## 数据结构面试专题

**1、常用数据结构简介**

数据结构是指相互之间存在着一种或多种关系的数据元素的集合和该集合中数据元素间的关系组成。常用的数据有：数组、栈、队列、链表、树、图、堆、散列表。

1）数组：在内存中连续存储多个元素的结构。数组元素通过下标访问，下标从0开始。优点：访问速度快；缺点：数组大小固定后无法扩容，只能存储一种类型的数据，添加删除操作慢。适用场景：适用于需频繁查找，对存储空间要求不高，很少添加删除。

2）栈：一种特殊的线性表，只可以在栈顶操作，先进后出，从栈顶放入元素叫入栈，从栈顶取出元素叫出栈。应用场景：用于实现递归功能，如斐波那契数列。

3）队列：一种线性表，在列表一端添加元素，另一端取出，先进先出。使用场景：多线程阻塞队列管理中。

4）链表：物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表的指针地址实现，每个元素包含两个结点，一个是存储元素的数据域，一个是指向下一个结点地址的指针域。有单链表、双向链表、循环链表。优点：可以任意加减元素，不需要初始化容量，添加删除元素只需改变前后两个元素结点的指针域即可。缺点：因为含有大量指针域，固占用空间大，查找耗时。适用场景：数据量小，需频繁增加删除操作。

5）树：由n个有限节点组成一种具有层次关系的集合。二叉树（每个结点最多有两个子树，结点的度最大为2，左子树和右子树有顺序）、红黑树（HashMap底层源码）、B+树（mysql的数据库索引结构）

6）散列表（哈希表）：根据键值对来存储访问。

7）堆：堆中某个节点的值总是不大于或不小于其父节点的值，堆总是一棵完全二叉树。

8）图：由结点的有穷集合V和边的集合E组成。

**2、并发集合了解哪些？**

1）并发List，包括Vector和CopyOnWriteArrayList是两个线程安全的List，Vector读写操作都用了同步，CopyOnWriteArrayList在写的时候会复制一个副本，对副本写，写完用副本替换原值，读时不需要同步。

2）并发Set，CopyOnWriteArraySet基于CopyOnWriteArrayList来实现的，不允许存在重复的对象。

3）并发Map，ConcurrentHashMap，内部实现了锁分离，get操作是无锁的。

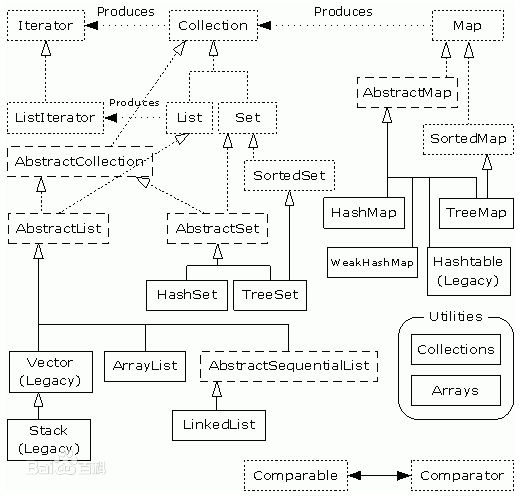
4）并发Queue，ConcurrentLinkedQueue适用于高并发场景下的队列，通过无锁方式实现。 BlockingQueue阻塞队列，应用场景，生产者-消费者模式，若生产快于消费，生产队列装满时会阻塞，等待消费。

5）并发Deque, LinkedBlockingDueue没有进行读写锁分离，同一时间只能有一个线程对其操作。

6）并发锁重入锁ReentrantLock，互斥锁，一次最多只能一个线程拿到锁。

7）读写锁ReadWriteLock，有读取和写入锁两种，读取允许多个读取线程同时持有，而写入只能有一个线程持有。

**3、列举java的集合以及集合之间的继承关系**



**5、容器类介绍以及之间的区别**

1）Collection接口：集合框架的根接口，它是集合类框架中最具一般性的顶层接口。

2）Map接口：提供了键值对的映射关系的集合，关键字不能有重复值，每个关键字至多可映射一个值。HashMap(通过散列机制，用于快速访问)，TreeMap（保持key处于排序状态，访问速度不如hashmap）, LinkedHashMap(保持元素的插入顺序)

