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;
        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;
        }
        friend bool operator==(const list_iterator& lhs, const list_iterator& rhs) {
            return lhs._M_node == rhs._M_node;
        }
        friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs) {
            return !(lhs == rhs);
        }
        friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs) {
            list_iterator* ans = lhs;
            for (size_t i = 0; i < rhs; i++) {
                (*ans)++;
            }
            return *ans;
        }
    };

```

```
public:
    typedef list::list_iterator iterator;
    typedef list::list_node node;
    list() : head(nullptr) {}
    list(const list& other);
    ~list() noexcept;
    size_t size() const;
    bool empty() const;
    void push_back(const T& item);
    void push_front(const T& item);
    void pop_back();
    void pop_front();
    void clear();
    void reverse();
    void insert(iterator pos, const T& item);
    void insert(size_t pos, const T& item);
    void remove(const T& value);
    iterator erase(iterator pos);
    iterator erase(iterator first, iterator last);
    void erase(size_t pos);
    void erase(size_t first, size_t last);
    void replace(const T& old_item, const T& new_item);
    iterator begin() const;
    iterator end() const;
    iterator find(const T& item) const;
    T& back() const;
    T& front() const;
    list& operator=(const list& other);
    void insertion_sort();
    void merge_sort();

protected:
    node* head = nullptr;

private:
    void _M_merge_sort(node*& list);
};

template <typename T>
list<T>::list(const list& other) {
    if (other.head == nullptr) return;
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
}

template <typename T>
bool list<T>::empty() const {
    return head == nullptr;
}

template <typename T>
size_t list<T>::size() const {
    size_t cnt = 0;
    for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
    return cnt;
}

template <typename T>
void list<T>::push_back(const T& item) {
    node *cur, *pre;
    if (head == nullptr) {
        cur = new node(item);
        head = pre = cur;
        return;
    }
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    node* new_node = new node(item);
    cur->next = new_node;
}

template <typename T>
void list<T>::pop_back() {
    node *cur, *pre;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == head) {
        delete cur;
        head = cur = pre = nullptr;
        return;
    }
    for (pre = head; pre->next != cur; pre = pre->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    pre->next = nullptr;
    cur = pre;
    for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
        ;
}

template <typename T>
T& list<T>::back() const {
    node* cur;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    return cur->entry;
}

template <typename T>
void list<T>::clear() {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
    pre = head = cur = nullptr;
}

template <typename T>
list<T>::~list() noexcept {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
}

template <typename T>
typename list<T>::iterator list<T>::begin() const {
    return iterator(head);
}

template <typename T>
typename list<T>::iterator list<T>::end() const {
    return iterator(nullptr);
}

template <typename T>
void list<T>::insert(iterator pos, const T& item) {
    node *cur, *pre;
    if (pos._M_node == head) {
        head = new node(item, head);
        return;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
        ;
    cur = pre->next;
    pre->next = new node(item, cur);
}

template <typename T>
void list<T>::insert(size_t pos, const T& item) {
    insert(begin() + pos, item);
}

template <typename T>
typename list<T>::iterator list<T>::erase(iterator pos) {
    node *cur, *pre;
    iterator res(pos._M_node->next);
    if (pos._M_node == head) {
        node* temp = head;
        head = temp->next;
        delete temp;
        return res;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
```

```cpp
        ;
    cur = pre->next;
    pre->next = cur->next;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    return res;
}

template <typename T>
void list<T>::erase(size_t pos) {
    erase(begin() + pos);
}

template <typename T>
void list<T>::erase(size_t first, size_t last) {
    erase(begin() + first, begin() + last);
}

template <typename T>
void list<T>::reverse() {
    list<T> temp(*this);
    clear();
    while (!temp.empty()) {
        push_back(temp.back());
        temp.pop_back();
    }
}

template <typename T>
list<T>& list<T>::operator=(const list<T>& other) {
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
    return *this;
}

template <typename T>
```

```
typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
    for (auto it = first; it != last; it = erase(it))
        ;
    return last;
}

template <typename T>
void list<T>::push_front(const T& item) {
    insert(begin(), item);
}

template <typename T>
void list<T>::pop_front() {
    erase(begin());
}

template <typename T>
typename list<T>::iterator list<T>::find(const T& item) const {
    for (auto it = begin(); it != end(); ++it) {
        if (*it == item) return it;
    }
    return end();
}

template <typename T>
T& list<T>::front() const {
    if (head == nullptr) throw std::underflow_error("underflow");
    return head->entry;
}

template <typename T>
void list<T>::remove(const T& value) {
    for (auto it = begin(); it != end();) {
        if (*it == value) {
            it = erase(it);
        } else {
            it++;
        }
    }
}

template <typename T>
void list<T>::replace(const T& old_item, const T& new_item) {
    for (auto& i : *this) {
        if (i == old_item) {
            i = new_item;
        }
    }
}

template <typename T>
void list<T>::insertion_sort() {
    node *last_sorted, *first_unsorted;
    if (head == nullptr) return;
    last_sorted = head;
    while (last_sorted->next != nullptr) {
        first_unsorted = last_sorted->next;
        node* pre = head;
        bool flag = false;
        for (node* cur = head; cur != first_unsorted; cur = cur->next) {
            if (cur == head) {
                if (first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    first_unsorted->next = cur;
                    head = first_unsorted;
                    flag = true;
                    break;
                }
            } else {
                if (first_unsorted->entry > pre->entry
                    && first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    pre->next = first_unsorted;
                    first_unsorted->next = cur;
                    flag = true;
                    break;
                }
            }
            pre = cur;
        }
        if (!flag) last_sorted = last_sorted->next;
    }
}

template <typename T>
void list<T>::merge_sort() {
    _M_merge_sort(head);
}
```

