```
Huffman示例代码
import heapq
from collections import defaultdict, namedtuple

class HuffmanNode(namedtuple("HuffmanNode", ["left", "right"])):
    def walk(self, code, acc):
        self.left.walk(code, acc + "0")
        self.right.walk(code, acc + "1")

class HuffmanLeaf(namedtuple("HuffmanLeaf", ["char"])):
    def walk(self, code, acc):
        code[self.char] = acc or "0"

def huffman_code(s):
    h = []
    for ch, freq in sorted((ch, s.count(ch)) for ch in set(s)):
        h.append((freq, len(h), HuffmanLeaf(ch)))
    heapq.heapify(h)
    count = len(h)
    while len(h) > 1:
        freq1, _count1, left = heapq.heappop(h)
        freq2, _count2, right = heapq.heappop(h)
        heapq.heappush(h, (freq1 + freq2, count, HuffmanNode(left, right)))
        count += 1
    code = {}
    if h:
        [(_freq, _count, root)] = h
        root.walk(code, "")
    return code

s = "this is an example for huffman encoding"
huff_code = huffman_code(s)
print(huff_code)
这段代码定义了霍夫曼编码算法。它首先统计每个字符的频率，然后使用一个最小堆（优先队列）构建
霍夫曼树，最后生成每个字符的霍夫曼编码。

Dijkstra 算法示例代码
python
import heapq

def dijkstra(graph, start):
    queue = [(0, start)]
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    while queue:
        current_distance, current_vertex = heapq.heappop(queue)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
```

```
                heapq.heappush(queue, (distance, neighbor))
    return distances

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

start_vertex = 'A'
print(dijkstra(graph, start_vertex))
这段代码实现了Dijkstra算法，用于计算图中从起始顶点到其他所有顶点的最短路径。它使用一个优先队列来有效地选择要处理的下一个顶点。

Prim 算法示例代码
python
import heapq

def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(cost, start, to) for to, cost in graph[start].items()]
    heapq.heapify(edges)
    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost))
            for to_next, cost in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost, to, to_next))
    return mst

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

start_vertex = 'A'
print(prim(graph, start_vertex))
这段代码实现了Prim算法，用于计算无向连通图的最小生成树（MST）。它从起始顶点开始，不断将最近的未访问顶点添加到生成树中。

二叉堆（Heap）示例代码
python
import heapq

heap = []
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

for item in data:
```

```
    heapq.heappush(heap, item)

sorted_data = []
while heap:
    sorted_data.append(heapq.heappop(heap))

print(sorted_data)
这段代码展示了如何使用Python的heapq模块创建一个最小堆，并将一组数据进行堆排序。heappush
将元素添加到堆中，heappop则从堆中取出最小元素。

二叉树示例代码
python
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val, end=" ")
        inorder_traversal(root.right)

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

inorder_traversal(root)
这段代码定义了一个简单的二叉树结构，并展示了中序遍历的实现方法。中序遍历按照左-根-右的顺序
访问节点。

AVL 树示例代码
python
class AVLNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.height = 1
        self.key = key

class AVLTree:
    def get_height(self, root):
        if not root:
            return 0
        return root.height

    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) - self.get_height(root.right)
```

```
    def right_rotate(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def left_rotate(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def insert(self, root, key):
        if not root:
            return AVLNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        root.height = 1 + max(self.get_height(root.left),
self.get_height(root.right))
        balance = self.get_balance(root)

        if balance > 1 and key < root.left.key:
            return self.right_rotate(root)
        if balance < -1 and key > root.right.key:
            return self.left_rotate(root)
        if balance > 1 and key > root.left.key:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)
        if balance < -1 and key < root.right.key:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)
        return root

    def pre_order(self, root):
        if not root:
            return
        print(root.key, end=" ")
        self.pre_order(root.left)
        self.pre_order(root.right)

avl = AVLTree()
root = None
keys = [10, 20, 30, 40, 50, 25]

for key in keys:
```

```
    root = avl.insert(root, key)

avl.pre_order(root)
这段代码展示了AVL树的插入和旋转操作。AVL树是一种自平衡二叉搜索树，通过插入和旋转保持树的平衡状态，从而保证搜索、插入、删除操作的时间复杂度为O(log n)。

有向图示例代码
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs_util(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbor in self.graph[v]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, v):
        visited = set()
        self.dfs_util(v, visited)

g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

g.dfs(2)
```