木構造　探索

Lait-au-Cafe

平成 30 年 3 月 7 日

1　設計方針

本課題の目的を, アルゴリズムに対する理解を深め, 二分探索木と AVL 木の特徴を理解する事とし, それに伴い, 煩瑣なメモリ関係の操作を明示的に行う必要のない言語として Haskell を実装に用いた. また木は再帰的な構造であるため, 関数型言語を用いることによって木の操作に関する関数を簡潔に実装できることが期待される. さらに木構造を型として扱うことができる点も Haskell の良いところである. このような点を踏まえ, 実装は Haskell を用いて行うこととした.

1.1　乱数列の生成

Haskell では System.Random を import することで乱数を生成する諸関数を利用することができ, その中に randomR という関数がある. この関数は, 範囲と乱数生成器を与えると与えた範囲の乱数を返す関数である. これを利用してフィッシャー・イェーツのアルゴリズムを実装し, [1..n] の整列をシャッフルすることによって乱数列を生成する.

1.2　乱数列のテスト

1.1 節で作った, 1 から n までの整列をシャッフルした乱数列を与える関数 (genRandArray) の生成する乱数列の一様性を確かめるためのプログラムを作る. 一般的には, 可能な乱数列の数は n!だけ存在するため, この中から爆発しない程度の数の配列を選び, それらについて出現頻度を調べるのがよいと考えられるが, その場合, 適当な配列の組を無作為に選ぶ際に乱数を用いる必要がある. 基本的には標準ライブラリの関数であれば十分に検証されたものとして扱ってよいであろうが, 今回は標準ライブラリ関数も含めてテストをするものとし, テストの際に乱数を用いることは避ける. よって今回は配列の数が爆発しない程度の n に限定して全ての配列の出現頻度を調べ, 一様性を確認するものとする.

各配列の出現頻度を格納するデータ構造には図 1 に示すような木構造を用いる. 型宣言を次に示す.

```
data Results a =
          Value Int | Node a [Results a]
```

この木はそれぞれの節 (以後ノードと呼ぶ) に 1 から n までの番号いずれかと, 自分の子ノードのリストを持っており, 葉 (Value) に頻度値を格納するための領域を持っている. 図 1 は黄緑色の丸がノードを, 水色の箱が葉を表している. 配列が入力されると, 配列はノードに到達するたびに先頭要素を消費して先頭要素と同じ値を持つノードに分岐し, 葉にたどり着いたら葉に格納されている頻度値に 1 を加える. このような構造にすることによって配列によらず比較回数が一定となり, また再帰的な処理によって操作することができる.

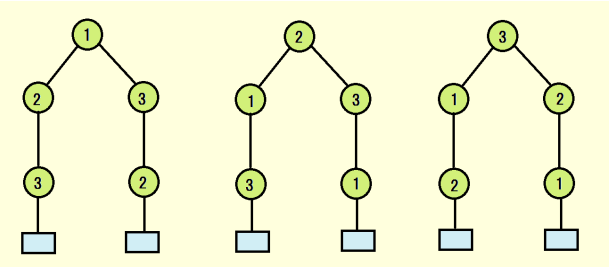

図 1: 配列の出現頻度を格納する木構造 (n=3 の場合)

1.3 二分探索木の高さの平均と分散の算出

次に示すような Tree 型を定義し, これを用いて二分探索木を表現する.

```
data Tree a =
        Null | Node a (Tree a) (Tree a)
```

二分探索木はノードによって構成され, 各ノードは型変数 a で表される任意の型のキー値と自分の子ノード二つを保持している. 子ノードが存在しない場合はノードの代わりに Null を持つ. このような木構造に対して, 新たにノードを一つ追加する関数 (addNode) を再帰的処理を用いて実装し, さらにこれを繰り返し用いることによって与えられた配列に対応する木を生成する関数 (createTree) を実装する. 同様にして木の高さを求める関数 (getTreeHeight) も再帰的処理で実装する. 最後にこれらの個別に作った関数をまとめ, 1.1 節で作った関数 (genRandArray) を用いて乱数列を発生させ, それに対応する木を生成し, 木の高さを求め, 結果を配列に格納する関数 (testTree) を作り, これを main 関数内で呼び出し, 結果を配列として受け取り, 配列の要素の平均と分散を求める.