```

template <typename T>
void list<T>::_M_merge_sort(node*& list) {
    if (list == nullptr || list->next == nullptr) return;
    // divide
    node *mid = list, *sec = list, *pos = list;
    while (pos != nullptr && pos->next != nullptr) {
        mid = sec;
        sec = sec->next;
        pos = pos->next->next;
    }
    mid->next = nullptr;
    // sort
    _M_merge_sort(list);
    _M_merge_sort(sec);
    // merge
    node* cur = new node;
    node* new_head = cur;
    node* fst = list;
    while (fst != nullptr && sec != nullptr) {
        if (fst->entry < sec->entry) {
            cur->next = fst;
            fst = fst->next;
            cur = cur->next;
        } else {
            cur->next = sec;
            sec = sec->next;
            cur = cur->next;
        }
    }
    if (fst == nullptr) {
        cur->next = sec;
    } else {
        cur->next = fst;
    }
    list = new_head->next;
    delete new_head;
}
} // namespace pdli

struct restaurant {
    string name;
    string type;
    int x;
    int y;
    double dis;
    restaurant() {}
    restaurant(const string& name, const string& type, int x, int y)
        : name(name), type(type), x(x), y(y), dis(0) {}
    bool operator<(const restaurant& other) const {
        return tie(dis, name) < tie(other.dis, other.name);
    }
};

int main() {
    IO;
    auto clk = clock();
    int m, n;
    cin >> m >> n;
    pdli::list<restaurant> ls;
    while (m--) {
        string name, type;
        int x, y;
        cin >> name >> x >> y >> type;
        auto t = restaurant{name, type, x, y};
        ls.push_front(t);
    }
    while (n--) {
        int x, y, k;
        string type;
            cin >> x >> y >> type >> k;
            pdli::list<restaurant> temp;
            for_each(ls.begin(), ls.end(), [&](restaurant& a) {
            if (a.type == type) {
                a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y))
                    ;
                temp.push_front(a);
            }
        });
        temp.merge_sort();
        int cnt = k;
        for (auto i : temp) {
            if (--cnt < 0) {
                break;
            }
            cout << i.name << ' ' << fixed << setprecision(3) << i.dis << endl;
        }
    }
```

```
#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;
        }
        friend bool operator==(const list_iterator& lhs, const list_iterator& rhs) {
            return lhs._M_node == rhs._M_node;
        }
        friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs) {
            return !(lhs == rhs);
        }
        friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs) {
            list_iterator* ans = lhs;
            for (size_t i = 0; i < rhs; i++) {
                (*ans)++;
            }
            return *ans;
        }
    };

public:
    typedef list::list_iterator iterator;
    typedef list::list_node node;
    list() : head(nullptr) {}
    list(const list& other);
    ~list() noexcept;
    size_t size() const;
    bool empty() const;
    void push_back(const T& item);
    void push_front(const T& item);
    void pop_back();
    void pop_front();
    void clear();
    void reverse();
    void insert(iterator pos, const T& item);
    void insert(size_t pos, const T& item);
    void remove(const T& value);
    iterator erase(iterator pos);
    iterator erase(iterator first, iterator last);
    void erase(size_t pos);
    void erase(size_t first, size_t last);
    void replace(const T& old_item, const T& new_item);
    iterator begin() const;
    iterator end() const;
    iterator find(const T& item) const;
    T& back() const;
    T& front() const;
    list& operator=(const list& other);
    void insertion_sort();
    void merge_sort();

protected:
    node* head = nullptr;

private:
    void _M_merge_sort(node*& list);
};

template <typename T>
list<T>::list(const list& other) {
    if (other.head == nullptr) return;
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
}

template <typename T>
bool list<T>::empty() const {
    return head == nullptr;
}

template <typename T>
size_t list<T>::size() const {
    size_t cnt = 0;
    for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
    return cnt;
}

template <typename T>
void list<T>::push_back(const T& item) {
    node *cur, *pre;
    if (head == nullptr) {
        cur = new node(item);
        head = pre = cur;
        return;
    }
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    node* new_node = new node(item);
    cur->next = new_node;
}

template <typename T>
void list<T>::pop_back() {
    node *cur, *pre;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == head) {
        delete cur;
        head = cur = pre = nullptr;
        return;
    }
    for (pre = head; pre->next != cur; pre = pre->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    pre->next = nullptr;
    cur = pre;
    for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
        ;
}

template <typename T>
T& list<T>::back() const {
    node* cur;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    return cur->entry;
}

template <typename T>
void list<T>::clear() {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
    pre = head = cur = nullptr;
}

template <typename T>
list<T>::~list() noexcept {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
}

template <typename T>
typename list<T>::iterator list<T>::begin() const {
    return iterator(head);
}

template <typename T>
typename list<T>::iterator list<T>::end() const {
    return iterator(nullptr);
}

template <typename T>
void list<T>::insert(iterator pos, const T& item) {
    node *cur, *pre;
    if (pos._M_node == head) {
        head = new node(item, head);
        return;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
        ;
    cur = pre->next;
    pre->next = new node(item, cur);
```

```
}

template <typename T>
void list<T>::insert(size_t pos, const T& item) {
    insert(begin() + pos, item);
}

template <typename T>
typename list<T>::iterator list<T>::erase(iterator pos) {
    node *cur, *pre;
    iterator res(pos._M_node->next);
    if (pos._M_node == head) {
        node* temp = head;
        head = temp->next;
        delete temp;
        return res;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
        ;
    cur = pre->next;
    pre->next = cur->next;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    return res;
}

template <typename T>
void list<T>::erase(size_t pos) {
    erase(begin() + pos);
}

template <typename T>
void list<T>::erase(size_t first, size_t last) {
    erase(begin() + first, begin() + last);
}

template <typename T>
void list<T>::reverse() {
    list<T> temp(*this);
    clear();
    while (!temp.empty()) {
        push_back(temp.back());
        temp.pop_back();
    }
```

```
}

template <typename T>
list<T>& list<T>::operator=(const list<T>& other) {
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
    return *this;
}

template <typename T>
typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
    for (auto it = first; it != last; it = erase(it))
        ;
    return last;
}

template <typename T>
void list<T>::push_front(const T& item) {
    insert(begin(), item);
}

template <typename T>
void list<T>::pop_front() {
    erase(begin());
}

template <typename T>
typename list<T>::iterator list<T>::find(const T& item) const {
    for (auto it = begin(); it != end(); ++it) {
        if (*it == item) return it;
    }
    return end();
}

template <typename T>
T& list<T>::front() const {
    if (head == nullptr) throw std::underflow_error("underflow");
    return head->entry;
}

template <typename T>
void list<T>::remove(const T& value) {
    for (auto it = begin(); it != end();) {
        if (*it == value) {
            it = erase(it);
        } else {
            it++;
        }
    }
}

