计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200100133

程序调试中的树形结构演变可视化模型

苏 庆 黎智洲 刘添添 吴伟民 黄剑锋 李小妹
广东工业大学计算机学院 广州 510006
(suqing@gdut.edu.cn)

摘 要 树形结构作为一种非线性数据结构,在程序执行过程中的演变过程较为抽象,尤其是在对其进行加工型操作时,容易发生内存泄漏。针对编程初学者难以掌握树形结构的逻辑演变过程,以及在程序中发生内存泄漏错误时调试较为困难的问题,文中提出了一种对程序调试过程中树形结构的演变过程进行可视化呈现的处理模型TEVM(Tree Evaluation Visualization Model)。针对单个可视化程序在调试步骤前和调试步骤后的两个树形结构,设计了一种将树形结构转换为线性表示的结构对比算法,得出了它们的包括泄漏树在内的结构差异;同时设计了一种树形结构布局方法,并计算它们的布局差异。根据结构差异和布局差异生成可视化演变序列,调用绘图引擎对该序列进行解析和执行,从而完成对树形结构及其演变过程的动态、平滑和直观的可视化呈现,帮助编程初学者快速理解树形结构相关程序的执行过程,提升调试效率。将TEVM模型集成于一个面向编程实训教学的集成开发环境原型Web AnyviewC中,取得了较好的应用效果。

关键词:程序调试过程;数据结构可视化;树形结构;泄漏树;树形结构演变可视化模型
中图法分类号 TP311

Tree Structure Evaluation Visualization Model for Program Debugging

SU Qing,LI Zhi-zhou,LIU Tian-tian,WU Wei-min,HUANG Jian-feng and LI Xiao-mei
School of Computers,Guangdong University of Technology,Guangzhou 510006,China

Abstract Tree structure is a nonlinear data structure,whose evolution process is abstract in the program execution. Memory leak happens while applying modification-like operation on it. It is an open challenge for programming beginner to control the evolution of tree structure in the program debugging procedure,especially to debug the error when memory leakage happens. To address this issue,this paper proposes a tree evaluation visualization model(TEVM) to visualize the evolution procedure of tree structure. Two tree structures are obtained after executing a visual debugging step. This paper designs a structure comparison algorithm to obtain their structural difference including leaked tree by transforming tree structure into linear representation. It also designs a tree layout method and computes their positional difference. A visual evaluation sequence is generated with these structural difference and positional difference. At last,it applys the drawing engine to interpret and execute actions of this sequence to visualize the tree structure evaluation dynamically,smoothly and intuitively. The visualization of tree structure helps programming beginner to understand the execution of program relating to tree structure and improves the efficiency of program debugging. TEVM model is applied in the Web AnyviewC,which is a prototype of the integrated development environment for programming training,and gains an excellent application effect.

Keywords Program debugging process,Data structure visualization,Tree structure,Leaked tree,Tree struct evaluation visualization model

1 引言

程序设计可视化客观上可起到降低编程学习门槛的作用,是在编程教育领域中广泛运用的技术手段之一。将程序设计可视化应用于编程教育领域时,一种可行的策略是基于各种程序调试功能,将数据关系及其变换过程配合程序调试步骤做动态可视化,从而帮助初学者理解抽象的数据关系及其控制逻辑。

到稿日期:2020-01-21 返修日期:2020-07-29 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家自然科学基金(618002072);广东省自然科学基金(2018A030313389);广东省高等教育教学改革项目(SJJG20191216)
This work was supported by the National Natural Science Foundation of China(618002072),Natural Science Foundation of Guangdong Province,China(2018A030313389) and Higher Education Teaching Reform Project Foundation of Guangdong Province,China(SJJG20191216).
通信作者:刘添添(ttliu@gdut.edu.cn)

因此，数据结构可视化是程序设计可视化的主要内容。在线性数据结构和非线性数据结构这两大类数据结构中，我们已经提出了一种线性结构的识别及可视化布局算法[1]，并阐述了如何在程序调试中对单链结构动态可视化方案进行设计、实施与应用。而树是一类重要的非线性数据结构，其在程序调试过程中的可视化方案则是本文的主要研究内容。

以二叉树[2]为例，在一般场景下可视化一棵符合定义的二叉树可以有多种方式，且只需遵循特定的布局方法[3]即可。然而如需达到可动态调试的要求，则需要考虑各种边缘情况检测以及容错方案。例如初学者在编写对二叉树的操作代码时存在的逻辑上的错误，又或者如果初学者处于某段修改二叉树的代码执行过程中但又尚未执行完毕的阶段，就可能会编写出现一种"类二叉树"结构，即其形态与二叉树相似，但某些结点却可能存在两个或两个以上的双亲结点。如图 1 所示，结点 3 存在两个双亲结点 1 和 4。这种类树结构比符合定义的二叉树结构更复杂，仅借助堆栈数据进行数据关系的理解存在较大的困难，因此在程序调试过程中，对树形结构（包括类树结构）进行直观和动态的可视化展示，将有助于编程初学者正确理解当前树形结构的逻辑组织。

