UNIT 2: SORTING AND TREES

L3. INSERTION SORT, MERGE SORT

WHY SORTING?

- Obvious applications
    - Organize an MP3 library
    - Maintain a telephone directory
- Problems that become easy once items are in sorted order
    - Find a median, or find closest pairs
    - Binary search, identify statistical outliers
- Non-obvious applications
    - Data compression: sorting finds duplicates
    - Computer graphics: rendering scenes front to back

INSERTION SORT

INSERTION-SORT (A, n) $\quad \triangleright A[1 \ldots n]$

- for $j \leftarrow 2$ to n
  - insert key $\mathrm{A}[j]$ into the (already sorted) sub-array $\mathrm{A}[1..j-1]$. by pairwise key-swaps down to its right position
- Complexity
  - $\Theta(n^2)$ comparisons and $\Theta(n^2)$ swaps

BINARY-INSERTION-SORT (A, n) $\quad \triangleright A[1 \ldots n]$

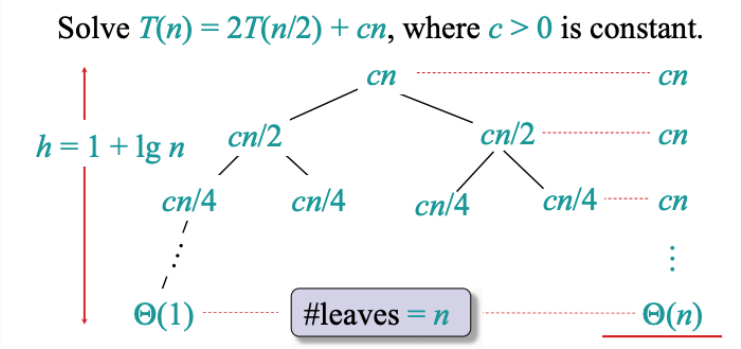
- for $j \leftarrow 2$ to n
  - insert key $\mathrm{A}[j]$ into the (already sorted) sub-array $\mathrm{A}[1..j-1]$. Use binary search to find the right position
- Complexity
  - $\Theta(\mathrm{n} \log \mathrm{n})$ comparisons
  - $\Theta\left(\mathrm{n}^2\right)$ swaps

MERGE SORT (DIVIDE & CONQUER)

MERGE-SoRT $A[1 \ldots n]$ $\quad$ —— $T(n)$

1. If $n = 1$, done (nothing to sort). $\quad$ —— $\Theta(n)$
2. Otherwise, recursively sort $A[1 \ldots n/2]$ and $A[n/2 + 1 \ldots n]$. $\quad$ —— $2T(n/2)$
3. "Merge" the two sorted sub-arrays. $\quad$ —— $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$



- Complexity: $\Theta(n \log n)$

THINKING

这一讲主要讨论了排序的两个算法：插入排序和归并排序。另外通过picture proof推导了归并排序的复杂度

R3. DOCUMENT DISTANCE, INSERTION AND MERGE SORT

THINKING

这节复习课继续讨论了Document Distance问题。由于快速的点积算法需要有序数组，因此引出了插入排序和归并排序的使用。

L4. HEAPS AND HEAP SORT

PRIORITY QUEUE

A data structure implementing a set S of elements, each associated with a key, supporting the following operations:

- insert (S, x) : insert element x into set S
- max (S): return element of S with largest key
- extract_max (S): return element of S with largest key and remove it from S
- increase_key (S, x, k) : increase the value of element x 's key to new value k (assumed to be as large as current value)

HEAP

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key of a node is $\geq$ than the keys of its children (Min Heap defined analogously)

Heap as a Tree

- root of tree: first element in the array, corresponding to $i = 1$
- parent(i) $= \mathrm{i}/2$: returns index of node's parent
- $\mathrm{left(i)} = 2\mathrm{i}:$ returns index of node's left child
- right $(\mathrm{i}) = 2\mathrm{i} + 1$: returns index of node's right child