3）Set接口：可包含重复的元素，LinkedHashSet TreeSet(用红黑树来存储元素) HashSet

4）List接口:可通过索引对元素进行精准的插入和查找，实现类有ArrayList LinkedList

5）Queue接口：继承自Collection接口，LinkedList实现了Queue接口，提供了支持队列的行为。

6）Iterator接口：为了迭代集合

7）Comparable接口：用于比较

**6、List,Set,Map的区别**

Set是一个无序的集合，不能包含重复的元素；

list是一个有序的集合可以包含重复的元素，提供了按索引访问的方式；

map包含了key-value对，map中key必须唯一，value可以重复。

**7、HashMap的实现原理**

1）数据结构

jdk1.7及以前，HashMap由数组+链表组成，数组Entry是HashMap的主体，Entry是HashMap中的一个静态内部类，每一个Entry包含一个key-value键值对，链表是为解决哈希冲突而存在。

从jdk1.8起，HashMap是由数组+链表/红黑树组成，当某个bucket位置的链表长度达到阀值8时，这个链表就转变成红黑树。

2）HashMap是线程不安全的，存储比较快，能接受null值，HashMap通过put(key, value)来储存元素，通过get(key)来得到value值，通过hash算法来计算hashcode值，用hashcode标识Entry在bucket中存储的位置。

3）HashMap中为什么要使用加载因子，为什么要进行扩容

加载因子是指当HashMap中存储的元素/最大空间值的阀值，如果超过这个值，就会进行扩容。加载因子是为了让空间得到充分利用，如果加载因子太大，虽对空间利用更充分，但查找效率会降低；如果加载因子太小，表中的数据过于稀疏，很多空间还没用就开始扩容，就会对空间造成浪费。

至于为什么要扩容，如果不扩容，HashMap中数组处的链表会越来越长，这样查找效率就会大大降低。

**7.1 HashMap如何put数据（从HashMap源码角度讲解）？**

当我们使用put(key, value)存储对象到HashMap中时，具体实现步骤如下：

1）先判断table数组是否为空，为空以默认大小构建table，table默认空间大小为16

2）计算key的hash值，并计算hash&(n-1)值得到在数组中的位置index，如果该位置没值即table[index]为空，则直接将该键值对存放在table[index]处。

3）如果table[index]处不为空，说明发生了hash冲突，判断table[index]处结点是否是TreeNode(红黑树结点)类型数据，如果是则执行putTreeVal方法，按红黑树规则将键值对存入；

4）如果table[index]是链表形式，遍历该链表上的数据，将该键值对放在table[index]处，并将其指向原index处的链表。判断链表上的结点数是否大于链表最大结点限制（默认为8），如果超过了需执行treeifyBin()操作，则要将该链表转换成红黑树结构。

5）判断HashMap中数据个数是否超过了（最大容量\*装载因子），如果超过了，还需要对其进行扩容操作。

**7.2 HashMap如何get数据？**

get(key)方法获取key的hash值，计算hash&(n-1)得到在链表数组中的位置first=table[hash&(n-1)]，先判断first（即数组中的那个）的key是否与参数key相等，不等的话，判断结点是否是TreeNode类型，是则调用getTreeNode(hash, key)从二叉树中查找结点，不是TreeNode类型说明还是链表型，就遍历链表找到相同的key值返回对应的value值即可。

**7.3 当两个对象的hashcode相同，即发生碰撞时，HashMap如何处理**

当两个对象的hashcode相同，它们的bucket位置相同，hashMap会用链表或是红黑树来存储对象。Entry类里有一个next属性，作用是指向下一个Entry。第一个键值对A进来，通过计算其key的hash得到index，记做Entry[index]=A。一会又进来一个键值对B，通过计算其key的hash也是index，HashMap会将B.next=A, Entry[index]=B.如果又进来C，其key的hash也是index,会将C.next=B, Entry[index]=C.这样bucket为index的地方存放了A\B\C三个键值对，它们能过next属性链在一起。数组中存储的是最后插入的元素，其他元素都在后面的链表里。

