Tree

beyond

ZUCC ACM Group

2032.01.03

目录

目录

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点
- 一条边直连的两个结点，上层结点被称为下层结点的父亲，下层结点被称为上层结点的儿子

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点
- 一条边直连的两个结点，上层结点被称为下层结点的父亲，下层结点被称为上层结点的儿子
- 除了根没有父亲，其余结点均有且只有一个父亲，但每个结点可以有多个儿子

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点
- 一条边直连的两个结点，上层结点被称为下层结点的父亲，下层结点被称为上层结点的儿子
- 除了根没有父亲，其余结点均有且只有一个父亲，但每个结点可以有多个儿子
- 没有儿子的结点被称为叶子

树的基本概念

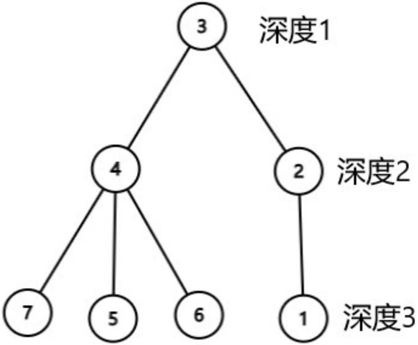
- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点
- 一条边直连的两个结点，上层结点被称为下层结点的父亲，下层结点被称为上层结点的儿子
- 除了根没有父亲，其余结点均有且只有一个父亲，但每个结点可以有多个儿子
- 没有儿子的结点被称为叶子
- 选择树中某一个点 u，断开 (fa[u]-u) 这条边，得到以 u 为根的一棵全新的树被称为 u 的子树

树的基本概念

- 树是一种特殊的图，是 n 个结点 $n-1$ 条边构成的连通图
- 一条边连接两个节点
- 一条边直连的两个结点，上层结点被称为下层结点的父亲，下层结点被称为上层结点的儿子
- 除了根没有父亲，其余结点均有且只有一个父亲，但每个结点可以有多个儿子
- 没有儿子的结点被称为叶子
- 选择树中某一个点 u，断开 (fa[u]-u) 这条边，得到以 u 为根的一棵全新的树被称为 u 的子树
- 一棵子树中所含的结点个数被称为这棵子树的大小

树的基本概念-图示

例如下图，是含有 7 个节点，6 条边的一棵树
结点 3 是根节点，结点 4 和结点 2 是结点 3 的儿子，同时结点 3 就是它们的父亲
结点 2 的子树大小为 2，结点 4 的子树大小为 4，结点 3 的子树大小为 7
结点 1，结点 6，结点 5，结点 7 是叶子结点



- 每个结点 (根结点除外) 有且仅有一个父亲

树的存储

记录每个结点的父亲

- 每个结点 (根结点除外) 有且仅有一个父亲
- 使用一个一维数组 fa 记录即可, fa[u] 表示结点 u 的父亲

  int fa[114514];
- 由于根结点没有父亲, 特殊处理即可, 例如 fa[root] = -1;

树的存储

记录每个结点的儿子

- 每个结点会有多个儿子，那么可以采用二维数组

  int e[114514][114514];

树的存储
记录每个结点的儿子

- 每个结点会有多个儿子，那么可以采用二维数组

  int e[114514][114514];
- 用 e[u][i] 来表示结点 u 的第 i 个儿子

树的存储

记录每个结点的儿子

- 每个结点会有多个儿子，那么可以采用二维数组

```
int e[114514][114514];
```

- 用 e[u][i] 来表示结点 u 的第 i 个儿子
- 显然直接使用二维数组会 MLE，于是可以采用二维数组 + 动态数组的方法

```
vector<int> e[114514];
```

树的存储

记录每个结点的儿子

- 每个结点会有多个儿子，那么可以采用二维数组

  int e[114514][114514];

- 用 e[u][i] 来表示结点 u 的第 i 个儿子
- 显然直接使用二维数组会 MLE，于是可以采用二维数组 + 动态数组的方法

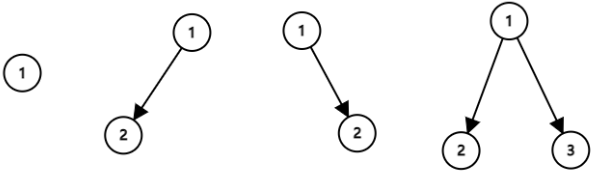
  vector<int> e[114514];

