函数式编程原理课程报告

姓　　名：　　刘文博

班　　级：　　计算机本硕博 2101 班

学　　号：　　U202115666

指导教师：　　郑然

分数	
教师签名	

2023 年 10 月 20 日

目　录

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）

(1) 编写函数 treecompare, SwapDown 和 heapify：

treecompare 函数：

- 描述：给定两棵树，该函数返回一个类型为 **order** 的值，该值基于哪棵树具有较大值的根节点。
- 参数：两个树作为输入。
- 返回值：类型为 **order** 的值，可以表示哪棵树的根节点值较大。
- 这个函数的目的是比较两个树的根节点的值，根据它们的大小关系返回一个 **order** 值，可能是 **Greater**、**Less** 或 **Equal**，表示第一个树的根节点值大、小或等于第二个树的根节点值。

SwapDown 函数：

- 描述：给定一个树，该函数确保返回一个 minheap，其中包含与输入树中相同的元素，同时保持 minheap 属性。
- 参数：一个树作为输入，要求输入树的子树都是 minheaps。
- 返回值：一个 minheap，其中包含与输入树相同的元素。
- 这个函数的目的是确保给定的树 **t** 满足 minheap 的性质。它要检查树的根节点和其子节点之间的大小关系，并进行必要的交换，以确保树仍然是一个 minheap。

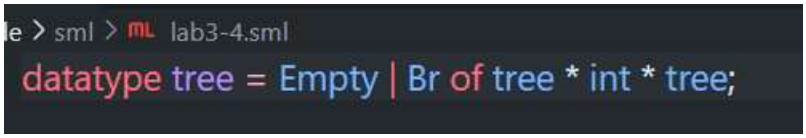
heapify 函数:

- 描述：给定任意形状和结构的树 **t**，该函数将这个树转换成一个 minheap，这个 minheap 包含了与输入树 **t** 中相同的元素
- 参数：一个任意的树 **t**。
- 返回值：返回一个 minheap 树，包含与输入树相同的元素。
- 这个函数的目的是将任意给定的树转换为一个 minheap 树，保持树的结构不变，只是根据 minheap 性质对节点进行必要的交换。

(2) 在作业中分析 SwapDown 和 heapify 两个函数的 work 和 span。

1.2 解题思路与代码

- 首先定义树类型，这里的 **Br** 是一个构造函数，它用于创建包含一个值的树节点，这个节点由左子树、整数值和右子树组成。**Empty** 代表一个空树，也就是没有节点的树。



- **treecompare** 函数：

函数接受两个参数，分别是两棵树，即 **(t1, t2)**，该函数思路较为简单，我们默认**非空树>空树**，将所有可能的结果列举出来，并分情况给出其相应的 order 即可。

```
(*begin*)
fun treecompare (Empty, Empty) = EQUAL
  | treecompare (Empty, _) = LESS
  | treecompare (_, Empty) = GREATER
  | treecompare (Br(L1, x1, R1), Br(L2, x2, R2)) =Int.compare(x1, x2);
```

- **SwapDown** 函数：

根据函数定义的要求，可知输入树的子树为 minheap 树，最后输出的树仍是一个 minheap 树。

我们仍分情况讨论：

1. 如果输入的树是空树（**Empty**），则直接返回空树。这是 **SwapDown** 函数的 **base case** 之一，表示没有子树可以继续进行交换操作。

```
fun swapDown (Empty) = Empty
```

2. 如果输入的树是一个单节点的树（**Br(Empty, x, Empty)**），那么这个树已经是一个最小堆，因为它只有一个节点，直接返回。同样是 **base case**。

```
  | swapDown (Br(Empty, x, Empty)) = Br(Empty, x, Empty)
```

