第10章

优先级队列

[10-1] a） 试按照代码 10.1 中的 ADT 接口，分别基于无序、有序列表和无序、有序向量实现优先级队列；

【解答】

请读者利用此前各章实现的列表及向量等数据结构，独立完成编码和调试任务。

b） 你所实现操作接口的时间复杂度各是多少？

【解答】

请读者针对自己的实现方式，给出对应的分析和结论。

c） 基于这些结构，可否使 getMax()接口的效率达到 $\mathcal{O}(1)$，同时 delMax()和 insert()接口的效率达到 $\mathcal{O}(\log n)$？

【解答】

列表和向量只能提供基本的数据操作，难以简明地直接同时兼顾以上接口的高效率。

有趣的是，只要采用教材10.2节的方法，将利用向量维护一个逻辑上的完全二叉堆，即可高效地同时支持以上操作接口。

[10-2] 基于向量实现完全二叉堆时，也可在向量中将各节点顺次后移一个单元，并在腾出的首单元中置入对应元素类型的最大值作为哨兵（比如，对于整型可取 INT_MAX）。如此，虽然多使用了一个单元，但在上滤过程中只需比较父子节点的大小，而无需核对是否已经越界。

a） 经如此转换之后，父子节点各自在物理上所对应的秩之间的换算关系，应如何调整？

【解答】

只需在教材10.2.1节所给方法的基础上，略作调整。

具体地，若节点v的编号（秩）记作i(v)，则根节点及其后代节点的编号分别为：

```
i(root)  =  1
i(lchild(root))  =  2
i(rchild(root))  =  3
i(lchild(lchild(root))  =  4
...
```

一般地有：

```
1）若节点v有左孩子，则i(lchild(v))  =  2·i(v);
2）若节点v有右孩子，则i(rchild(v))  =  2·i(v) + 1;
3）若节点v有父节点，则i(parent(v))  =  ⌊i(v)/2⌋  =  ⌈((i(v) - 1)/2⌉
```

b） 试在本章对应代码的基础上略作修改，实现上述改进；

【解答】

请读者根据以上介绍和提示，独立完成编码及调试任务。

c） 如此改进之后，insert()和 delMax()操作的时间复杂度有何变化？总体效率呢？

【解答】

如此调整之后，上滤的过程无需核对是否已经越界，因此可以在一定程度上提高插入操作的效率，但渐进时间复杂度依然保持为$\mathcal{O}(\log n)$。当然，以上调整对下滤的过程及效率没有影响。

d） 对于不易甚至无法定义最大值的元素类型（比如长度任意的字符串），以上技巧是否依然适用？

【解答】

不再适用。

[10-3] 如教材代码 10.7、代码 10.9 实现的 percolateUp、percolateDown 算法中，若实际上升或下降 k = $\mathcal{O}(\log n)$层，则 k 次 swap()操作共需 3k 次赋值。

试改进以上实现，将此类赋值操作降至 k + 1 次。

【解答】

在插入接口的上滤过程中，新元素可暂且不予插入，而只是将其上若干代祖先节点依次下移；待所有祖先均已就位之后，才将新元素置入腾出的空节点。删除接口的下滤过程，与此同理。

请读者按照以上提示，独立完成编码和调试任务。

[10-4] a） 试证明，在从堆顶通往任一叶节点的沿途上，各节点对应的关键码必然单调变化；

【解答】

由堆序性，显然。

b） 试给出一个算法，对于秩为 r 的任一节点，在$\mathcal{O}(1)$时间内确定其在任何高度 h 上祖先的秩；

【解答】

考查基于向量实现的任一完全二叉堆。

我们注意到，其中祖先与后代节点的秩之间存在某种关联关系。具体地，只需令各节点的秩统一地递增一个单位，则从秩的二进制表示的角度来看，祖先必是后代的前缀。

以如教材287页图10.2所示的完全二叉堆为例，考查其中节点18所对应的查找路径，沿途各节点的秩依次为：

0,	1,	3,	8,	18

统一递增之后，依次为：

1,	2,	4,	9,	19

对应的二进制表示依次为：

1,	10,	100,	1001,	10011

位数依次递增。而且更重要的是，相邻的每一对中，前者总是后者的前缀。

因此，对于任何秩为r的元素，其上溯第h代祖先（若存在）所对应的秩必然为：

```
(r + 1) >> h  -  1
```

c）试改进 percolateUp 算法（代码 10.7），将其中执行的关键码比较减少至 $O(\log\log n)$ 次；

【解答】

利用b）所指出的特性，在引入新节点但尚未上滤调整之前，可以将该节点对应的查找路径视作一个静态查询表，并使用二分查找算法。

具体地，每次都可在$O(1)$时间内确定高度居中的祖先的秩；将其当作轴点，只需再做$O(1)$次比较，即可将查找范围缩小一半。如此反复迭代，直至查找范围内仅剩单个节点。

