II. Red-Black Trees, B+Trees(And B-Trees*)

- Build B-Trees*, Never add new leaves at the bottom !
  - Insert
  - Delete** (especially in 2-3 Trees)
    - Firstly talk about how to fill empty boxes ANYWHERE (As Lemma), not just in the leaves
      - Lemma 1. the empty node’s adjacent sibling has multiple keys.
      - Lemma 2. siblings on the right all have one key, but parent has two
      - Lemma 3. The parent and all siblings have only one item.
    - Deal with the leaves
      - Case 1: Use Lemma 1
      - Case 2: Use Lemma 2
      - Case 3: Use Lemma 3 and Lemma 1
  - Summary
- Red-Black Trees
  - Representing Keys in B-Tree Node through Color*
  - Red-Black Trees (Rudolf Bayer 1972)
  - Insertion
    - case 1 父节点为黑
    - case 2 父节点为红，且父节点的兄弟节点也为红
      - case 2.1 祖父节点为 root
      - case 2.2 祖父节点不为 root
    - case 3 父节点为红，且父节点的兄弟节点为黑
      - case 3.1 插入节点大于父节点
      - case 3.2 插入节点小于父节点
  - Deletion
    - Case I: 兄弟节点（W）为黑且 W 的两个儿子节点为黑
    - Case II: W为黑且 W 的右儿子为红

> 本文大量参考了 CS61B 中的相关内容，特别是借鉴了某种 Red-Black Trees (LLRBT) 可以等价为一个 B-Trees 的观点，而我们课上的 B+Trees, 其实与 B-Trees 有点区别. 所以将先介绍 B-Trees, 然后介绍Red-Black Trees, 最后介绍 B+Trees. 另外CS61B中的对于阶的定义与课内有差异，本文中的阶的定义为: Tree中任意一个节点允许的最多孩子数。

Build B-Trees*, Never add new leaves at the bottom !

> 如果怕搞混这一部分可以不看，B-Trees 和 B+Trees 还是有区别的

如果插入操作不会改变原有树的叶子节点的结构，那么显然能递推的得到非常平衡的树, B-Trees就是基于这个想法创造的，其在两种特定场景中最为常见:

- 低阶情况（L=3或L=4）：用作概念简明的平衡搜索树
- 超高阶情况（如L达到数千）：实际应用于数据库和文件系统（即处理超大规模记录的系统）

那我们应该怎么操作呢?

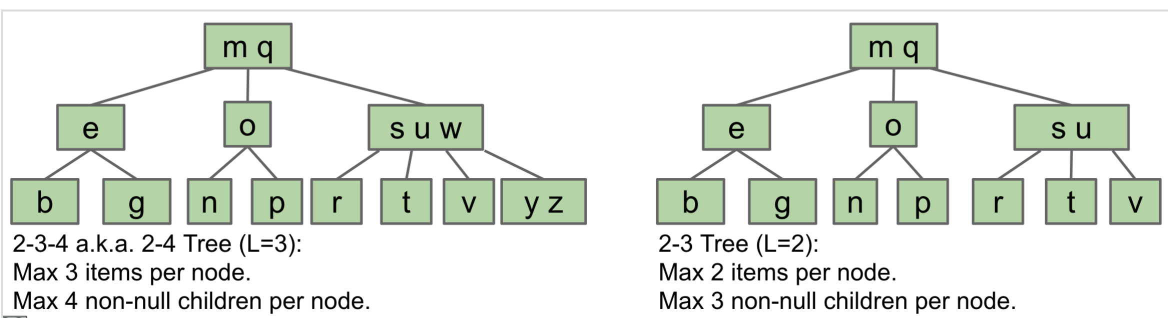
Insert

```
# For a general B-tree of order M
Btree Insert ( ElementType X, Btree T )
    Search from root to leaf for X and find the proper leaf node;
    Insert X; #先插入到叶子节点

    #判断每个节点内的 key 数量是否达到 M
    while (this Node has M keys) {
        split it into 2 nodes with ⌊M/2⌋ and ⌈M/2⌉ keys, respectivel
        if ( this node is the root )
            create a new root(usually the smallest key of the right
            with the split two children;
            #如果是根节点，那么会重新增高 1，这也是B-Trees唯一高度增加的情况
        else
        Simply send the smallest key of the right node to its parent
        check its parent;(Or this node = its parent)
    }
```

基于这样的设计，B-Trees 有很好的性质：

- All leaves must be the same distance from the root.
- A non-leaf node with k items must have exactly k+1 children.



