一、树

树（Tree）是 n（n ≥ 0）个结点的有限集。

n = 0 时称为空树。

在任意一颗非空树中：

- 有且仅有一个特定的成为根（Root）的结点；
- 当 n > 1 时，其余节点可分为 m（m > 0）个互不相交的有限集 T1、T2、.....、Tm，其中每一个集合本身又是一颗树，并且称为根的子树（SubTree）。

  注意：

- n > 0 时，根结点是唯一的，不可能存在多个根结点。
- m > 0时，子树的个数没有限制，但是他们一定是互不相交的。

二、二叉树

性质1：二叉树第i层上的结点数目最多为 $\mathbf{2^{\{i-1\}}}$ (i≥1)。

证明：下面用"数学归纳法"进行证明。
(01) 当i=1时，第i层的节点数目为$2^{\{i-1\}}=2^{\{0\}}=1$，命题成立。
(02) 假设当i>1，第i层的节点数目为$2^{\{i-1\}}$。这个是根据第一步推断出来的！
下面根据这个假设，推断出"第(i+1)层的节点数目为$2^{\{i\}}$"即可。由于二叉树的每个结点至多有两个孩子，故"第(i+1)层上的结点数目" 最多是 "第i层的结点数目的2倍"。即，第(i+1)层上的结点数目最大值=$2\times2^{\{i-1\}}=2^{\{i\}}$。故假设成立，原命题得证！

性质2：深度为k的二叉树至多有$\mathbf{2^{\{k\}}-1}$个结点(k≥1)。

证明：在具有相同深度的二叉树中，当每一层都含有最大结点数时，结点数最多。利用"性质1"可知，深度为k的二叉树的结点数至多为：$2^0+2^1+...+2^{k-1}=2^k-1$。故原命题得证！

性质3：包含n个结点的二叉树的高度至少为$\mathbf{\log_2 (n+1)}$。

证明：根据"性质2"可知，高度为h的二叉树最多有$2^{\{h\}}-1$个结点。反之，对于包含n个节点的二叉树的高度至少为$\log_2(n+1)$。

性质4：二叉树中,设叶子结点数为$\mathbf{n_0}$，度为2的结点数为$\mathbf{n_2}$，则$\mathbf{n_0=n_2+1}$。

证明：因为二叉树中所有结点的度数均不大于2，所以有等式一。

$n=n_0+n_1+n_2$(等式一)

另一方面，0度结点没有孩子，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是：n_1+2n_2。此外，只有根不是任何结点的孩子。故二叉树中的结点总数又可表示为等式二。

$n=n_1+2n_2+1$(等式二)

由(等式一)和(等式二)计算得到：$n_0=n_2+1$。原命题得证！

三、二叉搜索树

3.1 特点

1. 每个结点有唯一的值，且每个结点的值均不相同
2. 若它的左子树不为空，则它的左子树的所有结点均小于根节点的值
3. 若它的右子树不为空，则它的右子树的所有结点均大于根结点的值
4. 它的左右子树均为二叉搜索树。

3.2 操作

```
class Node:
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None

class BST:
    def __init__(self, node_list):
```

```python
        self.root = Node(node_list[0])
        for data in node_list[1:]:
            self.insert(data)

    # 搜索
    def search(self, node, parent, data):
        if node is None:
            return False, node, parent
        if node.data == data:
            return True, node, parent
        if node.data > data:
            return self.search(node.lchild, node, data)
        else:
            return self.search(node.rchild, node, data)

    # 插入
    def insert(self, data):
        flag, n, p = self.search(self.root, self.root, data)
        if not flag:
            new_node = Node(data)
            if data > p.data:
                p.rchild = new_node
            else:
                p.lchild = new_node

    # 删除
    def delete(self, root, data):
        flag, n, p = self.search(root, root, data)
        if flag is False:
            print "无该关键字，删除失败"
        else:
            if n.lchild is None:
                if n == p.lchild:
                    p.lchild = n.rchild
                else:
                    p.rchild = n.rchild
                del p
            elif n.rchild is None:
                if n == p.lchild:
                    p.lchild = n.lchild
                else:
                    p.rchild = n.lchild
                del p
            else:  # 左右子树均不为空
                pre = n.rchild
                if pre.lchild is None:
                    n.data = pre.data
                    n.rchild = pre.rchild
                    del pre
                else:
                    next = pre.lchild
                    while next.lchild is not None:
                        pre = next
                        next = next.lchild
                    n.data = next.data
                    pre.lchild = next.rchild
                    del p
```

```python
    # 先序遍历
    def preOrderTraverse(self, node):
        if node is not None:
            print node.data,
            self.preOrderTraverse(node.lchild)
            self.preOrderTraverse(node.rchild)

    # 中序遍历
    def inOrderTraverse(self, node):
        if node is not None:
            self.inOrderTraverse(node.lchild)
            print node.data,
            self.inOrderTraverse(node.rchild)

    # 后序遍历
    def postOrderTraverse(self, node):
        if node is not None:
            self.postOrderTraverse(node.lchild)
            self.postOrderTraverse(node.rchild)
            print node.data,
```

