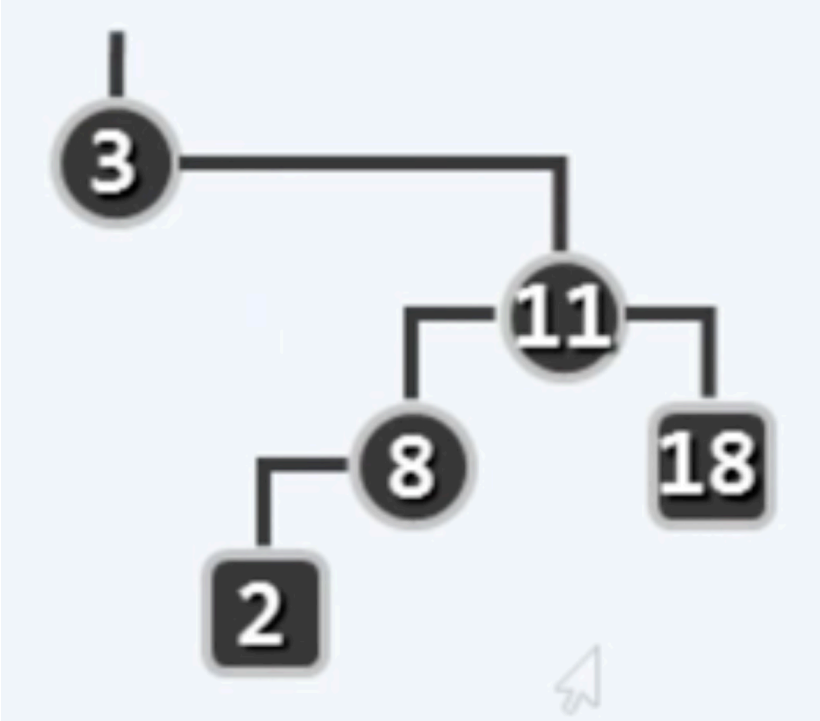
Binary Search Tree

物理上借助`List`的可拓展性，逻辑上借助有序`vector`的数据关系

顺序性

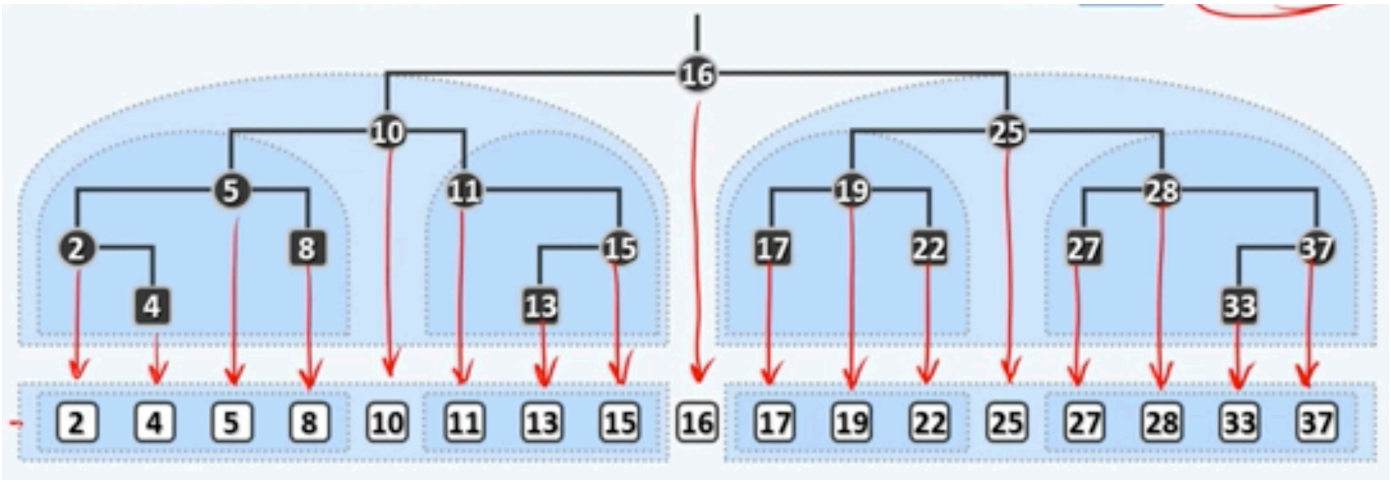
> a **rooted binary tree** whose internal nodes each store a key **greater than all the keys in the node's left subtree** and **less than those in its right subtree.**
>
> ### If Searching with duplicates allowed:
>
> you can put the duplicated entry in either left or right arbitrarily.

