树

结点的度：结点拥有的子树的数量，树中所有结点的度数的最大值为树的度

度为 0：叶子结点或者终端结点

度不为 0：分支结点或者非终端结点

祖先结点：根结点到该结点的唯一路径的任意结点

双亲结点：根结点到该结点的唯一路径上最接近该结点的结点

兄弟结点：有相同双亲结点的结点

层次：根为第一层，它的孩子为第二层，以此类推

结点的深度：根结点开始自顶向下累加

结点的高度：叶节点开始自底向上累加

树的高度（深度）：树中结点的最大层数

树的性质

总节点数 = 总度数 + 1

总度数 = 每个节点的度数之和

m 叉树：每个结点最多只能有 m 个孩子的树（可以是空树）

高度为 h 的 m 叉树至少有 h 个结点，至多有(m^h-1)/(m-1)个结点（等比数列）

度为 m 的树：任意结点的度<=m，至少有一个节点度为 m（一定是非空树）

[2016 年] 若森林 F 有 15 条边、 25 个结点，则 F 包含树的个数是（ ）

A. 8　　B. 9　　C. 10　　D. 11

[2022 年] 若三叉树 T 中有 244 个结点（叶结点的高度为 1），则 T 的高度至少是（　）.

A. 8　　B. 7　　C. 6　　D. 5

二叉树

二叉树是 n（n≥0）个结点的有限集合：

① 或者为空二叉树，即 n=0。

② 或者由一个根结点和两个互不相交的被称为根的左子树 和右子树组成。左子树和右子树又分别是一棵二叉树。

1. 每个结点最多有两棵子树。

2. 左右子树有顺序

性质：

1. 非空二叉树上叶子结点数等于度为 2 的结点数加 1 即 n0 = n2 + 1

2. 非空二叉树上第 K 层上至多有 2k-1 个结点（K≥1）

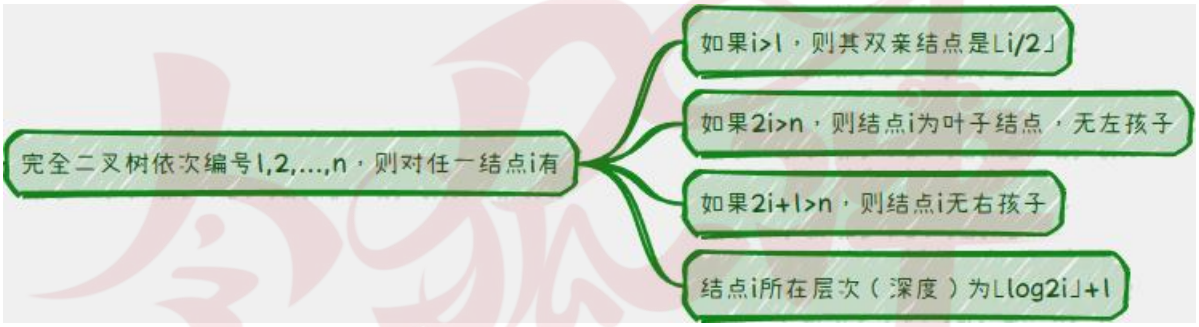
3. 高度为 H 的二叉树至多有 2H-1 个结点（H≥1）

4. 具有 N 个（N>0）结点的完全二叉树的高度为「 log2(N+1)┐或L log2N」 +1。

特殊二叉树

满二叉树：高度为 h，且含有 2^h-1 个结点的二叉树称为满二叉树

完全二叉树：高度为 h，有 n 个结点的二叉树，当且仅当其每个结点都与高度为 h 的满二叉树中编号为 1-n 的结点一一对应



[2009 年] 已知一棵完全二叉树的第 6 层（设根为第 1 层）有 8 个叶结点，则该完全二叉树的结点个数最多是（ ）.

A. 39　　B. 52　　C. 111　　D. 119

[2011 年] 若一棵完全二叉树有 768 个结点，则该二叉树中叶结点个数是（ ）

A. 257　　B. 258　　C. 384　　D. 385

二叉树存储结构

顺序存储

用一组地址连续的存储单元存储二叉树，为了反映二叉树结点间的逻辑关系，需要添加很多不存在的空结点

链式存储

```
typedef struct Node {
    int data;
    Node *left, *right;
    Node *parent; // 父节点指针，可选
} Node;
```

[2020 年] 对于任意一棵高度为 5 且有 10 个结点的二叉树，若采用顺序存储结构保存，每个结点占 1 个存储单元（仅存放结点的数据信息），则存放该二叉树需要的存储单元数量至少是（　）.

A. 31　　B. 16　　C. 15　　D. 10

二叉树先序遍历

依次递归遍历根节点、左子树、右子树，递归出口为当前节点为空

```
void preOrder(Node* root) {
    if (!root) return;//递归出口
    visit(root);// 具体的操作
    preOrder(root→left);
    preOrder(root→right);
}
```

二叉树中序遍历

依次递归遍历左子树、根节点、右子树，递归出口为当前节点为空

```
void inOrder(Node* root) {
    if (!root) return;//递归出口
    inOrder(root→left);
    visit(root);// 具体的操作
    inOrder(root→right);
}
```

二叉树后序遍历

依次递归遍历左子树、右子树、根节点，递归出口为当前节点为空

```
void postOrder(Node* root) {
    if (!root) return;//递归出口
    postOrder(root→left);
    postOrder(root→right);
    visit(root);// 具体的操作
}
```

二叉树的层序遍历(BFS)

若树为空，则什么都不做直接返回。否则从树的第一层开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

```
void levelOrder(Node* root) {
    if (!root) return;
    Queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* node = q.top();
        q.pop();
        if (q→left) q.push(q→left);
        if (q→right) q.push(q→right);
    }
}
```

[2012 年] 若一棵二叉树的前序遍历序列为 aebdc，后序遍历序列为 bcdea 则根结点的孩子结点（　）.

A. 只有 e B. 有 e、b C. 有 e、c D. 无法确定

线索二叉树

将结点空的指针指向其前驱和后继，这种改变指向的指针称为“线索”，加上线索的二叉树称为线索二叉树

优点：方便找到前驱和后继；遍历可以不从根结点开始

```
typedef struct Node {
    Node *left, *right;
    int data;
    bool ltag, rtag; //true表示线索，false表示孩子
} Node;
```

后序线索二叉树不能有效求后序后继

[2013 年] 若 X 是后序线索二叉树中的叶结点，且 X 存在左兄弟结点 Y，则 X 的右线索指向的是（ ）.

