CS211FZ Note 5-1 | Algorithms & Data Structures | DSA2

Key Points 5-1 (Graph)

- Graphs intro
- DFS & BFS

L6 Graphs

为什么要有图

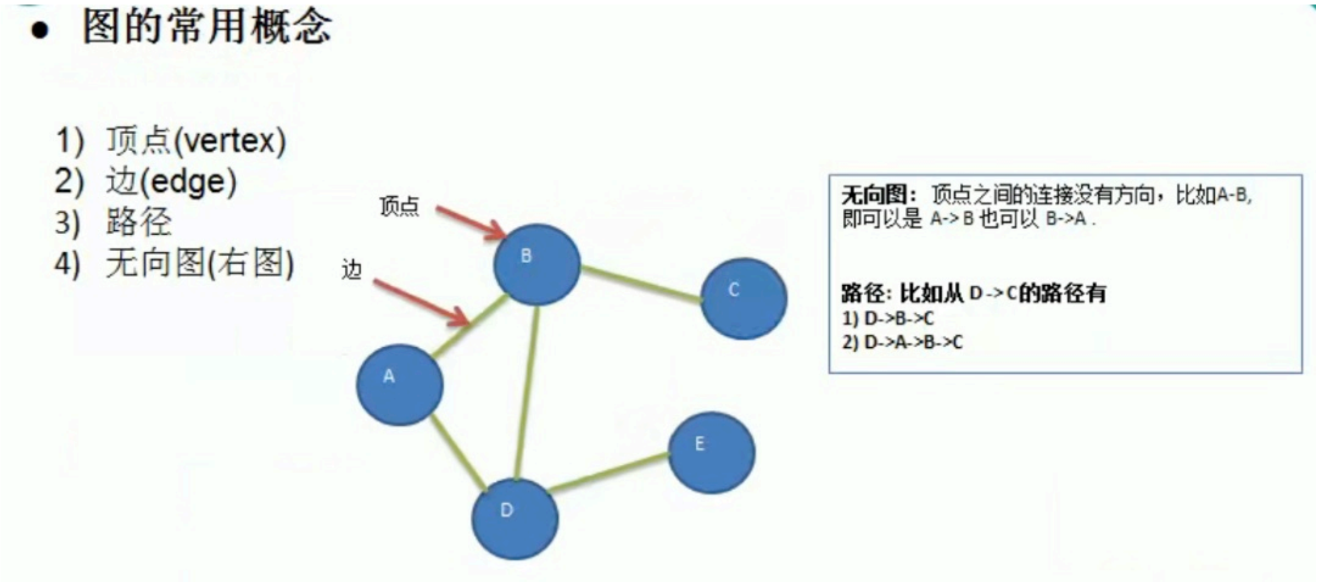
1) 前面我们学了线性表和树
2) 线性表局限于一个直接前驱和一个直接后继的关系
3) 树也只能有一个直接前驱也就是父节点
4) 当我们需要表示多对多的关系时，这里我们就用到了图

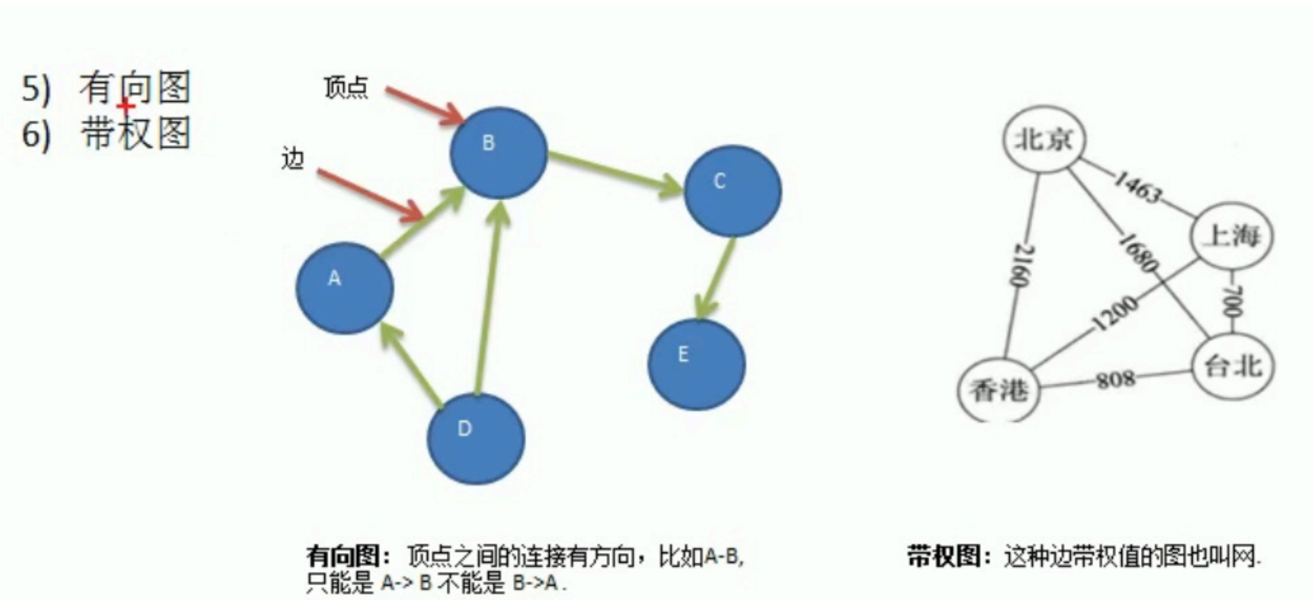
> 图：一种数据结构，其中*结点可以是具有零个或多个相邻元素，两个结点之间的连接称为边*，结点也称为顶点。

