注：本文旨在让菜鸟掌握 Splay 的各种操作，大牛勿看、勿鄙视

一、二叉搜索树(BST——Binary Search Tree)

二叉搜索树是一颗满足如下性质的二叉树：左子树值<=根节点值<=右子树值。因此理论上我们可以在 O(logN)的时间内完成插入、删除、查找等操作。是一种在“动态维护”中效果相当不错的数据结构。但由于题目数据的原因，可能造成二叉查找树并不平衡（严重的时候可能退化为线性），致使插入、删除、查找等操作的复杂度退化为 O(N)。为了尽量保持二叉搜索树的平衡，我们需要去维护二叉搜索树——平衡二叉树。

二、平衡二叉树(BBST——Balance Binary Search Tree)

I、首先介绍所有平衡二叉树都需要进行的操作——旋转(Rotate)

下面我们以右旋（ZIG）为例来分析旋转操作，我们想把 X 通过右旋旋转到目前 Y 的位置。我们知道 Y 的左子树中所有节点权值都小于等于 Y，于是我们完全可以让 X 的右子树去充当 Y 的左子树，由于 Y 节点权值大于 X，我们又可让 Y 来充当 X 的右子树，通过这样的旋转操作，X 便到了目前 Y 的位置。

II、一般常用的平衡二叉树有如下两类：

1、 Treap：即 Tree+Heap，为每一个节点随机引入一个权值，通过维护这些权值满足堆的性质来尽量保证树的平衡。由于随机权值的引入，能尽量保证树的平衡性，但仍然会存在不稳定的因素。

2、 Splay：即本文所要讲解重点——伸展树。伸展树不能保证每次操作都是 O(logN)的复杂度，但却能保证 m 次操作的复杂度为 O(mlogN)，伸展树的复杂度分析采用了平摊的思想，利用会计方法进行证明。

III、严格保持平衡的树——AVL：

AVL 通过平衡因子的引入，使一颗树左右子树的树高差严格保持不大于 1。因此在所有平衡二叉树中，AVL 的效果最好，但其编程复杂度较大，在平衡编程复杂度与时间效率的情况下，不推荐使用。

三、伸展树(Splay)

在此将通过程序语言的描述更加形象的解读伸展树的各种操作。

1、 程序初始化部分：

我们观察到，二叉搜索树的许多操作具有对称性，因此我们完全可以利用这种对称性将涉及到左右子树的两个操作合并为同一个，这样便可大大降低编程复杂度。

其中关键便是对于某一节点左右儿子的纪录：

Sons:Array[1..MaxP,1..2] Of LongInt;

另外我们仍然需要纪录的便是某一节点的父节点以及以这一节点构成的子树共有多少节点。

Father,Count:Array[1..MaxP] Of LongInt;

2、 旋转操作

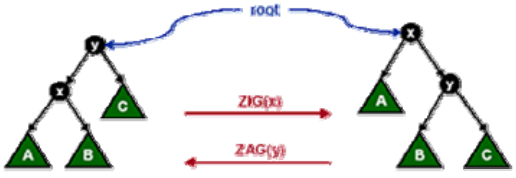
```
Procedure Rotate(X,W:LongInt);  //X表示当前需要旋转节点；W=1表示左旋，W=2表示右旋
Var
  Y:LongInt;
Begin
  Y := Father[X];  //Y为需要旋转节点的父节点

  Count[Y] := Count[Y] - Count[X] + Count[Sons[X,W]];
  Count[X] := Count[X] - Count[Sons[X,W]] + Count[Y];  //完成Count数组的操作

  Sons[Y,3-W] := Sons[X,W];  //如果是右旋，那么需要将父节点的左孩子设置为当前节点的右孩子，反之亦然
  If Sons[X,W]<>0 Then Father[Sons[X,W]] := Y;  //如果是右旋，设置当前节点右孩子的父亲节点，反之亦然

  Father[X] := Father[Y];
  If Father[Y]<>0 Then
    If Y=Sons[Father[Y],1] Then Sons[Father[Y],1] := X Else Sons[Father[Y],2] := X;
      //设置X节点在旋转后与其父亲节点的关联

  Father[Y] := X;  Sons[X,W] := Y;  //设置旋转后Y节点与X节点的关系
End;
```

3、 伸展操作

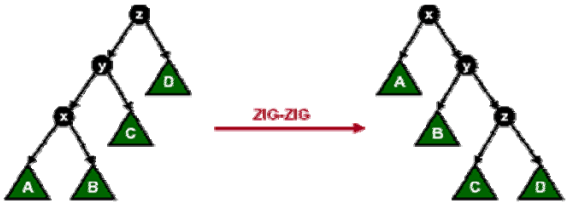
伸展操作 Splay(x,S)是在保持伸展树有序性的前提下，通过一系列旋转操作将伸展树 S 中的元素 x 调整至树的根部的操作。

