第8章

高级搜索树

[8-1] 试扩充Splay模板类（教材208页代码8.1），使之支持多个相等数据项的并存。
为此，需要增加searchAll(e)和removeAll(e)接口，以查找或删除等于指定目标e的所有节点。
同时，原先的search(e)和remove(e)接口，将转而负责查找或删除等于指定目标e的任一节点。

【解答】

原理及方法，均与习题[7-10]（149页）和习题[7-16]（152页）完全相同。

请读者独立完成编码和调试任务。

[8-2] 试证明，伸展树所有基本操作接口的分摊时间复杂度，均为$\mathcal{O}(\log n)$。

【解答】

关于伸展树可在任意情况下均保持良好的操作效率，教材208页图8.7的实例还不足以作为严格的证明。事实上，伸展树单次操作所需的时间量T起伏极大，并不能始终保证控制在$\mathcal{O}(\log n)$以内。故需沿用教材2.4.4节的方法，从分摊的角度做一分析和评判。具体地，可将实际可能连续发生的一系列操作视作一个整体过程，将总体所需计算时间分摊至其间的每一操作，如此即可得到其单次操作的分摊复杂度A，并依此评判伸展树的整体性能。

当然，就具体的某次操作而言，实际执行时间T与分摊执行时间A往往并不一致，如何弥合二者之间的差异呢？

实际上，分摊分析法在教材中已经而且将会多次出现，比如此前第2.4.4节的可扩充向量、第5.4节的各种迭代式遍历算法以及后面第11.3.7节的KMP串匹配算法等。相对而言，伸展树的性能分析更为复杂，以下将采用势能分析法（potential analysis）。

仿照物理学的思想和概念，这里可假想式地认为，每棵伸展树S都具有一定量（非负）的势能（potential），记作Φ(S)。于是，若经过某一操作并相应地通过旋转完成伸展之后S演化为另一伸展树S'，则对应的势能变化为：

$$\Delta\Phi = \Phi(S') - \Phi(S)$$

推而广之，考查对某伸展树S_0连续实施$m \gg n$次操作的过程。将第i次操作后的伸展树记作S_i，则有：

$$\Delta\Phi_i = \Phi(S_i) - \Phi(S_{i-1}), \quad 1 \le i \le m$$

而从该过程的整体来看，应有

$$\Delta\Phi = \sum_{i=1}^{m}[\Phi(S_i) - \Phi(S_{i-1})] = \Phi(S_m) - \Phi(S_0)$$

也就是说，整体的势能变化量仅取决于最初和最终状态——这与物理学中势能场的规律吻合。势能函数与物理学中势能的另一相似之处在于，它也可以被看作是能量（计算成本）的一种存在形式。比如，当某一步计算实际所需的时间小于分摊复杂度时，则可理解为通过势能的增加

将提前支出的计算成本存储起来；反之，在前者大于后者时，则可从此前积累的势能中支取相应量用于支付超出的计算成本。

以下，若将第i次操作的分摊复杂度取作实际复杂度与势能变化量之和，即

$$A = T_i + \Delta\Phi_i$$

则有

$$\sum_{i=1}^{m} A_i = \sum_{i=1}^{m} T_i + [\Phi(S_m) - \Phi(S_0)]$$

如此，总体的实际运行时间$\sum_{i=1}^{m} T_i$，将不会超过总体的分摊运行时间$\sum_{i=1}^{m} A_i$，故后者可以视作前者的一个上界。

比如，R. E. Tarjan[42]使用如下势能函数：

$$\Phi(S) = \sum_{v\in S} \log|v|, \quad 其中|v| = 节点v的后代数目$$

证明了伸展树单次操作的分摊时间复杂度为$\mathcal{O}(\log n)$。为此，以下将分三种情况（其余情况不过是它们的对称形式）证明：

在对节点v的伸展过程中，每一步调整所需时间均不超过v的势能变化的3倍，即：

$$3\cdot[\Phi'(v) - \Phi(v)]$$

情况A) zig

如教材第8.1.3节所述，这种情况在伸展树的每次操作中至多发生一次，而且只能是伸展调整过程的最后一步。作为单旋，这一步调整实际所需时间为$T = \mathcal{O}(1)$。同时由教材207页图8.5，这步调整过程中只有节点v和p的势能有所变化，且v（p）后代增加（减少）势能必上升（下降），故对应的分摊复杂度为：

$$A = T + \Delta\Phi = 1 + \Delta\Phi(p) + \Delta\Phi(v) \leq 1 + [\Phi'(v) - \Phi(v)]$$

情况B) zig-zag

作为双旋的组合，这一调整实际所需时间为$T = \mathcal{O}(2)$。于是由教材206页图8.4可知：

$$\begin{aligned}
A &= T + \Delta\Phi \\
&= 2 + \Delta\Phi(v) + \Delta\Phi(p) + \Delta\Phi(g) \\
&= 2 + \Phi'(g) - \Phi(g) + \Phi'(p) - \Phi(p) + \Phi'(v) - \Phi(v) \\
&= 2 + \Phi'(g) + \Phi'(p) - \Phi(p) - \Phi(v) \ldots\ldots\ldots\ldots \quad (\because \Phi'(v) = \Phi(g)) \\
&\leq 2 + \Phi'(g) + \Phi'(p) - 2\cdot\Phi(v) \ldots\ldots\ldots\ldots\ldots\ldots \quad (\because \Phi(v) < \Phi(p)) \\
&\leq 2 + 2\cdot\Phi'(v) - 2 - 2\cdot\Phi(v) \ldots \quad (\because \Phi'(g) + \Phi'(p) \leq 2\cdot\Phi'(v) - 2) \\
&= 2\cdot[\Phi'(v) - \Phi(v)]
\end{aligned}$$

这里的最后一步放大，需利用对数函数$f(x) = \log_2 x$的性质，即该函数属于凹函数（concave function），因此必有：

$$\frac{\log_2 a + \log_2 b}{2} \leq \log_2\frac{a + b}{2}$$

亦即：

$$\log_2 a + \log_2 b \leq 2\cdot\log_2\frac{a + b}{2} = 2\cdot[\log_2(a + b) - 1] < 2\cdot(\log_2 c - 1)$$

情况C) zig-zig

作为双旋的组合，这一调整实际所需时间也为$T = \mathcal{O}(2)$。于是由教材206页图8.3可知

$$\begin{aligned}
A &= T + \Delta\Phi \\
&= 2 + \Delta\Phi(v) + \Delta\Phi(p) + \Delta\Phi(g) \\
&= 2 + \Phi'(g) - \Phi(g) + \Phi'(p) - \Phi(p) + \Phi'(v) - \Phi(v) \\
&= 2 + \Phi'(g) + \Phi'(p) - \Phi(p) - \Phi(v) \ldots\ldots\ldots\ldots \quad (\because \Phi'(v) = \Phi(g)) \\
&\leq 2 + \Phi'(g) + \Phi'(p) - 2\cdot\Phi(v) \ldots\ldots\ldots\ldots\ldots\ldots \quad (\because \Phi(v) < \Phi(p)) \\
&\leq 2 + \Phi'(g) + \Phi'(v) - 2\cdot\Phi(v) \ldots\ldots\ldots\ldots\ldots \quad (\because \Phi'(p) < \Phi'(v)) \\
&\leq 3\cdot[\Phi'(v) - \Phi(v)] \ldots\ldots\ldots\ldots \quad (\because \Phi'(g) + \Phi(v) \leq 2\cdot\Phi'(v) - 2)
\end{aligned}$$

同样地，其中最后一步放大也需利用对数函数的凹性。

综合以上各种情况可知，无论具体过程如何，伸展操作的每一步至多需要$3\cdot[\Phi'(v) - \Phi(v)]$时间。因此，若在对伸展树的某次操作中，节点v经过一连串这样的调整上升成为根节点r，则整趟伸展操作总体所需的分摊时间为：

$$\begin{aligned}
A &\leq 1 + 3\cdot[\Phi(r) - \Phi(v)] \leq 1 + 3\cdot\Phi(r) \\
&= \mathcal{O}(1 + \log n) = \mathcal{O}(\log n)
\end{aligned}$$