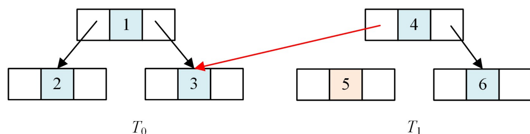


图 1　类树结构示例

Fig. 1　Example of tree-like structure

有学者对程序设计中的树形结构可视化方法进行了研究和应用。Wang 等[3]将 2D 树形结构布局方法分为连接方法和封闭方法两大类，为树形布局可视化方案的研究提供了理论支持。Bacher 等[4]提出了一种将源代码目录结构可视化为树形目录的方案，实现了对程序代码进行可视化图形映射，但此方案侧重工程应用领域需求，与编程教育领域特别注重的易用性尚有区别。Yuan[5]设计了一种树型编译系统，将用户程序代码以流程图的方式进行可视化展示，可表达编译与可视化的关系。上述可视化方案大都侧重数据关系的静态展示，未能建立程序调试和数据结构可视化呈现之间的联系。

此外，还有学者对数据可视化中的演变问题进行了研究。Heer 等[6]发现动画呈现方式可以改善关于统计数据图形变化的图形感知，有助于增进用户的理解和提高其参与度。Plaisant 等[7]开发了一个层次树形结构的可视化浏览工具 SpaceTree，并将可视化演变分为 3 个阶段：元素折叠、剩余元素移动和元素扩展。Maruthappan 等[8]的实验揭示了在数据结构可视化视图变化时，平滑的动画呈现形式对用户具有积极影响，然而他们却并未探讨可视化视图演变的具体实现技术。Guilmaine 等[9]针对径向布局的树形结构演变过程，对几种用于追踪径向树结点变化的可视化演变方法进行了研究。整体而言，上述工作并未针对树形结构演变的可视化呈现问题建立起通用的解决模型。

综合上述研究现状，目前尚未发现较为完善的适用于编程教育领域中面向程序调试的树形数据结构演变过程的可视化方法。而本文的贡献主要在于提出了一种面向程序调试中树形结构演变过程的可视化模型，解决了在编程教育领域的程序调试中，由于树形结构的演变过程抽象和复杂造成的程序设计学习者难以理解和控制数据结构变化过程的问题。最后将该模型实际应用于可视化编程实训平台原型 AnyviewC[10]。

由于篇幅所限，本文仅以二叉树为基础讨论树形结构演变的可视化模型。该模型可以方便地扩充至一般的树形结构。

2　相关定义

为便于阐述，本文将符合传统定义的树以及由树演化而成的、不符合树定义的但与树高度相似的类树结构都统称为树形结构，在不引起混淆以及无需区分的情况下，将它们都简称为树。

设有树 $T=\langle V,R\rangle$，V 为结点集，R 为各结点中所有指针域的集合。对于某结点 $v\in V$，令结点集 $P(v)$ 为 v 的双亲结点集，$C(v)$ 为 v 的直接孩子结点集，指针集 $U(v)$ 为 S 中所有指向 v 的指针集，$Q(v)$ 为所有 $P(v)$ 结点中指向 v 的指针集。

定义 1（主双亲和次双亲（Major Parent & Minor Parent））若 v 的入度 $D^+(v)=1$，则 $q_1\in Q(v)$ 为 v 的主双亲结点；若 $D^+(v)>1$，则 $Q(v)-q_1$ 为 v 的次双亲结点集合。

定义 2（单亲结点（Single-parent Node）和多亲结点（Mulit-parent Node））　若 $D^+(v)=1$，则 v 为单亲结点；若 $D^+(v)>1$，则 v 为多亲结点。

如果结点 v 为多亲结点，则必须根据某种规则为 v 从其双亲结点集合 $P(v)$ 中选定一个主双亲 w，此时 $P(v)-w$ 为 v 的次双亲结点集。

定义 3（泄漏树（Leaked Tree，LT））　设有树 T，若 $\forall v\in T$ 都不是外部指针指向的结点或多亲结点，则称 T 为泄漏树。

鉴于泄漏树的重要性，可在可视化视图中划分一个独立区域进行可视化呈现，称此区域为泄漏区。

定义 4（可视化调试步骤（Visual Debugging Step，VDS））　在执行一个单步程序调试步骤时，如果某树 T 的一个或多个结点 v 至少发生以下任意一种变化：

（1）结点 v 中的数据域被修改；

（2）结点 v 中的指针域被修改；

（3）指向结点 v 外部的指针域被修改。

则该步骤称为一个可视化调试步骤，简称 VDS。

一次程序调试过程可包含多个 VDS。这些 VDS 的顺序执行形成了树形结构的连续演变过程。因此只需研究针对单个 VDS 的树形结构演变可视化，即可将其扩展至整个调试过程。

3　树形结构演变可视化模型

在一次执行 VDS 前后，编译器分别产生两批树形结构数据。在对其做必要的数据清洗和转换[11]后，分别将其记为 T_0

和T_1。由于它们在逻辑结构上可能存在差异，进而它们在视图布局上也会存在差异。当这些差异较大时，在可视化展示中，如果直接擦除T_0然后立即呈现T_1，就会在可视化效果上表现为从T_0到T_1的突变，令用户瞬间失去焦点，不利于用户对树形结构演变过程的理解。因此，非常有必要动态且平滑地呈现从T_0演化到T_1的过程，便于用户理解 VDS 执行对数据关系的确切影响，进而更准确地掌控程序执行流程。

