* 旋转的力量——Splay Tree 初探

南京大学 杜星亮

* 目前我们接触到的数据结构：
  * 数组
  * 栈，队列
  * 树，堆
  * **二叉搜索树**
  * ……
* 多用于维护一段**序列**或一个**集合**

*许多数据结构的本质是序列

* 大家可能会说：
* ……
* 好麻烦？
* 不好写？
* 难调试？
* 不灵活？
* ……

* 平衡二叉搜索树?

* C++ STL 中自带 红黑树实现的 set, map
* 很是方便
* 但其封装过于严密，难以扩展
* 只能处理较为简单的序列与集合操作
* 面对复杂的操作要求
* 手中只有 set 和 map 往往只能望而却步

*一个悲剧

* 首先是性能问题
  * 广义上的平衡二叉树种类很多
  * AVL, RB-Tree, Treap 等等
  * 性能上，它们都能满足要求
* 如果考虑实现复杂度
  * 传统 AVL 和 RB-Tree 的实现都需要众多的情况讨论
* 如果考虑灵活性
  * 等等？什么是灵活性？
* ……

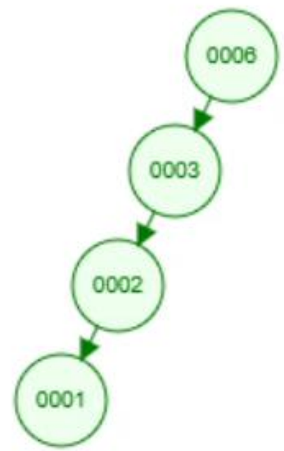
自己实现 BST

*第一印象

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Splay Tree

Insert Delete Find Print



Animation Completed

*旋转，旋转，再旋转？

除了旋转以外，还有什么？

*旋转到 根

* 缓存性质！
  * 最近访问过的结点会在离根较近的位置
* 为什么要这么做？

*Splay Tree，从何而来?

* Splay Tree 由 Daniel Dominic Sleator 和 Robert Endre Tarjan 发明
  * (1985) “Self-Adjusting Binary Search Trees”

*动态树结构！

* 但其实
* (1983) “A Data Structure for Dynamic Trees”
* 他们已经使用与 Splay 基本相同的数据结构来维护动态树结构
* 他们提出的维护动态树结构的方法中，经常需要**将BST中某个结点旋转到根**的操作

