Universidade Federal da Paraíba
Centro de Informática
Programa de Pós-Graduação em Informática

Um Interpretador Gráfico de Comandos baseado na JVM como ferramenta de ensino de Programação, Algoritmos e Estruturas de Dados

Tiago Davi Neves de Sousa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Computação Distribuída

Andrei de Araújo Formiga
(Orientador)

João Pessoa, Paraíba, Brasil

Resumo

Em disciplinas de Programação, Estruturas de Dados e Algoritmos de cursos de Computação, ferramentas que permitam que os alunos possam visualizar as alterações nas estruturas de dados ao longo da execução de um programa são de grande utilidade, já que elas auxiliam que os aprendizes entendam como os algoritmos operam sobre as estruturas de dados. Diversas ferramentas foram propostas desde o trabalho pioneiro de [Brown e Sedgewick 1984]. Em algumas delas, as visualizações gráficas das estruturas através de animações só podem ser feitas através de programação pelos usuários e outras carecem de recursos que as impossibilitam de serem empregadas em todo o processo pedagógico. Assim, neste trabalho foi desenvolvido um Interpretador para a ferramenta de ensino IGED (Interpretador Gráfico de Estruturas de Dados). Esse Interpretador foi projetado baseado na JVM e possibilita que códigos que implementam vários algoritmos em uma linguagem de programação orientada a objetos sejam executados pela ferramenta de forma que esta gere como saída as visualizações gráficas das estruturas de dados. A arquitetura do Interpretador desenvolvido neste trabalho e seus componentes foram detalhados e requisitos funcionais que ele pode ter como ferramenta de ensino, sendo útil para outras disciplinas de Computação, foram definidos. Além disso, foi justificado porque houve uma implementação própria de um Interpretador para o IGED se já existem implementações da JVM disponíveis e amplamente utilizadas. Nos experimentos, foi demonstrado que o Interpretador pode executar códigos com características que são úteis para essas disciplinas.

Palavras-chave: Interpretadores, Programação, Educação, Estruturas de Dados.

Abstract

In disciplines of Programming, Data Structures and Algorithms of Computer Science courses, tools that permit the visualization of the the data structures changing throughout the execution of a program by the students are very useful because they assist that the students learn how the algorithms operate over the data structures. Many tools were proposed since the pioneer work of [Brown e Sedgewick 1984]. In some of them, the graphical visualization of the data structures through the animations can only be done by the users programming and in others there are a lack of resources that forbid their use in the whole pedagogical process. Thus, in this work an Interpreter for the IGED (Graphical Interpreter of Data Structures) teaching tool was developed. This Interpreter was designed based in the JVM and enable that codes implementing various algorithms in an object oriented language be executed by the tool so that it generates as output the graphical visualization of the data structures. The architecture of the Interpreter developed in this work and its components were detailed and the functional requirements it may have as a teaching tool, being useful for other disciplines of Computer Science, were defined. Furthermore, was justified why an own implementation of an Interpreter for the IGED was done if there are JVM implementations available and widely used. In the experiments, was demonstrated that the Interpreter may execute code with useful characteristics for these disciplines.

Keywords: Interpreters, Programming, Education, Data Structures.

Conteúdo

Lista de Símbolos

AV : *Algorithm Visualization*

AVL : *Adel'son-Vel'skii and Landis*

Balsa : *Brown Algorithm Simulator and Animator*

CIFluxProg : Construtor e Interpretador de Algoritmos

COGO : *Coordinate Geometry*

CORBA : *Common Object Request Broker Architecture*

CPU : *Central Processing Unit*

E/S : Entrada e Saída

FIFO : *First In First Out*

IGED : Interpretador Gráfico de Estrutura de Dados

IDE : *Integrated Development Environment*

IR : *Intermediate Representation*

Jasmin : *Java Assembler Interface*

JDI : *Java Debugger Interface*

JDK : *Java Development Kit*

JHAVÉ : *Java Hosted Algorithm Visualization Environment*

JIVE : *Java Interactive Software Visualization Environment*

JRE : *Java Runtime Environment*

JVM : *Java Virtual Machine*

JVM TI : *Java Virtual Machine Tool Interface*

LIFO : *Last In First Out*

MVC : *Model-View-Controller*

NCM : *Nested Context Model*

OO : Orientação a Objetos

PCIL : *PseudoCode Interpreted Language*

SO : Sistema Operacional

SQL : *Structured Query Language*

TAD : Tipo Abstrato de Dado

Twist : *Tango's wonderful image-synthesis toolkit*

VDSM : *View Data Structure Manager*

VM : *Virtual Machine*

UML : *Unified Modeling Language*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Em cursos da área de Computação, disciplinas de programação, algoritmos e estruturas de dados são de fundamental importância na formação de um aluno. Isso porque com as atividades relacionadas a essas disciplinas, os alunos aprendem os conceitos básicos de desenvolvimento de software, que é uma das atividades da computação que possuem aplicações mais importantes tanto no mercado de trabalho quanto no âmbito científico.

Apesar das atividades de programação serem de fundamental importância para um aluno e profissionais da área de computação, em muitas instituições de ensino os alunos se deparam com dificuldades na aprendizagem de programação, visto que essa atividade requer um raciocínio lógico não trivial sobre a manipulação de diversas estruturas de dados. Um aprendiz, nesse caso, poderá ter dificuldades porque ele terá que desenvolver algoritmos e, ao mesmo tempo, entender como os passos de seus algoritmos efetuarão alterações nas estruturas de dados.

Para facilitar o desenvolvimento de programas, várias ferramentas são usadas pelos programadores aprendizes e profissionais, como editores de texto, compiladores, interpretadores, máquinas virtuais, IDEs, etc. Os ambientes de desenvolvimento e as ferramentas de programação, apesar de fornecerem diversos recursos que facilitam o trabalho de um programador, com suporte à execução e *debugging* dos programas por eles feitos, geralmente não contém elementos que permitam a visualização dos diversos tipos de estruturas de dados que podem ser manipuladas por um programa.

Programadores profissionais conseguem utilizar bem as ferramentas disponíveis para programação, mas programadores aprendizes geralmente possuem dificuldades de aprender como funcionam seus algoritmos quando não há uma forma de visualizar como as variáveis e as estruturas de dados são alteradas ao longo da execução de um programa. Muitos recorrem ao lápis e ao papel, para que com o desenho, possam verificar melhor como funcionam seus algoritmos e como eles devem projetar um. Conforme [Rocha 1991], está havendo um fracasso no ensino de programação com o uso dos métodos convencionais de ensino e um grande desinteresse dos alunos em cursos introdutórios à programação.

1.1 Motivação

No estudo realizado em [Netto 2010] foram identificadas algumas dificuldades dos alunos na aprendizagem dos conceitos apresentados na disciplina de Estruturas de Dados. Nesse estudo foi mostrado que 33% dos alunos tiveram dificuldades com a abstração e 17% consideraram a complexidade do assunto como barreira ao aprendizado. Além disso, foi identificado que a maioria dos professores não utiliza ferramentas específicas no ensino nessa disciplina, além das ferramentas de uso geral em programação.

Algumas ferramentas de apoio ao ensino e aprendizagem de estruturas de dados foram criadas desde o trabalho pioneiro de [Brown e Sedgewick 1984]. Contudo, em muitas ferramentas é necessário que se programe as animações, além da escrita dos algoritmos a serem animados. Em algumas, o código com as chamadas das rotinas de animação são embutidos no próprio algoritmo. Como por exemplo, no Tango [Stasko 1990] os algoritmos são anotados com chamadas de procedimentos que realizam as operações de animação gráfica. O Astral [Garcia e Rezende 1997] também é um ambiente onde para realizar as animações é necessário escrever chamadas a operações gráficas fornecidas pela interface da ferramenta durante o desenvolvimento de um algoritmo.

Isso causa alguns problemas e dificuldades para o aprendiz, já que com isso, o código com as ações das animações e o código do algoritmo propriamente dito ficam misturados, dificultando a legibilidade. Além disso, em algumas ferramentas como em [Brown e Sedgewick 1984] e em [Brown 1991], deve haver um usuário que projete ou programe as animações,

além de um usuário que codifique o algoritmo. Nesse caso, para efetivar a abordagem do ensino de programação com essas ferramentas, um estudante precisará de outra pessoa que saiba programar as animações, se ele não souber. Para que ele não dependa de outro usuário, ele teria que aprender a fazer as animações, o que pode atrapalhar no processo de aprendizagem, visto que o aluno só deveria se esforçar em aprender a desenvolver um algoritmo em si, já que aprender a programar animações não faz parte do processo de aprendizagem de uma disciplina de programação, algoritmos ou estruturas de dados.

Existem muitas ferramentas disponíveis na web que permitem a visualização dos algoritmos que elas executam. Em algumas, apresentadas no capítulo 3, os algoritmos a serem animados não são construídos pelos usuários. Dessa forma, a abordagem construtiva não é empregada com muita efetividade, já que não são os usuários que constroem os seus algoritmos.

Na abordagem construtiva, ao contrário da passiva, o processo de aquisição do conhecimento se dá por meio do ato de construir algo. No processo educacional, essa construção do conhecimento pode ser feita por meio da reflexão e da pesquisa sistemática que leva a novos conhecimentos como também no modo como cada um aprende de forma semelhante, mas não idêntica. Nesse segundo caso, o aprendiz não constrói propriamente o conhecimento. O conhecimento adquirido cientificamente é passado para ele em salas de aula e em materiais didáticos, mas o processo de aquisição do conhecimento é construtivo quando o conteúdo que é transmitido para o aprendiz ou que é descoberto por ele vai ser organizado e estruturado de modo pessoal e peculiar, onde sujeito interfere no objeto do conhecimento a seu modo [Werneck 2006].

Sendo assim, uma ferramenta chamada IGED (Interpretador Gráfico de Estruturas de Dados) está sendo desenvolvida para fornecer a aprendizes os recursos necessários para que eles possam aprender a desenvolver algoritmos através da codificação dos mesmos e da visualização da execução das estruturas de dados manipuladas por esses algoritmos. Ela possuirá como característica fundamental a possibilidade de que um aluno verifique as alterações nas estruturas de dados ocorridas durante a execução dos seus algoritmos, sem que para isso precise programar animações. Ele terá apenas que fornecer um algoritmo codificado na lin-

guagem de alto nível da ferramenta, que será uma linguagem com suporte à orientação a objetos e que possui sintaxe baseada nas linguagens C/C++ e Java.

Essa ferramenta também possui um tutor, onde um usuário professor pode cadastrar exercícios e usuários alunos podem responder a esses exercícios com soluções em código na linguagem da ferramenta. Para auxiliar os alunos na resolução desses exercícios e no entendimento dos assuntos, eles dispõem de recursos como exemplos de código fonte, slides e vídeos que podem ser inseridos através de uma linguagem declarativa de acordo com o modelo de autoria NCM [Filho et al. 2012]. Além disso, trabalhos estão sendo feitos para que o IGED possa determinar se uma solução informada por um aluno em um problema está correta ou não.

No entanto, a ferramenta IGED ainda não possuía um Interpretador, componente que seria responsável por executar os algoritmos definidos pelos usuários codificados em uma linguagem de programação, fazendo com que o IGED gere as animações de forma automática para um usuário. Dessa forma, não era possível a execução de códigos para a visualização das execuções dos algoritmos por meio das representações gráficas das estruturas de dados, que é a funcionalidade principal da ferramenta de acordo com os requisitos definidos em [Netto et al. 2011]. Sem um Interpretador, a abordagem construtiva no auxílio ao ensino, a qual a ferramenta se propõe a oferecer não pode ser empregada.

Isso seria possível com a incorporação do interpretador que foi desenvolvido para a ferramenta IGED e que é apresentado neste trabalho. Com a interpretação do código informado por um usuário, as operações de animação poderão ser invocadas pelo próprio interpretador, o que livra o usuário de realizar tal tarefa. Além disso, o interpretador está sendo implementado em Java, o que permite que o código de programação informado pela ferramenta IGED seja executado em diversas plataformas.

1.2 Objetivos

O objetivo principal deste trabalho é a adição de um interpretador ao IGED que auxilie a suprir a necessidade da execução de algoritmos codificados em uma linguagem de progra-

mação de alto nível com reprodução automática da exibição das animações das estruturas de dados, conforme um programa é executado, de forma gráfica dinâmica e totalmente transparente ao usuário, que só teria o trabalho de fornecer os códigos dos algoritmos. Com esse interpretador, são mostradas as características que ele possui e quais são as suas funcionalidades.

Para isso, esse interpretador foi integrado ao projeto IGED [Netto et al. 2011]. Assim, ele deve receber como entrada o código de baixo nível gerado pelo componente Tradutor que traduzirá o código de alto nível informado pelo usuário. O Interpretador executará o código traduzido em um subconjunto da linguagem Oolong [Engel 1999], que é utilizada como linguagem de montagem de *bytecodes* Java.

O Interpretador foi implementado na linguagem Java, para que possa ser executado de forma simples em diversas plataformas e também para que possa ser integrado de forma mais fácil aos outros componentes do IGED, já que eles também foram implementados em Java. Além disso, esse interpretador deve possuir uma estrutura baseada na especificação da JVM, embora o Interpretador não seja de fato uma implementação da JVM.

Com esse interpretador, pretende-se definir também requisitos gerais que ele pode ter para o ensino em Computação como uma forma de referência para trabalhos futuros que visem a melhoria da ferramenta IGED, adicionando funcionalidades ao interpretador.

Outro objetivo deste trabalho é efetuar uma análise comparativa com uma possível implementação da JVM. Assim, pretendem-se verificar quais são as vantagens e desvantagens de utilizar os interpretadores que possuam as seguintes características para o projeto IGED:

- Uma implementação de um interpretador em Java, que possua apenas as características necessárias para o projeto IGED, não englobando necessariamente todas as características que uma implementação da JVM deve possuir, de acordo com a Oracle. Dessa forma, esse interpretador não é uma implementação completa da JVM, mas uma implementação específica para o IGED. Esse interpretador é o que foi desenvolvido neste trabalho.

- Uma implementação da JVM, como a Java HotSpot VM da Oracle, que também possui um interpretador que opera sobre uma máquina virtual.

Com essa análise comparativa, pretende-se fornecer informações que sejam úteis a trabalhos futuros que tenham como objetivo a melhoria e o aperfeiçoamento do Interpretador do IGED, como adição de funcionalidades, otimização de desempenho e correção de erros. Isso porque nessa análise, serão mostradas vantagens e desvantagens das duas formas de interpretadores para o IGED, auxiliando na criação de trabalhos futuros que levem em consideração uma implementação de um Interpretador para o IGED que obtenha o máximo de características positivas, ou seja, vantagens de ambas as formas. Além disso, com essa análise comparativa, pretende-se justificar porque foi feita uma implementação específica de um interpretador para o IGED se já existem implementações da JVM disponíveis gratuitamente e com ampla utilização.

1.3 Metodologia

Neste trabalho, o método empregado no processo de abordagem do problema foi o método hipotético-dedutivo e o meio técnico utilizado na investigação foi o experimental, conforme são definidos em [Prodanov e Freitas 2013]. A adição de um interpretador para o IGED supre a necessidade da execução automática dos códigos informados por um usuário por meio de uma linguagem de programação. Isso porque, segundo [Aho, Sethi e Ullman 1995] interpretadores são tradutores que executam o código fonte recebido como entrada após traduzi-lo. Assim, aplicando um interpretador ao IGED, este executaria código de acordo com a entrada fornecida pelo usuário, gerando com essa execução, saídas com atualizações gráficas que exibiriam para o usuário as alterações nas estruturas de dados de forma dinâmica.

O IGED, como é mostrado no capítulo 4, possui um componente Abstração Gráfica que gera uma determinada atualização gráfica sobre uma estrutura de dados, exibindo essa atualização na tela. Essa atualização ocorre quando uma operação específica desse componente é invocada. Dessa forma, o Interpretador desenvolvido neste trabalho é o responsável por invocar tais operações para os usuários quando estiver executando código. Com isso, o pro-

cesso de geração das visualizações das alterações nas estruturas de dados é transparente para o usuário.

Para que isso seja possível, uma linguagem chamada Oolong foi utilizada, que é uma linguagem de montagem de *bytecodes* Java [Engel 1999]. Uma linguagem de montagem foi escolhida para que códigos nessa linguagem sejam executados pelo Interpretador porque de acordo com os requisitos da ferramenta IGED [Netto et al. 2011], o Interpretador deve executar código em uma linguagem de baixo nível, traduzido pelo componente Tradutor do IGED. Além disso, por ser uma linguagem de montagem de *bytecodes* Java, essa linguagem possui as mesmas características que a linguagem de programação Java possui e que a caracteriza como uma linguagem de programação orientada a objetos e de uso geral.

Ao utilizar essa linguagem para o Interpretador do IGED, obtém-se uma série de vantagens, já que com ela é permitido o uso de construções das linguagens de programação que são utilizadas em cursos de Estruturas de Dados, Algoritmos e Programação como recursividade, definição de classes e objetos, métodos, operações lógicas e aritméticas, dentre outras. Uma desvantagem de utilizar essa linguagem é que como ela é de propósito geral, seu conjunto de instruções possui muitas instruções não necessárias à ferramenta IGED, o que aumentaria a complexidade do desenvolvimento do Interpretador de forma desnecessária. Por isso, foi utilizado um subconjunto dessa linguagem, com apenas as instruções necessárias para o Interpretador, já que ele tem um propósito específico voltado ao ensino de Programação, Algoritmos e Estruturas de dados com a ferramenta IGED.

Neste trabalho, o problema a ser resolvido é o de prover um meio que execute código de forma automática na ferramenta IGED, como foi explicado anteriormente. Por isso, nesse contexto, a adição do Interpretador desenvolvido neste trabalho pode solucionar tal problema pelos motivos citados anteriormente. Com esse Interpretador, pretende-se verificar que ele fornece características ao IGED que possibilitam a execução automática de códigos com os principais recursos disponíveis em linguagens de programação estruturadas e orientadas a objetos e que são empregados em disciplinas de programação, algoritmos e estruturas de dados na construção de códigos.

Para validar que o Interpretador do IGED fornece tais características e que com ele é possível a reprodução das animações sobre as estruturas de dados do IGED, foram realizados experimentos com a execução de códigos em Oolong por meio do Interpretador desenvolvido neste trabalho.

Capítulo 2

Fundamentação Teórica

2.1 Algoritmos

Um algoritmo é definido em [Cormen et al. 2002] como qualquer procedimento computacional, formado por uma sequencia de passos que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída, resolvendo um problema computacional bem especificado. Além disso, é dito como correto se para cada instância de entrada ele pára com a saída correta, ou seja ,se resolve o problema considerado.

De acordo com [Sipser 2007], algoritmos também desempenham um papel importante na matemática, já que a literatura matemática contém descrições de algoritmos para uma variedade de tarefas. Ainda segundo [Sipser 2007], um algoritmo existe para um determinado problema se ele pode ser definido em uma sequência de passos que seja decidível por uma Máquina de Turing.

Segundo [Cormen et al. 2002], a eficiência de um algoritmo para um determinado conjunto de entradas pode ser determinado pela análise de como o tempo de execução de um algoritmo aumenta com o tamanho da entrada no limite, à medida que esse tamanho aumenta indefinidamente, sem limitação. Dessa forma, pode-se determinar a eficiência assintótica do algoritmo.

2.1.1 Notações Assintóticas

A notação Θ [Cormen et al. 2002] é definida de forma que uma função pertence ao conjunto $\Theta(g(n))$ se existem constantes positivas c_1 e c_2 tais que ela possa ser "imprensada"entre $c_1g(n)$ e $c_2g(n)$, para valores de n suficientemente grandes e menores que um n_0. Abaixo está uma declaração formal de Θ e na figura 2.1(a), uma visualização intuitiva desse conceito.

$\Theta(g(n))$ = {f(n): existem constantes positivas c_1, c_2 e n_0, tais que $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ para todo $n \geq n_0$}.

Como a notação Θ limita assintoticamente uma função acima e abaixo, usa-se a notação O [Cormen et al. 2002] para definir apenas o limite superior para uma função, dentro de um fator constante, ou seja, para todos os valores de n maiores que n_0, o valor da função f(n) está em ou abaixo de g(n). Abaixo está a definição formal e em seguida, na figura 2.1(b), uma visualização intuitiva.

O(g(n)) = {f(n): existem constantes positivas c e n_0, tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$}

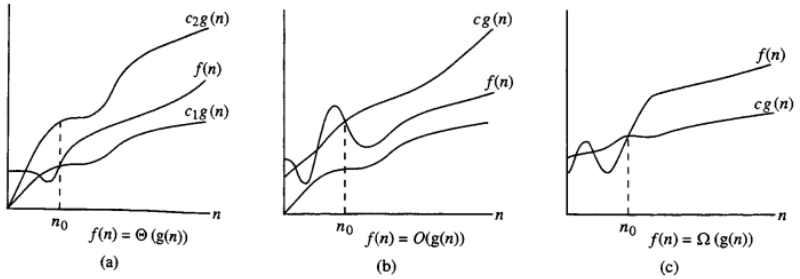
De forma similar à notação O, a notação Ω [Cormen et al. 2002] fornece um limite assintótico inferior para uma função. Sua visualização intuitiva está ilustrada na figura 2.1(c). Definição formal:

$\Omega(g(n))$ = {f(n): existem constantes positivas c e n_0, tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$}

Além dessas notações, há ainda as notações *o* e ω [Cormen et al. 2002] que definem respectivamente os limites assintóticos superiores e inferiores que não são assintoticamente restritos. Dessa forma, para qualquer constante positiva $c > 0$, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$ para *o* ou $0 \leq cg(n) < f(n)$ para ω para todo $n \geq n_0$. Exemplos:

- $2n = o(n^2)$, mas $2n^2 \neq o(n^2)$, embora ambas sejam $O(n^2)$;
- $n^2/2 = \omega(n)$, mas $n^2/2 \neq \omega(n^2)$, embora $n^2/2 = \Omega(n^2)$;

Figura 2.1: Visões intuitivas das notações assintóticas



Fonte: [Cormen et al. 2002]

2.2 Estruturas de Dados

Para solucionar problemas por meio de uma linguagem de programação, deve-se seguir várias etapas, que envolvem a identificação das propriedades dos dados e suas características funcionais. Para isso, tanto um programador profissional quanto um estudante devem conhecer como organizar de maneira estruturada os dados a serem manipulados [Celes, Cerqueira e Rangel 2004].

Segundo [Celes, Cerqueira e Rangel 2004], as estruturas de dados podem ser tanto estáticas como dinâmicas. As estruturas de dados estáticas são aquelas que não oferecem suporte adequado para inserção e remoção de elementos dinamicamente e são baseadas na utilização de formas primitivas de estruturação de dados disponíveis pelas linguagens de programação. Alguns exemplos são vetores e tipos estruturados. As estruturas de dados dinâmicas, por outro lado, oferecem o suporte adequado para a inserção e remoção de elementos com alocação dinâmica de memória para o armazenamento de cada um desses elementos , não sendo portanto estruturas pré-dimensionadas, ou seja, não há um limite para a quantidade máxima de elementos a serem informados, enquanto que em estruturas estáticas esse limite deve ser informado previamente em definições da linguagem de programação que estiver sendo empregada. Alguns exemplos de estruturas dinâmicas de dados são as listas, pilhas e filas encadeadas e árvores [Celes, Cerqueira e Rangel 2004]. A seguir será feita uma descrição das principais estruturas de dados estáticas e dinâmicas que devem ser consideradas na formulação de algoritmos para a construção de programas em uma linguagem de programa-

ção.

2.2.1 Vetores

É a forma mais primitiva de armazenar um conjunto de dados na memória do computador e a maioria das linguagens de programação permitem a definição de vetores de forma nativa [Celes, Cerqueira e Rangel 2004]. Em vetores, os dados correspondentes a seus elementos ficam armazenados de forma contígua na memória. Geralmente, para a definição dessas estruturas nessas linguagens, é necessário definir o tipo de dados que todos os seus elementos devem possuir e o tamanho ou quantidade máxima de elementos que essa estrutura deve possuir.

Algumas linguagens de programação, como C, permitem que o programador aumente o tamanho de um vetor depois que ele foi declarado em um processo conhecido como alocação dinâmica [Oliveira 2008]. Vale ressaltar que esse processo é dependente de linguagem de programação e que nem todas dão suporte à alocação dinâmica.

O acesso aos elementos é feito por meio da indexação, ou seja, é necessário apenas especificar o índice em que o elemento desejado ocupa no vetor [Celes, Cerqueira e Rangel 2004]. Isso constitui uma vantagem desse tipo de estrutura, visto que o acesso a um elemento é facilitado para o programador, que pode fazer isso em uma simples declaração, como a seguinte em C: *int x = v[5]*, onde a variável *x* recebe o valor do elemento de índice 5 do vetor *v*.

Por outro lado, uma desvantagem dessa estrutura está na alteração do vetor, com as operações de inserção e remoção de um determinado elemento, que não é tão simples e trivial quanto o acesso, visto que deverá ser feita uma atualização das posições de todos os elementos de índice superiores ao elemento que está sendo inserido ou removido. Assim, dependendo do tamanho do vetor e da posição a ser inserida ou removida, o processo poderá resultar numa grande quantidade de passos de execução de um algoritmo.

2.2.2 Matrizes

São estruturas de dados que correspondem aos vetores com duas ou mais dimensões [Celes, Cerqueira e Rangel 2004]. Para definir uma matriz, é necessário informar o tamanho que essa estrutura possuirá em cada uma de suas dimensões. Como por exemplo, uma matriz bidimensional pode ser declarada da seguinte forma na linguagem C: *float mat[4][3]* [Celes, Cerqueira e Rangel 2004]. Essa declaração representa uma matriz *mat* de valores reais com 4 linhas e 3 colunas. Sendo assim, os elementos dessa matriz são acessados com indexação dupla, da forma *mat[i][j]* , onde o primeiro índice acessa a linha e o segundo acessa a coluna. Uma matriz é representada como um vetor de ponteiros, onde cada ponteiro referencia uma estrutura de dados do tipo vetor [Oliveira 2008].

2.2.3 Listas Encadeadas

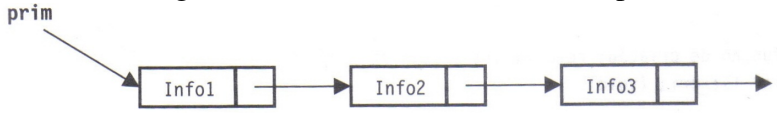
Vetores e matrizes, fornecem facilidades ao programador como o acesso randômico ou aleatório aos elementos com apenas uma referência ao primeiro elemento [Celes, Cerqueira e Rangel 2004]. Como foi mencionado anteriormente, o uso de vetores e matrizes possuem algumas desvantagens e limitações em relação à inserção e remoção de elementos e à alocação de mais espaço na memória em tempo de execução, caso seja necessário.

As listas encadeadas são estruturas de dados usadas para superar as limitações encontradas em vetores. São coleções de nós armazenadores de dados e de ligações com esses nós, de forma que eles podem estar em qualquer lugar da memória [Drozdek 2005]. Essas ligações são feitas quando um nó armazena o endereço de memória do nó sucessor. Em uma lista encadeada, há alocação de memória para cada elemento que é inserido e há a liberação de memória para cada elemento que é removido [Celes, Cerqueira e Rangel 2004].

Na figura 2.2, é ilustrada uma lista encadeada simples *prim*, onde são armazenados os valores *Info1*, *Info2* e *Info3* e as setas representam as referências ao primeiro elemento e aos outros nós.

Uma lista, de acordo com [Drozdek 2005], pode ser implementada de forma duplamente encadeada, quando cada nó, além de possuir referência ao seu sucessor (com exceção do

Figura 2.2: Uma lista encadeada simples



Fonte: [Celes, Cerqueira e Rangel 2004]

último), também possui referência ao seu antecessor (com exceção do primeiro), de forma circular, na qual os nós formam um anel, com outras formas de organização que resolvem o problema da necessidade da varredura sequencial nas duas últimas formas de implementação (listas com salto) e organizando a lista dinamicamente de forma a melhorar a eficiência em processos de busca (listas auto-organizadas).

Além dessas implementações, outras formas incluem tabelas esparsas, onde uma tabela, ou matriz é substituída por um sistema de listas ligadas e deques (*double ended queue*) que são listas que permitem o acesso a ambas as extremidades para a inserção e remoção de elementos, podendo ser implementadas como listas duplamente encadeadas [Drozdek 2005].

2.2.4 Pilhas

São estruturas de dados lineares que só podem ser acessadas por uma de suas extremidades para armazenar e recuperar dados [Drozdek 2005]. Essa extremidade é chamada de topo e corresponde àquela em que se situa o último elemento que foi inserido. Por isso, essa estrutura é do tipo LIFO (*last in/first out*), visto que só pode ser removido o elemento que estiver nessa extremidade [Drozdek 2005].

As duas operações básicas que ocorrem em uma estrutura de dados do tipo pilha são, do inglês, *push* e *pop*. A primeira é a operação de empilhar, que insere um elemento no topo da pilha e a segunda operação é a de desempilhar, que remove o elemento que está no topo [Celes, Cerqueira e Rangel 2004]. A sua implementação pode ser feita tanto de forma estática, por meio de vetores, quanto dinâmica, usando listas encadeadas.

2.2.5 Filas

Assim como as listas e as pilhas, estruturas do tipo fila possuem duas extremidades, mas ao contrário das pilhas, as duas extremidades são utilizadas. O que diferencia uma fila de um lista é o fato de uma extremidade ser usada apenas para remoção, enquanto a outra é usada apenas para inserção. Consequentemente, o último elemento só poderá ser removido quando todos os elementos que o precedem na fila sejam removidos [Drozdek 2005]. São consideradas estruturas do tipo FIFO (*first in/first out*), visto que o elemento a ser removido será necessariamente aquele que estiver há mais tempo na fila.

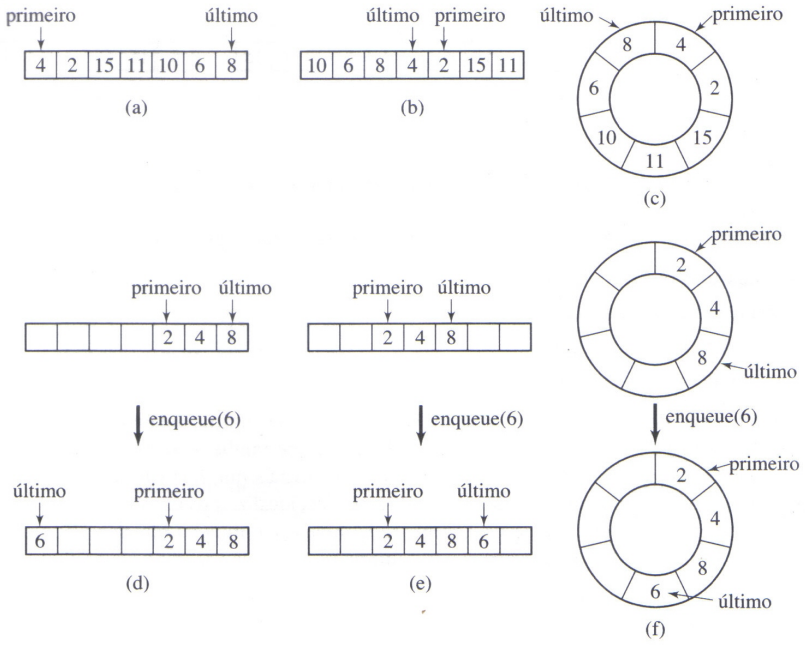
As implementações de filas podem ser feitas tanto de forma estática quanto dinâmica. Na forma estática, pode ser implementada como uma matriz unidimensional, onde os elementos inicialmente são adicionados no final dessa estrutura, mas podem ser removidos do início, liberando células da matriz que, consequentemente são usadas para enfileirar novos elementos. Sendo assim, o final da fila pode ocorrer no início da matriz. Essa estrutura pode melhor ser visualizada como uma matriz circular, como é mostrado na figura 2.3, onde é representada duas possíveis configurações de uma fila cheia (a) e (b), não cheias, com a inserção do elemento 6 (d) e (e) e suas respectivas visualizações circulares (c) e (f) [Drozdek 2005].

