华东师范大学软件工程学院上机实践报告

课程名称:	数据结构与算法实践	年　　级:	2022 级
实践名称:	空间和文本查询效率分析	指导教师:	王丽苹
姓　　名:	李鹏达	学　　号:	10225101460
实践时间:	2023 年 6 月		

1　内容与设计思想

随着智能手机的普及，地理信息在诸如高德地图、大众点评、饿了么等 App 中得到广泛的应用，此次数据结构期末大作业将模拟实际生活中的查询需求，完成基于地理信息和文本信息的查找任务。问题的说明如下：系统中已经收集到许多商户的信息，每家商户包括以下三项信息：

- 位置 (x, y)，$x > 0$ 且 $y > 0$；
- 商家名称；12 位 A-Z 字符串，不含小写；
- 菜系，6 位 A-Z 字符串，不含小写；

你的程序需要提供给用户以下查询的功能：

查询任务: 用户输入自己的位置点如 (ux, uy)、感兴趣的菜系和整数 k 值，程序按照由近到远输出商家名称和距离，距离相等时按照商家名称的字典序为准。在此距离精确到小数点后的 3 位(四舍五入)。若满足条件的商户不足 k 个，则输出所有满足条件的商家信息。

【输入】

第 1 行：商户的数量 m 和查询的数量 n，m 和 n 为整数，均不超过 109；

第 $2-(m+1)$ 行：商户的信息，包括商家名称，位置 x，位置 y 和菜系；

最后的 n 行：每一行表示一次查询，包括用户的位置 ux 和 uy、菜系名称、k 值；

【输出】

对应于每一次查询，按照顺序输出满足条件的商户信息，每一行对应于一家商户，若存在一次查询中无任何满足条件的商户，则输出空行即可。

例如：

【输入】

```
5 2
MCDONALD 260036 14362 FASTFOOD
HAIDILAO 283564 13179 CHAFINGDIS
KFC 84809 46822 FASTFOOD
```

```
DONGLAISHUN 234693 37201 CHAFINGDIS
SUBWAY 78848 96660 FASTFOOD
28708 23547 FASTFOOD 2    //查询离<28708 23547>最近的两家快餐店
18336 14341 CHAFINGDIS 3  //查询离<18336 14341>最近的3家火锅店
```

【输出】

```
KFC 60737.532
SUBWAY 88653.992
DONGLAISHUN 217561.327   //此时只有两家，按距离全部输出即可
HAIDILAO 265230.545
```

2 任务说明

请根据本学期学习的知识，设计算法实现上述的两类查询功能，并尝试分析算法的空间复杂度和时间复杂度，可结合数据规模、原始数据的特性等分析查询影响因素等。

1) 数据规模：200 个商家、4000 个商家、8×10^5 个商家等等；
2) 数据特性：每个规模的数据包含 1 组按商家名称升序，1 组按商家名称降序，10 组随机数据共 12 组数据集；任务中的类别和 k 值对算法的影响。
3) 查询任务的效率，可以统计不同的 k 值下的查询时间，例如在 $k = (3, 15, 75, 375, \cdots)$ 时，不同数据规模下（200 个商家、4000 个商家、8×10^5 个商家等）的查询时间。
4) 任务说明：统计任务在不同规模数据下，不同 k 值下的查询时间变化。
5) 书写要求：若采用教材内的算法实现查询，仅仅需要说明所用算法；若实践过程中涉及到自己设计的数据结构或者书本外的知识请在“实验记录和结果”中说明算法的基本思想。
6) 代码提交：请在 EOJ 平台提交查询代码，将统一统计代码运行时间。注意：允许提交多次，不计罚时。

3 本地实验环境

- CPU：11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
- 内存：16.0GB
- 操作系统：Windows 11 家庭中文版 22H2 22621.1848
- 编程语言：C++17
- 编译器：gcc 12.2.0 (MinGW-W64 x86_64-ucrt-posix-seh)

4 实验记录和结果

4.1 实验记录

4.1.1 算法一：暴力

其基本思想是构建一个 `resaurant` 结构体，记录每个餐馆的名称、坐标、菜系和与目标点的距离，并存入一个单链表中。

当进行查询时，遍历整个链表，将与所查询的菜系相同的数据存入一个新链表中。通过在结构体中重载 `operator<` 使用自定义的方式进行排序（在此处，使用的排序方式是归并排序）。排序完成后，输出前 k 组数据，不足 k 组，则输出全部数据。

实验记录	
数据存储结构	链式存储
查找算法	排序后顺序查找
数据规模	200, 4000, 80000, 800000
数据分组	4 组
是否有课堂外的算法	否

表 1: 算法一实验记录

4.1.2 算法二：使用哈希表

其基本思想是构建一个 `resaurant` 结构体，记录每个餐馆的名称、坐标和与目标点的距离。与算法一不同的是，本算法使用菜系作为 `key`，对输入数据实现按菜系分类，存入一个 `key = type, value = list<resaurant>` 的哈希表中。

当进行查询时，直接根据菜系在哈希表中找到相应的链表并根据自定义规则进行排序。排序完成后，输出前 k 组数据，不足 k 组，则输出全部数据。

实验记录	
数据存储结构	哈希表、链式存储
查找算法	排序后顺序查找
数据规模	200, 4000, 80000, 800000
数据分组	4 组
是否有课堂外的算法	否

表 2: 算法二实验记录

4.1.3 算法三：使用哈希表和 AVL 树

其基本思想是构建一个 `resaurant` 结构体，记录每个餐馆的名称、坐标和与目标点的距离。与算法一不同的是，本算法使用菜系作为 `key`，对输入数据实现按菜系分类，存入一个 `key = type, value = AVL_tree<resaurant>` 的哈希表中。

当进行查询时，直接根据菜系在哈希表中找到对应的 AVL 树，中序遍历即为按自定义顺序的结果。输出前 k 组数据，不足 k 组，则输出全部数据。

实验记录	
数据存储结构	哈希表、二叉树
查找算法	二分查找
数据规模	200, 4000, 80000, 800000
数据分组	4 组
是否有课堂外的算法	否

表 3: 算法三实验记录

4.1.4 算法四：使用 STL

其基本思想与算法二相同，但使用 `std::vector` 而不是链表和 `std::unordered_map`（哈希表），同时也使用 `std::sort`（快速排序、插入排序和堆排序的混合算法）进行排序。

实验记录	
数据存储结构	哈希表、顺序存储
查找算法	排序后顺序查找
数据规模	200, 4000, 80000, 800000
数据分组	4 组
是否有课堂外的算法	是，`std::sort`、`std::vector` 和 `std::unordered_map` 的实现

表 4: 算法四实验记录

4.2 实验结果

实验结果（查询时间统计）					
数据规模	查询中的 k 值	算法一 暴力	算法二 哈希表、链表	算法三 哈希表、AVL 树	算法四 STL
200	3	1ms	1ms	1ms	1ms
200	15	1ms	0ms	1ms	1ms
4000	3	4ms	3ms	6ms	3ms
4000	15	4ms	4ms	6ms	4ms
4000	75	7ms	5ms	7ms	5ms
80000	375	1114ms	110ms	180ms	110ms
80000	1875	1397ms	156ms	292ms	171ms
800000	3	26094ms	867ms	1523ms	713ms
800000	15	25271ms	884ms	1483ms	722ms
800000	75	25290ms	916ms	1537ms	763ms
800000	375	25139ms	1242ms	1848ms	1047ms
800000	1875	76372ms	60814ms	`duplicate error`	50530ms
EOJ 测试		TLE(16/20)	AC	AC	AC

表 5: 实验结果

最终选择的算法是**算法二**。

5 实验总结

5.1 实验结论

在实验数据规模较小时，四种算法的效率没有明显的差别。但当数据量达到 80000 及以上时，算法一的效率明显低于其他三种算法。在算法二、三、四中，算法三的效率始终低于其它两种，并且在其中一组数据中发生了异常，这可能是由于输入数据中出现了重复数据。算法四的效率略优于算法二，并且随着数据量的增大，优势逐渐提高。由于算法二中的数据结构是由自己实现的，而算法四使用了 STL，因此我们最终选择算法二。

5.2 实验收获

1. 当输入数据较多且可分类时，尤其是可能需要分类查询时，可以考虑使用哈希表进行分类存储。
2. `std::endl` 效率较低，可以考虑使用`'\n'` 代替。
3. 可以使用 `std::ios::sync_with_stdio(false), std::cin.tie(nullptr);` 来关闭流同步进而提高 `std::cin` 和 `std::cout` 的效率。
4. 对于单链表，`push_front` 的效率远高于 `push_back`。