四、堆

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

4.1 堆

堆是一个二叉树，它的每个父节点的值都只会小于或等于所有孩子节点（的值）。 它使用了数组来实现：从零开始计数，对于所有的 *k* ，都有 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]` 。 为了便于比较，不存在的元素被认为是无限大。 堆最有趣的特性在于最小的元素总是在根结点：`heap[0]` 。

这个API与教材的堆算法实现有所不同，具体区别有两方面：（a）我们使用了从零开始的索引。这使得节点和其孩子节点索引之间的关系不太直观但更加适合，因为 Python 使用从零开始的索引。（b）我们的 pop 方法返回最小的项而不是最大的项（这在教材中称为“最小堆”；而“最大堆”在教材中更为常见，因为它更适用于原地排序）。

基于这两方面，把堆看作原生的Python list也没什么奇怪的：`heap[0]` 表示最小的元素，同时 `heap.sort()` 维护了堆的不变性！

要创建一个堆，可以使用list来初始化为 `[]` ，或者你可以通过一个函数 `heapify()` ，来把一个list转换成堆。

定义了以下函数：

- `heapq.heappush`(*heap, item*)

  将 *item* 的值加入 *heap* 中，保持堆的不变性。

- `heapq.heappop`(*heap*)

  弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出 `IndexError` 。使用 `heap[0]` ，可以只访问最小的元素而不弹出它。

- `heapq.heappushpop`(*heap, item*)

  将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用 `heappush()` 再调用 `heappop()` 运行起来更有效率。

- `heapq.heapify`(*x*)

  将list *x* 转换成堆，原地，线性时间内。

- `heapq.heapreplace`(*heap, item*)

  弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。 堆的大小不变。 如果堆为空则引发 `IndexError`。这个单步骤操作比 `heappop()` 加 `heappush()` 更高效，并且在使用固定大小的堆时更为适宜。 pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。返回的值可能会比添加的 *item* 更大。 如果不希望如此，可考虑改用 `heappushpop()`。 它的 push/pop 组合会返回两个值中较小的一个，将较大的值留在堆中。

该模块还提供了三个基于堆的通用功能函数。

- `heapq.merge`(**iterables, key=None, reverse=False*)

  将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。 返回已排序值的 iterator。类似于 `sorted(itertools.chain(*iterables))` 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。具有两个可选参数，它们都必须指定为关键字参数。*key* 指定带有单个参数的 key function，用于从每个输入元素中提取比较键。 默认值为 `None` (直接比较元素)。*reverse* 为一个布尔值。 如果设为 `True`，则输入元素将按比较结果逆序进行合并。 要达成与 `sorted(itertools.chain(*iterables), reverse=True)` 类似的行为，所有可迭代对象必须是已从大到小排序的。*在 3.5 版更改:* 添加了可选的 *key* 和 *reverse* 形参。

- `heapq.nlargest`(*n, iterable, key=None*)

  从 *iterable* 所定义的数据集中返回前 *n* 个最大元素组成的列表。 如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。 等价于: `sorted(iterable, key=key, reverse=True)[:n]`。

- `heapq.nsmallest`(*n, iterable, key=None*)

  从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。 如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。 等价于: `sorted(iterable, key=key)[:n]`。

后两个函数在 *n* 值较小时性能最好。 对于更大的值，使用 `sorted()` 函数会更有效率。 此外，当 `n==1` 时，使用内置的 `min()` 和 `max()` 函数会更有效率。 如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

4.2 二叉堆

使用“完全二叉树”来简化问题，减少复杂度，平衡的二 叉树树根左右子树有着相同数量的节点。
使用完全二叉树的特性，对于完全树，如果节点在列表中的位置为 p,那么其左子节点的位置为 2p,其 右子节点的位置为 2p+1。当我们要找任意节点的父节点时,可以直接利用 python 的整数除法。若节点在列表中的位置为 n,那么父节点的位置是 n//2。(之后上浮或者下沉都可以利用这一特性，来寻找嵌套的列表)

