数据结构与算法

第 9 讲：二叉查找树

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

9.1 问题引入

9.1.1 静态查找表的不足

上一讲学习了静态查找表，两种主要的查找方法各有优缺点：

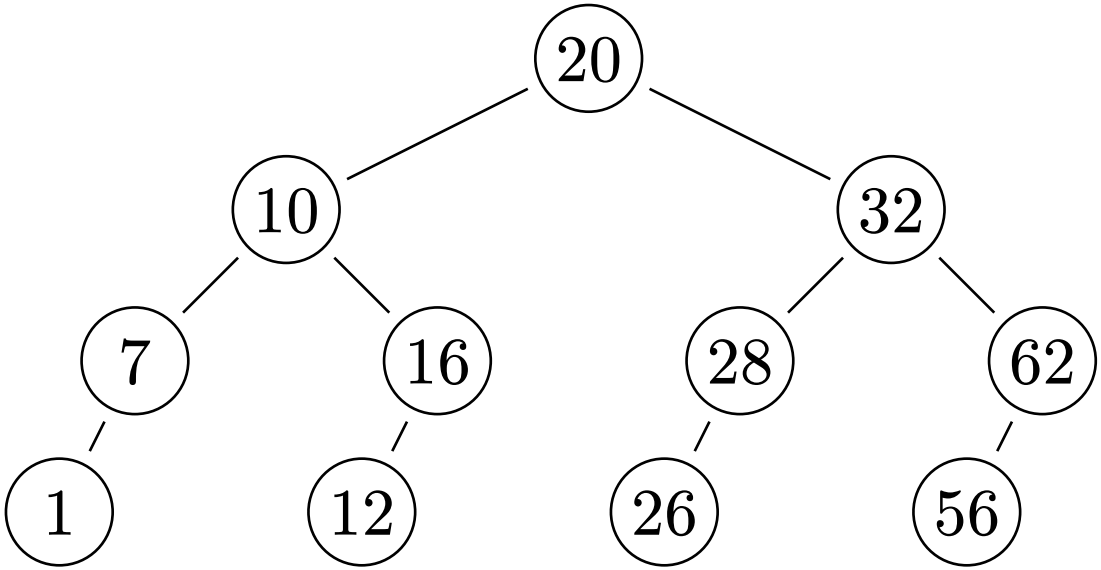
方法	对表的要求	建表复杂度	查找复杂度	修改复杂度
顺序查找	支持顺序访问	$O(1)$	$O(n)$	表尾插入 $O(1)$
二分查找	有序，随机访问	$O(n \log n)$	$O(\log n)$	插入平均 $O(n)$

对于修改操作（插入和删除）与查找操作同等频繁的场景来说，静态查找表不够高效。

使用普通的线性表（包括顺序表和链表）实现的静态查找表在这种情况下受限于查找或修改操作的线性复杂度，需要引入动态查找表。

9.1.2 考虑树状结构

考虑解释二分查找的原理时引入的二叉判定树：



能否用树状结构来设计同时支持高效查找和修改的数据结构?

