**inverted index look up document**

**follow up 1, needs to support delete**

**follow up 2, ask what happen to multiple thread try to add and delete at same time, how to prevent issue**

**主问题答案：Search + Add**

用 HashMap<String, Set<Integer>> 来存储倒排索引，key 是单词，value 是包含该单词的文档集合。

```
import java.util.*;

class InvertedIndex {

  private Map<String, Set<Integer>> index;

  public InvertedIndex() {

    index = new HashMap<>();

  }


  // 添加文档

  public void add(int docId, String content) {

    String[] words = content.split("\\s+");

    for (String word : words) {

      index.computeIfAbsent(word, k -> new HashSet<>()).add(docId);

    }

  }


  // 搜索文档

  public Set<Integer> search(String word) {

    return index.getOrDefault(word, new HashSet<>());
    }
}
```

**Follow-up 1: Delete**

实现 delete 方法时，要注意：

1. 可能文档里有的单词不在 index 里 → 要做检查
2. 删除后如果单词集合为空，要清理掉该单词的 key

```
    // 删除文档
    public void delete(int docId, String content) {
        String[] words = content.split("\\s+");
        for (String word : words) {
            if (index.containsKey(word)) {
                Set<Integer> set = index.get(word);
                set.remove(docId);
                if (set.isEmpty()) {
                    index.remove(word);
                }
            }
        }
    }
```

**Follow-up 2: 多线程 add/delete**

问题：

- HashMap 和 HashSet 不是线程安全的
- 多线程并发修改会导致 **数据丢失** 或 **ConcurrentModificationException**

## 使用并发数据结构

- ConcurrentHashMap<String, Set<Integer>>
- 里面的集合用 ConcurrentHashMap.newKeySet()

```
import java.util.concurrent.*;

class ConcurrentInvertedIndex {
  private ConcurrentHashMap<String, Set<Integer>> index;

  public ConcurrentInvertedIndex() {
    index = new ConcurrentHashMap<>();
  }

  public void add(int docId, String content) {
    for (String word : content.split("\\s+")) {
      index.computeIfAbsent(word, k -> ConcurrentHashMap.newKeySet()).add(docId);
    }
  }

  public Set<Integer> search(String word) {
    return index.getOrDefault(word, ConcurrentHashMap.newKeySet());
  }

  public void delete(int docId, String content) {
    for (String word : content.split("\\s+")) {
      Set<Integer> set = index.get(word);
      if (set != null) {
```

```
                set.remove(docId);
                if (set.isEmpty()) {
                    index.remove(word, set);
                }
            }
        }
    }
}
```

**主问题：Search + Add**

- **add(doc)**: 时间 O(k)（k = 文档单词数），空间 O(TotalWords + TotalPostings)
- **search(word)**: 时间 O(1 + r)（r = 含该词的文档数），空间 O(r)（如果复制结果集）

**Follow-up 1: Delete**

- **delete(doc)**: 时间 O(k)，空间 O(1)

**Follow-up 2: 并发 add/delete**

- **add/delete**: 时间 O(k)，空间 O(TotalWords + TotalPostings)（和单线程一致，额外开销是锁/CAS）
- **search**: 时间 O(1 + r)，空间 O(r)

**在一个社交媒体网络中，每个用户都有其好友列表。给定一个表示社交关系的图（用一个 Map<Integer, List<Integer>> 表示，其中键是用户 ID，值是该用户的直接好友列表），以及一个特定用户 ID，请为该用户推荐一个“最佳新朋友”。推荐的规则是：优先推荐：与当前用户有最多共同好友的用户。次级规则：如果共同好友数相同，推荐用户 ID 较小的用户。限制条件：不能推荐当前用户自己，且不能推荐当前用户的直接好友。**