HEAP OPERATIONS

Max_Heapify

```
l = left(i)
r = right(i)
if (l <= heap-size(A) and A[l] > A[i])
  then largest = l else largest = i
if (r <= heap-size(A) and A[r] > A[largest])
  then largest = r
if largest = i
  then exchange A[i] and A[largest]
    Max_Heapify(A, largest)
```

- Complexity: $\Theta(\log n)$

Build_Max_Heap

```
Build_Max_Heap(A):
  for i=n/2 downto 1
    do Max_Heapify(A, i)
```

- Complexity: $\Theta(n)$

Extract_Max

Exchange first and last items, remove last item, max_heapify

- Complexity: $\Theta(\log n)$ (max_heapify)

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A(1);
3. Swap elements A(n) and A(1):
    - now max element is at the end of the array!
4. Discard node n from heap (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to Step 2 unless heap is empty.

- Complexity: $\Theta(n \log n)$

THINKING

这一讲介绍了优先队列（priority queue），并介绍了其最常用的实现——堆（heap）。

优先队列是一种抽象的数据类型（Abstract data type（ADT））。ADT是由其上可执行的操作以及这些操作的效果与代价所间接定义的。

堆的特点是所有子节点的值都大于或小于该节点的值。而不考虑同级子节点的大小关系。因此构建起来只需要线性复杂度。

进而介绍了基于堆进行的排序，但没有展开讲其特点。

堆主要有三个操作：

- Max_Heapify
- Build_Max_Heap
- Heap_Sort

L5. BINARY SEARCH TREES, BST SORT

SCHEDULING: RUNWAY RESERVATION SYSTEM

- Airport with single (very busy) runway (Boston $6 \rightarrow 1$)
- "Reservations" for future landings
- When plane lands, it is removed from set of pending events
- Reserve req specify "requested landing time" t
- Add t to the set if no other landings are scheduled within k minutes either way. Assume that k can vary.
    - else error, don't schedule

Goal: Run this system efficiently in $O(\lg n)$ time

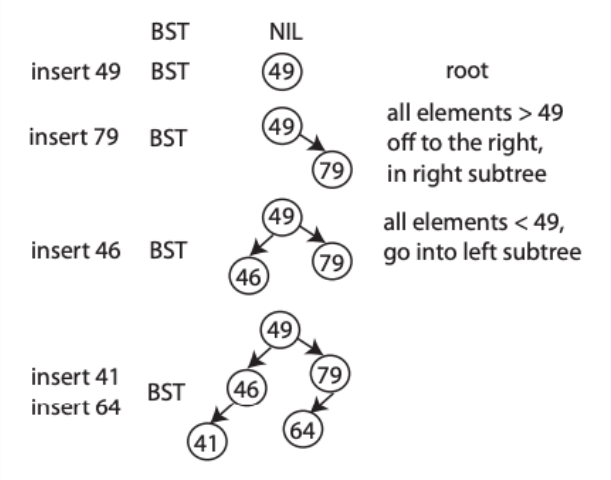
Implementation with Different Data Structures

- Sorted list: Appending and sorting takes $\Theta(n \lg n)$ time. However, it is possible to insert new time/plane rather than append and sort but insertion takes $\Theta(n)$ time. A k minute check can be done in $O(1)$ once the insertion point is found.
- Sorted array: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Using binary search, we find the smallest i such that $R[i] \geq t$
  i.e., the next larger element. We then compare $R[i]$ and $R[i-1]$ against t. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.
- Unsorted list/array: k minute check takes $O(n)$ time.
- Min-Heap: It is possible to insert in $O(\lg n)$ time. However, the k minute check will require $O(n)$ time.

- Dictionary or Python Set: Insertion is $O(1)$ time. k minute check takes $\Omega(n)$ time

Key Lesson: Need fast insertion into sorted list.

BINARY SEARCH TREES (BST)



Properties

Each node x in the binary tree has a key $key(x)$. Nodes other than the root have a parent $p(x)$. Nodes may have a left child left (x) and / or a right child $\text{right}(x)$. These are pointers unlike in a heap.