A. X 的父结点

C. X 的左兄弟结点 Y

B. 以 Y 为根的子树的最左下结点

D. 以 Y 为根的子树的最右下结点

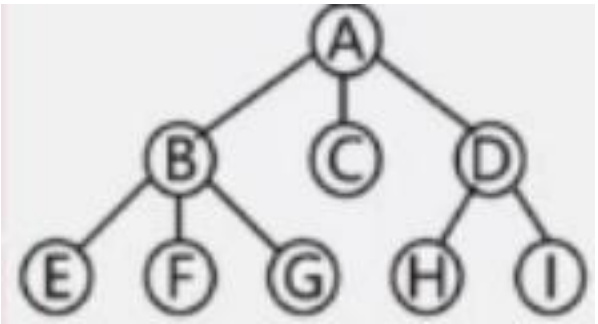
树、森林和二叉树的转换

树转换为二叉树

加线：在兄弟之间架一条线

抹线：对每个结点，除了左孩子外，去除其与其余孩子之间的关系

旋转：以树的根结点为轴心，将树顺时针旋转 45 度（重画即可）

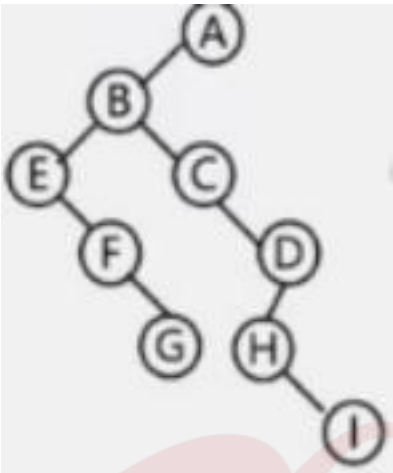


二叉树转换为树

加线：若 p 结点是双亲结点的左孩子，则将 p 的右孩子，右孩子的右孩子.....沿分支找到的所有右孩子，都与 p 的双亲连起来

抹线：抹掉原二叉树中双亲与右孩子之间的连线

调整：将结点按层次排序，形成树结构



[2011 年] 已知一棵有 2011 个结点的树，其叶结点个数为 116 ，该树对应的二叉树中无右孩子的结点个数是（ ）.

A. 115　　B. 116　　C. 1895　　D. 1896

树、森林的遍历

和二叉树的遍历对应

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

[2020 年] 已知森林 F 及与之对应的二叉树 T，若 F 的先根遍历序列是 abcdef，后根遍历序列是 badfec 则 T 的后序遍历序列是（ ）.

a b c d e f

b a d f e c

哈夫曼树

构造：每次从合成后存在的结点中选出两个权最小的进行构造二叉树，直到所有结点均在树中

哈夫曼编码：对频率高的字符赋以短编码，而对频率较低的字符则赋以较长编码，起到压缩效果

前缀编码：没有一个编码是另一个编码的前缀

1. 将每个出现的字符当作一个独立的结点，其权值为它出现的频度（或次数），构造出对应的哈夫曼树
2. 将字符的编码解释为从根至该字符的路径上边标记的序列
3. 其中边标记为 0 表示“转向左孩子”，标记为 1 表示“转向右孩子”

[2018 年] 已知字符集 {a, b, c, d, e, f}，若各字符出现的次数分别为 6, 3, 8, 2, 10, 4, 则对应字符集中各字符的哈夫曼编码可能是（ ）.

A. 00, 1011, 01, 1010, 11, 100

C. 10, 1011, 11, 0011, 00, 010

B. 00, 100, 110, 000, 0010, 01

D. 0011, 10, 11, 0010, 01, 000

[2019 年] 对 n 个互不相同的符号进行哈夫曼编码. 若生成的哈夫曼树共有 115 个结点, 则 n 的值是（ ）.

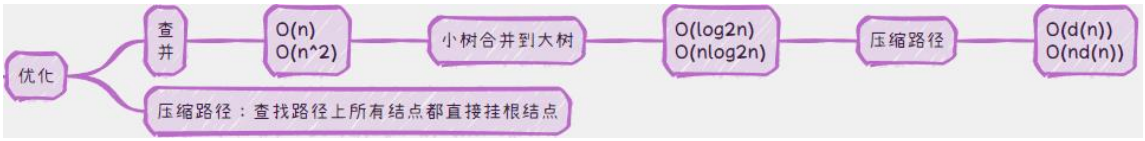
A. 56 B. 57 C. 58 D. 60

并查集

使用双亲表示法

所有表示子集合的树，构成表示全集合的森林，存放在双亲表示数组内

通常用数组元素的下标代表元素名，用根结点的下标代表子集合名



操作：

查找 x 的根：find(int x)

合并两颗树：union(int x, int y)