[8-3] 试扩充 RedBlack 模板类（教材 230 页代码 8.13），使之支持多个相等数据项的并存。为此，需要增加 searchAll(e)和 removeAll(e)接口，以查找或删除等于指定目标 e 的所有节点。同时，原先的 search(e)和 remove(e)接口，将转而负责查找或删除等于指定目标 e 的任一节点。

【解答】

原理及方法，均与习题[7-10]（149页）和习题[7-16]（152页）完全相同。

请读者独立完成编码和调试任务。

[8-4] 试对于任何指定的 m 和 N，构造一棵存有 N 个关键码的 m 阶 B 树，使得在其中插入某个特定关键码之后，需要进行$\Omega(\log_m N)$次分裂。

【解答】

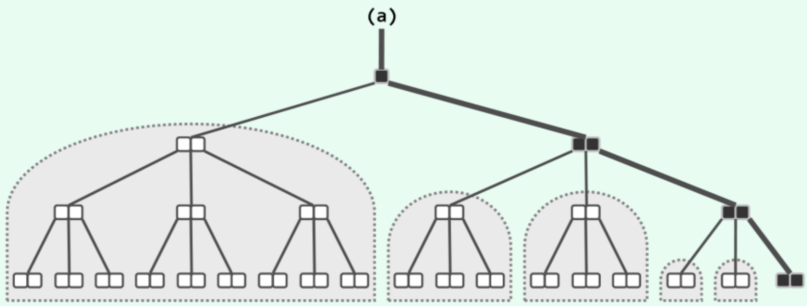
不妨设m为奇数（偶数的情况方法类似，请读者独立补充）。

首先，考查由尽可能少的关键码组成的高度为h的m阶B-树。

例如，如图x8.1所示即是一棵高

度h = 4的m = 5阶B-树，其使用的关键码总数为：

$2\cdot\lceil m/2\rceil^{h-1} - 1 = 53$



图x8.1 高度h = 4、由53个节点组成的一棵5阶B-树

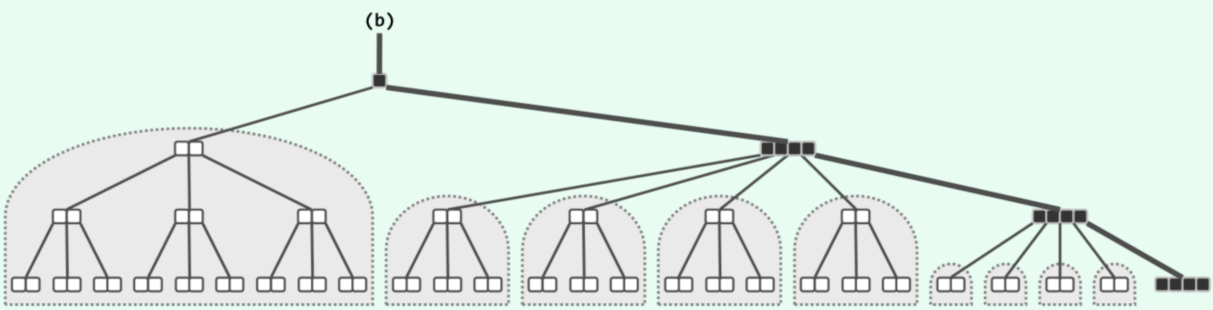
考查该树的最右侧通路。因该通路在图中以粗线条和黑色方格示意，故不妨将沿途的关键码称作黑关键码，其余称作白关键码。于是，如阴影虚框所示，可以将整棵树分割为一系列的子树。

进一步地，如此划分出来的子树，可与最右侧通路上的关键码建立起一一对应的关系：每棵子树的直接后继都是一个黑关键码——亦即不小于该子树的最小关键码。当然特别地，最右侧通路末端节点中的关键码可视作空树的直接后继。

不妨设此树所存的关键码为：

{ 1, 2, ..., n }

以下，若从n + 1起，按递增次序继续插入关键码，则只能沿最右侧通路发生分裂。而且，在根节点保持只有单个关键码的前提下，全树的高度必然保持不变。考查如此所能得到的规模最大的B-树，除根节点外，其最右侧通路上各节点都应含有m - 1个关键码（处于饱和状态）。这样的一个实例，如图x8.2所示。



图x8.2 高度h = 4、由79个节点组成的一棵5阶B-树

若将黑、白关键码所属的节点，亦分别称作黑节点、白节点，则此时它们应分别处于上溢和下溢的临界状态。接下来若再插入一个关键码，而且大于目前已有的所有关键码，则必然会沿着最右侧通路（持续）发生h - 1次分裂。

为统计该树的规模，依然如图中阴影虚框所示，沿着最右侧通路将所有节点分组。进一步地，如此划分出来的子树，同样与最右侧通路上的黑关键码一一对应。

以下，我们将每棵子树与对应的黑关键码归为一组。如此划分之后，考查其中高度为k的任

一子树所属的分组，不难发现其规模应为：

$\lceil m/2 \rceil^{k}$

因此，全树的总规模应为：

$$\hat{N} = \lceil m/2 \rceil^{h-1} + (m - 1) \cdot [\lceil m/2 \rceil^{h-2} + \lceil m/2 \rceil^{h-3} + \ldots + \lceil m/2 \rceil^{0})$$

$$= [\lceil m/2 \rceil^{h-1} \cdot (m + \lceil m/2 \rceil - 2) - m + 1] / (\lceil m/2 \rceil - 1) \ldots\ldots\ldots\ldots (*)$$

反之，便有：

$$h = 1 + \log_{\lceil m/2 \rceil} [((\lceil m/2 \rceil - 1) \cdot \hat{N} + m - 1) / (m + \lceil m/2 \rceil - 2)]$$

$$= \Theta(\log_{\lceil m/2 \rceil} \hat{N}) = \Theta(\log_{m} \hat{N})$$

因此，对于任意指定的规模N，若令：

$$h = 1 + \lfloor \log_{\lceil m/2 \rceil} [((\lceil m/2 \rceil - 1) \cdot N + m - 1) / (m + \lceil m/2 \rceil - 2)] \rfloor$$

并按(*)式估算出$\hat{N} \leq N$，则可按上述方法构造一棵高度为h、规模为$\hat{N}$的m阶B-树，且接下来只要再插入一个全局最大关键码，就会沿最右侧通路发生$h - 1 = \Omega(\log_{m}\hat{N})$次分裂。而其余$N - \hat{N}$个关键码，可在不影响最右侧通路的前提下，作为白关键码适当地插入并散布到各棵子树当中。

[8-5] 现拟将一组共 n 个互异的关键码，插入至一棵初始为空的 m 阶 B-树中，设 m << n。

a) 按照何种次序插入这批关键码，可使所得到的 B-树高度最大？

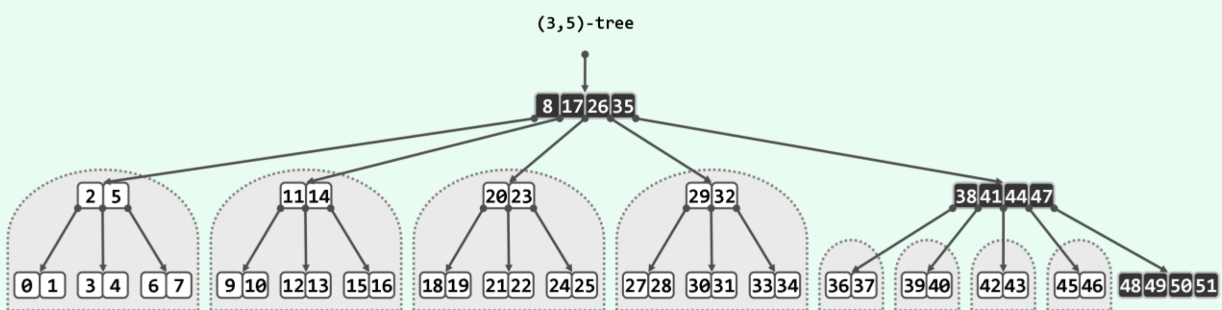
【解答】

保证B-树达到最大高度的一种简明方法，就是按单调次序插入所有关键码。

不妨设m为奇数（偶数的情况方法类似，请读者补充）。比如，按单调递增次序将：

{ 0, 1, 2, ..., 51 }

插入初始为空的5阶B-树，所生成B-树的结构应如图x8.3所示。



图x8.3 按递增插入[0, 52)而生成的5阶B-树

一般地，不难验证：在按递增次序插入各关键码的过程中，最右侧通路（沿途节点在图中以

黑色示意）以下的所有子树（以虚框包围的各组白色节点），始终都属于“稀疏临界”状态。在处于这种状态的子树中，任一节点的删除，都将引起持续的合并操作，并导致高度的下降。