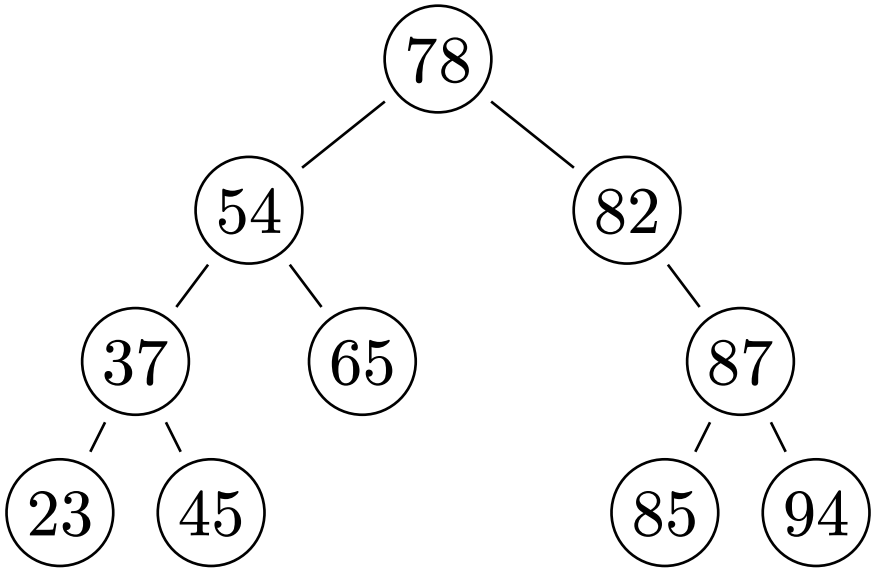
9.2 二叉查找树的概念

9.2.1 二叉查找树的定义

定义 9.1 (二叉查找树) 二叉查找树（binary search tree，BST）要么是一棵空树，要么是具有下列性质的二叉树：

1. 若左子树不为空，则左子树上所有结点的关键字均小于根结点；若右子树不为空，则右子树上所有结点的关键字均大于根结点。
2. 左、右子树本身也是一棵二叉查找树。

9.2.2 二叉查找树示例



中序遍历序列：$23, 37, 45, 54, 65, 78, 82, 85, 87, 94$

一般地，任何一棵二叉查找树的中序遍历序列都是一个有序序列。

9.3 二叉查找树的实现

9.3.1 二叉查找树的存储方式

伪码 9.1: 二叉查找树结点的链式存储

BSTNode:

> value: 结点所带的值，按关键字比较大小
>
> left: 指向左子结点的指针
>
> right: 指向右子结点的指针

设置 root 指针指向二叉查找树的根结点。

9.3.2 查找操作

根据定义 9.1，在二叉查找树上进行查找的过程可以这样实现：

1. 若二叉查找树为空，则查找失败。
2. 若二叉查找树非空，则比较要查找的关键字和根结点的大小关系。

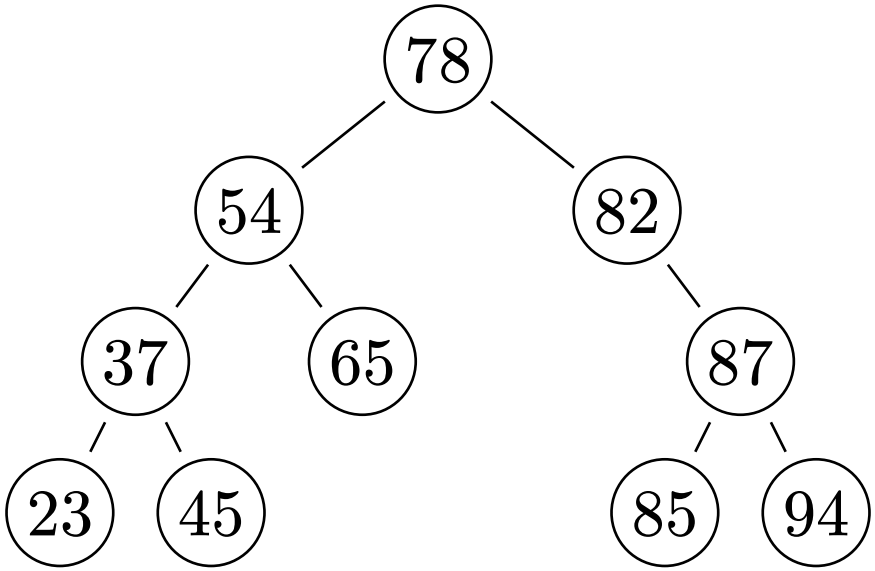
- 若相等，则查找成功。
- 若小于，说明要查找的关键字只可能出现在左子树上，在左子树上继续查找。
- 若大于，说明要查找的关键字只可能出现在右子树上，在右子树上继续查找。

9.3.3 查找算法

伪码 9.2: 二叉查找树的查找

```
Search(p, x):  # p: 当前根结点，x: 查找的关键字
1   if p = ∅  # 树为空
2     return ∅
3   if x = p.value
4     return p
5   elseif x < p.value
6     return Search(p.left, x)
7   else  # x > p.value
8     return Search(p.right, x)
```

9.3.4 查找示例



- 查找 65 需要进行 3 次大小比较，查找成功。
- 查找 86 需要进行 4 次大小比较，查找失败。

二叉查找树的查找算法时间复杂度为 $O(h)$，其中 h 为树的高度。

9.3.5 插入操作

向一棵二叉查找树插入元素可以这样实现：

1. 若二叉查找树为空，则要插入的元素就应该放置在当前位置。
2. 若二叉查找树非空，则比较要插入的关键字和根结点的大小关系。

- 若相等，则查找成功，此时不需要插入。
- 若小于，说明要插入的关键字应该出现在左子树上，在左子树上继续插入。
- 若大于，说明要插入的关键字应该出现在右子树上，在右子树上继续插入。

