# 51赫夫曼树

## 赫夫曼树

* 在数据膨胀、信息爆炸的今天，数据压缩的意义不言而喻。谈到数据压缩，就不能不提赫夫曼（Huffman）编码，赫夫曼编码是首个实用的压缩编码方案，即使在今天的许多知名压缩算法里，依然可以见到赫夫曼编码的影子。
* 另外，在数据通信中，用二进制给每个字符进行编码时不得不面对的一个问题是如何使电文总长最短且不产生二义性。根据字符出现频率，利用赫夫曼编码可以构造出一种不等长的二进制，使编码后的电文长度最短，且保证不产生二义性。
* 介绍赫夫曼编码前，我们必须得介绍赫夫曼树。
* 什么叫做赫夫曼树呢？我们先来看一个例子。
* 以下程序在效率上有什么问题呢？

if( a < 60 )

printf(“不及格”);

else if( a < 70 )

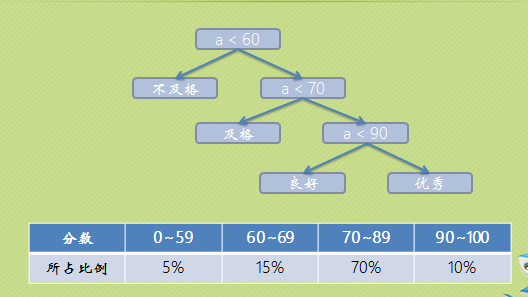
printf(“及格”);

else if( a < 90 )

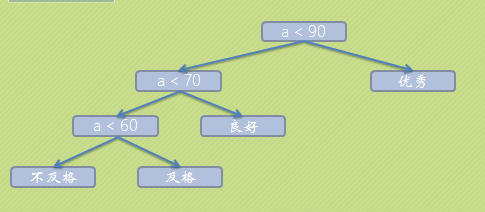
printf(“良好”);

else

printf(“优秀”);

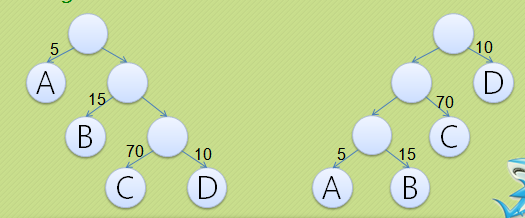


* 如果我们把判断流程改为以下，效果可能有明显的改善：



赫夫曼树定义与原理

* 我们先把这两棵二叉树简化成叶子结点带权的二叉树（注：树结点间的连线相关的数叫做权，Weight）。



* 结点的路径长度：
  + 从根结点到该结点的路径上的连接数。
* 树的路径长度：
  + 树中每个叶子结点的路径长度之和。
* 结点带权路径长度：
  + 结点的路径长度与结点权值的乘积。
* 树的带权路径长度：
  + WPL(Weighted Path Length)是树中所有叶子结点的带权路径长度之和。
* WPL的值越小，说明构造出来的二叉树性能越优。

