树与图论

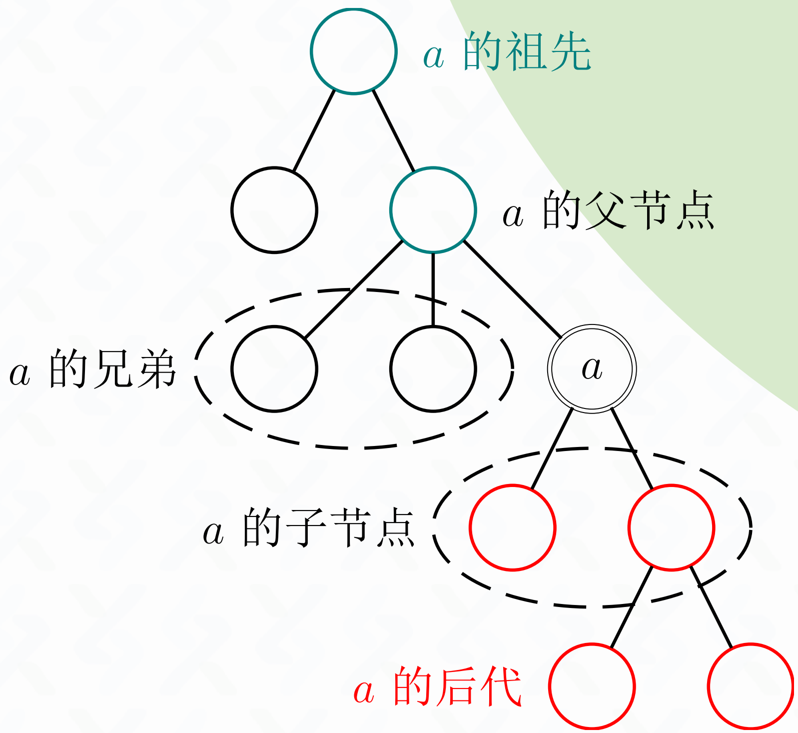
基础-提高衔接计划
览遍千秋
2025-08-30

树与二叉树

一些术语

- 有根树/无根树：有没有指定一个结点为树根。一般的无根树可以指定 1 为根结点转换为有根树。
- 祖先结点：一个结点到根结点路径上，除自己以外的其他结点。
- 后代结点：如果 u 是 v 的祖先结点，那么 v 是 u 的后代结点。
- 父结点：最近的祖先结点。
- 子结点：如果 u 是 v 的父节点，那么 v 是 u 的子结点。
- 子树：一个结点的全部后代结点组成的子图。

一些术语



特殊的树

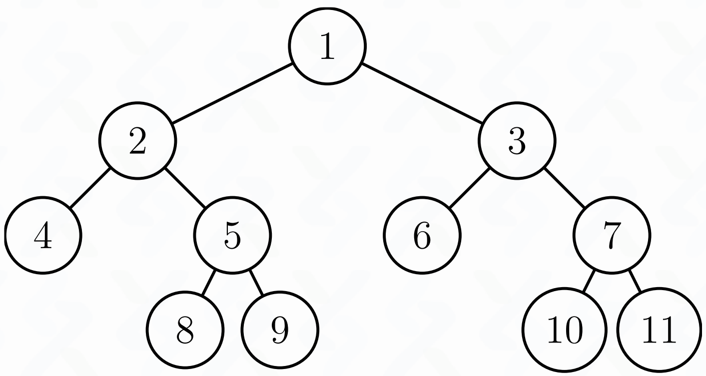
- 链：和任意结点相连的结点不超过两个。有且只有两个结点，只和另外一个结点相连。
- 菊花图：一个节点连接其他所有结点。

