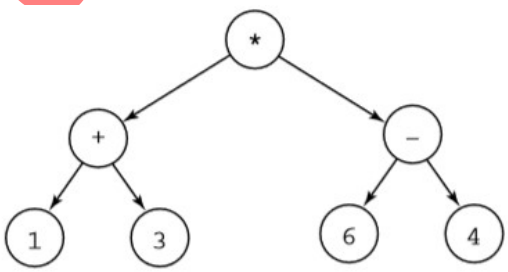
# 用树结构解决问题

二叉树是一种应用很广的非线性数据组织结构，它具有更强的应用扩展能力。除了课堂上所讲述的 BST、Huffman Tree 和 Heap，下面就让我们探索两种不同的二叉树的扩展应用。

## 题目 1：表达式树

二叉树可以用来表达一个算术表达式（当然也可以表达只要是二元运算的任何一种表达式）：叶结点储存操作数，分支结点储存操作符。括号这种运算符在二叉树中完全不需要，因为运算的先后顺序已经由二叉树的层次结构确定好了。更奇妙的是：当你对这个二叉树进行一次遍历就相当于对这个二叉树所表达的表达式完成了计算。

下面的表达式树表达了算术表达式：（1+3）*（6-4）




结合二叉树的遍历操作，对上面的表达式树进行先序遍历、中序遍历和后序遍历分别得到前缀表达式、中缀表达式和后缀表达式：

先序遍历（前缀表达式）：*+13-64

中序遍历（中缀表达式）：((1+3) *（6-4)) [1]

后序遍历（后缀表达式）：13+64-*

一个表达式的计算也可以蕴含在表达式的遍历过程中，下面我们就一步步完成这个功能。

### 任务 1

参照教材上的二叉树的抽象数据类型定义，为表达式树定义合适的数据成员和基础的成员方法。

### 任务 2

为任务 1 中创建的二叉树数据类型定义方法：readPostBinaryTree(String expression)。这个方法基于 expression 创建一棵二叉树，expression 是一个后缀算术表达式。具体的构建过程参看**附录 1**。

[1] 中缀表达式的运算符具有优先级和结合性，为了不丢失这些信息，在中序遍历时为每一个子树都增加了一对圆括弧。

## 任务 3

在任务 2 的基础上实现一个成员方法：postorderResult()，该方法主要完成如下任务：

- 对表达式树进行后序遍历从而生成一个后缀表达式，结点和结点之间用空格分隔；
- 计算该表达式树，给出最终的运算结果；
  - 提示：计算表达式可以利用后序遍历的递归框架实现。
- 输出的格式如下：如果使用题头对应的那棵表达式树的话，应该为：
  1 3 + 6 4 - * = 8

## 任务 4

为表达式树定义一个 output()函数，该函数将表达式树的层次结构以 2D 布局的方式输出。

举例：

后缀表达式为： d e a - + a b + c * *

output 函数的运行结果为：




## 任务 5

为表达式树定义一个 readInBinaryTree(String expression)，这个方法基于 expression 创建一棵二叉树，expression 是一个中缀算术表达式。

提示：可以使用栈转换算数表达式完成中缀转换成后缀表达式，转换的同时构建表达式树。

# 题目 2： Treap

BST 这种数据组织形式，使得查找、删除、插入的运算时间在平均情形下会得到 O(logN)的时间性能。但是在实际的应用中，往往会存在大量有序的数据，这时 BST 会退化，如何避免这种退化造成性能上的衰退？AVL 和红黑树都是比较好的解决方案，这两种二叉树的平衡性是严格的，稳定性表现得十分出色。AVL 和红黑树都是时间效率很高的经典算法，在许多专业的应用领域（如 STL、JavaAPI）有着十分重要的地位。然而 AVL 和红黑树的编程实现难度比较大，编程复杂，故在某些场合下（比如现场类的信息类竞赛中）不太受欢迎。本题的目的让大家掌握一种平衡性也可以得到保证的实现较简单的二叉树：Treap。

Treap 是一种平衡树，这个单词的构造选取了 Tree 的前两个字符和 Heap 的后三个字符。可以想见，Treap 把 BST 和 Heap 结合了起来。它具有 BST 在较好情形下的时间性能，引入堆就是为了帮助可能退化的 BST 维持树的平衡性，让树的高度增长速度控制在 O(logN)中。

