1.扩展 L25 文法的 EBNF 表示

<program> = "program" <ident> "{" {<func_def>} "main" "{" <stmt_list> "}" "}"

<func_def> = "func" <ident> "(" [<param_list>] ")" "{" <stmt_list> "return" <expr> ";" "}"

<param_list> = ["*" | "str"] <ident> { "," ["*" | "str"] <ident> }

<stmt_list> =<stmt> ";" {<stmt> ";" }

<stmt> = <declare_stmt> | <assign_stmt> | <if_stmt> | <while_stmt> | <input_stmt> | <output_stmt> | <func_call> | <alloc_stmt> | <free_stmt> | <return_stmt> | <update_stmt>

<declare_stmt> = "let" ["*" | "str" | "map" | "set"] <ident> ["=" <expr>]

<assign_stmt> = <ident> "=" <expr> | <dereference> "=" <expr>

<if_stmt>= "if" "(" <bool_expr> ")" "{" <stmt_list> "}" ["else" "{" <stmt_list> "}"]

<while_stmt> = "while" "(" <bool_expr> ")" "{" <stmt_list>"}"

<func_call> = <ident> "(" [<arg_list>] ")"

<arg_list> = <expr> { "," <expr> }

<input_stmt> = "input" "(" <ident> {"," <ident>} ")"

<output_stmt> = "output" "(" <expr> { "," <expr> } ")"

<alloc_stmt> = "alloc" "(" <ident> "," <expr> ")"

<free_stmt> = "free" "(" <ident> ")"

<return_stmt> = "return" <expr>

<update_stmt> = <ident> "." ("insert" "(" <expr> ["," <expr>] ")" | "delete" "(" <expr> ")") | <ident> "[" <expr> "]" "=" <expr>

<bool expr>= <expr> ("==" | "!=" | "<" | "<=" | ">" | ">=") <expr>

<expr> = ["+" | "-"] <term> { ("+" | "-") <term> }

<term> = <factor> {("*" | "/") <factor> }

<factor>= <ident> | <number> | "(" <expr> ")" | <func_call> | <reference> | <dereference> | <string> | <traversal> | <size> | <look_up> | <find>

<reference> = & <ident>

<dereference> = * <ident> | <ident> "[" <expr> "]"

<find> = <ident> "." "find" "(" <expr> ")"

<look_up> = <ident> "[" <expr> "]"

<traversal> = <ident> "." "traverse" "(" ")"

<size> = <ident> "." "size" "(" ")"

<ident> =<letter>{<letter>|<digit>}

<number> = <digit> {<digit>}

<letter>= "a" | "b" | ... | "z" | "A" | "B" |...| "Z"

<digit> = "0" |"1" |...| "9"

<string> = """ { <char> } """

<char> = set(ASCII) - { """ }

<comment> = "#" { set(ASCII) }

2.代码及扩展实现

项目使用 C 语言实现。除了基本文法外，编译器还实现了简单的以#开头的单行注释。

扩展项目中，实现了包含内存管理在内的指针运算，包括指针定义（"let" "*" <ident> ["=" <expr>]），引用（& <ident>），解引用（* <ident> | <ident> "[" <expr> "]"），指针赋值，手动分配空间（"alloc" "(" <ident> "," <expr> ")"），手动回收空间（"free" "(" <ident> ")"）。

之后，基于指针实现了字符串运算，包括字符串定义（let str a），字符串字面量（”abcd”），字符串的+和*运算；除此之外，读入、输出语句也支持了字符串，指针的解引用（<ident> "[" <expr> "]"）操作也适用于字符串。

同时，基于指针还实现了 map 类和 set 类，包括定义（"let" ("map" | "set") <ident>），插入（<ident> "." "insert" "(" <expr> ["," <expr>] ")" | <ident> "[" <expr> "]" "=" <expr>），删除（<ident> "." "delete" "(" <expr> ")"），查找（<ident> "." "find" "(" <expr> ")" | <ident> "[" <expr> "]"），遍历（<ident> "." "traverse" "(" ")"），返回大小（<ident> "." "size" "(" ")"）操作。