3. 如果输入的树是一个根节点和非空子树（minheap 树）组成的树，那么我们比较根节点 **y** 和两个子树的根节点 **x**、**z**，根据 **x**、**y**、**z** 大小关系进行交换，使其满足根节点<左子树的根节点且根节点<右子树的根节点。分别对左右子树进行 SwapDown 操作。以 SwapDown 左子树T_l为例，对于左子树T_l来说,虽然其根节点经过交换，但T_l的左右子树仍满足 minheap 性质，满足 SwapDown 函数的输入要求，根据定义 SwapDown T_l的输出是一个 minheap，且其根节点的值小于等于 SwapDown 之前的根节点的值，右子树同理。又由于经过了交换，左子树的根节点和右子树的根节点均大于根节点，故整棵树是一个 minheap 树，至此通过递归调用 Swap Down 函数实现了 SwapDown 操作。

```
57    | swapDown (Br(Empty, x, Br(L, y, R))) =
58        if x > y then Br(Empty, y, swapDown (Br(L, x, R)))
59        else Br(Empty, x, Br(L, y, R))
60    | swapDown (Br(Br(L, y, R), x, Empty)) =
61        if x > y then Br(swapDown (Br(L, x, R)), y, Empty)
62        else Br(Br(L, y, R), x, Empty)
63    | swapDown (Br(Br(L1, x, R1), y, Br(L2, z, R2))) =
64        if (x <= y) andalso (y <= z) then Br(swapDown (Br(L1, y, R1)), x, Br(L2, z, R2)) (*x<=y<=z*)
65        else if (y <= x) andalso (x <= z) then Br((Br(L1, x, R1)), y, Br(L2, z, R2)) (*y<=x<=z*)
66        else if (y <= z) andalso (z <= x) then Br((Br(L1, x, R1)), y, Br(L2, z, R2)) (*y<=z<=x*)
67        else if (x <= z) andalso (z <= y) then Br(swapDown (Br(L1, y, R1)), x, (Br(L2, z, R2))) (*x<=z<=y*)
68        else if (x <= y) andalso (z <= x)then Br(Br(L1, x, R1), z, swapDown (Br(L2, y, R2))) (*z<=x<=y*)
69        else Br(Br(L1, x, R1), z, swapDown (Br(L2, y, R2))); (*z<=y<=x*)
```

注意尽量避免不必要的 SwapDown，只有当子树的根节点和原本的根节点进行交换时，才对子树进行 SwapDown 操作。

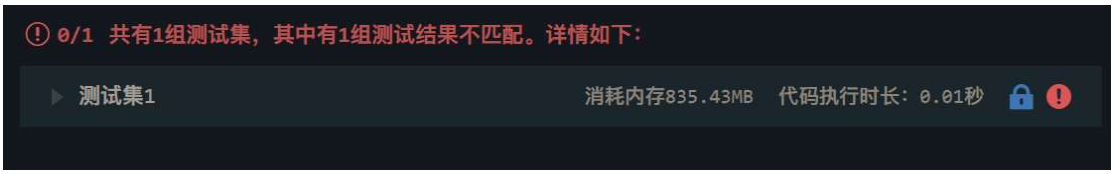
- **heapify** 函数：

根据 heapify 函数的定义，它将一棵树变为 minheap 树。可以将原问题分解为两个较小的子问题进行求解，再将两部分解进行组合得到最终解。即对左右子树进行 heapify 操作得到两个 minheap 树，此时满足 swapDown 函数的定义，调用 swapDown 函数组合两子问题解得到最终的 minheap 树。

```
fun heapify (Empty) = Empty
  | heapify (Br(left, x, right)) =
      let
        val leftHeap = heapify left
        val rightHeap = heapify right
      in
        swapDown (Br(leftHeap, x, rightHeap))
      end;
(*end*)
```

1.3 遇到的问题及运行结果

- 问题：未能通过头歌测试



- 运行结果：尽管未能通过头歌测试，为了检验代码的正确性，我通过分析 heapify 之

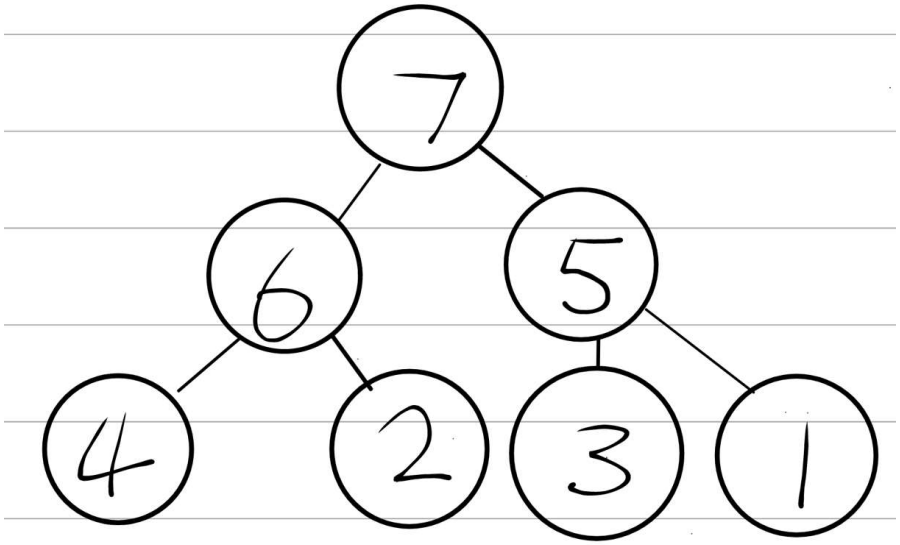
后树的结构来判断其是否为 minheap 树，得到运行结果如下。

```
val L = [7,6,5,4,3,2,1];
val h = 2;
printIntList (trav(listToTree L));
printIntList (trav1(listToTree L));
(*printIntList (trav(revT(listToTree L)));
printBool(binarySearch((listToTree L), h));*)
printIntList (trav(heapify(listToTree L)));
printIntList (trav1(heapify(listToTree L)));
```

```
val L = [7,6,5,4,3,2,1] : int list

val h = 2 : int

4 6 2 7 3 5 1 val it = () : unit

7 6 4 2 5 3 1 val it = () : unit

4 2 6 1 7 3 5 val it = () : unit

1 2 4 6 3 7 5 val it = () : unit
```

