数据结构 ： 用先序遍历建立二叉树 实验报告

毛子恒 李臻 张梓靖

2020 年 11 月 26 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：可视化 文档

目录

1 需求分析

1.1 题目描述

按照先序遍历输入一个二叉树，建立二叉树，输出该二叉树的各种表示形式。

1.2 输入描述

程序从标准输入中读入数据。
输入的第一行为一个字符串，表示二叉树按先序遍历所得的序列。用 * 字符代表空树。
之后按照提示输入选项，参考 5 用户使用说明。

1.3 输出描述

程序向标准输出中打印结果。
若选择输出图形化表示，则打印若干行，表示二叉树的图形。
若选择输出遍历结果，则打印一行，表示二叉树某种遍历的结果。
若程序出现运行时错误，则没有输出。

1.4 样例输入输出

1.4.1 样例输入输出 1

见图 1。

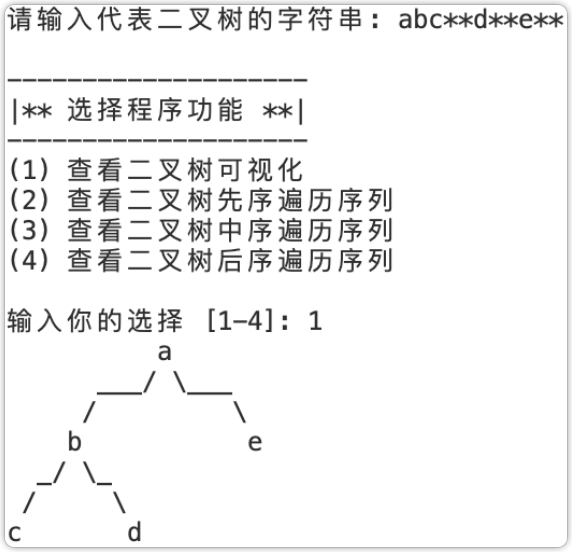

图 1: 样例输入输出 1

1.4.2 样例输入输出 2

见图 2。

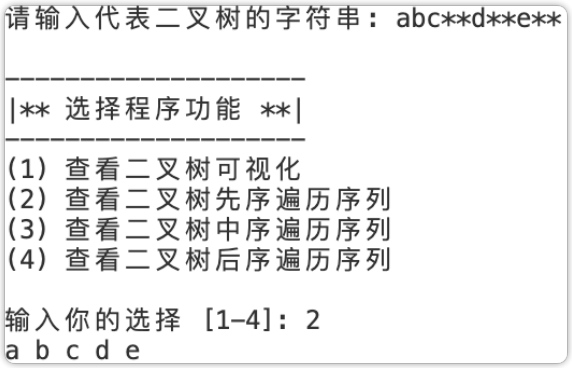

图 2: 样例输入输出 2

1.4.3 样例输入输出 3

见图 3。

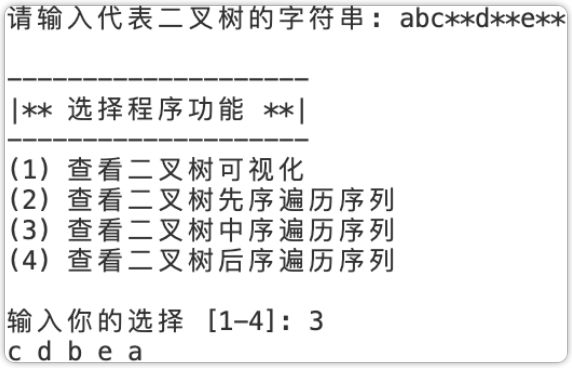

图 3: 样例输入输出 3

1.4.4 样例输入输出 4

见图 4。

1.5 程序功能

程序通过先序遍历序列生成二叉树，根据选项打印二叉树的图形或者某种遍历结果。

2 概要设计

2.1 问题解决的思路

使用二叉链表存储二叉树，并且设计输出二叉树图形的算法、二叉树的先序、中序、后序遍历算法。

2.2 二叉树的定义

```
1 //数据对象
2 typedef struct node
```

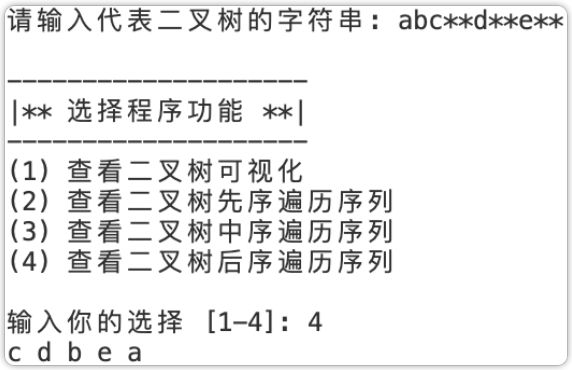

图 4: 样例输入输出 4

```
{
    char data;
    struct node * lc, * rc; // 左右孩子指针
    int pos; // 记录结点位置，用于输出二叉树图形
} Node;

/*
 * 操作：建二叉树
 * 前件：标准输入流中为一个字符串，代表二叉树
 * 后件：建立一个二叉树，返回指向该二叉树根节点的指针
 */
Node * buildTree();

/*
 * 操作：先序遍历
 * 前件：point指向一个二叉树的根节点
 * 后件：向标准输出中打印二叉树的先序遍历序列
 */
void preOrderTraverse(Node * point);

/*
 * 操作：中序遍历
 * 前件：point指向一个二叉树的根节点
 * 后件：向标准输出中打印二叉树的中序遍历序列
 */
void inOrderTraverse(Node * point);

/*
 * 操作：后序遍历
 * 前件：point指向一个二叉树的根节点
 * 后件：向标准输出中打印二叉树的后序遍历序列
 */
void postOrderTraverse(Node * point);

/*
 * 操作：输出二叉树图形
```

```
 * 前件：point指向一个二叉树的根节点
 * 后件：向标准输出中打印二叉树的图形
 */
void printGraph(Node * point);

/*
 * 操作：释放二叉树空间
 * 前件：point指向一个二叉树的根节点
 * 后件：释放该二叉树占用的空间
 */