De forma dinâmica, uma fila pode ser implementada em forma de lista encadeada, com duas referências ao primeiro e ao último elemento da fila, como é mostrado na figura 2.4, onde esses referências são chamadas respectivamente de *ini* e *fim* [Celes, Cerqueira e Rangel 2004].

2.2.6 Árvores

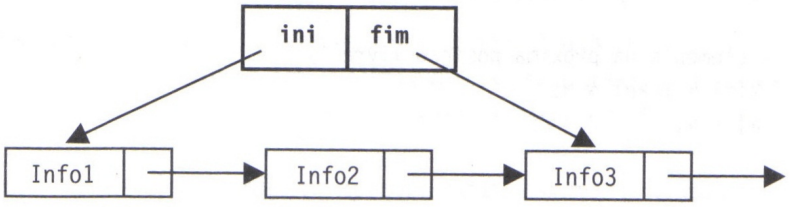
As estruturas anteriores, embora tenham importância fundamental em diversas formas de estruturação de dados utilizadas na resolução de vários problemas da computação, não são adequadas quando é necessário representar os dados dispostos de forma hierárquica [Celes, Cerqueira e Rangel 2004]. As árvores são as estruturas de dados adequadas para a representação hierárquica dos dados e são compostas por um conjunto de nós onde um desses nós é chamado de nó raiz que contém zero ou mais subárvores cujas raízes estão diretamente ligadas a ele, como é mostrado na figura 2.5.

Figura 2.3: Configurações de uma fila implementada como matriz



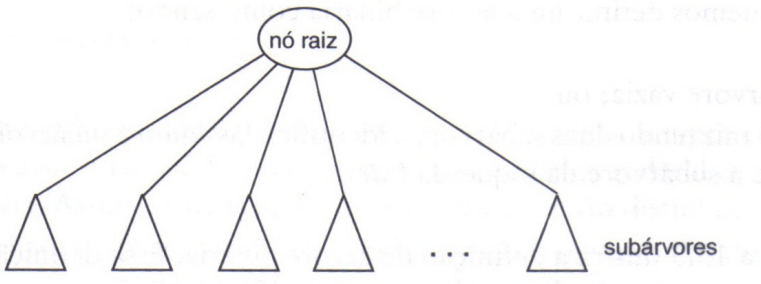
Fonte: [Drozdek 2005]

Figura 2.4: Fila implementada em forma de lista encadeada



Fonte: [Celes, Cerqueira e Rangel 2004]

Figura 2.5: Estrutura de uma árvore



Fonte: [Celes, Cerqueira e Rangel 2004]

Existem várias formas de implementação de árvores, e uma das mais utilizadas é a árvore binária [Celes, Cerqueira e Rangel 2004], onde cada nó possui zero, um ou dois filhos, ou seja, pode ser uma árvore vazia ou um nó raiz contendo duas subárvores. Um exemplo de utilização dessa árvore está na avaliação de expressões matemáticas . As árvores binárias de busca [Drozdek 2005] são bastante úteis em processos de localização de um determinado elemento. Com elas, uma busca pode ser feita em uma eficiência de *O(lg n)* a *O(n)*. Elas têm a seguinte propriedade: para cada nó *n* da árvore, todos os valores armazenados em sua subárvore da esquerda são menores que o valor *v* armazenado em *n*, e todos os valores armazenados na subárvore da direita são maiores que *v*.

Ainda existem outros tipos de árvores, de acordo com [Drozdek 2005] como árvores AVL, na qual as alturas das subárvores da esquerda e da direita de cada nó diferem no máximo por um. Isso é conseguido por meio do balanceamento, o que limita a busca em *O(lg n)*, mesmo no pior caso. Ainda há as árvores B que são utilizadas para aumentar a eficiência do acesso à informação na memória secundária do computador, como em aplicações de banco de dados e possuem variantes, como as árvores B* e B+.

2.3 Linguagens de Programação

Em teoria da computação, segundo definição de [Sipser 2007], uma linguagem é um conjunto formado por cadeias, que são sequências finitas de símbolos sobre um alfabeto. Ainda segundo [Sipser 2007], as gramáticas livres-de-contexto correspondem a métodos poderosos de descrição de linguagens, incluindo as linguagens de programação. A aplicação dessas gramáticas ocorre na especificação e compilação das linguagens de programação. Essas gramáticas são formadas por um conjunto de regras de produção que determinam como as linguagens, chamadas de linguagens livres-de-contexto são formadas. Analisadores sintáticos de compiladores e interpretadores usam essas gramáticas para extrair o significado de um programa antes de gerar o código compilado ou realizar a execução interpretada.

Em programação de computadores, uma linguagem de programação serve como uma forma de comunicação entre a pessoa com um problema e o computador usado para resolvê-lo [Tremblay e Sorenson 1985]. As linguagens de programação, assim sendo, correspondem

a um conjunto válido de sentenças [Parr 2010]. De acordo com [Tremblay e Sorenson 1985], computadores digitais somente aceitam e compreendem as suas próprias linguagens. Essas linguagens são consideradas de baixo-nível porque consistem em longas sequências de zeros e uns que são geralmente incompreensíveis para os seres humanos. A hierarquia das linguagens de programação, em ordem crescente de dependência da máquina, conforme [Tremblay e Sorenson 1985], está enumerada a seguir:

1. Linguagens de Máquina: É o nível mais baixo de uma linguagem computacional. Cada instrução em um programa nessa linguagem é representada por um código numérico, com endereços numéricos para referenciar localizações de memória.

2. Linguagem Assembly: É uma representação simbólica de uma linguagem de máquina, onde para cada operação é dado um código simbólico como ADD para adição e MUL para multiplicação, como também pode ocorrer com endereços de memória.

3. Linguagem de alto nível: Linguagens de alto nível fornecem um conjunto de características de linguagem mais rico do que as anteriores, como estruturas de controle, comandos aninhados, blocos e procedimentos. Apesar disso, muitos meios de acesso à caraterísticas em nível de sistema não estão disponíveis como nas anteriores. Alguns exemplos de linguagens de alto nível são Fortran, Pascal, C [Oliveira 2008] e Java [Deitel e Deitel 2010].

4. Linguagens orientadas a problemas específicos: São voltadas para problemas com aplicações específicas em uma determinada área, como SQL para banco de dados e COGO para aplicações de engenharia civil.

Dessa forma, quando um programador utiliza uma linguagem de alto nível, é necessário alguma aplicação que traduza seu código em alto nível para o código em linguagem de máquina. Sendo assim, conforme [Tremblay e Sorenson 1985], um tradutor é responsável por realizar essa tarefa, convertendo uma entrada de um programa fonte, em alto nível, para um programa objeto ou programa alvo. Dois tipos de tradutores são os compiladores e os interpretadores.

2.3.1 Tradutores

Nesta seção, para ficar mais claro sobre o que é um Interpretador, foi feita uma comparação com um tipo muito conhecido de tradutor, chamado de compilador. Com isso, pretende-se fazer uma explicação sobre interpretadores juntamente com os compiladores, já que esses últimos são tipos muitos usados de tradutores. Um compilador [Aho, Sethi e Ullman 1995] é um programa que lê um programa escrito numa linguagem, chamada de linguagem fonte e o traduz num programa escrito em outra linguagem, chamada de linguagem alvo, relatando aos seus usuários a presença de erros durante o processo.

O processo de compilação é feito em duas partes, de acordo com [Aho, Sethi e Ullman 1995]: a análise, que divide o programa fonte em suas partes constituintes e cria uma representação intermediária do mesmo e a síntese, que constrói o programa desejado a partir da representação intermediária. O processo de análise é dividido em três fases: análise léxica, sintática e semântica.

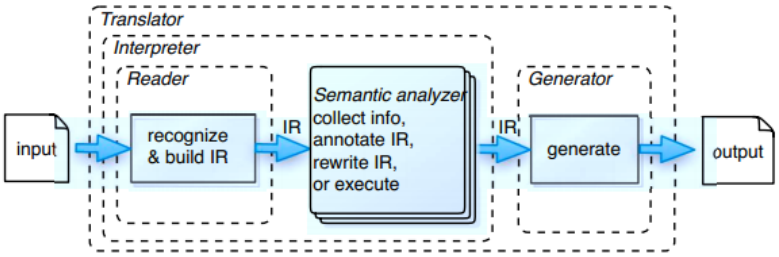
Na análise léxica, o fluxo de caracteres que constituem um programa de entrada é lido da esquerda para a direita e agrupado em *tokens*, que são sequências de caracteres que possuem um significado coletivo. A análise sintática é responsável por agrupar esses *tokens* hierarquicamente em frases gramaticais, usualmente representadas por uma árvore gramatical. Já na análise semântica, verificações semânticas são realizadas para se assegurar que os componentes de um programa se combinam de forma significativa [Aho, Sethi e Ullman 1995].

No processo de síntese [Aho, Sethi e Ullman 1995], a partir do analisador semântico pode haver a geração de um código intermediário, para facilitar a tradução no programa alvo, bem como otimizações nesse código para que o código de máquina seja mais rápido em tempo de execução. Depois dessas fases, é feita a geração do código alvo, que normalmente consiste de código de máquina relocável ou código de montagem.

Toda aplicação que manipula código fonte em uma determinada linguagem, como um tradutor, é dividida em componentes que se combinam entre si em um *pipeline* multiestágios

que analisa ou manipula um *stream* de entrada e converte gradualmente um conjunto válido de sentenças nessa entrada em uma estrutura de dados interna manipulável ou as traduzem em sentenças de outra linguagem [Parr 2010]. Na figura 2.6, é mostrada uma representação desse *pipeline*.

Figura 2.6: Pipeline multiestágios de um tradutor



Fonte: [Parr 2010]

O leitor reconhece a entrada, a partir dos analisadores léxicos e sintáticos, e constrói uma representação intermediária (IR - *Intermediate Representation*) a partir dela, que alimenta o resto da aplicação [Parr 2010]. Com essa representação intermediária, o analisador semântico efetua a análise semântica. Nessa análise, é feita a verificação semântica com identificação de erros e a definição de símbolos. O gerador percorre essa estrutura e emite uma saída. A diferença entre um compilador e um interpretador é sutil, o interpretador lê, decodifica e executa as instruções, não produzindo, portanto, um programa de saída [Aho, Sethi e Ullman 1995]. Um interpretador é, dessa forma, um tipo de tradutor que não emite uma saída com o código traduzido, diferentemente dos compiladores, a saída que ele emite é o resultado da execução desse código traduzido. É essencialmente um programa que executa outros programas, simulando um processador *hardware* em *software*. Exemplos de linguagens interpretadas: Java, Ruby e Python [Parr 2010].

2.3.2 Orientação a Objetos

A orientação a objetos (OO) consiste em um paradigma de programação usado no projeto e implementação do desenvolvimento de um software. De acordo com esse paradigma, o software é modelado em termos semelhantes àqueles que as pessoas utilizam para descrever

objetos do mundo real e a implementação é feita com linguagens de programação orientadas a objeto [Deitel e Deitel 2010]. O modo de programar é abstraído de acordo com entidades do mundo real, não apenas em procedimentos algorítmicos [Guedes 2006].

Dessa forma, é tirado proveito de relacionamentos de classe, onde numa analogia com o mundo real, objetos de uma classe de veículos como carros, caminhões e patins possuem muitas características em comum. Também é tirado proveito de relacionamentos de herança nos quais as classes de objetos são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessa mesma classe. Como por exemplo, uma classe "conversível"possui características da classe mais geral "automóvel", mas neste caso, o capô sobe e desce [Deitel e Deitel 2010].

No mundo da programação OO, vários termos e componentes são usados. De acordo com [Guedes 2006], alguns deles são:

- Classes: fornecem uma classificação para os objetos, onde objetos da mesma classe devem possuir os mesmos atributos e métodos;
- Objetos: São as instâncias de uma classe que ocupam memória no computador, com valores em seus atributos;
- Atributos: São as características dos objetos, identificados por um nome e um tipo de dado;
- Métodos: Assim como os atributos, são componentes de uma classe, consistem em atividades que podem ser executas por objetos e são formados por um conjunto de instruções. Podem ou não receber parâmetros, que valores utilizados para iniciar a execução de um método;
- Herança: Permite o reaproveitamento de atributos e métodos comuns entre classes. A classe herdada é chamada de superclasse e a herdeira de subclasse;
- Polimorfismo: Ocorre em consequência da herança, onde é permitida a atribuição a um objeto de uma superclasse de todos os objetos de suas subclasses e a invocação de seus métodos durante a execução de um programa.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados trabalhos relacionados tanto à ferramenta IGED como um todo como também especificamente ao Interpretador abordado neste trabalho. Além disso, para cada trabalho apresentado é feita uma discussão com a intenção de compará-lo à ferramenta IGED e ao Interpretador do IGED. Com isso, são mostradas vantagens e desvantagens deste trabalho com os trabalhos analisados para que as contribuições do Interpretador para o IGED e as contribuições do IGED como ferramenta de ensino com esse Interpretador possam ficar bem definidas. No final, é feita uma comparação do IGED com os trabalhos relacionados por meio de uma tabela com algumas características úteis ao ensino e aprendizagem.

3.1 Trabalhos Relacionados ao IGED

Alguns trabalhos disponíveis na web não permitem que o usuário defina seus próprios algoritmos. Nessas ferramentas, os algoritmos em linguagem de programação ou em pseudocódigo já estão cadastradas no sistema, tendo o aluno a tarefa de acompanhar a execução dos algoritmos com determinadas visualizações desses algoritmos e as suas estruturas de dados. Essa abordagem, de acordo com o processo de construção e aquisição do conhecimento discutido no capítulo 1, torna-se menos efetiva da que é empregada neste trabalho, onde os alunos podem construir seus próprios algoritmos, o que gera muito mais possibilidades de aprendizagem através deles, pois eles podem, com isso, visualizar uma quantidade imensa de algoritmos com as características possibilitadas com a adição do Interpretador desenvolvido neste trabalho ao IGED. Em outros trabalhos, ocorre a necessidade de mais de um usuário

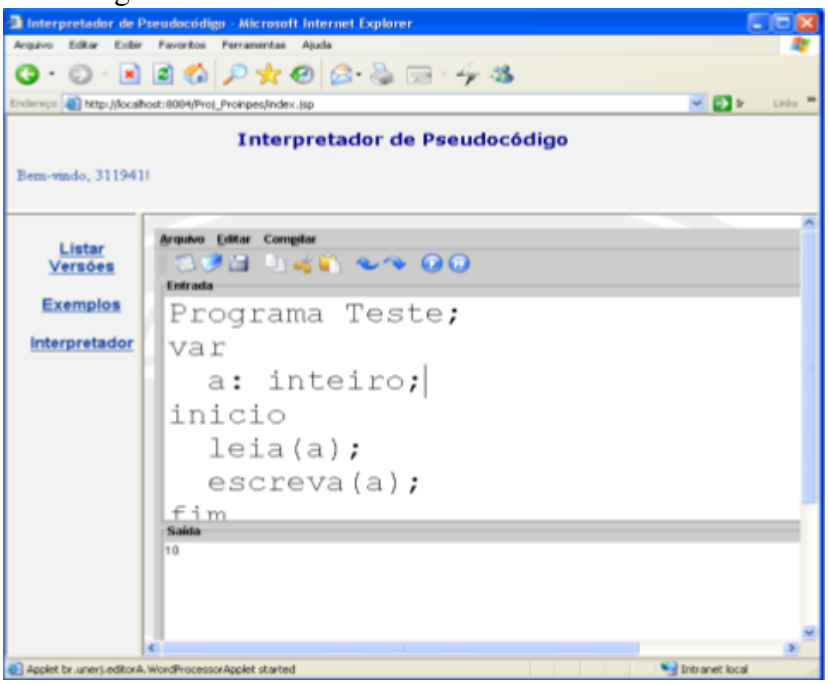
para animar um algoritmo, ficando o aluno dependente de outro usuário para visualizar as execuções de seus algoritmos, o que não é uma boa característica no ensino e aprendizagem.

3.1.1 WEB-UNERJOL

Uma dessas ferramentas é o WEB-UNERJOL [Ferradin e Stephani 2005], que por meio da web permite a visualização de algoritmos na pseudolinguagem Portugol, constituindo-se como um ambiente de programação a distância por meio da Internet. Esse ambiente pode ser utilizado tanto por professores quanto por alunos, onde os primeiros podem cadastrarar e excluir alunos e visualizar as execuções dos algoritmos cadastrados pelos alunos.

Na figura 3.1, é ilustrada a interface do aluno nessa ferramenta, onde o mesmo tem acesso a um compilador programado através de um Java Applet, atalhos para exemplos de códigos e para uma listagem de suas versões anteriores.

Figura 3.1: Interface do aluno no WEB-UNERJOL



Fonte: [Ferradin e Stephani 2005]

Discussão

Para essa ferramenta, não foi demonstrada as visualizações das estruturas de dados, ao contrário do IGED que permite visualizações para as estruturas de lista, vetor e árvore. Outra diferença é que nessa ferramenta, os códigos são executados na linguagem Portugol que é util no ensino de programação de uma forma geral, pois não está atrelada a nenhuma linguagem de programação específica, facilitando no entendimento dos conceitos gerais de programação. No entanto, o uso de uma linguagem de programação específica facilita no emprego dos conceitos aprendidos com o uso de uma linguagem na qual os programas podem ser desenvolvidos de fato. Por exemplo, em Portugol não é possível definir classes e instanciar objetos.

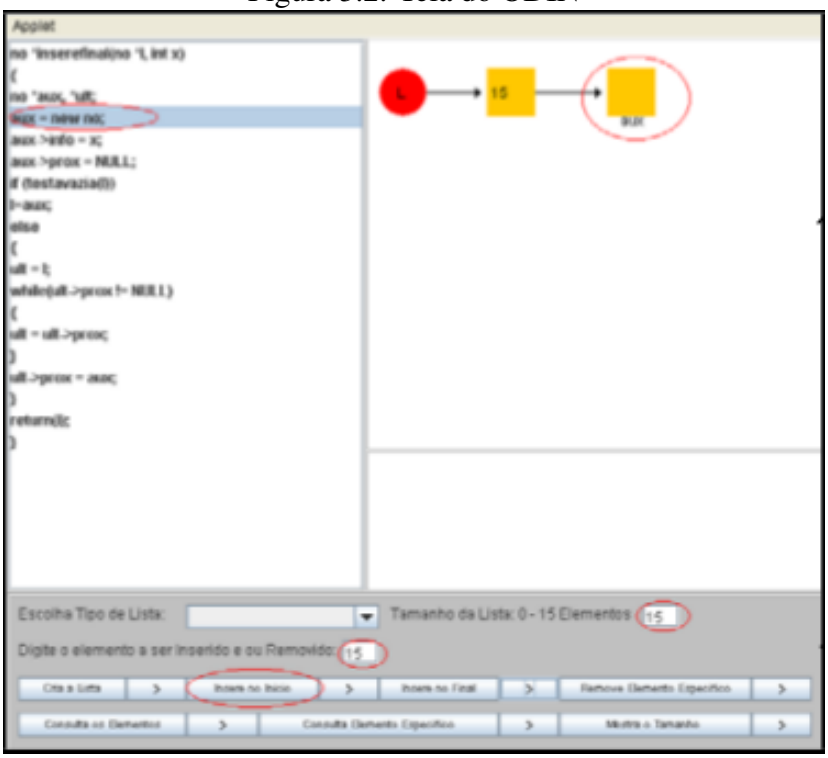
3.1.2 ODIN

O ODIN [Madeira et al. 2012] é um ambiente web de apoio ao ensino de lista encadeada por meio da visualização gráfica das operações primitivas de lista encadeada com algoritmos codificados na linguagem C++. A interface do ODIN é composta pelo espaço virtual, que é um *web site*, e um aplicativo que há dentro dele e que simula o funcionamento de uma lista encadeada. O módulo do ODIN que é responsável pela demonstração gráfica das funções primitivas de lista encadeada é composto por três partes: a parte de contém o código fonte, a parte onde a lista é exibida graficamente e um menu interativo que possui vários botões por meio dos quais os usuários executam o código e interagem com o ambiente. A tela desse módulo está ilustrada na Figura 3.2.

Discussão

Nessa ferramenta só é possível a visualização de um tipo de estrutura de dados. No IGED, atualmente, três tipos de estruturas de dados podem ser visualizadas. Como a ferramenta não possui um interpretador, é o usuário quem controla a execução do programa por meio dos botões dos menus interativos. Isso por um lado pode ser útil, já que ao ter total controle sobre a execução de um programa, um aluno poderá ter mais tempo, quanto tempo ele quiser para visualizar uma detereminada estrutura de dados, mas a impossiblidade da execução automática do código é uma grande desvantagem, principalamente quando se trata de algoritmos

Figura 3.2: Tela do ODIN



Fonte: [Madeira et al. 2012]

com muitos passos de execução ou com grande complexidade. Além disso, sem um tradutor (compilador ou interpretador) o usuário não pode definir seus próprios algoritmos.

3.1.3 Spyke

O Spyke [Simões et al. 2013] consiste em um ambiente voltando para o ensino de pilhas e filas. Essa ferramenta se baseia em um tutorial a partir do qual é demonstrado, para essas estruturas, as formas de alocação de memória estática e dinâmica e a evolução gráfica durante a execução das funções primitivas nas linguagens Pascal, C++ e Java. Nessa ferramenta, há diversas telas onde é possível a criação de vários formulários que possibilitam visualizações gráficas de pilhas e filas estáticas e dinâmicas e a interação do usuário na criação, inserção, remoção e consulta de elementos. Além disso, há a possibilidade de realizar comparações do TAD (Tipo Abstrato de Dado) entre as diferentes linguagens de programação.

A tela principal do sistema é dividida em quatro partes: os quadros de pilha estática, pilha dinâmica, fila estática e fila dinâmica. Em cada um desses quadros, há a possibilidade da visualização da execução dos algoritmos nas três linguagens. A Figura 3.3 ilustra a representação de uma pilha estática com quatro funções primitivas, onde no lado esquerdo da tela há botões por meio dos quais o usuário pode acompanhar a sequência de ações com as operções disponíveis sobre a estrutura.

Figura 3.3: Pilha estática no Spyke

Fonte: [Simões et al. 2013]

Discussão

Essa ferramenta permite visualizações de dois tipos de estruturas que o IGED até o momento não permite. O IGED lida com três estruturas que não são suportadas pelo Spyke. Um ponto positivo dessa ferramenta é o de permitir a visualição das estruturas tanto na forma estática quanto na dinâmica, outro ponto positivo é a possibilidade da execução de algoritmos em três linguagens de programação diferentes que são bastante utilizadas em disciplinas de Estruturas de Dados. Como ocorre na ferramenta anterior, o Spyke não possui um interpretador para executar de forma automática os códigos nessas linguagens, sem a interferência de um usuário. Assim, a ferramenta consiste em um tutorial, onde as operações e a ordem da execução delas são determinadas pela ferramenta e a quantidade de algoritmos que podem ser executados pela ferramenta é, dessa forma, bastante limitada.

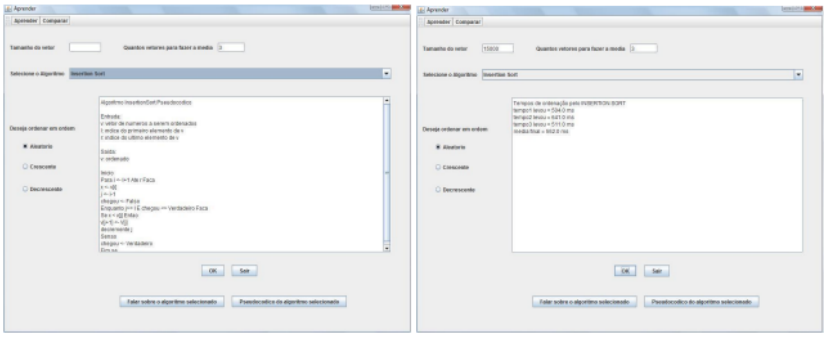
3.1.4 SEED

Em [Veras et al. 2010] é apresentado o SEED que consiste em uma ferramenta que provê ao aluno meios de estudo e comparação dos principais algoritmos de ordenção de estruturas de dados. Dessa forma, ele é dividido em dois módulos: um de aprendizagem e um de comparação. No módulo de aprendizagem o usuário escolhe um dos métodos de ordenação e visualiza o princípio do método, o pseudocódigo e a análise de complexidade. Além disso, ele pode executar o algoritmo, visualizar o tempo de execução e a quantidade de comparações realizadas. A figura 3.4 ilustra o módulo de aprendizado do SEED, onde na tela do lado esquerdo, são mostradas visualizações das informações de um algoritmo e na do lado direito, o resultado da execução desse algoritmo. As operações de ordenação permitidas pelo SEED são as seguintes:

- Ordenação por Inserção (Insertion Sort);
- Ordenação por Seleção (Selection Sort);
- Ordenação pelo método da bolha (Bubble Sort);
- Ordenação por Intercalação (Merge Sort);
- Ordenação por montes (Heap Sort);

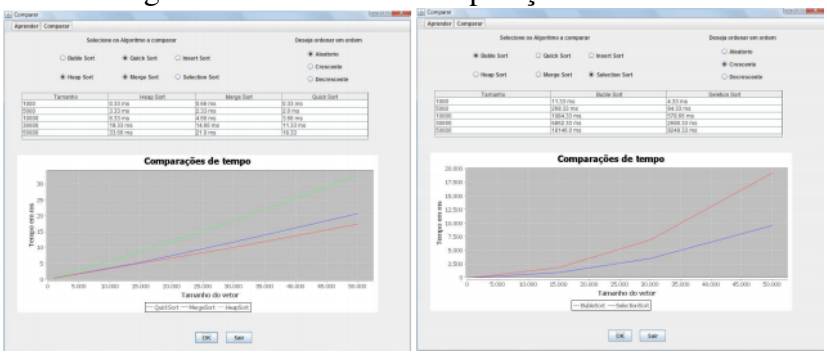
- Ordenação rápida (Quick Sort).

Figura 3.4: Módulo de aprendizagem do SEED



Fonte: [Veras et al. 2010]

Figura 3.5: Módulo de comparações do SEED



Fonte: [Veras et al. 2010]

No módulo de comparação, o usuário pode avaliar o desempenho de dois ou mais métodos de ordenação, selecionando quais ele deseja comparar para uma determinada disposição dos dados. O SEED compara os métodos para entradas de tamanhos de 1000, 5000, 10000, 20000 e 30000 elementos. A tela desse módulo é ilustrada na figura 3.5.

Discussão

Quando não se tem um interpretador para executar código, a quantidade de operações fica limitada, como pode ser observado no SEED. No IGED, com o uso do interpretador desenvolvido neste trabalho, a quantidade de algoritmos que podem ser executados é ilimitada.

Entretanto, o SEED tem uma funcionalidade útil ao ensino ainda não disponível no IGED, que é a comparação dos algoritmos na forma visual de gráficos.

3.1.5 TBC-AED

O TBC-AED (Treinamento Baseado em Computador, Algoritmos e Estruturas de Dados) [Santos e Costa 2005] é um software de visualização gráfica de algoritmos e estruturas de dados. Essa ferramenta possui como diferencial algumas características como *links* explicativos, conteúdo teórico, legendas explicativas que ilustram as etapas da apresentação dos algoritmos. O TBC-AED apresenta os seguintes temas: Busca Binária, Métodos de Ordenação Selection Sort, Insertion Sort, Bubble Sort, Merge Sort e Quick Sort, Alocação Estática e Dinâmica de Memória de Lista, Fila e Pilha e Árvore Binária de Busca.

Os algoritmos são mostrados em Portugol. As figuras 3.6 e 3.7 correspondem respectivamente as telas de operação de busca binária e de árvore binária. Nessas telas, ao passar o mouse por cima das áreas, os usuários visualizam uma breve descrição sobre a região posicionada. A execução do algoritmo é controlada pelo usuário através dos botões situados na parte inferior da tela.

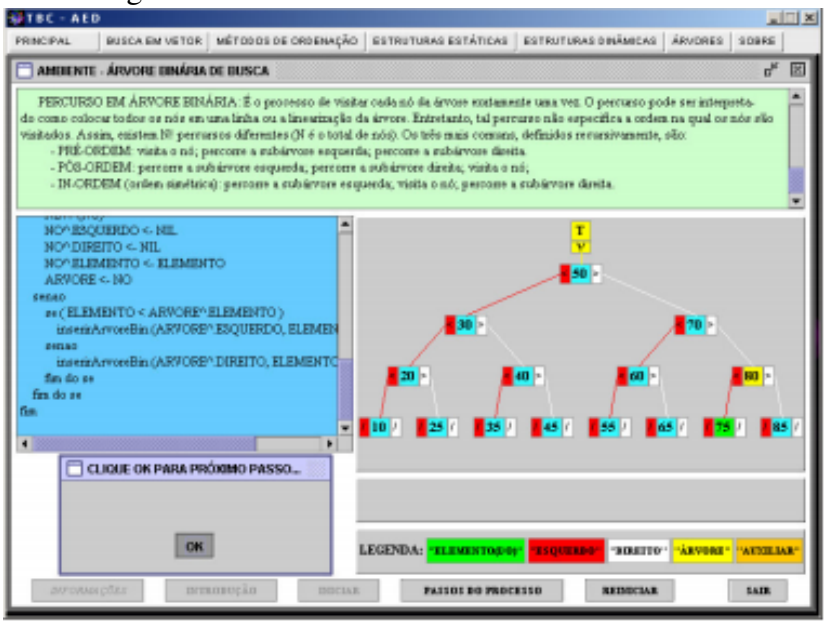
Discussão

Um grande diferencial do TBC-AED em relação ao IGED e aos demais trabalhos relacionados é a grande quantidade de informação apresentada pela ferramenta durante as execuções dos algoritmos. Além disso, essa ferramenta permite a visualização de mais estruturas de dados que o IGED. No entanto, como as demais ferramentas apresentadas anteriormente, a quantidade de algoritmos que podem ser executados e animados é limitada, já que eles estão previamente cadastrados no sistema e a execução deles é realizada pelo usuário. No IGED, o usuário não tem esse trabalho porque quem executa os algoritmos é o interpretador, podendo ele visualizar a execução de uma quantidade ilimitada de algoritmos.

Figura 3.6: Tela de Busca Binária do TBC-AED

Fonte: [Santos e Costa 2005]

Figura 3.7: Tela de Árvore Binária do TBC-AED



Fonte: [Santos e Costa 2005]

3.1.6 Balsa

O Balsa [Brown e Sedgewick 1984] é um ambiente a partir do qual é possível visualizar a execução de algoritmos em alto nível de forma dinâmica mediante a observação da manipulação das estruturas de dados decorrente dessa execução. Começou a ser utilizado por estudantes em Setembro de 1983 e foi aplicado no ensino de cursos de introdução a programação, algoritmos, estruturas de dados, equações diferenciais e linguagem assembly na Brown University.

Há três tipos de usuários desse ambiente: o *scriptwriter*, que prepara material para outros usuários. Outro tipo de usuário é o designer de algoritmos, ou seja, o programador que irá implementar os algoritmos que terão suas execuções apresentadas de forma gráfica e dinâmica. Ele pode utilizar implementações prévias e não precisa se importar com detalhes da apresentação gráfica de seus algoritmos. O animador é quem projeta e implementa os programas que irão exibir graficamente os programas em execução de interesse. Dessa forma, há dois tipos de programas envolvidos na apresentação da execução de um algoritmo: um que contém a implementação do algoritmo cuja execução será apresentada e outro que contém os comandos da apresentação gráfica da execução desse programa.

Algumas funcionalidades que o Balsa provê a seus usuários incluem o possibilidade de criar, redimensionar e posicionar as janelas da execução dos algoritmos, que contém "janelas de visão", ou seja, janelas com múltiplas visões da execução de um algoritmo, como mostrado na figura 3.10 do seu sucessor, o Balsa II. Como exemplo, a construção de uma árvore 2D de uma busca em extensão apresenta a visão da árvore, subdivisões planares induzidas pela árvore e um histórico dessas subdivisões depois que cada ponto é inserido. O usuário também poderá ver janelas contendo o código em execução. Além disso, é permitido ao usuário iniciar, parar, executar mais devagar, executar o algoritmo "de trás para frente", bem como adicionar *breakpoints* e realizar *stepping*, para depuração do código. O algoritmo pode ser executado novamente, visto que após sua execução, o histórico dessa execução é salvo e múltiplos algoritmos podem ser executados simultaneamente.

