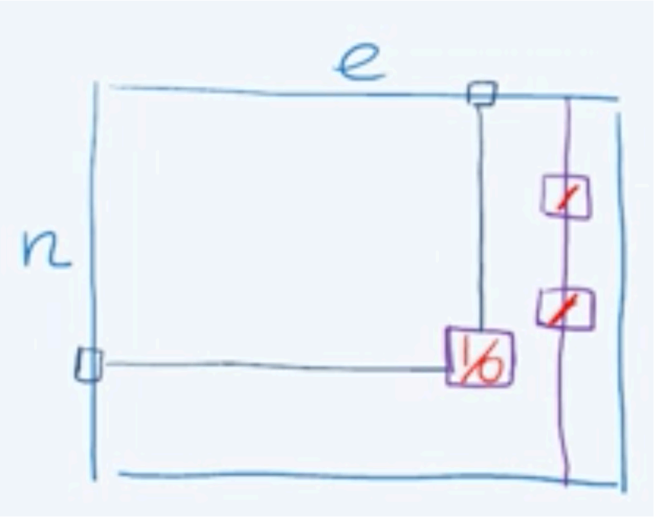
# Graph

## Representation

- adjacency matrix
- incidence matrix




- adjacency list

  ```
  Vertex* array [n];
  //array[i] is a link list which stores vertex i's neighbors(reach in exactly 1 step)
  ```

## Implementation

```
template <typename V,typename E>//Vertex type and
Edge type
class Graph{
  ...
};
```

# Graph Search

## Application

- web crawling
- social networking
- network broadcast
- grabage collection in many languages
- model checking(from one status to one/many other statuses)
- checking **mathematical conjecture**(usually used to find a counter example instead of a proof)
- solving games or puzzles

## Breadth-First-Search(BSF)

*The idea : visit neighbors of 'the start vertex', then neighbors of neighbors(not going back),.....,until we do not have neighbors*

- visit all vertices which are reachable from a give vertex $v \in V$
- in $O(|V| + |E|)$ time

- look at nodes reachable in 0 moves, in 1 moves(neighbors), in 2 moves(neighbors of neighbors),....., until we run out of the graph
- Carefully avoid the duplicated(prevent us from falling into an infinte loop)

