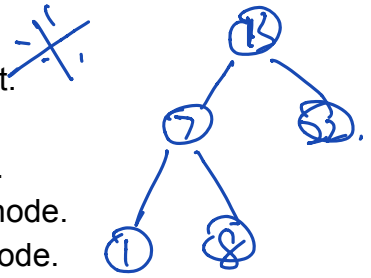
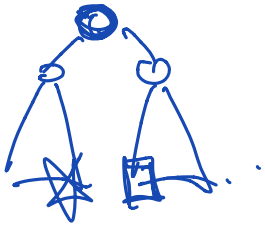
Section 7 Binary Search Tree

Binary Search Tree 简单来说是在 binary tree的基础上增加大小关系, 在binary search tree当中遵循着左边subtree一定小于root， root一定小于右边subtree的定式。所以在binary search tree当中， 如果我们需要找到一个node， 根据性质, 我们只需要检查单边就可以， 这样可以大大缩减running time.



1) 在BST中, inorder traversal可以返回一个sorted list.
2) 在常规BST中, 一般不存在duplicate value.
3) 对BST的任何操作都不能违反BST左小右大的原则.
4) 左边subtree 的最右是左边subtree中仅小于root的node.
5) 右边subtree的最左是右边subtree中仅大于root的node.

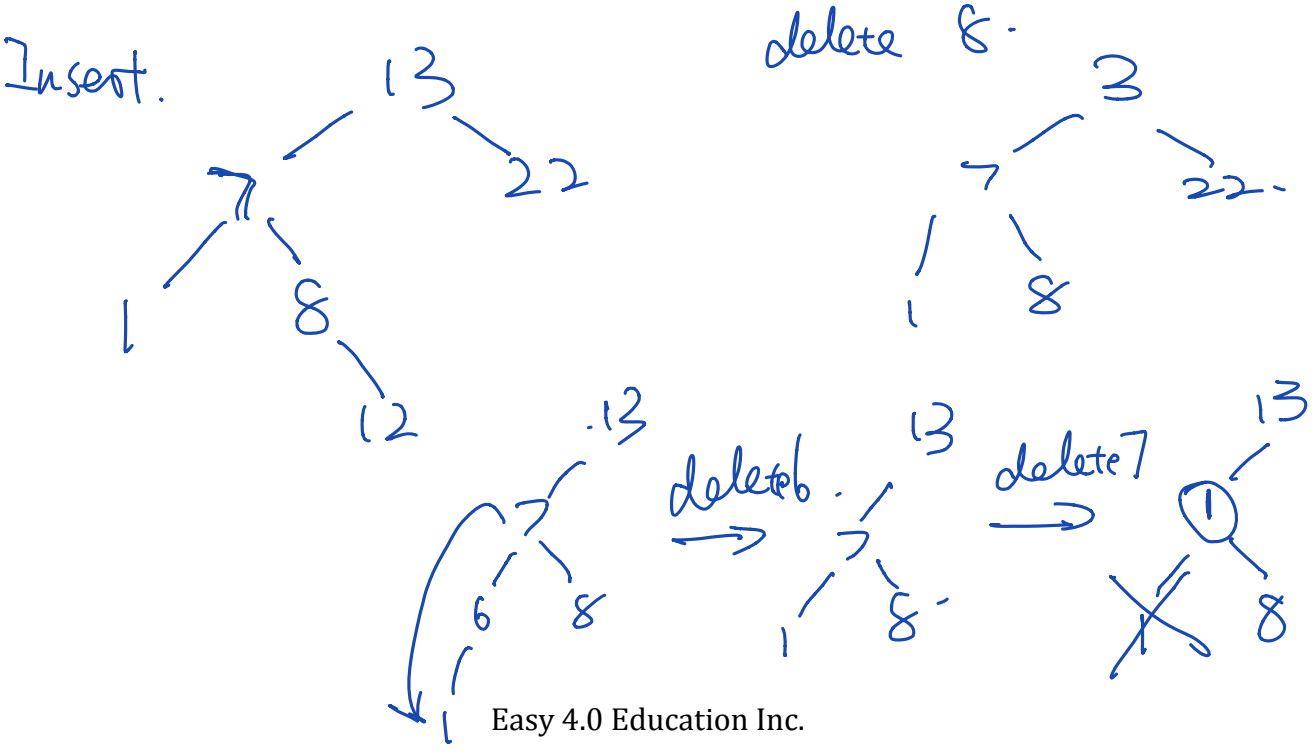


BST的insert

会从root开始按照左小右大原则向下寻找，加到leaf. 如果出现等于, 那么什么都不发生.

