线段跳表——跳表[1]的一个拓展

石家庄二中 李骥扬

摘要

本文主要介绍我对跳表(跳跃表,Skip Lists, SL,1987 年由 William Pugh 发明)的一个拓展：线段跳表(Segment Skip Lists,SSL)以及相关的效率证明。并简要分析跳表及其拓展在信息学竞赛中的优势。

关键词

跳表 平衡树 线段跳表 线段树

引言

在信息学竞赛中，数据结构是重要的一部分。通常考察我们维护、处理数据的能力。这类题目常常要求我们维护一个线性有序的结构，并对其进行插入查询等操作，常用的实现方式有平衡树、块状链表、树状数组等。

跳表是一种新型数据结构，支持有序表的插入、删除、查找等字典操作。跳表基于随机化和概率，插入、删除、查找操作的期望时间复杂度均为 $O(\log N)$。同时由于其线性表的本质，具有编写简单，易于拓展的性质。它能代替平衡树实现字典操作。且由于其结构简单，不需要旋转操作，跳表支持将各部分存放在不同介质上，进行并行操作[2]。

但是基本的跳表只支持插入、删除、查找的字典操作，并不支持区间操作。跳表的能力绝不限于此，只要更深入地分析一下跳表的结构，我们就能从跳表中得到一棵二叉树的结构。进而添加区间信息，得到线段树。本文将通过对插入、删除操作的升级，使得这些操作能够维护区间信息。

拓展后的跳表，称为线段跳表，能够完全替代各类平衡树，支持字典操作和区间操作。同时保持了其原有特点，简单易用，还支持顺序操作等特殊操作。线段跳表将成为信息学竞赛中的理想数据结构。

---

[1] 本文讨论跳表的范围仅限于基于期望的随机跳表(Skip Lists，SL)，不涉及确定性跳表（1-2-3 Deterministic Skip List，1-2-3 DSL)的内容。

[2] 相关内容参见 William Pugh 的《CONCURRENT MAINTENANCE OF SKIP LISTS》

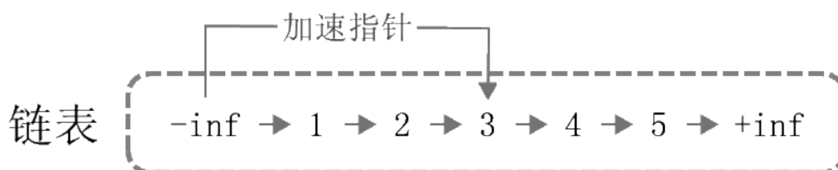
第一部分 跳表简介

❖ 从链表到跳表

链表作为一种常用而灵活的数据结构，常常用于维护线性数据。链表可以方便的执行插入删除操作，但是从链表中查找一个数的代价可能很大(时间复杂度 $O(N)$)。相反，维护一个有序数组可以使用二分方便地在 $O(\log N)$ 的时间内进行查找，但若要维护数组的有序性，插入和删除的复杂度又会很大(时间复杂度 $O(N)$)。如果能维护一个有序链表且支持二分，我们就可以在 $O(\log N)$ 的时间内完成一组规模为 n 的数据的插入、查找、删除操作(字典操作)。

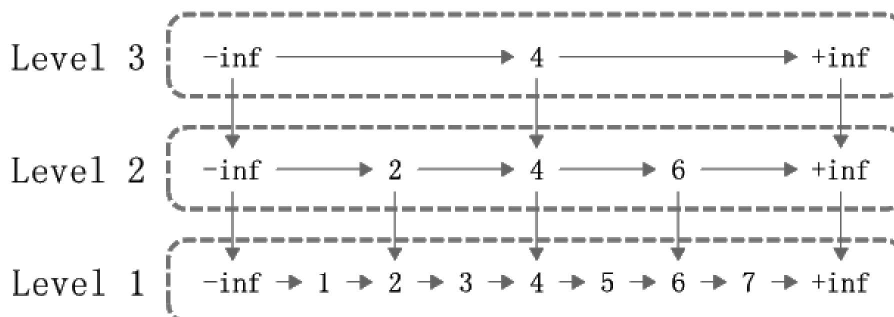
让我们首先考虑一个静态链表的查询。

一个简单的想法就是在链表中给一些节点增加一个指针（加速指针），使得我们能够在适当的时候跳过一些结点，从而达到加速查找的目的。（如下图）



[figure 1.1] 带加速指针的链表

然而如果我们限制对于一个结点只连出一个加速指针，使用类似块状链表的结构，我们能达到的最优的复杂度为 $O(\sqrt{n})$。能不能得到更好的结果呢？我们放宽每个节点有一个加速指针的限制，基于与静态数组的二分搜索的类比，我们能够得到如下结构：



