令和2年度 修士論文

ライブデータ構造プログラミングの大規模データ利用のための改善

東京工業大学大学院 情報理工学院
数理・計算科学系 数理・計算科学コース

学籍番号 19M30070

小串 智希

指導教員

増原 英彦 教授

令和3年1月22日

概要

プログラミング環境を改善する研究の一つにライブプログラミング環境がある。ライブプログラミング環境は、プログラムを編集すると実行結果を即座にフィードバックする。これによってプログラマは実行結果の確認とプログラムの編集を同時に行うことができる。

データ構造プログラミングを支援するライブプログラミング環境に岡らの Kanon [1] がある。Kanon はコードの編集をしながら生成されるオブジェクトの参照関係をノードリンク図として視覚化する。この論文では、Kanon でのグラフレイアウトに関する２つの研究について報告する。１つは Kanon に使用されている Ogushi レイアウトアルゴリズムについてのユーザー実験であり、もう１つは Kanon で大規模データを利用するためのビジュアル UI の改善に関しての提案である。

Ogushi レイアウトとは、Kanon で視認しやすい形に描画することのできるデータ構造の種類を増やすために提案されたレイアウト手法である。しかし Ogushi レイアウトによって本当に見やすく描画できるデータ構造の種類が増えたのかが明らかではなかったため、著者はユーザー実験を実施した。このユーザー実験は Ogushi レイアウトが導入される以前の Kanon でのレイアウト図と Ogushi レイアウトでの図を被験者がそれぞれ見ながらいくつかのタスクを行うものである。実験とインタビューの結果から Ogushi レイアウトアルゴリズムは Kanon の元のレイアウトよりも見やすく描画できるデータ構造の種類が増えており被験者のほとんども肯定的な印象を持っていることが分かったが、一部のデータ構造には適しているとは言えずまだ改善の余地があることが明らかとなった。

また、規模の大きなデータを利用したプログラミングで Kanon を役立てるためには２つの問題点がある。重要度の低いノードやエッジが増えるためプログラマがオブジェクトの参照関係が追いづらくなる点と、データグラフが巨大化してしまうためプログラマが注目したい箇所の発見・認識が困難になる点である。これらの問題点を解決するために、プログラマの注目点をエディタから自動推定し、注目点からの参照距離やクラス・フィールドの重要度を使ってグラフ内のそれぞれの箇所の拡大率を変更する手法を提案する。この提案手法によりプログラマはコードを編集するだけで、注目したい箇所付近が拡大表示・注目していない重要度の低い箇所

が縮小表示され、データ構造の変化を観察しやすくなる。

我々はKanonのユーザーインターフェースを拡張し、Ogushiレイアウトアルゴリズムを改良したFiFAレイアウトアルゴリズムを実装・導入することで提案手法を実現した。この論文では提案手法によってレイアウトされた大規模データの図をいくつか紹介している。

謝辞

まず、本研究を進めるにあたり多くのアドバイスやご指導をいただいた増原英彦教授、青谷知幸助教、叢悠悠助教に心より感謝いたします。

また、実験に協力していただき、日頃から多くの助言をしていただいている増原英彦研究室の学生のみなさまにも感謝を申し上げます。彼らの協力がなければこの論文は完成しなかったでしょう。

最後に、これまで経済的支援と多くの励ましを頂いた家族の皆様に感謝の意を申し上げたいと思います。

目 次

図 目 次

表 目 次

第1章　導入

1.1　プログラミング環境

プログラマがプログラミングを編集する際の負担を減らすために、これまで多くのプログラミング環境に関する研究が行われてきた。最も初歩的なプログラミング環境は、エディタ・コンパイラ・デバッガなどをそれぞれ用意したものである。

しかしこの環境では複数のソフトウェアを別々に用意しなければならずファイル構成が複雑になるほど管理も複雑化する。そのため、ソフトウェア開発で使用されるソフトウェアを一つの開発環境にまとめるために作られたのが統合開発環境である。

また、コンパイラを使用した開発環境ではプログラマはソースコードを編集するたびにコンパイルしファイルを実行させないと実行結果を確認できなかった。この問題を解決するために、REPL(Read-Eval-Print-Loop)環境やライブプログラミング環境が開発された。

REPL 環境ではコンパイラの代わりにインタプリタを使用し、命令列の読み込み、評価・実行、結果の出力を繰り返していく。インタプリタを用いるとソースコードをそのまま実行するかあるいは中間表現に変換して実行することができるのでコンパイルの時間をかけることなくテストを素早く実行することができる。

REPL 環境と同様、プログラムの編集から出力された実行結果の確認までの時間を短縮するために開発されたのがライブプログラミング環境である。

1.2　ライブプログラミング環境

ライブプログラミング環境とはソースコードの編集と実行結果のフィードバックを同時に行うことができる環境である。従来のプログラミング環境ではソースコードの編集、評価・実行、出力結果の確認といったフェーズを別々に行う必要があったが、ライブプログラミングではこれらのフェーズを同時に行うことでプログラミングにかかる時間やストレスを減らすことができる。

ライブプログラミング環境はソースコードの編集と実行結果の確認を同時に行えるという性質から、絵を描くためのプログラム [2]、音楽を合成するためのプログラム [3]、ゲームのキャラクターをアニメーション化するためのプログラム [4]、アルゴリズムを教えるためのプログラム [5] などテキストから実行結果が予想しにくい分野で主に使用されてきた。

データ構造プログラミングも同様のカテゴリに分類され、ライブプログラミング環境が役に立つと考えられている [1]。オブジェクト指向言語では、データ構造プログラムは通常クラス・メソッドの定義とテストケースの作成から構成される。このとき、プログラマがコードを編集しながら生成された各オブジェクトの参照関係やその変化をテキストから予測するのは簡単なものではない。ZStep [6]、jGRASP [7]、Python Tutor [8] などユーザーが定義したデータ構造を可視化するプログラミング環境は多くあるが、これらの環境は主にプログラムを完成させてから事後分析するものである。つまり、これらの環境でコード編集と実行結果の確認は異なるプロセスとなっている。

データ構造プログラムを編集しながらオブジェクト群の参照関係を図示してプログラマに示すような環境には岡らの Kanon [1] がある。

1.3　Kanon：ライブデータ構造プログラミング環境

Kanon はデータ構造プログラミングのためのライブプログラミング環境である。エディタで編集中のコードに対してエディタ内のカーソル位置に対応する実行ポイントまで生成された全てのオブジェクトとその参照関係をノードリンク図として画面に描画する。

プログラマがコードを編集しながらデータの参照関係を認識・理解し役立てるためには表示される図の読み取りやすさが必要不可欠である。しかし Kanon に使用されている元のレイアウト手法はリストと二分木のみに対応したものであり、その他のデータ構造ではプログラマが理解しやすいような形に描画されるとは限らなかった。そのため著者は以前、Kanon で扱えるデータ構造の種類に汎用性を持たせ、表示されるデータの参照関係の視認性を向上させるために Ogushi レイアウトアルゴリズムを提案した。

1.4　Ogushi レイアウトアルゴリズム

Ogushi レイアウトアルゴリズムはオブジェクト群の参照関係のみを入力として受け取り、各オブジェクトノードの適切な座標を計算するグラフアルゴリズムである。そのため多くの種類のデータ構造に対してプログラ

マの余計な入力の手間なしで理解しやすい形で描画することが可能である。このレイアウトアルゴリズムでは、同一のフィールドを表すエッジの向きを同じ方向に揃えることでオブジェクトの参照関係の視認性を向上させようとしている。また著者は Ogushi レイアウトアルゴリズムを実装し Kanon に組み込むことで Kanon のレイアウトシステムを改良した。

1.5　研究の目的・貢献

上記の Ogushi レイアウトアルゴリズムの実装後、我々は

- Ogushi レイアウトがプログラマにとって視認しやすいものであるのか？
- Ogushi レイアウトがプログラムするときに役立つのか？

の２つの疑問を検証する必要があった。そこで、この論文ではまず Ogushi レイアウトがプログラマにとって視認しやすいものであるかどうかを確かめるために行ったユーザー実験について報告している。また、Ogushi レイアウトでは規模の大きなデータを利用したプログラミングに役立てるためにはいくつかの課題があることが考察された。そのため、次に本論では大規模データをライブデータ構造プログラミングで利用する際のより有用性の高いビジュアライズのための手法について提案をしている。

本論の貢献は以下の通りである。

- ユーザー実験より、Oushi レイアウトは Kanon の元のレイアウトよりも視認性の高く描画できるデータ構造の種類が増えていることが明らかになった。
- 同一のフィールドを同一の向きに揃えるレイアウトは、根から順に枝分かれしていく木構造の描画には適しているが、閉路を持った構造やエッジが１つのノードに収束するような構造の描画には必ずしも適しているとは言えないことが明らかになった。
- エディタからプログラマの注目箇所を自動推定し、FiFA レイアウトと呼ばれる注目点から参照距離やクラス・フィールドの重要度を使ってグラフ内のそれぞれの箇所の拡大率を変更・決定する手法を提案する。プログラマはコードを編集しながら注目したい箇所が拡大表示・注目されていない不要な箇所が縮小表示され、データ構造の変化を観察しやすくなる。

- 著者は実際にいくつかのテストケースを作成しそのメソッドを記述する途中で大規模データでも注目しているノードが視認しやすくなっていることを確認した。

この論文は次のような構成となっている。第 2 章では研究背景について説明する。第 3 章では Ogushi レイアウトに関するユーザー実験について、第 4 章ではライブデータ構造プログラミングでの大規模データの利用についての問題点と提案手法・実装などについて述べる。第 5 章では全体のまとめを述べる。

第2章　研究背景

2.1　グラフレイアウト手法

グラフを綺麗に描画するためのアルゴリズムはその用途に応じて様々なものが研究・開発されている。ここでは本論に関わるレイアウトアルゴリズムを解説する。

2.1.1　力学グラフレイアウトアルゴリズム

力学グラフレイアウト手法では、ノード間に仮想的な力を働かせ、力が釣り合う安定座標を求めてレイアウトする。力学グラフレイアウトアルゴリズムの代表的なものにKamada-Kawaiアルゴリズム [10] やFruchterman-Reingoldアルゴリズム [9] が挙げられる。

Kamada-Kawaiアルゴリズムではノード間にばねを与え、ばねのエネルギー関数が最小になるようなノードの座標を求めることでグラフをレイアウトする。具体的には、各ノード間の理想的なばねの長さを $l_{i,j} = Ld_{i,j}$（L は定数、$d_{i,j}$ は2点間の最短経路長）で与え、またばねの強さを $k_{i,j} = \frac{K}{d_{i,j}}$（$K$ は定数）で定義する。するとこのばねの持つエネルギー E は

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{i,j} (\|p_i - p_j\| - l_{i,j})^2 \tag{2.1}$$

の式で求めることができる。この式を各点の座標 $p_i = (x_i, y_i)^T$ について最小化していく。

また、Fruchterman-Reingoldアルゴリズムでは隣接ノード間に引力、全ノード間に斥力を働かせ、力の合計ベクトルに沿って各ノードを微小距離だけ移動させるというプロセスを一定回数繰り返すことによって力の釣り合う安定座標を求める。引力は $f_a(d) = \frac{d^2}{k}$、斥力は $f_r(d) = -\frac{k^2}{d}$ という式で定義される。ただし、c を定数、$area$ を表示領域の面積として $k = c\sqrt{\frac{area}{|V|}}$ の式で k は定義される。この引力と斥力は隣接する2つのノード間の距離が k のときにちょうど合力が0になる。また、ノードの移動距離の最大値として温度パラメータ t が定められており、t はノードの移動を繰り返すごとに徐々に減少していく。

2.1.2　魚眼レンズグラフレイアウトアルゴリズム

魚眼レンズグラフレイアウトアルゴリズム [15] は通常にレイアウトされたグラフを注目点付近を拡大表示・注目点から離れるほど縮小表示するように変換するためのアルゴリズムである。このように表示させることでグラフ内の関心の高い領域の詳細と全体の構造を両方同時に確認できるというメリットがある。

通常のレイアウトで座標が P_{norm} で表される点の魚眼レンズレイアウトでの座標 P_{feye} は、

$$P_{feye} = g(P_{norm})D_{max} + P_{focus} \tag{2.2}$$

の式で計算される。ここで、

$$g(P_{norm}) = \frac{(d+1)\frac{D_{norm}}{D_{max}}}{d\frac{D_{norm}}{D_{max}} + 1} = \frac{d+1}{d + \frac{D_{max}}{D_{norm}}} \tag{2.3}$$

であり、P_{focus} は注目点の座標、D_{max} は注目点から画面境界までの距離、D_{norm} はその点から注目点までの距離である。ここで、x 座標と y 座標の 2 つの次元は独立して扱われることに注意する必要がある。また、d はゆがみ変数と呼ばれている。d の値が大きくなればなるほど $g(x)$ は $x = 0$ 付近で勾配が大きくなる。

2.2　Kanon

Kanon [1] はデータ構造プログラミングのためのライブプログラミング環境である。図 2.1 は Kanon のスクリーンショットである。画面左側がプログラムの記述されるエディタ画面、右上がデータ構造を表示する視覚化画面、右下が制御フローを表示するコールツリー画面である。Kanon は以下の前提の元で設計されている。

- プログラムは JavaScript で 1 つのファイルに記述される。このファイルはデータ構造およびそのメソッドの定義とテストケースとして機能するトップレベルの式で構成されている。
- データ構造をノードリンク図として描画する。視覚化画面の各楕円形はオブジェクトのクラス名でラベル付けされたオブジェクトを表す。楕円形からの矢印はオブジェクトから参照することのできる他のオブジェクトまたはプリミティブ値のフィールドを表し、始点のない緑の矢印は変数が参照するオブジェクトを示す。

- Kanon はプログラムを継続的に実行し、コードの始めからエディタ内のカーソル位置に対応する実行ポイントまで生成された全てのオブジェクトを可視化する。

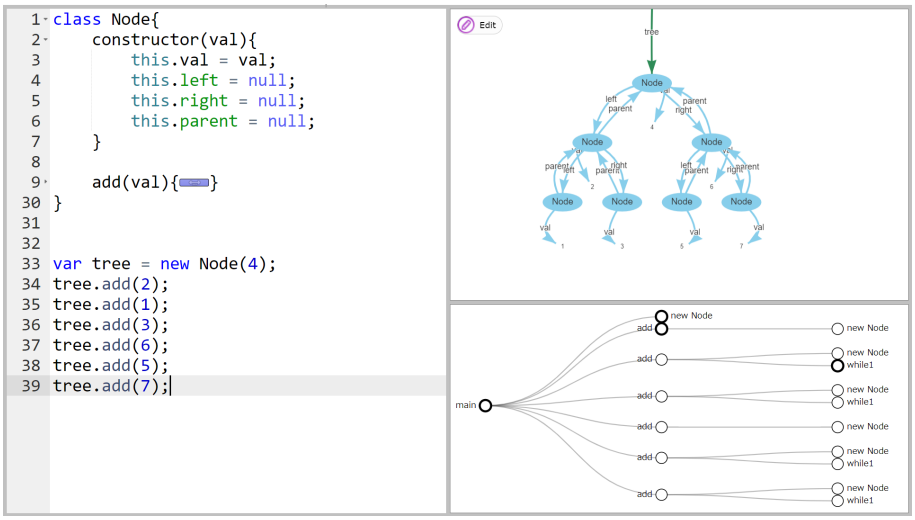


図 2.1: ライブデータ構造プログラミング環境 Kanon の画面

2.2.1　２つのレイアウト手法

現在 Kanon は、Kanon が従来から持つレイアウト手法（以下、Kanon レイアウトと呼ぶ）と Ogushi レイアウト手法の２通りのレイアウト手法でデータ構造をレイアウトすることが可能である。