WPL的值最小的二叉树称为最优二叉树，也叫做最优赫夫曼树。

那么现在的问题是，如何构造出最优的赫夫曼树呢？别急，赫夫曼大叔给了我们解决的方案。

* 看动画。。。。。。

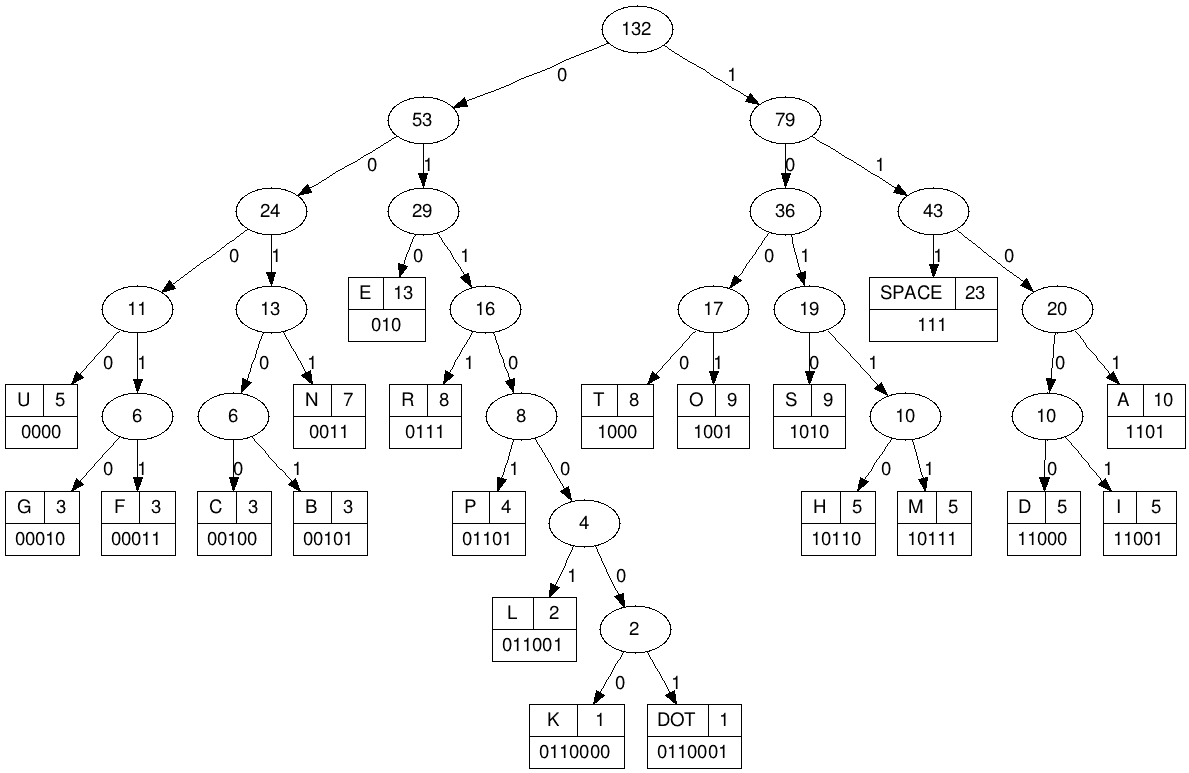


52赫夫曼编码

赫夫曼编码

* 上一节课我们已经谈了赫夫曼树的基本原理和构造方式，而赫夫曼编码可以很有效地压缩数据（通常可以节省20%~90%的空间，具体压缩率依赖于数据的特性）。
* 名词解释：定长编码，变长编码，前缀码
  + 定长编码：像ASCII编码
  + 变长编码：单个编码的长度不一致，可以根据整体出现频率来调节
  + 前缀码：所谓的前缀码，就是没有任何码字是其他码字的前缀

Huffman 编码设计图之一：



* 在教大家来编写赫夫曼编码实现代码之前，小甲鱼希望通过帮大家理清思路之后，大家先自己尝试写代码再看下一讲哦~
  + build a priority queue #字符出现次数
  + build a huffmanTree
  + build a huffmanTable #存放huffman编码
  + encode
  + decode
* 接下来的精髓内容还是让小甲鱼口语发挥吧，写成文字太限制了。。。。。。

# 53赫夫曼编码C语言实现

Huffman.cpp

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include "huffman.h"

#include "pQueue.h"

void traverseTree(htNode \*treeNode, hlTable \*\*table, int k, char code[256])

{

//If we reach the end we introduce the code in the table

if(treeNode->left == NULL && treeNode->right == NULL)

{

code[k] = '\0';

hlNode \*aux = (hlNode \*)malloc(sizeof(hlNode));

aux->code = (char \*)malloc(sizeof(char)\*(strlen(code)+1));

strcpy(aux->code,code);

aux->symbol = treeNode->symbol;

aux->next = NULL;

if((\*table)->first == NULL)

{

(\*table)->first = aux;

(\*table)->last = aux;

}

else

{

(\*table)->last->next = aux;

(\*table)->last = aux;

}

}

//We concatenate a 0 for each step to the left

if(treeNode->left!=NULL)

{

code[k]='0';

traverseTree(treeNode->left,table,k+1,code);

}

//We concatenate a 1 for each step to the right

if(treeNode->right!=NULL)

{

code[k]='1';

traverseTree(treeNode->right,table,k+1,code);

}

}

hlTable \* buildTable(htTree \* huffmanTree)

{

//We initialize the table

hlTable \*table = (hlTable \*)malloc(sizeof(hlTable));

table->first = NULL;

table->last = NULL;

//Auxiliary variables

char code[256];

//k will memories the level on which the traversal is

int k=0;

//We traverse the tree and calculate the codes

traverseTree(huffmanTree->root,&table,k,code);

return table;

}

htTree \* buildTree(char \*inputString)

