

第4章 图的遍历

本章复习图论的基本知识，介绍基本的搜索和遍历技术（包括二叉树、一般树及图的遍历和搜索技术）以及二叉树的简单应用（包括堆的基本知识和基本操作）。

- 图的基本概念和性质
- 图的表示方法
- 二叉树的遍历
- 堆的基本操作和实现方法*
- 一般树的遍历
- 图的宽度优先遍历
- 图的深度优先遍历

4.1 图的基本概念和性质

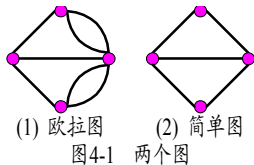
4.1.1 图的基本概念

1. 图的定义

图是一个二元组 $G=(V,E)$ ，其中， V 是一个非空有限的顶点集合， E 是一个有限的边集合，每条边对应一个顶点对。如图4-1所示。

如果多条边对应同一个顶点对，则称这些边为一组重边。如果一条边对应的顶点对的两个顶点相同，则称该边为一个自圈。没有重边和自圈的图称为简单图。如图4-1(2)所示。简单图的边通常用对应的顶点对表示，表示形式形如 (x,y) 或 xy 。

本书只讨论简单图。



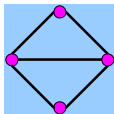
2. 边和顶点的关系

① 端点。一条边对应的2个顶点称为该边的端点。

② 关联。一条边对应2个顶点也可以说成该边关联或连接这2个顶点。

③ 相邻。与同一条边关联的顶点称为相邻（邻接）顶点，与同一个顶点关联的边称为相邻（邻接）边。

④ 度数。图中与顶点 v 关联的边数称为 v 的度数，记做 $d(v)$ 。图 $G=(V, E)$ 中顶点的度数与边数满足关系 $\sum_{v \in V} d(v) = 2|E|$ 。由此可知，图中奇度数的顶点必定是偶数个。



3. 路径

使用图4-2(1)所示的图为例说明。

① 途径。一条途径是一个顶点序列 $v_0v_1v_2\cdots v_n$ ，其中 $v_{i-1}v_i$ 是一条边， v_0 称为起点， v_n 称为终点，如图4-2(2)所示。

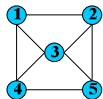
② 途径的长度。一条途径的长度是该途径包含的边数。

③ 闭途径。起点和终点相同的途径称为闭途径或回路，如图4-2(4)所示。

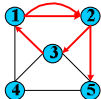
④ 路径。除起点和终点可以相同外不再出现相同顶点的途径称为路径或路，如图4-2(3)所示。

⑤ 圈。起点和终点相同的路径称为圈或环，如图4-2(4)所示。

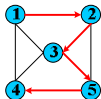
⑥ Hamilton圈。经过每个顶点正好一次的圈称为Hamilton圈，也可以称为Hamilton环或Hamilton回路，如图4-2(4)所示。



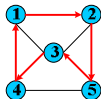
(1) 一个图



(2) 途径 $v_1v_2v_3v_1v_2v_5$



(3) 路径 $v_1v_2v_3v_5v_4$



(4) 圈 $v_1v_2v_5v_3v_4v_1$

图4-2 路径

4.1.2 无向图

如果边对应的顶点对是无序对，即顶点对 xy 和 yx 对应同一条边，则称该边是无向边。如果图中的边都是无向边，则称该图是一个无向图。

1. 空图

没有边的图称为空图，如图4-3中的K1。

2. 完全图

任何两个顶点都相邻的无向图称为完全图。 n 阶完全图有 n 个顶点，有 $n(n-1)/2$ 条边。如图4-3所示。

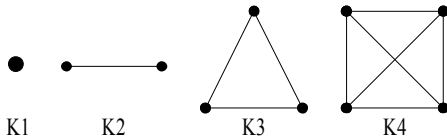


图4-3 1-4阶完全图

3. 连通图

① 可达性。称顶点 u 可达顶点 v 是指存在一条以 u 为起点、 v 为终点的途径（或路径）。

② 连通性。称一个无向图是连通的是指该图中任何两个顶点之间都是可达的。

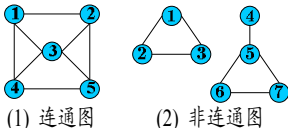


图4-4 连通图与非连通图

例如，图4-4(1)中的无向图是连通的，而图4-4(2)中的无向图不是连通的。

4. 补图

对于无向图 $G=(V,E)$ 和 $\bar{G}=(V,\bar{E})$ ，如果 $(u,v)\in\bar{E}$ 当且仅当 $(u,v)\notin E$ ，则称 \bar{G} 是 G 的补图。例如，在图4-5所示的图中， \bar{G} 是 G 的补图。

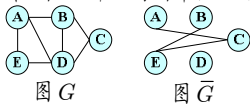


图4-5 一个图 G 及其补图 \bar{G}

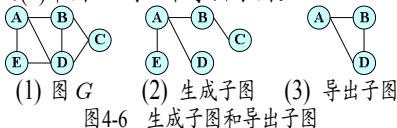
4.1.3 子图

1. 子图

(1) 子图。对于图 $G=(V,E)$ 和 $H=(V',E')$ ，如果 $V' \subseteq V$ ， $E' \subseteq E$ ，则称图 H 是图 G 的子图。例如，图4-6中的(2)和(3)分别给出了(1)中图 G 的两个子图。

(2) 生成子图。由原图去掉一些边（顶点集不变）得到的子图称为原图的生成子图。例如，图4-6(2)所示的图是图4-6(1)中图 G 的一个生成子图。

(3) 导出子图。由原图去掉一些顶点及与这些顶点关联的边得到的子图称为原图的导出子图，通常将导出子图的顶点集 U 称为原图的导出子图 U 。例如，图4-6(3)所示的图是图4-6(1)中图 G 的一个导出子图。

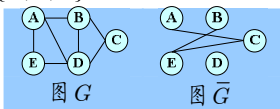


2. 空子图

如果图 G 的导出子图 U 是一个空图，则称 U 是 G 的空子图或 G 的独立集。独立集的顶点数称为它的大小。大小为 k 的独立集称为 k 独立集。

G 的独立集 U 是 G 的极大独立集当且仅当 U 不包含在 G 的更大独立集中。在如图4-5所示的图 \bar{G} 中， $\{A, B\}$ 是 \bar{G} 的2独立集。这个独立集不是极大独立集，因为它包含在3独立集 $\{A, B, D\}$ 中。

G 的最大独立集是指 G 中顶点数最多的独立集。在如图4-5所示的图 \bar{G} 中， $\{A, B, D\}$ 、 $\{A, D, E\}$ 和 $\{B, C, D\}$ 都是 \bar{G} 的最大独立集。