在旋转的过程中，要分三种情况分别处理：

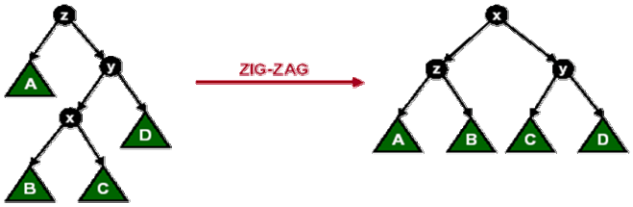
1)Zig 或 Zag 操作：节点 x 的父节点 y 是根节点。



2) Zig-Zig 或 Zag-Zag 操作：节点 x 的父节点 y 不是根节点，且 x 与 y 同时是各自父节点的左孩子或者同时是各自父节点的右孩子。



3)Zig-Zag 或 Zag-Zig 操作：节点 x 的父节点 y 不是根节点，x 与 y 中一个是其父节点的左孩子而另一个是其父节点的右孩子。




```
Procedure Splay(X:LongInt);
Var
  Y:LongInt;
Begin
  While Father[X]<>0 Do Begin
    Y := Father[X];
    If Father[Y]=0 Then
      If X=Sons[Y,1] Then Rotate(X,2) Else Rotate(X,1)
    Else
      If Y=Sons[Father[Y],1]
        Then
          If X=Sons[Y,1]
            Then Begin  Rotate(Y,2);  Rotate(X,2);  End
            Else Begin  Rotate(X,1);  Rotate(X,2);  End
        Else
          If X=Sons[Y,2]
            Then Begin  Rotate(Y,1);  Rotate(X,1);  End
            Else Begin  Rotate(X,2);  Rotate(X,1);  End;
  End;
  Root := X;   //此时X已为树根，改变根的标记
End;
```

4、 查找操作

即为普通 BST 的查找操作，如果待查找值等于当前节点值便返回当前节点编号，小于根节点值便在左子树找，否则便在右子树查找。

```
Function Search(X,W:LongInt):LongInt;  //在以X为根的树中查找
Begin
  While Data[X]<>W Do Begin
    If W<=Data[X] Then Begin
      If Sons[X,1]=0 Then Break;
      X := Sons[X,1];
    End
    Else Begin
      If Sons[X,2]=0 Then Break;
      X := Sons[X,2];
    End
  End;
  Search := X;
End;
```

5、 插入操作

首先通过查找操作确定当前需要插入点的位置，然后判断插入其左子树或者右子

树，最后不要忘记对插入点进行伸展操作。

```
Procedure Insert(W:LongInt);
Var
  I:LongInt;
Begin
  If Total=0 Then Begin
    Inc(Total);
    Father[1] := 0;  Count[1] := 1;  Data[1] := W;  Root := 1;
    Exit;
  End;
  I := Search(Root,W);
  Inc(Total);
  Data[Total] := W;  Father[Total] := I;  Count[Total] := 1;
  Inc(Count[I]);
  If Data[I]>=W Then Sons[I,1] := Total Else Sons[I,2] := Total;
  Splay(Total);
End;
```

6、 极值操作——W=1,2 分别为求以 X 为根的子树中的最大、最小值

```
Function Extreme(X,W:LongInt):LongInt;
Const
  E:Array[1..2] Of LongInt=(-MaxLongInt,MaxLongInt);
Var
  I:LongInt;
Begin
  I := Search(X,E[W]);
  Extreme := Data[I];  Splay(I);
End;
```

7、 删除操作

Splay 的删除操作不同于一般 BST 的删除操作，它首先将待删除点旋转到根节点，然后合并他的左右子树（即找到左子树的最大值当新树的根，将右子树现有的根与其相连）

```
Procedure Delete(W:LongInt);
Var
  I:LongInt;
Begin
  I := Search(Root,W);
  If Data[I]<>W Then Splay(I)
  Else Begin
    Splay(I);
    If Sons[I,1]=0 Then Root := Sons[I,2]
    Else Begin
      Father[Sons[I,1]] := 0;  //断开左子树与根的关联
      Root := Extreme(Sons[I,1],2);  //找到左子树中最大值
      Sons[Root,2] := Sons[I,2];  //将右子树与左子树合并
      Count[Root] := Count[Root] + Count[Sons[I,2]];
      Father[Sons[Root,2]] := Root;
    End;
  End;
End;
```

8、 前驱、后继操作——即为求第一个毕某一节点值小、大的元素

首先找到该节点，并旋转到跟，前驱即为其左子树的最大值、后继为其右子树最小

值

```
Function Adjacent(X,W:LongInt):LongInt;
Var
  I:LongInt;
Begin
  I := Search(Root,X);  Splay(I);
  Adjacent := Extreme(Sons[I,W],3-W);
End;
```

9、 求第 K 大（小）值

以求第 K 小值为例，进行解释：如果 K=左子树所有节点数+1，那么当前根节点即为所求，如果 K<左子树所有节点数，那么就在左子树中查找第 K 小值，否则就在右子树中查找第(K-左子树节点数-1)小值。

```
Function Kth(X,W:LongInt):LongInt;  //W=1为第X小值，W=2为第X大值
Var
  I:LongInt;
Begin
  I := Root;
  While X<>Count[Sons[I,W]]+1 Do Begin
    If X>Count[Sons[I,W]]+1 Then Begin
      X := X - Count[Sons[I,W]] -1;
      I := Sons[I,3-W];
    End
    Else I := Sons[I,W];
  End;
  Kth := I;
  Splay(I);
End;
```

四、总结

伸展树有以下三个优点：

1) 时间复杂度低，伸展树的各种基本操作的平摊复杂度都是 O(log2n)的。

2) 空间要求不高，伸展树不需要记录任何信息以保持树的平衡。

3) 算法简单。编程容易，调试方便。

在信息学竞赛中，我们不能一味的追求算法有很高的时间效率，而应该合理的选择算法，找到时间复杂度、空间复杂度、编程复杂度三者之间的平衡点。