图

图 G 由顶点集 V 和边集 E 组成，记 G=(V，E)，其中 V(G)表示图 G 中顶点的有限非空集；E(G)表示图 G 中顶点之间的关系（边）集合

邻接：有边/弧相连的两个顶点之间的关系（无向图的边：(vi，vj)，有向图的边：<vi，vj>）

关联（依附）：边/弧与顶点之间的关系

顶点的度：与该顶点相关联的边的数目，记为 TD(v)，有向图顶点的度等于其的入度+出度

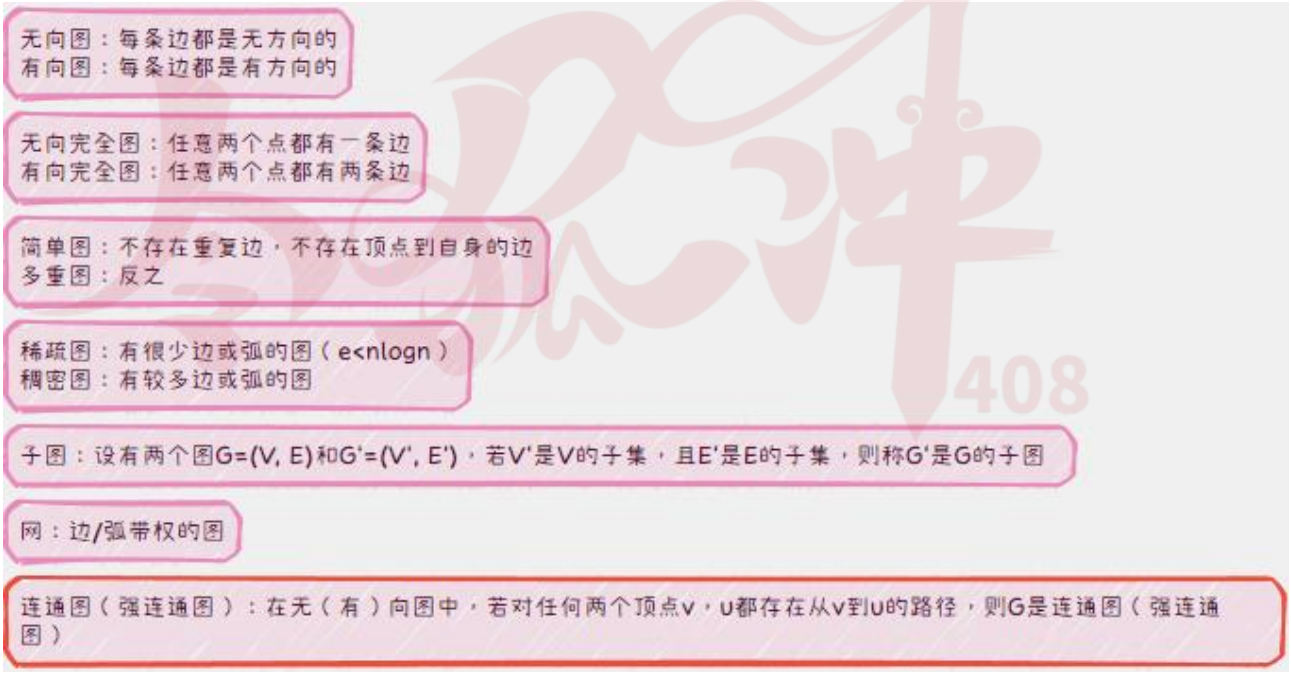
路径：连续的边构成的顶点序列

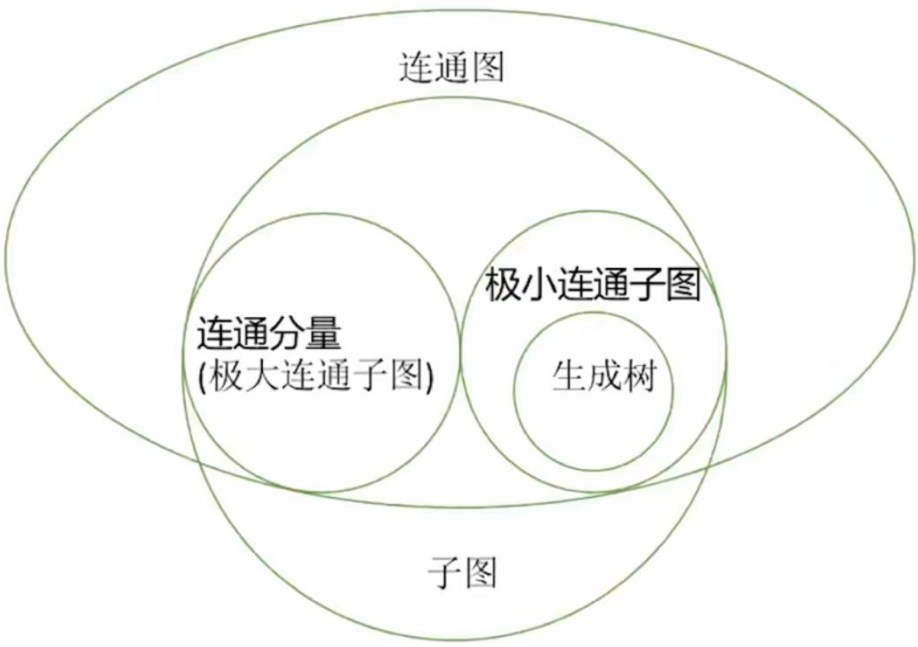
路径长度：路径上边或弧的数目/权值之和

回路（环）：第一个顶点和最后一个顶点相同的路径

简单路径：除路径起点和终点可以相同外，其余顶点均不相同的路径

简单回路（简单环）：除路径起点和终点相同外，其余顶点均不相同的路径





连通分量/极大连通子图：该子图是 G 连通子图，将 G 的任何不在该子图的顶点加入，子图不再连通

强连通分量/极大强连通子图：该子图是 G 强连通子图，将 G 的任何不在该子图的顶点加入，子图不再强连通

极小连通子图：该子图是 G 的连通子图，在该子图中删除任何一条边，子图不再连通

生成树：包含无向图 G 所有顶点的极小连通子图

生成森林：对非连通图，由各个连通分量的生成树的集合

[2010 年] 若无向图 G = (V, E) 中含有 7 个顶点，要保证图 G 在任何情况下都是连通的，则需要的边数最少是（ ）.

A. 6　　B. 15　　C. 16　　D. 21

图的存储

邻接矩阵法：用一个一位数组存储图中顶点的信息，用一个二维数组存储图中边的信息（邻接关系），二维数组即邻接矩阵

```
typedef struct {
    int edges[MaxSize][MaxSize];
    char vertices[MaxSize];
    int edge_num, vertice_num;
} Graph;
```

邻接表法：顶点用一个一维数组存储，元素包含指向第一个邻接点的指针，存储顶点和头指针的一维数组叫顶点表，每个顶点的所有邻接点构成一个单链表

```
typedef struct Arc {
    Arc* next;
    int vertice_no;
    int weight; // 权值，可选
} Arc;
typedef struct {
    int data;
    Arc* firstArc; // 该顶点所连边链表的头结点
} Vertice;
typedef struct {
    Vertice vertices[MaxSize];
    int vertice_num, arc_num;
} Graph;
```

[2013 年] 设图的邻接矩阵 A 如下所示，各顶点的度依次是（　）.

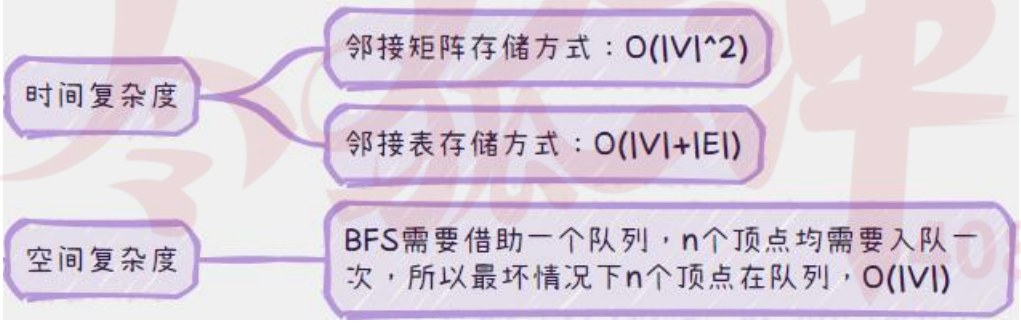
$$A=\begin{bmatrix}0&1&0&1\\0&0&1&1\\0&1&0&0\\1&0&0&0\end{bmatrix}$$

A. 1，2，1，2　　B. 2，2，1，1　　C. 3，4，2，3　　D. 4，4，2，2

图的广度优先遍历

类似于树的层序遍历，多了标记数组，用于确定已访问的结点

```
void BFS(Graph& g, int start) {
    bool* visited = new bool[g.vertice_num];// 记录访问过的节点
    visited[start] = true;
    queue<int> q;
    q.push(start);
    while (!q.empty()) {
        int cur = q.front();
        visit(cur); // 具体的访问操作
        q.pop();`
        for (neighbor : g.vertices[cur]) {
            if (!visited[neighbor]) {
                visit(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}
```