因此，若阶次为m，则此类子树中的每个节点均有$\lceil m/2 \rceil$分支；若其高度为h，则其下所含的外部节点总数应为$\lceil m/2 \rceil^h$，内部节点总数应为$\lceil m/2 \rceil^h - 1$。在上例中m = 5，于是高度为h = 1的（4棵）此类子树必然包含3个外部节点和2个内部节点，高度为h = 2的（4棵）此类子树必然包含9个外部节点和8个内部节点。

实际上若采用单调递增的次序，则每次插入的关键码在当前都属最大。因此，插入算法必然沿着最右侧通路做查找并确定其插入位置；而一旦出现上溢现象，也只能沿最右侧通路实施分裂操作。如此，尽管最右侧通路下属的子树可能会增加，但它们始终保持稀疏临界状态。

一般地，仿照教材8.2.4节的分析方法可知：如此插入[0, n)而生成的m阶B-树，高度应为：

$$h = h_{max} = \log_{\lceil m/2 \rceil} \lfloor (n + 1)/2 \rfloor + 1$$

仍以上述B-树为例，m = 5，n = 52，故树高应为：

$$h = \log_{\lceil 5/2 \rceil} \lfloor (52 + 1)/2 \rfloor + 1 = 3$$

若继续插入下一关键码52，则在持续分裂3次之后，树高将增至：

$$h = \log_{\lceil 5/2 \rceil} \lfloor (53 + 1)/2 \rfloor + 1 = 4$$

依然是此时所能达到的最大树高。

b） 按照何种次序插入这批关键码，可使所得到的B-树高度最小？

【解答】

请读者参照a）中思路，独立给出解答。

[8-6] 考查任意阶的B-树T。

a） 若T的初始高度为1，而在经过连续的若干次插入操作之后，高度增加至h且共有n个内部节点，则在此过程中T总共分裂过多少次？

【解答】

考查因新关键码的插入而引起的任何一次分裂操作。

被分裂的节点，无非两种类型。若它不是根节点，则树中的节点增加一个，同时树高保持不变，故有：

n += 1 和 h += 0

否则若是根节点，则除了原节点一分为二，还会新生出一个（仅含单关键码的）树根，同时树的高度也将相应地增加一层，故有：

n += 2 和 h += 1

可见，无论如何，n与h的差值均会恰好地增加一个单位——因此，n - h可以视作为分裂操作的一个计数器。该计数器的初始值为1 - 1 = 0，故最终的n - h即是从初始状态之最后，整个过程中所做分裂操作的总次数。

请注意，以上结论与各关键码的数值大小以及具体的插入过程均无关，仅取决于B-树最初

和最终的状态——高度和内部节点数。

b） 在如上过程中，每一关键码的插入，平均引发了多少次分裂操作？

【解答】

由上可见，累计发生的分裂操作次数，不仅取决于连续插入操作的次数，同时也取决于最终的树高。前者亦即树中最终所含关键码的总数N，后者即是h。

若关键码总数固定为N，则为使节点尽可能地多，内部节点各自所含的关键码应尽可能地少。注意到根节点至少包含1个关键码，其余内部节点至少包含$\lceil m/2 \rceil - 1$个关键码，故必有：

$$n \le 1 + (N - 1) / (\lceil m/2 \rceil - 1)$$

因此，在如上连续的N次插入操作中，分裂操作的平均次数必然不超过：

$$(n - h) / N < n / N < 1 / (\lceil m/2 \rceil - 1)$$

可见，平均而言，大致每经过$\lceil m/2 \rceil - 1$次插入，才会发生一次分裂。

根据习题[8-4]的结论，某一关键码的插入，在最坏情况下可能引发多达$\Omega(\log_m N)$次的分裂。对照本题的结论可知，这类最坏情况发生的概率实际上极低。

c） 若T的初始高度为h且含有n个内部节点，而在经过连续的若干次删除操作之后高度下降至1，则在此过程中T总共合并过多少次？

【解答】

与a）同理，若合并后的节点不是树根，则有

n -= 1 和 h -= 0

否则若是根节点，则有：

n -= 2 和 h -= 1

可见，无论如何，n与h的差值n - h均会恰好地减少一个单位。既然最终有：

n = h = 1 或等价地 n - h = 0

故其间所发生合并操作的次数，应恰好等于n - h的初值。

同样请注意，以上结论与各关键码的数值大小以及具体的删除过程均无关，仅取决于B-树最初和最终的状态——高度和内部节点数。

d） 设T的初始高度为1，而且在随后经过若干次插入和删除操作——次序任意，且可能彼此相间。试证明：若在此期间总共做过S次分裂和M次合并，且最终共有n个内部节点，高度为h，则必有：

S - M = n - h

【解答】

综合a）和c）的结论可知：在B-树的整个生命期内，n - h始终忠实反映了分裂操作次数与合并操作次数之差。

需要特别说明的是，以上前三问只讨论了连续插入和连续删除的情况，其结论并不适用于本问的情况——两种操作可以任意次序执行。下题将要考查的，即是其中的极端情况。

[8-7] 设 m ≥ 3 为奇数。试对任意的 h > 0，构造一棵高度为 h 的 m 节 B-树，使得若反复地对该树交替地执行插入、删除操作，则每次插入或删除操作都会引发 h 次分裂或合并。

【解答】

若从一棵空的m节B-树开始，按单调顺序依次插入以下关键码：

$\{ 1, 2, 3, 4, 5, ..., N \}$，　其中，$N = 2\cdot[((m + 1)/2)^h - 1]$

则易见，树高恰好为h，而且最右侧通路上的节点均有m个分支，其余节点各有(m + 1)/2个分支。

于是，接下来若继续插入关键码N + 1，则会沿最右侧通路发生h次分裂，全树增高一层；接下来若再删除关键码N + 1，则会沿着最右侧通路发生h次合并，全树降低一层。

更重要的是，如此经过一轮插入和删除，该树宏观的结构以及各节点的组成，都将完全复原。这就意味着，若反复地如此交替地插入和删除，则每一次操作都会在该树中引发h处结构性改变。

当然，此类最坏情况在实际应用中出现的概率同样极低，平均而言，B-树节点分裂与合并的次数依然极少。

[8-8] 对比本章所介绍的 B-树插入与删除算法后不难发现，二者并不完全对称。

比如，删除关键码时若发生下溢，则可能采用旋转（通过父亲间接地向兄弟借得一个关键码）或者合并两种手段进行修复；然而，插入关键码时若发生上溢，却只是统一通过分裂进行修复。

实际上从理论上讲，也可优先通过旋转来修复上溢：

只要某个兄弟仍处于非饱和状态，即可通过父亲，间接地向该兄弟借得一个关键码

a） 仿照代码 8.12（教材 226 页），在代码 8.10（教材 221 页）的基础上做扩充，按上述思路优先通过旋转来修复上溢；

【解答】

这种修复上溢的方法，原理与教材的图8.17（223页）或图8.18（223页）相同，过程恰好相反。请读者根据以上介绍和提示，独立完成编码和调试任务。

b） 在实际应用中，为何不倾向于采用这种手段，而是更多地直接通过分裂来修复上溢？

【解答】

表面上看，B-树的插入操作与删除操作方向相反、过程互逆，但二者并非简单的对称关系。在删除操作的过程中若当前节点发生下溢，未必能够通过合并予以修复——除非其兄弟节点亦处于下溢的临界状态。而在插入操作的过程中若当前节点发生上溢，则无论其兄弟节点的状态和规模如何，总是可以立即对其实施分裂操作。

实际上就算法的控制逻辑而言，优先进行分裂更为简明。而根据习题[8-6]的分析结论，在B-树的生命期内，分裂操作通常都不致过于频繁地发生。因此，不妨直接采用优先进行分裂的策略来修复上溢节点。