また, 正しく木が生成されていることを確認するために, Mathematica を用いて木を可視化する. このために, 生成した木を Mathematica が解釈できる式に変換する関数 (encodeTree) を定義し, さらにこれを Mathematica で木を表示する式と組み合わせて.dat ファイルに書き込む関数 (exportTree) を作る. このように木を表示する式も同時に書き込むことで, Mathematica 側での処理を最小限に抑えることができる. Mathematica 側では.dat ファイルをテキスト形式で読み込み, それを式形式に変換する事によって生成された木を見ることができる.

1.4 AVL 木の高さの平均と分散の算出

基本的な設計方針は 1.3 節と同じであるが, AVL 木の場合は挿入操作の後にバランシングが必要であるため, このための関数を用意する. 一度にすべて実装すると処理が煩雑になるので, まずは与えられたノード部分を時計回りまたは反時計回りに回転する関数 (rotateTree) を作る. 次に各ノードの子ノードをルートとする部分木と, 孫ノードをルートとする部分木の高さを比較し, その関係に応じて上で定義した関数 (rotateTree) を用いて木の回転を行ってバランシングする関数 (balanceTree) を作り, これをノードを一つ追加するたびに必ず呼ぶようにする.

2 ソースコード解説

2.1 動作環境

以下の環境において動作を確認している.

- OS: Windows10, CentOS 6.7
- GHC バージョン 8.0.1
  ソースコートを実行可能ファイルへコンパイルするのに使用.
- GHCi バージョン 8.0.1
  関数の確認, 利用に使用.
- Mathematica バージョン 10.3
  生成した木の可視化に使用.

2.2 内容物

- TreeUtils.hs
  必要な関数をまとめたサブプログラム. 他のファイルから import して使う.
- ex1.hs
  整数 n を与えたとき, 1 から n までの自然数のランダムな列を出力するプログラム.
- test_ex1.hs
  ex1.hs をテストするためのプログラム. 整数 n, k を与えたとき, 1 から n までの自然数のランダムな列を k 組生成し, 各乱数列が何回生成されたかを出力するプログラム. n は 1 以上 5 以下の整数である必要がある.
- ex2.hs
  整数 n を与えたとき, n 個のノードを持つ二分探索木をランダムに n 個生成し, それらの高さの平均と分散を出力するプログラム.

- ex3.hs
  整数 n を与えたとき, n 個のノードを持つ AVL 木をランダムに n 個生成し, それらの高さの平均と分散を出力するプログラム.

2.3 実行方法

ソースファイルのあるディレクトリでコマンドプロンプトを開き, コマンドライン上で

```
$ ghc ex1.hs
```

を実行することで実行ファイル ex1.exe(Linux 環境では ex1) が得られる. test_ex1.exe, ex2.exe. ex3.exe も同様にして得られる. これらはそれぞれ,

```
$ ex1.exe 配列の長さ
$ test_ex1.exe 配列の長さ 試行回数
$ ex2.exe ノードの数
$ ex3.exe ノードの数
```

のように実行できる. Linux 環境では ex1.exe を./ex1 のように変更する.
実行例を以下に示す.

```
$ ex1.exe 10
[5,7,9,8,2,4,10,3,1,6]
$ test_ex1.exe 3 1000
[([1,2,3],178),([1,3,2],167),([2,1,3],
173),([2,3,1],151),([3,1,2],169),([3,2
,1],162)]
$ ex2.exe 200
Average:15.91
Variance:3.0919037
$ ex3.exe 150
Average:8.746667
Variance:0.18915558
```

2.4 関数解説

TreeUtils.hs に定義されている主要な関数について簡単に解説する. 詳しくは添付のソースコードを参照されたい.

genRandArray
: 型: Int -> IO [Int]
処理: 与えられた長さの乱数列を返す.
[1, 2, ..., n] をフィッシャー・イェーツのシャッフルでシャッフルする事で乱数列を得ている.

