1. ## 如何判断一个点在三角形内部

   使用三角形面积公式（坐标的那个）

   然后使用面积来计算，内部时，面积等于三个小面积之和

2. ## 如何判断一个数是否为偶数

   判断末尾是否是0

3. ## 去除某个数的最右边的1

   n&（n-1）

4. ## 格雷码生成

   生成n+1位格雷码，就是n位格雷码加前缀1，倒序

   n^(n/2)

5. ## 二叉树的遍历

   1. 前序遍历

      去除栈顶元素，加入结果集

      压入右子树，压入左子树

   2. 中序遍历

      迭代压入左子树

      去除栈顶，加入结果，如果有右子树

      进入右子树，迭代压入左子树

   3. 后序遍历

      取出栈顶元素，加入结果集

      压入左子树，压入右子树

      反转

6. ## hash冲突的解决方法

   一说：hash碰撞是hash结果相同

   而说：hash冲突是hash结果放到了一个桶中

   1. 开放地址法

      开放地执法有一个公式:Hi=(H(key)+di) MOD m i=1,2,...,k(k<=m-1)
      其中，m为哈希表的表长。di 是产生冲突的时候的增量序列。如果di值可能为1,2,3,...m-1，称线性探测再散列。
      如果di取1，则每次冲突之后，向后移动1个位置.如果di取值可能为1,-1,2,-2,4,-4,9,-9,16,-16,...k*k*,-*k*k(k<=m/2)，称二次探测再散列。
      如果di取值可能为伪随机数列。称伪随机探测再散列。

   2. 再hash（使用不同的hash）

   3. 链地址法（拉链法）

4. 建立公共溢出区

7. 红黑树和AVL树的定义特点和区别(时间复杂度)

1. 平衡二叉树：左右子树高度相差不超过1
2. 排序树：中序从小到大
3. AVL：特殊的二叉排序树，树的左右子树都是二叉平衡树
4. 红黑树：特殊的二叉查找树，在每一个节点增设一个存储位表示节点的颜色。通过对根-叶子结点的颜色进行限制。

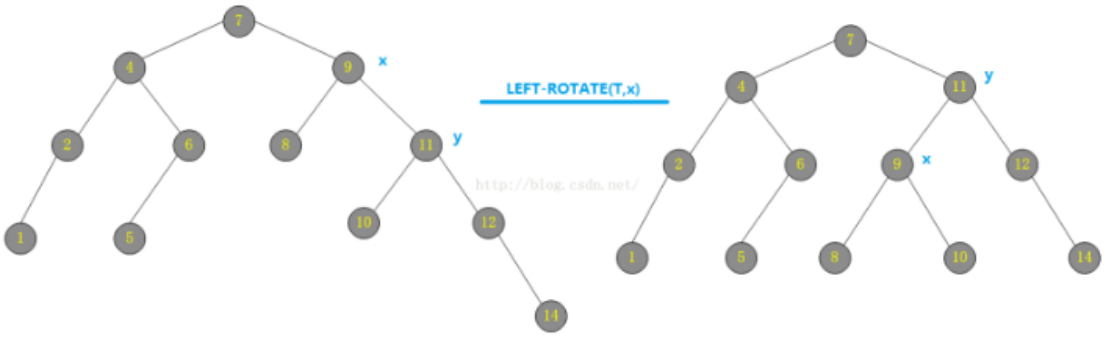
   特点为：没有一条路径会比其他路径长出2倍。是一种弱平衡二叉树。但是红黑树的旋转次数少，对于搜索插入删除较多时使用

   性质：

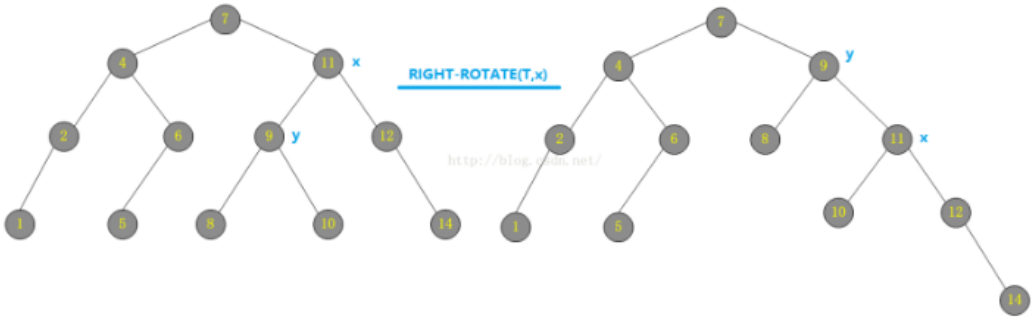
   1. 每个点非红即黑
   2. 根节点为黑色
   3. 每个叶节点为黑色（树尾端的null)
   4. 如果有一个节点为红，子树为黑
   5. 任意节点，到叶子树的每条路径黑节点数相同
   6. 插入最多两次旋转，删除最多三次旋转
   7. 红黑树树根的黑高度至少h/2
   8. 一个n个内节点的红黑树的高度之多2long(n+1)
5. 红黑树操作的时间复杂度

