Introduction

Many times, data structures are extended beyond their fundamental operations. Recall that a heap is a priority queue that mainly provides a push/pop logic. It is, however, possible to extend this functionality. In fact, many algorithms require a heap that allows one to erase specific items. In this assignment, you will implement a binary min-heap that allows such erasures, which we call the *erasing min-heap* or shortly *e-heap*.

To allow erasures, each push operation on the e-heap returns a *handle* representing the inserted item. When we want to erase an item from the e-heap, we pass the item's handle to the heap's `erase` method. Of course, to be efficient, the handle should point to a piece of data that allows us to locate the item in the heap. This eliminates the need to search the heap to find the item to be erased.

Task

You are expected to complete the implementation of an e-heap, whose partial implementation is given below. The expected behavior of each public function is specified with a comment.

```
template<typename T>
class EHeap
{
    private:
        class HandleData
        {
            friend EHeap;
            private:
                /* Private fields & methods to be completed by you. */
        };

        /* Private fields & methods to be completed by you. */

    public:
        typedef HandleData * Handle;

        // Constructs an empty e-heap with a fixed capacity.
        EHeap(unsigned capacity) { /* To be completed by you. */ }

        // Returns the number of elements in the e-heap.
        unsigned size() const { /* To be completed by you. */ }

        // Pushes a value into the e-heap. A new handle
        // representing the pushed value is returned.
        Handle push(T val) { /* To be completed by you. */ }

        // Pops the minimum value (or one of the minimum) from the e-heap.
        T pop() { /* To be completed by you. */ }

        // Erases the value whose handle is given. Returns the value back.
        T erase(Handle handle) { /* To be completed by you. */ }

        // Destructor. All associated memory is to be freed.
        ~EHeap() { /* To be completed by you */ }
};
```

Further Instructions & Clarifications

- In your implementation, you can assume that
  - No `pop()` will be issued to an empty heap.
  - No `push()` will be issued to a heap at its capacity.
  - The handle passed to `erase()` is always associated with a valid element in the heap.
- The template parameter `T` will be guaranteed to support copy/default construction, assignment operator and comparison operators.
- You are free to add additional members to the `EHeap` and `HandleData` classes. You can also change the order of the members or add more access modifiers if necessary. However, you are *not allowed to modify their public interface*.
- Your implementation should *allow duplicate keys*. That is, it is possible that two distinct items `x` and `y` are pushed into the heap such that `x == y` is `true`. Of course, such items are treated as distinct and are popped/erased separately.
- We *prohibit the use any libraries*. The prohibition includes the standard C++ library. Your code should only make use of the constructs made available by the vanilla C++ programming language. This implies that you cannot use `vector`, `set`, `map` or any similar classes. You can, however, allocate dynamic objects or arrays with `new`.
- *Performance* is critical for this assignment: All methods should be reasonably well-implemented and run in worst-case $O(\log n)$ time, where n is the number of items in the heap. Some of the tests will be focusing on performance.
- For this assignment, you can copy code from the *sample codes* we shared in our course page in ODTÜClass. Of course, you are *NOT allowed to copy from other sources*; that would be considered cheating.
- You are free to use any method to implement the e-heap. However, in our *Wednesday lecture on 8th of June at 12:40*, the course instructor will describe how such an e-heap can be implemented. If you do not have a good and clean idea of how an e-heap can be implemented, please attend the lecture. It is best if you know about heaps before the lecture. You can watch the online lectures if you were not attending the lectures real-time.

Submission

Submission is through ODTÜClass. We will provide the original $EHeap.tpp$ file that contains the partial declaration of the `EHeap` class. We expect an updated and completed $EHeap.tpp$ from you that contains your e-heap implementation.

We will compile your code with `g++` using the options: `-std=c++2a -O3 -Wall -Wextra -Wpedantic`.

Grading

We will (in the coming days) provide an auto-grader that will evaluate your functions in black-box manner. To get full points, you need to implement all of the functionality mentioned above with the expected performance. You **may** get partial points even if you implemented a subset of the functions or failed to meet the performance expectations. However, the focus of this assignment is on the erase/handle logic. If you do not implement this logic properly, it is likely that you will lose majority of the points.

The grade you get from the auto-grader is not final. We may do additional black-box and white-box evaluations and adjust your grade. Solutions that do not reasonably attempt to solve the given task may lose points.

Continue below for an example.

Example

The following code:

```
#include <iostream>
using std::cout;
using std::endl;

#include "EHeap.tpp"

int main()
{
    EHeap<int> e(7);
    int values[7] = {70, 60, 50, 10, 40, 30, 20};
    EHeap<int>::Handle handles[7];

    for (int i = 0; i < 7; i++)
    {
        handles[i] = e.push(values[i]);
    }

    cout << e.size() << " items are pushed into the heap." << endl;

    cout << "Erased " << e.erase(handles[2]) << "." << endl;
    cout << "Popped " << e.pop() << "." << endl;
    cout << "Erased " << e.erase(handles[5]) << "." << endl;
    cout << "Popped " << e.pop() << "." << endl;
    cout << "Erased " << e.erase(handles[4]) << "." << endl;
    cout << "Popped " << e.pop() << "." << endl;
    cout << "Popped " << e.pop() << "." << endl;

    return 0;
}
```

is expected to produce the output:

```
7 items are pushed into the heap.
Erased 50.
Popped 10.
Erased 30.
Popped 20.
Erased 40.
Popped 60.
Popped 70.
```

Good Luck!