

**函数式编程原理课程报告**

|  |  |
| --- | --- |
| 姓 名： | 王峰羽 |
| 班 级： | CS2202 |
| 学 号： | U202215396 |
| 指导教师： | 顾琳 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2024年11月10日

目 录

[一、 Heapify求解 1](#_Toc7721)

[1.1 问题需求 1](#_Toc7432)

[1.2 解题思路与代码 2](#_Toc19529)

[1.3 运行结果 4](#_Toc31127)

[1.4 性能分析（请用树的深度进行分析） 5](#_Toc17516)

[二、 函数式拓展学习调研 6](#_Toc8160)

[2.1 函数式程序应用场景 6](#_Toc32030)

[2.2 函数式特征延伸 7](#_Toc27719)

# Heapify求解

## 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: 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. \*)。

## 解题思路与代码

在heapify函数中，我们定义并使用了另外两个函数:

treecompare 和 Swapdown

(\*begin\*)

datatype order = LT

| EQ

| GT;

fun treecompare (t1, t2) = case (t1, t2) of (Empty, Empty) =>EQ

| (Empty, \_) => LT

| (\_, Empty) => GT

| (Br (\_, a1, \_), Br (\_, a2, \_)) =>

if a1 < a2 then LT

else if a1 = a2 then EQ

else GT;

treecompare 函数目的是比较多棵二叉树的根节点值，并返回一个order类型的值，表示哪棵树的根节点值更大。

检查两棵树t1和t2是否为空。如果两棵树都为空，返回EQUAL。

如果t1为空，返回LESS，因为任何树的根节点值都大于空树。

如果t2为空，返回GREATER，因为任何树的根节点值都大于空树。

如果两棵树都不为空，比较它们的根节点值。如果t1的根节点值大于t2的根节点值，返回GREATER；如果小于，返回LESS；如果相等，返回EQUAL。

fun SwapDown (Empty) = Empty

| SwapDown (Br (Empty, x, Empty)) = Br (Empty, x, Empty)

| SwapDown (Br (Empty, x, Br (rl, rx, rr))) =

if x <= rx then Br (Empty, x, Br (rl, rx, rr))

else Br (Empty, rx, SwapDown (Br (rl, x, rr)))

| SwapDown (Br (Br (ll, lx, lr), x, Empty)) =

if x <= lx then Br (Br (ll, lx, lr), x, Empty)

else Br (SwapDown(Br (ll, x, lr)), lx, Empty)

| SwapDown (Br (Br (ll, lx, lr), x, Br (rl, rx, rr))) =

if x >= lx andalso x <= rx then Br (Br (ll, lx, lr), x, Br (rl, rx, rr))

else if x <= lx andalso lx <= rx then Br (SwapDown(Br (ll, x, lr)), lx, Br (rl, rx, rr))

else if lx <= rx andalso rx <= x then Br (SwapDown(Br (ll, lx, lr)), rx, SwapDown(Br (rl, x, rr)))

else if x >= rx andalso x <= lx then Br (SwapDown(Br (ll, rx, lr)), x, SwapDown(Br (rl, lx, rr)))

else if lx >= rx andalso lx <= x then Br (SwapDown(Br (ll, rx, lr)), lx, SwapDown(Br (rl, x, rr)))

else Br (SwapDown(Br (ll, x, lr)), rx, SwapDown(Br (rl, lx, rr)));

SwapDown 函数用于将一个节点“下沉”到最小堆中正确的位置，以保持最小堆的性质。

首先，检查当前树t是否为空或者其所有直接子节点是否为空。如果是，直接返回t，因为没有需要交换的节点。

如果t不为空，获取其左右子树l和r。

检查t的值是否小于其左右子树的值。如果是，说明t已经是最小堆，直接返回t。

如果t的值大于其左子树l的值，且左子树l的值小于或等于右子树r的值，那么将t的值与左子树的根节点值交换，并对左子树进行SwapDown操作。

如果t的值大于其右子树r的值，那么将t的值与右子树的根节点值交换，并对右子树进行SwapDown操作。

fun heapify Empty = Empty

| heapify (Br (l, x, r)) = let val leftHeap = heapify l;

val rightHeap = heapify r;

in SwapDown (Br (leftHeap, x, rightHeap))

end;

(\*end\*)

heapify 函数将任意二叉树转换为最小堆。

首先，检查树t是否为空。如果为空，直接返回空。

如果t不为空，递归地对t的左右子树调用heapify函数，将它们转换为最小堆。然后，使用SwapDown函数对根节点进行下沉操作，以确保整个树满足最小堆的性质。SwapDown操作会将根节点与其子节点进行比较和交换，直到整个树满足最小堆的性质。