- *顺序性看似仅仅是一个局部特性，但是所有顺序性的组合却可以导出全局性的特征*
- 注意定义中是子树而非孩子（反例如下，下图不是BST）



单调性

> BST的中序遍历序列必然单调非降（充要条件）
>
> （可用数学归纳证明）



接口

```
virtual Node<T>* search(const T &);
virtual Node<T>* insert(const T &);
virtual bool remove(const T &);
```

实现

```
template <typename T>
Node<T>*& search(const T & data , Node<T>* & root){
   if (root==nullptr || root->element == data) return
root;
   return search(data , (data < root->element ? root-
>left : root->right));
}
```

- 复杂度正比于树高h，每递归一次，高度必定下降1
- `search`函数还可以返回结果的父亲，便于删除和插入操作的执行

```
template <typename T>
Node<T>* &search(const T & data , Node<T> * & root
, Node<T>* & parent){
    if (root==nullptr || root->element == data)
return root;
    parent = root;
    return search(data ,(data < root->element ?
root->left : root->right) , parent);
}
```

- `insert()`

```
template <typename T>
void insert(const T & data , Node<T>* & root){
    search(data , root) = new Node<T>(data);
}
```

```
//two star version
template <typename T>
Node<T>** search(const T & data , Node<T>** root){
   if ((*root)==nullptr || (*root)->element ==
data) return root;
   return search(data , &(data < (*root)->element
? (*root)->left : (*root)->right));
}
template <typename T>
void insert(const T & data , Node<T>** root){
    *search(data , root) = new Node<T>(data);
}
```

- `delete()`
  - 单分支直接取代
  - 双分支用中序遍历中的后继取代该节点
    - 维持顺序性
    - 实现：交换+单分支情况
    - 因为是中序遍历所以后继必为右子树左侧链末节点，必为单分支情况

```
template <typename T>
bool removeNode(const T & data , Node<T>** root){
    Node<T>** search_res = search(data , root);
    Node<T> * temp = *search_res;
    if (*search_res == nullptr) return false;
```

```
    if ((*search_res)->left == nullptr ) {
(*search_res) = (*search_res)->right;}
    else if ((*search_res)->right == nullptr) {
(*search_res) = (*search_res)->left; }
    else{
        Node<T>** succ = &(*search_res)->right;
        while((*succ)->left != nullptr){ *succ =
(*succ)->left; }
        std::swap((*succ)->element ,
(*search_res)->element);
        temp = *succ;
        *succ = (*succ)->right;
    }
    delete temp;
    return true;
}
//two star pointer can serve as reference to
pointer very well
```

$$T(n) = O(h) + \begin{cases} O(h) + O(1), & \text{two branches} \\ O(1), & \text{one branch} \end{cases}$$