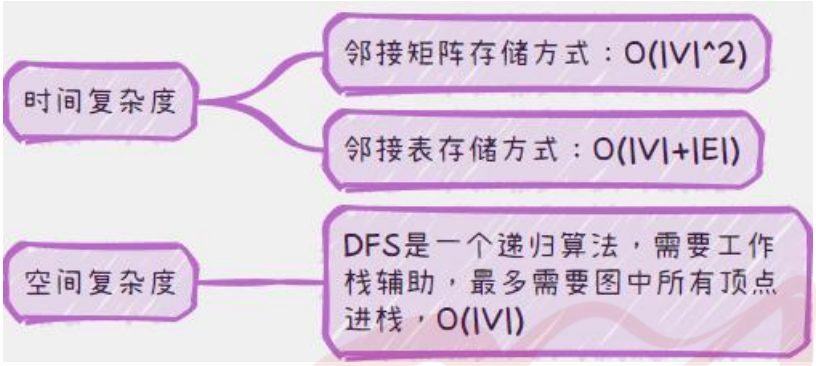


图的深度优先遍历

邻接表法：类似于树的先序遍历，搜索策略是尽可能深地搜索一个图

```
void DFS(Graph& g, int cur, bool* visited) {
    visited[cur] = true;
    visit(cur);
    for (neighbor : graph.vertices[cur])
        if (!visited[neighbor])
            DFS(g, neighbor, visited);
}
```

```
for (neighbor : g.vertices[cur])
//对于邻接矩阵:
for (int neighbor = 0;
        neighbor < g.vertice_num; neighbor++)
//对于邻接表
for (Arc* neighbor = g.vertices[cur].firstArc;
        neighbor; neighbor = neighbor→next)
```



[2015 年] 设有向图 G =(V, E)，顶点集 V ={V0, V1, V2, V3}，边集 E ={<v0, v1>, <v0, v2>, <v0, v3>, <v1, v3>}。若从顶点 V0 开始对图进行深度优先遍历，则可能得到的不同遍历序列个数是（ ）.

A. 2 B. 3 C. 4 D. 5

最小生成树

所有顶点均由边连接在一起，但不存在回路的图

最小生成树：**权值之和最小的那棵生成树**，**不唯一**，但是权值和唯一且最小，边数=顶点数-1

Prim 算法：每次将代价最小的新顶点加入生成树

开始时从图中任取一个顶点加入树 T

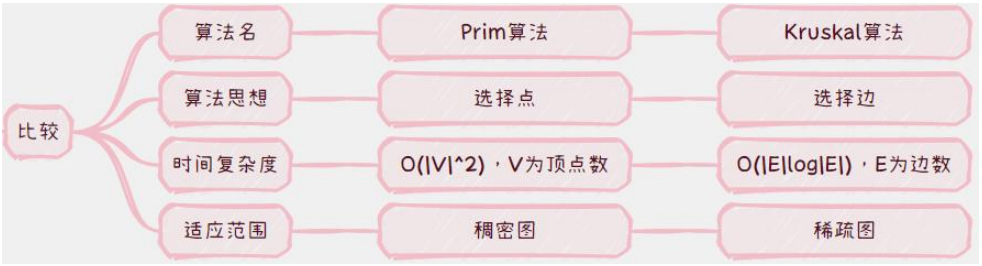
之后选择一个与当前 T 中顶点集合距离最近的顶点，将该点和对应边加入 T，每次操作后 T 中顶点数和边数都+1。以此类推，当所有点加入 T，必然有 n-1 条边，即 T 就是最小生成树

Kruskal 算法：每次选一条权值最小的边（边按权值排序），使其连通（用并查集判断并实现）

开始时为只有 n 个顶点而无边的非连通图 T={V, {}}，每个顶点自成一个连通分量

然后按照边的权值由小到大，不断选取当前未被选取过且权值最小的边，然后按

照边的权值由小到大，不断选取当前未被选取过且权值最小的边，以此类推，直到 T 中所有顶点都在一个连通分量上



最短路径

在有向网中 A 点（源点）到达 B 点（终点）的多条路径中，找到一条各边权值之和最小的路径

Dijkstra 算法（两点间最短路径）：维护一个最短路径数组，每次选取最短的顶点加入，更新加入后的最短路径，直到所有顶点都访问

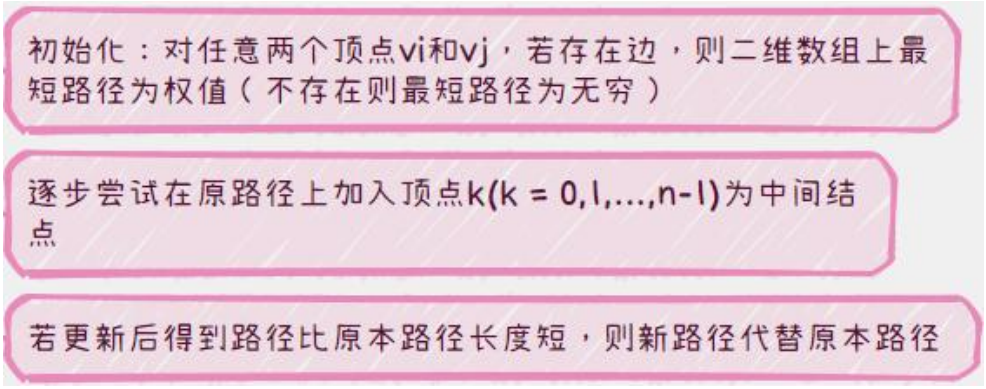
s[]记录是否访问

dist[]记录源地到各点最短路径

path[]记录前驱结点

Floyd 算法（各顶点之间最短路径）：维护一个各顶点间最短路径二维数组，不断试探加入中间结点，是否缩短距离（三重循环）

允许图中有带父权值的边，但不允许有包含带负权值的边组成回路



有向无环图描述表达式

流程：

操作数在最下层排成一排

按生效顺序，加入运算符（分层）

从底向上检查同层能否合并

[2019 年] 用有向无环图描述表达式 (x + y)((x + y) / x)，需要的顶点个数至少是（ ）.

