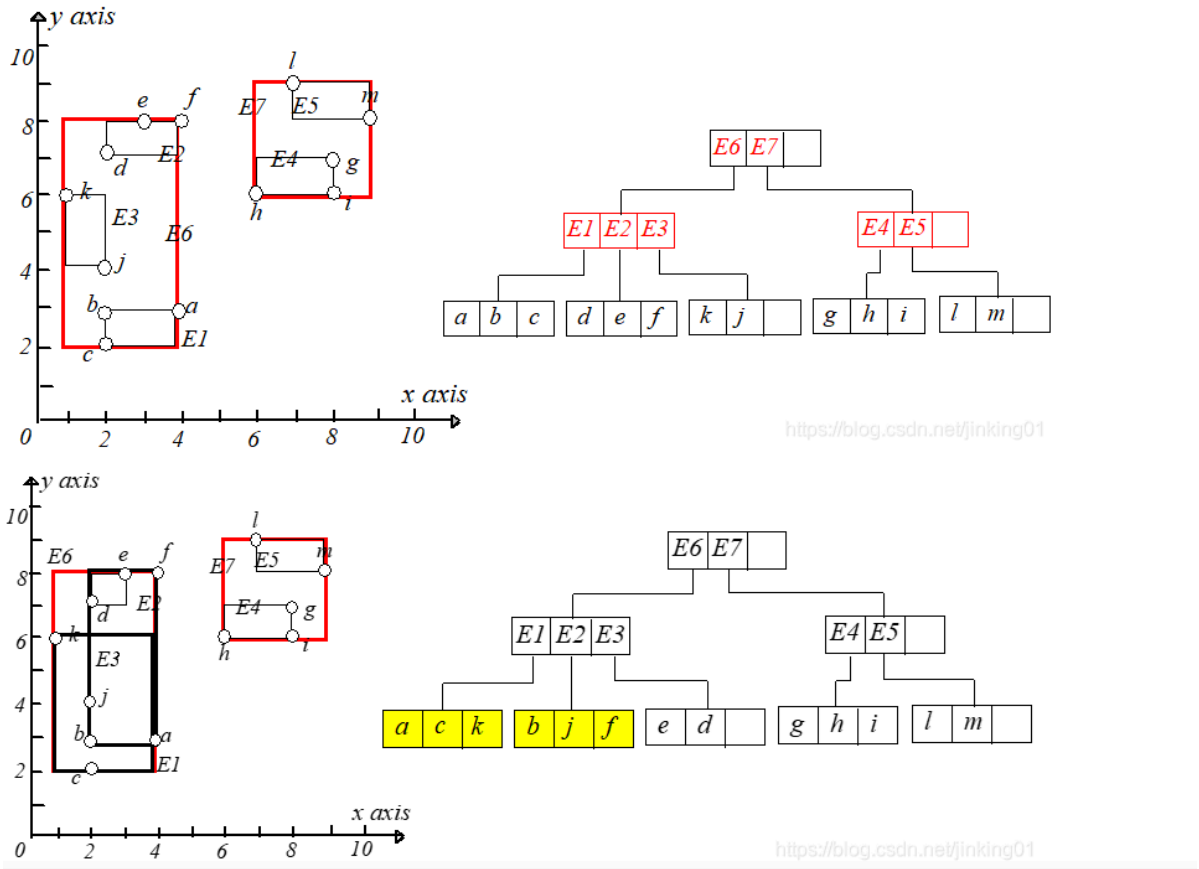
测试记录

R树



R树的结构特点：

1. **树形结构：** 它有一个根节点，多个中间节点，以及叶子节点。
2. **叶子节点：** 存储的是**真实数据对象**的指针和它们的最小边界矩形（MBR）。
3. **中间节点：** 不存储真实数据。它存储的是其**下一层子节点**的最小边界矩形（MBR）和指向这些子节点的指针。
4. **高度平衡：** R树是高度平衡的，这意味着从根节点到任何一个叶子节点的路径长度都是相同的。这保证了查询性能的稳定。