3. 完全子图

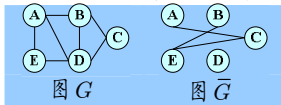
如果无向图 G 的导出子图 U 是一个完全图，则称 U 是 G 的完全子图或 G 的团。

团的顶点数称为它的大小。大小为 k 的团称为 k 团。

G 的团 U 是 G 的极大团当且仅当 U 不包含在 G 的更大团中。在如图4-5所示的无向图 G 中， $\{A, B\}$ 是 G 的2团。这个团不是极大团，因为它包含在3团 $\{A, B, D\}$ 中。

G 的最大团是指 G 中顶点数最多的团。在如图4-5所示的无向图 G 中， $\{A, B, D\}$ 、 $\{A, D, E\}$ 和 $\{B, C, D\}$ 都是 G 的最大团。

显然，对于无向图 G ， U 是 G 的 k 团当且仅当 U 是补图 \bar{G} 的 k 独立集，并且， U 是 G 的最大团当且仅当 U 是补图 \bar{G} 的最大独立集。

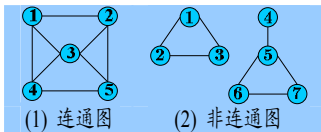


4. 连通子图

如果无向图 G 的一个子图 H 是一个连通图, 则称 H 是 G 的一个连通子图。

G 的连通子图 H 是 G 的连通分支(也称为极大连通子图)当且仅当 H 不是 G 的其它连通子图的子图。

非连通图可以分成几个连通分支。例如, 4-6(2)中的无向图不是连通的, 它有两个连通分支, 顶点集分别是 $\{1, 2, 3\}$ 和 $\{4, 5, 6, 7\}$ 。



4.1.4 有向图

如果边对应的顶点对是有序对，即顶点对 xy 和 yx 对应同不同的边，则称该边是有向边。如果图中的边都是有向边，则称该图是一个有向图。有向图实际上是在无向图的基础上进一步指定各个边的方向。如图4-7所示。

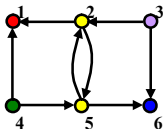


图4-7 有向图

注意，在有向图中，边、途径、闭途径、路径和圈等都是有序的。

1. 度数

在有向图中，顶点 v 的出度是指由顶点 v 出发的边数，记作 $d^+(v)$ ，而入度则是指向顶点 v 的边数，记作 $d^-(v)$ 。在有向图中，顶点的度是该顶点的出度与入度之和，即 $d(v) = d^+(v) + d^-(v)$ 。

2. 与无向图的关系

有向图和无向图可以相互转化。将一个无向图的每条边都规定方向后，即得到一个有向图，称为原图的一个定向图。将一个有向图的各条边的方向去掉，即得到一个无向图，称为原图的基础图。

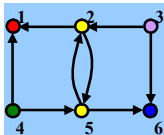
3. 连通性

① 连通。有向图是连通的是指其基础图是连通的。

② 可达。在有向图中，顶点 u 可达顶点 v 是指存在一条以 u 为起点、 v 为终点的途径或路径。

③ 双向连通。如果有向图 D 中任意两个顶点都是可达的，则说有向图 D 是双向连通的或强连通的。例如，图4-7所示的有向图是连通的，但不是双向连通的，因为没有到达顶点3的路径。

注意，双向连通不是双连通。双连通是一个针对无向图的概念。



4.1.5 树和森林

1. 树的定义和相关术语

不含圈的连通图称为树。如图4-8所示。

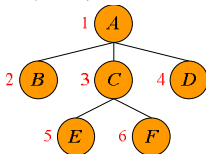


图4-8 树

通常在树中指定一个顶点作为树的根，与根相邻的顶点称为根的儿子，相应地，根称为这些儿子的双亲或父亲。如果在树中去掉根，则可以得到一些以原根的儿子为根的树，这些树称为原根的子树。没有儿子的顶点称为叶子，不是叶子的顶点称为非叶子顶点。双亲相同的顶点称为兄弟。从根到某个顶点的路径包含的顶点数称为该顶点的级或层次。顶点层次的最大值称为树的高度或深度。

2. 生成树

如果某连通图的一个生成子图是一棵树，则称这棵树为原图的生成树。在树中，顶点数 n 和边数 m 满足关系式 $m = n - 1$ 。生成树是其所在的无向图中边数最少的连通子图。因此，有 n 个顶点的连通图的边数至少是 $n - 1$ 。

3. 森林

森林是没有圈的无向图。森林的连通分支是树，也就是说，森林是由一些不相交的树组成的。非连通图没有生成树，但有生成森林。如图4-9所示。如果非连通图 G 有 k 个连通分支，则 G 的边数至少是 $n - k$ 。

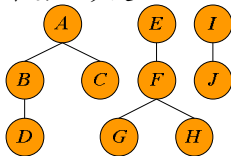
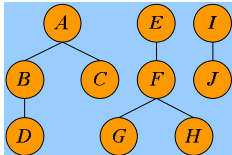


图4-9 森林

4. 相关性质

若无向图 G 的顶点数为 n ，边数为 m ，则下列结论成立。

- 若 G 是树，则 $m = n - 1$ 。
- 若 G 是连通图，而且满足 $m = n - 1$ ，则 G 是树。
- 若 G 不包含圈，而且满足 $m = n - 1$ ，则 G 是树。
- 若 G 是由 k 棵树构成的森林，则 $m = n - k$ 。
- 若 G 有 k 个连通分支，而且满足 $m = n - k$ ，则 G 是森林。



4.1.6 二叉树

1. 定义

① 二叉树。二叉树是一种特殊的树。在二叉树中，每个顶点至多有2个儿子，并且儿子有左、右之分，次序不能颠倒。图4-10给出了2棵二叉树。

② 满二叉树。叶子都在最大层上且非叶子顶点都有2个儿子的二叉树。如图4-10(1)所示。

③ 完全二叉树。所有叶子最多连续出现在相邻两级上的二叉树。显然，完全二叉树可以由满二叉树连续删除右边的若干叶子得到。如图4-10(2)所示。

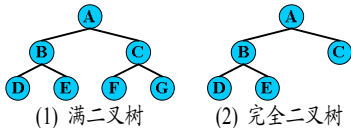


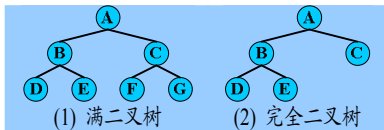
图4-10 二叉树

2. 基本性质

① 二叉树的第 i 层最多有 2^{i-1} 个顶点 ($i \geq 1$)。

② 深度为 k 的二叉树最多有 $2^k - 1$ 个顶点, 深度为 k 的满二叉树正好有 $2^k - 1$ 个顶点。

③ 含 n 个顶点的完全二叉树的深度不超过 $\log(2n)$ 。因为由完全二叉树的含义可知, 深度为 k 的完全二叉树的顶点数 n 不少于 2^{k-1} , 从而由 $2^{k-1} \leq n$ 可得 $k \leq \log(2n)$ 。



