矩阵连乘

实现矩阵连乘。

输入

第一行是矩阵的个数 `N`，保证 `N` 可用 `std::size_t` 容纳（可以通过引入头文件 `<cstddef>` 使用此类型），保证不小于 2.

从第二行开始是矩阵的内容。对于矩阵 m_k，首先一行是矩阵的行数 R_k 和列数 C_k，保证可用 `std::size_t` 容纳并且一定大于 0，接下来是矩阵的内容，每行 C_k 个值，一共 R_k 行，所有的值都可以用 `int` 类型容纳。

每行输入保证以 `\n` 结尾。

保证：

- 矩阵的形状是正确的
- 所有矩阵占用的内存数（以字节为单位）在 `std::size_t` 的范围内

输出

输出矩阵 m_1 到 m_N 连乘的结果。

示例

示例 1

输入

```
2
2 3
1 2 3
4 5 6
3 2
6 5
4 3
2 1
```

输出

```
20 14
56 41
```

提示

- 可用如下方式读取一个矩阵

```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>

int main() {
  int mat[10][10];
  std::size_t n_rows, n_cols;
  std::cin >> n_rows >> n_cols;
  for (int i = 0; i < n_rows; i++) {
    std::copy_n(std::istream_iterator<int>(std::cin), n_cols,
mat[i]);
  }
}
```

sort

`sort` 是 UNIX 及其变体中一个非常经典的命令，可用于将输入排序。

> 有兴趣的同学可以看看《AT&T Archives: The UNIX Operating System》。

在本题中，`sort` 默认将输入以行为单位并按照ASCII 码序排序，例如：

```
$ sort << EOF
heredoc> 1
heredoc> 10
heredoc> 2
heredoc> EOF
1
10
2
```

> 以 `$` 开始的行是用户的输入，有兴趣的同学可以了解下这种约定。
>
> `<< EOF` 称为 `heredoc`，有兴趣的同学可以戳这里了解。

其中以 `heredoc>` 开始的行是待排序的行（除了 `heredoc> EOF` 这一行），最后三行是程序的输出，可以看到其中 `10` 排在 `2` 前面（因为默认按照 ASCII 码序排序）。

要让 `sort` 按照数值排序，可以使用 `-n` 选项：

```
$ sort -n << EOF
heredoc> 1
heredoc> 10
heredoc> 2
heredoc> EOF
1
2
10
```

此外，还有逆序排序的 `-r` 选项：

```
$ sort -r << EOF
heredoc> 1
heredoc> 10
heredoc> 2
heredoc> EOF
2
10
1
```

本题中增加了 `-i` 选项表示忽略大小写进行比较：

```
$ sort << EOF
heredoc> a
heredoc> b
heredoc> Z
heredoc> EOF
Z
a
b

$ sort -i << EOF
heredoc> a
heredoc> b
heredoc> Z
heredoc> EOF
a
b
Z
```

解释：在 ASCII 码中，所有的大写英文字母排在小写英文字母之前，因此在前一种情况中 `Z` 在最前，忽略大小写之后，`Z` 就按照字母表的顺序排在了 `a` 的后面。

本题中还增加了 `-d` 选项表示只比较字母、数字和空格而忽略其他字符，例如：

```
$ sort << EOF
heredoc> a-c
heredoc> abd
heredoc> EOF
a-c
abd

$ sort -d << EOF
heredoc> a-c
heredoc> abd
heredoc> EOF
abd
a-c
```

解释：在 ASCII 码中，`-` 排在英文字母的前面，因此 `a-c` 会排在 `abd` 之前，而加上 `-d` 选项之后，两个字符串在第二次比较时就会使用 `c` 跟 `b` 比较，这样 `a-c` 就排在了后面。

所有可能的选项如下：

- 无选项，用 `-` 表示
- `n`
- `i`
- `d`
- `r`

使用函数指针实现 `sort` 命令的上述功能。

输入

第一行是数组的长度 `N`，保证 `N` 可用 `std::size_t` 容纳（可以通过引入头文件 `<cstddef>` 使用此类型）。

从第二行开始的 `N` 行均为字符串（可能包含空格！），字符串仅包含 ASCII 字母、数字、空格和标点符号，如果字符串是一个合法的整数（数学意义上的），不会出现前导零（例如，`023`），也不会出现正号 `+`，且必然能够用 `int` 容纳。

第 `N + 2` 行是命令数 `C`.

