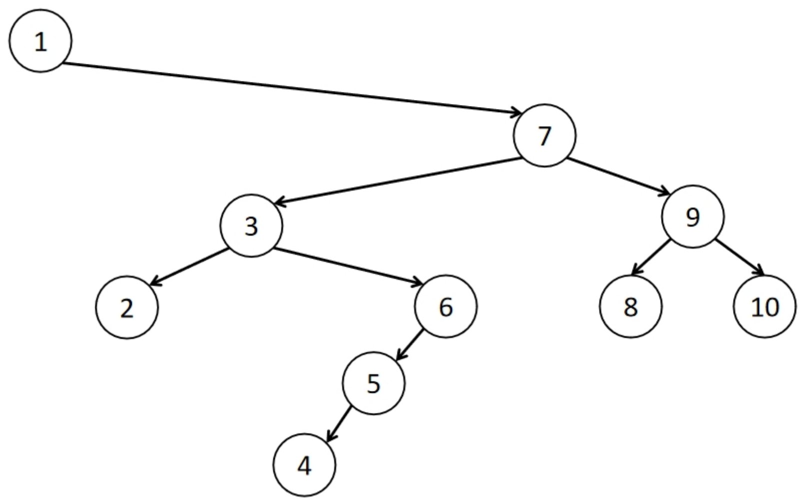
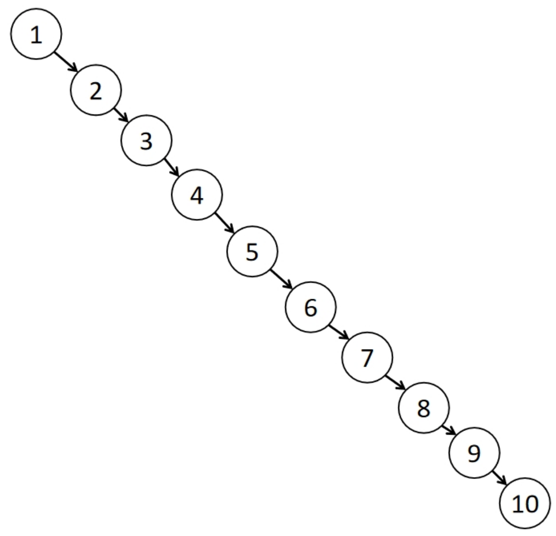
FHQ-Treap

一、简介

不同于经典的基于左旋、右旋的$Treap$，$FHQ-Treap$是基于分裂与合并的的一种$Treap$。虽然两者操作方式完全不同，但产生的结果是一样的。而且，$FHQ-Treap$具有 **好写**、**好理解**、**可持久化**、**区间操作** 等优点。