6.1 图概念

- 顶点(vertex)
- 边(edge)
- 路径(path)
- 无向图(undirected graph)
- 有向图（directed graph）
- 带权图（网，weighted graph）

- There are two types of graphs:
  - Unweighted graphs where the links between nodes are equal.
  - Weighted graphs where the links are each associated with an individual weight.





Directed and Weighted（方向&&权重）

- A non-directed graph means you don't have to go in a particular direction. You can follow an edge in both directions.
- Graphs are often used to model situations where you can go in only one direction along an edge – like a one-way street.
- These graphs are called *directed,* and the allowed direction is shown as an arrowhead.
- In some graphs, edges are given a weight that represents factors such as the physical distance between two vertices or the cost/time taken to get from one vertex to another.

Vertices and Edges（结点&&边）

Vertices and Edges

A **graph** is a collection of **vertices** and **edges**.

- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

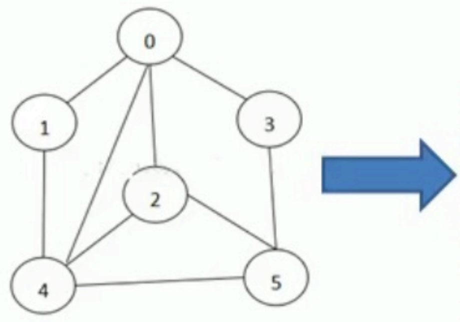
6.2 图的表示方式

1. 二维数组（邻接矩阵）Adjacent matrix

图的表示方式有两种：二维数组表示（邻接矩阵）；链表表示（邻接表）。

邻接矩阵

邻接矩阵是表示图形中顶点之间相邻关系的矩阵，对于n个顶点的图而言，矩阵是的row和col表示的是1....n个点。



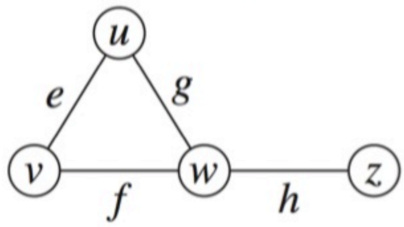
$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 1 & 1 \\
3 & 1 & 0 & 0 & 0 & 0 & 1 \\
4 & 1 & 1 & 1 & 0 & 0 & 1 \\
5 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

Adjacency Matrix Structure

The adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices.

If a graph has N vertices, then the adjacency matrix is an N x N array.