为动态展示程序调试过程中树形结构的演变过程，本文提出一种树形结构演变可视化模型，其模型结构如图 2 所示。

TEVM 以两批树形结构数据(T_0 & T_1)作为输入，在可视化转换阶段，首先对T_0和T_1做结构对比和布局对比，分别收集T_0和T_1在结构上(包括结点数目、指针域和数据域)和布局上(即结点位置)的差异信息。然后将这些信息映射成为一个树形结构可视化演变序列。最后，调用绘图引擎[12]对该序列进行解析和执行，绘制呈现树形结构演变过程的可视化视图。

需要特别指出一种编程初学者易犯且难以发现的错误是，在对树进行加工型操作时，由于代码逻辑错误，部分树结构丧失了所有的指针指向，不能再被访问，即出现了泄漏树结构。在 TEVM 模型的结构对比环节可以发现泄漏树结构，然后将其作为部分结构差异信息，用于产生可视化演变序列，最后再做专门的可视化呈现。本文将在第 4 节对泄漏树的分析和处理作详细讨论。

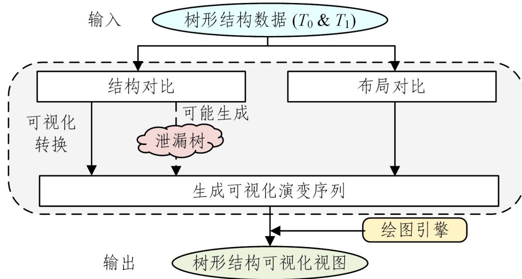


图 2 树形结构演变的可视化模型

Fig. 2 Framework of tree structure evolution visualization mode

下面将重点介绍 TEVM 模型的 3 个环节：结构对比、布局对比和生成可视化演变序列。

3.1 树形结构对比

树形结构对比的目的在于求解T_0与T_1间的结构差异集合(Structural Difference, SD)。SD是一个包含描述两个树形结构差异信息的集合，由多种结构差异子集构成，具体公式如下：

$$SD=\{SD_{ext}, SD_{lt}, SD_{cpf}, SD_{df}, SD_{epf}\} \tag{1}$$

表 1 列出了各个结构差异子集的具体含义。

表 1 结构差异子集的释义

Table 1 Introduction of structural difference subsets

子集合	释义
SD_{ext}	新增结点的信息
SD_{lt}	需移至泄漏区的结点信息
SD_{cpf}	结点内指针域变化的信息
SD_{df}	结点内数据域变化的信息
SD_{epf}	结点的外部指针域变化的信息

3.1.1 线性表示对比算法

树是一种递归结构，如果对其作递归遍历，算法效率较低。因此，本文提出一种非递归的线性表示对比算法，用于求解上述结构差异集合SD。

该算法的主要流程如图 3 所示，首先将树形结构转换为线性结构表示，然后分别进行线性结构间以及结点间的对比，以获取各种结构差异子集，最终将其构建为结构差异集合。

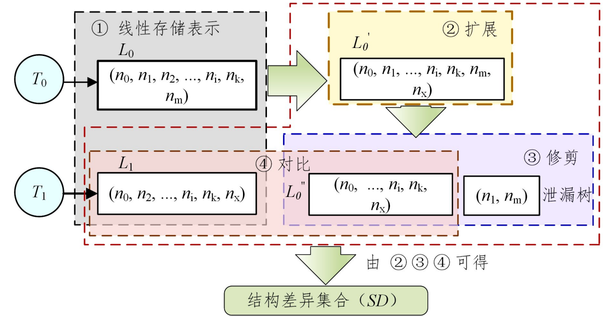


图 3 线性表示对比算法流程图

Fig. 3 Workflow of linear representation comparison algorithm

Step 1 树的线性表存储表示。对T_0和T_1分别作先序遍历，转化为线性表L_0和L_1存储表示。在L_0和L_1中只保存对结点的引用，而不是复制整个结点，以实现较低的性能开销。

Step 2 扩展。查找存在于L_1但不存在于L_0的结点，得到结点集N_{ext}。此时N_{ext}包含所有从T_0演化至T_1后的新增结点，即结点差异集合SD_{ext}。另外，将N_{ext}添加至L_0，得到L_0'。

Step 3 修剪。在L_0'中查找存在于L_0但不存在于L_1的结点，得到结点集N_{lt}。T_0演化至T_1后，$\forall n \in N_{lt}$都不存在于L_1，因为它们无任何指针指向所以无法再被访问。因此N_{lt}即泄漏结点集SD_{lt}，并且可以构成一棵或多棵泄漏树。另外，从L_0''中删除N_{lt}，得到L_0''。

Step 4 结点间对比。分别对比L_0''与L_1对应结点内的指针域(包括左孩子指针域和右孩子指针域)、数据域和外部指针域，依次得到结构差异子集SD_{cpf}，SD_{df}和SD_{epf}。

算法的执行结果为一个结构差异集合SD，其包含上述步骤得到的各差异子集。

3.1.2 时间和空间复杂度分析

在算法实施过程中，需要对树中的每个结点都设置唯一的标识 id(可以是 hash 值、内存地址或以某种规则人工指定的值)。在进行两树或线性结构对比的过程中，由于只需对比 id 相同的结点，避免了耗时的递归遍历比较。一般情况下，两个线性结构对比算法的时间复杂度应为 $O(n^2)$，因此每个结点都需再次遍历另一线性结构，以查找与其对应的结点。为提升算法效率，在实现时创建一个以结点 id 为 key 的哈希表，以将查找对应结点的复杂度从 $O(n)$降为 $O(1)$。