4.2 图的表示方法

这里只介绍图的邻接矩阵表示方法。树和二叉树具有较邻接矩阵更为方便的表示方法，这些表示方法将在后续部分介绍。

4.2.1 无向图的邻接矩阵

设无向图 G 的顶点集 $V = \{v_1, v_2, \dots, v_n\}$, 则顶点与顶点之间的邻接关系可以表示成邻接矩阵

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

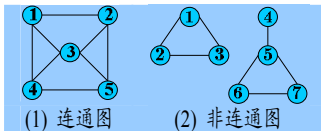
其中

$$a_{ij} = \begin{cases} 1 & v_i v_j \text{ 是边} \\ 0 & \text{否则} \end{cases}$$

例如，图4-4(2)所示的无向图对应的邻接矩阵为

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

显然，无向图的邻接矩阵一定是对称的。



4.2.2 有向图的邻接矩阵

有向图的邻接矩阵不一定对称。设有向图 D 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$ ，则顶点与顶点之间的邻接关系可以表示成邻接矩阵

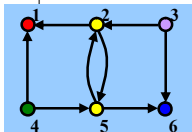
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

其中

$$a_{ij} = \begin{cases} 1 & v_i v_j \text{ 是有向边} \\ 0 & \text{否则} \end{cases}$$

例如，图4-7所示的有向图对应的邻接矩阵为

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



4.2.3 加权图的邻接矩阵

加权图(赋权图、带权图)是指对图(有向或无向)的每条边 e 赋予一个值 $w(e)$ (如图4-11所示)。例如,架设连接各城镇的交通路网需考虑各段线路的修建费用,在运输网络中要考虑网络各段线路的容量。

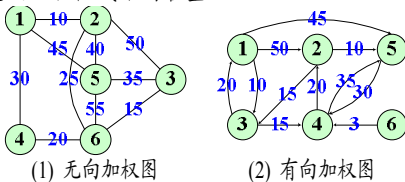


图4-11 加权图

1. 无向加权图的邻接矩阵

设无向加权图 G 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$, 则 G 的邻接矩阵为

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

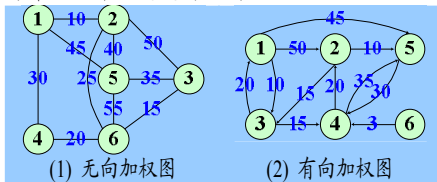
其中

$$a_{ij} = \begin{cases} \text{权值} & v_i v_j \text{ 是边} \\ \infty & v_i v_j \text{ 不是边} \\ 0 & v_i = v_j \end{cases}$$

例如，图4-11(1)所示的无向加权图对应的邻接矩阵为

$$A = \begin{pmatrix} 0 & 10 & \infty & 30 & 45 & \infty \\ 10 & 0 & 50 & \infty & 40 & 25 \\ \infty & 50 & 0 & \infty & 35 & 15 \\ 30 & \infty & \infty & 0 & \infty & 20 \\ 45 & 40 & 35 & \infty & 0 & 55 \\ \infty & 25 & 15 & 20 & 55 & 0 \end{pmatrix}$$

显然，无向加权图的邻接矩阵一定是对称的。



2. 有向加权图的邻接矩阵

设有向加权图 D 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$, 则 D 的邻接矩阵为

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

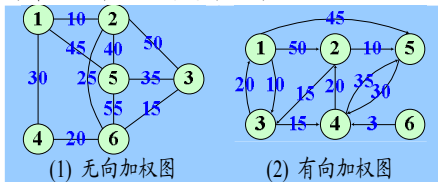
其中

$$a_{ij} = \begin{cases} \text{权值} & v_i v_j \text{ 是有向边} \\ \infty & v_i v_j \text{ 不是有向边} \\ 0 & v_i = v_j \end{cases}$$

例如，图4-11(2)所示的有向加权图对应的邻接矩阵为

$$A = \begin{pmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 10 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 35 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{pmatrix}$$

显然，有向加权图的邻接矩阵不一定是对称的。



4.2.4 矩阵的实现

后续章节一般使用邻接矩阵表示一个图，这里给出邻接矩阵的C++描述。保存在文件Matrix.hpp中。

```
// Matrix.hpp
#pragma once
#include <iostream>
#include <algorithm>
using namespace std;

template<class T> //
class Matrix
{   T *X; // 用一维数组保存矩阵元素
    int r, c; // 行数和列数
```

public:
Matrix(): r(0), c(0), X(0) { }
Matrix(int row, int col = 1): // 指定行数和列数 r(row), c(col), X(new T[row * col]) { }
Matrix(int row, int col, const T &v): Matrix(row, col) { fill_n(X, r * c, v); } // 指定行数, 列数和元素值, 耗时O(r*c)
Matrix(initializer_list<initializer_list<T>> m): // 不使用引用参数 Matrix(m.size(), begin(m)->size()) { // 使用初始化列表, 即使用{}初始化 auto W = X; for(auto il : m) W = copy_n(begin(il), c, W); } // 耗时O(r*c)
Matrix(const Matrix &m): Matrix(m.r, m.c) { copy_n(m.X, r * c, X); } // 使用另一个矩阵初始化, 耗时O(r*c)
~Matrix() { delete []X; }

```
void assign(const T &v) // 每个元素都是v
{
    fill_n(X, r * c, v);
} // 耗时O(r*c)
```

```
Matrix &operator=(const Matrix &m) // 复制
{
    if(this == &m) return *this; // 不能自己复制自己
    if(r * c < m.r * m.c) // 重新分配空间
        delete []X, X = new T[m.r * m.c];
    r = m.r, c = m.c;
    copy_n(m.X, r * c, X); // 复制元素
    return *this;
} // 耗时O(r*c)
```

```
T*operator[](int i) const // 便于使用二维数组的形式访问矩阵元素
{
    return X + i * c;
}
```

```
T&operator()(int i, int j) const // 使用M(i,j)的形式访问矩阵元素
{
    return X[i * c + j];
}
```

```
int Rows() const { return r; } // 行数
int Cols() const { return c; } // 列数
};
```



```
template<class T> // 输出矩阵的所有元素
ostream &operator<<(ostream &out, const Matrix<T> &m)
{   int r = m.Rows(), c = m.Cols();
    for(int i = 0; i < r; ++i)
    {   for(int j = 0; j < c; ++j) out << m[i][j] << ' ';
        out << endl; // 输出一行元素后换行
    }
    return out;
} // 耗时O(r*c)
```