从第 `N + 3` 行开始的 `C` 行是命令。

每行输入保证以 `\n` 结尾。

输出

输出 `C` 组，每组 `N` 行，分别对应于相应的命令对数组进行排序的结果。

注意：在忽略大小写的情况下，对于英文意义上的同一个字母（例如，`A` 和 `a`），输出中大写字母应该排在小写字母之前。例如，输入为：

```
2
abc
ABC
1
i
```

输出应为：

```
ABC
abc
```

示例

示例 1

输入

```
3
1
10
2
2
-
n
```

输出

```
1
10
2
1
2
10
```

示例 2

输入

```
2
a-C
abd
3
-
d
i
```

输出

```
a-C
abd
a-C
abd
a-C
abd
```

提示

- 本题不允许使用除了 `std::string` 和 `std::vector` 之外的容器
- `std::string` 可以使用 `<` `>` `==` `!=` `<=` `>=` 进行比较
- 了解 C++ 标准库的 `sort` 函数
- 可以使用 `<cctype>` 头文件中的 `std::isalpha` `std::isdigit` `std::ispunct` 分别判断一个 `char` 是否是字母、数字和标点符号
- 可以使用 `std::getline` 读取一行数据

```cpp
#include <iostream>
#include <string>

int main() {
    std::string line;
    std::getline(std::cin, line);
}
```

- 读取 `int` 之后，调用 `getline` 之前，需要使用 `std::cin >> std::ws` 跳过行中剩下的空白符

```cpp
#include <iostream>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::cin >> std::ws;  // try to comment out the line
    std::string s;
    std::getline(std::cin, s);
}
```

题目描述

使用**链表**实现一个**双端队列**，双端队列的头部和尾部均可以进行**插入值**和**取出值**操作，你可以使用以下5种指令操作双端队列：

H3

1. `pushFront [val]`，其中`val`为一个`int`，且`val` ≥ 0， 例如`pushFront 1`，该指令可以将`val`放在节点中 **插入** 到双端队列的 **最前面** 。
2. `pushBack [val]`，其中`val`为一个`int`，且`val` ≥ 0， 例如`pushBack 2`，该指令可以将`val`放在节点中 **插入** 到双端队列的 **最后面** 。
3. `popFront`，该指令可以 **打印** 双端队列 **最前面有效节点** 的`val`，并且将其 **移出** 双端队列， **如果双端队列内没有有效节点，则打印-1** 。
4. `popBack`，该指令可以 **打印** 双端队列 **最后面有效节点** 的`val`，并且将其 **移出** 双端队列， **如果双端队列内没有有效节点，则打印-1** 。
5. `getSize`，该指令可以 **打印** 双端队列的 **有效节点数** 。

结构设计

我们设计好了**部分**结构体和**部分**方法声明，可以将它们运用到自己的代码中，**也可以自己进行设计和实现，但必须使用链表。**

H3

```
//链表节点
struct Node{
    Node* next;//前一个节点
    Node* prev;//后一个节点
    int val;//该节点存放的数字
};

//双端队列
struct Deque{
    int size;//有效节点数
    Node* front;//虚拟头节点，不作为有效节点
    Node* rear;//虚拟尾节点，不作为有效节点
};

void push_front (Deque* self, int value);
void push_back (Deque* self, int value);
void pop_front (Deque* self);
void pop_back (Deque* self);
```