既然完全二叉堆的高度$h = O(\log n)$，故整个查找过程的迭代（比较操作）次数将不超过：

$$\log h = O(\log\log n)$$

请注意，因为任一节点通往其后代的路径并不唯一，故这一技巧并不适用于下滤操作。

d）经过以上改进，percolateUp 算法总体的渐进复杂度是否有所优化？

【解答】

以上方法固然可以有效地减少词条的比较操作，但词条交换操作却不能减少。事实上无论如何，在最坏情况下，依然需要执行$O(h) = O(\log n)$次交换操作。

由此可见，percolateUp算法总体的渐进复杂度将保持不变。

e）试通过实验确定，只有在完全二叉堆达到多大规模之后，以上改进才能实际地体现出效果。

【解答】

请读者独立完成编码和调试工作，并根据实测结果给出分析和结论。

需要指出的是，对于通常的应用问题规模n而言，$\log n$与$\log\log n$均已十分接近于常数，二者之间的差异并不明显。

以$n = 2^{32} = 4 \times 10^9$为例，有：

$$\log_2 n = 32$$

$$\log_2(\log_2 n) = 5$$

若再综合考虑到以上方法针对秩所额外引入的计算量，实际性能的差异将更不明显。

[10-5] 在摘除原堆顶元素后，为恢复堆的结构性，为何采用如教材 292 页代码 10.9 所示的 percolateDown()算法，而不是自上而下地，依次以更大的孩子节点顶替空缺的父节点？

【解答】

若仅就堆序性而言，这种调整方式并非不可行。

然而遗憾的是，经如此调整之后二叉堆的拓扑结构，未必依然是一棵完全二叉树，故其结构性将可能遭到破坏。

以如教材292页图10.6之a)所示的大顶堆为例，不难验证，在摘除堆顶元素（5）之后，若采用本题所建议的方法进行调整，则所得二叉堆将不再是一棵完全二叉树。

[10-6] 针对如教材第 290 页代码 10.7 所示的 percolateUp()上滤算法，10.2.2 节曾指出其执行时间为 $\mathcal{O}(\log n)$。然而，这只是对其最坏情况的估计；在通常的情况下，实际的效率要远高于此。试通过估算说明，在关键码均匀独立分布时，最坏情况极其罕见，且插入操作平均仅需常数时间。（提示：参考文献[3]）

【解答】

在此仅做一个粗略的估算。

根据堆的定义及调整规则，若新节点p通过上滤升高了k层，则意味着在2^{k+1}个随机节点（p的父亲、p，以及p的$2^{k+1} - 2$个后代）中，该节点恰好是第二大者。

于是，若将新节点累计上升的高度记作H，则H恰好为k的概率应为：

$$\Pr(H = k) = 1/2^{k+1} = (1/2)^k \cdot (1/2), \quad 0 \le k$$

这是一个典型的几何分布（geometric distribution），其数学期望为：

$$E(H) = 1/(1/2) - 1 = 1$$

也就是说，每个节点经上滤后平均大致上升1层，其间平均需做1 + 1 = 2次比较操作。

[10-7] Floyd 建堆算法中，同层内部节点下滤的次序

a）对建堆结果有无影响？若无影响，试说明原因；否则，试举一实例。

【解答】

没有影响。

同层节点的下滤，仅涉及到其各自的后代，它们之间完全相互独立，故改变次序不致影响最终的结果。

b）对建堆所需时间有无影响？若无影响，试说明原因；否则，试举一实例。

【解答】

也没有影响。

每个节点的下滤过程完全不变，所需时间不变，建堆所需的总体时间亦不变。

[10-8] 借助优先级队列高效的标准接口，教材 285 页代码 10.2 中的 generateTree()算法即可简明地在 $\mathcal{O}(n\log n)$时间内构造出 n 个字符的 Huffman 编码树。然而，这还不足以说明这一实现已属最优。试证明，任何 CBA 式 Huffman 树构造算法，在最坏情况下都需要运行$\Omega(n\log n)$的时间。

【解答】

只需建立一个从排序问题，到Huffman编码问题的线性归约（习题[2-12]）。

事实上，对于每一个待排序的输入序列，我们都将其视作一组字符的出现频率。不失一般性，这里可以假设每个元素均非负——否则，可以在$\mathcal{O}(n)$时间内令它们增加同一足够大的正数。

以下，以这组频率作为输入，可以调用任何CBA式算法构造出Huffman编码树。而一旦得到这样一棵编码树，只需一趟层次遍历，即可在$\mathcal{O}(n)$的时间内得到所有（叶）节点的遍历序列。根据Huffman树的定义，该序列必然是单调的。因此，整个过程也等效于同时完成了对原输入序列的排序。

[10-9] 在附加某些特定条件之后，问题的难度往往会有实质的下降。比如，若待编码字符集已按出现频率排序，则 Huffman 编码可以更快完成。在编码过程中，始终将森林 $\mathcal{F}$ 中的树分为两类：单节点（尚未参与合并）和多节点（已合并过）。每经过一次迭代，后者虽不见得增多，但必然有一个新成员。

a） 试证明，在后一类树中，新成员的权重（频率）总是最大；

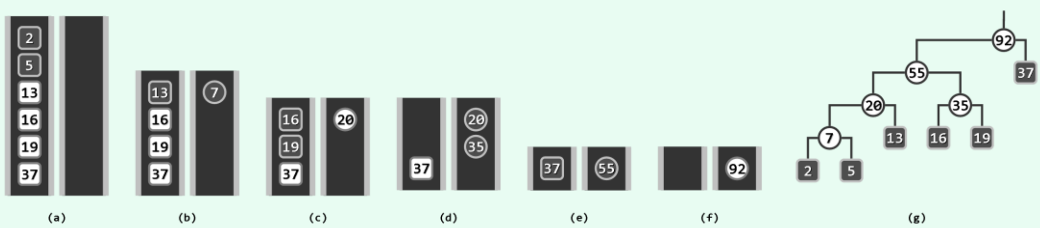
【解答】

根据Huffman编码算法的原理，每次迭代都是在当前森林中选取权重最小的两棵树做合并。因此，被选出的树的权重必然单调非降，故在当前所有（经合成生成的）多节点树中，最新者的权重必然最大。

b） 试利用以上性质设计一个算法，在 $\mathcal{O}$(n)时间内完成 Huffman 编码。

【解答】

将如上定义的两类节点，按权重次序组织为两个队列。初始状态如图x10.1(a)所示，所有字符都按照权重非降的次序，存入单节点树的队列（左）；而多节点树的队列（右），直接置空。此后的过程与常规的Huffman编码算法类似，也是反复地取出权重最小的两棵树，将其合并后插回森林。直至最后只剩一棵树。