Balanced Binary Search Tree

- 兄弟子树高度越接近，该树越**平衡**
- 对于有n个节点的树，当树高为$\log_2 n$时，称其为 ***理想平衡***

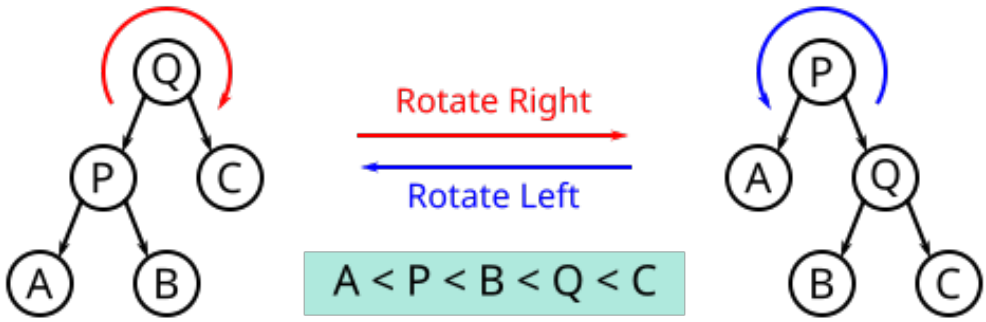
    - 对full binary tree和complete binary tree，理想情况可以达到，为何还是称之为理想?
    - 答：实际应用中几乎不会出现，即使出现，后续的插入删除操作也会破坏
    - 维护成本过高，考虑放宽标准
- 适度平衡
    - 插入删除操作时间复杂度为$O(h)$,与树高线性相关，所以我们只要树高渐进意义上为$\log_2 n$即可，称其为适度平衡
    - BBST即是适度平衡树
- BST等价
    - 左右不乱：**中序遍历序列完全一致**，单调性不得破坏
    - 上下可变：连接关系不尽相同，承袭关系可以颠倒

Balance

通过BST等价变换，把非平衡的BST调整为适度平衡的BST

- Tree rotation

> A tree rotation moves **one node up in the tree and one node down**. It is used to change the shape of the tree, and in particular to **decrease its height by moving smaller subtrees down and larger subtrees up**, resulting in improved performance of many tree operations.

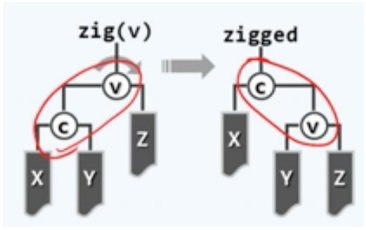


Note that we preserve the BST property!!! The in-order traversal sequence is exactly the same!!!

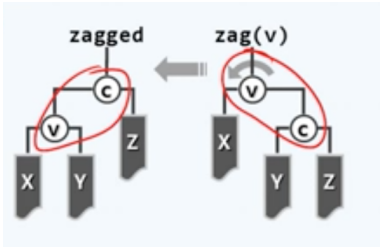
```
rotate_right(Q) == rotate_left(P)
```

Q is rotated to right and P is rotated to left.

- zig(right-rotation of V)



- zag(left-rotation of V)



- Zig,zag只在局部进行确保复杂度为$O(1)$
- 总共进行的zigzag操作数不超过$O(\log n)$

AVL Tree

- 定义平衡因子（平衡度）

$$\forall v \in Tree$$

```
int balfactor = abs(height(v->left) - height(v->right));
balfactor <= 1;
```

- Height of AVL tree?
  - 假设$h = height(AVL) \leq \log_2 n$,即要证明$2^h \geq n$

$$def: S(h) = \text{高度为}h\text{的}AVL\text{树的规模}$$

$$S(h) = 1 + \underbrace{S(h-1) + S(h-2)}_{\text{左右子树，高度差}1}$$

$$\therefore S(h) + 1 = (S(h-1) + 1) + (S(h-2) + 1)$$

$$\therefore T(h) = T(h-1) + T(h-2), \text{为斐波那契数列形式}$$

$$\because \text{斐波那契数列}f(n)\text{增长速率为}\Phi^n$$

$$\therefore S(h)\text{增长速率为}\Phi^h, \text{为指数级增长}$$

$$\therefore h\text{关于}n\text{的增长速率为对数级别}$$

$$\therefore \text{结论得证}$$

  - $S(h) = fib(h+3) - 1$
  - $h \approx 1.440 \log n$

- another way to figure it out:

$$
\begin{aligned}
N_h &= 1 + N_{h-1} + N_{h-2} \\
&> 1 + 2N_{h-2} \\
&> 2N_{h-2} \\
&= \Theta(2^{\frac{h}{2}})
\end{aligned}
$$

