1. 插入排序：插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序；首先将第一个作为已经排好序的，然后每次从后的取出插入到前面并排序；
   - 时间复杂度：$O(n^2)$
   - 空间复杂度：$O(1)$
   - 稳定性：稳定

```python
def insert_sort(ilist):
    for i in range(len(ilist)):
        for j in range(i):
            if ilist[i] < ilist[j]:
                ilist.insert(j, ilist.pop(i))
                break
    return ilist

ilist = insert_sort([4,5,6,7,3,2,6,9,8])
print ilist
```

2. 希尔排序：希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止
   - 时间复杂度：$O(n)$
   - 空间复杂度：$O(n\sqrt{n})$
   - 稳定性：不稳定

```python
def shell_sort(slist):
    gap = len(slist)
    while gap > 1:
        gap = gap // 2
        for i in range(gap, len(slist)):
            for j in range(i % gap, i, gap):
                if slist[i] < slist[j]:
                    slist[i], slist[j] = slist[j], slist[i]
    return slist

slist = shell_sort([4,5,6,7,3,2,6,9,8])
print slist
```

3. 冒泡排序：它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成
   - 时间复杂度：$O(n^2)$
   - 空间复杂度：$O(1)$
   - 稳定性：稳定

```python
def bubble_sort(blist):
    count = len(blist)
    for i in range(0, count):
        for j in range(i + 1, count):
            if blist[i] > blist[j]:
                blist[i], blist[j] = blist[j], blist[i]
    return blist

blist = bubble_sort([4,5,6,7,3,2,6,9,8])
print blist
```

4. 快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列
   - 时间复杂度：$O(n\log_2 n)$
   - 空间复杂度：$O(n\log_2 n)$
   - 稳定性：不稳定

```python
def quick_sort(qlist):
    if qlist == []:
        return []
    else:
        qfirst = qlist[0]
        qless = quick_sort([l for l in qlist[1:] if l < qfirst])
        qmore = quick_sort([m for m in qlist[1:] if m >= qfirst])
        return qless + [qfirst] + qmore

qlist = quick_sort([4,5,6,7,3,2,6,9,8])
print qlist
```

```python
quick_sort = lambda array: array if len(array) <= 1 else quick_sort(
    [item for item in array[1:] if item <= array[0]]) + [array[0]] + quick_sort(
    [item for item in array[1:] if item > array[0]])
```

5. 选择排序：第1趟，在待排序记录r1 ~ r[n]中选出最小的记录，将它与r1交换；第2趟，在待排序记录r2 ~ r[n]中选出最小的记录，将它与r2交换；以此类推，第i趟在待排序记录r[i] ~ r[n]中选出最小的记录，将它与r[i]交换，使有序序列不断增长直到全部排序完毕
   - 时间复杂度：$O(n^2)$
   - 空间复杂度：$O(1)$
   - 稳定性：不稳定

```python
def select_sort(slist):
    for i in range(len(slist)-1):
        min_ = i
        for j in range(i+1, len(slist)):
            if slist[j] < slist[min_]:
                min_ = j
        slist[i], slist[min_] = slist[min_], slist[i]
    return slist

slist = select_sort([4,5,6,7,3,2,6,9,8])
print slist
```

6. 堆排序：它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素。堆分为大根堆和小根堆，是完全二叉树。大根堆的要求是每个节点的值都不大于其父节点的值，即A[PARENT[i]] >= A[i]。在数组的非降序排序中，需要使用的就是大根堆，因为根据大根堆的要求可知，最大的值一定在堆顶
   - 时间复杂度：$O(n\log_2 n)$
   - 空间复杂度：$O(1)$
   - 稳定性：不稳定

```python
import copy

def heap_sort(hlist):
    def heap_adjust(parent):
        child = 2 * parent + 1  # left child
        while child < len(heap):
            if child + 1 < len(heap):
                if heap[child + 1] > heap[child]:
                    child += 1  # right child
            if heap[parent] >= heap[child]:
                break
            heap[parent], heap[child] = heap[child], heap[parent]
            parent, child = child, 2 * child + 1

    heap, hlist = copy.copy(hlist), []
    for i in range(len(heap) // 2, -1, -1):
        heap_adjust(i)
    while len(heap) != 0:
        heap[0], heap[-1] = heap[-1], heap[0]
        hlist.insert(0, heap.pop())
        heap_adjust(0)
    return hlist

hlist = heap_sort([4,5,6,7,3,2,6,9,8])
print hlist