4.3 二叉树的遍历

4.3.1 二叉树的遍历算法

对于二叉树的搜索可以分为先根次序搜索、中根次序搜索、后根次序搜索和按层次搜索等四种方式。

1. 二叉树的先根次序遍历

- ① 若T为空，则返回。
- ② 访问T的根。
- ③ 按二叉树的先根次序遍历T的左子树。
- ④ 按二叉树的先根次序遍历T的右子树。

图4-12演示了先根次序遍历二叉树的执行过程及结果。

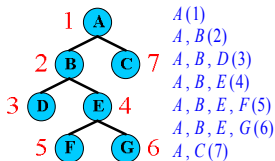


图4-12 二叉树的先根次序遍历

2. 二叉树的中根次序遍历

- ① 若T为空，则返回。
- ② 按二叉树的中根次序遍历T的左子树。
- ③ 访问T的根。
- ④ 按二叉树的中根次序遍历T的右子树。

图4-13演示了中根次序遍历二叉树的执行过程及结果。

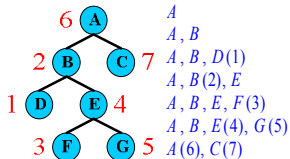


图4-13 二叉树的中根次序遍历

3. 二叉树的后根次序遍历

- ① 若T为空，则返回。
- ② 按二叉树的后根次序遍历T的左子树。
- ③ 按二叉树的后根次序遍历T的右子树。
- ④ 访问T的根。

图4-14演示了后根次序遍历二叉树的执行过程及结果。

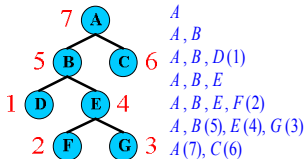
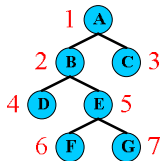


图4-14 二叉树的后根次序遍历

4. 二叉树的按层次遍历

- ① 若 T 非空，则访问 T 的根并将根加入队列。
- ② 若队列非空，取出队首元素 x 。
- ③ 访问 x 的左右儿子并将左右儿子加入队列。
- ④ 重复②~③，直到队列为空。

图4-15演示了按层次遍历二叉树的执行过程及结果。



x	队列
	A
A	B, C
B	C, D, E
C	D, E
D	E
E	F, G
F	G
G	

图 4-15 二叉树的按层次遍历

5. 复杂性分析

当树 T 有 n 个顶点时, 设 $t(n)$ 和 $s(n)$ 分别表示这些遍历算法中的任意一个算法所需要的最大时间和空间。如果访问一个顶点所需要的时间和空间是 $\Theta(1)$, 则 $t(n) = \Theta(n)$, $s(n) = O(n)$ 。

证明: 首先证明 $t(n) = \Theta(n)$ 。容易看出 $t(n) = \Omega(n)$ 。只需证明 $t(n) = O(n)$, 即存在 $c > 0$, 使得当 n 充分大时, 有 $t(n) \leq cn$ 。

遍历算法所做的工作由在递归这一级所做的工作以及递归调用所做的工作这两部分组成。第一部分工作要求的时间由常数 b 限界。当 T 的左子树顶点数为 n_1 时, $t(n) = \max_{n_1} \{t(n_1) + t(n - n_1 - 1) + b\}$ 。

下面使用数学归纳法证明 $t(n) \leq cn$ ，其中常数 $c \geq b$ 。

由 $t(0) \leq b$ 和 $t(1) \leq b$ 可知，当 $n=1$ 时 $t(n) \leq cn$ 成立。假定当 $1 \leq n < m$ 时， $t(n) \leq cn$ 成立，现在证明当 $n=m$ 时 $t(n) \leq cn$ 也成立。

设 T 是一棵 m 个顶点的树， m_1 是 T 的左子树的顶点数，于是

$$\begin{aligned} t(m) &= \max_{m_1} \{t(m_1) + t(m - m_1 - 1) + b\} \\ &\leq \max_{m_1} \{cm_1 + c(m - m_1 - 1) + b\} \\ &= \max_{m_1} \{cm + b - c\} \\ &\leq cm \end{aligned}$$

其次证明 $s(n) = O(n)$ 。只有为了保存递归调用时那些局部变量的值时，才需要附加空间。如果 T 的深度为 d ，则这个空间显然是 $\Theta(d)$ 。对于一棵 n 个顶点的二叉树，有 $d \leq n$ ，从而 $s(n) = O(n)$ 。

$$t(n) = \max_{n_1} \{t(n_1) + t(n - n_1 - 1) + b\}$$

4.3.2 二叉树遍历算法的C++实现

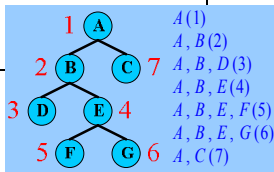
1. 二叉树的C++表示

```
#include "algorithm.h"

template<class T> // 二叉树结点
struct btnode
{
    T data;
    btnode *left = 0, *right = 0; // 左子树, 右子树
};
```

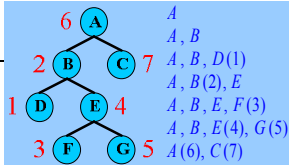
2. 二叉树的先根次序遍历

```
template<class T, class Func> // 先根遍历
void PreOrder(btnode<T> *x, Func Visit)
{
    if(x == 0) return;
    Visit(x); // 访问根
    PreOrder(x->left, Visit); // 遍历左子树
    PreOrder(x->right, Visit); // 遍历右子树
} // 耗时O(n)
```



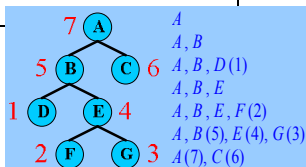
3. 二叉树的中根次序遍历

```
template<class T, class Func> // 中根遍历
void InOrder(btnode<T> *x, Func Visit)
{ if(x == 0) return;
  InOrder(x->left, Visit); // 遍历左子树
  Visit(x); // 访问根
  InOrder(x->right, Visit); // 遍历右子树
} // 耗时O(n)
```



