Programming Part

Problem 3 - Printing Queue (100 pts)

Problem Description

In this problem, we ask you to design a system to emulate the job queues of several printers within the *Computer Science and Information Engineering* (CSIE) department of *National Taiwan University* (NTU). Yes, we have more natural acronyms too! :-) This system needs to support the following three operations:

- Add Operation: Insert a job into the printer's queue.
- Print Operation: Print the document with the highest priority from the queue. This is a typical need for the priority queue (i.e. what the binary heap is designed for).
- Move Operation: Transfer all jobs from one printer to another. This happens when one printer "fails" for some reason, such as running out of paper or toner.

Although the add and print operations can be done efficiently with the binary heap data structure that we taught in class, the binary heap cannot execute the move operation very efficiently. In particular, the move operation takes $O(n \log(m+n))$ with the binary heap when moving n tasks to a printer that carries m tasks by running n insertions naïvely.

To design the system efficiently, we suggest you to learn related data structures that can handle the move operation efficiently. One possibility is the binomial heap (but there are also other possibilities that you can consider). The following description of the binomial heap is modified from Wikipedia (`https://en.wikipedia.org/wiki/Binomial_heap`):

A binomial min-heap is implemented as a set of binomial min-trees. Each binomial tree is defined recursively as follows:

- *(termination) binomial tree of order* 0 *is a single node.*
- *(recursion) A binomial tree of order* k *has a root node whose children are roots of binomial trees of orders* $k-1$, $k-2$, $\ldots$, 2, 1, 0, *usually sorted reversely by its order.*

Then, the binomial min-heap can be defined as a set of binomial trees, as follows:

- *There can be at most one binomial tree for each order, including the zero-th order.*
- *Each binomial tree is a min-tree. That is, the key of the root of any sub-tree is smaller than or equal to the key of any of its descendants.*

The second property ensures that the root of each binomial tree contains the smallest key in the tree. It follows that the smallest key in the entire heap is one of the roots.

The first property implies that a binomial heap with n nodes consists of at most $1 + \log_2 n$ binomial trees (so locating the minimum key takes $O(\log n)$ time). The number and orders of these trees are uniquely determined by the number of nodes n: there is one binomial tree for each nonzero bit in the binary representation of the number n. For example, the decimal number 13 is $(1101)_2$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0.

The core of the binomial heap is an efficient BinomialHeapUnion algorithm that merges two binomial heaps. The pseudo code (modified from an earlier version of the textbook) of the algorithm is listed in the next page. The algorithm contains an additional step of BinomialHeapMerge that merges the ordered linked list of binomial trees within binomial heaps H_1 and H_2 into another ordered linked list. The step is similar to the sparse vector summing algorithm that you have learned in Lecture 6. :-) Once you have implemented BinomialHeapUnion, you can conduct the following operations efficiently.

BinomialHeapInsert(H, x)

To insert a node x into a binomial heap H, you can create a new binomial heap H' containing only the node x, and then run BinomialHeapUnion(H, H').

BinomialHeapExtractMin(H)

To extract the node with the minimum priority, you can first find the binomial tree T with the minimum key in the root list of H. Then, you can remove T from the root list of H, remove $T.root$ from T, and then run BinomialHeapUnion$(H, \text{Reverse}(T.children))$.

```
BinomialHeapUnion($H_1, H_2$)
 1  $H$ = MakeEmptyBinomialHeap()
 2  $head[H]$ = BinomialHeapMerge($H_1, H_2$)
 3  if $head[H]$ = NIL
 4      return $H$
 5  $prev$ = NIL
 6  $x = head[H]$
 7  $next = sibling[x]$
 8  while $next \neq$ NIL
 9      if ($degree[x] \neq degree[next]$) or ($sibling[next] \neq$ NIL and $degree[sibling[next]] = degree[x]$)
10          $prev = x$
11          $x = next$
12      else
13          if $key[x] \leq key[next]$
14              $sibling[x] = sibling[next]$
15              insert $next$ as a child of $x$, and update $degree[x]$
16          else
17              if $prev$ = NIL
18                  $head[H] = next$
19              else
20                  $sibling[prev] = next$
21              insert $x$ as a child of $next$, and update $degree[next]$
22              $x = next$
23      $next = sibling[x]$
24  return $H$
```

In this problem, we suggest you implement the binomial *max-heap* instead of the min-heap. That is, the key of the root of any sub-tree is greater than or equal to the key of any of its descendants.

Input

Assuming that the printers' queues are initially empty, the input begins with two space-separated integers N and M, representing the number of printers and the total number of operations, respectively. Printers are 1-indexed. Each of the next M lines is given in one of the following formats:

- `1 job_id priority printer_id`: add operation
  Insert a job with `job_id` into the queue of printer `printer_id`, assigning it the priority `priority`.
- `2 printer_id`: print operation
  Instruct printer `printer_id` to print the document with the highest priority from its queue, removing the job from the queue in the process.
- `3 printer_id1 printer_id2`: move operation
  Transfer all jobs from printer `printer_id1` to printer `printer_id2`, maintaining their respective priorities.

Output

- For each **add** operation, print a line that indicates the number of jobs waiting on printer `printer_id` in the format:

  ```
  [num_of_jobs] jobs waiting on printer [printer_id]
  ```

- For each **print** operation, if there is a job to be printed, print a line of

  ```
  [job_id] printed
  ```

  Otherwise (the queue is empty), print a line of

  ```
  no documents in queue
  ```

- For each **move** operation, print a line that indicates the number of jobs waiting on printer `printer_id2` after moving.

  ```
  [num_of_jobs] jobs waiting on printer [printer_id2] after moving
  ```

Constraints

- $1 \leq N, M \leq 10^6$
- $1 \leq$ `printer_id, printer_id1, printer_id2` $\leq N$
- $1 \leq$ `job_id, priority` $\leq 10^9$
- For the **move** operation, `printer_id1` $\neq$ `printer_id2`
- It is guaranteed that all `priority` and `job_id` will be distinct, and all operations will be legal (i.e., all IDs will be within the specified range).

Subtasks

Subtask 1 (20 pts)

- $N = 1$
- Naturally, there are no move operations when $N = 1$. That is, the task can be solved with the binary heap taught in class.

Subtask 2 (20 pts)

- $1 \leq N, M \leq 5000$

Subtask 3 (60 pts)

- no other constraints

Sample Testcases

Sample Input 1

```
1 5
1 1 1 1
1 2 2 1
2 1
2 1
2 1
```

Sample Output 1

```

1 jobs waiting on printer 1
2 jobs waiting on printer 1
2 printed
1 printed
no documents in queue
```

Sample Input 2

```
2 15
1 1 5 1
1 2 4 1
1 3 3 1
1 4 2 2
1 5 1 2
3 1 2
2 1
2 2
2 2
3 2 1
2 2
2 1
2 1
2 1
2 1
```

Sample Output 2

```

1 jobs waiting on printer 1
2 jobs waiting on printer 1
3 jobs waiting on printer 1
1 jobs waiting on printer 2
2 jobs waiting on printer 2
5 jobs waiting on printer 2 after moving
no documents in queue
1 printed
2 printed
3 jobs waiting on printer 1 after moving
no documents in queue
3 printed
4 printed
5 printed
no documents in queue
```