void destroyTree(Node * point);
```

2.3 主程序的流程

1. 输入字符串
2. 建二叉树
3. 依照输入的选项打印相应的结果
4. 询问是否要继续选择其他选项，如果继续，回到第三步
5. 释放空间

2.4 各程序模块之间的层次关系

程序模块层次关系图如图 1。

图 5: 程序模块层次关系

3 详细设计

3.1 二叉树的实现

二叉树的设计中基本操作的伪代码算法如下：

```
// 生成二叉树
// 建二叉树
Node * buildTree()
{
    读入字符ch
    if (ch为'*')
        返回空指针
    创建新节点，并用point指针指向它，如果内存分配失败，异常退出
    point->data <- ch
    分别建point的左子树和右子树，并且更新lc和rc指针
    计算二叉树的图形中该点的位置
    每两个叶结点之间需要空出8个字符的宽度
    度为1的结点的空子树也要空出8个字符的宽度
```

```
    非叶结点的位置为两个孩子结点位置的平均值
    返回point
}

// 先序遍历
void preOrderTraverse(Node * point)
{
    输出point->data
    if (point的左子树不为空)
        preOrderTraverse(point->lc);
    if (point的右子树不为空)
        preOrderTraverse(point->rc);
}

// 中序遍历
void inOrderTraverse(Node * point)
{
    if (point的左子树不为空)
        inOrderTraverse(point->lc);
    输出point->data
    if (point的右子树不为空)
        inOrderTraverse(point->rc);
}

// 后序遍历
void postOrderTraverse(Node * point)
{
    if (point的左子树不为空)
        postOrderTraverse(point->lc);
    if (point的右子树不为空)
        postOrderTraverse(point->rc);
    输出point->data
}

// 输出二叉树图形
void printGraph(Node * point)
{
    定义队列和队列的头指针、尾指针
    初始化队列中的唯一一个元素为根结点
    while (h <= t)
    {
        每层的图形第一行：输出该层结点的字符数据，并用空格补齐间距
        换行
        每层图形的第二行：输出根节点下的斜杠和反斜杠，以及代表边下划线，并用空格补齐间距
        换行
        每层图形的第三行：输出孩子结点上的斜杠和反斜杠，孩子结点入队，并用空格补齐间距
        换行
    }
}
```

```
63
64 // 释放二叉树空间
65 void destroyTree(Node * point)
66 {
67     if (point的左子树不为空)
68         destroyTree(point->lc);
69     if (point的右子树不为空)
70         destroyTree(point->rc);
71     释放point指针
72 }
```

