北京工业大学

2020 - 2021 学年 第1学期

信息学部 计算机学院

|  |  |  |  |
| --- | --- | --- | --- |
| 课程名称： | 数据结构课程设计 | | |
| 报告性质： | 实验报告 | | |
| 学号： | 18074102 | 姓名： | 左帅 |
| 任课教师： | 王众 | 课程性质： | 学科基础必修课 |
| 学分： | 2 | 学时： | 60 |
| 班级： | 180701 | 成绩： |  |
| 教师评语： |  | | |

|  |  |  |
| --- | --- | --- |
| 需求分析 | 根据题中需求，提供功能划分说明 |  |
| 设计 | 逻辑结构、存储结构设计、算法描述 |  |
| 使用说明 | 界面是否友好 |  |
| 总结 | 是否感悟有收获 |  |
| 摘要 | 考察文字抽象能力 |  |
| 格式 | 是否有目录、页号 |  |

2020年10月23日

1需求分析

1.1需要完成的功能

电影分类可以按豆瓣评分排行、名著改编、地域（欧美，大陆，港台，日韩）、剧情（爱情、喜剧、犯罪、动画、历史等）、某导演（至少包含你本人欣赏的3位导演（彰显个性））、某演员作品（至少包含你本人关注的3位演员）、语言（英语、汉语、日语等）等。请设计一个查询系统，通过介绍用户感兴趣电影的剧情、导演、演员表、影评、豆瓣评分等信息，让用户充分了解此部电影，供用户参考。

1. 提供分类（至少包含上述7种），用户选择一种分类，在候选影片（要求按分类查询的搜索结果至少包含6部电影）中选择查询，而且各种分类之间要求有交叉；

1.1、豆瓣评分

1.2、名著改编

1.3、地域

1.4、剧情

1.5、导演

1.6、演员

1.7、语言

2、允许用户输入电影名称，直接查询；

3、按最热（依据本系统的点击率）、最新（本系统包含影片的上映日期）排行查询。

3.1、最热排行

3.2、最新排行

【要求】

1、选择最适合的数据结构和最适合的算法实现；

2、要求只能存储原始数据（电影基本信息及其分类特征（包含多个，格式自定义）），不允许存储统计结果（各种分类是按“分类特征”进行统计的结果）；对数据统计的结果要求能够依据原始数据的改变而变化。例如，文件保存的原始数据中(例如：某影片的剧情由喜剧改为了历史，在喜剧分类中，就看不到此片)，而不是存储最终的统计结果；

3、要求“原始数据”存储在文本文件中（如果数据是爬虫爬出来的大量真实数据，可以有限度的使用数据库，但不允许使用数据库中的插删查找功能解决本问题（要求用数据结构知识解决），并需要举例展示来证明没有使用插删查找功能），格式自定义。每次运行，要求从文件中读取数据。

4、能够展示每部影片的详细信息，包括 影片简介（编剧、导演、主演、类型、制片国家、语言、上映日期、片长）剧情简介、豆瓣评分、影评等等；

5、界面展示力争做到：言简意赅，形象生动，层次越少越好，界面信息越全越好，尽量减少用户的文字的输入；

2数据结构设计

2.1主要数据结构

2.1.1电影类（Film）

class film():

    '''def \_\_init\_\_(self,rating,adapt\_from,region,plot,types,director,actors,language,time):'''

    def \_\_init\_\_(*self*,*name*,*director*,*types*):

#构造函数，初始化电影名称，导演，类别等内容

    def clicked(*self*):

#调用click()后，clicked属性加1，位置后热度排序做准备

2.1.2AVL树类（AVLTree）

2.1.2.1节点类（film\_node）

    class film\_node:

        def \_\_init\_\_(*self*, *film\_name*=None,*director*=None,*types*=None):

            self.film\_name = film\_name

            self.left = None

            self.right = None

            self.height = 1

            self.film = film(film\_name,director,types)

*from* Film *import* film

class AVLTree:

    def \_\_init\_\_(*self*):

#构造函数，声明树的根节点self.root，设置树的节点数为0

    def get\_size(*self*):

#获取树的节点数

    def is\_empty(*self*):

#判断书是否为空

    def add\_from\_root(*self*, *film\_name*, *director*,*types*):

    def add(*self*, *node*, *film\_name*, *director*,*types*):

#添加节点

    def get\_height(*self*, *node*):

    #获取节点高度

    def get\_balance\_factor(*self*, *node*):

    #计算平衡因子balance\_factor

    def right\_rotate(*self*, *y*):

    #右旋

    def left\_rotate(*self*, *y*):

