report

Project A

程雨歌 12307110079

2016 年 6 月 8 日

1 Q1.1

1.1 任务

将一篇英文文章中出现过的所有单词以字典序排列（忽略大小写），并根据每种字符在文章中的出现频率进行 Huffman 编码（区分大小写）。

1.2 解决方法

1.2.1 按字典序排列

1. 建立 26 个链表，分别对应 26 个首字母的字典;
2. 在扫描文本时对单词进行分割，并将单词一律转为小写;
3. （关键步骤）将处理后的单词以插入排序法插入对应其首字母的字典中，每次比较用到字典序规则。
4. 依次打印 26 个字典（链表）。

1.2.2 Huffman 编码

1. 开辟数组统计所有单字节可打印字符的出现次数，并在扫描文本过程进行统计;
2. 建立一个储存 Huffman 树结点的堆，堆顶结点权重最小;
3. 将出现次数不为 0 的所有字符作为叶子结点放入堆中，其权重即为出现频率（次数）;
4. （关键步骤）构建 Huffman 树：每次弹出权重最小的两个节点，将其分别作为左右孩子连接到新建的父亲结点后将父亲重新放入堆中，父亲结点权重为子结点权重之和。反复此步骤直至堆中只有一个根结点;
5. 编码：遍历已构建的 Huffman 树并分配编码：根节点编码为空串，左孩子编码为父亲编码加一位'0'，右孩子加一位'1';
6. 按字典序打印所有字符的 Huffman 编码。

1.3 函数说明

1. ord // 返回字母序号，A/a 为 0（为 compareWord 服务）
2. compareWord // 返回两单词在字典序中先后顺序（为 insertWord 服务）
3. insertWord // 将单词按字典序插入已排序的指定字典中
4. goLowerCase // 将单词转为全小写
5. finNindexNcount // 从文件读入，将单词排序，统计字符出现次数
6. buildHuffman // 构建 Huffman 树（步骤即“解决方法”中相关部分）
7. printDict // 打印字典
8. swapNode // 堆操作：交换堆中两结点位置（为 push 与 pop 服务）
9. push // 堆操作：将结点压入堆中
10. pop // 堆操作：弹出堆顶结点
11. combine // Huffman 树操作：将两结点作为孩子链接到新建父亲结点
12. draw // 画树（检查程序用）
13. assignCode // 遍历 Huffman 树并分配编码
14. printHuff // 按字典序打印字符的 Huffman 编码

1.4 关键函数复杂度分析

1.4.1 insertWord

时间复杂度 若 L 为单词最大长度，N 为文章单词数量，则逐个字符比对的 compareWord 时间复杂度为 $\mathcal{O}(L)$，而 insertWord 采用从链表头线性逐个比较，符合条件后插入故时间复杂度为 $\mathcal{O}(LN)$。最坏情况既所有单词首字母相同且每次插入的单词的字典序都排在最后。可以使用二分查找法插入将时间复杂度降为 $\mathcal{O}(L\log N)$，但由于链表旅行操作来回定向反而更费时间，需要使用哈希表储存指针，用空间换取时间，以真正达到上述复杂度。

空间复杂度 $\mathcal{O}(1)$

1.4.2 buildHuffman

时间复杂度 buildHuffman 函数由两部分组成。第一部分为依次 push 所有字符，其权重即其出现次数。第二部分为不断合并结点直至成为一颗树，每次合并由两次 pop、一次 combine 和一次 push 组成。pop 为将堆顶元素和最后一个元素对调后将新堆顶元素不断和更小的孩子交换直至堆重新变为最小堆，push 则将待插入元素放入堆底并不断和大的父亲交换直至堆变为最小堆，两个函数时间复杂度都为 $\mathcal{O}(\log N)$；combine 复杂度为 $\mathcal{O}(1)$。整个过程需要重复 (N-1) 次。故 buildHuffman 时间复杂度为 $\mathcal{O}(N\log N + 3(N-1)\log N) = \mathcal{O}(N\log N)$