h每增加2, N翻一倍, 显然N关于h为指数增长

$$
\therefore h < 2\log N
$$

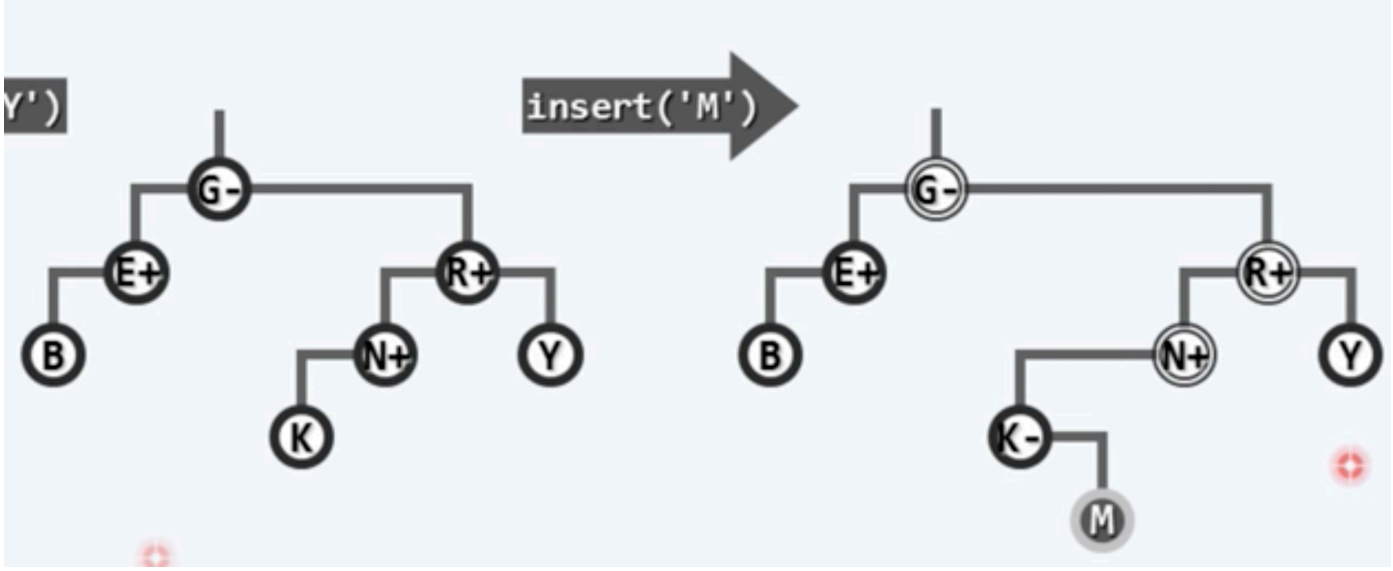
- Implementation

```
template <typename T> class AVL : public BST<T> {
    .....
}
```

  - we need to store the height of each node in itself, cause AVL tree count on height to work, it will cost a lot if we calculate it each time.(We can set 1 void tree height to -1 to faciliate our coding.)