template <typename T>
void list<T>::replace(const T& old_item, const T& new_item) {
    for (auto& i : *this) {
        if (i == old_item) {
            i = new_item;
        }
    }
}

template <typename T>
void list<T>::insertion_sort() {
    node *last_sorted, *first_unsorted;
    if (head == nullptr) return;
    last_sorted = head;
    while (last_sorted->next != nullptr) {
        first_unsorted = last_sorted->next;
        node* pre = head;
        bool flag = false;
        for (node* cur = head; cur != first_unsorted; cur = cur->next) {
            if (cur == head) {
                if (first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    first_unsorted->next = cur;
                    head = first_unsorted;
                    flag = true;
                    break;
                }
            } else {
                if (first_unsorted->entry > pre->entry
                    && first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    pre->next = first_unsorted;
                    first_unsorted->next = cur;
                    flag = true;
                    break;
                }
            }
            pre = cur;
        }
        if (!flag) last_sorted = last_sorted->next;
    }
}

template <typename T>
void list<T>::merge_sort() {
    _M_merge_sort(head);
}

template <typename T>
void list<T>::_M_merge_sort(node*& list) {
    if (list == nullptr || list->next == nullptr) return;
    // divide
    node *mid = list, *sec = list, *pos = list;
    while (pos != nullptr && pos->next != nullptr) {
        mid = sec;
        sec = sec->next;
        pos = pos->next->next;
    }
    mid->next = nullptr;
    // sort
    _M_merge_sort(list);
    _M_merge_sort(sec);
    // merge
    node* cur = new node;
    node* new_head = cur;
    node* fst = list;
    while (fst != nullptr && sec != nullptr) {
        if (fst->entry < sec->entry) {
            cur->next = fst;
            fst = fst->next;
            cur = cur->next;
        } else {
            cur->next = sec;
            sec = sec->next;
            cur = cur->next;
        }
    }
    if (fst == nullptr) {
        cur->next = sec;
    } else {
        cur->next = fst;
    }
    list = new_head->next;
    delete new_head;
}

template <typename T, typename U>
class hash_map {
    struct hash_map_record {
        T key;
        U value = U();
        hash_map_record(const T& key) : key(key){};
        hash_map_record(const T& key, const U& value) : key(key), value(value){};
        bool operator==(const hash_map_record& other) const {
            return key == other.key;
        }
    };
    struct hash_map_iterator {
        list<hash_map_record>* _M_node;
        typename list<hash_map_record>::iterator _M_list_iterator;
        hash_map_iterator() = default;
        hash_map_iterator(list<hash_map_record>* node, const typename list<
            hash_map_record>::iterator& it) : _M_node(node), _M_list_iterator(it){};
        hash_map_iterator operator++(int) {
            hash_map_iterator tmp = *this;
            do {
                if (_M_list_iterator != _M_node->end()) {
                    ++_M_list_iterator;
                } else {
                    ++_M_node;
                    _M_list_iterator = _M_node->begin();
                }
            } while (_M_list_iterator == nullptr);
            return tmp;
        }
        hash_map_iterator& operator++() {
            do {
```

```cpp
                if (_M_list_iterator != _M_node->end()) {
                    ++_M_list_iterator;
                } else {
                    ++_M_node;
                    _M_list_iterator = _M_node->begin();
                }
            } while (_M_list_iterator == nullptr);
            return *this;
        }
        bool operator==(const hash_map_iterator& other) {
            return _M_node == other._M_node && _M_list_iterator == _M_list_iterator;
        }
        bool operator!=(const hash_map_iterator& other) {
            return !(*this == other);
        }
        hash_map_record& operator*() noexcept {
            return *_M_list_iterator;
        }
    };

public:
    typedef hash_map_record record;
    typedef hash_map_iterator iterator;
    static const size_t npos = static_cast<size_t>(-1);
    hash_map();
    hash_map(size_t n);
    hash_map(const hash_map& other);
    ~hash_map();
    hash_map& operator=(const hash_map& other);
    void insert(const T& key, const U& value);
    U& operator[](const T& key) const;
    size_t get_position(const T& key) const;
    U& get_value(const T& key) const;
    iterator get_iterator(const T& key) const;
    void remove(const T& key);
    static size_t _hash_fun(const T& key, const size_t& size);
    iterator begin() const;
    iterator end() const;

protected:
    size_t hash_size = 100;
    list<record>* table;
};
```

```cpp

template <typename T, typename U>
hash_map<T, U>::hash_map() {
    table = new list<record>[hash_size + 1];
}

template <typename T, typename U>
hash_map<T, U>::~hash_map() {
    delete[] table;
}

template <typename T, typename U>
hash_map<T, U>::hash_map(size_t n) {
    hash_size = n;
    table = new list<record>[hash_size + 1];
}

template <typename T, typename U>
hash_map<T, U>::hash_map(const hash_map& other) {
    hash_size = other.hash_size;
    table = new list<record>[hash_size + 1];
    for (int i = 0; i < hash_size; ++i) {
        table[i] = other.table[i];
    }
}

template <typename T, typename U>
hash_map<T, U>& hash_map<T, U>::operator=(const hash_map<T, U>& other) {
    if (this == &other) return *this;
    delete[] table;
    hash_size = other.hash_size;
    table = new list<record>[hash_size + 1];
    for (int i = 0; i < hash_size; ++i) {
        table[i] = other.table[i];
    }
    return *this;
}

template <typename T, typename U>
void hash_map<T, U>::insert(const T& key, const U& value) {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) == table[pos].end()) {
        table[pos].push_front({key, value});
    } else {
        table[pos].replace(key, {key, value});
    }
}

template <typename T, typename U>
size_t hash_map<T, U>::get_position(const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) == table[pos].end()) {
        return npos;
    }
    return pos;
}

template <typename T, typename U>
U& hash_map<T, U>::get_value(const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return table[pos].find(key)->value;
    } else {
        throw std::runtime_error("Value of the key does not exist");
    }
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::get_iterator(const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return iterator(table + pos, table[pos].find(key));
    } else {
        return iterator(table + pos, table[pos].begin());
    }
}

template <typename T, typename U>
void hash_map<T, U>::remove(const T& key) {
    size_t pos = _hash_fun(key, hash_size);
    table[pos].remove(key);
}

template <typename T, typename U>
U& hash_map<T, U>::operator[](const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return table[pos].find(key)->value;
    } else {
        table[pos].push_front(key);
        return table[pos].begin()->value;
    }
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::begin() const {
    return iterator(table, table->begin());
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::end() const {
    return iterator(table + hash_size + 1, (table + hash_size + 1)->end());
}

template <typename T, typename U>
size_t hash_map<T, U>::_hash_fun(const T& key, const size_t& size) {
    unsigned seed = 31;
    unsigned hash = 0;
    T tmp = key;
    if (std::is_same<T, std::string>::value) {
        for (const auto& i : key) {
            hash = (hash * seed + i) % size;
        }
        return hash % size;
    }
    return hash % size;
}