由此，本算法的时间复杂度为 $O(n)$，优于时间复杂度为 $O(n^3)$的传统树形结构对比算法[13]。

本算法需要使用两个线性表和一个哈希表作为辅助空间(规模均为 n)，因此其空间复杂度为 $O(n)$。

3.2　树形结构布局对比

通过对比T_0和T_1的布局参数，可知它们在布局上的差异。其中T_0的布局参数在VDS执行前已知，而T_1则需要在VDS执行后对其进行逻辑布局，获得其布局参数。两者的布局参数的对比结果为位置差异集合（Positional Difference，PD）。具体而言，PD是记录同一个结点分别在T_0与T_1中的坐标位置差异信息的集合。因此，首先需要研究树形结构布局方法，对T_1进行布局（当前VDS的T_1将会是下一个VDS的T_0），然后再对比T_0和T_1的布局参数，即可计算得到PD。

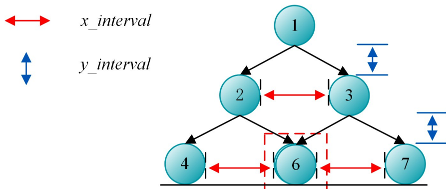
3.2.1　树形结构布局方法

根据树形结构中各结点和类树间的关系，计算各结点和类树在空间中的坐标位置从而进行层次树形布局[14]。树形结构布局方法分为两个层次：第一个层次为树内结点间布局，即以结点为最小粒度，做树内结点间水平和垂直方向上的布局，以确定树内各结点的相对位置。第二个层次为树间布局，即以树形结构为最小粒度，来确定各树形结构之间的相对位置，其中树形结构可以为符合树定义的常规二叉树结构或类树结构。

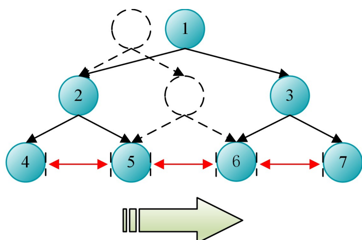
（1）结点间布局

结点间布局用于解决结点与其主双亲结点和兄弟结点间的位置关系。令左右孩子结点以一定水平间距和垂直间距分布于主双亲结点的两侧和下方，同时保证子树间不发生结点重叠和边交叉，以及保持主双亲结点于孩子结点间的对称性。具体可以分为以下两个步骤。

Step 1　从树的根结点开始，以主双亲结点的水平中点为中心，$x_interval$为水平间距，将左右孩子结点分别对称排布于主双亲结点中心的左右两侧。同时，以主双亲为始点，$y_interval$为垂直间距，置左右孩子结点于主双亲结点的下方。递归进行该步骤直至所有结点均被处理。该步骤被称为下降排布。图4(a)给出了一个示意图，此时水平方向上相邻的非兄弟结点有可能发生重叠。



(a)向下排布过程中出现结点重叠



(b)回推调整——右向水平偏移解决重叠

图4　向下排布与回推调整

Fig. 4　Downward layout and backward adjustment

Step 2　从树的叶子结点开始，从下往上按层次遍历，检查并确保水平方向上相邻非兄弟结点间的水平间距不小于$x_interval$。此时由于主双亲失去孩子结点的对称性，故需要进一步调整主双亲结点的位置。对树的每一层应用该步骤，直至根结点。该步骤被称为回推调整。图4(b)给出了调整后的布局状态。

（2）树间布局

当完成所有树内的结点间布局后，可对树集合$S=\{T_a, T_b, \cdots, T_n\}$进行树间布局，具体可以分为以下两个步骤。

Step 1　从S中识别和划分树单元(tree_unit)：对于$T_i \in S(i=1,2,\cdots,m)$，若$T_i$是常规二叉树，则$T_i$独立成为一个$tree_unit$；对于$T_i, T_j \in S$，若$T_i$与$T_j$组成一个类树结构，则由$T_i$和$T_j$所组成的类树成为一个$tree_unit$。

Step 2　从第一个$tree_unit$开始，以$tree_unit_interval$为间距，在水平方向上从左往右进行布局。如果某$tree_unit$是类树结构，则组成该$tree_unit$的所有常规二叉树或者类树都应在水平方向上相邻。

3.2.2　计算布局差异集合

依次将T_0和T_1中对应的结点的坐标进行对比，即可得到布局差异集合PD。对于$\forall n_i \in T_1(i=1,2,\cdots,m)$，有：

$$PD_i = Position_Compare(n_i, n_{cor}) \tag{2}$$

其中，$n_{cor} \in T_0$是与n_i的id相同的结点。$Position_Compare$是结点的坐标对比函数，其输出为两个结点的坐标位置之差。

由此可得位置差异集合PD为：

$$PD = \{PD_0, PD_1, PD_2, \cdots, PD_m\} \tag{3}$$

3.3　生成可视化演变序列

将通过结构对比得到的SD和布局对比得到的PD统称为差异集合(Difference，D)，即$D=\{SD, PD\}$。为最终进行树形结构可视化视图的绘制，需要将D映射为一个可由绘图引擎解析并执行的可视化演变序列(Transition Sequence，TS)。