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



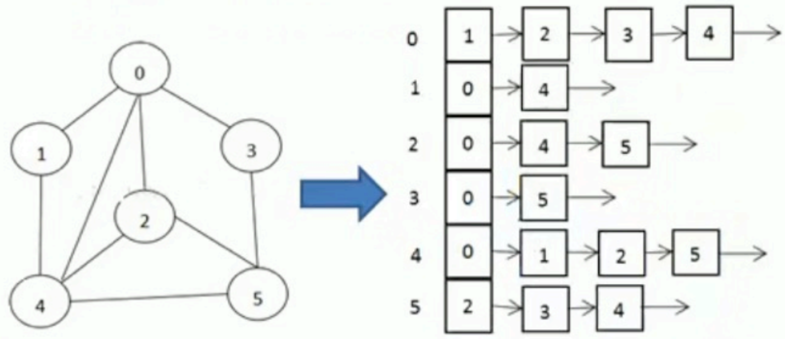
		0	1	2	3
$u \rightarrow$	0		e	g	
$v \rightarrow$	1	e		f	
$w \rightarrow$	2	g	f		h
$z \rightarrow$	3			h	

存在空间浪费

2. 链表（邻接表）Adjacency List

邻接表

1) 邻接矩阵需要为每个顶点都分配n个边的空间，其实有很多边都是不存在,会造成空间的一定损失.
2) 邻接表的实现只关心存在的边，不关心不存在的边。因此没有空间浪费，邻接表由数组+链表组成

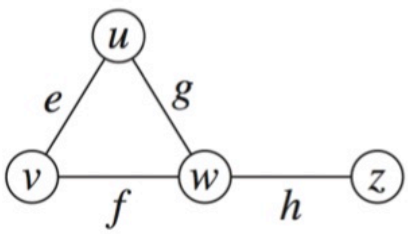


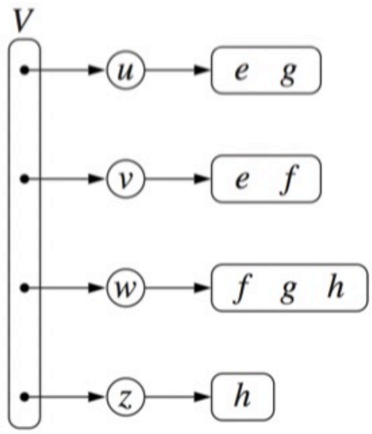
> 说明:
> 1) 标号为0的结点的相关联的结点为 1 2 3 4
> 2) 标号为1的结点的相关联结点为0 4,
> 3) 标号为2的结点相关联的结点为0 4 5
> 4) ....

Adjacency List Structure

The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex.

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

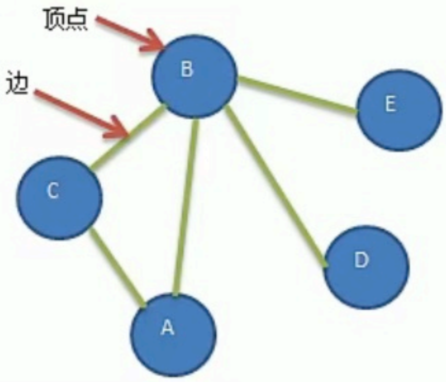




不存在空间浪费

6.3 图的代码实现 by Adjacent matrix

1) 要求: 代码实现如下图结构.



	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	0	0	0
E	0	1	0	0	0

//说明
//(1) 1 表示能够直接连接
//(2) 0 表示不能直接连接

2) 思路分析 (1) 存储顶点String 使用 ArrayList (2) 保存矩阵 int[][] edges
3) 代码实现

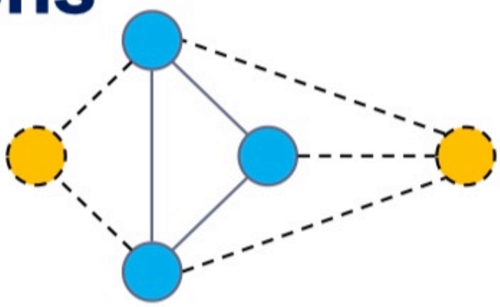
```java
class graph {
    private ArrayList<String> vertexList; // 存储顶点集合
    private int[][] edges; // 存储图对应的邻接矩阵
    private int numOfNodes; // 对应点的数目
    private int numOfEdges; // 对应边的数目

    // 构造器
    public graph(int n) {
        // 初始化矩阵 和 vertexList
        edges = new int[n][n];
        vertexList = new ArrayList<String>(n);

        numOfEdges = 0;
        numOfNodes = 0;
    }
```

```java
//插入节点
public void insertVertex(String vertex) {
    vertexList.add(vertex);
}

/**
 *
 * @param v1 表示点1的下标，即是第几个节点
 * @param v2 表示点2的下标
 * @param weight 表示是否关联，1or0
 */
//添加边
public void insertEdge(int v1, int v2, int weight) {
    edges[v1][v2] = weight;
    edges[v2][v1] = weight;
    numOfEdges++;
}
```