The invariant is: for any node x, for all nodes y in the left subtree of $x, \text{key}(y) \leq \text{key}(x)$. For all nodes y in the right subtree of $x \, \text{key}(y) \geq \text{key}(x)$.

New Requirement

Rank(t): How many planes are scheduled to land at times ≤ t? The new requirement necessitates a design amendment.

Cannot solve it efficiently with what we have but can augment the BST structure.

the algorithm for augmentation is as follows:

1. Walk down tree to find desired time
2. Add in nodes that are smaller
3. Add in subtree sizes to the left

In total, this takes O(h) time

THINKING

这一讲通过机场跑道预留系统，引出了二叉搜索树。通过对已经讨论过的数据结构的在这个问题上的复杂度的讨论，介绍了不同数据结构在不同问题上的优势。

二叉搜索树的特点在节点的所有左子节点的值都小于该节点的值，所有右子节点的值都大于该节点的值。因此无论是搜索还是插入都只需要树的高度的计算复杂度 $\Theta(h)$。

同时由于保留了同级子节点间的大小关系，对于预留系统的差值比较也只需要对于树的从上至下的复杂度 $\Theta(h)$。

另外介绍了数据结构的增强，从而在现有数据结构上增加更多的低复杂度操作。

R5. RECURSION TREES, BINARY SEARCH TREES

DATA STRUCTURE

A data structure is a collection of algorithms for storing and retrieving information

1. **Queries**: MIN(), MAX(), SEARCH(x)
2. **Updates**: INSERT(x), DELETE(x)

The salient property of a data structure is its **representation invariant (RI)**, which specifies how information is stored.

BINARY SEARCH TREE

A binary search tree is a data structure that allows for key lookup, insertion, and deletion. It is a binary tree, meaning every node x of the tree has at most two child nodes, a left child and a right child. Each node of the tree holds the following information:

- x.key - Value stored in node x.
- x.left- Pointer to the left child of node x. NIL if x has no left child.
- x.right - Pointer to the right child of node x. NIL if x has no right child.
- x.parent - Pointer to the parent node of node x. NIL if x has no parent, i.e. x is the root of the tree.

Binary search tree has the following invariants:

- For each node x, every key found in the left subtree of x is less than or equal to the key found in x.
- For each node x, every key found in the right subtree of x is greater than or equal to the key found in x.

BST OPERATIONS

next larger() and next smaller()

- Case 1: x has a right sub-tree where all keys are larger than x.key. The next larger key will be the minimum key of x's right sub-tree.
- Case 2: x has no right sub-tree. We can find the next larger key by traversing up x's ancestry until we reach a node that's a left child. That node's parent will contain the next larger key.

delete()

- Case 1: x has no children. Just delete it (i.e. change its parent node so that it doesn't point to x).
- Case 2: x has one child. Splice out x by linking x's parent to x's child.
- Case 3: x has two children. Splice out x's successor and replace x with x's successor.

AUGMENTED BSTS

The BST data structure can be easily augmented to implement new features without reinventing the wheel. In the lecutre, we have seen an example of augmenting BST to find the rank of a node. Here, we will look at another example.

For every node x, we can add a field x.min to keep track of the node with the minimum key in the subtree rooted at x. So find min() just returns root.min which is constant time. However, to maintain the invariant of x.min, we need to change insert(x) and delete() as well.

THINKING

这节复习课详细介绍了数据结构的定义、操作分类和最主要的特性——表示不变量（representation invariant（RI））。

然后详细介绍了二叉搜索树的操作：next_larger 和 delete。

一个数据结构支持的操作并非只是可以进行的操作，需要是在合理的运行复杂度内的操作。

L6. AVL TREES, AVL SORT

height of node = length (# edges) of longest downward path to a leaf

The Importance of Being Balanced

balanced BST maintains $h = O(\lg n) \Rightarrow$ all operations run in $O(\lg n)$ time.

AVL TREES

(Adel'son-Vel'skii & Landis 1962)

For every node, require heights of left & right children to differ by at most ±1.