2.1 基本语法的实现方式

对于非扩展部分的词法分析，沿用原有的 getch()和 getsym()函数进行设计，调用方式不变，只需更改保留字和符号的集合。为了加入注释功能，借用 getsym()对空格的处理，将#

加入，并在读取到#时将整行剩余部分读取。

对于<program>，在把 program 标识符加入符号表，生成跳转指令后，逐个处理函数定义。函数定义完毕后，确定 main 的代码地址，回填到开头跳转指令处，之后生成初始化指令分配空间。Main 处理完毕后，确定需要分配的栈空间，回填到初始化指令处。

对于<func_def>、<param_list>、<return_stmt>和<func_call>，形参列表被当作函数内定义的变量处理。在作用域的实现上，形式参数在函数后失效，符号表的管理与 stmt_list 一致，可见下文说明。在函数被调用时，传入的参数首先依序被压栈，之后在函数分配空间后被读入栈顶，再存入形参中。然而，考虑到函数释放时，在函数栈帧生成前被压入的这部分参数无法被原编译器的逻辑出栈，因此改写函数返回指令，将其不用的.l 字段作为函数参数个数保存在指令中，在函数栈帧释放后释放压入的参数。对于返回值，在函数最末尾被计算完毕后存在栈顶，随后被存入临时变量中，在函数栈帧和传入参数被释放后重新压回栈顶。对于函数中多次出现的 return 语句，编译器都将其编译为跳转语句，目标地址为函数全部编译完成后的函数返回指令。

对于<stmt_list>，发现所有左部为<stmt_list>的式中都需要两端有大括号，因此将两端的大括号也加入处理范围。同时，在作用域的实现上，处理一个 stmt_list 时首先记录符号表当前的指针，在 stmt_list 结束后将符号表指针还原到 stmt_list 开始时的位置。

对于<declare_stmt>和<assign_stmt>，处理逻辑与原编译器类似。对于定义时的初始化通过在定义后马上赋值实现。

对于<if_stmt>和<while_stmt>，处理逻辑也与原编译器类似。对于 else 语句的实现，需要在 if 后的 stmt_list 处理完毕后，额外生成一条 jmp 跳转指令跳转到 else 后，再将 if 后的条件跳转指令的目标设为 jmp 跳转指令的下一条语句即可。

对于<input_stmt>和<output_stmt>，处理逻辑也与原编译器类似，读入语句读入的变量压入栈顶，输出语句输出栈顶的值并将栈顶值出栈。

对于<bool expr>、<expr>和<term>，处理逻辑也与原编译器类似，这里不再赘述。

对于<factor>，由于上文介绍过<func_call>会将返回值压入栈顶，因此只需将上文实现的函数调用逻辑在此处复用即可。

至此，基本语法已经实现完毕。

2.2 指针的实现方式

由于 c/c++语言的字符串、字典和集合都是通过指针实现的，先考虑指针的实现。实现逻辑上，dereference 返回该变量在栈中的地址；reference 代表该变量所在层级对应地址内的

值。

在词法分析上，仍仅需更改保留字和符号的集合。

对于指针声明，需要在 let 后、变量名前加上*符号来表明这个变量是一个指针类。由于函数定义时就已经确定了形参的类型，指针类作为形参时需要在函数的参数列表内的变量名前显式加上*来代表这个变量是一个指针变量。指针可以在初始化的时候赋值，赋值的类型可以是指针变量、引用，考虑到空指针的存在，还可以给指针赋 0 值表示空指针。在符号表内加入 is_ptr 成员来表示这个变量是指针类变量，以防错误的分配空间或解引用。