3.2 函数的调用关系图

函数调用关系图如图 1。

图 6: 函数调用关系图

4 调试分析报告

4.1 调试过程中遇到的问题和思考

在建树和遍历的过程中没有遇到太大问题，主要在输出二叉树的图形部分遇到比较大的困难。

最后决定采用两种实现方式，实现细节在 4.2 节和 7 节分别讨论。

4.2 设计实现的回顾讨论

在 C 实现的图形化当中，给每个叶结点和度数为 1 的结点的空子树留下长度为 8 的空格。通过'/'（斜杠），'\'（反斜杠），'_'（下划线）符号来表示二叉树的边。

由于需要兼顾时间复杂度和编程复杂度，在树中存在较多的链时会略微缺少可读性。

4.3 算法复杂度分析

printGraph 函数的时间复杂度为 $O(n^2)$。

其余函数的复杂度为 $O(n)$。

主函数的时间复杂度为 $O(1)$，整体时间复杂度为 $O(n^2)$。

整体空间复杂度为 $O(n)$。

5 用户使用说明

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c binarytree.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，首先按照提示输入 1.2 节描述的字符串，之后标准输出中会打印提示信息，根据提示信息选择相应的选项（数字 1~4），打印完成后根据提示，如果输入字符 c 则继续选择选项，输入字符 q 则退出程序。

如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

测试环节分为两个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

测试边界条件。

见图 7。

7 可视化

随机生成若干组数据，由三元组表实现和二维数组实现分别计算，比对运行时间，并且使用 JavaScript 将结果可视化。

比对脚本 (testing/timecount.py) 如下：

```
import os
import json
import time
a = []
b = []
for i in range(100):
    os.system("./data >in.in")
    starttime = time.time()
    os.system("./main <in.in >out.out")
    endtime = time.time()
    a.append(endtime-starttime)
    starttime = time.time()
    os.system("./test <in.in >out1.out")
    endtime = time.time()
    b.append(endtime-starttime)
    print(i)

json_str = json.dumps([a, b])
with open("80%result.json", mode="w") as file:
    file.write(json_str)
```