{

//The array in which we calculate the frequency of the symbols

//Knowing that there are only 256 posibilities of combining 8 bits

//(256 ASCII characters)

int \* probability = (int \*)malloc(sizeof(int)\*256);

//We initialize the array

for(int i=0; i<256; i++)

probability[i]=0;

//We consider the symbol as an array index and we calculate how many times each symbol appears

for(int i=0; inputString[i]!='\0'; i++)

probability[(unsigned char) inputString[i]]++;

//The queue which will hold the tree nodes

pQueue \* huffmanQueue;

initPQueue(&huffmanQueue);

//We create nodes for each symbol in the string

for(int i=0; i<256; i++)

if(probability[i]!=0)

{

htNode \*aux = (htNode \*)malloc(sizeof(htNode));

aux->left = NULL;

aux->right = NULL;

aux->symbol = (char) i;

addPQueue(&huffmanQueue,aux,probability[i]);

}

//We free the array because we don't need it anymore

free(probability);

//We apply the steps described in the article to build the tree

while(huffmanQueue->size!=1)

{

int priority = huffmanQueue->first->priority;

priority+=huffmanQueue->first->next->priority;

htNode \*left = getPQueue(&huffmanQueue);

htNode \*right = getPQueue(&huffmanQueue);

htNode \*newNode = (htNode \*)malloc(sizeof(htNode));

newNode->left = left;

newNode->right = right;

addPQueue(&huffmanQueue,newNode,priority);

}

//We create the tree

htTree \*tree = (htTree \*) malloc(sizeof(htTree));

tree->root = getPQueue(&huffmanQueue);

return tree;

}

void encode(hlTable \*table, char \*stringToEncode)

{

hlNode \*traversal;

printf("\nEncoding\nInput string : %s\nEncoded string : \n",stringToEncode);

//For each element of the string traverse the table

//and once we find the symbol we output the code for it

for(int i=0; stringToEncode[i]!='\0'; i++)

{

traversal = table->first;

while(traversal->symbol != stringToEncode[i])

traversal = traversal->next;

printf("%s",traversal->code);

}

printf("\n");

}

void decode(htTree \*tree, char \*stringToDecode)

{

htNode \*traversal = tree->root;

printf("\nDecoding\nInput string : %s\nDecoded string : \n",stringToDecode);

//For each "bit" of the string to decode

//we take a step to the left for 0

//or ont to the right for 1

for(int i=0; stringToDecode[i]!='\0'; i++)

{

if(traversal->left == NULL && traversal->right == NULL)

{

printf("%c",traversal->symbol);

traversal = tree->root;

}

if(stringToDecode[i] == '0')

traversal = traversal->left;

if(stringToDecode[i] == '1')

traversal = traversal->right;

if(stringToDecode[i]!='0'&&stringToDecode[i]!='1')

{

printf("The input string is not coded correctly!\n");

return;

}

}

if(traversal->left == NULL && traversal->right == NULL)

{

printf("%c",traversal->symbol);

traversal = tree->root;

}

printf("\n");

}

Huffman.h

#pragma once

#ifndef \_HUFFMAN\_H

#define \_HUFFMAN\_H

//The Huffman tree node definition

typedef struct \_htNode {

char symbol;

struct \_htNode \*left, \*right;

}htNode;

/\*

We "encapsulate" the entire tree in a structure

because in the future we might add fields like "size"

if we need to. This way we don't have to modify the entire

source code.

\*/

typedef struct \_htTree {

htNode \*root;

}htTree;

//The Huffman table nodes (linked list implementation)

typedef struct \_hlNode {

char symbol;

char \*code;

struct \_hlNode \*next;

}hlNode;

//Incapsularea listei

typedef struct \_hlTable {

hlNode \*first;

hlNode \*last;

}hlTable;

htTree \* buildTree(char \*inputString);

hlTable \* buildTable(htTree \*huffmanTree);

void encode(hlTable \*table, char \*stringToEncode);

void decode(htTree \*tree, char \*stringToDecode);

#endif

pQueue.cpp

#include "pQueue.h"

#include <stdlib.h>

#include <stdio.h>

void initPQueue(pQueue \*\*queue)

{

//We allocate memory for the priority queue type

//and we initialize the values of the fields

(\*queue) = (pQueue \*) malloc(sizeof(pQueue));

(\*queue)->first = NULL;

(\*queue)->size = 0;

return;

}

void addPQueue(pQueue \*\*queue, TYPE val, unsigned int priority)

{

//If the queue is full we don't have to add the specified value.

