Homework 5: Binary Heap

In this assignment, you will implement a fixed-size binary heap. The structure of the heap is already defined for you in `binheap.rkt`. The elements of the heap are stored in a DSSL2 vector. Each heap also contains a comparison function for ordering the elements of the heap, so that your implementation can support heaps of integers, heaps of strings, heaps of whatsits, heaps of sporkles, etc.

You will also implement the *heap sort* algorithm using your binary heap, as well as write a small function which uses your heap sort.

The `BinaryHeap` class

Your `BinHeap` class must implement the priority queue abstract data type, as specified by this DSSL2 interface:

```
interface PRIORITY_QUEUE[X]:
    def len(self) -> nat?
    def find_min(self) -> X
    def remove_min(self) -> NoneC
    def insert(self, element: X) -> NoneC
```

That is, the `PRIORITY_QUEUE` interface is parameterized by an element type `X`, and declares four operations:

- The `len` method returns the number of elements in the priority queue.
- The `find_min` method returns the "smallest"[1] element of the priority queue, or calls `error` if it is empty. (If there is more than one element tied for smallest then it doesn't matter which of those is returned, so long as `find_min` is deterministic.)
- The `remove_min` method removes the smallest element of the priority queue, or calls `error` if it is empty. (If there is more than one element tied for smallest, then `remove_min` must be consistent with `find_min`, in the sense that it removes the same element that `find_min` would return.)
- The `insert` method takes a new element and adds it to the priority queue if there is sufficient space for an additional element; if there isn't room, then it calls `error` instead.

DSSL2 interfaces specify how objects can be used, but not how they are constructed. To understand that, we must describe the constructor of the `BinHeap` class. Like the `PRIORITY_QUEUE` interface that it implements, the `BinHeap` class is parameterized by an element type `X`; its constructor takes two parameters, `capacity` and `lt?`, as seen here:

[1] Through, when we say "smallest" we mean "smallest according to the ordering function the heap was created with", which may well implement some other order entirely. When the ordering function is greater-than then "smallest" actually means largest.

```
class BinHeap[X] (PRIORITY_QUEUE):
    # ... fields omitted ...

    def __init__(self, capacity, lt?):
        # ... your job ...

    # ... lots more ...
```

The value given for `capacity` must be a natural number, which will be the size of the vector used to store the elements. Thus, `capacity` determines the number of elements that can be stored before the `insert` method signals an error.

The purpose of the `lt?` parameter is to allow clients of the `BinHeap` class to specify what elements they want considered "least," that is, what ordering the heap should implement. A client does so by providing a function for the heap to use for comparing elements when restoring its invariant. If the client wants a min-heap then it will pass an actual less-than function for `lt?`, and that will cause the heap to return its smallest elements first; but if the client wants a max-heap then it can pass a greater-than function as `lt?` and it will get a largest-first heap. If a client wants some other order, or if it wants a bespoke element type, then it can provide a comparison function for that.

In general, the value given for `lt?` must be a binary predicate for comparing two elements of type `X`. In particular, given two elements (*i.e.,* `X`s) `a` and `b`, `lt?(a, b)` must return `True` if `a` is to be considered less than `b` for the purposes of ordering the heap, and it must return `False` if `a` is to be considered greater than or equal to `b`. The `BinHeap` class stores this function in a field and uses it to compare elements and maintain its invariant.

Thus, when constructing a heap, we need to provide a heap ordering predicate appropriate for the intended element type and order. For example, we could construct a heap for holding up to 20 integers like this:

```
let h = BinHeap(20, lambda a, b: a < b)
```

The `lambda` given as the second argument is a two argument function that compares its arguments using DSSL2's built-in `<` operator. Since that operator works on strings as well, we can construct a heap of strings, ordered lexicographically, the same way. To construct a heap containing other kinds of objects (shirts, songs, etc.), we'd need to define a custom comparison functions which compares these kinds of objects.

And as discussed above, supplying a heap ordering predicate also lets us make a heap that returns the largest element first instead of the smallest.

```
let h = BinHeap(10, lambda a, b: a > b)
```

In the literature, not everyone agrees on whether a "heap," order unspecified, is smallest-first or largest-first. It can be confusing. The heap you are implementing can be used as either a smallest-first "min-heap" or a largest-first "max-heap,"

depending on the predicate given for `lt?`. When you are implementing it, however, you will do best if you (1) think of it as a min-heap, where `lt?` means "less than," and (2) never assume you know how to compare elements any other way, because the client gets to determine the element type and order.

Your task

First, you should complete the definition of the `BinHeap` class, by implementing the constructor and the four required methods from the interface:

1. Complete the definition of the constructor (`__init__`). $\mathcal{O}(n)$ time
2. Complete the definition of method `len`. $\mathcal{O}(1)$ time
3. Complete the definition of method `find_min`. $\mathcal{O}(1)$ time
4. Complete the definition of method `remove_min`. $\mathcal{O}(\log n)$ time
5. Complete the definition of method `insert`. $\mathcal{O}(\log n)$ time

Note the required asymptotic time complexities of the operations.

I have provided a puny test for the `BinHeap` class, but you will need many more.

Advice

- Strictly speaking, data in the vector past `_size` is not technically part of the binary heap proper, and can be ignored. However, when debugging, you can avoid confusion by setting unreachable vector elements to `None` when removing from the heap.
- Beware, though: a `None` element in the vector doesn't mean that it is past the end of the binary heap's contents! Whoever is using your binary heap could have inserted a `None` intentionally. And that's ok, so long as they provided an `lt?` function that can accept it.
- Your code will be much more readable and manageable if you break out some of the interface methods' logic into helper functions.