Outras funcionalidades incluem o fato de o usuário poder salvar ou restaurar configurações de apresentação das janelas ou *scripts* com essas operações, configurações de execução de um algoritmo ou da exibição do histórico desse, que são definidas pelo *scriptwriter* e podem ser carregadas durante a execução de um algoritmo. Funcionalidades como exibição em formato impresso e comunicação com outros usuários do Balsa também são suportadas.

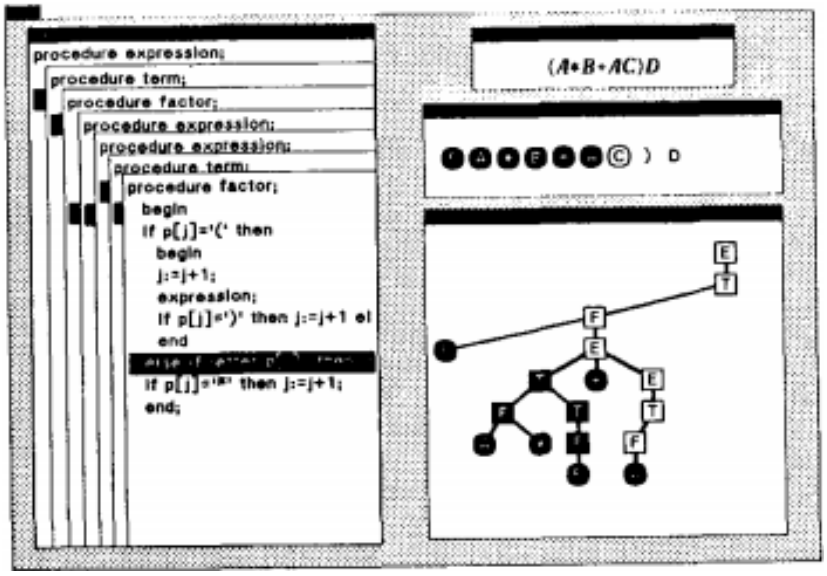
A execução de um algoritmo por meio de uma animação se dá após o projetista do algoritmo implementá-lo e passar informações ao animador, que compreendem os eventos de interesse que serão lançados durante a execução do algoritmo, o que resultará na atualização gráfica das animações. O Balsa IE Manager é responsável por receber esses eventos e executar os procedimentos de animação. Há também as VDSMs que consistem em um grupo de rotinas de atualizações gráficas que são compartilhadas entre múltiplas visões ou janelas de exibição das animações.

As figuras seguintes ilustram o funcionamento do Balsa para um processador de expressões regulares e as saídas gráficas para o reconhecimento dessas expressões definidas em um programa escrito em Pascal. A figura 3.8 contém as saídas gráficas processadas por um compilador descendente recursivo e na figura 3.9 é mostrado o funcionamento de um autômato finito não-determinístico que determina se um texto em uma *string* pode ser gerado por uma expressão regular.

Discussão

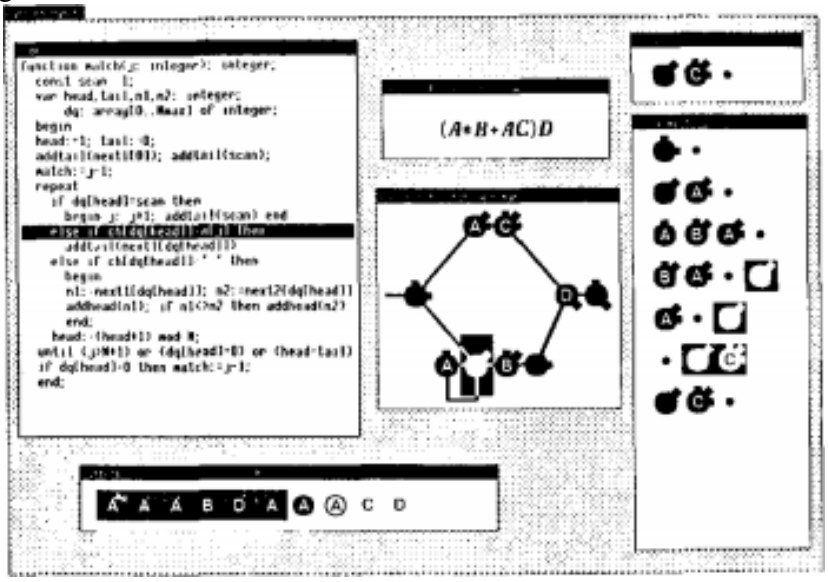
Uma grande vantagem do Balsa é o de suportar a execução de algoritmos com operações distintas, além da manipulação das estruturas de dados, podendo ser utilizado em outras disciplinas do ensino de computação. Por outro lado, a execução da animação de um algoritmo não se torna uma tarefa simples e trivial por parte do usuário, já que necessita de pelo menos dois (um para desenvolver o algoritmo e outro para realizar os procedimentos de animação). Apesar disso, o Balsa é apresentado como um sistema bastante completo e com várias utilidades para o acompanhamento e visualização da execução de um algoritmo que o IGED não possui, incluindo funcionalidades que permite o usuário acompanhar a execução do algoritmo por meio de vários modos como pausa, "de trás para frente", *stepping* e adição de

Figura 3.8: Compilador de expressões regulares



Fonte: [Brown e Sedgewick 1984]

Figura 3.9: Funcionamento de um autômato finito não-determinístico



Fonte: [Brown e Sedgewick 1984]

breakpoints, que são bastante úteis na depuração de um código, bem como a apresentação da animação de um algoritmo em múltiplas "visões". Ao contrário do IGED, o Balsa não dispõe de funcionalidades que permitam o cadastro de exercícios e meios para que um aluno possa verificar se sua solução está correta.

3.1.7 Balsa II

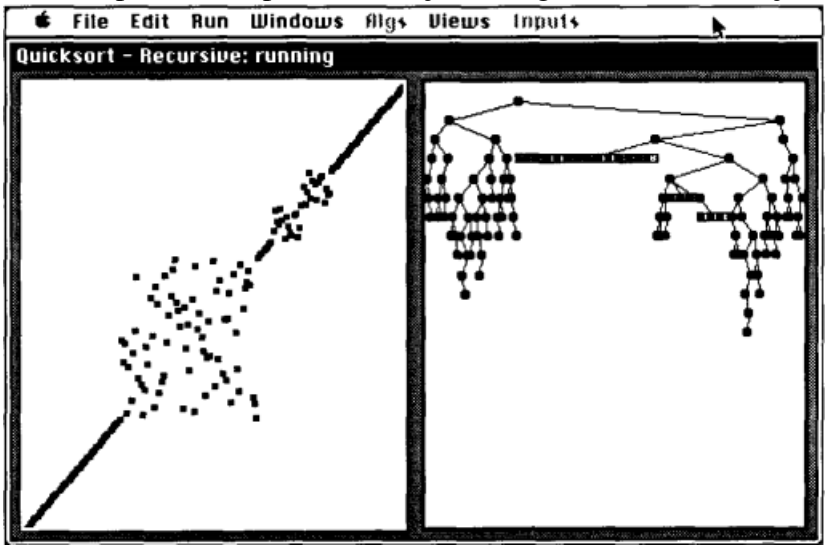
Com o sucessor do Balsa, o Balsa II [Brown 1988], foram adicionadas melhorias principalmente na interface, incluindo a parte de E/S com o qual o usuário interage com o sistema. O Balsa II opera sobre a plataforma Apple Macintosh, com os algoritmos de animação definidos pela linguagem Pascal. De forma similar ao seu antecessor, para animar um algoritmo há três componentes básicos a serem definidos: o algoritmo em si, geradores de entrada que proveem os dados a serem manipulados pelo algoritmo e diversas visões que apresentam a execução do algoritmo de forma animada.

As janelas da interface podem ser dispostas de múltiplas formas de acordo com a preferência do usuário e junto com elas há a janela do algoritmo, onde ao clicar duplamente com o mouse, são exibidas informações de há quanto tempo o algoritmo está sendo executado. Nas outras janelas, com essa mesma ação, são permitidas várias operações de janela, como redimensionamento, dentre outras.

Com o uso de parâmetros de visão, podem ser controlados vários atributos de como as informações são exibidas em uma janela particular. Como o Balsa II provê somente um *framework*, a natureza dos parâmetros de cada janela depende de como foram implementados pelos programadores. Esses parâmetros podem especificar, por exemplo, se os dados a serem ordenados por um algoritmo de ordenação devem ser apresentados como pontos, quadrados ou círculos. O Balsa II também provê controle sobre como os algoritmos devem ser sincronizados.

A forma como um algoritmo é executado pode ser determinada pelo usuário através da especificação dos dados de entrada e de parâmetros. Os parâmetros dos algoritmos afetam o próprio algoritmo e não os dados que ele manipula. Como por exemplo, a utilização desses

Figura 3.10: Múltiplas visões para a execução do algoritmo de ordenação Quicksort



Fonte: [Brown 1988]

parâmetros pode determinar se o analisador léxico de um compilador a ser animado deve usar uma tabela *hash* de um determinado tamanho ou uma árvore binária. Geradores de entrada proveem os dados que um algoritmo deve processar e através de parâmetros desses geradores, os tipos dos dados que são gerados podem ser definidos, como por exemplo, o quanto os dados de um arquivo de entrada estão ordenados para um algoritmo de ordenação. Informações especificadas pelo usuário no tempo de execução de um algoritmo também são permitidas, como por exemplo, em um algoritmo que percorre um grafo, o usuário pode escolher por qual vértice o algoritmo deve começar ou qual nó deve ser excluído em uma árvore binária. Essas informações podem ser passadas ao digitar o nome de um desses elementos ou selecionando-o através do clique do mouse.

Em cada evento anotado em um algoritmo, pode ser determinado se ele deve ser um ponto de *stop* ou de *step*, como são utilizados nos depuradores convencionais. Dessa forma, um algoritmo pode ser executado em uma de quatro formas:

- Go: pára no próximo ponto de *stop*;
- GoGo: pausa nos pontos de *stop*;
- Step: para no próximo ponto de *step*;

- StepStep: pausa nos pontos de *step*.

A performance de um algoritmo pode ser medida ao contar o número de ocorrências de determinados eventos de interesse, que correspondem na prática a operações de interesse desses algoritmos, podendo ser especificados custos para eles, em forma de quanto tempo o interpretador levou para executá-los. Do ponto de vista do programador, a construção de um algoritmo envolve dois passos: anotação, onde há a inserção de marcadores de eventos de interesse e a criação de pontos que o Balsa II irá invocá-los em resposta a ações do usuário.

Discussão

Da mesma forma que o seu antecessor, o Balsa II possui diversas funcionalidades que não são encontradas no IGED para a execução dos algoritmos e sua visualização, o que inclui o fato de o usuário poder verificar o custo de tempo de execução dos seus algoritmos, determinar como um algoritmo deve ser executado através de parâmetros, como os dados devem estar dispostos, bem como informações em tempo de execução que irão permitir que ele visualize a execução das estruturas na melhor forma que lhe convém. Uma desvantagem do Balsa II em relação o IGED é o fato de operar em apenas uma plataforma, que atualmente é proprietária e comercial. Além disso, embora tenha introduzido novos recursos, o Balsa II possui as mesmas desvantagens mencionadas para o seu antecessor.

3.1.8 Zeus

Baseado nas experiências adquiridas com o Balsa e o Balsa II, foi desenvolvido o sistema de animação de algoritmos Zeus [Brown 1991], que além da animação de algoritmos, também possui como funcionalidade um editor para a construção de interfaces gráficas. Os códigos dos algoritmos são definidos através da linguagem Modula-2.

O Zeus pode mostrar algumas janelas de exibição (visões) de forma automática, baseado no conjunto de eventos que o algoritmo gera. Essas janelas são a Janela de Transcrição, o Painel de Controle e a Janela de Barras. A Janela de Transcrição contém a exibição de expressões simbólicas dos eventos gerados pelas execuções dos algoritmos. O Painel de Controle contém botões correspondentes a cada evento, onde ao clicar nesses botões, os

eventos são gerados de acordo com parâmetros fornecidos pelos usuários. Os dados que estão sendo manipulados também podem ser apresentados em uma fileira de barras pelo outro tipo de janela, a Janela de Barras, como no caso de um algoritmo de ordenação.

O pré-processador do Zeus, chamado Zume, lê um arquivo de eventos como o da figura 3.11, e gera definições do algoritmo e classes de visão, bem como classes utilitárias de visão e eventos que são disparados entre os algoritmos e as visões. O arquivo de eventos contém as palavras-chave de definição ALGDATA, para os dados e seus tipos, e os procedimentos que gerarão os eventos que podem ser definidos com as palavras-chave OUTPUT, quando fluir do algoritmo para a visão e FEEDBACK, quando fluir da visão para o algoritmo. Esse último ocorre como resposta a ações do usuário. Como exemplo, uma ilustração de um arquivo de eventos para o algoritmo de ordenação sequencial:

Figura 3.11: Arquivo de eventos de um algoritmo de ordenação sequencial

```
EVENTS Sort;
ALGDATA
  a: ARRAY [1..100] OF Key;
  N: CARDINAL;
OUTPUT Init (N: CARDINAL);
OUTPUT SetVal (i: CARDINAL; old: Key);
OUTPUT SwapElts (i, j: CARDINAL);
FEEDBACK ChangeVal(i: CARDINAL; new: Key);
```

Fonte: [Brown 1991]

Um algoritmo é anotado com chamadas a essas rotinas e sua execução se dá após o usuário ter selecionado o algoritmo e os dados a serem processados através de uma janela da interface do Painel de Controle do Zeus. Para as janelas são gerados métodos para salvar e restaurar os dados que são fornecidos pela interação do usuário com essas janelas.

Para os animadores dos algoritmos, na criação de uma visão, o Zeus os permitem demonstrar como uma instância de um objeto usada pela visão deve ser apresentada e possui procedimentos para interpolar mudanças desses objetos ao longo do tempo. As visões também podem ser criadas por meio da reutilização de comportamentos delas através de herança,

já que as próprias visões são objetos, como o salvamento de estados. Outra forma de criar as visões é programando-as diretamente.

Para o Zeus, foi implementado o FormsEdit, um editor de criação de interfaces. Para isso, existem três tipos de janelas: um editor gráfico e um textual, onde alterações em uma delas gerarão alterações correspondentes na outra e vice-versa. A outra é a janela resultante, atualizada com a edição das duas janelas anteriores, que mostra como a interface irá aparecer em tempo de execução, como reagirá a eventos do mouse e do teclado e ao redimensionamento.

Discussão

O Zeus, assim como os anteriores, não possui uma integração maior da execução dos algoritmos com a sua visualização, de forma que essa visualização seja feita automaticamente. Embora forneça facilidades tanto no desenvolvimento das animações quanto na criação de interfaces dessas visualizações, o que não ocorre no IGED, ainda é necessário que essas visualizações sejam definidas por um animador. Um usuário aprendiz que esteja desenvolvendo um algoritmo deverá anotar o seu código com chamadas às operações que gerarão os eventos de animação, ficando dependente de um animador que defina como serão essas operações de visualização.

3.1.9 Tango

O Tango [Stasko 1990] se baseia na animação de algoritmos em um *framework* que segue o paradigma de transição por caminhos na criação de animações, uma forma simples e consistente de definição de mudanças graduais ou transições nas janelas de animação, permitindo que a interpolação entre estados de animação seja feita de forma mais fácil pelos designers da animação. O programa que está sendo animado não é executado com o Tango propriamente dito, assim qualquer programa pode ser animado apenas ao adicionar operações de algoritmos, sem que o Tango seja recompilado repetidamente. As rotinas de animação são ativadas pelas operações de um algoritmo de um determinado programa através de um mapeamento externo.

Esse *framework* contém três componentes: o componente dos algoritmos baseado em eventos, como no Balsa, com os algoritmos sendo anotados com chamadas de procedimentos, chamados operações . Essas chamadas serão realizadas em determinado ponto da execução de um algoritmo. Outro, chamado componente de animação, contém os objetos gráficos que mudam a localização, o tamanho e a cor através dos *frames* de uma animação e as operações que controlam as mudanças para simular uma animação. Os objetos responsáveis por essas operações são agrupados nos seguintes tipos:

- Imagens: Podem ser de dois tipos: as primárias que incluem linhas, retângulos, círculos e textos e as composições de imagens, que são coleções das imagens primárias com relacionamentos geométricos entre elas, para que os usuários criem instâncias mais completas de objetos;

- Localizações: Uma posição identificada por um par de coordenadas (x,y). Comumente denotam variáveis com as imagens sendo seus valores;

- Caminhos: Designa a magnitude de mudanças nos atributos de uma imagem de um frame para o próximo, sendo formalmente definidos como uma sequência ordenada finita de pares de coordenadas (x,y), onde cada par designa um deslocamento relativo de uma posição prévia. Além desses deslocamentos, também modifica a cor, o tamanho e contém um componente de intervalo de tempo que ajuda a controlar a "suavidade"da animação. Há três tipos de caminhos: linear, em sentido horário e anti-horário. Além disso, os caminhos podem ser preenchidos, truncados, interpolados, rotacionados, iterados, compostos e concatenados através das operações.

- Transições: As transições usam os parâmetros dos Caminhos para modificar a posição de uma imagem ou sua aparência, resultando em uma ação de animação. Uma transição típica é definida pelo tipo da transição, a imagem a ser alterada e o parâmetro que define o Caminho. Seus tipos podem ser de movimentação, redimensionamento, alteração de cor, preenchimento, elevação, abaixamento, atraso e alteração de visibilidade e composição. Esta última é a mais interessante porque permite que várias transições sejam executadas concorrentemente e provê uma forma natural de animação em que mais de um objeto mudam suas características.

O último componente é o de Mapeamento que compreende o mapeamento de um programa e sua execução para a animação. Uma parte desse componente, chamada de associação permite que os designers façam a conexão de um objeto de dados como uma imagem, localização ao valor de dado com os parâmetros recebidos do programa que contém o algoritmo. Uma implementação típica usa *hashing*, onde a chave é o nome dessa associação e uma lista de parâmetros. Essas associações são definidas e referenciadas nas cenas de animação. O mapeamento das operações de um algoritmo para cenas da animação é feito de acordo com um modelo de estados finitos, implementado no *framework*, o que permite que o mapeamento seja feito das seguintes formas: um-para-um, um-para-muitos, muitos-para-um e muitos-para-muitos.

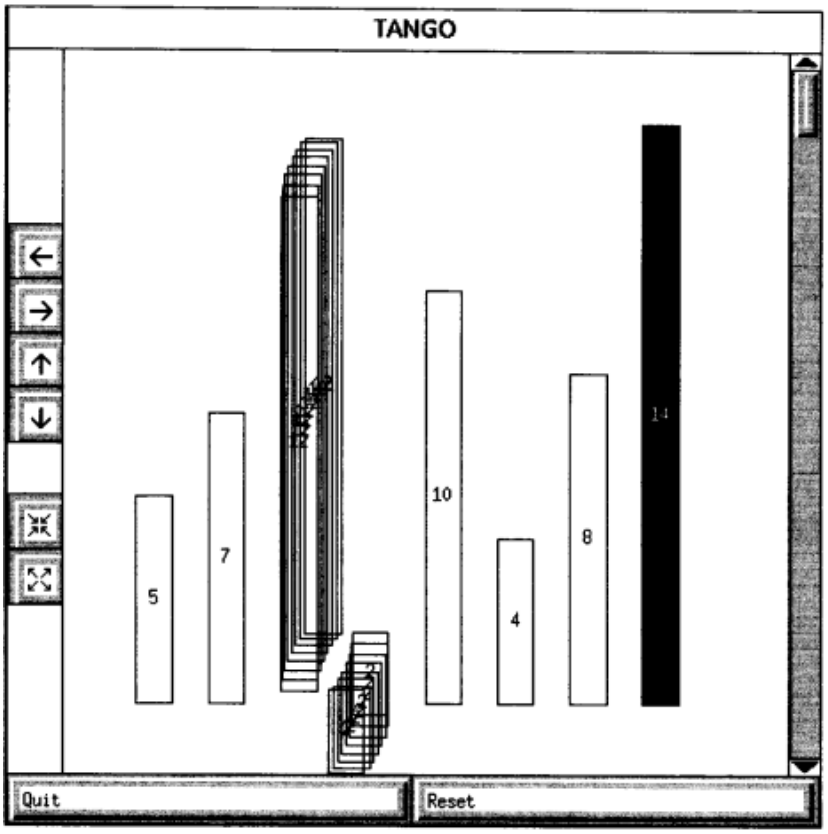
Dessa forma, o sistema Tango é baseado nesse *framework* e organizado de acordo com os seguintes componentes: o programa a ser animado, o código que controla a animação e o executável do Tango propriamente dito. Há dois processos pelos quais a animação é executada por meio da plataforma Unix. No processo de animação o Tango é iniciado, onde o sistema lê um arquivo de controle onde são especificados parâmetros para a animação, carrega dinamicamente os arquivos de descrição da animação que contém o código de definição da animação feita pelo designer, cria uma janela onde será mostrada a animação e espera pelo programa a ser animado. No outro processo, chamado de processo do algoritmo, é onde o usuário executa o programa a ser animado, que contém o algoritmo e as operações que serão distribuídas ao processo da animação assim que forem atingidos durante a execução.

O Tango também permite o uso do mouse nas janelas de visão, com a seleção arbitrária de coordenadas de janela e objetos de imagem para que informações da cena sejam formatadas em *string* e passadas por parâmetros aos procedimentos que irão passar essas informações ao processo do algoritmo. Como por exemplo, em um algoritmo de animação para encontrar o caminho mais curto em um grafo, o usuário pode selecionar os vértices inicial e final usados pelo algoritmo.

De forma geral, baseado nos três componentes do *framework* mencionados, para produzir a animação, o animador deve seguir os seguintes passos: anotar o programa com as operações necessárias, projetar as cenas de animação para implementar ações de animação, onde há um

pacote de rotinas chamado Twist (*Tango's wonderful image-synthesis toolkit*) por meio do qual podem ser criadas linhas, colunas e grids de localizações e imagens, bem como grafos e árvores binárias, e criar um arquivo de controle especificando o mapeamento das operações do algoritmo às cenas de animação. Na figura 3.12 é ilustrada a animação do algoritmo bubblesort no Tango.

Figura 3.12: Animação do algoritmo bubblesort no Tango



Fonte: [Stasko 1990]

Discussão

O Tango apresenta semelhanças com os trabalhos anteriores em relação à forma como um algoritmo é animado. A característica que mais se destaca em relação aos demais, no entanto, é o fato de o usuário determinar como os objetos gráficos sofrerão alterações de acordo com vários tipos de transição ao longo da animação. Ainda é necessário um usuário que defina as animações, um que programe o algoritmo e outro que faça o mapeamento entre o código do algoritmo e as operações de animação gráfica, enquanto que no IGED só é necessário que o

usuário informe o algoritmo a ser animado. O Tango possui mais opções de transição para uma determinada animação, visto que é o usuário quem define como serão essas transições. Deixar que o usuário defina isso não é uma vantagem em reação ao IGED, visto que a tarefa de desenvolver a animação torna-se mais complexa, enquanto que no IGED ela ocorre de forma transparente a qualquer usuário, sem que ele tenha que se preocupar como isso ocorre.

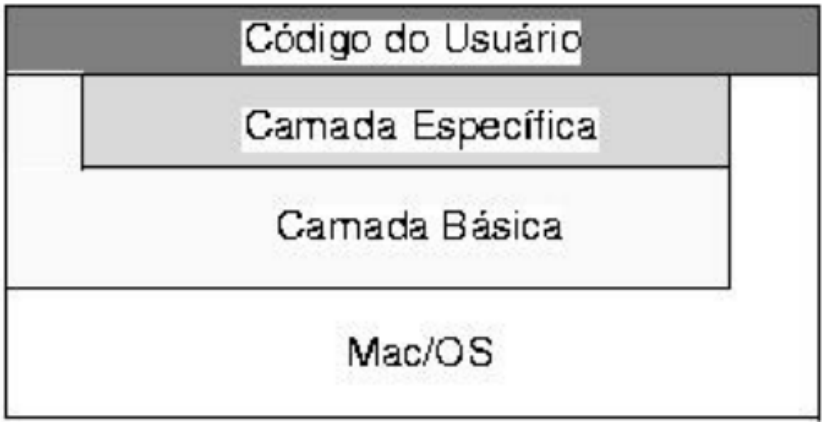
3.1.10 Astral

Nesse ambiente, conforme descrito em [Garcia e Rezende 1997], o estudante realiza suas próprias animações e tem acesso a aplicativos exemplos providos de animações gráficas que ilustram o funcionamento dos algoritmos, visando atender as necessidades de preparação de diversos exercícios de implementação de estruturas de dados utilizando animações gráficas. O ambiente Astral foi desenvolvido para a arquitetura Macintosh e com a linguagem Pascal para a implementação dos algoritmos.

A arquitetura do ambiente Astral é dividida em camadas que estão organizadas conforme a figura 3.13. A primeira delas é a camada básica que agrupa rotinas de interesse geral para a construção das animações, envolvendo manipulação de janelas de texto, a visualização e a comunicação com o usuário da aplicação, o tratamento de eventos e o gerenciamento de objetos gráficos. A camada específica é construída para cada animação e contém o código cuja execução mostrará de forma animada a manipulação das estruturas de dados. Esse algoritmo é escrito utilizando as interfaces fornecidas nas camadas inferiores. O uso dessas interfaces resulta na chamada de operações gráficas, como no exemplo apresentado em [Garcia e Rezende 1997], uma única chamada à função DrawTree é suficiente para a animação das alterações de uma árvore AVL devidas a uma inserção, incluindo os efeitos de rotação.

O gerenciador de eventos filtra eventos gerados pelo usuário, como seleção de itens de menu ou entradas de dados. Esses eventos são passados para a camada específica que irá ativar o código do usuário. Além disso, o Astral conta com uma camada de apoio. Essa camada contém recursos que facilitam a implementação do algoritmo pelo usuário e o auxilia no entendimento do assunto relacionado. Esses recursos incluem os seguintes itens: um arquivo-texto explicativo e um aplicativo exemplo, o ambiente de programação Think

Figura 3.13: Arquitetura do Ambiente Astral



Fonte: [Garcia e Rezende 1997]

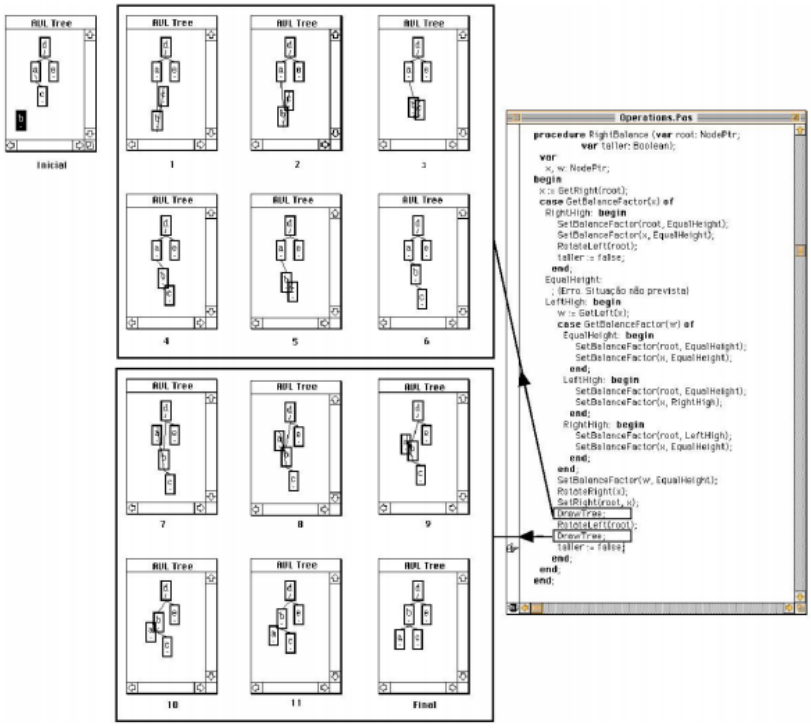
Pascal da Symantec [1], por meio do qual é permitida a execução passo a passo, e adição de *breakpoints* e de *watchpoints*, além de outras facilidades para a depuração do código. Na camada de apoio também há um arquivo texto com as interfaces das rotinas de visualização que são usadas para a realização das animações pelo estudante. Na figura 3.14 é ilustrada a animação de uma inserção em árvore AVL e na figura 3.15 são mostradas as saídas de três algoritmos de ordenação: *QuickSort*(a), *MergeSort*(b) e *HeapSort*(c).

Discussão

O ambiente Astral, ao contrário das ferramentas analisadas anteriormente possui as rotinas de animação gráficas pré-definidas, facilitando para o usuário aprendiz o processo de desenvolver algoritmos com animação, visto que não há necessidade de outro usuário animador. Contudo, ainda é necessário que ele faça chamadas a rotinas de animação gráfica no próprio algoritmo, o que não ocorre no IGED. Isso dificulta no processo de aprendizagem, visto que o aluno deverá saber quais funções ele deve chamar e em que parte do algoritmo isso deverá ocorrer, quando ele só deveria se empenhar em desenvolver o algoritmo em si. Em relação ao processo pedagógico, a camada de apoio do Astral mostra-se bastante útil, mas ainda não possui recursos multimídia, como é proposto para o IGED. Outra desvantagem é o fato de operar sobre uma única plataforma de SO, que é comercial e proprietária.

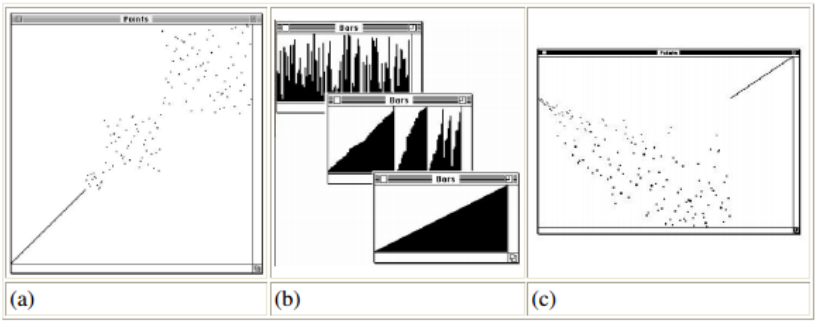
[1] http://www.symantec.com

Figura 3.14: Inserção em árvore AVL no Astral



Fonte: [Garcia e Rezende 1997]

Figura 3.15: Animação de algoritmos de ordenação no Astral



Fonte: [Garcia e Rezende 1997]

3.1.11 AnimA

Em [Amorim e Resende 1993], o AnimA é apresentado em fase final de implementação como um meio de auxiliar no processo de animações de algoritmos, provendo os usuários de facilidades para produções de animações de qualquer algoritmo de maneira sistemática. Em relação a outros trabalhos relacionados, o AnimA possui como diferencial a reusabilidade de componentes, que ao serem implementados ficam disponíveis em bibliotecas do ambiente. Além disso, permite a execução concorrente de animações. Esse ambiente é executado em estações de trabalho SparcStation sob Sun/OS, com a filosofia de interface gráfica OpenLook, biblioteca gráfica XView e os algoritmos são escritos na linguagem C++.

Há dois níveis de usuários no AnimA: o programador, que programa as animações e as incluem no ambiente e o usuário que configura, executa e interage com as animações de acordo com a sua necessidade. O programador deve implementar seu programa subdividindo-o em três módulos: o algoritmo em si, gerador de entradas e os visualizadores.