另外，优先进行分裂也不致于导致空间利用率的显著下降。实际上无论分裂多少次，无论分裂出多少个节点，根据B-树的定义，其空间利用率最差也不致低于50%。

最后，优先分裂策略也不致于导致树高——决定I/O负担以及访问效率的主要因素——的明显增加。实际上根据教材8.2.4节的分析结论，B-树的高度主要取决于所存关键码的总数，而与其中节点的数目几乎没有关系。

[8-9]　极端情况下，B-树中根以外所有节点只有$\lceil m/2 \rceil$个分支，空间使用率大致仅有 50%。而若按照教材 8.2 节介绍的方法，简单地将上溢节点一分为二，则有较大的概率会出现或接近这种极端情况。

为提高空间利用率，可将内部节点的分支数下限从$\lceil m/2 \rceil$提高至$\lceil 2m/3 \rceil$。于是，一旦节点 v 发生上溢且无法通过旋转完成修复，即可将 v 与其（已经饱和的某一）兄弟合并，再将合并节点等分为三个节点。采用这一策略之后，即得到了 B-树的一个变种，称作 B^*-树（B^*-tree）[39][40]。

当然，实际上不必真地先合二为一，再一分为三。可通过更为快捷的方式，达到同样的效果：从来自原先两个节点及其父节点的共计 m + (m - 1) + 1 = 2m 个关键码中，取出两个上交给父节点，其余 2m - 2 个则尽可能均衡地分摊给三个新节点。

a）按照上述思路，实现 B^*-树的关键码插入算法；

【解答】

如题中所述，若对空间利用率和树的高度十分在意，也不妨采用优先旋转的策略：一旦发生上溢，首先尝试从上溢节点将部分关键码转移至（尚未饱和的）兄弟节点。

请读者参照以上介绍和提示，独立完成编码和调试任务。

b）与 B-树相比，B^*-树的关键码删除算法又有何不同？

【解答】

与插入过程对称地，从节点v中删除关键码后若发生下溢，且其左、右兄弟均无法借出关键码，则先将v与左、右兄弟合并，再将合并节点等分为两个节点。同样地，实际上不必真地先合三为一，再一分为而。可通过更为快捷的方式，达到同样的效果：从来自原先三个节点及其父节点的共计：

$$(\lceil m/2 \rceil - 1) + 1 + (\lceil m/2 \rceil - 2) + 1 + (\lceil m/2 \rceil - 1) = 3\cdot\lceil m/2 \rceil - 2$$

个关键码中，取一个上交给父节点，其余$3\cdot\lceil m/2 \rceil - 3$个则尽可能均衡地分摊给两个新节点。

注意，以上所建议的方法，不再是每次仅转移单个关键码，而是一次性地转移多个——等效于上溢或下溢节点与其兄弟平摊所有的关键码。采用这一策略，可以充分地利用实际应用中普遍存在的高度数据局部性，大大减少读出或写入节点的I/O操作。

不难看出，单关键码的转移尽管也可以修复上溢或下溢的节点，但经如此修复之后的节点将依然处于上溢或下溢的临界状态。接下来一旦继续插入或删除近似甚至重复的关键码（在局部性较强的场合，这种情况往往会反复出现），该节点必将再次发生上溢或下溢。由此可见，就修复效果而言，多关键码的成批转移，相对单关键码的转移更为彻底——尽管还不是一劳永逸。

针对数据局部性的另一改进策略，是使用所谓的页面缓冲池（buffer pool of pages）。这是在内存中设置的一个缓冲区，用以保存近期所使用过节点（页面）的副本。

只要拟访问的节点仍在其中（同样地，在局部性较强的场合，这种情况也往往会反复出现），即可省略I/O操作并直接访问；否则，才照常规方法处理，通过I/O操作从外存取出对应的节点（页面）。缓冲池的规模确定后，一旦需要读入新的节点，只需将其中最不常用的节点删除即可腾出空间。

实际上，不大的页面缓冲池即可极大地提高效率。请读者通过实验统计，独立作出验证。

c） 按照你的构想，实现 B^*-树的关键码删除算法。

【解答】

请读者参照以上介绍和提示，独立完成编码和调试任务。

[8-10] Java 语言所提供的 java.util.TreeMap 类是用红黑树实现的。
试阅读相关的 Java 源代码，并就其实现方式与本章的 C++实现做一比较。

【解答】

请读者对照教材中实现的红黑树，独立完成代码阅读和比较任务。

[8-11] H. Olivie 于 1982 年提出的半平衡二叉搜索树（half-balanced binary search trees）[47]，非常类似于红黑树。这里所谓的半平衡（half-balanced），是指此树的什么性质？
试阅读参考文献，并给出你的理解。

【解答】

按照定义，在半平衡二叉搜索树中，每个节点v都应满足以下条件：v到其最深后代（叶）节点的距离，不得超过到其最浅后代叶节点距离的两倍。

若半平衡二叉搜索树所含内部节点的总数记作n，高度记作h，则可以证明必有：

$$h \le 2\cdot\log_2(n + 2) - 2$$

请读者在阅读相关文献之后，独立给出自己的理解。

[8-12] 人类所拥有的数字化数据的总量，在 2010 年已经达到 ZB（2^70 = 10^21）量级。
假定其中每个字节自成一个关键码，若用一棵 m = 256 阶的 B-树来存放它们，则

a） 该树的最大高度是多少？

【解答】

首先需要指出的是，鉴于目前常规的字节仅含8个比特位，可能的关键码只有2^8 = 256种，故数据集中必然含有大量重复，因此若果真需要使用B-树来存放该数据集，可参照习题[7-10]（149页）和习题[7-16]（152页）的方法和技巧，扩展B-树结构的功能，使之支持重复关键码。

根据教材8.2.4节的分析结论，存放N < 10^21个关键码的m = 256阶B-树，高度不会超过

$$\log_{\lceil m/2\rceil}\lfloor(N + 1)/2\rfloor + 1 = \log_{128}\lfloor(1 + 10^{21})/2\rfloor + 1$$
$$\sim \log_2 10^{21} / \log_2 128 + 1 \sim 70 / 7 + 1 = 11$$

b） 最小呢？

【解答】

同样根据教材8.2.4节的分析结论，该B-树的高度不会低于

$$\log_m(N + 1) = \log_{256}(10^{21} + 1) \sim \log_2 10^{21} / \log_2 256 \sim \lceil 70 / 8\rceil = 9$$

实际应用中，多采用128~256阶的B-树。综合以上分析结论，可以明确地看到，此类B-树的高度并不大，而且起伏变化的范围也不大。这也是在多层次存储系统中，该结构可以成功用以处理大规模数据的原因。

[8-13] 考查含有 2012 个内部节点的红黑树。

a） 该树可能的最小黑高度 d_{min} 是多少？

【解答】

将红黑树中内部节点的总数记作N，将其黑高度记作d。

若考查与之相对应的4阶B-树，则该B-树中存放的关键码恰有N个，且其高度亦为d。于是，再次根据教材8.2.4节的分析结论，最小黑高度应为：

$$d_{min} = \lceil \log_4(N + 1) \rceil = \lceil \log_4 2013 \rceil = 6$$

b） 该树可能的最大黑高度 d_{max} 是多少？

【解答】

与上同理，最大黑高度应为：

$$d_{max} = 1 + \lfloor \log_{4/2} \lfloor (N + 1)/2 \rfloor \rfloor$$
$$= 1 + \lfloor \log_2 \lfloor 2013/2 \rfloor \rfloor = 1 + \lfloor \log_2 1006 \rfloor = 10$$

c） 该树可能的最小高度 h_{min} 是多少？

【解答】

根据习题[7-3]，从常规二叉搜索树的角度看，树高不低于：

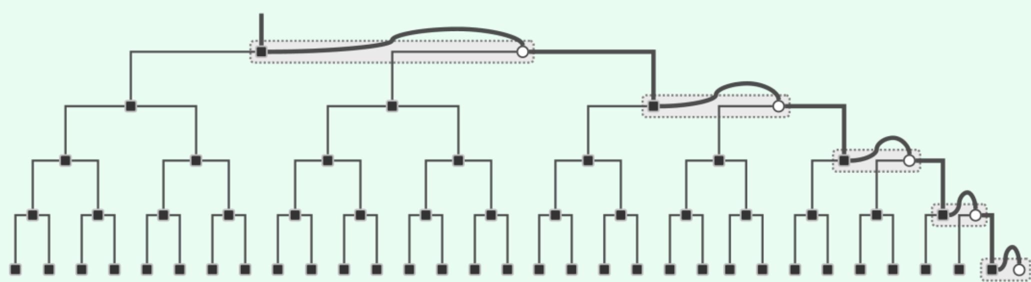
$$h_{min} = \lfloor \log_2 N \rfloor = \lfloor \log_2 2012 \rfloor = 10$$

