University of Waterloo
CS240E, Winter 2025
Programming Question 1

Due Date: Tuesday, February 11, 2025 at 5pm
Grace period: Submissions made before 11:59PM on Feb. 11 will be accepted without penalty.

a) [20+8+8=36 marks]

Implement three realizations of a priority queue that supports *insert*, *delete-max* and *merge*:

1. Realization 1 is a *meldable heap*, as seen in class. For testing purposes, your implementation should *not* use a random coin toss; instead each of our operations will pass you a bitstring, and you should read the bits to use from it.
2. Realization 2 is a deterministic version of a meldable heap that determines the place to merge by height.
3. Realization 3 is a deterministic version of a meldable heap that determines the place to merge by size.

See further explanations on the deterministic versions below. You should maintain six heaps at all times: Two with realization 1, two with realization 2, and two with realization 3. The operations listed below will specify to which heaps in which realizations they apply.

Submit: A C++ program `mergeHeaps.cpp` that provides a main function that accepts the following commands from stdin:

- `i` k h r $\mathbf{w}$, where k, h, r are integers (with $1 \leq h \leq 2$ and $1 \leq r \leq 3$) and $\mathbf{w}$ is a bitstring. This should insert the key with ID-number k into heap h of realization r. (You will store keys using the class `Key`; more on this below.)

  For realization 1, the insertion-routine should use bitstring $\mathbf{w}$ to determine "random" bits, i.e., the first call to *random*(2) should be replaced by $\mathbf{w}[0]$, the next one should use $\mathbf{w}[1]$, etc. For realizations 2 and 3, bitstring $\mathbf{w}$ has no influence on the resulting structure and should be ignored entirely.
- `d` h r $\mathbf{w}$ should remove the maximum key that is currently in heap h of realization r. The restrictions and meaning of h, r and $\mathbf{w}$ are as for `i`.
- `m` r $\mathbf{w}$ should merge heap 1 and heap 2 of realization r. The result of the merge should be in heap 1, while heap 2 becomes an empty heap. The restrictions and meaning of r and $\mathbf{w}$ are as for `i`.

– `p` h r should go to heap h of realization r, and print the stored ID-numbers in *pre-order*, i.e., first the ID-number at the root, then (recursively) all ID-numbers in the left subtree, then all ID-numbers in the right subtree. The restrictions on h and r are as for `i`.

  Print the ID-numbers on one line, space-separated (or print an empty line if the heap is empty). The operation should not change the structure of the heaps.

– `n` h r does the same as the previous command, except now the ID-numbers stored in the heap are printed in *in-order*, i.e., first (recursively) all ID-numbers in the left subtree, then the ID-number at the root, then the ID-numbers in the right subtree.

– `x` terminates the program.

Accessing and comparing keys: You will *not* store keys as `int`s or any other integer type. Instead, you will store instances of the class `Key`, initialized by an ID-number that identifies the key uniquely (and this is what you should construct the `Key` with). The class definition is given in `key.h`, which should be `#include`d by `mergeHeaps.cpp`.

In order to merge two heaps, you need to do key-comparisons. You must do this via the overloaded `<` operator for class `Key` that is provided for you. You can retrieve a `Key`'s ID number to print for commands `p` and `n` by using `Key::get_id`.

There is no particular relationship between the ID-number of a key and the actual key itself (e.g. while $3 < 4$, we do not necessarily have `Key(3) < Key(4)`). To aid debugging, for the public tests the ID-number behaves the same as the `Key` itself. This is not the case for secret tests.

Note also that `Key` has a function to count the total number comparisons performed so far. You may find this useful when doing experiments for parts (b) and (c), but you do not need to use it.

Deterministic meldable heap: To make a meldable heap deterministic, one stores extra information at each node, and uses it to determine where to merge. Two natural ideas come to mind.

In the first one (used by Realization 2) each node z of the heap has a field *height* that equals the height of the subtree rooted at z. (You will need to update this field when merging another heap M into the subtree rooted at z; details are for you to figure out.) The choice of where to merge M is then done by the height at the children: If one child of z has strictly smaller height then the other, then merge M with the sub-heap at this child. If both children of z have the same height, then merge M with the sub-heap at the left child.