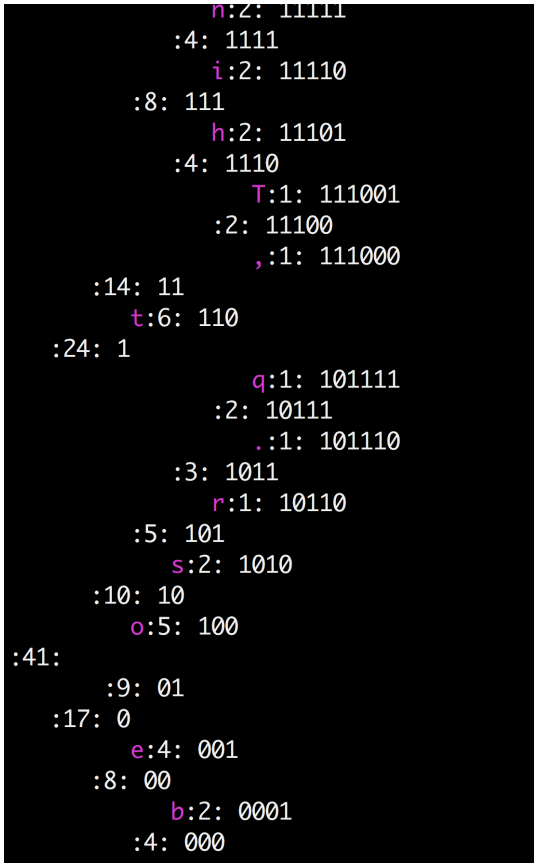
空间复杂度 $\mathcal{O}(1)$

1.5 边界情况分析

- 本程序采用逐行读取而非逐字读取，一次性将整行文字读入 buff 中，便于进行单词分割。然而 BUFFSIZE 设为了固定的值，若出现越界则超出部分无法读入。事实上若预先检查一下是否越界，若越界则使用 realloc 扩大 buff 大小即可，这样在申请的时候不用申请那么大节省了空间，同时也避免了越界问题。
- 同样本程序堆的大小 HEAPSIZE 也固定了，但由于只统计 ASCII 码为单字节的可打印字符，即 ASCII 码在 30 到 127 之间的字符，故堆大小足够，且也必须。
- 每个单词长度的上界 WORDLEN 设为固定。若考虑输入文本为无意义乱码或单词间隔被删除的情况，应先预处理文本再调用本程序，预处理超出了本程序应该解决的问题范畴。
- 字符 Huffman 编码的长度上限 CODELEN 也没有问题，最差情况即为 Huffman 树退化为一个每个结点其中一个儿子都是叶子的情况，也就是字符出现频率形如 1，2，4，8 这样前面最小的 n 个加起来也没有第 n+1 个大的这种情况，考虑所统计字符总数的限制，编码长度不会越界。
- 对于其他编码的文字，如 UTF-8 中汉字等字符需要占用多个字节（一般为 3 个）储存时，单独解析其中的每一个字节可能会得到无意义的内容，然而第一位符号位为负解决了这个问题（事实上也是由于数组下标出现负数报错而发现了这一情况），所以当碰到字符编码不在所统计范围内时并不报错，只是提出警告并跳过即可。其实这也应属于文本预处理的内容。

1.6 程序运行结果

下图是将生成的 Huffman 树打印出来：



文件夹中 input1.txt 为 Huffman 编码的中文维基百科页面中全部内容（包括网页上所有信息），程序处理结果也非常理想，详见 Q1.1/dictionary.txt 和 Q1.1/huffman.txt。

2 Q1.2

2.1 任务

分别用堆和多链表这两种形式维护一个优先队列，队列中有先来后到的、不同紧急级别的病人，级别高的优先，同级别的先来的优先。病人会来到（enqueue）、中途退出（quit）、更新紧急级别（update），医院需要随时知道下一个该让哪个病人看医生（dequeue），以及所有病人的优先顺序（print）。

2.2 用堆实现解决方法

1. 堆本身的实现不再赘述。注意 quit 的病人可能在堆中任意位置，故用一个数组来根据病人 ID 记录病人在堆中的位置。本质上是一个不太强的哈希表，将病人 ID 对 IDSCALE（本程序为质数 1000003）取模后为其储存位置。

2. quit 和 dequeue 后期对堆重新整理的过程都要用到“将代替其位置的元素不断下滚到合适位置”这一过程，故抽象出来一个函数叫做 rollDown。

3. 为区别同样级别病人，记录病人来到时间，时间随病人来到而增加 1，初始为 0。
4. update 采用病人先退出，再以更新后级别、更新时时间作为一个新的病人加入排队实现。
5. print 采用先复制整个堆，再依次出队实现。

