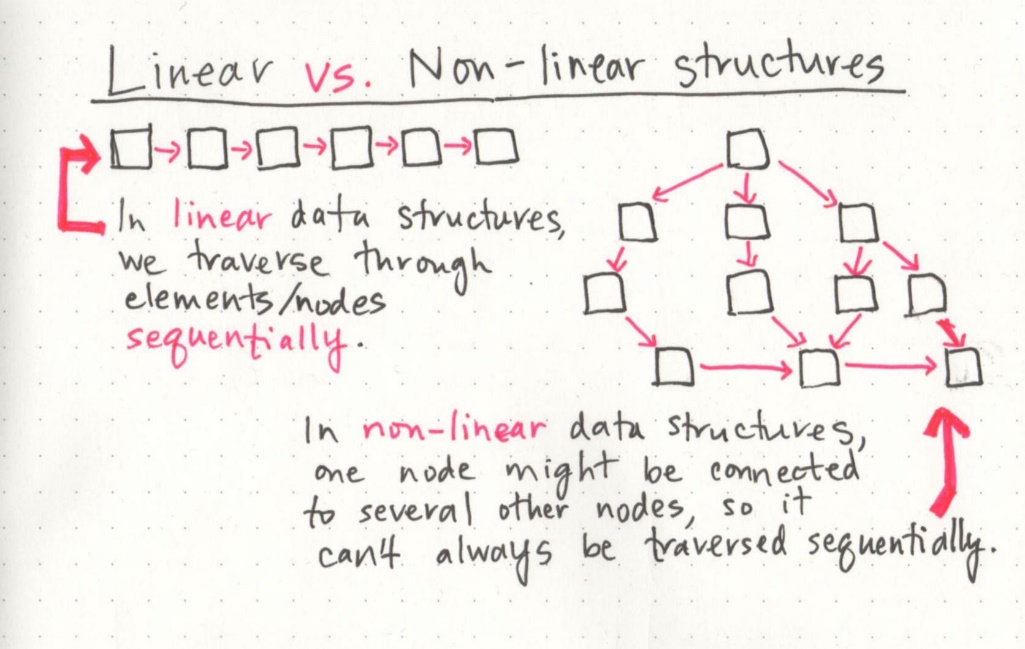
**线性数据结构 vs 非线性数据结构**

线性数据结构(Liner structures)代表着数据以线性排列，其特点是数据元素之间存在一对一的线性关系。线性结构拥有两种不同的存储结构，即顺序存储结构和链式存储结构。顺序存储的线性表称为顺序表，顺序表中的存储元素是连续的，链式存储的线性表称为链表，链表中的存储元素不一定是连续的，元素节点中存放数据元素以及相邻元素的地址信息。

非线性数据结构(non-Liner structures)代表着数据以非线性方式排列，我们可以朝着各种方向遍历数据。常见的非线性数据结构为树，图，字典等。

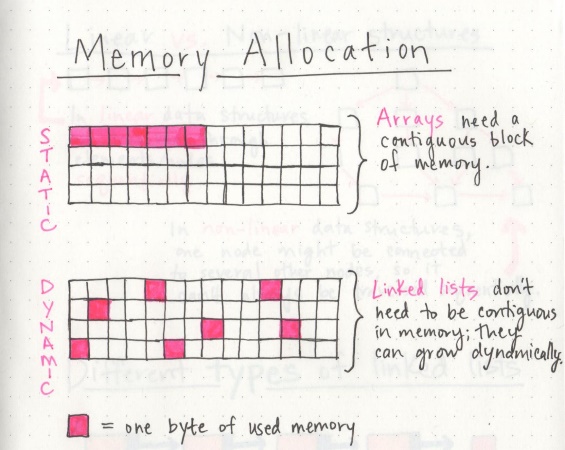


**数组(list) vs 链表(linkedList)内存空间对比**

数组和链表在内存分配(memory allocation)上有很大不同。虽然在Python, JavaScript, 以及Ruby等动态语言中，已经为数组提供灵活性，例如数组的长度可以变化等。但是事实上这些灵活性是这些动态编程语言提供的抽象，从而当数组改变时，让用户感觉不出内存空间的调整。

但是实质上数组是静态的数据类型，在申明数组变量的时候，便在开辟内存空间中连续，定长的空间。当数据插入数组，从而内存空间需要改变时，由这些动态编程语言来在内部首先回收该空间，再重新创建。