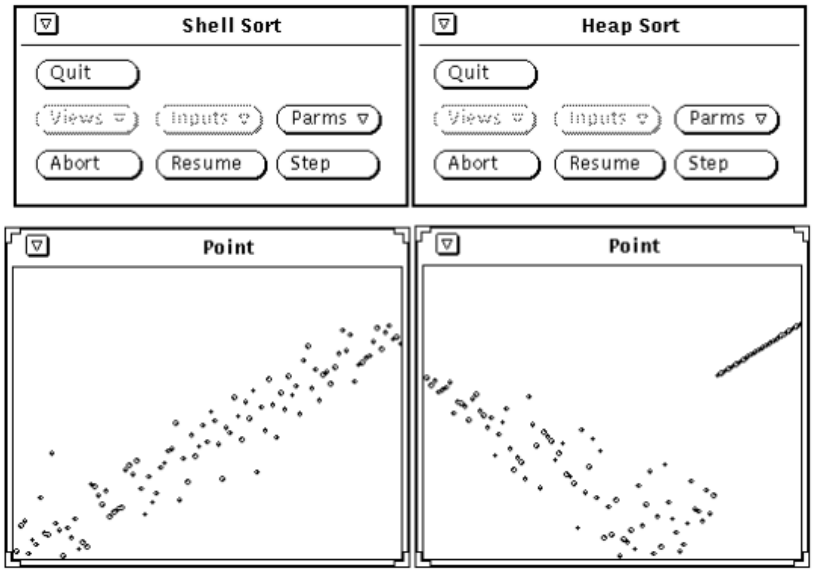
O algoritmo é estruturado pela subdivisão em métodos com funções específicas de inicialização, término, tratamento de parâmetros e do código do algoritmo propriamente dito. Os parâmetros servem para a interação do usuário final com as animações. As atualizações das visualizações das animações são feitas de acordo com eventos. As chamadas desses eventos são especificadas no código do algoritmo e a cada evento está associado um método específico de tratamento em um gerador de entradas ou em um visualizador.

O gerador de entradas é quem fornece os dados para o algoritmo e pode tanto produzir o dado quanto solicitá-lo ao usuário final. Já os visualizadores produzem os efeitos visuais com base nos eventos de saída do algoritmo, sendo responsáveis por atualizarem *displays* para refletir esses eventos durante a execução do algoritmo e também por mapearem coordenadas da tela para informações consistentes, no caso do gerador de entradas solicitar dados ao usuário final. Para um visualizador, no AnimA é utilizada uma biblioteca de objetos gráficos elementares, denominada EGOLib (*Elementary Graphic Objects Library*), implementada em C++. Nessa biblioteca, está implementado um conjunto de objetos gráficos com atributos e métodos próprios. Para a implementação dos visualizadores, é necessário conhe-

cimentos básicos de computação gráfica, mas ao serem implementados, os componentes dos visualizadores ficam disponíveis no ambiente como bibliotecas.

O AnimA provê além de uma poderosa interface com o usuário final, uma interface gráfica amigável para manutenção automática das bibliotecas do ambiente, por parte do programador, incluindo as opções de inclusão, alteração e exclusão de problemas, repertórios de eventos e de mensagens, algoritmos, geradores de entrada e visualizadores. O usuário final realiza duas etapas no controle da execução da animação: configuração, quando ele adapta a tela conforme as suas necessidades ou preferências, escolhe o problema, o algoritmo, o gerador de entrada e visualizadores, que podem ser mais de um para um gerador de entrada, e informa valores de parâmetros, caso necessário. A outra fase é a de execução, quando ele executa o algoritmo, tendo acesso às operações de interrupção temporária, execução passo a passo e encerramento da execução. Na figura 3.16, é mostrada a animação de dois algoritmos de ordenação ShellSort e HeapSort, com um mesmo gerador de entradas, já que o AnimA permite mais de um visualizador para um gerador de entradas.

Figura 3.16: Animação da execução de dois algoritmos de ordenação com a mesma entrada de dados no AnimA



Fonte: [Amorim e Resende 1993]

Discussão

Uma característica interessante do AnimA, não encontrada no IGED, é a possibilidade da execução concorrente de animações, o que permite mostrar como um algoritmo opera para mais de um conjunto de dados de entrada de forma comparativa. Geradores de entrada também não são encontrados no IGED, sendo assim uma desvantagem em reação ao AnimA e a outras ferramentas que possuem. No AnimA é necessário um usuário que programe as animações e que, além disso, tenha conhecimentos de computação gráfica. O fato dessas operações de animação poderem estar disponíveis em bibliotecas e serem reutilizadas constituem por um lado uma vantagem, já que múltiplos usuários podem enriquecer a ferramenta com objetos gráficos para a visualização. Por outro lado, ocorre uma desvantagem no sentido de que, por exemplo, um programador de algoritmos ficará dependendo de que haja uma determinada operação para uma visualização. Dessa forma, ele pode não encontrar uma operação de visualização que lhe seja necessária. Outra desvantagem é a dependência de plataforma.

3.2 Trabalhos Relacionados ao Interpretador do IGED

Nesta seção, são apresentadas ferramentas cujos modelos de execução são mais semelhantes ao IGED com o Interpretador apresentado neste trabalho. Isso porque as execuções dos algoritmos nessas ferramentas são feitas com o uso de interpretadores. Além disso, essas ferramentas são executadas em *desktop* e juntas permitem a visualização de diversas estruturas de dados.

3.2.1 jGRASP

O jGRASP [Cross et al. 2009] possui três formas de interação por meio das quais ocorrem as visualizações dinâmicas das estruturas de dados: *debugger*, *workbench*, e interações textuais. O *structure indentifier*, componente do jGRASP, identifica e renderiza as visualizações abstratas das estruturas de dados mais comuns, como pilhas, filas, listas encadeadas, árvores binárias, *heaps* e tabelas *hash*, quando uma janela de visualização da animação é aberta para uma das três formas de interação.

A forma mais comum de interação é através do *debugger*, onde o usuário posiciona um *breakpoint* em uma expressão próxima a criação da estrutura de dados e executa o programa no modo *debug*. O programa é então executado no modo de *step* após a parada nesse *breakpoint*. Quando um objeto é criado, o usuário poderá arrastá-lo da aba de *debug* para que o visualizador da animação seja aberto e a estrutura que representa o objeto seja renderizada pelo *structure identifier*.

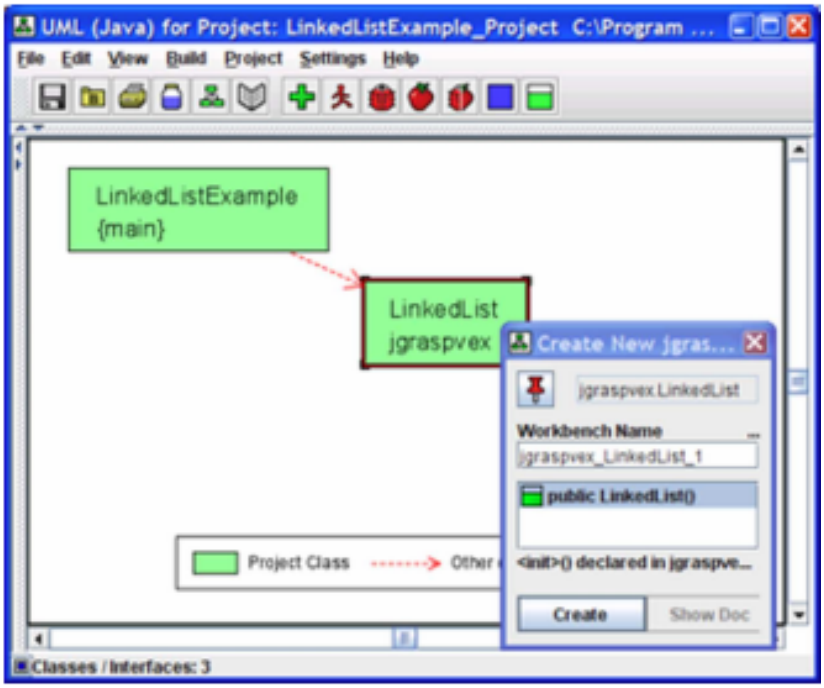
No *workbench*, objetos podem ser criados e posicionados e seus métodos podem ser invocados através de menus e botões em um diagrama de classe UML, como ilustrado na figura 3.17, onde é mostrada a criação de um objeto representando a estrutura de lista encadeada, e/ou através de janelas de edição de código-fonte. A visualização da alteração das estruturas de dados é atualizada a cada vez que o usuário invoca um método no objeto.

O mecanismo de *workbench* reúne classes e informações de métodos. A execução de métodos e a criação de instâncias nesse mecanismo são realizadas através do JDI (*Java Debugger Interface*). Este não permite a invocação de chamadas recursivas de métodos, nem na criação de instâncias.

Na terceira forma de interação, quando o usuário informa uma expressão em Java e pressiona um botão, essa expressão é imediatamente executada por meio de um interpretador, sem haver a necessidade da execução de um programa inteiro, apenas uma expressão é suficiente. A janela de visualização é atualizada à medida que as expressões são informadas na aba de interações. Quando uma dessas expressões for de criação de uma instância de classe, como por exemplo, na expressão: *LinkedList list = new LinkedList();*, o objeto criado é adicionado no *workbench* e um visualizador pode ser aberto ao arrastar esse objeto do *workbench* ou da aba de *debug*. Na figura 3.18, é mostrado do lado esquerdo, o código sendo parado em um laço *for* e três operações de inserção de elementos numa lista sendo executadas na aba de interações textuais e, do lado direito, detalhes de uma inserção em uma lista encadeada sendo exibidos na janela de visualização.

Nessa forma de interação, um interpretador Java efetua a análise sintática do código informado e constrói a árvore sintática. O código pode ser uma expressão ou uma sequência de

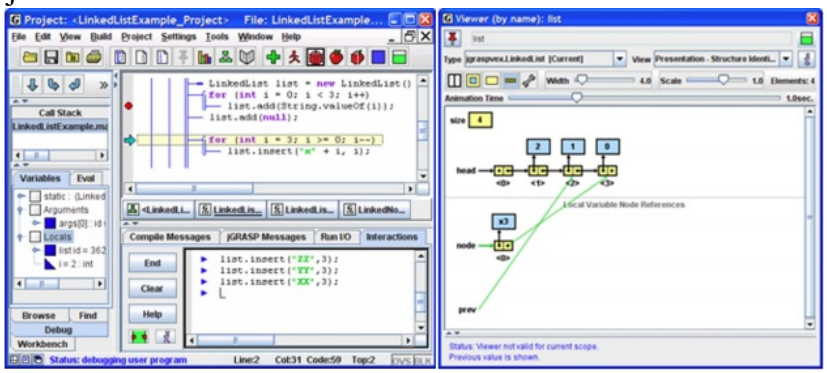
Figura 3.17: Criação de uma instância de lista encadeada por meio de um diagrama UML no jGRASP



Fonte: [Cross et al. 2009]

expressões. Esse interpretador, que só recebe expressões como entrada, é usado para avaliar argumentos que serão fornecidos para a invocação de métodos do *workbench*, expressões utilizadas pelos visualizadores de objetos, etc. A VM desse interpretador pode armazenar objetos e valores primitivos. Todas as avaliações de operadores são feitas pelo interpretador, mas este não permite definições de classes nem de interfaces.

Figura 3.18: Interações textuais e a visualização de detalhes de uma inserção em uma lista encadeada no jGRASP



Fonte: [Cross et al. 2009]

Discussão

O jGRASP possui algumas semelhanças com o IGED, que as ferramentas analisadas anteriormente não possuem: a fato dela ser executada em Java, portanto compatível com diversas plataformas, e também o fato das animações não precisarem ser previamente programadas. O jGRASP possui diversas funcionalidades não encontradas no IGED, o que inclui múltiplas formas de interação, uso de*breakpoints* e de visualização dos valores das variáveis, que são úteis na detecção e correção de erros de código. No entanto, possui sérias limitações. A primeira é a impossibilidade de executar métodos recursivos no *workbench*, o que inviabiliza a aprendizagem de conceitos importantes de programação, como a recursividade por meio dessa forma de interação. Outra limitação é o fato de classes e interfaces não poderem ser definidas nas interações textuais, dificultando na aprendizagem de orientação a objetos por meio dessa ferramenta e no emprego de técnicas de OO para a programação.

3.2.2 PCIL

Em [Malone et al. 2009] é apresentado o PCIL (*PseudoCode Interpreted Language*), uma linguagem de alto nível interpretada onde a visualização é incorporada à sua especificação. Cada elemento primitivo da linguagem, como uma variável, possui suporte a uma representação gráfica e o interpretador produz essas visualizações gráficas a partir da execução da implementação de um algoritmo.

As estruturas primitivas da linguagem, além de serem dos tipos inteiro e *string*, podem também ser grafos, listas, *priority queues* e dicionários. Ao invés de um programa ser executado todo de uma vez, o aluno é quem controla quando a execução irá avançar de uma linha do pseudocódigo, definindo pela linguagem, à próxima. Além disso, o estudante pode especificar valores iniciais para as variáveis dos algoritmos, responder a um questionamento sobre o comportamento futuro da execução de um algoritmo e receber um *feedback* sobre a corretude de sua resposta.

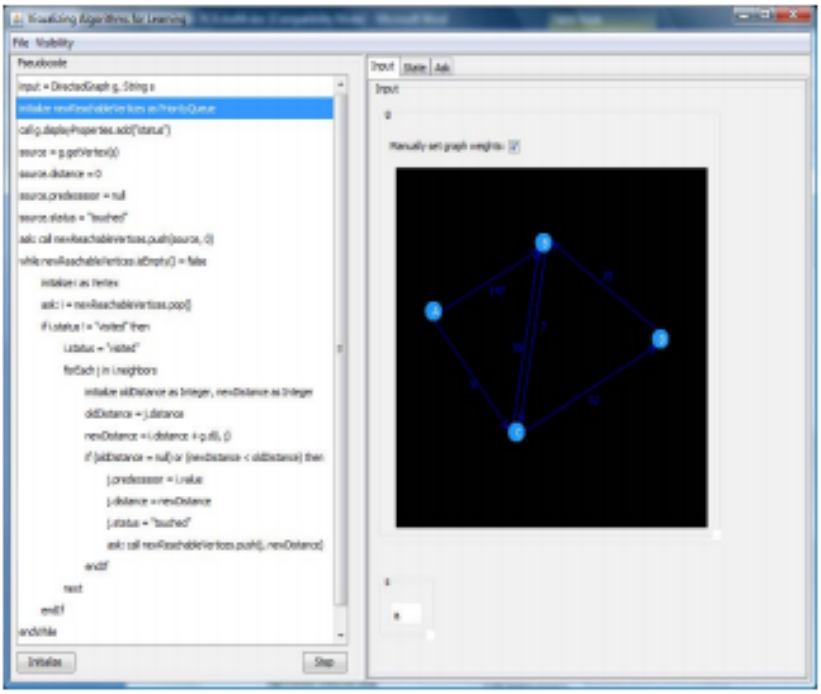
O interpretador recebe uma lista de funções e um *array* de *tokens* para serem processados e pausa a cada instrução que está sendo executada para que o estudante determine a execução do próximo passo. Esse interpretador foi implementado em Java, de acordo com a padrão de projeto MVC (*Model-View-Controller*), onde o controlador é quem contém o interpretador que interpreta o pseudocódigo e efetua as mudanças necessárias ao modelo, representado pela memória. Este informa à visão sobre a necessidade de atualização após a execução de uma instrução.

Cada classe que define um tipo de variável primitiva contém métodos que definem como essas variáveis serão renderizadas na tela, além de definirem também propriedades específicas do seu tipo. A figura 3.19 ilustra a visualização de um grafo da execução do algoritmo Dijkstra, a partir de um pseudocódigo.

Discussão

Uma vantagem do PCIL em relação o IGED é todas as variáveis podem ser renderizadas na tela, não apenas as estruturas de dados consideradas para um determinado problema,

Figura 3.19: Visualização do algoritmo Dijkstra no PCIL



Fonte: [Malone et al. 2009]

isso auxilia em tarefas de depuração dos programas. Além disso, o aluno pode controlar a execução do programa, o que também é uma vantagem visto que com isso é mais fácil para um aluno entender como funciona o código e para encontrar e corrigir erros. A proposta do IGED é mais eficiente no sentido pedagógico, já que o aluno pode verificar se o seu algoritmo como um todo está correto ou não, ou seja, se resolve ou não o problema e também é permitido a inserção de documentos multimídia para auxiliar os professores no ensino.

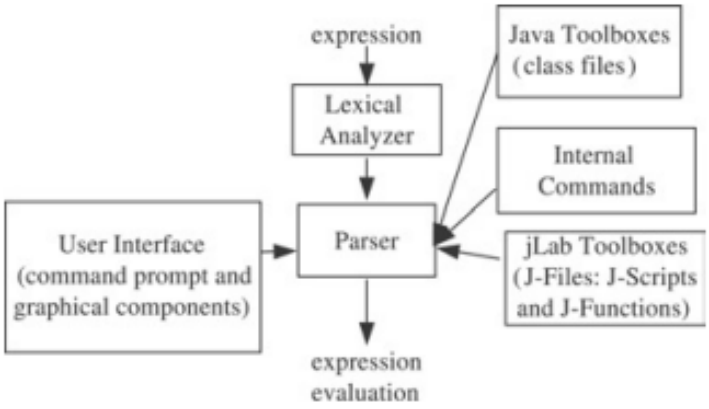
3.2.3 jLab

O ambiente de execução do jLab é apresentado em [Papadimitriou 2007], o qual possui uma linguagem similar às dos ambientes Matlab/Scilab executada por um interpretador implementado na linguagem Java. Umas das vantagens apresentadas desse ambiente em relação aos que foram implementados em C/C++ ou Fortran, como o Matlab, Sclilab ou Octave, é a independência de plataforma, além da compilação do código do jLab ser mais rápida e simples.

O jLab consiste em um ambiente de programação onde é possível carregar e executar classes Java de forma dinâmica, através dos arquivos binários de *bytecodes*, como também é possível executar arquivos chamados J-Files, que contém código que é interpretado pelo próprio sistema do jLab.

O interpretador do jLab, chamado de jExec, é formado por um analisador léxico e por um *parser*. A arquitetura do sistema é mostrada na figura 3.20 e na figura 3.21 é mostrado um *snapshot* desse ambiente. As *toolboxes* Java são as bibliotecas de classes Java a partir das quais os programadores dessa linguagem podem adicionar novas bibliotecas com diversas finalidades. As *toolboxes* do jLab contém os arquivos de código da linguagem interpretável do jLab, os J-Files, cuja sintaxe possui similaridade com a linguagem do Scilab, o que facilita a incorporação do repositório numérico do Scilab.

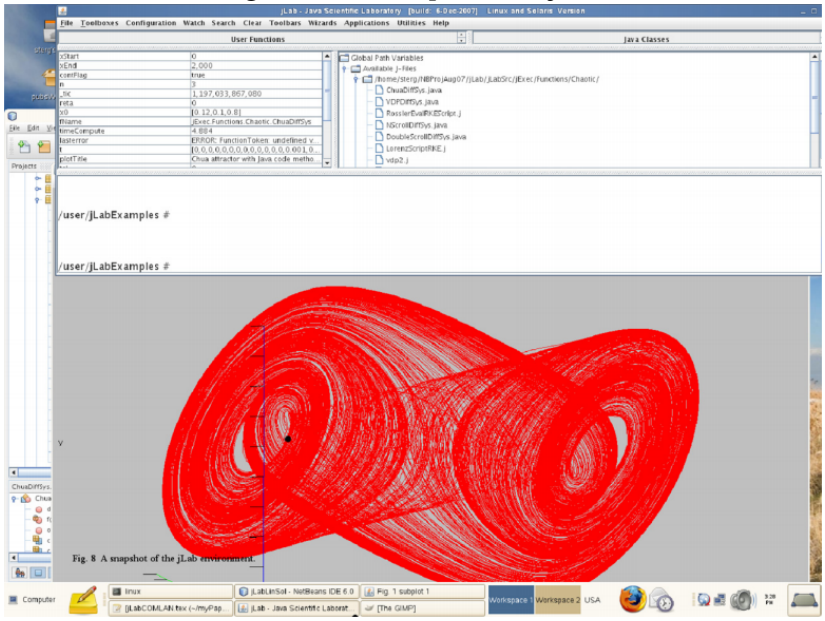
Figura 3.20: Arquitetura do jLab



Fonte: [Papadimitriou 2007]

No jLab, uma classe Java chamada *FunctionManager* possui a funcionalidade do gerenciamento das funções que estão nos arquivos de classes Java compilados, chamados de J-Classes, como também as que estão nos J-Files, chamadas de J-Functions. Os J-Files também podem conter J-Scripts, que são arquivos de código jLab em lotes, cuja execução é geralmente mais lenta do que as das funções do Scilab e do Matlab, mas se essas mesmas funções forem implementadas em J-Classes, executarão mais rápido.

Figura 3.21: *Snapshot* do jLab



Fonte: [Papadimitriou 2007]

Além desses, há um outo tipo de funções chamado de funções internas. Essas funções são implementadas em Java, mas seu código é integrado ao sistema, de forma que não podem ser estendidas dinamicamente pelo usuário, como no caso das J-Classes. Elas executam funcionalidades de aritmética complexa, operações com matrizes e outros tipos de funções matemáticas.

O jLab também possui seus próprios carregadores de códigos, para a flexibilidade e extensibilidade do sistema, que executam tão rápido quanto os de implementações da JVM no carregamento de classes Java. Há dois tipos de carregadores de código: o *jClassLoader* e o *jFileLoader* para carregar códigos binários Java (*bytecodes*) e códigos de J-Files, respetivamente. Esses carregadores mantém as classes carregadas em uma *hashtable* para que carregamentos sucessivos de uma classe possuam tempo O(1).

Discussão

A principal vantagem do jLab em relação ao IGED é que ele permite a execução não só de classes Java compiladas, mas também de scripts escritos numa linguagem própria da

ferramenta e um interpretador que integra ambos. Dessa forma, o interpretador do jLab é capaz de executar um conjunto maior de instruções do que o do IGED, já que a linguagem de bytecodes Java é maior do que a linguagem de bytecodes do IGED porque esta última é baseada em um subconjunto da primeira. No entanto, o jLab é um ambiente de programação de propósito geral, não sendo orientado a apresentação gráfica de determinadas estruturas de dados de interesse, ao contrário do IGED que tem o propósito de apresentar graficamente a animação de estruturas de dados específicas. Além disso, o jLab não apresenta algum recurso pedagógico para auxiliar professores e alunos no uso de materiais didáticos para o ensino.

3.2.4 JHAVÉ

O JHAVÉ (*Java Hosted Algorithm Visualization Environment*), apresentado em [Naps 2005], é um ambiente cliente-servidor cujo objetivo não é a visualização da execução de algoritmos propriamente dita, como em muitas ferramentas. Seu objetivo principal é o de prover um mecanismo que permita a construção de ferramentas voltadas à visualização de algoritmos (AV).

Esse ambiente não contribui apenas para a construção de ferramentas de visualização de algoritmos, já que muitas dessas ferramentas concentram-se mais na apresentação gráfica do que na contribuição pedagógica. Por isso, o JHAVÉ provê alguns mecanismos que complementam a representação gráfica da visualização de um algoritmo: Controles como os de um VCR (Videocassete), que permitem que os estudantes avancem e retrocedam os passos de um algoritmo, informações contendo explicações sobre a execução do algoritmo, apresentação do algoritmo que está sendo executado em pseudocódigo, questões que são direcionadas aos alunos durante a execução do algoritmo, geradores de dados de entrada para que os alunos possam ver como seria a execução do algoritmo com esses dados e ferramentas de geração de conteúdo que incluem uma quantidade de bibliotecas de classes que auxiliam na exibição gráfica do algoritmo e que possuem ferramentas de suporte à interação.

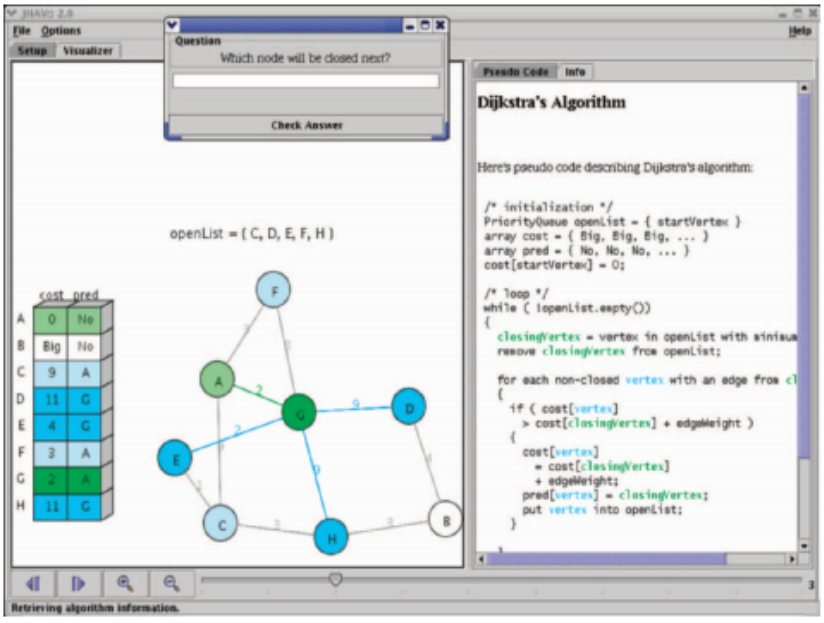
Um mecanismo de AV para o JHAVÉ precisa ser implementado em Java, de forma que seja criada uma classe que herde da classe *Visualizer* para que o mecanismo tenha acesso às funcionalidades de ir diretamente a um *frame* particular da animação, de *zoom* da figura que

está no *frame* atual e de mostrar a animação na forma de movimento com interpolação ou como apresentação de *slides*, além das funcionalidades de avançar e retroceder a execução de um algoritmo.

Além disso, um mecanismo de AV do JHAVÉ precisa produzir a sua visualização de um algoritmo após efetuar um *parsing* de um *script* que contém a visualização do algoritmo na forma textual, renderizando as figuras da animação de acordo com esse conteúdo. Cada mecanismo de AV é livre para determinar a sua própria linguagem de *script*.

Dessa forma, um algoritmo é executado no JHAVÉ de acordo com a sua arquitetura cliente-servidor. O usuário cliente seleciona um algoritmo em uma lista e faz uma requisição ao servidor. O servidor do JHAVÉ, ao invés de renderizar a visualização do algoritmo, escreve os comandos de visualização em um arquivo de *script*. O mecanismo que está do lado do cliente então efetua o *parsing* e renderiza o *script*. Na figura 3.22 é ilustrada a visualização de um algoritmo com o auxílio do JHAVÉ.

Figura 3.22: Visualização de um algoritmo possibilitada pelo JHAVÉ



Fonte: [Naps 2005]

Discussão

O JHAVÉ provê vários mecanismos para facilitar o entendimento da execução de um algoritmo e aprendizagem dos conceitos relacionados a um algoritmo, que o IGED não possui. Dessa forma, o JHAVÉ provê mecanismos de contribuição pedagógica que são realizados durante a execução de um algoritmo, fazendo com que o usuário aprendiz interaja com esses recursos de forma dinâmica. No IGED, os recursos pedagógicos, que incluem arquivos multimídia e exercícios estão armazenados de forma estática, não sendo utilizados assim, durante a execução de um algoritmo. No entanto, o JHAVÉ apenas provê uma forma de criar esses mecanismos, ficando a cargo do usuário implementar os visualizadores contendo esses recursos. No IGED, as funcionalidades de visualização dos algoritmos já são fornecidas pela ferramenta, bem como meios de armazenar e utilizar materiais didáticos.

3.2.5 JIVE

O JIVE (Java Interactive Software Visualization Environment) [Cattaneoa, Faruolo e Petrillo 2004] é um sistema baseado em Java que permite a visualização de algoritmos e estruturas de dados codificados em Java e define uma plataforma comum para a construção de visualizações de vários tipos de aplicações. No JIVE, os algoritmos e suas visualizações são tratados como entidades distintas que interagem entre si usando um canal de comunicação.

A animação de uma estrutura de dados é feita ao derivar de uma classe existente para a estrutura de dados desejada e redefinir os métodos que precisam ser animados. Cada um desses métodos contém o código que irá invocar a representação gráfica de uma determinada operação na janela de animação correspondente.

Há dois tipos de objetos que são manipulados durante a execução de um algoritmo: as estruturas de dados e suas representações gráficas na janela de animação. Esses objetos se comunicam entre si através de eventos de animações, gerados pelas operações que estão nos códigos dos algoritmos, por meio dos canais de comunicação. Assim, o JIVE permite a execução de animações de algoritmos em diversos tipos de aplicação, provendo as seguintes soluções de arquitetura:

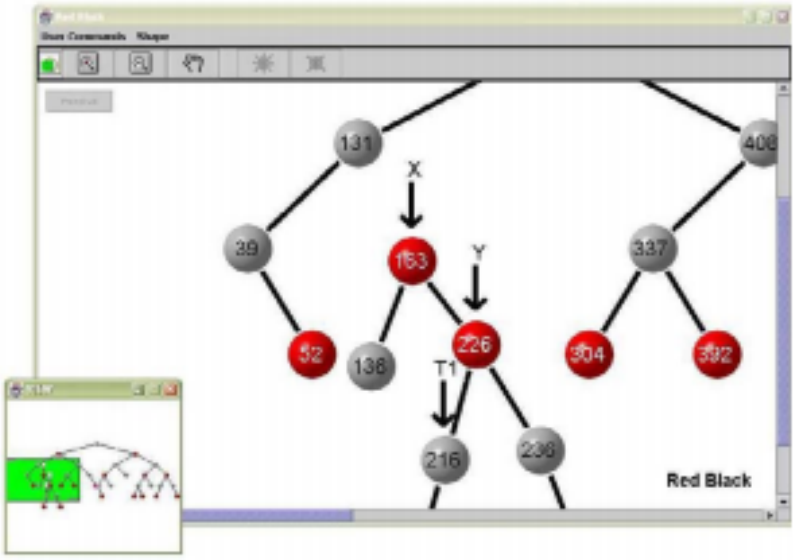
- Visualização Local: as estruturas de dados e as janelas de animações coexistem no mesmo processo.

- Visualização Remota: os algoritmos e suas visualizações são executados em diferentes processos. Dessa forma, os objetos são declarados como remotos e interagem entre si através da invocação remota dos métodos desses objetos por meio de uma rede de comunicação, usando a tecnologia CORBA.

- Salas de Aula Virtuais: Usuários que se conectam na mesma sala virtual compartilham a mesma visualização de um algoritmo. O JIVE implementa isso ao utilizar uma arquitetura distribuída de três camadas onde vários clientes que correspondem às máquinas dos usuários enviam eventos de animação a um servidor de animações, por intermédio de um objeto *proxy*. Assim, pode haver múltiplos canais de comunicação entre clientes e um servidor.

Na figura 3.23 é ilustrada remoção de um nó em uma árvore preta e vermelha, onde cada passo pode ser visualizado através de um *zoom* nos nós envolvidos.

Figura 3.23: Remoção em uma árvore preta e vermelha com o JIVE



Fonte: [Cattaneoa, Faruolo e Petrillo 2004]

Discussão

A principal diferença do JIVE em relação ao IGED é que ele permite a execução distribuída das visualizações das execuções dos algoritmos, onde múltiplos usuários podem colaborar com uma determinada animação. Com isso pode-se utilizar conhecimento agregado para a codificação de um algoritmo, o que útil no processo de aprendizagem já que os alunos podem compartilhar conhecimento entre si de forma mais direta. No IGED, embora a tradução de um código, sua execução e sua visualização gráfica sejam realizadas por componentes distintos, eles coexistem no mesmo processo e apenas um usuário pode fornecer código para uma determinada visualização. Assim como o JHAVÉ, o JIVE fornece apenas uma plataforma para a definição de algoritmos e suas visualizações, ao contrário do IGED, onde a visualização é produzida de forma automática para o usuário. Além disso, o JIVE não provê suporte ao uso de materiais didáticos para o ensino como no IGED.

