抽象地说，基本上有序列的地方就可以应用树，因为树结构即是一种序列索引结构。

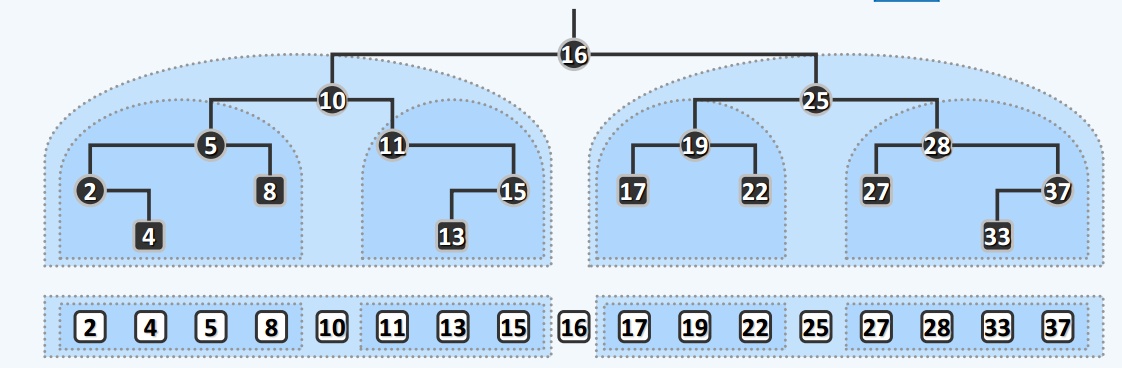
（我在这里提到的序列是一个广义的概念，一堆东西排在一起就叫序列。）

序列的核心接口就是三个：增、查、删。对于这个三个接口，我们要解决的核心问题是：①效率：怎么查得快

②稳定：如果不支持增删，那么序列就是静态结构，用处不大。而支持增删之后，需要考虑如何保证序列内部结构不会被增删操作破坏，导致查询效率受到影响。那么树是如何解决这两个问题的呢？

1.效率

对于效率问题，树相当于对序列建立了一个索引：



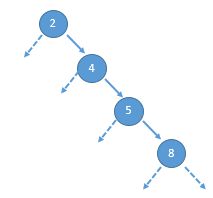
这个索引可以把原本O(n)的查找操作变为O(logn)，可以简单地理解为在数据结构的层面上构造了一个二分查找算法。

总之，树通过其结构来表达了一种划分查找方法，这一方法相比于遍历搜索的复杂度O(n)，一般情况下复杂度仅有O(logn)。

2.稳定

如果我们仅仅考虑效率问题，那散列比树要屌的多。查找复杂度为O(1)。之所以不能用散列来取代树，是因为散列需要预先开辟大量空间，并不是所有场景下都可以这么做；而如果空间不够，则会出现散列冲突（索引结构被破坏）。树则可以动态改变存储空间，且运用一些手段来保护自身索引结构。

自平衡二叉搜索树 (self-balanced BST)的精髓，便在于其维护自身稳定（平衡）的能力。当树不平衡时，其搜索复杂度便不再是O(logn)了——考虑一个极端情况：一棵树每个节点都只有右子节点，那么树就退化为了链表，查找复杂度也和链表一样是O(n)。



自平衡二叉搜索树通过不同的方式来保持平衡：AVL靠记录平衡因子，伸展树靠把节点转到树根，红黑树靠红黑规则（其实就是通过这个规则来让自身等价于(2,4)树）……总之就是在增删之后又加入一个自平衡步骤，来保证自身的稳定。

总之，当我们看到一个树形数据结构，主要应该考察以上两个方面：

①它是如何建立索引的？

②它是如何维持稳定的？

应用

AVL是一种高度平衡的二叉树，所以通常的结果是，维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，如果场景中对插入删除不频繁，只是对查找特别有要求，AVL还是优于红黑的。

红黑树的应用就很多了，除了上面同学提到的STL，还有

著名的linux进程调度Completely Fair Scheduler,

用红黑树管理进程控制块epoll在内核中的实现，用红黑树管理事件块nginx中，

用红黑树管理timer等

Java的TreeMap实现

B和B+主要用在文件系统以及数据库中做索引等，比如Mysql：B-Tree Index in MySqltrie 树的一个典型应用是前缀匹配，比如下面这个很常见的场景，在我们输入时，搜索引擎会给予提示。