**7.4 如果两个键的hashcode相同，如何获取值对象？**

当调用get方法时，hashmap会使用键对象的hashcode找到bucket位置，找到bucket位置后，会调用key.equals()方法去找到链表中正确的节点，最终找到值对象。

**7.5 hashMap如何扩容**

HashMap默认负载因为是0.75，当一个map填满了75%的bucket时，和其他集合类一样，将会创建原来HashMap大小两倍的bucket数组，来重新调整HashMap的大小，并将原来的对象放入新的bucket数组中。

在jdk1.7及以前，多线程扩容可能出现死循环。因为在调整大小过程中，存储在某个bucket位置中的链表元素次序会反过来，而多线程情况下可能某个线程翻转完链表，另外一个线程又开始翻转，条件竞争发生了，那么就死循环了。

而在jdk1.8中，会将原来链表结构保存至节点e中，将原来数组中的位置设为null，然后依次遍历e，根据hash&n是否为0分成两条支链，保存在新数组中。如果多线程情况可能会取到null值造成数据丢失。

**8、ConcurrentHashMap的实现原理**

1）jdk1.7及以前：一个ConcurrentHashMap由一个segment数组和多个HashEntry组成，每一个segment都包含一个HashEntry数组, Segment继承ReentrantLock用来充当锁角色，每一个segment包含了对自己的HashEntry的操作，如get\put\replace操作，这些操作发生时，对自己的HashEntry进行锁定。由于每一个segment写操作只锁定自己的HashEntry，可以存在多个线程同时写的情况。

jdk1.8以后：ConcurrentHashMap取消了segments字段，采用transient volatile HashEntry<K, V> table保存数据，采用table数组元素作为锁，实现对每一个数组数据进行加锁，进一小减少并发冲突概率。ConcurrentHashMap是用Node数组+链表+红黑树数据结构来实现的，并发制定用synchronized和CAS操作。

2）Segment实现了ReentrantLock重入锁，当执行put操作，会进行第一次key的hash来定位Segment的位置，若该Segment还没有初始化，会通过CAS操作进行赋值，再进行第二次hash操作，找到相应的HashEntry位置。

**9、ArrayMap和HashMap的对比**

1)存储方式不一样，HashMap内部有一个Node<K,V>[]对象，每个键值对都会存储到这个对象里，当用put方法添加键值对时，会new一个Node对象，tab[i] = newNode(hash, key, value, next);

ArrayMap存储则是由两个数组来维护，int[] mHashes; Object[] mArray; mHashes数组中保存的是每一项的HashCode值，mArray存的是键值对，每两个元素代表一个键值对，前面保存key，后面保存value。mHashes[index]=hash; mArray[index<<1]=key; mArray[(index<<1)+1]=value;

ArrayMap相对于HashMap，无需为每个键值对创建Node对象，且在数组中连续存放，更省空间。

2）添加数据时扩容处理不一样，进行了new操作，重新创建对象，开销很大；而ArrayMap用的是copy数据，所有效率相对高些；

3）ArrayMap提供了数组收缩功能，在clear或remove后，会重新收缩数组，释放空间；

4）ArrayMap采用二分法查找，mHashes中的hash值是按照从小到大的顺序连续存放的，通过二分查找来获取对应hash下标index，去mArray中查找键值对。mHashes中的index\*2是mArray中的key下标，index\*2+1为value的下标，由于存在hash碰撞情况，二分查找到的下标可能是多个连续相同的hash值中的任意一个，此时需要用equals比对命中的key对象是否相等，不相等，应当从当前index先向后再向前遍历所有相同hash值。

5）sparseArray比ArrayMap进一步优化空间，SparseArray专门对基本类型做了优化，Key只能是可排序的基本类型，如int\long，对value，除了泛型Value，还对每种基本类型有单独实现，如SparseBooleanArray\SparseLongArray等。无需包装，直接使用基本类型值，无需hash，直接使用基本类型值索引和判断相等，无碰撞，无需调用hashCode方法，无需equals比较。SparseArray延迟删除。