Subgraphs

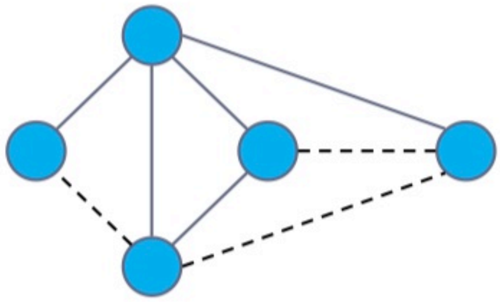
A subgraph S of a graph G is a graph such that:

- The vertices of S are a subset of the vertices of G.
- The edges of S are a subset of the edges of G.



Subgraph

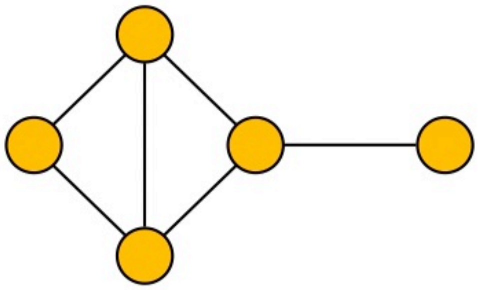
A spanning subgraph of G is a subgraph that contains all the vertices of G.



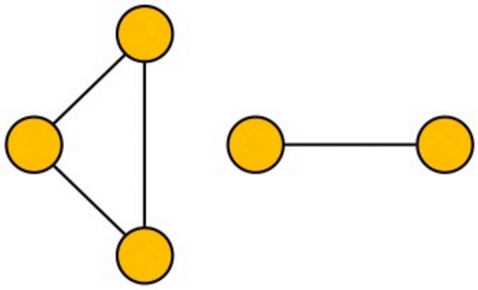
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph G is a maximal connected subgraph of G.



Connected graph

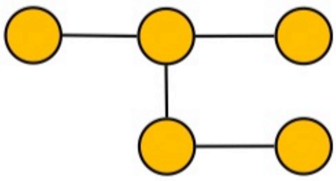


Non connected graph with two connected components

Trees and Forests

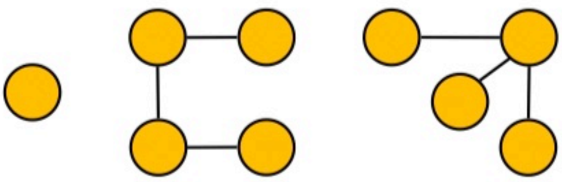
- A (free) tree is an undirected graph T such that:
  - T is connected
  - T has no cycles

  This definition of a tree is different from the one of a rooted tree



Tree

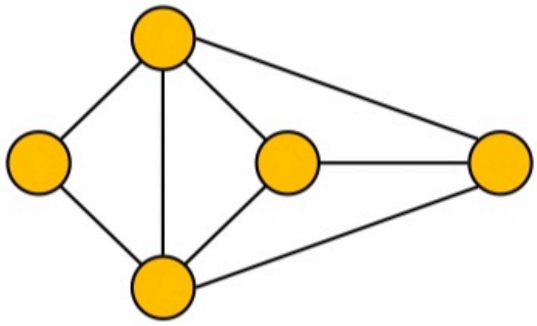
- A forest is an undirected graph without cycles.
- The connected components of a forest are trees.



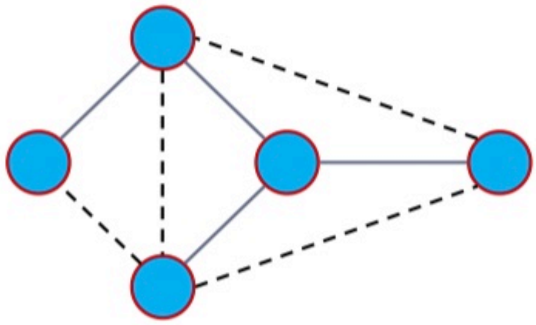
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
- A spanning tree is not unique unless the graph is a tree.
- Spanning trees have applications in the design of communication networks.
- A spanning forest of a graph is a spanning subgraph that is a forest.