五、图

5.1图的表示

通常有两种表示方法，邻接表法和邻接矩阵表示。

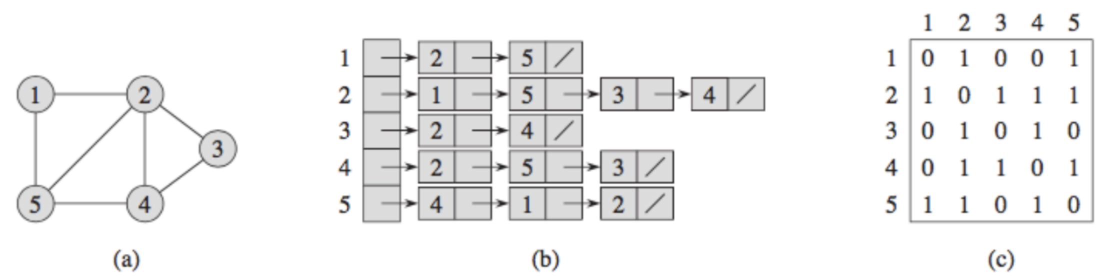
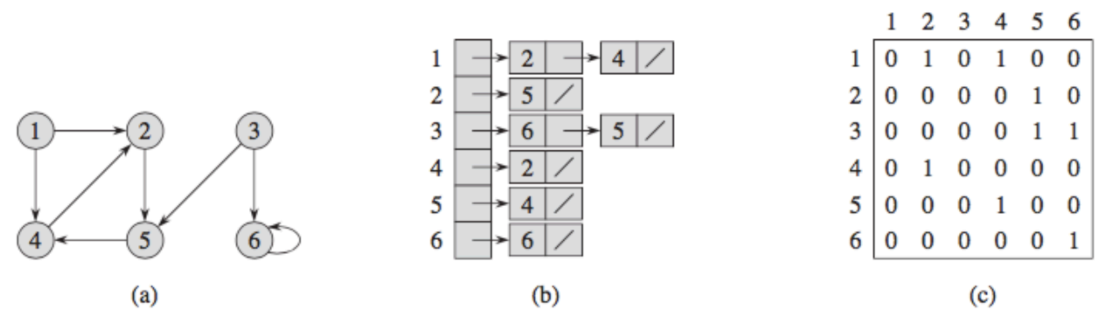


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



- 邻接表法：对于每个图中的点，将它的邻居放到一个链表里
- 邻接矩阵：对于 n 个点，构造一个 n * n 的矩阵，如果有从点 i 到点 j 的边，就将矩阵的位置 matrix[i][j] 置为 1.

<u>**大部分情况下矩阵是稀疏的，所以我们后边选择使用邻接表。**</u>

5.2：图的遍历

遍历图最常用的有两种方式，BFS 和 DFS.