常作为树/图试题的特殊性质出现。

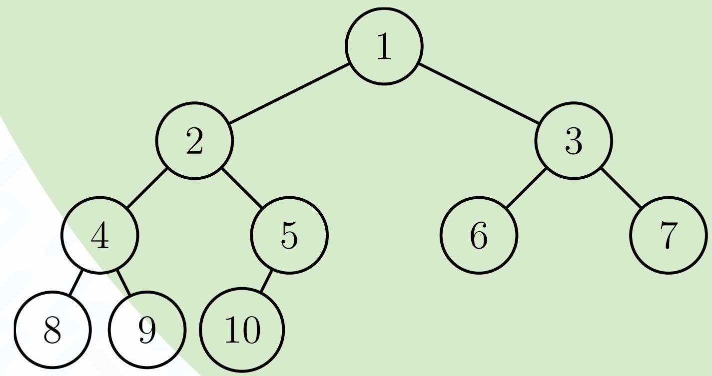
二叉树

- 二叉树：任意一个结点的子结点不超过两个的树。
- **完整二叉树（full/proper binary tree）**：每个结点的子结点数量均为 0 或者 2 的二叉树。换言之，每个结点或者是树叶，或者左右子树均非空。
- **完全二叉树（complete binary tree）**：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。
- **完美二叉树（perfect binary tree）**：所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树。（满二叉树多指完美二叉树。）

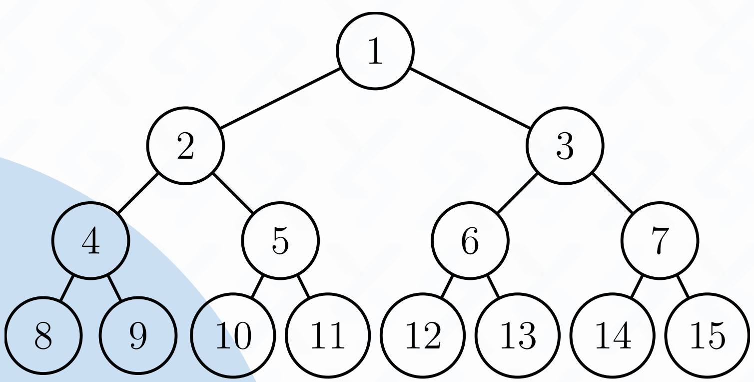
二叉树



完整二叉树（proper binary tree）



完全二叉树（complete binary tree）



完美二叉树（perfect binary tree）

二叉搜索树

二叉搜索树（Binary Search Tree, BST）是一棵具有特殊性质的二叉树。又可以叫二叉排序树。

二叉搜索树的性质:

- 对于结点 x，左子树全部结点的权值均比 x 的权值小
- 对于结点 x，右子树全部结点的权值均比 x 的权值大

二叉搜索树的插入

构建一棵二叉搜索树，就是根据二叉搜索树的性质，不断向二叉搜索树插入数值的过程。

二叉搜索树的性质：

- 对于结点 x，左子树全部结点的权值均比 x 的权值小
- 对于结点 x，右子树全部结点的权值均比 x 的权值大

用变量 p 表示当前正在考虑的结点。很容易根据结点 p 对应的值，确定需要插入的数应该在左子树还是右子树，并递归处理。直到 p 对应的结点为空，在对应位置插入。

二叉搜索树的插入

例：对序列 [4,1,5,3,2,6,7] 构建二叉搜索树。

二叉搜索树的插入

二叉搜索树的查找

在二叉搜索树中查找权值为 val 的结点。

二叉搜索树的性质:

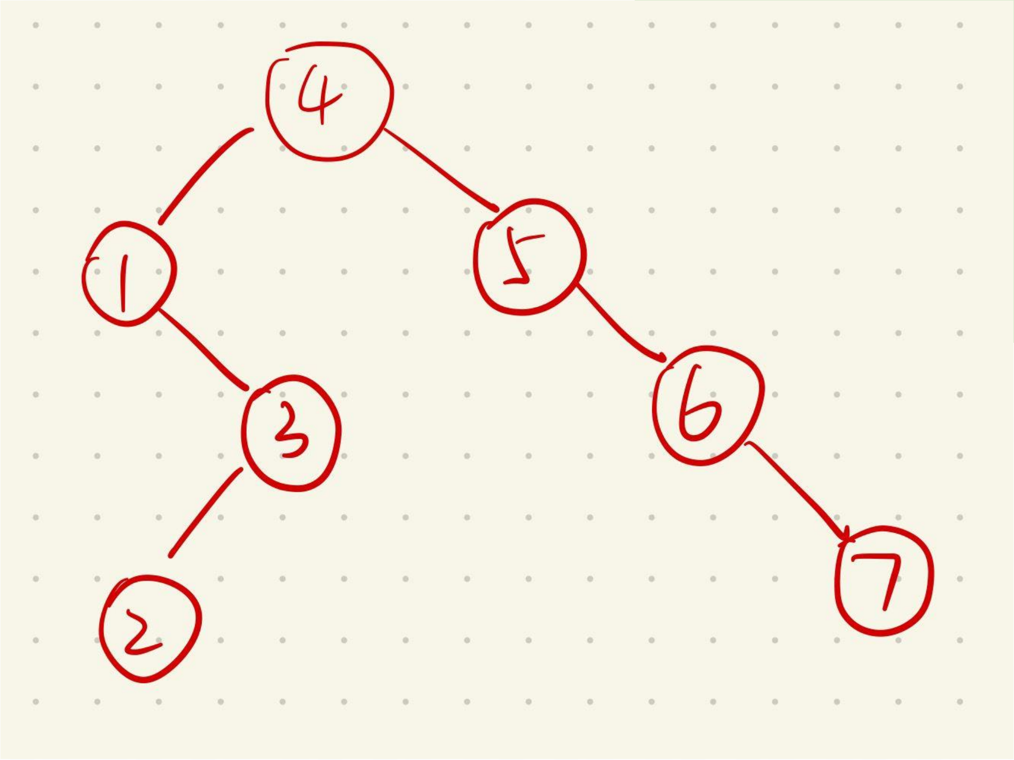
- 对于结点 x，左子树全部结点的权值均比 x 的权值小
- 对于结点 x，右子树全部结点的权值均比 x 的权值大