Graph



Spanning tree

L7 Graph Search Approaches

- DFS (Depth-First search)
- BFS (Breadth-First Search)

7.1 DFS 深度优先

> **深度优先遍历：从初始结点出发，首先访问第一个临接点，然后用被访问的结点再去访问它的下一个临接点。**
>
> **每次都在访问完当前节点后，首先访问当前节点的第一个邻接节点。**

图遍历介绍

所谓图的遍历，即是对结点的访问。一个图有那么多个结点，如何遍历这些结点，需要特定策略，一般有两种访问策略: (1)深度优先遍历 (2)广度优先遍历

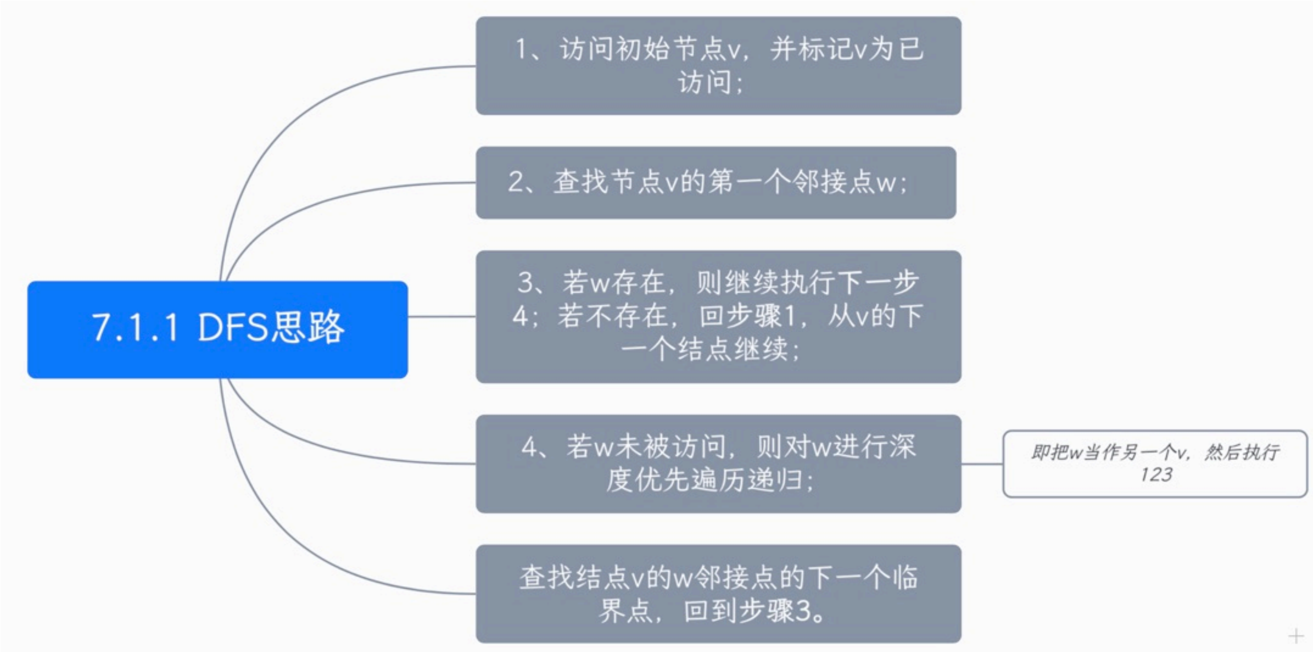
深度优先遍历基本思想

图的深度优先搜索(Depth First Search) 。

1) 深度优先遍历，从初始访问结点出发，初始访问结点可能有多个邻接结点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始结点，访问它的第一个邻接结点， 可以这样理解：每次都在访问完**当前结点**后首先访问**当前结点的第一个邻接结点**。
2) 我们可以看到，这样的访问策略是优先往纵向挖掘深入，而不是对一个结点的所有邻接结点进行横向访问。
3) 显然，深度优先搜索是一个递归的过程

7.1.1 DFS思路

- 1、访问初始节点v，并标记v为已访问；
- 2、查找节点v的第一个邻接点w；
- 3、若w存在，则继续执行**步骤4**；若不存在，回**步骤1**，从v的下一个结点继续；
- 4、若w未被访问，则对w进行深度优先遍历递归；
    - *即把w当作另一个v，然后执行**步骤123***
- 5、查找结点v的w邻接点的下一个临界点，回到**步骤3**。