testTree
: 型: Int -> Int -> IO [Int]
処理: 第 1 引数で与えられたノード数の二分探索木を第 2 引数で与えられた数だけランダムに生成して, それぞれの高さを配列として返す.

testAVLTree
: 型: Int -> Int -> IO [Int]
処理: 第 1 引数で与えられたノード数の AVL 木を第 2 引数で与えられた数だけランダムに生成して, それぞれの高さを配列として返す.

createTree
: 型: Ord a => [a] -> Tree a
処理: 第 1 引数で与えられたリストから二分探索木を生成して返す.

createAVLTree
: 型: Ord a => [a] -> Tree a
処理: 第 1 引数で与えられたリストから AVL 木を生成して返す.

addNode
: 型: Ord a => Tree a -> a -> Tree a
処理: 第一引数で与えられた木に第二引数で与えられたキー値をノードとして追加して返す.

balanceTree
: 型: Tree a -> Tree a
処理: 引数で与えられた木をバランシングして返す. 木にノードを一つ追加するたびに呼ぶ必要がある.

exportTree
: 型: (Show a, Ord a) => Tree a -> IO()
処理: 第 1 引数で与えられた二分探索木を Mathematica が解釈できる形式に変換してカレントディレクトリに graph.dat として出力する. Mathematica を起動して graph.dat と同じディレクトリにノートブックを作成し,

```
ToExpression[Import[NotebookDirectory[
    ] <> "graph.dat", "Text"]]
```

を実行することで二分探索木を可視化できる.

average

型: [Int] -> Float

処理: 引数で与えられた配列の要素の平均値を返す.

variance

型: [Int] -> Float

処理: 引数で与えられた配列の要素の分散を返す.

3 出力結果

3.1 実行環境

実行環境を表 1 に示す.

表 1: 実行環境

OS	Windows 10 Home
CPU	Intel Core i5-5200U
RAM	8.00GB

3.2 乱数列の生成

乱数列を生成するプログラム ex1.hs に関しては, 内部的には TreeUtils.hs 内に定義されている genRandArray を実行しているだけなので, この関数を検証するためのプログラム test_ex1.hs を用意し, これによって検証を行った. 乱数列の長さを 3, 試行回数を 100 回, 1,000 回, 10,000 回, 100,000 回, 1,000,000 回として実行し, それぞれについて各配列がどれくらいの確率で生成されたかを図にまとめたものを図 2 に, 各配列の出現確率の分散を図にまとめたものを図 3 に示す.

3.3 二分探索木の高さの平均と分散の算出

ex2, ex3 のそれぞれについてノード数を 100, 200, ..., 800 として平均と分散を算出し, 結果を比較したものを図 4, 5 に示す.