9.3.6 插入算法

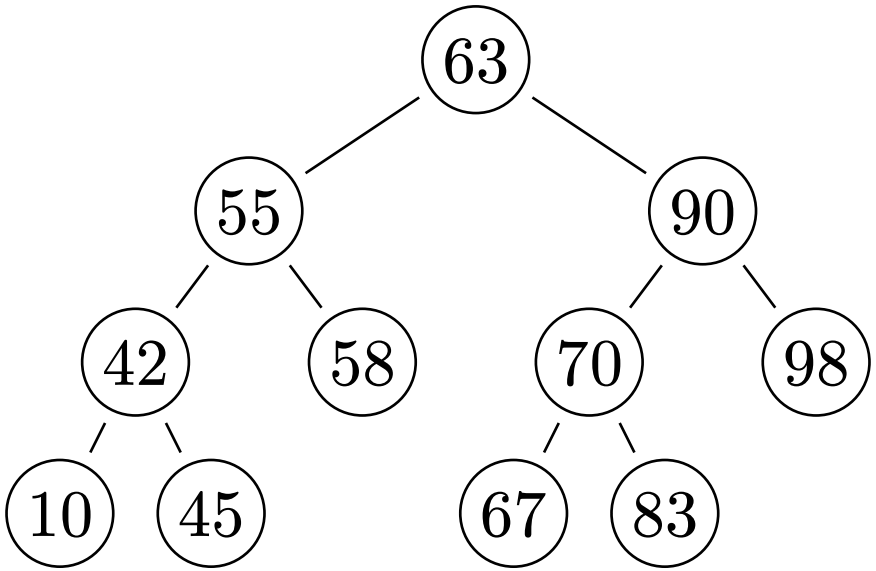
伪码 9.3: 二叉查找树的插入

```
Insert(p, x):  # p: 当前根结点，x: 插入元素
1   if p = ∅  # 树为空
2       p ← BSTNode(x)  # 创建新结点
3       return p
4   if x = p.value  # 找到元素直接返回位置，不需要插入
5       error x 已存在
6   elseif x < p.value
7       return Insert(p.left, x)
8   else  # x > p.value
9       return Insert(p.right, x)
```

9.3.7 插入示例

可以多次使用插入操作构建二叉查找树。

例. 以序列 $63, 90, 70, 55, 67, 42, 98, 83, 10, 45, 58$ 构造二叉查找树。



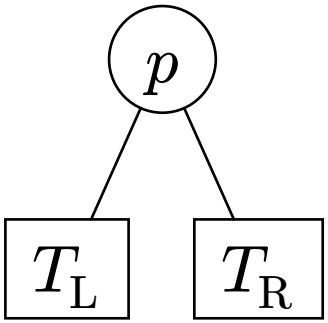
9.3.8 删除操作

要从二叉查找树上删除一个元素，难点在于删除后仍然要保持二叉查找树的性质。

首先在二叉查找树上查找要删除的元素，如果没找到，则删除失败。否则，考虑以下几种情况：

1. 若要删除的是叶结点，直接删除。
2. 若要删除的结点只有左子树，用左子树的根结点替代要删除的结点。
3. 若要删除的结点只有右子树，用右子树的根结点替代要删除的结点。

9.3.9 删除操作的第 4 种情况



最后考虑删除操作的第 4 种情况：要删除的结点既有左子树，也有右子树。此时，用左子树的根结点替代或用右子树的根结点替代都不合适。

思考：左子树（或右子树）上哪个元素放到被删除元素的位置上仍然能保持二叉查找树的性质?