当然，还需具体地构造出这样的一棵红黑树——这项任务请读者独立完成。

d） 该树可能的最大高度 h_{max} 是多少？

【解答】

我们来考查与原问题等价的逆问题：若高度固定为h，红黑树中至少包含多少个节点。不妨仍然考查与红黑树的对应的4阶B-树。



图x8.4 高度（计入扩充的外部节点）为10的红黑树，至少包含62个节点

先考查h为偶数的情况。如图x8.4所示，该B-树的高度应为h/2；其中几乎所有节点均只含单关键码；只有h/2个节点包含两个关键码（分别对应于原红黑树中的一个红、黑节点），它们在每一高度上各有一个，且依次互为父子，整体构成一条路径（这里不妨以最右侧通路为例）。于是，该B-树所含关键码（亦即原红黑树节点）的总数为：

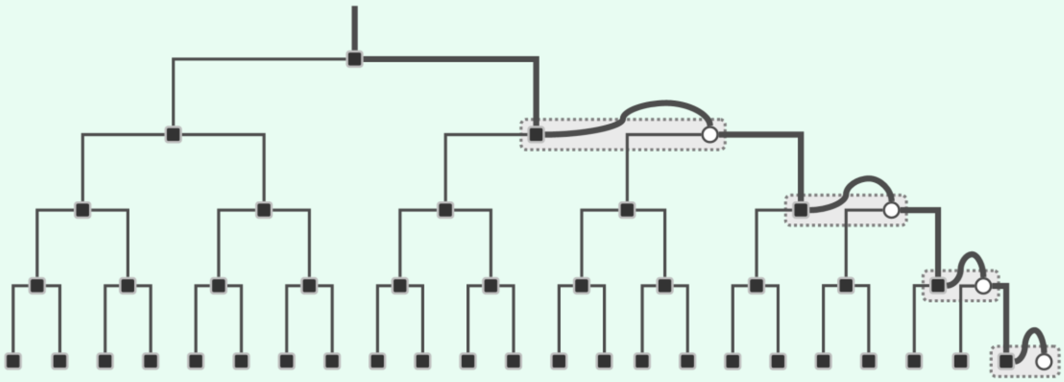
$$N_{min} = 2 \times (1 + 2 + 4 + 8 + 16 + \ldots + 2^{h/2 - 1}) = 2^{h/2 + 1} - 2$$

例如，如图x8.4所示的红黑树高度为10，对应B-树高度为5，所含关键码（节点）总数为：

$$N_{min} = 2^{10/2+1} - 2 = 2^{5+1} - 2 = 62$$

因此反过来，当节点总数固定为N时，最大高度不过

$$h_{max} = 2\cdot(\lfloor\log_2(N+2)\rfloor - 1) \text{ ......(1)}$$



图x8.5 高度（计入扩充的外部节点）为9的红黑树，至少包含46个节点

再考查h为奇数的情况。如图x8.5所示，该B-树的高度应为(h + 1)/2；其中几乎所有节点均只含单关键码；只有(h - 1)/2个节点包含两个关键码（分别对应于原红黑树中的一个红、黑节点），除了根节点，它们在每一高度上各有一个，且依次互为父子，整体构成一条路径（同样地，以最右侧通路为例）。于是，该B-树所含关键码（亦即原红黑树节点）的总数为：

$$N_{min} = 2 \times (1 + 2 + 4 + 8 + 16 + \ldots + 2^{(h-1)/2-1}) + 2^{(h+1)/2-1}$$
$$= 3\cdot2^{(h-1)/2} - 2$$

例如，如图x8.5所示的红黑树高度为9，对应B-树高度为5，所含关键码（节点）总数为：

$$N_{min} = 3\cdot2^{(h-1)/2} - 2 = 3\cdot2^4 - 2 = 46$$

因此反过来，当节点总数固定为N时，最大高度不过

$$h_{max} = 2\cdot\lfloor\log_2(\frac{N+2}{3})\rfloor + 1 \text{ ......(2)}$$

综合(1)和(2)两式可知，在N = 2012是，应有：

$$h_{max} = \max(\ 2\cdot(\lfloor\log_2(2012+2)\rfloor - 1),\ 2\cdot\lfloor\log_2(\frac{N+2}{3})\rfloor + 1\ )$$
$$= \max(18,\ 19)$$
$$= 19$$

读者不妨按照以上分析，示意性地绘出该红黑树（及其对应B-树）的结构。

[8-14] 就最坏情况而言，红黑树在其重平衡过程中可能需要对多达Ω(logn)个节点做重染色。然而，这并不足以代表红黑树在一般情况下的性能。

试证明，就分摊意义而言，红黑树重平衡过程中需重染色的节点不超过$\mathcal{O}(1)$个。

【解答】

不妨从初始为空开始，考查对红黑树的一系列插入和删除操作，将操作总数记作m >> 2。可以证明：存在常数c > 0，使得在此过程中所做的重染色操作不超过cm次。

为此，可以使用习题[8-2]的方法，定义势能函数如下：

$$\Phi(S) \;=\; 2\cdot BRR(S) + BBB(S)$$

其中，BBR(S)为当前状态S下，拥有两个红孩子的黑节点总数；BBB(S)则为当前状态S下，拥有两个黑孩子的黑节点总数。

不难验证，以上势能函数始终非负，且初始值为零。

为得出题中所述结论，只需进一步验证：每做一次重染色，无论属于何种情况，该势能函数都会至少减少1个单位；另外，每经过一次插入或删除操作，该势能函数至多会增加常数c个单位。请读者对照教材第8.3.3节和第8.3.4节中所列的各种情况，独立完成对以上性质的验证。

[8-15] 试证明，若中位点能够在线性时间内确定，则kd-树构造算法buildKdTree()（242页算法8.1）的总体执行时间可改进至$\mathcal{O}(n\log n)$，其中n = |P|为输入点集的规模。

【解答】

如此，在该分治式算法中，每个问题（kd-树的构造）都能在线性时间内均衡地划分为两个子问题（子树的构造）；而且子问题的解（子树）都能在常数时间内合并为原问题的解（kd-树）。于是，其时间复杂度T(n)所对应的递推式为：

$$T(n) \;=\; 2\cdot T(n/2) + \mathcal{O}(n)$$

解之即得：

$$T(n) \;=\; \mathcal{O}(n\log n)$$

[8-16] 关于kd-树查找算法kdSearch()（教材244页算法8.2），试证明以下结论：

a） 在树中某一节点发生递归，当且仅当与该节点对应的子区域，与查询区域的边界相交；

【解答】

按照该算法的控制逻辑，只要当前子区域与查询区域R的边界相交时，即会发生递归；反之，无论当前子区域是完全处于R之外（当前递归实例直接返回），还是完全处于R之内（直接遍历当前子树并枚举其中所有的点），都不会发生递归。

b） 若令Q(n) = 规模为n的子树中与查询区域边界相交的子区域（节点）总数，则有：

$$Q(n) \;=\; 2 + 2Q(n/4) \;=\; \mathcal{O}(\sqrt{n})$$

【解答】

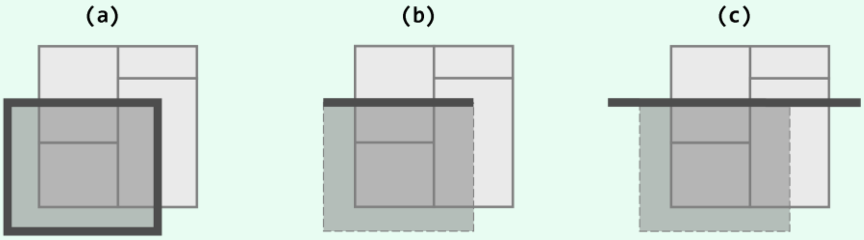
设R为任一查询区域。

根据其所对应子区域与R边界的相交情况，kd-树中的所有节点可以划分以下几类：

(0) 与R的边界不相交
(1) 只与R的一条边相交
(2) 同时与R的多条边相交

根据a），其中第(0)类节点对Q(n)没有贡献。

如图x8.6(a)所示，第(1)类节点又可以细分为四种，分别对应于R的上、下、左、右四边。既然是估计渐进复杂度，不妨只考虑其中一种——比如，如图(b)所示，只考查水平的上边。