5.3 待改进的问题

可以进一步优化 AVL 树，使其能够处理重复数据的插入。

6 代码附录

6.1 算法一：暴力

```cpp
#include <bits/stdc++.h>
#define endl '\n'
#define IO ios::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr)
using namespace std;
using ll = long long;
namespace pdli {
template <typename T>
class list {
    struct list_node {
        T entry;
```

```
11          list_node* next = nullptr;
12          list_node() {}
13          list_node(const T& item, list_node* add_on = nullptr) : entry(item), next(
                add_on) {}
14          friend bool operator==(const list_node& lhs, const list_node& rhs) {
15              return lhs.entry == rhs.entry && lhs.next == rhs.next;
16          }
17      };
18      class list_iterator {
19      public:
20          list_node* _M_node;
21          list_iterator() = default;
22          ~list_iterator() noexcept = default;
23          list_iterator(list_node* node) : _M_node(node) {}
24          list_iterator(const list_iterator& other) { _M_node = other._M_node; }
25          T& operator*() const { return _M_node->entry; }
26          T* operator->() const { return static_cast<T*>(&(_M_node->entry)); }
27          list_iterator operator++(int) {
28              list_iterator* tmp = this;
29              _M_node = _M_node->next;
30              return *tmp;
31          }
32          list_iterator operator++() {
33              _M_node = _M_node->next;
34              return *this;
35          }
36          friend bool operator==(const list_iterator& lhs, const list_iterator& rhs)
                {
37              return lhs._M_node == rhs._M_node;
38          }
39          friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs)
                {
40              return !(lhs == rhs);
41          }
42          friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs
                ) {
43              list_iterator* ans = lhs;
44              for (size_t i = 0; i < rhs; i++) {
45                  (*ans)++;
46              }
47              return *ans;
48          }
49      };
50
```

```
51 public:
52     typedef list::list_iterator iterator;
53     typedef list::list_node node;
54     list() : head(nullptr) {}
55     list(const list& other);
56     ~list() noexcept;
57     size_t size() const;
58     bool empty() const;
59     void push_back(const T& item);
60     void push_front(const T& item);
61     void pop_back();
62     void pop_front();
63     void clear();
64     void reverse();
65     void insert(iterator pos, const T& item);
66     void insert(size_t pos, const T& item);
67     void remove(const T& value);
68     iterator erase(iterator pos);
69     iterator erase(iterator first, iterator last);
70     void erase(size_t pos);
71     void erase(size_t first, size_t last);
72     void replace(const T& old_item, const T& new_item);
73     iterator begin() const;
74     iterator end() const;
75     iterator find(const T& item) const;
76     T& back() const;
77     T& front() const;
78     list& operator=(const list& other);
79     void insertion_sort();
80     void merge_sort();
81 
82 protected:
83     node* head = nullptr;
84 
85 private:
86     void _M_merge_sort(node*& list);
87 };
88 
89 template <typename T>
90 list<T>::list(const list& other) {
91     if (other.head == nullptr) return;
92     node *cur, *pre;
93     node* _head = other.head;
94     cur = new node(other.head->entry);
```

```
95      head = cur;
96      while (_head->next != nullptr) {
97          _head = _head->next;
98          cur->next = new node(_head->entry, _head->next);
99          pre = cur;
100         cur = cur->next;
101     }
102 }
103
104 template <typename T>
105 bool list<T>::empty() const {
106     return head == nullptr;
107 }
108
109 template <typename T>
110 size_t list<T>::size() const {
111     size_t cnt = 0;
112     for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
113     return cnt;
114 }
115
116 template <typename T>
117 void list<T>::push_back(const T& item) {
118     node *cur, *pre;
119     if (head == nullptr) {
120         cur = new node(item);
121         head = pre = cur;
122         return;
123     }
124     for (cur = head; cur->next != nullptr; cur = cur->next)
125         ;
126     node* new_node = new node(item);
127     cur->next = new_node;
128 }
129
130 template <typename T>
131 void list<T>::pop_back() {
132     node *cur, *pre;
133     for (cur = head; cur->next != nullptr; cur = cur->next)
134         ;
135     if (cur == head) {
136         delete cur;
137         head = cur = pre = nullptr;
138         return;
```

```
139     }
140     for (pre = head; pre->next != cur; pre = pre->next)
141         ;
142     if (cur == nullptr) throw std::underflow_error("underflow");
143     delete cur;
144     pre->next = nullptr;
145     cur = pre;
146     for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
147         ;
148 }
149
150 template <typename T>
151 T& list<T>::back() const {
152     node* cur;
153     for (cur = head; cur->next != nullptr; cur = cur->next)
154         ;
155     if (cur == nullptr) throw std::underflow_error("underflow");
156     return cur->entry;
157 }
158
159 template <typename T>
160 void list<T>::clear() {
161     node *cur, *pre;
162     if (head == nullptr) return;
163     pre = head;
164     cur = head->next;
165     while (cur != nullptr) {
166         delete pre;
167         pre = cur;
168         cur = cur->next;
169     }
170     delete pre;
171     pre = head = cur = nullptr;
172 }
173
174 template <typename T>
175 list<T>::~list() noexcept {
176     node *cur, *pre;
177     if (head == nullptr) return;
178     pre = head;
179     cur = head->next;
180     while (cur != nullptr) {
181         delete pre;
182         pre = cur;
```

```
183         cur = cur->next;
184     }
185     delete pre;
186 }
187 
188 template <typename T>
189 typename list<T>::iterator list<T>::begin() const {
190     return iterator(head);
191 }
192 
193 template <typename T>
194 typename list<T>::iterator list<T>::end() const {
195     return iterator(nullptr);
196 }
197 
198 template <typename T>
199 void list<T>::insert(iterator pos, const T& item) {
200     node *cur, *pre;
201     if (pos._M_node == head) {
202         head = new node(item, head);
203         return;
204     }
205     for (pre = head; pre->next != pos._M_node; pre = pre->next)
206         ;
207     cur = pre->next;
208     pre->next = new node(item, cur);
209 }
210 
211 template <typename T>
212 void list<T>::insert(size_t pos, const T& item) {
213     insert(begin() + pos, item);
214 }
215 
216 template <typename T>
217 typename list<T>::iterator list<T>::erase(iterator pos) {
218     node *cur, *pre;
219     iterator res(pos._M_node->next);
220     if (pos._M_node == head) {
221         node* temp = head;
222         head = temp->next;
223         delete temp;
224         return res;
225     }
226     for (pre = head; pre->next != pos._M_node; pre = pre->next)
```

```
227         ;
228     cur = pre->next;
229     pre->next = cur->next;
230     if (cur == nullptr) throw std::underflow_error("underflow");
231     delete cur;
232     return res;
233 }
234 
235 template <typename T>
236 void list<T>::erase(size_t pos) {
237     erase(begin() + pos);
238 }
239 
240 template <typename T>
241 void list<T>::erase(size_t first, size_t last) {
242     erase(begin() + first, begin() + last);
243 }
244 
245 template <typename T>
246 void list<T>::reverse() {
247     list<T> temp(*this);
248     clear();
249     while (!temp.empty()) {
250         push_back(temp.back());
251         temp.pop_back();
252     }
253 }
254 
255 template <typename T>
256 list<T>& list<T>::operator=(const list<T>& other) {
257     node *cur, *pre;
258     node* _head = other.head;
259     cur = new node(other.head->entry);
260     head = cur;
261     while (_head->next != nullptr) {
262         _head = _head->next;
263         cur->next = new node(_head->entry, _head->next);
264         pre = cur;
265         cur = cur->next;
266     }
267     return *this;
268 }
269 
270 template <typename T>
```

```
271 typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
272     for (auto it = first; it != last; it = erase(it))
273         ;
274     return last;
275 }
276
277 template <typename T>
278 void list<T>::push_front(const T& item) {
279     insert(begin(), item);
280 }
281
282 template <typename T>
283 void list<T>::pop_front() {
284     erase(begin());
285 }
286
287 template <typename T>
288 typename list<T>::iterator list<T>::find(const T& item) const {
289     for (auto it = begin(); it != end(); ++it) {
290         if (*it == item) return it;
291     }
292     return end();
293 }
294
295 template <typename T>
296 T& list<T>::front() const {
297     if (head == nullptr) throw std::underflow_error("underflow");
298     return head->entry;
299 }
300
301 template <typename T>
302 void list<T>::remove(const T& value) {
303     for (auto it = begin(); it != end();) {
304         if (*it == value) {
305             it = erase(it);
306         } else {
307             it++;
308         }
309     }
310 }
311
312 template <typename T>
313 void list<T>::replace(const T& old_item, const T& new_item) {
314     for (auto& i : *this) {
```

```
315         if (i == old_item) {
316             i = new_item;
317         }
318     }
319 }
320 
321 template <typename T>
322 void list<T>::insertion_sort() {
323     node *last_sorted, *first_unsorted;
324     if (head == nullptr) return;
325     last_sorted = head;
326     while (last_sorted->next != nullptr) {
327         first_unsorted = last_sorted->next;
328         node* pre = head;
329         bool flag = false;
330         for (node* cur = head; cur != first_unsorted; cur = cur->next) {
331             if (cur == head) {
332                 if (first_unsorted->entry <= cur->entry) {
333                     last_sorted->next = first_unsorted->next;
334                     first_unsorted->next = cur;
335                     head = first_unsorted;
336                     flag = true;
337                     break;
338                 }
339             } else {
340                 if (first_unsorted->entry > pre->entry
341                     && first_unsorted->entry <= cur->entry) {
342                     last_sorted->next = first_unsorted->next;
343                     pre->next = first_unsorted;
344                     first_unsorted->next = cur;
345                     flag = true;
346                     break;
347                 }
348             }
349             pre = cur;
350         }
351         if (!flag) last_sorted = last_sorted->next;
352     }
353 }
354 
355 template <typename T>
356 void list<T>::merge_sort() {
357     _M_merge_sort(head);
358 }
```

```
359
360 template <typename T>
361 void list<T>::_M_merge_sort(node*& list) {
362     if (list == nullptr || list->next == nullptr) return;
363     // divide
364     node *mid = list, *sec = list, *pos = list;
365     while (pos != nullptr && pos->next != nullptr) {
366         mid = sec;
367         sec = sec->next;
368         pos = pos->next->next;
369     }
370     mid->next = nullptr;
371     // sort
372     _M_merge_sort(list);
373     _M_merge_sort(sec);
374     // merge
375     node* cur = new node;
376     node* new_head = cur;
377     node* fst = list;
378     while (fst != nullptr && sec != nullptr) {
379         if (fst->entry < sec->entry) {
380             cur->next = fst;
381             fst = fst->next;
382             cur = cur->next;
383         } else {
384             cur->next = sec;
385             sec = sec->next;
386             cur = cur->next;
387         }
388     }
389     if (fst == nullptr) {
390         cur->next = sec;
391     } else {
392         cur->next = fst;
393     }
394     list = new_head->next;
395     delete new_head;
396 }
397 } // namespace pdli
398
399 struct restaurant {
400     string name;
401     string type;
402     int x;
```

```
403     int y;
404     double dis;
405     restaurant() {}
406     restaurant(const string& name, const string& type, int x, int y)
407         : name(name), type(type), x(x), y(y), dis(0) {}
408     bool operator<(const restaurant& other) const {
409         return tie(dis, name) < tie(other.dis, other.name);
410     }
411 };
412 
413 int main() {
414     IO;
415     auto clk = clock();
416     int m, n;
417     cin >> m >> n;
418     pdli::list<restaurant> ls;
419     while (m--) {
420         string name, type;
421         int x, y;
422         cin >> name >> x >> y >> type;
423         auto t = restaurant{name, type, x, y};
424         ls.push_front(t);
425     }
426     while (n--) {
427         int x, y, k;
428         string type;
429             cin >> x >> y >> type >> k;
430             pdli::list<restaurant> temp;
431             for_each(ls.begin(), ls.end(), [&](restaurant& a) {
432             if (a.type == type) {
433                 a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y))
                        ;
434                 temp.push_front(a);
435             }
436         });
437         temp.merge_sort();
438         int cnt = k;
439         for (auto i : temp) {
440             if (--cnt < 0) {
441                 break;
442             }
443             cout << i.name << ' ' << fixed << setprecision(3) << i.dis << endl;
444         }
445     }
```

```
#ifndef ONLINE_JUDGE
    cerr << "time : " << clock() - clk << "ms" << endl;
#endif
}
```