图x10.1 字符权重已排序时，可在线性时间内构造出Huffman编码树

这里与常规算法的本质不同，共有两点。首先，每次只需考查两个队列各自最前端的两棵树。也就是说，每次只需检查不超过四棵树，即可在$\mathcal{O}(1)$的时间内挑选出整个森林中权重最小的两棵树。另外，这两棵树合并之后，直接作为末元素插入多节点树的队列。根据a）的分析结论，这样依然可以保持该队列的单调性。

作为一个完整的实例，图x10.1(b~f)针对权重集{ 2, 5, 13, 16, 19, 37 }，依次给出了算法各步迭代之后，两个队列的具体组成。最终构造出的Huffman编码树，如图(g)所示。

[10-10] 试利用本章所介绍的各种堆结构，与如代码 10.2（教材 285 页）所示的 Huffman 树统一构造算法 generateTree()一起编译、链接、执行，并就其性能做一统计、对比和分析。

【解答】

请读者独立完成编码和调试任务，并通过实际测量给出分析结论。

[10-11] 与 AVL 树需要借助 bf 记录类似，左式堆也需要设置 npl 记录。然而在实际应用中，这一点既不自然，也影响代码开发与转换的效率。实际上，仿照由 AVL 树引出伸展树的思路，可以在保留左式堆优点的前提下消除 npl 记录，新的结构称作斜堆（skew heap）。

当然，与伸展树一样，斜堆各接口的时间复杂度也需要从分摊的角度加以分析和理解。

试搜集和阅读相关材料，并实现斜堆结构。

【解答】

请读者查阅相关资料，并独立完成编码和调试任务。

[10-12] 某些应用可能要求堆结构提供更多接口，比如提升或降低堆中任一指定词条的优先级。尽管此类调整并不影响堆的结构性，但往往会破坏堆序性，故也需要及时调整并使之恢复为合法的堆结构。试设计一个算法，在任一词条改变优先级后，尽快地恢复全局的堆序性。

（提示：借助上滤和下滤）

【解答】

根据词条优先级的变化方向，相应地套用已有的算法进行调整，以尽快恢复堆序性。

具体地，若优先级增加，则可仿照percolateUp()算法（教材290页代码10.7），对其做上滤调整；反之若优先级降低，则可仿照precolateDown()算法（教材292页代码10.9），对其做下滤调整。

请读者根据以上介绍和提示，独立完成新接口的定义、编码和调试任务。

[10-13] 在本章所给的左式堆模板类中（教材298页代码10.12），建堆操作仅实现了蛮力的$\mathcal{O}(n\log n)$算法。试采用Floyd建堆算法，将这一操作的效率改进至$\mathcal{O}(n)$。

【解答】

请读者仿照heapify()算法（教材294页代码10.10），独立完成编码和调试任务。

[10-14] 教材10.2.5节实现的就地堆排序是稳定的吗？若是，请给出证明；否则，试举一实例。

【解答】

不是稳定的。

在反复摘除堆顶并将末词条转移至堆顶，然后做下滤的过程中，雷同词条之间的相对次序不再保持，故它们在最终所得排序序列中必然是“随机”排列的。

作为一个实例，不妨仿照教材296页的图10.10，考查对堆：

{ 5, 4, 1a, 1b, 1c }

的排序。不难验证，最终所得的排序结果为：

{ 1c, 1a, 1b, 4, 5 }

实际上更糟糕的是，以上“随机性”是堆排序算法固有的不足，难以通过该算法自身的调整予以改进。当然，这一问题也并非不能解决——比如，读者不妨参考136页习题[6-24]中介绍的合成数（composite number）法，给出一种解决的办法。

[10-15] 如教材302页代码10.13所示的左式堆合并算法，采用了递归模式。尽管如此已足以保证合并操作的渐进时间复杂度为$\mathcal{O}(\log n)$，但为进一步提高实际运行效率，试将该算法改写为迭代模式。

【解答】

请读者参照递归算法转换为迭代版本的一般性模式和方法，独立完成编码和调试任务。

[10-16] 若能注意到教材 6.11.5 节 Prim 算法中定义的“优先级数”恰好对应于优先级队列中元素的优先级，即可利用本章介绍的优先级队列，改进如教材 177 页代码 6.8 所示的 $O(n^2)$版本。