2.3 用多链表实现解决方法

1. 动态地为每个级别的病人建立一个双向链表，元素顺序为病人优先顺序。将所有级别的链表头串成一个高级链表，链表头顺序为紧急级别顺序。
2. 每当新病人出现时，若已有其级别链表则将其插入该链表，若无则在恰当位置建立新链表并插入。
3. 出队时返回并删除将第一个链表的第一个元素。
4. 出队或中途离开时，需检查该级别链表是否变为空，若是则需将链表头在高级链表中删除。
5. 更新级别也采用离开再以新病人加入两部来实现。
6. 中途离开和更新级别同样用到根据 ID 储存每个病人指针的办法，快速找到病人位置并进行操作。

2.4 函数说明

1. goLowerCase // 将单词变全小写（用于对比识别操作指令）
2. myAtoi // 将字符串转换为整数（处理病人的 ID 和紧急级别）
3. isLetterDigit // 返回是否为字母或数字（用于分割输入的各参数）
4. swapPatient // 交换两病人在堆中位置（为 enQ 和 rollDown 服务）
5. whoRules // 比较两病人优先顺序（见任务描述）
6. enQ // 将病人加入堆／多链表中
7. rollDown // 将堆中制定位置病人下滚到合适位置（为 deQ 和 quit 服务）
8. deQ // 返回并删除优先级最高的病人
9. quitQ // 指定 ID 病人退出堆／多链表
10. update // 指定 ID 病人更新紧急级别
11. printQ // 按优先顺序打印所有病人
12. mainLoop // 主循环：读取操作类型、相关参数，并执行操作

2.5 关键函数复杂度分析

2.5.1 enQ（堆）

时间复杂度 N 为同时在排队的最大病人数，时间复杂度为 $\mathcal{O}(\log N)$。

空间复杂度 $\mathcal{O}(1)$

2.5.2 enQ（多链表）

时间复杂度 N 为同时在排队的同级别最大病人数，L 为紧急级别的数量。需要从高级链表头走到对应级别链表头，再走到队尾，时间复杂度为 $\mathcal{O}(L+N)=\mathcal{O}(N)$（因为级别数量不会大于病人数量，即最多每个病人都不同级别）。其实可以记录一下每个级别链表的尾结点指针，这样只用 $\mathcal{O}(1)$ 就可以了，但是需要空间 $\mathcal{O}(L)$。

空间复杂度 $\mathcal{O}(1)$

2.5.3 deQ（堆）

时间复杂度 N 为同时在排队的最大病人数，时间复杂度为 $\mathcal{O}(\log N)$。

空间复杂度 $\mathcal{O}(1)$

2.5.4 deQ（多链表）

时间复杂度 由于可以直接根据 ID 找到该病人指针，时间复杂度为 $\mathcal{O}(1)$。

空间复杂度 $\mathcal{O}(1)$

2.5.5 quit（堆）

时间复杂度 同 deQ（堆），时间复杂度为 $\mathcal{O}(\log N)$。

空间复杂度 $\mathcal{O}(1)$

2.5.6 quit（多链表）

时间复杂度 $\mathcal{O}(1)$。

空间复杂度 $\mathcal{O}(1)$

2.5.7 update（堆）

时间复杂度 quit+enQ，时间复杂度为 $\mathcal{O}(\log N)$。

空间复杂度 $\mathcal{O}(1)$

2.5.8 update（多链表）

时间复杂度 quit+enQ=$\mathcal{O}(N)$。

空间复杂度 $\mathcal{O}(1)$

2.5.9 print（堆）

时间复杂度 复制堆需要 $\mathcal{O}(N)$，N 次 deQ 需要 $\mathcal{O}(N\log N)$，故总时间复杂度为 $\mathcal{O}(N\log N)$。