**10、HashTable实现原理**

Hashtable中的无参构造方法Hashtable()中调用了this(11, 0.75f)，说明它默认容量是11，加载因子是0.75,在构造方法上会new HashtableEntry<?, ?>[initialCapacity]; 会新建一个容量是初始容量的HashtableEntry数组。HashtableEntry数组中包含hash\Key\Value\next变量，链表形式，重写了hashCode和equals方法。Hashtable所有public方法都在方法体上加上了synchronized锁操作，说明它是线程安全的。它还实现了Serializable接口中的writeObject和readObject方法，分别实现了逐行读取和写入的功能，并且加了synchronized锁操作。

（1） put(Key, Value)方法

1）先判断value是否为空，为空抛出空指针异常；

2）根据key的hashCode()值，计算table表中的位置索引(hash&0x7FFFFFFF)%tab.length值index，如果该索引处有值，再判断该索引处链表中是否包含相同的key，如果key值相同则替换旧值。

3）如果没有相同的key值，调用addEntry方法，在addEntry中判断count大小是否超过了最大容量限制，如果超过了需要重新rehash()，容量变成原来容量\*2+1，将原表中的值都重新计算hash值放入新表中。再构造一个HashtableEntry对象放入相应的table表头，如果原索引处有值，则将table[index].next指向原索引处的链表。

（2）get方法

根所key.hashCode()，计算它在table表中的位置，(hash&0x7FFFFFFF)%tab.length，遍历该索引处表的位置中是否有值，是否存在链表，再判断是key值和hash值是否相等，相等则返回对应的value值。

**11、HashMap和HashTable的区别**

1）Hashtable是个线程安全的类，在对外方法都添加了synchronized方法，序列化方法上也添加了synchronized同步锁方法，而HashMap非线程安全。这也导致Hashtable的读写等操作比HashMap慢。

2）Hashtable不允许值和键为空，若为空会抛出空指针。而HashMap允许键和值为空；

3）Hashtable根据key值的hashCode计算索引，(hash&0x7FFFFFFF)%tab.length，保证hash值始终为正数且不超过表的长度。而HashMap中计算索引值是通过hash(key)&(tab.length-1)，是通过与操作，计算出在表中的位置会比Hashtable快。

4）Hashtable容量能为任意大于等于1的正数，而HashMap的容量必须为2^n，Hashtable默认容量为11，HashMap初始容量为16

5）Hashtable每次扩容，新容量为旧容量的2倍+1，而HashMap为旧容量的2倍。

**12、HashMap与HashSet的区别**

HashSet底层实现是HashMap,内部包含一个HashMap<E, Ojbect> map变量

*private transient HashMap<E,Object> map;*

一个Object PRESENT变量（当成插入map中的value值）

*private static final Object PRESENT = new Object();*

HashSet中元素都存到HashMap键值对的Key上面。具体可以查看HashSet的add方法，直接调用了HashMap的put方法，将值作为HashMap的键，值用一个固定的PRESENT值。

*public boolean add(E e) {*

*return map.put(e, PRESENT)==null;*

*}*

HashSet没有单独的get方法，用的是HashMap的。HashSet实现了Set接口，不允许集合中出现重复元素，将对象存储进HashSet前，要先确保对象重写了hashCode()和equals方法，以保证放入set对象是唯一的。

**13、HashSet与HashMap怎么判断集合元素重复？**

HashMap在放入key-value键值对是，先通过key计算其hashCode()值，再与tab.length-1做与操作，确定下标index处是否有值，如果有值，再调用key对象的equals方法，对象不同则插入到表头，相同则覆盖；

HashSet是将数据存放到HashMap的key中，HashMap是key-value形式的数据结构，它的key是唯一的，HashSet利用此原理保证放入的对象唯一性。

**14、集合Set实现Hash怎么防止碰撞**

HashSet底层实现是HashMap，HashMap如果两个不同Key对象的hashCode()值相等，会用链表存储，HashSet也一样。

**15、ArrayList和LinkedList的区别，以及应用场景**

ArrayList底层是用数组实现的，随着元素添加，其大小是动态增大的；在内存中是连续存放的；如果在集合末尾添加或删除元素，所用时间是一致的，如果在列表中间添加或删除元素，所用时间会大大增加。通过索引查找元素速度很快。适合场合：查询比较多的场景