BST是二叉查找树的缩写，所有结点满足左子树所有点比自己小，右子树所有点比自己大，且形状不唯一，如下图所示，其中右图是一种极端的情况（**退化成链**）。



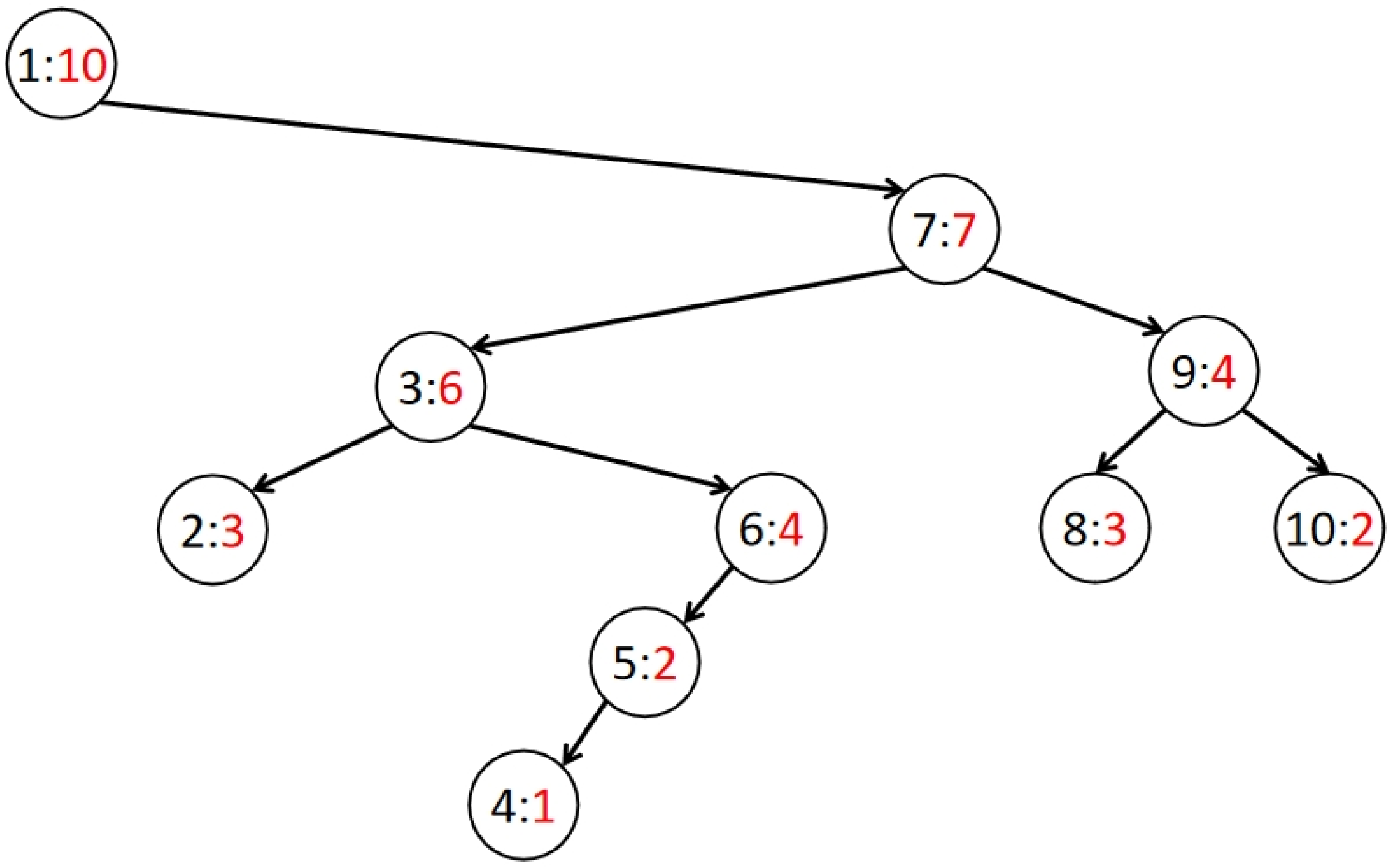


我们知道，一颗 BST 的时间复杂度是由树高决定的，由于 BST 在最坏情况下可能会退化成一条 **链** 导致较高的复杂度，因此单纯地用 BST 可能会超时。对于一组数据而言，构造出的 BST 是不唯一的，而$Treap$ 的思想正是让它尽可能地 **随机化**， 使其达到 **期望树高**，是一种**近似平衡**的平衡树。

$Treap$ 实际上是 $Tree$（二叉查找树）+$Heap$（堆）的缩写，其每个结点至少有如下两个属性：

1. 键值（key），满足 **左儿子键值 $\leq$ 该节点键值 $\leq$ 右儿子键值。**
2. 优先级（pri），满足 **左儿子优先级或右儿子的优先级 $\leq$ 该节点的优先级。**

我们需要在 **分裂** 操作与 **合并** 操作中根据这两点对 $Treap$ 进行维护，使得 $Treap$ 永远满足以上两条规则，而其他操作都建立在这两个操作上。简单来说，$FHQ-Treap$ 是基于 **分裂+合并** 的 **Treap**，如下图所示是其中一个 $Treap$，其中，黑色是权值（key），红色是优先级（pri）。为了使其随机化，这里的优先级均为**随机数**，若优先级互不相等，则能唯一确定该 $Treap$ 的形状，若出现个别相等也影响不大，目的只是让其形状足够随机。