对于指针的内存管理，需要更改运行堆栈，增加容量并使其分为堆区、栈区。其中，栈区为 s 数组的首 stacksize 个数据区，堆区为 s 数组的后 heapsize 个数据区。动态分配、释放的空间均在堆区，其余操作则均在栈区完成。出于便于分配的考虑，还需要维护一个内存可用性数组，其中 0 代表未使用，1 代表已使用。新定义 alloc、fre 指令，用于堆区的分配、释放。通过 alloc 分配 n 个数据空间时，首先在可用性数组内寻找可分配的连续空间。找到后，在数据空间头部加入分配的数据空间大小、引用计数器（在字符串中介绍），之后才开始分配数据空间；因此，实际分配的空间是 n+2 个数据空间。分配时，仅需将可用性数组中相应的空间标记为已使用。分配后，指针指向可用分配数据空间的第一块，即分配内存块的第三块数据空间。fre 释放指针时，只需读取堆中存储的空间大小，将可用性数组中分配空间与数据头一同标记为可用。

对于引用，新定义一个 addr 指令来获得变量在栈内的地址。

对于解引用，新定义 loda、stoa 指令来对某块地址进行读写。对[]操作，由于需要读取中括号内的地址偏移量之后获取偏移后的地址，可能访问不合法的内存，因此在 opr 内添加一个安全内存加法指令，在计算偏移量时检查偏移后的地址是否合法。

2.3 字符串的实现方式

借助指针实现字符串。在词法分析上，仍仅需更改保留字和符号的集合。

对于声明，需要在 let 后、变量名前加上 str 符号来表明这个变量是一个字符串类，符号表内通过 is_str 来表示这个变量是否为字符串变量。对于函数参数，处理与指针类相同，也需要在变量名前加上 str 符号。

一开始考虑的字符串的实现方式有两种：一种是在栈中存入指针，堆中存入字符串；一种是在栈中直接存入字符串。本次实现采用的是第一种方式。然而，要实现这种方式的字符串的完整内存管理，实际非常复杂。这里分别考虑字符串字面量与字符串变量的管理。

首先考虑字符串字面量。在词法分析上，字符串字面量通过双引号识别，处理容易。但

在实际实现上，字符串字面量的内存管理却很难处理。参考 c 语言的实现，在堆栈区后又加入静态区来保存只读的数据，例如字符串字面量。与其他类型的字面量不同，在编译器读取到字符串字面量时，就应该将其存入静态区中并保存其首地址。当需要使用时，使用 strlit 命令获取字符串首地址。如果需要运算或作为函数返回值的话，需要在堆区内创建一个副本（通过定义的 cpystr 指令实现），并在栈顶保存对应堆区的值。至此，程序的堆栈区结构变为：[栈区 | 堆区 | 静态字面量区]，并且在之后的修改中不变。

然而，字符串变量在堆区创建的副本和字符串变量则非常难处理，因为都需要考虑空间的分配、回收。由于字符串需要支持+、*运算，而且与指针的手动分配、回收不同，字符串的内存需要自动分配、回收。本次实现中采取的方案是：在为字符串分配的头部多存储一个引用计数器作为该空间的回收标志，当这个计数器为 0 时回收字符串空间。存储结构与手动分配的空间同步，采取[分配空间大小 | 引用计数器 | 实际可用空间]的模式。使用 refinc、refdec 指令管理字符串的引用计数器，字符串变量被创建或被引用时，这个引用计数器+1；当指向它的变量指向另一块存储区或变量离开作用域时，引用计数器-1，当引用计数器-1 后为 0 时，字符串的空间应当被回收。特别是作用域的变化，变量离开作用域的处理放在符号表处理之前，在恢复符号表指针前，需要递减这个作用域内所有字符串变量的引用计数器。作为参数传入时，与变量赋值同理；作为返回值返回时，需要将其计数器+1，防止其因离开作用域而马上失效。

在完成以上实现后，字符串运算则相对容易处理：定义 concat 和 repeat 指令处理字符串运算，在运算时，先预测需要多少内存，之后新分配一块符合大小的内存，将字符串运算的结果放入分配的内存中，并将两个变量的引用计数器减少。对于整数加字符串的操作，需要增加将栈顶整数转换为字符串的操作。这里发现需要运行类型检查，考虑在对 expr、term、factor 的处理中加入返回类型，这样编译器就能检测到这个表达式返回值的类型，从而可以区分不同的操作。