[figure 1.2]空想跳表

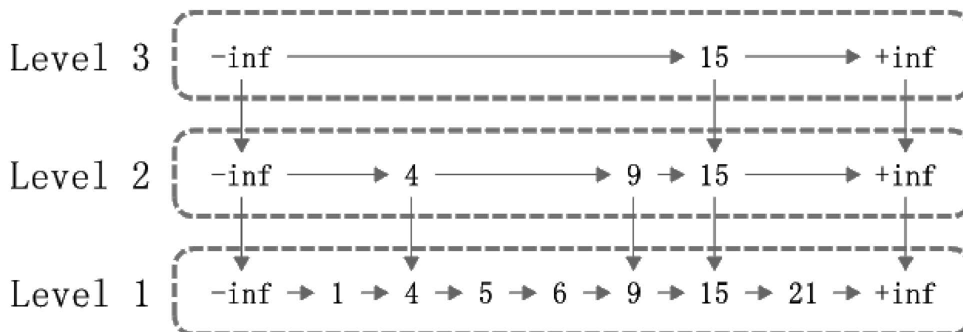
每个节点连出的加速指针指向该节点在二分过程中可能作为左端点的区间中点。这一结构可以在 $O(\log N)$ 的时间内实现对数据的搜索，搜索过程即二分，每次通过加速指针找到当前区间的中间节点，比较后确定更精确的区间继续二分。

但是当我们面对动态数据的时候，这一结构在插入删除过程中显然无法以较小代价进行维护。

随机化经常被用来降低算法的时间复杂度。在以上思路中增加随机化思想，我们便得到了跳表：

跳表由 Lev 层链表以及下行指针构成，插入数据 v 时随机化得出高度 h，然后向第 1 层至第 h 层插入索引值为 v 的节点，对于第 2 层至第 h 层的节点插入下行指针，指向第一层具有相同索引值的节点。这 h 个节点构成一个高度为 h 的块 (Block)。

如下图所示：



[figure 1.3]跳表

具有这样结构的跳表查找的期望时间复杂度为 $O(\log N)$，参见第三部分。

❖ **形式化地给出跳表结构**

跳表由多条链表（L_1，L_2…… $L_{\max lev}$）以及下行指针构成，满足以下条件

1> L_i 为一条链表，自左至右单调递增，包含两个特殊元素 $-\infty$ (起始节点) 和 $+\infty$ (终止节点)

2> L_1 包含所有元素，$L_i (i \geq 2)$ 包含部分元素，且对 $\forall x \in L_i$ 有 $x \in L_j \mid \forall 1 \leq j < i \leq \max lev$，即 $L_{\max lev} \subseteq L_{\max lev-1} \ldots\ldots L_2 \subseteq L_1$

3>对于 $L_i(i \ge 2)$ 中的任意节点，有一个下行指针，指向 L_{i-1} 中具有相同索引值的节点。

同时给出一些下文使用到的术语的解释：

横向边：即各层链表中的指针。

纵向边：即下行指针。

块：只考虑纵向边时的一个连通分支，其包含的节点数成为其高度。

前驱节点、后继节点：即某一节点在所在跳表中的前驱节点和后继节点。

底层节点：位于第 1 层的节点。

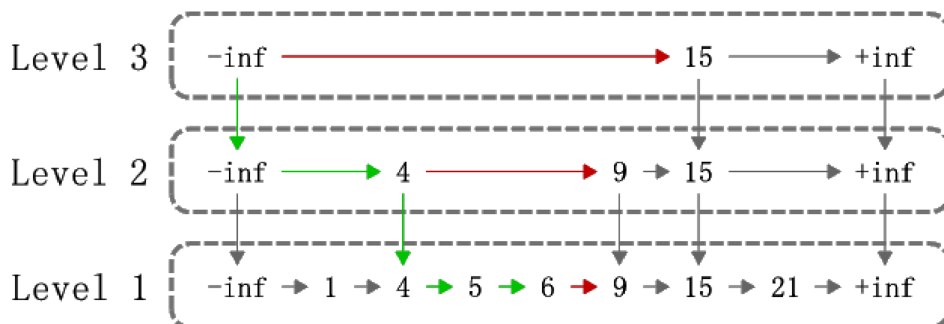
搜索路径(Path(v))：在跳表中搜索索引值 v 时所经历的节点序列。

逆搜索路径(Rev-Path(v))：将 Path(v)逆序得到的序列称为 Rev-Path(v)。

❖ **跳表的字典操作：查找、插入和删除**

查找

查找过程即构造 Path(v)的过程。下图所示为一查找过程：



[figure 1.4]绿色为查找路径，红色为尝试过但没有走的边

指针 p 从顶层(L_h)的起始节点($-\infty$)开始查找。每次若当前节点的后继节点检索值小于等于要寻找的值，则将 p 右移到原节点的后继节点，否则沿纵向边下移至下一层。直到无法移动(位于底层节点，且后继节点的检索值大于所要查找的值)，p 指向的节点的值即为数据中不大于搜索值的最大值。