图x8.6 统计与查询区域边界相交的子区域（节点）总数

根据定义，kd-树自顶而下地每经过k层，切分的维度方向即循环一轮。因此，不妨考查与R边界相交的任一节点，以及自该节点起向下的k代子孙节点。对于2d-树而言，也就是考查与R边界相交的任一节点，以及它的2个子辈节点（各自大致包含n/2个点）和4个孙辈节点（各自大致包含n/4个点）。

为简化分析，我们不妨如图(c)所示，进一步地将R的上边延长为其所在的整条直线。于是不难发现，无论这4个孙辈节点（子区域）的相对位置和大小如何，该直线至多与其中的2个相交；反过来，至少有两个节点（子区域）不再发生递归。于是，即可得到如下递推关系：

$$Q(n) \leq 2 + 2 \cdot Q(n/4) \quad \cdots\cdots (*)$$

再结合边界条件：

$$Q(1) = 1$$

解之即得：

$$Q(n) = \mathcal{O}(\sqrt{n})$$

请注意，以上并未统计第(2)类节点（子区域），但好在这类节点只占少数，就渐进的意义而言，并不影响总体的上界。

比如在图x8.6(a)中，包含R四个角点的那些节点（子区域）即属此列。以其中包含R左上角者为例，这类节点在kd-树的每一层至多一个，故其总数不超过树高$\mathcal{O}(\log n)$。相对于第一类节点的$\mathcal{O}(\sqrt{n})$，完全可以忽略。

当然，第(2)类节点（子区域）还有其它可能的情况，比如同时包含R的多个角点。但不难说明，其总数依然不超过$\mathcal{O}(\log n)$。

c) kdSearch()的运行时间为：$\mathcal{O}(r + \sqrt{n})$

其中 r 为实际命中并被报告的点数。

【解答】

从递归的角度看，若忽略对reportSubtree()的调用，kd-树范围查询算法的每一递归实例本身均仅需$\mathcal{O}(1)$时间。故由以上b）所得结论，查询共需$\mathcal{O}(\sqrt{n})$时间。

reportSubtree(v)是通过遍历子树v，在线性时间内枚举其中的命中点。整个算法对该例程所有调用的累计时间，应线性正比于输出规模r。

两项合计，即得题中所述结论。

d） 进一步地，试举例说明，单次查询中的确可能有多达$\Omega(\sqrt{n})$个节点发生递归，故以上估计是紧的。

【解答】

为确切地达到这一紧界，以上b）中所得递推式(*)必须始终取等号；反之，只有该递推式始终取等号，则必然可以实现紧界。请读者按照这一思路，独立给出具体实例。

需要指出的是，由此结论也可看出，c）中所做的简化与放大，在渐进意义上都是紧的。

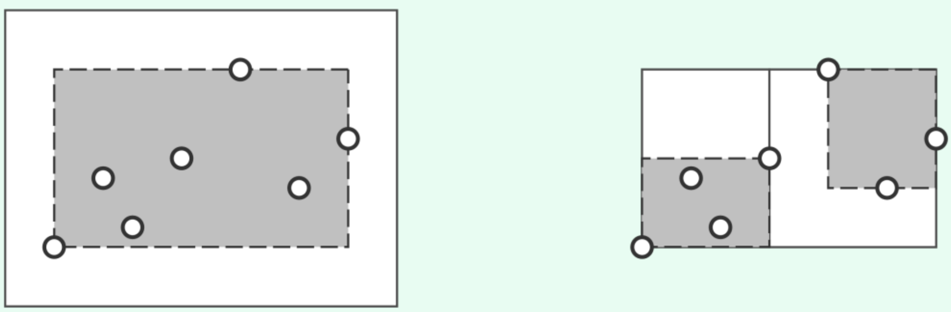
e） 若矩形区域不保证与坐标轴平行，甚至不是矩形（比如圆），则上述结论是否依然成立？

【解答】

依然成立。具体的分析方法及过程，可以参见[46]。

[8-17] 不难理解，kd-树中节点v所对应的矩形区域即便与查询范围R相交，其中所含的输入点也不见得会落在R之内。比如在极端的情况下，v中可能包含大量的输入点，但却没有一个落在R之内。当然，kdSearch()（教材244页算法8.2）在这类情况下所做的递归，都是不必进行的。

克服这一缺陷的一种简明方法，如图x8.7所示：在依然保持各边平行于坐标轴，同时所包含输入点子集不变的前提下，尽可能地收缩各矩形区域。其效果等同于，将原矩形替换为依然覆盖其中所有输入点的最小矩形——即所谓的包围盒（bounding-box）。其实，在如教材图8.41所示的实例中，正因为采用了这一技巧，才得以在节点{F, H}处，有效地避免了一次无意义的递归。



图x8.7 每次切分之后，都随即将子区域（实线）替换为包围盒（虚线），以加速此后的查找

试按照以上构思，在教材242页算法8.1的基础上，改进kd-树的构造算法。

【解答】

请读者参照以上介绍和提示，独立完成编码和调试任务。

[8-18] 若仅需报告落在指定范围内点的数目，而不必给出它们的具体信息，则借助kd-树需要多少时间？

【解答】

只需$\mathcal{O}(\sqrt{n})$时间。

既然无需具体地枚举所命中的点，故可令kd-树的每一节点分别记录其对应子树中所存放的点数。这样，对于经查找而被筛选出来的每一棵子树，都可以直接累计其对应的点数，而不必对其进行遍历。如此，原先消耗于遍历枚举的$\mathcal{O}(r)$时间即可节省；同时，对各子树所含点数的累加，耗时不超过被筛选出来的子集（子树）总数——亦即$\mathcal{O}(\sqrt{n})$。

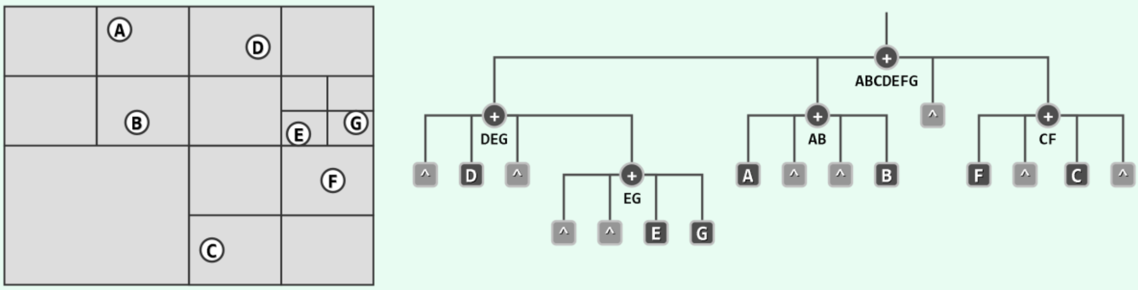
[8-19] 四叉树[51]（quadtree）是 2d-树的简化形式，其简化策略包括：

❶ 直接沿区域的（水平或垂直）平分线切分，从而省略了中位点的计算

❷ 沿垂直方向切出的每一对节点（各自再沿水平方向切分）都经合并后归入其父节点

❸ 被合并的节点即便原先（因所含输入点不足两个）而未继续切分，在此也需要强行（沿水平方向）切分一次

于是如图 x8.8 所示，每个叶节点各含 0 至 1 个输入点；每个内部节点则都统一地拥有四个孩子，分别对应于父节点所对应矩形区域经平均划分之后所得的四个象限，该树也由此得名。