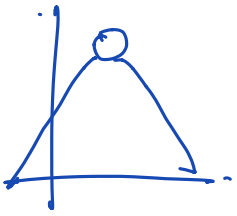
BST的delete

当两边都有时， 会使用左边subtree最大或者右边subtree最小 replace root.
当只有一边时, 会直接讲单边child 上移. 没有children时, 直接set 为None



BST的search

```
def contains(node: Union[BinaryTree, None], value: object) -> bool:
    """
    Return whether tree rooted at self contains value.

    >>> t = BinaryTree(5, BinaryTree(7), BinaryTree(9))
    >>> contains(t,5)
    True
    >>> contains(t,2)
    False
    >>> t1 = BinaryTree(5, BinaryTree(7,BinaryTree(3)), None)
    >>> contains(t1,1)
    False
    """
```

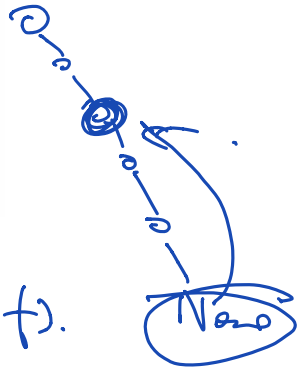


```
if not node:
    return False.
elif value < node.value:
    return contains(node.left, value)
elif value > node.value:
    return contains(node.right, value)
else:
    return True.    # node.value == value.
```

BST的Traversal

在BST中, 做题一定要使用到左小右大的性质, 并且记住 inorder 永远会得到一个sorted list, 做题过程中一旦遇到要求返回sorted list, 一定要使用inorder的顺序.

```
def largest_filtered_value(n: BTNode, f: 'bool function') -> object:
    """
    Return largest value in tree rooted at n that satisfies boolean function f, or None
    if there are no values that satisfy f.
    Assume the tree rooted at n is a binary search tree (BST).

    >>> def g(n: int) -> bool:
    ...     return n % 7 == 0 # is n a multiple of 7?
    ...
    >>> largest_filtered_value(BTNode(7, BTNode(0, None,None), BTNode(15, None, None)), g)
    7
    """
```




```
if not n:
    return None.
right_value = largest_filtered_value(n.right, f).
if right_value is not None:
    return right value.
if f(n.value): return n.value.
return largest_filtered_value(n.left, f)
```

```
def filter_nodes(n: BTNode, f: 'boolean function') -> list:
    """
    Return inorder list of values of tree rooted at n
    that satisfy boolean function f.

    >>> def h(n: int) -> bool:
    ...     return n % 5 == 0 # is n a multiple of 5?
    ...
    >>> filter_nodes(None, h)
    []
    >>> filter_nodes(BTNode(7, BTNode(0, None,None), BTNode(15, None, None)), h)
    [0, 15]
    >>> def g(n: int) -> bool:
    ...     return n % 7 == 0 # is n a multiple of 7?
    ...
    >>> filter_nodes(BTNode(7, BTNode(0, None,None), BTNode(15, None, None)), g)
    [0, 7]
    """
```

sorted list.

```
if not n:
    return []
acc = []
#Handle left.
acc += filter_nodes(n.left, f).
# Handle root.
if f(n.value):
    acc.append(n.value).
# Handle right
acc += filter_nodes(n.right, f)
return acc
```

L = L + [2]

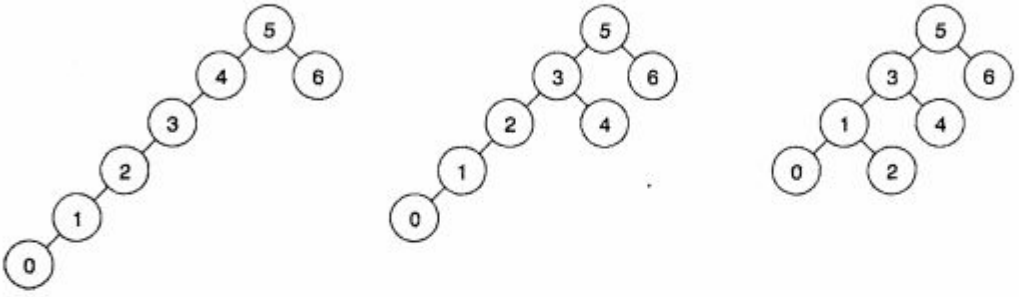
L += [2]

A *left streak* is defined as a sequence of three or more nodes on the leftmost side of the tree where none of the nodes in the sequence have a right child. A property of a left streak is that it can be broken to reduce the height of the tree by 1, while maintaining the binary search tree property. The *streak parent* is the node whose left child is the shallowest (closest to the root) node in the streak.

