实验报告：利用可持久化Treap实现点定位算法及其可视化

一、实验背景与目的

在计算几何中，**点定位问题（Point Location Problem）**是最基础、最核心的问题之一。它的形式定义是：给定一组几何对象（如线段、多边形、三角网等）组成的平面划分结构，以及一个任意查询点，判断该点位于哪个区域或属于哪一个对象。这个问题在理论计算几何、计算机图形学、地理信息系统（GIS）、导航、路径规划、空间数据库、计算机视觉等多个实际场景中有着广泛的应用。

尤其是在**GIS 地图系统**中，系统必须快速判断用户点击或输入的坐标属于哪个行政区域、哪片地块或哪种地貌区域；在**图形界面开发中**，需要根据鼠标坐标判断其位于哪个控件或图形元素上；在**机器人路径规划**或**物理仿真系统**中，也经常涉及到对空间区域进行快速判断和分类。由此可见，点定位问题是许多高级系统功能实现的基础能力之一。

我们通常会将点定位问题规约成查询当前点在x轴上上方/下方最近的一条线段，这个问题可以通过对横坐标排序+x轴每一个x暴力建平衡树的方式做到在$O(n^2)$预处理和$O(logn)$的查询时间内完成，但实践中，$O(n^2)$的预处理复杂度往往是不可接受的。

另一种做法是梯形图法，其利用**随机增量构建算法（Randomized Incremental Construction）**，并结合搜索结构（如DAG）来实现快速查询。在理想情况下，梯形图的查询时间为 $O(\log n)$，构建时间和空间复杂度为 $O(n \log n)$。但不幸的是，梯形图法很难处理线段相交的情况，或者需要加上一个生硬的预处理来预先分割相交的线段。 而基于Treap的方法可以很方便的扩展，修改为支持

为解决上述问题，我们考虑在该问题中使用可持久化数据结构。可持久化数据结构允许在保留旧版本数据的同时，对数据结构进行更新，从而支持对任意历史版本的访问。在点定位问题中，利用可持久化数据结构，可以在一次预处理后，根据查询点的坐标快速定位到对应的版本，并在该版本下执行搜索操作，确定其所属区域。

其中，**Treap**（树堆）是一种结合了二叉搜索树和堆性质的数据结构，通过随机化策略保持结构平衡。引入可持久化机制后，Treap不仅支持高效的插入、删除和查询操作，还能保留历史状态，实现对任意时间点的平面结构回溯。这使得可持久化Treap成为解决动态点定位问题的有力工具。

此外，在本文中，我们将使用 非旋Treap来维护。非旋Treap，也称为**FHQ Treap**。与传统Treap通过旋转操作（如左旋、右旋）来维护树的平衡不同，非旋Treap采用**分裂（Split）**和**合并（Merge）**操作来维护平衡，从而避免了旋转操作的复杂性。非旋Treap的分裂和合并操作具有良好的期望时间复杂度，通常为 $O(\log n)$。由于避免了旋转操作，非旋Treap在实际运行中表现出更稳定的性能，尤其在处理大量动态操作时更为高效。

二、相关数据结构

1. 事件队列

在点定位算法中，事件队列是算法执行的核心驱动机制。它按照事件的横坐标（x 值）进行排序，确保算法以从左到右的顺序处理所有事件。与线段相交算法相同，我们使用优先队列维护这个队列。

1.1 事件类型

事件队列中的事件主要包括：

- **线段起点事件**：表示一条线段的起始位置，需将该线段加入当前的状态结构中。
- **线段终点事件**：表示一条线段的结束位置，需将该线段从当前的状态结构中移除。
- **相交事件**：表示两条线段在此处相交。

这些事件按照其横坐标进行排序，若横坐标相同，则按照事件类型的优先级进行排序（例如，线段起点优先于相交点，相交点优先于线段终点）。

1.2 事件处理流程

处理事件队列的流程如下：

1. **初始化**：将所有线段的起点和终点事件以及所有查询点事件加入事件队列。
2. **排序**：按照事件的横坐标对事件队列进行排序。
3. **遍历事件队列**：依次处理每个事件：
    - **线段起点事件**：将对应的线段插入到当前的状态结构中。
    - **线段终点事件**：将对应的线段从当前的状态结构中删除。
    - **相交点事件**：在可持久化平衡树上交互2个点的位置。