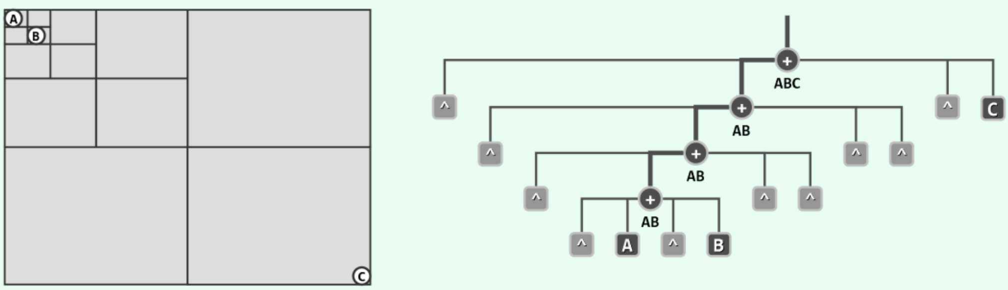


图x8.8 通过递归地将平面子区域均分为四个象限（左），构造对应的四叉树（右）

a） 与 kd-树不同，四叉树可能包含大量的空（即不含任何输入点的）节点。更糟糕的是，此类节点的数目无法仅由输入规模 n 界定。对于任意的 N > 0，试构造一个仅含 n = 3 个点的输入点集，使得在其对应的四叉树中，空节点的数目超过 N 个。

【解答】

这样的一个实例，如图x8.9所示。实际上，以此为基础，可以导出一系列的此类实例。



图x8.9 四叉树的空间利用率可能极低

这里只有A、B和C三个点，但A和B之间的距离非常接近，以至于必须持续划分4次，它们才不再属于同一子区域。在如此构造出来的四叉树中，每一层都有1个内部节点和3个叶节点；而除了最高的两层和最低的一层，其余各层的3个叶节点都对应于空的子区域。

限于篇幅，这里所给实例的深度仅为4。实际上仿照此例的构思，不难扩展并得出层数更多的例子，其中存放的依然只有三个点，但其空间利用率却可以无限地接近于0。为此，只需不断地令点A和点B相互靠近。当然，最为极端的情况也就是这两个点彼此重合。

作为对照，读者不妨绘出同样存放这三个点的kd-树，并体会二者在空间效率方面的差异，以及导致这种差异的根本原因。

为消除这一缺陷，可以仿照kd-树，将点集的划分策略由“按空间平分”改为“按点数平分”。尽管如此需要额外地记录各节点所对应的划分位置，但却可以严格地保证划分的均衡性，从而有效地提高整体的空间效率。

b） 对于任一输入点集P，若将其中所有点对的最长、最小距离分别记作D和d，则$\lambda = D/d$称作P的散布度（spread）。试证明，P所对应的四叉树高度为$\mathcal{O}(\log\lambda)$。

【解答】

与kd-树一样，四叉树中的每个节点也唯一地对应于某个矩形子区域；同一深度上各节点所对应的子区域面积相等（或渐进地同阶），彼此无交，且它们的并覆盖整个空间。

其中，根节点对应的子区域，边长为D；其下4个子节点所对应的子区域，边长为D/2；再下一层的16个孙辈节点所对应的子区域，边长为D/4；...；最底层（叶）节点所对应的子区域，边长为d（或者更严格地，d/2）。

由此可见，整个四叉树的高度不超过$\mathcal{O}(\log\lambda)$。

由此反观以上a）中实例，导致其中空节点过多的直接原因，也可以认为在于d相对于D过小，以致于散布度λ以及树高过大。

c） 按照以上描述，试用C/C++语言实现四叉树结构。

【解答】

请读者参照以上介绍和提示，独立完成编码和调试任务。

d） 试基于四叉树结构设计相应的范围查询算法，并利用你的四叉树结构实现该算法。

【解答】

与基于kd-树的查询算法基本相同。

从递归的角度来看，对于任一节点（子区域）的查询任务，都可以分解为对4个子节点（细分子区域）的查询子任务。其中，有些子任务需要继续递归（子区域与查询区域的边界相交），有些子任务则可立即以失败返回（子区域完全落在查询区域以外），有些子任务则可立即以成功返回（子区域完全落在查询区域以内）。

请读者参照以上介绍和提示，独立完成编码和调试任务。

e） 针对范围查询这一应用，试分别从时间、空间效率的角度，将四叉树与 2d-树做一比较。

【解答】

以上d）所给算法的原理与过程，尽管与采用kd-树的算法基本相同，但却有着本质的区别，从而导致其时间、空间性能均远不如kd-树。主要的原因，具体体现在以下方面。

首先由a）可见，四叉树中存在大量的空节点（子区域），因此在查找过程中即便能够确定某一节点（子区域）完全落在查询区域内部，也不能在线性时间内枚举出其中有效的各点。整体而言，不可能在$\mathcal{O}(r)$时间内枚举出所有的命中点——而且，通常情况会远远超过$\mathcal{O}(r)$。

另外由b）可见，（若不做改进）四叉树的高度取决于点集的散布度λ，而不是点集的规模。因此树高没有明确的上限，递归深度及查找长度也难以有效控制。在各点分布极其不均匀的场合，树高往往会远远超过kd-树的$\mathcal{O}(\log n)$。

以下对其平均情况做一估计。

不妨假定所有点均取自单位正方形$[0, 1] \times [0, 1]$，对应的四叉树高度为h。查询矩形区域R的长度和宽度分别为x和y。

在深度为k的任一层（$0 \leq k \leq h$），共有4^k个节点，分别对应于4^k个互不相交的子正方形（有些不含任何点），面积统一为4^{-k}。故节点总数为：

$$N = \sum_{k=0}^{h} 4^k = (4^{h+1} - 1) / 3 \sim 4^{h+1}/3$$

在深度为k的一层，与查询区域R相交（并因此需要耗费时间）的节点总数大致为：

$$(x \cdot 2^k + 1) \cdot (y \cdot 2^k + 1) = xy \cdot 4^k + (x + y) \cdot 2^k + 1$$

故所有各层与R相交者的总数大致为：

$$\begin{aligned} & \sum_{k=0}^{h} [xy \cdot 4^k + (x + y) \cdot 2^k + 1] \\ \sim\ & xy \cdot 4^{h+1}/3 + (x + y) \cdot 2^{h+1} + (h + 1) \\ =\ & xy \cdot N + (x + y) \cdot \sqrt{3N} + \log_4(3N) \\ =\ & \mathcal{O}(xy \cdot N) \end{aligned}$$

主要取决于查询区域R的面积xy，以及四叉树的划分粒度N（如上分析，取决于散布度λ）。

f） 试将上述思路推广至三维的情况，以三层为间隔对 3d-树的节点做类似的合并，从而实现所谓的八叉树（octree）结构。

【解答】

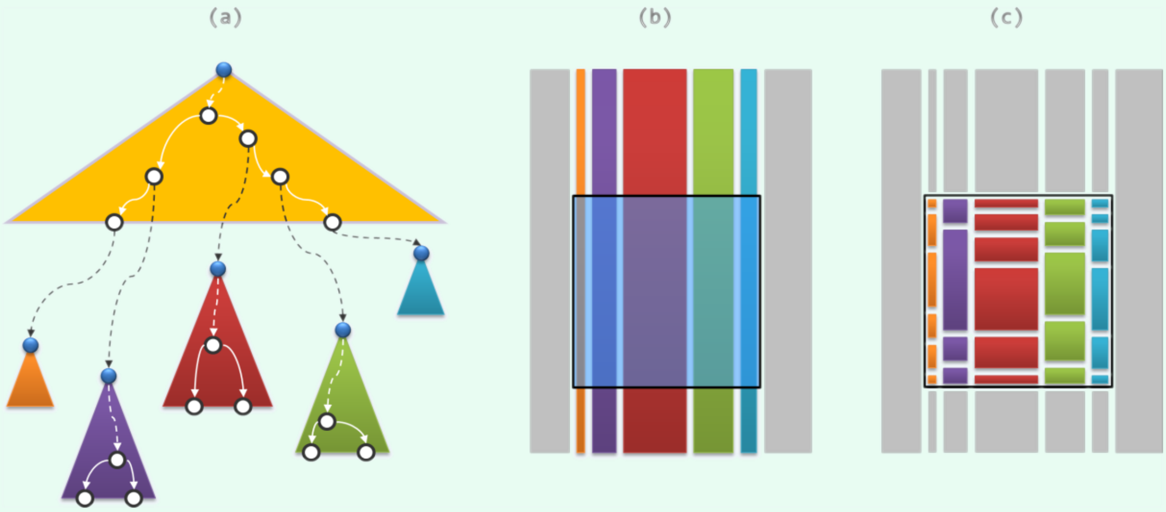
基本原理、方法与技巧，与四叉树完全一致。

请读者参照以上介绍和提示，独立完成编码和调试任务。

[8-20] 范围查询的另一解法需要借助范围树（range tree）[48]。

为此，首先仿照如图8.37（教材240页）和图8.38（教材241页）所示的策略，按x坐标将平面上所有输入点组织为一棵平衡二叉搜索树，称作主树（main tree）。