- 这样空间就从 $n \times n$ 被优化到了 $\sum e[i].size() = n - 1$，避免了 MLE

目录

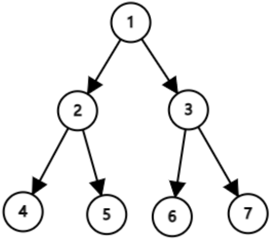
二叉树的概念

- 每个结点最多有两棵子树
- 二叉树的子树有左右之分，其子树的次序不能颠倒

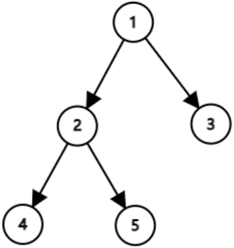


特殊的二叉树

- 满二叉树：一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树，也就是说，如果一个二叉树的层数为 k，且结点总数是 2^k-1，则它就是满二叉树
- 完全二叉树：对于深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时称之为完全二叉树，要注意的是满二叉树是一种特殊的完全二叉树



满二叉树



完全二叉树

二叉树的遍历-前序遍历

- 对于一棵二叉树，从根结点开始，先访问当前结点的值，再访问左儿子 (若有)，之后再访问右儿子 (若有)，这样的访问值的顺序就是前序遍历

```
function DFS(u)
    if u has been visited then
        return
    end if
    visit u
    for v ← minimum to maximum child of u do
        DFS(v)
    end for
    if u is not root then
        DFS(parent of u)
    end if
end function
```

二叉树的遍历-中序遍历

- 对于一棵二叉树，从根结点开始，先访问左儿子 (若有)，再访问当前结点的值，之后再访问右儿子 (若有)，这样的访问值的顺序就是中序遍历

```
function DFS
    if 当前结点存在左儿子 then
        DFS 访问左儿子
    end if
    打印当前结点值
    if 当前结点存在右儿子 then
        DFS 访问右儿子
    end if
end function
```

二叉树的遍历-后序遍历

- 对于一棵二叉树，从根结点开始，先访问左儿子 (若有)，再访问访问右儿子 (若有)，之后再当前结点的值，这样的访问值的顺序就是后序遍历

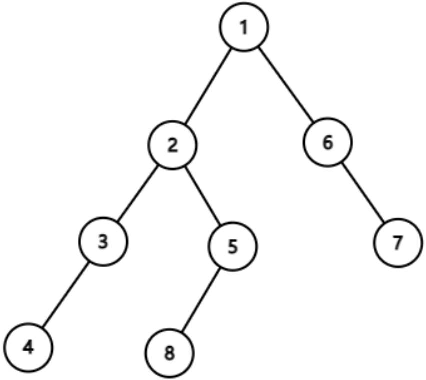
```
function DFS
    if 当前结点存在左儿子 then
        DFS 访问左儿子
    end if
    if 当前结点存在右儿子 then
        DFS 访问右儿子
    end if
    打印当前结点值
end function
```

二叉树的遍历-层序遍历

- 对于一棵二叉树，从根结点开始，一层一层，从上到下，每层从左到右，依次访问每个结点的值，这样的访问值的顺序就是层序遍历

```
function BFS
    定义一个空队列
    将根节点值入队
    while 队列非空 do
        取队头元素 x，并将 x 出队
        打印 x 的值
        if x 存在左儿子 then
            将 x 的左儿子入队
        end if
        if x 存在右儿子 then
            将 x 的右儿子入队
        end if
    end while
end function
```

二叉树的遍历-图示



- 前序遍历: [1 2 3 4 5 8 6 7]
- 中序遍历: [4 3 2 8 5 1 6 7]
- 后序遍历: [4 3 8 5 2 7 6 1]
- 层序遍历: [1 2 6 3 5 7 4 8]

二叉树-例题

给定一棵二叉树的后序遍历和中序遍历，请你输出其层序遍历的序列

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]

二叉树-例题

给定一棵二叉树的后序遍历和中序遍历，请你输出其层序遍历的序列

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]
- 层序遍历: [4 1 6 3 5 7 2]