3.2.6 CIFluxProg

O CIFluxProg (Construtor e Interpretador de Algoritmos) [Santiago e Dazzi 2004] é uma ferramenta que permite aos alunos construírem e executarem algoritmos tanto na forma em Portugol quanto na forma de Fluxogramas, além de visualizarem em detalhes os passos e os resultados dessa execução. Isso atende às necessidades de aprendizado de alunos que têm os perfis visual e textual.

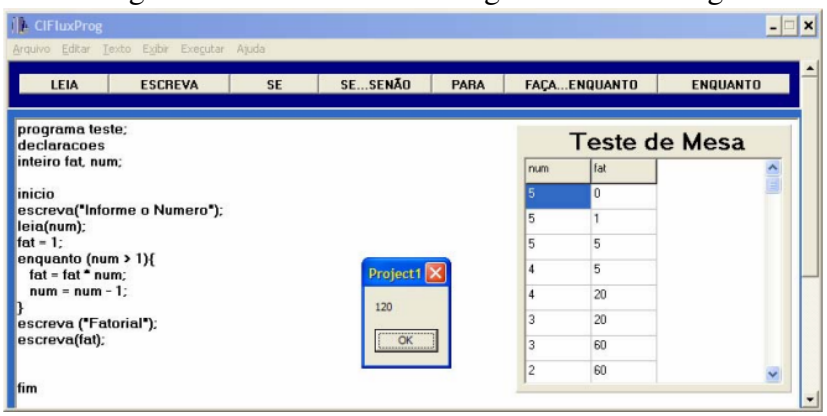
Para isso, a ferramenta possui dois módulos: um que permite a construção e teste de algoritmos em Portugol e o outro em Fluxogramas. A ferramenta possibilita a execução das soluções, as visualizações geradas dessas e dos erros cometidos. Ela foi implementada em C++ e possui um interpretador desenvolvido nessa mesma linguagem com o apoio da ferramenta Lex e Yacc.

Esse interpretador efetua a análise do código informado pelo usuário para um determinado problema, verificando os erros léxicos e sintáticos e o executa, caso não encontre erros. Para a execução de Fluxogramas, o CIFluxProg possui um algoritmo que monta na memória um código que pode ser executado pelo interpretador baseado na representação dos fluxogramas. Já no módulo dos algoritmos em Portugol, a interpretação é feita de forma direta e os códigos

que são interpretados pelos dois módulos são compatíveis, permitindo que o aluno possa testar e comparar o resultado da execução de sua solução na forma de algoritmos em Portugol e em Fluxogramas.

A ferramenta permite a visualização dos valores das variáveis em uma tabela de acompanhamento desses valores ao longo da execução de um algoritmo, no módulo de Portugol. No módulo dos Fluxogramas, é dado o suporte ao aninhamento de símbolos e edição dos elementos dos Fluxogramas. Na figura 3.24 é ilustrada a execução de um algoritmo que calcula o fatorial no módulo de Portugol. Já na figura 3.25, o módulo de Fluxogramas é ilustrado com a sua barra de ferramentas que possui os elementos utilizados na construção dos fluxogramas.

Figura 3.24: Módulo de Portugol do CIFluxProg

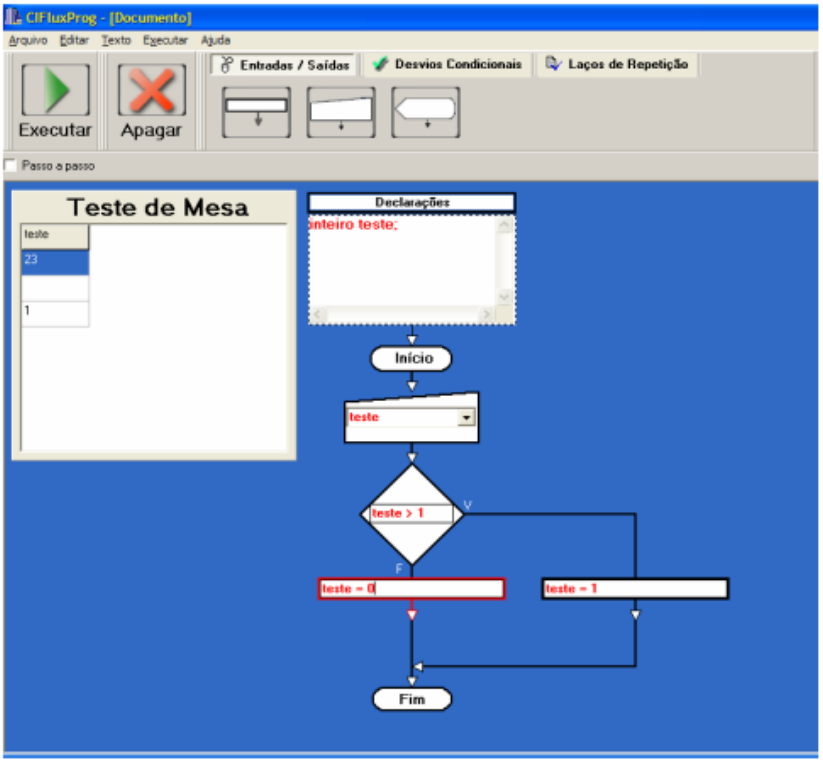


Fonte: [Santiago e Dazzi 2004]

Discussão

O CIFluxProg permite a definição e execução de algoritmos tanto na forma de fluxogramas como na forma textual, em Portugol, ao contrário do IGED que permite a execução dos algoritmos apenas na forma textual, sendo essa uma vantagem do CIFluxProg, já que a representação de algoritmos em Fluxogramas é uma das formas indicadas de aprender a desenvolver algoritmos em cursos introdutórios de programação porque os alunos, nesses cursos, podem não possuir ainda o domínio em uma linguagem de programação. No entanto, por ter sido implementado em C++ , a ferramenta não é portável de forma simples para ou-

Figura 3.25: Módulo de Fluxogramas do CIFluxProg



Fonte: [Santiago e Dazzi 2004]

tras plataformas, como no caso do IGED. A visualização dos valores das variáveis também é uma funcionalidade que não é encontrada no IGED, sendo útil em cursos introdutórios de algoritmos e programação para que os alunos aprendam o funcionamento básico de um algoritmo, já que tabelas de acompanhamento de variáveis é um recurso bastante empregado por professores. Dessa forma, o CIFluxProg fornece uma forma automatizada para isso, o que ainda não ocorre com o IGED. Como em outras ferramentas, o CIFluxProg carece de meios de facilitar a utilização de materiais didáticos de forma semelhante à que é proposta para o IGED.

3.3 Tabela comparativa entre os trabalhos

Um tabela comparativa entre os trabalhos relacionados e entre eles e o IGED, com o Interpretador desenvolvido neste trabalho, é exibida nesta seção. Nessa tabela, a comparação é feita com algumas características encontradas nesses trabalhos, algumas aparecendo em uns e em outros não, e que são úteis no processo de aprendizagem para uma ferramenta de ensino. Essas características são a quantidade necessária de usuários para animar um algoritmo, a plataforma sobre a qual a ferramenta executa, se possui interpretador como o IGED, se a ferramenta se constitui como apenas um *framework* e se com ela é possível executar um algoritmo com mais de uma forma de execução, como execução "de trás para frente", com *breakpoints* e com *stepping*.

Tabela 3.1: Análise comparativa entre os trabalhos relacionados

	Usuários necessários p/ animar o algoritmo	Plataforma	Possui Interpretador?	É um *framework*?	Mais de uma forma de execução?
WEB-UNERJOL	1	Web	Sim	Não	Não
ODIN	1	Web	Não	Não	Não
Spyke	1	Desktop/ Uniplataforma	Não	Não	Não
SEED	1	Desktop/ Multiplataforma	Não	Não	Não
TBC-AED	1	Web	Não	Não	Não
Balsa e Balsa II	2	Desktop/ Uniplataforma	Não	Não	Sim
Zeus	2	Desktop/ Uniplataforma	Não	Não	Não
Tango	2	Desktop/ Uniplataforma	Não	Não	Não
Astral	2	Desktop/ Uniplataforma	Não	Não	Sim
AnimA	2	Desktop/ Uniplataforma	Não	Não	Sim
jGRASP	1	Desktop/ Multiplataforma	Sim	Não	Sim
PCIL	1	Desktop/ Multiplataforma	Sim	Não	Sim
jLab	1	Desktop/ Multiplataforma	Sim	Não	Sim
JHAVÉ	1	Desktop/ Multiplataforma	Sim	Sim	Sim
JIVE	1	Desktop/ Multiplataforma	Sim	Sim	Sim

	Usuários necessários p/ animar o algoritmo	Plataforma	Possui Interpretador?	É um *framework*?	Mais de uma forma de execução?
CIFluxProg	1	Desktop/ Uniplataforma	Sim	Não	Sim
IGED	1	Desktop/ Multiplataforma	Sim	Não	Não

Capítulo 4

IGED

O IGED é um ferramenta desenvolvida para fornecer a estudantes e professores de cursos de Computação suporte didático ao ensino em disciplinas de Programação, Algoritmos e Estruturas de Dados. Para isso, ele possui vários componentes que o torna não apenas uma ferramenta de visualização de algoritmos e estruturas de dados e sim um ambiente com vários recursos para o ensino.

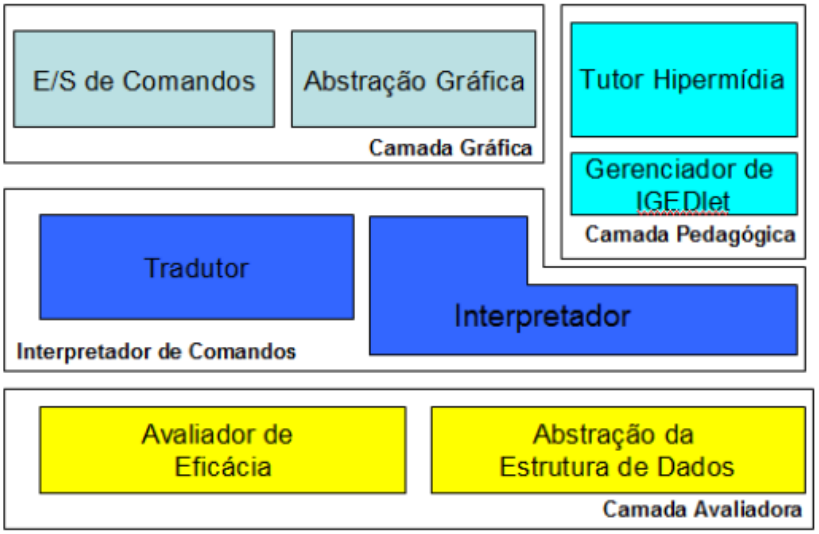
Neste trabalho, foi desenvolvido um Interpretador, que é um componente central na arquitetura do IGED, que permite a execução de código escrito em uma linguagem de programação. Na próxima seção será mostrada a arquitetura do IGED com os seus componentes, e na seção seguinte o Interpretador desenvolvido será detalhado com a sua arquitetura e também serão definidos alguns requisitos que ele poderá ter como ferramenta de ensino. Na última seção deste capítulo, será feita uma justificativa de porque houve uma implementação própria para o Interpretador do IGED se já há implementações da JVM disponíveis ao efetuar um comparativo entre ambos.

4.1 Arquitetura do IGED

A Arquitetura do IGED é composta pelos componentes ilustrados na figura 5.1. Esses componentes estão dispostos nas seguintes camadas: Camada Gráfica, Interpretador de Comandos e Camada Avaliadora, além do Gerador de Tarefas.

A Camada Gráfica é a que irá atuar na apresentação da *interface* com o usuário, a camada Interpretador de Comandos é a responsável por executar os algoritmos inseridos pelo usuário, já a camada Avaliadora é a que irá verificar se o algoritmo inserido pelo usuário está correto para um determinado problema e o quão eficiente ele é, de acordo com a proposta pedagógica da ferramenta. Além dessas camadas, há a Camada Pedagógica, que consiste no agrupamento de materiais didáticos hipermídia que segue o modelo NCM (*Nested Context Model*) [Filho et al. 2012]

Figura 4.1: Arquitetura do IGED



Fonte: [Filho et al. 2012]

4.1.1 Camada Gráfica

A Camada Gráfica é a responsável pela interação com o usuário, desde a entrada de dados até a saída, que corresponde ao resultado da animação gráfica das estruturas de dados. Essa camada contém os seguintes componentes: A E/S de Comandos, responsável por receber os dados do usuário na forma de um algoritmo escrito na linguagem de alto nível do IGED e repassá-lo à camada Interpretador de Comandos.

O outro componente da Camada Gráfica é chamado de Abstração Gráfica. A Abstração Gráfica irá representar as Estruturas de Dados e suas operações por meio de animações [Netto

et al. 2011]. Com as animações, serão mostradas aos usuários as ações que a execução dos passos de seus algoritmos irão realizar nas Estruturas de Dados.

Dessa forma, após receber os comandos do Interpretador de Comandos, a Abstração Gráfica deverá manipular as Estruturas de Dados, se adequando às especificidades de cada uma delas. Essas estruturas podem ser listas, vetores e árvores [Netto et al. 2011].

4.1.2 Camada Avaliadora

O componente Abstração das Estruturas de Dados contém as estruturas que serão manipuladas pela execução do Interpretador para posterior avaliação ao compará-las com as estruturas do professor, com o objetivo de determinar se as do aluno estão corretas [Filho et al. 2012].

Assim, com as tarefas cadastradas pelo instrutor do sistema, a Camada Avaliadora irá avaliar a corretude da resposta informada pelo aluno para um determinado problema, ou seja, se o algoritmo informado pelo aprendiz está de acordo com as expectativas do instrutor para a resolução do problema.

Essa avaliação será feita com a execução dos comandos recebidos pelo Interpretador de Comandos e suas alterações na Abstração das Estruturas de Dados através das ações de manipulação das estruturas em sua memória. A Camada Avaliadora irá trabalhar com dois tipos de instâncias: uma do professor e a outra do aluno. As instâncias do aluno receberão os comandos informados pelo usuário aprendiz e terá as ações desses comandos refletidas nelas, já as instâncias do professor recebem os comandos informados pelo instrutor, ou seja, os comandos considerados como corretos para a resolução de um determinado problema.

Dessa forma, o Avaliador terá acesso a esses dois tipos de instâncias e irá comparar seus estados finais. Se forem iguais, o código informado pelo aprendiz para a resolução do problema será considerado correto, caso contrário estará errado [Filho et al. 2012].

4.1.3 Camada Pedagógica

A Camada Pedagógica, uma extensão proposta em [Filho et al. 2012], consiste em um ambiente de criação de objetos hipermídia, que permitem aos professores a adoção de apresentação de slides aninhados, animações e atividades do contexto da disciplina. Os professores planejam e estruturam a sequência desses objetos, seguindo o modelo NCM, de acordo com o qual, com uma linguagem declarativa, é possível descrever entidades que representam as mídias, seus relacionamentos e eventos que os ativam [Filho et al. 2012].

Um IGEDlet é uma pequena aplicação que realiza uma tarefa específica no IGED. Essas aplicações podem ser:

- AnimaçãoIGEDlet: Representam animações que fazem manipulações sobre estruturas de dados e é composta por um código descrito pela linguagem interpretada pelo ambiente IGED. A apresentação dessas animações é feita de forma passiva para os aprendizes;

- AtividadeIGEDlet: Contém as atividades elaboradas pelo professor para a manipulação de estruturas de dados. Com isso, os alunos usarão a abordagem construtiva para resolvê-los, por meio da linguagem de alto nível do IGED. É composta pela questão a ser resolvida, pelos códigos da linguagem de alto nível do IGED que descrevem a inicialização das estruturas para, a partir dela, o aluno resolver a questão e a solução do professsor para o problema.

O Tocador de IGEDlets é o componente responsável pela execução de todas as aplicações do ambiente IGED. O componente Tutor Hipermídia permite a criação de sequências didáticas, como apresentações de conteúdo aninhadas e aplicações IGEDlet A sequência lógica de apreentação dos conteúdos deverá ser feita de acordo com modelo de autoria hipermídia NCM.

4.1.4 Interpretador de Comandos

O Interpretador de Comandos é a camada responsável pela execução propriamente dita do algoritmo informado pelo usuário na linguagem de alto nível do IGED. Como o código do

algoritmo está em linguagem de alto nível, o componente Tradutor é necessário para traduzir o código para uma linguagem de baixo nível a ser executado pelo Interpretador.

A vantagem de realizar tal abordagem é que a ferramenta poderá manipular as estruturas de dados de acordo com o gerenciamento das operações básicas dos algoritmos. Dessa forma, a animação poderá ser apresentada para o usuário em um nível mais rico de detalhes do que se a execução fosse feita diretamente pela linguagem de alto nível, já que a granularidade dos passos de execução é maior [Netto et al. 2011]. Além disso, com operações de baixo nível, a implementação de uma máquina virtual como a do Interpretador apresentado neste trabalho, torna-se mais simples, visto que esse código reflete de forma mais clara como as instruções devem ser executadas nessa máquina.

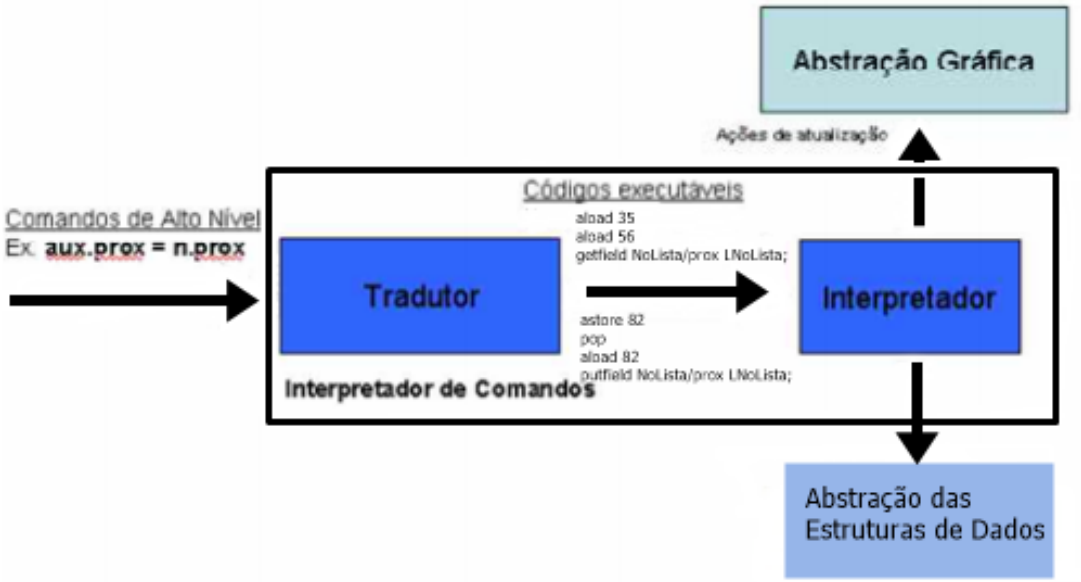
O Interpretador será o componente responsável pela execução dos passos básicos do algoritmo que terá efeito na Abstração Gráfica. Com ele, será possível adicionar novas linguagens de alto nível para a ferramenta com a simples troca do componente tradutor, ou também, a ferramenta poderá executar algoritmos em mais de uma linguagem de alto nível apenas com a adição de mais de um componente Tradutor.

Para a implementação de tais componentes do Interpretador de Comandos foram definidas gramáticas com o uso da ferramenta Antlr [Parr 2007] para o Montador do Interpretador que efetua a tradução do código na linguagem Oolong para a linguagem de máquina de *bytecodes* específica do IGED. Essa ferramenta também poderá ser utilizada no desenvolvimento do componente Tradutor.

O fluxo de informações nessa camada é exemplificado com a figura 5.2, onde o componente Tradutor recebe um código informado pelo usuário em uma linguagem de alto nível e o traduz para código em uma linguagem de baixo nível, cuja execução pelo Interpretador irá refletir ações de atualização no componente Abstração Gráfica.

Para dar suporte à execução de aplicações IGEDlet, o Interpretador irá operar de acordo com 4 modos distintos [Filho et al. 2012]. Esses modos estão listados abaixo:

Figura 4.2: Exemplo do Fluxo de Informações no Interpretador de Comandos



Fonte: Adaptado de [Netto et al. 2011]

- Modo gráfico: Suas saídas de atualização serão encaminhadas somente para a camada Abstração Gráfica;
- Modo professor: A Camada Gráfica ficará desabilitada e o Interpretador irá apenas atualizar as instâncias das estruturas de dados do professor;
- Modo aluno: As Camadas Gráfica e Avaliadora ficarão habilitadas, mas somente as instâncias das estruturas de dados do aluno serão atualizadas;
- Modo inicialização: As Camadas Gráfica e Avaliadora são habilitadas e tanto as instâncias das estruturas de dados do professor quanto as do aluno são atualizadas.

Para executar o código oriundo de uma AnimaçãoIGEDlet, O Interpretador será configurado no modo gráfico pelo tocador de IGEDlet e receberá esse código em intervalos de tempo pré-determinados. Quando receber o código de uma AtividadeIGEDlet, o Interpretador primeiro é configurado no modo de inicialização, recebendo assim, o código de inicialização das estruturas de dados [Filho et al. 2012].

Após realizar as operações de inicialização, o Interpretador será configurado no modo professor, receberá o código da solução do problema em questão e após terminar a execução deverá salvar as estruturas do professor. Em seguida, o Interpretador é configurado no modo aluno e deverá receber a resposta do código de um aprendiz para o referido problema, mantendo as alterações das estruturas do aluno na memória.

Se o Tocador de IGEDlet estiver tocando um IGEDlet e for solicitado o reinício da execução de uma animação ou atividade, o Interpretador deverá limpar as suas atualizações e o processo de execução será reiniciado [Filho et al. 2012].

4.2 Interpretador de Comandos

4.2.1 Arquitetura do Interpretador

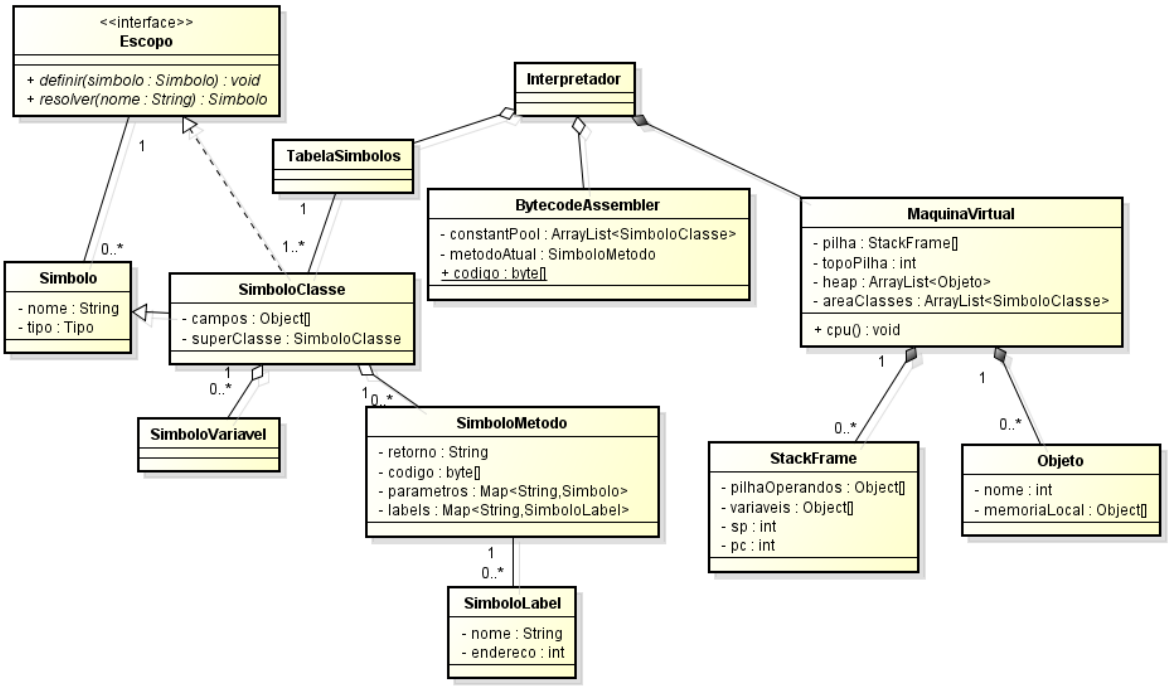
O Interpretador desenvolvido para esse trabalho tem o seu modelo de execução baseado no da JVM [Engel 1999]. Assim, é um Interpretador de *bytecodes* baseado em pilha, como é apresentado em [Parr 2010], e opera sobre uma Máquina Virtual. Esse Interpretador foi desenvolvido com a aplicação dos seguintes padrões apresentados em [Parr 2010]:

- *Tree Grammar*: Utilizado pelo Montador para efetuar os percursos sobre a árvore sintática gerada pelo *parser* desse componente, a partir do código informado pelo usuário;
- *Symbol Table for Classes*: Esse padrão foi aplicado para que o Interpretador gerenciasse os símbolos contendo os dados traduzidos pelo Montador na forma de classes e seus componentes: campos, métodos e *labels*;
- *Bytecode Assembler*: O Interpretador, por meio do seu componente Montador, traduz o código informado pelo usuário em linguagem de máquina de acordo com esse padrão.

O Interpretador possui os seus componentes dispostos de acordo com o diagrama de classes ilustrado na figura 4.1. Nesse diagrama, é mostrado um modelo das classes que representam os principais componentes do Interpretador, para facilitar o entendimento de como eles estão organizados e como eles operam, não correspondendo à forma exata de como foram

implementados, já que algumas informações de variáveis, métodos e outras classes foram omitidas para fins de simplificação.

Figura 4.3: Arquitetura do Interpretador



De acordo com esse diagrama, o Interpretador possui três componentes principais: Uma Máquina Virtual (VM), o Montador (*Bytecode Assembler*) e uma Tabela de Símbolos. O Interpretador inicialmente lê um arquivo de entrada contendo um código na linguagem de montagem Oolong, efetua a análise léxica e sintática, gerando com isso, uma árvore sintática. Para permitir referências a símbolos no código, como classes, métodos, variáveis e *labels* antes deles serem definidos (*forward references*), o Interpretador efetua duas passagens por essa árvore sintática: uma para definir todos esses símbolos quando são definidos no código, armazenando-os na Tabela Símbolos e outro para identificá-los quando são referenciados. A Tabela de Símbolos contém as classes e suas informações: campos e métodos.

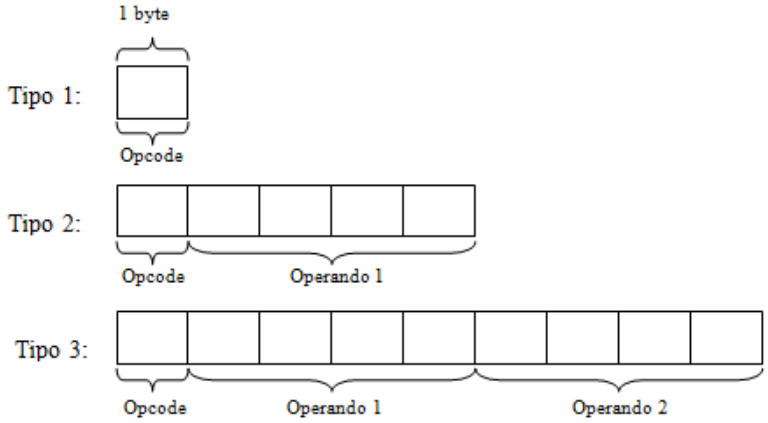
Com isso, o componente Montador armazena as classes referenciadas em uma estrutura de lista (*constant pool*), cujas identificações são seus respectivos índices nessa lista, e gera

código de máquina para cada método, baseado nos *opcodes* das instruções (ver apêndice A), seus operandos e os identificadores das referências. Essas referências de classes armazenadas formarão a área de classes da Máquina Virtual(VM).

A VM possui, além da área de classes, uma pilha para as chamadas de métodos e um *heap* para armazenar os objetos. Ela possui um método *cpu* que simula a execução de uma CPU para as instruções do método que está no topo da pilha (método corrente).

Cada instrução, na linguagem de máquina, é formada pelo seu *opcode* seguido de operandos, se houver. A memória do código de cada método é formada por um conjunto de células de um *byte* (inteiro curto, na linguagem Java) e cada uma dessas células armazena o opcode de uma instrução a ser executada ou parte de um dos operandos de uma instrução, visto que cada operando é representado pelo tipo de dados inteiro (4 *bytes* na linguagem Java), portanto ocupa quatro células da memória do código. Na figura 4.2, é ilustrado como a memória do código é organizada de acordo com os tipos de instruções.

Figura 4.4: Organização da memória de código



A pilha é organizada em *stackframes* que possuem as estruturas necessárias para a execução de um método. Cada *stackframe* possui uma pilha que contém os operandos que estão sendo manipulados durante a execução das instruções. Esses operandos podem ser do tipo de dados inteiro ou referência a objetos no *heap*. Em um *stackframe*, também há um espaço de

armazenamento que contém as variáveis locais a um método, onde os tipos de dados também podem ser inteiro ou referência a objetos.

Além desses, há o ponteiro de pilha que sempre apontará para o operando que estiver no topo da pilha e o contador de programa que deverá apontar para o início do código que representa a instrução que está sendo executada no momento na memória de código, ou seja, a célula que representa o *opcode*. Quando um método for chamado, um *stackframe* é empilhado e desempilhado com o término de sua execução.

O *heap* é a estrutura que contém os objetos manipulados diretamente pelo Interpretador. Cada objeto contém um nome e um espaço onde são armazenados os dados dos seus campos. O Interpretador, quando traduz o código de *bytecodes* para a linguagem da sua Máquina Virtual, armazena informações das classes em uma estrutura chamada área de classes, que funciona como um repositório de classes. Cada classe possui uma área reservada para os seus campos estáticos, bem como seus métodos. Um método, dentro de uma classe, contém o seu código em linguagem de máquina e informações necessárias para a sua execução, como parâmetros, tipo de retorno e tamanho da memória local.

Além dessas informações, uma classe, dentro da área de classes, possui uma referência à sua superclasse, caso possua uma. Dessa forma, o Interpretador do IGED fornece suporte à herança e também ao polimorfismo, visto que, quando a partir de uma referência a um objeto for necessária a execução de um método, o Interpretador conseguirá localizá-lo, se o método pertencer à própria classe ou a sua superclasse e também pelo fato de que quando um objeto é carregado no *heap*, o espaço de memória das superclasses desse objeto, se houver, também é acrescentado no próprio objeto.

Dessa forma, o Interpretador pode executar tanto métodos estáticos quanto métodos não estáticos. A única diferença entre a execução de um método estático e um método não estático é que, no segundo, a referência ao objeto que o invocou é passado como um de seus parâmetros, fazendo parte assim, da memória local do método no seu *stackframe*, o que não ocorre com os métodos estáticos.

Campos com variáveis estáticas e não estáticas também são permitidos. Essas variáveis podem ser de um dos tipos de dados suportados pelo Interpretador até o momento: inteiro ou referência a um objeto. Assim, em uma classe pode haver Composição.