通过上述流程，算法能够动态维护当前的线段状态。需要注意的是，相交点事件并不天生出现在事件队列中，而是通过每次起始事件，终点事件，或其他相交点事件出现后对新出现的相邻点对的检查发现获得

2. 非旋Treap及其持久化

2.1.Treap

Treap 是一种结合了二叉搜索树（BST）和堆（Heap）性质的数据结构。

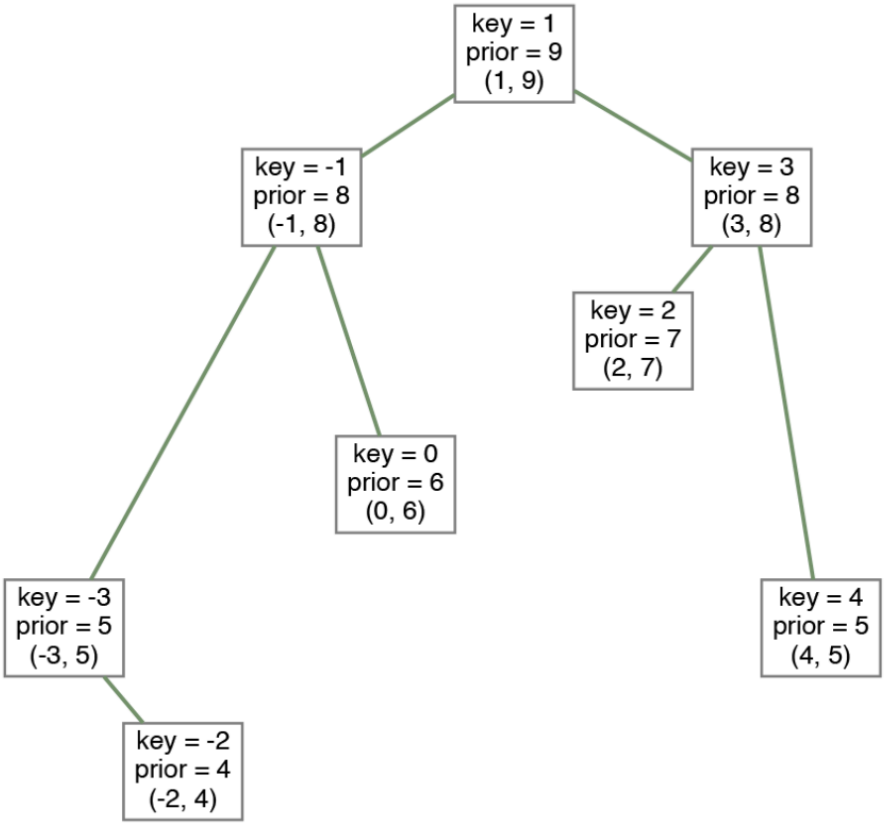
- **BST 性质**：对于任意节点，其左子树的所有节点键值小于该节点，右子树的所有节点键值大于该节点。
- **堆性质**：每个节点还拥有一个随机生成的优先级（priority），并满足堆的性质（通常为小根堆，即父节点的优先级小于子节点）。

这种结构通过随机化优先级来保持树的平衡，避免了最坏情况下 BST 退化为链表的问题。

操作与实现

- **插入**：将新节点插入 BST 中，随后根据优先级进行旋转操作，以维持堆的性质。
- **删除**：找到要删除的节点，通过旋转将其移至叶子节点位置，然后删除。
- **查找**：按照 BST 的性质进行查找操作。

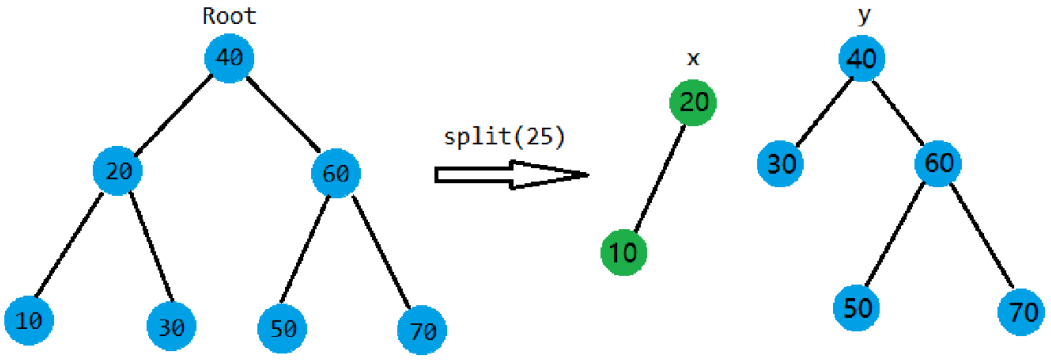
这些操作的平均时间复杂度为 O(log n).