二叉树-例题

解法:

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]
- 根据后序遍历遍历规则 (左儿子, 右儿子, 当前结点), 可以得出这棵二叉树的根节点为 4

二叉树-例题

解法:

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]
- 根据后序遍历遍历规则 (左儿子, 右儿子, 当前结点), 可以得出这棵二叉树的根节点为 4
- 再根据中序遍历的遍历规则 (左儿子, 当前结点, 右儿子), 结合中序遍历的序列就可以得知 [1, 2, 3] 是结点 4 的左子树上的结点, [5, 6, 7] 是结点 4 的右子树上的结点

二叉树-例题

解法:

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]
- 根据后序遍历遍历规则 (左儿子, 右儿子, 当前结点), 可以得出这棵二叉树的根节点为 4
- 再根据中序遍历的遍历规则 (左儿子, 当前结点, 右儿子), 结合中序遍历的序列就可以得知 [1, 2, 3] 是结点 4 的左子树上的结点, [5, 6, 7] 是结点 4 的右子树上的结点
- 继续结合后序遍历数列, 就可以得出结点 4 的左右子树的后序遍历以及中序遍历的序列:
- 结点 4 左子树: 后序遍历: [2 3 1] 中序遍历: [1 2 3]
- 结点 4 右子树: 后序遍历: [5 7 6] 中序遍历: [5 6 7]

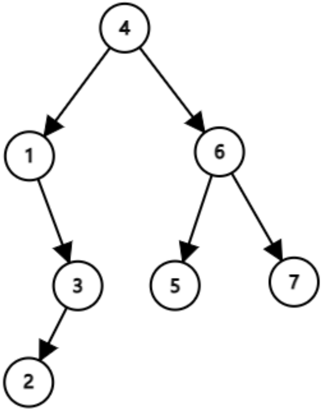
二叉树-例题

解法:

- 后序遍历: [2 3 1 5 7 6 4]
- 中序遍历: [1 2 3 4 5 6 7]
- 根据后序遍历遍历规则 (左儿子, 右儿子, 当前结点), 可以得出这棵二叉树的根节点为 4
- 再根据中序遍历的遍历规则 (左儿子, 当前结点, 右儿子), 结合中序遍历的序列就可以得知 [1, 2, 3] 是结点 4 的左子树上的结点, [5, 6, 7] 是结点 4 的右子树上的结点
- 继续结合后序遍历数列, 就可以得出结点 4 的左右子树的后序遍历以及中序遍历的序列:
- 结点 4 左子树: 后序遍历: [2 3 1] 中序遍历: [1 2 3]
- 结点 4 右子树: 后序遍历: [5 7 6] 中序遍历: [5 6 7]
- 对左右两颗子树进行同样的操作 (递归) 就可以还原出整棵树来

二叉树-例题

还原出的树:



目录

二叉搜索树的定义及性质

- 空树是二叉搜索树

二叉搜索树的定义及性质

- 空树是二叉搜索树
- 若二叉搜索树的左子树不为空，则其左子树上所有点的权值均小于其根节点的值
- 若二叉搜索树的右子树不为空，则其右子树上所有点的权值均大于其根节点的值

二叉搜索树的定义及性质

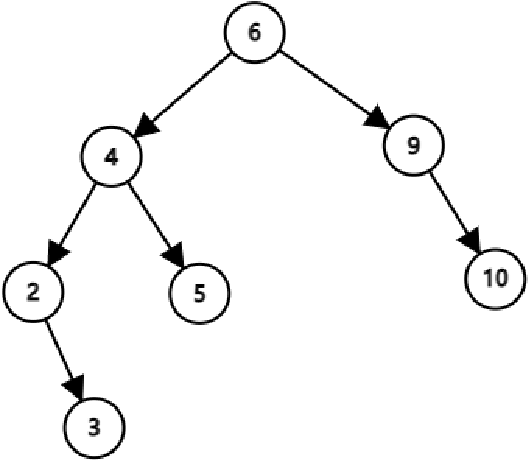
- 空树是二叉搜索树
- 若二叉搜索树的左子树不为空，则其左子树上所有点的权值均小于其根节点的值
- 若二叉搜索树的右子树不为空，则其右子树上所有点的权值均大于其根节点的值
- 二叉搜索树的左右子树均为二叉搜索树