- treat nil tree as height -1
- each node stores its height (DATA STRUCTURE AUGMENTATION) (like subtree size) (alternatively, can just store difference in heights)

Balance

Worst when every node differs by $1-$ let $N_h = (\min.)$ # nodes in height- h AVL tree

$$\begin{aligned} \implies N_h &= N_{h-1} + N_{h-2} + 1 \\ &> 2N_{h-2} \\ \implies N_h &> 2^{h/2} \\ \implies h &< 2 \lg N_h \end{aligned}$$

Rotation

basic operation for restoring AVL property

AVL Insert

1. insert as in simple BST
2. work your way up tree, restoring AVL property (and updating heights as you go).
    - see lecture notes

AVL sort

- insert each item into AVL tree $\Theta(n \lg n)$
- in-order traversal $\frac{\Theta(n)}{\Theta(n \lg n)}$

BIG PICTURE

Abstract Data Type(ADT): interface spec.

vs.

Data Structure (DS): algorithm for each op.

There are many possible DSs for one ADT.

Priority Queue ADT	heap	AVL tree
Q = new-empty-queue( )	$\Theta(1)$	$\Theta(1)$
Q.insert (x)	$\Theta(\lg n)$	$\Theta(\lg n)$
x = Q.deletemin ()	$\Theta(\lg n)$	$\Theta(\lg n)$
x = Q.findmin ()	$\Theta(1)$	$\Theta(\lg n) \to \Theta(1)$

Predecessor/Successor ADT	heap	AVL tree
S = new-empty ()	$\Theta(1)$	$\Theta(1)$
S.insert (x)	$\Theta(\lg n)$	$\Theta(\lg n)$
S.delete (x)	$\Theta(\lg n)$	$\Theta(\lg n)$
y = S.predecessor (x) $\to$ next-smaller	$\Theta(n)$	$\Theta(\lg n)$
y = S.successor (x) $\to$ next-larger	$\Theta(n)$	$\Theta(\lg n)$

THINKING

本讲主要介绍了平衡二叉搜索树（balanced BST）。平衡的重要性主要是确保运行复杂度为对数的而不是线性的。

本讲以AVL树为例，首先定义了其不变量：任意节点的子树高度差不大于1；进而证明了其高度是 $\Theta(\log n)$ 。

之后介绍了在构建AVL树的过程中，插入节点时如何保证其不变量——通过rotation操作。

最后在Lecture5-6的基础上引出了the big picture：抽象数据类型和数据结构。

PS2

THINKING

第二个Problem Set首先使用分形问题讨论了程序的复杂度。分形在总面积基本不变的情况下可以拥有无限长的边长。

然后通过一个实例练习了算法优化：

- 首先使用 `profiler` 找到程序中最为耗时的函数/方法
- 然后搜索/研究/设计更优的数据结构/算法
- 将此函数/方法使用新的数据结构/算法进行优化
    - （优化此函数的同时，注意是否影响其他函数的复杂度）

L7. LINEAR-TIME SORTING: COUNTING SORT, RADIX SORT, LOWER BOUNDS FOR SORTING

COMPARISON MODEL OF COMPUTATION

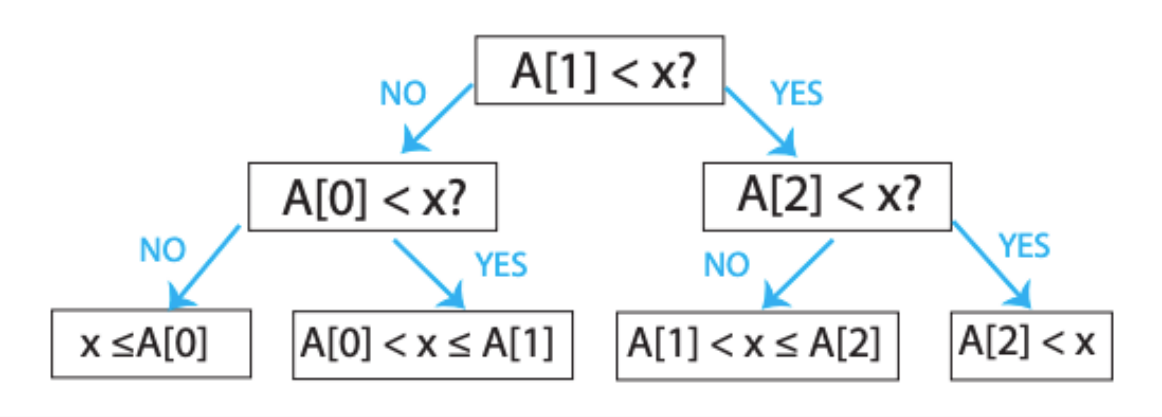
- input items are black boxes (ADTs)
- only support comparisons (<, >, ≤, etc.)
- time cost = # comparisons