4.2.2 Modelo de Execução

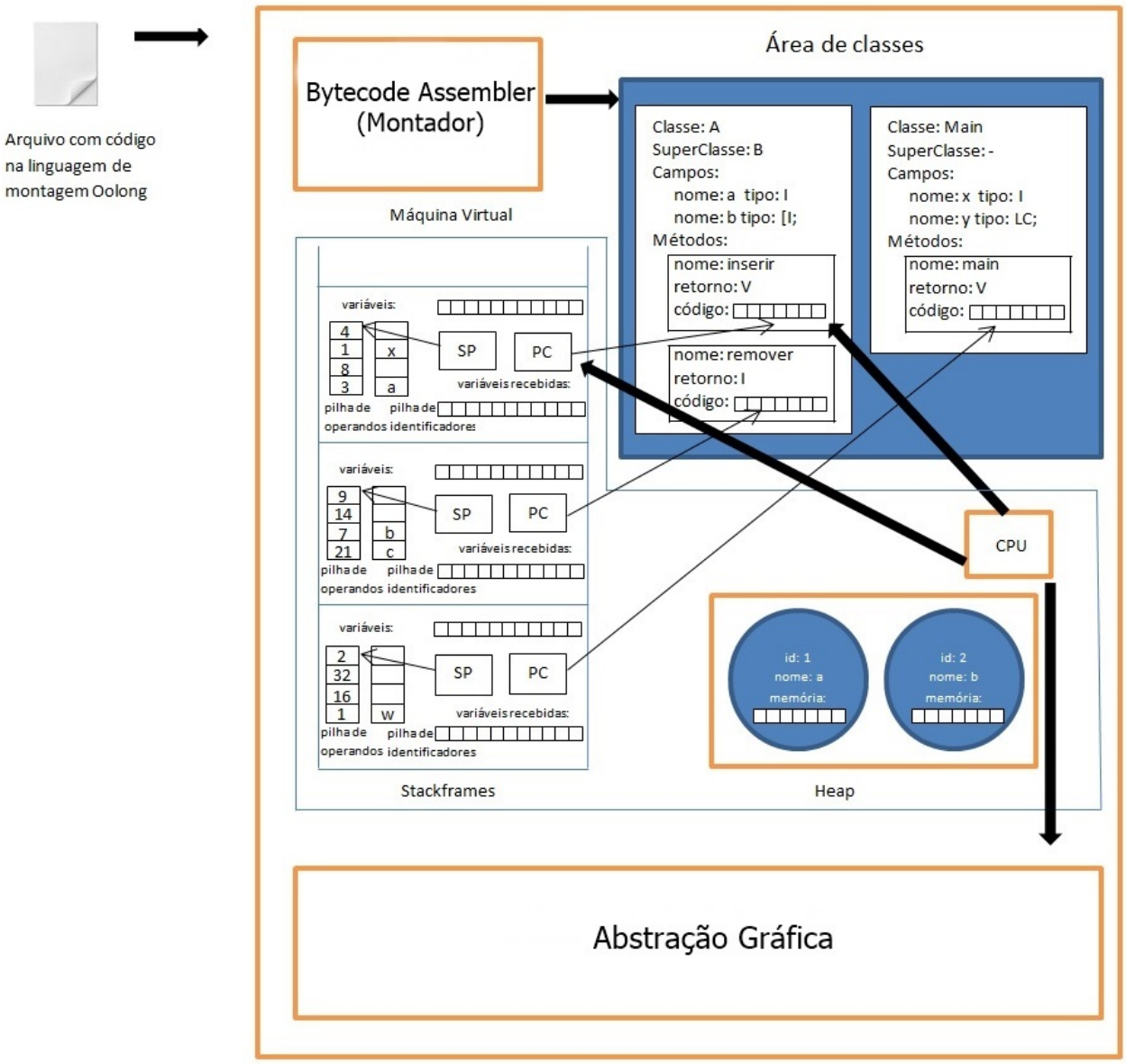
Na figura 4.5, é ilustrado o modelo de execução do Interpretador do IGED que recebe como entrada um arquivo contendo código na linguagem Oolong. O *Bytecode Assembler* (Montador), após traduzir esse código, gera definições de classes na área de classes, onde cada método possui a sua memória de código que armazena o código na linguagem de máquina do Interpretador a ser executado quando o método for chamado.

As classes devem estar em um diretório com nome "classes"no mesmo local do projeto. O Interpretador inicia a sua execução procurando por um método de nome *main* sem parâmetros e com o tipo de retorno *void* (tipo V na linguagem Oolong). Caso encontre esse método, um *stackframe* é empilhado para ele e a máquina virtual do Interpretador inicia a execução desse método acessando o seu código, já que o *stackframe* possui uma referência ao método de uma classe.

Quando um *stackframe* é empilhado, são alocados espaços de memória necessários a execução de um método: o conjunto de variáveis locais do método, uma pilha de operandos, variáveis recebidas e uma pilha de identificadores, além os registradores SP (Ponteiro de Pilha) e PC (Contador de Programa). A pilha de operandos contém os operandos de cada instrução que está sendo executada, são valores numéricos que podem ser tanto os valores dos operandos carregados da memória local ou empilhados como constantes, como também podem ser referências a endereços da memória local e referências a um objeto do *heap*, que possui como identificador um valor numérico.

A pilha de identificadores contém os identificadores das variáveis, com os valores que serão exibidos na tela do IGED para essas variáveis. Isso é necessário porque as instruções da linguagem de baixo nível a serem executados pelo Interpretador só referenciam endereços de memória, que são valores numéricos, ao passo que na tela do IGED esses valores são apre-

Figura 4.5: Modelo de execução do Interpretador do IGED



sentados com letras e números. Sendo assim, o Interpretador do IGED gera esses valores a serem exibidos na tela e os empilham na pilha de identificadores no mesmo instante que seu respectivo valor de endereço de memória é empilhado na pilha de operandos. Esses identificadores são gerados pelo Interpretador porque não há uma tabela de símbolos que mapeie os identificadores das variáveis aos seus respectivos endereços de memória. Quando houver um Tradutor que traduza código em linguagem de alto nível para código em linguagem de montagem Oolong, essa tabela de símbolos deverá estar disponível para que o Interpretador busque nessa tabela o identificador de uma variável para um determinado endereço.

Assim, quando for necessária uma operação de escrita, o Interpretador invocará uma rotina de atualização gráfica do componente Abstração Gráfica, passando como parâmetro o valor que está na pilha de identificadores. Para haver um controle dessa pilha, toda vez que ocorre um carregamento na pilha de operandos, há um carregamento na pilha de identificadores, cujo valor é vazio quando não é um identificador. Da mesma forma, há um desempilhamento da pilha de identificadores quando ocorre um desempilhamento na pilha de operandos.

Além da pilha de identificadores e do conjunto de variáveis locais também há um espaço de memória com mapeamento dos endereços de memória e dos identificadores das variáveis que são passadas como referência na ocasião da chamada de um método. Assim, quando o Interpretador solicita à Abstração Gráfica uma atualização gráfica na tela do IGED para uma estrutura de dados, ele busca o identificador primeiro fazendo uma consulta a esse mapeamento e se não encontrar ele gera um identificador. Posteriormente, quando houver o componente Tradutor, ele fará uma busca na tabela de símbolos. Isso é necessário para que o Interpretador referencie a estrutura de dados com o mesmo identificador quando a referência da estrutura de dados é passada como parâmetro a um método para que ele não crie uma outra estrutura de dados nesse caso, fazendo com que a abstração gráfica mostre para o usuário que o Interpretador está atuando sobre a mesma estrutura de dados quando ela é passada como referência a outro método.

Além desses, há o *Stack Pointer* (SP) que aponta para o topo da pilha de operandos e o *Program Counter* (PC) que aponta para a instrução que está sendo executada em um método. Quando um método invoca outro método, esses valores, como os demais valores dos espaços

de memória alocados em um *stackframe*, são salvos.

A CPU executa as instruções do método que está sendo referenciado no topo da pilha dos *stackframes*. A CPU executa as instruções acessando os dados do *stackframe* do topo da pilha e também invocando saídas de atualização gráfica, que consistem em procedimentos do componente Abstração Gráfica. Quando um objeto é criado, é alocado um espaço de memória para esse objeto no *heap* por meio de um carregador de classes. Cada objeto no *heap* possui um identificador numérico, um nome definido pelo programador e um espaço de memória para as suas variáveis locais, que é formado a partir das variáveis definidas para a classe do objeto como também das variáveis definidas nas superclasses dessa classe.

Na área de classes, para cada campo e método, são armazenadas informações sobre se eles são estáticos ou não. Além disso, em cada método também são armazenadas informações sobre os seus parâmetros, como nome e tipo. Essas informações não são apresentadas na figura 4.5 para fins de simplificação.

4.2.3 Requisitos do Interpretador como ferramenta de ensino

Pelo fato de o Interpretador possibilitar as visualizações gráficas das estruturas de dados ao longo da execução de um programa ao se ter acesso direto à forma pela qual ele executa as instruções, é possível que ele seja utilizado para o ensino de Computação em diversas disciplinas. Nesta seção, são definidos alguns requisitos funcionais que o Interpretador pode ter e que auxiliaria no processo educacional.

Esse Interpretador poderia ser empregado para o ensino de disciplinas de Introdução a Algoritmos, Programação e Estruturas de Dados. Uma das utilidades que o Interpretador poderia ter para o ensino de programação seria permitir que um aluno possa controlar a execução de um programa, podendo ele verificar graficamente como o seu algoritmo está sendo executado, identificando com isso possíveis erros, o que auxiliaria na correção deles e contribuiria no aprendizado do aluno, já que com isso, ele aprenderia a desenvolver algoritmos de forma cada vez mais rápida porque a probabilidade dele cometer os mesmos erros do passado tenderia a ser menor.

Uma funcionalidade que esse interpretador poderia prover a uma ferramenta de ensino é a execução passo a passo. Essa execução passo a passo poderia ser feita com o uso de *breakpoints*. Quando o usuário determinasse esses *breakpoints* ou pontos de parada no código, a execução do programa seria paralisada pelo Interpretador, que executaria a próxima instrução com um comando fornecido por um usuário.

Com a execução passo a passo, poderiam ser mostrados para um usuário aprendiz os estados das variáveis e a alteração desses estados conforme um programa é executado passo a passo. No caso de a variável ser uma referência a um objeto, o estado do objeto referenciado por essa variável e que está contido no *heap*, ou seja, os valores de suas variáveis não estáticas como também as estáticas, seria mostrado.

Com isso, o aluno ficaria sabendo que instrução está sendo executada no momento pelo Interpretador, podendo acompanhar a execução do algoritmo e saber em que parte do código as estruturas estão sendo alteradas. O Interpretador poderia não executar o mesmo código informado por um usuário de forma direta se houver um componente que traduza códigos em linguagem de alto nível para a linguagem do Interpretador. Nesse caso, um usuário informaria código em linguagem de alto nível e o Interpretador executaria o código traduzido por esse componente.

Dessa forma, para cada instrução que estiver sendo executada, o Interpretador deverá conter a informação do número da linha em que essa instrução está inserida no código informado pelo usuário, podendo esse estar definido em uma linguagem de alto nível ou na linguagem de baixo nível Oolong. Algo que poderia ser feito para isso seria quando o Tradutor de código em alto nível traduzir o código informado pelo usuário, ele fornecesse ao Interpretador não apenas o código traduzido em Oolong, mas também um mapeamento com os números de cada instrução na linguagem de baixo nível de acordo com a ordem em que elas são definidas em um método e os números de linha de um arquivo de código fonte, Os números de cada instrução em linguagem de baixo nível são seus endereços a partir dos quais o Interpretador as executam após o processo de montagem do código em baixo nível. Assim, quando o Interpretador executasse uma instrução de um determinado endereço, ele consultaria no mapeamento qual a linha correspondente no código em linguagem de alto nível, podendo gerar

com isso, uma saída gráfica que mostrasse para o usuário que linha está sendo executada no momento.

Quando o Interpretador executasse o código em linguagem Oolong diretamente, no caso de o usuário fornecer o código nessa linguagem, seria o próprio Interpretador que criaria esse mapeamento de endereços de instruções e de linhas dos arquivos de código em baixo nível. Dessa forma as funcionalidades de *stepping* ou de *breakpoints* também poderiam ser utilizadas nos códigos informados com a linguagem Oolong, o que seria útil para o ensino de programação em baixo nível, como ocorre em disciplinas de Arquitetura de Computadores.

Algumas estruturas que estão na memória da máquina virtual do Interpretador deverão ter um tratamento especial. Essas estruturas são as utilizadas em disciplinas de Estruturas de Dados, como vetores, listas, pilhas, filas, árvores e grafos.

O Interpretador também poderia ser utilizado no ensino da própria linguagem de programação Java. Isso porque, com ele, seria possível visualizar pela Abstração Gráfica as estruturas internas de sua Máquina Virtual, cujo modelo é semelhante ao apresentado em [Engel 1999] para a JVM. Com ele, os alunos poderiam programar de forma direta em uma linguagem de montagem de *bytecodes* Java e verificar como o seu programa seria executado pela JVM por meio das visualizações das alterações das estruturas internas da VM do Interpretador ao longo da execução de um programa.

Isso poderia ser útil na própria aprendizagem da linguagem de *bytecodes* Java, o que seria uma alternativa ao ensino de linguagens de baixo nível, que geralmente é feito em disciplinas de Arquitetura e Organização de Computadores, onde se aprende programação em linguagem assembly. Códigos em linguagem Oolong e códigos em linguagem assembly de uma forma geral possuem muitas semelhanças por serem ambas linguagens de montagem de baixo nível.

Além disso, caso esse modelo de execução baseado na visualização dos componentes internos da VM fosse integrado ao modelo de execução por *stepping* mencionado anteriormente, reforçaria na aprendizagem de vários conceitos de programação. Um desses concei-

tos seria a Orientação a Objetos porque dessa forma, seria possível por exemplo, um aluno visualizar o momento exato da criação de um objeto, como isso de fato ocorre, como também como os objetos e suas classes estão relacionados entre si dentro das estruturas internas da VM, por meio das quais pode ser mostrado como ocorre a herança, a composição e o polimorfismo, oferencendo assim uma forma concisa e alternativa a como esses conceitos geralmente são apresentados aos alunos, tanto em livros didáticos como em salas de aula.

Outro conceito importante de programação cujo aprendizado poderia ser reforçado por essa ferramenta seria a Programação Estruturada. Através da visualização da pilha da VM ao longo da execução de um programa, é possível que um aluno acompanhe os estados dos seus *stackframes* individuais, entendendo como é feita a chamada e execução de procedimentos, seja por métodos diferentes ou pelo mesmo método, no caso de ocorrer recursividade.

O Interpretador poderia ser utilizado para que se tenha noções sobre a complexidade dos algoritmos que ele executa. Para isso, ele poderia efetuar uma contagem dos passos básicos correspondentes às instruções que ele executa. Nesse contexto, cada instrução executada pelo Interpretador teria um mesmo custo, que equivaleria a uma unidade de tempo. Dessa forma, um componente adicional apresentaria para o usuário um gráfico de função matemática, onde a abscissa corresponderia aos valores numéricos de tamanho de determinadas entradas de dados, e a função f(x) seria o tempo que o interpretador levou para executar o algoritmo para cada entrada de dados.

Sendo assim, seria necessário o uso de geradores de entradas. Para cada algoritmo, cujo gráfico de tempo um usuário quisesse visualizar, ele informaria ao Interpretador um ou mais tipos de estruturas de dados de entrada, por exemplo, um vetor, uma matriz ou um grafo, para que com isso, o gerador de entradas gerasse entradas de tamanho em um intervalo de valores informado pelo usuário.

Com essa funcionalidade, o Interpretador teria um modo de execução no qual ele faria a contagem do tempo de execução do algoritmo para cada entrada de tamanho n gerada pelo gerador de entradas, fornecendo os dados, que seriam esse tamanho e o tempo de execução,

ou seja x e f(x), a um componente que atualizaria o gráfico de tempo do algoritmo ao término de sua execução para uma entrada de determinado tamanho.

Ao visualizar a função do tempo de seu algoritmo para um conjunto de entradas de tamanho variável, um aluno poderia verificar se era o esperado pelo cáculo de complexidade de seu algoritmo ou determinando essa complexidade, já que a própria visualização do gráfico de tempo de execução poderia fornecer para ele essa percepção, ou pelo menos o auxiliaria nisso. Um professor poderia usar essa ferramenta para mostrar aos seus alunos os limites assintóticos de determinados algoritmos, fornecendo para eles alternativas de algoritmos que resolvem um mesmo problema, mas que possuem tempos de execução diferentes para o pior caso, o melhor caso, ou até mesmo o caso médio. Essas são algumas aplicações didáticas que essa funcionalidade poderia fornecer.

O Interpretador, além de gerar saídas gráficas durante a execução das instruções, também poderia gerar saídas gráficas durante o seu processo de tradução, sendo útil assim para disciplinas de Compiladores. Essas saídas podem ser geradas durante as três etapas da tradução do código na linguagem de montagen de *bytecodes* na linguagem de máquina: as análises léxica, sintática, semântica e a síntese no código alvo.

Após a análise léxica, poderia ser mostrada uma listagem com os *tokens* gerados, com os seus identificadores e valores, relacionados ao código informado pelo usuário. Durante a análise sintática, a árvore sintática construída de acordo com as regras gramaticais do Interpretador poderia ser exibida.

Durante o processo de análise semântica, o Interpretador geraria de forma gráfica as saídas que mostrariam para o usuário aprendiz o percurso sobre essa árvore. Cada elemento dessa árvore seria destacado quando for o elemento corrente do processo de análise. O percurso da árvore sintática durante a análise semântica é feito duas vezes, como mencionado anteriormente, para permitir as *forward references*. Durante o primeiro percurso, podem ser mostradas para os usuários as inserções dos símbolos identificados na forma de suas representações gráficas na tabela de símbolos, onde seriam exibidos os símbolos de classes e seus componentes (símbolos de métodos, atributos e variáveis).

Na segunda passagem, seria mostrado o processo de geração de código para o usuário, com uma janela contendo o código traduzido em linguagem de máquina. Seriam exibidas as instruções sendo adicionadas nessa janela uma por uma, de acordo com a tradução do código relacionado ao elemento que estivesse em destaque na árvore sintática. Cada passo da tradução seria realizado com mensagens informativas explicando o que está ocorrendo. Essa funcionalidade poderia ser estendida também para o componente que traduziria código em linguagem de alto nível. Assim, um aluno poderia visualizar como ocorre um processo de tradução em uma linguagem de programação orientada a objetos.

4.2.4 Considerações sobre a possível utilização de uma implementação da JVM como Interpretador do IGED

Além da implementação de uma máquina virtual com o interpretador proposto, neste trabalho também são feitas considerações sobre o uso de uma implementação da JVM, como a Java HotSpot VM que está contida no JRE da Oracle, como interpretador do IGED. Dessa forma, pretende-se avaliar as vantagens e desvantagens dessa abordagem, justificar porque foi feita uma implementação própria de um interpretador para o IGED (se já existem implementações da JVM amplamente utilizadas), contribuir para trabalhos futuros que visem o aprimoramento e aperfeiçoamento do Interpretador do IGED, resultando na conclusão sobre qual seria a melhor abordagem a ser realizada, se é o Interpretador proposto neste trabalho ou a utilização de uma implementação da JVM para o Interpretador do IGED.

Vantagens

A utilização de uma implementação da JVM traz uma série de benefícios. O primeiro deles é em relação à segurança. O algoritmo de verificação detecta quais programas seguem as regras de segurança da JVM e quais não, antes deles serem executados. Existem várias formas pelas quais um programa poderia violar a segurança da JVM, como "estourar"a pilha, acessar endereços de memória não permitidos e conversões inapropriadas de objetos para obter ponteiros à endereços não permitidos de memória [Engel 1999].

Várias linguagens de propósitos específicos possuem compiladores com implementações que geram *bytecodes* para a JVM, incluindo Scheme, Prolog, Tcl, ML, Eiffel, Python e Ada [Engel 1999]. Com isso seria possível escrever código nessas linguagens ao passo que o IGED atualmente só suporta codificação em uma única linguagem. Seria necessário desenvolver outros tradutores para suportar a execução de código em mais de uma linguagem. Essas linguagens poderiam ser a melhor a opção para a implementação de algoritmos para determinados problemas, pois podem fornecer características que não são encontradas na linguagem Java ou na linguagem de alto nível do IGED. Isso evitaria a necessidade de implementação de compiladores dessas linguagens para a ferramenta IGED, visto que já existem.

Com a utilização de uma implementação da JVM, os programas das linguagens de alto nível seriam traduzidos para a linguagem de *bytecodes* dessa máquina virtual. Dessa forma, esses *bytes* poderiam ser mantidos não apenas em um arquivo para serem executados, mas também na memória física de um computador, em um servidor web, em um banco de dados ou em qualquer lugar que possa armazenar coleções de bytes [Engel 1999], ou seja, os programas poderiam ser executados a partir de vários tipos de fontes de dados. Com isso, outras formas de execução do IGED poderiam ser consideradas para melhorar a sua usabilidade, como a utilização de arquiteturas cliente-servidor ou na forma de *applets*, por exemplo.

Os objetos que não estão ativos e que não são mais usados podem ter os seus espaços de memória liberados pelo *garbage collector*, uma das formas utilizadas para otimizar os epaços de memória da JVM, já que esta é limitada [Engel 1999]. A memória do Interpretador do IGED, por ter sido implementada com a linguagem Java, possui limites estabelecidos pela própria JVM subjacente.

Com isso, esse recurso não deixa de ser utilizado para o Interpretador do IGED, porém, os objetos a serem reivindicados pelo *garbage collector* da JVM não se restringem apenas àqueles que estão no *heap* do Interpretador do IGED, são todos aqueles instanciados pela execução do Interpretador como um todo. Dessa forma, ao utilizar o Interpretador do IGED, não há um recurso de reciclagem dos espaços de memória alocados por objetos que não são mais usados no *heap*, como ocorre com a JVM, apesar de que trabalhos futuros possam tratar da implementação de um *garbage collector* para o Interpretador do IGED.

Outras vantagens em se utilizar uma implementação da JVM é que existem implementações da JVM utilizadas há bem mais tempo do que o Interpretador do IGED, sendo testada inúmeras vezes por milhões de usuários ao redor do mundo. Isso é importante já que, dessa forma, é mais estável e já solucionou diversos problemas que podem aparecer no Interpretador do IGED.

Em consequência disso, uma implementação da JVM poderia possuir mais otimizações de gerenciamento de memória e de desempenho do que o Interpretador do IGED, além de recursos avançados introduzidos em cada versão lançada. Além disso, seria mais confiável utilizar um interpretador desenvolvido e testado por um grupo maior de desenvolvedores, como é o caso da JVM da Oracle e de outras implementações de código aberto, como o OpenJDK.

Otimizações de desempenho da JVM incluem a obtenção do máximo proveito das capacidades dos processadores modernos [Engel 1999]. O desempenho de uma implementação da JVM seria melhor nesse caso, visto que o próprio Interpretador do IGED executa sobre uma implementação da JVM, enquanto que esta última não precisa de outro interpretador ou máquina virtual que possa operar entre ela e o hardware do computador, ao contrário do Interpretador do IGED que é dependente da JVM, precisando dela para operar, já que foi implementado na linguagem Java.

Com uma implementação da JVM como interpretador, não seria necessária a definição de uma linguagem de alto nível própria do IGED. A linguagem de alto nível poderia ser a própria linguagem Java e outras que podem ser executadas pela JVM. Dessa forma, classes e funcionalidades específicas do IGED poderiam ser fornecidas por meio de bibliotecas.

Desvantagens

Apesar de uma implementação da JVM como a Java HotSpot VM ser bastante utilizada e aprovada por muitos usuários pelo mundo e, como uma máquina de propósitos gerais, realizar tarefas de execução de métodos, operações aritméticas e diversas outras que um algoritmo implementado em uma linguagem de programação orientada a objetos necessite para

ser executado, ela carece de características com as quais possa realizar tarefas de exibição gráficas [Engel 1999].

O Interpretador do IGED deve possuir uma máquina que realize a exibição gráfica dos algoritmos que estão sendo executados, por meio das saídas do Interpretador. Dessa forma, a saída com o comando de apresentação gráfica do Interpretador deve ocorrer no momento da execução da instrução que efetue uma alteração nas estruturas de dados, de forma que seu resultado possa ser apresentado pelo componente Abstração Gráfica.

O problema em usar uma implementação da JVM como a Java HotSpot VM, nesse caso, é como saber o momento exato da execução de uma instrução para produzir a saída gráfica correspondente. Como a Java HotSpot VM e outras implementações foram implementadas por terceiros e não especificamente para o IGED, é mais complicado saber qual a operação que uma instrução executa e o momento exato que a JVM irá executá-la. Uma solução poderia ser utilizar uma implementação de código aberto da JVM para verificar em que momento exato essas instruções são executadas, mas isso demandaria um estudo desse código fonte.

Além disso, é mais complicado implementar um tradutor que traduza o código de alto nível do IGED para a linguagem de *bytecodes* Java do que implementar um tradutor para a linguagem de montagem Oolong. Com isso, tradutores para outras linguagens que poderiam ser utilizadas no IGED seriam implementados de forma mais simples. Nesse caso, usando uma implementação da JVM com tradutores para a linguagem Oolong, a implementação de um *assembler* para a JVM ou a utilização de um já existente, como o Jasmin ou o apresentado em [Engel 1999], seria necessário.

Com uma implementação própria de um Interpretador, pode-se determinar com mais facilidade como suas estruturas internas deverão ser manipuladas para os propósitos do IGED. Como por exemplo, o acesso às estruturas de dados que estão armazenadas na memória da máquina virtual para gerar as saídas de atualizações gráficas e também para refletirem mudanças nas estruturas do componente Abstração Gráfica.

A única forma de acessar os componentes internos da JVM é pela utilização de métodos nativos em C/C++ [Engel 1999]. Códigos nativos em C nem sempre são portáveis porque não existem em todas as implementações da JVM [Engel 1999]. A utilização de outro intepretador demandaria um tempo adicional para estudar como ele se adequaria aos componentes do IGED para prover as funcionalidades necessárias de acordo com a proposta didática da ferramenta. Com o uso do interpretador desenvolvido neste trabalho, já que ele foi projetado para se adequar aos requisitos do IGED, haverá mais tempo para a implementação de outros componentes e adição de novos que deixem a ferramenta mais adequada para o ensino e aprendizagem. Além disso, é mais fácil lidar com uma implementação própria do Interpretador do que usar outro que possua restrições para acesso aos seus componentes internos.

No caso da utilização da Java HotSpot VM como Interpretador do IGED, uma abordagem que poderia ser feita seria por meio da notificação da alteração dos objetos que representam as estruturas de dados e que estão sendo executados da Máquina Virtual por meio de eventos que podem ser acessados através de códigos nativos pela JVM TI[1], disponível no JDK da Oracle. Assim, quando um desses objetos fosse alterado, um componente auxiliar do Interpretador verificaria o estado anterior do objeto e o que essa alteração provocou para determinar qual a operação de animação a ser utilizada para atualizar a tela da Camada Gráfica. Dessa forma, um trabalho poderia ser feito para verificar como a Java HotSpot VM poderia ser empregada como Interpretador do IGED.

Uma desvantagem dessa abordagem, como foi mencionada, seria a perda da portabilidade, visto que o acesso às notificações geradas pela alteração desses objetos seria feita através de códigos nativos em C/C++. Outras formas de utilizar a Java HotSpot VM como Interpretador do IGED, sem que para isso seja necessário utilizar métodos nativos, poderiam ser investigadas.

Sendo assim, a utilização de uma implementação própria de um Interpretador com uma máquina virtual para o IGED foi escolhida porque não é necessária uma máquina de propósitos gerais, como uma implementação da JVM, como Interpretador do IGED. Por isso foi desenvolvida uma máquina de propósito específico que atenda aos requisitos do IGED e

[1] A documentação está disponível em http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

que possa ser mais fácil de ser alterada e adaptada com a adição de novas funcionalidades resultantes do estabelecimento de novos requisitos.

A utilização de uma implementação de código aberto com a sua adaptação ao IGED demandaria um estudo do código-fonte que seria mais complexo do que o código-fonte disponível do Interpretador desenvolvido neste trabalho. Com a utilização de uma implementação da JVM sem alterar o seu código-fonte e com a adição de funcionalidades específicas para o IGED, a portabilidade da ferramenta poderia ser comprometida pelos motivos mencionados anteriormente.

Capítulo 5

Avaliação Experimental

Neste capítulo, é detalhado como o Interpretador foi aplicado ao IGED. O IGED foi o estudo de caso usado para validar as funcionalidades do Interpretador desenvolvido neste trabalho por meio de experimentos, onde foram executados vários algoritmos de estruturas de dados que mostram as características que o Interpretador forneceu à ferramenta IGED. Esse experimento com os códigos dos algoritmos e os resultados das execuções desses códigos por meio do IGED são mostrados na seção 5.2.

5.1 Estudo de Caso

Anteriormente, as estruturas de dados do IGED possuíam as suas visualizações geradas por um interpretador bastante simples. Esse interpretador executava códigos em uma linguagem de baixo nível baseada em pilha, assim como a linguagem Oolong. As atividades no IGED eram definidas através de arquivos XML contendo código nessa linguagem, onde *tags* XML faziam a descrição da atividade e separavam os códigos de inicialização das estruturas e o código de manipulação para resolver um problema. No código fonte 5.1 é mostrado um exemplo de uma atividade de inversão de uma lista encadeada com códigos nessa linguagem.

Código Fonte 5.1: Inversão de Lista

```
1 <atividade>
2   <metadado>
3     <area>Lista</area>
4     <autor>Gilberto Farias de Sousa</autor>
5     <id>1</id>
```

```
6       <titulo>Inversao de lista encadeada.</titulo>
7     </metadado>
8     <codInicializacao>CREATE_REF LIST l
9   CREATE_STRUCT LIST
10  WRITE_REF
11  END_COMMAND
12  CREATE_REF NODE n
13  CREATE_STRUCT NODE
14  WRITE_REF
15  END_COMMAND
16  READ_REF n
17  WRITE_INFO 30
18  END_COMMAND
19  READ_REF l
20  WRITE_INFO 1
21  END_COMMAND
22  READ_REF n
23  READ_REF l
24  READ_REF_FIELD INIT
25  WRITE_REF_FIELD NEXT
26  END_COMMAND
27  READ_REF l
28  READ_REF n
29  WRITE_REF_FIELD INIT
30  END_COMMAND
31  READ_REF n
32  CREATE_STRUCT NODE
33  WRITE_REF
34  END_COMMAND
35  READ_REF n
36  WRITE_INFO 20
37  END_COMMAND
38  READ_REF l
39  WRITE_INFO 2
40  END_COMMAND
41  READ_REF n
42  READ_REF l
43  READ_REF_FIELD INIT
44  WRITE_REF_FIELD NEXT
45  END_COMMAND
46  READ_REF l
47  READ_REF n
48  WRITE_REF_FIELD INIT
49  END_COMMAND
50  READ_REF n
51  CREATE_STRUCT NODE
52  WRITE_REF
```

```
53 END_COMMAND
54 READ_REF n
55 WRITE_INFO 10
56 END_COMMAND
57 READ_REF l
58 WRITE_INFO 3
59 END_COMMAND
60 READ_REF n
61 READ_REF l
62 READ_REF_FIELD INIT
63 WRITE_REF_FIELD NEXT
64 END_COMMAND
65 READ_REF l
66 READ_REF n
67 WRITE_REF_FIELD INIT
68 DELETE_REF n
69 END_COMMAND</codInicializacao>
70    <codSolucao>CREATE_REF NODE t
71 END_COMMAND
72 CREATE_REF NODE n
73 READ_REF l
74 READ_REF_FIELD INIT
75 WRITE_REF
76 END_COMMAND
77 CREATE_REF NODE r
78 END_COMMAND
79 READ_REF t
80 READ_REF n
81 READ_REF_FIELD NEXT
82 WRITE_REF
83 END_COMMAND
84 READ_REF n
85 READ_REF r
86 WRITE_REF_FIELD NEXT
87 END_COMMAND
88 READ_REF r
89 READ_REF n
90 WRITE_REF
91 END_COMMAND
92 READ_REF n
93 READ_REF t
94 WRITE_REF
95 END_COMMAND
96 READ_REF t
97 READ_REF n
98 READ_REF_FIELD NEXT
99 WRITE_REF
```

```
100 END_COMMAND
101 READ_REF n
102 READ_REF r
103 WRITE_REF_FIELD NEXT
104 END_COMMAND
105 READ_REF r
106 READ_REF n
107 WRITE_REF
108 END_COMMAND
109 READ_REF n
110 READ_REF t
111 WRITE_REF
112 END_COMMAND
113 READ_REF t
114 READ_REF n
115 READ_REF_FIELD NEXT
116 WRITE_REF
117 END_COMMAND
118 READ_REF n
119 READ_REF r
120 WRITE_REF_FIELD NEXT
121 END_COMMAND
122 READ_REF r
123 READ_REF n
124 WRITE_REF
125 END_COMMAND
126 READ_REF l
127 READ_REF n
128 WRITE_REF_FIELD INIT
129 END_COMMAND
130 DELETE_REF r
131 DELETE_REF n
132 DELETE_REF t
133 END_COMMAND</codSolucao>
134   <descricao>Inverta a lista l inicializada abaixo.</descricao>
135 </atividade>
```

As instruções nessa linguagem que aparecem no código fonte 5.1 são as seguintes:

- CREATE_REF: criação de referências;
- CREATE_STRUCT: criação de estruturas;
- WRITE_REF: escrita de referências a estruturas;

- WRITE_REF_FIELD: escrita de campos de referências de estruturas;
- WRITE_INFO: escrita de campos contendo a informação de um nó;
- READ_REF: leitura de referências;
- READ_REF_FIELD: leitura de campos de referências de estruturas;
- END_COMMAND: limpeza de pilha;
- DELETE_REF: remoção de referências a estruturas.