7.1.1 DFS思路
1、访问初始节点v，并标记v为已访问；
2、查找节点v的第一个邻接点w；
3、若w存在，则继续执行下一步4；若不存在，回步骤1，从v的下一个结点继续；
4、若w未被访问，则对w进行深度优先遍历递归；
即把w当作另一个v，然后执行123
查找结点v的w邻接点的下一个临界点，回到步骤3。

7.1.2 DFS代码实现

```java
public void dfs() {
    checkList = new boolean[5];
    for (int i = 0;i<5;i++){
        if(!checkList[i]){
            dfs(i,checkList);
        }


    }
}

public void dfs(int a,boolean[] checklist){
    System.out.println(getValueByIndex(a)+"->");
    checklist[a] = true;
    int w = getFirstNeibour(a);
    while(w!=-1){
    if(checklist[w] == false){
        dfs(w,checklist);
    }
        w = getNextNeibour(a,w);
    }
}
```

7.2 BFS 广度优先 (Breadth-First Search)

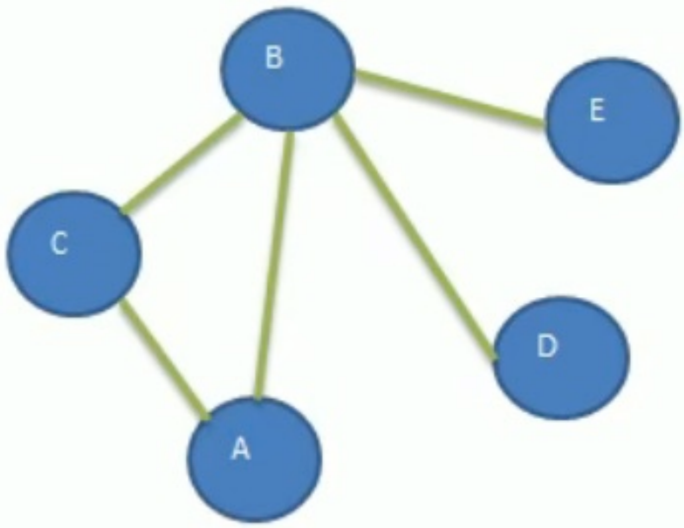
BFS：类似一个分层搜索的过程，广度优先遍历需要使用一个队列以保持访问过的结点的顺序，以便按这个顺序来访问这些结点的邻接结点。

7.2.1 BFS思路

广度优先遍历算法步骤

1) 访问初始结点v并标记结点v为已访问。
2) 结点v入队列
3) 当队列非空时，继续执行，否则算法结束。
4) 出队列，取得队头结点u。
5) 查找结点u的第一个邻接结点w。
6) 若结点u的邻接结点w不存在，则转到步骤3；否则循环执行以下三个步骤：

