JHAVEPOPC: VISUALIZING LINKED-LIST OPERATIONS IN C++ AND JAVA

David Furcy
University of Wisconsin Oshkosh
Oshkosh, WI, USA
furcyd@uwosh.edu

ABSTRACT

JHAVEPOP is a program visualization tool specialized in elementary pointer and linked-list operations. JHAVEPOP is a web-started application pre-packaged with ready-made programming exercises in both Java and C++. The only preparation needed by instructors to use JHAVEPOP is telling their students what URL to load in their browsers. Once there, students read the problem statement, enter their program, and interact with an automatically-generated, visual representation of the state of memory during program execution. JHAVEPOP has been used at both the high-school and college level. Student feedback indicates that JHAVEPOP is helping them debug their programs faster and with less frustration.

1. INTRODUCTION

JHAVEPOP is a new program visualization tool for linked lists. Such tools have been shown to foster student learning [2], which is important given that pointer-based data structures are notoriously hard for students to understand and thus present great challenges for instructors [1] [3]. JHAVEPOP lets students type their C/C++ or Java code and automatically generates a step-by-step depiction of the contents of memory as each program statement is executed. JHAVEPOP is delivered over the web and comes pre-packaged with a set of programming exercises that are only one click away and ready to use. Instructors and students have been quite enthusiastic about using JHAVEPOP either for laboratory activities or as pedagogical tools in the classroom.

In this paper, we first compare JHAVEPOP (http://jhave.org/jhavepop/) to existing tools. Then, we describe its architecture and demonstrate a typical use case. Next, we report user feedback. We conclude with a discussion of future work.
2. RELATED WORK

Quite a few program visualization tools have been used over the past three decades. “PASCAL Pointers” was a seminal program written by Jeffrey Popyack for the Macintosh in the mid-1980s, and whose goals and scope most closely resemble ours, since it did for PASCAL pointers what JHAVEPOP does for C/C++ pointers and Java references, namely help students learn how to use them to manipulate singly-linked lists.

Among modern software visualization tools that can automatically visualize programs, jGRASP [3] and Jeliot [6] are arguably the most widely used at present. While both tools have a wider scope than JHAVEPOP, which is specialized on visualizing linked lists, they are also less flexible than JHAVEPOP since they can visualize “only” Java programs. In contrast, JHAVEPOP has been used in data structures courses in either C/C++ or Java. Therefore, JHAVEPOP can help students detect such dynamic memory deallocation errors as dangling pointers and memory leaks. As will be discussed in Section 3.4, JHAVEPOP supports a common subset of Java and C++. Another significant difference is that JHAVEPOP comes pre-packaged with a set of programming exercises that instructors can choose from. These exercises are ready to use and accessible to students with one click. Finally, the ambitions of these tools are quite different, since JHAVEPOP is a standalone resource, with no learning curve, that is well-suited for use in one or two lectures or labs on one specific topic, while jGRASP is a full-fledged IDE and Jeliot aims to help novices learn Java programming, starting with expression evaluation and assignment statements. These different overarching goals explain other smaller differences, such as layout options, that may influence their pedagogical effectiveness, as we discuss further in Section 3.5.

More recent work includes tools such as problets [5], which present students with C++ code (but not Java) and help them debug it, as well as HDPV [8], a visual debugging tool for C++ and Java which, to the best of our knowledge, has not yet been made available to the general public.
3. VISUALIZING LINKED-LIST OPERATIONS WITH JHAVEPOP

Since JHAVEPOP uses the JHAVÉ platform, we start by highlighting those architectural features of JHAVÉ that are most relevant to JHAVEPOP. Then, we provide a user guide to JHAVEPOP, including a specification of the supported common subset of C++ and Java.

3.1 The JHAVE educational platform

The Java-Hosted Algorithm Visualization Environment, or JHAVÉ [7], supports a variety of algorithm visualization (AV) engines by providing them with:

- a common drawing context called the “main pane,”
- a standard set of DVR-like controls that allow students to step through a series of snapshots, one for each step in the algorithm being visualized in the main pane,
- two side panes where dynamically-generated HTML pages display accompanying text, such as pseudo-code and explanations of the visualization,
- input generators that gather student-generated input data needed by the algorithm,
- stop-and-think questions that pop up randomly during the visualization to foster active learning, and
- a back-end database that, for automatic assessment purposes, keeps a record of all the student’s answers.