2.2.1.1　Kanon レイアウト

Kanon レイアウトでは、リストと二分木の２種類のデータ構造のみを特別なレイアウト手法で描画し、その他のデータ構造のレイアウトに関しては vis.js 視覚化ライブラリ [12] が力学的手法を用いてレイアウトする。また、データ構造の識別にはクラス名とフィールド名の情報を利用している。

2.2.1.1.1　問題点

Kanon レイアウトはリストと二分木のみに特別なレイアウト手法が働くため、リスト・二分木以外のデータ構造を綺麗にレイアウトできるとは

限らない。図 2.2 と図 2.3 は Kanon レイアウトによるリスト・二分木の描画である。これに対して、三分探索木を描画しようとすると図 2.4 のようになる。

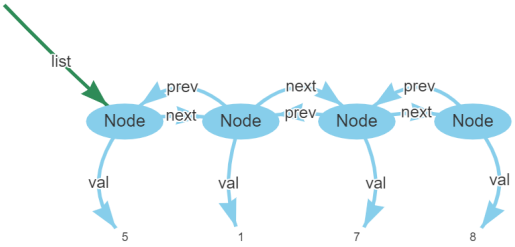


図 2.2: Kanon レイアウト：リスト

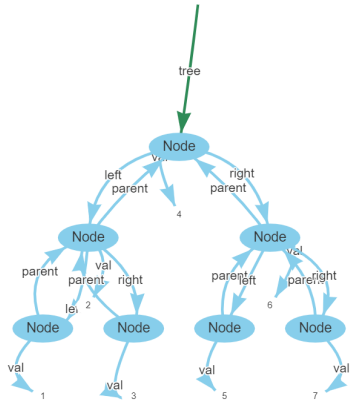


図 2.3: Kanon レイアウト：二分木

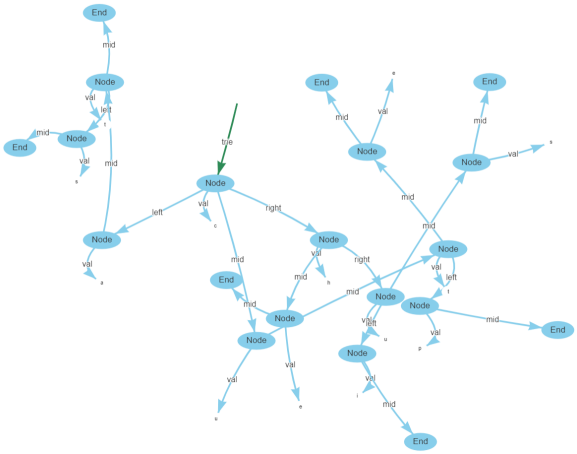


図 2.4: Kanon レイアウト：三分探索木

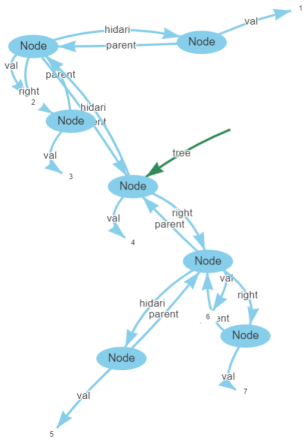


図 2.5: Kanon レイアウト：
フィールド名を変えた二分木

また、Kanon レイアウトではクラス名やフィールド名が特定の文字列と一致するかどうかでデータ構造の種類を判断している。例えば、Kanon レイアウトが二分木であると判断するのはクラス名が「Node」、フィールド名に「left」「right」が含まれるときに限る。そのため、二分木の「left」

のフィールド名を「hidari」と書き換えると Kanon レイアウトはこのデータを二分木と見なさなくなり、図 2.5 のようにレイアウトする。これは図 A.5 のイメージ図や図 2.3 のレイアウトとは大きくかけはなれている。

さらに、リストやツリーは単体では綺麗に描画されるが、例えば図 2.6 のようなリストのそれぞれのノードがツリーノードへの参照リンクを持っているような複数のデータ構造を組み合わせたような構造に対しては単体のような綺麗な描画がされない。

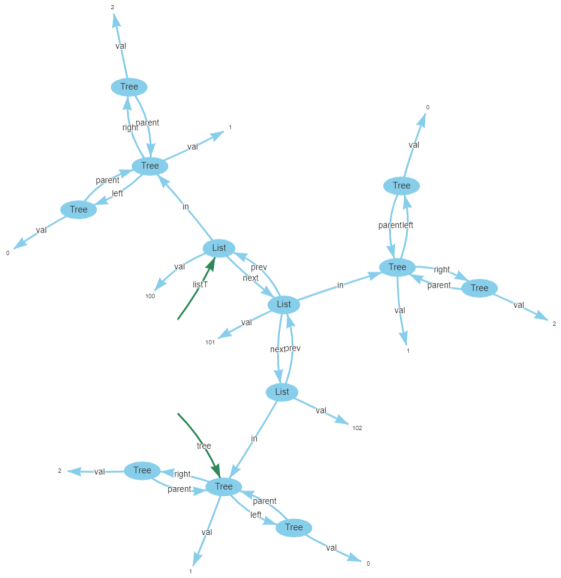


図 2.6: Kanon レイアウト：入れ子構造

2.2.1.2　Ogushi レイアウト

上記の Kanon レイアウトでの問題点を解決するために作られたのが Ogushi レイアウトである。Ogushi レイアウトアルゴリズムはエッジの角度に対しても力が働くような力学グラフレイアウトアルゴリズムである。従来の力学レイアウトアルゴリズムは各ノード間に引力や斥力を働かせるモデルを作ることで力が釣り合う安定座標を求めてレイアウトするものであるが、Ogushi レイアウトではエッジの傾きを定められた角度に矯正するような回転力を従来の力学レイアウトアルゴリズムに新たに追加している。データ構造を描画するとき、この力によって同一のフィールドを表すエッジを同一の向きに揃えることでオブジェクトの参照関係が視認しやすくなることが期待される。また、Ogushi レイアウトアルゴリズムはオブ

ジェクト群とその参照関係のみを入力として使用するので、プログラマの余計な作業なしでレイアウトをすることができ、上記の Kanon レイアウトの問題点を解決しデータ構造の種類に関する汎用性を向上させることが期待される。

2.2.1.2.1　レイアウト例

Ogushi レイアウトの例を図 2.7〜2.11 に示す。図 2.9 からもリストや二分木以外のデータ構造でも比較的プログラマの頭の中のイメージ図に近い形で描画されていることが確認できる。また、図 2.10 は二分木の「left」フィールドの名前を「hidari」に変えたものであるが、この場合も図 2.8 と同様な描画がなされる。図 2.11 はリストのそれぞれのノードがツリーノードへの参照リンクを持っているようなデータ構造であるが、この場合もリストや二分木のレイアウトに見られる特徴をそのまま失うことなく描画されていることが確認できる。

2.2.1.2.2　レイアウトアルゴリズム

Ogushi レイアウトではまずオブジェクト群の参照関係から各エッジの理想角度を計算する。次に、それぞれのエッジの理想角度を元に各ノードに働く引力・斥力・回転力の 3 つの力を計算し、安定座標を求めていく。

まずオブジェクト群をクラスが同じオブジェクト毎に区分する。次に、各クラスをオブジェクトの参照関係から、(1) リスト構造、(2)n 分木構造（$n \geq 2$）の 2 種類に分類する。これはオブジェクトが自身と同一のクラスオブジェクトをいくつ参照しているかで判断する。リスト構造のエッジの角度は水平横向き（0°）とし、n 分木構造でオブジェクト自身と同一のクラスを指すエッジの i 本目の角度は $90° \times \frac{2i-1}{n}$ と定める。例えば図 2.8 では Node クラスオブジェクトの left フィールドには 135°、right フィールドには 45° といった理想角度が割り当てられている。

オブジェクト自身と異なるクラスオブジェクトを参照するフィールドエッジはその本数 m によって、i 本目のエッジの角度を $90° \times \frac{2i-1}{m}$ とする。

また、閉路上にあるノードには理想角度を設定しない。このようにすることで、循環リストを図 2.12 のような環状に描くことができるようになる。

このようにして求めたそれぞれのエッジの理想角度を元に各ノードに働く力を計算しノードの座標を更新していく。このアルゴリズムは上記の Fruchterman-Reingold アルゴリズム [9] の計算方法を元に作られている。

座標の更新フェーズでは、各ノードに働く 3 つの力

1. 隣接ノードから働く引力 $f_a(d) = \frac{d^2}{k}$

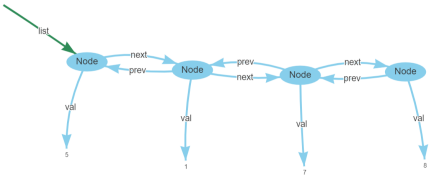


図 2.7: Ogushi レイアウト：リスト

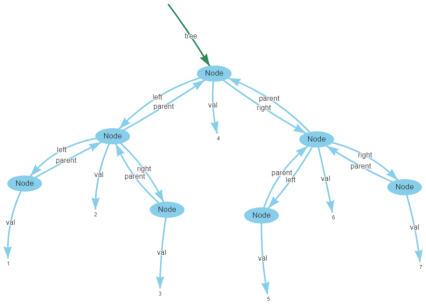


図 2.8: Ogushi レイアウト：二分木

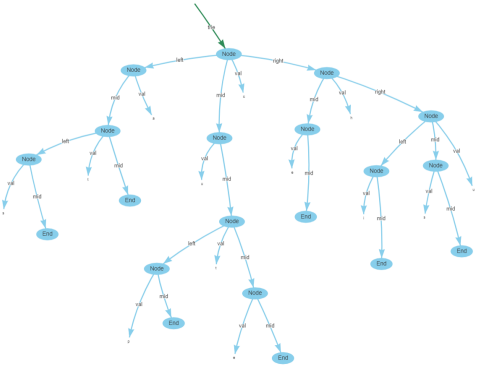


図 2.9: Ogushi レイアウト：三分探索木

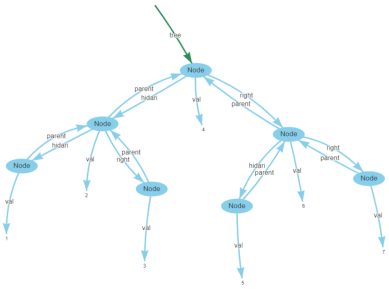


図 2.10: Ogushi レイアウト：フィールド名を変えた二分木

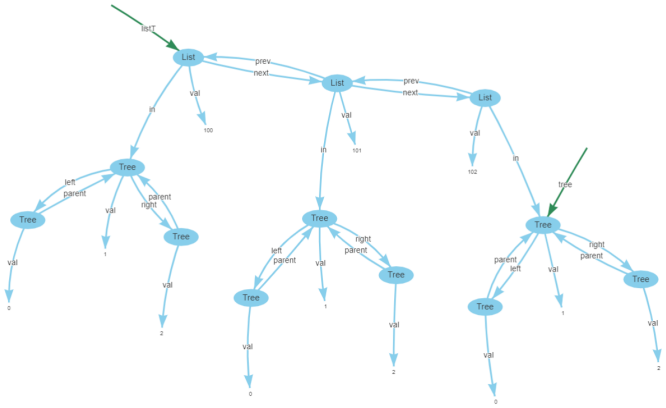


図 2.11: Ogushi レイアウト：入れ子構造

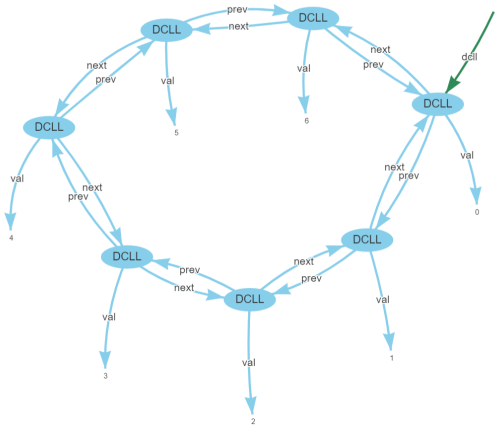


図 2.12: Ogushi レイアウト :双方向循環リスト

2. 他の全ノードから働く斥力 $f_r(d) = -\frac{k^5}{d^4}$

3. エッジを理想角度に近づけるように働く回転力 f_m

を計算し、3 つの力の合成ベクトルに沿って各ノードを微小距離だけ移動させる。d は 2 点間の距離、k は定数である。図 2.13 は理想角度 0° と定められたエッジの端点のノードに働く力のイメージ図である。これを十分な量の回数繰り返すことで全ノードに働く力の合計が十分小さくなりノードが安定座標に到達する。このとき、ノード座標が振動することを防ぐために各ノードが移動することのできる最大距離（温度パラメータ）を徐々に減少させていく。

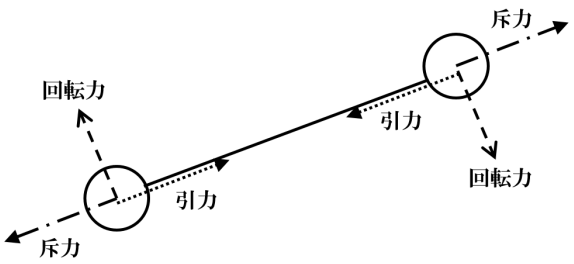


図 2.13: 各ノードに働く 3 つの力

第3章　評価実験

初期評価として、既存の Ogushi レイアウトアルゴリズムによるレイアウトに関するユーザー実験を実施した。この実験により、Ogushi レイアウトのメリットや課題点が見つかった。

3.1　実験目的

この実験の主な目的は、Ogushi レイアウトアルゴリズムによってライブプログラミング環境の画面に表示されるデータ構造の参照関係がプログラマにとってどれだけ理解しやすいものになるのかを明らかにすることである。比較対象として、Kanon レイアウトを利用する。また、Ogushi レイアウトアルゴリズムによるレイアウトに関するプログラマの意見を収集するために、ユーザー実験の後に簡単なインタビューも行った。

3.2　実験計画

以下のリサーチクエスチョンを念頭において実験を設計した。この実験では、参加者に Kanon レイアウトまたは Ogushi レイアウトのいずれかでデータ構造のレイアウトされた図を使用していくつかのタスクを解決させ、そのデータ構造への理解度がどれだけのものになるのかを観察し意見を収集した。

- **プログラマが頭の中に思い描くデータ構造の図は同一のフィールドエッジの向きが同一の向きになっているのか？**

  プログラマが頭の中に思い描くデータ構造の図を調べることによって「同一のフィールドエッジを同一の方向に揃える」ことが一般的にプログラマの図への理解を助長するものになるのかどうかを調べる。

- **Ogushi レイアウトは Kanon レイアウトよりも被験者にとってデータの参照関係を追いやすいものになっているのか？**

  被験者にとってレイアウトされた図がデータの参照関係を追いやすいものでなければライブプログラミング環境でデータ構造を図示す

ることの利点が薄れてしまう。被験者がデータの参照関係を追うことで解決できるタスクを難易度の異なるものを複数用意し、そのタスクの解決時間を測定することで図の参照関係の見やすさを測定する。