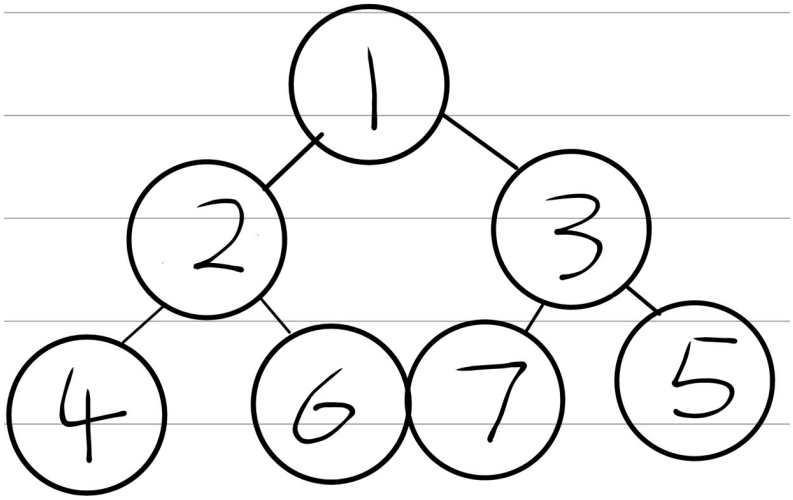
其中 listTOTree 将 int list 类型的数据转化为 tree 类型，trav 函数中序遍历一个 tree，trav1 先序遍历一个 tree，将 tree 转化为 int list。printIntList 将 int list 打印出来。

经过我的分析，listToTree 得到的树的结构为



可以看出它并不是一个 minheap 树

Heapify 之后树的结构为



此时是一个 minheap 树。

同时我通过自测运行，打印出了头歌平台上 int List L 的输入，虽然结果不一样，但经过类似分析，heapify 之后得到的同样是一个 minheap 树。

1.4 性能分析

- **SwapDown** 函数：

Work： 在最坏情况下，SwapDown 会递归调用到叶子节点。因为 SwapDown 函数在树的每层的 Work 为常数级，之后递归调用对子树的 SwapDown。故 SwapDown 的工作量与树的高度成正比。

因为 SwapDown 函数只进行节点的值的交换，不会改变树的结构，故树高由 ListToTree 函数决定。

```
fun split [ ]  = ([ ], [ ])
  | split [x] = ([ ], [x])
  | split (x::y::L) =
let val (A, B) =split L
in (x::A, y::B)
end;
```

又因为 split 函数将 int list 尽可能地进行均分，故左右子树的大小（所含节点的数目）大体相近，故树高的期望值为 O(log n),故 SwapDown 的 Work 为 O(log n)数量级。当然，若假设原本的树高为 h，则 Span 为 O(h)数量级。

Span： SwapDown 是一个递归函数，每次递归调用都必须等待其子调用完成。在最坏情况下，SwapDown 的并行性与树的高度成正比，为 O(log n)数量级。当然，若假设原本的树高为 h，则 Span 为 O(h)数量级。

- **heapify** 函数：

Work：SwapDown 阶段 work 是 O(log n)，其中 n 是树中的节点数。由于 heapify 递归调用 heapify left ，heapify right，设 heapify 的 Work 为 W(n),则平均状况下有递推表达式 W(n)=O(log n)+2W(n/2),故 W(n)=O(nlog n) 。当然，若假设原本的树高为 h，则 Work 的分析类似，Work 为 O($2^h * h$)数量级。

Span：SwapDown 阶段的 span 与树的高度成正比，其期望值为 O(log n)。 heapify 递归

调用 heapify left ， heapify right，两部分计算没有依赖关系，但 SwapDown 操作依赖递归调用的结果。故设 heapify 的 Span 为 S(n),则平均状况下有递推表达式 S(n)=O(log n)+S(n/2),故 S(n)=O($log^2 n$)。

若假设树高为 h,则 Span 的分析类似，为 O(h^2)数量级。

二、课程总结和建议

总结：这门课让我了解了函数式编程语言 SML，增强了我的并行思维和函数式编程的思想。

建议：希望这门课开设在并行数据结构与算法（CMU-15210）课程之前，因为该课程的实验需要自学 SML 语言，且实验难度较大。个人认为函数式编程原理是并行数据结构与算法的先导课程，但教务的排课顺序不是很合理。