JHAVÉ has a script-based, client-server architecture. In a typical session, the student web-starts the client program and selects an algorithm to visualize. If input is required, the server sends back an input generator. Once the student is done entering the input data, the server runs the algorithm and generates a script that contains a description of a sequence of snapshots that portray the behavior of the algorithm. The AV engine (client) then retrieves this script and visualizes it in the main pane, moving forward and backward through the snapshots as the student clicks on the DVR-like controls, asking questions at random times and recording the student’s answers.

3.2 The JHAVEPOP application

JHAVEPOP (JHAVÉ-Hosted Automatic Visualization of Elementary Pointer OPera-
tions) uses one of the existing AV engines supported by JHAVÉ. When using JHAVEPO\textsubscript{P}, a student needs to open up a web browser and their favorite text editor or IDE. On the main JHAVEPO\textsubscript{P} page, namely \url{http://jhave.org/jhavepop/<lang>/exercises.html} (where <lang> must be replaced with either java or cpp depending on the language used in the course) are the programming exercises. After reading the problem statement, the student starts up the JHAVEPO\textsubscript{P} client by clicking on the adjacent button. The JHAVÉ server sends back an input generator, which is a pop-up window for the student to paste his or her code into. The server then parses the input program, checks it for syntax errors, and if none are found, automatically generates and sends back to the client a sequence of snapshots that contain a graphical representation of the state of memory after each statement in the input program. At this point, the JHAVEPO\textsubscript{P} window displays the first snapshot in the main pane and the formatted source code in one of the side panes (the second side pane is used to display explanations on how to interpret the snapshots). The student is then free to navigate through these snapshots to verify that the program is working properly, and if not, find out why. The displayed snapshot always corresponds to a single statement or expression in the source code, which is highlighted in red in the side pane. JHAVEPO\textsubscript{P} makes debugging easier by enabling the student to compare his or her mental model of what should be happening to what is actually happening. If a discrepancy is detected, the student can go backward through the snapshots to study what happened right before the bug was detected, a particularly useful feature that is not available in most debuggers, including jGRASP and HDPV. The student may then modify their code in the IDE and paste it back into the JHAVEPO\textsubscript{P} window to verify their new code.

3.3 A sample JHAVEPO\textsubscript{P} exercise

The solution to each JHAVEPO\textsubscript{P} exercise is a code fragment (not a full program/ function/method) and assumes that one or more linked lists are already built. The student’s code is expected to modify them in some specific way. JHAVEPO\textsubscript{P} has a predefined Node class (or struct) containing two fields, namely a char field called ‘data’ and a pointer
Write a C++ program that, given a linked list of characters, removes all the elements in the list that are immediately preceded by an element with the same data. In other words, after executing your code, the list should not contain any consecutive duplicates (non-consecutive duplicates need not be deleted). All elements remaining in the linked list must appear in their original order. As shown in the test cases below (not shown in this paper), your code may assume that the pointers head and tail are initialized to the first and last nodes of the linked list, respectively, or to NULL when the linked list is empty. This invariant must still hold after execution of your code. Note that two additional pointers p1 and p2 are declared. For this exercise, you may not declare any additional pointers. Make sure that your code works on all possible linked lists.

Figure 1: First (top) and last (bottom) snapshots in a sample visualization

Figure 1 shows the first and last snapshots of the visualization for one solution to this problem. The source code in the right pane has been formatted and the lines numbered by JHAVEPOP. In each snapshot, the statement (or expression) in red in the right pane has
just been executed (or evaluated). The variables and objects that were involved in this step are also highlighted in red in the main pane. The pointers p1 and p2 are not displayed in the first snapshot since their declaration statement has not been executed yet.

Figure 2 shows the final snapshot of the visualization for another “solution” program to the same exercise. This code does not perform any memory deallocation. Note that the orphan nodes (memory leak) appear grayed out in the main pane.

**Figure 2: A memory leak is depicted as grayed-out orphan nodes**

Figure 3 shows the next-to-last snapshot of the visualization for yet another “solution” program to the same exercise. This code does not handle the tail pointer properly, and thus creates a dangling pointer.

