62304644 片山さくら

2024 年 12 月 24 日

1　設計方針・プログラム

1.1　実行環境

- OS: Windows 11
- 開発環境: Visual Studio 2019 v142
- ビルドツール: MSbuild 16.11.2.50704
- コンパイラ: MSVC 19.29.30157 for x86
- C 言語標準: C17

1.2　使用ライブラリ

- freeglut,glfw.3.4.0,glm.1.0.1：グラフィック用 (NuGet パッケージを使用)

1.3　プログラムの構成・概要

ファイル構成

- main.c：メイン関数
- binary_tree.c：二分探索木の生成・ノードの追加削除・探索などの関数などを定義
- tree_analysis.c：二分探索木の集合に対し、高さの平均・分散を算出する関数などを定義
- tree_graphic.c：二分木の描画関数を定義
- simulation_graphic.c：描画ループの実装
- utility.c：順列生成や階乗計算などの補助関数を定義
- position.c：座標を扱いやするするための関数を実装
- list.c：簡易的なリストの実装

- binary_tree.h：二分探索木の構造体定義や関数のプロトタイプ宣言・Node 構造体の定義
- tree_analysis.h：二分探索木の集合に対する分析関数のプロトタイプ宣言
- tree_graphic.h：二分木の描画関数のプロトタイプ宣言・NodeGrahic 構造体の定義
- simulation_graphic.h：描画ループのプロトタイプ宣言
- utility.h：補助関数のプロトタイプ宣言
- position.h：座標を扱うための関数のプロトタイプ宣言・XYi 構造体の定義

- list.h：簡易的なリストのプロトタイプ宣言・List 構造体の定義

2　①与えられた整数 n まで自然数で構成される順列の生成

Listing 1　utility.c

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include"utility.h"
4
5 int max_(int a,int b){
6         return a>b?a:b;
7 }
8
9 int min_(int a,int b){
10        return a<b?a:b;
11 }
12
13 int factorial(int n){
14        if (n<=1) return 1;
15        return n*factorial(n-1);
16 }
17
18 int permutation_num(int* num,int n){
19        int ind = 0;
20        for(int i = 0;i < n;i++){
21                ind *= (n-i);
22                for(int j = i+1;j<n;j++){
23                        if(num[i]>num[j]) ind++;
24                }
25        }
26        return ind;
27 }
28
29 void clear(int *array,int n){
30        for(int i = 0;i<n;i++) array[i] = 0;
31 }
32
33 static void calculate_permutation(int** list,int* nums,int* seen,int seed,int n,int depth){
34        seen[seed] = 1;
35        nums[depth] = seed;
36        for(int i = 1;i <= n;i++){
37                if(seen[i] == 0){
38                        calculate_permutation(list,nums,seen,i,n,depth+1);
39                        seen[i] = 0;
40                }
41        }
42        if(depth == n-1) for(int i = 0;i<n;i++) list[permutation_num(nums,n)][i] = nums[i];
43
44 }
```

```
45
46 void set_permutation_list(int** list, int n) {
47         int fact = factorial(n);
48
49         for (int i = 1; i <= n; i++) {
50                 int* seen = (int*)calloc(sizeof(int), (n + 1));
51                 int* nums = (int*)calloc(sizeof(int), (n + 1));
52                 calculate_permutation(list, nums, seen, i, n, 0);
53                 free(seen);
54                 free(nums);
55         }
56 }
57
58 void permutation_test() {
59         int n;
60         int fact;
61         int** list;
62         printf("input a number");
63         scanf("%d", &n);
64         fact = factorial(n);
65         list = (int**)malloc(sizeof(int*) * fact);
66         for (int i = 0; i < fact; i++) list[i] = (int*)malloc(sizeof(int) * n);
67
68         set_permutation_list(list, n);
69
70         for (int i = 0; i < fact; i++) {
71                 for (int j = 0; j < n; j++) {
72                         printf("%d ",list[i][j]);
73                 }
74                 putchar('\n');
75         }
76         putchar('\n');
77
78         for (int i = 0; i < fact; i++) free(list[i]);
79         free(list);
80 }
```

utility.c において重要な関数は以下のとおりである。

- calculate_permutation： 与えられた整数 n までの順列のうち seed で始まる順列を全て list に格納する関数
- permutation_num：与えられた順列に対応する番号を返す関数
- set_permutation_list：list に対し整数 n までの順列を全て格納する関数

set_permutation_list 関数が calculate_permutation 関数を呼び出すことで、与えられた整数 n までの順列を全て list に格納する。permutation_num 関数は calculate_permutation 関数内で順列を格納するインデックスを算出するのに使われている。permutation_num 関数により 1,2,3,4 は 0 へ、4,3,2,1 は 23 へと変換される。permutation_test 関数を実行することにより動作確認可能。

permutation_test の実行例

1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321

3 ② 生成した順列の順に数を入力したときに生成される相異なる二分探索木の列挙

Listing 2 binary_tree.c

