

1. Spatial Locality And Temporal Locality**08A1**

局部性可以体现在**空间**上，也可体现在**时间**上。二者有何联系与区别？

2. Splay-1**08A1**

讲义指出，若将关键码集存储为伸展树，并单调且周而复始地**查找**，则分摊时间成本将高达 $\Omega(n)$ 。
如果改为按某一**随机**次序，周期性地反复查找呢？

3. Splay-1**08A1**

现在考查对伸展树的一个规模为 r 的**子集**，同样**单调且周期性**地反复查找。试证明：

- 在第一轮查找之后，接下来的每一次查找过程中所造访的节点，**都来自上述子集**（与其余节点**无关**）；
- 单次查找的分摊时间成本为 $\mathcal{O}(r)$ 。

4. Splay-2 = Folding**08A2**

Tarjan发明的双层伸展法，之所以可破解讲义中所指出单层法的最坏情况，可以解释为双层伸展能产生一种**折叠**效果。试借助演示工具观察这种效果，并理解背后的原因。

5. Height Difference**08A3**

伸展树的插入、删除操作，都可能引起树高的变化。

- 树高何时会**降低**？何时反而会**增加**？
- 如果是降低，降低的比率**最大**可能达到多少？
- 分别针对插入、删除，**构造**出可以达到上述比率的实例。

6. Amortization**08A4**

本节讲义针对伸展树性能的分摊分析，讨论了三种典型的情况。其中第一种情况（被伸展节点的原深度为奇数时，最终需要单旋一次）幸亏在每次伸展过程中至多出现**一次**（最后一步）。

- 试设想一下，如果这种情况可能出现**多次**，分摊分析的结论是否依然成立？为什么？
- 从势能的角度来看，采用**逐层伸展**策略的BST，为何单次操作的分摊时间复杂度不能控制在 $\mathcal{O}(\log n)$ ？

7. Tightness Of Upper Bounds**08A4**

本节证明：在任一节点 v 的伸展过程中，它每上升两层（从第 $i-1$ 个状态转入第 i 个状态），所需的分摊时间成本总是满足：

$$A_i = T_i + \Delta\Phi_i = \mathcal{O}(\Delta\text{rank}_i(v)) = \mathcal{O}(\text{rank}_i(v) - \text{rank}_{i-1}(v))$$

特别地，当“ $v == p \rightarrow lc \ \&\& \ p == g \rightarrow rc$ ”或“ $v == p \rightarrow rc \ \&\& \ p == g \rightarrow lc$ ”时，必满足：

$$A_i < 2 \times \Delta\text{rank}_i(v)$$

- 该估计式在什么情况下**最紧**？

试画出此时该树在此处的**局部结构**的示意图，并**解释**为何最紧。

- 对任一**偶数** h ，试构造一棵高度为 h 的伸展树，并指定其中的一个节点 v ，使得：

$$\lim_{h \rightarrow +\infty} A(h)/\log(n(h)) = 1/2$$

其中， $A(h)$ 为将 v 伸展至根所需的分摊时间成本， $n(h)$ 为该树的规模。

试画出该树**整体结构**的示意图，标注 v 的位置以及其它必要的信息；同时推导出 $n(h)$ 的表达式。

8. Tightness Of Upper Bounds

08A4

本节证明：在任一节点 v 的伸展过程中，它每上升两层（从第 $i-1$ 个状态转入第 i 个状态），所需的分摊时间成本总是满足：

$$A_i = T_i + \Delta\Phi_i = \mathcal{O}(\Delta\text{rank}_i(v)) = \mathcal{O}(\text{rank}_i(v) - \text{rank}_{i-1}(v))$$

特别地，当 “ $v == p \rightarrow lc \ \&\& \ p == g \rightarrow lc$ ” 或 “ $v == p \rightarrow rc \ \&\& \ p == g \rightarrow rc$ ” 时，必满足：

$$A_i < 3 \times \Delta\text{rank}_i(v)$$

a) 该估计式在什么情况下**最紧**？

试画出此时该树在此处的**局部结构**的示意图，并**解释**为何最紧。

b) 对任一**偶数** h ，试构造一棵高度为 h 的伸展树，并指定其中的一个节点 v ，使得：

$$\lim_{h \rightarrow +\infty} A(h)/\log(n(h)) = 1/3$$

其中， $A(h)$ 为将 v 伸展至根所需的分摊时间成本， $n(h)$ 为该树的规模。

试画出该树**整体结构**的示意图，标注 v 的位置以及其它必要的信息；同时推导出 $n(h)$ 的表达式。

9. Cache

08B1

查阅资料了解Cache的**原理与机制**，了解在典型计算机系统中Cache的**组成**。

10. k-Shift

08B1

讲义中针对k-Shift问题给出了几种解法，并从**发挥**系统缓存作用的角度做了分析比较。试在示例代码中找到对应的代码（或自己重写），针对不同规模的数据做一性能测试，验证或推翻课上给出的结论。