**输入：connections：一个映射，表示每个用户及其直接好友。person：目标用户 ID。**

**输出：返回推荐的新朋友的用户 ID。**

```
import java.util.*;
/**
 * Recommend a "new friend" for `person`.
 * Rule 1: maximize number of mutual friends with `person`
 * Rule 2: tie-break by smallest user ID
 * Constraints: cannot recommend `person` themself, nor any direct friend of `person`
 *
 * Time:  O(d * f) on average (d = #direct friends of person, f = their avg friend count), worst-case O(n^2)
 * Space: O(n) for the mutual-count map and direct-friends set
 */
public class Solution {
  public static int bestRecommendation(Map<Integer, List<Integer>> connections, int person) {
    // Person's direct friends (O(1) lookups)
    Set<Integer> direct = new HashSet<>(connections.getOrDefault(person, Collections.emptyList()));

    // Count mutual friends for each valid candidate (friend-of-friend who is not self and not a direct friend)
    Map<Integer, Integer> mutualCount = new HashMap<>();
    for (int friend : direct) {
      for (int fof : connections.getOrDefault(friend, Collections.emptyList())) {
        if (fof == person || direct.contains(fof)) continue;
        mutualCount.put(fof, mutualCount.getOrDefault(fof, 0) + 1);
      }
    }

    // Select candidate with max mutual-count; if tie, smaller ID wins
    int best = -1;
    int bestCount = -1;
    for (Map.Entry<Integer, Integer> e : mutualCount.entrySet()) {
      int id = e.getKey(), cnt = e.getValue();
      if (cnt > bestCount || (cnt == bestCount && id < best)) {
        best = id;
        bestCount = cnt;
      }
    }

    // If no candidate exists, return -1
    return best;
  }
}
```

**基本就是 leetcode 1286 但是是两个 iterator merge 起来的 next 和 hasnext**

**union iterator，给两个 iterator, 要求实现 hasNext() and next()， followup 是给 n 个 iterator，怎么改代码**

### 第一问：两个 Iterator

```
import java.util.Iterator;

import java.util.NoSuchElementException;


public class UnionIterator<T extends Comparable<T>> implements Iterator<T> {

    private Iterator<T> it1;

    private Iterator<T> it2;

    private T next1;

    private T next2;


    public UnionIterator(Iterator<T> it1, Iterator<T> it2) {

        this.it1 = it1;

        this.it2 = it2;

        this.next1 = it1.hasNext() ? it1.next() : null;

        this.next2 = it2.hasNext() ? it2.next() : null;

    }


    @Override

    public boolean hasNext() {

        return next1 != null || next2 != null;

    }
    @Override
    public T next() {
        if (!hasNext()) throw new NoSuchElementException();

        T result;
        if (next1 == null) {
            result = next2;
            next2 = it2.hasNext() ? it2.next() : null;
        } else if (next2 == null) {
            result = next1;
            next1 = it1.hasNext() ? it1.next() : null;
        } else if (next1.compareTo(next2) <= 0) {
            result = next1;
            next1 = it1.hasNext() ? it1.next() : null;
        } else {
            result = next2;
            next2 = it2.hasNext() ? it2.next() : null;
        }
        return result;
    }
}
```

**Follow-up：扩展到 N 个 Iterator**

思路：维护一个 **最小堆 (PriorityQueue)**，每次从堆里取最小值，然后把该 iterator 的下一个元素放回堆中。
堆中存 (当前值, 来自哪个 iterator, 对应的 iterator)。

```
import java.util.*;

public class MultiUnionIterator<T extends Comparable<T>> implements Iterator<T> {
    private static class Node<T> {
        T value;
        Iterator<T> iterator;

        Node(T value, Iterator<T> iterator) {
            this.value = value;
            this.iterator = iterator;
        }
    }

    private PriorityQueue<Node<T>> pq;

    public MultiUnionIterator(List<Iterator<T>> iterators) {
        pq = new PriorityQueue<>(Comparator.comparing(n -> n.value));
        for (Iterator<T> it : iterators) {
            if (it.hasNext()) {
                pq.offer(new Node<>(it.next(), it));
            }
        }
    }

    @Override
    public boolean hasNext() {
```

```
    return !pq.isEmpty();
  }

  @Override
  public T next() {
    if (!hasNext()) throw new NoSuchElementException();
    Node<T> node = pq.poll();
    T result = node.value;
    if (node.iterator.hasNext()) {
      pq.offer(new Node<>(node.iterator.next(), node.iterator));
    }
    return result;
  }
```

### 两个 Iterator 的 UnionIterator

- **时间复杂度**
  - hasNext() → **O(1)**
  - next() → **O(1)**
  - 总体遍历所有元素时：**O(m + n)**，其中 m 和 n 是两个迭代器的长度。
- **空间复杂度**
  - 只存 next1、next2 两个指针，外加迭代器本身 → **O(1)**

---

### n 个 Iterator 的 MultiUnionIterator

- **时间复杂度**
  - hasNext() → **O(1)** （只检查堆是否为空）
  - next() → 堆 poll + 可能的 offer → **O(log n)**

    - 总体遍历所有元素时：**O(N log n)**，其中N是所有元素总数。
- **空间复杂度**
    - 需要维护一个大小为n的最小堆（每个迭代器最多存一个元素） → **O(n)**
    - 额外空间不依赖于元素总数N。