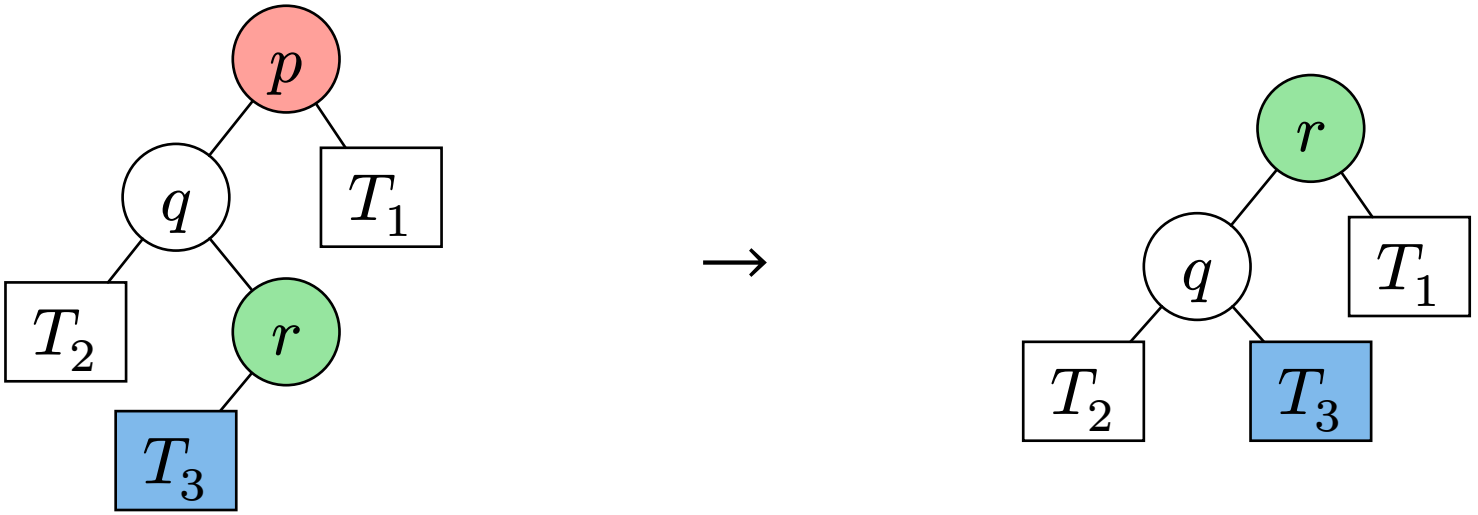
可以使用左子树上最大的元素（或右子树上最小的元素）。

这个元素在哪里？用它替代被删除元素后还需要做什么处理?

9.3.10 使用左子树的最大元素

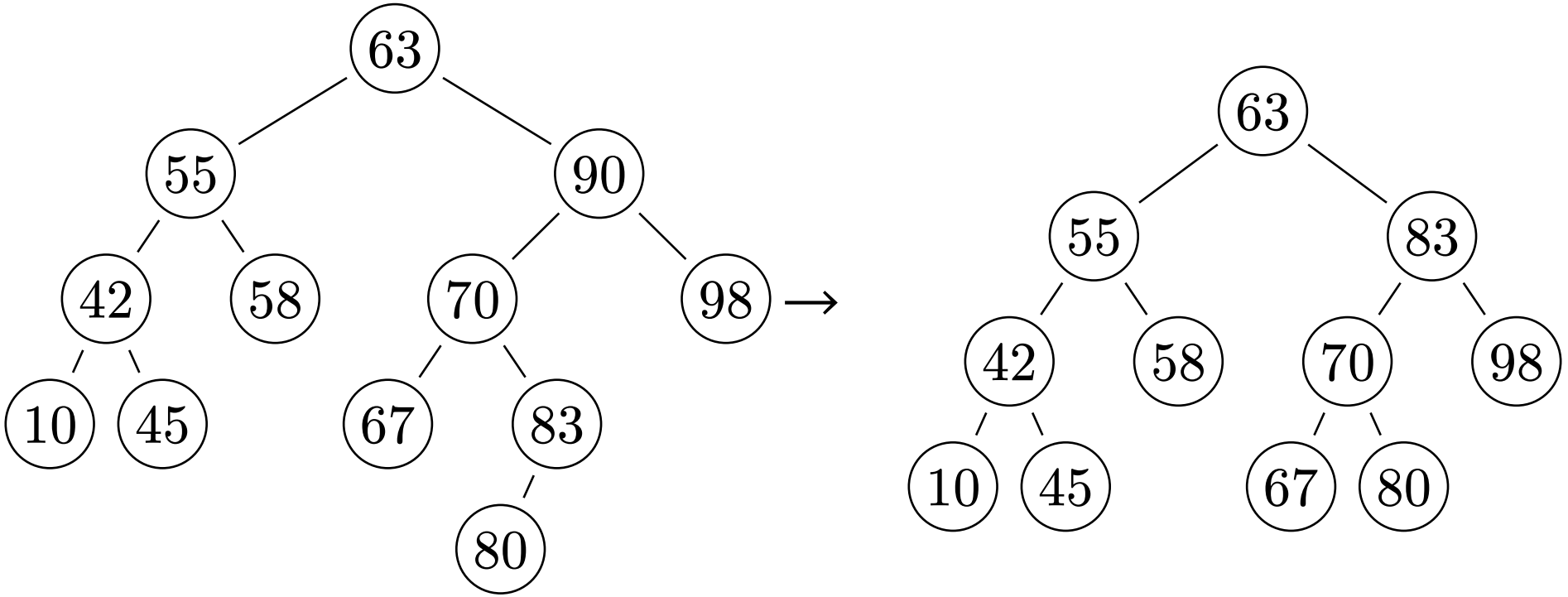
左子树上的最大元素是从左子树的根结点开始，沿着连接右子结点的边一直往右走（不能往左），直到没有右子结点。