用变量 p 表示当前遍历到 p 号结点。

- 若结点 p 为空节点，返回 Error
- 若结点 p 的权值即为 val，返回 p
- 若结点 p 的权值小于 val，递归到 p 的左子树查找
- 若结点 p 的权值大于 val，递归到 p 的右子树查找

二叉搜索树的查找

下面是一棵二叉搜索树，在其中查找权值为 2 的结点。



二叉搜索树的查找

二叉搜索树的删除

从二叉搜索树中删去权值为 val 的结点。

核心：保持二叉搜索树的性质

二叉搜索树的性质：

- 对于结点 x，左子树全部结点的权值均比 x 的权值小
- 对于结点 x，右子树全部结点的权值均比 x 的权值大

首先，需要找到权值为 val 的结点，与前面的查找过程相同，记这个结点为 p。

二叉搜索树的删除

除了被删除结点 p 以外，还需要处理以下结点的信息：

- p 的父节点
- p 的子节点

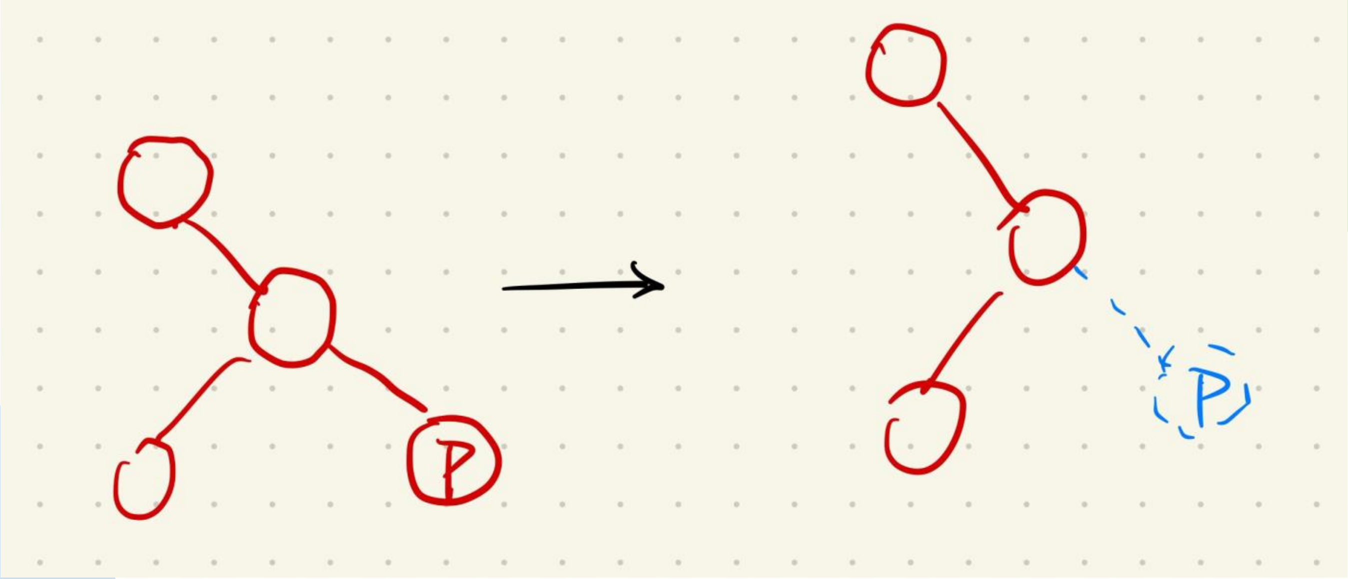