   除旋转外都是O(logn)

   二叉查找树的时间复杂度是(O(h))
6. 红黑树的旋转：保持二叉搜索树、红黑树的性质，需要进行旋转
   1. 左旋：新根节点的左子树变为左节点的右子树



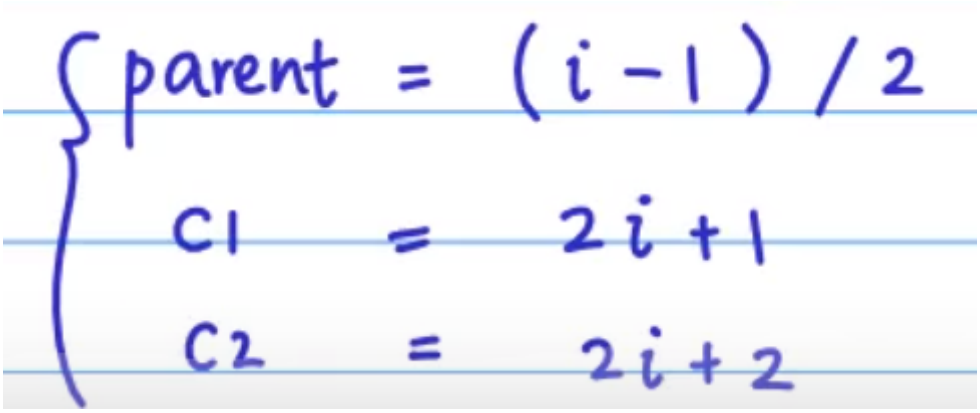
   2. 右旋：新节点右子树变为右节点的左子树



8. **快速幂模**

```
if(b&1)
{
    sum = sum*a%c;
    b--;
}
b/=2;
a = a*a%c;
```

9. **堆排序**

10. 


```
void heapify(int tree[], int n,int i)
{
    if (i >= n)
        return;
    int c1 = child1(i);
    int c2 = child2(i);
    int maxs = i;
    if (c1<n && tree[c1] > tree[maxs])
        maxs = c1;
    if (c2<n && tree[c2] > tree[maxs])
        maxs = c2;
    if (maxs != i)
    {
        swap(tree[maxs], tree[i]);
        heapify(tree, n, maxs);
    }
}
```

11. **整棵树找最大值：堆重构**

    1. 如果每个节点都是堆，那么heapify（0）
    2. 如果不是，从最后一个节点n-1的parent开始，到第一个进行heapify
    3. 堆排序：将头节点和尾节点交换，然后heapify头部，末尾-1

```
int parent(int i)
{
```

```
    return (i - 1) / 2;
}
int child1(int i)
{
    return 2 * i + 1;
}
int child2(int i)
{
    return 2 * i + 2;
}
void heapify(int tree[], int n,int i)
{
    if (i >= n)
        return;
    int c1 = child1(i);
    int c2 = child2(i);
    int maxs = i;
    if (c1<n && tree[c1] > tree[maxs])
        maxs = c1;
    if (c2<n && tree[c2] > tree[maxs])
        maxs = c2;
    if (maxs != i)
    {
        swap(tree[maxs], tree[i]);
        heapify(tree, n, maxs);
    }
}
void heap_sort(int tree[], int n)
{
    for (int i = n - 1; i >= 0; i--)
    {
        swap(tree[0], tree[i]);
        heapify(tree, i, 0);
    }
}
```

建堆算法：O(n)

堆排序：O(nlogn)

12. **快排**

```cpp
void quick_sort(vector<int>&myvec, int left, int right)
{
    if (right <= left || right >= myvec.size())
        return;
    int lp = left;
    int rp = right;
    int base = myvec[left];
    while (lp < rp)
    {
        while (lp < rp && myvec[rp] >= base)
            rp--;
        while (lp < rp && myvec[lp] <= base)
            lp++;
        if (lp < rp)
            swap(myvec[lp], myvec[rp]);
    }
    myvec[left] = myvec[lp];
    myvec[lp] = base;
    quick_sort(myvec, left, lp - 1);
    quick_sort(myvec, lp + 1, right);
}
```