//We output an error message to the console and return.

if((\*queue)->size == MAX\_SZ)

{

printf("\nQueue is full.\n");

return;

}

pQueueNode \*aux = (pQueueNode \*)malloc(sizeof(pQueueNode));

aux->priority = priority;

aux->val = val;

//If the queue is empty we add the first value.

//We validate twice in case the structure was modified abnormally at runtime (rarely happens).

if((\*queue)->size == 0 || (\*queue)->first == NULL)

{

aux->next = NULL;

(\*queue)->first = aux;

(\*queue)->size = 1;

return;

}

else

{

//If there are already elements in the queue and the priority of the element

//that we want to add is greater than the priority of the first element,

//we'll add it in front of the first element.

//Be careful, here we need the priorities to be in descending order

if(priority<=(\*queue)->first->priority)

{

aux->next = (\*queue)->first;

(\*queue)->first = aux;

(\*queue)->size++;

return;

}

else

{

//We're looking for a place to fit the element depending on it's priority

pQueueNode \* iterator = (\*queue)->first;

while(iterator->next!=NULL)

{

//Same as before, descending, we place the element at the begining of the

//sequence with the same priority for efficiency even if

//it defeats the idea of a queue.

if(priority<=iterator->next->priority)

{

aux->next = iterator->next;

iterator->next = aux;

(\*queue)->size++;

return;

}

iterator = iterator->next;

}

//If we reached the end and we haven't added the element,

//we'll add it at the end of the queue.

if(iterator->next == NULL)

{

aux->next = NULL;

iterator->next = aux;

(\*queue)->size++;

return;

}

}

}

}

TYPE getPQueue(pQueue \*\*queue)

{

TYPE returnValue;

//We get elements from the queue as long as it isn't empty

if((\*queue)->size>0)

{

returnValue = (\*queue)->first->val;

(\*queue)->first = (\*queue)->first->next;

(\*queue)->size--;

}

else

{

//If the queue is empty we show an error message.

//The function will return whatever is in the memory at that time as returnValue.

//Or you can define an error value depeding on what you choose to store in the queue.

printf("\nQueue is empty.\n");

}

return returnValue;

}

pQueue.h

#pragma once

#ifndef \_PQUEUE\_H

#define \_PQUEUE\_H

#include "huffman.h"

//We modify the data type to hold pointers to Huffman tree nodes

#define TYPE htNode \*

#define MAX\_SZ 256

typedef struct \_pQueueNode {

TYPE val;

unsigned int priority;

struct \_pQueueNode \*next;

}pQueueNode;

typedef struct \_pQueue {

unsigned int size;

pQueueNode \*first;

}pQueue;

void initPQueue(pQueue \*\*queue);

void addPQueue(pQueue \*\*queue, TYPE val, unsigned int priority);

TYPE getPQueue(pQueue \*\*queue);

#endif

main.cpp

#include <stdio.h>

#include <stdlib.h>

#include "huffman.h"

int main(void)

{

//We build the tree depending on the string

htTree \*codeTree = buildTree("beep boop beer!");

//We build the table depending on the Huffman tree

hlTable \*codeTable = buildTable(codeTree);

//We encode using the Huffman table

encode(codeTable,"beep boop beer!");

//We decode using the Huffman tree

//We can decode string that only use symbols from the initial string

decode(codeTree,"0011111000111");

//Output : 0011 1110 1011 0001 0010 1010 1100 1111 1000 1001

return 0;

}

？？？？？？？？？？？？？？？？//=============20170924

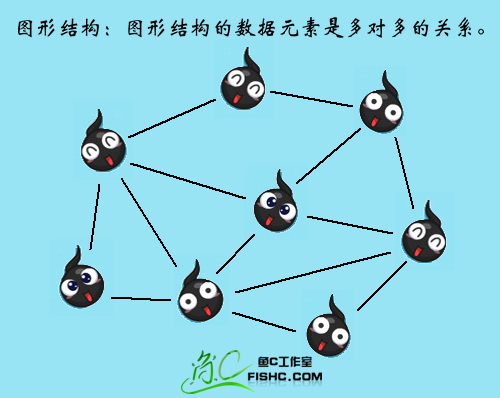
# 54图

## 图

* 在前边讲解的线性表中，每个元素之间只有一个直接前驱和一个直接后继，在树形结构中，数据元素之间是层次关系，并且每一层上的数据元素可能和下一层中多个元素相关，但只能和上一层中一个元素相关。
* 但这仅仅都只是一对一，一对多的简单模型，如果要研究如人与人之间关系就非常复杂了。
* 万恶图为首，前边可能有些童鞋会感觉树的术语好多，可来到了图这章节，你才知道什么叫做真正的术语多！