结点 p 共有几种情况：

- 结点 p 有 0 个子节点（叶结点）
- 结点 p 有 1 个子节点
- 结点 p 有 2 个子节点

二叉搜索树的删除

- 结点 p 为叶结点

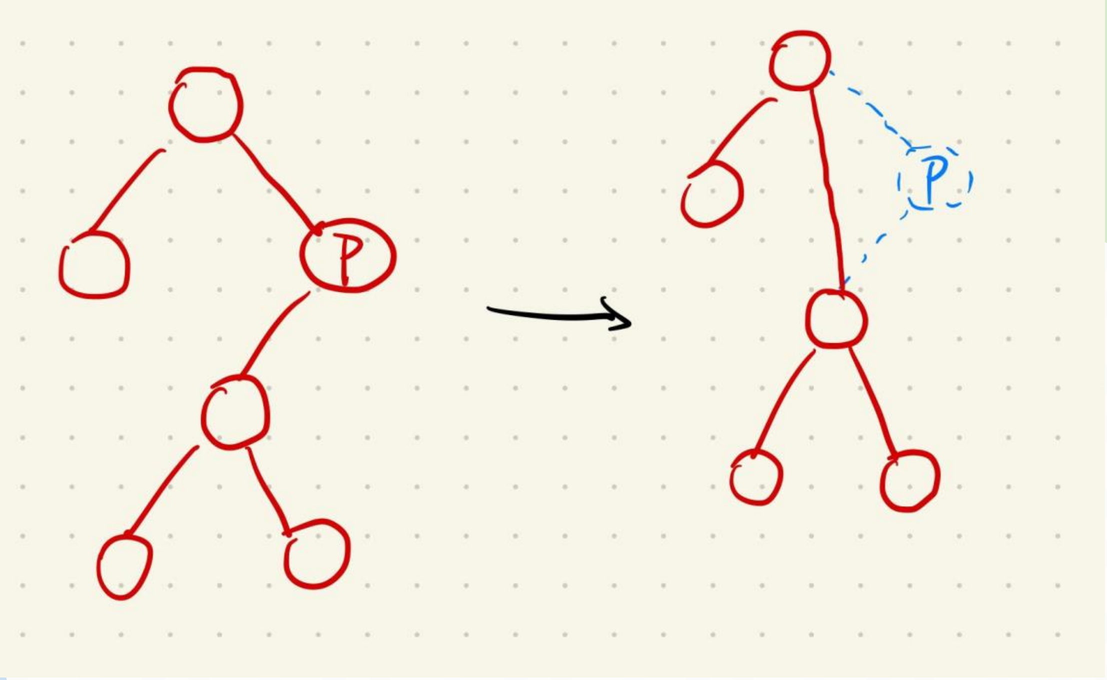
直接删去即可，将其父结点对应的子结点置空。



二叉搜索树的删除

- 结点 p 有 1 个子节点

直接将子节点链接到 p 的父节点， 作为同位子树。



二叉搜索树的删除

- 结点 p 有 2 个子节点

一般使用左儿子的最大值（或右儿子的最小值）来替代结点 p。

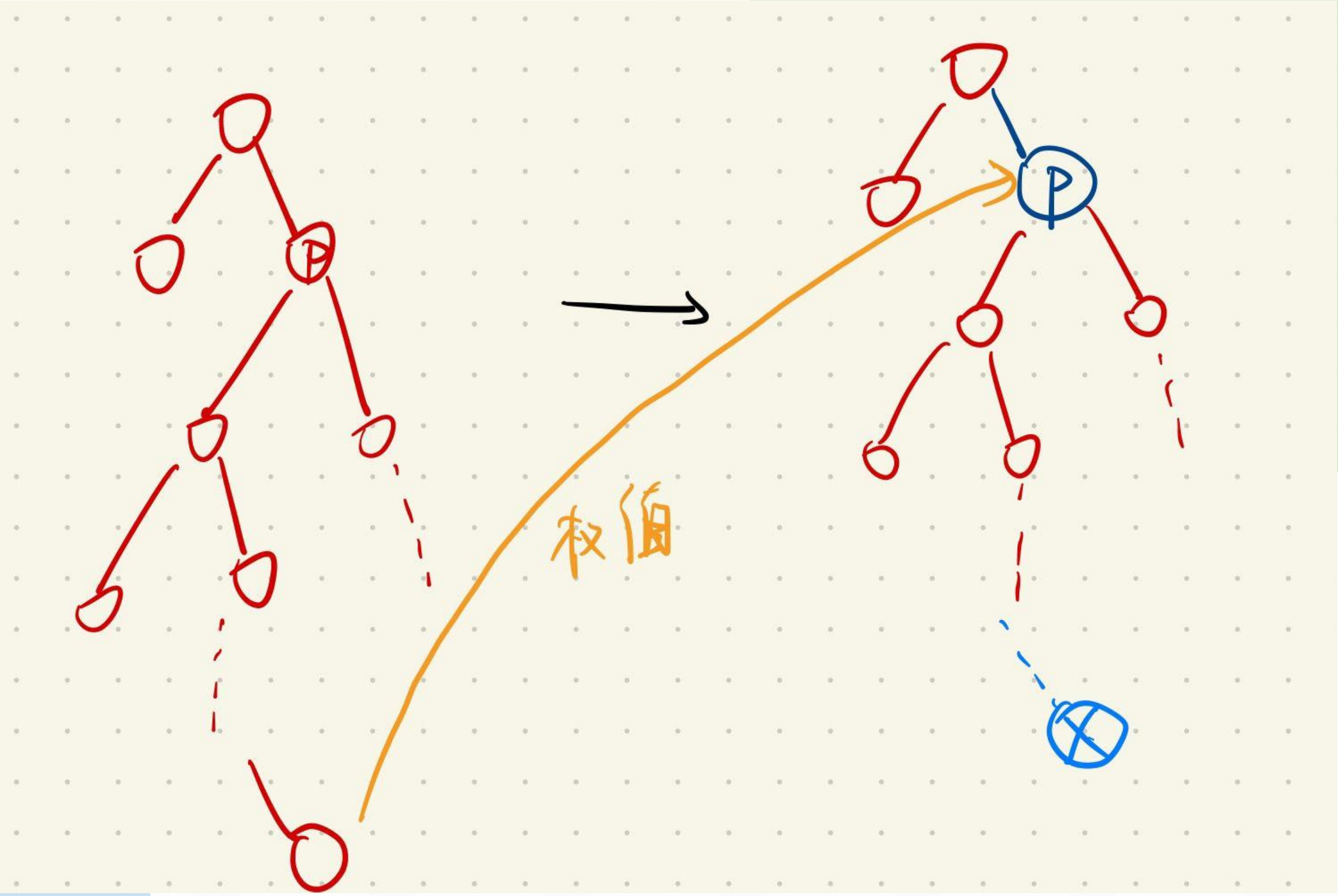
左儿子的最大值，即左子树中最靠右的叶结点

右儿子的最小值，即右子树中最靠左的叶结点

删去对应的叶结点，将 p 的权值修改为左子树的最大值/右子树的最小值。

二叉搜索树的删除

- 结点 p 有 2 个子节点



二叉搜索树的删除

最大/最小生成树

Kruskal 算法

并查集

并查集是一种用于管理元素所属集合的数据结构。

并查集表现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

树根为这个集合的“代表元”。

使用数组 $fa[x]$，记录元素 x 的父节点。

特别的，树根结点的父节点记录为自己。

即，初始化并查集时，将全部的 $fa[i]$ 赋值为 i。

并查集

例如，有集合 $\{1\},\{2\},\{3\},\{4\},\{5\}$。

合并集合 $\{1\},\{3\}$

并查集

- 查找一个集合的代表元

一个集合的代表元是该集合的树根。

查找 x 所在集合的代表元，沿着 $fa[x]$ 一路向上找到根结点即可。

```
int find(int x) {
    return fa[x] == x? x: find(fa[x]);
}
```