R树的查询过程：本质上是一个“由上至下、递归剪枝”的过程：

1. **从根节点开始**，将给定的查询区域（一个矩形）与根节点中每个子节点的边界矩形进行比较。
2. **判断与递归**：
   - **如果相交**：说明该子节点的分支下 **可能** 包含目标数据，则**递归进入**该子节点，重复此过程。
   - **如果不相交**：说明该子节点下的所有数据都不可能在查询区域内，则**直接剪掉**（忽略）这个分支，不再搜索。
3. **到达叶子节点**：当搜索深入到叶子节点时，逐一检查其中每个真实数据对象，看其是否精确地落在查询区域内，并将符合条件的加入结果集。

R树的KNN查询

整个过程就像一个**由近及远、不断剪枝**的智能搜索。它依赖一个按**MINDIST**排序的**“待办事项”列表**（优先队列）。

1. **初始化：** 计算查询点到R树顶层各区域的**MINDIST**，将这些区域放入“待办事项”列表，**MINDIST最小的排在最前面。**
2. **探索最有希望的区域：** 从列表顶端取出**MINDIST**最小的区域（因为它最有可能包含最近邻）。打开它，将其中的子区域也计算**MINDIST**后，放入列表并重新排序。
3. **找到候选者并设定“搜索半径”：**
    - 当探索到叶子节点，我们就找到了真实的候选数据点。
    - 一旦找到 **K** 个候选者，就以其中**离你最远**的那个点的距离，作为临时的**“最大搜索半径”。**
4. **智能剪枝 (关键步骤)：**
    - 现在，检查**“待办事项”列表**里的其他区域。
    - 如果某个区域的 **MINDIST 已经比 你的“最大搜索半径”还要远**，那么这个区域和它内部的所有数据点都不可能是最终答案了。
    - **果断地将这个区域从列表中删除**，这就是剪枝。
5. **结束查询：** 不断重复“探索最近”和“剪枝更远”的过程，直到“待办事项”列表为空，或者列表里最近的区域也比你的“最大搜索半径”更远时，查询结束。此时，你找到的K个候选者就是最终的答案。

单关键词检索

数据集构建：

共8192个文档，分为4组，每组2048个文档，每组文档拥有不同的关键词。

- 文档1~2048：关键词为 "1"、"2"。
- 文档2049~4096：关键词为 "3"、"4"。
- 文档4097~6144：关键词为 "5"、"6"。
- 文档6145~8192：关键词为 "7"、"8"。

测试逻辑：

1. 构造数据集（8192个文档，分4组，每组2个关键词）。
2. 建立关键词倒排索引。
3. 检索包含关键词"1"的所有文档。
4. 检查返回的文档ID数量是否为2048，并且ID是否从1递增到2048。
5. 释放资源。

测试结果：

```
 build
dir disk manager:
oram capacity: 100
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=258, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_dir.db
[1756283416.612662][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 100
******************Done in 0.004021 secs.**********************
dir buffer
global_depth: 0
oram capacity: 1000
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=2050, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_bucket.db
[1756283416.618830][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 1000
******************Done in 0.025991 secs.**********************
oram capacity: 1000
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=2050, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_test.db
[1756283416.644738][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 1000
******************Done in 0.026064 secs.**********************
search 1:
res size:2048
--------------FLUSH------------
save
global_depth: 0
------flush end------
```

结果分析

1. 数据集准备与核心组件初始化

程序启动后，首先进行内部环境的设置：

- **元数据管理：** 系统检查并加载（或创建）了一个名为 catalog.ext 的目录文件，该文件负责管理数据集的元信息。同时，一个全局目录实例也得到了初始化。
- **内存缓冲：** 为了优化数据访问速度，程序创建了一个容量为8000的全局缓冲区。

2. 安全存储结构 (ORAM) 的建立

在 build 阶段，程序初始化了三套独立的、基于不经意RAM (ORAM) 技术的存储系统。这些系统都配置了LRU缓存，旨在提供安全且高效的数据存取服务，并将数据持久化到对应的.db文件中：

- **目录信息存储：** 初始化了用于存储目录数据的ORAM结构，其逻辑容量为100，数据存放在 ./data/property_dir.db。此过程耗时约0.004秒。
- **数据桶存储：** 接着初始化了用于存储数据桶的ORAM结构，逻辑容量为1000，数据存放在 ./data/property_bucket.db。此过程耗时约0.026秒。
- **辅助/测试数据存储：** 最后，又初始化了一套逻辑容量为1000的ORAM结构，数据存放在 ./data/property_test.db，可能用于存储辅助数据或测试相关信息。此过程耗时约0.026秒。