LinkedList底层是通过双向链表实现的，LinkedList和ArrayList相比，增删速度快，但查询和修改值速度慢。在内存中不是连续内存。场景：增删操作比较多的场景。

-二叉树的深度优先遍历和广度优先遍历的具体实现

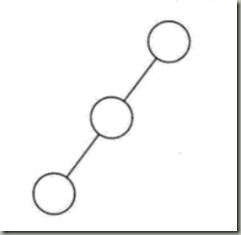
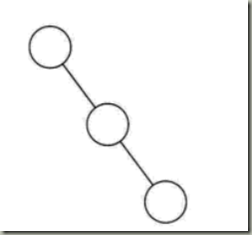
二叉树简介：

二叉树是树的特殊一种，具有如下特点：1、每个结点最多有两颗子树，结点的度最大为2。2、左子树和右子树是有顺序的，次序不能颠倒。3、即使某结点只有一个子树，也要区分左右子树。

# 一、特殊的二叉树及特点

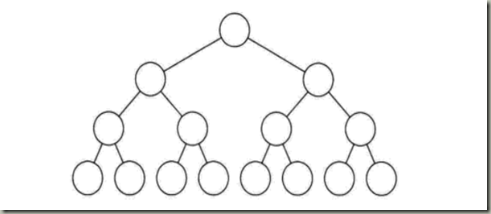
## 1、斜树

所有的结点都只有左子树（左斜树），或者只有右子树（右斜树）。这就是斜树，应用较少

[](http://images0.cnblogs.com/blog/451660/201508/181910524254834.png)[](http://images0.cnblogs.com/blog/451660/201508/181910544565246.png)

## 2、满二叉树

所有的分支结点都存在左子树和右子树，并且所有的叶子结点都在同一层上，这样就是满二叉树。就是完美圆满的意思，关键在于树的平衡。

[](http://images0.cnblogs.com/blog/451660/201508/181910558476787.png)

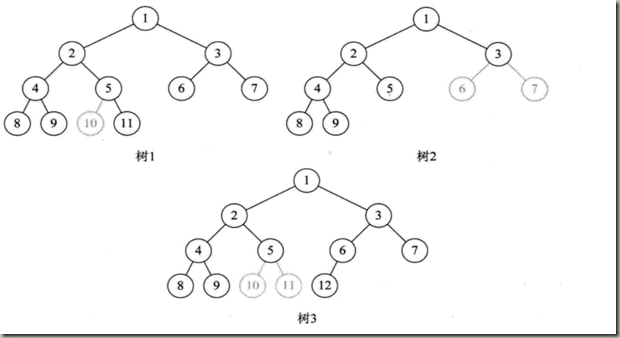
根据满二叉树的定义，得到其特点为：

1. 叶子只能出现在最下一层。
2. 非叶子结点度一定是2.
3. 在同样深度的二叉树中，满二叉树的结点个数最多，叶子树最多。

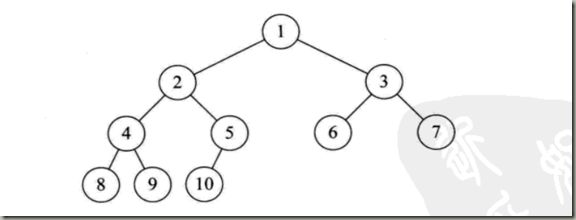
## 3、完全二叉树

对一棵具有n个结点的二叉树按层序排号，如果编号为i的结点与同样深度的满二叉树编号为i结点在二叉树中位置完全相同，就是完全二叉树。满二叉树必须是完全二叉树，反过来不一定成立。

其中关键点是按层序编号，然后对应查找。

[](http://images0.cnblogs.com/blog/451660/201508/181910572062273.png)

