is possible to use both of these rules at the same time.

Given a list of emails, we send one email to each address in the list. How many different addresses actually receive mails?

Example 1:

Input: ["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","testemail+david@lee.tcode.com"]
Output: 2
Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.com" actually receive mails

Note:
    1 <= emails[i].length <= 100
    1 <= emails.length <= 100
    Each emails[i] contains exactly one '@' character.
```

Answer: Use hashmap simply Set of tuple to save the corresponding sending exmail address: local name and domain name:

```
class Solution:
    def numUniqueEmails(self, emails):
        """
        :type emails: List[str]
        :rtype: int
        """
        if not emails:
            return 0
        num = 0
        handledEmails = set()
```

```
11          for email in emails:
12              local_name, domain_name = email.split('@')
13              local_name = local_name.split('+')[0]
14              local_name = local_name.replace('.','')
15              handledEmails.add((local_name,domain_name) )
16          return len(handledEmails)
```