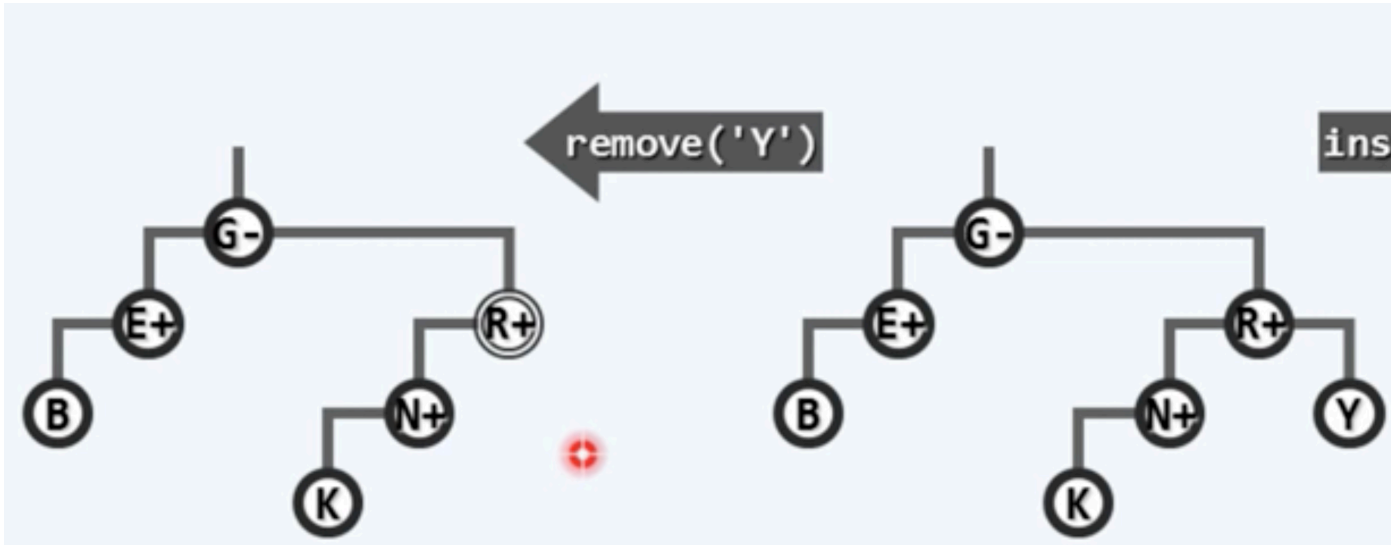
Detail

Insert



- 插入后只有被插入节点的祖先可能失衡，因为只有这些节点高度有变化
- 修复一个节点后，其余节点高度自然恢复
- 应用左旋右旋操作进行恢复(rebalance)，时间复杂度为$O(1)$，旋转后不要忘记更新高度