TS的基本单位是可视化演变动作。所有可视化演变动作的种类(见表2)构成一个动作集合$Action$。建立一个函数$f: D \rightarrow Action$，表示D与$Action$之间的映射关系。表2定义了具体的差异类型与动作种类之间的映射规则。

表2　映射规则及可视化演变动作释义

Table 2　Definitions of mapping of f and visualization actions

D	$Action$	可视化演变动作释义
$d \in SD_{ext}$	Add	新增一个结点
$d \in SD_{lt}$	$MoveToLA$	将某个结点移动至泄漏区
$d \in SD_{cpf}$	$AlterCPF$	结点内的指针域发生变化
$d \in SD_{epf}$	$AlterEPF$	结点的外部指针域发生变化
$d \in SD_{df}$	$AlterDF$	结点的数据域发生变化
$d \in PD$	$UpdatePOS$	结点坐标发生变化

最终，TS可表示为：

$$TS = \{act_0, act_1, act_2, \cdots, act_w\}, w = |D| \tag{4}$$

其中，$act_i \in Action$。

在呈现树形结构的演变过程时，以T_0视图为初始状态，调用绘图引擎解析和执行TS中的可视化演变动作，便可产生以动画形式呈现从T_0演变至T_1的可视化效果，以及将泄漏树LT(如果有)平滑移动至泄漏区的可视化效果。定义一个

apply 函数，用于调用绘图引擎执行 *TS* 序列，将 T_0 演变为 T_1 和 *LT*。

$$apply(T_0, TS) \xrightarrow{transition} (T_1, LT) \tag{5}$$

4 泄漏树的分析和处理

产生泄漏树的主要原因在于学生程序存在逻辑上的错误，导致部分树形结构无任何指针指向而无法再被访问。

判定泄漏树的关键在于执行某个 VDS 后，检查某树形结构（或其子树结构）是否存在指针指向。因此，本文将在对指针进行修改操作时，首先收集可能的泄漏树结构，再从中检查和筛选出真正的泄漏树。

4.1 泄漏树的查找

泄漏树 *LT* 通常源于指针域发生改变的情形中。设有结点 $p, q \in T$，q 为 p 的某个孩子结点，且 q 无其他指针指向。假设此时由于学生程序逻辑错误等，将 p 指向 q 的孩子指针域置空或改为指向其他结点，则以 q 为根的子树 $T(q)$ 或者 $T(q)$ 的部分子树如果无其他指针指向，就会成为 *LT*。

以下为表述简洁，将多亲结点或外部指针指向的结点统称为奇异结点（Odd Node，ON）。$T(q)$ 中可能包含一个或多个 *ON*。这些 *ON* 可以是 q 本身或 q 的子孙结点。对这些 *ON* 的访问有两种方式：1）通过根结点 q 遍历访问；2）通过不属于 $T(q)$ 的其他双亲结点或者通过外部指针访问。因此，所有以任意结点 $o_i \in ON$ 为根的子树 $T(o_i)$ 都不会成为泄漏树。

从 $T(q)$ 中剪除所有 $T(o_i)$，即可得到泄漏树 *LT*。然而，各 $T(o_i)$ 之间可能存在包含关系，因此，需要找到所有的包含尽可能多 $T(o_i)$ 的极大子树。在剪除一棵极大子树时，它包含的所有 $T(o_i)$ 也会同时被剪除。

由此可得，从 $T(q)$ 的角度出发，泄漏树 *LT* 的查找过程可以分为以下两个步骤。

Step 1 筛选 $T(q)$ 中所有的极大子树。记 $T(q)$ 中所有奇异结点的集合为 $ON=\{n_1, n_2, \cdots, n_m\}$。设有 $n_i \in ON$ 为根的子树 $T(n_i)$，如果对于其他任意子树 $T(n_j)$，$n_j \in ON, i \neq j$，都有：

$$T(n_i) \not\subseteq T(n_j) \tag{6}$$

则 $T(n_i)$ 为一棵极大子树。

Step 2 从 $T(q)$ 中剪除所有 $T(n_i)$，即可得到泄漏树 *LT*。定义所有极大子树的根结点集为 $RN=\{r_1, r_2, \cdots, r_k\}$，则有：

$$LT = T(q) - \sum_{i=1}^{k} T(r_i), r_i \in RN \tag{7}$$

称式(7)为泄漏树生成公式。

4.2 泄漏树识别规则

由泄漏树生成式(7)，可总结出以下泄漏树的识别规则。

规则 1 *LT* 为空树的充分必要条件是 $T(q)$ 为唯一的极大子树。

证明：必要性。若 $LT=\emptyset$，则由式(7)可知，必有 $T(q)=\sum_{i=1}^{k} T(r_i)$。由极大子树的定义可得 $k=1$，即只有一棵极大子树，且根结点为 q。否则，若 $k>1$，则由于 $\sum_{i=1}^{k} T(r_i)$ 至少不包含根结点 q 而必有 $\sum_{i=1}^{k} T(r_i) \subset T(q)$，$T(q) \neq \sum_{i=1}^{k} T(r_i)$。所以，$T(q)$ 为唯一的极大子树。