并查集

- 判断 u, v 是否属于一个集合

“代表元”的“代表”，唯一性。

查找 u 和 v 的代表元，如果相同，则属于同一集合。

并查集

- 合并

合并 u 和 v 所在的集合，是在 u,v 所在集合的代表元之间连边，建立父子关系。

首先，需要获取 u,v 的代表元，已经在前面讲过。分别记为 fu,fv。

建立父子关系，只需要 fa[fu]=fv 即可。

```
void merge(int u, int v) {
    int fu = find(u), fv = find(v);
    if(fu != fv) fa[fu] = fv;
}
```

路径压缩

在前面的 find 函数中，是按照 $fa[x]$ 逐级跳往根结点，如果树形态为一条链，则时间复杂度不可接受。

我们只关心一个元素的代表元是谁，因此不需要维持严格的祖先关系，$fa[x]$ 为 x 的任意一个祖先都可以。

因此，可以在 find 的过程中，同步修改 fa[x]，改浅树的深度。

代码修改很简单，因为重点内容是最小生成树，并查集只是工具，就不深入了。

```
int find(int x) {
    return fa[x] == x? x: fa[x]=find(fa[x]);
}
```

生成树

图 $G=(V,E)$，其中 V 是点集，E 是边集。

图 $G' = (V',E')$ 是图 G 的一棵生成树，当且仅当 $V'=V$，$E' \subset E$，G' 是一棵树。

省流：从图中选出 $n-1$ 条边连接 n 个结点，构成的树。

最小生成树

最大生成树与最小生成树性质类似，后面均以最小生成树为例。

当边有边权，边权和最小的生成树称为最小生成树。

Kruskal

一棵树可以理解为 n 个结点，$n-1$ 条边组成的**联通图**。

也就是说，断开一条边，树会被分为两个**联通块**。

类似地，断开 k 条边，树会被分为 $k+1$ 个联通块。

Kruskal

考虑一棵树从 n 个结点逐步联通的过程。

一开始是 n 个独立的点（联通块），每增加一条边，减少一个联通块。

直到选出 $n-1$ 条边，构造完成。

如何维护联通块?

并查集。

Kruskal

考虑最小生成树的构造过程，即选出 $n-1$ 条边的顺序。
有一个贪心策略:
按照边权升序排序，依次考虑是否选取。

选或不选的标准?
如果两个端点 u, v 在一个联通块里，不选。
否则，选。

Kruskal

流程:

- 建立并查集
- 按照边权排序，依次扫描每条边
- 如果 u, v 属于同一联通块，则忽略，扫描下一条
- 否则，选择这条边，合并两个集合

时间复杂度为 sort 的复杂度，$O(m \log m)$

Kruskal

为什么这样的贪心是对的?

考虑一条边 (u, v, w)

如果没有被选择，说明已经存在别的路径，使得 u, v 连通

如果要选择 (u, v, w)，就需要放弃另一条更短的边

必然使得总长度增加

P3366 最小生成树

例题 1：https://www.luogu.com.cn/problem/P3366

给出一个无向图，求其最小生成树。

```
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3
```

```
7
```

单源最短路

Dijkstra 算法

单源最短路问题 SSSP

给出 n 个结点 m 条边组成的图 G，给定起点 S，求从 S 出发，到其他任何结点的最短路径。

Dijkstra

Dijkstra 是一种用于解决单源最短路径问题（Single Source Shortest Path, SSSP）的算法。

其流程如下：

- 初始化 $dis(S)$ 为 0，其他 dis 为 $+\infty$
- 在未被标记的点中找到 $dis(x)$ 最小的 x，并标记结点 x
- 扫描 x 的所有出边 (x, y, z)，如果 $dis(y) > dis(x) + z$，则更新 $dis(y)$
- 重复步骤 2~3 直至所有结点被标记

Dijkstra

Dijkstra 只能处理非负边权的图。

其思想基于贪心：当边长为非负数，全局最小值必然不可能再被更新。

即，每次在第二步中取出的 x，$dis(x)$ 已经是起点到 x 的最短路径。

标记是因为，$dis(x)$ 只能更新别的结点一次，保证时间复杂度。

Dijkstra

Dijkstra

上面的代码时间复杂度为 $O(n^2)$。

事实上，其时间复杂度瓶颈为寻找结点 x，这个过程可以用优先队列 `priority_queue` 优化。

优化后均摊时间复杂度为 $O((n+m)\log n)$。

- `pair <type, type>`
- 优先以第一关键字比大小，第一关键字相同按第二关键字比
- `make_pair(a, b)`，得到一个以 a 为第一关键字，b 为第二关键字的 pair