**Figure 3: A dangling pointer is depicted as a blue arrow**
Figure 4: A NULL pointer “exception” was thrown by buggy code

Finally, Figure 4 shows the final snapshot of the visualization for the same code as in Figure 3. In this snapshot, JHAVEPOPO has “thrown an exception,” which simply means that the student code would have crashed at this step. But JHAVEPOP did not crash: the student is free to go back to the previous snapshot (using the button in the bottom left corner of the window) to study what the state of memory looked like just before the error occurred (see Figure 3). The last statement to be successfully executed (still shown in red in both Figures 3 and 4) is \texttt{delete p2}; The error thus occurred in the next statement, namely during the evaluation of the Boolean expression in the following conditional statement, since the value of \texttt{p1->next} is NULL. Note that the NULL pointer value is shown as a diagonal bar across the corresponding pointer variable (see the value of the 'next' field in Node D in Figure 3). In contrast, uninitialized or unreliable values, such as that of the variable \texttt{p2} in Figure 3, are depicted as an empty square, since the Node referenced by \texttt{p2} was just deallocated.

3.4 Language support in JHAVEPOPO

The reason why JHAVEPOPO does not crash when the student code does is because it does not invoke the Java Virtual Machine nor any machine-code generated by a C++ compiler. Instead, JHAVEPOPO contains its own interpreter for a subset of Java/C++. The parser used by JHAVEPOPO was generated with JAVACC and accepts most operations pertaining to memory (de)allocation, pointer/reference assignments, pointer dereferencing,
as well as standard control structures (if statements, for and while loops). As shown in the previous figures, the JHAVEPOP runtime system also includes a function called createList that creates a whole list in one step, which is convenient when setting up an exercise. For more details on the language subset supported by JHAVEPOP, refer to http://jhave.org/developer/david_furcy_jhavepop/JHAVEPOP.htm.

3.5 The JHAVEPOP user interface

One advantage of the JHAVÉ platform is that it makes available to each AV engine it supports an extensive set of navigation tools that afford a higher level of interactivity than is possible in either jGRASP or Jeliot. For example, JHAVEPOP supports zooming and panning in its main pane, allowing students to focus their attention on different subsets of the data structures being modified at each given point in the program.

As part of JHAVÉ, JHAVEPOP inherits multiple ways of navigating through the visualization, not only forward, but also backward, which we feel is important in an educational setting, as we argued above. Moreover, the JHAVEPOP window includes a slider (at the bottom right, in each figure above) that makes it possible to jump, with one click, to any point in the execution of the program, whether the beginning, end, or anywhere in between. Whether students are working on their home computer or with a faculty member during a lab session or office hours, being able to move through the visualization in both temporal directions (and with direct access in non-sequential order) speeds up the process of identifying the exact point where the student’s mental model breaks down. Therefore, the instructor can spend more time helping the student improve their mental model. And when working on their own, students tend to be less frustrated during debugging sessions (see Section 4).

One interesting aspect of the JHAVEPOP user interface is its layout scheme, since positioning the elements of the data structure(s) for maximum visual and pedagogical effectiveness is a hard problem. For linked lists, the standard scheme (used by both jGRASP and Jeliot), is a horizontal or vertical layout, which works well for lists that only grow at one end. If nodes are inserted in the middle, then either the list is not linear anymore (which reflects
the actual memory layout), or all the elements following the insertion point need to be moved on the (virtual) display. jGRASP uses the latter approach, which might seem unintuitive to the student since the data space in actual memory is not modified in this way. The problem worsens when the exercise involves more than one list, for example, when computing the union or intersection of two sets represented as lists. But even when implementing the list reversal algorithm in jGRASP, we observed that the two sub-lists (already reversed versus not yet reversed) kept swapping places on the screen, which is somewhat disconcerting. In contrast, Jeliot uses the latter approach, in which the nodes do not move. However, because of the fixed linear layout, the reversal of even a short (4-node, in our example) linked list create a “plate-of-spaghetti” effect that makes each operation harder to follow than the previous one. For this reason, the layout is customizable in JHAVEPOP. Both pointer variables and linked-list nodes may appear anywhere on a 2D grid. So, while the default layout is linear, the user (instructor or student) may decide to override the default and specify the position of some or all elements with a pair of indices into the 2D grid specified as regular comments (using the double-slash notation) within the code. Continuing with our list-reversal exercise, a circular layout of the linked list is much more visually appealing.