#左旋

    def get\_node(*self*, *node*, *film\_name*):

#搜索节点，根据node的film\_name值

    def get\_film\_from\_node(*self*,*film\_name*):

#获取搜索节点的film类

    def get\_film\_from\_node\_print(*self*,*film\_name*):

#获取搜索节点的film类，并打印其内容

    def contains(*self*, *film\_name*):

#返回是否包含对应的电影名

    def preorder\_traversal(*self*,*node*):

    def root\_preorder\_traversal(*self*):

#前序周游

    def root\_preorder\_traversal\_search(*self*,*types*):

    def preorder\_traversal\_search(*self*,*node*,*types*):

#前序周游，并比对电影的类型

    def delete(*self*):

*pass*

#节点的删除，尚未实现

2.1.3搜索类（Search）

*from* Film *import* film

*from* AVLTree *import* AVLTree

class Search():

    def \_\_init\_\_(*self*):

#构造函数，初始化函数，从文件读入数据，并建立avl树

    def show\_types(*self*):

#打印所有的类型

    def list\_all(*self*):

#打印所有储存的电影

    def search(*self*,*search\_str*):

#根据search\_str搜索对应的电影或对应的类型对应的电影

    def append(*self*):

*pass*

#添加，尚未实现

    def delete(*self*):

*pass*

#删除，尚未实现

3详细设计

3.1.1 Film.py

class film():

    '''def \_\_init\_\_(self,rating,adapt\_from,region,plot,types,director,actors,language,time):'''

    def \_\_init\_\_(*self*,*name*,*director*,*types*):

        self.name = name

*#self.rating = rating*

*#self.adapt\_from = adapt\_from*

*#self.region = region*

        self.types = types.split('/')

*#self.plot = plot*

        self.director = director

*#self.actors = actors*

*#self.language = language*

*#self.time = time*

*#self.types = types*

        self.click\_times = 0

    def clicked(*self*):

        self.click\_times = self.click\_times + 1

3.1.2 AVLTree.py

*from* Film *import* film

class AVLTree:

    class film\_node:

        def \_\_init\_\_(*self*, *film\_name*=None,*director*=None,*types*=None):

            self.film\_name = film\_name

            self.left = None

            self.right = None

            self.height = 1

            self.film = film(film\_name,director,types)

    def \_\_init\_\_(*self*):

        self.root = None

        self.size = 0

    def get\_size(*self*):

*return* self.size

    def add\_from\_root(*self*, *film\_name*, *director*,*types*):

        self.root=self.add(self.root, film\_name, director,types)

    def add(*self*, *node*, *film\_name*, *director*,*types*):

*if* not node:

            self.size += 1*#若节点为空，则创建一个节点*

*return* self.film\_node(film\_name, director,types)

*elif* node.film\_name > film\_name:

            node.left = self.add(node.left, film\_name, director,types)

*else*:

            node.right = self.add(node.right, film\_name, director,types)

*# 更新height,获取平衡因子*

        node.height = 1 + max(self.get\_height(node.left), self.get\_height(node.right))

        banlance\_factor = self.get\_balance\_factor(node)

*# 维护平衡性，共四种情况LL,RR,LR,RL*

*#LL*

*if* banlance\_factor > 1 and self.get\_balance\_factor(node.left) >= 0:

*return* self.right\_rotate(node)

*#RR*

*if* banlance\_factor < -1 and self.get\_balance\_factor(node.right) <= 0:

*return* self.left\_rotate(node)

*#LR*

*if* banlance\_factor > 1 and self.get\_balance\_factor(node.left) < 0:

            node.left = self.left\_rotate(node.left)

*return* self.right\_rotate(node)

*#RL*

*if* banlance\_factor < -1 and self.get\_balance\_factor(node.right) > 0:

            node.right = self.right\_rotate(node.right)

*return* self.left\_rotate(node)

*return* node

    def get\_height(*self*, *node*):

*if* not node:

*return* 0

*return* node.height

    def get\_balance\_factor(*self*, *node*):

*if* not node:

*return* 0

*return* self.get\_height(node.left) - self.get\_height(node.right)

    def right\_rotate(*self*, *y*):

        x = y.left

        T3 = x.right

*# 右旋转*

        x.right = y

        y.left = T3

*# 更新height*

        y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

        x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

*return* x

    def left\_rotate(*self*, *y*):

        x = y.right

        T2 = x.left

        x.left = y

        y.right = T2

*# 更新height*

        y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

        x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

*return* x

    def get\_node(*self*, *node*, *film\_name*):

*if* not node:

*return*

*if* node.film\_name == film\_name:

*return* node

*elif* node.film\_name > film\_name:

*return* self.get\_node(node.left, film\_name)

*else*:

*return* self.get\_node(node.right, film\_name)

    def get\_film\_from\_node(*self*,*film\_name*):

        s = self.get\_node(self.root, film\_name)

*if* s is not None:

*return* s.film

*else*:

*return* film('noname','nosuchdirector','')

    def get\_film\_from\_node\_print(*self*,*film\_name*):

        self.searched\_film = self.get\_film\_from\_node(film\_name)

*if* self.searched\_film.name != 'noname':

            print(self.searched\_film.name,self.searched\_film.director,self.searched\_film.types)

*else*:

            print('No Such Film!')

    def contains(*self*, *film\_name*):

*return* self.get\_node(self.root, film\_name) is not None

    def preorder\_traversal(*self*,*node*):

*if* node is not None:

            print(node.film\_name,node.film.director,node.film.types)

            self.preorder\_traversal(node.left)

            self.preorder\_traversal(node.right)

    def root\_preorder\_traversal(*self*):

        self.preorder\_traversal(self.root)

    def root\_preorder\_traversal\_search(*self*,*types*):

        self.flag = 0

        self.preorder\_traversal\_search(self.root,types)

*if* self.flag == 0:

            print("No Such Film!")

    def preorder\_traversal\_search(*self*,*node*,*types*):

*if* node is not None:

            i = 1

            kinds=types.split()

*for* kind *in* kinds:

*if* node.film.types.count(kind) < 1:

                    i = 0

*if* i == 1:

                self.flag = 1

                print(node.film\_name,node.film.director,node.film.types)

            self.preorder\_traversal\_search(node.left,types)

            self.preorder\_traversal\_search(node.right,types)

    def delete(*self*):

*pass*

3.1.3search.py

*from* Film *import* film

*from* AVLTree *import* AVLTree

class Search():

    def \_\_init\_\_(*self*):

        self.all\_types = []

        self.words = ''

*with* open('films.txt', 'r',*encoding*='utf-8') *as* f:

            self.words = f.read()

        self.words = self.words.split()

        self.avl=AVLTree()

*for* word *in* self.words:

            wd = word.split('#')

*if* self.avl.contains(wd[0]):

                print("duplicated")

*else*:

                self.avl.add\_from\_root(wd[0],wd[1],wd[2])

                tp=wd[2].split('/')

*for* typ *in* tp:

*if* self.all\_types.count(typ) < 1:

                        self.all\_types.append(typ)

    def show\_types(*self*):

        print('There are',len(self.all\_types),'types.')

        print(self.all\_types)

    def list\_all(*self*):

        print('There are',self.avl.size,'films.')

        self.avl.root\_preorder\_traversal()

    def search(*self*,*search\_str*):

        self.search\_word = search\_str.split()

        i = 0

*for* word *in* self.search\_word:

*if* self.all\_types.count(word) < 1:

                i = 1

*if* i == 0:

            self.avl.root\_preorder\_traversal\_search(search\_str)

*elif* search\_str == 'list':

            self.list\_all()

*elif* search\_str == 'types':

            self.show\_types()

*else*:

*#searched\_film = self.avl.get\_film\_from\_node(search\_str)*

            self.avl.get\_film\_from\_node\_print(search\_str)

    def append(*self*):

*pass*

    def delete(*self*):

*pass*

3.1.4 FilmSearchSystem.py

*#from AVLTree import AVLTree*

*#from time import time*

*from* search *import* Search

*if* \_\_name\_\_ == '\_\_main\_\_':

*#start\_time = time()*

    s=Search()

*#print('TreeCreate time: {} seconds'.format(time() - start\_time))*

*#s.list\_all()*

*#s.show\_types()*

    search\_str = input('Please Enter:')

*while* search\_str !='exit':

*#start\_time = time()*

        s.search(search\_str)

*#print('Search time: {} seconds'.format(time() - start\_time))*

        search\_str = input('Please Enter:')

4总结

在这次课设之前，我对这个问题是没有什么头绪的。在经历了一段时间的思考后，我对整个程序的逻辑结构有了一个大致的思路和方向，同时根据这些逻辑结构，我对索要采用的数据结构有了一个大致的了解。这个题目主要的问题就是对电影进行各种方式的查找。因此，程序的查找效率就是一个很大的问题，虽然在对名称查找时使用了avl树，但是在类型查找时仍需遍历所有节点。总的来说，虽然我对自己的进度还不是很满意，但是也算是完成了一定的目标。