函数式编程原理课程报告

姓　　名：　　王文涛

班　　级：　　CS2201

学　　号：　　U202215357

指导教师：　　郑然

分数	
教师签名	

年　月　日

目　录

一、 Heapify 求解

1.1 问题需求

正文统一采用小四号宋体/Times New Roman 和 1.25 倍行距。

一棵 minheap 树定义为：

t is Empty;

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: 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. *)

1.2 解题思路与代码

treecompare 函数比较两个树的根节点值，返回更大值的树。

首先进行特判，如果两个树均为空，则相等。

如果有一个树为空，则不为空的树更大。

如果两个树均不为空，则比较根节点的值。

代码如下：

```
fun treecompare (Empty, Empty) = EQUAL
| treecompare (Empty, _) = LESS
| treecompare (_, Empty) = GREATER
```

```
| treecompare (Br(_, x1, _), Br(_, x2, _)) = Int.compare(x1,x2);
```

SwapDown 函数交换当前节点与其最小子节点，直到树满足最小堆的性质。

如果当前节点为空，则直接返回空。

对于有左右子树的情况，继续分类讨论。使用模式匹配来处理左子树和右子树的不同情况。

如果两个子树都是空的，则直接返回当前节点。

如果只有一个子树，这个子树的值小于节点的值时，交换当前节点和子节点的值，并对交换后的子节点递归调用 SwapDown。否则不做改变。

如果左右子树都有，节点的值小于或等于其两个子节点的值时，无需进行交换。否则，交换当前节点和较小子节点的值，并对交换后的子节点递归调用 SwapDown。

代码如下：

```
fun SwapDown (Empty : tree) : tree = Empty
  | SwapDown (Br(l, x, r)) =
    case (l, r) of
      (Empty, Empty) =>
        Br(Empty, x, Empty)
      | (Empty, Br(rl, rx, rr)) =>
          if rx < x then Br(Empty, rx, SwapDown(Br(rl, x, rr)))
          else Br(l, x, r)
      | (Br(ll, lx, lr), Empty) =>
          if lx < x then Br(SwapDown(Br(ll, x, lr)), lx, Empty)
          else Br(l, x, r)
      | (Br(ll, lx, lr), Br(rl, rx, rr)) =>
            if x <= lx andalso x <= rx then
                Br(l, x, r)
            else
                if lx < rx
                  then Br(SwapDown(Br(ll, x, lr)), lx, r)
                else Br(l, rx, SwapDown(Br(rl, x, rr)))
```

heapify 函数从下到上对所有的的节点调用 SwapDown 函数，以达到构造最小堆的目的。

代码如下：

```
fun heapify Empty = Empty
| heapify (Br(l, x, r)) = SwapDown(Br(heapify(l), x, heapify(r)))
```

1.3 遇到的问题及运行结果

遇到什么问题，如何解决；运行结果及测试截图。

遇到的问题：在树结构比较复杂的情况下，SML/NJ 在输出类型时会输出#代替较低的节点。

解决方法：开始时自己手动将树画出来，之后编写函数，模仿解释器的格式输出树的结构。

测试截图如下：

```
val L = [6,5,4,3,2,1] : int list
val tree = Br (Br (Empty,5,Br #),6,Br (Br #,4,Br #)) : tree
变化前的树：
val it = () : unit
Br (Br (Empty, 5, Br (Empty, 3, Empty)), 6, Br (Br (Empty, 2, Empty), 4, Br (Empty, 1, Empty)))
val it = () : unit
val t1 = Br (Br (Empty,3,Br #),1,Br (Br #,2,Br #)) : tree
变化后的树：
val it = () : unit
Br (Br (Empty, 3, Br (Empty, 5, Empty)), 1, Br (Br (Empty, 6, Empty), 2, Br (Empty, 4, Empty)))
val it = () : unit
```

```
val L = [8,3,2,1,6,6] : int list
val tree = Br (Br (Empty,3,Br #),8,Br (Br #,2,Br #)) : tree
变化前的树：
val it = () : unit
Br (Br (Empty, 3, Br (Empty, 1, Empty)), 8, Br (Br (Empty, 6, Empty), 2, Br (Empty, 6, Empty)))
val it = () : unit
val t1 = Br (Br (Empty,3,Br #),1,Br (Br #,2,Br #)) : tree
变化后的树：
val it = () : unit
Br (Br (Empty, 3, Br (Empty, 8, Empty)), 1, Br (Br (Empty, 6, Empty), 2, Br (Empty, 6, Empty)))
val it = () : unit
```