输入输出

输入

第一行输入一个自然数`n`，表示接下来会进行`n`次操作。

接下来的`n`行每行为一条指令，具体如题目描述所示。

输出

`popFront`，`popBack`，`getSize`三条指令存在输出，均输出`int`，且需要进行换行。

输入输出示例

示例1

输入

```
5
pushFront 1
pushBack 2
popBack
popBack
popFront
```

输出

```
2 //popBack的输出
1 //popBack的输出
-1 //popFront的输出
```

示例2

输入

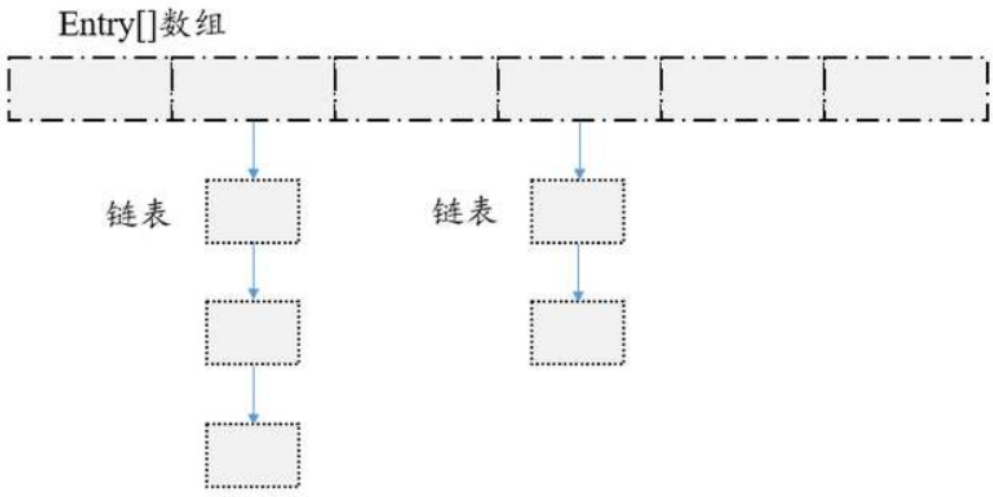
```
5
pushFront 1
pushFront 2
getSize
popBack
getSize
```

输出

```
2 //getSize的输出
1 //popBack的输出
1 //getSize的输出
```

HashDict

现在设计一个由数组和链表共同组成的一个存储键值对的数据结构HashDict，结构图如下。



该数据结构内部包含了一个 Entry 类型的数组 table。每个 Entry 存储着键值对。它包含了四个字段（hashCode, key, value, next），从 next 字段我们可以看出 Entry 是一个链表中的节点。即数组中的每个位置被当成一个桶，一个桶存放一个链表。其中键值对中key为整数，value为字符串。

这个数据结构存储数据时的几种操作说明如下：

- 添加元素：当要向该数据结构中添加一个键值对（key-value）时，先对key做哈希运算，哈希函数：hash = | 3*key^3+5*key^2+7*key+11 | ，上述公式中的 | 是绝对值符号，获取key的hash值，然后用hash值对数组table的长度length取模获取键值对应该存储的位置pos，公式为 pos = hash % length 。如果出现哈希冲突的情况，即计算出的位置pos已经存储了数据，则将键值对插入到当前位置已有的链表中，要求插入之后链表是按从小到大排序（按键排序）；如果没有出现哈希冲突，则在当前位置中保存一个单节点链表。
- 删除元素：按照和添加元素同样的逻辑获取对应的键值对所在的位置pos，然后在这个位置里的链表中剔除掉相应的链表节点，如果是单节点链表，则直接把当前位置的链表置为null。
- 扩容：

  在两种情况下需要进行扩容操作：

  1. HashDict中节点个数超过了现有数组长度（不包括相等）
  2. 向某一个桶中的链表增加元素后，该链表长度超过4（不包括4）

  每次扩容操作是将数组长度变为之前数组的**两倍+1**（如原来长度为8，扩容后为17），并将原有的键值对**按照添加元素的规则（重新计算hash值取模）重新添加到新的数组中。**
- 查询：查询数组的指定位置存储了哪些键值对。