具体地，可首先花费 $O(n)$时间，将起点 s 与其余顶点之间的 n - 1 条边组织为一个优先级队列 H。此后的每一步迭代中，只需 $O(\log n)$时间即可从 H 中取出优先级数最小的边（最短桥），并将对应的顶点转入最小支撑树中。不过，随后为了高效地对 H 中与刚转出顶点相关联的每一条边做松弛优化，需要增加一个 decrease(e)接口，在边 e 的优先级数减少后将 H 重新调整成一个堆。

a) 参照如代码 10.7 所示的 percolateUp()上滤算法为堆结构增加 decrease()接口，要求运行时间不超过 $O(\log n)$；

【解答】

这类decrease()接口的工作原理，可参见习题[10-12]。

请读者根据以上介绍和提示，独立完成编码和调试任务。

b) 试证明，如此改进之后 Prim 算法的效率为 $O((n + e)\log n)$，非常适用于稀疏图；

【解答】

按照该算法，取出每个顶点需要$O(\log n)$时间，累计$O(n\log n)$时间。

所有顶点的所有邻接顶点的松弛，在最坏情况下累计需要$O(e\log n)$时间。

两项合计，即得题中结论。

c) 这种改进策略是否也适用于 Dijkstra 算法？

【解答】

同样适用，只需改用Diskstra算法的优先级更新规则。

事实上，推而广之，这种改进策略适用于任何基于优先级搜索（PFS）策略的算法。

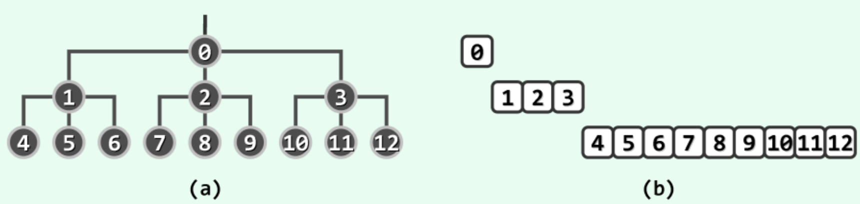
[10-17] 在多叉堆（d-heap）中，每个节点至多可拥有 d ≥ 3 个孩子，且其优先级不低于任一孩子。

a) 试证明，多叉堆 decrease()接口的效率可改进至 $O(\log_d n)$；

（当然，delMax()接口的效率因此会降至 $O(d\cdot\log n)$）

【解答】

如图x10.2所示，我们依然可以仿照二叉堆的方式实现多叉堆。



图x10.2 三叉堆：(a)逻辑结构及(b)物理结构

具体地，将所有元素组织为一个向量，且对于任意秩为k > 0的元素，其父亲对应的秩为：

$$\text{parent}(k) = \lfloor (k - 1)/d \rfloor$$

比如在图x10.2所示的三叉堆中，8号元素的父亲对应的秩为：

$$\text{parent}(8) \;=\; \lfloor(8 - 1)/3\rfloor \;=\; 2$$

反过来，对于任意秩为$k < \lceil(n - 1)/d\rceil$的元素，其第i个孩子（若存在）对应的秩为：

$$\text{child}(k,\ i) \;=\; k\cdot d + i,\quad (i = 1,\ 2,\ \ldots,\ d)$$

当然，当d不再是2的幂时，将不再能够借助移位运算来加速秩的换算。不过反过来，在计算效率并不主要依赖于秩换算效率的场合（比如数据规模大到跨越存储层次，涉及一定量的I/O操作时），这种推广完全行之有效。

按照以上实现方式，对于规模为n的d叉堆而言，高度应为：

$$\lceil \log_d(n\cdot(d - 1) + 1)\rceil - 1 \;=\; \mathcal{O}(\log_d n)$$

如此，在上滤过程中的每一步，只需将当前节点与其父节点做一次比较，因此整个上滤操作总体的耗时量不过：

$$\mathcal{O}(\log_d n)$$

与此同时，在下滤过程中的每一步，却需要遍历当前节点及其d个孩子，方可确定是否继续下降一层，以及向那个分支下降一层。相应地，总体耗时量为：

$$\mathcal{O}(d)\cdot\mathcal{O}(\log_d n) \;=\; \mathcal{O}(d\cdot\log_d n)$$

多叉堆中上滤操作与下滤操作之间的如上差异，既细微亦关键。请特别留意体会。

b） 试证明，若取 $d = e/n + 2$，则基于 d 叉堆实现的 Prim 算法的时间复杂度可降至 $\mathcal{O}(e\cdot\log_d n)$；

【解答】

根据以上分析，使用基于d叉堆的Prim算法，总体时间复杂度应为：

$$n\cdot d\cdot\log_d n + e\cdot\log_d n \;=\; (n\cdot d + e)\cdot\log_d n$$

特别地，当取：

$$d \;=\; e/n + 2$$

时，总体的渐进性能将达到渐进最优的：

$$\mathcal{O}(e\cdot\log_d n) \;=\; \mathcal{O}(e\cdot\log_{(e/n\,+\,2)} n)$$

c） 这种改进策略是否也适用于 Dijkstra 算法？

【解答】

实际上，在基于优先级搜索（PFS）策略的图算法中，只要各节点优先级的更新方向总是单调非降（相应地，优先级数总是单调非升），则堆结构的上滤、下滤操作必然各自累计需要执行$\mathcal{O}(e)$次、$\mathcal{O}(n)$次。

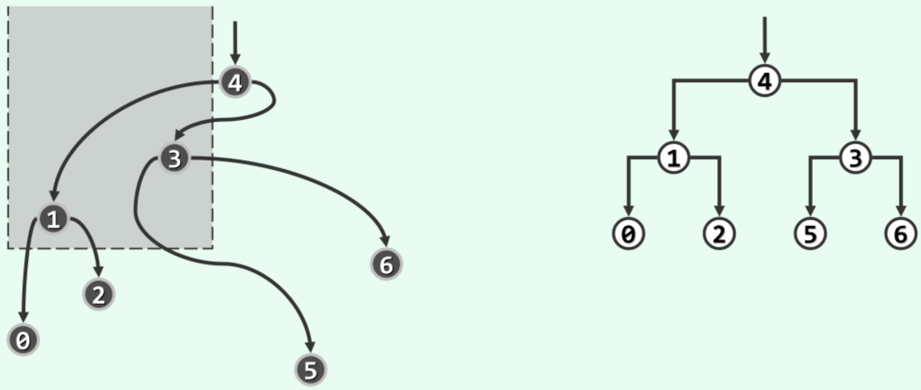
在通常情况下，前者相对于后者都要更多（甚至非常多），故以上技巧及其性能分析的过程和结论，亦将完全适用。