把这个元素移动到被删除元素的位置后，删除它原来所在的结点（情况 2）。



9.3.11 删除示例

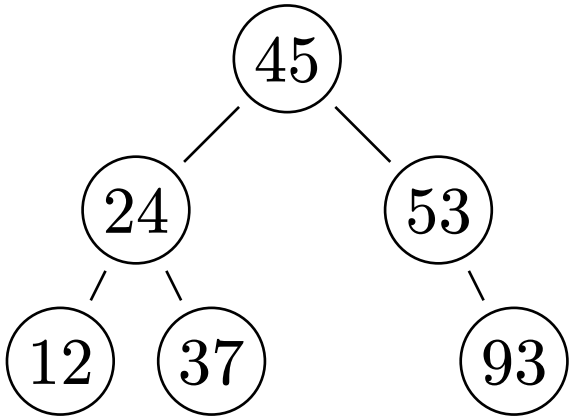
例. 插入示例构建的二叉查找树中再插入 80 后删除元素 90。

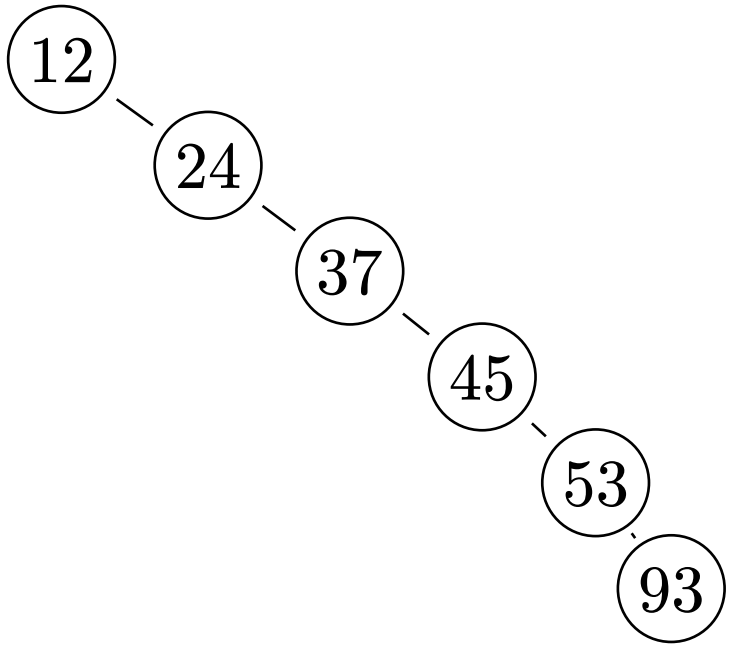


9.3.12 二叉查找树的查找效率

二叉查找树的查找时间复杂度与树的高度线性相关。