In the example below, the tree on the left contains a left streak whose parent is the node with data 5.

The tree in the middle has that streak fixed, but still contains a left streak whose parent is the node with data 3.

The tree on the right shows the same tree with all streaks fixed.

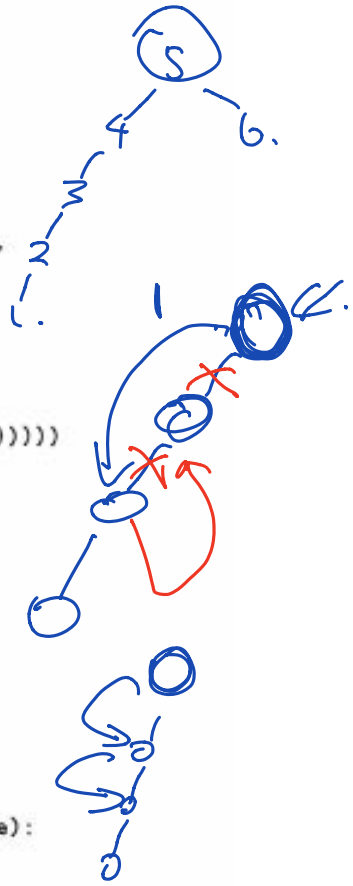


Your task is to implement the function that fixes all left streaks in a BST. You should use the provided helper function that finds a streak in your solution, and you do not need to handle any streaks that this helper function would not find.

Read the declaration of class BSTree and the find_left_streak helper function below and on the next page, and then implement function fix_left_streaks on Page 19. (Note: the height method has been included for use in a docstring example to help you understand the problem. We do not expect it to be useful to you in your solution.) Hint: draw diagrams!

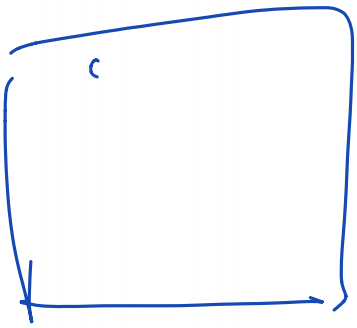
```
class BSTree:

    """ A simple binary search tree ADT """

    def __init__(self, data, left=None, right=None):
        """ Create BSTree self with content data, left child left,
        and right child right.

        @param BSTree self: this binary search tree
        @param object data: data contained in this tree
        @param BSTree left: left child of this tree
        @param BSTree right: right child of this tree
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```

```
    # BSTree class continued

    def height(self):
        """ Return the number of nodes on the longest path from the tree
        rooted at this binary search tree to a leaf.

        @param BSTree self: this binary search tree
        @rtype: int
        """
        if self.left is None and self.right is None:
            return 1
        if self.left is None:
            return 1 + self.right.height()
        if self.right is None:
            return 1 + self.left.height()
        return 1 + max(self.left.height(), self.right.height())

    # end of the BSTree class
```



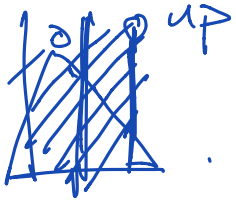
```
def find_left_streak(t):
    """ Return the parent node of the shallowest left streak in t,
    or None if there is no left streak.