> 这里左图应该是4阶，而右图应该是3阶

Delete** (especially in 2-3 Trees)

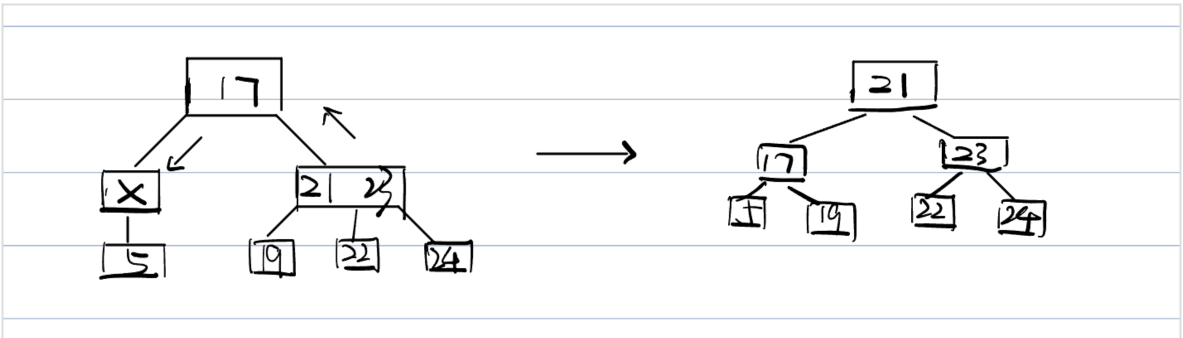
对于一个BST而言，Deletion 是非常容易的

- 删除的是叶子结点：直接删除
- 否则找左子树中最大的 Or 右子树中最小的，swap 两个节点的值然后删除叶子节点即可

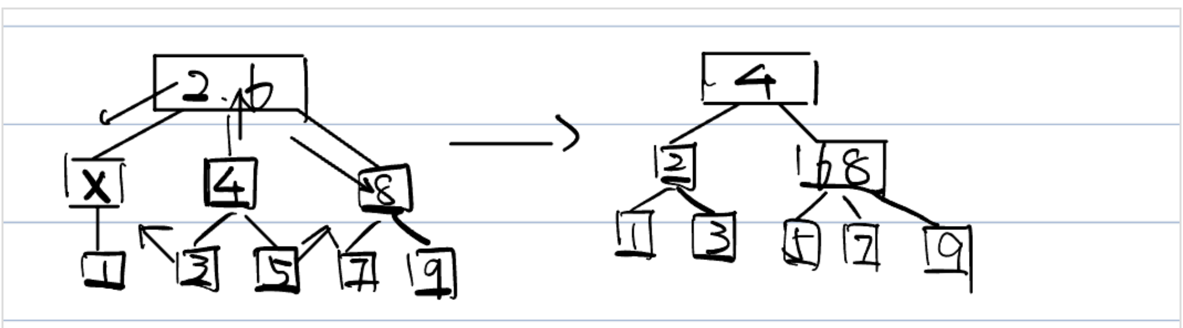
但是由于需要维护 B-Tree 的性质，我们不得不讨论一些特殊情况的处理方式，不过我们还是可以把 B-Trees 的删除规约到删除 B-Trees 的叶子节点。并且如果我们目标的叶子结点有多个 keys，可以直接删除。综上我们主要讨论删除单个叶子节点的情况

Firstly talk about how to fill empty boxes ANYWHERE (As Lemma), not just in the leaves

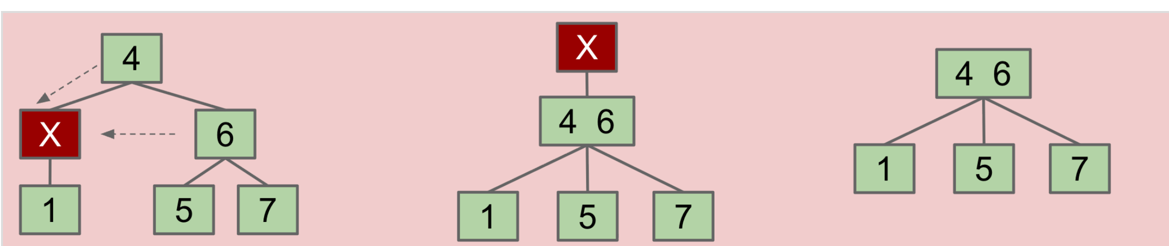
Lemma 1. the empty node's adjacent sibling has multiple keys.



Lemma 2. siblings on the right all have one key, but parent has two

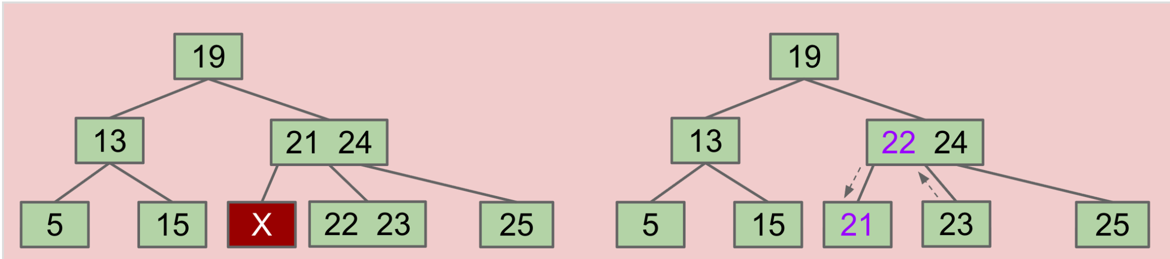


Lemma 3. The parent and all siblings have only one item.