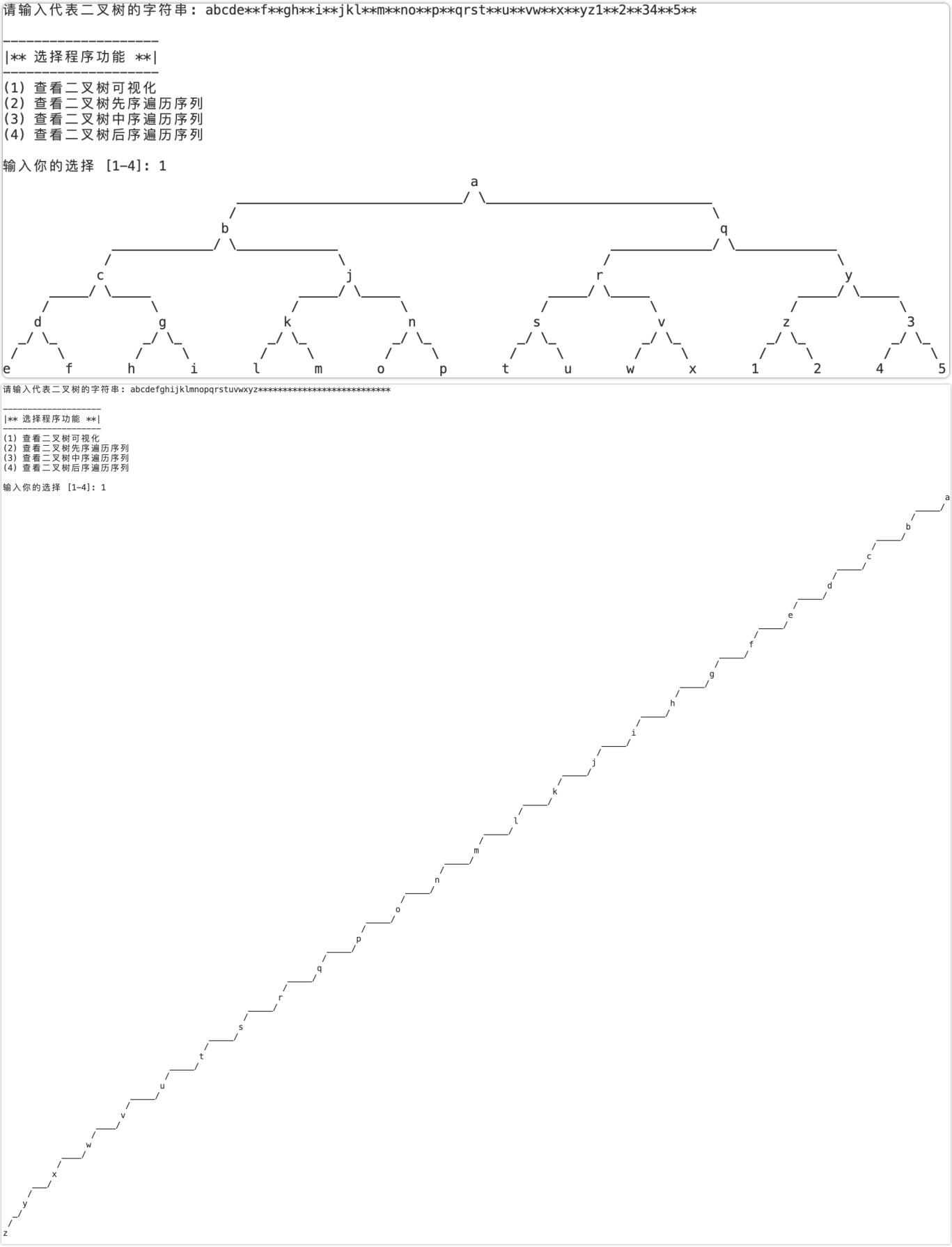
请输入代表二叉树的字符串： abcde**f**gh**i**jkl**m**no**p**qrst**u**vw**x**yz1**2**34**5**
(1) 查看二叉树可视化
(2) 查看二叉树先序遍历序列
(3) 查看二叉树中序遍历序列
(4) 查看二叉树后序遍历序列
输入你的选择 [1-4]: 1
请输入代表二叉树的字符串： abcdefghijklmnopqrstuvwxyz**********
(1) 查看二叉树可视化
(2) 查看二叉树先序遍历序列
(3) 查看二叉树中序遍历序列
(4) 查看二叉树后序遍历序列
输入你的选择 [1-4]: 1

图 7: 边界测试

数据规模如下：

$n = 1000$，20% 的元素是 0、50% 的元素是 0、80% 的元素是 0 的数据各 100 组，共 300 组数据。

7.0.1 实现细节

利用已有的时间数据，使用 Highcharts 库绘制二维柱状图，比较两种算法的运行时间差距。

7.0.2 用户使用说明

使用现代浏览器打开 Chart/index.html，即可看到柱状图，将鼠标指针放到某个柱上可以看到对应的值。

7.0.3 示例

见图 3。

图 8: 可视化示例

7.0.4 结论

可以看出在规模较大的数据中，相比二维数组的实现，三元组表的实现在绝大多数情况下都有明显的性能优势，并且当矩阵越稀疏时性能优势越明显。