于是如图x8.10(a)和(b)所示，该树中每个节点各自对应于一个竖直的条带区域；左、右孩子所对应的条带互不重叠，均由父节点所对应的条带垂直平分而得；同一深度上所有节点所对应的条带也互不重叠，而且它们合并后恰好覆盖整个平面。



图x8.10 利用范围树，可以实现更加高效的范围查询

接下来，分别对于主树中每一节点，将落在其所对应条带区域中的输入点视作一个输入子集，并同样采用以上方法，按照 y 坐标将各个子集组织为一棵平衡二叉搜索树，它们称作关联树（associative tree）①。于是如图x8.10(a)和(c)所示，每棵关联树所对应的竖直条带，都会进而逐层细分为多个矩形区域，且这些矩形区域也同样具有以上所列主树中各节点所对应条带区域的性质。至此，主树与这$\mathcal{O}(n)$棵关联树构成了一个两层的嵌套结构，即所谓的范围树。

利用范围树，可按如下思路实现高效的范围查询。对于任一查询范围$R = [x_1, x_2] \times [y_1, y_2]$，首先按照$[x_1, x_2]$对主树做一次 x 方向的范围查询。根据 8.4.1 节的分析结论，如此可以得到$\mathcal{O}(\log n)$个节点，而且如图x8.10(b)所示，它们所对应的竖直条带互不重叠，它们合并后恰好覆盖了x坐标落在$[x_1, x_2]$范围内的所有输入点。

接下来，深入这些节点各自对应的关联树，分别按照$[y_1, y_2]$做一次 y 方向的范围查询。如此从每棵关联树中取出的一系列节点，也具有与以上取自主树的节点的类似性质。具体地如图x8.10(c)所示，这些节点所对应的矩形区域互不重叠，且它们合并之后恰好覆盖了当前竖直条带内 y 坐标落在$[y_1, y_2]$范围内的所有输入点。换而言之，这些点合并之后将给出落在 R 中的所有点，既无重也不漏。

① 关联树的引入，只是为了便于将此结构推广至更高维度；就此特定的二维情况而言，完全可以代之以简单的有序向量

a) 按照以上描述，试用 C/C++语言实现二维的范围树结构；

【解答】

请读者参照以上介绍和提示，独立完成编码和调试任务。

b) 试证明，如此实现的范围树，空间复杂度为 $\mathcal{O}(n\log n)$；

【解答】

显然，主树自身仅需$\mathcal{O}(n)$空间。

这里的关联树共计有n棵，表面上看，每一棵的规模都可能达到$\Omega(n)$。然而以下将证明，总体空间$\mathcal{O}(n^2)$的上界远远不紧，更紧的上界应为$\mathcal{O}(n\log n)$。

以上之所以得出$\mathcal{O}(n^2)$这一不紧的上界，是因为我们采用的统计方法是：

> 对每一棵关联树，统计有多少个点可能出现在其中

为得出更紧的上界，我们不妨颠倒思路，采用如下统计方法：

> 对于每一个点，统计它可能出现在多少棵关联树中

稍作观察即不难发现，任一点p出现在某一关联树中，当且仅当在主树中，该关联树对应的节点是p所对应叶节点的祖先。而在平衡二叉搜索树中，每个节点的祖先均不超过$\mathcal{O}(\log n)$个。

c) 按照以上描述，试利用你的范围树实现新的范围查询算法；

【解答】

与kd-树的查询算法类似。

首先沿x方向做一次（一维的）范围查找，并在主树中挑选出不超过$\mathcal{O}(\log n)$个节点。

然后，对于其中的每个节点，在与之对应的关联树中，沿y方向各做一次（一维的）范围查找。关联树中每一棵命中的子树，都可通过遍历在线性时间内枚举其中节点。

d) 试证明，以上范围查询算法的时间复杂度为 $\mathcal{O}(r + \log^2 n)$，其中 r 为实际命中并被报告的点数；

【解答】

按照如上算法，可知如下性质：

> 1) 对主树的查找耗时$\mathcal{O}(\log n)$
> 2) 对$\mathcal{O}(\log n)$棵关联树的查找分别耗时$\mathcal{O}(\log n)$，累计耗时$\mathcal{O}(\log^2 n)$

再计入遍历枚举所需的$\mathcal{O}(r)$时间，即得题中待证的结论。

e） 继续改进[②]以上范围树，在不增加空间复杂度的前提下，将查询时间减至 $\mathcal{O}(r + \log n)$[49][50]。

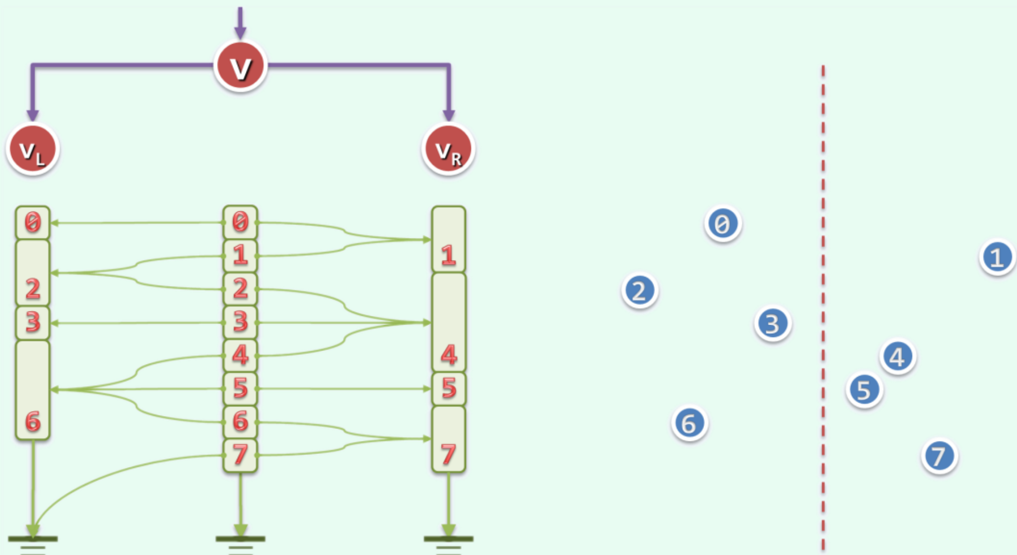
（提示：尽管每次查询均涉及 $\mathcal{O}(\log n)$次 y 坐标的范围查询，但其查找区间都同为$[y_1, y_2]$）

【解答】

正如以上提示所指出的：

在每一次范围查询中，在所涉及关联树的查找之间，具有极强的相关性
——它们的入口参数同为$[y_1, y_2]$

因此可如图x8.11所示，借助分散层叠（fractional cascading）的技巧加以改进。为此，需要在主树中每一对父子节点所对应的关联树之间，增加一系列的索引。



图x8.11 通过分散层叠，进一步提高范围树的查找性能

具体地如图x8.11所示，设主树中的节点v_L和v_R是v的左、右孩子；它们各自对应的关联树，则简化地表示为有序向量（等效于关联树的中序遍历序列）。于是，在v关联树中查找结果，可以直接为其孩子节点的关联树直接利用，相应地查找成本由$\mathcal{O}(\log n)$降至$\mathcal{O}(1)$。当然，对于最低公共祖先节点所对应的那棵关联树，还是需要做一次$\mathcal{O}(\log n)$的查找。

综上所述，改进之后的范围树可在：

$$\mathcal{O}(\log n + \log n) \quad = \quad \mathcal{O}(\log n)$$

时间完成查找，并在：

$$\mathcal{O}(r)$$

时间内报告查询结果。

② 严格地说，只有在经过如此改进之后方可称作范围树，否则只是一般的多层搜索树（multi-level search tree）