同时，基于指针实现的字符串虽然不支持引用操作，但也支持和指针一样的*、[]解引用操作，实现方式与指针也相同，这里不再赘述。为了支持字符串的输入输出，在 input 和 output 语句的处理中加入类型判断，如果 input 内变量为字符串则读取字符串，output 内变量为字符串则输出字符串。对输入输出的处理，仅需在 opr 内增加两个操作即可。

2.4 字典和集合的实现方式

由于有了指针类的实现，map 和 set 的实现也可以基于二叉搜索树的字符串表示实现。在词法分析上，仍仅需更改保留字和符号的集合。

对于声明，需要在 let 后、变量名前加上 set 或 map 符号来表明这个变量是一个集合/字典类，符号表内通过 is_set、is_map 来表示这个变量是否为集合/字典变量。需要注意的是，集合和字典类由于内存占用过大，且二叉树的数组表示的复制过程十分缓慢，因此集合和字典均不支持作为函数参数或者返回值，也不支持相互赋值。在定义时，这个 set 或 map 便被 set_init 指令或 map_init 指令初始化：对于 set 来说，这个变量的指针指向一个二叉搜索树的根节点，其根节点结构为[size|1|ptr1|value|ptr2]，节点结构为[ptr1|value|ptr2]；对于 map 来说，这个变量的指针也指向一个二叉搜索树的根节点，其根节点结构为[size|0|ptr1|key|value|ptr2]，节点结构为[ptr1|key|value|ptr2]，其中，根节点的 0 和 1 用于区分这个二叉搜索树是 map 还是 set，size 代表这个树有几个节点，对于 map 和 set 来说就是有多少个元素，返回大小操作的值也是这个 size 值。

与字符串一样，map 和 set 的内存也需要自动管理。不过，由于因上文提及过的复制副本的时间开销而移除了需要复制副本的所有操作（包括参数传递、相互赋值）后，管理起来就相对容易很多，仅需要考虑在变量生命周期结束后的释放即可。而且，由于 map 和 set 结构类似，可以定义一个通用的 tree_free 指令用于释放所有占用的内存。

对于 map 和 set 的更新（插入和删除），通过 insert 和 delete 指令实现。对于 map 的 insert，需要通过 key 在二叉树上找到待插入的位置，若 key 已经存在则只更新对应的 value，如果不存在则新建一个节点并更新树。对于 map 的 delete，需要通过 key 在二叉树上找到待删除的节点，若节点有一个子树，则可以直接删除节点并将子树连接到父节点；如果有两个子树，则使用该节点的中序后继节点的键值对替换该节点的键值对，并释放原来的中序后继节点。这样的 delete 操作可以最小化更新的开销。set 的 insert 和 delete 操作大致与 map 相同，区别在于查找、替换的时候使用的是 value，且当插入的 value 存在时 set 不更新。

map 和 set 的查找操作直接对应二叉搜索树的查找过程，这里省去描述。需要注意的是，map 的查找操作分为两种：<ident> "." "find" "(" <expr> ")"与 set 的相同，都是查找某个值/键是否存在，返回的是一个真值；而<ident> "[" <expr> "]"则是返回该键值对应的 value 值，在 set 中并没有对应操作。

对于遍历操作，map 和 set 实现的是中序遍历，这样可以保证遍历结果是有序的。对于 map 的遍历，先创建一个 2*size 大小的数组，在中序遍历时往这个数组内加入键值对，之后返回这个数组的头指针。对于 set 的遍历，先创建一个 size 大小的数组，在中序遍历时往这个数组内加入值，之后返回这个数组的头指针。需要注意的是，如果是 map 或 set 为空（size 为 0），遍历操作返回空指针 0。也因此，如果不验证遍历结果或者 size，直接解析遍历操作