```
1 #include"binary_tree.h"
2 #include"list.h"
3 #include"utility.h"
4
5 //------------ tree ----------------------
6 Tree create_tree(int key){
7     Tree new_ = (struct Node*)malloc(sizeof(struct Node));
8         if (new_ == NULL) return NULL;
9
10    new_->key = key;
11    new_->left = NULL;
```

```
12      new_->right = NULL;
13          new_->pointer = NULL;
14
15      return new_;
16  }
17
18  void delete_tree(Tree node){
19      if(node != NULL){
20          delete_tree(node->left);
21          delete_tree(node->right);
22          free(node);
23      }
24  }
25
26  int count_children(Tree root){
27          if (root == NULL) return 0;
28          else return count_children(root->left) + count_children(root->right) + 1;
29  }
30
31  int calculate_hight(Tree root){
32          if(root == NULL) return 0;
33          return max_(calculate_hight(root->left),calculate_hight(root->right))+1;
34  }
35
36  int is_AVL_tree(Tree root) {
37          int left_hight = calculate_hight(root->left);
38          int right_hight = calculate_hight(root->right);
39          int diff = left_hight - right_hight;
40          if ((-1 <= diff) && (diff <= 1)) {
41                  int left = 1;
42                  int right = 1;
43                  if (root->left != NULL) left = is_AVL_tree(root->left);
44                  if (root->right != NULL) right = is_AVL_tree(root->right);
45                  if ((left == 1) && (right == 1)) return 1;
46                  else return 0;
47          }
48          else return 0;
49  }
50
51  int is_complete_binary_tree(Tree root) {
52          int left_children_num = count_children(root->left);
53          int right_children_num = count_children(root->right);
54          int diff = left_children_num - right_children_num;
55          if ((-1 <= diff) && (diff <= 1)) {
56                  int left = 1;
57                  int right = 1;
58                  if(root->left != NULL) left = is_complete_binary_tree(root->left);
59                  if(root->right != NULL) right = is_complete_binary_tree(root->right);
60                  if ((left == 1) && (right == 1)) return 1;
61                  else return 0;
```

```
62          }
63          else return 0;
64 }
65
66 void insert_node(Tree* root,Tree node){
67      if(*root == NULL) {
68                  *root = node;
69                  return;
70          }
71      if(node->key <= (*root)->key) insert_node(&((*root)->left),node);
72      else insert_node(&((*root)->right),node);
73 }
74
75 static Tree extract_max(Tree root){
76          if(root->right == NULL) return root;
77          else return extract_max(root->right);
78 }
79
80 static Tree extract_min(Tree root){
81          if(root->left == NULL) return root;
82          else return extract_min(root->left);
83 }
84
85 void delete_node_from_tree(Tree* root,int key){
86      if(*root == NULL) return;
87          if((*root)->key < key) delete_node_from_tree(&((*root)->left),key);
88          else if((*root)->key > key) delete_node_from_tree(&(*root)->left,key);
89      else{
90                  if((*root)->left == NULL){
91                          *root = (*root)->right;
92                          free(*root);
93                          root = NULL;
94                  }
95                  else{
96                          Tree left_tree_max = extract_max((*root)->left);
97                          Tree tmp = left_tree_max->left;
98                          left_tree_max->left = (*root)->left;
99                          left_tree_max->right = (*root)->right;
100                         free(*root);
101                         *root = left_tree_max;
102                         ((*root)->left)->right = tmp;
103                 }
104         }
105 }
106
107 void search_pre(Tree root){
108         if (root == NULL) return;
109         printf("key:%d\n",root->key);
110         search_pre(root->left);
111         search_pre(root->right);
```