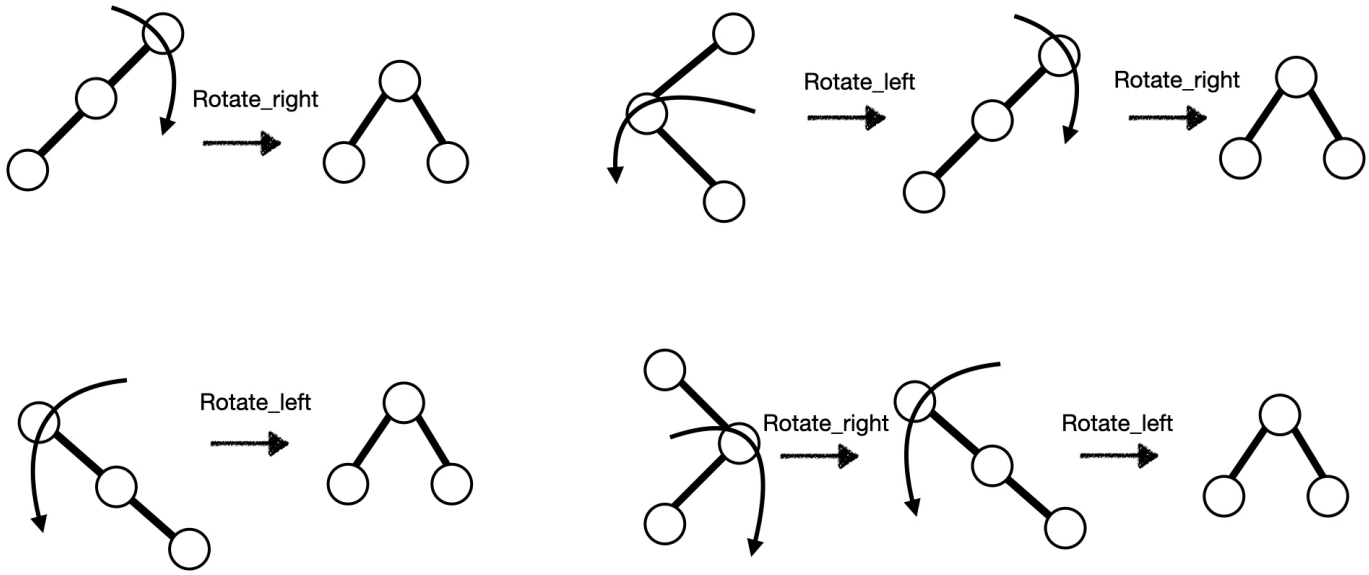
Remove



- 删除后只有被删除节点的父亲可能失衡，因为假如删除会引起失衡，那必然是删除了较短的分支，而父亲的高度是由较长的分支所决定的

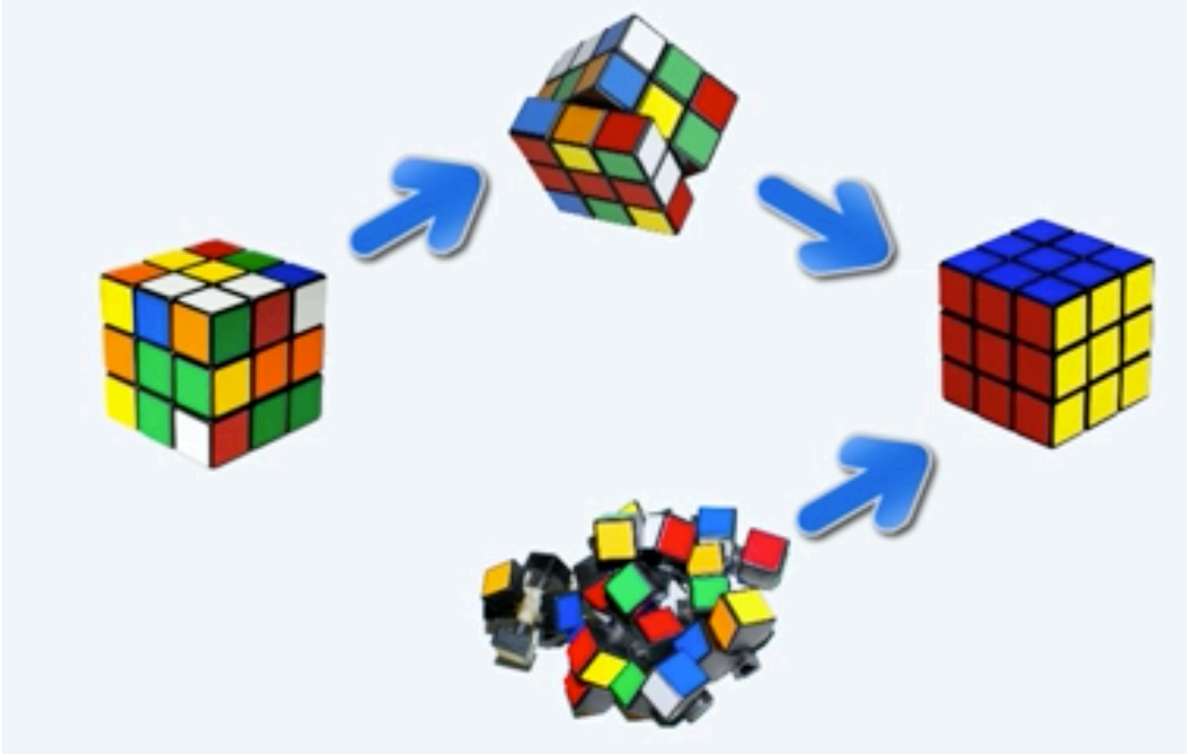
- 修复一个节点，上一个节点高度被改变，平衡被破坏。
- 利用左旋右旋操作进行恢复(rebalance)，修复后更新高度，有可能导致上层节点失衡，再次应用同样的手段修复，最多进行$O(\log N)$次

Summary



Lower the node is, “weighted” it will be, so we use rotation to lift it up

When writing the codes of rotation, we try to be as efficient as possible, connect the nodes directly!



- 将3个节点按中序遍历顺序排列a, b, c
- 将4棵子树按中序遍历顺序排列T_1, T_2, T_3, T_4
- 将他们拼接成一个中序遍历序列$T_1, a, T_2, b, T_3, c, T_4$
- 直接用该序列恢复成最终结果