## 图的定义

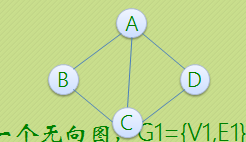
* 图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G(V,E)，其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合。



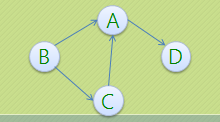
* 对于图的定义，我们需要明确几个注意的地方：
  + 线性表中我们把数据元素叫元素，树中叫结点，在图中数据元素我们则称之为顶点(Vertex)。
  + 线性表可以没有数据元素，称为空表，树中可以没有结点，叫做空树，而图结构在咱国内大部分的教材中强调顶点集合V要有穷非空。
  + 线性表中，相邻的数据元素之间具有线性关系，树结构中，相邻两层的结点具有层次关系，而图结构中，任意两个顶点之间都可能有关系，顶点之间的逻辑关系用边来表示，边集可以是空的。

## 图的各种奇葩定义

* 无向边：若顶点Vi到Vj之间的边没有方向，则称这条边为无向边(Edge)，用无序偶(Vi,Vj)来表示。



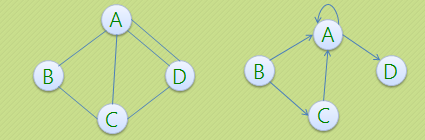
* 上图G1是一个无向图，G1={V1,E1}，其中
  + V1={A,B,C,D}，
  + E1={(A,B),(B,C),(C,D),(D,A),(A,C)}



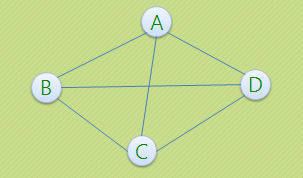
* 有向边：若从顶点Vi到Vj的边有方向，则称这条边为有向边，也成为弧(Arc)，用有序偶<Vi,Vj>来表示，Vi称为弧尾，Vj称为弧头。
* 上图G2是一个无向图，G2={V2,E2}，其中
  + V2={A,B,C,D}，

E2={<B,A>,<B,C>,<C,A>,<A,D>}

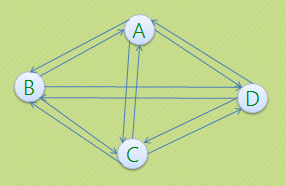
* 简单图：在图结构中，若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。
* 以下两个则不属于简单图：



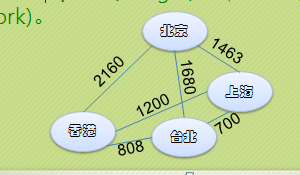
* 无向完全图：在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有n个顶点的无向完全图有n\*(n-1)/2（me：排列组合Cn2）条边。



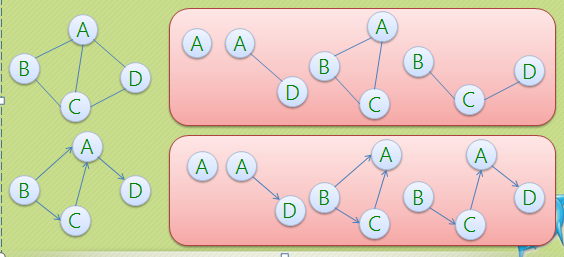
* 有向完全图：在有向图中，如果任意两个顶点之间都存在方向互为相反的两条弧，则称该图为有向完全图。含有n个顶点的有向完全图有n\*(n-1)条边。



* 稀疏图和稠密图：这里的稀疏和稠密是模糊的概念，都是相对而言的，通常认为边或弧数小于n\*logn（n是顶点的个数）的图称为稀疏图，反之称为稠密图。
* 有些图的边或弧带有与它相关的数字，这种与图的边或弧相关的数叫做权(Weight)，带权的图通常称为网(Network)。



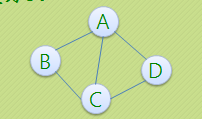
* 假设有两个图G1=(V1,E1)和G2=(V2,E2)，如果V2⊆V1，E2⊆E1，则称G2为G1的子图(Subgraph)。