在Dijkstra算法中，各节点优先级的更新方向也是单调非降的，故亦适用以上策略。

[10-18] 所谓半无穷范围查询（semi-infinite range query），是教材 8.4 节中所介绍一般性范围查询的特例。具体地，这里的查询区域是某一侧无界的广义矩形区域，比如 $R = [-1, +1] \times [0, +\infty)$，

即是对称地包含正半 y 坐标轴、宽度为 2 的一个广义矩形区域。当然，对查询的语义功能要求依然不变——从某一相对固定的点集中，找出落在任意指定区域 R 内部的所有点。

范围树（176 页习题[8-20]）稍作调整之后，固然也可支持半无穷范围查询，但若能针对这一特定问题所固有的性质，改用优先级搜索树（priority search tree, PST）[①]之类的数据结构，则不仅可以保持 $O(r + \log n)$的最优时间效率，而且更重要的是，可以将空间复杂度从范围树的 $O(n\log n)$优化至 $O(n)$。



图x10.3 优先级搜索树

如图 x10.3 所示，优先级搜索树除了首先在拓扑上应是一棵二叉树，还需同时遵守以下三条规则。

> ❶ 首先，各节点的 y 坐标均不小于其左、右孩子（如果存在）
> ——因此，整体上可以视作为以 y 坐标为优先级的二叉堆
> ❷ 此外，相对于任一父节点，左子树中节点的 x 坐标均不得大于右子树中的节点
> ❸ 最后，互为兄弟的每一对左、右子树，在规模上相差不得超过一[②]

a） 试按照以上描述，用 C/C++定义并实现优先级搜索树结构；

【解答】

请读者根据以上介绍及提示，独立完成编码和调试任务。

b） 试设计一个算法，在 $O(n\log n)$时间内将平面上的 n 个点组织为一棵优先级搜索树；

【解答】

首先，不妨按照x坐标对所有的点排序。然后，根据如上定义，可以递归地将这些点组织为一棵优先级搜索树。

具体地，为构造任一点集对应的子树，只需花费$O(n)$时间从中找出最高（y坐标最大）者，并将其作为子树树根。以下，借助x坐标的排序序列，可以在$O(1)$时间内将剩余的$n - 1$个点均衡地划分为在空间上分列于左、右的两个子集——二者各自对应的子树，可以通过递归构造。

如此，构造全树所需的时间不超过：

① 由E. M. McCreight于1985年发明[59]

② 若无需遵守最后一条规则，则可保证所有节点能够以x坐标为序组成一棵（未必平衡的）二叉搜索树
此时，该结构兼具二叉搜索树和堆的操作特性，故亦称作树堆（treap）。treap一词，源自tree和heap的组合

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

c) **试设计一个算法，利用已创建的优先级搜索树，在$\mathcal{O}(r + \log n)$时间内完成每次半无穷范围查询，其中r为实际命中并被报告的点数。**

【解答】

查询算法的过程，可大致地递归描述如算法x10.1所示。

```
queryPST( PSTNode v, SemiInfRange R ) { //R = [x1, x2] × [y, +∞)
   if ( ! v || R.y < v.y ) return; //y-pruning
   if ( R.x1 < v.x && v.x < R.x2 ) output(v); //hit
   if ( R.x1 <  v.xm ) queryPST( v.lc, R ); //recursion & x-pruning
   if ( v.xm <= R.x2 ) queryPST( v.rc, R ); //recursion & x-pruning
}
```

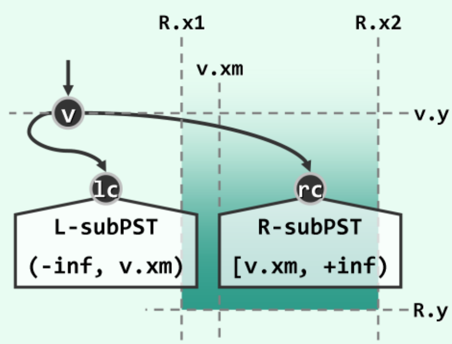
算法x10.1 基于优先级搜索树的半无穷范围查询算法

首先，根据y坐标，判断当前子树根节点v（及其后代）是否已经落在查询范围R之外。若是，则可立即在此处返回，不再深入递归——亦即纵向剪枝；否则，才需要继续深入查找。

以下，再检查根节点v的x坐标，若落在查询范围之内，则需报告该节点。

最后，若在节点v处的横向切分位置为xm，则通过将其与R的左（x1）、右（x2）边界相比较，即可确认是否有必要继续沿对应的子树分支，继续递归搜索——亦即横向剪枝。

具体地如图x10.4所示，唯有当R.x1位于v.xm左侧时，才有必要对左子树v.lc做递归搜索；唯有当R.x2不位于v.xm左侧时，才有必要对右子树v.rc做递归搜索。



图x10.4 基于优先级搜索树的半无穷范围查询算法

对于任意的查询区域$R = [x_1, x_2] \times [y, +\infty)$，考查被算法queryPST()访问的任一节点，设与之对应的点为v = (a, b)。于是，v无非三种类型：