*动态树结构！

* 对于BST中的插入，删除，查询等等操作
* 执行结束后
* 都将对应节点通过一些旋转操作移到树根

* 插入，删除，查询等操作的实现为传统实现

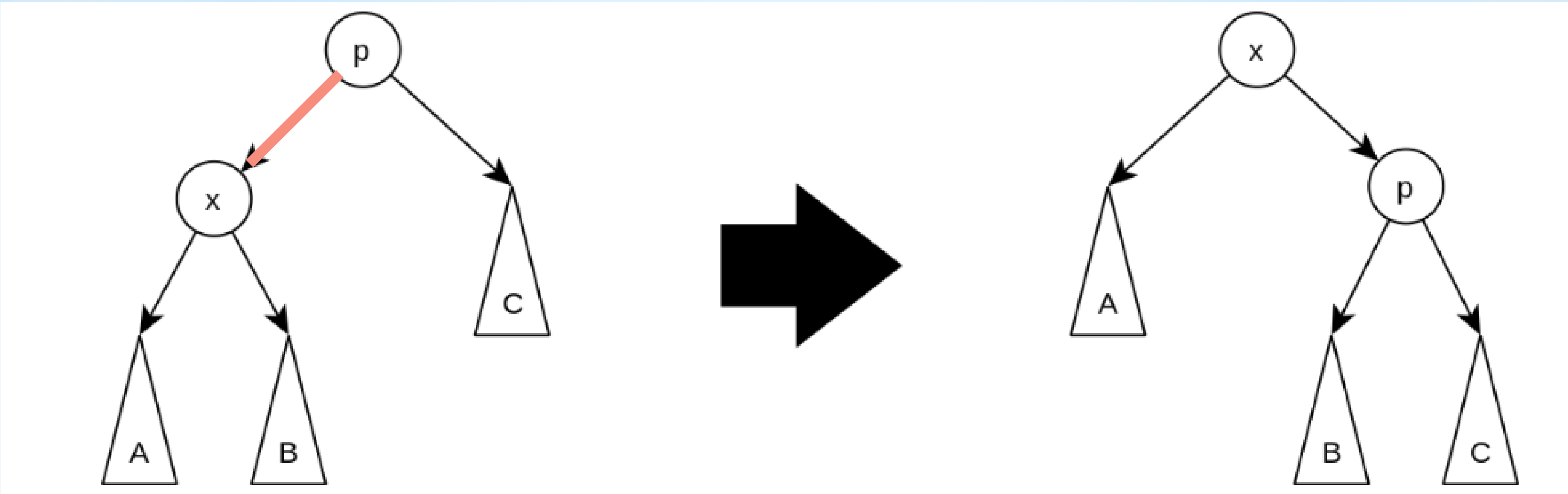
*核心思想：自调整

* 定义对某个结点 X 的操作
* Splay(X)
* 通过一些旋转对树进行调整
* 使 X 成为树根

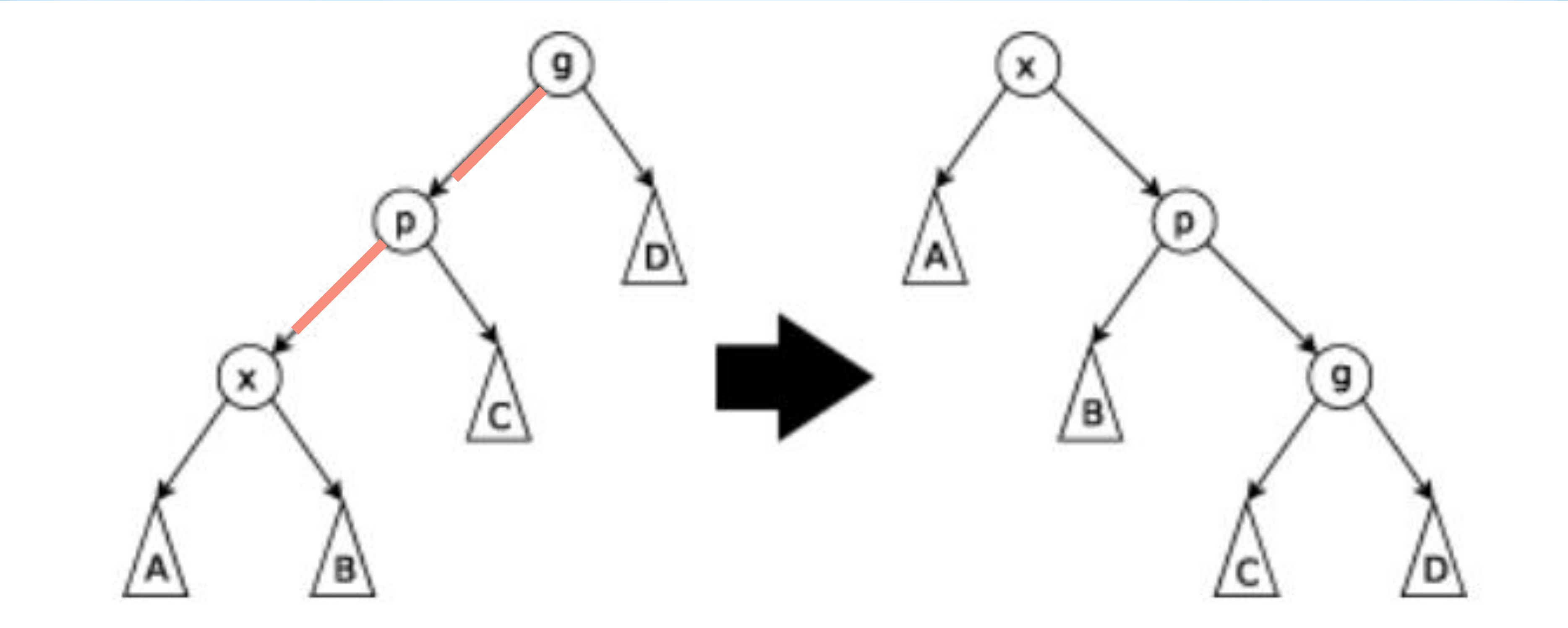
*核心操作：Splay

* 其旋转操作分三种情况
* 以下设
* 待旋转结点 X