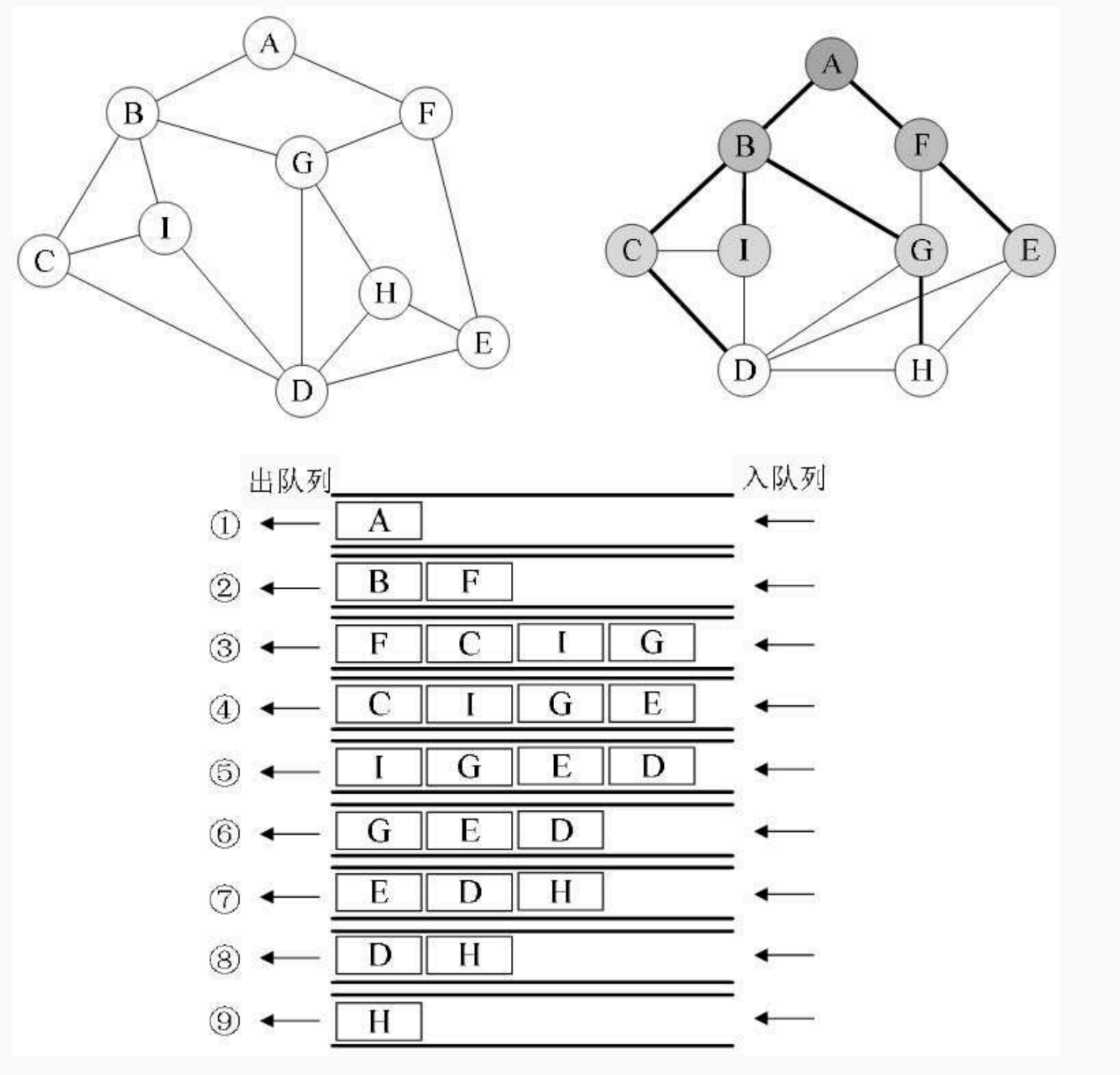
- BFS: Breadth First Search，广度优先搜索
- DFS: Depdth First Search，深度优先搜索

BFS

BFS 类似于树的层序遍历，从第一个节点开始，先访问离 A 最近的点，接着访问次近的点：

```python
graph = {
    'A': ['B', 'F'],
    'B': ['C', 'I', 'G'],
    'C': ['B', 'I', 'D'],
    'D': ['C', 'I', 'G', 'H', 'E'],
    'E': ['D', 'H', 'F'],
    'F': ['A', 'G', 'E'],
    'G': ['B', 'F', 'H', 'D'],
    'H': ['G', 'D', 'E'],
    'I': ['B', 'C', 'D'],
}
```

```python
# -*- coding: utf-8 -*-

from collections import deque


GRAPH = {
    'A': ['B', 'F'],
    'B': ['C', 'I', 'G'],
    'C': ['B', 'I', 'D'],
    'D': ['C', 'I', 'G', 'H', 'E'],
    'E': ['D', 'H', 'F'],
    'F': ['A', 'G', 'E'],
    'G': ['B', 'F', 'H', 'D'],
    'H': ['G', 'D', 'E'],
    'I': ['B', 'C', 'D'],
}


class Queue(object):
    def __init__(self):
        self._deque = deque()

    def push(self, value):
        return self._deque.append(value)

    def pop(self):
        return self._deque.popleft()

    def __len__(self):
        return len(self._deque)


def bfs(graph, start):
    search_queue = Queue()
    search_queue.push(start)
    searched = set()
    while search_queue:   # 队列不为空就继续
        cur_node = search_queue.pop()
        if cur_node not in searched:
            yield cur_node
            searched.add(cur_node)
            for node in graph[cur_node]:
                search_queue.push(node)
```



DFS

深度优先搜索(DFS)是每遇到一个节点，如果没有被访问过，就直接去访问它的邻居节点，不断加深。代码其实很简单：

```python
DFS_SEARCHED = set()


def dfs(graph, start):
    if start not in DFS_SEARCHED:
        print(start)
        DFS_SEARCHED.add(start)
    for node in graph[start]:
        if node not in DFS_SEARCHED:
            dfs(graph, node)


print('dfs:')
dfs(GRAPH, 'A')  # A B C I D G F E H
```