1. **数组和链表区别：**

数组和链表是线性表两种不同的存储结构。

1. 逻辑结构。数组必须事先定义固定的长度，不能动态增减数组的大小。当数据较少时，存在空间浪费，数据增加时，面临数组越界的危险，链表采用动态内存形式实现，可动态增减数据项，增加时使用new/malloc分配内存,删除时使用delete/free释放内存。
2. 内存结构。数组从栈中分配空间，自由度小，链表从堆中分配空间，自由度大，但申请管理比较麻烦。
3. 数组中元素顺序存储，逻辑上相邻的数据项在物理地址上也是相邻的；链表中数据随机存储，逻辑上相邻的数据项在物理地址上不一定是相邻的。数组的随机访问效率较高，但插入，删除效率较低，需要移动大量元素。链表的插入，删除相对数组有很高的效率，但其随机访问需要从表头逐一遍历，所以其随机访问效率较低。
4. 数组便于查询，链表便于删除和插入

**二、栈和队列**

栈和队列是两种比较重要的线性结构，他们的插入和删除受到更多的约束和限定，是一种限定性的线性表结构。栈是操作只限定在栈顶，是一种后进先出的结构。队的操作限定在对首和对尾，对首执行出队操作，对尾执行入队操作，是一种先进先出的结构。

**三、树(图)的遍历**

深度优先遍历：对从某一点开始的分支路径深入遍历到不能再深入为止，且每个节点访问一次。

广度优先遍历：从某一节点出发，依次访问与该节点邻接的所有节点。

**四、栈和堆**

1.内存分配：

堆：由程序员负责申请，并提供申请的大小。C语言中堆内存的分配是用malloc()函数或realloc()来完成的；C++中用new运算符来分配。堆在使用完成后，必须由程序员显示的释放。在C语言中释放的函数为free(),在C++中用delete操作符负责释放。若程序员忘记释放，就可能发生内存泄露。

栈：由系统自动分配与释放。系统在栈中为局部变量开辟空间，栈用于存放函数的参数值、局部变量的值等

在资源分配的具体实现上，栈和堆也有很大区别：

堆: 堆的分配是通过操作系统的一个记录空闲内存地址的链表来实现的。当系统收到程序的申请是，会遍历该链表，寻找第一个空间大于所申请空间的堆节点，然后将节点从空闲节点链中删除，并将该节点的空间分配给程序。另外，对于大多数系统，会在这块内存的首地址处记录本次分配的大小，这样代码中的delete语句才能正确的释放本内存空间。由于找到的堆节点的大小不一定刚好等于申请的大小，系统会自动将多余的那部分重新放入空闲链表中。

栈：每个进程都拥有自己的一个固定的栈空间。在系统分配的时候，只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

2 大小限制：

堆：是向高地址扩展的数据结构，是不连续的内存区域。由于系统使用链表来存储空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。（如果堆失败呢？）

栈：是向低地址扩展的数据结构，是一块连续的内存区域。（这句话的意思是栈顶的地址和栈的栈的最大容量是系统预先规定的，是编译时就确定的一个常数），如果申请的空间超过栈的剩余空间时，将提示栈溢出。（栈空间一般较小？）

3. 效率比较

堆：是由new分配的内存，一般速度比较慢，而且容易产生内存碎片，但使用起来方便。在Windows中，最好的方式是使用VirtualAlloc()分配内存，它既不是在堆中也不是在栈中，而是直接在进程的地址空间中保留一块内存，该方式虽然用起来最不方便，但是速度却很快，也最灵活。

栈：有系统自动分配，速度较快。但程序员无法控制。

4.存放内容

堆：一般是在堆的头部用一个字节存放堆的大小，剩余部分存储的内容由程序员根据程序计算的需要决定。

栈：栈是用来记录程序执行时函数调用过程中的活动记录。

**五、常用的查找算法和排序算法并分析算法复杂度**

**查找算法：(静态查找表和动态查找表)**

1. 顺序查找：元素可以顺序存储，也可以链式存储，不要求有序，时间复杂度O(n)

2. 折半查找：元素必须顺序存储，必须有序，时间复杂度O(logn)。插入，删除时需要移动大量节点。

3. 二叉排序查找：平均时间复杂度O(n),最坏情况下O(logn),插入和删除非常方便，无需移动大量节点。对于需要经常需要插入、删除、查找运算的表，宜采用二叉查找树结构

4. 平衡二叉树(AVL树)和红黑树

**AVL树：**是一种二叉排序树，其中每一个节点的左子树和右子树的高度差不超过1.

**RB树：**

**AVL树与RB树区别：**

插入，查找时间复杂度O(logn),适用于排序。

5. 多路查找树

**定义：**每一个节点孩子可以多于两个，且每一个节点可以存储多个元素。

**B树和B+树：**B树中每一个在该树中只出现一次，有可能在叶子节点上，也有可能在分支节点上。而B+树中，出现在分支节点的元素会在叶节点再次列出，而且，每一个叶节点都会保存一个指向后一叶节点的指针。随机查找与B树一致，但分支节点只提供索引的作用，不提供实际访问。对于范围查找和遍历提供很大方便。

6. Hash表查找：存储位置和关键字之间有一个对应关系。关键在于使用hash函数确定其key值，插入和查找效率都是O(1).不适合做范围查找，排序等。

**Hash函数设计原则**：简单，均匀，存储利用率高，尽量避免冲突

**Hash函数构造方法：**直接定址法(f=a\*key+b,需事先知道关键字分布情况)，数字分析法(抽取一部分做散列地址，适合关键字位数较长)，平方取中法(适合位数不大的),除留取余法(f=key%p)

**处理散列冲突**：

**开放地址法**：**线性探测**会产生堆积现象，存入和查找效率都很低。

**二次探测**双向寻找可以插入的位置，提高效率；增加平方运算为了防止堆积。

**随机探测** 位移量采用随机函数计算得到。(此处随机数是伪随机数，查找时也需要与插入一致的随机种子)

**再散列函数法：**发生冲突时，换一个散列函数计算，增加了计算时间

**链地址法：**将冲突的关键字以链表形式存储，带来了额外的存储开销，引入了单链表遍历的性能损耗。

**公共溢出区法**：如果关键字有冲突，将其移至公共溢出区。

**排序算法：**

稳定与不稳定：排序过程中不改变相同关键字的相对位置。

内排序与外排序：是否内外存多次交换。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **排序方法** | **平均情况** | **最好情况** | **最坏情况** | **辅助空间** | **稳定性** |
| 冒泡 |  |  |  |  | 稳定 |
| 选择排序 |  |  |  |  | 不稳定 |
| 插入排序 |  |  |  |  | 稳定 |
| 希尔排序 |  |  |  |  | 不稳定 |
| 堆排序 |  |  |  |  | 不稳定 |
| 归并 |  |  |  |  | 稳定 |
| 快速排序 |  |  |  |  | 不稳定 |