HW3: Binary Heaps

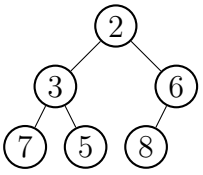
Due: Thursday, November 3, at 11:59 PM, on Canvas

You may work on your own or with one (1) partner.

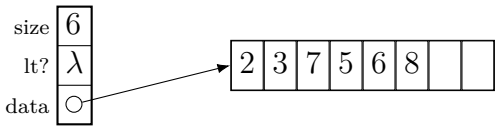
In this assignment, you will implement a fixed-size binary heap. The structure of the heap is already defined for you in `binheap.rkt`[1]. The heap is represented using a DSSL vector to contain the elements. Each heap also contains a comparison function for ordering the elements of the heap, so that your implementation can support heaps of integers, heaps of strings, heaps of whatits, heaps of sporkles, etc.

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, and the heap property means that each node's value is less than its children's values. Here is an example of a binary heap drawn as a tree:



In practice, binary trees are rarely represented as actual tree mades of links and nodes. Instead, the 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 struct of three fields. Here is the same heap as above in our actual heap representation:



The third 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

[1] `http://goo.gl/bt7imD`

of the vector; instead we store the actual number of elements in `size`, the first field.

The second field, `lt?`, stores the *heap ordering predicate*. This is necessary because your heap must work with any element type whatsoever, not just with numbers or strings. If elements were only numbers, you could compare them with `<`, but that won't work for strings; if elements were limited to strings, you could compare them with `string<?`, but that won't work for, say, bank account structs, which will require a different comparison function. So each `[Heap-of X]` has a field `lt?` containing the correct function for comparing its elements. Supposing that `h` is a `[Heap-of X]`, and `a` and `b` are two elements (*i.e.,* `X`s), we can compare them with `((heap-lt? h) a b)`.

Your task

In `binheap.rkt`, I've supplied a definition of a function `create` that returns a new, empty heap given a capacity and ordering function. Implementing the remaining operations is up to you:

```
insert!     : [Heap-of X] X -> Void    ; O(log n)
find-min    : [Heap-of X] -> X         ; O(1)
remove-min! : [Heap-of X] -> Void      ; O(log n)
```

For details, see the function headers provided in `binheap.rkt`, which include purpose statements as well. Each operation must have the worst-case shown above, where n is the number of elements in the heap.

In order to help you factor your program effectively, I've included at the bottom of `binheap.rkt` a list of helper functions with names, signatures (types), and purpose statements (brief functional descriptions). You are free to use as much or as little of my design as you like.

Extra credit

Make your heap expand as necessary to accomodate any number of assertions. To achieve this, instead of failing when the heap is full, `insert!` should allocate a new vector that doubles the capacity and copy the existing elements over from the old vector.

Deliverables

- The provided file `binheap.rkt`, containing:
  - the `insert!`, `find-min`, and `remove-min!` functions fully defined, and
  - sufficient tests to convince yourself your code's correctness.