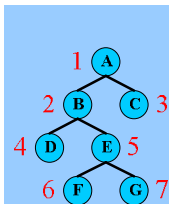
4. 二叉树的后根次序遍历

```
template<class T, class Func> // 后根遍历
void PostOrder(btnode<T> *x, Func Visit)
{ if(x == 0) return;
  PostOrder(x->left, Visit); // 遍历左子树
  PostOrder(x->right, Visit); // 遍历右子树
  Visit(x); // 访问根
} // 耗时O(n)
```



5. 二叉树的按层次遍历

```
template<class T, class Func> // 按层次遍历
void LevelOrder(btnode<T> *x, Func Visit)
{
    queue<btnode<T>*> Q;
    if(x != 0) Visit(x), Q.push(x); // 访问根并将根加入队列
    while(!Q.empty())
    {
        x = Q.front(), Q.pop();
        // 访问x的左右儿子并将左右儿子加入队列
        auto left = x->left, right = x->right;
        if(left != 0) Visit(left), Q.push(left);
        if(right != 0) Visit(right), Q.push(right);
    }
} // 耗时O(n)
```

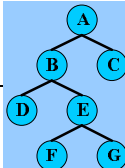


x	队列
	A
A	B, C
B	C, D, E
C	D, E
D	E
E	F, G
F	G
G	

6. 测试程序*

使用图4-12~4-15中使用的二叉树。

```
int main()
{
    auto Visit = [](bnode<char> *x) // 访问结点x
    {
        cout << x->data << ' ';
    };
    bnode<char> a{'A'}, b{'B'}, c{'C'}, d{'D'}, e{'E'}, f{'F'}, g{'G'};
    // 建立一棵以a为根的二叉树
    a.left = &b, a.right = &c, b.left = &d, b.right = &e,
        e.left = &f, e.right = &g;
    cout << "先根次序: ";
    PreOrder(&a, Visit), cout << endl; // A B D E F G C
    cout << "中根次序: ";
    InOrder(&a, Visit), cout << endl; // D B F E G A C
    cout << "后根次序: ";
    PostOrder(&a, Visit), cout << endl; // D F G E B C A
    cout << "按层次遍历: ";
    LevelOrder(&a, Visit), cout << endl; // A B C D E F G
}
```



$A.l=B, A.r=C$
 $B.l=D, B.r=E$
 $C.l=0, C.r=0$
 $D.l=0, D.r=0$
 $E.l=F, E.r=G$
 $F.l=0, F.r=0$
 $G.l=0, G.r=0$

4.4 堆的基本操作和实现方法*

4.4.1 堆的基本操作

1. 二叉树的顺序表示

用一个一维数组buf[n]来表示二叉树，其中n是允许存入二叉树中元素的最大数目。根元素存放在buf[0]中，第i号元素的左右儿子分别存放在buf[2*i+1]和buf[2*i+2]中。如图4-16所示。

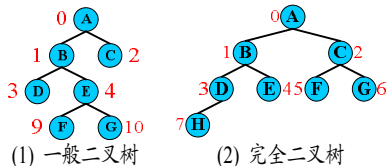


图4-16 二叉树的顺序表示

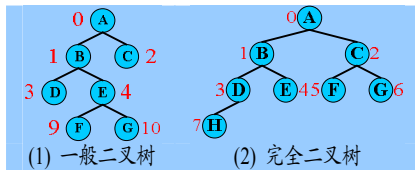
显然，如果二叉树中的元素在该数组中正好是从第0号位置开始连续存放的，则该二叉树是一棵完全二叉树。

2. 二叉树的顺序表示的优缺点

显然，二叉树的顺序表示可能会造成数组元素不能连续使用，有很多未用空间。例如，对于图4-16(1)所示的二叉树，第5, 6, 7, 8号位置都是未用空间。但是，这种表示方法能够直接访问每个元素的双亲和儿子，第 i 号元素的双亲位于 $\lfloor (i-1)/2 \rfloor$ 位置，儿子分别位于 $2i+1$ 和 $2i+2$ 位置。

3. 堆的含义

堆分为最小堆和最大堆2种。最小堆是一种元素能够比较大小的完全二叉树，其中每个元素均不大于它的儿子。最大堆是一种元素能够比较大小的完全二叉树，其中每个元素均不小于它的儿子。显然，在最小堆中，根元素是最小的元素，在最大堆中，根元素是最大的元素。



4. 插入元素

需要找到插入位置。以最小堆为例说明。如图4-17所示。从新加入元素开始，根据上层元素与待插入元素的比较结果，将较大的元素逐步下移，直到找到插入位置。

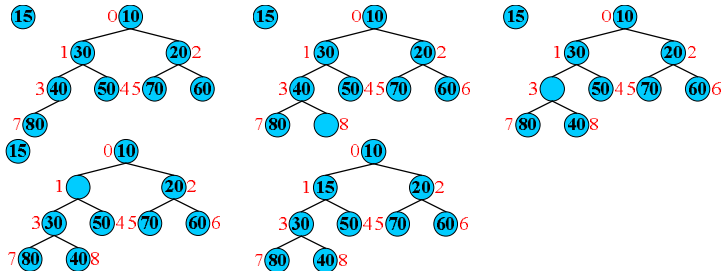


图4-17 在堆中插入元素

该过程耗时 $O(\log n)$ ， n 为插入后元素个数。这是因为有 n 个顶点的完全二叉树的深度不超过 $\log(2n)$ 。

5. 删除元素

只能删除根元素。以最小堆为例说明。如图4-18所示。从根元素开始，根据下层元素与原最后元素的比较结果，将较小的元素逐步上移，直到找到原最后元素的插入位置。

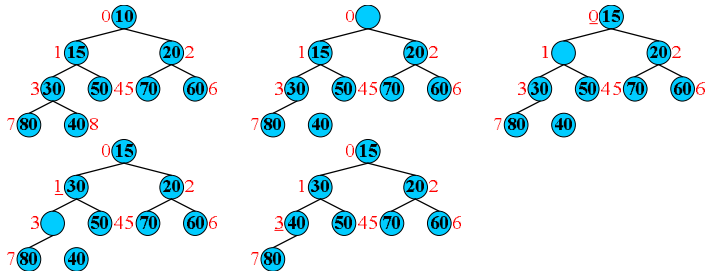


图4-18 删除堆的根元素

该过程耗时 $O(\log n)$ ， n 是删除前元素个数。这是因为有 n 个顶点的完全二叉树的深度不超过 $\log(2n)$ 。

4.4.2 堆的C++实现

以最小堆为例说明。

1. 预处理

```
// minheap.h  
#pragma once  
#include<algorithm> // copy_n, etc.  
using namespace std;  
using namespace rel_ops; // !=, <=, >, >=
```