- **Ogushi レイアウトはリストや二分木以外のデータ構造にも対応できているのか？**

  Kanon でのレイアウトの課題として「リスト・二分木以外の構造を分かりやすくレイアウトできるとは限らない」というものがあった。実験では、リスト・二分木以外の構造として三分探索木・リストと二分木の入れ子構造の 2 つについても取り扱った。

3.3　実験手順

我々の実験には 10 人の増原英彦研究室の学生（上級学部および大学院レベル）の参加者に協力してもらった。彼らは皆事前に Kanon でレイアウトされる図がおよそどのようなものであるか（ノード・エッジの表現方法など）を知っていることを留意する必要がある。

参加者ごとに 4 つの実験を行った。参加者はそれぞれの実験を定められた順番に従って受けていき、合計でおよそ 1 時間半から 2 時間ほどかかった。また、各実験ごとに終了後に簡単なインタビューを行った。実験に使用する図はコンピュータの画面に映し出されたものではなく紙に印刷したものを使用した。これは、参加者個々のコンピュータの操作練度によって実験データにバラつきが出るのを防ぐためである。

以下に各実験の詳細を記述する。

3.3.1　様々な種類のデータ構造のメンタルマップに関するテスト

3.3.1.1　目的

この実験の目的は、様々な種類のデータ構造について被験者が「最も理解しやすい」と思っているようなレイアウトの形を観察し、「同一のフィールドエッジを同一の方向に揃える」ことがどれだけ被験者のメンタルマップと合致しているかを確認することである。また、被験者が視認しているレイアウトと、視認してから「最も理解しやすい」と思う構造を想起するのにかかる時間との関係性を確認する。我々は、この時間が長ければ長いほど被験者が視認しているレイアウトは理解に時間がかかるものであると考える。

3.3.1.2　手順

４種類のデータ構造に関して、紙に印刷されたレイアウトを被験者に見てもらいそこから被験者本人が「最も理解しやすい」と思う形を別紙に書いてもらう。また、レイアウトされた図を見てから別紙に図を書き終えるまでの時間を計測する。４種類のデータ構造は以下のものである。

- B+木
- インタープリタ内の変数環境を表したレイアウト
- グラフ（ノードとエッジを実装したもの）
- スキップリスト

それぞれのデータ構造につき Ogushi レイアウトを使用したものと既存の Kanon によってレイアウトされたものの２つの紙を用意し、被験者には各データ構造につきどちらか片方の紙を見てもらった。このとき、被験者には自身に渡された紙に印刷された図がどちらのレイアウトなのかを伝えないこととする。また２つのレイアウトの偏りを防ぐために、被験者には実験ごとに２つのレイアウトを交互に使用した。例えば A と B という２人の被験者がいた場合、A には Ogushi レイアウトで描かれた B+木→ Kanon レイアウトで描かれた変数環境→ Ogushi レイアウトで描かれたグラフ→ Kanon レイアウトで描かれたスキップリスト、という順番で実験をしていき、B は Kanon レイアウトで描かれた B+木→ Ogushi レイアウトで描かれた変数環境→ Kanon レイアウトで描かれたグラフ→ Ogushi レイアウトで描かれたスキップリスト、という順番で実験を行った。図 3.1、3.2 は被験者に渡した B+木のレイアウトの１例である。

「最も理解しやすい」と思う形について、このままでは指示が抽象的なものになってしまうので、被験者には自身がクラスの指導者だとして生徒にそのデータ構造を教えるときを想定してもらった。

また、この実験のみ終了後のインタビューで「なぜそのような形に描画したのか」ということを聞いた。これはプログラマが頭の中に思い浮かべるメンタルマップの形の根拠を知るためである。

3.3.2　二分探索木に関するテスト

3.3.2.1　目的

この実験の目的は、Ogushi レイアウトによりレイアウトされた図によって使用者が二分木の参照関係をどれだけ追いやすくなっているかを確認す

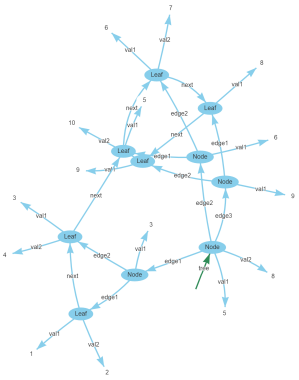


図 3.1: B+木の Kanon レイアウト

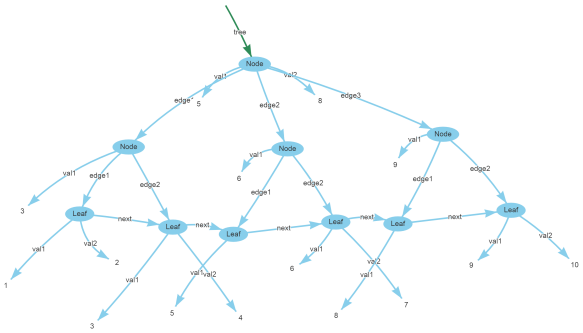


図 3.2: B+木の Ogushi レイアウト

ることである。具体的には、二分木にとある順序で格納されたノードの中から順序のルールに即していないノードを発見するまでの時間を計ることで参照関係の追いやすさを定量的に計ろうとした。我々は、同一のフィールドエッジの傾きが揃っているほど参照関係が追いやすくなりこの実験でのノード発見までの時間が短くなると考える。

3.3.2.2　手順

被験者には以下のことを事前に紙と口頭の両方で説明する。

1. とあるプログラマは以下の条件を満たした二分木を実装しようと考えた。
    - 各ノードには日本人の苗字と名前が格納されている。
    - 全てのノードについて、左の子ノード以下にある全てのノードに格納されている苗字はそのノードに格納されている苗字よりも辞書順で前になるものであり、右の子ノード以下にある全てのノードに格納されている苗字はそのノードに格納されている苗字よりも辞書順で後になる。
    - 同じ苗字を格納されているノードが複数あった場合は名前の辞書順に並べられる。
2. そのプログラマは上記の条件を満たすように二分木を JavaScript で実装し、実際にあるクラスの名簿（男女合わせて 11 人分）のデータをツリーに格納させ、Kanon によってレイアウトさせた。

3. しかし、この実装には誤りがあったため、順序が間違えて格納されているノードが存在している。誤った位置に存在しているノードが１つあるのでレイアウトされた図から探してほしい。

その後、被験者には実際にデータがレイアウトされた図の印刷された紙を渡し、紙を見始めてから誤った位置を指示するまでにかかる時間を計測した。被験者が指示した箇所が合っていればそこで測定を終了し、間違えていた場合は時間の計測を一旦停止し、間違っていた旨を被験者に伝えて再び計測を再開し、正しい箇所を指示するまで続けた。また、10 分以内に正しい箇所を指示できなかった場合はその時点で計測を終了した。これは被験者の心的負担を減らすためである。

我々は３つの異なるデータ（クラス名簿）を Kanon レイアウトと Ogushi レイアウトでそれぞれ表示し印刷した紙を合計６枚用意し、被験者には上記のタスクを３回行ってもらった。このとき、被験者には自身に渡された紙に印刷された図がどちらのレイアウトなのかを伝えないこととする。また２つのレイアウトの偏りを防ぐために、被験者には実験ごとに２つのレイアウトを交互に使用した。例えばとある人は、１つ目のデータを Ogushi レイアウトで描いた図での計測→２つ目のデータを Kanon レイアウトで描いた図での計測→３つ目のデータを Ogushi レイアウトで描いた図での計測、という順番で実験をしていき、別の人は１つ目のデータを Kanon レイアウトで描いた図での計測→２つ目のデータを Ogushi レイアウトで描いた図での計測→３つ目のデータを Kanon レイアウトで描いた図での計測、という順番で実験をしていった。図 3.3、3.4 は被験者に渡した二分木のレイアウトの１例である。

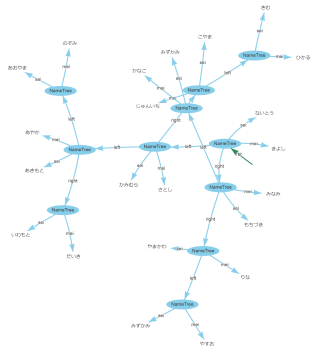


図 3.3: 二分木の Kanon レイアウト

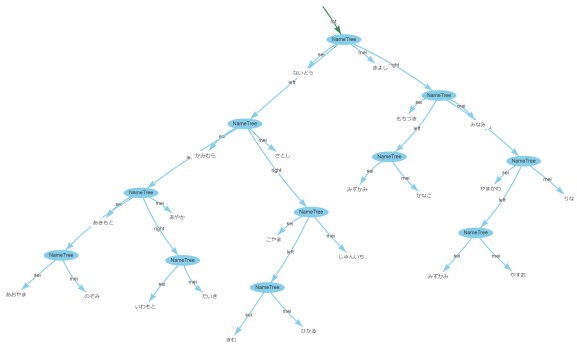


図 3.4: 二分木の Ogushi レイアウト

以下、三分探索木に関するテスト・リストが格納されている二分木に関するテストも同様の流れで１人につき３回ずつ実験を行った。

3.3.3　三分探索木に関するテスト

3.3.3.1　目的

この実験の目的は、Ogushi レイアウトによりレイアウトされた図によって使用者がリスト・二分木以外データ構造の参照関係をどれだけ追いやすくなっているかを確認することである。具体的には、三分探索木にとあるルールでで格納された英単語を全て羅列するまでの時間を計ることで参照関係の追いやすさを定量的に計ろうとした。三分探索木の性質上被験者は left・right フィールドと middle フィールドを区別しなければならないため、我々は同一のフィールドエッジの傾きが揃っているほどフィールドエッジが区別しやすくなり単語羅列までの時間が短くなると考える。

3.3.3.2　手順

被験者には以下のことを事前に紙と口頭の両方で説明する。また、事前の予備テストにより三分探索木というデータ構造があまり広く知られておらず、仕組みの理解が困難であることが分かっていたので、具体例を示した画像を一つ用意し実験前に一通りのタスクの流れを説明した。

1. とあるプログラマは以下の条件を満たした三分探索木を実装しようと考えた。

    - 各ノードには一つのアルファベットが格納されている。
    - 各ノードは、(1) そのノードに格納されているアルファベットの代わりにアルファベット順で前になる文字を指す左ノード、(2) そのノードに格納されているアルファベットの代わりにアルファベット順で前になる文字を指す右ノード、(3) そのノードに格納されているアルファベットの次の文字を指す中央ノード、の 3 種類の子ノードを持つ。
    - 三分探索木から格納されている文字列（単語）を取得するには次の操作を行う。
        (a) 根のノードから「子に中央ノードを持たないノード」まで辿り、その全ての経路上に格納されているアルファベットを羅列する。
        (b) また、それぞれの経路についてノード間に対応するエッジのフィールド名を書く。
        (c) それぞれの経路に対して根から順に left エッジと right エッジの出発点のアルファベットを消す。

2. そのプログラマは上記の条件を満たすように三分探索木を JavaScript で実装し、5 文字以内の長さの英単語をランダムに 5 個ツリーに格納させ、Kanon によってレイアウトさせた。

3. どのような英単語がツリーに格納されたのかを確かめるために、レイアウトされた図から全ての格納された英単語を列挙してほしい。

その後、被験者には実際にデータがレイアウトされた図の印刷された紙を渡し、紙を見始めてから全ての格納された英単語を羅列するのにかかる時間を計測した。被験者にはレイアウトされた図の紙とは別の白紙を渡し、そこに英単語を書いていってもらった。被験者が羅列した単語が全て合っていればそこで測定を終了し、間違えていた場合は時間の計測を一旦停止し、間違っていた旨を被験者に伝えて再び計測を再開し、正しい単語を全て列挙するまで続けた。また、10 分以内に完答できなかった場合はその時点で計測を終了した。これは被験者の心的負担を減らすためである。図 3.5、3.6 は被験者に渡した三分探索木のレイアウトの 1 例である。

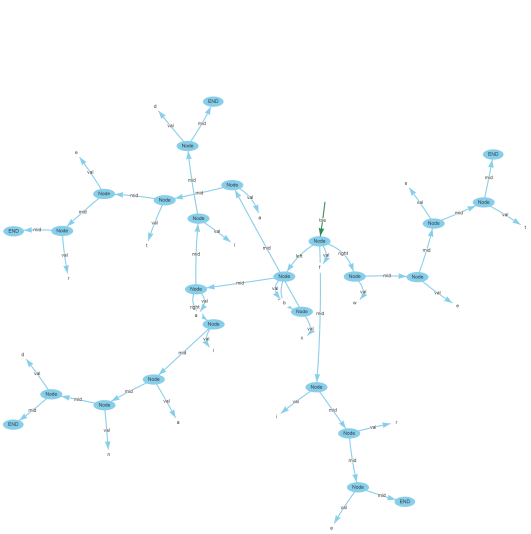


図 3.5: 三分探索木の Kanon レイアウト

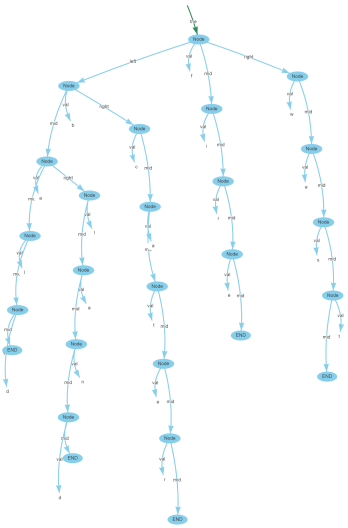


図 3.6: 三分探索木の Ogushi レイアウト

この一連の流れを一人につき 3 回繰り返した。また、この実験のみ 1 回目の計測のときに被験者が誤った単語を羅列していた場合にはこ著者から正しい答えとその求め方を提示し、問題の主旨を被験者に理解してもらうように努めた。

3.3.4　リストが格納されている二分木に関するテスト

3.3.4.1　目的

この実験の目的は、Ogushi レイアウトによりレイアウトされた図によって使用者がリスト・二分木が入れ子になっている構造の参照関係をどれだけ追いやすくなっているかを確認することである。具体的には、リストが格納された二分木の中から最も長さの長いリストを格納している二分木ノードを発見するまでの時間を計ることで参照関係の追いやすさを定量的に計ろうとした。複数のクラスノードがあるような複雑なデータ構造では異なるクラスノードを区別しなければいけないため、我々は同一のフィールドエッジの傾きが揃っているほどエッジの傾きからクラスノードの種類が判別しやすくなり特定のノード発見までの時間が短くなると考える。

3.3.4.2　手順

被験者には以下のことを事前に紙と口頭の両方で説明する。

1. とあるプログラマは以下の条件を満たしたデータ構造を実装しようと考えた。
    - 全国にあるサーキットと各サーキットでのタイム記録を管理したい。
    - リストノードにはあるサーキットでのタイム記録が格納されており、先頭からサーキット内でのタイムが速い順になっている。
    - 二分木ノードにはサーキット固有の ID 番号が格納されており、二分探索木となっているのでサーキットの挿入・削除・検索が素早く行える。
    - 二分木ノードからはそのサーキット内でのタイム記録を管理するリストの先頭を参照できる。
2. そのプログラマは上記の条件を満たすようにデータ構造を JavaScript で実装し、実際にいくつかのサーキットとタイム記録のデータをツリーに格納させ、Kanon によってレイアウトさせた。
3. このレイアウトされた図を見て、どのサーキット（二分木ノード）が最も多くの記録（リストノード）を持っているかを探してほしい。