And basically the last four lectures have all been about algorithms in this model.

DECISION TREE

Any comparison algorithm can be viewed/specified as a tree of all possible comparison outcomes & resulting output, for a particular n:

- example, binary search for n = 3:



- internal node = binary decision
  - leaf = output (algorithm is done)
  - root-to-leaf path = algorithm execution
  - path length (depth) = running time
  - height of tree = worst-case running time

Search Lower Bound

Claim

- searching among n preprocessed items requires $\Omega(\lg n)$ time

  $\implies$ binary search, AVL tree search optimal

Proof

- $\#$ leaves $\geq \#$ possible answers $\geq n$

  (at least 1 per $A[i]$)
- decision tree is binary
- $\implies$ height $\geq \lg \Theta(n) = \lg n \underbrace{\pm \Theta(1)}_{\lg \Theta(1)}$

Sorting Lower Bound

Claim

- sorting n items requires $\Omega(n \lg n)$

```
$\Longrightarrow$ mergesort, heap sort, AVL sort optimal
```

Proof

- $\#$ leaves $\geq \#$ possible answers $\geq n!$
- decision tree is binary

$$
\begin{aligned}
\implies \text{height} &\geq \lg n! \\
&= \lg(1 \cdot 2 \cdots (n-1) \cdot n) \\
&= \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n \\
&= \sum_{i=1}^{n} \lg i \\
\bullet \quad &\geq \sum_{i=n/2}^{n} \lg i \\
&\geq \sum_{i=n/2}^{n} \underbrace{\lg \frac{n}{2}}_{=\lg n - 1} \\
&= \frac{n}{2} \lg n - \frac{n}{2} = \Omega(n \lg n)
\end{aligned}
$$

- in fact $\lg n! = n \lg n - O(n)$ via Sterling's Formula:

$$
n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \implies \lg n! \sim n \lg n - \underbrace{(\lg e)n + \frac{1}{2}\lg n + \frac{1}{2}\lg(2\pi)}_{O(n)}
$$

LINEAR-TIME SORTING (INTEGER SORTING)

If n keys are integers (fitting in a word) ∈ 0, 1, . . . , k − 1, can do more than compare them

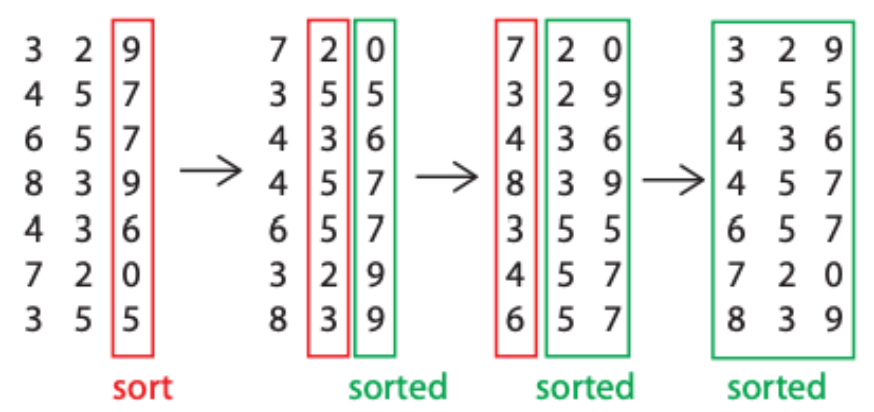
(6.854)