11. setvbuf()

08B2

C语言在<studio.h>库中提供的setvbuf()函数，可以帮助你定制缓冲区的大小，以及缓冲的方式。

试查阅相关资料，并通过编程、运行、统计，对不同缓冲设置的效果做一对比分析。

12. #Keys vs. #External Nodes

08B3

试证明：无论阶次高低，无论关键码多少，任何B-树中的关键码总是恰好比外部节点**少一个**。

13. Size Of Child Vector

08B3

示例代码中定义的BTreeNode里，child向量、key向量的容量都取作 m 。而实际上按照B-树的约定，同一节点中的key**不会超过** $m-1$ ，是否因此可以在定义中将该向量的容量减少一个单位？

14. BTreeNode Construction

08B3

示例代码中的BTreeNode只有一种构造方法：由单个关键码以及左、右孩子，构造一个BTreeNode。事实上在后续的分裂、合并操作中，可能会以一个BTreeNode的局部**片段**，或者两个BTreeNode的**整体**来创建一个BTreeNode。

试针对这些需求，补充对应的构造方法。

15. B-Tree Height

08B4

与BST一样，B-树的搜索性能也决定于其**高度**，讲义中则针对其可能的最小、最大高度做了严格界定。

- a) 试由此反推出, 任一特定高度的B-树中, 最多、最少含有多少个**关键码**?
- b) 随着高度的增大, 这两个数值之间的**范围跨度**将以什么速度增长?
- c) 这样的增长速度意味着什么?

16. B-Tree Traversal**08B4**

为了遍历一棵B-树, 不难理解采用**先序**、**后序**、**层次**策略都是可行的。

- a) 三者策略分别需要使用多少**缓存**空间?
- b) 若限制只能使用 $O(1)$ 缓存空间, 三者策略分别需做多少次I/O?

17. Vector::search In BTree::search**08B4**

示例代码中的B-树在搜索过程中对各BTNode的搜索, 都是直接调用了**向量**统一的查找接口, 而该接口采用的通常都是**二分查找**算法。正如课上指出的, 常规阶次的BTNode更宜采用朴素的**顺序查找**算法。

- a) 试调整此处的代码, 改为采用顺序查找算法;
- b) 通过实测及统计对比, 验证或推翻课上给出的这一建议。

18. Splitting An Even-Order BTNode**08B5**

我们在课上展示的是一棵**奇数** (7) 阶B-树。

- a) 试确认, solveOverflow()算法中的分裂策略, 对**偶数**阶的B-树同样适用;
- b) 试确认, 尽管此时的中位数关键码有两个, 但无论取选用何者都是可行的。

19. Worst Case Of Splits**08B5**

- a) 试举例说明, 在**最坏**情况下, B-树的一次插入操作可能引发 $\Omega(\log n)$ 次分裂;
- b) 在**持续**插入操作的过程中, 发生这种情况的**概率**有多大?
- c) 从**分摊**的角度来看, 每次插入操作会引发多少次分裂?

20. BTree::solveOverflow()**08B5**

上溢与下溢本是**对称**的缺陷, 故就其原理而言, 二者的修复方法及过程也应**互逆**。按照这一理解, 在修复上溢缺陷时除了分裂, 也完全可以优先考虑旋转: 只要不致引发上溢, 便向兄弟节点**出让**一个关键码。

- a) 试改写BTree::solveOverflow()接口, 加入这种策略;
- b) 在实际系统中, 这一对称的策略因何**并未**被普遍采用?

21. B*-Tree**08B5**

B*-Tree是对B-树的改进版本, 它采用了一种**联合分裂**的技巧: 上溢的节点不是独自地分裂, 而是由邻近的 k 个兄弟来共同均摊溢出的关键码, 得到的 $k+1$ 个节点各自至少含有 $\lfloor (m-1) \cdot k / (k+1) \rfloor$ 个关键码, 从而将空间使用率从50%提高至 $k/(k+1)$ 。

试查阅相关资料了解这一改进策略, 并就其效果做一评估。

22. Removing Keys In An Internal BTNode**08B6**

当待删除关键码位于内部节点中时, 我们需要仿照BST的策略, 先将其与直接**后继**或**前驱**交换。

试验证, 此时的直接后继与前驱, 必然都属于某个**叶节点**。

23. No Rotates But Merge**08B6**

- a) `solveUnderflow()`算法可否如`solveOverflow()`那样抛弃旋转策略，只是一味地通过合并修复下溢？
- b) 如我们所见，`solveUnderflow()`算法只有在无法旋转时，才会“不情愿”地做合并。
是否可以**颠倒**这两种策略的优先级——也就是说，只要可行，就总是**优先**通过合并来修复下溢？为什么？

24. Worst Case Of Merges**08B6**

- a) 试举例说明，在最坏情况下，B-树的一次删除操作可能引发 $\Omega(\log n)$ 次合并；
- b) 在**持续**删除操作的过程中，发生这种情况的概率有多大？
- c) 从**分摊**的角度来看，每次删除操作会引发多少次合并？