P4779 单源最短路径（标准版）

例题 2：https://www.luogu.com.cn/problem/P4779

给出一个 n 个点 m 条边的有向图，求从 s 出发到每个点的最短距离。

- $n \le 10^5$
- $m \le 2 \times 10^5$
- $w_i \le 10^9, \sum w_i \le 10^9$

```
4 6 1
1 2 2
2 3 2
2 4 1
1 3 5
3 4 3
1 4 4
```

```
0 2 4 3
```

P4568 飞行路线

https://www.luogu.com.cn/problem/P4568

Alice 和 Bob 现在要乘飞机旅行，他们选择了一家相对便宜的航空公司。该航空公司一共在 n 个城市设有业务，设这些城市分别标记为 0 到 $n-1$，一共有 m 种航线，每种航线连接两个城市，并且航线有一定的价格。

Alice 和 Bob 现在要从一个城市沿着航线到达另一个城市，途中可以进行转机。航空公司对他们这次旅行也推出优惠，他们可以免费在最多 k 种航线上搭乘飞机。那么 Alice 和 Bob 这次出行最少花费多少？

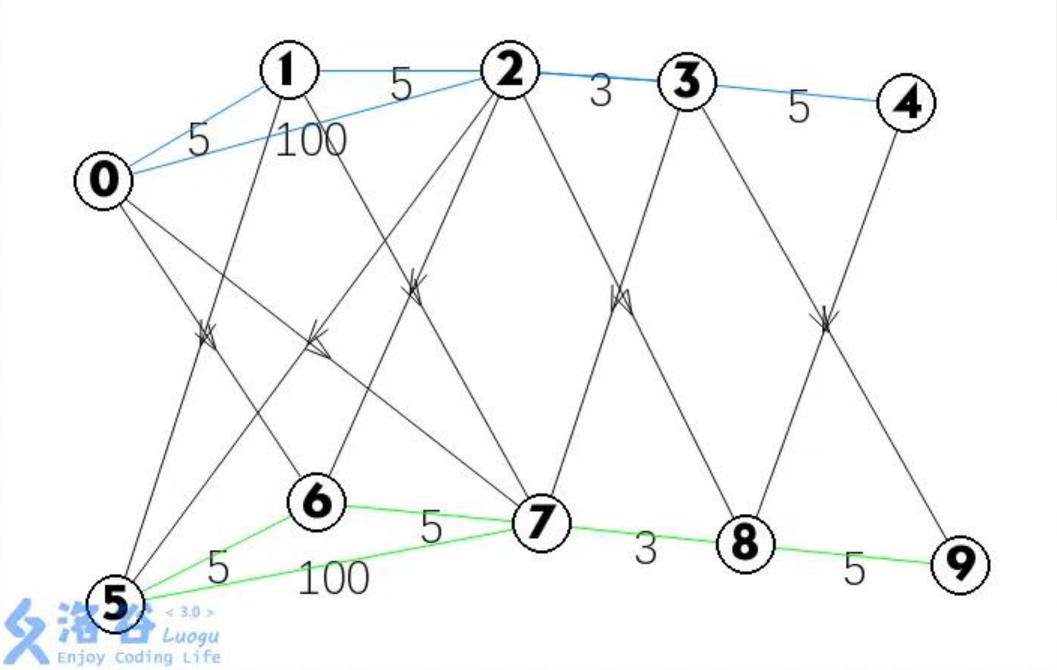
$$k \leq 10$$

分层图模型

将图重复建立 k 层，每层的图为原图

跨层边，由上一层起点指向下一层终点，边权为 0

每向下走一层，使用一次免费机会。



树上 dp

树上 dp

树上 dp 是在树形数据结构上进行的动态规划

往往以子树为单位进行状态设计

如 $dp[x][\ldots]$ 表示以 x 为根的子树中，满足某某条件的最大答案/最小步数

P1352 没有上司的舞会

https://www.luogu.com.cn/problem/P1352

P1352 没有上司的舞会

用 $dp[x][0/1]$ 表示子树 x 中且 x 选/不选可以取得的最大快乐指数

注意不确定根节点，需要找到没有上司的那个

$$dp[x][1]=\sum dp[v][0]$$

$$dp[x][0]=\sum \max(dp[v][0],dp[v][1])$$

P1040 加分二叉树

https://www.luogu.com.cn/problem/P1040

P1040 加分二叉树

中序遍历：左根右

设 $dp[l][r]$ 形成一棵树可以得到的最大分数

枚举 k，左子树：$[l, k-1]$，右子树 $[k+1, r]$

链式前向星（选学）

图的一种存储方式

链式前向星

正常情况下，用 vector 存储图就可以了

但是在网络流等算法中，需要求反向边时，链式前向星就存在一些优势。

选学内容，不要求必须掌握。

链式前向星

链式前向星以链表方式存储每一个结点出发的所有边。

链式前向星主要涉及到边计数器 tot 与以下一些数组