その後、被験者には実際にデータがレイアウトされた図の印刷された紙を渡し、紙を見始めてから最も長いリストを持つ二分木ノードを指示する

までにかかる時間を計測した。被験者が指示した箇所が合っていればそこで測定を終了し、間違えていた場合は時間の計測を一旦停止し、間違っていた旨を被験者に伝えて再び計測を再開し、正しい箇所を指示するまで続けた。また、10 分以内に正しい箇所を指示できなかった場合はその時点で計測を終了した。これは被験者の心的負担を減らすためである。図 3.7、3.8 は被験者に渡したレイアウトの 1 例である。

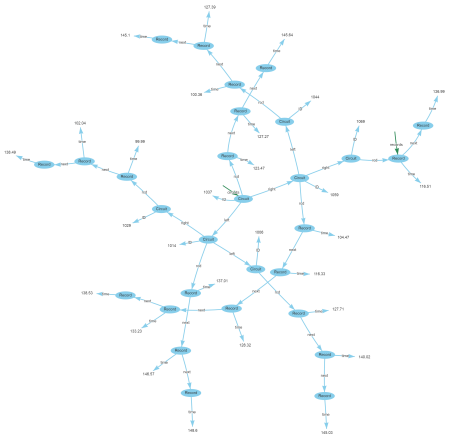


図 3.7: 入れ子構造の Kanon レイアウト

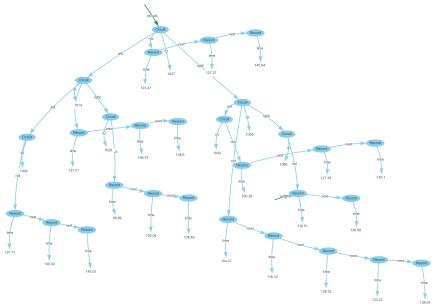


図 3.8: 入れ子構造の Ogushi レイアウト

この一連の流れを一人につき 3 回繰り返した。

3.3.5　インタビュー

各実験の終了後ごとに、著者から被験者にいくつかの質問を行った。行った質問は以下の通りである。

1. 図の読み取りと与えられたタスクの計算・理解にかかった時間の比率
2. 与えられたデータ構造の種類を知っていたか
3. どのレイアウトが最も視認しやすかったか、またその理由
4. どのようにすればより理解しやすいレイアウトになるか

1 つ目と 2 つ目の質問の役割は実験ごとのタスクの難易度を被験者がどのように感じていたかを確認することである。2 つ目の質問は、「名前を知っており、構造を他人にも説明することができる」「名前は知っているが、構造がどんなものであるかは知らない」「名前も知らない」の 3 択か

ら選ぶように尋ねた。3つ目の質問は Ogushi レイアウトと従来の Kanon レイアウトのどちらがより使用者の理解補助に役立っているのかを確認することが目的である。そして４つ目の質問ではレイアウトに関する具体的な意見や改善の可能性について尋ねることが目的である。

また、１つ目のデータ構造を自身で描く実験では「なぜそのような形に描画したのか・意識したこと」も被験者に尋ねた。

3.4　結果

3.4.1　様々な種類のデータ構造のメンタルマップに関するテスト

メンタルマップの描画にかかった時間は表 3.1 のようになった。

表 3.1: メンタルマップの描画にかかった時間

	B+木		変数環境		グラフ		スキップリスト	
	Kanon	Ogushi	Kanon	Ogushi	Kanon	Ogushi	Kanon	Ogushi
サンプル数	5	5	5	5	15	5	5	5
平均（秒）	604.2	355.2	299.8	265.2	179.6	262	428.4	514
標準誤差	73.7566	35.7245	40.8894	26.4469	34.2178	63.2242	63.7672	62.3643
t 値	3.03833		0.71052		1.1462		0.95971	
p 値	0.01406		0.49538		0.28127		0.42137	

また、10 人の被験者に描いてもらった図の各ノードの座標からそれぞれのフィールドエッジの角度を求め、同一のフィールドエッジでの平均と分散を計算した。この分散の値が小さければ小さいほど同一のフィールドエッジが同じ方向を向いていることになる。結果は図 A.9〜図 3.12 のようになった。この図では 10 人のフィールドエッジの分散を箱ひげ図を用いて表している。さらに被験者に紙で渡した Kanon でのレイアウトにおける各フィールドエッジの分散を四角の印で、Ogushi レイアウトにおける分散をバツの印で表している。ここで、縦軸の目盛りは対数になっていることに注意したい。

3.4.2　その他のテスト

二分木に関するテスト・三分探索木に関するテスト・リストが格納されている二分木に関するテストのそれぞれにおける結果は表 3.2 のようになった。

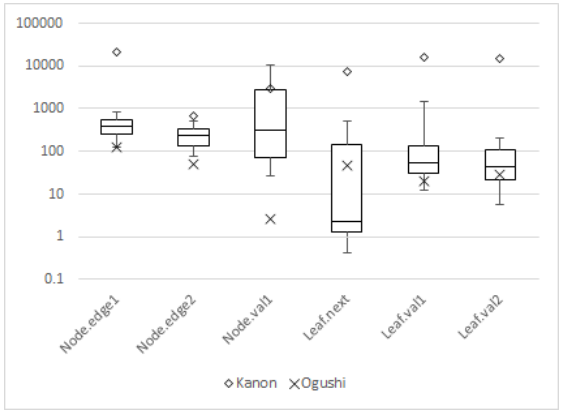


図 3.9: B+木：エッジ角度の分散

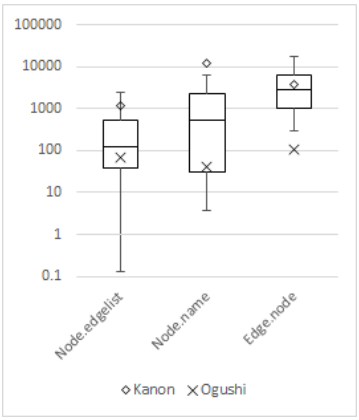


図 3.10: グラフ：エッジ角度の分散

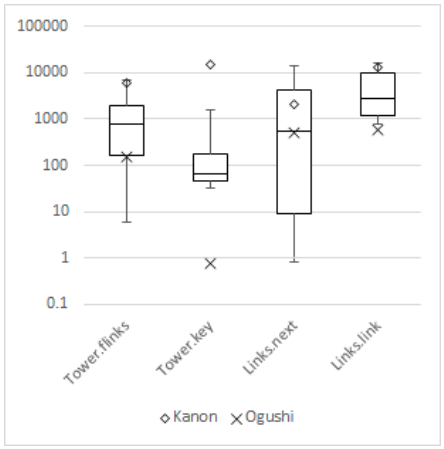


図 3.11: スキップリスト：エッジ角度の分散

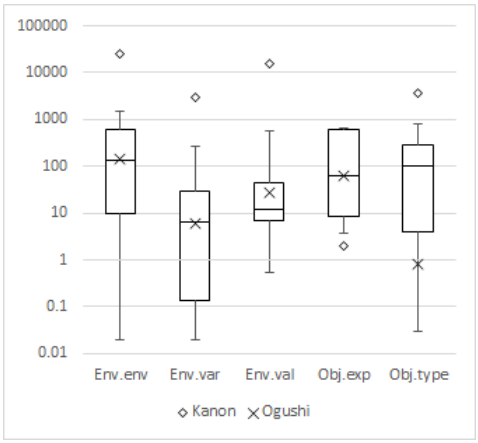


図 3.12: 変数環境：エッジ角度の分散

表 3.2: 各実験の結果

	二分木		三分探索木		入れ子構造	
	Kanon	Ogushi	Kanon	Ogushi	Kanon	Ogushi
サンプル数	15	15	15	15	15	15
平均（秒）	164.533	110.533	295.133	185.333	62.667	18.6
標準誤差	33.0729	36.0061	29.8762	26.5808	13.8188	3.4473
t 値		1.10452		2.74575		3.09407
p 値		0.27845		0.01026		0.00434

3.4.3　インタビュー結果

「与えられたデータ構造の種類を知っていたか」という質問について、B+木・スキップリスト・三分探索木については 8 割以上が「名前も知らない」と答え、二分木や入れ子構造については 9 割以上が「名前を知っており構造を他人に説明することができる」と答えた。

データ構造を自身の理解しやすい形に描画するときに意識したことについて、以下のような解答が挙げられた。

- 「同一の階層にあるノードを同一の高さになるようにした」
- 「エッジの交差を出来るだけ少なくしようとした」
- 「エッジの向きに統一性を持たせようとした」
- 「木構造の形では上から下に順番に描画していった」
- 「プリミティブ値を指すエッジの向きを同一方向に揃えようとした」
- 「異なるクラスは異なる形で表そうとした」
- 「根のノードを最も上に描くようにした」
- 「リストのようなノードは横向きに揃えるようにした」
- 「プリミティブ値はノードの内部に描くようにした」

図の読み取りと与えられたタスクの計算・理解にかかった時間の比率について、解答してもらった比率の平均値は表 3.3 のようになった。また、表 3.3、3.2 からそれぞれの実験でのタスクの計算・理解にかかった時間の平均を求めると、二分木ではおよそ 78 秒、三分探索木ではおよそ 107 秒、入れ子構造ではおよそ 21 秒となった。このことから、入れ子構造での実

表 3.3: 図の読み取りとタスクの計算にかかった時間の比率の平均

	二分木		三分探索木		入れ子構造	
	Kanon	Ogushi	Kanon	Ogushi	Kanon	Ogushi
	52：48	29.3：70.7	67.7：32.3	36：64	56：44	26.1：73.9
計	40.7：59.3		51.8：48.2		41.1：58.9	

験タスクは被験者からしてみたら比較的簡単なものであり、三分探索木での実験タスクは難しいものであったということができる。

また、「どのレイアウトが最も視認しやすかったか」という質問では、全ての実験において Ogushi レイアウトで描画されたものが選ばれた。その理由としては、

- 「同一のフィールドが同一の向きに揃っていて見やすかった」
- 「エッジの向きが上から下への向きに統一されているのが良かった」
- 「見てすぐに木構造であると理解できた」
- 「根のノードが最も上にあった」
- 「（入れ子構造で）異なるクラスオブジェクトの区別がつきやすかった」
- 「(Kanon レイアウトでは）エッジの重なりが多く見辛かった」

などが挙げられた。また、「どのようにすればより理解しやすいレイアウトになるか」という質問に対しては、

- 「異なるクラスオブジェクトやエッジを色分けすれば見やすくなると思った」
- 「オブジェクトを指すエッジとプリミティブ値を指すエッジの表現方法を変えてほしい」
- 「プリミティブ値はノードの中に描画してほしい」
- 「エッジと文字が重なっている部分があるのが見辛い」
- 「エッジの交差をもっと少なくしてほしい」
- 「異なるクラスのフィールドエッジの長さを変えたほうがより見やすい」

などの解答が返された。

3.5　考察

このセクションでは、章のはじめに挙げたリサーチクエスチョンに対しての答えをそれぞれ考察していく。

3.5.1　同一のフィールドエッジを同一の向きに揃えることの有用性

図 A.9〜3.12 より、中央値は全て Kanon レイアウトでの分散よりも小さくなっていることが分かる。このことから、個人差はあるもののおおよそ被験者が頭の中に思い描くデータ構造の図では同一のフィールドエッジを同一の向きに揃えていると言える。

しかし、図 3.10 の Edge.node や図 3.11 の Links.link は全体的にエッジの傾きの分散が高い傾向にある。これらのエッジはいずれも環状構造を作っていたり複数のエッジが同一のノードを指すなど、木構造ではなくなっているのが特徴である。このような構造は環状に描いたり放射状に描いている被験者が多かった。

このことから、「同一のフィールドエッジを同一の向きに揃えてのレイアウト」は根から順に枝分かれしていく木構造の描画には適しているが、閉路を持った構造やエッジが収束するような構造の描画には必ずしも適しているとは言えない。

3.5.2　Ogushi レイアウト vs. Kanon レイアウト

表 3.2 から、三分探索木とリスト・二分木の入れ子構造に関しては p 値 < 0.05 となっており、Ogushi レイアウトと Kanon レイアウトでのタスク時間に有意差ありと見なすことができる。また、表 3.3 から、難易度の異なるタスクにおいても Ogushi レイアウトでの図の読み取り時間は Kanon レイアウトでの図の読み取り時間よりも短くなっていることが分かる。さらに、インタビュー結果からも被験者にとって Ogushi レイアウトは肯定的な印象を持たれていることが明らかになった。

しかし、二分木の実験ではタスクの平均時間は Kanon レイアウトよりも Ogushi レイアウトのほうが短いが、p 値の結果より有意差有りとは見なせない。この理由について、著者は以下のように考える。

二分木の実験の 2 回目に用いた Kanon レイアウトの図と Ogushi レイアウトの図はそれぞれ図 3.13、3.14 である。どちらも同じデータを描画したものであるが、この 2 つのレイアウトに関してのみ Kanon レイアウトのほうがタスク終了にかかった時間の平均が短かった。インタビューでもこの実験については「難しかった」という意見が挙がっている。その理由について、まず根のノード付近に苗字が「おお」で始まる名前を格納した

ノードが 3 つもあり、これらのノードの大小関係を読みほどくのに時間がかかったのではないかと推察される。また、図 3.13 の Kanon レイアウトはエッジの重なりが少なく綺麗に放射状にノードが配置されており、また根のノードから反時計回りの向きに left フィールドと right フィールドが順番に出てくるようになっている。そのためある被験者からはこの Kanon レイアウトに関して「あっさり答えのノードを見つけることができた」と述べている。

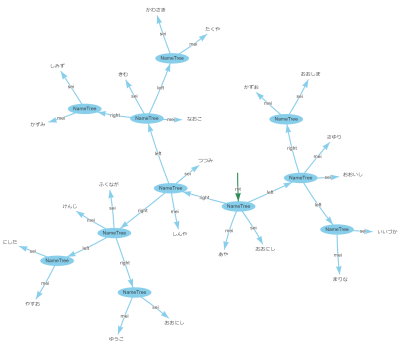
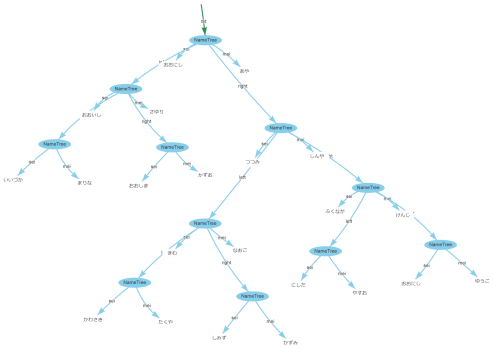


図 3.13: 二分木の Kanon レイア　図 3.14: 二分木の Ogushi レイアウト
ウト



このことより、Ogushi レイアウトはおよそ Kanon レイアウトよりも参照関係が視認しやすい形になりやすいが必ずしもそうなるわけではないということが言える。

3.5.3　レイアウトできるデータ構造の種類の汎用性

上記の考察や表 3.2 から、Kanon レイアウトよりも綺麗に描画できるデータ構造の種類は多くなったと言うことができる。しかし、複数のエッジが同一のノードを指すようなグラフ構造やスキップリストなどの構造に関しては Ogushi レイアウトで必ずしも綺麗に描画できるとは限らず、改善の余地がある。

3.6　関連研究

Collabode [13] は Web ベースの Java 統合開発環境であり、ユーザー実験によって定性的および定量的に評価されている。定性的評価はアンケート段階で観察された参加者の考えに基づいている。