6.1 若结点w尚未被访问，则访问结点w并标记为已访问。
6.2 结点w入队列
6.3 查找结点u的继w邻接结点后的下一个邻接结点w，转到步骤6。



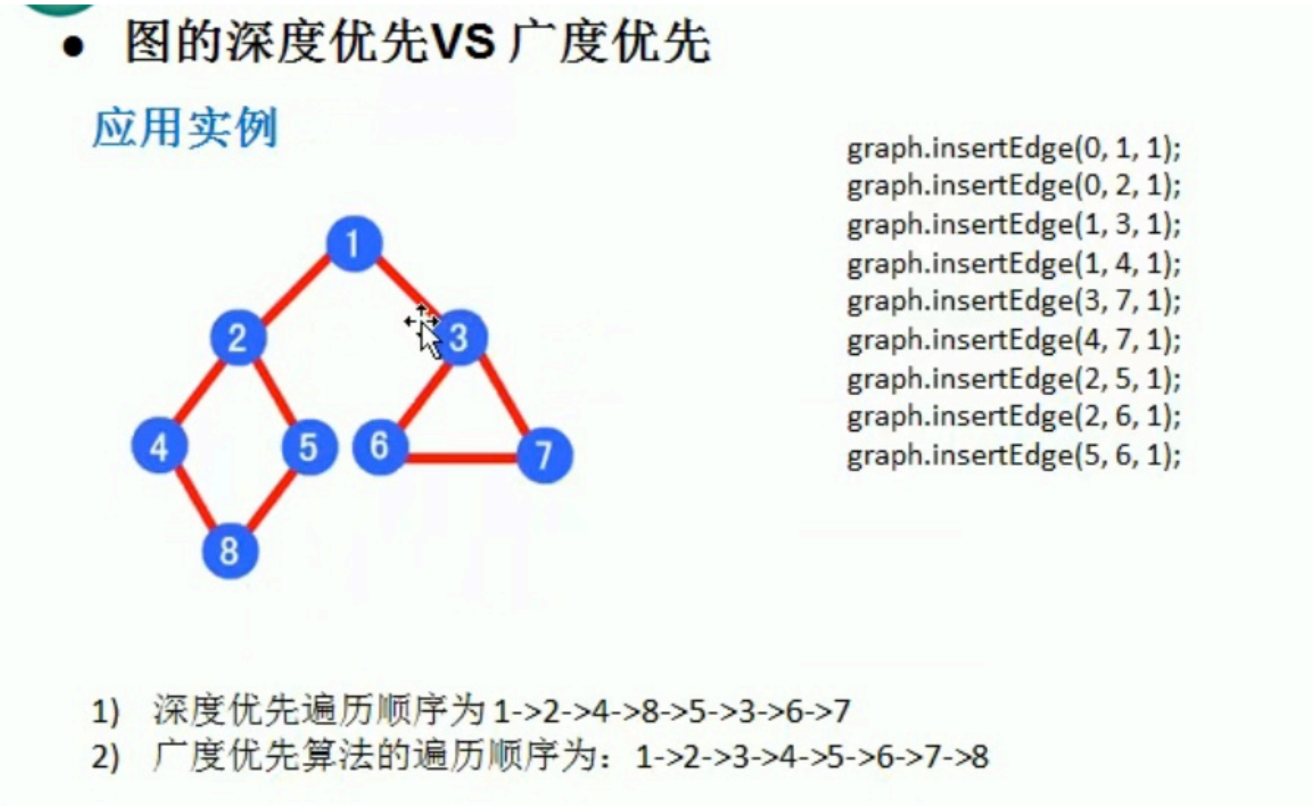
	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	0	0	0
E	0	1	0	0	0

//说明
//(1) 1 表示能够直接连接
//(2) 0 表示不能直接连接

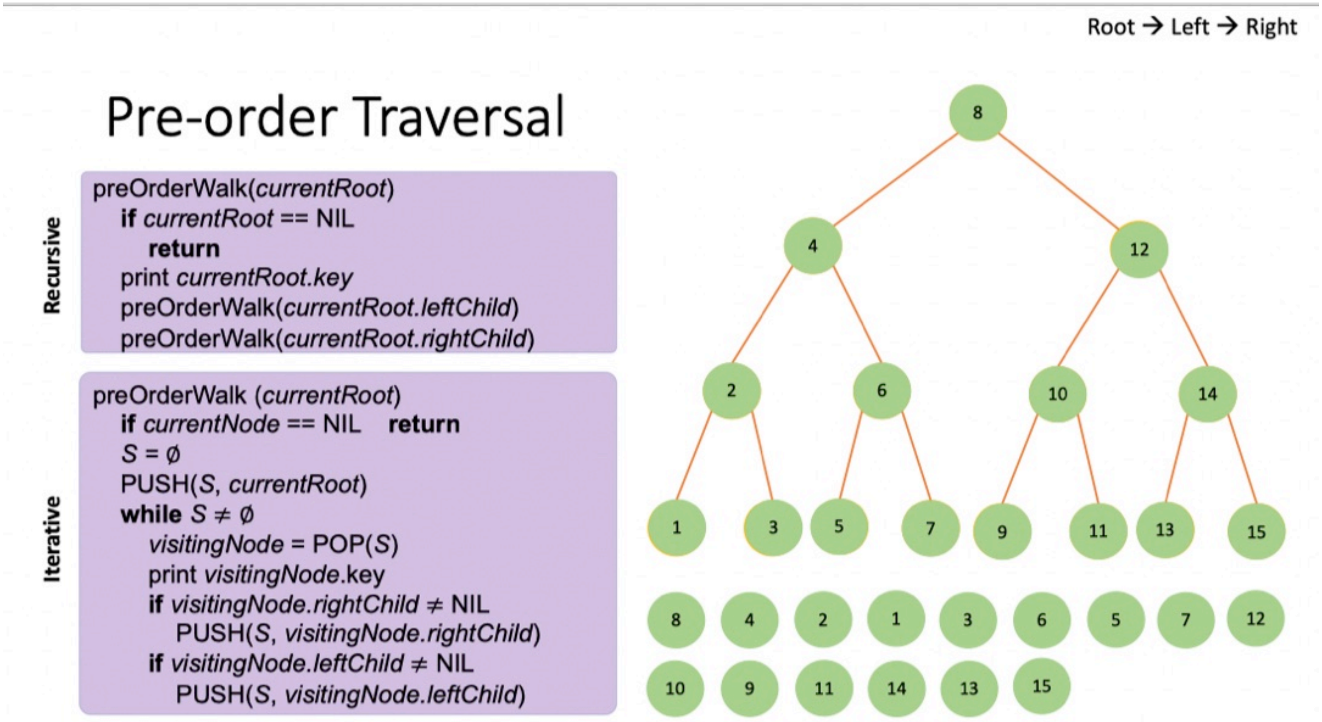
7.2.2 DFS代码实现

```java
public  void bfs(int a, boolean[] checkList){
    int u; //当前的列表下标
    int w;  //临接结点的下标
    LinkedList linkedList = new LinkedList();
    System.out.println(getValueByIndex(a)+"->");

    checkList[a] = true;
    linkedList.add(a);

    while(linkedList != null){
        u = (Integer) linkedList.removeFirst();
        w = getFirstNeibour(u);

        while(w!=-1){
            if(checkList[w] == false){
                System.out.println(getValueByIndex(w)+"->");
                checkList[w] = true;
                linkedList.add(w);
            }
            w = getNextNeibour(u,w);
        }
    }
}
```