A) 被访问，且被报告出来

——也就是说，v落在R之内（$x_1 \le a \le x_2$且$y \le b$）。此类节点恰有r个。

B) 虽被访问，却未予报告

——因其x坐标落在R之外（$a < x_1$或$x_2 < a$）而横向剪枝，不再深入递归。此类节点在每一层上至多只有两个，总数不超过$\mathcal{O}(2 \cdot \log n)$。

C) 虽被访问，却未予报告

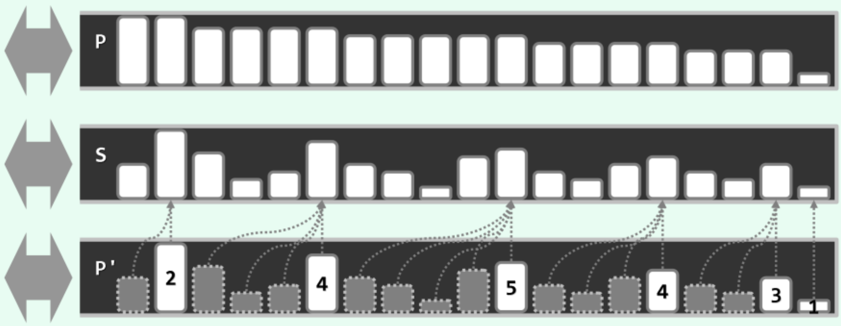
——尽管其x坐标落在R之内（$x_1 \le a \le x_2$），但因其y坐标却未落在R之内（$b < y$）而纵向剪枝，也不深入递归。实际上，此类节点的父节点，必然属于A或B类，其总数不超过这两类节点总数的两倍。

综合以上分析可知，以上queryPST()算法的渐进时间复杂度不超过$O(r + \log n)$。

[10-19] 试为第 4 章栈结构增加 Stack::getMax()接口，以在 $O(1)$时间内定位并读取栈中的最大元素。要求 Stack::push()和 Stack::pop()等接口的复杂度依然保持为 $O(1)$。

【解答】

如图x6.5所示，对于任何一个栈S，可以引入另一个与之孪生的“镜像”栈P。



图x10.5 高效支持getMax()接口的栈

具体地，P中的元素与S中的元素始终保持一一对应，前者的取值，恰好就是后者所有前驱中的最大者。当然，P中元素因此也必然按照单调非降的次序排列。如此，任何时刻栈P的顶元素，都是栈S中的最大元素。

为保持二者如上的对应关系，它们的push()和pop()操作必须同步进行。若执行：

```
S.pop();
```

则只需同步地执行：

```
P.pop();
```

而若执行：

```
S.push( e );
```

则需要同步地执行：

```
P.push( max( e, P.top() ) );
```

以上方案还可以进一步地优化。

仍如图x6.5所示，可将栈P“压缩”为栈P'。为此，需要注意到，P中相等的元素必然彼此相邻，并因此可以分为若干组。若假想式地令栈P中的每个元素通过指针指向栈S中对应的元素，而不是保留后者的副本，则可以将P中的同组元素合并起来，共享一个指针。当然，同时还需为合并后的元素增设一个计数器，记录原先同组元素的总数。

如此改进之后的“镜像”结构，如图中的栈P'所示：每一组元素只需保留一份（白色），其余元素（灰色）则不必继续保存。这样，附加空间的使用量可以大为降低。

相应地，在栈S每次执行出栈操作时，栈P（P'）必须同步地执行：

```
if ( ! ( -- P.top().counter ) ) P.pop();
```

而在栈S每次执行入栈操作时，栈P（P'）也必须同步地执行：

```
P.top() < e ? P.push( e ), P.top().counter = 1 : P.top().counter ++;
```

可见，S的push()和pop()接口，依然保持$\mathcal{O}(1)$的时间效率。

请读者根据以上介绍和提示，独立完成编码和调试任务。

[10-20] 试为第4章的队列结构增加Queue::getMax()接口，在$\mathcal{O}(1)$时间内定位并读取其中最大元素。要求Queue::dequeue()接口的时间复杂度依然保持为$\mathcal{O}(1)$，Queue::enqueue()接口的时间复杂度不超过分摊的$\mathcal{O}(1)$[③]。（提示：借助101页习题[4-22]中的双端队列结构Deque）

【解答】

习题[10-19]针对栈结构的技巧，可以推广至队列结构。比如，可以引入一个双端队列P并依然约定，其中每个元素也是始终指向队列Q中所有其前驱中的最大者。

为保持二者的对应关系，它们的dequeue()和enqueue()操作也必须同步进行。若执行：

```
Q.dequeue();
```

则只需同步地执行：

```
P.removeFront();
```

而若执行：

```
Q.enqueue(e);
```

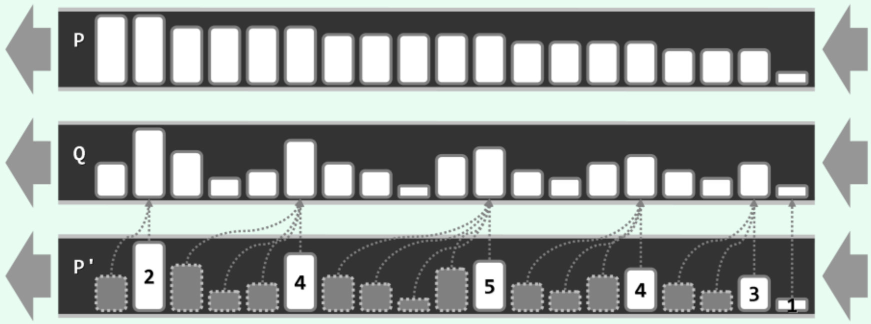
则只需同步地执行：

```
P.insertRear(e);
for (x = P.rear(); x & (x.key <= e); x = x.pred) //for each rear element x no greater than e
   x.key = e; //update its maximum record
```