插入

首先随机化待插入节点的高度。生成的方式如下：

每次插入节点，至少要插入到 L_0，所以高度至少为 0，在循环中，每次有 p 的概率使得节点高度加 1，1-p 的概率退出循环，得到所求随机高度。为了编程

方便，可以设置一个最高高度 max lev，固定维护 max lev 个链表。得到的高度若大于 max lev，则人为设为 max lev。

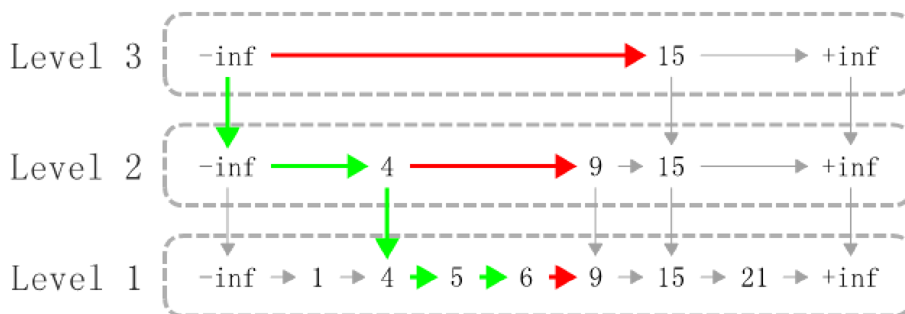
概率 p 的取值一般选为 $\frac{1}{4}$，具体分析留至第三部分。

给出伪代码：

```
Function random_height;
    p:= 1/4;
    h:=0;
    while random( )<=p do h:=h+1;
    //here random( ) return a value in [0,1)
    if h>maxlev then h:=maxlev;
    return h;
```

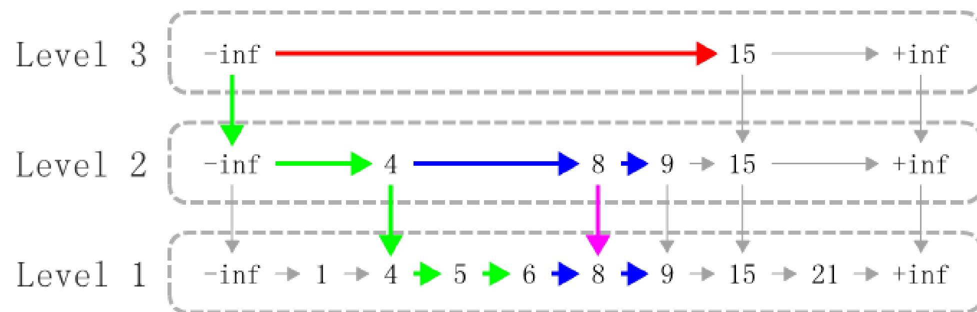
然后向跳表中 1..random_height 层插入节点。

如下图所示：



[figure 1.5.1]插入前

向 1..random_height 层中的红色边上插入节点，并引入纵向边：



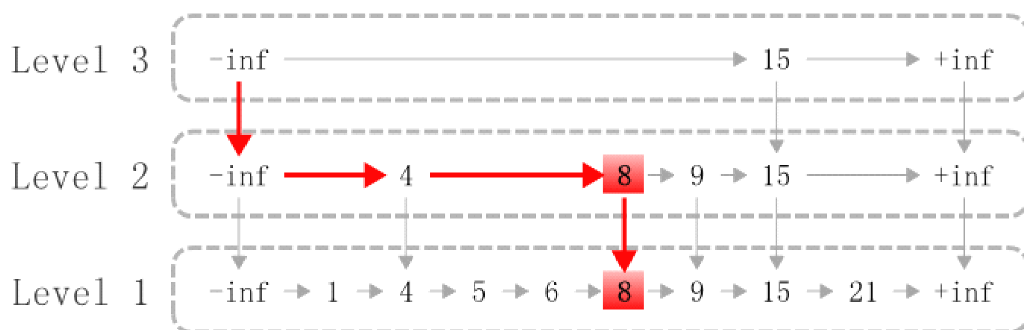
[figure 1.5.2]插入后

实现时可以先搜索待插入值，得到 Path(v)，再在 Path(v)中最后访问的 1..random_height 层的节点之后插入新节点。即向 1..random_heigth 各层中最后访问的节点之后插入新节点。