我们定义一个结构体，记录 $Treap$ 上每个点的信息。

代码实现

```
struct Node {
    int ls, rs;//左儿子，右儿子
    int key, pri, size;//权值，优先级，树的大小
} treap[N];
int tSize = 0,root = 0;//总节点数，根
```

其中 `tSize` 是节点个数， `root` 是 $Treap$ 的根。这里使用了数组的方式存储 $Treap$。

三、$FHQ-Treap$ 的基本操作

结点更新（Pushup）

代码实现

```
void Pushup(int u) {
    treap[u].siz = treap[treap[u].ls].siz + treap[treap[u].rs].siz + 1;
}
```

分裂Treap（Split）

一般来说，$Split$ 是将 $Treap$ 分裂成两个 $Treap$，一个是所有结点的权值小于等于某值的 $Treap$，一个是所有结点的权值大于该值的 $Treap$，并且返回两个 $Treap$ 的根。

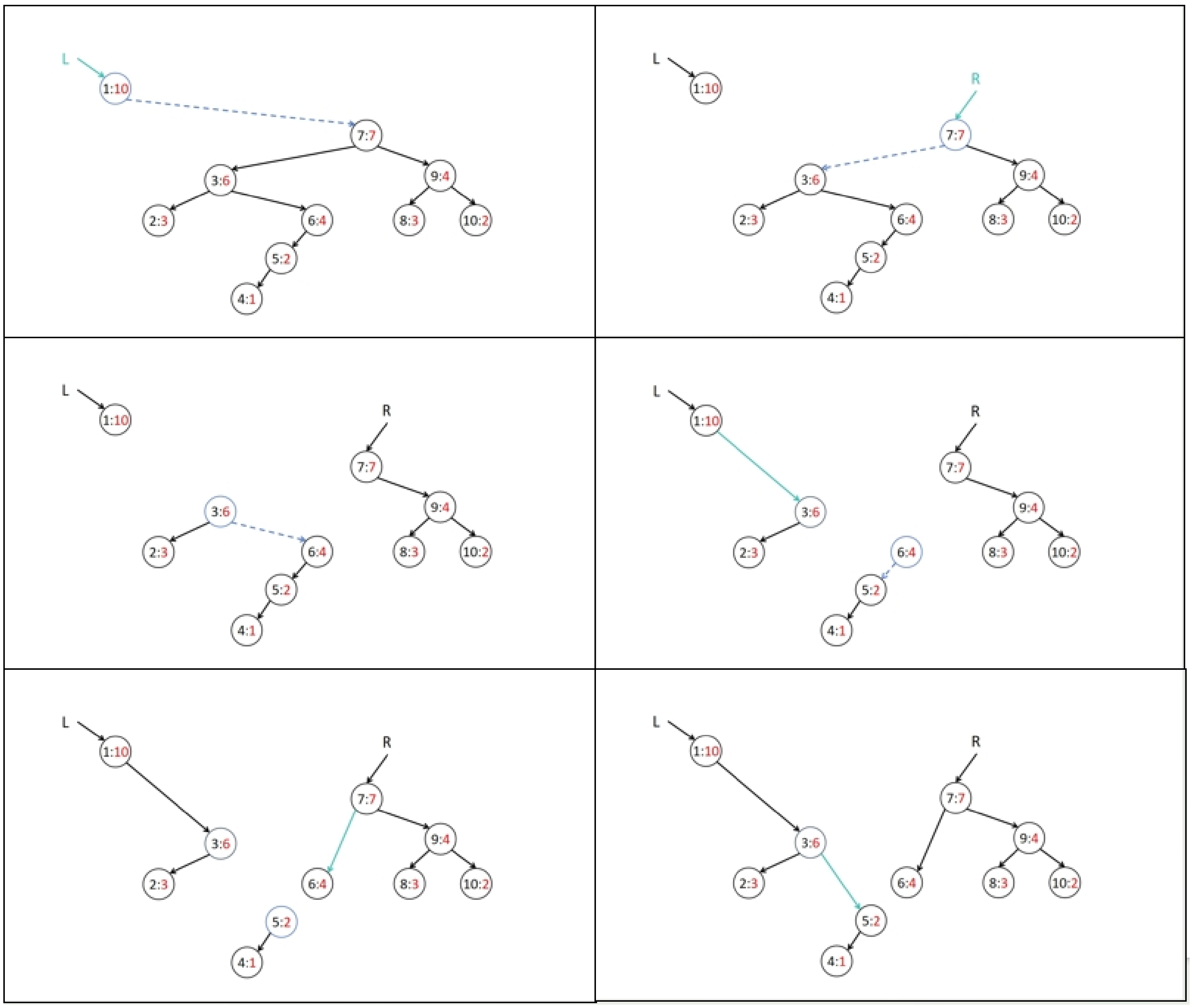
```
//u表示当前的结点，x表示分裂标准，L,R分别是分裂出的左右子树
void Split(int u,int x,int &L,int &R);
```