6.2 算法二：使用哈希表

```
#include <bits/stdc++.h>
#define endl '\n'
#define IO ios::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr)
using namespace std;
using ll = long long;
namespace pdli {
template <typename T>
class list {
    struct list_node {
        T entry;
        list_node* next = nullptr;
        list_node() {}
        list_node(const T& item, list_node* add_on = nullptr) : entry(item), next(
            add_on) {}
        friend bool operator==(const list_node& lhs, const list_node& rhs) {
            return lhs.entry == rhs.entry && lhs.next == rhs.next;
        }
    };
    class list_iterator {
    public:
        list_node* _M_node;
        list_iterator() = default;
        ~list_iterator() noexcept = default;
        list_iterator(list_node* node) : _M_node(node) {}
        list_iterator(const list_iterator& other) { _M_node = other._M_node; }
        T& operator*() const { return _M_node->entry; }
        T* operator->() const { return static_cast<T*>(&(_M_node->entry)); }
        list_iterator operator++(int) {
            list_iterator* tmp = this;
            _M_node = _M_node->next;
            return *tmp;
        }
        list_iterator operator++() {
            _M_node = _M_node->next;
            return *this;
        }
```

```
36          friend bool operator==(const list_iterator& lhs, const list_iterator& rhs)
                {
37              return lhs._M_node == rhs._M_node;
38          }
39          friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs)
                {
40              return !(lhs == rhs);
41          }
42          friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs
                ) {
43              list_iterator* ans = lhs;
44              for (size_t i = 0; i < rhs; i++) {
45                  (*ans)++;
46              }
47              return *ans;
48          }
49      };
50
51  public:
52      typedef list::list_iterator iterator;
53      typedef list::list_node node;
54      list() : head(nullptr) {}
55      list(const list& other);
56      ~list() noexcept;
57      size_t size() const;
58      bool empty() const;
59      void push_back(const T& item);
60      void push_front(const T& item);
61      void pop_back();
62      void pop_front();
63      void clear();
64      void reverse();
65      void insert(iterator pos, const T& item);
66      void insert(size_t pos, const T& item);
67      void remove(const T& value);
68      iterator erase(iterator pos);
69      iterator erase(iterator first, iterator last);
70      void erase(size_t pos);
71      void erase(size_t first, size_t last);
72      void replace(const T& old_item, const T& new_item);
73      iterator begin() const;
74      iterator end() const;
75      iterator find(const T& item) const;
76      T& back() const;
```

```
77      T& front() const;
78      list& operator=(const list& other);
79      void insertion_sort();
80      void merge_sort();
81
82  protected:
83      node* head = nullptr;
84
85  private:
86      void _M_merge_sort(node*& list);
87  };
88
89  template <typename T>
90  list<T>::list(const list& other) {
91      if (other.head == nullptr) return;
92      node *cur, *pre;
93      node* _head = other.head;
94      cur = new node(other.head->entry);
95      head = cur;
96      while (_head->next != nullptr) {
97          _head = _head->next;
98          cur->next = new node(_head->entry, _head->next);
99          pre = cur;
100         cur = cur->next;
101     }
102 }
103
104 template <typename T>
105 bool list<T>::empty() const {
106     return head == nullptr;
107 }
108
109 template <typename T>
110 size_t list<T>::size() const {
111     size_t cnt = 0;
112     for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
113     return cnt;
114 }
115
116 template <typename T>
117 void list<T>::push_back(const T& item) {
118     node *cur, *pre;
119     if (head == nullptr) {
120         cur = new node(item);
```

```
121         head = pre = cur;
122         return;
123     }
124     for (cur = head; cur->next != nullptr; cur = cur->next)
125         ;
126     node* new_node = new node(item);
127     cur->next = new_node;
128 }
129 
130 template <typename T>
131 void list<T>::pop_back() {
132     node *cur, *pre;
133     for (cur = head; cur->next != nullptr; cur = cur->next)
134         ;
135     if (cur == head) {
136         delete cur;
137         head = cur = pre = nullptr;
138         return;
139     }
140     for (pre = head; pre->next != cur; pre = pre->next)
141         ;
142     if (cur == nullptr) throw std::underflow_error("underflow");
143     delete cur;
144     pre->next = nullptr;
145     cur = pre;
146     for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
147         ;
148 }
149 
150 template <typename T>
151 T& list<T>::back() const {
152     node* cur;
153     for (cur = head; cur->next != nullptr; cur = cur->next)
154         ;
155     if (cur == nullptr) throw std::underflow_error("underflow");
156     return cur->entry;
157 }
158 
159 template <typename T>
160 void list<T>::clear() {
161     node *cur, *pre;
162     if (head == nullptr) return;
163     pre = head;
164     cur = head->next;
```

```
165     while (cur != nullptr) {
166         delete pre;
167         pre = cur;
168         cur = cur->next;
169     }
170     delete pre;
171     pre = head = cur = nullptr;
172 }
173 
174 template <typename T>
175 list<T>::~list() noexcept {
176     node *cur, *pre;
177     if (head == nullptr) return;
178     pre = head;
179     cur = head->next;
180     while (cur != nullptr) {
181         delete pre;
182         pre = cur;
183         cur = cur->next;
184     }
185     delete pre;
186 }
187 
188 template <typename T>
189 typename list<T>::iterator list<T>::begin() const {
190     return iterator(head);
191 }
192 
193 template <typename T>
194 typename list<T>::iterator list<T>::end() const {
195     return iterator(nullptr);
196 }
197 
198 template <typename T>
199 void list<T>::insert(iterator pos, const T& item) {
200     node *cur, *pre;
201     if (pos._M_node == head) {
202         head = new node(item, head);
203         return;
204     }
205     for (pre = head; pre->next != pos._M_node; pre = pre->next)
206         ;
207     cur = pre->next;
208     pre->next = new node(item, cur);
```

```
209 }
210
211 template <typename T>
212 void list<T>::insert(size_t pos, const T& item) {
213     insert(begin() + pos, item);
214 }
215
216 template <typename T>
217 typename list<T>::iterator list<T>::erase(iterator pos) {
218     node *cur, *pre;
219     iterator res(pos._M_node->next);
220     if (pos._M_node == head) {
221         node* temp = head;
222         head = temp->next;
223         delete temp;
224         return res;
225     }
226     for (pre = head; pre->next != pos._M_node; pre = pre->next)
227         ;
228     cur = pre->next;
229     pre->next = cur->next;
230     if (cur == nullptr) throw std::underflow_error("underflow");
231     delete cur;
232     return res;
233 }
234
235 template <typename T>
236 void list<T>::erase(size_t pos) {
237     erase(begin() + pos);
238 }
239
240 template <typename T>
241 void list<T>::erase(size_t first, size_t last) {
242     erase(begin() + first, begin() + last);
243 }
244
245 template <typename T>
246 void list<T>::reverse() {
247     list<T> temp(*this);
248     clear();
249     while (!temp.empty()) {
250         push_back(temp.back());
251         temp.pop_back();
252     }
```

```
253 }
254
255 template <typename T>
256 list<T>& list<T>::operator=(const list<T>& other) {
257     node *cur, *pre;
258     node* _head = other.head;
259     cur = new node(other.head->entry);
260     head = cur;
261     while (_head->next != nullptr) {
262         _head = _head->next;
263         cur->next = new node(_head->entry, _head->next);
264         pre = cur;
265         cur = cur->next;
266     }
267     return *this;
268 }
269
270 template <typename T>
271 typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
272     for (auto it = first; it != last; it = erase(it))
273         ;
274     return last;
275 }
276
277 template <typename T>
278 void list<T>::push_front(const T& item) {
279     insert(begin(), item);
280 }
281
282 template <typename T>
283 void list<T>::pop_front() {
284     erase(begin());
285 }
286
287 template <typename T>
288 typename list<T>::iterator list<T>::find(const T& item) const {
289     for (auto it = begin(); it != end(); ++it) {
290         if (*it == item) return it;
291     }
292     return end();
293 }
294
295 template <typename T>
296 T& list<T>::front() const {
```

```
297     if (head == nullptr) throw std::underflow_error("underflow");
298     return head->entry;
299 }
300 
301 template <typename T>
302 void list<T>::remove(const T& value) {
303     for (auto it = begin(); it != end();) {
304         if (*it == value) {
305             it = erase(it);
306         } else {
307             it++;
308         }
309     }
310 }
311 
312 template <typename T>
313 void list<T>::replace(const T& old_item, const T& new_item) {
314     for (auto& i : *this) {
315         if (i == old_item) {
316             i = new_item;
317         }
318     }
319 }
320 
321 template <typename T>
322 void list<T>::insertion_sort() {
323     node *last_sorted, *first_unsorted;
324     if (head == nullptr) return;
325     last_sorted = head;
326     while (last_sorted->next != nullptr) {
327         first_unsorted = last_sorted->next;
328         node* pre = head;
329         bool flag = false;
330         for (node* cur = head; cur != first_unsorted; cur = cur->next) {
331             if (cur == head) {
332                 if (first_unsorted->entry <= cur->entry) {
333                     last_sorted->next = first_unsorted->next;
334                     first_unsorted->next = cur;
335                     head = first_unsorted;
336                     flag = true;