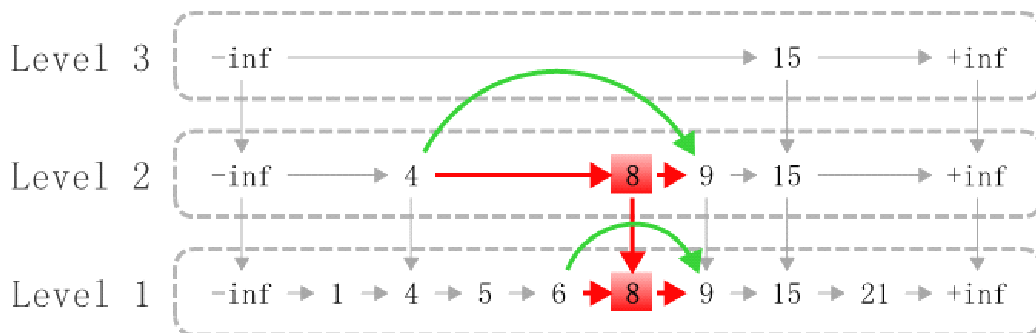
删除

删除与插入类似。先搜索待插入值得到 Path(v)，然后从跳表中删除检索值为 v 的节点。

过程如图所示：



[figure 1.6.1]先搜索要删除的节点



[figure 1.6.2]删除红色节点以及红色边 建立绿色边

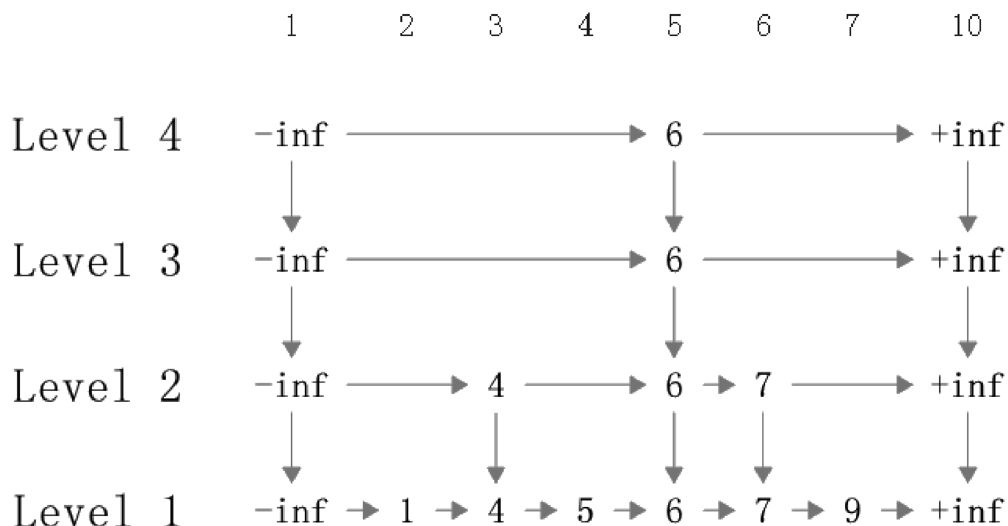
相关代码参见附件中的 SkipList.Pas

第二部分 线段跳表的结构与基本操作

❖ 提取线段树

本节从跳表的结构中提取线段树，进而得到线段跳表，维护区间信息。

对于给定跳表：



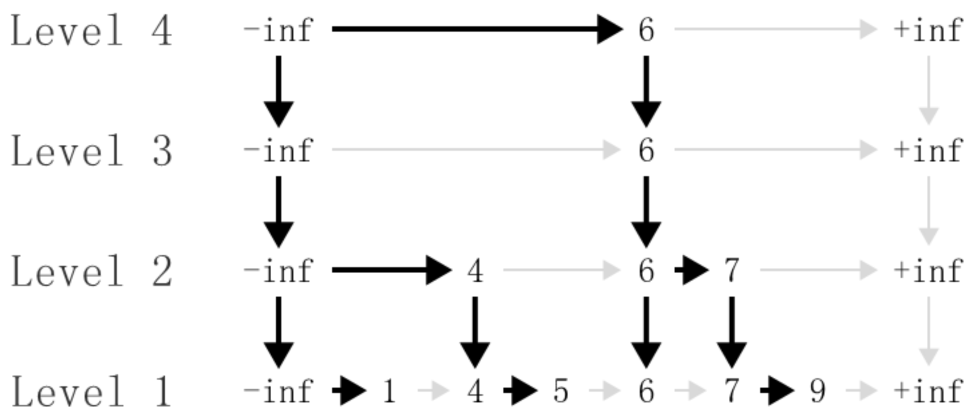
[figure 2.1]Skip List

如果我们定义：

实边：除了连向＋∞的边以外的边中所有纵向边以及连向某块的最高节点的横向边。

虚边：所有连向＋∞的边以及连向某块中最高节点以下节点的横向边。

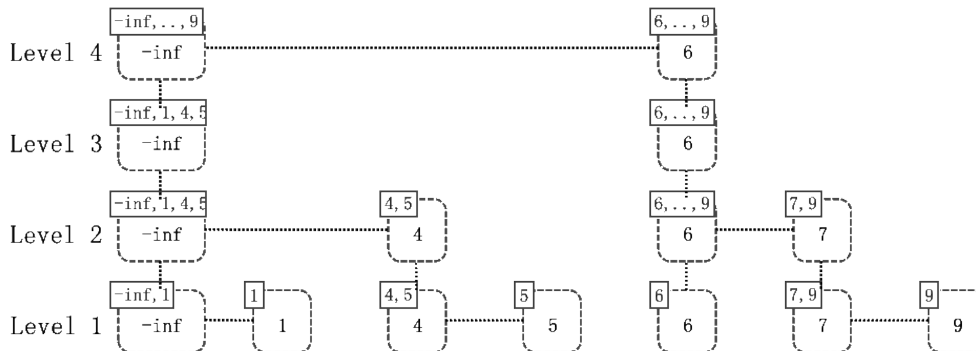
我们在跳表中隐藏所有虚边，可以得到：



[figure 2.2]BST in a Skip List

我们发现，各节点以及实边构成了一棵 BST。

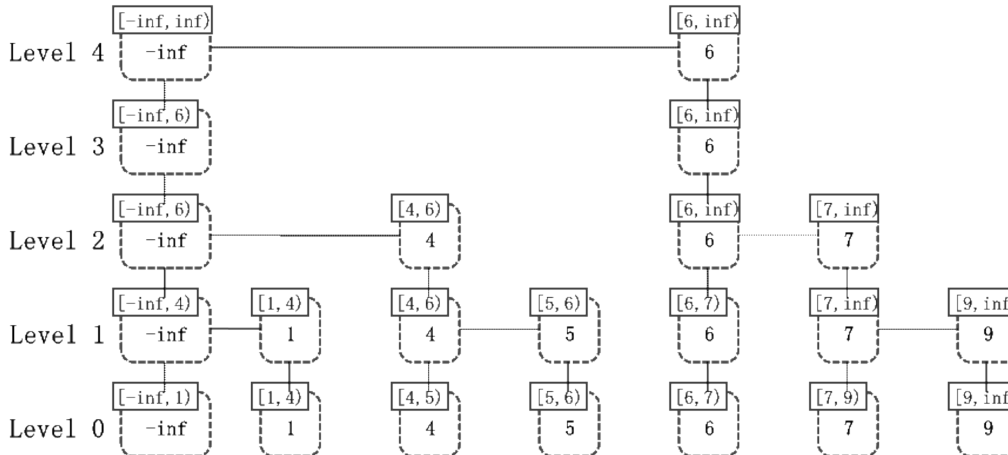
进一步的，对所有 v，在 Path(v)各节点上添加 v 标记，得到如下结构：



[figure 2.3.1]Segment Skip List 前身

我们已经可以从中看到线段树的影子了。