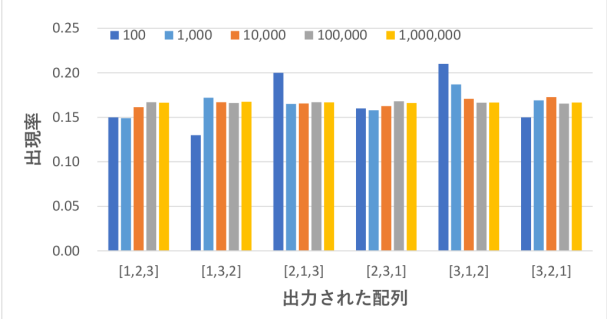

図 2: 試行回数と出現確率の関係

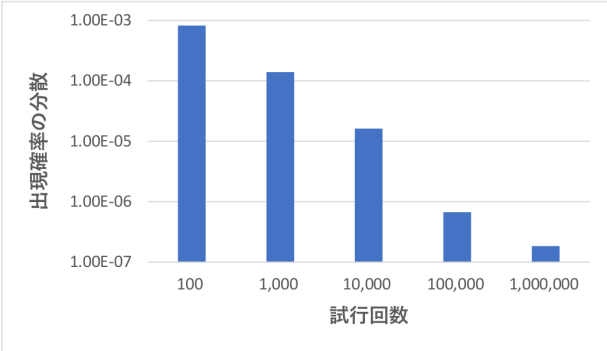

図 3: 試行回数と出現確率の分散の関係

4 結果の分析

4.1 乱数列の生成

図 2 を見ると, 試行回数が 100 回または 1,000 回の時にはそれより大きい試行回数に比べてばらつきが大きいが, 全体を見るとあきらかに生起しやすい組み合わせや明らかに生起しにくい組み合わせは存在せず, 十分に均一であることがわかる. 試行回数 100 回, 1,000 回の時点でまだばらつきが目でわかる程度に大きいのは, 発生させた乱数の周期が十分に大きいことが原因ではないかと考えられる. さらに図 3 から, 生起確率の分散は試行回数の増加に従って線形的に減少することがわかる. ただし, 試行回数が 1,000,000 回の時には線形から少しずれているため, これ以上試行回数を増やした場合には線形的ではなく 0 に漸近するようにして分散が減少するのではないかと考えられる. 結論として, Haskell の標準ライブラリの乱数生成関数は十分大きい周期を持った一様な乱数を発生させ, そのよう

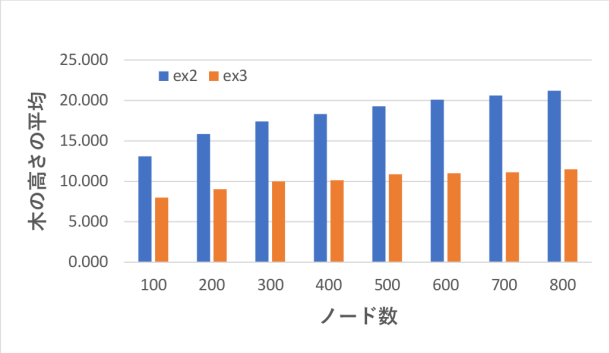

図 4: 二分探索木と AVL 木の高さの平均の比較

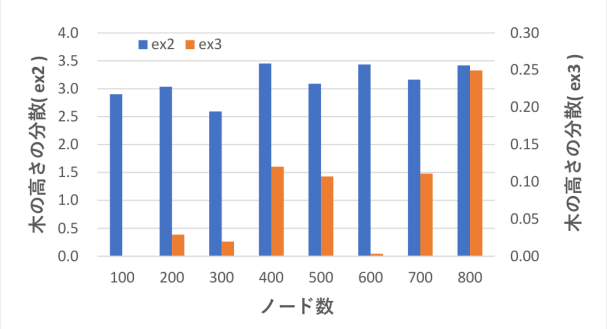

図 5: 二分探索木と AVL 木の高さの分散の比較

な乱数を用いてフィッシャー・イェーツのアルゴリズムによってシャッフルを行うことで, 一様な乱数列を生成させることが可能であるといえる.

4.2 二分探索木の高さの平均と分散の算出

図 4 から, AVL 木の高さの平均は, ノード数が 800 の時では二分探索木の 0.5 倍程度になり, ノード数の増加によってさらに差は広がると予想される. また通常の二分探索木と AVL 木のどちらの場合も, ノードの増加に対する木の高さの平均の増加は次第に小さくなり, 最終的にはほとんど変化しなくなることが予想される. 高さ n の二分探索木を構成する最小のノード数は n, 最大のノード数は $(2^n - 1)$ である. また, 高さ n の AVL 木を構成する最小のノード数を a_n とすると,

$$
\begin{aligned}
a_n &= a_{n-1} + a_{n-2} + 1 \\
&= \frac{1}{2}\left\{-2 + \frac{3}{\sqrt{5}}\left(\varphi^n - (-\varphi)^{-n}\right) + \left(\varphi^n + (-\varphi)^{-n}\right)\right\}
\end{aligned}
$$

が成り立つ. ただし φ は黄金比である. 最大のノード数は AVL 木に関しても二分探索木と同じ $(2^n - 1)$ である. 木の高さとノード数に関するこれらの理論的限界と ex2, ex3 の結果を比較したものを図 6 に示す.

