

第六课-树、二叉树、二叉搜索树，堆，图

tree

图 Tree

Linked List Linked List 是特殊化的 Tree

Tree 是特殊化的 Graph Graph

示例代码

```
Python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
class Tree:
    def __init__(self):
        self.root = None
    def insert(self, val):
        if self.root is None:
            self.root = TreeNode(val)
        else:
            self._insert(self.root, val)
    def _insert(self, node, val):
        if val < node.val:
            node.left = self._insert(node.left, val)
        else:
            node.right = self._insert(node.right, val)
    def search(self, val):
        return self._search(self.root, val)
    def _search(self, node, val):
        if node is None:
            return False
        if node.val == val:
            return True
        if val < node.val:
            return self._search(node.left, val)
        else:
            return self._search(node.right, val)
    def delete(self, val):
        return self._delete(self.root, val)
    def _delete(self, node, val):
        if node is None:
            return None
        if node.val == val:
            if node.left is None and node.right is None:
                return None
            elif node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                min_node = self._find_min(node.right)
                node.val = min_node.val
                min_node.left = node.left
                min_node.right = self._delete(min_node.right, val)
        else:
            if val < node.val:
                node.left = self._delete(node.left, val)
            else:
                node.right = self._delete(node.right, val)
        return node
    def _find_min(self, node):
        while node.left is not None:
            node = node.left
        return node
```

图 Graph



二叉树

二叉树 Binary tree

二叉树遍历

1. 前序 (Pre-order) : 根-左-右
2. 中序 (In-order) : 左-根-右
3. 后序 (Post-order) : 左-右-根

示例代码

```
def preorder(self, root):
    if root:
        self.traverse_path.append(root.val)
        self.preorder(root.left)
        self.preorder(root.right)
def inorder(self, root):
    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)
def postorder(self, root):
    if root:
        self.postorder(root.left)
        self.postorder(root.right)
        self.traverse_path.append(root.val)
```

二叉搜索树

二叉搜索树，也称二叉排序树、有序二叉树 (Ordered Binary Tree Binary Tree)、排序二叉树 (Sorted Binary Tree Binary Tree Binary Tree Binary Tree)，是指一棵空树或者具有下列性质的二叉树：

1. 左子树上所有结点的值均小于它根结点；
2. 右子树上所有结点的值均大于它根结点；
3. 以此类推：左、右子树也分别为二叉查找。（这就是重复性！）

中序遍历：升排列

二叉搜索树常见操作

1. 查询
2. 插入新结点 (创建)
3. 删除 —— 如果删掉以后要选择一个替换 —— 右边 —— 最近的一个节点替换 —— LogN

复杂度分析

操作	Best	Average	Worst
Search	O(1)	O(logN)	O(N)
Insert	O(1)	O(logN)	O(N)
Delete	O(1)	O(logN)	O(N)
Traverse	O(N)	O(N)	O(N)
Balance	O(1)	O(logN)	O(N)
Rotate	O(1)	O(logN)	O(N)
Split	O(1)	O(logN)	O(N)
Join	O(1)	O(logN)	O(N)
FindMin	O(1)	O(logN)	O(N)
FindMax	O(1)	O(logN)	O(N)
FindPredecessor	O(1)	O(logN)	O(N)
FindSuccessor	O(1)	O(logN)	O(N)
FindLCA	O(1)	O(logN)	O(N)
FindKthElement	O(1)	O(logN)	O(N)
FindKthSmallest	O(1)	O(logN)	O(N)
FindKthLargest	O(1)	O(logN)	O(N)
FindKthSmallestGreater	O(1)	O(logN)	O(N)
FindKthLargestSmaller	O(1)	O(logN)	O(N)
FindKthSmallestGreaterEqual	O(1)	O(logN)	O(N)
FindKthLargestSmallerEqual	O(1)	O(logN)	O(N)

堆

堆 Heap

Heap: 可以总结为一堆数中的最大或者最小值的堆结构。

堆排序: 堆排序是一种选择排序，它通过不断地将堆顶元素与堆中最后一个元素交换，然后将新的堆顶元素下沉，直到堆顶元素是堆中的最小或者最大元素为止。

堆排序的时间复杂度: O(N log N)

堆排序的空间复杂度: O(1)

堆排序的稳定性: 不稳定

二叉堆

二叉堆性质

通过完全二叉树来实现 (注意: 不是二叉搜索树);

二叉堆 (大顶) 它满足下列性质:

性质一: 是一棵完全树。

性质二: 树中任意节点的值总是 \geq 其子节点的值。

实现细节

1. 二叉堆一般通过 "数组" 来实现
2. 假设 "第一个元素" 在数组中的索引为 0 的话, 则父节点和子节点的位置关系如下:

(01) 索引为 i 的左孩子的索引是 $(2*i+1)$;

(02) 索引为 i 的右孩子的索引是 $(2*i+2)$;

(03) 索引为 i 的父节点的索引是 $\text{floor}((i-1)/2)$;

对于一维数组

一维数组: [110, 100, 90, 40, 80, 20, 60, 10, 30, 50, 70]

二叉堆

0. 根节点 (堆顶元素) 是: a[0]

1. 索引为 i 的左孩子的索引是 $(2*i+1)$;

2. 索引为 i 的右孩子的索引是 $(2*i+2)$;

3. 索引为 i 的父节点的索引是 $\text{floor}((i-1)/2)$;

Insert 插入操作

1. 新元素一律先插入到堆的尾部
2. 依次向上调整整个堆的结构 (一直到根即可)

HeapifyUp

Insert - O(logN)

Insert

Insert

Insert - O(logN)

Delete Max 删除堆顶操作

1. 将堆尾元素替换到顶部 (即对顶破堆删除)
2. 依次从根部向下调整整个堆的结构 (一直到堆尾即可)

HeapifyDown

Delete

用priorityQueue实现的 —— java 里面实现 —— 其他