## 运行结果

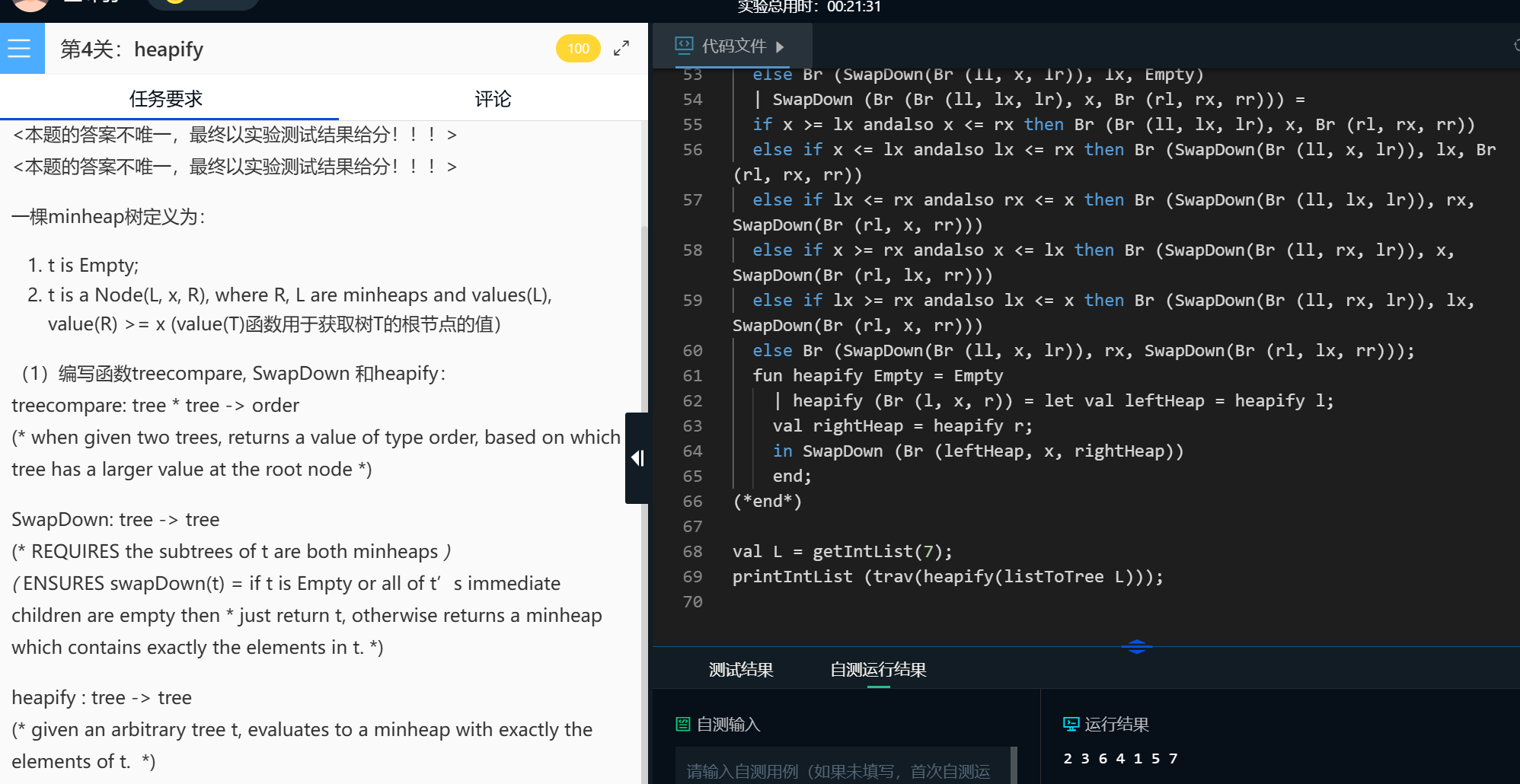


图1\_1

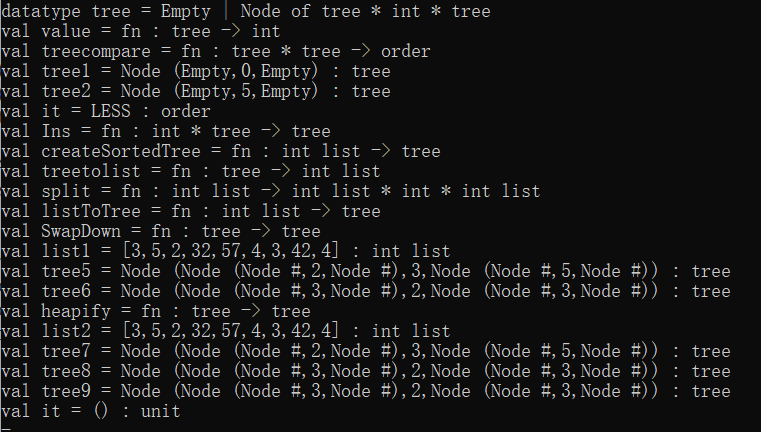


图1\_2

## 性能分析（请用树的深度进行分析）

heapify函数是递归的，并且通常以串行方式执行，因为它需要确保每个子树都是最小堆，然后才能正确地对根节点进行SwapDown操作。heapify操作可以并行化。如果树的节点可以分布在多个处理器上，可以同时对多个节点执行heapify操作。然而，由于父子节点之间的依赖关系，这种并行化需要在节点之间同步，以确保子树已经是最小堆。