相比之下链表在内存空间中是分散的，非连续的，链表通过每个节点所指定的地址，找寻各个节点。



**数组(list) vs 链表(linkedList)性能对比**

🡪在读取任意位置的数据，搜索指定数据时:

数组由于在内存空间配置了连续的空间，所以可以使用index，复杂度O(1)来读取数组中任意位置的数据。搜索指定数据时，也只需在该连续空间内搜索。而链表由于保存在非连续的内存空间，需要依靠节点中的引用，进行跳转遍历链表，无法使用index. 所以在访问某位置的数据，都需要从HEAD节点开始，遍历链表逐一查找，复杂度为O(n)。

🡪在添加，删除数据时:

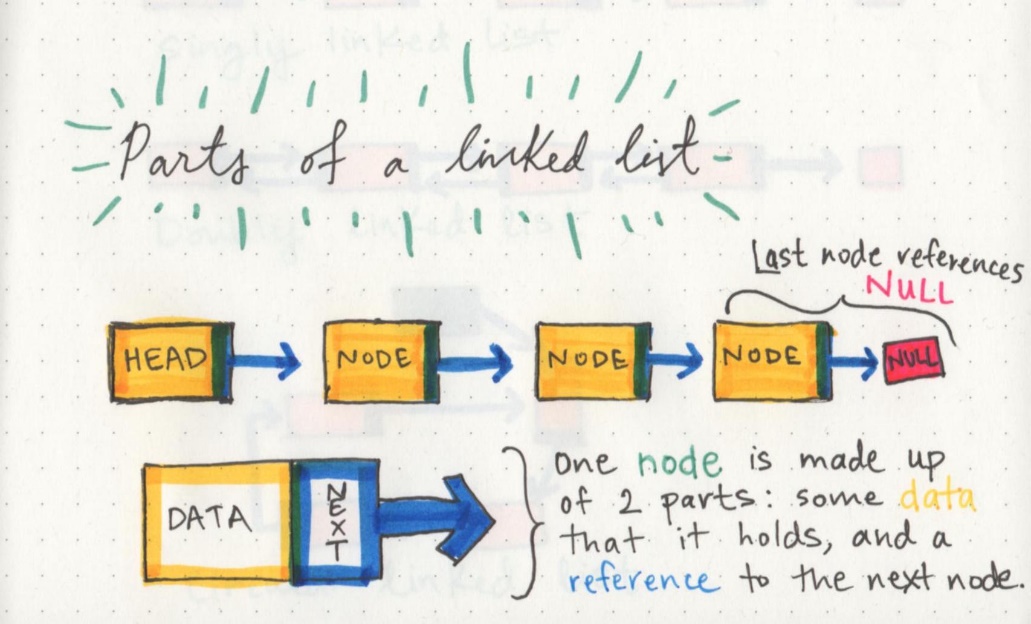
数组由于在内存空间配置了连续的空间，所以在添加数据时，如果超过了空间还需要重新创建空间。在删除数据时候，还要将后面数据进行整体上移，平均时间复杂度为O(n)。而链表由于依靠引用，当添加删除数据时候，只需要改变其中引用便可。不需要重新创建空间，也不需要任何对后面数据进行上移操作，时间复杂度为O(1).

**链表类型以及构成元素**

链表由一系列的节点组成，每一个节点存有数据以及引用。链表主要有三种类型，单向链表（singly linkedList）, 双向链表(doubly linkedList), 循环链表（circular linkedList）.

**单向链表（singly linkedList）**：

单向链表由一系列的节点构成。每个节点保存自己的数据，以及一个引用，该引用指向下一个节点。最后一个节点的引用指向NULL,表示是单向链表的结尾。并且单向链表的第一个节点称为HEAD节点，用于遍历单向链表的起点。由于每一个节点只有保存下一个节点地址，所以遍历单向链表时候只能以一个方向前进。每一个节点也只知道下一个邻居节点的引用地址。每个节点并不知道其他结点的情况，也不知道单向链表的长度。