* X 的父结点 P
* P 的父结点 G

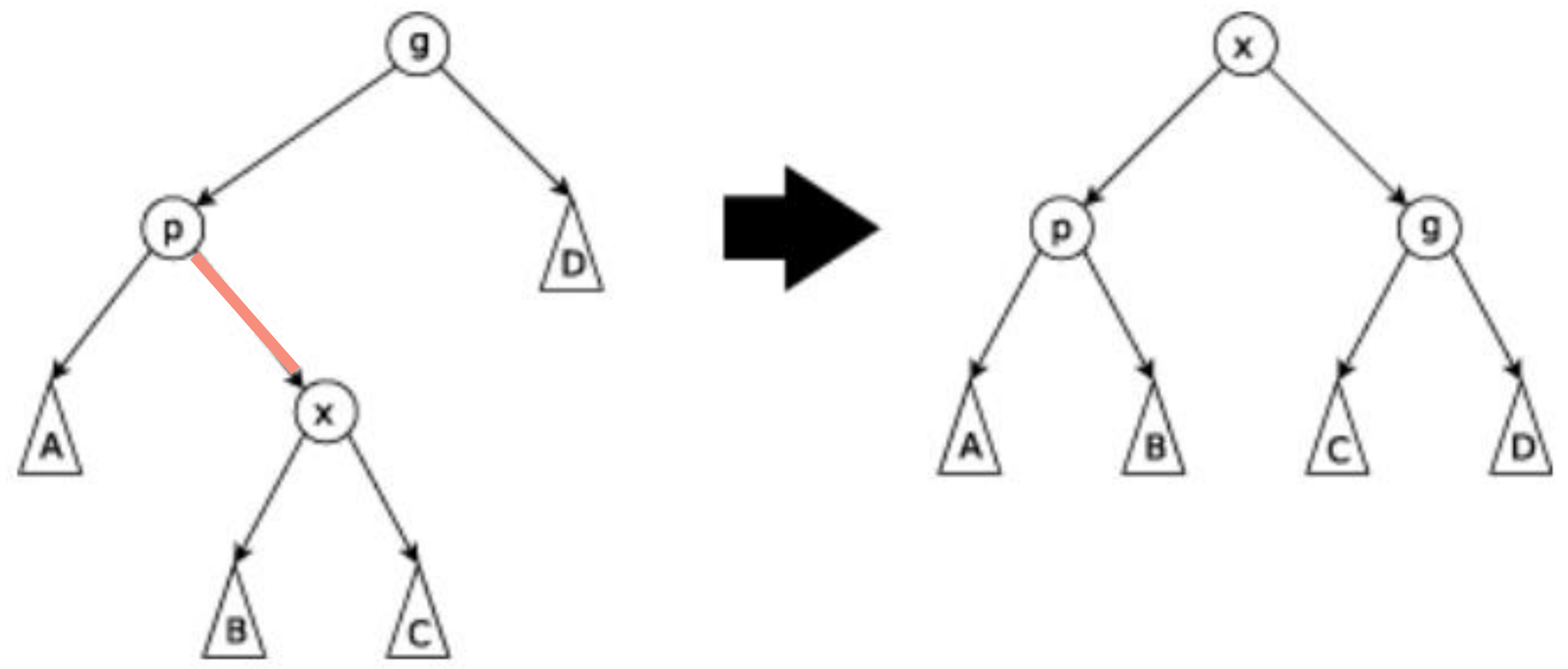
*实现



Zig



Zig-Zig



*Zig-Zag

*简要时间复杂度分析

* 以下为均摊分析

* 设 结点 X
* Size(X) 为 以 X 为根的子树中结点总数
* $\mathrm{rank}(x) = \log_2(\mathrm{size}(x))$
* 定义在 Splay 树组成的集合上的势能函数
* $\varphi(Tree) = \sum_{x \in Tree} rank(x)$