337                     break;
338                 }
339             } else {
340                 if (first_unsorted->entry > pre->entry
```

```
341                     && first_unsorted->entry <= cur->entry) {
342                     last_sorted->next = first_unsorted->next;
343                     pre->next = first_unsorted;
344                     first_unsorted->next = cur;
345                     flag = true;
346                     break;
347                 }
348             }
349             pre = cur;
350         }
351         if (!flag) last_sorted = last_sorted->next;
352     }
353 }
354
355 template <typename T>
356 void list<T>::merge_sort() {
357     _M_merge_sort(head);
358 }
359
360 template <typename T>
361 void list<T>::_M_merge_sort(node*& list) {
362     if (list == nullptr || list->next == nullptr) return;
363     // divide
364     node *mid = list, *sec = list, *pos = list;
365     while (pos != nullptr && pos->next != nullptr) {
366         mid = sec;
367         sec = sec->next;
368         pos = pos->next->next;
369     }
370     mid->next = nullptr;
371     // sort
372     _M_merge_sort(list);
373     _M_merge_sort(sec);
374     // merge
375     node* cur = new node;
376     node* new_head = cur;
377     node* fst = list;
378     while (fst != nullptr && sec != nullptr) {
379         if (fst->entry < sec->entry) {
380             cur->next = fst;
381             fst = fst->next;
382             cur = cur->next;
383         } else {
384             cur->next = sec;
```

```
385             sec = sec->next;
386             cur = cur->next;
387         }
388     }
389     if (fst == nullptr) {
390         cur->next = sec;
391     } else {
392         cur->next = fst;
393     }
394     list = new_head->next;
395     delete new_head;
396 }
397 
398 template <typename T, typename U>
399 class hash_map {
400     struct hash_map_record {
401         T key;
402         U value = U();
403         hash_map_record(const T& key) : key(key){};
404         hash_map_record(const T& key, const U& value) : key(key), value(value){};
405         bool operator==(const hash_map_record& other) const {
406             return key == other.key;
407         }
408     };
409     struct hash_map_iterator {
410         list<hash_map_record>* _M_node;
411         typename list<hash_map_record>::iterator _M_list_iterator;
412         hash_map_iterator() = default;
413         hash_map_iterator(list<hash_map_record>* node, const typename list<
                hash_map_record>::iterator& it) : _M_node(node), _M_list_iterator(it){};
414         hash_map_iterator operator++(int) {
415             hash_map_iterator tmp = *this;
416             do {
417                 if (_M_list_iterator != _M_node->end()) {
418                     ++_M_list_iterator;
419                 } else {
420                     ++_M_node;
421                     _M_list_iterator = _M_node->begin();
422                 }
423             } while (_M_list_iterator == nullptr);
424             return tmp;
425         }
426         hash_map_iterator& operator++() {
427             do {
```

```
428                 if (_M_list_iterator != _M_node->end()) {
429                     ++_M_list_iterator;
430                 } else {
431                     ++_M_node;
432                     _M_list_iterator = _M_node->begin();
433                 }
434             } while (_M_list_iterator == nullptr);
435             return *this;
436         }
437         bool operator==(const hash_map_iterator& other) {
438             return _M_node == other._M_node && _M_list_iterator == _M_list_iterator
                    ;
439         }
440         bool operator!=(const hash_map_iterator& other) {
441             return !(*this == other);
442         }
443         hash_map_record& operator*() noexcept {
444             return *_M_list_iterator;
445         }
446     };
447 
448 public:
449     typedef hash_map_record record;
450     typedef hash_map_iterator iterator;
451     static const size_t npos = static_cast<size_t>(-1);
452     hash_map();
453     hash_map(size_t n);
454     hash_map(const hash_map& other);
455     ~hash_map();
456     hash_map& operator=(const hash_map& other);
457     void insert(const T& key, const U& value);
458     U& operator[](const T& key) const;
459     size_t get_position(const T& key) const;
460     U& get_value(const T& key) const;
461     iterator get_iterator(const T& key) const;
462     void remove(const T& key);
463     static size_t _hash_fun(const T& key, const size_t& size);
464     iterator begin() const;
465     iterator end() const;
466 
467 protected:
468     size_t hash_size = 100;
469     list<record>* table;
470 };
```

```
471
472 template <typename T, typename U>
473 hash_map<T, U>::hash_map() {
474     table = new list<record>[hash_size + 1];
475 }
476
477 template <typename T, typename U>
478 hash_map<T, U>::~hash_map() {
479     delete[] table;
480 }
481
482 template <typename T, typename U>
483 hash_map<T, U>::hash_map(size_t n) {
484     hash_size = n;
485     table = new list<record>[hash_size + 1];
486 }
487
488 template <typename T, typename U>
489 hash_map<T, U>::hash_map(const hash_map& other) {
490     hash_size = other.hash_size;
491     table = new list<record>[hash_size + 1];
492     for (int i = 0; i < hash_size; ++i) {
493         table[i] = other.table[i];
494     }
495 }
496
497 template <typename T, typename U>
498 hash_map<T, U>& hash_map<T, U>::operator=(const hash_map<T, U>& other) {
499     if (this == &other) return *this;
500     delete[] table;
501     hash_size = other.hash_size;
502     table = new list<record>[hash_size + 1];
503     for (int i = 0; i < hash_size; ++i) {
504         table[i] = other.table[i];
505     }
506     return *this;
507 }
508
509 template <typename T, typename U>
510 void hash_map<T, U>::insert(const T& key, const U& value) {
511     size_t pos = _hash_fun(key, hash_size);
512     if (table[pos].find(key) == table[pos].end()) {
513         table[pos].push_front({key, value});
514     } else {
```

```
515         table[pos].replace(key, {key, value});
516     }
517 }
518 
519 template <typename T, typename U>
520 size_t hash_map<T, U>::get_position(const T& key) const {
521     size_t pos = _hash_fun(key, hash_size);
522     if (table[pos].find(key) == table[pos].end()) {
523         return npos;
524     }
525     return pos;
526 }
527 
528 template <typename T, typename U>
529 U& hash_map<T, U>::get_value(const T& key) const {
530     size_t pos = _hash_fun(key, hash_size);
531     if (table[pos].find(key) != table[pos].end()) {
532         return table[pos].find(key)->value;
533     } else {
534         throw std::runtime_error("Value of the key does not exist");
535     }
536 }
537 
538 template <typename T, typename U>
539 typename hash_map<T, U>::iterator hash_map<T, U>::get_iterator(const T& key) const
        {
540     size_t pos = _hash_fun(key, hash_size);
541     if (table[pos].find(key) != table[pos].end()) {
542         return iterator(table + pos, table[pos].find(key));
543     } else {
544         return iterator(table + pos, table[pos].begin());
545     }
546 }
547 
548 template <typename T, typename U>
549 void hash_map<T, U>::remove(const T& key) {
550     size_t pos = _hash_fun(key, hash_size);
551     table[pos].remove(key);
552 }
553 
554 template <typename T, typename U>
555 U& hash_map<T, U>::operator[](const T& key) const {
556     size_t pos = _hash_fun(key, hash_size);
557     if (table[pos].find(key) != table[pos].end()) {
```

```
558         return table[pos].find(key)->value;
559     } else {
560         table[pos].push_front(key);
561         return table[pos].begin()->value;
562     }
563 }
564 
565 template <typename T, typename U>
566 typename hash_map<T, U>::iterator hash_map<T, U>::begin() const {
567     return iterator(table, table->begin());
568 }
569 
570 template <typename T, typename U>
571 typename hash_map<T, U>::iterator hash_map<T, U>::end() const {
572     return iterator(table + hash_size + 1, (table + hash_size + 1)->end());
573 }
574 
575 template <typename T, typename U>
576 size_t hash_map<T, U>::_hash_fun(const T& key, const size_t& size) {
577     unsigned seed = 31;
578     unsigned hash = 0;
579     T tmp = key;
580     if (std::is_same<T, std::string>::value) {
581         for (const auto& i : key) {
582             hash = (hash * seed + i) % size;
583         }
584         return hash % size;
585     }
586     return hash % size;
587 }
588 
589 } // namespace pdli
590 
591 struct restaurant {
592     string name;
593     int x;
594     int y;
595     double dis;
596     restaurant() {}
597     restaurant(string name, int x, int y) : name(name), x(x), y(y) {}
598     bool operator<(const restaurant& other) const {
599         return tie(dis, name) < tie(other.dis, other.name);
600     }
601 };
```

```
602
603 int main() {
604     IO;
605     auto clk = clock();
606     int m, n;
607     cin >> m >> n;
608     pdli::hash_map<string, pdli::list<restaurant>> mp;
609     while (m--) {
610         string name, type;
611         int x, y;
612         cin >> name >> x >> y >> type;
613         auto t = restaurant{name, x, y};
614         mp[type].push_front(t);
615     }
616     while (n--) {
617         int x, y, k;
618         string type;
619         cin >> x >> y >> type >> k;
620         for_each(mp[type].begin(), mp[type].end(), [&](restaurant& a) {
621             a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
622         });
623         mp[type].merge_sort();
624         int cnt = k;
625         for (auto i : mp[type]) {
626             if (--cnt < 0) {
627                 break;
628             }
629             cout << i.name << ' ' << fixed << setprecision(3) << i.dis << endl;
630         }
631     }
632 #ifndef ONLINE_JUDGE
633     cerr << "time : " << clock() - clk << "ms" << endl;
634 #endif
635 }
```