A. 5 B. 6 C. 8 D. 9

拓扑排序

拓扑序列：对图中所有的顶点，如果存在一条从顶点 A 到顶点 B 的路径，那么在排序中顶点 A 出现在顶点 B 的前面

过程：

从 AOV 网中选择一个没有前驱的顶点并输出

从网中删除该顶点和所有以它为起点的有向边

重复前两个步骤，直到当前的 AOV 网为空或当前网中不存在无前驱的顶点为止（后者说明有向图中存在环）

若一个顶点有多个直接后继，则拓扑排序通常不唯一（若每个顶点有唯一前驱后继，则唯一）

关键路径

关键路径：从源点到汇点的所有路径中，具有最大路径长度的路径

关键活动：关键路径上的活动

事件的最早发生时间 ve()

事件的最迟发生时间 vl()

活动的最早发生时间 e()

活动的最迟发生时间 l()

活动的时间余量 d()

关键路径上的所有活动都是关键活动，是决定整个工程的关键因素，因此可通过加快关键活动来缩短整个工程的工期

不能任意缩短关键活动，因为一旦缩短到一定程度，该关键活动可能变成非关键

活动

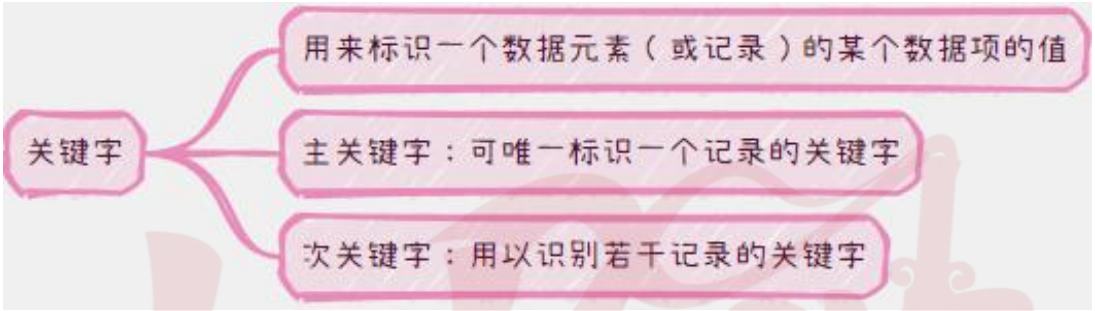
网中的关键路径不唯一，只有加快那些包含在所有关键路径上的关键活动才能达到缩短工期的目的

若关键活动耗时增加，则整个工程的工期增长

查找

查找：根据给定值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）（查找成功/查找不成功）

查找表：是由同一类型的数据元素（或记录）构成的集合（数据元素之间关系松散，应用灵活方便）



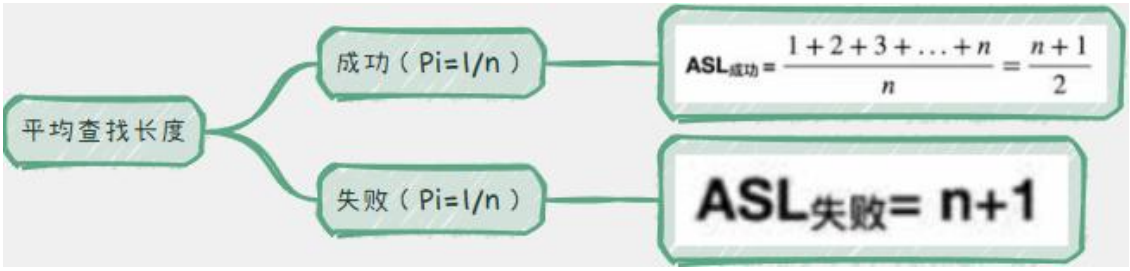
平均查找长度：$ASL = \sum_{i=1}^{n} p_i c_i$

线性表的查找——顺序查找

从线性表的一端开始，逐个检查关键字是否满足条件

若满足则查找成功，返回元素位置

若已经查到表的另一端，且还没有找到符合条件的元素，则返回查找失败的信息



线性表的查找——二分查找

有序的线性表中，每次将待查记录所在区间缩小一半

在[low, high]之间找目标关键字，将给定 key 值与表中中间位置 mid 比较，若

相等则查找成功

不等，则根据 mid 所指元素与 key 值大小比较，调整 low 或 high，缩小边界范围，不断重复

若 low>high，则查找失败

```
int binarySearch(int A[], int len, int target) {
    int low = 0, high = len - 1;
    while (low ≤ high) {
        int mid = (low + high) / 2;
        if (A[mid] == target) return mid;
        if (A[mid] > target) high = mid - 1;
        else low = mid + 1;
    }
    return -1;
}
```

线性表的查找——分块查找

将查找表分为若干子块，**块内的元素可以无序，但块间有序**，一个块的最大关键字小于下一个块中所有记录的关键字

再建立一个索引表，索引表中的每个元素含有各块的最大关键字和各块中第一个元素的地址，索引表按关键字排序

查找过程：在索引表中确定记录所在块，在块内顺序查找

查找成功长度：将长度为 n 的查找表均匀分为 b 块，每块 s 个记录，ASL=ASLb+ASLs

[2015 年] 下列选项中，不能构成折半查找中关键字比较序列的是（　）.

A. 500, 200, 450, 180

C. 180, 500, 200, 450

B. 500, 450, 200, 180

D. 180, 200, 500, 450

二叉树

二叉搜索树 BST

二叉排序树：左子树结点值（若左子树非空）<根结点值<右子树结点值（若右子树非空），其左右子树本树又各是一棵二叉排序树，

中序遍历非空的二叉排序树所得到的数据元素序列是一个按关键字排序的递增有序序列

插入：若关键字小于根结点值，插入左子树，若关键字大于根结点值，插入右子树

删除：

若被删除的结点是叶结点，直接删；

若结点只有一棵左子树或右子树，让子树成为该结点的父结点的子树，代替该结点；

若结点有左、右两棵子树

平均查找长度 ASL=查找树每层节点个数*对应深度之和

[2011 年]对下列关键字序列，不可能构成二叉排序树中一条查找路径的是（　）

A. 95，22，91，24，94，71

C. 21，89，77，29，36，38

B. 92，20，91，34，88，35

D. 12，25，71，68，33，34

平衡二叉树 AVL

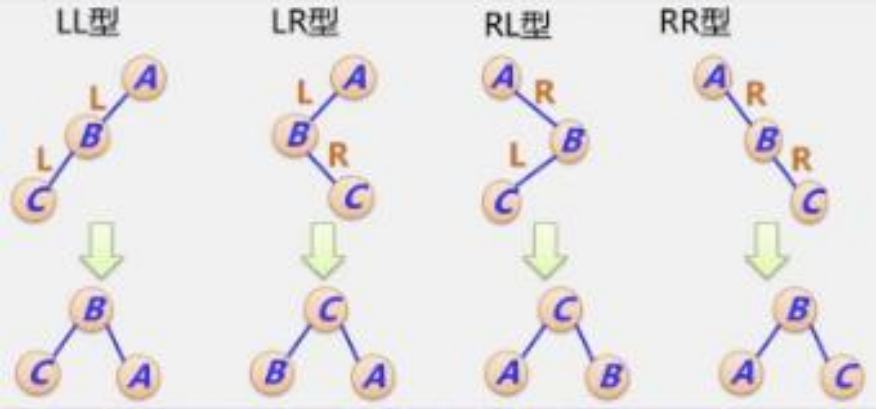
平衡二叉树是二叉排序树，且左子树与右子树的高度之差的绝对值小于等于 1，左子树和右子树也是平衡二叉排序树