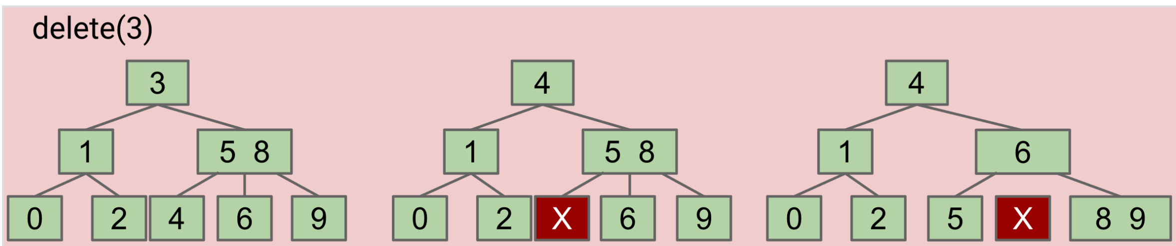


Deal with the leaves

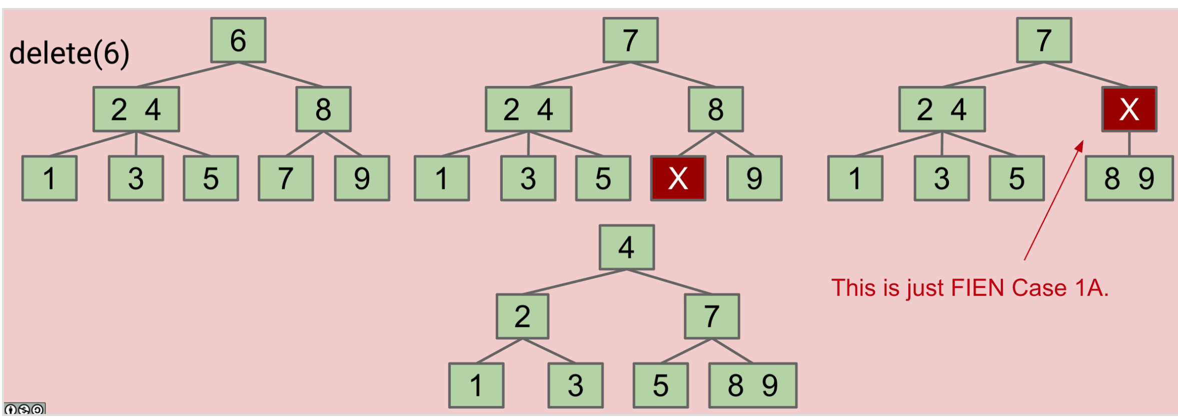
Case 1: Use Lemma 1



Case 2: Use Lemma 2



Case 3: Use Lemma 3 and Lemma 1



Summary

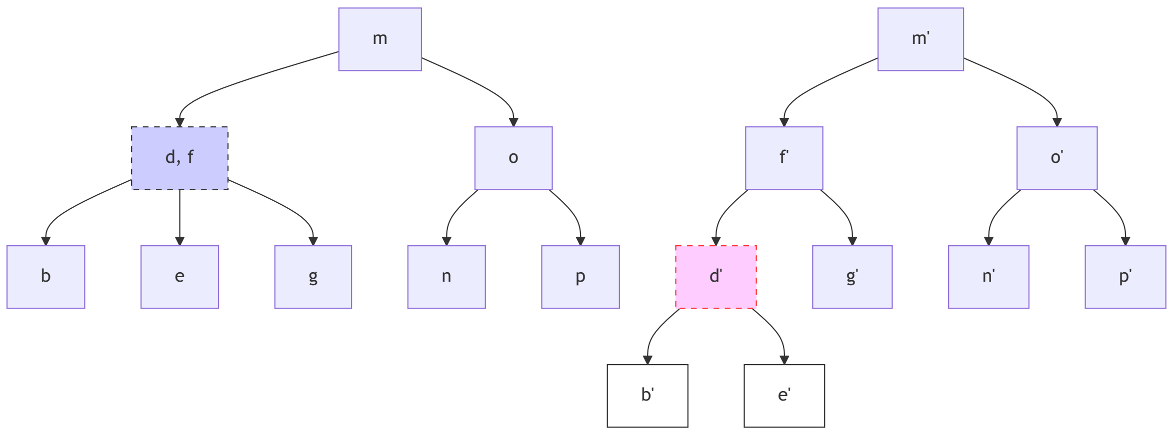
对于B-Tree, 其 Insert 和 Deletion 都是可行的，但是当尝试打其代码的时候就会发现十分困难。换而言之，这是一个好的算法，但并不实用

> B-Trees for small L, e.g. 2-3 trees and 2-3-4 trees, are a real pain to implement, and suffer from performance problems. Issues include: Maintaining different node types, Interconversion of nodes between 2-nodes and 3-nodes, Walking up the tree to split nodes.
>
> 在我看来，下面两个数据结构其实是对 B-Tree 可编程性的优化