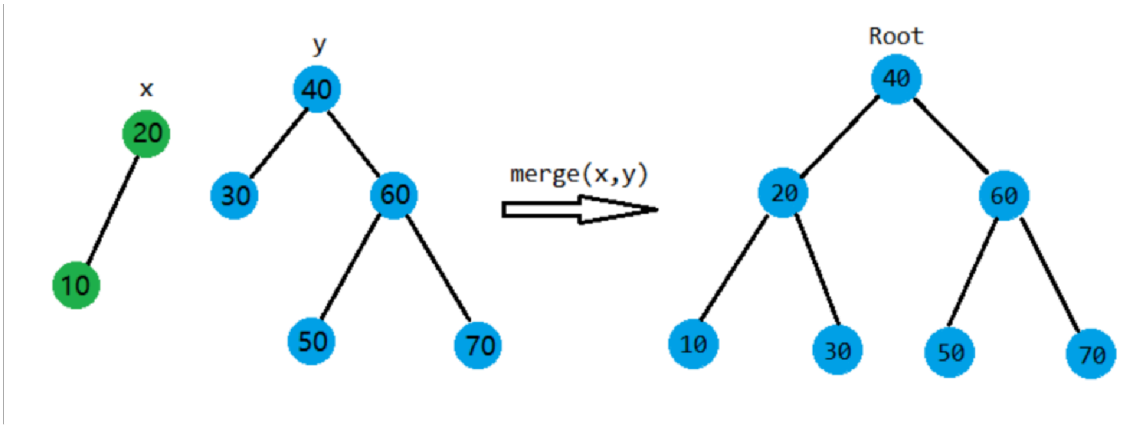
2.2非旋Treap

非旋Treap是一种不需要旋转操作的 Treap 变种，主要通过分裂（split）和合并（merge）操作来维护树的平衡。

- **分裂（split）**：将一棵 Treap 按照某个键值分裂为两棵子树，左子树的所有键值小于等于该键值，右子树的所有键值大于该键值。



- **合并（merge）**：将两棵 Treap 合并为一棵新的 Treap，前提是左子树的所有键值小于等于右子树的所有键值。



由于不需要支持旋转，在持久化过程中只需要对那些被split和merge操作波及的那2条链上的节点创建新节点即可。

- **插入**：通过 split 将原树分裂为两棵子树，然后将新节点与这两棵子树合并。
- **删除**：通过 split 将包含目标节点的子树分裂出来，然后将其删除，最后将剩余的子树合并。
- **查找**：按照 BST 的性质进行查找操作。

这些操作的时间复杂度为 $O(logn)$，且由于不涉及旋转，常数因子较小，性能稳定。

2.3.可持久化非旋 Treap

可持久化非旋Treap 是在非旋 Treap 的基础上实现的支持历史版本查询的数据结构。其核心思想是在每次修改操作（如插入、删除）时，不直接修改原有节点，而是复制并修改相关节点，从而保留原有版本的数据结构。

这种特性在需要历史版本查询的场景中非常有用，例如版本控制系统、时间旅行查询等。

- **节点复制**：在进行 split 和 merge 操作时，遇到需要修改的节点时，先复制该节点，然后在副本上进行修改。
- **版本管理**：每次操作后，保存新的根节点指针，形成一个版本链表，支持按版本号访问历史数据。

通过这种方式，可持久化非旋Treap 实现了在 $O(logn)$ 时间复杂度内的版本切换和查询，且空间复杂度为 $O(nlogn)$，在实际应用中表现良好。

三.算法流程

- **输入**：一组平面中的线段 , 以及多个查询点；
- **预处理**：
  1. 将所有线段按横坐标拆分为事件（插入/删除),并插入事件队列中；
  2. 使用可持久化Treap，按x坐标维护平衡树信息，其中平衡树的中序信息为当前x坐标下各个线段的y值由小到大排序；
  3. 处理事件队列队首的事件，依次插入/删除，并查询更新点对的前继/后驱中是否会出现相交情况并判断，如果出现相交情况将该事件插入即可。
- **查询**：当查询当前点的下方最近线段时，只需要找到所对应的历史版本平衡树，并在该树中查询即可。