    @param BSTree t: the root of the whole tree
    @rtype: BSTree|None

    >>> left = BSTree(4, (BSTree(3, BSTree(2, BSTree(1, BSTree(0))))))
    >>> t = BSTree(5, left, BSTree(6))
    >>> find_left_streak(t).data
    5
    >>> t.left.right = BSTree(4.5)
    >>> find_left_streak(t).data
    4
    """
    streak_parent = t
    while streak_parent.left is not None:
        node = streak_parent.left
        if (node.left is not None and node.right is None) and \
           (node.left.left is not None and node.left.right is None):
            return streak_parent
        streak_parent = node
    # if the end of the loop is reached, there is no left streak - return None
```

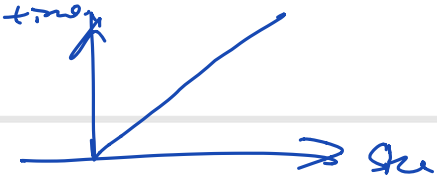
```
def fix_left_streaks(t):
    """ Modify t by fixing all left streaks.

    @param BSTree t: the tree to fix
    @rtype: None

    >>> left = BSTree(4, (BSTree(3, BSTree(2, BSTree(1, BSTree(0))))))
    >>> t = BSTree(5, left, BSTree(6))
    >>> t.height()
    6
    >>> t.left.right is None
    True
    >>> t.left.left.right is None
    True
    >>> fix_left_streaks(t)
    >>> t.height()
    4
    >>> t.left.right.data == 4
    True
    >>> t.left.left.right.data == 2
    True
    """
```




```
p = find_left_streak(t)
while p is not None:
    first = p.left
    second = first.left
    p.left = second
    second.right = first
    first.left = None
    p = find_left_streak(t)
```

```
def gather_in_between(t, low, up)
    """ Return the sorted list contains value in the binary search tree t between low and
    up.  exclusive.
    """
    if not t:
        return []
    acc = []
    # Handle left
    if low < t.value:
        acc += gather_in_between(t.left, low, up)
    # Handle root:
    if low < t.value < up:
        acc.append(t.value)
    # Handle right
    if up > t.value:
        acc += gather_in_between(t.right, low - up)
    return acc
```





Section 8 Runtime Complexity

Big-O notation: 描述程序的运行时间, 忽略constant, 忽略非dominate的项. 比如我们说O(n), 意图在于说这个function的running time的graph是类似于n的, 也就是linear的. 同时, 我们在148的卷子中, 也可能接触Big-Theta, 也就相当于我们在一些旧题中见到的 Big-O that best describes the runtime.

Worst-Case: 很多程序, 比如binary search tree contains, 他们的运行时间会由于一些condition变化, 比如function中存在可以快速结束的if-statement. worst-case描述的是在最坏的input family中的运行时间.

Best-Case: 类似于Worst-Case, 描述的是在最好的input family中的运行时间.

要注意的是, 我们在找Case的时候, 一定不要规定是某一个特定size, 比如 empty tree是best case一定是错的!

Sort the following functions (of positive integer n) in big-Oh order. I.e., write them in an order so that each function is in big-Oh of the one immediately after it. If functions are big-Oh of each other, then group them together using "==".

For example, given three functions f_1, f_2 and f_3 such that $f_1 \in \mathcal{O}(f_2)$ and $f_2 \in \mathcal{O}(f_1)$ and $f_1, f_2 \in \mathcal{O}(f_3)$, we would order them $f_1 == f_2, f_3$.

$$2n^3,\ 2^n + n^2 + \log n,\ 3^n,\ 4(\log n),\ 48,\ 7n^2,\ 92^n,\ \log(n^2)$$

48, $4(\log n)$ $\log(n^2)$, $7n^2$, $2n^3$, $2^n+n^2+\log n$ 3^n 92^n

基础Runtime 分析题型:

在基础runtime分析的题型中, 一般和108的题型类似, 需要搞清楚的就是funcion执行的次数, 乱入一部分CSC108的Package…

技巧总结:

并列相加, 缩进相乘, 循环次数数一数.

同一个缩进level的多个loop次数是相加的, loop之间的嵌套一定是相乘的, 里面转一圈, 外面走一下. 对于所有loop, 一定要注意数好loop的次数.

另外, 在分析的这种题的时候, 尽量少使用具体的loop次数, 而使用Big-O notation来表示.

```
sum, i = 0, 1
while 2 * i < n
    sum = sum + i
    i = 2 * i
```

$O(n)$ ← define n.

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

runtime is $O(\log_2 n)$.

every time. i is multiplied by 2. it takes $O(\log_2 n)$ times to finish the loop. Each iteration takes $O(1)$ As a result. $O(\log n)$

```
i, j, sum = 0, 1, 0
while i < n**2:  # a
    while j < n:  #b.
        sum = sum + i
        j = 2 * j
    i = i + n
```

$O(1)$ $O(\log_2 n)$ $O(\sqrt{n})$ **$O(n)$** $O(n \log_2 n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

while loop at #a takes $O(n)$ iterations. and the loop at #b only iterates $O(\log n)$ times at the first iteration of #a, and it takes $O(1)$. in other iterations. As a result, runtime is.

$O(n) + O(\log n) = O(n)$.