6.3 算法三：使用哈希表和 AVL 树

```
1 #include <bits/stdc++.h>
2 #define endl '\n'
3 #define IO ios::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr)
4 using namespace std;
5 using ll = long long;
6 namespace pdli {
```

```
7  template <typename T>
8  class list {
9      struct list_node {
10         T entry;
11         list_node* next = nullptr;
12         list_node() {}
13         list_node(const T& item, list_node* add_on = nullptr) : entry(item), next(
               add_on) {}
14         friend bool operator==(const list_node& lhs, const list_node& rhs) {
15             return lhs.entry == rhs.entry && lhs.next == rhs.next;
16         }
17     };
18     class list_iterator {
19     public:
20         list_node* _M_node;
21         list_iterator() = default;
22         ~list_iterator() noexcept = default;
23         list_iterator(list_node* node) : _M_node(node) {}
24         list_iterator(const list_iterator& other) { _M_node = other._M_node; }
25         T& operator*() const { return _M_node->entry; }
26         T* operator->() const { return static_cast<T*>(&(_M_node->entry)); }
27         list_iterator operator++(int) {
28             list_iterator* tmp = this;
29             _M_node = _M_node->next;
30             return *tmp;
31         }
32         list_iterator operator++() {
33             _M_node = _M_node->next;
34             return *this;
35         }
36         friend bool operator==(const list_iterator& lhs, const list_iterator& rhs)
               {
37             return lhs._M_node == rhs._M_node;
38         }
39         friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs)
               {
40             return !(lhs == rhs);
41         }
42         friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs
               ) {
43             list_iterator* ans = lhs;
44             for (size_t i = 0; i < rhs; i++) {
45                 (*ans)++;
46             }
```

```
47              return *ans;
48          }
49      };
50
51  public:
52      typedef list::list_iterator iterator;
53      typedef list::list_node node;
54      list() : head(nullptr) {}
55      list(const list& other);
56      ~list() noexcept;
57      size_t size() const;
58      bool empty() const;
59      void push_back(const T& item);
60      void push_front(const T& item);
61      void pop_back();
62      void pop_front();
63      void clear();
64      void reverse();
65      void insert(iterator pos, const T& item);
66      void insert(size_t pos, const T& item);
67      void remove(const T& value);
68      iterator erase(iterator pos);
69      iterator erase(iterator first, iterator last);
70      void erase(size_t pos);
71      void erase(size_t first, size_t last);
72      void replace(const T& old_item, const T& new_item);
73      iterator begin() const;
74      iterator end() const;
75      iterator find(const T& item) const;
76      T& back() const;
77      T& front() const;
78      list& operator=(const list& other);
79      void insertion_sort();
80      void merge_sort();
81
82  protected:
83      node* head = nullptr;
84
85  private:
86      void _M_merge_sort(node*& list);
87  };
88
89  template <typename T>
90  list<T>::list(const list& other) {
```

```
91      if (other.head == nullptr) return;
92      node *cur, *pre;
93      node* _head = other.head;
94      cur = new node(other.head->entry);
95      head = cur;
96      while (_head->next != nullptr) {
97          _head = _head->next;
98          cur->next = new node(_head->entry, _head->next);
99          pre = cur;
100         cur = cur->next;
101     }
102 }
103
104 template <typename T>
105 bool list<T>::empty() const {
106     return head == nullptr;
107 }
108
109 template <typename T>
110 size_t list<T>::size() const {
111     size_t cnt = 0;
112     for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
113     return cnt;
114 }
115
116 template <typename T>
117 void list<T>::push_back(const T& item) {
118     node *cur, *pre;
119     if (head == nullptr) {
120         cur = new node(item);
121         head = pre = cur;
122         return;
123     }
124     for (cur = head; cur->next != nullptr; cur = cur->next)
125         ;
126     node* new_node = new node(item);
127     cur->next = new_node;
128 }
129
130 template <typename T>
131 void list<T>::pop_back() {
132     node *cur, *pre;
133     for (cur = head; cur->next != nullptr; cur = cur->next)
134         ;
```

```
135     if (cur == head) {
136         delete cur;
137         head = cur = pre = nullptr;
138         return;
139     }
140     for (pre = head; pre->next != cur; pre = pre->next)
141         ;
142     if (cur == nullptr) throw std::underflow_error("underflow");
143     delete cur;
144     pre->next = nullptr;
145     cur = pre;
146     for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
147         ;
148 }
149 
150 template <typename T>
151 T& list<T>::back() const {
152     node* cur;
153     for (cur = head; cur->next != nullptr; cur = cur->next)
154         ;
155     if (cur == nullptr) throw std::underflow_error("underflow");
156     return cur->entry;
157 }
158 
159 template <typename T>
160 void list<T>::clear() {
161     node *cur, *pre;
162     if (head == nullptr) return;
163     pre = head;
164     cur = head->next;
165     while (cur != nullptr) {
166         delete pre;
167         pre = cur;
168         cur = cur->next;
169     }
170     delete pre;
171     pre = head = cur = nullptr;
172 }
173 
174 template <typename T>
175 list<T>::~list() noexcept {
176     node *cur, *pre;
177     if (head == nullptr) return;
178     pre = head;
```

```
179     cur = head->next;
180     while (cur != nullptr) {
181         delete pre;
182         pre = cur;
183         cur = cur->next;
184     }
185     delete pre;
186 }
187 
188 template <typename T>
189 typename list<T>::iterator list<T>::begin() const {
190     return iterator(head);
191 }
192 
193 template <typename T>
194 typename list<T>::iterator list<T>::end() const {
195     return iterator(nullptr);
196 }
197 
198 template <typename T>
199 void list<T>::insert(iterator pos, const T& item) {
200     node *cur, *pre;
201     if (pos._M_node == head) {
202         head = new node(item, head);
203         return;
204     }
205     for (pre = head; pre->next != pos._M_node; pre = pre->next)
206         ;
207     cur = pre->next;
208     pre->next = new node(item, cur);
209 }
210 
211 template <typename T>
212 void list<T>::insert(size_t pos, const T& item) {
213     insert(begin() + pos, item);
214 }
215 
216 template <typename T>
217 typename list<T>::iterator list<T>::erase(iterator pos) {
218     node *cur, *pre;
219     iterator res(pos._M_node->next);
220     if (pos._M_node == head) {
221         node* temp = head;
222         head = temp->next;
```

```
223         delete temp;
224         return res;
225     }
226     for (pre = head; pre->next != pos._M_node; pre = pre->next)
227         ;
228     cur = pre->next;
229     pre->next = cur->next;
230     if (cur == nullptr) throw std::underflow_error("underflow");
231     delete cur;
232     return res;
233 }
234 
235 template <typename T>
236 void list<T>::erase(size_t pos) {
237     erase(begin() + pos);
238 }
239 
240 template <typename T>
241 void list<T>::erase(size_t first, size_t last) {
242     erase(begin() + first, begin() + last);
243 }
244 
245 template <typename T>
246 void list<T>::reverse() {
247     list<T> temp(*this);
248     clear();
249     while (!temp.empty()) {
250         push_back(temp.back());
251         temp.pop_back();
252     }
253 }
254 
255 template <typename T>
256 list<T>& list<T>::operator=(const list<T>& other) {
257     node *cur, *pre;
258     node* _head = other.head;
259     cur = new node(other.head->entry);
260     head = cur;
261     while (_head->next != nullptr) {
262         _head = _head->next;
263         cur->next = new node(_head->entry, _head->next);
264         pre = cur;
265         cur = cur->next;
266     }
```

```
267     return *this;
268 }
269
270 template <typename T>
271 typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
272     for (auto it = first; it != last; it = erase(it))
273         ;
274     return last;
275 }
276
277 template <typename T>
278 void list<T>::push_front(const T& item) {
279     insert(begin(), item);
280 }
281
282 template <typename T>
283 void list<T>::pop_front() {
284     erase(begin());
285 }
286
287 template <typename T>
288 typename list<T>::iterator list<T>::find(const T& item) const {
289     for (auto it = begin(); it != end(); ++it) {
290         if (*it == item) return it;
291     }
292     return end();
293 }
294
295 template <typename T>
296 T& list<T>::front() const {
297     if (head == nullptr) throw std::underflow_error("underflow");
298     return head->entry;
299 }
300
301 template <typename T>
302 void list<T>::remove(const T& value) {
303     for (auto it = begin(); it != end();) {
304         if (*it == value) {
305             it = erase(it);
306         } else {
307             it++;
308         }
309     }
310 }
```

```
311
312 template <typename T>
313 void list<T>::replace(const T& old_item, const T& new_item) {
314     for (auto& i : *this) {
315         if (i == old_item) {
316             i = new_item;
317         }
318     }
319 }
320
321 template <typename T>
322 void list<T>::insertion_sort() {
323     node *last_sorted, *first_unsorted;
324     if (head == nullptr) return;
325     last_sorted = head;
326     while (last_sorted->next != nullptr) {
327         first_unsorted = last_sorted->next;
328         node* pre = head;
329         bool flag = false;
330         for (node* cur = head; cur != first_unsorted; cur = cur->next) {
331             if (cur == head) {
332                 if (first_unsorted->entry <= cur->entry) {
333                     last_sorted->next = first_unsorted->next;
334                     first_unsorted->next = cur;
335                     head = first_unsorted;
336                     flag = true;
337                     break;
338                 }
339             } else {
340                 if (first_unsorted->entry > pre->entry
341                     && first_unsorted->entry <= cur->entry) {
342                     last_sorted->next = first_unsorted->next;
343                     pre->next = first_unsorted;
344                     first_unsorted->next = cur;
345                     flag = true;
346                     break;
347                 }
348             }
349             pre = cur;
350         }
351         if (!flag) last_sorted = last_sorted->next;
352     }
353 }
354
```

```
355 template <typename T>
356 void list<T>::merge_sort() {
357     _M_merge_sort(head);
358 }
359 
360 template <typename T>
361 void list<T>::_M_merge_sort(node*& list) {
362     if (list == nullptr || list->next == nullptr) return;
363     // divide
364     node *mid = list, *sec = list, *pos = list;
365     while (pos != nullptr && pos->next != nullptr) {
366         mid = sec;
367         sec = sec->next;
368         pos = pos->next->next;
369     }
370     mid->next = nullptr;
371     // sort
372     _M_merge_sort(list);
373     _M_merge_sort(sec);
374     // merge
375     node* cur = new node;
376     node* new_head = cur;
377     node* fst = list;
378     while (fst != nullptr && sec != nullptr) {
379         if (fst->entry < sec->entry) {
380             cur->next = fst;
381             fst = fst->next;
382             cur = cur->next;
383         } else {
384             cur->next = sec;
385             sec = sec->next;
386             cur = cur->next;
387         }
388     }
389     if (fst == nullptr) {
390         cur->next = sec;
391     } else {
392         cur->next = fst;
393     }
394     list = new_head->next;
395     delete new_head;
396 }
397 
398 template <typename T, typename U>
```

```
399 class hash_map {
400     struct hash_map_record {
401         T key;
402         U value = U();
403         hash_map_record(const T& key) : key(key){};
404         hash_map_record(const T& key, const U& value) : key(key), value(value){};
405         bool operator==(const hash_map_record& other) const {
406             return key == other.key;
407         }
408     };
409     struct hash_map_iterator {
410         list<hash_map_record>* _M_node;
411         typename list<hash_map_record>::iterator _M_list_iterator;
412         hash_map_iterator() = default;
413         hash_map_iterator(list<hash_map_record>* node, const typename list<
                hash_map_record>::iterator& it) : _M_node(node), _M_list_iterator(it){};
414         hash_map_iterator operator++(int) {
415             hash_map_iterator tmp = *this;
416             do {
417                 if (_M_list_iterator != _M_node->end()) {
418                     ++_M_list_iterator;
419                 } else {
420                     ++_M_node;
421                     _M_list_iterator = _M_node->begin();
422                 }
423             } while (_M_list_iterator == nullptr);
424             return tmp;
425         }
426         hash_map_iterator& operator++() {
427             do {
428                 if (_M_list_iterator != _M_node->end()) {
429                     ++_M_list_iterator;