其中很妙的一点，L 与 R 都传递了它的地址，即可通过该操作修改某个 $Treap$ 的左子树、右子树信息或传递回分裂开的两个 $Treap$ 的编号。这一点在后面的应用中**十分重要**。

这里分裂的操作并不用到优先级，结点上的值都是键值，并且可以看出：

- 若当前枚举到的 $Treap$ 的根要**小于等于分裂标准**时，其**左子树都必然小于等于分裂标准**，只考虑右子树。
- 若当前枚举到的 $Treap$ 的根要**大于分裂标准**时，其**右子树必然都大于分裂标准**，只考虑左子树。

以 $x = 5$ 为例，分成两个 $Treap$ 过程如下图所示：



代码实现

```
//根为u的子树，以x分裂标准进行分裂，L,R分别是分裂出的两个Treap的根
void Split(int u, int x, int &L, int &R) {
    if (u == 0) {
        L = R = 0;
        return;
    }
    //如果根节点比x小，u一定是小Treap的根，并且更新u的右儿子，继续分裂u的右子树
    if (treap[u].key <= x) {
        L = u;
        Split(treap[u].rs, x, treap[u].rs, R);
    }
    else {
        R = u;
        Split(treap[u].ls, x, L, treap[u].ls);
    }
    Pushup(u);
}
```

由于 $Split$ 的时间复杂度就是树高，所以它的期望时间复杂度为 $O(\log n)$。

合并Treap（Merge）

$Merge$ 要满足合并的两个 $Treap$ （用根 L 与 R 表示）， L **中的值都小于等于** R 的条件。

合并的时候，就要考虑优先值了。先比较两颗 $Treap$ 的根节点的优先值大小。如果 L 大，把 R 合并到 L 的右子树上去，否则，把 L 合并到 R 的左子树上去（因为 L 中的值都小于 R）。最终，$Merge$ 要返回合并后的根。如下图所示：

代码实现

```
//合并L,R为根的子树并返回根
int Merge(int L,int R) {
    if (L == 0 || R == 0) return L + R;//如果是链结点或叶结点则无须合并
    //维护堆的性质，如果L大，把R合并到L的右子树上
    if (treap[L].pri > treap[R].pri) {
        treap[L].rs = Merge(treap[L].rs, R);
        Pushup(L);
        return L;
    }
    else {
        treap[R].ls = Merge(L, treap[R].ls);
        Pushup(R);
        return R;
    }
}
```

插入权值为x的结点（Insert）

首先创建一个只有一个节点的 $Treap$，再把这个 $Treap$ 合并到大 $Treap$ 上。注意 **Merge 的性质**，所以要先分裂，以下方法是允许存在权值相同的点，若只考虑数量上的问题可以**先查询结点**，再将结点属性 `cnt` ，数量上加 1 即可，若不存在该结点，再考虑新增结点。如下图所示：