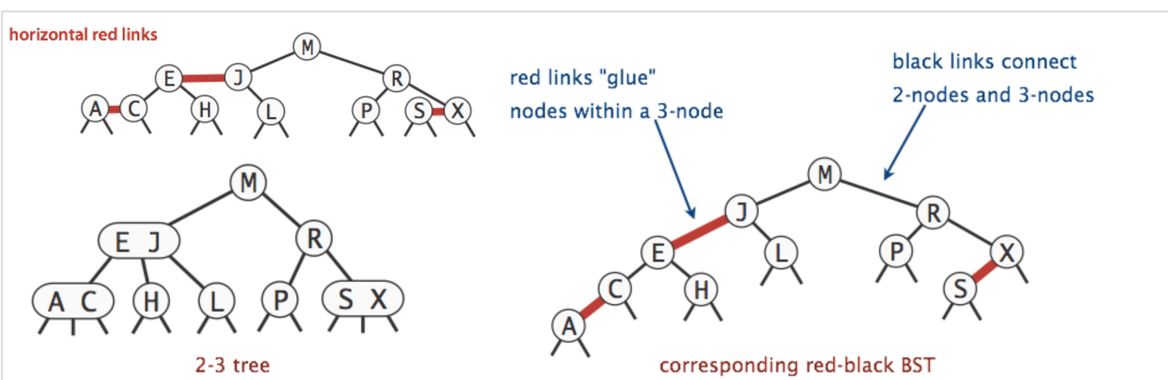
Red-Black Trees

Representing Keys in B-Tree Node through Color*

> 这一部分将从说明Red-Black Trees与2-3 B-Trees 关系出发，但是跟算法本身关系其实不大。



上图中左边是一个 2-3 Tree，而右边毫无疑问是一个 BST。我们是怎么建立起两者的一一对应关系的呢？ 对于 2-3 Tree 中含有 2 个 key 的节点，我们将其中一个染红，并作为另一个key的左子节点，而原先这个节点的子节点遵守 BST 的性质实现分配。据此我们就实现了从 2-3 B-Tree 向 BST 的转化。等介绍完 Red-Black Trees的性质 之后很容易发现，这个 BST 是一个 Red-Black Tree. 并且很容易某一类 Red-Black Tree(Left-Learning Red-Black Tree) 与 2-3 B-Tree 的一一对应



> 也可以用边着色，但是本质是一样的

Red-Black Trees (Rudolf Bayer 1972)

【定义】红黑树是一种满足以下红黑性质的二叉搜索树:

1. 每个节点要么是红色，要么是黑色
2. 根节点是黑色的 (这是很重要的一个点)
3. 每个叶子节点（NIL）都是黑色的
4. 如果一个节点是红色的，那么它的两个子节点都是黑色的
5. 对于每个节点，从该节点到其所有后代叶子节点的路径上，均包含相同数量的黑色节点

> 乍一看其实很复杂，这也是为什么我想引入 R-Trees 的原因，如果从 R-Trees 染色转换而来的话所有的性质都自动满足。此外对第 3 点做下说明：这是由于某些情况添加NIL叶子节点会比较清晰而已，不用在任何时候刻意考虑

【Definition】 The black-height of any node x, denoted by bh(x), is the number of black nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with N internal nodes has height at most $2\ln(N+1)$.