四. 实验结果，运行速度分析及可视化

- 使用 C++11实现算法整体逻辑，并使用Qt进行了可视化实现。

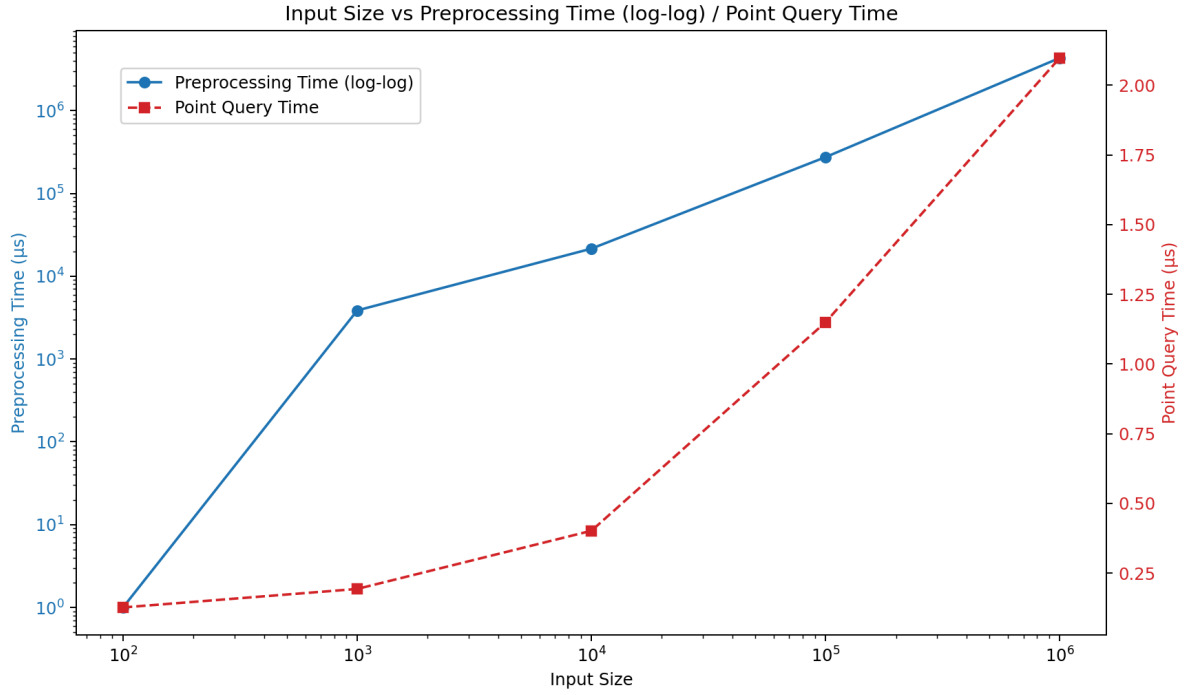
  由于该算法的性能依赖于交点数量，为了方便比较性能，我们采取了**假设输入线段无交点的版本**进行性能测试和可视化。

实验环境说明：

- **运行平台**：Intel i7-14700HX，32GB DDR5 内存，Windows 11
- **实现语言**：C++11，开启O2优化。
- 点查询事件由于过短，采取重复100W次求平均的策略。