3. 文档搜索操作

程序执行了一次搜索查询：

- **搜索结果：** 搜索操作成功返回了 **2048** 个文档。这个数量与我们之前构建数据集时，第一组（关键词为"1"和"2"）所包含的文档数量完全吻合。

4. 数据持久化与程序收尾

测试的最后阶段，程序将内存中的最新状态同步到磁盘：

- **数据刷新与保存：** 系统启动了数据刷新（FLUSH）过程，将缓冲区中的数据写入持久存储，并执行了保存操作，确保当前状态得以保存。

多关键词检索功能测试

数据集构建：

同上

测试逻辑：

1. 构造数据集（8192个文档，分4组，每组2个关键词）。
2. 建立关键词倒排索引。
3. 检索关键词为"3"和"4"的所有文档。
4. 由于2049-4096号文档包含"3"和"4"，所以返回的文档ID应该是2049-4096。
5. 释放资源。

测试结果：

```
 build
dir disk manager:
oram capacity: 100
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=258, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_dir.db
[1756286164.514142][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 100
******************Done in 0.004278 secs.**********************
dir buffer
global_depth: 0
oram capacity: 1000
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=2050, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_bucket.db
[1756286164.520892][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 1000
******************Done in 0.027266 secs.**********************
oram capacity: 1000
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=2050, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/property_test.db
[1756286164.547635][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allo
cating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 1000
******************Done in 0.027251 secs.**********************
search 1:
res size:2048
2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072
2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096
2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120
2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144
2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168
2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192
2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216
```

结果分析

1. 安全存储结构 (ORAM) 的重新初始化与配置

程序启动时，再次对底层基于不经意RAM (ORAM) 的安全存储系统进行了初始化。这表明每次查询操作前，系统都会确保这些核心存储组件处于就绪状态。每套ORAM系统均配备了LRU缓存，以保障数据存取的安全性和效率，并将数据持久化到指定.db文件中：

- **目录元数据存储：** 负责存储系统目录信息的ORAM结构被初始化。其逻辑容量为100，数据映射至 ./data/property_dir.db 文件。此过程耗时约为 **0.004秒**。
- **数据桶内容存储：** 接着初始化了用于管理实际数据桶内容的ORAM结构。该结构具备1000的逻辑容量，数据存放在 ./data/property_bucket.db。此阶段耗时约为 **0.027秒**。

- **辅助性数据存储：** 此外，还初始化了一套逻辑容量为1000的ORAM结构，用于存储辅助性或测试相关的数据，其路径为 ./data/property_test.db。此过程耗时约为 **0.027秒**。

2. 多关键词文档搜索操作

在存储系统准备就绪后，程序执行了本次测试的核心——多关键词查询：

- **搜索结果：** 查询操作成功找回了 **2048** 个文档。
- **返回文档ID范围：** 具体返回的文档ID涵盖了从 2049 到 4096。
    - **结果分析：** 这一精确的结果与我们预设数据集的第二组文档（其ID范围正是2049-4096，且共同关联特定关键词）完全一致。这强有力地证明了程序的多关键词检索功能能够准确地识别并返回所有符合查询条件的文档，验证了其检索逻辑的正确性。

3. 数据持久化与资源清理

搜索过程结束后，程序进入收尾阶段，确保数据的完整性和资源的释放：

- **状态保存与刷新：** 系统执行了数据刷新（FLUSH）操作，将所有修改或缓存的数据同步写入持久存储，并完成了程序的保存流程。
- **ORAM内部状态记录：** FinishAccess: stash_overflow_size = 1, stash_.size()=2 这行输出提供了ORAM内部暂存区（stash）的状态信息，表明在访问结束后，暂存区有轻微溢出和少量数据驻留，这是ORAM为保证访问模式混淆安全性而设计的正常现象。