输入格式：

首先输入一个数字L，L代表数组table的初始长度。

然后输入一个数字N，N代表操作次数，下面N行是具体的操作。

操作行的输入格式：

- 添加元素：`add [key] [value]`，add 代表该行执行添加操作，[key]和[value]是键值对的相应值。如`add 1 cpp`代表向HashDict中添加key为1，value为cpp的一个键值对。
- 删除元素：`delete [key]`， delete 代表该行执行删除操作，[key]是要删除的键值对的键值。保证这个键值一定在HashDict中已经存在。
- 查询：`search [pos]`，search代表执行查询操作，[pos]代表要查询的数组位置，需要输出该位置的链表。保证pos小于数组table的长度。如`search 0`代表查询数组table第一个位置中存储了哪些键值对。

输出格式:

只有查询操作需要输出，如果查询位置没有键值对，则直接输出null，如果有，则按照

`[key]:[value]->[key]:[value]`的格式输出（参考示例）。

示例1:

输入:

```
4
4
add 10 cpp
add 5 cat
add 3 dog
search 2
```

输出:

```
3:dog->5:cat
```

示例2:

输入:

```
2
11
add 5 cat
add 3 dog
search 0
add 10 cpp
search 0
search 1
add 7 bird
add 17 pig
search 4
delete 7
search 4
```

输出:

```
3:dog->5:cat
null
5:cat->10:cpp
7:bird->17:pig
17:pig
```

示例3：

输入：

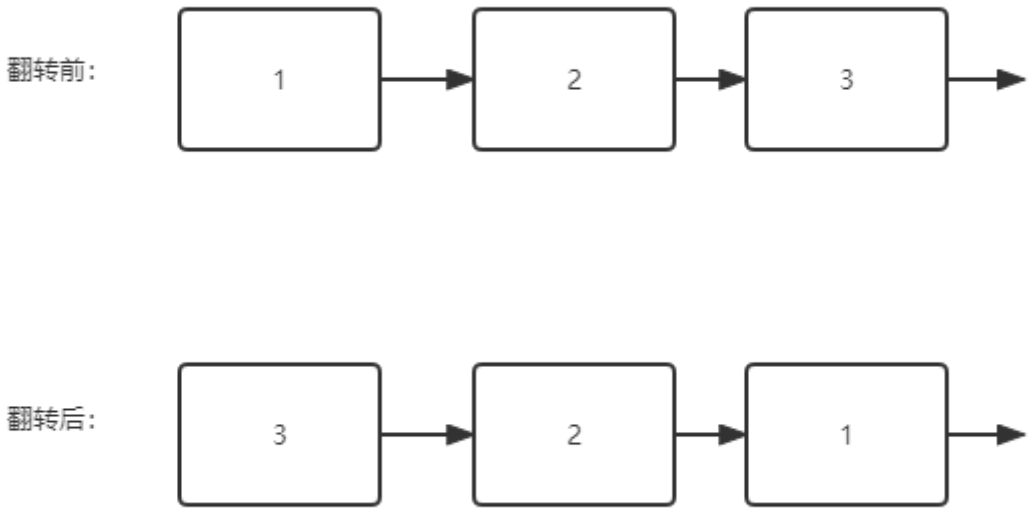
```
4
10
add 5 cat
add 3 dog
add 7 cat1
add 11 dog1
search 2
add 9 cpp
search 2
search 5
search 6
search 8
```

输出：

```
3:dog->5:cat->7:cat1->11:dog1
7:cat1->9:cpp
3:dog
5:cat->11:dog1
null
```

翻转链表

请用**链表**实现本题，图示如下。



链表节点的值是`int`类型且链表长度一定大于0。

输入描述

未知数量的`int`类型的数，空格隔开。

输出描述

输出翻转后的链表从头到尾的节点值，空格隔开。

示例

输入

```
1 2 3
```

输出

```
3 2 1
```