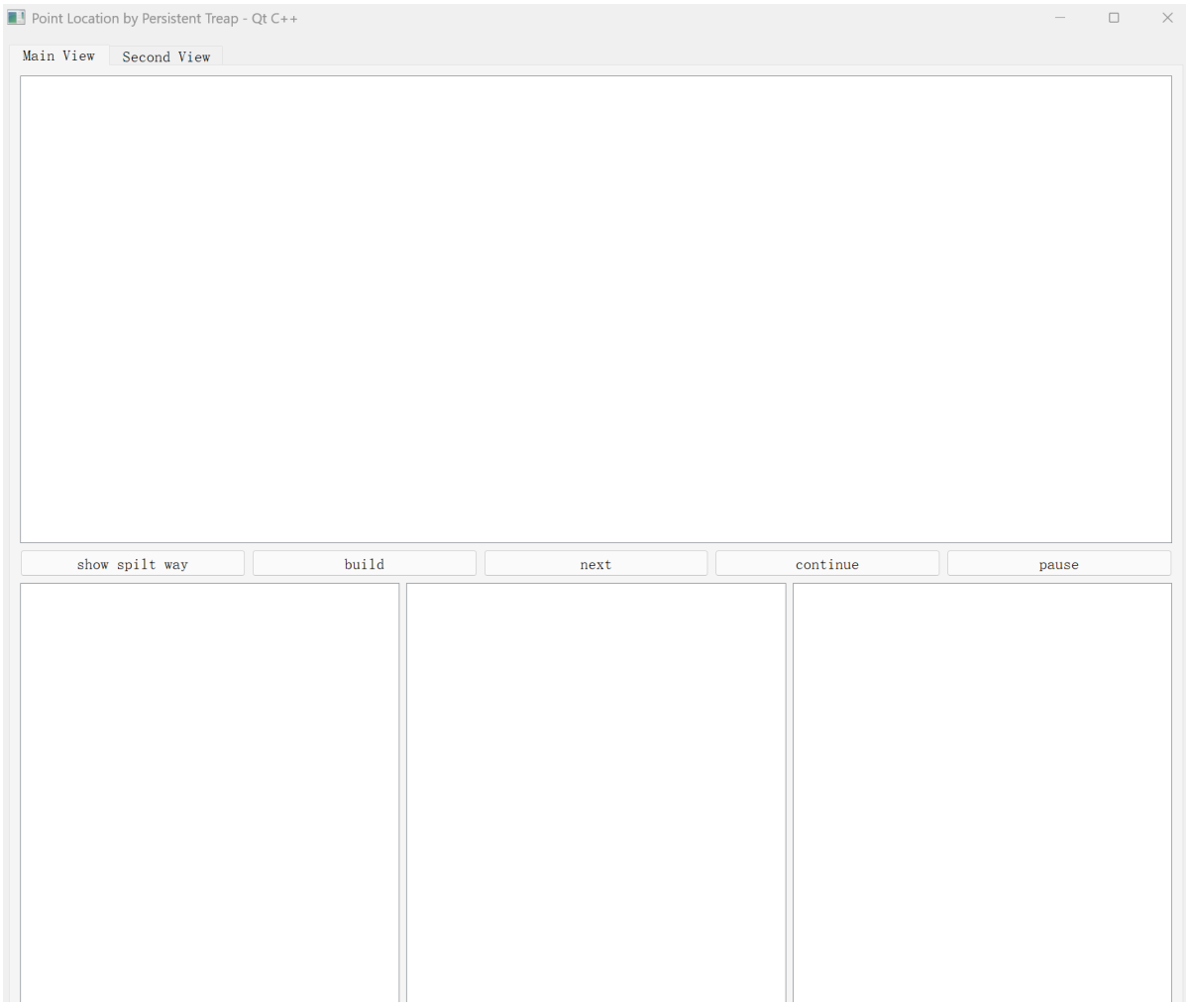
1. 性能测试

输入规模	100	1000	10000	100000	1000000
预处理时间	1 微秒	3860 微秒	21522 微秒	274238 微秒	4322737 微秒
点查询时间	0.126445 微秒	0.192793 微秒	0.401438 微秒	1.149626 微秒	2.098005 微秒

