# 一、 Heapify 求解

## 1.1 问题需求

一棵 minheap 树定义为：

1). t is Empty;

2). t is a Node(L, x, R), where R, L are minheaps and values(L), value(R) >= x (value(T)函数用于获取树 T 的根节点的值）

编写函数 treecompare, SwapDown 和 heapify：

treecompare: tree * tree -> order

(* when given two trees, returns a value of type order, based on which tree has a larger value at the root node *)

SwapDown: tree -> tree

(* REQUIRES the subtrees of t are both minheaps

* ENSURES swapDown(t) = if t is Empty or all of t’s immediate children are empty then just return t, otherwise returns a minheap which contains exactly the elements in t. *)

heapify : tree -> tree

(* given an arbitrary tree t, evaluates to a minheap with exactly the elements of t.*)

分析 SwapDown 和 heapify 两个函数的 work 和 span。

## 1.2 解题思路与代码

### treecompare 设计思路：

treecompare 函数用于比较两棵树的根节点值，从而比较两棵树的大小。具体逻辑如下：

• 处理空树的情况：

当两棵树均为空时，认为两棵空树相等，返回 EQUAL。

当第一棵树为空、第二棵树非空，返回 LESS，表示第二棵树的根节点更大。

当第二棵树为空、第一棵树非空，返回 GREATER，表示第一棵树的根节点更大。

• 比较非空树的根节点：

如果两棵树的根节点的值 x1 和 x2 进行比较，若 x1 < x2 返回 LESS；若 x1 > x2 返回 GREATER；若相等则返回 EQUAL。

具体代码如下：

```
fun treecompare (Empty, Empty) = EQUAL
| treecompare (Empty, Br (_, _, _)) = LESS (* 空树小于任意非空树 *)
| treecompare (Br (_, _, _), Empty) = GREATER(* 非空树大于空树 *)
| treecompare (Br (_, x1, _), Br (_, x2, _)) =Int.compare(x1, x2); (*比较根节点*)
```

## SwapDown 设计思路：

SwapDown 函数的要求是让树 t 的子树必须都是最小堆。如果 t 是空树或者 t 的所有直接子树都为空则返回 t，否则返回一个最小堆，这个最小堆要包含 t 中的所有元素。其具体的解题思路如下：

• 1、空树情况处理：

当树是空树的时，这个时候这棵树已经满足了最小堆的要求，故直接返回空树；当树只有一个节点，没有子树的时候，也满足最小堆的需求，直接返回这个节点。

• 2、处理只有一个子树的情况：

如果当前节点只有左子树或者右子树，则比较当前节点和子树的根节点的值，若当前节点的值大于这一个子树根节点的值，则进行交换。

• 3、处理同时存在左右子树的情况：

比较左右子树的根节点，决定将哪个节点和当前的根节点交换位置：

若左子树的根节点较小，则将当前节点与左子树的根节点交换，并递归调整（保持最小堆性质）；若右子树的根节点较小，将当前节点与右子树的根节点交换，并递归调整；若当前节点的值不大于左右子树根节点的值，则不需要交换，返回当前的树。

代码：

```
fun SwapDown(Empty) = Empty
| SwapDown (Br(Empty,x,Empty)) = Br(Empty,x,Empty)
(* 只有左子树，检查根节点和左子树的根值 *)
  | SwapDown (Br(Br(leftLeft, leftValue, leftRight), root, Empty)) =
      if leftValue < root then
          Br(SwapDown(Br(leftLeft, root, leftRight)), leftValue, Empty)
      else
          Br(Br(leftLeft, leftValue, leftRight), root, Empty)
(* 只有右子树，检查根节点和右子树的根值 *)
  | SwapDown (Br(Empty, root, Br(rightLeft, rightValue, rightRight))) =
      if root > rightValue then
          Br(Empty, rightValue, SwapDown(Br(Empty, root, Br(Empty, rightValue, rightRight))))
      else
          Br(Empty, root, Br(Empty, rightValue, rightRight))
(* 左右子树都存在时，进行比较并交换 *)
  | SwapDown (Br(Br(leftLeft, leftValue, leftRight), root, Br(rightLeft, rightValue, rightRight))) =
    if root > rightValue then
        Br(Br(leftLeft, leftValue, leftRight), rightValue, SwapDown (Br(rightLeft, root, rightRight)))
    else if root > leftValue then
        Br(SwapDown (Br(leftLeft, root, leftRight)), leftValue, Br(rightLeft, rightValue, rightRight))
    else
        Br(Br(leftLeft, leftValue, leftRight), root, Br(rightLeft, rightValue, rightRight));
```