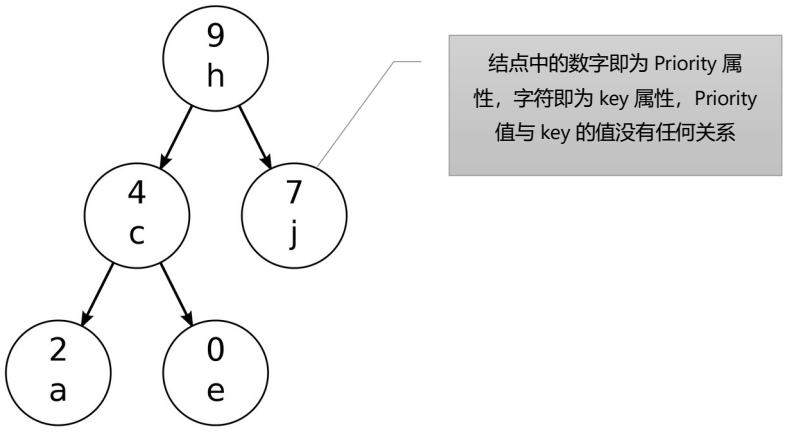
Treap 在 BST 的基础上，为每一个结点添加了一个 priority 属性（这个值是随机生成的）。也就是说 Treap 的结点包含两个值：key 和 priority。结点中 key 值满足 BST 的性质，结点的 priority 属性满足最大值堆性质（或者最小值堆）。Treap 可以定义为满足如下性质的二叉树：

1.任意结点，若其左子树不空，则左子树上所有结点的 key 均小于该结点的 key，而且它的 priority 值大于等于左子树根节点的 priority；

2.任意结点，若其右子树不空，则右子树上所有结点的 key 均不小于该结点的 key，而且它的 priority 值大于等于右子树根节点的 priority；

3.任意结点的左右子树也分别为 Treap。

下图所示的二叉树即为 Treap：




BST 会遇到不平衡的原因是因为有序的数据会使查找的路径退化成单链，而随机序列的数据使 BST 退化是小概率事件。Treap 能够保证二叉树的平衡，根本原因就在于 Priority 值的引入使得树的结构不仅仅取决于结点的 key 值，还取决于 Priority 值。而 Priority 值是随机产生的，由于有序的随机序列是小概率事件，故 Treap 的结构是趋向于随机平衡的。

关于 Treap 的详细介绍，可以参看附件 treap.pdf。

## 任务 1

实现 Treap 数据结构，具有插入、删除、查找等基本操作。

## 任务 2

实现普通的 BST，在此基础上，按照如下表格统计 BST 和 Treap 的相应运行数据，根据这些数据，写出总结。

| | | 有序序列数据（数据规模） | | | 随机序列数据（数据规模） | | |
|---|---|---|---|---|---|---|---|
| | | 1000 | 10000 | 100000 | 1000 | 10000 | 100000 |
| BST | 插入所有元素时间总计 | | | | | | |
| | 树高 | | | | | | |
| | 查找所有元素时间总计 | | | | | | |
| | 删除所有元素时间总计 | | | | | | |
| Treap | 插入所有元素时间总计 | | | | | | |
| | 树高 | | | | | | |
| | 查找所有元素时间总计 | | | | | | |
| | 删除所有元素时间总计 | | | | | | |



# 附录 1

The construction of the tree takes place by reading the postfix expression one symbol at a time. If the symbol is an operand, a one-node tree is created and its pointer is pushed onto a stack. If the symbol is an operator, the pointers to two trees T1 and T2 are popped from the stack and a new tree whose root is the operator and whose left and right children point to *T2* and *T1* respectively is formed . A pointer to this new tree is then pushed to the Stack.

## Example

The input in postfix notation is: a b + c d e + * * Since the first two symbols are operands, one-node trees are created and pointers to them are pushed onto a stack. For convenience the stack will grow from left to right.

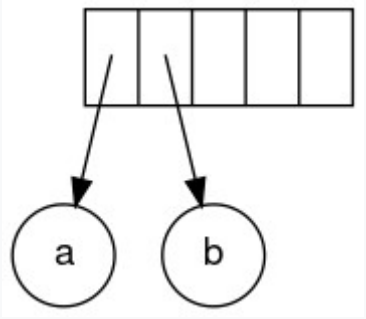



fig1： Stack growing from left to right

The next symbol is a '+'. It pops the two pointers to the trees, a new tree is formed, and a pointer to it is pushed onto the stack.

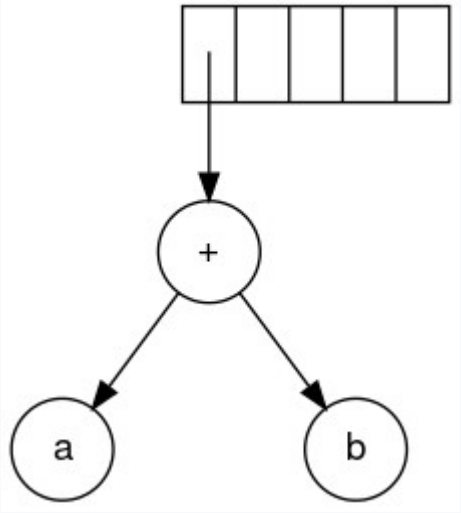



fig2:Formation of a new tree

Next, c, d, and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.