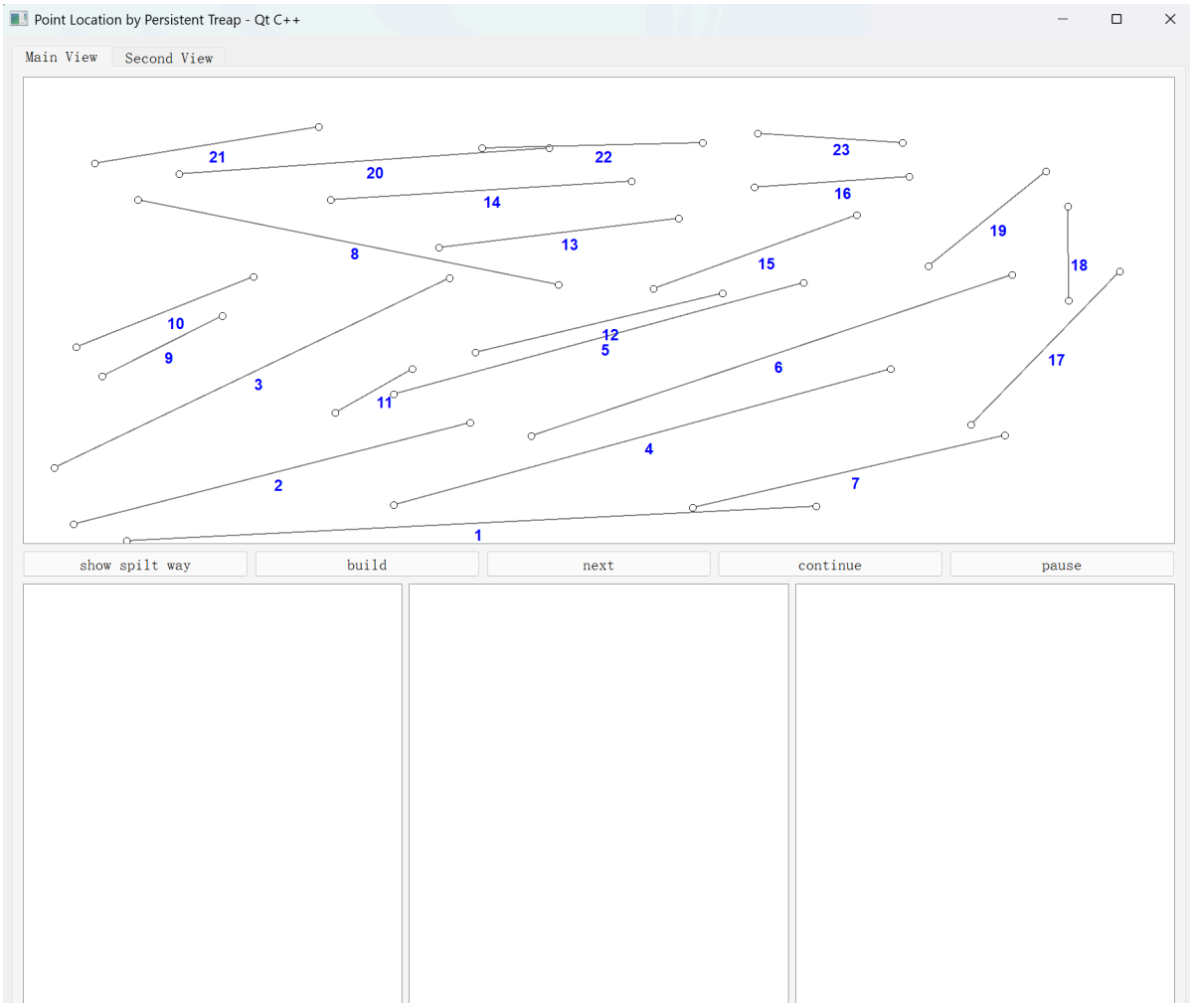


该性能测试符合预期，面对百万量级的输入，仍然能在5s内运行完毕，并在3微秒内运行完成。

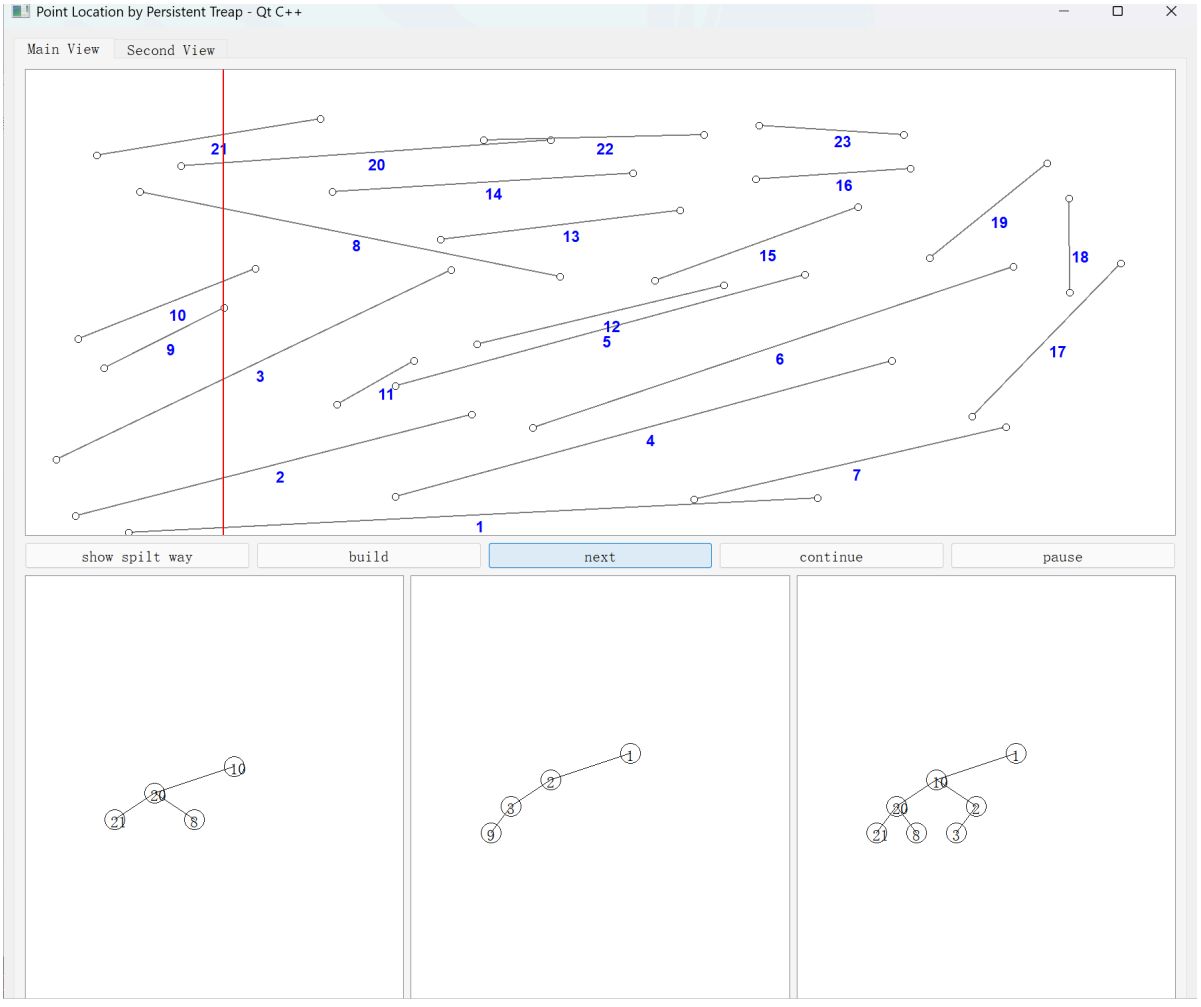
2. 可视化界面展示



我们可以在上图中手动绘制线段。当绘制完成时，我们可以点击build进行预处理建树。

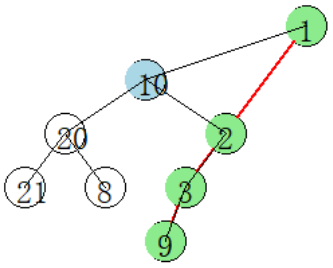


随着build，出现红线从左到右进行扫描，依次处理各个事件：



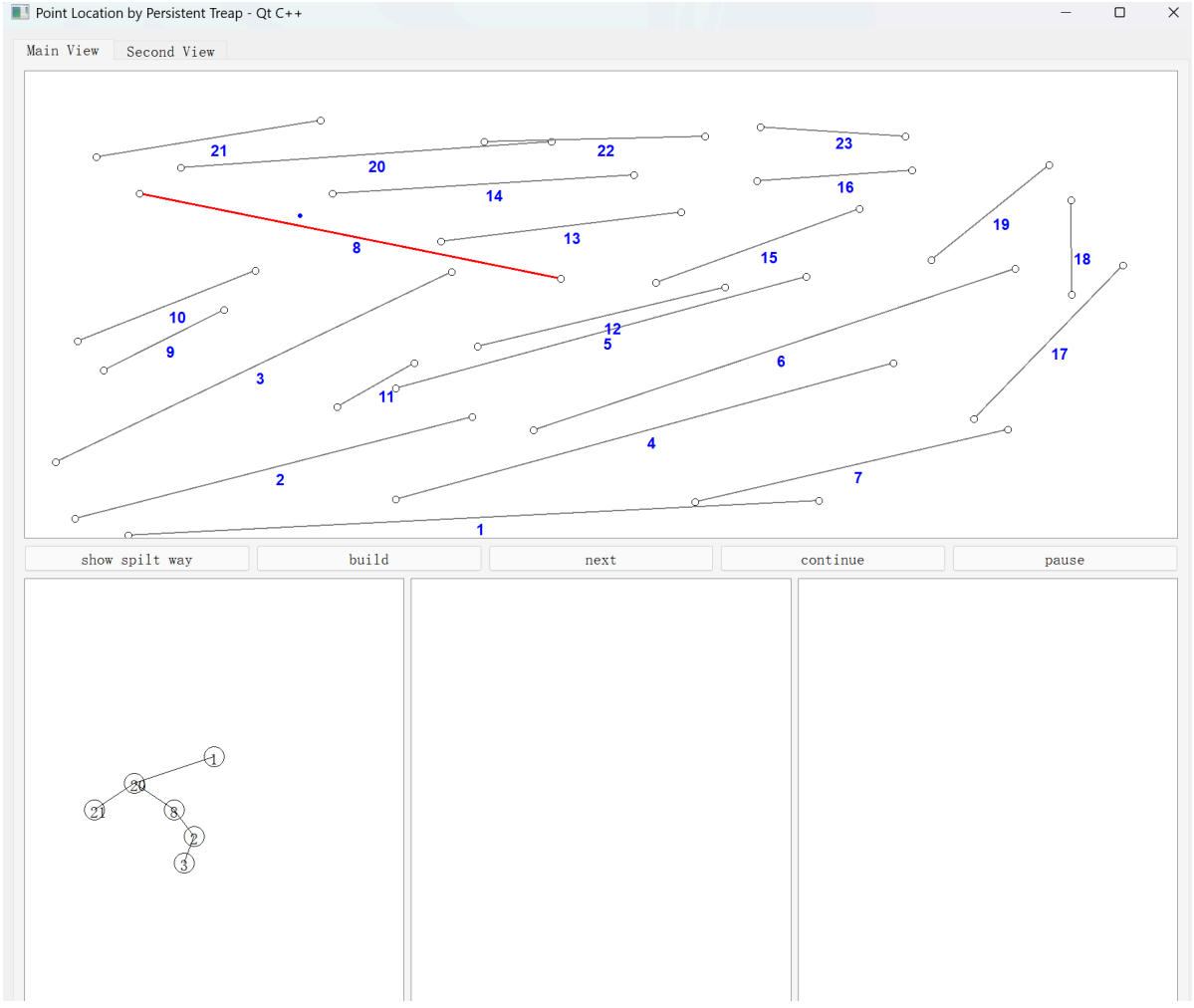
而下侧的3个框则表征了当前可持久化平衡树的状态，以上图为例，当前红线处于9号线的右端点，此时需要在原树中删除9号节点，此时我们需