平衡因子 = 结点左子树的高度-结点右子树的高度

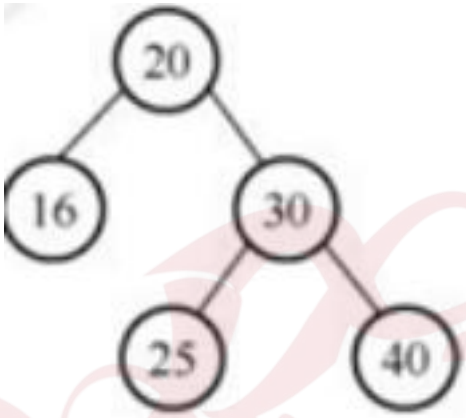
调整：当平衡二叉树插入（删除）一个结点时，首先检查其插入路径上的结点是否因此次操作导致不平衡，若不止一个失衡结点时，从最小失衡子树的根结点开始平衡，直到所有结点都满足平衡二叉树特性，根据操作位置分为：LL 右单旋、RR 左单旋、LR 先左后右旋转、RL 先右后左旋转。

原则 1：降低高度

原则 2：找出三个结点，保持二叉排序树性质



[2021 年] 给定平衡二叉树如下图所示，插入关键字 23 后，根中的关键字是（ ）



A. 16　　B. 20　　C. 23　　D. 25

红黑树

红黑树是二叉排序树，且满足：

根结点是黑色的（每个结点不是黑，就是红）

叶结点（外部结点/NULL 结点/失败结点）均是黑色

不存在两个相邻的红结点（即红结点的父结点和孩子结点均是黑色）

对每个结点，从该结点到任一叶结点的简单路径上，所含黑结点的数目相同

性质：根到叶结点最长路径不大于最短路径（全黑）2 倍，n 个结点高度 h<=2log2(n+1)

黑高：从某结点出发（不包含该结点）到达任一空叶结点路径上黑色结点数

若根结点黑高为 h，内部结点数（关键字）最少有 2h-1 个

B 树

又称多路平衡查找树（所有结点的平衡因子均为 0），B 树中所有结点的孩子个

数的最大值称为 B 树的阶，用 m 表示

树中每个结点至多有 m 棵子树，至多有 m-1 个关键字

关键字个数：⌈ m/2⌉ -1<=n<=m-1（根结点：1<=n<=m-1）

子树个数：⌈ m/2⌉ <= n <= m（根结点：2 <= n <= m）

所有的叶结点（失败节点）都出现在同一层次上，且不带信息

B 树的高度（磁盘存取次数）

对任意一颗包含 n 个关键字、高度为 h、阶数为 m 的 B 树，h > logm(n+1)

查找：

在 B 树上查找到某个结点后，先在有序表中进行查找

若找到则查找成功，否则按照对应的指针信息到所指的子树中去查找

查找到叶结点时（对应指针为空指针），则说明树中没有对应的关键字，查找失败

插入：

定位：利用查找算法，找出插入该关键字的最底层中的某个非叶结点

在插入后，若结点关键字个数大于等于 m，则从中间位置（⌈ m/2⌉ ）将关键字分为两部分

左部分留在原结点、右部分放到新的结点、中间位置的结点插入原结点的父结点

B 树——删除

[2018 年] 高度为 5 的 3 阶 B 树含有的关键字个数至少是（　）.

A. 15　　B. 31　　C. 62　　D. 242

B+树

每个分支结点最多 m 棵子树

非叶根结点最少有 2 棵子树，其他每个结点分支：⌈ m/2⌉ <= n <= m

结点的子树与关键字个数相等

所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排序，并且相邻叶结点按大小顺序相互链接起来

所有分支结点（可视为索引的索引）中仅包含它的各个结点（即下一级的索引块）中关键字的最大值及指向其子结点的指针

[2017 年] 下列应用中，适合使用 B+树的是（　）.

A. 编译器中的词法分析

B. 关系数据库系统中的索引

C. 网络中的路由表快速查找

D. 操作系统的磁盘空闲块管理

B 树和 B+树

对比项	m阶B树	m阶B+树
关键字个数范围	⌈ m/2⌉ -1 ≤ n ≤ m-1（根结点：1 ≤ n ≤ m-1）	⌈ m/2⌉ ≤ n ≤ m（根结点：1 ≤ n ≤ m）
关键字与子树关系	n个关键字对应 n+1棵子树	n个关键字对应 n棵子树
记录信息存储位置	所有结点均可能包含记录信息	仅最下层叶子结点包含记录信息（减少磁盘访问次数）
查找特性	✕ 不支持顺序查找 ✓ 查找成功可能出现在任何一层⏱ 速度不稳定	✓ 支持顺序查找（叶子结点链表串联） ✓ 查找成功/失败均在最后一层⏱ 速度稳定

散列查找

散列表：根据给定的关键字来计算出关键字在表中的地址的数据结构。建立关键字和存储地址之间的一种直接映射关系。

散列函数：一个把查找表中的关键字映射成该关键字对应的地址的函数，记为 Hash(key)=Addr。

散列函数可能会把两个或两个以上的不同关键字映射到同一地址，称这种情况为“冲突”，这些发生碰撞的不同关键字称为同义词。

构造散列函数

1）散列函数的定义域必须包含全部需要存储的关键字，而值域的范围则依赖于散列表的大小或地址范围。

2）散列函数计算出来的地址应该能等概率、均匀地分布在整个地址空间，从而减少冲突的发生。

3）散列函数应尽量简单，能够在较短的时间内就计算出任一关键字对应的散列地址。

散列查找——常用 Hash 函数的构造方法

开放定址法：直接取关键字的某个线性函数值为散列地址，散列函数为 H(key)=a×key+b。式中，a 和 b 是常数。这种方法计算最简单，并且不会产生冲突

除留余数法：假定散列表表长为 m，取一个不大于 m 但最接近或等于 m 的质数 p。散列函数为 H(key)=key % p 。关键是选好 p，使得每一个关键字通过该函数转换后等概率地映射到散列空间上的任一地址，从而尽可能减少冲突的可能性