} // namespace pdli

struct restaurant {
    string name;
    int x;
    int y;
    double dis;
    restaurant() {}
    restaurant(string name, int x, int y) : name(name), x(x), y(y) {}
    bool operator<(const restaurant& other) const {
        return tie(dis, name) < tie(other.dis, other.name);
    }
};
```

```

int main() {
    IO;
    auto clk = clock();
    int m, n;
    cin >> m >> n;
    pdli::hash_map<string, pdli::list<restaurant>> mp;
    while (m--) {
        string name, type;
        int x, y;
        cin >> name >> x >> y >> type;
        auto t = restaurant{name, x, y};
        mp[type].push_front(t);
    }
    while (n--) {
        int x, y, k;
        string type;
        cin >> x >> y >> type >> k;
        for_each(mp[type].begin(), mp[type].end(), [&](restaurant& a) {
            a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
        });
        mp[type].merge_sort();
        int cnt = k;
        for (auto i : mp[type]) {
            if (--cnt < 0) {
                break;
            }
            cout << i.name << ' ' << fixed << setprecision(3) << i.dis << endl;
        }
    }
#ifndef ONLINE_JUDGE
    cerr << "time : " << clock() - clk << "ms" << endl;
#endif
}
```

6.3 算法三：使用哈希表和 AVL 树

```
#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 {
```

```cpp
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;
        }
        friend bool operator==(const list_iterator& lhs, const list_iterator& rhs) {
            return lhs._M_node == rhs._M_node;
        }
        friend bool operator!=(const list_iterator& lhs, const list_iterator& rhs) {
            return !(lhs == rhs);
        }
        friend list_iterator& operator+(const list_iterator& lhs, const size_t& rhs) {
            list_iterator* ans = lhs;
            for (size_t i = 0; i < rhs; i++) {
                (*ans)++;
            }
```

```
                return *ans;
            }
    };

public:
    typedef list::list_iterator iterator;
    typedef list::list_node node;
    list() : head(nullptr) {}
    list(const list& other);
    ~list() noexcept;
    size_t size() const;
    bool empty() const;
    void push_back(const T& item);
    void push_front(const T& item);
    void pop_back();
    void pop_front();
    void clear();
    void reverse();
    void insert(iterator pos, const T& item);
    void insert(size_t pos, const T& item);
    void remove(const T& value);
    iterator erase(iterator pos);
    iterator erase(iterator first, iterator last);
    void erase(size_t pos);
    void erase(size_t first, size_t last);
    void replace(const T& old_item, const T& new_item);
    iterator begin() const;
    iterator end() const;
    iterator find(const T& item) const;
    T& back() const;
    T& front() const;
    list& operator=(const list& other);
    void insertion_sort();
    void merge_sort();

protected:
    node* head = nullptr;

private:
    void _M_merge_sort(node*& list);
};

template <typename T>
list<T>::list(const list& other) {
    if (other.head == nullptr) return;
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
}

template <typename T>
bool list<T>::empty() const {
    return head == nullptr;
}

template <typename T>
size_t list<T>::size() const {
    size_t cnt = 0;
    for (node* cur = head; cur != nullptr; cur = cur->next) cnt++;
    return cnt;
}

template <typename T>
void list<T>::push_back(const T& item) {
    node *cur, *pre;
    if (head == nullptr) {
        cur = new node(item);
        head = pre = cur;
        return;
    }
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    node* new_node = new node(item);
    cur->next = new_node;
}

template <typename T>
void list<T>::pop_back() {
    node *cur, *pre;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == head) {
        delete cur;
        head = cur = pre = nullptr;
        return;
    }
    for (pre = head; pre->next != cur; pre = pre->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    pre->next = nullptr;
    cur = pre;
    for (pre = head; pre->next != cur && pre->next != nullptr; pre = pre->next)
        ;
}

template <typename T>
T& list<T>::back() const {
    node* cur;
    for (cur = head; cur->next != nullptr; cur = cur->next)
        ;
    if (cur == nullptr) throw std::underflow_error("underflow");
    return cur->entry;
}

template <typename T>
void list<T>::clear() {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
    pre = head = cur = nullptr;
}

template <typename T>
list<T>::~list() noexcept {
    node *cur, *pre;
    if (head == nullptr) return;
    pre = head;
    cur = head->next;
    while (cur != nullptr) {
        delete pre;
        pre = cur;
        cur = cur->next;
    }
    delete pre;
}

template <typename T>
typename list<T>::iterator list<T>::begin() const {
    return iterator(head);
}

template <typename T>
typename list<T>::iterator list<T>::end() const {
    return iterator(nullptr);
}

template <typename T>
void list<T>::insert(iterator pos, const T& item) {
    node *cur, *pre;
    if (pos._M_node == head) {
        head = new node(item, head);
        return;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
        ;
    cur = pre->next;
    pre->next = new node(item, cur);
}

template <typename T>
void list<T>::insert(size_t pos, const T& item) {
    insert(begin() + pos, item);
}

template <typename T>
typename list<T>::iterator list<T>::erase(iterator pos) {
    node *cur, *pre;
    iterator res(pos._M_node->next);
    if (pos._M_node == head) {
        node* temp = head;
        head = temp->next;
        delete temp;
        return res;
    }
    for (pre = head; pre->next != pos._M_node; pre = pre->next)
        ;
    cur = pre->next;
    pre->next = cur->next;
    if (cur == nullptr) throw std::underflow_error("underflow");
    delete cur;
    return res;
}

template <typename T>
void list<T>::erase(size_t pos) {
    erase(begin() + pos);
}

template <typename T>
void list<T>::erase(size_t first, size_t last) {
    erase(begin() + first, begin() + last);
}