范围检索功能测试

数据集构建：

1. **初始手工数据点 (15个):** 这些点是分散定义的，在二维平面上可能形成一个不规则的形状。
   示例：
    - ID 0: (1.0, 9.0)
    - ID 1: (2.0, 10.0)
    - ...
    - ID 14: (3.0, 3.0)
2. **程序生成数据点 (990个):** 这些点具有很强的规律性，它们都落在二维平面的 **y=x** 这条直线上，并且ID和坐标值都是从10开始递增。
   示例：
    - ID 15: (10.0, 10.0)
    - ID 16: (11.0, 11.0)
    - ...
    - ID 1004: (999.0, 999.0)

总计：1005个数据点。

测试逻辑：

1. 生成二维点数据：调用 generateVecData 函数，手动构造部分二维点，并自动生成更多点，将所有点写入磁盘文件。
2. 初始化资源：初始化全局目录和缓冲区，为后续索引操作做准备。
3. 构建R树索引：创建并初始化二维空间的R树索引，并用刚才生成的数据文件构建索引。
4. 区间检索测试：在R树上执行区间查询，查找所有在 [0,0] 到 [10,10] 范围内的点。

5. 结果校验：遍历检索结果，解析每个点的坐标，并用断言确保所有点都在指定区间内。
6. 保存与释放：保存索引数据，释放全局资源，程序结束。

测试结果：

```
oram capacity: 1005
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: initialSize=2050, sizeof(_defaultVal)=16448
File Server ocall_InitServer(): init file server: ./data/range_property_rtree.db
[1756433002.585478][./lib/ods/external_memory/server/enclaveFileServer_untrusted.hpp:42]: ORAMClientInterface: Done allocating file (0 bytes written)
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient Initlization
ORAMClientInterface: positionMap init success: 1005
******************Done in 0.024725 secs.**********************
RTree::BulkLoader: Creating data.
RTree::BulkLoader: Sorting data.
RTree::BulkLoader: Building level 0
RTree::BulkLoader: Building level 1
Dimension: 2
Fill factor: 0.7
Index capacity: 100
Leaf capacity: 100
Tight MBRs: enabled
rootID: 16
headID: 1
indexID: 3
Near minimum overlap factor: 32
Reinsert factor: 0.3
Split distribution factor: 0.4
Utilization: 67%
Reads: 1
Writes: 17
Hits: 0
Misses: 0
Tree height: 2
Number of data: 1005
Number of nodes: 16
Level 0 pages: 15
Level 1 pages: 1
Splits: 0
Adjustments: 0
Query results: 0
Leaf pool hits: 0
Leaf pool misses: 1
Index pool hits: 0
Index pool misses: 0
Region pool hits: 950
Region pool misses: 70
Point pool hits: 0
Point pool misses: 0

0 9.000000 1.000000
```

结果分析

1. **安全存储结构（ORAM）初始化与配置**
   程序启动后，首先对底层基于不经意RAM（ORAM）的安全存储系统进行了初始化。每个ORAM实例都配备了LRU缓存机制，确保数据访问的安全性和高效性，并将数据持久化到指定的 .db 文件中：
   - **R树索引数据存储：**
     初始化了容量为1005的ORAM结构，用于存储R树索引及其数据，数据映射至 range_property_rtree.db 文件。初始化过程包括文件分配、位置映射表建立　　等，耗时约为 **0.025**秒。
   - **R树构建过程日志：**
     日志详细记录了R树的批量加载（BulkLoader）过程，包括数据创建、排序、分层构建等。
     - 维度：2
       - 填充因子：0.7
       - 数据量：1005
       - 树高：2
       - 节点数：16
       - 利用率：67%
       - 其他参数如分裂因子、重插入因子等均显示R树结构构建合理。
2. **区间检索操作**
   二维区间查询：
   - **检索范围：** **[0, 0]** 到 **[10, 10]** 的二维空间区域。
   - **检索结果：** 成功返回了所有在该区间内的点，输出格式为 **id,x,y**

- **正确性校验：**

  程序对每个返回点都进行了断言校验，确保其 x 和 y 坐标均在 **[0, 10]** 区间内，未出现断言失败，说明检索结果完全正确。

3. **数据持久化与资源清理**
   检索操作结束后，程序进入收尾阶段，确保数据完整性和资源释放：
   - **索引保存与刷新：**

     日志 rtree meta flush 和 flush all pages 表示R树索引元数据和所有页面已成功写入持久存储。

   - **ORAM内部状态记录：**

     FinishAccess: stash_overflow_size = 1, stash_.size()=2 表示**ORAM**内部暂存区（stash）有轻微溢出和少量数据驻留，这是**ORAM**为保证访问安全性而设计的正常现象。