```
112 }
113
114 void search_pre_with_depth(Tree root,int depth){
115         if (root == NULL) return;
116         printf("key:%d␣d:%d\n",root->key,depth);
117         search_pre_with_depth(root->left,depth+1);
118         search_pre_with_depth(root->right,depth+1);
119 }
120
121 void search_middle(Tree root){
122         if (root == NULL) return;
123         search_middle(root->left);
124         printf("key:%d\n",root->key);
125         search_middle(root->right);
126 }
127
128 void search_post(Tree root){
129         if (root == NULL) return;
130         search_post(root->left);
131         search_post(root->right);
132         printf("key:%d\n",root->key);
133 }
134
135
136 void get_key_record(Tree root,List* list){
137         if (root == NULL) return;
138         push_back_list(list,root->key);
139         get_key_record(root->left,list);
140         get_key_record(root->right,list);
141 }
142
143 Tree* get_different_trees(int n) {
144         int fact = factorial(n);
145         int different_tree_num = 0;
146
147         Tree* result = (Tree*)malloc(sizeof(Tree) * (fact + 1));
148         if (result == NULL) return 0;
149         for (int i = 0; i < (fact + 1); i++) result[i] = NULL;
150
151         int **plist = (int**)malloc(sizeof(int*) * fact);
152         if (plist == NULL) {
153                 free(result);
154                 return 0;
155         }
156
157         for (int i = 0; i < fact; i++) plist[i] = (int*)malloc(sizeof(int) * n);
158
159         int* is_exist = (int*)calloc(fact,sizeof(int));
160         if (is_exist == NULL) {
161                 free(result);
```