同一组元素，以不同顺序插入，构建的二叉查找树形状不同，查找效率也不同，可能会出现退化情况。

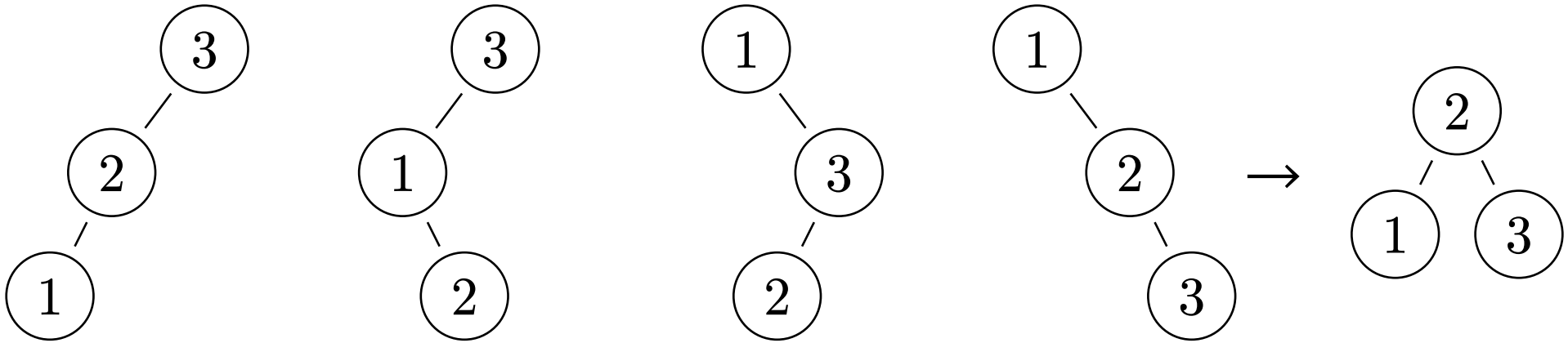




9.3.13 二叉查找树的平衡变换

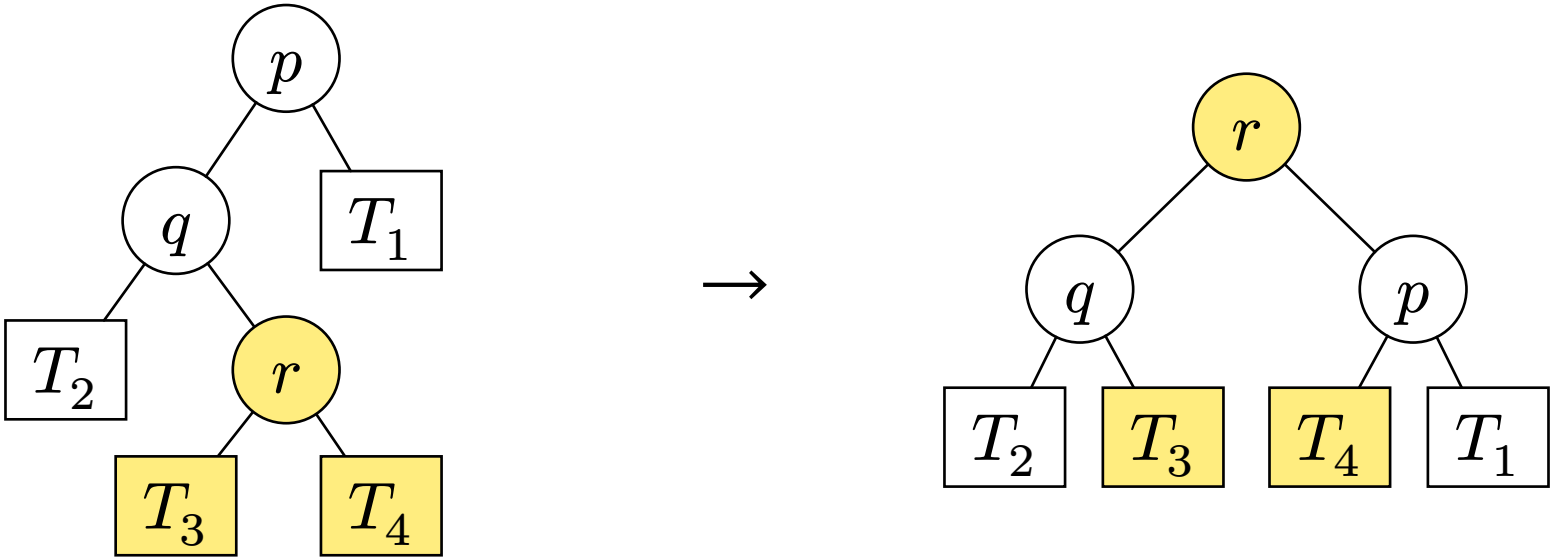
二叉查找树在平均情况下（插入序列随机分布）查找的时间复杂度为 $O(\log n)$。

然而存在一些退化情况，影响实际使用。有没有可能自动处理退化情况?