再进一步，将标记的离散点改为连续区间，同时在第一层之下加入第零层，同样包含所有的检索值，并向第一层所有节点添加指向第零层相应节点的下行指针。我们便得到：



[figure 2.3.2]Segment Skip List 最终版本

至此，我们得到了一棵隐藏在跳表中的线段树。我称之为线段跳表(Segment Skip List,SSL)。

❖ 形式化给出线段跳表

线段跳表由多条链表（L_0，L_1，L_2……$L_{\max lev}$）以及下行指针构成，满足以下条件

1> L_i 为一条链表，自左至右单调递增，包含两个特殊元素 $-\infty$ (起始节点)和 $+\infty$ (终止节点)

2> L_0 和 L_1 包含所有元素，$L_i(i \geq 2)$ 包含部分元素，且对 $\forall x \in L_i$ 有 $x \in L_j \mid \forall 0 \leq j < i \leq \max lev$，即 $L_{\max lev} \subseteq L_{\max lev-1} \ldots\ldots L_2 \subseteq L_1 = L_0$

3>对于 $L_i(i \geq 1)$ 中的任意节点，有一个下行指针，指向 L_{i-1} 中具有相同索引值的节点。

4>索引值小于 $+\infty$ 的每个节点对应一个左闭右开的区间，区间的左边界为该节点的索引值。设搜索该节点过程中最后一次经历的纵向边为u->v，则该节点右边界为u的后继节点的索引值。该值不需额外存储，每次搜索过程中即可动态获得。