充分性。若 $T(q)$ 为唯一极大子树，则 $\sum_{i=1}^{k} T(r_i) = T(q)$，因此 $LT=\emptyset$ 成立。

规则 2 *LT* 为非空树的充分必要条件是 $T(q)$ 并非极大子树。

证明：必要性。采用反证法，假设 $T(q)$ 为极大子树，则对于任意真子树 $T(n_j) \subset T(q)$，$T(n_j) \not\subseteq T(q)$ 都不可能成立，即任意真子树 $T(n_j)$ 都不可能为极大子树，因此 $T(q)$ 必定为唯一的极大子树。根据规则 1，$LT=\emptyset$，与题设 *LT* 为非空树矛盾。因此，$T(q)$ 并非极大子树。

充分性。当 $T(q)$ 并非极大子树时，q 作为 $T(q)$ 的根结点，至少有 $q \notin \sum_{i=1}^{k} T(r_i)$，因此必有 $T(q) > \sum_{i=1}^{k} T(r_i)$，由式(7)可知，必有 $LT \neq \emptyset$。

图 5 给出了一种典型的泄漏树产生的情形。在图 5 中，由于 $T(q)$ 不是极大子树，则 $T(q)$ 中必定存在泄漏树，且泄漏树正是从 $T(q)$ 中剪除所有极大子树后的剩余部分。

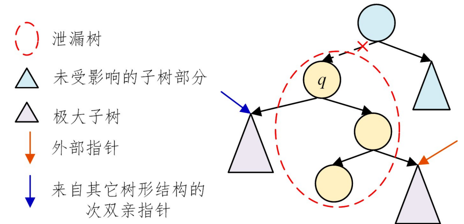


图 5 泄漏树产生的示意图

Fig. 5 Schematic interpretation of leaked tree generation

5 TEVM 在可视化编程实训平台 AnyviewC 上的应用

将 TEVM 模型应用于一个面向程序设计初学者的可视化在线编程作业平台 AnyviewC 中。该平台具有程序编辑、编译、运行和可视化调试等基本功能。AnyviewC 前端是一个 Web 应用程序，其可基于网页动画技术[16]将数据结构的演变过程进行动画呈现，直观地帮助用户理解程序执行过程中的数据关系变化。

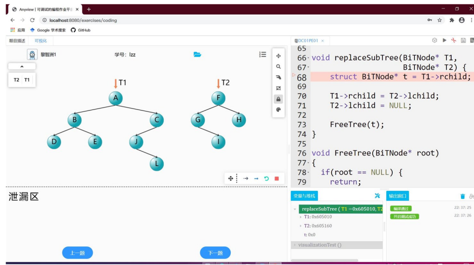
本文在对学生进行编程指导的教学实践中，采集了大量学生的树型结构训练程序代码进行分析，发现学生的理解难点集中于类树结构的出现和控制，且程序中隐藏最深且除错困难的情形在于出现了内存泄漏树结构。

因此，为展示 TEVM 在可视化类树结构和泄漏树方面的效果，以及说明树形结构可视化对于降低学生编程调试难度的意义，本文以一个关于树形结构处理的学生程序片段 *replaceSubTree*(*T1*, *T2*) 函数为例，分析其在 AnyviewC 平台上的调试过程。*replaceSubTree*(*T1*, *T2*) 函数的功能是将 *T1* 的右子树替换为 *T2* 的左子树，此功能涉及树的最重要的两类加工型操作：插入子树和删除子树。

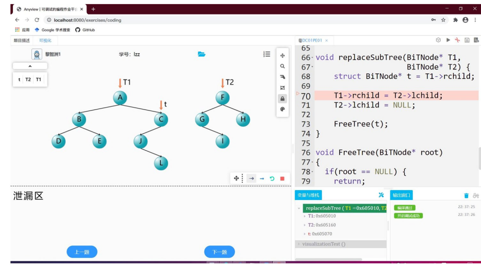
本实验将对 AnyviewC 平台与主流的集成开发环境 VisualStudio 2019（以下简称 VS2019）在程序调试过程中进行数据关系呈现方式上的对比，以验证树形结构可视化在调试中可起到帮助编程学习者快速理解数据关系、提升调试效率的作用。

5.1　类树结构可视化实验

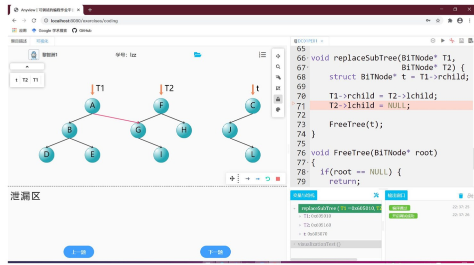
图 6(a)给出了某学生对 *replaceSubTree* 函数的实现代码。此时 *T*1 和 *T*2 已经在前面的代码中建立,并且由 AnyviewC 进行可视化呈现。当前程序第 68 行高亮,表示该行即将执行但尚未执行。当第 68 行代码执行完毕,一个指针 *t* 被定义并指向 *T*1 的右子树,可视化效果如图 6(b)所示。