2. 数据成员

```
template<class T> //  
class MinHeap  
{ T *buf = 0; // 缓冲区  
  int cap = 64, sz = 0; // 容量, 大小
```

3. 初始化

<code>public:</code>
<code>MinHeap &operator=(const MinHeap &) = delete; // 禁止复制</code>
<code>MinHeap() { // 空最小堆, 使用默认容量 buf = new T[cap]; // 创建数组 }</code>
<code>MinHeap(const MinHeap &X): cap(X.cap), sz(X.sz) { // 使用另一个最小堆初始化 buf = new T[cap]; // 创建数组 copy_n(X.buf, sz, buf); // 复制元素 } // 耗时O(sz)</code>
<code>~MinHeap() { delete[] buf; // 释放数组 }</code>

4. 访问元素

```
const T &top() const // 只读, 堆中元素不允许改写
{
    return buf[0];
}
```

5. 插入元素

```
void push(const T &v)
{
    reserve(sz + 1); // 扩大容量
    int i = sz; // 当前结点, 从新元素开始
    while(i > 0)
    {
        int up = (i - 1) / 2; // 上层结点
        if(buf[up] < v) break; // 找到插入位置
        buf[i] = buf[up], i = up; // 元素下移, 考虑上层结点
    }
    buf[i] = v, ++sz; // 插入
} // 耗时O(log(sz))
```

6. 删除元素

```
void pop()
{
    if(sz <= 0) return; // 空的
    const T &v = buf[sz - 1]; // 原最后位置元素
    int m = (sz - 1) / 2; // 原结束位置的双亲
    int i = 0; // 当前结点, 从根元素开始
    while(i < m)
    {
        int ch = 2 * i + 1, ch2 = ch + 1; // 左儿子, 右儿子
        if(buf[ch2] < buf[ch]) // 选取较小的儿子
            ch = ch2;
        if(v < buf[ch]) break; // 找到原最后元素的插入位置
        buf[i] = buf[ch], i = ch; // 元素上移, 考虑下层结点
    }
    buf[i] = v; // 插入原最后位置元素
    --sz; // 新大小
} // 耗时O(log(sz))
```

7. 大小

```
void reserve(int n) // 扩大容量, 新容量不小于n
{
    if(cap >= n) return; // 无需扩容
    T *buf0 = buf; // 原数组
    while(cap < n) cap *= 2; // 计算新容量
    buf = new T[cap]; // 创建新数组
    copy_n(buf0, sz, buf); // 复制原有元素
    delete []buf0; // 释放原数组
} // 耗时O(sz)

int size() const
{
    return sz;
}

bool empty() const
{
    return sz <= 0;
}

};
```

4.4.3 C++ STL中的堆

在C++ STL中，可用priority_queue表示堆。priority_queue使用与元素数目的对数成正比例的操作完成元素的插入和删除。但是C++ STL中的priority_queue使用起来不是很方便。1.5.6节对priority_queue进行了特殊化，专门定义了最小堆和最大堆，保存在文件queue.hpp中。1.5.6节介绍了文件queue.hpp的内容，并且举例说明了C++ STL中堆的使用方法。

4.5 一般树的遍历

4.5.1 一般树的遍历算法

可以将二叉树的父子关系推广到树上。每个父亲的儿子可以有多个，而且可以排出顺序。于是，关于二叉树的几种遍历算法完全可以移置到一般的树上来。

1. 树的先根次序遍历

- ① 若T为空，则返回。
- ② 访问T的根。
- ③ 按树的先根次序遍历T的第一棵子树。
- ④ 按树的先根次序依次遍历T的其余子树。

图4-19演示了先根次序遍历树的执行过程及结果。

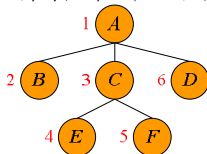


图4-19 树的先根次序遍历

2. 树的中根次序遍历

- ① 若T为空，则返回。
- ② 按树的中根次序遍历T的第一棵子树。
- ③ 访问T的根。
- ④ 按树的中根次序依次遍历T的其余子树。

图4-20演示了中根次序遍历树的执行过程及结果。

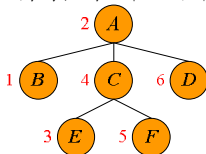


图4-20 树的中根次序遍历

3. 树的后根次序遍历

- ① 若T为空，则返回。
- ② 按树的后根次序遍历T的第一棵子树。
- ③ 按树的后根次序依次遍历T的其余子树。
- ④ 访问T的根。

图4-21演示了后根次序遍历树的执行过程及结果。

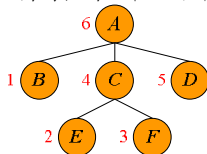


图4-21 树的后根次序遍历

4. 树的按层次遍历

- ① 若 T 非空，则访问 T 的根并将根加入队列。
- ② 若队列非空，取出队首元素 x 。
- ③ 访问 x 的所有儿子并将所有儿子加入队列。
- ④ 重复②~③，直到队列为空。

图4-22演示了按层次遍历树的执行过程及结果。

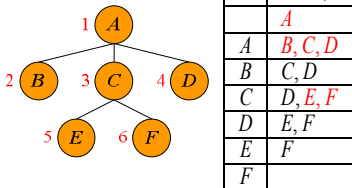
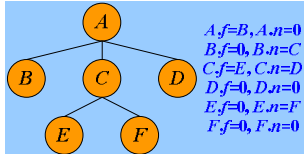


图 4-22 树的按层次遍历

4.5.2 树遍历算法的C++实现

1. 树的C++表示

使用孩子兄弟的表示方法。



```
#include "algorithm.h"
```

```
template<class T> // 树结点类
```

```
struct treenode
```

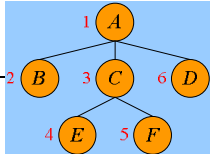
```
{ T data;
```

```
    treenode *first = 0, *next = 0; // 第一棵子树, 下一棵树(兄弟)
```

```
};
```

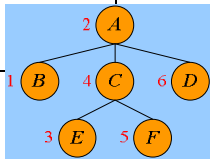
2. 树的先根次序遍历

```
template<class T, class Func> //  
void PreOrder(treenode<T> *x, Func Visit)  
{ if(x == 0) return;  
  Visit(x); // 访问根  
  PreOrder(x->first, Visit); // 遍历第一棵子树  
  for(x = x->first; x != 0; x = x->next)  
    PreOrder(x->next, Visit); // 遍历其余子树  
} // 耗时O(n)
```