Counting Sort

Radix Sort

- imagine each integer in base $b \implies d = \log_b k$ digits $\in \{0, 1, \ldots, b-1\}$
- sort (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time
- ...
- sort by most significant digit $\rightarrow$ can extract in $O(1)$ time

for example:



- use counting sort for digit sort
- $\implies \Theta(n + b)$ per digit
- $\implies \Theta((n + b)d) = \Theta\left((n + b)\log_b k\right)$ total time
- minimized when $b = n$
- $\implies \Theta\left(n \log_n k\right)$
- $= O(nc)$ if $k \leq n^c$

THINKING

本讲首先介绍了一个新的抽象计算模型（比较模型）来讨论某操作（如查找和排序）复杂度的下界（lower bounds）。

可以看到在计算理论中，不同的计算模型可以用来处理不同的理论问题。

本讲还介绍了线形复杂度的整数排序算法。

R7. COMPARISON SORT, COUNTING AND RADIX SORT

SORTING METHOD

Comparison Model: $\Omega(n \log n)$

Insertion Sort: $O(n^2)$ / stable

Merge Sort: $O(n \log n)$ / not in-place / could be stable

Heap Sort: $O(n \log n)$ / in-place. / not stable

RAM Model: Integer

Counting Sort: $O(n + k)$ / stable

Radix Sort: $O(n)$ / stable

SORT STABILITY

A sorting algorithm is stable if elements with the same key appear in the output array in the same order as they do in the input array

COUNTING SORT

Counting sort is an algorithm that takes an array A of n elements with keys in the range $\{1, 2, \ldots, , k\}$ and sorts the array in $O(n + k)$ time. It is a stable sort.

Implementation in CLRS

see recitation notes

THINKING

这节复习课详细介绍了排序算法的稳定性（stability）：计数排序（counting sort）的稳定性是基数排序（radix sort）的基础。

然后介绍了CLRS中介绍的另一种计数排序算法，计算key前的元素个数，然后遍历原始数据，将其放入对应的位置，并将对应的元素个数加一（遇到下一个相同key的元素时将其放入下一个位置）。

R8. SIMULATION ALGORITHMS

THINKING

这节复习课主要讨论了PSet3中的问题，添加、查找、删除、罗列、计数（两个值之间的），这些操作如何在一个数据结构中完成，它们的复杂度分别为多少。

对于BST中的 `list(tree, l, h)` ，TA详细分析了其复杂度，首先罗列的总数 I 是固定的复杂度，那么需要确定的就是不在罗列范围内需要访问的节点个数，对于LCA的左子树来说：如果访问左节点，意味着右节点及其所有子节点都在罗列范围内；如果访问右节点，意味着做节点及其所有子节点都小于 `l` ，因此不需要访问。因此不在罗列范围内需要访问的节点个数为$O(\log n)$。

PSET3

THINKING

一个问题：首先设计BST，然后再设计AVL Tree，这样似乎更简单一些，而AVL比BST多出的部分就是rebalance。这样也符合6.009的思想——尽量将复杂的问题拆分来处理。对于每次insert和delete，AVL的处理时间最多就是 $O(\log n)$，如果是recursion，只是先找下去，再返回来更新，那么和先查找再更新分为两个函数在渐进复杂度上是一样的。

关于复杂问题的一些思考：

1. 问题从大框架到小细节：

- 抽象：不要纠结于技术的细节，而是关注这个函数需要支持哪些操作。从而从上至下写函数。
- 没有最优唯一解——效率至关重要的项目中可以逐渐优化（学习过程中只需最快速完成目

标即可，60分的效率到80分可能要花费十倍的时间）

- 将问题尽量拆分，例如AVL Tree
- 插入时：1.数值的大小；2.平衡
- 删除时：1.找到对应的node并删除；2.旋转；3.平衡
- list：1.找到lowest common ancester；2.对比
- 等等