heapify函数的时间开销是O(n)，其中n是树中的节点数。由于n ≈ 2^h，因此时间开销也可以表示为O(2^h)，即与树的高度呈指数关系。

# 函数式拓展学习调研

## 2.1 函数式程序应用场景

##### 1. 大数据处理和并行计算

在大数据处理和并行计算中，数据的一致性和并行操作的无副作用特性非常重要。函数式编程语言如Scala、Haskell或Erlang等，提供了强大的并行处理能力和对不可变数据结构的支持，这使得它们非常适合处理大规模数据集。

优势：

不可变数据结构：保证了数据在并行操作中不会被意外修改，从而避免了竞态条件和数据不一致的问题。

纯函数：纯函数没有副作用，这使得并行计算更加安全，因为不用担心函数之间的相互影响。

高阶函数：可以轻松地将函数作为参数传递，或者返回函数，这在编写复杂的数据处理管道时非常有用。

##### 2. 金融系统和风险管理

金融系统，尤其是那些涉及高频交易、风险管理和定价模型的系统，需要精确和可靠的计算。函数式编程语言能够提供精确的控制和可验证的代码行为。

优势：

精确控制：函数式编程允许开发者对程序的执行有更精确的控制，这对于需要精确计算的金融应用至关重要。

代码可验证性：纯函数和不可变数据结构使得代码更容易测试和验证，这对于金融系统的安全性和稳定性非常重要。

容错性：函数式语言如Erlang的容错机制（例如进程隔离和消息传递）对于构建高可用性的金融系统非常有用。

##### 3. 嵌入式系统和物联网(IoT)

嵌入式系统和物联网设备通常需要在资源受限的环境中运行，同时要求高可靠性和响应性。函数式编程语言如OCaml或Rust（虽然Rust不是纯粹的函数式语言，但它受到函数式编程的影响）可以提供这些特性。

优势：

资源效率：函数式编程语言通常能够生成高效的代码，这对于资源受限的嵌入式系统非常重要。

并发性：函数式编程语言通常提供良好的并发模型，这对于管理多个设备和传感器的IoT应用非常有用。

内存安全：语言如Rust提供了内存安全保证，这对于防止嵌入式系统中的内存泄漏和缓冲区溢出攻击至关重要。

## 2.2 函数式特征延伸

尽管Java和C++不是纯粹的函数式编程语言，但它们在近年来的版本更新中引入了许多函数式编程的特征，以提高代码的表达力、简洁性和并发处理能力。

##### Java中的函数式特征

1.Lambda表达式

目的： Lambda表达式允许你以简洁的方式表示单方法接口（functional interface）的匿名实现。这使得代码更加简洁，特别是在使用集合框架（如Stream API）时，可以写出更少的代码来处理集合。

例如，在Java 8中，可以使用Lambda表达式来简化集合的遍历和操作。

2.方法引用和构造器引用

目的： 方法引用提供了一种简洁的方式来引用现有的方法或构造器。这使得使用Lambda表达式时代码更加简洁。

例如，可以使用方法引用来简化集合的排序。

3.Stream API

目的： Stream API提供了一种声明式处理集合的方式，支持并行操作，可以提高性能并简化并发代码的编写。

例如，可以使用Stream API来处理集合数据。

##### C++中的函数式特征

1.Lambda表达式

目的： C++11引入了Lambda表达式，允许在需要函数对象的地方创建匿名函数。这使得代码更加灵活和简洁，尤其是在算法和并行编程中。

例如，使用Lambda表达式来简化数组的遍历和操作。

2.std::function

目的： std::function是一个通用的多态函数对象封装器，可以存储、调用和复制任何可调用对象。这使得函数式编程更加灵活，允许将函数作为参数传递。

例如，使用std::function来存储和调用函数。

3.std::bind

目的： std::bind用于创建一个可调用对象（函数对象），它在被调用时会调用一个函数（或成员函数）并传递给定的参数。这提供了一种方式来绑定函数参数，实现类似于Lambda表达式的功能。

例如，使用std::bind来绑定函数参数。

这些函数式特征的引入，使得Java和C++这样的命令式语言能够更好地处理高阶函数、函数组合和并发编程等场景，同时也提高了代码的可读性和可维护性。