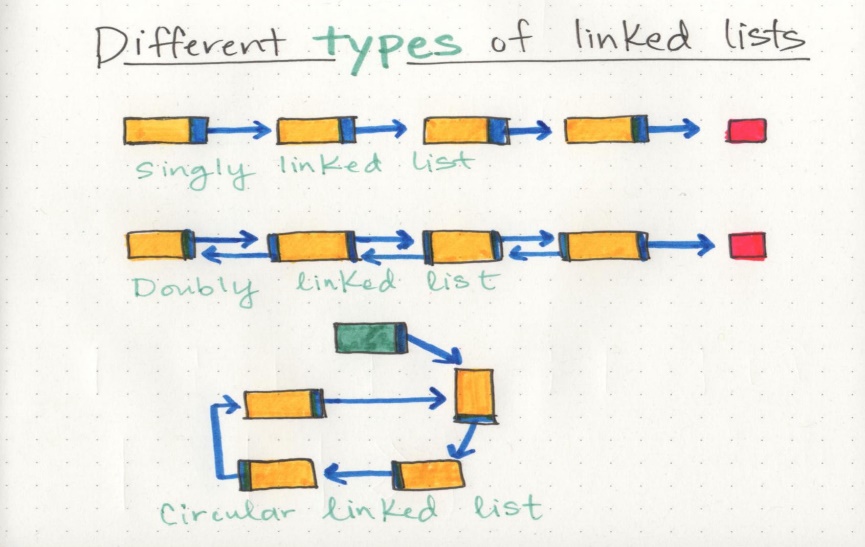
**双向链表(doubly linkedList)**：

双向链表由一系列的节点构成。每个节点保存由自己的数据，以及两个引用，一个引用指向下一个节点， 一个引用指向上一个节点。由于每个节点保存有上一个以及下一个结点的引用，所以遍历双向链表时候可以沿不同方向进行。

**循环链表（circular linkedList）**：

循环链表构建在单向链表的基础上，每个节点保存由自己的数据，以及一个引用，该引用指向下一个节点。不同之处在于，最后一个节点的引用指向首结点，形成封闭的循环。

循环链表的特点是我们可以在任一个节点开始进行完整的链表遍历。而相比之下单向链表如果需要进行完整的链表遍历，必须从HEAD结点开始。



**Python内置链表实现（collections.deque）**

我们可以使用Python自行实现链表数据结构，但同时Python对于链表这种基础的数据结构，Python已经提供了自带的默认实现。它存在于collections模块的deque类。

>>> from collections import deque # 从collections模块中默认引入Python自带实现链表deque

>>> llist = deque("abcde") # 实例化deque,可以传入一个iterable对象作为参数

>>> llist # 链表llist

deque(['a', 'b', 'c', 'd', 'e'])

>>> llist.append("f") # 链表自带append方法用于向链表末尾插入元素

>>> llist

deque(['a', 'b', 'c', 'd', 'e', 'f'])

>>> llist.pop() # 链表自带pop方法用于向对链表末尾移除一个元素

'f'

>>> llist

deque(['a', 'b', 'c', 'd', 'e'])

>>> llist.appendleft("z") # 链表自带appendleft方法用于向链表左侧加入元素

>>> llist

deque(['z', 'a', 'b', 'c', 'd', 'e'])

>>> llist.popleft() # 链表自带popleft方法用于向链表左侧移除一个元素

'z'

>>> llist

deque(['a', 'b', 'c', 'd', 'e'])

**基于Python自带链表collections.deque实现队列Queue**

队列数据结构拥有FIFO先进先出的特性。我们可以利用linkedList这个基本的数据结构构建出队列数据结构。在此，我们使用Python内置链表(collections.deque)链表实现一个队列。并且由于是基于链表实现的队列，所以相当于在内存中使用链式存储结构来实现队列这一线性数据结构。

>>> from collections import deque # 从collections模块中默认引入Python自带实现链表deque

>>> queue = deque() # 实例化空链表

>>> queue

deque([])

>>> queue.append("Mary") # 队列尾部加入元素

>>> queue.append("John") # 队列尾部加入元素

>>> queue.append("Susan") # 队列尾部加入元素

>>> queue

deque(['Mary', 'John', 'Susan'])

>>> queue.popleft() # 队列头部移除元素，达到FIFO特性

'Mary'

>>> queue

deque(['John', 'Susan'])

>>> queue.popleft() # 队列头部移除元素，达到FIFO特性

'John'

>>> queue

deque(['Susan'])

**基于Python自带链表collections.deque实现栈Stack**

栈数据结构拥有LIFO后进先出的特性。我们可以利用linkedList这个基本的数据结构构建出栈数据结构。在此，我们使用Python内置链表(collections.deque)链表实现一个栈。并且由于是基于链表实现的队列，所以相当于在内存中使用链式存储结构来实现栈这一线性数据结构。