的返回值，可能会出现解引用空指针的错误。

2.5 虚拟机指令集

指令名	参数 l	参数 a	指令含义
lit	—	常量值	将 a 压入栈顶
Opr	参数个数	0	函数调用结束后返回
	—	1	栈顶元素取反
	—	2	次栈顶项加上栈顶项，退两个栈元素，相加值进栈
	—	3	次栈顶项减去栈顶项
	—	4	次栈顶项乘以栈顶项
	—	5	次栈顶项除以栈顶项
	—	6	判断次栈顶项与栈顶项是否相等，退两个栈元素，真值进栈
	—	7	判断次栈顶项与栈顶项是否不等
	—	8	判断次栈顶项是否小于栈顶项
	—	9	判断次栈顶项是否大于等于栈顶项
	—	10	判断次栈顶项是否大于栈顶项
	—	11	判断次栈顶项是否小于等于栈顶项
	—	12	栈顶值出栈并输出
	—	13	栈顶字符串出栈并输出
	—	14	输出换行符
	—	15	读入一个整型输入置于栈顶
	—	16	读取字符串输入，将其堆地址压入栈顶
	—	17	执行安全内存加法
	—	18	将栈顶整数转换为字符串
	—	19	交换栈顶两个元素
	—	20	弹出栈顶元素
lod	层次差	相对地址	取相对当前过程的数据基地址为 a 的内存的值到栈顶

sto	层次差	相对地址	栈顶的值存到相对当前过程的数据基地址为 a 的内存
cal	层次差	过程代码起始地址	调用子过程
ini	—	分配单元数	在数据栈中为函数开辟 a 个单元的数据区
jmp	—	目标指令地址	直接跳转
jpc	—	目标指令地址	条件跳转
addr	层次差	相对地址	取当前过程的数据基地址为 a 的地址到栈顶
loda	—	—	将栈顶的地址所指向的内存值取出，替换栈顶的地址
stoa	—	—	栈顶的值存入次栈顶的地址所指向的内存位置，退两个栈元素
alloc	层次差	相对地址	为当前过程的数据基地址为 a 的指针分配栈顶值大小的空间，栈顶值出栈
fre	层次差	相对地址	释放为当前过程的数据基地址为 a 的指针分配的空间
cpystr	—	—	将栈顶字符串复制到堆中，压入新分配的地址
litstr	—	字面量地址	将字符串的值取到栈顶
refinc	—	—	增加栈顶字符串的引用计数
refdec	—	—	减少栈顶字符串的引用计数，若引用计数为 0 则释放
concat	—	—	栈顶两字符串连接，退两个栈元素，结果入栈
repeat	—	—	将次栈顶字符串重复栈顶整数次，退两个栈元素，结果入栈
set_init	—	—	在堆中初始化一个空集合，将其地址入栈

map_init	—	—	在堆中初始化一个空字典，将其地址入栈
tree_free	—	—	释放栈顶地址指向的集合或字典占用的所有内存
insert	层次差	相对地址	在当前过程的数据基地址为 a 的集合或字典内插入一个元素。如果是集合，栈顶是值；如果是字典，栈顶是值，次栈顶是键
delete	层次差	相对地址	从当前过程的数据基地址为 a 的集合或字典内删除一个元素。如果是集合，栈顶是值；如果是字典，栈顶是键
find	层次差	相对地址	在当前过程的数据基地址为 a 的集合或字典内查找栈顶值或键，将结果压栈。如果是字典，次栈顶用于区分查找后压入真值（0）还是键对应的值（1）
traverse	层次差	相对地址	中序遍历当前过程的数据基地址为 a 的集合或字典，将所有元素（集合是值，字典是键值对）存入一个新的数组，并将结果数组的起始地址入栈
size	层次差	相对地址	将当前过程的数据基地址为 a 的集合或字典的大小压栈

2.6 错误对应编号

1：声明错误

2：分号缺失

3：标识符未找到

4：赋值语句左部标识符类型错误

5：调用了非函数标识符

6：语句起始错误

7：缺失（

8：缺失）

9：未知标识符错误

10：错误的 program 引用

11：因子的开始符号错误

12：因子后续符号错误

13：数字位数太多

14：不等符号错误

15：未识别的符号

16：程序不以 program 开头

17：缺失{

18：函数定义后缺失

19：缺失 main 函数

20：缺失}

21：缺失函数命名

22：函数调用时参数个数不匹配

23：字符串过长

24：字符串字面量过多

25：正负号用于整数之外的类型

26：加法类型不匹配

27：减法类型不匹配

28：乘法类型不匹配

29：除法类型不匹配

30：[]内必须为整型

31：赋值类型不匹配

32：函数参数内不能有 map 或 set 类

33：标识符的未知行为

34：不应出现的 return 语句

35：函数无返回值

36：解引用对象错误

37：引用对象错误

38：预期赋值语句缺失=

39：缺失]