Além dessas, também havia instruções de criação, de carregamento, de escrita e de remoção de variáveis inteiras: CREATE_INT, LOAD_INT, WRITE_INT e DELETE_INT respectivamente, bem como instruções específicas para estruturas do tipo vetor e do tipo árvore binária.

Embora códigos definidos nessa linguagem gerem as saídas gráficas correspondentes através da Camada Gráfica, é possível notar no código fonte 5.1 que várias instruções se repetem. Isso porque não há nessa linguagem estruturas de controle de laços de repetição, já que não há instruções de desvio condicional e incondicional e também não há instruções aritméticas. Por esses motivos, ela tem pouca utilidade para o ensino de programação, algoritmos e estruturas de dados, embora tenha sido útil para o desenvolvimento do componente Abstração Gráfica do IGED.

Com o Interpretador desenvolvido neste trabalho foram adicionados vários tipos de instruções à ferramenta IGED, que podem ser consultadas no apêndice A. É possível executar códigos com instruções de desvio condicional e incondicional, operações lógicas e aritméticas, definição e execução de métodos com passagens de parâmetros por valor e por referência, definição de classes com herança e composição, criação de objetos com essas classes e acesso aos componentes dessas classes(campos e métodos). Campos e métodos podem ser estáticos e não estáticos. Também é possível executar métodos de uma classes que estejam definidas em arquivos separados.

5.2 Experimento

As estruturas de dados de interesse para a visualização implementadas foram as estruturas de vetor, lista e árvore, que são representadas como objetos como quaisquer outros dentro das estruturas internas da VM do Interpretador ao longo da execução de um programa. Quando ocorre uma atualização ou leitura de seus dados, o Interpretador invoca rotinas de atualização gráfica do componente Abstração Gráfica para a visualização das estruturas de dados.

As estruturas de dados que foram implementadas estão representadas através do diagrama de classes da figura 5.3. A classe *List*, que corresponde a uma estrutura de dados do tipo de lista, possui os campos *init*, que é a referência ao primeiro nó da lista e o campo *size*, que armazena o tamanho da lista. A classe *Vector* representa uma estrutura do tipo vetor e possui os campos *pos*, a posição (ou índice) do elemento do vetor a ser manipulado pelo Interpretador, o campo *size*, que contém o tamanho do vetor e o campo *data*, que contém os dados do vetor. A estrutura do tipo árvore está representada pela classe *BinaryTree*, que é basicamente uma árvore binária. Ela possui os campos *root*, que aponta para a raiz da árvore e o campo *size*, que armazena o tamanho da árvore.

As classes *NodeList* e *NodeTree* representam respectivamente os nós de uma lista e de uma árvore. *NodeList* possui um campo *info*, que contém o dado que o nó armazena e uma referência *next* ao próximo elemento da lista. A classe *NodeTree* possui os campos *left-child*: referência ao filho esquerdo; *rightchild*: referência ao filho direito; *info*: a informação contida no nó; e *height*: a altura do nó.

Na figura 5.4, é mostrada a representação interna da VM do Interpretador do IGED para a estrutura do tipo lista. Os objetos dessas estruturas residem no *heap* enquanto que as referências a esses objetos são parte da memória local do *stackframe* de um método.

5.2.1 Execução do Experimento

Foram executados algoritmos de manipulação de vetores, listas e árvores. Os algoritmos de vetores foram o *bubblesort*, o *insertionsort* e o *mergesort*. Já os algoritmos de lista executam códigos que incluem inserção, remoção e inversão de uma lista. Os algoritmos de árvores

Figura 5.1: Estruturas de Dados implementadas

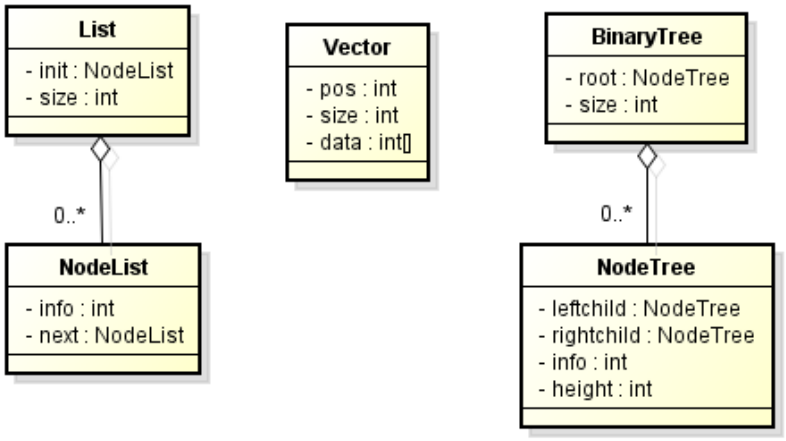
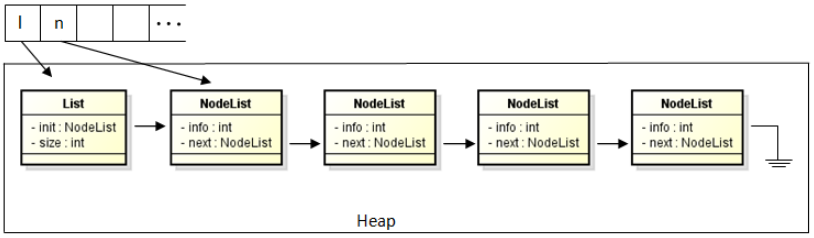


Figura 5.2: Exemplo de como a estrutura do tipo lista fica disposta na VM do Interpretador



fazem inserção e busca de um nó, como também percursos para calcular a altura dos nós durante uma inserção.

Os algoritmos foram implementados na linguagem de montagem Oolong. Para fins de simplificação, neste capítulo são mostrados os códigos equivalentes aos que foram implementados em uma linguagem de alto nível similar à linguagem Java. Os códigos da execução desse experimento que foram implementados em Oolong podem ser consultados no apêndice C. Neste capítulo, são mostrados apenas os códigos das classes principais da execução do experimento para as estruturas de vetor, lista e árvore. As classes auxiliares que contém os algoritmos implementados para os experimentos das estruturas de vetor, lista e árvore, que são respectivamente as classes *VetorUtils*, *ListaUtils* e *TreeUtils* podem ser consultadas no apêndice B.

A classe *Main* seguinte executa os algoritmos de vetor chamando os seus respectivos métodos da classe *VetorUtils*. Nessa classe, são definidos três vetores como campos estáticos. Antes da execução de cada algoritmo de ordenação, no método *main* o vetor a ser ordenado

é passado como parâmetro por referência ao método *preencherVetor* da classe *Main*, que preenche o vetor com quinze elementos.

O método *main* chama os métodos dos algoritmos de ordenação ao passar os vetores *vetor1*, *vetor2* e *vetor3* por referência aos métodos *bubblesort*, *insertionsort* e *mergesort* respectivamente. Aos três métodos também são passados parâmetros por valor que correspondem ao tamanho do vetor a ser ordenado. No método *mergesort*, é passado um parâmetro adicional que informa ao algoritmo que o elemento que ele deve considerar como primeiro elemento do vetor é o elemento 1, devido ao fato de que esse método é recursivo. O vetor inicial não ordenado possui os seguintes elementos, nessa ordem: 51, 100, 20, 93, 11, 65, 92, 76, 61, 40, 39, 52, 10, 62, 91.

Código Fonte 5.2: Classe Main que executa os algoritmos de vetores

```
1  class Main{
2
3          static Vector vetor1;
4          static Vector vetor2;
5          static Vector vetor3;
6
7          static void main() {
8
9                  vetor1 = {51, 100, 20, 93, 11, 65, 92, 76, 61, 40, 39, 52, 10, 62, 91};
10                 VetorUtils.bubblesort(vetor1, 15);
11
12                 vetor2 = {51, 100, 20, 93, 11, 65, 92, 76, 61, 40, 39, 52, 10, 62, 91};
13                 VetorUtils.insertionsort(vetor2, 15);
14
15                 vetor3 = {51, 100, 20, 93, 11, 65, 92, 76, 61, 40, 39, 52, 10, 62, 91};
16                 VetorUtils.mergesort(vetor3, 1, 15);
17
18         }
19
20 }
```

Na figura 5.3, é mostrado esse vetor sendo ordenado pelo método *bubblesort*, já na figura 5.4 é ilustrado o vetor com os mesmos elementos já ordenado pelo método *insertionsort*. Essas imagens são ilustrações da tela do IGED.

Figura 5.3: Vetor sendo ordenado pelo algoritmo Bubble Sort

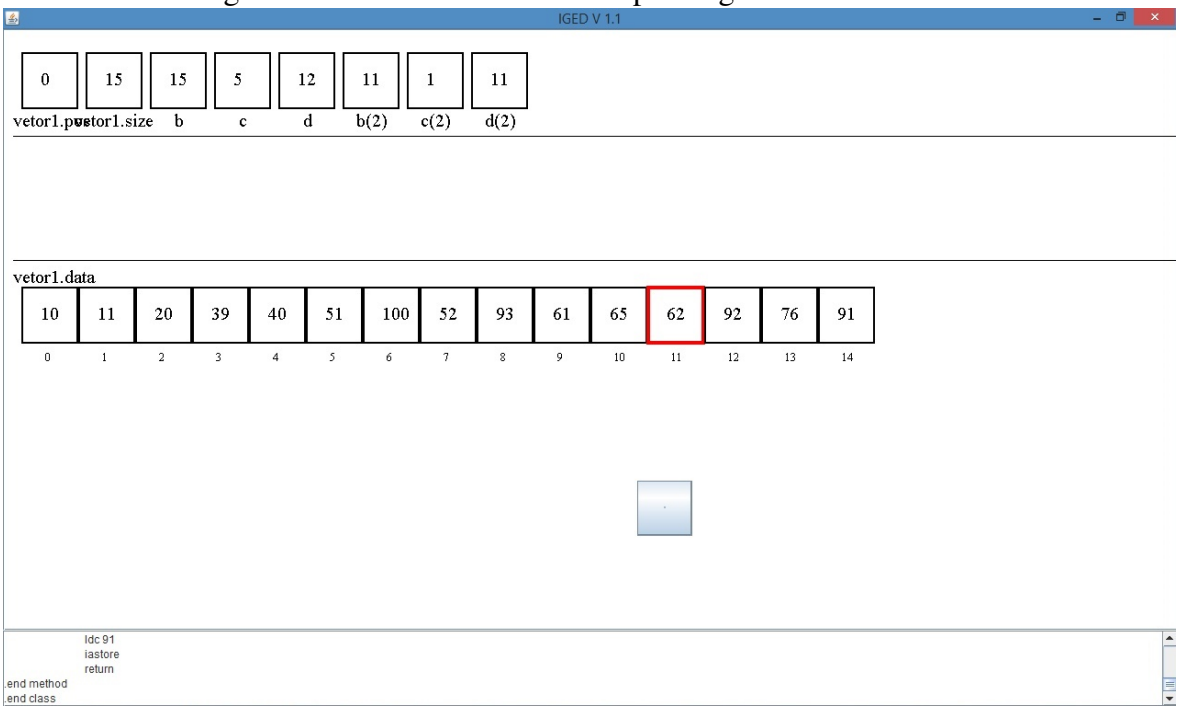
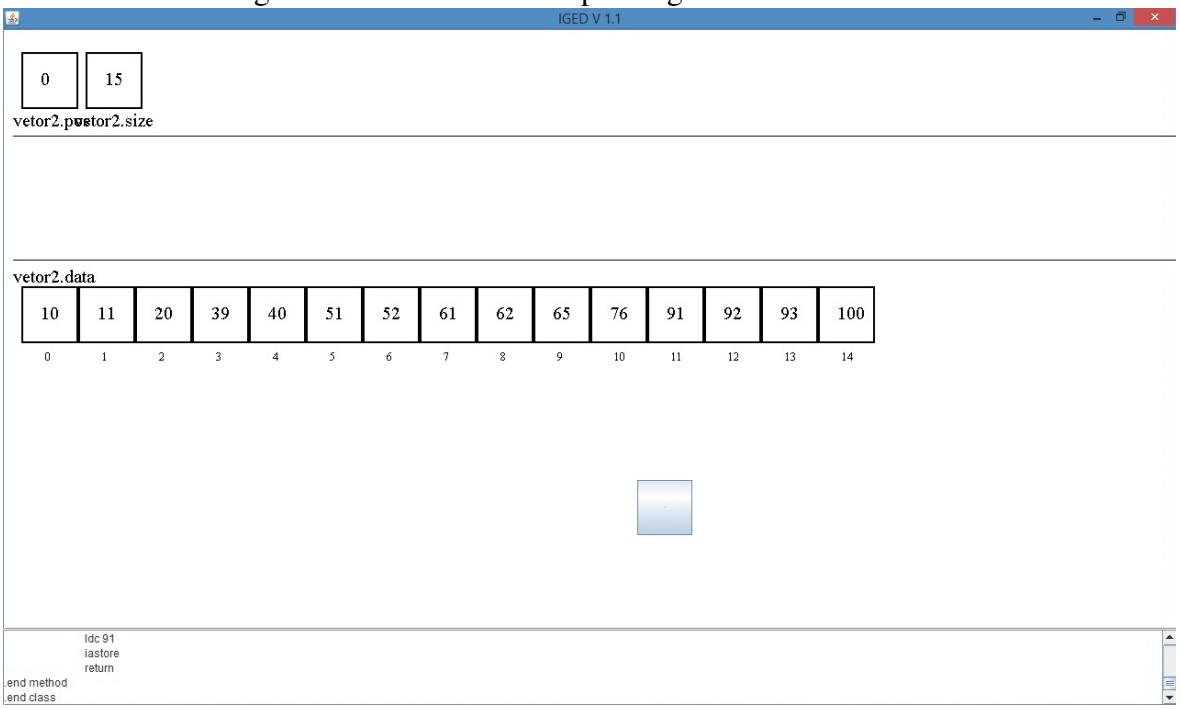
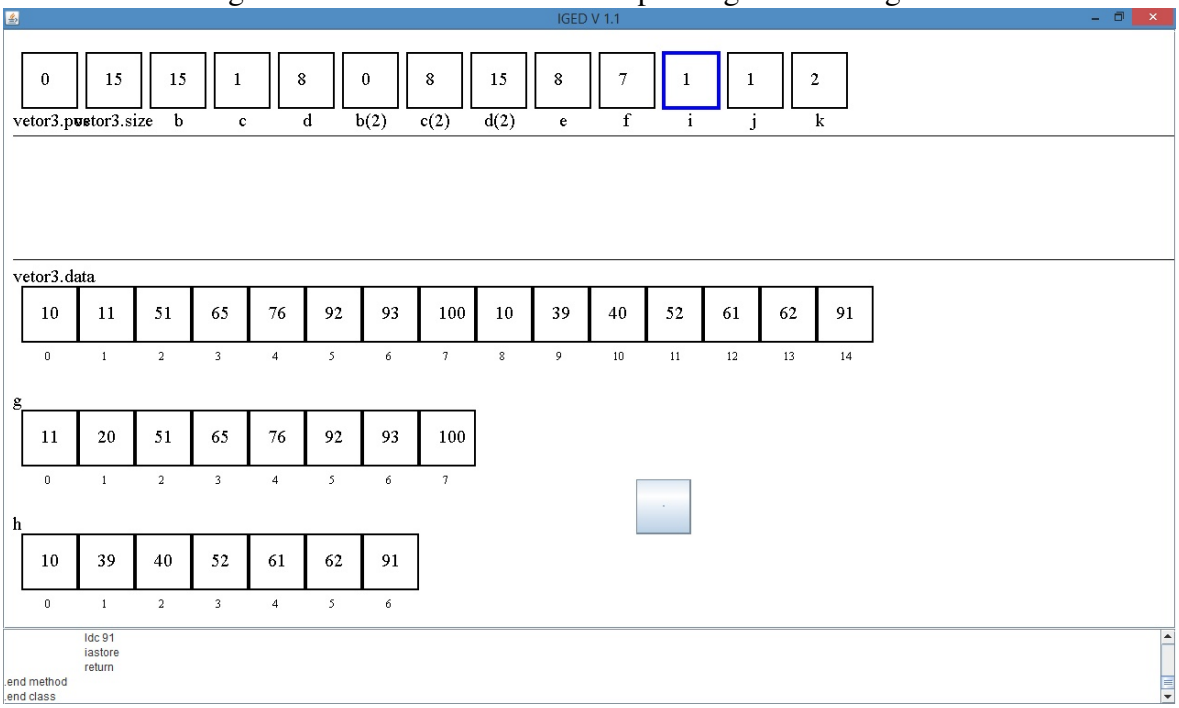


Figura 5.4: Vetor ordenado pelo algoritmo Insertion Sort



Na figura 5.5, é ilustrada na tela do IGED o *vetor3* com os mesmos elementos no processo final de ordenação pelo método *mergesort*, na qual pode-se notar que os dois últimos subvetores criados pelo método através de suas chamadas recursivas estão sendo intercalados ao vetor original.

Figura 5.5: Vetor sendo ordenado pelo algoritmo Merge Sort



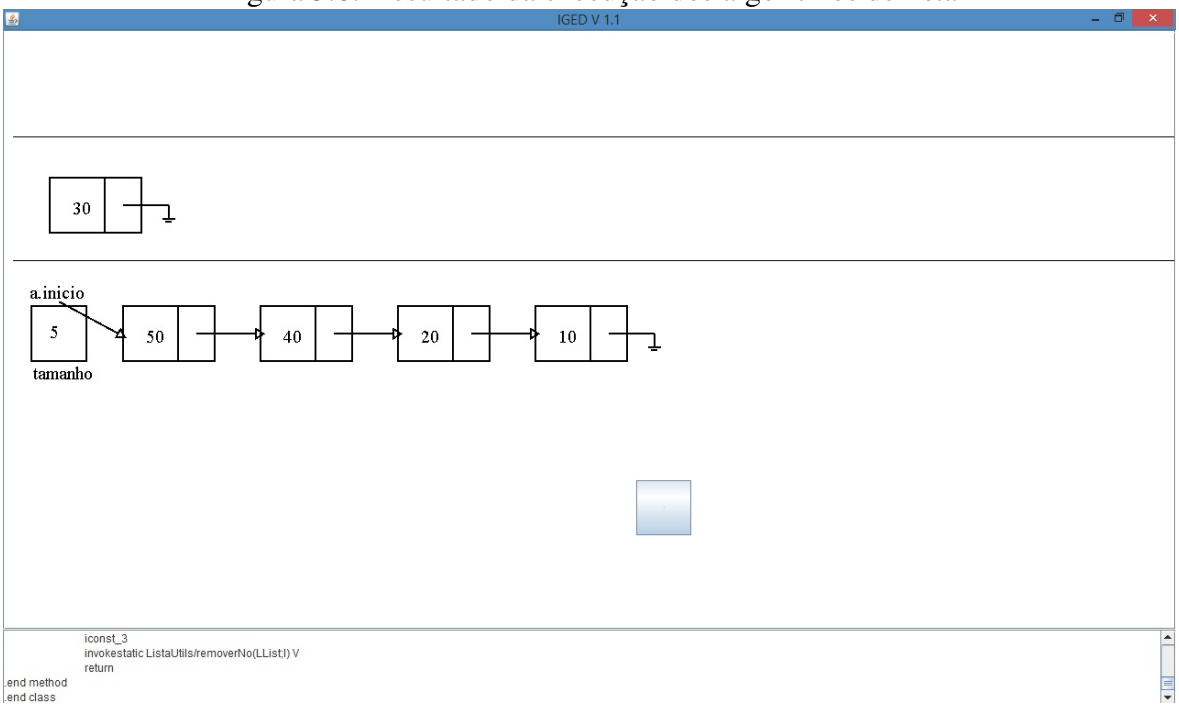
A classe *Main* do código 5.3 insere cinco elementos de valores 10, 20, 30, 40 e 50 numa lista encadeada, através do método *preencherLista* da classe *ListaUtils* e inverte essa lista com o método *inverterLista* também da classe *ListaUtils*. Após, o terceiro elemento dessa lista é removido da lista invertida. Em ambos os métodos, a referência da lista, definida como variável local do método *main*, é passada como parâmetro. A tela do IGED, após a execução dos algoritmos de lista, é mostrada na figura 5.6.

Código Fonte 5.3: Classe Main que executa os algoritmos de lista

```
1  class Main{
2
3      static void main() {
4
5                  List a = new List();
6
7                  ListaUtils.preencherLista(a, 5);
```

```
8
9                   ListaUtils.inverterLista(a);
10
11                  ListaUtils.removerNo(a, 3);
12
13          }
14
15  }
```

Figura 5.6: Resultado da execução dos algoritmos de lista



Na classe *Main* do código 5.4, é criada uma árvore e em seguida são inseridos elementos que formarão os nós dessa árvore. Quando cada nó é inserido na árvore, o método *inserir* da classe *TreeUtils* chamada um método dessa mesma classe que atualiza a altura dos nós. O método que atualiza a altura dos nós é um método recursivo que faz um percurso pré-ordem na árvore, no qual em cada visita é chamado um método que calcula a altura do nó que está sendo visitado, que também é recursivo. Após a inserção dos nós, o método *buscar* da classe *TreeUtils* é invocado para buscar o nó com o valor 41. Esse método retorna o nó buscado que é atribuído à variável *b* do método *main* da classe *Main*. Na figura 5.7, a tela do IGED após a execução dos algoritmos de árvore é exibida.

Código Fonte 5.4: Classe Main que executa os algoritmos de árvore

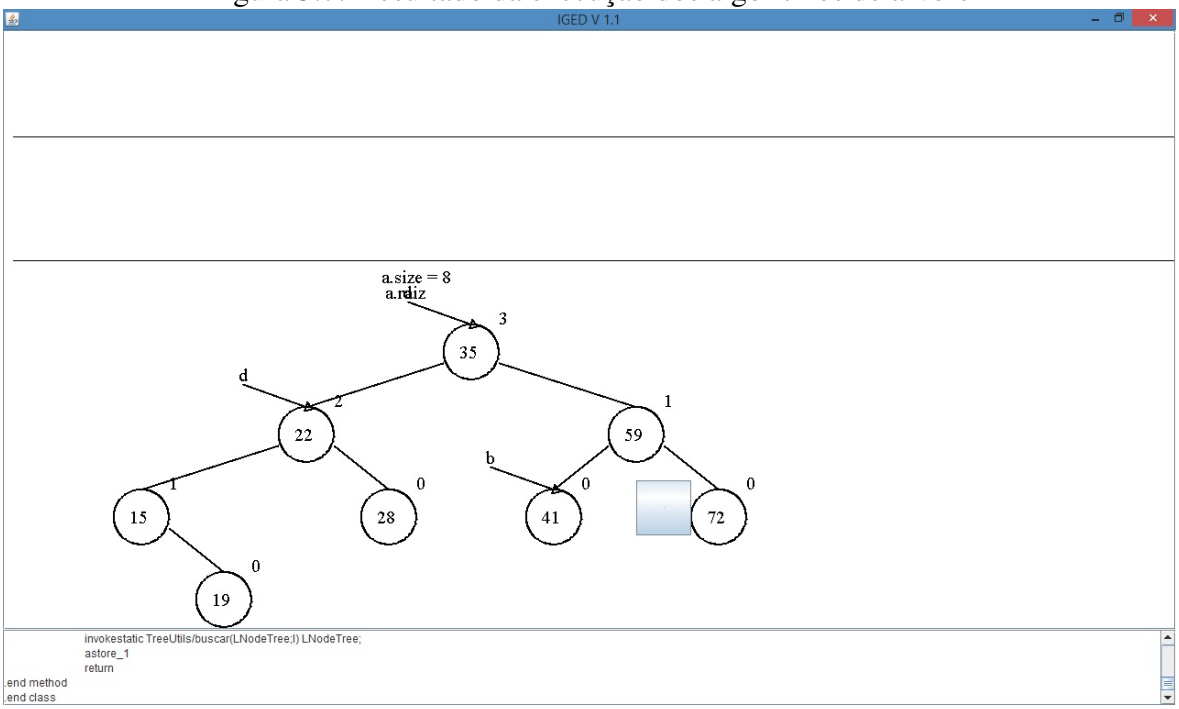
```
1  class Main{
2
3          static void main() {
4
5                  BinaryTree a = new BinaryTree();
6
7                  TreeUtils.inserir(a, 35);
8
9                  TreeUtils.inserir(a, 22);
10
11                 TreeUtils.inserir(a, 59);
12
13                 TreeUtils.inserir(a, 15);
14
15                 TreeUtils.inserir(a, 19);
16
17                 TreeUtils.inserir(a, 28);
18
19                 TreeUtils.inserir(a, 72);
20
21                 TreeUtils.inserir(a, 41);
22
23                 b = TreeUtils.buscar(a.root, 41);
24
25         }
26
27 }
```

5.2.2 Conclusão

Como o Interpretador executou todos os códigos mostrados neste experimento, foi verificado que ele executa algoritmos de forma automática sobre as estruturas de dados implementadas no IGED, que são as estruturas de vetor, lista e árvore. Além disso, também é possível verificar que ele forneceu ao IGED a possibilidade da execução de códigos com as seguintes características:

- Desvio condicional e incondicional;
- Operações lógicas e aritméticas;
- Definição e execução de métodos com passagens de parâmetros por valor e por referência;

Figura 5.7: Resultado da execução dos algoritmos de árvore



- Execução de métodos recursivos;
- Definição de classes, com campos e métodos;
- Criação de objetos;
- Acesso a campos e métodos de uma classe;
- Acesso a código de classes em arquivos separados.

Capítulo 6

Conclusão

A análise comparativa entre uma implementação da JVM e o Interpretador do IGED apresentou algumas vantagens e desvantagens da utilização dos dois possíveis interpretadores para a ferramenta IGED. Pôde-se verificar, com isso, que o uso de uma implementação da JVM seria melhor em aspectos relacionados à segurança, pois restringe o acesso somente a locais permitidos de memória e à possibilidade da utilização de linguagens de propósitos específicos. Também, com uma implementação da JVM, é possível que um programa possa ser executado a partir de várias fontes de dados, tirar proveito do recurso de *garbage collector* para as estruturas de dados da memória da VM, da maior estabilidade que ela oferece e do melhor desempenho que ela possui. Com isso, também não seria necessário a implementação de uma linguagem de alto nível para o IGED.

As desvantagens dessa abordagem incluem o fato de que uma implementação da JVM não fornece recursos de exibição gráfica dos seus componentes internos. Com isso, seria necessário a utilização de códigos nativos em C/C++ para o acesso a esses componentes, o que diminuiria a portabilidade da ferramenta. Além disso, tradutores podem ser implementados de forma mais simples e o tempo necessário para a utilização do interpretador para os propósitos do IGED poderia ser menor, porque o interpretador já está implementado e as funcionalidades que uma implementação da JVM deve possuir para colaborar com o projeto IGED ainda não são muito bem conhecidas.

Considerando uma implementação da JVM como interpretador do IGED, o componente Tradutor poderia traduzir o código fonte do usuário da linguagem de alto nível do IGED para a linguagem binária de *bytecodes* Java para que possa ser executado pela JVM. Outra alternativa para o componente Tradutor, com o uso dessa abordagem, seria deixá-lo na forma como está para o Interpretador proposto neste trabalho, isto é, traduzindo o código de alto nível para a linguagem de montagem Oolong, e utilizar uma ferramenta que possa gerar os *bytecodes* na forma binária, como Jasmin, para serem executados pela JVM. Ainda uma outra alternativa seria não produzir nenhum componente Tradutor para o IGED. Nesse caso, a linguagem de programação Java seria utilizada como a linguagem de alto nível do IGED e o compilador da JVM seria o próprio Tradutor.

Com esse trabalho, pode-se verificar como esse interpretador atende aos requisitos da ferramenta IGED e como ele pode ser útil para atender à proposta pedagógica da ferramenta. Foi demonstrado que, com ele, é possível fornecer funcionalidades de exibição gráfica das estruturas de dados de forma automática ao interpretar o código fornecido por um usuário. Além dessa utilidade, entretanto, esse interpretador pode possuir outras utilidades no ensino e aprendizagem de outras disciplinas de cursos de Computação, já que ela pode ser empregada tanto para o propósito de Estruturas de Dados, como também para o ensino de técnicas e conceitos de programação, análise de algoritmos, compiladores e arquitetura de computadores.

Como o Interpretador foi implementado, sua implementação foi justificada e foram mostradas características que o Interpretador forneceu à ferramenta IGED e que são úteis em disciplinas de programação, algoritmos e estruturas de dados, os objetivos deste trabalho foram atingidos. No estado atual, com o Interpretador desenvolvido neste trabalho, a ferramenta IGED pode gerar as visualizações gráficas com a implementação de vários algoritmos utilizados em disciplinas de programação e estruturas de dados. Porém, para que ela possa ser empregada de forma efetiva nessas disciplinas, é necessário que haja um componente Tradutor para traduzir os códigos em linguagem de alto nível, já que nessas disciplinas, os conceitos de programação e estruturas de dados são ensinados e aprendidos usando linguagens de programação de alto nível. Linguagens de programação de baixo nível geralmente

são utilizadas em disciplinas de Arquitetura de Computadores.

Como trabalhos futuros, tem-se o desenvolvimento de um ou mais tradutores para esse Interpretador e a aplicação do IGED em salas de aula, para avaliar e acompanhar a aprendizagem dos alunos com o uso da ferramenta. O Interpretador pode possibilitar a adição de componentes à ferramenta IGED, como um componente que exiba gráficos da execução dos algoritmos, que mostre os processos de tradução e de visualização das estruturas internas da máquina virtual do Interpretador ao longo da execução dos códigos. O Interpretador do IGED também pode ser cada vez mais aprimorado para o ensino, com a adição de funcionalidades que permitam a execução por *stepping* com *breakpoints*, execução de códigos com paralelismo, dentre outras.

Bibliografia

[Aho, Sethi e Ullman 1995]AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compiladores: Princípios, Técnicas e Ferramentas*. Rio de Janeiro, RJ, Brasil: LTC, 1995. 344 p.

[Amorim e Resende 1993]AMORIM, R. V.; RESENDE, P. J. de. Compreensão de Algoritmos através de Ambientes Dedicados a Animação. In: *XX SEMISH*. [S.l.: s.n.], 1993. p. 32.

[Brown 1988]BROWN, M. H. Exploring Algorithms Using Balsa-II. *IEEE Computer*, n. 5, 1988.

[Brown 1991]BROWN, M. H. Zeus: A System for Algorithm Animation and Multi-View Editing. *IEEE Workshop on Visual Languages*, 1991.

[Brown e Sedgewick 1984]BROWN, M. H.; SEDGEWICK, R. A System for Algorithm Animation. *ACM SIGGRAPH Computer Graphics*, n. 3, 1984.

[Cattaneoa, Faruolo e Petrillo 2004]CATTANEOA, G.; FARUOLO, P.; PETRILLO, U. F. JIVE: Java Interactive software Visualization Environment. In: *IEEE Symposium on Visual Languages and Human Centric Computing*. [S.l.: s.n.], 2004. p. 41–43.

[Celes, Cerqueira e Rangel 2004]CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução a Estruturas de Dados*. Rio de Janeiro, RJ, Brasil: Campus, 2004. 294 p. ISBN 85-352-1228-0.