430                 } else {
431                     ++_M_node;
432                     _M_list_iterator = _M_node->begin();
433                 }
434             } while (_M_list_iterator == nullptr);
435             return *this;
436         }
437         bool operator==(const hash_map_iterator& other) {
438             return _M_node == other._M_node && _M_list_iterator == _M_list_iterator
                    ;
439         }
440         bool operator!=(const hash_map_iterator& other) {
```

```
441             return !(*this == other);
442         }
443         hash_map_record& operator*() noexcept {
444             return *_M_list_iterator;
445         }
446     };
447 
448 public:
449     typedef hash_map_record record;
450     typedef hash_map_iterator iterator;
451     static const size_t npos = static_cast<size_t>(-1);
452     hash_map();
453     hash_map(size_t n);
454     hash_map(const hash_map& other);
455     ~hash_map();
456     hash_map& operator=(const hash_map& other);
457     void insert(const T& key, const U& value);
458     U& operator[](const T& key) const;
459     size_t get_position(const T& key) const;
460     U& get_value(const T& key) const;
461     iterator get_iterator(const T& key) const;
462     void remove(const T& key);
463     static size_t _hash_fun(const T& key, const size_t& size);
464     iterator begin() const;
465     iterator end() const;
466 
467 protected:
468     size_t hash_size = 100;
469     list<record>* table;
470 };
471 
472 template <typename T, typename U>
473 hash_map<T, U>::hash_map() {
474     table = new list<record>[hash_size + 1];
475 }
476 
477 template <typename T, typename U>
478 hash_map<T, U>::~hash_map() {
479     delete[] table;
480 }
481 
482 template <typename T, typename U>
483 hash_map<T, U>::hash_map(size_t n) {
484     hash_size = n;
```

```
485     table = new list<record>[hash_size + 1];
486 }
487
488 template <typename T, typename U>
489 hash_map<T, U>::hash_map(const hash_map& other) {
490     hash_size = other.hash_size;
491     table = new list<record>[hash_size + 1];
492     for (int i = 0; i < hash_size; ++i) {
493         table[i] = other.table[i];
494     }
495 }
496
497 template <typename T, typename U>
498 hash_map<T, U>& hash_map<T, U>::operator=(const hash_map<T, U>& other) {
499     if (this == &other) return *this;
500     delete[] table;
501     hash_size = other.hash_size;
502     table = new list<record>[hash_size + 1];
503     for (int i = 0; i < hash_size; ++i) {
504         table[i] = other.table[i];
505     }
506     return *this;
507 }
508
509 template <typename T, typename U>
510 void hash_map<T, U>::insert(const T& key, const U& value) {
511     size_t pos = _hash_fun(key, hash_size);
512     if (table[pos].find(key) == table[pos].end()) {
513         table[pos].push_front({key, value});
514     } else {
515         table[pos].replace(key, {key, value});
516     }
517 }
518
519 template <typename T, typename U>
520 size_t hash_map<T, U>::get_position(const T& key) const {
521     size_t pos = _hash_fun(key, hash_size);
522     if (table[pos].find(key) == table[pos].end()) {
523         return npos;
524     }
525     return pos;
526 }
527
528 template <typename T, typename U>
```

```
529 U& hash_map<T, U>::get_value(const T& key) const {
530     size_t pos = _hash_fun(key, hash_size);
531     if (table[pos].find(key) != table[pos].end()) {
532         return table[pos].find(key)->value;
533     } else {
534         throw std::runtime_error("Value of the key does not exist");
535     }
536 }
537
538 template <typename T, typename U>
539 typename hash_map<T, U>::iterator hash_map<T, U>::get_iterator(const T& key) const
        {
540     size_t pos = _hash_fun(key, hash_size);
541     if (table[pos].find(key) != table[pos].end()) {
542         return iterator(table + pos, table[pos].find(key));
543     } else {
544         return iterator(table + pos, table[pos].begin());
545     }
546 }
547
548 template <typename T, typename U>
549 void hash_map<T, U>::remove(const T& key) {
550     size_t pos = _hash_fun(key, hash_size);
551     table[pos].remove(key);
552 }
553
554 template <typename T, typename U>
555 U& hash_map<T, U>::operator[](const T& key) const {
556     size_t pos = _hash_fun(key, hash_size);
557     if (table[pos].find(key) != table[pos].end()) {
558         return table[pos].find(key)->value;
559     } else {
560         table[pos].push_front(key);
561         return table[pos].begin()->value;
562     }
563 }
564
565 template <typename T, typename U>
566 typename hash_map<T, U>::iterator hash_map<T, U>::begin() const {
567     return iterator(table, table->begin());
568 }
569
570 template <typename T, typename U>
571 typename hash_map<T, U>::iterator hash_map<T, U>::end() const {
```

```
572     return iterator(table + hash_size + 1, (table + hash_size + 1)->end());
573 }
574
575 template <typename T, typename U>
576 size_t hash_map<T, U>::_hash_fun(const T& key, const size_t& size) {
577     unsigned seed = 31;
578     unsigned hash = 0;
579     T tmp = key;
580     if (is_same<T, std::string>::value) {
581         for (const auto& i : key) {
582             hash = (hash * seed + i) % size;
583         }
584         return hash % size;
585     }
586     return hash % size;
587 }
588
589 template <typename T>
590 class AVL_tree {
591     enum class balance_factor {
592         left_higher,
593         right_higher,
594         equal_height
595     };
596     struct AVL_tree_node {
597         T data;
598         balance_factor balance = balance_factor::equal_height;
599         AVL_tree_node* left = nullptr;
600         AVL_tree_node* right = nullptr;
601         AVL_tree_node() = default;
602         AVL_tree_node(const T& item) : data(item) {}
603     };
604
605 public:
606     using node = AVL_tree_node;
607
608     AVL_tree() = default;
609
610     AVL_tree(const AVL_tree& other) {
611         clear();
612         std::function<void(node*)> m_insert = [&](node* sub_root) {
613             if (sub_root == nullptr) return;
614             insert(sub_root->data);
615             m_insert(sub_root->left);
```

```
616             m_insert(sub_root->right);
617         };
618         m_insert(other.root);
619     }
620 
621     AVL_tree(AVL_tree&& other) {
622         clear();
623         root = std::move(other.root);
624         other.root = nullptr;
625     }
626 
627     ~AVL_tree() {
628         clear();
629     }
630 
631     AVL_tree& operator=(const AVL_tree& other) {
632         if (this == &other) return *this;
633         clear();
634         std::function<void(node*)> m_insert = [&](node* sub_root) {
635             if (sub_root == nullptr) return;
636             insert(sub_root->data);
637             m_insert(sub_root->left);
638             m_insert(sub_root->right);
639         };
640         m_insert(other.root);
641         return *this;
642     }
643 
644     AVL_tree& operator=(AVL_tree&& other) {
645         if (this == &other) return *this;
646         clear();
647         root = std::move(other.root);
648         other.root = nullptr;
649         return *this;
650     }
651 
652     /**
653              * @brief Insert an item at the AVL_tree. If the item is already in the
                   tree, throw a duplicate_errer.
654              * @param item the item to insert
655              */
656     void insert(const T& item) {
657         AVL_insert(root, item);
658     }
```

```
659
660     /**
661              * @brief Clear the tree.
662              */
663     void clear() {
664         m_clear(root);
665         root = nullptr;
666     }
667
668     /**
669              * @brief Remove an item in the tree
670              * @param item the item to remove
671              */
672     void remove(const T& item) {
673         AVL_remove(root, item);
674     }
675
676     /**
677              * @brief Find an item in the tree.
678              * @param item the item to find
679              * @return reference of the item found in the tree
680              */
681     T& find(const T& item) {
682         node* cur = root;
683         while (cur != nullptr && cur->data != item) {
684             std::cout << cur->data.key << ' ';
685             if (item < cur->data) {
686                 cur = cur->left;
687             } else {
688                 cur = cur->right;
689             }
690         }
691         if (cur == nullptr) {
692             throw std::runtime_error("not present");
693         } else {
694             return cur->data;
695         }
696     }
697
698     void in_order(const std::function<void(T&)>& fun) {
699         std::function<void(node*, std::function<void(T&)>)> _inorder = [&](node*
            root, std::function<void(T&)> fun) {
700             if (root == nullptr) return;
701             _inorder(root->left, fun);
```

```
702             fun(root->data);
703             _inorder(root->right, fun);
704         };
705         _inorder(root, fun);
706     }
707 
708     void pre_order(const std::function<void(T&)>& fun) {
709         std::function<void(node*, std::function<void(T&)>)> _preorder = [&](node*
                root, std::function<void(T&)> fun) {
710             if (root == nullptr) return;
711             fun(root->data);
712             _preorder(root->left, fun);
713             _preorder(root->right, fun);
714         };
715         _preorder(root, fun);
716     }
717 
718     void post_order(const std::function<void(T&)>& fun) {
719         std::function<void(node*, std::function<void(T&)>)> _postorder = [&](node*
                root, std::function<void(T&)> fun) {
720             if (root == nullptr) return;
721             _postorder(root->left, fun);
722             _postorder(root->right, fun);
723             fun(root->data);
724         };
725         _postorder(root, fun);
726     }
727 
728 protected:
729     node* root = nullptr;
730 
731 private:
732     /**
733              * @brief Insert an item in the subtree. If the item is already in the
                    subtree, throw a duplicate_errer.
734              * @param subroot where to insert
735              * @param item the item to insert
736              * @return the subtree is increased in height or not
737             */
738     bool AVL_insert(node*& sub_root, const T& item) {
739         if (sub_root == nullptr) {
740             sub_root = new node(item);
741             return true;
742         } else if (item == sub_root->data) {
```

```
743             throw std::runtime_error("duplicate error");
744         } else if (item < sub_root->data) {
745             bool taller = AVL_insert(sub_root->left, item);
746             if (taller) {
747                 switch (sub_root->balance) {
748                 case balance_factor::left_higher:
749                     left_balance(sub_root);
750                     taller = false;
751                     break;
752                 case balance_factor::right_higher:
753                     sub_root->balance = balance_factor::equal_height;
754                     taller = false;
755                     break;
756                 case balance_factor::equal_height:
757                     sub_root->balance = balance_factor::left_higher;
758                     break;
759                 }
760             }
761             return taller;
762         } else {
763             bool is_taller = AVL_insert(sub_root->right, item);
764             if (is_taller) {
765                 switch (sub_root->balance) {
766                 case balance_factor::left_higher:
767                     sub_root->balance = balance_factor::equal_height;
768                     is_taller = false;
769                     break;
770                 case balance_factor::right_higher:
771                     right_balance(sub_root);
772                     is_taller = false;
773                     break;
774                 case balance_factor::equal_height:
775                     sub_root->balance = balance_factor::right_higher;
776                     break;
777                 }
778             }
779             return is_taller;
780         }
781     }
782
783     /**
784              * @brief Make a left subtree balanced
785              * @param sub_root root of the left subtree
786              */
```

```
787     void left_balance(node*& sub_root) {
788         node*& left_tree = sub_root->left;
789         switch (left_tree->balance) {
790         case balance_factor::left_higher: // case L-L
791             sub_root->balance = balance_factor::equal_height;
792             left_tree->balance = balance_factor::equal_height;
793             rotate_right(sub_root);
794             break;
795         case balance_factor::equal_height:
796             throw std::runtime_error("impossible case in left_balance");
797             break;
798         case balance_factor::right_higher: // case L-R
799             node* sub_tree = left_tree->right;
800             switch (sub_tree->balance) {
801             case balance_factor::equal_height:
802                 sub_root->balance = balance_factor::equal_height;
803                 left_tree->balance = balance_factor::equal_height;
804                 break;