除了将第零层节点定义为底层节点外，前文所述术语基本不变。

这样的定义得出的一定是一棵线段树，根节点表示区间范围为[-inf,+inf)。很容易证明，证明的过程在此略去。

- **两类区间信息的维护**

接下来便是设法利用跳表中隐含的线段树维护区间信息了。

首先将区间信息分为两类：

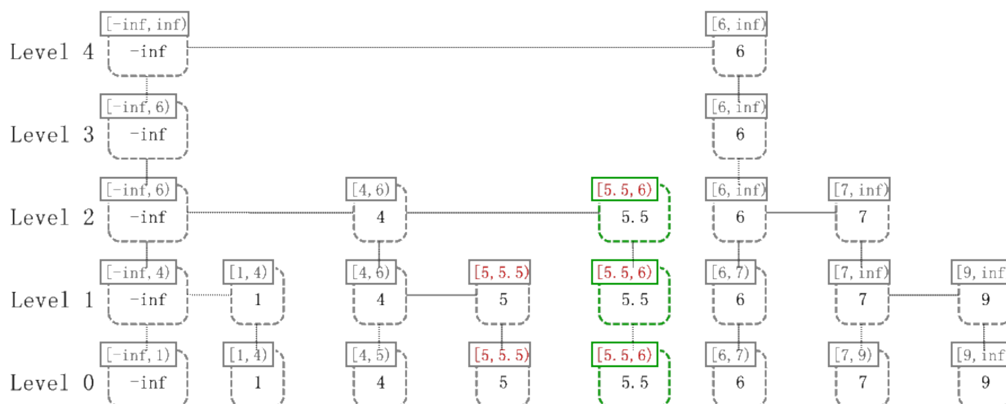
☼DP信息(DPi)：树形DP时存储的某一区间的信息。例如检索值在区间内的点的个数，最大值最小值等。特征是每一个节点都保存此信息。信息可以每个节点的两个孩子的信息计算得出。通常此计算过程时间复杂度为 $O(1)$。

☼ 线段信息(SEGi)：线段树中存储的信息。例如对于某一区间的染色情况。特征是并非所有节点都保存此信息，对区间操作时，若操作区间包含某一非叶子节点对应区间，可以将信息存在此非叶子节点处，降低复杂度。信息可以分配到两个孩子节点，例如某段区间颜色可以分配到两个孩子节点。

这两类信息具有不同的特点，操作时的方法也不同。接下来介绍线段跳表在插入、删除时应如何维护这两类信息。

插入

虽然插入节点之后，可能改变这棵线段树的结构，但是如下图所示，对应区间改变的节点数，只有 $O(\log N)$ 的规模：



[figure 2.4.1]插入过程中 SSL 节点表示区间改变情况

上图中，绿色框的节点是新添加的。造成了红色字标示出的区间发生了变化。可以看出，除了 Path(v)和 Path(v-ε)的部分，节点对应的区间不会改变。

因此，我们需要更新信息的节点数也只有 $O(\log N)$的规模。

给出插入 v 时操作顺序：

1. 沿 Path(v)下分 SEGi 信息，并通过此过程，得出 v 节点的 SEGi 信息。
2. 插入检索值为 v 的块，并且设置底层节点的 SEGi 值为之前求出的 SEGi 值。
3. 沿 Rev-Path(v)顺序更新路径上各节点的 DPi 信息。
4. 沿 Rev-Path(v-ε)顺序更新路径上个点的 DPi 信息。

只需证明以下两个结论，即可保证这样做的正确性：

1. Path(v)包含的节点，表示的区间，均包含 v。
2. 插入 v 之后，只有 Path(v)和 Path(v-ε)中的节点表示的区间范围会发生改变。

证明如下：

1. 每次走到节点的检索值一定小于等于搜索目标，即 Path(v)中所有点检索值小于等于 v，每个点表示区间的左端(闭合)一定小于等于 v。对于区间右端点，一定发生在某次比较，发现 v 小于此值后不再向右走而是向下，因而区间右端点(开)一定大于 v。

2. 首先，各个点索引值不变，因而各个区间左端点不变。于是可以只考虑右端点。根据定义，右端点为搜索过程中最后一次纵向边起始节点的后继节点的索引值。对于 Path(v)和 Path(v-ε)之外的节点，其纵向边起始节点连出的横向指针并未发生改变。综上，插入 v 之后，只有 Path(v)和 Path(v-ε)中的节点表示

的区间范围会发生改变。

需要改变区间信息的只有区间范围改变的节点以及对应区间包含 v 的节点，即只有 Path(v)和 Path(v-ε)上的点。

删除

删除与插入类似。大致过程如下：

1.清理被覆盖掉的 SEGi 信息。沿着 Path(v)，将第一次遇到$SEGi \neq NULL$之后所有节点 SEGi 置为空。再沿着 Path(v-ε)重复一遍。