template <typename T>
void list<T>::reverse() {
    list<T> temp(*this);
    clear();
    while (!temp.empty()) {
        push_back(temp.back());
        temp.pop_back();
    }
}

template <typename T>
list<T>& list<T>::operator=(const list<T>& other) {
    node *cur, *pre;
    node* _head = other.head;
    cur = new node(other.head->entry);
    head = cur;
    while (_head->next != nullptr) {
        _head = _head->next;
        cur->next = new node(_head->entry, _head->next);
        pre = cur;
        cur = cur->next;
    }
    return *this;
}

template <typename T>
typename list<T>::iterator list<T>::erase(iterator first, iterator last) {
    for (auto it = first; it != last; it = erase(it))
        ;
    return last;
}

template <typename T>
void list<T>::push_front(const T& item) {
    insert(begin(), item);
}

template <typename T>
void list<T>::pop_front() {
    erase(begin());
}

template <typename T>
typename list<T>::iterator list<T>::find(const T& item) const {
    for (auto it = begin(); it != end(); ++it) {
        if (*it == item) return it;
    }
    return end();
}

template <typename T>
T& list<T>::front() const {
    if (head == nullptr) throw std::underflow_error("underflow");
    return head->entry;
}

template <typename T>
void list<T>::remove(const T& value) {
    for (auto it = begin(); it != end();) {
        if (*it == value) {
            it = erase(it);
        } else {
            it++;
        }
    }
}
```

```cpp

template <typename T>
void list<T>::replace(const T& old_item, const T& new_item) {
    for (auto& i : *this) {
        if (i == old_item) {
            i = new_item;
        }
    }
}

template <typename T>
void list<T>::insertion_sort() {
    node *last_sorted, *first_unsorted;
    if (head == nullptr) return;
    last_sorted = head;
    while (last_sorted->next != nullptr) {
        first_unsorted = last_sorted->next;
        node* pre = head;
        bool flag = false;
        for (node* cur = head; cur != first_unsorted; cur = cur->next) {
            if (cur == head) {
                if (first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    first_unsorted->next = cur;
                    head = first_unsorted;
                    flag = true;
                    break;
                }
            } else {
                if (first_unsorted->entry > pre->entry
                    && first_unsorted->entry <= cur->entry) {
                    last_sorted->next = first_unsorted->next;
                    pre->next = first_unsorted;
                    first_unsorted->next = cur;
                    flag = true;
                    break;
                }
            }
            pre = cur;
        }
        if (!flag) last_sorted = last_sorted->next;
    }
}

```

```
template <typename T>
void list<T>::merge_sort() {
    _M_merge_sort(head);
}

template <typename T>
void list<T>::_M_merge_sort(node*& list) {
    if (list == nullptr || list->next == nullptr) return;
    // divide
    node *mid = list, *sec = list, *pos = list;
    while (pos != nullptr && pos->next != nullptr) {
        mid = sec;
        sec = sec->next;
        pos = pos->next->next;
    }
    mid->next = nullptr;
    // sort
    _M_merge_sort(list);
    _M_merge_sort(sec);
    // merge
    node* cur = new node;
    node* new_head = cur;
    node* fst = list;
    while (fst != nullptr && sec != nullptr) {
        if (fst->entry < sec->entry) {
            cur->next = fst;
            fst = fst->next;
            cur = cur->next;
        } else {
            cur->next = sec;
            sec = sec->next;
            cur = cur->next;
        }
    }
    if (fst == nullptr) {
        cur->next = sec;
    } else {
        cur->next = fst;
    }
    list = new_head->next;
    delete new_head;
}

template <typename T, typename U>
```

```
class hash_map {
    struct hash_map_record {
        T key;
        U value = U();
        hash_map_record(const T& key) : key(key){};
        hash_map_record(const T& key, const U& value) : key(key), value(value){};
        bool operator==(const hash_map_record& other) const {
            return key == other.key;
        }
    };
    struct hash_map_iterator {
        list<hash_map_record>* _M_node;
        typename list<hash_map_record>::iterator _M_list_iterator;
        hash_map_iterator() = default;
        hash_map_iterator(list<hash_map_record>* node, const typename list<hash_map_record>::iterator& it) : _M_node(node), _M_list_iterator(it){};
        hash_map_iterator operator++(int) {
            hash_map_iterator tmp = *this;
            do {
                if (_M_list_iterator != _M_node->end()) {
                    ++_M_list_iterator;
                } else {
                    ++_M_node;
                    _M_list_iterator = _M_node->begin();
                }
            } while (_M_list_iterator == nullptr);
            return tmp;
        }
        hash_map_iterator& operator++() {
            do {
                if (_M_list_iterator != _M_node->end()) {
                    ++_M_list_iterator;
                } else {
                    ++_M_node;
                    _M_list_iterator = _M_node->begin();
                }
            } while (_M_list_iterator == nullptr);
            return *this;
        }
        bool operator==(const hash_map_iterator& other) {
            return _M_node == other._M_node && _M_list_iterator == _M_list_iterator;
        }
        bool operator!=(const hash_map_iterator& other) {
            return !(*this == other);
        }
        hash_map_record& operator*() noexcept {
            return *_M_list_iterator;
        }
    };

public:
    typedef hash_map_record record;
    typedef hash_map_iterator iterator;
    static const size_t npos = static_cast<size_t>(-1);
    hash_map();
    hash_map(size_t n);
    hash_map(const hash_map& other);
    ~hash_map();
    hash_map& operator=(const hash_map& other);
    void insert(const T& key, const U& value);
    U& operator[](const T& key) const;
    size_t get_position(const T& key) const;
    U& get_value(const T& key) const;
    iterator get_iterator(const T& key) const;
    void remove(const T& key);
    static size_t _hash_fun(const T& key, const size_t& size);
    iterator begin() const;
    iterator end() const;

protected:
    size_t hash_size = 100;
    list<record>* table;
};

template <typename T, typename U>
hash_map<T, U>::hash_map() {
    table = new list<record>[hash_size + 1];
}

template <typename T, typename U>
hash_map<T, U>::~hash_map() {
    delete[] table;
}

template <typename T, typename U>
hash_map<T, U>::hash_map(size_t n) {
    hash_size = n;
```