> 这个引理很容易对树高 (而不是黑高) 归纳证明，但是如果从 2-3 B-Tree 的角度看会更加显然

Insertion

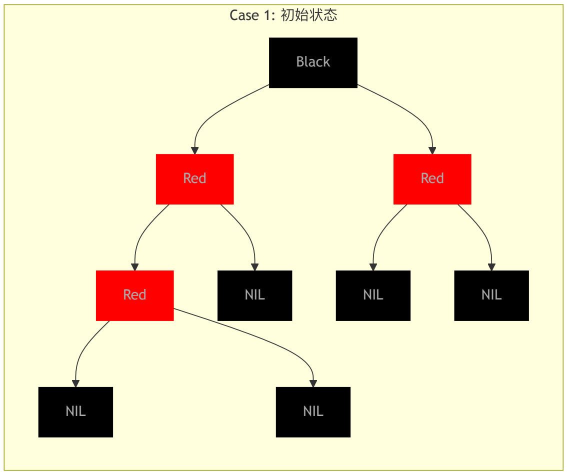
首先为了试图让整个树的黑高不发生变化，我们不妨先假设插入节点是红色。

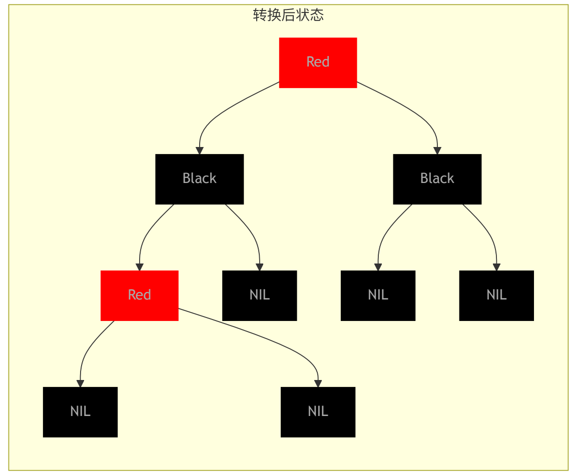
case 1 父节点为黑

那么不需要做任何操作

case 2 父节点为红，且父节点的兄弟节点也为红

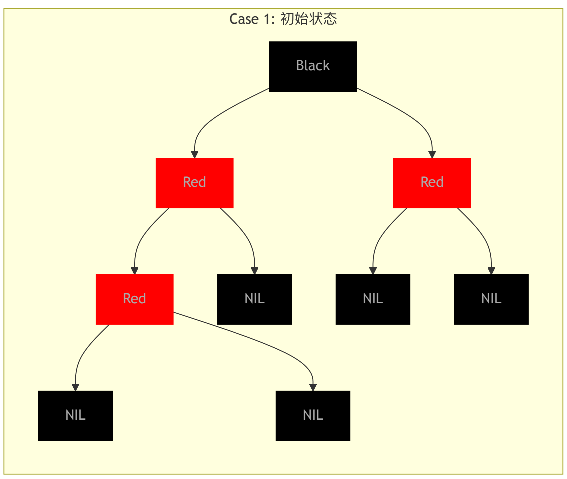
case 2.1 祖父节点为 root

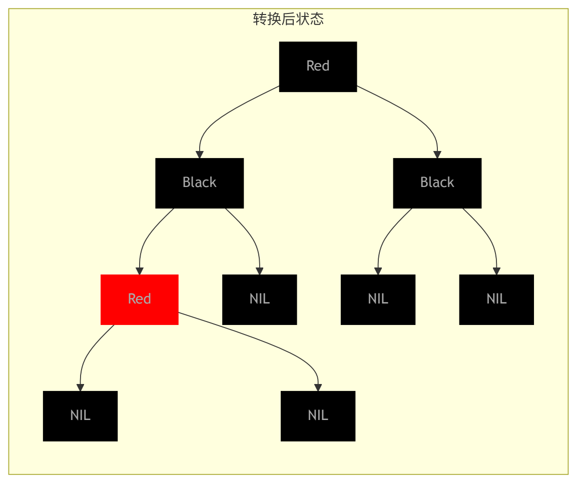




如图，相当于把Red传递给了祖父节点，由于祖父节点不是root，还将继续递归

case 2.2 祖父节点不为 root



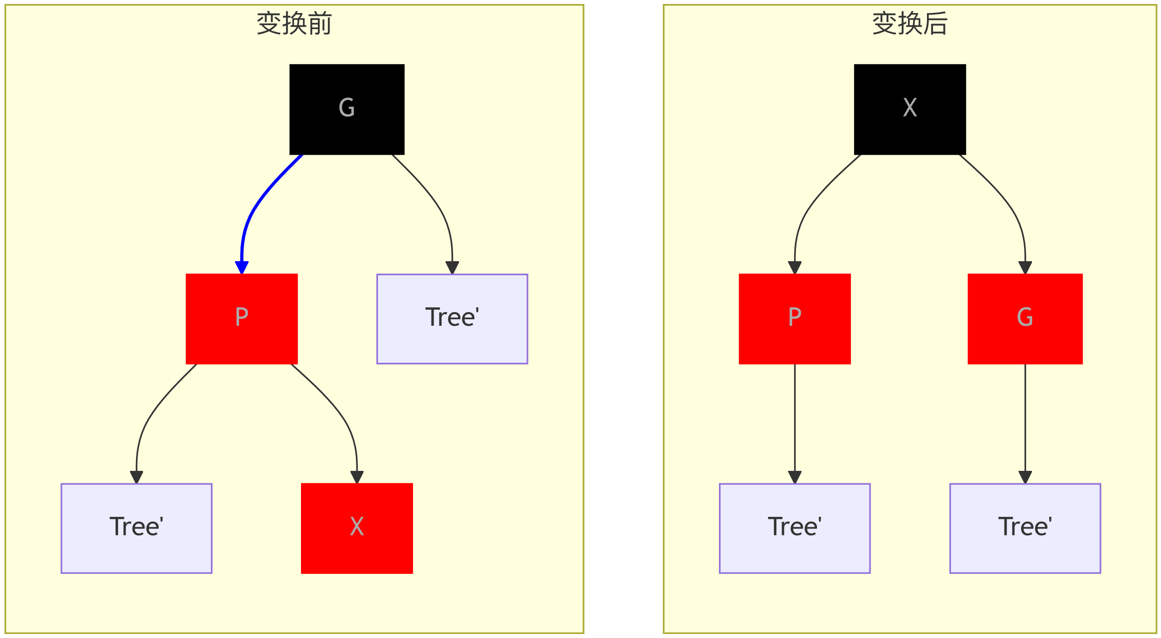