在上图中，树1，按层次编号5结点没有左子树，有右子树，10结点缺失。树2由于3结点没有字数，是的6,7位置空挡了。树3中结点5没有子树。

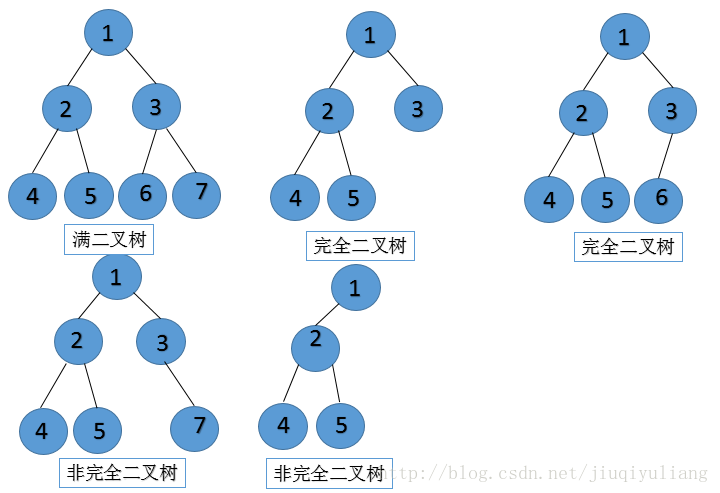
[](http://images0.cnblogs.com/blog/451660/201508/181910588167513.png)

上图就是一个完全二叉树。

结合完全二叉树定义得到其特点：

1. 叶子结点只能出现在最下一层（满二叉树继承而来）
2. 最下层叶子结点一定集中在左 部连续位置。
3. 倒数第二层，如有叶子节点，一定出现在右部连续位置。
4. 同样结点树的二叉树，完全二叉树的深度最小（满二叉树也是对的）。

根据下图加深理解，什么时候是完全二叉树。



# 三、二叉树性质

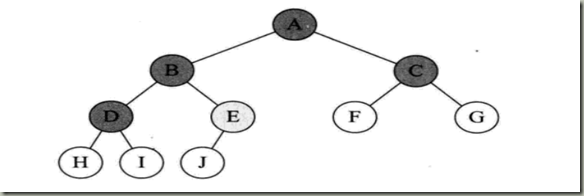
## 1、一般二叉树性质

1、在非空二叉树的i层上，至多有2i-1个节点(i>=1)。通过归纳法论证。

2、在深度为K的二叉树上最多有2k-1个结点（k>=1)。通过归纳法论证。

3、对于任何一棵非空的二叉树,如果叶节点个数为n0，度数为2的节点个数为n2，则有: n0 = n2 + 1

在一棵二叉树中，除了叶子结点（度为0）之外，就剩下度为2(n2)和1(n1)的结点了。则树的结点总数为T = n0+n1+n2;在二叉树中结点总数为T，而连线数为T-1.所以有：n0+n1+n2-1 = 2\*n2 +n1;最后得到n0 = n2+1;

[](http://images0.cnblogs.com/blog/451660/201508/181911001445541.png)

上图中结点总数是10，n2为4，n1为1，n0为5。

## 2、完全二叉树性质

a、具有n的结点的完全二叉树的深度为log2n+1.

满二叉树是完全二叉树，对于深度为k的满二叉树中结点数量是2k-1 = n，完全二叉树结点数量肯定最多2k-1,同时完全二叉树倒数第二层肯定是满的（倒数第一层有结点，那么倒是第二层序号和满二叉树相同），所以完全二叉树的结点数最少大于少一层的满二叉树，为2k-1-1。

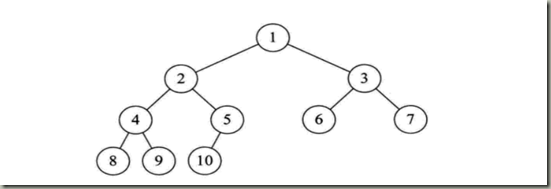
根据上面推断得出： 2k-1-1< n=<2k-1，因为结点数Nn为整数那么n<=2k-1可以推出n<=2k,n>2k-1-1可以推出 n>=2k-1,所以2k-1<n<=2k  。即可得k-1<=log2n<k 而k作为整数因此k=[log2n]+1。

b、如果有一颗有n个节点的完全二叉树的节点按层次序编号，对任一层的节点i（1<=i<=n）有

    1.如果i=1，则节点是二叉树的根，无双亲，如果i>1，则其双亲节点为[i/2]，向下取整

    2.如果2i>n那么节点i没有左孩子，否则其左孩子为2i

    3.如果2i+1>n那么节点没有右孩子，否则右孩子为2i+1

[](http://images0.cnblogs.com/blog/451660/201508/181911014886797.png)

在上图中验证

第一条：

当i=1时，为根节点。当i>1时，比如结点为7，他的双亲就是7/2= 3；结点9双亲为4.