```cpp
    table = new list<record>[hash_size + 1];
}

template <typename T, typename U>
hash_map<T, U>::hash_map(const hash_map& other) {
    hash_size = other.hash_size;
    table = new list<record>[hash_size + 1];
    for (int i = 0; i < hash_size; ++i) {
        table[i] = other.table[i];
    }
}

template <typename T, typename U>
hash_map<T, U>& hash_map<T, U>::operator=(const hash_map<T, U>& other) {
    if (this == &other) return *this;
    delete[] table;
    hash_size = other.hash_size;
    table = new list<record>[hash_size + 1];
    for (int i = 0; i < hash_size; ++i) {
        table[i] = other.table[i];
    }
    return *this;
}

template <typename T, typename U>
void hash_map<T, U>::insert(const T& key, const U& value) {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) == table[pos].end()) {
        table[pos].push_front({key, value});
    } else {
        table[pos].replace(key, {key, value});
    }
}

template <typename T, typename U>
size_t hash_map<T, U>::get_position(const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) == table[pos].end()) {
        return npos;
    }
    return pos;
}

template <typename T, typename U>
```

```
U& hash_map<T, U>::get_value(const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return table[pos].find(key)->value;
    } else {
        throw std::runtime_error("Value of the key does not exist");
    }
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::get_iterator(const T& key) const
    {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return iterator(table + pos, table[pos].find(key));
    } else {
        return iterator(table + pos, table[pos].begin());
    }
}

template <typename T, typename U>
void hash_map<T, U>::remove(const T& key) {
    size_t pos = _hash_fun(key, hash_size);
    table[pos].remove(key);
}

template <typename T, typename U>
U& hash_map<T, U>::operator[](const T& key) const {
    size_t pos = _hash_fun(key, hash_size);
    if (table[pos].find(key) != table[pos].end()) {
        return table[pos].find(key)->value;
    } else {
        table[pos].push_front(key);
        return table[pos].begin()->value;
    }
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::begin() const {
    return iterator(table, table->begin());
}

template <typename T, typename U>
typename hash_map<T, U>::iterator hash_map<T, U>::end() const {
```

```cpp
    return iterator(table + hash_size + 1, (table + hash_size + 1)->end());
}

template <typename T, typename U>
size_t hash_map<T, U>::_hash_fun(const T& key, const size_t& size) {
    unsigned seed = 31;
    unsigned hash = 0;
    T tmp = key;
    if (is_same<T, std::string>::value) {
        for (const auto& i : key) {
            hash = (hash * seed + i) % size;
        }
        return hash % size;
    }
    return hash % size;
}

template <typename T>
class AVL_tree {
    enum class balance_factor {
        left_higher,
        right_higher,
        equal_height
    };
    struct AVL_tree_node {
        T data;
        balance_factor balance = balance_factor::equal_height;
        AVL_tree_node* left = nullptr;
        AVL_tree_node* right = nullptr;
        AVL_tree_node() = default;
        AVL_tree_node(const T& item) : data(item) {}
    };

public:
    using node = AVL_tree_node;

    AVL_tree() = default;

    AVL_tree(const AVL_tree& other) {
        clear();
        std::function<void(node*)> m_insert = [&](node* sub_root) {
            if (sub_root == nullptr) return;
            insert(sub_root->data);
            m_insert(sub_root->left);
            m_insert(sub_root->right);
        };
        m_insert(other.root);
    }

    AVL_tree(AVL_tree&& other) {
        clear();
        root = std::move(other.root);
        other.root = nullptr;
    }

    ~AVL_tree() {
        clear();
    }

    AVL_tree& operator=(const AVL_tree& other) {
        if (this == &other) return *this;
        clear();
        std::function<void(node*)> m_insert = [&](node* sub_root) {
            if (sub_root == nullptr) return;
            insert(sub_root->data);
            m_insert(sub_root->left);
            m_insert(sub_root->right);
        };
        m_insert(other.root);
        return *this;
    }

    AVL_tree& operator=(AVL_tree&& other) {
        if (this == &other) return *this;
        clear();
        root = std::move(other.root);
        other.root = nullptr;
        return *this;
    }

    /**
            * @brief Insert an item at the AVL_tree. If the item is already in the tree, throw a duplicate_errer.
            * @param item the item to insert
            */
    void insert(const T& item) {
        AVL_insert(root, item);
    }

    /**
            * @brief Clear the tree.
            */
    void clear() {
        m_clear(root);
        root = nullptr;
    }

    /**
             * @brief Remove an item in the tree
             * @param item the item to remove
             */
    void remove(const T& item) {
        AVL_remove(root, item);
    }

    /**
             * @brief Find an item in the tree.
             * @param item the item to find
             * @return reference of the item found in the tree
             */
    T& find(const T& item) {
        node* cur = root;
        while (cur != nullptr && cur->data != item) {
            std::cout << cur->data.key << ' ';
            if (item < cur->data) {
                cur = cur->left;
            } else {
                cur = cur->right;
            }
        }
        if (cur == nullptr) {
            throw std::runtime_error("not present");
        } else {
            return cur->data;
        }
    }

    void in_order(const std::function<void(T&)>& fun) {
        std::function<void(node*, std::function<void(T&)>)> _inorder = [&](node* root, std::function<void(T&)> fun) {
            if (root == nullptr) return;
            _inorder(root->left, fun);
```

```cpp
            fun(root->data);
            _inorder(root->right, fun);
        };
        _inorder(root, fun);
    }

    void pre_order(const std::function<void(T&)>& fun) {
        std::function<void(node*, std::function<void(T&)>)> _preorder = [&](node* root, std::function<void(T&)> fun) {
            if (root == nullptr) return;
            fun(root->data);
            _preorder(root->left, fun);
            _preorder(root->right, fun);
        };
        _preorder(root, fun);
    }

    void post_order(const std::function<void(T&)>& fun) {
        std::function<void(node*, std::function<void(T&)>)> _postorder = [&](node* root, std::function<void(T&)> fun) {
            if (root == nullptr) return;
            _postorder(root->left, fun);
            _postorder(root->right, fun);
            fun(root->data);
        };
        _postorder(root, fun);
    }

protected:
    node* root = nullptr;