40：为非指针分配空间

41：分配空间语句错误

42：为非指针回收空间

43：关系运算错误

44：为集合或字典变量赋值

45：非字典或集合不能使用.运算

46：字典或集合运算类型不匹配

47：查找、遍历和返回大小不能独立使用

48：未知的字典或集合运算

49：更新操作不能作为因子

50：插入语法错误

3.执行方式

与 PL0 编译器相同，编译运行后，在命令行中输入文件名、是否输出中间代码、是否输出符号表，之后就可以看到输出结果。输出的文件包括所有输出 foutput.txt、中间代码 fcode.txt、符号表 ftable.txt、L25 输入输出 fresult.txt。

4.测试结果

MergeSort.txt：

```
Start pl0
Enter array Size (e.g., 5-10 for small test):
? 6
Enter element 0:
? 1
Enter element 1:
? 5
Enter element 2:
? 2
Enter element 3:
? 4
Enter element 4:
? 3
Enter element 5:
? 7
Original array:
1
5
2
4
3
7

Sorted array:
1
2
3
4
5
7

End pl0
```

GCD.txt：

```
Start pl0
? 35
? 49
7
End pl0
```

StringMatch.txt：

```
Start pl0
Enter the main text string:
(str)? abcd
Enter the pattern string to find:
(str)? bc
Searching for 'bc' in 'abcd':
modified: bcbcbc
Pattern found at index: 1
Enter the main text string:
(str)? aaaaaaaa
Enter the pattern string to find:
(str)? aab
Searching for 'aab' in 'aaaaaaaa':
modified: aabaabaab
Pattern not found in the text.
Enter the main text string:
(str)? aaaaaa
Enter the pattern string to find:
(str)? aaa
Searching for 'aaa' in 'aaaaaa':
modified: aaaaaaaaa
Pattern found at index: 0
```

pointerTest.txt：

```
Start pl0
123
502
finish test 1
200
100
100
300
finish test 2
789
987
finish test 3
should have runtime error in alloc 0
Error at line 91: Alloc size < 0
```

SetTest.txt：

```
Start pl0
--- SET TEST START ---

--- Phase 1: Initial State & Basic Inserts ---
Initial find(10): 0
Initial size: 0
Traverse of empty set: Returned null pointer (correct).

Inserting: 50, 20, 80, 10, 30, 70, 90
Size after 7 inserts: 7
Elements after 7 inserts (should be sorted):
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 50
Element 4: 70
Element 5: 80
Element 6: 90

Find 20 (exists): 1
Find 100 (non-existent): 0
Find 50 (root): 1
```

```
--- Phase 2: Inserting Duplicates and Negatives ---
Inserting duplicates: 20, 50
Size after duplicate inserts: 7
Inserting negative numbers: -5, -15
Inserting more positives: 40, 60
Size after mixed inserts: 11
Elements after mixed inserts (should be sorted):
Element 0: -15
Element 1: -5
Element 2: 10
Element 3: 20
Element 4: 30
Element 5: 40
Element 6: 50
Element 7: 60
Element 8: 70
Element 9: 80
Element 10: 90

Find -15: 1
Find 40: 1
Find 25 (non-existent): 0
```

```
--- Phase 3: Deletion Scenarios ---
Attempting to delete non-existent: 1000
Size after deleting 1000: 11
Deleting leaf node: 10
Size after deleting 10: 10
Elements:
Find 10: 0

Deleting node with one child: 70 (child 60)
Size after deleting 70: 9
Elements:
Find 70: 0

Deleting node with two children: 20 (children -5, 30)
Size after deleting 20: 8
Elements:
Element 0: -15
Element 1: -5
Element 2: 30
Element 3: 40
Element 4: 50
Element 5: 60
Element 6: 80
Element 7: 90
Find 20: 0

Deleting root node: 50 (should be replaced by its successor)
Size after deleting 50: 7
Elements:
Element 0: -15
Element 1: -5
Element 2: 30
Element 3: 40
Element 4: 60
Element 5: 80
Element 6: 90
Find 50: 0
```