805             case balance_factor::left_higher:
806                 sub_root->balance = balance_factor::right_higher;
807                 left_tree->balance = balance_factor::equal_height;
808                 break;
809             case balance_factor::right_higher:
810                 sub_root->balance = balance_factor::equal_height;
811                 left_tree->balance = balance_factor::left_higher;
812                 break;
813             }
814             sub_tree->balance = balance_factor::equal_height;
815             rotate_left(left_tree);
816             rotate_right(sub_root);
817             break;
818         }
819     }
820
821     /**
822              * @brief Make a right subtree balanced
823              * @param sub_root root of the right subtree
824              */
825     void right_balance(node*& sub_root) {
826         node*& right_tree = sub_root->right;
827         switch (right_tree->balance) {
828         case balance_factor::right_higher: // case R-R
829             sub_root->balance = balance_factor::equal_height;
830             right_tree->balance = balance_factor::equal_height;
```

```
831             rotate_left(sub_root);
832             break;
833         case balance_factor::equal_height:
834             throw std::runtime_error("impossible case in right_balance");
835             break;
836         case balance_factor::left_higher: // case R-L
837             node* sub_tree = right_tree->left;
838             switch (sub_tree->balance) {
839             case balance_factor::equal_height:
840                 sub_root->balance = balance_factor::equal_height;
841                 right_tree->balance = balance_factor::equal_height;
842                 break;
843             case balance_factor::left_higher:
844                 sub_root->balance = balance_factor::equal_height;
845                 right_tree->balance = balance_factor::right_higher;
846                 break;
847             case balance_factor::right_higher:
848                 sub_root->balance = balance_factor::left_higher;
849                 right_tree->balance = balance_factor::equal_height;
850                 break;
851             }
852             sub_tree->balance = balance_factor::equal_height;
853             rotate_right(right_tree);
854             rotate_left(sub_root);
855             break;
856         }
857     }
858
859     /**
860              * @brief Do a left rotation at the sub_tree
861              * @param sub_root the root of the sub_tree
862             */
863     void rotate_left(node*& sub_root) {
864         if (sub_root == nullptr || sub_root->right == nullptr) {
865             throw std::runtime_error("impossible case in rotate_left");
866         }
867         node* right_tree = sub_root->right;
868         sub_root->right = right_tree->left;
869         right_tree->left = sub_root;
870         sub_root = right_tree;
871     }
872
873     /**
874              * @brief Do a right rotation at the sub_tree
```

```
875             * @param sub_root the root of the sub_tree
876             */
877    void rotate_right(node*& sub_root) {
878        if (sub_root == nullptr || sub_root->left == nullptr) {
879            throw std::runtime_error("impossible case in rotate_right");
880        }
881        node* left_tree = sub_root->left;
882        sub_root->left = left_tree->right;
883        left_tree->right = sub_root;
884        sub_root = left_tree;
885    }
886
887    /**
888             * @brief Clear a subtree.
889             * @param sub_root the root of the subtree
890             */
891    void m_clear(node* sub_root) {
892        if (sub_root == nullptr) return;
893        m_clear(sub_root->left);
894        m_clear(sub_root->right);
895        delete sub_root;
896    }
897
898    /**
899             * @brief Remove an item in a subtree.
900             * @param sub_root the root of the subtree
901             * @param item the item to remove
902             * @return the sub_tree becomes shorter or not
903             */
904    bool AVL_remove(node*& sub_root, const T& item) {
905        if (sub_root == nullptr) {
906            throw std::runtime_error("not found in AVL_remove");
907        } else if (item < sub_root->data) {
908            return remove_left(sub_root, item);
909        } else if (item > sub_root->data) {
910            return remove_right(sub_root, item);
911        } else if (sub_root->right == nullptr) {
912            node* temp = sub_root;
913            sub_root = sub_root->left;
914            delete temp;
915            return true;
916        } else if (sub_root->left == nullptr) {
917            node* temp = sub_root;
918            sub_root = sub_root->right;
```

```
919             delete temp;
920             return true;
921         } else if (sub_root->balance == balance_factor::right_higher) {
922             // another solution
923             // node* temp = sub_root->right;
924             // while (temp->left != nullptr) {
925             //     temp = temp->left;
926             // }
927             // return remove_right(sub_root, temp->data);
928             node* temp = sub_root->left;
929             while (temp->right != nullptr) {
930                 temp = temp->right;
931             }
932             sub_root->data = temp->data;
933             return remove_left(sub_root, temp->data);
934         } else {
935             node* temp = sub_root->left;
936             while (temp->right != nullptr) {
937                 temp = temp->right;
938             }
939             sub_root->data = temp->data;
940             return remove_left(sub_root, temp->data);
941         }
942     }
943 
944     /**
945              * @brief Remove an item from a left subtree.
946              * @param sub_root the root of the left subtree
947              * @param item the item to remove
948              * @return the subtree becomes shorter or not
949              */
950     bool remove_left(node*& sub_root, const T& item) {
951         bool shorter = AVL_remove(sub_root->left, item);
952         if (shorter) {
953             switch (sub_root->balance) {
954             case balance_factor::left_higher:
955                 sub_root->balance = balance_factor::equal_height;
956                 break;
957             case balance_factor::equal_height:
958                 sub_root->balance = balance_factor::right_higher;
959                 shorter = false;
960                 break;
961             case balance_factor::right_higher:
962                 node* temp = sub_root->right;
```

```
963                 switch (temp->balance) {
964                 case balance_factor::equal_height:
965                     temp->balance = balance_factor::left_higher;
966                     rotate_left(sub_root);
967                     shorter = false;
968                     break;
969                 case balance_factor::right_higher:
970                     sub_root->balance = balance_factor::equal_height;
971                     temp->balance = balance_factor::equal_height;
972                     rotate_left(sub_root);
973                     break;
974                 case balance_factor::left_higher:
975                     node* temp_left = temp->left;
976                     switch (temp_left->balance) {
977                     case balance_factor::equal_height:
978                         sub_root->balance = balance_factor::equal_height;
979                         temp->balance = balance_factor::equal_height;
980                         break;
981                     case balance_factor::right_higher:
982                         sub_root->balance = balance_factor::left_higher;
983                         temp->balance = balance_factor::equal_height;
984                         break;
985                     case balance_factor::left_higher:
986                         sub_root->balance = balance_factor::equal_height;
987                         temp->balance = balance_factor::right_higher;
988                         break;
989                     }
990                     temp_left->balance = balance_factor::equal_height;
991                     rotate_right(sub_root->right);
992                     rotate_left(sub_root);
993                     break;
994                 }
995             }
996         }
997         return shorter;
998     }
999
1000    /**
1001             * @brief Remove an item from a right subtree.
1002             * @param sub_root the root of the right subtree
1003             * @param item the item to remove
1004             * @return the subtree becomes shorter or not
1005             */
1006    bool remove_right(node*& sub_root, const T& item) {
```

```
1007            bool shorter = AVL_remove(sub_root->right, item);
1008            if (shorter) {
1009                switch (sub_root->balance) {
1010                case balance_factor::right_higher:
1011                    sub_root->balance = balance_factor::equal_height;
1012                    break;
1013                case balance_factor::equal_height:
1014                    sub_root->balance = balance_factor::left_higher;
1015                    shorter = false;
1016                    break;
1017                case balance_factor::left_higher:
1018                    node* temp = sub_root->left;
1019                    switch (temp->balance) {
1020                    case balance_factor::equal_height:
1021                        temp->balance = balance_factor::right_higher;
1022                        rotate_right(sub_root);
1023                        shorter = false;
1024                        break;
1025                    case balance_factor::left_higher:
1026                        sub_root->balance = balance_factor::equal_height;
1027                        temp->balance = balance_factor::equal_height;
1028                        rotate_right(sub_root);
1029                        break;
1030                    case balance_factor::right_higher:
1031                        node* temp_right = temp->right;
1032                        switch (temp_right->balance) {
1033                        case balance_factor::equal_height:
1034                            sub_root->balance = balance_factor::equal_height;
1035                            temp->balance = balance_factor::equal_height;
1036                            break;
1037                        case balance_factor::left_higher:
1038                            sub_root->balance = balance_factor::right_higher;
1039                            temp->balance = balance_factor::equal_height;
1040                            break;
1041                        case balance_factor::right_higher:
1042                            sub_root->balance = balance_factor::equal_height;
1043                            temp->balance = balance_factor::left_higher;
1044                            break;
1045                        }
1046                        temp_right->balance = balance_factor::equal_height;
1047                        rotate_left(sub_root->left);
1048                        rotate_right(sub_root);
1049                        break;
1050                    }
```

```
1051                }
1052            }
1053            return shorter;
1054        }
1055    };
1056
1057    } // namespace pdli
1058
1059    struct restaurant {
1060        string name = "";
1061        int x = 0;
1062        int y = 0;
1063        double dis = 0;
1064        restaurant() {}
1065        restaurant(string name, int x, int y) : name(name), x(x), y(y) {}
1066        bool operator<(const restaurant& other) const {
1067            return tie(dis, name) < tie(other.dis, other.name);
1068        }
1069        bool operator==(const restaurant& other) const {
1070            return tie(name, x, y, dis) == tie(other.name, other.x, other.y, other.dis)
                    ;
1071        }
1072    };
1073
1074    int main() {
1075        IO;
1076        auto clk = clock();
1077        int m, n;
1078        cin >> m >> n;
1079        pdli::hash_map<string, pdli::list<restaurant>> unsorted_map;
1080        pdli::hash_map<string, pdli::AVL_tree<restaurant>> sorted_map;
1081        while (m--) {
1082            string name, type;
1083            int x, y;
1084            cin >> name >> x >> y >> type;
1085            auto t = restaurant{name, x, y};
1086            unsorted_map[type].push_front(t);
1087        }
1088        while (n--) {
1089            int x, y, k;
1090            string type;
1091            cin >> x >> y >> type >> k;
1092            for_each(unsorted_map[type].begin(), unsorted_map[type].end(), [&](
                    restaurant& a) {
```

```
1093            a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
1094            sorted_map[type].insert(a);
1095        });
1096        int cnt = k;
1097        sorted_map[type].in_order([&](const restaurant& a) {
1098            if (--cnt < 0) {
1099                return;
1100            }
1101            cout << a.name << ' ' << fixed << setprecision(3) << a.dis << endl;
1102        });
1103    }
1104 #ifndef ONLINE_JUDGE
1105    cerr << "time : " << clock() - clk << "ms" << endl;
1106 #endif
1107 }
```