快排快的原因：

大幅度减少了比较和交换的次数，因为从基准数切开的数组，其中一个再也不会和另外数组的元素进行比较

快排两种最差情况：

1）数组已经是正序排过序的。

2）数组已经是倒序排过序的。

3）所有的元素都相同（1、2的特殊情况）

改善：左中右的三个数的中位数作为基准点；

13. ## 冒泡排序

```cpp
vector<int> bubble_sort(vector<int> myvec)
{
    bool sorted = false;
    int n = myvec.size();
    while (!sorted)
    {
        sorted = true;
        for (int i = 1; i < n; i++)
        {
            if (myvec[i - 1] > myvec[i])
            {
                sorted = false;
                swap(myvec[i - 1], myvec[i]);
            }
        }
        n = n - 1;
    }
    return myvec;
}
```

14. ## 插入排序

```cpp
vector<int> insert_sort(vector<int> myvec)
{
    for (int i = 1; i < myvec.size(); i++)
    {
        for (int j = i - 1; j >= 0 && myvec[j+1]<myvec[j] ; j--)
        {
            swap(myvec[j + 1], myvec[j]);
        }
    }
    return myvec;
}
```

15. ## 交换距离等于1可满足稳定性

    插入排序，冒泡排序

16. ## TOP(K)最大

    1. 如果内存够用：直接全部排序
    2. 快速排序的变形：找到第K个，快排过程中，如果当前基准下表<K，那么递归后面的，如果基准下表大于K，递归前面的
    3. 最小堆：局部淘汰法

       读取前K个元素建立最小堆，如果后面的数据<=堆顶，则比较下一个，否则删除堆顶插入堆中，重新平衡
    4. 分治法

       全部数据分为N份，每份K个

找到每堆数据的最大K个，此时剩下N*K个数据，如果不能容纳N*K个，继续分治，知道全部容纳，然后比较出最大K个

5. hash法：如果重复度很高，使用hash法去重，然后读入内存找到

17. 各种排序算法

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(nlog_2n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定	较复杂

1. 插入排序（稳定排序）

   初始有序数组元素个数为1，然后从第二个元素开始插入到有序数组中，对于每一次插入操作，从后往前遍历查找可以插入的位置

2. 希尔排序

   希尔排序：先将整个待排序记录分割成若干子序列，然后分别进行直接插入排序，待整个序列中的记录基本有序时，在对全体记录进行一次直接插入排序。其子序列的构成不是简单的逐段分割，而是将每隔某个增量的记录组成一个子序列。希尔排序时间复杂度与增量序列的选取有关，其最后一个值必须为1.

3. 归并排序

   分治法：对于m个元素的待排序序列，两两合并直到长度最长

4. 冒泡排序：遍历数组，交换逆序数组对，每次迭代找到当前最大、最小值

5. 快速排序：通过一趟排序将排序数据分成两个独立的部分 -- 最快O(n^2)

6. 选择排序：找到最大/最小值交换

7. 堆排序

   近似完全二叉树

   1. 建堆
   2. 不断交换堆顶、堆尾，然后对堆顶重排，最后实现有序

8. 桶排序
   设置定量数组作为桶，将记录放入桶中，对不是空的桶进行排序，最后放回序列

18. hash

hash表的实现主要在构造hash函数和处理hash冲突

1. 常见的hash函数：直接地址、平方取中、除留余数
2. 处理hash冲突：开放定位法，再hash法，链地址法，建立公共溢出区

再STL中，使用质数设计hash桶长度，将28个指数计算，提供函数，找到最接近的某个质数

3. C++的rehash

   hash表中有一个负载因子，当loadFactor<=1时，期望复杂度为O(1)，当负载因子为1时，需要进行rehash，类似于vector扩容，开辟两倍空间，同时赋值桶中元素

4. 为什么hash桶个数为质数，最大程度减少冲突概率，使数据分布均匀

5. hash冲突：当关键字集合很大时，关键字不同的元素可能映射到同一地址上
    1. 开放定址法： 当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。

       二次探测法：相对于线性探测，从每次位移一位到位移2^(i)位
    2. 再hash法
    3. 链地址法
    4. 建立公共溢出区

19. git

1. merge和rebase的区别

   Merge会自动根据两个分支的共同祖先和两个分支的最新提交 进行一个三方合并，然后将合并中修改的内容生成一个新的 commit，即merge合并两个分支并生成一个新的提交,并且仍然后保存原来分支的commit记录