2.若 Path(v)中检索值为 v 的节点的 SEGi 信息不是全为空，则 v 作为某染色区间的左端点，不能删去。再判断 Path(v-ε)中层次编号小于等于检索值为 v 的块的高度的结点 SEGi 信息是否全空，若是，则 v 为某区间右端点，不能删去。（对于能合并的 SEGi 信息应先合并出 Path(v)以及 Path(v-ε)的值。）否则继续进行。

3.删除检索值为 v 的块。

4.沿 Rev-Path(v)更新 DPi 信息。

为了精简篇幅，我将两类信息的维护操作写到了一起。但是实际应用时通常只会用到一种操作。附件中针对两类信息分别提供了一个程序作为参考：SEGiSSL.Pas 和 DPiSSL.Pas

第三部分 跳表的优势和效率分析

❖ 跳表的效率分析

搜索、插入、删除的期望时间复杂度[3]

先设 $\max lev \to +\infty$。

我们先推导搜索的期望时间复杂度，亦即搜索的期望路径长度：

设 $f(N)$ 表示 N 个节点的跳表中搜索的期望路径长度

它分为三个部分

1> 1 至 $\max lev$ 层构成的跳表中搜索的期望路径长度。此部分相当于一个期望规模为 $O(pN)$ 的跳表的期望路径长度。

2> 从第二层下降至第一层的一条指针。(线段跳表中为从第一层下降至第零层的一条指针。)

3> 在第一层中右行的路径长度。(线段跳表中为在第零层中右行的路径长度。)每次能够右行的概率为 1-p，记为 q。

于是

$$
\begin{aligned}
f(N) &= f(pN) + 1 + q + q^2 + q^3 + q^4 + q^5 + \ldots \\
&= f(pN) + \frac{1}{p}
\end{aligned}
$$

解函数

$$f(N) = f(pN) + \frac{1}{p}$$

得到

$$f(N) = \log_a N \quad where \quad a = \left(\frac{1}{p}\right)^p$$

插入及删除的时间复杂度一部分与搜索路径长度有关，另外还有一部分与高度有关，亦即为 $O(\log_a N + \max height) = O(\log_a N)$，其中 $\max height$ 为跳表的高

[3] 本节给出一个简明直观但不慎严谨的推导，严格的证明请见 William Pugh 的《SKIP LISTS:A PROBABILISTIC ALTERNATIV TO BALANCED TREES》

度，当 $\max lev$ 是一个常数时（不趋于无穷时），$\max height = \max lev$。由于 $\max lev$ 的限制，这种情况下的时间复杂度不易分析，但在实践中并不会产生退化（除非将 $\max lev$ 设得很小），它使得编程简单，可以固定跳表的层数为 $\max lev$，推荐使用。$\max lev$ 的选取参见下文。

期望空间复杂度

同上 设 $\max lev \to +\infty$

1. 单一节点高度期望

$$\begin{aligned} h_x &= 1 + p + p^2 + p^3 + p^4 + \ldots \\ &= \frac{1}{1-p} \end{aligned}$$