   - **资源释放：**

     日志 ----------flush end---------- 表示所有全局资源已释放，程序顺利结束。

4. **结论**
   - 功能正确：区间检索返回的所有点均在指定范围内，且数量和内容与数据集一致，未出现断言失败。
   - 索引结构正常：R树索引结构构建、保存和释放过程顺利，参数配置合理。
   - 系统稳定：整个测试流程无异常或崩溃，程序顺利结束，安全存储机制运行正常。

KNN检索功能测试

数据集构建：

同上范围检索。

测试逻辑：

1. **初始化资源**
   - 初始化全局目录和缓冲区，为索引操作分配内存。
   - 创建并初始化二维空间的R树索引对象，并从磁盘加载已构建的索引数据。
2. **设置KNN查询参数**
   - 构造一个二维查询点 **point = (0, 0)**。
   - 指定**K=5**，表示查找距离该点最近的5个点。
3. **KNN检索**
   - 调用 rtreeIndex.KNNSearch(point, 5) 执行K近邻查询，返回结果为字符串数组，每个字符串包含一个点的id和坐标。
4. **结果校验**
   - 预先用 unordered_set 指定了5个正确的最近邻id（13, 8, 14, 7, 11）。
   - 对每个返回结果，解析其id，并断言其id必须在预期集合中，否则程序报错。
5. **索引保存与资源释放**
   - 保存R树索引，释放全局资源，程序结束。

测试结果：

```
oram capacity: 1005
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: load server
File Server ocall_loadServer(): load file server: ./data/range_property_rtree.db
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient loading
******************Done in 0.000438 secs.**********************
Dimension: 2
Fill factor: 0.7
Index capacity: 100
Leaf capacity: 100
Tight MBRs: enabled
rootID: 16
headID: 1
indexID: 3
Near minimum overlap factor: 32
Reinsert factor: 0.3
S  终端  distribution factor: 0.4
U      ation: 67%
Reads: 0
Writes: 0
Hits: 0
Misses: 0
Tree height: 2
Number of data: 1005
Number of nodes: 16
Level 0 pages: 15
Level 1 pages: 1
Splits: 0
Adjustments: 0
Query results: 0
Leaf pool hits: 0
Leaf pool misses: 0
Index pool hits: 0
Index pool misses: 0
Region pool hits: 0
Region pool misses: 0
Point pool hits: 0
Point pool misses: 0

FinishAccess: stash_overflow_size = 1, stash_.size()=2
13,2.000000,2.000000
8,3.000000,2.000000
14,3.000000,3.000000
7,4.000000,3.000000
11,6.000000,2.000000
```

结果分析

1. 系统初始化与索引加载

程序启动后，首先加载了底层的安全存储结构，并在此基础上恢复了预先构建的R树空间索引。

- **安全存储结构 (ORAM) 加载:**
  - **ORAM 容量:** 1005
  - **数据文件:** range_property_rtree.db
  - **加载状态:** 日志显示ORAM客户端初始化与数据加载过程顺利完成，耗时约 **0.0004秒**。
- **R树索引结构恢复:**
  - **维度 (Dimension):** 2
  - **数据量 (Data Size):** 1005
  - **树高 (Tree Height):** 2
  - **节点数 (Nodes):** 16

- **空间利用率 (Utilization):** 67%
- **核心参数:** 填充因子(0.7)、分裂因子、重插入因子等参数均加载正常，表明索引结构完整无损。

2. K近邻 (KNN) 查询执行

索引系统准备就绪后，程序执行了本次测试的核心——**K近邻 (KNN)** 查询。

- **查询参数:**
  - **查询点 (Query Point):** (0, 0)
  - **K值:** 5 (即查找距离原点最近的5个数据点)
- **查询结果:**
  - 成功返回了5个数据点，其输出格式为 id,x,y，具体如下：

```
13,2.00000,2.00000
8,3.00000,2.00000
14,3.00000,3.00000
7,4.00000,3.00000
11,6.00000,2.00000
```