Helen C. Purchase らの研究 [14] では UML クラス図の自動レイアウトにおける重要な美的観点をユーザー実験を通して明らかにしようとしている。

第4章　大規模データ利用のためのビジュアルUIの改善

この章では現在のライブデータ構造プログラミング環境における問題点の１つを解決するための機能を提案する。まず、既存のライブデータ構造プログラミング環境で大規模データを利用する際の問題点について説明する。次に、問題点を解決するためのメカニズムや改善したUIについて説明する。それから実装方法と機能が役立つことを示すケーススタディを紹介し、最後に評価手法についての議論を行う。

4.1　問題点

4.1.1　具体例：フィボナッチヒープの実装

具体例を用いて説明していく。プログラマはフィボナッチヒープをコードを書いているものとする。フィボナッチヒープを任意数の子ノードを持つことが出来るツリークラスと各ツリーを参照できる環状の双方向連結リスト、さらに双方向連結リストを操作するためのフォレストクラスを用いて定義したとし、ツリーは子ノードを配列から参照するものとする。フォレストクラスが必要な理由は、フィボナッチヒープの操作でルートの値が最も小さいツリーを参照しているリストノードを参照できるようにするためである。List4.1はフィボナッチヒープのクラス定義までを終えた時点でのコードである。

Listing 4.1: enum_sameProperty

```
1 class Tree {
2   constructor(val) {
3     this.val = val;
4     this.children = [];
5   }
6 }
7
8 class TreeList {
```

```
 9    constructor(tree) {
10      this.tree = tree;
11      this.next = this;
12      this.prev = this;
13      this.parent = null;
14    }
15  }
16
17  class Forest {
18    constructor() {
19      this.treelist = null;
20    }
21  }
```

プログラマは次にオブジェクト生成式を書いていくことでテストケースを作成していく。図 4.1 のようなイメージ図の具体例を生成する式を記述し、Kanon にレイアウトさせると図 4.2 のようになる。

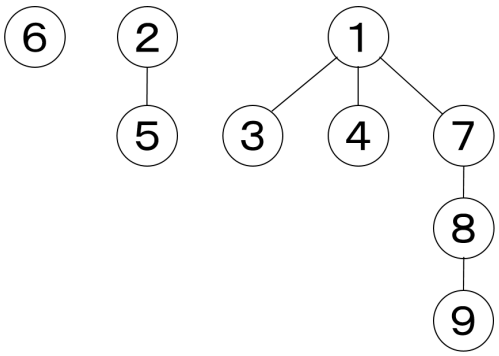


図 4.1: フィボナッチヒープのイメージ図

プログラマがここに、最小値ノードを削除するための removeMin メソッドを追加する場合を考える。新しいクラスメソッドを定義するとき、Kanon を使用するプログラマはまずメソッド呼び出し式を書き、その後具体的なメソッド定義式を書いていく。このとき、プログラマは画面右にレイアウトされた図を見ながらオブジェクトグラフのどの部分が変化するのかを発見し、その付近の構造を認識する。

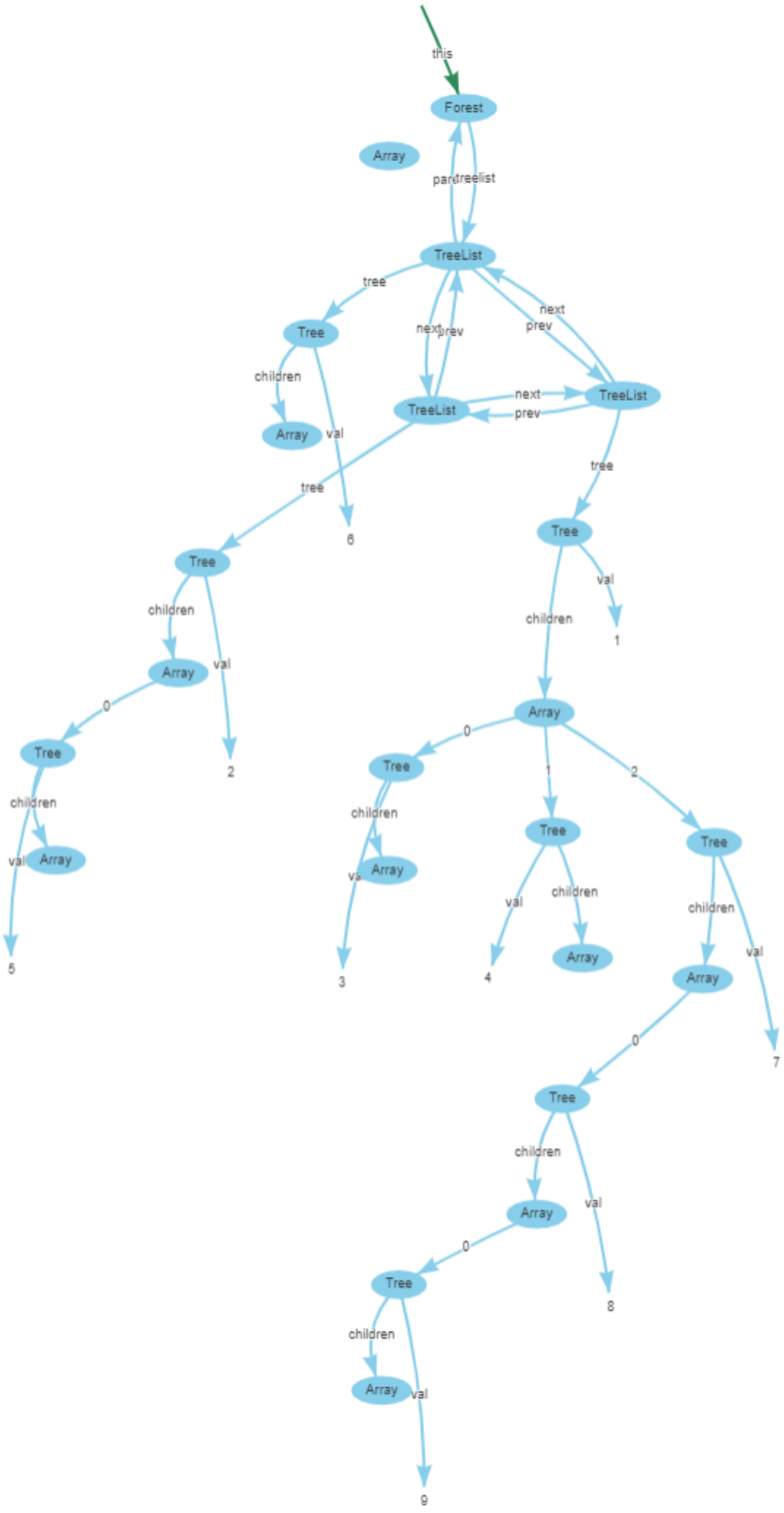


図 4.2: Kanon にレイアウトされるフィボナッチヒープ

次にメソッドの中身を記述し、画面右側のレイアウトされた図を見て変化の様子を観察する。List4.2 はメソッドの中身を記述中のコードである。

Listing 4.2: enum_sameProperty

```
1  ...
2  class Forest {
3    constructor() {
4      this.treelist = null;
5    }
6
7    removeMin() {
8      //writing second.
9    }
10 }
11
12 let forest = new Forest();
13 ...
14 forest.removeMin();    //writing first.
```

4.1.2　データの巨大化とその問題点

フィボナッチヒープを実装中のプログラマが頭の中に思い描いている図 4.1 と実際に Kanon に描画される図 4.2 ではレイアウトされている構造自体が大きく異なる。これは、図 4.1 ではプログラマはツリーノードしか思い描いていないが、実際に表示される図 4.2 にはツリーノード以外にもエディタ内で定義したリストオブジェクトや配列オブジェクトなどが加わっているためである。

このように、データ構造の種類によっては定義しなければならないクラスが増えてしまい、プログラマからの重要度が低いノードやエッジが増えてしまう。重要度の低いノードやエッジが増えることは、オブジェクトグラフが巨大化することに繋がる。このことによる問題点は２つあると考えられる。

１つ目は、オブジェクトの参照関係を追いづらくなることである。例えば、図 4.2 の中でプログラマが１の格納されているツリーノードに下に３つのツリーノードがあることを確認しようと思ったとき、実際に描画される図 4.2 では１を参照しているツリーノードの下には配列ノードが１つあ

るだけであり、配列ノードを経由してさらに下に３つのツリーオブジェクトがあることを確認しなければならない。このように重要度の低いノードやエッジが増えることでプログラマは図からオブジェクトの参照関係を追うときに余計な手間がかかることになる。

２つ目は、ノード数が多くなるため図の中でプログラマが注目している箇所の発見・認識が困難になってしまう点である。オブジェクトグラフが巨大化すると、相対的に１つ１つのノードは小さく表示されるようになる。マウス操作でプログラマは図の中の任意の箇所を拡大表示することができるが、巨大なオブジェクトグラフの中で注目箇所を手動で探し拡大操作を行うことはプログラマにとって非常に手間のかかる作業である。例えば、List4.2 を記述している時点でプログラマは最も小さい値１の格納されているツリーノードに注目しようとするが、図 4.2 の中から１の格納されているツリーノードを素早く見つけることはとても困難である。

4.2　提案手法

我々の目標は、巨大なデータ構造を可視化するライブプログラミングで重要度の高いノードやエッジを見やすくすることと、プログラマが注目する箇所の発見・認識を簡単にすることである。これらの目標のために、我々はエディタからプログラマのオブジェクトグラフ内での注目箇所を自動推定し、注目点から参照距離やクラス・フィールドの重要度を使ってグラフ内のそれぞれの箇所の拡大率を変更・決定する手法を提案する。

この提案手法により、プログラマはコードを編集するだけで注目したい箇所付近のグラフが拡大表示・注目されていない不要な箇所が縮小表示され、データ構造の変化を観察しやすくなる。　また、ノードやエッジの重要度をクラス別にプログラマが手動で指定することで重要度の低いクラスのノードやエッジが縮小表示され、より参照関係を視認しやすくすることも可能となる。

4.2.1　各ノードの重要度の決定

それぞれのノードの重要度を決定するために、

1. プログラマが指定した各クラスの興味の有無
2. エディタから自動推定した注目ノード

の２つの情報を利用する。プログラマがエディタにコードを記述しテストケースを作成すると画面に図のようなチェックボックスが表示され、プログラマは表示するのに不要であるクラスを選択することができる。また、

エディタ内のカーソル位置から見えるローカル変数の指すノードをプログラマの注目ノードと自動的に判断し、注目ノードからの遠さによって各ノードやエッジの重要度を変更する。これは、ローカル変数に束縛されたオブジェクトはそのメソッド内で何かしらの変更が加えられる可能性が高いからである。

図 4.3: 各クラスの興味の有無を決めるチェックボックス

4.2.2　FiFA レイアウト：重要度に応じて拡大率を変更するレイアウト手法

上記の手法で決定された各ノード・エッジの重要度を利用し、重要度の高いノードやエッジを拡大表示し、重要度が低ければ低いほど縮小表示されるような新しいレイアウト手法を提案する。このレイアウトを FiFA（Fisheye-view and Force-directed Automatic）レイアウトと呼ぶことにする。このように表示することで、プログラマは重要度の高いノード付近の詳細な情報と全体の構造を両方同時に確認することが可能になる。

4.3　実装

Kanon のユーザーインターフェースを拡張し、FiFA レイアウトのアルゴリズムを導入することで上記の提案手法を実現した。これはオンラインで入手することができる。このセクションでは上記の提案手法以外にも改善したユーザーインターフェースについても説明していく。

4.3.1　カーソル位置を利用した注目ノードの発見

クラスメソッドや関数の記述の中にマウスカーソルがある場合、マウスカーソルのある関数内部のローカル変数に束縛されているオブジェクトノードをプログラマの注目ノードとする。このとき、関数の外にあるグローバル変数に束縛されているオブジェクトノードは注目ノードとは見なさない。グローバル変数にもローカル変数同様何かしらの変更が加えられる可能性はあるが、グローバル変数まで全てプログラマの注目ノードとして見なすと注目ノードの数が増えてしまい、かえってレイアウトが見辛くなってしまうためグローバル変数はプログラマの注目ノードから除外する。

4.3.2　FiFA レイアウトアルゴリズム

FiFA レイアウトアルゴリズムはばねレイアウトアルゴリズムと魚眼レイアウトアルゴリズムを拡張して実装されている。

4.3.2.1　概要

まず各ノードと注目ノードとの最短経路長を計算する。最短経路長の計算にはフロイドワーシャル法を用いる。次に計算された最短経路長を元に各エッジの理想長と各ノードのサイズを計算する。この時点でプログラマが興味なしと指定したクラスに所属するオブジェクトノードのサイズとこれらのノードを指すエッジの理想長を小さくする。その後、各エッジが理想長・理想角度となるように各ノードにスプリング力・斥力・回転力の 3 つの力を働かせ安定座標を探索していく。

4.3.2.2　エッジの理想長の計算

エッジ e の理想長 $L_{1,2}$ は

$$L_{1,2} = c_1 \frac{|g(P_1) - g(P_2)|}{I(P_1, P_2)} \tag{4.1}$$

という計算式で導き出される。ただし、c_1 は定数、P_1 はエッジの始点、P_2 はエッジの終点であり、

$$g(P) = \frac{d+1}{d + \frac{D_{max}}{D_P}} D_{max} \tag{4.2}$$

と定義される。ここで D_P は点 P と注目ノードとの最短経路長、D_{max} は注目ノードとの最短経路長の最大値である。また、$I(P_1, P_2)$ は P_1 と P_2 のそれぞれの所属しているクラスがプログラマに興味ありと指定されたか否かによって異なる定数を返す関数である。また、変数 d はゆがみ変数と呼ばれ、この値が大きくなるほど拡大率が増加していく。d は D_{max} の 2 乗に比例するように定め、グラフ全体が大きくなればなるほど拡大率が増加するようにする。

$g(P)$ は点 P の注目ノードからの理想距離を表しており、隣り合う点同士の $g(P)$ の差分がエッジの理想距離として計算される。このとき、注目ノードから離れれば離れるほど $g(P)$ の差分は小さくなっていく。また、ここでの注目ノードとの距離とはグラフ内の最短経路長のことでありノード同士のユークリッド距離ではないことに注意したい。

4.3.2.3　ノードの拡大率の計算

ノード P のサイズ S_P は、

$$S_P = S_{max} - \frac{g(P)}{D_{max}}(S_{max} - S_{min}) \tag{4.3}$$

という計算式で導かれる。ここで S_{max} とはノードサイズの最大値を表す値であり、S_{min} はノードサイズの最小値を表す値である。ノードサイズは注目ノードとの距離に比例して小さくなるように計算される。

また、S_{max} と S_{min} はそれぞれ、

$$S_{max} = S_0 + N^{c_2} \tag{4.4}$$

$$S_{min} = c_3 S_0 (1 + \exp(-\min\{D_{max}, N\}) \tag{4.5}$$

と計算される。ここで N はノード数、S_0 はノードの基準となる大きさを表しており、c_2、c_3 はそれぞれ定数である。ノードサイズの最大値はノード数によって決定し、ノードサイズの最小値はノード数または最短経路長の最大値によって決定する。ノードサイズの最小値をこのような方法で定義しているのは、ノード数の大きなグラフでノードサイズの最小値がグラフ全体に対して極端に小さくなりすぎるのを防ぐためである。