代码实现

```
//在以u为根的子树上，插入权值为x的结点，并返回操作完后的根
int Insert(int u,int x) {
    ++tSize;
    treap[tSize].siz = 1;
    treap[tSize].ls = treap[tSize].rs = 0;
    treap[tSize].key = x ;
    treap[tSize].pri = rand();
    int L, R;
    //以x作为标准
    Split(u, x, L, R);
    return Merge(Merge(L, tSize), R);
}
```

删除权值为x的结点（Delete）

这里假设只删除一个与查询的值相同的结点。先分裂出一颗全是要删除的值的 $Treap$ ，然后把这个 $Treap$ 的左子树与右子树进行合并（把根丢掉），实现该操作。

代码实现

```
//在以u为根的子树中，删除一个权值为x的结点，并返回操作完后的根
int Delete(int u,int x) {
    int L, M, R;
    Split(u, x, L, R);
    Split(L, x - 1, L, M);
    M = Merge(treap[M].ls, treap[M].rs);//把根去掉
    return Merge(Merge(L, M), R);
}
```

查询值为x的结点（GetNode）

查找值为 x 的对应节点，原理即二叉查找树一层层查找即可。

代码实现

```
//在以u为根的子树中，查找值为x对应的结点
int GetNode(int u,int x) {
    while (u) {
        if (treap[u].key == x) return u;//查找成功
        if (x < treap[u].key) u = treap[u].ls;
        else u = treap[u].rs;
    }
    return 0;//查找失败
}
```

查询最值结点（GetMin）

以查找最小值为例，最大值则反之，从子树根结点开始，如果左子结点不为空，则访问左子结点，直到左子结点为空，当前结点就是该子树的最小值结点。

代码实现

```
//在以u为根的子树中，查找最小值
int GetMin(int u){
    while (treap[u].ls) u = treap[u].ls;
    return u;
}
```

查询值为x的排名（GetRank）

把小于 x 的分裂出来，答案即为它的大小加 1。

```
//在以u为根的子树中，查找值为x的排名，并且根可能会发生变化
int GetRank(int u,int x) {
    int L, R, res;
    Split(u, x - 1, L, R);
    res = treap[L].siz + 1;
    u = merge(L, R);//分裂完之后再合并，根不变
    return res;
}
```

查询排名为 k 的结点（GetKth）

判断每次应该走左子树还是右子树即可。

```
//在以u为根的子树中，查找排名为k的点
int GetKth(int u, int k) {
    if (treap[u].siz < k) return 0;//排除不存在的情况
    while (k != treap[treap[u].ls].siz + 1){
        //如果左子树个数比排名还大，说明该点在其左子树
        if (k <= treap[treap[u].ls].siz) u = treap[u].ls;
        else {
            u = treap[u].rs;
            k -= treap[treap[u].ls].siz + 1;//进入右子树，同时修改偏移量
        }
    }
    return u;
}
```

查询值为x的前驱结点（GetPre）

所谓前驱，指的是权值最大的小于 x 的结点。把 $Treap$ 按 $x-1$ 分裂后，答案即是左 $Treap$ 的最大结点。

代码实现

```
// 在以u为根的子树中，查询值为x的前驱
int GetPre(u, int x) {
    int L, R, res;
    Split(u, x - 1, L, R);
    res = GetMax(L);
    u = Merge(L, R);//记得分裂完要合并
    return res;
}
```

查询某个值的后继结点（GetSuc）

所谓后继，指的是权值最小的大于 x 的结点。把 $Treap$ 按 x 分裂后，答案即是右 $Treap$ 的最小结点。

代码实现

```
// 在以u为根的子树中，查询值为x的后继结点
int GetSuc(u, int x) {
    int L, R, res;
    Split(u, x, L, R);
    res = GetMin(R);
    u = merge(L, R);//记得分裂完要合并
    return res;
}
```