数字分析法：设关键字是 r 进制数（如十进制数），而 r 个数码在各位上出现的频率不一定相同，可能在某些位上分布均匀些，每种数码出现的机会均等；而在某些位上分布不均匀，只有某几种数码经常出现，则应选取数码分布较为均匀的若干位作为散列地址。这种方法适合于已知的关键字集合

散列查找——常用 Hash 函数的冲突处理办法

1.开放定址法：将冲突的 Hash 地址作为自变量，通过冲突解决函数得到一个新的空闲的 Hash 地址

1）线性探测法：冲突发生时，顺序在表中找出一个空闲单元（当表未填满时一定能找到一个空闲单元）或查遍全表。

2）平方探测法：设发生冲突的地址为 d，平方探测法得到的新的地址序列为 d+12，d-12，d+22，d-22..... 平方探测法是一种较好的处理冲突的方法，可以避免出现“堆积”问题，它的缺点是不能探测到散列表上的所有单元，但至少能探测到一半单元。

3）再散列法：又称为双散列法。需要使用两个散列函数，当通过第一个散列函数 H(Key)得到的地址发生冲突时，则利用第二个散列函数 Hash2(Key)计算该关键字的地址增量。

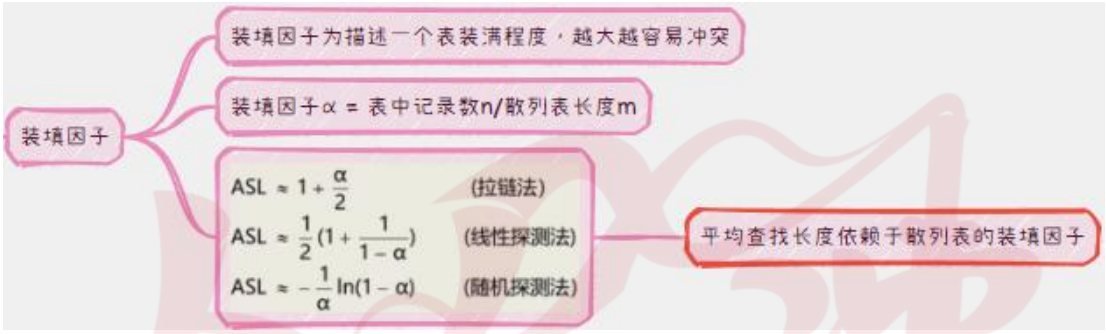
4）伪随机序列法：当发生地址冲突时，地址增量为伪随机数序列，称为伪随机序列法。

α 越大，表示装填的记录越“满”，发生冲突的可能性就越大，反之发生冲突的可能性越小

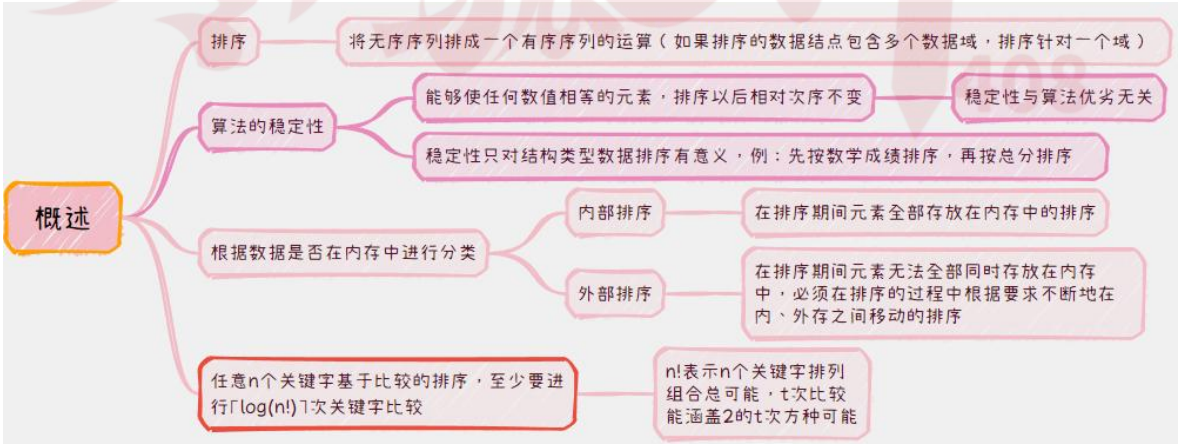
2. 拉链法：对于不同的关键字可能会通过散列函数映射到同一地址，为了避免非同义词发生冲突，可以把所有的同义词存储在一个线性链表中，这个线性链表由其散列地址唯一标识。拉链法适用于经常进行插入和删除的情况。

散列查找——查找

检测由散列函数形成的地址上是否有记录，若无记录则失败；
若有记录比较关键字值，若相等则查找成功，否则散列函数更新增量值，重复执行
求查找失败的 ASL 时，失败位置也计数；若 H(key) = k%7，则失败要算 0, 1, 2, 3, 4, 5, 6
查找效率：散列函数、冲突处理、装填因子



排序



插入类排序

直接插入排序：首先以一个元素为有序的序列，然后将后面的元素依次插入到有序的序列中合适的位置直到所有元素都插入有序序列。时间复杂度为 O(n^2)，是稳定的。

折半插入排序：将比较和移动这两个操作分离出来，也就是先利用折半查找找到插入的位置，然后一次性移动元素，再插入该元素。时间复杂度为 O(n^2)，是

稳定的。

希尔排序：本质上还是插入排序，但是把待排序序列分成几个子序列，再分别对这几个子序列进行直接插入排序。时间复杂度约为 O(n1.3) ，在最坏情况下希尔排序的时间复杂度为 O(n2)，空间复杂度为 O(1)，不稳定，由于不同的增量可能就会把相等的关键字划分到两个直接插入排序中进行排序， 可能就会造成相对顺序变化。

[2014 年] 用希尔排序算法对一个数据序列进行排序时，若第一趟排序结果为 9，1，4,13，7，8，20，23,15,则该趟排序采用的增量（间隔）可能是（ ）

A. 2　　B. 3　　C. 4　　D. 5

交换类排序——冒泡排序

一趟冒泡：假设待排序表长为 n，从后往前（或从前往后）两两比较相邻元素的值，若为逆序（即 A[i-1]>A[i]），则交换它们，直到序列比较完。结果将最小的元素交换到待排序列的第一个位置。

下一趟冒泡时，前一趟确定的最小元素不再参与比较，待排序列减少一个元素，每趟冒泡的结果把序列中的最小元素放到了序列的最终位置

这样最多做 n-1 趟冒泡就能把所有元素排好序。

空间复杂度为 O(1)，时间复杂度为 O(n2)，是稳定的。

交换类排序——快速排序

快速排序是一种基于分治法的排序方法。 每一趟快排选择序列中任一个元素作为枢轴(pivot)(通常选第一个元素)，将序列中比枢轴小的元素都移到枢轴前边，比枢轴大的元素都移到枢轴后边。

时间复杂度： 最好情况下时间复杂度为 O(nlogn) ，待排序序列越无序，算法效率越高。 最坏情况下时间复杂度为 O(n2)，待排序序列越有序，算法效率越低。

空间复杂度： 由于快速排序是递归的，需要借助一个递归工作栈来保存每一层递归调用的必要信息，其容量应与递归调用的最大深度一致。

最好情况下为⌈ log2(n+1)⌉(每次 partition 都很均匀)递归树的深度 O(logn)

最坏情况下，因为要进行 n-1 次递归调用，所以栈的深度为 O(n)；

稳定性：不稳定

[2014 年] 下列选项中，不可能是快速排序第二趟排序结果的是（ ）.