二叉搜索树的定义及性质

- 空树是二叉搜索树
- 若二叉搜索树的左子树不为空，则其左子树上所有点的权值均小于其根节点的值
- 若二叉搜索树的右子树不为空，则其右子树上所有点的权值均大于其根节点的值
- 二叉搜索树的左右子树均为二叉搜索树
- 二叉搜索树的中序遍历结果是一个有序的数列

二叉搜索树-图示

- 下图为一棵二叉搜索树



二叉搜索树-例题

给定一棵二叉搜索树进行前序遍历的结果序列，现请你输出层序遍历的结果

- 前序遍历: [8 6 5 7 10 9 11]

二叉搜索树-例题

给定一棵二叉搜索树进行前序遍历的结果序列，现请你输出层序遍历的结果

- 前序遍历：[8 6 5 7 10 9 11]
- 层序遍历：[8 6 10 5 7 9 11]

二叉搜索树-例题

做法:

- 前序遍历: [8 6 5 7 10 9 11]

二叉搜索树-例题

做法:

- 前序遍历: [8 6 5 7 10 9 11]
- 结点 8 是根节点

二叉搜索树-例题

做法:

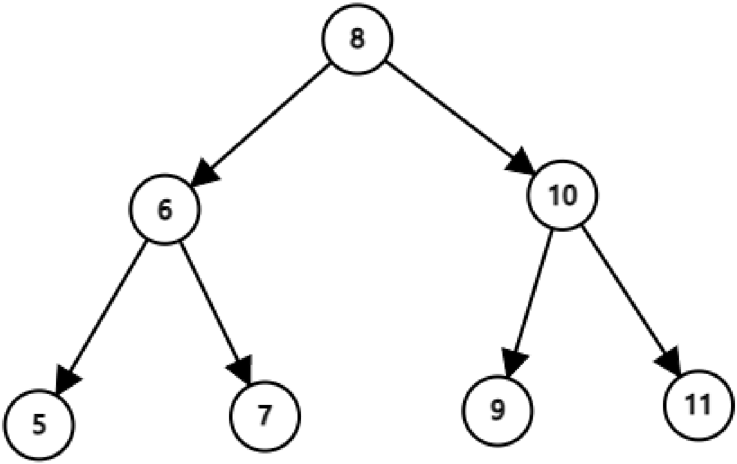
- 前序遍历: [8 6 5 7 10 9 11]
- 结点 8 是根节点
- 结合二叉搜索树的性质, 左子树的前序遍历为 [6 5 7], 右子树的前序遍历为 [10 9 11]

二叉搜索树-例题

做法:

- 前序遍历: [8 6 5 7 10 9 11]
- 结点 8 是根节点
- 结合二叉搜索树的性质, 左子树的前序遍历为 [6 5 7], 右子树的前序遍历为 [10 9 11]
- 对左右子树进行同样的判断即可还原出整棵树

还原出的树:

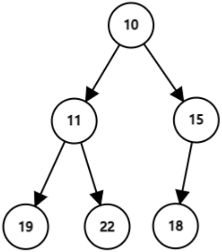


目录

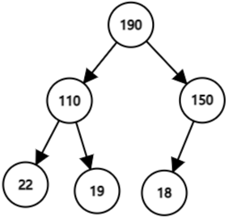
堆的概念

如果有一个关键码的集合 K $= [k_0, k_1, k_2, ..., k_{n-1}]$，把它的所有元素按完全二叉树的顺序存储方式存储在一个一维数组中

- 大根堆：$k_i >= k_{2i+1}$ 且 $k_i >= k_{2i+2}$，即对应完全二叉树中每个父节点的值都大于等于其两个儿子的值
- 小根堆：$k_i <= k_{2i+1}$ 且 $k_i <= k_{2i+2}$，即对应完全二叉树中每个父节点的值都小于等于其两个儿子的值



