Exercise 1

1. Representation Invariant & Abstraction Function

Representation： Heap = ⟨ A: array, size: int ⟩

Representation Invariant :

`RI` = 0 $\leq$ size $\leq$ A.length **&&** $\forall$ 1< i $\leq$ A.length, A[`⌊i/2⌋`] $\leq$ A[i]

Abstraction Function :

$Abs(R) = Abs(R, 1)$, where

$$Abs(R, i) = \begin{cases} \langle R.A[i], Abs(R, 2*i), Abs(R, 2*i+1)\rangle, \text{if } i \leq R.size \\ Nil \text{ , if } i > R.size \end{cases}$$

2. Check the Representation Invariant

[code in the wet part : `MinHeapRep`]

3. MIN-HEAPIFY

- ```
  MIN-HEAPIFY(A:T[], size:int, i:int):
      curr = i
      smallest = curr
      do :
          curr = smallest
          leftChild = 2 * curr + 1
          rightChild = 2 * curr + 2

          if leftChild < size and A[leftChild] < A[smallest]:
              smallest = leftChild
          if rightChild < size and A[rightChild] < A[smallest]:
              smallest = rightChild

          if smallest != curr:
              swap(A[curr], A[smallest])
      while(smallest != curr)
  ```

4. MIN-HEAPIFY Complexity Analysis

Our algorithm compares A[i] with its two children, swap if they don't satisfy the Min-Heap RI and then continue the loop with the child index. So the worst case will be: The MIN-HEAPIFY starts with the root, ends at one leaf, visiting all **height =** $\lfloor \log_2 n \rfloor$ levels. In each loop, the three conditional check and one possible swapping all run in O(1). So the worst-case time complexity of MIN-HEAPIFY is $\log_2 n \cdot O(1) = O(\log_2 n)$.

5. Correctness Proof

- Contract of MIN-HEAPIFY :
  - `Pc` : |A| = size **&&** 0$\leq$ i $<$size

    && the subtree rooted at left child of A[i] is a valid Min-Heap

    && the subtree rooted at right child of A[i] is a valid Min-Heap
  ```

- `Pc'` : |A| = size **&&** 0$\leq$ i $<$size

  && the subtree rooted at left child of A[i] is a valid Min-Heap

  && the subtree rooted at right child of A[i] is a valid Min-Heap

  && curr = i

- `Inv` : i $\leq$ curr $<$ size

  && the subtree rooted at left child of A[i] is a valid Min-Heap

  && the subtree rooted at right child of A[i] is a valid Min-Heap

- `Qc'` : A[curr] < A[leftChild], if it exists && A[curr] < A[rightChild], if exist

- `QC` : the tree rooted at index i is a valid Min-Heap

* Using the Loop Invariant Theorem :
  - **Initialization**: Assume `Pc` holds, we need to prove `Inv` hold:

    Before the loop starts, from `Pc'`, we know that curr = i,

    then i $\leq$ curr $<$ size, the loop invariant holds at the start.

  - **Maintenance**: Assume `Inv&&B` holds, we need to prove the loop make `Inv` true again :

    `Inv&&B` : i $\leq$ curr $<$ size **&&** smallest != curr

    && the subtree rooted at left child of A[i] is a valid Min-Heap

    && the subtree rooted at right child of A[i] is a valid Min-Heap

    Inside each loop, we compare A[curr] to its two children. If they satisfy the Min-Heap property, then break the loop and `smallest == curr`. Otherwise, swap A[curr] with the smaller child, and continue the loop at `curr = smallest`. So, in any case, whether the value of `curr` be updated or not, `curr` is always smaller than `size`(Because if 2 * curr +1 or 2 * curr +2 exceed `size`, then they have no chance to be `smallest`). $\Rightarrow$ After the loop, i $\leq$ curr $<$ size. And the swap on the level of A[curr] does not influence the subtrees, which are already valid Min-Heaps. Thus, `Inv` still holds after each loop.

  - **Termination**: Once the loop ends, `Inv&&(not)B` holds, we need to prove `Qc` holds :

    `Inv&&(not)B` : i $\leq$ curr $<$ size **&&** smallest == curr

    && the subtree rooted at left child of A[i] is a valid Min-Heap

    && the subtree rooted at right child of A[i] is a valid Min-Heap

    That means no swapping happens in the last loop, the Min-Heap property is eventually satisfied at the last `curr` position. `Inv&&(not)B` tells the two subtrees are initially valid Min-Heaps, then after we fix the possible violation, the tree rooted at index i is a valid Min-Heap now. Therefore, `Qc` holds.

  Thus, by the **Loop Invariant Theorem**, we've shown our algorithm `MIN-HEAPIFY` is correct.

Exercise 2

1. Representation invariant

Representation : Heap = ⟨ root: Node ⟩

$\leftarrow$ Node = ⟨ value: T, parent: Node, left: Node, right: Node ⟩

Representation invariant :

`RI` = root is a binary tree **&&** for $\forall$ node $\in$ Heap,

if node.left exists, then node.value.compareTo(node.left.value) ≤ 0

if node.right exists, then node.value.compareTo(node.right.value) ≤ 0

[code in the wet part : `MinHeapTreeRep`]

2. Insertion in Min-Heap

- the pseudo-code definition of `insertion` :

```
Node<T> insertion(Node<T> node, T value) :
    Node<T> newNode = new Node<>(value)
    if (node == null) :
        return newNode
    Node<T> parent = parentOfNew(node)
    if (parent.left == null) :
        parent.left = newNode
    else :
        parent.right = newNode
    newNode.parent = parent
    MIN-HEAPIFY(newNode)
    return node
```

- the pseudo-code definition of auxiliary operations :

```
Node<T> parentOfNew(Node<T> node):
    if (node == null) :
        return null
    if (node.left == null || node.right == null) :
        return node
    if (parentOfNew(node.left) != null) :
        return parentOfNew(node.left)
    return parentOfNew(node.right)
```

```
void MIN-HEAPIFY(Node<T> node) :
    while(node.parent != null) :
        if (node.value.compareTo(node.parent.value) >= 0) :
            break
        else :
            swap(node.value, node.parent.value)
            node = node.parent
```

4. Complexity of Insertion

- Inside the `insertion` algorithm, we :
  - Set up a Node $\Rightarrow O(1)$
  - Check the condition node == null, and possible return $\Rightarrow O(1)$
  - Set up a Node, given by `parentOfNew`, which runs recursively till one leaf is reached. In the worst case, it visits all the nodes in the tree $\Rightarrow O(n)$
  - Figure out which child of parent is null, and link the new node we create into the right place $\Rightarrow O(1)$
  - Heapify the new tree, where new node is taken as a leaf. In the worst case, the new node compares and swaps from bottle to the top, visiting all levels $\Rightarrow O(\log_2 n)$

  The time complexity of `insertion` is $O(1) + O(n) + O(\log_2 n) = O(n)$

5. Implementation for Insertion

[code in the wet part : `MinHeapInsert`]