## Heapify 设计思路：

Heapify 的作用是将任意二叉树转换为最小堆，堆中的元素与原始树中的元素完全相同。其具体思路如下：

• 1、若树 t 是空的则直接返回空树

• 2、若树 t 非空，递归地对 t 的左子树和右子树调用 heapify，使它们分别成为最小堆。

• 3、在左右子树都成为最小堆之后，构造新的节点，以当前节点值为根，heapify 后的左右子树为子节点。

• 4、对新的树调用 SwapDown 以确保整个树满足最小堆性质。

**具体代码如下：**

```
fun heapify(Empty) = Empty
  | heapify (Br(left, root, right)) =
      let
        val t1 = heapify (left)
        val t2 = heapify (right)
        val t3 = Br(t1, root, t2)
      in
        SwapDown (t3)
      end;
```

## 1.3 性能分析（请用树的深度进行分析）

前提假设：**假设树的深度记为$d$，树的节点数记为$n$。**

### SwapDown 的性能分析：

***Work：***

在每次调用 SwapDown 时，首先需要比较根节点和两个子节点的值，这一比较操作是常数的时间复杂度$O(1)$。在最坏情况下，SwapDown 可能得从根节点一直向下交换到叶子节点，这个过程与树的深度$d$成正比。所以对于一棵深度为$d$的树，$W_{SwapDown}=O(d)$。

***Span：***

由于 SwapDown 中的操作具有顺序依赖性，即必须在完成当前交换后才能进行下一步的比较和交换，因此$S_{SwapDown}=O(d)$。

经过上述的分析，对于平衡树，其深度$d$为$\log n$，对于最坏情况，其深度$d$为$n$。因此，在**平衡树的情况**下，SwapDown 的 work 和 span 均为$O(logn)$。在**最坏情况**下，SwapDown 的 work 和 span 为$O(n)$。

### Heapify 的性能分析：

heapify 的作用是将任意二叉树转换为包含相同元素的最小堆。其执行步骤如下：

- 1、处理空树：当输入的树 t 是空树时，heapify 函数直接返回空树。
- 2、处理非空树：对于一个非空的树 t，函数递归调用 heapify 处理 t 的左子树

和右子树，递归地将它们转化为最小堆，并且存储起来。

• 3、构建新节点：在将左右子树都转化成为最小堆后，构造一个新的节点，以当前节点值为根节点，heapify 函数处理后的左右子树为子节点。

• 4、调整堆结构：调用 SwapDown 函数对新构建的树进行调整，确保它符合最小堆的性质。

***Work***

**在 heapify 函数中，工作量主要来自以下两个部分：**

• 对于深度为$d$的树，对左、右子树的递归 heapify 调用各需要$W(d-1)$的工作量。

• 除了递归调用，每个节点还要执行一次外 SwapDown 操作，其工作量为$O(d)$。

因此，heapify 的工作量满足以下递归关系：$W(d)=2W(d-1)+O(d)$

求解这个递归关系得到 heapify 的总工作量为$W(d)=O(d\cdot 2^d)$

因此，**平衡树情况**下，Heapify 的 Work 为$O(n\log n)$，**最坏情况**下 heapify 的 Work 为$O(n\cdot 2^n)$。

***Span***

• 在 heapify 函数中，对左子树和右子树的 heapify 调用可以同时进行，这两个操作的时间可以并行计算，其 span 为$S_{heapify}(d-1)$。

• 一旦左右子树的 heapify 完成，接下来将进行 SwapDown 操作。这个操作的 span 为$O(d)$。

因此 heapify 的 span 满足以下递归关系：

$$S(d)=S(d-1)+O(d)$$

求解这个递归关系式得到 heapify 的 span 为$O(d^2)$。

因此，**平衡树情况**下，Heapify 的 Span 为$O(\log^2 n)$，**最坏情况**下 heapify 的 Span 为$O(n^2)$。