>>> from collections import deque # 从collections模块中默认引入Python自带实现链表deque

>>> history = deque() # 实例化空链表

>>> history.appendleft("https://realpython.com/")

>>> history.appendleft("https://realpython.com/pandas-read-write-files/")

>>> history.appendleft("https://realpython.com/python-csv/")

>>> history

deque(['https://realpython.com/python-csv/',

'https://realpython.com/pandas-read-write-files/',

'https://realpython.com/'])

>>> history.popleft()

'https://realpython.com/python-csv/'

>>> history.popleft()

'https://realpython.com/pandas-read-write-files/'

>>> history

deque(['https://realpython.com/'])

**Python自行构建单向链表（Singly linkedList）**

除了collections.deque中Python内置实现的链表。我们在此使用Python从零开始实现一个单向链表(Singly linkedList)

**第一步：创建linkedList类，以及linkedList中节点类**

linkedList类：

class LinkedList:  
 def \_\_init\_\_(self, nodes=None): # 初始化时侯，可以接受一个iterable的nodes对象  
 self.head = None  
 if nodes is not None:  
 node = Node(data=nodes.pop(0)) # 从nodes中取出一个元素，构建一个节点并且作为LinkedList的头节点  
 self.head = node  
 for elem in nodes: # 此时nodes与传入nodes相比，已经从头部取出一个元素。对于剩下的元素，每一个元素都构建一个Node实例。并且把前一个Node的next引用指向最新创建的Node实例  
 node.next = Node(data=elem)  
 node = node.next

def \_\_repr\_\_(self): # 当字符串显示LinkedList时，从头结点依次遍历到链表末尾，最后打印输出  
 node = self.head  
 nodes = []  
 while node is not None:  
 nodes.append(node.data)  
 node = node.next  
 nodes.append("None")  
 return " -> ".join(nodes)

def \_\_iter\_\_(self): # 为LinkedList实现迭代特性。从而可以允许for i in LinkedList实例，每次迭代返回当前遍历的Node节点  
 node = self.head  
 while node is not None:  
 yield node  
 node = node.next

linkedList Node节点类：

class Node:  
 def \_\_init\_\_(self, data): # 对于每个linkedList节点保存数据以及下一个结点的引用  
 self.data = data  
 self.next = None  
  
 def \_\_repr\_\_(self): # 当字符串显示时候，返回自身数据   
 return self.data

**第二步，测试LinkedList基本用例：**

llist = LinkedList(["a", "b", "c", "d", "e"]) # 构建一个LinkedList实例  
print(llist) # 打印LinkedList实例，调用类实例中\_\_repr\_\_方法  
  
for node in llist: # 迭代LinkedList实例，调用类实例中\_\_iter\_\_方法  
 print(node)

**第三步，添加LinkedList类的基本操作（增加节点）**

🡪向LinkedList头部添加节点：

def add\_first(self, node): # 定义向LinkedList头部添加节点add\_first方法，接受node实例作为参数  
 node.next = self.head # 将LinkedList的HEAD指向该新增节点  
 self.head = node # 将该新增节点中的next指向原本LinkedList的HEAD节点

测试：

llist = LinkedList(["a", "b", "c", "d", "e"])  
llist.add\_first(Node("aa"))  
print(llist) # 输出aa -> a -> b -> c -> d -> e -> None

🡪向LinkedList尾部添加节点

def add\_last(self, node): # 定义向LinkedList尾部添加节点add\_last方法，接受node实例作为参数  
 if self.head is None: # 如果是空链表，则直接头部为新增节点  
 self.head = node  
 return  
 for current\_node in self: # 否则遍历链表，穷尽链表，最后一个current\_node变量变为LinkedList的最后一个节点。因为是链表，所以我们无法直接通过index,或者其他方法一步获取链表最后一个节点。我们需要完整遍历链表一次。  
 pass  
 current\_node.next = node # 将最后一个结点的next, 从None改为新增的节点

测试：

llist = LinkedList(["a", "b", "c", "d"])  
llist.add\_last(Node("ee"))  
print(llist) # 输出a -> b -> c -> d -> ee -> None

🡪向LinkedList插入节点

向LinkedList中间插入节点时，分两种情况。

情况1：向目标节点的后方插入节点。当向目标节点后方插入新节点时候，要求我们遍历链表，当找到目标节点后，目标节点的next引用指向插入节点，插入节点的next引用指向目标节点原本的next.