要将原树沿着9号点段劈成2部分，此时下框1和下框2分别表示了沿着9号节点被劈成2半的平衡树，而图三则显示了9号节点被删除后平衡树的状态。

当然，我们可以点击show split way按钮显示原来那颗平衡树是怎么沿着9号节点劈开的。



我们将沿着split路径遇到的节点逐个进行染色，被划分成左边的我们将其染成蓝色，被划分成右边的将其染成绿色，所以染色的节点我们都有可能在后续对其进行修改，所以我们对其进行新建节点操作，并将新增的边画为红色。以此展示可持久化Treap的split过程。

当预处理完成后，可视化系统将切换到点定位模式，此时我们在平面中任意点一点，系统将可视化当前x坐标所在的平衡树以及当前点的下方最近的一条直线（这是因为图形学坐标原点在左上方）。



参考文献

[1] Seidel, R., & Aragon, C. R. (1989). Randomized search trees. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC)*, 540–545. https://doi.org/10.1145/73007.73059

[2] de Berg, M., van Kreveld, M., Overmars, M., & Schwarzkopf, O. (2008). *Computational Geometry: Algorithms and Applications* (3rd ed.). Springer. https://doi.org/10.1007/978-3-540-77974-2

[3] 范浩强. FHQ Treap（无旋 Treap）详解.