```
162             for (int i = 0; i < fact; i++) free(plist[i]);
163             free(plist);
164             return 0;
165     }
166
167     set_permutation_list(plist, n);
168
169     printf("<%d>\n", fact);
170     /*for (int i = 0; i < fact; i++) {
171             for (int j = 0; j < n;j++) {
172                     printf("%d ",plist[i][j]);
173             }
174             putchar('\n');
175     }
176     putchar('\n');*/
177
178     for (int i = 0; i < fact;i++) {
179             Tree current = create_tree(plist[i][0]);
180             Tree* current_p = &current;
181             int current_id = 0;
182             List* keyrecord = create_list(n);
183
184
185             for (int j = 1; j < n; j++) insert_node(current_p, create_tree(plist[i][j]));
186
187             get_key_record(current, keyrecord);
188             current_id = permutation_num(keyrecord->list, keyrecord->len);
189
190             delete_list(keyrecord);
191
192             if (is_exist[current_id]) {
193                     free(current);
194                     continue;
195             }
196             else {
197                     is_exist[current_id] = 1;
198                     result[different_tree_num] = current;
199                     different_tree_num++;
200             }
201     }
202
203     for (int i = 0; i < fact; i++) free(plist[i]);
204     free(plist);
205     free(is_exist);
206
207     return result;
208 }
```

---

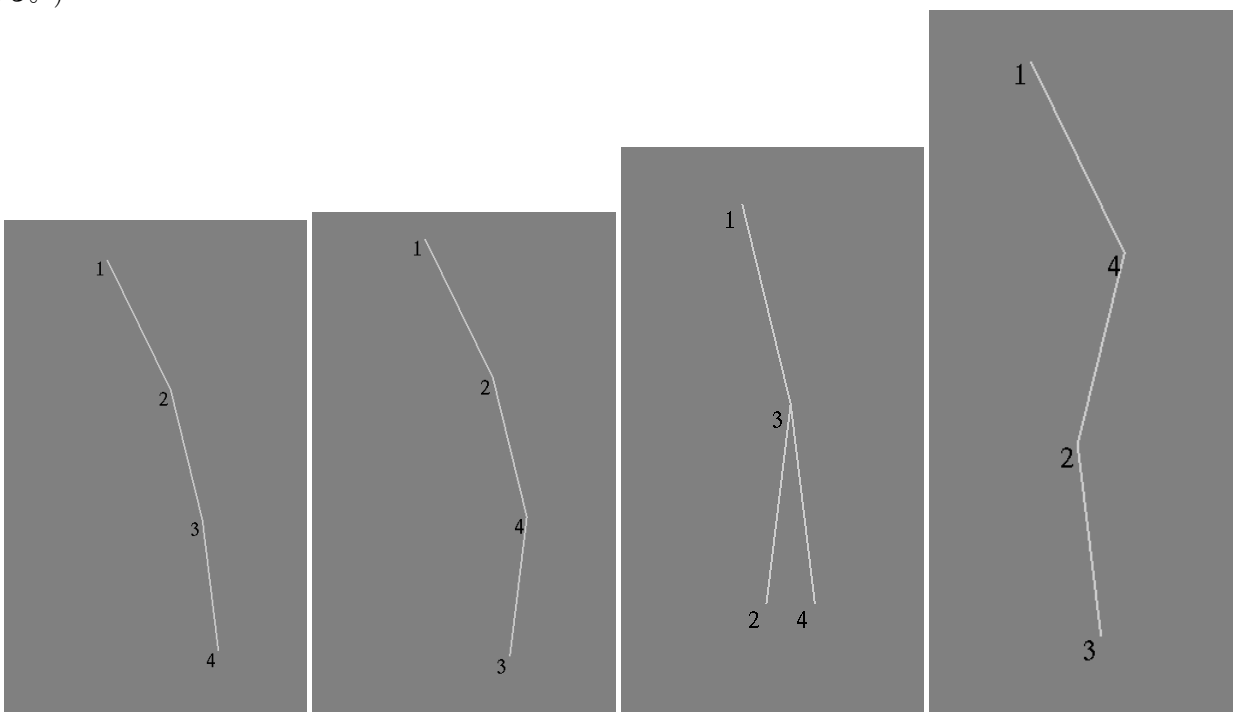
binary_tree.c において重要な関数は以下のとおりである。

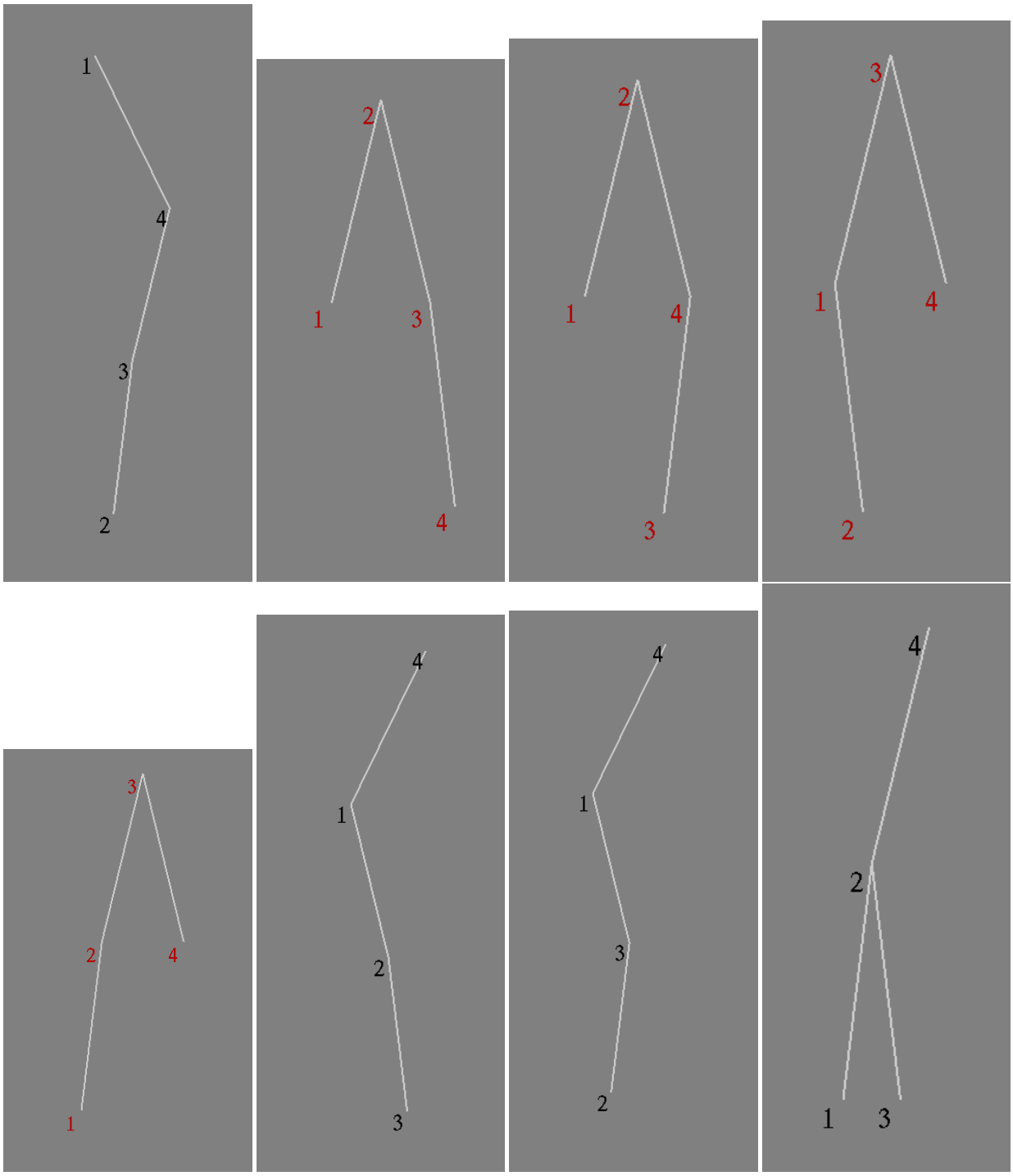
- get_key_record : あるノードを根とした部分木を前順に探索しキーの値を配列に格納する関数

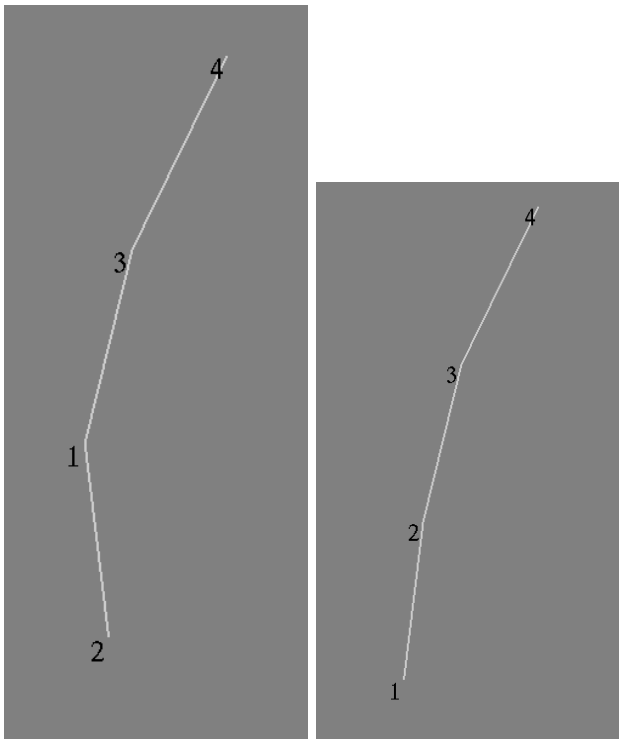
- get_different_trees：整数 n までの順列を順に二分探索木に入力し、生成したすべての相異なる二分探索木の根へのポインタをリストに格納する関数

とある順列を二分探索木に入力する際、同様の配置を取る二分探索木が生成することがある。そのため構造的な重複を避けるために、二分木の構造とノード前順で探索し順に記録して出来た順列が一対一に対応することを利用し、get_different_trees 関数内では utility.c の permutation_num 関数を用いて二分木に ID を付け重複したものは記録しないようにしている。

実行例 n=4 の場合 (なお、完全部分木の場合文字は赤、AVL 木の場合文字は青、それ以外は黒で表示される。)



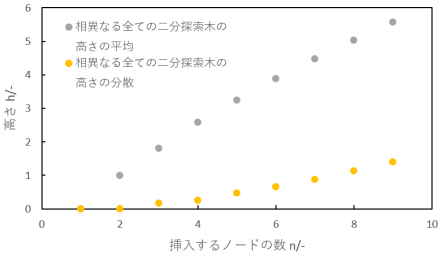

1
4
3
2
2
1
3
4
2
1
4
3
3
1
4
2



上の結果は SQ7 の結果と一致する。

n が 1 から 9 までの場合についてそれぞれの木の高さの平均と分散を算出した。

ノード数	all:平均	all:分散
1	0	0
2	1.000	0.000
3	1.800	0.160