```
i, sum = 0, 0
while  (i // 2) < n:  #a
    if i % 2 == 0:
        for j in range(n):  #b   n.
            sum = sum + j
    else:
        for j in range(n**2):  #c  . n^2
            sum = sum + i
    i = i + 1
```

$O(1)$ $O(\log_2 n)$ $O(\sqrt{n})$ $O(n)$ $O(n \log_2 n)$ $O(n^2)$ **$O(n^3)$** $O(2^n)$

The while loop #a takes $O(n)$ iterations. In each iteration. it either execute. #b or #C. #c takes the most iterations as $O(n^2)$ in each iteration it takes $O(1)$. As a result. it takes $O(n^3)$

```
i, sum = 0, 0
while i < n * 2:
    sum = sum + i
    i = i + 1
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

The while loop takes O(n) times, each iteration takes O(1). as a result O(n)

Runtime records 分析题型:

这种题型会给出一些input size下的runtime records, 我们主要是通过对delta (增长幅度) 的分析, 来推测得出runtime. 还有一种比较常见的方法, 我们可以通过已有的records 画出图像来得出结果.

We measure the following running times for f(n):

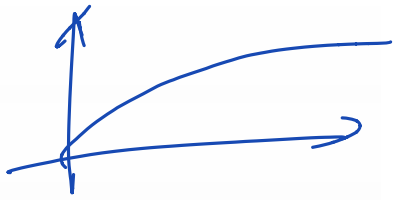
```
f(1):       1 second
f(10):      100 seconds
f(100):     10000 seconds
f(1000):    1000000 seconds
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n \lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

We measure the following running times for f(n):

```
f(2):     2 seconds
f(4):     4 seconds
f(8):     6 seconds
f(16):    8 seconds
f(32):    10 seconds
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n\lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$



Danny wants to get a mystery function's running time, however, he has no idea about csc148. So, he record the running times of the function under several input size.

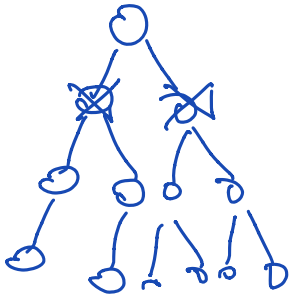
Input size	Running time
2	2.00134
3	4.75488
4	8.00267
5	11.60964
100	1000

nlogn

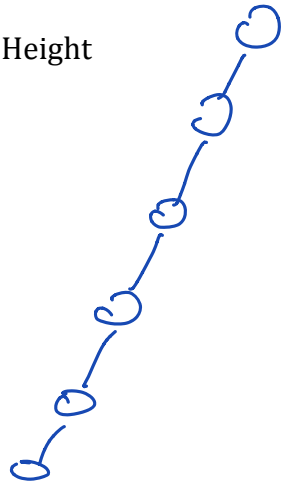
Tree 分析题型:

对于Tree而言, 一般来说runtime on each node都是O(1)的, 对于具体操作, 我们常见的有两种可能:

1. 是对所有node进行访问的, 这种一般没什么说的, size n就O(n)
2. 对一条path上所有node进行操作的, 那这个就关系到这条path上存在多少node了.

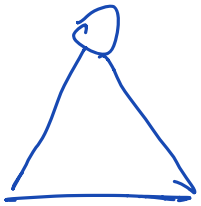


a. Tree Height



杨过

O(n)



O(log n)

$h = 2^0 + 2^1 + \cdots 2^{h-1}$

$h = 2^h - 1$ $h = O(\log n)$.

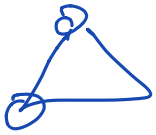
b. 常见Best-case scenario: 吃着火锅唱着歌, 刚开始就让return给停了

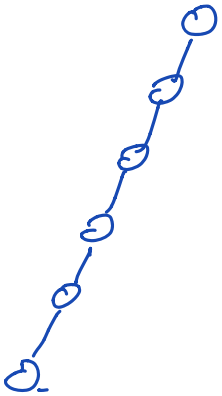
BST. insert:

For every size of tree. if the root is the place to insert the node (empty tree or root.value == value) it takes O(1) time to execute.

c. 常见worst-case scenario: 神雕大侠 杨过!