也就是说，除了首先也令e加入队列P，而且还需要将P尾部所有不大于e的元素，统一更新为e。很遗憾，在最坏情况下这需要$\Omega(n)$时间。而且更糟糕的是，这种情况可能持续发生（读者不妨独立构造出这样的一个实例）。

造成这一困难的原因在于，队列中任一元素的前驱集，不再如在栈中那样是固定的，而是可能增加，甚至新增的元素非常大。为此，可按照如图x10.6所示的思路进一步改进。

③ 经如此拓展之后，这一结构同时兼具队列和堆的操作特性，故亦称作队堆（queap）
queap一词，源自queue和heap的组合



图x10.6 高效支持getMax()接口的队列

具体地，可首先仿照习题[10-19]的改进技巧，通过合并相邻的同组元素，将队列P压缩为队列P'。然后，在队列Q每次执行出队操作时，队列P（P'）必须同步地执行：

```
if (!(-- P.front().counter)) P.removeFront();
```

而在栈S每次执行入栈操作时，栈P（P'）也必须同步地执行：

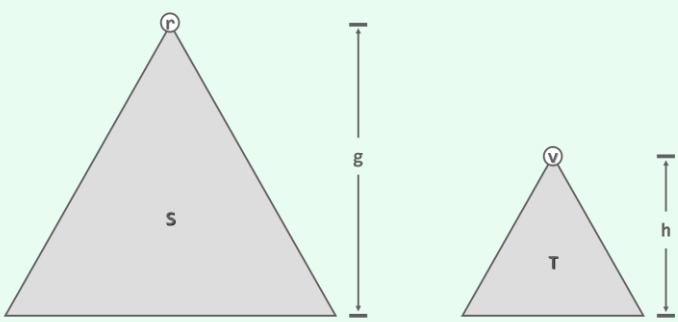
```
a = 1; //counter accumulator
while (!P.empty() && (P.rear().key <= e) //while the rear element is no greater than e
   a += P.removeRear().counter; //accumulate its counter before removing it
P.insertRear(e); P.rear().counter = a;
```

这里的while循环，在最坏情况下仍然需要迭代$O(n)$步，但因为参与迭代的元素必然随即被删除，故就分摊意义而言仅为$O(1)$步，时间性能大为改善。

另外，这里的队列P'并不需要具备双端队列的所有功能。实际上，它仅使用了Deque结构的removeFront()、insertRear()和removeRear()接口，而无需使用insertFront()接口——因此形象地说，它只不过是一个“1.5”端队列。

[10-21]　任给高度分别为 g 和 h 的两棵 AVL 树 S 和 T，且 S 中的节点均不大于 T 中的节点。试设计一个算法，在 $O(\max(g, h))$时间内将它们合并为一棵 AVL 树。

【解答】

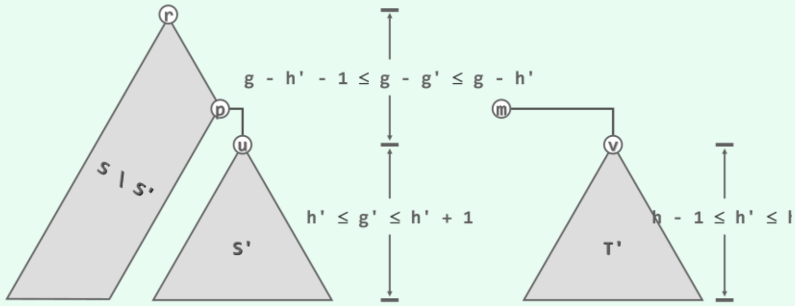


图x10.7 合并AVL树S和T：不妨假定g ≥ h

首先如图x10.7所示，不失一般性地，假定S的高度不低于T——否则，以下算法完全对称。

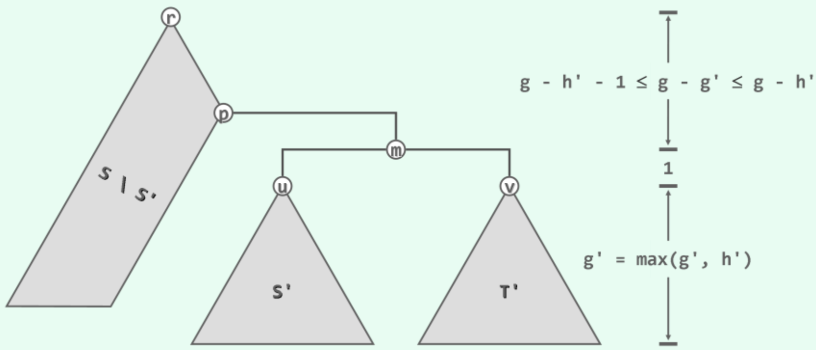
设m为T中的最小节点。若将m从T中摘出，则如图x10.8所示，新得到的AVL树T'的高度h'必然不致增加，而且至多降低一层。

根据AVL树的定义，这里沿着起自根节点的任一通路每下降一层，节点的高度虽必然降低，但至多降低2。因此如图x10.8所示，沿着树S的最右侧通路，必然可以找到某个节点u，其高度不低于h'，而且至多比T'高一层。将子树u记作S'。