```

7. 基数排序：透过键值的部份资讯，将要排序的元素分配至某些“桶”中，藉以达到排序的作用
   - 时间复杂度：$O(d(r+n))$
   - 空间复杂度：$O(rd+n)$
   - 稳定性：稳定

```python
def radix_sort(array):
    bucket, digit = [[]], 0
    while len(bucket[0]) != len(array):
        bucket = [[], [], [], [], [], [], [], [], [], []]
        for i in range(len(array)):
            num = (array[i] // 10 ** digit) % 10
            bucket[num].append(array[i])
        array.clear()
        for i in range(len(bucket)):
            array += bucket[i]
        digit += 1
    return array

```

8. 归并排序：采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并
   - 时间复杂度：$O(n\log_2 n)$
   - 空间复杂度：$O(1)$
   - 稳定性：稳定

```python

def merge_sort(array):
    def merge_arr(arr_l, arr_r):
        array = []
        while len(arr_l) and len(arr_r):
            if arr_l[0] <= arr_r[0]:
                array.append(arr_l.pop(0))
            elif arr_l[0] > arr_r[0]:
                array.append(arr_r.pop(0))
        if len(arr_l) != 0:
            array += arr_l
        elif len(arr_r) != 0:
            array += arr_r
        return array

    def recursive(array):
        if len(array) == 1:
            return array
        mid = len(array) // 2
        arr_l = recursive(array[:mid])
        arr_r = recursive(array[mid:])
        return merge_arr(arr_l, arr_r)

    return recursive(array)
```

二叉树

```python
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# @Time    : 2018/12/4 10:15
# @Author  : hbw
# @File    : 二叉树.py
```

```python
# @Software: PyCharm
# 构造节点对象
class Node(object):
    def __init__(self,item):
        self.item=item
        self.left=None
        self.right=None
# 构造二叉树对象
class Tree(object):
    # 初始化一棵树  根节点为空
    def __init__(self):
        self.root=None

    # 添加节点
    def add(self,item):
        # 创造节点
        node=Node(item)
        # 判断根节点是否为空
        if self.root is None:
           #为空  赋值
           self.root=node
           return
        # 不为空 将根节点放入队列中
        q=[self.root]
        while True:
        # 从队列中取出一个节点
            cur_node=q.pop(0)
        # 判断左子树
            if cur_node.left is None:
                # 左子树为空  填入
                cur_node.left=node
                return
        # 判断右子树
            elif cur_node.right is None:
                cur_node.right=node
                return
            else:
                q.append(cur_node.left)
                q.append(cur_node.right)
    # 先序遍历
    def preorder(self,root):
        if root is None:
            return []
        result=[root.item] # 遍历根节点
        left=self.preorder(root.left) # 遍历左子树
        right=self.preorder(root.right)  # 遍历右子树
        return result+left+right
    # 中序遍历
    def inorder(self,root):
        if root is None:
            return []
        result=[root.item] # 遍历根节点
        left=self.inorder(root.left) # 遍历左子树
        right=self.inorder(root.right)  # 遍历右子树
        return left+result+right

    # 后序遍历
```

```python
    def postorder(self, root):
        if root is None:
            return []
        result = [root.item]  # 遍历根节点
        left = self.postorder(root.left)  # 遍历左子树
        right = self.postorder(root.right)  # 遍历右子树
        return left + right + result
    # 层次遍历 （广度优先）
    def cengci(self):
        if self.root is None:
            return []
        else:
            results=[]
            q=[self.root]
            while q:  # 当待访问的队列为空时，结束循环
                node=q.pop(0) # 取出当前节点
                results.append(node.item)
                if node.left is not None:
                    q.append(node.left)
                if node.right is not None:
                    q.append(node.right)
            return results
if __name__ == '__main__':
    tree=Tree()
    for i in  range(9):
        tree.add(i+1)
    print(tree.preorder(tree.root))
    print(tree.inorder(tree.root))
    print(tree.postorder(tree.root))
    print(tree.cengci())

```