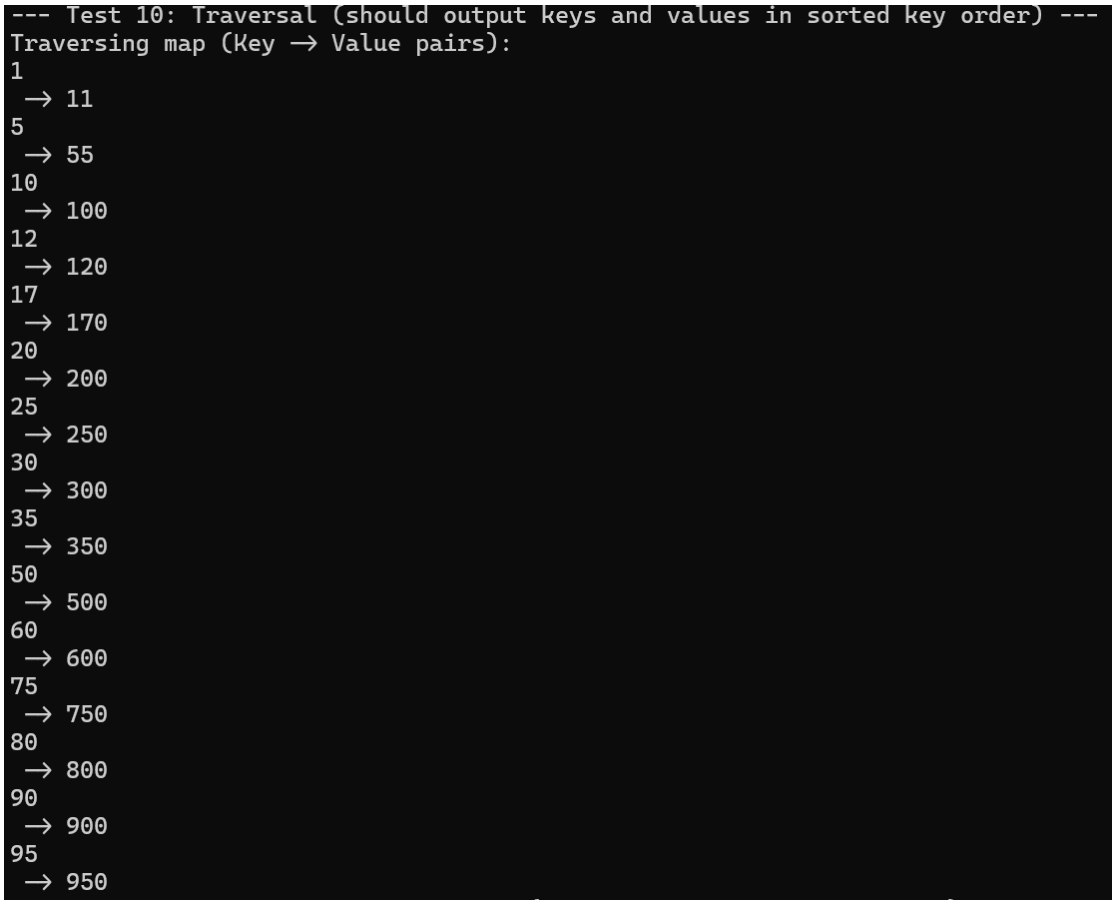
```
--- Phase 4: Emptying and Re-populating ---
Deleting remaining elements to empty the set...
Size after emptying: 0
Traverse of empty set after deletions: Returned null pointer (correct).
Find 1 in empty set: 0

Re-populating with new values: 10, 20, 30, 5, 15, 25, 35
Size after re-populating: 7
Elements after re-populating:
Element 0: 5
Element 1: 10
Element 2: 15
Element 3: 20
Element 4: 25
Element 5: 30
Element 6: 35

Find 20: 1
Find 35: 1
```

```
--- Phase 5: Mixed Operations & Final Checks ---
Deleting 15, inserting 12, deleting 30, inserting 28, 40
Final size: 8
Final elements:
Element 0: 5
Element 1: 10
Element 2: 12
Element 3: 20
Element 4: 25
Element 5: 28
Element 6: 35
Element 7: 40

Find 15: 0
Find 12: 1
Find 28: 1
Find 40: 1

--- SET TEST END ---
End pl0
```