   Rebase会从两个分支的共同祖先开始提取当前分支上的修改，然后将当前分支上的所有修改合并到目标分支的最新提交后面，如果提取的修改有多个，那git将依次应用到最新的提交后面。
   Rebase后只剩下一个分支的commit记录

20. 指针和数组

const char * arr = "123";

//字符串123保存在常量区，const本来是修饰arr指向的值不能通过arr去修改，但是字符串“123”在常量区，本来就不能改变，所以加不加const效果都一样

char * brr = "123";

//字符串123保存在常量区，这个arr指针指向的是同一个位置，同样不能通过brr去修改"123"的值

const char crr[] = "123";

//这里123本来是在栈上的，但是编译器可能会做某些优化，将其放到常量区

char drr[] = "123";

//字符串123保存在栈区，可以通过drr去修改

21. 数组和链表的区别

1. 数组：

   元素在内存中连续存放，占用空间相同，可以通过下标快速随机访问，插入和删除数据效率低。

   随机访问强、增删慢、可能浪费内存、对内存要求高（连续）、大小固定
2. 链表

   元素非顺序存储，而是通过指针联系到一起。如果需要访问某个元素。需要从开始进行遍历，数据随意增删

   插入删除快、内存利用率高、不能随即查找

22. 加密算法

1. 古典密码
2. 对称密码

   加解密使用相同的密钥

   速度较快

   密钥传输存在安全问题

   对称加密算法：算法公开、计算量小、加密效率高

   对称加密算法的缺点是在数据传送前，发送方和接收方必须商定好秘钥，然后使双方都能保存好秘钥。其次如果一方的秘钥被泄露，那么加密信息也就不安全了。另外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一秘钥，这会使得收、发双方所拥有的钥匙数量巨大，密钥管理成为双方的负担。
3. 非对称密码--公钥加密

   RSA、DSA、ECC
4. 单向加密：

   输入相同、输出必然相同

   雪崩效应：输入微小改变、结果发生巨大变化

   定长输出、可以将大、小子串处理成等长

   不可逆，无法根据特征码还原

   1. MD5
   2. SHA
   3. CRC-32

23. 随机数算法

1. 生成一个n-1的随机数x
2. 交换array[x] 与array[n]
3. 生成n-2的随机数
4. 以此类推

24. QT信号与槽机制

是QT的对象间进行通信的一种手段，通过某一个对象的事件触发告知另一对像

1. 信号与槽机制 和 回调的区别

   回调：将函数指针传递给调用方，不保证类型安全，强耦合
2. 凡是继承自QObject的类都具有信号和槽成员。可以有效减少函数指针的使用。

   函数和槽都采用函数作为实现形式。可以静态/动态的将槽与函数关联
3. 关联信号与槽的方式：connect

   connect(信号发出对象sender，对象触发事件，接受对象receiver，槽函数)

   disconnect：断开连接到该对象的信号

   事件：SIGNAL（xxx）

   SLOT（XXX）
4. 总结

   1. 信号与槽式机制安全的，相关联的信号、槽必须参数匹配

2. 信号与槽是松耦合的，信号发送者不需要知道接收者信息
3. 信号与槽可以使用任意类型的任意数量的参数

5. 当信号与多个槽函数进行关联时，槽函数按照建立连接的顺序进行执行

   多个信号也可以与一个槽进行关联时

   可以使用connect连接信号与信号

6. 信号与槽的参数个数和类型需要一致，至少信号的参数不少于槽的参数，如果不匹配，则编译/运行错误
7. 使用信号、槽的类，必须加入宏Q_OBJECT
8. 当信号发射时，关联的槽函数立即执行，只有当槽函数处理完毕后才会继续进行发射后的代码

25. 交叉验证

分出训练集和验证集进行验证

均方误差MSE：$$MSE = \frac{1}{M}\sum_{m=1}^{M}(y_m - \hat{y}_m)^2$$

1. 使用LOOCV交叉验证方法，

   这种方法每次取一个为测试值，其余全为训练值，然后计算全部的平均MSE

2. k-fold交叉验证

   将数据集分为k份，每次取一份为验证集，

$$\mathrm{CV}_{(k)} = \frac{1}{k}\sum_{i=1}^{k}\mathrm{MSE}_i.$$

26. 中缀表达式转后缀表达式

1. 数字直接输出
2. 左括号：进栈

   运算符：与栈顶符号进行优先级比较，如果比栈顶优先级低，符号进栈（默认栈顶左括号，左括号优先级最低）如果栈顶符号优先级高，则弹出栈顶符号并输出，然后进栈

   右括号：弹出元素直到匹配到左括号