空间复杂度 $\mathcal{O}(N)$（复制出来的堆）

2.5.10 print（多链表）

时间复杂度 $\mathcal{O}(N)$

空间复杂度 $\mathcal{O}(1)$

2.6 边界情况分析

- 文件读入手段同前一题，故 BUFFSIZE 分析同前一题，但规范输入不会太长，若假定病人 ID 和紧急级别都不会超过 int 整形大小，病人姓名不会超过 NAMESIZE（本程序默认为 200）。
- 在用堆实现中，病人到达时间 T 也为整形，堆大小 HEAPSIZE 为 1048576。根据经验，即使是北京／上海排队挂号最火热的大医院，一个科室门诊一天最多也就在数千个号这个数量级，所有门诊、急诊加起来也不会超过 10^6 这个数量级，故每天重启程序即可。
- 所有 dequeue 操作皆会判断队列是否为空，所有 update、quit 均会判断指定 ID 病人是否存在，其他各种判空情形也在程序中写出，并分别返回警告或错误。

2.7 程序运行结果

用堆实现：

```
$ ./Q1.2.heap
successfully opened input2.txt.
successfully initialized.
interpreting operation:[enqueue].
Aka, 1001, lvl.2 comes to queue.
interpreting operation:[enqueue].
Bob, 1002, lvl.4 comes to queue.
interpreting operation:[enqueue].
Cal, 1003, lvl.1 comes to queue.
interpreting operation:[quit].
Cal, 1003, lvl.1 quits the line.
interpreting operation:[enqueue].
Doh, 1004, lvl.1 comes to queue.
interpreting operation:[update].
Doh, 1004, lvl.1 -> lvl.3 updated.
interpreting operation:[dequeue].
Bob, 1002, lvl.4 goes to the doctor.
interpreting operation:[print].
╔═══════════════════════════════════
╟Waiting list (with descending priority):
╟   1. Doh, 1004, lvl.3
╟   2. Aka, 1001, lvl.2
╚═══════════════════════════════════
closing files.
exiting.
```

用多链表实现（可以容易判断某级别是否还有病人）：

```
$ ./Q1.2.multiQ
successfully opened input2.txt.
successfully initialized.
interpreting operation:[enqueue].
Aka, 1001, lvl.2 comes as first of his level.
interpreting operation:[enqueue].
Bob, 1002, lvl.4 comes as first of his level.
interpreting operation:[enqueue].
Cal, 1003, lvl.1 comes as first of his level.
interpreting operation:[quit].
Cal, 1003, lvl.1 quits the line, no more lvl.1 patients.
interpreting operation:[enqueue].
Doh, 1004, lvl.1 comes as first of his level.
interpreting operation:[update].
Doh, 1004, lvl.1 -> lvl.3 updated.
interpreting operation:[dequeue].
Bob, 1002, lvl.4 goes to the doctor, no more lvl.4 patients.
interpreting operation:[print].
╔═══════════════════════════════════
╟Waiting list (with descending priority):
╟   1. Doh, 1004, lvl.3
╟   2. Aka, 1001, lvl.2
╚═══════════════════════════════════
closing files.
exiting.
```

3 Q1.3（报告从略）

3.1 任务

根据二叉树的先序遍历和中序遍历结果重构树，将树镜像翻转，并输出翻转后树的后序遍历结果和双孩子结点个数。

3.2 解决方法

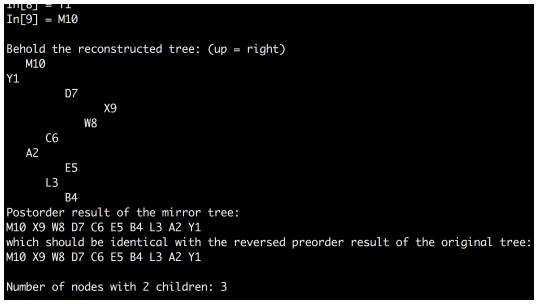
重构树：

1. 先序遍历结果第一个元素为根节点，在中序遍历中找到他；
2. 中序遍历中根节点左右串分别为其左右子树，在先序遍历中对应标记他们；
3. 分别对左右子树的先序遍历和中序遍历结果递归地运用本方法。