2. 期望空间复杂度

$$\text{节点数} = N \times h_x = \frac{N}{1-p}$$

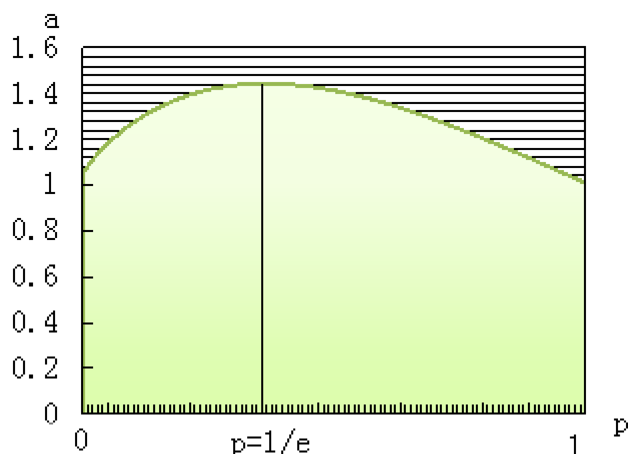
3. 最高高度分析

$$P(h_{\max} \le m) = \left(1 - p^{m-1}\right)^N$$

4. 当 $\max lev$ 是一个常数时（不趋于无穷时），节点高度期望小于等于以上分析的值，空间复杂度小于等于以上分析得到的值。

概率 p

$a = \left(\dfrac{1}{p}\right)^p$ 的图像如下：



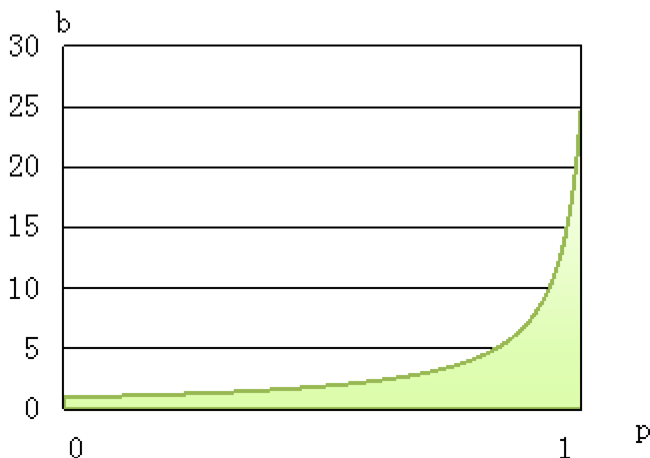
$$f(N)=\log_a N=\frac{\log_2 N}{\log_2\left(\frac{1}{p}\right)^p}$$

$p=\frac{1}{e}$ 时系数最小，$f(N)=\frac{\log_2 N}{\log_2(e)^{\frac{1}{e}}}\approx 1.88416939\log_2 N$

$p=\frac{1}{2}$ 和 $p=\frac{1}{4}$ 时系数 $f(N)=\frac{\log_2 N}{\log_2(2)^{\frac{1}{2}}}=\frac{\log_2 N}{\log_2(4)^{\frac{1}{4}}}=2\log_2 N$

$p=\frac{1}{e}$ 时比 $p=\frac{1}{2}$ 和 $p=\frac{1}{4}$ 时间短一些，但是不易实现，所以通常使用 $p=\frac{1}{2}$ 和 $p=\frac{1}{4}$，由于空间的关系，$p=\frac{1}{4}$ 更常用。

空间复杂度系数 $b=\frac{1}{1-p}$ 图像如下图：



可知 $p=\frac{1}{4}$ 时占用的空间比 $p=\frac{1}{2}$ 小（约为 $p=\frac{1}{2}$ 时的 $\frac{2}{3}$）。

所以最常用的 p 值为 $\frac{1}{4}$。

层数上限 maxlev

当没有最高层数限制的时候（$\max lev\to+\infty$）才是正常的 SL，但是实际应用中为了程序实现简单，往往设置这样一个常数，引入 maxlev 后的时间复杂度不易分析，但实践效果不错。参见对于最高高度分析的内容，可以选取一个适当的 $\max lev$，比如对于 $p=\frac{1}{4}$　$N=10^6$ 时取 $\max lev=16$，则 $P(h_{\max}\le\max lev)=\left(1-p^{\max lev-1}\right)^N\approx 0.999069111$ 就是一个不错的选择。

❖ 线段跳表（SSL）的效率分析

线段跳表的操作与跳表基本一致，效率同样取决于搜索路径的期望长度，时间复杂度同样为$O(\log N)$，只是常数较大。

至于空间复杂度，由于其比普通的跳表多一层，空间复杂度粗略可记为$O\left(\frac{2-p}{1-p}N\right)$看似较大，实际上当$p=\frac{1}{4}$时，不过是$O\left(\frac{7}{3}N\right)$仅比线段树大一点。由于每个节点需要记录区间信息，节点本身的大小也会增大。但空间复杂度仍为$O(N)$。

❖ 跳表的优势

Ｌ跳表本质为链表，操作简单，不需要旋转操作。跳表比一般的平衡树短很多。更主要的是它的操作容易思考，不需要特殊的技巧写法以缩短代码。

Ｌ支持并行操作。跳表不需要旋转，结构不会产生巨大改变，可以存储在不同介质上，支持并行操作。参见 William Pugh 的《CONCURRENT MAINTENANCE OF SKIP LISTS》。

Ｌ只需改变 p 值，即可得到调整时空效率。

Ｌ原始的跳表（仅支持字典操作），不需要存储任何用以维护平衡的附加信息，即可得到较优的时间复杂度。没有空间浪费。

Ｌ可以进行顺序操作(按检索值对各节点处理)等特殊操作。

❖ 线段跳表与线段树、平衡树的对比

线段树、平衡树是信息学竞赛中常用的数据结构，很多题目都可以使用它们。本文介绍的线段跳表，通过随机化得到一个相对平衡的结构，且能够支持线段树与平衡树的所有功能。相比线段树，具有能够动态处理信息，支持在线询问的特点；相比平衡树，具有编写简便，代码精简的特点。

综上，跳表及其拓展——线段跳表，是优秀的数据结构。可以成为参加信息学竞赛的各位选手的一把利器。

参考资料

1.《SKIP LISTS: A PROBABILISTIC ALTERNATIVE TO BALANCEDTREES》by William Pugh

2.《CONCURRENT MAINTENANCE OF SKIP LISTS》by William Pugh

3. 2005 年魏冉冬令营论文，《让算法的效率“跳起来”！—— 浅谈“跳跃表”的相关操作及其应用》by William Pugh