def add\_after(self, target\_node\_data, new\_node): # 接受参数1目标节点的data, 参数2目标节点Node实例  
 if self.head is None: # 如果链表为空则返回异常  
 raise Exception("List is empty")  
  
 for node in self:  
 if node.data == target\_node\_data: # 当找到目标节点以后  
 new\_node.next = node.next # 插入节点的next引用指向目标节点原本的next  
 node.next = new\_node # 目标节点的next引用指向插入节点  
 return  
  
 raise Exception("Node with data '%s' not found" % target\_node\_data) # 如果找不到目标节点则抛出异常

测试：

llist = LinkedList()  
llist.add\_after("a", Node("b")) # 向空链表插入，报错  
  
llist = LinkedList(["a", "b", "c", "d"])  
llist.add\_after("c", Node("cc"))  
print(llist) # 结果为a -> b -> c -> cc -> d -> None  
  
llist.add\_after("f", Node("g")) # 找不到目标节点f,报错

情况2：向目标结点的前方插入节点。当向目标节点后方插入新节点时候，要求我们遍历链表，当找到目标节点后，目标节点的前一个结点的next引用指向插入节点，插入节点的next引用指向目标节点。因为需要找到目标节点以后，获取目标节点之前的那个节点，所以方法中会保留一个变量，用于记录此次遍历节点的前一个结点。

def add\_before(self, target\_node\_data, new\_node): # 接受参数1目标节点的data, 参数2目标节点Node实例  
 if self.head is None: # 如果链表为空，则报错  
 raise Exception("List is empty")  
  
 if self.head.data == target\_node\_data: # 如果链表的头便是目标节点，则调用在链表头添加的方法  
 return self.add\_first(new\_node)  
  
 prev\_node = self.head # 开始遍历链表找寻目标节点，并且使用prev\_node变量，记录遍历节点的前一个结点。  
 for node in self:  
 if node.data == target\_node\_data: # 当找到目标节点以后  
 prev\_node.next = new\_node # 目标节点的前一个结点的next引用指向插入节点  
 new\_node.next = node # 插入节点的next引用指向目标节点。  
 return  
 prev\_node = node  
  
 raise Exception("Node with data '%s' not found" % target\_node\_data) # 找不到目标节点，报错

测试：

llist = LinkedList()  
llist.add\_before("a", Node("a")) # Exception: List is empty  
  
llist = LinkedList(["b", "c"])  
llist.add\_before("b", Node("a"))  
print(llist) # a -> b -> c -> None  
llist.add\_before("b", Node("aa"))  
llist.add\_before("c", Node("bb"))  
print(llist) # a -> aa -> b -> bb -> c -> None  
  
llist.add\_before("n", Node("m")) # Exception: Node with data 'n' not found

**第三步，添加LinkedList类的基本操作（删除节点）**

当需要删除链表目标节点的时候，要求我们遍历链表，当找到目标节点后，我们需要让目标节点的前一个结点的next引用指向目标节点的下一个节点，从而在链表中孤立目标节点，达到删除节点的目的。

def remove\_node(self, target\_node\_data): # 传入需要删除节点的data  
 if self.head is None: # 如果链表为空，则抛出异常  
 raise Exception("List is empty")  
  
 if self.head.data == target\_node\_data: # 如果删除的节点为头节点，则将链表的头节点指定为原头结点的next引用  
 self.head = self.head.next  
 return  
  
 previous\_node = self.head # 开始遍历链表找寻目标节点，并且使用prev\_node变量，记录遍历节点的前一个结点。  
 for node in self:  
 if node.data == target\_node\_data: # 当找到目标节点以后  
 previous\_node.next = node.next # 目标节点的前一个结点的next引用指向目标节点的下一个节点  
 return  
 previous\_node = node  
  
 raise Exception("Node with data '%s' not found" % target\_node\_data) # 找不到目标节点，报错

测试：

llist = LinkedList()  
llist.remove\_node("a") # Exception: List is empty  
llist = LinkedList(["a", "b", "c", "d", "e"])  
llist.remove\_node("a")  
print(llist) # b -> c -> d -> e -> None  
llist.remove\_node("e")  
print(llist) # b -> c -> d -> None  
llist.remove\_node("c")  
print(llist) # b -> d -> None  
llist.remove\_node("a") # Exception: Node with data 'a' not found