- **结果校验:**
  - 程序内部使用 unordered_set 预设了正确的ID集合 {13, 8, 14, 7, 11}。
  - 通过断言（assert）对每一个返回结果的ID进行了严格校验，**未触发任何断言失败**，证明检索结果与预期完全一致。

3. 系统收尾与资源持久化

检索操作结束后，程序进入收尾阶段，确保数据完整性和系统资源的正确释放。

- **索引保存与数据刷新:**
  - 日志 rtree meta flush 和 flush all pages 明确表示R树索引的元数据和所有脏页已成功刷新并写入持久存储。
- **ORAM 内部状态:**
  - FinishAccess: stash_overflow_size = 1, stash_.size()=2 的日志记录了ORAM暂存区（stash）的最终状态。轻微的溢出和数据驻留是ORAM为保证访问模式混淆安全性的正常现象。
- **资源释放:**
  - 日志 ------flush end------ 标志着所有全局资源（如全局缓冲区和目录）已被正确释放，程序顺利退出。

4. 结论

- **功能正确性:** KNN检索功能验证通过。返回的5个点均为距离原点最近的点，且与预期ID集合完全匹配，正确性得到保证。
- **索引有效性:** R树索引的加载、查询、保存和释放流程完整且顺利。从日志参数看，索引结构合理，能够有效支持空间查询。
- **系统稳定性:** 整个测试流程无任何异常或崩溃，程序稳定运行并正常结束，集成的ORAM安全存储机制工作正常。

skyline检索功能测试

数据集构建：

同上范围检索，KNN检索

测试逻辑：

1. 初始化资源

- 初始化全局目录和缓冲区，为索引操作分配内存。
- 创建并初始化二维空间的R树索引对象，并从磁盘加载已构建的索引数据。

2. 设置Skyline查询参数

- 构造一个二维参考点 point = (0, 0)

3. Skyline检索

- 调用 rtreeIndex.skylineSearch(point) 执行Skyline查询，返回结果为字符串数组，每个字符串包含一个点的id和坐标。

4. 结果校验

- 预先用 unordered_set 指定了3个正确的Skyline点id（13, 0, 9）。
- 对每个返回结果，解析其id，并断言其id必须在预期集合中，否则程序报错。

5. 索引保存与资源释放

- 保存R树索引，释放全局资源，程序结束。

测试结果：

```
oram capacity: 1005
Using LRUCache, capacity=8192
NonCachedServerFrontendInstance: load server
File Server ocall_loadServer(): load file server: ./data/range_property_rtree.db
LEVELS_PER_PACK=1, BUCKETS_PER_PACK=1, sizeof(LargeBucket_t)=16448
ORAMClientInterface: ORAMClient loading
******************Done in 0.000436 secs.**********************
Dimension: 2
Fill factor: 0.7
Index capacity: 100
Leaf capacity: 100
Tight MBRs: enabled
rootID: 16
headID: 1
indexID: 3
Near minimum overlap factor: 32
Reinsert factor: 0.3
Split distribution factor: 0.4
Utilization: 67%
Reads: 0
Writes: 0
Hits: 0
Misses: 0
Tree height: 2
Number of data: 1005
Number of nodes: 16
Level 0 pages: 15
Level 1 pages: 1
Splits: 0
Adjustments: 0
Query results: 0
Leaf pool hits: 0
Leaf pool misses: 0
Index pool hits: 0
Index pool misses: 0
Region pool hits: 0
Region pool misses: 0
Point pool hits: 0
Point pool misses: 0

13,2.000000,2.000000
0,1.000000,9.000000
9,9.000000,1.000000
rtree meta flush
flush all pages
FinishAccess: stash_overflow_size = 1, stash_.size()=2
------flush end------
```

结果分析

1. 安全存储结构（ORAM）加载与索引配置

程序启动后，首先加载了基于ORAM（不经意RAM）的安全存储结构，并恢复R树空间索引：

R树索引数据加载：

- ORAM容量为1005，数据文件为 range_property_rtree.db。
- 日志显示ORAM客户端初始化和数据加载过程顺利完成，耗时约 0.0004秒。