第二条：

结点6,6\*2 = 12>10，所以结点6无左孩子，是叶子结点。结点5，5\*2 = 10，左孩子是10,结点4，为8.

第三条：

结点5，2\*5+1>10,没有右孩子，结点4，则有右孩子。

# 四、二叉树遍历

二叉树遍历：从树的根节点出发，按照某种次序依次访问二叉树中所有的结点，使得每个结点被访问仅且一次。

这里有两个关键词：访问和次序。

## 1、前序遍历

基本思想：先访问根结点，再先序遍历左子树，最后再先序遍历右子树即根—左—右。

图中前序遍历结果是：1，2，4，5，7，8，3，6。

a/前序递归遍历的代码实现，如下所示

[制代码](javascript:void(0);)

//前序递归遍历

void PreOrderTraverse(BiTree t)

{

//注意跳出条件

if(t != NULL)

{

//注意访问语句顺序

printf("%c ", t->data);

PreOrderTraverse(t->lchild);

PreOrderTraverse(t->rchild);

}

}

[制代码](javascript:void(0);)

前序非递归遍历:

对于任一结点p：

        a. 访问结点p，并将结点p入栈；

        b. 判断结点p的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点p，循环置a；若不为空，则将p的左孩子置为当前结点p；

        c. 直到p为空，并且栈为空，则遍历结束。

[制代码](javascript:void(0);)

//前序非递归遍历

int NoPreOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree tmp = t;

if(tmp == NULL)

{

fprintf(stdout, "the tree is null.\n");

return ERROR;

}

//现将左子树压入栈，当到叶子结点后，出栈，获取右子树，然后在压入右子树的左子树。

//顺序不能变

while((tmp != NULL) || (IsEmpty(&s) != 1))

{

while(tmp != NULL)

{

Push(&s, tmp);

printf("%c ", tmp->data);

tmp = tmp->lchild;

}

if(IsEmpty(&s) != 1)

{

Pop(&s, &tmp);

tmp = tmp->rchild;

}

}

return OK;

}

[制代码](javascript:void(0);)

## 2、中序遍历

基本思想：先中序遍历左子树，然后再访问根结点，最后再中序遍历右子树即左—根—右。

图中中序遍历结果是：4，2，7，8，5，1，3，6。

中序遍历迭代代码

[制代码](javascript:void(0);)

//中序递归遍历

void InOrderTraverse(BiTree t)

{

if(t != NULL)

{

InOrderTraverse(t->lchild);

printf("%c ", t->data);

InOrderTraverse(t->rchild);

}

}

[制代码](javascript:void(0);)

2）中序非递归遍历

    根据中序遍历的顺序，对于任一结点，优先访问其左孩子，而左孩子结点又可以看做一个根结点，然后继续访问其左孩子结点，直到遇到左孩子结点为空的结点才停止访问，然后按相同的规则访问其右子树。其处理过程如下：

       对于任一结点：

       a. 若其左孩子不为空，则将p入栈，并将p的左孩子设置为当前的p，然后对当前结点再进行相同的操作；

       b. 若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的p置为栈顶结点的右孩子；

       c. 直到p为空并且栈为空，则遍历结束。

[制代码](javascript:void(0);)

//中序非递归遍历二叉树

int NoInOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree tmp = t;

if(tmp == NULL)

{

fprintf(stderr, "the tree is null.\n");

return ERROR;

}

while(tmp != NULL || (IsEmpty(&s) != 1))

{

while(tmp != NULL)

{

Push(&s, tmp);

tmp = tmp->lchild;

}

if(IsEmpty(&s) != 1)

{

Pop(&s, &tmp);

printf("%c ", tmp->data);

tmp = tmp->rchild;

}

}

return OK;

}

[制代码](javascript:void(0);)

## 3、后序遍历

基本思想：先后序遍历左子树，然后再后序遍历右子树，最后再访问根结点即左—右—根。

图中后序遍历结果是：4，8，7，5，2，6，3，1。

后序递归遍历代码实现，如下所示。

[制代码](javascript:void(0);)