BST. insert

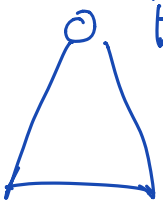




The tree is:

All nodes in the tree only have left children and the value to be inserted is smaller than all existing values. Then it takes O(n) time to go through all nodes and insert at leaf.

d. Tree is balanced.



BST is balanced (completed, full tree).

BEST Case 不变.

Worst Case O(height).

O(log n).

in a balanced BST, the height of tree is O(log n). It takes O(log n) to go through from root to leaf and ____.

Part I - 1. Quick Sort

Quick Sort, 快速排序, 取index 0 为pivot, 把比pivot 小的放左边, 比pivot大的放右边, 然后对他们分别quick sort 直到 length of sublist < 2.

```
def qs(list_):
    """
    Return a new list consisting of the elements of list_ in
    ascending order.

    @param list list_: list of comparables
    @rtype: list

    >>> qs([1, 5, 3, 2])
    [1, 2, 3, 5]
    """
    if len(list_) < 2:
        return list_[:]
    else:
        return (qs([i for i in list_ if i < list_[0]]) +
                [list_[0]] +
                qs([i for i in list_[1:] if i >= list_[0]]))
```

Runtime analysis:

qs([1, 2, 3, 4, 5]).

Worst-Case runtime: $O(n^2)$

[1] qs([2, 3, 4, 5]).

qs([3 4 5])

qs([4 5])

qs([5]).

Every pivot picked is either the smallest or the largest in the input list.

In worst case, it takes O(n) level of recursion steps. Each level has 1 less element than the caller. At most, it takes O(n) in each execution, as a result, it takes $O(n^2)$ in worst case.

Best-Case runtime:

[4 2 3 1 6 5 7]

[2 3 1] [4] [6 5 7].

[1] [2] [3] [5] [6] [7].

$O(\log n)$

Every pivot picked is the median of the input list. Then the list is evenly splitted in two parts. As shown, it only takes O(log n) levels to the base case. Every level it visits. O(n) nodes. spend O(1) on each. As a result it takes $O(n \log n)$ in best case.

Part I - 2. Merge Sort

Merge sort 分为两个phase.

Phase 1: Keep split the list evenly into two, until each chunk only contains one element.

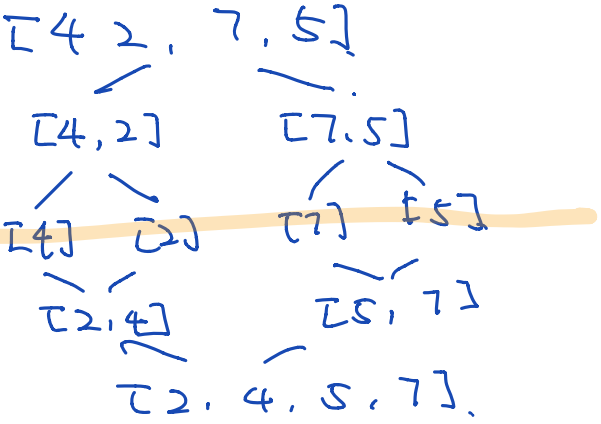
Phase 2: Merge two chunk at a time, keep the output sorted.

m n O(m+n)

[1, 3, 5]
[2, 4, 6]

1 2 3 4 5 6.

```
def merge(L1, L2):
    """return merge of L1 and L2

    >>> merge([1, 3, 5], [2, 4, 6])
    [1, 2, 3, 4, 5, 6]
    >>> merge([1, 2, 3], [0, 4, 5])
    [0, 1, 2, 3, 4, 5]
    >>> merge([0], [1, 2, 3, 4])
    [0, 1, 2, 3, 4]
    """
    L = []
    i1, i2 = 0, 0
    while i1 < len(L1) and i2 < len(L2):
        if L1[i1] < L2[i2]:
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    return L + L1[i1:] + L2[i2:]


def merge_sort(L):
    """Produce copy of L in non-decreasing order

    >>> merge_sort([1, 5, 3, 4, 2])
    [1, 2, 3, 4, 5]
    >>> L = list(range(20))
    >>> shuffle(L)
    >>> merge_sort(L) == list(range(20))
    True
    """
    if len(L) < 2 :
        return L[:]
    else :
        return merge(merge_sort(L[:len(L) // 2]),
                     merge_sort(L[len(L) // 2 :]))
```



Running time analysis:

it takes O(logn) levels to split the list into the base cases. and in each level it takes O(n) time to move the items. It takes O(nlogn) in total in splitting.

it takes O(logn) levels to merge all lists into the result. for each level all merge calls take O(n) in total. As a result it takes O(nlogn) to merge.