25. Concurrency**08C1**

除了课上列举的实例，你还见过哪些场合属于**并发**计算？

26. Persistence**08C1**

除了课上列举的实例，你还见过哪些数据结构是**持久**的？

27. Code Reading**08C1**

红黑树已被实现并集成到多种**语言**和**系统**之中，试选择其中的两种，阅读并理解对应的代码。

28. Definition Of Red-Black Tree**08C2**

讲义中定义的红黑树，是对**节点**做染色。实际上，也可以对树中的**边**做染色。

试查阅相关文献，了解其它的定义方式，并确认它们在实质上都是彼此**等效**的。

29. Imaginary External Nodes**08C2**

讲义建议我们在理解和实现红黑树的过程中，**假想地**为红黑树补充足够多的外部节点，将其转化为一棵**真实**二叉树。在后续的双红修复、双黑修复过程中，试体会这种**虚拟实验**式的想象有何妙用。

30. Minimum #Nodes**08C2**

- a) 如果一棵**红黑树**的高度（常规定义的高度而非黑高度）为2，那么它**至少**包含多少个节点？
- b) 高度为3呢？
- c) 一般地，如果将 $C(h)$ 定义为“高度为 h 的红黑树包含节点的最少数目”，试推导出 $C(h)$ ；
- d) 试证明：将 $\{1, 2, 3, \dots, C(h)\}$ 按**单调递增**的次序逐个插入，就能构造出这样的一棵红黑树；
- e) 颠倒过来，按**单调递减**的次序呢？
- f) 还有什么插入次序，可以达到**同样**的效果？

31. Initialized As Black**08C3**

在按照常规BST的算法将一个节点接入红黑树之后，我们建议随即将其**染红**。当然，反过来首先**染黑**必然会违背第四条规则（所有外部节点之**黑深度**统一），但即便如此，是否同样可以简明地予以修复呢？

32. Uncle**08C3**

- a) 试验证，新接入的节点如果造成双红且属于RR-1类型，则当时它的叔叔 u 必是一个（黑的）**外部节点**；
- b) 试验证，新接入的节点如果造成双红且属于RR-2类型，则当时的 u 必是一个（红的）**叶子节点**；
- c) 既如此，为何在修复双红缺陷的算法中，我们需要将 u 视作**一般性**的节点？

33. All Cases Of Double-Red**08C3**

无论RR-1或RR-2, 出现双红缺陷的局部都应各有四种情况, 而讲义则只是各取了其一, 并声称其它情况均由对称性**不难**类似地修复。

- a) 试针对讲义中尚未覆盖的其它情况, 逐一验证修复算法的**确**都可行;
- b) 试对照示例代码, 确认所有情况**均已**完整覆盖。

34. $\Omega(\log n)$ Recolorings**08C3**

在修复双红缺陷的过程中, 我们看到有可能在**每一层**都需要重染色, 累计可能达到 $\Omega(\log n)$ 次。
那么从**分摊**的角度来估计, 需要做多少次呢?

35. Persistence (Again)**08C3**

讲义中指出, 红黑树可以用来实现持久的BBST: 因为每次插入/删除之后只需常数次**旋转**, 红黑树的拓扑**结构**只有常数量级的变化, 故每个新版本的**空间**复杂度得以控制在 $\mathcal{O}(1)$ 。然而我们也注意到, 在一次插入之后的重染色次数**并不能**保证是常数, 极端情况下**甚至**有可能多达 $\Omega(\log n)$ 次。自然地, 新版本为记录这些颜色变化**也应**需要如此之大的空间——这部分的空间消耗, 为何没有考虑呢?

36. Imaginary Child**08C4**

在讲解双黑修复算法时我们提议**假想地**认为, 节点x在按照常规BST的算法被摘除之前, 除了有一棵**子树r**作为此后的替代者, 还有一棵**子树k**——该子树也**假想地**被认为与x同时被摘除。

- a) 试验证, 此时的k无非就是一个**外部节点** (NULL) ——当然, 的确可以视作随同x一并被摘除;
- b) 既如此, 为何在理解整个调整算法时, 我们需要将k视作为一棵**常规**的子树——尽管它从未真实存在过?

37. All Cases Of Double-Black**08C4**

双黑缺陷的四种情况各自都有多种**对称**的子情况, 而讲义为简洁起见, 只是各取了其中**一例**来做讲解。

- a) 试针对讲义中尚未覆盖的其它情况, 逐一验证修复算法的**确**可行;
- b) 试对照示例代码, 确认所有情况**均已**完整覆盖。

38. Decoupling Between BB-3 And BB-2B**08C4**

讲义指出, 尽管BB-3并不能一蹴而就地修复, 但必然可以转化为BB-1或BB-2R。

是的, BB-3不致转化为BB-2B是一种幸运。试设想一下, 如果二者之间不能如此地解耦合, 将会有何后果?