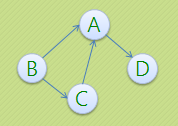
# 55图的定义与术语2

## 图的顶点与边之间的关系

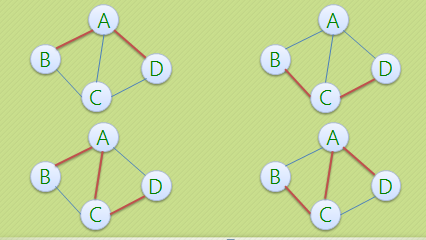
* 对于无向图G=(V,E)，如果边(V1,V2)∈E，则称顶点V1和V2互为邻接点(Adjacent)，即V1和V2相邻接。边(V1,V2)依附(incident)于顶点V1和V2，或者说边(V1,V2)与顶点V1和V2相关联。
* 顶点V的度(Degree)是和V相关联的边的数目，记为TD(V)，如下图，顶点A与B互为邻接点，边(A,B)依附于顶点A与B上，顶点A的度为3。



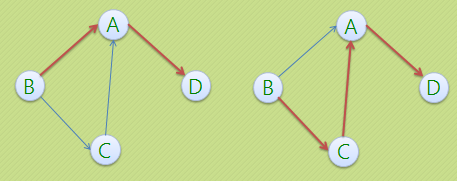
* 对于有向图G=(V,E)，如果有<V1,V2>∈E，则称顶点V1邻接到顶点V2，顶点V2邻接自顶点V1。
* 以顶点V为头的弧的数目称为V的入度(InDegree)，记为ID(V)，以V为尾的弧的数目称为V的出度(OutDegree)，记为OD(V)，因此顶点V的度为TD(V)=ID(V)+OD(V)。
* 下图顶点A的入度是2，出度是1，所以顶点A的度是3。



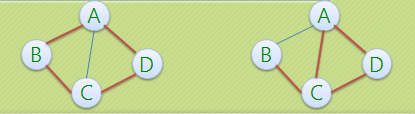
* 无向图G=(V,E)中从顶点V1到顶点V2的路径(Path)。
* 下图用红线列举了从顶点B到顶点D的四种不同路径：



* 如果G是有向图，则路径也是有向的。
* 下图用红线列举顶点B到顶点D的两种路径，而顶点A到顶点B就不存在路径啦：

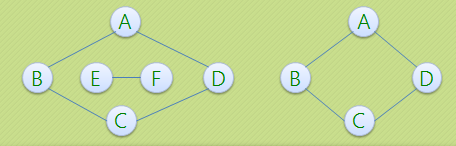


* 路径的长度是路径上的边或弧的数目。
* 第一个顶点到最后一个顶点相同的路径称为回路或环(Cycle)。
* 序列中顶点不重复出现的路径称为简单路径，除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。
* 下图左侧是简单环，右侧不是简单环：

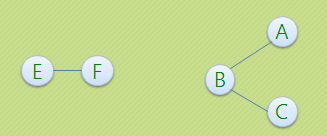


连通图

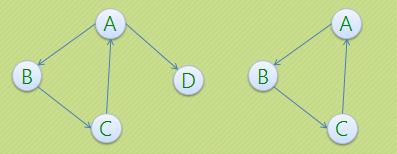
* 在无向图G中，如果从顶点V1到顶点V2有路径，则称V1和V2是连通的，如果对于图中任意两个顶点Vi和Vj都是连通的，则称G是连通图(ConnectedGraph)
* 下图左侧不是连通图，右侧是连通图：



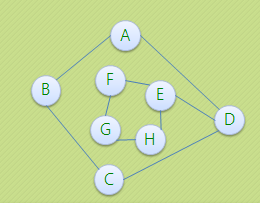
* 无向图中的极大连通子图称为连通分量。
* 注意以下概念：
  + 首先要是子图，并且子图是要连通的；
  + 连通子图含有极大顶点数；
  + 具有极大顶点数的连通子图包含依附于这些顶点的所有边。



* 在有向图G中，如果对于每一对Vi到Vj都存在路径，则称G是强连通图。
* 有向图中的极大强连通子图称为有向图的强连通分量。
* 下图左侧并不是强连通图，右侧是。并且右侧是左侧的极大强连通子图，也是左侧的强连通分量。



* 最后我们再来看连通图的生成树定义。
* 所谓的一个连通图的生成树是一个极小的连通子图，它含有图中全部的n个顶点，但只有足以构成一棵树的n-1条边。



* 如果一个有向图恰有一个顶点入度为0，其余顶点的入度均为1，则是一棵有向树。