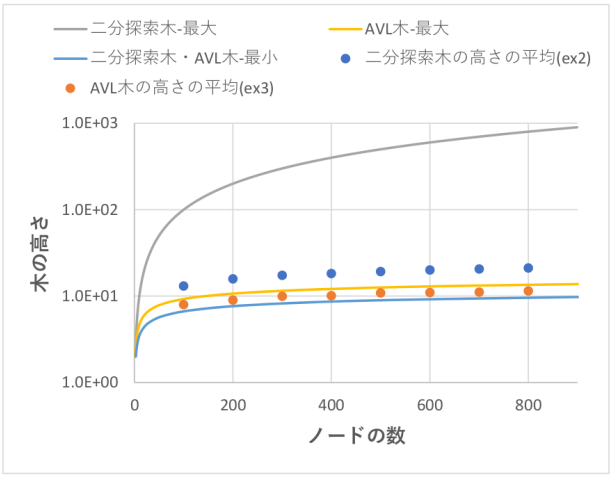

図 6: ある数のノードで構成される二分探索木/AVL 木の高さの最大/最小値と, ランダムに生成した二分探索木/AVL 木の高さの平均の関係

図 6 から, ex2, ex3 の結果は共に理論的限界内に収まっていることがわかる. さらに二分探索木の場合は同じノード数で構成できる木の高さの範囲が AVL 木に比べてはるかに大きいが, 実際に生成される木の高さはその範囲の中でも最小の高さに大きく偏っており, AVL 木の高さの理論的最大値よりも小さくなることはないものの, 自由度の割には木の高さが大きくならないことがわかる. これは, 高さが最大になる木を形成するキー値の並びが可能なキー値の並びの全体に比べてはるかに少ないことが原因であると考えられる. ノードの増加に対する木の高さの平均の増加は次第に小さくなるのも, ノード数の増加に対する可能なキー値の並びの数の増加に対して, 高さを最大にするようなキー値の並びの数の増加が小さいためであると考えられる.

次に図 5 で得られた木の高さの分散について考える. 図 6 からもわかる通り, 通常の二分探索木のほうが AVL 木に比べて取りえる高さの範囲が遥かに広いため, 分散の値は AVL 木の 10 倍程度になっているのがわかる. しかし図 6 よりノード数の増加によってとることのできる木の高さの範囲は単調増加しているにも関わらず, 分散の値は二分探索木も AVL 木も平均の

ように単調増加とはならず, 減少と増加を繰り返しながら大きくなることがわかる.

結論として, 二分探索木を実装する際に AVL 木を用いると, ノード数が多いほど効果的に木の高さを抑制することができ, また生成される木の高さの分散も小さくなるため, データに対して安定な構造を保証できる.

5 完成度の評価

今回は設計方針でも述べたように, アルゴリズムの理解や二分探索木, AVL 木の特徴の理解を目的としており, 実用性は重視しなかった. Haskell は正格評価ではなく遅延評価を行う言語であり, これによって計算の削減が見込める場合もあるが, 逆にこれによって冗長な計算をしてしまったり, 必要以上にメモリを消費してしまう場合もある. Haskell は正格評価を行う関数も用意してあるため, 本来であれば, 一通り実装を終えた後にボトルネックとなっている部分を探し, 遅延評価が原因であれば正格評価に置き換えるといった作業が必要である. 図 7, 8 に GHC のプロファイル機能を用いて ex2, ex3 をプロファイルした結果を示す. これらのプロファイル結果を見ると, 特に ex3 は大量にメモリを消費しており, 最大で 1MB 程度のメモリを使用している. 遅延評価は評価を遅らせる分処理を抱え込み, メモリの消費量が大きくなる.

また今回のコーディングでは単に再帰処理や型を意識したのみであるが, 本来 Haskell は数学的概念と密接に関わり合っており, それらの概念を利用した抽象的な構造を持つことが可能である. 抽象的な構造として扱うことによって, コードが簡潔になり, 保守性の向上が期待できる. 今回はそういった概念を用いずに実装しているため, コードがやや冗長になっており, 改善の余地がある.