- Head 数组
- Next 数组
- to 数组、边权数组（如有）

to 数组、边权数组、边计数器，按照顺序编号的方式存储在数组中。

Head 数组和 Next 数组，则是用来存储链表相关信息的数组。

链式前向星

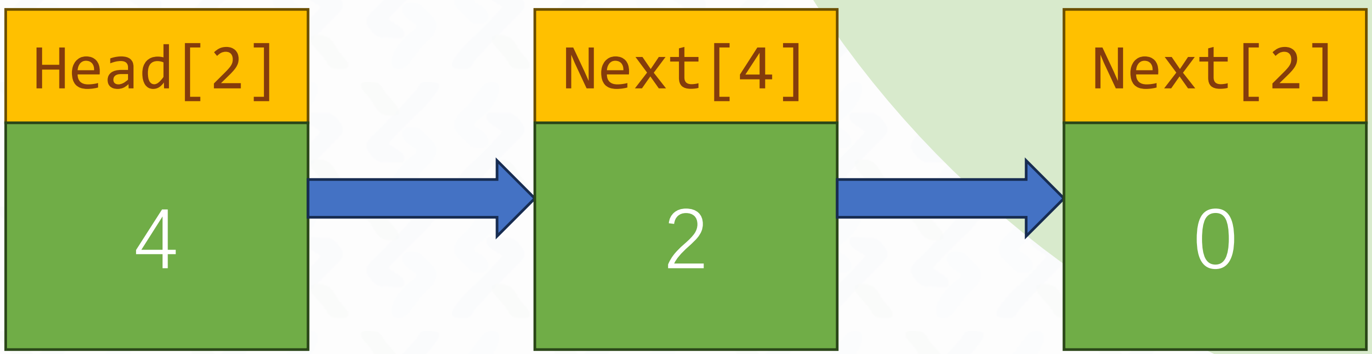
链式前向星为每一个结点建立一条链表。

`Head` 数组存储链表表头，`Head[x]` 表示结点 x 对应链表的第一条边的编号。

边的编号？边计数器 `tot`。

`Next` 数组存储的是边的下一条边的编号。`Next[i]` 表示编号为 i 的边对应的下一条边。

链式前向星



链式前向星

如何插入一条 $x \to y$ 的边?

首先，存储边的信息:

```
++tot;
to[tot] = y;
```

（如果还需要存边权什么的，就以此类推存下来就好）

接着，进行链表相关的操作:

```
Next[tot] = Head[x];
Head[x] = tot;
```

链式前向星

那么，如何遍历结点 x 的所有出边?

和链表一致，开始时取链头结点 Head，之后取 Next。

```
for(int i = Head[x]; i; i = Next[i])
```

这里的 `i` 是边的编号。

链式前向星

注意到加入链式前向星的是有向边 $x \to y$

那么无向图如何处理?

无向边 $x \leftrightarrow y$ 可以拆分成两条有向边 $x \to y$、$y \to x$。

分别加入链式前向星即可。

链式前向星

```
void add(int x, int y) {
    to[++tot]=y;
    Next[tot] = Head[x];
    Head[x] = tot;
}
void undirected_edge(int x, int y) {
    add(x, y);
    add(y,x);
}
```

链式前向星

```
void dfs(int x) {
    // do something
    for(int i = Head[x]; i; i = Next[i]) {
        int v = to[i], val = w[i];
        // do something
    }
    // do something
}
```

链式前向星

边的起始编号不影响链式前向星的存储。

编号从 1 开始也可以，从 2 开始也可以。

无向边 $x \leftrightarrow y$ 可以拆分成两条有向边 $x \to y$、$y \to x$。两者互为反边。

如果编号从 2 开始，那么第 i 条边的反边就是 $i \operatorname{xor} 1$，如果需要对反边进行一些操作，非常方便。