HW5: Binary Heap

Due: Wednesday, March 6, at 11:59 PM, on Canvas

You must work on your own for this assigment.

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 whatits, heaps of sporkles, etc.

Interface overview

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) -> VoidC
    def insert(self, element: X) -> VoidC
```

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 element of the priority queue, or calls `error` if it is empty. (If there is more than one element tied for smallest, it does not matter which 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's sufficient space for an additional element; if there isn't space, it calls `error`.

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:

```
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 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 heap implementation then stores this function and uses it to compare elements and maintain its invariant.

Thus, when constructing a heap, we need to provide an 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.

But what if we wanted to construct a heap of vertex–distance pairs, as used

in Dijkstra's algorithm? Then we'd need to pass `BinHeap` a function for comparing such things:

```
struct VertexWithDistance:
    let vertex: nat?
    let distance: num?

let h = BinHeap(n, lambda a, b: a.distance < b.distance)
```

Finally, supplying a heap ordering predicate lets us make a heap that returns the largest element first instead of the smallest.

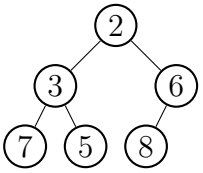
```
let h = BinHeap(10, lambda a, b: a > b)
```

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 to just think of it as a min-heap, where `lt?` means "less than."

Representation overview

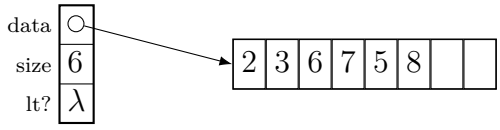
A binary heap is a *complete* binary tree satisfying the *heap property*. Completeness means that each level is full except the last one, which is filled from the left; the heap property means that each node's value is less than or equal to its children's values, according to the heap ordering predicate (`lt?`).

Here is an example of a binary heap of integers drawn as a tree:



In practice, a binary heap is rarely represented as actual tree made of links and nodes. Instead, the binary tree completeness property means that the elements can be stored in level order in an array, which is more space- and cache-efficient. In particular, in this assignment we will represent a heap as

a class with three fields. Here is the same heap as above in our actual heap representation:



The first field, `data`, stores a reference to the vector holding the elements. In order to allow the heap to grow, we do not necessarily use the full capacity of the vector; instead we store the actual number of elements in `size`, the second field. The remainder of the vector is then filled with some dummy value, such as `False`, depending on the language.

The third field, `lt?`, stores the heap ordering predicate that was given to the `BinHeap` constructor. This is necessary because your heap must work with any element type whatsoever, not just with numbers or strings. If elements were only numbers or strings, you could just compare them using DSSL2's built-in `<` operator, but that won't work for, say, bank account structs, employees, or weighted graph edges.

When restoring the invariant after insertion or removal, you have to compare elements. In particular, at each step, you will be considering some elements `parent` and `child`, where `parent` is directly above `child` in the heap. The invariant says that `parent` must be less than or equal to `child`. So, the invariant is broken if `child` is less than `parent`. Assuming we are in a method where the `BinHeap` object is named `self`, the invariant is broken if `self.lt?(child, parent)`.

Algorithms

Heap insertion and deletion both involve breaking and then restoring the heap invariant. Each requires a different algorithm for restoring the invariant.

Insertion

To insert into the heap, first we place the new element at the next available position in the vector, which is `self.size`, and we increment `self.size`.

Then we must restore the invariant. Since the new element is at the bottom of the tree, it can only possibly move up; hence the algorithm for restoring the invariant after an insertion is known as *bubbling up*. We begin by bubbling up the new element.

To bubble up an element, we compare it to its parent using the heap ordering predicate. If the invariant is broken—that is, if the element is less than its parent—then we swap the element with its parent, and continue checking upward at the element's new position, until the element rises high enough that either the invariant holds between the element and its parent, or the element has no parent.

Removal

The element to remove is always at the root of the tree (index 0 in the vector), but the binary tree completeness invariant requires that the heap shrink at the end (index `self.size - 1` in the vector). So to remove an element, first we move the last element to the root position, overwriting the old copy with `False` (which aids in debugging) and decrementing the size. Then we must restore the invariant, because the element that we moved to the root position might be in conflict with one or both of its children, if it has any. From the root of the tree, the element will only have to move down; hence the algorithm for restoring the invariant after a remove is known as *percolating down*. We begin percolating down at the root.

To percolate down an element, we need to find out whether it has a child smaller than itself; and if it has two children, we need to find the smaller of the two. This can be a bit tricky, since an element may have two children, a left child only, or no children. (Completeness forbids a right child without a left.) If the invariant is broken between the element and a child—that is, if the child is less than the element—then we swap the element with the smallest of its children, and continue checking downward at the elements new position, until the element sinks far enough that the invariant holds between the element and any children.

Heap sort

Given a vector and an ordering predicate, it is possible to permute the vector into heap order in $\mathcal{O}(n)$ time in bottom-up fashion, as follows. Iterate through the elements of the vector in reverse, and percolate down each element in turn. After that, it is possible to permute the heap-ordered vector into a sorted vector in $\mathcal{O}(n \log n)$ time, by repeated percolating down, as follows. Think of the vector as partitioned into an (initially empty) sorted section at one end, and the remaining heap-ordered section. Move an element from the heap-ordered section to the sorted section by simulating a removal from the heap-ordered section; this causes the "heap" to shrink and leaves the removed element in the correct position in the sorted section. Repeat until all elements are in the sorted section.

Thus, the ideas behind a binary heap can be used to sort a vector in-place, with no auxiliary storage, in $\mathcal{O}(n \log n)$ time. That said, you do not need to implement a fully in-place heap sort as described above for this course. Instead, the `heap_sort` function you implement will sort a vector of n elements using $O(n)$ auxiliary space in the form of a `BinHeap` that it creates and uses internally. The new heap will need to hold all the elements of the vector at once, so that determines its size. Iterate through the vector, inserting each element into the heap. Then iterate over the indices of the vector in order, overwriting each element in the vector with the next smallest element yielded by the heap.

Your task

In `binheap.rkt`, I've supplied a skeleton of a class `BinHeap` that implements the `PRIORITY_QUEUE` interface. The class already defines the three fields `data`, `size`, and `lt?` as described previously. I have also provided stubs for the constructor, the four required methods, and four suggested helper methods. After the class and some tests are stubs for three further suggested helper functions, and a stub for a `heap_sort` function that you must complete.

The four suggested helper methods and three suggested helper functions all have names beginning with underscore, which makes them private to the class and to the file, respectively. I've provided stubs for them in order to help you

factor your program effectively, but you need not use them, and can safely delete them if you prefer doing it another way.

The `BinHeap` class

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

Note the required asymptotic time complexities of the operations.

I have provided two small tests for the `BinHeap` class, but you will need many more.

The `heap_sort` function

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

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

Note the required asymptotic time complexity of the function.

I have provided three small tests 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 good way to exercise your heap implementation.

Deliverables

- The provided file `binheap.rkt`, containing:
  - the completed `BinHeap` class, with your definitions for the constructor and all four required methods;
  - the `heap_sort` function fully defined; and
  - sufficient tests to cover all cases.