6.4 算法四：使用 STL

```
1  #include <bits/stdc++.h>
2  #define endl '\n'
3  #define IO ios::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr)
4  using namespace std;
5  using ll = long long;
6  struct restaurant {
7      string name;
8      int x;
9      int y;
10     double dis = 0;
11 };
12 int main() {
13     IO;
14     auto clk = clock();
15     int m, n;
16     cin >> m >> n;
17     unordered_map<string, vector<restaurant>> mp;
18     while (m--) {
19         string name, type;
20         int x, y;
21         cin >> name >> x >> y >> type;
22         mp[type].emplace_back(std::move(restaurant{name, x, y}));
23     }
24     while (n--) {
25         int x, y, k;
```

```
26          string type;
27          cin >> x >> y >> type >> k;
28          for_each(mp[type].begin(), mp[type].end(), [&](restaurant& a) {
29              a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
30          });
31          sort(mp[type].begin(), mp[type].end(), [&](restaurant& a, restaurant& b) {
32              return tie(a.dis, a.name) < tie(b.dis, b.name);
33          });
34          int cnt = k;
35          for (auto i : mp[type]) {
36              if (--cnt < 0) {
37                  break;
38              }
39              cout << i.name << fixed << setprecision(3) << i.dis << endl;
40          }
41      }
42  #ifndef ONLINE_JUDGE
43      cerr << "time : " << clock() - clk << "ms" << endl;
44  #endif
45  }
```