4	2.571	0.245
5	3.238	0.467
6	3.879	0.652
7	4.471	0.879
8	5.031	1.129
9	5.563	1.392



4 ③ 生成した二分探索木のうち AVL 木または完全二分木の集合に対し、高さの平均・分散を算出

AVL 木および完全二分木の選別は以下の関数を用いて行った。

- count_children：あるノードを根とした部分木の根を含むノード数を返す関数
- calculate_hight：あるノードを根とした部分木の根の高さ +1 の値を返す関数
- is_AVL_tree：あるノードを根とした部分木が AVL 木であるかを判定する関数

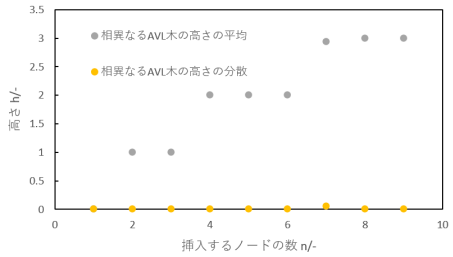
- is_complete_binary_tree： あるノードを根とした部分木が完全二分木であるかを判定する関数

is_AVL_tree 関数は各ノードに対し calculate_hight を実行し左右の部分木の高さの差が 1 以内に収まっているかどうかを調べている。同様に is_complete_binary_tree 関数は各ノードに対し count_children を実行し、左右の子ノード数の差が 1 以内に収まっているかどうかを調べている。

n が 1 から 9 までの場合についてそれぞれの木の高さの平均と分散を算出した。

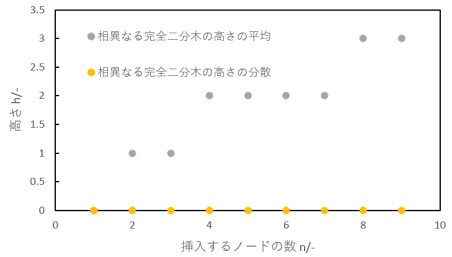
ACL 木の場合

ノード数	AVL:平均	AVL:分散
1	0	0
2	1	0
3	1	0
4	2	0
5	2	0
6	2	0
7	2.941	0.055
8	3	0
9	3	0

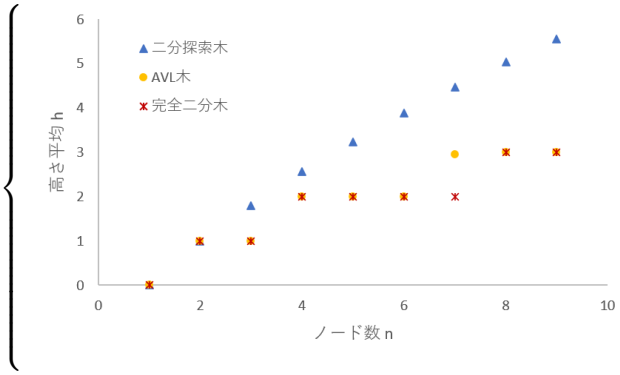


完全二分木の場合

ノード数	cmp:平均	cmp:分散
1	0	0
2	1	0
3	1	0
4	2	0
5	2	0
6	2	0
7	2	0
8	3	0
9	3	0



②と③の結果をまとめそれぞれの高さの平均について比較したグラフをいかに示す。



また、相異なる二分探索木の集合全体の要素数、AVL 木のみの要素数、完全二分木のみの要素数を以下の図にまとめた。なお、縦軸は対数軸である。

ノード数	all	AVL	cmp
1	1	1	1
2	2	2	2
3	5	1	1
4	14	4	4
5	42	6	4
6	132	4	4
7	429	17	1
8	1430	32	8
9	4862	44	16