小根堆



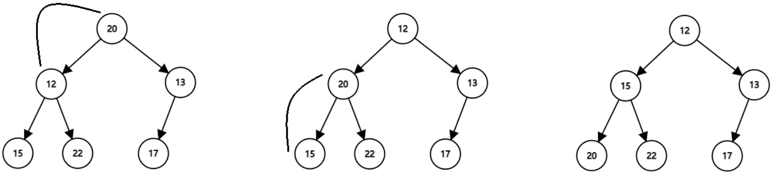
大根堆

堆的性质

- 堆中某个节点的值总是不大于或不小于其父节点的值
- 堆总是一棵完全二叉树
- 可以用堆来实现优先队列

- 先将根结点设置为当前结点 (记作 cur), 将当前结点与其左右孩子的最小值 (记作 a) 进行比较
- 若 cur > a, 不满足小根堆的规则, 将 cur 和 a 交换
- 若 cur <= a, 满足规则, 不进行交换, 调整结束
- 处理完当前结点后, 开始向下处理孩子结点

向下调整法图示 (小根堆):

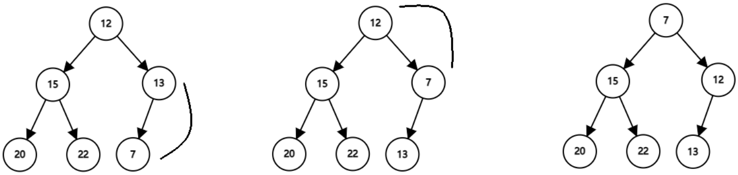


堆的实现

向上调整法-以小根堆为例

- 先将最后一个叶子结点设置为当前结点 (记作 cur), 将其与其父亲结点 (记作 fa) 进行比较
- 若 cur $<=$ fa, 不满足小根堆的规则, 将 cur 和 fa 交换
- 若 cur $>$ fa, 满足规则, 不进行交换, 调整结束
- 处理完当前结点后, 开始向上处理父亲结点

向上调整法图示 (小根堆):



堆的实现

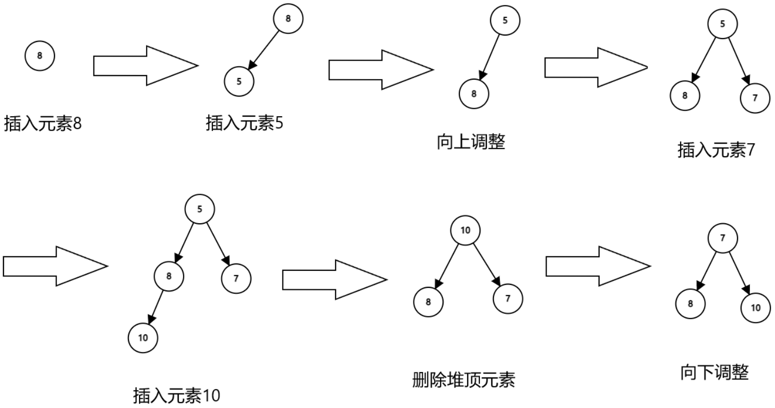
- 插入：将数据插入到数组的末尾，即当前完全二叉树的第一个空结点，然后进行向上调整法

堆的实现

- 插入：将数据插入到数组的末尾，即当前完全二叉树的第一个空结点，然后进行向上调整法
- 删除堆顶元素：将堆顶数据与最后一个数据交换，然后删除最后一个数据，再进行向下调整法

堆的实现

示例 (小根堆):



堆排序

- 将待排序序列依次插入一个空的堆中 (升序采用小根堆, 降序采用大根堆)

堆排序

- 将待排序序列依次插入一个空的堆中 (升序采用小根堆, 降序采用大根堆)
- 然后一个个删除堆顶元素, 直到堆为空, 并记录删除元素的顺序就能得到一个有序序列了

堆排序

- 将待排序序列依次插入一个空的堆中 (升序采用小根堆, 降序采用大根堆)
- 然后一个个删除堆顶元素, 直到堆为空, 并记录删除元素的顺序就能得到一个有序序列了
- 时间复杂度 $O(nlogn)$, 因为 n 个结点的完全二叉树深度为 $logn$, 所以执行一次向上调整或向下调整的时间复杂度为 $O(logn)$, 最多需要执行 n 次, 故总时间复杂度为 $O(nlogn)$

资料

- 二叉树
- 二叉树的遍历
- 二叉搜索树
- 堆

END