DFS vs. BFS



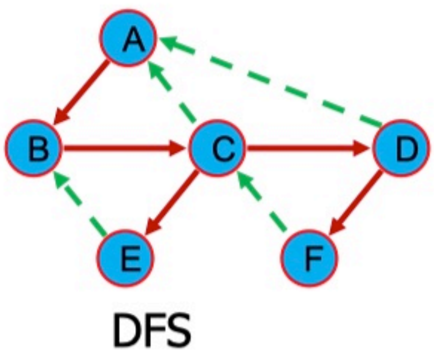
深度优先就类似前序遍历，**Root –> Left –> Right**

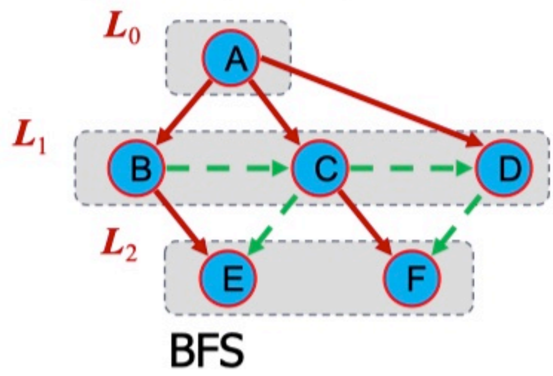


广度优先就是层序遍历，一层一层来

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	





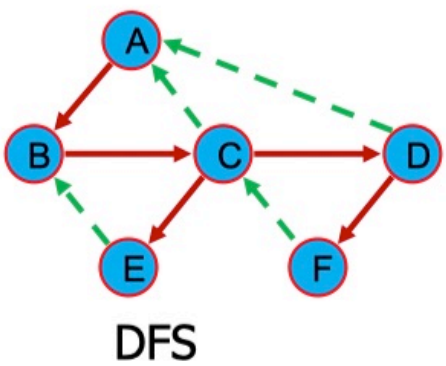
DFS vs. BFS (cont.)

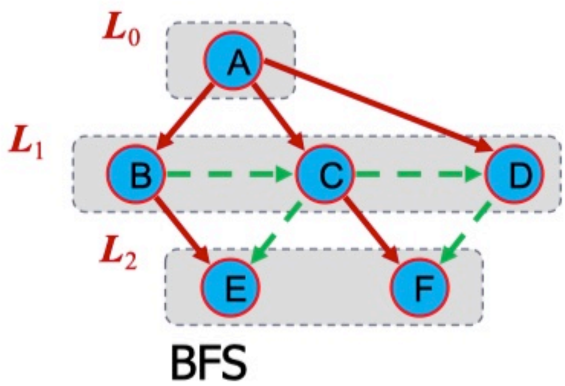
Back edge (v,w)

- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level





Note 5-1 Review

- Graphs intro
- DFS & BFS

`CS211 Note-5-1`

`by Lance Cai`

`2022/07/05`