A. 2, 3, 5, 4, 6, 7, 9

B. 2, 7, 5, 6, 4, 3, 9

C. 3, 2, 5, 4, 7, 6, 9

D. 4, 2, 3, 5, 7, 6, 9

[2023 年] 使用快速排序算法对数据进行升序排序，若经过一次划分后得到的数据序列是

68, 11, 70, 23, 80, 77, 48, 81, 93, 88，则该次划分的枢轴是（ ）.

A. 11 B. 70 C. 80 D. 81

交换类排序——快速排序

```
int partition(int nums[], int low, int high) {
    int i = low, j = high;
    int pivot = nums[low];
    while (i < j) {
        while (i < j && nums[j] ⩾ pivot) --j;
        while (i < j && nums[i] ⩽ pivot) ++i;
        if (i < j) swap(nums[i], nums[j]);
    }
    // 将基准值放到正确位置
    swap(nums[low], nums[i]);
    return i;
}

void quickSort(int arr[], int left, int right) {
    if (left ⩾ right) return;
    int pivotPos = partition(arr, left, right);
    quickSort(arr, left, pivotPos - 1);
    quickSort(arr, pivotPos + 1, right);
}
```

选择类排序——简单选择排序

每一趟（如第 i 躺）在后面 n-i+1（i=1,2,...,n-1）个待排序元素中选取关键字最小的元素，作为有序子序列的第 i 个元素，直到第 n-1 趟完成

空间复杂度：需要额外的存储空间仅为交换元素时借助的中间变量，所以空间复杂度是 O(1)

时间复杂度： O(n^2)

稳定性：不稳定

选择类排序——堆排序

堆：一棵完全二叉树，而且满足任何一个非叶结点的值都不大于(或不小于)其左右孩子结点的值。

如果是每个结点的值都不小于它的左右孩子结点的值，则称为大顶堆。

如果是每个结点的值都不大于它的左右孩子结点的值，则称为小顶堆。

堆排序：每次将无序序列调节成一个堆，然后从堆中选择堆顶元素的值，这个值加入有序序列，无序序列减少一个，再反复调节无序序列，直到所有关键字都加入到有序序列。

时间复杂度： 堆排序的总时间可以分为①建堆部分 + ②n-1 次向下调整堆

即 O(n)+O(nlog2n) = O(nlogn)

稳定性：不稳定

[2015 年] 已知小根堆为 8, 15, 10, 21, 34, 16, 12，删除关键字 8 之后需重建堆,在此过程中，关键字之间的比较次数是（ ）.

二路归并排序

假定待排序表含有 n 个记录，则可以看成是 n 个有序的子表，每个子表长度为 1，然后两两归并，得到 ⌈ n/2⌉ 个长度为 2 或 1 的有序表；再两两归并，如此重复，直到合并成一个长度为 n 的有序表为止，这种排序方法称为 2-路归并排序。

例如：49 38 65 97 76 13 27

①首先将整个序列的每个关键字看成一个单独的有序的子序列

②两两归并，49 和 38 并成{38 49} ，65 和 97 并成{65 97}，76 和 13 并成{13 76}，27 没有归并对象

③两两归并，{38 49}和{65 97}归并成{38 49 65 97}，{13,76}和 27 归并成{13 27 76}

④两两归并，{38 49 65 97}和{13 27 76}归并成{13 27 38 49 65 76 97}

时间复杂度：O(nlogn) 空间复杂度:需要将待排序序列转存到数组，为 O(n)

稳定性：稳定

[2013 年] 已知两个长度分别为 m 和 n 的升序链表，若将它们合并为长度为 m + n 的一个降序链表，则最坏情况下的时间复杂度是（　）.

A. O(n)

B. O(mn)

C. O(min(m, n))

D. O(max(m, n))

基数排序

不是基于比较排序，而是采用多关键字排序思想（即基于关键字各位的大小进行排序的），借助“分配”和“收集”两种操作对单逻辑关键字进行排序。基数排序又分为最高位优先（MSD）排序和最低位优先（LSD）排序。

空间复杂度：需要开辟关键字基的个数个队列，所以空间复杂度为 O(r)

时间复杂度：需要进行关键字位数 d 次"分配"和"收集"，一次"分配"需要将 n 个关键字放进各个队列中，一次"收集"需要将 r 个桶都收集一遍。所以一次"分配"和一次"收集"时间复杂度为 O(n+r)。d 次就需要 O(d(n+r))的时间复杂度。

稳定性：由于是队列，先进先出的性质，所以在分配的时候是按照先后顺序分配，也就是稳定的，所以收集的时候也是保持稳定的。即基数排序是稳定的排序算法。

各种内部排序的比较

类别	排序方法	最好情况时间复杂度	最坏情况时间复杂度	平均情况时间复杂度	空间复杂度（辅助存储）	稳定性
插入排序	直接插入排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	希尔排序	O(n^1.3)	$O(n^2)$	-	O(1)	不稳定
交换排序	冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	O(nlogn)	$O(n^2)$	O(nlogn)	O(nlogn)	不稳定
选择排序	直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
	堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定
-	归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
-	基数排序*	O(n+m)	O(k*(n+m))	O(k*(n+m))	O(n+m)	稳定

外部排序

需要将待排序的记录存储在外存上，排序时再把数据一部分一部分的调入内存进行排序。在排序过程中需要多次进行内存和外存之间的交换，对外存文件中的记录进行排序后的结果仍然被放到原有文件中。

得到初始的归并段：使用置换选择排序，解决排序段放入内存的问题
减少多个归并段的归并次数：采用最佳归并树（k 叉哈夫曼树），最少的归并次数（I/O 次数）

快速得到最小的关键字：采用败者树，减少比较次数
[2019 年] 设外存上有 120 个初始归并段，进行 12 路归并时，为实现最佳归并，需要补充的虚段个数是（　）.
A. 1　　B. 2　　C. 3　　D. 4