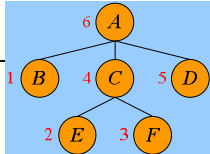
3. 树的中根次序遍历

```
template<class T, class Func> //  
void InOrder(treenode<T> *x, Func Visit)  
{ if(x == 0) return;  
  InOrder(x->first, Visit); // 遍历第一棵子树  
  Visit(x); // 访问根  
  for(x = x->first; x != 0; x = x->next)  
    InOrder(x->next, Visit); // 遍历其余子树  
} // 耗时O(n)
```



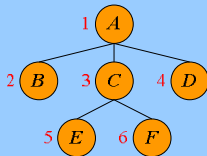
4. 树的后根次序遍历

```
template<class T, class Func> //  
void PostOrder(treenode<T> *x, Func Visit)  
{ if(x == 0) return;  
  auto t = x->first; // 第一棵子树  
  PostOrder(t, Visit); // 遍历第一棵子树  
  for(; t != 0; t = t->next)  
    PostOrder(t->next, Visit); // 遍历其余子树  
  Visit(x); // 访问根  
} // 耗时O(n)
```



5. 树的按层次遍历

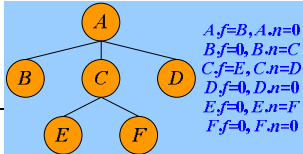
```
template<class T, class Func> //
void LevelOrder(treenode<T> *x, Func Visit)
{
    queue<treenode<T>*> Q;
    if(x != 0) Visit(x), Q.push(x); // 访问根并将根加入队列
    while(!Q.empty())
    {
        x = Q.front(), Q.pop();
        // 访问x的所有儿子并将所有儿子加入队列
        for(x = x->first; x != 0; x = x->next)
            Visit(x), Q.push(x);
    }
} // 耗时O(n)
```



x	队列
	A
A	B, C, D
B	C, D
C	D, E, F
D	E, F
E	F
F	

5. 测试程序*

使用图4-19~4-22中使用的树。



```
int main()
{
    auto Visit = [](treenode<char> *x) // 访问结点*x
    {
        cout << x->data << ' ';
    };
    treenode<char> a{'A'}, b{'B'}, c{'C'}, d{'D'}, e{'E'}, f{'F'};
    // 建立一棵以a为根的树
    a.first = &b, b.next = &c, c.first = &e, c.next = &d, e.next = &f;
    cout << "先根次序: ";
    PreOrder(&a, Visit), cout << endl; // A B C E F D
    cout << "中根次序: ";
    InOrder(&a, Visit), cout << endl; // B A E C F D
    cout << "后根次序: ";
    PostOrder(&a, Visit), cout << endl; // B E F C D A
    cout << "按层次遍历: ";
    LevelOrder(&a, Visit), cout << endl; // A B C D E F
}
```


4.6 图的宽度优先遍历

无论是二叉树还是一般树，由于不含圈，所以各个子树之间是相互独立的。因此遍历过程是对各个子树的分别遍历和对根遍历以及把这些遍历有机地组合起来。一般的图没有这种独立性，所以上述方法不能直接推广到一般的图。但是，上述方法的思想可以借鉴，于是产生了深度优先搜索方法和宽度优先搜索方法。

本节介绍宽度优先搜索，下一节介绍深度优先搜索方法。

4.6.1 连通图的宽度优先搜索 (BFS)

1. 算法思路

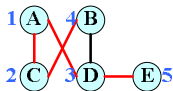
准备：起点 v 和一个空队列。

① 将 v 打上已访问标记，并将 v 放入队列。

② 取出队列的队首元素 u ，搜索所有与 u 相邻的顶点。如果 w 与 u 相邻且未访问，则将 w 打上已访问标记，并将 w 放入队列。

③ 重复②，直到队列为空。

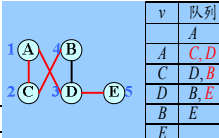
图4-23演示了BFS的执行过程及结果。



v	队列
	A
A	C, D
C	D, B
D	B, E
B	E
E	

图4-23 BFS的执行过程

2. 算法描述



```
#include "algorithm.h"
```

```
template<class Fun> // 搜索一个连通分支
```

```
void BFS(Matrix<bool> &G, int v, vector<bool> &Visited, Fun Visit)
```

```
{ // 邻接矩阵, 开始顶点, 访问标记(已经初始化为0), 访问函数
```

```
    int n = G.Rows();
```

```
    queue<int> Q; // 创建空队列
```

```
    Visit(v), Visited[v] = 1, Q.push(v); // 访问v并加入队列
```

```
    while(!Q.empty())
```

```
    { v = Q.front(), Q.pop(); // 取出队首元素v
```

```
        for(int w = 0; w < n; ++w)
```

```
            if(!Visited[w] && G[v][w] == 1) // w与v相邻且未访问
```

```
                Visit(w), Visited[w] = 1, Q.push(w); // 访问w, 入队
```

```
        }
```

```
    }
```

3. 复杂性分析

设连通图 G 有 n 个顶点, 用邻接矩阵表示。则BFS在图 G 上的时间复杂性 $t(n)$ 和空间复杂性 $s(n)$ 满足 $t(n) = O(n^2)$, $s(n) = O(n)$ 。

4. 宽度优先生成树

如果 G 是连通图，则 G 有生成树。注意到BFS算法中，while循环将所有与 u 相邻的未访问顶点 w 添加到队列中。如果在添加 w 的同时将边 (u, w) 收集起来，那么在算法结束时，所有这些边形成了图 G 的一棵生成树，称为图 G 的宽度优先生成树。如图4-24所示。

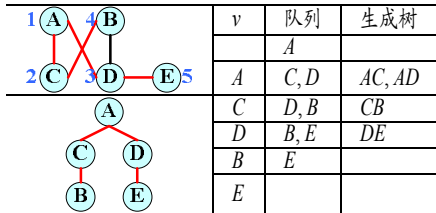


图4-24 宽度优先生成树

4.6.2 一般图的宽度优先遍历 (BFT)

对于非连通图，可以通过在每个连通分支中选取一个起点，应用宽度优先搜索遍历该图的所有顶点。如图4-25所示。

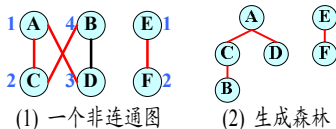


图4-25 一般图的宽度优先遍历

算法描述如下。

```
template<class Fun> // 一般图的宽度优先搜索
void BFT(Matrix<bool> &G, Fun Visit) // 邻接矩阵, 访问函数
{
    int n = G.Rows();
    vector<bool> Visited(n, 0); // 访问标记, 初始化为0
    for(int v = 0; v < n; ++v)
        if(!Visited[v]) // 一个新连通分支
            BFS(G, v, Visited, Visit), cout << endl;
} // 耗时O(n^2)
```