private:
    /**
             * @brief Insert an item in the subtree. If the item is already in the subtree, throw a duplicate_errer.
             * @param subroot where to insert
             * @param item the item to insert
             * @return the subtree is increased in height or not
            */
    bool AVL_insert(node*& sub_root, const T& item) {
        if (sub_root == nullptr) {
            sub_root = new node(item);
            return true;
        } else if (item == sub_root->data) {
            throw std::runtime_error("duplicate error");
        } else if (item < sub_root->data) {
            bool taller = AVL_insert(sub_root->left, item);
            if (taller) {
                switch (sub_root->balance) {
                case balance_factor::left_higher:
                    left_balance(sub_root);
                    taller = false;
                    break;
                case balance_factor::right_higher:
                    sub_root->balance = balance_factor::equal_height;
                    taller = false;
                    break;
                case balance_factor::equal_height:
                    sub_root->balance = balance_factor::left_higher;
                    break;
                }
            }
            return taller;
        } else {
            bool is_taller = AVL_insert(sub_root->right, item);
            if (is_taller) {
                switch (sub_root->balance) {
                case balance_factor::left_higher:
                    sub_root->balance = balance_factor::equal_height;
                    is_taller = false;
                    break;
                case balance_factor::right_higher:
                    right_balance(sub_root);
                    is_taller = false;
                    break;
                case balance_factor::equal_height:
                    sub_root->balance = balance_factor::right_higher;
                    break;
                }
            }
            return is_taller;
        }
    }

    /**
             * @brief Make a left subtree balanced
             * @param sub_root root of the left subtree
             */
    void left_balance(node*& sub_root) {
        node*& left_tree = sub_root->left;
        switch (left_tree->balance) {
        case balance_factor::left_higher: // case L-L
            sub_root->balance = balance_factor::equal_height;
            left_tree->balance = balance_factor::equal_height;
            rotate_right(sub_root);
            break;
        case balance_factor::equal_height:
            throw std::runtime_error("impossible case in left_balance");
            break;
        case balance_factor::right_higher: // case L-R
            node* sub_tree = left_tree->right;
            switch (sub_tree->balance) {
            case balance_factor::equal_height:
                sub_root->balance = balance_factor::equal_height;
                left_tree->balance = balance_factor::equal_height;
                break;
            case balance_factor::left_higher:
                sub_root->balance = balance_factor::right_higher;
                left_tree->balance = balance_factor::equal_height;
                break;
            case balance_factor::right_higher:
                sub_root->balance = balance_factor::equal_height;
                left_tree->balance = balance_factor::left_higher;
                break;
            }
            sub_tree->balance = balance_factor::equal_height;
            rotate_left(left_tree);
            rotate_right(sub_root);
            break;
        }
    }

    /**
             * @brief Make a right subtree balanced
             * @param sub_root root of the right subtree
             */
    void right_balance(node*& sub_root) {
        node*& right_tree = sub_root->right;
        switch (right_tree->balance) {
        case balance_factor::right_higher: // case R-R
            sub_root->balance = balance_factor::equal_height;
            right_tree->balance = balance_factor::equal_height;
            rotate_left(sub_root);
            break;
        case balance_factor::equal_height:
            throw std::runtime_error("impossible case in right_balance");
            break;
        case balance_factor::left_higher: // case R-L
            node* sub_tree = right_tree->left;
            switch (sub_tree->balance) {
            case balance_factor::equal_height:
                sub_root->balance = balance_factor::equal_height;
                right_tree->balance = balance_factor::equal_height;
                break;
            case balance_factor::left_higher:
                sub_root->balance = balance_factor::equal_height;
                right_tree->balance = balance_factor::right_higher;
                break;
            case balance_factor::right_higher:
                sub_root->balance = balance_factor::left_higher;
                right_tree->balance = balance_factor::equal_height;
                break;
            }
            sub_tree->balance = balance_factor::equal_height;
            rotate_right(right_tree);
            rotate_left(sub_root);
            break;
        }
    }

    /**
             * @brief Do a left rotation at the sub_tree
             * @param sub_root the root of the sub_tree
            */
    void rotate_left(node*& sub_root) {
        if (sub_root == nullptr || sub_root->right == nullptr) {
            throw std::runtime_error("impossible case in rotate_left");
        }
        node* right_tree = sub_root->right;
        sub_root->right = right_tree->left;
        right_tree->left = sub_root;
        sub_root = right_tree;
    }

    /**
             * @brief Do a right rotation at the sub_tree
             * @param sub_root the root of the sub_tree
            */
    void rotate_right(node*& sub_root) {
        if (sub_root == nullptr || sub_root->left == nullptr) {
            throw std::runtime_error("impossible case in rotate_right");
        }
        node* left_tree = sub_root->left;
        sub_root->left = left_tree->right;
        left_tree->right = sub_root;
        sub_root = left_tree;
    }

    /**
             * @brief Clear a subtree.
             * @param sub_root the root of the subtree
             */
    void m_clear(node* sub_root) {
        if (sub_root == nullptr) return;
        m_clear(sub_root->left);
        m_clear(sub_root->right);
        delete sub_root;
    }

    /**
             * @brief Remove an item in a subtree.
             * @param sub_root the root of the subtree
             * @param item the item to remove
             * @return the sub_tree becomes shorter or not
             */
    bool AVL_remove(node*& sub_root, const T& item) {
        if (sub_root == nullptr) {
            throw std::runtime_error("not found in AVL_remove");
        } else if (item < sub_root->data) {
            return remove_left(sub_root, item);
        } else if (item > sub_root->data) {
            return remove_right(sub_root, item);
        } else if (sub_root->right == nullptr) {
            node* temp = sub_root;
            sub_root = sub_root->left;
            delete temp;
            return true;
        } else if (sub_root->left == nullptr) {
            node* temp = sub_root;
            sub_root = sub_root->right;
            delete temp;
            return true;
        } else if (sub_root->balance == balance_factor::right_higher) {
            // another solution
            // node* temp = sub_root->right;
            // while (temp->left != nullptr) {
            //     temp = temp->left;
            // }
            // return remove_right(sub_root, temp->data);
            node* temp = sub_root->left;
            while (temp->right != nullptr) {
                temp = temp->right;
            }
            sub_root->data = temp->data;
            return remove_left(sub_root, temp->data);
        } else {
            node* temp = sub_root->left;
            while (temp->right != nullptr) {
                temp = temp->right;
            }
            sub_root->data = temp->data;
            return remove_left(sub_root, temp->data);
        }
    }