[Cormen et al. 2002]CORMEN, T. H. et al. *Algoritmos: Teoria e Prática*. Rio de Janeiro, RJ, Brasil: Campus, 2002. 916 p. ISBN 85-352-0926-3.

[Cross et al. 2009]CROSS, J. H. et al. Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches. *ACM Transactions on Computing Education*, n. 2, 2009.

[Deitel e Deitel 2010]DEITEL, P.; DEITEL, H. *Java: Como Programar*. São Paulo, SP, Brasil: Pearson, 2010. 1114 p. ISBN 978-85-7605-194-7.

[Drozdek 2005]DROZDEK, A. *Estrutura de Dados e Algoritmos em C++*. São Paulo, SP, Brasil: Thomson, 2005. 579 p. ISBN 85-221-0259-3.

[Engel 1999]ENGEL, J. *Programmming for the Java Virtual Machine*. [S.l.]: Addison Wesley, 1999. 512 p. ISBN 0-201-30972-6.

[Ferradin e Stephani 2005]FERRADIN, M.; STEPHANI, S. L. Ferramenta para o ensino de Programação via Internet. In: *Anais do SULCOMP*. [S.l.: s.n.], 2005.

[Filho et al. 2012]FILHO, G. F. S. et al. Tutor hipermídia baseado no modelo de autoria NCM para o Interpretador Gráfico de Estrutura de Dados . In: *Anais do XX Workshop sobre Educação em Computação*. [S.l.: s.n.], 2012.

[Garcia e Rezende 1997]GARCIA, I. C.; REZENDE, P. J. de. Astral: Um Ambiente para Ensino de Estruturas de Dados através de Animações de Algoritmos. *Revista Brasileira de Informática na Educação*, n. 1, 1997.

[Guedes 2006]GUEDES, G. T. A. *UML: Uma abordagem prática*. São Paulo, SP, Brasil: Novatec, 2006.

[Madeira et al. 2012]MADEIRA, M. F. et al. Odin-ambiente web de apoio ao ensino de estruturas de dados lista encadeada. *Anais do SULCOMP*, 2012.

[Malone et al. 2009]MALONE, B. et al. Pedagogically Effective Effortless Algorithm Visualization with a PCIL . In: *Frontiers in Education Conference, 2009. FIE '09. 39th IEEE*. [S.l.: s.n.], 2009. p. 1–6.

[Naps 2005]NAPS, T. L. JHAVÉ: Supporting Algorithm Visualization. *IEEE Computer Graphics and Applications*, n. 5, 2005.

[Netto 2010]NETTO, D. Análise dos dados do questionário da disciplina Estrutura de Dados. Departamento de Ciências Exatas, UFPB, 2010.

[Netto et al. 2011]NETTO, D. P. da S. et al. Desenvolvimento de um Interpretador de Comandos e Avaliador Gráfico para o Ensino de Estrutura de Dados. In: *Anais do XIX Workshop sobre Educação em Computação*. [S.l.: s.n.], 2011.

[Oliveira 2008]OLIVEIRA, U. de. *Programando em C: Fundamentos*. Rio de Janeiro, RJ, Brasil: Ciência Moderna, 2008. ISBN 978-85-7393-659-9.

[Papadimitriou 2007]PAPADIMITRIOU, S. Scientific programming with Java classes supported with a scripting interpreter. *IET Software*, n. 2, 2007.

[Parr 2007]PARR, T. *The Definitive Antlr Reference*. United States of America: The Pragmatic Bookshelf, 2007. 356 p. ISBN 978-09787392-4-9.

[Parr 2010]PARR, T. *Language Implementation Patterns*. United States of America: The Pragmatic Bookshelf, 2010. 369 p. ISBN 978-1-934356-45-6.

[Prodanov e Freitas 2013]PRODANOV, C. C.; FREITAS, E. C. de. *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. Novo Hamburgo, RS, Brasil: Universidade Feevale, 2013. 277 p. ISBN 978-85-7717-158-3.

[Rocha 1991]ROCHA, H. V. da. Representações Computacionais Auxiliares ao Entendimento de Conceitos de Programação. *Unicamp*, 1991.

[Santiago e Dazzi 2004]SANTIAGO, R. de; DAZZI, R. L. S. Ferramenta de apoio ao ensino de algoritmos. In: *Anais do Seminário de Computação - SEMINCO*. [S.l.: s.n.], 2004.

[Santos e Costa 2005]SANTOS, R. P. dos; COSTA, H. A. X. Tbc-aed: Um software gráfico para apresentação de algoritmos e estruturas de dados aos iniciantes em computação e informática. In: *Anais do I Congresso de Computaçã do Sul do Mato Grosso*. [S.l.: s.n.], 2005.

[Simões et al. 2013]SIMÕES, P. W. T. de A. et al. Spyke-ferramenta de apoio ao ensino de pilhas e filas. *Anais do SULCOMP*, 2013.

[Sipser 2007]SIPSER, M. *Introdução à Teoria da Computação*. São Paulo, SP, Brasil: Thomson, 2007. 459 p. ISBN 978-85-221-0499-4.

[Stasko 1990]STASKO, J. T. Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, n. 9, 1990.

[Tremblay e Sorenson 1985]TREMBLAY, J. P.; SORENSON, P. G. *The Theory and Practice of Compiler Writing*. United States of America: Mc-Graw Hill, 1985. 796 p. ISBN 0-07-065161-2.

[Veras et al. 2010]VERAS, R. M. S. et al. Ferramenta computacional para o ensino de algoritmos de ordenação. In: *Anais do Simpósio Brasileiro de Informática na Educação*. [S.l.: s.n.], 2010.

[Werneck 2006]WERNECK, V. R. Sobre o processo de construção do conhecimento: O papel do ensino e da pesquisa. *Ensaio: Avaliação e Políticas Públicas em Educação*, n. 21, 2006. ISSN 0104-4036.

Apêndice A

Instruções do Interpretador por *opcode*

Neste apêndice estão os *opcodes* das instruções[1] que são suportadas pelo Interpretador, com os seus respectivos valores em hexadecimal e decimal.

Tabela A.1: Opcodes das instruções

Opcode	Hexadecimal	Decimal
nop	00	0
aconst_null	01	1
iconst_m1	02	2
iconst_0	03	3
iconst_1	04	4
iconst_2	05	5
iconst_3	06	6
iconst_4	07	7
iconst_5	08	8
ldc	12	18
iload	15	21
aload	19	25
iload_0	1a	26
iload_1	1b	27

[1]Para mais detalhes sobre as instruções representadas por esses opcodes, acesse http://homepages.inf.ed.ac.uk/kwxm/JVM/home.html

Opcode	Hexadecimal	Decimal
iload_2	1c	28
iload_3	1d	29
aload_0	2a	42
aload_1	2b	43
aload_2	2c	44
aload_3	2d	45
iaload	2e	46
istore	36	54
astore	3a	58
istore_0	3b	59
istore_1	3c	60
istore_2	3d	61
istore_3	3e	62
astore_0	4b	75
astore_1	4c	76
astore_2	4d	77
astore_3	4e	78
iastore	4f	79
pop	57	87
pop2	58	88
dup	59	89
iadd	60	96
isub	64	100
imul	68	104
idiv	6c	108
irem	70	112
ineg	74	116
iand	7e	126
ior	80	128
ixor	82	130
iinc	84	132
ifeq	99	153
ifne	9a	154

Opcode	Hexadecimal	Decimal
iflt	9b	155
ifge	9c	156
ifgt	9d	157
ifle	9e	158
if_icmpeq	9f	159
if_icmpne	a0	160
if_icmplt	a1	161
if_icmpge	a2	162
if_icmpgt	a3	163
if_icmple	a4	164
if_acmpeq	a5	165
if_acmpne	a6	166
goto	a7	167
ireturn	ac	172
areturn	b0	176
return	b1	177
getstatic	b2	178
putstatic	b3	179
getfield	b4	180
putfield	b5	181
invokevirtual	b6	182
invokespecial	b7	183
invokestatic	b8	184
new	bb	187
newarray	bc	188
ifnull	c6	198
ifnonnull	c7	199

Apêndice B

Classes VetorUtils, ListaUtils e TreeUtils utilizadas no Experimento em linguagem de alto nível

Código Fonte B.1: Classe VetorUtils

```
1  class VetorUtils {
2
3          static void swap(int data[], int pos1, int pos2){
4
5                  int x = data[pos1];
6                  data[pos1] = data[pos2];
7                  data[pos2] = x;
8
9          }
10
11         static void bubblesort(Vector v, int n) {
12
13                 for(int i = 0; i < n - 1; i++){
14                         for(int j = n - 1; j > i; j--){
15                                 if (v.data[j] < v.data[j - 1])
16                                         swap(v.data, j, j - 1);
17                         }
18                 }
19
20         }
21
22         static void insertionSort(Vector v, int n) {
23
24                 int i, j, k;
```

```
25
26                  for (i = 1; i < n; i++) {
27
28                          k = v.data[i];
29
30                          j = i - 1;
31
32                          while  (j >= 0 && v.data[j] > k) {
33                                  v.data[j + 1] = v.data[j];
34                                  j--;
35                          }
36
37                          v.data[j + 1] = k;
38
39                  }
40
41
42
43          }
44
45          static void merge(int A[], int p, int q, int r){
46
47          int n1 = q - p + 1;
48          int n2 = r - q;
49          int i, j, k;
50
51          p--;
52
53          int L[] = new int[n1];
54          int R[] = new int [n2];
55
56          for (i = 0; i < n1; i++)
57                  L[i] = A[p + i];
58
59              for (i = 0; i < n2; i++)
60                  R[i] = A[q + i];
61
62          i = 0;
63          j = 0;
64
65          for (k = p; k < r; k++) {
66
67                  if (i == n1 && j < n2) {
68
69                      while (k < r) {
70                              A[k] = R[j];
71                                  k++;
```

```
72                                  j++;
73                          }
74
75              } else if (i < n1 && j == n2){
76
77                          while (k < r) {
78                              A[k] = L[i];
79                                  k++;
80                                  i++;
81                          }
82
83              } else if (i < n1 && j < n2) {
84
85                      if (L[i] <= R[j]){
86                                  A[k] = L[i];
87                                  i++;
88                          } else {
89                                  A[k] = R[j];
90                                  j++;
91                          }
92
93                  }
94
95          }
96
97
98
99      }
100
101     static void mergesort(Vector vet[], int p, int r) {
102
103         if (p < r) {
104                 int q = (p + r) / 2;
105                 mergesort(vet, p, q);
106                 mergesort(vet, q + 1, r);
107                 merge(vet.data, p, q, r);
108         }
109
110     }
111
112
113 }
```

Código Fonte B.2: Classe ListaUtils

```
1  class Lista {
2
```

```
3       static void preencherLista(List a, int b) {
4
5               NodeList c = new NodeList(10);
6               a.init = c;
7               a.size = 1;
8
9               NodeList d = a.init;
10
11              for (int e = 1; e < b; e++){
12                      c = new NodeList((e + 1) * 10);
13                      d.next = c;
14                      a.size++;
15                      d = d.next;
16              }
17
18      }
19
20      static void inverterLista(List a) {
21
22              NodeList b = a.init;
23
24              while(b.next != null)
25                      b = b.next;
26
27              NodeList c = a.init;
28
29              while (c != b) {
30
31                      a.init = c.next;
32                      c.next = b.next;
33                      b.next = c;
34                      c = a.init;
35
36              }
37
38      }
39
40      static void removerNo(List a, int b){
41
42              if (a.init == null)
43                      return;
44
45              if (b > a.size)
46                      return;
47
48              NodeList c = a.init;
49
```

```
50                  if (b == 1){
51                          a.init = c.next;
52                          c.next = null;
53                          return;
54                  }
55
56                  for (int d = 0; d < (b - 2); d++)
57                          c = c.next;
58
59                  NodeList e = c.next;
60                  c.next = c.next.next;
61                  e.next = null;
62
63          }
64
65  }
```

Código Fonte B.3: Classe TreeUtils

```
1   class TreeUtils{
2
3           static void inserir(BinaryTree t, int info) {
4
5                   if (t.root == null)
6                           t.root = new NodeTree(info);
7                   else {
8
9                           NodeTree p = t.root;
10                          NodeTree aux;
11
12                          do {
13
14                                  aux = p;
15
16                                  if (info > aux.info){
17                                          p = aux.rightchild;
18                                          if (p == null){
19                                                  aux.rightchild = new NodeTree(info);
20                                                  atualizarAlturaNos(t.root);
21                                          }
22                                  } else if (info < aux.info){
23                                          p = aux.leftchild;
24                                          if (p == null){
25                                                  aux.leftchild = new NodeTree(info);
26                                                  atualizarAlturaNos(t.root);
27                                          }
28                                  } else
```

```
29                                      return;
30
31                      } while (p != null);
32
33              }
34
35              t.size++;
36
37      }
38
39      static NodeTree buscar(NodeTree n, int info){
40
41              if (n == null)
42                      return null;
43
44              if (info == n.info)
45                      return n;
46
47              if (info > n.info)
48                      return buscar(n.rightchild, info);
49              else
50                      return buscar(n.leftchild, info);
51
52      }
53
54      static int altura(NodeTree n) {
55
56              if (n == null)
57                  return -1;
58              else {
59                  int he = altura(n.leftchild);
60                  int hd = altura(n.rightchild);
61                  if (he < hd)
62                      return hd++;
63                  else
64                      return he++;
65              }
66
67      }
68
69      static void atualizarAlturaNos(NodeTree n){
70
71              if (n == null)
72                      return;
73
74              n.height = altura(n);
75
```

```
76                      atualizarAlturaNos(n.rightchild);
77                      atualizarAlturaNos(n.leftchild);
78
79              }
80
81      }
```

Apêndice C

Classes implementadas em Oolong para a execução do experimento

Código Fonte C.1: Classe principal da execução dos algoritmos da estrutura do tipo vetor

```
1  .class Main
2  .field static vetor1 LVector;
3  .field static vetor2 LVector;
4  .field static vetor3 LVector;
5  .method static main() V
6  .limit locals 1
7      new Vector
8          putstatic Main/vetor1 LVector;
9          getstatic Main/vetor1 LVector;
10         invokestatic Main/preencherVetor(LVector;) V
11         getstatic Main/vetor1 LVector;
12         ldc 15
13         invokestatic VetorUtils/bubblesort(LVector;I) V
14         new Vector
15         putstatic Main/vetor2 LVector;
16         getstatic Main/vetor2 LVector;
17         invokestatic Main/preencherVetor(LVector;) V
18         getstatic Main/vetor2 LVector;
19         ldc 15
20         invokestatic VetorUtils/insertionsort(LVector;I) V
21         new Vector
22         putstatic Main/vetor3 LVector;
23         getstatic Main/vetor3 LVector;
24         invokestatic Main/preencherVetor(LVector;) V
25         getstatic Main/vetor3 LVector;
26         iconst_1
27         ldc 15
```

```
28          invokestatic VetorUtils/mergesort(LVector;II) V
29          return
30  .end method
31  .method static preencherVetor(LVector;) V
32  .limit locals 1
33          aload_0
34          ldc 15
35          invokespecial Vector/<init>(I) V
36          aload_0
37          getfield Vector/data [I;
38          iconst_0
39          ldc 51
40          iastore
41          aload_0
42          getfield Vector/data [I;
43          iconst_1
44          ldc 100
45          iastore
46          aload_0
47          getfield Vector/data [I;
48          iconst_2
49          ldc 20
50          iastore
51          aload_0
52          getfield Vector/data [I;
53          iconst_3
54          ldc 93
55          iastore
56          aload_0
57          getfield Vector/data [I;
58          iconst_4
59          ldc 11
60          iastore
61          aload_0
62          getfield Vector/data [I;
63          iconst_5
64          ldc 65
65          iastore
66          aload_0
67          getfield Vector/data [I;
68          ldc 6
69          ldc 92
70          iastore
71          aload_0
72          getfield Vector/data [I;
73          ldc 7
74          ldc 76
```

```
75          iastore
76          aload_0
77          getfield Vector/data [I;
78          ldc 8
79          ldc 61
80          iastore
81          aload_0
82          getfield Vector/data [I;
83          ldc 9
84          ldc 40
85          iastore
86          aload_0
87          getfield Vector/data [I;
88          ldc 10
89          ldc 39
90          iastore
91          aload_0
92          getfield Vector/data [I;
93          ldc 11
94          ldc 52
95          iastore
96          aload_0
97          getfield Vector/data [I;
98          ldc 12
99          ldc 10
100         iastore
101         aload_0
102         getfield Vector/data [I;
103         ldc 13
104         ldc 62
105         iastore
106         aload_0
107         getfield Vector/data [I;
108         ldc 14
109         ldc 91
110         iastore
111         return
112 .end method
113 .end class
```

Código Fonte C.2: Classe principal da execução dos algoritmos da estrutura do tipo lista

```
1  .class Main
2  .method static main() V
3  .limit locals 1
4           new List
5           astore_0
```

```
6           aload_0
7           invokespecial List/<init>() V
8           aload_0
9           ldc 5
10          invokestatic ListaUtils/preencherLista(LList;I) V
11          aload_0
12          invokestatic ListaUtils/inverterLista(LList;) V
13          aload_0
14          iconst_3
15          invokestatic ListaUtils/removerNo(LList;I) V
16          return
17  .end method
18  .end class
```

Código Fonte C.3: Classe principal da execução dos algoritmos da estrutura do tipo árvore

```
1   .class Main
2   .method static main() V
3   .limit locals 2
4           new BinaryTree
5           astore_0
6           aload_0
7           invokespecial BinaryTree/<init>() V
8           aload_0
9           ldc 35
10          invokestatic TreeUtils/inserir(LBinaryTree;I) V
11          aload_0
12          ldc 22
13          invokestatic TreeUtils/inserir(LBinaryTree;I) V
14          aload_0
15          ldc 59
16          invokestatic TreeUtils/inserir(LBinaryTree;I) V
17          aload_0
18          ldc 15
19          invokestatic TreeUtils/inserir(LBinaryTree;I) V
20          aload_0
21          ldc 19
22          invokestatic TreeUtils/inserir(LBinaryTree;I) V
23          aload_0
24          ldc 28
25          invokestatic TreeUtils/inserir(LBinaryTree;I) V
26          aload_0
27          ldc 72
28          invokestatic TreeUtils/inserir(LBinaryTree;I) V
29          aload_0
30          ldc 41
31          invokestatic TreeUtils/inserir(LBinaryTree;I) V
```

```
32          aload_0
33          getfield BinaryTree/root LNodeTree;
34          ldc 41
35          invokestatic TreeUtils/buscar(LNodeTree;I) LNodeTree;
36          astore_1
37          return
38  .end method
39  .end class
```

Código Fonte C.4: Classe VetorUtils

```
1   .class VetorUtils
2   .method static swap([I;II) V
3   .limit locals 4
4           aload_0
5           iload_1
6           iaload
7           istore_3
8           aload_0
9           iload_1
10          aload_0
11          iload_2
12          iaload
13          iastore
14          aload_0
15          iload_2
16          iload_3
17          iastore
18          return
19  .end method
20  .method static bubblesort(LVector;I) V
21  .limit locals 4
22          iconst_0
23          istore_2
24  L4:     iload_2
25          iload_1
26          iconst_1
27          isub
28          if_icmplt L1
29          goto L2
30  L1:     iload_1
31          iconst_1
32          isub
33          istore_3
34  L7:     iload_3
35          iload_2
36          if_icmpgt L3
```

```
37          iload_2
38          iinc
39          istore_2
40          goto L4
41  L3:     aload_0
42          getfield Vector/data [I;
43          iload_3
44          iaload
45          aload_0
46          getfield Vector/data [I;
47          iload_3
48          iconst_1
49          isub
50          iaload
51          if_icmplt L5
52  L6: iload_3
53          iconst_1
54          isub
55          istore_3
56          goto L7
57  L5:     aload_0
58          getfield Vector/data [I;
59          iload_3
60          iload_3
61          iconst_1
62          isub
63          invokestatic VetorUtils/swap([I;II) V
64          goto L6
65  L2:     return
66  .end method
67  .method static insertionsort(LVector;I) V
68  .limit locals 5
69          iconst_1
70          istore_2
71  L1:     iload_2
72          iload_1
73          if_icmplt L2
74          goto L3
75  L2:     aload_0
76          getfield Vector/data [I;
77          iload_2
78          iaload
79          istore_3
80          iload_2
81          iconst_1
82          isub
83          istore 4
```

```
84  L4: iload 4
85          ifge L5
86  L6:     aload_0
87          getfield Vector/data [I;
88          iload 4
89          iinc
90          iload_3
91          iastore
92          iload_2
93          iinc
94          istore_2
95          goto L1
96  L5: aload_0
97          getfield Vector/data [I;
98          iload 4
99          iaload
100         iload_3
101         if_icmpgt L7
102         goto L6
103 L7: aload_0
104         getfield Vector/data [I;
105         iload 4
106         iinc
107         aload_0
108         getfield Vector/data [I;
109         iload 4
110         iaload
111         iastore
112         iload 4
113         iconst_1
114         isub
115         istore 4
116         goto L4
117 L3: return
118 .end method
119 .method static merge([I;III) V
120 .limit locals 11
121         iload_2
122         iload_1
123         isub
124         iinc
125         istore 4
126         iload_3
127         iload_2
128         isub
129         istore 5
130         iload_1
```

```
131             iconst_1
132             isub
133             istore_1
134             iload 4
135             newarray int
136             astore 6
137             iload 5
138             newarray int
139             astore 7
140             iconst_0
141             istore 8
142     L1:     iload 8
143             iload 4
144             if_icmplt L2
145             iconst_0
146             istore 8
147     L3:     iload 8
148             iload 5
149             if_icmplt L4
150             iconst_0
151             istore 8
152             iconst_0
153             istore 9
154             iload_1
155             istore 10
156     L5:     iload 10
157             iload_3
158             if_icmplt L6
159             return
160     L6:     iload 8
161             iload 4
162             if_icmpeq L8
163             goto L9
164     L8:     iload 9
165             iload 5
166             if_icmplt L10
167     L9:     iload 8
168             iload 4
169             if_icmplt L12
170             goto L13
171     L12: iload 9
172             iload 5
173             if_icmpeq L14
174     L13: iload 8
175             iload 4
176             if_icmplt L16
177             goto L7
```

```
178 L16: iload 9
179          iload 5
180          if_icmplt L17
181          goto L7
182 L17: aload 6
183          iload 8
184          iaload
185          aload 7
186          iload 9
187          iaload
188          if_icmple L18
189          aload_0
190          iload 10
191          aload 7
192          iload 9
193          iaload
194          iastore
195          iload 9
196          iinc
197          istore 9
198          goto L7
199 L18: aload_0
200          iload 10
201          aload 6
202          iload 8
203          iaload
204          iastore
205          iload 8
206          iinc
207          istore 8
208          goto L7
209 L14: iload 10
210          iload_3
211          if_icmplt L15
212          goto L7
213 L15: aload_0
214          iload 10
215          aload 6
216          iload 8
217          iaload
218          iastore
219          iload 10
220          iinc
221          istore 10
222          iload 8
223          iinc
224          istore 8
```

```
225             goto L14
226     L10: iload 10
227             iload_3
228             if_icmplt L11
229             goto L7
230     L11: aload_0
231             iload 10
232             aload 7
233             iload 9
234             iaload
235             iastore
236             iload 10
237             iinc
238             istore 10
239             iload 9
240             iinc
241             istore 9
242             goto L10
243     L4:     aload 7
244             iload 8
245             aload_0
246             iload_2
247             iload 8
248             iadd
249             iaload
250             iastore
251             iload 8
252             iinc
253             istore 8
254             goto L3
255     L2:     aload 6
256             iload 8
257             aload_0
258             iload_1
259             iload 8
260             iadd
261             iaload
262             iastore
263             iload 8
264             iinc
265             istore 8
266             goto L1
267     L7:     iload 10
268             iinc
269             istore 10
270             goto L5
271             return
```

```
272 .end method
273 .method static mergesort(LVector;II) V
274         iload_1
275         iload_2
276         if_icmplt L1
277         goto L2
278 L1: iload_1
279         iload_2
280         iadd
281         iconst_2
282         idiv
283         istore_3
284         aload_0
285         iload_1
286         iload_3
287         invokestatic VetorUtils/mergesort (LVector;II) V
288         aload_0
289         iload_3
290         iinc
291         iload_2
292         invokestatic VetorUtils/mergesort (LVector;II) V
293         aload_0
294         getfield Vector/data [I;
295         iload_1
296         iload_3
297         iload_2
298         invokestatic VetorUtils/merge([I;III) V
299 L2:     return
300 .end method
301 .end class
```

Código Fonte C.5: Classe ListaUtils

```
1  .class ListaUtils
2  .method static preencherLista(LList;I) V
3  .limit locals 5
4          new NodeList
5          astore_2
6          aload_2
7          ldc 10
8          invokespecial NodeList/<init>(I) V
9          aload_0
10         aload_2
11         putfield List/init LNodeList;
12         aload_0
13         iconst_1
14         putfield List/size I
```

```
15          aload_0
16          getfield List/init LNodeList;
17          astore_3
18          iconst_1
19          istore 4
20  L1: iload 4
21          iload_1
22          if_icmplt L2
23          return
24  L2: new NodeList
25          astore_2
26          aload_2
27          iload 4
28          iinc
29          ldc 10
30          imul
31          invokespecial NodeList/<init>(I) V
32          aload_3
33          aload_2
34          putfield NodeList/next LNodeList;
35          aload_0
36          aload_0
37          getfield List/size I
38          iinc
39          putfield List/size I
40          aload_3
41          getfield NodeList/next LNodeList;
42          astore_3
43          iload 4
44          iinc
45          istore 4
46          goto L1
47  .end method
48  .method static inverterLista(LList;) V
49  .limit locals 3
50          aload_0
51          getfield List/init LNodeList;
52          astore_1
53  L1: aload_1
54          getfield NodeList/next LNodeList;
55          aconst_null
56          if_acmpne L2
57          aload_0
58          getfield List/init LNodeList;
59          astore_2
60  L3: aload_2
61      aload_1
```

```
62      if_acmpne L4
63      return
64  L2:     aload_1
65          getfield NodeList/next LNodeList;
66          astore_1
67          goto L1
68  L4: aload_0
69      aload_2
70      getfield NodeList/next LNodeList;
71      putfield List/init LNodeList;
72      aload_2
73      aload_1
74      getfield NodeList/next LNodeList;
75      putfield NodeList/next LNodeList;
76      aload_1
77      aload_2
78      putfield NodeList/next LNodeList;
79      aload_0
80      getfield List/init LNodeList;
81      astore_2
82          goto L3
83  .end method
84  .method static removerNo(LList;I) V
85  .limit locals 5
86          aload_0
87          getfield List/init LNodeList;
88          ifnull L1
89          iload_1
90          aload_0
91          getfield List/size I
92          if_icmpgt L1
93          goto L2
94  L1: return
95  L2: aload_0
96          getfield List/init LNodeList;
97          astore_2
98          iload_1
99          iconst_1
100         if_icmpeq L3
101         iconst_0
102         istore_3
103 L4: iload_3
104         iload_1
105         iconst_2
106         isub
107         if_icmplt L5
108         aload_2
```

```
109         getfield NodeList/next LNodeList;
110         astore 4
111         aload_2
112         aload_2
113         getfield NodeList/next LNodeList;
114         getfield NodeList/next LNodeList;
115         putfield NodeList/next LNodeList;
116         aload 4
117         aconst_null
118         putfield NodeList/next LNodeList;
119         return
120 L5: aload_2
121         getfield NodeList/next LNodeList;
122         astore_2
123         iload_3
124         iinc
125         istore_3
126         goto L4
127 L3: aload_0
128         aload_2
129         getfield NodeList/next LNodeList;
130         putfield List/init LNodeList;
131         aload_2
132         aconst_null
133         putfield NodeList/next LNodeList;
134         return
135 .end method
136 .end class
```

Código Fonte C.6: Classe TreeUtils

```
1  .class TreeUtils
2  .method static inserir(LBinaryTree;I) V
3  .limit locals 4
4           aload_0
5           getfield BinaryTree/root LNodeTree;
6           ifnull L1
7           aload_0
8           getfield BinaryTree/root LNodeTree;
9           astore_2
10 L3: aload_2
11          astore_3
12          iload_1
13          aload_3
14          getfield NodeTree/info I
15          if_icmpgt L4
16          iload_1
```

```
17          aload_3
18          getfield NodeTree/info I
19          if_icmplt L7
20          return
21  L7: aload_3
22          getfield NodeTree/leftchild LNodeTree;
23          astore_2
24          aload_2
25          ifnull L8
26          goto L6
27  L8: aload_3
28          new NodeTree
29          putfield NodeTree/leftchild LNodeTree;
30          aload_3
31          getfield NodeTree/leftchild LNodeTree;
32          iload_1
33          invokespecial NodeTree/<init>(I) V
34          aload_0
35          getfield BinaryTree/root LNodeTree;
36          invokestatic TreeUtils/atualizarAlturaNos(LNodeTree;) V
37          goto L6
38  L4: aload_3
39          getfield NodeTree/rightchild LNodeTree;
40          astore_2
41          aload_2
42          ifnull L5
43          goto L6
44  L5: aload_3
45          new NodeTree
46          putfield NodeTree/rightchild LNodeTree;
47          aload_3
48          getfield NodeTree/rightchild LNodeTree;
49          iload_1
50          invokespecial NodeTree/<init>(I) V
51          aload_0
52          getfield BinaryTree/root LNodeTree;
53          invokestatic TreeUtils/atualizarAlturaNos(LNodeTree;) V
54          goto L6
55  L6:     aload_2
56          ifnonnull L3
57          goto L2
58  L1:     aload_0
59          new NodeTree
60          putfield BinaryTree/root LNodeTree;
61          aload_0
62          getfield BinaryTree/root LNodeTree;
63          iload_1
```

```
64          invokespecial NodeTree/<init>(I) V
65  L2: aload_0
66          aload_0
67          getfield BinaryTree/size I
68          iinc
69          putfield BinaryTree/size I
70          return
71  .end method
72  .method static buscar(LNodeTree;I) LNodeTree;
73  .limit locals 2
74          aload_0
75          ifnull L1
76          iload_1
77          aload_0
78          getfield NodeTree/info I
79          if_icmpeq L2
80          iload_1
81          aload_0
82          getfield NodeTree/info I
83          if_icmpgt L3
84          aload_0
85          getfield NodeTree/leftchild LNodeTree;
86          iload_1
87          invokestatic TreeUtils/buscar(LNodeTree;I) LNodeTree;
88          areturn
89  L3: aload_0
90          getfield NodeTree/rightchild LNodeTree;
91          iload_1
92          invokestatic TreeUtils/buscar(LNodeTree;I) LNodeTree;
93          areturn
94  L2: aload_0
95          areturn
96  L1:     aconst_null
97          areturn
98  .end method
99  .method static altura(LNodeTree;) I
100 .limit locals 3
101         aload_0
102         ifnull L1
103         aload_0
104         getfield NodeTree/leftchild LNodeTree;
105         invokestatic TreeUtils/altura(LNodeTree;) I
106         istore_1
107         aload_0
108         getfield NodeTree/rightchild LNodeTree;
109         invokestatic TreeUtils/altura(LNodeTree;) I
110         istore_2
```

```
111         iload_1
112         iload_2
113         if_icmplt L2
114         iload_1
115         iinc
116         ireturn
117 L2:     iload_2
118         iinc
119         ireturn
120 L1:     iconst_m1
121         ireturn
122 .end method
123 .method static atualizarAlturaNos(LNodeTree;) V
124 .limit locals 1
125         aload_0
126         ifnull L1
127         aload_0
128         aload_0
129         invokestatic TreeUtils/altura(LNodeTree;) I
130         putfield NodeTree/height I
131         aload_0
132         getfield NodeTree/rightchild LNodeTree;
133         invokestatic TreeUtils/atualizarAlturaNos(LNodeTree;) V
134         aload_0
135         getfield NodeTree/leftchild LNodeTree;
136         invokestatic TreeUtils/atualizarAlturaNos(LNodeTree;) V
137 L1:     return
138 .end method
139 .end class
```