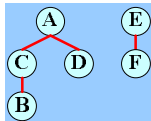
4.6.3 测试程序*

使用图4-25中的图。

```
void Visit(int v) // 访问函数
{
    cout << char('A' + v) << ' ';
}

int main()
{
    Matrix<bool> G =
    {
        {0, 0, 1, 1, 0, 0},
        {0, 0, 1, 1, 0, 0},
        {1, 1, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 1, 0}
    };
    BFT(G, Visit); // 一般图的宽度优先搜索
    cout << endl;
}
```

A C D B
E F



4.7 图的深度优先遍历

4.7.1 连通图的深度优先搜索 (DFS)

1. 算法思路

如图4-26所示。从顶点 v 出发, 将 v 打上已访问标记。以 v 作为当前扩展顶点。先访问 v 的第一个未访问的相邻顶点, 将这个顶点打上已访问标记, 并取该顶点作为当前扩展顶点。如果当前扩展顶点 u 没有未访问的相邻顶点, 则以前任扩展顶点为当前扩展顶点。这个过程一直持续到没有未访问顶点为止。

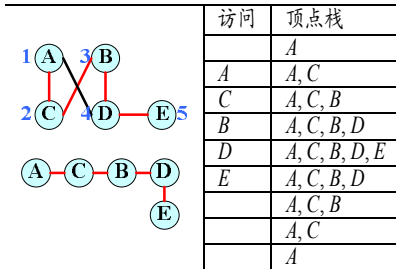


图4-26 连通图的深度优先遍历

2. 算法描述

```
#include "algorithm.h"
template<class Fun> // 搜索一个连通分支
void DFS(Matrix<bool> &G, int v, vector<bool> &Visited, Fun Visit)
{ // 邻接矩阵, 开始顶点, 访问标记(已经初始化为0), 访问函数
    int n = G.Rows();
    Visit(v), Visited[v] = 1; // 访问v
    for(int w = 0; w < n; ++w)
        if(Visited[w] == 0 && G[v][w] == 1) // w与v相邻且未访问
            DFS(G, w, Visited, Visit); // 访问w
}
```

3. 复杂性

设连通图 G 有 n 个顶点, 用邻接矩阵表示。则DFS在图 G 上的时间复杂性 $t(n)$ 满足 $t(n) = O(n^2)$ 。

4. 深度优先生成树

同样，如果在访问 w 的同时将边 (u, w) 收集起来，那么在算法结束时，所有这些边形成了连通图 G 的一棵生成树，称为图 G 的深度优先生成树。如图4-27所示。

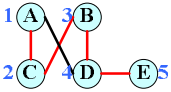
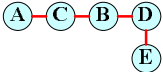
	访问	顶点栈	生成树
		A	
	A	A, C	AC
	C	A, C, B	CB
	B	A, C, B, D	BD
	D	A, C, B, D, E	DE
	E	A, C, B, D	
		A, C, B	
		A, C	
		A	

图4-27 深度优先生成树

4.7.2 一般图的深度优先遍历 (DFT)

对于非连通图，可以通过在每个连通分支中选取一个起点，应用深度优先搜索遍历该图的所有顶点。如图4-28所示。

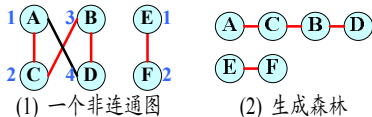


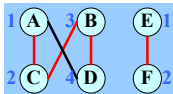
图4-28 一般图的深度优先遍历

算法描述如下。

```
template<class Fun> // 一般图的深度优先搜索
void DFT(Matrix<bool> &G, Fun Visit) // 邻接矩阵, 访问函数
{
    int n = G.Rows();
    vector<bool> Visited(n, 0); // 访问标记, 初始化为0
    for(int v = 0; v < n; ++v)
        if(!Visited[v]) // 一个新连通分支
            DFS(G, v, Visited, Visit), cout << endl;
} // 耗时O(n^2)
```

4.7.3 测试程序*

使用图4-28中的图。



```
void Visit(int v) // 访问函数
```

```
{ cout << char('A' + v) << ' ';  
}
```

```
int main()
```

```
{ Matrix<bool> G =
```

```
{ {0, 0, 1, 1, 0, 0},
```

```
{0, 0, 1, 1, 0, 0},
```

```
{1, 1, 0, 0, 0, 0},
```

```
{1, 1, 0, 0, 0, 0},
```

```
{0, 0, 0, 0, 0, 1},
```

```
{0, 0, 0, 0, 1, 0}
```

```
};
```

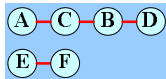
```
DFT(G, Visit); // 一般图的深度优先搜索
```

```
cout << endl;
```

```
}
```

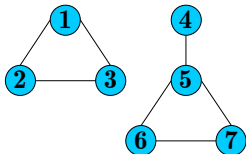
```
A C B D
```

```
E F
```

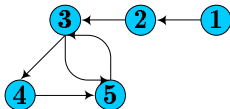


4.8 练习题

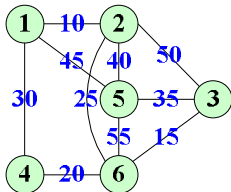
1、请写出下列无向图的邻接矩阵。



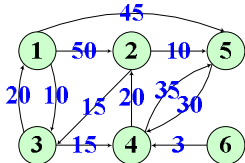
2、请写出下列有向图的邻接矩阵。



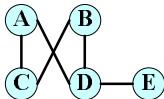
3、请写出下列无向加权图的邻接矩阵。



4、请写出下列有向加权图的邻接矩阵。



5、使用宽度优先搜索方法遍历下列无向图，请写出搜索过程中访问队列中的元素。



6、假设用一个 $n \times n$ 的数组来描述一个有向图的邻接矩阵，完成下列工作。

(1) 编写一个函数以确定某个顶点的出度，函数的复杂性应为 $\Theta(n)$ 。

(2) 编写一个函数以确定图中边的数目，函数的复杂性因为 $\Theta(n^2)$ 。

(3) 编写一个删除边 (i, j) 的函数，并确定代码的复杂性。

7、请使用伪代码或一种程序设计语言描述二叉树的4种遍历方法。

8、请使用伪代码或一种程序设计语言描述一般树的4种遍历方法（树使用儿子兄弟的方法表示）。

9、请使用伪代码或一种程序设计语言描述连通图的宽度优先搜索算法（图使用邻接矩阵表示）。

10、请使用伪代码或一种程序设计语言描述连通图的深度优先搜索算法（图使用邻接矩阵表示）。