参考文献

[1] Miran Lipovača. すごい Haskell たのしく学ぼう. 田中英行, 村主崇行. 東京, 株式会社オーム社, 2012, 391p

[2] Graham Hutton. プログラミング Haskell. 山本和彦. 東京, 株式会社オーム社, 2009, 217p

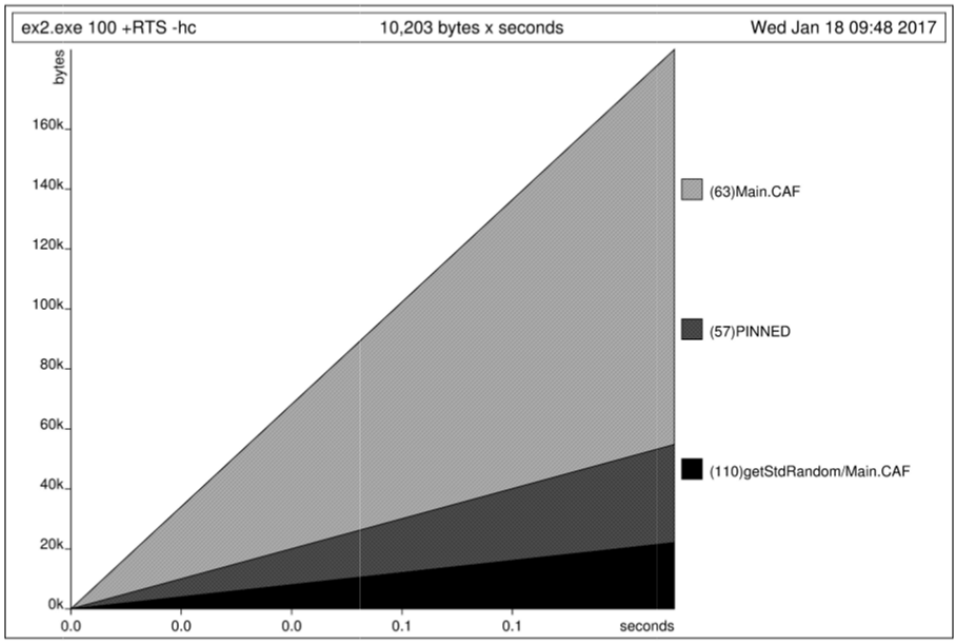

図 7: プロファイル結果 (ex2)

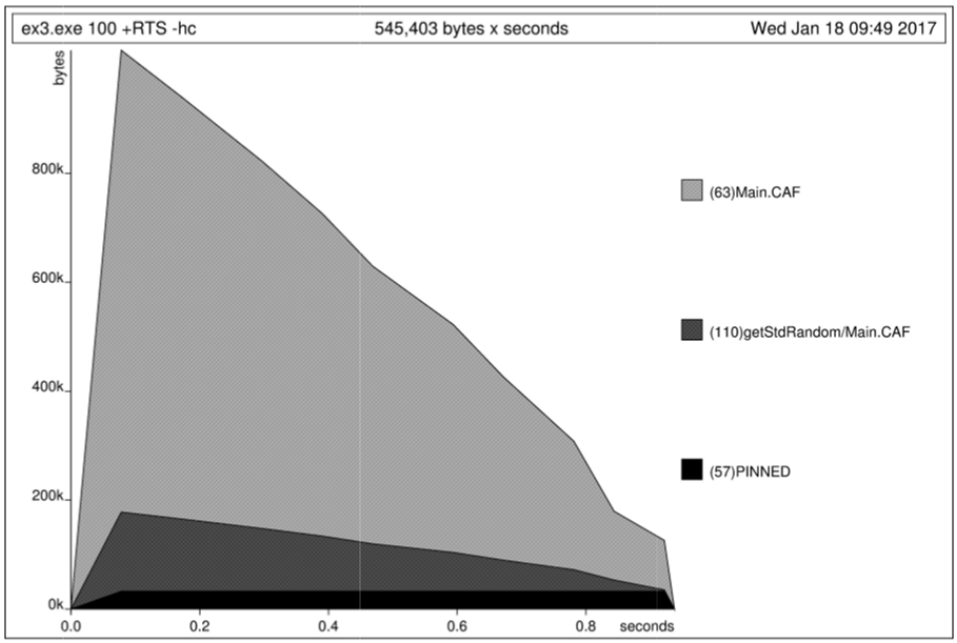

図 8: プロファイル結果 (ex3)