4.3.2.4　ノードに働く力の計算

既存の Ogushi レイアウトアルゴリズムでは、各ノードに引力 $f_a(d) = \frac{d^2}{k}$・斥力 $f_r(d) = -\frac{k^2}{d}$・回転力 f_m の 3 つの力を働かせ、ノードを力の方向に微小距離だけ動かすといったことを複数回繰り返すことで各ノードの座標を計算していた。ここでの d とは 2 つのノード間のユークリッド距離を表す。このアルゴリズムでは、ノード間の距離が k のときにちょうど引力と斥力が釣り合うので全てのエッジの長さが k に近づくようにレイアウトされた。しかし、このレイアウトアルゴリズムは FiFA レイアウトのような各エッジを異なる長さにしたい場合には不向きである。

そこで、FiFA レイアウトアルゴリズムでは各ノードに引力・斥力の代わりにスプリング力と斥力を働かせる。スプリング力はエッジで繋がれた 2 つのノード間に働き、

$$f_s(d) = c_s \log \frac{d}{L_{i,j}} \max\left\{1, \left(\frac{d}{L_{i,j}}\right)^{c_4}\right\} \frac{S_i S_j}{S_0^2} \tag{4.6}$$

の式で定義される。c_s、c_4 は定数である。スプリング力はエッジの長さが理想長よりも長い場合には引力として働き、理想長よりも短い場合には斥力として働く。また、斥力は非隣接２点間に働き、

$$f_r(d) = -c_r \frac{k}{d^2} \frac{S_i S_j}{S_0^2} \tag{4.7}$$

の式で定義される。c_r は定数である。既存の Ogushi レイアウトアルゴリズムでは全ノード間に斥力が働いていたが、FiFA レイアウトアルゴリズムでの斥力は隣接したノード間に斥力が働かないことに注意したい。また、エッジ e の端点に働く回転力の大きさは、

$$f_m(a) = c_m(a - A_e)^2 \frac{S_i S_j}{S_0^2} \tag{4.8}$$

の式で定義される。c_m は定数であり、a はエッジ e の傾き、A_e はエッジ e の理想角度を表している。回転力は Ogushi レイアウトアルゴリズムと同様エッジの傾きと理想角度の差の２乗に比例した大きさになる。

スプリング力・斥力・回転力の３つの力はいずれも２つのノードのサイズに比例して大きさが変化する。

4.3.3　その他ユーザーインターフェースの改善

4.3.4　配列ノードの表示方法の変更

既存の Kanon では配列ノードは図 4.5 のように表されていた。しかし、一般的にプログラマが頭に思い浮かべる配列図のイメージは図 4.4 のような箱型のものであり、レイアウトされる配列の図と大きく異なっている。これを解決するために、配列ノードのレイアウトをプログラマのイメージ図に近づけるための改善を行った。

具体的には、配列内の１つの要素ごとに１つリストノードのようなノードを生成し、配列ノード同士を結ぶエッジは真横の向き (0°) になるような回転力を働かせるようにした。また、通常のリストノードと区別するために配列ノードは他のノードと違った見た目になるような変更をした。図 4.6 は改善した後の配列ノードのレイアウトである。

4.3.5　マウスオーバーされたノードから辿れるノード群の色分け

よりデータが巨大化・複雑化していくと、ノードやエッジが重なって表示されてしまう可能性が増えてくる。力学的手法によるグラフレイアウトはエッジの交差が少なくなる傾向があるがエッジの交差を無くすものでは

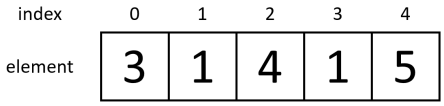


図 4.4: 配列のイメージ図

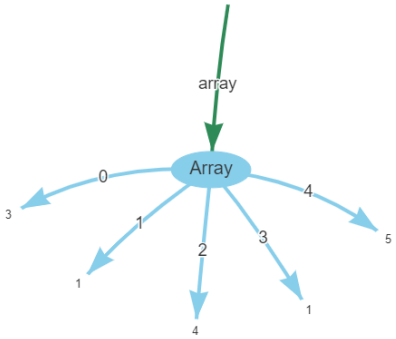


図 4.5: Kanon での配列の表示

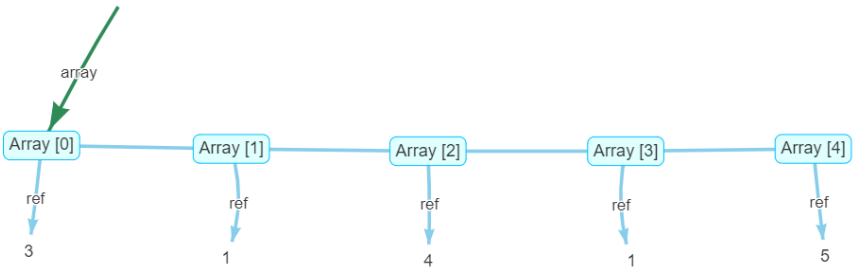


図 4.6: 変更された配列ノードレイアウト

ないため、エッジの交差によりプログラマが参照関係を視認しづらい状況が発生しうる。プログラマが参照関係を少しでも追いやすくするために、マウスカーソルをノード上に持っていくとそのノードから辿ることのできる全てのノードを異なる色で表示する機能を追加した。図 4.7 は Kanon 上のノードにマウスオーバーした際のスクリーンショットである。この図では最も左上にある配列ノードの上にマウスを持ってきている。

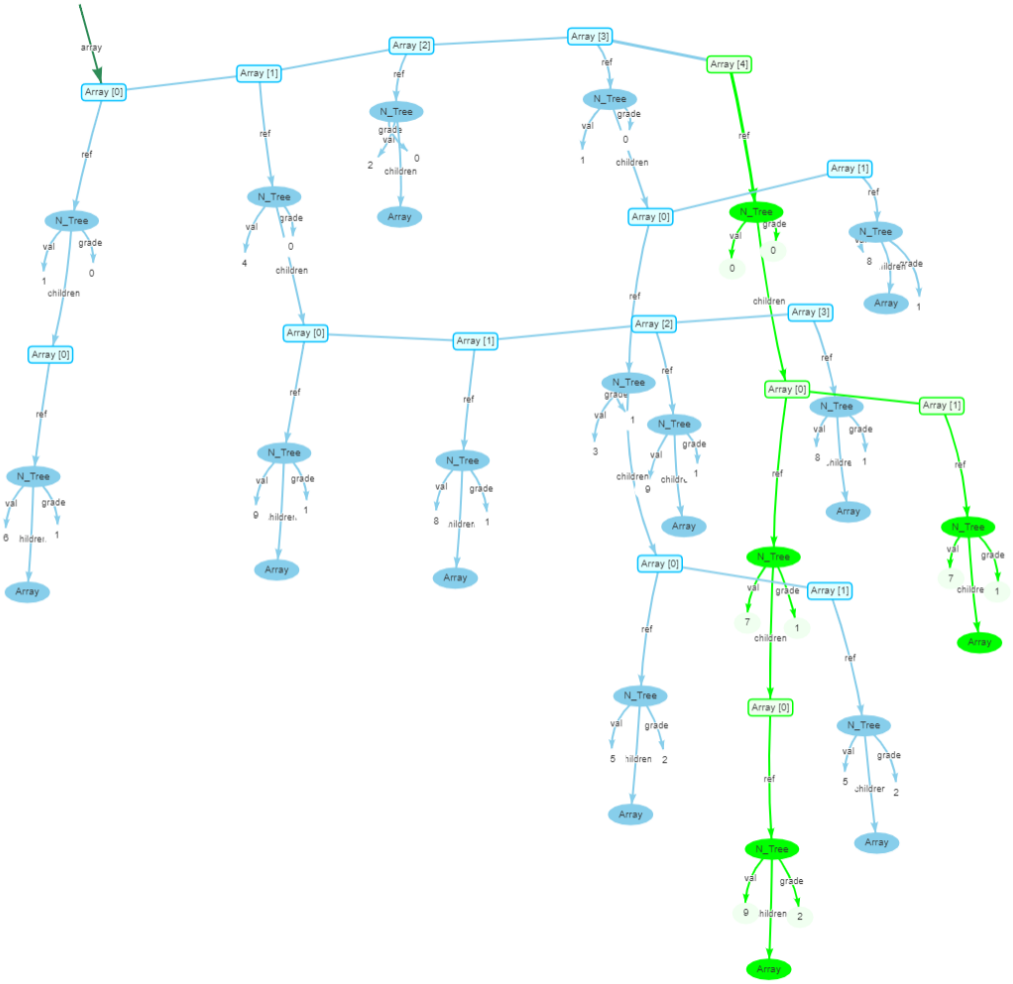


図 4.7: ノード上にマウスオーバーした例

4.4　レイアウト例

図 4.8〜4.11 はそれぞれリスト・二分木を注目ノードありの状態で FiFA レイアウトで描画した例である。全体のノード数が小さい場合は注目ノードがあるときでも Ogushi レイアウトとほぼ同じ形で描画されるが、ノード数が多くなればなるほど注目ノードがあるときにその近辺の拡大率が大きくなり注目ノードから離れれば離れるほど縮小表示されるようになる。

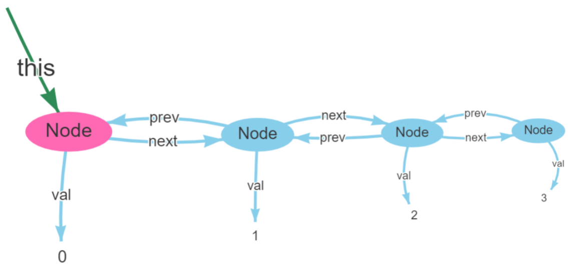


図 4.8: FiFA レイアウト：リスト

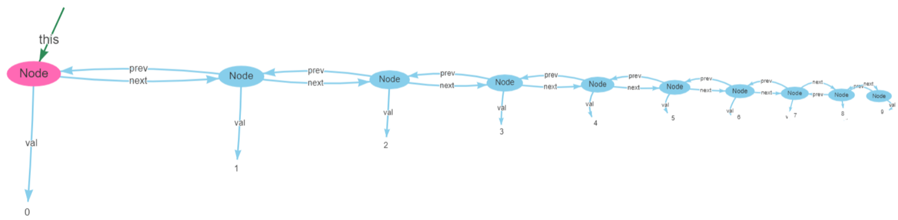


図 4.9: FiFA レイアウト：リスト（大）

図 4.12〜4.13 は二分木への参照を持ったリスト構造の注目ノードがある状態でのレイアウトである。また、図 4.14 は図 4.12 の状態からプログラマが二分木のクラスを「興味なし」と指定した場合のレイアウト結果である。二分木ノードが極小表示されることでプログラマはリスト構造をより明確に視認することができる。

図 4.15 はリストへの参照関係を持った二分木構造の描画結果であり、図 4.16 はリストノードを極小表示した結果である。通常のレイアウトでは二分木のエッジとリストのエッジに交差が見られるが、リストノードを極小表示することで二分木構造がより明確に視認できるようになる。

図 4.17 はセクション 2.1 でも述べた、グラフ構造を隣接リストで定義したもののレイアウト結果である。各 Node クラスオブジェクトは Edge クラスオブジェクトのリストへのリンクを持っており、Edge クラスオブジェクトは Node クラスオブジェクトへのリンクを持っている。この状態