Maptest.txt：

```
Start pl0
Starting Map Test Program
--- Test 1: Basic Insertions and Size ---
Map size after 4 insertions: 4
--- Test 2: Retrieval ---
myMap[10]: 100
myMap[20]: 200
myMap[5]: 50
myMap[15]: 150
--- Test 3: Update existing key ---
myMap[10] after update: 1000
--- Test 4: Find (existence check) ---
myMap.find(5): 1
myMap.find(10): 1
myMap.find(25): 0
--- Test 5: Value lookup for non-existent key ---
myMap[25]: 0
--- Test 6: Deletion (various cases) ---
Initial map for deletion tests size: 4
Deleting key 10 (node with two children):
Map size after deleting 10: 3
myMap.find(10): 0
myMap[10]: 0
Deleting key 5 (leaf node):
Map size after deleting 5: 2
myMap.find(5): 0
Deleting key 20 (node with one child, 15):
Map size after deleting 20: 1
myMap.find(20): 0
Deleting key 15 (the last node):
Map size after deleting 15: 0
myMap.find(15): 0
```

```
--- Test 7: Operations on an empty map ---
Empty map size: 0
Empty map.find(1): 0
Empty map[1]: 0
Empty map size after deleting non-existent key: 0
--- Test 8: Re-insert after full deletion ---
myMap[100]: 1
Map size: 1
Map size after deleting 100: 0
--- Test 9: Complex insertions (to test BST structure) ---
Map size after complex insertions: 15
Values after complex insertions:
myMap[50]: 500
myMap[25]: 250
myMap[75]: 750
myMap[10]: 100
myMap[35]: 350
myMap[60]: 600
myMap[90]: 900
myMap[5]: 55
myMap[80]: 800
myMap[95]: 950
myMap[1]: 11
myMap[12]: 120
myMap[17]: 170
myMap[20]: 200
myMap[30]: 300
```


--- Test 10: Traversal (should output keys and values in sorted key order) ---
Traversing map (Key → Value pairs):
1
→ 11
5
→ 55
10
→ 100
12
→ 120
17
→ 170
20
→ 200
25
→ 250
30
→ 300
35
→ 350
50
→ 500
60
→ 600
75
→ 750
80
→ 800
90
→ 900
95
→ 950

```
--- Test 11: More Deletion Scenarios (focused on tree restructuring) ---
Deleting key 1 (leaf):
Size: 14
Deleting key 12 (leaf):
Size: 13
Deleting key 10 (node with one child, 17):
Size: 12
myMap.find(10): 0
myMap[17]: 170
Deleting key 25 (node with two children, 5 and 35):
Size: 11
myMap.find(25): 0
myMap[30]: 300
Deleting key 50 (current root, has two children):
Size: 10
myMap.find(50): 0
--- Test 12: Negative Keys and Zero Key/Value ---
NegMap size: 5
NegMap values:
negMap[-10]: -100
negMap[-5]: -50
negMap[-15]: -150
negMap[0]: 0
negMap[-7]: 77
Traversal with negative/zero keys (sorted order):
-15
 → -150
-10
 → -100
-7
 → 77
-5
 → -50
0
 → 0
Deleting negative key (-10):
NegMap size: 4
negMap.find(-10): 0
```

```
--- Test 13: Chained operations and expressions with map values ---
Chained map size: 3
chainedMap[4]: 30
chainedMap[5]: 30
Chained map size after additions: 5
Chained map size after deleting 2: 4
--- All map tests finished ---
End pl0
```