2. AVL的一些技术细节：
    - 空树的root是node还是None：应为None
    - 空树是node的问题是其包含值为None的左右子节点，多出了空node类型
        - 空树是node的优势是其可以进行自我递归，所有操作都是对node的操作，但由于需要包含对空node的操作，每步递归中都多了一步操作
    - 使用递归还是迭代：用迭代
    - 用递归需要增加对空树根None的支持，因此需要两个函数（一个主函数及一个只针对node的递归函数），否则会出现前述问题。
        - 如果空树根及无左右子节点时对应的值为None，则使用迭代
        - 如果对应的值为空Node，则形成无限大的空树
    - 是否先有一个BST类：先有
    - 有更多类型的平衡二叉树，都可以调用BST类
        - 可以把复杂问题简化为多步问题
    - BST中是否有parent属性：
    - 没有的话需要在迭代中处理平衡问题，且无法查找次大值及次小值
3. 使用迭代过于复杂后再次尝试递归：
    - 迭代的问题是：你需要返回去查找父节点并更新父节点的指针，此时还需要确定当前节点是父节点的哪个子节点，就变得异常繁琐，优雅度极差。
    - **前述在类中使用递归的问题主要是将类本身作为递归的对象，但实际上可以将函数的参数作为递归的对象，这样可以在函数语句中再区分不同类型 *递归参数* 的对应操作**
    - 否则如果递归self，则递归的对象只能为该类
        - 实际上这种递归方式在看到的第一个AVL实现中就有，如果当时仔细读代码并且至少写一个例子出来，就可以避免很多不必要的时间浪费（但这样绕一圈回来后会对递归的各种细节有更深入的理解）

        - 如果公共方法中不希望出现node作为参数，则使用一个私有方法，被该公共方法调用即可
4. 使用递归过程中的思考：
    - 如果希望在递归过程中进行操作，那么类的继承就变得困难，但是可以通过在递归的两个方向中加入特定函数的调用，在子类中更新该特定函数即可。
    - 例如从BST到AVL时只需将 `_node_update()` 函数增加高度的更新、平衡即可。

这漫长的旅程！PS3确实很困难，AVL Tree有太多的技术细节需要考虑，太多的不同方法要找到自己认为更优的。

但是这个过程对类、继承、递归有了更充分的了解，从概念层面到操作层面。尽量抽象一下：

1. 对象（类）
    1. 数据结构类的类型：
        1. 数据类，只有属性没有方法（可以作为递归方法的参数的多种可能类型之一）
        2. 方法类，只有方法没有属性，作为数据类的方法集合，但支持更多的数据类型输入（如None）
        3. 混合类，既包含属性又包含方法
            1. 如果数据结构是严格递归的，如Trie（没有固定指针），其指针对象恒为Trie，则递归函数的对象为self
            2. 如果数据结构不是严格递归的，如BST（有两个固定指针），其指针对象可能为None，则需要使用本节的技巧将数据与方法分离，分离方法有二：
                1. 一个将数据类和方法类分开，方法的函数不必是数据类，也可以是None或其他，但调用时需要一个外部变量
                2. 使用混合类，但单个方法分开设置公共方法和隐私方法，隐私方法的参数可以为数据类或其他（如None），隐私方法的集合实际上相当于一个方法类
    2. 继承（类方法的继承与升级）
        1. 先调用父类的方法，再进行新的操作，常用于 `__init__` （如需遍历，无法在父类的方法中遍历）
        2. 针对递归：父类方法中有可更新函数，在当前类中更新该函数即可

2. 递归

    1. 递归的优势

        1. 自己调用自己，因此可以使用简洁的代码完成复杂的操作
        2. 双向操作，可以在两个方向上完成不同的工作（双向时没有空间占据和效率问题）
    2. 递归的对象：可以是self，也可以是参数（相对灵活）

        1. 递归对象为self时要求各层结构是严格相同的
    3. 递归的返回

        1. 可以层层return（使用递归效率不及迭代）
        2. 可以每层return都改变数据该层的值——先赋值再return（使用递归效率更高）
        3. 可以对每层return的多个数据进行 `for` 循环（树形递归）