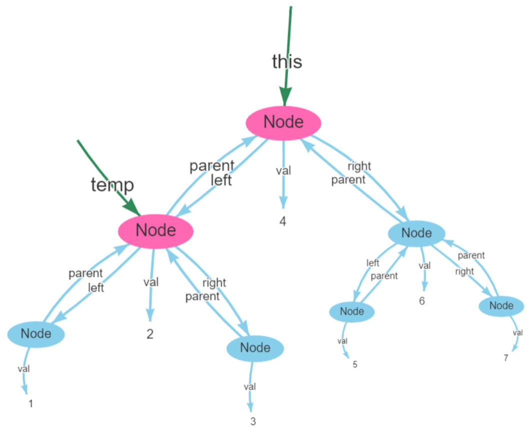


図 4.10: FiFA レイアウト :二分木

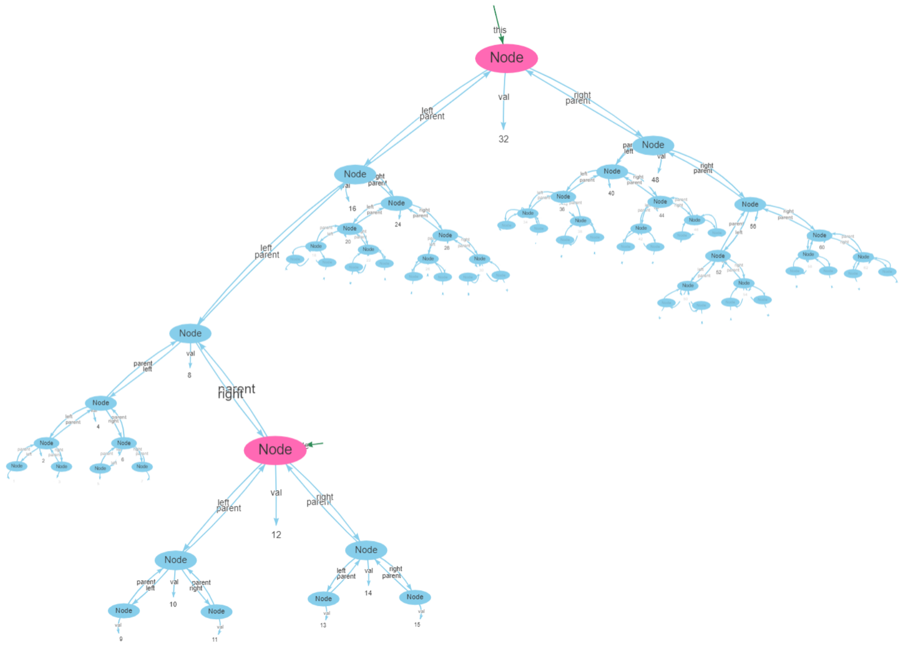


図 4.11: FiFA レイアウト :二分木（大）

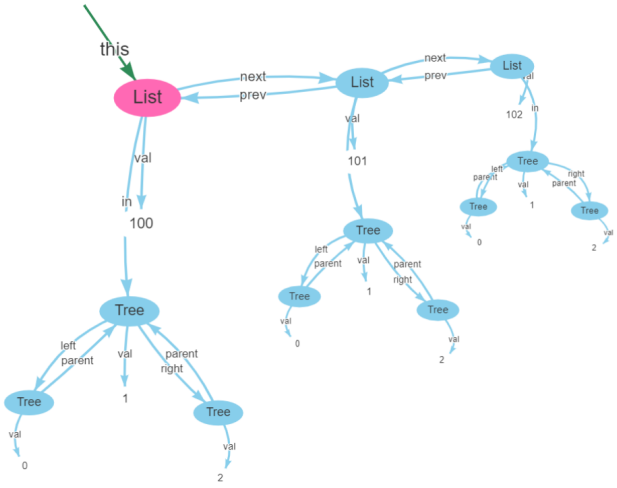


図 4.12: FiFA レイアウト：二分木を持ったリスト

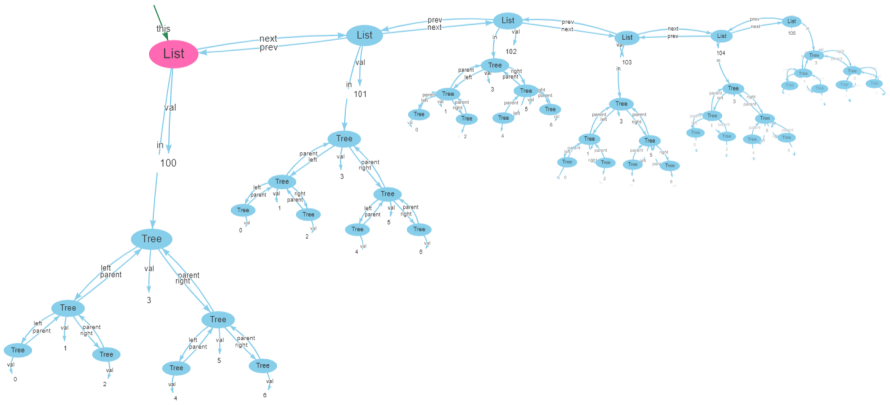


図 4.13: FiFA レイアウト：二分木を持ったリスト（大）

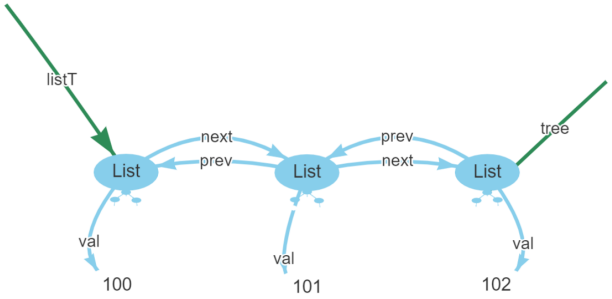


図 4.14: FiFA レイアウト：二分木を持ったリスト（二分木を極小表示）

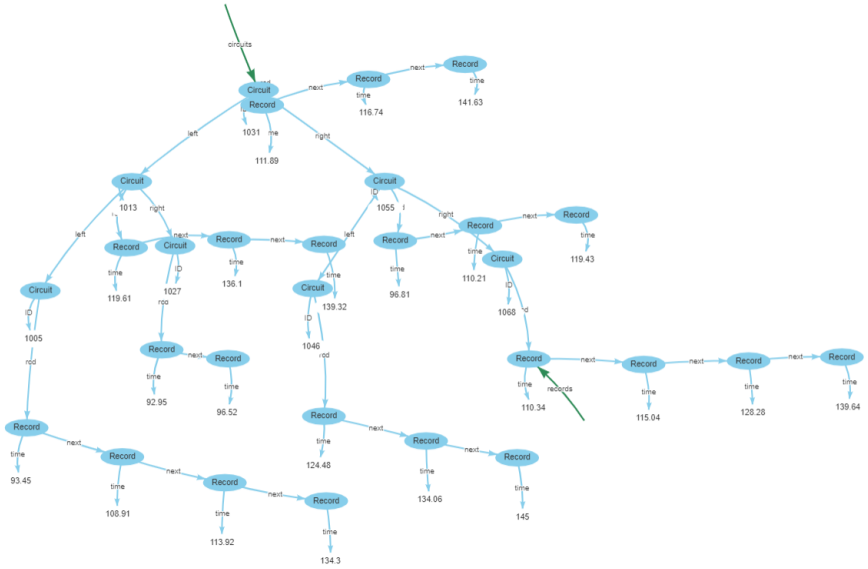


図 4.15: FiFA レイアウト：リストを持った二分木

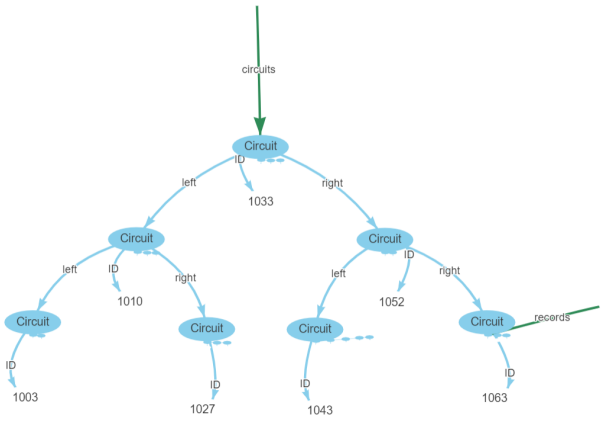


図 4.16: FiFA レイアウト：リストを持った二分木（リストを極小表示）

だとプログラマはどのような形のグラフを定義しているのか理解しづらいが、図 4.18 のように Edge クラスを「興味なし」と指定することでよりグラフ構造のイメージ図 A.8 に近い形で描画できるようになる。

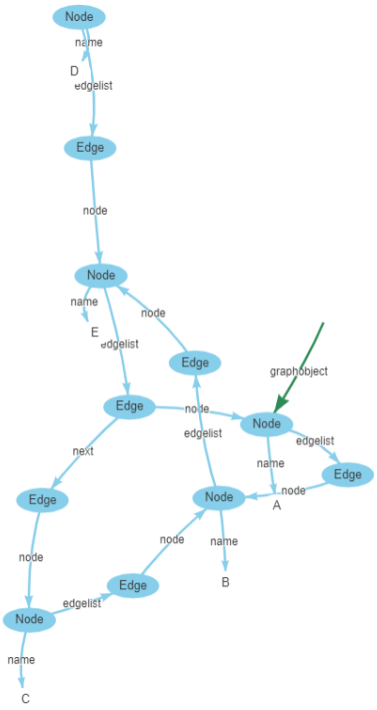


図 4.17: FiFA レイアウト：
隣接リストで定義されたグラフ

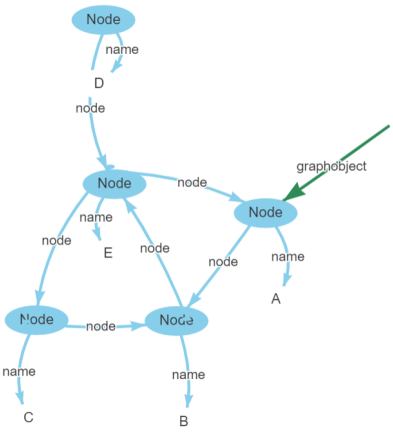


図 4.18: FiFA レイアウト：
隣接リストで定義されたグラフ
（Edge クラスを縮小表示）

同様に、この章のはじめで例として取り上げたフィボナッチヒープを FiFA レイアウトで描画した結果が図 4.19 であり、ここから Forest・TreeList・Array クラスを「興味なし」と指定してレイアウトしたものが図 4.20 である。このように描画することでよりプログラマの頭の中のイメージ図 4.1 に近い形で描画できるようになる。

4.5　評価手法について

本研究はプログラマがライブプログラミングで大規模データを利用してテストケースを作成する場合を想定している。Ogushi レイアウトの評価実験は印刷されたレイアウト図のみを利用しており「図の見やすさ」に焦点を当てた実験であったが、本研究を評価するためには実際にプログラ

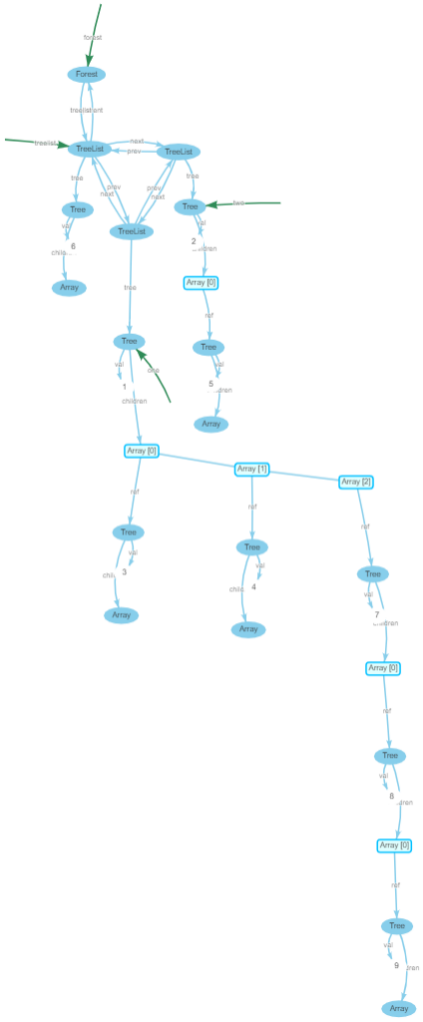


図 4.19: FiFA レイアウト：
フィボナッチヒープ

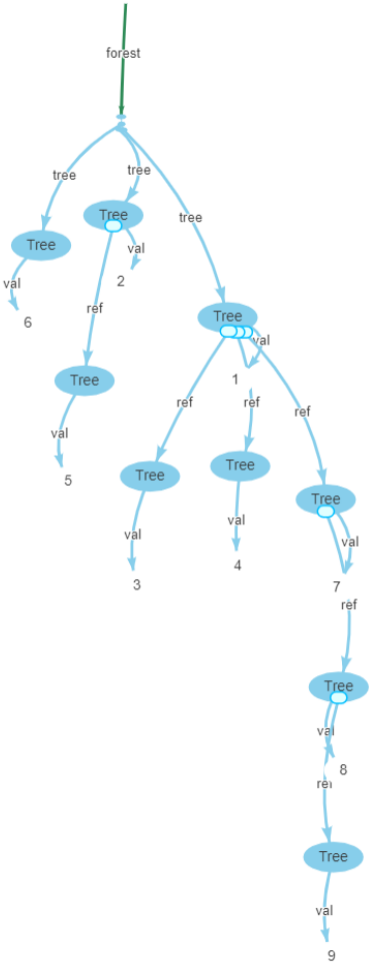


図 4.20: FiFA レイアウト：
フィボナッチヒープ
（Forest・TreeList・Array を縮小表示）

マに大規模データを利用したプログラムを FiFA レイアウトアルゴリズムの組み込んだ Kanon 上で行ってもらい、その使いやすさを計る必要がある。具体的に、フィボナッチヒープやスキップリストなど定義するクラスやフィールドの数が多くなるようなデータ構造の新たなメソッド式を定義するタスクを被験者に与え、タスクの終了時間を測定し使用感をインタビューするといった実験が考えられる。

しかしこの実験で問題となるのは Kanon の前処理の遅さである。著者が実際に挿入操作を 120 回ほど実行した二分探索木をテストケースで作成したところ、レイアウトに必要なグラフ情報の準備に 1 分以上かかっていた。このときプリミティブ値のノードも含めて合計で 250 ノードほどあるが、FiFA レイアウトアルゴリズムでのノード座標計算にかかった時間は 2.3 秒ほどであった。このことから、実際に大規模データを利用したユーザー実験を行うためには Kanon のライブ性を向上させるための工夫を考える必要がある。

4.6　関連研究

マグネティック・スプリング・モデル [11] は Ogushi レイアウトや FiFA レイアウト同様、エッジに対する回転力を導入したグラフレイアウトアルゴリズムである。このアルゴリズムはライブデータ構造プログラミングのためのものではないが、エッジに働かせる磁場を調整することで同一の種類のエッジでも異なる方向に向かせることができる。しかし、ノードやエッジの拡大率を場所によって変更するような機能は持っていない。

Willow [16] は Python Tutor [8] や Kanon 同様のデータ視覚化プログラミング環境である。エディタに記述されたコードからデータの参照関係やプログラムスタックを可視表示する機能を持っており、多くの種類のデータ構造の可視化において優れたレイアウトをすることができる。しかし、Willow は Python Tutor 同様コード編集と実行結果の確認は別々のプロセスとなっている。

第5章　結論

5.1　まとめ

本論では２つの研究について報告している。１つは Ogushi レイアウトについてのユーザー実験であり、もう１つは大規模データ利用のためのビジュアラル UI の改善についての提案・開発である。

Ogushi レイアウトに関するプログラマの意見を収集するため、定性的なユーザー実験を実施した。実験とインタビューの結果、Ogushi レイアウトは Kanon レイアウトよりも視認性の高く描画できるデータ構造の種類が増えていることが明らかになり、被験者の多くも肯定的な意見を持っていた。しかし、閉路を持った構造や複数のエッジが同一のノードを指すような構造の描画には必ずしも適しているとは言えず、まだ改善の余地があることがわかった。

また、エディタからプログラマの注目箇所を自動推定し、FiFA レイアウトと呼ばれる注目点から参照距離やクラス・フィールドの重要度を使ってグラフ内のそれぞれの箇所の拡大率を変更・決定する手法を提案する。これらの手法より、プログラマはコードを編集しながら注目したい箇所が拡大表示・注目されていない不要な箇所が縮小表示され、データ構造の変化を観察しやすくなる。また、プログラマ自身がレイアウトに不要なクラスを指定することでそのクラスに所属するオブジェクトを極小表示することも可能である。著者は実際にいくつかのテストケースを作成しそのメソッドを記述する途中で大規模データでも注目しているノードが視認しやすくなっていることを確認した。これらの機能は Kanon の拡張機能として実装されており、https://github.com/prg-titech/Kanon/tree/ogushi から入手可能である。

5.2　今後の計画

本論で提案した手法によって Kanon で大規模データを利用してデータ構造プログラミングするときの使いやすさを評価するためにユーザー実験を実施する必要がある。そのためには、コード編集後に Kanon 内で行われる前処理（ポインタの挿入、スナップショットの生成、コールツリーの

生成など）の速度をより高速化する必要がある。現在、二分探索木に挿入操作を 100 回以上実行するとこの前処理の段階で 1 分以上の時間がかかってしまう。現段階では大規模データを利用してプログラマにタスクを与えようとしてもライブ性がないため余計な負担をプログラマに与えてしまうことになる。

また、本論で提案した FiFA レイアウトやその元となった Ogushi レイアウトにはさらなる改善の余地がある。

Ogushi レイアウトでの実験を通して、閉路を持った構造や複数のエッジが同一のノードを指すような構造の描画には「同一のフィールドを表すエッジの向きを同一の方向に揃える」というレイアウト方法はあまり適していないことが明らかになった。現在、閉路上のエッジには理想角度を割り当てないようにすることで各ノード間に働く斥力に任せて描画しているが、これは必ずしも綺麗な環状にレイアウトされるとは限らない。また、複数のエッジが同一のノードを指している場合には「同一のフィールドを表すエッジの向きを同一の方向に揃える」力がむしろ余計な力として働いてしまい、かえって見辛いレイアウトになってしまう可能性がある。例えば、図 5.1 は 10 個のリストノードが全て同一のノードを指している場合のレイアウト結果である。

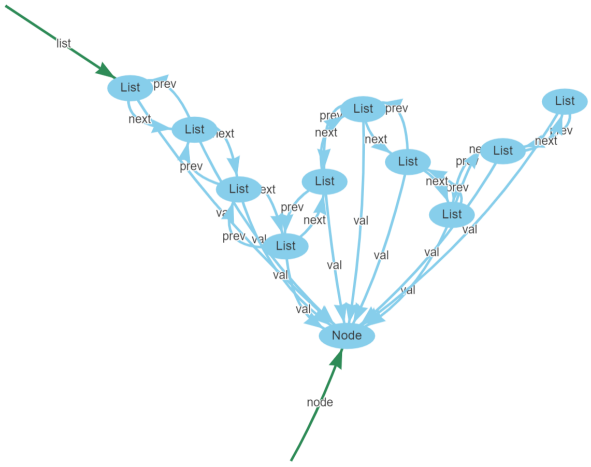


図 5.1: 複数のエッジが同一のノードを指している例

また、実験のインタビューから異なるクラスノードの色分け・プリミティブ値の表示方法の変更・同じ階層のノードの高さを揃えることなど、レイアウトに関するいくつかの改善点を被験者から指摘された。特にプリミティブ値の表示方法については複数の被験者や異なる研究 [16] からも指摘があり、改善の優先度は高いと思われる。

参考文献

[1] Oka Akio, Hidehiko Masuhara, and Tomoyuki Aotani. "Live, synchronized, and mental map preserving visualization for data structure programming." Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software(Onward! 2018). ACM, 2018.