图x10.8 合并AVL树S和T：删除T中的最小节点m，在S的最右侧通路上找到与树T'高度接近的节点u

于是接下来如图x10.9所示，只要以节点m为联接点，将S'和T'分别作为其左、右子树，即可拼接成为一棵AVL树。



图x10.9 合并AVL树S和T：以m为结合点合并S'和T'，在整体接入至S

以下，将该AVL树作为子树，并在节点u原先的位置（作为节点p的右子树）接入至树S中。请注意，至此，全树中仅有节点p可能失衡。而且若此时节点p的确失衡，则其平衡因子必然为-2。也就是说，其效果完全等同于将m插入其中之后所造成的失衡。因此，只需从p出发逐层上溯，即可通过不超过$\mathcal{O}(g - g') = \mathcal{O}(g - h)$次的旋转使全树恢复平衡。

再计入删除节点m所需的$\mathcal{O}(h)$时间，以及查找节点u所需的$\mathcal{O}(g - h)$时间，可见以上算法的总体时间复杂度为$\mathcal{O}(g) = \mathcal{O}(\max(g, h))$。

特别地，若节点m已经从T中摘出，且节点u已知，则以上算法只需$\mathcal{O}(g - h)$时间，线性正比于两棵待合并树的高度之差。对于以下的习题[10-22]，这一性质将至关重要。

[10-22] 任给高度为 h 的一棵 AVL 树 A，以及一个关键码 e。

试设计一个算法，在 $\mathcal{O}(h)$时间内将 A 分裂为一对 AVL 树 S 和 T，且 S 中的节点均小于 e，而 T 中的节点均不小于 e。（提示：借助 AVL 树的合并算法）

【解答】

以关键码e查找路径上的各节点为界，可以按照中序遍历次序将全树A划分为一系列的子树。

以如图x10.10所示的树A为例，若e的查找路径为：

t_1, s_1, s_2, t_2, t_3, t_4, s_3, s_4, t_5, ...

则相应地划分出来的子树依次是：

T_1, S_1, S_2, T_2, T_3, T_4, S_3, S_4, T_5, ...

不难看出，全树的中序遍历序列应是：

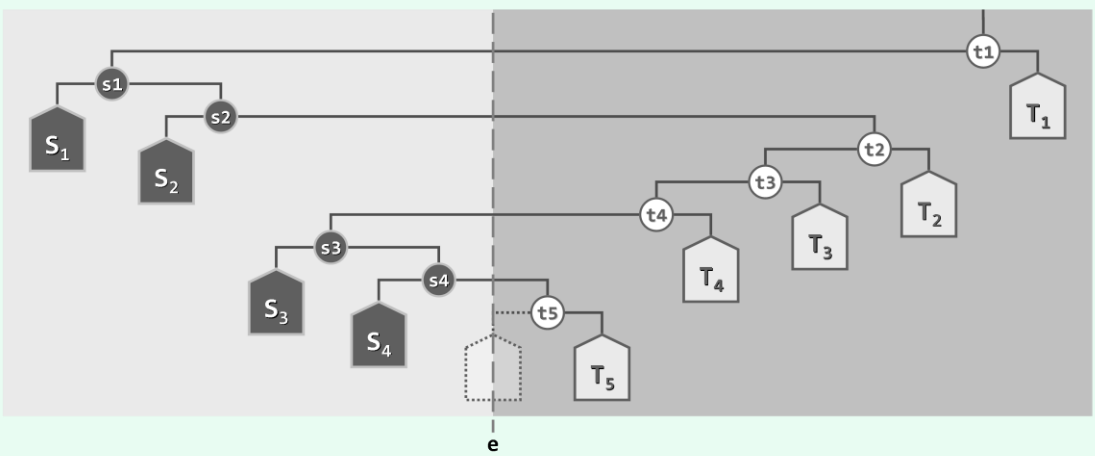
$\boxed{S_1, s_1}$; $\boxed{S_2, s_2}$; $\boxed{S_3, s_3}$; $\boxed{S_4, s_4}$; ...; $\boxed{t_5, T_5}$; $\boxed{t_4, T_4}$; $\boxed{t_3, T_3}$; $\boxed{t_2, T_2}$; $\boxed{t_1, T_1}$

而分裂出的两棵AVL子树的中序遍历序列，则应该分别是：

S = $\boxed{S_1, s_1}$; $\boxed{S_2, s_2}$; $\boxed{S_3, s_3}$; $\boxed{S_4, s_4}$; ...

T = ...; $\boxed{t_5, T_5}$; $\boxed{t_4, T_4}$; $\boxed{t_3, T_3}$; $\boxed{t_2, T_2}$; $\boxed{t_1, T_1}$

因此，我们可以自底而上，通过反复的合并构造出S和T。



图x10.10 以任意关键为界，分裂AVL树（这里只是示意性地绘出了各子树，并未严格地反映其高度）

鉴于这两棵树完全对称，这里不妨仅以树T为例，介绍具体的合并过程。

一旦以t_5为根的AVL子树已经构造出来，我们即可以节点t_4为联接点，将其与AVL子树T_4合并，得到一棵更大的AVL树；接下来，再以节点t_3为联接点，进一步地将其与AVL子树T_3合并；然后，再以节点t_2为联接点，继续将新得到的AVL树与树T_2合并；最后，以节点t_1为联接点，将新得到的AVL树与树T_1合并。如此，最终即可得到所求的AVL树T。

这里涉及的AVL树合并计算，属于习题[10-21]所指出的特殊情况：作为联接点的节点t_k，均等效于已经从待合并子树中摘出，且接入位置已知。因此每次合并所需的时间，不超过被合并子树的高度之差。考虑到前后项的依次抵消效果，累计时间应渐进地不超过原树高度$\mathcal{O}(h)$。