* 首先，计算势能函数变化
* Zig 操作

$\Delta\Phi$	$= \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$	[since only p and x change ranks]
	$= \text{rank}'(p) - \text{rank}(x)$	[since $\text{rank}'(x)=\text{rank}(p)$]
	$\le \text{rank}'(x) - \text{rank}(x)$	[since $\text{rank}'(p)<\text{rank}'(x)$]

* 简要时间复杂度分析

*Zig-Zig 操作

$$\Delta\Phi = \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$$

$$= \text{rank}'(g) + \text{rank}'(p) - \text{rank}(p) - \text{rank}(x) \quad [\text{since rank}'(x)=\text{rank}(g)]$$

$$\le \text{rank}'(g) + \text{rank}'(x) - 2\,\text{rank}(x) \quad [\text{since rank}(x)<\text{rank}(p) \text{ and rank}'(x)>\text{rank}'(p)]$$

$$\le 3(\text{rank}'(x)-\text{rank}(x)) - 2 \quad [\text{due to the concavity of the log function}]$$

*Zig-Zag 操作

$$\Delta\Phi = \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$$

$$\le \text{rank}'(g) + \text{rank}'(p) - 2\,\text{rank}(x) \quad [\text{since rank}'(x)=\text{rank}(g) \text{ and rank}(x)<\text{rank}(p)]$$

$$\le 2(\text{rank}'(x)-\text{rank}(x)) - 2 \quad [\text{due to the concavity of the log function}]$$

*简要时间复杂度分析

* 而每个操作的实际代价小于等于2
* 由计算总摊还代价可得
  * 参见《算法导论》 式17.3
* 将 X 移到根，总摊还代价有上界 $O(\log_2 n)$
* 故 m 次操作，总复杂度 $O(m\log_2 n)$

*简要时间复杂度分析

* 为使得势能函数始终非负
    * 参见《算法导论》 17.3 节
* 引入初态到终态最大势能差

$$\Phi_i - \Phi_f = \sum_x \mathrm{rank}_i(x) - \mathrm{rank}_f(x) = O(n \log n)$$

* 故总复杂度 $O(m\log_2 n + n\log_2 n)$

*简要时间复杂度分析

*为什么用 Splay Tree?

* 重新看看基本操作

我们现在有了 Splay 操作

* 查找?
  * 前驱? 后继?
* 插入?
* 删除?
* 查找排名?
* 查找某排名的元素?

* **分离**

BST 的拆分?

* 给定树和元素 X，得到两个新树
* 一个包含小于或等于 X 的所有元素
* 另一个包含大于 X 的所有元素。

* 将 X 旋转到根
* 其左侧的树包含小于 X 的所有元素，并且其右侧的树包含大于 X 的所有元素。
* 断开根与右孩子之间的边即可

* 连接

BST 的拼接?

* 给定两个树 S 和 T，且 S 的所有元素小于 T 的元素
* 可以很快地将两树合并：

* 将 S 中最大元素旋转到根
* 此时它有一个右孩子为空。
* 将其右孩子设为 T

* **再来看看基本操作**

我们现在有了 分离 和 连接 操作

* 截取一段排名连续的元素?
* 删除一段排名连续的元素?
* ……

* 查找，插入，删除
* 截取，分割，连接
* ……
* 到处都是 Splay

* 没有严格维护平衡的条件
* 只要每次都 Splay 一下

旋转的力量！

*客观看待

难道 Splay Tree 没有缺点吗？

优点

* 平摊复杂度优秀
* 缓存性质
* 结构灵活
* 维护树结构不需要额外空间

缺点

* 时间复杂度基于均摊，最坏情况复杂度目前无法保证
* 大多数操作都可能导致树的结构发生较大改变，不适合纯函数式编程与并发处理

*客观看待

References

* https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
* https://en.wikipedia.org/wiki/Splay_tree
* Sleator, D. D.; Tarjan, R. E. (1985). "Self-Adjusting Binary Search Trees"
* Sleator, D. D.; Tarjan, R. E. (1983). "A Data Structure for Dynamic Trees"

*谢谢大家