在两次测试中，树均被转换成了最小堆结构。

1.4 性能分析（请用树的深度进行分析）

Treecompare 函数只进行常数时间的比较， span 和 work 均为$O(1)$。

SwapDown 函数：

SwapDown 在每一层可能会对左右子树进行一次比较，并递归到较小的子树上。因此，SwapDown 的工作量与树的高度成正比。对于深度为n的树，SwapDown 的 work 应为$O(\mathrm{h})$。同时 SwapDown 并不存在并行的情况，span 也为$O(\mathrm{h})$。

heapify 函数：

heapify 对树的每个节点都会进行一次递归操作，并在每个节点调用

SwapDown。因此，heapify 的总工作量是每个节点的 SwapDown 工作量的累积。对于深度为n的树，work 是 $O(\mathrm{h}\cdot 2^{\mathrm{h}})$。同时 heapify 的递归会同时对左右子树调用，可以并行进行。对于高 h，节点数为 n 的节点可以列出递推式

$$f(\mathrm{n}) = \mathrm{f}(\frac{\mathrm{n}}{2}) + \mathrm{O}(\log \mathrm{n})$$

求解得到$f(n) \in O(log^2 n)$即 span 为$O(h^2)$。

二、 数制转换（选做）

2.1 解题思路和代码

toInt 函数将一个用列表表示的 b 进制的数转换成十进制。首先构造一个计算幂的辅助函数，根据题目提供的公式递归计算$\sum_{i=1}^{n} b^{i-1}\, d_i$。

代码如下：

```
fun toInt b [] = 0
  | toInt b (x::xs) =
  let
    fun power(_, 0) = 1
    | power(b, n) = b * power(b, n - 1);
  in x * power(b, length xs) + toInt b xs
  end;
```

toBase 函数将整数转换为对应基数的数字列表，是 toInt 函数的逆操作。

代码如下：

```
fun toBase b 0 = []
  | toBase b n = toBase b (n div b) @ [n mod b];
```

convert 函数先将列表从基数 b1 转换为十进制，再将结果转换为基数 b2 的列表，依次调用前两个函数即可。

代码如下：

```
fun convert (b1, b2) L = toBase b2 (toInt b1 L);
```

2.2 高阶函数和多态类型

高阶函数：

高阶函数是指能够接收其他函数作为参数或返回一个函数的函数。是函数式编程的核心特性之一。

高阶函数使得函数的使用更加灵活和强大，可以接受或返回其他函数，支持复杂的函数组合。

多态类型：

多态类型允许函数和数据结构在不同类型上工作。SML 支持的多态类型通常通过类型变量实现。

多态类型提供了类型通用性，允许编写适用于多种类型的函数，增强了代码的重用性和类型安全。

三、 函数式拓展学习调研

3.1 函数式程序应用场景

（1）并行和分布式计算

函数式编程语言具有良好的并行和并发处理能力，能有效地组织和调度大量任务在多核处理器或者分布式环境中运行。例如，Akka 框架基于 Scala 提供了一种强大的并行 Actor 模型，使得函数式编程应用于实时消息传递、微服务架构以及大规模数据处理成为可能。

（2）数据处理和分析

在大数据和数据科学领域，数据处理任务常常涉及对大量数据进行清洗、转换和分析。函数式编程的高阶函数和组合性使得这些操作更为简洁和易于理解。例如使用 Scala 和 Apache Spark 进行大规模数据处理，利用 map、filter 和 reduce 等高阶函数进行数据转换和汇总。

（3）编译器和解释器开发

函数式语言提供了强大的抽象能力，特别适合处理语言的抽象语法树（AST）和各种变换。它们的惰性求值和高阶函数能够简化许多编译过程中的操作。例如使用 OCaml 开发编译器，利用模式匹配和递归来处理复杂的语法结构和类型检查。

3.2 函数式特征延伸

（1）Lambda 表达式

在 C++11 中引入的 Lambda 表达式是一种简洁的表示匿名函数的方式。

代码如下：

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
auto evens = std::count_if(numbers.begin(), numbers.end(), [](int n) { return n % 2 == 0; });
```

目的：

简化函数定义，不再需要定义额外的类或函数来实现简单的逻辑。

提高可读性，通过使用 Lambda 表达式，代码变得更简洁且直观。

（2）高阶函数

高阶函数是指可以接受其他函数作为参数或返回函数的函数。在 Java 8 中引入的 Stream API，允许使用高阶函数进行集合操作。

代码如下：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evens = numbers.stream()
                                .filter(n -> n % 2 == 0)
                                .map(n -> n * 2)
                                .collect(Collectors.toList());
```

目的：

代码简洁，通过函数组合，减少了冗余代码。

增强可读性，使用函数式风格使意图更加清晰。

（3）不可变数据结构

不可变数据结构在创建后无法被修改，任何更改都会返回一个新实例，在 Java 中，可以使用 Collections.unmodifiableList() 创建不可变列表：

代码如下：

```
List<String> original = new ArrayList<>(Arrays.asList("A", "B", "C"));
List<String> immutableList = Collections.unmodifiableList(original);
```

目的：

减少副作用，不可变性避免了数据被意外修改的问题。

提升并发安全性，在多线程环境中，避免数据竞争问题。