binary search tree

binary search tree

AVL tree

avl tree

RB tree

rb tree

STL中的应用:

- set map
- multiset,multimap

```
typedef template<class Key, class Value> rb_tree< pair<Key,Value> > map;

typedef template<class T> rb_tree< T > set;
```

Splay tree

splay tree

三者比较

一、AVL树:

优点:查找、插入和删除,最坏复杂度均为O(logN)。实现操作简单

```
如过是随机插入或者删除,其理论上可以得到O(logN)的复杂度,但是实际情况大多不是随机的。如果是随机的,则AVL 树能够达到比RB树更优的结果,因为AVL树的高度更低。如果只进行插入和查找,则AVL树是优于RB树的,因为RB树 更多的优势还是在删除动作上。
```

缺点:1)借助高度或平衡因子,为此需要改造元素结构,或额外封装-->伸展树可以避免。

```
2)实测复杂度与理论复杂度上有差距。插入、删除后的旋转成本不菲。删除操作后,最多旋转O(logN)次,(Knuth证明,平 均最坏情况下概率为0.21次),若频繁进行插入/删除操作,得不偿失。
```

3)单词动态调整后,全树拓扑结构的变化量可达O(logN)次。-->红黑树为O(1)

二、伸展树(splay tree)、

优点、1)无序记录节点高度和平衡因子，编程实现简单易行

```
2)分摊复杂度为O(logN)

3)局部性强，缓存命中率极高时，效率甚至可以更高。
```

注：伸展树是根据数据访问的局部性而来的主要是：1)刚刚被访问的节点，极有可能在不就之后再次被访问到；2)将被访问 的下一个节点，极有可能就处于不就之前被访问过的某个节点的附近。

缺点：1)仍不能保证单词最坏情况的出现，不适用效率敏感的场合

```
2)复杂度分析比较复杂
```

三、红黑树

优点：1)所有的插入、删除、查找操作的复杂度都是O(logN)

```
2)插入操作能够在最多2次旋转后达到平衡状态，而删除操作更是能够在一次旋转后达到平衡状态。删除操作有可能导致          递归的双黑修正，但是在旋转之前，只是染色而树的结构没有任何实质性的改变，因此速度优于AVL树。
```

3)红黑树可以保证在每次插入或删除操作之后的重平衡过程中，全书拓扑结构的更新仅涉及常数个节点。尽管最坏情况下需对O(logN)个节点重染色，但就分摊意义而言，仅为O(1)个。

缺点：左右子树高度相差比AVL树大。

Trie树

huffman树

后缀树和后缀数组

- 后缀数组 后缀树

外排序 B树 B+树

- b tree and b+ tree