    /**
             * @brief Remove an item from a left subtree.
             * @param sub_root the root of the left subtree
             * @param item the item to remove
             * @return the subtree becomes shorter or not
             */
    bool remove_left(node*& sub_root, const T& item) {
        bool shorter = AVL_remove(sub_root->left, item);
        if (shorter) {
            switch (sub_root->balance) {
            case balance_factor::left_higher:
                sub_root->balance = balance_factor::equal_height;
                break;
            case balance_factor::equal_height:
                sub_root->balance = balance_factor::right_higher;
                shorter = false;
                break;
            case balance_factor::right_higher:
                node* temp = sub_root->right;
                switch (temp->balance) {
                case balance_factor::equal_height:
                    temp->balance = balance_factor::left_higher;
                    rotate_left(sub_root);
                    shorter = false;
                    break;
                case balance_factor::right_higher:
                    sub_root->balance = balance_factor::equal_height;
                    temp->balance = balance_factor::equal_height;
                    rotate_left(sub_root);
                    break;
                case balance_factor::left_higher:
                    node* temp_left = temp->left;
                    switch (temp_left->balance) {
                    case balance_factor::equal_height:
                        sub_root->balance = balance_factor::equal_height;
                        temp->balance = balance_factor::equal_height;
                        break;
                    case balance_factor::right_higher:
                        sub_root->balance = balance_factor::left_higher;
                        temp->balance = balance_factor::equal_height;
                        break;
                    case balance_factor::left_higher:
                        sub_root->balance = balance_factor::equal_height;
                        temp->balance = balance_factor::right_higher;
                        break;
                    }
                    temp_left->balance = balance_factor::equal_height;
                    rotate_right(sub_root->right);
                    rotate_left(sub_root);
                    break;
                }
            }
        }
        return shorter;
    }

    /**
             * @brief Remove an item from a right subtree.
             * @param sub_root the root of the right subtree
             * @param item the item to remove
             * @return the subtree becomes shorter or not
             */
    bool remove_right(node*& sub_root, const T& item) {
        bool shorter = AVL_remove(sub_root->right, item);
        if (shorter) {
            switch (sub_root->balance) {
            case balance_factor::right_higher:
                sub_root->balance = balance_factor::equal_height;
                break;
            case balance_factor::equal_height:
                sub_root->balance = balance_factor::left_higher;
                shorter = false;
                break;
            case balance_factor::left_higher:
                node* temp = sub_root->left;
                switch (temp->balance) {
                case balance_factor::equal_height:
                    temp->balance = balance_factor::right_higher;
                    rotate_right(sub_root);
                    shorter = false;
                    break;
                case balance_factor::left_higher:
                    sub_root->balance = balance_factor::equal_height;
                    temp->balance = balance_factor::equal_height;
                    rotate_right(sub_root);
                    break;
                case balance_factor::right_higher:
                    node* temp_right = temp->right;
                    switch (temp_right->balance) {
                    case balance_factor::equal_height:
                        sub_root->balance = balance_factor::equal_height;
                        temp->balance = balance_factor::equal_height;
                        break;
                    case balance_factor::left_higher:
                        sub_root->balance = balance_factor::right_higher;
                        temp->balance = balance_factor::equal_height;
                        break;
                    case balance_factor::right_higher:
                        sub_root->balance = balance_factor::equal_height;
                        temp->balance = balance_factor::left_higher;
                        break;
                    }
                    temp_right->balance = balance_factor::equal_height;
                    rotate_left(sub_root->left);
                    rotate_right(sub_root);
                    break;
                }
            }
        }
        return shorter;
    }
};

} // namespace pdli

struct restaurant {
    string name = "";
    int x = 0;
    int y = 0;
    double dis = 0;
    restaurant() {}
    restaurant(string name, int x, int y) : name(name), x(x), y(y) {}
    bool operator<(const restaurant& other) const {
        return tie(dis, name) < tie(other.dis, other.name);
    }
    bool operator==(const restaurant& other) const {
        return tie(name, x, y, dis) == tie(other.name, other.x, other.y, other.dis);
    }
};

int main() {
    IO;
    auto clk = clock();
    int m, n;
    cin >> m >> n;
    pdli::hash_map<string, pdli::list<restaurant>> unsorted_map;
    pdli::hash_map<string, pdli::AVL_tree<restaurant>> sorted_map;
    while (m--) {
        string name, type;
        int x, y;
        cin >> name >> x >> y >> type;
        auto t = restaurant{name, x, y};
        unsorted_map[type].push_front(t);
    }
    while (n--) {
        int x, y, k;
        string type;
        cin >> x >> y >> type >> k;
        for_each(unsorted_map[type].begin(), unsorted_map[type].end(), [&](restaurant& a) {
            a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
            sorted_map[type].insert(a);
        });
        int cnt = k;
        sorted_map[type].in_order([&](const restaurant& a) {
            if (--cnt < 0) {
                return;
            }
            cout << a.name << ' ' << fixed << setprecision(3) << a.dis << endl;
        });
    }
#ifndef ONLINE_JUDGE
    cerr << "time : " << clock() - clk << "ms" << endl;
#endif
}
```

6.4 算法四：使用 STL

```
#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;
struct restaurant {
    string name;
    int x;
    int y;
    double dis = 0;
};
int main() {
    IO;
    auto clk = clock();
    int m, n;
    cin >> m >> n;
    unordered_map<string, vector<restaurant>> mp;
    while (m--) {
        string name, type;
        int x, y;
        cin >> name >> x >> y >> type;
        mp[type].emplace_back(std::move(restaurant{name, x, y}));
    }
    while (n--) {
        int x, y, k;
        string type;
        cin >> x >> y >> type >> k;
        for_each(mp[type].begin(), mp[type].end(), [&](restaurant& a) {
            a.dis = sqrt((ll)(x - a.x) * (x - a.x) + (ll)(y - a.y) * (y - a.y));
        });
        sort(mp[type].begin(), mp[type].end(), [&](restaurant& a, restaurant& b) {
            return tie(a.dis, a.name) < tie(b.dis, b.name);
        });
        int cnt = k;
        for (auto i : mp[type]) {
            if (--cnt < 0) {
                break;
            }
            cout << i.name << fixed << setprecision(3) << i.dis << endl;
        }
    }
#ifndef ONLINE_JUDGE
    cerr << "time : " << clock() - clk << "ms" << endl;
#endif
}
```