这种情况下祖父节点为root，由于约定 root 一定是黑，所以此处将其染黑即可。这种情况是唯一 Red-Black Trees 黑高增加的情况。

> 注意到 Red-Black 黑高增加发生在树顶，这同样与 B-Tree 一致, 以及null节点其实可以省略

case 3 父节点为红，且父节点的兄弟节点为黑

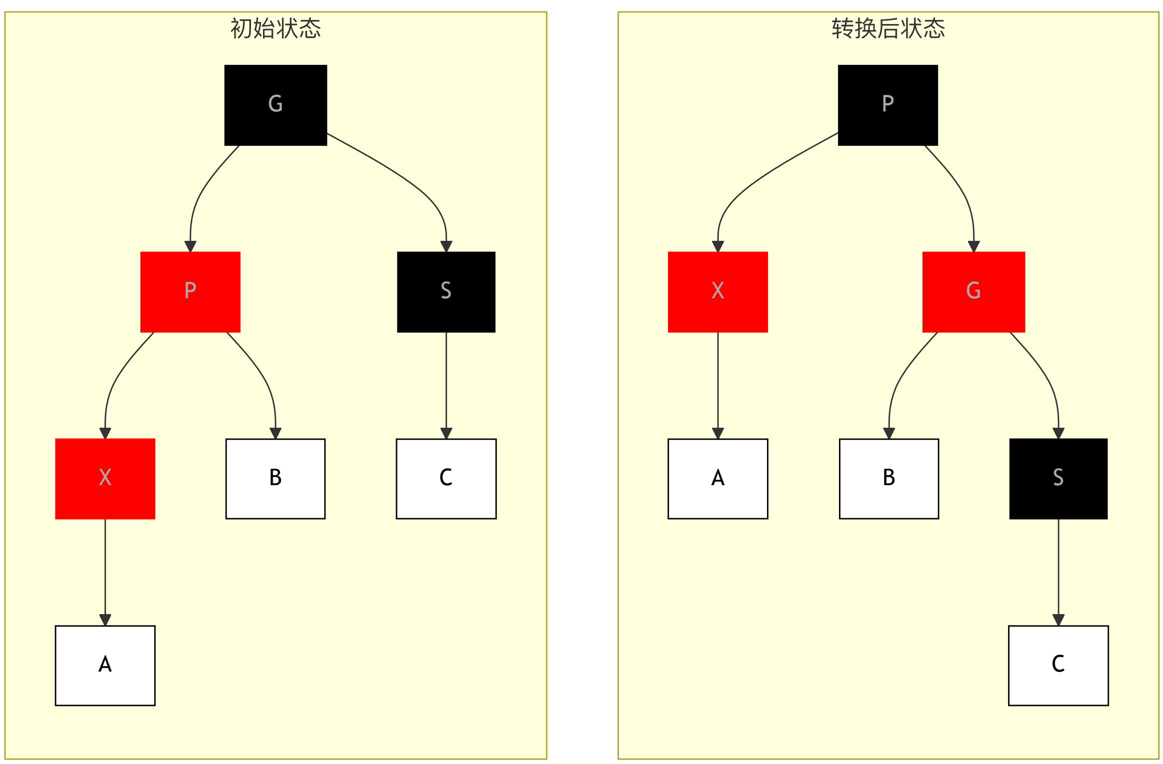
case 3.1 插入节点大于父节点



还是熟悉的两次 Double-Rotation 先 X-P，再 P-G 不过需要交换颜色

完成之后Insertion可以立即停止

case 3.2 插入节点小于父节点



注意这里只需要一次 P-G 的 Single Rotation 即可，完成后整个insertion可停止

Deletion

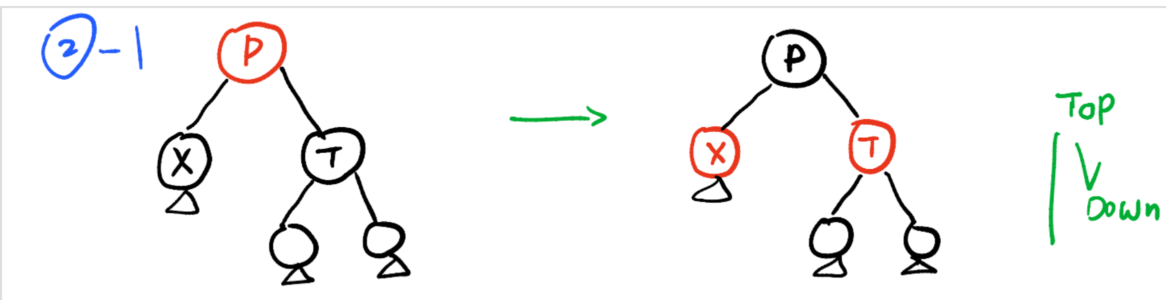
同样的，借鉴 BST 中删除的思路，这里也将仅讨论删除叶子节点的情况，并且下面给出的情况讨论不是并列的，而是类似 `if, else if` 的有顺序的讨论