Finally, the simple fact that JHAVEPOP draws the memory state in a single window is a significant advantage in that students need not switch their attention from window to window while debugging. In contrast, one side-effect of jGRASP’s powerful data-structure inference algorithm is that not all pointer variables are always visible. In some of our exercises, the only way to produce in jGRASP a complete picture of the local pointer variables and Node objects in the heap is to keep several windows open, one for each local variable. Having to switch attention among several windows increases the cognitive load. Furthermore, since several local pointer variables often point to the same list, and these local pointer variables are split across windows (the viewers), the linked list is displayed multiple times. As a result, it becomes hard to predict which window to focus on next in order to observe the effect of the next program statement.
4. USER FEEDBACK

The JHAVEPOP project started in June 2007. It was made available online and advertised on the SIGCSE mailing list in January 2008. Since then, the website has received approximately 2,000 visits from 92 countries, for a total of about 3,000 page views by 1,500 “absolute unique visitors” (as defined by the Google Analytics software). The split between the C++/Java sections of the website is approximately 55%/45%. The vast majority of users are in the US and in Finland (and for this reason, there exists a Finnish translation of the JHAVEPOP exercise website). Unfortunately, since users are not asked to sign up, the actual number of courses in which JHAVEPOP was used so far is not known. Therefore, from this point on, we will focus on users which we know for sure used JHAVEPOP in their courses. This category includes eight faculty members, two on our campus, two in Finland, one in a US high school, and the other three at other US universities. Out of these, at least three have used JHAVEPOP for more than one semester, including the Spring 2009 semester. We estimate the total number of student users to be over 300, out of which approximately 120 answered a questionnaire that we put together to collect their feedback on JHAVEPOP.

The questionnaire was made available as an online form. Questions ranged from the context of use (in lectures, in lab, or on your own), to open-ended comments on the strong and weak points of the tool. Most often, JHAVEPOP was not discussed at all in lectures, and in most other cases it was demoed for less than 5 minutes. JHAVEPOP was most often used in the context of a lab session. Here are the two most common types of positive comments:

- JHAVEPOP deepens student understanding of pointers by making them concrete; representative comments include: “telling all the time what is happening where, helped me to understand this difficult subject”; “Veeeery (sic) concrete. You can see all the steps one by one”; “[it was good] in concretizing the situation”; “this is a much better way to learn and to understand the code”; “With this you understand what is ‘really’ happening.”

- JHAVEPOP helps students not only understand but also debug their programs; repre-
sentative comments include: “you see exactly what your code actually does”; “The picture made my own mistakes easy to see.”; “Understanding logical errors is easier.”; “Visualization of the code helped show where I went wrong”; “It was quicker”; “You could see the result directly from the picture and not from the difficult error messages.”; “the pictures showed directly if you had dropped one node out by a mistake”; “no need to struggle with error messages.”

The vast majority of negative comments about JHAVEPOP were usability-related. First, many students did not like having to paste their code from an outside editor, which adds one step in the edit-compile-run (ECR) cycle. This extra step is needed because the JHAVE server does not remember the input from one cycle to the next. So, the input generator pops up empty every time. This is only frustrating if you end up retyping your code from scratch every time, as quite a few students apparently did! Even though the directions recommend the use of a separate editor, students (and instructors) did not always read them. In any case, we believe that introducing an extra copy/paste step in the ECR cycle may actually be advantageous if it forces students to reflect more about their design between each cycle [3].

The second most common criticism of JHAVEPOP is the fact that exercises are independent tasks that require a new session each time, which means having to close the window at the end of an exercise and to click a button to start the next one. This should not be much of a problem unless a student completes a large number of exercises in a short time, in which case they may not need to use JHAVEPOP in the first place. Unfortunately, it appears that at least some students ended up closing and re-opening the JHAVEPOP window after each execution, in spite of the instructions. The third most common negative comment was the fact that JHAVEPOP does not support a larger subset of Java/C++. Surprisingly (to us), several students wanted to use the constant 0 instead of the NULL macro in C++.

5. FUTURE WORK AND CONCLUSION

Our first order of business is to add an automatic assessment component to JHAVEPOP, which will be facilitated by JHAVE’s existing support for “stop-and-think” questions and
back-end database. Second, we plan to address the usability issues discussed above, which will involve some work on the server side, namely to allow session tracking and continuity in the contents of the input generator. Finally, we hope that more instructors will give JHAVEPOP a try so that we can conduct a formal empirical evaluation of its effectiveness.

6. REFERENCES