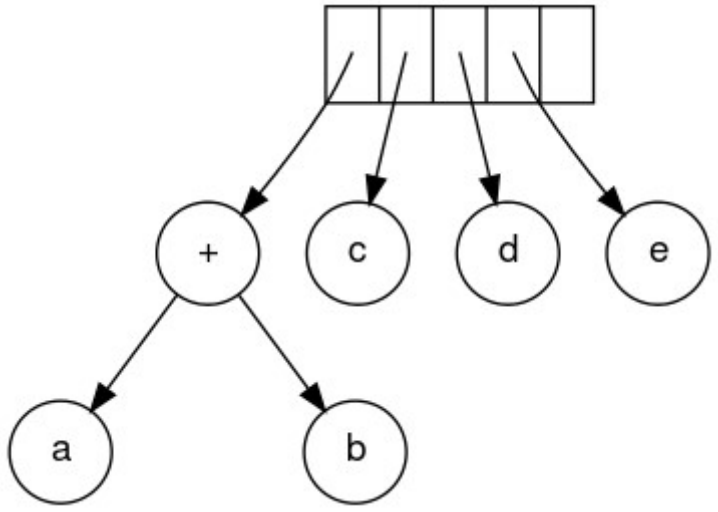



fig3:Creating a one-node tree

Continuing, a '+' is read, and it merges the last two trees.

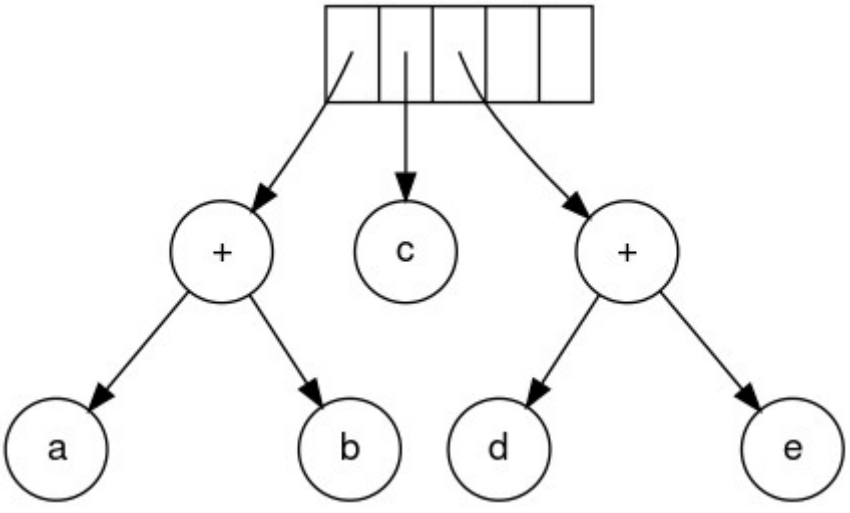



fig4:Merging two trees

Now, a '*' is read. The last two tree pointers are popped and a new tree is formed with a '*' as the root.

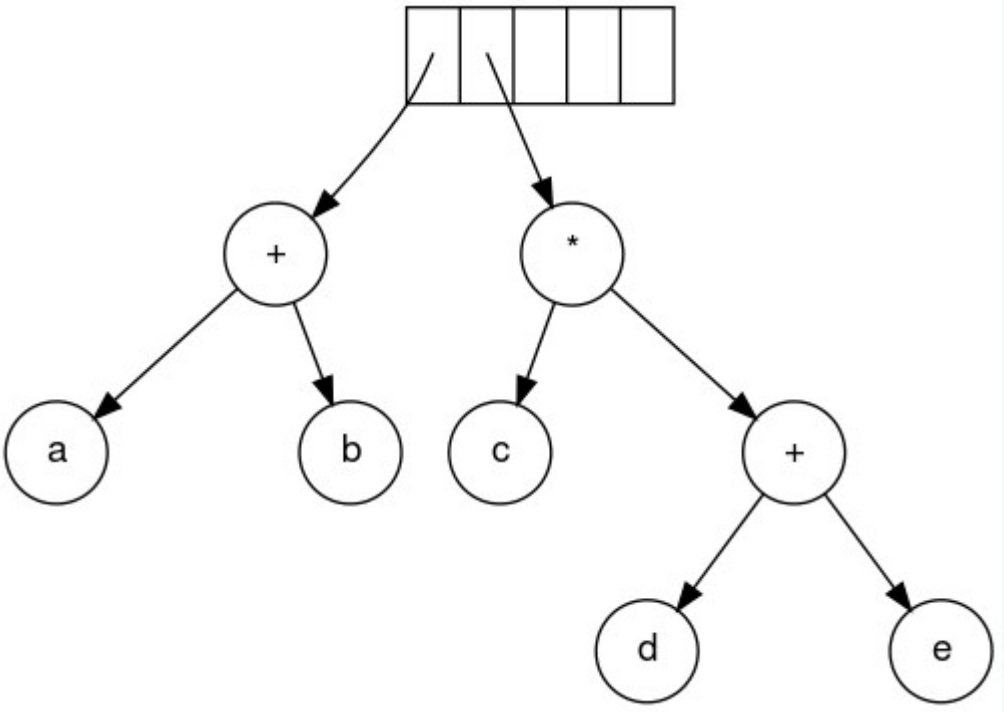



fig4:Forming a new tree with a root

Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack