# Extra Point Mini-Project: Heaps and Priority Queues in Haskell

Start Assignment

- Due Friday by 11:59pm
- Points 100
- Submitting a file upload
- File Types hs
- Available until Apr 19 at 11:59pm

In this project, you will implement a heap data structure, in particular **max-heaps,** which will be the basis for **priority queues**. In what follows, we will first review what these data structures are in general. Next, we will describe the project details.

**Max-Heaps:**

A max-heap is a complete binary tree that maintains the simple property that a node's key is greater than or equal to the node's children's keys. (Actually, a max-heap may be any tree, but is commonly a binary tree). Because x ≥ y and y ≥ z implies x ≥ z, the property results in a node's key being greater than or equal to all the node's descendants' keys. Therefore, a max-heap's root always has the maximum key in the entire tree.

An **insert** into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs. Inserts fill a level (left-to-right) before adding another level, so the tree's height is always the minimum possible. The upward movement of a node in a max-heap is called **percolating**.

A **remove** from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs. Because upon completion that node will occupy another node's location (which was swapped upwards), the tree height remains the minimum possible.

Heaps are typically stored using **lists**. Given a tree representation of a heap, the heap's list form is produced by traversing the tree's levels from left to right and top to bottom. The root node is always the entry at index 0 in the list, the root's left child is the entry at index 1, the root's right child is the entry at index 2, and so on. Because heaps are not implemented with node structures and parent/child pointers, traversing from a node to parent or child nodes requires referring to nodes by index. In general, for a parent node at index `i`, the children are at indexes `2*i+1` and `2*i+2`. Moreover, if a node is at index `i`, its parent is at index `(i-1)/2`.

**Percolate algorithm**: Following is the pseudocode in imperative style for the array-based percolate-up and percolate-down functions. The functions operate on an array that represents a max-heap and refer to nodes by array index.

```
MaxHeapPercolateUp(nodeIndex, heapArray) {
   while (nodeIndex > 0) {
      parentIndex = (nodeIndex - 1) / 2
      if (heapArray[nodeIndex] <= heapArray[parentIndex])
         return
      else {
         swap heapArray[nodeIndex] and heapArray[parentIndex]
         nodeIndex = parentIndex
      }
   }
}
```

```
MaxHeapPercolateDown(nodeIndex, heapArray, arraySize) {
   childIndex = 2 * nodeIndex + 1
   value = heapArray[nodeIndex]

   while (childIndex < arraySize) {
      // Find the max among the node and all the node's children
      maxValue = value
      maxIndex = -1
      for (i = 0; i < 2 && i + childIndex < arraySize; i++) {
         if (heapArray[i + childIndex] > maxValue) {
            maxValue = heapArray[i + childIndex]
            maxIndex = i + childIndex
         }
      }

      if (maxValue == value) {
         return
      }
      else {
         swap heapArray[nodeIndex] and heapArray[maxIndex]
         nodeIndex = maxIndex
         childIndex = 2 * nodeIndex + 1
      }
   }
}
```

**Heapsort** is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted list in reverse order. A list of unsorted values must first be converted into a heap. The **heapify** operation is used to turn a list into a heap. Since leaf nodes already satisfy the max heap property, heapifying to build a max-heap is achieved by percolating down on every non-leaf node in reverse order. The heapify operation starts on the internal node with the largest index and continues down to, and including, the root node at index 0. Given a binary tree with $N$ nodes, the largest internal node index is $N/2 - 1$.

Heapsort begins by heapifying the list into a max-heap and initializing an end index value to the size of the list minus 1. Heapsort repeatedly removes the maximum value, stores that value at the end index, and decrements the end index. The removal loop repeats until the end index is 0.

Here is the imperative style specification of Heapsort in pseudocode:

```
Heapsort(numbers, numbersSize) {
   // Heapify numbers array
   for (i = numbersSize / 2 - 1; i >= 0; i--) {
      MaxHeapPercolateDown(i, numbers, numbersSize)
   }

   for (i = numbersSize - 1; i > 0; i--) {
      Swap numbers[0] and numbers[i]
      MaxHeapPercolateDown(0, numbers, i)
   }
}
```

**Priority Queues:**

A **priority queue** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority. The priority queue **enqueue** operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority. The priority queue **dequeue** operation removes the item at the front of the queue, which has the highest priority. In addition to enqueue and dequeue, a priority queue usually supports **peeking**. A peek operation returns the highest priority item, without removing the item from the front of the queue.

A priority queue is commonly implemented using a heap. A heap will keep the highest priority item in the root node and allow access in `O(1)` time. Adding and removing items from the queue will operate in worst-case `O(log N)` time.

| Priority queue operation | Heap functionality used to implement operation | Worst-case runtime complexity |
|---|---|---|
| Enqueue | Insert | $O(logN)$ |
| Dequeue | Remove | $O(logN)$ |
| Peek | Return value in root node | $O(1)$ |

**Project Details:**

In what follows, the details of implementing heaps and priority queues are given.

1. Define `MaxHeap a` as a polymorphic type. You can define it as a type synonym to polymorphic lists.
2. Define function `percolateUp :: (Ord a) => Int -> MaxHeap a -> MaxHeap a` that receives a node index and a heap. It returns the heap resulted by percolating up the node in the

given index. This function is used in node insertion, given below. Make sure that the input index is a valid index.

3. Define function `percolateDown :: (Ord a) => Int -> MaxHeap a -> MaxHeap a` that receives a node index and a heap. It returns the heap resulted by percolating down the node in the given index. This function is used in node removal and heapify functions, given below. Make sure that the input index is a valid index.
4. Define function `insert :: (Ord a) => a -> MaxHeap a -> MaxHeap a` that receives a node and a heap. It returns the heap resulted by inserting the node in the heap. You need to use function `percolateUp` in the definition of this function.
5. Define function `remove :: (Ord a) => MaxHeap a -> MaxHeap a` that receives a heap. It removes the root and returns the resulting heap. This function is used in heap sort, given below. You need to use function `percolateDown` in the definition of this function.
6. Define function `heapify :: (Ord a) => [a] -> MaxHeap a` that receives a list and heapifies it. This function is used in heap sort. You need to use function `percolateDown` in the definition of this function (or whatever helper function that it relies on).
7. Define function `heapSort :: (Ord a) => [a] -> [a]` that receives a list and returns the sorted version by implementing heap sort algorithm. You need to use function `heapify` in the definition of this function.
8. Define `PQueue a` as a polymorphic type. You can define it as a type synonym to max-heaps.
9. Define function `enqueue :: (Ord a) => a -> PQueue a -> PQueue a` that receives an item and a priority queue. It inserts the item into the queue. Use max-heap insertion to define this function.
10. Define function `dequeue :: (Ord a) => PQueue a -> PQueue a` that receives a priority queue and removes the item with the highest priority in it. Use max-heap removal to define this function.
11. Define function `peek :: PQueue a -> Maybe a` that receives a priority queue and returns the item with the highest priority in it, if it exists. Otherwise, it returns `Nothing`.

**Dos and Don'ts:**

1. Submit your work as a single `.hs` file.
2. Follow the naming conventions suggested here for types and functions to help with grading.
3. Do not use any library associated with heaps and/or priority queues. The goal is to define these data structures from scratch, only relying on lists (and functions on lists).
4. Feel free to define arbitrary number of helper functions as part of defining the functions requested in the assignment. For example, you may realize that `heapify` function needs a helper function that recurses on the number of items being percolated down.
5. Annotate the types. This would help you make sure you are defining functions according to the spec. Moreover, if you don't annotate the type, type inference algorithm may come up with a type that is not what you needed.

6. Go step by step in defining your functions. After each definition unit test your functions with different input.
7. Note that this is an individual assignment (not group-based).

**Heaps and Priority Queues in Haskell**

| Criteria | Ratings | | Pts |
|---|---|---|---|
| MaxHeap type definition | **5 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 5 pts |
| percolateUp :: (Ord a) => Int -> MaxHeap a -> MaxHeap a | **15 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 15 pts |
| percolateDown :: (Ord a) => Int -> MaxHeap a -> MaxHeap a | **15 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 15 pts |
| insert :: (Ord a) => a -> MaxHeap a -> MaxHeap a | **10 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 10 pts |
| remove :: (Ord a) => MaxHeap a -> MaxHeap a | **10 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 10 pts |
| heapify :: (Ord a) => [a] -> MaxHeap a | **15 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 15 pts |
| heapSort :: (Ord a) => [a] -> [a] | **10 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 10 pts |
| PQueue type definition | **5 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 5 pts |
| enqueue :: (Ord a) => a -> PQueue a -> PQueue a | **5 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 5 pts |
| dequeue :: (Ord a) => PQueue a -> PQueue a | **5 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 5 pts |

| Criteria | Ratings | | Pts |
|---|---|---|---|
| | | | |
| peek :: PQueue a -> Maybe a | **5 to >0.0 pts**<br>**Full Marks** | **0 pts**<br>**No Marks** | 5 pts |
| | | | Total Points: 100 |