The second idea (used by Realization 3) is exactly the same, except that each node z stores *size*, i.e., the number of nodes (including z) in the subtree rooted at z. You then

merge with the sub-heap at the child that has strictly smaller size, or if the sizes are equal, use the left child.

Further specifications and assumptions:

- There is no length-limit on bitstring **w**—use the standard library class `std::string` to store it. We promise that it will always be long enough to have enough bits for the operation (but it may well be longer).
- We will do no illegal operations, i.e., parameters will be as specified above. We will never ask to remove the maximum from an empty heap. Also, ID-numbers will be distinct, but the (unknown) keys that they correspond to need not be distinct.
- "Printing on one line" means that the line must end with a newline. Trailing whitespace at the end of your output lines will be ignored by our test scripts.
- Note that the trees in your heaps are unique if the keys are unique. We will verify their structure using commands `p` and `n`, so it is important that you follow the pseudo-code from class (as well as the instructions for Realization 2 and 3) exactly.
- Partial marks can be obtained if you only implement some of the realizations correctly. The realization that passes the most tests will receive up to 20 marks, the other two realizations will receive up to 8 marks each.
- You are *allowed* to use `std::vector`, `std::list`, `std::string` and `std::stringstream` from the standard library, as well as any classes not relevant to CS240 material (see the assignment guidelines).

b) [4 marks]

Let the *x-insertion-cost* of a meldable heap T by the number of key-comparisons that are used during *insert*(x). (This depends on the structure of T, the value of key x, and the strategy for where to merge.) For the following question, let T_h be a deterministic meldable heap with realization 2 (i.e., it decides where to merge by height), while T_s is a deterministic meldable heap with realization 3. Which of the (mutually exclusive) following statements is true?

1. The insertion-costs of T_h and T_s are the same. Formally, for any sequence of operations *insert* and *merge*, followed by *insert*(x) for some key x, trees T_h and T_s have the same x-insertion-cost.
2. The insertion-costs are better for T_h than for T_s. Formally, for any sequence of operations *insert* and *merge*, followed by *insert*(x) for some key x, the x-insertion cost for tree T_h is never worse than the one of T_s, and for some sequences and/or keys it is strictly better.
3. The insertion-costs are better for T_s than for T_h. Formally, for any sequence of operations *insert* and *merge*, followed by *insert*(x) for some key x, the x-insertion

cost for tree T_s is never worse than the one of T_h, and for some sequences and/or keys it is strictly better.

4. The insertion-costs of T_h and T_s are incomparable. Formally, there exists a sequence of operations *insert* and *merge*, followed by *insert*(x) for some key x, such that the x-insertion cost for tree T_s is strictly better than the one of T_h. There also exists such a sequence of operations such that the x-insertion cost for tree T_s is strictly worse than the one of T_h.

You need not justify your answer. You are strongly encouraged to determine via experiments with your own implementation what the correct answer is (or at least to rule out some incorrect ones).

To give us the answer, please add one more option to your main-routine:

– `b`—this keeps the heaps unchanged, and writes a single line to the output that contains exactly one of the numbers $1, 2, 3, 4$, corresponding to the above answer.

c) (**Bonus**, up to 10 marks)

Create your own meldable-heap realization (call it "realization 4") with the goal of keeping the x-insertion-cost as small as possible. The 10 bonus-marks will be awarded as a competition; the submission that has the smallest insertion-cost on average over our test set will receive 10 bonus-marks, the next-best will receive 9 bonus-marks, etc. We will publish no details of our test set; you will need to design your own experimental set and hope for the best.

There is no specific "correct" approach for this; the main question is the choice of strategy when deciding where to merge. You are strongly encouraged to use experiments with your own implementation to see what works best. Your realization must support all operations listed in (a) (allowing $r = 4$ as parameter), and it must follow the general idea of a meldable heap; in particular during a *merge* you should only change links at nodes along two downward walks from the root, and the run-time should be proportional to the length of these walks.

The leaderboard will be updated regularly at: `https://student.cs.uwaterloo.ca/~cs240e/w25/leaderboard.phtml` Your most recent correct submission made before the due date is your entry on the leaderboard.