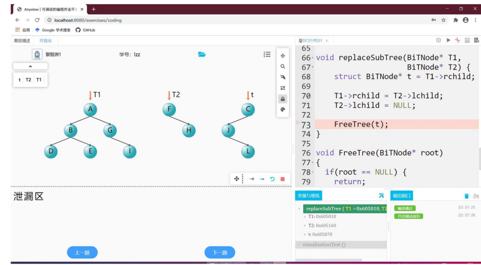
(a)树形结构可视化效果



(b)外部指针可视化效果



(c)类树结构可视化效果



(d)类树结构拆解可视化效果

图 6　树形结构演变过程的可视化效果

Fig. 6　Visualization of tree structure evaluation procedures

第 70 行“*T*1－＞*rchild*＝*T*2－＞*lchild*”是最关键的子树替换代码,用于将 *T*2 的左子树插入至 *T*1 的右子树位置。该行执行后的效果见图 6(c)。此时 *T*2 的左子树(*G*((),*I*))成为 *T*1 的右子树;而原 *T*1 的右子树(*C*(*J*,()))则成为一棵由指针 *t* 指向的新树,以动画形式被平滑移动至可视化区右侧。

图 6(c)中出现了一棵由 *T*1 和 *T*2 构成的类树结构,其中结点 *G* 存在 2 个双亲结点:*A* 为次双亲,*F* 为主双亲。该类树结构在 AnyviewC 平台中得到了直观的展示,便于编程学习者快速理解其逻辑结构。而图 7 给出了使用 VS2019 调试同一个 *replaceSubTree* 函数的情形。VS2019 在其变量窗口中采用表格形式展示各变量值,是一种细节详尽但欠缺直观性的数据关系呈现方式,编程初学者较难从中快速把握当前树形结构的演变情况。

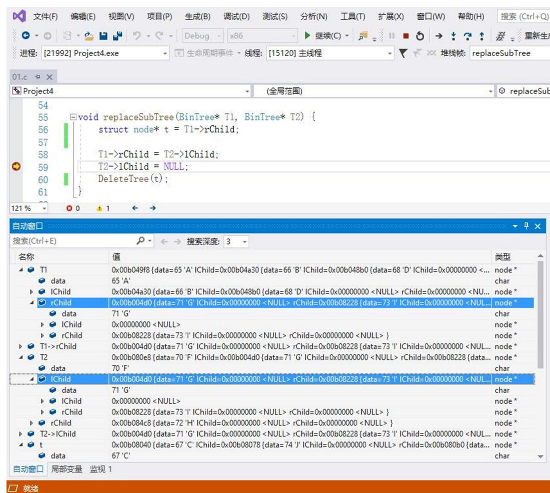


图 7　VS2019 中的表格式变量呈现

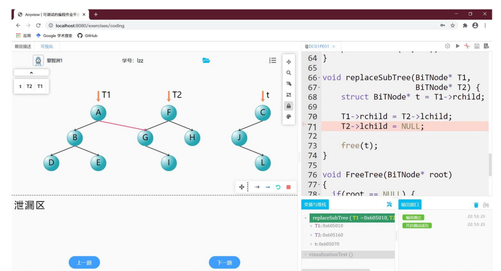
Fig. 7　Tabular representation of variables in VS2019

第 71 行代码“*T*2－＞*lchild*＝NULL;”则是将 *T*2 的左子树置空,即实现了对 *T*2 的左子树的删除。此时 *T*1 和 *T*2 重新成为了两棵独立的树,其可视化效果如图 6(d)所示。

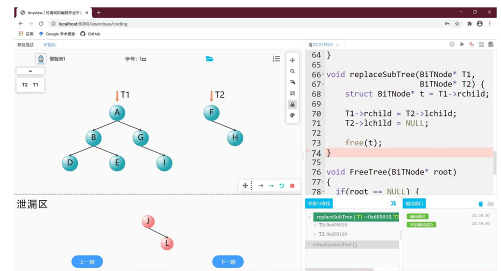
由该实验可见,树形结构可视化可以直观反映树形结构的演变过程,能够帮助编程初学者快速掌握数据关系的变化。

5.2　泄漏树可视化实验

由前述可知,在对树进行加工型操作时,一个初学者易犯且较难除错的错误是产生了泄漏树,即内存泄漏。而树形结构可视化可以有效帮助学生发现和改正上述错误。



(a)一种错误的 *replaceSubTree* 函数实现



(b)泄漏树的可视化效果

图 8　泄漏树演变过程的可视化效果

Fig. 8　Visualization of leaked tree structure evaluation procedure

如图 8(a)给出了一个错误版本的 *replaceSubTree* 函数实现代码，其中第 73 行“*free*(*t*);”实质上只释放了树 *t* 的根结点 *C* 所占内存，而其子树(*J*((),*L*))所占内存部分会被泄漏。这样会出现“丢失”部分子树的情况。本课题组教师在指导学生进行程序除错时经常发现：学生误认为如果释放了树的根结点所占内存，那么其子树所占内存也会一并被释放。

在 AnyviewC 中执行完第 73 行后，虽然结点 *C* 所占内存已被释放，但其子树((*J*(),*L*))因为无任何指针指向所以成为泄漏树，因此其将会以动画形式移动至可视化面板的泄漏区，如图 8(b)所示。当前 AnyviewC 的泄漏区可视化效果可以清晰地提示学生出现了内存泄漏的树结构，帮助其发现极易忽略的程序错误，可明显降低调试难度。