In total O(nlogn) for merge sort.

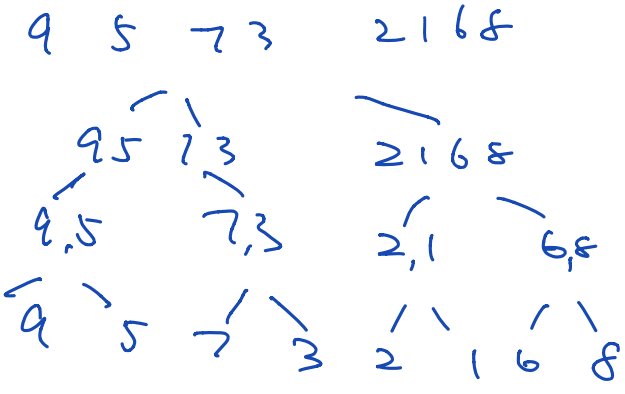
Consider running the `MergeSort` algorithm on the following list:

`[9,5,7,3,2,1,6,8]`

What sub-lists will `merge` be called on over the course of the sort?

For example, if `merge` will be called twice, once to merge the lists `[a,b,c]` and `[d,e,f]`, and once to merge the lists `[x,y,z]` and `[w]`, your answer would be:

```
merge([a,b,c],[d,e,f])
merge([x,y,z],[w])
```



merge([9], [5])
merge([7], [3])
merge ([5,9], [3,7])
merge ([2], [1]).
merge ([6], [8])
merge [[1,2], [6,8]].
merge ([3,5,7,9], [1,2,6,8]).

Part J. Hash

在之前我们介绍过可以对instance进行hash, 得到一个key 然后根据key, 我们可以快速的找到对应的instance所以在的位置. 这个function 我们叫它 hash function. 存储这些key对应的value的table, 我们叫它hash table.

Things can be hashed: Non-mutable build-in types & objects with hash function defined.

Example of using hash: Dictionary, Set

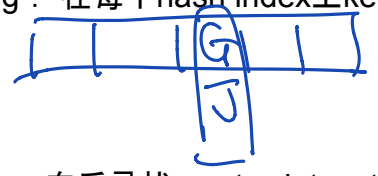
Hash的三大优势：

Fast	O(1) search insert.
Deterministic	相同的value一定会得到相同的hash key, 但是相同的hash key不一定代表相同的value (与 hash function 相关).
Well-distributed	Uniformly distributed

Collision：

在hash的时候有可能会出现撞车现象, 我们有两种解决方法

1. Chaining： 在每个hash index上keep一个list (linkedlist actually).



2. Probing： 向后寻找empty slot, extend the hash table if necessary (open addressing)
   Threshold: the maximum load percentage of hash table. Reach ➔ extend.

   a) Linear probing: check h(k), h(k) + 1, h(k + 2) ……

   b) Quadratic probing: check h(k), h(k) + 1, h(k) + 2, h(k) + 4 …..

   c) Double hashing: h1(k), h1(k) + h2(k), h1(k) + 2h2(k)

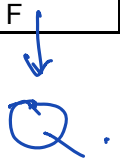
Question: given hash table as below

0	1	2	3	4	5	6	7
A	F	B				D	

Little Jacky got a Q, after he examine the hash key, he got a 1, use strategies below to add Q into the hash table.

a) Chaining

0	1	2	3	4	5	6	7
A	F	B				D	



b) Linear Probing

0	1	2	3	4	5	6	7
A	F	B				D	

c) Quadratic Probing with base 2.

0	1	2	3	4	5	6	7
A	F	B				D	

d) Second hash function with ~~gives 4.~~

0	1	2	3	4	5	6	7
A	F	B				D	