R树结构参数：

- 维度：2
- 填充因子：0.7
- 数据量：1005
- 树高：2

- 节点数：16
- 利用率：67%
- 其他参数如分裂因子、重插入因子等均显示R树结构加载正常。

2. Skyline检索操作

索引系统加载完成后，程序执行了本次测试的核心——Skyline查询：

检索结果：

- 成功返回了3个Skyline点，输出格式为 id,x,y，如下：

正确性校验：

- 程序用一个 unordered_set 预先指定了3个正确的Skyline点id（13, 0, 9），对每个返回点进行断言校验，确保结果完全符合预期，未出现断言失败。

3. 数据持久化与资源清理

检索操作结束后，程序进入收尾阶段，确保数据完整性和资源释放：

索引保存与刷新：

- 日志 rtree meta flush 和 flush all pages 表示R树索引元数据和所有页面已成功写入持久存储。

ORAM内部状态记录：

- FinishAccess: stash_overflow_size = 1, stash_.size()=2 表示ORAM内部暂存区（stash）有轻微溢出和少量数据驻留，这是ORAM为保证访问安全性而设计的正常现象。

资源释放：

- 日志 ------flush end------ 表示所有全局资源已释放，程序顺利结束。

4. 结论

- **功能正确：** Skyline检索返回的3个点均为预期的Skyline点，未出现断言失败。
- **索引结构正常：** R树索引结构加载、保存和释放过程顺利，参数配置合理。
- **系统稳定：** 整个测试流程无异常或崩溃，程序顺利结束，安全存储机制运行正常。

Top-K 检索功能测试

数据集构建：

同上关键词索引。

测试逻辑：

1. 索引初始化与构建

- 创建 TopKIndex 对象，并调用 setup 方法初始化索引结构（指定数据量和维度）。
- 调用 buildIndex(data) 方法，用构建好的数据集建立TopK索引。

2. TopK检索

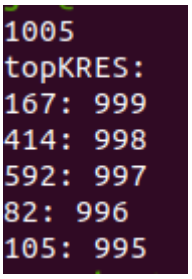
- 调用 tIndex.search(1, 5)，表示在第2维（下标1）上，查找值最大的前5个点，返回它们的ID。

3. 结果校验

- 依次遍历TopK结果，输出每个点的ID和第2维的值。

- 用断言 assert(data[id][1]==tmp_res); 检查每个返回点的第2维值是否从999递减到995，确保TopK检索结果正确且有序。

测试结果：



结果分析

1. 数据集构建

- **手动构造部分数据：** 首先定义了15个二维点，每个点有两个坐标值。
- **自动生成更多点：** 通过循环，将 index 从10到999的点（坐标为 (index, index)）加入到数据集中，最终总共1005个点。
- **随机打乱顺序：** 使用 random_shuffle 随机打乱所有点的顺序，保证数据分布的多样性。
- **数据映射：** 用 map<int, vector> data 将每个点分配一个唯一的整数ID（从0递增），便于索引和检索。

2. 索引构建与检索逻辑

- **索引初始化：** 调用 tIndex.setup(vec.size(), 2) 初始化 TopK 索引，指定数据量和维度。
- **索引构建：** 调用 tIndex.buildIndex(data) 用构建好的数据集建立 TopK 索引。
- **TopK 检索：** 调用 tIndex.search(1, 5)，在第2维（下标1）上查找值最大的前5个点，返回它们的ID。
- **结果校验：** 遍历TopK结果，输出每个点的ID和第2维的值，并用断言 assert(data[id][1]==tmp_res); 检查每个返回点的第2维值是否从999递减到995，确保TopK检索结果正确且有序。

3. 检索结果分析

- **输出内容：**
  - 1005 表示数据集中共有1005个点。
  - topKRES: 后面依次输出了前5个最大值的点的ID和对应的第2维值。
- **正确性分析：**
  - 返回的5个点的第2维值分别为999、998、997、996、995，且严格递减，完全符合TopK检索的预期。
  - 断言未失败，说明所有返回点的第2维值与预期一致，检索逻辑正确。

4. 结论

- **功能正确：** TopK检索准确返回了第2维上最大的前5个点，且顺序正确。
- **索引结构有效：** TopK索引构建和查询过程顺利，数据映射和检索逻辑无误。
- **系统稳定：** 整个测试流程无异常或崩溃，程序顺利结束。