对比在 VS2019 中执行该代码的效果(见图 9)可以发现，VS2019 在执行“free(*t*);”后，并无对内存泄漏的特别提示。对于熟练的程序员而言，其可以从自动窗口面板中的灰色变量部分观察到从“*t*－＞*lchild*”开始发生了内存泄漏，但对于编程初学者，则难以确切获知泄漏了哪一部分子树。

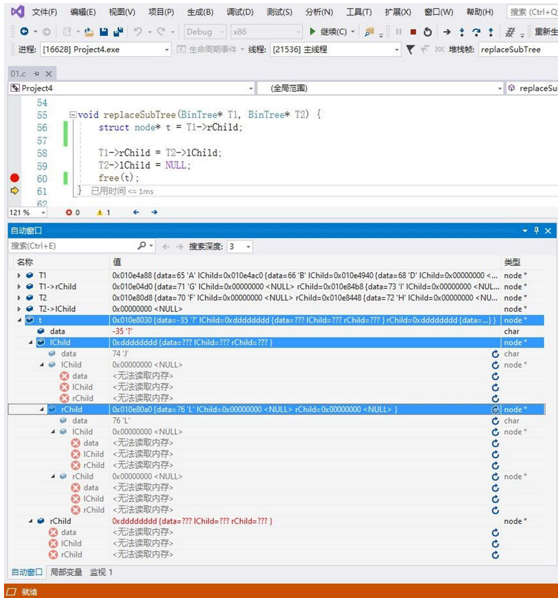


图 9 VS2019 中内存泄露的变量呈现方式

Fig. 9 Representation of memory-leaked variables in VS2019

结束语 本文提出了一个面向程序调试过程的树形结构演变可视化模型 TEVM，对树形结构演变过程进行切合编程初学者需求的可视化呈现。TEVM 可以对类树结构和泄漏树结构进行动态、平滑和直观的展示，弥补静态可视化容易产生突变的不足，帮助编程初学者在程序调试过程中快速地理解和掌握树形结构的演变过程，尤其是在发生内存泄漏的程序逻辑错误时，能够帮助他们更便捷地进行除错，提升程序调试效率。

在下一步工作中，我们将开展数据结构可视化通用性理论的研究，建立统一的数据结构演变可视化模型。

参 考 文 献

[1] SU Q, TANG Y H, ZENG Y A, et al. Research on Real-Time Identification and Visualization Layout Algorithm for Single Linklist Structure [J]. Computer Engineering and Applications, 2019, 55(16): 240-245.

[2] WU W M, LI X M, LIU T T, et al. Data Structure [M]. Beijing: Higher Education Press, 2017: 126-130.

[3] WANG G, NAKANISHI T, FUKUDA A. 2-D Layout for Tree Visualization: a survey[C]//MATEC Web of Conferences. EDP Sciences, 2016, 56: 1-12.

[4] BACHER I, MAC N B, KELLEHER J D. On using Tree Visualisation Techniques to support Source Code comprehension[C]//2016 IEEE Working Conference on Software Visualization (VISSOFT). IEEE, 2016: 91-95.

[5] YUAN T. System design tree compiler [J]. Information Technology and Network Security, 2014, 33(20): 10-11, 19.

[6] HEER J, ROBERTSON G. Animated transitions in statistical data graphics[J]. IEEE Transactions on Visualization and Computer Graphics, 2007, 13(6): 1240-1247.

[7] PLAISANT C, GROSJEAN J, BEDERSON B B. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation[C]//IEEE Symposium on Information Visualization, 2002. IEEE, 2002: 57-64.

[8] SHANMUGASUNDARAM M, IRANI P, GUTWIN C. Can smooth view transitions facilitate perceptual constancy in node-link diagrams? [C]//Proceedings of Graphics Interface 2007. 2007: 71-78.

[9] GUILMAINE D, VIAU C, MCGUFFIN M J. Hierarchically animated transitions in visualizations of tree structures[C]//Proceedings of the International Working Conference on Advanced Visual Interfaces. 2012: 514-521.

[10] SU Q, ZHANG S Y, HUANG J F, et al. Design and application of online visual programming cloud platform[J]. Experimental Technology and Management, 2020(7): 191-194, 203.

[11] CARD S K, MACKINLAY J D, SHNEIDERMAN B. Readings in Information Visualization: Using Vision to Think[M]. San Francisco, California: Morgan Kaufmann, 1999: 233-267.

[12] BOSTOCK M, OGIEVETSKY V, HEER J. D'data-driven documents[J]. IEEE Transactions on Visualization and Computer Graphics, 2011, 17(12): 2301-2309.

[13] BILLE P. A survey on tree edit distance and related problems [J]. Theoretical Computer Science, 2005, 337(1/2/3): 217-239.

[14] DE LUCA F, HOSSAIN I, KOBOUROV S, et al. Multi-level tree based approach for interactive graph visualization with semantic zoom[J]. arXiv: 1906.05996, 2019.

[15] WARD M O, GRINSTEIN G, KEIM D. Interactive data visualization: foundations, techniques, and applications[M]. Natick, Massachusetts: AK Peters/CRC Press, 2015: 120-132.

SU Qing, born in 1979, Ph.D, associate professor, is a member of China Computer Federation. His main research interests include visual computing and so on.

LIU Tian-tian, born in 1979, M.S, lecturer. Her main research interests include visual computing and so on.