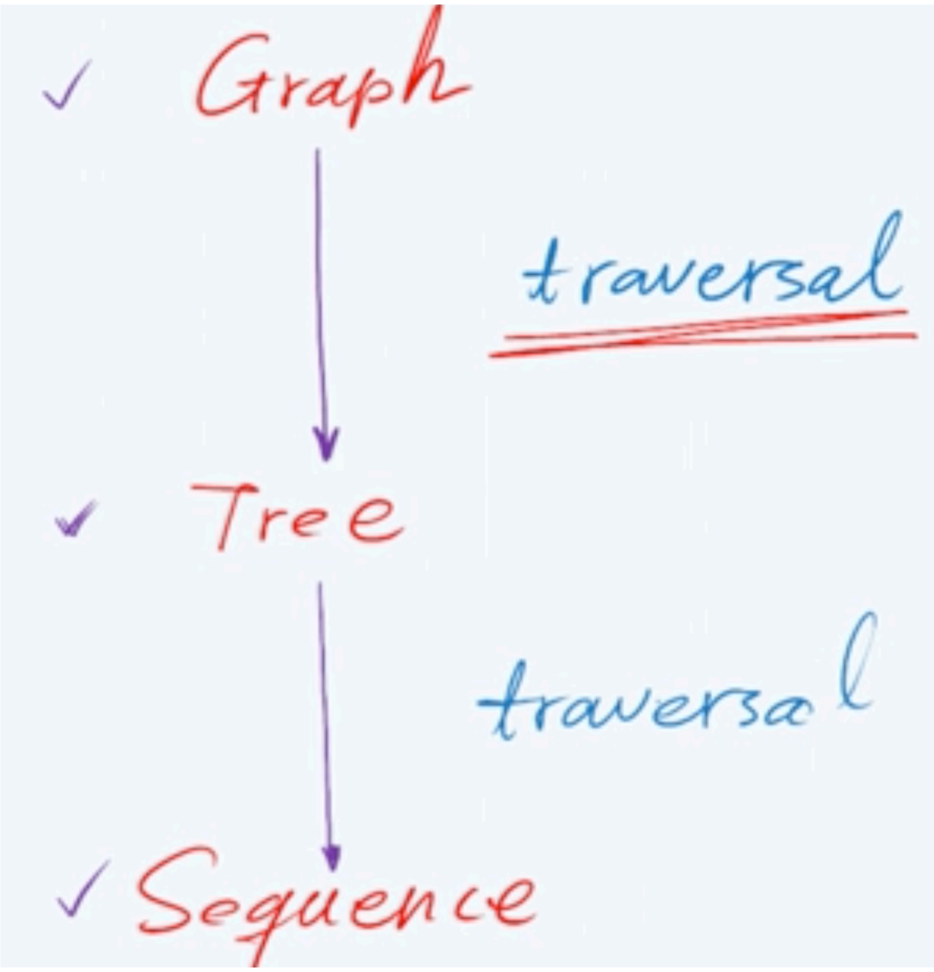
If we keep trace of the path we go through, we can construct a tree/forest whose root is 'the start vertex', and the pathes we obtained are all shortest pathes

```
def BFS(s , adj:dict):
    # initialization
    level = {s:0} # level is the distance from vertex i to vertex s
    parent = {s:None} # shortest path back to s
    i = 1
    # begin of the algorithm
    visitq = [s] # vertices which you can reach in 0(i-1) move , level 0
    while visitq:
        next = [] # vertices which you can reach in 1(i) move , level 1
        for u in visitq:
            for v in adj[u]:  # get all neighbors
                if v not in level:
                    level[v] = i
                    parent[v] = u
```

```
                    next.append(v)
        visitq = next
        i += 1
```

## 广度优先搜索(层次遍历)

*The idea : 化繁为简，把非线性结构转化为半线性结构*




### 实现：借助队列结构

- 只能遍历起始节点所在的联通域，不能遍历全图
- 解决：对所有节点遍历并应用BFS

## Depth-First-Search(DFS)

*The idea : it goes as deep as possible before backtracing, using recursion.(Just like solving a maze)*

- recursively explore a graph, backtracing as necessary
- Careful not to repeat
- visit each vertex and edge at most once, $O(|V|+|E|)$
- traverse all the vertices,$O(|V|)$, $\sum_{i=0}^{n} \mathrm{dfs}(v_i) = |E|$, $O(|V|+|E|)$

### Edge Classification

- Tree edge : visit a new vertex via that edge in `DFS`, they have a parent pointer(these edges form a forest/tree)
- Forward edge: go from one node to its descendant(in the tree/forest formed by Tree edge)
- Backward edge:go from one node to its ancestor(in the tree/forest formed by Tree edge)
- Cross edge: who not fit in the three categories above is a cross edge

keep trace of a variable which functions like a clock, recording the time when a node is Touched/Visited

```
edge(touched -> untouched) = TREE;
edge(touched -> touched) = BACK;
edge(touched -> visit) = touched.touchtime <
visit.touchtime ? FORWARD : CROSS;
```

NOTE: a 'global' clock for one graph!

### Cycle detection

*a graph has a cycle == has a backward edge*

### Topological Sort

*run `DFS`, then print the result in reverse*

- like job scheduling, some things must be done before other things can be done.(that is to say that some nodes must be visited before some other, exactly what we have done in DFS)

# 深度优先算法

*对于起始顶点S，若有未被访问的邻居则对任意一邻居执行DFS，否则返回*

## 括号引理

❖ 顶点的活动期：active[u] = ( dTime[u], fTime[u] )

❖ Parenthesis Lemma：给定有向图G = (V, E)及其任一DFS森林，则

u是v的后代 iff active[u] ⊆ active[v]

u是v的祖先 iff active[u] ⊇ active[v]

u与v"无关" iff active[u] ∩ active[v] = ∅