> 有一说一zgc老师竟然能讲清楚，yyds！！！~~懒得画图就盗老师的子~~Org

若要删除的叶子节点是红色，那么直接删除即可(这显然不会更改黑高)，因此应该试图将 **红色传递给该叶子节点**

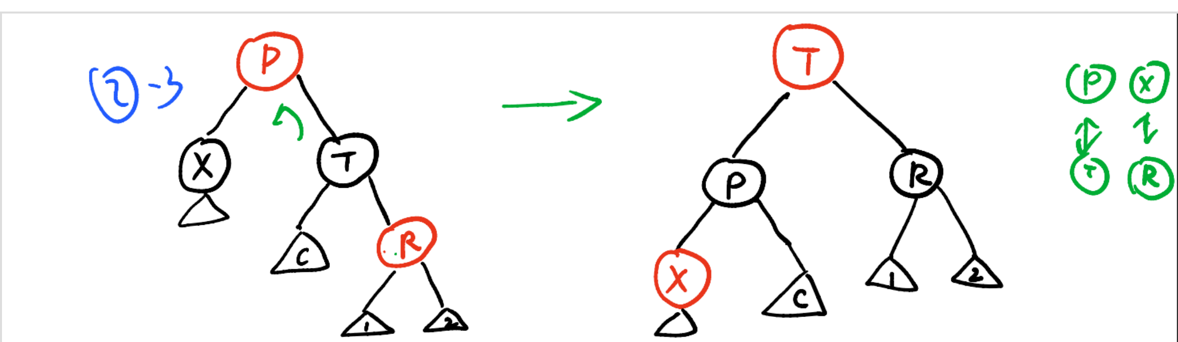
那么我们希望能在所找叶子节点(记为 X)的父节点，祖父节点，祖父的父节点……中能有红节点，据此我们先解决父节点为红的 3 种 case(I, II, III)

Case I: 兄弟节点（W）为黑且 W 的两个儿子节点为黑



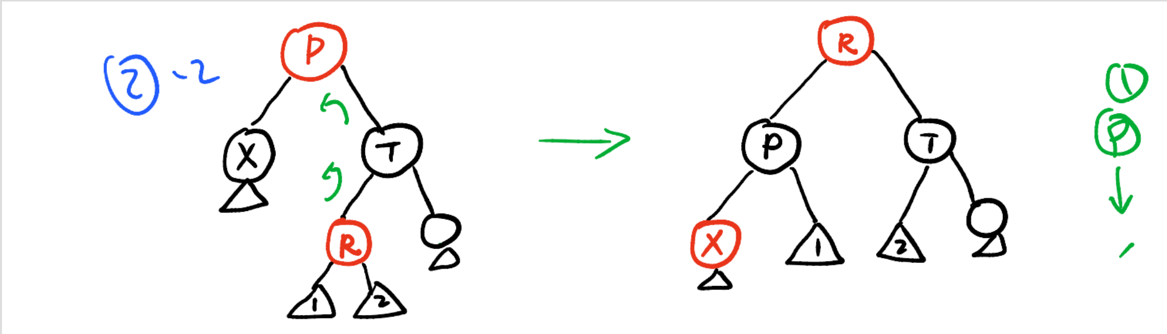
> 上图的变化其实就是 a变黑，然后X，W变红。仍然满足条件，并且红向下传递了1

Case II: W为黑且 W 的右儿子为红



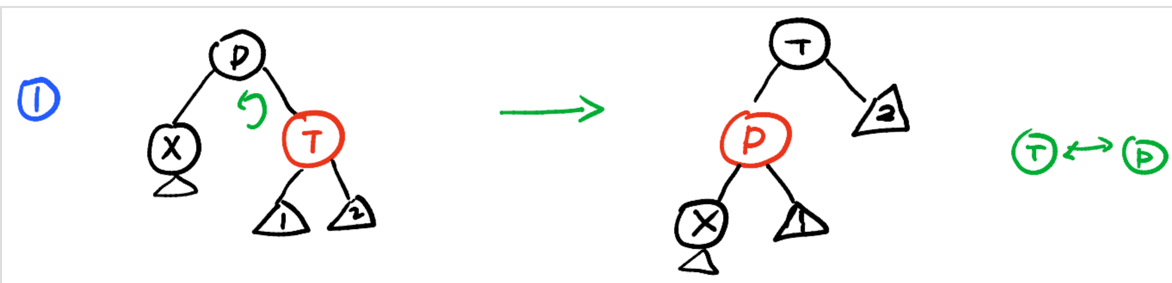
> 一个 P-T 的 single-Rotation 并且更改相应的颜色。

Case III: W为黑且 W 的左儿子为红



> R-T Rotation 然后 R-P Rotation 并更改相应颜色

至此，只要根节点到 X 的路上有一个红节点即可将红色传递给X
那么如果没有红色节点怎么办，或者怎么找那个最近的红节点？



可以在回溯的过程中先检查是否有红色，此时不妨检查一下其兄弟节点是否为红，如果为红，那么也不失为一个办法，上图就是干的这件事情
但是，如果一路黑到 root (上面所有情况都没发生) 了呢？ **其实只要把 root 变红即可再重复上面的操作即可。这和Insert时的思路一样，红黑树的定点在不造成连续两红的情况下其实是可以自由变换颜色的，因为其黑高会同步影响所有点**

B+ Trees

B+ 树 (Bayer and McCreight 1972)