[2] Victor, Bred (2012). Inventing on Principle. Keynote Talk at the Canadian University Software Engineering Conference (CUSEC). Montreal, Quebec.

[3] Aaron, Samuel and Alan F.Blackwell (2013). "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Language". In: Pro-ceeding of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design. FARM '13. Boston, Massachusetts, USA: ACM, pp. 35-46. ISBN: 978-1-4503-2386-4. DOI: 10.1145/2505341.2505346. URL: http://doi.acm.org/10.1145/2505341.2505346.

[4] McDirmid, Sean (2007). "Living it Up with a Live Programming Language". In: In Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) ACM, pp. 623–638.

[5] Khan Academy (2018). Intro to JS: Drawing & Animation. https://www.khanacademy.org/computing/computer-programming/programming. Accessed February 2017.

[6] Lieberman, Henry and Christopher Fry (1995). "Bridging the Gulf Between Code and Behavior in Programming". In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '95. Denver, Colorado, USA: ACM Press/AddisonWesley Publishing Co., pp. 480–486. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223969. URL: http://dx.doi.org/10.1145/223904.223969.

[7] Hendrix, T. Dean, James H. Cross II, and Larry A. Barowski (2004). "An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE". In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education. SIGCSE '04. Norfolk, Virginia, USA: ACM, pp. 387–391. ISBN: 1-58113-798-2. DOI: 10.1145/971300.971433. URL: http://doi.acm.org/10.1145/971300.971433.

[8] Guo, Philip J (2013). "Online Python Tutor: Embeddable Web-based Program Visualization for CS Education". In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. ACM, pp.579–584.

[9] T. Fruchterman & E. Reingold (1991). Graph drawing by force-directed placement. Software—Practice and Experience 21, 1129-1164.

[10] T. Kamada (1989). Visualizing abstract objects and relations, World Scientific.

[11] Sugiyama, K. and Misue, K. (1995). Graph drawing by the magnetic spring model. Journal of Visual Languages and Computing, 6(3):217-231.

[12] B.V., Almende (2018). vis.js - A dynamic, browser based visualization library. URL:http://visjs.org/.

[13] Goldman, Max, Greg Little, and Robert C. Miller (2011). "Real-Time Collaborative Coding in a Web IDE". In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology. UIST '11. Santa Barbara, California, USA: ACM, pp. 155–164. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047215. URL: http://doi.acm.org/10.1145/2047196.2047215.

[14] Purchase H. C., Mcgill M., Colpoys L., Carrington D.: Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study. In Australian Symposium on Information Visualisation (Sydney, Australia, 2001), ACS.

[15] Sarkar, Manojit and Brown, Marc H.. Graphical Fisheye Views of Graphs. Proceedings of CHI '92, (Monterey, CA, May 3-5, 1992) ACM, New York, (1992), pp. 83-91.

[16] P. Moraes and L. Teixeira, "Willow: A tool for interactive programming visualization to help in the data structures and algorithms teachinglearning process,"" in Proceedings of the XXXIII Brazilian Symposium on Software Engineering, 2019, pp. 553–558.

付 録A　データ構造の種類

データ構造はプログラマの用途に合わせて非常に多くの種類が存在している。ここでは本論で扱うデータ構造について説明する。

A.0.1　配列

配列（図 A.1）とは同一の型のデータをインデックスを元に格納していくものである。インデックスを指定することで素早くデータにアクセスすることができる。しかし、配列はあらかじめ要素数が指定されている場合が多く、データの追加や削除には向いていない。

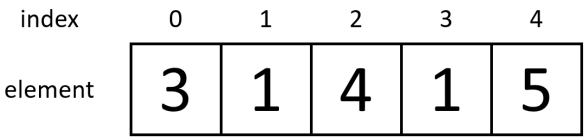


図 A.1: 配列

A.0.2　リスト

リストは各ノードが次の要素へのリンクを持っているようなデータ構造である。１方向へのリンクのみを持っているものを単方向リスト（図 A.2)、２方向へのリンクを持っているものを双方向リスト（図 A.3）と呼ぶ。また、末尾のノードが先頭のノードへのリンクを持っているようなリストは循環リスト（図 A.4）と呼ばれる。リストはデータのアクセスには O(n) の時間がかかってしまうが、要素の追加や削除はリンクを貼り替えるだけで良いためこれらの操作がしやすい。

A.0.3　二分木

木は階層構造を持つデータ構造であり、二分木は１つのノード（親ノード）から２つのノード（子ノード）にリンクを持っている。どのノード

図 A.2: 単方向リスト

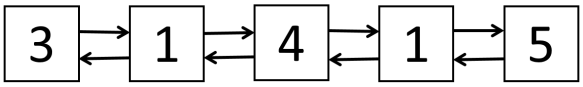


図 A.3: 双方向リスト

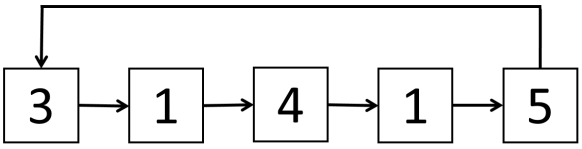


図 A.4: 循環リスト

の子にもなっていないノードを根、子ノードを持っていないノードを葉と呼ぶ。また２つの子ノードはそれぞれ右・左と呼ばれる。「左の子孫の値 ≤ 親ノードの値 ≤ 右の子孫の値」という条件を満たした二分木を二分探索木（図 A.5）と言い、リストよりもデータの検索が高速に行えるという利点を持っている。

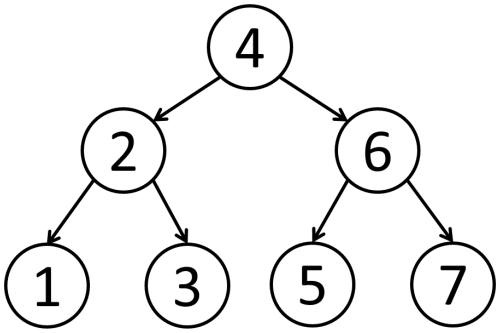


図 A.5: 二分探索木

A.0.4　トライ木

トライ木は主に文字列を扱う際に利用されるデータ構造であり、複数の文字列の共通接頭辞を１つのノードに格納する。根の要素は空であり、根から終端ノードまでを辿ることで文字列を取得することができる。トライ木は文字列の検索を高速で行うことができる。

図 A.6 は a、to、tea、ted、ten、i、in、inn の８つの英単語を格納したトライ木を表している。

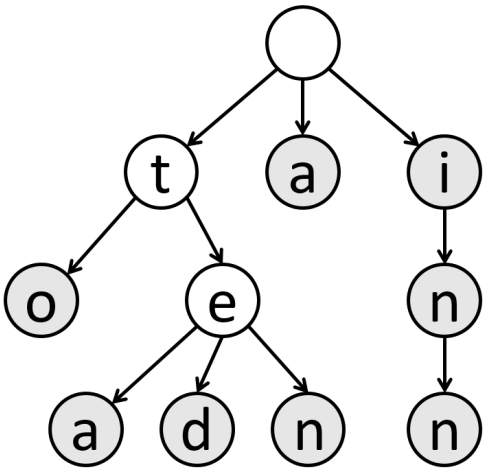


図 A.6: トライ木

A.0.5　三分探索木

三分探索木はトライ木と二分探索木の特徴を融合させたデータ構造である。各ノードは左ノード・右ノード・中央ノードの３つの子ノードを持ち、「左ノードの値 ≤ 親ノードの値 ≤ 右ノードの値」という規則を持つ。また、トライ木と同じく文字列を扱う際に用いられ、共通する接頭辞を１つのノードに格納する。トライ木では子ノードの実装に長さ 26 の配列を使用することが多く必要なメモリ容量が増えてしまうが、三分探索木ではトライ木と同等の検索時間を保ちながらメモリ使用量を抑えることができる。

図 A.7 は cute、cup、at、as、he、us、i の７つの英単語を格納した三分探索木である。この木から単語 as を検索するときの手順は以下のようなものになる。

1. 根ノードの値と検索文字列の先頭の a を比較する。a は c よりも辞書順で前になるため左ノードに進む。
2. 「a」のノードと a を比較する。文字が等しいため、中央ノードに進み、検索文字列を１文字進める。
3. 「t」のノードと s を比較する。s は t よりも辞書順で前になるため左ノードに進む。
4. 「s」のノードと s を比較する。文字が等しいため、中央ノードに進み、検索文字列を１文字進める。
5. 終端ノードに辿り着き検索文字列が終了したので検索終了。

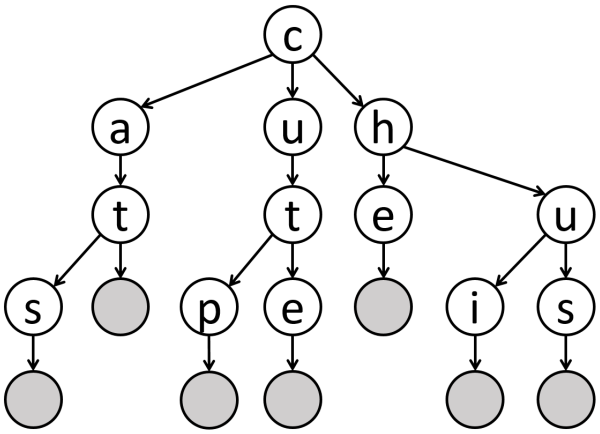


図 A.7: 三分探索木

A.0.6　グラフ

グラフはノードとエッジの集合からなるデータ構造である。グラフは各辺に向きが定められている有向グラフとエッジに向きがない無向グラフの２種類に分類される。各エッジには何かしらの属性（重みなど）が与えられることが多い。グラフの中でも閉路の無い連結グラフは木と呼ばれる。

グラフの実装には隣接行列を用いる方法と隣接リストを用いる方法の２通りがある。図 A.8 を隣接行列で表すと表 A.1 のようになる。隣接行列は２次元配列 g[i][j] で表され、g[i][j] は i 番目の頂点と j 番目の頂点の関係を表している。例えば、g[2][3]= 1 はノード D からノード C に向けてのエッジがあることを表している。

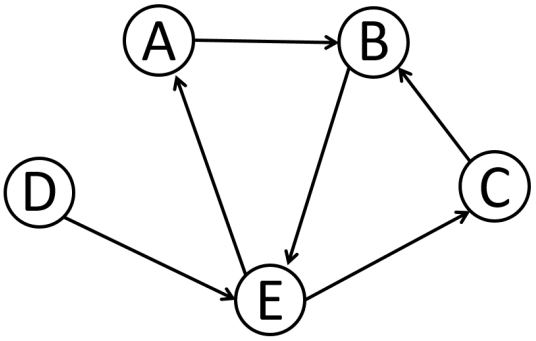


図 A.8: グラフ

i\j	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	1	0
C	0	1	0	0	0
D	1	0	1	0	0
E	0	0	0	1	0

表 A.1: 隣接行列での表現

頂点	隣接する頂点のリスト
A	B
B	E
C	B
D	E
E	A, C

表 A.2: 隣接リストでの表現

また、表 A.2 は図 A.8 を隣接リストで表現したものである。隣接リストは隣接行列よりもメモリ消費量を抑えることができる利点があるが、２点間にエッジがあるかどうかを調べるために $O(|E|)$ の時間がかかる。

A.0.7　B+木

B+木は木構造の１種であり、１つのノードが複数個の値を持つことで木の高さを抑えることができるという特徴がある。この特徴よりデータの検索を通常の二分探索木よりも効率よく行うことができる。各ノードが持つことのできるリンクの最大数はオーダーと呼ばれる。また、全てのデータは最下層の葉ノードに格納されており、葉ノード間はリンクで繋がれている。このため、順序付けされたデータに順々にアクセスする場合に検索の手間を省くことができる。図 A.9 はオーダー３の B+木を表している。

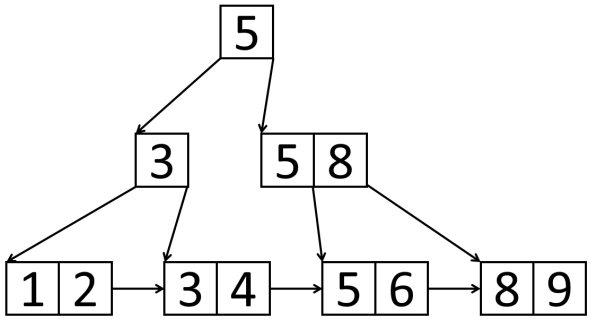


図 A.9: B+木

A.0.8　スキップリスト

スキップリスト（図 A.10）はリスト構造の１種であり、連結リストが多層構造になったデータ構造である。リストの各要素には階層の高さがランダムで定められており、自身よりも高さの低い要素を飛び越して次の要素へのリンクを持っている。そのため、リスト構造でありながら要素の挿入・削除・検索を平衡二分木と同等のパフォーマンスで行うことができる。

A.0.9　フィボナッチヒープ

フィボナッチヒープとはヒープ構造の１種である。ヒープとは「子の値が親の値より大きい又は等しい」という制約を持つ木構造のことである。

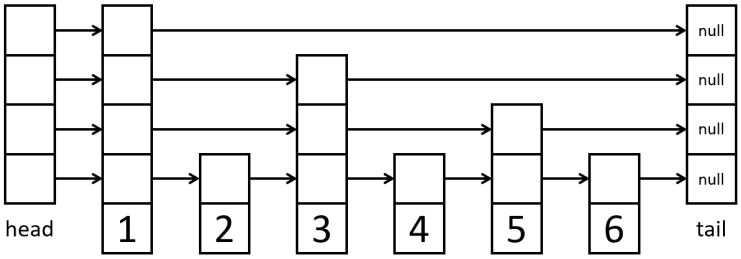


図 A.10: スキップリスト

フィボナッチヒープはフィボナッチ数の性質を使用することで通常のヒープよりも高速に操作を行うことができる。フィボナッチヒープはヒープ木の集合であり、各ヒープ木の根のノードは他の根へのリンクを持った双方向循環リストのような構造を持っている。また、各ノードの子の数 (=次数) はたかだか $O(\log n)$ である。図 A.11 は次数 0、1、3 の 3 つのヒープ木を持ったフィボナッチヒープの図である。

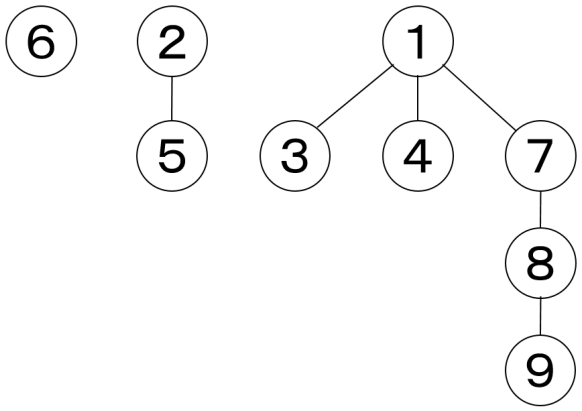


図 A.11: フィボナッチヒープ