//后序递归遍历

void PostOrderTraverse(BiTree t)

{

if(t != NULL)

{

PostOrderTraverse(t->lchild);

PostOrderTraverse(t->rchild);

printf("%c ", t->data);

}

}

[制代码](javascript:void(0);)

  后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子和右孩子都已被访问，并且左孩子在右孩子之前访问才能访问根结点，这就为流程控制带来了难题。下面介绍一种思路。

     要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点p，先将其入栈。若p不存在左孩子和右孩子，则可以直接访问它，或者p存在左孩子或右孩子，但是其左孩子和右孩子都已经被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将p的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子之前别访问，左孩子和右孩子都在根结点前面被访问。

[制代码](javascript:void(0);)

//后序非递归遍历二叉树

int NoPostOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree cur; //当前结点

BiTree pre = NULL; //前一次访问的结点

BiTree tmp;

if(t == NULL)

{

fprintf(stderr, "the tree is null.\n");

return ERROR;

}

Push(&s, t);

while(IsEmpty(&s) != 1)

{

GetTop(&s, &cur);//

if((cur->lchild == NULL && cur->rchild == NULL) || (pre != NULL && (pre == cur->lchild || pre == cur->rchild)))

{

printf("%c ", cur->data); //如果当前结点没有孩子结点或者孩子结点都已被访问过

Pop(&s, &tmp);

pre = cur;

}

else

{

if(cur->rchild != NULL)

{

Push(&s, cur->rchild);

}

if(cur->lchild != NULL)

{

Push(&s, cur->lchild);

}

}

}

return OK;

}

[制代码](javascript:void(0);)

# 五、二叉树的建立

其实而二叉树的建立就是二叉树的遍历，只不过将输入内容改为建立结点而已，比如，利用前序遍历建立二叉树

[制代码](javascript:void(0);)

//创建树

//按先后次序输入二叉树中结点的值(一个字符),#表示空树

//构造二叉链表表示的二叉树

BiTree CreateTree(BiTree t)

{

char ch;

scanf("%c", &ch);

if(ch == '#')

{

t = NULL;

}

else

{

t = (BitNode \*)malloc(sizeof(BitNode));

if(t == NULL)

{

fprintf(stderr, "malloc() error in CreateTree.\n");

return;

}

t->data = ch; //生成根结点

t->lchild = CreateTree(t->lchild); //构造左子树

t->rchild = CreateTree(t->rchild); //构造右子树

}

return t;

}

<https://www.cnblogs.com/polly333/p/4740355.html>

<https://www.jianshu.com/p/bf73c8d50dc2>

深度优先遍历和广度优先遍历

<https://www.javazhiyin.com/12976.html>

-堆的结构

<https://blog.csdn.net/szu_crayon/article/details/81812946>

-堆和树的区别

<https://blog.csdn.net/lifei128/article/details/82392940>

-堆和栈在内存中的区别是什么(解答提示：可以从数据结构方面以及实际实现方面两个方面去回答)？

<https://www.cnblogs.com/danbing/p/5023231.html>

<https://blog.csdn.net/weixin_41856078/article/details/79725852>

-什么是深拷贝和浅拷贝

<https://www.cnblogs.com/dolphin0520/p/3700693.html>

-手写链表逆序代码

<https://blog.csdn.net/caroline_wendy/article/details/39314465>

-讲一下对树，B+树的理解

<https://www.jianshu.com/p/86a1fd2d7406>

<https://blog.csdn.net/u011240877/article/details/80490663>

-讲一下对图的理解

<https://www.cnblogs.com/polly333/p/4760275.html>

-判断单链表成环与否？

<https://blog.csdn.net/weixin_40255793/article/details/80060243>

<https://blog.csdn.net/fu908323236/article/details/78205462>

-链表翻转（即：翻转一个单项链表）

<https://www.cnblogs.com/byrhuangqiang/p/4311336.html>

-合并多个单有序链表（假设都是递增的）

<https://blog.csdn.net/sinat_25926481/article/details/49047135>

<https://blog.csdn.net/liang_henry/article/details/52943150>