9.3.14 平衡变换示例

实际中，上述等价变换还要考虑相关结点的子树如何处理。



上述等价变换也称为树的**旋转操作**。

9.3.15 自平衡二叉查找树

能够自动保持平衡的二叉查找树称为**自平衡二叉查找树**（self-balancing binary search tree），简称平衡二叉树或平衡树。

平衡二叉树有很多不同的设计，其中 AVL 树是最早被发明的。它在树的每个结点上定义了**平衡因子**，其值为该结点左右子树高度之差。当所有结点的平衡因子的绝对值不超过 1 时，树是平衡的，否则出现了不平衡的情况，需要用等价变换使其再平衡。AVL 树的查找、插入、删除操作时间复杂度都是 $O(\log n)$。

除 AVL 树外，常见的平衡二叉树还有伸展树、红黑树、AA 树等。

9.3.16 C++ 中的平衡二叉树

C++ 对平衡二叉树进行了封装，提供 `set`、`multiset`、`map` 和 `multimap` 这四个类，称为关联容器（associative container）。要求元素类型支持大小比较操作，也可以给出比较函数。

其中，`set` 和 `multiset` 的元素只有关键字，而 `map` 和 `multimap` 的元素是由关键字和关联的值构成的二元组（pair）。

`set` 和 `map` 中元素的关键字必须各不相等，而 `multiset` 和 `multimap` 支持出现多个关键字相等的元素。

C++ 标准库中这些容器的实现一般采用红黑树。

9.3.17 关联容器的操作

名称	用途	说明
find	查找元素	成功时返回指向的迭代器，失败时返回 end()
count	统计元素	返回元素出现的次数
contains	包含元素	返回元素是否出现
insert	插入元素	插入元素并返回插入后在容器中的位置，对 set 和 map 还会返回插入是否成功
erase	删除元素	删除指定元素，返回删除后下一个元素的位置
[]	关联访问	只有 map 支持，返回给出的关键字所关联的值，如果关键字还没有在容器中，则插入该关键字并将关联的值置为类型的默认值

9.4 平衡二叉树的应用

9.4.1 词频统计

问题. 词频统计是自然语言处理（natural language processing, NLP）中一个非常基本且重要的环节。

给定一段文本，统计出现了多少个不同的单词，按降序列出出现次数最多的 10 个单词（次数相同时按字典序排），给出每个单词出现的次数。

简便起见，单词定义为由空白字符分隔的连续可见字符（字母、数字、标点符号），区分字母的大小写。

9.4.2 词频统计：实现

C++

```cpp
#include <algorithm>
#include <iostream>
#include <map>
#include <string>
#include <vector>

const std::size_t kNumTopWords = 10;

int main() {
  std::map<std::string, std::size_t> count_map;
  std::string word;

  // 依次输入每个单词，统计出现次数
  while (std::cin >> word) {
```

9.4.2 词频统计：实现

```
15      ++count_map[word];
16    }
17
18    // 对统计结果按出现次数排序，出现次数相同按字典序排序
19    std::vector<std::pair<std::string, std::size_t>> count_vec(
20        count_map.begin(), count_map.end());
21    std::sort(count_vec.begin(), count_vec.end(),
22              [](const auto& a, const auto& b) {
23                return a.second > b.second ||
24                       (a.second == b.second && a.first < b.first);
25              });
26
27    // 输出部分频率最高的单词
28    auto end = count_vec.begin() + kNumTopWords;
```

9.4.2 词频统计：实现

```
29    for (auto it = count_vec.begin(); it != end; ++it) {
30      std::cout << it->first << ": " << it->second << '\n';
31    }
32    std::cout << "不同的单词数量: " << count_map.size() << '\n';
33  }
```

9.5 小结

9.5.1 本讲小结

- 二叉查找树的概念
- 二叉查找树的查找、插入、删除操作
- 二叉查找树的效率、自平衡二叉查找树