  Our presentation of binary heaps in class described the larger operations in terms of smaller suboperations: bubbling up, percolating down, finding parents, finding children, etc. These suboperations have a meaning in their own right, and so are good candidates for helpers.

  At a finer-grained level, some of these suboperations rely on more basic building blocks. For example, both bubbling up and percolating down involve *swapping* elements of the vector. I highly recommend abstracting that work into a helper function: swapping is easy to get wrong, so best to get it right just once.

Heap sort

Earlier in the quarter, we have seen three sorting algorithms: selection sort, merge sort, and quick sort. There are many other sorting algorithms out there, including one which relies on binary heaps: *heap sort*.

The essence of heap sort is as follows. To sort a vector, first `insert` all the elements of the vector in a binary heap, one by one. Then, take them out one by one using `find_min` and `remove_min` and put them back in the vector starting at index 0, and going in increasing index order.

The first element to be taken out of the heap will be the smallest element from the original vector; after all, all the elements were added to the heap, and it was the one returned by `find_min`. The next one will be the second smallest; the smallest has already been removed. And so on, resulting in a sorted vector.

Heap sort, like merge sort, has complexity $\mathcal{O}(n \log n)$: it does n `insert`s followed by n `remove_min`s, both of which are $\mathcal{O}(\log n)$ apiece.

Aside Strictly speaking, the *true* heap sort is an *in-place* sorting algorithm: it can sort a vector without requiring any auxiliary storage. That can be done by treating the vector itself as a binary heap, with some additional subtleties. That version is really cool, but beyond the scope of this class.

Your task

Second, you must implement sorting, using your `BinHeap` class and the algorithm described above.

6. Complete the definition of the `heap_sort` function. $\mathcal{O}(n \log n)$ time

Note the required asymptotic time complexity of the function.

Your heap sort, like any code that uses your binary heap, must rely solely on the methods that are part of the `PRIORITY_QUEUE` interface, and you cannot add additional methods to the interface. Good news: these methods are all you need.

I have provided a small test for the `heap_sort`, but you almost certainly should write more. It's relatively easy to come up with vectors to sort, and that's a low-effort, high-impact way to exercise your whole heap implementation.

Advice If you are seeing failures in your heap sort, don't rule out the possibility that the cause may actually be in your binary heap. Heap sort uses the binary heap, so if the binary heap is broken, the heap sort will be too. And you could be spending a long time debugging your (actually correct) heap sort! As usual, the solution is testing: before attempting heap sort, make sure you're confident in the correctness of your binary heap.

Using heap sort

Your `heap_sort` function, just like your `BinaryHeap` class, should work on any kind of data, provided you can write a comparison function for it. Let's put that to the test by sorting people by their birthday. Complete the `earliest_birthday` function to sort a vector of at least five people of your choice (constructed using the provided `person` struct) by their birthday using your `heap_sort` function. Your `earliest_birthday` function should construct the vector, sort it, and return the name of the person whose birthday is the earliest in the year.

In case you're not inspired, you can go with some members of my family:

- Padmavathi: September 24th
- Satyasai: December 10th
- Krishna: April 15th
- Swati: August 20th
- Swaroop: June 30th
- Hemant: July 30th

Your task

7. Write the `earliest_birthday` function.

Honor code

Every homework assignment you hand in must begin with the following definition (taken from the Provost's website[2]; see that for a more detailed explanation of these points):

```
let eight_principles = ["Know your rights.",
     "Acknowledge your sources.",
     "Protect your work.",
     "Avoid suspicion.",
     "Do your own work.",
     "Never falsify a record or permit another person to do so.",
     "Never fabricate data, citations, or experimental results.",
     "Always tell the truth when discussing your work with your instructor."]
```

If the definition is not present, you receive no credit for the assignment.

Note: Be careful about formatting the above in your source code! Depending on your pdf reader, directly copy-pasting may not yield valid DSSL2 formatting. To avoid surprises, copy a working version of the eight principles from one of your prior submissions and be sure to test your code *after* copying the above definition.

[2] `http://www.northwestern.edu/provost/students/integrity/rules.html`

Grading

Please submit your completed version of `binheap.rkt`, containing:

- the `BinHeap` class, with your definitions for the constructor and all four required methods,
- the `heap_sort` and `earliest_birthday` functions fully defined,
- sufficient tests to be confident of your code's correctness,
- and the honor code.

Functional Correctness

We will use four separate test suites to test your submission:

- **Basic binheap**: insertion and removal with `<` as comparator.
- **Advanced binheap**: error cases, custom comparators and element types, and no more calls to comparator than strictly necessary.
- **Basic heapsort**: sorting numbers in ascending and descending order.
- **Advanced heapsort**: sorting with duplicates, and sorting with custom comparators.

To get credit for a test suite, your submission must pass *all* its tests.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all four test suites.
- **Almost there**: passes both basic test suites, and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve "on the way" requirements.
- **Cannot assess**: we could not successfully run our grading tests on your submission (usually due to a crash), which also means *we could not give you feedback*. **Please run your code before submitting!**

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Thorough testing, including edge cases
- Efficiency from the use of the correct representation and operations
- Reuse of code (*i.e.*, not copy-paste) where pertinent

As part of the self-evaluation we may also dig into non-technical components (i.e., ethical considerations) of designing prioritization or sorting systems in general. This would not relate to your code directly and as such does not require anything on your part in your code submission.