翻转树：递归地翻转根节点坐字数和右子树。

说明： 事实上无需翻转树，镜像树的后序遍历结果就是在原本树中：先遍历右子树再遍历左子树再访问根节点的结果。双孩子结点树也不随翻转而变化。

3.3 程序运行结果



4 Q1.4

4.1 任务

读入两个数 x、y，输出其和 x+y、差 x-y、乘积 x×y，若第二个数为整数则还要输出幂 x^y。

4.2 解决方法

4.2.1 加法

将两数按小数点对齐，从低位到高位逐位相加、进位。（仅处理两个正数）

4.2.2 减法

先处理两数大小和符号，通过再次调用加法或减法，然后符号单判断，使得只处理两个正数的大数减小数。

然后同样是按小数点对齐，从低位到高位逐位相减，若出现负数则借位并更新。

因为保证绝对值上是大数减小数，所以无需最后从高位开始检查是否为负数，逐渐向低位回退。

4.2.3 乘法

首先计算乘积的小数位数，将两个数转化为整数进行乘法运算，然后再加上小数点和符号。

整数乘法采用 karastuba 算法：将两个大整数从中间切开，计算高位和低位分别对应相乘结果，计算高位低位之和相乘，再用加减法和移位（十进制）实现原本两数乘法结果的计算。这样只用对原来大整数长度一半长的整数进行乘法运算，递归地调用 karastuba 算法即可。

在两个数位数均不超过 1000 时为平凡情况，按照小学竖式乘法运算即可：计算两个数每个位置之间互相乘积并加到结果的对应位上，最后从低位开始向高位进位，再从高位向低位回退前导零位数。

4.2.4 幂

若 y 为偶数，则 $x^y = x^{[y/2]} \times x^{[y/2]}$，否则 $x^y = x^{[y/2]} \times x^{[y/2]} \times x$，其中 $[y]$ 为不超过 y 的最大整数（即向下取整）。

按照以上规则递归地调用求幂函数，在 0、1、2 次方时给出边界情况的答案：1、x、x×x。

4.3 函数说明

1. initNum // 初始化 Num 类型：包括数字位数、符号、是否整数、小数点位置及各位数字
2. readNum // 从字符串指定位置将数读取到指定 Num 类型
3. printNum // 打印 Num 类型（可控制仅屏幕输出或同步输出到文件）
4. cutNum // 将指定 Num 类型在指定位置提取片段作为新的 Num 类型
5. paste // 左移位：在 Num 后补零（为 multiply 服务）
6. minus // 返回相反数（为 substract 服务）
7. compareNum // 比较两数大小（为 substract 服务）
8. fracToInt // 将小数转化为整数（为 multiply 服务）
9. intToFrac // 将整数在指定位置插入小数点（为 multiply 服务）
10. karastuba // 大整数快速乘法的递归实现（multiply 主要内容）
11. divideTwo // 返回除以二向下取整（为 power 服务）
12. add // 加法
13. substract // 减法
14. multiply // 乘法

15. power // 幂运算

16. mainLoop // 主循环：读取 x 和 y，计算并输出所需结果。

4.4 关键函数复杂度分析

设 N 为两个数中位数较长的那个数的位数。

4.4.1 add, substraction

时间复杂度 $\mathcal{O}(N)$

空间复杂度 $\mathcal{O}(N)$ 由于返回的 Num 类型皆为新开辟的，故此为最少需要空间，下同。

4.4.2 multiply

在平凡情况下：

时间复杂度 $\mathcal{O}(N^2)$

空间复杂度 $\mathcal{O}(N)$

调用的 karastuba 函数，它將兩個 n 位數字相乘所需的一位數乘法次數減少到了至多 $3n^{\log_2 3} \approx 3n^{1.585}$（引自 wiki）而平凡情况用了 1000 位乘法来加速，故时间复杂度要比 $\mathcal{O}(N^2)$ 小一点。

4.4.3 power

时间复杂度 $\mathcal{O}(N^2 \log N)$，N^2 为调用乘法的时间。

空间复杂度 空间复杂度上，由于每次递归只需要算一次一半幂，实际上可以用尾递归加速，一共要调用 $\log N$ 次，故总空间复杂度为 $\mathcal{O}(N \log N)$。

4.5 边界情况分析

- BUFFSIZE 分析，同前；数字位数上界 MAXLEN 采用固定值（本程序为 1048576），因为连续数组无法分配太大空间，但其实同样也可以用链表实现动态大小。
- 0^y 强行定义为 0，即使是 0^0。
- 在读入数后先将其和 0 相加，主要是清除前导零、小数点后尾缀零。

程序运行结果请见 Q1.4/output4.txt。

更多精细判断和操作都在程序源代码里啦！辛苦助教老师！