> 具体定义其实是有点争议的，我是参考wiki百科的，跟ppt上其实有区别

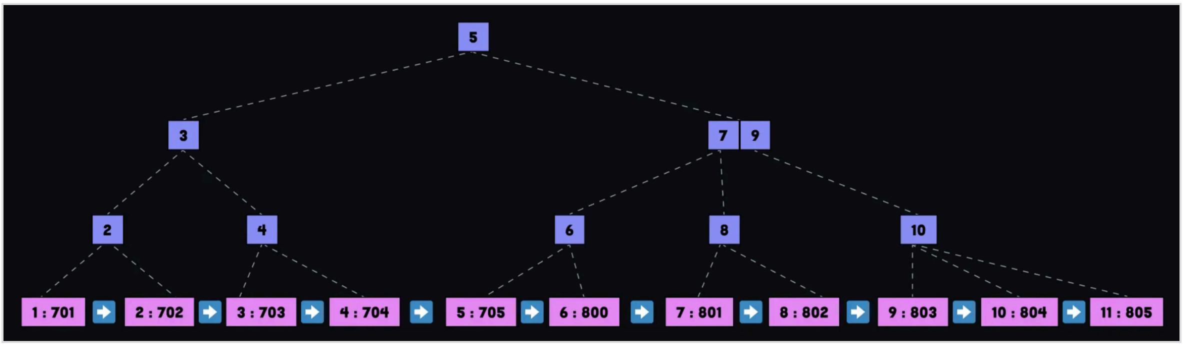
【定义】一棵阶为 M 的 B+ 树是具有以下结构特性的树：
(1) 节点要么是叶子节点，要么拥有 2 到 M - 1 个子节点

(2) 所有非叶子节点（根节点除外）拥有 ⌈M/2⌉ 到 M - 1 个子节点

(3) 所有叶子节点都位于同一深度

(4) 叶子节点中能存储的最大个数有争议，Wiki百科中为 M-1, 而ppt中为 M

> 似乎在实际应用时，叶子节点的 value 在内存空间中具有相互访问的性质。并且只有叶子节点存储 key-value 对，nonleaf 节点只存储 key,如下图所示



Insertion

```
Btree  Insert ( ElementType X,  Btree T, Wiki Definition version)
{
        Search from root to leaf for X and find the proper leaf node;
        Insert X;  # the node now has N keys

        while (N >= M) {
                split it into 2 nodes with ⌊N/2⌋ and ⌈N/2⌉  keys, re
                if (this node is the root)
                        create a new root with two children;
                check its parent;
        }
}
```

> 还是很简单的算法, 思路和 B-Tree 基本一致, 有两个区别:
>
> 1. 会将所有key存储在叶子节点中，而 nonleaf 节点只用来存储用来区分的标识(是叶子结点中的 copy)。
> 2. 在节点超额之后只会对 key 进行操作, 且操作时不会删除叶子节点中的 key

Deletion*

> 由于 B+Tree 的删除课上不会涉及，这里介绍的是数据库中定义的删除

首先可以找到需要删除 key 所在的节点 node，并删除这个 key

- 如果这个 key 被删除之后，该 node 和他的一个兄弟节点中的 key 可以在一个 node 中放下，那么 merge 这两个节点，删除右边那个节点。一直向上递归。
- 如果这个 key 被删除之后，该 node 和他的兄弟节点仍然无法放在一个 node 中，那么便向
  其任意一个兄弟节点借一个元素来满足，并向上递归
- 如果递归到根节点，使得根节点只有一个子节点，那么直接把根节点 merge 到子节点即可

> Assume record already deleted from file. Let V be the search key value of the record, and Pr be the pointer to the record.
>
> • Remove (Pr, V) from the leaf node
>
> • If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then merge siblings:
>
> • Insert all the search-key values in the two nodes into a single node (the one on the left, and delete the other node.
>
> • Delete the pair (Kr1, Pi), where P, is the pointer to the deleted node, from its parent, recursively using the above procedure.
>
> • Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then redistribute pointers:
>
> • Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
>
> • Update the corresponding search-key value in the parent of the node.
>
> • The node deletions may cascade upwards till a node which has [n/2] or more pointers is found.
>
> •If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Summary

主要想强调这三个算法的根本思想是: **插入永远让树高增加发生在树顶 (红黑树中指黑高), 这样天然就实现了高度的 balance, 而删除同样也希望树高的减少发生在树顶**

据此，B-Tree 和 B+Tree 在 Insertion 时的做法就是通过提高树的 Order, 把破坏 Order 的节点传递给父节点，而若传递到根节点，则完成最后一次分裂，树高完成加 1；在 Deletion 时的做法先规约到可能会影响树高的叶子节点删除问题，再引入 Deletion 空节点的引理来解决问题(B+Tree 可以调整叶子节点容量的定义来使得只要处理 merge)

红黑树的做法则是定义黑高，将 root 置黑，然后选择插入红节点。Insertion 的思路就是向上传递红色节点而不改变黑高，如果传递到根节点那么直接染为红色，此时树高加一；Deletion 的思路同样是先规约到删除黑色叶子节点的问题，然后试图在 root 到该叶子节点的 path 上找到一个红色节点并传递下来，如果找不到，那么将根节点染红，此时树高减一;